Oracle8 Server アプリケーション開発者ガイド

リリース 8.0

1998 **年** 2 **月**

部品番号 A56822-1

ORACLE®

Enabling the Information Age™

Oracle8 Server アプリケーション開発者ガイド

部品番号 A56822-1

第 1 版：1998 年 2 月

原本名：Oracle8 Application Developer’s Guide, Release 8.0

原本部品番号：A58241-01

原本著者：Denis Raphaely

協力著者：Paul Lane, Lefty Leverenz, Richard Mateosian

協力者：Richard Allen, Neerja Bhatt, Steven Bobrowski, George Buzsaki, Sashi Chandrasekaran, Atif Chaudry, Greg Doherty, Dieter Gawlick, Gary Hallmark, Michael Hartstein, Chin Hong, Kenneth Jacobs, Hakan Jakobsson, Amit Jasuja, Robert Jenkins, Jr.,Jonathan Klein, Robert Kooi, Susan Kotsovolos, Vishu Krishnamurthy, Ramkumar Krishnan, Juan Loaiza, William Maimone, Andrew Mendelsohn, Mark Moore, Ravi Narayaran, Goran Olsson, Edward Peeler, Thomas Portfolio, Maria Pratt, Tuomas Pystynen, Mark Ramacher, Madhu Reddy, Hasan Rizvi, Premal Shah, Timothy Smith, Alvin To, Usha Sundaram, Scott Urman, Peter Vasterd, Joyo Wijaya

グラフィック・デザイナー：Valarie Moore

Copyright© 1997, 1998, Oracle Corporation. All rights reserved.

Printed In Japan

制限付権利の説明

プログラムの使用、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。

本書の情報は、予告なしに変更されることがあります。本書に問題を見つけたら、当社にコメントをお送りください。オラクル社は本書の無謬性を保証しません。

危険な用途への使用について

当社製品は、原子力、航空産業、大量輸送、又は医療の分野など、本質的に危険が伴うアプリケーションを用途として特に開発されておりません。当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いただきたく、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、当社および開発元である米国 Oracle Corporation（その関連会社も含みます。）は一切責任を負いかねます。

ORACLE は、Oracle Corporation の登録商標です。

本文中の他社の商品名は、それぞれ各社の商標または登録商標です。

目次

はじめに xxiii

このマニュアルについて xxiv
このマニュアルの対象読者 xxiv
機能範囲と可用性 xxiv
関連資料 xxiv
本書の構成 xxv
このマニュアルの表記規則 xxvii

1　アプリケーション開発者の情報源

情報源 1-2
特定のトピック 1-2
業務ルール 1-3
クライアント側のツール 1-3
3GL プログラムとの通信 1-3
データベース制約 1-3
データベース設計 1-4
データ型 1-4
デバッグ 1-4
エラー処理 1-4
ゲートウェイ 1-4
Oracle が提供するパッケージ 1-4
PL/SQL 1-5
スキーマ・オブジェクト 1-5
セキュリティ 1-5
SQL 文 1-5

ツール 1-5

2 アプリケーション開発者

必要性を評価する 2-2
データベースを設計する 2-2
アプリケーションを設計する 2-4
利用可能な機能を使用する 2-4
Oracle コール・インタフェースの使用 2-6
SQL を作成する 2-7
アプリケーションでセキュリティを施行する 2-7
アプリケーションをチューニングする 2-7
アプリケーションをメンテナンスする、更新する 2-8

3 SQL 文の処理

SQL 文の実行 3-2
FIPS フラグ付け 3-2
トランザクションを制御する 3-4
パフォーマンスを改善する 3-4
トランザクションをコミットする 3-5
トランザクションをロールバックする 3-5
トランザクションのセーブポイントを定義する 3-6
..... 3-7
ランザクション管理のために必要な権限 3-7
読取り専用トランザクション 3-8
カーソルの使用 3-8
カーソルを宣言しオープンする 3-9
カーソルを使用した文の再実行 3-9
カーソルをクローズする 3-9
カーソルの取消し 3-10
明示的なデータのロック 3-10
表ロックを明示的に獲得する 3-11
必要な権限 3-14
行ロックを明示的に獲得する 3-14
SERIALIZABLE パラメータと ROW_LOCKING パラメータ 3-15
デフォルト以外のロック・オプションのまとめ 3-15

ユーザー・ロックを作成する 3-17
DBMS_LOCK パッケージ 3-17
セキュリティ 3-18
DBMS_LOCK パッケージを作成する 3-18
ALLOCATE_UNIQUE プロシージャ 3-18
REQUEST 関数 3-19
CONVERT 関数 3-21
RELEASE 関数 3-23
SLEEP プロシージャ 3-24
ユーザー・ロックの例 3-24
ロックを表示して監視する 3-25
直列可能トランザクションを使用した同時実行性の制御 3-26
直列可能トランザクションの相互作用 3-29
分離レベルを設定する 3-29
参照整合性と直列可能トランザクション 3-30
READ COMMITTED 分離と SERIALIZABLE 分離 3-32
アプリケーションのヒント 3-35

4 スキーマ・オブジェクトの管理

表を管理する 4-3
表を設計する 4-3
表を作成する 4-3
表を変更する 4-8
表を削除する 4-9
ビューを管理する 4-10
ビューを作成する 4-10
ビューを置き換える 4-12
ビューを使用する 4-13
ビューを削除する 4-14
結合ビューを修正する 4-14
キー保存された表 4-17
結合ビューの DML 文の規則 4-17
UPDATABLE_COLUMNS ビューを使用する 4-20
外部結合 4-20
順序を管理する 4-22

順序を作成する 4-23
順序を変更する 4-23
順序を使用する 4-24
順序を削除する 4-27
シノニムを管理する 4-27
シノニムを作成する 4-28
シノニムを使用する 4-28
シノニムを削除する 4-28
索引を管理する 4-29
索引を作成する 4-32
索引を削除する 4-33
クラスタ、クラスタ化した表、クラスタ索引を管理する 4-33
クラスタを作成するためのガイドライン 4-34
パフォーマンスの検討 4-34
クラスタ、クラスタ化した表、クラスタ索引を作成する 4-35
クラスタに対して記憶領域を手動で割り当てる 4-36
クラスタ、クラスタ化した表、クラスタ索引を削除する 4-37
ハッシュ・クラスタとクラスタ化した表を管理する 4-38
ハッシュ・クラスタおよびクラスタ化した表を作成する 4-38
ハッシュ・クラスタ内の領域使用を制御する 4-38
ハッシュ・クラスタを削除する 4-39
どんな場合にハッシュを使用するか 4-39
スキーマ・オブジェクトの管理について 4-40
一回の操作で複数の表とビューを作成する 4-41
スキーマ・オブジェクトを命名する 4-41
SQL 文における参照オブジェクトの名前変換 4-42
スキーマ・オブジェクトを改名する 4-43
スキーマ・オブジェクトに関する情報をリストする 4-44

5 データ型の選択

Oracle ビルトイン・データ型 5-2
文字データ型を使用する 5-5
NUMBER データ型を使用する 5-7
DATE データ型を使用する 5-7
LONG データ型を使用する 5-9

RAW データ型と LONG RAW データ型を使用する 5-11
ROWID と ROWID データ型 5-12
Trusted Oracle MLSLABEL データ型 5-15
ANSI/ISO データ型および DB2 データ型、SQL/DS データ型 5-15
データ変換 5-17
ルール 1: 代入 5-17
ルール 2: 式の評価 5-18
Trusted Oracle のデータ変換 5-19

6 ラージ・オブジェクト (LOB)

LOB の概要 6-2
LOB とは何か ? 6-4
内部 LOB と外部 LOB(BFILE) 6-5
LOB と、LONG 型および LONG RAW 型との比較 6-6
LOB の操作に使用するパッケージ 6-6
LOB データ型 6-6
内部 LOB と外部 LOB を表に定義する 6-8
内部 LOB 用の表領域と記憶特性を取り決める 6-8
内部 LOB を初期化する (SQL DML) 6-13
外部 LOB にアクセスする (SQL DML) 6-14
BFILE セキュリティ 6-16
ディレクトリのカタログ・ビュー 6-17
DIRECTORY 使用のガイドライン 6-18
BFILE の最大オープン数 6-19
MTS モードにおける BFILE 6-19
プログラム終了後に BFILE をクローズする 6-19
LOB 値とロケータ 6-20
LOB ロケータの操作 6-21
大量の LOB データを効率よく読み書きする。 6-35
LOB をコピーする 6-35
LOB を削除する 6-36
オブジェクト・キャッシュの LOB 6-43
LOB バッファリング・サブシステム 6-43
最大のパフォーマンスを引き出すためのユーザー・ガイドライン 6-52
可変幅文字データの処理 6-53

LOB リファレンス 6-55
リファレンス概要 6-55
EMPTY_BLOB() 関数と EMPTY_CLOB() 関数 6-55
BFILENAME() 関数 6-56
LOB の操作に OCI を使用する 6-59
DBMS_LOB パッケージ 6-62
パッケージ・ルーチン 6-62
データ型 6-63
型定義 6-63
定数 6-63
DBMS_LOB 例外 6-64
DBMS_LOB セキュリティ 6-65
DBMS_LOB の一般的な使用上の注意 6-65
BFILE に特有の使用上の注意 6-66
DBMS_LOB.APPEND() プロシージャ 6-68
DBMS_LOB.COMPARE() 関数 6-69
DBMS_LOB.COPY() プロシージャ 6-72
DBMS_LOB.ERASE() プロシージャ 6-74
DBMS_LOB.FILECLOSE() プロシージャ 6-76
DBMS_LOB.FILECLOSEALL() プロシージャ 6-77
DBMS_LOB.FILEEXISTS() 関数 6-77
DBMS_LOB.FILEGETNAME() プロシージャ 6-79
DBMS_LOB.FILEISOPEN() 関数 6-80
DBMS_LOB.FILEOPEN() プロシージャ 6-81
DBMS_LOB.GETLENGTH() 関数 6-82
DBMS_LOB.INSTR() 関数 6-84
DBMS_LOB.LOADFROMFILE() プロシージャ 6-86
DBMS_LOB.READ() プロシージャ 6-88
DBMS_LOB.SUBSTR() 関数 6-91
DBMS_LOB.TRIM() プロシージャ 6-93
DBMS_LOB.WRITE() プロシージャ 6-94
LOB 制約 6-97

7 ユーザー定義データ型 – 拡張例

概要 7-2

発注書の例 7-2
エンティティおよび関連 7-3
第 1 部　リレーショナル・アプローチ 7-4
第 2 部　オブジェクト表を使用したオブジェクト・リレーショナル・アプローチ　7-7

8　オブジェクト・ビュー － 拡張例

概要 8-2
発注書の例 8-2
オブジェクト・ビューの定義 8-3
オブジェクト・ビューの更新 8-5
更新のサンプル 8-8
データの選択 8-8

9　データの整合性の保持

整合性制約の使用 9-2
整合性制約を使用する場合 9-2
整合性制約の利用 9-2
NOT NULL 整合性制約の使用 9-3
デフォルトの列値の設定 9-4
表の主キーの選択 9-5
UNIQUE キー整合性制約の使用 9-6
参照整合性制約の使用 9-7
NULL と外部キー 9-7
親表と子表の関連 9-9
複数の FOREIGN KEY 制約 9-10
同時実行性制御、索引、外部キー 9-10
分散データベース内の参照整合性 9-12
CHECK 整合性制約の使用 9-13
CHECK 制約の制限 9-13
..... 9-14
CHECK 制約の設計 9-14
複数の CHECK 制約 9-14
CHECK および NOT NULL 整合性制約 9-14
整合性制約の定義 9-15

CREATE TABLE コマンド 9-15
ALTER TABLE コマンド 9-16
必要な権限 9-16
整合性制約の命名 9-17
定義時の制約の使用可能、使用禁止 9-17
UNIQUE キーおよび PRIMARY KEY、FOREIGN KEY 9-17
整合性制約の使用可能、使用禁止 9-18
制約を使用可能、使用禁止にする理由 9-18
整合性制約違反 9-18
定義について 9-18
整合性制約の使用可能、使用禁止 9-19
キー整合性制約の使用可能、使用禁止 9-20
パラレル・ダイレクト・パスのロード後の制約の使用可能 9-20
例外のレポート 9-21
整合性制約の変更 9-22
整合性制約の削除 9-23
FOREIGN KEY 整合性制約の管理 9-23
FOREIGN KEY 整合性制約の定義 9-23
FOREIGN KEY 整合性制約を使用可能にする 9-25
整合性制約定義のリスト 9-25
例 9-25

10 プロシージャとパッケージの使用

PL/SQL プロシージャとパッケージ 10-2
無名ブロック 10-2
データベース・トリガー 10-4
ストアド・プロシージャとストアド関数 10-4
ストアド・プロシージャと関数の作成 10-8
ストアド・プロシージャおよび関数を変更する 10-10
外部プロシージャ 10-10
PL/SQL パッケージ 10-10
パッケージを作成する 10-12
パッケージ・オブジェクトを作成する 10-13
パッケージとパッケージ・オブジェクトを命名する 10-13
パッケージとプロシージャを削除する 10-13

パッケージの無効性とセッションの状態 10-14
リモート依存性 10-14
タイムスタンプ 10-15
署名 10-16
リモート依存性を制御する 10-22
依存性を管理するための提案 10-23
カーソル変数 10-24
カーソル変数を宣言してオープンする 10-24
カーソル変数の例 10-24
PL/SQL コードを隠す 10-27
エラー処理 10-27
例外と例外処理ルーチンを宣言する 10-29
未処理例外 10-30
分散問合せでエラーを処理する 10-31
リモート・プロシージャでエラーを処理する 10-31
コンパイル時エラー 10-32
デバッグ 10-33
ストアド・プロシージャの呼出し 10-34
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 10-34
Oracle Tools からプロシージャを対話形式でコールする 10-34
3GL アプリケーション内でプロシージャをコールする 10-35
プロシージャをコールするときの名前変換 10-36
プロシージャを実行するために必要な権限 10-36
プロシージャ引数の値を指定する 10-37
リモート・プロシージャをコールする 10-37
リモート・オブジェクトを参照する 10-38
プロシージャとパッケージのシノニム 10-39
SQL 式からストアド関数をコールする 10-40
PL/SQL 関数を使用する 10-40
構文 10-41
命名規則 10-41
基本的な要件を満たす 10-43
副次作用を制御する 10-43
オーバーロードする 10-48
逐次再使用可能 PL/SQL パッケージ 10-48

必要な権限 10-56
提供されるパッケージ 10-56
SQL 機能をサポートするパッケージ 10-56
その他の機能性をサポートするパッケージ 10-61
ストアド・プロシージャを記述する 10-65
DBMS_DESCRIBE パッケージ 10-65
セキュリティ 10-65
タイプ 10-65
エラー 10-65
DESCRIBE_PROCEDURE プロシージャ 10-66
プロシージャとパッケージに関する情報をリストする 10-73
DBMS_ROWID パッケージ 10-74
サマリー 10-75
例外 10-76
ROWID_CREATE 関数 10-77
ROWID_INFO プロシージャ 10-77
ROWID_TYPE 関数 10-78
ROWID_OBJECT 関数 10-78
ROWID_RELATIVE_FNO 関数 10-79
ROWID_BLOCK_NUMBER 関数 10-79
ROWID_ROW_NUMBER 関数 10-79
ROWID_TO_ABSOLUTE_FNO 関数 10-80
ROWID_TO_EXTENDED 関数 10-80
ROWID_TO_RESTRICTED 関数 10-82
ROWID_VERIFY 関数 10-82
UTL_HTTP パッケージ 10-82

11 アドバンスト・キューイング

Oracle アドバンスト・キューイングについて 11-2
概要 11-2
複雑なシステム 11-3
実現可能なソリューション : 同期通信と切断 / 遅延通信 11-7
Oracle アドバンスト・キューイング ― 機能 11-7
Oracle アドバンスト・キューイング ― 主要コンポーネント 11-11
キュー・エンティティ・モデル 11-13

基本キューイング 11-14
..... 11-14
基本キューイングの説明 11-14
AQ を使用したクライアントサーバー通信の例 11-16
同一メッセージの複数コンシューマ・デキュー 11-17
同一メッセージの複数コンシューマによるデキューの例 11-18
指定受信者による指定メッセージのデキューの例 11-20
AQ を使用したワークフローの実現の例 11-22
メッセージの伝播 11-23
メッセージの伝播の例 11-25
Oracle アドバンスト・キューイングの例 11-25
概要 11-25
ロールおよび権限の割当て 11-26
キュー表およびキューの作成 11-26
オブジェクト型メッセージのエンキューおよびデキュー 11-28
Pro*C/C++ を使ったオブジェクト型メッセージのエンキューおよびデキュー 11-29
OCI を使ったオブジェクト型メッセージのエンキューおよびデキュー 11-31
RAW 型メッセージのエンキューおよびデキュー 11-33
Pro*C/C++ **を使った** RAW **型メッセージのエンキューおよびデキュー** 11-34
OCI を使った RAW 型メッセージのエンキューおよびデキュー 11-36
優先順位によるメッセージのエンキューおよびデキュー 11-37
基準によるプリビュー後のメッセージのデキュー 11-39
遅延および時間切れによるメッセージのエンキューおよびデキュー 11-42
Pro*C/C++ を使った、相関およびメッセージ ID によるメッセージのエンキューおよびデキュー 11-43
OCI を使った、相関およびメッセージ ID によるメッセージのエンキューおよびデキュー 11-47
PL/SQL を使った複数コンシューマ・キューでのメッセージのエンキューおよびデキュー 11-49
OCI を使った複数コンシューマ・キューでのメッセージのエンキューおよびデキュー 11-52
複数コンシューマ・キューへのリモートのサブスクライバ / 受信者に対するメッセージのエンキューおよび伝播スケジューリング 11-56
伝播のスケジュール解除 11-58
Message Grouping を使ったエンキューおよびデキュー 11-58
AQ オブジェクトの削除 11-60
ロールおよび権限の取消し 11-61
Oracle アドバンスト・キューイング・リファレンス 11-61

リファレンスの概要 11-61
INIT.ORA パラメータ 11-61
データ構造 11-63
エージェント 11-64
メッセージ・プロパティ 11-65
キューオプション 11-67
操作インタフェース 11-69
操作インタフェースの列挙定数 11-74
管理インタフェース 11-74
管理インタフェースの列挙定数 11-88
データベース・オブジェクト 11-89
エラー・メッセージ 11-93
管理項目 11-94
パフォーマンス 11-94
可用性 11-94
拡張性 11-94
伝播の最適化 11-95
信頼性および回復可能性 11-95
Enterprise Manager サポート 11-95
キュー・データのインポートおよびエクスポート 11-95
トラブルシューティング 11-96
動的統計ビュー 11-98
デモの参照先 11-99
互換性およびアップグレード 11-99

12 PL/SQL 入出力

データベース・パイプ 12-2
サマリー 12-2
DBMS_PIPE パッケージを作成する 12-3
パブリック・パイプ 12-3
プライベート・パイプ 12-3
エラー 12-4
CREATE_PIPE 12-4
PACK_MESSAGE プロシージャ 12-6
SEND_MESSAGE 12-7

RECEIVE_MESSAGE 12-8
NEXT_ITEM_TYPE 12-10
UNPACK_MESSAGE プロシージャ 12-10
REMOVE_PIPE 12-11
パイプを管理する 12-11
パイプの内容を削除する 12-11
メッセージ・バッファのリセット 12-12
一意なセッション名の取得 12-12
例 1: デバッグ 12-12
例 2: システム・コマンドの実行 12-14
ストアド・プロシージャとトリガーからの出力 12-20
サマリー 12-21
DBMS_OUTPUT パッケージの作成 12-21
エラー 12-21
ENABLE プロシージャ 12-22
DISABLE プロシージャ 12-22
PUT および PUT_LINE プロシージャ 12-22
GET_LINE および GET_LINES プロシージャ 12-23
DBMS_OUTPUT パッケージの使用例 12-24
PL/SQL のファイル I/O 12-27
サマリー 12-27
セキュリティ 12-28
宣言された型 12-30
例外 12-30
FOPEN 12-31
IS_OPEN 12-32
FCLOSE 12-33
FCLOSE_ALL 12-34
GET_LINE 12-34
PUT 12-35
NEW_LINE 12-36
PUT_LINE 12-37
PUTF 12-37
FFLUSH 12-39

13 データベース・トリガーの使用

トリガーを設計する 13-2
トリガーを作成する 13-2
トリガー作成の前提条件 13-3
トリガーを命名する 13-3
BEFORE および AFTER オプション 13-3
INSTEAD OF オプション 13-4
トリガー文 13-6
FOR EACH ROW オプション 13-6
WHEN 句 13-7
トリガー本体 13-8
トリガーとリモート例外の処理 13-10
トリガー作成の制限 13-11
トリガー・ユーザーとは誰か 13-15
トリガーを作成するために必要な権限 13-15
参照スキーマ・オブジェクトに対する権限 13-16
トリガーがコンパイルされるとき 13-16
依存性 13-16
トリガーを再コンパイルする 13-17
移行の問題 13-17
トリガーをデバッグする 13-17
トリガーを修正する 13-17
トリガーを使用可能、使用禁止にする 13-18
トリガーを使用禁止にする 13-18
トリガーを使用可能にする 13-18
トリガーを使用可能、使用禁止にするために必要な権限 13-19
トリガーに関する情報をリストする 13-19
トリガー・アプリケーションの例 13-20
トリガーで監査する 13-20
整合性制約とトリガー 13-24
複雑なセキュリティ認可とトリガー 13-31
透過的なイベント・ロギングとトリガー 13-32
導出列値とトリガー 13-33

14 動的 SQL の使用

動的 SQL の概要 14-2
DBMS_SQL パッケージを作成する 14-2
DBMS_SQL の使用 14-3
実行の流れ 14-4
動的 SQL のセキュリティ 14-7
Oracle Server のユーザーの場合 14-7
Trusted Oracle Server のユーザーの場合 14-7
プロシージャと関数 14-8
OPEN_CURSOR 関数 14-9
PARSE プロシージャ 14-10
BIND_VARIABLE プロシージャと BIND_ARRAY プロシージャ 14-11
問合せの処理 14-15
DEFINE_COLUMN プロシージャ 14-15
DEFINE_ARRAY プロシージャ 14-17
DEFINE_COLUMN_LONG プロシージャ 14-18
EXECUTE 関数 14-19
EXECUTE_AND_FETCH 関数 14-19
FETCH_ROWS 関数 14-19
COLUMN_VALUE プロシージャ 14-20
COLUMN_VALUE_LONG プロシージャ 14-22
VARIABLE_VALUE プロシージャ 14-23
更新、挿入、削除の処理 14-24
IS_OPEN 関数 14-24
DESCRIBE_COLUMNS プロシージャ 14-24
CLOSE_CURSOR プロシージャ 14-26
エラーの位置 14-28
LAST_ERROR_POSITION 関数 14-28
LAST_ROW_COUNT 関数 14-28
LAST_ROW_ID 関数 14-28
LAST_SQL_FUNCTION_CODE 関数 14-28
DBMS_SQL の使用例 14-28

15 スキーマ・オブジェクト間の依存性

依存性の問題 15-2

実行時の再コンパイルを避ける 15-2
リモート依存性 15-4
手動で再コンパイルする 15-4
ビューを手動で再コンパイルする 15-4
プロシージャと関数を手動で再コンパイルする 15-5
パッケージを手動で再コンパイルする 15-5
トリガーを手動で再コンパイルする 15-5
依存性の管理情報をリストする 15-6
依存性追跡ユーティリティ 15-7

16 アラート機能によるデータベースのイベントの通知

概要 16-2
DBMS_ALERT パッケージを作成する 16-3
セキュリティ 16-3
エラー 16-3
アラートの使用 16-4
REGISTER プロシージャ 16-5
REMOVE プロシージャ 16-5
SIGNAL プロシージャ 16-5
WAITANY プロシージャ 16-6
WAITONE プロシージャ 16-7
アラートの検査 16-8
SET_DEFAULTS プロシージャ 16-8
アラートの使用例 16-9

17 セキュリティ方針の設定

アプリケーションのセキュリティ方針 17-2
アプリケーション管理者 17-2
ロールとアプリケーションの権限管理 17-2
アプリケーション・ロールを使用可能にする 17-3
Tool ユーザーに対してアプリケーション・ロールを制限する 17-5
スキーマ 17-6
権限とロールを管理する 17-7
ロールを作成する 17-8
ロールを使用可能、使用禁止にする 17-9

ロールを削除する 17-12
権限とロールの付与および取消し 17-12
ユーザー・グループ PUBLIC に対して付与する、取り消す 17-17
付与と取消しはいつ有効になるか 17-17
権限付与は依存オブジェクトにどのように影響するか 17-17

18 Oracle XA

XA ライブラリ関連情報 18-2
Oracle XA に関する一般的な情報 18-2
README.doc 18-2
リリース 7.3 からリリース 8.0 への変更点 18-2
セッション・キャッシュが不要になった 18-2
動的な登録がサポートされた 18-3
疎結合トランザクション・ブランチがサポートされた 18-3
OCI アプリケーションに SQLLIB が不要になった 18-3
XA を実行するためのインストレーション・スクリプトが不要になった 18-3
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できるようになった 18-3
Oracle Parallel Server のトランザクション回復が改善された 18-4
グローバル・トランザクションとローカル・トランザクションの両方が可能になった 18-4
xa_open 文字列が修正された 18-5
一般的な問題および制限事項 18-6
データベース・リンク 18-6
Oracle Parallel Server オプション 18-7
SQL に基づく制限事項 18-7
その他の XA に関する問題 18-8
基本アーキテクチャ 18-10
X/Open Distributed Transaction Processing(DTP) 18-10
トランザクション回復管理 18-12
Oracle XA ライブラリ・インタフェース・サブルーチン 18-12
XA ライブラリ・サブルーチン 18-13
XA インタフェースの拡張 18-13
トランザクション処理モニター（TPM） 18-14
必須のパブリック情報 18-14
登録 18-15

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 18-16
DBA またはシステム管理者の責任 18-16
アプリケーション開発者の責任 18-17
xa_open 文字列の定義 18-17
xa_open 文字列の構文 18-17
必須のフィールド 18-18
オプションのフィールド 18-20
プリコンパイラおよび OCI へのインタフェース 18-22
Oracle XA ライブラリとプリコンパイラの使用 18-23
Oracle XA ライブラリと OCI の使用 18-24
トランザクション制御 18-26
プリコンパイラ・アプリケーションの例 18-27
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 18-28
XA ライブラリ・スレッド・セーフティ 18-29
オープン文字列の仕様 18-29
制限事項 18-30
障害追求 18-30
トレース・ファイル 18-30
トレース・ファイルの例 18-31
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 18-32
Oracle Server SYS アカウント表 18-32

索引

はじめに

このマニュアルでは、Oracle Server リリース 8.0 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稼働する 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。

「はじめに」は次のセクションで構成されています。

- このマニュアルについて
- 本書の構成
- このマニュアルの表記規則

このマニュアルについて

アプリケーションの開発者として、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することができる多数の Oracle Server 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ます。このマニュアルでは、PL/SQL 言語については言及せず、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直接説明はしません。記載内容の詳細は、目次および第 1 章を参照してください。第 1 章では、関連情報を含むその他の Oracle ドキュメンテーションについても示されています。

このマニュアルの対象読者

『Oracle8 Server アプリケーション開発者ガイド』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・アナリストおよびプロジェクト管理者、データベース・アプリケーションの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、構造化照会言語（SQL）を使用してリレーショナル・データベース・システム内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想定しています。

機能範囲と可用性

このマニュアルは、Oracle8 と Oracle8 Enterprise Edition 製品の特徴と機能を説明しています。Oracle8 と Oracle8 Enterprise Edition の基本機能は同じです。ただし、最新の機能のいくつかは、Enterprise Edition だけで使用可能であり、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 と「オブジェクト」オプションが必要です。

Oracle8 と Oracle8 Enterprise Edition の相違点、使用可能な機能とオプションの詳細は、Oracle8 と Oracle8 Enterprise Edition の解説を参照してください。

関連資料

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

Oracle コール・インタフェース（OCI）については、次のマニュアルで説明されています。

- 『Oracle コール・インタフェース・プログラマーズ・ガイド』

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。これを使用して、ご使用のアプリケーション・プログラムに SQL および PL/SQL を組み込めます。埋込み SQL を取り入れる 3GL アプリケーション・プログラムを Ada または C、C++、COBOL、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、『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 のマニュアルを参照してください。

SQL の情報は、『Oracle8 Server SQL リファレンス』および『Oracle8 Server 管理者ガイド』を参照してください。Oracle の基本概念は、『Oracle8 Server 概要』を参照してください。

本書の構成

『Oracle8 Server アプリケーション開発者ガイド』は、18 の章で構成されています。次に各章の内容を簡単にまとめて示します。

第 1 章「アプリケーション開発者の情報源」

このマニュアルおよび Oracle の他の技術資料で、特定のアプリケーション開発トピックについての情報の記載位置を見つける方法を示しています。

第 2 章「アプリケーション開発者」

Oracle Server アプリケーション開発のプロセスについて概要を説明します。

第 3 章「SQL 文の処理」

各種の SQL コマンドと PL/SQL 文を処理するために Oracle Server が実行するステップについて説明します。

第 4 章「スキーマ・オブジェクトの管理」

表およびビュー、数値順序、シノニムを含む、特定のユーザーのデータベース定義域（スキーマ）に作成できるオブジェクトの管理方法について説明します。索引とクラスタを使用してデータ検索のパフォーマンスを改善する方法についても説明します。

第 5 章「データ型の選択」

Oracle データ型の正しい選択方法について説明します。固定長文字列、可変長文字列、数値データ、日付、バイナリ・データ、行識別子（ROWID）などのデータ型について説明します。

第 6 章「ラージ・オブジェクト (LOB)」

LOB データ型のために拡張された SQL コマンドおよび PL/SQL インタフェースについて説明します。構造化されていないバイナリ・データについては BLOB、文字データについては CLOB および NCLOB、外部ファイルに格納されているデータについては BFILE があります。

第 7 章「ユーザー定義データ型 ー 拡張例」

特定のアプリケーション要件に応じて作成できる、複合データ型および集合データ型（可変長配列とネストした表）の定義および使用方法を説明します。

第 8 章「オブジェクト・ビュー ー 拡張例」

オブジェクト・ビューの定義および使用方法について説明します。

第 9 章「データの整合性の保持」

宣言整合性制約を使用して、Oracle データベース内のデータの整合性を実現する方法について説明します。

第 10 章「プロシージャとパッケージの使用」

データベースに格納して継続的に使用できるプロシージャの作成方法について説明します。また、複数のプロシージャをパッケージにグループ化する方法についてもこの章で説明します。

第 11 章「アドバンスト・キューイング」

クライアント / サーバー環境において処理の実行を延期または正規化するアドバンスト・キューイングの使用方法について説明します。

第 12 章「PL/SQL 入出力」

同一の Oracle Server インスタンス内のセッションが互いに、またはディスク・ファイルと通信できるようにするためのパブリックおよびプライベートのパイプの使用方法を説明します。

第 13 章「データベース・トリガーの使用」

データベース・トリガーの作成およびデバッグ方法について説明します。数多くのサンプルが掲載されています。

第 14 章「動的 SQL の使用」

動的 SQL を使用して、ストアド・プロシージャおよび無名 PL/SQL ブロックを作成する方法について説明します。

第 15 章「スキーマ・オブジェクト間の依存性」

関連するビュー、プロシージャ、パッケージ、トリガー間の依存性を管理する方法について説明します。

第 16 章「アラート機能によるデータベースのイベントの通知」

データベース内の関心のあるアプリケーションの値が変更されたら必ず通知されるようにアプリケーションを設計する方法について説明します。

第 17 章「セキュリティ方針の設定」

Oracle のセキュリティ機能を使用して、セキュリティ方針を設定する方法について説明します。

第 18 章「Oracle XA」

この章では、Oracle XA ライブラリの使用方法について説明します。

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およびフォーマットの規則は、次のとおりです。

[ ]

角カッコは、中に含まれている項目がオプションであることを示します。カッコは入力しないでください。

{ }

山カッコは、複数の項目を囲み、その中の 1 つだけが必要であることを示します。

|

縦棒は、山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ためにも使用します。

...

コードの断片部分における省略記号は、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意味します。

フォントの変更

SQL または C コード例は、固定幅フォントで表示されます。

イタリック

イタリックのテキストは、OCI パラメータおよび OCI ルーチン名、ファイル名、データ・フィールドに使用されます。

大文字

大文字のテキストは、SELECT または UPDATE などの SQL キーワードについて注意を促すために使用されます。

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、特別なテキスト・フォーマットを使用しています。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。次の各段落では、それぞれの見出しで示される各種の情報を説明します。

> **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なければならないこと、および行ってはならないことを「警告」として OCI プログラマに示します。

関連項目 : 説明しているトピック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セクション、または他のマニュアルを示します。

1

アプリケーション開発者の情報源

この章では、このマニュアルで説明するトピックのうちのいくつかを挙げて、各トピックについての情報が記載されている箇所を示します。

情報源

Oracle Server は大規模な製品です。Oracle Server および言語製品のマニュアルは、20 冊以上から構成されています。これらに加えて、Net8 を使用してクライアント・アプリケーションを Oracle Server に接続するための重要な情報を提供する、何冊かのネットワーク関連のマニュアルがあります。

この章では、Oracle のアプリケーション開発製品に関する基本的な情報と、それらの製品のマニュアルについて説明します。Oracle のマニュアルの他に、Oracle に関する一般書籍も多くなってきています。お近くの技術関連書店や大学の書店で、それらの書籍を入手できます。

特定のトピック

このセクションでは、次のトピックについての情報の入手場所を示します。

- 業務ルール
- クライアント側のツール
- 3GL プログラムとの通信
- データベース制約
- データベース設計
- データ型
- デバッグ
- エラー処理
- ゲートウェイ
- Oracle が提供するパッケージ
- PL/SQL
- スキーマ・オブジェクト
- セキュリティ
- SQL 文

業務ルール

表中の列に整合性制約を使用したり、トリガーを使用することによって、Oracle アプリケーションに業務ルールを施行することができます。

関連項目：

- 整合性制約についてはこのマニュアルの第 9 章「データの整合性の保持」を参照してください。
- データベース・トリガーについては第 13 章「データベース・トリガーの使用」を参照してください。
- 業務ルールの詳細は、『Oracle8 Server 概要』を参照してください。

クライアント側のツール

Oracle は、アプリケーションの開発を支援する、多くのツールを提供しています。Oracle の Developer/2000 の Tool セットは、Procedure Builder、クライアント側のデバッガが付属した PL/SQL 開発環境、フォームおよびレポートを作成するその他のツールを提供します。

データベース設計を支援する CASE ツールが、Oracle Designer/2000 製品の一部として利用できます。

3GL プログラムとの通信

3GL ルーチン（C や C++ の関数）や、オペレーティング・システムのサービスをサーバー上で動作している PL/SQL コードからアクセスする方法について、アプリケーションの開発者からよく質問を受けます。これを行う 1 つの方法としては、オラクル社 の DBMS_PIPES パッケージを使用するというものがあります。

関連項目： このパッケージの詳細は、第 12 章「PL/SQL 入出力」を参照してください。

警告： DBMS_PIPES パッケージはトランザクションについて安全性が保証されていないため、十分注意して使用してください。

Oracle Server リリース 7.3 では、PL/SQL から直接 3GL ルーチンをコールすることは できませんでした。Oracle8 でこの作業を行うための方法については、『PL/SQL ユーザーズ・ガイド・リファレンス』を参照してください。

データベース制約

NOT NULL、PRIMARY KEY、FOREIGN KEY といったデータベース制約の使用方法は、このマニュアルの第 9 章「データの整合性の保持」を参照してください。制約に関する基本的な説明は、『Oracle8 Server 概要』を参照してください。

データベース設計

データベース設計については、このマニュアルでは詳しく説明していません。基本的な説明は、『Oracle8 Server 概要』を参照してください。データベースのパフォーマンスを考慮した設計については、『Oracle8 Server チューニング』を参照してください。高度データベース設計用ツールについては、Oracle Designer/2000 製品も参照できます。

データ型

Oracle の内部データ型については、このマニュアルの第 5 章「データ型の選択」を参照してください。データ型の包括的な扱いは、『Oracle 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。

デバッグ

PL/SQL コードのデバッグの第一段階として、DBMS_OUTPUT および DBMS_PIPES パッケージを使用できます。このパッケージの詳細は、第 12 章「PL/SQL 入出力」を参照してください。

エラー処理

Oracle アプリケーションのコンパイルまたは実行時にエラーや警告が発生した際には、Oracle のエラー・コードと、通常は簡単なエラー・メッセージによって通知されます。完全なリストまたは Oracle サーバーおよびプリコンパイラ、PL/SQL のエラー・コードおよびメッセージは、『Oracle8 Server エラー・メッセージ』を参照してください。

PL/SQL は、実行時に例外を発生させることもできます。事前定義済みの PL/SQL 例外およびその原因については、『PL/SQL ユーザーズ・ガイド・リファレンス』を参照してください。

ゲートウェイ

Oracle のオープン・ゲートウェイ技術を使用して、Oracle 以外のデータベースのデータや、リレーショナルでないデータ・ソースにさえアクセスすることができます。ゲートウェイ・アプリケーションの開発については、『Oracle Open Gateway Toolkit Guide』を参照してください。

Oracle が提供するパッケージ

Oracle では、一連の PL/SQL パッケージを提供し、アプリケーション開発を支援しています。これらのパッケージの名前は、ほとんどが DBMS_OUTPUT や DBMS_SQL のように接頭辞 DBMS_ で始まります。UTL_FILE(PL/SQL ファイルの入出力) など、一部には異なる接頭辞で始まるものもあります。

Oracle が提供するこのパッケージについては、このマニュアルで、主に第 10 章「プロシージャとパッケージの使用」、および第 12 章「PL/SQL 入出力」、第 13 章「データベース・トリガーの使用」、第 14 章「動的 SQL の使用」、第 16 章「アラート機能によるデータベースのイベントの通知」で説明されています。

関連項目： Oracle が提供するパッケージとそれぞれの解説箇所については、10-56 ページの「提供されるパッケージ」を参照してください。

PL/SQL

PL/SQL 言語の説明は、主に、『PL/SQL ユーザーズ・ガイド・リファレンス』にあります。アプリケーション開発での PL/SQL の使用方法については、このマニュアルもあわせて参照してください。PL/SQL 言語を解説している一般書籍もあります。

スキーマ・オブジェクト

このマニュアルは、表およびビュー、パッケージ、プロシージャ、順序といったスキーマ・オブジェクトの作成および修正、削除について説明しています。ただし、入門書としては、『Oracle8 Server 概要』も参照してください。たとえば、このマニュアルの第 10 章「プロシージャとパッケージの使用」を参照する前に、『Oracle8 Server 概要』の「プロシージャおよびパッケージ」という章を読んでください。

セキュリティ

このマニュアルでは、セキュリティについては主に第 17 章「セキュリティ方針の設定」で説明しています。セキュリティについては、『Oracle8 Server 概要』マニュアルでも説明しています。

SQL 文

SQL 言語については、『Oracle8 Server SQL リファレンス』で主に説明しています。そのマニュアルでは、Oracle での SQL 言語の実行について詳しく説明しています。また、すべての SQL コマンドの形式の構文図も含まれています。

ツール

関連項目： 1-3 ページの「クライアント側のツール」を参照してください。

2

アプリケーション開発者

この章では、Oracle データベース・アプリケーションの設計とインプリメントの手順の概要を簡単に説明します。これらの任務を遂行するために必要となる詳細な情報については、後続の章で説明します。個々の作業は、開発しているアプリケーションの種類と複雑さによって異なりますが、一般的にアプリケーション開発者の責務は次のことを行います。

- アプリケーションのためのデータベース構造の設計
- データベース・アプリケーションの設計と開発
- SQL コードを記述する
- アプリケーションでセキュリティを施行する
- アプリケーションの調整
- アプリケーションをメンテナンスおよび更新する

このマニュアルは、データベースやアプリケーションの設計に関するテキストとして 使用すべきものではありません。これらの分野に精通していない場合には、適切なテキストを捜して学習する必要があります。このマニュアルの対象読者の要件に該当している場合は、このマニュアルの別の項に進んで、さらに情報を得てください。

必要性を評価する

実用的なアプリケーションを設計するための第 1 段階は、解決すべき問題を明確にすることです。つまり、データを漠然と捉えないことが重要になります。データがどのように使用されるのかを考えてください。アプリケーションの設計では、次のような問いに答えられなければなりません。

- 誰がこのアプリケーションを使用するのか。
- このアプリケーションを使用して何を達成しようとしているのか。
- どのようにしてこれらの業務を達成するのか。

設計の初期の段階では、できるかぎりエンドユーザーと密接な関係を保つ べきです。これは、アプリケーションの目的に関する誤解から発生し得る問題 をなくすための手助けとなります。アプリケーションのエンドユーザーが遂行しようとしている業務をよく理解してから、これらの業務を完了するために 必要なデータを決定します。この段階では、各業務について調査し、次のような問いに明確に答える必要があります。

- この業務を遂行するためには、どのデータが使用可能になっていなければならないか。
- このデータをどのように処理しなければならないか。
- これらの結果をどのように表現することができるか。
- このアプリケーションは将来どのように使用されるか。

重要なことは、アプリケーションのユーザー（依頼者）がこちらの提示した解決方法を明確に理解することです。また、アプリケーションをユーザーの変更依頼に対応可能に設計することも重要です。

データベースを設計する

この時点で、データ・モデルを設計するための準備が完了しています。このモデルによって、データを最も効率的に格納し、使用する方法を判断することが できます。エンティティ・リレーションシップ（E-R）モデルは実社会のシステムをリレーショナル・データベース管理システムに対応させる場合によく使用します。

エンティティ・リレーションシップ・モデルはシステムのすべての要素をエンティティ（人、場所、もの）またはエンティティ間のリレーションシップに分類します。どちらも表という同じ構造で表現されます。たとえば、注文入力システムでは、部品と注文がエンティティです。部品と注文の情報は両方とも、表に示されます。そして、どの注文がどの部品を要求しているのかというリレーションシップは第 3 の表で表現されます。エンティティ・リレーションシップ・モデルを適用するには、次のようなステップが必要です。

1. システムのエンティティを識別し、エンティティを表現する表を構成する。
2. エンティティ間のリレーションシップを識別し、現行の表を拡張するかあるいは新しい表を作成してこれらのリレーションシップを表現する。
3. 各エンティティの属性を識別し、表を拡張してこれらの属性を組み入れる。

エンティティ・リレーションシップ・モデルでシステムをモデリングする場合、正規化と呼ばれるステップが含まれることがあります。データベース設計に関するテキストで、第 3 正規形を実現する方法について説明しています。第 3 正規形では、それぞれの表に主キーが 1 つ存在し、表のすべてのデータがその表の主キーに完全に依存します。必要なパフォーマンス・レベルに達成するために、正規形に違反することも必要になる場合があります。

エンティティ・リレーションシップ・モデルを適切に適用すると、優れた表設計を行うことができます。表を適切に設計することによって、次のような利点が得られます。

- 冗長なデータ記憶の低減によって、重複したデータの更新コストを取り除き、重複による矛盾した結果がもたらす危険を回避する。
- 効果的に整合性制約を施行できるようになる。
- システムの機能拡張や変更に対して柔軟に対応できるようになる。
- 適切に設計されたリレーショナル・システムが本来持つ柔軟性を基盤として、生産性が向上する。

オラクル社のデータベース設計用の製品は、設計の改善、自動化、文書化に有効です。オラクル社のデータベース設計用製品には、Designer/2000 および Object Database Designer があります。

Designer/2000 はビジネスおよびアプリケーションのモデリング用ツールセットで、サーバーとアプリケーションの両方をグラフィカル・モデルから作成します。Designer/2000 リリース 2.0 はユーザー定義ビュー、およびオブジェクト・テーブル、参照オブジェクト型と埋込みオブジェクト型などのオブジェクト機能だけでなく、パーティション表、LOB データ型、索引構成表、遅延制約検査などのすべての Oracle8 拡張機能もサポートしています。

Object Database Designer は、オブジェクト・リレーショナル・データベース管理システム (ORDBMS) の設計と作成を全面的にサポートしています。型のモデル化はオブジェクト指向開発の中核となるものです。Object Database Designer は、UML を使用して型のモデル化をインプリメントし、さらにその型モデルを使用して Oracle8 データベース設計と透過性を保持する C++ クラスを生成することによって、データベース設計者とアプリケーション開発者の両方をサポートします。

Designer/2000 および Object Database Designer の詳細は、各製品のマニュアルを参照してください。

データベース内にある表の全体的な構造を決定したら、次に個々の表の構造を設計しなければなりません 。このプロセスでは、それぞれの列に対して適切なデータ型を選択し、意味のある名前を割り当てます 。このマニュアルの第 5 章「データ型の選択」に適切な Oracle データ型の選択についての情報が記載されています。

分散データベースで実行するアプリケーションを設計する場合は、データ自身の配置場所と、ネットワークからそのデータにアクセスするのに必要なリンクについても決定しなければなりません。

関連項目： 詳細は、『Oracle8 Server 分散システム』を参照してください。

アプリケーションを設計する

データベース設計が完了すると、アプリケーションの設計を開始することができます。これも非常に反復的なプロセスであり、これによってデータベース設計を再検討することもあるでしょう。可能な限り、アプリケーションのユーザーをこれらの設計の意思決定に参画させるべきです。可能な限り早くアプリケーションをエンドユーザーが使用できるようにして、必要なフィードバックを得ることで、設計を的確に調整すべきです。

他のベンダーだけでなくオラクル社からも、アプリケーションの開発とインプリメンテーションを支援するのに使用できるツールは数多く提供されています。最初の仕事は、使用可能なツールを評価し、最も適切なツールを見つけることです。

利用可能な機能を使用する

次に、前の段階で選択した他のツールおよびユーティリティに加えて、Oracle で利用可能な機能を使用して、要件をインプリメントする方法を決定しなければなりません。アプリケーションをインプリメントするために使用する機能とツールは、アプリケーションのパフォーマンスに著しい影響を与えます。効率のよいアプリケーションを設計することに努力すれば、結果的にアプリケーションのチューニングに費やさなければならない時間が少なくなります。

Oracle のアプリケーション開発者に利用可能なより有効な機能を、次にいくつかリストします。各機能については後で詳しく説明します。

整合性制約

整合性制約によって、表に収録できるデータについての特定の要件を定義し、データの 入力方法とは無関係に、これらの要件が満足されることを保証できます。これらの制約は表定義に含まれているため、プログラミングを行う必要はありません。

> **関連項目：** 整合性制約の使用方法は、第 9 章の「データの整合性の保持」を参照してください。

ストアド・プロシージャとパッケージ

共通に使用するプロシージャを PL/SQL で一箇所に記述して、データベースに格納し、アプリケーションで繰り返し使用できます。これによって、アプリケーション間で一貫した動作が保証され、開発とテストの時間を節約できます。

関連するプロシージャはグループ化してパッケージにすることができ、パッケージはパッケージ本体とは別にパッケージ仕様部を持ちます。パッケージ本体は、パッケージ仕様部に影響を及ぼさずに、変更し、再コンパイルすることができます。これによって、パッケージ本体を変更しても、エンドユーザーにはわからないようにし、そのパッケージ指定を参照しているオブジェクトを再コンパイルする必要がないようにできます。

> **関連項目：** 詳細は、第 10 章の「プロシージャとパッケージの使用」を参照してください。

データベース・トリガー

宣言整合性制約を使用して施行できない複雑な業務ルールを、トリガーを使用して施行できます。トリガーは、PL/SQL 無名ブロックに類似したものであり、ユーザーやアプリケーションとは関係なく、トリガー文が発行されると自動的に実行されます。

> **関連項目：** 詳細は、第 13 章の「データベース・トリガーの使用」を参照してください。

データベース・トリガーには、値ベースの監査の実行、導出データ値の維持管理、複雑なセキュリティと整合性規則の施行など、多様な用途があります。これらのコードをアプリケーションからデータベース・トリガーに移行することで、すべてのアプリケーションが同じように動作することが保証されます。

コスト・ベースのオプティマイザ

コスト・ベースの最適化手法は、使用可能な索引に関する情報とあわせて、表に関する統計情報を使用して、SQL 文の実行計画を選択します。これによって、経験の浅いユーザーもパフォーマンスを気にしないで複雑な問合せを実行できます。

アプリケーション設計者として、表のデータに関する知識を使用してよりよい実行パスを選択できるのに、オプティマイザではそのような知識を使用できない場合があるかもしれません。このような場合には、オプティマイザにヒントを与えて、適切な実行パスを選択させることができます。詳細は、『Oracle8 Server チューニング』を参照してください。

共有 SQL

共有 SQL によって、複数のユーザーが、プロシージャおよび SQL 文の単一の実行時コピーを共有して、メモリー要件を大幅に低減することができます。2 つの同じ SQL 文が発行されると、文の最初のインスタンスを処理するために使用される共有 SQL 領域が、同じ文の次のインスタンスを処理するために再使用されます。

その他のアプリケーション開発者だけではなく、データベース管理者（DBA）とも調整を行い、類似した業務を遂行する文とブロックが同一の共有 SQL 領域をできるだけ頻繁に使用できるようにガイドラインを作成してください。詳細は、『Oracle8 Server チューニング』を参照してください。

各国語サポート

Oracle はシングル・バイトとマルチバイトのコード体系のどちらもサポートします。言語依存のデータはコードとは別に格納されますので、アプリケーション・コードを変更せずに、新しい言語や言語固有の機能（日付書式など）を容易に追加することができます。各言語サポートの詳細は、『Oracle8 Server リファレンス』を参照してください。

ロック

デフォルトでは、Oracle は行レベル・ロックを適用して、ロックの競合なしで複数ユーザーが同じ表中の別の行をアクセスできるようにします。これによって、デッドロックが発生す

る確率はかなり小さくなりますが、アプリケーションを設計する上でも十分注意して、デッドロックが発生しないことを保証する必要があります。

行レベル・ロックによって最も利益を得るのは、オンライン・トランザクション処理アプリケーション、つまり、複数のユーザーが同一の表の別の行を同時に修正するようなアプリケーションです。このような機能に注意してアプリケーションを設計するとよいでしょう。

Oracle ロックは、アプリケーションでも同じように使用することができます。これらのロックは、DBMS_LOCK パッケージの一部として提供されています。詳細は、第 3 章の「SQL 文の処理」を参照してください。

プロファイル

プロファイルを使用して、1 回の問合せおよびセッションが使用するリソースに制限を設けることができます。アプリケーションを設計するときに、限られたリソースが原因でシステムにアクセスできなくなるかどうかを判断する必要があります。プロファイルを使用すれば、不定期のユーザーに、データベースへの制限付きアクセスを許可することもできます。このアクセスを許可する場合は、設計時に要件を検討しなければなりません。プロファイルは一般にデータベース管理者によって管理されます。追加のユーザーにアクセス権を付与することができるかどうかを判断し、このようなユーザーを識別するために、データベース管理者と相談してください。

順序

順序番号を使用して、データに対して一意なキーを自動的に生成し、複数の行または表にまたがるキーを調整できます。順序番号の生成では、最も最近使用した値をロックし、値を増分することにより、一意の数値をプログラム的に生成することによって発生する直列化を防ぐことができます。順序番号は、ディスクのかわりに、順序番号キャッシュから読み込んで、さらに処理を高速化することもできます。

業界標準

Oracle は、業界標準に準拠して設計されています。アプリケーションを業界標準に準拠させなければならない場合には、Oracle の SQL 標準への規格合致性について詳しく説明している『Oracle8 Server SQL リファレンス』を参照する必要があります。

Oracle コール・インタフェースの使用

Oracle コール・インタフェース (OCI) を使用するアプリケーションを開発する際には、OCI が次のものを提供することを知っておく必要があります。

- 接続機能
- LONG または LONG RAW 列について、以前からの機能である、それらの列の一部の選択に加えて、列の一部の挿入および削除機能
- 結合および定義操作での C 構造体の配列の使用
- OCI アプリケーション用のスレッド・セーフ・ライブラリ

詳細は、『Oracle 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。

SQL を作成する

Oracle データベース内の情報に対して行われる操作はすべて、SQL 文を使用して実行されます。アプリケーションの設計が完成したあと、この設計を実装するために使用する SQL 文の設計に取りかかる必要があります。アプリケーションのコーディングを開始する前に、SQL を徹底的に理解すべきです。SQL 文の実行については、このマニュアルの 3 章で説明します。

> **関連項目：** 詳細は、『Oracle8 Server SQL リファレンス』を参照してください。

使用する SQL 文を調整することによって、アプリケーションのパフォーマンスを大幅に改善できます。SQL 文の調整方法は、『Oracle8 Server チューニング』で説明しています。

アプリケーションでセキュリティを施行する

アプリケーションの設計は、アプリケーションのセキュリティ要件を決定するまで完了しません。ここまでの作業で、アプリケーション設計の一部として、それぞれのユーザーやユーザーのグループが遂行する必要のある業務を識別しました。次に、これらの業務を遂行するのにどんな権限が必要かを決定しなければなりません。データベースのセキュリティ上、ユーザーには業務の完了に必要なアクセス権だけを付与し、それ以上は付与しないようにすることが重要です。

アプリケーションで、ユーザーがそのアプリケーションを実行するときに適切なロールを使用可能にすることによって、計画したデータベースにだけユーザーがアクセスすることを保証できます。ロールは通常データベース管理者によってユーザーに付与されますので、データベース管理者と相談して、指定した業務についてアプリケーションで必要なロールへのアクセス権が各ユーザーに付与されるようにしてください。

> **関連項目：** セキュリティ方針の詳細は、第 17 章の「セキュリティ方針の設定」を参照してください。

アプリケーションをチューニングする

データベース・アプリケーションをチューニングするときには、次の重要な 2 つの観点から検討すべきです。

- SQL 文のチューニング
- アプリケーション設計のチューニング

コストベースの最適化の使用方法を含む、SQL 文のチューニングに関する説明は、『Oracle8 Server チューニング』に説明されています。アプリケーション設計のチューニングは、アプリケーションのインプリメントを開始する前に行うとよいでしょう。さらに、設計を開始する前に、このマニュアルで説明されているそれぞれの機能をよく理解し、自分の要件に最も

よく合う機能について検討すべきです。検討すべき設計上の問題には次のようなものがあります。

- プログラムで処理するかわりに、整合性制約によって業務ルールを施行する。

  関連項目： 整合性制約を使用する場合の説明については、9-2 ページの「整合性制約の使用」を参照してください。

- パフォーマンスを改善するために、PL/SQL を使用する。PL/SQL 文によるパフォーマンスの改善については、『Oracle8 Server チューニング』で説明しています。
- パフォーマンスをさらに改善し、実行時の再コンパイルを低減するために、パッケージを使用する。パッケージについては、第 10 章の「プロシージャとパッケージの使用」で説明しています。
- 主キーの値を生成するために順序番号をキャッシュして使用する。詳細は、4-23 ページの 「順序を作成する」を参照してください。
- Oracle へのコール数を低減するために配列処理を使用する。詳細は、『Oracle8 Server チューニング』を参照してください。
- 文字データを格納する場合は、データを空白で埋める CHAR のかわりに VARCHAR2 を使用する。詳細は、5-5 ページの 「文字データ型を使用する」を参照してください。
- LONG データ型または LONG RAW データ型のかわりに、LOB データ型を使用する。詳細は、第 6 章の「ラージ・オブジェクト (LOB)」を参照してください。
- LONG データ型または LONG RAW データ型を使用する場合は、データを関連するデータとは別の表に格納し、参照整合性を使用してそれらを関連付ける。LONG データや LONG RAW データを読み込まなくても、関連するデータにアクセスすることができます。詳細は、5-9 ページの 「LONG データ型を使用する」および 5-11 ページの 「RAW データ型と LONG RAW データ型を使用する」を参照してください。
- DBMS_APPLICATION_INFO パッケージ内の SET_MODULE プロシージャと SET_ACTION プロシージャを使用して、実行モジュールまたは実行トランザクションの名前をデータベースに記録し、後でさまざまなモジュールのパフォーマンスをトラックするときに使用できるようにしておく。アプリケーションを登録することによって、システム管理者およびパフォーマンス・チューニングのスペシャリストが、モジュールごとにパフォーマンスを追跡できます。システム管理者は、この情報を使用することによって、モジュールごとにリソースの使用方法を追跡することもできます。アプリケーションをデータベースに登録する際には、その名前および動作は、V$SESSION と V$SQLAREA ビューに記録されます。アプリケーションの登録については、『Oracle8 Server チューニング』を参照してください。

アプリケーションをメンテナンスする、更新する

既存のアプリケーションをアップグレードしたり、既存のデータベース上で実行するアプリケーションを新規作成したりする場合、前述した多くの手順に同じように従わなければなり

ません。アプリケーションのユーザーの要望を理解して特定し、それらに適合するようにアプリケーションを設計しなければなりません。

また、データベース管理者と協力して、次のようなことを判断しなければなりません。

- どの既存アプリケーションが使用可能で、それらがどのように使用されているか。
- データを削除できる場合、または新たに収集しなければならない場合、どのデータが使用可能か。
- データベース構造を変更する必要はあるか？その場合、どのように変更すれば影響を最小限にとどめられるか。

3

SQL 文の処理

この章では、Oracle が構造化照会言語（SQL）文を処理する方法について説明します。トピックは次のとおりです。

- SQL 文の実行
- トランザクションを制御する
- 読取り専用トランザクション
- カーソルの使用
- 明示的なデータのロック
- 行ロックを明示的に獲得する
- SERIALIZABLE パラメータと ROW_LOCKING パラメータ
- ユーザー・ロックを作成する
- ユーザー・ロックの例
- ロックを表示して監視する
- 直列可能トランザクションを使用した同時実行性の制御

いくつかの Oracle 提供のツールとアプリケーションでは、SQL の使用を簡素化したり、マスクしたりしていますが、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・アクセス方法では、Oracle に組み込まれているセキュリティを迂回し、データのセキュリティと整合性を損なわれる可能性があります。

SQL 文の実行

図 3-1 には、SQL 文の処理と実行に通常使用されるステージの概要が示されています。場合によって、これらのステップは少し異なる順序で実行されることもあります。たとえば、コードの書き方によっては、DEFINE ステージが FETCH ステージの直前に実行される場合もあります。

多くの Oracle 提供のツールでは、いくつかのステージは自動的に実行されるので、ほとんどのユーザーは、このレベルの詳細に関与したり、知る必要はありません。しかし、開発者にとっては、Oracle アプリケーションを作成するときに、この情報が役立つこともあるでしょう。各種 SQL 文に対する SQL 処理の各ステージについては、『Oracle8 Server 概要』を参照してください。

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 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 標準を超える拡張が含まれていないかどうか確認します。標準以外の構成体が見つかると、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。

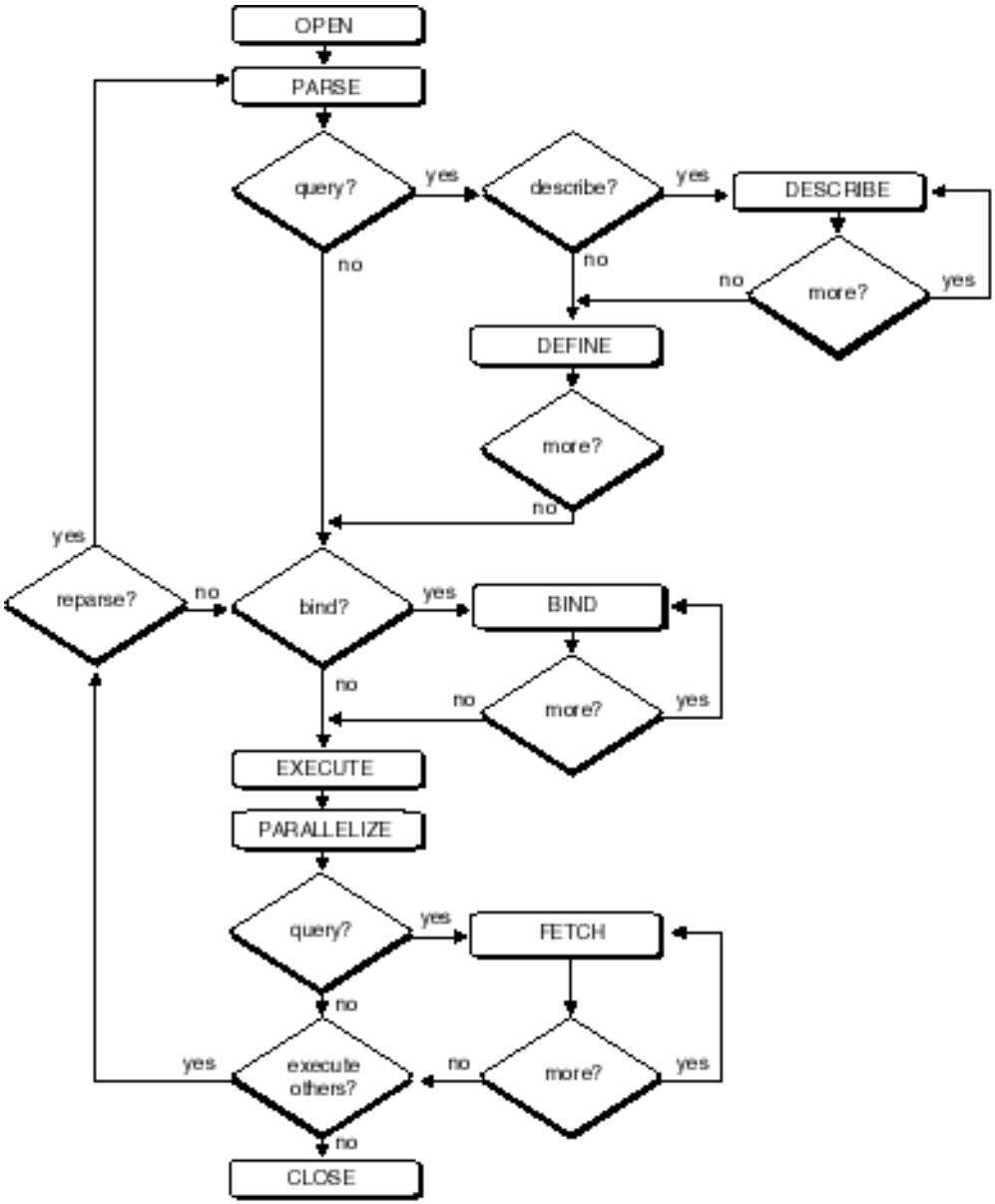
FIPS フラグ付け機能は、Enterprise Manager または SQL*Plus を使用して送られる対話型 SQL 文を介して、フラグ付けをサポートします。また、Oracle プリコンパイラと SQL*Module によって、埋込みおよびモジュール言語 SQL の FIPS フラグ付けもサポートされています。

フラグ付けがオンで、標準以外の 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` 機能の使用が `SQL92 level` レベルではありません。

この level には、ENTRY、INTERMEDIATE、FULL のいずれかが表示されます。

図 3-1　SQL 文の処理ステージ



トランザクションを制御する

一般に、Oracle のプログラム・インタフェースを使用しているアプリケーション設計者だけが、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まとめてグループ化すべきか、ということに関心を持っています。トランザクションは、作業が論理単位で完成され、データが一貫性を保たれるように、適切に定義されなければなりません。トランザクションは、1 つの論理作業単位（これより大きくも小さくもない）として必要な部分のすべてから構成される必要があります。参照されるすべての表の中のデータは、トランザクションが開始する前と、トランザクションが終了した後で、一貫した状態にある必要があります。トランザクションは、データへの 1 つの一貫した変更を含む SQL 文または PL/SQL ブロックだけから構成される必要があります。

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、どちらかです。つまり、借方の存在しない貸方は成立しません。一方の口座への預金など他方の口座と関連を持たない処理は、預金移動トランザクションに含まれません。

パフォーマンスを改善する

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以外に、パフォーマンスを改善するために、なんらかの処置を講じられるかどうかも判断しなければなりません。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の要件を考慮しなければなりません。特に注記しない限り、これらの機能についてはそれぞれ『Oracle8 Server チューニング』で説明しています。

- 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する。
- 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る。これにより、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のエクステントを、動的に割り当てる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 直列可能トランザクションを獲得する。

  関連項目：

  - 3-29 ページの「直列可能トランザクションの相互作用」
  - 『Oracle8 Server 概要』

- 共有 SQL 領域を利用できるように、SQL 文を書くための基準を作成する。Oracle は同一の SQL 文を認識して、メモリー領域の共有を許可します。これによって、データベース・サーバー上のメモリーの使用量が減少し、システム・スループットが増加します。

- SQL 文の最適化へのコストベースのアプローチをインプリメントする統計情報を収集するために、ANALYZE コマンドを使用する。必要に応じて、オプティマイザにさらに " 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ってパフォーマンスを測定するときに使用できるようにしておく。これにより、後でシステムをチューニングする者は、どのトラザクションが一番多くシステム・リソースを必要とするか知ることができます。
- 10-40 ページに記載されているように、SQL 式にユーザー作成 PL/SQL 関数を組み込むことにより、ユーザーの生産性が高まり、問合せの効率が良くなる。
- PL/SQL アプリケーションを書くときに、明示的カーソルを作成する。
- プリコンパイラ・プログラムを書くときに、MAX_OPEN_CURSORS を使用してカーソルの数を増やすと、解析頻度が少なくなり、パフォーマンスが改善されることがよくある。カーソルの使用については、このマニュアルの 3-8 ページで説明しています。

トランザクションをコミットする

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 つの文は等価であり、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、COMMENT パラメータを含めて、これといっしょにコミットされているトランザクションに関する情報を与えるコメント（50 文字未満）を収録できます。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ついての情報を収録する上で有効です。

```
COMMIT COMMENT 'Dallas/Accts_pay/Trans_type 10B';
```

インダウト分散トランザクションのコミットに関する詳細は、『Oracle8 Server 分散システム』を参照してください。

トランザクションをロールバックする

トランザクションの全体、またトランザクションの一部（つまり、セーブポイント）を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 オプションは、何も機能は持っていません。

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 オプションを使用しなければなりません。たとえば、次の文はどちらも、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。

```
ROLLBACK TO SAVEPOINT point1;
ROLLBACK TO point1;
```

インダウト分散トランザクションのロールバックに関する詳細は、次のマニュアルを参照してください。

関連項目： 『Oracle8 Server 分散システム』

トランザクションのセーブポイントを定義する

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に、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きたセーブポイントのことです。

COMMIT、SAVEPOINT、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、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 は定義されたまま残る。
ROLLBACK TO b;	INSERT 文がロールバックされ、セーブポイント C が失われ、セーブポイント B は定義されたまま残る。
ROLLBACK TO c;	ORA-01086 エラー。セーブポイント C は定義されていない。
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）と最後の DML 文（2 番目の INSERT 文）によって行われたアクションがすべてコミットされる。トランザクションのその他のすべての文 （2 番目と 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。

ランザクション管理のために必要な権限

自分のトランザクションを制御するのに必要な権限はありません。どのユーザーでも COMMIT 文、ROLLBACK 文、SAVEPOINT 文を発行できます。

読取り専用トランザクション

特になにも指定しない場合、Oracle の一貫性モデルは、文レベル読取り一貫性を保証しますが、トランザクション・レベル読取り一貫性（反復読取り）を保証しません。トランザクション・レベル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に指定した後は、読取り専用トランザクションでは各問合せの結果がある時刻に対して一貫していることがわかっているので、どのデータベース表に対しても 好き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は、トランザクション・レベル読取り一貫性を与えるために、いかなる追加のデータ・ロックも獲得しません。文レベル読取り一貫性のために使用されるマルチバージョン一貫性モデルが、トランザクションレベル読取り一貫性を与え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始まるとき決定されたシステム制御番号（SCN）に関係する情報を戻します。データ・ロックが獲得されないため、読取り専用トランザクションが問い合せているデータを他のトランザクションが同時に問い合せたり、更新したりできます。

変更されたデータ・ブロックは、読取り専用トランザクションから問合せがあると、ロールバック・セグメントのデータを使用して再構築されます。そのため、長時間実行の読取り専用トランザクションは、「スナップショットが古すぎます（ORA-01555）」というエラーを受け取ることがあります。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。あるいは、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したり、トランザクション中のその他の更新をすべて禁止して問合せ中の表に対して共有ロックを獲得したりすることもできます。

読取り専用トランザクションは、次のように、READ ONLY オプションを含む SET TRANSACTION 文で開始します。たとえば、次のとおりです。

```
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なければなりません。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）または COMMIT 文、ROLLBACK 文、DML 以外の文（たとえば、SET ROLE、ALTER SYSTEM、LOCK TABLE）だけが許可されます。これら以外の文では、エラーが戻されます。COMMIT 文または ROLLBACK 文、DDL 文が、読取り専用トランザクションを終了させます（DDL 文は、読取り専用トランザクションを暗黙にコミットする原因となり、その自分のトランザクションにおいてコミットします）。

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ために、明示的にカーソルを宣言することができます。

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。言い換えれば、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の取出しができます。カーソル変数を使用すると、3GL アプリケーション内のパラメータとしてカーソルを渡すことができます。カーソル変数については、『PL/SQL ユーザーズ・ガイド・リファレンス』に説明されています。

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利用しますが、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利用できる名前付きのリソースであり、特にアプリケーションに埋め込まれた SQL 文の解析に使用できます。

カーソルを宣言しオープンする

1 つのセッションで一度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ので、セッションのカーソルの総数は、プロセスに利用できるメモリーによって制限される。
- セッションごとのカーソルのシステムごとの制限は、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 によって設定される。パラメータについては、『Oracle8 Server リファレンス・マニュアル』で説明しています。

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ことで、それらをアプリケーションのチューニングで利用することができます。たとえば、カーソルの数を増やすことによって、解析の頻度が少なくなり、パフォーマンスが改善されることがよくあります。特定の時間に必要となるカーソル数がわかっていれば、その数を確認して、複数のカーソルを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 文についての情報を十分保持するため、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途中からその文を再実行できます。この様子は、3-3 ページの図 3-1 に示されています。文は、解析ステージを含まずに、再実行できます。

いくつかのカーソルをオープンすることによって、いくつかの SQL 文の解析済みの表現を保存することができます。同じ SQL 文を繰り返し実行する場合は、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と解析の繰返しによるコストは節約できます。

カーソルをクローズする

カーソルをクローズするとは、現在、対応するプライベート領域にある情報が失われ、そのメモリーの割当てが解除されることです。カーソルがいったんオープンされると、次のイベントのうちの 1 つが発生するまでクローズされません。

- ユーザー・プログラムは、サーバーとの接続を終了します。

- 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、宣言済みのカーソルを明示的にクローズできます（しかし、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た状態になります。

注意： Pro*C または PL/SQL を使ってカーソルを取り消すことはできません。

関連項目： カーソルの取消しの詳細は、『Oracle 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行います。ただし、デフォルト・ロック機構を無効にするオプションも利用できます。次のような場合、Oracle のデフォルト・ロックを無効にすると有利です。

- トランザクション・レベルの読取り一貫性、または「反復読取り」をアプリケーションが要求する場合。つまり、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が保証された上で、トランザクションがトランザクション存続時間にわたって一貫した一連のデータを問い合せなければならない場合。トランザクション・レベルの読取り一貫性は、明示的なロックまたは読取り専用トランザクション、直列可能トランザクション、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が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ない。

自動ロック機構は、次の 2 つの異なるレベルで無効にできます。

トランザクション・レベル	LOCK TABLE コマンド、FOR UPDATE 句を含んでいる 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 コマンドを含んでいる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獲得されたロックは、トランザクションのコミット後に解除されたり、またロールバックされたりします。

システム・レベル　初期化パラメータ SERIALIZABLE と 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立ち上げることができます。

次の項では、Oracle のデフォルト・ロックを無効にするために利用可能な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、許可される DML ロックの最大数が決まります（パラメータの説明は、『Oracle8 Server 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必要があるかもしれません。

警告：　いずれかのレベルで Oracle のデフォルト・ロックを無効にする場合には、新しいロック手順が正しく作動することを確認しなければなりません。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ことを確認してください。

表ロックを明示的に獲得する

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獲得します。LOCK TABLE 文は、デフォルト・ロックを手動で置き換えます。ビューに対して LOCK TABLE 文が発行されると、その実表がロックされます。次の文は、それらを含んでいるトランザクションのために、EMP 表と DEPT 表に対して排他表ロックを獲得します。

```
LOCK TABLE emp, dept
    IN EXCLUSIVE MODE NOWAIT;
```

同一のモードでロックするために、複数の表またはビューを指定できます。しかし、ひとつの LOCK TABLE 文では、ひとつのロックモードが指定できるだけです。

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、ロックが獲得されるのを待つ、待たないも指示できます。NOWAIT オプションを指定すると、そのロックがただちに可能である場合にだけ表ロックを獲得します。可能でない場合は、この時点で、ロックできないことを通知するエラーが戻されます。このエラーが通知された場合は、後から再度試みて、そのリソースをロックできます。NOWAIT を指定しないと、トランザクションは、要求した表ロックを獲得するまで処理を進めません。表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、後から再度試みることもできます。つまり、アプリケーションの中でそのような論理をコーディングすることを選ぶこともできます。

注意： 表ロックを待機中の分散トランザクションは、経過した時間が、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にできます。データが変更されていないので、タイムアウトの結果として、アクションはまったく必要ありません。アプリケーションは、デッドロックが検出されたときと同様に処理を続行します。分散トランザクションの詳細は、『Oracle8 Server 分散システム』を参照してください。

次に、LOCK TABLE コマンドを使用して、各種類の表ロックをいつ獲得すると有利であるか説明します。

ROW SHARE と ROW EXCLUSIVE

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

行共有表ロックと行排他表ロックは、最も高い度合の同時実行性を提供します。次のような場合に、行共有表ロックまたは行排他表ロックの明示的な獲得が認められます。

- トランザクション内で表を更新する前に、トランザクションでは、別のトランザクションが共有表ロック、行共有表ロック、排他表ロックの仲裁を獲得することを防止する必要がある場合。つまり、別のトランザクションが共有表ロック、行共有表ロック、排他表ロックの仲裁を獲得すると、他のいかなる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するか、またはロールバックするまで、表を更新することはできません。
- トランザクション内で後から表を修正するには、事前に表の変更や削除を防止する必要がある場合。

SHARE

```
LOCK TABLE table 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。次のような条件では、共有表ロックの明示的な獲得が認められます。

- トランザクションが、表を問い合せるだけで、トランザクションの存続期間にわたって一貫した表のデータの集合を必要とする場合（つまり、ロック中の表に対して、トランザクションレベルの読取り一貫性が必要である）。
- ロック中の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、共有表ロックを有するトランザクションがすべてコミットするか、またはロールバックするまで、待機しなければならない場合。
- トランザクションレベルの読取り一貫性のオプションも与えて、他のトランザクションに同じ表に対する同時実行の共有表ロックを獲得することを許す場合。

警告：　トランザクションによって、同じトランザクション内で後から表を更新できる場合と、できない場合とがあります。しかし、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、（行ロックが SELECT...FOR UPDATE 文の結果として保持されている場合であっても）、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実行の共有表ロックが共通であると、更新処理を進めることができず、デッドロックが共通になります。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 と BUDGET があり、これらの表は第 3 の表 DEPT の一貫した データ・セットを必要としている場合を想定します。つまり、特定の部門番号に対して、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ことを保証する場合を考えます。

このシナリオはきわめてまれな場合ですが、次の例文が示すように、SHARE MODE で DEPT 表を ロックすることによって対処できます。つまり、DEPT 表が更新されることはほとんどありませんが、更新される場合を考慮して、DEPT 表がロックされている間に、EMP 表と BUDGET 表を更新する必要があるでしょう。

```
LOCK TABLE dept IN SHARE MODE
UPDATE EMP
    SET sal = sal * 1.1
    WHERE deptno IN
      (SELECT deptno FROM dept WHERE loc = 'DALLAS')
UPDATE budget
    SET totsal = totsal * 1.1
    WHERE deptno IN
      (SELECT deptno FROM dept WHERE loc = 'DALLAS')

COMMIT /* ロックの解除 */
```

SHARE ROW EXCLUSIVE

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

次のような場合に、共有行排他表ロックの明示的な獲得が認められます。

- トランザクションが、指定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と、ロック中の表を更新することが必要な場合。
- 他のトランザクションによって、明示的な行ロックが取得される心配がない場合（つまり SELECT... FOR UPDATE によって）。これによって、ロック中のトランザクション内の UPDATE 文と 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（つまり、デッドロックが発生することがあります）。

- 上のように動作する単一のトランザクションだけが必要な場合。

EXCLUSIVE

```
LOCK TABLE table IN EXCLUSIVE MODE;
```

次のような場合に、排他表ロックの明示的な獲得が認められます。

- トランザクションが、ロックされている表にただちに更新アクセスすることを必要としている場合。そのため、トランザクションが排他表を保持すると、他のトランザクションはロックされた表の中の特定の行をロックすることはできません。
- トランザクションがコミットされるか、またはロールバックされるまで、トランザクションは、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も観察する場合。
- 順番に表を更新するために、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、行レベルでのデータ同時実行性を意識しなくてかまわない場合。

必要な権限

スキーマ内の表に対してどのタイプの表ロックでも自動的に獲得できます。しかし、別のスキーマ内の表に対して表ロックを獲得するには、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。

行ロックを明示的に獲得する

FOR UPDATE 句を含む SELECT 文によって、デフォルト・ロックを置き換えることができます。SELECT ... FOR UPDATE は、選択されている行を実際に更新することを予定して（UPDATE 文が行うように）、排他行ロックを獲得するために使用されます。

実際にその行を変更しなくても、行をロックするために、SELECT... FOR UPDATE 文を使用できます。その例として、第 13 章の 「データベース・トリガーの使用」のいくつかのトリガーで、参照整合性をインプリメントする方法を示しています。EMP_DEPT_CHECK トリガーでは (13-26 ページを参照)、参照された親キー値を含む行は、トランザクションの存続している間、同じ値を保つことを保証するためにロックされます。親キーが更新あるいは削除された場合、参照整合性は保証されません。

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修正（何度かすることもある）できるような対話形式のプログラムでよく使用されます。単一の対話形式のプログラム・ユーザーだけがある特定の時間に行を更新しているように、行に対する行ロックが獲得されます。

カーソル定義で SELECT ... FOR UPDATE 文が使用される場合、カーソルがオープンされ、最初の取出しの前に、戻される集合の中の行はロックされます。つまり、行はカーソルから取り出されるときに、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だけ、ロックは解除されます。つまり、カーソルがクローズされたときに、ロックが解除されるわけではありません。

SELECT ... FOR UPDATE 文で戻される集合の中の各行は、個別にロックされます。他のトランザクションが競合する行ロックを解除するまで、SELECT ... FOR UPDATE 文は待機します。通常レベルの更新処理の発生する表に対して SELECT ... FOR UPDATE 文を発行し、多くの行がロックされるような場合、そのかわりに排他表ロックを獲得すれば、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獲得する場合、ロックを獲得するのを待つ、待たないを指示できます。NOWAIT オプションを指定すると、そのロックがただちに可能である場合にだけ行ロックを獲得します。可能でない場合は、この時点で、ロックできないことを通知するエラーが戻されます。このエラーが戻された場合は、後で再度試みて、その行をロックできます。NOWAIT を指定しないと、要求した行ロックが獲得されるまで、トランザクションは処理を行いません。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、後から再度試みることもできます。つまり、アプリケーションの中でそのような論理をコーディングすることもできます。

3-11 ページに記述されているように、行ロックを待機中の分散トランザクションは、経過した時間が、初期化パラメータ DISTRIBUTED_LOCK_TIMEOUT で設定された時間間隔に達すると、要求されたロックに対する待機をタイムアウトできます。

SERIALIZABLE パラメータと ROW_LOCKING パラメータ

SET TRANSACTION コマンドまたは ALTER SESSION コマンドの SERIALIZABLE オプション、および ROW_LOCKING 初期化パラメータという 2 つの要素によって、インスタンスがロックを処理する方法が決まります。特に何も指定しない場合、SERIALIZABLE は FALSE に設定され、ROW_LOCKING は ALWAYS に設定されます。

多くの場合、これらのパラメータを変更すべきではありません。これらは、ANSI/ISO 互換モードで実行しなければならないサイト、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ています。つまり、このデフォルトの設定以外を使用するとパフォーマンスは大幅に低下するため、これらのサイトだけがパラメータの変更を検討すべきです。

> **関連項目：** これらのパラメータの詳細は、『Oracle8 Server リファレンス・マニュアル』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、これらのパラメータの設定を変更すべきです。複数インスタンスが単一の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一にすべきです。

デフォルト以外のロック・オプションのまとめ

トランザクションのロックの発生の方法を変更するのに、デフォルト設定以外で 3 通りの SERIALIZABLE と ROW_LOCKING の設定の組合わせができます。表 3-1 に、そのデフォルト以外の設定と、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が要約されています。

表 3-1　デフォルト以外のロック・オプションのまとめ

状況	説明	SERIALIZABLE	ROW_LOCKING
1	バージョン 5 以前の Oracle リリースに等しい（表に対して同時に挿入、更新、削除を行うことはできない）	不可 (デフォルト)	INTENT
2	ANSI 互換	可	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、削除を行うことはできない）	可	INTENT

表 3-2 は、表 3-1 で示した 3 通りの SERIALIZABLE オプションと 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がどうなるかを示しています。

表 3-2　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	RS	RS	RS
ROW EXCLUSIVE MODE	RX	RX	RX	RX	RX	RX
SHARE MODE	S	S	S	S	S	S
SHARE ROW EXCLUSIVE MODE	SRX	SRX	SRX	SRX	SRX	SRX
EXCLUSIVE MODE	X	X	X	X	X	X
DDL statements	-	X	-	X	-	X

ユーザー・ロックを作成する

使用の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で別のプロシージャと区別できる一意の名前を与え、ロック・モードを変更し、解除できます。確保したユーザー・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 ロックの機能をすべて持っています。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ことに注意してください。解除されていないと、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。

DBMS_LOCK パッケージ

Oracle ロック管理サービスは、DBMS_LOCK パッケージのプロシージャによって利用できます。表 3-3 で、DBMS_LOCK パッケージ内の使用可能なプロシージャについて要約しています。

表 3-3　DBMS_LOCK パッケージの関数とプロシージャ

関数 / プロシージャ	説明	参照ページ
ALLOCATE_UNIQUE	名前付きのロックに一意のロック ID を割り当てる	3-18 ページ
REQUEST	特定のモードのロックを要求する	3-19 ページ
CONVERT	あるモードのロックを別のモードに変換する	3-21 ページ
RELEASE	ロックを解除する	3-23 ページ
SLEEP	指定した時間プロシージャをスリープ状態にする	3-23 ページ

ユーザー・ロックは、接頭辞「UL」で識別されるため、Oracle ロックと名前が競合することはありません。これらのロックは、Enterprise Manager ロック・モニター画面または該当する固定ビューを使用して表示できます。

セッションが終了すると、ユーザー・ロックは自動的に解除されます。

警告：　このインプリメンテーションの場合、セッションあたりのロック数が数百以上あると、効率のよいサポートはできません。オラクル社では、このようなユーザー・ロックを使用するための標準規定を作成することを強くお薦めしています。このような規定を作成すれば、同じロックを使用しようとする複数のプロシージャ間での対立が回避できるからです。たとえば、Oracle 提供のアプリケーションで使用されている複数のロック名と対立しないように、ロック名の一部に会社名を含めるとよいでしょう。

セキュリティ

利用できるロックの最大総数については、オペレーティング・システム固有の制限を受ける場合があります。ロックを使用するとき、または他のユーザーがこのパッケージを利用できるようにするときは、このことを**考慮に入れなければなりません。**特定のユーザーやロールだけに EXECUTE 権限を付与することを検討してください。

あるいは、使用されるロック数を制限するカバー・パッケージを作成し、特定のユーザーに EXECUTE 権限を付与するとよいでしょう。カバー・パッケージの例は、DBMSLOCK.SQL パッケージ仕様部ファイルに示されています。

> **関連項目：** コメント表記のパッケージ LOCK_100_TO_200 を参照してください。

DBMS_LOCK パッケージを作成する

DBMS_LOCK パッケージを作成するには、ユーザー SYS として接続し、DBMSLOCK.SQL スクリプトと PRVTLOCK.PLB スクリプトを発行します。これらのスクリプトは、CATPROC.SQL スクリプトによって自動的に実行されます。

> **関連項目：** このパッケージを実行するユーザーへの必要な権限の付与については、10-56 ページを参照してください。

ALLOCATE_UNIQUE プロシージャ

ロック識別子を使用して、アプリケーションがロックの使用を調整できるようにします。ユーザーは、ロック識別子として 0 から 1073741823 までの数を割り当てることができます。または、ロックを名前で識別できます。名前によってロックを識別する場合、ALLOCATE_UNIQUE を使用して、これらの名前付きロックに一意のロック識別番号を生成できます。

> **警告：** **Oracle は、指定された名前に対応付けられているロックを判別するのに SQL を使用するため、名前付きユーザー・ロックは効率的ではありません。**

ALLOCATE_UNIQUE プロシージャのパラメータの説明は、表 3-4 にあります。このプロシージャの構文は、次のとおりです。

```
DBMS_LOCK.ALLOCATE_UNIQUE(lockname         IN VARCHAR2,
                          lockhandle      OUT VARCHAR2,
                          expiration_secs  IN INTEGER
                                  DEFAULT 864000);
```

表 3-4　DBMS_LOCK.ALLOCATE_UNIQUE プロシージャのパラメータ

パラメータ	説明
LOCKNAME	一意の ID を生成するロックの名前を指定してください。新しいロック名で ALLOCATE_UNIQUE をコールする最初のセッションによって、一意のロック ID が生成され、DBMS_LOCK_ALLOCATED 表に格納されます。この ID へのハンドルが、このコールと（通常は他のセッションによる）すべての後続するコールに戻されます。ALLOCATE_UNIQUE によって割り当てられるロック ID は、1073741824 ～ 1999999999 までの範囲に存在します。 ロック名の先頭は "ORA$" にしないでください。"ORA$" で始まる名前は、Oracle 提供の製品名だけに使用します。
LOCKHANDLE	ALLOCATE_UNIQUE によって生成されたロック ID へのハンドルを、コール元に戻します。このハンドルは、REQUEST、CONVERT、および RELEASE に対する次のコールで使用できます。LOCKHANDLE のデータ型は VARCHAR2(128) です。 実際のロック ID のかわりにハンドルを戻すのは、プログラミング・エラーのために不正な（しかし有効な）ロック ID が誤って作成される確率を少なくするためです。これによって、このパッケージを使用する複数のアプリケーションの分離が改善されます。 同じロック名を指定した ALLOCATE_UNIQUE によって戻されたロック・ハンドルを使用するセッションはすべて、同じロック名を参照します。セッションが異なると、同一のロックに対応し、異なるロック・ハンドルを持つことがあるので、1 つのセッションから別のセッションにロック・ハンドルを渡すのは避けてください。
EXPIRATION_SECS	所定のロックに対して ALLOCATE_UNIQUE を最後に実行してから、DBMS_LOCK_ALLOCATED 表 からそのロックの削除を許可するまでの時間を秒単位で指定してください。デフォルト の待機期間は 10 日です。この表からロックを削除してはなりません。引き続き ALLOCATE_UNIQUE をコールすると、期限を過ぎたロックを削除して、領域を回復できます。

REQUEST 関数

特定のモードでロックを要求するには、REQUEST 関数を使用します。REQUEST は、ユーザー定義のロック識別子、または ALLOCATE_UNIQUE プロシージャによって戻されるロック・ハンドルを受け入れるオーバーロード型関数です。

REQUEST 関数のパラメータの説明は、表 3-5 に記述され、この関数の戻り値とその意味は、表 3-6 に記述されています。この関数の構文は、次のとおりです。

```
DBMS_LOCK.REQUEST(id                IN INTEGER ||
                  lockhandle        IN VARCHAR2,
                  lockmode          IN INTEGER DEFAULT X_MODE,
                  timeout           IN INTEGER DEFAULT MAXWAIT,
                  release_on_commit IN  BOOLEAN DEFAULT FALSE,
RETURN INTEGER;
```

X_MODE および MAXWAIT のようなデフォルト値は、DBMS_LOCK パッケージ仕様部で定義されます。現行のデフォルト値は、オンラインで使用可能なパッケージ仕様部を参照してください。

表 3-5　DBMS_LOCK.REQUEST 関数のパラメータ

パラメータ	説明
ID または LOCKHANDLE	モードを変更するロックについて、ユーザーが割り当てた 0 ～ 1073741823 までのロック識別子、または ALLOCATE_UNIQUE によって戻されるロック・ハンドルを指定します。
LOCKMODE	ロックに対して要求するモードを指定します。利用できるモードとそれらに対応する整数識別子を次にリストします。V$ ビューと Enterprise Manager モニターの表示で使用されるロックの省略形をカッコ内に示します。 1 - NULL モード 2 - 行共有モード (ULRS) 3 - 行排他モード (ULRX) 4 - 共有モード（ULS） 5 - 行共有排他モード (ULRSX) 6 - 排他モード (ULX) それぞれのロック・モードについては、『Oracle8 Server 概要』で説明しています。
TIMEOUT	ロックを付与しようと継続する時間（単位は秒）を指定します。ロックをこの時間内で付与できないと値 1（タイムアウト）が戻されます。
RELEASE_ON_COMMIT	コミットまたはロールバックでロックを解除するには、このパラメータに TRUE を設定します。そうしないと、明示的に解除するか、あるいはセッションが終了するまでロックが保持されます。

表 3-6　DBMS_LOCK.REQUEST 関数戻り値

戻り値	説明
0	成功
1	タイムアウト
2	デッドロック
3	パラメータ・エラー
4	すでに所有しているロックが ID または LOCKHANDLE に指定されている
5	不正なロック・ハンドル

CONVERT 関数

あるモードから別のモードにロックを変換するには、CONVERT 関数を使用します。CONVERT は、ユーザー定義のロック識別子、または ALLOCATE_UNIQUE プロシージャによって戻されるロック・ハンドルを受け入れるオーバーロード型関数です。

CONVERT 関数のパラメータの説明は、表 3-7 に記述され、この関数の戻り値とその意味は、表 3-8 に記述されています。この関数の構文は、次のとおりです。

```
DBMS_LOCK.CONVERT(
        id         IN INTEGER ||
        lockhandle IN VARCHAR2,
        lockmode   IN INTEGER,
        timeout    IN NUMBER DEFAULT MAXWAIT)
RETURN INTEGER;
```

表 3-7　DBMS_LOCK.CONVERT 関数のパラメータ

パラメータ	説明
ID または LOCKHANDLE	モードを変更するロックについて、ユーザーが割り当てた 0 ～ 1073741823 までのロック識別子、または ALLOCATE_UNIQUE によって戻されるロック・ハンドルを指定します。
LOCKMODE	指定したロックに割り当てる新しいモードを指定します。利用できるモードとそれらに対応する整数識別子を次にリストします。V$ ビューと Enterprise Manager モニターの表示で使用されるロックの省略形をカッコ内に示します。 1 - NULL モード 2 - 行共有モード (ULRS) 3 - 行排他モード (ULRX) 4 - 共有モード（ULS） 5 - 行共有排他モード (ULRSX) 6 - 排他モード (ULX) それぞれのロック・モードについては、『Oracle8 Server 概要』で説明しています。
TIMEOUT	ロック・モードを変更しようと継続する時間（単位は秒）を指定します。ロックをこの時間内で変換できないと値 1（タイムアウト）が戻されます。

表 3-8　DBMS_LOCK.CONVERT 関数戻り値

戻り値	説明
0	成功
1	タイムアウト
2	デッドロック
3	パラメータ・エラー
4	所有していないロックが ID または LOCKHANDLE に指定されている
5	不正なロック・ハンドル

RELEASE 関数

REQUEST 関数を使用して獲得したロックを明示的に解除するには、RELEASE 関数を使用します。ロックはセッションが終了すると自動的に解除されます。RELEASE は、ユーザー定義のロック識別子、または ALLOCATE_UNIQUE プロシージャによって戻されるロック・ハンドルを受け入れるオーバーロード型関数です。

RELEASE 関数のパラメータの説明は、表 3-9 に記述され、この関数の戻り値とその意味は、表 3-10 に記述されています。この関数の構文は、次のとおりです。

```
DBMS_LOCK.RELEASE(id         IN INTEGER)
RETURN INTEGER;
DBMS_LOCK.RELEASE(lockhandle IN VARCHAR2)
RETURN INTEGER;
```

表 3-9　DBMS_LOCK.RELEASE 関数のパラメータ

パラメータ	説明
ID または LOCKHANDLE	解除するロックについて、ユーザーが割り当てた 0 ～ 1073741823 までのロック識別子、または、ALLOCATE_UNIQUE によって戻されるロック・ハンドルを指定します。

表 3-10　DBMS_LOCK.RELEASE 関数戻り値

戻り値	説明
0	成功
3	パラメータ・エラー
4	所有していないロックが ID または LOCKHANDLE に指定されている
5	不正なロック・ハンドル

SLEEP プロシージャ

特定の時間だけセッションを中断するには、SLEEP プロシージャを使用します。

SLEEP プロシージャのパラメータの説明は、表 3-11 に示してあります。SLEEP プロシージャの構文は、次のとおりです。

```
DBMS_LOCK.SLEEP(seconds  IN NUMBER);
```

表 3-11　DBMS_LOCK.SLEEP プロシージャのパラメータ

パラメータ	説明
SECONDS	セッションを中断する時間（単位は秒）を指定します。最小 100 分の 1 秒まで指定できます。たとえば、1.95 は有効な時間値です。

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。
- アプリケーション・レベルの読取りロックを施行する。
- ロックの解除を検出し、アプリケーションの終了後、クリーン・アップする。
- アプリケーションを同期させて、逐次処理を施行する。

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ことを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.
```

ロックを表示して監視する

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・モニターとラッチ・モニター)	Enterprise Manager の Monitor 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 モニターの詳しい説明は、『Oracle Server Manager ユーザーズ・ガイド』を参照してください。
`UTLLOCKT.SQL`	UTLLOCKT.SQL スクリプトは、ツリー構造で簡単なロック待機グラフを表示します。SQL スクリプトを実行する非定型 SQL ツール (SQL*Plus など) を使用して、システム内のロック待機中のセッションと、それらの対応するブロッキング・ロックを印刷します。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、スクリプト CATBLOCK.SQL を実行しなければなりません。）

直列可能トランザクションを使用した同時実行性の制御

デフォルトの場合、Oracle Server では、同時に実行するトランザクションは、同じ表および同じデータ・ブロック内で行の修正または追加、削除を行うことができます。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されません。

あるトランザクション（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 によりロックされている行を更新または削除しようとすると、トランザクション A の DML コマンドは、トランザクション B がコミットされるか、あるいはロールバックするまで、ブロックします。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、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該当します。しかし、トランザクションが直列可能になるようにした方が都合がよい場合もあります。直列可能トランザクションは、同時ではなく、（直列で）一度に 1 トランザクションずつ 実行しているように見える方法で実行しなければなりません。すなわち、逐次化モードで実行する同時トランザクションは、次から次へ実行するよう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 標準 SQL92 は、考えられる 3 種類のトランザクション相互作用と、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。これらの相互作用および分離レベルの要約を表 3-12 に示します。

表 3-12　ANSI 分離レベル

分離レベル	内容を保証しない読取り (1)	反復不能な読取り (2)	仮読取り (3)
READ UNCOMMITTED	可能	可能	可能
READ COMMITTED	不可能	可能	可能
REPEATABLE READ	不可能	不可能	可能
SERIALIZABLE	不可能	不可能	不可能

注意 :

(1) トランザクションは、別のトランザクションで変更された、コミットされていないデータを読み取ることができます。

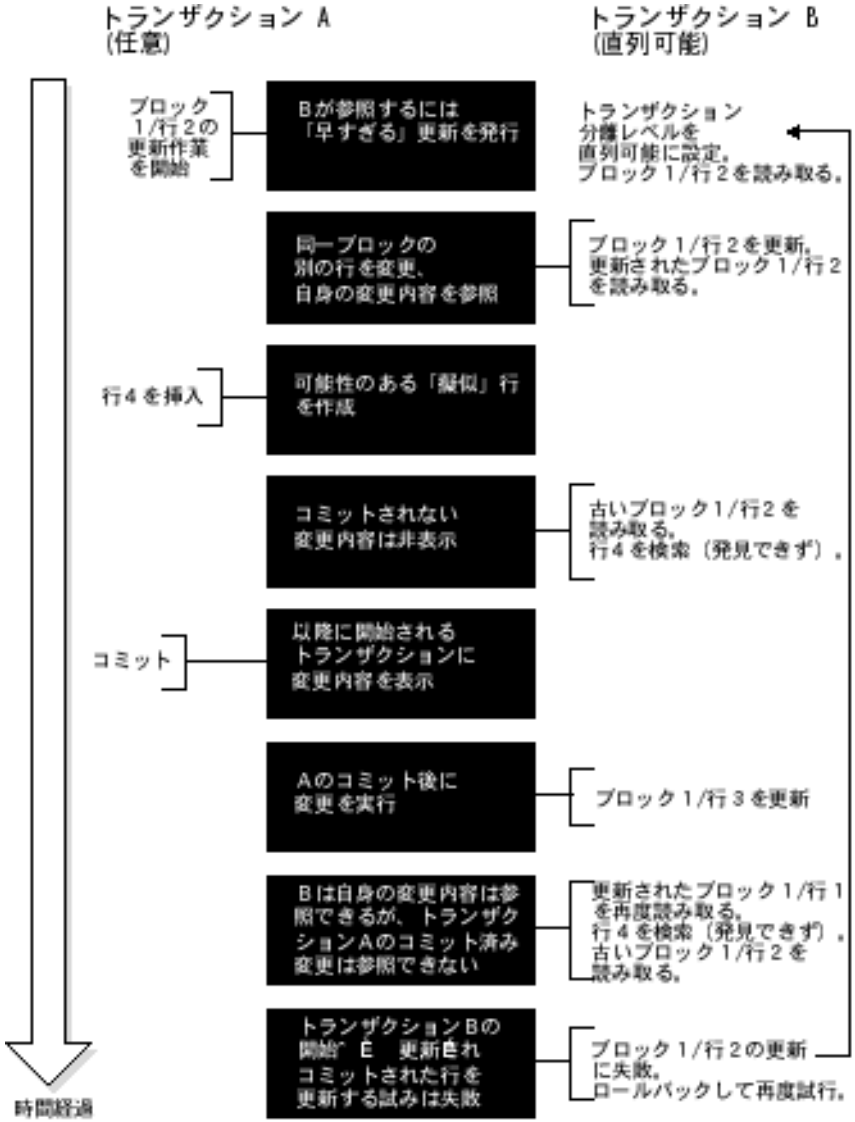
(2) トランザクションは、別のトランザクションでコミットされたデータを再度読み取り、その新しいデータを参照します。

(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見つけることができます。

これらの分離レベルに関する Oracle の動作について、次に要約します。

`READ UNCOMMITTED`	Oracle では「内容を保証しない読取り」は許可されません。これは、高スループットの Oracle では不要です。
`READ COMMITTED`	Oracle は、READ COMMITTED 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）でコミットされたデータだけを参照するため、Oracle は、READ COMMITTED 分離について実際に ANSI/ISO SQL92 標準に必要である以上の整合性を提供します。
`REPEATABLE READ`	Oracle は、SERIALIZABLE によって提供されるもの以外は、この分離レベルをサポートしていません。
`SERIALIZABLE`	この分離レベルは、3-29 ページに示したとおり、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 3-2　2 つのトランザクションの時系列的働き



直列可能トランザクションの相互作用

図 3–2 は、直列可能トランザクション (トランザクション B) が別のトランザクション (トランザクション A、これは SERIALIZABLE でも READ COMMITTED でもかまいません) とやりとりする方法を示しています。

直列可能トランザクションが ORA-08177 エラー（「アクセスを逐次化できません」）で失敗したとき、当該アプリケーションは、次のアクションのいずれかを行う可能性があります。

- そのポイントに対して実行された作業のコミット
- 追加の、様々な文の実行。この実行は、当該トランザクション内の直前のセーブポイントにロールバックした後であると考えられます。
- 当該トランザクション全体のロールバックおよびその再試行

Oracle 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。SERIALIZABLE 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、この制御情報の記憶域を取り消す必要があります。直列可能モードを使用するには、INITRANS を少なくとも 3 に設定しなければなりません。

分離レベルを設定する

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 コマンドがトランザクションで最初に 発行されるコマンドでなければなりません。これ以外の場合、次のエラーが出力されます。

`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。

ALTER SESSION コマンドを使用して、セッション内でのトランザクション分離レベルを設定します。

> **関連項目：** SET TRANSACTION コマンドと ALTER SESSION コマンドの完全な構文については、『Oracle8 Server SQL リファレンス』を参照してください。

INITRANS パラメータ

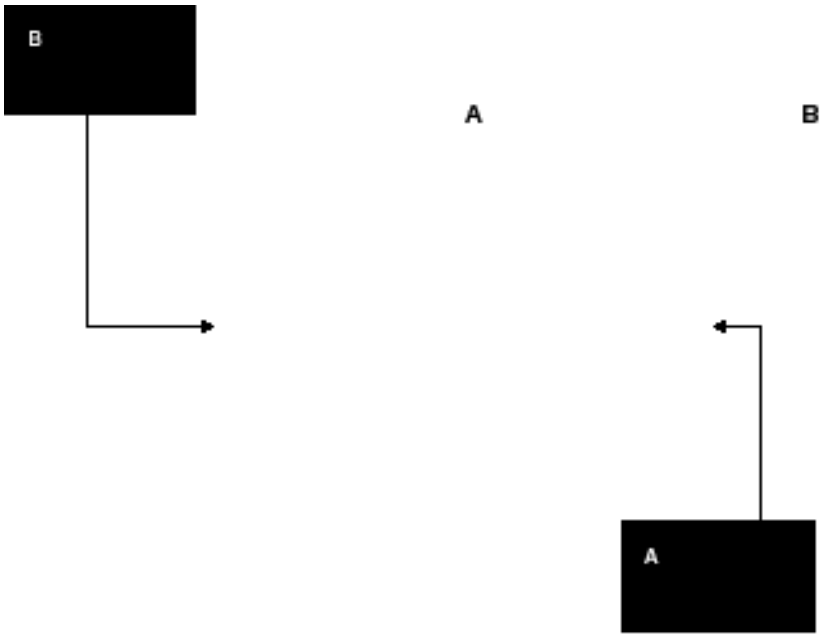
Oracle は、同時トランザクションによるアクセスを管理するために、各データ・ブロックに制御情報を格納します。したがって、トランザクション分離レベルを直列可能に設定する場合は、ALTER TABLE コマンドを使用して、INITRANS を少なくとも 3 に設定する必要があります。このパラメータを使用すると、Oracle は、各ブロックにアクセスした最新トランザクションの履歴を記録するのに十分な記憶域を、それぞれのブロック内に割り振ります。同じブロックを更新するトランザクションの数が多い表には、さらに大きな値を使用しなければなりません。

参照整合性と直列可能トランザクション

Oracle は SERIALIZABLE トランザクション内であっても読取り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取られたデータが、別のトランザクションで上書きされることがあります。アプリケーション・レベルでデータベースの整合性検査を実行するトランザクションでは、読み取ったデータがトランザクション実行中に変更されないという前提に立ってはいけません（そのような変更がトランザクションからわからないとしても）。SERIALIZABLE 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・レベルの整合性検査のプログラム・コードは十分注意して書かないと、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、READ COMMITTED トランザクションと SERIALIZABLE トランザクションの両方に当てはまるので注意してください。

図 3-3 は、2 つの表の間の参照整合性の親子関係を維持するためにアプリケーション・レベルの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取り、特定の主キー値を持つ行が存在するか判別します。もう一方のトランザクションは、親行の削除を行う前に、対応する細目行が存在しないかチェックします。この場合、両方のトランザクションが読み取ったデータは、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 3-3　参照整合性チェック



トランザクション A の実行した読取りのために、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もありません。そのため、上記のシナリオの場合、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取ったデータに対する変更を他方が防げないため、両トランザクションが SERIALIZABLE トランザクションであっても、上記の不整合は発生します。

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、アプリケーションの開発者は、一方のトランザクションで読み取ら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、特に注意する必要があります。これには、SQL92 SERIALIZABLE モードで定義されたトランザクション分離レベルよりも高いレベルが必要です。

SELECT FOR UPDATE を使用する

幸いなことに、Oracle では上記の例外が発生するの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、それによりトランザクション 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獲得しないように

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、上記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実現できます。たとえば、子表への INSERT により、PRE-INSERT 行レベル・トリガーを発動して、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、親行が（存在する場合）データベース内に残っていることを確認します。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、そのトリガーを発動させた SQL 文のコンテキスト内で実行します。1 つのトリガーで実行されるすべての SQL 文に見られるデータベースの状態は、トリガー発動文から見たデータベースの状態と同じです。そのため、READ COMMITTED トランザクションでは、トリガーに伴う SQL 文には、データベースがトリガー文の実行開始時点のデータベースのように見え、SERIALIZABLE モードで実行するトランザクションでは、SQL 文には、データベースがそのトランザクションの開始時点のデータベースのように見えます。いずれの場合も、トリガーで SELECT FOR UPDATE を使用すると、上記のとおり参照整合性が正しく実現されます。

READ COMMITTED 分離と SERIALIZABLE 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から 1 つを 選択できます。READ COMMITTED 分離レベルと SERIALIZABLE 分離レベルは両方とも高度な一貫性と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的な行レベル・ロックのインプリメンテーションによる競合状態の軽減という利点を提供するもので、実社会のアプリケーション開発に合わせて設計されています。この項の残りの部分では、上記の 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提供します。

トランザクション集合の整合性

Oracle の READ COMMITTED 分離レベルと SERIALIZABLE 分離レベルについて説明するには、次のことを考えてみるとよいでしょう。

- データベースの表の集合（または任意のデータ集合）
- それらの表内の行の特定の読取り順序
- 特定の時刻（任意）にコミットされたトランザクションの集合

ある操作（問合せまたはトランザクション）が、どの読取りでも、同じ集合に属するコミットされたトランザクションによって書き込まれたデータが戻る場合には、「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取りもあれば、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取りもあります。トランザクション集合不整合である操作には、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。

READ COMMITTED モードで実行するトランザクションは、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取られた行はすべて、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 SERIALIZABLE モードでは、SERIALIZABLE 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、READ COMMITTED 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だけしか見えない ため、トランザクション集合が整合していません。このことは、たとえば、細目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細目は見えない（その逆もまた同じである）ことを意味します。 Oracle の READ COMMITTED モードでは、このような影響を受けないので、読取りロック・システムよりも高い整合性が得られます。

読取りロック・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 分離により、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。仮プロテクション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。これらのシステムにおける、スループットを制限し、デッドロックされやすい SERIALIZABLE モードの場合だけ、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。

機能比較の要約

表 3-13 に、READ COMMITTED トランザクションと SERIALIZABLE トランザクションとの間の主な類似点と相違点を要約します。

表 3-13　コミットされたトランザクション対直列可能トランザクションの読取り

	Read Committed	Serializable
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取り	不可能	不可能
反復不能な読取り	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取り	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

表 3-13　コミットされたトランザクション対直列可能トランザクションの読取り

	Read Committed	Serializable
読取りブロック書込み機能	いいえ	いいえ
書込みブロック読取り機能	いいえ	いいえ
異なる行の書込みブロック書込み機能	いいえ	いいえ
同じ行の書込みブロック書込み機能	はい	はい
ブロック化トランザクションの待機	はい	はい
「アクセスを逐次化できない」エラーが発生しやすい	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 それぞれ別個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割合と応答時間の要求に関するトランザクション・パフォーマンス要件を査定して、パフォーマンスの予想を満足させながら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際、整合性と同時実行性との間でのトレードオフが必要です（トランザクションのスループット）。

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り、高レベルの整合性と同時実行性（およびパフォーマンス）を提供します。Oracle では読取りと書込みの相互干渉がないため、問合せで整合性のあるデータが参照できる一方、READ COMMITTED 分離と SERIALIZABLE 分離により、コミットされていない (「内容を保証しない」) データの読取りを防止し、高レベルの同時実行性を提供することで高パフォーマンスを実現しています。

READ COMMITTED 分離レベルでは、一部のトランザクションについては（仮読取りおよび反復不能読取りのため）矛盾のある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。SERIALIZABLE 分離レベルの場合は、仮読取りおよび反復不能読取りから保護されているため、より高い整合性が提供され、読取り / 書込みトランザクションが問合せを 2 回以上実行する場合にはこの分離レベルは重要です。しかし、SERIALIZABLE モードでは、アプリケーションは、" アクセスを逐次化できない " エラーの有無をチェックする必要があるので、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します。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論理は、いずれのモードでも読取りが書込みをブロックしないという事実を考慮しなければなりません。

アプリケーションのヒント

トランザクションが直列可能モードで実行するとき、直列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って変更されたデータを変更しようとすると、結果として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、適切なアクションは、現在のトランザクションをロールバックし、それを再度実行することです。ロールバックの後、当該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獲得し、DML 操作は成功すると思われます。

当該トランザクションのロールバックと反復が必要であるため、他の同時トランザクションと矛盾する可能性のある DML 文を可能な限り、トランザクションの始めの方へ入れると、開発がうまくいきます。

4

スキーマ・オブジェクトの管理

この章では、ユーザーのスキーマ内に異なるタイプのオブジェクトを作成し、それらを管理するために必要な手順について説明します。この章を構成しているトピックは、次のとおりです。

- 表を管理する
- ビューを管理する
- 結合ビューを修正する
- 順序を管理する
- シノニムを管理する
- 索引を管理する
- クラスタ、クラスタ化した表、クラスタ索引を管理する
- ハッシュ・クラスタとクラスタ化した表を管理する
- スキーマ・オブジェクトの管理について

関連項目： 以下の情報については、次の各章およびマニュアルで説明しています。

- プロシージャ、関数、パッケージ ― 第 10 章
- オブジェクト型 ― 第 7 章
- 依存性情報 ― 第 15 章
- 対称レプリケーションを使用している場合は、『Oracle8 Server レプリケーション』を参照して、スナップショットなどのスキーマ・オブジェクトの管理について詳しい情報を得てください。
- Trusted Oracle を使用している場合、スキーマ・オブジェクトを管理するときにさらに必要になる権限と検討すべき問題がいくつかあります。Trusted Oracle については、Trusted Oracle のマニュアルを参照してください。

表を管理する

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。表は行と列で構成されます。

1 つの表を使用して、システムの中で記録する 1 つのエンティティを表現します。このような表の例として、従業員リストや製品の受注リストが挙げられます。

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や注文と製品の関連がその例です。表内では、関係を表現するのに外部キーが使用されます。

的確に設計すれば、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。しかし、多くの表はエンティティまたは関連の一方だけを表現します。たとえば従業員（EMP）表は従業員について記述します。この表には部門番号（DEPTNO）という外部キー列も含まれていて、これによって部門に対する従業員の関連が表現されます。

次の項では、表を作成、変更、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に参考にする簡単なガイドラインもいくつか示します。より具体的な説明は『Oracle8 Server 管理者ガイド』を参照してください。リレーショナル・データベースと表設計について説明している書籍も参照するとよいでしょう。

表を設計する

表を設計するときには、次のガイドラインを考慮してください。

- 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する。
- 表名や列名に対して省略形や単数形、複数形を使用する場合には首尾一貫したものにする。
- COMMENT コマンドを使用してそれぞれの表や列の意味を文書化する。
- 各表を正規化する。
- 各列に適したデータ型を選択する。
- NULL を許す列を最後に定義して、記憶領域を節約する。
- 適切なとき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する。

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するとよいでしょう。整合性制約を表の列に定義して、データベースについてのビジネス・ルールを自動的に試行できます。ガイドラインについては、第 9 章「データの整合性の保持」を参照してください。

表を作成する

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、クラスタ化しない表 EMP が SCOTT のスキーマに作成され

ます。物理的には表 EMP 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 (
   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)
   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を管理する

以降の項では、PCTFREE パラメータと PCTUSED パラメータを次のような目的で使用する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と検索のパフォーマンスを向上させる。
- データ・ブロック内の利用されていない領域を少なくする。
- データ・ブロック間の行連鎖を少なくする。

PCTFREE を指定する

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 までの整数を指定できます。（PCTFREE が 99 に設定されると、行サイズとは関係なく、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、より多くの行が収めら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する領域が小さくなる。
- 挿入によりブロックがより満杯になる。
- 表または索引の全体のデータを格納するブロックが少なくなる（ブロックあたりの行またはエントリは多くなる）ので、領域が節約される。
- ブロックは、その空き領域が新しいデータまたは更新データで満杯になるように、頻繁に再編成する必要があるため、処理コストが増大する。

- 行または索引エントリの更新によって行が大きくなってブロックにまたがる場合、処理コストと必要な領域が増大する。（UPDATE 文および DELETE 文、SELECT 文は、与えられた行に対してより多くのブロックを読み込む必要があり、連鎖された行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する領域が大きくなる。
- 同じ量の挿入データに必要なブロックが増える（ブロックあたりに挿入する行は少なくなる）。
- ブロックは空き領域の再編成が必要となることはほとんどないため、処理コストは小さくなる。
- Oracle は行断片を頻繁に連鎖しないようになるため、更新パフォーマンスが改善される。

PCTFREE の設定では、表データや索引データの特性を理解すべきです。更新によって行が長くなることがあります。NUMBER または VARCHAR2、LONG、LONG RAW を使用している場合、新しい値は、これらが置き換える値と同じサイズにならない場合があります。データ値を長くす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、満杯のブロック）と優れた更新パフォーマンス（高い PCTFREE、余裕のあるブロック）の間で満足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る、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を及ぼします。読取り一貫性を保証することによって、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。

クラスタ化しない表の PCTFREE　クラスタ化しない表の行のデータ・サイズが増えそうな場合、これらの更新に備えてある程度領域を確保してください。更新に備えて領域を確保しておかなければ、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。

クラスタ化した表の PCTFREE　クラスタ化しない表の説明は、クラスタ化し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・キーに連鎖される新しいデータ・ブロックに収まります。

索引の PCTFREE　索引データの更新のために、索引が空き領域を使用することがまれにあります。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、通常は非常に小さな値になります（たとえば、5 以下）。

PCTUSED を指定する

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、新しい行はいっさいそのブロックには挿入されません。

Oracle は、最低でも PCTUSED を満杯にするようにデータ・ブロックを維持しようとします。その割合は、全体の領域からオーバーヘッドを引いた後の、データに対して利用可能なブロック領域です。

PCTUSED のデフォルト値は 40 パーセントです。PCTUSED と PCTFREE の合計が 100 を超えない限り、0 から 99 まで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、PCTUSED の値がより大きいときよりもブロックを余裕のある状態に保つ。
- その使用の割合よりも低くなったときに、ブロックの空きリストへ移動するために UPDATE 文と DELETE 文の間に必要な処理コストを削減する。
- データベース内の利用されていない領域を増やす。

PCTUSED を大きくすると、次のようになります。

- 通常、PCTUSED の値がより小さいときよりもブロックを満杯の状態に保つ。
- 領域効率を改善する。
- INSERT と UPDATE の間の処理コストを増やす。

関連する PCTUSED と PCTFREE の値を選択する

PCTFREE と 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 以下でなければならない。
- 合計が 100 より少ない場合、領域使用と I/O パフォーマンスの理想的な妥協点は、平均の行が占有する利用可能なブロック内の領域の割合だけ 100 から引いて、その値が PCTFREE と PCTUSED の合計になることである。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、データに対して利用できるとします。平均の行が 195 バイト、または 1948 の 10% を必要とする場合、合計して 90% になる PCTUSED と PCTFREE の適切な組合せが、データベース領域を最もよい状態で使用します。
- 合計が 100 に等しい場合、Oracle は、空き領域が PCTFREE を超えないようにする。そして処理コストは最も高くなります。
- 固定ブロックのオーバーヘッドは、PCTUSED または PCTFREE の計算に含めない。
- 100 と PCTFREE と PCTUSED の合計の間の差が小さいほど（PCTUSED が 75、PCTFREE が 20 の場合など）、いくつかのパフォーマンス・コストで領域使用の効率がよくなる。

PCTFREE と PCTUSED の値を選択する例

与えられたシナリオで、PCTFREE と PCTUSED に対して値を指定する具体的な例を次に示します。

例 1

シナリオ :	一般的なアクティビティは、行のサイズを増やす UPDATE 文を含みます。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、サイズが増える行に対して十分な領域を与えます。領域要求が高い更新アクティビティの間に行われる処理が少なくなるように、PCTUSED を 40 に設定して、パフォーマンスを改善します。

例 2

シナリオ :	多くのアクティビティは、影響を受ける行のサイズを増やさない INSERT 文、DELETE 文、UPDATE 文を含みます。 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 文は行サイズを増やさないため、PCTFREE を 5 に設定します。DELETE 文によって解放される領域が比較的すぐに使用されるように、PCTUSED を 60 に設定し、処理量も最小化します。

例 3

シナリオ :	表が非常に大きいため、記憶域について第 1 に考慮しなければなりません。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す。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、表のデータを格納するのに必要なデータ・ブロックの数が減少し、問合せのために走査するデータ・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。

表を作成するために必要な権限

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 システム権限を持ってい

なければなりません。さらに、表の所有者は、その表を含む表領域に対する割当て制限、または UNLIMITED TABLESPACE システム権限を持っていなければなりません。

表を変更する

次のような理由で Oracle データベース内の表を変更することもできます。

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

表にレコードが存在していない限り、表の列定義を変更して、既存の列長さを大きくできます。また、空の表の列長さを小さくすることもできます。列のデータ型が CHAR である場合、特に表に数多くの行が含まれている場合、列の長さを拡大することは、相当な追加の記憶を必要とし、時間を浪費する操作となる場合があります。これは、各行の CHAR 値に新しい列長さになるまで、空白が埋め込まれるからです。

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わりません。しかし、空白埋込みが必要なので、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 を使用してください。例を示します。

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

表を変更することは、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、最初、列は NULL となる。表が行を含まない場合だけ、NOT NULL 制約付きの列をその表に追加できる。
- ビュー、または PL/SQL プログラム・ユニットが実表に依存する場合、実表を変更すると、依存オブジェクトに影響が及ぶ場合があり、依存オブジェクトは必ず無効になる。

表を変更するために必要な権限

表を変更するには、その表が使用するスキーマに含まれているか、あるいはその表に対する 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なければなりません。

表を削除する

表を削除するには、SQL コマンド DROP TABLE を使用してください。たとえば、次の文は EMP 表を削除します。

```
DROP TABLE emp;
```

削除する表に、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 FOREIGN KEY 制約を削除するのであれば、次のように DROP TABLE コマンドに CASCADE オプションを指定してください。

```
DROP TABLE emp CASCADE CONSTRAINTS;
```

表を削除する影響は、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る。その表のすべての行はアクセスできなくなる。
- 表に対応付けられた索引とトリガーはすべて削除される。
- 削除された表に依存していたビューと PL/SQL プログラム・ユニットはすべてそのまま残るが、無効になる（使用できない）。
- 削除された表に対するシノニムはすべてそのまま残るが、使用するとエラーを戻す。
- 削除された表がクラスタ化していない表の場合、割り当てられていたすべてのエクステントが、表領域の空き領域に戻され、新しいエクステントを必要とする他のオブジェクトによってその領域を使用できる。
- クラスタ化した表に対応する行はすべて、そのクラスタのブロックから削除される。
- そのテーブルがスナップショットのマスター表である場合は、そのスナップショットは削除されないが、スナップショットのログは削除される。スナップショットは依然として使用できるが、表を再作成しない限り、リフレッシュすることはできません。

表の行をすべて削除する場合、表の定義を残すのであれば、TRUNCATE TABLE コマンドを使用すべきです。このコマンドについては、『Oracle8 Server 管理者ガイド』で説明しています。

表を削除するために必要な権限

表を削除するには、その表が使用するスキーマに含まれているか、あるいは DROP ANY TABLE システム権限を持っていなければなりません。

ビューを管理する

ビューは、別の表または複数の表の組合せを論理的に表示するものです。ビューはベースになる表からデータを導出します。これらの表は、実表と呼ばれます。実表は、表でもビューそのものでもかまいません。

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

ビューによって、他の表やビューに存在するデータを別の外観（たとえば、サブセットやスーパーセット）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えるためにビューは威力を発揮します。

以降の項では、SQL コマンドを使用して、ビューを作成し、置き換え、削除する方法について説明します。

ビューを作成する

ビューを作成するには、SQL コマンド CREATE VIEW を使用してください。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 句や FOR UPDATE 句を含むことはできません。たとえば、次の文は、EMP 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
    WHERE deptno = 10
    WITH CHECK OPTION CONSTRAINT sales_staff_cnst;
```

SALES_STAFF ビューを定義する問合せは、部門 10 の行だけを参照します。さらに、WITH CHECK OPTION は、ビューに対して発行された INSERT 文と UPDATE 文が、ビューが選択できない行を作成したり、結果とすることを許可しない制約付きのビューを作成します。

上の例で考えると、次に示す INSERT 文は、SALES_STAFF ビューを介して EMP 表に行を挿入します。

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

しかし、次の INSERT 文は、SALES_STAFF ビューを使用して、選択できない部門番号 30 の行を挿入しようとするため、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 表と DEPT 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, dept
    WHERE emp.deptno IN (10, 30)
```

```
AND emp.deptno = dept.deptno;
```

DIVISION1_STAFF ビューは、EMP 表と DEPT 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や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。4-13 ページおよび 4-14 ページ以降を参照してください。

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に、Oracle は ANSI/ISO 標準に従って、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いる場合、問合せを構文上正しくするには引用符が必要となることを考慮して、このような処理を行います。

DEPT ビューが次の文で作成される例を考えます。

```
CREATE VIEW dept AS SELECT * FROM scott.dept;
```

Oracle は、DEPT 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
```

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。しかし、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を作成する

構文エラーがないと想定すると、ビューを定義している問合せを実行できなくても、（エラーを伴う）ビューを作成できます。たとえば、ビューが存在しない表や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 ...;
```

ビューがエラーを伴って作成されるとき、Oracle はビューがエラーを伴って作成されたということを示すメッセージを戻します。このようなビューの状態は INVALID として記録されます。その後状況が変化して、無効なビューの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。

ビューを作成するために必要な権限

ビューを作成するには、次の権限が付与されていなければなりません。

- 自分のスキーマにビューを作成するには、CREATE VIEW システム権限を持っていなければならない。または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 システム権限を持っていなければなりません。これらの権限は、明示的に獲得されるか、あるいはロールを介して獲得されます。
- ビューの**所有者**は、ビューの定義で参照する全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なければならない。ただし、ロールを通して必要な権限を獲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ています。たとえば、Scott の EMP 表に対して INSERT 権限だけを付与されている場合、Scott の EMP 表に対してビューを作成することはできます。しかし、このビューは、EMP 表に新しい行を挿入するために使用できるだけです。
- ビューの所有者は、ビューにアクセスする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を持っていなければならない。そうでないと、他のユーザーにビューに対するアクセス権を付与する権限が不十分です。

ビューを置き換える

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、ビューを置き換えなければなりません。

- ビューは、削除してから、再作成することができる。ビューが削除されるとき、対応するビュー権限の付与がすべて、ロールと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しなければなりません。
- 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、ビューを置き換えることができる。このオプションは、ビューの現行の定義は置き換えても、現在のセキュリティ許可はそのまま残す場合に使用されます。たとえば、先に示した例のように、SALES_STAFF ビューを作成すると想定します。また、ロールと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。しかし、部門番号は 30 にすべきであるため、定義している問合せの WHERE 句に指定されている部門番号を訂正するために、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、次の文によって、SALES_STAFF ビューの現行バージョンを置き換えることができます。

```
CREATE OR REPLACE VIEW sales_staff AS
    SELECT empno, ename, deptno
    FROM emp
    WHERE deptno = 30
    WITH CHECK OPTION CONSTRAINT sales_staff_cnst;
```

ビューを置き換える影響は、次のとおりです。

- ビューの置換えは、データ・ディクショナリ内のビューの定義を置き換える。ビューによって参照される、基礎となるすべてのオブジェクトは影響を受けません。
- 前に定義されたが、新しいビュー定義に含められなかった場合、ビュー定義の WITH CHECK OPTION に関連付けられている制約が削除される。

- 置き換えられたビューに依存するビューと PL/SQL プログラム・ユニットはすべて、無効になる。

ビューを置き換えるために必要な権限

ビューを置き換えるには、ビューを作成するために必要なすべての権限と、ビューを削除するために必要なすべての権限が必要です。

ビューを使用する

表と同じ方法でビューを問い合せることができます。たとえば、DIVISION1_STAFF ビューを問い合せるには、そのビューを参照する有効な SELECT 文を入力してください。

```
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
```

いくつかの制限がありますが、ビューを使用して、実表に対して、行を挿入、更新、削除できます。次の文は、SALES_STAFF ビューを使用して、EMP 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 操作制限は、次の基準をリストされている順序で使用します。

1. ビューが、SET 演算子、または DISTINCT 演算子、GROUP BY 句、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実表に対して、行を挿入、更新、削除することはできません。
2. ビューが、WITH CHECK OPTION 付きで定義されている場合、そのビューが、実表から行を選択できなければ、実表に対して、（ビューを使用して）行を挿入、更新することはできません。
3. ビューから DEFAULT 句を持たない NOT NULL 列が省略されている場合、そのビューを使用して実表に行を挿入することはできません。
4. ビューが、DECODE(deptno, 10, "SALES", ...) のような式を使用して作成された場合、そのビューを使用して実表に行を挿入したり、実表の行を更新したり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 表で部門番号が 10 の行だけを挿入したり、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と仮定するとします（つまり DEPTNO 列を除外しています）。

```
CREATE VIEW sales_staff AS
    SELECT empno, ename
    FROM emp
    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 フィールドと ENAME フィールドを更新できます。しかし、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 表に行を挿入することはできません。DEPTNO フィールドの DEFAULT 値として 10 が設定されていれば、挿入を実行できます。

無効なビューを参照する　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、基礎を形成す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。

ビューを使用するために必要な権限

ビューに対して、問合せまたは INSERT 文、UPDATE 文、DELETE 文を発行するには、そのビューに対する SELECT、INSERT、UPDATE、DELETE の各オフ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持っていなければなりません。

ビューを削除する

ビューを削除するには、SQL コマンドの DROP VIEW を使用します。

```
DROP VIEW sales_staff;
```

ビューを削除するために必要な権限

使用する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 システム権限を持っていなければなりません。

結合ビューを修正する

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修正できます。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;
```

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 実表が変更され、EMP 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とします。

```
CREATE VIEW emp_dept AS
  SELECT e.empno, e.ename, e.deptno, d.dname, d.loc
    FROM emp e, dept d    /* 結合操作 */
     WHERE e.deptno = d.deptno
       AND d.loc IN ('DALLAS', 'NEW YORK', 'BOSTON');
```

このビューを介して EMP 実表または DEPT 実表を変更するのには制限があります。これはたとえば、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';
```

修正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
- 集約関数： AVG または COUNT、GLB、MAX、MIN、STDDEV、SUM、VARIANCE
- 設定操作： UNION、UNION、ALL、INTERSECT、MINUS
- GROUP BY 句または HAVING 句
- START WITH 句または CONNECT BY 句
- ROWNUM 擬似列

結合ビューが修正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なければならないということがあります。マージ可能なビューの詳細は、『Oracle8 Server 概要』を参照してください。

表の例

この項に示されている例では、他でも例示している EMP 表と DEPT 表を使用しています。しかし、これらの例は、表で主キーと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ければ役に立ちません。EMP および DEPT のための適切な制約付き表定義を、次に示します。

```
CREATE TABLE dept (
```

```
        deptno   NUMBER(4) PRIMARY KEY,
        dname    VARCHAR2(14),
        loc      VARCHAR2(13));

CREATE TABLE emp (
        empno NUMBER(4) PRIMARY KEY,
        ename    VARCHAR2(10),
        job      varchar2(9),
        mgr      NUMBER(4),
        hiredate DATE,
        sal      NUMBER(7,2),
        comm     NUMBER(7,2),
        deptno   NUMBER(2),
FOREIGN KEY (DEPTNO) REFERENCES DEPT(DEPTNO));.
```

上記の主キーおよび外部キーの制約を省略し、DEPT(DEPTNO) に UNIQUE INDEX を作成して、以下に示す例を使用できます。

キー保存された表

キー保存された表という概念は、結合ビューの修正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、キー保存されます。したがって、キー保存された表では、結合によりそのキーを保存します。

注意：

- ある表をキー保存された表にするために、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、むしろ、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 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 と DEPT とを結合すれば DEPT.DEPTNO は一意になりますが、DEPT はキー保存された表にはなりません。

4-14 ページの「結合ビューを修正する」で指定されている 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 表のキーであり、この結合の結果のキーでもあるため、EMP はキー保存された表です。DEPTNO は、DEPT 表のキーですが、結合のキーではないため、キー保存された表ではありません。

結合ビューの DML 文の規則

結合ビューの UPDATE 文または INSERT 文、DELETE 文はいずれも、基礎を形成する実表を 1 つしか修正できません。

UPDATE 文

次の例は、EMP_DEPT ビュー (4-14 ページで示されている) を正常に修正する UPDATE 文です。

```
UPDATE emp_dept
  SET sal = sal * 1.10
    WHERE deptno = 10;
```

次の UPDATE 文は、EMP_DEPT ビューでは使用できません。

```
UPDATE emp_dept
  SET loc = 'BOSTON'
    WHERE ename = 'SMITH';
```

この文は、基礎を形成する DEPT 表（EMP_DEPT ビューではキー保存されていません）を修正しようとしたため、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、結合ビューの修正可能な列はすべて、キー保存された表の列にマップしなければなりません。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修正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 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
    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、失敗します。

DELETE 文

結合内でキー保存された表が 1 つだけある場合は、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
    WHERE ename = 'SMITH';
```

EMP_DEPT ビュー上のこの DELETE 文は、基礎となる EMP 表で DELETE 操作に変換することができ、しかもこの EMP 表は結合内の唯一のキー保存された表であるため、有効です。

次のビューでは、E1 と E2 は両方ともキー保存された表であるため、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, deptno
        FROM emp e1, emp e2
        WHERE e1.empno = e2.empno;
```

WITH CHECK OPTION 句を指定してビューを定義し、キー保存された表が反復する場合、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 e1, emp e2
            WHERE e1.mgr = e2.empno
            WITH CHECK OPTION;
```

このビューは、キー保存された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 ビューで次の INSERT 文を使用すると、成功します。

```
INSERT INTO emp_dept (ename, empno, deptno)
    VALUES ('KURODA', 9010, 40);
```

これは、修正するキー保存された表は 1 つだけ（EMP）であり、40 は DEPT 表で有効な DEPTNO である（したがって、EMP 表の FOREIGN KEY 整合性制約に合致します）ためです。

次のような INSERT 文は、基礎となる EMP 表の UPDATE が失敗するのと同じ理由で失敗します。

```
INSERT INTO emp_dept (ename, empno, deptno)
    VALUES ('KURODA', 9010, 77);
```

失敗する理由は、EMP 表の FOREIGN KEY 整合制約に違反するためです。

次のような INSERT 文は、基礎となる EMP 表の UPDATE が失敗するのと同じ理由で失敗します。

```
INSERT INTO emp_dept (empno, ename, loc)
    VALUES (9010, 'KURODA', 'BOSTON');
```

ここでは、ORA-01776 エラー (「結合ビューを介して複数の基本表を変更できません。」) が発生します。

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 句を使用して結合ビューを定義すると、そのビューに対して INSERT を実行することはできません。

UPDATABLE_COLUMNS ビューを使用する

結合ビューの修正に使用できる 3 つのビューを表 4–1 に示します。

表 4–1　UPDATABLE_COLUMNS ビュー

ビュー名	説明
USER_UPDATABLE_COLUMNS	ユーザーの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修正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修正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修正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修正可能である場合があります。たとえば、次のビューは外部結合を含みます。

```
CREATE VIEW emp_dept_oj1 AS
    SELECT empno, ename, e.deptno, dname, loc
    FROM emp e, dept d
    WHERE e.deptno = d.deptno (+);
```

次の文の場合、

```
SELECT * FROM emp_dept_oj1;
```

次のような結果となります。

```
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 は結合内のキー保存された表であるため、EMP_DEPT_OJ1 の基礎 EMP 表内の列は修正可能です。

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 e, dept d
    WHERE e.deptno (+) = d.deptno;
```

次の文の場合、

```
SELECT * FROM emp_dept_oj2;
```

次のような結果となります。

```
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（上記 SELECT 内の最後の行）、EMP はすでにキー保存された表ではありません。そのため、UPDATE、DELETE、INSERT の各操作をこのビューで実行することはできません。

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次々にマージされ、そのマージが一番上まで達していなければなりません。外部結合されるビューは、「単純」である場合にだけ、この時点でマージされます。たとえば、次のとおりです。

```
SELECT col1, col2, ... FROM T;
```

つまり、このビューの選択リストには式はなく、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 emp_v AS
```

```
    SELECT empno, ename, deptno
        FROM emp;

CREATE VIEW emp_dept_oj1 AS
    SELECT e.*, loc, d.dname
        FROM emp_v e, dept d
            WHERE e.deptno = d.deptno (+);
```

上記の例では、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 はキー保存された表です。しかし、EMP_V が次のように変更された場合、

```
CREATE emp_v_2 AS
    SELECT empno, ename, deptno
        FROM emp
            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 は、すでにキー保存された表ではありません。

ビューが修正可能であるかどうか不明な場合、ビュー USER_UPDATABLE_COLUMNS から選択（SELECT）して確認できます。たとえば、次のとおりです。

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

次のような結果が戻されます。

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

順序を管理する

順序生成機能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、複数の行や表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のみ生成できます。新しい主キーの値は、最も最近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、トランザクション中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しなければなりません。このような待機を「直列化（serialization ）」と呼びます。アプリケーションがこのような構成になっている場合、順序にアクセスするように変更するべきです。順序によって、直列化がなくなり、アプリケーションの同時実行性が改善されます。

以降の項では、SQL コマンドを使用して、順序を作成、使用、変更、削除する方法を説明します。

順序を作成する

順序を作成するには、SQL コマンド CREATE SEQUENCE を使用してください。次の文は、EMP 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;
```

いくつかのパラメータは、順序の機能を制御するために指定でき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、最大値と最小値、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。

Oracle Parallel Server を使用するときの順序番号のキャッシュで注意すべき問題については、『Oracle8 Parallel Server 概要および管理』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、4-26 ページの「順序番号をキャッシュする」を参照してください。

順序を作成するために必要な権限

使用する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。または別のユーザーのスキーマに順序を作成するには、CREATE ANY SEQUENCE システム権限を持っていなければなりません。

順序を変更する

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、再作成しなければなりません。

順序を変更するには、次のように SQL コマンド ALTER SEQUENCE を使用してください。

```
ALTER SEQUENCE emp_sequence
    INCREMENT BY 10
    MAXVALUE 10000
    CYCLE
    CACHE 20;
```

順序を変更するために必要な権限

順序を変更するには、自分のスキーマにその順序が含まれているか、あるいは ALTER ANY SEQUENCE システム権限を持っていなければなりません。

順序を使用する

次の項では、順序が定義されている場合の順序の使用方法を説明します。いったん定義すると、順序は多くのユーザーで使用できます。順序は複数のユーザーが待機することなく、アクセスし、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、その順序を増分できます。

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、注文入力システムが部分的に 2 つの表、ORDERS（マスター表）と LINE_ITEMS（ディテール表）によって構成されていると想定してください。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;
```

順序を参照する

順序は、NEXTVAL 擬似列と CURRVAL 擬似列を使用した SQL 文で参照されます。現行の順序番号は、順序の擬似列 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、擬似列 NEXTVAL を参照することによって生成されます。

NEXTVAL と CURRVAL は、予約語やキーワードではなく、SELECT、INSERT、UPDATE などの SQL 文において擬似列名として使用できます。

NEXTVAL を使用して順序番号を生成する　順序番号を生成して使用するには、seq_name.NEXTVAL を参照してください。たとえば、顧客が注文する場合を想定します。次のようにして、順序番号を値リストで参照できます。

```
INSERT INTO orders (orderno, custno)
    VALUES (order_seq.NEXTVAL, 1032);
```

次に示すとおり、UPDATE 文の SET 句で使用することもできます。

```
UPDATE orders
    SET orderno = order_seq.NEXTVAL
    WHERE orderno = 10112;
```

次のように、問合せや副問合せの最も外側の SELECT で使用することもできます。

```
SELECT order_seq.NEXTVAL FROM dual;
```

定義によって、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。擬似列 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。しかし、各行に生成できる順序番号は 1 つだけです。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その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。

いったん順序番号が生成されると、その順序番号は、その番号を生成したセッションに対してのみ利用可能です。トランザクションのコミットやロールバックとは無関係に、ORDER_SEQ.NEXTVAL を参照している他のユーザーは、一意の値を獲得します。2 人のユーザーが同時に同じ順序を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 を使用して順序番号を使用する　セッションの現行の順序値を使用したり、参照したりするには、seq_name.CURRVAL を参照してください。seq_name.NEXTVAL が（現行のトランザクションや前の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だけ、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、注文に対してライン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 (orderno, partno, quantity)
    VALUES (order_seq.CURRVAL, 20321, 3);

INSERT INTO line_items (orderno, partno, quantity)
    VALUES (order_seq.CURRVAL, 29374, 1);
```

前項で与えられた 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両方の行が、順序番号 347 を持つ行を挿入します。

NEXTVAL と CURRVAL の使用と制限　CURRVAL と NEXTVAL は、次の位置で使用できます。

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

CURRVAL と NEXTVAL は、次の位置では使用できません。

- 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、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

順序番号をキャッシュする

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。ディスクから読み込むよりも高速に順序キャッシュ内の順序番号にアクセスできます。

順序キャッシュはエントリから構成されます。それぞれのエントリは、単一の順序に対する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ために、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順序をすべて保持できることを確認する。
- 順序キャッシュに保持する各順序の値の数を増やす。

順序キャッシュ内のエントリの数　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き、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ない順序にアクセスすると、順序をディスクからキャッシュに読み込んでから順序番号を使用する必要があります。

アプリケーションが同時に数多くの順序を使用する場合、順序キャッシュがすべての順序を保持できるほど十分に大きくない場合があります。この場合、順序番号にアクセスすると、ディスクの読込みが頻繁に必要となる可能性があります。すべての順序に高速にアクセスするために、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順序をすべて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の数は、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だけでなく、これらの順序も保持できるように、順序キャッシュが十分なエントリを持っ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の最も長い間使用されていない順序に置き換わります。この置き換えられた順序の残りの値はすべて失われます。つまり、置き換えられた順序が最初に 10 個の順序値をキャッシュし、その中の 1 つの順序値だけが使用されていた場合、順序が置き換えられた時点で 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でアクセスできます。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えます。ただし、インスタンス障害が発生すると、キャッシュ内の順序値はすべて失われます。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を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、ディスクの読込みが必要になります。このようなディスクの読込みは、順序へのアクセスを遅くします。次の CREATE SEQUENCE 文は、順序値がキャッシュに格納されないように、SEQ3 順序を作成します。

```
CREATE SEQUENCE seq3
    NOCACHE
```

順序を使用するために必要な権限

順序を使用するには、使用する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。

順序を削除する

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きに、その定義は、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、参照されたときにエラーを戻します。

順序を削除するために必要な権限

自分の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 システム権限を持っていなければなりません。

シノニムを管理する

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、関数、あるいはパッケージの別名です。以降の項では、SQL コマンドを使用して、シノニムを作成、使用、削除する方法について説明します。

シノニムを作成する

シノニムを作成するには、SQL コマンド CREATE SYNONYM を使用してください。次の文は、JWARD のスキーマに含まれる EMP 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;
```

シノニムを作成するために必要な権限

使用するスキーマにプライベート・シノニムを作成するための CREATE SYNONYM システム権限を持っているか、あるいは別のユーザーのスキーマにプライベート・シノニムを作成するための CREATE ANY SYNONYM システム権限を持っていなければなりません。パブリック・シノニムを作成するには、CREATE PUBLIC SYNONYM システム権限を持っていなければなりません。

シノニムを使用する

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 SQL 文で参照できます。たとえば、シノニム EMP 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 (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);
```

シノニムを使用するために必要な権限

使用可能なロールから明示的に、または PUBLIC から、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、使用する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。付与されているオブジェクト権限を使用して、別のユーザーのシノニムを参照できるだけです。たとえば、JWARD.EMP シノニムに対して SELECT 権限を持っている場合、JWARD.EMP シノニムを問い合せることはできますが、JWARD.EMP のシノニムを使用して、行を挿入することはできません。

シノニムを削除する

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 キーワードを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には、PUBLIC キーワードを指定してください。次の文は、EMP 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;
```

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き、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。

シノニムを削除するために必要な権限

自分の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 システム権限を持っていなければなりません。パブリック・シノニムを削除するには、DROP PUBLIC SYNONYM システム権限を持っていなければなりません。

索引を管理する

Oracle では、索引を使用して、表の行に高速にアクセスできます。表全体の行のわずかな部分を戻す処理に対して、索引はデータへの高速なアクセスを実現します。

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の利点とデータベース・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すべきです。

以降の項では、SQL コマンドを使用して、索引を作成、変更、削除する方法を説明します。索引を管理するときに従うべき簡単なガイドラインも示します。索引作成のパフォーマンスに関する面については、『Oracle8 Server 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

注意すべき例外は 1 つありますが、索引の作成は、通常、データが挿入された後、またはロードされた（SQL*Loader またはインポートによって）後で行ってください。索引を持っていない表にデータの行を挿入してから、これからの問合せなどに対して索引を作成することが、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あらゆる索引が更新されなければなりません。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対して索引を作成しておかなければならないということです。

データをすでに持っている表に対して索引が作成されるとき、Oracle は索引を作成するためにソート領域を使用しなければなりません。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（初期化パラメータ SORT_AREA_SIZE によって決定されるユーザーあたりの容量）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ソート情報をスワップしなければなりません。索引が非常に大きい場合、次のステップを完了すると有効となる場合があります。

1. CREATE TABLESPACE コマンドを使用して新しい一時表領域を作成する。

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する。
3. CREATE INDEX コマンドを使用して索引を作成する。
4. DROP TABLESPACE コマンドを使用してこの表領域を削除する。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再設定します。

ある条件の下では、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。詳細は、『Oracle8 Server ユーティリティ』を参照してください。

正しい表と列の索引　索引を作成すべき場合を判断するために、次のガイドラインを使用してください。

- 大規模な表の行の 15% 未満を頻繁に検索する場合、索引を作成してください。この行の割合は、表走査の相対的な速度と索引キーについて行データをクラスタ化している方法に従って大きく変動します。表走査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。
- 複数の表の結合に対す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。

  注意 : 主キーと一意キーは、自動的に索引を持ちますが、外部キーに対しても索引を作成するとよいでしょう。詳細は、9-10 ページの「同時実行性制御、索引、外部キー」を参照してください。

- 小さな表は索引を必要としません。しかし、問合せがかなり大規模なときには、表が大きくなっていることがあります。

索引を付ける対象として有力な候補列がいくつか存在します。次の特徴を 1 つ以上持っている列は、このような候補列となります。

- 列の値は比較的一意である。
- 値の範囲が広い。
- 列には多くの NULL が含まれるが、問合せは値を持つ行を頻繁に選択する。この場合、次の句

  ```
  WHERE COL_X > -9.99 x 10^125
  ```

  は次の句より好ましい。

  ```
  WHERE COL_X IS NOT NULL
  ```

  この理由は、最初の句が、COL_X に対する索引を使用するからである（COL_X は数値列であると想定している）。

次の特徴を持っている列は、索引を付ける対象としてあまり適していません。

- 列が異なる値をほとんど持っていない（たとえば、従業員の性別の列）。
- 列の中に多くの NULL があり、NULL でない値を探索しない。

LONG 列と LONG RAW 列には索引を付けられません。

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（そこから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、索引によって必要となる領域を判断してください。

表あたりの索引の数の制限 表は、かなり多くの索引を持つことができます。しかし、索引が多いほど、表を変更するときより多くのオーバーヘッドが発生します。行の挿入や削除を行うとき、その表の索引もすべて同じように更新しなければなりません。列を更新するときには、その列を含む索引もすべて更新しなければなりません。

したがって、表に対する問合せの検索速度とその表に対して更新を行う速度の間に、妥協点が存在します。たとえば、表が主に読取り専用であれば、索引を増やすと有効ですが、表がかなり頻繁に更新されるのであれば、索引数を減らす方が望ましいでしょう。

パフォーマンスのための索引列の順序 列が CREATE INDEX コマンドの中で指定する順序は、それらが表に現れる順序に対応している必要はありません。しかし、CREATE INDEX 文における列の順序は重要です。問合せのパフォーマンスが、その選ばれた順序によって影響を受ける可能性があるからです。一般に、最も使用されると予想される列を索引の先頭に置くべきです。

たとえば、図 4-1 に示すような VENDOR_PARTS 表の列を想定してください。

図 4-1　VENDOR_PARTS 表

Table VENDOR_PARTS

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、次のような SQL 文によって問い合せることにします。

```
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的な例（つまり、最も多くの値を持つ列）を先頭にした複合索引を作成するとよいでしょう。

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

複合索引は、索引の「先頭部分」を使用して、問合せの検索速度を向上させます。したがって、上の例では、PART_NO 列だけを使用する 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。異なる値が 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。

索引を作成する

表に対する索引を作成して、その対応する表に対して発行した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 16 列までの複数列に対して複合索引を作成できます。複合索引キーは、データ・ブロック内の使用可能領域のおよそ 2 分の 1（そこから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、UNIQUE 整合性制約または PRIMARY KEY 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、そのような制約を作成するのがよく、廃止された CREATE UNIQUE INDEX 構文を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 を使用してください。次の文は、EMP 表の ENAME 列に対して索引 EMP_ENAME を作成します。

```
CREATE INDEX emp_ename ON emp(ename)
    TABLESPACE users
    STORAGE (INITIAL 20K
```

```
        NEXT 20k
        PCTINCREASE 75)
    PCTFREE 0;
```

いくつかの記憶領域設定が、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。

索引を作成するために必要な権限

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して割当て制限を持っているか、または UNLIMITED TABLESPACE システム権限を持っていなければなりません。別のユーザーのスキーマに索引を作成するには、CREATE ANY INDEX システム権限を持っていなければなりません。

索引を削除する

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、索引により予想されたほどパフォーマンスが改善されていない（表が非常に小さい、あるい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は索引を使用する問合せが含まれない。
- 必要でなくなった索引は、再作成の前に削除しなければならない。

索引が削除されると、その索引のセグメントのエクステントはすべて、索引を含んでいる表領域に戻され、表領域内の他のオブジェクトに対して利用可能になります。

索引を削除するには、SQL コマンド DROP INDEX を使用してください。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、対応付けられた索引はすべて削除されます。

索引を削除するために必要な権限　索引を削除するには、その索引が自分のスキーマに含まれるか、あるいは DROP ANY INDEX システム権限を持っていなければなりません。

クラスタ、クラスタ化した表、クラスタ索引を管理する

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、クラスタ化した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（つまり関連した値）は、値を含む異なる表の列の行にかかわりなく、一度格納されるだけです。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領域は、クラスタ化しない表の形式で必要となる記憶領域よりも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

次にクラスタを作成するためのガイドラインをいくつか示します。パフォーマンスの特性については、『Oracle8 Server チューニング』を参照してください。

クラスタ化に適切な表を選択する　問合せが主体で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、あるいは単一の表から関連したデータを検索したりする 1 つ以上の表を格納するために、クラスタを使用してください。

クラスタ、キーに対して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・キーを複合キーにしてください。一般に、よい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。これらのガイドラインについては、4-30 ページの「正しい表と列の索引」を参照してください。

各キー値に対応している行のグループがおおよそ 1 つのデータ・ブロックを満杯にするように、よいクラスタ・キーは、十分な一意の値を持ちます。クラスタ・キー値あたりの行が少なすぎると、領域を浪費し、パフォーマンスの向上は取るに足りない結果となります。わずかな行だけが共通の値を共有するような特性のクラスタ・キーは、クラスタ作成時に小さな SIZE を指定しないかぎり、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、余分な探索が発生する可能性があります。また、あまりにも変化のない値（たとえば性別）に対するクラスタ・キーは、余分な探索が発生し、クラスタ化しない場合よりもパフォーマンスが悪くなる可能性があります。

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

パフォーマンスの検討

索引を付けた表を別々に格納することと比較して、クラスタが DML 文（INSERT および UPDATE、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、表を走査するために検査しなければならない領域の使用量と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ので、クラスタ化した表を格納するのに使用しなければならないブロックの数は、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。このような長所短所を念頭に置いて、クラスタを使用するかどうか判断する必要があります。

クラスタ化しない形式よりもクラスタ化した形式で格納するほうがよいデータを識別するには、参照整合性制約を介して関連する表と、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捜します。表データを結合するために使用される列について表をクラスタ化すると、問合せを処理するためにアクセスしなければならないデータ・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行はすべて同じブロッ

クに入っています。そのため、結合に対する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 表は、同じ部門の従業員に対する行をクラスタ化するために、DEPTNO 列についてクラスタ化することもできます。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、これは有効です。

索引のように、クラスタはアプリケーション設計に影響を及ぼしません。クラスタの存在は、ユーザーとアプリケーションに対して透過的です。クラスタ化した表に格納されたデータは、クラスタ化しない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、クラスタ化した表、クラスタ索引を作成する

問合せで頻繁に結合する 1 つまたは複数の表を格納する場合には、クラスタを使用してください。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。しかし、クラスタ化した表に行を挿入する前に、クラスタ索引を作成しなければなりません。クラスタの使用は、クラスタ化した表に対する追加の索引の作成には影響を及ぼしません。つまり、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 を使用してください。次の文は、EMP 表と DEPT 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てください。たとえば、次の文を使用して、EMP_DEPT クラスタ内に EMP 表と DEPT 表を作成できます。

```
CREATE TABLE dept (
    deptno NUMBER(3) PRIMARY KEY,
    . . . )
    CLUSTER emp_dept (deptno);

CREATE TABLE emp (
    empno NUMBER(5) PRIMARY KEY,
    ename VARCHAR2(15) NOT NULL,
    . . .
    deptno NUMBER(3) REFERENCES dept)
    CLUSTER emp_dept (deptno);
```

クラスタ内に作成する表は、CREATE TABLE 文で指定したスキーマに含まれます。クラスタ化した表は、そのクラスタを含む同じスキーマに存在しない可能性もあります。

クラスタ化した表に行を挿入する前に、クラスタ索引を作成しなければなりません。たとえば、次の文は、EMP_DEPT クラスタに対して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、一意にすることはできません。それだけではなく、Oracle では、UNIQUE 制約または PRIMARY KEY 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、クラスタ化した表、クラスタ索引を作成するために必要な権限

自分のスキーマにクラスタを作成するには、CREATE CLUSTER システム権限と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、しかも所有者がそのクラスタを含む予定の表領域に対する割当て制限を持っているか、または UNLIMITED TABLESPACE システム権限を持っていなければなりません。

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 システム権限を持っていなければなりません。クラスタ内に表を作成するために、表領域割当て制限または 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、自分のスキーマにクラスタが含まれるか、または次の権限を持っていなければなりません。

- CREATE ANY INDEX システム権限、またはクラスタを所有していれば 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

クラスタに対して記憶領域を手動で割り当てる

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、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ことが望ましい場合もあります。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す。

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。詳細は、『Oracle8 Parallel Server 概要および管理』を参照してください。

クラスタ、クラスタ化した表、クラスタ索引を削除する

クラスタ内に現在ある表が必要でなくなった場合は、クラスタを削除してください。クラスタを削除すると、そのクラスタ内にある表と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、それらを含んでいる表領域に戻され、表領域内の他のセグメントに対して利用可能になります。

クラスタ化した表は、その表のクラスタ、他のクラスタ化した表、またはクラスタ索引に影響を及ぼさずに、個別に削除できます。クラスタ化しない表を削除する場合と同様に、SQL コマンド DROP TABLE を使用して、クラスタ化した表を削除してください。表を個別に削除する方法は、4-9 ページの「表を削除する」を参照してください。

注意： 単一のクラスタ化した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しなければなりません。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予定であれば、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。クラスタの残りの部分がそのまま残るときには、個々の表をクラスタから削除するために DROP TABLE コマンドを使用すべきです。

クラスタ索引は、クラスタまたはそのクラスタ化した表に影響を及ぼさずに、削除できます。しかし、クラスタ索引がないと、クラスタ化した表を使用することはできません。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。詳細は、4-33 ページの「索引を削除する」を参照してください。

表を含まないクラスタとそのクラスタ索引（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した表を含んでいて、その表も削除する予定であれば、DROP CLUSTER コマンドの INCLUDING TABLES オプションを追加してください。

```
DROP CLUSTER emp_dept INCLUDING TABLES;
```

INCLUDING TABLES オプションを指定しない場合に、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 FOREIGN KEY 制約によって参照される主キーまたは一意キーを含んでいる場合、その依存関係を持つ FOREIGN KEY 制約も削除しないかぎり、そのクラスタを削除することはできません。次のように、DROP CLUSTER コマンドの CASCADE CONSTRAINTS オプションを使用してください。

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

上のオプションを使用しないと、エラーが戻される場合があります。

クラスタを削除するために必要な権限

クラスタを削除するには、自分のスキーマにそのクラスタが含まれているか、あるいは DROP ANY CLUSTER システム権限を持っていなければなりません。クラスタ化した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に特別の権限を持っている必要はありません。

ハッシュ・クラスタとクラスタ化した表を管理する

以降の項では、SQL コマンドを使用して、ハッシュ・クラスタとクラスタ化した表を作成、変更、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した表を作成する

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した表のグループや個別の表を格納するために使用されます。ハッシュ・クラスタを作成してから、表を作成できます。ハッシュ・クラスタを作成するには、SQL コマンド CREATE CLUSTER を使用してください。次の文は、TRIAL 表を格納するために使用するクラスタ TRIAL_CLUSTER を作成し、TRIALNO 列によってクラスタ化します。

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

CREATE TABLE trial (
    trialno NUMBER(5) PRIMARY KEY,
    ...)
    CLUSTER trial_cluster (trialno);
```

ハッシュ・クラスタ内の領域使用を制御する

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の使用方法が望ましいものになるように、クラスタ・キーを正しく選択し、HASH IS、SIZE、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だけでなく手引も提供します。

キーを選択する

正しいクラスタ・キーの選択は、クラスタ化した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 表について検討します。問合せが従業員番号によって行をよく選択する場合、EMPNO 列をクラスタ・キーにするとよいでしょう。問合せが部門番号によって行をよく選択する場合には、DEPTNO 列をクラスタ・キーにすべきです。単一の表を含むハッシュ・クラスタの場合、通常クラスタ・キーをその含まれる表の主キー全体にします。複合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しなければなりません。

HASH IS を設定する

クラスタ・キーが NUMBER データ型の単一の列であり、しかも一様に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てください。この条件が適用される場合、各々一意のクラスタ・キー値が、一意のハッシュ値に対してハッシュするように（競合なしで）、クラスタ内に行を分布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べきです。

ハッシュ・クラスタを削除する

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

```
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・クラスタとハッシュ・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。クラスタの削除と必要な権限の詳細は、4-37 ページを参照してください。

どんな場合にハッシュを使用するか

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代わるものです。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・キーに対して等号条件を指定した問合せである場合。たとえば、次のとおりです。

  ```
  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、通常単一の読込みによって見つけられます。索引を付けられた表では、キー値を最初に索引で見つけなければならず（通常複数の読込み）、次に行が表から読み込まれます（別の読込み）。

- ハッシュ・クラスタ内の表が、サイズにおいて本来固定的であり、行の数と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、オーバーフロー・ブロックが必要となるために、パフォーマンスの低下は相当なものになる可能性があります。

- HASH IS col および、HASHKEYS n、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。たとえば、次のとおりです。

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

  上の配列は次のように表現できる。

  ```
  CREATE CLUSTER c(subscript INTEGER)
  ```

```
    HASH IS subscript HASHKEYS 100 SIZE 10;
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。

- 表に対するほとんどの問合せが、クラスタ・キー値の範囲外の行を検索する場合。たとえば、全表走査、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ない。そのかわりに、問合せに対して行を取り出すために、全表走査に等しい処理を行わなければなりません。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。

- 表が固定的でなく絶えず成長する場合。表が無制限に成長する場合、表（したがってそのクラスタ）の存続期間にわたって必要となる領域を前もって定義することはできません。
- アプリケーションが表に対して頻繁に全表走査を行い、その表内のデータが散在している場合。この条件で全表走査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のか、索引を使用するのかを（上の情報に基づいて）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なのか、クラスタの一部として格納することが最適なのかを検討してください。このガイダンスについては、4-34 ページを参照してください。

ハッシュを使用することを決めた場合でも、表はどの列に対しても個別に索引（クラスタ・キーも含めて）を持つことができます。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、『Oracle8 Server チューニング』を参照してください。

スキーマ・オブジェクトの管理について

以降の項では、この章で述べたいろいろなスキーマ・オブジェクトの管理に関するその他のトピックについて説明します。

- 一回の操作で複数の表とビューを作成する
- スキーマ・オブジェクトを命名する
- SQL 文における参照オブジェクトの名前変換
- スキーマ・オブジェクトを改名する
- スキーマ・オブジェクトに関する情報をリストする

一回の操作で複数の表とビューを作成する

SQL コマンド CREATE SCHEMA を使用して、一回の操作で複数の表やビューを作成したり、複数の権限を付与したりできます。一回の操作で複数の表やビューの作成と権限付与を確実に行いたい場合は、CREATE SCHEMA コマンドが有効です。個々の表や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、文全体がロールバックされるため、オブジェクトの作成や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
        WHERE deptno = 30  WITH CHECK OPTION CONSTRAINT
                           sales_staff_cnst
CREATE TABLE emp (
    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)

CREATE TABLE dept (
    deptno      NUMBER(3) PRIMARY KEY,
    dname       VARCHAR2(15),
    loc         VARCHAR2(25))
GRANT SELECT ON sales_staff TO human_resources;
```

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

複数のスキーマ・オブジェクトを作成するために必要な権限

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して必要な権限を持っていなければなりません。

スキーマ・オブジェクトを命名する

ビュー、シノニム、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のかを判断する必要があります。データベース名は固定すべきであり、ネットワーク内で必要もなくデータベースを移動させるべきではないことを忘れないでください。

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べきです。グローバル名は、データベース名と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・オブジェクト名とグローバル・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 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を保証するので、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り、すべてのデータベース間に渡ってそのオブジェクト名が一意であることを保証できます。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りますので、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べきです。

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
scott.emp
scott.emp@personnel
```

セッションは、ユーザーがデータベースにログオンした時点で確立されます。オブジェクト名は、現行のユーザー・セッションを元に変換されます。現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オ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領域を持っています。同じ名前領域のオブジェクトはすべて異なる名前を持っていなければなりませんが、異なる名前領域にある 2 つのオブジェクトが、同じ名前を持つことはできます。表およびビュー、スナップショット、順序、シノニム、プロシージャ、関数、パッケージは、同じ名前領域に存在します。トリガーおよび索引、クラスタは、それぞれ別々の名前領域を持っています。たとえば、表およびトリガー、索引のすべてに SCOTT.EMP 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、適切な名前領域を探索します。たとえば、次の文の場合、

```
DROP CLUSTER test
```

Oracle はクラスタの名前領域内の 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 内に暗示的にありますが、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することはできません。

シノニムを使用して参照できるのは、表と同じ名前領域内のオブジェクトだけです。名前がシノニムである可能性があるため、表の名前領域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ために、次のルールを使用します。

1. 表の名前領域の名前を調べます。
2. 名前がシノニムではないオブジェクトに変換される場合、それ以上の作業は必要ありません。
3. 名前がプライベート・シノニムに変換された場合、その名前をシノニムの定義で置き換えてステップ 1 に戻ります。
4. 名前がスキーマで修飾されていた場合、エラーを戻します。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き（明示的に、あるいはシノニム内で間接的に）、ローカル Oracle セッションは、ローカルで必要な分だけ、その参照を変換します（たとえば、シノニムをリモート表のグローバル・オブジェクト名に変換する）。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、上に説明したようにオブジェクトの変換を完了します。

分散データベースにおける名前変換の詳細は、『Oracle8 Server 概要』を参照してください。

スキーマ・オブジェクトを改名する

必要に応じて、2 つの異なる方法を使用して、いくつかのスキーマ・オブジェクトを改名できます。つまり、オブジェクトを削除してから再作成するか、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は失われます。オブジェクトを再作成するときに、再度権限を付与しなければなりません。

表またはビュー、順序あるいは表のプライベート・シノニムを改名するのに 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、クラスタを改名することはできません。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、次のような影響が及びます。

- 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と PL/SQL プログラム・ユニットは、無効になる（次の使用の前に再コンパイルしなければならない）。
- 改名されたオブジェクトに対するすべてのシノニムは、使用されるとエラーを戻す。

オブジェクトを改名するために必要な権限

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なければなりません。

スキーマ・オブジェクトに関する情報をリストする

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与えます。次に、スキーマ・オブジェクトに関係するビューをまとめます。

- 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_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: タイプ別に異なるスキーマ・オブジェクトをリストする 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オブジェクトのすべてをリストします。

```
SELECT object_name, object_type FROM user_objects;
```

上の問合せが戻す結果は、次の表示例と類似したものになります。

```
OBJECT_NAME                    OBJECT_TYPE
------------------------------ -------------------
EMP_DEPT                       CLUSTER
EMP                            TABLE
```

```
DEPT                      TABLE
EMP_DEPT_INDEX            INDEX
PUBLIC_EMP                SYNONYM
EMP_MGR                   VIEW
```

例 2: 列情報をリストする 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、位取り、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 表と DEPT 表に対するデフォルト列値のすべてをリストします。

```
SELECT table_name, column_name, data_default
    FROM user_tab_columns
    WHERE table_name = 'DEPT' OR table_name = 'EMP';
```

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        DEPTNO
DEPT        DNAME
DEPT        LOC             ('NEW YORK')
EMP         EMPNO
EMP         ENAME
EMP         JOB
EMP         MGR
EMP         HIREDATE        (sysdate)
EMP         SAL
EMP         COMM
EMP         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るとき、列の値として NULL が想定されます。

例 3: ビューとシノニムの依存性をリストする ビューまたはシノニムを作成するとき、ビューやシノニムはその基礎になるベース ・オブジェクトに基づきます。_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、_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、次の表示例と類似したものになります。

```
TABLE_OWNER                    TABLE_NAME
------------------------------ ------------
SCOTT                          DEPT
SCOTT                          EMP
```

5

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 ビルトイン・データ型の使用方法について説明します。トピックは次のとおりです。

- Oracle ビルトイン・データ型
- ROWID と ROWID データ型
- Trusted Oracle MLSLABEL データ型
- ANSI/ISO データ型および DB2 データ型、SQL/DS データ型
- データ変換

関連項目： ユーザー定義のデータ型の情報は、『Oracle8 Server 概要』およびこのマニュアルの第 7 章「ユーザー定義データ型 ― 拡張例」を参照してください。

Oracle ビルトイン・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、またはプロシージャか関数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 データ型の値を加算することはできますが、RAW データ型の値を加算することはできません。

Oracle は、次のビルトイン・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

上記の他にデータ型 ROWID があり、表の各行の単一のアドレスを表す ROWID 疑似列中の値に使用されます。

> **関連項目：** 表 5–2 は、各 Oracle ビルトイン・データ型に関する情報をまとめたものです。これらのデータ型の一般的な説明については『Oracle8 Server 概要』を、LOB データ型についてはこのマニュアルの第 6 章「ラージ・オブジェクト (LOB)」を参照してください。

表 5–1　ビルトイン・データ型のサマリー

データ型	説明	列の長さとデフォルト
CHAR (size)	長さが size バイトの固定長の文字データ。	表の各行で固定される (後続に空白が付く)。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。サイズ設定する前に、キャラクタ・セット (1 バイトかマルチバイトか) を考慮してください。
VARCHAR2 (size)	可変長の文字データ。	各行で可変で、1 行あたり最大 4000 バイト。サイズ設定する前に、キャラクタ・セット (1 バイトかマルチバイトか) を考慮してください。最大の size を指定しなければなりません。
NCHAR(size)	長さが size 文字または size バイト (各国文字キャラクタ・セットに応じて決まる) の固定長の文字データ。	表の各行で固定される (後続に空白が付く)。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 文字を格納するの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 (キャラクタ・セットに応じて決まる) です。
NVARCHAR2 (size)	長さが size 文字または size バイト (各国語キャラクタ・セットに応じて決まる) の可変長の文字データ。最大の size を指定しなければなりません。	各列で可変。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 文字を格納するの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 (キャラクタ・セットに応じて決まる) です。
CLOB	シングルバイト文字データ。	最大 2 の 32 乗 - 1 バイト、すなわち 4GB。

表 5–1　ビルトイン・データ型のサマリー

NCLOB	シングル・バイトまたは固定長マルチバイトの各国文字キャラクタ・セット (NCHAR) データ。	最大 2 の 32 乗 - 1 バイト、すなわち 4GB。
LONG	可変長の文字データ。	表の各行で可変で、1 行あたり最大 2 の 31 乗 - 1 バイト、すなわち 2GB。下位互換性のために提供されています。
NUMBER (p, s)	可変長の数値データ。精度 p または位取り s(あるいはその両方) の最大値は 38 です。	各列で可変。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。
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。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-YY) です。
BLOB	非構造化バイナリ・データ。	最大 2 の 32 乗 - 1 バイト、すなわち 4GB。
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、すなわち 4GB。
RAW(size)	可変長のバイナリ・データ。	各列で可変で、1 行あたり最大 2000 バイト。最大の size を指定しなければなりません。下位互換性のために提供されています。
LONG RAW	可変長のバイナリ・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、すなわち 2GB。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。	表の各列で、10 バイト (拡張 ROWID の場合) または 6 バイト (制限 ROWID の場合) に固定される。
MLSLABEL	Trusted Oracle データ型。	『Trusted Oracle』のマニュアルを参照。

文字データ型を使用する

文字データ型は、英数字データを格納するのに使用します。

- CHAR データ型および NCHAR データ型は、固定長文字列を格納する。
- VARCHAR2 データ型および NVARCHAR2 データ型は、可変長文字列を格納する。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する。

  関連項目： 第 6 章「ラージ・オブジェクト (LOB)」

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

  関連項目： 5-10 ページ「LONG データおよび LONG RAW データに関する制限」

  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。一般に、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の空白を埋め込み、固定列長最後まで後続する空白を格納しますが、VARCHAR2 データ型では列値の空白を埋め込まず、後続する空白も格納しません。

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する空白が重要でない（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する空白が重要である（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 データ型と VARCHAR2 データ型は、将来にわたって完全なサポートを保証されています。現時点では、VARCHAR データ型は VARCHAR2 データ型と同義であると自動的にみなされるため、将来も使用できる予定です。

CHAR データおよび VARCHAR2 データ、LONG データは、データベースのキャラクタ・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は) 自動的に変換されます。

シングル・バイト文字セットおよびマルチバイト文字セットの列の長さ

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 列と 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されます。

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。

> **関連項目：** Oracle の各国語サポート機能と各種文字コード体系のサポートの詳細は、『Oracle8 Server リファレンス・マニュアル』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り 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。後続する空白の文字数だけが異なる場合、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だけ、等しいものと見なされます。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、より大きい値と見なされます。

Oracle では、CHAR 列に格納されている値には空白を埋め込みますが、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 列に値を格納すれば、CHAR 列に格納した場合よりも領域を節約できます。アプリケーションが文字データを含む大規模な表に対して全表走査を行う場合、CHAR 列よりも 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めるときに考慮すべき要因は、パフォーマンスだけではありません。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ほうがよい場合があります。たとえば、文字値の比較で、Oracle に後続の空白を無視させるのであれば、これらの値は CHAR 列に格納しなければなりません。

関連項目： これらのデータ型の比較方法の詳細は、『Oracle8 Server SQL リファレンス』を参照してください。

NUMBER データ型を使用する

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のに使用します。このデータ型を使用している数値は、さまざまな 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^-130) から 9.99... × (10^125) までの正と負の数値とゼロを、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）と「位取り」（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と位取りを指定することによって、入力時に整合性検査を追加できます。精度が指定されないと、列は与えられたとおりに値を格納します。表 5–2 は、異なる位取り位置を使用した、データの格納方法の例を示しています。

表 5–2　位取り位置がどのように数値データの記憶に影響を及ぼすか

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

関連項目： NUMBER データ型の内部形式については、『Oracle8 Server 概要』を参照してください。

DATE データ型を使用する

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、世紀および年、月、日、時間、分、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、秒に対応します。Oracle 内部日付形式の完全な説明は、『Oracle 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、次のような DD-MON-YY です。

```
'13-NOV-92'
```

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 文を使用してください。現行のデフォルト日付書式以外の日付を入力するには、次のように、書式マスク付き TO_DATE 関数を使用します。

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

注意： 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、『Oracle8 Server 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したい場合、異なる書式マスクを使用してください。

時間書式

時間は 24 時間形式 HH:MM:SS で格納されます。デフォルトでは、時間部分になにも入力しなければ、日付フィールドの中の時間は 12:00:00 A.M.(真夜中) となります。時間だけの入力では、日付部分は現在の月の最初の日が想定されます。日付の時間部分を入力するには、次のように、時間部分を示す書式マスク付きの TO_DATE 関数を使用します。

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

時間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のであれば、SQL 関数 TRUNC を使用してください。システム日付と時刻に戻すには、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。

世紀と西暦 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

れる他の日付もすべて 4 桁の年を使用します。インポートおよびエクスポート、回復などの Oracle ユーティリティでも、4 桁の年を正常に処理します。

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、プログラマの予期するものとは異なったりする可能性があります。したがって、アプリケーション・プログラマは、西暦 2000 年に関して、自分たちのコードを検査およびテストする必要があります。

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 桁の年に従って世紀を複数の異なる値にデフォルトで設定できます。それによって、50 から 99 までの年は 19xx 形式に、00 ～ 49 までの年は 20xx の形式にデフォルトで設定されます。これによって、アプリケーションでは新しい世紀に簡単に変換できます。

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 年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 (1901 年から始まる) ではなく、19 世紀です。

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したい場合は、次の式を使用します。

```
DECODE (TO_CHAR (userdate, 'YY'),
    '00', TO_CHAR (userdate - 366, 'CC'),
    TO_CHAR (userdate, 'CC'))
```

この式は、次のような処理を行います。まず、年の最後の 2 桁を取ります。'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る)。「00」以外は、Oracle 世紀を使用します。

> **関連項目：** 日付書式コードの詳細は、『Oracle8 Server SQL リファレンス』を参照してください。

LONG データ型を使用する

> **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 データ型に関する情報は、第 6 章「ラージ・オブジェクト (LOB)」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ー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 リスト、および UPDATE 文の SET 句、INSERT 文の VALUES 句で使用できます。LONG 列は、VARCHAR2 列と同じ特性を数多く持っています。

LONG データおよび LONG RAW データに関する制限

LONG(および LONG RAW、下記参照) 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 列だけ使用できる。
- LONG 列には索引を付けられない。
- LONG 列は整合性制約に指定できない。
- LONG 列は、WHERE 句または GROUP BY 句、ORDER BY 句、CONNECT BY 句の中、SELECT 文の DISTINCT 演算子といっしょには使用できない。
- LONG 列は、SUBSTR や INSTR などの SQL 関数によって参照できない。
- LONG 列は、副問合せの SELECT リスト、または集合演算子（UNION または UNION ALL、INTERSECT、MINUS）による複合型の問合せでは使用できない。
- LONG 列は SQL 式では使用できない。
- 問合せを使用して表を作成する (CREATE TABLE... AS SELECT...) とき、または問合せを使用して表またはビューに挿入する (INSERT INTO... SELECT...) とき、LONG 列を参照できない。
- 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することはできない。
- 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することはできない。
- :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 列といっしょに使用できない。
- LONG 列および LONG RAW 列は、分散 SQL 文では使用できない。
- LONG 列および LONG RAW 列は、レプリケートできない。

注意 : LONG データまたは LONG RAW データを含む表を設計する場合には、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、それぞれの LONG 列または LONG RAW 列を関連する他のデータとは別の表に格納してください。それらの 2 つの表は参照整合性制約によって関連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、関連するデータだけをアクセスする SQL 文は、LONG データまたは LONG RAW データを読み込まずに済みます。

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、次の 2 つの表を作成します。

```
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、それぞれの記事のテキストだけを格納します。ARTICLE_HEADER 表は、タイトル、第一著者、雑誌、発行日を含む、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関連付けられます。

このように設計することによって、SQL 文は、記事のテキストを読み込まずに、テキスト以外のデータを問い合せることができます。1991 年の 7 月に発行された雑誌 NATURE の第一著者をすべて選択する場合、ARTICLE_HEADER 表を問い合せる次の文を発行します。

```
SELECT first_author
    FROM article_header
    WHERE journal = 'NATURE'
        AND TO_CHAR(pub_date, 'MM YYYY') = '07 1991')
```

各記事のテキストが第一著者、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るでしょう。

RAW データ型と LONG RAW データ型を使用する

注意： RAW データ型と 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の場合は、大量のバイナリ・データには BLOB データ型および BFILE データ型を使用してください。BLOB データ型および BFILE データ型に関する情報は、第 6 章「ラージ・オブジェクト (LOB)」を参照してください。

RAW データ型と LONG RAW データ型は、Oracle によって解釈されない (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 進データとバイト列のために用意されています。たとえば、LONG RAW は、図形データまたは

音声データ、文書データ、2 進データの配列に使用できます。解釈の方法は使用方法によって異なります。

Net8 およびエクスポート・ユーティリティ、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実行しません。Oracle が RAW データまたは LONG RAW データと CHAR データ間を自動的に変換する場合（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 データに索引を付けることはできませんが、RAW データは索引を付けることができます。LONG RAW データの制限の詳細は、5-10 ページ「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 と ROWID データ型

Oracle データベースのクラスタ化していない表の中の各行に、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。クラスタ化した表の場合、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。

> **関連項目：** ROWID 疑似列および ROWID データ型についての一般的な情報は、『Oracle8 Server 概要』を参照してください。

拡張 ROWID 形式

Oracle Server では、表のパーティション、索引のパーティション、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。たとえば、次の問合せがあるとします。

```
SELECT DATA_OBJECT_ID FROM DBA_OBJECTS
    WHERE OWNER = 'SCOTT' AND OBJECT_NAME = 'EMP';
```

この問合せにより、SCOTT スキーマ内の EMP 表のデータ・オブジェクト識別子が戻されます。10-74 ページ「DBMS_ROWID パッケージ」では、DBMS_ROWID パッケージ関数を使用してデータ・オブジェクト識別子を得るための、その他の方法を説明しています。

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異なる使い方をしています。この項では、そのようなさまざまな使い方を説明します。

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 バイトとなります。

ROWID 疑似列　表および非結合ビューはそれぞれ、ROWID と呼ばれる疑似列を持っています。たとえば次のような文を見てみます。

```
CREATE TABLE T1 (col1 ROWID);

INSERT INTO T1 SELECT ROWID FROM EMP WHERE empno = 7499;
```

これらの文では、問合せを満たす EMP 表の行の ROWID 擬似列が戻され、それが T1 表に挿入されます。

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、OOOOOOFFFBBBBBBRRR 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・セグメントを識別する (上記の例では、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** (上記の例では、ファイル AAL)。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 (上記の例では、ブロック AAAAQk)。ブロック番号は、その**表領域に対してではなく**、データ・ファイルに関連する。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** (上記の例では、行 AAA)。

外部 ROWID はデコードする必要はありません。DBMS_ROWID パッケージの関数を使用して、拡張 ROWID の個々の コンポーネントを得ることができます。

関連項目：　10-74 ページ「DBMS_ROWID パッケージ」

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・ブロックおよび行、データ・ファイル・コンポーネントを伴って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 struct で表すと、次のようになります。

```
struct riddef {
    ub4    ridobjnum; /* データ・オブジェクト番号。このフィールドは
                         制限付き ROWID では使用されない */
    ub2    ridfilenum;
    ub1    filler;
    ub4    ridblocknum;
    ub2    ridslotnum;
}
```

ROWID の移行と互換性の問題

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だけで必要となります。新しい表では、常に拡張 ROWID が得られます。

> **関連項目：**『Oracle8 Server 管理者ガイド』

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。

> **関連項目：** 移行の詳細は、10-80 ページ「ROWID_TO_EXTENDED 関数」を参照してください。『Oracle8 Server 移行ガイド』も参照してください。

Oracle8 クライアントから Oracle7 データベースにアクセスする　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 句で使用する。
- WHERE CURRENT OF カーソル操作
- ROWID 型または CHAR 型のユーザー列に ROWID を格納する

- 16 進エンコードを使用して、ROWID を変換する（推奨できない方法であるため、DBMS_ROWID 関数を使用すること）。

Oracle7 クライアントから Oracle8 データベースにアクセスする 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 句で使用する
- WHERE CURRENT OF カーソル操作を使用する。
- ROWID を CHAR(18) データ型のユーザー列に格納する。

インポートとエクスポート 表の行に拡張 ROWID 値が含まれる場合、Oracle7 クライアントが、ROWID 列（ROWID 擬似列ではない）を持つ Oracle8 の表をインポートすることはできません。

Trusted Oracle MLSLABEL データ型

Trusted Oracle では、MLSLABEL データ型が提供されます。このデータ型は、Trusted Oracle のマルチレベル保護 (MLS) オペレーティング・システムで生成されたラベルの Oracle 内部表現を格納します。Trusted Oracle では、データベース・アクセスを制御するのにラベルを使用します。

Trusted Oracle のアプリケーションと互換性を持たせるために、MLSLABEL データ型を使用して列を定義できますが、Oracle8 の列に使用できる有効値は NULL だけです。

Trusted Oracle 内に表を作成するときは、ROWLABEL という名前の列が自動的に表に追加されます。この列は、表のすべての行について MLSLABEL データ型のラベルを収録します。

関連項目： MLSLABEL データ型、ROWLABEL 列、Trusted Oracle の詳細は、『Trusted Oracle』のマニュアルを参照してください。

ANSI/ISO データ型および DB2 データ型、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、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。

ANSI データ型から Oracle データ型への変換を表 5–3 に示します。ANSI/ISO データ型 NUMERIC および DECIMAL、DEC は、固定小数点数だけを指定できます。これらのデータ型に対して、s は 0 をデフォルトにとります。

表 5–3　ANSI データ型の 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 および TIMESTAMP、GRAPHIC、VARGRAPHIC、LONG VARGRAPHIC には、対応する Oracle データ型がありませんので、これらは使用できません。TIME データ型と TIMESTAMP データ型は、Oracle データ型 DATE の副構成要素です。

表 5–4 に、DB2 データ型と SQL/DS データ型の変換を示します。

表 5–4　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。一般に、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()`

暗黙のデータ型変換が、この後に説明するルールに従って行われます。

> **関連項目：** Trusted Oracle を使用している場合、データ変換および MLSLABEL データ型の詳細は、5-19 ページ「Trusted Oracle のデータ変換」を参照してください。

ルール 1: 代入

代入では、Oracle は次のような変換を自動的に行うことができます。

- VARCHAR2 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 MLSLABEL へ
- MLSLABEL から VARCHAR2 へ
- VARCHAR2 または CHAR から HEX へ
- HEX から VARCHAR2 へ

Oracle が、代入する値のデータ型を、代入先のデータ型に変換できる場合、代入は成功します。

次のリストの中の例で、次の文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ください。

```
var1  CHAR(5);
CREATE TABLE table1 (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なければなりません。たとえば、ストアド・プロシージャ本体内で次に示す代入で与えられるデータを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そのデータ型に変換可能でなければなりません。たとえば、Oracle は、TABLE1 に対して次に示す INSERT 文で与えられるデータを自動的に変換します（上の表定義を参照）。

  ```
  INSERT INTO table1 VALUES ('19');
  ```

- `UPDATE` *table* `SET` *column* = *expression*

  expression のデータ型は、column のデータ型と同じかそのデータ型に変換可能でなければなりません。たとえば、TABLE1 に対して発行された次の UPDATE 文で与えられるデータを自動的に変換します。

  ```
  UPDATE table1 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そのデータ型に変換可能でなければなりません。たとえば、Oracle は表から選択されたデータを次に示す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 WHERE col1 = 30;
  ```

ルール 2: 式の評価

式の評価では、Oracle は代入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タイプに変換されます。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だけ、CHAR から NUMBER への変換が成功します。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だけ、CHAR から DATE への変換が成功します。

一般的な式のタイプは次のとおりです。

- 単純式、たとえば次のような式

  ```
  comm + '500'
  ```

- ブール式、たとえば次のような式

  ```
  bonus > sal / '10'
  ```

- 関数・コールとプロシージャ・コール、たとえば次のような式

  ```
  MOD (counter, '2')
  ```

- WHERE 句の条件、たとえば次のような式

  ```
  WHERE hiredate = TO_DATE('1997-01-01','yyyy-mm-dd')
  ```

- WHERE 句の条件、たとえば次のような式

  ```
  WHERE rowid = 'AAAAaoAATAAAADAAA'
  ```

一般に、代入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 は式評価のルールを使用します。

次のような代入で、

```
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。成功すると、expression の評価は、結果として単一の値とデータ型になります。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。

Trusted Oracle のデータ変換

Trusted Oracle では、ラベルはコンパクトなバイナリ構造体として内部的に格納されます。Trusted Oracle は、ラベルを内部バイナリ形式から外部文字形式に変換する TO_LABEL 関数を提供します。Trusted Oracle でラベルを文字形式からバイナリ形式に変換するには、TO_CHAR 関数を使用します。

TO_LABEL 関数は、Trusted Oracle アプリケーションとの互換性のために提供されています。Oracle8 では、NULL 値を戻します。

関連項目： TO_LABEL 関数および TO_CHAR 関数を使用したラベル形式の変換に関する詳細は、『Trusted Oracle』のマニュアルを参照してください。

6

ラージ・オブジェクト (LOB)

Oracle8 では、ラージ・オブジェクト (LOB) を定義し、操作するためのサポートを提供しています。また、表内の LOB 列またはオブジェクト型の LOB 属性を作成および更新するため、SQL DDL コマンドおよび DML コマンドの機能を拡張しています。さらに、Oracle8 では、LOB に対するランダムなピース単位の操作を実施するための、Oracle コール・インタフェース (OCI) および PL/SQL パッケージ API を提供しています。

この章では、LOB 用の拡張 SQL コマンドおよび PL/SQL パッケージ API について説明します。また、LOB 操作用の OCI API についても簡単にふれますが、詳しくは『Oracle コール・インタフェース・プログラマーズ・ガイド』に説明があります。

この章には、次の 2 つの項があります。

- LOB の概要 － Oracle8 での LOB の概要、および LOB 操作の主要な点について説明します。
- LOB リファレンス － PL/SQL DBMS_LOB パッケージについてのすべての技術仕様部の詳しいリストです。

LOB の概要

概要のまとめ

この項では、Oracle8 における LOB の取扱いについて、次に示す見出しに従って説明します。本文中に明示的に述べてはいませんが、この項で説明するさまざまな事柄は、いくつかの包括的なトピックにまとめることができます。

第 1 のトピックは、一般的な概要です。

- LOB とは何か？
- 内部 LOB と外部 LOB(BFILE)
  - 内部 LOB
  - 外部 LOB(BFILE)
- LOB データ型
  - 内部 LOB データ型
  - 外部 LOB データ型
- LOB と、LONG 型および LONG RAW 型との比較
- LOB の操作に使用するパッケージ

第 2 のトピックでは、LOB の操作を始める前に必要なステップについて説明します。

- 内部 LOB と外部 LOB を表に定義する (SQL DDL)
- 内部 LOB 用の表領域と記憶特性を取り決める
  - 表領域と LOB 索引
  - `PCTVERSION`
  - `CACHE / NOCACHE`
  - `LOGGING / NOLOGGING`
  - `CHUNK`
- 内部 LOB を初期化する (SQL DML)

第 3 のトピックでは、外部 LOB(BFILE) の処理に特有の事柄について説明します。次の事柄が含まれます。

- 外部 LOB にアクセスする (SQL DML)
  - BFILENAME() と OCIFileSetName() を使用して BFILES を初期化する

  - DIRECTORY オブジェクト
  - DIRECTORY 名の指定
- BFILE セキュリティ
  - 所有権とセキュリティ
  - BFILE セキュリティ用の SQL DDL
  - BFILE セキュリティ用の SQL DML
- ディレクトリのカタログ・ビュー
- DIRECTORY 使用のガイドライン
- `BFILES` の最大オープン数
- MTS モードにおける BFILE
- プログラム終了後に BFILE をクローズする

第 4 のトピックでは、ロケータによって LOB を処理する方法について検討します。

- LOB 値とロケータ
  - LOB 値のインライン記憶域
  - LOB ロケータ
  - 内部 LOB ロケータ
  - BFILE ロケータ
- LOB ロケータの操作
  - LOB 列 / 属性を設定する
  - ロケータによって LOB にアクセスする
  - 読取り一貫性のあるロケータ
  - 更新済みロケータ
  - LOB バインド変数

5 番目のトピックでは、LOB の基本操作について説明します。

- 大量の LOB データを効率よく読書きする。
- LOB をコピーする
  - 内部 LOB をコピーする
  - 外部 LOB をコピーする

- LOB を削除する
  - 内部 LOB を削除する
  - 外部 LOB を削除する
- LONG から LOB にデータをコピーする

最後のトピックでは、クライアント / サーバー環境でのパフォーマンスおよび最適化に関する事柄を検討します。

- オブジェクト・キャッシュの LOB
- LOB バッファリング・サブシステム
- 最大のパフォーマンスを引き出すためのユーザー・ガイドライン
- 可変幅文字データの処理

LOB とは何か？

次のようなアプリケーション・シナリオを想定してみます。

アプリケーション・シナリオ 1: ある法律事務所がデータベースを使ってある重要な訴訟事件の成果物を管理しようとしています。弁護士は、情報に X 線写真（イメージ・データ）、専門家の分析結果（文字テキスト）、証言（音声 / 映像）、図面（グラフィック）があることがわかっています。公判中は、コンピュータによる事件のシミュレーション（アニメーション）も利用するようになります。

アプリケーション・シナリオ 2: ある放送局がデータベースを使って特別番組の制作を管理しようとしています。番組の管理者は、この情報には通常、写真（イメージ・データ）、インタビュー（音声 / 映像）、音響効果（サウンドウェーブ・フォーム）、音楽（サウンドウェーブ・フォーム）、脚本（文字テキスト）があることがわかっています。デジタル化と記憶装置の技術が進んだおかげで、古い技術を使用した無声映画（映像）を組み込めることもわかっています。

アプリケーション・シナリオ 3: 海底の石油を探している地質調査団が、データベースを使ってそのプロジェクトを管理したいと思っています。プロジェクト管理者は、情報には、衛星画像（イメージ・データ）と複雑なオーバーレイ、図面（イメージ・データ）、ソナーの録音とそのグラフィック表示（サウンドウェーブ・フォームとイメージ・データ）、化学分析（イメージ・データと文字テキスト）があるとわかっています。プロジェクト展開中には、天候予測のコンピュータ・モデリング（文字テキストとイメージ・データ）も使うようになります。

これらのモデル・ケースは異なる分野のものですが、複数のメディアの管理がビジネス・アプリケーションでは当然になっていることは容易に理解できます。この章で取り上げたのは、Oracle8 では、LOB つまり 4 ギガバイトまでの RAW またはバイナリ・データ（グラフィック・イメージ、サウンドウェーブ・フォーム、ビデオ・クリップなど）、文字テキスト・データを格納できる**ラージ・オブジェクト**をサポートしているためです。

Oracle8 では、LOB を、データベースに対してどのような位置にあるかによって、内部 LOB と外部 LOB(BFILE) との 2 種類に分けています。単に LOB という場合は、内部 LOB と外部 LOB の両方を指します。LOB に格納されたデータを LOB の値と呼びます。

内部 LOB と外部 LOB(BFILE)

内部 LOB

内部 LOB は、その名の通り、領域を最適化し、効率よいアクセスを提供できるように、データベース表領域内に格納されます。内部 LOB はコピー・セマンティクスを使用し、サーバーのトランザクション・モデルで使用されます。また、内部 LOB は、トランザクションまたはメディアの故障が発生しても回復可能です。つまり、データベース・オブジェクトの使用に関するすべての ACID プロパティは、内部 LOB にも適用されるということです。内部 LOB 値に対する変更は、コミットまたはロールバックできます。

外部 LOB(BFILE)

外部 LOB は、BFILES とも呼ばれ、データベース表領域外にあるオペレーティング・システム・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。これらのオペレーティング・システム・ファイルはリファレンス・セマンティクスを使用します。これらのファイルはハード・ディスクまたは CD-ROM、PhotoCD、その他のデバイス上に配置できますが、1 つの LOB が複数のデバイス間にまたがることはできません。Oracle8 の SQL および PL/SQL では、データベース・サーバーのファイル・システムに存在する大規模なファイルに対する、読込み専用の、バイト・ストリーム入出力アクセスを可能にするために、SQL データ型 BFILE をサポートしています。Oracle Server は、基礎を形成するサーバーのオペレーティング・システムがこれらのファイルへのストリーム・モードのアクセスをサポートしている場合に限り、これらのファイルにアクセスできます。

警告： 外部 LOB はトランザクションには関係ないことに注意してください。LOB の整合性および耐久性は、オペレーティング・システムで管理されるファイル・システムによってサポートされなければなりません。

LOB と、LONG 型および LONG RAW 型との比較

LOB は、LONG 型および LONG RAW 型と類似していますが、次の点が異なります。

- 1 行に複数の LOB が可能。
- LOB は、ユーザー定義のデータ型（オブジェクト）の属性である可能性がある。
- LOB ロケータだけが表の列に格納される。BLOB データと CLOB データは、別々の表領域に格納され、BFILE データは、外部ファイルとして格納されます。
- LOB 列にアクセスすると、ロケータが戻る。
- LOB は、最大 4GB まで可能。BFILE の最大値は、オペレーティング・システムによって異なりますが、4GB までです。有効なアクセス可能範囲は、1 ～ $(2^{32}-1)$ です。
- LOB により、ランダムに、ピース単位でデータにアクセスおよび操作できる。

LOB の操作に使用するパッケージ

内部 LOB の値は、直接 SQL DML によって変更できます。変更は内部 LOB の全体に対して、または、内部 LOB の始め、中、終わりのいずれに対しても、OCI によってまたは PL/SQL DBMS_LOB API によって、ピース単位で実行できます。読込み目的で内部 LOB と外部 LOB の両方にアクセスすることもでき、内部 LOB には書き込むこともできます。

- OCI LOB インタフェースについては、6-59 ページの「LOB の操作に OCI を使用する」に簡単な説明があります。詳細は、『Oracle コール・インタフェース・プログラマーズ・ガイド』に記述されています。
- PL/SQL DBMS_LOB API については、6-62 ページの「DBMS_LOB パッケージ」に記述されています。

LOB データ型

内部 LOB データ型

内部 LOB のインスタンスを定義するには、次の 3 つの SQL データ型があります。

- BLOB。非構造化バイナリ（「ロー（raw）」）データで構成される値を持つ LOB。
- CLOB。Oracle8 データベース用に定義されたデータベース・キャラクタ・セットに対応する、1 バイトの固定幅の文字データで構成される値を持つ LOB。

- NCLOB。Oracle8 データベース用に定義された各国語キャラクタ・セットに対応する、固定幅のマルチバイト・キャラクタ・データで構成される値を持つ LOB。

可変幅文字データ（例 : JA16SJIS、JA16EUC）は BLOB および CLOB、NCLOB ではサポートされません。

関連項目： 6-53 ページの「可変幅文字データの処理」

外部 LOB データ型

外部 SQL の LOB データ型は 1 種類しかありません。

- BFILE。バイナリ (「ロー（raw）」) データで構成され、データベース表領域外のサーバー側オペレーティング・システム・ファイル内に格納される値を持つ LOB。

内部 LOB と外部 LOB を表に定義する

LOB を表に格納するには、2 通りの方法があります。

- LOB を表の列にする ― LOB が他のデータ・エンティティと「関連している」場合。
- LOB をオブジェクトの属性にする ― データ・エンティティ（つまりオブジェクト型）が 1 つまたは複数の LOB を属性として持っている場合です。

いずれの場合にも、SQL DDL が、表の LOB 列およびオブジェクト型の LOB 属性を定義するために使用されます。次に示す DDL コマンドで LOB を使用する際の詳細は、『Oracle8 Server SQL リファレンス』を参照してください。

- `CREATE TABLE` および `ALTER TABLE`
  - BLOB 列および CLOB 列、NCLOB 列、BFILE 列
  - 内部 LOB 列 / 属性用の LOB 記憶域句
- `CREATE TYPE` および `ALTER TYPE`
  - BLOB 属性および CLOB 属性、BFILE 属性（NCLOB はオブジェクト型属性には定義できないことに注意してください）。

次に示すコードの断片部分は、表 lob_table の作成を説明するものです。この例は、この項を通して使用します。

```
CREATE TABLE lob_table (
  key_value      INTEGER,
  b_lob          BLOB,
  c_lob          CLOB,
  n_lob          NCLOB,
  f_lob          BFILE);
```

内部 LOB 用の表領域と記憶特性を取り決める

表で LOB を定義するときは、各内部 LOB の表領域と記憶特性を明示的に指定できます。外部 LOB はデータベース中に格納されていないため、外部 LOB 用の特別な表領域や記憶特性はありません。

LOB データ・セグメントと LOB 索引に対して名前を指定すると、より直観的な作業環境が作成されます。LOB データ・ディクショナリ・ビューの USER_LOBS、ALL_LOBS、DBA_LOBS(『Oracle8 Server リファレンス・マニュアル』を参照) を問い合せるときは、非

直観的なシステムの名前ではなく、自分で選択した LOB データ・セグメント名および LOB 索引名を参照します。

LOB 列または LOB 属性に特有の記憶特性には、PCTVERSION、CACHE、NOCACHE、LOGGING、NOLOGGING、CHUNK、ENABLE/DISABLE STORAGE IN ROW などがあります。ほとんどのユーザーには、これらの記憶特性のデフォルトで十分です。LOB 記憶域を微調整するには、次のガイドラインを考慮する必要があります。

表領域と LOB 索引

LOB で最大のパフォーマンスを得るには、LOB を含む表に使用する記憶域とは別の記憶域を、表領域内で LOB 用に指定することが必要です。頻繁にアクセスする LOB が多数ある場合は、デバイスの競合を避けるため、それぞれの LOB の列 / 属性に別々の表領域を指定するとよいでしょう。

LOB 索引は、LOB 記憶域に強く対応付けられている内部構造体です。そのため、ユーザーは LOB 索引を削除して再作成してはいけません。LOB 索引は、ALTER INDEX 文では変更できませんが、ALTER TABLE 文では変更できます。また、LOB 索引を改名してはいけません。システムは LOB 記憶域句内のユーザーの指定に従って、どの表領域を LOB データと LOB 索引に使用するかを決めます。

- ユーザーが LOB データと LOB 索引のいずれにも表領域を指定しない場合、LOB データと LOB 索引の両方がその表の表領域を使用します。
- ユーザーが LOB データに表領域を指定し、LOB 索引には指定しない場合、LOB データと LOB 索引の両方が LOB データに指定された表領域を使用します。
- ユーザーが LOB 索引に表領域を指定し、LOB データには指定しない場合、LOB 索引は指定された表領域を使用し、LOB データはその表の表領域を使用します。
- ユーザーが LOB データと LOB 索引の両方に表領域を指定した場合、LOB データと LOB 索引はそれぞれ指定された表領域を使用します。

LOB 記憶域セグメントごとに別々の表領域を指定すると、表の表領域での競合を少なくできます。極端な場合には、表データ、LOB データ・セグメント、LOB 索引セグメントにそれぞれ別の表領域を使用すると便利なこともあります。このような方法は、ある LOB データに頻繁にアクセスするような場合には便利でしょう。通常は、2 つの表領域 (表データに 1 つ、LOB データと LOB 索引に 1 つ) を使用すれば十分です。

PCTVERSION

LOB が変更されると、前のバージョンの LOB 値の一貫性のある読込みをサポートするため、新しいバージョンの LOB ページが作成されます。

PCTVERSION では、使用するすべての LOB データのうち何パーセントを旧バージョンの LOB データ・ページに使用するかを指定します。旧バージョンの LOB データ・ページが、この PCTVERSION に指定した量を超えて LOB データ・ページを使用するとすぐに、Oracle はデータ・ページを再使用するよう旧バージョンに要求します。言い換えれば、

PCTVERSION は、全使用 LOB データ・ブロックのうち旧バージョンの LOB データが使用できるパーセントのことです。

デフォルト : 10(%) 最小値 : 0(%) 最大値 : 100(%)

PCTVERSION を概算する方法の 1 つは、PCTVERSION = (ある時点で更新済みの LOB のパーセント) * (LOB の 1 つが更新された際の更新済みの各 LOB パーセント) * (任意の時点で読み込まれている LOB のパーセント) に設定することです。基本的に、この考え方は、更新されたデータの一貫性のある読込みを行えるだけの LOB 記憶領域を確保しておくためのものです。

例 1: LOB を大量に読み込むと同時に、LOB の更新を時々行う。

PCTVERSION = 20% に設定します。

PCTVERSION をデフォルトの 2 倍の値に設定すると、旧バージョンのデータ・ページに使用できる空きページが増えます。大量の問合せをする場合、一貫性のある LOB の読込みが必要となるため、旧バージョンのデータ・ページを通常より多く使用できると便利です。Oracle は空きページを積極的に再使用しないため、LOB 記憶域は若干大きくなります。

例 2: LOB を作成し、1 度だけ書込みを行うが、その後は主に読込み専用で使用する。更新はめったに行わない。

PCTVERSION = 5%、またはそれ以下に設定してください。

LOB の更新をほとんど行わない、あるいは更新する部分が非常に少ない場合は、LOB データの旧バージョンに使用する領域を少なくします。既存の LOB が読込み専用とわかっている場合には、PCTVERSION を 0% に設定し、旧バージョンのデータがページを使用しないようにしても問題ありません。

CACHE / NOCACHE

同じ LOB データに頻繁にアクセスする場合は、LOB の CACHE オプションを使用してください。LOB データに 1 度しかアクセスしない、あるいはあまり頻繁にアクセスしない場合は、NOCACHE オプション (デフォルト) を使用してください。

LOGGING / NOLOGGING

LOB 使用時の [NO] LOGGING アプリケーションは、他の表操作の場合と同じです。通常、[NO]LOGGING 句の省略は、NO LOGGING、LOGGING のどちらも指定されておらず、表や表のパーティションのロギング属性のデフォルトが、その表領域のロギング属性に設定されていることを意味します。

LOB には、CACHE の設定によって他にも方法があります。

- [NO]LOGGING 句を省略し、CACHE を指定すると、LOGGING は、自動的にインプリメントされます（CACHE NOLOGGING を持つことができないため）。

- [NO]LOGGING 句を省略し、CACHE が未指定だと、プロセスのデフォルトは、表やパーティション表と同様になります。つまり、[NO]LOGGING 値は、LOB 値がある表領域から取得されます。

ただし、次の事柄を考慮しなければなりません。

- LOB では、常に LOB 索引ページに対して取消しが生成されます。LOGGING または NOLOGGING が設定されているかどうかにかかわらず、古い LOB データはバージョンごとに保存されるので、LOB では LOB データ・ページのロールバック情報（取消し）は生成されません。LOB について作成されるロールバック情報は、LOB の索引ページの変更専用なので、サイズが小さくなります。
- LOGGING が設定されている場合、Oracle では、LOB データ・ページの完全な再実行を生成します。NOLOGGING は、ユーザーがメディア回復を必要としない場合に使用されます。そのため、ディスク、テープ、記憶領域などのメディアに障害が発生した場合、変更のログは作成されていないので、ログから変更を回復できません。

  NOLOGGING の例として大量のロード、または挿入をあげます。たとえば、LOB にデータをロードしている場合、回復を必要とせず、ロードに失敗した場合にロードを簡単に再開始できるなら、LOB のデータ・セグメント保存領域の特性を NOCACHE NOLOGGING に設定してください。これにより、データの初期ロードのパフォーマンスが向上します。データのロードが完了すれば、ALTER TABLE を使用して LOB データ・セグメントの LOB 保存領域特性を変更し、CACHE または NOCACHE LOGGING など、通常の LOB 操作に対する特性を希望どおりに設定できます。

注意： CACHE を設定すると、LOGGING も行われます。

CHUNK

CHUNK には、一度にアクセスする LOB データのブロック数、つまり LOB の値への一度のアクセスで OCILobRead() または OCILobWrite()、DBMS_LOB.READ()、DBMS_LOB.WRITE() を介して読み込まれたり、書き込まれるブロックの数を設定します。たとえば、1 度に 1 ブロックの LOB データにしかアクセスしない場合には、CHUNK を 1 ブロックのサイズに設定します。たとえば、データベース・ブロック・サイズが 2K の場合には、CHUNK を 2K に設定します。

LOB の保存領域特性を明示的に指定する場合、LOB データ・セグメントの記憶領域の INITIAL および NEXT が CHUNK のサイズより大きいことを確認してください。たとえば、データベース・ブロックのサイズが 2K で、CHUNK に 8K を指定した場合、INITIAL および NEXT が 8K よりかなり大きいこと（少なくとも 16K）を確認してください。

他の方法 : INITIAL または NEXT、LOB CHUNK に値を指定する場合には、次のことを確認してください。

- CHUNK <= NEXT

および

- CHUNK <= INITIAL

ENABLE | DISABLE STORAGE IN ROW

ENABLE | DISABLE STORAGE IN ROW 句を使って、LOB をインライン（つまり行内）に保存するかどうかを指定します。これは、一度指定すると変更できない可能性があります。つまり、ENABLE STORAGE IN ROW から DISABLE STORAGE IN ROW に、あるいはその逆に変更できません。デフォルトは ENABLE STORAGE IN ROW です。

行内に保存される LOB データの最大サイズは、最大 VARCHAR サイズ（4000）です。この最大値には LOB 値と制御情報が含まれることに注意してください。ユーザーによって、LOB を行内に保存すると指定されている場合、LOB 値と制御情報が 4000 を超えると、LOB 値は自動的に行の外に保存されます。

このため、次のようなガイドラインが考えられます。LOB が小さい場合（つまり、<4000 バイト）、LOB データを行外に保存すると、パフォーマンスが低下します。ただし、LOB を行内に保存すると、行のサイズが大きくなります。これは、ユーザーが全表走査または複数行アクセス（範囲指定走査）、LOB 列以外の列への多数の UPDATE/SELECT など、多くの実表処理を行っている場合、パフォーマンスに影響します。LOB データが < 4000 にならないとわかっている場合、つまりすべての LOB が大きい場合には、次のような理由で、デフォルトを使用するのが最適と言えます。

(a) LOB データは、4000 より大きくなると自動的に行の外に移動する（LOB データはもともと大きいので、ここでは 4000 より大きい場合とする）。

(b) LOB データを行外に保存しても、制御情報は行内に保存されているので、パフォーマンスは多少よくなる。

内部 LOB を初期化する (SQL DML)

内部 LOB、つまり表の LOB 列、またはユーザー定義のオブジェクト型の LOB 属性を空または NULL にするように設定できます。表に格納された空の LOB は、ロケータを持つ長さ 0（ゼロ）の LOB です。空の LOB 列 / 属性に対し SELECT を実行すると、OCI ルーチンまたは DBMS_LOB ルーチン経由で LOB をデータに移入するのに使用できるロケータを入手できます。これについては、次に詳しく説明します。

かわりに、LOB 列（LOB 属性ではない）は、値に初期化することもできます。つまり、内部 LOB 属性は、LOB 属性が NULL または空以外の値には初期化できないという点で、LOB 列とは異なっています。次で説明するように、外部 LOB（つまり、BFILE）は、NULL またはファイル名に初期化できます。

たとえば、表 *lob_table* を作成するとします。

```
CREATE TABLE lob_table (
  key_value      INTEGER,
  b_lob          BLOB,
  c_lob          CLOB,
  n_lob          NCLOB,
  f_lob          BFILE);
```

次の SQL INSERT 文を使用して、LOB を初期化できます。

```
INSERT INTO lob_table VALUES (1001, EMPTY_BLOB(), NULL,
    EMPTY_CLOB(), NULL);
```

これによって、b_lob と n_lob の値が空値に、c_lob と f_lob が NULL に設定されます。

LOB を NULL に設定する

INSERT 文を実行するときに LOB データがない場合、または後に次のような SELECT 文を発行する場合は、行を挿入するときに LOB の値を NULL に設定すると便利です。

```
SELECT * FROM a_table WHERE a_lob_col != NULL;
```

または

```
SELECT * FROM a_table WHERE a_lob_col == NULL;
```

ただし、この方法を使うと、後に SQL UPDATE 文を発行して、NULL の LOB 列を EMPTY_BLOB()/EMPTY_CLOB() に設定するかまたは内部 LOB または外部 LOB のファイル名の値 ('abc' など) に NULL を設定しなければならない、という欠点があります。NULL の LOB に対して OCI または PL/SQL DBMS_LOB 関数をコールすることはできません。これらの関数は、ロケータがある場合にだけ機能します。LOB 列が NULL の場合は、行にロケータがありません。

内部 LOB を空に設定する

もう 1 つのオプションとして、INSERT 文で EMPTY_BLOB()/EMPTY_CLOB() 関数を使用することにより、LOB 値を空に設定できます。

```
INSERT INTO a_table values (empty_blob());
```

RETURNING 句を使用して（これによって、次の SELECT に必要な処理の往復が排除される)、その後すぐに OCI または PL/SQL DBMS_LOB 関数をコールして LOB にデータを移入するとさらによいでしょう。

関連項目： 6-55 ページの「EMPTY_BLOB() 関数と EMPTY_CLOB() 関数」

外部 LOB にアクセスする (SQL DML)

ディレクトリ・オブジェクト

DIRECTORY オブジェクトによって、Oracle8 Server での BFILE へのアクセスおよび使用を管理できます (『Oracle8 Server リファレンス・マニュアル』を参照してください)。DIRECTORY によって、アクセスするファイルが置かれているサーバーのファイル・システムの物理ディレクトリの論理別名が指定されます。サーバーのファイル・システム内にあるファイルへのアクセスは、DIRECTORY オブジェクトについて必要なアクセス権限が付与されている場合に限り、実行できます。

また、DIRECTORY オブジェクトによって、ファイルの位置を柔軟に管理できるようになり、アプリケーション内の物理ファイルの絶対パス名のハードコード化が強制されません。DIRECTORY 別名は、BFILE ロケータの初期化の際に、BFILENAME() 関数 (SQL および PL/SQL) または OCILobFileSetName()(OCI) で使用されます。

警告：　Oracle では、指定したディレクトリとパス名が実際に存在するか検証されません。オペレーティング・システムで有効なディレクトリを指定するよう注意してください。オペレーティング・システムがパス名を大 / 小文字区別している場合は、必ず正しい形式でディレクトリを指定してください。最後のスラッシュは指定する必要はありません（たとえば、/tmp/ ではなく、単に /tmp）。

BFILENAME() を使用して BFILES を初期化する

オペレーティング・システム・ファイルを BFILE に関連付けるために、まず、オペレーティング・システム・ファイルへのフルパス名の別名である DIRECTORY オブジェクトを作成しなければなりません。

関連項目： 6-14 ページの「ディレクトリ・オブジェクト」

既存のオペレーティング・システム・ファイルを特定の表の関連のあるデータベース・レコードに対応付けるには、Oracle8 の SQL DML を使用します。BFILE の列がサーバーのファイル・システム内の既存ファイルを参照するように、BFILE を初期化するには、SQL INSERT 文を使用できます。また、BFILE の参照先を変更するには、SQL UPDATE 文を使用できます。BFILENAME() 関数を使用して、BFILE を NULL に初期化してからオペレーティング・システム・ファイルを参照するように更新することもできます。OCI ユーザーは、OCILobFIleSetName() を使用して、INSERT 文の VALUES 句で使用される BFILE ロケータ変数を初期化できます。

たとえば、次の文では、ファイル image1.gif と image2.gif を 21 と 22 の key_value を持つレコードにそれぞれ関連付けます。「IMG」は、image1.dif と image2.dif が格納される物理ディレクトリを示す DIRECTORY オブジェクトです。

```
INSERT INTO lob_table VALUES
     (21, NULL, NULL, NULL, BFILENAME('IMG', 'image1.gif'));
 INSERT INTO lob_table VALUES
     (22, NULL, NULL, NULL, BFILENAME('IMG', 'image2.gif'));
```

次の UPDATE 文によって、key_value が 22 の行について、ターゲット・ファイルは image3.gif に変更されます。

```
UPDATE lob_table SET f_lob = BFILENAME('IMG', 'image3.gif')
    WHERE key_value = 22;
```

BFILENAME() は、BFILE 列が外部ファイルを参照するように、BFILE を初期化するのに使用する組込み関数です。

関連項目： 6-56 ページの「BFILENAME() 関数」

SQL DML を使用して物理ファイルがレコードにいったん対応付けられると、BFILE に対するその後の読取り操作は、PL/SQL の DBMS_LOB パッケージと OCI を使用して実行できます。ただし、これらのファイルは、読込み専用なので、BFILES によって更新したり削除したりすることはできません。

BFILE に対する参照ベースの方法により、同一レコード内、または異なるレコード内に、同じファイルを参照する複数の BFILE 列を持つことが可能になりました。たとえば、次の UPDATE 文では、lob_table 内で key_value21 の行を持つ BFILE が、key_value が 22 の行と同じファイルを参照するように設定します。

```
UPDATE lob_table
   SET f_lob = (SELECT f_lob FROM lob_table WHERE key_value = 22)
   WHERE key_value = 21;
```

DIRECTORY 名の指定

Oracle8 における DIRECTORY オブジェクトの命名規則は、表や索引の命名規則と同じです。つまり、通常の識別子は大文字で解釈されますが、区切り識別子はそのまま解釈されます。たとえば、次のような文があるとします。

```
CREATE DIRECTORY scott_dir AS '/usr/home/scott';
```

この文により、名前が「SCOTT_DIR」(大文字) のディレクトリ・オブジェクトが作成されます。ただし、次の文のように、DIRECTORY 名に区切り識別子が使用されるとします。

```
CREATE DIRECTORY "Mary_Dir" AS '/usr/home/mary';
```

この文では、ディレクトリ・オブジェクトの名前は「Mary_Dir」になります。BFILENAME() をコールするときは、「SCOTT_DIR」と「Mary_Dir」を使用します。たとえば、次のとおりです。

```
BFILENAME('SCOTT_DIR', 'afile')
BFILENAME('Mary_Dir', 'afile')
```

BFILE セキュリティ

この項は、BFILE セキュリティ・モデルおよびそれに関連する SQL DDL と DML を紹介します。Oracle 8.0 における BFILE セキュリティの主要な機能を次に示します。

- DIRECTORY オブジェクトに対して、CREATE および REPLACE/ALTER を実行する SQL DDL
- DIRECTORY に対して、READ システムおよびオブジェクト権限を GRANT および REVOKE する SQL DML

所有権と権限

DIRECTORY は、システム所有のオブジェクトです。システム所有のオブジェクトの詳細は、『Oracle8 Server SQL リファレンス』を参照してください。Oracle8 では、2 つの新しいシステム権限をサポートしており、これらは DBA アカウントだけに付与されます。

- CREATE ANY DIRECTORY ― ディレクトリ・オブジェクトの作成または変更
- DROP ANY DIRECTORY ― ディレクトリ・オブジェクトの削除

DIRECTORY オブジェクトに対する READ 権限によって、そのディレクトリ下に置かれたファイルを読み取ることができます。DIRECTORY オブジェクトの作成者には、READ 権限が自動的に付与されます。GRANT オプション付きの READ 権限を付与されている場合は、その権限を他のユーザーやロールに付与して、自分の権限ドメインに追加することもできます。

READ 権限は、DIRECTORY オブジェクトだけに定義されることに注意してください。DIRECTORY オブジェクトが表す物理ディレクトリには、Oracle Server プロセスに対応するオペレーティング・システム権限（この場合は読取り）がない場合もあります。物理ディレ

クトリの存在の有無、および Oracle Server プロセス用に、ファイルおよびディレクトリ、ディレクトリへのパスの読取り許可が使用可能になっているかどうかを確認することは、DBA の責務です。また、データベース・ユーザーによってファイル・アクセスが行われている間、ディレクトリと読取り許可を確実に使用可能な状態に保つことも、DBA の責務です。

Oracle8 Server の場合、権限とは、ディレクトリ内のファイルからの読込みが行えることを意味します。これらの権限は、実際のファイル操作時に、PL/SQL の DBMS_LOB パッケージおよび OCI API によってチェックおよび施行されます。

警告： CREATE ANY DIRECTORY 権限と DROP ANY DIRECTORY 権限によって、サーバーのファイル・システムが全データベース・ユーザーにさらされる可能性があるため、DBA は、不慮または故意のセキュリティ違反を防止するため、一般のデータベース・ユーザーに対するこれらの権限の付与は慎重に行わなければなりません。

BFILE セキュリティ用の SQL DDL

ディレクトリ・オブジェクトを作成および置換、削除する、次の SQL DDL コマンドの詳細は、『Oracle8 Server SQL リファレンス』を参照してください。

- `CREATE DIRECTORY`
- `DROP DIRECTORY`

BFILE セキュリティ用の SQL DML

BFILE に対してセキュリティを提供する、次の SQL DML コマンドの詳細は、『Oracle8 Server SQL リファレンス』を参照してください。

- GRANT（システム権限）
- GRANT（オブジェクト権限）
- REVOKE（システム権限）
- REVOKE（オブジェクト権限）
- AUDIT（新規文）
- AUDIT（スキーマ・オブジェクト）

ディレクトリのカタログ・ビュー

カタログ・ビューは、ディレクトリ・オブジェクト用に提供され、これによってユーザーは、オブジェクトの名前、およびそれに対応するパスと権限を見ることができます。サポートされるビューは次のとおりです。

- ALL_DIRECTORIES (OWNER, DIRECTORY_NAME, DIRECTORY_PATH)

  このビューによって、ユーザーがアクセスできる全ディレクトリが記述されます。

- DBA_DIRECTORIES(OWNER, DIRECTORY_NAME, DIRECTORY_PATH)

  このビューによって、全データベースについて指定されたディレクトリがすべて記述されます。

DIRECTORY 使用のガイドライン

Oracle8 における DIRECTORY 機能の主な目的は、サーバーのファイル・システム内の大きなファイルへのアクセスを DBA が管理する上で、単純で柔軟性があり、他操作の妨害にならず、しかも安全性の高いメカニズムを提供することです。しかし、この目的を実現するには、DBA がディレクトリ・オブジェクトの使用時に以下のガイドラインに従うことが非常に重要です。

- DIRECTORY は、Oracle データ・ファイルおよび制御ファイル、ログ・ファイル、その他のシステム・ファイルを含む物理ディレクトリにマップしないこと。これらのファイルを（不意にまたはなんらかの理由で）変更した場合、データベースまたはサーバーのオペレーティング・システムを破壊する恐れがあります。

- CREATE ANY DIRECTORY などのシステム権限（DBA にあらかじめ付与される）は、細心の注意を払って使用し、他のユーザーへ無分別に付与しないこと。多くの場合、データベース管理者だけが、これらの権限を持ちます。

- ディレクトリ・オブジェクトへの権限を他のユーザーへ付与する場合は、注意すること。権限をユーザーへ付与するときは、WITH GRANT OPTION 句の使用方法にも注意すること。

- DIRECTORY オブジェクトを、データベース運用中に、勝手に削除したり、置換したりしないこと。DIRECTORY オブジェクトを削除または置換してしまった場合、このディレクトリ・オブジェクトに関連する全ファイルでの全セッションの DBMS_LOB または OCI 操作が失敗します。さらに、これらのファイルを正常にクローズする前に、DROP コマンドや REPLACE コマンドを実行した場合は、プログラムにおけるこれらのファイルへの参照が失われ、これらのファイルに関連するシステム・リソースも、セッションを停止するまで解放されません。

  PL/SQL ユーザーに残される唯一の手段は、たとえば、DBMS_LOB FILECLOSEALL() をコールするプログラム・ブロックを実行しファイル操作を再開するか、またはセッションを完全に終了するかのいずれかである。したがって、これらのコマンドを使用する際は、細心の注意を払い、できればメンテナンスのダウン時間中に使用すること。

  関連項目： 6-77 ページの「DBMS_LOB.FILECLOSEALL() プロシージャ」

- 同様に、REVOKE 文を使用してユーザーのディレクトリ権限を取り消した場合、ユーザーのセッションに従属するファイルに対する後続の操作は失敗する。ユーザーは、権

限を再獲得してファイルをクローズするか、またはセッションで FILECLOSEALL() を実行し、ファイル操作を再開しなければならない。

一般的に、ファイル・アクセスの管理に DIRECTORY オブジェクトを使用することは、オペレーティング・システム・レベルでのシステム管理作業の延長です。なんらかの計画を立てることによって、Oracle プロセス用の読取り権限を持つ適切なディレクトリへファイルを論理的に編成し、これらの物理ファイルへマップする READ 権限とともに DIRECTORY オブジェクトを作成し、特定のデータベース・ユーザーにこれらのディレクトリへのアクセス権限を付与できます。

BFILE の最大オープン数

1 回のセッションにつき同時にオープンできる BFILE の数には制限があります。最大数は、新しい初期化パラメータである SESSION_MAX_OPEN_FILES パラメータによって指定されます。

SESSION_MAX_OPEN_FILES は、1 つのセッションで同時にオープンできるファイル数の上限を定義します。このパラメータのデフォルト値は 10 です。つまり、デフォルト値を使用していれば、1 回のセッションにつき最大 10 ファイルを同時にオープンできます。データベース管理者は、init.ora ファイルでこのパラメータの値を変更できます。たとえば次のとおりです。

```
SESSION_MAX_OPEN_FILES=20
```

MTS モードにおける BFILE

Oracle8 リリース 8.0 では、MTS モードでの BFILE のセッション移行をサポートしていません。これは、オープンされた BFILE に対する操作が、MTS サーバーへのコール終了後も続けられることを意味します。BFILE 操作が含まれるセッションは、1 台の共有サーバーに限定する必要があり、サーバー間での移行はできません。

プログラム終了後に BFILE をクローズする

PL/SQL プログラム・ブロックまたは OCI プログラムが正常に終了するかまたは異常終了した後で、オープンしているファイル (1 つまたは複数) をクローズすることはお客様の責務となります。たとえば、すべての DBMS_LOB FILEOPEN コールについて、対応する DBMS_LOB FILECLOSE コールが必要です。PL/SQL ブロックまたは OCI プログラムの終了前、およびエラーの発生時に、オープン・ファイルをクローズしなければなりません。例外または異常終了が発生する前にオープンされていたファイルをクローズするために、例外ハンドラを用意する必要があります。

これが実行されなかった場合、Oracle Server ではこれらのファイルはクローズされていないと認識され、クローズされていないファイルの数が SESSION_MAX_OPEN_FILES 値を超えた場合に、そのセッションではそれ以上のファイルをオープンできなくなります 。すべてのオープン・ファイルをクローズするには、FILECLOSEALL コールを使用します。

関連項目： PL/SQL プログラミングの詳細は、6-65 ページの「DBMS_LOB の一般的な使用上の注意」を参照してください。

LOB 値とロケータ

LOB 値のインライン記憶域

LOB に格納されたデータを LOB の値と呼びます。内部 LOB の値は、他の行データとインラインで格納できる場合とできない場合とがあります。内部 LOB の値が約 4000 バイトより小さい場合は、その値はインラインで格納されます。それよりも大きい場合は、行の外に格納されます。LOB はオブジェクトとして大きくなりがちなので、アプリケーションで「小さい」LOB と「大きい」LOB が混在する場合には、インライン記憶域が適切です。

これまでにも説明したとおり (6-12 ページの「ENABLE | DISABLE STORAGE IN ROW」)、LOB 値はいったん約 4000 バイトを超えると、自動的に行の外に出されます。

LOB ロケータ

内部 LOB の値が格納されている場所に関わらず、ロケータが行内に格納されます。LOB ロケータは、LOB 値の実際の位置へのポインタとして考えることができます。BFILE ロケータが外部 LOB を示すロケータであるのに対して、LOB ロケータは内部 LOB を示すロケータです。単にロケータという場合は、LOB ロケータと BFILE ロケータの両方を指します。

内部 LOB ロケータ

内部 LOB については、データベース表領域に格納される LOB の値に対するロケータが LOB 列に格納されます。ある行におけるそれぞれの LOB 列 / 属性は固有の LOB ロケータと、データベース表領域内に格納された LOB 値を持っています。

外部 LOB ロケータ (BFILE ロケータ)

BFILES では、値はサーバー側のオペレーティング・システム・ファイル、つまりデータベースの外部に格納されます。そのファイルを参照する BFILE ロケータは、行内に格納されます。DBMS_LOB FILEOPEN() で使用される BFILE ロケータ変数 (たとえば L1) が、別のロケータ変数 (たとえば L2) に割り当てられた場合、L1 および L2 はどちらも同じファイルを参照します。つまり、BFILE 列を持つ 2 つの行は、同じファイルを参照することも、2 つの異なるファイルを参照することもできます。このことは、開発者が慎重であれば利点にもなりますが、そうでなければ落とし穴にもなり得ます。

PL/SQL または OCI プログラム内の BFILE ロケータ変数は、他の自動変数と同じように機能します。ファイル操作に関しては、最も標準的なプログラミング言語の標準 I/O の一部として利用できるファイル記述子と同じように機能します。BFILE ロケータを定義および初期化し、このロケータによって参照されるファイルをオープンした場合は、このファイルをクローズするまでの操作はすべて、このロケータまたはこのロケータのローカル・コピーを使用して、同じプログラム・ブロック内から実行しなければならないことを意味します。

BFILE ロケータ変数は、スカラーと同様、他のプロシージャまたはメンバー・メソッド、外部関数コールアウトのパラメータとして使用できます。ただし、PL/SQL および OCI プログラムでは、ファイルのオープンおよびクローズは、同じネスト・レベルにある、同じプログラム・ブロックから行うことをお薦めします。

LOB ロケータの操作

LOB 列 / 属性がロケータを含むように設定する

内部 LOB へのデータの書込みを始める前に、LOB 列 / 属性を非 NULL に、つまりロケータを含むようにしなければなりません。同様に、BFILE 値へのアクセスを開始する前に、BFILE 列 / 属性を非 NULL にしなければなりません。

- 内部 LOB については、BLOB については EMPTY_BLOB() 関数、CLOB と NCLOB の場合は EMPTY_CLOB() 関数を使用して、INSERT/UPDATE 文で内部 LOB を空に初期化することによって、これを実現できる。

  関連項目： 6-55 ページの「EMPTY_BLOB() 関数と EMPTY_CLOB() 関数」

- 外部 LOB については、BFILENAME() 関数を使用することによって、外部ファイルが参照するように BFILE 列を初期化できる。

  関連項目： 6-56 ページの「BFILENAME() 関数」

EMPTY_BLOB() または EMPTY_CLOB() 関数の起動事体では例外は発生しません。ただし、空に設定された LOB ロケータを使用して、PL/SQL DBMS_LOB または OCI ルーチンで LOB 値をアクセスまたは操作すると、例外が発生します。空の LOB ロケータが有効となるのは、INSERT 文の VALUES 句および UPDATE 文の SET 句などです。

次の INSERT 文では、次のことを行います。

- b_lob を NULL に設定する
- c_lob に文字列「abcde」を移入する
- n_lob を NULL に設定する
- 論理ディレクトリ「SCOTT_DIR」の下に配置されたファイル「scott.dat」を参照するよう、f_lob を初期化する (『Oracle8 Server リファレンス・マニュアル』の「CREATE DIRECTORY コマンド」を参照してください)。文字列は、インスタンス用のデフォルトのキャラクタ・セットを使用して挿入されます。

```
INSERT INTO lob_table VALUES (1002, NULL 'abcde',
    NULL,BFILENAME('SCOTT_DIR', 'scott.dat'));
```

同様に、列が LOB 属性を持つオブジェクトである表 person_objcol_table については、次のように、LOB 属性を NULL に初期化、または空に設定できます。

```
INSERT INTO person_objcol_table VALUES (1001,person_type
    ('Scott', EMPTY_CLOB(), EMPTY_BLOB(),
    BFILENAME('SCOTT_DIR', 'scott.dat')));
```

ロケータによって LOB にアクセスする

LOB を SELECT する LOB に対して SELECT を実行すると、LOB 値のかわりにロケータが戻されます。次の部分的 PL/SQL では、b_lob の LOB ロケータを選択し、プログラム・ブロック内で定義された PL/SQL ロケータ変数 image1 内に、それを配置します。LOB 値の操作に PL/SQL の DBMS_LOB 関数を使用する場合は、ロケータを使用して LOB を参照します。

```
DECLARE
    image1       BLOB;
    image_no     INTEGER := 101;
BEGIN
    SELECT b_lob INTO image1 FROM lob_table
        WHERE key_value = image_no;
    DBMS_OUTPUT.PUT_LINE('Size of the Image is: ' ||
        DBMS_LOB.GETLENGTH(image1));
    -- more LOB routines
END;
```

OCI の使用時には、ロケータは LOB 値の操作に使用されるロケータ・ポインタにマップされます。OCI LOB インタフェースの簡単な説明は、6-59 ページの「LOB の操作に OCI を使用する」を、詳しい説明は、『Oracle 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。

内部 LOB を更新する前にロックする PL/SQL DBMS_LOB パッケージまたは OCI を使用して LOB を更新する前に、LOB を含む行をロックしなければなりません。SQL INSERT 文と UPDATE 文は暗黙的に行をロックしますが、ロックは SQL プログラムおよび PL/SQL プログラム内で SQL SELECT FOR UPDATE 文を使用するか、OCI プログラム内で OCI ピンまたはロック機能を使用して明示的に実行します。

読取り一貫性のあるロケータ

Oracle では、他のデータベースの読取りおよび更新処理と同様に、LOB に対しても同様の読取り一貫性メカニズムを提供しています（読取り一貫性の一般的な説明は *Oracle8 Server 概要 <Default ¶Font>* を参照）。ただし、LOB ロケータの場合、読取り一貫性にはいくつかの特別なアプリケーションがあるので、正しく理解しておく必要があります。

SELECT されたロケータは、FOR UPDATE 句の有無にかかわらず、読取り一貫性のあるロケータとなり、LOB 値がそのロケータによって更新されるまでは、読取り一貫性のあるロケータとして存在します。読取り一貫性のあるロケータには、SELECT 実行時点のスナップショット環境が含まれています。

これにはやや複雑な意味があります。たとえば、SELECT を実行して読取り一貫性のあるロケータを作成したとします。L1 によって内部 LOB 値を読み取るとき、SELECT 文に FOR UPDATE が含まれていても、SELECT 文実行時点の LOB 値が読み取られます。さらに、同じトランザクションの別のロケータ (L2) によって LOB 値が更新されても、L1 では L2 の更新が認識されません。また、L1 では、別のトランザクションによってコミットされた LOB の更新も認識されません。

さらに、読取り一貫性のあるロケータ L1 が別のロケータ L2 にコピーされた場合（たとえば、2 つのロケータ変数の PL/SQL 割当て－ L2:= L1 によって）、L2 は L1 と同様に読取り一貫性のあるロケータとなり、読み取られるデータは L1 に対する SELECT 実行時点におけるデータです。

複数のローケータが存在することを利用して、LOB 値の異なる変換にアクセスできます。ただし、この場合、どのロケータでどの値にアクセスしているかが常にわかっているように注意しなければなりません。次のコードは、読取り一貫性と更新の関係を簡単な例で示しています。

先に定義した lob_table と PL/SQL を使用して、clob_selected および clob_updated、clob_copied の 3 つの CLOB ロケータを作成します。

- SELECT INTO の最初の実行時には (t1)、clob の値がロケータ clob_selected に対応付けられます。
- 次の実行時には (t2)、c_lob の値はロケータ clob_updated に対応付けられます。t1 と t2 で c_lob の値は変わらないため、clob_selected と clob_updated の両方のロケータが、違う時点でのスナップショットを反映しているにもかかわらず、事実上同じ値の、読取り一貫性のあるロケータとなります。
- 3 回目の実行時 (t3) には、clob_selected の値が clob_copied にコピーされます。この時点で、3 つのロケータが同じ値になります。例では、一連の dbms_lob 読込みコールがこのことを示しています。
- この時点で (t4)、プログラムでは dbms_lob.write を使用して clob_updated が値を変更され、dbms_lob.read により新規の値がわかります。
- ただし、clob_selected(t5) を介した値の dbms_lob.read により、これが読取り一貫性のあるロケータであることがわかり、SELECT の発行時と同じ値が参照され続けます。
- 同様に、*clob_copied* (t6) を介した値の dbms_lob.read により、これが読取り一貫性のあるロケータであることがわかり、*clob_selected* と同じ値が参照され続けます。

読取り一貫性のあるロケータの例

```
INSERT INTO lob_table
   VALUES (1, NULL, 'abcd', NULL, NULL);

COMMIT;

DECLARE
  num_var           INTEGER;
```

```
  clob_selected     CLOB;
  clob_updated      CLOB;
  clob_copied       CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  write_amount      INTEGER;
  write_offset      INTEGER;
  buffer            VARCHAR2(20);

BEGIN
  -- At time t1:
  SELECT c_lob INTO clob_selected
     FROM lob_table
     WHERE key_value = 1;

  -- At time t2:
  SELECT c_lob INTO clob_updated
     FROM lob_table
     WHERE key_value = 1
     FOR UPDATE;

  -- At time t3:
  clob_copied := clob_selected;
  -- After the assignment, both the clob_copied and the
  -- clob_selected have the same snapshot as of the point in time
  -- of the SELECT into clob_selected

  -- Reading from the clob_selected and the clob_copied will
  -- return the same LOB value. clob_updated also sees the same
  -- LOB value as of its select:
  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'
```

```
  -- At time t4:
  write_amount := 3;
  write_offset := 5;
  buffer := 'efg';
  dbms_lob.write(clob_updated, write_amount, write_offset,
       buffer);

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'

  -- At time t5:
  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t6:
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'
END;
/
```

更新済みロケータ

内部 LOB の値を LOB ロケータ (L1) によって更新するとき、L1(つまり、ロケータ自体) は更新され、ロケータ L1 によって LOB 値に対する操作が完了した時点での、現行スナップショット環境を持ちます。これによって、L1 は、更新済みロケータと呼ばれます。この操作によって、LOB 値に対して行った変更を、同じロケータ L1 による次の読取り時に見ることができます。

注意： 単に LOB 値を読み取るためにロケータを使用した場合、ロケータ内のスナップショット環境は更新されません。PL/SQL DBMS_LOB パッケージまたは OCI LOB API によってロケータを介して LOB 値を変更した場合に限り、更新されます。

別のトランザクションによってコミットされた更新は、そのトランザクションが読取りコミットされたトランザクションで、他のトランザクションがコミットされた後に L1 を使用して LOB 値を更新する場合にだけ、L1 によって認識されます。

注意： 内部 LOB の値を更新すると、変更は常に最新の LOB 値に対して行われます。

OCI LOB API または PL/SQL DBMS_LOB パッケージによって内部 LOB の値を更新することは、LOB 値を更新して、新しい LOB を参照するロケータを再選択するということです。

SQL による LOB 値の更新は、UPDATE 文と同じことです。UPDATE 文によって行われた変更をロケータが認識できるようにするには、LOB ロケータを再選択するか、UPDATE 文の RETURNING 句を使用するかのいずれかをユーザーが実行しなければなりません (『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照)。LOB ロケータを再選択せず RETURNING 句も使用しない場合、LOB 値が更新されていない状態での最新の値を読み取ることになります。このため、OCI での SQL DML と DBMS_LOB のピース単位操作を混合しないようにしてください。

始めに定義した lob_table を使用して CLOB ロケータ clob_selected を作成します。

- SELECT INTO の最初の実行時には (t1)、c_lob の値がロケータ clob_selected に対応付けられます。
- 次の実行時には (t2)、c_lob の値が clob_selected ロケータをバイパスして、SQL UPDATE コマンドによって変更されます。ロケータは元の SELECT 時点での LOB 値を参照します。つまり、ロケータは SQL UPDATE コマンドによって実行された更新を認識しません。例では、後続の dbms_lob.read コールがこのことを示しています。
- 3 回目の実行時 (t3) には、LOB 値がロケータ clob_selected に再選択されます。これによって、ロケータは最新のスナップショット環境に更新され、先の SQL UPDATE コマンドによって行われた変更を認識できるようになります。このため、次の dbms_lob 読込みでは、LOB 値が空である (データがない) ことを理由にエラーが戻されます。

SQL DML と DBMS_LOB の混合の影響の例

```
INSERT INTO lob_table VALUES (1, NULL, 'abcd', NULL, NULL);
COMMIT;

DECLARE
  num_var           INTEGER;
  clob_selected     CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  buffer            VARCHAR2(20);

BEGIN

  -- At time t1:
  SELECT c_lob INTO clob_selected
  FROM lob_table
  WHERE key_value = 1;
```

```
  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t2:
  UPDATE lob_table SET c_lob = empty_clob()
      WHERE key_value = 1;
  -- although the most current current LOB value is now empty,
  -- clob_selected still sees the LOB value as of the point
  -- in time of the SELECT

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset,
     buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t3:
  SELECT c_lob INTO clob_selected FROM lob_table WHERE
       key_value = 1;
  -- the SELECT allows clob_selected to see the most current
  -- LOB value

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  -- ERROR: ORA-01403: no data found
END;
/
```

警告： 全体として、同じ **LOB** を別のロケータで更新しないことをお薦めします。同じ **LOB** 値を変更するのに **1** つのロケータだけを使えば、多くの危険を回避できます。

先に定義した lob_table を使用して、clob_updated と clob_copied の 2 つの CLOB ロケータを作成します。

- SELECT INTO の最初の実行時には (t1)、c_lob の値がロケータ clob_updated に対応付けられます。

- 次の実行時には (t2)、clob_updated の値は clob_copied にコピーされます。この時点では、両方のロケータが同じ値を参照します。例では、一連の dbms_lob 読込みコールがこのことを示しています。
- この時点で (t3)、プログラムでは dbms_lob.write を使用して clob_updated の値が変更され、dbms_lob.read により新規の値がわかります。
- ただし、clob_copied(t4) を介した値の dbms_lob.read では、clob_updated から割り当てられた時点 (t2) の LOB の値を参照していることがわかります。
- clob_updated が clob_copied に割り当てられた時点 (t5) で初めて、clob_copied は clob_updated による更新があったことを認識します。

更新済み LOB ロケータの例

```
INSERT INTO lob_table
   VALUES (1, NULL, 'abcd', NULL, NULL);

COMMIT;

DECLARE
  num_var          INTEGER;
  clob_updated     CLOB;
  clob_copied      CLOB;
  read_amount      INTEGER; ;
  read_offset      INTEGER;
  write_amount     INTEGER;
  write_offset     INTEGER;
  buffer           VARCHAR2(20);
BEGIN

-- At time t1:
  SELECT c_lob INTO clob_updated FROM lob_table
      WHERE key_value = 1
      FOR UPDATE;

  -- At time t2:
  clob_copied := clob_updated;
  -- after the assign, clob_copied and clob_updated see the same
  -- LOB value

  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
```

```
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'


  -- At time t3:
  write_amount := 3;
  write_offset := 5;
  buffer := 'efg';
  dbms_lob.write(clob_updated, write_amount, write_offset,
        buffer);

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'


  -- At time t4:
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'


  -- At time t5:
  clob_copied := clob_updated;

  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcdefg'
END;
/
```

LOB バインド変数

別の内部 LOB を更新するためのソースとして LOB ロケータが使用される場合 (SQL INSERT 文または UPDATE 文、DBMS_LOB.COPY ルーチンなどにおいて)、ソース LOB ロケータ内のスナップショット環境によって、ソースとして使用される LOB 値が決定されます。ソース・ロケータ（たとえば L1）が読取り一貫性のあるロケータの場合、L1 の SELECT 実行時点の LOB 値が使用されます。ソース・ロケータ（たとえば L2）が更新済みロケータの場合、更新時における L2 のスナップショット環境に対応する LOB 値が使用されます。

先に定義した lob_table を使用して、ロケータとして clob_selected および clob_updated、clob_copied の 3 つの CLOB ロケータを作成します。

- SELECT INTO の最初の実行時には (t1)、c_lob の値がロケータ clob_updated に対応付けられます。
- 次の実行時には (t2)、clob_updated の値は clob_copied にコピーされます。この時点では、両方のロケータが同じ値を参照します。
- 次に (t3)、プログラムでは dbms_lob.write を使用して clob_updated の値が変更され、dbms_lob.read により新規の値がわかります。
- ただし、clob_copied(t4) を介した値の dbms_lob.read により、clob_copied では clob_updated による変更を参照しないことがわかります。
- このため (t5)、clob_copied が INSERT 文の値のソースとして使われる場合、clob_copied と対応付けられた値が挿入されます（つまり、clob_updated による新規の変更は反映されない）。これは、次の、挿入された値の dbms_lob.read によってわかります。

PL/SQL 変数を使用した LOB の更新の例

```
INSERT INTO lob_table
   VALUES (1, NULL, 'abcd', NULL, NULL);

COMMIT;

DECLARE
  num_var           INTEGER;
  clob_selected     CLOB;
  clob_updated      CLOB;
  clob_copied       CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  write_amount      INTEGER;
  write_offset      INTEGER;
  buffer            VARCHAR2(20);
BEGIN

  -- At time t1:
  SELECT c_lob INTO clob_updated FROM lob_table
      WHERE key_value = 1
      FOR UPDATE;

  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'


  -- At time t2:
  clob_copied := clob_updated;
```

```
  -- At time t3:
  write_amount := 3;
  write_offset := 5;
  buffer := 'efg';
  dbms_lob.write(clob_updated, write_amount, write_offset,
       buffer);

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'
  -- note that clob_copied doesn't see the write made before
  -- clob_updated


  -- At time t4:
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t5:
  -- the insert uses clob_copied view of the LOB value which does
  -- not include clob_updated changes
  INSERT INTO lob_table values (2, NULL, clob_copied, NULL,
      NULL) RETURNING c_lob INTO clob_selected;

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'
END;
/
```

LOB ロケータは複数のトランザクションにまたがることはできない

DBMS_LOB または OCI、SQL INSERT 文、UPDATE 文を経由して、LOB ロケータによって内部 LOB の値を更新すると、読取り一貫性のあるロケータが更新済みロケータに変わります。さらに、INSERT 文や UPDATE 文によって、トランザクションが自動的に開始され、行がロックされます。これがいったん発生すると、ロケータを現行トランザクション以外では使用できなくなる可能性があります。つまり、LOB ロケータを複数のトランザクションにまたがって使用はできません。

先に定義した lob_table を使用して、clob_updated という CLOB ロケータを作成します。

- SELECT INTO の最初の実行時には (t1)、c_lob の値がロケータ clob_updated に対応付けられます。
- 2 番目の操作で (t2)、dbms_lob.write コマンドを使用して clob_updated の値が変更され、dbms_lob.read により新規の値がわかります。
- COMMIT 文 (t3) により現行のトランザクションが終了します。
- トランザクションを終了すると (t4)、clob_updated ロケータが別のトランザクション（コミット済み）を参照することになるため、次の dbms_lob.read 操作に失敗します。これは、エラーが戻されることによって通知されます。この LOB ロケータを、さらに dbms_lob（および OCI）で使うには、もう一度 SELECT しなければなりません。

トランザクションにまたがらないロケータの例

```
INSERT INTO lob_table
     VALUES (1, NULL, 'abcd', NULL, NULL);
COMMIT;

DECLARE
  num_var          INTEGER;
  clob_updated     CLOB;
  read_amount      INTEGER;
  read_offset      INTEGER;
  write_amount     INTEGER;
  write_offset     INTEGER;
  buffer           VARCHAR2(20);

BEGIN

     -- At time t1:
     SELECT      c_lob
     INTO        clob_updated
     FROM        lob_table
     WHERE       key_value = 1
     FOR UPDATE;

     read_amount := 10;
     read_offset := 1;
     dbms_lob.read(clob_updated, read_amount, read_offset,
          buffer);
     dbms_output.put_line('clob_updated value: ' || buffer);
     -- This produces the output 'abcd'

     -- At time t2:
     write_amount := 3;
     write_offset := 5;
```

```
     buffer := 'efg';
     dbms_lob.write(clob_updated, write_amount, write_offset,
          buffer);

     read_amount := 10;
     dbms_lob.read(clob_updated, read_amount, read_offset,
         buffer);
     dbms_output.put_line('clob_updated value: ' || buffer);
     -- This produces the output 'abcdefg'


    -- At time t3:
    COMMIT;


    -- At time t4:
    read_amount := 10;
    dbms_lob.read(clob_updated, read_amount, read_offset,
         buffer);
    -- ERROR: ORA-22990: LOB locators cannot span transactions
END;
/
```

トランザクションにまたがらないロケータの例

```
Assume the following tables:

CREATE TABLE tdrslob01 (
     a    CLOB,
     b    BLOB,
     c    NUMBER);

CREATE TABLE foo (
     key  NUMBER);

CONNECT to the database
EXECUTE "SELECT C,'I',A FROM tdrslob01 ORDER BY 1"
[ARRAY FETCH GET 16 rows INTO OCILobLocator ARRAY]
for (i=0; i<16; i++)
{
  EXECUTE "INSERT INTO foo VALUES(5)"
  OCITransCommit(...);
  FETCH TEXT in CLOB USING locators fetched
}
```

ロケータの SELECT はトランザクション外で発生するため、これらの文は問題なく実行されます。つまり、選択したロケータは、トランザクションに対応付けられません。たとえ 'for'

ループにある INSERT で暗黙的にトランザクションが開始されたとしても、'for' ループにあるそれに続く COMMIT によって、トランザクションは終了します。トランザクション外の SELECT から戻されたロケータを介した LOB データの FETCH は、正常に処理されます。ロケータの SELECT とロケータ・データの FETCH はトランザクション外で発生します。

ただし、1 つの文を追加すると、エラーが発生します。

```
CONNECT to the database
EXECUTE "INSERT INTO foo VALUES(5)"  <===
EXECUTE "SELECT C,'I',A FROM tdrslob01 ORDER BY 1"
[GET 16 rows]
for (i=0; i<16; i++)
{
  EXECUTE "INSERT INTO foo VALUES(5)"
  OCITransCommit(...);
  FETCH text in CLOB USING locators fetched <== get ORA-22990
}
```

第 2 の例では、ロケータの SELECT は、トランザクション内で発生します（INSERT 文は、暗黙的にトランザクションを開始します）。つまり、選択したロケータは、トランザクションに対応付けられます。'for' ループの COMMIT は、ロケータが選択されたトランザクションをコミットします。そのため、次の FETCH では、すでにコミット済みの以前のトランザクションのロケータ値をフェッチしようとします。その結果、22990 エラーが返されます。

最初に INSERT を実行したすぐ後に、COMMIT を実行すると正常に処理されます。

```
CONNECT to the database
EXECUTE "INSERT INTO foo VALUES(5)"  <===
OCITransCommit(...);  <===
EXECUTE "SELECT C,'I',A FROM tdrslob01 ORDER BY 1"
[get 16 rows]
for (i=0; i<16; i++)
{
  EXECUTE "INSERT INTO foo VALUES(5)"
  OCITransCommit(...);
  FETCH text in CLOB using locators fetched
}
```

この例では、INSERT によりトランザクションが暗黙的に開始され、COMMIT によりトランザクションが終了します。そのため、ロケータの SELECT はトランザクション外で発生します。つまり、選択したロケータは、トランザクションに対応付けられません。繰り返しますが、'for' ループにある INSERT が暗黙的にトランザクションを開始したとしても、'for' ループにあるそれに続く COMMIT によって、トランザクションは終了します。そのため、トランザクション外で発生した SELECT から戻されたロケータを介した LOB データの FETCH は、正常に処理されます。ロケータの SELECT とロケータ・データの FETCH はトランザクション外で発生します。

大量の LOB データを効率よく読み書きする。

大量の LOB データを最も効率よく読み書きする方法は、ポーリングまたはコールバックを介してストリーム・メカニズムを可能にして OCILobRead() または OCILobWrite() を使用することです。

> **関連項目：** これらの API とその使用方法のサンプルについては、『Oracle 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 値の読込み

LOB 値を読み込むとき、LOB の最後を超えて読み込んでもエラーにはなりません。開始オフセットと LOB のデータ量にかかわらず、通常 4GB の入力を指定できます。読み込む量を決定するために、サーバーへの OCILobGetLength() コールを繰り返し、LOB 値の長さを判断する必要はありません。

たとえば、LOB の長さが 5,000 バイトで、オフセット 1,000 から開始して LOB 値全体を読み込むとします。また、LOB 値の現在の長さがわからないとします。この場合の OCI 読込みコールを示します（初期化パラメータはすべて省略してあります）。

```
#define MAX_LOB_SIZE 4294967295
ub4  amount =  MAX_LOB_SIZE;
ub4  offset = 1000;
OCILobRead(svchp, errhp, locp, &amount, offset, bufp, bufl, 0, 0, 0, 0)
```

LOB 値の書込み

前述したとおり、LOB にデータを移入するまたは大量のデータを LOB に書き込む最良の方法は、ストリーム方式で OCILobWrite() コールを使用することです。LOB に書き込むデータの量がわかっている場合は、OCILobWrite() のコール時にそのデータ量を指定します。こうすると、ディスクに LOB データが連続的に書き込まれます。領域を効率的に使用できるだけでなく、LOB データの連続的な構造により、その後の操作で読書きの速度が速くなります。

LOB をコピーする

内部 LOB をコピーする

BFILE で使われるリファレンス・セマンティクスに対し、内部 LOB タイプの BLOB および CLOB、NCLOB では、コピー・セマンティクスが使用されます。BLOB または CLOB、NCLOB が、ある行から同じ表または別の表の行にコピーされる場合、LOB ロケータだけがコピーされるのではなく、実際の LOB 値がコピーされます。たとえば、lob_table1 と lob_table2 に、上述の lob_table と同一のスキーマがあると仮定して、次のような文があるとします。

```
INSERT INTO lob_table1 (key_value, b_lob)
    (SELECT key_value, b_lob FROM lob_table2 T2
     WHERE T2.key_value = 101);
```

この文によって、表 lob_table1 内に新しい LOB ロケータが作成され、表 lob_table1 に挿入された新しいロケータによって参照される位置に lob_table2 からの LOB データがコピーされます。

外部 LOB をコピーする

BFILE タイプでは、コピー・セマンティクスではなく、リファレンス・セマンティクスが使用されます。つまり、BFILE のロケータだけが、ある行から別の行にコピーされます。言い換えれば、オペレーティング・システムのコマンドを発行して、そのファイルをコピーしなければ、外部 LOB の値のコピーを作成することはできません。

LOB を削除する

内部 LOB を削除する

内部 LOB の列 / 属性を含む行を削除するには、(a) 明示的な SQL DML コマンド DELETE を使用する、(b) DROP TABLE または TRUNCATE TABLE、DROP TABLESPACE などの、行を事実上削除する SQL DDL コマンドを使用するという、2 つの方法があります。いずれの場合も、LOB ロケータだけでなく *LOB 値*も削除することになります。

しかし、読取り一貫性の機構により、LOB ロケータを戻す文（SELECT など）の実行時の古い LOB 値にはアクセス可能であることに注意してください。

関連項目： 6-22 ページの「読取り一貫性のあるロケータ」

もちろん、LOB 列を持つ表の 2 つの個別の行には、LOB 値が同じであるかないかにかかわらず、それぞれ 個別の LOB ロケータと個別の LOB 値のコピーがあります。つまり、行の削除は、LOB がもともと別の行からコピーされたものであっても、他の行のデータまたは LOB ロケータに影響を与えません。

外部 LOB を削除する

ただし、BFILE 内の LOB 値は、SQL DDL コマンドまたは SQL DML コマンドでは削除できません。これらのコマンドを使用すると、BFILE ロケータだけが削除されます。BFILE 列を含むレコードを削除すると、物理オペレーティング・システム・ファイルそのものが削除されるのではなく、そのレコードが既存のファイルからリンク解除されます。ある行に対して SQL DELETE 文を発行すると、その行の BFILE ロケータが削除されるため、オペレーティング・システム・ファイルへの参照を削除することになります。

次に示す DELETE または DROP TABLE、TRUNCATE TABLE 文は、行、つまり image1.gif を参照する BFILE ロケータを削除しますが、ファイルシステムにあるオペレーティング・システム・ファイルは残ります。

```
DELETE FROM lob_table
     WHERE key_value = 21;
```

```
DROP TABLE lob_table;

TRUNCATE TABLE lob_table;
```

LONG から LOB にデータをコピーする

ユーザーが直面する問題の 1 つに、LONG データ型から LOB 形式へのデータの変換方法があります。PL/SQL プログラム loadlob.sql では、DBMS_LOB.LOADFROMFILE メソッドを使用して LONG から LOB に変換する方法の例を示します。このプログラムでは、次のステップを実行する必要があります。

1. 古い表の LONG 列をそのまま残し、ALTER TABLE コマンドを使用して新しい LOB 列を追加します。

   注意： ALTER TABLE コマンドでは、LONG 列のタイプを LOB 列に変更できません。LONG 列と LOB 列は異なるデータ型のため、LONG 列を LOB 列に割り当てることはできません。

2. LONG または LONG RAW のデータをフラット・ファイルに書き込みます。
3. CREATE DIRECTORY を使用して、BFILE ファイル（フラット・ファイル）が書き込まれたディレクトリを参照します。
4. OCI か PL/SQL のどちらかを使用して、データをサーバー側のフラット・ファイルから LOB にコピーする 3 つの方法があります。

   a. OCI コマンド OCILobLoadFromFile または PL/SQL コマンド DBMS_LOB.LOADFROMFILE(): サーバー側のオペレーティング・システムのフラット・ファイルから LOB にコピーする 1 番速い方法です。

   b. サーバー側の外部プロシージャの OCI コマンド OCILobWrite(): サーバー側の外部プロシージャをコールするプログラムがクライアントから実行されても、フラット・ファイルは、サーバー側に残ります。これは、サーバー側のオペレーティング・システムのフラット・ファイルから LOB に転送する 2 番目に速い方法です。

   c. OCI コマンド OCILobWrite(): この方法は、後述の bull_lob プログラムで使用されています。これは、LONG を LOB に変換する最も速い方法ではありませんが、この状況では唯一可能な方法です。プログラムがリモート・クライアント・マシンで実行されている場合、サーバーの LONG データを、クライアント・マシンのフラット・ファイルに書き込み、その書き込まれたファイルを、サーバーの LOB 列に戻さなければなりません。これら 2 つの転送がネットワークを介するので、頻繁に繰り返される操作の場合、パフォーマンスの負荷を考慮しなければなりません。

注意： フラット・ファイルまたは BFILE では、データはバイナリまたはロー・データで保存されるため、ユーザーは、CLOBS および NCLOBS に対してそれぞれキャラクタ・セットの変換を行う必要があります。

警告： **エクスポート・ユーティリティおよびインポート・ユーティリティでは、現在、LONG から LOB に変換できません。**

例

次の例に、/tmp/sound_clip というフラット・ファイルに書き込まれた LONG を LOB 列にロードする上記の 4(a) の方法の PL/SQL バージョンを示します。

注意： OCILobWrite コマンドを使用して LONG を LOB に変換する方法については、別の説明があります。

次のステップで、PL/SQL スクリプト loadlob.sql を実行します。

1. 次の内容のファイル sound_clip を作成して、/tmp ディレクトリにコピーします。

```
sound_clip: abcdefghijklmnopqrstuvwxyz
```

2. 次の SQL スクリプトを実行します。

```
% sqlplus scott/tiger @loadlob
loadlob.sql
-----------
set echo on;
connect sys/change_on_install;
grant all on dbms_lob to scott;
grant create any directory to scott;
connect scott/tiger;
drop directory some_dir_alias;
create directory some_dir_alias as '/tmp';
drop table multimedia;

/* 表を作成する */

CREATE TABLE multimedia
(
```

```
   id           NUMBER,
   video_clip   CLOB DEFAULT empty_clob(),
   audio_clip   CLOB DEFAULT NULL,
   some_file    BFILE DEFAULT NULL
) ;

/* 表にデータをロードする。*/
/* 表に 10 行挿入する。これが初期値になる。*/
/* video_clip を空に、audio_clip および some_file を NULL にする。*/

/* 最も速いのは、OCI で配列の挿入を使用する方法。*/
/* (OCIBindArrayOfStruct を参照 ) */
/* 高速ではないが、次のように PL/SQL でループを使用する方法。*/

declare
   loop_count integer;
begin
   loop_count := 1;
   while loop_count <= 10 loop
     insert into multimedia (id) values (loop_count);
     loop_count := loop_count + 1;
   end loop;
end;
/

/* 最初のオーディオ・クリップを実際の値に初期化する。*/
/* 次にこの値を表のすべての列にコピーする。*/

declare
  ac      clob;
  amount  integer;
  a_file  bfile := BFILENAME('SOME_DIR_ALIAS', 'sound_clip');
begin
  update multimedia set audio_clip = empty_clob() where id = 1 returning
        audio_clip into ac;

/* オーディオ・クリップを含むサーバー側のファイルをオープンして、*/
/* それを CLOB にロードしてから、そのファイルをクローズする。オーディオ・クリップのサイズは
*/
/* 32,000 バイトしかなく、ファイル内の 1 の位置から始まる。*/

  dbms_lob.fileopen(a_file, dbms_lob.file_readonly);
  amount := 26;

/* 宛先およびソースのオフセットのデフォルトは、1 であることに注意してください。*/

  dbms_lob.loadfromfile(ac, a_file, amount);
```

```
  dbms_lob.fileclose(a_file);
  commit;

/* 表のすべての列を、ロードしたオーディオ・クリップに更新する。*/
  update multimedia set audio_clip =
        (select audio_clip from multimedia where id = 1)
        where audio_clip is null;
end;
/

select id, audio_clip from multimedia;
```

3. 次のように出力されます。

```
SQL> @loadlob
SQL> set echo on;
SQL> connect sys/change_on_install;
Connected.
SQL> GRANT ALL on dbms_lob to scott;
Grant succeeded.
SQL> GRANT CREATE ANY DIRECTORY to scott;
Grant succeeded.
SQL> CONNECT scott/tiger;
Connected.
SQL> DROP DIRECTORY some_dir_alias;
Directory dropped.
SQL> CREATE DIRECTORY some_dir_alias as '/tmp';
Directory created.
SQL> DROP TABLE multimedia;
Table dropped.
SQL>
SQL> /* 表を作成する */
SQL>
SQL> create table multimedia
  2  (
  3  id     number,
  4  video_clip   clob default empty_clob(),
  5  audio_clip   clob default null,
  6  some_file    bfile default null
  7  ) ;

Table created.

SQL>
SQL>
SQL> /* 表にデータをロードする */
SQL> /* 表に 10 行挿入する。これが初期値になる。*/
DOC> /* video_clip を空に、audio_clip および some_file を NULL にする。*/
```

```
DOC> */
SQL>
SQL> /* 最も速いのは、OCI で配列の挿入を使用する方法。*/
DOC> /* (OCIBindArrayOfStruct を参照 ) */
DOC> /* 高速ではないが、次のように plsql でループを使用する方法。*/

SQL>
SQL> declare
  2  loop_count integer;
  3  begin
  4  loop_count := 1;
  5  while loop_count <= 10 loop
  6    insert into multimedia (id) values (loop_count);
  7    loop_count := loop_count + 1;
  8  end loop;
  9  end;
 10  /

PL/SQL procedure successfully completed.

SQL> /* 最初のオーディオ・クリップを実際の値に初期化する。*/
DOC> /* 次にこの値を表のすべての列にコピーする。*/

SQL> DECLARE
  2    ac      CLOB;
  3    amount  INTEGER;
  4    a_file  BFILE := BFILENAME('SOME_DIR_ALIAS', 'sound_clip');
  5 BEGIN
  6    UPDATE multimedia SET audio_clip = empty_clob() WHERE id = 1 returning
  7       audio_clip into ac;
  8  /* オーディオ・クリップを含むサーバー側のファイルをオープンして、*/
  9  /* それを CLOB にロードしてから、そのファイルをクローズする。オーディオ・クリップのサ
イズは */
 10  /* 32,000 バイトしかなく、*/
 11  /* ファイル内の 1 の位置から始まる。*/
 12    dbms_lob.fileopen(a_file, dbms_lob.file_readonly);
 13    amount := 26;
 14    /* 宛先およびソース・オフセットのデフォルトは、1 であることに注意してください。*/
 15    dbms_lob.loadfromfile(ac, a_file, amount);
 16    dbms_lob.fileclose(a_file);
 17    COMMIT;
 18
 18  /* 表のすべての列を、ロードしたオーディオ・クリップに更新する。*/
 19    UPDATE multimedia SET audio_clip =
 20       (SELECT audio_clip FROM multimedia WHERE id = 1)
 21       WHERE audio_clip is null;
 22  end;
```

```
 23  /

PL/SQL procedure successfully completed.

SQL>
SQL> select id, audio_clip from multimedia;

        ID
-------
AUDIO_CLIP
----------------------------------------------------------
           1
abcdefghijklmnopqrstuvwxyz
         2
abcdefghijklmnopqrstuvwxyz
         3
abcdefghijklmnopqrstuvwxyz

        ID
-------
AUDIO_CLIP
----------------------------------------------------------
         4
abcdefghijklmnopqrstuvwxyz
         5
abcdefghijklmnopqrstuvwxyz
         6
abcdefghijklmnopqrstuvwxyz

        ID
-------
AUDIO_CLIP
----------------------------------------------------------
         7
abcdefghijklmnopqrstuvwxyz
         8
abcdefghijklmnopqrstuvwxyz
         9
abcdefghijklmnopqrstuvwxyz

        ID
-------
AUDIO_CLIP
----------------------------------------------------------
        10
abcdefghijklmnopqrstuvwxyz
```

```
10 rows selected.

SQL>
SQL> quit
```

オブジェクト・キャッシュの LOB

内部 LOB 属性を持つオブジェクト・キャッシュ内にオブジェクトを作成するとき、その LOB 属性は暗黙的に空に 設定されます。データを LOB に書き込むのに、この空の LOB ロケータを使用してはいけません。最初にオブジェクトをフラッシュし、それによって、表に行を挿入し、空の LOB、つまり長さ 0（ゼロ）の LOB を作成しなければなりません。オブジェクトがオブジェクト・キャッシュ内にリフレッシュされ (OCI_PIN_LATEST を使用します)、実際の LOB ロケータが属性に読み込まれると、OCI LOB API をコールして LOB にデータを書き込むことができます。

BFILE 属性を持つオブジェクトを作成する場合、BFILE は NULL に設定されます。ファイルからの読込みを始める前に、有効なディレクトリ別名とファイル名を使用して BFILE を更新しておかなければなりません。

LOB ロケータ属性を持つオブジェクト・キャッシュ内で、あるオブジェクトを別のオブジェクトにコピーする場合、LOB ロケータだけがコピーされます。つまり、これら 2 つの異なるオブジェクトの LOB 属性には、1 つの同じ LOB 値を参照する、同一のロケータが含まれることを意味します。ターゲット・オブジェクトがフラッシュされたときに限り、LOB 値の個別の、物理的なコピーが作成されます。これは、ソースの LOB 値とは別の値となります。

> **関連項目：** ロケータの 1 つを使用して書込みが実行される場合、各オブジェクトでどのバージョンの LOB 値を参照できるかの詳細は、6-23 ページの「読取り一貫性のあるロケータの例」を参照してください。

したがって、コピーのターゲットになった LOB を変更する場合には、ターゲット・オブジェクトをフラッシュしてリフレッシュし、ロケータ属性を介して LOB に書き込まなければなりません。

LOB バッファリング・サブシステム

LOB バッファリング

Oracle8 では、LOB バッファリング・サブシステム（LBS）が提供されており、DataCartridge、Web サーバー、その他のクライアント・ベースのアプリケーションなど、1 つ以上の LOB のコンテンツをクライアントのアドレス領域にバッファリングする必要のある、拡張 OCI ベースのアプリケーションに対応しています。バッファリング・サブシステムに必要なクライアント側のメモリーは、最大使用時で 512 キロバイトです。これは、バッファリング・アクセス用に使用可能な LOB での 1 回の読込みまたは書込み操作に対して指定できる最大サイズでもあります。

LOB バッファリングの利点

バッファリングは、LOB の特定の部分に一連の小規模の読込みと書込みを実行する（繰返し行われる場合が多い）アプリケーションについて、特に次の 2 点で有効です。

- バッファリングによってサーバーへの遅延書込みが可能になる。LOB バッファへの書込みを数回までクライアントのアドレス領域にバッファリングし、最後にサーバーへとフラッシュできます。これにより、クライアント・アプリケーションとサーバー間のネットワークの往復回数が減り、LOB 更新の総合的なパフォーマンスが向上します。
- バッファリングにより、サーバーでの LOB の総更新回数が減り、LOB のバージョンの数とロギングの量が減る。これにより、総合的な LOB パフォーマンスが向上し、ディスク領域が効率よく使用できます。

LOB バッファリングの使用についての考慮事項

バッファリングされた LOB 操作については次の点に注意しなければなりません。

- Oracle8 で提供されるのは、単純バッファ方式で、キャッシュではない。具体的には、Oracle8 では、LOB のバッファの内容がサーバーの LOB の値に同期することは保証されていません。LOB のバッファの内容を明示的にフラッシュしなければ、サーバーの実際の LOB を反映した、バッファリングされた書込みの結果を参照できません。
- バッファリングされた LOB 操作のエラー回復はユーザーの責任で行う。実際の LOB の更新には遅れがあるため、バッファリングされた特定の読込みまたは書込み操作のエラー報告はサーバー・ベースの LOB に次回アクセスするまで行われません。
- バッファリングされた LOB 操作が行われるトランザクションは、他のユーザー・セッションに移行できない。LBS はシングル・ユーザーで、単一スレッド・システムだからです。
- Oracle8 では、バッファリングされた LOB 操作に対するトランザクションのサポートは保証されていない。バッファリングされた LOB 更新に対してトランザクション・セマンティクスを保証するには、アプリケーションで論理セーブポイントを保持して、エラーの際には、LOB に対するすべての変更がロールバックされるようにしなければなりません。論理セーブポイント内のバッファリングされた LOB の更新を常にラップする必要があります。
- どのトランザクションでも、バッファリングされた書込みを使用して LOB の更新を開始した後は、ユーザーの責任で、同じトランザクション内で、バッファリング・サブシステムを使用しない別の操作によって、同じ LOB が更新されないことを確認する。

  これは、サーバー・ベースの LOB を更新する SQL 文を使用することによって可能な場合もあります。Oracle8 ではこのような操作を区別できないので、操作を防ぐことができません。これは、アプリケーションの正確さと整合性に影響します。
- LOB のバッファリングされた操作は、通常の場合と同様にロケータを介して実行される。バッファリング用に使用可能にされたロケータは、そのロケータを使用した LOB への書込み操作が実行されるまでは、LOB の読取り一貫性のあるバージョンを提供します。

関連項目： 6-22 ページの「読取り一貫性のあるロケータ」

ロケータがバッファリングされた書込みに使用されていることによって、更新済みロケータになった後は、バッファリング・サブシステムによってわかるように、常に LOB の最新バージョンへのアクセスが提供されます。この更新済みのロケータについて、バッファリングのもう 1 つの重要な点は、これ以降の LOB へのバッファリングされた書込みがこの更新済みのロケータからしか実行できないことです。Oracle8 では、バッファリング用に使用可能にされたその他のロケータを介して LOB への書込みを行おうとすると、エラーが戻されます。

関連項目： 6-25 ページの「更新済みロケータ」

- バッファリング用に使用可能にされた更新済みのロケータは、PL/SQL プロシージャへの IN パラメータとして渡せる。ただし、IN OUT または OUT パラメータを渡すと、更新済みロケータを戻そうとした場合と同様に、エラーが発生します。
- バッファリング用に使用可能にされた更新済みロケータを別のロケータに割り当てることはできない。ロケータの割当ては、OCILobAssign() の使用、PL/SQL 変数の割当て、オブジェクトに LOB 属性が含まれているオブジェクトに対する OCIObjectCopy() の使用などによって発生します。バッファリング用に使用可能にされた読取り一貫性のあるロケータを、使用可能なバッファリングを持たないロケータに割り当てると、対象となるロケータに対してバッファリング機能がオンになります。同様に、バッファリング用に使用可能にされなかったロケータを、使用可能なバッファリングを持たないロケータに割り当てると、対象となるロケータに対してバッファリング機能がオフになります。

  また、もともとバッファリングが使用可能にされていたロケータに対して SELECT INTO を発行した場合、ロケータは新規のロケータ値で上書きされ、バッファリング機能はオフになります。
- バッファリングされた書込みを使用した LOB 値への追加は、これらの書込みを開始するオフセットが、BLOB（または CLOB/NCLOB）の最後から 1 バイト（または 1 文字）の場合にだけ可能。言い換えると、バッファリング・サブシステムでは 0 バイトのフィラーまたはスペースをサーバー・ベースの LOB 内に追加作成することはサポートされていません。
- CLOB の場合、Oracle8 では、クライアント側のロケータ・バインド変数のキャラクタ・セット・フォームがサーバーの LOB と同じものでなければならない。たいていの OCI LOB プログラムでは、この条件が必要です。ロケータがリモート・データベースから SELECT されると、OCI プログラムによって現在アクセスしているデータベースからのキャラクタ・セット・フォームと異なっている場合がありますが、これは例外です。この場合、エラーが戻されます。ユーザーによってキャラクタ・セット・フォームが入力されていない場合、これは SQLCS_IMPLICIT と見なされます。

LOB バッファリング操作

LOB バッファの物理構造 Oracle 8.0 の場合、各ユーザー・セッションには、16 ページの固定ページ・プールがあり、そのセッションからバッファリング・モードでアクセスされるす

べての LOB で共有されます。各ページは、32 キロバイト（文字ではない）までの固定サイズです。LOB のバッファは、1 セッションあたり 1 ページ以上、最大 16 ページまでのこれらのページからなっています。バッファリングされた読込みまたは書込み操作に指定できるサイズの最大値は 512 キロバイトで、ぞれぞれの環境によって、読込みまたは書込みの最大値が小さくなる可能性があることを覚えておいてください。

LOB は、固定サイズの論理領域に分割できることを考慮してください。各ページはこれらの固定サイズの領域の 1 つにマップされ、基本的には、これらの領域のメモリー内コピーにもマップされます。入力オフセットと、読込みまたは書込み操作用に指定したサイズに従って、Oracle8 では、ページ・プールから 1 ページ以上の空きページが LOB のバッファに割り当てられます。空きページとは、バッファリングされた読込みまたは書込み操作によって、読込みまたは書込みのされていないページです。

LOB バッファリング・システムを使用する たとえば、入力オフセットが 1000 で、指定された読込み / 書込みのサイズが 30000 でページ・サイズが 32K の場合、Oracle8 では、LOB の最初の 32 キロバイトの領域が LOB のバッファのページに読み込まれます。入力オフセットが 33000 で、読込み / 書込みのサイズが 30000 の場合、LOB の次の 32 キロバイトの領域がページに読み込まれます。入力オフセットが 1000 で、読込み / 書込みのサイズが 35000 の場合、LOB のバッファは 2 ページになります。最初の部分は、LOB の 1 から 32 キロバイト目までの領域に、次の部分は 32 キロバイト +1 バイト目から 64 キロバイト目までの領域にマップされます。

ページと LOB 領域の間のこのマッピングは、Oracle8 によって別の領域がページにマップされるまでの一時的なものです。LOB のバッファ内で満杯になっていてもう使用できない LOB の領域にアクセスしようとすると、Oracle8 では、ページ・プールから任意の使用可能な空きページが LOB のバッファに割り当てられます。ページ・プールに使用可能なページがない場合は、Oracle8 では、次のようにページが割り当て直されます。LOB のバッファで変更されていないページのうち、使用頻度の最も低いページがエージングによって解放され、現在の操作に割り当て直されます。

LOB のバッファで使用できるそのようなページがない場合は、同じセッションにある、別のバッファリングされた LOB の変更されていないページのうち、使用頻度の最も低いページがエージングによって開放されます。それでも使用できるページがない場合は、ページ・プールのすべてのページがダーティである（つまり、変更されている）ことが暗黙指定され、その場合、現在アクセス中の LOB または別の LOB をフラッシュする必要があります。Oracle8 では、この状態がエラーとしてユーザーに通知されます。Oracle8 では、ダーティ・ページが暗黙的にフラッシュされたり、割り当て直されることはありません。これらは、ユーザーが明示的にフラッシュしたり、LOB のバッファリング機能を使用禁止にして破棄できます。

上記の説明をわかりやすくするために、バッファリング・モードで L1 および L2 の 2 つの LOB にアクセス中で、各 LOB には 8 ページのバッファがある場合を考えます。L1 のバッファの 8 ページのうち 6 ページがダーティで、残りの 2 ページには、サーバーから読み込まれた変更されていないデータが入っているとします。L2 のバッファの状態も同様であるとします。ここで、L1 の次のバッファリングされた操作のために、Oracle8 によって、L1 のバッファの変更されていない 2 ページから使用頻度の最も低いページが割り当て直されます。L1 のバッファの 8 ページすべてが LOB 書込みのために使用されると、Oracle8 最低使用頻度の

方針を使って、L2 のバッファから変更されていない 2 ページを割り当てることによって、L1 にさらに 2 つの操作を提供できます。L1 または L2 でさらに、バッファリングされた操作を実行しようとすると、Oracle8 ではエラーが戻されます。

バッファがすべてダーティで、バッファリングされた LOB に別の読込みと書込みを行うと次のエラーが発生します。

```
エラー 22280: その操作に使用可能なバッファは、これ以上ありません。
```

これには、2 つの原因が考えられます。

1. バッファ・プールにあるすべてのバッファが、以前の操作に使用されていた。

   この場合、LOB の更新に使用されているロケータを介して LOB をフラッシュします。

2. 以前バッファリングされた更新操作がないのに LOB をフラッシュしようとしている。

   この場合、バッファをフラッシュする前に、バッファリングが可能なロケータを介して LOB に書き込みます。

LOB バッファをフラッシュする フラッシュとは、一連のプロセスを指します。ロケータを介してバッファの LOB にデータを書き込むと、そのロケータは、更新済みロケータになります。いったん、更新済みロケータを介してバッファの LOB データを更新すると、フラッシュ・コールは、次のことを行います。

- LOB のバッファのダーティ・ページをサーバー・ベースの LOB に書き込み、それによって LOB 値を更新する。
- 更新済みロケータをリセットして、読取り一貫性のあるロケータにする。
- フラッシュ済みバッファを解放するか、バッファのページの状態をダーティから変更されていない状態に戻す。

フラッシュの後、ロケータは、読取り一貫性のあるロケータとなり、別のロケータに割り当てることができます (L2 := L1)。

たとえば、L1 と L2 の 2 つのロケータがあるとします。両方とも読取り一貫性のあるロケータであり、サーバーの LOB データの状態と一貫性があるとします。バッファに書き込みをして LOB を更新すると、L1 は更新済みロケータになります。L1 と L2 は、現在 LOB 値の別のバージョンを示しています。サーバーの LOB を更新する場合、L1 を使用して L2 で得た読取り一貫性のある状態を維持しなければなりません。フラッシュ操作で、新しいスナップショット環境が、フラッシュに使用されたロケータに書き込まれます。注意しなければならない重要な点は、LOB バッファをフラッシュするとき、更新済みロケータ（L1）を使用しなければならないことです。読取り一貫性のあるロケータをフラッシュしようとすると、エラーが発生します。

ここで、問題が起こります。LOB バッファのデータに何が起こるのか？2 つの可能性があります。デフォルト・モードでは、フラッシュ操作では、変更したページのデータが維持され

ます。この場合、同じバイト範囲に対する読込みまたは書込みのために、サーバーとのやりとりはせずに済みます。ここでのフラッシュでは、バッファのデータはクリアされないことに注意してください。また、フラッシュ済みバッファに占有されているメモリーは、クライアントのアドレス領域に戻されません。

注意： 変更されていないページは、必要ならばエージング処理を行うことができます。

第 2 の場合、OCILobFlushBuffer() のフラグ・パラメータを OCI_LOB_BUFFER_FREE に設定して、バッファ・ページを解放すると、メモリーは、クライアントのアドレス領域に戻されます。ここでのフラッシュとは、サーバーの LOB 値を更新し、読取り一貫性のあるロケータを戻し、バッファ・ページを解放することです。

更新済み LOB をフラッシュする LBS を使って更新した LOB は、次の時点で必ずフラッシュしなければならないことに注意してください。

- トランザクションをコミットする前
- 現行のトランザクションから別のトランザクションに移行する前
- LOB のバッファリング操作を使用禁止にする前
- 外部コールアウトの実行から PL/SQL の関数 / プロシージャ / メソッドのコールに戻る前

注意： 外部コールアウトが、PL/SQL ブロックからコールされ、ロケータがパラメータとして渡される場合、コールを使用可能にすることを含めてすべてのバッファリング操作がコールアウトの内部で行われなければなりません。言い換えれば、次の手順に従うことをお薦めします。

- 外部コールアウトをコールする
- バッファリング用ロケータを使用可能にする
- ロケータを使用して読込む / 書込む
- LOB をフラッシュする
- バッファリング用のロケータを使用禁止にする。
- PL/SQL での関数 / プロシージャ / メソッドのコールに戻る。

Oracle8 では、LOB は暗黙的にはフラッシュされないことに注意してください。

バッファリング用のロケータを使用可能にする バッファ対応のロケータを使用できる場合と使用できない場合があることに注意してください。

- バッファリング用に使用可能にされたロケータは、次の OCI API がある場合にだけ使用できます。

  OCILobRead(), OCILobWrite(), OCILobAssign(), OCILobIsEqual(), OCILobLocatorIsInit(), OCILobLocatorSize(), OCILobCharSetId(), OCILobCharSetForm().

- 次の OCI API は、バッファリング用に使用可能にされたロケータと一緒に使用すると、エラーが戻されます。

  `OCILobCopy(), OCILobAppend(), OCILobErase(), OCILobGetLength(), OCILobTrim().`

  これらの API は、バッファリング用に使用可能になっていないロケータとともに使用して、ロケータが示す LOB がその他のロケータを介してバッファリング・モードですでにアクセスされている場合にも、エラーが戻されます。

- 入力 lob ロケータでバッファリングを使用可能にすると、DBMS_LOB API からエラーが戻されます。

- その他のすべてのロケータの場合と同様に、LOB バッファリング用に使用可能にされたロケータはトランザクション全体に反映できません。

再選択されないように、ロケータの状態を保存する さらに LOB バッファに書込む前に、LOB の現在の状態を保存するとします。LOB バッファリングの使用中に更新を実行すると、既存のバッファへの書込みで、サーバーへのアクセスはせずに済むため、ロケータのスナップショット環境はリフレッシュされません。これは、LOB バッファリングを使用しないで直接 LOB を更新する場合にはあてはまりません。その場合、更新するたびにサーバーへの往復が必要となるため、ロケータのスナップショットはリフレッシュされます。LOB バッファを使用して書き込まれた LOB の状態を保存するには、次のことを実行する必要があります。

1. `LOB` をフラッシュして、`LOB` とロケータ `(L1)` のスナップショット環境を更新する。この時点では、ロケータ（`L1`）の状態と `LOB` は同じです。
2. フラッシュと更新に使用されたロケータ (L1) を別のロケータ (L2) に割り当てる。この時点では、2 つのロケータの状態 (L1 および L2) と LOB はすべて同じです。

これで、L2 は読取り一貫性のあるロケータになり、フラッッシュが行われるまでは L1 を介して行われた変更にアクセスできますが、フラッシュ後はアクセスできません。この割当てにより、ロケータを L2 に選択し直すためにサーバーへはアクセスせずに済みます。

LOB バッファリングの例

lob_table スキーマに基づく、次の OCI プログラム用の疑似コードは、上で挙げた概念を説明しています。

```
OCI_BLOB_buffering_program()
{
   int            amount;
   int            offset;
   OCILobLocator  lbs_loc1, lbs_loc2, lbs_loc3;
   void          *buffer;
   int            bufl;

   -- Standard OCI initialization operations - logging on to
   -- server, creating and initializing bind variables etc.

   init_OCI();

   -- Establish a savepoint before start of LBS operations
   exec_statement("savepoint lbs_savepoint");

   -- Initialize bind variable to BLOB columns from buffered
   -- access:
   exec_statement("select b_lob into lbs_loc1 from lob_table
       where key_value = 12");
   exec_statement("select b_lob into lbs_loc2 from lob_table
       where key_value = 12 for update");
   exec_statement("select b_lob into lbs_loc2 from lob_table
       where key_value = 12 for update");

   -- Enable locators for buffered mode access to LOB:
   OCILobEnableBuffering(lbs_loc1);
   OCILobEnableBuffering(lbs_loc2);
   OCILobEnableBuffering(lbs_loc3);

   -- Read 4K bytes through lbs_loc1 starting from offset 1:
   amount = 4096; offset = 1; bufl = 4096;
   OCILobFileRead(.., lbs_loc1, offset, &amount, buffer, bufl,
      ..);
      if (exception)
          goto exception_handler;
          -- This will read the first 32K bytes of the LOB from
          -- the server into a page (call it page_A) in the LOB's
          -- client-side buffer.
          -- lbs_loc1 is a read consistent locator.

          -- Write 4K of the LOB throgh lbs_loc2 starting from
          -- offset 1:
```

```
    amount = 4096; offset = 1; bufl = 4096;
    buffer = populate_buffer(4096);
    OCILobFileWrite(.., lbs_loc2, offset, amount, buffer,
        bufl, ..);

if (exception)
    goto exception_handler;
    -- This will read the first 32K bytes of the LOB from
    -- the server into a new page (call it page_B) in the
    -- LOB's buffer, and modify the contents of this page
    -- with input buffer contents.
    -- lbs_loc2 is an updated locator.

    -- Read 20K bytes through lbs_loc1 starting from
    -- offset 10K
    amount = 20480; offset = 10240;
    OCILobFileRead(.., lbs_loc1, offset, &amount, buffer,
        bufl, ..);

if (exception)
  goto exception_handler;
    -- Read directly from page_A into the user buffer.
    -- There is no round-trip to the server because the
    -- data is already in the client-side buffer.

    -- Write 20K bytes through lbs_loc2 starting from offset
    -- 10K
    amount = 20480; offset = 10240; bufl = 20480;
    buffer = populate_buffer(20480);
    OCILobFileWrite(.., lbs_loc2, offset, amount, buffer,
         bufl, ..);

if (exception)
    goto exception_handler;
    -- The contents of the user buffer will now be written
    -- into page_B without involving a round-trip to the
    -- server.  This avoids making a new LOB version on the
    -- server and writing redo to the log.

    -- The following write through lbs_loc3 will also
    -- result in an error:
    amount = 20000; offset = 1000; bufl = 20000;
    buffer = populate_buffer(20000);
    OCILobFileWrite(.., lbs_loc3, offset, amount, buffer,
         bufl, ..);

if (exception)
```

```
            goto exception_handler;
            -- No two locators can be used to update a buffered LOB
            -- through the buffering subsystem

        -- The following update through lbs_loc3 will also
        -- result in an error
        OCILobFileCopy(.., lbs_loc3, lbs_loc2, ..);

        if (exception)
            goto exception_handler;
            -- Locators enabled for buffering cannot be used with
            -- operations like Append, Copy, Trim etc.

        -- When done, flush LOB's buffer to the server:
        OCILobFlushBuffer(.., lbs_loc2, OCI_LOB_BUFFER_NOFREE);

        if (exception)
           goto exception_handler;
           -- This flushes all the modified pages in the LOB's buffer,
           -- and resets lbs_loc2 from updated to read consistent
           -- locator. The modified pages remain in the buffer
           -- without freeing memory.  These pages can be aged
           -- out if necessary.

        -- Disable locators for buffered mode access to LOB */
        OCILobDisableBuffering(lbs_loc1);
        OCILobDisableBuffering(lbs_loc2);
        OCILobDisableBuffering(lbs_loc3);

        if (exception)
           goto exception_handler;
           -- This disables the three locators for buffered access,
           -- and frees up the LOB's buffer resources.

        exception_handler:
        handle_exception_reporting();
        exec_statement("rollback to savepoint lbs_savepoint");
}
```

最大のパフォーマンスを引き出すためのユーザー・ガイドライン

- LOB のサイズは大きいので、LOB 値の大きなまとまり（chunk）を一度に書き込んだり、読み込んだりすることによって、最高のパフォーマンスが得られます。これは次のような場合に便利です。

a. クライアント側から LOB にアクセスしており、クライアントがサーバーと異なるノードにいる場合、大量の読込みまたは書込みをする際にネットワーク・オーバーヘッドが削減される。

b. 'NOCACHE' オプションを使っている場合、少量の読込み / 書込みを行うたびに I/O が発生する。大量の読込み / 書込みを行うことで I/O が削減されます。

c. LOB に書込みを行うと、LOB CHUNK の新しいバージョンが作成される。そのため、一度の書込みの量が少ないと、書込みが行われるたびに新しいバージョンを作成するコストがかかります。ロギングがオンになっている場合は、CHUNK は、REDO ログにも保存されます。

- クライアントの LOB データの小さい部分に対して読取り / 書込みを行う必要がある場合、LOB バッファリングを使用します。OCILobEnableBuffering()、OCILobDisableBuffering()、OCILobFlushBuffer()、OCILobWrite()、OCILobRead() を参照してください。基本的には、LOB データの小さい部分に対して読取り / 書込みを行う前には LOB バッファリングをオンにします。

  関連項目： LOB バッファリングの詳細は、6-43 ページの「LOB バッファリング・サブシステム」を参照してください。

- コールバックとともに OCILobWrite() および OCILobRead() を使うと、データが、LOB へまたは LOB からストリームの形で送られます。入力時に「amount」パラメータに、書込み全体の長さが設定されていることを確認してください。

- LOB に対してチェックアウト / チェックイン・モデルを使います。LOB は、次の操作について最適化されます。

  a. SQL UPDATE を発行して LOB 全体を置換する。

  b. クライアントに LOB データ全体をコピーし、その LOB データをクライアント側で変更し、LOB データ全体をデータベースにコピーする。この処理は、ストリームのある OCILobRead() と OCILobWrite() を使用して行われます。

可変幅文字データの処理

可変幅文字データ（例 : JA16SJIS、JA16EUC）は BLOB および CLOB、NCLOB ではサポートされません。ただし、BLOB には任意のデータを含むことができます。CLOB/NCLOB には可変幅文字セットを保存できないので、可変幅文字を BLOB に保存して、キャラクタ・セットの変換を行う必要性が生じる場合があります。この方法には、これらの変換を自分で行わなければならないことと、オフセットと amount パラメータを文字数ではなくバイト数に変えなければならないという欠点があります。つまり、指定したバイト数が正確でないために、テキスト情報を BLOB から取り出せても、可変幅文字が半分にカットされてしまう危険性があります。このような処置を行わないように、十分注意してください。

同様に、BFILE にもテキストを含む任意のデータが含められます。もう一度述べますが、テキストを保存する場合は、自らキャラクタ・セットの変換を行い、オフセットと amount パラメータをバイト数で指定しなければなりません。

上述のとおり、CLOB では固定幅シングル・バイト・データが保存され、NCLOB では、固定幅マルチ・バイト・レコードが保存されます。どちらにおいても、可変幅データはサポートされていません。

このことからもわかるように、データベースのキャラクタ・セットが可変幅で、ユーザーが CLOB 列を持つ表を作成しようとすると失敗します。ほとんどがこのケースですが、実際は多少異なります。

- システム・ユーザー以外のユーザーが CLOB 列を持つ表を作成しようとすると失敗します。
- システム・ユーザーは CLOB 列を持つ表を作成できます。ただし、CLOB 列に NULL 以外の値がある場合、引き続き表に挿入を行うと失敗します。

これらは、NCLOB とデータベース各国文字キャラクタ・セットについても同様です。

ユーザーが可変幅のキャラクタ・セットを持つ LOB に NULL 以外の値を挿入すると SQL DML が必ず失敗する一方で、SQL DDL を成功させる理由は、基礎を形成する CHAR(NCHAR) キャラクタ・セットにかかわらず、いくつかの異なるデータベースに同じ表を作成して存在させるためです。ユーザーは、アプリケーションを 1 つ作成して、CHAR(NCHAR) キャラクタ・セットが、挿入により可変幅の LOB が NULL に設定されるような可変幅であるデータベースに対しては少しだけ変更します。

LOB リファレンス

リファレンス概要

明示的に示してはいませんが、この項は大きく分けると次の内容で構成されています。

- 初期化に使用する SQL DML 関数の EMPTY_BLOB()、および EMPTY_CLOB()、BFILENAME()(この概要のすぐ後から始まります)。
- LOB を操作するために OCI が提供する方法のサマリー (6-59 ページの「LOB の操作に OCI を使用する」から始まります)。
- DBMS_LOB パッケージ、およびすべての関数とプロシージャのリスト (6-62 ページの「DBMS_LOB パッケージ」から始まります)。この項では、LOB の基礎となる技術仕様部の主な部分について説明します。
- Oracle8 の最初のリリース時点における LOB の使用に関する制約の簡単なリスト。

EMPTY_BLOB() 関数と EMPTY_CLOB() 関数

NULL または非 NULL の内部 LOB を空 (EMPTY) に初期化するために、SQL DML の INSERT 文または UPDATE 文で特別関数 EMPTY_BLOB () および EMPTY_CLOB () を使用できます。これらの関数は、Oracle8 SQL DML では特別関数として使用できますが、DBMS_LOB パッケージには含まれていません。

OCI または DBMS_LOB パッケージを使用して内部 LOB にデータの書込みを始める前に、LOB 列を非 NULL にする必要があります。つまり、LOB 列には、空の LOB 値または移入された LOB 値を参照するロケータが含まれるようにしなければなりません。INSERT 文の VALUES 句で EMPTY_BLOB() 関数を使用することによって、BLOB 列の値を空に初期化できます。同様に、関数 EMPTY_CLOB() を使用することによって、CLOB 列または NCLOB 列の値を初期化できます。

構文

```
FUNCTION EMPTY_BLOB( ) RETURN BLOB;
FUNCTION EMPTY_CLOB( ) RETURN CLOB;
```

パラメータ

なし。

戻り値

EMPTY_BLOB() によって BLOB 型の空のロケータが戻され、EMPTY_CLOB() によって CLOB 型の空のロケータが戻されます。EMPTY_CLOB() は、NCLOB に対しても使用できます。

プラグマ

なし。

例外

これらの関数を SQL INSERT 文の VALUES 句か、SQL UPDATE 文の SET 句のソースとして使用しないと例外が発生します。

例

SQL DML での EMPTY_BLOB() の使用例を次に示します。

```
INSERT INTO lob_table VALUES (1001, EMPTY_BLOB(), 'abcde', NULL, NULL);
UPDATE lob_table SET c_lob = EMPTY_CLOB() WHERE key_value = 1001;
INSERT INTO lob_table VALUES (1002, NULL, NULL, NULL, NULL);
```

PL/SQL プログラムにおける、EMPTY_BLOB () と EMPTY_CLOB () の正しい使用例と誤った使用例を次に示します。

```
DECLARE
  loba         BLOB;
  lobb         CLOB;
  read_offset  INTEGER;
  read_amount  INTEGER;
  rawbuf       RAW(20);
  charbuf      VARCHAR2(20);
BEGIN
  loba := EMPTY_BLOB();
  read_amount := 10; read_offset := 1;
  -- the following read will fail
  dbms_lob.read(loba, read_amount, read_offset, rawbuf);

  -- the following read will succeed;
  UPDATE lob_table SET c_lob = EMPTY_CLOB() WHERE key_value =
         1002 RETURNING c_lob INTO lobb;
dbms_lob.read(lobb, read_amount, read_offset, charbuf);
  dbms_output.put_line('lobb value: ' || charbuf);
```

BFILENAME() 関数

BFILENAME() 関数は、特定の行の BFILE 列または BFILE 属性をサーバーのファイル・システム内の物理ファイルに関連付けることによって初期化する、SQL INSERT の一部としてコールしなければなりません。

この関数への directory_alias パラメータで表される DIRECTORY オブジェクトは、この関数が SQL DML または PL/SQL プログラムでコールされる前に、SQL DDL を使用して定義されていなければなりません。BFILENAME() の後に、CREATE DIRECTORY() コマンドをコールできます。ただし、ターゲット・オブジェクトは、実際に BFILE ロケータを使用する時点で（たとえば、OCILobFileOpen() または DBMS_LOB.FILEOPEN() のような操作にパラメータとして使用される場合）存在していなければなりません。

BFILENAME() では、この DIRECTORY オブジェクトに対する権限の妥当性チェックは行われず、DIRECTORY オブジェクトが表す物理ディレクトリが実際に存在するかどうかもチェックされないことに注意してください。このようなチェックは、BFILENAME() 関数によって初期化された BFILE ロケータを使用するファイル・アクセス時に限り実行されます。

BFILE 列を初期化するために、SQL の INSERT 文および UPDATE 文の一部として、BFILENAME() を使用できます。また、BFILENAME() を使用して PL/SQL プログラムの BFILE ロケータ変数を初期化し、そのロケータをファイル操作に使用することもできます。ただし、対応するディレクトリ別名またはファイル名 (あるいはその両方) が存在しない場合、この変数を使用する PL/SQL DBMS_LOB ルーチンでエラーが発生します。

BFILENAME() 関数の directory_alias パラメータは、ディレクトリ名の大文字と小文字の区別に注意して指定しなければなりません。これについて次の例で説明します。

関連項目： 6-16 ページの「DIRECTORY 名の指定」

構文

```
FUNCTION BFILENAME(directory_alias IN VARCHAR2,
                   filename IN VARCHAR2)
RETURN BFILE;
```

関連項目： ディレクトリ名に大文字を使用する際の詳細は、6-16 ページの「DIRECTORY 名の指定」を、同等の OCI ベースのルーチンについては、『Oracle コール・インタフェース・プログラマーズ・ガイド』の「OCILobFileSetName()」を参照してください。

パラメータ

表 6–1　FILENAME パラメータ

パラメータ名	意味
directory_alias	CREATE DIRECTORY コマンドを使用して作成した DIRECTORY オブジェクトの名前
filename	サーバーのオペレーティング・システム・ファイルの名前

戻り値

成功した場合は、BFILE ロケータ。

directory_alias が定義されていない場合は、NULL。

プラグマ

なし。

例外

なし。

例

SCOTT_DIR に置かれたファイル「scott.dat」および Mary_Dir に置かれたファイル「mary.dat」にアクセスするには、BFILE ロケータを次のように初期化しなければなりません。

```
DECLARE
    fil_1, fil_2 BFILE;
  result INTEGER;
BEGIN
    fil_1 := BFILENAME('SCOTT_DIR', 'scott.dat');
    fil_2 := BFILENAME('Mary_Dir', 'mary.dat');
    DBMS_LOB.FILEOPEN(fil_1);
    DBMS_LOB.FILEOPEN(fil_2);
    result := DBMS_LOB.COMPARE(fil_1, fil_2);
    IF (result != 0)
    THEN
      DBMS_OUTPUT.PUT_LINE('The two files are different');
    END IF;
    DBMS_LOB.FILECLOSE(fil_1);
    DBMS_LOB.FILECLOSE(fil_2);
    -- FILEOPEN will fail with the following initialization (in
       lowercase)
    fil_1 := BFILENAME('scott_dir', 'scott.dat');
    DBMS_LOB.FILEOPEN(fil_1);

-- this is an error
END;

INSERT INTO lob_table VALUES (21, NULL, NULL, NULL,
    BFILENAME('SCOTT_DIR','scott.dat'));
INSERT INTO lob_table VALUES (12, NULL, NULL, NULL,
    BFILENAME('Mary_Dir','mary.dat'));

DECLARE
    fil_1, fil_2 BFILE;
    result INTEGER;

BEGIN
    SELECT f_lob INTO fil_1 FROM lob_table WHERE key_value = 21;
    SELECT f_lob INTO fil_2 FROM lob_table WHERE key_value = 12;
    DBMS_LOB.FILEOPEN(fil_1);
    DBMS_LOB.FILEOPEN(fil_2);
    result := DBMS_LOB.COMPARE(fil_1, fil_2);
    IF (result != 0)
    THEN
```

```
      DBMS_OUTPUT.PUT_LINE('The two files are different');
    END IF;
    DBMS_LOB.FILECLOSE(fil_1);
    DBMS_LOB.FILECLOSE(fil_2);
END;
```

関連項目： DMBS_LOB.FILEGETNAME()

LOB の操作に OCI を使用する

OCI には、BLOB および CLOB、NCLOB、BFILE に格納されたデータのアクセスに使用できる関数が含まれています。これらの関数を次に簡単に説明します。

関連項目： パラメータ、パラメータ型、戻り値、サンプル・コードも含めた詳しい説明は、『Oracle 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。

表 6–2 LOB 操作用の OCI 関数

OCI 関数	説明
OCILobAppend()	LOB 値を別の LOB に追加します。
OCILobAssign()	LOB ロケータを別の LOB ロケータに割り当てます。
OCILobCharSetForm()	LOB のキャラクタ・セット・フォームを戻します。
OCILobCharSetId()	LOB のキャラクタ・セット ID を戻します。
OCILobCopy()	LOB の一部を別の LOB にコピーします。
OCILobDisableBuffering()	バッファリング・サブシステムを使用禁止にします。
OCILobEnableBuffering()	これ以降の LOB データの読取りおよび書込みに、LOB バッファリング・サブシステムを使用します。
OCILobErase()	指定のオフセットから開始して、LOB の一部を消去します。
OCILobFileClose()	オープンしている 1 つの BFILE をクローズします。
OCILobFileCloseAll()	オープンしている BFILE をすべてクローズします。
OCILobFileExists()	BFILE が存在するかどうかをチェックします。
OCILobFileGetName()	BFILE の名前を戻します。
OCILobFileIsOpen()	BFILE がオープンしているかどうかをチェックします。
OCILobFileOpen()	1 つの BFILE をオープンします。

表 6–2　LOB 操作用の OCI 関数

OCI 関数	説明
`OCILobFileSetName()`	ロケータ内に BFILE の名前を設定します。
`OCILobFlushBuffer()`	LOB バッファリング・サブシステムからデータベースへフラッシュ変更します（切断）。
`OCILobGetLength()`	LOB または BFILE の長さを戻します。
`OCILobIsEqual()`	2 つの LOB ロケータが同じ LOB を参照しているかどうかをチェックします。
`OCILobLoadFromFile()`	BFILE のデータを内部 LOB にロードします。
`OCILobLocatorIsInit()`	LOB ロケータが初期化されているかどうかをチェックします。
`OCILobLocatorSize()`	LOB ロケータのサイズを戻します。
`OCILobRead()`	非 NULL の LOB または BFILE の指定部分をバッファに読み込みます。
`OCILobTrim()`	LOB を切り捨てます。
`OCILobWrite()`	バッファのデータを LOB に書き込み、既存データを上書きします。

次のチャートでは、LOB アクセスに関する 2 つのインタフェースを比較します。

表 6–3　LOB アクセスに関する DBMS_LOB インタフェースと OCI インタフェースの比較

OCI (ociap.h)	DBMS_LOB (dbmslob.sql)
N/A	DBMS_LOB.COMPARE()
N/A	DBMS_LOB.INSTR()
N/A	DBMS_LOB.SUBSTR()
OCILobAppend	DBMS_LOB.APPEND()
OCILobAssign	N/A [Pl/SQL 割当てオペレータを使用する]
OCILobCharSetForm	N/A
OCILobCharSetId	N/A
OCILobCopy	DBMS_LOB.COPY()
OCILobDisableBuffering	N/A
OCILobEnableBuffering	N/A
OCILobErase	DBMS_LOB.ERASE()
OCILobFileClose	DBMS_LOB.FILECLOSE()
OCILobFileCloseAll	DBMS_LOB.FILECLOSEALL()
OCILobFileExists	DBMS_LOB.FILEEXISTS()
OCILobFileGetName	DBMS_LOB.FILEGETNAME()
OCILobFileIsOpen	DBMS_LOB.FILEISOPEN()
OCILobFileOpen	DBMS_LOB.FILEOPEN()
OCILobFileSetName	N/A (BFILENAME オペレータを使用する)
OCILobFlushBuffer	N/A
OCILobGetLength	DBMS_LOB.GETLENGTH()
OCILobIsEqual	N/A [Pl/SQL 等価オペレータを使用する]
OCILobLoadFromFile	DBMS_LOB.LOADFROMFILE()
OCILobLocatorIsInit	N/A [常に初期化する]
OCILobRead	DBMS_LOB.READ()
OCILobTrim	DBMS_LOB.TRIM()
OCILobWrite	DBMS_LOB.WRITE()

DBMS_LOB パッケージ

DBMS_LOB パッケージでは、BLOB および CLOB、NCLOB、BFILE にアクセスするためのルーチンが提供されます。DBMS_LOB を使用して、LOB 全体だけでなく LOB の特定部分にアクセスして操作できます。DBMS_LOB では、BLOB および CLOB、NCLOB の修正だけでなく、読取りも可能で、BFILE に対しては読取り専用の操作も行えます。

すべての DBMS_LOB ルーチンは、LOB ロケータを基に動作します。DBMS_LOB ルーチンを正常に完了させるには、データベース表領域または外部ファイル・システムにすでに存在する LOB を表す入力ロケータを用意しなければなりません。

内部 LOB の場合、最初に SQL DDL を使用して LOB 列を含む表を定義し、次に、SQL DML を使用して、これらの LOB 列内のロケータを初期化、または移入（populate）しなければなりません。

関連項目： 6-21 ページの「LOB ロケータの操作」

外部 LOB の場合は、有効な、既存の物理ディレクトリを表す DIRECTORY オブジェクトが定義され、Oracle 用の読取り許可のある物理ファイルが存在することを確認しなければなりません。オペレーティング・システムがパス名を大 / 小文字区別している場合は、必ず正しい形式でディレクトリを指定してください。

関連項目： 6-16 ページの「BFILE セキュリティ」

LOB がいったん定義、作成されたら、SELECT を実行して LOB ロケータをローカルの PL/SQL LOB 変数に割り当て、LOB 値にアクセスするために、DBMS_LOB への入力パラメータとしてこの変数を使用できます。次の項では、各 DBMS_LOB ルーチンで提供されている例によって、上記の操作を図示します。

パッケージ・ルーチン

BLOB および CLOB、NCLOB の値を修正できるルーチンは、次のとおりです。

- `APPEND()` ― ソース `LOB` の内容を宛先 `LOB` に追加する。
- `COPY()` ― ソース `LOB` の全体または一部を対象となる `LOB` にコピーする。
- `ERASE()` ― `LOB` の全体または一部を消去する。
- LOADFROMFILE() ― BFILE のデータを内部 LOB にロードする。
- TRIM() ― LOB 値を指定された長さまで切り捨てる。
- WRITE() ― LOB の指定されたオフセット以降にデータを書き込む。

LOB 値を読み取る、またはテストするルーチンは、次のとおりです。

- GETLENGTH() ― LOB 値の長さを得る。
- `INSTR()` ― `LOB` におけるパターンの `n` 番目のオカレンスの、一致する位置を戻す。

- READ() ― 指定されたオフセット以降の LOB データを読み込む。
- SUBSTR() ― 指定されたオフセット以降の LOB 値の部分を戻す。

BFILE 固有の読取り専用ルーチンは、次のとおりです。

- FILECLOSE() ― ファイルをクローズする。
- FILECLOSEALL() ― オープンしているすべてのファイルをクローズする。
- FILEEXISTS() ― ファイルがサーバー上に存在するかどうかをチェックする。
- FILEGETNAME() ― ディレクトリ別名およびファイル名を得る。
- FILEISOPEN() ― ファイルが入力 BFILE ロケータを使用してオープンされたかどうかをチェックする。
- FILEOPEN() ― ファイルをオープンする。

データ型

DBMS_LOB ルーチンのパラメータは、次のデータ型を使用します。

- BLOB。ソースまたは宛先のバイナリ LOB 用。
- RAW。ソースまたは宛先のロー・バッファ用（BLOB とともに使用）。
- CLOB。ソースまたは宛先のキャラクタ型 LOB 用（NCLOB を含む）。
- VARCHAR2。ソースまたは宛先の文字バッファ用（CLOB および NCLOB とともに使用）。
- INTEGER。バッファまたは LOB のサイズ、LOB でのオフセット、あるいはアクセス量を指定する。

型定義

DBMS_LOB パッケージでは、特別な型は定義されません。NCLOB は、固定幅のマルチバイト各国語文字セット用 CLOB の特別なケースです。CLOB 用 DBMS_LOB ルーチン仕様部内の「ANY_CS」句によって、入力として CLOB または NCLOB ロケータ変数を受け入れることができます。

関連項目：『Oracle8 Server SQL リファレンス』の「LOB データ型」

定数

DBMS_LOB パッケージでは、次の定数が定義されます。

```
LOBMAXSIZE              4294967295
FILE_READONLY            0
```

Oracle 8.0 でサポートされる最大 LOB サイズは、4GB(2^32) です。ただし、パッケージの amount パラメータおよび offset パラメータは、1 から 4294967295(2^32-1) の範囲の値を持つことができます。

PL/SQL 3.0 言語では、RAW または VARCHAR2 変数の最大サイズが、32767 バイトに指定されています。

注意： 32767 バイトという値は、次の項で MAXBUFSIZE によって表されます。

DBMS_LOB 例外

DBMS_LOB 関数またはプロシージャで、表 6–4 に示す、名前付きの例外が発生する場合があります。

表 6–4 DBMS_LOB 例外

例外	エラー・メッセージの意味	コード
INVALID_ARGVAL	21560	「引数 %s が NULL または無効、範囲外です。」
ACCESS_ERROR	22925	<n> の最大 LOB サイズを超えて読取り / 書込みを行おうとしました。
NO_DATA_FOUND	1403	読取り操作のループ用の EndofLOB インディケータです。
VALUE_ERROR	6502	パラメータ内に無効な値があります。

access_error 22925 「LOB 値の最大許容サイズを超えています。」

noexist_directory 22285 「%s 操作用のディレクトリまたはファイルが存在しません。」

nopriv_directory 22286 「%s 操作を実行するにはファイルまたはディレクトリの権限が不十分です。」

invalid_directory 22287 「%s 操作中に無効または修正されたディレクトリがあります。」

invalid_operation 22288 「ファイル操作： %s に失敗しました。¥n%s」

unopened_file 22289 「オープンしていないファイルで %s 操作を実行できません。」

open_toomany 22290 「オープン・ファイルが最大数を超えました。」

これらのルーチンへの入力パラメータのいずれかが NULL または無効の場合、DBMS_LOB 関数では、NULL 値が戻されますが、DBMS_LOB プロシージャでは例外が発生します。この動作は、Oracle8 の SQL 関数、および Oracle8 の他の組込み PL/SQL パッケージのプロシージャでも一貫しています。

DBMS_LOB セキュリティ

この項では、Oracle Server の使用時に内部 LOB(つまり、BLOB および CLOB、NCLOB) 上で動作する DBMS_LOB ルーチン用のセキュリティ・ドメインについて説明します。

注意 : アノニマス PL/SQL ブロックからコールされた DBMS_LOB ルーチンは、現ユーザーの権限を使用して実行されます。ストアド・プロシージャからコールされた DBMS_LOB ルーチンは、そのストアド・プロシージャの所有者の権限を使用して実行されます。

6-56 ページの「BFILENAME() 関数」で説明した DIRECTORY 機能を使用すれば、BFILE への安全性の高いアクセスが得られます。

DBMS_LOB の一般的な使用上の注意

1. 長さおよび量、オフセットのパラメータは、BLOB と BFILE についてはバイト単位で、CLOB と NCLOB については文字単位で指定されます。

2. PL/SQL 3.0 言語では、RAW および VARCHAR2 のバッファ制約がバイト単位で指定されることに注意してください。たとえば、変数を次のように宣言したとします。

   ```
   charbuf VARCHAR2(3000)
   ```

   charbuf は、3000 の 1 バイト文字または 1500 の 2 バイト固定幅文字を保持できます。このことは、CLOB と NCLOB 用の DBMS_LOB ルーチンにとって、重要な意味があります。

3. CLOB 用 DBMS_LOB ルーチンにある VARCHAR2 バッファのキャラクタ・セットは、CLOB 用のキャラクタ・セットと完全に一致しなければなりません。パッケージの仕様部では、%CHARSET 句によって、キャラクタ・セットの一致を部分的に保証していますが、固定幅キャラクタ・セットが実際は可変幅キャラクタ・セットのサブセットであるといった特定のケースでは、これを保証できない場合もあります。

   したがって、バッファに正しいキャラクタ・セットを提供し、すべての文字を保持できるだけの十分なバッファ・サイズを提供することはユーザーの責任となります。セッション初期化パラメータを基にした変換は一切実行されません。

4. AMOUNT パラメータおよび OFFSET パラメータには、正の、非ゼロの値（つまり、1 以上の値）だけが許可されます。これは、Oracle SQL 文字列関数および演算子に見られる、負のオフセットと範囲は許可されないことを意味します。

5. 特に注記がない限り、offset パラメータのデフォルト値は 1 です。これは、BLOB データや BFILE データの最初のバイト、および CLOB 値や NCLOB 値の最初の文字を指します。AMOUNT パラメータのデフォルト値はありません。したがって、値を明示的に入力しなければなりません。

6. APPEND または COPY、ERASE、TRIM、WRITE などの LOB を変更するルーチンをコールする前に、対象となる内部 LOB を含む行のロックは、お客様が実行する必要があります。これらのルーチンでは、LOB を含む行が暗黙的にロックされません。

BFILE に特有の使用上の注意

1. COMPARE() および INSTR()、SUBSTR() は、DBMS_LOB に特有のものであり、COMPARE() および INSTR()、READ()、SUBSTR()、FILECLOSE()、FILECLOSEALL()、LOADFROMFILE() といった操作は、オープンされた BFILE ロケータ上でだけ動作することに注意してください。つまり、これらのルーチンをコールする前に、FILEOPEN() コールが正常に実行されなければなりません。

2. FILEEXISTS() および FILEGETNAME()、GETLENGTH() の各関数については、ファイルのオープン / クローズ状態は重要ではありません。ただし、ファイルは物理的に存在していなければならず、DIRECTORY オブジェクトとファイルに対して、適切な権限を持っている必要があります。

3. DBMS_LOB パッケージでは、BFILE 操作の同時実行制御メカニズムをサポートしていません。

4. セッションに複数のオープン・ファイルがあり、それらのクローズが適切に処理されなかった場合、FILECLOSEALL() ルーチンを使用して、セッションでオープンしたファイルをすべてクローズし、ファイル操作を最初からやり直すことができます。

5. DIRECTORY の作成者である場合、またはシステム権限を持っている場合、SQL の CREATE OR REPLACE 文および DROP 文、REVOKE 文の使用には、特に注意してください。

   関連項目： 6-18 ページの「DIRECTORY 使用のガイドライン」

   お客様または特定のディレクトリ・オブジェクトの権限受領者がセッションで複数のオープン・ファイルを持っている場合、上記のコマンドはいずれもファイル操作に悪影響をもたらす可能性があります。そのような異常終了が発生した場合に行えることは、FILECLOSEALL() をコールするプログラムまたは無名（アノニマス）ブロックを起動し、ファイルを再度オープンして、ファイル操作を再開することだけです。

6. ユーザー・セッションの間にオープンされたファイルはすべて、セッション終了時に暗黙的にクローズされます。ただし、BFILE に対する操作が正常終了および異常終了した後は、ファイルをクローズすることを強くお薦めします。

   関連項目： 6-19 ページの「BFILE の最大オープン数」

プログラムが正常に終了した場合は、ファイルを適切にクローズすることによって、セッションで同時にオープンされているファイルの数が、SESSION_MAX_OPEN_FILES の値よりも少なくすることができます。

PL/SQL プログラムでプログラム異常終了が発生した場合、PL/SQL プログラムでオープンされていた全ファイルのクローズを実行する、例外ハンドラを必ず提供しなければなりません。これが必要になるのは、例外がいったん発生すると、最新の状態の BFILE 変数にアクセスできるのは例外ハンドラだけだからです。

関連項目： 6-19 ページの「プログラム終了後に BFILE をクローズする」

例外によりプログラム制御が PL/SQL プログラム・ブロックの外にいったん転送されると、オープンした BFILE へのすべての参照が失われます。その結果、オープン・ファイルの件数はさらに大きくなり、SESSION_MAX_OPEN_FILES の値を超える場合もあります。

たとえば、READ 操作が BFILE 値の終わりを超え、NO_DATA_FOUND 例外が生成されたとします。

```
DECLARE
       fil bfile;
       pos INTEGER;
       amt binary_INTEGER;
       buf RAW(40);
BEGIN
       SELECT f_lob INTO fil FROM lob_table WHERE key_value = 21;
       dbms_lob.FILEOPEN(fil, dbms_lob.file_readonly);
       amt := 40; pos := 1 + dbms_lob.getlength(fil); buf := '';
       dbms_lob.read(fil, amt, pos, buf);
       dbms_output.put_line('Read F1 past EOF: '||
           utl_raw.cast_to_varchar2(buf));
       dbms_lob.fileclose(fil);
END;

ORA-01403: no data found
ORA-06512: at "SYS.DBMS_LOB", line 373
ORA-06512: at line 10
```

例外がいったん発生すると、BFILE ロケータの変数ファイルは有効範囲外となり、その変数を使用して、ファイルを操作できません。解決方法は、下に示す例外ハンドラを使用することです。

```
DECLARE
     fil bfile;
     pos INTEGER;
     amt binary_INTEGER;
     buf RAW(40);
BEGIN
```

```
        SELECT f_lob INTO fil FROM lob_table WHERE key_value = 21;
        dbms_lob.FILEOPEN(fil, dbms_lob.file_readonly);
        amt := 40; pos := 1 + dbms_lob.getlength(fil); buf := '';
        dbms_lob.read(fil, amt, pos, buf);
        dbms_output.put_line('Read F1 past EOF: '||
             utl_raw.cast_to_varchar2(buf));
        dbms_lob.fileclose(fil);
        exception
        WHEN no_data_found
        then
          BEGIN
            dbms_output.put_line('End of File reached. Closing file');
            dbms_lob.fileclose(fil);
            -- or dbms_lob.filecloseall if appropriate
          END;
END;
     /

Statement processed.
End of File reached. Closing file
```

一般的に、DBMS_LOB を使用して PL/SQL ブロック内でオープンされたファイルを、ブロックが正常 / 異常終了する前にクローズすることは正しいコーディングの習慣です。

DBMS_LOB.APPEND() プロシージャ

APPEND() プロシージャをコールして、ソース内部 LOB の内容を宛先 LOB に追加できます。このプロシージャで は、ソース LOB 全体が追加されます。次の構文の項に示すとおり、2 つのオーバーロードのプロシージャがあります。

構文

```
PROCEDURE APPEND (dest_lob IN OUT  BLOB,
                  src_lob  IN      BLOB);
PROCEDURE APPEND (dest_lob IN OUT  CLOB  CHARACTER SET ANY_CS,
                  src_lob  IN      CLOB  CHARACTER SET dest_lob%CHARSET);
```

パラメータ

表 6–5　APPEND パラメータ

パラメータ名	意味
dest_lob	データを追加する、内部 LOB 用のロケータ
src_lob	データを読み取る、内部 LOB 用のロケータ

例外

ソース LOB または宛先 LOB のどちらかが NULL の場合は VALUE_ERROR。

例

```
PROCEDURE Example_1a IS
    dest_lob, src_lob BLOB;
BEGIN
    -- get the LOB locators
    -- note that the FOR UPDATE clause locks the row
    SELECT b_lob INTO dest_lob
        FROM lob_table
        WHERE key_value = 12 FOR UPDATE;
    SELECT b_lob INTO src_lob
        FROM lob_table
        WHERE key_value = 21;
    DBMS_LOB.APPEND(dest_lob, src_lob);
    COMMIT;
EXCEPTION
    WHEN some_exception
    THEN handle_exception;
END;

PROCEDURE Example_1b IS
    dest_lob, src_lob BLOB;
BEGIN
    -- get the LOB locators
    -- note that the FOR UPDATE clause locks the row
    SELECT b_lob INTO dest_lob
        FROM lob_table
        WHERE key_value = 12 FOR UPDATE;
    SELECT b_lob INTO src_lob
        FROM lob_table
        WHERE key_value = 12;
    DBMS_LOB.APPEND(dest_lob, src_lob);
    COMMIT;
EXCEPTION
    WHEN some_exception
    THEN handle_exception;
END;
```

DBMS_LOB.COMPARE() 関数

COMPARE() 関数をコールして、2 つの LOB の全体または一部を比較できます。LOB を比較するときは、同じデータ型のものに限られます。つまり、BLOB 型の LOB と他の BLOB、CLOB 型の LOB と他の CLOB、BFILE 型の LOB と他の BFILE といった組合せになります。BFILE については、正常な比較操作を行うために、FILEOPEN() 操作を行ってファイルをあらかじめオープンしておかなければなりません。

offset パラメータおよび amount パラメータで指定された範囲と、データが完全に一致する場合、COMPARE() によって 0（ゼロ）が戻されます。データが一致しない場合、非ゼロの INTEGER が戻されます。

固定幅 n バイトの CLOB については、COMPARE で (4294967295/n) を超える入力量が指定された場合、(4294967295/n) または Max(length(clob1),length(clob2)) のサイズのうち、小さい方の範囲で COMPARE によって文字の一致が行われます。

構文

```
FUNCTION COMPARE (
    lob_1            IN BLOB,
    lob_2            IN BLOB,
    amount           IN INTEGER := 4294967295,
    offset_1         IN INTEGER := 1,
    offset_2         IN INTEGER := 1)
RETURN INTEGER;

FUNCTION COMPARE (
    lob_1            IN CLOB  CHARACTER SET ANY_CS,
    lob_2            IN CLOB  CHARACTER SET lob_1%CHARSET,
    amount           IN INTEGER := 4294967295,
    offset_1         IN INTEGER := 1,
    offset_2         IN INTEGER := 1)
RETURN INTEGER;
FUNCTION COMPARE (
    lob_1            IN BFILE,
    lob_2            IN BFILE,
    amount           IN INTEGER,
    offset_1         IN INTEGER := 1,
    offset_2         IN INTEGER := 1)
RETURN INTEGER;
```

パラメータ

表 6–6　COMPARE パラメータ

パラメータ名	意味
lob_1	最初の比較対象の LOB ロケータ
lob_2	2 番目の比較対象の LOB ロケータ
amount	比較するバイトまたは文字の数
offset_1	比較対象となる最初の LOB(origin: 1) 上のオフセットをバイト数または文字数で表したもの

表 6–6　COMPARE パラメータ

パラメータ名	意味
offset_2	比較対象となる最初の LOB(origin: 1) 上のオフセットをバイト数または文字数で表したもの

戻り値

- INTEGER -- 比較が正常に行われた場合は 0（ゼロ）、そうでない場合はゼロ以外の値。
- 次の場合は、NULL。
  - amount < 1
  - amount > LOBMAXSIZE
  - offset_1 または offset_2 < 1
  - offset_1 または offset_2 > LOBMAXSIZE

プラグマ

```
PRAGMA RESTRICT_REFERENCES(compare, WNDS, WNPS, RNDS, RNPS);
```

例外

BFILE 操作では、入力ロケータを使用してファイルがオープンされていない場合は UNOPENED_FILE、ディレクトリが存在しない場合は NOEXIST_DIRECTORY、ディレクトリに対する権限を持っていない場合は NOPRIV_DIRECTORY、ファイルがオープンされた後、ディレクトリが無効にされた場合は INVALID_DIRECTORY、ファイルが存在しない、またはファイルへのアクセス権限を持っていない場合は INVALID_OPERATION になります。

例

```
PROCEDURE Example2a IS
    lob_1, lob_2      BLOB;
    retval            INTEGER;
BEGIN
    SELECT b_col INTO lob_1 FROM lob_table
        WHERE key_value = 45;
    SELECT b_col INTO lob_2 FROM lob_table
        WHERE key_value = 54;
    retval := DBMS_LOB.COMPARE(lob_1, lob_2, 5600, 33482,
         128);
    IF retval = 0 THEN
      ; /* 比較が正常に行われた場合の処理 */
    ELSE
      ; /* 比較が行われなかった場合の処理 */
```

```
    END IF;
END;

PROCEDURE Example_2b IS
    fil_1, fil_2 BFILE;
    retval INTEGER;
BEGIN
    SELECT f_lob INTO fil_1 FROM lob_table WHERE key_value = 45;
    SELECT f_lob INTO fil_2 FROM lob_table WHERE key_value = 54;
    DBMS_LOB.FILEOPEN(fil_1, DBMS_LOB.FILE_READONLY);
    DBMS_LOB.FILEOPEN(fil_2, DBMS_LOB.FILE_READONLY);
    retval := DBMS_LOB.COMPARE(fil_1, fil_2, 5600,
                               3348276, 2765612);
    IF (retval = 0)
    THEN
        ; /* 比較が正常に行われた場合の処理 */
    ELSE
        ; /* 比較が行われなかった場合の処理 */
    END IF;
    DBMS_LOB.FILECLOSE(fil_1);
    DBMS_LOB.FILECLOSE(fil_2);
END;
```

DBMS_LOB.COPY() プロシージャ

COPY() プロシージャをコールして、ソース内部 LOB の全部または一部を宛先内部 LOB にコピーできます。ソース LOB と宛先 LOB の両方のオフセット、およびコピーするバイト数または文字数を指定できます。

宛先 LOB に指定したオフセットがこの LOB 内に現在あるデータの終わりを超えると、宛先 BLOB または宛先 CLOB に対して、0（ゼロ）バイトの充てん文字または空白が、それぞれ挿入されます。オフセットが宛先 LOB の現在の長さより短い場合、既存のデータは上書きされます。

ソース LOB のデータ長さを超える量を指定しても、エラーにはなりません。したがって、src_offset からの データをソース LOB の終わりにコピーする、ソース LOB からの大量のコピーを指定できます。

構文

```
PROCEDURE COPY (
  dest_lob    IN OUT BLOB,
  src_lob     IN     BLOB,
  amount      IN     INTEGER,
  dest_offset IN     INTEGER := 1,
  src_offset  IN     INTEGER := 1);

PROCEDURE COPY (
```

```
  dest_lob    IN OUT CLOB  CHARACTER SET ANY_CS,
  src_lob     IN     CLOB  CHARACTER SET dest_lob%CHARSET,
  amount      IN     INTEGER,
  dest_offset IN     INTEGER := 1,
  src_offset  IN     INTEGER := 1);
```

パラメータ

表 6–7　COPY パラメータ

パラメータ名	意味
dest_lob	コピー・ターゲットの LOB ロケータ
src_lob	コピー・ソースの LOB ロケータ
amount	コピーするバイトまたは文字の数
dest_offset	コピーを開始するための、宛先 LOB(origin: 1) 内のオフセットをバイト数または文字数で表したもの
src_offset	コピーを開始するための、ソース LOB（origin: 1）内のオフセットを、バイト数または文字数で表したもの

戻り値

なし。

プラグマ

なし。

例外

入力パラメータのいずれかが NULL または無効の場合は VALUE_ERROR。次の場合は、INVALID_ARGVAL。

- src_offset または dest_offset < 1
- src_offset または dest_offset > LOBMAXSIZE
- amount < 1
- amount > LOBMAXSIZE

例

```
PROCEDURE Example_3a IS
    lobd, lobs     BLOB;
    amt            INTEGER := 3000;
BEGIN
```

```
    SELECT b_col INTO lobd
        FROM lob_table
        WHERE key_value = 12 FOR UPDATE;
    SELECT b_col INTO lobs
        FROM lob_table
        WHERE key_value = 21;
    DBMS_LOB.COPY(lobd, lobs, amt);
    COMMIT;
   EXCEPTION
        WHEN some_exception
        THEN handle_exception;
END;

PROCEDURE Example_3b IS
    lobd, lobs     BLOB;
    amt            INTEGER := 3000;
BEGIN
    SELECT b_col INTO lobd
        FROM lob_table
        WHERE key_value = 12 FOR UPDATE;
    SELECT b_col INTO lobs
        FROM lob_table
        WHERE key_value = 12;
    DBMS_LOB.COPY(lobd, lobs, amt);
    COMMIT;
   EXCEPTION
        WHEN some_exception
        THEN handle_exception;
END;
```

DBMS_LOB.ERASE() プロシージャ

ERASE() プロシージャをコールして、内部 LOB の全体または一部を消去できます。offset パラメータによって消去のための開始オフセットが指定され、amount パラメータによって消去するバイト数または文字数が指定されます。

LOB の中央からデータが消去されると、BLOB または CLOB に対して、0（ゼロ）バイトの充てん文字または空白がそれぞれ挿入されます。

実際に消去されるバイト数や文字数は、指定の数だけ消去される前に LOB 値の終わりに達した場合には、amount パラメータで指定した数と異なる場合があります。実際に消去されたバイト数または文字数は、amount パラメータ内に戻されます。

構文

```
PROCEDURE ERASE (
    lob_loc          IN OUT       BLOB,
    amount           IN OUT       INTEGER,
```

```
    offset          IN          INTEGER := 1);

PROCEDURE ERASE (
    lob_loc         IN OUT      CLOB,
    amount          IN OUT      INTEGER,
    offset          IN          INTEGER := 1);
```

パラメータ

表 6–8　ERASE パラメータ

パラメータ名	意味
lob_loc	消去する LOB のロケータ
amount	消去するバイト数（BLOB の場合）または文字数（CLOB の場合）
offset	LOB の先頭からの絶対オフセットをバイト数（BLOB の場合）または文字数（CLOB の場合）で表したもの

戻り値

なし。

プラグマ

なし。

例外

入力パラメータのいずれかが NULL の場合は VALUE_ERROR。

次の場合は、INVALID_ARGVAL。

- AMOUNT < 1 または AMOUNT > LOBMAXSIZE
- OFFSET < 1 または OFFSET > LOBMAXSIZE

例

```
PROCEDURE Example_4 IS
    lobd       BLOB;
    amt        INTEGER := 3000;
BEGIN
    SELECT b_col INTO lobd
        FROM lob_table
        WHERE key_value = 12 FOR UPDATE;
    DBMS_LOB.ERASE(dest_lob, amt, 2000);
    COMMIT;
END;
```

関連項目： DBMS_LOB.TRIM()

DBMS_LOB.FILECLOSE() プロシージャ

FILECLOSE() プロシージャをコールして、入力ロケータによってすでにオープンされている BFILE をクローズできます。Oracle は BFILE への読取りアクセスしかできないことに注意してください。つまり、Oracle から BFILE への書込みはできません。

構文

```
PROCEDURE FILECLOSE (
    file_loc IN OUT BFILE);
```

パラメータ

表 6–9 FILECLOSE パラメータ

パラメータ名	意味
file_loc	クローズする BFILE のロケータ

戻り値

なし。

プラグマ

なし。

例外

file_loc の入力値が NULL の場合は VALUE_ERROR となります。入力ロケータを使用してファイルがオープンされていない場合は UNOPENED_FILE、ディレクトリが存在しない場合は NOEXIST_DIRECTORY、ディレクトリに対する権限を持っていない場合は NOPRIV_DIRECTORY、ファイルがオープンされた後にディレクトリが無効にされた場合は INVALID_DIRECTORY、ファイルが存在しない、またはファイルへのアクセス権限を持っていない場合は INVALID_OPERATION となります。

例

```
PROCEDURE Example_5 IS
    fil BFILE;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 99;
    DBMS_LOB.FILEOPEN(fil);
    -- file operations
    DBMS_LOB.FILECLOSE(fil);
    EXCEPTION
        WHEN some_exception
```

```
        THEN handle_exception;
END;
```

関連項目： DBMS_LOB.FILEOPEN()、DBMS_LOB.FILECLOSEALL()

DBMS_LOB.FILECLOSEALL() プロシージャ

FILECLOSEALL() プロシージャをコールして、セッションでオープンされたすべての BFILE をクローズできます。

構文

```
PROCEDURE FILECLOSEALL;
```

戻り値

なし。

プラグマ

なし。

例外

セッションでオープンされたファイルがない場合は UNOPENED_FILE。

例

```
PROCEDURE Example_6 IS
    fil BFILE;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 99;
    DBMS_LOB.FILEOPEN(fil);
    -- file operations
DBMS_LOB.FILECLOSEALL;
    EXCEPTION
        WHEN some_exception
        THEN handle_exception;
END;
```

関連項目： DBMS_LOB.FILEOPEN()、DBMS_LOB.FILECLOSE()

DBMS_LOB.FILEEXISTS() 関数

FILEEXISTS() 関数をコールして、指定の BFILE ロケータがサーバーのファイル・システム上に実際に存在するファイルを参照しているかどうかを確認できます。

構文

```
FUNCTION FILEEXISTS (
```

```
      file_loc     IN    BFILE)
RETURN INTEGER;
```

パラメータ

表 6–10　FILEEXISTS パラメータ

パラメータ名	意味
file_loc	BFILE のロケータ

戻り値

INTEGER: 物理ファイルが存在する場合は 1、存在しない場合は 0。

次の場合は、NULL。

- file_loc が NULL である。
- file_loc に必要なディレクトリ権限および OS 権限がない。
- OS エラーのため、file_loc を読み取ることができない。

プラグマ

```
PRAGMA RESTRICT_REFERENCES(fileexists, WNDS, RNDS, WNPS, RNPS);
```

例外

ディレクトリが存在しない場合は NOEXIST_DIRECTORY、ディレクトリに対する権限を持っていない場合は NOPRIV_DIRECTORY、ファイルがオープンされた後、ディレクトリが無効にされた場合は INVALID_DIRECTORY となります。

例

```
PROCEDURE Exsmple_7 IS
    fil BFILE;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 12;
    IF (DBMS_LOB.FILEEXISTS(fil))
    THEN
        ; -- file exists code
    ELSE
        ; -- file does not exist code
    END IF;
    EXCEPTION
        WHEN some_exception
        THEN handle_exception;
END;
```

関連項目：　DBMS_LOB.FILEISOPEN

DBMS_LOB.FILEGETNAME() プロシージャ

FILEGETNAME() プロシージャをコールして、任意の BFILE ロケータの dir_alias およびファイル名を判断できます。この関数は、ロケータに割り当てられたディレクトリ別名とファイル名だけを示し、物理ファイルやディレクトリが実際に存在するかどうかは示しません。dir_alias バッファの最大制約値は 30 で、パス名全体の最大制約値は 2000 です。

構文

```
PROCEDURE FILEGETNAME (
      file_loc   IN    BFILE,
      dir_alias  OUT   VARCHAR2
      filename   OUT   VARCHAR2);
```

パラメータ

表 6–11　FILEGETNAME パラメータ

パラメータ名	意味
file_loc	BFILE のロケータ
dir_alias	ディレクトリ別名
filename	BFILE の名前

戻り値

なし。

プラグマ

なし。

例外

入力パラメータのいずれかが NULL または無効の場合は VALUE_ERROR。dir_alias または filename が NULL の場合は INVALID_ARGVAL となります。

例

```
PROCEDURE Example_8 IS
    fil BFILE;
    dir_alias VARCHAR2(30);
    name VARCHAR2(2000);
BEGIN
    IF (DBMS_LOB.FILEEXISTS(fil))
    THEN
        DBMS_LOB.FILEGETNAME(fil, dir_alias, name);
```

```
        DBMS_OUTPUT.PUT_LINE ("Opening " || dir_alias || name);
        DBMS_LOB.FILEOPEN(fil, DBMS_LOB.FILE_READONLY);
        -- file operations
        DBMS_OUTPUT.FILECLOSE(fil);
    END IF;
END;
```

関連項目： BFILENAME() 関数

DBMS_LOB.FILEISOPEN() 関数

FILEISOPEN() 関数をコールして、BFILE が任意の FILE ロケータによってオープンされたかどうかを確認できます。入力 FILE ロケータが DBMS_LOB.FILEOPEN プロシージャに渡されていなければ、そのファイルはこのロケータによってオープンされなかったとみなされます。ただし、別のロケータがこのファイルをオープンした可能性はあります。これはつまり、オープン状態が特定のロケータと関連付けられるということです。

構文

```
FUNCTION FILEISOPEN (
      file_loc   IN    BFILE)
RETURN INTEGER;
```

パラメータ

表 6–12　FILEISOPEN パラメータ

パラメータ名	意味
file_loc	BFILE のロケータ

戻り値

整数。

プラグマ

```
PRAGMA RESTRICT_REFERENCES(fileisopen, WNDS, RNDS, WNPS, RNPS);
```

例外

ディレクトリが存在しない場合は NOEXIST_DIRECTORY、ディレクトリに対する権限を持っていない場合は NOPRIV_DIRECTORY、ファイルがオープンされた後、ディレクトリが無効にされた場合は INVALID_DIRECTORY となります。ファイルが存在しない、またはファイルへのアクセス権限を持っていない場合は INVALID_OPERATION です。

例

```
PROCEDURE Example_9 IS
DECLARE
```

```
    fil      BFILE;
    pos      INTEGER;
    pattern  VARCHAR2(20);
BEGIN
    SELECT f_lob INTO fil FROM lob_table
        WHERE key_value = 12;
    -- open the file
    IF (FILEISOPEN(fil))
    THEN
        pos := DBMS_LOB.INSTR(fil, pattern, 1025, 6);
         -- more file operations
         DBMS_LOB.FILECLOSE(fil);
   ELSE
        ; -- return error
    END IF;
END;
```

関連項目： DBMS_LOB.FILEEXISTS

DBMS_LOB.FILEOPEN() プロシージャ

FILEOPEN プロシージャをコールして、読取り専用アクセスのために BFILE をオープンできます。Oracle から BFILE への書込みはできません。

構文

```
PROCEDURE FILEOPEN (
      file_loc   IN OUT  BFILE,
      open_mode  IN      BINARY_INTEGER := file_readonly);
```

パラメータ

表 6–13　FILEOPEN パラメータ

パラメータ名	意味
file_loc	BFILE のロケータ
open_mode	オープン・モード

戻り値

なし。

プラグマ

なし。

例外

file_loc または open_mode が NULL の場合は、VALUE_ERROR 例外が発生します。open_mode が FILE_READONLY に等しくない場合は、INVALID_ARGVAL 例外が発生します。セッション内のオープン・ファイル数が SESSION_MAX_OPEN_FILES を超える場合は OPEN_TOOMANY、ディレクトリが存在しない場合は NOEXIST_DIRECTORY、ファイルがオープンされた後、ディレクトリが無効にされた場合は INVALID_DIRECTORY、ファイルが存在しない、またはファイルへのアクセス権限を持っていない場合は INVALID_OPERATION が発生します。

例

```
PROCEDURE Example_10 IS
    fil BFILE;
BEGIN
    -- open BFILE
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 99;
    IF (DBMS_LOB.FILEEXISTS(fil))
    THEN
        DBMS_LOB.FILEOPEN(fil, DBMS_LOB.FILE_READONLY);
        -- file operation
        DBMS_LOB.FILECLOSE(fil);
    END IF;
    EXCEPTION
        WHEN some_exception
        THEN handle_exception;
END;
```

関連項目： DBMS_LOB.FILECLOSE()、DBMS_LOB.FILECLOSEALL()

DBMS_LOB.GETLENGTH() 関数

GETLENGTH() 関数をコールすると、指定した LOB の長さが得られます。長さは、バイト数または文字数で戻されます。BFILE について戻される長さには、EOF（もしあれば）が含まれます。以前に行った ERASE() 操作または WRITE() 操作で LOB 内に挿入された 0（ゼロ）バイトまたは空白の充てん文字も、長さに含まれることに注意してください。空の内部 LOB の長さは、0（ゼロ）です。

構文

```
FUNCTION GETLENGTH (
  lob_loc IN BLOB)
RETURN INTEGER;

FUNCTION GETLENGTH (
  lob_loc IN CLOB CHARACTER SET ANY_CS)
RETURN INTEGER;

FUNCTION GETLENGTH (
  lob_loc IN BFILE)
```

```
RETURN INTEGER;
```

パラメータ

表 6–14　GETLENGTH パラメータ

パラメータ名	意味
lob_loc	長さが戻される LOB のロケータ

戻り値

LOB の長さを INTEGER としてバイト数または文字数で表したもの。入力 LOB が NULL の場合は、NULL が戻されます。BFILE については、次のような場合に NULL が戻されます。

- lob_loc が NULL である。
- lob_loc に必要なディレクトリ権限および OS 権限がない。
- OS エラーのため、lob_loc を読み取ることができない。

プラグマ

```
PRAGMA RESTRICT_REFERENCES(getlength, WNDS, WNPS, RNDS, RNPS);
```

例外

なし。

例

```
PROCEDURE Example_11a IS
    lobd        BLOB;
    length      INTEGER;
BEGIN
    -- get the LOB locator
    SELECT b_lob INTO lobd FROM lob_table
        WHERE key_value = 42;
    length := DBMS_LOB.GETLENGTH(lob_loc);
    IF length IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('LOB is null.');
    ELSE
        DBMS_OUTPUT.PUT_LINE('The length is '
            || length);
    END IF;
END;
PROCEDURE Example_11b IS
DECLARE
    len INTEGER;
    fil BFILE;
```

```
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 12;
    len := DBMS_LOB.LENGTH(fil);
END;
```

DBMS_LOB.INSTR() 関数

INSTR 関数をコールして、指定したオフセットから開始して LOB におけるパターンの n 番目のオカレンスの、一致する位置を戻すことができます。CLOB については、VARCHAR2 バッファ（PATTERN パラメータ）および LOB 値は、同じキャラクタ・セット（1 バイトまたは固定幅マルチバイト）からのものでなければなりません。BFILE については、正常な比較操作を行うために、FILEOPEN() 操作を行ってファイルをあらかじめオープンしておかなければなりません。

パターン一致のために RAW パラメータまたは VARCHAR2 パラメータを受け付ける操作 (たとえば INSTR) では、パターン・パラメータまたはサブストリングでの正規表現または特別な一致文字 (SQL LIKE の場合のような) はサポートされません。

構文

```
FUNCTION INSTR (
  lob_loc    IN   BLOB,
  pattern    IN   RAW,
  offset     IN   INTEGER := 1,
  nth        IN   INTEGER := 1)
RETURN INTEGER;

FUNCTION INSTR (
  lob_loc    IN   CLOB      CHARACTER SET ANY_CS,
  pattern    IN   VARCHAR2  CHARACTER SET lob_loc%CHARSET,
  offset     IN   INTEGER := 1,
  nth        IN   INTEGER := 1)
RETURN INTEGER;

FUNCTION INSTR (
  lob_loc    IN   BFILE,
  pattern    IN   RAW,
  offset     IN   INTEGER := 1,
  nth        IN   INTEGER := 1)
RETURN INTEGER;
```

パラメータ

表 6–15　INSTR パラメータ

パラメータ名	意味
lob_loc	テストする LOB のロケータ

表 6–15　INSTR パラメータ

パラメータ名	意味
pattern	テストするパターン。パターンは、BLOB の場合は RAW バイトのグループ、CLOB の場合は文字列（VARCHAR2）です。
offset	パターン一致を開始する位置をバイト数（BLOB）または文字数（CLOB）で表した絶対オフセット
nth	1 から始まる、オカレンス番号

戻り値

一致したパターンの開始オフセットを、INTEGER としてバイト数または文字数で表したもの。パターンが見つからない場合は、0 が戻されます。

次の場合は、NULL が戻されます。

- 1 つまたは複数の IN パラメータが NULL または無効。
- OFFSET < 1 または OFFSET > LOBMAXSIZE
- nth < 1
- nth > LOBMAXSIZE

プラグマ

```
PRAGMA RESTRICT_REFERENCES(instr, WNDS, WNPS, RNDS, RNPS);
```

例外

BFILE では、ファイルが入力ロケータを使用してオープンされていない場合には UNOPENED_FILE、ディレクトリが存在しない場合は NOEXIST_DIRECTORY、ディレクトリに対する権限を持っていない場合は NOPRIV_DIRECTORY、ファイルがオープンされた後にディレクトリが無効にされた場合は INVALID_DIRECTORY、ファイルが存在しない、またはファイルへのアクセス権限を持っていない場合は INVALID_OPERATION となります。

例

```
PROCEDURE Example_12a IS
    lobd        CLOB;
    pattern     VARCHAR2 := 'abcde';
    position    INTEGER := 10000;
BEGIN
-- get the LOB locator
    SELECT b_col INTO lobd
        FROM lob_table
        WHERE key_value = 21;
    position := DBMS_LOB.INSTR(lobd,
                        pattern, 1025, 6);
```

```
    IF position = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Pattern not found');
    ELSE
        DBMS_OUTPUT.PUT_LINE('The pattern occurs at '
                || position);
    END IF;
END;

PROCEDURE Example_12b IS
DECLAR    E
    fil BFILE;
    pattern VARCHAR2;
    pos INTEGER;
BEGIN
    -- initialize pattern
    -- check for the 6th occurrence starting from 1025th byte
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 12;
    DBMS_LOB.FILEOPEN(fil, DBMS_LOB.FILE_READONLY);
    pos := DBMS_LOB.INSTR(fil, pattern, 1025, 6);
    DBMS_LOB.FILECLOSE(fil);
END;
```

関連項目： DBMS_LOB.SUBSTR()

DBMS_LOB.LOADFROMFILE() プロシージャ

LOADFROMFILE() プロシージャをコールして、ソース外部 LOB(BFILE) の全部または一部を宛先内部 LOB にコピーできます。ソース LOB と宛先 LOB の両方のオフセット、およびソース BFILE からコピーするバイト数を指定できます。コピーする量と src_offset は BFILE を参照しているためバイト数で指定しますが、宛先オフセットは BLOB と CLOB のそれぞれに対してバイト数でも文字数でも指定できることに注意してください。

注意： このプロシージャを使用する前に、入力 BFILE をあらかじめオープンしておかなければなりません。また、バイナリ BFILE データの CLOB へのロード時には、キャラクタ・セットの変換は暗黙的には実行されません。BFILE データは、あらかじめデータベースの CLOB と同じキャラクタ・セットにしておかなければなりません。キャラクタ・セットが一致しているかどうかのエラー検査は行われません。

宛先 LOB に指定したオフセットがこの LOB 内に現在あるデータの終わりを超えると、宛先 BLOB または宛先 CLOB に対して、0（ゼロ）バイトの充てん文字または空白が、それぞれ挿入されます。オフセットが宛先 LOB の現在の長さより短い場合、既存のデータは上書きされます。

ソース BFILE のデータ長さを超える量を指定しても、エラーにはなりません。したがって、src_offset からのデータを BFILE の終わりまでコピーする、BFILE からの大量のコピーを指定できます。

構文

```
PROCEDURE loadfromfile (
   dest_lob    IN OUT BLOB,
   src_file    IN     BFILE,
   amount      IN     INTEGER,
   dest_offset IN     INTEGER := 1,
   src_offset  IN     INTEGER  := 1);

PROCEDURE LOADFROMFILE(
   dest_lob    IN OUT CLOB CHARACTER SET ANY_CS,
   src_file    IN     BFILE,
   amount      IN     INTEGER,
   dest_offset IN     INTEGER := 1,
   src_offset  IN     INTEGER := 1);
```

パラメータ

表 6–16　INSTR パラメータ

パラメータ名	意味
dest_lob	ロード・ターゲットの LOB ロケータ
src_file	ロード・ソースの BFILE ロケータ
amount	BFILE からロードするバイト数
dest_offset	ロードを開始するための、宛先 LOB(origin: 1) 内のオフセットをバイト数または文字数で表したもの
src_offset	ロードを開始するための、ソース BFILE(origin: 1) 内のオフセットをバイト数で表したもの

戻り値

なし

プラグマ

なし。

例外

入力パラメータのいずれかが NULL または無効の場合は VALUE_ERROR。

次の場合は、INVALID_ARGVAL。

- src_offset または dest_offset < 1
- src_offset または dest_offset > LOBMAXSIZE
- amount < 1
- amount > LOBMAXSIZE

例

```
PROCEDURE Example_l2f IS
  lobd       BLOB;
  fils       BFILE := BFILENAME('SOME_DIR_OBJ','some_file');
  amt        INTEGER := 4000;
BEGIN
  DBMS_LOB.FILEOPEN(fils, dbms_lob.file_readonly);
  DBMS_LOB.LOADFROMFILE(lobd, fils, amt);
  COMMIT;
  DBMS_LOB.FILECLOSE(fils);
END;
```

DBMS_LOB.READ() プロシージャ

READ() プロシージャをコールして LOB の一部を読み取り、LOB の先頭からの絶対 offset から開始して、指定の量を buffer パラメータ内に戻すことができます。

実際に読み取られたバイト数または文字数は、amount パラメータ内に戻されます。READ() 実行中に LOB 値の終わりに達した場合、amount は 0（ゼロ）に設定され、NO_DATA_FOUND 例外が発生します。

構文

```
PROCEDURE READ (
  lob_loc   IN      BLOB,
  amount    IN OUT  BINARY_INTEGER,
  offset    IN      INTEGER,
  buffer    OUT     RAW);

PROCEDURE READ (
  lob_loc   IN      CLOB     CHARACTER SET ANY_CS,
  amount    IN OUT  BINARY_INTEGER,
  offset    IN      INTEGER,
  buffer    OUT     VARCHAR2 CHARACTER SET lob_loc%CHARSET);

PROCEDURE READ (
  lob_loc   IN       BFILE,
  amount    IN OUT   BINARY_INTEGER,
  offset    IN       INTEGER,
  buffer    OUT      RAW);
```

パラメータ

表 6–17　READ パラメータ

パラメータ名	意味
lob_loc	読み取る LOB のロケータ
amount	読み取るバイト数または文字数
offset	LOB の先頭（origin: 1）からのオフセットをバイト数（BLOB の場合）または文字数（CLOB の場合）で表したもの
buffer	読取り操作用の出力バッファ

戻り値

なし。

プラグマ

なし。

例外

READ によって、次のような例外が発生する場合があります。

- VALUE_ERROR
  - lob_loc または amount、offset のいずれかのパラメータが NULL である
- INVALID_ARGVAL
  - AMOUNT < 1
  - AMOUNT > MAXBUFSIZE
  - OFFSET < 1
  - OFFSET > LOBMAXSIZE
  - バイト数または文字数の単位で AMOUNT が BUFFER 容量よりも大きい。
- NO_DATA_FOUND
  - LOB の終わりに達し、LOB から読み取るバイトや文字が残っていない。AMOUNT の値が 0 である。
- BFILE 操作では、ファイルが入力ロケータを使用してオープンされていない場合には UNOPENED_FILE、ディレクトリが存在しない場合は NOEXIST_DIRECTORY、ディレクトリに対する権限を持っていない場合は NOPRIV_DIRECTORY、ファイルがオープンされた後、ディレクトリが無効にされた場合は INVALID_DIRECTORY、ファイルが存在しない、またはファイルへのアクセス権限を持っていない場合は INVALID_OPERATION となります。

例

```
PROCEDURE Example_13a IS
    src_lob        BLOB;
    buffer         RAW;
    amt            BINARY_INTEGER := 32767;
    pos            INTEGER := 2147483647;
BEGIN
    SELECT b_col INTO src_lob
        FROM lob_table
        WHERE key_value = 21;
    LOOP
        DBMS_LOB.READ (src_lob, amt, pos, buffer);
        /* バッファの処理 */
        pos := pos + amt;
    END LOOP;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('End of data');
END;

PROCEDURE Example_13b IS
    fil BFILE;
    buf RAW(32767);
    amt BINARY_INTEGER := 32767;
    pos INTEGER := 2147483647;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 21;
    DBMS_LOB.FILEOPEN(fil, DBMS_LOB.FILE_READONLY);
    LOOP
        DBMS_LOB.READ(fil, amt, pos, buf);
        -- process contents of buf
        pos := pos + amt;
    END LOOP;
    EXCEPTION
      WHEN NO_DATA_FOUND
      THEN
      BEGIN
        DBMS_OUTPUT.PUTLINE ('End of LOB value reached');
        DBMS_LOB.FILECLOSE(fil);
      END;
END;

/* ストリーム I/O よりもブロック I/O の方が */
/* より効率的に OS 上で実行される I/O の例 */
PROCEDURE Example_13c IS
    fil BFILE;
    amt BINARY_INTEGER := 1024; -- or n x 1024 for reading n
```

```
    buf RAW(1024); -- blocks at a time
    tmpamt BINARY_INTEGER;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 99;
    DBMS_LOB.FILEOPEN(fil, DBMS_LOB.FILE_READONLY);
    LOOP
        DBMS_LOB.READ(fil, amt, pos, buf);
        -- process contents of buf
        pos := pos + amt;
    END LOOP;
    EXCEPTION
        WHEN NO_DATA_FOUND
        THEN
            BEGIN
                DBMS_OUTPUT.PUTLINE ('End of data reached');
                DBMS_LOB.FILECLOSE(fil);
            END;
END;
```

DBMS_LOB.SUBSTR() 関数

SUBSTR() 関数をコールして、LOB の先頭からの絶対 offset から開始し、LOB の amount 分のバイト数または文字数を戻すことができます。

固定幅で n バイトの CLOB については、SUBSTR() の入力量が (32767/n) よりも多く指定された場合、長さ (32767/n) の文字バッファ、または CLOB の長さのうち、短いほうが、SUBSTR() によって戻されます。

構文

```
FUNCTION SUBSTR(
  lob_loc     IN    BLOB,
  amount      IN    INTEGER := 32767,
  offset      IN    INTEGER := 1)
RETURN RAW;

FUNCTION SUBSTR(
  lob_loc     IN    CLOB   CHARACTER SET ANY_CS,
  amount      IN    INTEGER := 32767,
  offset      IN    INTEGER := 1)
RETURN VARCHAR2 CHARACTER SET lob_loc%CHARSET;

FUNCTION SUBSTR(
  lob_loc     IN    BFILE,
  amount      IN    INTEGER := 32767,
  offset      IN    INTEGER := 1)
RETURN RAW;
```

パラメータ

表 6–18　SUBSTR パラメータ

パラメータ名	意味
lob_loc	読み取る LOB のロケータ
amount	読み取るバイト数または文字数
offset	LOB の先頭（origin: 1）からのオフセットをバイト数（BLOB の場合）または文字数（CLOB の場合）で表したもの

戻り値

パラメータ内に BLOB または BFILE を持つ、関数のオーバーロードの場合は RAW。

CLOB バージョンの場合は VARCHAR2。

次の場合は、NULL。

- 入力パラメータのいずれかが NULL である。
- AMOUNT < 1
- AMOUNT > 32767
- OFFSET < 1
- OFFSET > LOBMAXSIZE

プラグマ

```
PRAGMA RESTRICT_REFERENCES(substr, WNDS, WNPS, RNDS, RNPS);
```

例外

BFILE 操作では、ファイルが入力ロケータを使用してオープンされていない場合には UNOPENED_FILE、ディレクトリが存在しない場合は NOEXIST_DIRECTORY、ディレクトリに対する権限を持っていない場合は NOPRIV_DIRECTORY、ファイルがオープンされた後、ディレクトリが無効にされた場合は INVALID_DIRECTORY、ファイルが存在しない、またはファイルへのアクセス権限を持っていない場合は INVALID_OPERATION となります。

例

```
PROCEDURE Example_14a IS
    src_lob        CLOB;
    pos            INTEGER := 2147483647;
    buf            VARCHAR2(32000);
BEGIN
    SELECT c_lob INTO src_lob FROM lob_table
        WHERE key_value = 21;
```

```
    buf := DBMS_LOB.SUBSTR(src_lob, 32767, pos);
    /* データの処理 */
END;

PROCEDURE Example_14b IS
    fil BFILE;
    pos INTEGER := 2147483647;
    pattern RAW;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 21;
    DBMS_LOB.FILEOPEN(fil, DBMS_LOB.FILE_READONLY);
    pattern := DBMS_LOB.SUBSTR(fil, 255, pos);
    DBMS_LOB.FILECLOSE(fil);
END;
```

関連項目： DBMS_LOB.INSTR()、DBMS_LOB.READ()

DBMS_LOB.TRIM() プロシージャ

TRIM() プロシージャをコールして、newlen パラメータで指定する長さに合わせて内部 LOB 値を切り捨てることができます。長さは、BLOB の場合はバイト数で、CLOB の場合は文字数で指定します。

空の LOB に対して TRIM() を実行すると、何も起こらず、TRIM() によってエラーも戻されません。newlen に指定する長さが LOB のサイズより大きい場合は、例外が発生します。

構文

```
FUNCTION TRIM (
    lob_loc        IN    BLOB,
    newlen         IN    INTEGER);

FUNCTION TRIM (
    lob_loc        IN    CLOB,
    newlen         IN    INTEGER):
```

パラメータ

表 6–19　TRIM パラメータ

パラメータ名	意味
lob_loc	長さを切り捨てる内部 LOB のロケータ
newlen	切り捨てられた後の LOB 値の長さを、BLOB の場合はバイト数で、CLOB の場合は文字数で表したもの

戻り値

なし。

プラグマ

なし。

例外

lob_loc が NULL の場合 VALUE_ERROR。

次の場合は、INVALID_ARGVAL。

- NEW_LEN < 0
- NEW_LEN > LOBMAXSIZE

例

```
PROCEDURE Example_15 IS
    lob_loc BLOB;
BEGIN
-- get the LOB locator
SELECT b_col INTO lob_loc
        FROM lob_table
        WHERE key_value = 42 FOR UPDATE;
    DBMS_LOB.TRIM(lob_loc, 4000);
    COMMIT;
END;
```

関連項目： DBMS_LOB.ERASE()

DBMS_LOB.WRITE() プロシージャ

WRITE() プロシージャをコールして、LOB の先頭からの絶対 offset から開始し、指定した amount のデータを内部 LOB に書き込めます。データは、buffer パラメータから書き込まれます。

WRITE() では、LOB のオフセット位置にすでに存在するデータが、指定した長さだけ置換（上書き）されます。

入力量がバッファ内のデータ量を超えると、エラーとなります。入力量がバッファ内のデータ量よりも少なければ、バッファからのバイト数または文字数だけが LOB 内に書き込まれます。指定したオフセットが LOB 内に現在あるデータの終わりを超えると、BLOB または CLOB に対して、0（ゼロ）バイトの充てん文字または空白が、それぞれ挿入されます。

構文

```
PROCEDURE WRITE (
  lob_loc  IN OUT  BLOB,
  amount   IN      BINARY_INTEGER,
  offset   IN      INTEGER,
  buffer   IN      RAW);

PROCEDURE WRITE (
  lob_loc  IN OUT  CLOB   CHARACTER SET ANY_CS,
  amount   IN      BINARY_INTEGER,
  offset   IN      INTEGER,
  buffer   IN      VARCHAR2 CHARACTER SET lob_loc%CHARSET);
```

パラメータ

表 6–20　WRITE パラメータ

パラメータ名	意味
lob_loc	書き込む先の内部 LOB のロケータ
amount	これから書き込む、あるいはすでに書き込まれたバイト数または文字数
offset	書込み操作のために、LOB の先頭（origin: 1）からのオフセットをバイト数（BLOB の場合）または文字数（CLOB の場合）で表したもの
buffer	書込み用の入力バッファ

戻り値

なし。

プラグマ

なし。

例外

- VALUE_ERROR
  - LOB_LOC または AMOUNT、OFFSET のいずれかのパラメータが NULL または範囲外、無効の場合。

- INVALID_ARGVAL
  - AMOUNT < 1
  - AMOUNT > MAXBUFSIZE
  - OFFSET < 1
  - OFFSET > LOBMAXSIZE

例

```
PROCEDURE Example_16 IS
    lob_loc        BLOB;
    buffer         RAW;
    amt            BINARY_INTEGER := 32767;
    pos            INTEGER := 2147483647;
    i              INTEGER;
BEGIN
    SELECT b_col INTO lob_loc
        FROM lob_table
        WHERE key_value = 12;
    FOR i IN 1..3 LOOP
        DBMS_LOB.WRITE (lob_loc, amt, pos, buffer);
        /* さらにデータを充てん */
        pos := pos + amt;
    END LOOP;
    EXCEPTION4
        WHEN some_exception
        THEN handle_exception;
END;
```

関連項目： DBMS_LOB.APPEND()、DBMS_LOB.COPY()

LOB 制約

LOB の使用には、いくつかの制約があります。

- LOB は表に格納しなければならない -- 一時的状態（transient/temporary）にしておくことはできません。
- LONG データ型から LOB データ型への変換および移行、または LOB データ型から LONG データ型への変換および移行を行ってはならない。

  そのかわりに、次のようにすることができます。

  1. LONG RAW のデータをサーバー側のファイルに書き込む。
  2. Oracle8 のコマンド CREATE DIRECTORY を使用して、ファイルが書き込まれたディレクトリを参照する。
  3. Oracle8 のコマンド OCILobLoadFromFile() または DBMS_LOB.LOADFROMFILE() を使用して、LOB にファイル内のデータを移入する。

  LONG があまり大きくない場合には、LONG をバッファに読み込んでから OCILobWrite または DBMS_LOB.WRITE() をコールし、LONG データを LOB に書き込むという方法もあります。

  いずれの場合も、元の表に LOB 列を 1 列追加するか、または LOB 列を含む新しい表を作成する必要があります。Oracle8 では、列のデータ型を LOB 型に変更することはできません。
- 分散 LOB はサポートされていない。特に、ユーザーは、SELECT 句および WHERE 句のリモート・ロケータを使用することはできません。これには DBMS_LOB パッケージ関数の使用も含まれます。さらに、LOB 属性の有無にかかわらず、リモート表のオブジェクトへの参照はできません。

  たとえば、次の操作は無効となります。

  – SELECT lobcol from table1@remote_site;
  – INSERT INTO lobtable select type1.lobattr from table1@remote_site;
  – SELECT dbms_lob.length(lobcol) from table1@remote_site;

  リモート表の LOB 列での有効な操作には、次のものがあります。

  – CREATE TABLE as select * from table1@remote_site;
  – INSERT INTO t select * from table1@remote_site;
  – UPDATE t set lobcol = (select lobcol from table1@remote_site);
  – INSERT INTO table1@remote...
  – UPDATE table1@remote...
  – DELETE table1@remote...

- LOB にはローダー（ダイレクト・パスまたは従来型パス）サポートがない。そのかわりに、OCILobLoadFromFile() または DBMS_LOB.LOADFROMFILE()、ストリーム付き OCILobWrite() を使用してください。
- ピース単位の INSERT/UPDATE を使用するために内部 LOB をバインディングするとき、バインド変数は SQLT_CHR 型 と SQLT_LBI 型のどちらでもよいが、4k までに限られる。SQLT_LNG を LOB または 4k よりも長い SQLT_LBI にバインドすることはできません。

また、LOB を次の場所で使用することはできません。

- LOB は、パーティション表やクラスタ化した表には格納できない。したがって、LOB をクラスタ・キーにすることはできません。
- LOB は、GROUP BY および ORDER BY、SELECT DISTINCT、集約操作、JOINS には使用できない。ただし、LOB を持つ表で UNION ALL を使用することはできます。UNION および MINUS、SELECT DISTINCT は、オブジェクト型が MAP 機能または ORDER 機能を持っている場合は、LOB 属性で使用できます。
- LOB を ANALYZE... COMPUTE/ESTIMATE STATISTICS 文で分析することはできない。
- LOB は、索引のみの表に格納することはできない。
- LOB は、VARRAY では使用できない。
- NCLOB は、オブジェクト型の属性としては使用できないが、NCLOB パラメータをメソッドで使用することはできる。
- LOB ではトリガーはサポートされない。ただし、LOB を次のトリガーの中に使用することはできます。
  - どの種類のトリガーでも、LOB(:old または :new の値) に書き込むことはできない。
  - 通常のトリガーでは、:old の値の読込みはできるが、:new の値の読込みはできない。
  - INSTEAD OF トリガーでは、:old の値と :new の値は読み込むことができるが、書込みはできない。
  - トリガーが定義されている基礎表を更新せずに BFILE 型を更新できるので、LOB 型の列を OF 句に指定できない。
  - OCI 関数または DBMS_LOB パッケージを使用して、オブジェクト列の LOB 値または LOB 属性を更新すると、列または属性を含む表に定義されたトリガーは起動されない。
- クライアント側の PL/SQL プロシージャは DBMS_LOB パッケージ・ルーチンをコールできない。ただし、サーバー側の PL/SQL または PRO*C のアノニマス・ブロックを使用して、DBMS_LOB パッケージ・ルーチンをコールすることはできます。

7

ユーザー定義データ型 – 拡張例

この章では、ユーザー定義型の使用方法の応用例を説明します。この章には、大きく分けて次の 2 つのセクションがあります。

- 概要
- 発注書の例

概要

ユーザー定義型とは、アプリケーション内のデータ構造および操作を形式化するための、スキーマ・オブジェクトのことです。

> **関連項目：** ユーザー定義型の説明と使用方法は、『Oracle8 Server 概要』を参照してください。

この章で示す例では、ユーザー定義型の定義と使用における重要な特徴を解説します。object type メソッドの定義には、PL/SQL 言語を使用します。それ以外では、Oracle SQL を使用します。

> **関連項目：** SQL 構文と使用方法の詳細は、『Oracle8 Server SQL リファレンス』を参照してください。

PL/SQL を使うと、特にコレクションの要素へのアクセスおよびその操作の面で、ここに示す以上の機能が実現できます。

> **関連項目：** PL/SQL 機能の詳細は、『PL/SQL ユーザーズ・ガイド・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）を使うクライアント・アプリケーションでは、その広範な機能により、クライアント側でオブジェクトやコレクションにアクセスしたり、それらを操作したりすることができます。

> **関連項目：** この機能の詳細は、『Oracle 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。

発注書の例

この例は、顧客からの発注のデータ管理という、簡単な業務活動に基づいています。例は、3 部に分かれています。最初の 2 部はこの章にあり、第 3 部は、第 8 章の「オブジェクト・ビュー ― 拡張例」にあります。

それぞれの部では、スキーマをインプリメントして、基本アクティビティをサポートします。第 1 部では、Oracle のビルトイン・データ型だけを使ってスキーマをインプリメントします。これを、「リレーショナル・アプローチ」といいます。この方法を使用して、アプリケーションのデータを保持する表を作成し、アプリケーションの E-R を従来の方法で構築します。

第 2 部と第 3 部ではユーザー定義型 (UDT) を使い、エンティティと関連を、データベース管理システムで操作可能なスキーマ・オブジェクトに直接変換します。これを、「オブジェクト・リレーショナル・アプローチ」といいます。第 2 部と第 3 部では、同一の UDT を使います。基本的には、データを格納させる方式だけが異なります。

- 例の第 2 部では、オブジェクト表を作成してデータを格納し、第 1 部の表は使用しません。

- 第 3 部では、第 1 部で作成するリレーショナル表を使います。オブジェクト表は作成せずに、オブジェクト・ビューを使って仮想オブジェクト表を実現します。

エンティティおよび関連

この例の基本エンティティは、次のとおりです。

- 顧客
- 販売する製品の在庫
- 発注書

1 人の顧客が複数の発注を行うことはできますが、ある 1 つの発注書を発行できる顧客は 1 人だけなので、顧客と発注書の関連は 1 対多になります。

1 つの発注書に複数の在庫項目が入れることができ、同じ在庫項目を複数の発注書に組み込むこともできるので、発注書と在庫項目との関連は多対多です。

発注書と在庫との多対多の関連を管理するには、通常、明細品目という別のエンティティを導入します。発注書には任意の数の明細品目を組み込めますが、明細品目はそれぞれ 1 つの発注書に対応します。同じ在庫項目を複数の明細品目に入れることはできますが、各明細品目にはそれぞれ 1 つの在庫項目しか入れられません。

表 7-1 は、顧客の注文のニーズを管理するアプリケーションに必要な各エンティティの情報のリストです。

表 7-1　発注書例のエンティティに関する必須情報

エンティティ	必須情報
顧客	連絡先
在庫	項目識別、コスト、課税コード
発注書	顧客、発注日、出荷日、出荷先住所
明細品目	各明細品目の在庫項目、数量、価格、割引き

実社会の属性エンティティの複雑さが問題であり、データ構造をマップするには複雑な属性のセットが必要になります。住所は通常、郵便番号、都道府県名、市町村名などの属性を持ちます。1 人の顧客が複数の電話番号を持っている場合もあります。明細品目はそれ自体がエンティティであり、また、発注書の属性でもあります。標準ビルトイン型では、それらを直接表すことはできません。オブジェクトリレーショナル・アプローチでは、様々な方法でこの複雑な構造体を処理できます。

第 1 部　リレーショナル・アプローチ

リレーショナル・アプローチでは、エンティティおよびその他の属性を正規化します。顧客および発注書、在庫エンティティを表にします。住所は、標準的な住所の属性を作って列にします。顧客の電話番号の数を任意に制限し、番号ごとに列を割り当てます。

リレーショナル・アプローチでは、発注書と明細品目を別にし、それぞれの表に組み込みます。明細品目表には、在庫表および発注書表への外部キーの列を設定します。

表の作成

リレーショナル・アプローチの表は、次のようになります。

```
CREATE TABLE customer_info (
  custno       NUMBER,
  custname     VARCHAR2(200),
  street       VARCHAR2(200),
  city         VARCHAR2(200),
  state        CHAR(2),
  zip          VARCHAR2(20),
  phone1       VARCHAR2(20),
  phone2       VARCHAR2(20),
  phone3       VARCHAR2(20),
  PRIMARY KEY (custno)
  ) ;

CREATE TABLE purchase_order (
  pono         NUMBER,
  custno       NUMBER REFERENCES customer_info,
  orderdate    DATE,
  shiptodate   DATE,
  shiptostreet VARCHAR2(200),
  shiptocity   VARCHAR2(200),
  shiptostate  CHAR(2),
  shiptozip    VARCHAR2(20),
  PRIMARY KEY (pono)
  ) ;

CREATE TABLE stock_info (
  stockno      NUMBER PRIMARY KEY,
  cost         NUMBER,
  tax_code     NUMBER
  ) ;

CREATE TABLE line_items (
  lineitemno   NUMBER,
  pono         NUMBER REFERENCES purchase_order,
  stockno      NUMBER REFERENCES stock_info,
```

```
  quantity     NUMBER,
  discount     NUMBER,
  PRIMARY KEY (pono, lineitemno)
  ) ;
```

最初の表、CUSTOMER_INFO には、顧客に関する情報を格納します。この表は他の表を参照しませんが、PURCHASE_ORDER 表には CUSTNO 列があり、CUSTOMER_INFO 表への外部キーに設定しています。

この外部キーは、発注書と顧客との、複数対 1 の関連を実現します。1 人の顧客から複数の発注書が発行されることはありますが、複数の顧客がある 1 つの発注書を発行することはありません。

LINE_ITEMS 表には、PURCHASE_ORDER 表への外部キー PONO と、STOCK_INFO 表への外部キー STOCKNO を定義します。

値の挿入

上で定義した表を基盤とするアプリケーションでは、次のような命令を発行して表にデータを挿入します。

```
INSERT INTO customer_info
  VALUES (1, 'Jean Nance', '2 Avocet Drive',
         'Redwood Shores', 'CA', '95054',
         '415-555-1212', NULL, NULL) ;

INSERT INTO customer_info
  VALUES (2, 'John Nike', '323 College Drive',
         'Edison', 'NJ', '08820',
         '609-555-1212', '201-555-1212', NULL) ;

INSERT INTO purchase_order
  VALUES (1001, 1, SYSDATE, '10-MAY-1997',
          NULL, NULL, NULL, NULL) ;

INSERT INTO purchase_order
  VALUES (2001, 2, SYSDATE, '20-MAY-1997',
         '55 Madison Ave', 'Madison', 'WI', '53715') ;

INSERT INTO stock_info VALUES(1004, 6750.00, 2) ;
INSERT INTO stock_info VALUES(1011, 4500.23, 2) ;
INSERT INTO stock_info VALUES(1534, 2234.00, 2) ;
INSERT INTO stock_info VALUES(1535, 3456.23, 2) ;

INSERT INTO line_items VALUES(01, 1001, 1534, 12, 0) ;
INSERT INTO line_items VALUES(02, 1001, 1535, 10, 10) ;
INSERT INTO line_items VALUES(10, 2001, 1004, 1, 0) ;
INSERT INTO line_items VALUES(11, 2001, 1011, 2, 1) ;
```

データの検索

データの検索

値が通常の方法でこれらの表に挿入されたとすると、アプリケーションでは、次の種類の問合せを実行して、格納されたデータから必要な情報を検索します。

発注書 1001 の顧客および明細品目データ

```
SELECT   C.custno, C.custname, C.street, C.city, C.state,
         C.zip, C.phone1, C.phone2, C.phone3,

         P.pono, P.orderdate,

         L.stockno, L.lineitemno, L.quantity, L.discount

 FROM    customer_info  C,
         purchase_order P,
         line_items     L

 WHERE   C.custno = P.custno
  AND    P.pono = L.pono
  AND    P.pono = 1001;
```

各発注書の合計金額

```
SELECT   P.pono, SUM(S.cost * L.quantity)

 FROM    purchase_order P,
         line_items     L,
         stock_info     S

 WHERE   P.pono = L.pono
  AND    L.stockno = S.stockno

 GROUP BY P.pono;
```

在庫項目 1004 の発注書と明細品目

```
SELECT  P.pono, P.custno,
        L.stockno, L.lineitemno, L.quantity, L.discount

 FROM   purchase_order P,
        line_items     L

 WHERE    P.pono = L.pono
```

```
  AND    L.stockno = 1004;
```

データの更新

上で説明したスキーマ・オブジェクトでは、次のような命令でデータを更新します。

発注書が 01 で在庫項目が 1001 の数量の更新

```
UPDATE  line_items

 SET    quantity = 20

 WHERE  pono     = 1
  AND   stockno  = 1001 ;
```

データの削除

最初に定義した表を基盤とするアプリケーションでは、次のような命令が発行され格納されたデータが削除されます。

発注書 1001 の削除

```
DELETE
 FROM   line_items
 WHERE  pono = 1001 ;

DELETE
 FROM   purchase_order
 WHERE  pono = 1001 ;
```

第 2 部　オブジェクト表を使用したオブジェクト・リレーショナル・アプローチ

別のアプローチが必要となる理由

C++ のような第 3 世代言語 (3GL) で記述されたアプリケーションでは、データをメソッドでカプセル化する、高度で複雑なユーザー定義の型をインプリメントできます。これに対して、SQL では、基本的なスカラー型だけが提供され、関連する操作でこれらをカプセル化する方法はありません。

3GL を使用してアプリケーションを作成しない理由はいくつかあります。まず、DBMS では、レプリケートするのに何百万人 / 時間かかるような機能を提供しています。第 2 に、3GL を使用した情報の管理の問題の 1 つは、永続的ではない点です。たとえ、永続的であったとしても、同一アドレス領域にアプリケーション・ロジックとデータ・ロジックを配置することで必要なパフォーマンスを確保しているため、セキュリティが低下します。このトレードオフは、永続性とセキュリティが基本要件である DBMS のユーザーには受け入れられません。

この結果、アプリケーションの開発者には、SQL にマップするいくつかのフォームで複雑な型をシミュレーションするという問題が残ります。多くの人・時間が必要なことは別として、このことは、実現についての重要な問題に関連しています。次のことを順次実行する必要があります。

- 「書込み」についてアプリケーション・ロジックからデータ・ロジックに変換する。
- 「読込み」について逆のプロセスを実行する (その逆もあり)

明らかに、クライアントとサーバーのアドレス領域の間には、大量の通信量があります。また、クライアントとサーバーが異なるマシンにある場合は、ネットワークの往復のパフォーマンスについての通信料を考慮する必要があります。

O-R テクノロジは、これらの問題を解決します。この章以降では、この新しい機能をインプリメントする例について設定します。

オブジェクトリレーショナル（O-R）の方法

ここまで出てきた例の O-R 的アプローチでは、7-3 ページの 「エンティティおよび関連」で概説したものと同じ E-R から始まります。しかし、このアプローチでは、ユーザー定義型を使うことによって、その構造体をより多くデータベース・スキーマに取り入れることができます。

O-R アプローチを使うと、顧客の住所や連絡用電話番号を、リレーショナル表の無関連な列に分けて入れるのではなく、型を定義して表します。また、明細品目を切り離して別の表に入れるのではなく、それぞれの発注書と一緒にして、ネストした表を作ります。

O-R アプローチでは、メイン・エンティティである顧客および在庫、発注書が、オブジェクトになります。オブジェクト参照によって、n:1 の関連を表します。コレクション型を使って、多値属性をモデル化しています。

O-R モデルには、次の 2 つの具現化される手法があります。

- オブジェクト表を作成してデータを移入する
- オブジェクト・ビューを使って既存のリレーショナル・データから仮想オブジェクト表を実現する

この章の残りでは、O-R スキーマを解説し、それをオブジェクト表で実現する方法を示します。第 8 章の「オブジェクト・ビュー ― 拡張例」では、同じスキーマをオブジェクト・ビューで実現します。

型の定義

次の文で、アプリケーションの土台を作ります。

```
CREATE TYPE line_item_t ;
CREATE TYPE purchase_order_t ;
CREATE TYPE stock_info_t ;
```

この 3 つの文では、不完全なオブジェクト型が定義されます。不完全な定義をすると、Oracle では、完全な定義は後でされると解釈されます。このような不完全な定義のおかげで、これらの型を参照する型は正常にコンパイルされます。C などのプログラム言語で言うところのプロトタイプのようなものです。

次の文で、配列型を定義します。

```
CREATE TYPE phone_list_t AS VARRAY(10) OF VARCHAR2(20) ;
```

上記の文では、PHONE_LIST_T 型が定義されます。型 PHONE_LIST_T のデータ単位は、すべて最大 10 の電話番号の VARRAY であり、各番号は型 VARCHAR2 のデータ項目で表されています。

電話番号のリストには、VARRAY またはネストした表を使えます。この場合、リストは 1 人の顧客の連絡用電話番号の集まりです。次に挙げる理由から、ネストした表よりも VARRAY の方をお薦めします。

- 番号の順序が重要な場合もあります。VARRAY は順序付けられていますが、ネストした表は順序付けられていません。
- 特定の顧客の電話番号の数は多くありません。VARRAY では、要素の最大数（この場合は 10）を事前に指定する必要があります。そのため、サイズ制限のないネストした表よりも、記憶域を効率よく使えます。
- 電話番号リストに対して問合せを行うことはないので、ネストした表の形式は必要ありません。

順序付けと記憶域制限が設計上重要な問題ではないときは、原則として、次の経験則に従って VARRAY かネストした表かを決めることができます。コレクションに問合せをする必要がある場合はネストした表を使い、コレクションの全体を 1 つとして取り扱う場合は VARRAY を使います。

```
CREATE TYPE address_t AS OBJECT (
  street  VARCHAR2(200),
  city    VARCHAR2(200),
  state   CHAR(2),
  zip     VARCHAR2(20)
  ) ;
```

上記の文で、オブジェクト型 ADDRESS_T が定義されます。この型のデータ単位が住所を表します。属性はすべて文字列で、住所をやや単純化して最小限の部分に分けたものです。

次の文で、他のユーザー定義型も使ってオブジェクト型を定義します。このオブジェクト型には、比較メソッドも定義しています。

```
CREATE TYPE customer_info_t AS OBJECT (
  custno     NUMBER,
  custname   VARCHAR2(200),
  address    address_t,
  phone_list phone_list_t,
```

```
  ORDER MEMBER FUNCTION
    cust_order(x IN customer_info_t) RETURN INTEGER,

  PRAGMA RESTRICT_REFERENCES (
    cust_order, WNDS, WNPS, RNPS, RNDS)
  ) ;
```

上記の文で、オブジェクト型 CUSTOMER_INFO_T が定義されます。この型のデータ単位は、特定の顧客に関する各情報を表すオブジェクトになります。CUSTOMER_INFO_T オブジェクトの属性は、数値および文字列、ADDRESS_TO オブジェクト型、PHONE_LIST_T 型の VARRAY です。

CUSTOMER_INFO_T オブジェクトには、2 種類の比較メソッドの 1 つ、順序メソッドが対応付けられています。Oracle では、2 つの CUSTOMER_INFO_T オブジェクトの比較が必要になると、CUST_ORDER メソッドが起動されます。

2 種類の比較メソッドとは、マップ・メソッドと順序メソッドです。このアプリケーションでは、参考のため、両方のメソッドをそれぞれ使用します。

注意： 順序メソッドは、比較する 2 つのオブジェクトの両方にコールする必要があるのに対して、マップ・メソッドは各オブジェクトにつき 1 回コールします。一般的にオブジェクトのセットをソートするとき、順序メソッドのコール回数は、マップ・メソッドより多くなります。システムは、スカラー値の比較を非常に効率的に実行できることと、ユーザー定義関数のコールが、カーネル実行関数をコールするよりも遅いという事実を考慮すると、順序メソッドを使用するオブジェクトのソートは、マップされたスカラー値 (MAP 関数によりに戻される) をソートするよりも遅くなります。

関連項目：

- 順序メソッドとマップ・メソッドの説明は、『Oracle8 Server 概要』を参照してください。
- プラグマ宣言の使用方法の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

上記の文には、メソッド CUST_ORDER の本体である実際の PL/SQL プログラムは含まれていません。これについては、後の項で示します。

次の文は、この項の始めに宣言した不完全なオブジェクト型 LINE_ITEM_T の定義を完成するものです。

```
CREATE TYPE line_item_t AS OBJECT (
  lineitemno NUMBER,
  STOCKREF   REF stock_info_t,
```

```
  quantity   NUMBER,
  discount   NUMBER
  ) ;
```

LINE_ITEM_T 型のデータ単位は、明細品目を表すオブジェクトです。数値属性が 3 つと REF 属性が 1 つによって構成されます。明細品目エンティティをモデル化しており、対応する在庫オブジェクトへのオブジェクト参照が含まれます。

```
CREATE TYPE line_item_list_t AS TABLE OF line_item_t ;
```

上記の文で、表型 LINE_ITEM_LIST_T が定義されます。この型のデータ単位はネストした表で、各行は LINE_ITEM_T オブジェクトになります。発注書の明細品目の配列を表すには、次の理由から、LINE_ITEM_T オブジェクトの VARRAY よりも明細品目のネストした表をお薦めします。

- 明細品目の内容の問合せは、ほとんどのアプリケーションで必要条件となっていると考えられます。VARRAY ではまず、ネストした表に VARRAY を変換しなければならないので、操作の効率が悪くなります。
- アプリケーションによっては、明細品目データの索引付けが必要条件になっている場合もあります。これは、ネストした表では可能ですが、VARRAY ではできません。
- 明細品目の順序は、通常は重要ではなく、必要なときには明細品目番号を使って順序を指定できます。
- 発注書の明細品目の数には、実質的な上限はありません。VARRAY を使うと、要素の数の上限を指定しなければなりません。

次の文は、この項の始めに宣言した不完全なオブジェクト型 PURCHASE_ORDER_T の定義を完成するものです。

```
CREATE TYPE purchase_order_t AS OBJECT (
  pono           NUMBER,
  custref        REF customer_info_t,
  orderdate      DATE,
  shipdate       DATE,
  line_item_list line_item_list_t,
  shiptoaddr     address_t,

  MAP MEMBER FUNCTION
    ret_value RETURN NUMBER,
    PRAGMA RESTRICT_REFERENCES (
      ret_value, WNDS, WNPS, RNPS, RNDS),

  MEMBER FUNCTION
    total_value RETURN NUMBER,
    PRAGMA RESTRICT_REFERENCES (total_value, WNDS, WNPS)
  ) ;
```

上記の文で、オブジェクト型 PURCHASE_ORDER_T が定義されます。この型のデータ単位は発注書を表すオブジェクトです。これには、REF、および型 LINE_ITEM_LIST_T のネストした表、ADDRESS_T オブジェクトを含む、6 つの属性があります。

PURCHASE_ORDER_T 型のオブジェクトは、RE_VALUE と TOTAL_VALUE の 2 つのメソッドが使用できます。1 つはマップ・メソッドで、2 種類ある比較メソッドの 1 つです。マップ・メソッドは、オブジェクト内でレコードの順序内での指定レコードの相対的な位置を戻します。Oracle では、2 つの PURCHASE_ORDER_T オブジェクトの比較が必要になると、RET_VALUE メソッドが暗黙的にコールされます。

2 つのプラグマ宣言で、2 つのメソッドがデータベースに対してどのようなアクセス制約を受けるかが、PL/SQL に通知されます。

> **関連項目：** プラグマ宣言の使用方法の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

RET_VALUE メソッドおよび TOTAL_VALUE メソッドの本体である実際の PL/SQL プログラムは含まれていません。これについては、後の項で示します。

次の文は、この項の始めに宣言した 3 つの不完全なオブジェクト型の最後の 1 つ、STOCK_INFO_T の定義を完成するものです。

```
CREATE TYPE stock_info_t AS OBJECT (
  stockno    NUMBER,
  cost       NUMBER,
  tax_code   NUMBER
  ) ;
```

STOCK_INFO_T は、顧客が発注する在庫項目を表すオブジェクトです。これには、3 つの数値属性があります。

メソッドの定義

この項では、CUSTOMER_INFO_T および PURCHASE_ORDER_T オブジェクト型のメソッドを指定する方法を示します。

```
CREATE OR REPLACE TYPE BODY purchase_order_t AS
  MEMBER FUNCTION total_value RETURN NUMBER IS
    i          INTEGER;
    stock      stock_info_t;
    line_item  line_item_t;
    total      NUMBER := 0;
    cost       NUMBER;

  BEGIN
    FOR i IN 1..SELF.line_item_list.COUNT LOOP

      line_item := SELF.line_item_list(i);
      SELECT DEREF(line_item.stockref) INTO stock FROM DUAL ;
```

```
      total := total + line_item.quantity * stock.cost ;

      END LOOP;
    RETURN total;
  END;

  MAP MEMBER FUNCTION ret_value RETURN NUMBER IS
  BEGIN
    RETURN pono;
  END;
END;
```

上記の文で、PURCHASE_ORDER_T オブジェクト型の本体、つまり、そのメソッドをインプリメントする PL/SQL プログラムが定義されます。

RET_VALUE メソッド

RET_VALUE メソッドは、単純です。これは単に対応付けられた PURCHASE_ORDER_T オブジェクト番号を戻すだけです。

TOTAL_VALUE メソッド

TOTAL_VALUE メソッドは、複数の O-R 手段を使用して、それに対応付けられた PURCHASE_ORDER_T オブジェクトの明細品目の請求額の合計を戻します。

- 前述のとおり、TOTAL_VALUE メソッドの基本的な機能は、それに対応付けられた PURCHASE_ORDER_T オブジェクトの明細品目の請求額の合計を戻すことです。キーワード SELF は、あらゆる関数のパラメータとして暗黙的に作成され、そのオブジェクトを参照します。
- キーワード COUNT は、PL/SQL 表または配列にある要素の総数を表します。ここで、ループを組み合わせると、アプリケーションは、コレクションのすべての要素（この場合、発注書の項目）で処理を繰り返します。このように、SELF.LINE_ITEM_LIST.COUNT は、ネストした表内の要素の数を表し、これは、SELF で表される PURCHASE_ORDER_T オブジェクトの LINE_ITEM_LIST 属性と一致します。
- DEREF 演算子は、引数として参照値を受け入れ、行オブジェクトを戻します。この場合、DEREF(LINE_ITEM.STOCKREF) は、引数として STOCKREF 属性を受け入れ、STOCK_INFO_T オブジェクトを戻します。データ定義に戻ると、STOCKREF が、それ自体 LINE_ITEM_LIST の要素である LINE_ITEM_T オブジェクトの属性であることがわかります。このリスト・オブジェクトは、ネストした表として構成され、SELF で表される PURCHASE_ORDER_T オブジェクトの属性になります。このことは、発注書 (PURCHASE_ORDER_T) に品目 (LINE_ITEM_T) のリスト (LINE_ITEM_LIST) が含まれ、そのリストには品目 (STOCK_INFO_T) に関する情報への参照 (STOCKREF) が含まれるという、実社会の状況をもう一度思い出して考えないと複雑に見えます。これまで説明してきた操作では、OR 手段によって、必要なデータが簡単にフェッチされます。

- これらすべてのエンティティは、抽象的なデータ型で、PURCHASE_ORDER_T は、すべての発注書オブジェクトのテンプレートなので、オブジェクトのテンプレートとして参照できます。次に、実際の在庫オブジェクトの値の抽出方法を考えます。Oracle では、PL/SQL プログラム内の REF の暗黙的デリファレンスがサポートされていないので、SQL SELECT 文に明示的な DEREF コールが必要です。PL/SQL 変数 STOCK は、STOCK_INFO_T 型です。これは、SELECT 文によって、DEREF (LINE_ITEM.STOCKREF) で表されるオブジェクトに設定されます。このオブジェクトは、i 番目の明細品目を参照した実際の在庫項目です。
- 対象の在庫項目を抽出したら、次のステップはそのコストの計算方法です。このプログラムでは、STOCK.COST、STOCK オブジェクトの COST 属性として、在庫項目のコストが参照されます。しかし、項目のコストを計算するには、品目の発注量を把握する必要があります。LINE_ITEM.QUANTITY は、各 LINE_ITEM_T オブジェクトの QUANTITY 属性を表します。

メソッドの残りのソースコードは簡単です。繰り返しにより明細品目の請求額を合計し、メソッドはその合計を値として戻します。

CUST_ORDER メソッド

次の文で、CUSTOMER_INFO_T オブジェクト型の CUST_ORDER メソッドを定義します。

```
CREATE OR REPLACE TYPE BODY customer_info_t AS
  ORDER MEMBER FUNCTION
  cust_order (x IN customer_info_t) RETURN INTEGER IS
  BEGIN
    RETURN custno - x.custno;
  END;
END;
```

最初に述べたとおり、CUST_ORDER 操作の機能は、2 つの顧客の発注についての情報を比較することです。その操作のメカニズムは非常に簡単です。順序メソッド CUST_ORDER は、別の CUSTOMER_INFO_T オブジェクトを入力の引数として受け取り、2 つの CUSTNO 数値の差を戻します。他の CUSTOMER_INFO_T オブジェクトの CUSTNO が、内部にあるオブジェクトの CUSTNO から引かれます。内部にあるオブジェクトの CUSTNO 値の方が小さければ負の数、大きければ正の数が戻され、2 つのオブジェクト CUSTNO 値が同等なら 0(ゼロ) が戻されます。どの場合も、自己参照されます。CUSTNO に実社会における方法を使用するなら（たとえば、小さい番号が大きい番号よりも早く作成される）、この関数によって戻される実際の値を利用できます。

以上が、発注書アプリケーションで使われるユーザー定義型の定義です。ここまでの宣言では、表を作成したり、データ記憶領域を予約したりしません。

オブジェクト表の作成

ここまでは、オブジェクト表を作成してデータを移入する場合も、例の最初の部分で示したリレーショナル表とオブジェクト・ビューを使ってアプリケーションをインプリメントする場合も、例は同じです。この章の残りの部分では、引き続きオブジェクト表を使った例を紹

介します。第 8 章の「オブジェクト・ビュー ― 拡張例」では、この点を取り上げ、続けてオブジェクト・ビューの例を説明します。

一般的に、" オブジェクト " と " 表 " の関連は、次のように考えられます。

- 表へのクラス（エンティティを表す）のマップ
- 列への属性のマップ
- 行へのオブジェクトのマップ

この方法では、各表は、オブジェクト（特定の行）がそれぞれ同じ属性（列値）を持つ暗黙的な型です。抽象的なデータ型およびオブジェクト表を明示的に作成すると、機能性の新しいレベルが導入されます。

オブジェクト表 CUSTOMER_TAB

オブジェクト表の作成 : 基本構文 次の文で、CUSTOMER_INFO_T 型のオブジェクトを持つオブジェクト表 CUSTOMER_TAB を定義します。

```
CREATE TABLE customer_tab OF customer_info_t
 (custno PRIMARY KEY);
```

見てわかるように、オブジェクト表の定義に構文上の違いがあります。つまり、"OF" の使用方法です。次の、最初に定義した CUSTOMER_INFO_T オブジェクトの属性を思い出してください。

```
custno     NUMBER
custname   VARCHAR2(200)
address    address_t
phone_list phone_list_t
po_list    po_reflist_t
```

このことは、表 CUSTOMER_TAB に、CUSTNO および CUSTNAME、ADDRESS、PHONE_LIST、PO_LIST の列があり、各行には、オブジェクト型 CUSTOMER_INFO_T があることを意味します。また、この行オブジェクトの概念では、機能性が非常に向上することがわかります。

オブジェクト表のテンプレートとしての抽象的なデータ型 最初に、型 CUSTOMER_INFO_T があるということは、大量の型 CUSTOMER_INFO_T の表を作成できることに注意してください。たとえば、型 CUSTOMER_INFO_T の表 CUSTOMER_TAB2 も作成できます。しかし、この機能がないと、各表を個々に定義しなければなりません。

同じ型の表を作成できるということは、バリエーションがなくなるということではありません。CUSTOMER_TAB を作成した文によって、CUSTNO 列に主キー制約が定義されたことに注意してください。この制約は、この表だけに適用されます。CUSTOMER_INFO_T オブジェクトとは別のオブジェクト表（たとえば ,CUSTOMER_TAB2）は、この制約を受けません。これは、制約は型の定義にではなく、表に対して適用されるという、重要な概念を示しています。

埋込みオブジェクト付きのオブジェクト表 CUSTOMER_TAB の定義をよく見ると、ADDRESS 列に ADDRESS_T オブジェクトがあることがわかります。抽象的なデータ型には、抽象的なデータ型である属性がある場合があります。これらの型が、オブジェクトとしてインスタンシエートされるとそれに含まれるオブジェクトも同様にインスタンシエートされます（ただし、NULL 値を許可しない場合。この場合その値のプレースホルダーが作成されます。）ADDRESS_T オブジェクトには、CUSTOMER_INFO_T のリーフレベルのスカラー属性であることを示すビルトインの型の属性があります。Oracle では、ADDRESS_T オブジェクトとその属性の列が、オブジェクト表 CUSTOMER_TAB に作成されます。これらの列は、ドット表記法を使って参照できます。たとえば、ZIP 列に索引を作成すると、それを ADDRESS.ZIP として参照できます。

PHONE_LIST_T 列には、PHONE_LIST_T 型の VARRAY があります。PHONE_LIST_T 型の各オブジェクトは、すべて最大 10 の電話番号の VARRAY と定義され、各番号は VARCHAR2 型のデータ項目で表されていたことを思い出してください。

```
CREATE TYPE phone_list_t AS VARRAY(10) OF VARCHAR2(20) ;
```

PHONE_LIST_T 型の VARRAY には、最大 200 文字（10 x 20）と多少のオーバーヘッドしかありません。そのため、Oracle では、VARRAY は PHONE_LIST 列に 1 個のデータ単位として格納されます。Oracle では、4000 バイトを超える VARRAY は BLOB に格納されます。つまり、表外に格納されます。ここで、重要な問題に戻ります。つまり、これらの外部オブジェクトをデータベース管理システムでどのようにして参照するか、という問題です。

オブジェクト表 STOCK_TAB

次の文で、STOCK_INFO_T オブジェクト型のオブジェクト表を作成します。

```
CREATE TABLE stock_tab OF stock_info_t
 (stockno PRIMARY KEY) ;
```

ここでは、新しいものは何もありません。上記の文で、STOCK_TAB オブジェクト表が作成されます。表の各行は、STOCK_INFO_T オブジェクトで、それぞれ次の 3 つの数値属性があります。

```
  stockno    NUMBER,
  cost       NUMBER,
  tax_code   NUMBER
```

Oracle では、各属性に列が割り当てられ、CREATE TABLE 文により、STOCKNO 列に主キー制約が設定されます。

オブジェクト表 PURCHASE_TAB

次の文で、PURCHASE_ORDER_T オブジェクトのオブジェクト表を定義します。

```
CREATE TABLE purchase_tab OF purchase_order_t (
  PRIMARY KEY (pono),
  SCOPE FOR (custref) IS customer_tab
  )
  NESTED TABLE line_item_list STORE AS po_line_tab ;
```

上記の文で、PURCHASE_TAB オブジェクト表が作成されます。表の各行は、PURCHASE_ORDER_T オブジェクトです。PURCHASE_ORDER_T オブジェクトの属性は、次のとおりです。

```
  pono           NUMBER
  custref        REF customer_info_t
  orderdate      DATE
  shipdate       DATE
  line_item_list line_item_list_t
```

```
  shiptoaddr      address_t
```

REF 演算子 ここで紹介する第 1 の要素では、文によって、CUSTREF 列の REF に有効範囲を設定する方法を処理しなければなりません。有効範囲に制限がない場合（デフォルト）、REF 演算子によって、どの行オブジェクトでも参照できます。ただし、これらの CUSTREF の REF は、CUSTOMER_TAB オブジェクト表の行オブジェクトしか参照できません。有効範囲制限が適用されるのは、CUSTOMER_TAB オブジェクト表の CUSTREF 列だけです。これは、他のオブジェクト表で格納された PURCHASE_ORDER_T オブジェクトの CUSTREF 属性には適用されません。

ネストした表 第 2 の新しい要素は、各行にネストした表の列 LINE_ITEM_LIST があることです。最終行では、PO_LINE_TAB という表を作成して、PURCHASE_TAB 表のすべての行の LINE_ITEM_LIST 列を格納します。この発注書の例のように、ネストした表は特に、マスター / ディテールの関連のコーディングにうまく適応します。これから説明するように、ネストした表により、リレーショナル結合の複雑さをアプリケーションから取り除くことができます。

ネストした表のすべての行は、別々の記憶表に格納されます。記憶表には、NESTED_TABLE_ID という非表示列があり、対応する親行の行と一致します。特定の親に属するネストした表のすべての要素に、同一の NESTED_TABLE_ID 値があります。

たとえば、PURCHASE_TAB の、ある 1 行に列としてある、ネストした表の要素はすべて、NESTED_TABLE_ID の値が同じです。PURCHASE_TAB の別の行に属するネストした表の要素には、異なる NESTED_TABLE_ID の値があります。

ネストした表の型のトップ・レベルの属性は、記憶表の列にマップされます。オブジェクト型ではない要素を持つネストした表には、名前のない列が 1 列できます。Oracle では、キーワード COLUMN_VALUE が、その列の名前のかわりになります。たとえば、REF のネストした表の REF 列に有効範囲を設定するには、COLUMN_VALUE 列名を使用して参照できます。

Oracle では、PURCHASE_ORDER_T オブジェクトの残りのリーフレベルのスカラー属性（ORDERDATE、SHIPDATE、SHIPTOADDR の ADDRESS_T オブジェクトの属性）の列を CUSTOMER_TAB に作成します。

以上で、発注書アプリケーションのすべての表が設定されました。次の項では、これらの表にその他の機能を追加する方法を示します。

表の変更

次の文で、PURCHASE_TAB オブジェクト表のネストした表 LINE_ITEM_LIST の列を持つ PO_LINE_TAB 記憶表を変更して、内部にある REF に有効範囲を設定します。

```
ALTER TABLE po_line_tab
  ADD (SCOPE FOR (stockref) IS stock_tab) ;
```

PO_LINE_TAB 記憶表は、LINE_ITEM_LIST_T 型のネストした表の列を持ちます。この型の定義は（この章の最初に説明したように）次のとおりです。

```
CREATE TYPE line_item_list_t AS TABLE OF line_item_t ;
```

LINE_ITEM_T オブジェクトの属性であり、そのため PO_LINE_TAB 記憶域表の列でもある STOCKREF は、STOCK_INFO_T REF 型に属します。オブジェクト表 STOCK_TAB は、STOCK_INFO_T 型の行オブジェクトを持ちます。ALTER 文で STOCKREF 列の REF の有効範囲がオブジェクト表 STOCK_TAB に制限されます。

次の文で、PO_LINE_TAB 記憶表をさらに変更して、索引用記憶域を指定します。

```
ALTER TABLE po_line_tab
  STORAGE (NEXT 5K PCTINCREASE 5 MINEXTENTS 1 MAXEXTENTS 20) ;
```

次の文で、PO_LINE_TAB 記憶表に索引を作成します。

```
CREATE INDEX po_nested_in
  ON         po_line_tab (NESTED_TABLE_ID) ;
```

オブジェクト表のネストした表の列の記憶表には、NESTED_TABLE_ID という隠された列があります。上記の文で、その列に索引が作成されるため、PURCHASE_TAB オブジェクト表の LINE_ITEM_LIST 列の内容へのアクセス効率が高まります。

次の文では、NESTED_TABLE_ID を使って囲み表の各行内のネストした表列を一意にする方法を示します。これにより、PO_LINE_TAB 記憶域表に一意の索引が作成されます。表には、PURCHASE_TAB 表のすべての行の LINE_ITEM_LIST 列が保持されます。

```
CREATE UNIQUE INDEX po_nested
  ON                po_line_tab (NESTED_TABLE_ID, lineitemno) ;
```

索引キーに、LINEITEMNO 列を組み込み、UNIQUE を指定することによって、各発注書の LINEITEMNO 列の値は、確実に一意になります。

値の挿入

この項では、作成したオブジェクト表に 7-1 ページでリレーショナル表に挿入したのと同じデータを挿入する方法を示します。

stock_tab

```
INSERT INTO stock_tab VALUES(1004, 6750.00, 2);
INSERT INTO stock_tab VALUES(1011, 4500.23, 2);
INSERT INTO stock_tab VALUES(1534, 2234.00, 2);
INSERT INTO stock_tab VALUES(1535, 3456.23, 2);
```

customer_tab

```
INSERT INTO customer_tab
  VALUES (
```

```
    1, 'Jean Nance',
    address_t('2 Avocet Drive', 'Redwood Shores', 'CA', '95054'),
    phone_list_t('415-555-1212')
    ) ;

INSERT INTO customer_tab
  VALUES (
    2, 'John Nike',
    address_t('323 College Drive', 'Edison', 'NJ', '08820'),
    phone_list_t('609-555-1212','201-555-1212')
    ) ;
```

purchase_tab

```
INSERT INTO purchase_tab
  SELECT  1001, REF(C),
          SYSDATE,'10-MAY-1997',
          line_item_list_t(),
          NULL
   FROM   customer_tab C
   WHERE  C.custno = 1 ;
```

上記の文で、次の属性を持つ PURCHASE_ORDER_T オブジェクトが構成されます。

```
  pono            1001
  custref         REF to customer number 1
  orderdate       SYSDATE
  shipdate        10-MAY-1997
  line_item_list  an empty line_item_list_t
  shiptoaddr      NULL
```

上記の文では、問合せを使って、CUSTNO 値が 1 である CUSTOMER_TAB オブジェクト表の行オブジェクトに対する REF を構成します。

次の INSERT 文では、キーワード THE をつけてフラット化された副問合せを使って、挿入のターゲット、つまり、PONO 値が 1001 である PURCHASE_TAB オブジェクト表の行オブジェクトの LINE_ITEM_LIST 列にあるネストした表を特定します。

```
INSERT INTO THE (
  SELECT  P.line_item_list
   FROM   purchase_tab P
   WHERE  P.pono = 1001
  )
  SELECT  01, REF(S), 12, 0
   FROM   stock_tab S
   WHERE  S.stockno = 1534;
```

上記の文で、フラット化された副問合せで識別されたネストした表に、明細品目が挿入されます。挿入される明細品目には、STOCKNO 値が 1534 である STOCK_TAB オブジェクト表の行オブジェクトに対する REF があります。

次の文は、前述の 2 つの文と似ています。

```
INSERT INTO purchase_tab
  SELECT  2001, REF(C),
          SYSDATE,'20-MAY-1997',
          line_item_list_t(),
          address_t('55 Madison Ave','Madison','WI','53715')
   FROM   customer_tab C
   WHERE  C.custno = 2;

INSERT INTO THE (
  SELECT  P.line_item_list
   FROM   purchase_tab P
   WHERE  P.pono = 1001
  )
  SELECT  02, REF(S), 10, 10
   FROM   stock_tab S
   WHERE  S.stockno = 1535;

INSERT INTO THE (
  SELECT  P.line_item_list
   FROM   purchase_tab P
   WHERE  P.pono = 2001
  )
  SELECT  10, REF(S), 1, 0
   FROM   stock_tab S
   WHERE  S.stockno = 1004;

INSERT INTO THE (
  SELECT  P.line_item_list
   FROM   purchase_tab P
   WHERE  P.pono = 2001
  )
  VALUES( line_item_t(11, NULL, 2, 1) ) ;
```

次の文で、表の別名を使って（必須）、フラット化された副問合せの結果を参照します。

```
UPDATE THE (
  SELECT  P.line_item_list
   FROM   purchase_tab P
   WHERE  P.pono = 2001
  ) plist

  SET plist.stockref =
```

```
  (SELECT REF(S)
    FROM  stock_tab S
    WHERE S.stockno = 1011
    )

 WHERE plist.lineitemno = 11 ;
```

データの選択

次の問合せ文では、比較メソッドが暗黙的に呼ばれます。ここでは、比較メソッドで定義した PURCHASE_ORDER_T オブジェクト型の順序付けを Oracle でどのように使用するかを示します。

```
SELECT p.pono
 FROM purchase_tab p
 ORDER BY VALUE(p);
```

上記の命令によってそれぞれの PURCHASE_ORDER_T オブジェクトに対し、マップ・メソッド RET_VALUE が呼ばれます。このメソッドは、オブジェクトの PONO 属性の値を戻すだけなので、選択の結果は発注書番号が昇順に並んだものになります。

次の問合せは、7-6 ページの 「データの検索」の問合せに対応しています。

発注書 1001 の顧客および明細品目データ

```
SELECT  DEREF(p.custref), p.shiptoaddr, p.pono,
        p.orderdate, line_item_list

 FROM   purchase_tab p

 WHERE  p.pono = 1001 ;
```

各発注書の合計値

```
SELECT   p.pono, p.total_value()

 FROM    purchase_tab p ;
```

在庫項目 1004 の発注書および明細品目データ

```
SELECT   po.pono, po.custref.custno,

         CURSOR (
           SELECT  *
            FROM   TABLE (po.line_item_list) L
            WHERE  L.stockref.stockno = 1004
```

```
            )
 FROM    purchase_tab po ;
```

データの削除

次の例は、リレーショナルの例で必要とされた 2 つの削除文と同じ結果になります (7-7 ページの 「データの削除」を参照)。この場合、Oracle では、削除される発注書に属するすべての明細品目が自動的に削除されます。リレーショナルの場合には、もう 1 文の命令を発行しなければなりません。

発注書 1001 の削除

```
DELETE
 FROM   purchase_order
 WHERE  pono = 1001 ;
```

以上で、オブジェクト表を使った発注書例は完了です。次の章では、リレーショナル表とオブジェクト・ビューを使った例を説明します。

8

オブジェクト・ビュー ー 拡張例

この章では、オブジェクト・ビューの使用方法の応用例を説明します。この章には、大きく分けて次の 2 つのセクションがあります。

- 概要
- 発注書の例

概要

オブジェクト・ビューは、表またはビューのデータを具体化した仮想オブジェクト・ビューです。

> **関連項目：** オブジェクト・ビューの説明と使用方法は、『Oracle8 Server 概要』を参照してください。

この章で示す例は、オブジェクト・ビューの定義と使用方法の中でも特に重要なものを挙げています。トリガーの定義では、PL/SQL 言語を使います。それ以外では、Oracle SQL を使用します。

> **関連項目：** SQL 構文と使用方法の詳細は、『Oracle8 Server SQL リファレンス』を参照してください。

PL/SQL を使うと、ここに示す以上の機能が、特にコレクションの要素へのアクセスおよびその操作の面で実現できます。

> **関連項目：** PL/SQL 機能の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

Oracle コール・インタフェース（OCI）を使うクライアント・アプリケーションでは、その広範な機能を利用して、オブジェクト・ビューで定義されたオブジェクトやコレクション型にアクセスし、それらをクライアント・サイドで操作できます。

> **関連項目：** この機能の詳細は、『Oracle 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。

発注書の例

第 7 章の「ユーザー定義データ型 ― 拡張例」では、次の手順に従って、発注書のモデルを作成しています。

1. エンティティおよび関係を設定します。
2. 関係表を作成し、データを移入して、E-R 構造体を構築します。
3. ユーザー定義型のオブジェクト - 関係スキーマを定義して、E-R 構造体のモデルを作成します。
4. オブジェクト - 関係モデルを使って E-R 構造体を構築して、オブジェクト・テーブルを作成し、データを移入します。

この章のアプローチは、最初の 3 ステップは第 7 章と同じですが、最後のステップが異なります。このアプローチでは、オブジェクト・テーブルを作成し、データを移入するかわりに、オブジェクト・ビューを使ってリレーショナル表のデータを仮想オブジェクト・テーブル中に具体化させます。

オブジェクト・ビューの定義

第 7 章で作成した例には、CUSTOMER_TAB および STOCK_TAB、PURCHASE_TAB の 3 つのオブジェクト・テーブルがあります。この章では、それに対応する CUSTOMER_VIEW および STOCK_VIEW、PURCHASE_VIEW の 3 つのオブジェクト・ビューを作成します。

オブジェクト・ビューの作成に使われる文は、次の 4 つの部分から構成されます。

- ビューの名前。
- 基盤とするオブジェクト型の名前。
- 主キー・ベースのオブジェクト識別子のソース。
- 仮想オブジェクト・テーブルにデータを正しく移入する SELECT 文。

customer_view ビューの定義

CUSTOMER_INFO_T オブジェクト型の定義は、7-8 ページを参照してください。次に示すオブジェクト・ビューは、このオブジェクト型を基盤としています。

```
CREATE OR REPLACE VIEW
customer_view OF customer_info_t WITH OBJECT OID(custno) AS
  SELECT  C.custno, C.custname,
          address_t(C.street, C.city, C.state, C.zip),
          phone_list_t (C.phone1, C.phone2, C.phone3)
   FROM   customer_info C ;
```

このオブジェクト・ビューでは、CUSTOMER_INFO 表からデータを選択します。この表の定義は、7-4 ページを参照してください。

CUSTOMER_INFO_T オブジェクト型には、次の属性が定義されています。

```
custno     NUMBER
custname   VARCHAR2(200)
address    address_t
phone_list phone_list_t
```

このオブジェクト・ビューの定義では、CUSTNO および CUSTNAME 属性を、CUSTOMER_INFO 表の対応する名前付きの列から取得します。この定義では、ADDRESS_T オブジェクト型のコンストラクタ関数への引数として、CUSTOMER_INFO 表の STREET および CITY、STATE、ZIP 列を使います。ADDRESS_T オブジェクト型の定義は、7-8 ページを参照してください。

stock_view ビューの定義

STOCK_INFO_T オブジェクト型の定義は、7-8 ページを参照してください。次に示すオブジェクト・ビューは、このオブジェクト型を基盤としています。

```
CREATE OR REPLACE VIEW
stock_view OF stock_info_t WITH OBJECT OID(stockno) AS
```

```
  SELECT  *
   FROM   stock_info ;
```

このオブジェクト・ビューでは、STOCK_INFO 表からデータを選択します。この表の定義は、7-4 ページを参照してください。

このオブジェクト・ビューを埋めるために使うデータの選択は、オブジェクト型の定義と表定義がまったく一致しているので、非常に簡単です。

purchase_view ビューの定義

PURCHASE_ORDER_T オブジェクト型の定義は、7-8 ページを参照してください。次に示すオブジェクト・ビューは、このオブジェクト型を基盤としています。

```
CREATE OR REPLACE VIEW
purchase_view OF purchase_order_t WITH OBJECT OID (pono) AS
  SELECT  P.pono,
          MAKE_REF (customer_view, P.custno),
          P.orderdate, P.shiptodate,
          CAST(
            MULTISET(
              SELECT  line_item_t (
                        L.lineitemno,
                        MAKE_REF(stock_view, L.stockno),
                        L.quantity, L.discount
                        )
               FROM   line_items L
               WHERE  L.pono= P.pono
              )
            AS line_item_list_t
            ),
           address_t (P.shiptostreet, P.shiptocity,
                      P.shiptostate, P.shiptozip)
   FROM   purchase_order P ;
```

このオブジェクト・ビューは、4 ページで定義されている LINE_ITEM 表および 4 ページで定義されている PURCHASE_ORDER 表、前述の CUSTOMER_VIEW および STOCK_VIEW オブジェクト・ビューを基盤としています。

PURCHASE_ORDER_T オブジェクト型には、次の属性が定義されています。

```
pono           NUMBER
custref        REF customer_info_t
orderdate      DATE
shipdate       DATE
line_item_list line_item_list_t
shiptoaddr     address_t
```

このオブジェクト・ビューの定義では、PONO 列を PURCHASE_ORDER 表の PONO 列から取得します。この定義では、MAKE_REF(CUSTOMER_VIEW、CUSTNO) 句を使って、

customer_view オブジェクト・ビュー内の CUSTNO で識別される行オブジェクトに対する REF が作成されます。この REF が、CUSTREF 列になります。

このオブジェクト・ビューの定義では、ORDERDATE および SHIPDATE 列を、PURCHASE_ORDER 表の ORDERDATE および SHIPTODATE 列から取得します。

このオブジェクト・ビューでは、LINE_ITEM_LIST 列を具体化するために、次の項を使います。

```
CAST(
  MULTISET(
    SELECT  line_item_t (
              L.lineitemno,
              MAKE_REF(stock_view, L.stockno),
              L.quantity, L.discount
              )
     FROM   line_items L
     WHERE  L.pono= P.pono
    )
  AS line_item_list_t
  ),
```

演算子 MAKE_REF(STOCK_VIEW、STOCKNO) を使って、STOCK_VIEW オブジェクト・ビュー内の STOCKNO で識別される行オブジェクトに対する REF が作成されます。この REF が、LINE_ITEM_T オブジェクト型のコンストラクタ関数に対する入力引数の 1 つになります。その他の引数には、LINE_ITEMS 表の LINEITEMNO および QUANTITY、DISCOUNT 列を使います。

選択の結果、外側の選択で現在検査されている PURCHASE_ORDER 表の行の PONO 列と一致する PONO 列を持つ LINE_ITEMS 表の各行に対して、1 つずつ LINE_ITEM_T オブジェクトが作成され、一連の LINE_ITEM_T オブジェクトになります。MULTISET 演算子によって、Oracle では一連の LINE_ITEM__T オブジェクトが多重集合と見なされ、CAST 演算子に対する適切な引数となります。これは CAST 演算子により、AS 句での指定に従って LINE_ITEM_LIST_T 型のネストした表に変換されます。

結果的に、ネストした表は、オブジェクト・ビューの LINE_ITEM_LIST_T 列になります。

最後に、この定義では、オブジェクト・ビューの SHIPTOADDR 列を具体化するために、PURCHASE_ORDER 表の SHIPTOSTREET および SHIPTOCITY、SHIPTOSTATE、SHIPTOZIP 列が、ADDRESS_T オブジェクト型のコンストラクタ関数に対する引数として使われています。

オブジェクト・ビューの更新

Oracle では、複合オブジェクト・ビューを更新するときは、INSTEAD OF トリガーを使います。この項では、先ほど定義したオブジェクト・ビューを更新するために必要な INSTEAD OF トリガーを説明します。

Oracle では、ビュー内の行オブジェクトの属性の値を変更するコマンドが発行されると、オブジェクト・ビューの INSTEAD OF トリガーが起動されます。Oracle では、行オブジェクト

の現在の設定値および要求された新しい値のどちらも、トリガー・プログラムから使用できます。トリガー・プログラムでは、現行の値を表す :OLD と、新しい値を表す :NEW の 2 つのキーワードが認識されます。

purchase_view 用の INSTEAD OF トリガー

```
CREATE OR REPLACE TRIGGER
poview_insert_tr INSTEAD OF INSERT ON purchase_view

DECLARE
 line_itms     line_item_list_t ;
 i             INTEGER ;
 custvar       customer_info_t ;
 stockvar      stock_info_t ;
 stockvartemp REF stock_info_t ;

BEGIN
 line_itms := :NEW.line_item_list ;

 SELECT DEREF(:NEW.custref) INTO custvar FROM DUAL ;

 INSERT INTO purchase_order VALUES (
   :NEW.pono, custvar.custno, :NEW.orderdate, :NEW.shipdate,
   :NEW.shiptoaddr.street,    :NEW.shiptoaddr.city,
   :NEW.shiptoaddr.state,     :NEW.shiptoaddr.zip ) ;

 FOR i IN 1..line_itms.COUNT LOOP
   stockvartemp := line_itms(i).stockref ;
   SELECT DEREF(stockvartemp) INTO stockvar FROM DUAL ;

   INSERT INTO line_items VALUES (
     line_itms(i).lineitemno, :NEW.pono, stockvar.stockno,
     line_itms(i).quantity,    line_itms(i).discount ) ;
 END LOOP;

END;
```

このトリガー・プログラムで、PURCHASE_ORDER 表に新しい値を挿入します。次に、ループさせて、新しい LINE_ITEM_LIST 列のネストした表にある LINE_ITEM_T オブジェクトごとに、LINE_ITEM 表に新しい値が挿入されます。

STOCKVARTEMP 変数は、LINE_ITMS(i).STOCKREF で表される REF の暗黙的デリファレンスのかわりとして使っています。

customer_view 用の INSTEAD OF トリガー

```
CREATE OR REPLACE TRIGGER
```

```
custview_insert_tr INSTEAD OF INSERT ON customer_view

DECLARE
  phones  phone_list_t;
  tphone1 customer_info.phone1%TYPE := NULL;
  tphone2 customer_info.phone2%TYPE := NULL;
  tphone3 customer_info.phone3%TYPE := NULL;
BEGIN
  phones := :NEW.phone_list;

  IF phones.COUNT > 2 THEN
    tphone3 := phones(3);
  END IF;

  IF phones.COUNT > 1 THEN
    tphone2 := phones(2);
  END IF;

  IF phones.COUNT > 0 THEN
    tphone1 := phones(1);
  END IF;

  INSERT INTO customer_info VALUES (
   :NEW.custno,       :NEW.custname,      :NEW.address.street,
   :NEW.address.city, :NEW.address.state, :NEW.address.zip,
    tphone1,           tphone2,            tphone3);
END;
```

このトリガー・プログラムで、CUSTOMER_INFO 表を新しい情報に更新します。このプログラムの大部分では、電話番号の :NEW.PHONE_LIST VARRAY をもとに、顧客表の 3 つの電話番号列を更新する処理が行われます。IF 文によって、このプログラムでは :NEW.PHONE_LIST.COUNT より大きい索引を持つ :NEW.PHONE_LIST の要素にはアクセスしないようになっています。

:NEW.phone_list VARRAY と顧客表には若干の不一致があります。VARRAY は最大 10 件の電話番号を保持するように定義されていますが、それに対して顧客表は 3 件の電話番号列しかありません。このトリガー・プログラムでは、索引が 3 より大きい :NEW.PHONE_LIST の要素は廃棄されます。

stock_view 用の INSTEAD OF トリガー

```
CREATE OR REPLACE TRIGGER
stockview_insert_tr INSTEAD OF INSERT ON stock_view

BEGIN
  INSERT INTO stock_info VALUES (
    :NEW.stockno, :NEW.cost, :NEW.tax_code );
```

```
END;
```

このトリガー関数で、STOCK_INFO 表を新しい情報に更新します。

更新のサンプル

次の文で、CUSTOMER_VIEW トリガーが呼ばれます。

```
INSERT INTO customer_view VALUES (
   13, 'Ellan White',
   address_t('25 I Street', 'Memphis', 'TN', '05456'),
              phone_list_t('615-555-1212') );
```

このトリガー文では、CUSTOMER_VIEW オブジェクト・ビューを介して新規顧客がデータベースに挿入されます。

次の文で、PURCHASE_VIEW トリガーが呼ばれます。

```
INSERT INTO purchase_view
  SELECT 3001, REF(c), SYSDATE, SYSDATE,
          CAST(
            MULTISET(
               SELECT line_item_t(41, REF(S), 20, 1)
                FROM  stock_view S
                WHERE S.stockno = 1535
                )
            AS line_item_list_t
            ),
          address_t('22 Nothingame Ave','Cockstown','AZ','44045')

   FROM   customer_view c
   WHERE  c.custno = 1
```

このトリガー文では、PURCHASE_VIEW オブジェクト・ビューを介して新規の発注書がデータベースに挿入されます。番号 1 の顧客は、在庫番号 1535 の品目を 20 点注文しました。このトリガー文によって、発注書には 3001、また明細品目には 41 という番号が割り当てられます。

データの選択

7-22 ページの「データの選択」にある 3 つの問合せは、記述どおりに機能しますが、オブジェクト・テーブル名 PURCHASE_TAB は、オブジェクト・ビュー名 PURCHASE_VIEW に置き換える必要があります。他のオブジェクト・テーブルの問合せも、同様に名前を置き換えることで機能します。

9

データの整合性の保持

この章では、データベースに関連付けたビジネス・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。トピックは次のとおりです。

- 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、使用禁止
- 整合性制約の変更
- 整合性制約の削除
- FOREIGN KEY 整合性制約の管理
- 整合性制約定義のリスト

関連項目： Trusted Oracle を使用している場合、整合性制約の定義、使用可能、使用禁止、削除についての追加情報は、『Trusted Oracle』のマニュアルを参照してください。

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わなければなりません。その後、表のデータを更新する SQL 文を発行すると、Oracle はその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と、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しなければなりません。

整合性制約を使用する場合

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も コストがかかりません。整合性制約の意味は明確に定義され、Oracle がそれらを施行するための内部処理は、SQL よりも下位レベルで最適化されています。アプリケーションは SQL を使用するため、このレベルの最適化を行うことができません。

SQL 文はネットワークを介して送信しなければならないので、SQL 文を使用してのビジネス・ルールの施行だと、ネットワーク化された環境ではさらにコストがかかる場合があります。この場合、整合性制約を使用することによって、転送時に発生するパフォーマンス上のオーバーヘッドをなくすことができます。

例 EMP 表の各従業員が DEPT 表にリストされている部門だけを指定することを保証するには、最初に DEPT 表の DEPTNO 列に PRIMARY KEY 制約を次のように作成します。

```
ALTER TABLE dept
    ADD PRIMARY KEY (deptno)
```

次に、DEPT 表の主キーを参照する EMP 表の DEPTNO 列に参照整合性制約を作成します。

```
ALTER TABLE emp
    ADD FOREIGN KEY (deptno) REFERENCES dept(deptno)
```

これ以降、表に新しい従業員レコードを追加すると、Oracle はその部門番号が部門表に存在していることを保証します。

整合性制約を使用しないでこのルールを施行するには、アプリケーションが、それぞれの新しい従業員レコードを検査し、その部門番号が既存の部門に属していることを保証しなければなりません。この検査は、SELECT 文を発行して DEPT 表を問い合せるなどして行います。

整合性制約の利用

最高のパフォーマンスを得るには、整合性制約を定義し、使用可能にし、それらを利用してアプリケーションを開発します。つまり、アプリケーションの SQL 文でビジネス・ルールを施行することは避けるようにします。

ただし、場合によっては、整合性制約だけでなく、アプリケーションによってビジネス・ルールを施行してもかまいません。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、整合性制約よりも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後には、そのことをただちにユーザーに知らせる必要があります。

整合性制約は SQL 文の発行時に施行されるため、整合性制約では、ユーザーが 20 件の値をすべて入力し、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。しかし、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らただち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。

NOT NULL 整合性制約の使用

デフォルトで、すべての列を NULL にできます。NOT NULL 制約は、表の列で常に値が必要とされるものに定義します。

たとえば、EMP 表で、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いなくても、特に問題はありません。また、従業員の中には、コミッション（歩合）を受けていない者もいます。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。しかし、従業員名が空白ということは、通常は認められません。このため、この列は NOT NULL 整合性制約の適用に適しています。

NOT NULL 制約は他の整合性制約と組み合せて、表の特定の列に存在できる値をさらに制限します。NOT NULL と UNIQUE キー整合性制約を組み合せると、必ず値が入力され、新しい行のデータと既存の行のデータが重複する可能性はなくなります。このような組合せの詳細は、次の項を参照してください。

関連項目： 9-9 ページの「親表と子表の関連」

図 9-1　NOT NULL 整合性制約

Table EMP

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　NULL　NOT NULL　NULL

デフォルトの列値の設定

正当なデフォルト値は、列 LEVEL、または ROWNUM、PRIOR を参照しない式、あるいはリテラルを含みます。デフォルト値は、式 SYSDATE および USER、USERENV、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、その列のデータ型に一致しているか、あるいは変換可能でなければなりません。

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

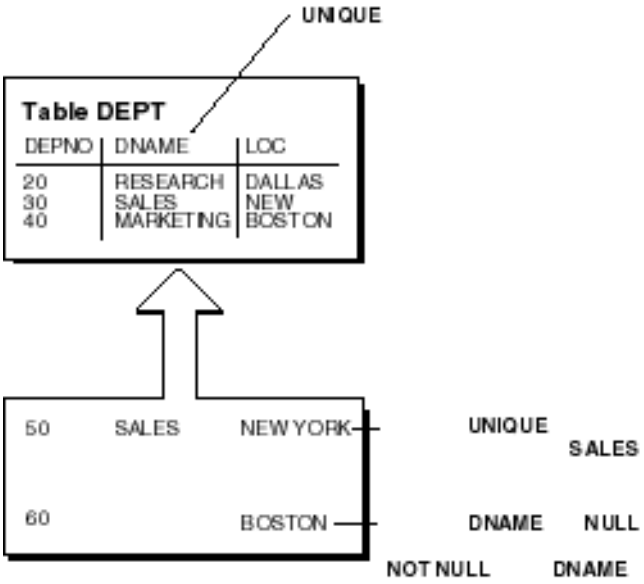
デフォルト値は、典型的な値を含む列にだけ割り当てられます。たとえば、DEPT 表では、多くの部門が 1 つのサイトに位置する場合、LOC 列に対するデフォルト値を、この値に設定できます（たとえば、NEW YORK）。

また、表の一部の列を公開するビューを作成するときにも、デフォルトは有効になります。たとえば、ユーザーはビューを通して表に行を挿入することを許されていることがあります。ビューは、ユーザーに関係があるすべての列を示すために定義されています。しかし、実表には、表の各行を挿入するユーザーを記録する列 INSERTER があり、この列はビューの定義に含まれていない可能性もあります。列 INSERTER は、USER 関数を使用してこの列を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。

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

デフォルト列値の割当ての別の例は、4-3 ページの「表を作成する」を参照してください。

図 9-2　UNIQUE 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となり、行が重複しないことを保証できます。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。

- データ値が一意となる列を選択する。

  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なければなりません。

- データ値が変更されない列を選択する。

  主キー値は、表の行を識別するためにだけ使用されます。主キー値は、他の目的のために使用されるデータを含んではいけません。このため主キー値は、変更する必要性のほとんどないデータにしなければなりません。

- NULL をまったく含まない列を選択する。

  定義により PRIMARY KEY 制約は、主キーの一部分である任意の列が NULL に設定されている行を入力することはできません。

- 短いか数値型の列を選択する。

  主キーが短ければ、入力は容易となります。数値の主キーは、順序番号を使用して容易に生成できます。

- 複合主キーの選択を避ける。

  複合主キーが許可される場合でも、前述の条件を満足しません。たとえば、複合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。

UNIQUE 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とすべき列に、間違って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の項を参照してください）。UNIQUE キー制約を使用するかどうかを決定するときに、各行の間でキー値の重複を避ける場合に限り、この規則を使用してください。一意キーのデータは、表上で重複させることはできません。

注意： UNIQUE キー制約では、NULL を入力できますが、UNIQUE キー制約の検索は複数列にまたがって行われるため、部分的に NULL 値が入る複合 UNIQUE キー制約の NULL 以外の列内には、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。

一意キーの適切な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）によって関連付けられた場合は、2 つの表の間の関連を維持するために、必ず PRIMARY または UNIQUE キーの制約をその親表の列で定義し、FOREIGN KEY 制約を子表の列で定義します。この関連に応じて、9-9 ページの「親表と子表の関連」にリストするように、外部キーを含むその他の整合性制約を定義できます。

図 9-3　は、EMP 表の DEPTNO 列に定義された外部キーを示しています。これは、この列の各値が DEPT 表の主キー（DEPTNO 列）の値と一致していることを保証します。このため、EMP 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、このような複合外部キーは、正確に同じ構造を持つ複合主キーまたは複合一意キーを参照しなければなりません。つまり、これらは列の数とデータ型が同一でなければなりません。複合主キーまたは複合一意キーには 16 列までという制限があるため、複合外部キーも最大 16 列に制限されています。

NULL と外部キー

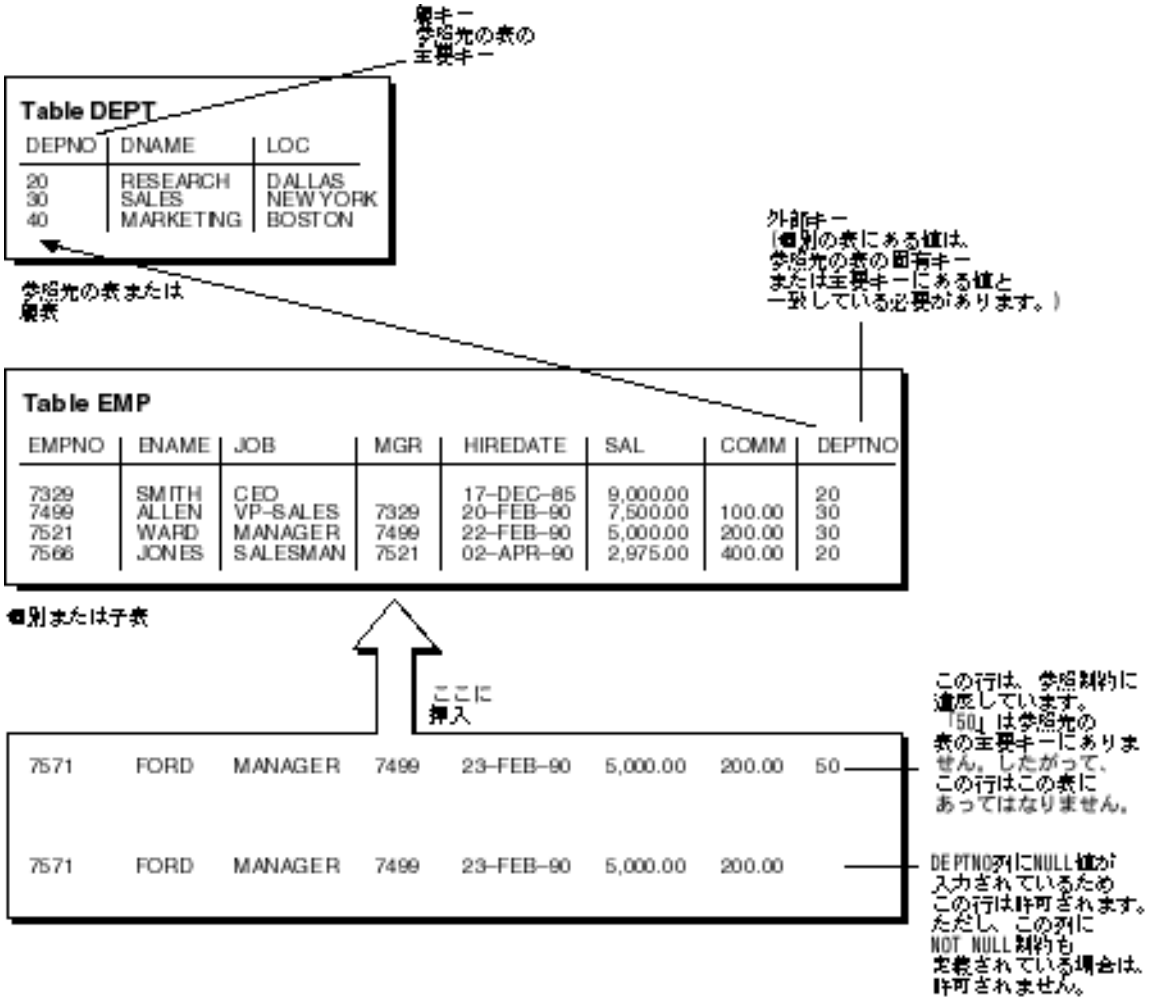
デフォルト（つまり、NOT NULL 句または CHECK 句を指定しない場合）の場合では、ANSI/ISO 標準にもよると、FOREIGN KEY 制約は複合外部キーに対して「不一致規則」を施行します。CHECK と NOT NULL 制約を使って、次のような「完全一致」および「部分一致」の規則を施行することもできます。

- 複合外部キーの NULL に対する「完全一致」規則（キーのすべての構成要素が NULL であるか、NULL 以外のものであることを要求する）を施行するためには、複合外部キーのすべてが NULL か NULL 以外であるものだけを許可する CHECK 制約を定義する。たとえば、列 A、B、C で構成される複合キーを次のとおり指定することができます。

  ```
  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、複合外部キーの NULL に対する「部分一致」規則（NULL 以外の列が、参照先の主キーや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ない。この場合の処理に、第 13 章「データベース・トリガーの使用」で説明するトリガーを使用できることがよくあります。

図 9-3　参照整合性制約


親キー
参照先の表の
主要キー
Table DEPT
DEPNO | DNAME | LOC
20 | RESEARCH | DALLAS
30 | SALES | NEW YORK
40 | MARKETING | BOSTON
参照先の表または
親表
外部キー
（子別の表にある値は、
参照先の表の固有キー
または主要キーにある値と
一致している必要があります。）
Table EMP
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 | 20
子別または子表
ここに
挿入
7571 FORD MANAGER 7499 23-FEB-90 5,000.00 200.00 50
この行は、参照制約に
違反しています。
「50」は参照先の
表の主要キーにありま
せん。したがって、
この行はこの表に
あってはなりません。
7571 FORD MANAGER 7499 23-FEB-90 5,000.00 200.00
DEPTNO列にNULL値が
入力されているため
この行は許可されます。
ただし、この列に
NOT NULL制約も
定義されている場合は、
許可されません。

親表と子表の関連

親表と子表の間の関連を、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知ることができます。

外部キーに他の制約がない場合　他の制約が外部キーについてまったく定義されていないときは、子表の任意の数の行は、同一の親キー値を参照できます。このモデルは、外部キーに NULL を許可します。

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の 「1 対多」関連を確立します。EMP と DEPT の間のこのような関連の例は、図 9-3 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（外部キーで NULL）場合があります。

外部キーに NOT NULL 制約がある場合　NULL が外部キーで許可されていないときは、そのため、子表の各行は、親キーの値を明示的に参照しなければなりません。ただし、子表の任意の数の行は、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の「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なければならず、外部キーに値の欠如（NULL）があってはいけません。前の項の具体例を使用して、そうした関連を説明できます。ただし、このモデルでは、必ず従業員は特定の部門への参照を持っていなければなりません。

外部キーに UNIQUE 制約がついている場合　UNIQUE 制約が外部キーで定義されているときは、子表の中の 1 つの行だけが、親キー値を参照できます。このモデルは、外部キーに NULL を許可します。

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の「1 対 1」関連を確立します。たとえば EMP 表に、会社の保険プランで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仮定します。また、INSURANCE という表には、MEMBERNO という主キーがあり、表の他の列は従業員の保険証券に関連する情報を保持しています。次の理由により、EMP 表の MEMBERNO は、外部キーかつ一意キーでなければなりません。

- EMP 表と INSURANCE 表の参照整合性規則を施行するため（FOREIGN KEY 制約）
- 各従業員の会員番号を一意にするため（UNIQUE キー制約）

外部キーに UNIQUE および NOT NULL 制約がついている場合　UNIQUE 制約と 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 行だけがある親キー値を参照できます。NULL が外部キーで許可されていないので、子表の各行は、明示的に親キーの値を参照しなければなりません。

このモデルは、外部キーに未定の値（NULL）を許可しない親キーと外部キーとの間の「1 対 1」関連を確立します。前の例を拡張して、それぞれの従業員が一意の会員番号を持つことを保証することに加えて、EMP 表の MEMBERNO 列に NOT NULL 制約を追加すれば、EMP 表の MEMBERNO 列に不定の値（NULL）は許可しないことを施行できます。

複数の FOREIGN KEY 制約

Oracle では、1 つの列を複数の FOREIGN KEY 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 つの異なる複合外部キーの一部になっている場合に、この状況が起こる可能性があります。

同時実行性制御、索引、外部キー

Oracle は、依存外部キー値に関連する親キーの同時実行性が最大になるように制御します。親子キーの関連の維持のために使用する同時実行性メカニズムを選択できるので、これは状況によっては大いに役立ちます。次の項では、可能性として考えられる各種状況を説明するとともに、各状況に合った設定を紹介していきます。

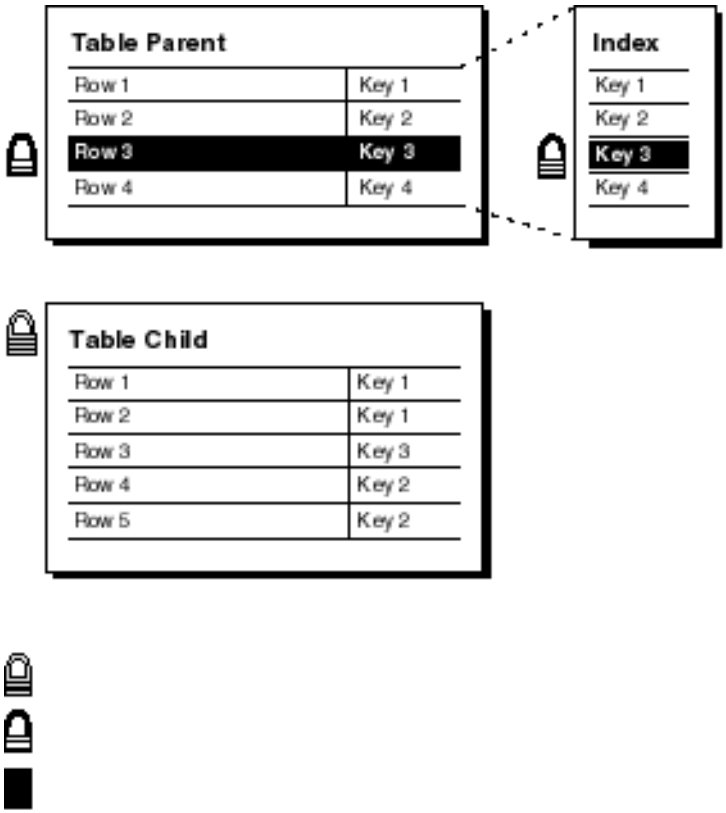
外部キーに索引が定義されていない場合　図 9-4 は、外部キーに索引が定義されてなく、親表で行が更新または削除されるときに Oracle で使用されるロック・メカニズムを示しています。親表に挿入するのに、子表をロックする必要はありません。

親表に対して DELETE 文を指定するトランザクションがコミットされるまで、子表全体への共有ロックが要求されます。外部キーが ON DELETE CASCADE を指定している場合に DELETE 文を発行すると、子表に対して表レベルの共有副排他ロックが発生します。子表が参照する列に影響を与える親表の UPDATE 文には、子表全体の共有ロックも必要です。共有ロックは読取りだけを許可します。したがって、UPDATE または DELETE を含むトランザクションがコミットされるまで、子表に対して INSERT 文、UPDATE 文、DELETE 文は発行できません。なお、問合せは子表において許可されています。

この状況は、親表に対する更新および削除が回避できる場合は許容できます。

子表に対して INSERT 文、UPDATE 文、DELETE 文を発行しても、親表に対するロックは獲得されません。ただし、INSERT 文と UPDATE 文は、親表の索引に対する行ロックが解除されるのを待ちます。

図 9-4　外部キーに索引が定義されていない場合のロック・メカニズム



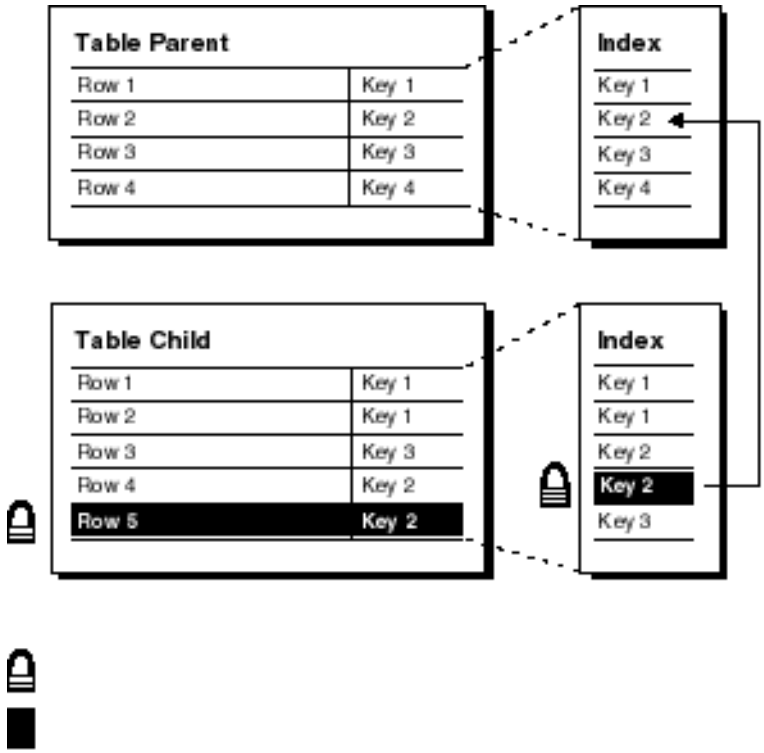
外部キーに索引が定義されている場合　図 9-5 は、外部キーに索引が定義され、子表で新規行が挿入または更新、削除されるときに、Oracle で使用されるロック・メカニズムを示しています。

挿入、更新、削除によって、親表またはその索引に対して、いかなる種類の表ロックも獲得されないので注意してください。このため、挿入、更新、削除、問合せなどの任意の DML 文を、親表に対して発行できます。

子表において更新アクティビティが発生している間に、親表に任意の更新または削除アクティビティがある場合、この状況は最適です。親表に対する挿入、更新、削除には、子表に

対するロックは必要ありません。ただし、更新と削除は、子表の索引に対する行レベルのロックが解除されるのを待ちます。

図 9-5　外部キーに索引が定義されている場合のロック・メカニズム



子表が ON DELETE CASCADE を指定している場合に親表からの削除を行うと、子表からも削除される場合があります。この場合の待ち規則とロック規則は、親表から削除した後に、自分で子表から削除した場合と同じです。

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ノードにまたがって定義することはできません（つまり、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 指定することはできません）。しかし、トリガーを使用すると、複数のノードにわたって親子の表の関連を 維持管理できます。参照整合性を施行す

るトリガーの詳細は、第 13 章「データベース・トリガーの使用」を参照してください。参照整合性の維持管理のためにトリガーを使用する場合、分散オプションが必要です。詳細は、『Oracle8 Server 分散システム』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わたる参照整合性を定義することにした場合には、ネットワーク障害が親表だけ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 つのデータベースの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しなければならないので、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

整合性規則を施行する必要性があり、論理式を基にした評価が可能な場合は、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。

関連項目： 9-14 ページの「CHECK および NOT NULL 整合性制約」

適切な CHECK 制約の具体例は次のとおりです。

- 給与の値が 10000 を超えないように、EMP 表の SAL 列に CHECK 制約を定義する。
- 「BOSTON」、「NEW YORK」、「DALLAS」に限り許可されるように、DEPT 表の LOC 列に CHECK 制約を定義する。
- 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 列と COMM 列に CHECK 制約を定義する。

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定でなければなりません。文が条件を偽に評価させるような場合、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なければならない。
- 条件に副問合せや順序を含めることはできない。
- 条件に SQL 関数 SYSDATE または UID、USER、USERENV を含めることはできない。

- 条件に疑似例 LEVEL または PRIOR、ROWNUM を含めることはできない。

  関連項目： これらの疑似例については、『Oracle8 Server 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ない。

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だけ CHECK 制約に違反する、という ANSI/ISO 標準に留意してください。つまり、真と不定は検査条件には違反しません。したがって、定義した CHECK 制約で施行する必要のある規則が実際に施行されていることを確認してください。

たとえば、次の CHECK 制約について考えます。

```
CHECK (sal > 0 OR comm >= 0)
```

この規則は、「従業員の給与が 0 より大きい、または従業員の歩合が 0 以上でなければ、EMP 表の行として認められない」と解釈できます。ただし、給与に NULL、歩合に負の値を持つ行は、検査条件全体が不定と評価されるために、挿入しても CHECK 制約には違反しないので注意してください。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 条件になるかの説明は、『Oracle8 Server SQL リファレンス』の論理演算子 AND と OR の真理値表を参照してください。

複数の CHECK 制約

一列には、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。

CHECK および NOT NULL 整合性制約

ANSI/ISO 標準によれば、NOT NULL 整合性制約は CHECK 整合性制約の 1 つの具体例であり、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。使い易さという点では、「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べきです。

複合キーがすべて NULL またはすべて値を持つ場合は、CHECK 整合性制約を使用しなければなりません。たとえば、次の CHECK 整合性制約の式を指定すると、列 C1 と C2 を構成する複合キーのキー値が、すべて NULL またはすべて値を持つことができます。

```
CHECK ((c1 IS NULL AND c2 IS NULL) OR
    (c1 IS NOT NULL AND c2 IS NOT NULL))
```

整合性制約の定義

整合性制約は、SQL コマンド CREATE TABLE または ALTER TABLE の制約句を使用して定義します。次の 2 項では、これらのコマンドを使用して整合性制約を定義する方法について説明します。

注意：『Trusted Oracle』を使用している場合には、追加事項があります。詳細は、『Trusted Oracle』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 (
    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 (
    empno    NUMBER(5) PRIMARY KEY,
    ename    VARCHAR2(15) NOT NULL,
    job      VARCHAR2(10),
    mgr      NUMBER(5) CONSTRAINT mgr_fkey
                     REFERENCES emp,
    hiredate DATE,
    sal NUMBER(7,2),
    comm     NUMBER(5,2),
    deptno   NUMBER(3) NOT NULL
               CONSTRAINT dept_fkey
                   REFERENCES dept 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。

```
ALTER TABLE dept
    ADD PRIMARY KEY (deptno);

ALTER TABLE emp
    ADD CONSTRAINT dept_fkey FOREIGN KEY (deptno) REFERENCES dept
    MODIFY (ename VARCHAR2(15) NOT NULL);
```

ALTER TABLE コマンドでの整合性制約の制限

ALTER TABLE 文が発行されるときは、表にはおそらくデータが入っているため、把握しておかなければならない制限がいくつかあります。表 9–1 は、各制約の種類と ALTER TABLE コマンドに関連する制限を列挙しています。

表 9–1　ALTER TABLE コマンドによる整合性制約定義の制限

制約の種類	表の既存列に対する追加	表に新しい列を追加
NOT NULL	この列に NULL 値を持つ行があると定義することはできない。*	表に行があると定義することはできない。
UNIQUE	キーに重複値があると定義することはできない。*	常に定義可能。
PRIMARY KEY	キーに重複値または NULL 値があると定義することはできない。*	表に行があると定義することはできない。
FOREIGN KEY	外部キーに親キー値を参照しない値があると定義することはできない。*	常に定義可能。
CHECK	列に検査条件を満たさない値があると、定義することはできない。*	常に定義可能。

*DISABLE 句を文に指定しないと想定

ALTER TABLE 文を持つ制約を定義して、これらの制限の 1 つでも違反する場合、この文はロールバックされ、違反を説明するエラー情報が戻されます。

必要な権限

制約の作成ユーザーは、制約のある表を作成する権限（つまり、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）を持っていなければなりません。さらに、UNIQUE キーと PRIMARY KEY 整合性制約では、表の所有者が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を持っている必要があります。FOREIGN KEY 整合性制約では、その他にもいくつかの権限が必要です。

関連項目： 特定の情報については、9-24 ページの「FOREIGN KEY 整合性制約に必要な権限」を参照してください。

整合性制約の命名

NOT NULL、UNIQUE KEY、PRIMARY KEY、FOREIGN KEY、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なければなりません。制約名を指定しないと、Oracle が生成して名前を割り当てます。

制約句の CONSTRAINT オプションの具体例として、前述の例中の CREATE TABLE と ALTER TABLE 文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。

関連項目： データ・ディクショナリ・ビューの例は、9-25 ページの「整合性制約定義のリスト」を参照してください。

定義時の制約の使用可能、使用禁止

整合性制約が、特に DISABLE 句を使用して使用禁止の状態で作成されない限り、CREATE 文または ALTER TABLE 文で定義されたときは必ず、デフォルトにより Oracle は自動的に制約を使用可能（強制的）にします。

関連項目： 制約を使用可能および使用禁止にする際の重要な問題の詳細は、9-20 ページの「キー整合性制約の使用可能、使用禁止」を参照してください。

UNIQUE キーおよび PRIMARY KEY、FOREIGN KEY

UNIQUE キーおよび PRIMARY KEY、FOREIGN KEY の整合性制約を定義する際には、必ずいくつかの重要な問題といくつかの必要条件を 把握していなければなりません。FOREIGN KEY 制約の定義と管理の詳細は次の項を参照してください。

関連項目： 9-23 ページの「FOREIGN KEY 整合性制約の管理」UNIQUE キーおよび PRIMARY KEY の制約は、通常、データベース管理者が使用可能にします。詳細は『Oracle8 Server 管理者ガイド』を参照してください。

整合性制約の使用可能、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と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、制約により定義した規則は、制約を定義す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、制約により定義した規則は、制約で指定した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となります。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ので、この文は必ずしも真であるとは限りません。

制約を使用可能、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ためには、必ずその制約を使用可能にしなければなりません。ただし、状況によっては、パフォーマンスの理由から、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、次の場合などが考えられます。

- SQL*Loader を使用して、表に大量のデータをロードする場合
- 表に対して大規模な変更を行うバッチ作業を実施する場合（たとえば、既存の番号に 1000 を加えて全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

上述のような場合に、作業のパフォーマンス改善のために、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遵守しない場合、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、**制約を使用可能にすることはできません。**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しなければなりません。

制約を使用可能にしようとする際に、特定の整合性制約に対する例外を識別できます。この手順は、9-21 ページの「例外のレポート」で説明します。

定義について

CREATE TABLE 文や ALTER TABLE 文に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、両方とも、整合性制約を定義し、使用可能にします。

```
CREATE TABLE emp (
    empno NUMBER(5) PRIMARY KEY, . . . );
 ALTER TABLE emp
    ADD PRIMARY KEY (empno);
```

整合性制約を定義し使用可能にしようとする ALTER TABLE 文は、表の行がその整合性制約に違反している場合、異常終了する可能性があります。この場合には文はロールバックされ、制約定義は格納されず、また使用可能にもされません。整合性制約に違反する行の詳細は、9-21 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、両方とも、整合性制約を定義し、使用禁止にします。

```
CREATE TABLE emp (
    empno NUMBER(5) PRIMARY KEY DISABLE, . . . );

ALTER TABLE emp
    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、異常終了す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ので許可されます。

整合性制約の使用可能、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。
- 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。

使用禁止になっている制約を使用可能にする

次の文は、使用禁止にされた整合性制約を使用可能にする 2 つの例です。

```
ALTER TABLE dept
    ENABLE CONSTRAINT dname_ukey;

ALTER TABLE dept
    ENABLE PRIMARY KEY,
```

```
    ENABLE UNIQUE (dname, loc);
```

整合性制約を使用可能にしようとする ALTER TABLE 文は、表の行がその整合性制約に違反すると、異常終了します。この場合には文はロールバックされ、制約は使用可能になりません。整合性制約に違反する行の詳細は、9-21 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 2 つの例です。

```
ALTER TABLE dept
    DISABLE CONSTRAINT dname_ukey;

ALTER TABLE dept
    DISABLE PRIMARY KEY,
    DISABLE UNIQUE (dname, loc);
```

助言 － 参照のためのデータ・ディクショナリの使用： 前の項の例文において、制約を使用可能あるいは禁止にするために、制約についての情報を把握していなくてはなりません。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、9-25 ページの「整合性制約定義のリスト」および『Oracle8 Server 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、使用禁止

UNIQUE キー、PRIMARY KEY、FOREIGN KEY の各整合性制約を使用可能または使用禁止にする際には、必ずいくつかの重要な問題といくつかの必要条件を把握していなければなりません。FOREIGN KEY 制約の使用可能化および使用禁止化、管理の詳細は、次の項を参照してください。UNIQUE キー制約と PRIMARY KEY 制約は、通常データベース管理者が管理します。詳細は、次のマニュアルを参照してください。

関連項目： 9-23 ページの「FOREIGN KEY 整合性制約の管理」および『Oracle8 Server 管理者ガイド』

パラレル・ダイレクト・パスのロード後の制約の使用可能

SQL*Loader では、複数の並行セッションで同じ表へのダイレクト・パスのロードが可能です。個々の SQL*Loader セッションは、ダイレクト・パスのロード後に、表に対する制約を再度使用可能にしようとするので、別セッションでデータのロードが終了する前に、ある

セッションが制約を再度使用可能にしようとする危険性があります。この場合、ロードを完了する最初のセッションは、制約を使用可能にできません。これは、残りのセッションが、表に対する共有ロックを持っているためです。

制約によっては、ダイレクト・パスのロード後に、再度使用可能にならない危険性があるので、ロードの完了後に制約の状態をチェックして、正しく使用可能になったことを確認する必要があります。

PRIMARY KEY 制約と UNIQUE KEY 制約

PRIMARY KEY 制約と UNIQUE キー制約は、使用可能な状態になると表の索引を作成します。したがって、表が非常に大きい場合、ダイレクト・パスのロード・セッション後に使用可能にするには、かなりの時間がかかる場合があります。

これらの制約を、ロード後に手動で使用可能にする（そして自動使用可能化機能を指定しない）ことをお薦めします。そうすれば、制約を使用可能にする前に、必要な索引を並行して手動で作成し、時間を節約できます。

> **関連項目：** 索引の並行作成の詳細は、『Oracle8 Server チューニング』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... 文を発行する際に例外がまったくなければ、整合性制約は使用可能となり、後続のすべての DML 文がその使用可能となった整合性制約に従います。

制約が使用可能となる際に、例外が存在する場合は、エラーが戻され、整合性制約は使用禁止の状態のままになります。整合性制約の例外が存在しているため文が正常に実行されない場合、文はロールバックされます。例外が存在している場合には、制約に対するすべての例外を更新または削除するまで、制約を使用可能にできません。

整合性制約に違反している行を判断するためには、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。EXCEPTIONS オプションにより、すべての例外となる行の ROWID、表所有者、表名、制約名がある特定の表に格納されます。たとえば、次の文は DEPT 表の主キーを使用可能にするための指定ですが、例外が存在する場合は、EXCEPTIONS という名前の表に情報が挿入されます。

```
ALTER TABLE dept ENABLE PRIMARY KEY EXCEPTIONS INTO exceptions;
```

ENABLE 句の EXCEPTIONS オプションから情報を受け取るためには、適切な例外レポート表を作成します。UTLEXCPT.SQL スクリプトを実行することによって、例外表を作成できます。このスクリプトは、EXCEPTIONS という名前の表を作成します。スクリプトを修正して、再実行することによって、別の名前を持つ例外表を追加作成できます。

例として、重複する主キー値が DEPT 表に存在し、DEPT の PRIMARY KEY 制約の名前が SYS_C00301 である場合には、前の文を実行することによって、次のような行が EXCEPTIONS 表に格納されます。

```
SELECT * FROM exceptions;
```

```
ROWID                  OWNER   TABLE_NAME       CONSTRAINT
------------------     ------  ------------     -----------
AAAA5bAADAAAAEQAAA     SCOTT   DEPT             SYS_C00301
AAAA5bAADAAAAEQAAB     SCOTT   DEPT             SYS_C00301
```

レポートをわかりやすくするには、問合せ情報により、例外レポート表とマスター表の行を結合し、特定の制約に違反している実際の行をリストします。たとえば、次のとおりです。

```
SELECT deptno, dname, loc FROM dept, exceptions
    WHERE exceptions.constraint = 'SYS_C00301'
    AND dept.rowid = exceptions.row_id;

DEPTNO     DNAME          LOC
---------- -------------- -------------
10         ACCOUNTING     NEW YORK
10         RESEARCH       DALLAS
```

制約が指定された表から、制約に違反しているすべての行を更新または削除しなければなりません。例外を更新する場合には、制約に違反する値を、違反しないような値あるいはNULL（可能な場合）にしなければなりません。制約を施行する表（マスター表）の行を更新または削除した後は、対応する例外レポート表の例外行は、他の例外レポートとの混同を避けるために削除します。マスター表と例外レポート表を更新する文は、必ず同一トランザクション上で行い、トランザクション一貫性を保証しなければなりません。

たとえば、前回の使用例の例外を訂正するためには、次のトランザクションを発行できます。

```
UPDATE dept SET deptno = 20 WHERE dname = 'RESEARCH';
DELETE FROM exceptions WHERE constraint = 'SYS_C00301';
COMMIT;
```

例外を管理するとき、最終的な目標は、例外レポート表の例外をすべて除去することにあります。すべての例外を除去した後、制約を再度使用可能にしなければなりません。つまり、例外が処理された後、制約は自動的に使用可能になりません。

制約が使用禁止になっている表に対する現行の例外を訂正している間、他のユーザーは新しい例外を作成する文を発行できます。

整合性制約の変更

整合性制約は変更できません。整合性制約によって定義されているアクションを変更する必要がある場合には、まず既存の整合性制約を削除してから、新しい制約を作成します。

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、またはその制約が必要なく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
    DROP UNIQUE (dname, loc);

ALTER TABLE emp
    DROP PRIMARY KEY,
    DROP CONSTRAINT dept_fkey;

DROP TABLE emp CASCADE CONSTRAINTS;
```

UNIQUE キー、PRIMARY KEY、FOREIGN KEY の各整合性制約を削除する際には、あらかじめいくつかの特記事項や いくつかの必要条件を把握しておかなければなりません。FOREIGN KEY 制約の削除の詳細は、次の項を参照してください。UNIQUE キー制約と PRIMARY KEY 制約は、通常データベース管理者が管理します。次のマニュアルを参照してください。

> **関連項目：** 9-23 ページの「FOREIGN KEY 整合性制約の管理」および『Oracle8 Server 管理者ガイド』

FOREIGN KEY 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、削除に関する一般的な情報については、前の項で記載しました。次の項で、特に FOREIGN KEY 整合性制約の問題に重点を置きながら、この情報を補足します。

FOREIGN KEY 整合性制約の定義

次の内容は、FOREIGN KEY 整合性制約を定義する際に重要な情報です。

データ型の一致

参照整合性制約を定義する際に、依存する表と参照される表の対応する列名は、一致している必要はありません。ただし、データ型は同じでなければなりません。

複合外部キー

外部キーは、親表の主キーまたは一意キーを参照し、PRIMARY KEY 制約と UNIQUE キー制約は索引を基に施行されるため、複合外部キーは 16 列以内に制限されています。

主キーの暗黙的な参照

FOREIGN KEY 制約（一列または複合列）を定義する際、REFERENCES オプションに列リストを指定しない場合には、Oracle は指定した表の主キーを参照す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 により自動的にチェックされます。そうでない場合には、エラー情報が戻されます。

FOREIGN KEY 整合性制約に必要な権限

FOREIGN KEY 制約を作成する場合には、制約を作成するユーザーは必ず親表と子表に対するアクセス権限を持っていなければなりません。

- **親表** 参照整合性制約を作成するユーザーは、親表を所有するか、親表の親キーを構成する列に対して REFERENCES オブジェクト権限を持っていなければなりません。
- **子表** 参照整合性制約の作成者は、表を作成する権限（CREATE TABLE または CREATE ANY TABLE のシステム権限）、または子表を変更する権限（子表の ALTER オブジェクト権限または ALTER ANY TABLE システム権限）を持っていなければなりません。

どちらの場合も、必要な権限をロールを介して取得することは**できません**。つまり、特権は明示的に制約の作成ユーザーに付与されなければなりません。

これらの制限により、次のことが可能となります。

- 子表の所有者は、所有者自身の表に対する制約や、自身の表の制約を作成できる他のユーザーを、明示的に決定できる。
- 親表の所有者は、外部キーが所有者自身の表の主キーと一意キーに依存可能であるかどうか、明示的に決定できる。

外部キーに対する参照アクションの指定

Oracle では、FOREIGN KEY 制約の定義中での指定で、2 つの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 FOREIGN KEY 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。たとえば、次のとおりです。

  ```
  CREATE TABLE emp (
  . . .,
  FOREIGN KEY (deptno) REFERENCES dept);
  ```

- **ON DELETE CASCADE アクション** このアクションにより、親キーの参照中のデータを削除できます（更新はできません）。親キーの参照中のデータが削除されると、削除された親キー値に依存する子表のすべての行も削除されます。この参照アクションを指定

するためには、FOREIGN KEY 制約の定義に ON DELETE CASCADE オプションを指定します。たとえば、次のとおりです。

```
CREATE TABLE emp ( . . .,
    FOREIGN KEY (deptno) REFERENCES dept
        ON DELETE CASCADE);
```

FOREIGN KEY 整合性制約を使用可能にする

参照中の主キーまたは一意キーの制約が存在しなかったり使用可能になっていない場合には、FOREIGN KEY 整合性制約を使用可能にすることはできません。

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

各ビューの詳細は、『Oracle8 Server リファレンス・マニュアル』を参照してください。

例

いくつかの整合性制約を定義する次の CREATE TABLE 文とその使用例について考えてみます。

```
CREATE TABLE dept (
    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 (
    empno    NUMBER(5) PRIMARY KEY,
    ename    VARCHAR2(15) NOT NULL,
    job      VARCHAR2(10),
    mgr      NUMBER(5) CONSTRAINT mgr_fkey
             REFERENCES emp ON DELETE CASCADE,
    hiredate DATE,
    sal      NUMBER(7,2),
    comm     NUMBER(5,2),
    deptno   NUMBER(3) NOT NULL
    CONSTRAINT dept_fkey REFERENCES dept);
```

例 1: アクセス可能なすべての制約を表示 すべての表に定義されているアクセス可能な全制約が一覧で表示されます。

```
SELECT constraint_name, constraint_type, table_name,
        r_constraint_name
    FROM user_constraints;
```

この項の冒頭の例文の場合、次のようなリストが戻されます。

```
CONSTRAINT_NAME  C TABLE_NAME  R_CONSTRAINT_NAME
---------------  - ----------- ------------------
SYS_C00275       P DEPT
DNAME_UKEY       U DEPT
LOC_CHECK1       C DEPT
SYS_C00278       C EMP
SYS_C00279       C EMP
SYS_C00280       P EMP
MGR_FKEY         R EMP         SYS_C00280
DEPT_FKEY        R EMP         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）のものもあればシステム指定（たとえば、SYS_C00275）のものもある。
- 各制約タイプは、CONSTRAINT_TYPE 列に 1 文字として表示される。次の表に、各制約タイプに対応する文字を掲載します。

制約タイプ	文字
PRIMARY KEY	P
UNIQUE KEY	U
FOREIGN KEY	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。各ビューと WITH CHECK OPTION の詳細は、第 4 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 制約を CHECK 制約と区別する 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 表と DEPT 表の CHECK 制約であるかとを区別するためには、次の問合せを発行します。

```
SELECT constraint_name, search_condition
    FROM user_constraints
    WHERE (table_name = 'DEPT' OR table_name = 'EMP') AND
        constraint_type = 'C';
```

この項の冒頭の CREATE TABLE 文の場合、次のような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る。
- ユーザー定義による CHECK 制約の条件は、明示的に SEARCH_CONDITION 列にリストされる。

例 3: 整合性制約を構成する列名をリストする 次の問合せの結果、すべての表で定義されているアクセス可能な制約を構成する全列が一覧で表示されます。

```
SELECT constraint_name, table_name, column_name
    FROM user_cons_columns;
```

この項の冒頭の例文の場合、次のようなリストが戻されます。

```
CONSTRAINT_NAME  TABLE_NAME  COLUMN_NAME
---------------  -----------  ---------------
DEPT_FKEY        EMP          DEPTNO
DNAME_UKEY       DEPT         DNAME
DNAME_UKEY       DEPT         LOC
LOC_CHECK1       DEPT         LOC
MGR_FKEY         EMP          MGR
SYS_C00275       DEPT         DEPTNO
```

```
SYS_C00278      EMP         ENAME
SYS_C00279      EMP         DEPTNO
SYS_C00280      EMP         EMPNO
```

10

プロシージャとパッケージの使用

この章では、Oracle のプロシージャ機能について説明します。トピックは次のとおりです。

- PL/SQL プロシージャとパッケージ
- PL/SQL パッケージ
- リモート依存性
- カーソル変数
- PL/SQL コードを隠す
- エラー処理
- ストアド・プロシージャの呼出し
- SQL 式からストアド関数をコールする
- 提供されるパッケージ
- ストアド・プロシージャを記述する
- プロシージャとパッケージに関する情報をリストする
- DBMS_ROWID パッケージ
- UTL_HTTP パッケージ

注意： Trusted Oracle を使用している場合、Trusted Oracle のマニュアルも参照して詳しい情報を得てください。

PL/SQL プロシージャとパッケージ

PL/SQL は、最新のブロック構造化プログラミング言語です。この言語が持っている多くの機能を使えば、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。例を上げると、PL/SQL は、標準 SQL にはないプロシージャ要素、たとえばループ文や条件文などを備えています。

PL/SQL ブロック内部で SQL データ操作言語（DML）文を直接発行できます。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ます。したがって、PL/SQL を使用すれば、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、保守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低減できます。

注意： Oracle Forms などの一部の Oracle Tools には、PL/SQL エンジンが組み込まれていて、PL/SQL をローカルで実行できます。

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 や 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、次のような種類があります。

- 無名の PL/SQL ブロック
- トリガー
- スタンドアロン・ストアド・プロシージャおよび関数
- ストアド・プロシージャおよび関数を含むことができるパッケージ

関連項目： PL/SQL 言語の完全な説明は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

無名ブロック

無名の PL/SQL ブロックは、オプションの宣言部分および実行可能部分、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ています。

PL/SQL 変数、例外、およびカーソルを宣言するには、宣言部分を使用します。実行可能部分には PL/SQL コードおよび SQL 文が含まれていて、ネストされたブロックを含むことができます。例外ハンドラには、事前定義された PL/SQL 例外（NO_DATA_FOUND または ZERO_DIVIDE）として、またはユーザーが定義する例外として、例外状況が発生するとコールされるコードが含まれています。

次の PL/SQL 無名ブロックの簡単な例では、DBMS_OUTPUT パッケージ (12-21 ページを参照) を使用して、EMP 表の部門 20 に所属する全従業員の名前を表示します。

```
DECLARE
```

```
    emp_name    VARCHAR2(10);
    CURSOR      c1 IS SELECT ename FROM emp
                       WHERE deptno = 20;
BEGIN
    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実行する場合は、必ず SET SERVEROUTPUT ON コマンドを発行して、PUT_LINE などの DBMS_OUTPUT プロシージャを使用する出力がアクティブになるようにしてください。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。

例外を使用すれば、PL/SQL プログラム論理内の 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、サーバーがクライアント・アプリケーションを異常終了させる可能性があるエラーを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（これが処理されないと ORA-01403 エラーが発生する）を処理します。

```
DECLARE
    emp_number   INTEGER := 9999;
    emp_name     VARCHAR2(10);
BEGIN
    SELECT ename INTO emp_name FROM emp
        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、それをブロックの宣言部分で宣言し、ブロックの例外部分で定義できます。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
            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;
```

関連項目： 例外処理の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

無名ブロックは、SQL*Plus などのツールから対話形式で使用されるか、またはプリコンパイラ、OCI、SQL*Module アプリケーションで最も頻繁に使用されます。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用されます

関連項目： カーソル変数の詳細は、10-24 ページを参照してください。

データベース・トリガー

データベース・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。詳細は、このマニュアルの第 13 章の「データベース・トリガーの使用」を参照してください。

ストアド・プロシージャとストアド関数

ストアド・プロシージャまたは関数は、次のような特徴を持つ PL/SQL プログラム・ユニットです。

- 固有の名前を持つ。
- パラメータをとり、値を戻すことができる。
- データ・ディクショナリに格納される。
- 多数のユーザーが起動できる。

注意： このマニュアルでは、ストアド・プロシージャという用語は、包括的な意味で使用している場合があり、その場合にはストアド・プロシージャとストアド関数の両方をカバーしています。

プロシージャ名

プロシージャはデータベースに格納されるので、他のストアド・プロシージャと区別し、アプリケーションがそのプロシージャをコールできるように、名前を付ける必要があります。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は、一意の名前を持っていなければなりません。名前は、正当な PL/SQL 識別子でなければなりません。

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（たとえば、Ada または C）で正当な識別子でなければなりません。

パッケージの一部であるプロシージャ名と関数名は、オーバーロードできます。つまり、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、別のサブプログラムに対して同じ名前を使用できます。サブプログラム名のオーバーロードの詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

プロシージャのパラメータ

ストアド・プロシージャおよび関数には、パラメータを指定できます。次の例は、10-2 ページの無名ブロックに類似したストアド・プロシージャを示しています。

```
PROCEDURE get_emp_names (dept_num IN NUMBER) IS
    emp_name       VARCHAR2(10);
    CURSOR         c1 (depno NUMBER) IS
                      SELECT ename FROM emp
                        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、パラメータ付きカーソル 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　　パラメータの名前です。これは、正当な PL/SQL 識別子でなければなりません。

モード	パラメータのモードで、パラメータが入力だけのパラメータ（IN）、出力だけのパラメータ（OUT）、入力と出力の両方のパラメータ（IN OUT）のどれであるかを示します。モードを指定しないと、IN が想定されます。
データ型	パラメータのデータ型は、標準 PL/SQL データ型です。

パラメータ・モード

パラメータ・モードを使用して、仮パラメータの動作を定義できます。3 つのパラメータ・モード、IN（デフォルト）、OUT、および IN OUT は、どん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 モードと IN OUT モードは関数とともに使用しないでください。関数の目的は、0 個以上の引数をとって、1 つの値を戻すことです。関数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次作用の影響を、関数が受けることがあってはなりません。

表 10-1 には、パラメータ・モードが要約されています。パラメータ・モードの詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

表 10-1　パラメータ・モード

IN	OUT	IN OUT
デフォルト	指定しなければなりません。	指定しなければなりません。
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働きます。	仮パラメータが未初期化変数として働きます。	仮パラメータが初期化変数として働きます。
仮パラメータに値を割り当てることができません。	仮パラメータを式の中で使用できません。値を割り当てなければなりません。	仮パラメータに値を割り当てなければなりません。
実パラメータを、定数または初期化変数、リテラル、式にできます。	実パラメータは変数でなければなりません。	実パラメータは変数でなければなりません。

パラメータのデータ型

仮パラメータのデータ型は、次のいずれかで構成されています。

- NUMBER や VARCHAR2 などの無制約型名
- %TYPE または %ROWTYPE 属性を使用して制約される型

注意： NUMBER(2) や VARCHAR2(20) などの数値が制約される型は、パラメータ・リストでは許可されません。

%TYPE 属性および %ROWTYPE 属性 ただし、型属性 %TYPE と %ROWTYPE を使用して、パラメータを制約できます。たとえば、10-5 ページの「プロシージャのパラメータ」の GET_EMP_NAMES プロシージャの仕様部は、次のように記述します。

```
PROCEDURE get_emp_names(dept_num IN emp.deptno%TYPE)
```

これによって、DEPT_NUM パラメータに EMP 表の DEPTNO 列と同じデータ型をとることができます。%TYPE（または %ROWTYPE）を使用した宣言を作成する場合は、列と表が使用可能でなければなりません。

表の列のタイプが変更されても、アプリケーション・コードを変更する必要はないので、%TYPE を使用することをお薦めします。

GET_EMP_NAMES プロシージャがパッケージの一部である場合は、前に宣言したパブリック（パッケージ）変数を使用して、パラメータのデータ型を制約できます。たとえば、次のとおりです。

```
dept_number    number(2);
...
PROCEDURE get_emp_names(dept_num IN dept_number%TYPE);
```

%ROWTYPE 属性を使用して、指定された表のすべての列を制約するレコードを作成します。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 PL/SQL レコード内の EMP 表の列をすべて戻します。

```
PROCEDURE get_emp_rec (emp_number IN emp.empno%TYPE,
                       emp_ret OUT emp%ROWTYPE) IS
BEGIN
    SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno
        INTO emp_ret
        FROM emp
        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%ROWTYPE;     -- declare a record matching a
                                -- row in the EMP table
BEGIN
  get_emp_rec(7499, emp_row);   -- call for emp# 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;
```

ストアド関数は、%ROWTYPE を使用して宣言される値も戻すことができます。たとえば、次のとおりです。

```
FUNCTION get_emp_rec (dept_num IN emp.deptno%TYPE)
    RETURN emp%ROWTYPE IS ...
```

表とレコード

PL/SQL 表を、パラメータとしてストアド・プロシージャや関数に渡せます。レコードの表も、パラメータとして渡せます。

デフォルト・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。DEFAULT キーワードまたは代入演算子を使用して、パラメータにデフォルト値を指定できます。たとえば、10-5 ページの GET_EMP_NAMES プロシージャの仕様部は、次のように使用します。

```
PROCEDURE get_emp_names (dept_num IN NUMBER DEFAULT 20) IS ...
```

また、次のようにも記述できます。

```
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、デフォルト値が上書きされます。

DECLARE キーワード

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、変数およびカーソル、例外の宣言の前にキーワード DECLARE を使用しません。実際には、それを使用するとエラーが発生します。

ストアド・プロシージャと関数の作成

プロシージャを作成するには、通常のテキスト・エディタを使用します。プロシージャの先頭に、次のコマンドを記述します。

```
CREATE PROCEDURE procedure_name AS ...
```

たとえば、10-7 ページの例を使用する場合は、次のコードを含む get_emp.sql という名前のテキスト（ソース）ファイルを作成できます。

```
CREATE PROCEDURE get_emp_rec (emp_number  IN emp.empno%TYPE,
                              emp_ret    OUT emp%ROWTYPE) AS
BEGIN
```

```
    SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno
        INTO emp_ret
        FROM emp
        WHERE empno = emp_number;
END;
/
```

その後、SQL*Plus などの対話形式のツールを使用して、次のコマンドを入力して、プロシージャを含むテキスト・ファイルをロードします。

```
SQL> @get_emp
```

プロシージャが、get_emp.sql ファイル (.sql はデフォルトのファイル拡張子です) 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なく、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。

警告： **新しいプロシージャを作成する場合、通常は CREATE OR REPLACE...PROCEDURE コマンドを使用した方が便利です。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。

関数を格納するには、CREATE [OR REPLACE] FUNCTION... コマンドを使用します。CREATE PROCEDURE コマンドと CREATE FUNCTION コマンドの完全な構文については、『Oracle8 Server SQL リファレンス』を参照してください。

プロシージャと関数を作成するために必要な権限

スタンドアロン・プロシージャまたは関数、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。

- 自分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 PROCEDURE システム権限が必要である。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ある。

注意： エラーを発生させずにプロシージャまたはパッケージを作成する、つまり、正しくコンパイルするには、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対するオブジェクト権限を持っていなければなりません。所有者はロールを通じて必要な権限を得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わったら、実行前にそのプロシージャを再認証しなければなりません。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下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。これにより、データベース・アプリケーションと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方針が可能になります。なお、すべてのプロシージャとパッケージが（SYSTEM 表領域内の）データ・ディクショナリに 格納されます。プロシージャや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制御されません。

ストアド・プロシージャおよび関数を変更する

ストアド・プロシージャまたはストアド関数を変更するには、まず DROP PROCEDURE コマンドまたは DROP FUNCTION コマンドを使用して削除（DROP）してから、CREATE PROCEDURE コマンドまたは CREATE FUNCTION コマンドを使用して再作成しなければなりません。あるいは、CREATE OR REPLACE PROCEDURE コマンドまたは CREATE OR REPLACE FUNCTION コマンドを使用します。このコマンドは、プロシージャ、または関数が存在すると、まずそれを削除してから、指定されたとおりに再作成します。

警告：　プロシージャまたは関数は警告なく削除されます。

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。

関連項目：　外部プロシージャの詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

PL/SQL パッケージ

パッケージは、PL/SQL タイプおよびオブジェクト、ストアド・プロシージャ、ストアド関数のグループです。パッケージの仕様部は、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・タイプおよび変数、定数、サブプログラムを宣言します。パッケージの本体は、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・オブジェクトだけではなく、仕様部で宣言されているオブジェクトも定義します。

次の例は、EMPLOYEE_MANAGEMENT という名前のパッケージのパッケージ仕様部を示しています。パッケージには、1 つのストアド関数と 2 つのストアド・プロシージャが含まれています。

```
CREATE PACKAGE employee_management AS
   FUNCTION hire_emp (name VARCHAR2, job VARCHAR2,
      mgr NUMBER, hiredate DATE, sal NUMBER, comm NUMBER,
      deptno NUMBER) RETURN NUMBER;
```

```
   PROCEDURE fire_emp (emp_id NUMBER);
   PROCEDURE sal_raise (emp_id NUMBER, sal_incr NUMBER);
END employee_management;
```

このパッケージの本体は、関数と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

-- The 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.

       new_empno    NUMBER(10);

   BEGIN
      SELECT emp_sequence.NEXTVAL INTO new_empno FROM dual;
      INSERT INTO emp VALUES (new_empno, name, job, mgr,
         hiredate, sal, comm, deptno);
      RETURN (new_empno);
   END hire_emp;

   PROCEDURE fire_emp(emp_id IN NUMBER) AS

-- The procedure deletes the employee with an employee
-- number that corresponds to the argument EMP_ID.  If
-- no employee is found, an exception is raised.

   BEGIN
      DELETE FROM emp 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

-- The 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.
   BEGIN
```

```
-- If employee exists, update salary with increase.
      UPDATE emp
         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には、まず順序番号 EMP_SEQUENCE を作成します。次の SQL*Plus 文を使用して実行できます。

```
SQL> EXECUTE CREATE SEQUENCE emp_sequence
   > START WITH 8000 INCREMENT BY 10;
```

パッケージを作成する

パッケージの各部は、異なるコマンドを使用して作成します。パッケージ仕様部は、CREATE PACKAGE コマンドを使用して作成します。CREATE PACKAGE コマンドでパブリック・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 コマンドを使用します。CREATE PACKAGE BODY コマンドは、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関数の手続き型コードを定義します。パッケージ本体にプライベート（またはローカル）・パッケージ・プロシージャや変数などを定義することもできます。10-13 ページの「ローカル・オブジェクト」を参照してください。

OR REPLACE 句

初めてアプリケーションを開発する際に、CREATE PACKAGE コマンドまたは CREATE PACKAGE BODY コマンドに 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 パッケージ本体を削除することです。このとき、CREATE コマンド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 ...
```

パッケージを作成するために必要な権限

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関数の作成に必要な権限と同じです。10-9 ページを参照してください。

パッケージ・オブジェクトを作成する

パッケージの本体には、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
- パッケージ仕様部に宣言されている関数
- 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関数
- 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および関数、カーソル、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実行許可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、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関数の場合は、戻り型だけではなく、パラメータの名前とデータ型も一致しなければなりません。

ローカル・オブジェクト

ローカル変数、ローカル・プロシージャ、ローカル関数は、パッケージ本体に定義できます。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関数でだけアクセスできます。どんな権限を持っていても、外部ユーザーは参照できません。

パッケージとパッケージ・オブジェクトを命名する

パッケージと、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なければなりません。パッケージ仕様部とその本体は、同じ名前を持っていなければなりません。また、パッケージ要素の名前は、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なければなりません。

パッケージとプロシージャを削除する

SQL コマンド DROP PROCEDURE、DROP FUNCTION、DROP PACKAGE BODY、DROP PACKAGE を使用して、スタンドアロン・プロシージャ、スタンドアロン関数、パッケージ本体、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。

次の例によって、自分のスキーマ内の OLD_SAL_RAISE プロシージャが削除されます。

```
DROP PROCEDURE old_sal_raise;
```

プロシージャとパッケージを削除するために必要な権限

プロシージャまたはパッケージを削除するには、それらが使用しているスキーマ内にあるか、または DROP ANY PROCEDURE 権限を持っていなければなりません。パッケージ内の個々のプロシージャは削除できません。これらは削除せずに、パッケージ仕様部と本体を再作成する必要があります。

パッケージの無効性とセッションの状態

パッケージ・オブジェクトを参照する各セッションは、パブリック変数およびプライベート変数、カーソル、定数に対する持続状態を含め、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。セッションの依存したパッケージ（仕様部または本体）が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への依存（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 と 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 のインスタンシエーションが失われます。このような状況で、セッションが無効なパッケージ依存のオブジェクトを使用しようとすると、1 度目は次のエラーを受け取ります。

```
ORA-04068: パッケージ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、上の例では、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を受け取りますが、P1 をコールすると、このメッセージは受け取りません。

多くの本稼動環境では、無効にする原因となり得る DDL 操作は、通常業務時間外に行われます。したがって、エンド・ユーザー・アプリケーションでは、このような状況は問題にならない可能性もあります。ただし、パッケージの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。たとえば、ユーザー側のアプリケーションで、セッションのパッケージ状態（失われた状態）に依存するユーザー側の状態を再初期化し、パッケージ・コールを再発行できます。

リモート依存性

PL/SQL ライブラリ・ユニット（パッケージおよびストアド・プロシージャ、ストアド関数）間の依存性は、次の 2 つの方法で処理されます。

- タイムスタンプ

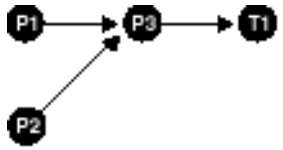
- 署名

タイムスタンプ

PL/SQL ライブラリ・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、ライブラリ・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。

各ライブラリ・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、サーバーによりタイムスタンプが設定されています。図 10-1 に、これを図示します。プロシージャ P1 と 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 は表 T1 を参照します。この例では、各プロシージャはいずれも、表 T1 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 10-1　依存性の関係



P1 と 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 と P2 に無効のマークが即時に付けられます。P1 と 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 と P2 のコンパイル中に P3 に対して記録された値と一致しなくなります。

P1 と 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・モデルの欠点

この依存性モデルの欠点は、必要以上に制限的であるという点です。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稼働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・モデルは問題を提示できます。まず初めに、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、その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しなければなりません。しかし、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稼動できません。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しなければなりません。

署名

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 では署名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。署名機能は、リモート依存性にだけ影響します 。ローカル（同一サーバー）依存性には影響はありません。この環境では、再コンパイルは、常に可能なためです。

サブプログラムの署名には、次のものについての情報が含まれています。

- サブプログラムの名前
- サブプログラムのパラメータの基本タイプ
- パラメータのモード（IN、OUT、IN OUT）

注意： 重要なのはパラメータのタイプとモードだけです。パラメータの名前は署名に影響しません。

ユーザーは、署名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。詳細は、10-22 ページの「リモート依存性を制御する」を参照してください。署名依存性モデルが有効であれば、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署名が矛盾する方法で変更された場合には、リモート・ライブラリ・ユニットでの依存性によって、その依存ユニットは無効化されます。

たとえば、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。

```
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きに、その署名は、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は、次のように、ある PL/SQL コードは、BOSTON_SERVER と呼ばれる DB リンクを使用してそのサーバーを識別する GET_EMP_NAME をコールすると想定します。

```
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(hiredate);
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る
- GET_EMP_NAME の署名がカリフォルニアのサーバーに転送される
- 署名は、PRINT_ENAME のコンパイルされた状態で記録される

実行時には、変更の有無に関係なく、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 の記録済みの署名が、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、コールした側のプロシージャにエラー・ステータスが戻されます。

しかし、署名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 のコンパイル済み状態にある GET_EMP_NAME の記録済み署名が、ボストンのサーバーの GET_EMP_NAME の現行署名と比較されます。2 つの署名が一致した場合は、コールが成功します。2 つの署名が一致しない場合には、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、たとえば、新しいリリースでサーバーが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。署名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。

署名とはなにか

署名は、各コンパイル済みのストアド・ライブラリ・ユニットと関連付けられます。署名は、次の基準によりユニットを識別します。

- ユニットの名前。つまり、パッケージまたはプロシージャ、関数の名前
- サブプログラムの各パラメータのタイプ
- パラメータのモード
- パラメータの数
- 関数のための戻り値のタイプ

署名はいつ変更されるか

データ型 署名は、あるクラスのデータ型を別のクラスに変更した場合に変更されます。各データ型クラス内には、複数のタイプ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タイプに変更しても、署名が変更されることはありません。

表 10-2 に型のクラスを示します。

表 10-2　データ型のクラス

Varchar のタイプ :	**数値のタイプ :**
VARCHAR2	NUMBER
VARCHAR	INTEGER
STRING	INT
LONG	SMALLINT
ROWID	DECIMAL
	DEC
文字のタイプ :	REAL
CHARACTER	FLOAT
CHAR	NUMERIC

表 10-2　データ型のクラス

	DOUBLE PRECISION
Raw のタイプ :	NUMERIC
RAW	
LONG RAW	
整数のタイプ :	**日付のタイプ :**
BINARY_INTEGER	DATE
PLS_INTEGER	
BOOLEAN	**MLS ラベルのタイプ :**
NATURAL	MLSLABEL
POSITIVE	
POSITIVEN	
NATURALN	

モード デフォルトのパラメータ・モード IN の明示的指定との間で変更があっても、サブプログラムの署名は変更されません。たとえば、次のような変更があったとします。

```
PROCEDURE P1 (param1 NUMBER);
```

これを、次のように変更します。

```
PROCEDURE P1 (param1 IN NUMBER);
```

しかし、これによって署名は変更されません。これ以外のパラメータ・モードの変更が行われた場合は、署名は変更されます。

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、署名が変更されることはありません。たとえば、プロシージャ P1 は、次の 2 つの例では同じ署名を持っています。

```
PROCEDURE P1 (param1 IN NUMBER := 100);
PROCEDURE P1 (param1 IN NUMBER := 200);
```

コール側が新しいデフォルト値を取得することを必要とするアプリケーションの開発者は、コールされたプロシージャを再コンパイルしなければなりませんが、デフォルトのパラメータ値の割当てが変更されたときに、署名に基づいて無効化されることはありません。

署名の例

10-5 ページで定義されている GET_EMP_NAME プロシージャで、プロシージャ本体が次のように変更された場合、

```
BEGIN
-- date format model changes
    SELECT ename, to_char(hiredate, 'DD/MON/YYYY')
        INTO emp_name, hire_date
        FROM emp
        WHERE empno = emp_number;
END;
```

プロシージャの指定は変更されていないため、その署名も変更されません。

しかし、プロシージャ指定が次のように変更された場合、

```
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、署名は変更されます。これは、パラメータ HIRE_DATE が別のデータ型を持っているためです。

しかし、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、署名は変更されません。仮パラメータの名前が変更されても、そのユニットの署名は変更されません。

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
        WHERE empno = emp_data.emp_number;
END;
```

レコードのフィールド名は変更されても、タイプはそのままであるようにパッケージ指定が変更された場合、これは署名に影響しません。たとえば、次のパッケージ指定は、以前のパッケージ指定の例と同じ署名を持っています。

```
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;
```

タイプが以前のままである場合には、パラメータのタイプの名前を変更しても、署名は変更されません。たとえば、次のような EMP_PACKAGE のパッケージ仕様部は、10-21 ページの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;
```

リモート依存性を制御する

タイムスタンプまたは署名の依存性モデルが有効であるか否かは、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場合、

  ```
  REMOTE_DEPENDENCIES_MODE = TIMESTAMP
  ```

  これが動的に明示的に上書きされないと、タイムスタンプだけを使用して、依存性が解決されます。

- 初期化パラメータ・ファイルに、パラメータ指定が含まれている場合、

  ```
  REMOTE_DEPENDENCIES_MODE = SIGNATURE
  ```

  これが動的に明示的に上書きされない場合、署名を使用して、依存性が解決されます。

- モードは、DDL コマンドを使用することによって動的に変更できます。

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

  上記は、現行セッションの依存性モデルを変更するためのコマンドです。または、

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

  上記は、起動後にシステムで依存性モデルを変更するためのコマンドです。

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 コマンドまたは ALTER SYSTEM DDL コマンドを使用して指定されていない場合には、TIMESTAMP がデフォルト値です。そのため、明示的に REMOTE_DEPENDENCIES_MODE パラメータまたは適切な DDL コマンドを使用しない限り、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ています。

REMOTE_DEPENDENCIES_MODE=SIGNATURE を使用するときには、次の点に注意しなければなりません。

- 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化されない。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たい場合は、コール側プロシージャを手動で再コンパイルしなければなりません。

- 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ない。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コールが TIMESTAMP モードで再バインドされると判明した場合、ローカル・プロシージャは無効化されないため、SIGNATURE モードではこの再バインドは行われません。新規の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しなければなりません。

- 新しいタイプが古いタイプ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タイプが変更された場合は、ローカルのコール側プロシージャが自動的に無効化 / 再コンパイルされることはない。新しいタイプの意味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しなければなりません。

依存性の解決度

REMOTE_DEPENDENCIES_MODE = TIMESTAMP（デフォルト値）の場合、ライブラリ・ユニット間の依存性は、実行時にタイムスタンプと比較して処理されます。コールされたリモート・プロシージャのタイムスタンプがコールされたプロシージャのタイムスタンプと一致しないと、コール側の（依存）ユニットは無効にされるため、再コンパイルしなければなりません。この場合、ローカル PL/SQL コンパイラがなければ、コール側のアプリケーションは処理できません。

タイムスタンプ依存性モードでは、署名は比較されません。ローカル PL/SQL コンパイラがある場合には、コール側プロシージャが実行されると自動的に再コンパイルされます。

REMOTE_DEPENDENCIES_MODE = SIGNATURE の場合、コール側のユニット内に記録されたタイムスタンプは、まず最初に、コールされたリモート・ユニット内の現在のタイムスタンプと比較されます 。それらのタイムスタンプが一致した場合は、コールは正常に進行します 。タイムスタンプが一致しなかった場合は、コールされたリモート・サブプログラムの署名は、コール側サブプログラムに記録されたままの状態で、コールされたサブプログラムの現在の署名と比較されます。それらが一致しない場合は、10-18 ページの「署名とはなにか」の項に示されている基準により、コール側セッションにエラーが戻されます。

依存性を管理するための提案

Oracle では、REMOTE_DEPENDENCIES_MODE パラメータを設定するために次のガイドラインに従うことをお薦めします。

- サーバー側の PL/SQL ユーザーは、このパラメータを TIMESTAMP に設定して（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る。
- サーバー側の PL/SQL ユーザーは、分散システムを備えており、考え得る不要な再コンパイルを回避する場合には、署名依存性モードの使用を選択できる。
- クライアント側の PL/SQL ユーザーは、このパラメータを SIGNATURE に設定しなければならない。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る。
  - 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る。

- サーバー側で SIGNATURE モードを使用するときには、必ず、パッケージ仕様部内のプロシージャ（または関数）宣言の終わりに新しいプロシージャを追加すること。新しいプロシージャを宣言リストの中間に追加すると、依存プロシージャが不必要に無効化され、再コンパイルされる可能性があります。

カーソル変数

カーソル変数は、カーソルへの参照です。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。カーソル変数はポインタであるため、プロシージャおよび関数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（示す）こともできます。

上記以外のカーソル変数の利点として、次のことが挙げられます。

- カプセル化 : カーソル変数をオープンするストアド・プロシージャでは、問合せが集中化される。
- メンテナンスの容易性 : カーソルを変更しなければならない場合に、1 つの場所、つまりストアド・プロシージャを変更するだけで済む。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、アプリケーションをサーバーに接続するときに使用するユーザー名で認識される。ユーザーは、カーソルをオープンするストアド・プロシージャに対する実行許可を持っていなければなりません。しかし、ユーザーは、問合せで使用される表に対する読込み許可を持っている必要はありません。この機能は、表の列へのアクセスと、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。

カーソル変数の完全な説明は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

カーソル変数を宣言してオープンする

適切な ALLOCATE コマンドを使用して、クライアント・アプリケーションのカーソル変数にメモリーを割り当てなければなりません。Pro*C では、EXEC SQL ALLOCATE <cursor_name> コマンドを使用します。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・セッションだけで実行するアプリケーションで、カーソル変数を使用できます。PL/SQL サブプログラムでカーソル変数を宣言してオープンし、他の PL/SQL サブプログラムのパラメータとして使用できます。

カーソル変数の例

この項には、PL/SQL でのカーソル変数の使用例がいくつか示されています。プログラム・インタフェースを使用するカーソル変数の例がさらに必要な場合は、次のマニュアルを参照してください。

- 『Pro*C/C++ プリコンパイラ・プログラマーズ・ガイド』

- 『Oracle プリコンパイラ・プログラマーズ・ガイド』
- 『Oracle コール・インタフェース・プログラマーズ・ガイド』
- 『SQL*Module ユーザーズ・ガイドおよびリファレンス』

データをフェッチする

次のパッケージは PL/SQL カーソル変数タイプ EMP_VAL_CV_TYPE と、2 つのプロシージャを定義します。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 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%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%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 WHERE deptno = dept_number;
  END open_emp_cv;

  PROCEDURE fetch_emp_data (emp_cv       IN emp_val_cv_type,
                            emp_row     OUT emp%ROWTYPE) IS
  BEGIN
    FETCH emp_cv INTO emp_row;
  END fetch_emp_data;
END emp_data;
```

次の例では、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ます。

```
DECLARE
-- declare a cursor variable
  emp_curs emp_data.emp_val_cv_type;

  dept_number dept.deptno%TYPE;
  emp_row emp%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;
```

バリアント・レコードをインプリメントする

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、判別子を使用してカーソル変数をオープンし、2 つの異なるカーソルのうちの 1 つを指します。

```
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 WHERE sal > 2000;
    ELSIF discrim = 2 THEN
      OPEN cv FOR SELECT * FROM dept;
    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 表または DEPT 表に関する問合せを示すことができます。これはどのようにしたらよいでしょうか。次の PL/SQL ブロックは、カーソル変数を使用して対象情報である各表をフェッチし、次に ROWTYPE_MISMATCH 事前定義済み例外を使用して各フェッチ・レコードを処理できることを示しています。

```
DECLARE
  emp_rec  emp%ROWTYPE;
  dept_rec dept%ROWTYPE;
  cv       emp_dept_data.cv_type;

BEGIN
  emp_dept_data.open_cv(cv, 1); -- open CV for EMP fetch
  FETCH cv INTO dept_rec;       -- but fetch into DEPT 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 data...');
      FETCH cv into emp_rec;
      DBMS_OUTPUT.PUT(emp_rec.deptno);
      DBMS_OUTPUT.PUT_LINE('  ' || emp_rec.ename);
    END;
END;
```

PL/SQL コードを隠す

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 コマンドを入力します。

```
WRAP INAME=input_file [ONAME=ouput_file]
```

> **関連項目：** PL/SQL ラッパーの使用に関する完全な説明は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

エラー処理

Oracle では、ユーザー定義エラーの番号と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と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 プロシージャによって戻されます。

```
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終了し、プロシージャの影響をロールバックして、ユーザー定義エラー番号とメッセージを戻します（例外ハンドラによってエラーが検出されない場合）。ERROR_NUMBER は、-20000 から -20999 の範囲内になければなりません。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すべきです。TEXT は、2KB 以下の文字式でなければなりません （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、既存のエラーと置き換える場合は FALSE にできます。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 から -20005 まで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 PL/SQL コードのロジックに多用されます。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.name = "NLS_LANGUAGE";
   raise_application_error(-20101, message);
...
```

前のいくつかの例で、RAISE_APPLICATION_ERROR プロシージャの使用について説明しました。また、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引き渡す例を示します。リモート・プロシージャをコールするときの例外処理の詳細は、10-31 ページの「リモート・プロシージャでエラーを処理する」を参照してください。

例外と例外処理ルーチンを宣言する

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。例外が発生すると（通知されると）、通常の PL/SQL ブロックの実行は中断され、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・コードで条件を検査できますが、IF 文の使用には特に注意が必要です。エラー条件がある場合は、次の 2 つのオプションを選択できます。

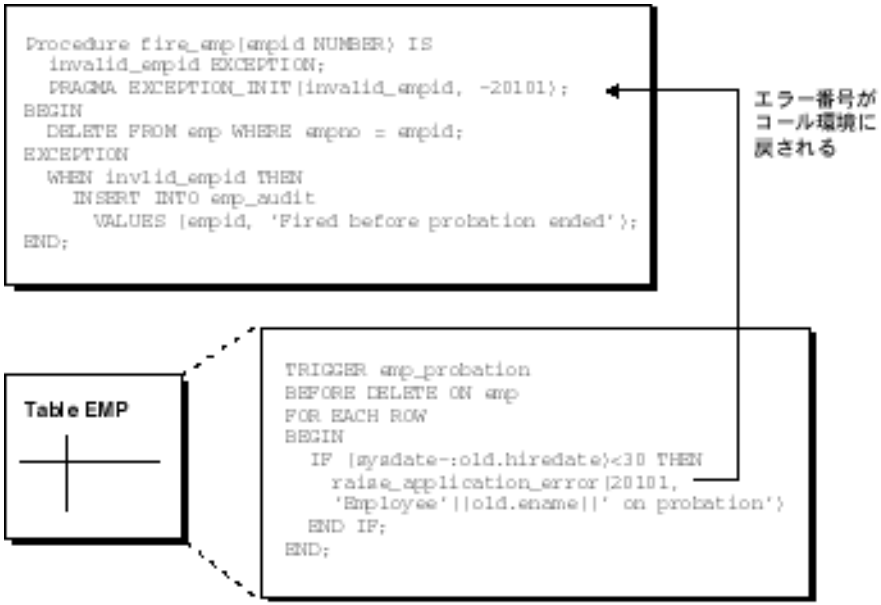
- 一定の例外を指示する RAISE 文を発行する。RAISE 文によってプロシージャの実行は中断され、例外ハンドラがあれば制御が渡されます。
- RAISE_APPLICATION_ERROR プロシージャをコールして、ユーザー定義エラーの番号とメッセージを戻せる。

ユーザー定義エラー・メッセージを処理するために、例外ハンドラも定義できます。たとえば、図 10–2 では、次の事項を説明しています。

- プロシージャ中の例外と例外ハンドラ
- エラーを検査する条件文（無効データの転送など）と、ユーザー定義エラーの番号とメッセージをトリガーに出力
- ユーザー定義エラーの番号とメッセージをコールした環境（この場合はプロシージャ）に戻す方法と、アプリケーションに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）、パッケージの仕様部で宣言します（パブリック例外）。例外ハンドラは、プロシージャの本体（スタンドアロンまたはパッケージ）に定義します。

図 10-2　例外と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。そのプログラム・ユニット中の、処理されない例外ポイントの前に COMMIT 文が含まれている場合、プログラム・ユニットの暗黙的なロールバックは、直前のコミットに戻る場合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。そのようなアプリケーションでは、その例外はデータベースに SQL 文として送信されるため、アプリケーション・プログラム・ユニット・コールだけ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 コードを変更しなければなりません。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。詳細は、10-31 ページの「リモート・プロシージャでエラーを処理する」を参照してください。

分散問合せでエラーを処理する

トリガーやストアド・プロシージャを使用して、分散問合せを作成できます。この分散問合せは、ローカル Oracle によって、対応する数のリモート問合せに分解され、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返します。その後、ローカル・ノードは、必要な後処理を行い、ユーザーやアプリケーションに結果を戻します。

たとえば、整合性制約の違反のために、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。後続する文やプロシージャ・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実行されるまで、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・メッセージを検査するように、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進める前に、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・プロシージャでエラーを処理する

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 つのタイプの例外が発生する可能性があります。

- キーワード EXCEPTION を使用して宣言しなければならない、PL/SQL のユーザー定義例外
- NO_DATA_FOUND などの PL/SQL 事前定義済み例外
- ORA-00900 と 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とき、これら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
    /* ... 例外の処理 */
```

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、プラグマ 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さなければなりません。先の例に示したように、ローカル・プロシージャが例外を処理します。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。その他のリモート例外は、すべてローカル例外と同じ方法で処理できます。

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を通知されますが、エラーの種類を直接示すものは提示されません。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 を次のように送信したとします。

```
SVRMGR> @proc1
```

そして、コードに 1 つまたは複数のエラーが存在すると、次のような通知を受け取ります。

```
MGR-00072: Warning: Procedure PROC1 created with compilation errors
```

この場合、SQL*Plus で SHOW ERRORS コマンドを使用して、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、一番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、プロシージャまたは関数、パッケージ、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
```

SHOW ERRORS コマンドの完全な説明は、『SQL*Plus User's Guide and Reference』を参照してください。

注意： SHOW ERRORS コマンドを発行する前に、SET CHARWIDTH コマンドを使用して、長い行が出力されるようにしてください。通常は、値に次のように 132 を指定することをお薦めします。

```
SET CHARWIDTH 132
```

たとえば、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するとします。

```
CREATE PROCEDURE fire_emp(emp_id NUMBER) AS
   BEGIN
      DELETE FROM emp WHER empno = emp_id;
   END
/
```

上記の 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 文がエラーとなると、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 コマンドによって、エラーが発生した行または列の番号が表示されます。

他の Tool や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・ビューを利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 の各ビューを使えば、データ・ディクショナリからオリジナルのソース・コードを取り出せます。

> **関連項目：** これらのデータ・ディクショナリ・ビューの詳細は、『Oracle8 Server リファレンス・マニュアル』を参照してください。

デバッグ

ストアド・プロシージャおよびトリガーは、DBMS_OUTPUT 提供のパッケージを使用してデバッグできます。コードには PUT と PUT_LINE 文を入れて、変数および式の値を端末に出力します。DBMS_OUTPUT パッケージの詳細は、12-20 ページの「ストアド・プロシージャとトリガーからの出力」を参照してください。

プラットフォームが Oracle Procedure Builder(Oracle Developer/2000 のツール・セットの一部) をサポートしていれば、それを使用することにより、さらに簡単にデバッグできます。

Procedure Builder を使用すると、PL/SQL プロシージャおよびトリガーを、制御されたデバッグ環境で実行し、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。詳細は、『Oracle Procedure Builder 開発者ガイド』を参照してください。

ストアド・プロシージャの呼出し

プロシージャは、様々な環境から呼び出すことができます。たとえば、次のとおりです。

- 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る。
- プロシージャは、Oracle Tool（たとえば、SQL*Plus）を使用して、ユーザーが対話式でコールできる。
- プロシージャは、アプリケーション（たとえば、SQL*Forms またはプリコンパイラ・アプリケーション）内で明示的にコールすることができる。
- ストアド関数は、LENGTH や ROUND などの組込み SQL 関数のコールと似た方法で、SQL 文からコールできる。

以降の項では、これらの環境からのプロシージャの呼出しに共通する例を紹介します。詳細は、10-40 ページの「SQL 式からストアド関数をコールする」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);
. . .
```

この行は SAL_RAISE プロシージャと呼ばれます。このコンテキスト中に EMP_ID 変数を指定できます。なお、PL/SQL 内では再帰プロシージャ・コールが可能です。つまり、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 からプロシージャを対話形式でコールする

SQL*Plus などの Oracle Tool を使用して、プロシージャを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 という名前のプロシージャをコールするために、次のように無名 PL/SQL ブロックを使用できます。

```
BEGIN
    sal_raise(1043, 200);
END;
```

注意： SQL*Plus などの対話形式の Tools を使用する場合は、PL/SQL ブロックを実行するために、これらの行の終わりにスラッシュ (/) を付けなければなりません。

ブロックをより簡単に実行する方法は、SQL*Plus コマンド EXECUTE を使用することです。これにより入力するコードを BEGIN 文と END 文で効率的にまとめることができます。たとえば、次のとおりです。

```
EXECUTE sal_raise(1043, 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。たとえばある関数を実行して、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
```

SQL*Plus については、『SQL*Plus User's Guide and Reference』を参照してください。開発 Tool を使用した同様の操作に関する説明は、使用している使用 Oracle Tools のマニュアルを参照してください。

3GL アプリケーション内でプロシージャをコールする

プリコンパイラまたは OCI アプリケーションなどの 3GL データベース・アプリケーションに、そのアプリケーションのコード内のプロシージャへのコールを記述できます。

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_emp(:empno);
```

:EMPNO は、アプリケーションのコンテキスト中のホスト（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わなければなりません。たとえば次の文は、プリコンパイラ・アプリケーションから FIRE_EMP プロシージャをコールします。

```
EXEC SQL EXECUTE
   BEGIN
      fire_emp(:empno);
   END;
END-EXEC;
```

:EMPNO はホスト（バインド）変数です。

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に関する詳細は、次のマニュアルを参照してください。

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

プロシージャをコールするときの名前変換

プロシージャとパッケージの参照は、4-42 ページの「SQL 文における参照オブジェクトの名前変換」の項で説明されているアルゴリズムに従って変換されます。

プロシージャを実行するために必要な権限

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、前述したように、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、次の条件を満たさなければなりません。

- 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 権限を持っているか、あるいは EXECUTE ANY PROCEDURE システム権限を持っている。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、ロールを介さずに、直接付与されていなければなりません。
- 次のように、所有者の名前をコールに指定する。

  ```
  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定義域で実行されます。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なければなりません。

プロシージャ引数の値を指定する

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の方法を使用するか、あるい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する。
- 引数の名前とその値を、任意の順序で指定する。

たとえば、次の文はいずれも UPDATE_SAL 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 増やします。

```
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、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、上記 2 つの方法を組み合せて引数の値を指定します。引数の名前と順序を組み合せて指定する場合、順序で指定する値は、名前で指定する値よりも前になければなりません。

サブプログラム（『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照）に対する IN パラメータのデフォルト値を定義するために、DEFAULT オプションを使用している場合、異なる数のパラメータをサブプログラムに引き渡して、デフォルト値を受け入れたり、無効にしたりできます。実際に値を引き渡さなければ、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には、その値だけでなく、引数の名前も明示的に指定しなければなりません。

リモート・プロシージャをコールする

リモート・プロシージャは、適切なデータベース・リンクとその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、NY という名前のローカル・データベース・リンクが示すデータベース内のプロシージャ FIRE_EMP を実行します。

```
EXECUTE fire_emp@NY(1043);
```

関連項目： リモート・プロシージャをコールするときの例外処理の詳細は、10-31 ページを参照してください。

リモート・プロシージャ・コールとパラメータ値

デフォルトがある場合でも、すべてのリモート・プロシージャのパラメータに値を明示的に渡さなければなりません。リモート・パッケージ変数と定数にはアクセスできません。

リモート・オブジェクトを参照する

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の本体で参照できます。次のプロシージャではリモート従業員表の行を削除します。

```
CREATE PROCEDURE fire_emp(emp_id NUMBER) IS
BEGIN
    DELETE FROM emp@sales 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次に示します。

- リモート・プロシージャ（スタンドアロン・プロシージャとパッケージ・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、リモート・プロシージャの引数を指定することによって、プロシージャまたは OCI アプリケーション、プリコンパイラ・アプリケーションからコールできる。

  ```
  CREATE PROCEDURE local_procedure(arg1, arg2) AS
  BEGIN
    ...
    remote_procedure@dblink(arg1, arg2);
    ...
  END;
  ```

- 上の例で、REMOTE_PROCEDURE@DBLINK に対してシノニムを作成できる 。これによって、プロシージャまたは OCI アプリケーション、プリコンパイラ・アプリケーションだけでなく、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 PROCEDURE local_procedure(arg1, arg2) AS
  BEGIN
    ...
    synonym(arg1, arg2);
    ...
  END;
  ```

- シノニムを使用しないのであれば、リモート・プロシージャをコールするローカル・プロシージャを作成する方法もある。

  ```
  BEGIN local_procedure(arg1, arg2);
  END;
  ```

  LOCAL_PROCEDURE は、このリストの最初の項目として定義する。

注意： シノニムを使用して、関連するリモート・プロシージャの位置透過性を作成できます。

警告： **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、常にトランザクションは 2 フェーズ・コミットが行われます（リモート・プロシージャが読込み専用の場合でも）。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、COMMIT、ROLLBACK、SAVEPOINT 文を実行できません。

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。リモート・ノードに送信された要素中の文の実行は、正常終了または異常終了します。分散更新の一部がエラーとなり、一部が正常終了した場合、処理を続けるには（トランザクション全体またはセーブポイントまでの）ロールバックが必要です。分散更新を実行するプロシージャを作成する場合は、この点を考慮する必要があります。

リモート・プロシージャのコールにローカル・プロシージャを使用する場合は、特に注意が必要です。ローカル・プロシージャの実行中にタイムスタンプの不一致が見つかると、リモート・プロシージャは実行されず、そのローカル・プロシージャは無効となります。

プロシージャとパッケージのシノニム

スタンドアロン・プロシージャと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と所有者の名前を隠す。
- リモート・ストアド・プロシージャの位置の透過性が得られる（スタンドアロンまたはパッケージに対して）。

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必要がある場合、その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（つまり、パッケージがオブジェクトです）、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 式からストアド関数をコールする

ユーザー作成の PL/SQL 関数を 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 関数を使用すれば、次のことができます。

- SQL の拡張によって、ユーザーの生産性が向上する。実行する内容が SQL 文だけ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。
- 問合せの効率がよくなる。問合せの WHERE 句で関数を使用すると、複雑なロジックも表現でき、非常に容易にデータのフィルタ処理が可能だが、それ以外の場合は、アプリケーション側で判断基準を埋め込む必要があります。
- 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、文字列を操作できる。
- パラレル問合せ実行を提供する。問合せがパラレル化されると、PL/SQL 関数内の SQL 文もパラレルに実行できます（パラレル問合せオプションを使用）。

PL/SQL 関数を使用する

PL/SQL 関数は、トップレベルの関数として作成するか、パッケージ仕様部内部で宣言しなければ、SQL 文内で命名できません。ストアド PL/SQL 関数は、組込み Oracle 関数（たとえば、SUBSTR、ABS など）と同じ方法で使用できます。

PL/SQL 関数は、SQL 文内で Oracle 関数を入れられる場所、つまり、SQL 内で式を入れられる場所であればどこでも入れることができます。たとえば、PL/SQL 関数は、次の場所からコールできます。

- SELECT コマンドの選択リスト
- WHERE 句および HAVING 句の条件
- CONNECT BY 句、START WITH 句、ORDER BY 句、GROUP BY 句
- INSERT コマンドの VALUES 句
- UPDATE コマンドの SET 句

CREATE コマンドまたは ALTER TABLE コマンドの CHECK 制約句からストアド PL/SQL 関数をコールしたり、ストアド PL/SQL 関数を使っ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関数と異なり、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 文から直接コールできません。ただし、PL/SQL 文からコールされる関数や、SQL 式で参照される関数は、PL/SQL プロシージャをコールできます。

構文

SQL から PL/SQL 関数を参照するには、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 という名前の関数を参照するには、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 つだけが指定されている場合は、最初の識別子をスキーマ名またはパッケージ名にできます。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する。
- 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 関数を含んでいる PAYROLL という名前のスキーマを探す。PAYROLL スキーマ内に TAX_RATE 関数が見つからない場合は、エラー・メッセージが戻されます。
- 現行スキーマ内に PAYROLL パッケージが見つからない場合、Oracle は、PAYROLL パッケージ内で TAX_RATE 関数を探す。PAYROLL パッケージ内に TAX_RATE 関数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関数に対して定義したシノニムを使用して、その関数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、パラメータなしの関数の名前より優先します。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(new_sal NUMBER ...);
CREATE FUNCTION new_sal RETURN NUMBER IS ...;
```

そして、次の 2 つの文では、NEW_SAL への参照は、列 EMP.NEW_SAL を参照します。

```
SELECT new_sal FROM emp;
SELECT emp.new_sal FROM emp;
```

NEW_SAL 関数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;
```

例 スキーマ SCOTT から TAX_TABLE 内の SS_NO 列と SAL 列を引数として、TAX_RATE PL/SQL 関数をコールして変数 INCOME_TAX にその結果を入れるには、次のように指定します。

```
SELECT scott.tax_rate (ss_no, sal)
    INTO income_tax
    FROM tax_table
    WHERE ss_no = tax_id;
```

これらの PL/SQL 関数のコール例は、SQL 式として認識されます。

```
circle_area(radius)
payroll.tax_rate(empno)
scott.payroll.tax_rate(dependents, empno)@ny
```

引数

任意の数の引数を 1 つの関数に渡すには、引数をカッコに入れます。この場合、位置表記法を使用しなければなりません。名前表記法は、現在ではサポートされていません。引数の不要な関数の場合は、カッコを省きます。

引数のデータ型と関数の戻りタイプは、SQL がサポートしているタイプに限定されます。たとえば、SQL 文から PL/SQL BINARY_INTEGER を戻す PL/SQL 関数をコールすることはできません。

デフォルト値を使用する

ストアド関数 gross_pay では、次のように DEFAULT 句を使っ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。

```
CREATE FUNCTION gross_pay
    (emp_id IN NUMBER,
    st_hrs IN NUMBER DEFAULT 40,
    ot_hrs IN NUMBER DEFAULT 0) RETURN NUMBER AS
  ...
```

手続き型文から gross_pay をコールするときは、必ず 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、次に示すようにパラメータをスキップする名前表記法を使用できるからです。

```
IF gross_pay(eenum,ot_hrs => otime) > pay_limit THEN ...
```

ただし、SQL 式から gross_pay をコールするときは、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。なぜなら、名前表記法を使用できないからです。

基本的な要件を満たす

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 関数が次の基本的な要件を満たさなければなりません。

- PL/SQL ブロックまたはサブプログラム内に定義された関数ではなく、ストアド関数でなければならない。
- 列（グループ）関数ではなく、行関数でなければならない。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 パラメータでなければならない。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 または DATE、NUMBER など、Oracle Server の内部型でなければならない。BOOLEAN または 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 の内部型でなければならない。

たとえば、次のストアド関数は、これらの基本的な要件を満たしています。

```
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;
```

副次作用を制御する

ストアド関数をコールする SQL 文を実行するには、Oracle Server がその関数の「純粋さレベル」を認識していなければなりません。つまり、純粋さレベルが高いほど、その関数は「副次作用」が少なくなります。ここでいう副次作用とは、データベースの表またはパッケージ変数に対する参照です。

副次作用によって、問合せのパラレル処理を妨げたり、処理順序に依存する（したがって、不確定な）結果が生じたり、ユーザー・セッションごとにパッケージ状態を変更してしまう（これは許可されない）場合があります。したがって、SQL 式からコールされたストアド関数には、次の規則が適用されます。

- この関数はデータベースの表を変更できない。したがって、INSERT 文、UPDATE 文、DELETE 文は実行できません。

- パッケージ変数の値を読み書きする関数は、リモートまたはパラレルでは実行できない。
- SELECT 句、VALUES 句、SET 句からコールされる関数だけが、パッケージ変数の値を記述できる。
- この関数は、前述の規則のいずれかに従っていない別のサブプログラムをコールすることはできない。この関数は、前述の規則のいずれかに従っていないビューを参照することはできません（Oracle は、ビューへの参照を、関数・コールを含むことができるストアド SELECT 操作と置き換えます）。

スタンドアロン関数の場合は、Oracle は関数本体をチェックして、これらの規則を施行できます。しかし、パッケージ関数の本体は隠され、その仕様部だけしか参照できません。したがって、パッケージ関数の場合は、プラグマ（コンパイラ・ディレクティブ）RESTRICT_REFERENCES を使用して、これらの規則を施行しなければなりません。

このプラグマは、データベースの表、またはパッケージ変数、あるいはその両方に対して、パッケージ関数の読込み / 書込みアクセスを否認するように PL/SQL コンパイラに指示します。このプラグマに違反する関数本体をコンパイルしようとすると、コンパイル・エラーを受け取ります。

パッケージ関数をコールする

SQL 式からパッケージ関数をコールするには、プラグマ 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て、純粋さレベルを宣言しなければなりません。プラグマは関数宣言の後になければなりませんが、直後にある必要はありません。所定の関数宣言を参照できるプラグマは、1 つだけです。

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]);
```

ここで、

WNDS	「writes no database state」（データベースの表を変更しない）を意味します。
RNDS	「reads no database state」（データベースの表を問い合せない）を意味します。
WNPS	「writes no package state」（パッケージ変数の値を変更しない）を意味します。
RNPS	「reads no package state」（パッケージ変数の値を参照しない）を意味します。

引数はどんな順序でも渡すことはできますが、引数 WNDS は必ず渡さなければなりません。どの引数も、別の引数を暗黙指定することはありません。たとえば、RNPS は WNPS を暗黙指定しません。

次の例では、関数 compound により、データベース状態またはパッケージ状態の読込みおよび書込みは行われないため、最大の純粋さレベルを宣言できます。関数で可能な最高の純粋さレベルを、常に保つようにしてください。そうすれば、PL/SQL コンパイラが、関数を不必要に拒否することはありません。

```
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;
```

後から、次のように、PL/SQL ブロックから compound をコールできます。

```
BEGIN
   ...
    SELECT finance.compound(yrs,amt,rte)  -- function call       INTO interest
FROM accounts       WHERE acctno = acct_id;
```

初期化部分でパッケージを参照する

パッケージは初期化部分を持つことができます。初期化部分はパッケージ本体に隠されています。通常、初期化部分には、パブリック変数を初期化する文を記述します。

次の例では、SELECT 文によって、パブリック変数 prime_rate が初期化されます。

```
CREATE PACKAGE loans AS
   prime_rate REAL; -- public packaged variable
   ...
END loans;

CREATE PACKAGE BODY loans AS
   ...
BEGIN -- initialization part
```

```
   SELECT prime INTO prime_rate FROM rates;
END loans;
```

初期化コードは、パッケージが最初に参照されるときに 1 回だけ実行されます。コードが、データベース状態または自分以外のパッケージ状態を読み書きする場合は、副次作用が生じる可能性があります。また、パッケージを参照する（そして、それによって初期化コードを実行する）ストアド関数によって、間接的に副次作用が生じる可能性があります。したがって、SQL 式から関数をコールするには、プラグマ RESTRICT_REFERENCES を使用して、初期化コードの純粋さレベルを宣言または暗黙指定しなければなりません。

初期化コードの純粋さレベルを宣言するには、プラグマ RESTRICT_REFERENCES の可変レコードを使用します。ここで、関数名はパッケージ名と置き換えられます。パッケージ仕様部にプラグマをコーディングします。パッケージ仕様部では、このプラグマを他のユーザーが参照できます。このように、パッケージを参照する人はすべて、制限を確認して、それに従うことができます。

可変プラグマ 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 (
    package_name, WNDS [, WNPS] [, RNDS] [, RNPS]);
```

この引数 WNDS、WNPS、RNDS、RNPS は、通常の意味です。

次の例では、初期化コードはデータベース状態を読み込み、パッケージ状態を書き込みます。ただし、このコードは、許可されているそれ自身のパッケージの状態を書き込んでいるので、WNPS を宣言できます。そのため、関数で使用できる最高の純粋さレベル WNDS、WNPS、RNPS を宣言できます。（パブリック変数 prime_rate が別のパッケージにある場合は、WNPS は宣言できません）。

```
CREATE PACKAGE loans AS
    PRAGMA RESTRICT_REFERENCES (loans, WNDS, WNPS, RNPS);
    prime_rate REAL;
    ...
END loans;

CREATE PACKAGE BODY loans AS
...
BEGIN
    SELECT prime INTO prime_rate FROM rates;
END loans;
```

プラグマは、パッケージ仕様部のどこにでも入れることができますが、目立つように先頭に入れることをお薦めします。

初期化コードの純粋さレベルを暗黙指定するには、パッケージが、宣言する関数のいずれか 1 つに対して RESTRICT_REFERENCES プラグマを持っていなければなりません。そのプラグマから、Oracle は初期化コードの純粋さレベルを推論できます（このコードは、プラグマが施行する規則を破ることができないからです）。次の例では、関数 discount のプラグマに

よって、初期化コードの純粋さレベルが少なくとも WNDS であることが暗黙指定されています。

```
CREATE PACKAGE loans AS
    ...
    FUNCTION discount (...) RETURN NUMBER;
    PRAGMA RESTRICT_REFERENCES (discount, WNDS);
END loans;
...
```

推論を引き出すために、Oracle はすべての RESTRICT_REFERENCES プラグマのアサートを結合できます。たとえば、次のプラグマ（結合されている）は、初期化コードの純粋さレベルが少なくとも WNDS、RNDS であることを暗黙指定しています。

```
CREATE PACKAGE loans AS
    ...
    FUNCTION discount (...) RETURN NUMBER;
    FUNCTION credit_ok (...) RETURN CHAR;
    PRAGMA RESTRICT_REFERENCES (discount, WNDS);
    PRAGMA RESTRICT_REFERENCES (credit_ok, RNDS);
END loans;
...
```

問題を回避する

SQL 式からパッケージ関数をコールするには、プラグマ RESTRICT_REFERENCES を使用して、純粋さレベルを宣言しなければなりません。しかし、パッケージに初期化部分があると、PL/SQL コンパイラは、関数で可能な最高の純粋さレベルを宣言させない場合があります。その結果、リモート、またはパラレルで関数をコールしたり、ある SQL 句から関数をコールできない場合があります。

これは、パッケージ関数の純粋さレベルが、パッケージ初期化コードよりも高い場合に起こります。パッケージが初めて参照されるときに、

その初期化コードが実行されることを覚えておいてください。その参照が関数・コールの場合、そのコール中に初期化コードによる別の副次作用が生じます。したがって、事実上、初期化コードは関数の純粋さレベルを低下させます。

この問題を回避するには、パッケージ初期化コードをサブプログラムに移動します。こうすれば、アプリケーションは、パッケージ関数に影響を与えずに、コードを明示的に（パッケージ・インスタンシエーション中に暗黙的にではなく）実行できます。

パッケージ関数がコールするサブプログラムの純粋さレベルが、そのパッケージ関数よりも低いと、同様の問題が発生します。これによって、関数の純粋さレベルが低下します。したがって、この関数のためのプラグマ RESTRICT_REFERENCES は、もっと低い純粋さレベルを指定しなければなりません。そうしないと、PL/SQL コンパイラがその関数を拒否します。次の例では、プラグマが RNDS を宣言し、関数が、データベース状態を読み込むプロシージャをコールするため、コンパイラは関数を拒否します。

```
CREATE PACKAGE finance AS
   ...
   FUNCTION compound (years  IN NUMBER,
                      amount IN NUMBER) RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES (compound, WNDS, WNPS, RNDS, RNPS);
END finance;

CREATE PACKAGE BODY finance AS
   ...
   FUNCTION compound (years  IN NUMBER,
                      amount IN NUMBER) RETURN NUMBER IS
      rate  NUMBER;
      PROCEDURE calc_loan_rate (loan_rate OUT NUMBER) IS
         prime_rate REAL;
      BEGIN
         SELECT p_rate INTO prime_rate FROM rates;
         ...
      END;
   BEGIN
      calc_loan_rate(rate);
      RETURN amount * POWER((rate / 100) + 1, years);
   END compound;
END finance;
```

オーバーロードする

PL/SQL では、パッケージ関数（ただしスタンドアロンでない）のオーバーロードが可能です。これにより、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、別の関数に対して同じ名前を使用できます。

しかし、RESTRICT_REFERENCES プラグマは、1 つの関数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関数の名前を参照するプラグマは、常に前にある、最も近い関数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);
    ...
```

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、拡張性が制限されます。これを解決するには、プラグマ構文を使っ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、別の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、作業単位内でだけ使用できることを意味します。そのため、作業単位の終了時にメモリーはそのプールに解放され、（すべてのグローバル変数の初期化コードの実行後）別のユーザーにより再使用されます。

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 コールまたは 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 CALL 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、グローバル変数および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 CALL 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規インスタンシエーション（後述のとおり）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 CALL 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 CALL 時に逐次再使用可能パッケージの新規インスタンシエーションが作成されても、Oracle によりメモリーが割り当てられる、または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。単に、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みの）インスタンス化作業域を探すだけです。サーバーへの CALL 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う理由

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、ログオン・セッションは一般に何日間も持続します。通常、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だけで使う必要があります。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。

SERIALLY_REUSABLE パッケージを使えば、アプリケーション開発者は、アプリケーションをモデル化し、メモリーをより適切に管理して拡張性を得ることができます。サーバーへの CALL 中にだけ管理されるパッケージ状態は、SERIALLY_REUSABLE パッケージ内に獲得されます。

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が付きます。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れば、本体にもそのプラグマがなければなりません。また、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。

方法

SERIALLY_REUSABLE のマークが付いたパッケージには、次のプロパティがあります。

- パッケージ変数は、サーバーへの CALL に対応した WORK 境界内でしか使えません（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）。

  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った場合、このアプリケーション・プログラムは失敗することがあります。PL/SQL では、このようなケースはチェックされません。

- パッケージ・インスタンシエーションのプールが保持され、「作業単位」にこのパッケージが必要な場合は必ず、このインスタンシエーションの 1 つが次のように「再使用」されます。
  - パッケージ変数は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は、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（収集変数または長い VARCHAR2 のメモリーなど）。
  - このパッケージ・インスタンシエーションは、このパッケージ用に保持された再使用可能インスタンシエーションのプールに戻されます。
- トリガー内から逐次再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しようとすると、「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 < パッケージ名 > にはアクセスできません」というエラー・メッセージが出されます。

例 1

この例は、逐次再使用可能パッケージ仕様部（本体はありません）です。これは、パッケージ変数が CALL 境界全域で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;
/
```

上記のプログラムでは、次のように出力されます。

```
5
```

注意： パッケージにプラグマ SERIALLY_REUSABLE がない場合は、'10' が出力されます。

例 2

これは、逐次再使用可能なパッケージ仕様部および本体の例です。例 1 と同様に、この例ではパッケージ変数が CALL 境界全域でどのように機能するかを示しています。

```
SQL> connect scott/tiger;
Connected.
SQL>
SQL> drop package SR_PKG;
Statement processed.
SQL>
SQL> create or replace package SR_PKG is
```

```
  2>
  3>  pragma SERIALLY_REUSABLE;
  4>
  5>  type str_table_type is table of varchar2(200) index by binary_integer;
  6>
  7>  num     number        := 10;
  8>  str     varchar2(200) := 'default-init-str';
  9>  str_tab str_table_type;
 10>
 11>  procedure print_pkg;
 12>  procedure init_and_print_pkg(n number, v varchar2);
 13>
 14> end SR_PKG;
 15> /
Statement processed.
SQL>
SQL>
SQL> create or replace package body SR_PKG is
  2>
  3>  -- the body is required to have the pragma since the
  4>  -- specification of this package has the pragma
  5>  pragma SERIALLY_REUSABLE;
  6>
  7>  procedure print_pkg is
  8>  begin
  9>    dbms_output.put_line('num: ' || SR_PKG.num);
 10>    dbms_output.put_line('str: ' || SR_PKG.str);
 11>
 12>    dbms_output.put_line('number of table elems: ' || SR_PKG.str_tab.count);
 13>    for i in 1..SR_PKG.str_tab.count loop
 14>      dbms_output.put_line(SR_PKG.str_tab(i));
 15>    end loop;
 16>  end;
 17>
 18>  procedure init_and_print_pkg(n number, v varchar2) is
 19>  begin
 20>
 21>    -- init the package globals
 22>    SR_PKG.num := n;
 23>    SR_PKG.str := v;
 24>    for i in 1..n loop
 25>      SR_PKG.str_tab(i) := v || ' ' || i;
 26>    end loop;
 27>
 28>    -- now print the package
 29>    print_pkg;
 30>  end;
```

```
    31>
    32> end SR_PKG;
    33> /
Statement processed.
SQL> show errors;
No errors for PACKAGE BODY SR_PKG
SQL>
SQL> set serveroutput on;
Server Output                   ON
SQL>
SQL> Rem SR package access in a CALL
SQL> begin
     2>
     3>   -- initialize and print the package
     4>   dbms_output.put_line('Initing and printing pkg state..');
     5>   SR_PKG.init_and_print_pkg(4, 'abracadabra');
     6>
     7>   -- print it in the same call to the server.
     8>   -- we should see the initialized values.
     9>   dbms_output.put_line('Printing package state in the same CALL...');
    10>   SR_PKG.print_pkg;
    11>
    12> end;
    13> /
Statement processed.
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
SQL>
SQL> Rem SR package access in subsequent CALL
SQL> begin
     2>
     3>  -- print the package in the next call to the server.
     4>  -- We should that the package state is reset to the initial (default)
```

```
values.
     5>  dbms_output.put_line('Printing package state in the next CALL...');
     6>  SR_PKG.print_pkg;
     7>
     8> end;
     9> /
Statement processed.
Printing package state in the next CALL...
num: 10
str: default-init-str
number of table elems: 0
SQL>
```

例 3

この例は、逐次再使用可能パッケージのオープン・カーソルが WORK 境界（CALL）の終了時に自動的にクローズし、新規 CALL 内でこれらのカーソルが再度オープンされなければならないことを示しています。

```
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  Since the cursor is closed - every time we fetch we
Rem  will start at the first row again.

SQL> connect scott/tiger;
Connected.
SQL>
SQL> drop package SR_PKG;
Statement processed.
SQL> drop table people;
Statement processed.
SQL>
SQL>
SQL> create table people (name varchar2(20));
Statement processed.
SQL>
SQL> insert into  people  values ('ET');
1 row processed.
SQL> insert into  people  values ('RAMBO');
1 row processed.
SQL>
SQL> create or replace package SR_PKG is
     2>
     3>  pragma SERIALLY_REUSABLE;
     4>  cursor c is select name from people;
     5>
```

```
     6> end SR_PKG;
     7> /
Statement processed.
SQL> show errors;
No errors for PACKAGE SR_PKG
SQL>
SQL> set serveroutput on;
Server Output                   ON
SQL>
SQL> create or replace procedure fetch_from_cursor is
     2>   name varchar2(200);
     3> begin
     4>
     5>  if (SR_PKG.c%ISOPEN) then
     6>   dbms_output.put_line('cursor is already open.');
     7>  else
     8>   dbms_output.put_line('cursor is closed; opening now.');
     9>   open SR_PKG.c;
    10>  end if;
    11>
    12>  -- fetching from cursor.
    13>  fetch SR_PKG.c into name;
    14>  dbms_output.put_line('fetched: ' || name);
    15>
    16>  fetch SR_PKG.c into name;
    17>  dbms_output.put_line('fetched: ' || name);
    18>
    19>  -- Oops forgot to close the cursor (SR_PKG.c).
    20>  -- But, since it is a Serially Reusable pkg's cursor,
    21>  -- it will be closed at the end of this CALL to the server.
    22>
    23> end;
    24> /
Statement processed.
SQL> show errors;
No errors for PROCEDURE FETCH_FROM_CURSOR
SQL>
SQL> set serveroutput on;
Server Output                   ON
SQL>
SQL> execute fetch_from_cursor;
Statement processed.
cursor is closed; opening now.
fetched: ET
fetched: RAMBO
SQL>
SQL> execute fetch_from_cursor;
```

```
Statement processed.
cursor is closed; opening now.
fetched: ET
fetched: RAMBO
SQL>
SQL> execute fetch_from_cursor;
Statement processed.
cursor is closed; opening now.
fetched: ET
fetched: RAMBO
SQL>
```

必要な権限

SQL から PL/SQL 関数をコールするには、使用者がその所有者であるか、またはその関数に対して EXECUTE 権限を持っていなければなりません。PL/SQL 関数を使って定義されているビューから選択するには、そのビューに対して SELECT 権限を持っていなければなりません。そのビューから選択するためには、別の EXECUTE 権限は必要ありません。

提供されるパッケージ

データベースの機能性を拡張できるように、または PL/SQL で一部の SQL 機能を利用できるように、数種類のパッケージ・プロシージャが Oracle Server で提供されています。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性を利用したり、単に、独自のストアド・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場所を示します。これらのパッケージは、パッケージ所有者ではなく、呼び出しているユーザーとして実行します。パッケージ・プロシージャは、同じ名前のパブリックのシノニムを通じてコールできます。

SQL 機能をサポートするパッケージ

PL/SQL で一部の SQL 機能を利用できるように、次のパッケージ・プロシージャが提供されています。

- DBMS_DDL
- DBMS_SESSION
- DBMS_TRANSACTION
- DBMS_UTILITY

表 10-6 に、各パッケージが説明されています。表 10-6 の脚注には、各プロシージャの使用上の制限が示されています。これらのパッケージについての最新情報を得るには、パッケージ仕様部を調べてください。

表 10-3　DBMS_DDL

パッケージ	プロシージャ（引数）	等価な SQL コマンド
DBMS_DDL	alter_compile(type varchar2, schema varchar2, name varchar2)（注意1、2、3、4）	ALTER PROCEDURE Pro*C COMPILE
		ALTER FUNCTION func COMPILE
		ALTER PACKAGE pack COMPILE
	analyze_object(type varchar2, schema varchar2, name varchar2, method varchar2, estimate_rows number default null, estimate_percent number default null)	ANALYZE INDEX
		ANALYZE TABLE
		ANALYZE CLUSTER

注意 1: トリガーでは禁止されます。

注意 2: SQL*Forms からコールされるプロシージャでは禁止されます 。

注意 3: 読込み専用トランザクションでは禁止されます。

注意 4: リモート（整合）セッションでは禁止されます 。

注意 5: 再帰セッションでは禁止されます。

注意 6: ストアド・プロシージャでは禁止されます。

表 10-4　DBMS_SESSION

パッケージ	プロシージャ（引数）	等価な SQL コマンド
DBMS_ SESSION	close_database_link(dblink varchar2)	ALTER SESSION CLOSE DATABASE dblink
	reset_package （注意 5 を参照）	このプロシージャは、すべてのパッケージの状態を再初期化する。 これに相当する SQL はありません。
	set_nls(param varchar2, value varchar2) （注意 1、4）	ALTER SESSION SET nls_param = nls_param_values
	set_role(role_cmd varchar2) （注意 1、6）	SET ROLE ...
	set_sql_trace(sql_trace boolean)	ALTER SESSION SET SQL_TRACE = [TRUE | FALSE]
	unique_session_id return varchar2	この関数は一意のセッション ID を戻す。 これに相当する SQL はありません。
	is_role_enabled return boolean	この関数は、ロールが使用可能な状態かどうかを判断するのに使用される。 これに相当する SQL はありません。
	et_close_cached_open_cursors(close_cursors boolean)	ALTER SESSION SET CLOSE_CACHED_OPEN_ CURSORS
	free_unused_user_memory	このプロシージャは、未使用のメモリーを再生させる。 これに相当する SQL はありません。

注意 1: トリガーでは禁止されます。

注意 2: SQL*Forms からコールされるプロシージャでは禁止されます 。

注意 3: 読込み専用トランザクションでは禁止されます。

注意 4: リモート（整合）セッションでは禁止されます 。

注意 5: 再帰セッションでは禁止されます。

注意 6: ストアド・プロシージャでは禁止されます。

表 10-5　DBMS_TRANSACTION

パッケージ	プロシージャ（引数）	等価な SQL コマンド
DBMS_TRANSACTION	advise_commit	ALTER SESSION ADVISE COMMIT
	advise_rollback	ALTER SESSION ADVISE ROLLBACK
	advise_nothing	ALTER SESSION ADVISE NOTHING
	commit （注意 1、2、4）	COMMIT
	commit_Comment(cmnt varchar2) （注意 1、2、4）	COMMIT COMMENT text
	commit_force(xid varchar2, scn varchar2 default null) （注意 1、2、3、4）	COMMIT FORCE text ...
	read_only （注意 1、3、4）	SET TRANSACTION READ ONLY
	read_write （注意 1、3、4）	SET TRANSACTION READ WRITE
	rollback （注意 1、2、4）	ROLLBACK
	rollback_force(xid varchar2) （注意 1、2、3、4）	ROLLBACK ... FORCE text ...
	rollback_savepoint(svpt varchar2) （注意 1、2、4）	ROLLBACK ... TO SAVEPOINT ...
	savepoint(savept varchar2) （注意 1、2、4）	SAVEPOINT savepoint
	use_rollback_segment(rb_name varchar2) （注意 1、2、4）	SET TRANSACTION USE ROLLBACK SEGMENT segment
	purge_mixed(xid in number)	関連項目：『Oracle8 Server 分散システム』
	begin_discrete_transaction （注意 1、3、4、5）	関連項目：『Oracle8 Server チューニング』
	local_transaction_id(create_transaction BOOLEAN default FALSE) return VARCHAR2	関連項目：『Oracle8 Server 分散システム』
	step_id return number	関連項目：『Oracle8 Server 分散システム』

表 10-6　DBMS_UTILITY

パッケージ	プロシージャ（引数）	等価な SQL コマンド
DBMS_UTILITY	compile_schema(schema varchar2) （注意 1、2、3、4）	このプロシージャは、アクセス可能なすべてのプロシージャ、関数、パッケージに対して alter_compile をコールすることに等しい。コンパイルは、依存順序で行われます。
	analyze_schema(schema varchar2, method varchar2, estimate_rows number default null, estimate_percent number default null)	このプロシージャは、所定のスキーマ内のすべてのオブジェクトに対して analyze_object をコールすることに等しい。
	analyze_part_object(schema in varchar2 default null, object_name in varchar2 default null, object_type in char default 'T' , command_type in char default 'E', command_opt in varchar2 default null, sample_clause in varchar2 default 'sample 5 percent')	ジョブ待ち行列を使って、オブジェクトの各パーティションについて、ANALYZE TABLE | INDEX [<schema>.]<object_name> PARTITION <pname> [<command_type>] [<command_opt>] [<sample_clause>]" がパラレルで実行される。このプロシージャでは、各パーティションのジョブが送られます。初期化パラメータ JOB_QUEUE_PROCESSES を使って並列ジョブの数を指定できます。 オブジェクト型は、T（表）または I（索引）でなければなりません。command_type は、次のとおり。 - C（統計の計算） - E（統計の推定） - D（統計の削除） - V（構造体の妥当性検査） V の場合、object_type が T であれば、command_opt は 'CASCADE' にできます。C または E の場合、command_opt は FOR table、または FOR all LOCAL indexes、FOR all columns、分析統計（表）の 'for' オプションの一部の組合せにできます。 sample_clause では、command_type が E の場合に使うサンプルの句を指定します。
	format_error_stack return varchar2	この関数は、エラー・スタックを変数にフォーマットする。

表 10-6　DBMS_UTILITY

パッケージ	プロシージャ（引数）	等価な SQL コマンド
DBMS_UTILITY（続き）	`format_error_stack return varchar2`	この関数は、エラー・スタックを変数にフォーマットする。
	`format_call_stack return varchar2`	この関数は、現行のコール・スタックを変数にフォーマットする。
	`name_resolve(name in varchar2, context in number, schema out varchar2, part1 out varchar2, part2 out varchar2, dblink out varchar2, part1_type out number, object_number out number)`	関連項目：『Oracle8 Server 分散システム』

注意 1: トリガーでは禁止されます。

注意 2: SQL*Forms からコールされるプロシージャでは禁止されます 。

注意 3: 読込み専用トランザクションでは禁止されます。

注意 4: リモート（整合）セッションでは禁止されます 。

注意 5: 再帰セッションでは禁止されます。

注意 6: ストアド・プロシージャでは禁止されます。

それぞれの等価な SQL コマンドの詳細は、『Oracle8 Server SQL リファレンス』を参照してください。

COMMIT および ROLLBACK、ROLLBACK... TO SAVEPOINT、SAVEPOINT は、直接 PL/SQL によってサポートされており、DBMS_TRANSACTION パッケージにすべて含まれています。

その他の機能性をサポートするパッケージ

データベースの機能性を拡張するための数種類のパッケージ（DBMS_* および UTL_* パッケージ）が提供されています。表 10-7 のクロス・リファレンスの欄には、これらの個々のパッケージの説明箇所を示します。

表 10-7　提供されるパッケージ : その他の機能性

パッケージ名	説明	クロス・リファレンス
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。	第 16 章の「アラート機能によるデータベースのイベントの通知」
DBMS_DESCRIBE	ストアド・プロシージャの引数を記述します。	10-65 ページの「DBMS_DESCRIBE パッケージ」
DBMS_JOB	周期的に実行する管理プロシージャを設定します。	『Oracle8 Server 管理者ガイド』
DBMS_LOCK	ご使用のアプリケーションに対して、Oracle Lock Management サービスを使用します。	3-17 ページの「DBMS_LOCK パッケージ」
DBMS_OUTPUT	トリガー、プロシージャ、パッケージからメッセージを出力します。	12-20 ページの「ストアド・プロシージャとトリガーからの出力」
DBMS_PIPE	同一インスタンスのセッションが、相互に通信できるようにします。	第 12 章の「PL/SQL 入出力」
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、メモリーからスワップアウトしません。	『Oracle8 Server チューニング』
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。	『Oracle8 Server チューニング』
DBMS_SYSTEM	システム・レベルのユーティリティを提供します。たとえば、セッションに対する SQL トレースを使用可能にします。	『Oracle8 Server チューニング』
DBMS_SPACE	標準ビューで使用できないセグメント・スペース情報を提供します。	『Oracle8 Server 管理者ガイド』
DBMS_SQL	動的 SQL を使用してストアド・プロシージャおよび無名 PL/SQL ブロックを作成します。DML 文または DDL 文を解析します。	第 14 章の「動的 SQL の使用」

表 10-7　提供されるパッケージ : その他の機能性

パッケージ名	説明	クロス・リファレンス
DBMS_ROWID	データ・ブロック番号、オブジェクト番号、その他のコンポーネントを含む、ROWID についての情報を入手します。	10-74 ページの「DBMS_ROWID パッケージ」
DBMS_LOB	Oracle Server 上で稼動する PL/SQL プログラムを使用して、大規模なオブジェクトを操作します。	6-62 ページの「DBMS_LOB パッケージ」
DBMS_AQ	メッセージ（事前定義済みのオブジェクト型の）を待ち行列に追加、またはメッセージを待ち行列から削除します。	第 11 章の「アドバンスト・キューイング」
DBMS_AQADM	事前定義済みオブジェクト型のメッセージ用の待ち行列または待ち行列表に関する管理機能を実行します。	第 11 章の「アドバンスト・キューイング」
DBMS_DISTRIBUTED_TRUST_ADMIN	Trusted Servers List をメンテナンスします。Trusted Servers List 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を判別するために中央認可レベルのリストと共に使われます。	『Oracle8 Server 分散システム』
DMBS_HS	分散外部プロシージャおよびリモート・ライブラリ、非 Oracle システムの登録または削除によって、異機種間サービスを管理します。また、非 Oracle システムの一部の初期化変数を作成または削除できます。	『Oracle8 Server 分散システム』
DMBS_HS_EXTPROC	分散外部プロシージャのセキュリティを確立するために異機種間サービスを使用できます。	『Oracle8 Server 分散システム』
DMBS_HS_PASSTHROUGH	非 Oracle システムにパススルー SQL 文を送信するために異機種間サービスを使用できます。	『Oracle8 Server 分散システム』

表 10-7　提供されるパッケージ : その他の機能性

パッケージ名	説明	クロス・リファレンス
DBMS_REFRESH	トランザクション的に一貫した時点に、一緒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。この機能を使用するには、分散オプションが必要です。	『Oracle8 Server レプリケーション』
DBMS_SNAPSHOT	同じリフレッシュ・グループの一部ではない 1 つまたは複数のスナップショットをリフレッシュして、スナップショット・ログを削除します。この機能を使用するには、分散オプションが必要です。	『Oracle8 Server レプリケーション』
DBMS_DEFER, DMBS_DEFER_SYS, DBMS_DEFER_QUERY	遅延リモート・プロシージャ・コールを作成および管理します。この機能を使用するには、レプリケーション・オプションが必要です。	『Oracle8 Server レプリケーション』
DBMS_REPCAT	Oracle の対称レプリケーション機能を使用します。この機能を使用するには、レプリケーション・オプションが必要です。	『Oracle8 Server レプリケーション』
DBMS_REPCAT_AUTH, 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。この機能を使用するには、レプリケーション・オプションが必要です。	『Oracle8 Server レプリケーション』
UTL_HTTP	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ために、PL/SQL および SQL から HTTP コールアウトができます。	10-82 ページの「UTL_HTTP パッケージ」

ストアド・プロシージャを記述する

DBMS_DESCRIBE パッケージを使用すると、ストアド・プロシージャまたは関数についての情報を得ることができます。

このパッケージにより、Oracle コール・インタフェースの OCIDescribeAny() コールと同じ機能性が提供されます。このパッケージに入っているプロシージャ DESCRIBE_PROCEDURE は、ストアド・プロシージャの名前、およびそのプロシージャとその各パラメータの説明を受け取ります。OCIDescribeAny() コールの詳細は、『Oracle 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。

DBMS_DESCRIBE パッケージ

DBMS_DESCRIBE パッケージを作成するには、ユーザー SYS として接続し、DBMSDESC.SQL スクリプトと PRVTDESC.PLB スクリプトを実行します。これらのスクリプトは、CATPROC.SQL スクリプトによって自動的に実行されます。このパッケージを実行するユーザーに必要な権限の付与の詳細は、10-36 ページの「プロシージャを実行するために必要な権限」を参照してください。

セキュリティ

このパッケージは、PUBLIC で使用可能です。これによって、記述されているスキーマ・オブジェクトに基づいて独自のセキュリティ検査が行われます。

タイプ

DBMS_DESCRIBE パッケージは 2 つの PL/SQL 表のタイプを宣言します。これらのタイプは、DESCRIBE_PROCEDURE が戻すデータをその OUT パラメータ内に保持するために使用されます。このタイプは、次のとおりです。

```
TYPE VARCHAR2_TABLE IS TABLE OF VARCHAR2(30)
    INDEX BY BINARY_INTEGER;

TYPE NUMBER_TABLE IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
```

エラー

DBMS_DESCRIBE では、-20000 から -20004 までの範囲のアプリケーション・エラーが発生する場合があります。このエラーは、次のとおりです。

```
-20000: ORU 10035: cannot describe a package ('X') only a
        procedure within a package
-20001: ORU-10032: procedure 'X' within package 'Y' does not
        exist
-20002: ORU-10033 object 'X' is remote, cannot describe; expanded
        name 'Y'
```

```
-20003: ORU-10036: object 'X' is invalid and cannot be described
-20004: syntax error attempting to parse 'X'
```

DESCRIBE_PROCEDURE プロシージャ

構文

表 10-8 に、DESCRIBE_PROCEDURE のパラメータを示します。構文は次のとおりです。

```
PROCEDURE DESCRIBE_PROCEDURE(
        object_name    IN VARCHAR2,
        reserved1      IN VARCHAR2,
        reserved2      IN VARCHAR2,
        overload       OUT NUMBER_TABLE,
        position       OUT NUMBER_TABLE,
        level          OUT NUMBER_TABLE,
        argument_name  OUT VARCHAR2_TABLE,
        datatype       OUT NUMBER_TABLE,
        default_value  OUT NUMBER_TABLE,
        in_out         OUT NUMBER_TABLE,
        length         OUT NUMBER_TABLE,
        precision      OUT NUMBER_TABLE,
        scale          OUT NUMBER_TABLE,
        radix          OUT NUMBER_TABLE
        spare          OUT NUMBER_TABLE);
```

表 10-8　DBMS_DESCRIBE.DESCRIBE_PROCEDURE パラメータ

パラメータ	モード	説明
object_name	IN	記述するプロシージャの名前です。このパラメータの構文は、SQL の識別子に使用されている規則に従っています。名前はシノニムでかまいません。このパラメータは必須であり、NULL 指定はできません。名前の長さは、合計で 197 バイトを超えることはできません。OBJECT_NAME の指定が正しくないと、次の例外のいずれかが発生します。 ORA-20000 - パッケージが指定されました。指定できるのは、ストアド・プロシージャ、ストアド関数、パッケージ・プロシージャ、パッケージ関数だけです。 ORA-20001 - 指定したプロシージャまたは関数は、所定のパッケージ内に存在しません。 ORA-20002 - 指定したオブジェクトは、リモート・オブジェクトです。このプロシージャは、現在、リモート・オブジェクトを記述できません。 ORA-20003 - 指定したオブジェクトは、無効であり、記述できません。 ORA-20004 - オブジェクトが指定されましたが、構文エラーがあります。
reserved1 reserved2	IN	将来の使用に備えて確保されています。NULL または空の文字列に設定しておかなければなりません。
overload	OUT	プロシージャの署名に割り当てられた一意の数値です。プロシージャがオーバーロードされると、このフィールドは、プロシージャの各バージョンに対して異なる値を保持します。
position	OUT	パラメータ・リスト内の引数の位置です。位置 0 は、関数の戻り型の値を戻します。
level	OUT	引数がレコードのような複合タイプの場合、このパラメータはデータ型のレベルを戻します。ODESSP コールの例および使用方法の詳細は、『Oracle 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。
argument_name	OUT	記述するプロシージャに対応付ける引数の名前です。

表 10-8　DBMS_DESCRIBE.DESCRIBE_PROCEDURE パラメータ

パラメータ	モード	説明
datatype	OUT	記述する引数の Oracle データ型です。データ型およびその数値タイプ・コードは、次のとおりです。 0 引数のないプロシージャのプレースホルダ 1　VARCHAR, VARCHAR, STRING 2　NUMBER, INTEGER, SMALLINT, REAL, FLOAT, DECIMAL 3　BINARY_INTEGER, PLS_INTEGER, POSITIVE, NATURAL 8　LONG 11　ROWID 12　DATE 23　RAW 24　LONG RAW 96　CHAR (ANSI FIXED CHAR), CHARACTER 106　MLSLABEL 250　PL/SQL RECORD 251　PL/SQL TABLE 252　PL/SQL BOOLEAN
default_value	OUT	記述する引数がデフォルト値を持っている場合、このパラメータは 1 です。それ以外の場合は 0 になります。
in_out	OUT	パラメータのモードを記述します。 0　IN 1　OUT 2　IN OUT
length	OUT	記述する引数のデータ長（バイト単位）。
precision	OUT	記述する引数のデータ型が 2(NUMBER) の場合、このパラメータはその数値の精度です。
scale	OUT	記述する引数がデータ型 2(NUMBER など) である場合、このパラメータはその数値の位取りです。
radix	OUT	記述する引数がデータ型 2(NUMBER など) である場合、このパラメータはその数値の基数です。

表 10-8　DBMS_DESCRIBE.DESCRIBE_PROCEDURE パラメータ

パラメータ	モード	説明
spare	OUT	将来の機能に備えて確保されています。

戻り値

DESCRIBE_PROCEDURE からの値はすべて、その OUT パラメータに戻されます。これらのデータ型は、パラメータの変数値に適応する、PL/SQL 表です。

例

DESCRIBE_PROCEDURE プロシージャは、外部サービス・インタフェースとして使用することもできます。

たとえば、SCOTT.ACCOUNT_UPDATE の OBJECT_NAME を与えるクライアントについて考えます。ここで、ACCOUNT_UPDATE は、次の指定によるオーバーロードの関数です。

```
table account (account_no number, person_id number,
               balance number(7,2))
table person  (person_id number(4), person_nm varchar2(10))

function ACCOUNT_UPDATE (account_no   number,
                         person       person%rowtype,
                         amounts      dbms_describe.number_table,
                         trans_date   date)
                         return       accounts.balance%type;

function ACCOUNT_UPDATE (account_no   number,
                         person       person%rowtype,
                         amounts      dbms_describe.number_table,
                         trans_no     number)
                             return       accounts.balance%type;
```

このプロシージャにより、次に示すような出力が得られます。

```
overload position  argument level  datatype length prec scale rad
-------- --------- -------- ------ -------- ------ ---- ----- ---
       1         0                0        2     22    7     2  10
       1         1  ACCOUNT       0        2      0    0     0   0
       1         2  PERSON        0      250      0    0     0   0
       1         1  PERSON_ID     1        2     22    4     0  10
       1         2  PERSON_NM     1        1     10    0     0   0
       1         3  AMOUNTS       0      251      0    0     0   0
       1         1                1        2     22    0     0   0
       1         4  TRANS_DATE    0       12      0    0     0   0
       2         0                0        2     22    7     2  10
       2         1  ACCOUNT_NO    0        2     22    0     0   0
```

```
2        2   PERSON      0        2     22    4     0  10
2        3   AMOUNTS     0      251     22    4     0  10
2        1               1        2      0    0     0   0
2        4   TRANS_NO    0        2      0    0     0   0
```

次の PL/SQL プロシージャは、すべての PL/SQL データ型を、そのパラメータとして持っています。

```
CREATE OR REPLACE PROCEDURE p1 (
        pvc2    IN     VARCHAR2,
        pvc     OUT    VARCHAR,
        pstr    IN OUT STRING,
        plong   IN     LONG,
        prowid  IN     ROWID,
        pchara  IN     CHARACTER,
        pchar   IN     CHAR,
        praw    IN     RAW,
        plraw   IN     LONG RAW,
        pbinint IN     BINARY_INTEGER,
        pplsint IN     PLS_INTEGER,
        pbool   IN     BOOLEAN,
        pnat    IN     NATURAL,
        ppos    IN     POSITIVE,
        pposn   IN     POSITIVEN,
        pnatn   IN     NATURALN,
        pnum    IN     NUMBER,
        pintgr  IN     INTEGER,
        pint    IN     INT,
        psmall  IN     SMALLINT,
        pdec    IN     DECIMAL,
        preal   IN     REAL,
        pfloat  IN     FLOAT,
        pnumer  IN     NUMERIC,
        pdp     IN     DOUBLE PRECISION,
        pdate   IN     DATE,
        pmls    IN     MLSLABEL) AS

BEGIN
    NULL;
END;
```

次のパッケージを使用して、このプロシージャを記述します。

```
CREATE OR REPLACE PACKAGE describe_it AS

    PROCEDURE desc_proc (name VARCHAR2);
```

```
END describe_it;

CREATE OR REPLACE PACKAGE BODY describe_it AS

  PROCEDURE prt_value(val VARCHAR2, isize INTEGER) IS
    n INTEGER;
  BEGIN
    n := isize - LENGTHB(val);
    IF n < 0 THEN
      n := 0;
    END IF;
    DBMS_OUTPUT.PUT(val);
    FOR i in 1..n LOOP
      DBMS_OUTPUT.PUT(' ');
    END LOOP;
  END prt_value;

  PROCEDURE desc_proc (name VARCHAR2) IS

      overload     DBMS_DESCRIBE.NUMBER_TABLE;
      position     DBMS_DESCRIBE.NUMBER_TABLE;
      c_level      DBMS_DESCRIBE.NUMBER_TABLE;
      arg_name     DBMS_DESCRIBE.VARCHAR2_TABLE;
      dty          DBMS_DESCRIBE.NUMBER_TABLE;
      def_val      DBMS_DESCRIBE.NUMBER_TABLE;
      p_mode       DBMS_DESCRIBE.NUMBER_TABLE;
      length       DBMS_DESCRIBE.NUMBER_TABLE;
      precision    DBMS_DESCRIBE.NUMBER_TABLE;
      scale        DBMS_DESCRIBE.NUMBER_TABLE;
      radix        DBMS_DESCRIBE.NUMBER_TABLE;
      spare        DBMS_DESCRIBE.NUMBER_TABLE;
      idx          INTEGER := 0;

  BEGIN
      DBMS_DESCRIBE.DESCRIBE_PROCEDURE(
              name,
              null,
              null,
              overload,
              position,
              c_level,
              arg_name,
              dty,
              def_val,
              p_mode,
              length,
              precision,
```

```
            scale,
            radix,
            spare);

      DBMS_OUTPUT.PUT_LINE('Position    Name        DTY  Mode');
      LOOP
          idx := idx + 1;
          prt_value(TO_CHAR(position(idx)), 12);
          prt_value(arg_name(idx), 12);
          prt_value(TO_CHAR(dty(idx)), 5);
          prt_value(TO_CHAR(p_mode(idx)), 5);
          DBMS_OUTPUT.NEW_LINE;
      END LOOP;
  EXCEPTION
     WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.NEW_LINE;
        DBMS_OUTPUT.NEW_LINE;

  END desc_proc;
END describe_it;
```

次のように、PL/SQL データ型のすべての数値コードが示されます。

```
Position  Name    Datatype_Code  Mode
1         PVC2      1              0
2         PVC       1              1
3         PSTR      1              2
4         PLONG     8              0
5         PROWID    11             0
6         PCHARA    96             0
7         PCHAR     96             0
8         PRAW      23             0
9         PLRAW     24             0
10        PBININT   3              0
11        PPLSINT   3              0
12        PBOOL     252            0
13        PNAT      3              0
14        PPOS      3              0
15        PPOSN     3              0
16        PNATN     3              0
17        PNUM      2              0
18        PINTGR    2              0
19        PINT      2              0
20        PSMALL    2              0
21        PDEC      2              0
22        PREAL     2              0
23        PFLOAT    2              0
24        PNUMER    2              0
```

```
25        PDP       2             0
26        PDATE     12            0
27        PMLS      106           0
```

プロシージャとパッケージに関する情報をリストする

次のデータ・ディクショナリ・ビューには、プロシージャとパッケージに関する情報が表示されます。

- ALL_ERRORS、USER_ERRORS、DBA_ERRORS
- ALL_SOURCE、USER_SOURCE、DBA_SOURCE
- USER_OBJECT_SIZE、DBA_OBJECT_SIZE

OBJECT_SIZE ビューは PL/SQL オブジェクトのサイズを示します。これらのデータ・ディクショナリ・ビューの詳細は、『Oracle8 Server リファレンス・マニュアル』を参照してください。

次の文は、例 1 から例 3 で使用します。

```
CREATE PROCEDURE fire_emp(emp_id NUMBER) AS
BEGIN
   DELETE FROM em WHERE empno = emp_id;
END;
/
CREATE PROCEDURE hire_emp (name VARCHAR2, job VARCHAR2,
   mgr NUMBER, hiredate DATE, sal NUMBER,
   comm NUMBER, deptno NUMBER)

IS
BEGIN
   INSERT INTO emp VALUES (emp_sequence.NEXTVAL, name,
      job, mgr, hiredate, sal, comm, deptno);
END;
/
```

最初の CREATE PROCEDURE 文では、DELETE 文にエラーがあります（「emp」の「p」がありません）。

例 1: オブジェクトのコンパイル・エラーのリスト 次の問合せでは、関係するスキーマ内のオブジェクトのエラーがすべて戻されます。

```
SELECT name, type, line, position, text
    FROM user_errors;
```

次の結果が戻されます。

```
NAME     TYPE LIN POS TEXT
-------- ---- --- --- -------------------------------------
FIRE_EMP PROC   3  15 PL/SQL-00201: identifier 'EM' must be
```

```
                                    declared
FIRE_EMP PROC 3 3 PL/SQL: SQL Statement ignored
```

例 2: プロシージャのソース・コードのリスト 次の問合せは、この項のはじめの例で作成した、HIRE_EMP プロシージャのソース・コードを戻します。

```
SELECT line, text FROM user_source
    WHERE name = 'HIRE_EMP';
```

次の結果が戻されます。

```
LINE   TEXT
------ ----------------------------------------------------
     1 PROCEDURE hire_emp (name VARCHAR2, job VARCHAR2,
     2    mgr NUMBER, hiredate DATE, sal NUMBER,
     3    comm NUMBER, deptno NUMBER)
     4 IS
     5 BEGIN
     6    INSERT INTO emp VALUES (emp_seq.NEXTVAL, name,
     7    job, mgr, hiredate, sal, comm, deptno);
     8 END;
```

例 3: プロシージャのサイズ情報のリスト 次の問合せは、HIRE_EMP プロシージャの保存に必要となる、SYSTEM 表領域のサイズに関する情報を戻します。

```
SELECT name, source_size + parsed_size + code_size +
      error_size "TOTAL SIZE"
    FROM user_object_size
    WHERE name = 'HIRE_EMP';
```

次の結果が戻されます。

```
NAME                           TOTAL SIZE
------------------------------ ----------
HIRE_EMP                             3897
```

DBMS_ROWID パッケージ

このパッケージの関数によって、ROWID に関する情報が得られます 。ROWID のデータ・ブロック番号およびオブジェクト番号、その他のコンポーネントが、base-64 キャラクタの外部 ROWID を変換することなく、得ることができます。

DBMS_ROWID パッケージの仕様部は、ファイル dbmsutil.sql 内にあります。このパッケージは、データベースを作成し、catproc.sql を実行するとロードされます。

このパッケージの関数の中には、単一のパラメータ、つまり単一の ROWID をとるものがあります。これは、キャラクタ ROWID または 2 進 ROWID のいずれかで、必要に応じて、制

限付きまたは拡張になる場合もあります。この項で説明する各関数については、そのパラメータ・タイプと戻り型の両方が記述されています。

PL/SQL コードから DBMS_ROWID 関数および DBMS_ROWID プロシージャをコールしたり、関数を SQL 文の中で使用したりできます。

注意： ROWID_INFO はプロシージャです。これは、PL/SQL コードでだけ使用できます。

SQL の例 DBMS_ROWID パッケージの関数は、組込み SQL 関数と同じように使用できます。これは、式を使用する場合にいつでも関数を使用できることを意味します。この例では、EMP 表内の単一行のブロック番号だけを戻すために、ROWID_BLOCK_NUMBER 関数が使用されています。

```
SELECT dbms_rowid.rowid_block_number(rowid)
  FROM emp
  WHERE ename = 'KING';
```

PL/SQL の例 この例では、EMP 表内の行の ROWID を戻し、DBMS_ROWID パッケージ内の ROWID_OBJECT 関数を使用して、ROWID からデータ・オブジェクト番号を取り出し、オブジェクト番号を表示しています。

```
DECLARE
  object_no   INTEGER;
  row_id      ROWID;
  ...
BEGIN
  SELECT ROWID INTO row_id FROM emp
    WHERE empno = 7499;
  object_no := dbms_rowid.rowid_object(row_id);
  dbms_output.put_line('The obj. # is '|| object_no);
  ...
```

サマリー

表 10-9 には、DBMS_ROWID パッケージ内の関数およびプロシージャがリストされています。

表 10-9　DBMS_ROWID 関数

関数名	説明	参照ページ
ROWID_CREATE	テスト用の ROWID だけを作成します。	77
ROWID_INFO	ROWID のタイプおよびコンポーネントを戻すプロシージャです。	77
ROWID_TYPE	ROWID 型を戻します。0 は制限付き ROWID、1 は拡張 ROWID です。	78
ROWID_OBJECT	拡張 ROWID のオブジェクト番号を戻します。	78
ROWID_RELATIVE_FNO	ROWID のファイル番号を戻します。	79
ROWID_BLOCK_NUMBER	ROWID のブロック番号を戻します。	79
ROWID_ROW_NUMBER	行番号を戻します。	79
ROWID_TO_ABSOLUTE_FNO	特定表内の行について、その ROWID に関連する絶対ファイル番号を戻します。	80
ROWID_TO_EXTENDED	制限付き ROWID を拡張フォーマットに変換します。	80
ROWID_TO_RESTRICTED	拡張 ROWID を制限付きフォーマットに変換します。	82
ROWID_VERIFY	ROWID を ROWID_TO_EXTENDED 関数によって正しく拡張できるかどうかをチェックします。	82

例外

DBMS_ROWID パッケージの関数およびプロシージャによって、ROWID_INVALID 例外が発生する場合があります。例外は、DBMS_ROWID パッケージに次のように定義されています。

```
PRAGMA EXCEPTION_INIT(ROWID_INVALID, -1410);
```

ROWID_CREATE 関数

ROWID_CREATE 関数では、コンポーネントの各部分がパラメータとして与えられた場合、ROWID を作成できます。この関数では、データベース内のデータを指し示す有効な ROWID を作成できるのは Oracle Server だけなので、主に ROWID の操作のテストに役立ちます。

構文

```
FUNCTION DBMS_ROWID.ROWID_CREATE(
    rowid_type        IN  NUMBER,
    object_number     IN  NUMBER,
    relative_fno      IN  NUMBER,
    block_number      IN  NUMBER,
    row_number        IN  NUMBER)
  RETURN ROWID;
```

制限付き ROWID の場合、ROWID_TYPE パラメータを 0 に設定し、拡張 ROWID を作成する場合は 1 に設定します。

ROWID_TYPE を 0 に指定した場合、必要となる OBJECT_NUMBER パラメータは無視され、ROWID_CREATE によって制限付きの ROWID が戻されます。

例

ダミーの拡張 ROWID を作成します。

```
my_rowid := DBMS_ROWID.ROWID_CREATE(1, 9999, 12, 1000, 13);
```

ROWID_OBJECT 関数から戻される値を確認します。

```
obj_number := DBMS_ROWID.ROWID_OBJECT(my_rowid);
```

可変 OBJ_NUMBER には 9999 が含まれています。

ROWID_INFO プロシージャ

このプロシージャによって、ROWID のタイプ（制限付きか拡張か）およびそのコンポーネントとともに、ROWID に関する情報が戻されます。これはプロシージャであるため、SQL 文では使用できません。

構文

```
DBMS_ROWID.ROWID_INFO(
    rowid_in          IN  ROWID,
    rowid_type        OUT  NUMBER,
    object_number     OUT  NUMBER,
    relative_fno      OUT  NUMBER,
    block_number      OUT  NUMBER,
    row_number        OUT  NUMBER);
```

IN パラメータである ROWID_IN によって、ROWID が制限付き（0）か、拡張（1）かどうかが判断されます。

OUT パラメータでは、その名前の示すとおり、ROWID に関する情報が戻されます。

ROWID_TYPE パラメータの詳細は、10-78 ページの「ROWID_TYPE 関数」を参照してください。

例

ROWID_CREATE で作成した ROWID の値を読み込むには、たとえば次のようにします。

```
DBMS_ROWID.ROWID_INFO(my_rowid, rid_type, obj_num,
  file_num, block_num, row_num);

DBMS_OUTPUT.PUT_LINE('The type is ' || rid_type);
DBMS_OUTPUT.PUT_LINE('Data object number is ' || obj_num);
-- and so on...
```

ROWID_TYPE 関数

この関数によって、ROWID が制限付き ROWID の場合は 0、拡張 ROWID の場合は 1 が戻されます。

構文

```
FUNCTION DBMS_ROWID.ROWID_TYPE(rowid_val IN ROWID)
  RETURN NUMBER;
```

例

```
IF DBMS_ROWID.ROWID_TYPE(my_rowid) = 1 THEN
  my_obj_num := DBMS_ROWID.ROWID_OBJECT(my_rowid);
```

ROWID_OBJECT 関数

この関数によって、拡張 ROWID のデータ・オブジェクト番号が戻されます。入力した ROWID が制限付き ROWID の場合は、0 が戻されます。

構文

```
DBMS_ROWID.ROWID_OBJECT(rowid_val IN ROWID)
  RETURN NUMBER;
```

例

```
SELECT dbms_rowid.rowid_object(ROWID)
  FROM emp
```

```
  WHERE empno = 7499;
```

ROWID_RELATIVE_FNO 関数

この関数によって、IN パラメータとして指定した、ROWID の相対ファイル番号が戻されます （ファイル番号は表領域に対して相対的です）。

構文

```
DBMS_ROWID.ROWID_RELATIVE_FNO(rowid_val IN ROWID)
  RETURN NUMBER;
```

例

サンプルの PL/SQL コード断片部分によって、相対ファイル番号が戻されます。

```
DECLARE
  file_number    INTEGER;
  rowid_val      ROWID;
BEGIN
  SELECT ROWID INTO rowid_val
    FROM dept
    WHERE loc = 'Boston';
  file_number :=
    dbms_rowid.rowid_relative_fno(rowid_val);
  ...
```

ROWID_BLOCK_NUMBER 関数

この関数によって、入力した ROWID のデータベース・ブロック番号が戻されます。

構文

```
DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid_val IN ROWID)
  RETURN NUMBER;
```

例

サンプルの SQL 文によって、ROWID からブロック番号が選択され、別の表へ挿入されます。

```
INSERT INTO T2 (SELECT dbms_rowid.rowid_block_number(ROWID)
  FROM some_table
  WHERE key_value = 42);
```

ROWID_ROW_NUMBER 関数

この関数によって、ROWID IN パラメータから行番号が取り出されます。

構文

```
DBMS_ROWID.ROWID_ROW_NUMBER(rowid_val IN ROWID)
  RETURN NUMBER;
```

例

行番号を選択します。

```
SELECT dbms_rowid.rowid_row_number(ROWID)
  FROM emp
  WHERE ename = 'ALLEN';
```

ROWID_TO_ABSOLUTE_FNO 関数

この関数によって、ROWID から絶対ファイル番号が取り出されますが、その場合、任意のスキーマおよび表内の行について、ファイル番号は絶対ファイル番号となります。スキーマ名およびスキーマ・オブジェクトの名前（表名など）は、この関数の IN パラメータとして提供されます。

構文

```
DBMS_ROWID.ROWID_TO_ABSOLUTE_FNO(
  rowid_val          IN  ROWID,
  schema_name        IN  VARCHAR2,
  object_name        IN  VARCHAR2)
      RETURN NUMBER;
```

例

```
DECLARE
  rel_fno        INTEGER;
  rowid_val      CHAR(18);
  object_name    VARCHAR2(20) := 'EMP';
BEGIN
  SELECT ROWID INTO rowid_val
    FROM emp
    WHERE empno = 9999;
  rel_fno := dbms_rowid.rowid_to_absolute_fno(
    rowid_val, 'SCOTT', object_name);
```

ROWID_TO_EXTENDED 関数

この関数によって、指定したスキーマおよび表内の行に対応する制限付き ROWID が、拡張 ROWID 形式に変換されます。

構文

```
DBMS_ROWID.ROWID_TO_EXTENDED(
```

```
    restr_rowid        IN ROWID,
    schema_name        IN VARCHAR2,
    object_name        IN VARCHAR2)
         RETURN ROWID;
```

例

スキーマ SCOTT 内に RIDS という名前の表があり、その表の中には、ROWID（制限付き）を持つ列、ROWID_COL、および SCOTT スキーマ内の他の表を参照する列、TABLE_COL が含まれると仮定します。次の文によって、ROWID を拡張フォーマットに変換できます。

```
UPDATE SCOTT.RIDS
  SET rowid_col =
  dbms_rowid.rowid_to_extended(rowid_col,
    'SCOTT", TABLE_COL);
```

使用方法

ROWID_TO_EXTENDED によって、拡張キャラクタ・フォーマットの ROWID が戻されます。入力した ROWID が NULL の場合、この関数によって NULL が戻されます。ゼロ値の ROWID が提供された場合（00000000.0000.0000）、ゼロ値の制限付き ROWID が戻されます。

スキーマ名およびオブジェクト名が IN パラメータとして提供された場合、この関数は、指定の表に対する SELECT 権限を検証し、表のデータ・オブジェクト番号を使用して、提供された制限付き ROWID を拡張 ROWID に変換します。ただし、この ROWID_TO_EXTENDED によって値が戻されても、変換された ROWID が、関数がコールされたときまたは拡張 ROWID が実際に使用されるときに、表内の有効な行を実際に参照するという保証はありません。

スキーマ名およびオブジェクト名が提供されない場合（NULL として渡される場合）、この関数は、提供された制限付き ROWID によって 指定されるページをフェッチしようと試みます。また、この ROWID に格納されたファイル番号は絶対ファイル番号として扱われます。ファイルが削除され、移行の前にその番号が再使用された場合などは、問題が発生する可能性があります。フェッチされたページが有効な表に属する場合、その表のデータ・オブジェクト番号は、拡張 ROWID 値への変換において使用されます。これは非常に非効率的であるため、ターゲット表がわからない場合に最後の手段として使用することをお薦めします。その場合でもユーザーは、変換した値を使用する時点で正しい表名を知っておかなければなりません。

拡張 ROWID 値が提供される場合、入力した拡張 ROWID 内のデータ・オブジェクト番号は、表名パラメータから計算されるデータ・オブジェクト番号に対して検証されます。2 つの数字が一致しない場合、INVALID_ROWID 例外が発生します。2 つの数字が一致する場合は、入力した ROWID が戻されます。

ROWID を拡張形式に変換できるかどうかを判別する方法は、10-82 ページの「ROWID_VERIFY 関数」を参照してください。

ROWID_TO_RESTRICTED 関数

この関数によって、拡張 ROWID が制限付き ROWID フォーマットに変換されます。

構文

```
DBMS_ROWID.ROWID_TO_RESTRICTED(ext_rowid IN ROWID)
RETURN ROWID;
```

例

```
INSERT INTO RID_T2@V7db1
  SELECT dbms_rowid.rowid_to_restricted(ROWID)
    FROM scott.emp@O8db1
    WHERE ename = 'SMITH';
```

ROWID_VERIFY 関数

この関数では、入力したスキーマ名および表名が与えられた場合、入力した制限付き ROWID が拡張形式に変換できる場合は 0、変換できない場合は 1 を戻します。この関数は、次の例に示すとおり、SQL 文内の BOOLEAN コンテキストで使用できます。

構文

```
DBMS_ROWID.ROWID_VERIFY(
  restr_rowid        IN ROWID,
  schema_name        IN VARCHAR2,
  object_name        IN VARCHAR2)
       RETURN ROWID;
```

例

10-80 ページの「ROWID_TO_EXTENDED 関数」の例にあるスキーマでは、変換前に不正な ROWID を検索するために次の文を使用できます。

```
SELECT ROWID, rowid_col
  FROM SCOTT.RIDS
  WHERE dbms_rowid.rowid_verify(rowid_col, NULL, NULL);
```

UTL_HTTP パッケージ

ストアド・パッケージ UTL_HTTP によって、PL/SQL および SQL からの HTTP（ハイパーテキスト転送プロトコル）コールアウトが作成されます。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に使用できます。パッケージには、2 つの類似したエントリ・ポイントがあります。各エントリ・ポイントには、文字列の URL（ユニバーサル・リソース・ロケータ）があり、これがそのサイトに接続してそのサイトから取得したデータ（一般に、HTML ―ハイパーテキスト・マークアップ言語）が戻されます。

次は、パッケージ関数 UTL_HTTP.REQUEST の仕様部です。

```
function request (url in varchar2) return varchar2;
```

UTL_HTTP.REQUEST により、指定した URL から取り出されたデータの最初の 2000 バイトまでが戻されます。たとえば、次のとおりです。

```
SVRMGR> select utl_http.request('http://www.oracle.com/') from dual;
UTL_HTTP.REQUEST('HTTP://WWW.ORACLE.COM/')
--------------------------------------------------------------------
<html>
<head><title>Oracle Corporation Home Page</title>
<!--changed Jan. 16, 19
1 row selected.
```

次は、型 UTL_HTTP.HTML_PIECES を使用するパッケージ関数 UTL_HTTP.REQUEST_PIECES の仕様部です。

```
type html_pieces is table of varchar2(2000) index by binary_integer;
function request_pieces (url in varchar2,
         max_pieces natural default 32767)
  return html_pieces;
```

UTL_HTTP.REQUEST_PIECES により、指定した URL から取り出されたデータの 2000 バイト分の PL/SQL 表が戻されます。オプションの 2 番目の引数により、取り出されるデータの個数が制限されます。たとえば、次のブロックでは、URL から 100 個までのデータ（最後のデータを除き、それぞれ 200 バイト）が取り出されます。取り出されたデータの個数および全体の長さ（バイト単位）が出力されます。

```
  set serveroutput on
  /
  declare
    x utl_http.html_pieces;
  begin
    x := utl_http.request_pieces('http://www.oracle.com/', 100);
    dbms_output.put_line(x.count || ' pieces were retrieved.');
    dbms_output.put_line('with total length ');
    if x.count < 1
    then dbms_output.put_line('0');
    else dbms_output.put_line
    ((2000 * (x.count - 1)) + length(x(x.count)));
    end if;
  end;
  /
```

次のように出力されます。

```
  Statement processed.
  4 pieces were retrieved.
  with total length
```

```
  7687
```

次は、パッケージ UTL_HTTP の仕様部です。これには、関数 REQUEST および REQUEST_PIECES によって発生する可能性がある例外が説明されています。

```
create or replace package utl_http is
-- Package UTL_HTTP contains functions REQUEST and REQUEST_PIECES for
-- making HTTP callouts from PLSQL programs.

-- Function REQUEST takes a URL as its argument.  Its return-type is a
-- string of length 2000 or less, which contains up to the first 2000 bytes
-- of the html result returned from the HTTP request to the argument URL.

function request (url in varchar2) return varchar2;
pragma restrict_references (request, wnds, rnds, wnps, rnps);

-- Function REQUEST_PIECES also takes a URL as its argument.  Its
-- return-type is a PLSQL-table of type UTL_HTTP.HTML_PIECES.  Each
-- element of that PLSQL-table is a string of length 2000. The
-- final element may be shorter than 2000 characters.

type html_pieces is table of varchar2(2000) index by binary_integer;

function request_pieces (url in varchar2,
      max_pieces natural default 32767)
  return html_pieces;
pragma restrict_references (request_pieces, wnds, rnds, wnps, rnps);

-- The elements of the PLSQL-table returned by REQUEST_PIECES are
-- successive pieces of the data obtained from the HTTP request to that
-- URL.  Here is a typical URL:
--     http://www.oracle.com
-- So a call to REQUEST_PIECES could look like the example below. Note the
-- use of the plsql-table method COUNT to discover the number of pieces
-- returned, which may be zero or more:
--
-- declare pieces utl_http.html_pieces;
-- begin
--   pieces := utl_http.request_pieces('http://www.oracle.com/');
--   for i in 1 .. pieces.count loop
--     .... -- process each piece
--   end loop;
-- end;
--

-- The second argument to REQUEST_PIECES, "MAX_PIECES", is optional.  It is
-- the maximum number of pieces (each 2000 characters in length, except for
```

```
-- the last, which may be shorter), that REQUEST_PIECES should return.  If
-- provided, that argument should be a positive integer.

-- Exceptional conditions:

-- If initialization of the http-callout subsystem fails (for
-- environmental reasons, for example, lack of available memory)
-- then exception UTL_HTTP.INIT_FAILED is raised:

init_failed exception;

-- When the HTTP call fails (e.g., because of failure of the HTTP daemon;
-- or because of the argument to REQUEST or REQUEST_PIECES cannot be
-- interpreted as a URL because it is NULL or has non-HTTP syntax) then
-- exception UTL_HTTP.REQUEST_FAILED is raised.

request_failed exception;

-- Note that the above two exceptions, unless explicitly caught by an
-- exception handler, will be reported by this generic message:
--   ORA-06510: PL/SQL: unhandled user-defined exception
-- which reports them as "user-defined" exceptions, although
-- they are defined in this system package.

-- If any other exception is raised during the processing of the http
-- request (for example, an out-of-memory error), then function REQUEST
-- or REQUEST_PIECES reraises that exception.

-- When no response is received from a request to the given URL
-- (for example, because no site corresponding to that URL is contacted)
-- then a formatted html error message may be returned.  For example:
--
--  <HTML>
--  <HEAD>
--  <TITLE>Error Message</TITLE>
--  </HEAD>
--  <BODY>
--  <H1>Fatal Error 500</H1>
--  Can't Access Document:  http://home.nothing.comm.
--  <P>
--  <B>Reason:</B> Can't locate remote host:  home.nothing.comm.
--  <P>
--
--  <P><HR>
--  <ADDRESS><A HREF="http://www.w3.org">
--   CERN-HTTPD3.0A</A></ADDRESS>
--  </BODY>
```

```
--  </HTML>
--

-- You should not expect for UTL_HTTP.REQUEST or UTL_HTTP.REQUEST_PIECES
-- to succeed in contacting a URL unless you can contact that URL by using
-- a browser on the same machine (and with the same privileges, environment
-- variables, etc.)  If REQUEST or REQUEST_PIECES fails (i.e., if it raises
-- an exception, or returns a HTML-formatted error message, yet you believe
-- that the URL argument is correct), please try contacting that same URL
-- with a browser, to verify network availability from your machine.

end utl_http;
```

11

アドバンスト・キューイング

この章では 4 つの項を説明します。

- Oracle アドバンスト・キューイングについて ー
  - 分散環境での複雑な情報処理の要件とキューイングの関係を説明します
  - Oracle AQ の機能を示します
  - Oracle AQ の基本コンポーネントについて説明します
- Oracle アドバンスト・キューイングの使用例 ー Oracle AQ を使うときのステップ・アプローチ
- Oracle アドバンスト・キューイングのリファレンス ー Oracle AQ のための技術的な仕様の詳細を説明します
- 互換性とアップグレード ー Oracle AQ 8.0.4 と Oracle AQ 8.0.3 の比較、最新のバージョンにアップグレードするのに必要なステップを説明します

警告 :

- **Oracle8 を購入された場合は、Oracle AQ を使用できません。**
- **「オブジェクト」オプションなしの Oracle8 Enterprise Edition を購入された場合は、RAW 型のキューでだけ Oracle AQ を使用できます。**
- **「オブジェクト」オプション付きの Oracle8 Enterprise Edition を購入された場合は、Oracle AQ のすべての機能を使用できます。**

Oracle アドバンスト・キューイングについて

概要

この項は導入部であり、内容は次のとおりです。

- 3 つのサンプル・シナリオで、分散環境での複雑な情報の処理の要件を紹介します
- 各シナリオ共通の問題に対する 2 つのソリューションを考察します
  - 同期通信モデル
  - Oracle AQ によって実現した遅延メッセージ・システム
- Oracle AQ の機能を説明します
  - 一般機能
  - エンキュー機能
  - デキュー機能
  - 伝播 : エンキューおよびデキュー
- Oracle AQ の基本コンポーネントの詳細を説明します
  - キュー・エンティティ
  - 基本キューイング
  - 同一メッセージの複数コンシューマ・デキュー

複雑なシステム

次のようなアプリケーション・シナリオを想定します。

アプリケーション・シナリオ 1: 証券会社 Makers & Breakers は、クライアントが価格のパラメータに加えて時間を要求できることを新しいサービスとして広告しています。つまり、売り買いは特定の時間内（たとえば 15 分以内）に発生しないかぎり履行されません。このキャンペーンは大成功を収め、投資信託会社 America’s Standard Guarantee はその技術を利用して、交換の終了時ではなく日中に単位の売買を行う機会をクライアントに提供しています。ただし、M&B は、証券取引委員会が次の 2 点の苦情を受け取ったことを通知されます。

- 売り / 買い注文に応じる際に、M&B が投資信託会社などの大規模な顧客に不公平に利益を提供している。
- 時間パラメータを管理する際に、M&B が顧客につけこんで、市場が下落しているときに、時間内でクライアントに通知するよりも早い時間に売却を行い、差額を着服している。

Makers & Breakers が直面している問題は、これらの根拠のない嫌疑にただ答えるのではなく、営業の公正さに関して社会的に疑念を残さない方法で素早く答えるということです。

アプリケーション・シナリオ 2: 大規模な州立大学（学生数 35,000 名）で受講手続きを自動化することを決定しました。学生は、自宅で Web テンプレートを使って、または大学の本校や分校の端末を使ってその大学が提供する数千の講座の受講登録をできるようになります。管理者は次のパラメータを適用することを通知します。

- 優先順位 : 登録は先着順に行われますが、次の例外があります。

- 上級レベルの講座では、4 年生、3 年生、2 年生、1 年生の順に優先する。

- 入門レベルの講座では、1 年生を優先し、卒業のためにその講座が必要な 4 年生を次に優先する。

- 登録フェーズ : 上記の優先順位は定義済みの特定の時間フェーズだけで使われます。つまり、別のフェーズでは、4 年生と 3 年生は、上級レベルの講座に関しては等しく先着順に処理されますが、2 年生と 1 年生に対しては優先されます。
- 全日制の学部生は、少なくとも 3 講座に登録する必要があり、特別な許可がなくても最大 4 講座に登録できます。学生は、希望講座に登録できない場合に備えて、10 講座まで（1 ～ 10 まで優先度を付けて）登録申込みできますが、第 3 希望の講座までが「有効」とされます。ある講座が満員になると、その学生の次の希望講座が「有効」になります。ただし、満員の講座に欠員ができると、一番高い優先順位の学生がその講座に登録されます。同時に、その学生の優先度が一番低い講座の登録が取り消されます。

アプリケーション・シナリオ 3:

電力会社 Most Power は、資源の供給方法を決定するための高度なモデルを開発しています。このシステムは、この電力会社が、次のような報告を継続して得ることによって稼動します。

- 現在の状況に関する複数の気象予報センターからの報告
- 進行中の使用状況に関する発電所からの報告

この情報を過去のデータと比較して、特定の時間内の特定の地域における電力需要を予測します。このモデル化の重要な部分は、天候の変化と電力の供給を示す報告における迅速性や変化の程度とはまったく関係しません。

長期化する嵐の際には、発電所の運転停止によって問題が複雑になり、気象データも転送されます。Most Power が直面する問題は、近接する電力会社から電力を購入するかどうかということです。購入の手配は 5 日前には計画しておかなくてはなりません。

メッセージ受渡しの複雑さとキューイング

すべてのアプリケーションの開発者がこれらのタイプのシナリオで作業するわけではありませんが、これらの問題の基本要素はよくあるものです。これらのシナリオはそれぞれ、分散コンピューティング環境において、メッセージが発生する状況と、メッセージが複数のクライアント（ノード）に分配される状況を示しています。メッセージはクライアントとサーバーの間で受け渡されるだけではなく、サーバー上のプロセス間で複雑に割り込まれます。

これらのシナリオのメッセージに注目してみると、大規模アプリケーションは、複数ステップのプロセスで構成されていると見ることができます。各ステップは 1 つ以上のメッセージによって引き起こされ、1 つ以上のメッセージを発生させます。言い換えると、メッセージは他のメッセージイベントを引き起こすイベントです。

上記のようなメッセージとイベントの相互関係の認識を基盤とする、業務処理管理またはワークフローは、アプリケーションを開発するための基礎的な技術としてますます認識されてきました。キューイングはメッセージの遅延実行をインプリメントするため、このようなアプリケーションにとって重要なテクノロジの 1 つです。このように「サービスの要求」と「サービスの提供」を切り離すことによって効率が高くなり、複雑なスケジューリングのためのインフラストラクチャが供給されます。

不安定な環境でのメッセージの保護

問題は複雑なメッセージの受渡しだけではありません。あいにく、電力会社のシナリオでは、ネットワークおよびコンピュータ・ハードウェア、ソフトウェア・アプリケーションのすべてにときどき障害が発生します。それにもかかわらず、情報の ACID プロパティを保存する必要があります。買い注文が「紛失」したり、学生の変更状況が講座の登録状況と一致しなかったり、電力が必要とする地域に配分されなかったりするとすぐに混乱が起きることになります。つまり、メッセージ交換には持続性が必要です。トランザクション処理と

キューイング・テクノロジを統合することによって、キューイングという形式での持続メッセージ交換が可能になりました。キューイングの重要性は、そのような機能を通常備えている TP モニターによって証明されています。

メッセージの持続性の時間的および空間的拡張

必要とされるメッセージの持続性は、システム障害が起きたときに情報を回復するための機能を上回ります。アプリケーションは、外部クライアントまたは内部プログラムから同時に到着する複数の未処理のメッセージを処理する必要がある場合があります。これらのデータベース間の通信リンクは常に使用できるわけではなく、または他の目的のために確保されている可能性もあります。システムにこれらのメッセージを即座に処理する容量が不足している場合、アプリケーションはメッセージが処理可能になるまで保存しておく必要があります。同様に、外部クライアントまたは内部プログラムが、処理済みのメッセージを受信する準備ができていないこともあります。

さらに重要なこととして、アプリケーションが優先順位の変更を処理できる必要があります。後から到着したメッセージが先に到着したメッセージよりも高い優先順位を持つ場合、先に到着したメッセージが後のメッセージを待ってからアクションを実行する場合、同一のメッセージが別のプロセスからアクセスされなければならない場合などです。これらの場合はすべて、メッセージがリモートの場所の間で通信されている状況でより重大になります。

さらに、優先順位は固定されていません。メッセージ持続性の動的な面を処理するときの 1 つの重要な点は、拡大および縮小する機会の枠をどうするかということです。特定のキューにあるメッセージは、他のキューにあるメッセージより重要になるので、遅延しないようにまたは他のキューのメッセージの介入を受けることのないよう処理される必要があります。同様に、メッセージは、ある宛先への送信が他の宛先への送信よりも重要である場合があります。株式仲買アプリケーションの場合は、売りを完了するための枠は、売申込みのメッセージを受信したときから縮小し始め最後はなくなります（つまり、売申込みの期限切れ）。学生登録アプリケーションの場合は、さまざまな時間フェーズで異なる優先順位を適用し、データを 1 つのフェーズから別のフェーズに移すときに再評価する必要があります。電力会社の場合は、意思決定プロセス全体が、気象条件が安定しているか（大きな枠の持続）、変化に富んでいるか（枠の目まぐるしい出現と消失）に依存しています。

必須情報としての制御データ

すべてのシナリオで言えるのは、メッセージが受信またはディスパッチされた時間がメッセージの重要な部分であるということです。これは、メッセージの制御コンポーネント（この場合はタイム・マーカー）が メッセージの内容データと同様に重要であるという意味です。つまり、メッセージは実行された後で業務資産としての重要性を得ます。

このように、持続メッセージ交換には、過去の傾向と将来の動向を分析するためにメッセージを正確に文書化するという意味も含まれます。たとえば、次のとおりです。

- メッセージの順序を検索する機能は、第 1 のシナリオで証券会社が SEC から受けた疑惑に反論するためになくてはならないものです。証券会社は、`client_A` による売申込みが `client_B` による最初の有効な買申込みに合致していることを示す必要があります。

- 学生の登録の場合は、満員の講座から 1 名の学生を取り消すには次のことが必要です。(1) 優先順位および申込み日時、希望順に基づいて、次に権利がある学生を判断する。(2) その学生を登録している講座から外して、欠員が出た講座に入れる。(3) 伝播効果を処理する、つまり、メッセージ間の関係を追跡し、問合せに基づいてあるメッセージから別のメッセージにナビゲートする。
- 電力会社の場合は、気象および電力利用についてのメッセージは、メッセージ・ウェアハウスを問い合せて傾向を分析できるようにしばらく保存する必要があります。電力会社が特に関心を持つのは次のようなイベント間の経過時間です。
  - 電力を必要とする地域を識別してから電力の分配までの時間
  - 電力を分配するためのメッセージを送信してから実際に電力が分配されるまでの時間

メッセージ・システムの特定要件

前述の問題を考える場合の持続メッセージ・システムの重要な要件を次に示します。

- ますます複雑になるアプリケーションを扱うためには、開発者は、ACID プロパティを保存しながら、メッセージの受信、保持、分配を行う機能を提供するメカニズムが必要です。これらのメッセージは、ネットワーク内のクライアントとサーバー上のプログラム間や、サーバー上のプログラム間で通信できなければなりません。
- キューイング・システムは、メッセージが制御情報と内容を結合できるように、統合されたソリューションを提供する必要があります。理想的なソリューションでは、メッセージのキューイングができ、他のメッセージを引き起こすイベントとしてキューイング中のメッセージを扱えなければなりません。
- プログラムの最適化には、キューイング・システムとデータベースが同一の資源管理プログラムを共有し、それによって 2 フェーズ・コミットのオーバーヘッドの発生を回避していることが必要です。
- メッセージ・システムは、次の測定基準で測定して高性能の特性を示す必要があります。
  - 1 秒間にエンキュー / デキューされるメッセージ数
  - メッセージ・ウェアハウスに対する複雑な問合せの評価にかかる時間
  - 障害後にメッセージ処理の回復 / 再起動にかかる時間
- メッセージ・システムは高拡張性を示す必要があります。つまり、アプリケーションを使うプログラム数が増加しても、メッセージ数が増加しても、またメッセージ・ウェアハウスのサイズが増加しても、システムは引き続き高性能を示す必要があります。
- メッセージの内容（データ・ペイロード）と制御ディメンションが 両方とも使用可能でなければなりません。たとえば、アプリケーションで内容に基づくルーティング、内容に基づく予約、内容に基づく問合せをインプリメントできなければなりません。
- 追跡と文書化は、開発者ではなくメッセージ・システムによって処理されなければなりません。

実現可能なソリューション : 同期通信と切断 / 遅延通信

一般に、プログラム間の通信を提供する試みは 2 種類に分類できます。同期通信と切断 / 遅延通信です。

同期通信

この通信モデルは、「オンライン」または「接続」とも呼ばれ、要求 / 応答パラダイムに基づいています。このモデルでは、プログラムは別のプログラムに要求を送信し、応答が届くまで待機します （ブロックします)。この通信モデルは、メッセージの送信側と受信側が密接に結合しているため、なんらかのタスクを処理する前に応答を得る必要があるプログラムに適しています。伝統的なクライアント / サーバー・アーキテクチャはこのモデルに基づいています。

同期通信モデルの主な欠点は、アプリケーションが作動するためにプログラムが使用可能かつ実行中でなければならないことです。ネットワークまたはマシンに障害が発生した場合、または必要なプログラムがビジー状態の場合、アプリケーション全体が一時停止します。

切断 / 遅延メッセージ交換

このモデルでは、生産者役のプログラムがキューに要求を入れてから処理を開始します。消費者役のプログラムはキューから 要求を取り出して、それを処理します。このモデルは、応答を待機してブロックされないので、要求をキューに入れた後でも処理を続けられるアプリケーションに適しています。また、取り出すメッセージが発生するまで処理を続けられるアプリケーションにも適しています。

ネットワークおよびマシン、アプリケーションに障害が発生したときでも遅延実行を正常に動作させるためには、要求を持続して保存し、正確に一度だけ処理する必要があります。これは、持続キューイングをトランザクション保護と組み合わせることで達成できます。Oracle8 では、TP モニターやその他のメッセージ指向ミドルウェア (MOM) インフラストラクチャの使用に依存しないキューイング・テクノロジが提供されます。

Oracle アドバンスト・キューイング ー 機能

Oracle AQ(Oracle アドバンスト・キューイング) では、Oracle Server の統合部分としてメッセージ・キューイングが提供されます。Oracle AQ では、キューイング・システムをデータベースと統合することでこの機能が提供され、メッセージ使用可能データベースが作成されます。統合ソリューションを提供することで、Oracle AQ によってアプリケーション開発者は、メッセージ交換インフラストラクチャを組み立てる必要がなくなり、特定の業務ロジックに力を注ぐことができるようになりました。

一般機能

- **SQL アクセス :** メッセージはデータベース表の通常の行に入れられます。それらは標準 SQL を使って問い合せられます。つまり、ユーザーは SQL を使って、メッセージのプロ

パティ、履歴、ペイロードにアクセスできます。索引など、使用可能なすべての SQL テクノロジを使って、メッセージへのアクセスを最適化できます。

- **統合データベース・レベルの操作サポート :** 回復、再起動、Enterprise Manager など、すべての標準データベース機能がサポートされます。Oracle AQ のキューはデータベース表にインプリメントされています。このため、高い可用性および拡張性、信頼性を備えた操作上のすべての利点が、キュー・データにも適用されます。さらに、データベース開発ツールや管理ツールをキューにも使えます。たとえば、キューの表のインポートおよびエクスポートができます。
- **構造化されたペイロード :** ユーザーはオブジェクト型を使って、ペイロードを構造化して管理できます。一般に RDBMS は、メッセージ・システムよりもはるかに豊富な型指定のシステムを備えています。Oracle8 はオブジェクトリレーショナル DBMS であり、伝統的なリレーショナル型とユーザー定義型を両方ともサポートします。具体的に型指定された内容（形式が外部の型指定システムによって定義されている）を持つことによって、多数の高性能な機能が利用可能になりました。次のものが含まれます。
  - 内容に基づくルーティング : 外部エージェントが内容を調査し、その内容に基づいてメッセージを別のキューにルーティングできます。
  - 内容に基づく予約 : 発行および予約システムがメッセージ・システム上に組み込まれており、予約に基づいた内容を提供できます。
  - 問合せ : メッセージの内容に問合せを実行できる機能により、メッセージ・ウェアハウスが使用可能です。
- **保存とメッセージ履歴 :** 使った後もメッセージを保存しておくようにユーザーが指定できます。システム管理者が、保存するメッセージと保存期間を指定できます。Oracle AQ では、各メッセージの履歴、遅延のキューとメッセージのプロパティの保存、時間切れ、ローカルまたはリモートの受信者に対するメッセージの保存についての情報も格納されます。この情報には、ENQUEUE/DEQUEUE 時刻と、各要求を実行したトランザクションの識別子が含まれます。これによって、ユーザーは関係のあるメッセージの履歴を保持できます。履歴は、追跡およびデータ・ウェアハウス、データ・マイニング操作で使われます。
- **追跡とイベント・ジャーナル :** メッセージが保存されている場合、それらは互いに関連付けられます。たとえば、メッセージ m2 が メッセージ m1 を使った結果として生成された場合、m1 は m2 に関連付けられます。これによって、ユーザーは関連するメッセージの順序を追跡できます。これらの順序は「イベント・ジャーナル」を表しており、しばしばアプリケーションによって作成されます。Oracle AQ では、アプリケーションが自動的にイベント・ジャーナルを作成するように設計されています。
- **統合されたトランザクション :** 制御情報と内容（データ・ペイロード）の統合により、アプリケーション開発および管理が単純化されました。

ENQUEUE 機能

- **相関識別子 :** ユーザーが各メッセージに識別子を割り当てられるので、後で特定のメッセージを検索する手段ができます。

- **サブスクライドと宛先リスト :** 単一のメッセージが複数の コンシューマで使われるように設計できます。キュー・アドミニストレータによって、キューからメッセージを取り出せるサブスクライバのリストが指定されます。さまざまなキューがさまざまなサブスクライバを持つことができ、1 つのコンシューマ・プログラムは 1 つ以上のキューのサブスクライバになることもできます。さらに、キューの特定のメッセージを特定の受信者宛てにできます。この受信者はこのキューのサブスクライバであってもなくてもかまいません。つまり、サブスクライバ・リストを上書きできます。
- **エンキューでのメッセージの優先順位と順序付け :** エンキューされたメッセージの優先順位を指定できます。また、エンキューされたメッセージは、指定されたキュー内で正確な位置に入れられます。つまり、メッセージが使われる順序をユーザーが指定するには次の 3 つのオプションがあります。(a) ソート順では、キューのすべてのメッセージを順序付けするために使うプロパティを指定します。(b) 優先順位をメッセージごとに割り当てられます。(c) 順序逸脱では、メッセージの位置を他のメッセージとの関連で指定できます。さらに、複数のコンシューマが同一のキューに作用している場合、1 つのコンシューマがすぐに使える最初のメッセージを取得します。別のコンシューマで使用処理中のメッセージはスキップされます。
- **メッセージのグループ化 :** 1 つのキューに所属しているメッセージをグループ化して 1 つのセットにし、一度に 1 ユーザーだけしか使えないようにできます。このためには、メッセージのグループ化を使用可能にしたキュー表にキューを作成する必要があります。1 つのグループに所属するすべてのメッセージは、同一のトランザクションで作成される必要があります。また、1 つのトランザクションで作成されるすべてのメッセージは同一のグループに所属します。この機能を使って、複雑なメッセージをユーザーがセグメント化できます。つまり、あるキュー宛てのメッセージに請求書情報が含まれている場合、そのメッセージは、ヘッダーのメッセージ、詳細情報のメッセージ、フッターのメッセージで構成されるグループとして作成できます。
- **伝播** : この機能により、同じデータベースまたは同じキューに接続しなくてもアプリケーション同士が互いに通信できます。メッセージは、ローカルまたはリモートに関わらず、ある Oracle AQ から別の Oracle AQ に伝播できます。伝播は、一般的なデータベース・リンクおよび Net8 を使用して行われます。
- **時間の指定およびスケジューリング :** エンキューされたメッセージに対して遅延間隔または時間切れ間隔（あるいはその両方）を指定して、実行の枠を提供できます。メッセージをマークして、指定した時間（遅延時間）が経過しないと処理できないようにしたり、指定した制限時間になる前に処理されなければならないようにしたりできます。メッセージは、キューからローカルまたはリモートの宛先に伝播するようスケジュールできます。管理者は、開始時間、および伝播枠、次の伝播枠（定期的なスケジュールの場合）を決定する関数を指定します。

DEQUEUE 機能

- **複数の受信者 :** 同一メッセージの複数のコピーがなくても、複数の受信者がキューにある 1 つのメッセージを取り出せます。

- **ローカルおよびリモートの受信者** : 指定受信者をローカルまたはリモート（あるいはその両方）のサイトに設定できます。
- **デキューでのメッセージのナビゲーション :** キューからメッセージ を選択するにはいくつかのオプションがあります。最初のメッセージを選択するか、いったんメッセージを選択して位置を設定してから次の メッセージを選択できます。選択肢は順序付けに影響されたり、相関識別子を指定することで限定されることもあります。また、メッセージ識別子を使って特定のメッセージを取り出せます。
- **デキューのモード :** デキュー要求によってメッセージのブラウズまたは取消しが行えます。メッセージがブラウズされても、そのメッセージは後で処理可能です。メッセージが取り消されると、そのメッセージはデキュー要求に対して使用可能ではなくなります。キューのプロパティによっては、取り消されたメッセージもキュー表に保存されることがあります。
- **メッセージ到着待機の最適化 :** 空のキューに対してデキューを発行できます。新しいメッセージの到着についてポーリングを回避するために、要求がメッセージ到着を待機できるようにするかどうか、およびその場合の待機時間をユーザーが指定できます。
- **遅延による再試行 :** メッセージは、必ず 1 回だけ使用されなければなりません。メッセージのデキューに失敗し、トランザクションがロールバックされた場合、何人かのユーザーが指定した遅延が終了すると、メッセージの 再処理が可能になります。再処理は、ユーザーが指定した回数まで試行されます。
- **オプションのトランザクション保護 :** エンキュー / デキュー要求は、通常は複数の要求を含むトランザクションの一部で、それによって必要なトランザクション動作を実現しています。しかしユーザーは、特定の要求を 1 つのトランザクションに指定して、その要求の結果を即時に他のトランザクションから参照できるようにできます。これによって、エンキューまたはデキュー文が発行されるとただちに、またはそのトランザクションがコミットされた後にだけ、メッセージを外部から参照できるようにできます。
- **例外処理 :** メッセージは、たとえば実行時間中や再試行回数の制限内にある場合などの、一定の制約が当てはまる場合には、使用されてはいけません。このような制約に反する状況が発生すると、メッセージはユーザーが指定した例外キューに移動されます。

伝播機能

- **エンキューとデキューの自動調整** : 前述のとおり、受信者をローカルまたはリモートに設定できます。Oracle8.0.4 では、分散オブジェクト型はサポートされていないため、標準データベースリンクを使用したリモートでのエンキューまたはデキューは正常に処理されません。ただし、Oracle 8.0.4 では、AQ のメッセージ伝播を使用してリモートのキューにエンキューできます。

  たとえば、データベース X に接続して、そこにある "DROPBOX" キューのメッセージにエンキューできます。また、AQ を構成して、"DROPBOX" キューにエンキューされたすべてのメッセージを、データベース Y(ローカルまたはリモートに関わらず) の別のキューに自動的に伝播できます。AQ により、データベース Y のリモート・キューのタイプが、データベース X のローカルキューのタイプと構造的に同一であるか自動的にチェックされ、メッセージが伝播されます。

伝播されたメッセージの受信者は、アプリケーションまたはキューのどちらかです。受信者がキューである場合には、実際の受信者は、受信者キューの受信者リストによって判断されます。キューがリモートの場合には、メッセージは指定したデータベース・リンクを使用して伝播されます。AQ から AQ へのメッセージの伝播だけがサポートされています。

Oracle アドバンスト・キューイング – 主要コンポーネント

キューイング・エンティティ

メッセージ メッセージは、キューに挿入され、またそこから取り出される情報の最小単位です。メッセージは、制御情報（メタデータ）および ペイロード（データ）から構成されます。制御情報は、AQ がメッセージを管理するために使うメッセージ・プロパティを表示します。ペイロード・データはキューに格納された情報で、Oracle AQ に対して透過的です。1 つのメッセージは、1 つのキューにだけ常駐できます。メッセージは、エンキューのコールによって作成され、デキューのコールで使用されます。

キュー キューは、メッセージのリポジトリです。キューには、ユーザー・キュー（通常キューとも呼ばれる）と、例外キューの 2 種類があります。ユーザー・キューは、通常のメッセージ処理に使われます。なんらかの理由で取り出して処理できないメッセージは、例外キューに転送されます。キューは、「Oracle AQ 管理インタフェース」を使うことで作成および変更、起動、停止、削除できます。

キュー表 キューは、キュー表に格納されます。キュー表はデータベースの表で、1 つ以上のキューが含まれています。それぞれのキュー表には、デフォルトの例外キューが 1 つずつ含まれています。

メッセージおよびキュー、キュー表の関係を次の図に示します。各列はメッセージキューを、各行は個々のメッセージを表します。

エージェント エージェントはキューのユーザーです。エージェントには、メッセージをキューに入れる（エンキューする）プロデューサと、メッセージを取り出す （デキューする）コンシューマの 2 種類があります。任意の時点でキューに アクセスできるプロデューサおよびコンシューマの数に制限はありません。エージェントは、「Oracle AQ 操作インタフェース」から操作することで、メッセージをキューに挿入したり、取り出すことができます。

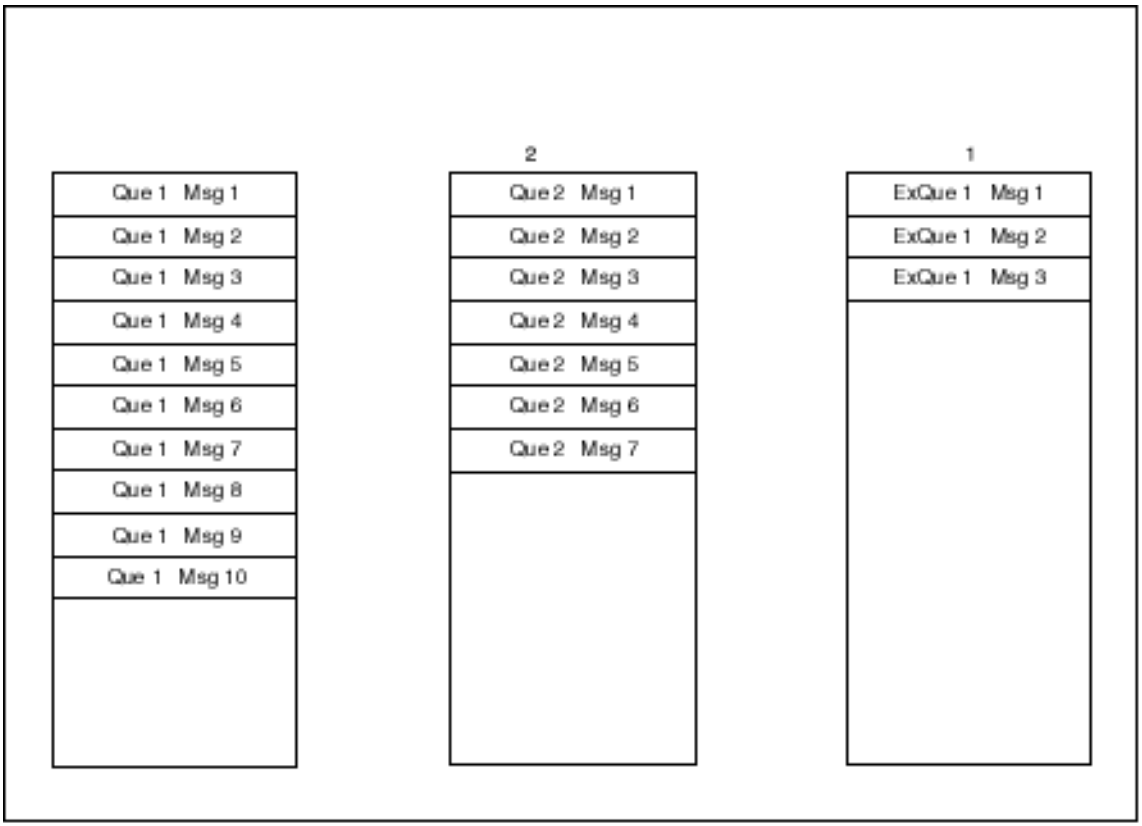
エージェントは、名前およびアドレス、プロトコルで識別されます。アドレス・フィールドは、最大 1024 バイトの文字フィールドであり、プロトコルのコンテキストで解釈されます。たとえば、プロトコルのデフォルト値は 0 で、データベース・リンクのアドレスを示します。この場合、このプロトコルのアドレスは、次のようになります。

```
queue_name@dblink
```

ここで queue_name の形式は [schema.]queue で、dblink は完全に修飾されたデータベース・リンク名またはドメイン名がないデータベース・リンク名のどちらかです。このとき唯一サポートされているプロトコル値は 0 です。

キュー・モニター キュー・モニターは、キュー内のメッセージをモニターするバックグラウンド・プロセスです。キュー・モニターでは、メッセージの時間切れおよび再試行、遅延のための機能を提供します。

図 11-1　キュー・エンティティ・モデル



キュー・エンティティ・モデル

図 11-1(上記) に、2 つのキューと 1 つの例外キューを含むキュー表を示します。

- キュー 1 － 10 個のメッセージを含む。
- キュー 2 － 7 個のメッセージを含む。
- 例外キュー 1 － 3 個のメッセージを含む。

基本キューイング

基本キューイング － プロデューサ、コンシューマがともに 1 つの場合

最も基本的な場合として、1 つのプロデューサが、複数の異なるメッセージを 1 つのキューにエンキューする場合が考えられます。各メッセージは、1 つのコンシューマによって 1 回だけデキューされ、処理されます。メッセージは、コンシューマによってデキューされるか、または有効期限が切れるまでは、キュー内に格納され続けます。プロデューサでは、メッセージが使用可能になるまでの遅延およびメッセージの有効期限を指定できます。同じように、使用可能なメッセージが存在しない場合には、コンシューマではデキューの試行を待機できます。エージェント・プログラムまたはアプリケーションは、プロデューサとコンシューマの両方として動作できる点に注意してください。

基本キューイング － プロデューサが複数で、コンシューマが 1 つの場合

やや複雑な場合として、多数のプロデューサがメッセージを 1 つのキューにエンキューし、そのすべてのメッセージが 1 つのコンシューマによって処理される場合があります。

基本キューイング － 離散的メッセージのプロデューサとコンシューマがともに複数の場合

次のこのステージでは、多数のプロデューサがメッセージをエンキューし、それぞれのメッセージを型および相関する識別子によって異なるコンシューマが処理する場合を考えます。次の図で、このシナリオを説明します。

基本キューイングの説明

図 11-2(下図) に、メッセージがエンキューおよびデキューされる 1 つのキューを含むキュー表を示します。

プロデューサ

図ではメッセージのプロデューサが 6 つあり、そのうちの 4 つだけが示されています。つまり、他の 2 つのプロデューサ (P4 と P5) は、メッセージをエンキューする権利はあるけれども、図に示された時点ではメッセージをエンキューしていないと仮定しています。図では、次のことを示しています。

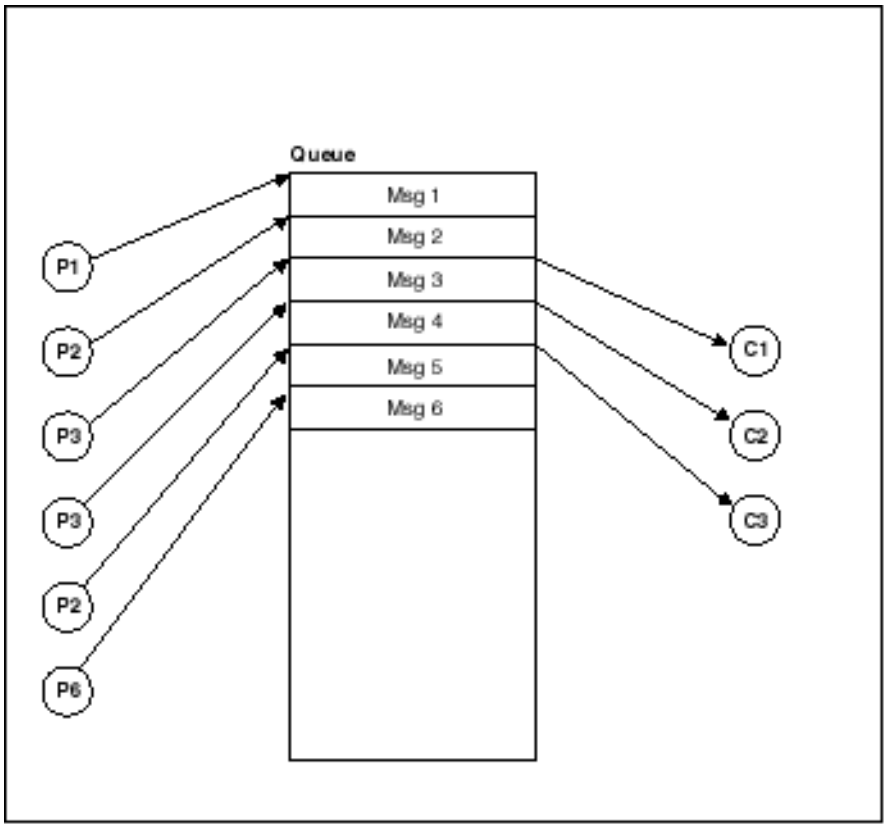
- 1 つのプロデューサが 1 つ以上のメッセージをエンキューできる。
- プロデューサは、メッセージをどの順序でもエンキューできる。

コンシューマ

図では、メッセージのコンシューマが 3 つあり、コンシューマはこれですべてです。図では、次のことを示しています。

- メッセージは、デキューされた順序でエンキューする必要はない。
- メッセージは、デキューされなくてもエンキューできる。

図 11-2　基本キューイング・モデル

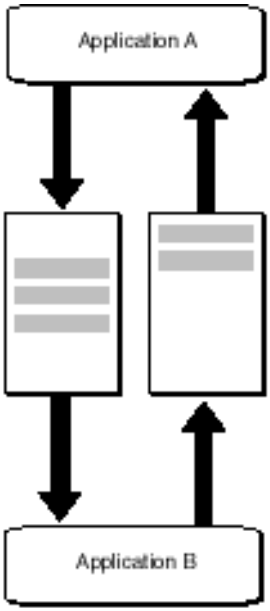


AQ を使用したクライアントサーバー通信の例

前述の図では、一連のプロデューサにより複数のメッセージがエンキューされ、一連のコンシューマによりメッセージがデキューされることを示しました。この図で明らかにされていないのは、時間の概念と Oracle AQ による利点です。

クライアントサーバー・アプリケーションは、通常、同期方式で実行され、上述の緊密な結合のための不利な点があります。図 11-3 では、AQ を使用して非同期で処理する方法を示しています。この例では、「Application B」(サーバー) は、要求 / 応答キューを使用して、「Application A」(クライアント) にサービスを提供しています。

図 11-3　AQ 使用したクライアントサーバー通信



1. 「Application A」は、要求を要求キューにエンキューする。
2. 「Application B」は、要求をデキューする。
3. 「Application B」は、要求を処理する。
4. 「Application B」は、応答キューにある結果をエンキューする。
5. 「Application A」は、応答キューにある結果をデキューする。

このように、クライアントはサーバーに対する接続を確立するために待つ必要はなく、サーバーでは、独自の処理タイミングでメッセージをデキューします。サーバーのメッセージの処理が終了したときに、クライアントは結果を受け取るために待つ必要はありません。このような二重遅延処理により、クライアントとサーバーの両方が自由に処理できるようになります。

注意： 様々なエンキューおよびデキューの操作は、異なるトランザクションの 1 部です。

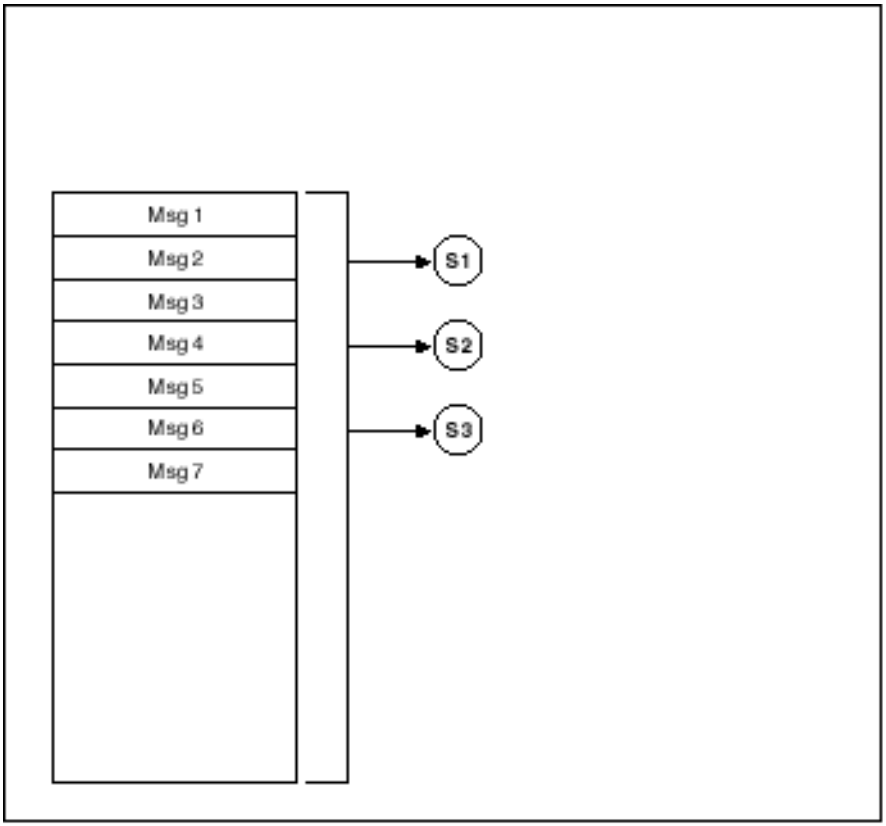
同一メッセージの複数コンシューマ・デキュー

メッセージは、1 度に 1 つのキューにだけ入れることができます。複数のコンシューマに届けるために、プロデューサから同じメッセージを複数のキューに挿入しなければならない場合には、非常に多くのキューを管理する必要が生じます。Oracle AQ では、複数のコンシューマから同じメッセージをデキューできるようにするため、「キューのサブスクライバ」および「メッセージの受信者」という 2 つの機能を用意しています。これを利用する場合、キューを常駐させるキュー表は、サブスクライバおよび受信者リストの余裕を取るために、複数コンシューマ・オプションを使って作成しなければなりません。各メッセージは、すべての所定のコンシューマによって使用されるまで、キュー内にとどまります。

キューのサブスクライバ このアプローチを使うことで、複数のコンシューマ・サブスクライバを 1 つのキューに対応付けることができます。これによって、キューに入れられたすべてのメッセージを、それぞれのキューのサブスクライバから使用できるようにできます。キューに対応するサブスクライバは、メッセージまたはメッセージ・プロデューサを一切変更することなしに、動的に変更できます。Oracle AQ 管理パッケージを使うことで、キューに対応するサブスクライバを追加および削除できます。次の図で、メッセージが複数のプロデューサによってキューに入れられ、それぞれのメッセージが複数のコンシューマ・サブスクライバによって使用される様子を説明します。

メッセージの受信者 メッセージ・プロデューサでは、メッセージがエンキューされた時点での受信者のリストを送付できます。これによって、キュー内の各メッセ ージに、一意となる受信者の集合を対応付けることができます。メッセージに対応付けられた受信者リストは、キューに対応付けられたサブスクライバ・リストが存在する場合には、それを上書きします。受信者は、サブスクライバ・リストに含まれている必要はありません。ただし、受信者はサブスクライバの中から選択できます。

図 11-4　同一メッセージの複数コンシューマによるデキュー



同一メッセージの複数コンシューマによるデキューの例

図 11-4 では、3 つのコンシューマがキューのサブスクライバとしてすべてリストされている場合について説明します。これは、3 つのコンシューマが、そのキューにエンキューされるすべてのメッセージを待機しているということです。図は、いくつかの重要な点を示しています。

- 図では、3 つのコンシューマが、すでにエンキューされている 7 つのメッセージに対するサブスクライバであり、まだエンキューされていないメッセージのサブスクライバになる可能性がある状況を示しています。

- どのメッセージも、最終的にすべてのサブスクライバによってデキューされます。
- サブスクライバの間に優先順位はありません。これは、どのサブスクライバがどのメッセージをどのような順番でデキューするかはわからないということです。つまり、サブスクライバによってデキューされる順番は決められていません。
- この図からは、メッセージがすでにデキューされたのか、そしてキューから削除されたのかについてはわかりません。

図 11-5　複数のコンシューマを使用した通信

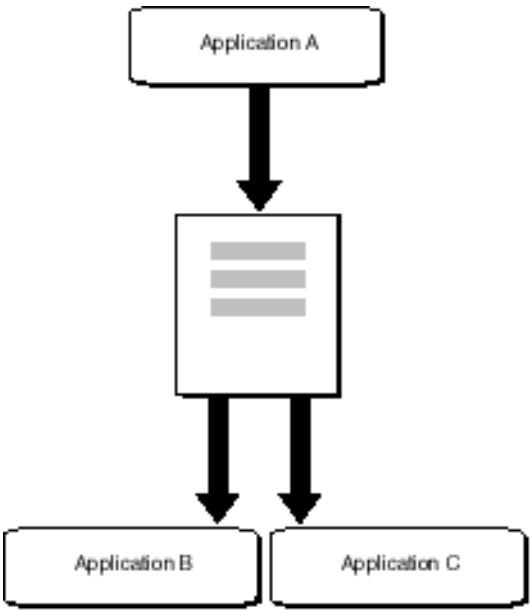
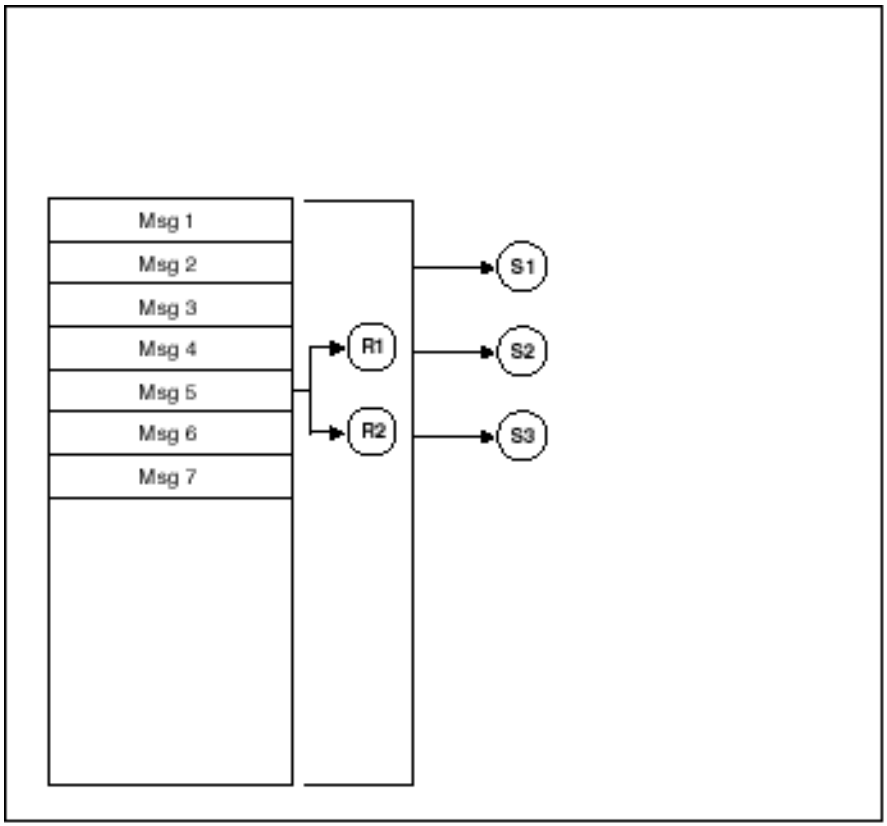


図 11-5 では、同じ処理を動的に示しています。この例では、複数のアプリケーションが、あるアプリケーションによって作成された結果を必要としているシナリオを示しています。「Application A」によってエンキューされるメッセージはすべて、「Application B」および「Application C」にデキューされます。これを可能にするには、複数のコンシューマ・キューが、「Application B」および「Application C」をキュー・サブスクライバとして特別に構成されます。結果的に、それらはキューに格納されるすべてのメッセージの暗黙的な受信者になります。

注意： キュー・サブスクライバは、アプリケーションまたは他のキューである場合があります。

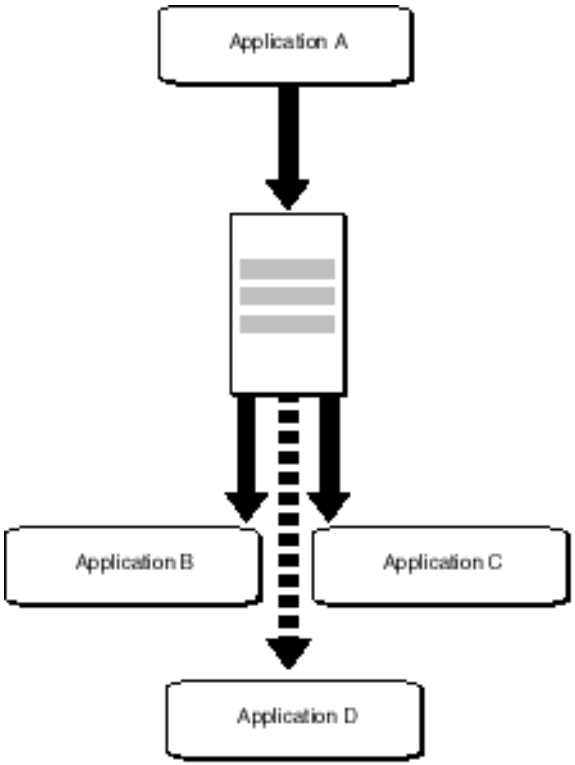
図 11-6　指定受信者による指定メッセージのデキュー



指定受信者による指定メッセージのデキューの例

図 11-6 は、どのようにメッセージを 1 つ以上の受信者に指定できるかを示しています。この場合、「Message 5」は、「Recipient-1」および「Recipient-2」にデキューされるように指定されています。この図が示すように、受信者はどれも、キューに対する 3 つのサブスクライバの 1 つではありません。

図 11-7　メッセージの明示的および暗黙的な受信者



最初に述べたように、サブスクライバは、特定のキューに格納されたすべてのメッセージをデキューできるという点から、「暗黙的な受信者」と呼ばれます。これは、雑誌の購読契約をしていて、暗黙的にすべての記事にアクセスすることと似ています。「受信者」と呼ばれるコンシューマのカテゴリは、特定のメッセージのターゲットに指定されるという点から「明示的な受信者」と呼ばれます。

図 11-7 に、Oracle AQ が両方の種類のコンシューマに対応するためにどのように動的に調整できるかを示します。このシナリオでは、「Application B」および「Application C」は、暗黙的受信者（サブスクライバ）です。しかし、メッセージは、キューのサブスクライバであるかどうかに関わらず特定のコンシューマ（サブスクライバ）に明示的に転送することもでき

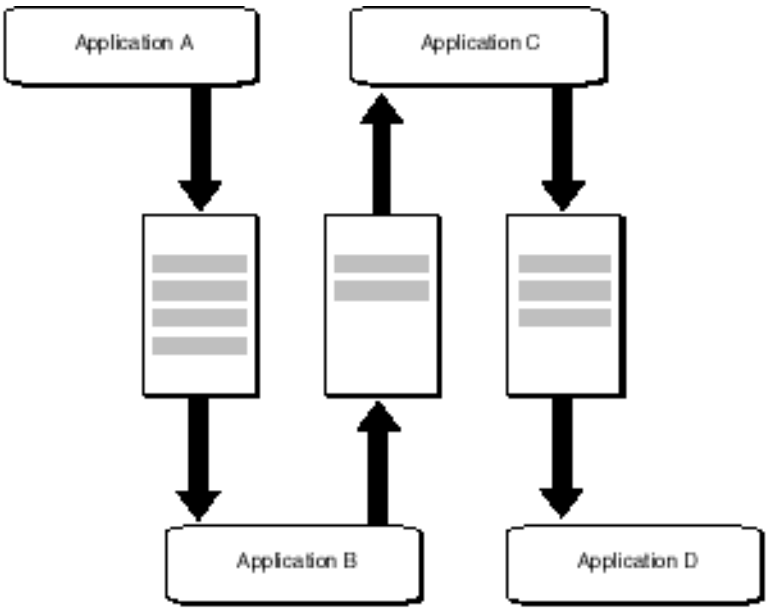
ます。このような受信者のリストは、そのメッセージのエンキュー・コールで指定され、そのキューのサブスクライバのリストを上書きします。図では、「Application D」は、「Application A」によってエンキューされてたメッセージの唯一の受信者に指定されています。

注意： 複数のプロデューサにより、異なるターゲットの受信者宛てに同時にメッセージがエンキューされることがあります。

AQ を使用したワークフローの実現の例

図 11-8 では、ワークフロー（連鎖したアプリケーションのトランザクションともいいます）の実現のために AQ が使用されています。ワークフローは、Applications A、B、C、D に実行される 4 つのステップで構成されています。キューは、業務処理の異なる処理段階の間の情報のフローをバッファするために使用されます。メッセージの遅延間隔および有効期限を指定することで、実行の枠を各アプリケーションに提供できます。

図 11-8　AQ を使用したワークフローの実現



ワークフローから考えると、メッセージの受渡しは、ペイロード・データの値に加えて業務資産となります。このため、AQ では、過去の傾向を分析して、将来の動向を予測するため

に、オプションでメッセージの保管がサポートされています。たとえば、この章のはじめに示した 3 つのアプリケーション・シナリオのうちの 2 つは、ワークフロー分析の実現を基盤にしています。

注意： メッセージ 1 および 2、3 の内容は同一であることも異なることもあります。異なる場合には、メッセージには、前のメッセージの内容が一部含まれます。

メッセージの伝播

メッセージのファンアウト

AQ では、メッセージの受信者はコンシューマまたは他のキューのどちらかになりえます。キューの場合には、実際の受信者は、そのキュー（順番で他のキューの場合もある）に対応しているサブスクライバによってに決定されます。これによって、多くの受信者に、単一のキューからメッセージをデキューすることを要求しなくても、メッセージをファンアウトできます。

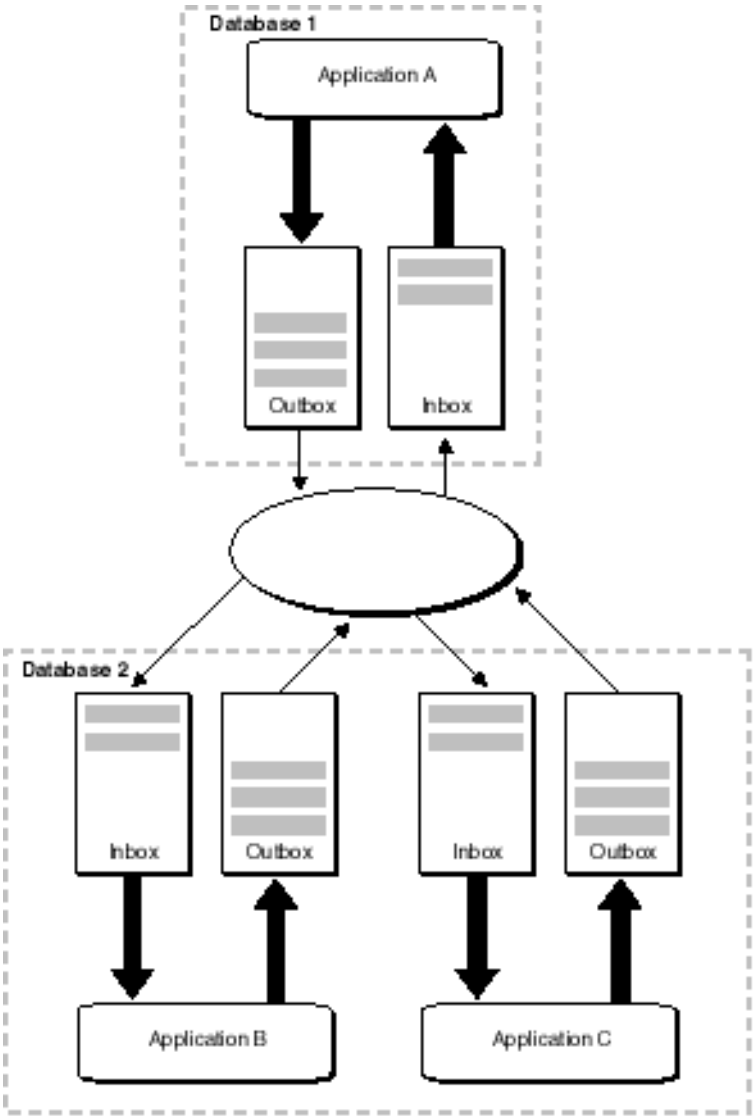
たとえば、「ソース」となるキューには、そのサブスクライバ・キューとして dispatch1@dest1 および dispatch2@dest2 があります。キュー dispatch1@dest1 にもまた、そのサブスクライバとしてキュー outerreach1@dest3 および outerreach2@dest4 があり、キュー dispatch2@dest2 には、サブスクライバとしてキュー outerreach3@dest21 および outerreach4@dest4 があります。このようにして、「ソース」にエンキューされたメッセージは、4 つの異なるキューのサブスクライバに伝播されます。

メッセージのファネルイン

メッセージ受信者としてのキューの別の使用方法としては、異なるキューのメッセージを単一のキューに結合する機能があります。このプロセスは、「複合」と呼ばれることもあります。

たとえば、キュー composite@endpoint がキュー funnel1@source1 および funnel2@source2 の両方に対応するサブスクライバである場合には、キュー composite@endpoint に対応するサブスクライバは、直接エンキューされるメッセージだけでなく、それらのキューにエンキューされるメッセージをすべて得ることができます。

図 11-9　メッセージの伝播



メッセージの伝播の例

図 11-9 は、AQ を介して通信している異なるデータベースにあるアプリケーションを示しています。各アプリケーションには、受信メッセージと送信メッセージを処理するためにインボックスとアウトボックスがあります。アプリケーションでは、メッセージが送信されるアプリケーションが、ローカル（同一ノード上）かまたはリモート（異なるノード上）かに関わらず、メッセージをアウトボックスにエンキューします。

同様に、アプリケーションでは、メッセージがローカルまたはリモートで作成されたかどうかは問題にはなりません。すべての場合に、アプリケーションでは、そのインボックスにあるメッセージがデキューされます。

Oracle AQ は、この交換を容易にし、メッセージを同じ基盤で扱います。

Oracle アドバンスト・キューイングの例

概要

「Oracle アドバンスト・キューイングの例」では、段階的なアプローチでユーザーをガイドします。

- ロールおよび権限の割当て
- 表およびキューの作成
  - オブジェクト型の場合
  - RAW 型の場合
  - 優先メッセージ用のオブジェクト型の場合
  - 複数コンシューマ用のオブジェクト型の場合
- オブジェクト型メッセージのエンキューおよびデキュー
- Pro*C/C++ を使ったオブジェクト型メッセージのエンキューおよびデキュー
- OCI を使ったオブジェクト型メッセージのエンキューおよびデキュー
- RAW 型メッセージのエンキューおよびデキュー
- Pro*C/C++ を使った RAW 型メッセージのエンキューおよびデキュー
- OCI を使った RAW 型メッセージのエンキューおよびデキュー
- 優先順位によるエンキューおよびデキュー
- 基準によるプリビュー後のメッセージのデキュー
- 遅延および時間切れによるメッセージのエンキューおよびデキュー
- Pro*C/C++ を使った、相関およびメッセージ ID によるメッセージのエンキューおよびデキュー

- OCI を使った、相関およびメッセージ ID によるメッセージのエンキューおよびデキュー
- 複数コンシューマ・キューでのメッセージのエンキューおよびデキュー
  - PL/SQL を使ってサブスクライバおよび特定のメッセージ受信者を指定する
  - OCI を使ってサブスクライバおよび特定のメッセージ受信者を指定する
- 複数コンシューマ・キューへのリモートのサブスクライバ / 受信者に対するメッセージのエンキューおよび伝播スケジューリング
- 伝播のスケジュール解除
- Message Grouping を使ったエンキューおよびデキュー
- AQ オブジェクトの削除
  - オブジェクト型のキューおよびキュー表を停止および削除する
  - RAW 型のキューおよびキュー表を停止および削除する
- ロールおよび権限の取消し

ロールおよび権限の割当て

```
/* ユーザーの作成および権限付与 : */
CONNECT sys/change_on_install as sysdba;
CREATE user aq identified by AQ;
GRANT AQ_ADMINISTRATOR_ROLE TO aq;
GRANT CONNECT TO aq;
GRANT RESOURCE TO aq;
EXECUTE dbms_aqadm.grant_type_access('aq');
CONNECT aq/AQ;

SET ECHO ON;
SET SERVEROUTPUT ON;
```

キュー表およびキューの作成

オブジェクト型のキュー表およびキューの作成

```
/*
```

```
queue_payload_type => 'aq.message_type');

EXECUTE dbms_aqadm.create_queue (
queue_name         => 'msg_queue',
queue_table        => 'aq.msg');

EXECUTE dbms_aqadm.start_queue (
queue_name         => 'msg_queue');
```

RAW 型のキュー表およびキューの作成

```
/* RAW型のキュー表およびキューの作成： */
EXECUTE dbms_aqadm.create_queue_table (
queue_table          => 'aq.raw_msg',
queue_payload_type   => 'RAW');

EXECUTE dbms_aqadm.create_queue (
queue_name          => 'raw_msg_queue',
queue_table         => 'aq.raw_msg');

EXECUTE dbms_aqadm.start_queue (
queue_name          => 'raw_msg_queue');
```

優先メッセージのキュー表およびキューの作成

```
EXECUTE dbms_aqadm.create_queue_table (
queue_table        => 'aq.priority_msg',
sort_list          => 'PRIORITY,ENQ_TIME',
queue_payload_type => 'aq.message_type');

EXECUTE dbms_aqadm.create_queue (
queue_name         => 'priority_msg_queue',
queue_table        => 'aq.priority_msg');

EXECUTE dbms_aqadm.start_queue (
queue_name         => 'priority_msg_queue');
```

複数のコンシューマ用のキュー表およびキューの作成

```
EXECUTE dbms_aqadm.create_queue_table (
queue_table        => 'aq.msg_multiple',
multiple_consumers => TRUE,
queue_payload_type => 'aq.message_type');
```

```
EXECUTE dbms_aqadm.create_queue (
queue_name         => 'msg_queue_multiple',
queue_table        => 'aq.msg_multiple');

EXECUTE dbms_aqadm.start_queue (
queue_name         => 'msg_queue_multiple');
```

伝播を検証するキューの作成

```
EXECUTE dbms_aqadm.create_queue (
queue_name        => 'another_msg_queue',
queue_table       => 'aq.msg_multiple');

EXECUTE dbms_aqadm.start_queue (
queue_name         => 'another_msg_queue');
```

オブジェクト型メッセージのエンキューおよびデキュー

他のパラメータを使わずに 1 つのメッセージをエンキューするには、キュー名およびペイロードを指定します。

```
/* msg_queue へのエンキュー : */
DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN
message := message_type('NORMAL MESSAGE',
'enqueued to msg_queue first.');

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

COMMIT;
END;
/

/* msg_queue からのデキュー : */
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
message_properties  dbms_aq.message_properties_t;
```

```
message_handle      RAW(16);
message             aq.message_type;

BEGIN
dbms_aq.dequeue(queue_name => 'msg_queue',
              dequeue_options    => dequeue_options,
              message_properties => message_properties,
              payload            => message,
              msgid              => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
                                     ' ... ' || message.text );
COMMIT;
END;
/
```

Pro*C/C++ を使ったオブジェクト型メッセージのエンキューおよびデキュー

```
#include  <stdio.h>
#include  <string.h>
#include  <sqlca.h>
#include  <sql2oci.h>
/* オブジェクト型
'aq.message_type' を処理して生成されるヘッダー・ファイル : */
#include  "pceg.h"

void sql_error(msg)
char *msg;
{
EXEC SQL WHENEVER SQLERROR CONTINUE;
printf("%s¥n", msg);
printf("¥n% .800s ¥n", sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK WORK RELEASE;
exit(1);
}

main()
{
message_type     *message = (message_type*)0;  /* ペイロード */
char             user[60]="aq/AQ";  /* ユーザー・ログオン・パスワード */
char             subject[30];  /* ペイロード型の */
char             txt[80];  /* コンポーネント */

/* 「オブジェクト・キュー」に対する「エンキュー」および「デキュー」*/

/*  データベースに接続 : */
```

```
EXEC SQL CONNECT :user;

/* Oracle エラーの場合エラー番号を出力する :*/
EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle Error :");

/* ホスト変数にオブジェクト・キャッシュからメモリーを割り当てる : */
EXEC SQL ALLOCATE :message;

/* エンキュー */

strcpy(subject, "NORMAL ENQUEUE");
strcpy(txt, "The Enqueue was done through PLSQL embedded in PROC");

/* メッセージ のコンポーネントの初期化 : */
EXEC SQL OBJECT SET SUBJECT, TEXT OF :message TO :subject, :txt;

/* 埋込み PLSQL の AQ エンキュー・プロシージャのコール : */
EXEC SQL EXECUTE
DECLARE
message_properties   dbms_aq.message_properties_t;
enqueue_options      dbms_aq.enqueue_options_t;
msgid                RAW(16);
BEGIN
 /* ホスト変数 'message' をペイロードに結合する : */
dbms_aq.enqueue(queue_name => 'msg_queue',
message_properties => message_properties,
enqueue_options => enqueue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* 作業のコミット */
EXEC SQL COMMIT;

printf("Enqueued Message ¥n");
printf("Subject  :%s¥n",subject);
printf("Text     :%s¥n",txt);

/* デキュー */

/* 埋込み PLSQL の AQ デキュー・プロシージャのコール : */
EXEC SQL EXECUTE
DECLARE
message_properties  dbms_aq.message_properties_t;
dequeue_options     dbms_aq.dequeue_options_t;
msgid RAW(16);
BEGIN
```

```
/* ペイロードをホスト変数 'message' に戻す :  */
dbms_aq.dequeue(queue_name => 'msg_queue',
message_properties => message_properties,
dequeue_options => dequeue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
 /* 作業のコミット :*/
EXEC SQL COMMIT;

/* メッセージのコンポーネントの抽出 : */
EXEC SQL OBJECT GET SUBJECT,TEXT FROM :message INTO :subject,:txt;

printf("Dequeued Message ¥n");
printf("Subject  :%s¥n",subject);
printf("Text     :%s¥n",txt);
}
```

OCI を使ったオブジェクト型メッセージのエンキューおよびデキュー

```
#ifndef SL_ORACLE
#include <sl.h>
#endif

#ifndef OCI_ORACLE
#include <oci.h>
#endif

struct message
{
  OCIString   *subject;
  OCIString   *data;
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_subject;
  OCIInd    null_data;
};
typedef struct null_message null_message;

int main()
{
```

```
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  dvoid     *tmp;
  OCIType *mesg_tdo = (OCIType *) 0;
  message  msg;
  null_message nmsg;
  message *mesg = &msg;
  null_message *nmesg = &nmsg;
  message *deqmesg = (message *)0;
  null_message *ndeqmesg = (null_message *)0;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
(dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
   52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
   52, (dvoid **) &tmp);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
   52, (dvoid **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
   52, (dvoid **) &tmp);

  OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
   (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

  /* message_type の TDO の取得 */
  OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
       (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
       (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

  /* メッセージ・ペイロードの準備 */
  mesg->subject = (OCIString *)0;
  mesg->data = (OCIString *)0;
  OCIStringAssignText(envhp, errhp,
    (CONST text *)"NORMAL MESSAGE", strlen("NORMAL MESSAGE"),
    &mesg->subject);
```

```
  OCIStringAssignText(envhp, errhp,
    (CONST text *)"OCI ENQUEUE", strlen("OCI ENQUEUE"),
    &mesg->data);
  nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

  /* msg_queue へのエンキュー */
  OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
       mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);
  OCITransCommit(svchp, errhp, (ub4) 0);

  /* msg_queue からのデキュー */
  OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
       mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
  printf("Subject: %s¥n", OCIStringPtr(envhp, deqmesg->subject));
  printf("Text: %s¥n", OCIStringPtr(envhp, deqmesg->data));
  OCITransCommit(svchp, errhp, (ub4) 0);

}
```

RAW 型メッセージのエンキューおよびデキュー

```
/* RAW を含むメッセージのエンキュー : */
DECLARE
      enqueue_options     dbms_aq.enqueue_options_t;
      message_properties  dbms_aq.message_properties_t;
      message_handle      RAW(16);
      message             RAW(4096);

BEGIN
      message := hextoraw(rpad('FF',4095,'FF'));
      dbms_aq.enqueue(queue_name => 'raw_msg_queue',
              enqueue_options    => enqueue_options,
              message_properties => message_properties,
              payload  => message,
              msgid              => message_handle);

      COMMIT;
END;
/* raw_msg_queue からのデキュー : */
DECLARE
      dequeue_options     dbms_aq.dequeue_options_t;
      message_properties  dbms_aq.message_properties_t;
      message_handle      RAW(16);
      message             RAW(4096);

BEGIN
```

```
 dbms_aq.dequeue(queue_name => 'raw_msg_queue',
dequeue_options    => dequeue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

COMMIT;
END;
```

Pro*C/C++ を使った RAW 型メッセージのエンキューおよびデキュー

```
#include  <stdio.h>
#include  <string.h>
#include  <sqlca.h>
#include  <sql2oci.h>

void sql_error(msg)
char *msg;
{
EXEC SQL WHENEVER SQLERROR CONTINUE;
printf("%s¥n", msg);
printf("¥n% .800s ¥n", sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK WORK RELEASE;
exit(1);
}

main()
{
OCIEnv         *oeh;   /* OCI Env ハンドル */
OCIError       *err;   /* OCI エラー・ハンドル */
OCIRaw         *message= (OCIRaw*)0;   /* ペイロード */
ub1            message_txt[100];   /* ペイロードのデータ */
char           user[60]="aq/AQ";   /* ユーザー・ログオン・パスワード */
int            status;   /* OCI コールのステータスを戻す */

/* RAW キューに対するエンキューおよびデキュー */

/* データベースに接続： */
EXEC SQL CONNECT :user;

/* Oracle エラーの場合エラー番号を出力する： */
EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle Error :");

/* OCI Env ハンドルの取得： */
if (SQLEnvGet(SQL_SINGLE_RCTX, &oeh) != OCI_SUCCESS)
{
```

```
printf(" error in SQLEnvGet ¥n");
exit(1);
}
/* OCI エラー・ハンドルの取得 : */
if (status = OCIHandleAlloc((dvoid *)oeh, (dvoid **)&err,
(ub4)OCI_HTYPE_ERROR, (ub4)0, (dvoid **)0))
{
printf(" error in OCIHandleAlloc %d ¥n", status);
exit(1);
}

/* エンキュー */
/* ロー・ペイロードに格納されるバイト :*/
strcpy(message_txt, "Enqueue to a Raw payload queue ");

/* バイトを OCIRaw ポインタに割り当てる :
明示的に OCIRaw* に割り当てる必要があるメモリー : */
if (status=OCIRawAssignBytes(oeh, err, message_txt, 100,
 &message))
{
printf(" error in  OCIRawAssignBytes  %d ¥n", status);
exit(1);
}

/*  埋込み PLSQL の AQ エンキュー・プロシージャのコール :*/
EXEC SQL EXECUTE
DECLARE
message_properties   dbms_aq.message_properties_t;
enqueue_options      dbms_aq.enqueue_options_t;
msgid                RAW(16);
BEGIN
/* ホスト変数メッセージをロー・ペイロードに結合する : */
dbms_aq.enqueue(queue_name => 'raw_msg_queue',
message_properties => message_properties,
enqueue_options => enqueue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* 作業のコミット : */
EXEC SQL COMMIT;

/* デキュー */
/*  埋込み PLSQL の AQ デキュー・プロシージャのコール :*/
EXEC SQL EXECUTE
DECLARE
message_properties  dbms_aq.message_properties_t;
```

```
dequeue_options     dbms_aq.dequeue_options_t;
msgid RAW(16);
BEGIN
/* ロー・ペイロードをホスト変数 'message' に戻す :*/
dbms_aq.dequeue(queue_name => 'raw_msg_queue',
message_properties => message_properties,
dequeue_options => dequeue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* 作業のコミット : */
EXEC SQL COMMIT;
}
```

OCI を使った RAW 型メッセージのエンキューおよびデキュー

```
#ifndef SL_ORACLE
#include <sl.h>
#endif

#ifndef OCI_ORACLE
#include <oci.h>
#endif

int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  dvoid     *tmp;
  OCIType *mesg_tdo = (OCIType *) 0;
  char  msg_text[100];
  OCIRaw  *mesg = (OCIRaw *)0;
  OCIRaw*deqmesg = (OCIRaw *)0;
  OCIInd   ind = 0;
  dvoid  *indptr = (dvoid *)&ind;
  inti;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
(dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
   52, (dvoid **) &tmp);
```

```
  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
   52, (dvoid **) &tmp);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
   52, (dvoid **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
   52, (dvoid **) &tmp);

  OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
   (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

  /* RAW データ型の TDO の取得 */
  OCITypeByName(envhp, errhp, svchp, (CONST text *)"SYS", strlen("SYS"),
       (CONST text *)"RAW", strlen("RAW"),
       (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

  /* メッセージ・ペイロードの準備 */
  strcpy(msg_text, "Enqueue to a RAW queue");
  OCIRawAssignBytes(envhp, errhp, msg_text, strlen(msg_text), &mesg);

 /* メッセージを raw_msg_queue にエンキューする */
  OCIAQEnq(svchp, errhp, (CONST text *)"raw_msg_queue", 0, 0,
       mesg_tdo, (dvoid **)&mesg, (dvoid **)&indptr, 0, 0);
  OCITransCommit(svchp, errhp, (ub4) 0);

  /* 同じメッセージを C 変数 deqmesg にデキューする */
  OCIAQDeq(svchp, errhp, (CONST text *)"raw_msg_queue", 0, 0,
   mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&indptr, 0, 0);
  for (i = 0; i < OCIRawSize(envhp, deqmesg); i++)
    printf("%c", *(OCIRawPtr(envhp, deqmesg) + i));
  OCITransCommit(svchp, errhp, (ub4) 0);
}
```

優先順位によるメッセージのエンキューおよびデキュー

2 つのメッセージが同じ優先順位でエンキューされると、先にエンキューされた方が、先にデキューされます。ただし、2 つのメッセージが異なる優先順位を持っている場合には、値の小さい（優先順位の高い）メッセージが先にデキューされます。

```
/* 優先順位が 30 および 5 の 2 つのメッセージをエンキューする : */
DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
      message             aq.message_type;

BEGIN
message := message_type('PRIORITY MESSAGE',
enqued at priority 30.');

message_properties.priority := 30;

dbms_aq.enqueue(queue_name => 'priority_msg_queue',
enqueue_options    => enqueue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

message := message_type('PRIORITY MESSAGE',
'Enqueued at priority 5.');

message_properties.priority := 5;

      dbms_aq.enqueue(queue_name => 'priority_msg_queue',
              enqueue_options    => enqueue_options,
              message_properties => message_properties,
              payload            => message,
              msgid              => message_handle);
END;
/

/* 優先順位によるキューからのデキュー : */
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message aq.message_type;

BEGIN
dbms_aq.dequeue(queue_name  => 'priority_msg_queue',
dequeue_options       => dequeue_options,
message_properties    => message_properties,
payload               => message,
msgid                 => message_handle);
```

```
dbms_output.put_line ('Message: ' || message.subject ||
' ... ' || message.text );

COMMIT;

dbms_aq.dequeue(queue_name => 'priority_msg_queue',
dequeue_options      => dequeue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
' ... ' || message.text );
COMMIT;
END;
/
On return, the second message with priority set to 5 will be retrieved before the
message with priority set to 30 since priority takes precedence over enqueue time.
```

基準によるプリビュー後のメッセージのデキュー

アプリケーションでは、ブラウズ・モードまたはロックモードで、メッセージを削除せずにプリビューできます。必要なメッセージは、プリビューした後でキューから削除できます。

```
/* 次の 6 つのメッセージを msg_queue にエンキューする
— GREEN、GREEN、YELLOW、VIOLET、BLUE、RED */

DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN
message := message_type('GREEN',
'GREEN enqueued to msg_queue first.');

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

message := message_type('GREEN',
'GREEN also enqueued to msg_queue second.');
```

```
dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

message := message_type('YELLOW',
'YELLOW enqueued to msg_queue third.');

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

dbms_output.put_line ('Message handle: ' || message_handle);

message := message_type('VIOLET',
'VIOLET enqueued to msg_queue fourth.');

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

message := message_type('BLUE',
'BLUE enqueued to msg_queue fifth.');

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

message := message_type('RED',
'RED enqueued to msg_queue sixth.');

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

      COMMIT;
END;
```

```
/

/* RED が検索されるまで「ブラウズ」モードでデキューして
キューから RED を削除する : */
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN
dequeue_options.dequeue_mode := dbms_aq.BROWSE;

      LOOP
dbms_aq.dequeue(queue_name          => 'msg_queue',
                 dequeue_options    => dequeue_options,
                 message_properties => message_properties,
                 payload            => message,
                 msgid              => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
                                        ' ... ' || message.text );

EXIT WHEN message.subject = 'RED';

END LOOP;

dequeue_options.dequeue_mode := dbms_aq.REMOVE;
dequeue_options.msgid        := message_handle;

dbms_aq.dequeue(queue_name => 'msg_queue',
dequeue_options    => dequeue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
' ... ' || message.text );

      COMMIT;
END;
/

/* BLUE が検索されるまで、「ロック」モードでデキューして
キューから BLUE を削除する : */
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
```

```
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN
dequeue_options.dequeue_mode := dbms_aq.LOCKED;

LOOP

dbms_aq.dequeue(queue_name => 'msg_queue',
                dequeue_options    => dequeue_options,
                message_properties => message_properties,
                payload            => message,
                msgid              => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
          ' ... ' || message.text );

EXIT WHEN message.subject = 'BLUE';
      END LOOP;

dequeue_options.dequeue_mode := dbms_aq.REMOVE;
dequeue_options.msgid        := message_handle;

dbms_aq.dequeue(queue_name => 'msg_queue',
dequeue_options      => dequeue_options,
message_properties   => message_properties,
payload              => message,
msgid => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
' ... ' || message.text );

      COMMIT;
END;
/
```

遅延および時間切れによるメッセージのエンキューおよびデキュー

エンキューでは、デキュー・コールによってメッセージが取出し可能な時間を指定できます。そのために、プロデューサ（メッセージをエンキューしているエージェント）では、メッセージをエンキューする際にパラメータ「delay」を使って、メッセージの取出し禁止時間が満了する時刻を指定します。またプロデューサは、メッセージが時間切れになる時刻も指定できます。時間切れになると、メッセージは例外キューに移動されます。

注意： 時間切れは最も早いデキュー時間から計算されています。たとえば、アプリケーション側でメッセージを、今から 1 週間後以降、3 週間後までの間にデキューしたい場合には、時間切れ時刻を 2 週間に設定する必要があります。このシナリオを、次のコード・セグメントで説明します。

```
/* 遅延の使用のためのメッセージのエンキュー： */
DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN
message := message_type('DELAYED',
'This message is delayed one week.');
message_properties.delay := 7*24*60*60;
message_properties.expiration := 2*7*24*60*60;

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload            => message,
msgid              => message_handle);

      COMMIT;
END;
```

Pro*C/C++ を使った、相関およびメッセージ ID によるメッセージのエンキューおよびデキュー

```
#include  <stdio.h>
#include  <string.h>
#include  <sqlca.h>
#include  <sql2oci.h>
/* オブジェクト型
'aq.message_type' を処理して生成されるヘッダー・ファイル： */
#include  "pceg.h"

void sql_error(msg)
char *msg;
{
EXEC SQL WHENEVER SQLERROR CONTINUE;
printf("%s¥n", msg);
```

```
printf("¥n% .800s ¥n", sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK WORK RELEASE;
exit(1);
}

main()
{
OCIEnv            *oeh;  /* OCI Env ハンドル */
OCIError          *err;  /* OCI エラー・ハンドル */
message_type      *message = (message_type*)0; /* キューのペイロード */
OCIRaw            *msgid = (OCIRaw*)0;  /* メッセージ ID */
ub1               msgmem[16]="";  /* msgid のメモリー */
char              user[60]="aq/AQ";  /* ユーザー・ログイン・パスワード */
char              subject[30];  /* message_type の */
char              txt[80];  /* コンポーネント */
char              correlation1[30];  /* メッセージの相関  */
char correlation2[30];
int               status;  /* OCI コールに戻されるコード */

/* 相関および msgid によるデキュー */

/* データベースに接続 : */
EXEC SQL CONNECT :user;
EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle Error :");

/* オブジェクト・キャッシュ内にホスト変数の領域を割り当てる : */
EXEC SQL ALLOCATE :message;

/* OCI Env ハンドルの取得 : */
if (SQLEnvGet(SQL_SINGLE_RCTX, &oeh) != OCI_SUCCESS)
{
 printf(" error in SQLEnvGet ¥n");
 exit(1);
}
/* OCI エラー・ハンドルの取得 : */
if (status = OCIHandleAlloc((dvoid *)oeh, (dvoid **)&err,
(ub4)OCI_HTYPE_ERROR, (ub4)0, (dvoid **)0))
{
printf(" error in OCIHandleAlloc %d ¥n", status);
exit(1);
}


/* メモリーを msgid に割り当てる :
明示的に OCIRaw* に割り当てる必要があるメモリー : */
if (status=OCIRawAssignBytes(oeh, err, msgmem, 16, &msgid))
{
```

```
printf(" error in  OCIRawAssignBytes  %d ¥n", status);
exit(1);
}

/* 第 1 のエンキュー */

strcpy(correlation1, "1st message");
strcpy(subject, "NORMAL ENQUEUE1");
strcpy(txt, "The Enqueue was done through PLSQL embedded in PROC");

/* メッセージのコンポーネントの初期化 : */
EXEC SQL OJECT SET SUBJECT, TEXT OF :message TO :subject, :txt;

/* 埋込み PLSQL の AQ エンキュー・プロシージャのコール : */
EXEC SQL EXECUTE
DECLARE
message_properties   dbms_aq.message_properties_t;
enqueue_options      dbms_aq.enqueue_options_t;
BEGIN
/* ホスト変数 'correlation1': をメッセージ相関に結合 */
message_properties.correlation := :correlation1;

/* ホスト変数 'message' をペイロードに結合し、
メッセージ ID をホスト変数 'msgid' に戻す : */
dbms_aq.enqueue(queue_name => 'msg_queue',
message_properties => message_properties,
enqueue_options => enqueue_options,
payload => :message,
msgid => :msgid);
END;
END-EXEC;
/* 作業のコミット : */
EXEC SQL COMMIT;

printf("Enqueued Message ¥n");
printf("Subject  :%s¥n",subject);
printf("Text     :%s¥n",txt);

/* 第 2 のエンキュー */

strcpy(correlation2, "2nd message");
strcpy(subject, "NORMAL ENQUEUE2");
strcpy(txt, "The Enqueue was done through PLSQL embedded in PROC");

/* メッセージのコンポーネントの初期化 : */
EXEC SQL OBJECT SET SUBJECT, TEXT OF :messsage TO :subject,:txt;
```

```
/* 埋込み PLSQL の AQ エンキュー・プロシージャのコール : */
EXEC SQL EXECUTE
DECLARE
message_properties   dbms_aq.message_properties_t;
enqueue_options      dbms_aq.enqueue_options_t;
msgid                RAW(16);
BEGIN
/* ホスト変数 'correlation2': をメッセージ相関に結合 */
message_properties.correlation := :correlation2;

/* ホスト変数 'message': をペイロードに結合 */
dbms_aq.enqueue(queue_name => 'msg_queue',
message_properties => message_properties,
enqueue_options => enqueue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* 作業のコミット : */
EXEC SQL COMMIT;
printf("Enqueued Message ¥n");
printf("Subject  :%s¥n",subject);
printf("Text     :%s¥n",txt);

/* 相関による第 1 のデキュー */

EXEC SQL EXECUTE
DECLARE
message_properties  dbms_aq.message_properties_t;
dequeue_options     dbms_aq.dequeue_options_t;
msgid               RAW(16);
BEGIN
/* ホスト変数 'correlation2' における相関によるデキュー : */
dequeue_options.correlation := :correlation2;

/* ホスト変数 'message': にペイロードを戻す */
dbms_aq.dequeue(queue_name => 'msg_queue',
message_properties => message_properties,
dequeue_options => dequeue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* 作業をコミットする : */
EXEC SQL COMMIT;

/* メッセージのコンポーネントの値を抽出する : */
```

```
EXEC SQL OBJECT GET SUBJECT, TEXT FROM :message INTO :subject,:txt;

printf("Dequeued Message ¥n");
printf("Subject  :%s¥n",subject);
printf("Text     :%s¥n",txt);

/*MSGIDによる第2のデキュー */

EXEC SQL EXECUTE
DECLARE
message_properties  dbms_aq.message_properties_t;
dequeue_options     dbms_aq.dequeue_options_t;
msgid               RAW(16);
BEGIN
/* ホスト変数'msgid'におけるmsgidによるデキュー: */
dequeue_options.msgid := :msgid;

/* ホスト変数'message'にペイロードを戻す: */
dbms_aq.dequeue(queue_name => 'msg_queue',
message_properties => message_properties,
dequeue_options => dequeue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* 作業のコミット: */
EXEC SQL COMMIT;
}
```

OCI を使った、相関およびメッセージ ID によるメッセージのエンキューおよびデキュー

```
#ifndef SL_ORACLE
#include <sl.h>
#endif

#ifndef OCI_ORACLE
#include <oci.h>
#endif

struct message
{
  OCIString   *subject;
  OCIString   *data;
};
```

```
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_subject;
  OCIInd    null_data;
};
typedef struct null_message null_message;

int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  dvoid     *tmp;
  OCIType *mesg_tdo = (OCIType *) 0;
  message  msg;
  null_message nmsg;
  message *mesg = &msg;
  null_message *nmesg = &nmsg;
  message *deqmesg = (message *)0;
  null_message *ndeqmesg = (null_message *)0;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
(dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
   52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
   52, (dvoid **) &tmp);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
   52, (dvoid **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
   52, (dvoid **) &tmp);

  OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
   (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);
```

```
  /* message_type の TDO の取得 */
  OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
       (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
       (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

  /* メッセージ・ペイロードの準備 */
  mesg->subject = (OCIString *)0;
  mesg->data = (OCIString *)0;
  OCIStringAssignText(envhp, errhp,
    (CONST text *)"NORMAL MESSAGE", strlen("NORMAL MESSAGE"),
    &mesg->subject);
  OCIStringAssignText(envhp, errhp,
    (CONST text *)"OCI ENQUEUE", strlen("OCI ENQUEUE"),
    &mesg->data);
  nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

  /* msg_queue へのエンキュー */
  OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
       mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);
  OCITransCommit(svchp, errhp, (ub4) 0);

  /* msg_queue からのデキュー */
  OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
       mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
  printf("Subject: %s¥n", OCIStringPtr(envhp, deqmesg->subject));
  printf("Text: %s¥n", OCIStringPtr(envhp, deqmesg->data));
  OCITransCommit(svchp, errhp, (ub4) 0);

}
```

PL/SQL を使った複数コンシューマ・キューでのメッセージのエンキューおよびデキュー

```
/* サブスクライバ・リストの作成： */
DECLARE
subscriber sys.aq$_agent;

/* サブスクライバ RED および GREEN をサブスクライバ・リストに追加する： */
BEGIN
subscriber := sys.aq$_agent('RED', NULL, NULL);
dbms_aqadm.add_subscriber(queue_name => 'msg_queue_multiple',
subscriber => subscriber);

subscriber := sys.aq$_agent('GREEN', NULL, NULL);
```

```
dbms_aqadm.add_subscriber(queue_name => 'msg_queue_multiple',
subscriber => subscriber);
END;
/
DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
recipients          dbms_aq.aq$_recipient_list_t;
message_handle      RAW(16);
message             aq.message_type;

/* キューに対するサブスクライバ（RED および GREEN）に
MESSAGE 1 をエンキューする： */
BEGIN
message := message_type('MESSAGE 1',
'This message is queued for queue subscribers.');

dbms_aq.enqueue(queue_name => 'msg_queue_multiple',
enqueue_options    => enqueue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

/* MESSAGE 2 を指定の受信者、つまり RED および BLUE にエンキューする */
message := message_type('MESSAGE 2',
'This message is queued for two recipients.');
recipients(1) := sys.aq$_agent('RED', NULL, NULL);
recipients(2) := sys.aq$_agent('BLUE', NULL, NULL);
message_properties.recipient_list := recipients;

dbms_aq.enqueue(queue_name => 'msg_queue_multiple',
enqueue_options    => enqueue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

COMMIT;
END;
/
```

RED はキューのサブスクライバであると同時に、MESSAGE 2 の指定された受信者であることに注意してください。それと対照的に GREEN は、受信者が指定されていないキュー内のメッセージ（この場合 MESSAGE）に対するサブスクライバでしかありません。BLUE は、キューのサブスクライバではありませんが、MESSAGE 2 の宛先に指定されています。

```
/* msg_queue_multiple からメッセージをデキューする： */
```

```
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;
no_messages   exception;
pragma exception_init (no_messages, -25228);

BEGIN

dequeue_options.wait := dbms_aq.NO_WAIT;

/* コンシューマ BLUE は、MESSAGE 2 を取得する : */
dequeue_options.consumer_name := 'BLUE';

LOOP

dbms_aq.dequeue(queue_name   => 'msg_queue_multiple',
          dequeue_options    => dequeue_options,
          message_properties => message_properties,
          payload => message,
          msgid              => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
         ' ... ' || message.text );

END LOOP;
EXCEPTION
WHEN no_messages THEN
dbms_output.put_line ('No more messages for BLUE');
COMMIT;
END;

BEGIN
/* コンシューマ RED は MESSAGE 1 および MESSAGE 2 を取得する : */
dequeue_options.consumer_name := 'RED';

LOOP
dbms_aq.dequeue(queue_name   => 'msg_queue_multiple',
          dequeue_options    => dequeue_options,
          message_properties => message_properties,
          payload            => message,
          msgid              => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
                                        ' ... ' || message.text );
END LOOP;
```

```
EXCEPTION
WHEN no_messages THEN
dbms_output.put_line ('No more messages for RED');
COMMIT;
END;

BEGIN
/* コンシューマ GREEN は MESSAGE 1 を取得する : */
dequeue_options.consumer_name := 'GREEN';

LOOP
dbms_aq.dequeue(queue_name   => 'msg_queue_multiple',
          dequeue_options    => dequeue_options,
          message_properties => message_properties,
          payload            => message,
          msgid              => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
          ' ... ' || message.text );
END LOOP;
EXCEPTION
WHEN no_messages THEN
dbms_output.put_line ('No more messages for GREEN');
COMMIT;
END;
/
```

OCI を使った複数コンシューマ・キューでのメッセージのエンキューおよびデキュー

```
#ifndef SL_ORACLE
#include <sl.h>
#endif

#ifndef OCI_ORACLE
#include <oci.h>
#endif

struct message
{
  OCIString   *subject;
  OCIString   *data;
};
typedef struct message message;

struct null_message
```

```
{
  OCIInd    null_adt;
  OCIInd    null_subject;
  OCIInd    null_data;
};
typedef struct null_message null_message;

int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  dvoid     *tmp;
  OCIType *mesg_tdo = (OCIType *) 0;
  message  msg;
  null_message nmsg;
  message *mesg = &msg;
  null_message *nmesg = &nmsg;
  message *deqmesg = (message *)0;
  null_message *ndeqmesg = (null_message *)0;
  OCIAQMsgProperties *msgprop = (OCIAQMsgProperties *)0;
  OCIAQAgent *agents[2];
  OCIAQDeqOptions *deqopt = (OCIAQDeqOptions *)0;
  ub4wait = OCI_DEQ_NO_WAIT;
  ub4 navigation = OCI_DEQ_FIRST_MSG;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
(dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
   52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
   52, (dvoid **) &tmp);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
   52, (dvoid **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
   52, (dvoid **) &tmp);

  OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
   (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);
```

```
   OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

   /*message_type の TDO の取得 */
   OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
        (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
        (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

   /* メッセージ・ペイロードの準備 */
   mesg->subject = (OCIString *)0;
   mesg->data = (OCIString *)0;
   OCIStringAssignText(envhp, errhp,
     (CONST text *)"MESSAGE 1", strlen("MESSAGE 1"),
     &mesg->subject);
   OCIStringAssignText(envhp, errhp,
     (CONST text *)"mesg for queue subscribers",
     strlen("mesg for queue subscribers"), &mesg->data);
   nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

   /* キューに対するサブスクライバ、つまり RED および GREEN に MESSAGE 1 をエンキューする */
   OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue_multiple", 0, 0,
        mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);

   /* 特定の受信者、つまり RED および BLUE に MESSAGE 2 をエンキューする */
   /* メッセージ・ペイロードの準備 */
   OCIStringAssignText(envhp, errhp,
     (CONST text *)"MESSAGE 2", strlen("MESSAGE 2"),
     &mesg->subject);
   OCIStringAssignText(envhp, errhp,
     (CONST text *)"mesg for two recipients",
     strlen("mesg for two recipients"), &mesg->data);

   /* AQ メッセージ・プロパティおよびエージェント記述子の割当て */
   OCIDescriptorAlloc(envhp, (dvoid **)&msgprop,
OCI_DTYPE_AQMSG_PROPERTIES, 0, (dvoid **)0);
   OCIDescriptorAlloc(envhp, (dvoid **)&agents[0],
  OCI_DTYPE_AQAGENT, 0, (dvoid **)0);
   OCIDescriptorAlloc(envhp, (dvoid **)&agents[1],
  OCI_DTYPE_AQAGENT, 0, (dvoid **)0);

   /* 受信者リスト、RED および BLUE の準備 */
   OCIAttrSet(agents[0], OCI_DTYPE_AQAGENT, "RED", strlen("RED"),
      OCI_ATTR_AGENT_NAME, errhp);
   OCIAttrSet(agents[1], OCI_DTYPE_AQAGENT, "BLUE", strlen("BLUE"),
      OCI_ATTR_AGENT_NAME, errhp);
   OCIAttrSet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES, (dvoid *)agents, 2,
       OCI_ATTR_RECIPIENT_LIST, errhp);
```

```
 OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue_multiple", 0, msgprop,
      mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);

 OCITransCommit(svchp, errhp, (ub4) 0);

 /* 異なるコンシューマ名を使用してメッセージをデキューする */
 /* デキュー・オプション記述子を割り当て、デキュー・オプションを設定する */
 OCIDescriptorAlloc(envhp, (dvoid **)&deqopt, OCI_DTYPE_AQDEQ_OPTIONS, 0,
    (dvoid **)0);

 /* 待機パラメータを NO_WAIT に設定すると、デキューはすぐに戻される */
 OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&wait, 0,
    OCI_ATTR_WAIT, errhp);

 /* デキュー・オプションに新しい consumer_name が設定された後で */
 /* ナビゲーションを FIRST_MESSAGE に設定すると、デキューは位置をリセットする。*/
 OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&navigation, 0,
    OCI_ATTR_NAVIGATION, errhp);

 /* コンシューマ BLUE として msg_queue_multiple からデキュー */
 OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)"BLUE", strlen("BLUE"),
    OCI_ATTR_CONSUMER_NAME, errhp);
 while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
== OCI_SUCCESS)
 {
   printf("Subject: %s¥n", OCIStringPtr(envhp, deqmesg->subject));
   printf("Text: %s¥n", OCIStringPtr(envhp, deqmesg->data));
 }
 OCITransCommit(svchp, errhp, (ub4) 0);

 /* コンシューマ RED として msg_queue_multiple からデキュー */
 OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)"RED", strlen("RED"),
    OCI_ATTR_CONSUMER_NAME, errhp);
 while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
== OCI_SUCCESS)
 {
   printf("Subject: %s¥n", OCIStringPtr(envhp, deqmesg->subject));
   printf("Text: %s¥n", OCIStringPtr(envhp, deqmesg->data));
 }
 OCITransCommit(svchp, errhp, (ub4) 0);

 /* コンシューマ GREEN として msg_queue_multiple からデキュー */
 OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,(dvoid *)"GREEN",strlen("GREEN"),
    OCI_ATTR_CONSUMER_NAME, errhp);
```

```
  while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
 mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
 == OCI_SUCCESS)
  {
    printf("Subject: %s¥n", OCIStringPtr(envhp, deqmesg->subject));
    printf("Text: %s¥n", OCIStringPtr(envhp, deqmesg->data));
  }
  OCITransCommit(svchp, errhp, (ub4) 0);
}
```

複数コンシューマ・キューへのリモートのサブスクライバ / 受信者に対するメッセージのエンキューおよび伝播スケジューリング

```
/* サブスクライバ・リストの作成 : */
DECLARE
subscriber sys.aq$_agent;

/* 異なるアドレスを持つサブスクライバ RED および GREEN をサブスクライバ・リストに追加する : */
BEGIN
/* another_msg_queue キューからメッセージをデキューするサブスクライバ RED を同じデータベース内
に追加する */
subscriber := sys.aq$_agent('RED', 'another_msg_queue', NULL);
dbms_aqadm.add_subscriber(queue_name => 'msg_queue_multiple',
subscriber => subscriber);

  /* msg_queue_multiple から同じデータベース内の他のキューに対する伝播を計画する : */
dbms_aqadm.schedule_propagation(queue_name => 'msg_queue_multiple');

/* データベース・リンク another_db.world によって到達する別のデータベースの、msg_queue キュー
からメッセージをデキューする GREEN を追加する */
subscriber := sys.aq$_agent('GREEN', 'msg_queue@another_db.world', NULL);
dbms_aqadm.add_subscriber(queue_name => 'msg_queue_multiple',
subscriber => subscriber);

/* msg_queue_multiple からデータベース "another_database" 内の他のキューへの伝播を計画する :
*/
BEGIN
dbms_aqadm.schedule_propagation(queue_name => 'msg_queue_multiple', destination  =>
'another_db.world');

END;
/

DECLARE
```

```
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
recipients          dbms_aq.aq$_recipient_list_t;
message_handle      RAW(16);
message             aq.message_type;

/* キューに対するサブスクライバ、つまり、アドレス another_msg_queue の RED
およびアドレス msg_queue@another_db.world の GREEN に MESSAGE 1 をエンキューする : */
BEGIN
message := message_type('MESSAGE 1',
'This message is queued for queue subscribers.');

dbms_aq.enqueue(queue_name => 'msg_queue_multiple',
enqueue_options    => enqueue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

  /* 特定の受信者、つまり、アドレス another_msg_queue の RED および BLUE に MESSAGE 2 をエン
キューする */
message := message_type('MESSAGE 2',
'This message is queued for two recipients.');
recipients(1) := sys.aq$_agent('RED', 'another_msg_queue', NULL);
recipients(2) := sys.aq$_agent('BLUE', NULL, NULL);
message_properties.recipient_list := recipients;

dbms_aq.enqueue(queue_name => 'msg_queue_multiple',
enqueue_options    => enqueue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

COMMIT;
END;
/
```

注意 : アドレス another_msg_queue の RED はキューのサブスクライバであると同時に、MESSAGE 2 の指定された受信者です。それと対照的にアドレス msg_queue@another_db.world の GREEN は、受信者が指定されていないキュー内のメッセージ（この場合 MESSAGE 1）に対するサブスクライバでしかありません。BLUE は、キューのサブスクライバではありませんが、MESSAGE 2 の宛先に指定されています。

伝播のスケジュール解除

```
/* msg_queue_multiple から宛先 another_db.world への伝播の計画を解除する */
execute dbms_aqadm.unschedule_propagation(queue_name => 'msg_queue_multiple',
destination => 'another_db.world');
```

Message Grouping を使ったエンキューおよびデキュー

```
CONNECT aq/aq

EXECUTE dbms_aqadm.create_queue_table (
queue_table        => 'aq.msggroup',
queue_payload_type => 'aq.message_type',
message_grouping => dbms_aqadm.TRANSACTIONAL);

EXECUTE dbms_aqadm.create_queue (
queue_name   => 'msggroup_queue',
queue_table        => 'aq.msggroup');

EXECUTE dbms_aqadm.start_queue(queue_name => 'msggroup_queue');

/* 各トランザクションで 3 つのメッセージをエンキューする */
DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN

  /* 3 回ループし、繰りかえすごとにコミットする */
  FOR txnno in 1..3 LOOP

    /* 3 回ループし、繰りかえすごとにエンキューする */
    FOR mesgno in 1..3 LOOP
      message := message_type('GROUP#' || txnno,
      'Message#' || mesgno ||  ' in group' || txnno);

      dbms_aq.enqueue(queue_name      => 'msggroup_queue',
              enqueue_options        => enqueue_options,
      message_properties     => message_properties,
      payload                => message,
      msgid                  => message_handle);
    END LOOP;
    /* トランザクションのコミット */
```

```
    COMMIT;
  END LOOP;
END;
/

/* メッセージをグループ化してデキューする  */
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

no_messages   exception;
end_of_group  exception;

pragma exception_init (no_messages, -25228);
pragma exception_init (end_of_group, -25235);

BEGIN
dequeue_options.wait    := DBMS_AQ.NO_WAIT;
dequeue_options.navigation := DBMS_AQ.FIRST_MESSAGE;

LOOP
   BEGIN
   dbms_aq.dequeue(queue_name      => 'msggroup_queue',
          dequeue_options    => dequeue_options,
          message_properties => message_properties,
          payload            => message,
          msgid              => message_handle);

  dbms_output.put_line ('Message: ' || message.subject ||
  ' ... ' || message.text );

  dequeue_options.navigation := DBMS_AQ.NEXT_MESSAGE;

  EXCEPTION
    WHEN end_of_group THEN
      dbms_output.put_line ('Finished processing a group of messages');
      COMMIT;
      dequeue_options.navigation := DBMS_AQ.NEXT_TRANSACTION;
  END;
END LOOP;
EXCEPTION
  WHEN no_messages THEN
    dbms_output.put_line ('No more messages');
END;
/
```

AQ オブジェクトの削除

```
/* オブジェクト型に関連するオブジェクトをすべてクリーン・アップする： */
CONNECT aq/AQ;

EXECUTE dbms_aqadm.stop_queue (
queue_name => ’msg_queue’);

EXECUTE dbms_aqadm.drop_queue (
queue_name => ’msg_queue’);

EXECUTE dbms_aqadm.drop_queue_table (
queue_table => ’aq.msg’);

/* RAW 型に関連するオブジェクトをすべてクリーン・アップする ： */
EXECUTE dbms_aqadm.stop_queue (
        queue_name      => 'raw_msg_queue');

EXECUTE dbms_aqadm.drop_queue (
        queue_name      => 'raw_msg_queue');

EXECUTE dbms_aqadm.drop_queue_table (
            queue_table => 'aq.raw_msg');

/* 優先順位キューに関連するオブジェクトをすべてクリーン・アップする： */
EXECUTE dbms_aqadm.stop_queue (
        queue_name     => ’priority_msg_queue’);

EXECUTE dbms_aqadm.drop_queue (
        queue_name     => ’priority_msg_queue’);

EXECUTE dbms_aqadm.drop_queue_table (
queue_table   => ’aq.priority_msg’);

/* 複数コンシューマ・キューに関連するオブジェクトをすべてクリーン・アップする */
EXECUTE dbms_aqadm.stop_queue (
queue_name  => ’msg_queue_multiple’);

EXECUTE dbms_aqadm.drop_queue (
queue_name  => ’msg_queue_multiple’);

EXECUTE dbms_aqadm.drop_queue_table (
queue_table => ’aq.msg_multiple’);

drop type aq.message_type;
```

ロールおよび権限の取消し

```
CONNECT sys/change_on_install;
drop user aq;
```

Oracle アドバンスト・キューイング・リファレンス

リファレンスの概要

この項では、次の技術的仕様の詳細を説明します。

- Init ora パラメータ
- データ構造
- エージェント
- メッセージ ・プロパティ
- キューオプション
- 操作インタフェース
- 管理インタフェース
- 管理項目
- データ・オブジェクト

INIT.ORA パラメータ

AQ_TM_PROCESSES

キューメッセージに対して時間監視を行う場合には、init.ora パラメータ・ファイルに AQ_TM_PROCESSES というパラメータを指定する必要があります。このパラメータは、遅延および時間切れプロパティが指定されているメッセージに対して使われます。このパラメータは、0 ～ 10 の範囲に設定できます。それ以外の数値に設定すると、エラーとなります。このパラメータを 1 に設定すると、メッセージを監視するためのバックグラウンド・プロセスとして、キュー・モニター・プロセスが 1 つ作成されます。このパラメータが未指定の場合、または 0 に設定した場合、キュー・モニター・プロセスは作成されません。
キュー・モニターを起動および停止する管理インタフェースは、このパラメータを指定してキュー・モニター・プロセスをインスタンスの一部として起動した場合にだけ有効です。

パラメータ名 :	aq_tm_processes
パラメータ型 :	整数
パラメータ・クラス :	動的

設定可能値 :	0 〜 10
構文 :	aq_tm_processes = <0 〜 10>
プロセス名 :	ora_qmon_<oracle sid>
例 :	aq_tm_processes = 1

JOB_QUEUE_PROCESSES

伝播は、ジョブ・キュー (SNP) プロセスにより処理されます。インスタンスで開始されるジョブ・キュー・プロセスの数は、init.ora パラメータ JOB_QUEUE_PROCESSES で制御されます。このパラメータのデフォルト値は 0 です。メッセージを伝播するためには、このパラメータを少なくとも 1 に設定しなければなりません。伝播するメッセージがあるキューが多数ある場合、または伝播するメッセージの宛先が多数ある場合、ジョブ・キューに他のジョブがある場合、DBA では、このパラメータをより高い値に設定できます。

> **関連項目 :** JOB_QUEUE_PROCESSES の詳細は、『Oracle8 Server リファレンス・マニュアル』を参照してください。

COMPATIBLE

AQ の伝播機能を使用するためには、init.ora の COMPATIBLE パラメータを 8.0.4 に設定しなければなりません。特に、COMPATIBLE パラメータは、次の 3 つの条件でチェックされます。

1. AQ エージェント (sys.aq$_agent を参照) のアドレス・フィールドが、DBMS_AQADM.ADD_SUBSCRIBER コマンドで指定されている。
2. AQ エージェント (sys.aq$_agent を参照) のアドレス・フィールドが、dbms_aq.message_properties_t の recipient_list に指定されている。
3. DBMS_AQADM.SCHEDULE_PROPAGATION コマンドが使用されている。

次のコマンドを使用すると、8.0.4 の機能を使用した後に、8.0.3 にダウングレードできます。

```
ALTER DATABASE RESET COMPATIBILITY
```

次の条件下では、データベースを８ .0.3 互換モードで再起動できません。

1. まだ宛先に伝播されていないメッセージがキューにある。
2. まだ保留中の伝播するスケジュールがある。この場合、DBMS_AQADM.UNSCHEDULE_PROPAGATION コマンドを使用してそのスケジュールを削除できます。
3. リモート・サブスクライバがあるキューがある（つまり、sys.aq$_agent のアドレス・フィールドが NULL ではない）。この場合、DBMS_AQADM.REMOVE_SUBSCRIBER コマンドを使用して、リモート・サブスクライバを削除できます。

> **関連項目 :** 互換性の詳細は、移行ガイドのアップグレード / ダウングレードの項を参照してください。

データ構造

次のデータ構造は、「操作」および「管理」インタフェースで使われます。

オブジェクト名

用途 :

データベース・オブジェクトに命名します。この命名規則は、キューおよびキュー表、オブジェクト型に適用されます。

構文 :

```
object_name := VARCHAR2
object_name := [<schema_name>.]<name>
```

使用方法 :

オブジェクトの名前は、オプションのスキーマ名および名前で指定されます。スキーマ名を指定しない場合は、現行のスキーマが想定されます。スキーマ名および名前の長さは、どちらも最大 30 文字までです。ただし、キュー名およびキュー表名は最大 24 文字までです。

型名

用途 :

キューの型を定義します。

構文 :

```
type_name := VARCHAR2
type_name := <object_type> | "RAW"
```

使用方法 :

表 11–1　型名

パラメータ	説明
<object_types>	オブジェクト型作成の詳細は、『Oracle8 Server 概要』マニュアルを参照してください。オブジェクト型の属性の最大数は 900 です。
"RAW"	RAW 型のペイロードを格納するために、AQ ではペイロード・リポジトリとして LOB 列を持つキュー表を作成します。ペイロードのデータ・サイズは、最大で 32KB です。LOB 列は RAW ペイロードの格納に使われるため、AQ 管理者は、キュー表作成時に LOB 記憶域文字列を storage_clause パラメータに記述することで、LOB 表領域を選択し、LOB 記憶域を構成できます。

エージェント

用途 :

メッセージのプロデューサまたはコンシューマを識別します。

構文 :

```
TYPE sys.aq$_agent IS OBJECT (
    name            VARCHAR2(30),
    address         VARCHAR2(1024),
    protocol        NUMBER)
```

使用方法 :

表 11–2　エージェント

パラメータ	説明
name	メッセージのプロデューサまたはコンシューマの名前。
address	受信者のプロトコル固有のアドレス。プロトコルが 0(デフォルト) であれば、アドレスの形式は [schema.]queue[@dblink] です。
protocol	アドレスを解釈してメッセージを伝播するプロトコル。デフォルト（および現在唯一サポートされている）値は、0 です。

メッセージ・プロパティ

用途 :

「メッセージ・プロパティ」には、AQ で個々のメッセージを管理するために使われる情報が記述されています。これはエンキュー時に設定され、デキュー時にその値が戻されます。

構文 :

```
TYPE message_properties_t IS RECORD (
    priority          BINARY_INTEGER default 1,
    delay             BINARY_INTEGER default NO_DELAY,
    expiration        BINARY_INTEGER default NEVER,
    correlation       VARCHAR2(128) default NULL,
    attempts          BINARY_INTEGER,
    recipient_list    aq$_recipient_list_t,
    exception_queue   VARCHAR2(51) default NULL,
    enqueue_time      DATE,
    state BINARY_INTEGER)

TYPE aq$_recipient_list_t IS TABLE OF sys.aq$_agent
    INDEX BY BINARY_INTEGER
```

使用方法 :

表 11–3　メッセージ ・プロパティ

パラメータ	説明
priority	メッセージの優先順位を指定します。数値が小さいほど、優先順位が高いことを表します。優先順位には、負数も含め、任意の数値を指定できます。
delay	エンキューされたメッセージの遅延を指定します。遅延は、メッセージがデキュー可能になるまでの秒数を表します。msgid によるデキューでは、遅延仕様部が上書きされます。遅延付きでエンキューされたメッセージは、WAITING 状態になり、遅延が時間切れになると、READY 状態に変わります。DELAY 処理では、キュー・モニターが起動されている必要があります。遅延は、メッセージをエンキューするプロデューサによって設定されることに注意してください。 NO_DELAY: メッセージは、ただちにデキューできます。 数値 : メッセージを遅延する秒数です。

表 11–3　メッセージ ・プロパティ

パラメータ	説明
expiration	メッセージの時間切れまでの時間を指定します。このパラメータでは、メッセージがデキュー可能な期間を秒単位で指定します。このパラメータは遅延からのオフセットです。時間切れ処理では、キュー・モニターが起動されている必要があります。 NEVER: メッセージは時間切れになりません。 数値 : メッセージが READY 状態にとどまる秒数です。メッセージが時間切れになるまでデキューされなかった場合には、EXPIRED 状態で例外キューに移動されます。
correlation	メッセージに対してエンキュー時にプロデューサによって提供される識別を指定します。
attempts	このメッセージに対するデキューの試行回数を指定します。このパラメータは、エンキュー時には設定できません。
recipient_list	型定義の詳細は、「エージェント」の項を参照してください。 このパラメータは、複数コンシューマに対応したキューに対してだけ有効です。デフォルトの受信者は、キューのサブスクライバです。このパラメータは、デキュー時にコンシューマに対して戻されません。
exception_queue	メッセージが正常に処理されなかった場合に、そのメッセージを移動するキューの名前を指定します。メッセージは、デキューの異常終了した試行回数が max_retries を超えるか、またはそのメッセージが時間切れになった場合に移動されます。例外キューに移動されたすべてのメッセージは、EXPIRED 状態になっています。 デフォルトは、キュー表に対応付けられた例外キューです。メッセージを移動する時に指定した例外キューが存在しない場合には、そのメッセージはキュー表に対応付けられたデフォルトの例外キューに移動され、アラート・ファイルに警告が記録されます。デフォルトの例外キューが使われる場合は、デキュー時にこのパラメータによって NULL 値が戻されます。
enqueue_time	メッセージがエンキューされた時刻を指定します。この値はシステムによって決定され、ユーザーは設定できません。このパラメータは、エンキュー時には設定できません。
state	デキュー時のメッセージの状態を指定します。このパラメータは、エンキュー時には設定できません。 0: メッセージは処理可能状態です。 1: メッセージはまだ遅延状態にあります。 3: メッセージは処理され、保存されています。 4: メッセージは例外キューに移動されています。

キューオプション

エンキュー・オプション

用途 :

エンキュー操作で使用できるオプションを指定します。

構文 :

```
TYPE enqueue_options_t IS RECORD (
    visibility          BINARY_INTEGER default ON_COMMIT,
    relative_msgid      RAW(16) default NULL,
    sequence_deviation  BINARY_INTEGER default NULL)
```

使用方法 :

表 11–4

パラメータ	説明
visibility	エンキュー要求のトランザクション動作を指定します。 ON_COMMIT: エンキューは現行トランザクションの一部です。トランザクションがコミットされると、操作は完了です。これはデフォルトです。 IMMEDIATE: エンキューは現行トランザクションの一部ではありません。この操作は、それ自身で 1 つのトランザクションを構成しています。
relative_msgid	順序逸脱操作で参照されるメッセージのメッセージ識別子を指定します。このフィールドは、sequence_deviation に BEFORE を指定した場合にかぎり有効です。順序逸脱が指定されていない場合、このパラメータは無視されます。
sequence_deviation	エンキューするメッセージが、すでにキュー内にある他のメッセージより先にデキューされるべきかどうかを指定します。 *BEFORE: メッセージは、relative_msgid で指定したメッセージより前の順位にエンキューされます。* TOP: メッセージは、他のすべてのメッセージより前の順位にエンキューされます。 NULL: デフォルトです。

デキュー・オプション

用途 :

デキュー操作で使用できるオプションを指定します。

構文 :

```
TYPE dequeue_options_t IS RECORD (
    consumer_name    VARCHAR2(30) default NULL,
    dequeue_mode     BINARY_INTEGER default REMOVE,
    navigation       BINARY_INTEGER default NEXT_MESSAGE,
    visibility       BINARY_INTEGER default ON_COMMIT,
    wait             BINARY_INTEGER default FOREVER
    msgid            RAW(16) default NULL,
    correlation      VARCHAR2(128) default NULL)
```

使用方法 :

表 11–5　デキュー・オプション

パラメータ	説明
consumer_name	コンシューマの名前です。コンシューマ名が一致するメッセージだけがアクセスされます。キューが複数コンシューマ用に設定されていない場合には、このフィールドを NULL に設定する必要があります。
dequeue_mode	デキューに対応付けられたロック動作を指定します。 BROWSE: メッセージのロックを取得せずに、メッセージを読み込みます。これは SELECT 文と等価です。 LOCKED: メッセージを読み込み、書込みロックを取得します。ロックは、そのトランザクションが終了するまで継続します。これは SELECT FOR UPDATE 文と等価です。 REMOVE: メッセージを読み込み、更新または削除します。これはデフォルトです。メッセージは、保存プロパティに基づき、キュー表に保存できます。

表 11–5　デキュー・オプション

パラメータ	説明
navigation	取出し対象のメッセージの順位を指定します。まず最初に、順位が決定されます。次に、検索基準が適用されます。最後に、メッセージが取り出されます。 NEXT_MESSAGE: 取出し可能で、検索基準に適合する、次のメッセージを取り出します。前のメッセージがメッセージ・グループに属している場合には、AQ では、検索基準に適合し、同じメッセージ・グループに属する、次の取出し可能メッセージを取り出します。これはデフォルトです。 NEXT_TRANSACTION: 現行のトランザクション・グループの残りのメッセージ（存在する場合）をスキップして、次のトランザクション・グループの最初のメッセージを取り出します。このオプションは、現行のキューでメッセージのグループ化を使用できる場合にだけ使えます。 FIRST_MESSAGE: 取出し可能で、検索基準に適合する、最初のメッセージを取り出します。このオプションを使うと、順位がキューの先頭にリセットされます。
visibility	新規メッセージを、現行トランザクションの一部としてデキューするかどうかを指定します。visibility パラメータは、BROWSE モードを使っている場合には無視されます。 ON_COMMIT: デキューは現行トランザクションの一部として行われます。これはデフォルトです。 IMMEDIATE: デキューされたメッセージは、現行トランザクションの一部ではありません。このメッセージは、それ自身で 1 つのトランザクションを構成しています。
wait	現在検索基準に適合する取出し可能メッセージが存在しない場合の、待ち時間を指定します。 FOREVER: 無期限に待機します。これはデフォルトです。 NO_WAIT: 待機しません。 数値 : 待ち時間の秒数です。
msgid	デキューされるメッセージのメッセージ識別子を指定します。
correlation	デキューされるメッセージの相関識別子を指定します。パーセント符号 (%) およびアンダースコア (_) のような特別なパターン・マッチング文字が使用できます。2 つ以上のメッセージがそのパターンを満たしている場合には、デキューの順序は決定されません。

操作インタフェース

次のインタフェース・コールは、メッセージをキューからエンキューおよびデキューする場合に使用できます。

DBMS_AQ.ENQUEUE

用途 :

メッセージを、指定したキューに追加します。最も単純なケースとして、ユーザーが他のパラメータを指定せずにメッセージをエンキューする場合には、キュー名およびペイロードだけを指定します。

構文 :

```
DBMS_AQ.ENQUEUE (
    queue_name          IN      VARCHAR2,
    enqueue_options     IN      enqueue_options_t,
    message_properties  IN      message_properties_t,
    payload             IN      "<type_name>",
    msgid               OUT     RAW)
```

使用方法 :

表 11–6　DBMS_AQ.ENQUEUE

パラメータ	説明
queue_name (IN VARCHAR2)	このメッセージをエンキューするキュー名を指定します。このキューには、例外キューは指定できません。
enqueue_options (IN enqueue_option_t)	定義の詳細は、「エンキュー・オプション」の項を参照してください。
message_properties (IN message_properties_t)	定義の詳細は、「メッセージ・プロパティ」の項を参照してください。
payload (IN "<type_name>")	Oracle AQ では解釈されません。 ペイロードは、対応付けられたキュー表の仕様に従って指定する必要があります。NULL は 許容パラメータです。<type_name> の定義の詳細は、「型名」の項を参照してください。
msgid (OUT RAW)	システムで生成された、メッセージの識別です。これは、デキュー時にメッセージを識別するために使用できる、グローバルな一意識別子です。

順序逸脱の使用方法 :

enqueue_options の sequence_deviation パラメータを使うことで、2 つのメッセージ間の処理順序を変更できます。もう一方のメッセージが存在する場合は、その個別性を enqueue_options

のパラメータ relative_msgid で指定します。両方のメッセージの関連は、sequence_deviation パラメータで識別されます。

メッセージに対して sequence_deviation を指定すると、そのメッセージに対して指定できる遅延および優先順位の値に、いくつかの制限が生じます。このメッセージの遅延は、このメッセージが前にエンキューされるように指定されたメッセージの遅延に等しいか、またはそれより短くなければなりません。このメッセージの優先順位は、このメッセージが前にエンキューされるように指定されたメッセージの優先順位に等しいか、またはそれより高くなければなりません。

DBMS_AQ.DEQUEUE

用途 :

メッセージを、指定したキューからデキューします。

構文 :

```
DBMS_AQ.DEQUEUE (
    queue_name          IN      VARCHAR2,
    dequeue_options     IN      dequeue_options_t,
    message_properties  OUT     message_properties_t,
    payload             OUT    "<type_name>",
    msgid               OUT     raw)
```

使用方法 :

表 11–7　DBMS_AQ.DEQUEUE

パラメータ	説明
queue_name (IN VARCHAR2)	キュー名を指定します。
dequeue_options (IN dequeue_option_t)	定義の詳細は、「デキュー・オプション」の項を参照してください。
message_properties (OUT message_properties_t)	定義の詳細は、「メッセージ・プロパティ」の項を参照してください。
payload (OUT "<type_name>")	Oracle AQ では解釈されません。 ペイロードは、対応付けられたキュー表の仕様に従って指定する必要があります。 <type_name> の定義の詳細は、「型名」の項を参照してください。
msgid (OUT RAW)	システムで生成された、メッセージの識別です。

メッセージの検索基準とデキュー順序

デキューするメッセージの検索基準は、dequeue_options の consumer_name および msgid、correlation パラメータによって決定されます。msgid によって、デキューされるメッセージが一意に識別されます。相関識別子は、AQ では解釈されない、アプリケーション定義の識別子です。

msgid を指定しない場合は、READY 状態のメッセージだけがデキューされます。

デキューの順序は、dequeue_options の msgid および相関 ID で上書きされない限りは、キュー表の作成時に指定された値によって決定されます。

キュー操作には、データベース一貫性読込み機能が適用可能です。たとえば、BROWSE コールでは、ブラウズ・トランザクションの開始後にエンキューされたメッセージを参照できません。

キューによるナビゲート

デキュー時のデフォルトの NAVIGATION パラメータは、NEXT_MESSAGE です。つまり、後続のデキューでは、最初のデキューで取得したスナップショットを基盤とするキューからメッセージが抽出されます。特に、最初のデキュー・コマンドの後にエンキューされたメッセージは、キューに残っているメッセージがすべて処理されてから処理されます。すべてのメッセージがすでにキューにエンキューされている場合、またはキューに優先順位がない場合にはこれで十分です。しかし、キューにある最初のメッセージがデキュー・コマンドごとに処理される必要がある場合は、アプリケーションで FIRST_MESSAGE ナビゲーション・オプションを使用する必要があります。このことは、すでにエンキューされたメッセージの処理中に、優先順位の高いメッセージがキューに到着した場合に必要になります。

注意： 同時にエンキューされているメッセージがある場合には、FIRST_MESSAGE ナビゲーション・オプションを使用するとより効果的です。FIRST_MESSAGE を指定していないと、AQ では、最初のデキュー・コマンドの時点のスナップショットを継続して作成しなければならず、パフォーマンスの低下につながります。FIRST_MESSAGE オプションを指定すると、AQ では、すべてのデキュー・コマンドに新しいスナップショットを使用します。

メッセージのグループ化によるデキュー

同一トランザクションのメッセージが、メッセージのグループ化が可能なキューにエンキューされると、グループが作成されます。トランザクションに 1 つのメッセージしかエンキューされていない場合、実際に 1 つのグループが作成されます。1 つのトランザクションにグループ化できるメッセージ数の上限はありません。

メッセージのグループ化が使用できないキューでは、LOCKED または REMOVE モードのデキューは 1 つのメッセージだけをロックします。これに対し、グループの 1 部であるメッセージをデキューするデキュー操作では、グループ全体がロックされます。これは、グループのすべてのメッセージが最小単位として処理される必要がある場合に役に立ちます。

グループのすべてのメッセージがデキューされると、デキューによって、グループのすべてのメッセージが処理されたことを示すエラーが戻されます。その後で、アプリケーションでは、NEXT_TRANSACTION を使用して、次に利用できるグループからメッセージのデキューが開始されます。使用可能なグループがない場合は、デキューは、指定の WAIT 期間経過後タイムアウトになります。

操作インタフェースの列挙定数

BROWSE、LOCKED、REMOVE などの列挙定数を使う場合、PL/SQL 定数はそれを定義するパッケージの有効範囲とともに指定する必要があります。操作インタフェースに対応付けられたすべての型には、前に dbms_aq を付けなければなりません。たとえば、次のとおりです。

```
dbms_aq.BROWSE
```

表 11–8　操作インタフェースの列挙型

パラメータ	オプション
visibility	IMMEDIATE、ON_COMMIT
mode	BROWSE、LOCKED、REMOVE
navigation	FIRST_MESSAGE、NEXT_MESSAGE、NEXT_TRANSACTION
state	WAITING、READY、PROCESSED、EXPIRED
sequence_deviation	BEFORE、TOP
wait	FOREVER、NO_WAIT
delay	NO_DELAY
expiration	NEVER

管理インタフェース

構成情報は、DBMS_AQADM パッケージ内のプロシージャによって管理できます。管理インタフェースの誤った使用は、データベース・システムのパフォーマンスに大きな影響を与える可能性があるため、管理インタフェースは特権コマンドとして扱う必要があり、また管理パッケージへのアクセス権は、指定されたキュー管理者または特権ユーザーだけに付与する必要があります。最初は、SYS だけが DBMS_AQADM および DBMS_AQ 内のプロシージャに対する実行権限を付与されています。

権限およびアクセス制御

AQ 操作へのアクセス権限は、ロールを通じてユーザーに付与されます。これらのロールによって、AQ プロシージャの実行権限が与えられます。現在、オラクル社では、データベース・オブジェクト・レベルでのきめ細かなアクセス制御はサポートしていません。したがって、AQ_USER_ROLE を持つユーザーは、システム内のすべてのキューに対してエンキューおよびデキューできることになります。

管理者ロール AQ_ADMINISTRATOR_ROLE によって、DBMS_AQADM および DBMS_AQ パッケージ内のプロシージャに対する実行権限が付与されます。これには、すべての管理および操作インタフェースが含まれます。ユーザー 'SYS' は、AQ 管理者に AQ_ADMINISTRATOR_ROLE を付与しなければなりません。

ユーザー・ロール AQ_USER_ROLE によって、DBMS_AQ パッケージ内のプロシージャに対する実行権限が付与されます。これには、すべての操作インタフェースが含まれます。AQ 管理者は、AQ ユーザーに AQ_USER_ROLE を付与しなければなりません。

AQ オブジェクト型へのアクセス ユーザー 'SYS' は、まず最初にプロシージャ grant_type_access を実行して、AQ 管理者に AQ オブジェクト型へのアクセス権限を付与しなければなりません。権限を付与された AQ 管理者は、このプロシージャを実行して、他の AQ ユーザーに AQ オブジェクト型へのアクセス権限を付与できます。このプロシージャは、ユーザーが複数コンシューマ・キューに関係するなんらかの管理操作を行うことを希望した場合には、必ず実行されなければなりません。これには、CREATE_QUEUE_TABLE および CREATE_QUEUE、ADD_SUBSCRIBER、REMOVE_SUBSCRIBER が含まれます。

構文 :

```
PROCEDURE grant_type_access (user_name IN VARCHAR2);
```

PL/SQL 関数またはプロシージャから DBMS_AQ をコールする PL/SQL 関数またはプロシージャから DBMS_AQ をコールする場合は、明示的に EXECUTE 権限が付与される必要があります。AQ_USER_ROLE または AQ_ADMINISTRATOR_ROLE のどちらかからこの権利を継承することはできません。

構文 :

```
GRANT EXECUTE ON DBMS_AQ TO <user>;
```

例

1. Scott を AQ 管理者に任命します。

```
CONNECT sys/change_on_install
GRANT AQ_ADMINISTRATOR_ROLE to scott with admin option;
execute dbms_aqadm.grant_type_access('scott');
```

2. Scott は、Jones に AQ の使用権限を付与します。

```
CONNECT scott/tiger
GRANT AQ_USER_ROLE to jones;
```

3. Jones は、複数デキューが可能なキュー表を作成することを希望しています。

```
CONNECT scott/tiger
execute dbms_aqadm.grant_type_access('jones');
```

DBMS_AQADM.CREATE_QUEUE_TABLE

用途 : 事前定義済みの型のメッセージ用のキュー表を作成します。デキュー順序付け用のソート・キーを使う場合は、表の作成時に定義しなければなりません。この時点では、次のオブジェクトが作成されます。

1. キュー表に対応付けられた aq$_<queue_table_name>_e という名前のデフォルトの例外キュー。
2. AQ アプリケーションでキューデータの問合せに使われる、aq$<queue_table_name> という名前の読込み専用ビュー。
3. aq$_<queue_table_name>_t という名前の、キュー・モニター操作の索引。
4. デキュー操作用の複数コンシューマ・キューの場合の、aq$_<queue_table_name>_i という名前の索引または索引構成表（IOT）。

構文 :

```
DBMS_AQADM.CREATE_QUEUE_TABLE (
    queue_table        IN      VARCHAR2,
    queue_payload_type IN      VARCHAR2,
    storage_clause     IN      VARCHAR2 default NULL,
    sort_list          IN      VARCHAR2 default NULL,
    multiple_consumers IN      BOOLEAN default FALSE,
    message_grouping   IN      BINARY_INTEGER default NONE,
    comment            IN      VARCHAR2 default NULL,
    auto_commit        IN      BOOLEAN default TRUE)
```

使用方法 :

表 11–9　DBMS_AQADM.CREATE_QUEUE_TABLE

パラメータ	説明
queue_table (IN VARCHAR2)	作成するキュー表の名前を指定します。
queue_payload_type (IN VARCHAR2)	格納されているユーザー・データの型を指定します。このパラメータの有効値の詳細は、「型名」の項を参照してください。
storage_clause (IN VARCHAR2)	記憶領域パラメータを指定します。記憶領域パラメータは、 このキュー表が作成される時に、’CREATE TABLE’ 文に挿入されます。記憶領域パラメータは、次のパラメータの任意の組合せで構成されます。 PCTFREE および PCTUSED、INITRANS、MAXTRANS、TABLEPSACE、LOB、表記憶領域句。 これらのパラメータの使用方法は、SQL リファレンス・ガイドを参照してください。

表 11–9　DBMS_AQADM.CREATE_QUEUE_TABLE

パラメータ	説明
sort_list (IN VARCHAR2)	ソート・キーとして使う列を昇順で指定します。 Sort_list の書式は次のとおりです。'<sort_column_1>,<sort_column_2>'. 使用できる列名は、priority および enq_time です。両方の列が指定されている場合は、<sort_column_1> によって、最も左側になる列を定義できます。 いったんキュー表を特定の順序付け機能で作成すると、キュー表内のすべてのキューに同じデフォルトが継承されます。キュー表の順序は、いったんキュー表を作成した後には変更できません。 ソート・リストを指定しない場合は、キュー表内のすべてのキューは、エンキュー時刻に基づいて、昇順にソートされます。この順序は、FIFO 順序と同じです。 デフォルトの順序付けが定義されている場合でも、デキュー元は msgid または correlation を指定することで、デキューするメッセージを選択できます。msgid および correlation、sequence_deviation が指定されている場合、これらはデフォルトの順序付けより優先されます。
multiple_consumers (IN BOOLEAN)	FALSE: 表内に作成されたキューでは、1 つのメッセージは 1 つのコンシューマだけがデキューできます。これはデフォルトです。 TRUE: 表内に作成されたキューでは、1 つのメッセージを複数のコンシューマがデキューできます。ユーザーに対して、grant_type_access プロシージャを実行して型アクセス権限を付与しておく必要があります。
message_grouping (IN BINARY_INTEGER)	表内に作成されたキューについて、メッセージをグループ化する動作を指定します。 NONE: 各メッセージは個別に扱われます。 TRANSACTIONAL: 1 つのトランザクションの一部としてエンキューされた複数のメッセージを同じグループに属するものと見なします。デキューの際には、関連メッセージのグループとしてデキューしなければなりません。
comment (IN VARCHAR2)	キュー表に関するユーザー指定の説明を指定します。このユーザー・コメントは、キューカタログに追加されます。
auto_commit (IN BOOLEAN)	TRUE: 現行のトランザクションが存在する場合、CREATE_QUEUE_TABLE 操作が実行される前にコミットされます。この操作は、コールが戻されると継続されます。これはデフォルトです。 FALSE: 操作は現行トランザクションの一部とされ、コール元からコミットが発行された場合にだけ継続されます。

DBMS_AQADM.CREATE_QUEUE

用途 : 指定したキュー表にキューを作成します。すべてのキュー名は、スキーマ内において一意でなければなりません。キューは、CREATE_QUEUE で作成した後、START_QUEUE をコールすると使用可能になります。デフォルトでは、キューはエンキューとデキューの両方を使用禁止にして作成されます。

構文 :

```
DBMS_AQADM.CREATE_QUEUE (
    queue_name          IN      VARCHAR2,
    queue_table         IN      VARCHAR2,
    queue_type          IN      BINARY_INTEGER default
                                NORMAL_QUEUE,
    max_retries         IN      NUMBER default 0,
    retry_delay         IN      NUMBER default 0,
    retention_time      IN      NUMBER default 0,
    dependency_tracking IN      BOOLEAN default FALSE,
    comment             IN      VARCHAR2 default NULL,
    auto_commit         IN      BOOLEAN default TRUE)
```

使用方法 :

表 11–10　DBMS_AQADM.CREATE_QUEUE

パラメータ	説明
queue_name (IN VARCHAR2)	作成するキューの名前を指定します。
queue_table (IN VARCHAR2)	キューを格納するキュー表の名前を指定します。
queue_type (IN BINARY_INTEGER)	作成するキューが、例外キューか、通常キューかを指定します。 NORMAL_QUEUE: キューを、通常キューとして作成します。これはデフォルトです。 EXCEPTION_QUEUE: キューを、例外キューとして作成します。例外キューに対しては、デキュー操作だけが可能です。
max_retries (IN NUMBER)	REMOVE モードでの、メッセージに対するデキューの試行回数を制限します。回数のカウントは、アプリケーションがデキュー実行後にロールバックを発行するたびに増分されます。試行回数が max_retries に達すると、メッセージは例外キューに移動されます。デフォルトは 0 で、再試行が許可されていないことを意味します。

表 11–10　DBMS_AQADM.CREATE_QUEUE

パラメータ	説明
retry_delay (IN NUMBER)	アプリケーションのロールバック後、このメッセージの処理を再度スケジュールするまでの遅延時間を秒単位で指定します。デフォルトは 0 で、可能な限り早く、メッセージ処理が再試行されることを意味します。max_retries を 0 に設定すると、このパラメータは無効になります。retry_delay は、複数のコンシューマ・キューに対しては指定できません。
retention_time (IN NUMBER)	メッセージがキューからデキューされた後に、キュー表内に保存される秒数を指定します。 INFINITE: メッセージは無期限に保存されます。 数値 : メッセージを保存する秒数です。デフォルトは 0 で、保存されません。
dependency_tracking (IN BOOLEAN)	将来の使用に備えて確保されています。 FALSE: これはデフォルトです。 TRUE: このリリースでは許可されていません。
comment (IN VARCHAR2)	キューに関するユーザー指定の説明です。このユーザー・コメントは、キュー・カタログに追加されます。
auto_commit (IN BOOLEAN)	TRUE: 現行のトランザクションが存在する場合、CREATE_QUEUE 操作が実行される前にコミットされます。この操作は、コールが戻されると継続されます。これはデフォルトです。 FALSE: 操作は現行トランザクションの一部とされ、コール元からコミットが発行された場合にだけ継続されます。

DBMS_AQADM.DROP_QUEUE_TABLE

用途 :

既存のキュー表を削除します。キュー表を削除する前には、そのキュー表内のすべてのキューを停止して削除しなければなりません。

構文 :

```
DBMS_AQADM.DROP_QUEUE_TABLE (
    queue_table       IN    VARCHAR2,
    force             IN    BOOLEAN default FALSE,
    auto_commit       IN    BOOLEAN default TRUE)
```

使用方法 :

表 11–11　DBMS_AQADM.DROP_QUEUE_TABLE

パラメータ	説明
queue_table (IN VARCHAR2)	削除するキュー表の名前を指定します。
force (IN BOOLEAN)	FALSE: 表内にキューが存在する場合、操作は失敗します。これはデフォルトです。 TRUE: 表内のすべてのキューは、自動的に停止および削除されます。
auto_commit (IN BOOLEAN)	TRUE: 現行のトランザクションが存在する場合、DROP_QUEUE_TABLE 操作が実行される前にコミットされます。この操作は、コールが戻されると継続されます。これはデフォルトです。 FALSE: 操作は現行トランザクションの一部とされ、コール元からコミットが発行された場合にだけ継続されます。

DBMS_AQADM.DROP_QUEUE

用途 :

既存のキューを削除します。STOP_QUEUE がすでにコールされ、そのキューに対するエンキューおよびデキューの両方が使用禁止にされていない限り、DROP_QUEUE は許可されません。すべてのキューデータは、削除操作の一部として削除されます。

構文 :

```
DBMS_AQADM.DROP_QUEUE (
    queue_name        IN          VARCHAR2,
    auto_commit       IN          BOOLEAN default TRUE)
```

使用方法 :

表 11–12　DBMS_AQADM.DROP_QUEUE

パラメータ	説明
queue_name (IN VARCHAR2)	削除するキューの名前を指定します。

表 11–12　DBMS_AQADM.DROP_QUEUE

auto_commit (IN BOOLEAN)	TRUE: 現行のトランザクションが存在する場合、DROP_QUEUE 操作が実行される前にコミットされます。この操作は、コールが戻されると継続されます。これはデフォルトです。 FALSE: 操作は現行トランザクションの一部とされ、コール元からコミットが発行された場合にだけ継続されます。

DBMS_AQADM.ALTER_QUEUE

用途 :

キューの既存のプロパティを変更します。変更できるのは、max_retries および retry_delay、retention_time だけです。

構文 :

```
DBMS_AQADM.ALTER_QUEUE (
    queue_name        IN    VARCHAR2,
    max_retries       IN    NUMBER default NULL,
    retry_delay       IN    NUMBER default NULL,
    retention_time    IN    NUMBER default NULL,
    auto_commit       IN    BOOLEAN default TRUE)
```

使用方法 :

表 11–13　DBMS_AQADM.ALTER_QUEUE

パラメータ	説明
queue_name (IN VARCHAR2)	変更するキューの名前を指定します。
max_retries (IN NUMBER)	REMOVE モードでの、メッセージに対するデキューの試行回数を制限します。回数のカウントは、アプリケーションがデキュー実行後にロールバックを発行するたびに増分されます。再試行のいずれかが時間切れ時刻を過ぎると、再試行はそれ以上行われません。デフォルトは NULL で、この値が変更されないことを意味します。
retry_delay (IN NUMBER)	アプリケーションのロールバック後、このメッセージの処理を再度スケジュールするまでの遅延時間を秒単位で指定します。デフォルトは NULL で、この値が変更されないことを意味します。
retention_time (IN NUMBER)	メッセージがデキューされた後に、キュー表内に保存される時間を秒単位で指定します。デフォルトは NULL で、この値が変更されないことを意味します。

表 11–13　DBMS_AQADM.ALTER_QUEUE

パラメータ	説明
auto_commit (IN BOOLEAN)	TRUE: 現行のトランザクションが存在する場合、ALTER_QUEUE 操作が実行される前にコミットされます。この操作は、コールが戻されると継続されます。これはデフォルトです。 FALSE: 操作は現行トランザクションの一部とされ、コール元からコミットが発行された場合にだけ継続されます。

DBMS_AQADM.START_QUEUE

用途 :

指定したキューに対するエンキューまたはデキュー、あるいはその両方を使用可能にします。管理者は、キューを作成した後、START_QUEUE を使ってそのキューを使用可能にしなければなりません。デフォルトでは、ENQUEUE および DEQUEUE の両方を使用可能にします。例外キューに対しては、デキュー操作だけが可能です。この操作は、コールが完了し、かつコールにトランザクションの特性が一切ない場合にだけ有効になります。

構文 :

```
DBMS_AQADM.START_QUEUE (
    queue_name      IN      VARCHAR2,
    enqueue         IN      BOOLEAN default TRUE,
    dequeue         IN      BOOLEAN default TRUE)
```

使用方法 :

表 11–14　DBMS_AQADM.START_QUEUE

パラメータ	説明
queue_name (IN VARCHAR2)	使用可能にするキューの名前を指定します。
enqueue (IN BOOLEAN)	このキューについて、ENQUEUE を使用可能にするかどうかを指定します。 TRUE: ENQUEUE を使用可能にします。これはデフォルトです。 FALSE: 現行の設定を変更しません。
dequeue (IN BOOLEAN)	このキューについて、DEQUEUE を使用可能にするかどうかを指定します。 TRUE: DEQUEUE を使用可能にします。これはデフォルトです。 FALSE: 現行の設定を変更しません。

DBMS_AQADM.STOP_QUEUE

用途 :

指定したキューに対するエンキューまたはデキュー、あるいはその両方を使用禁止にします。デフォルトでは、ENQUEUE および DEQUEUE の両方が使用禁止になります。キューは、それに対する未完了のトランザクションが存在する場合には停止できません。この操作は、コールが完了し、かつコールにトランザクションの特性が一切ない場合にだけ有効になります。

構文 :

```
DBMS_AQADM.STOP_QUEUE (
    queue_name      IN      VARCHAR2,
    enqueue         IN      BOOLEAN default TRUE,
    dequeue         IN      BOOLEAN default TRUE,
    wait            IN      BOOLEAN default TRUE)
```

使用方法 :

表 11–15 DBMS_AQADM.STOP_QUEUE

パラメータ	説明
queue_name (IN VARCHAR2)	使用禁止にするキューの名前を指定します。
enqueue (IN BOOLEAN)	このキューについて、ENQUEUE を使用禁止にするかどうかを指定します。 TRUE: ENQUEUE を使用禁止にします。これはデフォルトです。 FALSE: 現行の設定を変更しません。
dequeue (IN BOOLEAN)	このキューについて、DEQUEUE を使用禁止にするかどうかを指定します。 TRUE: DEQUEUE を使用禁止にします。これはデフォルトです。 FALSE: 現行の設定を変更しません。
wait (IN BOOLEAN)	wait パラメータでは、未完了のトランザクションが完了するまで待機するかどうかを指定できます。 TRUE: 未完了のトランザクションがある場合には待機します。この状態では、新しいトランザクションがこのキューに対してエンキューまたはデキューすることは許可されません。 FALSE: 正常またはエラー状態で即時に復帰します。

DBMS_AQADM.ADD_SUBSCRIBER

用途 :

デフォルトのサブスクライバをキューに追加します。プログラムでは、メッセージを受信者の特定のリストまたはサブスクライバのデフォルトのリストにエンキューできます。この操作は、複数コンシューマに対応したキューに対してだけ正常に実行できます。この操作はただちに有効となり、この操作を含んでいるトランザクションはコミットされます。このコールが完了した後に実行されるエンキュー要求には、新しい動作が反映されます。ユーザーに対して、grant_type_access プロシージャを実行して型アクセス権限を付与しておく必要があります。

構文 :

```
DBMS_AQADM.ADD_SUBSCRIBER(
    queue_name     IN   VARCHAR2,
    subscriber     IN   sys.aq$_agent)
```

使用方法 :

表 11–16　DBMS_AQADM.ADD_SUBSCRIBER

パラメータ	説明
queue_name (IN VARCHAR2)	キューの名前を指定します。
subscriber (IN aq$_agent)	「エージェント」の項にある定義を参照してください。

DBMS_AQADM.REMOVE_SUBSCRIBER

用途 :

デフォルトのサブスクライバをキューから取り消します。この操作はただちに有効となり、この操作を含んでいるトランザクションはコミットされます。既存メッセージ内のこのサブスクライバに対するすべての参照は、操作の一部として取り消されます。ユーザーに対して、grant_type_access プロシージャを実行して型アクセス権限を付与しておく必要があります。

構文 :

```
DBMS_AQADM.REMOVE_SUBSCRIBER(
    queue_name         IN         VARCHAR2,
    subscriber         IN         sys.aq$_agent)
```

使用方法 :

表 11–17

パラメータ	説明
queue_name (IN VARCHAR2)	キュー名を指定します。
subscriber (IN aq$_agent)	「エージェント」の項にある定義を参照してください。

DBMS_AQADM.SCHEDULE_PROPAGATION

用途 :

キューから特定の dblink で識別される宛先へのメッセージの伝播が計画されます。また、メッセージは、NULL の宛先を指定すると、同じデータベース内の他のキューに伝播されます。メッセージが同じキューにでも異なるキューにでも、同じ宛先に複数の受信者を持つ場合、メッセージは、すべての受信者に同時に伝播されます。

構文 :

```
DBMS_AQADM.SCHEDULE_PROPAGATION(
    src_queue_name  IN    VARCHAR2,
    destination     IN    VARCHAR2 default NULL
    start_time      IN    DATE default SYSDATE,
    duration        IN    NUMBER default NULL,
    next_time       IN    VARCHAR2 default NULL,
    latency         IN    NUMBER default 60)
```

使用方法 :

表 11–18　DBMS_AQADM.SCHEDULE_PROPAGATION

パラメータ	説明
src_queue_name (IN VARCHAR2)	伝播するメッセージがあるソース・キューの名前（スキーマ名を含む）を指定します。スキーマ名を指定しない場合は、管理ユーザーのスキーマ名がデフォルトになります。
destination (IN VARCHAR2)	宛先の dblink を指定します。この宛先の受信者にソース・キューのメッセージが伝播されます。NULL の場合は、宛先はローカル・データベースであり、メッセージはローカル・データベースの他のキューに伝播します。このフィールドの長さは現在最大 128 バイトに制限されており、名前が完全な修飾名ではない場合、デフォルトのドメイン名が使用されます。
start_time (IN DATE)	ソース・キューから宛先へのメッセージ用の伝播枠を最初に開始する時間を指定します。
duration (IN NUMBER)	伝播枠の継続期間を秒で指定します。NULL 値は、伝播枠が永続的であるまたは伝播の計画が取り消されるまでを意味します。
next_time (IN VARCHAR2)	現在の枠の終了から次の伝播枠の開始を計算する日付関数。この値が NULL であれば、伝播は現行の枠で停止します。たとえば、毎日同時刻に枠を開始するには、next_time には、’SYSDATE + 1 - duration/86400’ と指定します。
latency (IN NUMBER)	エンキュー後、伝播されるメッセージの伝播枠における最大待機時間 (秒)。たとえば、待機時間が 60 秒で伝播枠の期間中の場合は、伝播するメッセージがなければ、その宛先のそのキューのメッセージは、少なくとも 60 秒間は伝播されません。キューが、指定の宛先に伝播されるメッセージを再びチェックするまで少なくとも 60 秒かかります。待機時間が 600 の場合、キューは 10 分間チェックされません。待機時間が 0 の場合は、ジョブ・キュー・プロセスは、宛先にエンキューされるメッセージを待機し、メッセージがエンキューされるとすぐに伝播します。

DBMS_AQADM.UNSCHEDULE_PROPAGATION

用途 :

あるキューから特定の dblink で識別される宛先への以前に計画されたメッセージの伝播計画を解除します。

構文 :

```
DBMS_AQADM.UNSCHEDULE_PROPAGATION(
```

```
    src_queue_name   IN    VARCHAR2,
    destination      IN    VARCHAR2 default NULL)
```

使用方法 :

表 11–19　DBMS_AQADM.UNSCHEDULE_PROPAGATION

パラメータ	説明
src_queue_name (IN VARCHAR2)	伝播するメッセージがあるソース・キューの名前（スキーマ名を含む）を指定します。スキーマ名を指定しない場合は、管理ユーザーのスキーマ名がデフォルトになります。
destination (IN VARCHAR2)	宛先の dblink を指定します。この宛先の受信者にソース・キューのメッセージが伝播されます。NULL の場合は、宛先はローカル・データベースであり、メッセージはローカル・データベースの他のキューに伝播します。このフィールドの長さは現在最大 128 バイトに制限されており、名前が完全な修飾名ではない場合、デフォルトのドメイン名が使用されます。

DBMS_AQADM.VERIFY_QUEUE_TYPES

用途 :

ソースおよび宛先に同じ型があるかを検証します。検証の結果は、sys.aq$_message_types 表に格納され、このコマンドの以前の出力はすべて上書きされます。

構文 :

```
DBMS_AQADM.SCHEDULE_PROPAGATION(
    src_queue_name    IN    VARCHAR2,
    dest_queue_name   IN    VARCHAR2,
    destination       IN    VARCHAR2 default NULL
    rc                OUT   BINARY_INTEGER)
```

使用方法 :

表 11–20　DBMS_AQADM.SCHEDULE_PROPAGATION

パラメータ	説明
src_queue_name (IN VARCHAR2)	伝播するメッセージがあるソース・キューの名前（スキーマ名を含む）を指定します。スキーマ名を指定しない場合は、ユーザーのスキーマ名がデフォルトになります。
dest_queue_name (IN VARCHAR2)	伝播するメッセージがある宛先キューの名前（スキーマ名を含む）を指定します。スキーマ名を指定しない場合は、ユーザーのスキーマ名がデフォルトになります。
destination (IN VARCHAR2)	宛先の dblink を指定します。宛先キュー名は dblink に指定されているデータベースにあります。宛先が NULL の場合、宛先キューはソース・キューと同じデータベースになります。このフィールドの長さは現在最大 128 バイトに制限されており、名前が完全な修飾名ではない場合、デフォルトのドメイン名が使用されます。
rc (OUT BINARY_INTEGER)	プロシージャの結果のコードが戻されます。エラーがなく、ソースと宛先キューのタイプが一致する場合結果は 1 で、一致しない場合結果は 0 です。Oracle エラーが発生した場合は、rc に戻されます。

管理インタフェースの列挙定数

BROWSE、LOCKED、REMOVE などの列挙定数を使う場合、それを定義するパッケージの有効範囲とともにその記号を指定する必要があります。管理インタフェースに対応付けられたすべての型には、前に dbms_aqadm を付けなければなりません。たとえば、次のとおりです。

```
dbms_aqadm.NORMAL_QUEUE
```

表 11–21　管理インタフェースの列挙型

パラメータ	オプション
retention	INFINITE
message_grouping	TRANSACTIONAL、NONE
queue_type	NORMAL_QUEUE、EXCEPTION_QUEUE

データベース・オブジェクト

キュー表のビュー

これは、メッセージ・データが格納されているキュー表のビューです。このビューは、aq$<queue_table_name> という名前で、キュー表ごとに自動的に作成されます。このビューは、キュー・データの問合せに使う必要があります。デキュー履歴データ（時刻およびユーザー識別、トランザクション識別）は、1 人のコンシューマのキューについてだけ有効です。複数コンシューマのキューでのメッセージのデキュー履歴の詳細は、次の項を参照してください。

管理者は、キューまたはキュー表の内容を分析および検討するために、すべての SQL 文または SQL ツールを使用できます。SQL を使うことで、メッセージのメタデータまたはペイロード（あるいはその両方）に完全にアクセスできます。複数のトランザクションを関係付けるには、ENQ_TXN_ID および DEQ_TXN_ID を使います。メッセージ m2 の ENQ_TXN_ID が m1 の DEQ_TXN_ID と同じである場合、m2 は m1 を使用したトランザクションに作成されます （関連するメッセージを識別するためには、SQL 文の中で CONNECT BY を使うこともできます)。保存されたメッセージが、AQ で自動的に取り消されない場合には、取り消してください。キュー・メタデータの一貫性が損なわれる恐れがあるため、メッセージを更新または変更してはいけません。SQL を使って AQ のエラーを修正する場合は、事前にオラクル社技術サポートまでご連絡ください。

表 11–22　キュー表のビュー

列名および説明	Null かどうか	型
QUEUE ― キュー名		VARCHAR2(30)
MSG_ID― メッセージの一意識別子		RAW(16)
CORR_ID ― ユーザー指定の相関識別子		VARCHAR2(128)
MSG_PRIORITY ― メッセージ優先順位		NUMBER
MSG_STATE― このメッセージの状態		VARCHAR2(9)
DELAY ― メッセージが遅延する秒数		DATE
EXPIRATION ― メッセージが READY 状態になってから､時間切れになるまでの秒数		NUMBER
ENQ_TIME ― エンキュー時刻		DATE
ENQ_USER_ID ― エンキューしたユーザーのユーザー ID		NUMBER
ENQ_TXN_ID ― エンキューしたトランザクションのトランザクション ID	Not Null	VARCHAR2(30)

表 11–22　キュー表のビュー

列名および説明	Null かどうか	型
DEQ_TIME — デキュー時刻		DATE
DEQ_USER_ID — デキューしたユーザーのユーザー ID		NUMBER
DEQ_TXN_ID — デキューしたトランザクションのトランザクション ID		VARCHAR2(30)
RETRY_COUNT — 再試行回数		NUMBER
EXCEPTION_QUEUE_OWNER — 例外キュースキーマ		VARCHAR2(30)
EXCEPTION_QUEUE — 例外キュー名		VARCHAR2(30)
USER_DATA — ユーザー・データ		BLOB

DBA_QUEUE_TABLES

このビューでは、データベース内で作成されたすべてのキュー表の名前および型が記述されます。

表 11–23　DBA_QUEUE_TABLES

列名および説明	Null かどうか	型
OWNER — キュー表スキーマ		VARCHAR2(30)
QUEUE_TABLE — キュー表名		VARCHAR2(30)
TYPE — ペイロードの型		VARCHAR2(7)
OBJECT_TYPE — オブジェクト型が存在する場合、その名前		VARCHAR2(61)
SORT_ORDER — ユーザー指定のソート順		VARCHAR2(22)
RECIPIENTS — SINGLE または MULTIPLE		VARCHAR2(8)
MESSAGE_GROUPING — NONE または TRANSACTIONAL		VARCHAR2(13)
USER_COMMENT — キュー表に対するユーザー・コメント		VARCHAR2(50)

USER_QUEUE_TABLES

このビューは、ユーザーのスキーマ内のキュー表だけを表示する点を除いては、DBA_QUEUES_TABLES と同じです。このビューには、OWNER の列は含まれません。

DBA_QUEUES

ユーザーは、個々のキューについて操作上の特性を指定できます。DBA_QUEUES には、データベース内のあらゆるキューについての関連情報を含むビューが含まれます。

表 11–24　DBA_QUEUES

列名および説明	Null かどうか	型
OWNER — キューのスキーマ名	Not Null	VARCHAR2(30)
NAME — キュー名	Not Null	VARCHAR2(30)
QUEUE_TABLE — このキューが格納されるキュー表	Not Null	VARCHAR2(30)
QID — 一意のキュー識別子	Not Null	NUMBER
QUEUE_TYPE — キューの型		VARCHAR2(15)
MAX_RETRIES — デキューの許容試行回数		NUMBER
RETRY_DELAY — 再試行までの秒数		NUMBER
ENQUEUE_ENABLED — YES または NO		VARCHAR2(7)
DEQUEUE_ENABLED — YES または NO		VARCHAR2(7)
RETENTION — メッセージがデキュー後に保存される秒数		VARCHAR2(40)
USER_COMMENT — キューに対するユーザー・コメント		VARCHAR2(50)

USER_QUEUES

このビューは、ユーザーのスキーマ内のキューだけを表示する点を除けば、DBA_QUEUES と同じです。このビューには、OWNER の列は含まれません。

DBMS_AQADM.QUEUE_SUBSCRIBERS

用途 : キューに対するサブスクライバのリストを取得します。

構文 :

```
DBMS_AQADM.QUEUE_SUBSCRIBERS(
    queue_name    IN     VARCHAR2)
RETURN aq$_subscriber_list_t
```

使用方法 : この関数は、aq$_agent の PL/SQL 表を戻します。これは、1 つのキューのすべてのサブスクライバのリストを得るために使用できます。

例 :

```
DECLARE
    subs            dbms_aqadm.aq$_subscriber_list_t;
    nsubs           BINARY_INTEGER;
    i               BINARY_INTEGER;
BEGIN
    subs  := dbms_aqadm.queue_subscribers('Q1DEF');
    nsubs := subs.COUNT;
    FOR i IN 0..nsubs-1 LOOP
        dbms_output.put_line(subs(i).name);
    END LOOP;
END;
/
```

DBA_QUEUE_SCHEDULES

用途 : このビューは、メッセージの伝播の現行スケジュールを示します。

表 11–25 DBA_QUEUE_SCHEDULES

列名および説明	Null かどうか	型
SCHEMA	Not Null	VARCHAR2(30)
ソース・キューのスキーマ名		
QNAME	Not Null	VARCHAR2(30)
ソース・キュー名		
DESTINATION	Not Null	VARCHAR2(128)
宛先名。現在は DBLINK 名に制限されています。		
START_DATE		DATE
伝播開始日付。デフォルトの日付書式。		

表 11–25　DBA_QUEUE_SCHEDULES

列名および説明	Null かどうか	型
START_TIME		VARCHAR2(8)
伝播を開始する時刻。HH:MI:SS 形式。		
PROPAGATION_WINDOW		NUMBER
伝播枠の持続期間（秒）		
NEXT_TIME		VARCHAR2(128)
次の伝播枠の開始を計算する関数。		
LATENCY		NUMBER
伝播枠の間でメッセージを伝播するための最大待機時間		

複数コンシューマ・メッセージの受信者およびデキュー履歴

キュー表のビューを使うと、1 つのコンシューマのキュー・メッセージのデキュー履歴を得ることができます。複数コンシューマ・キューに含まれるメッセージの受信者またはデキュー履歴のリストを問い合せるには、そのキュー表に対して、対象のメッセージについての SQL 問合せを実行する必要があります。

たとえば、キュー表 sys.queue_tab にある msgid’105E7A2EBFF11348E03400400B40F149’ のメッセージのデキュー履歴を表示するには、次の問合せを実行する必要があります。この問合せでは、メッセージのコンシューマが 1 行に 1 つずつ戻されます。

```
SELECT consumer, transaction_id, deq_time, deq_user
FROM THE(select cast(history as sys.aq$_dequeue_history_t)
    FROM sys.queue_tab
    WHERE msgid='105E7A2EBFF11348E03400400B40F149');
```

エラー・メッセージ

AQ に関するエラーメッセージは、次の 2 つの範囲に分けられます。

24000 ― 24099

25200 ― 25299

管理項目

パフォーマンス

キューは、データベースの表に格納されます。キュー操作のパフォーマンス特性は、基になるデータベース操作にきわめて類似しています。

表および索引の構造

キューのパフォーマンス特性を理解するには、AQ オブジェクト用の表および索引レイアウトを理解することが重要です。

キュー表を作成すると、およそ 25 列のデータベースの表が作成されます。これらの列には、AQ のメタデータおよびユーザー定義のペイロードが格納されます。このペイロードの型は、オブジェクト型または RAW です。AQ メタデータには、オブジェクト型およびスカラー型が含まれています。キュー表には、1 つのビューと 2 つの索引が作成されます。このビューを使うことで、ユーザーはメッセージ・データを問い合せることができます。索引は、メッセージ・データへのアクセス時間を短縮するために使われます。作成されるオブジェクトの詳細は、「キュー表の作成」コマンドの項を参照してください。

スループット

エンキュー操作のコード・パスは、2 つの索引を持つ複数列表への挿入に類似しています。デキュー操作のコード・パスは、同様の表に対する選択および削除操作に似ています。これらの操作は、PL/SQL 関数を使って行われます。

可用性

Oracle Parallel Server(OPS) を使うことで、キューデータに対する可用性の高いアクセスを実現できます。キューは、データベースの表を使ってインプリメントされます。キューの「テール」と「ヘッド」は、極度のホット・スポットになる可能性があります。ホット・スポットが存在すると、OPS は十分に性能を発揮できないので、キューへの通常のアクセスを、1 つのインスタンスだけからに制限することをお薦めします。インスタンス障害の場合、障害が起こったインスタンスによって管理されていたメッセージは、障害のないインスタンスの内の 1 つによって即時に処理されます。

拡張性

キュー操作の拡張性は、基になるデータベース操作の拡張性に類似しています。マルチスレッド・サーバー（MTS）環境で、wait オプション付きのデキュー操作が発行されると、共有サーバー・プロセスでは、待ち時間を含むコール所要時間の間、デキュー操作だけが行われます。このようなプロセスが多数存在すると、パフォーマンスおよび可用性に重大な問題が生じ、共有サーバー・プロセスのデッドロックを引き起こす可能性があります。そのため、wait オプション付きのデキュー要求は、専用のサーバー・プロセスだけから発行することをお薦めします。この制限は、強制ではありません。

伝播の最適化

JOB_QUEUE_PROCESSES の数の設定において、DBA は、このニーズが、伝播するメッセージのあるキューの数およびメッセージが伝播される宛先（キューではなく）の数によって決まることに注意する必要があります。

信頼性および回復可能性

キューデータに対しては、データベースの標準的な信頼性および回復可能性特性が当てはまります。

Enterprise Manager サポート

Enterprise Manager では、「管理インタフェース」の項にリストした管理機能の一部について、GUI をサポートします。

次のものが含まれます。

1. プロパティを参照するためのスキーマ・マネージャの 1 部としてのキュー
2. キューの起動および停止
3. 伝播のスケジュールおよびスケジュール解除
4. サブスクライバの追加および削除
5. 現在の伝播スケジュールの表示

キュー・データのインポートおよびエクスポート

キューは、表にインプリメントされます。キューをインポート / エクスポートすると、基になるキュー表および関連するディクショナリ表もインポート / エクスポートされます。キューは、キュー表に格納されたキューを単位としてだけインポート / エクスポートできます。

キュー表をエクスポートすると、表定義情報とキュー・データの両方がエクスポートされます。キュー表をインポートすると、キュー・ディクショナリはエクスポート動作プロシージャによってメンテナンスされます。キュー表データも同時にエクスポートされるので、キュー表データを移動する際のアプリケーション・レベルでのデータ整合性は、ユーザー側の責任でメンテナンスする必要があります。

既存データのあるキュー表にキュー・データをインポートすることは、お薦めできません。表モードでのインポートの際に、インポート先のサイトにすでにキュー表が存在すると、古いキュー表定義およびキュー定義は削除および再作成されます。そのため、インポート以前に作成されたキュー表およびキュー定義は失われます。

複数受信者のキュー表の EXPORTS および IMPORTS の実行

複数受信者をサポートするすべてのキュー表には、重要なキュー・メタデータを収めた索引構成表（IOT）があります。このメタデータはキューの操作には不可欠なので、ユーザーはインポート後の作業で使う表に、キューのキュー表だけでなく、IOT もエクスポートおよびインポートしなければなりません。キュー表を含むスキーマをエクスポートすると、IOT も自動的にエクスポートされます。インポート時の動作も、これと同じです。このメタデータ表は、キュー表内の一部の行の ROWID を含んでいるので、メタデータ表のインポートの際に古くなった ROWID についての注意が発行されます。古くなった ROWID は、インポート・プロセスの一部としてキューイング・システムによって自動的に訂正されるので、このメッセージは無視してもかまいません。ただし、インポート中に他の問題（ロールバック・セグメント領域の不足など）が発生した場合は、この問題を修正し、インポートを再実行する必要があります。

トラブルシューティング

この項では、メッセージの伝播の問題を診断するためのいつくかのトラブルシューティングについて助言します。

メッセージ履歴

メッセージが正常に宛先に伝播されると、AQ では、メッセージ履歴が更新されます。メッセージ履歴は、キュー表にコレクションとして保存されます。管理者は、SQL 問合せを実行して、メッセージが伝播されたかどうかを判断できます。たとえば、キュー表 aqadmn.queue_tab に次の msgid を持つメッセージがあるとします。

105E7A2EBFF11348E03400400B40F149'

このメッセージが宛先 'boston' に伝播されたかをチェックするために、次の問合せを実行できます。

```
SELECT consumer, transaction_id, deq_time, deq_user, propagated_msgid
   FROM THE(select cast(history as sys.aq$_dequeue_history_t)
   FROM adadmn.queue_tab
      WHERE msgid='105E7A2EBFF11348E03400400B40F149')
      WHERE consumer LIKE '%BOSTON%';
```

NULL 以外の transaction_id は、メッセージが正常に伝播されたことを示します。さらに、deq_time は伝播時刻、deq_user は伝播に使用されたユーザー ID、propagated_msgid は宛先にエンキューされたメッセージの msgid を示します。msgid でメッセージをキュー表で検索できない場合には、管理者は、メッセージの履歴について例外キュー（例外キューが他のキュー表にある場合）をチェックできます。

伝播スケジュール

管理者は、DBA_QUEUE_SCHEDULES ビューをチェックして、伝播が特定のソース・キューおよび宛先の組合わせにスケジュールされているかをチェックできます。伝播がスケ

ジュールされていると、メッセージの伝播に使用されるジョブの jobno を、sys.aq$_schedules 表から判断できます。jobno を使用すると、DBA_JOBS ビューに問い合せて、最後に伝播がソース・キューと宛先の組み合わせに対して行われたときを判断します。DBA_JOBS ビューでは、次回の伝播スケジュールおよびジョブに「broken」のマークが設定されているかどうかも示されます。ジョブに「broken」のマークが設定されている場合は、$ORACLE_HOME/log ディレクトリの job_queue プロセスに作成されたトレース・ファイルのエラーをチェックしてください。

データベース・リンク

伝播が失敗するには、いくつかのポイントがあります。

- 宛先へのネットワーク接続が可能であるかについて、宛先が到達可能であるかどうかを判断するとします。単純な分散問合せを実行する、または同じ接続文字列を持つ接続記述子を作成して、リモート・データベースへの接続を試行することで判断できます。
- 伝播を計画した（dbms_aqadm.schedule_propagation を使用して）ユーザー ID が宛先のデータベース・リンクにアクセスできることを確認する必要があります。
- データベース・リンクを介して宛先にログインするために使用されたユーザー ID が、AQ 使用の権限を付与されているかを検証します。
- aq$_agent 型のアドレス属性（ソース・キューのサブスクライバ・リストまたはエンキュー元の受信者リスト）に指定されているキュー名が、(a) 特定の宛先に存在しているか (b) エンキューが可能か、の両方についてチェックします。これらすべておよび伝播担当者が直面したその他のエラーは、$ORACLE_HOME/log ディレクトリの job_queue プロセスで作成されるトレース・ファイルにログが作成されます。

型のチェック

2 つのキューのペイロードの型が等しくない場合、AQ では、あるキューから別のキューにメッセージを伝播しません。管理者は、DBMS_AQADM.VERIFY_QUEUE_TYPES プロシージャを実行して、ソースおよび宛先のペイロードの型が合致するか検証できます。型のチェックの結果は、sys.aq$_message_types 表に格納されます。この表には、ソース・キューの OID および宛先キューのアドレス (つまり、[schema.]queue_name[@destination]) を使用してアクセスできます。

動的統計ビュー

次に示すように、GV$ ビューと V$ ビューは、まったく同じです。

表 11–26　GV$AQ

列名	型
QID	NUMBER
WAITING	NUMBER
READY	NUMBER
EXPIRED	NUMBER
TOTAL_WAIT	NUMBER
AVERAGE_WAIT	NUMBER

表 11–27　V$AQ

列名	型
QID	NUMBER
WAITING	NUMBER
READY	NUMBER
EXPIRED	NUMBER
TOTAL_WAIT	NUMBER
AVERAGE_WAIT	NUMBER

列名	説明
QID	キューの ID。user_queues および dba_queues の qid と同じです。
WAITING	'WAITING' 状態にあるメッセージの数。
READY	'READY' 状態にあるメッセージの数。
EXPIRED	'EXPIRED' 状態にあるメッセージの数。
TOTAL_WAIT	'READY' 状態で、メッセージがキューに待機している秒数。
AVERAGE_WAIT	'READY' 状態で、メッセージがデキューを待機している平均秒数。

これら 2 つのビューの違いは、GV$ ビューではデータベース全体について異なる状態のメッセージ数についての情報が提供されるに対し、V$ ビューでは、特定のインスタンスについての情報が提供される点です。この方法では、各インスタンスはそれぞれの AQ 統計情報を

それぞれの SGA に保有し、他のインスタンスによって収集された統計については認識されません。そのため、GV$AQ ビューの問合せがインスタンスによって行われたときに、他のすべてのインスタンスは、その AQ 統計情報を問合せ発行したインスタンスに送ります。

注意： 特定のキューを特定のインスタンスと関連付ける必要がある場合は、アプリケーション・レベルで施行する必要があります。

デモの参照先

次のデモが、それぞれに記したディレクトリに配置されています。

`$ORACLE_HOME/demo/aqdemo00.sql` デモのメイン・ドライバ
`$ORACLE_HOME/demo/aqdemo01.sql` AQ 管理インタフェースを使った、キュー表およびキューの作成
`$ORACLE_HOME/demo/aqdemo02.sql` デモ・パッケージのロード
`$ORACLE_HOME/demo/aqdemo03.sql` イベント・ハンドラをジョブとしてジョブキューに送付
`$ORACLE_HOME/demo/aqdemo04.sql` エンキュー・メッセージ

互換性およびアップグレード

Oracle AQ 8.0.4 の操作インタフェースは、8.0.3 の Oracle AQ インタフェースと下位互換性があります。

AQ$_AGENT データ型に使用可能な新規フィールド

最新のリリースでは、アドレス・フィールドが、aq$_agent データ型として使用可能です。このため、メッセージ・プロパティの受信者リスト内および DBMS_AQADM.ADD_SUBSCRIBER 管理インタフェースなどのインタフェースが「エージェント」を引数としてとる箇所ならどこでも、このフィールドを指定できます。

拡張されたアドレス・フィールド

aq$_agent データ型のアドレス・フィールドは、1024 バイトに拡張されました。この拡張されたアドレス・フィールドを使用するには、次のステップを完了しなければなりません。

1. Export ユーティリティを使用して、既存のすべてのキューの内容を保存します。
2. CATNOQUEUE.SQL を実行して、既存のディクショナリおよびキュー表を削除します。

   SVRMGRL> @CATNOQUEUE.SQL
3. CATQUEUE.SQL を実行して、新しい型およびディクショナリ表を再定義します。

   SVRMGRL> @CATQUEUE.SQL
4. Import ユーティリティを使用して、エクスポートしたキューをインポートします。

注意： アプリケーションで、アドレス・フィールドを拡張する必要がない場合は、これらのステップを完了する必要はありません。その場合は、スクリプトの実行およびエクスポートとインポートの操作も必要ありません。

新しいディクショナリ表

- 8.0.4 (CAT8004.SQL) のアップグレード・スクリプトでは、追加のディクショナリ表、SYS.AQ$_MESSAGE_TYPES が作成されます。
- `SYSTEM.AQ$_SCHEDULES`
- `SYS.AQ$_QUEUE_STATISTICS`

12

PL/SQL 入出力

この章では、Oracle が提供するパッケージを使用して、PL/SQL によって外部のプロセスおよびセッション、ファイルと通信する方法を説明します。

次のパッケージがあります。

- DBMS_PIPE。セッション間で非同期に情報の送受信を行います。
- DBMS_OUTPUT。ある PL/SQL プログラムから、同一セッションの他の PL/SQL プログラムや、SQL*Plus を実行しているディスプレイ・ウィンドウへメッセージを送信します。
- UTL_FILE。PL/SQL プログラムでディスク・ファイルから情報を読み込んだり、情報をファイルに書き込んだりできます。

データベース・パイプ

DBMS_PIPE パッケージによって、同一インスタンスの複数のセッションが相互に通信できます。Oracle パイプは、UNIX で使用されているパイプと似ていますが、オペレーティング・システムのパイプ・メカニズムを使用して実装されているのではありません。Oracle パイプ経由で送信される情報は、システム・グローバル領域（SGA）にバッファリングされます。インスタンスがシャットダウンされると、パイプ内のすべての情報が失われます。

セキュリティの要件に応じて、パブリック・パイプか、プライベート・パイプかを選択できます。

警告：　パイプはトランザクションとは独立したものです。トランザクション制御に影響を与える可能性がある場合には、十分に注意してパイプを使用してください。

サマリー

表 12-1 に、DBMS_PIPE パッケージ内でコールできるプロシージャの概要を示します。

表 12-1　DBMS_PIPE パッケージの関数およびプロシージャ

関数 / プロシージャ	説明	参照ページ
CREATE_PIPE	パイプを明示的に作成する（プライベート・パイプに必要）	12-4 ページ
PACK_MESSAGE	ローカル・バッファにメッセージを作成する	12-6 ページ
SEND_MESSAGE	名前付きパイプにメッセージを送信する。名前付きパイプが存在しない場合には、パブリック・パイプが暗黙的に作成されます。	12-7 ページ
RECEIVE_MESSAGE	名前付きパイプからローカル・バッファへメッセージをコピーする	12-8 ページ
NEXT_ITEM_TYPE	バッファ内の次の項目のデータ型を戻す	12-10 ページ
UNPACK_MESSAGE	バッファ内の次の項目へアクセスする	12-10 ページ
REMOVE_PIPE	名前付きパイプを削除する	12-11 ページ
PURGE	名前付きパイプの内容を削除する	12-11 ページ
RESET_BUFFER	ローカル・バッファの内容を削除する	12-12 ページ
UNIQUE_ SESSION_NAME	一意のセッション名を戻す	12-12 ページ

DBMS_PIPE パッケージを作成する

DBMS_PIPE パッケージを作成するには、ユーザー SYS として接続し、DBMSPIPE.SQL スクリプト と PRVTPIPE.PLB スクリプトを実行します。これらのスクリプトは、CATPROC.SQL スクリプトによって自動的に実行されます。このパッケージを実行するユーザーへの必要な権限の付与の詳細は、10-36 ページの 「プロシージャを実行するために必要な権限」を参照してください。

パブリック・パイプ

パブリック・パイプは、暗黙的または明示的に作成できます。暗黙的なパブリック・パイプは、最初の参照時に自動的に作成され、データがなくなると消去されます。パイプ記述子は SGA に格納されるので、空のパイプがキャッシュから消去されるまで、領域使用オーバーヘッドが若干存在します。

明示的なパブリック・パイプは、PRIVATE フラグを FALSE に設定して CREATE_PIPE 関数をコールすると作成できます。明示的に作成されたパイプの割当て解除は、REMOVE_PIPE 関数をコールします。

パブリック・パイプの定義域は、パブリック・パイプが明示的あるいは暗黙的に作成されたスキーマです。

書込みおよび読取り

各パブリック・パイプは非同期に動作します。DBMS_PIPE パッケージに対する EXECUTE 許可を持っている限り、任意の数のスキーマ・ユーザーが、パブリック・パイプに書き込むことができ、パブリック・パイプの名前を知ることができます。

適切な権限と知識を持つスキーマ・ユーザーであれば、パブリック・パイプから情報を読み取ることができます。ただし、あるユーザーがバッファ内の情報をいったん読み取ると、その情報はバッファから消去され、同じパイプの他のユーザーは読み取れません。

送信セッションは、PACK_MESSAGE プロシージャへ 1 つまたは複数のコールを使用して、メッセージを作成します。このプロシージャは、そのメッセージをセッションのローカル・メッセージ・バッファに追加します。このバッファ内の情報は、メッセージの送信に使用するパイプ名を指定する SEND_MESSAGE プロシージャをコールすることによって送信されます。SEND_MESSAGE がコールされると、ローカル・バッファにスタックされているすべてのメッセージが送信されます。

メッセージを受信する必要のあるプロセスは、メッセージの受信元となるパイプの名前を指定して RECEIVE_MESSAGE プロシージャをコールします。次に、プロセスは UNPACK_MESSAGE プロシージャをコールして、メッセージ内の個々の項目にアクセスします。

プライベート・パイプ

プライベート・パイプは、CREATE_PIPE 関数をコールして明示的に作成しなければなりません。プライベート・パイプは一度作成されると、REMOVE_PIPE 関数をコールして明示的

に割当て解除するまで、共有メモリー内に存在し続けます。また、プライベート・パイプは、データベース・インスタンスのシャットダウン時にも割当て解除されます。

メモリー内に暗黙のパイプが存在し、作成しようとしているプライベート・パイプと同じ名前を持っている場合、プライベート・パイプは作成できません。このような場合には、CREATE_PIPE からエラーが戻されます。

プライベート・パイプへのアクセスは、次のものに限定されます。

- パイプの作成者と同じユーザー ID で動作するセッション
- パイプの作成者と同じユーザー ID 権限定義域内で実行されているストアド・サブプログラム
- `SYSDBA` または `INTERNAL` として接続されているユーザー

他の任意のユーザーがパイプでメッセージを送信または受信しようとしたり、パイプを削除しようとすると、即時エラーが発生します。別のユーザーが同じ名前を持つパイプを作成しようとしても、エラーが発生します。

パブリック・パイプを使用する場合のように、SEND_MESSAGE をコールする前に PACK_MESSAGE へのコールを使用して、まずメッセージを作成しなければなりません。同様に、UNPACK_MESSAGE をコールしてメッセージ内の項目にアクセスする前に、RECEIVE_MESSAGE をコールしてメッセージを検索しなければなりません。

エラー

DBMS_PIPE パッケージのルーチンは、次のエラーを戻すことがあります。

```
ORA-23321: Pipename may not be null
ORA-23322: Insufficient privilege to access pipe
```

ORA-23321 は、パイプ名をパラメータに持つ CREATE_PIPE またはサブプログラムから戻されます。ORA-23322 は、パラメータ・リスト内でプライベート・パイプを参照する任意のサブプログラムから戻されます。

CREATE_PIPE

CREATE_PIPE をコールして、パブリック・パイプまたはプライベート・パイプを明示的に作成します。PRIVATE フラグが TRUE の場合は、パイプ作成者はプライベート・パイプ所有者として指定されます。明示的に作成されたパイプを削除するには、REMOVE_PIPE をコールするか、または、インスタンスをシャットダウンしてください。

警告：　パイプ名の先頭は "ORA$" にしないでください。"ORA$" で始まる名前は、オラクル社で使用するために予約されています。

構文

CREATE_PIPE 関数のパラメータを表 12-2 に、この関数の戻り値とその意味を表 12-3 に示します。この関数の構文は、次のとおりです。

```
DBMS_PIPE.CREATE_PIPE(pipename    IN VARCHAR2,
                      maxpipesize IN INTEGER DEFAULT 8192,
                      private     IN BOOLEAN DEFAULT TRUE)
RETURN INTEGER;
```

表 12-2　DBMS_PIPE.CREATE_PIPE 関数のパラメータ

パラメータ	説明
pipename	作成するパイプの名前を指定します。SEND_MESSAGE および RECEIVE_MESSAGE をコールするときに、この名前を使用する必要があります。この名前は、すべてのインスタンス間で一意でなければなりません。
maxpipesize	パイプの最大サイズ（単位はバイト）を指定します。パイプのメッセージすべての合計サイズがこの数を超えることはできません。この最大値を超えるメッセージは無効です 。デフォルトの MAXPIPESIZE は 8192 バイトです。 パイプの MAXPIPESIZE は、パイプ特性の一部となるため、パイプが存続する限り有効です。SEND_MESSAGE のコール元でこれより大きなパイプ・サイズが必要なときは、MAXPIPESIZE の値を大きくします。コール元の値の方が小さいときは、MAXPIPESIZE の既存値を使用します。
private	デフォルト値 TRUE を使用して、プライベート・パイプを作成します。パブリック・パイプは、SEND_MESSAGE のコール時に暗黙的に作成できます。

表 12-3　DBMS_PIPE.CREATE_PIPE 関数戻り値

戻り値またはエラー	説明
0	パイプが正常に作成されたことを示します。 すでに存在するパイプを作成しようとした場合、ユーザーにそのパイプの使用権限があると、ステータス 0 が戻され、正常な実行を示します。また、そのパイプ内の既存データはすべて元のままです。 SYSDBA/SYSOPER としてログインしているユーザーが、パイプを再作成しようとすると、ステータス 0 が戻りますが、パイプの所有権は変わりません。

表 12-3　DBMS_PIPE.CREATE_PIPE 関数戻り値

戻り値またはエラー	説明
ORA-23322	命名重複が原因で失敗したことを示します。 同じ名前のパイプがすでに存在し、かつ別のユーザーが作成したパイプの場合は、エラー ORA-23322 が通知され、命名の誤りを示します。

PACK_MESSAGE プロシージャ

メッセージを送信するには、まず PACK_MESSAGE を 1 回または複数回コールして、ローカル・メッセージ・バッファにメッセージを作成します。次に、SEND_MESSAGE をコールして、名前付きパイプのローカル・バッファ内のメッセージを送信します。

PACK_MESSAGE プロシージャがオーバーロードされて、VARCHAR2 型、NUMBER 型、DATE 型の項目を受け入れます。データ・バイトの他に、バッファ内の各項目は、項目タイプを示すのに 1 バイト、項目の長さを格納するのに 2 バイトを必要とします。メッセージを終了するためにさらに 1 バイトが必要となります。メッセージ・バッファが 4096 バイトを超えると、ORA-6558 例外が発生します。

SEND_MESSAGE をコールしてこのメッセージを送信する場合は、メッセージを送信するパイプの名前を指定しなければなりません。このパイプがすでに存在する場合は、このパイプにアクセスするための十分な権限を持っていなければなりません。そのパイプが存在しない場合は、自動的に作成されます。

警告：　パイプ名の先頭は "ORA$" にしないでください。"ORA$" で始まる名前は、オラクル社で使用するために予約されています。

構文

PACK_MESSAGE プロシージャの構文は、次のとおりです。PACK_MESSAGE プロシージャ自身がオーバーロードされて、VARCHAR2 型、NCHAR 型、NUMBER 型、DATE 型の項目を受け入れることに注意してください。RAW 項目と ROWID 項目をパックするためのプロシージャがさらに 2 つあります。

```
DBMS_PIPE.PACK_MESSAGE       (item  IN  VARCHAR2);
DBMS_PIPE.PACK_MESSAGE       (item  IN  NCHAR);
DBMS_PIPE.PACK_MESSAGE       (item  IN  NUMBER);
DBMS_PIPE.PACK_MESSAGE       (item  IN  DATE);
DBMS_PIPE.PACK_MESSAGE_RAW   (item  IN  RAW);
DBMS_PIPE.PACK_MESSAGE_ROWID (item  IN  ROWID);
```

SEND_MESSAGE

SEND_MESSAGE 関数のパラメータを表 12-4 に、この関数の戻り値とその意味を表 12-5 に示します。この関数の構文は、次のとおりです。

```
DBMS_PIPE.SEND_MESSAGE(pipename     IN VARCHAR2,
                       timeout      IN INTEGER DEFAULT MAXWAIT
                       maxpipesize  IN INTEGER DEFAULT 8192)
RETURN INTEGER;
```

表 12-4　DBMS_PIPE.SEND_MESSAGE 関数のパラメータ

パラメータ	説明
pipename	メッセージを格納するパイプの名前を指定します。明示的なパイプを使用している場合は、この名前は CREATE_PIPE のコール時に指定した名前になります。
timeout	タイムアウト間隔（単位は秒）を指定します。これはパイプにメッセージを置こうとする間待機する時間です。戻り値については後述します。デフォルト値は定数 (MAXWAIT) であり、86400000(1000 日) として設定されています。
maxpipesize	パイプの最大サイズ（単位はバイト）を指定します。パイプのメッセージすべての合計サイズがこの数を超えることはできません。この最大値を超えるメッセージは無効です 。デフォルトの MAXPIPESIZE は 8192 バイトです。 パイプの MAXPIPESIZE は、パイプ特性の一部となるため、パイプが存続する限り有効です。SEND_MESSAGE のコール元でこれより大きなパイプ・サイズが必要なときは、MAXPIPESIZE の値を大きくします。コール元の値の方が小さいときは、MAXPIPESIZE の既存値を使用します。SEND_MESSAGE プロシージャ内に MAXPIPESIZE を指定すると、パイプをオープンするためだけのコールは必要なくなります。明示的にパイプを作成した場合、MAXPIPESIZE のオプション・パラメータを使用して、パイプ作成時のサイズ指定を変更できます。

表 12-5　DBMS_PIPE.SEND_MESSAGE 関数戻り値

戻り値またはエラー	説明
0	パイプが正常に作成されたことを示します。 すでに存在するパイプを作成しようとした場合、ユーザーにそのパイプの使用権限があると、ステータス 0 が戻され、正常な実行を示します。また、そのパイプ内の既存データはすべて元のままです。 SYSDBA/SYSOPER としてログインしているユーザーが、パイプを再作成しようとすると、ステータス 0 が戻りますが、パイプの所有権は変わりません。
1	パイプがタイムアウトされたことを示します。パイプのロックを取得できない、またはパイプが満杯のままで使用できないため、このプロシージャはタイムアウトする可能性があります。暗黙的に作成されて、空になっていると、そのパイプは削除されます。
3	割込みが発生したことを示します。暗黙的に作成されて、空になっていると、そのパイプは削除されます。
ORA-23322	パイプに書込みするための十分な権限を持っていないことを示します。 同じ名前のパイプがすでに存在し、かつ別のユーザーが作成したパイプの場合は、エラー ORA-23322 が通知され、命名の誤りを示します。

RECEIVE_MESSAGE

パイプからメッセージを受信するには、まず RECEIVE_MESSAGE をコールして、メッセージをローカル・メッセージ・バッファにコピーします。メッセージを受信すると、そのメッセージはパイプから削除されます。つまり、メッセージは 1 回しか受信できません。暗黙的に作成されたパイプの場合、最後のレコードがパイプから削除されてから、パイプが削除されます。

RECEIVE_MESSAGE のコール時に指定したパイプが存在していないと、Oracle はそのパイプを暗黙的に作成して、メッセージの受信を待ちます。指定したタイムアウト間隔以内にメッセージが到着しないと、そのコールは復帰して、パイプは削除されます。

メッセージを受信したら、UNPACK_MESSAGE を 1 回または複数回コールして、メッセージ内の各項目にアクセスしなければなりません。UNPACK_MESSAGE プロシージャはオーバーロードされ、DATE 型、NUMBER 型、VARCHAR2 型の項目をアンパックします。RAW および ROWID 項目をアンパックするには、2 つの追加プロシージャがあります。アンパックしようとするデータの型がわからない場合には、NEXT_ITEM_TYPE をコールして、バッファの中の次の項目の型を判別することができます。

構文

RECEIVE_MESSAGE 関数のパラメータの説明を表 12-6 に、この関数の戻り値とその意味を表 12-7 に示します。この関数の構文は、次のとおりです。

```
DBMS_PIPE.RECEIVE_MESSAGE(pipename     IN VARCHAR2,
                          timeout      IN INTEGER
                                       DEFAULT maxwait)
RETURN INTEGER;
```

表 12-6　DBMS_PIPE.RECEIVE_MESSAGE 関数のパラメータ

パラメータ	説明
pipename	メッセージを受け取るパイプの名前を指定します。ORA$ で始まる名前はオラクル社で予約されています。
timeout	タイムアウト間隔（単位は秒）を指定します。これはパイプでメッセージを受け取るために 待機する時間です。デフォルト値は定数 (MAXWAIT) であり、86400000(1000 日) として設定されています。タイムアウトを 0 に設定することによって、中断することなく読み込むことができます。

表 12-7　DBMS_PIPE.RECEIVE_MESSAGE 関数戻り値

戻り値またはエラー	説明
0	メッセージが正しく受信されたことを示します。
1	パイプがタイムアウトされたことを示します。暗黙的に作成されて、空になっていると、そのパイプは削除されます。
2	パイプ内のレコードが、そのバッファに入れるには大きすぎることを示します（これは発生しません)。
3	割込みが発生したことを示します。
ORA-23322	パイプから読み取るための十分な権限をユーザーが持っていないことを示します。

NEXT_ITEM_TYPE

RECEIVE_MESSAGE をコールしてローカル・バッファにパイプ情報を配置した後に、NEXT_ITEM_TYPE をコールしてローカル・メッセージ・バッファの次の項目のデータ型を判別できます。NEXT_ITEM_TYPE が 0 を戻した場合は、ローカル・バッファは空です。

構文

表 12-8 に、NEXT_ITEM_TYPE 関数の戻り値およびその意味を示します。この関数の構文は、次のとおりです。

```
DBMS_PIPE.NEXT_ITEM_TYPE RETURN INTEGER;
```

表 12-8　DBMS_PIPE.NEXT_ITEM_TYPE 関数戻り値

戻り値	説明
0	項目がないことを示します。
6	NUMBER
9	VARCHAR2
12	DATE

UNPACK_MESSAGE プロシージャ

RECEIVE_MESSAGE をコールしてローカル・バッファにパイプ情報を配置した後に、UNPACK_MESSAGE をコールしてバッファから項目を取り出せます。

構文

UNPACK_MESSAGE プロシージャの構文は、次のとおりです。UNPACK_MESSAGE プロシージャがオーバーロードされて、VARCHAR2 型、NCHAR 型、NUMBER 型、DATE 型の項目を戻すことに注意してください。RAW 項目と ROWID 項目をアンパックするためのプロシージャが 2 つあります。

```
DBMS_PIPE.UNPACK_MESSAGE       (item  OUT VARCHAR2);
DBMS_PIPE.UNPACK_MESSAGE       (item  OUT NCHAR);
DBMS_PIPE.UNPACK_MESSAGE       (item  OUT NUMBER);
DBMS_PIPE.UNPACK_MESSAGE       (item  OUT DATE);
DBMS_PIPE.UNPACK_MESSAGE_RAW   (item  OUT RAW);
DBMS_PIPE.UNPACK_MESSAGE_ROWID (item  OUT ROWID);
```

メッセージ・バッファに項目がなくなった場合、または受け取った項目が要求されているデータ型と異なる場合、ORA-2000 例外が発生します。

REMOVE_PIPE

SEND_MESSAGE によって暗黙的に作成されたパイプは、空になると自動的に削除されます。

CREATE_PIPE によって明示的に作成されたパイプは、REMOVE_PIPE をコールするか、またはインスタンスをシャットダウンしないと削除されません。パイプ内の使われていないレコードはすべて、パイプを削除する前に削除されます。これは、暗黙的に作成されたパイプで PURGE をコールするのと似ています。

構文

REMOVE_PIPE 関数では、パラメータを 1 つだけ（削除したいパイプの名前）受け入れます。表 12-9 に、戻り値とその意味を示します。この関数の構文は、次のとおりです。

```
DBMS_PIPE.REMOVE_PIPE(pipename  IN  VARCHAR2)
RETURN INTEGER;
```

表 12-9　DBMS_PIPE.REMOVE_PIPE 関数戻り値

戻り値またはエラー	説明
0	パイプが正しく削除されたことを示します。 パイプが存在しない場合、または既存のパイプを削除しようとしているユーザーに削除権限がある場合、ステータス 0 が戻され、正常な実行を示します。また、パイプ内のデータはすべて削除されます。
ORA-23322	権限が不十分なために失敗したことを示します。 既存のパイプに対するアクセス権限がユーザーにない場合、ORA-23322 が通知され、権限不足を示します。

パイプを管理する

DBMS_PIPE パッケージには、その他にも便利なプロシージャと関数があります。

パイプの内容を削除する

PURGE をコールして、パイプの内容を空にしてください。暗黙的に作成された空のパイプは、最低使用頻度アルゴリズムに従って、共有グローバル領域から削除されます。したがって、PURGE をコールすると、暗黙的に作成されたパイプと関係付けられているメモリーが解放されます。

PURGE は RECEIVE_MESSAGE をコールするので、パイプからメッセージが削除されるときに、ローカル・バッファはそれらのメッセージで上書きされます。また、十分なアクセス権を持っていないパイプを削除しようとすると、ORA-23322 エラー（権限が不十分）を受け取ることがあります。

```
DBMS_PIPE.PURGE(pipename IN VARCHAR2);
```

メッセージ・バッファのリセット

RESET_BUFFER をコールして、PACK_MESSAGE および UNPACK_MESSAGE の位置設定標識を 0 にリセットしてください。すべてのパイプが 1 つのバッファを共有するので、新しいパイプを使用する前にバッファをリセットすると便利です。これによって、初めてメッセージをパイプに送信するときに、バッファ内に残っている期限切れのメッセージを送信してしまうことはなくなります。

構文

RESET_BUFFER プロシージャの構文は、次のとおりです。

```
DBMS_PIPE.RESET_BUFFER;
```

一意なセッション名の取得

UNIQUE_SESSION_NAME をコールして、現在データベースに接続されているすべてのセッションの間で一意な名前を受信してください。同一セッションからこの関数への複数のコールは、常に同じ値を戻します。戻り値は最大 30 バイトです。この関数を使用して、SEND_MESSAGE コールおよび RECEIVE_MESSAGE コールの PIPENAME パラメータを指定すると便利です。

```
DBMS_PIPE.UNIQUE_SESSION_NAME RETURN VARCHAR2;
```

例 1: デバッグ

次の例では、PL/SQL プログラムからコールして、デバッグ情報をパイプに格納できるプロシージャを示します。

```
CREATE OR REPLACE PROCEDURE debug (msg VARCHAR2) AS
    status NUMBER;
BEGIN
  dbms_pipe.pack_message(LENGTH(msg));
  dbms_pipe.pack_message(msg);
  status := dbms_pipe.send_message('plsql_debug');
  IF status != 0 THEN
    raise_application_error(-20099, 'Debug error');
  END IF;
END debug;
```

この例は、前述の PL/SQL の例で、PLSQL_DEBUG かパイプからメッセージを受信して、そのメッセージを表示する Pro*C コードを示しています。Pro*C セッションを別のウィンドウで実行すると、このセッションは、別のセッションで実行している PL/SQL プログラムからデバッグ・プロシージャへ送信されるメッセージを表示できます。

```
#include <stdio.h>
#include <string.h>
```

```
EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR username[20];
   int     status;
   int     msg_length;
   char    retval[2000];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;

void sql_error();

main()
{
/* ユーザー名の準備 */
   strcpy(username.arr, "SCOTT/TIGER");
   username.len = strlen(username.arr);

   EXEC SQL WHENEVER SQLERROR DO sql_error();
   EXEC SQL CONNECT :username;

   printf("connected¥n");

/* 無限ループを開始してパイプにあるメッセージを、
   検索および出力する */
   for (;;)
   {
      EXEC SQL EXECUTE
         DECLARE
            len INTEGER;
            typ INTEGER;
            sta INTEGER;
            chr VARCHAR2(2000);
         BEGIN
            chr := '';
            sta := dbms_pipe.receive_message('plsql_debug');
            IF sta = 0 THEN
               dbms_pipe.unpack_message(len);
               dbms_pipe.unpack_message(chr);
            END IF;
            :status := sta;
            :retval := chr;
            IF len IS NOT NULL THEN
               :msg_length := len;
            ELSE
               :msg_length := 2000;
            END IF;
```

```
         END;
      END-EXEC;
      if (status == 0)
         printf("¥n%.*s¥n", msg_length, retval);
      else
         printf("abnormal status, value is %d¥n", status);
   }
}


void sql_error()
{
   char msg[1024];
   int rlen, len;
   len = sizeof(msg);
   sqlglm(msg, &len, &rlen);
   printf("ORACLE ERROR¥n");
   printf("%.*s¥n", rlen, msg);
   exit(1);
}
```

例 2: システム・コマンドの実行

次の例では、PL/SQL ストアド・プロシージャ（または無名ブロック）から PL/SQL プロシージャをコールして、コマンドを監視している Pro*C プログラムに、パイプを介してコマンドを送信できる、PL/SQL および Pro*C のコードを示します。

Pro*C プログラムは、名前付きパイプにメッセージが到着するのを待ちながらスリープします。メッセージが到着すると、C プログラムはそれを処理し、system() コールによって UNIX コマンドを実行したり、埋込み SQL を使用して SQL コマンドを実行したりして、必要なアクションを実行します。

DAEMON.SQL は、PL/SQL パッケージ用のソース・コードです。このパッケージには、DBMS_PIPE パッケージを使用して Pro*C デーモンに対してメッセージを送受信するプロシージャが含まれています。完全なハンドシェークが使用されることに注意してください。このデーモンは、常にメッセージをパッケージに送り返します（STOP コマンドの場合は除きます）。つまり、これによって、PL/SQL プロシージャは Pro*C デーモンが動作していることを確認できるからです。

SQL*Plus または Oracle Enterprise Manager を使用して、アノニマス PL/SQL ブロックから DAEMON パッケージ・プロシージャをコールできます。たとえば、次のとおりです。

```
SVRMGR> variable rv number
SVRMGR> execute :rv := DAEMON.EXECUTE_SYSTEM('ls -la');
```

とすると、UNIX システムでは、Pro*C デーモンによってコマンド system("ls -la") が実行されます。

デーモンは最初に実行しなければならないことを覚えておいてください。したがって、デーモンをバックグラウンドで、あるいはデーモンをコールした SQL*Plus または Oracle Enterprise Manager セッションの近くにある別のウィンドウから実行するとよいでしょう。

DAEMON.SQL では、DBMS_OUTPUT パッケージを使用して、結果の表示も行います。この例を動作させるためには、このパッケージに対する実行権限を持っていなければなりません。

関連項目： 12-20 ページの 「ストアド・プロシージャとトリガーからの出力」

DAEMON.SQL

PL/SQL DAEMON パッケージのコードは次のとおりです。

```
CREATE OR REPLACE PACKAGE daemon AS
  FUNCTION execute_sql(command VARCHAR2,
                       timeout NUMBER DEFAULT 10)
    RETURN NUMBER;

  FUNCTION execute_system(command VARCHAR2,
                          timeout NUMBER DEFAULT 10)
    RETURN NUMBER;

  PROCEDURE stop(timeout NUMBER DEFAULT 10);
END daemon;
/
CREATE OR REPLACE PACKAGE BODY daemon AS

  FUNCTION execute_system(command VARCHAR2,
                          timeout NUMBER DEFAULT 10)
  RETURN NUMBER IS

    status       NUMBER;
    result       VARCHAR2(20);
    command_code NUMBER;
    pipe_name    VARCHAR2(30);
  BEGIN
    pipe_name := DBMS_PIPE.UNIQUE_SESSION_NAME;

    DBMS_PIPE.PACK_MESSAGE('SYSTEM');
    DBMS_PIPE.PACK_MESSAGE(pipe_name);
    DBMS_PIPE.PACK_MESSAGE(command);
    status := DBMS_PIPE.SEND_MESSAGE('daemon', timeout);
    IF status <> 0 THEN
      RAISE_APPLICATION_ERROR(-20010,
        'Execute_system: Error while sending.  Status = ' ||
         status);
    END IF;
```

```
  status := DBMS_PIPE.RECEIVE_MESSAGE(pipe_name, timeout);
  IF status <> 0 THEN
    RAISE_APPLICATION_ERROR(-20011,
      'Execute_system: Error while receiving.
       Status = ' || status);
  END IF;

  DBMS_PIPE.UNPACK_MESSAGE(result);
  IF result <> 'done' THEN
    RAISE_APPLICATION_ERROR(-20012,
      'Execute_system: Done not received.');
  END IF;

  DBMS_PIPE.UNPACK_MESSAGE(command_code);
  DBMS_OUTPUT.PUT_LINE('System command executed.  result = ' ||
                       command_code);
  RETURN command_code;
END execute_system;

FUNCTION execute_sql(command VARCHAR2,
                     timeout NUMBER DEFAULT 10)
RETURN NUMBER IS

  status       NUMBER;
  result       VARCHAR2(20);
  command_code NUMBER;
  pipe_name    VARCHAR2(30);

BEGIN
  pipe_name := DBMS_PIPE.UNIQUE_SESSION_NAME;

  DBMS_PIPE.PACK_MESSAGE('SQL');
  DBMS_PIPE.PACK_MESSAGE(pipe_name);
  DBMS_PIPE.PACK_MESSAGE(command);
  status := DBMS_PIPE.SEND_MESSAGE('daemon', timeout);
  IF status <> 0 THEN
    RAISE_APPLICATION_ERROR(-20020,
      'Execute_sql: Error while sending.  Status = ' || status);
  END IF;

  status := DBMS_PIPE.RECEIVE_MESSAGE(pipe_name, timeout);


  IF status <> 0 THEN
    RAISE_APPLICATION_ERROR(-20021,
      'execute_sql: Error while receiving.
```

```
        Status = ' || status);
    END IF;

    DBMS_PIPE.UNPACK_MESSAGE(result);
    IF result <> 'done' THEN
      RAISE_APPLICATION_ERROR(-20022,
        'execute_sql: done not received.');
    END IF;

    DBMS_PIPE.UNPACK_MESSAGE(command_code);
    DBMS_OUTPUT.PUT_LINE
        ('SQL command executed.  sqlcode = ' || command_code);
    RETURN command_code;
  END execute_sql;

  PROCEDURE stop(timeout NUMBER DEFAULT 10) IS
    status NUMBER;
  BEGIN
    DBMS_PIPE.PACK_MESSAGE('STOP');
    status := DBMS_PIPE.SEND_MESSAGE('daemon', timeout);
    IF status <> 0 THEN
      RAISE_APPLICATION_ERROR(-20030,
        'stop: error while sending.  status = ' || status);
    END IF;
  END stop;
END daemon;
```

daemon.pc

これは、Pro*C デーモン用のコードです。バージョン 1.5.x 以降の Pro*C プリコンパイラを使用して、これをプリコンパイルしなければなりません。埋込み PL/SQL コードを含む例では、USERID および SQLCHECK オプションを指定する必要もあります。たとえば、次のとおりです。

```
proc iname=daemon userid=scott/tiger sqlcheck=semantics
```

次に、通常の方法で C コンパイルして、リンクします。

```
#include <stdio.h>
#include <string.h>

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
  char *uid = "scott/tiger";
  int status;
  VARCHAR command[20];
```

```
  VARCHAR value[2000];
  VARCHAR return_name[30];
EXEC SQL END DECLARE SECTION;

void
connect_error()
{
  char msg_buffer[512];
  int msg_length;
  int buffer_size = 512;

  EXEC SQL WHENEVER SQLERROR CONTINUE;
  sqlglm(msg_buffer, &buffer_size, &msg_length);
  printf("Daemon error while connecting:¥n");
  printf("%.*s¥n", msg_length, msg_buffer);
  printf("Daemon quitting.¥n");
  exit(1);
}

void
sql_error()
{
  char msg_buffer[512];
  int msg_length;
  int buffer_size = 512;

  EXEC SQL WHENEVER SQLERROR CONTINUE;
  sqlglm(msg_buffer, &buffer_size, &msg_length);
  printf("Daemon error while executing:¥n");
  printf("%.*s¥n", msg_length, msg_buffer);
  printf("Daemon continuing.¥n");
}
main()
{
  EXEC SQL WHENEVER SQLERROR DO connect_error();
  EXEC SQL CONNECT :uid;
  printf("Daemon connected.¥n");

  EXEC SQL WHENEVER SQLERROR DO sql_error();
  printf("Daemon waiting...¥n");
  while (1) {
    EXEC SQL EXECUTE
      BEGIN
        :status := DBMS_PIPE.RECEIVE_MESSAGE('daemon');
        IF :status = 0 THEN
          DBMS_PIPE.UNPACK_MESSAGE(:command);
        END IF;
```

```
  END;
END-EXEC;
if (status == 0)
{
  command.arr[command.len] = '¥0';
  if (!strcmp((char *) command.arr, "STOP"))
  {
    printf("Daemon exiting.¥n");
    break;
  }

  else if (!strcmp((char *) command.arr, "SYSTEM"))
  {
    EXEC SQL EXECUTE
      BEGIN
        DBMS_PIPE.UNPACK_MESSAGE(:return_name);
        DBMS_PIPE.UNPACK_MESSAGE(:value);
      END;
    END-EXEC;
    value.arr[value.len] = '¥0';
    printf("Will execute system command '%s'¥n", value.arr);

    status = system(value.arr);
    EXEC SQL EXECUTE
      BEGIN
        DBMS_PIPE.PACK_MESSAGE('done');
        DBMS_PIPE.PACK_MESSAGE(:status);
        :status := DBMS_PIPE.SEND_MESSAGE(:return_name);
      END;
    END-EXEC;

    if (status)
    {
      printf
       ("Daemon error while responding to system command.");
      printf(" status: %d¥n", status);
    }
  }
  else if (!strcmp((char *) command.arr, "SQL")) {
    EXEC SQL EXECUTE
      BEGIN
        DBMS_PIPE.UNPACK_MESSAGE(:return_name);
        DBMS_PIPE.UNPACK_MESSAGE(:value);
      END;
    END-EXEC;
    value.arr[value.len] = '¥0';
    printf("Will execute sql command '%s'¥n", value.arr);
```

```
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXEC SQL EXECUTE IMMEDIATE :value;
        status = sqlca.sqlcode;

        EXEC SQL WHENEVER SQLERROR DO sql_error();
        EXEC SQL EXECUTE
          BEGIN
            DBMS_PIPE.PACK_MESSAGE('done');
            DBMS_PIPE.PACK_MESSAGE(:status);
            :status := DBMS_PIPE.SEND_MESSAGE(:return_name);
          END;
        END-EXEC;

        if (status)
        {
          printf("Daemon error while responding to sql command.");
          printf(" status: %d¥n", status);
        }
      }
      else
      {
        printf
          ("Daemon error: invalid command '%s' received.¥n",
            command.arr);
      }
    }
    else
    {
      printf("Daemon error while waiting for signal.");
      printf(" status = %d¥n", status);
    }
  }
  EXEC SQL COMMIT WORK RELEASE;
  exit(0);
}
```

ストアド・プロシージャとトリガーからの出力

Oracle は、ストアド・プロシージャおよびパッケージ、トリガーからのメッセージを送信するために、DBMS_OUTPUT という名前のパブリック・パッケージを用意しています。このパッケージ内のプロシージャ PUT と PUT_LINE によって、別のトリガーまたはプロシージャ、パッケージが読み取れる情報をバッファ内に格納できます。

Oracle Enterprise Manager または SQL*Plus では、DBMS_OUTPUT プロシージャでバッファに入れられたメッセージの表示もできます。このためには、Oracle Enterprise Manager または SQL*Plus で SET SERVEROUT PUT ON コマンドを発行しなければなりません。

別個の PL/SQL プロシージャまたは無名ブロックでは、GET_LINE プロシージャをコールして、バッファに入っている情報を表示できます。GET_LINE をコールしなかったり、SQL*Plus または Oracle Enterprise Manager で 画面にメッセージを表示しないと、バッファに入っているメッセージは無視されます。DBMS_OUTPUT パッケージは、PL/SQL デバッグ情報の表示に特に便利です。

注意： DBMS_OUTPUT を使用して送信したメッセージは、送信元サブプログラムまたはトリガーが完了するまで、実際には送信されません。プロシージャの実行中に出力をフラッシュするメカニズムはありません。

サマリー

表 12-10 に、DBMS_OUTPUT パッケージからコール可能なプロシージャを示します。

表 12-10　DBMS_OUTPUT パッケージ関数とプロシージャ

関数 / プロシージャ	説明	参照ページ
ENABLE	メッセージの出力を可能にします。	12-22 ページ
DISABLE	メッセージの出力を不能にします。	12-22 ページ
PUT_LINE	バッファに 1 行入れます。	12-22 ページ
PUT	バッファに部分行を入れます。	12-22 ページ
NEW_LINE	PUT で作成された行を終了します。	12-36 ページ
GET_LINE	バッファから情報を 1 行取り出します。	12-23 ページ
GET_LINES	バッファから行の配列を取り出します。	12-23 ページ

DBMS_OUTPUT パッケージの作成

DBMS_OUTPUT パッケージを作成するには、ユーザー SYS として接続し、DBMSOTPT.SQL スクリプトと PRVTOTPT.PLB スクリプトを実行します。これらのスクリプトは、CATPROC.SQL スクリプトによって自動的に実行されます。

関連項目： このパッケージを実行するユーザーに必要な権限の付与の詳細は、10-36 ページの 「プロシージャを実行するために必要な権限」を参照してください。

エラー

DBMS_OUTPUT パッケージ・ルーチンがアプリケーション・エラー -20000 を発生させ、出力プロシージャが次のエラーを戻す場合があります。

```
ORU-10027: buffer overflow
ORU-10028: line length overflow
```

ENABLE プロシージャ

このプロシージャは、PUT および PUT_LINE、NEW_LINE、GET_LINE、GET_LINES のコールを使用可能にします。DBMS_OUTPUT パッケージが使用可能になっていない場合、これらの プロシージャへのコールは無視されます。Oracle Enterprise Manager または SQL*Plus の SERVEROUTPUT オプションを使用している場合、このプロシージャをコールする必要はありません。

バッファへの情報量（単位はバイト）を指定しなければなりません。項目は DBMS_OUTPUT パッケージに格納されます。バッファ・サイズを超えると、次のエラー・メッセージを受け取ります。

```
ORA-20000, ORU-10027: buffer overflow, limit of <buffer_limit> bytes.
```

ENABLE への複数のコールが可能です。ENABLE へのコールが複数ある場合、BUFFER_SIZE は指定した値の最大値となります。最大サイズは 1000000、最小サイズは 2000 です。

構文

ENABLE プロシージャの構文は、次のとおりです。

```
DBMS_OUTPUT.ENABLE(buffer_size IN INTEGER DEFAULT 2000);
```

DISABLE プロシージャ

DISABLE プロシージャは、PUT および PUT_LINE、NEW_LINE、GET_LINE、GET_LINES のコールを使用禁止にして、バッファに残っている情報を除去します。ENABLE の場合と同様に、Oracle Enterprise Manager または SQL*Plus の SERVEROUTPUT オプションを使用している場合、このプロシージャをコールする必要はありません。

構文

DISABLE プロシージャの構文は、次のとおりです。

```
DBMS_OUTPUT.DISABLE;
```

PUT および PUT_LINE プロシージャ

PUT_LINE をコールして行全体の情報をバッファに入れたり、PUT を複数回コールして行の情報を何回かに分けて作成したりできます。これらのプロシージャはどちらもオーバーロードされて、VARCHAR2、NUMBER、DATE 型の項目を受け入れ、バッファに入れます。

検索されると、すべての項目は VARCHAR2 に変換されます。型が NUMBER または DATE の項目を渡す場合、その項目を検索すると、デフォルト書式を使用して TO_CHAR で書式化さ

れます。異なる書式を使用する場合、VARCHAR2 として項目に渡してから明示的に書式化してください。

PUT_LINE をコールすると、指定した項目が行端マーカーによって自動的に追跡されます。PUT をコールして行を作成する場合は、NEW_LINE をコールして、独自の行端マーカーを追加しなければなりません。GET_LINE と GET_LINES は、改行文字で終わっていない行を戻しません。

行がバッファの制限を超えると、エラー・メッセージを受け取ります。

注意： PUT または PUT_LINE を使用して作成した出力は SGA にバッファリングされます。その出力は、バッファリングを行った PL/SQL プログラム・ユニットがコール側に戻るまでは取り出せません。そのため、たとえば Oracle Enterprise Manager または SQL*Plus では、PL/SQL プログラムが完了するまで、DBMS_OUTPUT メッセージは表示されません。このリリースでは、PL/SQL プログラム内で DBMS_OUTPUT バッファをフラッシュするメカニズムはありません。

構文

PUT および PUT_LINE プロシージャはオーバーロードされます。これらは NUMBER、VARCHAR2、DATE のいずれかの IN パラメータをとります。PUT、PUT_LINE、NEW_LINE プロシージャの構文は、次のとおりです。

```
DBMS_OUTPUT.PUT     (item IN NUMBER);
DBMS_OUTPUT.PUT     (item IN VARCHAR2);
DBMS_OUTPUT.PUT     (item IN DATE);
DBMS_OUTPUT.PUT_LINE(item IN NUMBER);
DBMS_OUTPUT.PUT_LINE(item IN VARCHAR2);
DBMS_OUTPUT.PUT_LINE(item IN DATE);
DBMS_OUTPUT.NEW_LINE;
```

GET_LINE および GET_LINES プロシージャ

バッファから 1 行だけ選択して取り出したり、複数の行を取り出したりすることができます。バッファに入っている情報を 1 行取り出すには、GET_LINE プロシージャをコールします。サーバーのコール数を減らすには、GET_LINES プロシージャをコールして、バッファから行の配列を取り出します。Oracle Enterprise Manager または SQL*Plus を使用している場合は、特殊な SET SERVEROUTPUT ON コマンドを使用して、この情報を自動的に表示できます。

GET_LINE または GET_LINES のコール後、次に PUT、PUT_LINE、NEW_LINE をコールする前に取り出されなかった行は、次のメッセージと混同されないように、すべて廃棄されます。

構文

表 12-11 に、GET_LINE プロシージャのパラメータを示します。このプロシージャの構文は、次のとおりです。

```
DBMS_OUTPUT.GET_LINE(line   OUT VARCHAR2,
                     status OUT INTEGER);
```

表 12-11　DBMS_OUTPUT.GET_LINE プロシージャのパラメータ

パラメータ	説明
line	バッファに入っている情報を 1 行戻します（最後の改行文字を除く）。このパラメータの最大長さは 255 バイトです。
status	コールが正常に完了すると、状態 0 が戻されます。バッファ内に行がなくなると、状態は 1 になります。

表 12-12 に、GET_LINES プロシージャのパラメータを示します。このプロシージャの構文は、次のとおりです。

```
DBMS_OUTPUT.GET_LINES(lines       OUT  CHARARR,
                      numlines IN OUT  INTEGER);
```

この CHARARR は VARCHAR2(255) の表であり、DBMS_OUTPUT パッケージ仕様部に型として定義されます。

表 12-12　DBMS_OUTPUT.GET_LINE プロシージャのパラメータ

パラメータ	説明
lines	バッファに入っている情報の行の配列を戻します。配列内の各行の最大長は 255 バイトです。
numlines	バッファから取り出す行の数を指定します。指定した行数を検索した後、プロシージャは 実際に検索した行数を戻します。この実際に検索した行数が要求した行数よりも少ない場合、バッファ内に行は存在しません。

DBMS_OUTPUT パッケージの使用例

DBMS_OUTPUT パッケージは、例 1 で示すように、ストアド・プロシージャとトリガーをデバッグするために使用します。また、このパッケージを使用することによって、例 2 で示すように、ユーザーはオブジェクトに関する情報を検索し、この出力を書式化できます。

例 1 次に、従業員表を問い合せて、指定した部門の給与総額を戻す関数の例を示します。この関数には、PUT_LINE プロシージャへのコールが複数含まれています。

```
CREATE FUNCTION dept_salary (dnum NUMBER) RETURN NUMBER IS
   CURSOR emp_cursor IS
      SELECT sal, comm FROM emp WHERE deptno = dnum;
   total_wages    NUMBER(11, 2) := 0;
   counter        NUMBER(10) := 1;
BEGIN
   FOR emp_record IN emp_cursor LOOP
      emp_record.comm := NVL(emp_record.comm, 0);
      total_wages := total_wages + emp_record.sal
         + emp_record.comm;
      DBMS_OUTPUT.PUT_LINE('Loop number = ' || counter ||
         '; Wages = '|| TO_CHAR(total_wages));  /* デバッグ行 */
      counter := counter + 1; /* デバッグ・カウンタの増加 */
   END LOOP;
   /* デバッグ行 */
   DBMS_OUTPUT.PUT_LINE('Total wages = ' ||
     TO_CHAR(total_wages));
   RETURN total_wages;
END dept_salary;
```

EMP 表に次の行が含まれているとします。

```
EMPNO          SAL     COMM     DEPT
-----        ------- -------- -------
1002           1500      500      20
1203           1000               30
1289           1000               10
1347           1000      250      20
```

Oracle Enterprise Manager の SQL Worksheet の入力ペインで、次の文を実行するとします。

```
SET SERVEROUTPUT ON
VARIABLE salary NUMBER;
EXECUTE :salary := dept_salary(20);
```

これで、次の情報が出力ペインに表示されます。

```
Loop number = 1; Wages = 2000
Loop number = 2; Wages = 3250
Total wages = 3250

PL/SQL procedure successfully executed.
```

例 2 この例では、ユーザーが EXPLAIN PLAN コマンドを使用して文の実行計画に関する情報を検索し、その情報を PLAN_TABLE に格納する場合を想定します。ユーザーはこの文に対して文 ID を割り当てているとします。例の EXPLAIN_OUT プロシージャは、この表から

情報を検索して、SQL 文の処理経過を順序どおりに表現するネストした方法で出力を書式化します。

```
 /*****************************************************************/
/* EXPLAIN_OUT プロシージャを作成する。ユーザーは、STATEMENT_ID を */
/* プロシージャに渡して、文を一意に識別しなければならない。      */
/*****************************************************************/
CREATE OR REPLACE PROCEDURE explain_out
   (statement_id IN VARCHAR2) AS

   -- Retrieve information from PLAN_TABLE into cursor
   -- EXPLAIN_ROWS.
   CURSOR explain_rows IS
      SELECT level, id, position, operation, options,
         object_name
      FROM plan_table
      WHERE statement_id = explain_out.statement_id
      CONNECT BY PRIOR id = parent_id
         AND statement_id = explain_out.statement_id
      START WITH id = 0
       ORDER BY id;

BEGIN
   -- Loop through information retrieved from PLAN_TABLE
   FOR line IN explain_rows LOOP

      -- At start of output, include heading with estimated cost.
      IF line.id = 0 THEN
         DBMS_OUTPUT.PUT_LINE ('Plan for statement '
            || statement_id
            || ', estimated cost = ' || line.position);
      END IF;

      -- Output formatted information. LEVEL is used to
      -- determine indention level.
      DBMS_OUTPUT.PUT_LINE (lpad(' ',2*(line.level-1)) ||
         line.operation || ' ' || line.options || ' ' ||
         line.object_name);
   END LOOP;
END;
```

PL/SQL のファイル I/O

リリース 7.3 の Oracle Server では、PL/SQL にファイル入出力機能が追加されました。この機能は、Oracle が提供する UTL_FILE パッケージによって実現されています。

このファイル入出力機能は、標準オペレーティング・システムのストリーム・ファイル入出力（OPEN、GET、PUT、CLOSE）と同様ですが、いくつかの制限があります。たとえば、FOPEN 関数をコールすると、ファイル・ハンドルが戻されます。後続の GET_LINE または PUT のコールでこのファイル・ハンドルを使用して、ファイルへのストリーム入出力を実行できます。ファイルの入出力が終了した場合には、FCLOSE をコールして出力を完了し、そのファイルに関連するリソースを解放します。

サマリー

表 12-13 に、UTL_FILE パッケージ内でコールされるプロシージャの概要を示します。

表 12-13　UTL_FILE プロシージャ

関数 / プロシージャ	説明	参照ページ
FOPEN	入力または出力用ファイルをオープンします。ファイルが存在しない場合には、出力ファイルを作成します。	12-31 ページ
IS_OPEN	ファイル・ハンドルが、オープンされているファイルを参照しているかどうかを判別します。	12-32 ページ
FCLOSE	ファイルをクローズします。	12-33 ページ
FCLOSE_ALL	オープンしているファイル・ハンドルをすべてクローズします。	12-34 ページ
GET_LINE	オープンしているファイルから、テキストを 1 行読み込みます。	12-34 ページ
PUT	ファイルに 1 行を書き込みます。行の終了記号は付加されません。	12-35 ページ
PUT_LINE	ファイルに 1 行を書き込みます。OS 固有の行終了記号が付加されます。	12-37 ページ
PUTF	フォーマット付きの PUT プロシージャです。	12-37 ページ
NEW_LINE	1 個以上の OS 固有の行終了記号をファイルに書き込みます。	12-36 ページ
FFLUSH	ファイルにまだ出力されていない保留中のデータをすべて物理的に書き込みます。	12-39 ページ

セキュリティ

PL/SQL のファイル入出力機能は、PL/SQL のクライアント側とサーバー側の両方で使用できます。導入済みクライアントは、オペレーティング・システムによる通常のファイル許可検査を受けるため、セキュリティ制約を追加する必要はありません。しかし、導入済みサーバーは、特権モードで稼働することがあるため、このファイル入出力機能の範囲を制限するために、セキュリティの制約を追加する必要があります。

注意： UTL_FILE パッケージは、現在 Oracle Procedure Builder に付属しているクライアント側の TEXT_IO パッケージと似ています。導入済みサーバーを対象とした制約では、UTL_FILE と TEXT_IO の間で異なる API を使用することが必要です。PL/SQL のファイル入出力では、PL/SQL 例外を使用して、コール側にエラーを戻します。

サーバーのセキュリティ

PL/SQL のファイル入出力のサーバー・セキュリティは、アクセス可能なディレクトリの制限によって構成されます。アクセス可能なディレクトリは、インスタンス・パラメータの初期設定ファイル（INIT.ORA）で指定しなければなりません。

初期設定ファイル内では、次のように UTL_FILE_DIR パラメータを使用して、UTL_FILE 関数がアクセス可能なディレクトリを指定します。

```
UTL_FILE_DIR = <directory name>
```

たとえば、インスタンス用の初期設定ファイルに次の行が含まれている場合は、

```
UTL_FILE_DIR = /usr/jsmith/my_app
```

ディレクトリ /usr/jsmith/my_app は、FOPEN 関数からアクセス可能です。大文字と小文字を区別するオペレーティング・システムでは、/usr/jsmith/My_App という名前のディレクトリはアクセス可能にならないことに注意してください。

次のようなパラメータ指定は、

```
UTL_FILE_DIR = *
```

特別な意味を持ちます。この項目は、実際にディレクトリのアクセス検査をオフにして、UTL_FILE 関数がすべてのディレクトリにアクセスできるようにします。

警告 :

- **'*' オプションを使用する際には、細心の注意を払う必要があります。セキュリティ上の明白な理由により、オラクル社は、実際のシステムではこのオプションを使用しないことを推奨します。また、アクセス可能ディレクトリのリストに '.' （UNIX の現行ディレクトリ）を含めないようにしてください。**
- **シンボリック・リンクが使用可能なファイル・システム上でセキュリティを保証するには、PL/SQL ファイル入出力関数からアクセス可能なディレクトリへの書込み許可をユーザーに与えてはいけません。シンボリック・リンクと PL/SQL ファイル入出力を結びつけると、通常のオペレーティング・システムの許可検査を回避することができ、この結果、他の方法ではアクセスできないディレクトリにユーザーが読込み / 書込みアクセスできる状態になります。**

ファイルの所有権および保護

UNIX システムでは、FOPEN 関数を使用して作成したファイルは、インスタンスを実行するシャドウ・プロセスが所有者になります。通常の場合、この所有者は oracle です。FOPEN を使用して作成したファイルは、UTL_FILE ルーチンを使用すれば常に読込みおよび書込みができます。しかし、非特権ユーザーがこれらのファイルを PL/SQL 以外によって読み込むには、システム管理者にアクセス権を設定してもらう必要があります。

例（UNIX 固有）

パラメータ初期設定ファイルに次のものしか含まれていない場合には、

```
UTL_FILE_DIR=/appl/gl/log
UTL_FILE_DIR=/appl/gl/out
```

次のファイルの場所およびファイル名が有効になります。

```
FILE LOCATION            FILENAME
/appl/gl/log             L10324.log
/appl/gl/out             O10324.out
```

次のファイルの場所およびファイル名は無効になります。

```
FILE LOCATION            FILENAME
/appl/gl/log/backup      L10324.log          # subdirectory
/APPL/gl/log             L10324.log          # uppercase
/appl/gl/log             backup/L10324.log   #dir in name
/usr/tmp                 T10324.tmp       # not in INIT.ORA
```

警告：　ユーザー・レベルのファイル許可はありません。UTL_FILE_DIR パラメータによって指定したすべてのファイルの場所は、ファイル入出力プロシージャを使用するすべてのユーザーが、読込みおよび書込みのために使用できます。これによって、オペレーティング・システムのファイル許可を上書きできます。

宣言された型

UTL_FILE パッケージの指定では、PL/SQL 型の FILE_TYPE を宣言します。この宣言は次のようになります。

```
TYPE file_type IS RECORD (id BINARY_INTEGER);
```

FILE_TYPE の内容は、UTL_FILE パッケージの専用です。このパッケージのユーザーは、このレコードの構成要素を参照したり変更したりしないでください。

例外

UTL_FILE パッケージの指定では、7 つの例外を宣言しています。これらの例外は、エラー条件を示すために発生します。表 12-14 に例外を示します。

表 12-14　UTL_FILE パッケージの例外

例外名	説明
INVALID_PATH	ファイルの場所またはファイル名が無効
INVALID_MODE	FOPEN の open_mode パラメータが無効
INVALID_FILEHANDLE	ファイル・ハンドルが無効
INVALID_OPERATION	要求どおりにファイルをオープンできない、または操作できない
READ_ERROR	読込み操作中にオペレーティング・システムのエラーが発生
WRITE_ERROR	書込み操作中にオペレーティング・システムのエラーが発生
INTERNAL_ERROR	PL/SQL 内の未指定エラー

これらのパッケージ例外に加えて、UTL_FILE パッケージ内のプロシージャが、NO_DATA_FOUND や VALUE_ERROR などの事前定義済みの PL/SQL 例外を発生させることもあります。

関数とプロシージャ

このセクションの残りの部分では、UTL_FILE パッケージを構成する、個々の関数およびプロシージャについて解説します。

FOPEN

FOPEN は、入力または出力用にファイルをオープンします。ファイルの場所は、インスタンスの初期化パラメータ UTL_FILE_DIR で定義されているアクセス可能なディレクトリでなければなりません。ディレクトリのパスはすでに存在していなければなりません。FOPEN では作成されません。FOPEN は、そのファイルでの後続するすべての入出力操作に必要なファイル・ハンドルを戻します。

表 12-15 に、このプロシージャのパラメータを示します。このプロシージャの構文は次のとおりです。

構文

```
FUNCTION FOPEN(location  IN VARCHAR2,
               filename  IN VARCHAR2,
               open_mode IN VARCHAR2)
    RETURN UTL_FILE.FILE_TYPE;
```

表 12-15　FOPEN 関数のパラメータ

パラメータ	説明
location	ファイルをオープンするディレクトリまたは場所を指定する、オペレーティング・システム固有の文字列。
filename	ファイルの名前。拡張子（ファイルの型）を含み、ディレクトリ・パス情報は含めない （UNIX オペレーティング・システムでは、ファイル名を「/」で終了させることはできない）。
open_mode	ファイルのオープン方法を指定する文字列（大文字または小文字を使用できる）。サポートされている値およびその値を指定して使用できる UTL_FILE パッケージ・プロシージャは次の とおりである。 'r' テキストの読込み (GET_LINE) 'w' テキストの書込み (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH) 'a' テキストの追加 (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH)

注意 : 存在しないファイルを、OPEN_MODE に値「a」を使用してオープンすると、そのファイルは書込み (「w」) モードで作成されます。

戻り値

FOPEN は、そのファイルを操作する後続のすべてのプロシージャに渡す必要のあるファイル・ハンドルを戻します。ファイル・ハンドルの特定の内容は、UTL_FILE パッケージに専用なものであるため、UTL_FILE のユーザーは、個々の構成要素を参照したり変更したりしないでください。

注意 :

- ファイルの場所およびファイル名のパラメータは、FOPEN 関数に対して別々の文字列として指定するため、初期化ファイルで指定したアクセス可能なディレクトリのリストに対してファイルの場所をチェックできます。ファイルの場所と名前を合わせたものが、システムで有効なファイル名を構成して、ディレクトリが アクセス可能でなければなりません。アクセス可能なディレクトリのサブディレクトリはアクセス可能である必要はありません。初期設定ファイルに、完全なパス名を使用して指定しなければなりません。
- UNIX での C シェルの環境変数など、オペレーティング・システム固有のパラメータは、ファイルの場所またはファイル名のパラメータでは使用できません。

例外

FOPEN は、次の例外を発生させることがあります。

- `UTL_FILE.INVALID_PATH`
- `UTL_FILE.INVALID_MODE`
- `UTL_FILE.INVALID_OPERATION`

IS_OPEN

IS_OPEN は、ファイル・ハンドルがオープンされているファイルを識別しているかどうかをテストします。IS_OPEN は、ファイル・ハンドルが、オープンされ、まだクローズされていないファイルを示している場合にだけ報告されます。そのファイル・ハンドルを使用した場合にオペレーティング・システムのエラーが発生しないとは限りません。

表 12-16 に、この関数のパラメータを示します。この関数の構文は次のとおりです。

構文

```
FUNCTION IS_OPEN(file_handle  IN FILE_TYPE)
    RETURN BOOLEAN;
```

表 12-16　IS_OPEN 関数のパラメータ

パラメータ	説明
file_handle	FOPEN のコールで戻された有効なファイル・ハンドル。

戻り値

TRUE または FALSE

例外

IS_OPEN は例外を発生させません。

FCLOSE

FCLOSE は、ファイル・ハンドルで識別済みのオープンされているファイルをクローズします。FCLOSE の実行の際には、ファイルのクローズ時に、まだ書き込んでいないデータがバッファに残っていると、WRITE_ERROR 例外を受け取る場合があります。

表 12-17 に、この関数のパラメータを示します。この関数の構文は次のとおりです。

構文

```
PROCEDURE FCLOSE (file_handle IN OUT FILE_TYPE);
```

表 12-17　FCLOSE プロシージャのパラメータ

パラメータ	説明
file_handle	FOPEN のコールで戻された有効なファイル・ハンドル。

例外

FCLOSE は、次の例外を発生させることがあります。

- UTL_FILE.WRITE_ERROR
- UTL_FILE.INVALID_FILEHANDLE

FCLOSE_ALL

FCLOSE_ALL は、そのセッションのすべてのオープンされているファイル・ハンドルをクローズします。これは、PL/SQL プログラムの例外終了などの、非常時のクリーン・アップ・プロシージャとして使用できます。

注意： FCLOSE_ALL では、ユーザーが保持しているオープンされたファイル・ハンドルの状態は変更されません。つまり、FCLOSE_ALL をコールした後、ファイルはクローズされていても、ファイル・ハンドルの IS_OPEN テストは TRUE を戻します。FCLOSE_ALL の前にオープンされたファイルには、それ以降は読込み操作や書込み操作を行うことはできません。

構文

```
PROCEDURE FCLOSE_ALL;
```

例外

FCLOSE_ALL は、次の例外を発生させることがあります。

- UTL_FILE.WRITE_ERROR

GET_LINE

GET_LINE は、ファイル・ハンドルで識別済みのオープンされているファイルからテキストを 1 行読み込んで、出力バッファ・パラメータに格納します。テキストは、行の終了記号またはファイルの終わりまで読み込まれますが、行の終了記号は含まれません。

行がバッファに収まらない場合には、VALUE_ERROR 例外が発生します。「ファイルの終わり」に到達したために、テキストが読み込まれなかった場合には、NO_DATA_FOUND 例外が発生します。

行の終了記号の文字はバッファに読み込まれないため、空行を読み込むと、空の文字列が戻されます。

入力レコードの最大サイズは 1022 バイトです。

表 12-18 に、この関数のパラメータを示します。この関数の構文は次のとおりです。

構文

```
PROCEDURE GET_LINE(file_handle     IN  FILE_TYPE,
                   buffer          OUT VARCHAR2);
```

表 12-18　GET_LINE プロシージャのパラメータ

パラメータ	説明
file_handle	FOPEN のコールで戻された有効なファイル・ハンドル。ファイルは読込み（モード「r」）としてオープンしなければなりません。それ以外の場合には、 INVALID_OPERATION 例外が発生する。
buffer	ファイルから読み込んだ行を受け取るデータ・バッファ。

例外

GET_LINE は、次の例外を発生させることがあります。

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.READ_ERROR
- NO_DATA_FOUND
- VALUE_ERROR

PUT

PUT は、ファイル・ハンドルで識別済みのオープンされているファイルに、バッファ・パラメータ内に格納されたテキスト文字列を書き込みます。ファイルは書込みモードでオープンされていなければなりません。PUT では行の終了記号は付加されません。行を区切るには、NEW_LINE を使用するか、または PUT_LINE を使用して行の終了記号付きの完全な 1 行を書き込んでください。

表 12-19 に、この関数のパラメータを示します。この関数の構文は次のとおりです。

構文

```
PROCEDURE PUT(file_handle  IN FILE_TYPE,
              buffer       IN VARCHAR2);
```

表 12-19　PUT プロシージャのパラメータ

パラメータ	説明
file_handle	FOPEN のコールで戻された有効なファイル・ハンドル。
buffer	ファイルに書き込むテキストを含むバッファ。ファイルはモード「w」またはモード「a」を使用してオープンされていなければなりません。これ以外のモードを使用すると、INVALID_OPERATION 例外が発生します。

例外

PUT は、次の例外を発生させることがあります。

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.WRITE_ERROR

NEW_LINE

NEW_LINE は、入力ファイル・ハンドルで識別されるファイルに、1 つまたは複数の行終了記号を書き込みます。行の終了記号はプラットフォーム固有の文字または文字列であるため、このプロシージャは PUT とは異なります。

表 12-20 に、この関数のパラメータを示します。この関数の構文は次のとおりです。

構文

```
PROCEDURE NEW_LINE (file_handle  IN FILE_TYPE,
                    lines        IN NATURAL := 1);
```

表 12-20　NEW_LINE プロシージャのパラメータ

パラメータ	説明
file_handle	FOPEN のコールで戻された有効なファイル・ハンドル。
lines	ファイルに書き込む行終了記号の数。

例外

NEW_LINE は、次の例外を発生させることがあります。

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION

- UTL_FILE.WRITE_ERROR

PUT_LINE

PUT_LINE は、ファイル・ハンドルで識別済みのオープンしているファイルに、バッファ・パラメータに格納されたテキスト文字列を書き込みます。ファイルは書込みモードでオープンされていなければなりません。PUT_LINE は、プラットフォーム固有の行終了記号で行を区切ります。

出力レコードの最大サイズは 1023 バイトです。

表 12-21 に、この関数のパラメータを示します。この関数の構文は次のとおりです。

構文

```
PROCEDURE PUT_LINE(file_handle  IN FILE_TYPE,
                   buffer       IN VARCHAR2);
```

表 12-21　PUT_LINE プロシージャのパラメータ

パラメータ	説明
file_handle	FOPEN のコールで戻された有効なファイル・ハンドル。
buffer	ファイルに書き込む行を含むテキスト・バッファ。

例外

PUT_LINE は、次の例外を発生させることがあります。

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.WRITE_ERROR

PUTF

PUTF は、形式化 PUT プロシージャです。これは、制限付きの printf() のように動作します。フォーマット文字列には任意の文字列を指定できますが、文字シーケンス ’%s’ および ’¥n’ には次のような特別な意味があります。

%s　　これは、引数リスト内の次の引数の文字列値に置き換えられます（後述の「構文」の項を参照してください）。

¥n　　これは、適切なプラットフォーム固有の行終端記号に置き換えられます。

表 12-22 に、この関数のパラメータを示します。この関数の構文は次のとおりです。

構文

```
PROCEDURE PUTF(file_handle  IN FILE_TYPE,
               format       IN VARCHAR2,
               [arg1        IN VARCHAR2,
                . . .
                arg5 IN VARCHAR2]);
```

表 12-22　PUTF プロシージャのパラメータ

パラメータ	説明
file_handle	FOPEN のコールで戻された有効なファイル・ハンドル。
format	テキストやフォーマット文字「¥n」および「%s」を含むことができるフォーマット文字列。
arg1..arg5	1 個から 5 個までのオプションの引数文字列。引数文字列は、フォーマット文字列内の「%s」フォーマッタと、順番に置き換えられる。 引数よりも多くのフォーマッタがフォーマット・パラメータ文字列内にある場合には、引数のない「%s」は空文字列で置き換えられる。

例

次にその例を示します。

```
Hello, world!
I come from Zork with greetings for all earthlings.

my_world  varchar2(4) := 'Zork';
...
PUTF(my_handle, 'Hello, world!¥nI come from %s with %s.¥n',
                my_world,
                'greetings for all earthlings');
```

引数よりも多くの「%s」フォーマッタがフォーマット・パラメータ内にある場合には、対応する引数のない「%s」は空文字列で置き換えられます。

例外

PUTF は、次の例外を発生させることがあります。

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.WRITE_ERROR

FFLUSH

FFLUSH は、ファイル・ハンドルで識別されるファイルに、まだ実際には書き込まれていない保留中のすべてのデータを物理的に書き込みます。通常は、書き込むデータはバッファリングされます。FFLUSH プロシージャは、バッファリングされているデータを強制的にファイルに書き込みます。

フラッシュは、まだオープンされているファイルを読み込む必要がある場合に役に立ちます。たとえば、デバッグ・メッセージをファイルにフラッシュして、即時に読込み可能にできます。

表 12-23 に、この関数のパラメータを示します。この関数の構文は次のとおりです。

構文

```
PROCEDURE FFLUSH (file_handle IN FILE_TYPE);
```

表 12-23　FFLUSH プロシージャのパラメータ

パラメータ	説明
file_handle	FOPEN のコールで戻された有効なファイル・ハンドル。

例外

FFLUSH は、次の例外を発生させることがあります。

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.WRITE_ERROR<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">

13

データベース・トリガーの使用

この章では、データベースに格納され、表が変更されたときに暗黙的に実行（「起動」）されるプロシージャとなるデータベース・トリガーについて説明します。トピックは次のとおりです。

- トリガーを設計する
- トリガーを作成する
- トリガーがコンパイルされるとき
- トリガーをデバッグする
- トリガーを修正する
- トリガーを使用可能、使用禁止にする
- トリガーに関する情報をリストする
- トリガー・アプリケーションの例

注意： Trusted Oracle を使用している場合、データベース・トリガーの定義と使用方法の詳細は、Trusted Oracle のマニュアルを参照してください。

トリガーを設計する

データベース・トリガーの設計では、次のガイドラインに従ってください。

- ある操作を実行したときに、関係するアクションも実行する場合にトリガーを使用する。
- データベース・トリガーは、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文を発行するかにかかわらず、トリガーを実行する文によって起動されなければならない、集中グローバル操作にだけ使用する。
- トリガーには、Oracle にすでに組み込まれている機能を重複して定義することはできない。たとえば、データ整合性の保証のためにトリガーを定義してはいけません。宣言整合性制約を使用すれば簡単に実現できます。
- トリガーのサイズを制限する（60 行以下が適当です）。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、ほとんどのコードをストアド・プロシージャに組み込んで、トリガーからそのプロシージャをコールすることをお薦めします。
- **再帰的トリガーを作成しないように注意する。**たとえば、それ自身が EMP 表に対して UPDATE 文を発行する AFTER UPDATE 文トリガーを EMP 表に作成すると、トリガーはメモリーが不足するまで再帰的に起動します。

トリガーを作成する

CREATE TRIGGER コマンドを使用してトリガーを作成します。このコマンドは、SQL*Plus や Oracle Enterprise Manager などの対話形式のツールで使用できます。対話形式のツールを使用する場合は、最終行にスラッシュ（/）を 1 つだけ付けて、CREATE TRIGGER 文をアクティブにします。

次の文は、EMP 表に対するトリガーを作成します。

```
CREATE TRIGGER print_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON emp
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;
/
```

SQL 文を次のように入力します。

```
UPDATE emp SET sal = sal + 500.00 WHERE deptno = 10
```

これにより、トリガーは更新される各行ごとに 1 回起動し、新旧の給与と、その差額分を出力します。

PL/SQL ブロックにエラーがあると、CREATE 文（または CREATE OR REPLACE 文）は失敗します。

以降の項では、トリガーのパーツの指定方法を説明します。CREATE TRIGGER 文の実例の詳細は、13-20 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てください。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。

注意： 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。使用しているプラットフォーム固有の Oracle マニュアルを参照してください。

トリガーを命名する

トリガーの名前は、同一スキーマ内で一意でなければなりません。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が、間違えやすいため違う名前を付けることをお薦めします。

BEFORE 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るトリガー本体の実行時期を指定できます。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 行トリガーを使うと、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（論理的な読込みであり、物理的なものではない）、トリガーをかけた更新文のために再び読み込まなければなりません。AFTER 行トリガーでは、データ・ブロックを更新文とトリガーのために一度読み込むだけで済みます。

INSTEAD OF オプション

INSTEAD OF オプションは、CREATE TRIGGER 文において、BEFORE または AFTER オプションのかわりに使用します。INSTEAD OF トリガーを使えば、UPDATE および INSERT、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。これらのトリガーが INSTEAD OF トリガーと呼ばれるのは、他のタイプのトリガーと違って、トリガーを実行する文を実行するかわりに、Oracle によってトリガーが起動されるためです。トリガーによって、更新または、挿入、削除などの操作が基礎表に対して直接実行されます。

ビューに対して標準的な UPDATE および INSERT、DELETE 文を書き込むと、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。

デフォルトでは、INSTEAD OF トリガーは、各行ごとにアクティブになります (13-6 ページの「FOR EACH ROW オプション」を参照)。

変更不能のビュー

ビューの問合せに次のいずれかの構成体が含まれている場合、UPDATE または INSERT、DELETE の各文を使ってビューを変更することはできません。

- 集合演算子
- グループ関数
- GROUP BY 句または CONNECT BY 句、START WITH 句
- DISTINCT 演算子
- 結合（結合ビューのサブセットは更新できます）

擬似列または式が含まれたビューを更新するには、擬似列または式のいずれも参照しない UPDATE 文を使います。

INSTEAD OF トリガーの例

次の例では、MANAGER_INFO ビューに行を挿入する場合の INSTEAD OF トリガーを示します。

```
CREATE VIEW manager_info AS
    SELECT e.name, e.empno, d.dept_type, d.deptno, p.level,
           p.projno
        FROM   emp e, dept d, project p
        WHERE  e.empno =  d.mgr_no
        AND    d.deptno = p.resp_dept;

CREATE TRIGGER manager_info_insert
INSTEAD OF INSERT ON manager_info
REFERENCING NEW AS n                  -- new manager information

FOR EACH ROW
BEGIN
```

```
  IF NOT EXISTS SELECT * FROM emp
        WHERE emp.empno = :n.empno
  THEN
        INSERT INTO emp
            VALUES(:n.empno, :n.name);
  ELSE
   UPDATE emp SET emp.name = :n.name
         WHERE emp.empno = :n.empno;
    END IF;

 IF NOT EXISTS SELECT * FROM dept
     WHERE dept.deptno = :n.deptno
  THEN
       INSERT INTO dept
        VALUES(:n.deptno, :n.dept_type);
  ELSE
     UPDATE dept SET dept.dept_type = :n.dept_type
        WHERE dept.deptno = :n.deptno;
    END IF;

  IF NOT EXISTS SELECT * FROM project
     WHERE project.projno = :n.projno
  THEN
      INSERT INTO project
                          VALUES(:n.projno, :n.project_level);
  ELSE
        UPDATE project SET project.level = :n.level
            WHERE project.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が行われると、まず、MANAGER_INFO の導出元の実表に該当する行があるかどうかを調べるテストが行われます。その後、適宜、新規行が挿入されるか、既存の行が更新されます。同種のトリガーを使って、UPDATE および DELETE 実行時の適切な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 トリガーでは、OCI コールを介して、クライアント側でオブジェクト・ビューのインスタンスを変更できます。(『Oracle 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・キャッシュ上のオブジェクト・ビューによって具体化されたオブジェクトを変更し、持続記憶領域にそれをフラッシュ・バックする場合、オブジェクト・ビューが変更できなければ、INSTEAD OF トリガーを指定する必要があります。ただし、オブジェクトが読込み専用であれば、トリガーを定義して確保する必要はありません。

関連項目： INSTEAD OF トリガーの詳細は、8-5 ページの「オブジェクト・ビューの更新」を参照してください。

トリガー文

トリガー文では、次のものを指定します。

- トリガー本体を実行する SQL 文のタイプを指定します。

  トリガー文には、DELETE、INSERT、UPDATE などのオプションを複数指定できます。これらのオプションのうちの 1 つ、2 つ、またはすべてをトリガー文の仕様部に組み込めます。

- トリガーの処理対象となる表。

注意： トリガー文には表（ビューではありません）を 1 つだけ指定できます。

たとえば、13-2 ページの PRINT_SALARY_CHANGES トリガーは、EMP 表の DELETE または INSERT、UPDATE のいずれかの後で起動します。次のいずれかの文によって、前出の例で指定した PRINT_SALARY_CHANGES トリガーが実行されます。

```
DELETE FROM emp;
INSERT INTO emp VALUES ( . . . );
INSERT INTO emp SELECT . . . FROM . . . ;
UPDATE emp SET . . . ;
```

UPDATE の列のリスト

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組み込むことができます。列リストを含めると、指定した列が更新された場合に限り、UPDATE 文のトリガーが実行されます。列リストを指定しないと、関係する表が更新された場合に限り、トリガーが実行されます。INSERT や DELETE トリガー文に列リストを指定することはできません。

前出の PRINT_SALARY_CHANGES トリガーの例では、トリガー文に次のように列リストを指定しています。

```
. . . BEFORE DELETE OR INSERT OR UPDATE OF ename ON emp . . .
```

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まります。FOR EACH ROW を指定すると、トリガー文の処理対象となる表の各行に対してトリガーが 1 回起動します。FOR EACH ROW オプションを指定しないと、個々の該当する文に対してトリガーが 1 回だけ起動しますが、その文の処理対象となる各行に対して別々に起動す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。

```
CREATE TRIGGER log_salary_increase
AFTER UPDATE ON emp
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、SQL 文を発行します。

```
UPDATE emp SET sal = sal + 1000.0
    WHERE deptno = 20;
```

部門 20 に 5 人の従業員がいる場合、この文が発行されると、トリガーが 5 回起動します。これは、5 つの行が処理対象となるためです。

次のトリガーは、EMP 表の各 UPDATE に対して 1 回だけ起動します。

```
CREATE TRIGGER log_emp_update
AFTER UPDATE ON emp
BEGIN
    INSERT INTO emp_log (log_date, action)
        VALUES (SYSDATE, 'EMP COMMISSIONS CHANGED');
END;
```

関連項目： トリガーの起動の順序の詳細は、『Oracle8 Server 概要』を参照してください。

WHEN 句

行トリガー定義に、トリガーの実行条件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（文トリガーの定義に WHEN 句を指定することはできません）。このオプションによって、WHEN 句の式がトリガーの処理対象となる各行ごとに評価されます。式が TRUE と評価されると、その行トリガーの本体が実行されます。ただし、行の式が（つまり、不定または NULL のため）FALSE または NOT TRUE と評価された場合、その行のトリガー本体は実行されません。WHEN 句の評価は、トリガー SQL 文の実行に対する影響を持ちません（つまり、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 の新しい値がゼロまたは NULL、負の場合、トリガー本体は実行されません。より具体的な例では、ある列の値が他の列の値より小さいかどうか検査する場合です。

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 式でなければならず、副問合せを含むことはできません。WHEN 句では PL/SQL 式（ユーザー定義関数を含む）を使用できません。

トリガー本体

トリガー本体は PL/SQL を実行するブロックであり、SQL または PL/SQL 文を指定できます。これらの文は、トリガーを実行する文が発行され、トリガー実行条件で TRUE と評価された場合に実行されます。行トリガーのトリガー本体には、相関名、REFERENCEING オプション、条件述語の INSERTING、DELETING、UPDATING などの特殊な要素を含められます。これらの要素は、PL/SQL ブロックのコードにも含められます。

行トリガーの列値にアクセスする

行トリガーのトリガー本体では、PL/SQL コードと SQL 文は、トリガー文の処理対象となる現在の行に含まれる列の old 値と new 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 値用と new 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 文によって実行されるトリガーは、列の new 値にだけアクセスする意味がある。その行は INSERT によって作成されており、old の値が NULL のためです。
- UPDATE 文によって実行されるトリガーは、BEFORE と AFTER の両方の行トリガーの、列の old と new の両方の値にアクセスできる。
- DELETE 文によって実行されるトリガーは、列の old 値にだけアクセスできる。行を削除すれば行はなくなり、値は NULL となるためです。

元の列の値は、列名の前に OLD の修飾子を指定して参照でき、新しい列の値は列名の前に NEW を指定して参照できます。たとえば、トリガー文が EMP 表（列 SAL、COMM など）と関連する場合、次のようにトリガー本体に文を指定できます。

```
IF :new.sal > 10000 . . .
IF :new.sal < :old.sal . . .
```

old と new の値は、BEFORE と AFTER 行トリガーに指定できます。BEFORE 行トリガーでは列の NEW 値を割り当てられます。しかし、（トリガー文は AFTER 行トリガーを実行する前に有効となるため）AFTER 行トリガーに列の NEW 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、BEFORE 行トリガーによって割り当てられた変更と同じ更新文によって、AFTER 行トリガーが実行されます。

WHEN 句のブール式に相関名を指定することもできます。それらをトリガー本体で使用する場合、OLD と NEW 修飾子の前にコロン（:）を付けなければなりません。ただし、WHEN 句または REFERENCING オプションでは、コロンは必要ありません。

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」と命名される相関名や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こらないため、このオプションはほとんど使用されません。

たとえば、FIELD1（数字）と FIELD2（文字）の列を含む表 NEW があるとします。次の CREATE TRIGGER の列は、相関名を指定できる表 NEW に関するトリガーの例です。相関名と表名の重複を避けられます。

```
CREATE 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、トリガー本体に指定していることに注意してください。

条件述語

複数のタイプの 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」）、トリガー本体に INSERTING、DELETING、UPDATING 条件述語を指定し、トリガーを起動する文のタイプに応じて、コードの特定の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
```

トリガー本体のコードに、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、トリガーを実行する文が INSERT 文の場合に限り TRUE と評価されます。2 番目の条件は、トリガーを実行する文が UPDATE 文の場合に限り 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、指定した列を更新できます。次のようにトリガーが定義されているとします。

```
CREATE TRIGGER . . .
. . . UPDATE OF sal, comm ON emp . . .
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限り、実行されます。次の文は上記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 SET sal = sal + 100;
```

トリガー本体のエラー条件と例外

トリガー本体の実行中に、ユーザー定義または事前定義されているエラー条件や例外が発生すると、トリガー文と同様にトリガー本体の処理結果がロールバックされます（エラーが例外ハンドラによって特別に処理される場合を除きます）。このため 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、セキュリティ許可と整合性制約を施行するトリガーに共通して使用できます。

> **関連項目：** PL/SQL プログラム・ユニットでのエラー処理の詳細は、10-27 ページの「エラー処理」および 10-29 ページの「例外と例外処理ルーチンを宣言する」を参照してください。

トリガーとリモート例外の処理

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・リンクが使用不能であると、リモート例外処理を実施できません。たとえば、次のとおりです。

```
CREATE TRIGGER example
AFTER INSERT ON emp
FOR EACH ROW
BEGIN
  INSERT INTO emp@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しなければならないときにリモート・サイトが使用不能であると、Oracle はリモート・データベースにアクセスする文の妥当性を検査できず、コンパイルは失敗します。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み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 TRIGGER example
AFTER INSERT ON emp
FOR EACH ROW
BEGIN
   insert_row_proc;
END;


CREATE PROCEDURE insert_row_proc
```

```
BEGIN
  INSERT INTO emp@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に、妥当性チェック済み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。以降の項では、トリガーの制約を説明します。

トリガー本体で有効な SQL 文

トリガー本体には、DML SQL 文を含められます。また、SELECT 文を含められますが、SELECT...INTO... 文またはカーソル定義中の SELECT 文でなければいけません。

DDL 文はトリガー本体に含められません。また、トリガー内ではトランザクション制御文は実現されません。コマンドの ROLLBACK および COMMIT、SAVEPOINT は使用できません。

注意： トリガーによってコールされたプロシージャは、トリガー本体のコンテキスト内で実行するため、上記のトランザクション制御文を実行できません。

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・トリガーからリモート・プロシージャをコールするときには、特に注意する必要があります。トリガーの実行中にタイムスタンプまたは署名の不一致が見つかると、リモート・プロシージャは実行されず、トリガーが無効となるためです。

LONG と LONG RAW データ型

トリガー内の LONG データ型および LONG RAW データ型は、次の制限を受けます。

- トリガー内の SQL 文を使用して、LONG または LONG RAW データ型の列にデータを挿入できる。
- LONG または LONG RAW 列のデータを制約データ型（CHAR や VARCHAR2 など）に変換できる場合、トリガー内の SQL 文から LONG または LONG RAW 列を参照できる。なお、この場合の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ない。
- LONG または LONG RAW 列では、:NEW と :OLD は使用できない。

パッケージ変数の参照

UPDATE 文と DELETE 文が同時進行中の UPDATE との衝突を検出すると、Oracle はセーブポイントまでの透過的ロールバックを実行して、更新をリスタートします。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。文が再実行されるたびに、BEFORE STATEMENT 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ていた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とよいでしょう。

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を作成してはなりません。たとえば、グローバル変数の現在の値が、行トリガーが処理中の行の影響を受ける場合は、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てはいけません。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とよいでしょう。

トリガー本体内の文によって他のトリガーが起動される場合、それらのトリガーは「カスケード」しているといいます。Oracle では、最大 32 のトリガーを一度に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。トリガーを実行するたびに、カーソルはオープンされるからです。

トリガーの評価順序

トリガーは、インラインまたはプロシージャをコールすることにより、一連のオペレーションを実行できますが、同じ表上で同じタイプの複数のトリガーが使用できるので、アプリケーションのモジュール化インストレーションが可能となり、データベース管理が強化されます。

Oracle は、別のタイプのトリガーを実行する前に同じタイプのトリガーをすべて実行します。1 つの表に同じタイプの複数のトリガーを持っていると、Oracle 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。

> **関連項目：** トリガーの起動順序の詳細は、『Oracle8 Server 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 値と new 値を参照できます。old 値とは元の値で、new 値と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設定値です。

トリガーされた複数のアクションが特定の順序で実行されることを保証するには、これらのアクションをまとめて 1 つのトリガーにしなければなりません（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 のバージョンを使用している場合は、同タイプの複数のトリガーを含んでいるデータベースをオープンすることはできません。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。

変更表と制約表

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。制約表とは、トリガー中の SQL 文で直接、宣言参照整合性制約では間接的に読み込む必要のある表です。表は、処理中の文を発行したセッション内でだけ、変更表または制約表となります。

DELETE CASCADE の結果としてトリガーが起動しない場合には、表は文トリガーの変更表または制約表とは見なされません。

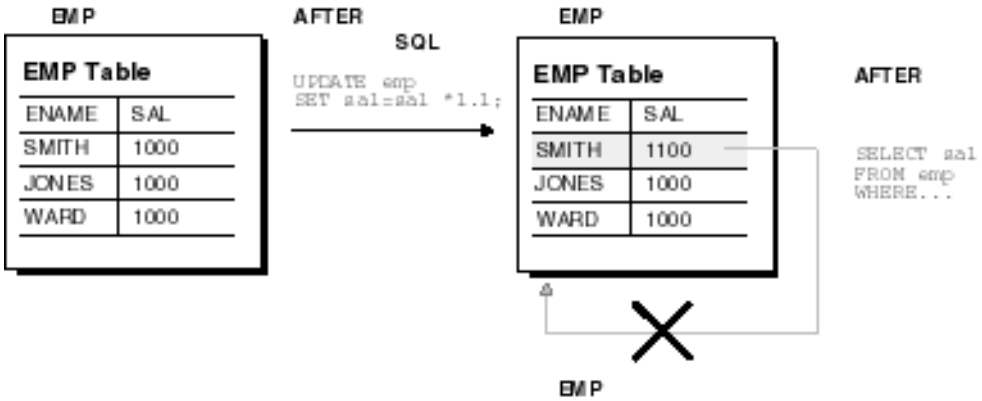
すべての行トリガー、または DELETE CASCADE の結果起動された文トリガーには、変更表および制約表に関する 2 つの重要な制限があります。この制限によって、トリガーは一貫性に欠けるデータを参照しません。

- 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（問い合せる）ことはできない。
- トリガーの文は、トリガーを実行する文の制約表の PRIMARY 列、FOREIGN 列、UNIQUE KEY 列を変更できない。

  この制限には例外があります。表への単一行 INSERT によって起動された BEFORE ROW トリガーと AFTER ROW トリガーは、その表を修正表または制約表と見なしません。INSERT INTO emp SELECT . . . などのように複数行にわたる INSERT 文は、結果的に 1 行だけ挿入されるだけの場合でも、1 行の挿入とは見なされないので注意してください。

図 13–1 には変更表に対する制約が示されています。

図 13-1　変更表



SQL 文は表の最初の行に対して実行され、次に AFTER ROW トリガーが実行されることに注意してください。ここで、AFTER ROW トリガー本体内の文が、元の表を問い合せます。ただし、EMP 表は変更表であるため、この問合せは Oracle では認められません。無理に実行すれば実行時エラーが発生し、トリガー本体の処理結果と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されます。

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
FOR EACH ROW
DECLARE
    n INTEGER;
BEGIN
    SELECT COUNT(*) INTO n FROM emp;
    DBMS_OUTPUT.PUT_LINE(' There are now ' || n ||
        ' employees.');
END;
```

次の SQL 文を実行します。

```
DELETE FROM emp WHERE empno = 7499;
```

これを発行すると、次のようなエラーが戻されます。

```
ORA-04091: 表 SCOTT.EMP は変更中なのでトリガー / 関数はこの表を読込みすることができません
```

最初の行が削除されたときに、表が変更表であるという理由で、トリガーが起動すると Oracle はこのエラーを戻します（EMPNO は主キーなので、文によって 1 行しか削除されません。しかし、Oracle は、削除文が単一行しか削除していないことを認識していません）。

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、起動すると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。

変更表や制約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 行トリガーと一時表からの値を使っ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検査されます。

> **関連項目：** トリガー間の相互作用と整合性制約の詳細は、『Oracle8 Server 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の宣言参照整合性制約は、現在サポートされていないので、制約表の制限は、リモート・ノードにアクセスするトリガーには適用されません。これらの制限は、ループバック・データベース・リンクで接続されている同一データベース内の表の間でも施行されません。ループバック・データベース・リンクでは、リンクを含むデータベースに戻る 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のに、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。

トリガー・ユーザーとは誰か

次のような文を発行するとします。

```
SELECT username FROM USER_USERS
```

トリガー内では、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。

トリガーを作成するために必要な権限

使用するスキーマにトリガーを作成するには、CREATE TRIGGER システム権限と次のいずれかの権限が必要です。

- トリガー文に指定した表の所有
- トリガー文中の表に対する ALTER 権限
- ALTER ANY TABLE システム権限

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。この権限があれば、任意のスキーマ内にトリガーを作成し、任意のユーザー表と関連付けることができます。

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）明示的に付与しなければなりません。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・プロシージャに類似しています。

関連項目： 10-36 ページの「プロシージャを実行するために必要な権限」

トリガーがコンパイルされるとき

トリガーは、:NEW および :OLD の機能を追加した PL/SQL 無名ブロックと類似していますが、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. 構文検査 : PL/SQL 構文がチェックされ、解析ツリーが生成される。
2. 意味上の検査 : タイプ検査と解析ツリーに対する追加処理
3. コード生成 : pcode が生成される。

これに対して、トリガーは、CREATE TRIGGER コマンドが発行されたときに完全にコンパイルされ、データ・ディクショナリに pcode が格納されます。そのため、トリガーを起動する際に、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、必ず DML 文は失敗します）。トリガーの作成時にあらゆるコンパイル・エラーが表示されるように、SQL*Plus または Enterprise Manager 内で SHOW ERRORS コマンドを使うこともできますし、あるいは USER_ERRORS ビューからエラーを SELECT することもできます。

依存性

コンパイル済み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関数やストアド・プロシージャといった、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、次にコール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、次のような文があるとします。

```
SELECT NAME, REFERENCED_OWNER, REFERENCED_NAME, REFERENCED_TYPE
    FROM ALL_DEPENDENCIES
    WHERE OWNER = 'SCOTT' and TYPE = 'TRIGGER';
```

SCOTT スキーマ内のトリガーの依存性が示されます。

トリガーを再コンパイルする

トリガーを手動で再コンパイルするには、ALTER TRIGGER コマンドを使用してください。次に、コマンドの例を示します。

```
ALTER TRIGGER print_salary_changes COMPILE;
```

これで、PRINT_SALARY_CHANGES トリガーが再コンパイルされます。

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、コンパイル済みのトリガーのリリース（Oracle 7.3 および Oracle8）では起動しません。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みのトリガーのリリースにアップグレードする場合、既存のトリガーはすべてコンパイルしなければなりません。アップグレード・スクリプト cat73xx.sql を実行すると、すべてのトリガーが無効にされ、最初の起動時に自動的に再コンパイルされます。（xx は、可変のマイナー・リリース番号を表します）。

Oracle 7.3 またはそれ以降のリリースを 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 cat73xxd.sql を実行しなければなりません。このスクリプトによって、格納済みのトリガーのリリースと格納されていないトリガーのリリースとの間の移植に関する問題が処理されます。

トリガーをデバッグする

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って、トリガーをデバッグできます。

関連項目： 10-33 ページの「デバッグ」

トリガーを修正する

ストアド・プロシージャと同様に、トリガーを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なければなりません （ALTER TRIGGER コマンドは、トリガーを再コンパイルしたり、使用可能にしたり、使用禁止にしたりするためだけに使用します）。

トリガーを置き換えるときには、CREATE TRIGGER 文に OR REPLACE オプションを指定しなければなりません。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を与えずに、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 コマンドを使用して削除でき、削除してから CREATE TRIGGER コマンドを再実行できます。

トリガーを削除するには、そのトリガーが使用するスキーマ内にあるか、または DROP ANY TRIGGER システム権限を持っていなければなりません。

トリガーを使用可能、使用禁止にする

トリガーは、次の 2 つのモードのいずれかにできます。

使用可能　トリガー文を発行し、トリガー制限が 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　トリガー文を発行し、トリガー制限が 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。

トリガーを使用禁止にする

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。

- オブジェクトが参照不能の場合
- 大規模なデータ・ロードを実行しなければならず、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。トリガーは、ALTER TRIGGER コマンドの DISABLE オプションを使用して使用禁止にします。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 コマンドの DISABLE 句の ALL TRIGGERS オプションを使えば、1 つの文で、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 の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;
```

トリガーを使用可能にする

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。ただし、トリガーは後で使用禁止にできます。いったん使用禁止にしたトリガーも、必要に応じて再度使用可能にできます。

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 句の ALL TRIGGERS オプションを使えば、1 つの文で、ある表に定義してあるトリガーをすべて使用可能にできます。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;
```

トリガーを使用可能、使用禁止にするために必要な権限

ALTER TABLE コマンドを使用してトリガーを使用禁止または使用可能にするには、表を所有しているか、または表に対して ALTER スキーマ・オブジェクト権限を付与されているか、ALTER ANY TABLE システム権限が必要です。

ALTER TRIGGER コマンドを使用してトリガーを使用禁止または使用可能にするには、トリガーを所有しているか、または ALTER ANY TRIGGER システム権限が必要です。

トリガーに関する情報をリストする

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

これらのデータ・ディクショナリ・ビューの詳細は、『Oracle8 Server 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。

```
CREATE 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 name = 'REORDER';

TYPE              TRIGGERING_STATEMENT       TABLE_NAME
```

```
---------------- -------------------------- ------------
AFTER EACH ROW   UPDATE                     INVENTORY

SELECT trigger_body
   FROM user_triggers
   WHERE 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 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や、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）
- 複雑なビジネス・ルールの実施
- 複雑なセキュリティ認可の実施
- 透過的なイベントのログ
- 導出列の値の自動生成

この項では、上記のトリガー・アプリケーションの例を紹介します。これらの例をそのまま使用することはできませんが、トリガーを設計するときの参考にするとよいでしょう。

トリガーで監査する

トリガーは、通常、Oracle の組込み監査機能を補うために使用されます。トリガーを使って、AUDIT コマンドによって記録されたものと同様の情報も記録できますが、トリガーはより

詳細な監査情報が必要な場合に使用してください。たとえば、トリガーを使用して、表について行の値に基づく行単位の監査ができます。

トリガーが「ファイナンシャル」監査機能として動作するのに対し、Oracle AUDIT コマンドは、多くの場合、「機密保護」監査機能として動作します。

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には、Oracle の監査機能によって実現可能なことをトリガーで定義される監査と比較して検討してください。

DDL 監査と DML 監査	標準監査オプションによって、あらゆるタイプのスキーマ・オブジェクトと構造に関する DML と DDL 文の監査がなされます。これに対して、トリガーは、表に対して発行された DML 文の監査だけとなります。
集中監査証跡	すべてのデータベース監査情報は、一括して記録され、Oracle の監査機能によって自動的に使用されます。
宣言方式	トリガーで監査機能を実現させる場合と比べ、標準 Oracle の機能を使用して使用可能となる監査機能は、宣言とメンテナンスが簡単で、エラーはまず発生しません。
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。
セッションと実行時の監査	データベースの監査機能を使用して、監査文が発行されるたび（BY ACCESS）、または監査文を発行するセッションごとに（BY SESSION）、レコードを作成できます。トリガーではセッション単位の監査はできません。トリガーで監査された表が参照されるたびに、監査レコードが作成されます。
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に、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、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。
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、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）と同様に、接続とその切離しを記録できます。

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、整合性制約に従うトリガー文の後に、監査情報が記録されます。監査処理において、整合性制約の例外を生成する文の無意味な実行を防止します。

AFTER 行と 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、行トリガーによって表上の行の値を行単位で監査できます。トリガーを使用して、監査済み SQL 文を発行して「理由コード」を出力できます。これは、行と文レベルの監査状況の両方に便利です。

次の例では、EMP 表に対する変更を行ベースで監査するトリガーを具体的に示します。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。

例 次のトリガーを具体的に示します。

- トリガーを使用して値に基づく監査を行う方法
- パブリック・パッケージ変数の使用方法

コード内のコメントには、トリガーの機能が記述されています。

```
CREATE TRIGGER audit_employee
AFTER INSERT OR DELETE OR UPDATE ON emp
FOR EACH ROW
BEGIN
/* AUDITPACKAGE は、変数 REASON のパブリック・パッケージがあるパッケージです。
    REASON は、EXECUTE AUDITPACKAGE.SET_REASON(reason_string) のような
   コマンドによって、アプリケーションで設定できます。
   パッケージ変数には、セッションの期間の状態があり、
   各セッションには
   すべてのパッケージ変数の個々のコピーがあることに
   注意してください。*/

IF auditpackage.reason IS NULL THEN
   raise_application_error(-20201, 'Must specify reason'
      || ' with AUDITPACKAGE.SET_REASON(reason_string)');
END IF;

/* 上の条件が TRUE であれば、
   ユーザー定義のエラー番号およびメッセージが発生し、
   トリガーは、実行を停止して、
   トリガーを実行する文はロール・バックされます。TRUE でなければ、
   新しい行が、事前定義済みの監査表に挿入されます
   この表名は、AUDIT_EMPLOYEE であり、EMP 表の既存の値と新規の値
   および EMP 表の AUDITPACKAGE の REASON 変数によって定義された
   理由コードも含みます。「old」値は、
   トリガー文が INSERT の場合 NULL であり、
   「new」値は、トリガー文が DELETE の場合
   NULL です。*/


INSERT INTO audit_employee VALUES
   (:old.ssn, :old.name, :old.job_classification, :old.sal,
   :new.ssn, :new.name, :new.job_classification, :new.sal,
   auditpackage.reason, user, sysdate );
END;
```

更新のたびに理由コードが設定されるようにするのであれば、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 TRIGGER audit_employee_reset
AFTER INSERT OR DELETE OR UPDATE ON emp
BEGIN
   auditpackage.set_reason(NULL);
END;
```

先のトリガーは 2 つとも同じタイプの SQL 文によって実行されます。ただし、トリガー文の実行が終了した時に AFTER 文トリガーは 1 度だけ実行されるのに対し、AFTER 行トリガーは、トリガー文が処理対象とする表の各行ごとに 1 度実行されます。

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 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。

```
CREATE OR REPLACE TRIGGER audit_emp
   AFTER INSERT OR UPDATE OR DELETE ON emp
   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', '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', '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', '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;
/
```

整合性制約とトリガー

トリガーと宣言整合性制約を使用してデータ入力を制限できます。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、データベースについての文で、これは常に真です。表内の既存のデータおよび表を操作するあらゆる文に制約が適用されます。

関連項目： 第 9 章「データの整合性の保持」

トリガーは、トランザクションが処理できるものを制約します。トリガーは、それ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、多数の同じルールを施行するためにトリガーを記述できます。ただしトリガーを使う場合は、標準の整合性制約では定義できない複雑なビジネス・ルールを施行するときに限定してください。Oracle の宣言整合性制約機能には、トリガーの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検査	データは、アクセスするどの時点においても、各スキーマ・オブジェクトに対応する整合性制約に定義されたルールのグローバル・セットに準拠します。
宣言方式	標準整合性制約機能によって定義された制約は、トリガーによって定義された同じ制約と比較して、より書き込みやすくエラーが発生しにくい利点があります。

宣言整合性制約によって、たいていのデータ整合性を定義して施行できます。トリガーでは、宣言整合性制約では定義できない複雑なビジネス制約を施行できます。たとえばトリガーは次を施行します。

- UPDATE と DELETE SET NULL、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 制約で認められる式では定義できない複雑な検査制約

トリガーを使用した参照整合性の施行

多くの場合、トリガーを使用して参照整合性を施行できます。ただし、トリガーが使用できるのは、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、関連の従属データはすべて 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、関連の従属データはすべてデフォルト値に置き換えられます）を施行する場合か、あるいは、分散データベースの異なるノードの親表と子表の間に参照整合性を施行する場合に限ります。

トリガーによって参照整合性をメンテナンスするときに、親表に PRIMARY(または UNIQUE)KEY 制約を 宣言しておきます。同じデータベース上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、子表に外部キーも宣言できます。ただし、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 PRIMARY KEY 制約は（CASCADE オプションを使用して、PRIMARY KEY 制約を明示的に削除しない限り）削除はされません。

トリガーによる参照整合性は、次のようにメンテナンスします。

- 1 つのトリガーを子表に定義しなければならない。子表では、値の挿入または更新時に外部キーに対応する親キーがあることをトリガーによりチェックします。
- 親表には、1 つまたは複数のトリガーを定義しなければならない。これら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際に、外部キーの値に対する参照アクション（RESTRICT、CASCADE、SET NULL）を行います。親表への挿入にアクションは不要です（外部キーには依存しない）。

以降の項では、参照整合性の施行に必要なトリガーの例を紹介します。これらの例では EMP 表と DEPT 表を使用します。

行をロックする文が含まれているトリガーもあります (SELECT ... FOR UPDATE)。この操作は、処理中の行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 または UPDATE 文の前に、外部キーの値が親キー内に存在することを保証します。次の例に含まれる変更表の例外によって、このトリガーを UPDATE_SET_DEFAULT と UPDATE_CASCADE トリガーとともに使用できます。このトリガーを単独で使用する場合には、例外を取り除くことができます。

```
CREATE TRIGGER emp_dept_check
BEFORE INSERT OR UPDATE OF deptno ON emp
FOR EACH ROW WHEN (new.deptno IS NOT NULL)

-- Before a row is inserted, or DEPTNO is updated in the EMP
-- table, fire this trigger to verify that the new foreign
-- key value (DEPTNO) is present in the DEPT 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 PRINT_SALARY_CHANGES_cursor (dn NUMBER) IS
   SELECT deptno
   FROM dept
   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 トリガーと DELETE RESTRICT トリガー 次のトリガーは、DEPT 表の主キーに対する UPDATE および DELETE RESTRICT 参照アクションを施行するため、DEPT 表に定義します。

```
CREATE TRIGGER dept_restrict
BEFORE DELETE OR UPDATE OF deptno ON dept
FOR EACH ROW

-- Before a row is deleted from DEPT or the primary key
-- (DEPTNO) of DEPT is updated, check for dependent
-- foreign key values in EMP; 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 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; /* 依存行が存在する場合 */
   ELSE
      RAISE employees_not_present; /* 依存行が存在しない場合 */
   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;
```

警告： このトリガーは、自己参照型の表（つまり、主キーまたは一意キーと外部キーの両方がある表）では動作しません。また、このトリガーは、トリガーの循環（A を実行する B を A が実行するなど）を認めません。

親表に対する UPDATE トリガーと DELETE SET NULL トリガー 次のトリガーは、DEPT 表の主キーの UPDATE と DELETE SET NULL 参照アクションを施行するため、DEPT 表に定義します。

```
CREATE TRIGGER dept_set_null
AFTER DELETE OR UPDATE OF deptno ON dept
FOR EACH ROW

-- Before a row is deleted from DEPT or the primary key
-- (DEPTNO) of DEPT is updated, set all corresponding
-- dependent foreign key values in EMP to NULL.
BEGIN
   IF UPDATING AND :OLD.deptno != :NEW.deptno OR DELETING THEN
      UPDATE emp SET emp.deptno = NULL
         WHERE emp.deptno = :old.deptno;
   END IF;
END;
```

親表に対する DELETE CASCADE トリガー 次のトリガーは DEPT 表上にあり、DEPT 表の主キーの DELETE CASCADE 参照アクションを施行します。

```
CREATE TRIGGER dept_del_cascade
AFTER DELETE ON dept
FOR EACH ROW

-- Before a row is deleted from DEPT, delete all
-- rows from the EMP table whose DEPTNO is the same as
-- the DEPTNO being deleted from the DEPT table.
BEGIN
   DELETE FROM emp
      WHERE emp.deptno = :old.deptno;
END;
```

注意： 通常、DELETE CASCADE のコードは、更新と削除の両方の可能性を考慮して、UPDATE SET NULL または UPDATE SET DEFAULT のコードと組み合されます。

親表に対する UPDATE CASCADE トリガー 次のトリガーは、DEPT 表で部門番号が更新されると、その変更が、変更に依存する EMP 表の外部キーに反映されることを保証します。

```
-- 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 PACKAGE integritypackage AS
   updateseq NUMBER;
END integritypackage;

CREATE or replace PACKAGE BODY integritypackage AS
END integritypackage;
ALTER TABLE emp ADD update_id NUMBER; -- create flag col.

CREATE TRIGGER dept_cascade1 BEFORE UPDATE OF deptno ON dept
DECLARE
   dummy NUMBER;

-- Before updating the DEPT 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 TRIGGER dept_cascade2 AFTER DELETE OR UPDATE
   OF deptno ON dept FOR EACH ROW

-- For each department number in DEPT that is updated,
-- cascade the update to dependent foreign keys in the
-- EMP table.  Only cascade the update if the child row
-- has not already been updated by this trigger.
BEGIN
```

```
   IF UPDATING THEN
      UPDATE emp
         SET deptno = :new.deptno,
         update_id = integritypackage.updateseq /*1番目から */
         WHERE emp.deptno = :old.deptno
         AND update_id IS NULL;
         /* この同じトリガー文によって起動された
            3番目のトリガーで更新されない場合だけNULL */
   END IF;
   IF DELETING THEN

   -- Before a row is deleted from DEPT, delete all
   -- rows from the EMP table whose DEPTNO is the same as
   -- the DEPTNO being deleted from the DEPT table.
      DELETE FROM emp
      WHERE emp.deptno = :old.deptno;
   END IF;
END;
CREATE TRIGGER dept_cascade3 AFTER UPDATE OF deptno ON dept
BEGIN UPDATE emp
   SET update_id = NULL
   WHERE update_id = integritypackage.updateseq;
END;
```

注意： このトリガーによって EMP 表が更新されるので、使用可能であれば、EMP_DEPT_CHECK トリガーも起動します。結果として発生する変更表のエラーは、EMP_DEPT_CHECK トリガーによってトラップされます。エラーのトラップを必要とするトリガーは、運用環境で常に正しく作動することを保証するために、慎重にテストしなければなりません。

複雑な検査制約を施行する

トリガーは、参照整合性以外の整合性ルールも施行できます。たとえば、次のトリガーは、トリガー文の実行時に複雑な検査を実施します。コード内のコメントには、トリガーの機能が記述されています。

```
CREATE TRIGGER salary_check
BEFORE INSERT OR UPDATE OF sal, job ON emp
FOR EACH ROW
DECLARE
   minsal               NUMBER;
   maxsal               NUMBER;
   salary_out_of_range  EXCEPTION;
BEGIN

/* 従業員の新しい職位分類のために
```

```
   SALGRADE 表から最低額と最高額の給与を取り出して
   MINSAL および MAXSAL に挿入する。*/

SELECT minsal, maxsal INTO minsal, maxsal FROM salgrade
   WHERE job_classification = :new.job;


/* 従業員の新しい給与が職位分類の制限よりも
   少なかい場合または多い場合には、
   例外が発生します。すると例外メッセージが戻され、
   トリガーを起動した保留中の INSERT または UPDATE 文は
   ロール・バックされます。*/

   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.name);
   WHEN NO_DATA_FOUND THEN
      raise_application_error(-20322,
         'Invalid Job Classification '
         ||:new.job_classification);
END;
```

複雑なセキュリティ認可とトリガー

一般に、トリガーは表データの複雑なセキュリティの施行に使用します。Oracle のデータベース・セキュリティ機能では定義できないセキュリティの施行にだけ、トリガーを使用します。たとえば、週末や休日、休業日には EMP 表の給与データを更新できないようにすることが可能です。

トリガーを使用して複雑なセキュリティ認可を施行するときには、BEFORE 文トリガーを使用するのが最もよい方法です。BEFORE 文を使用すると、次のような利点があります。

- トリガー文の実行前にセキュリティ検査が実施されるため、未許可の文が含まれていた場合の無駄な作業をせずに済む。
- セキュリティ検査は、トリガー文の処理対象となる各行ごとではなく、トリガー文 1 つにつき 1 回実施される。

例 次に、セキュリティを施行するために使用するトリガーの例を示します。コメントには、トリガーの機能が記述されています。

```
CREATE TRIGGER emp_permit_changes
```

```
BEFORE INSERT OR DELETE OR UPDATE ON emp
DECLARE
   dummy INTEGER;
   not_on_weekends EXCEPTION;
   not_on_holidays EXCEPTION;
   non_working_hours EXCEPTION;
BEGIN
   /* 週末のチェック */
   IF (TO_CHAR(sysdate, 'DY') = 'SAT' OR
      TO_CHAR(sysdate, 'DY') = 'SUN') THEN
      RAISE not_on_weekends;
   END IF;
   /* 会社の休日のチェック */
   SELECT COUNT(*) INTO dummy FROM company_holidays
      WHERE TRUNC(day) = TRUNC(sysdate);
      /* TRUNC により、日付の時間部分は除かれる */
   IF dummy > 0 THEN
      RAISE not_on_holidays;
   END IF;
   /* 就業時間のチェック（午前８時〜午後６時）*/
   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 table during non-working hours');
END;
```

透過的なイベント・ロギングとトリガー

特定のイベントに従ってデータベース中の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。

例 13-19 ページの REORDER トリガーの例は、一定の条件を満たすと（つまり、トリガー文が発行され、PARTS_ON_HAND の値が REORDER_POINT の AND より小さい場合）、パーツを再度注文するトリガーを示しています。

導出列値とトリガー

トリガーは、INSERT 文または UPDATE 文に値を指定することによって、例の値を取り出せます。このタイプの トリガーは、同じ行上の異なる列の値に基づいて、特定の列の値を取り出す場合に便利です。このタイプのオペレーションを実行するには、BEFORE 行トリガーが必要です。

- 取り出された値をトリガー文で使用するには、INSERT または UPDATE を実行する前に、導出値が取り出されていなければならない。
- INSERT または UPDATE 文の処理対象となる各行ごとに、トリガーを実行しなければならない。

例 次に、行の挿入や更新が行われるたびに、新しい列値を導出するトリガーの使用例を示します。コード内のコメントは、その機能を説明しています。

```
BEFORE INSERT OR UPDATE OF ename ON emp

/* ENAME フィールドを更新する前に、
   UPPERNAME および SOUNDEXNAME フィールドの値を導出します。ユーザーは、
   これらのフィールドを直接更新することを制限されなければなりません。*/
FOR EACH ROW

BEGIN
   :new.uppername := UPPER(:new.ename);
   :new.soundexname := SOUNDEX(:new.ename);
END;
```

14

動的 SQL の使用

この章では、動的 SQL パッケージ DBMS_SQL について説明します。この章のトピックは次のとおりです。

- DBMS_SQL パッケージと Oracle コール・インタフェースの違い
- DBMS_SQL パッケージを使用した DDL の実行
- DBMS_SQL パッケージで提供されるプロシージャおよび関数

動的 SQL の概要

動的 SQL を使用して、ストアド・プロシージャおよび無名 PL/SQL ブロックを作成できます。動的 SQL 文は、ソース・プログラムに埋め込まれているのではなく、実行時にプログラムにインプットされるか、またはプログラムによって作成される文字列に格納されます。これにより、さらに広い用途のプロシージャを作成できます。たとえば、動的 SQL を使用すれば、実行時まで名前がわからない表上で動作するプロシージャの作成もできます。

また、データ操作言語（DML）文やデータ定義言語（DDL）文はいずれも、DBMS_SQL パッケージを使用して解析できます。これは、直接 PL/SQL を使用してもデータ定義言語文を解析できないという問題を解決するために役立ちます。たとえば、DBMS_SQL パッケージで提供される PARSE プロシージャを使用して、ストアド・プロシージャ内部から DROP TABLE 文を発行できます。

DBMS_SQL パッケージを作成する

DBMS_SQL パッケージを作成するには、ユーザー SYS として接続し、DBMSSQL.SQL スクリプトと PRVTSQL.PLB スクリプトを実行します。これらのスクリプトは、CATPROC.SQL スクリプトによって自動的に実行されます。

> **関連項目：** このパッケージを実行するユーザーに必要な権限の付与の詳細は、10-36 ページの「プロシージャを実行するために必要な権限」を参照してください。

DBMS_SQL の使用

ストアド・プロシージャ内部から動的 SQL を使用する機能は、通常、Oracle コール・インタフェース （OCI）の手順に従っています。この章で示されている概念の詳細は、『Oracle 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。

PL/SQL は、C などの他の一般的なプログラム言語とは、多少異なります。たとえば、PL/SQL ではアドレス（ポインタとも呼ばれる）は、ユーザーには参照できません。そのため、Oracle コール・インタフェースと DBMS_SQL パッケージの間には、いくつか相違点があります。相違点は、次のとおりです。

- OCI はアドレスによるバインドを使用するのに対して、DBMS_SQL パッケージは値によるバインドを使用する。
- 無名ブロックの OUT パラメータの値を検索するには、DBMS_SQL を使用して VARIABLE_VALUE をコールしなければならない。また、行をフェッチして、行内の列の値を実際にプログラムに取り出した後、COLUMN_VALUE をコールしなければなりません。
- DBMS_SQL パッケージの現在のリリースは CANCEL カーソル・プロシージャを提供していない。
- NULL を PL/SQL 変数の値として完全にサポートしているので、インジケータ変数を用意する必要はない。

DBMS_SQL パッケージの使用例は、次のとおりです。Oracle コール・インタフェースのユーザーには、このコードはかなり簡単に思われるかもしれません。この例で使用されている個々の関数とプロシージャの説明は、この章の後半に記述されています。

14-28 ページから始まる詳細な例は、DBMS_SQL パッケージを使用して問合せ文を動的に作成する方法を示しています。この文のテキストはコンパイル時にわかるので、この例では、実際には 動的 SQL を使用する必要はありませんが、ここでは、このパッケージの概念をわかりやすく説明しています。

```
/* DEMO プロシージャでは、DEMO の実行時に指定した給与よりも
 * 高い給与のすべての従業員を EMP 表から
 * 削除します。*/

CREATE OR REPLACE PROCEDURE demo(salary IN NUMBER) AS
    cursor_name INTEGER;
    rows_processed INTEGER;
BEGIN
    cursor_name := dbms_sql.open_cursor;
    dbms_sql.parse(cursor_name, 'DELETE FROM emp WHERE sal > :x',
                   dbms_sql);
    dbms_sql.bind_variable(cursor_name, ':x', salary);
    rows_processed := dbms_sql.execute(cursor_name);
    dbms_sql.close_cursor(cursor_name);
EXCEPTION
```

```
WHEN OTHERS THEN
    dbms_sql.close_cursor(cursor_name);
END;
```

実行の流れ

図 14-1 は、プロシージャ・コールの一般的な流れを示しています。この図の後に、これらのプロシージャが概説されています。

これらのプロシージャのそれぞれは、14-8 ページで詳細に説明しています。

OPEN_CURSOR

SQL 文を処理するには、オープン・カーソルを持っていなければなりません。OPEN_CURSOR 関数をコールすると、Oracle が保持している有効なカーソルを表すデータ構造のカーソル ID 番号を受け取ることができます。これらのカーソルは、プリコンパイラ、OCI、または PL/SQL レベルで定義されたカーソルとは異なり、DBMS_SQL パッケージでだけ使用されます。

PARSE

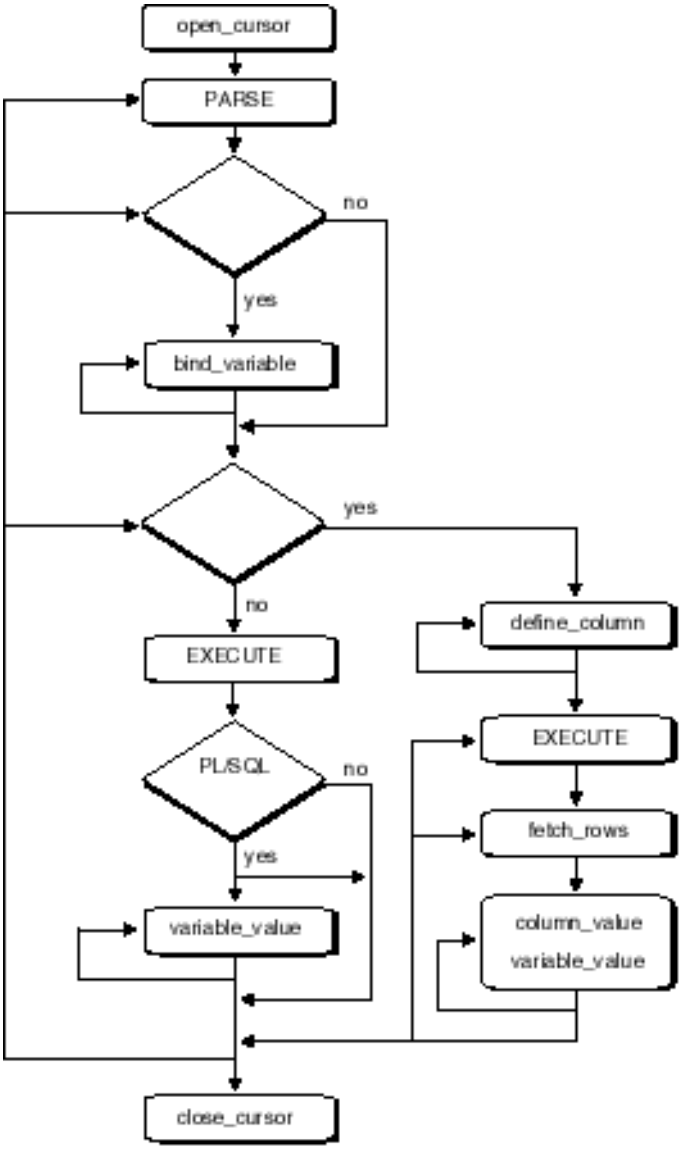
SQL 文はすべて、PARSE プロシージャをコールして解析しなければなりません。この文を解析すると、文の構文がチェックされ、それがプログラム内のカーソルと関連付けられます。

> **関連項目：** SQL 文の解析方法は、『Oracle8 Server 概要』を参照してください。

データ操作言語文またはデータ定義言語文をすべて解析できます。データ定義言語文は、解析時に実行され、それにより暗黙のコミットが実行されます。

注意： パッケージやプロシージャを削除するためにデータ定義言語文を解析する場合、パッケージ内のプロシージャがまだ使用中であると、デッドロックが起こる可能性があります。プロシージャのコール後、実行がユーザー側に戻されるまで、そのプロシージャは使用中であるとみなされます。このようなデッドロックは、5 分後にタイムアウトします。

図 14-1　DBMS_SQL 実行の流れ



BIND_VARIABLE または BIND_ARRAY

多くのデータ操作言語文では、プログラム内のデータが Oracle に入力されることが必要です。実行時に提供される入力データを含んでいる SQL 文を定義する場合、SQL 文内でプレースホルダを使用して、データが提供されなければならない場所にマークを付ける必要があります。

SQL 文内の個々のプレースホルダに対して BIND_VARIABLE プロシージャまたは BIND_ARRAY プロシージャをコールして、プログラム内の変数の値（または配列の値）をプレースホルダに提供しなければなりません。SQL 文が引き続き実行されると、Oracle はプログラムが入力変数および出力変数、またはバインド変数に入れたデータを使用します。

DBMS_SQL では、何回でも DML 文を実行できますが、その都度異なるバインド変数を使用します。BIND_ARRAY プロシージャを使用すると、スカラーの配列をバインドでき、各値は 1 回の EXECUTE につき 1 度だけ入力変数として使用されます。これは、OCI によってサポートされている配列インタフェースに類似しています。

DEFINE_COLUMN または DEFINE_COLUMN_LONG、DEFINE_ARRAY

SELECT 文内で選択されている行の列は、選択リスト内での相対位置（左から右）によって識別されます。問合せの場合は、INTO 句が静的問合せに対して行うのとほとんど同様の方法で、定義プロシージャのいずれか（DEFINE_COLUMN または DEFINE_COLUMN_LONG、DEFINE_ARRAY）をコールして SELECT 値を受け取る変数を指定しなければなりません。

DEFINE_COLUMN を使用して LONG 列以外の列を定義するのと同様に、DEFINE_COLUMN_LONG プロシージャを使用して LONG 列を定義します。COLUMN_VALUE_LONG を使用して LONG 列からフェッチする前に、DEFINE_COLUMN_LONG をコールしなければなりません。

DEFINE_ARRAY プロシージャを使用して、単一の SELECT 文内に行をフェッチ する、PL/SQL 配列を定義します。COLUMN_VALUE を使用して行をフェッチする前に、DEFINE_ARRAY をコールしなければなりません。

EXECUTE

EXECUTE 関数をコールして、SQL 文を実行します。

FETCH_ROWS または EXECUTE_AND_FETCH

問合せを満たす行を検索するには、FETCH_ROWS 関数をコールします。引き続きフェッチすると、そのたびに別の行が検索され、これは、フェッチが行を検索できなくなるまで続きます。反復が 1 回のときに EXECUTE をコールしている場合は、EXECUTE をコールしてから FETCH_ROWS をコールするより、EXECUTE_AND_FETCH をコールする方が効率的です。

VARIABLE_VALUE または COLUMN_VALUE、COLUMN_VALUE_LONG

問合せの場合は、COLUMN_VALUE をコールして、FETCH_ROWS コールで検索する値を判断します。PL/SQL プロシージャのコールを含む無名ブロックの場合は、VARIABLE_VALUE をコールして、実行時に PL/SQL プロシージャの出力変数に割り当てられた値を検索します。

LONG データベース列（サイズは最大 2GB まで可能）の一部だけをフェッチするには、COLUMN_VALUE_LONG プロシージャを使用します。列値のオフセット（単位はバイト）と、フェッチするバイト数を指定できます。

CLOSE_CURSOR

セッションにカーソルが必要なくなると、CLOSE_CURSOR をコールして、カーソルをクローズします。Oracle Open Gateway を使用している場合は、これ以外のときにも カーソルをクローズしなければならないことがあります。詳細は、Oracle Open Gateway 関連マニュアルを参照してください。

カーソルのクローズをしないと、カーソルが必要なくなっても、そのカーソルが使用しているメモリーは割り当てられたままになります。

動的 SQL のセキュリティ

この項では、Oracle Server または Trusted Oracle Server 使用時の、DBMS_SQL プロシージャのセキュリティ・ドメインについて説明します。

Oracle Server のユーザーの場合

無名 PL/SQL ブロックからコールされた DBMS_SQL プロシージャは、現ユーザーの権限を使用して実行されます。ストアド・プロシージャからコールされた DBMS_SQL プロシージャは、そのストアド・プロシージャの所有者の権限を使用して実行されます。

したがって、あるユーザーがプロシージャを作成し、その EXECUTE 権限を 2 番目のユーザーに付与する場合、2 番目ユーザーには、ストアド・プロシージャの本体で実行されるすべての DML 操作の特権も明示的に付与しなければなりません。（ロールはストアド・プロシージャの中では使用禁止であるため、ロールを介してこれらの権限を付与することはできません。）DML 操作の権限を付与しないと、実行時にエラー・メッセージ ORA-01031: 「権限が不足しています。」が示される場合があります。

Trusted Oracle Server のユーザーの場合

無名 PL/SQL ブロックからコールされた DBMS_SQL プロシージャは、現ユーザーの権限を使用して実行されます。ストアド・プロシージャからコールされた DBMS_SQL プロシージャは、任意アクセス制御（DAC）、およびストアド・プロシージャと現ユーザーに付与されている必須アクセス管理（MAC）権限の共用体の所有者のシステム特権を使用して実行されます。

プロシージャと関数

表 14–1 では、DBMS_SQL パッケージに関する各プロシージャおよび関数を概説しています。詳細は、この章の後半で説明します。

関連項目： これらの関数の使用例は、14-28 ページの「DBMS_SQL の使用例」を参照してください。

表 14–1　DBMS_SQL パッケージの関数とプロシージャ

関数 / プロシージャ	説明	参照ページ
OPEN_CURSOR	新しいカーソルのカーソル ID 番号を戻します。	14-9 ページ
PARSE	指定された文を解析します。	14-10 ページ
BIND_VARIABLE	所定の値を所定の変数にバインドします。	14-11 ページ
BIND_ARRAY	所定の値を、所定の配列にバインドします。	14-11 ページ
DEFINE_COLUMN	指定されたカーソルから選択され、SELECT 文だけで使用される列を定義します。	14-15 ページ
DEFINE_ARRAY	指定されたカーソルから選択され、SELECT 文だけで使用される配列を定義します。	14-17 ページ
DEFINE_COLUMN_LONG	指定されたカーソルから選択され、SELECT 文だけで使用される LONG 列を定義します。	14-18 ページ
EXECUTE	所定のカーソルを実行します。	14-19 ページ
EXECUTE_AND_FETCH	所定のカーソルを実行して、行をフェッチします。	14-19 ページ
FETCH_ROWS	所定のカーソルから行をフェッチします。	14-19 ページ
COLUMN_VALUE	所定のカーソルの所定の位置のカーソル要素の値を戻します。	14-20 ページ
COLUMN_VALUE_LONG	DEFINE_COLUMN_LONG を使用して定義された LONG 列の選択された部分を戻します。	14-22 ページ

表 14–1　DBMS_SQL パッケージの関数とプロシージャ

関数 / プロシージャ	説明	参照ページ
VARIABLE_VALUE	指定された変数の値を所定のカーソルに戻します。	14-23 ページ
IS_OPEN	指定されたカーソルがオープンの場合に TRUE を戻します。	14-24 ページ
DESCRIBE_COLUMNS	DBMS_SQL によってオープンおよび解析されたカーソルの列を記述します。	14-24 ページ
CLOSE_CURSOR	指定されたカーソルをクローズして、メモリーを解放します。	14-26 ページ
LAST_ERROR_POSITION	エラーが発生した SQL テキスト内のバイト・オフセットを戻します。	14-28 ページ
LAST_ROW_COUNT	フェッチされた行数の累積を戻します。	14-28 ページ
LAST_ROW_ID	最後に処理された行の ROWID を戻します。	14-28 ページ
LAST_SQL_FUNCTION_CODE	文の SQL 機能コードを戻します。	14-28 ページ

OPEN_CURSOR 関数

OPEN_CURSOR をコールして、新しいカーソルをオープンします。このカーソルが必要なくなったら、CLOSE_CURSOR をコールして、明示的にクローズしなければなりません。

カーソルを使用して、同じ SQL 文を繰り返し実行したり、新しい SQL 文を実行することができます。カーソルが再使用されると、新しい SQL 文の解析時に、対応するカーソル・データ域の内容がリセットされるので、再使用する前にクローズし、再度オープンする必要はありません。

OPEN_CURSOR の構文

OPEN_CURSOR 関数は、新しいカーソルのカーソル ID 番号を戻します。この関数の構文は、次のとおりです。

```
DBMS_SQL.OPEN_CURSOR RETURN INTEGER;
```

PARSE プロシージャ

PARSE をコールして、指定されたカーソル内の指定された文を解析します。すべての文はただちに解析されます。これは、将来のバージョンで変更される可能性があるので、この動作に依存しないでください。

PARSE の構文

PARSE プロシージャのパラメータは、表 14–2 で説明します。このプロシージャの構文は、次のとおりです。

```
DBMS_SQL.PARSE(
        c                   IN INTEGER,
        statement           IN VARCHAR2,
        language_flag       IN INTEGER);
```

上記構文を持つ SQL 文の解析のサイズ制限は **size_t** で、使用しているマシン上に連続して割り当てられる最大サイズになります。また、PARSE プロシージャでは、大規模な SQL 文について次の構文もサポートしています。

```
DBMS_SQL.PARSE(
        c                   IN INTEGER,
        statement           IN VARCHAR2S,
        lb                  IN INTEGER,
        ub                  IN INTEGER,
        lfflg               IN BOOLEAN,
        language_flag       IN INTEGER);
```

このプロシージャは、PL/SQL 表の文の要素を連結し、結果となる文字列を解析します 。このプロシージャを使用し、文を分割することによって、単一の VARCHAR2 変数の制限より長い文を解析できます。

表 14–2　DBMS_SQL.PARSE プロシージャのパラメータ

パラメータ	説明
c	文を解析するカーソルの ID 番号を指定します。
statement	解析する SQL 文を指定します。SQL 文に最終セミコロンが含まれていてはなりません。
lb	文内の要素についての下限を提供します。
ub	文内の要素についての上限を提供します。

表 14–2　DBMS_SQL.PARSE プロシージャのパラメータ

パラメータ	説明
lfflg	TRUE の場合、連結している各要素の後に改行を挿入します。
language_flag	このパラメータは、Oracle が SQL 文をどのように処理するかを決定します。このパラメータで、次のオプションが認識されます。 V6 - バージョン 6 の動作を指定します。 V7 - Oracle7 の動作を指定します。 NATIVE - プログラムの接続先のデータベースに対する通常の動作を指定します。

大規模な SQL 文字列を解析するための VARCHAR2S データ型

32 KB を超える SQL 文を解析するために、DBMS_SQL パッケージでは、PL/SQL 表を利用して文字列の表を PARSE プロシージャに渡します。これらの文字列は連結された後、Oracle Server に渡されます。

ローカル変数を VARCHAR2S データ型として宣言し、次に、PARSE プロシージャを使用して、大規模な SQL 文を VARCHAR2S として解析できます。

VARCHAR2S データ型の定義は、次のとおりです。

```
type varchar2s is table of varchar2(256) index by binary_integer;
```

指定のカーソル内の大規模な SQL 文を解析するには、DBMS_SQL.PARSE をコールします。すべての文はただちに解析されます。

BIND_VARIABLE プロシージャと BIND_ARRAY プロシージャ

文内の変数の名前に基づいて、カーソル内の所定の変数に所定の値または値のセットをバインディングするには、BIND_VARIABLE または BIND_ARRAY をコールします。この変数が IN 変数または IN/OUT 変数である場合は、指定されたバインド値が、変数タイプまたは配列タイプに有効でなければなりません。OUT 変数のバインド値は無視されます。

SQL 文のバインド変数または配列は、名前によって識別されます。バインド変数またはバインド配列に値をバインドする場合は、次の例に示されているように、文内のバインド変数を識別する文字列の先頭にはコロンを付ける必要があります。

```
SELECT emp_name FROM emp WHERE SAL > :X;
```

この例では、対応するバインド・コールは次のようになります。

```
BIND_VARIABLE(cursor_name, ':X', 3500);
```

BIND_VARIABLE の構文

BIND_VARIABLE のプロシージャおよび関数のパラメータは、表 14–3 で説明します。これらのプロシージャおよび関数の構文は、次のとおりです。BIND_VARIABLE がオーバーロードされると、さまざまなデータ型を受け入れることに注意してください。

```
DBMS_SQL.BIND_VARIABLE(
        c               IN INTEGER,
        name            IN VARCHAR2,
        value           IN <datatype>);
```

この <datatype> は、次のいずれかのタイプでなければなりません。

```
NUMBER
DATE
MLSLABEL
VARCHAR2 CHARACTER SET ANY_CS
BLOB
CLOB CHARACTER SET ANY_CS
BFILE
```

> **関連項目：** 第 6 章「ラージ・オブジェクト (LOB)」で、BLOB および CLOB、BFILE データ型を説明しています。

BIND_VARIABLE では、次の構文もサポートされています。大カッコ [ ] は、BIND_VARIABLE 関数のオプション・パラメータを示します。

```
DBMS_SQL.BIND_VARIABLE(
        c               IN INTEGER,
        name            IN VARCHAR2,
        value           IN VARCHAR2 CHARACTER SET ANY_CS
      [,out_value_size IN INTEGER]);
```

CHAR および RAW、ROWID データをバインドするには、構文で次のバリエーションを使用できます。

```
DBMS_SQL.BIND_VARIABLE_CHAR(
        c               IN INTEGER,
        name            IN VARCHAR2,
        value           IN CHAR CHARACTER SET ANY_CS
      [,out_value_size IN INTEGER]);
DBMS_SQL.BIND_VARIABLE_RAW(
        c               IN INTEGER,
        name            IN VARCHAR2,
        value           IN RAW
      [,out_value_size IN INTEGER]);
DBMS_SQL.BIND_VARIABLE_ROWID(
        c               IN INTEGER,
        name            IN VARCHAR2,
```

```
        value          IN ROWID);
```

表 14–3　DBMS_SQL.BIND_VARIABLE プロシージャのパラメータ

パラメータ	説明
c	値をバインディングするカーソルの ID 番号を指定します。
name	文内の変数の名前を提供します。
value	カーソル内の変数にバインディングする値を指定します。IN 変数と IN/OUT 変数の場合、この値は、このパラメータで渡される値のタイプと同じタイプです。
out_value_size	VARCHAR2、RAW、CHAR OUT、IN/OUT などの変数の最大予測 OUT 値サイズ（単位はバイト）です。サイズが指定されないと、現行値の長さが使用されます。

バルク配列バインド

バルク選択および挿入、更新、削除では、多くのコールを 1 つにまとめることによって、アプリケーションのパフォーマンスを改善できます。DBMS_SQL パッケージを使用することによって、PL/SQL 表型を使用して、データ配列に対して処理を実行できます。

表項目は、バインドされていない、同種のコレクションです。持続記憶域においては、表項目は他の関係表に似ており、組込みの配列を持ちません。ただし、表項目が、作業領域に持ち込まれたり（問合せまたは持続データのナビゲーション・アクセスによって）、PL/SQL の変数またはパラメータの値として作成されたりした場合、要素の値を取得および設定するための配列スタイルの構文とともに使用できるサブスクリプトが表項目の要素に与えられます。

これらの要素のサブスクリプトは稠密である必要はなく、負数を含むあらゆる数値を使用できます。たとえば、表項目は、-10 および 2、7 の位置だけにある要素を含むことができます。

表項目が一時作業領域から持続記憶域に移動されるとき、サブスクリプトは格納されません。つまり、表項目は、持続記憶域内では順番が付いていません。

表は、バインド実行時に、PL/SQL バッファからローカルの DBMS_SQL バッファへコピーされ（全スカラー・タイプについて同様）、ローカルの DBMS_SQL バッファから操作されます。したがって、バインド・コールの後に表を変更した場合でも、その変更によって実行方法が影響を受けることはありません。

スカラーおよび LOB 配列のタイプ

ローカル変数を、次のいずれかの表項目型として宣言できます。これらの表項目型は、DBMS_SQL において、パブリック・タイプとして定義されています。

```
type Number_Table is table of Number index by binary_integer;
type Varchar2_Table is table of varchar2(2000) index by binary_integer;
type Date_Table is table of Date index by binary_integer;
```

```
type Blob_Table is table of Blob index by binary_integer;
type Clob_Table is table of Clob index by binary_integer;
type Bfile_Table is table of Bfile index by binary_integer;
```

BIND_ARRAY の構文

配列全体またはある範囲の配列をバインドする BIND_ARRAY プロシージャのパラメータは、表 14–4 で説明します。このプロシージャの構文は、次のとおりです。BIND_ARRAY プロシージャはオーバーロードされて、さまざまなデータ型を受け入れることに注意してください。

```
DBMS_SQL.BIND_ARRAY(
        c                  IN INTEGER,
        name               IN VARCHAR2,
        <table_variable>   IN <datatype>
      [,index1              IN INTEGER,
        index2              IN INTEGER)] );
```

ここで、<table_variable> とそれに対応する <datatype> は、次のいずれかの組合せになります。

```
<num_tab>      Number_Table
<vchr2_tab>    Varchar2_Table
<date_tab>     Date_Table
<blob_tab>     Blob_Table
<clob_tab>     Clob_Table
<bfile_tab>    Bfile_Table
```

表 14–4　DBMS_SQL.BIND_ARRAY プロシージャのパラメータ

パラメータ	説明
c	値をバインディングするカーソルの ID 番号を指定します。
name	文内の配列の名前を提供します。
table_variable	<datatype> として宣言されたローカル変数を指定します。
index1	範囲の下限を示す表要素の索引を提供します。
index2	範囲の上限を示す表要素の索引を提供します。

範囲をバインディングするためには、表は範囲を指定する要素を含んでいなければなりませんが ― タブ（index1）とタブ（index2）― 範囲は稠密でなくても構いません。index1 は、index2 以下でなければなりません。タブ（index1）とタブ（index2）の間にあるすべての要素は、バインドで使用されます。

バインド・コールで索引を指定せず、文内の 2 つの異なるバインドで、異なる数の要素を含む表が指定された場合、実際に使用される要素の数は、すべての表における最小値となりま

す。索引を指定する場合もこの例で、最小範囲はすべての表について 2 つの索引の間で選択されます。

問合せ内のすべてのバインド変数が、配列バインドである必要はありません。その一部は通常のバインドであっても構わず、式の評価などでは、配列の各要素について同じ値が使用されます。

> **関連項目：** 配列のバインド方法の例は、14-31 ページの「例 3 および例 4、例 5: バルク DML」を参照してください。

問合せの処理

動的 SQL を使用して問合せを処理する場合は、次のステップを実施しなければなりません。

1. DEFINE_COLUMN または DEFINE_COLUMN_LONG、DEFINE_ARRAY をコールして、SELECT 文が戻した値を受け取る変数を指定します。
2. EXECUTE をコールして、SELECT 文を実行します。
3. FETCH_ROWS（または EXECUTE_AND_FETCH）をコールして、問合せを満足させる行を検索します。
4. 問合せに対する FETCH_ROWS コールによって検索された列の値を判断するには、COLUMN_VALUE または COLUMN_VALUE_LONG をコールします。PL/SQL プロシージャのコールを含む無名ブロックを使用した場合は、VARIABLE_VALUE をコールして、これらのプロシージャの出力変数に割り当てられた値を検索します。

DEFINE_COLUMN プロシージャ

このプロシージャが使用できるのは、SELECT カーソルだけです。指定されたカーソルから選択される列を定義するには、DEFINE_COLUMN をコールします。定義されている列は、指定されたカーソル内の文の SELECT リスト内での相対位置によって識別されます。COLUMN 値のタイプによって、定義されている列のタイプが決まります。

DEFINE_COLUMN の構文

DEFINE_COLUMN プロシージャのパラメータは、表 14–5 で説明します。このプロシージャの構文は、次のとおりです。DEFINE_COLUMN がオーバーロードされて、さまざまなデータ型を受け入れることに注意してください。

```
DBMS_SQL.DEFINE_COLUMN(
        c               IN INTEGER,
        position        IN INTEGER
         column            IN <datatype>);
```

この <datatype> は、次のいずれかのタイプでなければなりません。

```
NUMBER
DATE
```

```
MLSLABEL
BLOB
CLOB CHARACTER SET ANY_CS
BFILE
```

関連項目： 第 6 章「ラージ・オブジェクト (LOB)」で、BLOB および CLOB、BFILE データ型を説明しています。

DEFINE_COLUMN プロシージャでは、次の構文もサポートされています。

```
DBMS_SQL.DEFINE_COLUMN(
        c               IN INTEGER,
        position        IN INTEGER,
        column          IN VARCHAR2 CHARACTER SET ANY_CS,
        column_size     IN INTEGER);
```

CHAR および RAW、ROWID データを持つ列を定義するには、プロシージャ構文で次のバリエーションを使用できます。

```
DBMS_SQL.DEFINE_COLUMN_CHAR(
        c               IN INTEGER,
        position        IN INTEGER,
        column          IN CHAR CHARACTER SET ANY_CS,
        column_size     IN INTEGER);
DBMS_SQL.DEFINE_COLUMN_RAW(
        c               IN INTEGER,
        position        IN INTEGER,
        column          IN RAW,
        column_size     IN INTEGER);
DBMS_SQL.DEFINE_COLUMN_ROWID(
        c               IN INTEGER,
        position        IN INTEGER,
        column          IN ROWID);
```

表 14–5　DBMS_SQL.DEFINE_COLUMN プロシージャのパラメータ

パラメータ	説明
c	選択されることが定義されている行のカーソルの ID 番号です。
position	定義されている行内の列の相対位置です。文の最初の列は、位置 1 を持っています。
column	定義されている列の値です。この値のタイプによって、定義されている列のタイプが決まります。
column_size	VARCHAR2 型、CHAR 型、および RAW 型の列に対する列値の最大予測サイズ（単位はバイト）です。

DEFINE_ARRAY プロシージャ

DEFINE_ARRAY をコールして、（FETCH_ROWS コールによって）行をフェッチする、適切な表の変数を定義します。このプロシージャを使用することによって、単一の SELECT 文から、行を一括してフェッチできます。1 回のフェッチ・コールで、PL/SQL の集合体オブジェクトに多数の行をフェッチできます。

行をフェッチすると、この行は COLUMN_VALUE コールを実行するまで DBMS_SQL バッファにコピーされ、実行時点で、COLUMN_VALUE コールに引数として渡された表にコピーされます。

配列用のスカラー型および LOB 型

ローカル変数を、次のいずれかの表項目型として宣言し、DBMS_SQL を使用して、任意の数の行をその中にフェッチできます （これらの表項目型は、BIND_ARRAY プロシージャに指定できる型と同じです）。

```
type Number_Table is table of Number index by binary_integer;
type Varchar2_Table is table of varchar2(2000) index by binary_integer;
type Date_Table is table of Date index by binary_integer;
type Blob_Table is table of Blob index by binary_integer;
type Clob_Table is table of Clob index by binary_integer;
type Bfile_Table is table of Bfile index by binary_integer;
```

DEFINE_ARRAY の構文

DEFINE_ARRAY プロシージャのパラメータは、表 14–6 で説明します。このプロシージャの構文は、次のとおりです。DEFINE_ARRAY プロシージャはオーバーロードされて、さまざまなデータ型を受け入れることに注意してください。

```
DBMS_SQL.DEFINE_ARRAY(
        c                 IN INTEGER,
        position          IN INTEGER,
        <table_variable>  IN <datatype>,
        count             IN INTEGER,
        indx              IN INTEGER);
```

ここで、<table_variable> とそれに対応する <datatype> は、次のいずれかの組合せになります。

```
<num_tab>      Number_Table
<vchr2_tab>    Varchar2_Table
<date_tab>     Date_Table
<blob_tab>     Blob_Table
<clob_tab>     Clob_Table
<bfile_tab>    Bfile_Table
```

このプロシージャでは、カーソル「c」の位置「position」にある列用の適切な「table」が定義されます。後続の FETCH_ROWS コールによって、「count」行がフェッチされ ます。

COLUMN_VALUE がコールされると、これらの行は位置 indx および indx+1、indx+2 などと配置されます。行を表内に挿入している間、ユーザーは FETCH_ROWS コールまたは COLUMN_VALUE コールの発行を続けます。行は、COLUMN_VALUE コールで引数として指定された表内に蓄積します。

表 14–6　DBMS_SQL.DEFINE_ARRAY プロシージャのパラメータ

パラメータ	説明
c	配列のバインド先にするカーソルの ID 番号を指定します。
name	文内の定義変数の名前を提供します。
table_variable	<datatype> として宣言されたローカル変数を指定します。
position	定義されている配列内の列の相対位置です。文の最初の列は、位置 1 を持っています。
indx	行を配置するための開始位置を提供します。

「count」はゼロより大きい整数でなければなりません。それ以外の値が指定されると、例外が発生します。「indx」は、正または負、ゼロになります。DEFINE_ARRAY コールが発行された問合せには、配列バインドを含むことはできません。

> **関連項目：** 配列の定義方法の例は、14-33 ページの「例 6 および例 7: 配列の定義」を参照してください。

DEFINE_COLUMN_LONG プロシージャ

SELECT カーソルに対して LONG 列を定義するには、このプロシージャをコールします。定義されている列は、指定されたカーソルの文の SELECT リスト内での相対位置によって識別されます。COLUMN 値のタイプによって、定義されている列のタイプが決まります。

DEFINE_COLUMN_LONG の構文

DEFINE_COLUMN_LONG のパラメータは、表 14–7 で説明します。構文は次のとおりです。

```
DBMS_SQL.DEFINE_COLUMN_LONG(
        c               IN INTEGER,
        position        IN INTEGER);
```

表 14–7　DBMS_SQL.DEFINE_COLUMN_LONG プロシージャのパラメータ

パラメータ	説明
c	選択されることが定義されている行のカーソルの ID 番号です。
position	定義されている行内の列の相対位置です。文の最初の列は、位置 1 を持っています。

EXECUTE 関数

所定のカーソルを実行するには、EXECUTE をコールします。この関数はカーソルの ID 番号を受け入れて、処理された行数を戻します。戻り値は、INSERT 文、UPDATE 文、DELETE 文に対してだけ有効です。DDL を含む、その他のタイプの文では、戻り値は不定なので、無視します。

EXECUTE の構文

EXECUTE 関数の構文は、次のとおりです。

```
DBMS_SQL.EXECUTE (
        c               IN INTEGER)
RETURN INTEGER;
```

EXECUTE_AND_FETCH 関数

指定されたカーソルを実行して行をフェッチするには、EXECUTE_AND_FETCH をコールします。この関数は、EXECUTE をコールしてから、FETCH_ROWS をコールするのと同じ機能を提供します。ただし、リモート・データベースで使用する場合には、かわりに EXECUTE_AND_FETCH をコールすると、ネットワークの往復伝搬数を低減できます。

EXECUTE_AND_FETCH の構文

EXECUTE_AND_FETCH 関数は、実際にフェッチされた行の数を戻します。このプロシージャのパラメータは、表 14–8 で説明します。構文は次のとおりです。

```
DBMS_SQL.EXECUTE_AND_FETCH(
        c               IN INTEGER,
        exact           IN BOOLEAN DEFAULT FALSE)
RETURN INTEGER;
```

表 14–8　DBMS_SQL.EXECUTE_AND_FETCH 関数のパラメータ

パラメータ	説明
c	実行してフェッチするカーソルの ID 番号を指定します。
exact	問合せに実際に一致している行の数が 1 以外の場合は、TRUE に設定して例外状況を発生させます。例外状況が発生しても、行は依然としてフェッチされ、使用できます。

FETCH_ROWS 関数

FETCH_ROWS をコールして、所定のカーソルから行をフェッチします。フェッチする行が残っている限り、繰り返し FETCH_ROWS をコールできます。これらの行は、FETCH_ROWS の各コール後に、バッファに取り出され、COLUMN_VALUE をコールすることによって、各列ごとに読み込まれなければなりません。

FETCH_ROWS の構文

FETCH_ROWS 関数は、フェッチするカーソルの ID 番号を受け入れて、実際にフェッチされた行の数を戻します。この関数の構文は、次のとおりです。

```
DBMS_SQL.FETCH_ROWS(
        c               IN INTEGER)
RETURN INTEGER;
```

COLUMN_VALUE プロシージャ

このプロシージャは、所定のカーソル内の所定の位置にあるカーソル要素の値を戻します。このプロシージャは、FETCH_ROWS をコールしてフェッチされたデータにアクセスするために使用します。

COLUMN_VALUE の構文

COLUMN_VALUE プロシージャのパラメータは、表 14–9 で説明します。このプロシージャの構文は、次のとおりです。大カッコ [ ] はオプション・パラメータを示しています。

```
DBMS_SQL.COLUMN_VALUE(
        c                 IN  INTEGER,
        position          IN  INTEGER,
        value             OUT <datatype>
       [,column_error     OUT NUMBER]
       [,actual_length    OUT INTEGER]);
```

この <datatype> は、次のいずれかのタイプでなければなりません。

```
NUMBER
DATE
MLSLABEL
VARCHAR2 CHARACTER SET ANY_CS
BLOB
CLOB CHARACTER SET ANY_CS
BFILE
```

> **関連項目：** 第 6 章「ラージ・オブジェクト (LOB)」で、BLOB および CLOB、BFILE データ型を説明しています。

COLUMN_VALUE プロシージャでは、次の構文もサポートされています。

```
DBMS_SQL.COLUMN_VALUE(
        c                 IN  INTEGER,
        position          IN  INTEGER,
        <table_variable>  IN  <datatype>);
```

ここで、<table_variable> とそれに対応する <datatype> は、次のいずれかの組合せになります。

```
<num_tab>      Number_Table
<vchr2_tab>    Varchar2_Table
<date_tab>     Date_Table
<blob_tab>     Blob_Table
<clob_tab>     Clob_Table
<bfile_tab>    Bfile_Table
```

CHAR および RAW、ROWID データを含む列には、構文で次のバリエーションを使用できます。

```
DBMS_SQL.COLUMN_VALUE_CHAR(
        c               IN  INTEGER,
        position        IN  INTEGER,
        value           OUT CHAR CHARACTER SET ANY_CS
      [,column_error    OUT NUMBER]
      [,actual_length   OUT INTEGER]);
DBMS_SQL.COLUMN_VALUE_RAW(
        c               IN  INTEGER,
        position        IN  INTEGER,
        value           OUT RAW
      [,column_error    OUT NUMBER]
      [,actual_length   OUT INTEGER]);
DBMS_SQL.COLUMN_VALUE_ROWID(
        c               IN  INTEGER,
        position        IN  INTEGER,
        value           OUT ROWID
      [,column_error    OUT NUMBER]
      [,actual_length   OUT INTEGER]);
```

表 14–9　DBMS_SQL.COLUMN_VALUE プロシージャのパラメータ

パラメータ	モード	説明
c	IN	値をフェッチしているカーソルの ID 番号を指定します。
position	IN	カーソル内の列の相対位置を指定します。文の最初の列は、位置 1 を持っています。
value	OUT	指定された列および行の値を戻します。 指定した行番号が、フェッチされた行数の合計よりも大きいと、エラー・メッセージを受け取ります。 この出力パラメータのタイプが DEFINE_COLUMN のコールによって定義された値の実際のタイプと異なると、タイプの不整合を示す例外エラー ORA-06562 が発生します。

表 14–9　DBMS_SQL.COLUMN_VALUE プロシージャのパラメータ

パラメータ	モード	説明
table_variable	IN	<datatype> として宣言されたローカル変数を指定します。
column_error	OUT	指定された列値のエラー・コードを戻します。
actual_length	OUT	指定された列内の値の（切捨て前の）実際の長さを戻します。

COLUMN_VALUE_LONG プロシージャ

このプロシージャは、所定の位置のカーソル要素の値およびオフセット、所定のカーソルのサイズを戻します。このプロシージャは、FETCH_ROWS をコールしてフェッチされたデータにアクセスするために使用します。

COLUMN_VALUE_LONG の構文

COLUMN_VALUE_LONG プロシージャのパラメータは、表 14–10 で説明します。このプロシージャの構文は、次のとおりです。

```
DBMS_SQL.COLUMN_VALUE_LONG(
        c               IN  INTEGER,
        position        IN  INTEGER,
        length          IN  INTEGER,
        offset          IN  INTEGER,
        value           OUT VARCHAR2,
        value_length    OUT INTEGER);
```

表 14–10　DBMS_SQL.COLUMN_VALUE_LONG Procedure Parameters

Parameter	Description
c	選択されることが定義されている行のカーソルの ID 番号です。
position	定義されている行内の列の相対位置です。文の最初の列は、位置 1 を持っています。
length	選択される列値のセグメントの長さ（単位はバイト）です。
offset	SELECT が開始される LONG 列のバイト位置です。
value	SELECT される列セグメントの値です。
value_length	SELECT された値の（戻される）長さです。

VARIABLE_VALUE プロシージャ

このプロシージャは、所定のカーソルに対して指定された変数の値を戻します。これは、PL/SQL 内部のバインド変数の値を戻すのにも使用されます。

VARIABLE_VALUE の構文

VARIABLE_VALUE プロシージャのパラメータは、表 14–11 で説明します。このプロシージャの構文は、次のとおりです。

```
DBMS_SQL.VARIABLE_VALUE(
        c               IN  INTEGER,
        name            IN  VARCHAR2,
        value           OUT <datatype>);
```

この <datatype> は、次のいずれかのタイプでなければなりません。

```
NUMBER
DATE
MLSLABEL
VARCHAR2 CHARACTER SET ANY_CS
BLOB
CLOB CHARACTER SET ANY_CS
BFILE
```

CHAR および RAW、ROWID データを含む変数には、構文で次のバリエーションを使用できます。

```
DBMS_SQL.VARIABLE_VALUE_CHAR(
        c               IN  INTEGER,
        name            IN  VARCHAR2,
        value           OUT CHAR CHARACTER SET ANY_CS);
DBMS_SQL.VARIABLE_VALUE_RAW(
        c               IN  INTEGER,
        name            IN  VARCHAR2,
        value           OUT RAW);
DBMS_SQL.VARIABLE_VALUE_ROWID(
        c               IN  INTEGER,
        name            IN  VARCHAR2,
        value           OUT ROWID);
```

表 14–11　DBMS_SQL.VARIABLE_VALUE プロシージャのパラメータ

パラメータ	モード	説明
c	IN	値を取得するカーソルの ID 番号を指定します。
name	IN	値を検索している変数の名前を指定します。
value	OUT	指定された位置の変数の値を戻します。 この出力パラメータのタイプが BIND_VARIABLE のコールによって定義された値の実際のタイプと異なると、タイプの不整合を示す例外エラー ORA-06562 が発生します。
position	IN	カーソル内の列の相対位置を指定します。文の最初の列は、位置 1 を持っています。

更新、挿入、削除の処理

動的 SQL を使用して INSERT、UPDATE、DELETE を処理する場合は、次のステップを実行しなければなりません。

1. EXECUTE をコールして、まず、INSERT 文、UPDATE 文、DELETE 文を実行しなければなりません。EXECUTE プロシージャは、14-19 ページで説明しています。
2. PL/SQL プロシージャのコールを含む無名ブロックを使用した場合は、VARIABLE_VALUE をコールして、これらのプロシージャの出力変数に割り当てられた値を検索します。VARIABLE_VALUE プロシージャは、14-23 ページで説明しています。

IS_OPEN 関数

指定されたカーソルが現在オープンしていると、IS_OPEN 関数は TRUE を戻します。

IS_OPEN の構文

カーソルが現在オープンしている場合は、IS_OPEN 関数はカーソルの ID 番号を受け入れて TRUE を戻し、オープンしていない場合は、FALSE を戻します。この関数の構文は、次のとおりです。

```
DBMS_SQL.IS_OPEN(
        c               IN INTEGER)
RETURN BOOLEAN;
```

DESCRIBE_COLUMNS プロシージャ

DBMS_SQL によってオープンおよび解析されたカーソルに対する列を記述するには、このプロシージャをコールします。

DESC_REC タイプ

DBMS_SQL パッケージでは、DESC_REC レコード・タイプを次のように宣言します。

```
type desc_rec is record (
        col_type            binary_integer := 0,
        col_max_len         binary_integer := 0,
        col_name            varchar2(32)   := '',
        col_name_len        binary_integer := 0,
        col_schema_name     varchar2(32)   := '',
        col_schema_name_len binary_integer := 0,
        col_precision       binary_integer := 0,
        col_scale           binary_integer := 0,
        col_charsetid       binary_integer := 0,
        col_charsetform     binary_integer := 0,
        col_null_ok         boolean        := TRUE);
```

DESC_REC のパラメータは、表 14–12 で説明します。

表 14–12　DESC_REC タイプのパラメータ

パラメータ	説明
col_type	記述されている列のタイプです。
col_max_len	列の最大長です。
col_name	列の名前です。
col_name_len	列名の長さです。
col_schema_name	列のタイプが定義された、スキーマの名前です（オブジェクト型の場合）。
col_schema_name_len	スキーマの長さです。
col_precision	数値の場合は、列の精度を示します。
col_scale	数値の場合は、列の位取りを示します。
col_charsetid	列のキャラクタ・セットの識別子です。
col_charsetform	列のキャラクタ・セットのフォームです。
col_null_ok	列が NULL になる可能性のある場合は、真です。

DESC_TAB タイプ

DESC_TAB タイプは、DESC_REC レコードの PL/SQL 表です。

```
type desc_tab is table of desc_rec index by binary_integer;
```

ローカル変数を PL/SQL 表型である DESC_TAB として宣言し、次に、DESCRIBE_COLUMNS プロシージャをコールして、表を各列の記述で埋めることができます。すべての列を記述しなければならず、単一の列だけを記述することはできません。

DESCRIBE_COLUMNS の構文

DESCRIBE_COLUMNS のパラメータは、表 14–13 で説明します。構文は次のとおりです。

```
DBMS_SQL.DESCRIBE_COLUMNS(
        c               IN  INTEGER,
        col_cnt         OUT INTEGER,
        desc_t          OUT DESC_TAB);
```

表 14–13　DBMS_SQL.DESCRIBE_COLUMNS プロシージャのパラメータ

パラメータ	モード	説明
c	IN	記述されている列のカーソルの ID 番号です。
col_cnt	OUT	
desc_t	OUT	

関連項目： DESCRIBE_COLUMNS の使用方法の例は、14-35 ページの「例 8: 列の記述」を参照してください。

CLOSE_CURSOR プロシージャ

所定のカーソルをクローズするには、CLOSE_CURSOR をコールします。

CLOSE_CURSOR の構文

CLOSE_CURSOR プロシージャのパラメータは、表 14–14 で説明します。このプロシージャの構文は、次のとおりです。

```
DBMS_SQL.CLOSE_CURSOR(
        c               IN OUT INTEGER);
```

表 14–14　DBMS_SQL.CLOSE_CURSOR プロシージャのパラメータ

パラメータ	モード	説明
C	IN	クローズするカーソルの ID 番号を指定します。
	OUT	カーソルは NULL に設定されています。CLOSE_CURSOR を コールした後、カーソルに割り振られたメモリーは 解放され、そのカーソルからはフェッチできなくなります。

エラーの位置

DBMS_SQL パッケージには、セッションで最後に参照されたカーソルについての情報を取得するための関数がいくつかあります。これらの関数によって戻される値は、SQL 文の実行直後にだけ意味を持ちます。また、エラー切分け用関数は、特定の DBMS_SQL コール後にだけ意味を持ちます。たとえば、PARSE 直後に LAST_ERROR_POSITION をコールします。

LAST_ERROR_POSITION 関数

エラーが発生した SQL 文テキスト内のバイト・オフセットを戻します。SQL 文内の最初の文字は、位置 0 にあります。

```
DBMS_SQL.LAST_ERROR_POSITION RETURN INTEGER;
```

その他の DBMS_SQL プロシージャまたは関数をコールする前に、PARSE をコールした後で、この関数をコールします。

LAST_ROW_COUNT 関数

フェッチされた行数の累積を戻します。

```
DBMS_SQL.LAST_ROW_COUNT RETURN INTEGER;
```

FETCH_ROWS コールまたは EXECUTE_AND_FETCH コール後に、この関数をコールします。EXECUTE コール後にこの関数がコールされた場合には、戻される値はゼロになります。

LAST_ROW_ID 関数

処理された最後の行の ROWID を戻します。

```
DBMS_SQL.LAST_ROW_ID RETURN ROWID;
```

FETCH_ROWS コールまたは EXECUTE_AND_FETCH コール後に、この関数をコールします。

LAST_SQL_FUNCTION_CODE 関数

文の SQL 機能コードを戻します。これらのコードは、『Oracle コール・インタフェース・プログラマーズ・ガイド』に一覧表示されています。

```
DBMS_SQL.LAST_SQL_FUNCTION_CODE RETURN INTEGER;
```

この関数は、SQL 文の実行直後にコールしなければなりません。そうしないと、戻り値は不定になります。

DBMS_SQL の使用例

この項には、DBMS_SQL パッケージを使用するプロシージャの例が記述されています。

例 1 次のプロシージャの例は、SQL 文に渡され、次にその SQL 文を解析して実行します。

```
CREATE OR REPLACE PROCEDURE exec(STRING IN varchar2) AS
    cursor_name INTEGER;
    ret INTEGER;
BEGIN
   cursor_name := DBMS_SQL.OPEN_CURSOR;

   --DDL statements are executed by the parse call, which
   --performs the implied commit
   DBMS_SQL.PARSE(cursor_name, string, DBMS_SQL);
   ret := DBMS_SQL.EXECUTE(cursor_name);
   DBMS_SQL.CLOSE_CURSOR(cursor_name);
END;
```

このようなプロシージャを作成すると、次の操作が行えます。

- コール側のプログラムによって、実行時に SQL 文を動的に生成できる。
- SQL 文を DDL 文にできる。

たとえば、このプロシージャを作成した後、次のコールを行うことができます。

```
exec('create table acct(c1 integer)');
```

次の例に示されているように、このプロシージャはリモートでコールすることも できます。これによって、リモート DDL を実行できます。

```
exec@hq.com('CREATE TABLE acct(c1 INTEGER)');
```

例 2 次のプロシージャの例は、ソース表と宛先表の名前が渡され、ソース表から宛先表へ行をコピーします。このプロシージャの例は、ソース表と宛先表にはどちらも次の列があることを前提としています。

```
ID of type NUMBER
NAME of type VARCHAR2(30)
BIRTHDATE of type DATE
```

このプロシージャには、動的 SQL を使用する必要は特にありません。しかし、このパッケージの概念をわかりやすく説明しています。

```
CREATE OR REPLACE PROCEDURE copy(source      IN VARCHAR2,
                                 destination IN VARCHAR2) is

-- This procedure copies rows from a given source table to a
-- given destination table assuming that both source and
-- destination tables have the following columns:
--   - ID of type NUMBER,
--   - NAME of type VARCHAR2(30),
--   - BIRTHDATE of type DATE.
```

```
  id                 NUMBER;
  name               VARCHAR2(30);
  birthdate          DATE;
  source_cursor      INTEGER;
  destination_cursor INTEGER;
  ignore             INTEGER;
BEGIN

  -- prepare a cursor to select from the source table
  source_cursor := dbms_sql.open_cursor;
  DBMS_SQL.PARSE(source_cursor,
       'SELECT id, name, birthdate FROM ' || source,
        DBMS_SQL);
  DBMS_SQL.DEFINE_COLUMN(source_cursor, 1, id);
  DBMS_SQL.DEFINE_COLUMN(source_cursor, 2, name, 30);
  DBMS_SQL.DEFINE_COLUMN(source_cursor, 3, birthdate);
  ignore := DBMS_SQL.EXECUTE(source_cursor);

  -- prepare a cursor to insert into the destination table
  destination_cursor := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(destination_cursor,
                 'INSERT INTO ' || destination ||
                 ' VALUES (:id, :name, :birthdate)',
                  DBMS_SQL);

  -- fetch a row from the source table and
  -- insert it into the destination table
  LOOP
    IF DBMS_SQL.FETCH_ROWS(source_cursor)>0 THEN
      -- get column values of the row
      DBMS_SQL.COLUMN_VALUE(source_cursor, 1, id);
      DBMS_SQL.COLUMN_VALUE(source_cursor, 2, name);
      DBMS_SQL.COLUMN_VALUE(source_cursor, 3, birthdate);

      -- bind the row into the cursor that inserts into the
      -- destination table
      -- You could alter this example to require the use of
      -- dynamic SQL by inserting an if condition before the
      -- bind.
      DBMS_SQL.BIND_VARIABLE(destination_cursor, 'id', id);
      DBMS_SQL.BIND_VARIABLE(destination_cursor, 'name', name);
      DBMS_SQL.BIND_VARIABLE(destination_cursor, 'birthdate',
                             birthdate);
      ignore := DBMS_SQL.EXECUTE(destination_cursor);
    ELSE

    -- no more row to copy
```

```
      EXIT;
    END IF;
  END LOOP;

  -- commit and close all cursors
  COMMIT;
  DBMS_SQL.CLOSE_CURSOR(source_cursor);
  DBMS_SQL.CLOSE_CURSOR(destination_cursor);

EXCEPTION
  WHEN OTHERS THEN
    IF DBMS_SQL.IS_OPEN(source_cursor) THEN
      DBMS_SQL.CLOSE_CURSOR(source_cursor);
    END IF;
    IF DBMS_SQL.IS_OPEN(destination_cursor) THEN
      DBMS_SQL.CLOSE_CURSOR(destination_cursor);
    END IF;
    RAISE;
END;
```

例 3 および例 4、例 5: バルク DML 次の一連の例では、DELETE および INSERT、UPDATE の各 SQL DML 文における、バルク配列バインド（表項目）の使用方法を示します。

たとえば、DELETE 文では、WHERE 句に配列をバインドし、配列内の要素ごとに文を実行できます。

```
declare
  stmt varchar2(200);
  dept_no_array dbms_sql.Number_Table;
  c number;
  dummy number;
begin
  dept_no_array(1) := 10; dept_no_array(2) := 20;
  dept_no_array(3) := 30; dept_no_array(4) := 40;
  dept_no_array(5) := 30; dept_no_array(6) := 40;
  stmt := 'delete from emp where deptno = :dept_array';
  c := dbms_sql.open_cursor;
  dbms_sql.parse(c, stmt, dbms_sql.native);
  dbms_sql.bind_array(c, ':dept_array', dept_no_array, 1, 4);
  dummy := dbms_sql.execute(c);
  dbms_sql.close_cursor(c);

  exception when others then
    if dbms_sql.is_open(c) then
      dbms_sql.close_cursor(c);
    end if;
    raise;
```

```
end;
/
```

上記の例では、1 から 4 までの要素だけが、bind_array コールで指定されたとおりに使用されます。配列の各要素は、大量の従業員をデータベースから削除する可能性があります。

バルク INSERT 文の例を、次に示します。

```
declare
  stmt varchar2(200);
  empno_array dbms_sql.Number_Table;
  empname_array dbms_sql.Varchar2_Table;
  c number;
  dummy number;
begin
  for i in 0..9 loop
    empno_array(i) := 1000 + i;
    empname_array(I) := get_name(i);
  end loop;
  stmt := 'insert into emp values(:num_array, :name_array)';
  c := dbms_sql.open_cursor;
  dbms_sql.parse(c, stmt, dbms_sql.native);
  dbms_sql.bind_array(c, ':num_array', empno_array);
  dbms_sql.bind_array(c, ':name_array', empname_array);
  dummy := dbms_sql.execute(c);
  dbms_sql.close_cursor(c);

  exception when others then
    if dbms_sql.is_open(c) then
      dbms_sql.close_cursor(c);
    end if;
    raise;
end;
/
```

実行時には、10 人の従業員はすべて表に挿入されます。

最後に、バルク UPDATE 文の例を示します。

```
declare
  stmt varchar2(200);
  emp_no_array dbms_sql.Number_Table;
  emp_addr_array dbms_sql.Varchar2_Table;
  c number;
  dummy number;
begin
  for i in 0..9 loop
    emp_no_array(i) := 1000 + i;
    emp_addr_array(I) := get_new_addr(i);
```

```
  end loop;
  stmt := 'update emp set ename = :name_array
    where empno = :num_array';
  c := dbms_sql.open_cursor;
  dbms_sql.parse(c, stmt, dbms_sql.native);
  dbms_sql.bind_array(c, ':num_array', empno_array);
  dbms_sql.bind_array(c, ':name_array', empname_array);
  dummy := dbms_sql.execute(c);
  dbms_sql.close_cursor(c);

  exception when others then
    if dbms_sql.is_open(c) then
      dbms_sql.close_cursor(c);
    end if;
    raise;
end;
/
```

実行がコールされると、全従業員のアドレスが一度に更新されます 。常に 2 つの配列が同時にステップされます。WHERE 句で 2 つ以上の行が戻される場合、これら全従業員は、そのときに「addr_array」が参照するアドレスを得ます。

例 6 および例 7: 配列の定義 DEFINE_ARRAY プロシージャの使用方法の例を、次に示します。

```
declare
  c       number;
  d       number;
  n_tab   dbms_sql.Number_Table;
  indx    number := -10;
begin
  c := dbms_sql.open_cursor;
  dbms_sql.parse(c, 'select n from t order by 1', dbms_sql);

  dbms_sql.define_array(c, 1, n_tab, 10, indx);

  d := dbms_sql.execute(c);
  loop
    d := dbms_sql.fetch_rows(c);

    dbms_sql.column_value(c, 1, n_tab);

    exit when d != 10;
  end loop;

  dbms_sql.close_cursor(c);
```

```
  exception when others then
    if dbms_sql.is_open(c) then
      dbms_sql.close_cursor(c);
    end if;
    raise;
end;
/
```

上記の例で FETCH_ROWS をコールするたびに、DBMS_SQL バッファに保持されている 10 行がフェッチされます。COLUMN_VALUE コールが実行されると、それらの行は、指定された PL/SQL 表（この場合は n_tab）に移動し、DEFINE 文で指定されたとおり、-10 から -1 の位置に収まります。2 番目のバッチがループ内でフェッチされると、行は 0 から 9 の位置に移動するなどです。

各配列への現行の索引は、自動的にメンテナンスされます。この索引は、EXECUTE 時に「indx」に初期化され、COLUMN_VALUE がコールされるたびに更新されます。任意の時点で再実行した場合、各 DEFINE の現行の索引は「indx」に再初期化されます。

このように、問合せのすべての結果が、表内にフェッチされます。FETCH_ROWS で 10 行をフェッチできないときは、実際にフェッチした行の数を戻し（行をまったくフェッチできなかった場合はゼロを戻す）、ループを終了します。

DEFINE_ARRAY プロシージャの別の使用例を、次に示します。

次のように定義された MULTI_TAB 表について、考えてみます。

```
create table multi_tab (num number,
                        dat1 date,
                        var varchar2(24),
                        dat2 date)
```

この表からすべてを選択し、4 つの PL/SQL 表に移動させるには、次の簡単なプログラムを使用できます。

```
declare
  c       number;
  d       number;
  n_tab  dbms_sql.Number_Table;
  d_tab1 dbms_sql.Date_Table;
  v_tab  dbms_sql.Varchar2_Table;
  d_tab2 dbms_sql.Date_Table;
  indx number := 10;
begin

  c := dbms_sql.open_cursor;
  dbms_sql.parse(c, 'select * from multi_tab order by 1', dbms_sql);

  dbms_sql.define_array(c, 1, n_tab,  5, indx);
  dbms_sql.define_array(c, 2, d_tab1, 5, indx);
```

```
  dbms_sql.define_array(c, 3, v_tab,  5, indx);
  dbms_sql.define_array(c, 4, d_tab2, 5, indx);

  d := dbms_sql.execute(c);

  loop
    d := dbms_sql.fetch_rows(c);

    dbms_sql.column_value(c, 1, n_tab);
    dbms_sql.column_value(c, 2, d_tab1);
    dbms_sql.column_value(c, 3, v_tab);
    dbms_sql.column_value(c, 4, d_tab2);

    exit when d != 5;
  end loop;

  dbms_sql.close_cursor(c);

/*
```

これで 4 つの表はどんな用途にも使用できます。使用方法の 1 つは、'INSERT into SOME_T values(:a, :b, :c, :d); のような問合せによって、BIND_ARRAY を使用して行を他の表に移動することです。

```
*/

exception when others then
    if dbms_sql.is_open(c) then
      dbms_sql.close_cursor(c);
    end if;
    raise;
end;
/
```

例 8: 列の記述 この例は、記述する表に対する SELECT * による問合せを使用することによって、SQL*Plus の DESCRIBE コールのかわりに使用できます。

```
declare
  c number;
  d number;
  col_cnt integer;
  f boolean;
  rec_tab dbms_sql.desc_tab;
  col_num number;
  procedure print_rec(rec in dbms_sql.desc_rec) is
  begin
    dbms_output.new_line;
```

```
    dbms_output.put_line('col_type            =    '
                         || rec.col_type);
    dbms_output.put_line('col_maxlen          =    '
                         || rec.col_max_len);
    dbms_output.put_line('col_name            =    '
                         || rec.col_name);
    dbms_output.put_line('col_name_len        =    '
                         || rec.col_name_len);
    dbms_output.put_line('col_schema_name     =    '
                         || rec.col_schema_name);
    dbms_output.put_line('col_schema_name_len =    '
                         || rec.col_schema_name_len);
    dbms_output.put_line('col_precision       =    '
                         || rec.col_precision);
    dbms_output.put_line('col_scale           =    '
                         || rec.col_scale);
    dbms_output.put('col_null_ok         =    ');
    if (rec.col_null_ok) then
      dbms_output.put_line('true');
    else
      dbms_output.put_line('false');
    end if;
  end;
begin
  c := dbms_sql.open_cursor;

  dbms_sql.parse(c, 'select * from scott.bonus', dbms_sql);

  d := dbms_sql.execute(c);

  dbms_sql.describe_columns(c, col_cnt, rec_tab);

/*
 * 次のループは単に for j in 1..col_cnt ループです。
 * ここでは、PL/SQL 表のいくつかの機能について
 * 簡単に示しています。
 */
  col_num := rec_tab.first;
  if (col_num is not null) then
    loop
      print_rec(rec_tab(col_num));
      col_num := rec_tab.next(col_num);
      exit when (col_num is null);
    end loop;
  end if;

  dbms_sql.close_cursor(c);
```

```
end;
/
```

15

スキーマ・オブジェクト間の依存性

ビューおよびプロシージャなどの特定のスキーマ・オブジェクトの定義では、表などの他のスキーマ・オブジェクトを参照します。つまり、あるスキーマ・オブジェクトはその定義が参照する他のオブジェクトに依存します。この章では、スキーマ・オブジェクト間の依存性を管理する方法について説明します。トピックは次のとおりです。

- 依存性の問題
- 手動で再コンパイルする
- 依存性の管理情報をリストする

関連項目： Trusted Oracle を使用している場合、Trusted Oracle でのスキーマ・オブジェクト間の依存性の処理方法は、Trusted Oracle のマニュアルも参照してください。

依存性の問題

Oracle は、ストアド・プロシージャや関数が作成されるとき、その処理によってアクセスされるスキーマ・オブジェクトが利用可能であるか検証します。たとえば、ストアド・プロシージャに表から列を選択する SELECT 文が含まれている場合、Oracle はその表が存在し、指定された列が含まれていることを検証します。表が後から再定義されて、列の 1 つが存在していない場合、ストアド・プロシージャは適切に動作しません。このような理由で、ストアド・プロシージャは表に依存すると言います。

このような場合、Oracle では、スキーマ・オブジェクト間の依存性が自動的に管理されます。スキーマ・オブジェクトが再定義されると、その後、再定義されたオブジェクトに依存するデータベース内のストアド・プロシージャと関数がコールされたときに、Oracle はそれらを自動的に再コンパイルします。この再コンパイルによって、Oracle はプロシージャや関数が新たに定義されたオブジェクトに基づいて適切に実行できるか検証します。

実行時の再コンパイルを避ける

実行時の再コンパイルは、実行時のパフォーマンスを低下させます。実行時に再コンパイルして、コンパイル・エラーが発生すると、アプリケーションが停止する場合があります。実行時の再コンパイルを避けるために、次の方法に従ってください。

- 本番アプリケーションの実行中に、スキーマ・オブジェクト（表、ビュー、ストアド・プロシージャ、ストアド関数など）を再定義しないでください。オブジェクトを再定義すると、Oracle ではそれらのオブジェクトに依存するストアド・プロシージャと関数を再コンパイルします。

- スキーマ・オブジェクトを再定義したあと、依存するストアド・プロシージャ、ストアド関数、パッケージを手動で再コンパイルしてください。これによって、実行時の再コンパイルによるパフォーマンスの影響をなくすだけでなく、コンパイル・エラーをすぐに発見し、本番使用前にエラーを訂正することもできます。

  ALTER PROCEDURE または ALTER FUNCTION、ALTER PACKAGE コマンドの COMPILE オプションによって、ストアド・プロシージャまたはストアド関数、パッケージを手動で再コンパイルできます。

  関連項目： これらのコマンドの詳細は、『Oracle8 Server SQL リファレンス』を参照してください。

  SQL スクリプト UTLDTREE.SQL を実行して、データベース内のスキーマ・オブジェクト間の依存性を判断できます。

  関連項目： UTLDTREE.SQL スクリプトの正確な名前および位置は、オペレーティング・システムによって異なります。このスクリプトの使用方法については、スクリプト自体を調べてください。

- プロシージャおよび関数は、できる限りパッケージに格納するようにします。プロシージャや関数をパッケージに格納すれば、その定義を修正しても、その他のプロシージャや関数が Oracle によって再コンパイルされることはありません。

プロシージャまたはパッケージを削除する前に検討すべき依存性の問題がいくつかあります。依存性の問題に関する詳細は、『Oracle8 Server 概要』に説明されています。依存性を管理するためのガイドラインを次に示します。

使用可能なときのパッケージの使用 パッケージは、不要な依存性検査が行われるのを防ぐ最も効果的な方法です。次の例でその利点を示します。

次のような状態を想定します。

- スタンドアロン・プロシージャ PROC は、パッケージ・プロシージャ PACK_PROC に依存する。
- PACK_PROC プロシージャの定義は、パッケージ本体を再コンパイルすることによって変更される。
- PACK_PROC プロシージャ仕様部を、パッケージ仕様部で変更することはできない。

パッケージ本体を再コンパイルしている場合でも、パッケージ仕様部が変更されていないため、パッケージ・プロシージャ PACK_PROC に依存するスタンドアロン・プロシージャ PROC は無効となりません。

この方法は、分散環境において特に有効です。プロシージャが常にパッケージの一部であれば、パッケージド・プロシージャに依存するリモート・プロシージャは、パッケージ仕様部を置き換えるか、または無効に設定しない限り、無効とはなりません。

プロシージャを再コンパイルするときには常に、他のデータベース管理者とアプリケーション開発者に相談して、リモートの依存プロシージャを識別し、それらも再コンパイルされることを保証する必要があります。こうすることによって、実行時の再コンパイルを避け、アプリケーション・ユーザーが見ることになったかもしれないコンパイル時のエラーを検出することができます。

> **関連項目：** 詳細は、15-4 ページの「手動で再コンパイルする」を参照してください。

%TYPE 属性および %ROWTYPE 属性 %TYPE 属性は変数、定数、または列のデータ型を指定します。この属性は、データベース表上の列を参照する変数またはプロシージャ引数の宣言に特に便利です。%ROWTYPE 属性は、表上の行またはビュー、あるいはカーソルの取出しによって戻される行と同じ要素を持つレコードに変数を宣言する場合に、特に便利です。

%TYPE と %ROWTYPE を使用して要素を宣言する場合は、表の列または要素のデータ型がわかっている必要はありません。たとえば、EMP 表に行を挿入するプロシージャの引数リストは、次のように宣言できます。

```
CREATE PROCEDURE hire_fire(emp_record emp%ROWTYPE) AS ... END;
```

列または表要素を変更すると、定義されているデータ型または要素も自動的に変更されます。

しかし、ある依存性のタイプは %TYPE または %ROWTYPE を使用して排除されますが、また別のタイプの依存性が作成されます。object%TYPE または object%ROWTYPE を使用して要素を定義する場合、その要素は object に依存します。object を変更すると、その object に依存する要素は無効となります。

リモート依存性

PL/SQL ライブラリ・ユニット（パッケージおよびストアド・プロシージャ、ストアド関数）間の依存性は、タイムスタンプまたは署名のいずれかによって対処できます。

- タイムスタンプ・メソッドでは、サーバーは、各ライブラリ・ユニットの作成時または再コンパイル時に、タイムスタンプを設定し、コンパイルされた状態にあるその依存ライブラリ・ユニットには、そのタイムスタンプのレコードが含まれます。親ユニットまたは関連するスキーマ・オブジェクトを変更した場合、すべての依存ユニットに無効のマークが付けられるため、再コンパイルしないかぎり、実行できません。
- 署名メソッドでは、コンパイル済みの各ストアド・ライブラリ・ユニットは、その名前およびそのパラメータのタイプおよびモード、パラメータ数、戻り値のタイプ（関数の場合）を識別する署名と対応付けられます。その署名が矛盾するように変更された親ユニットを、依存するユニットがコールする場合に、依存するユニットは無効とマークされます。

関連項目： 詳細は、10-14 ページの「リモート依存性」を参照してください。

手動で再コンパイルする

Oracle は、無効なビューまたは PL/SQL プログラム・ユニットを、それらが次回使用されるときに、動的に再コンパイルします。あるいは、SQL コマンドに COMPILE パラメータを指定して、無効なビューまたはプログラム・ユニットを強制的にコンパイルすることも可能です。

強制コンパイルは、依存ビューまたはプログラム・ユニットが無効であることがわかっており、現在使用していない場合に、エラーを検査する目的で指定します。したがって、ビューまたはプログラム・ユニットが実行されないかぎり、自動再コンパイルは実行されません。

無効な依存オブジェクトは、データ・ディクショナリ・ビュー USER_OBJECTS および ALL_OBJECTS、DBA_OBJECTS を問い合せることによって識別できます。例は、15-6 ページの「依存性の管理情報をリストする」を参照してください。

ビューを手動で再コンパイルする

ビューを再コンパイルするには、COMPILE パラメータを指定して ALTER VIEW コマンドを使用します。次の文は、自分のスキーマ内の EMP_DEPT ビューを再コンパイルします。

```
ALTER VIEW emp_dept COMPILE;
```

ビューを再コンパイルするために必要な権限

ビューを手動で再コンパイルするには、ビューがユーザーのスキーマ内にあるか、または ALTER ANY TABLE システム権限が必要です。

プロシージャと関数を手動で再コンパイルする

プロシージャまたは関数（スタンドアロン）を再コンパイルするには、COMPILE 句を指定して ALTER PROCEDURE または ALTER FUNCTION コマンドを使用します。たとえば、次の文は、自分のスキーマ内のストアド・プロシージャ UPDATE_SALARY を再コンパイルします。

```
ALTER PROCEDURE update_salary COMPILE;
```

パッケージを手動で再コンパイルする

パッケージ本体またはパッケージ本体と仕様部の両方を再コンパイルするには、COMPILE パラメータを指定して ALTER PACKAGE コマンドを使用します。たとえば、次の SQL*Plus 文は、ACCT_MGMT_PACKAGE パッケージの本体だけ、および本体と仕様部をそれぞれ再コンパイルします。

```
SQLPLUS> ALTER PACKAGE acct_mgmt_package COMPILE BODY;
SQLPLUS> ALTER PACKAGE acct_mgmt_package COMPILE PACKAGE;
```

すべてのパッケージ、プロシージャ、関数は、次の構文を使用して再コンパイルできます。オブジェクトは依存性の順序でコンパイルされ、それぞれ 1 度だけコンパイルされるようにします。

```
SQLPLUS> EXECUTE DBMS_UTILITY.COMPILE_ALL;
```

プロシージャまたはページを再コンパイルするために必要な権限

プロシージャまたはパッケージがユーザーのスキーマに含まれる場合および ALTER ANY PROCEDURE システム権限を持っている場合にだけ、プロシージャまたはパッケージを手動で再コンパイルできます。

トリガーを手動で再コンパイルする

既存のトリガー（使用可能状態または使用禁止状態）は、ALTER TRIGGER コマンドを使用して手動で再コンパイルできます。たとえばトリガー REORDER を強制的にコンパイルするには、次のように指定します。

```
ALTER TRIGGER reorder COMPILE;
```

トリガーを再コンパイルするために必要な権限

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。

依存性の管理情報をリストする

次のデータ・ディクショナリ・ビューには、直接依存性と依存性管理の情報がリストされます。

- USER_DEPENDENCIES, ALL_DEPENDENCIES, DBA_DEPENDENCIES
- USER_OBJECTS, ALL_OBJECTS, DBA_OBJECTS

関連項目： これらのデータ・ディクショナリ・ビューの詳細は、『Oracle8 Server リファレンス・マニュアル』を参照してください。

例 1 と例 2 では、次の文を考えます。

```
CREATE TABLE emp . . .;

CREATE PROCEDURE hire_emp BEGIN . . . END;

ALTER TABLE emp . . . ;
```

例 1: オブジェクトの状態をリストする ALL_OBJECTS データ・ディクショナリ・ビューは、現行ユーザーに利用可能なすべてのオブジェクトに関する情報と、その現在の状態（つまり有効または無効）をリストします。たとえば、次の問合せは、現行ユーザーに利用可能なすべてのオブジェクトの名前、タイプ、現在の状態をリストします。

```
SELECT object_name, object_type, status
    FROM all_objects;
```

次のような結果が戻されます。

```
OBJECT_NAME     OBJECT_TYPE     STATUS
--------------- --------------- ------------
EMP             TABLE           VALID
HIRE_EMP        PROCEDURE       INVALID
```

例 2: 依存性をリストする DBA_DEPENDENCIES データ・ディクショナリ・ビューには、データベース中のすべての依存オブジェクトと、そのオブジェクトが直接依存するオブジェクトがリストされています。たとえば、次の問合せは、JWARD スキーマ内の依存オブジェクトをすべてリストします。

```
SELECT name, type, referenced_name, referenced_type
    FROM sys.dba_dependencies
    WHERE owner = 'JWARD';
```

この項のはじめに示した文を JWARD が発行した場合には、次のような結果が戻されます。

```
NAME        TYPE        REFERENCED_NAME     REFERENCED_TYPE
----------- ----------- ------------------- ---------------
HIRE_EMP    PROCEDURE   EMP                 TABLE
```

依存性追跡ユーティリティ

*_DEPENDENCIES データ・ディクショナリ・ビューには、オブジェクトの直接依存性だけが一覧表示されます。依存性の追跡ユーティリティを使用して、オブジェクトの直接、間接の依存性をリストできます。

依存性の追跡ユーティリティを作成するには、SQL スクリプトの UTLDTREE.SQL を実行しなければなりません。このファイルの位置は、オペレーティング・システムによって異なります。UTLDTREE.SQL スクリプトによって、次のスキーマ・オブジェクトが作成されます。

表 DEPTREE_TEMPTAB DEPTREE_FILL プロシージャによって戻される依存情報を格納するための一時表として使用されます。

構造：

```
object_id NUMBER
referenced_object_id NUMBER
nest_level NUMBER
seq# NUMBER
```

ビュー DEPTREE DEPTREE_TEMPTAB 表の依存情報をリストするためのビューです。親オブジェクトはゼロの NESTED_LEVEL にリストされ、依存オブジェクトはゼロ以上のネスト・レベルにリストされます。

列名： `nested_level, object_type, owner, object_name, seq#`

ビュー IDEPTREE DEPTREE_TEMPTAB 表の依存情報をリストするためのビューです。オブジェクトの依存関係をインデントを付けて示し、一目でわかる形式で出力されます。

列名： `dependencies`

順序 DEPTREE_SEQ DEPTREE_TEMPTAB 表に格納される依存情報の一意のセットを識別するための順序です。

プロシージャ DEPTREE_FILL 文を実行するユーザーのスキーマ内の DEPTREE_TEMPTAB 表を最初にクリアしてから、指定したオブジェクトに直接的または間接的に依存する（つまり参照する）オブジェクトで、クリアした表を満たすプロシージャです。指定したオブジェクトを再帰的に参照するオブジェクトがすべて一覧表示されますが、このときユーザーには、その存在を確認する権限が付与されている必要があります。

構文： `DEPTREE_FILL(object_type CHAR, object_owner CHAR, object_name CHAR)`

INTERNAL として接続されているときに UTLDTREE を使用する

INTERNAL として接続して、UTLDTREE.SQL スクリプトを実行し、ユーティリティを使用する場合、依存オブジェクトだけではなく、依存カーソル（共有 SQL 領域）の依存情報も収集され、表示されます。

例 次の SQL 文は、UTLDTREE ユーティリティを使用してオブジェクトの依存性を追跡する方法です。次の SQL 文を想定します。

```
CONNECT scott/tiger;
CREATE TABLE scott.emp ( .... );
CREATE SEQUENCE scott.emp_sequence;
CREATE VIEW scott.sales_employees AS
   SELECT * FROM scott.emp WHERE deptno = 10;
CREATE PROCEDURE scott.hire_salesperson (name VARCHAR2,
   job VARCHAR2, mgr NUMBER, hiredate DATE, sal NUMBER,
   comm NUMBER)
IS
BEGIN
   INSERT INTO scott.sales_employees
      VALUES (scott.emp_sequence.NEXTVAL, name, job, mgr,
```

```
        hiredate, sal, comm, 10;
END;
CREATE PROCEDURE scott.fire_salesperson (emp_id NUMBER) IS
BEGIN
   DELETE FROM scott.sales_employees WHERE empno = emp_id;
END;
SELECT * FROM scott.emp;
SELECT * FROM scott.sales_employees;
EXECUTE scott.hire_salesperson ('ARNALL', 'MANAGER', 7839, /
                                SYSDATE, 1000, 500);
EXECUTE scott.fire_salesperson (7934);
```

SCOTT が EMP 表を変更する前に、EMP 表を変更することによって無効となる、すべての依存オブジェクトを確認するとします。次のプロシージャの実行によって、EMP 表に関する依存性の情報によって DEPTREE_TEMPTAB 表が満たされます（Enterprise Manager を使用して実行）。

```
EXECUTE deptree_fill('TABLE', 'SCOTT', 'EMP');
```

次の 2 つの問合せによって、それまでに収集された EMP 表の依存情報が示されます。

```
SELECT * FROM deptree;
NESTED_LEV TYPE        OWNER          NAME                SEQ#
---------- ---------- -------------- ------------------ ----
         0 TABLE      SCOTT          EMP                   0
         1 VIEW       SCOTT          SALES_EMPLOYEES       1
         2 PROCEDURE  SCOTT          FIRE_SALESPERSON      2
         2 PROCEDURE  SCOTT          HIRE_SALESPERSON      3
SELECT * FROM ideptree;
DEPENDENCIES
----------------------------------------
TABLE SCOTT.EMP
VIEW SCOTT.SALES_EMPLOYEES
PROCEDURE SCOTT.FIRE_SALESPERSON
PROCEDURE SCOTT.HIRE_SALESPERSON
```

また、UTLDTREE ユーティリティを使用して、EMP 表（共有プール中の、依存共有 SQL 領域）に依存するすべてのカーソルをリストできます。INTERNAL として接続し、SCOTT.EMP 表の依存情報を収集した後、次の 2 つの問合せを発行します。

```
SELECT * FROM deptree;
NESTED_LEV TYPE   OWNER    NAME                          SEQ#
---------- ------ -------- ---------------------------- ----
         0 TABLE  SCOTT    EMP                             0
         1 CURSOR <shared> "select * from scott.emp      0.5
         2 CURSOR <shared> "select * from scott.sa. . .  7.5
         3 CURSOR <shared> "BEGIN hire_salesperson. . .  9.5
         3 CURSOR <shared> "BEGIN fire_salesperson. . .  8.5
```

```
SELECT * FROM ideptree;
DEPENDENCIES
------------------------------------------------------------
TABLE STEVE.EMP
   CURSOR <shared>."select * from scott.emp"
   CURSOR <shared>."select * from scott.sales_employee"
   CURSOR <shared>."BEGIN hire_salesperson ('ARN. . .
   CURSOR <shared>."BEGIN fire_salesperson (7934) END"
```

16

アラート機能によるデータベースのイベントの通知

この章では、DBMS_ALERT パッケージを使用して、データベースのイベントの通知、または「アラート」の方法を説明します。トピックは次のとおりです。

- 概要
- アラートの使用
- アラートの検査
- アラートの使用例

概要

DBMS_ALERT パッケージでは、非同期（ポーリングとは対照的に）のデータベース・イベントの通知をサポートできます。このパッケージとデータベース・トリガーを適切に使用することによって、データベースで特定の値が変更されるたびにアプリケーション自身にそのことを通知できます。

たとえば、グラフィック・ツールがデータベース表のデータをグラフ表示する場合を想定します。データを読み込んでグラフ表示したあと、グラフィック・ツールにその読み込んだデータを対象とするデータベース・アラート（DBMS_ALERT.WAITONE）を待機させることができます。他のユーザーがそのデータを変更すると、ツールが自動的に起動します。起動されると必ず信号（DBMS_ALERT.SIGNAL）を送信するトリガーをデータベース表に設定するだけでこのようなことが可能となります。

アラートはトランザクションを単位とします。つまり、待機中のセッションは、アラートの信号を送信するトランザクションがコミットされるまでアラートを受け取りません。

特定のアラートについて同時実行の送信と待機がいくつあってもかまいません。

待機中のアプリケーションはデータベースでの実行を中断され、他の作業を行えません。

注意： データベースのアラート機能は COMMITS を発行するため、Oracle Forms では使えません。Oracle Forms（ロムフォード）がアクティブな状態でストアド・プロシージャをコールする際の制限は、Oracle Forms 関連マニュアルを参照してください。

次のプロシージャは、DBMS_ALERT パッケージからコールできます。

表 16–1　DBMS_ALERT パッケージの関数とプロシージャ

関数 / プロシージャ	説明	参照ページ
REGISTER	アラートからメッセージを受信する	16-5 ページ
REMOVE	アラートからの通知を使用禁止にする	16-5 ページ
SIGNAL	アラートを通知する（登録されたセッションにメッセージを送信する）	16-5 ページ
WAITANY	TIMEOUT（単位は秒）を待って、セッションに登録されているアラートからの警告メッセージを受け取る	16-6 ページ
WAITONE	TIMEOUT（単位は秒）を待って、名前付きのアラートからのメッセージを受け取る	16-7 ページ
SET_DEFAULTS	ポーリング間隔を設定する	16-8 ページ

DBMS_ALERT パッケージを作成する

DBMS_ALERT パッケージを作成するには、ユーザー SYS として接続し、DBMSALRT.SQL と PRVTALRT.PLB を実行します。これらのスクリプトは、CATPROC.SQL スクリプトによって自動的に実行されます。

関連項目：　このパッケージを実行するユーザーへの必要な権限の付与については、10-56 ページの「必要な権限」を参照してください。

セキュリティ

このパッケージのセキュリティは、選ばれたユーザーやロールにこのパッケージの EXECUTE 権限を付与することで制御できます。使用されるアラート名を制限するカバー・パッケージをこのパッケージの先頭に作成するとよいでしょう。このパッケージではなく、このカバー・パッケージで EXECUTE 権限を付与できます。

エラー

DBMS_ALERT は、エラー条件でアプリケーション・エラー -20000 を呼び出します。表 16–2　に、メッセージとそれらを呼び出すプロシージャを示します。

表 16–2　DBMS_ALERT エラー・メッセージ

エラー・メッセージ	プロシージャ
ORU-10001 ロック要求エラー、ステータス： N	SIGNAL
ORU-10015 エラー： N パイプのステータスを待機中	WAITANY
ORU-10016 エラー： N パイプ 'X' で送信中	SIGNAL
ORU-10017 エラー： N パイプ 'X' で受信中	SIGNAL
ORU-10019 エラー： N ロック要求	WAIT
ORU-10020 エラー： N ロック要求	WAITANY
ORU-10021 ロック要求エラー、ステータス： N	REGISTER
ORU-10022 ロック要求エラー、ステータス： N	SIGNAL
ORU-10023 ロック要求エラー、ステータス N	WAITONE
ORU-10024 登録済みアラートがありません	WAITANY
ORU-10025 ロック要求エラー、ステータス N	REGISTER
ORU-10037 同一セッションからのコミットされない通知を待機しようとしています	WAITONE

アラートの使用

アプリケーションに複数のイベントを登録し、WAITANY コールを使用して、どれかが発生するのを待機できます。

また、アプリケーションは、WAITONE コールや WAITANY コールにオプションの TIMEOUT パラメータを指定することもできます。TIMEOUT を 0 にしておいた場合、保留中のアラートが存在しないと、すぐに戻されます。

通知セッションから、待機中のセッションにメッセージを渡すこともできます。

対応するアプリケーション WAIT コールの回数よりも多くのアラートを通知できます。そのような場合、古いアラートが廃棄されます。常にアプリケーションは（トランザクションのコミットごとに）最新のアラートを受け取ります。

アプリケーションにトランザクション・ベースのアラートが必要ない場合は、DBMS_PIPE パッケージを利用するとよいでしょう。

関連項目： 12-2 ページの「データベース・パイプ」

DBMS_ALERT.SIGNAL へのコールの後にトランザクションがロールバックされると、アラートは発生しません。

アラートを受け取って、データを読み込んでみると、データが変更されていない場合があります。これは、「前の」アラートを受け取った後、そのアラートのデータを読み込む前に、データが変更されたためです。

REGISTER プロシージャ

REGISTER プロシージャによって、セッションはアラートを登録できます。このアラートの名前が IN パラメータです。セッションは、アラートをいくつでも登録できます。セッションで必要なくなったアラートは、REMOVE をコールして登録を取り消すべきです。

警告： "ORA$" で始まるアラート名は、オラクル社が提供する製品用に予約されています。

構文

REGISTER プロシージャの構文は、次のとおりです。

```
DBMS_ALERT.REGISTER(name IN VARCHAR2);
```

REMOVE プロシージャ

REMOVE プロシージャによって、セッションは必要のなくなったアラートをその登録リストから取り消せます。アラートを抹消すると、アラートの送信による作業量が減少します。

セッションがアラートを抹消しないで終了しても、そのアラートは最終的にクリーン・アップされます（ただちに行われるわけではありません）。

構文

REMOVE プロシージャの構文は、次のとおりです。

```
DBMS_ALERT.REMOVE(name IN VARCHAR2);
```

SIGNAL プロシージャ

アラートを通知するには、SIGNAL をコールします。トランザクションがコミットしたときだけ SIGNAL コールの効果があります。トランザクションがロールバックすると、SIGNAL コールはなにもしません。

このアラートを登録しているすべてのセッションに通知が送られます。登録しているセッションが現在待機中であれば起動されます。登録しているセッションが現在待機中でなければ、次に待機したときに通知されます。複数のセッションが同時に同じアラートの通知を受けることができます。各セッションは、そのアラートを通知する際に、コミットするまで他の並行セッションの実行を阻止します。これによって、トランザクションは直列化されます。

構文

表 16–3　に、SIGNAL プロシージャのパラメータを示します。このプロシージャの構文は、次のとおりです。

```
DBMS_ALERT.SIGNAL(name IN VARCHAR2,
                  message IN VARCHAR2);
```

表 16–3　DBMS_ALERT.SIGNAL プロシージャのパラメータ

パラメータ	説明
name	通知するアラートの名前を指定します。
message	1800 バイトまたはそれ未満からなるメッセージを指定して、このアラートと対応付けます。このメッセージは、待機中のセッションに渡されます。待機中のセッションは、このメッセージの情報の使用によって、アラートが発生した後にデータベースを読み込まないようにすることもできます。

WAITANY プロシージャ

WAITANY をコールして、現行のセッションに登録されているアラートが発生するのを待機します。アラートを待機する同じセッションが、最初にアラートを通知する場合もあります。この場合には、通知の後と待機の前に忘れずにコミットしてください。そうしないと、DBMS_LOCK.REQUEST(DBMS_ALERT によってコールされます) がステータス 4 を戻します。

構文

表 16–4　に、WAITANY プロシージャのパラメータを示します。このプロシージャの構文は、次のとおりです。

```
DBMS_ALERT.WAITANY(name     OUT  VARCHAR2,
                   message  OUT  VARCHAR2,
                   status   OUT  INTEGER,
                   timeout   IN  NUMBER DEFAULT MAXWAIT);
```

表 16–4　DBMS_ALERT.WAITANY プロシージャのパラメータ

パラメータ	説明
name	発生したアラートの名前を戻します。
message	そのアラートと対応付けられているメッセージを戻します。これは SIGNAL コールによって与えられたメッセージです。WAITANY コールの前にこのアラートの通知が複数発生した場合、メッセージは最新の SIGNAL コールに対応します。それ以前の SIGNAL コールのメッセージは廃棄されます。
status	戻される値です。それぞれの値が持つ意味は以下のとおりです。 0 - アラートが発生した 1 - タイムアウトが発生した
timeout	アラートを待機する最大時間（単位は秒）を指定します。TIMEOUT（単位は秒）の前にアラートが発生しないと、ステータス 1 が戻されます。

WAITONE プロシージャ

WAITONE をコールして、特定のアラートが発生するのを待機します。最初にアラートを通知したセッションが、後続のトランザクションでアラートを待機することもできます。この場合には、通知の後と待機の前に忘れずにコミットしてください。そうしないと、DBMS_LOCK.REQUEST(DBMS_ALERT がコールします) がステータス 4 を戻します。

構文

表 16–5　に、WAITONE プロシージャのパラメータを示します。このプロシージャの構文は、次のとおりです。

```
DBMS_ALERT.WAITONE(name      IN  VARCHAR2,
                   message  OUT  VARCHAR2,
                   status   OUT  INTEGER,
                   timeout   IN  NUMBER DEFAULT MAXWAIT);
```

表 16–5　DBMS_ALERT.WAITONE プロシージャのパラメータ

パラメータ	説明
name	待機するアラートの名前を指定します。
message	そのアラートと対応付けられているメッセージを戻します。これは SIGNAL コールによって与えられたメッセージです。WAITONE コールの前にこのアラートの通知が複数発生した場合、メッセージは最新の SIGNAL コールに対応します。それ以前の SIGNAL コールのメッセージは廃棄されます。
status	戻される値です。それぞれの値が持つ意味は以下のとおりです。 0 - アラートが発生した 1 - タイムアウトが発生した
timeout	アラートを待機する最大時間（単位は秒）を指定します。指定したアラートが TIMEOUT（単位は秒）の前に発生しないと、ステータス 1 が戻されます。

アラートの検査

通常、Oracle はイベント駆動です。つまり、ポーリング・ループは存在しません。次の 2 つの場合、ポーリング・ループが発生する可能性があります。

- 共有モード。データベースが共有モードで稼働している場合、別のインスタンスからのアラートを検査するためにポーリング・ループが必要となります。特に何も指定しないと、ポーリング・ループは 1 秒間隔で実行され、SET_DEFAULTS コールで設定できます。
- WAITANY コール。WAITANY コールを使用する場合、通知するセッションが通知を行っても、その通知の 1 秒以内にコミットを行わないと、このコミットされていないアラートが他のアラートを隠さないように、ポーリング・ループが必要となります。ポーリング・ループは 1 秒間隔で始まり、30 秒間隔まで指数関数的に変化します。

SET_DEFAULTS プロシージャ

ポーリング・ループが必要な場合は、SET_DEFAULTS プロシージャを使用して、POLLING_INTERVAL を設定します。POLLING_INTERVAL は、ポーリング間のスリープの時間（単位は秒）です。デフォルトの間隔は 5 秒です。

構文

この SET_DEFAULTS プロシージャの構文は、次のとおりです。

```
DBMS_ALERT.SET_DEFAULTS(polling_interval IN NUMBER);
```

アラートの使用例

全従業員を対象に、部門別の平均給与をグラフで示す場合を考えます。アプリケーションはEMP の変更について常に注意している必要があります。このアプリケーションのコーディング例を次に示します。

```
dbms_alert.register('emp_table_alert');
    readagain:
   /* ... emp 表を読み込み、それをグラフ化する */
      dbms_alert.waitone('emp_table_alert', :message, :status);
      if status = 0 then goto readagain; else
      /* ... エラー条件 */
```

EMP 表のトリガーの例を次に示します。

```
CREATE TRIGGER emptrig AFTER INSERT OR UPDATE OR DELETE ON emp
    BEGIN
      dbms_alert.signal('emp_table_alert', 'message_text');
   END;
```

アプリケーションは、アラートが必要なくなった場合、次のような要求を行います。

```
dbms_alert.remove('emp_table_alert');
```

これによって、アラートの通知のための負荷が減少します。登録されたアラートが存在するにのに、セッションが終了（または、異常終了）すると、このパッケージを次に利用したユーザーによってアラートはクリーン・アップされます。

上の例では、アプリケーションは、即時性のある値をすべて参照することはできませんが、常に最新のデータを参照することが保証されます。

17

セキュリティ方針の設定

Oracle データベースのセキュリティを維持するために多くの種類のメカニズムが提供されているので、次に示す項目について検討した上で、任意のセキュリティ方針を設計し、インプリメントしてください。

- アプリケーション・レベルでのセキュリティのレベル
- システムと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および取消し
- ロールの作成、変更、削除
- ロールの使用と制御

この章では、セキュリティ方針の立案にかかわるこれらのトピックと手引について説明します。Trusted Oracle を使用している場合、システム全体のセキュリティ方針の設定の追加情報は、Trusted Oracle のマニュアルを参照してください。

アプリケーションのセキュリティ方針

セキュリティ方針は、データベース・アプリケーションごとに立案しなければなりません。たとえば、開発した各データベース・アプリケーション（たとえば、プリコンパイラ・プログラム、Oracle Forms のフォームなど）は、アプリケーション実行時に異なるセキュリティ・レベルを与えるアプリケーション・ロールを持っていなければなりません。アプリケーション・ロールは、ユーザー・ロール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。

制限のない SQL 文を実行できるアプリケーション（SQL*Plus、SQL*ReportWriter など）についても、機密扱いのスキーマ・オブジェクトや重要なスキーマ・オブジェクトに対する不法なアクセスを事前に防ぐために、厳密な制御が必要です。

アプリケーション管理者

大量のデータベース・アプリケーション（プリコンパイラ・アプリケーション、Oracle Forms アプリケーションなど）を使用する大規模なデータベース・システムでは、アプリケーション管理者が必要です。アプリケーション管理者は次のようなことに対して責任があります。

- アプリケーションに対するロールの作成と各アプリケーション・ロールの権限の管理
- データベース・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と管理
- 必要に応じて、アプリケーション・コードや Oracle プロシージャおよびパッケージのメンテナンスと更新

アプリケーション開発者として、アプリケーション管理者の責務も果たすこともできます。しかし、データベース・アプリケーションに精通した別の担当者ごとに仕事を任せることもできます。

ロールとアプリケーションの権限管理

ほとんどのデータベース・アプリケーションは、各種のスキーマ・オブジェクトに様々な権限を必要とするため、各アプリケーションごとに権限を記録することは非常に複雑で困難な業務となる場合があります。さらに、アプリケーションを実行するユーザーの認証には、大量の GRANT オペレーションが必要になります 。アプリケーション権限管理を簡易にするためには、ロールを作成し、各アプリケーションを実行するために必要な権限を付与しなければなりません。実際には、アプリケーションはロールを数多く持っていて、アプリケーション実行中の利用能力の強弱を決める異なる権限が各ロールに付与されています。

例 管理者のアシスタントが部門のメンバーが取った休暇を記録するために、VACATION アプリケーションを使用する場合を想定します。次の手順に従ってください。

1. VACATION ロールを作成する。
2. VACATION ロールに対する VACATION アプリケーションで必要な権限をすべて付与する。

3. 管理者のアシスタント全員または ADMIN_ASSITS 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 ロールを付与する。

1 つのロールにアプリケーション権限をまとめておけば、権限を管理する上で便利です。次の管理オプションについて検討してください。

- 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る。そのため、従業員がジョブを変更するときに、（多数の権限を付与したり取り消すのではなく）1 つのロールを付与または取り消すだけで済みます。
- アプリケーションのユーザー全員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だけを修正することによって、アプリケーションに対応する権限を変更できる。
- ROLE_TAB_PRIVS および ROLE_SYS_PRIVS の各データ・ディクショナリ・ビューを問い合せることによって、特定のアプリケーションを実行するのに必要な権限を判断できる。
- DBA_ROLE_PRIVS データ・ディクショナリ・ビューを問い合せることによって、アプリケーション上の権限を有するユーザーを判断できる。

アプリケーション・ロールを使用可能にする

単一ユーザーは、複数のアプリケーションとそれに対応するロールを使用できます。ただし、現在アプリケーションを実行しているロールに対応する権限を 1 人のユーザーに対してだけ許可しなければなりません。たとえば、次のシナリオを考えてみます。

- ORDER ロール（ORDER アプリケーション用）は、INVENTORY 表に対する UPDATE 権限を含んでいる。
- INVENTORY ロール（INVENTORY アプリケーション用）は、INVENTORY 表に対する SELECT 権限を含んでいる。
- 注文記入担当者は、ORDER と INVENTORY ロールを両方とも付与されていなければならない。

このため、両方のロールを付与されている注文記入担当者は、INVENTORY 表を更新する INVENTORY アプリケーションを実行するときに、おそらく ORDER ロールの権限を使用できるはずです。ただし、INVENTORY 表に対する修正の更新は、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限り許可されます。

こうした問題を避けるために、各アプリケーションの最初に SET ROLE 文を発行して、自動的にそれに対応するロールを使用可能にします。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。SET ROLE コマンドを使用することによって、各アプリケーションは、必要に応じて、ユーザーに対して特定の権限を動的に使用可能にします。ユーザーは、一定のアプリケーションの実行時にだけアプリケーションの権限を利用できるので、意図的かどうかにかかわらず、アプリケーションのコンテキスト以外での権限の乱用を未然に防ぐことができます。

SET ROLE 文は、ユーザーがアクセス可能な情報を制御すること以外にも、ユーザーが情報にアクセス可能な時期を制御したりできるという点で、権限管理が非常に容易になります。また、SET ROLE 文によって、ユーザーは正しく定義された権限定義域内で操作できます。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。詳細は、17-9 ページの「ロールを使用可能、使用禁止にする」を参照してください。

SET_ROLE プロシージャ

DBMS_SESSIONS.SET_ROLE プロシージャは SET ROLE 文と同様の動作をするもので、PL/SQL からアクセスできます。ストアド・プロシージャから SET_ROLE はコールできません。この制限によって、ストアド・プロシージャは実行中にセキュリティ定義域を変更できません。ストアド・プロシージャは、プロシージャの作成者のセキュリティ定義域下で実行されます。

DBMS_SESSION.SET_ROLE は、PL/SQL ブロックからだけコールできます。アノニマス・ブロックがコンパイルされると、PL/SQL は SQL に対してセキュリティ・チェックを行うので、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 SET_ROLE が影響を与えることはありません（つまり、使用可能にしたロールは影響を与えません）。

たとえば、JOE スキーマ内の FINANCE 表から選択でき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、次のブロックは失敗します。

```
DECLARE
    n NUMBER
BEGIN
    SYS.DBMS_SESSION.SET_ROLE('acct')
    SELECT empno INTO n FROM JOE.FINANCE
END;
```

このブロックが失敗するのは、JOE.FINANCE 表に対する 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からです。コンパイル時には、まだ ACCT ロールを使用可能にしないでください。このロールは、ブロックが実行されるまで使用禁止です。

しかし、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は、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 のコールは、DBMS_SQL パッケージのコールを使用して実行された SQL に影響を与えます。次のブロックは成功します。

```
DECLARE
    n NUMBER
BEGIN
    SYS.DBMS_SESSION.SET_ROLE('acct');
    SYS.DBMS_SQL.PARSE
      ('SELECT empno FROM JOE.FINANCE');
    . . .
    --other calls to SYS.DBMS_SQL
```

```
    . . .
END;
```

Tool ユーザーに対してアプリケーション・ロールを制限する

あらかじめ構築されているデータベース・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にしたり、使用禁止にしたりすることも含めて、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 Tools を使用すると、ユーザーは付与されたロールを使用可能、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 （正常終了または異常終了のいずれであってもかまいません）。この結果、重大なセキュリティ上の問題が発生する可能性があります。予防措置をとらなければ、アプリケーション・ユーザーが非定型ツールの使用中に、アプリケーション・ロールによって取得した権限を使用して、データベースの表に対して破壊的な SQL 文を故意に、または無意識に発行する可能性があります。

たとえば、次のシナリオを考えてみます。

- VACATION アプリケーションはそれに対応する VACATION ロールを持っている。
- VACATION ロールは、SELECT、INSERT、UPDATE、DELETE の各文を EMP 表に対して発行する権限を含んでいる。
- VACATION アプリケーションは、VACATION ロールを介して受領した権限の使用を制御する。つまり、文が発行された時にアプリケーションが制御する。

ここで VACATION ロールを付与されたユーザーについて考えてみます。ただし、VACATION アプリケーションを使用するかわりに、ユーザーは SQL*Plus を実行します。この時点で、明示的に付与された権限、または VACATION ロールを含むロールを介して付与されている権限だけによって、ユーザーは制限されます。SQL*Plus は非定型の問合せツールなので、設計したデータベース・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済みのアクションに制限されません。ユーザーは選択するとおりに、EMP 表のデータを問い合せたり、修正したりできます。

このような起こり得る問題を回避するために、アプリケーション・ロールに対して次の方針を検討します。

1. 各アプリケーションは別々のロールを持たなければなりません。
   - 1 つのロールには、アプリケーションの使用に必要な**すべての**権限が含まれていなければなりません。状況に応じて、アプリケーションの実行中にセキュリティを実現するための権限を多少なりとも含んでいるロールがいくつか存在している可能性があります。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、パスワードによって（またはオペレーティング・システム認証によって）保護されていなければなりません。
   - 上記以外の別のロールは、そのアプリケーションに対応する破壊的でない権限（つまり、そのアプリケーションに対応する特定の表またはビューに対する SELECT 権限）だけを含んでいなければなりません。読取り専用ロールは、アプリケーション・ユーザーに、SQL*Plus、SQL*ReportWriter、SQL*Graph などの非定型の Tools

を使用したカスタム・レポートの生成を許可します。ただし、このロールは、そのアプリケーション以外の表データの修正については許可しません。非定型の問合せ Tool のために設計されたロールは、パスワードで（またはオペレーティング・システム認証で）保護または非保護のいずれでも実施できます。

2. 起動時に、各アプリケーションは SET ROLE コマンドを使用して、そのアプリケーションに対応するアプリケーション・ロールを使用可能にしなければなりません。ロールを許可するためにパスワードを使用する場合には、そのパスワードをアプリケーション内の SET ROLE 文に指定しなければなりません（可能な場合には、アプリケーションによって暗号化します）。ロールをオペレーティング・システムによって許可する場合には、システム管理者は、アプリケーション・ユーザーがアプリケーション使用時に適切なオペレーティング・システム権限を受領できるように、必ずユーザー・アカウントとアプリケーションをセットアップしなければなりません。
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しなければなりません。
4. 必要に応じて、アプリケーション・ユーザーにアプリケーション・ロールを付与すべきです。管理者は、ユーザーに対して破壊的でないロールを付与しないことで、ユーザーが非定型の Tools でアプリケーション・データを使用するのを禁止できます。

この構成を使用すれば、アプリケーションの起動時に各アプリケーションは適切なロールを使用可能にします。また、アプリケーションの終了時に各アプリケーションはロールを使用禁止にします。アプリケーション・ユーザーが非定型のツールの使用を決定した場合、そのツール向けの安全なロールだけを使用可能にできます。

さらに次の処理が可能です。

- PRODUCT_USER_PROFILE 表を使用して、SQL*Plus の起動時にそのロールを使用可能になるように指定できる。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 OCI アプリケーションの機能に類似しています。
- PRODUCT_USER_PROFILE 表を使用して、SQL*Plus ユーザーに対して SET ROLE コマンドの使用を禁止する。これによって、SQL*Plus ユーザーは SQL*Plus の起動時にそのロールに対応する権限だけが許可されます。

SQL*ReportWriter、SQL*Graph などの他の非定型の問合せやレポートの Tools もまた、その製品を実行している間、各ユーザーが使用可能なロールやコマンドを制限するために、PRODUCT_USER_PROFILE 表を使用できます。これらの機能に関する説明は、各 Tool のマニュアルを参照してください。

スキーマ

各データベースのユーザー名は、スキーマと呼ばれています。これは、スキーマ・オブジェクトが含まれるセキュリティ・ドメインです。データベースやそのオブジェクトに対するアクセスは、各スキーマに対して付与された権限によって制御されます。

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セットアップされたアカウントです。ただし、一意のスキーマはデータベースへの接続を許可しませんが、一連の関連性のあるオブジェクトを含むために使用します。こうしたスキーマは、通常のユーザーとして作成されますが、まだ（明示的またはロールを介して）CREATE SESSION システム権限は付与されません。しかし、単一トランザクションで複数の表とビューを作成するために CREATE SCHEMA コマンドを使用する場合、このようなスキーマに対する CREATE SESSION 権限を一時的に付与しなければなりません。

たとえば、特定のアプリケーションに対するスキーマ・オブジェクトは、スキーマによって所有されません。アプリケーション・ユーザー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、権限がある場合にはそのアプリケーションとそれに対応するオブジェクトを使用できます。ただし、ユーザーはそのアプリケーションに対してセットアップ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。このため、このスキーマを介して対応するオブジェクトへのアクセスはできません。このセキュリティ構成によって、スキーマ・オブジェクトの別の保護レイヤーが提供されます。

権限とロールを管理する

アプリケーション設計の一段階として、アプリケーションを使用するユーザーのタイプや特定の業務を遂行するために付与しなければならないアクセス・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しなければならない権限を決定します。

通常、エンドユーザにはオブジェクト権限が付与されています。オブジェクト権限により、ユーザーは特定の表、ビュー、順序、プロシージャ、関数、パッケージに対して特定のアクションを実行できます。オブジェクトのタイプによって、さまざまな種類のオブジェクト権限があります。表 17–1 には、オブジェクトの各タイプに使用可能なオブジェクト権限がまとめてあります。

表 17-1　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		

- 1 「プロシージャ」は、スタンドアロン型ストアド・プロシージャ、関数、パブリック・パッケージ構成を指します。
- 2 権限はロールに付与できません。
- 3 スナップショットにも付与できません。
- 表 17–2 には、表 17–1 に記載されたオブジェクト権限によって許可される SQL 文がリストされています。

アプリケーションをインプリメントおよびテストする際に、各ロールを作成し、各ロールの用途をテストして、アプリケーション・ユーザーがデータベースに対して適切にアクセスするように設定しなければなりません。テストの完了後、アプリケーション管理者と調整の上、各ユーザーに適切なロールを割り当てていることを確認しなければなりません。

表 17-2　データベース・オブジェクト権限により許可されている SQL 文

オブジェクト権限	許可されている 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関数）
	パブリック・パッケージ変数への参照
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）に関する FOREIGN KEY 整合性制約を定義
	した CREATE または ALTER TABLE 文
SELECT	SELECT...FROM オブジェクト（表、ビュー、スナップショット）
	順序を使用した SQL 文

ロールを作成する

ロールの使用は、次の例に示されているように、対応するパスワードで保護できます。

```
CREATE ROLE clerk IDENTIFIED BY bicentennial;
```

パスワードによって保護されたロールが付与されている場合、ロールを使用可能または使用禁止にするには、SET ROLE コマンドによりそのロールの正しいパスワードを入力するしかありません（詳細は、17-10 ページの「明示的にロールを使用可能にする」を参照してください。また、オペレーティング・システムの情報を使用して、ロールの使用 を許可するため

に、ロールを作成することもできます。ロールの認可にオペレーティング・システムを使用する方法の詳細は、『Oracle8 Server 管理者ガイド』を参照してください。

保護されずに作成されたロールの場合には、任意の権限受領者が、そのロールを使用可能または使用禁止にできます。

データベース・アプリケーションは、通常、ロール認可機能を使用して、個々にアプリケーション・ロールを使用可能にしたり、ユーザーの他の全ロールを使用禁止にしたりできます。この方法ではユーザーは、別のアプリケーション向けの権限（ロールから）を使用できません。非定型の問合せ Tools（たとえば、SQL*Plus、Enterprise Manager）を使用すれば、ユーザーは許可されるロールに限り（つまり、パスワードを把握したロール、またはオペレーティング・システムによって許可されているロールの場合）、明示的に使用可能にできます。詳細は、17-5 ページの「Tool ユーザーに対してアプリケーション・ロールを制限する」を参照してください。

新しいロールを作成する場合は、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の名前を使用しなければなりません。ロールはユーザーのスキーマ上に指定できません。

ロール作成直後は、その作成に対応する権限をロールは持っていません。権限を新しいロールと対応付けるには、新しく作成されたロールに権限または他のロールを付与しなければなりません。

ロールを作成するために必要な権限

ロールを作成するには、CREATE ROLE システム権限が必要です。

ロールを使用可能、使用禁止にする

ユーザーにはロールを付与できますが、そのロールをあらかじめ使用可能にしておかないと、それに対応する権限がユーザーの現在のセッションで使用できません。ユーザー・ロールは、任意に使用可能にしたり、使用禁止したりできます。次の項で、ロールを使用可能または使用禁止にすべき時期、そのための各種方法について解説します。

どんな場合にロールを使用可能にするか

通常、ユーザーのセキュリティ定義域は、現在のタスクを実行できるように許可し、必ず、ユーザーが現行のジョブに不要な権限を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・アプリケーションを使用して処理するために必要な全権限を持っていなければなりませんが、他のデータベース・アプリケーションに対する権限は必要としません。つまり、ユーザーが多数の権限を持っていると、意図していない事象により、情報にアクセスできない場合があります。

ユーザーに対して直接付与された権限であれば、いつでもユーザーはこの権限を使用できます。このため、直接付与された権限の場合、現在のタスクに応じて選択的に使用可能にしたり、使用禁止にしたりすることはできません。かわりに、ロールに付与された権限は、ロールの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、ユーザーのロールを選択的に使用可能にする方法（および使用禁止する方法）について解説します。

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るロールです。デフォルトのロールのユーザー・リストには、必ず代表的なジョブ関数に対応するロールが含まれていなければなりません。

各ユーザーはロールを持っていないか、1 つまたは複数のデフォルトのロールを持っています。ユーザーに直接付与されたロールは、そのユーザーのデフォルトのロールにすることもできます。一方、間接的に付与されたロール（ロールに付与されたロール）を、デフォルトのロールにすることはできません。したがって、直接的に付与されたロールに限り、ユーザーのデフォルトのロールにできます。

ユーザーのデフォルトのロールの数は、ユーザーごとに許可されている使用可能ロールの数を超えてはいけません (初期化パラメータ MAX_ENABLED_ROLES の説明を参照してください)。ユーザーのデフォルト・ロールの最大数を超えると、接続しようとしてもエラーが戻され、接続が許可されません。

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。デフォルト・ロールのユーザー・リスト中のロールを指定すると、そのロールに対する認証をバイパス（無視）できます。つまり、そのロールがパスワードやオペレーティング・システムによって認証されません。

デフォルトのロールのユーザー・リストを設定することができ、このリストは、SQL コマンド ALTER USER を使用して変更できます。デフォルト・ロールのユーザー・リストを ALL に指定すると、ユーザーに対して付与された全ロールが、自動的にデフォルト・ロールのユーザー・リストに追加されます。ユーザーのデフォルト・ロールの直後の修正に限り、デフォルト・ロールのユーザー・リスト上から、新たに付与されたロールを除去できます。

ユーザーのデフォルト・ロールのリストに対する修正は、変更後またはロールの付与の後で作成されたセッションに限り適用されます。いずれの方法も、ユーザー変更やロール付与の処理中のセッションには適用されません。

明示的にロールを使用可能にする

ユーザー（またはアプリケーション）は、SQL コマンド SET ROLE を使用して、ロールを明示的に使用可能にできます。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 文は指定したすべてのロールを使用可能にします。SET ROLE 文中に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に至るまですべて、使用禁止になります。

他のロールを含んでいるロールを使用可能にすると、これらの間接的に付与されたすべてのロールが明確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には、そのロールに SET ROLE 文でロールのパスワードを指示することでだけ使用可能にできます。ロールがパスワードによって保護

されていない場合には、単純に SET ROLE 文を指定するだけで、そのロールを使用可能にできます。たとえば、Morris のセキュリティ定義域の前提条件が、次のとおりだと想定します。

- PAYROLL_CLERK（パスワードの BICENTENNIAL）、ACCTS_PAY（パスワードの GARFIELD）、ACCTS_REC（外部的に識別される）といった 3 種類のロールが付与されている。PAYROLL_CLERK ロールは、間接的に付与されたロール PAYROLL_REPORT（外部的に識別される）を含んでいなければなりません。
- この唯一のデフォルト・ロールは PAYROLL_CLERK である。

次の文を指定することによって、Morris の現在使用可能なロールとして、そのデフォルト・ロール PAYROLL_CLERK から、ACCTS_PAY と ACCTS_REC へ変更できます。

```
SET ROLE accts_pay IDENTIFIED BY garfield, accts_rec;
```

なお、この最初の指定文では、単一の SET ROLE 文で複数のロールを使用可能にしています。また、SET ROLE コマンドの ALL と ALL EXCEPT オプションによって、ユーザーに付与されている複数のロールを、次のような 1 つの文で使用可能にすることもできます。

```
SET ROLE ALL EXCEPT payroll_clerk;
```

上記の指定文は、SET ROLE コマンドの ALL EXCEPT オプションの使用方法を示しています。使用禁止の対象となるロールが少なくて、大量のユーザー・ロールを使用可能にする場合には、このオプションを使用します。同様にして、Morris の全ロールを次の文で使用可能にできます。

```
SET ROLE ALL;
```

SET ROLE コマンドの ALL または ALL EXCEPT オプションを使用する際、使用可能となる全ロールの認証は、パスワードを不要とするもの、またはオペレーティング・システムによる認証のいずれかでなければなりません。ロールがパスワードを必要とする場合に、SET ROLE ALL または ALL EXCEPT 文を指定すると、ロールバックされ、エラーが戻されます。

別のロールの明示的な付与を介しても、間接的に付与された任意のロールを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;
```

明示的にロールを使用可能にするために必要な権限　必要に応じて、権限受領者がロール・パスワードを指定している場合、ユーザーは SET ROLE コマンドを使用して、付与されたロールを任意に使用可能にできます。

OS_ROLES=TRUE の場合にロールを使用可能、使用禁止にする

OS_ROLES が TRUE に設定されている場合には、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 コマンド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については、たとえ GRANT 文を使用してロールを付与していたとしても、SET ROLE 文では指定できません（無視されます）。

OS_ROLES に TRUE を設定しているとき、初期化パラメータの MAX_ENABLED_ROLES で指定した数のロールだけ使用できます。ロールの認可にオペレーティング・システムを使用する方法の詳細は、『Oracle8 Server 管理者ガイド』を参照してください。

ロールを削除する

ロールを削除するとき、そのロールに付与されている全ユーザーとロールのセキュリティ定義域は、削除の結果、欠落したロール権限を反映するため即座に変更されます。削除されたロールによって間接的に付与された全ロールも、影響を受けたセキュリティ定義域から削除されます。ロールを削除することによって、自動的に全ユーザーのデフォルト・ロールのリストからそのロールが削除されます。

オブジェクトの作成はロールを介して受領した権限に従属していませんので、ロールを削除するときに、オブジェクトにカスケードする影響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）。

次のように、SQL コマンド DROP ROLE を使用してロールを削除します。

```
DROP ROLE clerk;
```

ロールを削除するために必要な権限

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なければなりません。

権限とロールの付与および取消し

以降の項では、システム権限およびロール、スキーマ・オブジェクト権限を付与および取り消す方法を説明します。

システム権限とロールを付与する

システム権限やロールは、次の使用例に示すように、SQL コマンド GRANT を使用して、他のロールやユーザーに付与できます。

```
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　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際には無効となります）。このオプションを使用する権限受領者は、さらに次の機能を使用できます。

- 権限受領者は、**任意の**ユーザーまたはデータベース上のその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る。（ただし、ユーザーは受領したロールを自ら取り消すことはできません）。

- 権限受領者は、これ以降 ADMIN OPTION を使用して、システム権限やロールを付与できる。
- ロールの権限受領者は、ロールを変更したり削除したりすることができる。

ADMIN OPTION を指定しなければ、権限受領者は上記のオペレーションを実行できません。

ユーザーがロールを作成するときには、ADMIN OPTION を使用した作成者に対して自動的にロールが付与されます。

次の文を使用して、MICHAEL に対して 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;
```

ユーザーの MICHAEL は、NEW_DBA ロールにおいて全権限を暗黙的に使用できるだけではなく、必要に応じて NEW_DBA ロールを付与したり、取り消したり、削除したりできます。

システム権限またはロールを付与するために必要な権限　システム権限やロールを付与するためには、権限付与者はその対象となるシステム権限とロールすべてに対して、ADMIN OPTION を指定する必要があります。さらに、GRANT ANY ROLE システム権限を有する任意のユーザーは、データベース上で任意のロールを付与できます。

スキーマ・オブジェクト権限を付与する

SQL コマンド GRANT を使って、ロールまたはユーザーにスキーマ・オブジェクト権限を付与します。たとえば、以下の文は JWARD と TSMITH の各ユーザーに対して、EMP 表の全列に関する SELECT、INSERT、DELETE の各オブジェクト権限を付与します。

```
GRANT select, insert, delete ON emp TO jward, tsmith;
```

JWARD と TSMITH の各ユーザーに対して、EMP 表の ENAME 列と JOB 列に限り INSERT オブジェクト権限を付与するためには、次のように入力します。

```
GRANT insert(ename, job) ON emp 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、次のように ALL を使います。

```
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　スキーマ・オブジェクト権限は、GRANT OPTION を有するユーザーに対して付与できます。この特殊な権限によって、権限受領者は次のような権限がさらに許可されます。

- 権限受領者は、データベース上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る。

- 権限受領者は GRANT OPTION の有無にかかわらず、他のユーザーに対してスキーマ・オブジェクト権限を付与できる。
- 権限受領者が GRANT OPTION 付きで表に対するスキーマ・オブジェクト権限を受領するときに、CREATE VIEW または CREATE ANY VIEW の各システム権限を持っていれば、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およびロールに対してそのビューに対応する権限を付与できる。

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： スキーマ・オブジェクト権限をロールに付与するとき GRANT OPTION は無効です。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未然に防ぐので、ロールの権限受領者は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はできません。

スキーマ・オブジェクト権限を付与するために必要な権限 スキーマ・オブジェクト権限を付与するには、権限付与者は必ず次のいずれかの条件に該当していなければなりません。

- 指定の対象となるスキーマ・オブジェクトの所有者であること。
- 権限付与の対象となるオブジェクト権限が `GRANT OPTION` 付きであらかじめ付与されていること。

システム権限とロールを取り消す

システム権限やロールは、次の使用例に示すように、SQL コマンド REVOKE を使用して、取り消すことができます。

```
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 を使用しないで、その権限とロールを再付与しなければなりません。

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 を使用するユーザーは、他の任意のデータベース・ユーザーやロールから権限やロールを取り消すことができます（この場合、ユーザーがその権限や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 を有するユーザーは、**任意の**ロールを取り消すことができます。

スキーマ・オブジェクト権限を取り消す

スキーマ・オブジェクト権限は、SQL コマンド REVOKE を使って取り消すことができます。たとえば、ユーザーがもとの権限付与者である場合に、JWARD や TSMITH の各ユーザーから、EMP 表の SELECT 権限と INSERT 権限を取り消すためには、次のように入力します。

```
REVOKE select, insert ON emp
    FROM jward, tsmith;
```

権限付与者は、次のように入力すれば、HUMAN_RESOURCES ロールに対して付与した DEPT 表の全権限を取り消すこともできます（たとえ単一の権限が付与されていたとしてもそれが取り消されます）。

```
REVOKE ALL ON dept FROM human_resources;
```

上記の指定文は、権限付与者が許可した権限だけを取り消し、他のユーザーによる権限付与については取り消しません。スキーマ・オブジェクト権限の GRANT OPTION だけを選択して取り消すことはできません。まずオブジェクト権限を取り消してから、GRANT OPTION なしで再付与しなければなりません。ユーザーは自分のスキーマ・オブジェクト権限を取り消すことはできません。

列選択スキーマ・オブジェクト権限を取り消す 前に記述したとおり、列固有の INSERT、UPDATE、REFERENCES 権限を、表やビューの列に選択的に付与できます。しかし、それに類似する 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、新規の列固有の権限を選択して再度付与しなければなりません。

たとえば、HUMAN_RESOURCES ロールに、DEPT 表の DEPTNO と DNAME の各列に対する UPDATE 権限が付与されていると想定します。その UPDATE 権限を DEPTNO 列だけから取り消すためには、次の 2 つの文を入力します。

```
REVOKE UPDATE ON dept FROM human_resources;
GRANT UPDATE (dname) ON dept TO human_resources;
```

REVOKE 文により、HUMAN_RESOURCES ロールから DEPT 表のすべての列の UPDATE 権限を取り消します。GRANT 文が、HUMAN_RESOURCES ロールに対して、DNAME 列に関する UPDATE 権限を再付与します。

REFERENCES スキーマ・オブジェクト権限を取り消す REFERENCES オブジェクト権限の権限受領者が外部キーの制約（現行の制約）を設定するための権限を使用している場合、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、唯一その権限を取り消すことができます。

```
REVOKE REFERENCES ON dept FROM jward CASCADE CONSTRAINTS;
```

CASCADE CONSTRAINTS オプションを指定すると、現在定義されており、取り消した REFERENCES 権限を使う外部キーの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削除するには、取消しを行うユーザーが必ず、取消しの対象となるオブジェクト権限の付与者でなければなりません。

権限の取消しによるカスケードの影響

権限の種類に応じて、権限の取消しによるなんらかの影響がカスケードする場合とそうでない場合とがあります。次の項では、このように付随して発生する各種カスケードの影響について解説します。

システム権限 DDL オペレーションに関連したシステム権限を取り消すときには、カスケードする影響はありません。このとき、ADMIN OPTION の使用や、権限付与の有無には関係ありません。たとえば、次の条件を想定します。

1. WITH OPTION を使用して、JWARD に対して CREATE TABLE システム権限を付与します。
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 システム権限を取り消します。
6. JWARD の表は引き続き存在しています。TSMITH は引き続き CREATE TABLE システム権限を有し、その表もなお存在しています。

カスケードする影響は、DML オペレーションに関連するシステム権限を削除するときに発生しています。たとえば、SELECT ANY TABLE をユーザーに対して付与するときに、そのユーザーがプロシージャを事前に作成してある場合、ユーザーのスキーマに含まれている全プロシージャは、再度使用可能となる前に（その取消しの後で）必ず許可し直さなければなりません。

スキーマ・オブジェクト権限 スキーマ・オブジェクト権限を取り消すと、それに付随してカスケードによる様々な変更が発生するため、REVOKE 文を指定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・オブジェクト定義にまで影響を及ぼす可能性がある。たとえば、TEST プロシージャの中に、EMP 表のデータを問い合せる SQL 文を指定したと想定します。EMP 表の SELECT 権限が TEST プロシージャの所有者から削除されると、それ以降そのプロシージャを正常に実行することはできません。
- 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 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ない。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でも存在します。
- 表に対する REFERENCES 権限をユーザーから取り消すと、その取り消された REFERENCES 権限を必要とするユーザーが定義した外部キーの整合性制約が自動的に削除される。たとえば、ユーザーの JWARD には DEPT 表の DEPTNO 列に対する REFERENCES 権限が付与されているときに、その DEPTNO 列を参照する EMP 表の DEPTNO 列上で外部キーを作成する場合を想定します。DEPT 表の DEPTNO 列の

REFERENCES 権限を取り消すと、その EMP 表の DEPTNO 列上の外部キーの制約まで、同一オペレーション上で削除されます。

- 権限付与者のオブジェクト権限が取り消されると、GRANT OPTION によっ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る。たとえば、GRANT OPTION を使用して、SELECT オブジェクト権限が USER1 に付与されているときに、USER2 に対し EMP 表の SELECT 権限を付与する場合を想定します。次に SELECT 権限は USER1 から取り消されます。この取消しは USER2 にも同様に適用されます。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。

ユーザー・グループ PUBLIC に対して付与する、取り消す

権限やロールを、ユーザー・グループ PUBLIC に対して付与したり、その設定を取り消したりできます。PUBLIC は全データベース・ユーザーがアクセス可能であるため、PUBLIC に対して付与された権限やロールには、全データベース・ユーザーがアクセスできます。

ある権限またはロールを全データベース・ユーザーが必要としている場合だけ、その権限またはロールを PUBLIC に付与します。ここでも一般的な規則として、各データベース・ユーザーは現在のタスクの正常実行のために必要な権限だけを保持することをお薦めします。

取り消される権限に応じて、PUBLIC の取消しによってカスケードする深刻な影響が発生す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）、データベース上の全プロシージャ（関数とパッケージを含む）は、許可し直さない限り再度使用可能にはなりません。DML 関連の権限を PUBLIC に対して付与するときには、十分な注意が必要です。

付与と取消しはいつ有効になるか

付与とその取消しの処理内容に応じて、それぞれの処理の実施時期は異なります。

- 任意の対象（ユーザー、ロール、PUBLIC）への権限（システムおよびスキーマ・オブジェクト）の付与および取消しはすべて、ただちに実施される。
- 任意の対象（ユーザー、ロール、PUBLIC）に対する、ロールの全付与 / 取消しは、現在のセッション・ユーザーがその付与 / 取消しを実施した後でロールを再使用できるように SET ROLE 文を発行するとき、またはその付与 / 取消しを実施した後で新規ユーザー・セッションが作成されるときに限り、実施される。

権限付与は依存オブジェクトにどのように影響するか

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。これにより、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 パッケージ本体内の依存スキーマ・オブジェクト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。

18

Oracle XA

この章では、Oracle XA ライブラリの使用方法について説明します。内容は次のとおりです。

- XA ライブラリ関連情報
- リリース 7.3 からリリース 8.0 への変更点
- 一般的な問題および制限事項
- 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_open 文字列の定義
- プリコンパイラおよび OCI へのインタフェース
- トランザクション制御
- 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
- XA ライブラリ・スレッド・セーフティ
- 障害追求

XA ライブラリ関連情報

Oracle XA に関する一般的な情報

Oracle XA ライブラリについて次のマニュアルを前もって読んでおいてください。また、その他の参照情報についても次のマニュアルを参照してください。

- 『Oracle コール・インターフェース・プログラマーズ・ガイド』

  関連項目： ライブラリ・リンク・ファイル名の詳細は、Oracle オペレーティング・システム固有のマニュアルを参照してください。

README.doc

README.doc ファイルは、Oracle オペレーティング・システム固有のマニュアルで指定されているディレクトリ内にあり、使用している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以降の変更点またはバグ、制限事項について記述しています。

リリース 7.3 からリリース 8.0 への変更点

次に示す変更が行われました。

- セッション・キャッシュが不要になった
- 動的な登録がサポートされた
- 疎結合トランザクション・ブランチがサポートされた
- OCI アプリケーションに SQLLIB が不要になった
- XA を実行するためのインストレーション・スクリプトが不要になった
- 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できるようになった
- Oracle Parallel Server のトランザクション回復が改善された
- グローバル・トランザクションとローカル・トランザクションの両方が可能になった
- xa_open 文字列が修正された

セッション・キャッシュが不要になった

セッション・キャッシュは、新しい OCI では不要です。そのため、旧 xa_open 文字列パラメータ、SesCacheSz は排除されました。結果として、init.ora の sessions パラメータを減らすこともできます。かわりに、init.ora の transactions 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。グローバル・トランザクションが再開されるときにセッションは移行されないため、アプリケーションで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。使用しているアプリケーションのサービスへの

編成方法については、トランザクション処理モニターに付属のマニュアルを参照してください。具体的には、セーブポイントとカーソル・フェッチの状態は、トランザクションが中断されたときに取り消されます。これは、ある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、別のサービスでは無効になるということです。たとえ、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セーブポイントは無効です。

動的な登録がサポートされた

動的な登録は、XA アプリケーションと Oracle Server がともにバージョン 8 である場合にだけ使用できます。

関連項目： 18-13 ページの「XA インタフェースの拡張」

疎結合トランザクション・ブランチがサポートされた

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、単一のインスタンスにおいては密結合トランザクション・ブランチだけがサポートされ、疎結合トランザクション・ブランチは異なる複数のインスタンスにおいてサポートされます。

OCI アプリケーションに SQLLIB が不要になった

以前の OCI アプリケーションでは 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性がない場合でも、OCI プログラマは SQLLIB を購入しなければなりませんでした。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レーション・スクリプトが不要になった

Oracle バージョン 8 では、XA アプリケーションを実行するための SQL スクリプト XAVIEW.SQL が必要なくなります。ただし、XAVIEW.SQL はバージョン 7.3 アプリケーションのために必要です。

関連項目： 18-16 ページの「DBA またはシステム管理者の責任」

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できるようになった

バージョン 7 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 ライブラリを使用できませんでした。分散ロック・マネージャのプラットフォームの処理系がプロセス・ベースのロックではなくトランザクション・ベースである場合にかぎり、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した。現在のリリースではこの制限がなくなりました。Oracle Parallel Server オプションを実行した後で、Oracle XA ライブラリを実行できます。

Oracle Parallel Server のトランザクション回復が改善された

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回復できます。**xa_recover** コールを使って、保留トランザクションのスナップショットを指定します。

グローバル・トランザクションとローカル・トランザクションの両方が可能になった

現在は XA 接続内で、グローバル・トランザクションとローカル・トランザクションの両方を持てます。ローカル・トランザクションとは、Oracle Server が完全に調整を行う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ています。

```
CONNECT SCOTT/TIGER;
UPDATE EMP set sal = sal + 1; /* ローカル・トランザクションの開始 */
COMMIT;                       /* トランザクションのコミット */
```

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 は従属として動作し、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ています。

```
xa_open(oracle_xa+acc=p/SCOTT/TIGER+sestm=10", 1, TMNOFLAGS);
                            /* トランザクション・マネージャが */
                            /* Oracle Server との接続をオープンする */
tpbegin();                  /* グローバル・トランザクションの開始 */
                            /* トランザクション・マネージャは Oracle Server に */
                            /*XA コマンドを発行してグローバル・トランザクションを開始する
*/
UPDATE EMP set sal = sal + 1;
                            /* グローバル・トランザクション内で */
                            /* 更新が実行される */
tpcommit();                 /* グローバル・トランザクションのコミット */
                            /* トランザクション・マネージャは Oracle Server に */
                            /* XA コマンドを発行して */
                            /* グローバル・トランザクションをコミットする */
```

Oracle7 Server は、ローカル・トランザクションが XA 接続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-2041 エラー・コードを戻します。

```
xa_open("oracle_xa+acc=p/SCOTT/TIGER+sestm=10" , 1, TMNOFLAGS);
                            /* トランザクション・マネージャは Oracle Server との */
                            /* 接続をオープンします */
UPDATE EMP set sal = sal + 1; /* Oracle 7 がエラーを戻します */
```

これに対して、Oracle8 Server では、XA 接続でローカル・トランザクションを開始できます。唯一の制限事項は、接続においてグローバル・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）しなければならないということです。

関連項目： グローバル・トランザクションの詳細は、『Oracle 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_open 文字列が修正された

次に示す 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 True に設定しないでください。True に設定されている場合は、グローバル・トランザクション・ブランチが疎結合になる、つまり、ロックがブランチ間で共有されないことを示します。

- `SesWt`

  このパラメータの値は、別のセッションによって使用されているトランザクションを待機するときの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・ブランチにスイッチす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。Oracle Server リリース 7.3 への接続時にだけ使用してください。

- `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・ブランチにスイッチできます。

- `SesCacheSz`

  セッション・キャッシュが行われなくなったため、Oracle8 ではこのパラメータは使用されません。

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、次に示す制限事項に違反しないかぎり、データベース・リンクにより他の 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こと。

  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・リンクのために必要な 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 が完了したら、他のサービスまたは RPC が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では、SQL*Net バージョン 2 または Net8 が使われなければなりません。

- 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こと。

これらの制限事項を満たしていると想定して、Oracle Server はそのようなリンクを許可し、トランザクション・プロトコル（準備、ロールバック、コミット）を他の Oracle Server データベースに伝播します。

警告：　これらの制限事項に適合していない場合に、XA トランザクション内でデータベース・リンクを使うと、TPM サーバー・プロセスに接続している 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あるプロセスから別のプロセスへ移動できないので、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すると、ORA#24777 エラーが受信されます。

マルチスレッド・サーバー構成を使用できない場合には、EXEC SQL AT 構文を使用して Pro*C/C++ アプリケーションからリモート・データベースにアクセスします。

パラメータ open_links_per_instance では、移行可能なオープン・データベース・リンク接続の数が指定されます。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を Oracle Parallel Server の任意のインスタンスから回復できます。任意のインスタンスからインダウト・トランザクションをヒューリスティックにコミットすることもできます。XA 回復コールは、すべてのインスタンスについて準備ができた全トランザクションのリストを提供します。

SQL に基づく制限事項

SQL に基づく制限事項をこの項に示します。

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や監視はトランザクション・マネージャが行うので、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 特有の文をアプリケーションに入れないでください。ただし、ローカル・トランザクションではロールバックおよびコミットを使用できます。

グローバル・トランザクションの途中では、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 等価の orol() は実行されません。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は、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 等価の ocom() は実行されません。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 DDL SQL 文は暗黙的なコミットを行うことを暗示しているので、Oracle XA アプリケーションで DDL SQL 文を実行することはできません。

セッションの状態

Oracle では、セッションの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する別のサービスには この変更が検知されないことがあります。1 つのサービス内のセーブポイントを使用してください。アプリケーションは、別のサービスで作成されたセーブポイントを参照してはなりません。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を行ってはなりません。

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 を使用しての接続または切断

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。

その他の XA に関する問題

Oracle XA に関してその他に、次に示す情報について注意してください。

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いずれ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、ロックがトランザクション・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す他のブランチでも確認できます。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ことの利点は、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）。欠点としては、すべてのトランザクション・ブランチがコミットの 2 つのフェーズを通過しなければならないということです。つまり、XA の 1 フェーズ最適化を使用できません。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は、表 18–1 に示されています。

表 18-1　密結合および疎結合トランザクション・ブランチ

属性	密結合ブランチ	疎結合ブランチ
2 フェーズ・コミット	読込み専用最適化	2 フェーズ
	[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]	[全ブランチについて準備およびコミット]
直列化	データベース・コール	なし

結合移行

Oracle Server では、結合移行はサポートされません（結合移行とは、トランザクション・マネージャが中断されているブランチ結合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

init.ora の transactions 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 では、移行可能なオープン・データベース・リンク接続の数が指定されます。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。

> **関連項目：** 詳細は、18-6 ページの「データベース・リンク」を参照してください。

スレッドごとの接続の最大値

スレッドごとの xa_opens の最大値は 32 です。以前の最大値は 8 でした。

インストレーション

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ために xaview.sql スクリプトを実行する必要があります。XA インタフェースを通して 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 Oracle Server にはリリース 7.2 の XA ライブラリを使用しなければなりません。8.3 のライブラリはリリース 7.3 の Oracle Server でも使用できます。リリース 8.0 OCI を使用する XA アプリケーションは、SQL 文を実行する前に sqlld2 を使用して lda_def を取得する場合にだけ、Oracle Server リリース 7.3 でも動作させることができます。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・ハンドルに変換しなければなりません。

基本アーキテクチャ

Oracle XA ライブラリは、Oracle8 Server 以外のトランザクション・マネージャがグローバル・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。これにより、リソース・マネージャ（RM）と呼ばれる非 Oracle8 Server 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。

関連項目： 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。このドキュメントのコピーを希望する場合は、下記宛てに X/Open Document No.XO/CAE/91/300 または ISBN 1 872630 24 3 をお求めください。

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

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・マネージャによって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 18-1 では、X/Open DTP モデルの一例を示しています。

リソース・マネージャ（RM）は、障害発生後に定常状態に戻すことができる、共有および回復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 ログおよびアンドゥ・セグメントを使用して定常状態に復帰します。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。

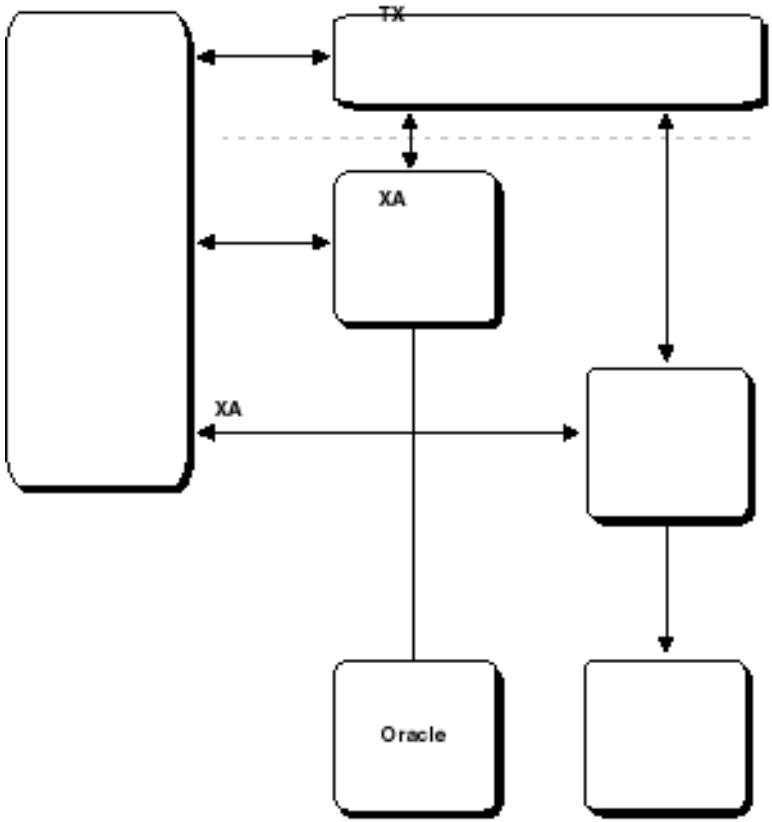
トランザクション・マネージャ（TM）は、トランザクションの境界を指定するためのアプリケーション・プログラム・インタフェース（API）を提供し、コミットおよび回復プロシージャを管理します。

通常、Oracle8 Server は自身が TM として動作し、そのコミットおよび回復を管理します。ただし、標準ベースの 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複数の異機種の RM と協調できます。

TM は、通常は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。トランザクション内のすべての RM を把握しているので、それに基づいて、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 のネイティブなインタフェース （たとえば、SQL）を通して RM のリソース上で動作します。ただし、AP は、トランザクション・マネージャを介して TX と呼ばれるインタフェースから全トランザクション操作を開始し、完了します。AP 自体は、XA インタフェースを直接使用することはありません。

図 18-1　DTP モデルの一例



注意： TX インタフェースおよび対応付けられたサブルーチンの命名規則はベンダー固有で、ここで使用している例とは異なる場合があります。たとえば、tx_open コールは、使用しているシステムでは tp_open と呼ばれているかもしません。使用されている用語について調べたい場合は、トランザクション処理モニターに付属のマニュアルを参照してください。

トランザクション回復管理

Oracle XA ライブラリ・インタフェースは、準備フェーズとコミット・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 にトランザクションのすべての部分をコミットできることを保証するように求めます。これが可能である場合は、RM はその準備ができた状態を記録し、TM に肯定的な返答をします。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 に否定的な返答をし、そのトランザクションに関する知識を消去します。プロトコルにより、アプリケーションまたは RM は、準備フェーズが完了する前にトランザクションを一方的にロールバックできます。

第 2 のフェーズであるコミット・フェーズでは、TM はコミット決定を記録します。その後で、TM はトランザクションに参加しているすべての RM に対してコミットまたはロールバックを発行します。

注意： TM は、すべての RM がフェーズ 1 に対して肯定的な返答をした場合にかぎり、RM のコミットを発行できます。

Oracle XA ライブラリ・インタフェース・サブルーチン

Oracle XA ライブラリ・サブルーチンを使用することで、TM はトランザクションに関する処理を Oracle8 Server に 指示できます。一般に、TM ではリソースを " オープン " しなければなりません（xa_open を使用）。これは、AP で tx_open をコールした結果です。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処理を完了したときには、クローズする必要があります（xa_close を使用）。これは、AP (アプリケーション) で tx_close がコールされるか、アプリケーションを終了するときに行われます。

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を開始し、それを ID に関連付ける
- トランザクションをロールバックする
- トランザクションを準備およびコミットする

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・マネージャへ接続する。
`xa_close`	リソース・マネージャから切断する。
`xa_start`	新しい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) と関連付ける、またはプロセスと既存のトランザクションを関連付ける。
`xa_end`	指定の XID からプロセスを切断する。
`xa_rollback`	指定の 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する。
`xa_prepare`	指定の XID に対応付けられているトランザクションを準備する。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`	指定の XID に対応付けられているトランザクションをコミットする。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`	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準備リストを回復する。
`xa_forget`	指定の 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する。

一般に、AP では、xa_open 文字列で実行されるロールを理解する以外に、これらのサブルーチンについて考慮する必要はありません。

XA インタフェースの拡張

XA インタフェースには、2 つの関数（XA 接続に関連付けられた OCI サービス・ハンドルを戻す関数と XA エラー・コードを戻す関数）が追加されました。

1. OCISvcCtx *xaoSvcCtx(text *dbname):

この関数では、指定した XA 接続の OCI サービス・ハンドルが戻されます。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なければなりません。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、コール sqlld2 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 ライブラリとリンクする必要がありません。サービス・ハンドルは、OCISvcCtxToLda() を使用してバージョン 7 OCI ログオン・データ領域（LDA）に変換できます [Version 8 OCI]。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・ハンドルに変換しなければなりません。

2. OCIEnv *xaoEnv(text *dbname):

   この関数では、指定した XA 接続の OCI 環境ハンドルが戻されます。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なければなりません。

3. int xaosterr(OCISvcCtx *SvcCtx, sb4 error):

   動的登録にだけ適用できるこの関数では、Oracle エラー・コードが XA エラー・コードに変換されます。最初のパラメータは、データベースで作業を行う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。この関数を使用して、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かどうかを判断してください。この関数は、エラーが XA モジュールによって生成されたものでない場合は XA_OK を戻し、エラーが XA モジュールによって生成されたものである場合は有効な XA エラーを戻します。

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、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・サーバーとの間のトランザクション要求のフローを調整します。TPM は基本的に、ネットワーク上で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、さまざまな種類のバックエンド・プロセスに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時期を制御します。このため、分散アプリケーション・プログラムが 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 はこの制御を行います。

TM は分散コミットまたはロールバックを制御するので、Oracle XA ライブラリ・インタフェースを通して 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。

必須のパブリック情報

リソース・マネージャとして、Oracle は次の情報を公表する必要があります。

xa_switch_t 構造体	静的登録の Oracle Server xa_switch_t 構造体名は xaosw です。動的登録の Oracle Server xa_switch_t 構造体名は、xaoswd 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・ポイントと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、Oracle_XA です。
クローズ文字列	xa_close() で使用される文字列は無視され、NULL として扱われます。

オープン文字列	xa_open() で使用されるオープン文字列の形式の詳細は、18-16 ページの「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」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・システムに特有の名前が付いています。TPM 特有のライブラリをリンクしなければならないことを除けば、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。sqllib を使用していない場合は、$ORACLE_HOME/lib/xaonsl.o で確実にリンクしてください。
必要物件	購入し、インストールした分散データベース・オプション

登録

Oracle8 Server では、動的な登録と静的な登録がサポートされます。基本的な可能性を表 18-2 に示します。

表 18-2　XA 登録

クライアント	サーバー	XA 登録
8.0 XA アプリケーション	8.0	動的
8.0 XA アプリケーション	7.3	静的
7.3 XA アプリケーション	8.0	静的

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

このセクションは、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述べています。DBA（またはシステム管理者）とアプリケーション開発者の両方の責任について説明します。また、オープン文字列の構築方法を定義しています。

DBA またはシステム管理者の責任

DBA またはシステム管理者の責任は次のとおりです。

1. アプリケーション開発者の支援によりオープン文字列を定義する。

   これは、18-17 ページの「xa_open 文字列の定義」で説明されています。

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する。

Oracle Server リリース 7.3 の場合 :

V$XATRANS$ が存在することを確認する。

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るはずです。SQL スクリプト XAVIEW.SQL を実行することにより、必要な場合には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 として実行します。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 アカウントの V$XATRANS$ ビューに SELECT 権限を付与します。

関連項目 : XAVIEW.SQL スクリプトの位置については、使用している Oracle オペレーティング・システム固有のマニュアルを参照してください。

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 user の DBA_PENDING_TRANSACTIONS ビューに SELECT 権限を付与します。

3. TPM ベンダーの指示に従い、オープン文字列情報を用いてリソース・マネージャを TPM 構成にインストールする。

   DBA またはシステム管理者は、TPM システムが Oracle8 Server に接続するプロセスを開始することに注意しなければなりません。TPM のマニュアルを参照して、このプロセスのためにどのような環境が存在するのか、およびユーザー 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かめてください。

   次に、そのユーザー ID に XA トレース・ファイルが書き込まれるディレクトリに対する書込み許可を付与します。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、18-17 ページの「xa_open 文字列の定義」を参照してください。

また、ユーザーに DBA_PENDING_TRANSACTIONS に対する SELECT 権限を必ず付与してください。

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開始する。

   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りオープン文字列を定義する。

   オープン文字列の定義方法についてこの項の後の方で説明します。

2. アプリケーションを開発する。

   プリコンパイラで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。18-22 ページの「プリコンパイラおよび OCI へのインタフェース」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する。

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。

このセクション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 には次に示す項目が入ります。

`Acc=P//`

または

`Acc=`*P/user/password*

`SesTm=`*session_time_limit*

optional_fields には次に示す項目が入ります。

`DB=`*db_name*

`LogDir=`*log_dir*

`MaxCur=`*maximum_#_of_open_cursors*

`SqlNet=`*connect_string*

`Loose_Coupling=`*true/false*

`SesWt=`*session_wait_limit*

`Threads=`*true/false*

注意：

- オープン文字列の構築時に、必須のフィールドと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文字と小文字が区別されません。その値の大文字と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「+」文字を使用することはできません。

必須のフィールド

このセクションでは、オープン文字列の必須のフィールドについて説明します。

```
Acc=P//
```

または

```
Acc=P/user/password
```

`Acc`	ユーザー・アクセス情報を指定します。
`P`	明示的なユーザーおよびパスワード情報が提供されることを示します。
`P//`	明示的なユーザーおよびパスワード情報が提供されていないことと、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。 詳細は、『Oracle8 Server 管理者ガイド』を参照してください。
user	有効な Oracle Server アカウント。
password	対応する現行のパスワード。

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。

先に述べたように、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、ユーザーまたは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*

`SesTm`
: システムによって自動的に異常終了されるまで、トランザクションが非アクティブ状態でいられる最大長を指定します。

session_time_limit
: この値は、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の間に、トランザクション内で許容される最大時間にしてください。

  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 の完了と次の RPC または **`tx_commit`**、**`tx_rollback`** の開始の間の時間に適用されます。

  この時間限度の単位は秒です。0 の値は制限のないことを示しますが、0 の値の入力は避けるようにお薦めします。たとえば、`SesTM=`15 は、セッション・アイドル時間制限が 15 秒であることを示します。

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。

db_name Oracle プリコンパイラがデータベースを識別するために使用する名前を示します。

Oracle プリコンパイラのデフォルト・データベースだけを使用する（つまり、SQL 文の 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 `DB=`*db_name* を省略しなければなりません。

明示的に指定したデータベースを使用するアプリケーションは、`DB=`*db_name* フィールドでそのデータベース名を指示する必要があります。

バージョン 7 の OCI プログラムは、正しい lda_def（サービス・コンテキストと等価）を取得するために、sqlld2() 関数をコールする必要があります。バージョン 8 OCI プログラムでは、`xaoSvcCtx` 関数をコールして、OCISvcCtx サービス・コンテキストを取得する必要があります。

db_name は *sid* ではありません。オープンするデータベースを検索するために使用することはできません。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た名前とを関連付けます。*sid* は、TPM アプリケーション・サーバーの環境変数 ORACLE_SID か、オープン文字列の Net8（以前の SQL*Net）句で指定した *sid* から設定されます。Net8 句については、この項の後の方で説明します。

一部の TPM ベンダーは、同じオープン文字列を使用するサーバーのグループを指名する方法を提供しています。DBA では、その用途と *db_name* の両方に対して同じ名前を選択する方が便利な場合があります。

たとえば、DB=payroll はデータベース名が " payroll " であり、アプリケーション・サーバー・プログラムがこの名前を AT 句で使用することを示しています。

LogDir=*log_dir*

LogDir	Oracle XA ライブラリ・エラーおよびトレース情報が記録される、ローカル・マシン上のディレクトリを指定します。
log_dir	トレース情報が格納されるディレクトリのパス名を示します。デフォルトは、ORACLE_HOME が設定されている場合 $ORACLE_HOME/rdbms/log で、設定されていない場合は現行ディレクトリです。

たとえば、LogDir=/xa_trace は、/xa_trace ディレクトリの下にエラーおよびトレース情報があることを示しています。

注意： ロギングに指定したディレクトリが存在し、アプリケーション・サーバーでの書込みが可能であることを確認してください。

MaxCur=*maximum_#_of_open_cursors*

MaxCur	データベースがオープンしているときに割り当てられるカーソルの数を指定します。これは、プリコンパイラ・オプション maxopencursors と同じ用途で機能します。
maximum_#_of_open_cursors	キャッシュするオープン・カーソルの数を示します。

たとえば、MaxCur=5 は、プリコンパイラで 5 つのオープン・カーソルがキャッシュされることを示します。

注意： このパラメータは、プリコンパイラ・オプション maxopencursors がソース・コードで、またはコンパイル時に指定されていた場合に、このオプションを上書きします。

関連項目： `maxopencursors` の詳細は、『`Pro*C/C++` プリコンパイラ・プログラマーズ・ガイド』の第 8 章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	システムにログオ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した構文と同じです。

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 である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。

たとえば、次のとおりです。

```
SqlNet=localsid1
```

ここで

`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・リンクによりアクセスするすべてのデータベースが、/etc/oratab 内にそのエントリを持つようにしてください。

Loose_Coupling=*true/false*

`Loose_Coupling`	ロックをブランチ間で共有するかどうかを指定します。Oracle7 Server へ接続しているときは、このパラメータを true に設定しないでください。デフォルト値は False です。
true/false	ロックがブランチ間で共有されている場合は、設定は false です。

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。

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。

プリコンパイラおよび OCI へのインタフェース

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には、トランザクションの存続期間だけ有効です。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してください。

プリコンパイラとのインタフェースでは、次の 2 つのオプションから選択できます。

- デフォルトのデータベースでプリコンパイラを使用
- 指定のデータベースでプリコンパイラを使用

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プリコンパイラを使用

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用されている DB=*db_name* フィールドが提示されてい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 つのプロセスにつき 1 つのデフォルト接続が許可されます。

次に示すのは、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です。

```
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 SET SAL = sal*1.5;
```

指定のデータベースでプリコンパイラを使用

指定し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_name field を組み込んでください。参照するデータベースはすべて、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、デフォルトのデータベースの他に、1 つまたは複数の指定のデータベースが含まれることがあります。

たとえば、あるデータベースである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。そのような場合は、トランザクション・マネージャで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、次のように宣言を入力します。

```
EXEC SQL DECLARE PAYROLL DATABASE;
EXEC SQL DECLARE MANAGERS DATABASE;
```

もう一度述べますが、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）では、宣言は必要ありません。

更新を行うときは、次に示す文と同様の文を入力します。

```
EXEC SQL AT PAYROLL UPDATE EMP SET SAL=4500 WHERE EMPNO=7788;
EXEC SQL AT MANAGERS UPDATE EMP SET MGR=7566 WHERE EMPNO=7788;
EXEC SQL UPDATE EMP SET DEPTNO=30 WHERE EMPNO=7788;
```

最後の文はデフォルトのデータベースを参照しているので 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...
```

警告：　Oracle では、接続を達成するためにだけ XA アプリケーションを使用することをお薦めしません。実行された作業はすべてグローバル・トランザクションの範囲からはずれることになり、個別にコミットしなければなりません。

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・マネージャにログオンするために OCISessionBegin()(バージョン 7 の olon() または orlon()) をコールしてはいけません。ログオンは、TPM を通して行うようにしてください。このアプリケーションでは、アプリケー

ションがリソース・マネージャにアクセスする必要があるサービス・コンテキスト (バージョン 7 の lda) 構造体を取得するために、関数で xaoSvcCtx() (バージョン 7 の sqlld2()) を実行できます。

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ので、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 を取得します。

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 を取得します。

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);
```

これで、このリソース・マネージャの xaoSvcCtx を取得します。

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。

```
xaoSvcCtx(db_name);
```

これで、このリソース・マネージャの OCISvcCtx を取得します。

> **関連項目：** OCISvcCtx の使用方法の詳細は、『Oracle 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)。

トランザクション制御

このセクションでは、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 ライブラリを使用している場合、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 によって受け入れられた API によって行われます。ほとんどの TM は、このために TX インタフェースを使用します。次の関数があります。

`tx_open`	リソース・マネージャにログインする
`tx_close`	リソース・マネージャからログアウトする
`tx_begin`	新しいトランザクションを開始する
`tx_commit`	トランザクションをコミットする
`tx_rollback`	トランザクションをロールバックする

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、" credit " という名前のサービスがクレジット対象の口座番号と金額を受け取ると、このサービスはデータベースの特定の表内の情報を更新する SQL 文を実行します。さらに、あるサービスがその他のサービスを要求することもあります。たとえば、「transfer fund」サービスは、" credit " および " debit " サービスからの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します。

例に示しているように、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いずれ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 は参加しているプロセス間でトランザクション情報が伝達できるように通信 API を提供しています。通信 API の例として、RPC、擬似 RPC 関数、送信 / 受信関数があります。

リーディング・ベンダーは、異なる通信関数をサポートするので、次の例では、通信 API の例とし通信疑似関数 tpm_service を使用しています。

X/Open では、その準備的な仕様部で通信関数を提供するためのいくつかの代替の方法が取り入れられました。これらの代替の方法のうち、主要な TPM ベンダーはそれぞれ最低でも 1 つの方法を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。すでにアプリケーション・サーバーは、TPM 特有の方法で TPM システムにログオンしていると想定してください。

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た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たトランザクションを示しています。

例 1: アプリケーション・サーバーによって開始されたトランザクション

クライアント：

```
tpm_service("ServiceName");              /* サービスの要求 */
```

サーバー：

```
ServiceName()
{
<get service specific data>
tx_begin();                              /* トランザクションの開始 */
EXEC SQL UPDATE ....;

/* このアプリケーション・サーバーは一時的にクライアントになり */
/* 別のサービスを要求する */

tpm_service("AnotherService");
tx_commit();                             /* トランザクションのコミット */
<return service status back to the client>
}
```

例 2: アプリケーション・サーバーによって開始されたトランザクション

クライアント：

```
tx_begin();                              /* トランザクションの開始 */
tpm_service("Service1");
tpm_service("Service2");
tx_commit();                             /* トランザクションのコミット */
```

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 ライブラリを使用する TPM アプリケーションに移行する場合は、次の操作を行う必要があります。

1. アプリケーションを「サービス」というフレームワークに再編成する。

   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・サービスからのサービスを要求していることを意味します。

   通常、TPM はログオンおよびログオフを暗黙的に行いますが、TPM の一部にはアプリケーションに tx_open 関数および tx_close 関数を使用するように求めるものもあります。

   オープン文字列の sqlnet パラメータを指定しないと、アプリケーションではデフォルトの Net8 ドライバが使用されます。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。詳細は、TPM ベンダーのマニュアルを参照してください。

2. アプリケーションで正規の接続文および切断文を置き換える。

   たとえば、tx_open() によって、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）は、tx_close() で置き換えます。OCISessionBegin() の V7 等価は olon() で、OCISessionEnd() では ologof() でした。

3. アプリケーションで正規のコミット / ロールバック文を置き換え、トランザクションを明示的に開始する。

   たとえば、tx_commit()/tx_rollback() によって、commit/rollback 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() をコールし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する。一般的には、release_cursor=no が使用されます。文が一度だけしか実行されない場合だけ、release_cursor=yes を使用します。

表 18–3 は、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 関数がリストされています。

表 18-3　TPM で置き換えるコマンド

正規の Oracle コマンド	TPM 関数
`CONNECT` *user/password*	tx_open (可能な場合は暗黙的)
トランザクションの明示的な開始	`tx_begin`
`SQL`	SQL を実行するサービス
`COMMIT`	`tx_commit`
`ROLLBACK`	`tx_rollback`
切断	tx_close (可能な場合は暗黙的)
`SET TRANSACTION READ ONLY`	使用できない

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・モニターを使用している場合は、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、特定の事項について注意しておく必要があります。

制御用スレッド（またはスレッド）では、リソース・マネージャとの接続のセットが参照されています。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考えることができます。これは、プロセスには各自、リソース・マネージャへの接続のセットがあり、それぞれが独立したリソース・マネージャ表を保持しているからです。

スレッド化されているシステムでは、各スレッドにリソース・マネージャとの接続の自立走行式セットがあり、プライベートなリソース・マネージャ表がメンテナンスされます。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それぞれについて割り当てる必要があり、またスレッドが終了したときは（異常終了であっても）割当てを解除しなければなりません。

注意： Oracle システムでは、あるスレッドが開始され、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だけです。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。

オープン文字列の仕様

xa_open 文字列パラメータの xa_info では、句 Threads= が指定されます。この句は、トランザクション・モニターでスレッドの使用を可能にするために true に指定されなければなりません。デフォルトは false です。たいていの場合、スレッドはトランザクション・モニターに

よって作成されるということと、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わからないということに注意してください。このため、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た各サービスでは、スタックにサービス・コンテキスト（バージョン 7 では lda）を割り当てることをお薦めします。そのサービスで Oracle 関連のコールを行う場合は、その前に xaoSvcCtx（バージョン 7 の OCI では sqlld2）関数をコールして、サービス・コンテキストを初期化する必要があります。以降は、この LDA をそのサービス内のすべての OCI コールに使用できます。

制限事項

スレッドを使用するとき、次の制限事項が適用されます。

- 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する Pro* または OCI コードは、新しいアプリケーション・スレッドそれぞれがいつ開始されるか明示的にトランザクション・モニターに通知されない場合は、スレッド化できません。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。

障害追求

このセクションでは、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について説明します。また、トレース・ファイルについてと、保留中のトランザクションの回復についても説明します。

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・コードを補足するために役立ちます。たとえば、xa_open 障害が、誤ったオープン文字列または Oracle Server インスタンスの検索失敗、ログイン認可の失敗が原因であるかどうか示します。

トレース・ファイルの名前は次のとおりです。

```
xa_db_namedate.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* を指定しないと、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

通常は、XA トレース・ファイルは、エラーが検出された場合にだけオープンされます。xa_open 文字列 DbgFl では、XA ライブラリについてのその詳細を記録するトレース機能が提供されます。デフォルト値は 0 です。次の値の組合せに設定できます。これらの組合せは独立していて、複数のフラグを印刷出力するには、それぞれを設定する必要があります。

- 0x1 XA インタフェースで各プロシージャの入口と終了をトレースします。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・ルーチンの入口と終了をトレースします。これは普通、Oracle 開発者専用の機能です。
- 0x4 Oracle コール・インタフェースの特定のコールのような、XA ライブラリで作成された他の様々な興味深いコールをトレースします。これは普通、Oracle 開発者専用の機能です。

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置くことができます。

- 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 を指定しなかった場合、Oracle XA アプリケーションは、$ORACLE_HOME の位置を検出できたら、$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 の指定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

次にトレース・ファイルについて 2 種類の例をあげて説明します。

例 xa_NULL040292.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 フィールドはオープン文字列に指定されませんでした。

例 xa_Finance121591.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 " と指定されました。

注意： オープン文字列に同じ DB フィールドと 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、同じ日に発生した全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_INVAL; logon denied
```

ここでは、" 1032 " は、情報が記録されたときの時刻、" 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_INVAL は XA 標準の指定により戻されたエラー、そして ORA-**1017** 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、回復します。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上書きしなければならないことがあります。

- 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
- 十分と思われる時間が経っても解決されない

上記のような状況でインダウト・トランザクションを上書きする方法について、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については、TPM のマニュアルを参照してください。

Oracle Server SYS アカウント表

`Oracle Server SYS` アカウントの下には 4 つの表があり、正規の `Oracle Server` アプリケーションや `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` および `V$GLOBAL_TRANSACTIONS`、`DBA_2PC_PENDING`、`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、次に示す列情報が特に DBA_2PC_NEIGHBORS 表に適用されます。

- `DBID` 列は常に `xa_orcl` である
- `DBUSER_OWNER` 列は常に `db_namexa.oracle.com` である

db_name は常に、オープン文字列で 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 NULLxa.oracle.com になります。

たとえば、Oracle XA アプリケーションで生成したインダウト・トランザクションの詳細な情報を取得するために、以下の SQL 文を使用できます。

```
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 を知っている場合は、DBA_PENDING_TRANSACTIONS または V$GLOBAL_TRANSACTIONS を使用できます。DBA_PENDING_TRANSACTIONS ではアクティブな準備済みトランザクションと失敗した準備済みトランザクションの両方のリストが指定されますが、V$GLOBAL_TRANSACTIONS ではすべてのアクティブなグローバル・トランザクションが指定されます。

索引

記号

%ROWTYPE 属性 10-7, 15-3
　ストアド・ファンクションで使用される 10-8
%TYPE 属性 10-7, 15-3

数字

1 対 1 関連
　外部キーとの 9-9
1 対多関連
　外部キーとの 9-9

A

ADMIN オプション 17-12
ADVISE_COMMIT プロシージャ 10-59
ADVISE_NOTHING プロシージャ 10-59
ADVISE_ROLLBACK プロシージャ 10-59
AFTER トリガー
　監査および～ 13-21, 13-23
　指定 13-3
　相関名および～ 13-8
ALL_ERRORS ビュー
　ストアド・プロシージャをデバッグする 10-33
ALL_SOURCE ビュー 10-33
ALTER CLUSTER コマンド 4-6
　ALLOCATE EXTENT オプション 4-36
ALTER FUNCTION コマンド 15-5
ALTER INDEX コマンド 4-6
ALTER PACKAGE コマンド 15-5
ALTER PROCEDURE コマンド 15-5
ALTER SEQUENCE コマンド 4-23
ALTER SESSION コマンド
　SERIALIZABLE 3-15, 3-29
ALTER TABLE コマンド 4-6, 4-8
　DISABLE ALL TRIGGERS オプション 13-18
　DISABLE 整合性制約オプション 9-20
　DROP 整合性制約オプション 9-23
　ENABLE ALL TRIGGERS オプション 13-18
　ENABLE 整合性制約オプション 9-19
　INITRANS パラメータ 3-29
　整合性制約を定義する 9-16
ALTER TRIGGER コマンド 15-5
　DISABLE オプション 13-18
　ENABLE オプション 13-18
ALTER VIEW コマンド 15-4
ALTER_COMPILE プロシージャ 10-57
ANALYZE_OBJECT プロシージャ 10-57
ANALYZE_PART_OBJECT プロシージャ 10-60
ANALYZE_SCHEMA プロシージャ 10-60
ANSI SQL92
　FIPS フラグ付け 3-2
ANSI（米国規格協会）
　ANSI 互換のロック 3-15
AQ オブジェクトの削除 11-60

B

BEFORE トリガー
　指定 13-3
　相関名および～ 13-8
　導出列値 13-33
　複雑なセキュリティ認可 13-31
BEGIN_DISCRETE_TRANSACTION プロシージャ 10-59
BFILE 6-5
　コピーする 6-36
　最大オープン数 6-19
　初期化 6-14

マルチスレッド・サーバー（MTS）6-19
BFILENAME() 6-14, 6-55, 6-56
BFILE データ型 6-8
BFILE に対する参照方法 6-15
BIND_ARRAY プロシージャ 14-6, 14-11
BIND_VARIABLE プロシージャ 14-6, 14-11
BLOB データ型 6-6

C

CACHE / NOCACHE 6-10
CACHE オプション
CREATE SEQUENCE コマンド 4-26, 4-27
CASCADE オプション
整合性制約 4-37
CASE ツール 1-3
CAST 演算子 8-5
CATPROC.SQL ファイル 3-18, 12-21, 13-3
CC 日付書式 5-9
CHARARR データ型
DBMS_OUTPUT の 12-24
CHARTOROWID 関数 5-17
CHAR データ型 5-2, 5-5
どんな場合に使用するか 5-5
列長さを大きくする 4-8
列の長さ 5-6
CHECK 制約
NOT NULL 制約および～ 9-14
設計する 9-14
データの整合性 9-19
トリガーおよび～ 13-25, 13-30
どんな場合に使用するか 9-13
～の数 9-14
～の制限 9-13
～を使用して NULL を制限する 9-14
CHUNK 6-11
CLOB データ型 6-6
NCLOB 6-7, 6-63
CLOSE_CURSOR プロシージャ 14-7, 14-26
CLOSE_DATABASE_LINK プロシージャ 10-58
COLUMN_VALUE_LONG プロシージャ 14-7, 14-22
COLUMN_VALUE プロシージャ 14-7, 14-20
COLUMN_VALUE、～で名前なしのネストした表を表す 7-18
COMMIT_COMMENT プロシージャ 10-59
COMMIT_FORCE プロシージャ 10-59
COMMIT コマンド 3-5
COMMIT プロシージャ 10-59
COMPILE_SCHEMA プロシージャ 10-60
COMPILE オプション
ALTER PROCEDURE コマンドの～ 15-5
CREATE CLUSTER コマンド 4-6, 4-35
HASH IS オプション 4-39
HASHKEYS オプション 4-39
ハッシュ・クラスタ 4-38
CREATE INDEX コマンド 4-6, 4-32
ON CLUSTER オプション 4-35
CREATE PACKAGE BODY コマンド 10-12
CREATE PACKAGE コマンド 10-12
CREATE ROLE コマンド 17-8
CREATE SCHEMA コマンド 4-41
必要な権限 4-41
CREATE SEQUENCE コマンド
CACHE オプション 4-23, 4-26
NOCACHE オプション 4-27
例 4-27
CREATE TABLE コマンド 4-3, 4-6
CLUSTER オプション 4-35
整合性制約を定義する 9-15
～の INITRANS パラメータ 3-29
CREATE TRIGGER コマンド 13-2
REFERENCING オプション 13-9
CREATE TYPE 文 7-9
CREATE VIEW コマンド 4-10
OR REPLACE オプション 4-12
WITH CHECK OPTION 4-10, 4-13
CREATE_PIPE プロシージャ 12-4
CURRVAL 擬似列 4-24
制限 4-25
C 構造体の配列 2-6

D

DATE データ型 5-7
世紀 5-8
データ変換 5-17
DBA_ERRORS ビュー
ストアド・プロシージャをデバッグする 10-33
DBA_QUEUES 11-91
DBA_QUEUE_TABLES 11-90
DBA_ROLE_PRIVS ビュー 17-3
DBA_SOURCE ビュー 10-33
DBMS_ALERT パッケージ 10-62
作成 16-3

～について 16-2
DBMSALRT.SQL ファイル 16-3
DBMS_APPLICATION_INFO パッケージ 10-62
DBMS_AQ.DEQUEUE 11-72
DBMS_AQ.ENQUEUE 11-70
DBMS_AQADM.ADD_SUBSCRIBER 11-84
DBMS_AQADM.CREATE_QUEUE 11-78
DBMS_AQADM.CREATE_QUEUE_TABL 11-76
DBMS_AQADM.DROP_QUEUE 11-80
DBMS_AQADM.DROP_QUEUE_TABLE 11-79
DBMS_AQADM.QUEUE_SUBSCRIBER 11-92
DBMS_AQADM.START_QUEUE 11-81
DBMS_AQADM.STOP_QUEUE 11-83
DBMS_AQADM パッケージ 10-63
DBMS_AQ パッケージ 10-63
DBMS_DDL パッケージ 10-56, 10-57
DBMS_DEFER_QUERY パッケージ 10-64
DBMS_DEFER_SYS パッケージ 10-64
DBMS_DEFER パッケージ 10-64
DBMSDESC.SQL ファイル 10-65
DBMS_DESCRIBE パッケージ 10-62, 10-65
作成 10-65
DBMS_DISTRIBUTED_TRUST_ADMIN パッケージ 10-63
DBMS_HS_EXTPROC パッケージ 10-63
DBMS_HS_PASSTHROUGH パッケージ 10-63
DBMS_HS パッケージ 10-63
DBMS_JOB パッケージ 10-62
DBMS_LOB.APPEND() 6-68
DBMS_LOB.COMPARE() 6-69
DBMS_LOB.COPY() 6-72
DBMS_LOB.ERASE() 6-74
DBMS_LOB.FILECLOSE() 6-76
DBMS_LOB.FILECLOSEALL() 6-77
DBMS_LOB.FILEEXISTS() 6-77
DBMS_LOB.FILEGETNAME() 6-79
DBMS_LOB.FILEISOPEN() 6-80
DBMS_LOB.FILEOPEN() 6-81
DBMS_LOB.GETLENGTH() 6-82
DBMS_LOB.LOADFROMFILE() 6-86
DBMS_LOB.READ() 6-88
DBMS_LOB.SUBSTR() 6-91
DBMS_LOB.TRIM() 6-93
DBMS_LOB.WRITE() 6-94
DBMS_LOB パッケージ 6-62, 10-63
BFILES の使用方法 6-65
一般的な使用方法 6-65
セキュリティ 6-65
定数 6-63
マルチスレッド・サーバー（MTS）6-19
ルーチン 6-62
データ型 6-62
例外 6-64
DBMSLOCK.SQL ファイル 3-18
DBMS_LOCK パッケージ 3-17, 10-62
SLEEP プロシージャ 3-24
作成 3-18
セキュリティ 3-18
DBMSOTPT.SQL ファイル 12-21
DBMS_OUTPUT パッケージ 10-62, 12-20
GET_LINE プロシージャ 12-21
NEW_LINE プロシージャ 12-21
PUT_LINE プロシージャ 12-21
PUT プロシージャ 12-21
作成 12-21
例 12-24
DBMSPIPE.SQL ファイル 12-3
DBMS_PIPE パッケージ 10-62, 12-2
作成 12-3
DBMS_REFRESH パッケージ 10-64
DBMS_REPCAT_ADMIN パッケージ 10-64
DBMS_REPCAT_AUTH パッケージ 10-64
DBMS_REPCAT パッケージ 10-64
DBMS_ROWID パッケージ 10-63
DBMS_SESSION パッケージ 10-56, 10-58
DBMS_SHARED_POOL パッケージ 10-62
DBMS_SNAPSHOT パッケージ 10-64
DBMS_SPACE パッケージ 10-62
DBMSSQL.SQL ファイル 14-2
DBMS_SQL パッケージ 10-62, 14-2
作成 14-2
ファンクション 14-4, 14-8
DBMS_SYSTEM パッケージ 10-62
DBMS_TRANSACTION パッケージ 10-56, 10-59
DBMS_UTILITY パッケージ 10-56, 10-60
DDL 文
動的 SQL 14-2
パッケージの状態と～ 10-14
DECLARE
ストアド・プロシージャで使用されない 10-8
DEFINE_ARRAY ファンクション 14-17
DEFINE_ARRAY プロシージャ 14-6
DEFINE_COLUMN_LONG プロシージャ 14-6, 14-18
DEFINE_COLUMN プロシージャ 14-6, 14-15

DELETE コマンド
　参照整合性のためのトリガー 13-27, 13-28
　データ整合性 3-10
　列値とトリガー 13-8
DEREF 演算子 7-13
DESCRIBE_COLUMNS プロシージャ 14-24
DESCRIBE_PROCEDURE プロシージャ 10-66
DESC_TAB データ型 14-26
Designer/2000 1-3, 2-3
Developer/2000 1-3
DIRECTORY 名の指定 6-16
DISABLE プロシージャ 12-21, 12-22
DMBS_ROWID パッケージ 10-74
DMBS_SQL パッケージ
　エラー位置を見つける 14-28
DML_LOCKS パラメータ 3-11
DROP CLUSTER コマンド 4-37, 4-39
DROP INDEX コマンド 4-33
　必要な権限 4-33
DROP ROLE コマンド 17-12
DROP TABLE コマンド 4-9
DROP TRIGGER コマンド 13-17

E

EMPTY_BLOB() 関数 6-55
EMPTY_CLOB() 関数 6-55
ENABLE プロシージャ 12-21, 12-22
EXECUTE_AND_FETCH ファンクション 14-6, 14-19
EXECUTE ファンクション 14-6, 14-19

F

FCLOSE_ALL プロシージャ 12-34
FCLOSE プロシージャ 12-33
FETCH_ROWS ファンクション 14-6, 14-19
FFLUSH プロシージャ 12-39
FIPS フラグ付け
　対話型 SQL 文および～ 3-2
FIXED_DATE 初期化パラメータ 5-8
FOPEN ファンクション 12-31
FOR EACH ROW 句 13-6
FOR UPDATE 句
　LOB 6-22
FOREIGN KEY 制約
　1 対 1 関連 9-9
　1 対多関連 9-9
　NOT NULL 制約および～ 9-9
　UNIQUE キー制約および～ 9-9
　親表を参照する行の数 9-9
　使用可能にする 9-19, 9-25
　定義 9-23
　定義する 9-24
　表を更新する 9-10, 9-11
FORMAT_CALL_STACK ファンクション 10-61
FORMAT_ERROR_STACK ファンクション 10-60, 10-61
FREE_UNUSED_MEMORY プロシージャ 10-58

G

GET_LINES プロシージャ 12-21, 12-23
GET_LINE プロシージャ 12-21, 12-23, 12-34
GRANT コマンド 17-12
　ADMIN オプション 17-12
　WITH GRANT オプション 17-13
　いつ有効になるか 17-17
　オブジェクト権限 17-13
　システム権限 17-12

H

HEXTORAW 関数 5-17
HTTP コールアウト 10-64, 10-82

I

ICX
　UTL_HTTP パッケージ 10-82
IN OUT パラメータ・モード 10-6
INIT.ORA パラメータ 11-61
INITRANS パラメータ 3-29
INSERT コマンド
　読取り一貫性 3-10
　列値とトリガー 13-8
INSTEAD OF トリガー 8-5, 13-4
IN パラメータ・モード 10-6
ISOLATION LEVEL
　SERIALIZABLE 3-29
　変更 3-29
IS_OPEN ファンクション 12-32, 14-24
IS_ROLE_ENABLED ファンクション 10-58

L

LAST_ERROR_POSITION ファンクション 14-28
LAST_ROW_COUNT ファンクション 14-28
LAST_ROW_ID ファンクション 14-28
LAST_SQL_FUNCTION_CODE ファンクション 14-28
LBS
　LOB バッファリング・サブシステムを参照
LOB 6-1
　DBMS_LOB パッケージ 6-62
　EMPTY_BLOB() 6-55
　EMPTY_CLOB() 6-55
　LOB ロケータ 6-22
　NULL に設定する 6-13
　performance, best practices 6-52
　値 6-20
　インライン記憶域 6-20
　オブジェクト・キャッシュ 6-43
　オブジェクト・キャッシュの 6-43
　外部 LOB
　　コピーする 6-36
　　削除する 6-36
　可変幅文字データ 6-7, 6-53
　更新済み LOB ロケータ 6-25
　削除する 6-36
　代表的な使用例 6-4
　～で SELECT を実行する 6-22
　定義 6-5
　内部 LOB 6-5
　　CACHE / NOCACHE 6-10
　　CHUNK 6-11
　　ENABLE | DISABLE STORAGE IN ROW 6-12
　　LOGGING / NOLOGGING 6-10
　　PCTVERSION 6-9
　　空に設定する 6-14
　　更新する前にロックする 6-22
　　コピーする 6-35
　　削除する 6-36
　　初期化 6-13
　　表領域と LOB 索引 6-9
　　表領域と記憶特性 6-8
　　ロケータ 6-20
　バインド変数 6-29
　バッファリング
　　caveats 6-44
　　エージング処理ができるページ 6-47
　バッファリング操作 6-45
　バッファリング・サブシステム 6-43
　ピース単位の操作 6-6, 6-26
　フラッシュ 6-44
　読取り一貫性のあるロケータ 6-22
　ロケータ 6-20
　ロケータによってアクセスする 6-22
　ロケータを含むように設定する 6-21
LOBS
　外部 (BFILE) 6-5
LOB の値 6-20
LOB バッファリング・システム（LBS）6-43
LOB¨ 6-44
LOB ロケータは複数のトランザクションにまたがることはできない 6-31
LOB を NULL に設定する 6-13
LOB をコピーする 6-35
　外部 6-36
　内部 LOB 6-35
LOB を削除する 6-36
LOCAL_TRANSACTION_ID ファンクション 10-59
LOCK TABLE コマンド 3-11
LOGGING / NOLOGGING 6-10
LONG RAW データ型 5-10, 5-11
　トリガーでの使用 13-11
　～の制限 5-10
LONG データ型 5-9
　トリガーでの使用 13-11
　～の制限 5-10

M

MAKE_REF 演算子 8-4
MAX_ENABLED_ROLES パラメータ
　デフォルトのロールおよび～ 17-10
MAXTRANS オプション 4-6
MLSLABEL データ型 5-15
MULTISET 演算子 8-5

N

NAME_RESOLVE プロシージャ 10-61
NCHAR データ型 5-2, 5-5
NCLOB データ型 6-7
NESTED_TABLE_ID 非表示列 7-19
NEW
　相関名 13-8
NEW_LINE プロシージャ 12-21, 12-36

NEXT_ITEM_TYPE ファンクション 12-10
NEXTVAL 擬似列 4-24
　制限 4-25
NLS_DATE_FORMAT パラメータ 5-8
NOCACHE オプション
　CREATE SEQUENCE 文 4-27
NOT NULL 制約
　CHECK 制約および～ 9-14
　データの整合性 9-19
　どんな場合に使用するか 9-3
NOWAIT オプション 3-11
NUMBER データ型 5-7
NVARCHAR2 データ型 5-2, 5-5

O

Object Database Designer 2-3
OCI
　「Oracle コール・インタフェース」を参照。
OLD
　相関名 13-8
OPEN_CURSORS パラメータ 3-9
OPEN_CURSOR ファンクション 14-4, 14-9
OR REPLACE 句
　パッケージを作成するための 10-12
Oracle Procedure Builder 1-3
Oracle Web Server カートリッジ 10-82
Oracle アドバンスト・キューイング（Oracle AQ）11-1
　DBMS_AQADM パッケージ 11-74
Oracle エラー 10-3
Oracle が提供するパッケージ 10-56, 10-61
　説明箇所 1-5
Oracle コール・インタフェース 10-2
　アプリケーション 10-4
　カーソルの取消し 3-10
　カーソルをクローズする 3-10
　機能 2-6
Oracle プリコンパイラ
　ストアド・プロシージャとパッケージをコールする
　　10-35
ORDBMS 7-2
　Object Database Designer 2-3
OUT パラメータ・モード 10-6

P

PACK_MESSAGE プロシージャ 12-6
PARSE プロシージャ 14-4, 14-10
　大規模な SQL 文を解析する 14-11
pcode
　トリガーのために生成されるとき 13-16
PCTFREE 記憶領域パラメータ
　クラスタ化されてない表 4-5
　設定のためのガイドライン 4-4, 4-5, 4-6
　デフォルト 4-4
　～の索引 4-5
　ブロック・オーバーヘッドおよび～ 4-6
　変更する 4-8
PCTUSED 記憶パラメータ
　クラスタ化されてない表 4-6
　設定のためのガイドライン 4-6
　デフォルト 4-6
　ブロック・オーバーヘッドおよび～ 4-6
　変更する 4-8
PCTVERSION 6-9
PL/SQL 10-2
　DBMS_LOB パッケージ 6-62
　RAISE 文 10-29
　カーソル変数 10-24
　コードを隠すためのラッパー 10-27
　ソース・コードを隠す 10-27
　逐次再使用可能パッケージ 10-48
　データ型 10-70
　　～の数値コード 10-72
　データ・ディクショナリ・ビュー 10-73
　動的 SQL 14-2
　トリガー本体 13-8
　パッケージ 10-10
　表 10-8
　　レコードの 10-8
　ファイル I/O 12-27
　　セキュリティ 12-29
　ファンクション
　　RESTRICT_REFERENCES プラグマ 10-44
　　オーバーロードする 10-48
　　純粋さレベル 10-43
　　使用する 10-40
　　パラメータのデフォルト値 10-42
　　引数 10-42
　プログラム・ユニット 10-2
　　置き換えられたビューおよび～ 4-13
　　削除された表および～ 4-9
　無名ブロック 10-2
　ユーザー定義エラー 10-29

ライブラリ・ユニット間の依存性 10-14
リモート・ストアド・プロシージャをコールする 10-38
例外ハンドラ 10-2
PL/SQL コードを隠す 10-27
PL/SQL コードを隠すためのラッパー 10-27
PL/SQL のプログラム・ユニット 10-2
PRIMARY KEY 制約
UNIQUE キー制約対～ 9-6
主キーを選択する 9-5
使用可能にする 9-19
使用禁止にする 9-19
～における複数の列 9-6
変更する 9-22
Pro*C デーモン 12-17
PUBLIC ユーザー・グループ
プロシージャおよび～ 17-17
～への権限付与と取消し 17-17
PURGE_MIXED プロシージャ 10-59
PUTF プロシージャ 12-37
PUT_LINE プロシージャ 12-21, 12-22, 12-37
最大出力サイズ 12-37
PUT プロシージャ 12-21, 12-22, 12-35
最大出力サイズ 12-37

R

RAISE_APPLICATION_ERROR プロシージャ 10-27
リモート・プロシージャ 10-31
RAISE 文 10-29
RAWTOHEX 関数 5-17
RAW 型のキュー表およびキューの作成 11-27
RAW データ型 5-11
READ_ONLY プロシージャ 10-59
READ_WRITE プロシージャ 10-59
RECEIVE_MESSAGE ファンクション 12-8
REF, オブジェクト識別子からの組立て 8-4
REFERENCING オプション 13-9
REF、の暗黙的デリファレンス 7-14, 8-6
REF、～のデリファレンス 7-14
REF、有効範囲を設定した 7-18
REGISTER プロシージャ 16-5
REMOTE_DEPENDENCIES_MODE パラメータ 10-22
REMOVE_PIPE プロシージャ 12-11
REMOVE プロシージャ 16-5
RENAME コマンド 4-43
RESET_PACKAGE プロシージャ 10-58
RESTRICT_REFERENCES プラグマ
可変 10-46
～の構文 10-44
副次作用の制御に使用する 10-44, 10-46
REVOKE コマンド 17-14
いつ有効になるか 17-17
RNDS 引数 10-44
RNPS 引数 10-44
ROLE_SYS_PRIVS ビュー 17-3
ROLE_TAB_PRIVS ビュー 17-3
ROLLBACK_FORCE プロシージャ 10-59
ROLLBACK_SAVEPOINT プロシージャ 10-59
ROLLBACK コマンド 3-5
ROLLBACK プロシージャ 10-59
ROWIDTOCHAR 関数 5-17
ROWID データ型 5-12
DBMS_ROWID パッケージ 10-74
移行 5-14
拡張 ROWID 形式 5-12
拡張形式 10-80
ROWLABEL 列 5-15
ROW_LOCKING パラメータ 3-11, 3-15
ROWTYPE_MISMATCH 例外 10-27
RR 日付書式 5-9
RS ロック
LOCK TABLE コマンド 3-12
RX ロック
LOCK TABLE コマンド 3-12

S

SAVEPOINT コマンド 3-6
SAVEPOINT プロシージャ 10-59
SELECT コマンド
FOR UPDATE 6-22
SELECT ... FOR UPDATE 3-14
読取り一貫性 3-10, 6-22
SELF キーワード 7-13
SEND_MESSAGE ファンクション 12-7
SEQUENCE_CACHE_ENTRIES パラメータ 4-26
SERIALIZABLE オプション 3-15
ISOLATION LEVEL の 3-29
SERIALIZABLE パラメータ 3-11
SERIALLY_REUSABLE プラグマ 10-50
Server Manager
DMBS_OUTPUT メッセージ 12-23
出力用 ENABLE プロシージャ 12-22

SESSION_MAX_OPEN_FILES パラメータ 6-19
SET ROLE コマンド 17-3, 17-10
　オペレーティング・システム権限を使用する場合 17-11
SET TRANSACTION コマンド 3-8
　ISOLATION LEVEL 句 3-29
　SERIALIZABLE 3-15, 3-29
SET_CLOSE_CACHED_OPEN_CURSORS プロシージャ 10-58
SET_DEFAULTS プロシージャ 16-8
SET_NLS プロシージャ 10-58
SET_ROLE プロシージャ 10-58
SET_SQL_TRACE プロシージャ 10-58
SGA
　システム・グローバル領域を参照
SIGNAL プロシージャ 16-5
SLEEP プロシージャ 3-24
SORT_AREA_SIZE パラメータ
　索引の作成および～ 4-29
SQL DDL
　BFILE セキュリティ 6-17
SQL DML
　BFILE セキュリティ 6-17
SQL*Loader
　索引および～ 4-30
SQL*Module
　アプリケーション 10-4
　ストアド・プロシージャを～からコールする 10-5
SQL*Plus
　DMBS_OUTPUT メッセージ 12-23
　SET SERVEROUTPUT ON コマンド 10-3
　SHOW ERRORS コマンド 10-32
　コンパイル時エラー 10-32
　出力用 ENABLE プロシージャ 12-22
　順序を作成する 10-12
　ストアド・プロシージャを呼び出す 10-34
　プロシージャをロードする 10-9
　無名ブロック 10-4
SQL 文
　32 KB より大きい 14-11
　PL/SQL でのアクセス 10-56
　アプリケーションの設計および～ 2-7
　実行 3-2
　動的 SQL 14-2
　トリガーでは許可されていない 13-11
　トリガー本体における～ 13-8, 13-11
　どんな場合に制約チェックが発生するか 9-15
　～に必要な権限 17-8
　パススルー SQL 10-63
SRX ロック
　LOCK Table コマンド 3-13
STEP_ID ファンクション 10-59
SYSDATE 関数 5-8
S ロック
　LOCK TABLE コマンド 3-12

T

TO_CHAR 関数 5-17
　CC 日付書式 5-9
　RR 日付書式 5-9
　Trusted Oracle ラベルを変換する 5-19
TO_DATE 関数 5-8, 5-17
　RR 日付書式 5-9
TO_LABEL 関数
　Trusted Oracle ラベルを変換する 5-19
TO_NUMBER 関数 5-17
TRUNCATE TABLE コマンド 4-9
TRUNC 関数 5-8
Trusted Oracle Server
　MLSLABEL データ型 5-15
　Trusted Database List のメンテナンス 10-63
　動的 SQL 14-7
　ラベルを変換する 5-19

U

UNIQUE_SESSION_ID ファンクション 10-58
UNIQUE キー制約
　NOT NULL 制約と組み合せる 9-4
　PRIMARY KEY 制約対～ 9-6
　使用可能にする 9-19
　使用禁止にする 9-19
　データの整合性 9-22
　どんな場合に使用するか 9-6
　複合キーおよび NULL 9-6
　変更する 9-22
UNPACK_MESSAGE プロシージャ 12-10
UPDATE コマンド
　参照整合性のためのトリガー 13-27, 13-28
　データ整合性 3-10
　トリガーおよび～ 13-6, 13-9
　列値とトリガー 13-8
USER_ERRORS ビュー

ストアド・プロシージャをデバッグする 10-33
USE_ROLLBACK_SEGMENT プロシージャ 10-59
USER_QUEUES 11-91
USER_QUEUE_TABLES 11-91
USER_SOURCE ビュー 10-33
USER 関数 9-4
UTLDTREE.SQL ファイル 15-2, 15-7
UTLEXCPT.SQL ファイル 9-21
UTL_FILE パッケージ 12-27
セキュリティの問題 12-28
UTL_HTTP パッケージ 10-64, 10-82
UTLLOCKT.SQL スクリプト 3-25

V

VARCHAR2S データ型 14-11
VARCHAR2 データ型 5-2, 5-5
どんな場合に使用するか 5-5
列の長さ 5-6
VARCHAR データ型 5-5
VARIABLE_VALUE プロシージャ 14-7, 14-23
VARRAY 対ネストした表 7-9, 7-11
VARRAY、「配列」を参照

W

WAITANY プロシージャ 16-6
WAITONE プロシージャ 16-7
WHEN 句 13-7
EXCEPTION の例 13-10, 13-26, 13-30, 13-31
PL/SQL 式を含むことができない 13-8
相関名 13-8
例 13-2, 13-7, 13-19, 13-26
WITH GRANT OPTION 17-13
WNDS 引数 10-44
WNPS 引数 10-44

X

X ロック
LOCK TABLE コマンド 3-14

あ

アクセス
オブジェクト
順序 4-24
スキーマ・オブジェクト
権限を取り消す 17-14
権限を付与する 17-13
トリガー 13-2, 13-31
リモート整合性制約 9-13
データベース
権限を取り消す 17-14
権限を付与する 17-12
アドバンスト・キューイング 11-1
DBMS_AQADM パッケージ 11-74
エラー・メッセージ 11-93
管理インタフェース 11-74
権限およびアクセス制御 11-74
列挙定数 11-88
管理項目 11-89
機会の枠 11-5
機能 11-7
SQL アクセス 11-7
エンキューでのメッセージの優先順位と順序付け 11-9
オプションのトランザクション保護 11-10
構造化されたペイロード 11-8
時間の指定 11-9
相関識別子 11-8
遅延による再試行 11-10
追跡とイベント・ジャーナル 11-8
デキューでのメッセージのナビゲーション 11-10
デキューのモード 11-10
統合されたトランザクション 11-8
統合データベース・レベルのサポート 11-8
複数の宛先 11-9
保存とメッセージ履歴 11-8
メッセージ待機の最適化 11-10
メッセージのグループ化 11-9
予約リストと宛先リスト 11-9
例外処理 11-10
キューオプション
エンキュー・オプション 11-67
デキュー・オプション 11-68
キュー表およびキューの作成 11-26
操作インタフェース 11-69
メッセージの検索基準とデキュー順序 11-72
データ構造
オブジェクト名 11-63
データベース・オブジェクト 11-89
デモの参照先 11-99
典型的アプリケーション 11-3

メッセージの順序 11-5
メッセージの遅延実行 11-4
メッセージの保護 11-4
メッセージ・システムの要件 11-6
メッセージ・プロパティ 11-64
ロールおよび権限の取消し 11-61
アドバンスト・キューイング , 基本 11-14
アドバンスト・キューイング , 同一メッセージの複数コンシューマ・デキュー 11-17
アドバンスト・キューイングの機能 11-7
アプリケーション
ストアド・プロシージャとパッケージをコールする 10-35
セキュリティ 17-2, 17-5
設計する 2-2, 2-4
調整 2-7
データベース設計 2-2
～の未処理例外 10-30
メンテナンスする 2-8
ロール 17-3
アプリケーションを更新する 2-8
アプリケーションを設計する 2-4
必要性を評価する 2-2
アプリケーションをメンテナンスする 2-8
アラート 16-2
暗黙的デリファレンス 7-14, 8-6

い

異機種間サービス 10-63
パススルー SQL 10-63
分散外部プロシージャのセキュリティ 10-63
移行
ROWID 形式 5-14
依存性
PL/SQL ライブラリ・オブジェクト間の～ 10-14
格納されたトリガーの～ 13-16
スキーマ・オブジェクト
UTLDTREE.SQL 15-7
トリガー管理 13-11
タイムスタンプ・モデル 10-15
～に関する情報をリストする 15-6
一時セグメント
索引の作成および～ 4-29
一貫性
読取り専用トランザクション 3-8
イベント
アラートで通知する 16-2
イベントとしてのメッセージ 11-4
インターネット・データ 10-64, 10-82
インタフェース
操作 , アドバンスト・キューイング用 11-69

う

埋込み SQL 10-2

え

エージェント , 定義 11-11
エクステント
削除された表および～ 4-9
割り当てる 4-36
エラー
DBMS_ALERT パッケージによって戻される 16-3
DBMS_DESCRIBE パッケージによって戻される 10-65
DBMS_OUTPUT パッケージにより戻される 12-21
DBMS_PIPES パッケージによって戻される 12-4
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 10-28
エラーを伴うビューを作成する 4-11
データ・ディクショナリ・ビュー 10-73
動的 SQL で位置を見つける 14-28
ユーザー定義 10-27, 10-29
リモート・プロシージャ 10-31
エンティティ・リレーションシップ（E-R）モデル 2-2

お

オーバーロードする
RESTRICT_REFERENCES を使用して 10-48
ストアド・プロシージャ名 10-5
パッケージ・ファンクションの～ 10-48
オブジェクト型のキュー表およびキューの作成 11-26
オブジェクト型のメソッド 7-12
オブジェクト型、～の比較メソッド 7-12
オブジェクト型、～のメソッド 7-12
オブジェクト型、不完全な 7-9
オブジェクト表 7-14
オブジェクトを改名する 4-43
オブジェクト、スキーマ
オブジェクト権限を取り消した場合 17-16
改名する 4-43

権限を取り消す 17-14
権限を付与する 17-13
情報をリストする 4-44
名前変換 4-42
オブジェクト・キャッシュ 6-43
LOB 6-43
オブジェクト・ビュー 8-2
オブジェクト・ビュー , 更新 8-5
オブジェクト・ビュー , 作成 8-3
オブジェクト・リレーショナル・アプローチ、オブジェクト表でインプリメントする 7-8
オブジェクト・リレーショナル・データベース管理システム（ORDBMS）7-2
オプティマイザ
アプリケーションでヒントを使用する 2-5
オペレーティング・システム
ロールおよび～ 17-11

か

カーソル 3-8
DBMS_SQL パッケージ 14-4
クローズする 3-9, 14-7
取消し 3-10
～の最大数 3-9
プライベート SQL 領域および～ 3-8
～へのポインタ 10-24
カーソルの取消し 3-10
カーソル変数 10-24
宣言してオープンする 10-24
カートリッジ 10-64, 10-82
解析ツリー 13-16
外部 LOB(BFILE) 6-5
外部 LOB を削除する 6-36
外部キー、～で多対 1 の E-R を表す 7-5
外部結合 4-20
～でキー保存された表 4-21
外部コールアウト 6-48
拡張 ROWID 形式 5-12
拡張性
逐次再使用可能パッケージ 10-48
各国語サポート 2-5
NCLOB 6-7, 6-63
監査する
トリガーおよび～ 13-20
関数を変換する 5-17
TO_CHAR 関数 5-9, 5-19
TO_DATE 関数 5-9
TO_LABEL 関数 5-19
Trusted Oracle Server 5-19

き

キー
一意の
複合 9-6
外部キー 9-23
キー保存された表
外部結合で～ 4-21
結合ビュー内の～ 4-17
キー、外部 7-5
記憶域
オブジェクト表 7-18
記憶パラメータ
PCTFREE 4-8
PCTUSED 4-8
擬似列
ビューの変更 13-4
機能 2-4
キャッシュ
オブジェクト・キャッシュ 6-43
順序キャッシュ 4-26
順序番号 4-23
キャラクタ・セット
ANY_CS 6-63
キュー , 定義 11-11
キューイング 11-1, 11-4
DBMS_AQADM パッケージ 11-74
キューのサブスクライバ , 定義 11-17
キュー表 , 定義 11-11
キュー表およびキューの作成 11-26
行
ROWID で示される 5-13
形式 4-3
サイズ 4-3
整合性制約に違反する 9-18
ブロック間の連鎖 4-5
ヘッダー 4-3
業界標準への準拠 2-6
行トリガー
REFERENCING オプション 13-9
UPDATE 文および～ 13-6, 13-9
タイミング 13-3
定義する 13-6

業務処理管理 11-4
業務ルール 1-3
共有 SQL 領域
　アプリケーションで使用する 2-5
共有行排他ロック（SRX）
　LOCK TABLE コマンド 3-13
共有ロック（S）
　LOCK TABLE コマンド 3-12
行ロック
　手動でロックする 3-14

く

空白埋込みデータ
　パフォーマンスの検討 5-6
クライアント側の開発ツール 1-3
クラスタ
　エクステントを割り当てる 4-36
　キー 4-33
　索引の作成 4-35
　削除 4-37
　削除された表および～ 4-9
　作成 4-35
　整合性制約および～ 4-36
　データを選択する 4-34
　パフォーマンスの検討 4-34
　～を作成するための権限 4-36

け

結合ビュー 4-14
　DELETE 文 4-18
　UPDATE 文 4-18
　修正可能なとき 4-15
　修正する
　～の規則 4-17
　～でキー保存された表 4-17
　マージ可能な 4-15
権限
　オブジェクト権限を取り消した場合 17-16
　オブジェクトを改名する 4-44
　管理する 17-7, 17-12
　クラスタの作成 4-36
　索引の作成 4-33
　シノニムの作成 4-28
　順序を削除する 4-27
　順序を作成する 4-23
　順序を使用する 4-27
　順序を変更する 4-23
　ストアド・プロシージャの実行 10-36
　整合性制約を作成する 9-16
　選択した列に対して 17-15
　トリガー 13-15
　トリガーの作成 13-15
　トリガーを再コンパイルする 13-17, 15-5
　トリガーを削除する 13-17
　トリガーを使用可能にする 13-19
　トリガーを使用禁止にする 13-19
　取り消す 17-12, 17-14
　パッケージまたはプロシージャを再コンパイルする 15-5
　ビューを置き換える 4-13
　ビューを再コンパイルする 15-5
　ビューを削除する 4-14
　ビューを作成する 4-11
　ビューを使用する 4-14
　表を削除する 4-9
　表を作成する 4-7
　表を変更する 4-9
　付与する 17-12, 17-13
　ロールおよび～を使用可能にする 17-9
　ロックを手動で獲得する 3-14
権限とロールを取り消す
　REVOKE コマンド 17-14
　選択した列に対して 17-15
権限とロールを付与する 17-12

こ

更新済みロケータ 6-22, 6-25, 6-30, 6-31, 6-47
構造化されたペイロード 11-8
構造体
　C における～の配列 2-6
コレクション
　表項目 14-13
コレクション型の COUNT 属性 7-13, 8-7
コンパイル時エラー 10-32

さ

サーバーへの往復 , 回避 6-44, 6-49
再コンパイル
　実行時を避けた～ 15-2
再使用可能パッケージ 10-48

索引
SQL*Loader および～ 4-30
一時セグメントおよび～ 4-29
ガイドライン 4-30
権限 4-33
削除 4-33
削除された表および～ 4-9
作成 4-32
どんな場合に作成するか 4-29
～に対して PCTFREE を指定する 4-5
列の順序 4-31
削除
クラスタ 4-37
索引 4-33
シノニム 4-28
順序 4-27
整合性制約 9-23
トリガー 13-17
パッケージ 10-13
ハッシュ・クラスタ 4-39
ビュー 4-14
表 4-9
プロシージャ 10-10
ロール 17-12
作成
クラスタ 4-35
索引 4-32
シノニム 4-28
順序 4-27
整合性制約 9-2
トリガー 13-2, 13-11
パッケージ 10-12
ハッシュ・クラスタ 4-38
ビュー 4-10
表 4-3
複数のオブジェクト 4-41
参照整合性
1 対 1 関連 9-9
1 対多関連 9-9
外部キーを作成するために必要な権限 9-24
自己参照型制約 13-28
トリガーおよび～ 13-25, 13-26, 13-27, 13-28, 13-29
分散データベースおよび～ 9-12
サンプル・プログラム
daemon.pc 12-17
daemon.sql 12-15

し

システム固有の Oracle マニュアル 3-18, 10-65, 12-3, 12-21, 13-3, 14-2, 15-7, 16-3
PL/SQL ラッパー 10-27
UTLDTREE.SQL スクリプト 15-2
システム・グローバル領域
DBMS_OUTPUT データのバッファリング 12-23
順序番号キャッシュの保持 4-26
バッファ・パイプ情報 12-2
実行の流れ
動的 SQL における～ 14-4
シノニム
権限 4-28
削除 4-28
削除された表および～ 4-9
作成 4-28
使用する 4-28
ストアド・プロシージャおよびパッケージ 10-39
修正可能な結合ビュー
～の定義 4-15
手動ロック 3-10
LOCK TABLE コマンド 3-11
順序
CURRVAL 4-24, 4-25
NEXTVAL 4-24
アクセス 4-24
アプリケーションで使用する 2-6
削除 4-27
削除するための権限 4-27
作成 4-23, 4-27
順序番号をキャッシュする 4-26
使用するための権限 4-27
初期化パラメータ 4-23
直列化を低減する 4-24
パラレル・サーバー 4-23
番号のキャッシュ 4-23
変更する 4-23
変更するための権限 4-23
～を作成するための権限 4-23
順序メソッド 7-10, 7-14
純粋さレベル 10-43
使用可能にする
整合性制約 9-19
違反がある場合 9-18
作成時に 9-17, 9-18
例外をレポートする 9-21

トリガー 13-18
ロール 17-10
使用禁止にする
整合性制約 9-18
トリガー 13-18
条件述語
トリガー本体 13-8, 13-9
状態
パッケージ・オブジェクトのセッション 10-14
初期化パラメータ
DML_LOCKS 3-11
OPEN_CURSORS 3-9
REMOTE_DEPENDENCIES_MODE 10-22
ROW_LOCKING 3-11, 3-15
SERIALIZABLE 3-11
書式マスク
TO_DATE 関数 5-8
署名
PL/SQL ライブラリ・ユニット依存性 10-14
リモート依存性を管理するための 10-16

す

スキーマ 17-6
ストアド・ファンクション 10-4
作成 10-8
ストアド・プロシージャ 10-4
PUBLIC に付与されている権限を使用する 17-17
アプリケーションで使用する 2-4
格納する 10-8
権限 10-36, 15-5
再コンパイルする 15-2, 15-4, 15-5
作成 10-8
実行時のコンパイルを避ける 15-2
シノニム 10-39
提供される 10-56
動的 SQL 14-2
～に関する情報をリストする 10-73
～の名前 10-5
～の名前をオーバーロードする 10-5
～の例外 10-27
パラメータ
デフォルト値 10-8
引数の値 10-37
分散問合せの作成 10-31
呼び出す 10-34
リモート 10-37
リモート・オブジェクトと～ 10-38
例外 10-29
スレッド・セーフティ
OCI アプリケーションでの～ 2-6

せ

世紀 5-8
日付書式マスク 5-8
正規化 2-3
整合性制約
アプリケーションが使用する 9-2
アプリケーションで使用する 2-4
違反 9-18
違反がある場合に使用可能にする 9-18
クラスタおよび～ 4-36
削除 9-23
作成時に使用可能にする 9-17
作成に必要な権限 9-16
使用可能にする 9-18
使用禁止にする 9-17, 9-18, 9-19, 9-20
追加または削除に関する制限 9-16
定義する 9-15
どんな場合に使用禁止にするか 9-18
どんな場合に使用するか 9-2
～に対するトリガー 13-2, 13-24
～の定義をリストする 9-25
～の例外 9-21
パフォーマンスの検討 9-2
変更する 9-22
命名する 9-17
例 9-2
整合性制約に違反する 9-18
制約 7-16
ストアド・ファンクションの制限 10-40
複合 UNIQUE キー 9-6
制約表 13-13
西暦 2000 年 5-8
セーブポイント
～の最大数 3-6
～までロールバックする 3-6
セキュリティ
PL/SQL ファイル入出力中 12-29
UTL_FILE パッケージを使用する場合 12-28
アプリケーションでの施行 2-7
アプリケーションに対する方針 17-2, 17-5
動的 SQL 14-7

ロール、利点 17-3
セッション
SLEEP プロシージャ 3-24
～間で通信する 12-2
パッケージの状態と～ 10-14

そ

相関識別子 11-8
相関名 13-8, 13-9
NEW 13-8
OLD 13-8
REFERENCING オプションおよび～ 13-9
列にコロンが付く場合 13-8
属性 7-18

た

第 3 世代言語 10-2
タイムスタンプ
PL/SQL ライブラリ・ユニット依存性 10-14
対話形式のブロック実行 10-34

ち

遅延メッセージ交換 11-7
逐次再使用可能 PL/SQL パッケージ 10-48
調整
LONG を使用する 5-10
概要 2-7
直列可能トランザクション 3-26

つ

通信
セッション間 12-2

て

提供されるプロシージャ 10-56
ディクショナリ
データ・ディクショナリを参照
ディレクトリ
カタログ・ビュー 6-17
使用のガイドライン 6-18
所有権と権限 6-16
ディレクトリ・オブジェクト 6-14
データ型 5-2
ANSI/ISO 5-15
CHAR 5-2, 5-5
DATE 5-7, 5-8
DB2 5-15
DBMS_DESCRIBE 10-68
DESC_TAB 14-26
LONG 5-9
LONG RAW 5-10, 5-11
MLSLABEL 5-15
NCHAR 5-2, 5-5
NCLOB 6-63
NUMBER 5-7
NVARCHAR2 5-2, 5-5
PL/SQL
～の数値コード 10-72
RAW 5-11
ROWID 5-12, 10-74
SQL/DS 5-15
VARCHAR 5-5
VARCHAR2 5-2, 5-5
VARCHAR2S 14-11
データ型のサマリー 5-2
データ変換 5-17
文字型の列の長さ 5-6
文字データ型の選択 5-5
データベース
管理者
アプリケーション管理者と～ 17-2
正規化する 2-3
セキュリティ
アプリケーションおよび～ 17-2
スキーマおよび～ 17-6
設計する 2-2
トリガー
アプリケーションで使用する 2-5
分散システムにおけるグローバル名 4-42
データベース・リンク
Trusted Database List 10-63
データ変換 5-17
ANSI データ型 5-16
SQL/DS データ型と DB2 データ型 5-16
Trusted Oracle ラベル 5-19
式の評価 5-18
代入 5-17
データを変換する 5-17
ANSI データ型 5-16

SQL/DS データ型と DB2 データ型 5-16
Trusted Oracle Server 5-19
式の評価 5-18
代入 5-17
データ・オブジェクト番号
拡張 ROWID 5-12, 5-13
データ・ディクショナリ
コンパイル時エラー 10-33
削除された表および～ 4-9
スキーマ・オブジェクト・ビュー 4-44
～における整合性制約 9-25
プロシージャとパッケージに関する情報 10-73
プロシージャ・ソース・コード 10-33
データ・ファイル
ROWID で示される 5-13
データ・ブロック
ROWID で示される 5-13
～のサイズに影響を与える要因 4-6
デーモン , Pro*C 12-17
デバッグする
ストアド・プロシージャ 10-33
トリガー 13-17
デフォルト
PCTFREE オプション 4-4
PCTUSED オプション 4-6
最大のセーブポイント 3-6
ストアド・ファンクション内のパラメータ 10-42
列値 9-4, 10-40
ロール 17-10
デリファレンス 7-14
デリファレンス、暗黙的 7-14, 8-6

と

問合せ
分散問合せのエラー 10-31
同期通信 11-7
同時実行性 3-26
動的 SQL
DBMS_SQL パッケージ 14-2, 14-8
DBMS_SQL ファンクション、～を使用する 14-3
LAST_ERROR_POSITION ファンクション 14-28
LAST_ROW_COUNT ファンクション 14-28
LAST_ROW_ID ファンクション 14-28
LAST_SQL_FUNCTION_CODE ファンクション 14-28
エラー、位置を見つける 14-28
セキュリティ 14-7
～における実行の流れ 14-4
無名ブロックおよび～ 14-3
例 14-28
トランザクション
LOB ロケータは～にまたがることができない 6-31
SET TRANSACTION コマンド 3-8
外部 LOB は～に関係ない 6-6
～からの移行 6-48
手動ロック 3-10
直列可能 3-26
内部 LOB は～に完全に関与する 6-5, 6-6
読取り専用 3-8
トリガー
AFTER 13-3, 13-8, 13-21, 13-23
BEFORE 13-3, 13-8, 13-31, 13-33
FOR EACH ROW 句 13-6
INSTEAD OF トリガー 13-4
LONG、LONG RAW データ型の使用 13-11
REFERENCING オプション 13-9
UPDATE での列リスト 13-6, 13-9
WHEN 句 13-7
アプリケーションで使用する 2-5
移行の問題 13-17
イベント 13-6
エラー条件と例外 13-10
格納 13-16
行 13-6
行の評価順序 13-12
権限 13-15
検査制約 13-30, 13-31
コンパイル 13-16
再コンパイルする 13-17, 15-5
再コンパイルの権限 15-5
削除された表および～ 4-9
削除するための権限 13-17
作成 13-2, 13-11, 13-15
作成前の前提条件 13-3
参照整合性および～ 13-25, 13-27, 13-28, 13-29
修正する 13-17
使用可能にする 13-18
使用禁止にする 13-18
使用禁止にするにするための権限 13-19
条件述語 13-8, 13-9
制限 13-7, 13-11
設計する 13-2
走査順序 13-12

データ・アクセスの制限 13-31
〜で監査する 13-20, 13-22
デバッグする 13-17
〜でレポートされているユーザー名 13-15
導出列値を生成する 13-33
トリガーの評価順序 13-12
〜に関する情報をリストする 13-19
〜に対する整合性制約 13-2, 13-24
〜について 10-4
パッケージ変数および〜 13-12
複数の同じタイプ 13-12
プロシージャおよび〜 13-11
分散問合せの作成 10-31
変更表および〜 13-13
本体 13-8, 13-9, 13-10, 13-11
無効な SQL 文 13-11
命名する 13-3
リモート依存性および〜 13-11
リモート例外 13-10
例 13-20, 13-22, 13-23, 13-26, 13-30, 13-31, 13-33
列値にアクセスする 13-8

な

内部 LOB 6-5
内部 LOB のコピー・セマンティクス 6-35
内部 LOB を空に設定する 6-14
内部 LOB を削除する 6-36
名前変換 4-42

ね

ネストした表 7-18
ネストした表対 VARRAY 7-9, 7-11
ネストした表、問合せ 7-11
ネストした表、〜の一意性 7-19

は

排他ロック
LOCK TABLE コマンド 3-14
バイナリ・データ
RAW と LONG RAW 5-11
パイプ 12-2
管理する 12-11
セッション間の通信 12-2
〜の定義域 12-3
パブリックまたはプライベート 12-2
例 12-12
配列 7-17
BIND_ARRAY プロシージャ 14-6, 14-11
DBMS_SQL を使用するバルク DML 14-13
DEFINE_ARRAY プロシージャ 14-17
VARRAY を参照
パススルー SQL 10-63
パッケージ
DBMS_DESCRIBE 10-65
DBMS_OUTPUT
使用例 10-2
DBMS_PIPE 12-2
DBMS_ROWID 10-74
DMBS_OUTPUT 12-20
Oracle が提供する 10-56, 10-61
PL/SQL における 10-10
UTL_FILE 12-27
UTL_HTTP 10-82
アプリケーションで使用する 2-4
オブジェクトの依存性の最小化 15-3
権限 15-5
再コンパイルする 15-2, 15-4, 15-5
削除 10-13
作成 10-12
作成に必要な権限 10-12
実行時のコンパイルを避ける 15-2, 15-3
実行のための権限 10-36
シノニム 10-39
セッションの状態と〜 10-14
説明箇所 1-5, 10-56
逐次再使用可能パッケージ 10-48
データ・ディクショナリ・ビュー 10-73
〜に関する情報をリストする 10-73
プロシージャを作成するために必要な権限 10-9
〜を命名する 10-13
パッケージ仕様部 10-10
パッケージの初期化部分
〜の問題を回避する 10-47
パッケージ本体 10-10
ハッシュ・クラスタ
キーを選択する 4-38
削除 4-39
作成 4-38
どんな場合に使用するか 4-38
ルート・ブロック 4-38
発注書の例 7-2

バッファ
LOB 6-43
パフォーマンス
ROW_LOCKING パラメータ 3-15
SERIALIZABLE オプション 3-15
クラスタ 4-34
索引列の順序 4-31
パラメータ
デフォルト値 10-8
ストアド・ファンクション付き 10-42
ファイル
INIT.ORA 12-28, 12-29
モード 10-6
パラレル・サーバー
順序番号および～ 4-23
分散ロック 3-10
反復読取り 3-8, 3-10

ひ

比較演算子
空白埋込みデータ 5-6
日付の比較 5-8
比較メソッド 7-12
日付算術 5-18
ビュー
FOR UPDATE 句および～ 4-10
ORDER BY 句および～ 4-10
WITH CHECK OPTION 4-10
エラーを伴って作成 4-11
置き換える 4-12
擬似列 13-4
結合ビュー 4-14
権限 4-11, 15-5
固有に変更可能 13-4
再コンパイルする 15-4, 15-5
削除 4-14
削除された表および～ 4-9
作成 4-10
式を含む 13-4
使用する 4-13
制限 4-13
データ・ディクショナリを参照
どんな場合に使用するか 4-10
変更可能 13-4
無効な 4-14
表
PL/SQL における 10-8
位置 4-4
ガイドライン 4-3
キー保存された 4-17
切り捨てる 4-9
クラスタ化した～のスキーマ 4-35
削除 4-9
削除するための権限 4-9
作成 4-3
作成のための権限 4-7
制約 13-13
設計する 4-3
～に対して PCTFREE を指定する 4-5
～に対して PCTUSED を指定する 4-6
配列としての表項目 14-13
表領域を指定する 4-4
変更 13-13
変更する 4-8
変更するための権限 4-9
列長さを大きくする 4-8
標準
ANSI 3-15
準拠 2-6
表を更新する
親キーを使用して 9-10, 9-11
表、オブジェクト、「オブジェクト表」を参照
表、ネストした 7-18

ふ

ファイル I/O
PL/SQL における 12-27
ファイルの所有権
UTL_FILE パッケージ 12-29
ファンクション
PL/SQL を参照
ブール式 5-19
不完全なオブジェクト型 7-9
複合キー
～で NULL を制限する 9-14
副次作用 10-6, 10-43
プライベート SQL 領域
カーソルおよび～ 3-8
プラグマ 7-12, 10-44
EXCEPTION_INIT プラグマ 10-76
RESTRICT_REFERENCES プラグマ 10-44, 10-46
SERIALLY_REUSABLE プラグマ 10-48, 10-50

プリコンパイラ 10-35
　アプリケーション 10-4
プリビュー後のメッセージのデキュー 11-39
プロシージャ
　SLEEP 3-24
　アプリケーションで使用する 2-4
　コンパイル・エラーのリスト 10-73
　サイズ情報 10-74
　実行時のコンパイルを避ける 15-2
　ソース・コードのリスト 10-74
　提供される 10-56
　データ・ディクショナリ・ビュー 10-73
　トリガーのコール先 13-11
　～に関する情報をリストする 10-73
　ローカルな 10-13
プロシージャとパッケージに関する情報をリストする 10-73
プロファイル
　アプリケーションの設計および～ 2-6
分散データベース
　参照整合性および～ 9-12
　トリガーおよび～ 13-11
　リモート・ストアド・プロシージャ 10-37, 10-38
分散問合せ
　エラーを処理する 10-31
文トリガー
　SQL 文の指定 13-6
　UPDATE 文および～ 13-6, 13-9
　行の評価順序 13-12
　タイミング 13-3
　トリガーの評価順序 13-12
　文のための条件付きコード 13-9
　有効な SQL 文 13-11

へ

変更する
　記憶パラメータ 4-8
　表 4-8
変更表 13-13

ほ

方法
　BFILE に対する参照ベースの 6-15
　内部 LOB に対するコピー・ベースの 6-35
保存とメッセージ履歴 11-8
本体
　トリガー 13-8, 13-9, 13-10, 13-11

ま

マップ・メソッド 7-10
マルチスレッド・サーバー（MTS）
　BFILE 6-19

み

未処理例外 10-30

む

無効なビュー 4-14
無名 PL/SQL ブロック
　動的 SQL および～ 14-2, 14-3
　トリガーに比較された 10-4
　～について 10-2

め

明示的なロック
　手動ロック 3-10
メソッド、順序 7-10, 7-14
メソッド、比較 7-12
メソッド、マップ 7-10
メッセージ
　セッション間 12-2
　プロデューサおよびコンシューマ 11-11
メッセージ , 定義 11-11
メッセージのエンキューおよびデキュー
　Pro*C/C++ を使った、相関およびメッセージ ID による 11-43
　Pro*C/C++ を使った RAW 型の 11-34, 11-36
　RAW 型の 11-31
　オブジェクト型の 11-28
　遅延および時間切れによる 11-42
　複数コンシューマキューでの 11-49, 11-52
　優先順位による 11-31
メッセージのグループ化 11-9
メッセージの受信者 , 定義 11-17
メッセージの順序
　検索 11-5
メッセージ・システム
　メトリック 11-6

要件 11-6
メッセージ・プロパティ , 仕様部 11-64
メモリー
拡張性 10-48

も

モード
パラメータの 10-6

ゆ

有効範囲を設定した REF 7-18
ユーザー
アプリケーション・ロールの制限 17-5
削除されたロールおよび～ 17-12
パブリック・グループ 17-17
ロールを使用可能にする 17-3
ユーザー定義エラー 10-27, 10-29
ユーザー名
スキーマおよび～ 17-6
トリガー内でレポートされているとおりの 13-15
ユーザー・ロック
要求する 3-17
優先メッセージのキュー表およびキューの作成 11-27

よ

読取り一貫性
LOB 6-22
読取り一貫性のあるロケータ 6-22, 6-23, 6-29, 6-31, 6-47, 6-49, 6-50, 6-52
読取り専用トランザクション 3-8
予約リストと宛先リスト 11-9

ら

ライブラリ・ユニット
リモート依存性 10-14
ラベル
MLSLABEL データ型 5-15
データ変換 5-19

り

リーフ・レベル・スカラー属性 7-18
リモート依存性 10-14, 15-4
署名 10-16
タイムスタンプまたは署名を指定する 10-22
リモート例外処理 10-31, 13-10

れ

例
LOB バッファリング 6-50
PL/SQL 変数を使用した LOB の更新 6-30
SQL DML と DBMS_LOB の混合の影響 6-26
更新済み LOB ロケータ 6-28
読取り一貫性のあるロケータ 6-23
例外
ROWID_INVALID 10-76
UTL_FILE パッケージ 12-30
アプリケーションに与える影響 10-30
トリガー実行中 13-10
未処理 10-30
無名ブロック 10-3
リモート・プロシージャ 10-31
例外ハンドラ
PL/SQL における 10-2
例外を発生させる
トリガー 13-10
例、発注書 7-2
列
CHECK 制約の制限数 9-14
UPDATE トリガーでリストする 13-6, 13-9
～権限を取り消す 17-15
選択した～に権限を付与する 17-13
デフォルト値 9-4
トリガーで導出列値を生成する 13-33
トリガー内でアクセスする 13-8
長さを大きくする 4-8
複数の FOREIGN KEY 制約 9-10

ろ

ローカル・プロシージャ
パッケージ本体に 10-13
ロール
ADMIN OPTION オプションおよび～ 17-13
GRANT および REVOKE コマンド 17-11
SET ROLE コマンド 17-11
WITH GRANT OPTION および～ 17-14
アプリケーション 17-2, 17-3, 17-5, 17-7
アプリケーション・セキュリティ方針 17-2, 17-5

オペレーティング・システムによる～の付与 17-11
管理する 17-7
削除 17-12
作成 17-8
使用可能にする 17-3, 17-10
デフォルト 17-10
どんな場合に使用可能にするか 17-9
付与する 17-12
ユーザー 17-3, 17-5, 17-7
ユーザー権限および～を使用可能にする 17-9
利点 17-3
～を作成するための権限 17-9
ロールおよび権限の取消し（AQ）11-61
ロールバック・トランザクション
セーブポイントまで 3-6
ロケータ 6-20
更新済み 6-22, 6-25, 6-30, 6-31, 6-47
選択する 6-22
～によって LOB にアクセスする 6-22
複数の 6-23
複数のトランザクションにまたがることはできない 6-31
読取り一貫性のある 6-22, 6-23, 6-29, 6-31, 6-47, 6-49, 6-50, 6-52
読取り一貫性のあるロケータ 6-22
～を含むように列 / 属性を設定する 6-21
ロック
LOCK TABLE コマンド 3-11, 3-12
UTLLOCKT.SQL スクリプト 3-25
アプリケーションの設計および～ 2-5
行ロックモード 3-15
索引付き外部キーおよび～ 9-11
手動獲得するための権限 3-14
手動（明示的な）3-10
直列可能モード 3-15
分散 3-10
無索引外部キーおよび～ 9-10
ユーザー・ロック 3-17

わ

ワークフロー 11-4
ワールド・ワイド・ウェブ（WWW）・コールアウト 10-64, 10-82
割当て
エクステント 4-36