

Oracle Rdb7

SQL プログラミングのためのガイド

リリース 7.0

2001 年 12 月

部品番号 : J03807-01

Oracle Rdb7 SQL プログラミングのためのガイド, リリース 7.0

部品番号 : J03807-01

原本名 : Oracle Rdb7 Guide to SQL Programming Release7.0

原本部品番号 : A42867-1

Copyright © 1996, 2001, 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。著作権、特許権およびその他の知的財産権に関する法律により保護されています。

当ソフトウェア（プログラム）のリバース・エンジニアリングは禁止されています。

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

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

危険な用途への使用について

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

Restricted Rights Legend

Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication and disclosure of the Programs shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-14, Rights in Data -- General, including Alternate III (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

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

目次

はじめに	xiii
このマニュアルについて	xiii
対象読者	xiii
マニュアルの構成	xiv
関連マニュアル	xv
SQL 標準	xv
表記規則	xv
技術的な変更と新機能	xvii

第 I 部 SQL プログラミングの概要

1 SQL プログラミングの概要

1.1 2つの SQL プログラミング・インタフェース	1-2
1.1.1 SQL モジュール・プロセッサ	1-2
1.1.2 SQL プリコンパイラ	1-3
1.2 プログラミング・インタフェースの選択	1-3
1.3 オンライン・プログラムのサンプルの検索	1-5

第 II 部 SQL を使用したアプリケーション・プログラムの開発

2 SQL プログラム開発工程

2.1 アプリケーション・プログラム開発工程の概要	2-2
2.2 エンド・ユーザーの要求の理解	2-2
2.3 メタデータとデータの調査	2-3
2.4 プロトタイプ作成	2-3

2.5 プロトタイプからアプリケーション・プログラムへの変換	2-5
2.6 アプリケーション・プログラムの開発	2-6

3 SQL モジュール言語の概要

3.1 SQL モジュール言語アプリケーション・プログラムの開発方法：基本手順	3-2
3.2 SQL モジュール・ソース・ファイルの作成方法	3-4
3.2.1 SQL モジュール内に空白行とコメントを含める方法	3-6
3.2.2 モジュールの命名	3-6
3.2.3 表現形式 (dialect) の指定方法	3-7
3.2.4 セッションに対するキャラクタ・セットの指定方法	3-7
3.2.5 モジュール・プロシージャをコールするホスト言語を識別	3-8
3.2.6 カタログの指定	3-9
3.2.7 スキーマの指定	3-9
3.2.8 認可識別子 (Authorization Identifier) の指定	3-10
3.2.9 エイリアスの指定	3-10
3.2.10 パラメータにコロンをつける設定	3-11
3.2.11 モジュール内での DECLARE 文の指定	3-11
3.3 ホスト言語プログラムからの SQL モジュール・プロシージャのコール	3-12
3.4 SQL モジュール・プロセッサによる移植可能アプリケーションの記述	3-14
3.5 SQL モジュール・プロセッサの詳細情報の参照先	3-14

4 SQL モジュール・プロシージャの記述方法

4.1 SQL モジュール・プロシージャについて	4-2
4.2 SQL モジュール・プロシージャの共通要素の指定方法	4-3
4.2.1 プロシージャの命名	4-3
4.2.2 プロシージャ・パラメータの宣言	4-4
4.2.3 異なる種類のプロシージャに対して必要なパラメータ	4-5
4.2.4 プロシージャ・パラメータと実パラメータの関連	4-9
4.2.5 パラメータ・データ型の指定	4-10
4.2.6 パラメータ・データ型の LANGUAGE 句の効果	4-11
4.2.7 パラメータ引渡しメカニズムにおける LANGUAGE 句の効果	4-12
4.2.8 プロシージャ・パラメータ用のデフォルトの引渡しメカニズムの上書き	4-13
4.2.9 コール側モジュールによって使用されるパラメータの実行時のチェック要求	4-14
4.3 プロシージャでの単一 SQL 文の使用法	4-14
4.4 複数文プロシージャにおける複合文の使用法	4-15

4.5 SQL モジュール言語の制限について	4-16
5 SQL モジュールとホスト言語ファイルの処理	
5.1 SQL モジュール・プロセッサの起動	5-2
5.2 SQL モジュールとホスト言語モジュールの処理	5-3
5.3 高速コンパイルでのパラメータ・チェックの回避	5-4
5.4 リモート・データベース用の SQL モジュール・プロセッサのパフォーマンスの改善	5-4
5.5 SQL モジュール言語でのコンテキスト・ファイルの使用	5-5
5.6 SQL モジュールのスコープの決定	5-6
6 プリコンパイル SQL の使用方法	
6.1 プリコンパイラ・プロセスについて	6-2
6.2 ホスト・プログラムへの SQL 文の埋込み	6-3
6.3 プリコンパイラの起動	6-6
6.4 プリコンパイル時およびコンパイル時のエラー検出	6-10
6.5 リモート・データベースに対するプリコンパイラのパフォーマンスの改善	6-10
6.6 コンパイル時と実行時のオプションの指定	6-11
6.6.1 DECLARE MODULE 文の使用法	6-11
6.6.2 SQL コンテキスト・ファイル内に宣言を記述する方法	6-12
6.7 SQL プリコンパイラの使用に関する言語固有のガイドライン	6-13
6.7.1 Ada ソース・ファイルでの SQL 文の埋込み	6-14
6.7.2 Ada プログラムのプリコンパイル	6-14
6.7.3 C ソース・ファイルでの SQL 文の埋込み	6-16
6.7.4 COBOL ソース・ファイルでの SQL 文の埋込み	6-18
6.7.5 FORTRAN ソース・ファイルでの SQL 文の埋込み	6-20
6.7.6 Pascal ソース・ファイルでの SQL 文の埋込み	6-22
6.7.7 PL/I ソース・ファイルでの SQL 文の埋込み	6-23
7 プログラム実行のためのイメージの作成	
7.1 実行可能イメージと共有可能イメージ	7-2
7.2 OpenVMS Linker の使用方法	7-2
7.2.1 OpenVMS VAX システム上の Compaq Tru64 C コンパイラでコンパイルした プログラムのリンク	7-3
7.2.2 共有可能イメージにリンクする実行可能イメージの作成	7-3
7.2.3 Ada オブジェクトのリンク	7-4

7.3 共有可能イメージの作成	7-5
7.3.1 データベース接続を共有しない実行可能イメージと共有可能イメージの作成	7-6
7.3.2 データベース接続を共有する実行可能イメージと共有可能イメージの作成	7-7
7.4 共有可能イメージのインストール	7-10
7.5 CompaqTru64 UNIX 上でのモジュールのリンク	7-10
7.5.1 複数のモジュールでのアプリケーションの構築	7-11
7.6 オブジェクト・ライブラリおよびアーカイブへのプリコンパイル SQL モジュールの挿入	7-12
7.7 プログラムの実行	7-12
7.8 SQL 文とプログラム・コードのデバッグ	7-13

8 パラメータの宣言と使用方法

8.1 パラメータの宣言と使用方法の概要	8-2
8.2 用語について	8-3
8.3 パラメータ機能と宣言オプション	8-3
8.4 パラメータのデータ型の宣言	8-5
8.5 プログラムの外部ソースからのパラメータ宣言のコピー	8-8
8.6 SQL INCLUDE 文の使用方法	8-8
8.6.1 SQL INCLUDE FROM DICTIONARY 文の使用方法	8-9
8.6.2 INCLUDE SQLCA 文の使用方法	8-10
8.6.3 INCLUDE SQLD 文または SQLDA2 文の使用方法	8-11
8.6.4 INCLUDE ファイル文の使用方法	8-11
8.7 SQL モジュール言語 FROM path-name 句の使用方法	8-11
8.8 ホスト言語 COPY 文と INCLUDE 文の使用方法	8-12
8.9 メイン・パラメータの宣言と使用方法	8-12
8.9.1 メイン・パラメータの宣言	8-13
8.9.2 メイン・パラメータの使用方法	8-15
8.10 インジケータ・パラメータの宣言と使用方法	8-18
8.10.1 インジケータ・パラメータの宣言	8-18
8.10.2 インジケータ・パラメータの使用方法	8-20
8.10.3 検索でのインジケータ・パラメータ	8-21
8.10.4 ソートでのインジケータ・パラメータ	8-22
8.11 パラメータを宣言して使用する場合の誤りの防止	8-24
8.11.1 埋込み SQL を使用する場合の誤りの防止	8-26
8.11.2 SQL モジュールを使用する場合の誤りの防止	8-26
8.12 特定の言語でのパラメータの宣言方法と使用方法	8-27
8.12.1 Ada ソース・ファイルにおけるパラメータの宣言方法と使用方法	8-27

8.12.2 C ソース・ファイルにおけるパラメータの宣言方法と使用方法	8-27
8.12.3 COBOL ソース・ファイルにおけるパラメータの宣言方法と使用方法	8-30
8.12.4 FORTRAN ソース・ファイルにおけるパラメータの宣言方法と使用方法	8-30
8.12.5 Pascal ソース・ファイルにおけるパラメータの宣言方法と使用方法	8-31
8.12.6 PL/I ソース・ファイルにおけるパラメータの宣言方法と使用方法	8-31
8.12.7 SQL モジュールとコール側プログラムにおけるパラメータの宣言方法と使用方法	8-31

9 日付時刻データ型の使用方法

9.1 日付時刻データ型によるデータの格納	9-2
9.2 プログラムでの日付時刻データ型の使用方法	9-3
9.2.1 プログラム開発のための日付時刻データ型の変換	9-4
9.2.2 SQL プリコンパイラでの日付時刻データ型の使用方法	9-5
9.2.3 SQL モジュール言語での日付時刻データ型の使用方法	9-8
9.3 日付時刻データ型を使用する場合の移植性の向上	9-12
9.4 アプリケーションとデータベースの変換	9-13
9.5 アプリケーションにおける DATE VMS データ型の処理	9-14
9.5.1 OpenVMS に固有のアプリケーションで DATE VMS を使用する方法	9-15
9.5.2 DATE VMS データ型を使用するアプリケーションの移植	9-15
9.6 動的 SQL での日付時刻データ型の使用方法	9-17
9.6.1 パラメータ・マーカーを伴う CAST の使用方法	9-17
9.6.2 日付をテキスト文字列として動的 SQL 文に渡す方法	9-19

第 III 部 実行時処理

10 実行時エラーの処理

10.1 SQL エラー処理の概要	10-2
10.2 SQL 文の監視によるエラー検出	10-4
10.2.1 SQLSTATE の使用方法	10-5
10.2.2 SQLCODE の使用方法	10-6
10.2.3 WHENEVER 文の使用法	10-10
10.2.4 sql_get_message_vector ルーチンと RDB\$LU_STATUS の使用方法	10-13
10.2.5 SQL エラー処理ルーチンの使用方法	10-18
10.3 エラー・メッセージの表示	10-27
10.3.1 sql_signal のコール	10-27
10.3.2 sql_get_error_text のコール	10-28

10.3.3 ユーザーが提供するエラー・メッセージの表示	10-30
10.3.4 インクルード・ファイルを使用した SQL ルーチン	10-30
10.4 重複値エラーと制約違反の処理	10-31
10.4.1 制約違反と重複値エラーのステータス値	10-31
10.4.2 制約評価の制御	10-32
10.5 ロックの競合とデッドロックの処理	10-32
10.5.1 ロックの競合エラーの処理	10-32
10.5.2 デッドロック・エラーの処理	10-34
10.6 データベース接続とトランザクション開始の失敗によるエラーの処理	10-35
10.7 エラーと制約を処理するプログラムの移植性の改善	10-36

11 動的 SQL

11.1 動的 SQL の紹介	11-2
11.1.1 動的に実行できる SQL 文のカテゴリ	11-2
11.1.2 動的 SQL 文による他の SQL 文の処理	11-4
11.1.3 動的 SQL における SQL 文の処理	11-5
11.2 パラメータ・マーカーのない SELECT 文以外の SQL 文の実行	11-5
11.3 パラメータ・マーカーと選択リスト項目の処理	11-8
11.3.1 SQLDA 構造と SQLDA2 構造の使用方法	11-9
11.3.2 SQLDA 構造と SQLDA2 構造の宣言	11-10
11.4 パラメータ・マーカーを伴う SELECT 文以外の SQL 文の実行	11-13
11.5 SELECT 文か否かのテスト	11-17
11.6 SELECT 文の処理	11-19
11.6.1 パラメータ・マーカーのない SELECT 文の実行 : 動的カーソルと 拡張動的カーソルの宣言	11-19
11.6.2 パラメータ・マーカーを伴う SELECT 文の実行	11-22
11.6.3 SQLDA2 構造と SQLERRD 構造によるパラメータ・マーカーと SELECT 文のテスト	11-29
11.7 動的に生成された SQL 文の処理	11-32
11.7.1 文識別子とカーソル名の格納	11-32
11.7.2 複数の SELECT 文以外の SQL 文の実行	11-33
11.7.3 複数の SELECT 文の実行	11-35
11.8 この章で使用するサンプル・プログラムの格納場所	11-39

第 IV 部 プログラムの構造

12 SQL における複合文の使用

12.1 複合文の概要	12-2
12.2 複合文によるパフォーマンスの向上	12-2
12.3 複合文の記述	12-3
12.3.1 変数の宣言と割当て	12-4
12.3.2 IF 文の使用法	12-6
12.3.3 CASE 文の使用法	12-8
12.3.4 LOOP 文の使用法	12-9
12.3.5 FOR 文の使用法	12-10
12.3.6 複合文でのラベルの使用法	12-12
12.3.7 LEAVE 文の使用法	12-13
12.3.8 ストアド・プロシージャや外部プロシージャの起動	12-14
12.4 複合文のアトム性の制御	12-15
12.5 複合文内のトランザクションの制御	12-17
12.6 複合文の動的な処理	12-18
12.7 複合文のデバッグ	12-18
12.8 複合文に関する情報の取得	12-21
12.9 例外条件と完了条件の処理	12-22
12.9.1 例外条件の取得	12-22
12.9.2 完了条件の取得	12-23

13 ストアド・ルーチンの使用方法

13.1 ストアド・ルーチンについて	13-2
13.2 データベースにルーチンを格納する利点	13-2
13.3 ストアド・モジュールの作成	13-3
13.3.1 ストアド・プロシージャの作成	13-5
13.3.2 ストアド・ファンクションの作成	13-7
13.3.3 ストアド・ファンクションによる新しい順序番号の生成	13-9
13.4 ストアド・プロシージャの起動	13-10
13.5 ストアド・ファンクションの起動	13-12
13.6 ストアド・ルーチンの削除	13-12
13.7 ストアド・ルーチンの依存性の追跡	13-13
13.7.1 プロシージャ依存関係タイプ	13-15

13.7.2 言語セマンティクス依存関係タイプ	13-16
13.7.3 トランザクション依存関係タイプ	13-17
13.8 ストアド・ルーチンの無効化	13-18
13.9 ストアド・ルーチンの再有効化	13-20
13.9.1 無効化されたストアド・ルーチンの再有効化	13-21
13.9.2 言語セマンティクスの依存関係で無効化されたストアド・ルーチンの再作成	13-22

14 外部ルーチンの使用方法

14.1 外部ルーチンの概要	14-2
14.2 外部ルーチンの開発	14-3
14.3 外部ルーチン定義の作成	14-4
14.3.1 外部関数の定義の作成	14-4
14.3.2 外部プロシージャ定義の作成	14-6
14.4 外部ルーチン定義の変更と削除	14-8
14.5 外部ルーチンの作成	14-9
14.5.1 既存のルーチンに基づく外部ルーチンの作成	14-9
14.5.2 ユーザー定義の外部ルーチンの記述	14-12
14.5.3 データベースにアクセスする外部ルーチンの記述	14-16
14.5.4 外部ルーチンをコールするジャケット・ルーチンの記述	14-23
14.6 外部ルーチンの共有可能イメージの作成	14-25
14.6.1 OpenVMS VAX における外部ルーチンの共有可能イメージの作成	14-26
14.6.2 OpenVMS Alpha における外部ルーチンの共有可能イメージの作成	14-27
14.7 Compaq Tru64 UNIX における共有オブジェクトの作成	14-29
14.8 外部ルーチンの起動	14-30
14.8.1 外部関数の起動	14-30
14.8.2 トリガー内での外部関数の起動	14-31
14.8.3 外部プロシージャの起動	14-32
14.9 ルーチンの実行特性の指定	14-32
14.10 ルーチンのアクティブ化と非アクティブ化	14-34
14.11 パラメータの宣言および引渡しと戻り値	14-36
14.12 外部ルーチンの記述に関する言語固有のガイドライン	14-38
14.12.1 Ada での外部ルーチンの使用方法	14-38
14.12.2 C での外部ルーチンの使用方法	14-39
14.12.3 COBOL での外部ルーチンの使用方法	14-40
14.12.4 FORTRAN での外部ルーチンの使用方法	14-40
14.12.5 Pascal での外部ルーチンの使用方法	14-40

14.13 通知ルーチンの使用方法	14-41
14.14 外部ルーチンでの例外処理	14-42
14.15 外部ルーチンの制限について	14-43
14.16 外部ルーチンのトラブルシューティング	14-44
14.17 外部ルーチンの移植性と効率の向上	14-45

第 V 部 プログラムのコンテキスト

15 データベースへの接続

15.1 データベースの指定と接続	15-1
15.1.1 データベース接続に対するファイルまたはリポジトリ・アクセスの指定	15-1
15.1.2 データベース名の指定	15-2
15.1.3 様々なデータベースのコンパイル時と実行時の指定	15-3
15.2 リモート・ノードでのデータベースの指定	15-4
15.2.1 リモート・ユーザー認証での USER 句および USING 句の使用法	15-6
15.2.2 コマンド行修飾子によるリモート・ユーザー認証	15-7
15.2.3 構成ファイルによるリモート・ユーザー認証	15-8
15.2.4 リモート・サーバー・アカウントとしてのプロキシ・アカウントの使用	15-8
15.2.5 リモート・サーバー・アカウントとしての RDB\$REMOTE アカウントの使用	15-9
15.2.6 代替 UCX またはインターネット・サービスの使用法	15-10
15.3 非同期システム・トラップの回避	15-11
15.4 分散トランザクションにおけるデータベース接続	15-11
15.4.1 分散トランザクションによる検出不可能なデッドロックの回避	15-11
15.4.2 分散トランザクションにおける権限エラーの回避	15-11
15.5 エイリアスによる複数の接続	15-12
15.6 データベースの接続解除	15-15

16 トランザクション・コンテキストの管理

16.1 トランザクションについて	16-2
16.1.1 トランザクションの特性	16-2
16.1.2 トランザクション特性を変更する基準	16-4
16.2 SQL プログラムにおけるトランザクション特性の指定	16-5
16.2.1 読取り専用トランザクションの使用法	16-6
16.2.2 読取り / 書込みトランザクションの使用法	16-8
16.2.3 バッチ更新トランザクションの使用法	16-8

16.2.4 RESERVING 句の使用方法	16-10
16.2.5 ロックの取得を待つかどうかの選択	16-12
16.2.6 分離レベルの選択	16-14
16.2.6.1 直列可能トランザクションの使用方法	16-15
16.2.6.2 反復可能読取りトランザクションの使用方法	16-15
16.2.6.3 コミット読込みトランザクションの使用方法	16-17
16.2.7 様々な分離レベルを使用する利点	16-19
16.2.8 エイリアスを使用して1つのトランザクション内で複数のデータベースに アクセスする方法	16-21
16.3 トランザクション・スコープについての理解	16-21
16.4 分散トランザクションの使用方法	16-24
16.5 データベース・リソースのロック	16-25
16.5.1 ロックの方法	16-25
16.5.2 明示的ロック	16-26
16.5.3 ロックの競合	16-28
16.5.4 読取り専用トランザクションとスナップショット・ファイル	16-30
16.5.5 読取り専用トランザクションでのロックの競合エラー	16-30
16.5.6 同時アクセスの改善	16-31
16.6 端末 I/O 操作のないトランザクションの設計	16-33
16.7 制約を評価するタイミングの指定	16-39
16.7.1 制約評価のタイミングの指定	16-40
16.7.2 制約を評価するタイミングに関する推奨事項	16-41
16.8 トランザクションのコミットとロールバック	16-42

17 プログラムにおける複数接続の管理

17.1 接続について	17-2
17.1.1 セッションの定義	17-2
17.1.2 データベース環境の定義	17-3
17.1.3 接続の定義	17-4
17.2 接続の作成、切替え、終了	17-5
17.2.1 接続の作成	17-5
17.2.2 デフォルトのデータベース環境の複製	17-6
17.2.3 異なるデータベースに対する同じエイリアスの指定	17-7
17.2.4 実行時の接続の指定	17-8
17.2.5 接続の切替え	17-9
17.2.6 接続の終了	17-10

17.3 接続を伴うトランザクションの使用方法	17-10
17.4 プログラムにおける接続の有効化と無効化	17-11
17.4.1 モジュール・プログラミングにおける接続の有効化と無効化	17-11
17.4.2 プリコンパイル・プログラムにおける接続の有効化と無効化	17-11
17.5 アプリケーションでの接続の使用方法	17-11

第 VI 部 プログラムにおけるデータ操作

18 カーソルの使用方法

18.1 カーソルについて	18-2
18.1.1 カーソルの機能	18-2
18.1.2 カーソルとビューの比較	18-5
18.1.3 カーソルが必要な場合	18-5
18.2 カーソルの分類	18-6
18.3 カーソルのオープンとクローズの制御	18-8
18.4 テーブル・カーソルの使用方法	18-9
18.5 保持可能カーソルの使用方法	18-11
18.6 リスト・カーソルの使用方法	18-13
18.7 スクロール可能なリスト・カーソルの使用方法	18-15
18.8 動的カーソルの使用方法	18-17
18.9 拡張動的カーソルの使用方法	18-18

19 データの挿入、更新、削除

19.1 データベースのロード	19-2
19.2 行の挿入	19-2
19.2.1 INSERT...VALUES 文の使用方法	19-2
19.2.2 INSERT...SELECT 文の使用方法	19-3
19.3 リスト・カーソルを使用した大きなデータ構造の挿入	19-4
19.4 行の更新	19-5
19.4.1 UPDATE 文でのデータの選択	19-5
19.4.2 カーソルを使用した UPDATE 文	19-6
19.4.3 UPDATE...RETURNING 文の使用方法	19-7
19.5 行の削除	19-8
19.6 リスト・データの削除	19-9
19.7 挿入、更新、削除操作でトリガーを使用する方法	19-9

20 複数のスキーマ・オプションの使用方法

20.1 マルチスキーマ・データベースについて	20-2
20.2 SQL モジュール・プロセッサでのマルチスキーマ・データベースの使用方法	20-3
20.2.1 SQL モジュールのデフォルトの設定	20-3
20.2.2 SQL モジュール・ファイルと C プログラムにおけるマルチスキーマ命名の使用方法	20-4
20.3 SQL プリコンパイラでのマルチスキーマ・データベースの使用方法	20-6
20.3.1 SQL プリコンパイラに対するデフォルトの設定	20-7
20.3.2 プリコンパイル・プログラムにおけるマルチスキーマ命名の使用方法	20-8

A SQL 国際言語化オプションの使用方法

A.1 入力および表示形式の制御	A-2
A.2 照合順序の指定	A-3
A.3 照合順序の使用方法	A-3
A.4 Oracle Rdb キャラクタ・セットでの照合順序	A-4

はじめに

このマニュアルについて

このマニュアルでは、SQL (Structured Query Language: 構造化問合せ言語) を使用して Oracle Rdb のデータを格納、変更、検索するホスト言語アプリケーション・プログラムを設計し、開発する方法を説明します。

多くのビジネスにとって、発展を続けるためには、情報を注意深く管理することが必要です。自社のビジネスにとって必須の情報を収集し、保存し、処理する方法を見つけ出さなければなりません。また、データを、一貫性を確保しつつ、安全かつ最新の状態に維持する方法を確立する必要があります。最近、上記のサポートを提供する業界標準のデータベース・ソフトウェア・ツール (SQL など) を採用する企業がますます増えています。

対象読者

このマニュアルは、データベース・アプリケーションを作成、保守するプログラマを対象としています。このマニュアルを十分に活用するには、次の知識が必要です。

- 1つ以上のホスト言語によるプログラミング
- SQL 言語の構文
- リレーショナル・データベース管理システムに関する基本的な概念と用語

SQL を十分理解していない場合には、まず『Oracle Rdb7 Introduction to SQL』をお読みください。この関連マニュアルでは、SQL インタフェースについて紹介しています。

マニュアルの構成

このマニュアルでは、内容を参照しやすいように、各章をカテゴリ別に分類しています。次の表に、各章の簡単な説明を示します。

SQL プログラミングの概要

1 章 (1-1) SQL プログラミング・インタフェースの紹介。

SQL を使用するアプリケーション・プログラムの開発

2 章 (2-1) ホスト言語プログラミングの概要を説明し、プログラム内で SQL 文を記述する方法を示します。

3 章 (3-1) SQL モジュール・プロセッサの紹介。

4 章 (4-1) SQL モジュール言語でプロシージャを記述する際の共通の要素について説明し、単一文および複数文による SQL モジュール・プロシージャを記述する方法を示します。

5 章 (5-1) SQL モジュールとホスト言語ファイル进行处理して実行可能イメージを作成する方法を示します。

6 章 (6-1) ホスト言語ソース・ファイルに SQL 文を埋め込む方法と、SQL プリコンパイラを使用してこのようなファイル进行处理する方法を示します。

7 章 (7-1) オブジェクト・モジュールをリンクして実行可能イメージを作成する方法と、プログラムをデバッグし、実行する方法を示します。

8 章 (8-1) ホスト言語ソース・ファイルでパラメータを宣言し、使用方法を示します。

9 章 (9-1) プログラムで日付時刻型データを使用する方法を示します。

実行時処理

10 章 (10-1) 実行時エラーを検出し、エラー・メッセージを取り出し、エラーのリカバリまたはトランザクションのロールバックを行う方法を示します。

11 章 (11-1) 動的 SQL の使用方法を説明します。

プログラム構造

12 章 (12-1) 複合文の使用方法を説明します。

13 章 (13-1) ストアド・プロシージャとストアド・ファンクションの使用方法を説明します。

14 章 (14-1) 外部プロシージャと外部関数の使用方法を説明します。

プログラムのコンテキスト

15 章 (15-1) リモート・ノード上のデータベースを含めて、データベースに対する接続と接続解除の方法を説明します。

-
16 章 (16-1)	トランザクションを指定して開始する方法を示し、データ・アクセスのオプションについて説明します。
17 章 (17-1)	クエリー、テスト、プロトタイププログラムで使用する SQL 接続の概念を紹介します。

プログラムによるデータ操作

18 章 (18-1)	テーブル・カーソルとリスト・カーソルについて説明し、カーソルを使用してデータを検索する方法を示します。
19 章 (19-1)	データベース内にデータを挿入、更新、削除する方法を示します。
20 章 (20-1)	マルチスキーマ・データベース用のプログラムを記述する方法を示します。
付録 A (A-1)	英語以外の国際的なデータを処理する SQL オプションを示し、Oracle Rdb で使用する照合順番について説明します。

関連マニュアル

Oracle Rdb の詳細は、このドキュメント・セットを構成する他のマニュアル（特に次のマニュアル）を参照してください。

- Oracle Rdb7 Introduction to SQL
- Oracle Rdb7 SQL Reference Manual

『Oracle Rdb7 Release Notes』には、Oracle Rdb ドキュメント・セットを構成するすべてのマニュアルの一覧が掲載されています。

SQL 標準

SQL は、リレーショナル・データベース用のデータ定義言語 (DDL) であると同時にデータ操作言語 (DML) でもあります。SQL インタフェースを使用すると、データベースの作成、データベースとデータのロード、データとデータ定義の読取りと更新を行うことができます。Oracle Rdb への SQL インタフェースは、エントリレベルの SQL 標準 ANSI X3.135-1992、ISO 9075:1992 に準拠しています。この標準は、一般に ANSI/ISO SQL 標準または SQL92 と呼ばれています。

表記規則

このマニュアルで Oracle Rdb とは、Oracle Rdb for OpenVMS およびソフトウェアを表します。Oracle Rdb ソフトウェア・バージョン 7.0 を V7.0 と表します。

Oracle CDD/Repository ソフトウェアは、ディクショナリ、データ・ディクショナリ、またはリポジトリを意味しています。

OpenVMS は、OpenVMS Alpha および OpenVMS VAX オペレーティング・システムを表します。

このマニュアルでは、オペレーティング・システムまたはプラットフォームに固有の情報を識別するアイコンを使用します。複数のプラットフォームまたはオペレーティング・システムに関連する場合は、アイコンを組み合わせるか汎用アイコンを使用します。次に例を示します。

Compaq Tru64 UNIX
=====

このアイコンは、Compaq Tru64 UNIX オペレーティング・システムに固有の情報の始まりを表します。

OpenVMS VAX ===== OpenVMS Alpha =====

このアイコンの組合せは、OpenVMS VAX オペレーティング・システムと OpenVMS Alpha オペレーティング・システムに固有の情報の始まりを示します



この菱形の記号は、特定のオペレーティング・システムまたはプラットフォームに固有の情報の終わりを示します。

例では、特に指定されていない限り、各行の終わりには改行があるものとします。各行を入力したら、最後に [Enter] キーを押してください。

例では、プロンプトが記載されていない場合があります。一般に、対話型シーケンスを正確に示す場合は明記しますが、それ以外の場合は省略します。

このマニュアルでは、次の表記規則も使用します。

e, f, t 印刷されたマニュアルの索引欄には、ページ番号の後に小文字の e, f, t が記載されている場合があります。e, f, t は、それぞれ、例、図、表の参照を表します。

⋮

例に縦の省略記号がある場合は、例に直接関連しない情報が省略されていることを表します。

...

文の横の省略記号は、文の中で例に直接関連のない要素が省略されていることを表します。

太字のテキスト テキストの太字の部分は、そのテキストで定義する用語を表します。

<>

山カッコは、ユーザーが指定する名前を囲みます。

[]

大カッコは、オプションの句を囲みます。1 つを選択するか、指定を省略することができます。

\$

ドル記号は、OpenVMS では DIGITAL Command Language プロンプトを表し、Compaq Tru64 UNIX では Bourne シェル・プロンプトを表します。

技術的な変更と新機能

この項では、Oracle Rdb バージョン 6.0 の前回リリースから、新しく追加または変更された内容について説明します。

バージョン 7.0 の新機能や技術的な変更は、主に次のようなものがあります。

- 複数のトランザクションに渡ってカーソルをオープンした状態に保持
複数のトランザクションに渡って、SQL カーソルをオープンしておくことが可能になりました。DECLARE CURSOR 文の WITH HOLD 句を使用すると、カーソルはトランザクションが終了した後もオープンした状態になります。新しい SQL トランザクションを開始するとき、オープンしたカーソルの位置は維持されます。詳細は、18.5 項 (18-11) を参照してください。
- ストアド関数の作成
ストアド・プロシージャの定義に加えて、ストアド関数の定義が可能になりました。ストアド関数は、1 つ以上の SQL 文によって、Oracle Rdb データベース上で実行する一連の操作です。ストアド関数は、入力パラメータを受け取り、結果を 1 つ返します。ストアド関数の起動には、値式に関数名を指定します。詳細は、13 章 (13-1) を参照してください。
- ストアド関数の戻り値
SQL は、RETURN 文で、ストアド関数の結果を返します。詳細は、13.3.2 項 (13-7) を参照してください。
- 複合文での CALL 文の使用
複合文で CALL 文を使用できるようになりました。したがって、ストアド・プロシージャやストアド関数で複合文を使用すれば、他のストアド・プロシージャをコールできます。また、CALL 文で外部プロシージャを起動することも可能です。詳細は、12.3.8 項 (12-14) を参照してください。
- 複合文での SIGNAL 文の使用
複合文で、SIGNAL 文を使用できるようになりました。SIGNAL は、SQLSTATE として使用される 1 文字の値式を受け取ります。現在のルーチンとすべてのコール先ルーチンは終了し、シグナルを発行した SQLSTATE は、アプリケーションに渡されます。詳細は、12.9 項 (12-22) を参照してください。
- 外部プロシージャの使用と、外部ルーチンによるデータベース操作
SQL は、外部プロシージャを提供し、外部ルーチンに SQL 文を含めることにより、新しいスキーマ・インスタンスのバインドと、外部ルーチンからのデータベース操作を可能にしています。外部ルーチンとは、外部プロシージャまたは外部関数であり、C や FORTRAN などの 3GL 言語で記述され、共有可能イメージや共有オブジェクトにリンクして、データベース・スキーマ内に登録します。詳細は、14 章 (14-1) を参照してください。

- モジュールのカスケード削除
DROP MODULE CASCADE 文は、モジュールと、それを参照するオブジェクトを削除します。詳細は、13.6 項 (13-12) を参照してください。
- 関数とプロシージャの削除
外部プロシージャや、ストアド関数やストアド・プロシージャの削除が可能になりました。詳細は、13.6 項 (13-12) と 14.4 項 (14-8) を参照してください。
- 複合文での変数の宣言での DEFAULT、CONSTANT、UPDATABLE 句の指定
変数のデフォルト値を、サブクエリー、条件、文字、日付や時間、数値など、任意の値式に設定できます。さらに、変数は、指定されたドメインから、デフォルトを継承できるようになりました。詳細は、12.3.1 項 (12-4) を参照してください。
- ヘッダー・ファイルを使った Compaq Tru64 C 情報メッセージの削除
SQL は、明示的にコールした SQL ルーチンのプロトタイプを提供することにより、情報メッセージを排除したヘッダー・ファイルを提供します。詳細は、6.7.3 項 (6-16) を参照してください。
- C プログラムでの sql_sqlda.h ヘッダー・ファイルの使用
C プログラムでヘッダー・ファイルを使用して、SQLDA と SQLDA2 構造の定義を取得できるようになりました。前のバージョンでは、この構造の定義を取得するには、SQL プリコンパイラの EXEC SQL INCLUDE SQLDA 文や EXEC SQL INCLUDE SQLDA2 文を実行する方法しかありませんでした。詳細は、11.3.2 項 (11-10) を参照してください。
- SQLCA 構造への外部参照の宣言
C 言語で SQL プリコンパイラを使用する場合、SQLCA 構造への外部参照を宣言できるようになりました。オプションの EXTERNAL キーワードを EXEC SQL INCLUDE SQLCA 文の最後に指定してください。詳細は、8.6.2 項 (8-10) を参照してください。
- SQL を使用したデバッグ・フラグの設定
SQL は、対話型で動的な SQL 用に、新しい SET FLAGS 文をサポートします。SET FLAGS 文は、実行中に、データベース・システムのデバッグ・フラグを有効化または無効化します。また、複合文の変数の内容のモニターも可能です。詳細は、12.7 項 (12-18) を参照してください。
- SQL_ALTERNATE_SERVICE_NAME 構成パラメータの使用
構成パラメータ SQL_ALTERNATE_SERVICE_NAME では、代替 TCP/IP サービスの指定が可能です。代替サービスは、OpenVMS や Compaq Tru64 UNIX クライアントから、TCP/IP を使用してバージョンが異なる OpenVMS データベースにアクセスするときに特に便利です。詳細は、15.2.6 項 (15-10) を参照してください。

バージョン 6.1 での新機能や技術的な変更は、主に次のようなものがあります。

- Compaq Tru64 UNIX システムでの SQL の使用
このマニュアルでは、Compaq Tru64 UNIX でのコンパイルやリンクなど、Compaq Tru64 UNIX システムでの SQL の使用方法に関する情報を掲載しています。

-
- リモート・アクセスでのユーザー認証
Oracle Rdb では、データベースに接続する SQL 文で、ユーザー名とパスワード情報を明示的に指定することができます。さらに、パラメータや新しいコマンド行修飾子を指定すれば、SQL モジュール言語やプリコンパイルした SQL プログラムに情報を渡すことができます。構成パラメータを使用して、Oracle Rdb に情報を渡すこともできます。詳細は、15 章 (15-1) を参照してください。
 - SQL モジュール言語向けの INTEGER データ型の変更
SQL モジュール言語構文の拡張により、正確な INTEGER モジュール・パラメータをバイト数で指定することが可能になりました。詳細は、8.12.2 項 (8-27) を参照してください。
 - 移植可能な SQL ルーチンの使用
次のルーチンは、OpenVMS と Compaq Tru64 UNIX の両方のオペレーティング・システムで使用できます。
 - `sql_get_error_text`
エラー・テキストと書式付きの出力を、プログラムに渡します。
`SQL$GET_ERROR_TEXT` ルーチンと似ていますが、`SQL$GET_ERROR_TEXT` ルーチンは、OpenVMS システムのみで使用可能です。詳細は、10.3.2 項 (10-28) を参照してください。
 - `sql_get_message_vector`
最後の SQL 文のステータス情報を、メッセージ・ベクターから取得します。詳細は、10.2.4 項 (10-13) を参照してください。
 - `sql_get_error_handler`、`sql_register_error_handler`、`sql_deregister_error_handler`
このルーチンは、Compaq Tru64 UNIX 上で動作するという点以外は、以前の Oracle Rdb バージョンと同じです。詳細は、10.2.5 項 (10-18) を参照してください。
 - `sql_signal`
シグナルを発行し、SQL 文の実行中にエラーが発生したことを示します。
`SQL$SIGNAL` ルーチンに相当しますが、`SQL$SIGNAL` ルーチンは OpenVMS システムのみで使用可能です。詳細は、10.3.1 項 (10-27) を参照してください。

このマニュアルでは、Oracle Rdb の技術的な変更点と新機能を掲載するとともに、さらに明確な説明や修正が加えられています。

第I部

SQL プログラミングの概要

第1部では、アプリケーション・プログラムでSQLを使用する方法について、高度な視点から説明します。

SQL プログラミングの概要

この章では、Oracle Rdb に対する SQL インタフェースを使ったプログラミングについて説明します。この章の各項では、次の内容を説明します。

- 使用可能なプログラミング・インタフェース
- プリコンパイルした SQL よりも優れた SQL モジュール言語の特長
- SQL インタフェースの主な特徴を示すサンプル・プログラムの検索方法

Oracle Rdb は、複雑なデータベース・ニーズに対応するため、SQL モジュール・プロセッサと SQL プリコンパイラという 2 つのプログラミング・インタフェースを提供しています。このプログラミング・インタフェースを使用して、フォーマットしたレポートの作成、複雑な計算、データの更新などを実行するアプリケーションを開発することができます。

1.1 2つのSQLプログラミング・インタフェース

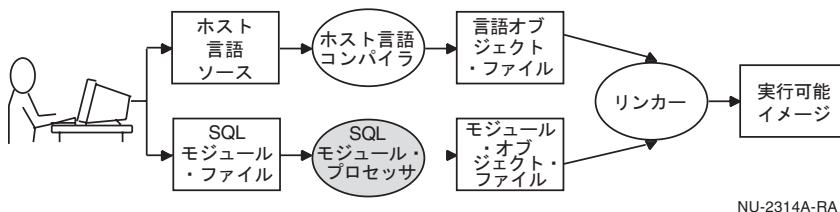
SQL とホスト言語プログラムを組み合わせるには、2つの方法があります。SQL 言語の別モジュールを作成する方法と、ホスト言語プログラムに直接入力する方法です。

- **SQL モジュール・プロセッサ**は、SQL 文を含むモジュール・ファイルをコンパイルし、オブジェクト・ファイルを作成します。モジュールのオブジェクト・ファイルとホスト言語プログラムのオブジェクト・ファイルをリンクすると、実行可能イメージが作成されます。
- **SQL プリコンパイラ**は、ホスト言語プログラムに埋め込んだ SQL 文とプログラム・コードを、ホスト言語プログラムのコンパイラが処理可能な形式に変換します。

1.1.1 SQL モジュール・プロセッサ

サポート対象の任意のホスト言語について、SQL モジュール言語を使用して、SQL プロシージャの SQL モジュール・ファイルを作成できます。ホスト言語のソース・ファイル内に、SQL プロシージャのコールを指定できます。図 1-1 (1-2) は、SQL モジュール言語を使用してプログラムを作成する手順を示しています。

図 1-1 プログラム開発での SQL モジュール・プロセッサの使用方法



この手順では、ホスト言語のソース・ファイルと SQL モジュール・ファイルを作成し、2つを並行して処理します。ホスト・プログラムはホスト・コンパイラでコンパイルし、SQL モジュール・ファイルは SQL モジュール・プロセッサでコンパイルします。ホスト言語のコンパイルによってホスト言語のオブジェクト・ファイルが作成され、SQL モジュールのコンパイルによって SQL オブジェクト・ファイルが作成されます。この2つのオブジェクト・ファイルをリンクすると、実行可能なイメージが作成されます。

SQL モジュール・ファイルは、1つ以上の SQL プロシージャで構成され、各プロシージャは1つ以上の SQL 文で構成されます。SQL モジュール・プロセッサは、SQL モジュール・ファイルのソース・コードの構文とセマンティクスをチェックし、単一文や複数文の SQL プロシージャをコンパイルします。

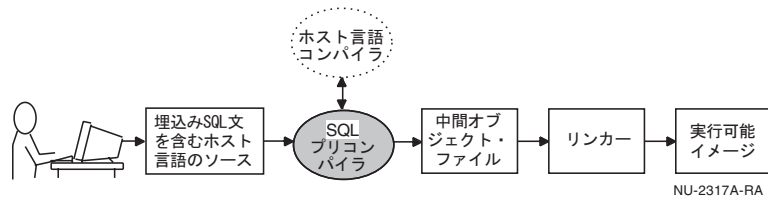
SQL モジュール・プロセッサでは、次のような処理も実行できます。

- モジュール名、ユーザー認証 ID、モジュールの最適化レベルなど、モジュールの各種特性の定義
- エラー・メッセージ用のリスティング・ファイルの作成など、出力情報の制御
- 違反または標準に適合しない SQL 構文の検出

1.1.2 SQL プリコンパイラ

ホスト言語が SQL プリコンパイラのサポート対象であれば、SQL 文をホスト言語のソース・ファイルに直接埋め込むことができます。図 1-2 (1-3) は、SQL 文を使用するプログラムを、SQL プリコンパイラを使って作成する手順を示しています。

図 1-2 プログラム開発での SQL プリコンパイラの使用手順



SQL プリコンパイラを実行して、SQL 文を埋め込んだプログラムを処理します。SQL プリコンパイラは、埋め込まれた SQL 文の構文とセマンティクスをチェックし、SQL 構文をコンパイルし、さらにホスト言語のコンパイラを起動してオブジェクト・モジュールを作成します。オブジェクト・モジュールをリンクすると、実行可能なイメージを作成できます。

SQL プリコンパイラでは、次のような作業も実行できます。

- エラー・メッセージ用のリスティング・ファイルの作成など、出力情報の制御
- 違反または標準に適合しない SQL 構文の検出
- SQL プリコンパイラにおいて、DOUBLE データ型の G 浮動小数点または D 浮動小数点の解釈を指定

1.2 プログラミング・インタフェースの選択

Oracle Rdb では、アプリケーション・プログラムの開発には、SQL プリコンパイラではなく、SQL モジュール・プロセッサを使用することを推奨しています。Oracle Rdb は、既存のアプリケーションとの互換性と、プリコンパイラが必要になるプログラミング環境向けに、SQL プリコンパイラを提供しています。

SQL モジュール・プロセッサは、処理能力、柔軟性、効率の点でプリコンパイラよりも優れており、他にも次のような利点があります。

- さらに明確になった言語の抽象化
SQL 文は別のモジュールに含まれるため、ホスト言語のソース・プログラムには、ホスト言語の文のみが格納されています。
- ANSI/ISO 規格が存在する言語でプログラムを記述すると、SQL 文を SQL モジュールに分離することによって、規格外のコードを埋め込む必要がなくなります。^{*1}
- モジュール性と共有性の向上
SQL モジュール言語は別のモジュールに分離されているので、別のホスト言語プログラムで SQL プロシージャをコールでき、アプリケーション・プログラムの柔軟性や保守性が向上します。
たとえば、複雑なデータベース・トランザクションを実行する 1 つのモジュールを、様々なアプリケーション・プログラムで使用することができます。または、このモジュールにあるプロシージャを、様々なホスト言語で記述した各種アプリケーション・プログラムからコールすることもできます。
- CASE ツールとの統合性を強化
SQL モジュール・プロセッサと、コンピュータ支援ソフトウェア・エンジニアリング (Computer-Aided Software Engineering: CASE) ツールとの統合性が向上しています。一般的に、SQL モジュール・プロセッサでソース・コードやオブジェクト・コードを作成すると、格納、保守、共有などの作業を簡易化できます。
これに対して、SQL プリコンパイラは、ホスト言語のソース・コードを変更するので、たとえば言語特有のエディタやシンボリック・デバッガをフルに活用することはできません。
- アプリケーション・ジェネレータのターゲットを改善
アプリケーション・ジェネレータ (SQL コードを自動生成するプログラム) では、SQL プリコンパイラ用に正しいホスト言語コードを生成するのではなく、直接コンパイル可能な SQL モジュール・プロセッサ向けのコードを作成するので、作業は大幅に簡易化されています。
- あらゆるホスト言語での使用が可能
モジュール言語によって、SQL 文をプロシージャに埋め込み、あらゆるホスト言語からコールできます。SQL プリコンパイラがサポートする言語には、制限があります。
- 多彩な言語機能
SQL モジュール言語では、プリコンパイラがサポートしないホスト言語機能も使用できます (C のポインタ変数、ブロック構造、マクロ、ユーザー定義のデータ型、配列要素の参照など)。
ポインタ変数をサポートするプログラムは、動的 SQL の機能をフルに活用し、SQL モジュール言語によって、SQL 記述子領域 (SQL Descriptor Area : SQLDA) や拡張 SQL 記述子領域 (SQL Descriptor Area : SQLDA2) を使用できます。

^{*1} SQL プリコンパイラは、埋込み文を外部プロシージャ・コールに置き換えて、ホスト言語ソース・ファイルを書き直すため、ファイルは ANSI/ISO 規格準拠となります。

1.3 オンライン・プログラムのサンプルの検索

SQL では、SQL プログラミングの理解を深めるために、サンプルのデータベースとプログラムが用意されています。

OpenVMS VAX ≡≡≡ OpenVMS Alpha ≡≡≡

OpenVMS では、論理名 SQL\$SAMPLE で、サンプル・ディレクトリが定義されています。



Compaq Tru64 UNIX ≡≡≡≡≡

Compaq Tru64 UNIX では、次のディレクトリにサンプル・ディレクトリがあります。

`/usr/lib/dbs/sql/vnn/examples`

サブディレクトリ vnn は、v70 のように、Oracle Rdb のバージョンを表します。◆

サンプル・ディレクトリには、サンプルの **personnel** データベースの定義とクエリーの両方を実行するソース・プログラムが格納されています。システムの言語サポートの対象になっているソース・プログラムの実行可能イメージを作成し、イメージを実行して、プログラムの処理を確認してください。プログラム・ソースは印刷できるので、カスタム・アプリケーションを作成するときの参考にしてください。

第II部

SQL を使用したアプリケーション・プログラムの開発

第2部では、SQL を使用したアプリケーション・プログラムの開発方法を説明します。ここでは、次の内容について説明します。

- アプリケーションの開発サイクルについて
- SQL モジュール言語の使用方法
- プリコンパイル SQL の使用方法
- プログラムのリンク、実行およびデバッグ
- SQL と情報を交換するためにプログラムが使用するパラメータの指定方法
- SQL の日付時間データ型の使用方法

SQL プログラム開発工程

SQL アプリケーション・プログラムの設計は、ホスト言語プログラムの設計と同じ手順ですが、特別な考慮事項が必要になります。たとえば、SQL 文の構文とホスト言語プログラムに SQL 文を記述する方法（埋込みまたはモジュール）は、SQL アプリケーション・プログラムの設計に影響を及ぼします。

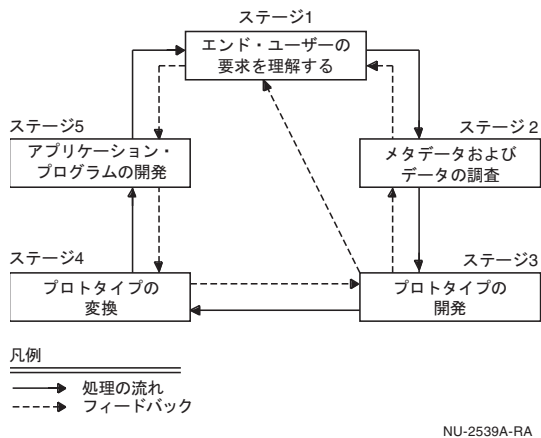
この章では、アプリケーション・プログラム開発工程の各ステージの概要を説明します。各項の内容は次のとおりです。

- アプリケーション・プログラム開発工程の概要
- エンド・ユーザーの要求の理解
- メタデータとデータの調査
- アプリケーションのプロトタイプの開発
- プロトタイプからアプリケーション・プログラムへのエンド・ユーザー変換
- アプリケーション・プログラムの開発

2.1 アプリケーション・プログラム開発工程の概要

図 2-1 (2-2) に、アプリケーション・プログラム開発工程の各ステージとステージ間のエンド・ユーザー相互作用を示します。

図 2.1 アプリケーション・プログラム開発工程の各ステージ



多くのプロセスと同様に、アプリケーション・プログラム開発工程には各ステージを経由する最適な制御フローがありますが、ステージ間のタイムリーなフィードバックと繰り返しによって効果と効率が向上します。

2.2 エンド・ユーザーの要求の理解

最初のステージは、エンド・ユーザーがデータベース・システムから何を要求するかを理解することです。それから、管理者はその要求を SQL に変換できます。すべての目標は、最終的なアプリケーション・プログラムでユーザーの要求を満たすことです。

しかし、完成したアプリケーションではユーザーが十分に満足しない場合がほとんどです。「言ったのは確かだがそういう意味ではない」というのはよくあるクレームです。この問題を回避するために、管理者とユーザーが要求を確実に理解し、両者の理解に相違がないことを確認してください。

場合によっては、ユーザーがデータベース・システムとその使用方法について十分に理解していないことがあります。管理者は、データベース・システムの概要とデータベース・システムを使用できることとできないことをユーザーに説明し、ユーザーがより具体的な要求を作成できるようにしてください。結果として、要求が変わる場合もあります。また、時間が経過してユーザーがアプリケーションについて考えたりアプリケーションを使用したりする間に元の要求が発展します。当初は要求の原因になっていた条件が変化したために、プ

ログラムの要件が変わることもあります。Oracle Rdb では、様々なインタフェースや柔軟な分散環境によって、このような条件の変化に対応します。

エンド・ユーザーの要求を管理者が適切に理解しているかどうか、定期的に確認してください。アプリケーション開発工程の全期間を通じて定期的にユーザーとのコミュニケーションをとれば（特にプロトタイプの検証）、ユーザーが最終的なアプリケーション・プログラムに満足する可能性は高くなります。

2.3 メタデータとデータの調査

エンド・ユーザーの要求を適切に満足するクエリーすなわち SQL 文を記述するには、データベース・システムで使用可能なメタデータとデータを調べる必要があります。

一般には、次の問題を確認してください。

- 要求を満たすデータベース内に現在どんなテーブル、ビュー、列、関数、プロシージャが存在するか。
- メタデータ（テーブル、ビュー、列、インデックス、制約）を作成する必要があるか。要求を満たすためにデータを変更する必要があるか。
- 列にはどんなインデックスや制約が定義されているか。インデックスや制約を追加または削除する必要があるか。
- データベース・システム内のデータの量と質はどうか。要求を満たすのに十分か。データは正確か、妥当か、最新か。データは完全か、または少なくとも要求を満たすのに十分な包括性があるか。
- このクエリーは今回限りの実行か、ユーザーは何度も同じ問合せを要求するのか。クエリーを頻繁に実行する場合は、格納して再利用するか、それともプログラムに組み込むか。
- データにアクセスするのに必要な権限。

このステージで上記の問題を確認する場合は、対話型のユーザー・インタフェースまたはプログラミング・インタフェースを利用できます。たとえば、対話型 SQL の SHOW 文を使用するとメタデータを確認でき、SELECT 文を使用するとシステム・テーブルを確認できます。

2.4 プロトタイプの作成

次のステージでは、エンド・ユーザーの要求を満足するのに役立つプロトタイプを作成します。プロトタイプを作成する場合、しばしば繰り返しユーザーに確認しないと適切なプロトタイプが得られないことがあります。

プロトタイプを作成するのは、対話型 SQL でもプログラミング SQL でもかまいません。対話型 SQL 文で作成し、テストしてからモジュールに組み込んだりホスト言語プログラムに埋め込んだりすれば、アプリケーション・プログラムのデバッグが容易になり、プログラム開発工程が短縮されます。

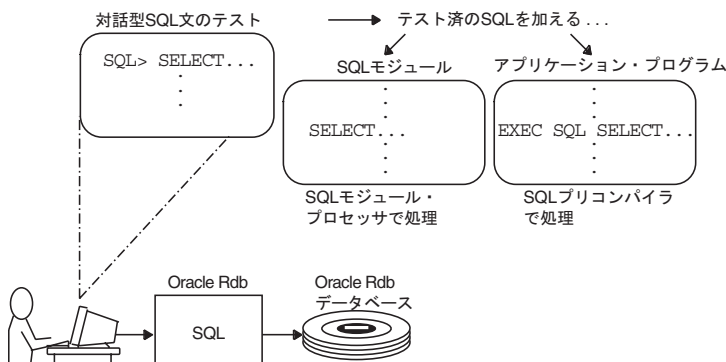
対話型でプロトタイプを作成すると、次の処理が可能です。

- 構文上や意味上のエラーの検出と修正
対話型 SQL は、SQL 文の構文エラーの修正に便利です。EDIT 文を使用すると、構文エラーを修正できます。
- クエリーの有効性の評価
クエリーの効率が上がるように改善できます。たとえば、いくつかの方法でクエリーを試して（サブクエリーの結合との比較など）どれが効率的かを調べることができます。インデックスを追加してパフォーマンスを改善することもできます。
- プログラムで実行するデータ検証のタイプの理解
対話型 SQL を使用すると、入力と出力の値をテストし、アプリケーションで適切に処理できます。また、現在のテーブルまたは列の制約を超える入力操作がないかを確認する妥当性チェックを追加する必要があるかを判断できます。
- 実行時エラーの予測
対話型 SQL では、プログラムで発生する見込みのある実行時エラーを明確に認識できます。無効な入力、制約違反、リソースの競合によるロックの競合など、特定の実行時エラーが発生する状況をテストできます。

適切な（構文上正しく、しかも当初のエンド・ユーザー要求を満足する）プロトタイプを作成すれば、それをスクリプトに保存して容易に再実行することも、ファイルに保存してプログラムに変換することもできます。図 2-2 (2-4) は、このプロセスを説明しています。

要求に含まれるクエリーを一度だけ実行する場合は、アプリケーション・プログラムを開発する必要はありません。出力をデータ・ファイルに保存してください。ユーザーがクエリーを頻繁に実行すると思われる場合は、クエリーからビューへの変換を考えてください。

図 2-2 対話型 SQL をプロトタイプ作成ツールとして使用する方法



NU-2406A-RA

2.5 プロトタイプからアプリケーション・プログラムへの変換

対話方式で作成した SQL 文 (1 つ以上) を、選択したプログラミング・インタフェースで使用する構文に変換します。SQL 文によっては、対話方式で使用する構文とプログラミング・インタフェースで使用する構文が若干異なる場合があります。また、値を表現するのにリテラルでなく変数を使用する必要があります。

次の点を確認することが重要です。

- アプリケーション・プログラムと SQL 文の間でどんな値の伝達が必要か。
モジュール・プロセッサのメイン・パラメータ、またはプリコンパイラのホスト変数を定義する必要があります。
- どの列 (データの取得用または記憶用) で NULL 値をとり得るか。
このような列に対応するメイン・パラメータにはインジケータ・パラメータを宣言する必要があります。
- アプリケーション・プログラムを実行する場合に、予期されるエラーと予期されないエラーの両方をどう処理するか。
少なくとも、実行ステータス・パラメータを定義する必要があります。通常は、アプリケーション・プログラム内でエラーを処理して問題を回避するか、アプリケーション・プログラム実行中に制御機能を追加します。
メイン・パラメータとインジケータ・パラメータの使用の詳細は、8 章 (8-1) を参照してください。10 章 (10-1) では、実行ステータス・パラメータの使用方法について説明します。
- 各パラメータの SQL データ型は何か。これに対応して、アプリケーション・プログラムではどんなデータ型を使用するのか。
ホスト言語プログラムとプロシージャの間で、データ型を適切にマップする必要があります。
- アプリケーション・プログラムではどんなデータ妥当性コードが必要か。
現行の制約が実行する妥当性チェック以外に、アプリケーション・プログラムでデータの妥当性チェックを行う必要があります。さらに、データベースの制約を使用するかわりに、エンド・ユーザーが入力するデータを検査してから、データベースに挿入することもできます。アプリケーション・プログラム内でデータを検査すると、妥当性チェックがデータベース・トランザクションのスコープの外で行われるので、ロックが最小になります。
- どんな SQL キーワードやユーザー定義の名前 (列やテーブルなど) がホスト言語プログラム・キーワードと競合するのか。
確立されたデータベース・システムでは、列名の変更は容易ではありません。ただし、ホスト言語がサポートする名前のビューを作成できます。それから、ビューの各列に対応するパラメータを宣言してください。
キーワードの一覧は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

2.6 アプリケーション・プログラムの開発

最終ステージは、アプリケーション・プログラムの開発です。このステージには複数のステップがあり、どのプログラミング・インタフェースを使用するかによって変わります。

基本的なステップは次のとおりです。

1. モジュールとホスト言語のソース・コードを編集します。
2. モジュールとホスト言語のソース・コードをコンパイルし、必要に応じて構文エラーと意味上のエラーを修正します。
3. オブジェクト・ファイルとライブラリをリンクして実行可能イメージを作成します。
4. 実行可能イメージを実行して結果を検証し、必要に応じてイメージを簡素化します。

SQL モジュール・プロセッサを使用してアプリケーション・プログラムを開発する手順の詳細は、3 章 (3-1) から 5 章 (5-1) までを参照してください。

SQL プリコンパイラを使用してアプリケーション・プログラムを開発する手順の詳細は、6 章 (6-1) を参照してください。

SQL モジュール言語の概要

この章では、SQL 文を使用してデータベースへアクセスするアプリケーションを作成する SQL モジュール・プロセッサ（SQL モジュール言語）の使用方法について説明します。以降の各項では、次に示す項目を説明します。

- SQL モジュール言語を使用したアプリケーションの開発
- SQL モジュールのソース・ファイルの作成
- ホスト言語プログラムからの SQL モジュール・プロシージャのコール
- 移植可能なコードの記述
- SQL モジュール・プロセッサについての詳細情報の検索

3.1 SQL モジュール言語アプリケーション・プログラムの開発方法：基本手順

SQL モジュール言語を使用すると、1 つ以上のプロシージャを含む **SQL モジュール・ファイル**を作成できます。各プロシージャには、1 つ以上の SQL 文が含まれています。

従来の第三世代言語（3GL）で記述されたプログラムまたは Oracle Rally 外部リンクから、SQL モジュール内のプロシージャをコールします。

図 3-1 (3-3) の番号に対応する次の手順に従って、SQL モジュール言語アプリケーション・プログラムを開発できます。

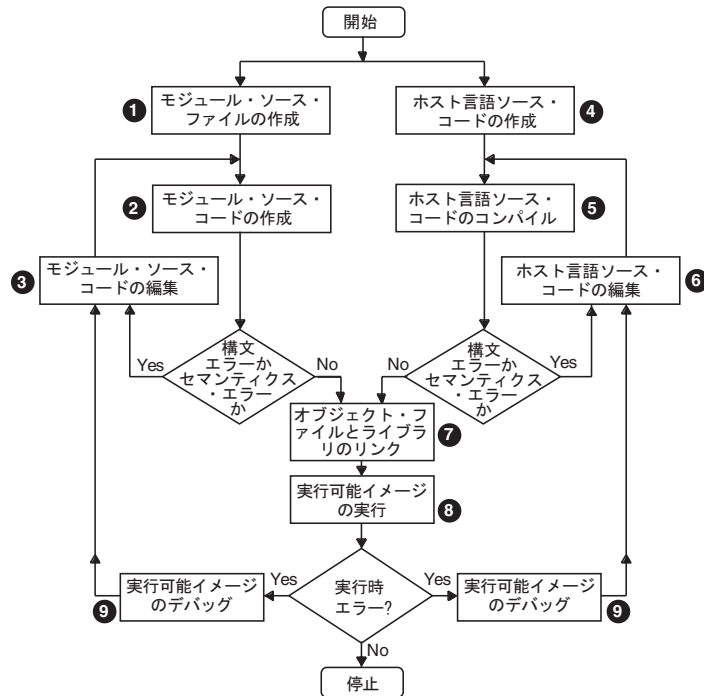
1. SQL モジュール・ファイルを作成します。
SQL モジュール・プロセッサには、SQL モジュール・ファイルを作成する SQL モジュール言語があります。このモジュールには、1 つ以上のプロシージャが含まれ、それぞれにパラメータ宣言と 1 つ以上の SQL 文が含まれています。ホスト言語プログラムは、特別な SQL モジュール・プロシージャをコールし、そのモジュール・ファイルのプロシージャ内のパラメータ宣言に対する数値とデータ型に対応した実際のパラメータを指定します。
2. SQL モジュール・プロセッサを使用して、SQL モジュール・ソース・コード・ファイルをコンパイルします。
SQL モジュール・プロセッサ実行可能イメージを実行し、同じコマンド行で SQL モジュール・ファイル名といくつかのオプションの修飾子を指定して、SQL モジュール・ファイルをコンパイルします。
各修飾子とその使用例の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。
3. 必要に応じて、SQL モジュール・ソース・コードを編集します。
SQL モジュール・ファイルを作成するか、コンパイルした後、ユーザーや SQL モジュール・プロセッサが発見したエラーを修正するためにモジュール・ファイルを十分に確認してください。
4. ホスト言語ソース・コードの作成。
OpenVMS 上では、OpenVMS コール標準をサポートしている任意のホスト・プログラム言語で SQL モジュール・プロセッサを使用できます。◆
CompaqTru64 UNIX 上では、Compaq Tru64 C、Compaq Tru64 UNIX C、Compaq Tru64 COBOL、Compaq Tru64 FORTRAN および Compaq Tru64 Pascal で SQL モジュール・プロセッサを使用できます。◆
5. ホスト言語ソース・コードのコンパイル。
ホスト言語コンパイラを使用して、オブジェクト・モジュールを作成します。コンパイルが成功すると、コンパイル・プロセスはホスト言語オブジェクト・ファイルを作成します。

OpenVMS OpenVMS
VAX Alpha

Compaq Tru64 UNIX

6. 必要に応じて、ホスト言語モジュール・ソース・コードを編集します。
コンパイル中にホスト言語プログラムがエラーを生成したら、ホスト言語ソース・コードを編集して、エラーを修正します。
7. ホスト言語が作成したオブジェクト・モジュールと SQL モジュール・プロセッサのコンパイル結果をリンクして、実行可能イメージを作成します。
8. 実行可能イメージを実行して、その結果を検証します。
9. 実行時エラーを検出した場合には、実行可能イメージをデバッグします。
エラーの原因を発見した後、ホスト言語プログラム、モジュール・ファイルまたはその両方を編集し、有効なオブジェクト・ファイルが作成されるまで、必要に応じて 1 つ以上のファイルを再コンパイルします。

図 3-1 SQL モジュール・プロセッサでのアプリケーションの開発



NU-2542A-RA

3.2 SQL モジュール・ソース・ファイルの作成方法

SQL モジュール・ファイルを作成する場合、SQL 文、プロシージャおよび特別な SQL キーワードのみを含む言語である **SQL モジュール言語** で記述されたソース・ファイルを作成します。SQL モジュール・ファイルには、次のコンポーネントが含まれています。

- コメント (オプション)
- 次を含むモジュール・ヘッダー情報
 - MODULE 名句
 - DIALECT 句 (オプション)
 - CHARACTER SET 句 (オプション)
 - LANGUAGE 句
 - CATALOG 名句 (オプション)
 - SCHEMA 名句 (オプション)
 - AUTHORIZATION 識別子句 (オプション)
 - ALIAS 句 (オプション)
 - 他のオプション句、たとえば文字の長さ、日付形式、キーワードの取り扱い、引用符の解釈、実行時の権限の確認、表示の更新規則およびパラメータの前にコロンがつくかどうかなどを設定する句
- DECLARE 文セクション (オプション)
- それぞれが複合文内に 1 つの SQL 文または 1 つ以上の SQL 文とこれらの文で使用されるパラメータを含む 1 つ以上のプロシージャ。
複合文を含むプロシージャは、**複合文プロシージャ**と呼ばれます。

3.2.1 項 (3-6) から 3.2.11 項 (3-11) では、SQL モジュールの作成方法についての詳細を説明しています。4 章 (4-1) では、SQL モジュール・プロシージャの記述についての詳細を説明しています。

例 3-1 (3-4) に、モジュール・コンポーネントを識別する一連の番号を付けた SQL モジュール・ファイルを示します。各番号については、例の後に示す番号付きリストで説明します。

例 3-1 SQL モジュールの部分

```
-- The procedures in this SQL module are called by the C program (1) (3-6)
-- sql_intro_load_empl_h.c.
--
-----
-- Header Information Section
-----
```

```

(2) (3-6)
MODULE          INTRO_LOAD_EMPL_C      -- Module name
DILECT          SQL92
LANGUAGE        C                      -- Language of calling program
AUTHORIZATION    SAMPLE_USER           -- Authorization ID
ALIAS            RDB$DBHANDLE           -- Default alias
PARAMETER COLONS                                -- Parameters are prefixed by colons
-----
-- DECLARE Statements Section
-----
-- Declare the alias using the file name.
DECLARE ALIAS FOR FILENAME intro_personnel (3) (3-6)
-----
-- Procedure Section
-- In every procedure, declare SQLCODE, a parameter that stores a value
-- that represents the execution status of SQL statements.
-----
-- This procedure uses the executable statement, SET TRANSACTION, to start
-- a transaction. The EMPLOYEES, JOB_HISTORY, and DEPARTMENTS tables are
-- reserved because they are used in constraint checking.
PROCEDURE SET_TRANS (4) (3-6)
  (SQLCODE);
  SET TRANSACTION READ WRITE RESERVING
    EMPLOYEES FOR EXCLUSIVE WRITE,
    JOB_HISTORY FOR SHARED READ,
    DEPARTMENTS FOR SHARED READ;
-- This procedure inserts the employee data into the EMPLOYEES table.
PROCEDURE INSERT_DATA (4) (3-6)
  (SQLCODE
-- Declare the parameters by which the values are passed between the SQL
-- module and the host language.
:EMPLOYEE_RECORD RECORD
  P_EMPLOYEE_ID CHAR(5),
  P_LAST_NAME CHAR(14),
  P_FIRST_NAME CHAR(10),
  P_MIDDLE_INITIAL CHAR(1),
  P_ADDRESS_DATA CHAR(25),
  P_CITY CHAR(20),
  P_STATE CHAR(2),
  P_POSTAL_CODE CHAR(5),
  P_BIRTHDAY CHAR(10),
  P_SEX CHAR(1)
  END RECORD);
-- The list of names that follows the INSERT clause identifies the columns
-- in the table that are to be used for the insert operation.
-- The list of names in the VALUES clause corresponds to the variables
-- declared in the procedure.

```

```
INSERT INTO EMPLOYEES
    (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
     ADDRESS_DATA, CITY, STATE, POSTAL_CODE, BIRTHDAY, SEX )
VALUES
    (:EMPLOYEE_RECORD.P_EMPLOYEE_ID, :EMPLOYEE_RECORD.P_LAST_NAME,
     :EMPLOYEE_RECORD.P_FIRST_NAME, :EMPLOYEE_RECORD.P_MIDDLE_INITIAL,
     :EMPLOYEE_RECORD.P_ADDRESS_DATA, :EMPLOYEE_RECORD.P_CITY,
     :EMPLOYEE_RECORD.P_STATE, :EMPLOYEE_RECORD.P_POSTAL_CODE,
     CAST(:EMPLOYEE_RECORD.P_BIRTHDAY AS DATE ANSI),
     :EMPLOYEE_RECORD.P_SEX );
.
.
.
```

例 3-1 (3-4) の番号は、次の各項目を示しています。

- (1) コメント
- (2) モジュール・ヘッダー
- (3) 宣言文
- (4) プロシージャ

3.3 項 (3-12) では、ホスト言語プログラムから SQL モジュール・プロシージャをコールする方法を説明しており、例 3-1 (3-4) で示されている SQL モジュール `sql_intro_load_empl_c.sqlmod` をコールするプログラムを示しています。

3.2.1 SQL モジュール内に空白行とコメントを含める方法

二重ハイフン (--) は、行のこれ以降のテキストはコメントであることを示します。SQL モジュール・プロセッサがソース・ファイル进行处理する場合、二重ハイフンの右側にあるテキストを無視します。例 3-1 (3-4) で示したように、SQL モジュールのソース・ファイルを読んで理解しやすくするためにコメントを指定したり、行を空白のままにすることができます。

3.2.2 モジュールの命名

モジュールを命名するには、MODULE 句を指定する必要があり、オプションでモジュールの最初の行（コメント行と空白行を除く）にモジュール名を含めます。次のモジュール・セグメントは、モジュール名として MSDB_MOD を割り当てます。

```
MODULE          MSDB_MOD          -- Module name
```

モジュール名を省略する場合、SQL はデフォルトで名前 SQL_MODULE を使用します。モジュール名が必要でなくても、MODULE キーワードは必要です。

複数のプログラム・モジュールを結合して 1 つの実行可能イメージまたは共有イメージにするつもりならば、ストアード・モジュールを含むすべてのモジュールの名前は一意である必要

があります。たとえば、SQL モジュールは、今後リンクする場合、ホスト言語モジュールまたは他の SQL モジュールと同じ名前を付けることはできません。モジュール名が一意でなければ、イメージ内にオブジェクト・ファイルをリンクする場合にあいまいなグローバル・シンボルによるエラーが発生します。

モジュールの実ファイル名とモジュール・ヘッダー内のモジュール名には、まったく関連がありません。

3.2.3 表現形式 (dialect) の指定方法

DIALECT 句は、次の各項目を含むセッションに対する多くの設定を制御します。

- 文字列リテラルの長さ、列およびドメインが文字として解釈されるか、オクテットとして解釈されるか
- 二重引用符 (") が文字リテラルとして解釈されるか、区切り文字として解釈されるか
- 識別子がキーワードになれるかどうか
- どちらの表示が読取り専用であるか
- DATE データ型または CURRENT_TIMESTAMP データ型の列は、VMS フォーマットまたは ANSI/ISO フォーマットで解釈されるか
- パラメータ名はコロンで始まるかどうか
- セッション文字セットは指定した表現形式に従って変更されるかどうか

次の例は、表現形式に SQL92 を設定する方法を示しています。

```
DIALECT    SQL92                -- Set the dialect.
```

DIALECT 句を指定すると、個々の設定を行うかわりに、1 つの句でモジュールに対する設定を指定します。表現形式設定には、SQL92、SQL89、MIA および SQLV40 があります。SQLV40 設定は、デフォルトです。前の表現形式との互換性を維持する必要がなければ、表現形式に SQL92 または MIA を設定します。各表現形式に対する特別な設定の詳細は、『Oracle Rdb7 SQL Reference Manual』の SET DIALECT 文の説明を参照してください。

モジュール・プロセッサは、モジュール句を順次処理するので、DIALECT 句は、この前に指定した句の設定を上書きでき、この後に SQL モジュール・ファイル内の後ろで指定した句によって上書きされます。

3.2.4 セッションに対するキャラクタ・セットの指定方法

SQL セッションまたはモジュールに対して次のキャラクタ・セットを指定できます。

- デフォルトのキャラクタ・セット
デフォルトのキャラクタ・セットは、キャラクタ・セットまたは各国語キャラクタ・

セットによって指定されないキャラクタ・データ型であるデータベース列のキャラクタ・セットを決定します。

- 識別子キャラクタ・セット
識別子キャラクタ・セットは、ユーザーが供給したデータベース・オブジェクト名、たとえばテーブル名、ドメイン名および列名に対するキャラクタ・セットを決定します。キャラクタ・セットには、ASCII が含まれる必要があります。
- 各国語キャラクタ・セット
各国語キャラクタ・セットは、データ型 NCHAR または NCHAR VARYING のすべての列および各国語キャラクタ・セットによって制限された文字列リテラルに対するキャラクタ・セットを決定します。
- リテラル・キャラクタ・セット
リテラル・キャラクタ・セットは、キャラクタ・セットまたは各国語キャラクタ・セットによって制限されないリテラルに対するキャラクタ・セットを決定します。

『Oracle Rdb7 SQL Reference Manual』には、デフォルト・キャラクタ・セット、識別子キャラクタ・セット、各国語キャラクタ・セットおよびリテラル・キャラクタ・セットに対して使用できるキャラクタ・セットをリストしているテーブルのセットがあります。次の例では、デフォルトのキャラクタ・セットを KANJI に設定しています。

```
DEFAULT CHARACTER SET      KANJI      -- Set the default character set
```

SQL は、SET コマンドを使用してこれらのキャラクタ・セットを指定する場合やセッションの表現形式を MIA に設定していない場合は、デフォルト、識別子、各国語およびリテラル・キャラクタ・セットに対して、DEC_MCS を使用します。しかし、表現形式に MIA を設定した後で変更する場合、キャラクタ・セットは変更しません。新しい表現形式に関連した規則のみが変更されます。

3.2.5 モジュール・プロシージャをコールするホスト言語を識別

モジュールに名前を付けた後、LANGUAGE 句と ADA、BASIC、C、COBOL、FORTRAN、PASCAL、PLI (PL/I 用) または GENERAL などのキーワードの 1 つを指定して、ホスト言語プログラムのコールに使用される言語を識別する必要があります。たとえば、次のように示します。

```
LANGUAGE      C      -- Language of calling program
```

言語識別子は、後でプロシージャ・パラメータを宣言するときに SQL モジュール・プロセッサが有効であると考えるデータ型の種類と、SQL が実行時にパラメータ値を渡すために使用するコール・モジュールを予測するメカニズムの両方を決定します。

LANGUAGE 句内に対応するキーワードを持たない言語に GENERAL キーワードを指定します。

3.2.6 カタログの指定

CATALOG オプションの句を使用すると、モジュール・ヘッダー内にカタログ名を指定できます。SQL は、マルチスキーマ・データベース内でスキーマ・オブジェクトを指定し、カタログ名を明示的に含んでいない場合、デフォルトでこの名前を使用します。モジュール・ファイル内で CATALOG 句を省略する場合、SQL は、デフォルトとして RDB\$CATALOG を使用します。

デフォルトのカタログ名を指定して、RDB\$CATALOG のデフォルトを上書きすると、カタログ名を含めずにそのカタログ内でスキーマとスキーマ・オブジェクトを指定することができます。カタログ名を省略すると、毎回名前を入力して、モジュール内のオブジェクトを指定する作業を節約できます。

SQL モジュール内で識別しようとしているすべてのスキーマ・オブジェクトに ADMINISTRATION カタログ名を指定しない場合は、次に示す CATALOG 句を含めて、モジュールのデフォルトのカタログを変更します。

```
CATALOG ADMINISTRATION
```

次に示す一意なマルチスキーマ名を入力して、PERSONNEL スキーマ内の EMPLOYEES テーブルなどのスキーマ・オブジェクトを指定することができます。

```
SELECT * FROM PERSONNEL.EMPLOYEES;
```

SCHEMA 句を使用してデフォルトのスキーマを変更する場合は、3.2.7 項 (3-9) で示したように、スキーマ名を省略することもできます。

同じ名前であるオブジェクトを持つスキーマを含むマルチスキーマ・データベースがあれば、明示的にカタログやスキーマ名を SQL 文内で指定しなくても様々なスキーマで使用するために SQL モジュールを複製できますが、そのかわりモジュール・ヘッダー内のデフォルト・カタログとスキーマ名を変更します。

CATALOG 句は、オプションのマルチスキーマ属性を有効にしたデータベースを使用するこれらのモジュール・プロシージャのみに影響を与えます。

3.2.7 スキーマの指定

SCHEMA オプションの句を使用すると、モジュール・ヘッダー内にスキーマ名を指定できます。SQL は、マルチスキーマ・データベース内でスキーマ・オブジェクトの名前を付ける場合、スキーマ名を明示的に含んでいなければ、デフォルトでこの名前を使用します。

SCHEMA 句で明示的にスキーマを命名しなければ、SQL は、デフォルトのスキーマとしてモジュールをコンパイルするユーザーの名前である明示的な認可識別子を使用します。認可識別子は、AUTHORIZATION 句で明示的に名づけられていれば、SQL はデフォルトのスキーマとしてその名前を使用します。

SCHEMA 句内でスキーマ名を指定すると、スキーマのデフォルトを上書きし、スキーマ名を含めることなくスキーマとカタログ内でスキーマ・オブジェクトを指定できるようになり

ます。スキーマ名を省略すると、モジュール内でスキーマ・オブジェクトの名前を完全に入力するタスクを節約できます。

SQL モジュール内で識別しようとしているスキーマ・オブジェクトに対する ADMINISTRATION カタログ名または PERSONNEL スキーマを指定しない場合は、次に示すモジュール句を含めて、モジュールのデフォルトのカタログとスキーマを変更します。

```
CATALOG ADMINISTRATION  
SCHEMA PERSONNEL
```

そして、テーブル名を入力するだけで、ADMINISTRATION カタログの PERSONNEL スキーマ内の EMPLOYEES テーブルなどのスキーマ・オブジェクトを指定することができます。

```
SELECT * FROM EMPLOYEES;
```

SCHEMA 句は、マルチスキーマ・データベースを使用するこれらのモジュール・プロシージャのみに影響を与えます。

3.2.8 認可識別子（Authorization Identifier）の指定

オプションの AUTHORIZATION 句は、モジュールに対する認可識別子を指定します。認可識別子を省略すると、SQL はデフォルトの認可としてモジュールをコンパイルするユーザーのユーザー名を選択します。したがって、モジュール・ヘッダーで RIGHTS 句を使用すると、SQL はモジュールを実行するユーザー名とモジュールをコンパイルしたときの認可識別子を比較し、モジュールをコンパイルしたユーザー以外のユーザーによる実行を防止します。RIGHTS 句を使用する場合、SQL は、デフォルトの認可識別子（ANSI/ISO 標準に準拠）上でチェックした権限に基づきます。デフォルトでは、SQL は認可識別子を使った権限のチェックを行わず、実行者の権限を使用します。

次の例では、認可識別子を設定します。

```
AUTHORIZATION    SAMPLE_USER          -- Authorization ID
```

SCHEMA 句または ALIAS 句内のエイリアスで明示的にスキーマに名前を付けない場合、モジュールに対する認可識別子がデフォルトのスキーマおよびデフォルトのエイリアスになります。そして、この場合デフォルトのスキーマ、デフォルトの認可識別子およびデフォルトのエイリアスは同じ名前です。

3.2.9 エイリアスの指定

オプションの ALIAS 句は、モジュールに対してデフォルトのデータベース・エイリアスを指定します。モジュール・ヘッダーでエイリアスを省略すると、SQL は AUTHORIZATION 句で明示的に指定された認可識別子をデフォルトのエイリアスとして使用します。モジュールのヘッダーで認可識別子を明示的に指定しない場合、SQL は、RDB\$DBHANDLE をデフォルトのエイリアスとして使用します。モジュール内で2つ以上のデータベースを参照する必要がある場合、ALIAS 句を使用します。

次の例は、デフォルトのエイリアスを指定します。

```
ALIAS                      RDB$DBHANDLE          -- Default alias
```

Oracle Rdb は、アプリケーションが複数のモジュールに渡って唯一のデータベースを参照する必要がある場合、リンクして実行可能イメージを作成するすべてのモジュール内でデフォルトのデータベースに対して同じエイリアス名を使用するように推奨します。

イメージが SQL プリコンパイラで処理されるモジュールを含んでいたら、イメージ内のすべての SQL モジュールの AUTHORIZATION 句内で RDB\$DBHANDLE を指定する必要があります。エイリアス RDB\$DBHANDLE は、常にプリコンパイル SQL プログラム内のデフォルト・データベースとして指定されます。

3.2.10 パラメータにコロンを付ける設定

次の例に示すように、SQL モジュール・ヘッダーで PARAMETER COLONS 句を使用すると、モジュール・プロシージャ内のパラメータの前にコロン (:) をつけるように指定できます。

```
PARAMETER COLONS          -- Parameters are prefixed by colons
```

現在、SQL モジュール・プロセッサのデフォルトでは、コロンは使用しないよう設定されています。しかし、コロンの使用は ANSI/ISO SQL 標準で必要であるため、Oracle Rdb は PARAMETER COLONS 句を指定して、コロンを付けてパラメータを使用するように推奨します。表現形式に SQL92 を指定すると、PARAMETER COLONS 句を指定していない場合でも、SQL は、コロンを使用するよう強制します。

3.2.11 モジュール内での DECLARE 文の指定

次に示すように SQL モジュール・ヘッダーで DECLARE 文を指定できます。

- DECLARE ALIAS 文
この文は、モジュールが使用するデータベースの名前とソースを指定します。1つのデータベースで作業していて、そのデータベースへ接続するときにはリポジトリ・アクセスが必要でなければ、モジュールから DECLARE ALIAS 文を省略して、論理名 SQL\$DATABASE または構成パラメータ SQL_DATABASE にファイル仕様を割当てることができます。そうでない場合は、アクセスする予定の各データベース用のモジュールに DECLARE ALIAS 文を含めます。
データベース接続オプションの詳細は、15.1 項 (15-1) を参照してください。
- DECLARE CURSOR 文または動的 DECLARE CURSOR 文
この文は、カーソルを宣言します。モジュール・プロシージャ内の文で参照する各カーソルに対して、DECLARE CURSOR 文を含める必要があります。カーソルを宣言して使用する文の詳細は、18 章 (18-1) を参照してください。

- DECLARE STATEMENT 文
この文は、動的 SQL で後ほど使用する文の名前を説明します。動的 SQL の詳細は、11 章 (11-1) を参照してください。
- DECLARE TABLE 文
この文は、ソース・ファイル内のテーブルと表示定義を指定します。実行時にプログラムがアクセスするテーブルとビューに対する定義が SQL ソース・ファイルに含まれている場合、SQL モジュール・プロセッサは、コンパイル時にデータベースに接続したり、リポジトリにアクセスする必要はありません。まだ存在していないテーブルまたはビューを参照する SQL モジュールをコンパイルしたい場合や、コンパイル時にデータベースの接続に関連するリソースのオーバーヘッドを取り除きたい場合、DECLARE TABLE 文は役立ちます。
- DECLARE TRANSACTION 文
この文は、プログラムによって暗黙的に開始されるすべてのトランザクションに対するデフォルトの特性を設定します。トランザクション用の SQL デフォルトが必要であるか、SET TRANSACTION 文に対するプロシージャ・コールによってトランザクションが明示的に開始されるように計画していれば、DECLARE TRANSACTION 文を指定する必要はありません。トランザクションについての詳細情報は、16 章 (16-1) を参照してください。

SQL モジュールでは、DECLARE 文の最後にセミコロン (;) を使用しないでください。セミコロンは、モジュール・プロシージャ内でのみ有効です。

3.3 ホスト言語プログラムからの SQL モジュール・プロシージャのコール

ホスト言語プログラムから SQL モジュール・プロシージャをコールするフォーマットは、ホスト言語に固有です。個々のフォーマットについては、ご使用のホスト言語のドキュメントを参照してください。ホスト言語のドキュメントには、変数受渡しメカニズムのデフォルトを変更する方法についても詳しく説明されています。

例 3-2 (3-12) に、C ホスト言語プログラム sql_intro_load_empl.h.c の一部を示します。このプログラムでは、例 3-1 (3-4) に示す SQL プロシージャ sql_intro_load_empl.c.sqlmod をコールします。

例 3-2 SQL モジュールをコールするホスト言語

```
.
.
.
/*
This structure represents the record to be inserted into the database.
Character strings are one character longer to hold the null value that
terminates the string. */
```

```

struct Employee_struct {
    char employee_id[6];
    char last_name[15];
    char first_name[11];
    char middle_initial[2];
    char address_data[26];
    char city[21];
    char state[3];
    char postal_code[6];
    char ascii_birthdate[11];
    char sex[2];
};

/* This structure represents the format of each record in the stream file. */
struct Employee_struct_buf {
    char employee_id[5];
    char last_name[14];
    char first_name[10];
    char middle_initial[1];
    char address_data[25];
    char city[20];
    char state[2];
    char postal_code[5];
    char ascii_birthdate[10];
    char sex[1];
    char linefeed[1];
};

/* Return status and SQLCODE variables for error handling. */
int return_status;
long sqlcode;
/* Function prototypes */
void SET_TRANS (long *sqlcode);
void INSERT_DATA (long *sqlcode, struct Employee_struct *employee_rec_ptr);
void COMMIT_TRANS (long *sqlcode);
.
.
.
/* Start the transaction by calling the SQL module language procedure
   SET_TRANS in sql_intro_empl_h.sqlmod. If the call to SET_TRANS returns
   a value other than 0, use sql_signal to display the error. */
    SET_TRANS(&sqlcode);
    if (sqlcode != 0)
    {
        printf("\nStart transaction failed.\n");
        sql_signal();
    }
.
.
.

```

```
/* Invoke the INSERT_DATA procedure to insert a row in the EMPLOYEES table.
   The date is converted from text to binary by the CAST function in the SQL
   procedure. If the call to INSERT_DATA returns a value other than 0,
   use sql_signal to display the error. */
   INSERT_DATA (&sqlcode, &employee_rec);
   if (sqlcode != 0) sql_signal();
}
/* Commit the transaction by calling the SQL module language procedure.
   COMMIT_TRANS. */
   COMMIT_TRANS (&sqlcode);
.
.
.
```

3.4 SQL モジュール・プロセッサによる移植可能アプリケーションの記述

1つのSQL実装から別の実装に移行しても、編集がほとんどまたは全く不要なアプリケーションを作成することが必要になる場合があります。SQLモジュール言語は、SQLのANSI/ISO標準に包含されているため、標準に準拠したプログラムを記述できます。SQLモジュール内でANSI/ISO標準に対するOracle Rdb拡張を明示するには、モジュール・ファイルのコンパイル時にFLAG_NONSTANDARDまたは-stdコマンド行修飾子を使用します。

多くのベンダーによるSQLの実装では、モジュール言語サポートが実装されません。個々の実装で提供されるモジュール言語サポートのレベルを確認してから、その実装に移植するSQLモジュール言語プロシージャを記述してください。

3.5 SQL モジュール・プロセッサの詳細情報の参照先

SQLモジュール・プロセッサの使用方法については、次の詳細情報を参照してください。

- 第4章 (4-1) では、単一SQL文と複合文を含む複数文のプロシージャなど、SQLモジュール・プロシージャを記述する方法を示します。
- 第5章 (5-1) では、SQLモジュールとホスト言語プログラムをコンパイルしてリンクする方法と、コンテキスト・ファイルを使用して移植性を改善する方法を示します。
- 第7章 (7-1) では、SQLとホスト言語のオブジェクト・モジュールをリンクして実行可能イメージを作成する方法を示します。
- サンプル・ディレクトリには、SQLモジュールとそれをコールするホスト言語プログラムのオンライン・サンプルがあります。サンプル・プログラムの詳細は、サンプル・ディレクトリのオンライン・ファイルabout_sql_examples.txtを参照してください。
- 『Oracle Rdb7 SQL Reference Manual』には、SQLモジュール言語の参照情報があります。

SQL モジュール・プロシージャの記述方法

この章では、SQL モジュール・プロシージャの記述方法を説明します。SQL モジュール言語の概要および SQL モジュールとモジュール・ヘッダー情報の作成方法の説明は、3 章 (3-1) を参照してください。この項では、次の項目を説明します。

- SQL モジュール・プロシージャのコンポーネントと各種類のプロシージャ
- プロシージャ名やパラメータおよびそのデータ型など、SQL モジュール・プロシージャの共通コンポーネントの指定方法
- プロシージャ内での単一 SQL 文の指定方法
- プロシージャ内での複合文の指定方法
- SQL モジュール言語の制限

4.1 SQL モジュール・プロシージャについて

SQL モジュール内のプロシージャは、ホスト言語モジュールからのコールを通して一連の SQL 文を実行するメカニズムです。

SQL モジュール・プロシージャは、次のものを含む一連の共通プロシージャ要素から構成されています。

- プロシージャを導入するキーワード PROCEDURE
- プロシージャ名
- エラー処理用のステータス・パラメータ
- 1 つ以上のパラメータ宣言
- 1 つの SQL 文、または複合文内の 1 つ以上の SQL 文

4.2 項 (4-3) では、共通プロシージャ要素について説明します。

2 つのタイプのプロシージャを定義できます。

- **単純文プロシージャ**

単純文プロシージャは、共通プロシージャ要素と単一の実行 SQL 文のみで構成されています。次の単純文プロシージャは、SET_TRANS プロシージャを実行すると、読取り / 書込みトランザクションを開始します。

```
PROCEDURE SET_TRANS
  (SQLSTATE);
  SET TRANSACTION READ WRITE;
```

4.3 項 (4-14) では、単純文プロシージャを記述する方法を説明します。

- **複数文プロシージャ**

複数文プロシージャは、共通プロシージャ要素と 1 つ以上の SQL 文を含む複合文から構成されています。複合文内では、SQL IF および FOR などのフロー制御文を使用できます。

次の複数文プロシージャには、2 つの INSERT 文が含まれています。

```
PROCEDURE INSERT_DATA
-- Declare the parameters by which the values are passed between the SQL
-- module and the host language.
  (SQLSTATE,
   :P_EMPLOYEE_ID    CHAR(5),
   :P_LAST_NAME      CHAR(14),
   :P_FIRST_NAME     CHAR(10),
   :P_JOB_CODE       CHAR(4),
   :P_DEPARTMENT_CODE CHAR(4));
BEGIN
  INSERT INTO EMPLOYEES
```

```

        (EMPLOYEE_ID, LAST_NAME, FIRST_NAME)
VALUES
    (:P_EMPLOYEE_ID, :P_LAST_NAME, :P_FIRST_NAME);
INSERT INTO JOB_HISTORY
    (EMPLOYEE_ID, JOB_CODE, JOB_START, DEPARTMENT_CODE)
VALUES
    (:P_EMPLOYEE_ID, :P_JOB_CODE, CURRENT_TIMESTAMP, :P_DEPARTMENT_CODE);
END;
```

4.4 項 (4-15) では、複数文プロシージャの記述方法を説明し、12 章 (12-1) では、複合文の記述に関するチュートリアル情報を提供します。

4.2 SQL モジュール・プロシージャの共通要素の指定方法

単純文プロシージャ、複数文プロシージャのどちらを使用する場合でも、プロシージャの定義は次のようになります。

- PROCEDURE という用語で開始
- プロシージャ名を指定
- SQL モジュールおよびコール側ホスト言語モジュールの間で値を交換するためのパラメータを宣言
- パラメータが記述子によって渡されることを指定（任意指定）
- コール側モジュールが記述子によってパラメータを受け渡さない場合に、SQL が実行時エラーを返すことを指定（BY DESCRIPTOR 句を含むパラメータ宣言の場合は任意指定）

4.2.1 プロシージャの命名

SQL モジュール・プロシージャは、PROCEDURE キーワードで開始する必要があり、プロシージャの名前がそれに続きます。実行可能イメージ内のプロシージャ名は、一意でなければなりません。これらが SQL モジュール内で一意でない場合は、SQL モジュール・プロセッサがコンパイル時エラーを発行します。

さらに、複数の SQL モジュールを結合して 1 つの実行可能イメージまたは共有イメージにする場合は、プロシージャの名前がこのイメージ内で一意である必要があります。この場合、プロシージャ名が一意でなければ、イメージ内にオブジェクト・ファイルをリンクする場合に、あいまいなグローバル・シンボルによるエラーが発生します。

システム・プロシージャやアプリケーションが使用するすべてのライブラリ内のプロシージャを含めて、アプリケーションにリンクする他のプロシージャが使用する名前を選択しないでください。

Compaq Tru64 UNIX

CompaqTru64 UNIX 環境では大文字と小文字を区別し、さらに CompaqTru64 UNIX 上では SQL モジュール・プロセッサがプロシージャ名をデフォルトで大文字に変換するので、モジュール・プロシージャ名の使用には注意が必要です。

たとえば、SQL モジュール言語プロシージャへのコールを C プログラムから記述する場合、小文字を使用してこれらのプロシージャを起動すると、オブジェクト・ファイルをリンクするときにエラーが発生します。SQL は、名前を大文字に変換しますが、C 言語は小文字のまま保持します。

この問題を避けるには、次のいずれかの方法を使用します。

- ホスト言語から大文字の名前を使用して SQL プロシージャを起動
- `-lc_proc` コマンドライン修飾子を使用して、モジュール言語プロシージャの名前を小文字に変換 ◆

4.2.2 プロシージャ・パラメータの宣言

プロシージャ内の SQL 文の実行時に SQL モジュールやホスト言語モジュールが交換するすべての値に対して、パラメータを宣言します。パラメータ宣言は、次の種類の値に対して必要です。

- プロシージャの実行ステータスを示す値
各プロシージャで、次に示すステータス・パラメータの 1 つを指定する必要があります。
 - `SQLCODE`
 - `SQLCA`
 - `SQLSTATE`

Oracle Rdb は、新しいアプリケーションに対して `SQLSTATE` ステータス・パラメータを使用するように推奨します。`SQLSTATE` は、`SQLCODE` よりも標準的なステータス・コードをサポートしており、これは ANSI/ISO SQL 標準により適合するメカニズムです。`SQLCA` ステータス・パラメータは、ANSI/ISO SQL 標準の拡張です。

ステータス・パラメータを宣言する場合、データ型は暗黙的であり指定できません。

これらのパラメータの説明は、10 章 (10-1) を参照してください。

- テーブル検索のベースとなる任意の値
これらのパラメータは、プロシージャの実行可能文で指定するほとんどの `WHERE` 句や `HAVING` 句の評価に必要です。また、カーソルの宣言でも必要です。
- プロシージャで検索され、保存される任意の列値
列値に対するパラメータのデータ型は、4.2.5 項 (4-10) で説明します。
- 列内の `NULL` 値を処理するインジケータ値
列値と `NULL` の処理についての詳細は、8.10 項 (8-18) を参照してください。

- キーワード SQLDA
SQLDA (SQL 記述子領域) は、動的 SQL 文の情報を提供するデータ構造です。選択リスト・パラメータとパラメータ・マーカが実行時までわからない動的 SQL 文で使用するために、ホスト言語モジュールで SQLDA を宣言する必要があります。SQLDA の詳細は、11 章 (11-1) を参照してください。
- キーワード SQLDA2
SQLDA2 (SQL 記述子領域 2) は、SQLDA 構造体の拡張版です。SQLDA2 データ構造体には、動的 SQL 文についての追加フィールドとフィールド・サイズ情報があります。次のいずれかがパラメータ・マーカまたは選択リスト項目に適用されている場合、SQLDA のかわりに SQLDA2 を宣言する必要があります。
 - 列名の長さが 30 オクテット (8 ビット) を超えている
 - 列のデータ型が日付時刻データ型である
 - データ型が CHAR、CHAR VARYING、CHARACTER、CHARACTER VARYING、VARCHAR または LONG VARCHAR であり、キャラクタ・セットがデフォルトの 8 ビットではないか、オクテットの最長が 65,535 を超えている、またはこの両方が真である
 選択リスト・パラメータとパラメータ・マーカが実行時までわからない動的 SQL 文で使用するホスト言語モジュールで、SQLDA2 を宣言します。
SQLDA2 の詳細は、11 章 (11-1) を参照してください。

SQL が提供するパラメータ (SQLSTATE、SQLCODE、SQLCA、SQLDA および SQLDA2) を宣言する場合、データ型を宣言しませんが、ユーザー定義パラメータを宣言する場合は、パラメータ用のデータ型を宣言する必要があります。

SQL に対する ANSI/ISO 標準に準拠するため、また読みやすさを改善し、あいまいさを避けるため、Oracle Rdb は、各パラメータの前にコロン (:) を付け、各パラメータをカンマ (,) で区切り、すべてのパラメータをカッコで括り、さらにパラメータ宣言のリストを 1 個のセミコロン (;) で終了するよう推奨しています。

4.2.3 異なる種類のプロシージャに対して必要なパラメータ

すべてのモジュール・プロシージャで、特定種類のパラメータを宣言する必要があります。プロシージャが実行する SQL 文に従って宣言するパラメータは、次のようになります。

- SQL モジュールにデータベース仕様に対するプログラム・パラメータを含む DECLARE ALIAS 文が含まれるならば、SQL モジュール内の各プロシージャでそのパラメータを宣言する必要があります。さらに、SQL モジュール内のそのプロシージャに対する各ホスト言語コールでパラメータを指定する必要があります。DECLARE ALIAS 文には、ホスト言語モジュールが実行時にデータベースの位置を識別するためにファイル名またはリポジトリ・パス名値を受け入れる場合は、プログラム・パラメータを持ちます。

例 4-1 (4-6) は、sql_insert_degrees_module.sqlmod モジュール・ファイルを変更して、ホスト言語モジュール、またはユーザーが、実行時にデータベースのファイル仕様を入力できる方法を示しています。

例 4-1 実行時にデータベース名を受け入れる SQL モジュールの記述方法

```
-- This SQL module accepts the database name at run time.
-----
-- Header Information Section
-----
MODULE          sql_insert_degrees_mod_param -- Module name
LANGUAGE        COBOL                      -- Language of calling program
AUTHORIZATION   RDB$DBHANDLE                -- Authorization identifier
PARAMETER COLONS                      -- Use colons
-----
-- DECLARE statement section
-----
DECLARE ALIAS FOR COMPILETIME FILENAME personnel
                        RUNTIME FILENAME :DB_NAME
-----
-- PROCEDURE SECTION
-----
PROCEDURE INSERT_DEGREES
    (:DB_NAME          CHAR(80),
     SQLCODE,
     :P_EMPLOYEE_ID    CHAR(5),
     :P_COLLEGE_CODE   CHAR(4),
     :P_YEAR_GIVEN     SMALLINT,
     :P_DEGREE         CHAR(3),
     :P_DEGREE_FIELD   CHAR(15));
INSERT INTO DEGREES
    (EMPLOYEE_ID, COLLEGE_CODE, YEAR_GIVEN,
     DEGREE, DEGREE_FIELD)
VALUES
    (:P_EMPLOYEE_ID, :P_COLLEGE_CODE, :P_YEAR_GIVEN,
     :P_DEGREE, :P_DEGREE_FIELD);
PROCEDURE START_TRANS
    (:DB_NAME CHAR(80),
     SQLCODE);
SET TRANSACTION READ WRITE;
PROCEDURE COMMIT
    (:DB_NAME CHAR(80),
     SQLCODE);
COMMIT;
```

ホスト言語プログラムを変更して、データベース名を SQL モジュールのすべてのプロシージャに渡す必要があります。例 4-2 (4-7) は、データベース名を SQL モジュール内

のすべてのプロシージャに渡すように変更した `sql_insert_degrees_program.cob` プログラムからの引用を示します。

例 4-2 SQL モジュールへのデータベース名の引渡し

```

.
.
.
* Declare the database name.
01   P_DB_NAME          PIC X(80) .
.
.
.
GET_DATABASE.
    DISPLAY " Enter the database name".
    ACCEPT P_DB_NAME.
.
.
.
    CALL "START_TRANS" USING P_DB_NAME, SQLCODE.
    CALL "INSERT_DEGREES" USING P_DB_NAME, SQLCODE, P_EMPLOYEE_ID,
        P_COLLEGE_CODE, P_YEAR_GIVEN, P_DEGREE, P_DEGREE_FIELD.
    CALL "COMMIT" USING P_DB_NAME, SQLCODE.

```

- データ定義、COMMIT 文および ROLLBACK 文では、すべてのプロシージャに必要なこれらのパラメータのみを宣言する必要があります。DECLARE ALIAS 文にパラメータが含まれていなければ、次の例で示すように SQLCODE、SQLCA または SQLSTATE のみを指定する必要があります。

```

PROCEDURE COMMIT_TRANSACTION
    (SQLCODE);
COMMIT;

```

- DECLARE CURSOR 文がパラメータを含んでいれば、OPEN 文を含むプロシージャでそれらを宣言することでそれにパラメータを渡します。さらに、OPEN 文を含むプロシージャに対するホスト言語コールでパラメータを指定する必要があります。DECLARE CURSOR 文がプロシージャではなく、SQL モジュールの宣言セクションに現れるので、パラメータを直接 DECLARE CURSOR 文に渡すことはできません。カーソルでパラメータを使用する例として例 4-3 (4-7) を参照してください。
- OPEN 文では、開いているカーソルに対応する DECLARE CURSOR 文に含まれるパラメータに加えて、すべてのプロシージャに必要なすべてのパラメータを宣言します。例 4-3 (4-7) は、カーソル内でパラメータを宣言して、使用方法を示します。

例 4-3 カーソルでパラメータを使用する方法

```

DECLARE ID_AND_NAME CURSOR FOR
    SELECT EMPLOYEE_ID, FIRST_NAME, MIDDLE_INITIAL, LAST_NAME

```

```
FROM EMPLOYEES
WHERE EMPLOYEE_ID = :INPUT_ID
PROCEDURE OPEN_ID_AND_NAME
(SQLCODE,
:INPUT_ID CHAR(5));
OPEN ID_AND_NAME;
```

例 4-3 (4-7) は、WHERE 句にパラメータがある場合のみを示しています。DECLARE CURSOR 文のその他の句にパラメータを含めることはできますが、そうすると、カーソルを開くプロシージャでもこれらのパラメータを宣言する必要があります。SQL モジュールの各 DECLARE CURSOR 文に対して、1 つの OPEN 文プロシージャのみを指定できます。

- FETCH 文の場合、すべてのプロシージャに対して必要なパラメータに加えて INTO 句に対して指定されたすべてのパラメータを宣言します。たとえば、次のように示します。

```
PROCEDURE FETCH_ID_AND_NAME
(SQLCODE,
:ID_VAR CHAR(5),
:FIRST_NAME_VAR CHAR(10),
:MIDDLE_INIT_VAR CHAR(1),
:MIDDLE_INIT_IND SMALLINT),
:LAST_NAME_VAR CHAR(14);
FETCH EMP_ROW_CURSOR INTO
:ID_VAR,
:FIRST_NAME_VAR,
:MIDDLE_INIT_VAR :MIDDLE_INIT_IND,
:LAST_NAME_VAR;
```

- データ操作文 INSERT、UPDATE または DELETE の場合、すべてのプロシージャに対して必要なパラメータを宣言する必要があります。さらに、列値またはインジケータ値を保管するパラメータを宣言するか、値の比較に使用するパラメータを宣言します。たとえば、次のように示します。

```
PROCEDURE UPDATE_JH
(SQLSTATE,
:JOB_END_DATE_BIN DATE,
:INPUT_EMP_ID CHAR(5));
UPDATE JOB_HISTORY
SET JOB_END = :JOB_END_DATE_BIN
WHERE EMPLOYEE_ID = :INPUT_EMP_ID;
PROCEDURE DELETE_JH
(:INPUT_ID CHAR(5),
:INPUT_DB_FILE CHAR(20),
SQLSTATE);
DELETE FROM JOB_HISTORY
WHERE EMPLOYEE_ID = :INPUT_ID;
```

サンプル・ディレクトリ内の `sql_all_datatypes_ada.sqlmod` モジュールは、`FETCH` 文および `UPDATE` 文に対するパラメータの宣言方法を示します。

4.2.4 プロシージャ・パラメータと実パラメータの関連

各プロシージャ・パラメータ、すなわち SQL モジュール・プロシージャ内の仮パラメータは、ホスト言語モジュールで宣言し、SQL モジュール内のプロシージャへのホスト言語コールで指定する**実パラメータ**に対応します。ホスト言語によっては、SQL モジュール内のパラメータは、外部プロシージャ、ルーチンまたは関数の宣言で識別された仮パラメータにも対応します。

ホスト言語モジュールにおける SQL モジュールのパラメータと実パラメータの間の対応は、仕様の順序によって決まります。SQL モジュール・プロシージャで宣言された最初のパラメータは、ホスト言語コールで指定された最初のパラメータに対応し、SQL モジュール・プロシージャで宣言された 2 番目のパラメータは、ホスト言語コールで指定された 2 番目のパラメータに対応し、以下同様に続きます。

図 4-1 (4-9) は、SQL モジュール・プロシージャ・パラメータ、実パラメータ、ホスト言語宣言およびデータベースの間の対応を説明しています。この図では、コール側モジュールで使用する言語は C ですが、対応する原理はすべての言語で同様です。

図 4-1 実パラメータ、プロシージャ・パラメータおよび列の対応

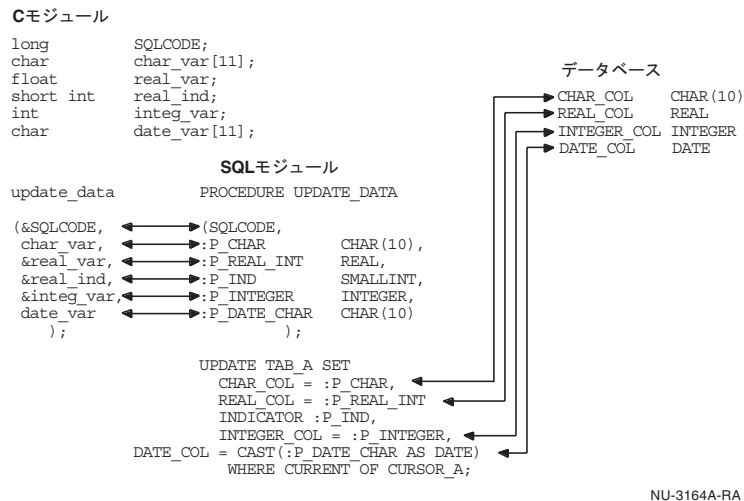


図 4-1 (4-9) では、`char_var` および `date_var` は、それらの対応するプロシージャ・パラメータよりも 1 文字長く宣言されていることに注意してください。このサイズの違いは、C 言語を使用する場合のみです（詳細は、4.5 項 (4-16) を参照してください）。

一貫した順序は、SQL モジュールのプロシージャ・パラメータとホスト言語モジュールの実パラメータの間での対応のみで重要です。(図 4-1 (4-9) では、列定義および列参照の並列仕様は、対応する行を適切に保つためのみに存在します。)

さらに、対応する実パラメータおよびプロシージャ・パラメータの名前が一致していることや、1つの種類のパラメータ(たとえば、列)に対して他の種類のパラメータ(たとえば、インジケータ)に関連した順序を指定することは重要ではありません。しかし、対応するプロシージャに対して同じ名前を使用する場合、また1つのプロシージャから次のプロシージャに対して一貫した順序方式を使用する場合には、パラメータ対応を誤る可能性が少なくなります。

プロシージャ・パラメータに指定した順序と異なる順序で実パラメータを指定する場合(またはプロシージャと比較して異なる数の実パラメータを指定した場合)、デバッグが困難な実行時エラーが発生します。BY DESCRIPTOR CHECK 句は、この種のエラーの検出に役立ちます。BY DESCRIPTOR CHECK 句の詳細は、4.2.8 項 (4-13) を参照してください。

4.2.5 パラメータ・データ型の指定

SQL モジュールのプロシージャ・パラメータ用に、いくつかのデータ型を選択できます。これらのデータ型の詳細な説明は、『Oracle Rdb7 SQL Reference Manual』の、言語と構文要素に関する章のデータ型の項を参照してください。データ型キーワードのかわりに、ドメイン定義の名前を指定することもできます。

次の引用は、3つのパラメータの宣言方法を示します。

```
(SQLSTATE,                -- Status parameter.  No data type specified.
:LAST_NAME_VAR  CHAR(14),  -- Character data type.
:EMP_ID        ID_DOM)    -- Data type based on the domain ID_DOM.
```

パラメータ・データ型を決定する場合、次のすべてに対応するデータ型を考慮する必要があります。

- 列の定義
- SQL モジュールにおけるパラメータ宣言
- ホスト言語モジュールにおけるパラメータ宣言

ほとんどの場合、データベースとプログラムの間で、データ記憶域フォーマットの一貫性を維持することは可能です。しかし、次の条件のどちらかが真であれば、列の定義で指定したデータ型とは異なるデータ型になるように SQL モジュールとホスト言語モジュールでパラメータを宣言する必要があります。

- コール側モジュールを記述した言語がテーブル列定義で指定したデータ記憶域フォーマットをサポートしていない場合で、さらに、プログラムがその列に保存された値で動作する必要がある場合

- ユーザー・プログラムがファイルからデータベースに書き込み（またはデータベースからファイルに書き込み）、列値に関連するレコード・フィールドが列とは異なる記憶域フォーマットを持つ場合

通常、いずれの問題も SQL に、データ記憶域フォーマットの変換を指示することで解決できます。文字データ型と Oracle Rdb データ型 DATE との間の変換についての方式は、図 4-1 (4-9) で説明します。この図は、date_var および P_DATE_CHAR の宣言と DATE_COL の列定義についても示しています。

そのかわり、ホスト・プログラム言語は、データ記憶域フォーマット間での変換をサポートしている可能性があります。データ型の変換に対してホスト言語で利用可能な機能には何があるかを判断するため、ホスト言語のドキュメントを参照してください。

参考：『Oracle Rdb7 SQL Reference Manual』の SQL モジュール言語の章には、種々のホスト言語に対するデータ型宣言の項があります。

4.2.6 パラメータ・データ型の LANGUAGE 句の効果

SQL モジュールにインクルードしたプロシージャには、コール側モジュールで宣言された実パラメータのデータ型に対応するデータ型を持つパラメータの宣言が含まれています。プロシージャ・パラメータの詳細は、4.2.2 項 (4-4) を参照してください。

SQL モジュール・ヘッダーの LANGUAGE 句で識別した言語コンパイラがサポートしていないプロシージャ・パラメータのデータ型を指定すると、SQL モジュール・プロセッサはコンパイル時の警告を返します。たとえば、SQL は、BIGINT データ型をサポートしていますが、PL/I 言語コンパイラはサポートしていません。この警告を返すことで、モジュール・プロセッサは、プログラム言語では宣言できないコール側モジュール・パラメータで値を保存するつもりであると仮定します。（多くの場合、プロシージャ・パラメータおよび実パラメータは同じデータ型を持つ必要があります。）

場合によっては、このコンパイル時の警告を無視することもできます。たとえば、次のように示します。

- ある列が BIGINT として定義されていて、このデータ型はプログラム言語ではサポートされていないとします。しかし、このデータ型は、プログラムからコールして値を解釈する forms 製品がサポートしています。この場合、プログラム言語が許可している方法で 8 バイトの実パラメータを定義するよう選択して（おそらく 8 文字のテキスト文字列として）、SQL モジュールのプロシージャ・パラメータと関連するフォーム・フィールドの両方で BIGINT として定義します。実行時に、SQL と forms 製品が 8 バイト記憶領域に保管された BIGINT 値を正しく検索して、保管します。プログラム言語の実行時ルーチンは、値を処理しません。
- 使用しているプログラム言語が、言語の実行時ルーチンがサポートしていないデータ型に一致するユーザー定義のデータ型を作成できるとします。この場合、正しく解釈する

独自のルーチンを作成して、特定の記憶領域の値を処理するつもりであれば、コンパイル時の警告を無視できます。

しかし、その他の場合、プロシージャをコールするプログラム言語がサポートしているデータ型を使用して、SQL モジュールでパラメータを宣言する必要があります。したがって、多くの場合、プログラム言語がデータ型をサポートしていないという SQL モジュール・プロセッサの警告は、プロシージャ・パラメータの宣言で何かが間違っているか、LANGUAGE 句で別の言語を指定しているかを示しています。種々の言語がサポートしているデータ型の宣言と変換の詳細は、8 章 (8-1) を参照してください。さらに、SQL のデータ型とホスト言語のデータ型の対応は、『Oracle Rdb7 SQL Reference Manual』の表を参照してください。

言語識別子として、GENERAL を指定すると、SQL モジュール・プロセッサは、SQL が有効であると見なすすべてのデータ型を有効と考えます。この場合、この方法でパラメータを宣言しているので、コンパイル時には、何の警告も受け取りませんが、実行時にエラーが発生するか、予期せぬ結果が生じる可能性があります。これは、使用するホスト言語の機能と制限を十分理解していない場合に起こりうるデメリットです。

4.2.7 パラメータ引渡しメカニズムにおける LANGUAGE 句の効果

言語識別子は、実行時に SQL が予測するパラメータ値をホスト言語モジュールとの間で受け渡すメカニズムを暗黙的に指定します。GENERAL を除いたすべての言語識別子で、SQL はコール側モジュールから渡されるパラメータがその言語のデフォルトを使用すると予測します。GENERAL を指定する場合、SQL はすべてのパラメータがコール側モジュールからの参照によって渡されると予測します。SQL モジュール内のプロシージャをコールする言語を識別する利点は、ほとんどの部分で、コール側モジュールを書いたプログラマが、コールされたモジュールによって予測されるパラメータ引渡しメカニズムの識別を心配する必要がないことです。

通常、ほとんどの言語に対するデフォルトの引渡しメカニズムは、データ型が異なれば変化します。外部プロシージャ・コール用のホスト言語ドキュメントには、デフォルトの引渡しメカニズムについての情報があります。次のいずれかの文が真であれば、この情報を理解する必要があります。

- コールされる SQL モジュールの LANGUAGE 句で、コールしたモジュールを記述した言語以外の言語を指定するか、GENERAL を指定します。
- SQL モジュール内のプロシージャに対するコールは、汎用データ型の言語に特定のサブタイプを使用するホスト言語パラメータを指定し、SQL は汎用データ型のみをサポートします。

注意: 言語によっては、与えられたデータ型に対するサブタイプを含み、各サブタイプに対して異なるデフォルトの引渡しメカニズムを割り当てることがあります。ホスト言語パラメータ宣言がこれらのサブタイプを指定する場合、SQL か、ホスト言語モジュールのいずれかのサブタイプに対するデフォルトの引渡しメカニズムを上書きする必要があります。

先に示した注意のポイントを説明すると、テキスト文字列データ型を使用する場合、Ada では文字数を含む文字列（デフォルトの引渡しメカニズムは参照）を文字数を含まない文字列（デフォルトの引渡しメカニズムは記述子）と同じように宣言できます。

しかし、SQL モジュールでは、文字数を指定しなければ、テキスト文字列データ型は宣言できません。したがって、SQL モジュールで LANGUAGE ADA を指定した場合、SQL は引き渡されるテキスト文字列パラメータは、常に参照であると予測します。これは、仮パラメータを SQL プロシージャの Ada 宣言内の文字列として宣言し、その文字列に対して文字数を指定しなければ、Ada モジュールか、SQL モジュールのいずれかで、そのパラメータに対するデフォルトの引渡しメカニズムを上書きする必要があることを意味します。

SQL は、値による引渡しメカニズムをサポートしていません。SQL モジュール内のプロシージャをコールする場合、パラメータを参照または記述子によって引き渡す必要があります。

パラメータ引渡しメカニズムの詳細は、4.2.8 項 (4-13) を参照してください。

4.2.8 プロシージャ・パラメータ用のデフォルトの引渡しメカニズムの上書き

記述子によって渡されるパラメータ値が必要ならば、BY DESCRIPTOR 句を SQL モジュール内のプロシージャ・パラメータ宣言の一部として指定できます。たとえば、次のように指定します。

```
SIZE_PARAMETER    INTEGER    BY DESCRIPTOR
```

BY DESCRIPTOR 句を指定しなければ、パラメータに対する引渡しメカニズムは、モジュールの LANGUAGE 句で指定した言語によって決まります。LANGUAGE 句がコールする言語を識別すると、引渡しメカニズムのデフォルトは、コール側のホスト言語モジュールとコールされた SQL モジュールで同じです。

次に示す理由により、SQL モジュールのプロシージャ・パラメータ宣言で BY DESCRIPTOR 句を指定する場合、注意が必要です。

- ホスト言語モジュールで実パラメータを指定する場合、参照によるデフォルトを上書きするよう計画します。この場合、SQL モジュール内の対応するパラメータに対して、記述子による引渡しメカニズムを明示的に指定する必要があります。

- プロシージャ・パラメータの宣言で CHECK キーワードを指定して、コール側モジュールが SQL モジュールで予測したようにパラメータを引き渡すことを確認します。4.2.9 項 (4-14) では、パラメータ引渡しメカニズムの実行時の確認について説明します。
- SQL 構文は、汎用データ型（テキスト文字列など）の唯一のサブタイプの正確なコピーです。そして、デフォルトの引渡しメカニズムが SQL 構文で一致できたものとは異なるサブタイプを使用して、仮パラメータを宣言します。この場合、SQL またはホスト言語モジュール内のパラメータに対するデフォルトの引渡しメカニズムを上書きする必要があります。（SQL のデフォルトが記述子によるものであれば、引渡しメカニズムのデフォルトをホスト言語モジュールでのみ変更できます。）

4.2.9 コール側モジュールによって使用されるパラメータの実行時のチェック要求

ホスト言語モジュールが、SQL モジュールの対応するプロシージャ・パラメータのデータ型とサイズに一致しない実パラメータを宣言すると、実行時の結果は未定義になります。プログラムをデバッグする場合、プロシージャと実パラメータが一致しなかったときに通知される実行時エラーの受信は有効です。

この実行時エラーの戻りを有効にするには、SQL モジュールのパラメータ宣言で BY DESCRIPTOR CHECK 句を指定して、対応するホスト言語パラメータを SQL を使用して確認します。デフォルトの引渡しメカニズムが記述子であるとしても、BY DESCRIPTOR CHECK 句を含める必要があります。たとえば、次のように示します。

```
SIZE_PARAMETER INTEGER BY DESCRIPTOR CHECK
```

記述子によるメカニズムがデフォルトの引渡しメカニズムでない場合、ホスト言語モジュールでもパラメータに対して、記述子による引渡しメカニズムを指定します。

4.3 プロシージャでの単一 SQL 文の使用法

実行可能 SQL 文が 1 文だけ含まれる SQL モジュール・プロシージャは、単純文プロシージャと呼ばれます。単純文プロシージャでは、セミコロン (;) を 2 度入力します。1 つは最後のパラメータ宣言の後に入力し、2 番目は実行可能 SQL 文の後に入力します。次に、単純文プロシージャがオープン・カーソルからデータをフェッチする例を示します。

```
PROCEDURE FETCH_REPORT_RECORD      -- Procedure name
(                                   -- Beginning parenthesis
  SQLSTATE,                        -- Status parameter with no data type
  :P_EMPLOYEE_ID                   CHAR(5),  -- Parameters prefixed by colons
  :P_LAST_NAME                     CHAR(14), -- and separated by commas
  :P_FIRST_NAME                    CHAR(10),
  :P_JOB_CODE                      CHAR(4),
  :P_DEPARTMENT_CODE              CHAR(4),
  :P_SALARY_AMOUNT                 REAL
```

```

);                                -- Ending parenthesis and semicolon
FETCH REPORT_CURSOR INTO         -- Simple statement
:P_EMPLOYEE_ID, :P_LAST_NAME,
:P_FIRST_NAME, :P_JOB_CODE,
:P_DEPARTMENT_CODE, :P_SALARY_AMOUNT; -- Ending semicolon

```

4.4 複数文プロシージャにおける複合文の使用法

複合文内に1つ以上の実行可能SQL文を含むSQLモジュール・プロシージャは、複数文プロシージャと呼ばれています。

単純文プロシージャは、データベースへのアクセスを一度に1文に制限しています。複数文プロシージャと唯一のホスト言語プログラム・コールを使用すると、包括的なビジネス・トランザクション、複雑なプログラムの制御および構造化エラー処理を、すべてデータベース環境内で実行できます。多くの場合、1つの物理モジュールが1つの論理データベース機能を実行する形で機能を抽象化できます。

複数文プロシージャでは、最後のパラメータ宣言、複合文内に含まれる各SQL分の最後および複合文自体の最後にセミコロン (;) を入力します。

次の例は、ネストされたIF文を持つFORループを含む複数文プロシージャを示しています。このプロシージャは、いくつかのジョブについて最低給与を引き上げた後、最高給与が最低給与よりも低ければ、最大給与を引き上げます。さらに、各カテゴリについて更新された行数を数えます。

```

PROCEDURE salary_inc              -- Procedure name
(                                -- Beginning parenthesis
    SQLSTATE,                    -- Status parameter with no data type
    :inc REAL,                   -- Parameters prefixed by colons
    :min_count SMALLINT,        -- and separated by commas
    :max_count SMALLINT
);                                -- Ending parenthesis and semicolon
BEGIN                            -- Beginning of compound statement
    SET :min_count = 0;          -- Assignment statements
    SET :max_count = 0;
    -- FOR statement.
    -- The :jobrec variable represents a record that holds columns from the
    -- selected row.
    FOR :jobrec
        AS EACH ROW OF TABLE CURSOR JOB_CURSOR FOR
    -- The select expression.
        SELECT MINIMUM_SALARY, MAXIMUM_SALARY FROM JOBS
            WHERE MINIMUM_SALARY < 20000
    DO
    -- Update the current row in the JOB_CURSOR
        UPDATE JOBS
            SET MINIMUM_SALARY = MINIMUM_SALARY + (MINIMUM_SALARY * :inc)

```

```
        WHERE CURRENT OF JOB_CURSOR;
        SET :min_count = :min_count +1;
-- Nested IF statement.
-- If the minimum salary is now greater than the maximum salary, increase the
-- maximum salary.
        IF :jobrec.MINIMUM_SALARY > :jobrec.MAXIMUM_SALARY
        THEN
            UPDATE JOBS
                SET MAXIMUM_SALARY = MAXIMUM_SALARY * :inc
                WHERE CURRENT OF JOB_CURSOR;
            SET :max_count = :max_count +1;
        END IF;
    END FOR;
END;
```

複合文の記述方法の詳細は、12 章 (12-1) を参照してください。

4.5 SQL モジュール言語の制限について

SQL モジュール言語を使用する場合、SQL は、次の制限を強制的に適用します。

- キーワード、ユーザー定義名またはリテラル（引用符で囲まれた文字列）は、次の行に継続できません。これらは、SQL モジュール・ファイルの 1 行内に完全に収める必要があります。
- **WHENEVER** 文を指定できません。さらに、**WHENEVER** 文をホスト言語ソース・ファイル内に指定しても、SQL モジュール内のプロシージャに対するコールに適用されるよう期待することはできません。**WHENEVER** 文は、SQL プリコンパイラでのみサポートされており、これはホスト言語ソース・ファイル内に埋め込まれた SQL 文のみを識別できます。SQL プリコンパイラは、ユーザー定義プロシージャ・コールを、SQL 操作に関連しているものとしては認識できません。
SQL モジュール内のプロシージャへのコールによって返されたエラーは、コール側のホスト言語モジュールで処理してください。ホスト言語条件文を使用して、コールの直後に **SQLSTATE** や **SQLCODE** などの SQL ステータス・パラメータを評価します。
- モジュールが **CREATE DATABASE** 文を含んでいる場合は、字句的にすべての **DECLARE TABLE** 文の前に記述する必要があります。**DECLARE TABLE** 文を **CREATE DATABASE** 文の前に記述すると、SQL は、デフォルトのデータベース (**SQL\$DATABASE** など) に接続しようとします。
- データ定義文は、そのデータ定義文の前に記述されていないデータベース・オブジェクトや、コンパイル時のデータベースで定義されていないデータベース・オブジェクトを参照することはできません。たとえば、**CREATE STORAGE MAP** 文がテーブルを参照する場合、そのテーブルはコンパイル時のデータベース内に存在するか、または SQL

モジュールが字句的に CREATE STORAGE MAP 文の前の文内でそのテーブルを作成する必要があります。

- SQL INCLUDE 文を使用して、リポジトリまたはテキスト・ソース・ファイルからホスト言語宣言をコピーすることはできません。しかし、SQL モジュール・プロシージャで FROM パス名句を使用して、リポジトリからレコード定義をコピーできます。また、ホスト言語の INCLUDE 文または COPY 文を使用して、ホスト言語定義をホスト言語ソース・ファイルにコピーできます。
- LANGUAGE 句で言語を C として指定した場合、SQL は、すべての C 文字列を NULL 終了文字列に変換します。これは、SQL がこれらの文字列をデータベースからプログラムに渡す場合に、文字列の末尾に NULL 用の空白を予約していることを意味します。プログラムが入力用に文字列をデータベースに渡す場合、SQL は NULL を検索して、データベースに何文字格納するかを決定します。SQL は、NULL の前にある文字のみを保存します。NULL 自体は、保存されません。
SQL が C 文字列を変換する方法が原因となって、C 文字列内のバイナリ・データをデータベースに渡す、またはデータベースから受け取るアプリケーションで問題が発生するかもしれません。バイナリ・データには、NULL が含まれている可能性があり、したがって SQL が予期せぬデータの切り捨てを起こす可能性があります。C ホスト言語プログラムで SQL モジュール言語を使用する場合にこの種の問題を回避するには、モジュール言語を GENERAL として指定することが考えられます。しかし、多くの場合、モジュール言語として C のかわりに GENERAL を使用する必要はありません。

これらの制限のいくつかは、8.6 項 (8-8)、8.12.2 項 (8-27) および 10.2 項 (10-4) でも説明します。

SQL モジュールとホスト言語ファイルの処理

この章では、SQL モジュールとホスト言語ファイルの処理方法について説明します。以降の各項では、次に示す項目を説明します。

- SQL モジュール・プロセッサの起動
- SQL モジュールとホスト言語モジュールの処理
- 高速コンパイルでのパラメータ・チェックの回避
- リモート・データベースにアクセスする場合の SQL モジュール・プロセッサのパフォーマンスを改善
- SQL モジュール言語でのコンテキスト・ファイルの使用
- SQL モジュールのスコープの決定

5.1 SQL モジュール・プロセッサの起動

SQL モジュール・プロセッサは、SQL モジュール・ソース・ファイルの文をチェックし、処理する実行可能イメージです。致命的エラーがなければ、モジュール・プロセッサはオブジェクト・ファイルを生成します。モジュール・プロセッサを使用するには、モジュール・プロセッサを起動して、入力ファイル名として SQL モジュール・ソース・ファイルを指定します。SQL モジュール・ソース・ファイルのデフォルトのファイル拡張子は、.sqlmod です。

Compaq Tru64 UNIX

CompaqTru64 UNIX では、sqlmod コマンドを使用して、入力ファイル名を指定し、モジュール・プロセッサを起動します。たとえば、SQL モジュール・ファイル my_program.sqlmod を処理するには、次のコマンドを使用します。

```
$ sqlmod my_program.sqlmod
```

デフォルトでは、SQL モジュール・プロセッサはリンカーの入力ファイルとしてファイル拡張子 .o を持つオブジェクト・ファイルを生成します。◆

OpenVMS VAX
OpenVMS Alpha

OpenVMS の場合、ログイン・コマンド・ファイルまたは DCL コマンド・レベルで、次に示すシンボルを定義して、SQL モジュール・プロセッサを実行します。

```
$ SQLMOD ::= $SQL$MOD
```

前の例で示したように SQLMOD をシンボルとして定義した場合、コマンドラインで入力ファイルを指定するか、入力用の SQL モジュール・プロセッサ・プロンプトを表示できます。たとえば、次のように示します。

```
$ SQLMOD my_program.sqlmod
$ SQLMOD
INPUT FILE> my_program
```

SQL モジュール・プロセッサは、OpenVMS Linker ユーティリティの入力ファイルとしてファイル拡張子 .obj を持つオブジェクト・ファイルを生成します。◆

入力ファイル名に加えて、SQL モジュール・プロセッサ・コマンドラインでオプションのコンテキスト・ファイルを指定できます。(コンテキスト・ファイルの詳細については、5.5 項 (5-5) を参照してください。)

SQL モジュール・プロセッサ・コマンドラインには、複数の修飾子を入力できます。これについては、『Oracle Rdb7 SQL Reference Manual』で説明しています。

Compaq Tru64 UNIX

次の例は、結果として生じるオブジェクト・ファイルの名前を指定する -o 修飾子と CompaqTru64 UNIX システム上のリスト・ファイルの名前を指定する -list 修飾子を使用する方法を示しています。

```
$ sqlmod -list my_program.lis -o my_mod_program.obj ◆
```

致命的なエラーが発生すると、SQL モジュール・プロセッサは、コンパイルを停止します。致命的エラーを複数見つけるには、複数回コンパイルを行う必要があります。

OpenVMS
Alpha

OpenVMS Alpha 上でエラーのある SQL モジュール言語プログラムをコンパイルした場合、OpenVMS VAX や CompaqTru64 UNIX 上とは異なり、.obj ファイルは削除されません。◆

5.2 SQL モジュールとホスト言語モジュールの処理

SQL モジュール内のプロシージャをコールするプログラムの処理には、次の 4 つのステップがあります。

1. SQL モジュール・ファイルをオブジェクト・ファイルに処理（コンパイル）する
5.1 項 (5-2) で説明したように SQL モジュール・プロセッサを使用して、SQL ソース・ファイルをコンパイルします。
2. ホスト言語ソース・ファイルをオブジェクト・ファイルに処理（コンパイル）する
適切なホスト言語コンパイラを使用して、ホスト言語ソース・ファイルをコンパイルします。
3. オブジェクト・ファイルをリンクして、実行可能ファイルまたは共有イメージを作成する
ホスト言語コンパイラでコールされるリンカーで処理を行い、オブジェクト・ファイルをリンクします。オブジェクト・ファイルを SQL ライブラリとともにリンクする必要があります。
4. 実行可能イメージを実行するか、（必要であれば）共有イメージをインストールする

Compaq Tru64 UNIX

次の例では、Bourne シェルを使用して実行可能イメージを CompaqTru64 UNIX 上で作成します。

```
$ # Invoke the SQL module processor.
$ sqlmod my_sql_module
$
$ # Define a global symbol for the SQL libraries.
$ SQLLIBS='-lrdbsql -lrdbshr -lcosi -lots'
$ export SQLLIBS
$
$ # Invoke the Digital C compiler and link together the SQL module object
$ # file, the host language object file, and the SQL libraries.
$ cc -o my_calling_module my_calling_module.c my_sql_module ${SQLLIBS}
$ # Run the application.
$ my_calling_module ◆
```

OpenVMS
VAX

OpenVMS
Alpha

次の例では、OpenVMS 上で実行可能イメージを作成します。

```
$ ! Define a symbol to invoke the module processor.
$ SQLMOD := $SQL$MOD
$ ! Invoke the SQL module processor.
$ SQLMOD my_sql_module
$ ! Invoke the Pascal compiler.
$ PASCAL my_calling_module
```

```
$ ! Link the files.  
$ LINK my_calling_module, my_sql_module  
$ ! Run the application.  
$ RUN my_calling_module
```

前の例では、論理名 LNK\$LIBRARY に SQL\$USER.OLB が定義されていると仮定しています。



次の項では、SQL ソース・ファイルのコンパイルを説明します。ホスト言語ソース・ファイルのコンパイル命令は、各ホスト言語のドキュメントを参照してください。Ada ソース・ファイルは、前に示した例と同じ方法でコンパイルされないことに注意してください。Ada ソース・ファイルのコンパイルについては、Ada のドキュメントを参照してください。

プログラム開発の最終フェーズの詳細は、7 章 (7-1) を参照してください。

5.3 高速コンパイルでのパラメータ・チェックの回避

OpenVMS VAX ≡≡≡ OpenVMS Alpha ≡≡≡

[NO]PARAMETER_CHECK 修飾子は、SQL モジュール・プロセッサがコンパイル中に SQL 文のプロシージャで指定したパラメータの総数とともに宣言した仮パラメータの総数を比較するかどうかを決定します。修飾子は、次のとおりです。

- PARAMETER_CHECK (デフォルト)
パラメータの数が一致しているかを確認し、一致していなければ、実行時（コンパイル時ではない）にエラーを生成します。
- NOPARAMETER_CHECK
パラメータのチェックを停止して、モジュールのコンパイル時間を短縮します。SQL モジュールをデバッグした後で、NOPARAMETER_CHECK 修飾子を使用することを考慮してください。◆

5.4 リモート・データベース用の SQL モジュール・プロセッサのパフォーマンスの改善

SQL モジュール・プロセッサは、プログラムが参照するテーブルおよびビューの定義にアクセスする必要があります。デフォルトでは、モジュール・プロセッサは、データベース・システム・テーブルにアクセスして、テーブルおよびビューの定義を検索します。データベースがリモートにある場合、SQL モジュールのコンパイルはデータベースがローカルにある場合よりも定義の検索に時間も追加リソースも必要になる可能性があります。

リモート・データベースにアクセスする場合、3つのオプションを使用して SQL モジュール・プロセッサのパフォーマンスを改善できます。最初の2つのオプションの詳細な説明に関しては、15.1.3 項 (15-3) を参照してください。3番目のオプションは、ここで説明します。

- DECLARE ALIAS 文の COMPILETIME オプションを使用して、データベースのローカル・リポジトリで作成したノードのパス名を指定します。

- DECLARE ALIAS 文の COMPILETIME オプションを使用して、ローカルな Oracle Rdb データベース・ファイルを指定します。
- プログラムで DECLARE TABLE 文を使用して、SQL 文で指定したテーブルとビューの定義を指定します。
SQL モジュール・プロセッサは、この方法で宣言したテーブル定義用のデータベースにアクセスしません。DECLARE TABLE 文に他のテーブルを参照する制約が含まれていれば、これらのテーブルも同様に宣言するようにします。コンパイル時間のパフォーマンスを改善するオプションに加えて、DECLARE TABLE 文は未作成のテーブルにアクセスするプログラムをコンパイルする方法を提供します。

参考: DECLARE TABLE 文の詳細は、『Oracle Rdb7 SQL Reference Manual』の SQL 文の章の DECLARE TABLE 項を参照してください。
データベースに接続するコンパイル時間のオプションについての説明は、『Oracle Rdb7 SQL Reference Manual』の SQL 文の章の DECLARE ALIAS 項を参照してください。

5.5 SQL モジュール言語でのコンテキスト・ファイルの使用

SQL モジュール言語で SQL コンテキスト・ファイルを使用できます。コンテキスト・ファイルは、プログラムをコンパイルして実行する場合に適用する DECLARE 文を含む SQL コマンド・プロシージャです。コンテキスト・ファイルを使用すると、コンパイルされたソース・ファイルの移植性を改善できます。

1つの例外として、SQL モジュール言語で使用するコンテキスト・ファイルの書式はプリコンパイル SQL で使用するものと同じです。SQL モジュール言語でコンテキスト・ファイルを使用する場合、DECLARE 文をセミコロン (;) で終了する必要はありません。しかし、セミコロン (;) を入れると、SQL モジュール言語とプリコンパイル SQL の両方で、コンテキスト・ファイルを使用できます。

あるアプリケーションに異なる SQL 表現形式を使用してコンパイルする必要があるモジュールが含まれていると仮定します。モジュールのコピーを2つ持ち、それを並行して維持する問題を持つよりも、1つのモジュールと2つのコンテキスト・ファイルを持つほうがよいでしょう。このモジュールは、すべてのコードを含み、各コンテキスト・ファイルは表現形式 (dialect) 宣言文を含みます。

たとえば、モジュール test は、dialcet SQL92 および MIA を使用してコンパイルする必要があります。コンテキスト・ファイル test-sql92 には、次の DECLARE MODULE 文が含まれています。

```
DECLARE MODULE
    DIALECT SQL92
```

コンテキスト・ファイル test-mia には、次の DECLARE MODULE 文が含まれています。

```
DECLARE MODULE  
DIALECT MIA
```

表現形式は、test モジュールを適切なコンテキスト・ファイルでコンパイルすることで制御されます。test モジュールで SQL92 セマンティクスを使用するには、test-sql92 コンテキスト・ファイルを使用して test モジュールをコンパイルします。次の例で示すように 2 番目のパラメータにコンテキスト・ファイルを指定します。

```
$ SQLMOD  
INPUT FILE> test test-sql92
```

test モジュールで MIA セマンティクスを使用する場合、次の例で示すように test-mia コンテキスト・ファイルを使用してコンパイルを行います。

```
$ SQLMOD  
INPUT FILE> test test-mia
```

5.6 SQL モジュールのスコープの決定

実行可能ファイルや共有イメージは、2 つ以上のホスト言語モジュールと 2 つ以上の SQL 言語モジュールでアセンブルできます。次に示す方法を使用すると、SQL 操作を SQL ソース・ファイルに編成できます。

- SQL モジュールを実行可能イメージに関連付けることができます。

この場合、イメージへ組み込まれる任意のホスト言語ソース・モジュールからコールできる 1 つの SQL ソース・モジュールを作成します。このストラテジは、1 つのアプリケーションに対しては最も簡単です。しかし、モジュールのコールが様々なプログラム言語で記述されている場合、このストラテジは、コール側モジュールのデフォルトと異なる可能性がある 1 つ、またはいくつか、あるいはすべてのデータ型に対する SQL 受渡しメカニズムの確立に欠点があります。

指定されたデータ型に対するデフォルトの受渡しメカニズムが、SQL モジュールが期待しているデフォルトの受渡しメカニズムと異なっている場合、デフォルトを上書きする必要があります。(モジュール間でパラメータを受け渡す方法の詳細は、3.2.5 項、4.2.7 項および 4.2.8 項を参照。)

- 各 SQL モジュールは、1 種類のタスクにのみ関連付けることができます。

たとえば、1 つの SQL モジュールを、1 つのテーブル内または制約によって関係付けられた複数のテーブル内に含まれる列のセットの更新と関連付けたい場合があります。この場合、該当するタスクがまた、単一のコール側ホスト言語ソース・ファイルのスコープを定義すると、SQL モジュールとそれをコールするホスト言語モジュールの両方が、特定の種類のタスクに関して 1 対 1 の対応関係を持つことになります。データベース・アプリケーションの数が増加するのに従って、このストラテジは、モジュール間のお互いの関連を追跡しやすく、またプログラムのメンテナンスを簡素化します。

プリコンパイル SQL の使用方法

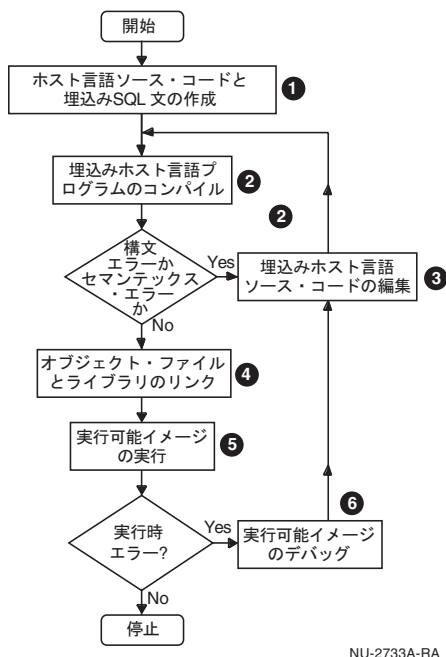
この章では、ホスト言語ソース・ファイルに SQL 文を埋め込むための一般的および言語固有のガイドラインについて説明します。次の項目を詳細に説明します。

- SQL 文を含むプログラムを記述する場合に従うプロセス
- 埋込み SQL 文を含むホスト言語プログラムの記述
- SQL プリコンパイラの起動
- コンパイル・エラーの検出
- リモート・データベースに対するパフォーマンスの改善
- コンパイル時とランタイム特性の指定
- SQL プリコンパイラがサポートする各言語における埋込み SQL 文

6.1 プリコンパイラ・プロセスについて

SQL プリコンパイラを使用すると、ホスト言語プログラムに SQL 文を埋め込むようになります。図 6-1 (6-2) のフローチャートは、プリコンパイル SQL 文を使用してプログラムを作成する方法に従って説明します。図の後で、番号を付けた項目について説明します。

図 6-1 SQL プリコンパイラを使用したアプリケーション・プログラム開発



次に、図 6-1 (6-2) の番号に対応する説明を示します。

1. ホスト言語プログラムを作成し、コード内に SQL 文を埋め込み、各 SQL 文の前に EXEC SQL フラグを付けます。
2. SQL プリコンパイラを起動して、プリコンパイル SQL 文およびホスト言語プログラムを処理します。
3. コンパイル・エラーが発生したら、ソース・コードを編集します。実行フェーズ（ステップ）でエラーが発生したら、このステップに戻ります。ソース・コードの編集が終わったら、コンパイル・フェーズに戻ります（ステップ）。
4. オブジェクト・ファイルとライブラリをリンクして、実行イメージまたは共有イメージまたは共有モジュールを作成します。（共有イメージまたは共有モジュールを他のプログラムとリンクできます。）7 章 (7-1) では、両方のオプションを説明します。

5. リンク・フェーズ（ステップ）から作成された実行可能イメージをテストして、プログラムが正しく動作するか検証します。共有イメージをインストールする、または他のモジュールにリンクする必要がある場合があります。
6. 実行時エラーを発見したら、実行イメージをデバッグして、編集フェーズ（ステップ）に戻り、ホスト言語ソース・コードまたは埋込み SQL 文を修正します。

Compaq Tru64 UNIX

次の例では、Compaq Tru64 UNIX 上で Bourne シェルを使用して、my_program.sc という C 言語ソース・ファイルを使用して、実行可能イメージを作成します。

```
$ # Invoke the SQL precompiler and specify the language as C.
$ sqlpre -l cc my_program.sc
$
$ # Define a global symbol for linking.
$ SQLLIBS='-lrdbsql -lrdbshr -lcosi -lots'
$ export SQLLIBS
$
$ # Invoke the DEC C compiler and link the object file.
$ cc -o my_program my_program.o ${SQLLIBS}
$ # Run the application.
$ my_program ◆
```

OpenVMS VAX
OpenVMS Alpha

次の例では、OpenVMS 上で my_program.sc という C 言語ソース・ファイルを使用して、実行可能イメージを作成します。

```
$ ! Invoke the SQL precompiler and specify the language as C.
$ SQLPRE
INPUT FILE> my_program /CC
$ ! Link the file.
$ LINK my_program
$ ! Run the application.
$ RUN my_program
```

OpenVMS 上では、リンカーを起動する前に論理名 LNK\$LIBRARY を SQL\$USER として定義します。◆

参考: オブジェクト・モジュールをリンクしてイメージにする方法と実行可能イメージを実行する方法の詳細は、7 章 (7-1) を参照してください。

6.2 ホスト・プログラムへの SQL 文の埋込み

SQL プリコンパイラを使用すると、ホスト言語プログラム内に単純 SQL 文と複合文の両方を埋め込めます。

- 単純 SQL 文
単純 SQL 文は、1 つの SQL 文から構成されます。C プリコンパイル・プログラムから

の次の引用で示すように、文の前に EXEC SQL フラグを置いて単純 SQL 文を埋め込み、言語に特定の終端文字で終了します。

```
EXEC SQL UPDATE DEPARTMENTS
        SET MANAGER_ID = :mgrid
        WHERE DEPARTMENT_CODE = 'SALE';
```

- 複合文
1 つまたは複数の SQL 文で構成される複合文は、キーワード BEGIN および END で区切ります。複合文内で、SQL IF および FOR などのフロー制御文をインクルードできます。単純 SQL 文で行ったように、文の前に EXEC SQL フラグを置き、言語に固有の終端文字で複合文全体を終端することで複合文を埋め込みます。さらに、セミコロン (;) を使用して、複合文内の各 SQL 文を終了します。
12 章 (12-1) では、SQL プリコンパイラで複合文を使用する方法を説明します。

SQL プリコンパイラは、プログラム言語ソース・ファイル进行处理し、これらのファイルに SQL 文を含める方法についてのある規則を加えます。これらの規則は、任意のプリコンパイラ・ファイルに適用されます。

- 各 SQL 文は EXEC SQL フラグで始まります。
EXEC SQL フラグは、SQL プリコンパイラに対して、SQL 文の先頭を識別させます。EXEC SQL フラグは、マルチライン SQL 文の最初の行の先頭だけに置くことができます。(複数行に渡る SQL 文に対するホスト言語継続文字の必要性は、言語固有の必要性です。使用しているプログラム言語の規則に従って、SQL 文を 1 行から次の行に続けます。)
- EXEC SQL フラグは、その他のホスト言語要素として同じ行に置くことができます。たとえば、SQL 文は、プログラムのセクションに対してラベルの前に来る可能性があります。EXEC SQL フラグは、同じ行の言語文 (たとえば、IF 文) に続くこともできます。
- SQL 文内の複数行リテラルを 1 行から別の行に続けるには、リテラルの継続についてのホスト言語規則に従います。
- 実行可能文が使用できるプログラムの一部で、実行可能なホスト言語文、実行可能 SQL 文、またはその両方をコピーする INCLUDE 文を指定します。ホスト言語宣言をコピーする INCLUDE SQLCA、INCLUDE FROM リポジトリ、および INCLUDE 文は、変数の宣言が行えるプログラムの一部に必要があります。
変数宣言内では、部分的なホスト言語宣言をコピーする INCLUDE 文を指定できません。たとえば、FILE.DAT には、次の行が含まれると仮定します。

```
05 FIELD1          PIC X(10) .
05 FIELD2          PIC X(10) .
```

この状況では、プリコンパイラは COBOL プログラムの次に示すセクションに含まれる INCLUDE 文を受け入れません。

```
WORKING-STORAGE SECTION.
```



```

01  DEPT_REC                                PIC X(24) .
01  COMMAREA
EXEC SQL INCLUDE 'FILE.DAT'      END-EXEC.

```

- プログラムの任意の部分に DECLARE 文を埋め込みます。しかし、DECLARE ALIAS、DECLARE TABLE および DECLARE TRANSACTION 文は、これらの DECLARE 文が含む情報に依存した文の前に置く必要があります。たとえば、データベースを明示的に宣言し、ソース・ファイル内にトランザクションを指定する場合、DECLARE ALIAS 文または DECLARE TABLE 文は、ソース・ファイル内の OPEN 文の前に置く必要がある DECLARE（または SET）TRANSACTION 文の前に置く必要があります。しかし、DECLARE CURSOR 文は、ソース・ファイル内の関連する OPEN 文の前に置く必要はありません。
- 任意の CREATE DATABASE 文を埋め込む場合、これらは字句的に DECLARE TABLE 文の前に現れる必要があります。DECLARE TABLE 文が CREATE DATABASE 文より前に置かれると、SQL はデフォルトのデータベース（SQL\$DATABASE など）に接続しようとしします。
- データ定義文は、コンパイル時のデータベースで定義されていないデータベース・オブジェクトまたはデータ定義文の前にはデータベース・オブジェクトを参照できません。たとえば、CREATE STORAGE MAP 文はテーブルを参照するので、そのテーブルはコンパイル時のデータベース内に存在するか、または字句的に CREATE STORAGE MAP 文より前にある埋込み SQL プログラムでそのテーブルを作成する必要があります。
- SQL プリコンパイラは、パラメータ名の処理に対して、言語固有の規則に従います。たとえば、C のソース・ファイルを処理する場合、SQL プリコンパイラは、C パラメータ名に関して大文字と小文字を異なる文字として扱います。しかし、SQL プリコンパイラは、データベース・エンティティの名前および SQL キーワードの名前に対して常に SQL 規則を適用します。
したがって、パラメータ名に適用する規則に関係なく、次の項目を実行できます。
 - SQL キーワードデータベース・エンティティの名前を入力するときはハイフンではなくアンダースコア（_）を指定する必要があります。
COBOL で使用する END-EXEC フラグは、この規則には当てはまりません。
ANSI/ISO SQL 標準に従って、END-EXEC フラグを指定します。END_EXEC フラグはコンパイルしますが、標準には準拠しません。
 - SQL キーワードおよびデータベース・エンティティの名前を入力する場合に大文字または小文字で指定できます。
しかし、SQLCODE、SQLSTATE、SQLCA、SQLDA および SQLDA2 は、SQL と同様にホスト言語で処理されるパラメータを参照するキーワードです。C 言語などのパラメータに対して大文字小文字の区別を適用する言語で記述している場合、SQLCODE、SQLSTATE、SQLCA、SQLDA および SQLDA2 を大文字で指定する必要があります。

- ホスト言語の必要に応じて、各文を終了します。表 6-1 (6-6) では、各ホスト言語で SQL 文を終了させる規則を示しています。

表 6-1 プリコンパイル・ホスト言語ソース・ファイルでの SQL 文の終了

言語	SQL 文の終了
Ada	セミコロン (;) で SQL 文を終了。
C	セミコロン (;) で SQL 文を終了。
COBOL	END-EXEC フラグ ^{*1} で SQL 文を終了。その後 SQL 文の最後の単語の後に少なくとも 1 つの空白を入れて、END-EXEC フラグを挿入するか、行内の唯一の単語として SQL 文のすぐ後に END-EXEC フラグを置くことができます。
FORTTRAN	FORTTRAN 文として、SQL 文を終了。言い換えると、次の行に続く継続文字を指定していなければ、SQL 文は行の終端で終了します。
Pascal	Pascal の IF-THEN-ELSE 文内であっても、SQL 文は、セミコロン (;) で終了。たとえば、ELSE 句の前に SQL 文を埋め込む場合、SQL 文を BEGIN-END ブロックで囲み、SQL 文をセミコロン (;) で終了します。
PL/I	セミコロン (;) で SQL 文を終了。

^{*1} プログラムの構造内で SQL 文がどこに表示されるかによりますが、END-EXEC フラグのすぐ後にピリオド (.) を加える必要がある場合もあります。COBOL 文の同じ位置でピリオドが必要であれば、加えてください。

6.7 項 (6-13) に、SQL プリコンパイラがサポートする各言語に対する追加規則およびガイドラインを示しています。

6.3 プリコンパイラの起動

SQL プリコンパイラを使用するには、プリコンパイラを起動し、入力ファイル名と、ソース・ファイルを記述したホスト言語を指定します。

Compaq Tru64 UNIX

Compaq Tru64 UNIX では、sqlpre コマンドを使用して、プリコンパイラを起動します。さらに、入力ファイル名、-l 修飾子および言語固有の修飾子を指定してください。たとえば、C ホスト言語のソース・ファイル进行处理するには、次の例のように、-l 修飾子と言語固有の cc 修飾子を指定します。

```
$ sqlpre -l cc my_program.sc
```

文字列をホスト言語コンパイラに渡すときは、次のように、文字列を一重引用符 (') で囲みます。

```
-l fortran='-extend_source'
```

SQL は、渡した文字列が有効かどうかをチェックしません。◆

OpenVMS
VAX ≡≡≡
Alpha ≡≡≡

OpenVMS では、SQL プリコンパイラを起動するには、ログイン・コマンド・プロシージャ内または DCL コマンド・レベルでシンボルを定義します。SQL プリコンパイラの起動には、次のシンボルを定義できます。

```
$ SQLPRE := $SQL$PRE
```

C 言語のソース・ファイルを処理するために SQL プリコンパイラを起動するには、次のように、SQLPRE シンボルに、ファイル名と言語固有の修飾子 /CC を指定します。

```
$ SQLPRE  
INPUT_FILE> my_program.sc /CC ◆
```

表 6-2 (6-7) は、プログラムのプリコンパイルで利用できるデフォルトの入力ファイルのタイプと言語識別修飾子を示します。この表では、プログラムをホスト言語コンパイラに渡すときに、プリコンパイラが使用するデフォルトのファイル・タイプも示します。

表 6-2 プログラムのプリコンパイルで使用する言語識別子およびデフォルトのファイル拡張子

SQL プリコンパイラ				ホスト言語コンパイラ			
ホスト言語	入力 ファイル	OpenVMS 修飾子	Compaq Tru64 UNIX 修飾子	出力 ファイル	入力 ファイル	OpenVMS 出力ファイル	Compaq Tru64 UNIX 出力ファイル
Ada	.sqlada	/ADA	なし	.ada	.ada	.obj	なし
C	.sc	/CC	-l cc	.c	.c	.obj	.o
COBOL	.sco	/COBOL	-l cobol	.cob	.cob	.obj	.o
FORTRAN	.sfo	/FORTRAN	-l fortran	.for	.for	.obj	.o
Pascal	.spa	/PASCAL	-l pascal	.pas	.pas	.obj	.o
PL/I	.spl	/PLI	なし	.pli	.pli	.obj	なし

OpenVMS
VAX ≡≡≡
Alpha ≡≡≡

OpenVMS では、プリコンパイラを起動するシンボルを、言語固有として定義できます。この場合、シンボル定義には、言語修飾子も指定します。次の例に、SQL プリコンパイラと COBOL 言語のシンボル定義を示します。

```
$ SCOB      := $SQL$PRE/COB
```

SQL は、コンパイラに定義したシンボルを使用して、言語コンパイラを起動します。ここではシンボルは、コマンド・プロシージャを実行しないものとします。たとえば、COBOL/DEBUG を意味するならば、シンボル COBOL はサポートされていますが、@cob_proc を意味するならばサポートされません。

(Ada プログラムでは、コンパイルの前に、Ada ライブラリを作成および有効化しておく必要があります。詳細は、6.7.2 項 (6-14) を参照してください)。

言語コンパイラの起動用に定義したシンボルを使用して、[NO]OBJECT や [NO]G_FLOAT 修飾子を渡すことはできません。この修飾子は、プリコンパイル時に指定してください。たとえば、次のようなシンボル定義を使用すると、プリコンパイラは、G_FLOAT 修飾子を認識しなくなります。

```
$ MYC ::= CC /GFLOAT
$ MYSQL ::= SQLPRE/CC
$ MYSQL FILE_A
```

[NO]OBJECT と [NO]G_FLOAT 修飾子を、SQL プリコンパイラ起動用のシンボル定義の一部として指定するか、シンボルを使用して SQL プリコンパイラを起動するときに、コマンド行に指定してください。◆

Compaq Tru64 UNIX

Compaq Tru64 UNIX では、SQL プリコンパイラは、次の修飾子をホスト言語コンパイラに引き渡しません。

- -list
- -match
- -form ansi

コンパイラのスイッチ・リストで、適切な修飾子を明示的に指定してください。たとえば、COBOL ホスト言語コンパイラの場合、-form ansi 修飾子を使用するには、-ansi を指定する必要があります。

```
$ sqlpre -l cobol="-ansi" -form ansi
```

コンパイラのスイッチ・リストで使用する修飾子の詳細は、コンパイラのドキュメントを参照してください。◆

SQL プリコンパイラを起動するために入力するコマンドには、次の 2 つのパラメータを指定できます。

- ホスト言語のソース・ファイルのファイル指定（必須）
ホスト言語ファイルには、ホスト言語コードとプリコンパイルした SQL 文を格納する必要があります。表 6-2 (6-7) のように、ソース・ファイルのデフォルトのファイル・タイプは、言語修飾子で指定されているホスト言語によって異なります。
- SQL コンテキスト・ファイルのファイル指定（オプション）
SQL は、**コンテキスト・ファイル**を特殊用途のコマンド・プロシージャとして定義し、プログラムのプリコンパイルや実行時に適用する宣言を含みます。コンテキスト・ファイルが必要になるのは、SQL の実装環境から別の環境に移植するために、プリコンパイルしたソース・ファイルを作成する場合のみです。コンテキスト・ファイルの詳細は、6.6.2 項 (6-12) を参照してください。

SQL プリコンパイラのコマンド行には、次のように、プリコンパイラとホスト言語コンパイラの両方の修飾子が指定されます。

- SQL プリコンパイラ修飾子
プリコンパイラ修飾子を指定して、ソースを記述したホスト言語を指定する必要があります。SQL プリコンパイラのコマンド行には、言語識別修飾子の他にも、いくつかの修飾子を指定できます。上記の修飾子の詳細は、『Oracle Rdb7 SQL Reference Manual』の SQL プリコンパイラの章を参照してください。
- ホスト言語コンパイラ修飾子
プリコンパイルしたソース・ファイルを言語コンパイラに渡すときに、プリコンパイラで使用する言語コンパイラ修飾子を指定できます。プリコンパイラのコマンド行で使用可能な言語プリコンパイラ修飾子の詳細は、ホスト言語のドキュメントを参照してください。

プリコンパイラのコマンド行は、255 文字を超えることはできません。

注意: SQL プリコンパイラで作成した言語ソース・モジュールを編集した後で、ホスト言語コンパイラを使用してソース・モジュールをコンパイルしないでください。変更内容が、SQL 文に影響を与えない場合も同様です。これは、次にソース・ファイルをコンパイルするときに、プリコンパイラで作成した一時的な言語ソース・モジュールへの変更内容を、プリコンパイル済の SQL モジュールが上書きしてしまうためです。

OpenVMS
VAX

OpenVMS VAX では、プリコンパイラは、次のような操作を行います。

1. 渡されたファイルから、次の 2 つのソース・ファイルを作成します。
 - SQL 文をマクロ・プロシージャに変換するマクロ・モジュール（ファイル・タイプは .mar）。
 - 言語コンパイラが処理可能なホスト言語モジュール。このホスト言語モジュールでは、プリコンパイラは、SQL 文を、マクロ・モジュールの外部プロシージャへのコールに置き換えます。
2. マクロ・アセンブラを起動し、マクロ・モジュール用オブジェクト・ファイル（ファイル・タイプは .mob）を作成し、プリコンパイルした言語ソース・ファイルを適切な言語コンパイラに渡します。
3. 言語コンパイラが出力したオブジェクト・ファイルと、マクロ・モジュールのオブジェクト・ファイルを 1 つのファイルにまとめます。プリコンパイラは、.mar ファイルと .mob ファイルを削除します。（Ada プログラムの処理では、プリコンパイラは、オブジェクト・ファイルの連結を実行しません。Ada プログラムのプリコンパイルの詳細は、6.7.2 項 (6-14) を参照してください）。◆

OpenVMS
Alpha

OpenVMS Alpha では、プリコンパイラは、出力ファイルを 1 つ作成し、適切な言語コンパイラに渡します。マクロ・モジュールを作成しません。◆

Compaq Tru64 UNIX では、プリコンパイラは、出力ファイルを 1 つ作成し、適切な言語コンパイラに渡します。マクロ・モジュールを作成しません。◆

6.4 プリコンパイル時およびコンパイル時のエラー検出

SQL プリコンパイラがプログラムの処理中にエラーを検出すると、端末にエラーを表示するか、ホスト言語ソース・ファイルにエラーを書き込みます。エラー・メッセージの詳細情報を取得する方法は、『Oracle Rdb7 SQL Reference Manual』の付録のエラー・メッセージの項を参照してください。

ホスト言語コンパイラがエラーを検出すると、コンパイラは、SQL プリコンパイラがアクセス可能な場所にエラーを書き込みます。SQL プリコンパイラは、エラー・メッセージを取り出し、端末またはログ・ファイルに表示して、`sqlerr.log` というログ・ファイルに書き込みます。LIST または `-list` 修飾子を指定している場合、ホスト言語コンパイラは、エラー・メッセージをリスティング・ファイルに書き込みます。

注意： SQL プリコンパイラは、プリコンパイルしたソース・コードをホスト言語コンパイラに渡した後で、`sqlerr.log` を作成します。発生したエラーによっては、プリコンパイラは、ホスト言語コンパイラへソース・コードを渡さないで、`sqlerr.log` を作成しない場合があります。

プリコンパイラは、`sqlerr.log` ファイルを作成しない場合は、表示や編集が可能なホスト言語ソース・ファイルを作成するので、エラーを参照してください。たとえば、プリコンパイラに渡したファイルの名前が `myprog.sfo` の場合、`myprog.for` という名前のファイルを表示または編集すると、プリコンパイル時エラーを参照できます。

コンパイル時のエラーに関する情報は、ホスト言語のドキュメントを参照してください。

6.5 リモート・データベースに対するプリコンパイラのパフォーマンスの改善

SQL プリコンパイラは、プログラムが実行時にアクセスするテーブルおよびビューの定義にアクセスする必要があります。デフォルトでは、プリコンパイラは、データベース・ファイルのシステム・テーブルにアクセスして、テーブルおよびビューの定義を検索します。データベースがリモートに存在する場合には、ローカルに存在する場合よりもプリコンパイルに時間がかかり、定義の検索により多くのリソースが必要になる可能性があります。

リモート・データベースにアクセスする場合、プリコンパイラのパフォーマンスを改善するオプションが 3 つあります。最初の 2 つのオプションの詳細な説明に関しては、15.1.3 項 (15-3) を参照してください。3 番目のオプションは、ここで説明します。

- DECLARE ALIAS 文の COMPILETIME オプションを使用して、データベースのローカル・リポジトリで作成したノードのパス名を指定します。
- DECLARE ALIAS 文の COMPILETIME オプションを使用して、ローカルな Oracle Rdb データベース・ファイルを指定します。
- プログラムで DECLARE TABLE 文を使用して、SQL 文で指定したテーブルとビューの定義を指定します。
定義はプログラムの一部として含まれているので、SQL プリコンパイラは、データベースにアクセスする必要はありません。DECLARE TABLE 文を使用する場合に複数のソース・ファイルのメンテナンスを削減するには、これらの文を 1 つのテキスト・ファイルに指定します。SQL INCLUDE file-spec 文を使用して、プログラムにテキスト・ファイルをコピーします。
DECLARE TABLE 文に他のテーブルを参照する制約定義が含まれていれば、これらのテーブルも同様に宣言するようにします。

参考: DECLARE TABLE 文および DECLARE ALIAS 文の詳細は、『Oracle Rdb7 SQL Reference Manual』の SQL 文の章を参照してください。

6.6 コンパイル時と実行時のオプションの指定

SQL を使用すると、プリコンパイラ修飾子と DECLARE MODULE 文を通して、多くの設定を制御できます。これらの設定は、処理を行っているデータのキャラクタ・セットなどの特性や、プリコンパイラが ANSI/ISO SQL 標準に従わない構文をフラグで知らせるかどうかに影響を与えます。

プリコンパイラ修飾子の完全な一覧やその使用法に対する規則については、『Oracle Rdb7 SQL Reference Manual』を参照してください。次の項では、これらの設定が使用できるいくつかの方法を説明します。

6.6.1 DECLARE MODULE 文の使用法

DECLARE MODULE 文を直接ホスト言語プログラムまたはコンテキスト・ファイルに記述することで、プリコンパイラ・プログラムに対して様々なコンパイル時や実行時の SQL 設定に影響を与えることができます。例 6-1 (6-11) は、ホスト言語プログラムに記述できる DECLARE MODULE 文を示します。

例 6-1 文を使用してコンパイル時と実行時の設定を変更

```
EXEC SQL DECLARE MODULE employee_module
      DIALECT SQL92
      NAMES ARE DEC_KANJI
      NATIONAL CHARACTER SET KANJI
      CATALOG ADMINISTRATION
```

```
SCHEMA ACCOUNTING
DEFAULT CHARACTER SET DEC_KANJI
AUTHORIZATION RDB$DBHANDLE
CHARACTER LENGTH OCTETS
ALIAS RDB$DBHANDLE;
```

DECLARE MODULE 文を使用して、次のオプションのカテゴリを指定できます。

- 表現形式オプション
DIALECT 句は、1 つの電子スイッチで複数のライトを制御する場合と同じように、1 つの句で、その他の複数の句に対する設定を制御します。句が影響を与える設定の一覧については、『Oracle Rdb7 SQL Reference Manual』を参照してください。
- キャラクタ・セット・オプション
キャラクタ・セット・オプションは、モジュールに対するデフォルト、識別子、各国語およびテラル・キャラクタ・セットを制御します。
- モジュール言語オプション
モジュール言語オプションは、ALIAS および RIGHTS などの複数の句で構成されます。そのうちのいくつかは DIALECT 句で設定できます。
- 認可識別子とマルチスキーマ・オプション
これらのオプションは、デフォルトの認可識別子、スキーマおよびカタログを指定します。

『Oracle Rdb7 SQL Reference Manual』では、DECLARE MODULE 文の詳細と、同時に指定できる句を説明しています。

6.6.2 SQL コンテキスト・ファイル内に宣言を記述する方法

コンテキスト・ファイルを使用すると、プリコンパイル SQL プログラムを他のプラットフォームへ簡単に移動できます。コンテキスト・ファイルは、プログラムをコンパイルして実行する場合に適用する DECLARE 文を含む SQL コマンド・プロシージャです。

たとえば、OpenVMS では、test.sco という名前の COBOL ソース・ファイルをプリコンパイルして、test_declares.sql ファイルの DECLARE 文を適用するには、次のコマンドを使用します。

```
$ SQLPRE/COBOL test test_declares
```

例 6-2 (6-12) は、プリコンパイルするソース・ファイルに対する宣言を含むコンテキスト・ファイル test_declares.sql を示します。

例 6-2 プリコンパイル SQL のコンパイルに対するコンテキスト・ファイル

```
DECLARE ALIAS FILENAME pers;
DECLARE TRANSACTION READ WRITE
        RESERVING EMPLOYEES FOR PROTECTED WRITE,
        JOB_HISTORY FOR PROTECTED WRITE,
```



```

DEPARTMENTS FOR SHARED READ,
JOBS FOR SHARED READ;

DECLARE MODULE employee_module
  DIALECT SQL92
  NAMES ARE DEC_KANJI
  NATIONAL CHARACTER SET KANJI
  CATALOG ADMINISTRATION
  SCHEMA ACCOUNTING
  DEFAULT CHARACTER SET DEC_KANJI
  AUTHORIZATION RDB$DBHANDLE
  CHARACTER LENGTH OCTETS
  ALIAS RDB$DBHANDLE;

```

2つ以上のソース・ファイルで同じコンテキスト・ファイルを使用できるか、各ソース・ファイルに対して異なるコンテキスト・ファイルを使用できるか、または1つのソース・ファイルで異なるコンテキスト・ファイルを使用できます。

あるアプリケーションに異なる SQL 表現形式を使用してコンパイルする必要があるモジュールが含まれていると仮定します。モジュールのコピーを2つ持ち、それを並行して維持する問題を抱えるよりも、1つのモジュールと2つのコンテキスト・ファイルを持つほうが優れています。このモジュールは、すべてのコードを含み、各コンテキスト・ファイルは表現形式（dialect）宣言文を含みます。

表現形式は、test モジュールを適切なコンテキスト・ファイルでコンパイルすることで制御されます。

コンテキスト・ファイルのフォーマットは、任意の SQL コマンド・プロシージャと同じで、ファイル内の DECLARE 文の前には、EXEC SQL フラグ（これらはソース・ファイルに埋め込まれた SQL 文内にあるので）がなく、最後は常にセミコロン (;) で終わっています。

プログラムの移植性を改善するためにコンテキスト・ファイル内に DECLARE TABLE および DECLARE STATEMENT など、その他の DECLARE 文を含めることができます。

コンテキスト・ファイルを使用する場合、プログラムのプリコンパイルに必要な DECLARE 文はコンテキスト・ファイルに入っていることをホスト言語ソース・ファイル内に記録する必要があります。

6.7 SQL プリコンパイラの使用に関する言語固有のガイドライン

この項では、SQL プリコンパイラがサポートする言語に関して、言語ごとのガイドラインについて説明します。ここでは、Ada、C、COBOL、FORTRAN、Pascal、PL/I について説明します。

6.7.1 Ada ソース・ファイルでの SQL 文の埋込み

この項では、Ada プログラム固有の情報について説明します。Ada プログラムでは、プリコンパイルとリンクの操作で注意が必要です（6.7.2 項 (6-14) と 7.2.3 項 (7-4) を参照してください）。Ada 言語に固有のパラメータの宣言や使用方法については、8.12.1 項 (8-27) を参照してください。

ファイル名の長さの制限

Ada プリコンパイラで使用できるファイル名の長さは、最大 27 文字に制限されています。Ada コンパイラでは、最大 31 文字ですが、SQL プリコンパイラは "SQL_" という接頭辞をファイル名に追加してパッケージ名を作成します。

名前付きリテラルと範囲

Ada プリコンパイラは、名前付きリテラルと範囲をサポートしません。この制限を回避するには、SQL モジュール言語を使用してください。

複数のプロシージャを含む Ada ファイルでの SQL 文の埋込み

SQL プリコンパイラは、Ada プログラムのブロック構造をサポートします。したがって、同じ Ada ソース・ファイルに含まれる複数のプロシージャの中で、SQLCODE などの SQL 文が参照するパラメータを宣言することができ、プリコンパイラはこれを独立したパラメータとして認識します。

サブプログラムのオーバーロード

SQL Ada プリコンパイラは、サブプログラムのオーバーロードをサポートしません。Ada プリコンパイラは、オーバーロードしたプログラムすべてを、1 つのネームスペースとして扱います。

解決方法の 1 つとして、SQL 文で使用しているすべての名前が、オーバーロードしたすべてのプロシージャ内で一意であることを確認してください。ANSI/ISO 規格に準拠するには、ホスト言語の変数名は、プログラム全体で一意でなければなりません。

6.7.2 Ada プログラムのプリコンパイル

Ada コンパイラと、SQL プリコンパイラがサポートする他のホスト言語コンパイラは特性が異なるため、プリコンパイラが Ada ソース・ファイルを処理する方法や、プリコンパイルする方法も異なります。Ada ソース・ファイルの処理に関連するファイルの概要は、表 6-3 (6-15) を参照してください。

Ada コンパイラと、SQL プリコンパイラがサポートするその他のコンパイラには、次のような相違点があります。

Ada ライブラリの作成と有効化

SQL 文を含む Ada プログラムをプリコンパイルするには、まず Ada ライブラリを作成し、有効にしてください。これには、DEC Ada プログラム・ライブラリ・マネージャの ACS コマンドである CREATE LIBRARY と SET LIBRARY を使用します。

ACS の詳細は、Ada のドキュメントを参照してください。

SQL プリコンパイラが作成するもう 1 つの .ada ファイル

プリコンパイラは、Ada プログラムを処理すると、.ada ファイルをもう 1 つ作成します。これは、SQL 文プロシージャ用の Ada パッケージ指定ファイルです。パッケージ指定ファイルのファイル名は、頭文字 SQL とアンダースコア (_) に、入力ファイル名を追加したものであり、ファイル・タイプは .ada です。Ada コンパイラは、パッケージ指定を使用します。これは、プリコンパイラが作成するパッケージの定義です。

Ada ライブラリへの中間ファイルのコピー

Ada 以外の言語では、プリコンパイラは、SQL 文を処理して作成したオブジェクト・ファイルと、ホスト言語コンパイラが作成したオブジェクト・ファイルを連結してから、中間オブジェクト・ファイルを削除します。

これに対して、Ada プログラムでは、プリコンパイラはオブジェクト・ファイルの連結や削除を実行しません。プリコンパイラが、埋込み SQL 文を処理して作成したオブジェクト・ファイルのファイル名は、頭文字 SQL とアンダースコア (_) に、入力ファイル名を追加したものです。ファイル拡張子は .obj です。ただし、プリコンパイラは、ハイフン (-) やドル記号 (\$) をアンダースコア (_) を置き換え、アンダースコアが続けて複数ある場合は、1 つに減らします。プログラムをリンクする前に、ACS COPY FOREIGN コマンドでこのファイルを Ada ライブラリにコピーするか、ACS LINK コマンドで明示的にファイル名を付ける必要があります。

表 6-3 (6-15) は、Ada ソース・モジュールの処理中に、使用および作成されるファイルの一覧です。

表 6-3 Ada ソース・モジュールのプリコンパイルに関連するファイル

ファイル名	目的
ファイル名 .sqlada	SQL プリコンパイラに渡す、SQL 文を埋め込んだ Ada ソース・ファイル。
ファイル名 .ada	プリコンパイラが作成し、Ada コンパイラに渡される Ada の中間ソース・ファイル。
ファイル名 .obj	ファイル名 .ada から Ada プリコンパイラが作成するオブジェクト・ファイル。このファイルは、.sqlada ファイルと同じディレクトリではなく、Ada ライブラリ・ディレクトリ内にあります。

表 6-3 Ada ソース・モジュールのプリコンパイルに関連するファイル（続き）

ファイル名	目的
sql_ ファイル名 .mar ^{*1}	SQL プリコンパイラが作成する中間マクロ・ソース・ファイル。 .sqlada ファイルに埋め込まれた SQL 文を実装するプロシージャを含みます。プリコンパイラは、ファイルをマクロ・プロセッサに渡し、論理名 SQL\$KEEP_PREP_FILES が定義されていなければ、これを削除します。
sql_ ファイル名 .obj	マクロ・プロセッサが .mar ファイルから作成したオブジェクト・ファイル。他のホスト言語でのオブジェクト・ファイルとは異なり、プリコンパイラは、このファイルとファイル名 .obj と連結することではなく、削除しません。リンクの前に、このファイルを Ada ライブラリにコピーしておくか、ACS LINK コマンドで明示的に指定してください。
sql_ ファイル名 .ada	.mar ファイルで実装されている SQL 文プロシージャ用に、プリコンパイラが作成する Ada パッケージ指定。

^{*1} OpenVMS VAX のみ。

例 6-3 (6-16) は、my_program.sqlada という名前の Ada ファイルをプリコンパイルする手順を示しています。

例 6-3 Ada ファイルのプリコンパイル

```
$ ! Create an Ada library:
$ ACS CREATE LIBRARY [.ADALIB]
$ ! Enable the library:
$ ACS SET LIBRARY [.ADALIB]
%ACS-I-CL LIBIS, Current program library is DVD15: [PROGRAMS.ADALIB]
$ ! Invoke the SQL precompiler:
$ SQLPRE
INPUT FILE> my_program.sqlada/ADA
```

Ada プログラムのリンクの例は、7.2.3 項 (7-4) を参照してください。

6.7.3 C ソース・ファイルでの SQL 文の埋込み

この項では、C プログラム固有の情報について説明します。C 言語に固有のパラメータの宣言や使用方法については、8.12.2 項 (8-27) を参照してください。

C での文字列の使用方法

SQL C プリコンパイラを使用すると、SQL は、すべての C 文字列を NULL で終了する文字列に変換します。したがって、SQL がこの文字列をデータベースからプログラムに渡すときには、文字列の最後に、NULL 文字用のスペースを予約します。プログラムが入力のために

文字列をデータベースに渡すと、SQL は NULL 文字を探すことによって、データベースに格納する文字数を認識します。SQL は、NULL 文字の前にある文字列のみを格納し、NULL 文字自体は格納しません。

SQL は、このような方法で C 文字列を変換するため、バイナリ・データをデータベースとやり取りするアプリケーションでは、問題が発生することがあります。SQL C プリコンパイラでこのような問題が発生しないように、SQL が提供する \$SQL_VARCHAR データ型を使用してください。\$SQL_VARCHAR データ型を宣言すると、バイナリ・データのやり取りが可能になります。

複数のプロシージャを含む C ファイルでの SQL 文の埋込み

SQL プリコンパイラは、C プログラムのブロック構造をサポートします。したがって、同じ C ソース・ファイルに含まれる複数のプロシージャの中で、SQLCODE などの SQL 文が参照するパラメータを宣言することができ、プリコンパイラはこれを独立したパラメータとして認識します。

C での浮動小数点データ

OpenVMS VAX
OpenVMS Alpha

C プログラムで、浮動小数点データに D 浮動小数点形式を使用する場合、NOG_FLOAT 修飾子を指定してコンパイルしてください。これにより、SQL は、データベース内の倍精度データを、Oracle Rdb で使用する G 浮動小数点形式から、C で使用する D 浮動小数点形式へ変換します。◆

OpenVMS VAX

OpenVMS VAX では、コール可能なイメージを、VAXCRTLG にリンクする必要があります。◆

C ポインタの使用

C 言語用の SQL プリコンパイラは、SQL 文のパラメータの受渡しで、ポインタの使用をサポートしていません。

インクルード・ファイルを使用した SQL ルーチンの宣言

C プログラムでは、SQL ルーチンのコールで、コンパイル情報メッセージを生成できます。Oracle Rdb は、インクルード・ファイル sql_rdb_headers.h を提供し、明示的にコールする SQL 関数のプロトタイプを提供することにより、メッセージを排除しています。詳細は、10.3.4 項 (10-30) を参照してください。

:VAX C 拡張子

Compaq Tru64 UNIX

Compaq Tru64 UNIX では、:VAX C 拡張子を含むプログラムをコンパイルするときに、次のオプションを sqlpre コマンド行で使用できます。

```
-l cc="migrate -vaxc" ◆
```

6.7.4 COBOL ソース・ファイルでの SQL 文の埋込み

この項では、COBOL プログラム固有の情報について説明します。

複数行に渡るリテラル

COBOL プログラムに埋め込んだ文では、SQL は、次の 2 通りの方法により、複数行に渡ってリテラルを指定することができます。

- ANSI/ISO SQL 規格に準拠する方法として、一重引用符 (') を開始の引用符文字、ハイフン (-) をインジケータ領域、二重引用符 (") を継続文字として使用します。
- プログラムが ANSI/ISO SQL 規格に準拠する必要がない場合は、COBOL のルールを使用することができ、継続文字と、リテラルの開始を示す引用符文字は合わせる必要があります。

COBOL ソース・ファイルが正しいフォーマットであることの確認

SQL プリコンパイラでは、コマンド行での COBOL ソース・ファイルの形式を端末形式と ANSI/ISO 形式のいずれかに指定できます。デフォルトは、端末形式です。ソース・ファイルが ANSI/ISO 形式の場合、コマンド行で、ANSI_FORMAT または -form ansi 修飾子を指定してください。

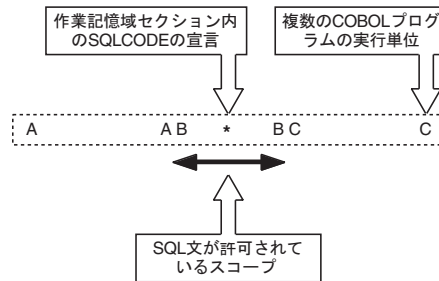
複数のプログラムを含む COBOL ファイルでの SQL 文の埋込み

プリコンパイルした COBOL ソース・ファイルが複数のプログラムを含んでいる場合、すべての SQL 文を、その中の 1 つのプログラムのスコープ内で実行する必要があります。

SQL プリコンパイラは、COBOL プログラムのブロック構造をサポートしていないので、ソース・ファイルに複数のプログラムが含まれている場合でも、SQLCODE 宣言は、プリコンパイルしたソース・ファイルで 1 つしか指定できません。ソース・ファイルのワーキング・ストレージ・セクションの 1 つで直接 SQLCODE を宣言するか、ファイルのワーキング・ストレージ・セクションで INCLUDE SQLCA 文を指定して間接的に宣言してください。SQLCODE を直接または間接的に宣言した場合も、SQLCODE パラメータのロケーションによって、SQL コンパイラが処理するプログラムが決まります。実行時に、SQL は SQLCODE パラメータを更新し、SQL 文の実行ステータスを反映します。したがって、実行可能な SQL 文は、SQLCODE パラメータが宣言されているプログラム内に格納してください。コンパイラは、すべての SQL 文が同じプログラム内にあるものとします。

図 6-2 (6-19) では、アスタリスク (*) が SQLCODE 宣言のロケーションを示しています。大文字アルファベットのペアは、ファイル内のプログラムの区切りです。矢印は、SQL 文を指定できるファイルのセクションを示しています。プログラム B の END PROGRAM 文の後にあるプリコンパイルした SQL 文では、ホスト言語コンパイラ・エラーが発生します。

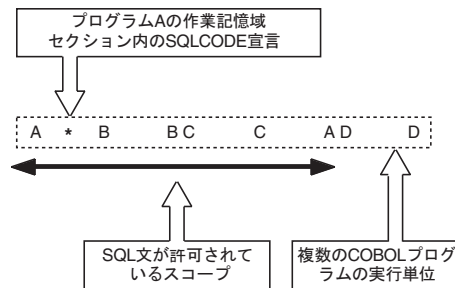
図 6-2 SQLCODE 宣言のスコープ (COBOL)



NU-2722A-RA

プログラムはネスト構造になっているので、SQLCODE 宣言をしたプログラムに含まれるプログラムにも、SQL を含めることができます。たとえば、図 6-3 (6-19) のプログラム A には、プログラム B と C が含まれています。プログラム A のワーキング・ストレージ・セクションには SQLCODE 宣言があるので、プログラム A、B および C に SQL 文を含めることができます。プログラム D の SQL 文は、プログラム A の END PROGRAM の後にくるので、ホスト言語コンパイラ・エラーが発生します。

図 6-3 SQL 文が許可されるスコープ (COBOL)



NU-2723A-RA

実行可能プログラム・イメージに、ネスト化されていない COBOL プログラムがいくつか含まれていて、複数のプログラムで SQL 文を実行させる場合は、プログラムを別のソース・ファイルに分割します。

構造の宣言

COBOL では、ワードとそれに続く文字列として宣言された構造は、1 つの変数として扱われます。このタイプは、SQL の VARCHAR(n) に相当します。次の例は、1 つの変数として扱われる構造を示しています。

```
VARCHAR (32)    01 STRUCTURE.
```

```
49 STRUCT_LENGTH PIC S9(4) COMP.  
49 STRUCT_STRING PIC X(32).
```

6.7.5 FORTRAN ソース・ファイルでの SQL 文の埋込み

この項では、FORTRAN プログラム固有の情報について説明します。

1 行あたりの文字数の制限

プリコンパイルした FORTRAN プログラムでは、FORTRAN オプションである EXTEND_SOURCE または -extend_source を使用しない場合、SQL プリコンパイラは、1 行あたり 72 文字という上限を継承しています。上記のオプションを使用すると、1 行あたりの上限は 132 文字になります。これよりも長い文は、6 桁目に継続文字を入力すると、次の行へ続けることができます。

Compaq Tru64 UNIX

Compaq Tru64 UNIX では、次の例のように、-extend_source 修飾子を使用するだけでなく、コンパイラ・オプションとしての指定も必要です。

```
sqlpre -l fortran='-extend_source' -extend_source test.sfo ◆
```

継続行の数の制限

OpenVMS
VAX

VAX FORTRAN コンパイラでは、文で使用できる継続行の最大数（最大 99）を、CONTINUATIONS 修飾子で指定できます。継続行のデフォルト数は、19 です。DEC FORTRAN コンパイラのデフォルト値は、99 です。

プログラムがレコード定義を使用するときは、SQL プリコンパイラは、レコードをアンパックして各要素に展開し、別々の行に配置します。レコード内の要素の数が継続行の最大数よりも大きい場合、FORTRAN コンパイラはエラーを生成します。

このような場合、FORTRAN コマンド行で CONTINUATIONS 修飾子を指定し、継続行の数を増やしてください。レコードに含まれる要素の数が、FORTRAN で許可される最大数（99 要素）を上回る場合は、中間ファイル（.for）を編集し、複数の要素を 1 行に配置してください。◆

SQL で始まる FORTRAN フィールド名の長さの上限

プリコンパイルした FORTRAN プログラムでは、INCLUDE SQLCA 文を使用したために SQL が宣言した、SQL で始まるパラメータ名には、長さが 6 文字以内という制限があります。たとえば、SQL プリコンパイラは、SQLCODE パラメータを SQLCOD、SQLERRD を SQLERR のように自動的に宣言します。

このマニュアルでは、SQLCA のフィールドには完全な名前を使用します。ただし、FORTRAN 文で指定するときは、SQL で始まる SQLCA 名の先頭 6 文字のみを使用してください。RDB\$MESSAGE_VECOR のパラメータ名とそのフィールドには、この制限は適用されないで、全体を指定する必要があります。

また、INCLUDE SQLCA 文で明示的にパラメータを宣言する場合、プリコンパイルされた FORTRAN プログラムでのパラメータ名は SQLCOD でなければなりません。

FOTRAN IF 文での SQL 文の埋込み

FORTRAN では、SQL 文は、ブロック IF (IF...THEN...END_IF) 文と算術 IF 文のみに埋め込んでください。論理 IF 文 (IF <条件> <文>) には、SQL 文を埋め込まないでください。SQL 文を論理 IF 文に埋め込むと、SQL 文の実行ステータスを評価できなくなります。

FOTRAN DO ループでの SQL 文の埋込み

SQL 文は、FORTRAN DO ループの最後の文として、ラベルを付けることはできません。プリコンパイラは、SQL の WHENEVER 文を、各実行可能 SQL 文に続く 1 つ以上のホスト言語の IF 文へ変換します。IF 文は、FOTRAN DO ループを終了できません。(SQL の WHENEVER 文で、SQL 文の実行ステータスをモニターしない場合は、最後の IF 文を入力して、条件とエラーを処理してください)。

SQL 文で、DO ループの最後の処理を実行する場合、END DO 文、またはラベルと CONTINUE 文でループを終了させます。

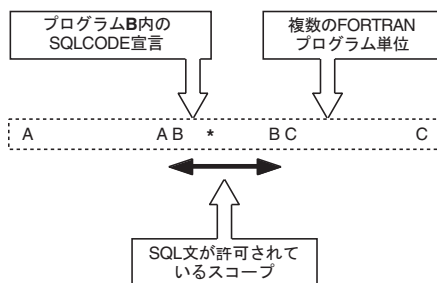
複数の END 文を含む FORTRAN ファイルでの SQL 文の埋込み

FORTRAN ソース・ファイルが複数のプログラム単位 (メイン・プログラムとコールするサブルーチン) を含んでいる場合、すべての SQL 文は、その中の 1 つのプログラム単位のスコープ内で実行する必要があります。

SQL プリコンパイラは、FORTRAN プログラムのブロック構造をサポートしていません。したがって、ソース・ファイルで使える SQLCOD 宣言は 1 つのみです。ソース・ファイルのプログラム単位の 1 つで直接 SQLCOD を宣言するか、ソース・ファイルのプログラム単位の 1 つで INCLUDE SQLCA 文を指定して間接的に宣言してください。SQLCOD を直接または間接的に宣言した場合も、SQLCOD パラメータのロケーションによって、SQL プリコンパイラが処理するプログラムが決まります。実行時に、SQL は SQLCOD パラメータを更新し、SQL 文の実行ステータスを反映します。したがって、実行可能な SQL 文は、SQLCOD パラメータが宣言されているプログラム単位内に指定してください。プリコンパイラは、すべての SQL 文が同じプログラム単位内にあるものとします。

図 6-4 (6-22) では、アスタリスク (*) が SQLCOD 宣言のロケーションを示しています。大文字アルファベットのペアは、ファイル内のプログラム単位の区切りです。矢印は、SQL 文を指定できるファイルのセクションを示しています。このファイルでは、プログラム単位 B の END 文の後にあるプリコンパイルした SQL 文で、ホスト言語コンパイラ・エラーが発生します。

図 6-4 SQL 文が許可されるスコープ (FORTRAN)



NU-2724A-RA

複数のプログラム単位に SQL 文を含めるには、単位を別々のソース・ファイルに分割します。

構造の宣言

FORTRAN では、ワードとそれに続く文字列として宣言された構造は、1 つの変数として扱われます。このタイプは、SQL の VARCHAR(n) に相当します。次の例は、1 つの変数として扱われる構造を示しています。

```
STRUCTURE /struct_name
      INTEGER*2          length
      CHARACTER*32       string
END STRUCTURE
```

6.7.6 Pascal ソース・ファイルでの SQL 文の埋込み

この項では、Pascal プログラム固有の情報について説明します。

ディクショナリからのデータのインクルード

プリコンパイラは、INCLUDE FROM DICTIONARY 文をサポートしていません。

複数のプロシージャを含む Pascal ファイルでの SQL 文の埋込み

SQL プリコンパイラは、Pascal プログラムのブロック構造をサポートします。したがって、同じ Pascal ソース・ファイルに含まれる複数のプロシージャの中で、SQLCODE などの SQL 文が参照するパラメータを宣言することができ、プリコンパイラはこれを独立したパラメータとして認識します。

SQL の制約に従った Pascal ソース・ファイルの変更

SQL プリコンパイラは、Pascal のルールと文のほとんどを認識します。ただし、次のような制限があるので、注意してください。

- SQL プリコンパイラは、Pascal ホスト変数の HIDDEN 属性のみを認識します。

- SQL プリコンパイラがサポートできるポインタ・レベルは1のみです。
- SQL プリコンパイラは、コードを順次読取りするので、プログラムでの文の順序は重要な意味を持ちます。たとえば、変数の宣言は、変数の割当てよりも先に記述する必要があります。

6.7.7 PL/I ソース・ファイルでの SQL 文の埋込み

この項では、PL/I プログラム固有の情報について説明します。

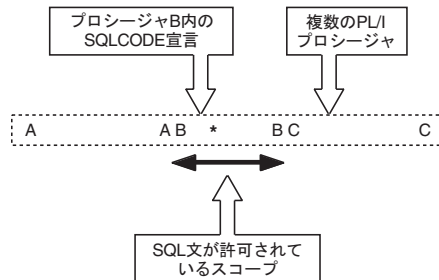
複数のプロシージャを含む PL/I ファイルでの SQL 文の埋込み

プリコンパイルした PL/I ソース・ファイルに、複数のプロシージャまたは関数が含まれている場合、SQL 文は、SQLCODE 宣言のスコープ内で実行する必要があります。

SQL プリコンパイラは、PL/I プログラムのブロック構造をサポートしていません。したがって、ソース・ファイルに複数のプロシージャが含まれている場合でも、SQLCODE 宣言（または INCLUDE SQLCA 文）は、プリコンパイルしたソース・ファイル内で1つしか指定できません。実行時に、SQL は SQLCODE パラメータを更新し、SQL 文の実行ステータスを反映します。実行可能文は、SQLCODE を返すことのできるプロシージャ内で指定する必要があります。

図 6-5 (6-23) では、アスタリスク (*) が SQLCODE 宣言のロケーションを示しています。大文字アルファベットのペアは、ファイル内のプロシージャの区切りです。矢印は、SQL 文を指定できるファイルのセクションを示しています。プログラム単位 B の END 文の後にあるプリコンパイルした SQL 文では、ホスト言語のコンパイラ・エラーが発生します。

図 6-5 SQLCODE 宣言のスコープ (PL/I)

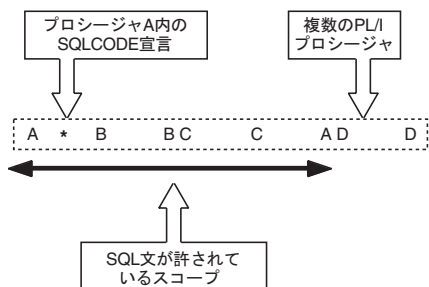


NU-2725A-RA

プログラムはネスト構造になっているので、SQLCODE を宣言したプロシージャに含まれるプロシージャにも、SQL 文を含めることができます。たとえば、図 6-6 (6-24) のプロシージャ A には、プロシージャ B と C が含まれています。プロシージャ A で SQLCODE を宣言しているので、プロシージャ A、B および C に SQL 文を含めることができます。ただし、

プロシージャ D の SQL 文は、プロシージャ A の END 文の後にくるので、SQL プリコンパイラは、プロシージャ D の SQL 文を処理しません。

図 6-6 SQL 文が許可されるスコープ (PL/I)



NU-2726A-RA

プログラム実行のためのイメージの作成

アプリケーション開発では、コンパイル・フェーズが完了したら、1つ以上のオブジェクト・モジュールをリンクして実行可能イメージを作成します。リンカーで実行可能イメージを作成しないと、オペレーティング・システムは、アプリケーション・プログラムの実行を許可しません。リンカーで1つ以上のオブジェクト・モジュールをバインドする際は、共有可能イメージなどの情報のリンクが必要になることがあります。

この章では、次の内容について説明します。

- 実行可能イメージ、共有可能イメージ、共有オブジェクトの概念
- OpenVMS でのオブジェクト・モジュールのリンク
- OpenVMS 上で共有可能イメージ・ファイルを作成することによる、システム・リソースの保存とプログラム・パフォーマンスの改善
- 共有可能イメージのインストール
- Compaq Tru64 UNIX でのオブジェクト・モジュールのリンク
- オブジェクト・ライブラリやアーカイブへのプリコンパイル SQL モジュールの挿入
- リンク・フェーズで作成した実行可能イメージの実行
- 実行可能イメージで実行時エラーが発生した場合の、ホスト言語、SQL コード、または両方のデバッグ

参考: オブジェクト・モジュールのリンク、共有可能イメージおよび共有オブジェクトの作成、実行可能イメージの実行、プログラムのデバッグに関する詳細は、オペレーティング・システムやコンパイラのドキュメントを参照してください。

7.1 実行可能イメージと共有可能イメージ

各ソース・ファイルからオブジェクト・モジュールを作成した後、これをリンクして、実行可能イメージ、共有可能イメージまたは共有オブジェクトを作成します。

実行可能イメージとは、プロセスで実行されるイメージです。実行すると、実行可能イメージが、ファイルから読み込まれ、プロセスで実行されます。

共有可能イメージとは、実行可能イメージや他の共有可能イメージがコールするプロシージャ群です。共有可能イメージは直接実行できませんが、1つ以上のイメージとリンクして、実行可能イメージを作成することは可能です。

異なるバージョンの Oracle Rdb を使用してモジュールの1つを再コンパイルする場合、リンクする前に、すべてのモジュールを再コンパイルしてください。

7.2 OpenVMS Linker の使用方法

OpenVMS OpenVMS
VAX Alpha

注意：Ada コンパイラで生成したオブジェクトからた実行可能イメージを作成する場合は、この項の説明とは異なる手順に従う必要があります。Ada オブジェクト・モジュールのリンクの詳細は、7.2.3 項 (7-4) を参照してください。

SQL 文を含むオブジェクト・モジュールをリンクするには、次の指定を行います。

- プログラム・イメージに必要なすべてのオブジェクト・ファイル
- SQL\$USER ライブラリ・ファイル

コマンド・ラインで直接指定することも、オプション・ファイルで間接的に指定することもできます。プログラムをリンクするコマンド・ラインは（オプション・ファイルを使用しない場合）、次のようになります。

```
$ LINK main_sql_mod, SQL$USER/LIBRARY
```

/LIBRARY 修飾子によって、ファイルがライブラリ・ファイルであることをリンカーに知らせます。

SYS\$SHARE:SQL\$USER.OLB でなく SQL\$USER 論理名に確実にリンクします。

Oracle Rdb をインストールした後に、システム・マネージャが LNK\$LIBRARY 論理名（または LNK\$LIBRARY_1 など、後に続く適切な論理名）を定義して SQL\$USER をデフォルトのユーザー・ライブラリとして定義している場合があります。SQL\$USER がデフォルトのユーザー・ライブラリとして定義された場合は、LINK コマンドにこの SQL ライブラリを指定する必要はありません。

次の例では、SQL\$USER が定義された場合に、1 つのオブジェクト・モジュールをリンクします。

```
$ LINK my_program
```

次の例では、3 つのオブジェクト・モジュールをリンクして my_main_program.exe という 1 つの実行イメージを構成します。

```
$ LINK my_main_program, my_called_program, prog_messages
```

参考: SQL と併用するホスト言語では、言語特有のサポート・ライブラリまたはイメージが必要な場合があります。プログラムがホスト言語に固有のファイルにリンクする必要がある場合は、ホスト言語のドキュメントでプログラムのリンクに関する項を参照してください。
イメージ作成の詳細は、OpenVMS Linker ユーティリティのドキュメントを参照してください。



7.2.1 OpenVMS VAX システム上の Compaq Tru64 C コンパイラでコンパイルしたプログラムのリンク

OpenVMS
VAX

OpenVMS VAX システム上の Compaq Tru64 C コンパイラでコンパイルしたプログラムをリンクする場合は、特定のプログラム・セクション (PSECT) 属性をセット・アップする必要があります。この制限は、SQL プリコンパイラと SQL モジュール・プロセッサの両方に適用されます。この属性をセット・アップしないと、LINK-W-MULPSC 競合属性エラーが返されます。

このエラーを回避するには、次に示すように LINK コマンドに PSECT_ATTR を指定します。

```
$ LINK/EXE=SQL$IVPC -
    SQL$IVPC, -
    SQL$USER/LIB, -
    SYS$INPUT/OPT
    PSECT_ATTR = RDB$MESSAGE_VECTOR,NOSHR,NOPIC ◆
```

7.2.2 共有可能イメージにリンクする実行可能イメージの作成

OpenVMS OpenVMS
VAX Alpha

共有可能イメージを入力として指定するには、オプション・ファイルを使用する必要があります。入力ファイルを共有可能イメージ・ファイルとして特定する /SHAREABLE 位置修飾子は、オプション・ファイル内でしか使用できません。それ以外の場合、リンカーはこれを位置修飾子でなくコマンド修飾子として解釈します。例 7-1 (7-4) に、入力ファイルの指定とリンク・オプションを記載した project3.opt というファイルを示します。

例 7-1 オプション・ファイルを使用した共有可能イメージとのリンク

```
MOD1,MOD7,LIB3/LIBRARY,-  
LIB4/LIBRARY/INCLUDE=(MODX,MODY,MODZ),-  
MOD12/SELECTIVE_SEARCH  
STACK=75  
SYMBOL=JOB CODE,5  
$ LINK/MAP/CROSS_REFERENCE PROGA, PROGB, PROGC, project3/OPTIONS
```

LINK コマンドをコマンド・プロシージャに挿入し、LINK コマンドにオプション・ファイルを指定する場合は、オプション・ファイルとして SYS\$INPUT を指定します。DCL コマンド・インタープリタは、LINK コマンドに続く行をオプション・ファイル内の行として指定します。たとえば、コマンド・プロシージャ linkprog.com には次の行が記載されています。

```
$ LINK, MAIN, SUB1, SUB2, SYS$INPUT:/OPTIONS  
MYPROG/SHAREABLE  
SYS$LIBRARY:APPLPCKGE/SHAREABLE  
STACK=75
```



7.2.3 Ada オブジェクトのリンク

OpenVMS VAX ≡≡≡ OpenVMS Alpha ≡≡≡

コンパイラで生成されたオブジェクトから実行可能イメージを作成するプロセスは、他の言語によるプロセスとは異なります。(7.2 項 (7-2) に、Ada 以外のホスト言語プログラムのオブジェクト・モジュールをリンクする方法を示します)。この項では、次の相違点について詳しく説明します。

- Ada では、DCL LINK コマンドでなく ACS LINK コマンドを使用します。
- Ada コンパイラで作成した .obj ファイル名は、ACS LINK コマンドのパラメータとしては指定できません。かわりに、Ada モジュール内のメイン・プロシージャの名前を ACS LINK コマンドの最初のパラメータとして指定します。
- SQL 文を埋め込んだ Ada ソース・モジュールをコンパイルしてオブジェクト・ファイルが作成される場合は、SQL_filename.obj ファイル（プリコンパイル時に作成された）が SQL_filename.ada ファイルのパッケージ仕様部に対応するパッケージ定義であることを Ada プログラム・ライブラリ・マネージャに指定する必要があります。次の 2 つの方法があります。

```
$ ACS SET LIBRARY [.ADALIB]  
%ACS-I-CL_LIBIS, Current program library is DVD15:[PROGRAMS.ADALIB]  
$ ACS COPY FOREIGN SQL_filename.obj
```

- 次のように、.obj ファイルを ACS LINK コマンドの 2 番目のパラメータとして明示的に指定します。


```
$ ACS LINK main_procedure_name SQL_filename.obj
```

論理名 LNK\$LIBRARY（または後に続く適切な論理名）が SQL\$USER に定義されていない場合、オブジェクト・ファイルが SQL 文を埋め込んだ Ada ソース・モジュールをプリコンパイルして作成されている場合は、次のように LINK コマンドで SQL ファイルを指定する必要があります。

```
$ ACS LINK main_procedure_name SQL_filename.obj, SQL$USER
```

次の例では、MY_PROGRAM というメイン・プロシージャを表すプリコンパイル Ada プログラム sql_my_program.obj をリンクしています。この例では ACS COPY コマンドは指定されません。かわりに、プリコンパイル時に作成した .obj ファイルをメイン・プロシージャ名で指定します。

```
$ ! This LINK command requires that the logical name
$ ! LNK$LIBRARY is defined as SQL$USER.OLB
$ !
$ ! The name MY_PROGRAM, immediately following the word LINK,
$ ! is a procedure name, not a file name.
$ !
$ ACS LINK MY_PROGRAM          sql_my_program.obj
```

ACS LINK コマンドによって、OpenVMS Linker ユーティリティを呼び出します。この項に示す相違点以外に、この章内の他の項に示す情報は Ada オブジェクトにも適用されます。



7.3 共有可能イメージの作成

OpenVMS VAX  OpenVMS Alpha 

OpenVMS Linker ユーティリティは、共有可能イメージを作成できます。次の理由でアプリケーションの一部を共有可能イメージにすることがあります。

- 多くの実行可能イメージがディスク上にある共有可能イメージの 1 つのコピーにリンクするので、ディスク領域を節約します。
- 応答時間が非常に重要なので制御変数とデータの読取りをメイン・メモリーに保持する必要がある場合は、共有可能イメージによってアプリケーションの実装を簡素化します。
- 共有可能イメージを慎重に構築し、ユニバーサル・シンボル（OpenVMS VAX 上の転送ベクトルまたは OpenVMS Alpha 上のシンボル・ベクトル）と位置に依存しないコーディング方法を使用することで、接続するすべてのイメージを再リンクすることなく共有可能イメージへの大幅な変更や機能拡張が可能になります。

共有可能イメージはそれだけでは完了しないので、直接実行することはできません。共有可能イメージを実行するには、実行可能イメージを作成するリンク操作の入力として共有可能イメージを指定します。

共有可能イメージを実行するには、コマンドライン上で /SHAREABLE 修飾子を使用します。実行可能イメージと同様に、共有可能イメージは SQL\$USER ライブラリ・ファイルにリンクします。

共有可能イメージを構築するには、リンカーがプログラム・セクション (PSECT) を処理する方法に関する指定を追加する必要があります。**プログラム・セクション**は、メモリーの領域を表します。PSECT には、メモリーの一部の名前、長さ、使用目的と使用許可を表す属性セットがあります。たとえば、宣言によってはメモリーの一部への書き込みを禁止するように設定できます。

リンカーがオブジェクト・ファイルからイメージを構築する場合、リンカーは同様の属性を保持するプログラム・セクションを組み合わせてイメージ・セクションを構成します。**イメージ・セクション**は、システム・メモリーの一部のサイズと属性を指定します。イメージ・アクティベータは、この情報を使用して物理メモリー・ページの特性を決定します。たとえば、前の項で紹介したプログラム・セクションは nowrite イメージ・セクションの要素になり、そのイメージ・セクションは書き込み禁止の物理メモリーの一部にマップされます。

7.3.1 項 (7-6) では、共有可能イメージとそれにリンクする実行可能イメージがデータベース接続とトランザクションを共有しない場合、または共有可能イメージは SQL モジュールを含むが実行可能イメージは含まない場合に共有可能イメージを作成する方法を説明します。いずれのイメージも CONNECT 修飾子を指定してコンパイルされたモジュールは含みません。

7.3.2 項 (7-7) では、共有可能イメージとそれにリンクする実行可能イメージがデータベース接続とトランザクションを共有する場合、または両方のイメージが接続を使用する (SQL モジュールが CONNECT 修飾子を指定してコンパイルされた) 場合に共有可能イメージを作成する方法を説明します。

実行可能イメージと共有可能イメージの両方が SQL モジュールを含み、接続を使用するがデータベースとトランザクションの処理を共有しない場合は、イメージを作成しないでください。この組合せはサポートされないので、動作が予測不可能になる場合があります。両方のイメージがデータベースとトランザクションの処理を共有する場合は問題ありません。◆

7.3.1 データベース接続を共有しない実行可能イメージと共有可能イメージの作成

OpenVMS
VAX ≡≡≡
OpenVMS
Alpha ≡≡≡

この項では、データベース接続を共有しないイメージを作成する方法を説明します。これは、SQL モジュールを含む共有可能イメージを作成するための最も単純な方法です。この種のイメージを作成するのに必要なのは、共有可能イメージのプログラム・セクションを適切に定義することだけです。例 7-2 (7-6) に適切な定義を示します。

例 7-2 共有可能イメージのリンクと作成

```
$ LINK /SHAREABLE=MYSHARE.EXE MYVECTORS, MYSHARE, SQL$USER/LIB, SYS$INPUT/OPTIONS
PSECT_ATTR=RDB$DBHANDLE, NOSHR, LCL
PSECT_ATTR=MYALIAS, NOSHR, LCL
```

```

PSECT_ATTR=RDB$MESSAGE_VECTOR,      NOSHR,LCL
PSECT_ATTR=RDB$TRANSACTION_HANDLE,  NOSHR,LCL
PSECT_ATTR=SQLCA,                    NOSHR,LCL
PSECT_ATTR=SQLDA,                    NOSHR,LCL
PSECT_ATTR=SQL$CALLER_PC,            NOSHR,LCL
$

```

プログラム内で宣言された各エイリアスは、NOSHR,LCL に設定されている必要があります。NOSHR とは、これらのデータ構造が共有されないこと、つまりプログラムで共有可能イメージとは異なるメモリーの一部を使用することを意味します。共有可能イメージと実行可能イメージの両方がデータベースに接続してトランザクションを開始すると、2つのデータベース接続と2つの別々のトランザクション（競合する可能性の高い）が保持されます。

一方のトランザクションが他方とのロック競合に直面すると、デッドロック・エラーが発生します。◆

7.3.2 データベース接続を共有する実行可能イメージと共有可能イメージの作成

OpenVMS VAX  OpenVMS Alpha 

両方のイメージがデータベース接続を共有する場合は、実行可能イメージによって共有可能イメージ内の SQL データ構造に書き込む必要があります。この作業が正しく実行されるためには、書き込み先のデータ構造はそのイメージ内の固定のオフセットに存在する必要があるため、イメージが後で再リンクされても移動できません。共有可能イメージ内の SQL データ構造の配置は、概念的に共有可能イメージの転送ベクトルの配置に似ています（転送ベクトルの詳細は、OpenVMS Linker のドキュメントを参照してください）。

この項では、リンカーに対して該当する SQL プログラム・セクション（PSECT）を共有可能イメージ内に適切に配置するように指示する方法と、エイリアスを初期化してモジュールがエイリアスを共有できるようにする方法を説明します。

複数イメージ間でエイリアスを共有する場合は、共有エイリアスごとに1つ、しかも唯一の定義を保持する必要があります。共有エイリアスを定義するには、次のように DECLARE ALIAS 文で GLOBAL キーワードを指定します。

```
DECLARE MYALIAS GLOBAL ALIAS FOR FILENAME mf_personnel
```

他のモジュールでは、定義済みのエイリアスを繰り返し参照できますがキーワード、EXTERNAL キーワードを使用してエイリアスを宣言する必要があります。

コマンド・ライン修飾子 NOEXTERNAL_GLOBAL か SQLOPTIONS=NOEXTERNAL_GLOBAL を指定した DECLARE ALIAS 文の定義を含むモジュールをコンパイルします。上記の修飾子は、GLOBAL として定義されたエイリアスはグローバルとして処理され EXTERNAL として定義されたエイリアスは外部として処理されることを示します。

このモジュールをリンクするイメージは、このエイリアスを定義する他のイメージにはリンクできません。

たとえば、共有可能イメージ `myshare.exe` を作成するとします。`myshare.sc` ファイルは、SQL ルーチンと GLOBAL キーワードを使用して MYALIAS を定義する DECLARE ALIAS 文を含む SQL プリコンパイル・プログラムです。NOEXTERNAL_GLOBAL 修飾子を使用して SQL モジュールをコンパイルします。得られたオブジェクト・モジュールが `myshare.obj` です。◆

OpenVMS VAX では、`myvectors.obj` のように転送ベクトルを含むオブジェクト・モジュールを作成する必要があります。それから、7-3 項 (7-8) に示すコマンドを使用して共有可能イメージを作成してください。

例 7-3 OpenVMS VAX 上のハンドルを共有する共有可能イメージのリンク

```
$ LINK/SHAREABLE=myshare.exe MYSHARE, SQL$USER/LIB, SYS$INPUT/OPTIONS
CLUSTER=TRANSFER_VECTOR, , MYVECTORS
CLUSTER=SQL_PSECTS
COLLECT=SQL_PSECTS, RDB$DBHANDLE, MYALIAS, RDB$MESSAGE_VECTOR, -
    RDB$TRANSACTION_HANDLE, SQL$CALLER_PC, SQL$TRANSACTION_PTR, -
    SQLCA, SQLDA
PSECT_ATTR=RDB$DBHANDLE,          NOSHR, GBL
PSECT_ATTR=MYALIAS,              NOSHR, GBL
PSECT_ATTR=RDB$MESSAGE_VECTOR,   NOSHR, GBL
PSECT_ATTR=RDB$TRANSACTION_HANDLE, NOSHR, GBL
PSECT_ATTR=SQL$CALLER_PC,        NOSHR, GBL
PSECT_ATTR=SQL$TRANSACTION_PTR,  NOSHR, GBL
PSECT_ATTR=SQLCA,                NOSHR, GBL
PSECT_ATTR=SQLDA,                NOSHR, GBL
$
```

OpenVMS Alpha では、例 7-4 (7-8) に示すコマンドを使用して共有可能イメージを作成してください。このコマンドには、シンボル・ベクトルが含まれます。

例 7-4 OpenVMS Alpha 上のハンドルを共有する共有可能イメージのリンク

```
$ LINK/SHAREABLE=myshare.exe MYSHARE, SQL$USER/LIB, SYS$INPUT/OPTIONS
CLUSTER=SQL_PSECTS
COLLECT=SQL_PSECTS, RDB$DBHANDLE, MYALIAS, RDB$MESSAGE_VECTOR, -
    RDB$TRANSACTION_HANDLE, SQL$CALLER_PC, SQL$TRANSACTION_PTR, -
    SQLCA, SQLDA
PSECT_ATTR=RDB$DBHANDLE,          NOSHR, GBL
PSECT_ATTR=MYALIAS,              NOSHR, GBL
PSECT_ATTR=RDB$MESSAGE_VECTOR,   NOSHR, GBL
PSECT_ATTR=RDB$TRANSACTION_HANDLE, NOSHR, GBL
PSECT_ATTR=SQL$CALLER_PC,        NOSHR, GBL
PSECT_ATTR=SQL$TRANSACTION_PTR.  NOSHR, GBL
PSECT_ATTR=SQLCA,                NOSHR, GBL
```

OpenVMS
VAX

OpenVMS
VAX

```
PSECT_ATTR=SQLDA,                                NOSHR,GBL
SYMBOL_VECTOR = (                                -
    RDB$DBHANDLE=PSECT,                            -
    MYALIAS=PSECT,                                  -
    RDB$MESSAGE_VECTOR=PSECT,                      -
    RDB$TRANSACTION_HANDLE=PSECT,                  -
    SQLCA=PSECT,                                    -
    SQLDA=PSECT,                                    -
    MYSHAREDPROC=PROCEDURE                          -
```



OpenVMS
VAX ≡≡≡

OpenVMS
Alpha ≡≡≡

共有可能イメージを再リンクし、データ構造を移動する場合は、変更する共有可能イメージにリンクする実行可能イメージと共有可能イメージを再リンクする必要があります。
SQL_PSECTS クラスタ内で収集したプログラム・セクションの長さが変わらないように注意してください。たとえば、プログラムに SQLDA はあるが SQLCA は使用しない場合に後で追加すると、SQLDA プログラム・セクションのオフセットが変わります。同様に、共有可能イメージに別のデータベース・エイリアスを追加する場合は、実行可能イメージを再リンクする必要があります。

同様に、転送ベクトルまたはシンボル・ベクトルのプログラム・セクションの長さが変わる場合も、実行可能イメージを再リンクしてください。転送ベクトルまたはシンボル・ベクトルのエントリを追加してもすべての実行可能イメージを再リンクしなくても済むように、今後の使用に備えて転送ベクトルまたはシンボル・ベクトルの領域を割り当てておくことをお勧めします。

データベース・エイリアスを使用する場合、エイリアス名は SQL_PSECT 内で収集する必要があります。また、プログラム・セクション属性を NOSHR GBL に設定する必要もあります。



OpenVMS
VAX ≡≡≡

たとえば、エイリアス ANOTHER_DB でデータベースを宣言する場合、
COLLECT=SQL_PSECTS コマンドに ANOTHER_DB を追加し、リンカー・オプションに次の行を追加する必要があります。

```
PSECT_ATTR=ANOTHER_DB,                                NOSHR,GBL ◆
```

OpenVMS
VAX ≡≡≡

OpenVMS Alpha では、さらにシンボル・ベクトルに次の行を追加します。

```
ANOTHER_DB=PSECT ◆
```

共有可能イメージを作成してから、共有可能イメージを使用する実行可能イメージを作成してください。実行可能イメージが適切に機能するためには、次の作業が必要です。

- 共有エイリアスを参照する場合は、エイリアスを EXTERNAL で宣言します。
ただし、共有エイリアスとは共有可能イメージ内で GLOBAL 宣言したエイリアスです。前述のように、共有エイリアスごとに唯一の定義が可能です。ただし、共有エイリアスは何度でも参照できますが、参照を EXTERNAL として宣言する必要があります。
- NOEXTERNAL_GLOBAL オプションを使用して SQL モジュールをコンパイルします。

7.4 共有可能イメージのインストール

- 共有可能イメージ内のプログラム・セクション属性に適したプログラム・セクション属性を定義します。

たとえば、次の例に示すように、埋込みの SQL と COBOL プログラムをコンパイルします。

```
$ SQLPRE
INPUT_FILE> mymain.sco /COB /SQLOPTIONS=(NOEXTERNAL_GLOBAL)
```

それから、例 7-5 (7-10) に示すコマンドを使用して mymain をリンクします。

例 7-5 共有可能イメージを使用する実行可能イメージのリンク

```
$ LINK mymain, SQL$USER/LIB, SYS$INPUT/OPTIONS
PSECT_ATTR=RDB$DBHANDLE,          NOSHR,GBL
PSECT_ATTR=MYALIAS,                NOSHR,GBL
PSECT_ATTR=RDB$MESSAGE_VECTOR,    NOSHR,GBL
PSECT_ATTR=RDB$TRANSACTION_HANDLE, NOSHR,GBL
PSECT_ATTR=SQLCA,                  NOSHR,GBL
PSECT_ATTR=SQLDA,                  NOSHR,GBL
MYSHARE/SHARE
$
```



7.4 共有可能イメージのインストール

OpenVMS VAX  OpenVMS Alpha 

共有可能イメージを複数のプロセスで使用するには、多くの場合、共有可能イメージをインストールしてシステムに認識させる必要があります。すべてのプロセスがメモリー内にある共有可能イメージの同一のコピーを共有するには、OpenVMS Install ユーティリティの INSTALL コマンドに /SHARE を指定して共有可能イメージのファイル指定を変更します。

共有可能イメージのインストールについては、OpenVMS Install ユーティリティのドキュメントを参照してください。Oracle Rdb がインストールする共有イメージと、システム・マネージャが手動でインストールしなければならない共有イメージについては、『Oracle Rdb7 Installation and Configuration Guide』を参照してください。◆

7.5 CompaqTru64 UNIX 上でのモジュールのリンク

Compaq Tru64 UNIX

CompaqTru64 UNIX では、ホスト言語コンパイラで処理することでオブジェクト・ファイルをリンクできます。SQL 文を含むオブジェクト・モジュールをリンクするには、次のように指定します。

- SQL モジュール言語オブジェクト・ファイルやホスト言語オブジェクト・ファイルまたは SQL プリコンパイラ・オブジェクト・ファイルなど、プログラムで必要なすべてのオブジェクト・ファイル
- SQL ディレクトリ・ツリーにある SQL ライブラリ

リンクを簡単にするため、SQL ライブラリに変換されるグローバル・シンボル、SQLLIBS を定義します。シンボルを次のように定義します。

```
-lrdbsql -lrdbsshr -lcosi -lots
```

たとえば、Bourne シェルで次のコマンドを入力します。

```
$ SQLLIBS='-lrdbsql -lrdbsshr -lcosi -lots'
$ export SQLLIBS
```

SQL モジュール言語プログラムをリンクする場合、最初に SQL モジュール・プロセッサで SQL モジュールを処理する必要があります。そして、1 回の手順で、ホスト言語モジュールをコンパイルし、ホスト言語モジュールのオブジェクト・ファイルと SQL モジュールを SQL ライブラリとリンクできます。

次の例は、SQL モジュール・オブジェクト・ファイル test_mod.o と C 言語プログラム test_h.c をリンクする方法を示しています。

```
$ cc -o test_h test_h.c test_mod.o ${SQLLIBS}
```

SQL モジュール言語プログラムのコンパイルの詳細は、5.2 項 (5-3) を参照してください。

プリコンパイル SQL プログラムをリンクする場合、SQL プリコンパイラで最初にプログラムを処理する必要があります。そして、ホスト言語コンパイラを使用して、オブジェクト・ファイルと SQL ライブラリをリンクします。

次の例は、C 言語で記述されたプリコンパイル SQL プログラムであるオブジェクト・ファイル test.o をリンクする方法を示しています。

```
$ cc -o test test.o ${SQLLIBS}
```

SQL プリコンパイル・プログラムのコンパイルの詳細は、6.3 項 (6-6) を参照してください。

Oracle Rdb は、/usr/lib/sqllibs.make ファイルを makefile にインクルードして、SQLLIBS を定義できます。sqllibs.make ファイルには、必要な 4 つの共有オブジェクト・ライブラリが含まれていますが、アプリケーションが使用する追加シンボルを解決するためにライブラリを追加する必要がある可能性があります。◆

7.5.1 複数のモジュールでのアプリケーションの構築

Compaq Tru64 UNIX

SQL が作成したオブジェクト・ファイル内に SQL が生成したエイリアスなどの内部変数がいくつかあります。これらの変数は文が正しく実行されることを保証するため、正しく初期化される必要があります。デフォルトでは、SQL はこれらの変数を初期化するコードを生成します。

アプリケーション内の 2 つ以上の SQL オブジェクト・ファイルをリンクする場合、CompaqTru64 UNIX リンカーは、変数を初期化する唯一のモジュールと、その変数への参照を生成するその他すべてのモジュールが必要です。

2つ以上のモジュールがこれらの変数を初期化する場合、CompaqTru64 UNIX のリンカーはシンボルが重複して定義されていることを示すエラーを生成します。これらの変数を初期化するモジュールがなければ、SQL 文は予測できない動作をするか、Oracle Rdb の正しくないハンドルに対するエラーを出すか、またはその両方です。

これらのオブジェクトのそれぞれに対して、これらの変数に対する定義者として動作する1つのモジュールを選択する必要があります。たとえば、各共有エイリアスに対してただ1つの定義を持つ必要があります。共有エイリアスを定義するには、次の例で示すように DECLARE ALIAS 文で GLOBAL キーワードを使用します。

```
DECLARE MYALIAS GLOBAL ALIAS FOR FILENAME mf_personnel
```

その他のモジュールでは、定義したエイリアスに対して多くの参照を持つことができますが、EXTERNAL キーワードを使用したエイリアスを宣言する必要があります。

すべてのモジュールを -noextern または -s'-noextern' オプション（デフォルト）でコンパイルする

このオプションは、これらのエイリアスで GLOBAL として定義されたものは GLOBAL として処理され、EXTERNAL として定義されたものは外部として処理されることを指定します。

◆

7.6 オブジェクト・ライブラリおよびアーカイブへのプリコンパイル SQL モジュールの挿入

SQL プリコンパイラ・オブジェクト・ファイルを OpenVMS オブジェクト・ライブラリまたは CompaqTru64 UNIX アーカイブに挿入できます。SQL プリコンパイラ・オブジェクト・ファイルには、2つの個別モジュールがあることに注意してください。1つのモジュールは、ホスト言語プログラムであり、もう1つは、生成された SQL モジュールです。生成された SQL モジュールは、RDB\$<モジュール>と呼ばれています。ここで、<モジュール>は、SQL DECLARE MODULE 文で宣言されている名前か、ホスト・モジュールの名前（デフォルト）です。

7.7 プログラムの実行

OpenVMS OpenVMS
VAX ≡≡≡ Alpha ≡≡≡

OpenVMS 上でプログラムを実行するには、DCL プロンプトで、RUN コマンドとプログラム名を入力します。

```
$ RUN MYPROGRAM ◆
```

Compaq Tru64 UNIX
≡≡≡≡≡

CompaqTru64 UNIX 上でプログラムを実行するには、コマンドライン・プロンプトでプログラム名を入力します。

```
$ myprogram ◆
```


入力と出力をテストするためにプログラムを実行する場合、プログラムが正しく処理できない、またはまったく処理できないエラー条件が発生することがあります。プログラムでのエラー処理の詳細は、10 章 (10-1) を参照してください。

参考: エラー・メッセージについての情報は、『Oracle Rdb7 SQL Reference Manual』を参照してください。エラー・メッセージの付録では、リレーショナル・データベース機能から発行されるメッセージに対する解説とユーザーの動作を記述したオンライン・ファイルのロケーションが記載されています。

7.8 SQL 文とプログラム・コードのデバッグ

プログラムをデバッグするには、多くの場合次に示す SQL 文とホスト言語コードを検査する必要があります。

- デバッグする SQL コード
対話型 SQL を使用して、SQL モジュール・ファイル内またはプログラムに埋め込まれた SQL 文内の構文エラーを探し、排除します。
- ホスト言語コードのデバッグ
問題が簡単に解決できなければ、デバッガ・ユーティリティを使用して、実行時のプログラムの実行状況を監視する方法を取ることができます。

多くのデバッガ・ユーティリティで、次のことが実行できます。

- プログラムを一度に 1 行ずつ実行する
- 文とデータ値の検査と変更
- 指定した位置でプログラムの実行を停止
- 指定した点でのメッセージを表示

OpenVMS VAX ≡≡≡

OpenVMS Alpha ≡≡≡

OpenVMS デバッガを使用するには、/DEBUG 修飾子を含めて、プログラム・ファイルのコンパイルとリンクを行います。

FORTRAN、Pascal および Ada プログラムの場合、/DEBUG 修飾子とともに /NOOPTIMIZE コンパイル修飾子を使用します。デフォルトでは、FORTRAN、Pascal および Ada のプログラムには最適化が含まれています。コードを最適化すると、プログラム・エラーの検出が困難になります。

共有イメージ内のグローバル・ルーチンをデバッグする場合は、実行可能イメージ内の同様のルーチンをデバッグする場合には必要でないコマンド（たとえば、SET IMAGE）が必要です。共有イメージ内でルーチンをデバッグする場合、OpenVMS デバッガ・ユーティリティ・ドキュメントのこれらのコマンドについての説明を必ずご覧ください。◆

注意: プリコンパイル・プログラム内に SQL 文を含める場合、ホスト言語ファイルではなく、SQL プリコンパイラを通す前のソース・ファイルに対して、ホスト言語文への変更を含むすべての変更を入力する必要があります。

パラメータの宣言と使用方法

この章では、列値をプログラムへ、またはプログラムから列値を転送する場合の、データ型の互換性の確認について説明します。プログラムに関する説明の大部分は、コンパイル時に使用する SQL プロセッサには関係しません。

この章では、次の項目について説明します。

- モジュール・プロセッサと SQL プリコンパイラがパラメータを処理する方法の違い
- パラメータの説明に使用される用語
- 関数のパラメータおよびオプションの宣言
- パラメータのデータ型の宣言
- ソースの外部からのパラメータ宣言のコピー
- 主要なパラメータの宣言と使用方法
- インジケータ・パラメータの宣言と使用方法
- パラメータを宣言して使用する場合によく発生する間違いの防止
- 特定のホスト言語でのパラメータの宣言と使用方法

ストアド・ルーチンに対するパラメータの宣言の詳細は、13 章 (13-1) を参照してください。

8.1 パラメータの宣言と使用方法の概要

パラメータの宣言と使用方法は、SQL プリコンパイラと SQL モジュール・プロセッサの間では異なります。

- SQL プリコンパイラにおけるホスト言語変数の指定
SQL プリコンパイラを使用する場合、ホスト言語ソース・ファイルに SQL 文を埋め込み、ホスト言語変数への直接参照を SQL 文に含めます。コンパイル時に、SQL プリコンパイラは、ホスト言語ソース・ファイル内の SQL 文をプロセッサが実装したプロシージャへのコールと置き換えます。プログラマが最初に SQL 文に指定したホスト言語変数は、ホスト言語コールのパラメータとして指定されます。
- SQL モジュール・プロセッサでモジュール・パラメータを指定
SQL モジュール・プロセッサで作業をする場合、SQL 操作を実行するプロシージャを含むソース・ファイルを作成します。これらのプロシージャには、SQL 文とその文が参照するパラメータの宣言が含まれています。そして、ホスト言語ソース・ファイルで、追加のパラメーター式を宣言し、これらを外部プロシージャへのコールで指定します。図 4-1 (4-9) は、SQL 文を指定して宣言したホスト言語構文とプロシージャ・パラメータを使用して宣言した実パラメータ間の対応を説明しています。

SQL プリコンパイラ、SQL モジュール・プロセッサのどちらを使用しても、DML 文内の値式にパラメータを使用できますが、データ定義文では使用できません。

ホスト言語ソース・ファイルで宣言するパラメータに対するオプションには、次のものがあります。

- パラメータの機能に依存
- プログラムで SQL 文を指定して処理する方法によって影響を受ける
- SQL がホスト言語でサポートしたパラメータ宣言のタイプによって制限される

参考: パラメータ宣言の詳細は、『Oracle Rdb7 SQL Reference Manual』の言語と構文要素の章を参照してください。この章には、メイン・パラメータおよびインジケータ・パラメータ、構造およびインジケータ配列、データ型、サポートしているホスト言語パラメータ宣言およびデータ変換のトピックについての項があります。

さらに、この章では、SQL キーワードおよびユーザー定義名の入力規則に対する言語と構文要素を説明しています。大文字と小文字、ハイフン (-) およびアンダースコア (_) の解釈についての情報は、プログラム環境では特に重要です。

8.2 用語について

この章では、パラメータの効果的な宣言と使用方法を理解するために必要な用語を説明しています。これらの用語には、次が含まれます。

- **変数とパラメータ**
SQL プリコンパイラ・ユーザーと SQL モジュール・プロセッサ・ユーザーの両方に対処する必要があるため、変数という用語よりもパラメータという用語のほうが、SQL 文が参照するプログラムでの宣言の説明に最もよく使用されます。パラメータは、コールおよびプロシージャのコンテキストで適切な用語です。
SQL モジュール・プロセッサを使用している場合、パラメータは SQL モジュール言語プロシージャで指定した仮パラメータを参照します。
SQL プリコンパイラを使用している場合、パラメータは、埋込み SQL 文に含めた宣言されたホスト言語変数を参照します。SQL プリコンパイラを使用する場合、プロシージャとして SQL 操作を直接実装していないので、用語「フィールド」または「変数」を用語「パラメータ」に置き換えたほうがよいでしょう。
用語「変数」は、複合文で見られるようなローカル変数を参照します。
- **構造体とレコード**
用語「**構造体**」は、その他のフィールドを含むパラメータを参照します。(1 次元の配列を作る項目の繰返しとは異なり、構造体内の基本要素は、同様に定義される必要はありません。) ホスト言語によっては、用語「レコード」と用語「構造体」は、同じ宣言を参照します。FORTRAN など、その他の言語では、構造体とレコードを別に宣言します。使用している言語が、構造体とは別にレコードの宣言を行うならば、プリコンパイル SQL 文でレコード名を指定する必要があります。この場合、レコード宣言によってリストされるか、暗示されるフィールドは、SQL への構造体を構成します。

8.3 パラメータ機能と宣言オプション

パラメータの機能によって、パラメータの宣言にいくつか異なるオプションがあります。次の一覧では、SQL 文で使用されるパラメータ宣言のオプションについて説明します。特に記載されていなければ、SQL プリコンパイラまたは SQL モジュール・プロセッサで記載されているオプションを使用できます。

- **メイン・パラメータ**は、列の選択、操作、保存に関する値を含むパラメータであり、次の方法で宣言できます。
 - 直接的に宣言。プログラムのホスト言語文を使用。
 - 間接的に宣言。SQL INCLUDE 文を使用して、リポジトリまたはテキスト・ファイルからホスト言語ソース・ファイルにパラメータ宣言をコピー (SQL プリコンパイラのみ)。
 - 間接的に宣言。SQL モジュール・プロシージャで、FROM path-name 句を使用して、リポジトリからレコード定義をコピー。

- 間接的に宣言。ホスト言語 INCLUDE 文または COPY 文を使用して、別のソースからホスト言語ソース・ファイルにパラメータ定義をコピー。
 - **インジケータ・パラメータとインジケータ配列**は個別に宣言されます。いずれも NULL 値を取り扱うパラメータであり、次の方法で宣言できます。
 - 直接的に宣言。ホスト言語ソース・ファイルに宣言を入力。
 - 間接的に宣言。SQL INCLUDE 文を使用して、インジケータ・パラメータ用の一組の宣言をリポジトリまたはテキスト・ファイルからホスト言語ソース・ファイルにコピー（SQL プリコンパイラのみ）。
 - 間接的に宣言。ホスト言語 COPY 文または INCLUDE 文を使用して、別のソースからホスト言語ソース・ファイルにインジケータ・パラメータ用の一組の宣言をコピー。
 - SQL 文の実行状態を保存するパラメータは、次の方法で、宣言できます。
 - 直接的に宣言。ホスト言語ソース・ファイルに宣言を入力して、SQLSTATE パラメータを宣言。
SQLSTATE パラメータの説明は、『Oracle Rdb7 SQL Reference Manual』の 10 章 (10-1) および付録の SQLSTATE を参照してください。
 - 直接的に宣言。ホスト言語ソース・ファイルに宣言を入力して、SQLCODE パラメータ（FORTRAN プログラムの SQLCOD）と使用したい他のフィールドを宣言。
 - 間接的に宣言。パラメータ宣言が有効であるホスト言語ソース・ファイルのセクションで INCLUDE SQLCA 文を使用（SQL プリコンパイラのみ）。
この文には、一連のパラメータ宣言が含まれています。定義されたパラメータには、SQLCODE およびエラー処理プロシージャで有効なパラメータがあります。
 - 間接的に宣言。ホスト言語 COPY 文または INCLUDE 文を使用して、別のソースからホスト言語ソース・ファイルに宣言をコピー（SQL モジュール・プロセッサのみ）。
- 『Oracle Rdb7 SQL Reference Manual』の SQL Communications Area (SQLCA) の付録には、SQLCODE と、SQLCA を構成する他のフィールドの言語固有の宣言が含まれています。この付録は、INCLUDE SQLCA 文を使用すると SQL が自動的に作成する宣言を示しています。ホスト言語文を使用して、エラー処理に必要なパラメータを定義すると、この宣言は有効な参照を提供します。
エラー処理に関連したパラメータの宣言方法と使用方法の詳細は、8.6 項 (8-8) および 10 章 (10-1) を参照してください。
- 動的に実行される SQL 文の入力値と出力値の数とデータ型を記述するパラメータは、次に示す方法で宣言できます。
 - 直接的に宣言。ホスト言語ソース・ファイルに宣言を入力。

- 間接的に宣言。パラメータ宣言が有効であるホスト言語ソース・ファイルのセクションで INCLUDE SQLCA 文または INCLUDE SQLDA2 を使用（SQL プリコンパイラのみ）。
- 間接的に宣言。ホスト言語の COPY 文または INCLUDE 文を使用して、別のソースからホスト言語ソース・ファイルに宣言をコピー（SQL モジュール・プロセッサのみ）。
- 間接的に宣言。C プログラムで SQL_SQLDA2.H ヘッダー・ファイルを使用。ヘッダー・ファイルの使用の詳細は、11.3.2 項 (11-10) を参照してください。

これらのパラメータの間には、キーワードを表すフィールドおよび SQL 文のテーブルと列名がありますが、実行時までわかりません。

参考: 11 章 (11-1) では、動的 SQL を処理するパラメータの宣言を行うときを説明し、例を示しています。さらに、『Oracle Rdb7 SQL Reference Manual』には、INCLUDE SQLDA 文と INCLUDE SQLDA2 文を使用できない場合に必要のパラメータについて説明する SQL Descriptor Areas (SQLDA および SQLDA2) に関する付録があります。

8.6 項 (8-8) は、SQL INCLUDE 文、および等価なホスト言語文の使用方法の詳細を示します。

SQL 文で使用するパラメータ宣言に対するオプションは、使用するホスト言語がサポートしている SQL の宣言の型に限定されます。たとえば、ホスト言語プログラムでは、SQL 文内のパラメータ名で指定した場所にのみ、構造体（レコード）や配列要素を指定できます。さらに、SQL プリコンパイラは、ホスト言語に固有のすべての構造要素をサポートしているわけではありません（たとえば、FORTRAN では暗黙的な宣言）。

パラメータを宣言する場合、使用するデータ型が SQL データ型と互換性があることを確認してください。

8.4 パラメータのデータ型の宣言

SQL プリコンパイラまたは SQL モジュール・プロセッサを使用すると、SQL 文のパラメータ宣言は、次の 2 つの条件が真であれば、無効です。

- パラメータ宣言のデータ型が SQL データ型と等価でない
- SQL はパラメータの値を変換できない
 - 列に対して定義したデータ型へ（データを列へ転送、または列からデータを転送）
 - ホスト・パラメータまたは関連するパラメータが比較する値式に対するデータ型へ（条件を評価するため）

データ型サポートに関する問題は、言語またはデータベース・システムが一般的な型（たとえば、バック 10 進数、日付、浮動小数、テキスト、固定小数点バイナリ）の記憶形式をサポートしているかどうかだけではありません。数値データの場合、符号付きデータや符号なしデータ、スケール計算、サイズおよび倍精度浮動小数点値（D-Floating や G-Floating）の解釈などのサポートに関連する問題が考えられます。

場合によっては、入力操作や出力操作に関連する中間ホスト言語変数を定義する必要があります。

値を UPDATE 文や INSERT 文で使用する前にプログラムによる中間処理が必要な場合には、たとえば端末やデータ・ファイルからの値の入力を含む変数などを追加する必要がある場合があります。逆に、端末やデータ・ファイルに出力する前にホスト言語によって処理する必要がある値が FETCH 文や SELECT 文によって返される場合、変数を追加する必要がある場合もあります。一時プログラム記憶域に対する追加変数を最も必要とする値の種類には、日付、時間、タイムスタンプ、間隔（ソース列またはターゲット列が日付時間データ型を使用して定義されていると仮定）およびテキスト文字列（通常文字列をすべて大文字に変換するか、大文字と小文字混合のどちらか）があります。

ホスト言語に対してプリコンパイラをサポートするの主な利点の 1 つは、プリコンパイラがエラーまたは警告を生成することで、SQL にとって無効な宣言を警告することです。したがって、プリコンパイラは、プログラムとデータベース間でデータを転送する場合に、サポートされていない、または適切でない宣言によって起こる多くの実行時エラーと予期せぬ結果を防止できます。しかし、プリコンパイル時には、プログラムとデータベース間でデータを転送する場合に問題であると考えられる、すべての実行時の結果を検出することはできません。

Oracle Rdb は、文字値および数値の切捨ては、状況によっては適切な操作であると考えます。また、Oracle Rdb は、異なるデータ型の間における変換は、場合によってはプログラムにとって便利であり、有効であるので、適切であると考えます。したがって、誤ったデータ型または誤ったサイズでパラメータを宣言することは可能ですが、実行するまで問題は発見できません。

さらに、SQL プリコンパイラは、内部プログラム処理のみに対して使用されるパラメータの宣言をチェックしません（パラメータは SQL 文では指定されません）。これらのパラメータは、SQL プリコンパイラから処理されたソース・コードを受け取った後、ホスト言語コンパイラによってのみチェックされます。

注意：前の節は、プリコンパイル・プログラム内のすべてのパラメータに対して適用されるものではありません。任意のパラメータを倍精度浮動小数点として宣言する場合（これらのパラメータをプリコンパイル SQL 文で使わない場合でも）、『Oracle Rdb7 SQL Reference Manual』の G-Floating 数および D-Floating 数に対する SQL プリコンパイラ修飾子についての情報を参照してください。

SQL モジュール・プロセッサは、ホスト言語モジュールのパラメータの宣言をまったくチェックしません。したがって、SQL 言語モジュール内のプロシージャをコールするホスト言語モジュールを作成した場合、SQL プロシージャへのコールで使用するときにホスト言語ファイルのパラメータが正しく宣言されているかどうかの確認はユーザー自身の責任で行ってください。特に、ホスト言語ソース・ファイルの実パラメータの記憶フォーマットと仕様の順序が、対応する SQL ソース・ファイルのプロシージャ・パラメータの記憶フォーマットと仕様の順序に正確に一致しているかどうかは、ユーザー自身の責任で確認してください。また、コールされたモジュールおよびコールしたモジュールで、対応する値の引渡しメカニズム（参照または記述子による）が一致しているかどうかに、注意する必要があります。

注意： SQL モジュールでは、ホスト言語の記憶フォーマットが列の記憶フォーマットに一致できなかった場合、関連するプロシージャ・パラメータの記憶フォーマットは、列ではなくホスト言語パラメータの記憶フォーマットに一致している必要があります。

前の規則には、例外が 1 つあります。バイナリ書式の DATE VMS 値をホスト言語プログラムとデータベースの間で転送したい場合には、ほとんどのホスト言語が DATE データ型をサポートしていないにもかかわらず、関連するプロシージャ・パラメータを常に DATE データ型として定義します。DATE 値をバイナリ解釈する場合、SQL は、8 バイトのホスト言語パラメータ（4 ワード値を保存）を予測し、ホスト言語プログラムでそのパラメータに対して定義したデータ型または引渡しメカニズムに固有ではないと考えます。その一方で、文字列書式からの DATE 値の変換、または DATE 値から文字列書式への変換は、対象のホスト言語パラメータのプロシージャ・パラメータに関する一般的な規則に従います。DATE 値を文字列として解釈する場合、ホスト言語パラメータとプロシージャ・パラメータは、いずれも 8 バイトから 16 バイトの文字フィールドとして定義されます。

日付時間データ型の情報は、9 章 (9-1) を参照してください。

SQL モジュール言語には、プロシージャおよび実パラメータが引渡しメカニズムに正しく一致しているかどうかを判断するための、実行時 CHECK オプションがあります。ホスト言語プログラムの実パラメータと SQL のプロシージャ・パラメータの間の正しい対応に関する説明は、4.2.4 項 (4-9) を参照してください。

SQL モジュール・プロシージャ内のパラメータの宣言に関する情報は、4.2.2 項 (4-4) から 4.2.9 項 (4-14) を参照してください。

言語に固有の項については、この章で後ほど説明しますが、パラメータの使用方法についてさらに詳しく解説します。しかし、Oracle Rdb が適用するデータ変換規則の詳細は、『Oracle Rdb7 SQL Reference Manual』のみに掲載されていることに注意してください。

8.5 プログラムの外部ソースからのパラメータ宣言のコピー

SQL 文を SQL プリコンパイラで処理するか、SQL モジュール・プロセッサで処理するかによって、プロシージャ宣言で、SQL INCLUDE 文、同様の機能を持つホスト言語文、または FROM path-name 句を使用できます。これらの文および句を使用すると、サイトのプログラマがアプリケーションにコピーできる宣言の標準セットを開発して、使用できます。サイトで一組の標準宣言を維持することで、次の項目を削減できます。

- ソース・ファイル内で宣言が正しく入力されないために起こるプログラマ・エラーの数
- プログラム開発時間
- プログラムが使用された後でテーブル、列またはビュー定義の変更が必要な場合にソース・ファイルを編集する量

8.6 SQL INCLUDE 文の使用法

ホスト言語ソース・ファイルに SQL 文を組み込んだ場合のみ、SQL INCLUDE 文を使用できます。SQL プリコンパイラは、SQL 文で指定したすべてのパラメータの宣言にアクセスする必要があり、ホスト言語の COPY 文または INCLUDE 文を処理しません。

しかし、SQL モジュールで SQL INCLUDE 文を使用できません。ホスト言語プログラム・パラメータの宣言は、ホスト言語ソース・ファイルにコピーする必要があります。SQL プリコンパイラとは異なり、SQL モジュール・プロセッサは、ホスト言語ソース・ファイルを処理しません。これらのファイルを処理するホスト言語コンパイラは、ホスト言語 COPY 文または INCLUDE 文のみを認識します。

SQL INCLUDE 文には、次のバリエーションがあります。

OpenVMS
VAX ≡≡≡

OpenVMS
Alpha ≡≡≡

- SQL INCLUDE FROM DICTIONARY 文は、リポジトリ・ノードからテーブル、ビューまたはレコードの定義をコピーします。◆
- INCLUDE ファイルスベック文は、プログラムによって共有される言語ソース・コードを含むテキスト・ファイルをコピーします。
- INCLUDE SQLCA 文は、エラー処理用にパラメータを宣言します。
- INCLUDE SQLDA 文または INCLUDE SQLDA2 文は、動的 SQL 文で使用されたパラメータを宣言します。

OpenVMS
VAX ≡≡≡

OpenVMS
Alpha ≡≡≡

SQL INCLUDE 文は、OpenVMS ライブラリ・ユーティリティで作成されたファイルをサポートしていません。したがって、埋込み SQL 文で使用されたパラメータ宣言を含む場合、ライブラリ・ファイルからプログラムにコピーできません。◆

参考:『Oracle Rdb7 SQL Reference Manual』の SQL 文の章には、INCLUDE 文の項があります。この項には、文のバリエーションの追加説明と完全なプログラムの例が載っています。

8.6.1 SQL INCLUDE FROM DICTIONARY 文の使用法

OpenVMS
VAX ≡≡≡
Alpha ≡≡≡

SQL プリコンパイル・プログラムの INCLUDE FROM DICTIONARY 文を使用して、次に示す定義をコピーします。

- CREATE DATABASE または INTEGRATE 文によって作成され、維持されたデータベースのリポジトリ・ノードからのテーブルまたはビュー
- データベースに対して作成され、維持されたもの以外のリポジトリ・ノードからのレコード

SQL とともに使用しているホスト言語が、テーブルの列に対して定義されたすべてのデータ型をサポートしているならば、使用しているテーブルまたはビューの列に一致しているプログラム・パラメータを迅速に定義する方法は、データベースを保存するリポジトリ・ノードからの定義を含めることです。

たとえば、従業員データベースに対するディレクトリ WARREN で作成されたりポジトリ・ノードから DEPARTMENTS テーブルへの列定義を含めるには、次の文を使用します。一重引用符 (') でリポジトリ・パス名が囲まれていることを確認してください。

```
EXEC SQL INCLUDE FROM DICTIONARY
      'DISK3:[DEPT32.CDD]WARREN.PERSONNEL.RDB$RELATIONS.DEPARTMENTS'
```

前の例は、定義をコピーするリポジトリ・ノードが CREATE DATABASE 文または INTEGRATE 文によって作成されている場合に、テーブルのパス名の一部として RDB\$RELATIONS を指定したことを説明しています。

ホスト言語がサポートしていないデータ型を含むテーブルの場合、データベース内のデータ型と互換性があるデータ型でリポジトリ・レコード定義を作成し、維持します。INCLUDE FROM DICTIONARY 文を使用して、これらを同様にコピーできます。データベース操作によって生成、保存されないリポジトリ・レコード定義を含めるには、次の文を使用します。一重引用符 (') でリポジトリ・パス名が囲まれていることを確認してください。

```
EXEC SQL INCLUDE FROM DICTIONARY
      'DISK3:[DEPT32.CDD]WARREN.PERSONNEL.DEPARTMENTS'
```

C 言語に対して SQL プリコンパイラを使用する場合、SQL はレコード内の文字列フィールドを NULL 終端文字列として変換します。SQL が C 言語の文字列を処理する方法については、8.12.2 項 (8-27) を参照してください。◆

8.6.2 INCLUDE SQLCA 文の使用法

INCLUDE SQLCA 文には、SQL プリコンパイラでコード化されている宣言が含まれています。この文は、エラー処理用のパラメータを宣言します。SQLCODE および RDB\$MESSAGE_VECTOR 配列の宣言に加えて、INCLUDE SQLCA 文は SQLERRD 配列を宣言します。

次の例で示すように SQLCA を宣言します。

```
EXEC SQL INCLUDE SQLCA
```

SQL のプリコンパイル C プログラムで、INCLUDE SQLCA 文を使用する複数のモジュールが存在すると仮定すると、その中の 1 つを除いて EXTERNAL キーワードを追加することができます。

INCLUDE SQLCA 文は、広域的に可視的な SQLCA 構造体のインスタンスを定義します。EXTERNAL キーワードを使用すると、SQLCA 構造体に対する外部参照を宣言できます。OpenVMS 上では、アプリケーションが複数のイメージで SQLCA を共有する場合、1 つのイメージで SQLCA を定義し、その他すべてのイメージは SQLCA を参照する必要があります。INCLUDE SQLCA EXTERNAL 文を使用して、SQLCA を参照します。CompaqTru64 UNIX では、アプリケーションが複数のイメージを使用するかどうかに関係なく、SQLCA の定義を 1 つ持つ必要があります。

SQLERRD 配列の 2 番目および 3 番目の要素は、プログラマにとって最も興味深いものです。

SQLERRD 配列の 2 番目の要素は、動的な SQL 要素が SELECT 文であるかどうかについての情報を返します。動的 SQL の SQLERRD 配列からの情報を使用するには、PREPARE 文内および EXECUTE 文内の両方で SQLCA を参照する必要があります。

SQLERRD 配列の 3 番目の要素は、各 FETCH、INSERT、UPDATE および DELETE 文を実行した後で、その文によって処理された行の数を含むように更新されます。FETCH 文の場合、カーソル内でフェッチされた行の序数位置で示されます。プログラムの一部として、このパラメータ内の情報を表示するか、また記録することもできます。

行カウント値を使用するつもりでなければ、プログラムに SQLERRD 宣言を含めないでください。この場合、INCLUDE SQLCA 文のかわりに SQLCODE の明示的な宣言に置き換えることができます。

INCLUDE SQLCA 文を指定せずに、RDB\$MESSAGE_VECTOR 配列を参照するつもりであれば、RDB\$MESSAGE_VECTOR 配列も明示的に宣言する必要があります。

INCLUDE SQLCA 文および SQLCODE と RDB\$MESSAGE_VECTOR の宣言方法と使用法の詳細は、10 章 (10-1) を参照してください。

8.6.3 INCLUDE SQLD 文または SQLDA2 文の使用法

INCLUDE SQLDA 文および INCLUDE SQLDA2 文は、SQL プリコンパイラのコードでパラメータを宣言して、動的 SQL で使用します。

次の例で示すように、SQLDA 構造体または SQLDA2 構造体を宣言します。

```
EXEC SQL INCLUDE SQLDA
EXEC SQL INCLUDE SQLDA2
```

SQLDA または SQLDA2 の宣言方法および使用法の詳細は、11.3.2 項 (11-10) を参照してください。

8.6.4 INCLUDE ファイル文の使用法

データベース・アクセス用にプログラムが共有する言語ソース・コードを含むテキスト・ファイルが存在する場合には、INCLUDE file-spec 文を使用してそのコードをコピーできます。テキスト・ファイルには、SQL 文、パラメータの宣言、またはその両方が含まれます。COBOL など、ホスト言語によっては、宣言および実行可能文を別々のプログラム・セクションに分ける必要があります。プログラム言語がこの要件に該当する場合には、宣言内と実行可能文を別々にコピーするため、2 つ以上の INCLUDE 文を指定します。プログラムの 1 つのセクションで、宣言を含むテキスト・ファイル（および非実行可能 SQL 文）をコピーできます。プログラムの別のセクションでは、実行可能 SQL とホスト言語文を含むテキスト・ファイルをコピーできます。

宣言のテキスト・ファイルを含めるには、次に示すような文を使用します。

```
EXEC SQL INCLUDE dcls_for_depts_table.cob
```

8.7 SQL モジュール言語 FROM path-name 句の使用法

OpenVMS
VAX ≡≡≡

OpenVMS
Alpha ≡≡≡

SQL モジュール言語を使用する場合、FROM path-name 句を指定して、リポジトリから SQL モジュールのプロシージャヘレコード定義をコピーできます。FROM path-name 句を使用すると、データベース内のものと互換性があるデータ型を使用して、リポジトリ・レコード定義を作成し、維持できます。これを使用すると、パラメータがテーブルまたはビュー内の列に一致することを確認して、プロシージャに対するパラメータを迅速に定義できます。

次の例は、SQL モジュール・プロシージャ内にレコードを宣言して、リポジトリからレコードをコピーする方法を示します。

```
PROCEDURE store_proc
  (SQLCODE,
   emp_rec RECORD
   FROM cddplus.records.employees END RECORD);
```

SQL モジュールで FROM path-name 句を使用すると、SQL モジュール内で宣言したパラメータが、リポジトリからそのパラメータをコピーするホスト言語プログラム内のパラメータに一致することを簡単に確認できます。8.8 項 (8-12) では、ホスト言語プログラムにおけるホスト言語 COPY 文と INCLUDE 文の使用法を説明します。

注意: SQL モジュール・プロセッサを使用し、かつモジュール言語を C と指定すると、SQL は、文字列フィールドを NULL 終了文字列として変換します。SQL が C 言語で文字列を処理する方法については、8.12.2 項 (8-27) を参照してください。◆

8.8 ホスト言語 COPY 文と INCLUDE 文の使用法

SQL モジュール言語を使用する場合、ホスト言語プログラムでホスト言語 COPY 文または INCLUDE 文を指定すると、SQL モジュール内のプロシージャのコールで使用するパラメータ宣言を追加できます。

ホスト言語ソース・ファイルに埋め込まれる SQL 文で使用するパラメータ宣言を追加するためには、ホスト言語 COPY 文または INCLUDE 文を指定できません。SQL モジュール・プロセッサとは異なり、SQL プリコンパイラは、SQL 文が参照するパラメータの定義にアクセスする必要があります。プリコンパイラは、ホスト言語 COPY 文または INCLUDE 文を認識できません。

ホスト言語 COPY 文または INCLUDE 文の使用法の詳細は、ご使用のホスト言語ドキュメントを参照してください。

8.9 メイン・パラメータの宣言と使用法

プリコンパイルしたプログラムでは、通常の場合、ホスト言語パラメータの名前のかわりに、リテラル（定数）や、対話型 SQL で使用するその他の値式を指定します。プリコンパイルした SQL 文では、すべてのパラメータ名の前にコロン（:）を付けてください。

SQL モジュールでは、SQL 文でホスト言語パラメータを指定するのではなく、プロシージャ・パラメータを指定します。プロシージャ・パラメータのデータ型は、プロシージャのコールに使用するホスト言語パラメータのデータ型と一致しなくてはなりません。SQL モジュールでの SQL 文の記述では、パラメータ名のコロン（:）の使用は、オプションです。ただし、次の点に注意してください。

- コロンを使用する場合
Oracle Rdb では、コロンの使用は任意ですが、各パラメータ名の先頭にコロン（:）を付けるよう推奨しています。モジュール・ヘッダーで PARAMETER COLONS 句を指定すると、SQL では、すべてのパラメータの先頭にコロンを指定する必要があります。パラメータの先頭にコロンを指定した場合、パラメータ定義やパラメータの参照でも、コロンを使用してください。コロンの使用は、モジュール内で統一してください。パラ

メータの先頭でのコロンの使用は、今後の Oracle Rdb のリリースではデフォルトになる予定です。

- コロンを使用しない場合
モジュール・ヘッダーで PARAMETER COLONS 句を使用しないと、パラメータ定義やパラメータの参照内でのコロンの使用は無効になります。コロンを使用しない場合は、モジュール内で統一してください。

メイン・パラメータは、列の選択、操作、格納に使用するために宣言するパラメータです。SQL 文では、メイン・パラメータは次のように指定されます。

- INSERT 文の VALUES 句
- UPDATE 文の SET 句
- 次の文で行を選択するときの WHERE 句の条件値
 - UPDATE 文
 - DELETE 文
 - 1 行を検索する 1 行形式の SELECT 文
 - 列 SELECT 式
 - 1 行単位の更新または削除の実行で行を取り出すために、後でオープンするカーソル宣言の SELECT 式
- SELECT 文または FETCH 文の INTO 句

通常、メイン・パラメータには、データベースから取り出した値、ファイルのレコードから読み出した値、プログラムのプロンプトにユーザーが端末から入力した値などを格納します。パラメータには、リテラルやシステム・ルーチンが返した値が格納されることもあります。

8.9.1 メイン・パラメータの宣言

プログラムの外部のソースからパラメータ宣言をコピーしない場合には、使用する前にパラメータを宣言する必要があります。SQL 文で使用するパラメータの宣言に関する選択肢は、ホスト言語がサポートする宣言型に限られます。SQL プリコンパイラを使用するか SQL モジュール・プロセッサを使用するかに関係なく、ホスト言語がサポートしている限り、メイン・パラメータは、個別にも、また構造体としても宣言できます。

SQL プリコンパイラを使用する場合、他の宣言と同様、ソース・ファイルでメイン・パラメータを宣言してください。たとえば、プリコンパイルした COBOL プログラムで EMPLOYEE_RECORD 構造体を宣言するには、次のようなコードを使用します。

```
01  EMPLOYEE_RECORD.
    05  EMP_ID_P          PIC X(5) .
    05  L_NAME_P         PIC X(15) .
```

```
05 F_NAME_P          PIC X(11) .
05 M_INIT_P          PIC X(1) .
05 ADDRESS_1_P       PIC X(25) .
05 ADDRESS_2_P       PIC X(20) .
05 CITY_P            PIC X(20) .
05 STATE_P           PIC X(2) .
05 POSTAL_CODE_P     PIC X(5) .
05 SEX_P             PIC X(1) .
05 BIRTH_DATE_P      PIC X(23) .
05 STATUS_P          PIC X(1) .
```

SQL モジュール・プロセッサを使用する場合は、ホスト言語プログラムとモジュール・プロシージャでパラメータを宣言してください。ホスト言語プログラムで `EMPLOYEE_RECORD` 構造体を宣言するには、上の例と同じコードを使用します。モジュール・プロシージャで構造体を宣言するには、次のようなコードを使用します。

```
PROCEDURE fetch_record
(:emp_rec RECORD
    emp_id_p char(5),
    l_name_p char(15),
    f_name_p char(11),
    m_init_p char(1),
    address_1_p char(25),
    address_2_p char(20),
    city_p char(20),
    state_p char(2),
    postal_code_p char(5),
    sex_p char(1),
    birth_date_p char(23),
    status_p char(1)
END RECORD
SQLCODE);
```

個々のパラメータ名と同様に、レコード名の先頭にはコロン (:) を指定できます。

メイン・パラメータは、構造体で宣言する他に、個別に宣言することもできます。次の例では、COBOL ホスト言語プログラムでメイン・パラメータを宣言する方法を示します。

```
01 EMP_ID_P          PIC S9(4) COMP.
01 L_NAME_P          PIC S9(4) COMP.
01 F_NAME_P          PIC S9(4) COMP.
01 M_INIT_P          PIC S9(4) COMP.
01 ADDRESS_1_P       PIC S9(4) COMP.
01 ADDRESS_2_P       PIC S9(4) COMP.
01 CITY_P            PIC S9(4) COMP.
01 STATE_P           PIC S9(4) COMP.
01 POSTAL_CODE_P     PIC S9(4) COMP.
01 SEX_P             PIC S9(4) COMP.
```



```

01 BIRTH_DATE_P          PIC S9(4) COMP.
01 STATUS_P              PIC S9(4) COMP.

```

SQL モジュール・プロシージャでのパラメータ宣言の詳細は、4.2.3 項 (4-5) を参照してください。

8.9.2 メイン・パラメータの使用方法

この項では、データの検索や更新において、パラメータを使用する方法をいくつか説明します。この例では、プログラムは、8.9.1 項 (8-13) で説明した EMPLOYEE_RECORD を宣言しているものとします。この構造には、EMPLOYEES テーブルのすべての列に対応するメイン・パラメータが含まれ、次のようなフィールドで構成されています。

```

EMPLOYEE_RECORD (name of structure)
EMP_ID_P          (1st field)
L_NAME_P          (2nd field)
F_NAME_P          (3rd field)
M_INIT_P          (4th field)
ADDRESS_1_P       (5th field)
ADDRESS_2_P       (6th field)
CITY_P            (7th field)
STATE_P           (8th field)
POSTAL_CODE_P     (9th field)
SEX_P             (10th field)
BIRTH_DATE_P      (11th field)
STATUS_P          (12th field)

```

次の例は、SQL モジュール言語とプリコンパイルした SQL の SQL 文でパラメータを使用する方法を示しています。

次の例では、処理対象となるすべての行の列には、必ず値（NULL 値も含む）が格納されていることを前提とします。サンプルの personnel データベースを使って実際に文を実行すると、EMPLOYEES テーブルの行によっては、実効値が格納されている列と NULL 値が格納されている列があるため、エラーが発生します。NULL 値の条件処理を実行するインジケータ・パラメータの宣言と使用方法については、8.10 項 (8-18) を参照してください。次の例では、メイン・パラメータの使用方法のみを示します。

- 行の表示やレポートへの出力を行うには、プログラムで行をカーソルからフェッチし、列の値をパラメータに格納して、ホスト言語文でこれを操作します。
次の例では、FETCH 文は、EMPLOYEE_RECORD 構造体を指定します。

```

DECLARE EMPLOYEES_CURSOR CURSOR FOR
  SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
         ADDRESS_DATA_1, ADDRESS_DATA_2, CITY, STATE,
         POSTAL_CODE, SEX, BIRTHDAY, STATUS_CODE
  FROM EMPLOYEES
  WHERE STATUS_CODE = 1

```

```
OPEN EMPLOYEES_CURSOR  
FETCH EMPLOYEES_CURSOR INTO :EMPLOYEE_RECORD
```

EMPLOYEE_RECORD 構造体を指定するかわりに、変数を個別に宣言し、次の FETCH 文で各パラメータをリストすることもできます。

```
FETCH EMPLOYEES_CURSOR INTO :EMP_ID_P, :L_NAME_P, :F_NAME_P, :M_INIT_P,  
                             :ADDRESS_1_P, :ADDRESS_2_P, :CITY_P, :STATE_P,  
                             :POSTAL_CODE_P, :SEX_P, :BIRTH_DATE_P, :STATUS_P
```

カーソルの宣言と使用方法は、18 章 (18-1) を参照してください。

- 1 行を検索するには、カーソルではなく、1 行形式の SELECT 文を使用できます。1 行形式の SELECT 文には、2 つのパラメータと選択リスト項目を指定します。指定しない場合には、2 つのパラメータについてはカーソル宣言で指定し、選択リスト項目については FETCH 文で指定します。

次の例では、1 行形式の SELECT 文で、EMPLOYEE_RECORD 構造体を指定しています。

```
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,  
       ADDRESS_DATA_1, ADDRESS_DATA_2, CITY, STATE, POSTAL_CODE,  
       SEX, BIRTHDAY, STATUS_CODE  
       INTO :EMPLOYEE_RECORD  
FROM EMPLOYEES  
WHERE EMPLOYEE_ID = :INPUT_ID
```

次の例では、1 行形式の SELECT 文で、パラメータを個別にリストしています。

```
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,  
       ADDRESS_DATA_1, ADDRESS_DATA_2, CITY, STATE, POSTAL_CODE,  
       SEX, BIRTHDAY, STATUS_CODE  
       INTO  
       :EMP_ID_P, :L_NAME_P, :F_NAME_P, :M_INIT_P,  
       :ADDRESS_1_P, :ADDRESS_2_P, :CITY_P, :STATE_P, :POSTAL_CODE_P,  
       :SEX_P, :BIRTH_DATE_P, :STATUS_P  
FROM EMPLOYEES  
WHERE EMPLOYEE_ID = :INPUT_ID
```

1 行形式の SELECT 文の詳細は、『Oracle Rdb7 Introduction to SQL』および『Oracle Rdb7 SQL Reference Manual』を参照してください。

- 行を挿入するには、次のように、ホスト言語でパラメータに値を格納し、SQL でその値を行として格納します。

```
INSERT INTO EMPLOYEES (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,  
                      ADDRESS_DATA_1, ADDRESS_DATA_2, CITY, STATE,  
                      POSTAL_CODE, SEX, BIRTHDAY, STATUS_CODE)  
VALUES  
      (:EMPLOYEE_RECORD)
```

行の中の一部の列にしか値を格納しない場合は、次のように、INSERT 文では、使用する列と、関連するパラメータのみを指定します。

```
INSERT INTO EMPLOYEES (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL)
VALUES
    (:EMP_ID_P, :L_NAME_P, :F_NAME_P, :M_INIT_P)
```

行の挿入で、一部の列にしか値を格納しない場合、INSERT 文で指定していないテーブルの列には NULL 値が格納されます。

- 行の更新や削除では、ホスト言語は、行の選択基準となる値をパラメータに格納します。次の例では、INPUT_ID に、端末から入力される従業員 ID 番号を格納します。

```
DELETE FROM EMPLOYEES WHERE EMPLOYEE_ID = :INPUT_ID
```

- 行の更新では、ホスト言語は、新しい値をパラメータに格納します。SQL は、この値を使って、行の列に現在格納されている値を置き換えます。次の例は、カーソルを使って、パラメータ INPUT_ID で行を更新する方法を示しています。

```
DECLARE FIND_EMPLOYEE CURSOR FOR
    SELECT LAST_NAME, ADDRESS_DATA_1, ADDRESS_DATA_2,
           CITY, STATE, POSTAL_CODE
    FROM EMPLOYEES
    WHERE EMPLOYEE_ID = :INPUT_ID
OPEN FIND_EMPLOYEE
FETCH FIND_EMPLOYEE INTO :L_NAME_P, :ADDRESS_1_P, :ADDRESS_2_P,
                          :CITY_P, :STATE_P, :POSTAL_CODE_P
UPDATE EMPLOYEES
SET
    LAST_NAME      = :L_NAME_P,
    ADDRESS_DATA_1 = :ADDRESS_1_P,
    ADDRESS_DATA_2 = :ADDRESS_2_P,
    CITY           = :CITY_P,
    STATE          = :STATE_P,
    POSTAL_CODE    = :POSTAL_CODE_P
WHERE CURRENT OF FIND_EMPLOYEE
```

SQL モジュールでは、DECLARE CURSOR 文、FETCH 文、UPDATE 文でのパラメータの使用に加えて、カーソルをオープンするプロシージャのパラメータとして、INPUT_ID を宣言する必要があります。

SQL モジュール・プロシージャでのパラメータの使用の詳細は、4.2.3 項 (4-5) を参照してください。8.10 項 (8-18) では、この項で説明した例と似ていますが、インジケータ・パラメータを追加して NULL 値の条件処理を可能にした例を示します。

8.10 インジケータ・パラメータの宣言と使用方法

インジケータ・パラメータは、パラメータの1つであり、関連付けられているメイン・パラメータに NULL 値が割り当てられているかどうかを指定します。メイン・パラメータとデータベース間で NULL 値の受渡しがある場合、SQL 文でメイン・パラメータの名前を付けたインジケータ・パラメータを指定してください。インジケータ・パラメータを使用しない状態で行に NULL 値が格納されると、SQL でエラーが発生し、文の実行は強制終了し、エラーが返されます。このようなエラーが発生すると、パラメータに渡された値には信頼性がなくなります。

また、インジケータ・パラメータは、データベースから渡されたテキスト文字列が切り捨てられているかどうかを指定します。テキスト文字列が切り捨てられている場合、SQL は、データベース内の文字列の長さを、インジケータ・パラメータに割り当てます。また、SQL92 を表現形式として指定すると、SQLSTATE に警告メッセージを返します。

SQL が、列に値を返す式の評価で 1 行も処理しない場合、列には通常 NULL 値が格納されます。結果テーブルの列に、算術計算や、MIN、MAX、AVG および SUM などの関数の結果を格納する場合、結果が NULL 値になる可能性があります。したがって、このような行には、インジケータ・パラメータを定義してください。

COUNT 関数は、指定した条件に合う行が見つからないとき、NULL 値でなく 0 を返すので、COUNT 関数が返す結果の取出しではインジケータ・パラメータは必要ありません。

8.10.1 インジケータ・パラメータの宣言

インジケータ・パラメータは、ホスト言語プログラムで直接宣言するか、プログラム以外のソースから宣言をコピーすることができます。インジケータ・パラメータは、データベース・リポジトリ・ノードでは定義されませんが、リポジトリ・ユーティリティを使用して、別のノードでレコードを作成すれば、1 つまたは複数のテーブルにインジケータ・パラメータを定義することができます。次に、そのレコードをプログラムにコピーします。

ほとんどの言語では、インジケータ・パラメータを、符号付きのロングワードまたは符号付きのロングワード配列の要素として宣言します。個々に名前が付いたメイン・パラメータを参照する場合は、個々に名前を付けたインジケータ・パラメータを定義し、参照してください。個々に名前が付いたメイン・パラメータでなく、ホスト言語構造体を使用する場合は、インジケータ配列を必ず宣言してください。つまり、構造体（レコード）を使用してメイン・パラメータを格納する場合は、インジケータ配列を使用してインジケータ・パラメータを格納するということです。たとえば、8.9.1 項 (8-13) で設定した EMPLOYEE_RECORD 構造体を使う場合、IND_ITEM というフィールド名の要素を 12 要素を持ったインジケータ配列を宣言します（配列の要素の数は、すくなくとも、参照先の構造体のフィールド数が必要です）。

次の例では、プリコンパイルした COBOL プログラムで、インジケータ配列を宣言します。

```
01  IND_ARRAY.  
05  IND_ITEM OCCURS 12 TIMES
```

```
PIC S9(9) COMP.
```

SQL モジュール言語を使用する場合は、次のように、ホスト言語プログラムの中でインジケータ配列を宣言するだけでなく、SQL モジュール・プロシージャ内でインジケータ配列を宣言する必要があります。

```
PROCEDURE proc_1
  (:ind_array RECORD
    INDICATOR ARRAY OF 12 INTEGER
  END RECORD,
  SQLCODE);
```

インジケータ配列の要素を明示的に参照することはできないので、個々に名前を付けたメイン・パラメータの参照では、個々に名前を付けたインジケータ・パラメータを宣言し、参照する必要があります。

たとえば、プリコンパイルした COBOL プログラムや、SQL モジュール・プロシージャをコールするホスト言語プログラムでは、次のコードを使用して、EMPLOYEE_RECORD 構造体の 12 フィールドに対応するインジケータ・パラメータを宣言してください。

```
01 EMP_ID_IND          PIC S9(9) COMP.
01 L_NAME_IND          PIC S9(9) COMP.
01 F_NAME_IND          PIC S9(9) COMP.
01 M_INIT_IND          PIC S9(9) COMP.
01 ADDRESS_1_IND       PIC S9(9) COMP.
01 ADDRESS_2_IND       PIC S9(9) COMP.
01 CITY_IND            PIC S9(9) COMP.
01 STATE_IND           PIC S9(9) COMP.
01 POSTAL_CODE_IND     PIC S9(9) COMP.
01 SEX_IND             PIC S9(9) COMP.
01 BIRTH_DATE_IND      PIC S9(9) COMP.
01 STATUS_IND          PIC S9(9) COMP.
```

SQL モジュール言語を使用する場合は、次のように、SQL モジュール・プロシージャ内で、個々に名前を付けたインジケータ・パラメータを宣言する必要があります。

```
PROCEDURE proc_1
  (:EMP_ID              CHAR(5),
   :EMP_ID_IND          INTEGER,
   :L_NAME              CHAR(15),
   :L_NAME_IND          INTEGER,
   :F_NAME              CHAR(11),
   :F_NAME_IND          INTEGER,
   :M_INIT              CHAR(1),
   :M_INIT_IND          INTEGER,
   :ADDRESS_1           CHAR(25),
   :ADDRESS_1_IND       INTEGER,
   :ADDRESS_2           CHAR(20),
   :ADDRESS_2_IND       INTEGER,
```

```
:CITY                CHAR(20) ,
:CITY_IND            INTEGER,
:STATE              CHAR(2) ,
:STATE_IND          INTEGER,
:POSTAL_CODE        CHAR(5) ,
:POSTAL_CODE_IND    INTEGER,
:SEX                CHAR(1) ,
:SEX_IND            INTEGER,
:BIRTH_DATE         CHAR(23) ,
:BIRTH_DATE_IND     INTEGER,
:STATUS             CHAR(1) ,
:STATUS_IND         INTEGER);
```

8.10.2 インジケータ・パラメータの使用方法

8.10.1 項 (8-18) で説明したように、ホスト言語構造体を使用する場合は、インジケータ配列を必ず使用してください。個々に名前を付けたメイン・パラメータを使用する場合は、個々に名前を付けたインジケータ・パラメータを必ず使用してください。さらに、SQL の UPDATE 文や WHERE 句では、インジケータ配列やホスト言語構造体は使用できないので、個々に名前を付けたインジケータ・パラメータを使用してください。

インジケータ・パラメータ名をメイン・パラメータ名に追加する場合は、次の例のように、メイン・パラメータと同様、コロンを使用します。

```
INSERT INTO EMPLOYEES
VALUES (:EMPLOYEE_RECORD INDICATOR :IND_ITEM);
```

インジケータ配列の要素は、明示的に参照できません。

例で使った EMPLOYEE_RECORD には、EMPLOYEES テーブルの列に対応したフィールドが含まれています。この EMPLOYEE_RECORD を使用して、NULL 値を受け渡す可能性のあるフィールドを参照するときには、必ずインジケータ・パラメータを宣言および参照します。

たとえば、ミドルネームを持たない従業員もいるため、EMPLOYEES テーブルの MIDDLE_INITIAL 列には、NULL 値が含まれる可能性があります。M_INIT_P (EMPLOYEE_RECORD 内) を参照するには、インジケータ・パラメータを参照する必要があります。

M_INIT_IND という名前のインジケータ・パラメータを宣言する場合は、次の例のように、インジケータ・パラメータ名をメイン・パラメータ名に追加します。

```
UPDATE EMPLOYEES
SET MIDDLE_INITIAL = :M_INIT_P INDICATOR :M_INIT_IND
```

上記の例では、インジケータ・パラメータ M_INIT_IND は、符号付きロングワードとして個々に宣言されているホスト言語の実際のパラメータに対応付けるか、符号付きロングワード配列の要素に対応付けることができます。SQL プロセッサは、インジケータ・パラメータの値を格納している実際のパラメータの配列要素を、レコードにマッピングすることはあり

ません。したがって、ホスト言語構造体で使用するためにインジケータ配列を宣言するときは、NULL 値が格納される可能性のある列数だけ要素の数を指定し、配列の要素とインジケータ・パラメータの対応は SQL 文で決定します。

上記の例では、パラメータ M_INIT_P と、EMPLOYEES テーブルの列に対応している他のパラメータには値が格納され、検索やデータベースへの移動が実行されます。インジケータ・パラメータ M_INIT_IND や IND_ITEM のオカレンスには数字が格納され、これによって、プログラム (SQL) は、M_INIT_P の値にアクセスすべきかどうかを判断します。インジケータ・パラメータに、0 や正の整数が格納されている場合、プログラム (SQL) は、メイン・パラメータ M_INIT_P には値が格納されていると判断します。インジケータ・パラメータに負の整数が格納されている場合、プログラム (SQL) は、M_INIT_P には NULL 値が格納されていると判断します。

プログラムがインジケータ・パラメータを使用する方法は、パラメータを使用して、データベースからの読出しを行うか、書込みを行うかで決まります。

8.10.3 検索でのインジケータ・パラメータ

プログラムがデータベースから値を取り出すときには、次のような操作を実行します。

1. プログラムは、行をフェッチまたは選択して、個々に名前を付けたパラメータと、それに対応するインジケータ・パラメータに格納するか、ホスト言語構造体とインジケータ配列に格納します。
たとえば、プリコンパイルしたプログラムや SQL モジュール・プロシージャに、次のような文があるとしします。

```

FETCH EMPLOYEES_CURSOR INTO :EMP_ID_P      :EMP_ID_IND,
                               :L_NAME_P      :L_NAME_IND,
                               :F_NAME_P      :F_NAME_IND,
                               :M_INIT_P      :M_INIT_IND,
                               :ADDRESS_1_P    :ADDRESS_1_IND,
                               :ADDRESS_2_P    :ADDRESS_2_IND,
                               :CITY_P         :CITY_IND,
                               :STATE_P        :STATE_IND,
                               :POSTAL_CODE_P  :POSTAL_CODE_IND,
                               :SEX_P          :SEX_IND,
                               :BIRTH_DATE_P   :BIRTH_DATE_IND,
                               :STATUS_P       :STATUS_IND

```

プリコンパイルしたプログラムや SQL モジュール・プロシージャに、次のような文があるとしします。

```

FETCH EMPLOYEES_CURSOR
      INTO :EMPLOYEE_RECORD :IND_ITEM

```

2. プログラムは、各インジケータ・パラメータや配列要素をチェックし、NULL 値がある場合は、どの列に NULL 値が格納されているかを判断します。

FETCH 文や SELECT 文では、SQL は、フェッチした行の列に対応するインジケータ・パラメータの値を設定します。SQL は、列に NULL 値がある場合は -1、値が格納されている場合は正の整数をインジケータ・パラメータに設定します。

列に切り捨てられたテキスト文字列が含まれている場合、SQL は、元の文字列の長さをインジケータ・パラメータに設定します。ただし、別のデータ型がテキスト文字列に変換された結果、文字列が切り捨てられた場合、SQL は、元の文字列の長さではなく、変換後のテキスト文字列すべての長さをインジケータ・パラメータに設定します。たとえば、日付フィールドを文字のホスト言語変数に割り当てると、SQL は日付をテキストに変換します。日付をテキストで表すと 16 文字なので、ホスト言語変数が 16 文字未満ならば、SQL は、インジケータ・パラメータの値を 16 に設定します。

切り捨てられたテキスト文字列以外の値が列に含まれている場合、SQL は、インジケータ・パラメータに 0 を設定します。

3. プログラムが使用できるのは、対応するインジケータ・パラメータに 0 または正の整数が格納されているホスト言語パラメータの値のみです。
インジケータ・パラメータの値が -1 のパラメータから、プログラムが値を表示または書き込む場合、このプログラムは、現在の行ではなく、前にフェッチした行に格納された値を処理していることになります。

8.10.4 ソートでのインジケータ・パラメータ

プログラムが、パラメータを使用してデータベースに値を転送するときには、次のような処理を行います。

1. ホスト言語のインジケータ・パラメータまたは配列要素を 0（デフォルト値）に初期化します。
2. 値をメイン・パラメータに移動し、NULL でない値を割り当てます。
3. プログラムが値を移動していないメイン・パラメータについては、対応するインジケータ・パラメータまたはインジケータ配列の要素に -1（NULL）を移動します。
4. UPDATE 文または INSERT 文を実行し、行を処理します。

次の INSERT 文は、次のような内容を前提とします。

- 4 番目の要素 MIDDLE_INITIAL には、NULL や実効値が格納されます。したがって、この列にはインジケータ・パラメータを使用します。
- その他の列には、実効値が格納されます。したがって、インジケータ・パラメータを使用しないでリストします。
- 6 番目の要素 ADDRESS_DATA_2 には、INSERT 文が ADDRESS_DATA_2 をメイン・パラメータのリストで指定していないので、常に NULL 値が格納されます。

```
INSERT INTO EMPLOYEES  
(
```



```

EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
ADDRESS_DATA_1, CITY, STATE, POSTAL_CODE,
SEX, BIRTHDAY, STATUS_CODE
)
VALUES
(
:EMPLOYEE_ID_P, :LAST_NAME_P, :FIRST_NAME_P,
:MIDDLE_INITIAL_P :INDICATOR :MIDDLE_INITIAL_IND,
:ADDRESS_1_P, :CITY_P, :STATE_P, :POSTAL_CODE_P,
:SEX_P, :BIRTH_DATE_P, :STATUS_P
)

```

また、INSERT 文は、構造体を使用してパラメータを格納します。この場合、ホスト言語プログラムは、インジケータ配列 IND_ITEM のインジケータ・パラメータにおいて、6 番目以外のすべての要素を 0（格納値）、6 番目の要素を -1（NULL）に初期化し、MIDDLE_INITIAL_P が実効値か NULL 値かどうかによって、4 番目の要素を 0 または -1 に設定します。

次の SQL プリコンパイラの例は、EMPLOYEE_RECORD 構造体を使用して、パラメータを格納します。

```

INSERT INTO EMPLOYEES
(
EMPLOYEE_ID, LAST_NAME, FIRST_NAME,
MIDDLE_INITIAL, ADDRESS_DATA_1,
ADDRESS_DATA_2, CITY, STATE,
POSTAL_CODE, SEX, BIRTHDAY, STATUS_CODE
)
VALUES
( :EMPLOYEE_RECORD:IND_ITEM )

```

プリコンパイルしたプログラムでは、SQL は、対応するインジケータ・パラメータやインジケータ配列の要素に 0 または正の整数が格納されている場合にのみ、メイン・パラメータからの値を格納します。インジケータ・パラメータに負の整数が設定されている場合、SQL は、プログラムがパラメータに値を移動するかどうかに関係なく、対応するメイン・パラメータから値を格納しません。

SQL モジュールでは、SQL は、コール側のホスト言語モジュールにおいて、対応するインジケータ・パラメータやインジケータ配列の要素が 0 または正の整数が格納されてる場合にのみ、メイン・パラメータからの値を格納します。コールしているホスト言語モジュールで、インジケータ・パラメータに負の整数が設定されている場合、プログラムが列に一致した値をメイン・パラメータに移動するかどうかに関係なく、SQL は、プロシージャ・パラメータから列に値を格納しません。

注意: ブール式では、インジケータ・パラメータは予期しない動作をすることがあります。たとえば、A=NULL では、A が NULL でも真になりません。

8.11 パラメータを宣言して使用する場合の誤りの防止

この項では、この章の前の項で説明した SQL 文が参照するパラメータの宣言方法と使用方法についての要点を要約します。

SQL プリコンパイラか、SQL モジュール・プロセッサを使用している場合は、次の規則を考慮してください。

- SQL がサポートしていない宣言方法やデータ型を使用してパラメータを宣言しない。
- 列をメイン・パラメータに関連付ける場合、SQL が構造体を拡張する方法を忘れない。EMPLOYEES テーブルの ADDRESS_DATA_2 列に対するフィールド (ADDRESS_2_P) が含む構造体定義をプログラム内にインクルードすると仮定します。また、この列に常に NULL 値を格納すると仮定します。この場合、行を挿入またはフェッチする場合に構造体の名前をいままでどおり参照すると、SQL 文の列のリストから ADDRESS_DATA_2 列を省略できません。
リストした列から ADDRESS_DATA_2 列を省略しますが、構造体定義 (ADDRESS_2_P をインクルードした) を使用する場合、SQL プリコンパイラと SQL モジュール・プロセッサは、構造体内のフィールドに対して列を正しく関連付けられません。言い換えると、ADDRESS_2_P フィールド内の値は列 CITY に保存されており、CITY_P フィールド内の値は列 STATE に保存されているなどと仮定されます。構造体の名前を指定すると、次のような場合にコンパイル時エラーが発生します。
 - 列に対する記憶形式定義と構造体内の関連したフィールドに対する記憶形式定義の間の無効な変換
 - SQL 文によって指定された列数より多くのフィールドを含む構造体
 - SQL 文によって指定された列数より少ないフィールドを含む構造体

しかし、列リストが構造体のリストと一致しない場合、常にコンパイル時エラーが発生するとは限りません。この誤りは、警告（文字列の切捨てに対する警告）の発行にとどまるかもしれません。また、コンパイル時には誤りは検出されない場合もあります。この状況は、目的であった列の名前のかわりに別の列名を使用し、両方の列が互換性のある定義である場合に発生する可能性があります。また、誰かがテーブルを削除した後、列を再配列するためにテーブル（同様のデータ型を使用した列を含む）を再定義した場合や、使用する SQL 文が、すべての列の名前をリストするのではなく、アスタリスク (*) を使用して、列を指定する場合にも発生します。アスタリスクのかわりに列名を明示的に使用する必要があります。

- インジケータ配列を使用する場合、構造体内のフィールドの数に対して十分な配列要素があることを確認してください。
インジケータ配列内よりも多くのフィールドが構造体内にあれば、SQL は警告を返します。
- 個々に名付けたインジケータ・パラメータとインジケータ配列を適切に使用します。
SQL 文が列に関連して個々に名付けたメイン・パラメータを参照する場合、個々に名付けたインジケータ・パラメータを更新、評価および参照する必要があります。SQL 文が構造体を参照する場合、インジケータ配列要素を更新、評価および参照する必要があります。
SQL 文内で 2 つのインジケータ・パラメータを混同した場合、コンパイル時にエラーが発生します。インジケータ・パラメータを評価または更新するホスト言語文で、2 つのタイプのインジケータ・パラメータ（同じプログラム内で両方のタイプのインジケータ・パラメータを使用していると仮定します）を混同してしまうと、プログラムは、実行時に正しく実値および NULL 値を認識または保存できません。
- 構造体内に従属する構造体が含まれる場合、SQL がフィールドに対してどのように列に関連付けるかを忘れないでください。
SQL は、構造体内の基本フィールドのリストにすべての構造体を拡張します。このリストは、任意の従属構造体に対する基本フィールドを含みます。そして、SQL は構造体内の基本要素に対して列に関連付けます。これは、場合によっては、列のフィールド数より多い基本フィールド数を含む構造体を作成することを示します。
たとえば、基本フィールド YEAR_P、MONTH_P および DAY_P を含む構造体を定義し、名前 DATE_GROUP を使用して、この構造体を参照すると仮定します。
DATE_GROUP がテーブル内のただ 1 つの列に論理的に関連しているとしても、DATE_GROUP は、3 つの基本フィールドをレコードに追加する構造体です。したがって、SQL は、DATE_GROUP の 3 つのフィールドが 3 つの異なる列に関連すると仮定します。
この情報を忘れると、解決方法が直感的には明確でない、様々な予期せぬ実行時エラーが発生する可能性があります。
ホスト言語で同じ記憶領域を再定義できる場合は、この問題に対する簡単な解決方法があります。SQL は、構造体で与えられた記憶領域の最初の定義に基づく構造体を拡張するので、この問題を解決できます。したがって、複数のフィールド名をただ 1 つの列に関連させなければならない領域にマップする場合、基本フィールドを指定した領域の最初の定義を確認してください。同じ領域の 2 番目（およびそれに続く）の定義は、フィールドのグループを指定できます。
samples ディレクトリのサンプル・プログラムに、この解決方法の例があります。サンプル・プログラムは、レコード内で構造体をネストしません。しかし、プログラムは、複数のフィールドとして領域を定義する前に、1 つの基本フィールドとして与えられた記憶領域を最初に定義する一般的な原理を説明します。1 つの列に 2 つ以上のフィールドに関連付けるときや、行に適用するレコード内でこれを行いたい場合、常にこの原理を適用します。

詳細については、samples ディレクトリ内のホスト言語に対するサンプル・プログラムを確認し、『Oracle Rdb7 SQL Reference Manual』のサポートしている宣言とデータ型についての情報を参照してください。

8.11.1 埋込み SQL を使用する場合の誤りの防止

埋込み SQL を使用する場合に、8.11 項 (8-24) で示した規則に注意してください。さらに、SQLCODE を宣言し、かつ同じソース・ファイルで INCLUDE SQLCA 文を指定しないでください。これは、SQLCODE を 2 つ宣言する結果になり、SQL は SQLCA のみを使用します。したがって、INCLUDE SQLCA 文を指定する場合は、SQLCODE を別に宣言しないでください。

8.11.2 SQL モジュールを使用する場合の誤りの防止

SQL モジュールの使用時に注意すべき最も重要な規則として、実パラメータのデータ型、サイズおよび受渡しメカニズムは、コールされる SQL モジュール・プロシージャのパラメータに対するデータ型、サイズおよび受渡しメカニズムと同じである必要があります。したがって、プロシージャおよび実パラメータの両方は、ホスト言語と SQL の両方がサポートしているデータ型のセットに限定されます。場合によっては、データ型の間で変換する場合もあるでしょう。そうであれば、データ変換は、どちらかのモジュール内で常に行われ、モジュール間で値が受け渡されるときに変換されることはありません。

さらに、SQL の実装は、次に示す 2 つの方法の 1 つで、任意の SQL データ型キーワードをサポートしていることに留意してください。

- 列定義およびパラメータ宣言の両方に対して
- パラメータ宣言のみ

たとえば、SQL モジュール内でパラメータを宣言する場合でも、テーブル列を宣言する場合でも、CHAR(n) は常に n 文字の文字列記憶形式を指定します。しかし、DECIMAL(n) は、SQL モジュール・プロシージャで宣言されたパラメータに対してのみ n 桁のパック 10 進フィールドを定義します。列の場合、DECIMAL(n) は n の値に従って、固定小数点バイナリ記憶形式か、浮動小数点バイナリ記憶形式を定義します。(Oracle Rdb は、パック 10 進記憶形式を実装していません。) したがって、DECIMAL を使用して SQL モジュール内でプロシージャ・パラメータを定義する場合、SQL はプロシージャ・パラメータに対応する列内に保存されたデータをホスト言語モジュールに転送する前にパック 10 進数へ変換します。SQL データ型の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

次に、SQL モジュール言語を使用する場合の、パラメータの宣言方法および指定方法に対するさらなる考察を示します。

- デフォルトではコロンの使用は任意指定ですが、Oracle Rdb は、SQL 文内でパラメータ名を指定する前にコロン (:) を使用するように推奨しています。コロンを使用する場合は、モジュール・ヘッダーで PARAMETER COLONS 句を使用します。

- SQL 文では、同じ名前のパラメータおよび列を指定すると、パラメータでコロンを使用しない場合、テーブル名を使用して列名を修飾する必要があります。モジュール・ヘッダーで `PARAMETER COLONS` 句を使用すると、SQL 文が有効になるためには、パラメータの前にコロンを置く必要がありますので、SQL は列を列として認識します。

8.12 特定の言語でのパラメータの宣言方法と使用方法

この項では、特定のホスト言語でのパラメータの使用方法について説明します。

`samples` ディレクトリ下の `sql_all_datatypes` サンプル・プログラムは、この章で説明する宣言と操作のほとんどに関係しています。この例は、SQL プリコンパイラがサポートするすべての言語と、SQL モジュール言語で利用できます。プリコンパイル・プログラム内では、これらのプログラム内のコメントは、メイン・パラメータとインジケータ・パラメータではなく、メイン変数とインジケータ変数を意味しています。

8.12.1 Ada ソース・ファイルにおけるパラメータの宣言方法と使用方法

この項では、Ada 言語プログラムでのパラメータの宣言方法と使用方法について、個々のガイドラインを説明します。詳細は、6.7.1 項 (6-14) を参照してください。

Ada パッケージの使用方法

SQL では、直接的、または Ada パッケージの `SQL_STANDARD` をコールすることによって、ホスト言語変数を宣言できます。Ada パッケージにおける SQL の使用方法の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

ドル記号は使用できない

Ada は、プログラム内で指定する名前にドル記号 (\$) の使用を許可していません。このマニュアルでは `SQL$` や `RDB$` で開始するように指定しているパラメータ名やコール名を参照する場合、Ada プログラムでは、ドル記号 (\$) のかわりにアンダースコア (_) を使用します。たとえば、`RDB$LU_STATUS` のかわりに `RDB_LU_STATUS` と指定します。

8.12.2 C ソース・ファイルにおけるパラメータの宣言方法と使用方法

この項では、C 言語プログラムでのパラメータの宣言と使用方法に関する固有のガイドラインを説明します。詳細については、6.7.3 項 (6-16) を参照してください。

SQLCODE および SQLSTATE における大文字と小文字の区別

SQL プリコンパイラを使用している場合、宣言や `SQLCODE` パラメータまたは `SQLSTATE` パラメータの参照には、大文字を使用する必要があることに注意してください。SQL プリコンパイラは、SQL 文をホスト言語コールに変換します。これらのそれぞれのホスト言語コールでパラメータをリストする場合に、`SQLCODE` および `SQLSTATE` を大文字で指定します。

SQLCODE の宣言

SQLCODE を独立に宣言する場合、符号付き longword として宣言します。C 言語を使用する場合、実際の長さ（32 ビットまたは 64 ビット）は、ホスト言語コンパイラが longword を解釈する方法に従うことに注意してください。SQL は、32 ビット longword および 64 ビット longword の両方をサポートしています。

SQL プリコンパイラを使用する場合、プリコンパイラが SQLCODE の定義を確認するので、自動的にサポートされます。

SQL モジュール言語を使用する場合、ホスト言語プログラムで SQLCODE を longword として宣言し、-lsqcode または LONG_SQLCODE スイッチを使用して、SQL モジュールをコンパイルします。

データの整列

OpenVMS
Alpha

C プログラムがコールする SQL モジュールを使用する場合で、モジュール・ヘッダーで指定した LANGUAGE 句が C の場合、ALIGN_RECORDS 修飾子を使用してください。

特に、C を使用して、FLOAT=D_FLOAT 修飾子を使用してホスト言語プログラムをコンパイルする場合、この修飾子を使用する必要があります。◆

SQLCA の宣言と整列

OpenVMS
Alpha

OpenVMS Alpha では、SQL モジュール・プロセッサを使用して、モジュール・ヘッダー言語句で C を指定する場合、または C に対して SQL プリコンパイラを使用する場合には、SQL はデフォルトで構造体のフィールドを整列します。しかし、SQLCA のメンバーの整列を許可しないでください。

SQLCA 構造体を明示的に定義したときは、構造体の定義を C プリプロセッサ・ディレクティブ #pragma nomember_alignment で囲み、構造体の整列を防止します。

SQL INCLUDE を使用する場合は、プリプロセッサ・ディレクティブを使用する必要はありません。◆

文字列

SQL C プリコンパイラまたは SQL モジュール・プロセッサを使用して、C をモジュール言語として指定する場合、SQL はすべての C 文字列を NULL 終了文字列として変換します。これは、SQL がこれらの文字列をデータベースからプログラムに渡す場合に、文字列の最後に NULL 文字列の空白を予約していることを意味します。プログラムが入力用に文字列をデータベースに渡す場合、SQL は NULL 文字を検索して、何文字をデータベースに格納するかを決定します。SQL は、NULL 文字の前にある文字のみを格納します。NULL 文字自体は、格納されません。

SQL が C 文字列を変換する方法が原因で、バイナリ・データをデータベースに渡す、またはデータベースから受け取るアプリケーションで問題が発生する可能性があります。SQL C プリコンパイラを使用する場合に発生するこれらの問題を回避するには、SQL が提供する

\$SQL_VARCHAR データ型を使用します。C 言語ホスト・プログラムで SQL モジュール言語を使用する場合に発生するこれらの問題を回避するには、モジュール言語を GENERAL として指定します。

さらに、SQL が C 文字列を変換する方法は、SQL INCLUDE 句または SQL FROM DICTIONARY 句を使用して、リポジトリからレコード定義をコピーするプログラムに影響を与えます。

CompaqTru64 UNIX における整数データ型

Compaq Tru64 UNIX

CompaqTru64 UNIX では、OpenVMS 上では存在しない整数変数と longword 変数の間に区別があります。次の表は、CompaqTru64 UNIX 上の C コンパイラに対する 3 つの整数データ型を示しています。

データ型	データ型のサイズ
short (int)	16 ビット
int	32 ビット
long (int)	64 ビット

C で記述されたアプリケーションは、SQL モジュール言語プログラムをコールする場合に正しい宣言を使用する必要があります。ANSI/ISO 標準は、コール側プログラムはポインタを longword で渡す必要があると規定しています。これは、INTEGER 型の SQL モジュール言語仮パラメータと C の実パラメータに適用されます。

SQL は、INTEGER パラメータは、C "long" へのポインタとしてコールされると仮定します。
-int32 のコマンドライン・オプションを使用すると、SQL は、パラメータが C "int" へのポインタとしてコールされると仮定します。

-int コマンドライン・オプションで指定したサイズを上書きするには、次の例で示すように INTEGER パラメータを 4 バイトまたは 8 バイトとして指定します。

```
salary_amount INTEGER is 4 BYTES
```

SQL プリコンパイラは、これらのデータ型のサイズを正しく解釈します。◆

SMALLINT データ型の使用方法

OpenVMS
Alpha

VAX C は正確な数値スケールをサポートしていないため、テーブル内の列で SMALLINT(2) データ型を使用している場合には、SQL を使用してデータを変換する必要があります。たとえば、カーソルを使用して、テーブルから 3 列をフェッチする場合には、次のような SQL モジュール言語宣言とプロシージャを使用します。

```
DECLARE my_cursor CURSOR FOR
  SELECT X, Y, Z
  FROM MYTABLE
```

```
WHERE W = IN_DATA
.
.
.
PROCEDURE FETCH_EM
  SQLCODE
  A      CHAR(5)
  B      CHAR(10)
  C      REAL;  -- Corresponds to column Z of type SMALLINT(2)
  FETCH my_cursor INTO A, B, C;
```

C 言語ホスト・プログラムは、次の例に示すようにプロシージャ FETCH_EM をコールします。

```
.
.
.
static char in[5] = "0001";
static char out_a[6];
static char out_b[11];
static float out_c;
.
.
.
  FETCH_EM(&sqlcode, out_a, out_b, &out_c);
  if (sqlcode == 100)
    break;
  if (sqlcode < 0)
.
.
.
```

REAL データ型のパラメータは、浮動小数点型のホスト言語変数にマップされることに注意してください。◆

8.12.3 COBOL ソース・ファイルにおけるパラメータの宣言方法と使用方法

SQL プリコンパイラを使用している場合、COBOL ソース・ファイル内の埋込み SQL 文に適用する一般的な規則を説明する 6.7.4 項 (6-18) を参照してください。

8.12.4 FORTRAN ソース・ファイルにおけるパラメータの宣言方法と使用方法

SQL プリコンパイラを使用している場合、FORTRAN ソース・ファイル内の埋込み SQL 文に適用する一般的な規則を説明する 6.7.5 項 (6-20) を参照してください。

8.12.5 Pascal ソース・ファイルにおけるパラメータの宣言方法と使用方法

SQL プリコンパイラを使用している場合、Pascal ソース・ファイル内の埋込み SQL 文に適用する一般的な規則を説明する 6.7.6 項 (6-22) を参照してください。

8.12.6 PL/I ソース・ファイルにおけるパラメータの宣言方法と使用方法

SQL プリコンパイラを使用している場合、PL/I ソース・ファイル内の埋込み SQL 文に適用する一般的な規則を説明する 6.7.7 項 (6-23) を参照してください。

8.12.7 SQL モジュールとコール側プログラムにおけるパラメータの宣言方法と使用方法

SQL モジュール言語におけるパラメータの宣言方法と使用方法の詳細は、4 章 (4-1) を参照してください。4.2.4 項 (4-9) では、コール側プログラムとコールされるプログラムのパラメータの関係について説明します。

samples ディレクトリ内の `sql_all_datatypes_ada.sqlmod` モジュール・ファイルは、SQL モジュール内でパラメータを使用する方法を説明しています。samples ディレクトリ内の Ada ソース・プログラム `sql_all_datatypes.ada` は、`sql_all_datatypes_ada.sqlmod` モジュール内のプロシージャをコールします。

日付時刻データ型の使用方法

Oracle Rdb では、ANSI/ISO SQL 標準に規定された規則に準拠する一連の日付時刻データ型と値関数のセットを提供します。この日付時刻データ型と値関数によって、プログラマとデータベース管理者が利用できる日付、時刻、間隔データの制御機能が拡張されます。

この章の各項では、次の方法について解説します。

- 日付時刻データ型によるデータの格納
- プログラム内での日付時刻データ型の使用方法
- 日付時刻データ型の移植性に関する計画
- 別の日付時刻データ型を使用するためのアプリケーションとデータベースの変換
- DATE VMS データ型の処理
- DATE VMS データ型を使用するアプリケーションの移植
- 動的 SQL での日付時刻データ型の使用方法

この章では、ユーザーが日付時刻データ型と値関数を使用してプログラミングする際に役立つ一連の例を示します。例では、corporate_data サンプル・データベースを使用します。

参考: 日付時刻データ型を初めてご使用になる場合は、『Oracle Rdb7 Introduction to SQL』を参照してください。日付時刻データ型を使用したテーブルとドメインの作成の詳細は、『Oracle Rdb7 Guide to Database Design and Definition』を参照してください。

日付時刻データ型書式、データ型の変換規則、日付時刻リテラル、値関数、有効な操作、言語に固有の日付時刻データ型の宣言については、『Oracle Rdb7 SQL Reference Manual』を参照してください。

9.1 日付時刻データ型によるデータの格納

日付時刻データ型の列にデータを格納する場合は、次の2つの点を考慮してください。

- 日付時刻リテラルとデフォルト値を使用する場合、デフォルト値の小数点以上の桁数と小数点以下の桁数の精度が列と一致する必要があります。間隔修飾子も同じでなければなりません。
- 指定された秒の小数点以上の桁数の精度と小数点以下の桁数の精度がリテラルの値と同じかそれより大きい場合は、リテラル自体の先頭にゼロを付加して精度を一致させる必要はありません。たとえば、次に示す2つの間隔リテラル式は等価であり、有効です。

```
INTERVAL '12:30' HOUR(4) TO MINUTE
INTERVAL '0012:30' HOUR(4) TO MINUTE
```

例 9-1 (9-2) に、corporate_data データベース内の DAILY_HOURS テーブルに日付時刻データ型を挿入する方法を示します。

例 9-1 日付時刻データの挿入

```
SQL> INSERT INTO ADMINISTRATION.ACCOUNTING.DAILY_HOURS
1>      (EMPLOYEE_ID, START_TIME, END_TIME)
2>      VALUES ('00415',
3>              TIMESTAMP '1995-07-15 07:30:45.33',
4>              TIMESTAMP '1995-07-15 17:15:22.18');
1 row inserted
SQL>
SQL> SELECT * FROM ADMINISTRATION.ACCOUNTING.DAILY_HOURS;
   EMPLOYEE_ID   START_TIME                END_TIME
-----
00415          1995-07-15 07:30:45.33    1995-07-15 17:15:22.18
1 row selected
SQL>
```

次の例では、日付時刻データをデータベースに格納する方法を示します。たとえば、日付時刻データ型を使用して次のようにテーブルを作成するとします。

```
SQL> CREATE TABLE DATE_TABLE
1> ( CHAR_VAR   CHAR(20),
2>   COL_DATE   TIMESTAMP(0)
3>   CHECK (COL_DATE BETWEEN TIMESTAMP '1995-1-1 00:00:00'
4>         AND TIMESTAMP '1996-1-1 00:00:00') NOT DEFERRABLE,
5>   COL_YEAR_TO_MONTH   INTERVAL YEAR TO MONTH
6>   CHECK (COL_YEAR_TO_MONTH >= INTERVAL '00-00' YEAR TO MONTH)
7>   NOT DEFERRABLE,
8>   COL_DAY_TIME        INTERVAL HOUR TO SECOND);
```

INSERT 文を使用して日付時刻データを格納する場合は、小数点以上と以下の精度に関して特に注意が必要です。次のように、精度が TIMESTAMP(2) のリテラルのタイムスタンプ・

データを、TIMESTAMP(0) の精度で定義された COL_DATE 列に挿入しようとする、SQL はエラーメッセージを発行します。

```
SQL> INSERT INTO DATE_TABLE (COL_DATE)
1> VALUES (TIMESTAMP '1995-6-12 16:12:32.05');
%SQL-F-UNSDATASS, Unsupported date/time assignment from <Source> to COL_DATE
-SQL-F-DATESCANEQ, Date/time expressions with different fractional seconds
precision are not comparable
```

次の例は、YEAR TO MONTH 間隔の小数点以上の精度が低すぎるのでエラーが発生します。

```
SQL> INSERT INTO DATE_TABLE (COL_YEAR_TO_MONTH)
1> VALUES (INTERVAL '26512-10' YEAR(4) TO MONTH);
%SQL-F-DATCONERR, Data conversion error for string '26512-10'
-COSI-F-IVTIME, invalid date or time
```

次の例は、精度がリテラル表現と同じかそれより大きいので正常に実行されます。この例は、元のドメインの定義と変更されたドメインの定義の両方で正常に実行されます。リテラルが同じ間隔型の場合、SQL は、データをドメインの宣言済みデータ型に変換します。

```
SQL> INSERT INTO DATE_TABLE
1> VALUES ('STRING',
2>          TIMESTAMP '1995-6-12 16:12:32',
3>          INTERVAL '265-10' YEAR(3) TO MONTH,
4>          INTERVAL '73:23:34' HOUR(4) TO SECOND);
```

9.2 プログラムでの日付時刻データ型の使用方法

SQL モジュール言語と SQL プリコンパイラでは、日付時刻データ型を処理する方法が異なります。この項では、それぞれのプロセッサを使用するプログラムで、日付時刻型データを使用する例を示します。日付時刻文字列を Oracle Rdb が認識する日付時刻の書式に変換する方法と、文字列に戻す方法を示します。この種のデータ型変換は、日付時刻データ型を使用した正しいプログラミングには非常に重要です。

文字列リテラルまたは変数を DATE ANSI 列に代入する場合は、次の書式を使用します。ただし、yyyy は 4 桁の年、nn は 2 桁の月、dd は 2 桁の日を表します。

yyyy-mm-dd

他の日付時刻データ型の書式については、『Oracle Rdb7 SQL Reference Manual』を参照してください。

DATE データ型と CURRENT_TIMESTAMP 値関数用の ANSI/ISO 環境を設定するには、DECLARE MODULE 文の DEFAULT DATE FORMAT 句かモジュール・ヘッダーを使用します。

ただし、CURRENT_TIMESTAMP 値関数はデフォルトの日付書式によって動作が異なります。詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

9.2.1 プログラム開発のための日付時刻データ型の変換

日付時刻データ型を正しく処理するために、場合によっては日付時刻文字列を Oracle Rdb が認識する日付時刻の書式に変換する必要があります。たとえば、日付時刻データ型を含むレコード構造をとる行全体を格納する場合は、これらのデータ型を Oracle Rdb の日付時刻書式に変換します。同様に、プログラムで表示する場合は日付時刻書式を文字列に戻します。

例 9-2 (9-4) に、文字列と Oracle Rdb が要求する日付時刻書式との変換を行う SQL モジュール・プロシージャを示します。SQL モジュール言語を使用する場合、バイナリ表現に使用するホスト言語変数は 4 ワードまたは 4 ワードの言語同等として宣言してください。

例 9-2 日付時刻データ型変換のための SQL モジュール・セグメント

```
-- This procedure converts a character string to the DATE ANSI format.
PROCEDURE CVT_TO_DATETIME
  (SQLCODE,
   :CHAR_VAR      CHAR(23),
   :DT_VAR        DATE ANSI);
BEGIN
  SET :DT_VAR = CAST(:CHAR_VAR AS DATE ANSI); (1) (9-4)
END;

-- This procedure converts a character string to the INTERVAL format.
PROCEDURE CVT_TO_INTERVAL
  (SQLCODE,
   :CHAR_VAR      CHAR(23),
   :DT_VAR        INTERVAL DAY(6) TO SECOND);
BEGIN
  SET :DT_VAR = CAST(:CHAR_VAR AS INTERVAL DAY(6) TO SECOND); (2) (9-4)
END;

-- This procedure converts a binary value to a character string.
PROCEDURE CVT_INTERVAL_TO_CHAR
  (SQLCODE,
   :CHAR_VAR      CHAR(23),
   :DT_VAR        INTERVAL DAY(6) TO SECOND);
BEGIN
  SET :CHAR_VAR = CAST (:DT_VAR AS VARCHAR(23)); (3) (9-4)
END;
```

次の説明は、例 9-2 (9-4) の番号に対応しています。

- (1) CAST 関数によって、文字列を DATE ANSI のデータ型に変換し、8 バイトのバイナリ値として返します。
- (2) CAST 関数によって、文字列を間隔 DAY TO SECOND のバイナリ・データ型に変換します。
- (3) CAST 関数によって、間隔 DAY TO SECOND を文字列に変換します。

C では、例 9-3 (9-5) に示すように、日付時刻データ型をすべて 4 ワード (char[8]) で宣言する必要があります。

例 9-3 日付時刻データ型変換のための C プログラム

```
char    int_ds_string[24],
        date_ansi_string[24];
int     sql_return_status;
struct {
    char        employee_id[6];
    char        last_name[21];
    char        q_date_ansi[8];
    char        q_date_vms[8];
    char        q_time[8];
    char        q_timestamp[8];
    char        q_int_ym[8];
    char        q_int_ds[8];
}date_rec;

main()
{
    strcpy(date_ansi_string,"1995-9-12¥0");
    cvt_to_datetime(&sql_return_status,date_ansi_string,date_rec.q_date_ansi);
    strcpy(int_ds_string,"-12:10:35:59.05¥0");
    cvt_to_interval(&sql_return_status,int_ds_string,date_rec.q_int_ds);
}
```

9.2.2 SQL プリコンパイラでの日付時刻データ型の使用方法

SQL プリコンパイラを使用する場合は、日付時刻データ型を表す変数専用 SQL で定義されたデータ型を使用できます。

SQL では、DATE、TIME、TIMESTAMP、INTERVAL データ型をバイナリのホスト言語変数にフェッチできますが、多くの場合、内部書式を認識するのは Oracle Rdb に限定されます。バイナリのホスト言語変数は、クエリーからデータを受け取って他のクエリーの入力として SQL に戻す場合にのみ使用してください。

ホスト言語変数の定義を容易にするために、SQL では SQL プリコンパイラで使用する一連の日付時刻データ型を定義します。たとえば、COBOL プログラムでは次のデータ型を使用して変数を宣言します。

01 P_DATE	SQL_DATE.
01 P_DATE_V	SQL_DATE_VMS.
01 P_TIME	SQL_TIME(0).
01 P_TIMESTAMP	SQL_TIMESTAMP(2).
01 P_INTER_1	SQL_INTERVAL (YEAR TO MONTH).
01 P_INTER_2	SQL_INTERVAL (DAY TO HOUR).

例 9-4 (9-6) では、SQL プリコンパイラのデータ型を使用して日付時刻ホスト変数を宣言しています。

例 9-4 日付時刻データ型を使用する SQL プリコンパイラ・プログラム

```
#include <string.h>
main()
{
char str_var[24];
long SQLCODE;
struct {
    char                employee_id[6];
    char                last_name[21];
    SQL_DATE_ANSI       my_date_ansi;
    SQL_DATE_VMS        my_date_vms; (1) (9-6)
    SQL_TIME            my_time;
    SQL_TIMESTAMP(1)    my_timestamp;
    SQL_INTERVAL(YEAR(3) TO MONTH) my_int_yr32mo;
    SQL_INTERVAL(DAY(4) TO SECOND (2)) my_int_d42s2;
}date_rec;
strcpy(str_var,"33-4¥0");
EXEC SQL DECLARE ALIAS FILENAME personnel;
/* Conversion from a string to a date-time format. */
EXEC SQL BEGIN
    SET :date_rec.my_int_yr32mo = CAST(:str_var AS INTERVAL YEAR TO MONTH);
    END;
/* Conversion from date-time format to a string. */
EXEC SQL BEGIN
    SET :str_var = CAST(:date_rec.my_int_yr32mo AS CHAR(24)); (2) (9-6)
    END;
}
```

次の説明は、例 9-4 (9-6) の番号に対応しています。

- (1) バイナリ書式が必要な場合は、SQL プリコンパイラのデータ型を使用します。
- (2) 日付時刻データを表示する場合は、CAST 関数を使用して文字列に変換します。

SQL プリコンパイラで使用する特殊なデータ型の一覧は、『Oracle Rdb7 SQL Reference Manual』を参照してください。samples ディレクトリにはプログラム `sql_all_datatypes_date` が含まれており、様々な言語の日付時刻データ型を使用する方法を示しています。

例 9-5 (9-7) に、サンプル・データベース `corporate_data` から週給を計算する SQL プリコンパイラ用の C プログラムを示します。これは、9.2.3 項 (9-8) に示す SQL モジュール・プログラムと同等です。

例 9-5 SQL プリコンパイラによる日付時刻データ型を使用した C プログラム

```

#include <stdio.h>
#include <sql_rdb_headers.h>
/* Include the SQLCA. */
EXEC SQL INCLUDE SQLCA;
/* Declare the alias. */
EXEC SQL DECLARE ALIAS FILENAME corporate_data;
main()
{
/* Declare return status variable for error handling. */
int    sql_return_status;
/* Variables for program use. */
char   start_date[11],
        end_date[11];
char   emp_id[6],
        last_name[21],
        hours_worked[12];
int    hours,min;
float  hr_rate,wages,sec;
printf("Enter START DATE (YYYY-MM-DD):");
gets(start_date);
printf("Enter END DATE (YYYY-MM-DD):");
gets(end_date);
/* Declare a cursor. */
EXEC SQL DECLARE WEEKLY_HRS_CURSOR CURSOR FOR
        SELECT E.EMPLOYEE_ID, LAST_NAME, HOURLY_RATE,
               CAST(SUM(HOURS_WORKED) AS CHAR(12)),
               EXTRACT(HOUR FROM SUM(HOURS_WORKED)),
               EXTRACT(MINUTE FROM SUM(HOURS_WORKED)),
               EXTRACT(SECOND FROM SUM(HOURS_WORKED))
        FROM ADMINISTRATION.ACCOUNTING.DAILY_HOURS D,
             ADMINISTRATION.PERSONNEL.HOURLY_HISTORY H,
             ADMINISTRATION.PERSONNEL.EMPLOYEES E
        WHERE (H.EMPLOYEE_ID = D.EMPLOYEE_ID)
              AND (E.EMPLOYEE_ID = H.EMPLOYEE_ID)
              AND CAST(START_TIME AS DATE ANSI)
              BETWEEN CAST('1995-07-01' AS DATE ANSI) AND
                     CAST('1995-07-19' AS DATE ANSI)
        GROUP BY E.EMPLOYEE_ID, LAST_NAME,HOURLY_RATE;
/* Open the cursor. */
EXEC SQL OPEN WEEKLY_HRS_CURSOR;
printf("%20s WEEKLY PAYROLL ( %s - %s)\n\n", " ",start_date,end_date);
printf("Emp Id      Last Name %11s Hours %14s Rate %9s Wages\n\n",
       " "," "," ");
/* Get data for employees until end of file. */
while (SQLCA.SQLCODE == 0)

```

```
{
    EXEC SQL FETCH WEEKLY_HRS_CURSOR INTO
        :emp_id, :last_name, :hr_rate,
        :hours_worked, :hours, :min, :sec;
    if (SQLCA.SQLCODE != 0 && SQLCA.SQLCODE != 100)
    {
        sql_signal();
        EXEC SQL ROLLBACK;
    }
    else
        if (SQLCA.SQLCODE != 100)
        {
            if (hours < 40)
                wages = (float)hours * hr_rate +
                    (float)min/60 * hr_rate +
                    (float)sec/3600 * hr_rate;
            else
                wages = 40.00 * hr_rate +
                    (float)(hours - 40) * hr_rate * 1.5 +
                    (float)min/60 * hr_rate * 1.5 +
                    (float)sec/3600 * hr_rate * 1.5;
            printf("%s %25s %10s %15.2f %15.2f\n",
                emp_id,last_name,hours_worked,hr_rate,wages);
        }
    }
    EXEC SQL ROLLBACK;
}
```

9.2.3 SQL モジュール言語での日付時刻データ型の使用方法

この項では、CAST 関数と EXTRACT 関数を使用して日付時刻データ型を日付時刻書式に変換し、40 時間を超えた場合の 5 割増給に基づいて週給のレポートを生成する方法を示します。

例 9-6 (9-8) に、日付時刻データ型を使用して行をフェッチする SQL モジュール mod_weekly_pay.sqlmod を示します。

例 9-6 日付時刻データ型を使用する SQL モジュール

```
-----
-- The module illustrates how to use SQL module language to use a cursor to
-- fetch columns of date-time data type.
-----
-- Header Information Section
-----
MODULE          MOD_WEEKLY_PAY          -- Module name
LANGUAGE        C                      -- Language of calling program
ALIAS           RDB$DBHANDLE
```

```

PARAMETER          COLONS
-----
-- DECLARE Statements Section
-----
DECLARE ALIAS FILENAME corporate_data  -- Declaration of the database
DECLARE WEEKLY_HRS_CURSOR CURSOR FOR (1) (9-10)
      SELECT E.EMPLOYEE_ID, LAST_NAME, HOURLY_RATE,
             CAST(SUM(HOURS_WORKED) AS CHAR(12)), (2) (9-10)
             EXTRACT(HOUR FROM SUM(HOURS_WORKED)), (3) (9-10)
             EXTRACT(MINUTE FROM SUM(HOURS_WORKED)),
             EXTRACT(SECOND FROM SUM(HOURS_WORKED))
      FROM ADMINISTRATION.ACCOUNTING.DAILY_HOURS D,
            ADMINISTRATION.PERSONNEL.HOURLY_HISTORY H,
            ADMINISTRATION.PERSONNEL.EMPLOYEES E
      WHERE ( (H.EMPLOYEE_ID = D.EMPLOYEE_ID)
              AND (E.EMPLOYEE_ID = H.EMPLOYEE_ID)
              AND (CAST(D.START_TIME AS DATE ANSI) (4) (9-10)
                    BETWEEN CAST(:DATE1 AS DATE ANSI) AND (5) (9-10)
                        CAST(:DATE2 AS DATE ANSI)))
      GROUP BY E.EMPLOYEE_ID, LAST_NAME, HOURLY_RATE
-----
-- Procedure Section
-----
-- This procedure opens the cursor that has been declared for the
-- WEEKLY_PAY table.
PROCEDURE OPEN_CURSOR
  (SQLCODE,
   :DATE1      CHAR(10),
   :DATE2      CHAR(10));
  OPEN WEEKLY_HRS_CURSOR;
-- This procedure fetches the data from the opened cursor.
PROCEDURE FETCH_WEEKLY_HOURS
  (SQLCODE,
   :EMP_ID      CHAR(5),
   :LAST_NAME   CHAR(20),
   :HRLY_RATE   REAL,
   :HRS_WORKED  CHAR(11),
   :WEEK_HOURS  INTEGER,
   :WEEK_MIN    INTEGER,
   :WEEK_SEC    REAL);
  FETCH WEEKLY_HRS_CURSOR INTO :EMP_ID, :LAST_NAME, :HRLY_RATE,
                              :HRS_WORKED, :WEEK_HOURS,
                              :WEEK_MIN, :WEEK_SEC;
-- This procedure rolls back the transaction.
PROCEDURE ROLLBACK_TRANSACTION
  (SQLCODE);
  ROLLBACK;

```

次の説明は、例 9-6 (9-8) の番号に対応しています。

- (1) カーソルを宣言してレコード・ストリームを作成し、各時間従業員の週ごとの時間数を求めます。
- (2) CAST 関数によって、日付をバイナリ書式から文字列に変換します。
SUM(HOURS_WORKED) 式では、データ型間隔 DAY TO SECOND を仮定しています。
これらは、文字列に変換してから表示されます。
- (3) EXTRACT 関数を使用して HOUR、MINUTE、SECOND の数値を作成します。上記の数値は、週給を計算するアルゴリズムで使用します。
- (4) CAST 関数を使用して、列 START_TIME を TIMESTAMP から DATE ANSI に変換します。この例では、ユーザーに給与計算で使用する 2 つの日付、すなわち希望する開始日と終了日の入力进行を要求します。日付のみが有効なので、CAST 操作によって START_TIME 列の日付の部分のみを比較できます。
- (5) CAST 関数を使用して、ユーザーが入力した日付文字列 (YYYY-MM-DD) を DATE ANSI に変換します。

例 9-7 (9-10) に、mod_weekly_pay.sqlmod モジュールで使用する日付時刻データ型を取り出す C 言語プログラム mod_datetime.c.c を示します。

例 9-7 日付時刻演算を使用した C プログラム

```
/* ABSTRACT:
This sample C program is used with mod_weekly_pay.sqlmod to demonstrate
the retrieval of date-time data types. Note the use of the CAST and
EXTRACT functions to convert to and from a date-time format.
This application prompts the user for a START_DATE and END_DATE. The
total hours worked for each employee during this time period are returned
along with the employee's hourly rate. Weekly pay is calculated assuming
time and a half after 40 hours.
*/
#include <stdio.h>
#include <sql_rdb_headers.h>
main()
{
/* Declare return status variable for error handling. */
    int    sql_return_status;
/* Variables for program use. */
    char    start_date[11], (1) (9-12)
            end_date[11];
    char    emp_id[6],
            last_name[21],
            hours_worked[12];
    int     hours,
            min;
```

```

float hr_rate,
    wages,
    sec; (2) (9-12)
printf("Enter START DATE (YYYY-MM-DD):");
gets(start_date);
printf("Enter END DATE (YYYY-MM-DD):");
gets(end_date);
/* Open the cursor. */
open_cursor(&sql_return_status,start_date,end_date);
if (sql_return_status < 0)
{
    sql_signal();
    rollback_transaction(&sql_return_status);
}
printf("%20s WEEKLY PAYROLL ( %s - %s)¥n¥n", " ",start_date,end_date);
printf("Emp Id      Last Name %11s Hours %14s Rate %9s Wages¥n¥n",
        " "," "," ");
/* Get data for employees until end of file. */
while (sql_return_status == 0)
{
    fetch_weekly_hours(&sql_return_status,emp_id,last_name,&hr_rate,
                      hours_worked,&hours,&min,&sec); (3) (9-12)
    if (sql_return_status != 0 && sql_return_status != 100)
    {
        sql_signal();
        rollback_transaction(&sql_return_status);
    }
    else
        if (sql_return_status != 100)
        {
            if (hours < 40)
                wages = (float)hours * hr_rate +
                        (float)min/60 * hr_rate +
                        (float)sec/3600 * hr_rate;
            else
                wages = 40.00 * hr_rate +
                        (float)(hours - 40) * hr_rate * 1.5 +
                        (float)min/60 * hr_rate * 1.5 +
                        (float)sec/3600 * hr_rate * 1.5;
            printf("%s %25s %10s %15.2f %15.2f¥n",
                    emp_id,last_name,hours_worked,hr_rate,wages);
        }
    }
    rollback_transaction(&sql_return_status);
    if (sql_return_status < 0)
    {
        sql_signal();
    }
}

```

```
        rollback_transaction(&sql_return_status);
    }
}
```

次の説明は、例 9-7 (9-10) の番号に対応しています。

- (1) ホスト文字変数を、NULL 終了記号を考慮して `length = length + 1` で宣言します。
- (2) ホスト変数 `sec` を、モジュール言語パラメータ `REAL` にあわせて浮動小数点数として定義します。間隔から `SECOND` フィールドを抽出すると SQL データ型 `INTEGER(2)` になり、これは C ではサポートされないためです。変数 `hr_rate` も、同じ理由により浮動小数点型として宣言します。
- (3) ストリーム (`SQLCODE = 100`) が終了するまで各従業員のデータをフェッチします。

このプログラムをコンパイルし、リンクし、実行すると、次のレポートが表示されます。

```
Enter START DATE (YYYY-MM-DD):1995-07-15
Enter END DATE   (YYYY-MM-DD):1995-07-19

          WEEKLY PAYROLL ( 1995-07-15 - 1995-07-19)

Emp Id      Last Name      Hours      Rate      Wages
00415       Mistretta     51:18:16.7  12.50     711.96
00416       Ames          48:34:16.7  12.50     660.71
```

9.3 日付時刻データ型を使用する場合の移植性の向上

複数の SQL 製品での処理を要するアプリケーションの場合、または SQL 標準に準拠するアプリケーションが必要な場合は、次の指針を考慮してください。

- 日付時刻データ型から個々のフィールドを取り出す場合は、`EXTRACT` 関数を使用します。次に例を示します。

```
SELECT  EXTRACT(YEAR FROM x), EXTRACT(MONTH FROM x)
        INTO      :years, :months
        FROM  DATE_TAB;
```

- 各フィールドをそれぞれの間隔に格納したり、それぞれの間隔の各フィールドを使用して式を評価する場合は、`CAST` 関数を使用します。次に例を示します。

```
UPDATE DATE_TAB
        SET  I_COL = CAST(:hours AS INTERVAL HOUR) +
                  CAST(:minutes AS INTERVAL MINUTE) +
                  CAST(:seconds AS INTERVAL SECOND);

SELECT ID, I_COL FROM  DATE_TAB
        WHERE  I_COL > CAST(:hours AS INTERVAL HOUR) +
                  CAST(:minutes AS INTERVAL MINUTE) +
                  CAST(:seconds AS INTERVAL SECOND);
```

- INSERT 文では、VALUES 句に式を指定できます。したがって、次のようにホスト・データを直接テーブルに挿入できます。

```
INSERT INTO x (LAST_NAME, AGE)
VALUES (:last_name, CAST(:age AS INTERVAL YEAR));
```

- 日付時刻フィールドをテキストとして取り出します。次に例を示します。

```
SELECT CAST(BIRTHDAY AS VARCHAR(30))
FROM EMPLOYEES;
```

得られたテキスト文字列は、間隔データ型と日付時刻データ型用に定義したレイアウトに準拠します。間隔の先頭のフィールドは、間隔の先頭のフィールドの精度で指定した桁数を使用します。常に符号があります（0 または正の間隔では空白、負の間隔ではマイナス符号）。

9.4 アプリケーションとデータベースの変換

アプリケーションとデータベースは、次のように日付時刻データ型を使用するように変換できます。

- データ型を DATE VMS から DATE ANSI、TIME、または TIMESTAMP に変換するには、ALTER DOMAIN 文を使用します。古いバージョンの行に格納されたデータは、Oracle Rdb で自動的に変換されます。
- DATE ANSI のリテラル値の代入は DATE VMS のリテラル値の代入とまったく同様に機能するわけではないので、ANSI 日付時刻書式を使用してリテラル値を指定するか、または CAST 関数を使用して DATE VMS から ANSI データ型に変換する必要があります。次に例を示します。

```
CAST(CAST(:host_char AS DATE VMS) AS TIMESTAMP)
```

- ANSI 日付時刻データ型で定義された列を述語 LIKE、CONTAINS、または STARTING WITH を使用して検索する場合、Oracle Rdb が使用する内部表現は次のリテラル構文と同等です。

```
'199501'
```

これらの関数は暗黙的にテキストに変換するので、DATE VMS データ型を使用した場合と同じ比較はできません。

9.5 アプリケーションにおける DATE VMS データ型の処理

アプリケーションで DATE VMS データ型を使用する場合は、対話型 SQL の場合と同様に文字列を代入できます。しかし、次のことに注意してください。

- DATE VMS 列に文字列のリテラルを代入する場合は、次のいずれかの書式を使用してください。

- dd-mmm-yy hh:mm:ss:cc

次の例では、DATE VMS 列に文字列値を挿入しています。

```
EXEC SQL INSERT INTO TIME_SHEET (START_TIME)
VALUES ('12-NOV-1979 12:34:56.78');
```

- yyyyymmddhhmmsscc

この書式を使用する場合は、次の例に示すように CAST 関数を使用して DATE VMS に変換してください。

```
EXEC SQL INSERT INTO TIME_SHEET (START_TIME)
VALUES (CAST ('1979111212345678' AS CHAR(16)));
```

- 文字列変数またはパラメータを使用する場合は、次の書式を使用してください。
yyyyymmddhhmmsscc

次の例では、ホスト言語変数に文字列を代入し、この変数を使用して DATE VMS 列にデータを挿入しています。

```
char st[17];
strcpy(st, "1979111212345678");
EXEC SQL INSERT INTO TIME_SHEET (START_TIME)
VALUES (:st);
```

次に示すプリコンパイルされた C プログラムの抜粋では、Oracle Rdb で DATE VMS と定義された列 TBL_DATE_VMS に文字列（16 文字の書式）を直接挿入する方法を示します。Oracle Rdb は、文字列から DATE VMS 書式への間隔の変換を自動的行います。この例では、逆の操作（DATE VMS データ型の TBL_DATE_VMS 列を 16 文字の書式に直接挿入する）も示します。

```
main()
{
    char    string_date[17];
    strcpy(string_date, "1989111212345678");
    EXEC SQL INSERT INTO ALL_DATE_TABLE (TBL_DATE_VMS)
        VALUES (:string_date);
    strcpy(string_date, "          ");
    EXEC SQL SELECT TBL_DATE_VMS INTO :string_date
        FROM ALL_DATE_TABLE;
    printf("Date VMS (16-character string format): %s\n", string_date);
}
```


9.5.1 OpenVMS に固有のアプリケーションで DATE VMS を使用する方法

場合によっては、日付時刻データが他のアプリケーションやソースから抽出されたり、他のレイヤ製品に送信されたりして ANSI/ISO 日付時刻データ型をサポートしないことがあります。このような場合は、CAST 関数を使用して文字列から DATE VMS 書式へ変換してください。OpenVMS に固有のアプリケーションでは、実行時ライブラリ関数を使用して国際的な日付と時刻を書式指定できます。

今後の Oracle Rdb のバージョンについても、次の CAST 関数からの出力が OpenVMS と互換性のある日付フォーマットであることを保証します。DATE VMS は、ANSI/ISO SQL 標準への Oracle Rdb 拡張です。

```
SELECT CAST(birthday AS DATE VMS), LAST_NAME, FIRST_NAME
FROM      ...;
```

9.5.2 DATE VMS データ型を使用するアプリケーションの移植

データ型 DATE VMS の列に格納されたデータを伴う OpenVMS 上で実行するアプリケーションでは、多くの場合、OpenVMS 実行時ライブラリ・ルーチンをコールして文字列とバイナリ表現間でデータを書式指定し、変換します。

これらのサービスは他のオペレーティング・システムでは提供されないので、他のオペレーティング・システム上で実行するアプリケーションは、日付の列を適切に処理するように変更する必要があります。次の 2 通りの方法が考えられます。

- データベースを ANSI 日付時刻型に変換します。これらの型は DATE VMS とは若干異なります。これは最適な方法ですが、他のアプリケーションに影響があるので適用できない場合があります。
- 実行時に日付を送受信する方法として、バイナリ表現ではなく文字表現を使用します。

ホスト・プログラムでは、8 バイトの文字変数を定義してバイナリの日付を返すことがたびたびあります。SQL では、8 バイトの文字変数と DATE VMS 列間の代入を変換なしの代入として処理します。バイナリ・データは、変数に渡されます。16 バイトの文字変数 (C プログラムでは char x[17]) は文字表現を保持するのに十分な長さなので、SQL は変換された値を変数に代入します。

16 文字のフォーマット (YYYYNNDDHHMMSSCC) を使用すると、2001 年 1 月 1 日午後 7:25 は次のようになります。

```
DATE VMS '2001010119250000'
```

この項の例で示すのは次のとおりです。

- 単純なストレージと DATE VMS データの取得
- 文字表現に関する操作

次のプログラムはユーザーが記述した関数を使用して DATE VMS フォーマットを DATE ANSI フォーマットに変換する OpenVMS システム・サービス・コールを置き換える方法を示します。結果を保持する変数は、8 バイトでなく 16 バイトです。ただし、システム・サービスへのコールを削除すると、OpenVMS 記述子は不要になります。

```
simple_date () {
    /* Note that because these variables are treated as character strings,
     * their size must include the null terminator for C.
     */
    char in_date[17];
    char ret_date[17];
    char string_date[MAX_DATE_SIZE];
    long SQLCODE;
    /* Ask the user for the date. */
    ReadInputDate(string_date);
    /* Convert the date to SQL character format. */
    ToSQLDate(string_date,in_date);
    EXEC SQL INSERT INTO SALARY_HISTORY (EMPLOYEE_ID,SALARY_START)
        VALUES ('09999', :in_date);
    check_sqlcode( &SQLCODE );
    EXEC SQL SELECT SALARY_START INTO :ret_date
        FROM SALARY_HISTORY WHERE EMPLOYEE_ID = '09999';
    check_sqlcode( &SQLCODE );
    /* Convert the date back to something that the user can read.*/
    FromSQLDate(string_date,ret_date);
    printf("Date: %s\n",string_date);
}
```

アプリケーションでは、特定のオペレーティング・システム用のルーチン ToSQLDate と FromSQLDate を記述できます。アプリケーションでは、ユーザー向けに日付をどう表現するかによって、オペレーティング・システムに固有の日付書式もすべてのオペレーティング・システムで一貫した日付書式も設定できます。

次の例では、変数のサイズを変更することで、SQL に対して変数を様々なデータ型として処理するように強制する方法を示します。例は SQL モジュール・プロシージャなので、パラメータのデータ型を DATE から CHAR に変更する必要があります。

次の例に示すように、オリジナルのコードでは 8 バイトの変数を使用して結果を保持します。

```
PROCEDURE INSERT_SAL_HIST
    (SQLCODE,
     :IN_DATE DATE VMS);
    INSERT INTO SALARY_HISTORY (EMPLOYEE_ID,SALARY_START)
        VALUES ('09999', :IN_DATE);
```

次の例に示すように、変数のサイズを 16 バイトに変更します。

```
PROCEDURE INSERT_SAL_HIST
  (SQLCODE,
   :IN_DATE CHAR(16) );
  INSERT INTO SALARY_HISTORY (EMPLOYEE_ID,SALARY_START)
    VALUES ('09999', :IN_DATE);
```

日付の書式を文字列に変換すると、この文字列を他の文字列と同様に処理できます。CAST 関数を使用するとデータベース内にある間にデータを変換できます。

たとえば、SALARY_START 列の日付の部分のみを取り出すには、次の文を使用します。

```
SELECT SUBSTRING( CAST( SALARY_START AS CHAR(16) ) FROM 1 FOR 8 )
  FROM SALARY_HISTORY;
```

この例では、CAST 関数を使用して DATE VMS データ型を文字表現に変換します。それから、得られた文字列を SUBSTRING 関数で処理します。

ただし、CAST 関数と連結を使用して日付を構成することもできます。次の例では、複数の文字列を連結して文字表現を構成しています。

```
INSERT INTO SALARY_HISTORY (SALARY_START)
  VALUES (CAST( :yr || :mm || :yy || '00000000' AS DATE VMS));
```

正しく変換するには、時間、分、秒、1/100 秒を表すゼロの文字列が必要です。

CAST 関数を使用して DATE VMS データ型を操作する例は、サンプル・ディレクトリの名前が "sql_load" で始まるプログラムを参照してください。

9.6 動的 SQL での日付時刻データ型の使用方法

動的 SQL でプログラムする場合は、動的 SQL で日付時刻データ型を使用する上で次の点に留意してください。

- 動的 SQL は、プリコンパイルされ、実行されるモジュールから DATE ANSI データ型設定を継承します。
- 日付時刻データ型では SQLDA2 構造を使用する必要があります。
- SQL では、CAST(? AS INTEGER) の形の式を他の式とは異なる方法で処理します。詳細は、9.6.1 項 (9-17) を参照してください。

9.6.1 パラメータ・マーカを伴う CAST の使用方法

SQL は、パラメータ・マーカを伴う CAST を使用した式を他の式とは異なる方法で処理します。SQL は、この種の式のコンテキストからはデータ型を抽出できません。したがって、SQL はパラメータ・マーカ (?) のデータ型が CAST 関数で指定したものと同じであると

解釈します。この場合は INTEGER です。したがって、次の例に示すように、2 つの CAST 関数が必要になります。

```
CAST(CAST(? AS CHAR(10)) AS INTEGER)
```

サンプル・ディレクトリ内の `sql_dynamic.c` プログラムには、日付時刻データ型と `SQLDA2` 構造を使用する方法を示します。このプログラムは、日付時刻データ型を文字列に変換してから動的 SQL 文 `PREPARE` と `EXECUTE` で処理する方法を示しています。したがって、2 つの CAST 関数を使用する必要はなくなります。

例 9-8 (9-18) に、サンプル・ディレクトリの `sql_dynamic` プログラムを使用した対話型のダイアログの一部を示します。この例では、SQL が論理演算子 (`>`) を正しく実行するには、`SELECT` 文でパラメータ・マーカ (`?`) を適切な日付時刻フォーマット (この場合は間隔の `HOURL`) にキャストする必要があることを示します。

例 9-8 動的 SQL プログラムにおけるパラメータ・マーカのキャスト

```
DynamicSQL> SELECT * FROM ADMINISTRATION.ACCOUNTING.DAILY_HOURS
1> WHERE HOURS_WORKED > ?;
The SQL statement to be executed dynamically is:
      SELECT * FROM ADMINISTRATION.ACCOUNTING.DAILY_HOURS WHERE HOURS_WORKED > ?;
Enter value for parameter 'HOURS_WORKED' Interval Hour (2) To Second (2): 10
Error -304 returned from open_cursor
Error message:
%RDB-E-ARITH_EXCEPT, truncation of a numeric value at runtime
-SQL-F-DATCONERR, Data conversion error for string '10'
-COSI-F-IVTIME, invalid date or time
Error returned from sql_dynamic()
Enter the SQL statement to process on the following line(s), terminating your
statement with a semicolon <;> literal<(<)<CR> or <CTL-Z> to exit>:
DynamicSQL> SELECT * FROM ACCOUNTING.DAILY_HOURS
1> WHERE HOURS_WORKED > CAST(? AS INTERVAL HOUR);
The SQL statement to be executed dynamically is:
      SELECT * FROM ADMINISTRATION.ACCOUNTING.DAILY_HOURS WHERE
HOURS_WORKED > CAST(? AS INTERVAL HOUR);
Enter value for parameter ''Interval Hour (2): 10
-----
Field EMPLOYEE ID:00415
Field START_TIME:[Timestamp (2)] 1995-07-17 07:15:06.62
Field END_TIME:[Timestamp (2)] 1995-07-17 17:45:24.19
Field HOURS_WORKED:[Interval Hour (2) To Second (2)] 10:30:17.57
No more records found.
.
.
.
```

9.6.2 日付をテキスト文字列として動的 SQL 文に渡す方法

Compaq Tru64 UNIX

動的 SQL には、YYYYNNDDHHMMSSCC 書式の日付文字列を Compaq Tru64 UNIX 上の DATE VMS 列に格納する場合に制限があります。ユーザーがデータをテキスト文字列として動的 SQL 文 EXECUTE に渡す場合 (SQL/ サービスは常に日付をテキストとして渡します)、文字列は不適切なバイナリの日付に変換されます。次のような文は使用できません。

```
INSERT INTO TBL (DATE_COL) VALUES ( ? )
```

かわりに、次の文を使用して日付に関するデータベース・システムに固有のクエリーを実行します。

```
INSERT INTO TBL (DATE_COL) VALUES (CAST( ? AS CHAR(16) ) )
```

この文は、前述の文よりも効率的です。将来のリリースではこの制限を解除する可能性があります。◆

第III部

実行時処理

第3部では、次の処理を実行する方法を説明します。

- 完了条件、予測できるエラーおよび予測できないエラー
- 動的 SQL を通した SQL 文の実行時処理

実行時エラーの処理

この章では、実行時エラーの検出、エラー・メッセージの取得、エラーからのリカバリまたはトランザクションのロールバックを行う方法を説明します。この章では、次の内容について説明します。

- エラー処理のために SQL が提供するオプション
- エラーに関して SQL 文の実行を監視
- エラー・メッセージの表示
- データベースの整合性に関するエラー処理と制約評価のタイミングの設定
- ロックの競合とデッドロック・エラーの処理
- データベース接続やトランザクションの開始の失敗によって発生するエラーの処理
- エラー処理におけるプログラムの移植性の向上

参考: この章では、SQL と、基盤のデータベース・システムが返すエラーのみを説明します。ホスト言語またはシステムの実行時エラーの処理方法については、プログラミング言語のドキュメントを参照してください。

10.1 SQL エラー処理の概要

どのようなプログラムでも、実行時エラーが発生する可能性があり、予期したエラーと予期しないエラーがあります。表 10-1 (10-2) は、このような 2 種類のエラーをまとめたものです。

表 10-1 実行時エラーの種類

エラーの種類	説明
予期したエラー	予期されるエラーは、2 つに分類されます。 ■ "no next row" ("no data") 条件などの完了条件であり、プログラムは、ループ処理を続行するか終了するかを判断します。この条件は、実質的なエラーではありませんが、通常の実行サイクルを制御します。 ■ デッドロック、ロックの競合、データベースの一貫性、重複値エラーなどの例外条件であり、時々発生する可能性があります、予期できません。多くのプログラムでは、このようなエラーが発生した場合、リカバリを行い、実行を続行します。
予期しないエラー	プログラムではリカバリできないエラーであり、算術計算の例外、誤ったデータベースの宣言、古いメタデータ（データ定義）の使用、データベースの破損などがあります。 プログラムで予期しないエラーが発生したら、エラーに関する情報を表示し、データの変更がある場合はロールバックを行い、実行を停止します。

SQL は、例外条件やエラーが返されても、プログラムを終了するシステム・ルーチンをコールしません。したがって、プログラムは予期しない処理を続ける可能性があり、これが原因となってさらに他のエラーが発生することもあります。このため、SQL 文では、完了条件や例外条件を常にチェックしてください。

完了条件や例外条件を処理するには、次の処理が必要です。

- SQL 文の実行ステータスを表す値を格納するためのパラメータを宣言する。
- 各 SQL 文の実行後に、パラメータに格納されている値をチェックする。
- 値に基づいて、処理する内容を指定する。
トランザクションのロールバックやプログラムの終了など。

表 10-2 (10-3) は、SQL のエラー処理手法を示します。

表 10-2 SQL のエラー処理手法

エラー処理手法	標準への準拠	説明
SQLSTATE パラメータ	Yes	SQL 文の実行ステータスを表す、5 文字の文字列を格納します。ANSI/ISO SQL 規格では、プログラムで SQLSTATE または SQLCODE（または両方）を宣言する必要があります。Oracle Rdb への SQL インタフェースにより、この規則が必要になります。 ステータス・パラメータには、SQLSTATE を使用することをお薦めします。新しく開発するアプリケーションでは、SQLSTATE を使用してください。
SQLCODE パラメータ	Yes ^{*1}	SQL 文の実行ステータスを表す整数値を格納します。ANSI/ISO SQL 規格では、プログラムで SQLCODE または SQLSTATE（または両方）を宣言する必要があります。Oracle Rdb への SQL インタフェースにより、この規則が必要になります。
GET DIAGNOSTICS 文	Yes	複合文で、直前に実行した文の実行内容に関する診断情報を取得します。
SIGNAL 文	Yes	複合文の例外条件に関する情報を提供します。
WHENEVER 文	Yes ^{*1}	後続のすべての SQL 文に対するエラー処理を提供します。WHENEVER 文の使用は、埋込み SQL に限定されます。
RDB\$MESSAGE_VECTOR アドレス配列	No	SQL 文の実行ステータスに関する情報を格納します。ANSI/ISO SQL 規格には、RDB\$MESSAGE_VECTOR 配列は含まれません。さらに、この配列は Oracle Rdb 内部データ構造であるため、Oracle Rdb のバージョンによって、戻り値が異なる可能性があります。 RDB\$MESSAGE_VECTOR 配列は、SQLCODE または SQLSTATE パラメータよりも詳細なエラー情報を提供します。RDB\$MESSAGE_VECTOR 配列は、推奨する方法では必要な情報を取得できない場合にのみ使用してください。
sql_get_message_vector ルーチン	No	最後の SQL 文のステータス情報を、RDB\$MESSAGE_VECTOR 配列から取得します。

表 10-2 SQL のエラー処理手法（続き）

エラー処理手法	標準への準拠	説明
sql_signal ルーチン	No	RDB\$MESSAGE_VECTOR 配列で返されたエラー情報を使用して、条件のシグナルを発行します。sql_signal ルーチンは、プログラムの条件ハンドラに基づいて、最初のエラーや続いて発生するエラーのメッセージを表示し、プログラムを続行または停止することができます。
sql_get_error_text ルーチン	No	プログラムにエラー・テキストを返します。RDB\$MESSAGE_VECTOR 配列を使用して返されたエラー情報を使用します。
SQL エラー処理ルーチン	Yes	SQL プログラムでは、次の 3 つのルーチンにより、ユーザー記述のエラー・ハンドラを使用できます。 ■ sql_register_error_handler（エラー・ハンドラの作成） ■ sql_get_error_handler（エラー・ハンドラの起動） ■ sql_deregister_error_handler（エラー・ハンドラの削除） 上記のルーチンによって、任意の数のエラー処理ルーチンを設定でき、SQL が提供するエラー処理オプションの上書きや機能格納が可能です。

^{*1} SQLCODE は、SQL89 ANSI/ISO SQL 規格に準拠していますが、SQL92 ANSI/ISO SQL 規格には準拠していません。WHENEVER 文は、SQLCODE を使用します。

サンプル・ディレクトリにあるサンプル・プログラムでは、エラー処理の様々な手法を参照できます。sql_terminate プログラムでは、様々なエラー処理手法を使用でき、他のプログラムよりも詳しいコメントが提供されています。

10.2 SQL 文の監視によるエラー検出

この項では、SQLSTATE ステータス・パラメータ、SQLCODE ステータス・パラメータ、プリコンパイルした SQL での WHENEVER 文、RDB\$MESSAGE_VECTOR 配列、SQL エラー処理ルーチンによるエラー検出などの使用方法について、完了条件と例外条件を含めて説明します。

次に、任意のステータス・パラメータを使用してエラーを監視するのに適用できるガイドラインの概要を示します。

- ANSI/ISO SQL 規格では、SQLSTATE または SQLCODE (または両方) をプログラムで宣言する必要があります。Oracle Rdb への SQL インタフェースにより、この規則が必要になります。ANSI/ISO SQL 規格に準拠するプログラムを記述する場合、単文および複合文を実行するたびに、SQLSTATE または SQLCODE パラメータに格納されている値をチェックしてください。
実行可能ではない文はステータス・パラメータに値を返さないの、実行不可能な文の後ではステータス・パラメータをチェックしないでください。たとえば、DECLARE CURSOR 文の後には、ステータス・パラメータの値をチェックしないでください。
- 多くの場合、エラー処理ルーチンは、トランザクションをロールバックします。ただし、ロールバックが成功するのは、トランザクションがすでに開始している場合のみであり、必ずしも開始しているとは限りません。さらに、ROLLBACK 文の成否は、SQLSTATE または SQLCODE の値に影響を与えます。したがって、SQL 文の実行をさらに続行する前に、元のエラーに関連するメッセージを表示してください。ロールバックに関するエラー処理の詳細は、10.6 項 (10-35) を参照してください。

参考: 『Oracle Rdb7 SQL Reference Manual』の付録には、SQL 通信領域 (SQL Communications Area : SQLCA)、SQLSTATE および SQLCODE ステータス・パラメータ、RDB\$MESSAGE_VECTOR 配列に関する説明が掲載されています。
また、SQL 文の章には、WHENEVER 文の説明が掲載されています。

10.2.1 SQLSTATE の使用方法

SQLSTATE ステータス・パラメータは、各 SQL 文の実行ステータスを表します。SQLSTATE ステータス・パラメータは、ANSI/ISO SQL 規格に準拠しています。この規格は、SQL 内でのステータス・コードの移植性を向上させることを目的としています。SQLSTATE ステータス・パラメータに対して標準で定義されているステータス値の数は、SQLCODE で定義されている数を上回りますが、個々の実装環境でさらにステータス値を定義することもできます。

新しいアプリケーションでは、SQLCODE でなく、SQLSTATE を使用してください。SQLSTATE を使用して完了コードを監視するには、表現形式 (dialect) を SQL92 に設定する必要がありますが、表現形式の設定に関係なく、エラーは SQLSTATE で参照できます。プログラムでは、単一文や複合文を実行した後で、SQLSTATE パラメータに格納されている値をチェックしてください。

SQLSTATE ステータス・パラメータは、5 文字の文字列として宣言してください。最初の 2 文字はクラス・コード、最後の 3 文字はサブクラス・コードです。たとえば、ステータス・パラメータの値が 00000 の場合 (クラス・コード 00 とサブクラス・コード 000)、SQL 文の

実行が成功したことを示します。ステータス・パラメータには、A ～ Z の大文字と 0 ～ 9 の数字を使用できます。ステータス・パラメータの値は、次の 2 つに分類されます。

- 標準定義のステータス・パラメータの値
ANSI/ISO SQL 規格で定義されたステータス・パラメータの値は、A ～ H の文字と 0 ～ 4 の数字で始まります。
- 実装定義のステータス・パラメータの値
実装定義のステータス・パラメータの値は、I ～ Z の文字と 5 ～ 9 の数字で始まります。
Oracle Rdb と SQL は、R と S で始まるコードを使用します。

『Oracle Rdb7 SQL Reference Manual』の付録には、SQLSTATE ステータス・パラメータの値と、対応する例外条件が掲載されています。プログラムでは、少なくとも次の内容をチェックしてください。

- SQLSTATE の値が '00000' の場合、SQL 文の実行が成功したことを示します。
- SQLSTATE の値が '01000' の場合、SQL 文で警告が発生したことを示します。
- SQLSTATE の値が '02000' の場合、SQL 文は、それ以上データを返せないことを示します。
- SQLSTATE の値が '00'、'01'、'02' 以外の値で始まる場合、SQL 文でエラーが発生したことを示します。このエラーは、処理または修正が必要ですが、致命的なエラーではありません。

大部分のプログラムでは、デッドロック、ロックの競合、接続の失敗、ロールバック、ロールバックできるアクティブ・トランザクションがない、などの条件をチェックして処理を行ってください。

複合文では、GET DIAGNOSTICS 文を使用して、最後に実行した SQL 文の SQLSTATE の値を返すことができます。詳細は、12.9 項 (12-22) を参照してください。

10.2.2 SQLCODE の使用方法

SQLCODE ステータス・パラメータは、各 SQL 文の実行ステータスを表します。ANSI/ISO SQL 規格で定義されている SQLCODE は、0 (成功)、100 (NO DATA)、負の値 (エラー) の 3 つのみです。これに加えて、Oracle Rdb への SQL インタフェースが提供する SQLCODE の値があります。SQLCODE の値の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

プログラムでは、各 SQL 文を実行した後で、SQLCODE の値をチェックしてください。複合文では、GET DIAGNOSTICS 文を使用して、最後に実行した SQL 文の SQLCODE の値を返すことができます。詳細は、12.9 項 (12-22) を参照してください。

SQLCODE パラメータは、個別に宣言することも、SQLCA と呼ばれる構造の一部として宣言することもできます。SQLCODE を個別に宣言する場合は、符号付きのロングワードとして

宣言してください（SQL モジュールをコールする C ホスト言語プログラムでの宣言や、プリコンパイルした C プログラムでの宣言については、8.12.2 項 (8-27) を参照してください）。

SQLCA (SQL 通信領域) は、SQL 文に関する情報をアプリケーション・プログラムに提供するデータ構造体です。SQLCA では、SQL 文の実行が成功したかどうかを参照できます。一部のエラーでは、SQL 文が成功しなかった場合、特定のエラーを提示します。SQLCA は、ANSI/ISO SQL 規格に含まれていません。

SQL のエラー処理では、次のような SQLCA のフィールドを使用できます。

- SQLCODE フィールド
- SQLERRD という名前の配列の 3 番目の要素。
この配列要素には、SQL 文が処理した行のカウントが格納されるか、FETCH 文では、カーソル内での現在の行位置のカウントが格納されます。SQLCA 構造体全体からは独立し、SQLERRD 配列を宣言することはできないことに注意してください。

SQLCA の一部として SQLCODE を宣言するかどうかは、SQL プリコンパイラを使用するか、SQL モジュール・プロセッサを使用するかで決まります。

- プリコンパイルしたプログラムでは、INCLUDE SQLCA 文を指定してください。SQL がブロック構造をサポートしない言語では、1 つのソース・ファイルに対して、SQLCODE を 1 つ独立して宣言するか、INCLUDE SQLCA 文を 1 つ使用してください。
- SQL モジュール言語を使用する場合、SQLCODE のフィールドを独立して宣言するか、SQLCA 全体を表すホスト言語構造内で宣言してください（SQLCA の明示的な定義の例は、『Oracle Rdb7 SQL Reference Manual』の付録に掲載されています）。
SQL 文の処理で SQLCA のすべてのフィールドを更新するには、SQL モジュールのプロシージャ・コールで使用するパラメータとして、ホスト言語構造のレコード名を指定します。指定したレコード名は、キーワード SQLCA に対応します。これは、SQL モジュールでコールされるプロシージャのパラメータ・リストで指定します。SQLERRD 配列の行カウントの値を使用しない場合は、一部のコールで、SQLCODE を示すフィールドのみを指定することもできます。このような場合は、コールするプロシージャのパラメータ・リストのキーワードとして、SQLCA でなく SQLCODE を指定してください。
SQL モジュールのプロシージャをプログラムがコールするときには、SQL ステータス・パラメータのホスト言語の宣言に、SQLCODE とは異なる名前を選択することができます。ただし、ソース・コードを見れば、ステータス・パラメータ SQLCODE の名前から、その機能が明確にわかります。コールごとに（SQL がブロック構造をサポートしない言語でプリコンパイルしたプログラムは除く）異なる SQL ステータス・パラメータを宣言する場合は、各ステータス・パラメータに異なる名前を付けてください。ネーミング規則として、プロシージャに関連する数字や名前を SQLCODE に追加すると、わかりやすいでしょう。

SQL モジュールをコールする C ホスト言語プログラムや、プリコンパイルした C プログラムでの SQLCA の宣言については、8.12.2 項 (8-27) を参照してください。

『Oracle Rdb7 SQL Reference Manual』の SQLCA の付録には、SQLCODE フィールドに格納される可能性のある数値または文字の値の一覧と、それぞれの値の意味が掲載されています。一般的に、各値には次のような意味があります。

- 数値の 0、またはリテラルの `SQLCODE_SUCCESS` は、文の実行が成功したことを示します。
- 正の整数は、完了コードです。文の実行は完了しましたが、何か異常が発生しました。最も重要なリテラルは、数値では 100、リテラルでは `SQLCODE_EOS` であり、行が検索できなかったことを示します。
- 負の整数や、エラーを示すリテラルは、文の実行が完了しなかったことを示します。

一般的に、プログラムでは、次のような条件や操作をチェックしてください。記述には、ホスト言語の文を使用します。

- SQL 文の実行が成功している間は (`SQLCODE = 0` または `SQLCODE_SUCCESS`)、一連の文を実行してデータを処理します。
- SQL 文で行を取り出せなくなるまで (`SQLCODE = 100` または `SQLCODE_EOS`)、一連の文を実行してデータを処理します。
- SQL 文の実行中にエラーが発生したら (`SQLCODE` が 0 未満)、一連のエラー処理文を実行します。
ホスト言語の文には、このような条件をいくつか指定できます。予期されるエラーについては、各エラーごとにプログラム・リカバリ・プロシージャを指定し、"else" によって予期しないエラーのエラー処理文を指定します。

`SQLCODE` を使ったエラー処理の方法は、例 10-1 (10-8) から例 10-3 (10-10) を参照してください。

例で示したホスト言語の構造は、ほとんどの言語で共通のネスト化した IF 文ブロックを示しています。この例は、特定のホスト言語の IF 文の書式や構文に従ったものではありません。各プログラミング言語には、EVALUATE (COBOL) や SWITCH (C) などの文が用意されており、IF 文よりも効率的にプログラム・フローの条件制御を実行できます。

例 10-1 (10-8) は、`SQLCODE` の値を監視して、FETCH ループを終了するかどうかを判断する方法を示しています。この例では、ホスト言語と SQL モジュールのコールを使用します。

例 10-1 SQL モジュール言語での `SQLCODE` の監視

```
Execute the following statements until SQLCODE equals 100:  
  Host language statements to initialize indicator parameters  
  Call to name-of-fetch-procedure passing SQLCODE and  
    a list of main parameters and indicator parameters  
    as call parameters  
  If SQLCODE is less than 0  
    then branch to error-handling-section
```



```

else
    Host language statements to evaluate indicator array
    items and manipulate fields in record
End of execution block

```

例 10-1 (10-8) で参照しているエラー処理セクションは、SQLCODE をチェックして、どのようなエラーが発生したかを判断し、適切な処理を実行します。プログラムがエラーのリカバリを実行できない場合、プログラムは、エラーのロギング、主なエラーとその後に発生したエラーのメッセージ表示、実行中のトランザクションのロールバックと終了など、クリーンアップ操作を行います。SQLCODE フィールドが負の値になっていないか、ホスト言語のステータス・フィールドに障害ステータスがないかを監視することにより、FETCH 文で致命的なエラーが発生したかどうかを判断できます。この時点で、プログラムはループを終了します。

ホスト言語の条件文のみを使用して SQL 文のステータスを監視する場合、各 SQL 文を実行した後に、ホスト言語プログラムの条件文を記述する必要があります。また、SQL モジュール・プロセッサを使用する場合は、SQL モジュールのプロシージャをコールした後に、条件文を記述する必要があります。

例 10-2 (10-9) は、どのような致命的エラーが発生しても、プログラムのクリーンアップと終了を実行する簡単な方法を示しています。

例 10-2 SQLCODE の監視とエラーでの終了

```

Precompiled SQL statement or SQL module procedure call
IF SQLCODE is less than 0
    branch to error-handler-1
End of IF block
.
.
.
error-handler-1
    Display SQLCODE value and error messages
    Roll back transaction
    IF SQLCODE is less than 0
        Display messages and continue
    End of IF block
    Close any files
    Stop program execution

```

データベースで発生する条件の多くは、プログラムを停止する必要のないものです。多くの場合、トランザクションを再開するか、他の対処措置を実行するだけで十分です。たとえば、データベースが存在しないために接続に失敗した場合は、新しいデータベース名をユーザーに入力してもらうことができます。例 10-3 (10-10) では、SQLCODE で複数の評価を実行し、その結果として、リカバリ処理を実行するか、プログラムを停止します。

例 10-3 SQLCODE の値を使ったりカバリ処理

```
Precompiled SQL statement or SQL module procedure call
IF SQLCODE is less than 0
    branch to error-handler-1
End of IF block
.
.
.
error-handler-1
    IF SQLCODE equals -1003 or -913 (lock conflict or deadlock error)
        Display or log messages about error
        Execute actions for locking problems
    else
        IF SQLCODE equals -803 (duplicate value in unique index error)
            Display or log messages about error
            Execute actions for duplicate value violations
        else
            IF SQLCODE equals -1002 or -1001 (violation of constraint)
                Display or log messages about error
                Execute actions for validation and integrity violations
            else
                Display or log message about unexpected error
                Execute orderly termination of transaction and program
            End of IF block
        End of IF block
```

プログラム・リカバリを実行するエラーの詳細は、10.4 項 (10-31) から 10.5 項 (10-32) を参照してください。

10.2.3 WHENEVER 文の使用法

プリコンパイルしたプログラムでは、SQL の WHENEVER 文を指定すると、それに続くすべての SQL 文のエラー処理を提供できます。SQL モジュール言語では、WHENEVER 文は使えません。次の例で示すように、WHENEVER 文は、個々の SQL 文ではなく、エラーのグループを処理します。

```
EXEC SQL WHENEVER SQLERROR GOTO label-of-section-handling-errors
EXEC SQL WHENEVER NOTFOUND GOTO label-of-section-handling-condition
EXEC SQL WHENEVER SQLWARNING GOTO label-of-section-handling-warnings
```

SQL プリコンパイラによる WHENEVER 文の処理では、ソース・ファイルの最後か、同じタイプの WHENEVER 文が見つかるまで、同じホスト言語の条件文を各 SQL 文の後に挿入します。たとえば、WHENEVER SQLERROR 文では、プリコンパイラは、実行可能な SQL 文の後に、次の IF 文をホスト言語で表した文を挿入します。

```
If SQLCODE is less than 0, GOTO same-section-for-handling-errors
```

したがって、使用するホスト言語の条件文を指定し、ホスト言語の文固有のエラー処理または条件処理を実行する必要があります。通常は、条件付きのエラー制御で例外条件を処理し、他の種類のエラーにもこれを使用します。特に、SELECT や FETCH 文、データベース・システムで制約を評価する文などには、文固有のエラー処理を記述してください（制約違反の処理方法は、10.4 項 (10-31) を参照してください）。

SQL の WHENEVER 文では、プリコンパイルしたプログラムのセクション名を指定します。このセクションが、各種エラーを処理します。

WHENEVER 文は、次のように、3 つの完了および例外の条件やエラーを検出します。

- 新しい行がない (NOT FOUND)
これは、SQLCODE = 100 または SQLCODE_EOS、SQLSTATE = '02000' に相当し、"no data" とも呼ばれます。
- 警告 (SQLWARNING)
これは、0 より大きい SQLCODE、'01' で始まる SQLSTATE に相当します。ただし、SQLCODE = 100 や SQLCODE_EOS、SQLSTATE = '02000' は除外します。
- 致命的なエラー (SQLERROR)
これは、0 より小さい値を含む SQLCODE、および '00'、'01'、'02' 以外で始まる SQLSTATE、さらに実行に失敗したことを判断するローカルのホスト言語の手法に相当します。

次の例では、1 つの WHENEVER SQLERROR 文で、SQL 文の A、B および C のエラーを処理します。次の WHENEVER SQLERROR 文では、statement-C の後にある SQL 文のエラー処理を行います。各 WHENEVER 文は、エラーを処理するプログラムのセクションのラベルを指定します。このセクションに含まれる文は、端末にフィールドの値やメッセージを表示し、手順通りにプログラムを停止するか、リカバリ処理を実行します。プログラムの実行では、statement-D でエラーが発生すると、error-handler-2 が実行されます。statement-B でエラーが発生すると、error-handler-1 が実行されます。

```
EXEC SQL WHENEVER SQLERROR GOTO label-of-error-handler-1
EXEC SQL statement-A
EXEC SQL statement-B
EXEC SQL statement-C
EXEC SQL WHENEVER SQLERROR GOTO label-of-error-handler-2
EXEC SQL statement-D
EXEC SQL statement-E
EXEC SQL statement-F
```

WHENEVER 文は、実行可能な文ではありません。WHENEVER 文による監視対象となる SQL 文は、実行時ではなく、プリコンパイルの時点で決まります。ソース・コードの中で、同じ条件を監視する 2 つの WHENEVER 文に挟まれているすべての SQL 文は、NOT FOUND、警告、エラーなどの検出対象となります。

上記の例では、プログラムを実行すると、プログラムのフローによっては、statement-E が statement-B よりも先に実行される可能性もあります。ただし、statement-B のエラーは必ず error-handler-1 で処理され、statement-E のエラーは必ず error-handler-2 で処理されます。

ホスト言語の条件文で SQL のエラー処理を行うときに、SQL 文の前に WHENEVER 文があり、同じ条件やエラーを処理してしまう場合、SQL 文に対する WHENEVER 文の制御をオフにする必要があります。これには、アクション CONTINUE を指定した WHENEVER 文を追加してください。たとえば、IF 文で SQL 文 statement-C を監視するときに、SQL 文 statement-C の後に発生する SQL 文については、WHENEVER で監視する場合、次のようになります。

```
EXEC SQL WHENEVER SQLERROR GOTO ERROR-HANDLER-1
EXEC SQL statement-A
EXEC SQL statement-B
EXEC SQL WHENEVER SQLERROR CONTINUE
EXEC SQL statement-C
IF SQLCODE < 0 ...
EXEC SQL WHENEVER SQLERROR GOTO ERROR-HANDLER-1
```

WHENEVER 文の GOTO 句に指定されているエラー・ハンドラに SQL 文を含めると、この SQL 文もエラー条件の対象になります（一般的には ROLLBACK 文）。この場合、WHENEVER 文が指定し、WHENEVER 文が監視してエラーを処理するセクションでは、SQL 文でエラーが発生してはなりません。このセクションの SQL 文でエラーが発生すると、WHENEVER 文とエラー処理のセクションの間で無限ループが発生します。このようなループは、アクション CONTINUE で WHENEVER 文を指定することで回避できます。次に例を示します。

```
EXEC SQL WHENEVER SQLERROR GOTO ERROR-HANDLER-1
.
.
.
ERROR-HANDLER-1
  Display messages
  EXEC SQL WHENEVER SQLERROR CONTINUE
  EXEC SQL ROLLBACK
  If SQLCODE < 0...
```

WHENEVER 文が実行する条件およびエラー・モニターは、ホスト言語の条件文でも実行できます。WHENEVER 文は、複数の SQL 文の実行ステータスを監視できますが、実行時のプログラム・フローには関係なく動作します。ホスト言語の条件文は、実行可能であり、実行時の条件分岐によるエラー処理が可能です。ただし、WHENEVER 文は複数の SQL 文を監視できるのに対して、ホスト言語の条件文で監視できるのは 1 つの文のみです。

ソース・プログラムに WHENEVER 文を追加する場合、プログラムをプリコンパイルする前に、ソース行をチェックしてください。この作業は、プログラムが長い場合や、既存のプログラムにソース行を追加または削除した場合には、特に重要です。各 SQL 文について

(条件やエラーを監視するホスト言語の条件文がプログラムに含まれているかどうかに関係なく)、次のような点を確認してください。

1. この SQL 文の前に、WHENEVER SQLERROR 文があるか
2. この SQL 文の前に、WHENEVER SQL WARNING 文があるか
3. この SQL 文の前に、WHENEVER NOT FOUND 文があるか

いずれかに当てはまる場合、GOTO 句で参照されているセクションの実行内容が、SQL 文に適切かどうかを判断してください。エラー処理が適切でない場合は、同じタイプの WHENEVER 文を追加し、不適切なエラー処理を実行しないか、別のエラー処理を指定します。WHENEVER 文で CONTINUE オプションを使用すると、ホスト言語の文を使ったエラー処理ができます。

変更する前に実行されていたエラー処理を再度指定する場合は、元の WHENEVER 文を、エラー処理を変更した SQL 文の後に指定してください。

プログラムをプリコンパイルする前に、WHENEVER 文で指定されているプログラム・セクションに、実行可能な SQL 文が含まれているかどうかをチェックしてください。含まれている場合、セクションと WHENEVER 文でループが発生しないように、その SQL 文の条件およびエラー処理を確実に実行してください。

10.2.4 sql_get_message_vector ルーチンと RDB\$LU_STATUS の使用方法

sql_get_message_vector ルーチンを使用すると、RDB\$MESSAGE_VECTOR 配列から情報を取得できます。sql_get_message_vector ルーチンは他のプラットフォームに移植できるので、Oracle Rdb プラットフォームから別のプラットフォームへのアプリケーションの移植作業が簡単になります。ただし、このルーチンは、ANSI/ISO SQL 規格に準拠していません。

RDB\$MESSAGE_VECTOR 配列は、SQL 文の実行ステータスに関する情報を提供します。RDB\$MESSAGE_VECTOR 配列の 2 番目の要素である RDB\$LU_STATUS は、SQLCODE ステータス・パラメータに相当するフィールドです。sql_get_message_vector ルーチンを使用してこのフィールドを監視すれば、SQL 文の成功や失敗を判断できます。

注意：RDB\$MESSAGE_VECTOR 配列は、Oracle Rdb 内部のデータ構造です。したがって、バージョンによって返す値が異なる可能性があります。RDB\$LU_STATUS フィールドは、SQLCODE および SQLSTATE ステータス・パラメータで必要な情報を取得できない場合にのみ使用してください。

ANSI/ISO SQL 規格には、配列 RDB\$MESSAGE_VECTOR またはフィールド RDB\$LU_STATUS は含まれません。

RDB\$MESSAGE_VECTOR 配列は、データベース・システムやシステム上の各種機能が返す主なエラーと、それに続いて発生するエラーのメッセージにアクセスできます。このメッセージを追加情報としてプログラムで利用すれば、SQL 文の失敗の原因となった問題を、さらに詳しく表示し、正確に特定できます。

sql_get_message_vector ルーチンの使用方法

sql_get_message_vector ルーチンは、RDB\$MESSAGE_VECTOR 配列から情報を取得しますが、プログラムで配列を明示的に宣言する必要はありません。

このルーチンは、ベクトル要素を受け取る変数のアドレスと、返すベクトル要素のインデックスの値という、2つの引数をとります。次の表は、インデックスの値と、各ベクトル要素に含まれる情報です。

インデックスの値	返す情報
1	ベクトルの引数の数
2	最後の SQL 文の主ステータス・コード
3	主メッセージに対する FAO 引数の数
4-20	後に続くメッセージがある場合は、ステータスを返す

Oracle Rdb は、sql_get_message_vector が返すインデックスを、RDB\$MESSAGE_VECTOR 配列のフィールドにマッピングします。インデックス 2 を RDB\$LU_STATUS フィールドにマッピングします。マッピングの詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

例 10-4 (10-14) は、SQL モジュールをコールし、sql_get_message_vector を使用して SQL 文のステータスを返す C プログラムの抜粋です。

例 10-4 sql_get_message_vector ルーチンの使用方法

```
.
.
.
/* Error handler, using sql_get_message_vector. */
get_msgvec( )
{
    int index;
    int status_code;
    int arg_cnt;
    /* Declare the literal for constraint violation status. */
    int RDB$_INTEG_FAIL;
    /* Get the message vector argument count. */
    index = 1;
    sql_get_message_vector(&arg_cnt, index);
```

```

/* Get the status code. */
index = 2;
sql_get_message_vector(&status_code, index);
    if (status_code == RDB$_INTEG_FAIL)
        printf("Constraint violation. ");
        printf("You are trying to insert a department code%yn");
        printf("that already exists in the table.");
        exit(1);
/* You can also check for follow-on arguments, if the arg_cnt is greater
 * than 1.
 */
}
main( )
{
    .
    .
    .
    insert_data (&SQLCODE, department_code, department_name, manager_id);
    if (SQLCODE != 0)
        get_msgvec();
}

```

例 10-4 (10-14) のように、インデックス 2 の値を調べるとき、シンボリック・コードの値が数値でないことをチェックしてください。

sql_get_message_vector ルーチンや、その他 SQL ルーチンのプロトタイプを C プログラムで宣言するには、インクルード・ファイル sql_rdb_header.h を使用できます。詳細は、10.3.4 項 (10-30) を参照してください。

RDB\$LU_STATUS フィールドの使用方法

推奨はできませんが、RDB\$LU_STATUS フィールドを直接使用することもできます。これには、RDB\$MESSAGE_VECTOR 配列を宣言します。

OpenVMS OpenVMS
VAX Alpha

OpenVMS では、SQL プリコンパイラを使用し、INCLUDE SQLCA 文を指定する場合、プリコンパイラは自動的に配列 RDB\$MESSAGE_VECTOR を宣言します。◆

Compaq Tru64 UNIX

Compaq Tru64 UNIX では、-s 'msgvec' スイッチを使用して、プリコンパイラが RDB\$MESSAGE_VECTOR 配列を自動的に宣言するように指定できます。このスイッチは、特殊文字であるドル記号 (\$) をサポートする言語コンパイラで使用できます。◆

プログラムでは、RDB\$MESSAGE_VECTOR 配列を明示的に宣言できます。『Oracle Rdb7 SQL Reference Manual』の付録の SQLCA とメッセージ・ベクトルの項では、配列 RDB\$MESSAGE_VECTOR を宣言する方法を説明しています。

プログラムで SQLCODE や SQLSTATE を使用しない場合も、いずれかを宣言し、SQL モジュールのプロシージャをコールするときには、フィールドをパラメータとして引き渡す必要があります。

RDB\$LU_STATUS フィールドを使用するには、シンボリック・コードとして値を宣言します。SQL 文を実行した後に RDB\$LU_STATUS の値をチェックするときには、シンボリック・コードの値が数値でないかどうかをチェックします。たとえば、RDB\$_NO_DUP の値をチェックし、制約違反を監視します。

プログラムは、RDB\$LU_STATUS の値と、Oracle Rdb または SQL 機能が返したエラーに対応するシンボリック・エラー・コードを比較します。チェックしたいエラーや条件は、シンボリック・エラー・コードとして明示的に宣言してください。シンボリック・エラー・コードは、符号なしのロングワードの外部値として宣言します。表 10-3 (10-16) は、SQL プリコンパイラがサポートする言語について、シンボリック・エラー・コードを宣言する方法を示しています。

表 10-3 埋込みホスト言語でのシンボリック・エラー・コードの宣言

言語	宣言
Ada	<pre>---Declare the sytem package. with system; -- Declare the symbolic error code. lock_conflict : system.unsigned_longword := system.import_value("RDB\$_LOCK_CONFLICT");</pre>
C	<pre>globalvalue RDB\$_LOCK_CONFLICT;</pre>
COBOL	<pre>05 RDB\$_LOCK_CONFLICT PIC S9(9) COMP VALUE IS EXTERNAL RDB\$_LOCK_CONFLICT.</pre>
FORTTRAN	<pre>INTEGER*4 RDB\$_LOCK_CONFLICT EXTERNAL RDB\$_LOCK_CONFLICT</pre>
Pascal	<pre>RDB\$_LOCK_CONFLICT : [value,external] INTEGER;</pre>
PL/I	<pre>DECLARE RDB\$_LOCK_CONFLICT GLOBALREF FIXED BINARY (31);</pre>

シンボリック・エラー・コードをプログラムでどのように使用するかによって、変数または定数として宣言してください。テストは、ホスト・プログラムの言語構文を使用して直接実行できます。

すべての致命的なエラーをチェックする場合は、ホスト言語の文の実行を監視するために、プログラムで宣言した同じステータス・フィールドを使用して、SQL 文の実行を監視できます。この場合、プログラム制御の転送先となるエラー処理セクションは、一連の文を実行できます。

制約違反が予想される場合や、SQLCODE や SQLSTATE ステータス・パラメータで取得できない情報を返したい場合は、RDB\$LU_STATUS の値を監視すると便利です。例 10-5 (10-17) では、RDB\$LU_STATUS フィールドの値を評価し、制約違反や重複値エラーが発生した場合に、リカバリ処理を実行します。

例 10-5 RDB\$LU_STATUS を使った制約違反のトラップ

```

Declaration of ret-status
Declaration of RDB$MESSAGE_VECTOR  ! The INCLUDE SQLCA statement can
                                     ! replace an explicit declaration
                                     ! of RDB$MESSAGE_VECTOR in
                                     ! precompiled programs.

Declaration of RDB$_NO_DUP
Declaration of RDB$_INTEG_FAIL
.
.
.
Precompiled SQL statement (or a call to a procedure in an SQL module)
Assignment of RDB$LU_STATUS to ret-status
IF ret-status does not equal success
    Branch to error-handler-1
End of IF block
.
.
.
error-handler-1
    Evaluate RDB$LU_STATUS
        When RDB$_NO_DUP
            Display or log messages about error
            Execute actions for duplicate value violations
        When RDB$_INTEG_FAIL
            Display or log messages about error
            Execute actions for validation and integrity violations
        When other values
            Display or log message about error
            Execute appropriate actions
    End of evaluate block

```

プログラム・リカバリを実行するエラーの詳細は、10.4 項 (10-31) から 10.5 項 (10-32) を参照してください。

OpenVMS OpenVMS
VAX Alpha

OpenVMS では、OpenVMS のランタイム・ライブラリ・ルーチンである LIB\$MATCH_COND をコールし、戻り値を評価します。RDB\$LU_STATUS フィールドと宣言したシンボリック・エラー・コードは、コール・パラメータとして、参照渡しが行われます。コールの後に、ホスト言語の条件文は、シンボリック・エラー・コードがある場合は、SQL 文の実行で返されたエラー・コードと一致するものはどれかを評価し、それによってプログラム・フローを決定します。

LIB\$MATCH_COND ルーチンは、エラー・コードの重大度をマスクするので、将来のソフトウェア・リリースでシンボリック・エラー・コードの重大度が変更された場合の対応は必要ありません。

次の例は、Pascal プログラムの抜粋であり、LIB\$MATCH_COND を使用して、返されたステータスの値を評価します。

```
[EXTERNAL] FUNCTION LIB$MATCH_COND
(ret_status:INTEGER; sym_name:[LIST] INTEGER):INTEGER;EXTERNAL;
begin
  (* Use LIB$MATCH_COND to determine which error has occurred. *)
  error := LIB$MATCH_COND (RDB$MESSAGE_VECTOR[2],
                           RDB$_DEADLOCK,           {1}
                           RDB$_LOCK_CONFLICT,        {2}
                           RDB$_NO_DUP,               {3}
                           RDB$_NOT_VALID,            {4}
                           RDB$_INTEG_FAIL,           {5}
                           RDB$_STREAM_EOF,           {6}
                           RDB$_NO_RECORD             {7}
                           );
```

LIB\$MATCH_COND ランタイム・ライブラリ・ルーチンの詳細は、OpenVMS システム・ルーチンのドキュメントを参照してください。◆

サンプル・ディレクトリには、オンライン・プログラム `sql_terminate` の各言語用のバージョンがあり、RDB\$LU_STATUS フィールドの宣言やシンボリック・エラー・コードに関する説明が収録されています。ホスト言語を使った実行ステータスの監視の詳細は、ホスト言語のドキュメントを参照してください。

注意: Ada では、ドル記号 (\$) を名前に使用できません。RDB\$ で始まる名前やシンボリック・コードについては、ドル記号のかわりに、アンダースコア (_) を使用してください。たとえば、RDB\$MESSAGE_VECTOR 配列は、RDB_MESSAGE_VECTOR として宣言および参照します。

10.2.5 SQL エラー処理ルーチンの使用方法

SQL は、エラー発生時にアプリケーションへ制御を返す処理を簡易化するために、3 つのルーチンを提供しています。この SQL ルーチンを使えば、任意の数のエラー処理ルーチンを設定でき、SQL が提供するエラー処理の上書きや機能拡張を可能にします。さらに、この SQL ルーチンでは、エラー処理ルーチンにコンテキスト固有のデータを引き渡すことができます。

SQL は、プリコンパイルした SQL プログラムやモジュール言語プログラムに対して、次のようなルーチンを提供しています。

- `sql_register_error_handler`
アプリケーションのエラー処理ルーチンのアドレスと、コンテキスト固有のデータのアドレスを登録します。

- `sql_get_error_handler`
現在登録されているエラー処理ルーチンのアドレスと、コンテキスト固有のデータのアドレスを取得します。
- `sql_deregister_error_handler`
現在のエラー処理ルーチンの登録を解除します。

この SQL ルーチンでは、SQL コードとは異なるエラー処理を実行する固有のエラー・ハンドラを登録できます。たとえば、制約違反が発生した場合に、SQL が返すメッセージとは異なるメッセージを表示するルーチンを作成し、メイン・プログラムからそのルーチンへ情報を引き渡すことができます。

SQL ルーチンは、メイン・プログラムからエラー処理ルーチンにコンテキスト固有のデータを渡すことができるので、1 つの共通したエラー処理ルーチンを使用して、様々なエラーを処理できます。この共通のエラー処理ルーチンは、渡されたデータを評価し、そのデータに基づいて、エラー・メッセージの表示、トランザクションのロールバック、その他の処理を実行します。

SQL エラー処理ルーチンを使用するには、次のような操作が必要です。

- `sql_register_error_handler` ルーチンをコールして、エラー・ハンドラの名前とコンテキスト固有のデータのアドレスを引数として渡します。次の例では、C 言語プログラムで `sql_register_error_handler` ルーチンをコールする方法を示しています。

```
sql_register_error_handler (appl_error_handler,&user_data)
```

他の言語で `sql_register_error_handler` ルーチンをコールする方法については、「SQL エラー処理ルーチンのコール (10-23)」(この項)を参照してください。

- エラー処理ルーチンは、4 つのパラメータを渡すように宣言してください。最初の 3 つのパラメータである `RDB$MESSAGE_VECTOR`、`SQLCODE`、`SQLSTATE` は、参照渡しを行います。4 番目のパラメータは、コンテキスト固有のデータのアドレスであり、値渡しを行います。次の例では、C 言語プログラムでエラー処理ルーチンを宣言する方法を示しています。

```
static
void appl_error_handler(
    RDB$MESSAGE_VECTOR *msgvec,
    int                 *sqlcode,
    char                *sqlstate,
    void                *user_info)
```

他の言語でエラー処理ルーチンを宣言する方法については、「ユーザー記述のエラー処理ルーチンの宣言 (10-25)」(この項)を参照してください。

アプリケーションで実行した SQL 文がエラーを返した場合、SQL は、現在登録されているエラー処理ルーチンを実行します。このルーチンは、メッセージの表示、処理 (`SQLCODE` の値の変更を含む)、アプリケーションの終了などを実行できます。

注意：エラー処理ルーチン内では、例外でシグナルを発行しないでください。無限ループが発生します。SQL エラー処理ルーチンを使用すると、SQL は、例外シグナルを遮断し、ユーザー記述のエラー・ハンドラをコールします。ユーザー記述のエラー・ハンドラは、SQL が遮断した例外のシグナルを発行します。このサイクルは、無限ループで繰り返します。

エラー処理ルーチンが、SQLCODE の値を 0 に変更すると、実行の成功を意味するため、プログラムは SQL に制御を返します。

エラー処理ルーチンが、SQLCODE の値を 0 に変更しない場合や、エラー処理ルーチンが登録されていない場合、SQL は、プログラム内の WHENEVER 文を評価します。次に、プログラムは SQL に制御を返します。

プログラムから SQL に制御を返さない場合は、エラー処理ルーチンで、次のような処理を行ってください。

1. エラー処理ルーチンの登録を解除し、それに続く ROLLBACK 文がエラー処理ルーチンをコールしないようにします。次の例では、ルーチンの登録を解除する方法を示します。

```
sql_deregister_error_handler ( );
```

2. トランザクションをロールバックします。次の文は、プリコンパイルした C プログラムで、トランザクションをロールバックします。

```
EXEC SQL ROLLBACK;
```

3. イメージを終了します。このとき、SQL と Oracle Rdb は、ハンドラの実行を終了します。C プログラムでは、次の文を使用します。

```
exit;
```

sql_get_error_handler ルーチンは、1 つのアプリケーションで、複数のエラー処理ルーチンを使用できます。たとえば、メイン・プログラムとは異なるエラー処理ルーチンを使用するサブルーチンがプログラム内にある場合、サブルーチンは sql_get_error_handler をコールし、現在登録されているルーチンとコンテキスト固有のデータのアドレスを取得します。次に、このサブルーチンは、アドレスを変数に格納し、専用のエラー処理ルーチンを登録します。サブルーチンは、メイン・プログラムに制御を返す前に、変数に格納されている値を使用して、メイン・プログラムのルーチンとデータを再度登録します。

SQL エラー処理ルーチンやその他 SQL ルーチンのプロトタイプを C プログラムで宣言するには、インクルード・ファイル sql_rdb_headers.h を使用できます。詳細は、10.3.4 項 (10-30) を参照してください。

例 10-6 (10-21) では、プリコンパイルした C プログラムで SQL エラー処理ルーチンを使用する方法を示しています。

例 10-6 SQL エラー処理ルーチンの使用方法

```

/* This program demonstrates the use of the SQL error-handling routines,
 * sql_register_error_handler, sql_deregister_error_handler, and
 * sql_get_error_handler.  Although the use of the sql_get_error_handler
 * routine is not necessary in this simple program, it is included here
 * to demonstrate how to use the routine and how to store the address of the
 * currently registered routine and the address of user data in variables.
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sql_literals.h>
#include <sql_rdb_headers.h>
/* Definition of rdb$message_vector. */
typedef struct {
    int  RDB$LU_NUM_ARGUMENTS;
    int  RDB$LU_STATUS;
    int  RDB$LU_ARGUMENTS[18];
} RDB$MESSAGE_VECTOR;
/* Definition of structure to hold user data. */
typedef struct {
    char sql_proc_name[31];
    char sql_col_value[31];
} err_struct;
/* Error-handling routine for constraint violations.  This routine traps
 * constraint violations and prints out an error message.  */
static
void dupl_error_handler(
    RDB$MESSAGE_VECTOR *msgvec,
    int                 *sqlcode,
    char                *sqlstate,
    void                *user_info)
{
    err_struct *my_info;
    my_info = (err_struct *)user_info;
    if ((*sqlcode == SQLCODE_INTEG_FAIL) &&
        ((strcmp(my_info->sql_proc_name, "INSERT_JOBS")) == 0))
    {
        printf(" The Job Code %s is already in use.\n", my_info->sql_col_value);
    }
}
/* You can add more conditional statements to this error-handling procedure
 * to handle errors from several SQL statements.  */
}
/* Error-handling routine for errors that occur when you start a transaction.
 * This routine prints out an error message.  */
static

```

```
void txn_error_handler(
    RDB$MESSAGE_VECTOR *msgvec,
    int                  *sqlcode,
    char                 *sqlstate,
    void                 *user_info1)
{
    if ((*sqlcode == SQLCODE_DEADLOCK) || (*sqlcode == SQLCODE_BAD_TXN_STATE)
        || (*sqlcode == SQLCODE_LOCK_CONFLICT))
        printf("Unable to start a transaction. %n");
}

main( )
{
    /* Variables used by the main program. */
    void      (*rtn_ptr)();
    err_struct *err_struct_ptr = 0;
    char      j_code[5];
    char      w_class[2];
    char      j_title[21];
    char      release_screen;

    /* Define the SQLCA. */
    EXEC SQL INCLUDE SQLCA;

    /* Initialize user-defined information. */
    err_struct err_s = {" ", " "};

    /* Declare the database. */
    EXEC SQL DECLARE ALIAS FILENAME 'PERSONNEL';

    /* Register the first error-handling routine. */
    sql_register_error_handler(txn_error_handler,0);

    /* Store the address of the currently registered pointer in a variable. */
    sql_get_error_handler(&rtn_ptr, &err_struct_ptr);
    printf("Please enter the Job Code (or EXIT):%n");
    scanf(" %s", j_code);
    release_screen = getchar();
    while (((strcmp(j_code,"exit")) != 0) &&
           ((strcmp(j_code,"EXIT")) != 0))
    {
        printf("Enter the Wage Class: ", w_class);
        scanf(" %s", w_class);
        release_screen = getchar();
        while (((strcmp(w_class,"1")) !=0) &&
               ((strcmp(w_class,"2")) !=0) &&
               ((strcmp(w_class,"3")) !=0) &&
               ((strcmp(w_class,"4")) !=0))
        {
            printf("Please enter one of the following values for Wage Class:%n");
            printf(" 1  2  3  4%n");
            scanf(" %s", w_class);
            release_screen = getchar();
        }
    }
}
```

```

    }
    printf("Please enter the Job Title: ¥n");
    scanf(" %s", j_title);
    release_screen = getchar();
/* Start a transaction. */
    EXEC SQL SET TRANSACTION READ WRITE NOWAIT
                EVALUATING JOB_CODE_PRIMARY AT VERB TIME
                RESERVING JOBS FOR EXCLUSIVE WRITE;
/* Register the second error-handling routine. */
    sql_register_error_handler(dupl_error_handler, &err_s);
/* Store information in a structure for use by the error-handling routine. */
    strcpy(err_s.sql_proc_name, "INSERT_JOBS");
    strcpy(err_s.sql_col_value, j_code);
    EXEC SQL INSERT INTO JOBS
                (JOB_CODE, WAGE_CLASS, JOB_TITLE)
                VALUES
                (:j_code, :w_class, :j_title );
    if (SQLCA.SQLCODE == SQLCODE_SUCCESS)
        EXEC SQL COMMIT;
    else
        EXEC SQL ROLLBACK;
/* Deregister the error-handling routine. */
    sql_deregister_error_handler();
    printf("Please enter the Job Code (or EXIT):¥n");
    scanf(" %s", j_code);
    release_screen = getchar();
/* Register the txn_error_handler routine again. Use the address stored in
*   rtn_ptr. */
    sql_register_error_handler(rtn_ptr, 0);
}
return;
}

```

共有可能イメージから `sql_register_error_handler` ルーチンを起動する場合、共有可能イメージを、`sql_register_error_handler` ルーチンをコールするジャケット・ルーチンにリンクする必要があります。次は、C 言語のジャケット・ルーチンの例です。

```

extern void call_sql_regis_err_hand ( void (*user_error_handler) (),
                                     context_block_t *context_block)
{
    sql_register_error_handler( user_error_handler, context_block);
    return;
}

```

共有可能イメージを複数作成する場合、すべてのイメージが同じエラー・ハンドラを共有できますが、個々のイメージにエラー・ハンドラを登録する必要があります。

SQL エラー処理ルーチンのコール

この項では、SQL がサポートするホスト言語において、エラー処理ルーチンをコールする方法について説明します。

■ Ada

Ada では、SQL エラー処理ルーチンをコールする前に、プロシージャとして宣言する必要があります。次に、プロシージャを宣言する例を示します。

```
procedure sql_deregister_error_handler; -- no arguments
pragma interface( external, sql_deregister_error_handler );
pragma IMPORT_PROCEDURE (
    INTERNAL => sql_deregister_error_handler );
procedure sql_get_error_handler(
    handler : out system.address;
    context : out system.address );
pragma interface( external, sql_get_error_handler );
pragma IMPORT_PROCEDURE (
    INTERNAL => sql_get_error_handler,
    PARAMETER_TYPES => (system.address, system.address),
    MECHANISM => (REFERENCE,REFERENCE) );
procedure sql_register_error_handler(
    handler : in system.address;
    context : in system.address );
pragma interface( external, sql_register_error_handler );
pragma IMPORT_PROCEDURE (
    INTERNAL => sql_register_error_handler,
    PARAMETER_TYPES => (system.address, system.address),
    MECHANISM => (VALUE,VALUE) );
```

次に、プロシージャをコールする例を示します。

```
pkg_handler.sql_deregister_error_handler;
pkg_handler.sql_get_error_handler(rtn_ptr, foo_ptr);
pkg_handler.sql_register_error_handler(
    pkg_other.my_error_handler'ADDRESS,
    foo'ADDRESS );
```

■ BASIC

```
EXTERNAL LONG FUNCTION BASIC_ERRHND
CALL sql_deregister_error_handler
CALL sql_get_error_handler BY REF (ERR_HAND_ADDR, ERR_USER_CONTEXT)
CALL sql_register_error_handler &
    (LOC(BASIC_ERRHND) BY VALUE, REGISTERED_CONTEXT BY REF)
```

BASIC は、SQL モジュール言語のみでサポートされています。

■ C

```
sql_deregister_error_handler();
sql_get_error_handler(&appl_error_handler_ptr,&user_data_ptr);
```



```
sql_register_error_handler(&appl_error_handler,&user_data);
```

■ COBOL

```
01 appl_error_handler USAGE POINTER VALUE EXTERNAL MY_ERROR_HANDLER.
01 appl_error_handler_ptr USAGE POINTER VALUE 1.
01 user_data_ptr USAGE POINTER VALUE 1.
CALL "sql_deregister_error_handler".
CALL "sql_get_error_handler" USING BY REFERENCE appl_error_handler_ptr,
                                BY REFERENCE user_data_ptr.
CALL "sql_register_error_handler" USING BY VALUE appl_error_handler,
                                BY REFERENCE user_data.
```

■ FORTRAN

```
CALL sql_deregister_error_handler()
CALL sql_get_error_handler(%ref(rtn_ptr),%ref(ctx_ptr))
CALL sql_register_error_handler(%ref(my_error_handler),%ref(reg_ctx))
```

■ Pascal

Pascal では、SQL エラー処理ルーチンをコールする前に、外部プロシージャとして宣言する必要があります。次に、プロシージャを宣言する例を示します。

```
PROCEDURE sql_deregister_error_handler; EXTERNAL;
PROCEDURE sql_get_error_handler(
  var routine :      integer;
  var data :        integer
); EXTERNAL;
PROCEDURE sql_register_error_handler(
  procedure err_hand (
    var msgvec :      RDB$MESSAGE_VECTOR_REC;
    var sqlcode :     integer;
    var sqlstate :    sqlstate_type;
    var data_addr :   user_data_type);
  var data:          user_data_type
); EXTERNAL;
```

次に、プロシージャをコールする例を示します。

```
sql_deregister_error_handler;
sql_get_error_handler(routine_addr, data_addr);
sql_register_error_handler(%immed my_error_handler,user_data);
```

■ PL/I

PL/I では、エラー処理ルーチンをコールする前に、外部プロシージャとして宣言する必要があります。

ユーザー記述のエラー処理ルーチンの宣言

SQL プリコンパイラやホスト言語プログラムを使用して、ユーザー記述のエラー処理ルーチンを宣言するには、次のような書式を使用します。

■ Ada

```
procedure my_error_handler (  
    msgvec : in pkg_handler.rdb_message_vector;  
    sqlcode : in out integer;  
    sqlstate : in integer;  
    user_info : in pkg_handler.foo_t );)
```

■ BASIC

```
SUB BASIC_ERRHND BY REF (INTEGER MSG_VECTOR, &  
                        SQLCODE_VALUE, &  
                        SQLSTATE_VALUE, &  
                        CTX_REC USER_CTX )  
  
RECORD CTX_REC  
    STRING    VALUE1 = 9  
    STRING    VALUE2 = 10  
    INTEGER    VALUE3  
END RECORD CTX_REC  
DECLARE INTEGER SQLCODE  
SQLCODE = SQLCODE_VALUE)
```

BASIC は、SQL モジュール言語のみでサポートされています。

■ C

```
static  
void appl_error_handler (  
    RDB$MESSAGE_VECTOR *msgvec,  
    int *sqlcode,  
    char *sqlstate,  
    void *user_info)
```

■ COBOL

```
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 SQLCODE PIC S9(9) COMP.  
LINKAGE SECTION.  
01 MSG_VECTOR USAGE POINTER.  
01 SQLCODE_VALUE USAGE POINTER.  
01 SQLSTATE_VALUE USAGE POINTER.  
01 USER_CTX.  
    02 VALUE1 PIC X(9) .  
    02 VALUE2 PIC X(10) .  
    02 VALUE3 PIC 9(9) USAGE COMP.  
PROCEDURE DIVISION USING MSG_VECTOR, SQLCODE_VALUE, SQLSTATE_VALUE, USER_CTX.  
HANDLE_ERROR.  
    MOVE SQLCODE_VALUE TO SQLCODE.
```

■ FORTRAN

```

subroutine my_error_handler (msgvec,sqlcode,sqlstate,usrctx)
integer*4 msgvec
integer*4 sqlcode
integer*4 sqlstate
structure /user_ctx/
  character*10 value1
  character*10 value2
  integer*4 value3
end structure
record /user_ctx/ usrctx)

```

- Pascal

```

procedure my_error_handler (
  var msgvec :      RDB$MESSAGE_VECTOR_REC;
  var sqlcode :      integer;
  var sqlstate :      sqlstate_type;
  var data :          user_data_type);)

```

10.3 エラー・メッセージの表示

エラー・メッセージの表示は、きわめて一般的なエラー処理操作の1つです。SQL には次の方法があります。

- `sql_signal` ルーチンをコールし、あらゆる機能からのメッセージを表示し、(任意指定で) プログラムを終了します。
- `sql_get_error_text` ルーチンをコールして、あらゆる機能が発行したメッセージのエラー・テキストをプログラムに渡し、プログラムでこれを処理します。

さらに、ホスト言語ルーチンまたはシステム・サービス・コールを使用してあらゆる機能からのメッセージを表示してプログラムの実行を継続するか、プログラム用に作成されたメッセージ・ファイルのメッセージを表示することもできます。

サンプル・ディレクトリ内のソース・プログラムには、様々な技法を使用してエラー・メッセージを表示する例があります。

10.3.1 `sql_signal` のコール

`sql_signal` ルーチンは、`RDB$MESSAGE_VECTOR` 配列を介して返されるエラー・メッセージを表示します。`sql_signal` がオペレーティング・システムの条件ハンドラでなくプログラムの条件ハンドラにエラーを通知するため、次の場合には利点があります。

- プログラムを記述する言語によって、ホスト言語の文の条件ハンドラが自動的に提供される場合。
たとえば、OpenVMS の DEC COBOL では条件ハンドラが提供されています。エラーが通知された場合は、これを迂回しないでください。

- プログラムで独自の条件ハンドラを確立し、エラーが通知されたときは必ずその条件ハンドラで制御する場合。

Compaq Tru64 UNIX

Compaq Tru64 UNIX では、エラーが発生すると `sql_signal` ルーチンがエラー・メッセージを印刷してプログラムを終了します。◆

OpenVMS VAX OpenVMS Alpha

OpenVMS では、`sql_signal` ルーチンが SQL 文実行時に発生したエラーをプログラムの条件ハンドラに通知します。プログラムに条件ハンドラがない場合は、エラーが発生すると `sql_signal` ルーチンがエラー・メッセージを印刷してプログラムを終了します。

プログラムの条件ハンドラが RDB、SQL、またはリポジトリ・ファシリティ・コードによるエラーを条件ハンドラで処理できるエラーと認識しない場合は、プログラムの条件ハンドラから OpenVMS の条件ハンドラに再通知します。この状況は、プログラムの条件ハンドラがご使用のプログラム言語によって自動的に提供される場合にたびたび発生します。OpenVMS の条件ハンドラは、メッセージを表示し、エラーが重大な場合はプログラムを停止し、それ以外の場合はプログラムを続行します。

エラー通知に関する概念の詳細は、OpenVMS 実行時ライブラリのドキュメントで条件ハンドラの章を参照してください。◆

C プログラムで `sql_signal` ルーチンと SQL ルーチンのプロトタイプを宣言する場合は、インクルード・ファイル `sql_rdb_headers.h` を使用します。詳細は、10.3.4 項 (10-30) を参照してください。

`sql_signal` ルーチンには、返す値も必要なパラメータ也没有。次の例は、C 言語プログラムで `sql_signal` ルーチンを使用する方法を示しています。

```
sql_signal();
```

`sql_signal` をコールする書式は言語に固有です。様々なプログラミング言語でこのルーチンをコールする方法は、『Oracle Rdb7 SQL Reference Manual』のルーチンに関する章を参照してください。

10.3.2 `sql_get_error_text` のコール

書式付き ASCII 出力 (FAO) 置換のエラー・テキストをプログラムに渡して処理する場合は、`sql_get_error_text` ルーチンを使用します。

`sql_get_error_text` ルーチンを使用するには、SQL から渡されるテキストを受け取るバッファ (フィールド) をプログラムで宣言する必要があります。RDB\$LU_STATUS フィールドのメッセージと後続のすべてのメッセージが十分に入る長さのテキスト文字列としてこのフィールドを宣言します。オプションで、バッファ長を個別のフィールドとして宣言できます (符号付きワードとして)。

`sql_get_error_text` ルーチンの宣言には、次の 3 つの引数を伴います。

- テキストを受け取るバッファ

- テキストを受け取るバッファの長さ
- 返されるエラー・メッセージに割り当てられた文字数

次の例は、C プログラムで `sql_get_error_text` ルーチンを宣言し、使用方法を示しています。

```
/* This function uses the sql_get_error_text routine to display the messages
   returned by SQL and Rdb for unexpected error conditions. */
void display_sqlget_message()
{
    char    get_error_buffer[1024];
    long    error_msg_len;
    return_status = sql_get_error_text(get_error_buffer, 1024, &error_msg_len);
    get_error_buffer[error_msg_len] = '\0';
    printf("\n\nThis condition was not expected.\n\n");
    printf("*.s\n", get_error_buffer, error_msg_len);
    release_screen = getchar();
    printf("\n");
    return;
}
```

様々なプログラミング言語で `sql_get_error_text` ルーチンを宣言し、使用方法は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

ただし、ルーチンは改行と LF 文字を返し、初期のメッセージと後続のメッセージ、または後続のメッセージ同士を分離します。

C プログラムで `sql_get_error_text` ルーチンと SQL ルーチンのプロトタイプを宣言する場合は、インクルード・ファイル `sql_rdb_headers.h` を使用します。詳細は、10.3.4 項 (10-30) を参照してください。

`sql_get_error_text` ルーチンは、テキストを受け取るために宣言されたバッファにメッセージ間の区切りとして文字を挿入します。通常は、区切り文字によってテキストを端末画面に表示しやすくなります。

ただし、プログラムが `forms` 製品を使用してメッセージを表示する場合は、改行と LF 文字は印刷されない文字として処理されます。

次に示す COBOL の例は、バッファ内の改行と LF 文字を処理する 1 つの方法を示しています。

```
* CRLF is a PIC XX field that contains <cr><lf>.
* MSG-TXT-RDBFEL is an array of lines to be displayed for the error message.
*
      STRING CARRIAGE-RET, LINE-FEED DELIMITED BY SIZE INTO CRLF.
      CALL "sql_get_error_text" USING BY REFERENCE BUFFER,
                                      BY VALUE BUF_LEN
```

```
BY REFERENCE MSG_LEN.  
UNSTRING BUFFER DELIMITED BY CRLF INTO MSG-TXT-RDBFEL(1),  
MSG-TXT-RDBFEL(2), MSG-TXT-RDBFEL(3).
```

OpenVMS
VAX ≡≡≡ OpenVMS
Alpha ≡≡≡

OpenVMS 上では、SQL は SQL\$GET_ERROR_TEXT ルーチンもサポートしますが、他のプラットフォームには移植できないので、Oracle Rdb では sql_get_error_text ルーチンの使用をお勧めします。◆

10.3.3 ユーザーが提供するエラー・メッセージの表示

プログラムで端末への書込みを行うホスト言語の文を使用すると、リテラルをエラー・メッセージとして表示できます。

ユーザー用にカスタマイズされたメッセージ・ファイルからメッセージを表示することもできます。

OpenVMS
VAX ≡≡≡ OpenVMS
Alpha ≡≡≡

この種のファイルの編集と処理については、OpenVMS ドキュメント・セットのメッセージ・ユーティリティについて解説するマニュアルを参照してください。◆

メッセージ・ファイルの例は、サンプル・ディレクトリ内の sql\$persmsg.msg ファイルを参照してください。sql\$persmsg.msg ファイルには、プログラム sql_terminate.sco が関連します。sql_terminate.sco プログラムには、sql\$persmsg のイメージ・バージョンからメッセージを取り出すためのパラメータ宣言とコールが記述されています。

10.3.4 インクルード・ファイルを使用した SQL ルーチン

C プログラムで特定の SQL ルーチンをコールすると、プログラムはコンパイル情報に関するメッセージを生成できます。たとえば、sql_signal を関数として明示的に定義しない場合は、次の情報を提供するメッセージが表示されます。

```
          sql_signal();  
          ^  
          .....  
%CC-I-IMPLICITFUNC, In this statement, the identifier "sql_signal" is  
  implicitly declared as a function.
```

Oracle Rdb では、明示的にコールされた SQL 関数のプロトタイプを明示的に提供することによって、メッセージを消去するインクルード・ファイルが提供されています。

OpenVMS
VAX ≡≡≡ OpenVMS
Alpha ≡≡≡

OpenVMS のインクルード・ファイルを次に示します。

SYS\$LIBRARY:SQL_RDB_HEADERS.H ◆

Compaq Tru64 UNIX
≡≡≡

Compaq Tru64 UNIX のインクルード・ファイルを次に示します。

/usr/lib/dbs/sql/v<nn>/include/sql_rdb_headers.h ◆

10.4 重複値エラーと制約違反の処理

リレーショナル・データベースの整合性は、列に格納された有効な値と、様々なテーブルに格納された行の間で保持される意味のある関係にもとづいています。Oracle Rdb データベースでは、テーブル定義の制約によって以上の要件を実現します。列に格納する値が一意であるという要件は、テーブル定義の制約かインデックス定義の UNIQUE キーワードで実現されます。

この項では、次の内容について解説します。

- プログラムで発生する制約違反のエラーと一意のインデックスに重複値を格納しようとするエラー
- トランザクション実行中に制約が評価された場合の制御オプション

例 10-5 (10-17) は、制約違反をチェックする疑似コードの例を示しています。

10.4.1 制約違反と重複値エラーのステータス値

SQL では、すべてのデータの妥当性に関する要件をテーブル作成時に制約として定義します。プログラムでテーブル更新時に制約違反が発生すると、データベース・システムは IRDB\$LU_STATUS に RDB\$_INTEG_FAIL、SQLCODE に -1001、SQLSTATE に '23000' という値を返します。

ただし、列の値が一意という要件がある場合や、列に基づく一意インデックスがある場合は、データベース定義者がインデックス定義のみによる一意性の要件を指定することも少なくありません（少なくともテーブルをロードした後と一意のインデックスを配置した後）。この場合、プログラムでは SQL が一意のインデックスを重複値で更新しようとする場合にのみエラーが発生します。このエラーで返される値は、RDB\$LU_STATUS に RDB\$_NO_DUP、SQLCODE に -803、SQLSTATE に 'R2000' です。

一意性の要件が制約とインデックス定義の両方で指定された場合、プログラムで制約違反と重複値違反のどちらが先に発生するかは確定できません。

注意: バージョン 2.0 より前の SQL では、データベース制約オプションでなく列の "valid if" オプションによる NOT NULL 要件が実装されています。SQL 以外のインタフェースでデータベースのドメインとテーブルを定義する場合、これらのインタフェースは現在 "valid if" オプションを使用することで NOT NULL と列に関する他の妥当性の要件を両方とも実装します。"valid if" 要件の違反は同等の制約違反とは異なるエラーを返します。したがって、バージョン 2.0 より前の Oracle Rdb で作成したデータベースにアクセスする場合に整合性違反が発生したかどうかを判断するには、制約違反と重複値エラーのエラー・コード以外に、SQLCODE の値 1002、SQLCODE_NOT_VALID のリテラル値 RDB\$LU_STATUS または RDB\$_NOT_VALID を監視してください。

一意性の違反と "valid if" 要件の違反は、常に INSERT、UPDATE、DELETE などの SQL 文が実行された場合に検出されます。

10.4.2 制約評価の制御

制約違反は、SQL 文 INSERT、UPDATE、DELETE、ALTER TABLE が実行された場合に検出されるか、検出されない場合は COMMIT 文が実行されます。エラーが検出される厳密なタイミングは、データベースが作成または変更されたときとトランザクションが開始されたときに指定されたオプションによって変わります。たとえば、Oracle Rdb は制約を定義したときにテーブルにデータがあればそれを評価します。ALTER TABLE 文を使用して制約を追加すると、Oracle Rdb では制約を評価してテーブル内のデータが制約で定義した基準を満たさない場合に制約違反を返します。

制約の評価を制御するオプションについては、16.7 項 (16-39) を参照してください。この項では、制約の評価によって発生するエラーの処理方法を説明します。

一般的な規則として、データベース・システムによるチェックはユーザーの入力を検証するホスト言語や forms 製品のルーチンに置き換えることができません。プログラムまたは forms 製品のルーチンを使用すると、行の挿入や更新のためのユーザーの入力をチェックし、必要であればユーザーに訂正を要求してからデータベースを変更するトランザクションを開始できます。こうすれば、トランザクションが完了するまでの時間が最小になり、トランザクションでテーブルの行とインデックスに適用するロックの時間が短くなります。

データ値の制約などの要件に関するエラーを効率的に処理するには、まずプログラムで更新するテーブルに関連するテーブル、ドメイン、インデックス、制約の定義をよく理解する必要があります。実施できる範囲で、データベースに関して定義したのと同じ妥当性の要件と、必要な場合は入力するデータに関する追加の要件を指定するプログラムや forms 製品のルーチンを記述してください。このようにプログラムを設計すると、多くの整合性エラー、"valid if" エラー、重複インデックス値エラーを、予期されるエラーでなく予期されないエラーとして処理できます。

10.5 ロックの競合とデッドロックの処理

ロックの競合とデッドロックのエラーは、マルチユーザー・データベース・アクセスで発生します。このエラーは、競合が頻繁に発生するマルチユーザー環境で発生し、特に、データ更新とデータ検索を同時に実行する読取り / 書込みトランザクションを大量に処理する環境で起こります。シングル・ユーザーのデータベースや、読取り / 書込み、共有読取り、読取り専用などのモードでしかデータベースにアクセスしない環境では、ロックの競合やデッドロックはあまり発生しませんが、発生する可能性もあるので、対処が必要です。

10.5.1 ロックの競合エラーの処理

ロックの競合エラーが発生するのは、DECLARE TRANSACTION 文や SET TRANSACTION 文で NOWAIT オプションを指定した状態で、プログラムが要求したデー

タが他のユーザーのトランザクションにロックされた場合です。同じテーブルにアクセスするすべてのトランザクションに、共有された共有モードを適用すると、トランザクションがテーブル行の処理を開始するまでは、一般にロックの競合エラー（可能性がある場合）が発生することはありません。ただし、次のような場合には、ロックの競合エラーが発生する可能性があります。

- NOWAIT オプションを指定した場合は、トランザクションの開始時
DECLARE TRANSACTION 文や SET TRANSACTION 文の RESERVING 句で指定したテーブルが、他のユーザーが開始したトランザクションによって、保護モードまたは排他的共有モードでロックされている場合、トランザクションの開始時にロックの競合が発生します。トランザクション文で、保護モードまたは排他モードで明示的に予約しているテーブルを、他のユーザーが処理している場合にも、トランザクションの開始時にロックの競合が発生します。
- 選択したオプションに関係なく、データベースの接続時
対話型 SQL では、Oracle Rdb データベースのすべてのシステム・テーブルを他のユーザーがロックしている場合、ATTACH 文を入力すると、ロックの競合エラーが発生する可能性があります。（データ定義文を実行しているユーザーが、システム・テーブルのロックの原因になることが多くあります）。Oracle Rdb は、データベースの接続時に、システム・テーブルを使用して内部トランザクションを開始しなければならないので、システム・テーブルがロックされていると、ロックの競合エラーを返します。これと同じ理由で、SET TRANSACTION、OPEN、SELECT、UPDATE、DELETE、INSERT、CREATE、ALTER、DROP 文を使ったプログラムでもロックの競合エラーが発生する可能性があります。この文では、データベース接続が発生します。
- WAIT オプションを指定した状態での、読取り専用で宣言したトランザクションの実行中
Oracle Rdb は、排他的書き込みモードは、読取り専用で宣言したトランザクションとは互換性がないものと見なします。トランザクションで排他的書き込みモードを指定すると、スナップショット・ファイルにはデータを書き込みません。ただし、読取り専用トランザクションが開始する前に、データベースにコミットされたすべてのデータを、読取り専用トランザクションでも使用可能な状態にしておく必要があります。排他的書き込みモードのトランザクションがデータベースヘデータを書き込んだかどうかは、スナップショット・ファイルからは判断できないため、このような読取り専用トランザクションの要件は満たされません。この場合、Oracle Rdb は、WAIT オプションを無視します。
たとえば、ユーザーが排他的書き込みモードでトランザクションを開始し、プログラムはトランザクションを WAIT オプションで読取り専用として宣言した場合、プログラムは、最初のトランザクションが終了するまで待機します。そして、SQL の最初の実行可能文でエラーが発生すると、Oracle Rdb は、プログラムの WAIT オプションを拒否し、ロックの競合エラーをプログラムに発行します。

したがって、すべてのトランザクションに WAIT オプションを指定している場合でも、トランザクション処理以外でロックの競合が発生することを考慮してください。

ロックの競合を検出するには、SQLCODE が -1003 またはリテラル SQLCODE_LOCK_CONFLICT になっていないか、または SQLSTATE が R1002 になっていないかを監視してください。

プログラムは、トランザクションが開始した後にロックの競合エラーを検出すると、トランザクションをロールバックし、次のいずれかの処理を行います。

- トランザクションをすぐに再開する
- 設定された時間の経過後にトランザクションを再開する
- WAIT オプションでトランザクションを再開する
- ユーザーに、後でプログラムを実行してくださいというメッセージを表示し、実行を停止する

トランザクションを再開する場合は、DECLARE TRANSACTION 文ではなく、SET TRANSACTION 文を使用して WAIT モードで開始してください。SET TRANSACTION 文では、待機間隔を指定できます。

ロックの競合エラーの後にトランザクションをロールバックしないと、それに続く文で、予期しない不明なエラーが発生する可能性があります。

サンプル・ディレクトリのサンプル・プログラム sql_terminate は、ロックの競合エラーを検出および処理する方法を示しています。

10.5.2 デッドロック・エラーの処理

ユーザーがお互いに必要なデータをロックしあっていると、デッドロック・エラーが発生します。この場合、両方のユーザーが、処理を続けられなくなります。データベース・システムは、一方のユーザーにデッドロック・エラーを返します。デッドロックが発生したら、もう一人のユーザーが処理を続行できるように、トランザクションをロールバックしてロックを解放する必要があります。

デッドロック・エラーは、テーブルよりもデータベース・リソースでよく発生します。たとえば、同じノードのインデックスに対して、読取りと更新を同時に実行すると、デッドロック・エラーが発生することがあります。デッドロック・エラーは、頻繁に発生することではなく、予測不可能ですが、共有された共有モードでテーブルに同時アクセスするユーザー数が増えるほど、発生率は高くなります。

ロックの競合を検出するには、SQLCODE が -913 またはリテラル SQLCODE_DEADLOCK になっていないか、または SQLSTATE が R1001 になっていないかを監視してください。

注意: SQLV40 表現形式を使用する環境では、データベースへの接続時にプログラムでロックの競合エラーが発生する場合や、トランザクションの開始時にロックの競合またはデッドロックが発生する場合は、トランザクションを開始できないので、COMMIT 文や ROLLBACK 文は失敗します。適切な対処方法は、10.6 項 (10-35) を参照してください。

Oracle Rdb は、デッドロック・エラーを検出した後でもデータベースに更新を発行できますが、デッドロックの原因となる文は、データベースに適用できません。

samples ディレクトリのサンプル・プログラム sql_terminate は、デッドロック・エラーを検出および処理する方法を示しています。

10.6 データベース接続とトランザクション開始の失敗によるエラーの処理

致命的エラーを処理するためのエラー処理セクションの記述が必要な場合があります。すべてのケースでロールバック操作が適切とは限りません。たとえば、プログラムで DECLARE TRANSACTION 文を使用してトランザクション・コンテキストを指定すると、データベース接続とトランザクションの開始の失敗は SELECT...INTO、OPEN、UPDATE、DELETE、INSERT などの SQL 文が最初に実行されたときに発生します。上記の文が COMMIT を含むループ内で指定された場合は、ループの繰返しの間に新しいトランザクションの開始が失敗する恐れがあります。

注意: FETCH 文はループ内に記述はできますが、FETCH 文によってトランザクションを開始することはできません。カーソルがクローズしている場合は、そのカーソルによる行のフェッチはできません。

SQLV40 言語を使用している場合に通常は開始できるトランザクションを開始できないために SQL 文が失敗したときは、ROLLBACK 文を実行するプログラムのエラー処理セクションに分岐しても ROLLBACK 文は失敗します。アクティブなトランザクションがないと、COMMIT 文と ROLLBACK 文は失敗します。この状況を処理しないと、ROLLBACK 文の失敗によってエラー処理セクションが早期に終了したり、プログラムの後続の動作が予測不可能になる恐れがあります。

アクティブなトランザクションがあるかどうかをテストする 1 つの方法は、次のように GET DIAGNOSTICS 文を伴う複数文の手続きを使用することです。

```
BEGIN
  DECLARE :txn INTEGER;
  GET DIAGNOSTICS :txn = TRANSACTION_ACTIVE;
```

```
IF :txn = 1
    THEN ROLLBACK;
END IF;

END;
```

SQLV40 以外の言語を使用する場合は、アクティブなトランザクションがなくても COMMIT 文と ROLLBACK 文は失敗しません。

この問題を処理するもう 1 つの方法は、すべてのトランザクションを SET TRANSACTION 文で明示的に開始することです。それから、SET TRANSACTION 文に固有のエラー処理操作を記述してください。この処理では ROLLBACK 文を省略できます。安全のために、データベース接続とトランザクション開始のいずれかまたは両方が失敗することを想定しているからです。

10.7 エラーと制約を処理するプログラムの移植性の改善

この項は、複数の SQL 製品で機能する必要があるアプリケーションのみに適用されます。この項に示すガイドラインは、『Oracle Rdb7 SQL Reference Manual』またはホスト言語のドキュメントに示す SQL またはホスト言語への Oracle Rdb による拡張に関する説明の代替ではなく、補足です。

SQL のすべての実装に標準のあらゆる側面が含まれるわけではないことに留意してください。したがって、標準の実装でも移植不可能な場合があります。次のガイドラインについて考えてみましょう。

- SQL の 1 つの実装から別の実装への変更を最小限に抑えるコードを記述するためには、文の実行を監視し、SQLSTATE または（プリコンパイル・プログラムでは）SQL WHENEVER 文のみを使用してエラー情報を表示します。
- SQLSTATE の値は、すべてのベンダーに一貫して定義されますが、すべてのベンダーが実装しているわけではありません。
- すべての SQLCODE の値が移植できるとは限りません。たとえば、0、100、負の値、正の値は、SQL の 1 つの実装から別の実装まで常に一貫した意味があります。しかし、たとえば -1003 など、正または負の個々の SQLCODE の値の意味は、多くの場合、ベンダーが定義したデータベースの動作によって変わります。
- SQLCA は、ANSI/ISO SQL 標準には含まれません。
- OpenVMS システム・コールなど、オペレーティング・システムに固有のコールホストを言語ソース・ファイルに挿入しないでください。
- 移植可能な SQL ルーチンを使用します。たとえば、移植不可能な SQL\$GET_ERROR_TEXT でなく sql_get_error_text ルーチンを使用します。
- システムに固有のルーチン、構造体、SQLCODE の値からしか得られない柔軟なエラー処理機能と詳細な情報を犠牲にしたくない場合は、プログラムをコンパイルするプロセ

スの前またはそのプロセスの一貫として、ソース・ファイルにコピーまたはインクルードできる開発システムに固有のモジュールの開発を考えてください。プリコンパイル・プログラムでこの方法を使用する場合は、SQLCODE 文か WHENEVER 文を使用して SQL 文の実行を監視し、システムに固有のエラー処理セクションにコピーします。次に例を示します。

```

IF SQLCODE is less than 0
  branch to error-section
  .
  .
  .
error-section
  IF SQLCODE equals value-list      -
    system-specific actions         |
  else IF SQLCODE equals value-list | Statements to be
    system-specific actions         | copied.
  else                             |
    system-specific actions         |
  end of IF block                  -
end-of-error-section

```

- Oracle Rdb への SQL インタフェースでは、動詞実行時とコンパイル時の制約チェックが可能です。ANSI/ISO SQL 標準では各 SQL 文の終了時に制約をチェックする必要があります。ANSI/ISO 標準の SQL との互換性が必要な場合は、SQL モジュール・プロセッサと SQL プリコンパイラのコマンド行修飾子を使用してコミット時に制約を評価するモードに変更します。
コミット時の制約を各文の終了時に評価するには、SQL モジュール言語プロセッサの CONSTRAINTS=immediate または -cm immediate 修飾子、あるいは SQL プリコンパイラの SQLOPTIONS=(CONSTRAINTS=IMMEDIATE) または -s '-cm immediate' 修飾子を使用します。

動的 SQLでは、プログラムは SQL 文を実行時に受け取ったり生成したりできます。SQL 文がコンパイル時に認識されるプリコンパイル SQL や SQL モジュール言語とは異なり、動的 SQL では実行時にユーザーが文を指定できます。

この章では、動的 SQL インタフェースの使用方法について解説します。この章の各項では、次の作業の方法を説明します。

- SQL 文の動的な実行に関連する用語と概念の使用
- パラメータ・マーカのない SELECT 文以外の SQL 文を動的に実行するプログラムの記述
- プログラムにおけるパラメータ・マーカと選択リスト項目の使用
- パラメータ・マーカを伴う SELECT 文以外の SQL 文を動的に実行するプログラムの記述
- パラメータ・マーカのない SELECT 文を動的に実行するプログラムの記述
- 一連の動的 SQL 文による動的に生成された任意の数の SQL 文の同時処理
- サンプル・ディレクトリの動的 SQL プログラムの使用

11.1 動的 SQL の紹介

プリコンパイル SQL と SQL モジュール言語は、両方とも**静的 SQL** の形式の 1 つです。すなわち、プログラムで処理するデータベース、データ構造、SQL 文に関して必要な情報は、プログラムをコンパイルするときに用意されています。どんな SQL 文を処理すればよいかがプログラム実行時までわからない場合は、**動的 SQL** を使用する必要があります。動的 SQL を使用すると、プログラムで処理する SQL 文の書式や種類を予測できない場合にもアプリケーション・プログラムを記述できます。

この章では、動的 SQL 文、および動的に生成され、実行される SQL 文という用語を頻繁に使用します。**動的 SQL 文**は、プログラムのソース・コードに埋め込まれた、**動的に生成、実行される SQL 文**です。動的に生成され、実行される SQL 文は、必ずしもプログラムのソース・コードの一部ではありません。プログラム実行中に指定し、実行することができます。

参考：『Oracle Rdb7 SQL Reference Manual』には、動的に生成され、実行される SQL 文の詳細と、それを処理する動的 SQL 文の一覧があります。

11.1.1 動的に実行できる SQL 文のカテゴリ

動的に実行される SQL 文は、プログラムが実行時に処理する SQL 文の種類と、プログラムがこれらの文を処理する前に文に関して保有する情報の量によって複数のカテゴリに分類されます。これに従って、プログラムに要求される複雑さが増加します。

SQL では、プリコンパイルされ、動的に実行される SQL 文にはホスト言語変数を指定できません。ただし、SQL 文に埋め込まれた場合はホスト言語変数を使用できる場所に、かわりにパラメータ・マーカーを指定できます。**パラメータ・マーカー**は、プリコンパイルされ、動的に実行される SQL 文内のパラメータを表し、SQL 文内では疑問符 (?) で表されます。

動的 SQL で SELECT 文を処理する場合は、選択リスト項目の数に関する情報を検出し、出力を処理する必要があります。

パラメータ・マーカーと選択リスト項目の有無によって、SQL 文で必要な処理の種類が変わります。動的 SQL 文には、次のカテゴリがあります。

- SELECT 文以外でパラメータ・マーカーのないの SQL 文
動的 SQL 文 EXECUTE IMMEDIATE を使用して、これらの文を実行できます。SQL 文を動的に繰り返し実行するには、EXECUTE IMMEDIATE 文でなく PREPARE 文と EXECUTE 文を使用します。11.2 項 (11-5) を参照してください。
- SELECT 文以外でパラメータ・マーカーを伴う SQL 文
パラメータ・マーカーを伴う SQL 文には、EXECUTE IMMEDIATE 文を使用できません。かわりに、SQL 文を PREPARE 文に渡し、さらに DESCRIBE...MARKERS 文に渡します。SQL は、パラメータ・マーカーの数を決定し、その情報を SQLDA 構造に記述し

ます。プログラムはこの情報を使用してパラメータにストレージを割り当て、EXECUTE 文を使用して文を実行します。

プログラムでは、プリコンパイルされた SELECT 文以外の SQL 文で指定するすべてのパラメータ・マーカーに対応して、ホスト言語変数を宣言するか動的メモリーを割り当てる必要があります。

パラメータマーカーは、SELECT 文の他に、INSERT 文、UPDATE 文、DELETE 文で使用できます。

11.3 項 (11-8) では、パラメータ・マーカーと選択リスト項目の処理方法を概説します。パラメータ・マーカーを伴う SELECT 文以外の SQL 文の処理については、11.4 項 (11-13) を参照してください。11.3.1 項 (11-9) では、SQLDA 構造の使用方法を説明します。

- パラメータ・マーカーのない SELECT 文
プログラムで SELECT 文に対する動的カーソルか拡張動的カーソルを宣言し、SELECT 文で作成した結果テーブルの行の値を受け取る必要があります。さらに、SQL 文の選択リスト項目に関する情報を受け取る SQLDA 構造を作成する必要があります。プログラムで DESCRIBE...SELECT LIST 文を使用して選択リスト項目に関する情報を取得し、この情報を使用して SELECT 文内の各選択リスト項目にストレージを割り当てます。SELECT 文に関するカーソルをオープンしてから、FETCH 文を発行してホスト言語変数値に結果テーブルの値を返します。ただし、FETCH 文自体を動的に実行することはできません。
11.6 項 (11-19) では、SELECT 文の処理方法を詳細に説明します。11.6.1 項 (11-19) では、パラメータ・マーカーのない SELECT 文に限定して解説します。
- パラメータ・マーカーを伴う SELECT 文
SELECT 文にパラメータ・マーカーと選択リスト項目を指定した場合は、プログラムでパラメータ・マーカーに対してホスト言語変数をセットアップし、それに値を割り当てる必要があります。したがって、2つの SQLDA 構造、すなわち選択リスト項目に関する情報を受け取る構造とパラメータ・マーカーに関する情報を受け取る構造を割り当てる必要があります。
11.6 項 (11-19) では、SELECT 文の処理方法を詳細に説明します。11.6.2 項 (11-22) では、パラメータ・マーカーを伴う SELECT 文に限定して解説します。
- パラメータ・マーカーまたは選択リスト項目、あるいはその両方について、その数とデータ型がわからない文
パラメータ・マーカーまたは選択リスト項目の数とデータ型に関する情報をプログラムが保持しない場合も少なくありません。これらは動的 SQL で処理する上で最も複雑な文です。
11.7 項 (11-32) では、このような状況を処理するプログラムの記述方法について説明します。

SQL では、パラメータ・マーカーと選択リスト項目の数とデータ型を決定し、その情報をプログラムに伝える必要があります。プログラムはパラメータの数とデータ型を認識すると、それらにメモリーを割り当て、そのメモリーの位置を SQL に伝えます。動的 SQL では、

SQLDA (SQL 記述領域) と呼ばれる構造を使用してパラメータ・マーカーまたは選択リスト項目 (あるいはその両方) に関する情報をプログラムに伝えます。

SQLDA の詳細は、11.3.1 項 (11-9) を参照してください。11.8 項 (11-39) には、この章で解説する多くの処理方法を示すサンプル・プログラム `sql_dynamic` の説明があります。この章に示す多くの例は、C によるサンプル・プログラムの抜粋です。サンプル・ディレクトリには、他の言語によるプログラムもあります。

11.1.2 動的 SQL 文による他の SQL 文の処理

動的に生成され、実行される SQL 文を処理するために、動的 SQL には次の文が用意されています。

- EXECUTE IMMEDIATE: パラメータ・マーカーのない任意の SQL 文 (SELECT 文以外) のプリコンパイルと実行を 1 ステップで行います。
- PREPARE: 動的に実行される SQL 文にエラーがないかを確認し、識別子を割り当てます。この識別子は DESCRIBE 文、EXECUTE 文、動的 DECLARE CURSOR 文、拡張動的 DECLARE CURSOR 文で参照されます。PREPARE 文によって、実行するコードが生成されます。
- DESCRIBE: プリコンパイルされた文について、パラメータ・マーカーまたは選択リスト項目の有無を確認します。ある場合は、その数とデータ型に関する情報を SQL 記述領域 (SQLDA) に格納します。
- EXECUTE: 事前にプリコンパイルされた SELECT 文以外の SQL 文を実行します。
- 動的 DECLARE CURSOR: 事前にプリコンパイルされた SELECT 文のカーソルを宣言します。動的 DECLARE CURSOR 文では、コンパイル時にカーソル名を指定しますが、SELECT 文は明示的には指定しません。かわりに、通常は SELECT 文を配置する場所にプリコンパイルされた SQL 文の名前を指定します。
- 拡張動的 DECLARE CURSOR: カーソルを宣言しますが、カーソル名も SELECT 文も実行時まで指定されません。かわりに、カーソル名と SELECT 文に対してパラメータを指定します。
SQL では拡張動的 DECLARE CURSOR 文を実行文として処理します。これは、SQL が非拡張動的 DECLARE CURSOR 文を処理する方法とは異なります。
- OPEN: プリコンパイルされた SELECT 文に対して宣言されたカーソルをオープンします。
- FETCH: プリコンパイルされた SELECT 文に対して宣言されたカーソルから値を取得します。
- RELEASE: プリコンパイルされた SQL 文で使用するリソースをすべて解放します。この SQL 文は再実行できなくなります。

動的に実行される SQL 文を処理するために使用する上記の文は、それ自体が動的に実行可能なことに注意してください。

参考：『Oracle Rdb7 SQL Reference Manual』の SQL 文に関する章で、前述の各文に関する項を参照してください。
『Oracle Rdb7 SQL Reference Manual』の表に、SQL で動的に実行できる SQL 文を示します。ただし、パラメータ・マーカと選択リスト項目、およびそれらを処理する埋込み SQL 文を使用できるものとできないものがあります。

11.1.3 動的 SQL における SQL 文の処理

SELECT 文またはパラメータ・マーカを伴う文を使用するプログラムでは、次の一般的な手順に従って実行時に SQL 文を処理します。

1. 文をプログラム変数として受け取ります。
2. このプログラム変数を PREPARE 文に渡します。
3. DESCRIBE...MARKERS 文を使用して、文にパラメータ・マーカがあるかを確認し、ある場合はその変数にストレージを割り当てます。
4. 文が SELECT 文かどうかをテストします。SELECT でない場合は、EXECUTE 文で処理します。
5. SELECT 文の場合は次のように処理します。
 - a. DESCRIBE...SELECT ITEMS 文を使用して、一連のパラメータ内の選択リスト項目を検出します。それに対応するストレージを割り当てます。
 - b. カーソルをオープンします。
 - c. 割り当てられたストレージを使用して結果テーブルの各行を取得します。
6. プログラムを終了し、プリコンパイルされた SQL 文を解放します。

パラメータ・マーカを使用しない SELECT 文以外の SQL 文のみを処理する場合は、11.2 項 (11-5) で説明する EXECUTE IMMEDIATE 文を使用できます。

11.2 パラメータ・マーカのない SELECT 文以外の SQL 文の実行

処理しようとする SQL 文が SELECT 文以外であり、パラメータ・マーカがない場合は、プリコンパイルしてから実行する必要はありません。唯一の EXECUTE IMMEDIATE 文でこの両方のステップを実行できます。

パラメータ・マーカーのない SQL 文の文字列を生成する書込み専用またはデータ定義のアプリケーションには、EXECUTE IMMEDIATE 文が便利です。

例 11-1 (11-6) は、このようなケースを示しています。COBOL プログラムは、端末から SQL 文を受け取り、EXECUTE IMMEDIATE 文でこれを動的に処理します。このプログラムでは、動的に実行する唯一の SELECT 文以外の SQL 文のみを処理できます。文が対話方式で生成されるので、パラメータ・マーカーはありません。

例 11-1 EXECUTE IMMEDIATE 文による SELECT 文以外の SQL 文の実行

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXECUTE_IMMEDIATE_EXAMPLE.
*
* Illustrate EXECUTE IMMEDIATE with a dynamic INSERT statement.
*
DATA DIVISION.
WORKING-STORAGE SECTION.
* Variable for DECLARE ALIAS:
01 FILESPEC PIC X(20). (1) (11-7)
* Variable to hold the command to be dynamically executed:
01 COMMAND_STRING PIC X(240). (2) (11-7)
* Buffer for error handling:
01 GETERRVARS.
    02 error-buffer-len PIC S9(9) COMP VALUE 132.
    02 error-msg-len PIC S9(9) COMP.
    02 error-buffer PIC X(132).
* Load definition for SQL Communications Area (SQLCA) for error handling:
EXEC SQL INCLUDE SQLCA END-EXEC.
*****
*
* PROCEDURE DIVISION
*
*****
PROCEDURE DIVISION.
START-UP.
* Assign value to FILESPEC:
DISPLAY "Enter the file spec for the database you want to declare:"
ACCEPT FILESPEC. (3) (11-7)
* Declare the database:
* (You can name any file for COMPILETIME FILENAME because no
* embedded statement in this example requires that SQL attach
* to the database.)
EXEC SQL DECLARE ALIAS (4) (11-7)
    COMPILETIME FILENAME personnel
    RUNTIME FILENAME :FILESPEC
END-EXEC
DISPLAY "Enter an SQL statement (not SELECT):"
```

```

ACCEPT COMMAND_STRING. (5) (11-7)
* Use EXECUTE IMMEDIATE to execute the statement in COMMAND_STRING:
EXEC SQL EXECUTE IMMEDIATE :COMMAND_STRING (6) (11-7)
END-EXEC
PERFORM CHECK
EXEC SQL EXECUTE IMMEDIATE 'ROLLBACK' END-EXEC.
PERFORM CHECK.
DISPLAY "Rolled back changes. All done.".
CLEAR-IT-EXIT.
STOP RUN.
CHECK.
IF SQLCODE NOT = 100 AND SQLCODE NOT = 0 (7) (11-7)
    DISPLAY "Error: SQLCODE = ", SQLCODE WITH CONVERSION
    CALL "sql_get_error_text" USING BY REFERENCE error-buffer,
                                BY VALUE error-buffer-len,
                                BY REFERENCE error-msg-len
    DISPLAY error-buffer
END-IF.

```

次の番号は、例 11-1 (11-6) の番号に対応しています。

- (1) 宣言するデータベースのファイル名を表す変数 FILESPEC を宣言します。
- (2) 動的に実行する SQL コマンドを表す変数 COMMAND_STRING を宣言します。
- (3) 端末からの入力を FILESPEC 変数に受け取ります。
- (4) エイリアスを宣言します。DECLARE ALIAS 文はソース・プログラムに埋め込まれているので、このプログラムでは動的に実行可能な SQL 文とは見なしません。静的 SQL で文に変数を渡すのと同じ方法で、DECLARE ALIAS 文に変数を渡します。
- (5) 端末からの入力を COMMAND_STRING 変数に受け取ります。
- (6) EXECUTE IMMEDIATE 文を使用して、端末で入力された文を実行します。
- (7) エラーをチェックします。SQL がエラーを返す場合は、プログラムによって SQLCODE の値とエラーの説明が表示されます。

プログラムで特定の SQL 文を動的に繰り返し実行する場合は、PREPARE 文を発行してから EXECUTE 文を使用して実行を繰り返す方が効率的です。たとえば、例 11-1 (11-6) で EXECUTE IMMEDIATE :COMMAND_STRING 文のかわりに次の行を挿入します。

```

* Use these commands to execute one statement several times.
EXEC SQL PREPARE STMT1 FROM :COMMAND_STRING
END-EXEC
PERFORM CHECK
DISPLAY "Statement prepared."
PERFORM UNTIL RESPONSE = "NO"
    EXEC SQL EXECUTE STMT1

```

```
END-EXEC
PERFORM CHECK
DISPLAY "Do you want to execute the statement again:"
ACCEPT RESPONSE
END-PERFORM.
```

動的 SQL と EXECUTE IMMEDIATE 文を使用すると、1 つのプログラム内で同じスキーマ・オブジェクトを削除してから再度作成する場合の一般的な問題も回避できます。

たとえば、プリコンパイル SQL プログラムに DROP INDEX 文と CREATE INDEX 文の両方が記述された場合は、プリコンパイラが次のようなエラーを返します。

```
%SQL-F-IND_EXISTS, Index DEG_EMP_ID already exists in this database or schema
```

インデックスはコンパイル時には削除されないで、コンパイル時のデータベースにはまだインデックスが存在しており、エラーが生成されます。この制限を回避するには、EXECUTE IMMEDIATE 文を使用してインデックスを削除してから再作成してください。

11.3 パラメータ・マーカーと選択リスト項目の処理

この項では、パラメータ・マーカーと選択リスト項目の処理方法を説明します。パラメータ・マーカーのみを伴う文の処理については、11.4 項 (11-13) を参照してください。

SELECT 文の処理の詳細は、11.6 項 (11-19) を参照してください。

SELECT 文と同様に、INSERT 文、UPDATE 文、DELETE 文でもパラメータ・マーカーを使用できます。SQL は、パラメータ・マーカーをホスト言語変数の値で置き換えます。プリコンパイルされた SQL 文が後続の EXECUTE 文によって動的に実行される場合は、動的メモリの値で置き換えます。

SQL 文にパラメータ・マーカーが含まれる場合は、プログラムでホスト言語変数を定義するか、プリコンパイルされた SELECT 文以外の SQL 文で指定したすべてのパラメータ・マーカーに対応する動的メモリーを割り当てる必要があります。

パラメータ・マーカーまたは選択リスト項目を伴う文には、EXECUTE IMMEDIATE 文を使用できません。かわりに、動的に実行する文を PREPARE 文でプリコンパイルしてください。さらに、プログラムでホスト言語変数を宣言するか、すべてのパラメータ・マーカーと選択リスト項目に動的メモリーを割り当てる必要があります。この文を実行するには、プログラムで埋込みの EXECUTE 文を発行するか、EXECUTE 文を含む SQL モジュール言語プロシージャをコールします。

プログラムでは、SQLDA と呼ばれるデータ構造を宣言し、DESCRIBE 文を使用してパラメータ・マーカーまたは選択リスト項目に関する情報を取得します。

SQL プロセッサが DESCRIBE 文を処理すると、選択リスト項目 (DESCRIBE...SELECT LIST 文の場合) またはプリコンパイルされた文のパラメータ・マーカー (DESCRIBE...MARKERS 文の場合) の数とデータ型を SQLDA に書き込みます。ホスト言語プログラムは SQLDA からこの情報を読み取り、個々のパラメータにストレージ (ホスト言

語変数または動的メモリー）を割り当て、そのストレージのアドレスを SQLDA に書き込みます。さらに、パラメータ・マーカーに代入する値を指定し、パラメータ・マーカーに割り当てたストレージにその値を書き込みます。

SQL 文でパラメータ・マーカーを使用する場合は、パラメータのデータ型に関する仮定はありません。SQL は、パラメータを特定の操作に適したデータ型に変換する場合があります。たとえば、パラメータ・マーカーを LIKE 述語の値式の 1 つとして使用すると SQL は VARCHAR データ型を返し、他の値式では CHAR データ型を返す場合があります。

参考: MARKERS 句と SELECT LIST 句の詳細は、『Oracle Rdb7 SQL Reference Manual』の DESCRIBE 文に関する項を参照してください。

11.3.1 SQLDA 構造と SQLDA2 構造の使用方法

SQL では、パラメータ・マーカーと選択リスト項目がある場合にその数とデータ型に関する情報をプログラムに伝える必要があります。プログラムは、パラメータの数とデータ型を認識するとそれらにメモリーを割り当て、SQL にそのメモリーの位置を伝えます。プログラムはパラメータ・マーカーと選択リスト項目の数およびデータ型を SQL 記述子領域に書き込む必要があります。SQL 記述子領域 (SQLDA) は、SQL がこの情報を保持するために提供するデータ構造です。

SQLDA は、プリコンパイルされた SQL 文のパラメータ・マーカーと選択リスト項目に関する情報を保持します。SQL は、DESCRIBE...SELECT LIST 文か DESCRIBE...MARKERS 文を処理するときにパラメータ・マーカーと選択リスト項目の数とデータ型に関する情報を SQLDA に書き込みます。SQLDA を使用してパラメータ・マーカーと選択リスト項目の有無を調べる方法と、SQLERRD フィールドを使用して文が SELECT 文かどうかを調べる方法については、11.6.3 項 (11-29) を参照してください。

プリコンパイラでサポートする言語のうち、Ada、C、PL/I では SQLDA を使用できます。ポインタ変数をサポートする他の言語は SQLDA を使用できますが、SQL 文をソース・コードに埋め込むかわりに SQL 文を含む SQL モジュール・プロシージャをコールする必要があります。

ポインタ変数をサポートしないホスト言語は、動的 SQL の柔軟性を十分に利用できません。すなわち、動的に生成された SQL 文にパラメータ・マーカーと選択リスト項目があるかどうかを判断し、ある場合はその数とデータ型を他の方法で確認する動的な SQL アプリケーションに限定されます。

SQL では、SQLDA の拡張版、SQL 記述子領域 2 (SQLDA2) も提供します。SQLDA2 では、追加のフィールドとフィールド・サイズをサポートします。記述子領域を必要とする任意の動的 SQL プログラムで SQLDA か SQLDA2 を使用できますが、日付時刻データ型を使用する場合は SQLDA2 を使用する必要があります。11.3.2 項 (11-10) に、C プログラムで

この構造を宣言する方法を示します。sql_dynamic サンプル・プログラムの C バージョンは、動的 SQL プログラムで SQLDA2 構造を使用する方法を示しています。

注意: SQL で使用するパラメータ・マーカーと選択リスト項目のデータ型を、ご使用の言語でもサポートするとは限らないのでご注意ください。DESCRIBE 文は、パラメータ・マーカーと選択リスト項目のデータ型に関する情報を SQLDA の SQLTYPE フィールドと SQLLEN フィールドに返します。必要に応じて、SQLDA の SQLTYPE フィールドと SQLLEN フィールドの値を SQL とホスト言語の両方がサポートするデータ型と長さにプログラムで変更してください。

11.3.2 SQLDA 構造と SQLDA2 構造の宣言

SQLDA と SQLDA2 を宣言するには、INCLUDE 文を使用するか、ポインタ変数をサポートするホスト言語で記述したプログラム内で SQLDA か SQLDA2 を明示的に宣言します。さらに、Ada プリコンパイル・プログラムでは SQLDA_ACCESS 型を指定でき、C プリコンパイル・プログラムと C ホスト言語プログラムでは sql_sqlda.h ヘッダー・ファイルを使用できます。

次の例は、C プログラムにファイルをインクルードする方法を示しています。

```
#include <sql_sqlda.h>
```

sql_sqlda.h ヘッダー・ファイルには、SQL_T_SQLDA データ型と SQL_T_SQLDA2 データ型を定義する SQLDA 構造と SQLDA2 構造の typedef 文が記述されています。さらに、SQLDA 構造と SQLDA2 構造のスーパーセット定義として、SQL_T_SQLDA_FULL データ型と SQL_T_SQLDA2_FULL データ型を定義しています。SQL_T_SQLDA_FULL データ型は、CALL 文を記述するときに SQL が使用する追加のフィールドを伴う追加の共用体が含まれる以外は SQL_T_SQLDA データ型と同じレイアウトです。同様に、SQL_T_SQLDA2_FULL データ型は、CALL 文を記述するときに SQL が使用する追加のフィールドを伴う追加の共用体が含まれる以外は SQL_T_SQLDA2 データ型と同じレイアウトです。これらのフィールドは、次のように SQL_T_SQLDA_FULL データ型と SQL_T_SQLDA2_FULL データ型の両方で同じ名前を使用します。

- SQLPRCSN--- 数値フィールドの長さ
- SQLSCALE--- 数値フィールドの位取り
- SQLARGPOS--- 引数の位置
- SQLFLAGS--- 16 個のフラグ・ビット。次のフィールドで使用します。
 - SQLFLAGS_VALID--- フラグと位置が有効
 - SQLPARAM_IN--- パラメータが入力

■ SQLPARAM_OUT--- パラメータが出力

SQL_T_SQLDA データ型と SQL_T_SQLDA2 データ型のフィールド参照式は、INCLUDE SQLDA 文または INCLUDE SQLDA2 文を使用して生成されるものと同じです。

SQL_T_SQLDA_FULL データ型または SQL_T_SQLDA2_FULL データ型を使用する場合は、sql_sqlda.h ヘッダー・ファイルを参照し、適切なフィールド参照式を決定してください。

INCLUDE SQLDA 文を使用すると、同じプログラム内で INCLUDE SQLDA2 文は使用できません。2つの文は、グローバルに参照できるオブジェクト・データを同じ名前で定義するからです。ただし、sql_sqlda.h ヘッダー・ファイルはグローバルに参照できるデータ定義または参照を定義しないので、sql_sqlda.h ヘッダー・ファイルはいずれの INCLUDE 文と同じプログラムでも使用できます。ヘッダー・ファイルを使用するプログラムでは、SQLDA または SQLDA2 あるいはその両方の typedef を使用する定義または参照を作成できます。

Compaq Tru64 UNIX

Compaq Tru64 UNIX では、アプリケーション内で INCLUDE SQLDA 文か INCLUDE SQLDA2 文を 1 度のみ使用できます。

Compaq Tru64 UNIX では、SQLDA 構造と SQLDA2 構造のフィールドは本来 4 ワードの境界に整列します。ホスト言語で SQLDA 構造か SQLDA2 構造を定義する場合は、SQL 内部書式と互換性を持つように整列する必要があります。たとえば、COBOL の -align スイッチを使用できます。◆

OpenVMS VAX
OpenVMS Alpha

OpenVMS では、アプリケーションでグローバルに参照できるデータ・オブジェクトを複数イメージ間で共有する場合に、アプリケーション内で INCLUDE SQLDA 文か INCLUDE SQLDA2 文を 1 度のみ使用できます。◆

OpenVMS Alpha

OpenVMS Alpha では、SQL モジュール・プロセッサを使用してモジュール・ヘッダーの言語句で C を指定する場合、または C 用の SQL プリコンパイラを使用する場合、SQL はデフォルトで構造内のフィールドを整列します。ただし、SQLDA と SQLDA2 のメンバーの整列は許可しないでください。◆

SQLDA 構造と SQLDA2 構造を明示的に定義する場合は、構造の定義を C プリプロセッサ・ディレクティブ #pragma nomember_alignment で囲み、構造の整列を回避します。SQL の INCLUDE 文を使用すると、プリプロセッサ・ディレクティブを使用する必要がなくなります。

11.8 項 (11-39) に記載する sql_dynamic プログラムには、2つの別々の SQLDA2 構造 SQLDA_IN と SQLDA_OUT の宣言があります。例 11-2 (11-12) にこれらの宣言を示します。SQLDA_IN 構造は、DESCRIBE...MARKERS 文のターゲットであり、ユーザーが入力した SQL 文の文字列におけるパラメータ・マーカーに関する情報を受け取ります。SQLDA_OUT 構造は、PREPARE 文と DESCRIBE...SELECT LIST 文のターゲットであり、SQL 文の文字列内に指定された選択リスト項目に関する情報を受け取ります。

プログラムでは両方の構造をポインタ構造として明示的に宣言しているので、2つの構造には実行時に動的にメモリーが割り当てられます。例 11-2 (11-12) に、sql_dynamic.c による

SQLDA2 の宣言を示します。『Oracle Rdb7 SQL Reference Manual』には、他の言語で SQLDA2 を宣言した例があります。

例 11-2 SQLDA2 構造の宣言

```
/* Declare the SQLDA2. */
typedef
struct
{
    char  sqldaid[8];
    int   sqlabc;
    short sqln;
    short sqld;
    struct sqlvar_struct
    {
        short sqltype;
        long  sqllen;
        int   sqloctet_len;
        char  *sqldata;
        int   *sqlind;
        int   sqlchrono_scale;
        int   sqlchrono_precision;
        short sqlname_len;
        char  sqlname[IDLEN];
        char  sqlchar_set_name[IDLEN];
        char  sqlchar_set_schema[IDLEN];
        char  sqlchar_set_catalog[IDLEN];
    } sqlvar[MAXPARAMS];
} sqlda_rec, *sqlda;

.
.
.
int  sql_dynamic (psql_stmt, input_sqlda, output_sqlda, stmt_id, is_select)
char *psql_stmt;
sqlda *input_sqlda;
sqlda *output_sqlda;
long  *stmt_id;
int   *is_select;
{
    sqlda sqlda_in, sqlda_out;    /* declare the SQLDA structures */
    .
    .
    .
}
```

参考：『Oracle Rdb7 SQL Reference Manual』の付録には、SQLDA の説明があります。

11.4 パラメータ・マーカを伴う SELECT 文以外の SQL 文の実行

SQL 文にパラメータ・マーカが指定された場合、プログラムではプリコンパイルされた SQL 文のすべてのパラメータ・マーカに対応するホスト言語変数の宣言または動的メモリの割当てが必要になります。プログラムでは、SQLDA の情報を使用してパラメータ・マーカにストレージを割り当てます。

例 11-3 (11-13) に、パラメータ・マーカを伴う SELECT 文以外の SQL 文を動的に実行するプリコンパイルされた単純な C プログラムを示します。

例 11.3 パラメータ・マーカを伴う SELECT 文以外の SQL 文の実行

```
/* This program dynamically executes non-SELECT statements that contain
 * parameter markers.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql_rdb_headers.h>
/* Maximum number of parameters or select list items is 50.*/
#define MAXPARAMS 50
#if defined (__osf__) && defined (__DECC)
#pragma member_alignment save
#pragma member_alignment
#elif defined (__DECC)
#pragma member_alignment save
#pragma nomember_alignment
#endif
#pragma nostandard
struct SQLDA_STRUCT { (1)
    char SQLDAID[8];
    int SQLDABC;
    short SQLN;
    short SQLD;
    struct {
        short SQLTYPE;
        short SQLLEN;
        char *SQLDATA;
        int *SQLIND;
        short SQLNAME_LEN;
        char SQLNAME[30];
    } SQLVAR[MAXPARAMS];
} *SQLDA;
#if defined (__osf__) && defined (__DECC)
#pragma member_alignment restore
#elif defined (__DECC)
```

```

#pragma member_alignment restore
#endif
#pragma standard
main()
{
    char          command_string[256];    /* SQL statement text */
    long          SQLCODE;
    short         param;
    int           loop_cntr;
    char          column_name[31];
    /* Declare pointer variables for each possible data type. The program
    uses the pointers to dynamically allocate memory to hold values for
    parameter markers.
    */
    char          *charbuf;
    long          *longbuf;
    short         *smallintbuf;
    SQL_DATE_VMS my_date_vms;
    /* Allocate SQLDA structures. */ (2)
    SQLDA = malloc(534);
    SQLDA->SQLN = MAXPARAMS;
    /* Declare the alias. */
    EXEC SQL DECLARE ALIAS FILENAME personnel;
    /* Start an SQL transaction. */
    EXEC SQL SET TRANSACTION READ WRITE;
    if (SQLCODE != 0)
        goto err;
    printf("Enter any dynamically-executable SQL statement, %n"); (3)
    printf("except SELECT. %n");
    gets(command_string);
    if (SQLCODE != 0)
        goto err;
    /* Prepare the statement from input entered at run time. */
    EXEC SQL PREPARE STMT_NAME FROM :command_string; (4)
    if (SQLCODE != 0)
        goto err;
    /* Write information about the parameter markers into the SQLDA. */
    EXEC SQL DESCRIBE STMT_NAME MARKERS INTO SQLDA; (5)
    /* For each parameter marker, check the value of SQLDA.SQLVAR.SQLTYPE
    to determine the data type of the parameter marker, and branch to
    the appropriate code to prompt and store the value for that data type.
    */
    param = 0;
    for (param = 0; param < SQLDA->SQLD; param++) (6)
    {
        /* Handle null-terminated column name and prompt with the column name. */
        strncpy (&column_name[0], &SQLDA->SQLVAR[param].SQLNAME[0],

```

```

        SQLDA->SQLVAR[param].SQLNAME_LEN);
column_name[SQLDA->SQLVAR[param].SQLNAME_LEN] = '\0';
printf ("Enter value for parameter '%s'", column_name);
switch (SQLDA->SQLVAR[param].SQLTYPE) (7)
{
case 453 :                                /* character variable */
case 449 :                                /* variable char variable */
    printf("\n(Maximum length is %d)",SQLDA->SQLVAR[param].SQLLEN);
    printf("\ndsql> ");
    /* Allocate storage and get the address for the character
       string. */
    charbuf = (char *) malloc (SQLDA->SQLVAR[param].SQLLEN);
    /* Get the string. */
    gets (charbuf);
    /* Assign the address to SQLDA. */
    SQLDA->SQLVAR[param].SQLDATA = charbuf;
    /* Change the character data type to null-terminated. */
    SQLDA->SQLVAR[param].SQLTYPE = 506; (8)
    break; /* Exit the switch block. */
case 497 :                                /* integer variable */
    printf ("\ndsql> ");
    /* Allocate storage for the character string version of
       the integer to be entered.
       */
    charbuf = (char *) malloc (14);
    /* Get the string. */
    gets(charbuf);
    /* Allocate storage for the integer and convert the string. */
    longbuf = (long *) malloc (SQLDA->SQLVAR[param].SQLLEN);
    *longbuf = atol(charbuf);
    /* Assign the address to SQLDA. */
    SQLDA->SQLVAR[param].SQLDATA = (char *) longbuf;
    break; /* Exit the switch block. */
case 501 :                                /* short int variable */
    printf ("\ndsql> ");
    /* Allocate storage for the character string version of
       the short integer to be entered. */
    charbuf = (char *) malloc (5);
    /* Get the string. */
    gets (charbuf);
    /* Allocate storage for the short int and convert the string. */
    smallintbuf = (short *) malloc(SQLDA->SQLVAR[param].SQLLEN);
    *smallintbuf = atoi(charbuf);
    /* Assign the address to SQLDA. */
    SQLDA->SQLVAR[param].SQLDATA = (char *) smallintbuf;
    break; /* Exit the switch block. */
case 503 :                                /* Date VMS variable */

```

```

printf ("%nDate/time format is DD-MMM-YYYY HH:MM:SS.HH ");
printf ("%ndsql> ");
/* Allocate storage for the character string version of
   the date and time to be entered.    */
charbuf = (char *) malloc (23);
/* Get the string. */
gets (charbuf);
/* Use the CAST statement to convert the input character string
   to the DATE VMS data type. */
EXEC SQL
    BEGIN SELECT CAST(substring(:charbuf FROM 8 FOR 4) ||
-- Convert the month to a number.
        (CASE SUBSTRING(:charbuf FROM 4 FOR 3)
            WHEN 'JAN' THEN '01'
            WHEN 'FEB' THEN '02'
            WHEN 'MAR' THEN '03'
            WHEN 'APR' THEN '04'
            WHEN 'MAY' THEN '05'
            WHEN 'JUN' THEN '06'
            WHEN 'JUL' THEN '07'
            WHEN 'AUG' THEN '08'
            WHEN 'SEP' THEN '09'
            WHEN 'OCT' THEN '10'
            WHEN 'NOV' THEN '11'
            WHEN 'DEC' THEN '12'
        END) ||
-- Parse the day, hour, minutes, seconds.
        SUBSTRING(:charbuf FROM 1 FOR 2) ||
        SUBSTRING(:charbuf FROM 13 FOR 2) ||
        SUBSTRING(:charbuf FROM 16 FOR 2) ||
        SUBSTRING(:charbuf FROM 19 FOR 2) ||
        SUBSTRING(:charbuf FROM 22 for 2)
    AS DATE VMS) into :my_date_vms
    FROM rdb$database LIMIT TO 1 ROW;
END;
if (SQLCODE != 0)
    goto err;
/* Assign the address to SQLDA. */
SQLDA->SQLVAR[param].SQLDATA = (char *) &my_date_vms;
break; /* Exit the switch block. */
default :
    printf ("%nError: no data type match on %d",
            SQLDA->SQLVAR[param].SQLTYPE);
    break;
} /* End switch. */
} /* End the for loop. */
/* Execute the statement and replace the parameter markers with the

```

```

values in the SQLDA. */
EXEC SQL EXECUTE STMT_NAME USING DESCRIPTOR SQLDA;
    if (SQLCODE != 0)
        goto err;
EXEC SQL COMMIT;
return;
err:    printf("%n unexpected error %d", SQLCODE);
        sql_signal();
}

```

次の番号は、例 11-3 (11-13) の番号に対応しています。

- (1) SQLDA 構造を宣言します。
- (2) SQLDA 構造を割り当てます。
- (3) 実行する SQL コマンドの入力を要求します。
- (4) ユーザーが入力した文をプリコンパイルします。エラーが返される場合は、エラー処理ルーチンに分岐します。
- (5) DESCRIBE...MARKERS 文を使用してパラメータ・マーカに関する情報を SQLDA 構造に記述します。
- (6) SQLD フィールドに格納された値と同じ回数、つまり一連の文のパラメータ・マーカごとに 1 回だけ実行するループをセット・アップします。
- (7) case 文で、SQLDA の SQLTYPE フィールドの値をテストし、パラメータ・マーカのデータ型を判断します。SQLTYPE フィールドの情報に基づいて、malloc() 関数で適切な長さのストレージを割り当てます。ユーザーに列の値の入力を要求し、そのアドレスを SQLDA に割り当てます。
- (8) CAST 関数によって、文字列データのデータ型を文字または可変文字列から NULL 終了文字列に変更します。

11.5 SELECT 文か否かのテスト

例 11-3 (11-13) では、ユーザーが SELECT 文を入力しないと仮定しています。多くのプログラムでは、このように仮定することはできません。

10 章 (10-1) に示す SQLCA 構造内の SQLERRD 配列の第 2 要素は、プリコンパイルされた文が SELECT 文かどうかを示しています。SQLERRD 配列の 2 番目の要素が 0 の場合は、SQL 文の文字列は SELECT 文ではないので直接実行することができます。この値が 1 の場合は、SQL 文の文字列が SELECT 文です。11.6 項 (11-19) では、SELECT 文を処理する方法について説明します。

例 11-4 (11-18) の `sql_dynamic` プログラムの抜粋は、C プログラムで `SQLERRD` 配列の第 2 要素をテストする方法を示しています。文が `SELECT` 文でない場合、プログラムは SQL モジュール言語プロシージャ内で `EXECUTE` 文をコールします。

例 11-4 SQLERRD のテストによる SELECT 文以外の SQL 文の識別

```
.
.
.
/* Save the value of the SQLCA.SQLERRD[1] field so that the program
   can determine if the statement is a SELECT statement or not.
   If the value is 1, the statement is a SELECT statement.*/
*is_select = SQLCA.SQLERRD[1];
.
.
.
   if (*is_select)
.
.
.
} /* end if SQLCA.SQLERRD[1] == 1) */
else
{
.
.
.
/*
** If the SQLCA.SQLERRD[1] field is not 1, then the prepared statement is not a
** SELECT statement and only needs to be executed. Call an SQL module language
** procedure to execute the statement, using information about parameter
** markers stored in sqlda_in by the local routine get_in_params:
*/
/*      if (SQLCA.SQLERRD[1] != 1)*/
{
execute_stmt (&SQLCA, stmt_id, sqlda_in);
if (SQLCA.SQLCODE != sql_success)
{
printf("\n\nError %d returned from execute_stmt",SQLCA.SQLCODE);
display_error_message();
return (-1);
}
} /* end if SQLCA.SQLERRD[1] != 1 */
} /* end if SQLCA.SQLERRD[1] == 1 */
```


11.6 SELECT 文の処理

SELECT 文を実行するには、パラメータ・マーカがあるかどうかにかかわらず、プログラムで動的カーソルか拡張動的カーソルを宣言する必要があります。カーソルは、SELECT 文で作成した結果テーブルの行の値を受け取ります。

動的カーソルの場合は、コンパイル時にカーソル名は指定されますが、SELECT 文は明示的には指定されません。かわりに、SELECT 文に対してプリコンパイルされた SQL 文の名前を指定します。動的 DECLARE CURSOR 文では実行時に SELECT 文を指定できるので、動的 DECLARE CURSOR 文を使用すると動的に生成された SQL 文をいくつでも処理できます。

拡張動的カーソルの場合は、コンパイル時にカーソル名をも SELECT 文も指定されません。かわりに、カーソル名と SELECT 文に対してパラメータを指定します。動的 DECLARE CURSOR 文では実行時にこの情報を指定できるので、動的 DECLARE CURSOR 文を使用すると動的に生成された SQL 文を同時にいくつでも処理できます。

プログラムでは、動的カーソルか拡張動的カーソルをセットアップして使用する以外に、SELECT 文をプリコンパイルし (PREPARE 文を使用)、DESCRIBE...SELECT LIST 文を使用して選択リスト項目の数とデータ型に関する情報を SQLDA に書き込む必要があります。

プログラムでは、SQLDA の情報を使用して選択リスト項目にストレージを割り当てます。SQLDA を宣言して使用方法の詳細は、11.3.1 項 (11-9) と 11.6.3 項 (11-29) を参照してください。

また、パラメータが NULL 値を返す可能性がある場合は、プログラムでインジケータ・パラメータを宣言して使用する必要があります (8 章 (8-1) を参照)。

SELECT 文に関するカーソルをオープンしてから、FETCH 文を使用してホスト言語変数に SQLDA の各行の値を返します (ただし、FETCH 文自体を動的に実行することはできません)。

11.6.1 パラメータ・マーカのない SELECT 文の実行 : 動的カーソルと拡張動的カーソルの宣言

例 11-5 (11-19) に、パラメータ・マーカのない SELECT 文を動的に実行する SQL モジュール `c_mod_dyn_curs` を示します。このモジュールでは、動的カーソルを宣言して行の値を保持する結果テーブルを指定します。

この SQL モジュールをコールするホスト言語プログラムを例 11-6 (11-20) に示します。

例 11-5 SQL モジュールにおけるパラメータ・マーカのない SELECT 文の実行

```
-- This program uses dynamic cursors to fetch a row from a table.
--
MODULE          C_MOD_DYN_CURS
LANGUAGE        GENERAL
```

```
AUTHORIZATION RDB$DBHANDLE
PARAMETER COLONS
DECLARE ALIAS FILENAME personnel
-- Declare the dynamic cursor. Use a statement name to identify a
-- prepared SELECT statement.
DECLARE CURSOR1 CURSOR FOR STMT_NAME
-- Prepare the statement from a statement entered at run time
-- and specify that SQL write information about the number and
-- data type of select list items to the SQLDA.
PROCEDURE PREP_STMT
    (SQLCODE,
     :COMMAND_STRING CHAR (256));
    PREPARE STMT_NAME FROM :COMMAND_STRING;
PROCEDURE DESCRIBE
    (SQLCODE,
     SQLDA);
    DESCRIBE STMT_NAME SELECT LIST INTO SQLDA;
PROCEDURE OPEN_CURSOR
    (SQLCODE);
    OPEN CURSOR1;
PROCEDURE FETCH_CURSOR
    (SQLCODE,
     SQLDA);
    FETCH CURSOR1 USING DESCRIPTOR SQLDA;
PROCEDURE CLOSE_CURSOR
    (SQLCODE);
    CLOSE CURSOR1;
PROCEDURE ROLLBACK
    (SQLCODE);
    ROLLBACK;
```

例 11-6 (11-20) に、SQL の C_MOD_DYN_CURS モジュール内のルーチンをコールするプログラムを示します。例 11-5 (11-19) に示すように LANGUAGE 句で GENERAL を指定すると、プリコンパイル・ルーチン (command_string[256]) に渡された文字列を空白で埋める必要があります。SQL で GENERAL キーワードを指定すると、パラメータとして渡された文字列の NULL 終了記号を探さないからです。それ以外の場合、SQL はエラーを生成します。

一方、LANGUAGE 句で C を指定すると、SQL モジュールはパラメータとして渡された文字列の NULL 終了記号を探すので、フィールドの空白埋めは不要です。

例 11-6 ホスト言語プログラムにおけるパラメータ・マーカのない SELECT 文の実行

```
/* This program uses dynamic cursors to fetch a row from a table. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql_rdb_headers.h>
```

```

/* Maximum number of parameters or select list items is 50.*/
#define MAXPARAMS 50
/* Declare the SQLDA structure. */
struct SQLDA_STRUCT {
    char SQLDAID[8];
    int SQLDABC;
    short SQLN;
    short SQLD;
    struct {
        short SQLTYPE;
        short SQLLEN;
        char *SQLDATA;
        int *SQLIND;
        short SQLNAME_LEN;
        char SQLNAME[30];
    } SQLVAR[MAXPARAMS];
} *SQLDA;

main()
{
    /* General purpose local variables. */
    int i;
    long sqlcode;
    char command_string[256];
    /* Function prototypes. */
    void PREP_STMT ();
    void DESCRIBE ();
    void OPEN_CURSOR ();
    void FETCH_CURSOR ();
    void CLOSE_CURSOR ();
    void ROLLBACK ();
    /* Allocate SQLDA structures. */
    SQLDA = malloc(534);
    SQLDA->SQLN = 20;
    /* Get the SELECT statement at run time. */
    printf("\n Enter a select statement that retrieves only character ");
    printf("\n and integer data. \n");
    printf("\n Do not end the statement with a semicolon.\n");
    gets(command_string);
    /* Pad the string to end with spaces when using LANGUAGE keyword GENERAL
       in module. */
    for (i=strlen(command_string);i<=256;i++)
        command_string[i] = ' ';
    /* Prepare the SELECT statement. */
    PREP_STMT( &sqlcode, &command_string);
    if (sqlcode != 0)
        goto err;
    /* Write the information about the number and data type of the select list

```

```
    items to the SQLDA. */
    DESCRIBE( &sqlcode, SQLDA);
    if (sqlcode != 0)
        goto err;
    /* Open the cursor. */
    OPEN_CURSOR( &sqlcode );
    if (sqlcode != 0)
        goto err;
    /* Allocate memory. */
    for (i=0; i < SQLDA->SQLD; i++) {
        SQLDA->SQLVAR[i].SQLDATA = malloc( SQLDA->SQLVAR[i].SQLLEN );
        SQLDA->SQLVAR[i].SQLIND  = malloc( 2 );
    }
    /* Fetch a row. */
    FETCH_CURSOR( &sqlcode, SQLDA );
    if (sqlcode != 0)
        goto err;
    /* Use the SQLDA to determine the data type of each column in the row
       and print the column. For simplicity, test for only two data types. */
    for (i=0; i < SQLDA->SQLD; i++) {
        switch (SQLDA->SQLVAR[i].SQLTYPE) {
            case 453: /* Character */
                printf( "%.s ",SQLDA->SQLVAR[i].SQLLEN, SQLDA->SQLVAR[i].SQLDATA );
                break;
            case 497: /* Integer */
                printf( "%d", *SQLDA->SQLVAR[i].SQLDATA );
                break;
        }
    }
    /* Close the cursor. */
    CLOSE_CURSOR( &sqlcode );
    ROLLBACK( &sqlcode );
    return;
err:  printf("\n unexpected error %d", sqlcode);
      sql_signal();
      ROLLBACK( &sqlcode );
      exit (0);
}
```

11.6.2 パラメータ・マーカを伴う SELECT 文の実行

パラメータ・マーカを伴う SELECT 文では、動的カーソルか拡張動的カーソルを使用する以外に、パラメータ・マーカ用と選択リスト項目用の 2 つの SQLDA 構造を宣言し、使用する必要があります。

DESCRIBE...MARKERS 文と DESCRIBE...SELECT LIST 文の両方を使用してください。
DESCRIBE...MARKERS 文はパラメータ・マーカーに関する情報を SQLDA に書き込み、
DESCRIBE...SELECT LIST は選択リスト項目に関する情報を SQLDA に書き込みます。

プログラムでは、SQLDA の情報を使用して選択リスト項目とパラメータ・マーカーにストレージを割り当てます。さらに、割り当てたストレージにパラメータ・マーカーの値を書き込む必要があります。SQL は、この文を動的に実行するときにパラメータ・マーカーの値を代入します。

例 11-7 (11-23) に、プリコンパイル SQL を使用してパラメータ・マーカーを伴う SELECT 文を動的に実行するプログラム c_dyn_extcurs.sc を示します。このプログラムでは、動的に生成された任意の数の SELECT 文を拡張動的カーソルで処理します。

例 11-7 SQL プリコンパイル・プログラムにおけるパラメータ・マーカーを伴う SELECT 文の実行

```
/* This program dynamically executes SELECT statements that contain
   parameter markers. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql_rdb_headers.h>
/* Maximum number of parameters or select list items is 50.*/
#define MAXPARAMS 50
/* Declare two SQLDA structures -- one for the parameter markers and one
   for the select list items. */
struct SQLDA_STRUCT {
    char SQLDAID[8];
    int SQLDABC;
    short SQLN;
    short SQLD;
    struct {
        short SQLTYPE;
        short SQLLEN;
        char *SQLDATA;
        int *SQLIND;
        short SQLNAME_LEN;
        char SQLNAME[30];
    } SQLVAR[MAXPARAMS];
} *SQLDA, *SQLDA_SL;

main()
{
    long          SQLCODE;
    short         param, *indicator_param;
    int           loop_cntr;
    int           X;
    char          column_name[31];
```

```
    char          curs_name[10];          /* Name of statement */
/* An array to hold two SQL statements. */
    char          command_string[2][256];
    char          *command_string1;
/* Declare pointer variables for each data type. The program uses
   the pointers to dynamically allocate memory to hold values for
   parameter markers and select list items. */
    char          *charbuf;
    long          *longbuf;
    short         *smallintbuf;
    char          *datebuf;
    SQL_DATE_VMS  my_date_vms;
/* Allocate SQLDA structures. */
    SQLDA_SL = malloc(534);
    SQLDA_SL->SQLN = 12;
    SQLDA_SL = malloc(534);
    SQLDA_SL->SQLN = 12;
/* Declare the alias. */
    EXEC SQL DECLARE ALIAS FILENAME personnel;
/* Execute a loop to get two SELECT statements from the user. */
X=0;
while ( X < 2 )
{
    printf("Enter any SELECT statement. %n");
    printf("Do not end the statement with a semicolon. %n");
    gets(&command_string[X][0]);
/* Assign a statement name to each SELECT statement. */
    if (X == 0)
        strcpy(curs_name, "ONE");
    else
        if (X == 1)
            strcpy(curs_name, "TWO");
        else printf("%n No such statement. %n");
        X++;
}
/* Prompt the user for the name of a statement. */
    printf("%n Which statement do you want to execute, %n");
    printf("ONE or TWO? To quit, type EXIT.%n");
    gets(curs_name);
/* Assign the correct SQL statement to the cursor name. */
    if (strcpy(curs_name, "ONE" ))
        command_string1 = &command_string[0][0];
    if (curs_name == "TWO" )
        command_string1 = &command_string[1][0];
    if (curs_name == "EXIT")
        goto end_program;
/* Start an SQL transaction. */
```

```

EXEC SQL SET TRANSACTION READ WRITE;
if (SQLCODE != 0)
    goto err;
/* Prepare the statement from input entered at run time. */
EXEC SQL PREPARE stmt_id FROM :command_string1;
if (SQLCODE != 0)
    goto err;
/* Write information about the parameter markers into the SQLDA. */
EXEC SQL DESCRIBE stmt_id MARKERS INTO SQLDA;
if (SQLCODE != 0)
    goto err;
/* Write information about the select list items into the SQLDA. */
EXEC SQL DESCRIBE stmt_id SELECT LIST INTO SQLDA_SL;
if (SQLCODE != 0)
    goto err;
/* For each parameter marker, check the value of SQLDA.SQLVAR.SQLTYPE
to determine the data type of the parameter marker, and branch to
the appropriate code to prompt and store the value for that data type. */
param = 0;
for (param = 0; param < SQLDA->SQLD; param++)
{
    /* Handle a null-terminated column name and prompt with the column name. */
    strncpy (&column_name[0], &SQLDA->SQLVAR[param].SQLNAME[0],
            SQLDA->SQLVAR[param].SQLNAME_LEN);
    column_name[SQLDA->SQLVAR[param].SQLNAME_LEN] = '\0';
    printf ("%nEnter value for parameter '%s'", column_name);
    switch (SQLDA->SQLVAR[param].SQLTYPE)
    {
        case 453 :                /* character variable */
        case 449 :                /* variable char variable */
            printf ("%n(Maximum length is %d)", SQLDA->SQLVAR[param].SQLLEN);
            printf ("%ndsql> ");
            /* Allocate storage and get the address for the character
            string. */
            charbuf = (char *) malloc (SQLDA->SQLVAR[param].SQLLEN);
            /* Get the string. */
            gets (charbuf);
            /* Assign the address to SQLDA. */
            SQLDA->SQLVAR[param].SQLDATA = charbuf;
            break; /* Exit the switch block */
        case 497 :                /* integer variable */
            printf ("%ndsql> ");
            /* Allocate storage for the character string version of
            the integer to be entered. */
            charbuf = (char *) malloc (14);
            /* Get the string. */
            gets (charbuf);

```

```
/* Allocate storage for the integer and convert the string. */
longbuf = (long *) malloc (SQLDA->SQLVAR[param].SQLLEN);
*longbuf = atol (charbuf);
/* Assign the address to SQLDA */
SQLDA->SQLVAR[param].SQLDATA = (char *) longbuf;
break; /* Exit the switch block */
case 501 : /* short int variable */
    printf ("%ndsql> ");
    /* Allocate storage for the character string version of
       the short integer to be entered.*/
    charbuf = (char *) malloc (5);
    /* Get the string. */
    gets (charbuf);
    /* Allocate storage for the short int and convert the string. */
    smallintbuf = (short *) malloc (SQLDA->SQLVAR[param].SQLLEN);
    *smallintbuf = atoi (charbuf);
    /* Assign the address to SQLDA. */
    SQLDA->SQLVAR[param].SQLDATA = (char *) smallintbuf;
    break; /* Exit the switch block */
case 503 : /* date type variable */
    printf ("%nDate-time format is DD-MMM-YYYY HH:SS:MM.CC ");
    printf ("%ndsql> ");
    /* Allocate storage for the character string version of
       the date-time. */
    charbuf = (char *) malloc (23);
    /* Get the string. */
    gets (charbuf);
EXEC SQL SELECT
CAST(substring(:charbuf FROM 8 FOR 4) ||
-- Convert the month to a number.
(CASE SUBSTRING(:charbuf FROM 4 FOR 3)
    WHEN 'JAN' THEN '01'
    WHEN 'FEB' THEN '02'
    WHEN 'MAR' THEN '03'
    WHEN 'APR' THEN '04'
    WHEN 'MAY' THEN '05'
    WHEN 'JUN' THEN '06'
    WHEN 'JUL' THEN '07'
    WHEN 'AUG' THEN '08'
    WHEN 'SEP' THEN '09'
    WHEN 'OCT' THEN '10'
    WHEN 'NOV' THEN '11'
    WHEN 'DEC' THEN '12'
END) ||
-- Parse the day, hour, minutes, seconds.
SUBSTRING(:charbuf FROM 1 FOR 2) ||
SUBSTRING(:charbuf FROM 13 FOR 2) ||
```



```

SUBSTRING(:charbuf FROM 16 FOR 2) ||
SUBSTRING(:charbuf FROM 19 FOR 2) ||
SUBSTRING(:charbuf FROM 22 for 2)
AS DATE VMS) into :my_date_vms
FROM rdb$database LIMIT TO 1 ROW;
/* Assign the address to SQLDA. */
SQLDA->SQLVAR[param].SQLDATA = (char *) &my_date_vms;
break; /* Exit the switch block. */
default :
    printf("%n\nError: no data type match on %d",
           SQLDA->SQLVAR[param].SQLTYPE);
    break;
} /* End switch. */
} /* End the for loop. */
/* Declare the extended dynamic cursor, using the name the user enters. */
EXEC SQL DECLARE curs_name CURSOR FOR stmt_id;
if (SQLCODE != 0)
    goto err;
/* Open the cursor. */
EXEC SQL OPEN curs_name USING DESCRIPTOR SQLDA;
if (SQLCODE != 0)
    goto err;
/* Allocate memory. */
for (param=0; param < SQLDA_SL->SQLD; param++) {
    SQLDA_SL->SQLVAR[param].SQLDATA = malloc( SQLDA_SL->SQLVAR[param].SQLLEN );
    SQLDA_SL->SQLVAR[param].SQLIND = malloc( 2 );
}
/* Fetch the cursor and get information about the select list items
from the SQLDA. */
EXEC SQL FETCH curs_name USING DESCRIPTOR SQLDA_SL;
if (SQLCODE != 0)
    goto err;
/* Allocate storage for select list items in the statement string supplied
by the user. Also, allocate storage for indicator parameters associated
with the select list items, which indicate the occurrence of a NULL
value being returned for an item in the database. */
/* For each select list item, execute a loop. */
for (param = 0; param < SQLDA_SL->SQLD; param++)
{
    /* Allocate storage for an indicator array. */
    indicator_param = (short *) malloc (sizeof (short));
    /* Write the address of indicator_param to SQLDA_SL.SQLVAR.SQLIND */
    SQLDA_SL->SQLVAR[param].SQLIND = indicator_param;
    /* Check the value of SQLDA_SL.SQLVAR.SQLTYPE to determine the data
type of the select list item and branch to the allocation block
for that data type. */
    switch (SQLDA_SL->SQLVAR[param].SQLTYPE)

```

```

{
case 453 :                               /* character variable */
case 449 :                               /* variable char variable */
    /* Allocate storage and get address for the character string. */
    charbuf = (char *) malloc (SQLDA_SL->SQLVAR[param].SQLLEN);
    charbuf = (char *) malloc(SQLDA_SL->SQLVAR[param].SQLLEN+1);
    /* Copy the value to a string with a null terminator. */
    strncpy (charbuf, SQLDA_SL->SQLVAR[param].SQLDATA,
            SQLDA_SL->SQLVAR[param].SQLLEN);
    *(charbuf + SQLDA_SL->SQLVAR[param].SQLLEN) = '\0';
    /* Display the value in the buffer. */
    printf ("%s ", charbuf);
    break; /* Exit the switch block. */
case 497 :                               /* integer variable */
    /* Allocate storage for the integer. */
    longbuf = (long *) malloc (SQLDA_SL->SQLVAR[param].SQLLEN);
    printf ("%ld ", (long) (*SQLDA_SL->SQLVAR[param].SQLDATA));
    break; /* Exit the switch block. */
case 501 :                               /* short int variable */
    /* Allocate storage for the short int. */
    smallintbuf = (short *) malloc (SQLDA_SL->SQLVAR[param].SQLLEN);
    printf ("%d ", (short) (*SQLDA_SL->SQLVAR[param].SQLDATA));
    break; /* Exit the switch block. */
case 503 :                               /* date type variable */
    memcpy( &my_date_vms,SQLDA_SL->SQLVAR[param].SQLDATA,
            sizeof( SQL_DATE_VMS ) );
    datebuf = (char *) malloc (24);
    charbuf = (char *) malloc (24);
    EXEC SQL
        BEGIN
            SET :datebuf = CAST(:my_date_vms AS CHAR(24));
        END;
    if (SQLCODE != 0)
        goto err;
    printf ("%s ", datebuf);
    break; /* Exit the switch block. */
} /* End switch. */
} /* End the for loop. */

/* Close the cursor. */
EXEC SQL CLOSE curs_name;
if (SQLCODE != 0)
    goto err;
EXEC SQL COMMIT;
return;
err:  printf ("%n unexpected error %d", SQLCODE);
      sql_signal();
end_program:

```

```

        return;
    }

```

11.6.3 SQLDA2 構造と SQLERRD 構造によるパラメータ・マーカと SELECT 文のテスト

sql_dynamic サンプル・プログラムでは、SQLDA2 構造と SQLERRD 構造を使用して、ユーザーが入力した SQL 文の文字列がどんな処理が要求しているかを判断しています。例 11-8 (11-29) にメイン・プログラムのロジックを示します。

例 11-8 SELECT 文かどうかのテスト

```

/* Declare arrays for storage of original data types and allocate memory. */
mem_ptr output_save;
mem_ptr input_save;
/* If NULL sqlda was passed, then a statement is being prepared. */
if ((*input_sqlda == NULL) && (*output_sqlda == NULL))
{
    new_statement = TRUE;
    /* Allocate separate SQLDAs for parameter markers (SQLDA_IN) and select
       list items (SQLDA_OUT). Assign value of the constant MAXPARAMS to
       the SQLN field of both SQLDA structures. SQLN specifies to SQL the
       maximum size of the SQLDA. */
                                                                    (1)

    if ((sqlda_in = (sqlda) calloc (1, sizeof (sqlda_rec))) == 0)
    {
        printf ("YnYn*** Error allocating memory for sqlda_in: Abort");
        return (-1);
    }
    else /* Set number of possible parameters. */
        sqlda_in->sqln = MAXPARAMS;
    if ((sqlda_out = (sqlda) calloc (1, sizeof (sqlda_rec))) == 0)
    {
        printf ("YnYn*** Error allocating memory for sqlda_out: Abort");
        return (-1);
    }
    else
        /* Set number of possible select list items. */
        sqlda_out->sqln = MAXPARAMS;
    /* Copy name SQLDA2 to identify the SQLDA. */
    strncpy(&sqlda_in->sqldaid[0], "SQLDA2 ", 8);
    strncpy(&sqlda_out->sqldaid[0], "SQLDA2 ", 8);
    /* Call the SQL module language procedures prepare_stmt and
       describe_select which contain PREPARE and DESCRIBE...SELECT LIST
       statements to prepare the dynamic statement and write
       information about any select list items in it to sqlda_out. */

```

```
*stmt_id = 0; /* if <> 0 the BADPREPARE error results in the PREPARE*/
prepare_stmt (&SQLCA, stmt_id, psql_stmt); (2)
if (SQLCA.SQLCODE != sql_success)
{
    printf ("%n\nDSQL-E-PREPARE, Error %d encountered in PREPARE",
            SQLCA.SQLCODE);
    display_error_message();
    return (-1);
}
describe_select (&SQLCA, stmt_id, sqlda_out); (3)
if (SQLCA.SQLCODE != sql_success)
{
    printf ("%n\nDSQL-E-PREPARE, Error %d encountered in PREPARE",
            SQLCA.SQLCODE);
    display_error_message();
    return (-1);
}

/* Call an SQL module language procedure, describe_parm, which contains a
   DESCRIBE...MARKERS statement to write information about any parameter
   markers in the dynamic statement to sqlda_in. */
describe_parm (&SQLCA, stmt_id, sqlda_in); (4)
if (SQLCA.SQLCODE != sql_success)
{
    printf ("%n\n*** Error %d returned from describe_parm: Abort",
            SQLCA.SQLCODE);
    display_error_message();
    return (-1);
}

/* Save the value of the SQLCA.SQLERRD[1] field so that the program
   can determine if the statement is a SELECT statement or not.
   If the value is 1, the statement is a SELECT statement.*/
*is_select = SQLCA.SQLERRD[1];
.
.
.

/* Check to see if the prepared dynamic statement contains any parameter
   markers by looking at the SQLD field of sqlda_in. SQLD contains the
   number of parameter markers in the prepared statement. If SQLD is
   positive, the prepared statement contains parameter markers. The program
   executes a local routine, get_in_params, that prompts the user for
   values, allocates storage for those values, and updates the SQLDATA field
   of sqlda_in. */
if (sqlda_in->sqld > 0) (5)
{
    if ((status = get_in_params(sqlda_in, input_save)) != 0)
    {
        printf ("%nError returned from get_in_params. Abort");
        return (-1);
    }
}
```

```

    }
/* Check to see if the prepared dynamic statement is a SELECT statement
   by looking at the value in is_select, which stores the value of
   the SQLCA.SQLERRD[1] field. If that value is equal to 1, the
   the prepared statement is a SELECT statement. The program allocates
   storage for rows for SQL module language procedures to open and
   fetch from a cursor and displays the rows on the terminal.  */
if (*is_select) (6)
{
    if (new_statement == TRUE)      /* allocate buffers for output */
    {
        /* assign a unique name for the cursor */
        sprintf(cursor_name,"%2d",++cursor_counter);
        if ((status = allocate_buffers(sqlda_out)) != 0)
        .
        .
        .
/* If the SQLCA.SQLERRD[1] field is not 1, the prepared statement is not a
   SELECT statement and only needs to be executed. Call an SQL module language
   procedure to execute the statement, using information about parameter
   markers stored in sqlda_in by the local routine get_in_params.  */
    if (SQLCA.SQLERRD[1] != 1)
    {
        execute_stmt (&SQLCA, stmt_id, sqlda_in);
        .
        .
        .

```

次の番号は、例 11-8 (11-29) の番号に対応しています。

- (1) 宣言された SQLDA_IN 構造と SQLDA_OUT 構造にストレージを割り当てます。
- (2) SQL モジュール・プロシージャ内で PREPARE 文をコールし、SQL 文をチェックして識別子を割り当てます。
- (3) SQL モジュール・プロシージャ内で DESCRIBE...SELECT LIST 文をコールし、SQL 文の文字列内の選択リスト項目に関する情報を SQLDA_OUT に書き込みます。
- (4) SQL モジュール・プロシージャ内で DESCRIBE...MARKERS 文をコールし、SQL 文の文字列内のパラメータ・マーカに関する情報を SQLDA_IN に書き込みます。
- (5) SQLDA_IN.SQLD の値をチェックします。この値が正の場合は SQL 文の文字列にパラメータ・マーカが含まれ、プログラムは GET_IN_PARAMS サブルーチンに分岐します。この値が正でない場合は SQL 文の文字列にパラメータ・マーカがないので、プログラムの制御は分岐しません。
- (6) SQLERRD 配列の 2 番目の要素の値をチェックします。この値が 1 の場合、SQL 文の文字列は SELECT 文でプログラムは ALLOCATE BUFFERS サブルーチンに分岐します。

この値が 0 の場合、SQL 文の文字列は SELECT 文ではない (CALL 文でもない) ので、プログラムは EXECUTE 文をコールしてこれを処理します。

11.7 動的に生成された SQL 文の処理

PREPARE、DESCRIBE、拡張動的 DECLARE CURSOR、OPEN、FETCH などの一連の動的 SQL 文を使用すると、対応する動的 SQL 文で文とカーソル名にパラメータを指定することで、動的に生成された任意の数の SQL 文を同時に処理できます。明示的な文の名前のかわりにパラメータを使用すると、文とカーソルの名前をコンパイル時でなく実行時に指定でき、動的に生成された SQL 文をいくつでも処理できます。

SELECT 文以外の SQL 文の場合は、PREPARE 文で明示的な文の名前のかわりに文識別子として整数のパラメータを使用できます。PREPARE 文は文識別子の値を返し、この値を EXECUTE 文に渡すことができます。

SELECT 文では、SELECT 文以外の SQL 文の場合と同様に、PREPARE 文で明示的な文の名前のかわりに文識別子としてパラメータを使用できます。文識別子を使用して、プリコンパイルされた SQL 文を拡張動的 DECLARE CURSOR 文に渡します。拡張動的 DECLARE CURSOR 文を使用すると、ソース・コードでカーソルの名前を明示的に宣言するかわりにカーソル名にパラメータを指定できます。カーソル名と SELECT 文の両方を実行時に指定します。

実行時に任意の数の SQL 文を処理するには、文識別子とカーソル名のパラメータの値を保存し、プリコンパイルされた SQL 文または宣言されたカーソルを参照できるようにする必要があります。配列をセットアップし、上記のパラメータの値をデータ構造に格納してください。文識別子の値を保持する 1 つの配列をセットアップし、PREPARE 文か拡張動的 DECLARE CURSOR 文を実行するたびにパラメータの値を配列要素に割り当てます。SELECT 文を処理する場合は、カーソル名パラメータの値も配列に格納する必要があります。拡張動的 DECLARE CURSOR 文を実行するたびに、そのパラメータの値を配列要素に割り当てます。

次の項では、sql_multi_stmt_dyn.sqlada サンプル・プログラムの抜粋を使用して、SQL 文とカーソル名のパラメータの使用方法を説明します。プログラムでは、実行時にユーザーの入力から SQL 文を構成します。プログラムでは、一連の動的 SQL 文を使用して、動的に生成された多くの SQL 文を処理できます。

11.7.1 文識別子とカーソル名の格納

例 11-9 (11-33) は、サンプル・プログラムのコードの一部です。この例は、文識別子とカーソル名の値を保持する配列を宣言し、識別子とカーソル名の値を配列要素に割り当てる方法を示しています。さらに、サンプル・プログラムで各文の追跡に使用する配列と対応するパラメータを示します。

例 11-9 文識別子とカーソル名の配列への格納

```

-- Declare tables of procedure names, identifiers, and cursor names,
-- a counter, and an index.
-- The array PROCEDURE_NAMES holds the name that the user gives each
-- statement. You may decide to keep track of the statements in a
-- different way. The array PROCEDURE_IDS holds the statement
-- identifiers generated by the PREPARE statement. The array
-- CURSOR_NAMES holds the cursor names.
    PROCEDURE_NAMES : array(1..maxprocs) of string(1..name_strlng);
    PROCEDURE_IDS : array(1..maxprocs) of integer;
    CURSOR_NAMES : array(1..maxprocs) of string(1..name_strlng);
-- The NUMBER_OF_PROCS procedure increments the array elements when
-- the program stores information. The STMT_INDEX procedure increments
-- the array elements when the program looks up information.
    NUMBER_OF_PROCS : short_integer := 0;
    STMT_INDEX : short_integer := 0;
.
.
.
-- This section of the program stores the values of the parameters for
-- statement identifiers and cursor names in arrays.
number_of_procs := number_of_procs + 1;
-- Assign information about parameter markers and select list items to
-- array elements.
sqllda_in_array(number_of_procs) := sqllda_in;
sqllda_out_array(number_of_procs) := sqllda_out;
-- Assign the user-specified statement name to an array element. Assign
-- the statement identifier returned by the PREPARE statement to an
-- array element. In addition, if the statement is a SELECT statement,
-- assign the cursor name to an array element.
procedure_names(number_of_procs) := cur_name;
procedure_ids(number_of_procs) := cur_procid;
if cur_op(1) = 'R' then
    cursor_names(number_of_procs) := cur_cursor;
end if;

```

11.7.2 複数の SELECT 文以外の SQL 文の実行

一連の動的 SQL 文を使用して複数の SELECT 文以外の SQL 文を処理するには、PREPARE 文で文の名前のかわりに整数のパラメータを指定します。パラメータのデータ型は整数です。PREPARE 文はこのパラメータに文識別子を返し、この値を DESCRIBE...MARKERS 文と EXECUTE 文に渡します。

例 11-10 (11-34) の sql_multi_stmt_dyn.sqllda サンプル・プログラムの抜粋は、一連の動的 SQL 文を使用して任意の数の SELECT 文以外の SQL 文を処理する方法を示しています。

PREPARE_SQL プロシージャには、PREPARE 文、DESCRIBE...SELECT LIST 文、DESCRIBE...MARKERS 文が記述されています。プログラムでは、PREPARE_SQL プロシージャを使用して SELECT 文と SELECT 文以外の SQL 文の両方をプリコンパイルします。これに対して EXECUTE_SQL プロシージャでは、SELECT 文以外の SQL 文のみを実行します。PREPARE 文で SELECT 文以外の SQL 文のみを処理することが想定される場合は、DESCRIBE...SELECT LIST 文を記述する必要はありません。

例 11-10 複数の SELECT 文以外の SQL 文の実行

```
-- This procedure prepares a statement for dynamic execution from the
-- string passed to it. It can prepare any number of statements because
-- the statement is passed to it as the parameter cur_stmt.
procedure PREPARE_SQL is
    CUR_CURSOR : string(1..31) := (others => ' ');
    CUR_PROCID : integer := 0;
    CUR_STMT : string(1..1024) := (others => ' ');
begin
    -- Allocate separate SQLDAs for parameter markers (sqlda_in) and select list
    -- items (sqlda_out). Assign the value of the constant MAXPARMS (set in the
    -- declarations section) to the SQLN field of both SQLDA structures. SQLN
    -- specifies to SQL the maximum size of the SQLDA.
    sqlda_in := new sqlda_record;
    sqlda_in.sqln := maxparms;
    sqlda_out := new sqlda_record;
    sqlda_out.sqln := maxparms;
    -- Assign the SQL statement that was constructed in the procedure
    -- CONSTRUCT_SQL to the variable cur_stmt.
    cur_stmt := sql_stmt;
    -- Use the PREPARE statement to prepare the dynamic statement
    -- for dynamic execution from the string passed to it.
    EXEC SQL PREPARE :cur_procid FROM :cur_stmt;
    case sqlca.sqlcode is
        when sql_success => null;
        when others => raise syntax_error;
    end case;
    -- Use the DESCRIBE ... SELECT LIST statement to write information
    -- about the number and data type of any select list items in the
    -- statement to an SQLDA (specifically, the sqlda_out SQLDA specified).
    EXEC SQL DESCRIBE :cur_procid SELECT LIST INTO :sqlda_out;
    case sqlca.sqlcode is
        when sql_success => null;
        when others => raise syntax_error;
    end case;
    -- Use the DESCRIBE...MARKERS statement to write information about any
    -- parameter markers in the dynamic statement to sqlda_in. The statement
    -- writes information to an SQLDA (specifically, the sqlda_in SQLDA
    -- specified) about the number and data type of any parameter markers in
```



```

-- the prepared dynamic statement.  SELECT statements may also have
-- parameter markers.
EXEC SQL DESCRIBE :cur_procid MARKERS INTO sqlda_in;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise syntax_error;
end case;
.
.
.
procedure EXECUTE_SQL is
begin
--
-- This procedure dynamically executes a non-SELECT statement.
-- SELECT statements are processed by DECLARE CURSOR, OPEN CURSOR,
-- and FETCH statements.
--
-- The EXECUTE statement specifies an SQLDA (specifically, the sqlda_in
-- SQLDA specified) as the source of addresses for any parameter
-- markers in the dynamic statement.
--
-- The EXECUTE statement with the USING DESCRIPTOR clause also
-- handles statement strings that contain no parameter markers.
-- If a statement string contains no parameter markers, SQL sets
-- the SQLD field of the SQLDA to zero.
--
sqlda_in := new sqlda_record;
sqlda_in := sqlda_in_array(stmt_index);
cur_procid := procedure_ids(stmt_index);
EXEC SQL EXECUTE :cur_procid USING DESCRIPTOR :sqlda_in;
end EXECUTE_SQL;

```

11.7.3 複数の SELECT 文の実行

複数の SELECT 文を一連の動的 SQL 文で処理するには、PREPARE 文で明示的な文の名前のかわりにパラメータを使用します。PREPARE 文はこのパラメータに文識別子を返し、この値を DESCRIBE...SELECT LIST 文、DESCRIBE...MARKERS 文、拡張動的 DECLARE CURSOR 文に渡します。

さらに、拡張動的 DECLARE CURSOR 文ではカーソルの名前にパラメータを使用します。次に、カーソル名のパラメータを OPEN 文、FETCH 文、CLOSE 文に渡します。パラメータのデータ型は整数です。PREPARE 文の文識別子のデータ型も整数なので、PREPARE 文の前に初期化する必要があります。カーソル名パラメータは最大 31 文字の文字列です。

拡張動的 DECLARE CURSOR 文を使用する場合は、カーソル名と文識別子の両方に明示的な文の名前でなくパラメータを指定します。PREPARE、DESCRIBE、FETCH、OPEN、

CLOSE などの動的 SQL 文に対応する明示的なカーソルと文の名前のかわりにもパラメータを使用する必要があります。

例 11-11 (11-36) の `sql_multi_stmt_dyn.sqlada` サンプル・プログラムの抜粋は、一連の動的 SQL 文を使用して任意の数の SELECT 文を処理する方法を示しています。PREPARE_SQL プロシージャには、PREPARE 文、DESCRIBE...SELECT LIST 文、DESCRIBE...MARKERS 文、および拡張動的 DECLARE CURSOR 文が記述されています。メイン・プログラムでは、PREPARE_SQL 文を使用して SELECT 文と SELECT 文以外の SQL 文の両方をプリコンパイルします。

DISPLAY_DATA プロシージャには、パラメータを使用してカーソル名を参照する OPEN 文、FETCH 文、CLOSE 文が記述されています。

例 11-11 複数の SELECT 文の実行

```
-- This procedure prepares a statement for dynamic execution from the
-- string passed to it. It can prepare any number of statements because
-- the statement is passed to it as the parameter cur_stmt.
procedure PREPARE_SQL is
    CUR_CURSOR : string(1..31) := (others => ' ');
    CUR_PROCID : integer := 0;
    CUR_STMT : string(1..1024) := (others => ' ');
begin
    -- Allocate separate SQLDAs for parameter markers (sqlda_in) and select list
    -- items (sqlda_out). Assign the value of the constant MAXPARMS (set in the
    -- declarations section) to the SQLN field of both SQLDA structures. SQLN
    -- specifies to SQL the maximum size of the SQLDA.
    sqlda_in := new sqlda_record;
    sqlda_in.sqln := maxparms;
    sqlda_out := new sqlda_record;
    sqlda_out.sqln := maxparms;
    -- Assign the SQL statement that was constructed in the procedure
    -- CONSTRUCT_SQL to the variable cur_stmt.
    cur_stmt := sql_stmt;
    -- Use the PREPARE statement just to prepare the dynamic statement
    -- for dynamic execution from the string passed to it and not
    -- to write information about select list items to an SQLDA.
    EXEC SQL PREPARE :cur_procid FROM :cur_stmt;
    case sqlca.sqlcode is
        when sql_success => null;
        when others => raise syntax_error;
    end case;
    -- Use the DESCRIBE ... SELECT LIST statement to write information
    -- about the number and data type of any select list items in the
    -- statement to an SQLDA (specifically, the sqlda_out SQLDA specified).
    EXEC SQL DESCRIBE :cur_procid SELECT LIST INTO :sqlda_out;
    case sqlca.sqlcode is
```

```

        when sql_success => null;
        when others => raise syntax_error;
    end case;
-- Use the DESCRIBE...MARKERS statement to write information about any
-- parameter markers in the dynamic statement to sqlda_in. The statement
-- writes information to an SQLDA (specifically, the sqlda_in SQLDA
-- specified) about the number and data type of any parameter markers in
-- the prepared dynamic statement. SELECT statements may also have
-- parameter markers.
EXEC SQL DESCRIBE :cur_procid MARKERS INTO sqlda_in;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise syntax_error;
end case;
-- If the operation is 'R' (read), create a unique name for the cursor
-- name so that the program can pass the cursor name to the extended
-- dynamic DECLARE CURSOR statement.
if cur_op(1) = 'R' then
    cur_cursor(1) := 'C';
    cur_cursor(2..name_strlen) := cur_name(1..name_strlen - 1);
-- Declare the extended dynamic cursor.
EXEC SQL DECLARE :cur_cursor CURSOR FOR :cur_procid;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise syntax_error;
end case;
end if;
-- This section of the program stores the values of the parameters for
-- statement identifiers and cursor names in arrays.
number_of_procs := number_of_procs + 1;
-- Assign information about parameter markers and select list items to
-- array elements.
sqlda_in_array(number_of_procs) := sqlda_in;
sqlda_out_array(number_of_procs) := sqlda_out;
-- Assign the user-specified statement name to an array element. Assign
-- the statement identifier returned by the PREPARE statement to an
-- array element. In addition, if the statement is a SELECT statement,
-- assign the cursor name to an array element.
procedure_names(number_of_procs) := cur_name;
procedure_ids(number_of_procs) := cur_procid;
if cur_op(1) = 'R' then
    cursor_names(number_of_procs) := cur_cursor;
end if;
.
.
.
end PREPARE_SQL;

```

```
.
.
.
procedure DISPLAY_DATA is
.
.
.
begin -- procedure body DISPLAY_DATA
-- Before displaying any data, allocate buffers to hold the data
-- returned by SQL.
--
--     allocate_buffers;
-- Allocate and assign SQLDAs for the requested SQL procedure.
--
sqlda_in := new sqlda_record;
sqlda_in := sqlda_in_array(stmt_index);
sqlda_out := new sqlda_record;
sqlda_out := sqlda_out_array(stmt_index);
cur_cursor := cursor_names(stmt_index);
-- Open the previously declared cursor. The statement specifies
-- an SQLDA (specifically, sqlda_in) as the source of addresses for any
-- parameter markers in the cursor's SELECT statement.
--
EXEC SQL OPEN :cur_cursor USING DESCRIPTOR sqlda_in;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise unexpected_error;
end case;
-- Fetch the first row from the result table. This statement fetches a
-- row from the opened cursor and writes it to the addresses specified
-- in an SQLDA (specifically, sqlda_out).
--
EXEC SQL FETCH :cur_cursor USING DESCRIPTOR sqlda_out;
case sqlca.sqlcode is
-- Check to see if the result table has any rows.
    when sql_success => null;
    when stream_eof =>
        put_line("No records found.");
        new_line;
    when others => raise unexpected_error;
end case;
-- Set up a loop to display first row, fetch and display second and
-- subsequent rows.
    rowcount := 0;
    while sqlca.sqlcode = 0 loop
        rowcount := rowcount + 1;
--     Execute the DISPLAY_ROW procedure.
```

```

display_row;
-- To display only 5 rows, exit the loop if the loop counter
-- equals MAXROW (hardcoded as 5 in this program).
if rowcount = maxrows then exit; end if;
-- Fetch another row, exit the loop if no more rows.
EXEC SQL FETCH :cur_cursor USING DESCRIPTOR sqlda_out;
case sqlca.sqlcode is
    when sql_success => null;
    when stream_eof => exit;
    when others => raise unexpected_error;
end case;
end loop;
-- Close the cursor.
EXEC SQL CLOSE :cur_cursor;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise unexpected_error;
end case;
.
.
.
end DISPLAY_DATA;

```

11.8 この章で使用するサンプル・プログラムの格納場所

11.7 項 (11-32) で使用する `sql_multi_stmt_dyn.sqllada` プログラム全体は、サンプル・ディレクトリ内でオンラインで使用可能です。

`sql_dynamic` プログラムの全体もオンラインで利用できます。このプログラムは、SQL モジュール言語を使用した最も一般的な動的 SQL アプリケーションの例です。例 11-1 (11-6) のプログラムと同様に、`sql_dynamic` では端末から入力した SQL 文を受け取りますが、SELECT 文と SELECT 文以外の SQL 文でパラメータ・マーカーのあるものとなないものを処理します。また、このプログラムでは `SQLDA2` データ構造を使用します。したがって、日付時刻データ型に必要な情報を格納できます。

C によるメイン・プログラム `sql_dynamic` では、ユーザー向けの 2 種類の実装を示します。別々のドライバ・モジュールを使用してそれぞれの実装を示します。アプリケーション全体は次のソース・モジュールで構成されます。

- `sql_dynamic_driver_i.c` モジュール。ユーザーから SQL 文を受け取り、`sql_dynamic.c` モジュールに渡します。このドライバ・モジュールは、`sql_dynamic.c` および `sql_dynamic_c.sqlmod` と組み合わせて使用され、対話型 SQL をシミュレートします。スクリプト・ファイルに記載される SQL クエリーを、ユーザーが実行することもできます。
- `sql_dynamic_driver_m.c` モジュール。事前に設定された一連のクエリーを繰り返し実行します。`sql_dynamic_driver_m.c` モジュールは 1 つの INSERT 文、1 つの SELECT 文、

そして1つのUPDATE文で構成され、ユーザーが終了するまで繰り返し実行されます。このプログラムを変更すると、ファイルからクエリーを受け取ることができます。この例では、SQL\$DATABASE 論理名またはSQL_DATABASE 構成パラメータも定義する必要があります。

- sql_dynamic.c モジュール。ドライバ・モジュールから渡された文を処理して、結果を端末画面に表示します。
- sql_dynamic_c.sqlmod ファイル。ホスト言語モジュールでコールするすべてのサブルーチンを記述するSQLモジュール言語ファイルです。

sql_dynamic_i プログラムは、対話方式による使用と、スクリプト・ファイルによるクエリーの実行の両方が可能です。ファイルからクエリーを実行するには、次のようにファイルを作成してDynamicSQL> プロンプトで@filename.sql 文を使用します。

```
This program processes any valid SQL statement using dynamic SQL.
Enter the SQL statement to process on the following line(s), terminating your
statement with a semicolon <;> <<CR> or <CTRL-Z> to exit>:
DynamicSQL> @dynamic.sql;
```

たとえば、次のようなdynamic.sql ファイルを作成します。

```
ATTACH 'FILENAME personnel';
SELECT LAST_NAME,FIRST_NAME,EMPLOYEE_ID FROM EMPLOYEES
ORDER BY EMPLOYEE_ID;
SELECT LAST_NAME,BIRTHDAY FROM EMPLOYEES WHERE EMPLOYEE_ID > ?;
.
.
.
```

次の例に、sql_dynamic_i の対話モードによる実行を示します。

```
This program processes any valid SQL statement using dynamic SQL.
Enter the SQL statement to process on the following line(s), terminating your
statement with a semicolon <;> <<CR> or <CTRL-Z> to exit>:
DynamicSQL> ATTACH 'FILENAME mf_personnel';
The SQL statement to be executed dynamically is:
  ATTACH 'FILENAME mf_personnel';
Enter the SQL statement to process on the following line(s), terminating your
statement with a semicolon <;> <<CR> or <CTRL-Z> to exit>:
DynamicSQL> SELECT * FROM EMPLOYEES WHERE LAST_NAME = ?;
The SQL statement to be executed dynamically is:
  SELECT * FROM EMPLOYEES WHERE LAST_NAME = ?;
Enter value for parameter 'LAST_NAME'
(Maximum length is 14)Nash
-----
Field EMPLOYEE_ID:00168
Field LAST_NAME:Nash
Field FIRST_NAME:Norman
```

```
Field MIDDLE_INITIAL: NULL
```

```
  .
  .
  .
```

```
DynamicSQL> [Ctrl]+[Z]
```

```
$
```

sql_dynamic プログラムは、ユーザーが端末から入力した SQL 文を受け取り、これを処理するモジュールに渡します（このモジュールの完全なソース・コードを参照する場合は、サンプル・ディレクトリにアクセスしてください）。

ドライバ・モジュールから渡される SQL 文の文字列を処理するために、sql_dynamic は PREPARE 文と DESCRIBE 文を実行する SQL モジュール言語プロシージャをコールします。さらに、PREPARE 文と DESCRIBE 文で SQLCA 構造の SQLERRD フィールドに記述する情報を使用して SQL 文の文字列が SELECT 文かどうかをテストし、SQLDA2 構造を使用して文にパラメータ・マーカがあるかどうかをテストします。

テストの結果によって、sql_dynamic は次のいずれかまたは両方のサブルーチンを実行します。

- GET_IN_PARAMS (SQL 文の文字列にパラメータ・マーカがある場合)
GET_IN_PARAMS サブルーチンは、SQL 文の文字列内のすべてのパラメータ・マーカにストレージ・バッファを割り当て、バッファのアドレスを SQLDA2 構造に書き込み、各パラメータ・マーカに対応する値の入力をユーザーに要求し、その値を変数に格納します。
- ALLOCATE_BUFFERS (SQL 文の文字列が SELECT 文の場合)
ALLOCATE_BUFFERS サブルーチンは、SQL 文の文字列内のすべての選択リスト項目にストレージ・バッファを割り当て、バッファのアドレスを SQLDA2 構造に書き込みます。

SELECT 文の場合は、sql_dynamic は SQL モジュール言語プロシージャをコールしてカーソルをオープンし、ループを実行します。ループでは、OPEN 文で作成した結果テーブルの行ごとに次の処理を行います。

- SQL モジュール言語プロセッサをコールして行をフェッチします。
- 別のサブルーチン DISPLAY_ROW をコールします。このルーチンは ALLOCATE_BUFFERS サブルーチンが割り当てたストレージを使用して行の値を保持し、それを端末に表示します。

SELECT 文でない場合は、sql_dynamic はプリコンパイルされた SQL 文を EXECUTE 文で処理します。

sql_dynamic プログラム全体を参照する場合は、サンプル・ディレクトリからコピーしてください。

第Ⅳ部

プログラムの構造

第4部では、プログラムに対してより高度な制御を提供する SQL コンポーネントについて説明します。

- 複合文
- ストアド・ルーチン
- 外部ルーチン

SQL における複合文の使用

SQL では、フロー制御などの標準的なプログラミング構造を使用できますが、そのためには、複合文の中でこのような構造（プログラム構造）を使用する必要があります。複合文では、フロー制御文を含む 1 つ以上の SQL 文が単一文のコンテキストにグループ化されます。

以降の各項では、次の内容について解説します。

- 複合文の概要
- 複合文の使用によるデータベース・アプリケーションのパフォーマンスの向上
- 複合文の記述
- 複合文のアトム性の制御
- 複合文内のトランザクションの制御
- 複合文の動的な処理
- 複合文のデバッグ
- 行数、接続情報、トランザクション数など、複合文に関する情報の取得
- 複合文の例外および完了条件の処理

12.1 複合文の概要

複合文では、SQL 内でフロー制御文など、SQL の標準的なプログラミング構造群を使用して、SQL 文を単一文のコンテキストに統合することができます。複合文を含むプロシージャは、**マルチステートメント・プロシージャ**と呼ばれます。

複合文では、文の実行順序を制御することが可能になり、また複合文内では単純または複雑な意志決定も可能です。複合文には次の要素を記述できます。

- SQL データ操作文
- ローカル変数
- フロー制御文（IF 文や FOR ループなど）
- 文のアトム性を制御する句
- CALL 文。外部プロシージャまたはストアド・プロシージャを呼び出す
- トランザクションを開始または終了する文

複合文は SQL モジュール言語、埋込み SQL、対話型 SQL で使用でき、動的 SQL を使用して複合文を処理できます。さらに、ストアド・プロシージャまたはストアド・ファンクションを定義する場合は、複合文を使用する必要があります。

12.2 複合文によるパフォーマンスの向上

複合文を使用すると、データベース環境内で総合的なビジネス・トランザクションや複雑なプログラム制御が可能になります。

複数の SQL 文を複合文中に統合することで、クライアント / サーバー間のパフォーマンスが改善されます。リモート・アクセスのパフォーマンスは、Oracle Rdb システムとデータベース・システムにアクセスするアプリケーション・プログラムの相互作用が減少するので、特に顕著に向上します。さらに、すべての相互作用が 1 つのノード上で発生する場合にも、パフォーマンスが改善されます。

複合文、すなわちマルチステートメント・プロシージャには、次の利点があります。

- プロシージャ・コールを削減し、1 つの要求でデータベース・システムに関する多くの操作を可能にすることでアプリケーション・プログラムのロジックを簡素化し、文ごとにアプリケーション・プログラムに戻る必要をなくすることで、アプリケーションのパフォーマンスが向上します。
- 多くの場合、プロシージャ抽象化を実現します。**プロシージャ抽象化**とは、1 つの物理モジュールによって 1 つの論理データベース機能を実行することです。
- データベース要求とアクセスとの分離がより明確になります。

- クライアント / サーバー構成においてプロシージャ・コールを削減することでネットワーク・トラフィックを削減し、結果としてパフォーマンスを改善します。
- 実行速度が向上します（特に高パフォーマンス・データベース・アプリケーションの場合）。
- 複数の操作を 1 つのものとして処理できます（ATOMIC キーワードを使用）。
- アプリケーションとデータベースとのデータ型の不一致を削減します。

12.3 複合文の記述

複合文の記述では、BEGIN キーワードで複合文の先頭、END キーワードで末尾を指定します。次の対話型 SQL の例で示すように、複合文では、1 つ以上の SQL 文を記述できます。

```
SQL> BEGIN
1>     UPDATE DEPARTMENTS
2>         SET MANAGER_ID = '00167'
3>         WHERE DEPARTMENT_CODE = 'SALE';
4>     UPDATE JOB_HISTORY
5>         SET JOB_END = CURRENT_TIMESTAMP
6>         WHERE EMPLOYEE_ID = '00167' AND JOB_END IS NULL;
7>     INSERT INTO JOB_HISTORY
8>         (EMPLOYEE_ID, DEPARTMENT_CODE, JOB_CODE, JOB_START)
9>         VALUES
10>         ('00167', 'SALE', 'DMGR', CURRENT_TIMESTAMP);
11> END;
SQL>
```

SQL は、複合文に含まれる SQL 文を順次実行します。

複合文の中では、次のような内容を記述できます。

- 変数の宣言と割当て
- SQL データ操作文
複合文では、1 行形式の SELECT 文、UPDATE 文、INSERT 文、DELETE 文を使用できます。（1 行形式の SELECT 文以外の SELECT 文は使用できないので注意してください）。
- CASE 文
CASE 文は、複合文の中の代替文ブロックの 1 つを実行します。
- FOR 文
FOR 文は、クエリー式の各行について、SQL 文を実行します。FOR 文は、複合文でのカーソル機能を提供します。（複合文では、DECLARE CURSOR 文、OPEN 文、FETCH 文、CLOSE 文は使用できません）。

- IF 文
IF 文は、1 つ以上の SQL 文を、条件に従って実行します。
- LOOP 文
LOOP 文は、エラーが発生したり、LEAVE 文が実行されるまでの間、または WHILE 述語文の評価が TRUE である間、複合文に含まれる 1 つ以上の SQL 文を繰り返し実行します。
- LEAVE 文
LEAVE 文は、複合文やフロー制御文に含まれるプログラムを終了します。
- TRACE 文
TRACE 文は、文が評価する各値式について、値をログ・ファイルに書き込みます。
TRACE 文は、複合文のデバッグに使用します。
- CALL 文
CALL 文により、プログラムは、ストアド・プロシージャや外部プロシージャを起動します。外部プロシージャのコールでは、CALL 文は複合文で使用する必要があります。
- GET DIAGNOSTICS 文
GET DIAGNOSTICS 文は、現在の環境や、複合文の完了条件に関する情報を提供します。
- SIGNAL 文
SIGNAL 文は、複合文の例外条件に関する情報を提供します。
- 文のアトム性を制御する句
- トランザクションを開始または終了する文

複合文は、任意の深さにまでネストできます。後の項では、ネストした複合文を使った例を数多く掲載しています。

12.3.1 変数の宣言と割当て

複合文では、変数の宣言、値の割当て、SQL 文での変数の使用が可能です。複合文では、実行可能な SQL 文の前に、変数を宣言する必要があります。

変数を宣言するには、複合文の DECLARE 変数句を使用します。変数に値を割り当てるには、SET 文、または DECLARE 変数句の DEFAULT 句を使用します。

次の例では、変数 :mgrid を宣言し、値を割り当てて、UPDATE 文で変数を使用する方法を示します。

```
SQL> BEGIN
1>   DECLARE :mgrid CHAR(5);
2>   SET :mgrid = '00167';
3>   UPDATE DEPARTMENTS
4>         SET MANAGER_ID = :mgrid
```

```

5>          WHERE DEPARTMENT_CODE = 'SALE';
6> END;
SQL>

```

データ型は、明示的に指定するか、ドメイン名を使って暗黙的に指定することができます。ただし、複合文では、LIST データ型は指定または使用できません。

次の例のように、変数の宣言では、DEFAULT 句を使用すると、変数にデフォルト値を割り当てることができます。

```
DECLARE :mgrid CHAR(5) DEFAULT '00167';
```

DEFAULT キーワードのかわりに、等号 (=) を使用することもできます。

デフォルト値には、サブクエリー、条件、文字、日付や時間、数値など、任意の値式を設定できます。ただし、デフォルト値には、集計関数は使用できません。デフォルト値を指定しないで、データ型としてドメイン名を指定し、デフォルト値がドメインで定義されている場合、SQL は、ドメインのデフォルト値を変数のデフォルト値として使用します。

UPDATABLE または CONSTANT キーワードを指定することによって、変数を更新するかしないかを指定できます。次の例では、:mgrid は定数として宣言されているので、変数の値は変更できません。

```
DECLARE :mgrid CONSTANT CHAR(5) DEFAULT '00167' ;
```

CONSTANT キーワードを使用する場合は、デフォルト値を指定してください。

デフォルトは、UPDATABLE キーワードであり、変数の値の変更が可能です。SET 文、INTO 句、OUT または INOUT プロシージャ・パラメータで変数を使用する場合は、変数を更新可能として宣言する必要があります。

SQL モジュール・プロシージャや埋込み SQL 文の仮パラメータでは、NULL 値を処理するためにインジケータ・パラメータが必要になりますが、複合文の変数には、NULL 値を格納できます。複合文の変数には、インジケータ・パラメータは使用しません。

SET 文または DECLARE 変数文の DEFAULT 句を使って、変数に NULL 値を直接割り当てるか、SQL データ操作文で、変数を使って間接的に割り当てることができます。たとえば、次の複合文は、:job_end_date 変数に NULL 値を返します。

```

BEGIN
    DECLARE :job_end_date DATE;
    SELECT JOB_END INTO :job_end_date FROM JOB_HISTORY
           WHERE EMPLOYEE_ID = '00180';
END;

```

複合文で変数を宣言すると、変数のスコープは、複合文のスコープに限定されます。つまり、複合文が実行を終了すると、再度宣言しなければ、変数は使用できなくなります。

ネストした複合文では、外側の複合文で使用されている変数を、内側の複合文で宣言および値を割り当てることができます。この場合、内側の複合文で行った変数宣言と割当ては、外側の複合文での変数宣言と割当てを見えないように（つまり隠蔽）します。

次の SQL モジュールの抜粋は、ネストした複合文の例であり、外側の複合文で変数 `:inc` を宣言し、値を割り当てています。内側の複合文では、変数 `:inc` の宣言と割当てを再度実行していますが、異なる精度と値を使用しています。

```
BEGIN
-- Because :inc is declared as SMALLINT(3), SQL multiplies the salaries
-- of employees in NH by 1.055.
  DECLARE :inc SMALLINT(3);
  SET :inc = 1.055;
  UPDATE SALARY_HISTORY
    SET SALARY_AMOUNT = (SALARY_AMOUNT * :inc)
    WHERE SALARY_END IS NULL
      AND EMPLOYEE_ID IN (SELECT EMPLOYEE_ID FROM EMPLOYEES
                          WHERE STATE = 'NH');

  BEGIN
    -- Because :inc is declared as INTEGER(2), SQL rounds the number to
    -- two decimal places (1.07) before it multiplies the salaries
    -- of employees in MA.
    DECLARE :inc INTEGER(2);
    -- The SET statement assigns a new value to the variable :inc.
    SET :inc = 1.066;
    UPDATE SALARY_HISTORY
      SET SALARY_AMOUNT = (SALARY_AMOUNT * :inc)
      WHERE SALARY_END IS NULL
        AND EMPLOYEE_ID IN (SELECT EMPLOYEE_ID FROM EMPLOYEES
                            WHERE STATE = 'MA');

  END;
-- Because the following statement is in the outer compound statement,
-- :inc is declared as SMALLINT(3) with the value 1.055. As a result,
-- SQL multiplies the salaries of employees in CT by 1.055.
  UPDATE SALARY_HISTORY
    SET SALARY_AMOUNT = (salary_amount * :inc)
    WHERE SALARY_END IS NULL
      AND EMPLOYEE_ID IN (SELECT EMPLOYEE_ID FROM EMPLOYEES
                          WHERE STATE = 'CT');

END;
```

SET 文と DECLARE 変数句の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

12.3.2 IF 文の使用法

IF 文は、1 つ以上の SQL 文を、条件に従って実行します。

次の埋込み SQL プログラムの抜粋では、IF 文と、ホスト言語変数と複合文の変数を使って、UPDATE 文の実行条件を制御しています。

この例では、ホスト言語プログラムは、変数 :mgrid を宣言し、変数の値を複合文に渡します。複合文は、1 行形式の SELECT 文を使用して、INTO 句の変数に値を割り当てることによって、部門の現在のマネージャの MANAGER_ID を検索します。次に、IF 文は、現在の MANAGER_ID が、割り当てるマネージャの ID と異なるかどうかをチェックします。異なる場合、SQL は、THEN 句の UPDATE 文を実行します。MANAGER_ID が同じ場合は、SQL は THEN 句の UPDATE 文を実行しません。

```
-- Use the EXEC SQL keywords with compound statements, just as with simple
-- SQL statements.
EXEC SQL BEGIN
    DECLARE :cur_mgrid CHAR(5);
    SELECT MANAGER_ID INTO :cur_mgrid FROM DEPARTMENTS
        WHERE DEPARTMENT_CODE = 'SALE';
-- You must precede host language variables with a colon when you use them
-- in compound statements.
    IF :cur_mgrid <> :mgrid
    THEN
        UPDATE DEPARTMENTS
            SET MANAGER_ID = :mgrid
            WHERE DEPARTMENT_CODE = 'SALE';
    END IF;
END;
```

フロー制御文と同様に、IF 文は、任意のレベルまでネストできます。ただし、IF 文で ELSEIF 句を使用する方が、コードは読みやすくなります。次の SQL モジュールの抜粋は、IF 文をネストした場合と、ELSEIF 句を使用した場合の例です。

```
BEGIN
    DECLARE :state_code CHAR(2);
    DECLARE :mgrid CHAR(5);
    DECLARE :cur_mgrid CHAR(5);
    SET :mgrid = '00167';
    SELECT D.MANAGER_ID, E.STATE INTO :cur_mgrid, :state_code
        FROM DEPARTMENTS D, EMPLOYEES E
        WHERE DEPARTMENT_CODE = 'SALE'
            AND D.MANAGER_ID = E.EMPLOYEE_ID;
-- Outer IF statement.
-- If the manager is a new manager, update the table.
    IF :cur_mgrid <> :mgrid
    THEN
        UPDATE DEPARTMENTS
            SET MANAGER_ID = :mgrid
            WHERE DEPARTMENT_CODE = 'SALE';
-- Nested IF statement.
```

```
-- Give the new manager a raise. The amount depends on the manager's
-- state.
IF :state_code = 'NH'
THEN
    UPDATE SALARY_HISTORY S
        SET SALARY_AMOUNT = (salary_amount * 1.05)
        WHERE SALARY_END IS NULL
            AND EMPLOYEE_ID = :mgrid;
-- ELSEIF clause of nested IF statement.
ELSEIF :state_code = 'MA'
THEN
    UPDATE SALARY_HISTORY S
        SET SALARY_AMOUNT = (salary_amount * 1.07)
        WHERE SALARY_END IS NULL
            AND EMPLOYEE_ID = :mgrid;
-- ELSE clause executes if the :state_code is neither NH nor MA.
ELSE
    UPDATE SALARY_HISTORY S
        SET SALARY_AMOUNT = (salary_amount * 1.06)
        WHERE SALARY_END IS NULL
            AND EMPLOYEE_ID = :mgrid;
END IF;
END IF;
END;
```

上の例では、税差額を考慮するかしないかによって、従業員の給与を増額しています。

12.3.3 CASE 文の使用方法

CASE 文は、いくつかの操作をリストしておき、実行時の式の値によって、実行する操作を1つ選択します。

次の例は、12.3.2 項 (12-6) の IF...ELSEIF の例と同じ処理を実行しますが、IF 文のネストではなく、CASE 文を使用しています。

```
BEGIN
    DECLARE :state_code CHAR(2);
    DECLARE :mgrid CHAR(5);
    DECLARE :cur_mgrid CHAR(5);
    SET :mgrid = '00167';
    SELECT D.MANAGER_ID, E.STATE INTO :cur_mgrid, :state_code
        FROM DEPARTMENTS D, EMPLOYEES E
        WHERE DEPARTMENT_CODE = 'SALE'
            AND D.MANAGER_ID = E.EMPLOYEE_ID;
-- Outer IF statement.
IF :cur_mgrid <> :mgrid
THEN
```

```

UPDATE DEPARTMENTS
    SET MANAGER_ID = :mgrid
    WHERE DEPARTMENT_CODE = 'SALE';
-- CASE statement. Give the new manager a raise. Because of differences in
-- state taxes, the amount varies with the state in which the manager resides.
CASE :state_code
    WHEN 'NH'
    THEN UPDATE SALARY_HISTORY S
        SET SALARY_AMOUNT = (salary_amount * 1.05)
        WHERE SALARY_END IS NULL
        AND EMPLOYEE_ID = :mgrid;
    WHEN 'MA'
    THEN UPDATE SALARY_HISTORY S
        SET SALARY_AMOUNT = (salary_amount * 1.07)
        WHERE SALARY_END IS NULL
        AND EMPLOYEE_ID = :mgrid;
-- The ELSE clause executes if the :state_code is neither NH or MA.
ELSE UPDATE SALARY_HISTORY S
    SET SALARY_AMOUNT = (salary_amount * 1.06)
    WHERE SALARY_END IS NULL
    AND EMPLOYEE_ID = :mgrid;
END CASE;
END IF;
END;

```

CASE 文に ELSE 句がなく、式の値がいずれの WHEN 句にも一致しない場合、SQL は例外を生成します。

12.3.4 LOOP 文の使用法

LOOP 文は、WHILE 述語文の評価が TRUE である間、またはエラーが発生するか、LEAVE 文が実行されるまでの間、複合文に含まれる 1 つ以上の SQL 文を繰り返し実行します。

次の例は、LOOP 文を使って、会社の管理階層を下位から追跡する方法を示しています。この例は、1 人の従業員から処理を開始し、その従業員のスーパーバイザを検索します。そのスーパーバイザのスーパーバイザを検索するという処理を、EMPLOYEE_ID が SUPERVISOR_ID に一致するまで繰り返します。

```

-- This procedure begins with the employee with EMPLOYEE_ID 00180 and
-- finds his supervisor, tracing the hierarchy of the company.
BEGIN
    DECLARE :emp_id CHAR(5);
    DECLARE :sup_id CHAR(5);
    DECLARE :id CHAR(5);
    SET :emp_id = '00180';
-- Initialize :id and :sup_id to be different values, so that the LOOP
-- will execute at least once.

```

```
SET :id = '00000';
SET :sup_id = '00001';
WHILE :id <> :sup_id
LOOP
    SELECT EMPLOYEE_ID, SUPERVISOR_ID INTO :id, :sup_id
    FROM JOB_HISTORY
    WHERE EMPLOYEE_ID = :emp_id
    AND JOB_END is NULL;
    SET :emp_id = :sup_id;
END LOOP;
END;
```

12.3.5 FOR 文の使用法

FOR 文は、SELECT 式が返した各行について、複合文を含む SQL 文を実行します。FOR 文は、複合文で、カーソルに似た機能を提供します。FOR 文は、カーソルを暗黙的に宣言およびオープンし、行を繰り返しフェッチして、カーソルが END-OF-STREAM に達したらカーソルをクローズします。

FOR 文は、AS EACH ROW OF 句で指定する SELECT 文を評価します。FOR ループを反復するたびに、SQL は、レコード・フィールドの AS EACH ROW OF 句で指定した各列の値を格納します。FOR キーワードのすぐ後に、レコードの名前を指定します。たとえば、次の例では、SQL は、レコード :jobrec の列 MINIMUM_SALARY と MAXIMUM_SALARY を格納します。

```
-- The :jobrec variable represents a record that holds columns from the
-- selected row.
FOR :jobrec
    AS EACH ROW OF TABLE CURSOR JOB_CURSOR FOR
-- The select expression specifies that only two columns be stored in the
-- record :jobrec.
    SELECT MINIMUM_SALARY, MAXIMUM_SALARY FROM JOBS
    WHERE MINIMUM_SALARY < 20000
```

レコードに格納されている列を参照するには、次のように、列名をレコード名で修飾する必要があります。

```
IF :jobrec.MINIMUM_SALARY > :jobrec.MAXIMUM_SALARY
```

FOR 文は、任意の深さにまでネストできます。

次の例は、1 つの FOR 文を使って、最低給与が \$200000.00 未満の行をすべて選択し、選択した行を、最低給与にまで増額します。さらに、この例では IF 文をネストして、最高給与が最低給与よりも低い場合は、これを増額します。

```
BEGIN
-- The :jobrec variable represents a record that holds columns from the
-- selected row.
```

```

FOR :jobrec
  AS EACH ROW OF TABLE CURSOR JOB_CURSOR FOR
-- The select expression.
  SELECT MINIMUM_SALARY, MAXIMUM_SALARY FROM JOBS
        WHERE MINIMUM_SALARY < 20000
  DO
-- Update the current row in the JOB_CURSOR.
  UPDATE JOBS
-- No need to qualify the column names.
        SET MINIMUM_SALARY = MINIMUM_SALARY * 1.10
        WHERE CURRENT OF JOB_CURSOR;
-- If the minimum salary is now greater than the maximum salary, increase the
-- maximum salary.
  IF :jobrec.MINIMUM_SALARY > :jobrec.MAXIMUM_SALARY
  THEN
    UPDATE JOBS
      SET MAXIMUM_SALARY = MAXIMUM_SALARY * 1.10
      WHERE CURRENT OF JOB_CURSOR;
  END IF;
END FOR;
END;

```

カーソルを使用する UPDATE 文と同様に、WHERE CURRENT OF 句を使用すると、名前を指定したカーソルが配置されている行のみを更新するように指定できます。

次は、FOR 文の中で IF 文をネストした例であり、SQL モジュール・プロシージャに複合文が含まれています。このプロシージャでは、従業員がこれまでに受け取った最高昇給率を計算します。ホスト言語プログラムは、パラメータ :ID を使って、従業員 ID をプロシージャに渡します。プロシージャは、パラメータ :RESULT を使って、最大昇給率をホスト言語プログラムに返します。

```

PROCEDURE max_raise (SQLCODE,
                    :ID CHAR(5),
                    :RESULT REAL);

BEGIN
  DECLARE :last_salary INTEGER(2) DEFAULT 0;
  DECLARE :pct_raise INTEGER(2) DEFAULT 0;
  SET :RESULT =0;
  FOR :salhist_rec
    AS EACH ROW OF
      SELECT SALARY_AMOUNT FROM SALARY_HISTORY
        WHERE EMPLOYEE_ID = :ID
        ORDER BY SALARY_START ASC
  DO
    IF :last_salary <> 0
    THEN
      SET :pct_raise = (:salhist_rec.salary_amount - :last_salary)

```

```
                                / :last_salary;
END IF;
IF :pct_raise > :RESULT
THEN
    SET :RESULT = :pct_raise;
END IF;
SET :last_salary = :salhist_rec.salary_amount;
END FOR;
END;
```

SQL モジュールでの複合文の使用の詳細は、4.4 項 (4-15) を参照してください。

12.3.6 複合文でのラベルの使用法

複合文や、FOR 文および LOOP 文は、ラベルで名前を付けることができます。複合文にラベルを付けると、複合文や FOR および LOOP 文をネストし、特定の文からプログラムを終了する場合に特に便利です。(LEAVE 文で複合文を終了する方法の詳細は、12.3.7 項 (12-13) を参照してください)。

プログラムを読みやすくするために、複合文の最後には、同じラベルを指定してください。次の SQL モジュールの抜粋では、複合文でラベル UPD_MGRID を指定しています。

```
UPD_MGRID:      -- Beginning label for compound statement
BEGIN
    DECLARE :mgrid CHAR(5);
    SET :MGRID = '00167';
    UPDATE DEPARTMENTS
        SET MANAGER_ID = :mgrid
        WHERE DEPARTMENT_CODE = 'SALE';
END
UPD_MGRID      -- Ending label for compound statement
;              -- Ending semicolon for compound statement
```

開始ラベルには、ラベル名の後にコロン (:) を指定してください。次のような場合に、ラベルを使用できます。

- 複合文の最初と最後
BEGIN キーワードの前には開始ラベル、END キーワードの後には終了ラベルを指定します。
ただし、次の点に注意してください。
 - 対話型 SQL やプリコンパイルした SQL では、複合文の内側にネストした複合文でラベルは使用できますが、最も外側の複合文ではラベルは使用できません。
 - SQL モジュール言語では、最も外側の複合文も、内側の複合文と同様に、ラベルの使用は可能です。最も外側の複合文でラベルを指定しないと、SQL は、SQL モジュール・プロシージャの名前を複合文に割り当てます。

- LOOP 文の最初と最後
WHILE キーワードの前に開始ラベル、END LOOP キーワードの後に終了ラベルを指定します。
- FOR 文の最初と最後
FOR キーワードの前に開始ラベル、END FOR キーワードの後に終了ラベルを指定します。

12.3.7 LEAVE 文の使用法

LEAVE 文は、複合文やフロー制御文に含まれるプログラムを終了します。LEAVE 文を使用すると、次のようなプログラム構造から、プログラムを終了できます。

- 複合文
- FOR 文
- LOOP 文
- SQL モジュール・プロシージャ

LEAVE 文を使用するためには、プログラムを終了したい文にラベルを指定します。たとえば、次の文は、loop1 というラベルが付いている LOOP 文で終了します。

```
LEAVE loop1;
```

SQL モジュール・プロシージャを終了するには、プロシージャの名前を、LEAVE 文の引数として指定します。

次の例では、スーパーバイザになった経験がある従業員の数と、スーパーバイザとマネージャの両方になった経験がある従業員の数をカウントします。この例では、従業員の JOB_CODE が 'DMGR' の場合、LEAVE 文を使って内側の FOR 文を終了し、同じ従業員を複数回カウントしないようにしています。

```
BEGIN
    DECLARE :supnum INTEGER;
    DECLARE :bothnum INTEGER;
    SET :supnum = 0;
    SET :bothnum = 0;
    -- Count the number of employees who have been supervisors.
    jobloop:
    FOR :job_rec
        AS EACH ROW OF
            SELECT DISTINCT(EMPLOYEE_ID) FROM JOB_HISTORY
                WHERE JOB_CODE = 'DSUP'
    DO
        SET :supnum = :supnum + 1;
    -- Count the number of employees who have been managers, as well as
    -- supervisors.
```

```
innerjobloop:
FOR :innerjob_rec
  AS EACH ROW OF
    SELECT EMPLOYEE_ID, JOB_CODE FROM JOB_HISTORY
      WHERE EMPLOYEE_ID = :job_rec.employee_id
DO
  IF :innerjob_rec.JOB_CODE = 'DMGR'
  THEN
    SET :bothnum = :bothnum + 1;
-- Exit from the inner FOR loop.
    LEAVE innerjobloop;
  END IF;
END FOR innerjobloop;
END FOR jobloop;
END;
```

ネストの深さには関係なく、複合文、FOR 文、LOOP 文を終了できます。上の例で、LEAVE innerjobloop ではなく、LEAVE jobloop を指定すると、SQL は、内側の FOR ループだけでなく、外側の FOR ループも終了します。

12.3.8 ストアド・プロシージャや外部プロシージャの起動

ストアド・プロシージャや外部プロシージャの起動には、CALL 文を使用します。複合文で CALL 文を使用すると、IN パラメータには任意の値式を使用できますが、単一文で CALL 文を使用する場合は、ホスト変数、数値リテラル、文字列リテラルに限られます。(パラメータには dbkey や集計関数は使用できません)。

外部プロシージャを起動するには、複合文の中で CALL 文を使用してください。ストアド・プロシージャを起動するには、単一文または複合文で CALL 文を使用してください。

次の例では、ストアド・プロシージャと、それをコールする複合文を示しています。

```
-- Create the module that contains the stored procedure.
CREATE MODULE DEPT_BUDG_MOD
  LANGUAGE SQL
  PROCEDURE DEPT_BUDG_PROC ( IN :incr INTEGER (2), INOUT :new_budg INTEGER,
                              INOUT :cur_budg INTEGER );
BEGIN
  SELECT BUDGET_PROJECTED INTO :cur_budg
    FROM DEPARTMENTS
   WHERE DEPARTMENT_NAME = 'Engineering';
  SET :new_budg = :cur_budg + (:cur_budg * :incr);
  UPDATE DEPARTMENTS
    SET BUDGET_PROJECTED = :new_budg
   WHERE DEPARTMENT_NAME = 'Engineering';
END;
END MODULE;
```



```
-- Call the DEPT_BUDG_PROC stored procedure.
BEGIN
    DECLARE :cur_budg_var INTEGER;
    DECLARE :new_budg_var INTEGER;
    DECLARE :incr_var INTEGER (2) DEFAULT .08;
    CALL DEPT_BUDG_PROC ( :incr_var, :new_budg_var, :cur_budg_var);
END;
```

ストアド・プロシージャやストアド関数の中で CALL 文を使用すれば、他のストアド・プロシージャをコールできます。この方法を使用すると、システム・リソースが対応できる限り、ストアド・プロシージャを任意の深さにまでネストできます。ただし、再帰はサポートしていません。つまり、現在の CALL 文で使用しているストアド・プロシージャをコールすることは不可能です。

12.4 複合文のアトム性の制御

個々の SQL 文と同様に、複合文にもアトム性と呼ばれるトランザクション特性があります。アトム性は、例外が発生した場合に、単一 SQL 文または複合文がどのように処理されるかを定義します。すべての単一 SQL 文の実行は、常にアトミック・イベントとして処理されます。アトミック・イベントとは、文が正常に終了するか、または SQL が例外を返した場合には、文の実行が取り消されるかのいずれかであることを表します。ただし、複合文のアトム性はユーザーが制御できます。デフォルトでは、複合文はアトミックではありません。

複合文のアトム性は、次の例に示すように、BEGIN キーワードの後に ATOMIC キーワードと NOT ATOMIC キーワードを付けて指定します。

```
SQL> BEGIN ATOMIC
1>     UPDATE DEPARTMENTS
2>         SET MANAGER_ID = '00167'
3>         WHERE DEPARTMENT_CODE = 'SALE';
4>     UPDATE JOB_HISTORY
5>         SET JOB_END = CURRENT_TIMESTAMP
6>         WHERE EMPLOYEE_ID = '00167' AND JOB_END IS NULL;
7>     INSERT INTO JOB_HISTORY
8>         (EMPLOYEE_ID, DEPARTMENT_CODE, JOB_CODE, JOB_START)
9>         VALUES
10>         ('00167', 'SALE', 'DMGR', CURRENT_TIMESTAMP);
11> END;
SQL>
```

外側の複合文がアトミックであることを指定する場合は、ネストした複合文に対して ATOMIC キーワードを指定します。

ATOMIC な複合文は、NOT ATOMIC な文の中でネストできます。

複合文で SQL に例外が発生した場合の処置は、複合文の定義が ATOMIC か NOT ATOMIC によって決まります。それぞれ、次の処置を行います。

- **ATOMIC**

複合文を構成する個々の SQL 文は、1 つの単位としては、正常に終了することも失敗することもあります。複合文を構成するすべての SQL 文が正常に終了すると、複合文のブロックが全体として正常に終了します。

しかし、SQL 文の 1 つで例外が発生すると、SQL は、すべての変更をロールバックし、失敗した文より後の文は実行せず、その時点で複合文を終了します。

SQL は、文が失敗しても変数の割当ては取り消しません。

- **NOT ATOMIC (デフォルト)**

文が失敗した時点で正常に終了している SQL 文はロールバックされません。NOT ATOMIC な複合文内では、一部の文が正常に終了することもあります。しかし、SQL 文が例外を返すと、SQL はただちに NOT ATOMIC の複合文内の処理を終了します。複合文が終了する原因になった文の不完全な作業は、常にロールバックされます。

前の例では、文がアトミックなので、すべての文が完了するか 1 つも完了しないかのいずれかの結果になります。

次の例で、アトム性の原理を詳しく説明します。INSERT INTO JOB_HISTORY 文で例外が発生すると仮定します。この状況で、ネストされた複合文内のどの SQL 文が完了し、どの SQL 文が実行を取り消されるかを示します。

```
BEGIN NOT ATOMIC
  BEGIN ATOMIC
    UPDATE DEPARTMENTS ... (4) (12-16)
    UPDATE JOB_HISTORY ... (4) (12-16)
  END;
  BEGIN ATOMIC
    INSERT INTO DEPARTMENTS ... (2) (12-16)
    INSERT INTO JOB_HISTORY ... (1) (12-16) (2) (12-16)
    INSERT INTO SALARY_HISTORY ... (3) (12-16)
  END;
END;
```

上の例に示す複合文を実行すると、SQL は例外が発生するまで SQL 文を実行します。次の説明は、上の例の番号に対応しています。

(1) INSERT INTO JOB_HISTORY 文で例外が発生します。

(2) 内側の複合文は ATOMIC なので、SQL は、次に示す文をロールバックします。

- INSERT INTO DEPARTMENTS
- INSERT INTO JOB_HISTORY

(3) INSERT INTO JOB_HISTORY 文によって発生した例外の影響で内側の複合文が終了するので、SQL は INSERT INTO SALARY_HISTORY 文を実行しません。

(4) 外側の複合文は NOT ATOMIC なので、SQL は UPDATE 文の実行を取り消しません。

ATOMIC 複合文と NOT ATOMIC 複合文のどちらにおいても、たとえばデータなしなどの完了状態が返された場合、SQL の実行は継続します。

ATOMIC 複合文には、SET TRANSACTION、COMMIT、ROLLBACK の各文は使えません。

12.5 複合文内のトランザクションの制御

複合文には、SET TRANSACTION、COMMIT、ROLLBACK の各文を挿入できます。ただし、これは NOT ATOMIC な複合文に限定されており、ATOMIC な複合文には使用できません。

複合文の中でトランザクションを開始し、終了すると、Oracle Rdb とアプリケーション・プログラムのやり取りが減るためパフォーマンスが改善されます。

たとえば、次のように複合文で SET TRANSACTION、COMMIT、ROLLBACK の各文を使用できます。

- 1つの複合文の中でトランザクションを開始し、終了します。
- 1つの複合文の中でトランザクションを開始し、別の SQL プロシージャの中でそのトランザクションを終了します。
- 1つの SQL プロシージャの中でトランザクションを開始し、複合文の中でそのトランザクションを終了します。または、1つの複合文の中でトランザクションを開始し、複合文の外でそのトランザクションを終了します。
- 複合文の中でフロー制御文を使用し、特定の条件に基づいてトランザクションを開始または終了します。

次の例では、パラメータ値が 0 の場合は読取り / 書込みトランザクションを開始してデータを更新し、それ以外の場合は読取り専用トランザクションを開始します。

```
PROCEDURE update_or_select
    (SQLSTATE,
     :upd_sel,
     :mgrid);
BEGIN
    IF :upd_sel = 0
    THEN
        SET TRANSACTION READ WRITE;
        UPDATE DEPARTMENTS
            SET MANAGER_ID = :mgrid
            WHERE DEPARTMENT_CODE = 'SALE';
    ELSE
        SET TRANSACTION READ ONLY;
        SELECT MANAGER_ID INTO :mgrid
            FROM DEPARTMENTS
            WHERE DEPARTMENT_CODE = 'SALE';
```

```
END IF;  
END;
```

1つの複合文の中では複数のデータベースを参照できないため、複数のデータベースに対するトランザクションの開始、コミット、ロールバックはできません。たとえば、複合文の外で複数のデータベースに対するトランザクションを開始した場合、複合文の中ではトランザクションのコミットもロールバックもできません。

DECLARE TRANSACTION 文は、複合文内のトランザクションに対しても単一 SQL 文内のトランザクションに対しても同じ効果があります。

アクセス・モードやステータスなど、複合文内のトランザクションに関する情報を取得するには、GET DIAGNOSTICS 文を使用します。GET DIAGNOSTICS 文の詳細は、12.8 項 (12-21) を参照してください。

トランザクションの詳細は、第 16 章 (16-1) を参照してください。

12.6 複合文の動的な処理

複合文内で動的 SQL 文を使用することはできません。ただし、SQL では、次の動的 SQL 文を使用することで、複合文を処理し、実行することができます。

- PREPARE
- DESCRIBE
- EXECUTE
- EXECUTE IMMEDIATE

たとえば、ホスト言語プログラムで複合文をパラメータ :STMT に代入し、文識別子をパラメータ :DYN_STMT_ID に代入した場合は、次の手順に従って文を動的に実行するよう準備します。

```
PROCEDURE prepare_stmt  
(SQLSTATE,  
 :DYN_STMT_ID INTEGER,  
 :STMT CHAR(16000));  
PREPARE :DYN_STMT_ID FROM :STMT;
```

12.7 複合文のデバッグ

複合文の記述、テストには、特別な注意が必要です。Oracle Rdb 内で実行されている複合文に対しプログラミング言語シンボリック・デバッガを使用することはできないため、複合文のデバッグや実行フローのトレースを行うのは困難です。

ただし、TRACE 文を使用すると、複合文で使用する変数の内容を監視できます。トレース・ロギングを有効化するには、SET FLAGS 'TRACE' 文を使用するか、

RDMS\$DEBUG_FLAGS 論理名または RDB_DEBUG_FLAGS 構成パラメータを "Xt" と定義します。X は常に大文字、t は常に小文字です。そしてこの 2 文字を二重引用符で囲みます。次に例を示します。

```
$ DEFINE RDMS$DEBUG_FLAGS "Xt"
```

論理名 RDMS\$DEBUG_FLAGS_OUTPUT または構成パラメータ RDB_DEBUG_FLAGS_OUTPUT を使用すると、TRACE 文の出力を指定できます。

SET FLAGS 'TRACE' 文は、論理名 RDMS\$DEBUG_FLAGS と構成パラメータ RDB_DEBUG_FLAGS より優先されます。必ずデータベースに接続してから、SET FLAGS 'TRACE' 文を使用してください。ただし、この文は複合文の中では使用できません。複合文の前に使用してください。

TRACE 文を使用すると、リテラル、サブクエリー、変数など、値式の値をトレースできます。次の例では、SET FLAGS 文を使用してトレースを有効化する方法と TRACE 文の使用方法を説明し、TRACE 文の出力を示します。

```
-- Enable trace logging.
SET FLAGS 'TRACE';
BEGIN
    DECLARE :state_code CHAR(2);
    DECLARE :mgrid CHAR(5);
    DECLARE :cur_mgrid CHAR(5);
    SET :mgrid = '00167';
-- Trace a built-in function.
    TRACE 'Trace the current time ', CURRENT_TIMESTAMP;
    SELECT D.MANAGER_ID, E.STATE INTO :cur_mgrid, :state_code
        FROM DEPARTMENTS D, EMPLOYEES E
        WHERE DEPARTMENT_CODE = 'SALE'
            AND D.MANAGER_ID = E.EMPLOYEE_ID;
-- Trace variables.
    TRACE 'After SELECT mgrid is ', :mgrid;
    TRACE 'After SELECT cur_mgrid is ', :cur_mgrid;
    IF :cur_mgrid <> :mgrid
    THEN
        UPDATE DEPARTMENTS
            SET MANAGER_ID = :mgrid
            WHERE DEPARTMENT_CODE = 'SALE';
-- Trace variables.
    TRACE 'After UPDATE mgrid is ', :mgrid;
    IF :state_code = 'NH'
    THEN
-- Trace a literal.
        TRACE 'Entering IF loop.';
        UPDATE SALARY_HISTORY S
            SET SALARY_AMOUNT = (salary_amount * 1.05)
```

```
        WHERE SALARY_END IS NULL
          AND EMPLOYEE_ID = :mgrid;
      END IF;
    END IF;
  END;
~Xt: Trace the current time 1994-02-16:12:32:20.46
~Xt: After SELECT mgrid is 00167
~Xt: After SELECT cur_mgrid is 00205
~Xt: After UPDATE mgrid is 00167
~Xt: Entering IF loop.
```

接頭辞 "~Xt:" を非表示にするには、SET FLAGS 'NOPREFIX' 文を使用します。

列名の値をトレースするには、SQL は、列のコンテキストを認識できなければなりません。たとえば、次の例に示すように、FOR 文内の列名をレコード名で修飾すると列名をトレースできます。

```
-- Enable trace logging and turn off display of the prefix.  You can combine
-- the TRACE and NOPREFIX keywords in the same statement.
SET FLAGS 'TRACE, NOPREFIX';
BEGIN
  DECLARE :supnum INTEGER;
  SET :supnum = 0;
  FOR :job_rec
    AS EACH ROW OF
      SELECT EMPLOYEE_ID, JOB_CODE FROM JOB_HISTORY
  DO
    -- Trace the values of the columns JOB_CODE and EMPLOYEE_ID.
    TRACE 'Job code: ', :job_rec.job_code;
    TRACE 'Employee_id: ', :job_rec.employee_id;
    IF :job_rec.JOB_CODE = 'DSUP'
    THEN
      SET :supnum = :supnum + 1;
    END IF;
  END FOR;
END;
Job code: SPMG
Employee_id: 00164
Job code: DMGR
Employee_id: 00164
.
.
.
```

トレース・ロギングを有効化しなければ、TRACE 文は実行状態ではないため、このプロシージャの実行によるオーバーヘッドはありません。

12.8 複合文に関する情報の取得

複合文で最後に実行した SQL 文に関する情報を取得するには、GET DIAGNOSTICS 文を使用します。GET DIAGNOSTICS 文は、直前の SQL 文の実行に関する診断情報を取り出します。GET DIAGNOSTICS 文は、**診断領域**と呼ばれる Oracle Rdb データベース構造から、次のような診断情報を取得します。

- 行に関するステータス情報
GET DIAGNOSTICS 文は、INSERT、UPDATE、DELETE 文で処理した行の数、または FOR 文でフェッチした行数を返します。
- トランザクションに関する情報
GET DIAGNOSTICS 文は、アクセス・モード、分離レベル、コミットまたはロールバックしたトランザクションの数など、トランザクションに関する情報を返します。
- 現在の接続の名前
- コール側のルーチンの名前
ルーチンは、GET DIAGNOSTICS 文を使用して、そのルーチンをコールしているルーチンの名前を参照できます。

GET DIAGNOSTICS 文は、12.9 項 (12-22) の SQLSTATE または SQLCODE ステータス・パラメータの値を返します。

GET DIAGNOSTICS 文は、複合文でしか使えません。

次の例では、GET DIAGNOSTICS 文を使って、FOR 文でフェッチした行の数を返します。

```
SET FLAGS 'TRACE, NOPREFIX';
BEGIN
-- Declare a variable to hold the number of the current row.
  DECLARE :currow INTEGER;
-- The :jobrec variable represents a record that holds columns from the
-- selected row.
  FOR :jobrec
    AS EACH ROW OF TABLE CURSOR JOB_CURSOR FOR
      SELECT MINIMUM_SALARY FROM JOBS
        WHERE MINIMUM_SALARY < 20000
  DO
-- Update the current row in the JOB_CURSOR.
    UPDATE JOBS
      SET MINIMUM_SALARY = MINIMUM_SALARY * 1.10
      WHERE CURRENT OF JOB_CURSOR;
-- Use the GET DIAGNOSTICS statement to retrieve the number of rows that have
-- been fetched by the FOR statement.
    GET DIAGNOSTICS
      :currow = CURRENT_ROW;
-- Use the TRACE statement to print out the number of the current row.
```

```
        TRACE 'Current Row: ', :currow;  
    END FOR;  
END;  
Current Row: 1  
Current Row: 2  
Current Row: 3  
Current Row: 4  
Current Row: 5  
Current Row: 6
```

上記の例では、TRACE 文で、現在の行の値を端末に出力しています。

GET DIAGNOSTICS 文の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

12.9 例外条件と完了条件の処理

複合文の SQL 文では、例外条件や完了条件が発生します。SQL は、次のように、例外条件と完了条件を異なる方法で処理します。

- SQL 文の実行で、制限違反などの例外条件を返した場合、SQL は、複合文の実行を強制終了します。SIGNAL 文を使用すれば、SQL 文に関する情報をアプリケーションに渡すことができます。
- SQL 文を実行し、データがないなどの完了条件が発生した場合、SQL は、複合文の実行を続行します。GET DIAGNOSTICS 文を使うと、最後に実行した SQL 文のステータスを取得できます。

12.9.1 例外条件の取得

複合文の SQL 文がエラーを検出した場合、SIGNAL 文を使って例外が発生させ、ステータスをアプリケーションに返すことができます。

SIGNAL 文の引数として文字列を指定します。Oracle Rdb は、SQLSTATE ステータス・パラメータの値として、これを使用します。Oracle Rdb が SIGNAL 文を検出すると、現在の複合文またはルーチン、およびコール中のルーチンをすべて終了し、制御をアプリケーションに返します。この処理において、Oracle Rdb は、指定した SQLSTATE の値をアプリケーションに渡します。

文字列は、ANSI/ISO SQL 規格に準拠した SQLSTATE の値でなければなりません。つまり、アルファベット大文字（A ～ Z）または数字（0 ～ 9）のみを使用してください。たとえば、次の文は、数値が範囲外の場合に、アプリケーションにシグナルを発行します。

```
SIGNAL '22003';
```

Oracle Rdb は、SQLSTATE の値として 22003 をアプリケーションに返します。この処理では、SQLSTATE 値を SQLCODE の値にマッピングします。上の例では、SQLSTATE の値を

SQLCODE にマッピングすると、-304 になります。SQLSTATE の値を複数の SQLCODE 値にマッピングできる場合や、SQLSTATE 値が不明の場合、Oracle Rdb は、SQLCODE の値として -1042 を返します。

次の例は、複合文が 0 による除算を実行しようとするときに、SIGNAL を使用方法を示します。

```
-- Calculate the difference between two job titles and put the results in the
-- table PAY_ADJUST.
BEGIN
  DECLARE :job_code1, :job_code2 CHAR (5);
  DECLARE :min_sal, :min_sal2 INTEGER;
  DECLARE :diff INTEGER;
  -- In a stored procedure, you pass the values using parameters,
  -- rather than hard coding values.
  SET :job_code1 = 'PRGM';
  SET :job_code2 = 'MENG';
  SELECT MINIMUM_SALARY INTO :min_sal FROM JOBS
    WHERE JOB_CODE = :job_code1;
  SELECT MINIMUM_SALARY INTO :min_sal2 FROM JOBS
    WHERE JOB_CODE = :job_code2;
  IF :min_sal - :min_sal2 = 0
    -- If condition is not trapped, it results in division by 0 in ELSE
    -- clause.
    THEN SIGNAL '22012';
    -- Calculate the difference in minimum salaries and store the percentage
    -- in the PAY_ADJUST table.
  ELSE SET :diff = :min_sal - :min_sal2;
    INSERT INTO PAY_ADJUST
      (JOB_CODE1, JOB_CODE2, DIFF)
    VALUES (:job_code1, :job_code2, :min_sal / :diff);
  END IF;
END;
%RDB-E-SIGNAL_SQLSTATE, routine "(unnamed)" signaled SQLSTATE "22012"
```

上の例では、Oracle Rdb は、SQLSTATE の値として 22012 を返し、0 による除算が発生することを示します。

SIGNAL 文の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

12.9.2 完了条件の取得

GET DIAGNOSTICS 文は、直前の SQL 文の実行に関する診断情報を取り出します。GET DIAGNOSTICS 文は、SQLSTATE または SQLCODE ステータス・パラメータから、診断情報を取得します。

SQLSTATE ステータス情報は EXCEPTION...RETURNED_SQLSTATE 句、SQLCODE ステータス情報は EXCEPTION...RETURNED_SQLCODE 句を使って取得できます。

次の例では、EXCEPTION...RETURNED_SQLSTATE 句を使って、SQLSTATE ステータス・パラメータの値を返します。

```
SET FLAGS 'TRACE, NOPREFIX';
BEGIN ATOMIC
    DECLARE :mgrid CHAR(5);
    DECLARE :cur_mgrid CHAR(5);
    DECLARE :dept_code CHAR(4);
    DECLARE :sqlstate_var CHAR(5);
    SET :mgrid = '00166';
    SET :dept_code = 'SALE';
    SELECT MANAGER_ID INTO :cur_mgrid FROM DEPARTMENTS
        WHERE DEPARTMENT_CODE = :dept_code;
-- The GET DIAGNOSTICS statement returns an SQLSTATE code of 00000 because the
-- SELECT statement executes successfully.
    GET DIAGNOSTICS
        EXCEPTION 1 :sqlstate_var = RETURNED_SQLSTATE;
    TRACE 'After SELECT, SQLSTATE is: ', :sqlstate_var;
-- Test the value of SQLSTATE and take some action if it is 00000.
    IF :sqlstate_var = '00000'
    THEN
        IF :cur_mgrid <> :mgrid
        THEN
            UPDATE DEPARTMENTS
                SET MANAGER_ID = :mgrid
                WHERE DEPARTMENT_CODE = 'Ssss';
-- Because there is no department with a DEPARTMENT_CODE 'Ssss', the
-- GET DIAGNOSTICS statement returns an SQLSTATE code of 02000.
            GET DIAGNOSTICS
                EXCEPTION 1 :sqlstate_var = RETURNED_SQLSTATE;
            TRACE 'After UPDATE, SQLSTATE is: ', :sqlstate_var;
        END IF;
    END IF;
END;
After SELECT, SQLSTATE is: 00000
After UPDATE, SQLSTATE is: 02000
```

GET DIAGNOSTICS 文の詳細と、SQLSTATE ステータス・パラメータの値については、『Oracle Rdb7 SQL Reference Manual』を参照してください。

ストアド・ルーチンの使用方法

この章では、ストアド・ルーチン（ストアド・プロシージャとストアド関数）の詳細や、Oracle Rdb データベースのモジュールにストアド・ルーチンを格納する方法、アプリケーション・プログラムでストアド・ルーチンを実行する方法について説明します。この章の各項では、次の内容を説明します。

- ストアド・ルーチンについて
- ストアド・ルーチンを使用する利点
- ストアド・ルーチンを格納するストアド・モジュールの作成
- ストアド・プロシージャの起動
- ストアド関数の起動
- ストアド・ルーチンの削除
- 依存性の追跡
- ストアド・ルーチンが無効化される条件
- 無効化されたストアド・ルーチンの再有効化または再作成

13.1 ストアド・ルーチンについて

ストアド・ルーチンとは、ストアド・プロシージャやストアド関数を指します。

ストアド・プロシージャは、1つ以上の SQL 文によって、Oracle Rdb データベース上で実行する一連の操作です。ストアド・プロシージャは、入力パラメータを受け取り、出力パラメータで結果を返します。

ストアド関数は、1つ以上の SQL 文によって、Oracle Rdb データベース上で実行する一連の操作です。ストアド関数は、入力パラメータを受け取り、RETURN 文で、結果を1つ返します。

ストアド関数に含まれる SQL 文は、幅広いデータベース操作を単位として実行します。ストアド・ルーチンは、複数文のプロシージャが提供する、プログラムのようなプロシージャ機能を使います。これにより、1つのアプリケーション・ドメインとして、確立した操作をデータベースにシフトでき、ホスト言語ライブラリと同様に、プログラミング・コミュニティとして機能できます。

ストアド・ルーチンは、3GL アプリケーション・プロシージャと類似しています。ストアド・ルーチンは、コンパイルの対象となる**ストアド・モジュール**内に格納され、更新、削除、挿入などの操作をカプセル化します。アプリケーション・プロシージャと異なるのは、ストアド・ルーチンは、C や Fortran などの 3GL プログラム言語ではなく、SQL で記述するという点です。

ストアド・モジュールは、テーブルやビューのように、Oracle Rdb データベース内にスキーマ・オブジェクトとして格納されています。

これに対して、**ノン・ストアド・モジュール**は、Oracle Rdb データベースの外にあるファイルに格納されます。ノン・ストアド・モジュールを使用すると、複数のデータベースを参照できますが、ストアド・モジュールを使用すると、ストアド・モジュールが格納されているデータベースしか参照できません。

その他の点では、ストアド・モジュールとノン・ストアド・モジュールは同じです。ノン・ストアド・モジュールの作成に使用する SQL モジュール言語は、ストアド・モジュールの作成で使用する SQL 構文と類似しています。ストアド・モジュールには1つ以上のストアド・ルーチンを格納しなければならないように、ノン・ストアド・モジュールにも、1つ以上のプロシージャを格納する必要があります。

Oracle Rdb データベースの外に格納したモジュールの作成および使用方法の詳細は、3 章 (3-1) と 4 章 (4-1) を参照してください。

13.2 データベースにルーチンを格納する利点

データベースにルーチンを格納すると、次のような利点があります。

- 操作のカプセル化
ストアド・ルーチンでは、他のユーザーにも使用できるように、データベースに操作

(または一連の操作)を配置することができます。複数のストアド・ルーチンをまとめると、アプリケーションによるリンクが可能な実行可能オブジェクトのライブラリのような役割を果たします。たとえば、ADD EMPLOYEE という操作があり、このアプリケーションの操作では、いくつかの入力情報が必要になるとします。ストアド・ルーチンの複雑な内部操作を知らなくても、アプリケーションで1つのルーチンをコールすれば、従業員レコードをデータベースに追加できます。

現在のところ、プログラマは、このような操作をオブジェクト・ライブラリを使用して実行していますが、ストアド・ルーチンを使用すると、オブジェクト・ライブラリのメンテナンスが不要になり、SQL クエリーから機能を把握できるようになります (この点がオブジェクト・モジュールのライブラリとは異なります)。

- 権限の継承
ストアド・ルーチンの定義では、ストアド・ルーチンの参照先となる基本オブジェクト (テーブル、ビュー、制約など) へのアクセスを取得する必要があります。ストアド・ルーチンの作成で認証識別子を指定すると、ストアド・ルーチンの他のユーザーは、定義を実行したユーザーのアクセス権限を継承します。これにより、他のユーザーは、ルーチンの基本オブジェクトへのアクセス権限を直接持っていないとしても、定義したルーチンを使用することができます。ルーチンを起動するユーザーは、ストアド・ルーチンの起動に必要な EXECUTE 権限があれば、他のデータベース・オブジェクト権限は必要ありません。ユーザーは、基盤となるテーブルにアクセス権限がなくても、一定の操作は実行できます。
- クライアント / サーバー処理
クライアント・システムには十分なリソースがないため、SQL 文を含んだデータベース要求をローカルに格納およびメンテナンスすることはできません。サーバー・システムにストアド・ルーチンを格納することにより、クライアント・システム・アプリケーションは、ストアド・ルーチンに含まれる複雑なデータベース操作に簡単にアクセスおよび実行できます。アクセスが発生すると、ストアド・ルーチンが格納されている Oracle Rdb データベースに接続し、ルーチンとパラメータの名前を指定します。
- メタデータの依存性の制御を改善
ルーチンをデータベースに格納して使用すると、Oracle Rdb は、ルーチンの参照先となるメタデータ・オブジェクトを追跡します。

13.3 ストアド・モジュールの作成

CREATE MODULE 文を使用すると、Oracle Rdb データベースのストアド・モジュールを作成できます。これは、CREATE TABLE 文でデータベース・テーブルを作成するのと同様です。CREATE MODULE 文では、SQL 言語による1つ以上のストアド・ルーチンを定義します。このモジュールは、Oracle Rdb データベースにルーチンを格納する SQL のメカニズムの役割を果たします。

例 13-1 (13-4) に、ストアド・モジュール DATA_UPDATE の定義を示します。

例 13-1 ストアド・モジュールの作成

```
CREATE MODULE DATA_UPDATE (1) (13-4)
  LANGUAGE SQL (2) (13-4)
  AUTHORIZATION LAWLER (3) (13-4)
  DECLARE LOCAL TEMPORARY TABLE MODULE.DATA_UPDATE_TAB
    (EMPLOYEE_ID ID_DOM,
     SALARY INTEGER(2)) (4) (13-4)
  <routine definitions> (5) (13-5)
END MODULE; (6) (13-5)
```

ストアド・モジュールを作成する場合は、次の句を使用します（例 13-1 (13-4) の番号に対応します）。

(1) CREATE MODULE module-name

CREATE MODULE 文によって、Oracle Rdb データベースの永続オブジェクトとして DATA_UPDATE というモジュールを作成します。ストアド・モジュールの名前は、アプリケーション内で一意でなければなりません。ストアド・モジュールとノン・ストアド・モジュールの両方にアクセスするアプリケーションでは、アプリケーションで使用するストアド・ルーチンと同名のノン・ストアド・モジュールを使用できません。

(2) LANGUAGE SQL 句

LANGUAGE 句に指定した SQL 引数は、ストアド・モジュール内のルーチンを SQL 文でコールすることを示しています。一方、ノン・ストアド・モジュール内の LANGUAGE 句の引数は、このモジュールのプロシージャをコールするプログラムが記述されたホスト言語を表します。

(3) AUTHORIZATION 句

AUTHORIZATION 句で LAWLER と指定された AUTHORIZATION 識別子によって、Oracle Rdb はモジュールの作成者または定義者を特定します。

ストアド・モジュールの定義で AUTHORIZATION 識別子を指定すると、そのストアド・モジュールは**定義者権限のモジュール**と呼ばれます。この種のモジュールによって、モジュールの EXECUTE 権限を持つユーザーはルーチンが参照する基本的なスキーマ・オブジェクトに関する権限がなくても、モジュールの任意のルーチンを実行できます。ルーチンは、それを実行するユーザーのユーザー名ではなく、モジュール定義者のユーザー名で実行されます。このように、ユーザーがスキーマ・オブジェクトに直接アクセスするのでなく、ストアド・ルーチンのコールを介してスキーマ・オブジェクトにアクセスできるのは、大きな利点です。

一方、ストアド・モジュールの定義で AUTHORIZATION 句を省略すると、そのストアド・モジュールは**実行者権限のモジュール**と呼ばれます。この種のモジュールでは、特定のモジュールの EXECUTE 権限を持つユーザーは、実行するルーチンに伴う基本的なスキーマ・オブジェクトのすべてに関する権限も持っていなければなりません。

(4) 宣言されたローカル一時テーブル

ストアド・モジュールでは、宣言されたローカル一時テーブルを使用できます。テーブルを宣言したモジュール内からのみのテーブルを参照できます。宣言されたローカル

時テーブル内のメタデータとデータは、ストアド・モジュールの外部からは参照できません。一時テーブルの詳細は、『Oracle Rdb7 Guide to Database Design and Definition』を参照してください。

- (5) ルーチンの定義
 ルーチンの定義では、ルーチンの名前、パラメータ、1つの単純文または複合文を指定します。
 ストアド・プロシージャの定義の作成については、13.3.1 項 (13-5) を参照してください。ストアド・ファンクションの定義の作成については、13.3.2 項 (13-7) を参照してください。
- (6) END MODULE
 ストアド・モジュールの定義の終了は、END MODULE キーワードで明示する必要があります。

13.3.1 ストアド・プロシージャの作成

CREATE MODULE 文にストアド・プロシージャの定義を記述すると、ストアド・プロシージャが作成されます。

例 13-2 (13-5) に、DATA_UPDATE モジュール内の NEW_SALARY_PROC ストアド・プロシージャの定義を示します。

例 13-2 ストアド・プロシージャの作成

```
CREATE MODULE DATA_UPDATE
LANGUAGE SQL
AUTHORIZATION LAWLER
DECLARE LOCAL TEMPORARY TABLE MODULE.DATA_UPDATE_TAB
  (EMPLOYEE_ID ID_DOM,
   SALARY INTEGER(2))
PROCEDURE NEW_SALARY_PROC (1)
  (:ID CHAR (5), (2)
   :NEW_SALARY INTEGER (2)); (2)
BEGIN (3)
  UPDATE SALARY_HISTORY (3)
    SET SALARY_END = CURRENT_TIMESTAMP
      WHERE EMPLOYEE_ID = :ID AND SALARY_END IS NULL;
  INSERT INTO SALARY_HISTORY (EMPLOYEE_ID, SALARY_AMOUNT,
    SALARY_START, SALARY_END)
    VALUES (:ID, :NEW_SALARY, CURRENT_TIMESTAMP, NULL);
  INSERT INTO MODULE.DATA_UPDATE_TAB
    (EMPLOYEE_ID, SALARY)
    SELECT EMPLOYEE_ID, SALARY_AMOUNT FROM SALARY_HISTORY
      WHERE CAST(SALARY_START AS DATE ANSI) =
        CURRENT_DATE;
END; (3)
```

END MODULE;

ストアド・プロシージャを作成する場合は、次の句を使用します（例 13-2 (13-5) の番号に対応します）。

(1) PROCEDURE 句

PROCEDURE 句には、ストアド・プロシージャの名前を指定します。

プロシージャ名は、モジュール定義内および Oracle Rdb データベースのモジュールに格納された他のルーチンのすべてを通じて一意でなければなりません。また、同じデータベース内で定義された外部ルーチン名とも区別されなければなりません。

(2) パラメータ・リスト

プロシージャのパラメータ・リストには、次のようなパラメータ・モード、パラメータ名、SQL データ型を指定します。

- ストアド・プロシージャのパラメータには、IN、OUT、INOUT の 3 つのパラメータ・モードがあります。
パラメータの宣言でモードを省略すると（例 13-2 (13-5) のように）、SQL はパラメータの使用方法によってパラメータのモードを決定します。ストアド・プロシージャ・パラメータのパラメータ・モードを決定するには、"ストアド・プロシージャ内の SQL 文が値を受け取るか（IN パラメータ・モード）、値を生成するか（OUT パラメータ・モード）、または両方の操作を行うか（INOUT パラメータ・モード）"を確認します。
- ストアド・プロシージャ定義内で使用する個々のパラメータの名前は、プロシージャ内で一意でなければなりません。
- ストアド・プロシージャ内では、LIST OF BYTE VARYING 以外の任意のデータ型を指定できます。宣言では、データ型の代わりにドメイン名も指定できます。たとえば、:ID CHAR (5) を :ID ID_DOM として宣言できます。

ストアド・プロシージャのパラメータに関しては、次の点に注意してください。

- ストアド・プロシージャ内では、ステータス・パラメータ（SQLCODE、SQLSTATE、SQLCA）は宣言できません。ただし、ストアド・プロシージャ内で GET DIAGNOSTIC 文を使用すると、SQLCODE または SQLSTATE の値を調べることができます。12.9 項 (12-22) では、GET DIAGNOSTIC 文を使用して SQLCODE と SQLSTATE の値を返し、完了の状態を処理する方法を説明します。SIGNAL 文を使用して例外の状態を処理する方法も示します。
例 13-4 (13-10) に示すように、ノン・ストアド・プロシージャからストアド・プロシージャをコールする場合は、ノン・ストアド・モジュール内の任意のプロシージャと同様に、ノン・ストアド・プロシージャ内でステータス・パラメータを宣言する必要があります。
- ストアド・プロシージャは NULL 値をサポートします。したがって、ストアド・プロシージャ・パラメータにはインジケータ・パラメータを使用できません。
ただし、コールされたストアド・プロシージャが NULL 値を返すと思われる場合

は、ストアド・プロシージャをコールするノン・ストアド・プロシージャ内で必ずインジケータ・パラメータを宣言し、確実に NULL 値を処理できるようにしてください。インジケータ・パラメータは、対応するメイン・パラメータに格納された値が NULL かどうかを示します。

(3) BEGIN...END ブロック

ストアド・プロシージャでは、1つの単純文か1つの複合文を使用できます。この例では、BEGIN...END ブロックで囲まれた複合文を示します。

ストアド・プロシージャのコールについては、13.4 項 (13-10) を参照してください。CREATE MODULE 文を使用してストアド・プロシージャを作成する方法の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

13.3.2 ストアド・ファンクションの作成

CREATE MODULE 文にストアド・ファンクションの定義を記述すると、ストアド・ファンクションが作成されます。

例 13-3 (13-7) に、CHECK_SALARY_MOD モジュール内の CHECK_SALARY_RANGE_FUNC ストアド・ファンクションの定義を示します。このストアド・ファンクションは、従業員の給与が職務に対応する範囲内にあるかどうかを確認します。

例 13-3 ストアド・ファンクションの作成

```
CREATE MODULE CHECK_SALARY_MOD
LANGUAGE SQL
FUNCTION CHECK_SALARY_RANGE_FUNC (1)
(:JOB_TITLE CHAR(20), :CUR_SALARY INTEGER(2)) (2)
RETURNS INTEGER; (3)
BEGIN (4)
DECLARE :min_sal, :max_sal INTEGER(2);
DECLARE :sal_check INTEGER;
SELECT MINIMUM_SALARY, MAXIMUM_SALARY INTO :min_sal, :max_sal
FROM JOBS
WHERE JOB_TITLE = :job_title;
RETURN (CASE (5)
WHEN (:cur_salary >= :min_sal AND :cur_salary <= :max_sal)
THEN 0
ELSE 1
END );
END; (4)
END MODULE;
```

ストアド・ファンクションを作成する場合は、次の句を使用します (例 13-2 (13-5) の番号に対応します)。

(1) FUNCTION 句

FUNCTION 句には、ストアド・ファンクションの名前を指定します。

関数名は、モジュール定義内および Oracle Rdb データベース内のモジュールに格納された他のルーチンのすべてを通じて一意でなければなりません。また、同じデータベース内で定義された外部ルーチン名とも区別されなければなりません。

(2) パラメータ・リスト

関数のパラメータ・リストには、パラメータ・モード、パラメータ名、SQL データ型を指定します。

- ストアド・ファンクションでは、IN のパラメータ・モード以外は指定できません。
- ストアド・ファンクション定義で使用する個々のパラメータの名前は、関数内で一意でなければなりません。
- LIST OF BYTE VARYING 以外の任意のデータ型を指定できます。宣言では、データ型のかわりにドメイン名を指定できます。たとえば、JOB_TITLE CHAR(20) を JOB_TITLE JOB_TITLE_DOM として宣言できます。

ストアド・ファンクションのパラメータに関しては、次の点に注意してください。

- ストアド・ファンクション内では、ステータス・パラメータ (SQLCODE、SQLSTATE、SQLCA) は宣言できません。ただし、ストアド・ファンクション内で GET DIAGNOSTIC 文を使用すると、SQLCODE または SQLSTATE の値を調べることができます。12.9 項 (12-22) では、GET DIAGNOSTIC 文を使用して SQLCODE と SQLSTATE の値を返し、完了の状態を処理する方法を説明します。SIGNAL 文を使用して例外の状態を処理する方法も示します。
ノン・ストアド・プロシージャからストアド・ファンクションをコールする場合は、ノン・ストアド・モジュール内の任意のプロシージャと同様に、ノン・ストアド・プロシージャ内でステータス・パラメータを宣言する必要があります。
- ストアド・ファンクションは NULL 値をサポートします。したがって、ストアド・ファンクション・パラメータにはインジケータ・パラメータを使用できません。ただし、コールされたストアド・ファンクションが NULL 値を返すと思われる場合は、ストアド・ファンクションをコールするノン・ストアド・プロシージャ内で必ずインジケータ・パラメータを宣言し、確実に NULL 値を処理できるようにしてください。インジケータ・パラメータは、対応するメイン・パラメータに格納された値が NULL かどうかを示します。

(3) RETURNS 句

RETURNS 句 (必須) には、関数で返す値のデータ型を指定します。上の例に示す関数は整数を返します。

(4) BEGIN...END ブロック

この例は、BEGIN...END ブロックで囲まれた複合文を示しています。

ストアド・プロシージャでは、1 つの単純文か 1 つの複合文を使用できます。しかし、

ストアド・ファンクションは値を返し、その値を RETURN 文で取得するので、ストアド・ファンクションでは複合文を使用する必要があります。

(5) RETURN 文

RETURN 文は、ストアド・ファンクションの値を返します。RETURN 文で返す値のデータ型は RETURNS 句で指定したデータ型と互換性がなければなりません。

ストアド・ファンクションでは、SET TRANSACTION 文、COMMIT 文、ROLLBACK 文を除いて複合文で利用できる任意の文を使用できます。ストアド・ファンクションでストアド・プロシージャをコールする場合、プロシージャは前述の文を実行できません。

ストアド・ファンクションのコールについては、13.5 項 (13-12) を参照してください。CREATE MODULE 文を使用してストアド・ファンクションを作成する方法の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

13.3.3 ストアド・ファンクションによる新しい順序番号の生成

アプリケーション・プログラムは、主キーとして使用する一意の順序番号の生成に関する問題にたびたび直面します。1つのソリューションは、ストアド・ファンクションを使用して一意の順序番号付きテーブルを保持し、次の値を発行することです。

AFTER INSERT トリガーの使用など、他のソリューションもありますが、ストアド・ファンクションを使用する主な利点は、最後の値が挿入されるので、データ行への後続の更新が不要なことです。これで、主キーフィールドに関する不要な検証クエリーを節約できます。

このソリューションを実装するには、テーブル NEXT_KEY_TABLE を使用してキー名のリストを保持します。まず、1つのキーを直接テーブルにロードします。それから、ストアド・ファンクションをコールするたびに、NEXT_KEY_TABLE の値をフェッチし、次の値を返します。

次の例に、ドメインとテーブルの定義を示し、最初の値をテーブルにロードする方法を示します。

```
SQL> CREATE DOMAIN KEY_NAME
1>   CHAR(31)
2>   CHECK (VALUE IS NOT NULL)
3>   NOT DEFERRABLE;
SQL> --
SQL> CREATE TABLE NEXT_KEY_TABLE
1>   ( NEXT_KEY_VAL INTEGER NOT NULL NOT DEFERRABLE,
2>     NEXT_KEY_NAME KEY_NAME UNIQUE NOT DEFERRABLE);
SQL> --
SQL> INSERT INTO next_key_table (next_key_name, next_key_val)
1>   VALUES ('EMPLOYEE_ID', 0);
1 row inserted
```

次の例で、ストアド・ファンクションの定義を示します。

```
SQL> CREATE MODULE TOOLS
1>   LANGUAGE SQL
2>   FUNCTION NEXT_KEY (IN :KEY_NAME KEY_NAME)
3>   RETURNS INTEGER;
4>   BEGIN
5>       DECLARE :rc, :new_val INTEGER DEFAULT 0;
6>       DECLARE :key_name_upper key_name DEFAULT UPPER(:KEY_NAME);
7>       DECLARE :invalid_parameter CONSTANT CHAR(5) = '22023';
8>
9>       UPDATE NEXT_KEY_TABLE
10>      SET NEXT_KEY_VAL = NEXT_KEY_VAL + 1
11>          WHERE NEXT_KEY_NAME = :key_name_upper
12>          RETURNING NEXT_KEY_VAL INTO :NEW_VAL;
13>
14>       GET DIAGNOSTICS :rc = ROW_COUNT;
15>       TRACE 'NEXT_KEY is ', COALESCE(:new_val, 'NULL'), ', RC is ', :rc;
16>
17>       IF :rc = 0 THEN
18>           TRACE 'No entry exists for KEY_NAME: ', :key_name_upper;
19>           SIGNAL :invalid_parameter;
20>       ELSE
21>           TRACE 'Returning new value for ', :key_name_upper, :new_val;
22>           RETURN :new_val;
23>       END IF;
24>   END;
25> END MODULE;
```

次の例に示すように、INSERT 文の VALUES 句で関数を指定すると、その関数をコールすることができます。

```
SQL> INSERT INTO employee (employee_id, last_name, birthday)
1>   VALUES (next_key('EMPLOYEE_ID'), 'Smith', DATE'1970-1-1');
```

13.4 ストアド・プロシージャの起動

ストアド・プロシージャは、単純文または複合文から、CALL 文を使って起動できます。ストアド・プロシージャは、別のストアド・プロシージャや他のストアド・ファンクションからもコールできます。

例 13-4 (13-10) は、ノン・ストアド・プロシージャ CALL_NEW_SALARY を含む SQL モジュールのコードの一部分です。これは、例 13-2 (13-5) で定義するストアド・プロシージャをコールします。

例 13-4 ストアド・プロシージャのコール

```
PROCEDURE CALL_NEW_SALARY
(:ID CHAR(5), (1) (13-4)
```

```

:ID_IND SMALLINT, (2) (13-4)
:NEW_SALARY INTEGER (2), (1) (13-4)
:NEW_SALARY_IND SMALLINT, (2) (13-4)
SQLCODE); (3) (13-4)
CALL NEW_SALARY_PROC (4) (13-4)
(:ID INDICATOR :ID_IND, :NEW_SALARY INDICATOR :NEW_SALARY_IND); (5) (13-5)

```

次の番号は、例 13-4 (13-10) に対応しています。

- (1) CALL_NEW_SALARY プロシージャのメイン・パラメータは、:ID と :NEW_SALARY です。
- (2) インジケータ・パラメータは、:ID_IND と :NEW_SALARY_IND です。
 ストアド・プロシージャのパラメータには NULL 値が許可されるので、各メイン・パラメータにはインジケータ・パラメータを指定します。インジケータ・パラメータを指定しないと、ストアド・プロシージャがメイン・パラメータに NULL 値を返した場合、SQL はエラーを返し、プロシージャは失敗します。ストアド・プロシージャのパラメータが NULL 値を許可する問題は、インジケータ・パラメータで解決できますが、ノン・ストアド・プロシージャのパラメータでは解決できません。
- (3) ノン・ストアド・プロシージャには、ステータス・パラメータを使用する必要があります。
 ストアド・プロシージャとノン・ストアド・プロシージャでは、ステータス・パラメータの宣言が異なります。ノン・ストアド・プロシージャでは、ステータス・パラメータを宣言する必要があります。ストアド・プロシージャでは、ステータス・パラメータは不要であり、使用できません。
- (4) CALL 文は、ストアド・プロシージャを起動します。
 例 13-2 (13-5) で定義した NEW_SALARY_PROC ストアド・プロシージャを、CALL 文で起動します。
- (5) CALL 文は、:ID と :NEW_SALARY 変数を、コールされたストアド・プロシージャに引数として渡します。
 CALL 文で使用する 2 つの変数のデータ型は、コールする NEW_SALARY_PROC プロシージャで使用するデータ型と一致する必要があります。
 単純文の CALL 文は、リテラル、パラメータ値（動の実行用のパラメータ・マーカー）、変数などのリストをコールされたストアド・プロシージャに渡すことができ、引数を渡さないことも可能です。複合文の CALL 文は、dbkey や集計関数以外の値式を、コールしたストアド・プロシージャに渡すことができ、引数を渡さないことも可能です。

CALL 文や、引渡し可能な引数の型の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

13.5 ストアド・ファンクションの起動

ストアド・ファンクションの起動では、SQL 文の値式として関数名を使用します。ストアド・ファンクションは、トリガーを除き、値式が許可される任意の位置から起動できます。

例 13-5 (13-12) は、IF 文を使って、例 13-3 (13-7) で定義されたストアド・ファンクションをコールする複合文です。IF 文は、ストアド・ファンクション CHECK_SALARY_RANGE をコールし、2 つのパラメータ :salrec.JOB と :salrec.SALARY を関数に渡します。関数の結果が 1 の場合、IF 文は従業員の姓をテーブル SALARY_ADJUST に挿入します。

例 13-5 ストアド・ファンクションの起動

```
BEGIN
  DECLARE :lname char(14);
  FOR :salrec
    AS EACH ROW OF TABLE CURSOR SAL_CURSOR FOR
      SELECT LAST_NAME, JOB, SALARY FROM CURRENT_INFO
  DO
    SET :lname = :salrec.LAST_NAME;
    -- Call the stored function.
    IF CHECK_SALARY_RANGE(:salrec.JOB, :salrec.SALARY) = 1
      THEN
        INSERT INTO SALARY_ADJUST VALUES (:lname);
      END IF;
    END FOR;
  END;
```

13.6 ストアド・ルーチンの削除

DROP MODULE 文は、Oracle Rdb データベースから、ストアド・モジュールとそのルーチンを削除します。削除には、モジュールに対する DROP 権限が必要です。

例 13-6 (13-12) は、DATA_UPDATE モジュールを削除する方法を示しています。

例 13-6 ストアド・モジュールの削除

```
DROP MODULE DATA_UPDATE;
```

モジュール全体ではなく、ストアド・プロシージャのみを削除するには、次の例のように、DROP PROCEDURE 文を使用します。

```
DROP PROCEDURE NEW_SALARY_PROC;
```

モジュール全体ではなく、ストアド・ファンクションのみを削除するには、次の例のように、DROP FUNCTION 文を使用します。

```
DROP FUNCTION NEW_FUNC;
```

RESTRICT キーワード（デフォルト）を使用すると、そのルーチンを参照しているストアド・ルーチンがデータベース内に存在する場合は、ストアド・ルーチンを削除できなくなります。

CASCADE キーワードを使用してストアド・ルーチンを削除すると、Oracle Rdb は DROP 文を実行し、削除されたルーチンを参照しているストアド・ルーチンがあれば、それを無効にします。

ストアド・ルーチンに必要な権限の詳細は、『Oracle Rdb7 Guide to Database Design and Definition』を参照してください。DROP MODULE 文、DROP PROCEDURE 文、DROP FUNCTION 文の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

13.7 ストアド・ルーチンの依存性の追跡

実行時にストアド・ルーチンのコンパイルを成功させるために、Oracle Rdb は、ストアド・ルーチンが依存するスキーマ・オブジェクトの追跡を行います。たとえば、テーブルや列など、ストアド・ルーチンが依存するオブジェクトがなんらかの理由で削除または変更されている場合、Oracle Rdb は、ストアド・ルーチンをコンパイルできなくなることがあります。ストアド・ルーチンが依存しているオブジェクトを認識するために、Oracle Rdb は、**依存性の追跡**という方法を使用します。

CREATE MODULE 文を入力すると、Oracle Rdb は、モジュールと各ルーチンをチェックし、列、制約、ドメイン、関数、テーブル、ビューなどを参照していないかどうかを確認します。このように、参照先となっているオブジェクトは**参照オブジェクト**と呼ばれ、1つ以上の参照オブジェクトが見つかったら、Oracle Rdb は RDB\$INTERRELATIONS システム・テーブルに依存情報を記録します。

正しく実行するために、参照オブジェクトに依存するストアド・ルーチンを、**依存オブジェクト**と呼びます。

Oracle Rdb は、表 13-1 (13-13) に一覧されているオブジェクトについて、メタデータの依存関係情報と各オブジェクトの依存関係タイプを格納します。

表 13-1 依存性の追跡テーブル

RDB\$INTERRELATIONS に格納される情報						
参照オブジェクト	オブジェクト名	サブオブジェクト名	エンティティ名 1	エンティティ名 2	使用方法	制約名
列	テーブル	列	モジュール	プロシージャ または関数	P/F	
制約			モジュール	プロシージャ または関数	P/F	制約
制約			モジュール		DE	制約

表 13-1 依存性の追跡テーブル（続き）

RDB\$INTERRELATIONS に格納される情報						
参照オブジェクト	オブジェクト名	サブオブジェクト名	エンティティ名 1	エンティティ名 2	使用方法	制約名
ドメイン		ドメイン	モジュール	プロシージャ または関数	P/F	
関数（外部）		関数	モジュール	プロシージャ または関数	P/F	
関数（ストアド）	モジュール	関数	モジュール	プロシージャ または関数	P/F	
プロシージャ （外部）		関数	モジュール	プロシージャ または関数	P/F	
プロシージャ （ストアド）	モジュール	関数	モジュール	プロシージャ または関数	P/F	
テーブル	テーブル		モジュール	プロシージャ または関数	P/F、 LS	
テーブル	テーブル		モジュール		DR	
ビュー	ビュー		モジュール	プロシージャ または関数	P/F、 LS	
ビュー	ビュー		モジュール		DR	

使用方法の列の依存関係タイプ：

- DE - デフォルトの評価
- DR - デフォルトの予約
- LS - 言語セマンティクス
- P/F - プロシージャまたは関数

Oracle Rdb は、ドメインがプロシージャ・ブロック内にある場合、依存関係情報を RDB\$INTERRELATIONS に格納します。これに対して、ストアド・ルーチンのパラメータ・リスト内でドメインが定義されている場合、Oracle Rdb は、依存関係を、RDB\$INTERRELATIONS ではなく、RDB\$PARAMETERS に格納します。

システム・テーブル RDB\$INTERRELATIONS に格納されている依存関係情報の中でも、依存関係タイプは重要な要素です。Oracle Rdb は、参照オブジェクト名によって、メタデータの依存関係を一意に認識できますが、依存関係タイプを使用することにより、参照オブジェクトが追加または削除された場合に、依存オブジェクトを無効化することができます。

ルーチンが無効化されると、無効化前の状態のように、すべての参照オブジェクトがリストアされるまでは、ルーチンは起動できなくなります。

次の項では、ストアド・ルーチンの4種類の依存関係タイプである、プロシージャ、言語セマンティクス、デフォルトの予約、デフォルトの評価について説明します。

13.7.1 プロシージャ依存関係タイプ

複合文のプロシージャ・ブロックで参照オブジェクトを指定すると、オブジェクトと、そのオブジェクトを参照するストアド・ルーチンの間に、**プロシージャ依存関係**または**関数依存関係**が生成されます。次のストアド・プロシージャでは、CANDIDATES テーブルと、指定した各列 (LAST_NAME、FIRST_NAME、MIDDLE_INITIAL、CANDIDATE_STATUS) が、プロシージャ依存関係を生成します。

```
PROCEDURE SIMPLE_P
BEGIN
    INSERT INTO CANDIDATES
    (LAST_NAME,FIRST_NAME,MIDDLE_INITIAL, CANDIDATE_STATUS)
    VALUES
    ('test_lname','test_fname','t','test_status');
END;
```

例 13-7 (13-15) のように RDB\$INTERRELATIONS をクエリーすると、前の例で INSERT 文が参照している CANDIDATES テーブルの5つのオブジェクトについて、Oracle Rdb は、プロシージャ依存関係を格納していることがわかります。

例 13-7 プロシージャ依存関係タイプの確認

```
SQL> SELECT * FROM RDB$INTERRELATIONS WHERE RDB$OBJECT_NAME = 'CANDIDATES';
```

RDB\$OBJECT_NAME	RDB\$SUBOBJECT_NAME	
RDB\$ENTITY_NAME1	RDB\$ENTITY_NAME2	
RDB\$USAGE	RDB\$FLAGS	
RDB\$CONSTRAINT_NAME	RDB\$SECURITY_CLASS	
CANDIDATES (1)		
DEPENDENCY_LIST (3)	SIMPLE_P (4)	
Procedure (5)		5
	NULL	
CANDIDATES (1)	LAST_NAME (2)	
DEPENDENCY_LIST (3)	SIMPLE_P (4)	
Procedure (5)		5
	NULL	
CANDIDATES (1)	FIRST_NAME (2)	
DEPENDENCY_LIST (3)	SIMPLE_P (4)	
Procedure (5)		5
	NULL	
CANDIDATES (1)	MIDDLE_INITIAL (2)	
DEPENDENCY_LIST (3)	SIMPLE_P (4)	

Procedure (5)	5
	NULL
CANDIDATES (1)	CANDIDATE_STATUS (2)
DEPENDENCY_LIST (3)	SIMPLE_P (4)
Procedure (5)	5
	NULL

次の番号は、例 13-7 (13-15) に対応しています。

- (1) オブジェクト名：テーブル名
- (2) サブオブジェクト名：列名
- (3) エンティティ名 1：モジュール名
- (4) エンティティ名 2：プロシージャ名
- (5) 使用方法：依存関係タイプ
ストアド関数が CANDIDATES テーブルの列を参照している場合、このフィールドは、「プロシージャ」ではなく「関数」として表示されます。

13.7.2 言語セマンティクス依存関係タイプ

ストアド・ルーチンのプロシージャ・ブロックが、テーブルやビューを参照するが、列の参照は暗黙的である場合、ストアド・ルーチンと参照オブジェクトの間に、**言語セマンティクス依存関係**が作成されます。

自然結合と SELECT * 操作でも、言語セマンティック依存関係が作成されます。

次の例では、INSERT 文が CANDIDATES テーブルの列を明示的に参照していないため、Oracle Rdb は、言語セマンティクス依存関係を設定します。

```
PROCEDURE LANG_SEMANTICS_P
BEGIN
  INSERT INTO CANDIDATES VALUES
    ('test_lname','test_fname','t','test_status');
END;
```

INSERT 文には列リストが含まれていないので、SQL は、列の順序と名前を、現在のテーブル定義から取得します。列リストの省略により、Oracle Rdb は、CANDIDATES テーブルと LANG_SEMANTICS_P ストアド・プロシージャの間に、言語セマンティクス依存関係を設定します。

DELETE 文は、列を暗黙的に参照しますが、明示的には参照しません。

LANG_SEMANTICS_P ストアド・プロシージャは、CANDIDATES テーブルに明示的に依存し、すべての列に暗黙的に依存しているので、CANDIDATES テーブルのすべての列は、参照オブジェクトとして認識されます。したがって、CASCADE キーワードを使ってテーブルを削除したり、テーブルに列を追加すると、ストアド・プロシージャは無効になります。

言語セマンティクス依存関係によって無効化されたストアド・ルーチンを再作成する方法については、13.9.2 項 (13-22) を参照してください。

13.7.3 トランザクション依存関係タイプ

Oracle Rdb は、次の 2 つのトランザクション依存関係タイプを認識します。

- デフォルトの予約依存関係
DECLARE TRANSACTION RESERVING 文または SET TRANSACTION RESERVING 文でテーブルを指定すると、参照テーブルには、**デフォルトの予約依存関係**が生成されます。
- デフォルトの評価依存関係
DECLARE TRANSACTION EVALUATING 文または SET TRANSACTION EVALUATING 文で制約を指定すると、参照先の制約には、**デフォルトの評価依存関係**が生成されます。

次の例では、Oracle Rdb は、JOBS テーブルにデフォルトの予約依存関係、JOB_CODE_REQUIRED 制約にはデフォルトの評価依存関係を設定しています。

```
CREATE MODULE DEPENDENCY_LIST LANGUAGE SQL
-- Create default reserving and default evaluating dependencies.
DECLARE TRANSACTION READ WRITE
    RESERVING JOBS
    FOR SHARED READ EVALUATING JOB_CODE_REQUIRED AT VERB TIME
```

例 13-8 (13-17) の RDB\$INTERRELATIONS をクエリーすると、JOBS テーブルにはデフォルトの予約依存関係、DECLARE TRANSACTION 文の参照先である JOB_CODE_REQUIRED 制約にはデフォルトの評価依存関係を格納していることがわかります。

例 13-8 トランザクション依存関係タイプの確認

```
SQL> SELECT * FROM RDB$INTERRELATIONS WHERE RDB$OBJECT_NAME = 'JOBS';
.
.
.
DEPENDENCY_LIST (2)
  Default Txn Evaluating (3)                1
    JOB_CODE_REQUIRED (4)                NULL
JOBS (1)
  DEPENDENCY_LIST (2)
    Default Txn Reserving (3)                1
                                         NULL
```

次の番号は、例 13-8 (13-17) に対応しています。

- (1) オブジェクト名：テーブル名
- (2) エンティティ名 1：モジュール名

(3) 使用方法：依存関係タイプ

(4) 制約名

プロシージャ、関数、言語セマンティクスなどの依存関係は、プロシージャ・ベースの追跡方法を採用していますが、トランザクション依存関係は、モジュール・レベルでストアド・ルーチンに影響を与えます。例 13-8 (13-17) では、Oracle Rdb が、両方のトランザクション依存関係タイプのみについて、モジュール名を格納していることがわかります。DECLARE TRANSACTION 文は、モジュール内の一部のルーチンではなく、すべてのルーチンにリンクしているため、プロシージャ名や関数名は格納されていません。これは、テーブルや制約を削除または変更したときに影響します。

CASCADE キーワードを指定してテーブルを削除し、ストアド・モジュールの DECLARE TRANSACTION 文がそのテーブルを参照している場合、Oracle Rdb は、テーブルを削除し、モジュール内のすべてのルーチンを無効化します。したがって、そのテーブルに依存するルーチンを実行すると、Oracle Rdb はエラーを返し、ルーチンの実行は失敗します。トランザクション依存関係タイプはモジュール単位であるため、モジュールに含まれるストアド・ルーチンを起動すると、モジュール内の各ルーチンは失敗します。

13.8 ストアド・ルーチンの無効化

1 つ以上のスキーマ・オブジェクトが変更または削除されると、そのオブジェクトに依存しているストアド・ルーチンは無効化されます。Oracle Rdb は、その無効になったステータスをシステム・テーブルに格納します。無効化は、ストアド・ルーチンの基盤のスキーマ・オブジェクトを再作成しないと、ストアド・ルーチンの実行が失敗する可能性があることを示しています。

参考: Oracle_Rdb ヘルプの Oracle Rdb システム・テーブルの説明を参照してください。Oracle Rdb が無効化で設定するフラグについては、RDB\$ROUTINES および RDB\$INTERRELATIONS システム・テーブルの説明を参照してください。

表 13-2 (13-18) は、ストアド・ルーチンの無効化の原因となるデータ定義文です。

表 13-2 ストアド・ルーチンを無効化する文

オブジェクト タイプ	SQL 文	実行の失敗	ルーチン の無効化	依存関係 タイプ
列	ALTER TABLE DROP COLUMN	[あり]	なし	PF または LS
	ALTER TABLE ADD COLUMN	なし	[あり]	LS

表 13-2 ストアド・ルーチンを無効化する文 (続き)

オブジェクト タイプ	SQL 文	実行の失敗	ルーチン の無効化	依存関係 タイプ
制約	ALTER TABLE ADD COLUMN	なし	なし	P/F
	ALTER TABLE DROP CONSTRAINT	[あり]	なし	P/F
ドメイン	ALTER TABLE ADD CONSTRAINT	なし	なし	P/F または DE
	ALTER DOMAIN (パラ メータ・リスト内)	[あり]	なし	適用なし ^{*1}
	ALTER DOMAIN (プロ シージャ・ブロック内)	なし	なし	P/F ^{*2}
関数	DROP DOMAIN	[あり]	なし	P/F
	DROP FUNCTION CASCADE	なし	[あり]	P/F
	DROP FUNCTION RESTRICT	[あり]	なし	P/F
モジュール	DROP MODULE CASCADE	なし	[あり]	P/F
	DROP MODULE RESTRICT	[あり]	なし	P/F
プロシー ジャ	DROP PROCEDURE CASCADE	なし	[あり]	P/F
	DROP PROCEDURE RESTRICT	[あり]	なし	P/F
テーブル	DROP TABLE CASCADE	なし	[あり]	PF または LS
	DROP TABLE RESTRICT	[あり]	なし	P/F、LS または DR
ビュー	DROP VIEW CASCADE	なし	[あり]	PF または LS
	DROP VIEW RESTRICT	[あり]	なし	P/F、LS または DR

^{*1} Oracle Rdb は、ドメイン・パラメータ・リストの依存関係を、RDB\$INTERRELATIONS ではなく、RDB\$PARAMETERS システム・テーブルに格納します。

^{*2} Oracle Rdb は、ドメインがプロシージャ・ブロック内にある場合、プロシージャの依存関係を RDB\$INTERRELATIONS に格納します。

依存関係タイプへのキー：

- DE--- デフォルトの評価
- DR--- デフォルトの予約
- LS--- 言語セマンティクス
- P/F--- プロシージャまたは関数

RESTRICT キーワードを文で指定すると、ストアド・ルーチンに依存するオブジェクトを削除できなくなるので、制限付きの DROP 文は、無効化には影響しません。DROP のカスケード操作は、実行は成功しますが、無効化の原因になります。

対話型 SQL で SHOW PROCEDURE 文や SHOW FUNCTION 文を使用すると、ストアド・プロシージャまたは関数が有効か無効かを確認できます。次の例では、P1 ストアド・プロシージャが依存しているテーブルを削除したため、Oracle Rdb は、P1 を無効化します。

```
SQL> SHOW PROCEDURE P1
Procedure name is: P1
Status is INVALID
      Can be revalidated
```

上の例では、Oracle Rdb は、プロシージャは再有効化が可能であることを示しています。ルーチンの再有効化の詳細は、13.9.1 項 (13-21) を参照してください。

ストアド・ルーチンで、"Status is INVALID" というメッセージが表示されなければ、そのルーチンは有効です。

13.9 ストアド・ルーチンの再有効化

無効化されたルーチンは、再有効化できます。ただし、必要な操作は、次のように、無効化されたストアド・ルーチンの依存関係タイプによって異なります。

- 無効化されたストアド・ルーチンの依存関係タイプが次の場合、再有効化が可能です。
 - プロシージャ
 - デフォルトの評価
 - デフォルトの予約

13.9.1 項 (13-21) では、ルーチンを再有効化する方法について説明しています。

- 言語セマンティクスの依存関係が原因で無効化されたストアド・ルーチンは、再有効化できません。このようなルーチンは、13.9.2 項 (13-22) で説明する手順に従って再作成してください。

13.9.1 無効化されたストアド・ルーチンの再有効化

無効化されたストアド・ルーチンの依存関係タイプが、プロシージャ、デフォルトの評価、デフォルトの予約のいずれかであれば、ほとんどの場合、再有効化が可能です。

例 13-9 (13-21) の NEW_SALARY_PROC ストアド・ルーチンは、依存するオブジェクト SALARY_HISTORY テーブルが、ストアド・ルーチンにプロシージャの依存関係を作成するので無効化されましたが、再有効化が可能です。

例 13-9 依存関係タイプがプロシージャのストアド・モジュール定義

```
CREATE MODULE NEW_SALARY_PROC LANGUAGE SQL
  PROCEDURE NEW_SALARY_PROC
    (:ID CHAR(5),
     :NEW_SALARY INTEGER(2));
  BEGIN
    UPDATE SALARY_HISTORY
      SET SALARY_END = CURRENT_TIMESTAMP
        WHERE EMPLOYEE_ID = :ID;
    INSERT INTO SALARY_HISTORY (EMPLOYEE_ID, SALARY_AMOUNT,
                                SALARY_START, SALARY_END)
      VALUES
        (:ID, :NEW_SALARY, CURRENT_TIMESTAMP, NULL);
  END;
END MODULE;
```

CASCADE オプションを使って SALARY_HISTORY テーブルを削除すると、Oracle Rdb は、NEW_SALARY_PROC を無効化します。Oracle Rdb がストアド・プロシージャを無効化したのは、このストアド・プロシージャは SALARY_HISTORY テーブルにプロシージャの依存関係タイプを持ち、これに基づいてストアド・プロシージャは正しく実行されるからです。

ストアド・ルーチンを再有効化するには、次の手順に従います。

- (1) ルーチンが依存しているデータベース・オブジェクトを再作成します。たとえば、NEW_SALARY_PROC プロシージャを有効化するには、SALARY_HISTORY テーブルを再作成します。
- (2) 論理名 RDMS\$VALIDATE_ROUTINE または構成パラメータ RDB_VALIDATE_ROUTINE を 1 に定義すると、無効化されたルーチンは有効としてマークされます。読取り / 書き込みトランザクションでプロセスがストアド・ルーチンをコールすると、Oracle Rdb は、無効なルーチンを有効としてマークします。
- (3) 対話型 SQL を起動し、データベースに接続します。
このインタフェースでは、プリコンパイルした SQL と SQL モジュール言語を使用できませんが、SET NOEXECUTE 文をサポートしません。
- (4) 読取り / 書き込みトランザクションを開始します。

- (5) トランザクション文をストアド・ルーチン内で使用することでエラーが発生しないように、また、誤ってデータを変更しないように、次の SQL 文を使用してください。

```
SQL> SET NOEXECUTE
```

- (6) ストアド・ルーチンを起動します。
たとえば、次の文は、ストアド・プロシージャ NEW_SALARY_PROC をコールします。

```
SQL> CALL NEW_SALARY_PROC('00196', 2);
```

SET NOEXECUTE 文を使用したので、ルーチンの文は実行されず、データもデータベースに格納されません。

他にも無効なストアド・ルーチンがある場合、依存するデータベース要素を再作成すると、各ルーチンを起動することによって、ストアド・ルーチンを有効化できます。

- (7) SET EXECUTE 文を発行します。

- (8) COMMIT 文を発行します。

13.9.2 言語セマンティクスの依存関係で無効化されたストアド・ルーチンの再作成

言語セマンティクスの依存関係でストアド・ルーチンが無効化された場合は、13.9.1 項 (13-21) の方法では再有効化できません。有効化するには、次の手順で示すように、親モジュールと依存するスキーマ・オブジェクトを再作成する必要があります。

1. RMU Extract コマンドで Item=Module および Output 修飾子を指定し、ストアド・モジュール定義を抽出してファイルに書き込みます。
抽出したモジュール定義を変更する方が、1 行ずつ定義を再入力するよりも簡単です。
RMU Extract コマンドの詳細は、『Oracle RMU Reference Manual』を参照してください。
2. DROP MODULE 文を使用して、ストアド・モジュールを削除します。
3. 依存するスキーマ・オブジェクトと、言語セマンティクスの依存関係による無効化の原因となったオブジェクトを再作成します。
4. CREATE MODULE 文を使用して、ストアド・モジュールを再作成します。
抽出したファイル（手順 1 で作成）を変更し、CREATE MODULE 文でモジュール定義をデータベースに格納します。

Oracle Rdb が、言語セマンティクスによって無効化されたルーチンの有効化を許可しない理由は、無効化の原因となっているスキーマ・オブジェクトを再作成したとしても、Oracle Rdb は、ストアド・ルーチンで使用できるように、そのオブジェクトが作成されているかどうかを判断できないためです。たとえば、同じ列名でテーブルを再作成できますが、列の順序を変えた場合を考えてみましょう。テーブルを再作成すれば、Oracle Rdb によるストアド・ルーチンのコンパイルと実行は可能です。ただし、ルーチンは、予想とは異なる動作をする可能性があります。Oracle Rdb は、テーブルの再作成で、元の定義が使用されていると

いう保証はないので、言語セマンティクスの依存関係によって無効化されたストアド・ルーチンは、有効化できなくなっています。

外部ルーチンの使用方法

この章では、外部ルーチンの作成や使用方法について説明します。外部ルーチンとは、外部プロシージャまたは外部関数であり、3GL 言語で記述され、共有可能イメージや共有オブジェクトにリンクして、データベース内に登録します。この章の各項では、次の内容を説明します。

- 外部ルーチンの概要
- 外部ルーチンを使用するアプリケーションの開発
- データベースでの外部ルーチン定義の作成
- データベースでの外部ルーチン定義の変更と削除
- 外部ルーチンの記述と、ルーチンを起動するプログラムの記述
- 外部ルーチンに使用する OpenVMS の共有可能イメージの作成
- 外部ルーチンに使用する Compaq Tru64 UNIX の共有オブジェクトの作成
- SQL 文での外部ルーチンの起動
- ルーチンの実行条件の指定
- ルーチンのアクティブ化と非アクティブ化の概念
- 外部ルーチンでのパラメータの宣言および受渡しと、戻り値の宣言
- 特定のホスト言語での外部ルーチンのコーディング
- 通知ルーチンを使用した初期化とクリーンアップ操作
- 外部ルーチンで発生する例外の処理
- 外部ルーチンでの制約事項
- 外部ルーチンのトラブルシューティング
- 移植可能で効率的な外部ルーチンの記述

14.1 外部ルーチンの概要

外部ルーチンは、プログラミング言語のプロシージャ機能を提供します。外部ルーチンは、3GL ルーチンを使って実行できるタスクの領域にまで、SQL 文の機能を拡張します。外部ルーチンは、平方根の計算、データベースへの接続、データベースでのデータ操作などのタスクを実行できます。

SQL は、次のような外部ルーチンを実装します。

- 外部関数

外部関数とは、3GL プログラム（C や COBOL などの言語で記述）であり、SQL 文で関数名を値として指定して起動します。たとえば、次の INSERT 文は OpenVMS 実行時ライブラリ（Runtime Library : RTL）ルーチン MTH\$SQRT を起動し、SQUARE_ROOTS テーブルの列に平方根の値を格納します。

```
INSERT INTO SQUARE_ROOTS VALUES (P_NUM, SQRT(CAST(P_NUM AS REAL)));
```

組込みまたはストアド・ファンクションと同様に、外部関数は、入力引数リストを受け取り、必ず 1 つの値を返します。この値は、SQL 文の評価で SQL が使用します。ただし、組込みまたはストアド・ファンクションと異なるのは、外部関数のコードは、SQL や Oracle Rdb の外部にあるファイルに格納されている点です。

外部関数の名前をデータベースに登録するには、SQL CREATE FUNCTION 文を使って関数の定義を指定します。

- 外部プロシージャ

外部プロシージャとは、SQL の CALL 文で起動する 3GL プログラム（C や COBOL などの言語で記述）です。ストアド・プロシージャと同様に、外部プロシージャは、入力、出力および入出力の引数リストを受け取りますが、戻り値はありません。ただし、ストアド・プロシージャと異なるのは、外部プロシージャのコードは、SQL や Oracle Rdb の外部にあるファイルに格納されている点です。

次の例では、CALL 文で、外部プロシージャ ADD_SOUNDEX_NAME を起動しています。

```
CALL ADD_SOUNDEX_NAME (:error);
```

外部プロシージャの名前をデータベースに登録するには、SQL CREATE PROCEDURE 文を使ってルーチンの定義を指定します。

外部ルーチンは、実行時ライブラリ、オペレーティング・システム・サービスまたはその他の既存のライブラリにある既存のルーチンを使用して記述することができ、新しい外部ルーチンを記述することも可能です。たとえば、英語の発音が似ている名前をデータベース内で検索する外部関数を作成する例を考えてみましょう。Barns、Barnes、Barnse は同じになります。次の文は、ユーザー定義の外部関数 SOUNDEX を起動します。この外部関数は、名前を文字列にエンコードして、発音が似ている名前がないかデータベースを検索します。

```
SELECT LAST_NAME, SOUNDEX(LAST_NAME) FROM EMPLOYEES;
```

14.2 外部ルーチンの開発

外部ルーチンを開発し、Oracle Rdb データベースで使用するには、次のような手順に従ってください。

1. 外部ルーチンが存在しない場合は、3GL 言語で作成します。
2. ルーチンをコンパイルします。
Oracle Rdb データベースへのコールがルーチンに含まれない場合は、ソース・コードをホスト言語のコンパイラでコンパイルします。Oracle Rdb データベースへのコールがルーチンに含まれる場合は、ソース・コードを SQL プリコンパイラでコンパイルします。ルーチンが SQL モジュール・プロシージャを参照し、それが Oracle Rdb データベースをコールする場合は、ルーチンのソース・コードをホスト言語のコンパイラでコンパイルし、SQL モジュール・プロシージャを SQL モジュール・プロセッサでコンパイルします。
3. 共有可能イメージと共有オブジェクトを作成します。
OpenVMS で外部ルーチンを使用するには、既存のルーチンが既存の共有可能イメージ内に存在している場合を除いて、共有可能イメージを作成する必要があります。◆
Compaq Tru64 UNIX で外部ルーチンを使用するには、既存のルーチンが既存の共有オブジェクト内に存在している場合を除いて、共有オブジェクトを作成する必要があります。◆
4. ルーチンをテストします。
外部ルーチンのコーディング、コンパイル、リンクが完了したら、ルーチンを個別にテストし、動作を確認します。通常、このテストには、テスト・プログラムを記述する必要があります。
5. データベースにルーチンの定義を作成します。
CREATE FUNCTION または CREATE PROCEDURE 文を使って、外部ルーチンの定義を Oracle Rdb データベースに作成します。外部ルーチンの定義には、外部ルーチンが使用するパラメータ、外部ルーチンの名前、ルーチンを格納している実行可能イメージやオブジェクトのロケーション、ルーチンのコーディングを行った言語など、ルーチンに関する情報が格納されています。
6. SQL 文でルーチンを起動します。
外部関数は、SQL 文で許可されていれば、任意の式で起動できます。外部プロシージャは、複合文の中で SQL CALL 文を使って起動できます。
外部ルーチンの代表的な起動例は、14.8 項 (14-30) を参照してください。

一般的に、外部ルーチンのアクティブなスコープは、外部ルーチンを起動したデータベース上のアクティブな接続のスコープです。起動データベースから切断するときに、外部ルーチンを物理的に非アクティブ化（コードを削除）する場合もあります。

外部ルーチンの操作は、ルーチンの起動時に実行されているデータベース・トランザクションのコンテキストの外側で実行されている点に注意してください。外部ルーチンが、ファイルへの書込みやプロセスの生成などの操作を実行した場合、ロールバックが発生しても、外



部ルーチンが実行した操作は、トランザクションのロールバックの一部として取り消されることはありません。

14.3 外部ルーチン定義の作成

SQL 文から外部ルーチンを起動するには、Oracle Rdb データベースに、外部ルーチンの名前を登録し、外部ルーチンの定義を作成する必要があります。これは、外部ルーチンの作成で必要になる最初の手順ではありませんが、どのような外部ルーチン情報が Oracle Rdb データベースに格納されるかを理解できます。

14.3.1 項 (14-4) では、外部関数定義を作成する方法を説明します。14.3.2 項 (14-6) では、外部プロシージャの定義を作成する方法を説明します。外部関数と外部プロシージャの作成では、同じ句を多く使用します。

14.3.1 外部関数の定義の作成

外部関数の定義は、テーブルやビューなど他のスキーマ・オブジェクトと同様に、Oracle Rdb データベースに格納されます。外部関数の定義を作成するには、例 14-1 (14-4) で示すように、CREATE FUNCTION 文を使用します。

例 14-1 CREATE FUNCTION 文を使用した外部関数の定義

```
-- SQRT external function.  
CREATE FUNCTION SQRT (1) (14-4) (IN :PARAM1 REAL) (2) (14-4)  
  RETURNS REAL; (3) (14-5)  
  EXTERNAL NAME MTH$SQRT (4) (14-5)  
    LOCATION 'SYS$SHARE:MTHRTL.EXE' (5) (14-5)  
    LANGUAGE GENERAL (6) (14-5)  
    GENERAL PARAMETER STYLE (7) (14-5)  
  NOT VARIANT (8) (14-5)  
  COMMENT IS 'Square Root of an F-floating value'; (9) (14-6)
```

番号は例 14-1 (14-4) に対応しています。

(1) CREATE FUNCTION SQRT

Oracle Rdb データベースに含まれるオブジェクトとして外部関数を作成し、外部関数の定義の名前を指定します。
定義名は、外部ルーチンの定義および格納されているルーチン名のすべての中で、一意でなければなりません。

(2) IN :PARAM1 REAL

外部関数で使用する IN パラメータについて、パラメータ名と SQL データ型 (REAL) を指定します。パラメータ名はオプションですが、使用する場合は、先頭にコロン (:) を指定してください。外部関数で使えるのは、IN パラメータのみです。
SQL データ型は、LIST OF BYTE VARYING 以外はすべて指定できます。また、データ型の指定には、ドメイン名を使用することもできます。

さらに、各パラメータの引渡し方法も指定できます。引渡し方法を指定しないと、SQL は、REFERENCE でパラメータを渡します。パラメータの引渡し方法は、パラメータのデータ型と外部関数の言語によって異なります。パラメータとサポートされる引渡し方法の詳細は、14.11 項 (14-36) と 14.12 項 (14-38) を参照してください。

(3) RETURNS REAL

外部関数の戻り値の SQL データ型を指定します。

SQL データ型は、LIST OF BYTE VARYING 以外はすべて指定できます。また、データ型の指定には、ドメイン名を使用することもできます。

戻り値の引渡し方法も指定できます。引渡し方法を指定しないと、SQL は、数値データ型の戻り値は VALUE、文字データ型の戻り値は REFERENCE で渡します。引渡し方法は、戻り値のデータ型と外部関数の言語によって異なります。サポートされる引渡し方法の詳細は、14.11 項 (14-36) と 14.12 項 (14-38) を参照してください。

(4) EXTERNAL NAME MTH\$SQRT

外部ルーチン・プログラムで宣言する外部関数の名前を指定します。

(5) LOCATION 'SYS\$SHARE:MTHRTL.EXE'

外部関数のファイル指定を指定します。

OpenVMS では、外部関数のイメージの位置を指定する方法として、ファイル指定または論理名を使用できます。◆

Compaq Tru64 UNIX では、外部関数の共有オブジェクトの位置を指定する方法として、絶対パス名または相対パス名を使用できます。◆

ロケーション句を指定しない場合や DEFAULT LOCATION 句を指定する場合、SQL は RDB\$ROUTINES をイメージ位置として使用します。

(6) LANGUAGE GENERAL

外部関数を記述するホスト言語を指定します。

指定できるキーワードは、ADA、C、COBOL、FORTRAN、PASCAL、GENERAL です。GENERAL キーワードを指定すると、外部関数を使用しているデータ型とパラメータの引渡し方法が Oracle Rdb でサポートされていれば、任意の言語で記述された外部関数をコールできます。関数を記述した言語がわからない場合や、SQL でサポートされない言語で記述されている場合には、このキーワードを使用してください。

(7) GENERAL PARAMETER STYLE

外部関数に引数を渡し、戻り値を受け取る方法を指定します。現在の SQL では、OpenVMS の呼出し規則に最も近いスタイル GENERAL を使用するように限定されています。Oracle Rdb の今後のバージョンでは、SQL 標準に準拠して、他のパラメータ・スタイルを実装する予定です。

(8) NOT VARIANT

1 つのクエリーのスコープ内で、毎回関数を起動するかどうかを指定します。NOT VARIANT 句を使用すると、1 つのクエリーにおいて、対応する関数式の計算を 1 回実行します (1 回のみとは限りません)。計算結果は、対応する関数式のすべてのオカレン

OpenVMS

Compaq Tru64 UNIX

スで使用されます。外部関数の評価を 1 回に限定するかどうかの最終的な決定は、Oracle Rdb クエリー・オブティマイザが行います。
VARIANT 句を使用すると、関数が起動されるたびに、関数を強制的に評価します。

(9) COMMENT IS 'Square Root of an F-floating value'

外部関数に関するコメントを追加できます。

CREATE FUNCTION 文、構文、引数の詳細は、『Oracle Rdb7 SQL Reference Manual』の Create Routine 文を参照してください。

14.3.2 外部プロシージャ定義の作成

外部プロシージャの定義は、テーブルやビューなど他のスキーマ・オブジェクトと同様に、Oracle Rdb データベースに格納されます。外部プロシージャの定義を作成するには、例 14-2 (14-6) で示すように、CREATE PROCEDURE 文を使用します。

例 14-2 CREATE PROCEDURE 文を使用した外部プロシージャの定義

```
CREATE PROCEDURE ADD_SOUNDEX_NAME (1) (14-6)
  (INOUT :PARAM1 INTEGER BY REFERENCE) (2) (14-6) ;
EXTERNAL NAME ADD_SOUNDEX_NAME (3) (14-7)
LOCATION 'ADD_SOUNDEX.EXE' (4) (14-7)
LANGUAGE FORTRAN (5) (14-7)
GENERAL PARAMETER STYLE (6) (14-7)
BIND ON SERVER SITE (7) (14-7)
BIND SCOPE TRANSACTION (8) (14-7)
NOTIFY ADD_SOUNDEX_NOTIFY ON BIND, TRANSACTION; (9) (14-8)
```

番号は例 14-2 (14-6) に対応しています。

(1) CREATE PROCEDURE ADD_SOUNDEX_NAME

Oracle Rdb データベースに含まれるオブジェクトとして外部プロシージャを作成し、外部プロシージャの定義の名前を指定します。
定義名は、外部ルーチンの定義および格納されているルーチン名のすべての中で、一意でなければなりません。

(2) INOUT :PARAM1 INTEGER BY REFERENCE

外部プロシージャで使用する INOUT パラメータについて、パラメータ名と SQL データ型 (INTEGER)、引渡し方法を指定します。パラメータ名はオプションですが、使用する場合は、先頭にコロン (:) を指定してください。外部プロシージャでは、IN、INOUT、OUT パラメータを使用できます。

SQL データ型は、LIST OF BYTE VARYING 以外はすべて指定できます。データ型の指定には、ドメイン名を使用することもできます。

各パラメータの引渡し方法を指定することもできます。引渡し方法を指定しないと、SQL は、REFERENCE でパラメータを渡します。パラメータの引渡し方法は、パラメータのデータ型と外部プロシージャの言語によって異なります。

パラメータとサポートされる引渡し方法の詳細は、14.11 項 (14-36) と 14.12 項 (14-38) を参照してください。.

(3) EXTERNAL NAME ADD_SOUNDEX_NAME

外部ルーチン・プログラムで宣言する外部プロシージャの名前を指定します。

(4) LOCATION 'ADD_SOUNDEX.EXE'

OpenVMS では、外部プロシージャのイメージ位置を指定する方法として、ファイル指定または論理名を使用できます。◆

Compaq Tru64 UNIX では、外部プロシージャの共有オブジェクトの位置を指定する方法として、絶対パス名または相対パス名を使用できます。◆

ロケーション句を指定しない場合や DEFAULT LOCATION 句を指定する場合、SQL は RDB\$ROUTINES をイメージの位置として使用します。

(5) LANGUAGE GENERAL

外部プロシージャを記述するホスト言語を指定します。

指定できるキーワードは、ADA、C、COBOL、FORTRAN、PASCAL、GENERAL です。GENERAL キーワードを指定すると、外部プロシージャが使用しているデータ型とパラメータの引渡し方法が Oracle Rdb でサポートされていれば、あらゆる言語で記述された外部プロシージャをコールできます。関数を記述した言語がわからない場合や、SQL でサポートされない言語で記述されている場合には、このキーワードを使用してください。

(6) GENERAL PARAMETER STYLE

外部プロシージャに引数を渡し、戻り値を受け取る方法を指定します。現在の SQL では、OpenVMS の呼出し規則に最も近いスタイル GENERAL を使用するように限定されています。Oracle Rdb の今後のバージョンでは、SQL 標準に準拠して、他のパラメータ・スタイルを実装する予定です。

(7) BIND ON SERVER SITE

SERVER SITE バインディングまたは CLIENT SITE バインディングを指定できます。

CLIENT SITE バインディングを使用できるのは、OpenVMS のみです。バインディングを指定すると、Oracle Rdb は、Oracle Rdb サーバーと同じプロセスで、外部ルーチンをアクティブ化して実行します。

SERVER SITE バインディングを指定すると、Oracle Rdb は、Oracle Rdb サーバーと同じノード上で、別のエグゼキュータ・プロセスで外部ルーチンをアクティブ化します。クライアント・サイト・バインディングとサーバー・サイト・バインディングの詳細は、14.10 項 (14-34) を参照してください。

(8) BIND SCOPE TRANSACTION

外部ルーチンがアクティブな状態の間と、ルーチンが非アクティブなときのスコープを指定します。TRANSACTION キーワードは、トランザクションの終了時に外部ルーチンを非アクティブ化することを示します。また、CONNECT キーワードを指定すると、外部ルーチンの定義が格納されているデータベースが切断されたときに、外部ルーチンは非アクティブ化されます。デフォルトは CONNECT です。

OpenVMS

Compaq Tru64 UNIX

(9) NOTIFY ADD_SOUNDEX_NOTIFY ON BIND, TRANSACTION

外部ルーチン・イメージまたは共有オブジェクトにある通知ルーチンのエントリ・ポイントの名前と、通知ルーチンを起動するイベントを指定します。通知ルーチンは、変数の初期化やデータベースへの接続や切断など、初期化やクリーンアップ操作を実行できます。通知ルーチンは、データベース関連のイベント情報を、外部ルーチンの本体と共有することもできます。

NOTIFY 句は、通知ルーチンを起動するイベントの種類を指定します。指定できるイベントは、BIND、CONNECT、TRANSACTION です。

NOTIFY 句の詳細は、14.13 項 (14-41) を参照してください。

CREATE PROCEDURE 文、構文、引数の詳細は、『Oracle Rdb7 SQL Reference Manual』の Create Routine 文を参照してください。

14.4 外部ルーチン定義の変更と削除

外部ルーチンの定義を変更するには、削除してから、変更後の内容で作成する必要があります。SQL では、ALTER FUNCTION 文や ALTER PROCEDURE 文などで、外部ルーチンの定義を変更することはできません。

Oracle Rdb データベースから外部関数の定義を削除するには、次の例のように、DROP FUNCTION 文を使用します。

```
DROP FUNCTION SOUNDEX;
```

Oracle Rdb データベースから外部プロシージャの定義を削除するには、次の例のように、DROP PROCEDURE 文を使用します。

```
DROP PROCEDURE CLEAR_SOUNDEX;
```

RESTRICT キーワード（デフォルト）を使用すると、そのルーチンを参照しているオブジェクトが Oracle Rdb データベース内に存在する場合は、外部ルーチンの定義を削除できなくなります。

次のオブジェクトが外部ルーチンを参照している場合、SQL は例外を返し、外部ルーチン定義の削除を禁止します。

- テーブル定義の COMPUTED BY 句
- 制約の定義
- トリガー定義
- ストアド・プロシージャ
- アクティブな要求

CASCADE キーワードを使用して外部ルーチンを削除すると、Oracle Rdb は削除操作を実行し、削除されたルーチンを参照しているストアド・プロシージャがあれば、それを無効に

します。DROP FUNCTION 文と DROP PROCEDURE 文の CASCADE キーワードの詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

14.5 外部ルーチンの作成

この項では、次の外部ルーチンを作成する方法を説明します。

- 既存のルーチンに基づく外部ルーチン
14.5.1 項 (14-9) を参照してください。
- ユーザー自身が記述する（ユーザー定義）外部ルーチン
14.5.2 項 (14-12) を参照してください。
- データベースにアクセスする外部ルーチン
14.5.3 項 (14-16) を参照してください。
- ジャケット・ルーチンが必要な外部ルーチン
14.5.4 項 (14-23) を参照してください。

14.5.1 既存のルーチンに基づく外部ルーチンの作成

この項では、一連の数値の平方根を計算する既存の外部関数の使用方法を説明します。

この関数はすでに存在しており、実行可能イメージは既存の共有可能イメージすなわち共有オブジェクトなので、ユーザーに要求される作業は、SQL で関数定義を作成し、SQL プログラムか対話型 SQL スクリプトを作成してこの関数をコールすることだけです。

OpenVMS
VAX

例 14-3 (14-9) に、OpenVMS VAX の実行時ライブラリ・ルーチン MTH\$SQRT を使用して、一連の数値の平方根を計算する関数の定義を作成する方法を示します。

例 14-3 OpenVMS VAX 上の既存の関数を使用した外部関数の定義

```
CREATE FUNCTION SQRT (IN :PARAM1 REAL)
  RETURNS REAL;
  EXTERNAL NAME MTH$SQRT LOCATION 'SYS$SHARE:MTHRTL.EXE'
  LANGUAGE GENERAL
  GENERAL PARAMETER STYLE; ◆
```

OpenVMS
Alpha

例 14-4 (14-9) に、OpenVMS Alpha の実行時ライブラリ・ルーチン MTH\$SQRT を使用して、一連の数値の平方根を計算する関数の定義を作成する方法を示します。

例 14-4 OpenVMS Alpha 上の既存の関数を使用した外部関数の定義

```
CREATE FUNCTION SQRT (IN :PARAM1 REAL)
  RETURNS REAL;
  EXTERNAL NAME MTH$SQRT LOCATION 'SYS$SHARE:DPML$SHR.EXE'
  LANGUAGE GENERAL
  GENERAL PARAMETER STYLE;
```

OpenVMS VAX の例との唯一の違いは、ルーチンが存在する場所です。◆

Compaq Tru64 UNIX

Compaq Tru64 UNIX 上では、F-float データと IEEE の S-float データを変換するジャケッ・ルーチンが必要です。次の例に示す `sqrt.c` ジャケッ・ルーチンは、`ftof.c` 内でユーティリ・ルーチンをコールし、システムが提供する `sqrtf` ルーチンを使用します。

```
#include <math.h>
extern void cvt_ff_to_fs( unsigned int *, float * );
extern void cvt_fs_to_ff( float *, unsigned int * );
extern void SQR( unsigned int *ffout, unsigned int *ffin ) {
float fsin,fsout;
cvt_ff_to_fs( ffin, &fsin );
fsout = sqrtf( fsin );
cvt_fs_to_ff( &fsout, ffout ); }
```

次の例に、システムが提供する浮動小数点数変換 (cvt) ルーチンをコールするユーティリ・ルーチン `ftof.c` を示します。

```
#include <cvt.h>
#include <excpt.h>
static int sts;
extern void cvt_ff_to_fs( unsigned int *ff, float *fs ) {
    sts = cvt_ftof( ff,CVT_VAX_F, fs,CVT_IEEE_S, CVT_FORCE_ALL_SPECIAL_VALUES);
    if (sts != CVT_NORMAL) exc_raise_status_exception( sts ); }
extern void cvt_fg_to_ft( unsigned long *fg, double *ft ) {
    sts = cvt_ftof( fg,CVT_VAX_G, ft,CVT_IEEE_T, CVT_FORCE_ALL_SPECIAL_VALUES);
    if (sts != CVT_NORMAL) exc_raise_status_exception( sts ); }
extern void cvt_fs_to_ff( float *fs, unsigned int *ff ) {
    sts = cvt_ftof( fs,CVT_IEEE_S, ff,CVT_VAX_F, CVT_FORCE_ALL_SPECIAL_VALUES);
    if (sts != CVT_NORMAL) exc_raise_status_exception( sts ); }
extern void cvt_ft_to_fg( double *ft, unsigned long *fg ) {
    sts = cvt_ftof( ft,CVT_IEEE_T, fg,CVT_VAX_G, CVT_FORCE_ALL_SPECIAL_VALUES);
    if (sts != CVT_NORMAL) exc_raise_status_exception( sts ); }
```

ジャケッ・ルーチンとユーティリ・ルーチンをコンパイルおよびリンクして共有オブジェクトを作成するには、次のコマンドを使用します。

```
$ cc -c -o sqrt.o sqrt.c
$ cc -c -o ftof.o ftof.c
$ ld -shared -soname $HOME/sqrt.so -o $HOME/sqrt.so sqrt.o ftof.o -lexc -lm -lc
```

例 14-5 (14-10) に、SQL で関数定義を作成する方法を示します。

例 14-5 Compaq Tru64 UNIX 上の既存の関数を使用した外部関数の定義

```
CREATE FUNCTION SQR( IN :PARAM1 REAL)
    RETURNS REAL BY REFERENCE;
    EXTERNAL NAME SQR LOCATION 'sqrt.so'
    LANGUAGE C
    GENERAL PARAMETER STYLE
```

BIND ON SERVER SITE; ◆

例 14-6 (14-11) は、外部関数をコールするプロシージャを含む SQL モジュール・ファイル `sql_sqrt_c.sqlmod` からの抜粋です。

例 14-6 SQL モジュールを使用した事前定義済み外部関数の呼出し

```
.
.
.
-----
-- Procedure to invoke the SQRT external function and insert values
-- into the SQUARE_ROOTS table
-----
PROCEDURE INSERT_SQUARE_ROOTS_TABLE
  (SQLCODE,
   :P_NUM  INTEGER);
  INSERT INTO SQUARE_ROOTS VALUES (:P_NUM, SQRT(CAST(:P_NUM AS REAL)));
.
.
.
```

例 14-7 (14-11) に、`sql_sqrt_c.sqlmod` 内のプロシージャをコールする C プログラム `sql_sqrt_mod.c` を示します。

例 14-7 事前定義済み関数の C プログラムによるコール

```
.
.
.
/* Declarations of entry points in the SQL module. */
extern void INSERT_SQUARE_ROOTS_TABLE(int *sqlcode, int *p_num);
main()
{
    int sqlcode = 0;
    int i, n;
    float s;
.
.
.
/* Call the SQL module procedure to insert the square root values into the
database. */
for(i=0;i<10;i++)
    INSERT_SQUARE_ROOTS_TABLE(&sqlcode, &i);
OPEN_E1(&sqlcode);
do {
    FETCH_E1(&sqlcode, &n, &s);
    if (sqlcode==0)
        printf("number=%d square_root=%f\n", n, s);
}
```

```
        } while (sqlcode==0);
    .
    .
    .
}
```

14.5.2 ユーザー定義の外部ルーチンの記述

この項では、一連の例を使用して、ユーザー定義の外部関数、すなわちユーザー自身が記述したコードに基づいて外部関数を記述する方法を説明します。

表 14-1 (14-12) に、ユーザー定義の SOUNDEX 関数のコールに必要なコンポーネントを示します。

表 14-1 ユーザー定義の外部関数を構成するコンポーネント

コンポーネント	説明
soundex_c.c	SOUNDEX 関数を記述した C プログラム
soundex_c.opt	ユニバーサル・シンボルやシンボル・ベクトルを定義するオプション・ファイル (OpenVMS)
sql_soundex_c.sqlmod	ユーザー定義の SOUNDEX 関数をコールするプロシージャを記述した SQL モジュール・ファイル
sql_soundex_mod.c	関連付けられた SQL モジュール・ファイルをコールする C のメイン・プログラム

例 14-8 (14-12) は、mf_personnel データベースに格納された従業員の姓に関する 4 文字のコードのリストを生成するユーザー定義の外部関数からの抜粋です。

例 14-8 C によるユーザー定義の外部関数の記述

```
/* This function is an implementation of the SOUNDEX routine,
   originally developed by Margaret K. Odell and Robert C. Russell
   and described in Knuth's "Sorting and Searching, Vol. 3, The
   Art of Computer Programming." */
#include <stdio.h>
.
.
.
void SOUNDEX (char *out_string, char *source_str)
.
.
.
```

次の例に、OpenVMS VAX システム用に宣言された soundex_c.opt オプション・ファイルの内容を示します。

```
UNIVERSAL = SOUNDEX
```

オプション・ファイル内で使用する名前は、CREATE FUNCTION 文で関数を定義するときに指定した名前と一致する必要があります。

プログラムのオブジェクト・ファイルとオプション・ファイルをリンクして、共有可能イメージを作成する必要があります。例 14-9 (14-13) に、SOUNDEX 関数を記述したプログラムをコンパイルし、このプログラムとオプション・ファイルをリンクして共有可能イメージを作成する方法を示します。

例 14-9 ユーザー定義の外部関数のコンパイルとリンク

```
$ CC/G_FLOAT SOUNDEX_C.C
$ LINK/SHAREABLE=SOUNDEX_C_IMAGE.EXE SOUNDEX_C, -
_$ SOUNDEX_C.OPT/OPT
```

例 14-10 (14-13) に、SQL で SOUNDEX 関数定義を作成する方法を示します。

表 14-10 ユーザー定義の関数を使用した外部関数の定義

```
CREATE FUNCTION SOUNDEX (IN CHAR(32)) RETURNS CHAR(4);
    EXTERNAL LOCATION 'SOUNDEX_C_IMAGE.EXE'
    LANGUAGE C
    GENERAL PARAMETER STYLE;
```

例 14-11 (14-13) に、SOUNDEX 関数をコールするプロシージャを記述した SQL モジュール・ファイルを示します。

例 14-11 ユーザー定義の外部関数の SQL モジュールによるコール

```
.
.
.
-----
-- Declare Statement Section
-----
DECLARE E1_CURSOR CURSOR FOR SELECT LAST_NAME, SOUNDEX_LAST_NAME
    FROM EMPLOYEES
DECLARE E2_CURSOR CURSOR FOR SELECT LAST_NAME, SOUNDEX_LAST_NAME
    FROM EMPLOYEES WHERE SOUNDEX_LAST_NAME=SOUNDEX('Varmelker')
.
.
.
-----
-- Procedure to fetch a row.
-----
PROCEDURE FETCH_E2
    (SQLCODE,
     :P_LAST_NAME CHAR(32),
     :P_SOUNDEX_LAST_NAME CHAR(4));
```

```
        FETCH E2_CURSOR INTO   :P_LAST_NAME, :P_SOUNDEX_LAST_NAME;
-----
-- Procedure to alter table.
-----
PROCEDURE ALTER_EMPLOYEES_TABLE
    (SQLCODE);
    ALTER TABLE EMPLOYEES ADD SOUNDEX_LAST_NAME CHAR(4);
-----
-- Procedure to update values in employees table.
-----
PROCEDURE UPDATE_EMPLOYEES
    (SQLCODE);
    UPDATE EMPLOYEES SET SOUNDEX_LAST_NAME = SOUNDEX(LAST_NAME);
```

例 14-12 (14-14) に、SOUNDEX 関数をコールする SQL モジュール・プロシージャをコールする C プログラムを示します。

例 14-12 ユーザー定義の外部関数の C プログラムによるコール

```
#include <stdio.h>
typedef char STR33[33];
typedef char STR5[5];
typedef int *sqlcode;
/* Declarations of entry points in the SQL module. */
.
.
.
extern void OPEN_E1(int *sqlcode);
extern void CLOSE_E1(int *sqlcode);
extern void FETCH_E1(int *sqlcode, STR33 last_name, STR5 soundex_last_name);
extern void ALTER_EMPLOYEES_TABLE(int *sqlcode);
extern void UPDATE_EMPLOYEES(int *sqlcode);
main()
{
    char last_name[33];
    char slast_name[5];
    int sqlcode = 0;
    .
    .
    .
    printf("\nAltering EMPLOYEES table to add SOUNDEX_LAST_NAME column\n");
    ALTER_EMPLOYEES_TABLE(&sqlcode);
    printf("\nUpdating EMPLOYEES table, setting SOUNDEX_LAST_NAME equal to\n");
    printf("SOUNDEX(LAST_NAME)\n");
    UPDATE_EMPLOYEES(&sqlcode);
    printf("\nPrint LAST_NAME and SOUNDEX_LAST_NAME\n");
    printf("LAST_NAME          SOUNDEX_LAST_NAME\n");
    OPEN_E1(&sqlcode);
```



```

do {
    FETCH_E1(&sqlcode, last_name, slast_name);
    if (sqlcode==0)
        printf("%s %s\n",
                last_name, slast_name);
    } while (sqlcode==0);
if (sqlcode!=100)
    printf("SQL error code = %d\n", sqlcode);
CLOSE_E1(&sqlcode);
printf("\nPrint LAST_NAME and SOUNDEX_LAST_NAME where SOUNDEX_LAST_NAME\n");
printf(" sounds like 'Varmelker')\n");
printf("LAST_NAME                SOUNDEX_LAST_NAME\n");
OPEN_E2(&sqlcode);
do {
    FETCH_E2(&sqlcode, last_name, slast_name);
    if (sqlcode==0)
        printf("%s %s\n",
                last_name, slast_name);
    } while (sqlcode==0);
if (sqlcode!=100)
    printf("SQL error code = %d\n", sqlcode);
CLOSE_E2(&sqlcode);
.
.
.
}

```

次の例に、SOUNDEX 関数をコールする SQL モジュールとホスト言語プログラムをコンパイルしてリンクする方法を示します。

```

$ CC/G_FLOAT SQL_SOUNDEX_MOD
$ SQL$MOD SQL_SOUNDEX_C
$ LINK SQL_SOUNDEX_MOD, SQL_SOUNDEX_C
$ RUN SQL_SOUNDEX_MOD

```

SQL_SOUNDEX_MOD プログラムを実行すると、次の結果が表示されます。

```

Altering EMPLOYEES table to add SOUNDEX_LAST_NAME column
Updating EMPLOYEES table, setting SOUNDEX_LAST_NAME equal to
SOUNDEX(LAST_NAME)
Print LAST_NAME and SOUNDEX_LAST_NAME
LAST_NAME                SOUNDEX_LAST_NAME
Smith                    S530
O'Sullivan                0024
Lasch                    L200
.
.
.

```

```

Keisling                K245
Vormelker               V654
.
.
.
Print LAST_NAME and SOUNDEX_LAST_NAME where SOUNDEX_LAST_NAME
sounds like 'Varmelker'
LAST_NAME              SOUNDEX_LAST_NAME
Vormelker              V654

```

14.6 項 (14-25) と 14.7 項 (14-29) では、サポートするすべてのプラットフォーム上で外部ルーチンをコンパイルしてリンクする方法を説明します。

14.5.3 データベースにアクセスする外部ルーチンの記述

ユーザーは、Oracle Rdb データベースにアクセスする外部ルーチンを記述できます。すなわち、外部ルーチンにはプリコンパイル SQL プログラムまたは SQL モジュール・プロシージャ内にデータ操作のための SQL 文を記述できるので、これらの文を使用してデータベースのデータを挿入、更新、削除することができます。

この項では、データベースにドキュメントを追加すると同時に、ドキュメントにインデックスを付ける外部ルーチンの使用方法を示します。この外部ルーチンは、SQL モジュール `sql_add_doc_mod.sqlmod` と C 言語プログラム `sql_add_doc.c` を使用して作成されています。外部ルーチンは、2 つのテーブル `DOC` と `DOC_INDEX` を含むデータベース `docdb` を使用します。次の例に、テーブルの定義を示します。

```

SQL> -- Create the table DOC to hold the document.
SQL> CREATE TABLE DOC (DOC_ID INTEGER, DOC_TEXT VARCHAR (500));
SQL>
SQL> -- Create the table DOC_INDEX to hold the index and information
SQL> -- about the occurrence of the index tokens.
SQL> CREATE TABLE DOC_INDEX
1>      (DOC_ID INTEGER, TOKEN CHAR (32),
2>      TOKEN_OCC_USED INTEGER, TOKEN_OCC VARCHAR (100),
3>      PRIMARY KEY (TOKEN, DOC_ID) NOT DEFERRABLE);

```

例 14-13 (14-16) に、C 言語プログラム `sql_add_doc.c` を示します。

例 14-13 データベースにアクセスする外部ルーチンの記述

```

/* External routine ADD_DOC
   Given the text of a document, create tokens by breaking the document into
   words, create an occurrence record for each unique token, and store a
   uniformly space-delimited form of the document.
   This external routine uses the SQL module, sql_add_doc_mod.sqlmod, for
   database operations, including a notify routine to control the attach
   and detach on routine binding and unbinding.  */
#include <string.h>

```

```

#include <stdio.h>
extern int ADD_DOC( int *, char * );
int add_token( int *, char *, int );
extern void ADD_DOC_NOTIFY( int *, int *, int *, int * );
void ATTACH_DB( int * );
void DETACH_DB( int * );
void START_TRAN( int * );
void COMMIT_TRAN( int * );
void ROLLBACK_TRAN( int * );
void STORE_DOC( int *, char *, int * );
void GET_DOC_INDEX_BY_TOKEN_ID( char *, int *, int *, char *, int * );
void STORE_DOC_INDEX( int *, char *, int *, char *, int * );
void UPDATE_DOC_INDEX( int *, char *, int *, char *, int * );
void sql_signal( void );
extern int ADD_DOC( int *doc_id, char *doc_text ) {
    int tokoff = 0, toklen = 0, docoff = 0, ndocoff = 0, dummy = 0, status = 0;
    char new_text[500+1], token[32+1];
    /* Start the transaction. */
    START_TRAN( &status );
    if (status !=0) return status;
    for (docoff =0; docoff<= strlen( doc_text ); docoff++) {
        /* At token break, capture token from new text and add to index. */
        if (doc_text[docoff] == ' ' || doc_text[docoff] == '\t' ||
            doc_text[docoff] == '\0') {
            if (toklen > 0) {
                strcpy( token, &new_text[tokoff], toklen );
                token[toklen] = '\0';
                status = add_token( doc_id, token, tokoff );
                if (status != 0) break;
                toklen = 0; tokoff = 0;
            }
        } else {
            /* Move document text to uniformly delimited new doc text. */
            if (ndocoff > 0 && toklen == 0)
                { new_text[ndocoff++] = ' '; tokoff = ndocoff; }
            new_text[ndocoff++] = doc_text[docoff]; toklen++;
        }
    }
    if (status !=0) { ROLLBACK_TRAN( &dummy ); return status; }
    /* Save uniformly delimited doc text. Store it in the database. */
    new_text[ndocoff] = '\0';
    STORE_DOC ( doc_id, new_text, &status );
    if (status !=0) { ROLLBACK_TRAN( &dummy ); return status; }
    /* Commit the transaction. */
    COMMIT_TRAN( &status );
    if (status !=0) { ROLLBACK_TRAN( &dummy ); return status; }
    return status;
}

```

```

}
int add_token( int *doc_id, char *token, int tokoff ) {
    int    used, status = 0;
    char   occ[100+1];
    /* Insert or update token occurrence index. */
    GET_DOC_INDEX_BY_TOKEN_ID( token, doc_id, &used, occ, &status );
    if (status == 100) { status = 0; used = 1; }
    else                { if (status == 0) used++; else return status; }
    sprintf( &occ[(used-1)*4], "%04d", tokoff );
    if (used == 1) STORE_DOC_INDEX( doc_id, token, &used, occ, &status );
    else            UPDATE_DOC_INDEX( doc_id, token, &used, occ, &status );
    return status;
}
extern void ADD_DOC_NOTIFY( int *func, int *u1, int *u2, int *u3 ) {
    int status;
    /* Attach to database on routine activation.
       Detach on routine deactivation. */
    switch (*func) {
        case 1: ATTACH_DB( &status ); if (status != 0) sql_signal(); break;
        case 2: DETACH_DB( &status ); if (status != 0) sql_signal(); break;
    }
    return;
}

```

例 14-13 (14-16) では、例 14-14 (14-18) の SQL モジュール・プロシージャをコールします。このプロシージャは、データベースに接続し、トランザクションを制御し、データベースにドキュメントを挿入し、ドキュメントにインデックスを付けます。

例 14-14 SQL モジュールによるデータベースへのアクセス

```

-- This SQL module, linked together with the C language program sql_add_doc.c
-- forms an external function, ADD_DOC.  The SQL module procedures perform
-- database operations, including attaching to the database,
-- controlling transactions, inserting, retrieving, and updating data.
--
-- This SQL module is also used by the C program, sql_locate_doc.c to retrieve
-- information from the docdb database.
MODULE ADDDOC
LANGUAGE C
PARAMETER COLONS
DECLARE ALIAS FOR FILENAME docdb
DECLARE FIND_DOC_INDEX_BY_TOKEN CURSOR FOR
    SELECT DOC_ID FROM DOC_INDEX WHERE TOKEN = :tok
PROCEDURE ATTACH_DB
    (SQLCODE);
    ATTACH 'FILENAME docdb';
PROCEDURE DETACH_DB
    (SQLCODE);

```

```

DISCONNECT DEFAULT;
PROCEDURE START_READ_TRAN
  (SQLCODE);
  SET TRANSACTION READ ONLY;
PROCEDURE START_TRAN
  (SQLCODE);
  SET TRANSACTION READ WRITE;
PROCEDURE COMMIT_TRAN
  (SQLCODE);
  COMMIT;
PROCEDURE ROLLBACK_TRAN
  (SQLCODE);
  ROLLBACK;
-- Insert a document into the database.
PROCEDURE STORE_DOC
  (:ID INTEGER, :TEXT CHAR(500), SQLCODE);
  INSERT INTO DOC VALUES (:ID, :TEXT);
-- Retrieve the index to determine if the index token already exists
-- or if it is a new token.
PROCEDURE GET_DOC_INDEX_BY_TOKEN
  (:TOK CHAR (32), :ID INTEGER, :USED INTEGER, :OCC CHAR (100), SQLCODE);
  SELECT DOC_ID, TOKEN_OCC_USED, TOKEN_OCC INTO :ID, :USED, :OCC
    FROM DOC_INDEX WHERE :TOK = TOKEN;
PROCEDURE GET_DOC_INDEX_BY_TOKEN_ID
  (:TOK CHAR (32), :ID INTEGER, :USED INTEGER, :OCC CHAR (100), SQLCODE);
  SELECT TOKEN_OCC_USED, TOKEN_OCC INTO :USED, :OCC
    FROM DOC_INDEX WHERE :TOK = TOKEN AND :ID = DOC_ID;
-- Insert new index tokens.
PROCEDURE STORE_DOC_INDEX
  (:ID INTEGER, :TOK CHAR (32), :USED INTEGER, :OCC CHAR (100), SQLCODE);
  INSERT INTO DOC_INDEX VALUES (:ID, :TOK, :USED, :OCC);
-- Update the occurrence count if the index token already exists.
PROCEDURE UPDATE_DOC_INDEX
  (:ID INTEGER, :TOK CHAR (32), :USED INTEGER, :OCC CHAR (100), SQLCODE);
  UPDATE DOC_INDEX SET TOKEN_OCC_USED = :USED, TOKEN_OCC = :OCC
    WHERE :TOK = TOKEN AND :ID = DOC_ID;

```

例 14-15 (14-19) に、データベース docdb に格納された外部ルーチン定義を示します。

例 14-15 データベースにアクセスする外部ルーチンの定義

```

SET QUOTING RULES 'SQL92';
-- Define the external routine that loads the index.
CREATE FUNCTION ADD_DOC
  (IN INTEGER, IN VARCHAR (500))
  RETURNS INTEGER;
  EXTERNAL NAME "ADD_DOC" LOCATION 'ADDDOC.EXE'
  LANGUAGE C GENERAL PARAMETER STYLE

```

```
      BIND ON SERVER SITE
      NOTIFY "ADD_DOC_NOTIFY" ON BIND;
```

データベースにアクセスするときに SQL が課す制限の詳細は、14.15 項 (14-43) を参照してください。

ADD_DOC 外部関数をコンパイルしてリンクする場合は、次の手順に従います（この手順は、OpenVMS VAX 上でのコンパイルとリンクに関するものです。14.6 項 (14-25) と 14.7 項 (14-29) に、サポートするすべてのプラットフォーム上で外部ルーチンをコンパイルしてリンクする方法を説明します）。

1. SQL モジュール・プロセッサを使用して SQL モジュールをコンパイルします。

```
$ SQLMOD
INPUT FILE> SQL_ADD_DOC_MOD
```

2. C ホスト言語プログラムをコンパイルします。

```
$ CC SQL_ADD_DOC
```

3. 次の行を含むリンカー・オプション・ファイル sql_add_doc.opt を作成します。

```
UNIVERSAL = ADD_DOC
UNIVERSAL = ADD_DOC_NOTIFY
PSECT_ATTR=RDB$MESSAGE_VECTOR,NOSHR
PSECT_ATTR=RDB$DBHANDLE,NOSHR
PSECT_ATTR=RDB$TRANSACTION_HANDLE,NOSHR
```

4. 共有可能イメージを作成します。

```
$ LINK/SHARE=SYS$LOGIN:ADDDOC.EXE SQL_ADD_DOC.OBJ, SQL_ADD_DOC_MOD.OBJ, -
$_SQL_ADD_DOC.OPT/OPT, -
$_SQL$USER/LIB
```

次の例に、対話型 SQL から ADD_DOC 外部関数を使用して、ドキュメントのテキストをデータベースにロードする方法を示します。

```
SQL> ATTACH 'FILENAME docdb';
SQL>
SQL> -- Declare a variable to retrieve the status of the statement.
SQL> DECLARE :STATUS INTEGER;
SQL> -- Populate the database with documents and index them.
SQL> BEGIN
1> SET :STATUS =
2>     ADD_DOC (1, 'An external function is a 3GL program that you invoke
by using the function name as a value expression in an SQL statement. ');
3> END;
SQL> PRINT 'The SQL error code is ', :status;
                                STATUS
The SQL error code is          0
SQL> BEGIN
```

```

1> SET :STATUS =
2> ADD_DOC (2, 'An external procedure is a 3GL program that you invoke
using the SQL CALL statement. ');
3> END;
SQL> PRINT 'The SQL error code is ', :status;
                                STATUS
The SQL error code is          0
SQL>
SQL> BEGIN
1> SET :STATUS =
2> ADD_DOC (3, 'An external routine is an external procedure or external
function, linked into a shareable image or shared module, and registered in the
database. ');
3> END;
SQL> PRINT 'The SQL error code is ', :status;
                                STATUS
The SQL error code is          0

```

次の例は、SQL モジュール内のプロシージャをコールして、データベース内の文字を検索する C 言語プログラムからの抜粋です。

```

.
.
.
int main (void) {
    int seaidx, toklen=0, status = 0, exitval = 0, doc_id;
    char search_text[101];
    /* Get the search string. */
    printf( "Search string: " );
    scanf( "%[^\n]", search_text );
    printf( "%n" );
    for (seaidx=0; seaidx<=strlen( search_text ); seaidx++) {
        if (search_text[seaidx] == ' ' || search_text[seaidx] == ' ' ||
            search_text[seaidx] == '¥0') {
            if (toklen > 0) { token[tokcnt][toklen] = '¥0';
                            toksiz[tokcnt++] = toklen; toklen = 0; }
            } else { token[tokcnt][toklen++] = search_text[seaidx]; }
        }
    }
    /* Search for relevant documents. */
    if (tokcnt > 0) {
        ATTACH_DB( &status );
        if (status != 0) { exitval = 1; goto done; }
        START_READ_TRAN( &status );
        if (status != 0) { exitval = 1; goto det_db; }
        OPEN_FIND_DOC_INDEX_BY_TOKEN( (char *)&token[0][0], &status );
        if (status == 100) goto rbk_db;
        if (status != 0) { exitval = 1; goto rbk_db; }
        while (1) {

```

```

        FETCH_ID_FROM_DOC_INDEX( &doc_id, &status );
        if (status == 100) break;
        if (status != 0) { exitval = 1; goto rbk_db; }
        status = check_doc_occ( 0, doc_id, 0 );
        if (status != 0) { exitval = 1; goto rbk_db; }
    }
    .
    .
    .
int check_doc_occ( int tokidx, int doc_id, int curoff ) {
    int idx, fndlen, nxtidx, nxtoff = 0, occoff = 0, status = 0, used = 0;
    char occ[100+1], doc[500+1];
    /* Match next token in document (with the id) using prior end offset. */
    GET_DOC_INDEX_BY_TOKEN_ID( (char *)&token[tokidx][0], &doc_id, &used, occ,
                                &status );
    if (status == 100 && tokidx != 0) return 0;
    if (status != 0) return status;
    for (idx=0; idx<used; idx++) {
        sscanf( &occ[idx*4], "%04d", &occoff );
        if (tokidx == 0) begoff = occoff;
        if (tokidx == 0 || curoff == occoff) {
            if (tokidx < tokcnt-1) {
                nxtoff = occoff + toksiz[tokidx] + 1; nxtidx = tokidx + 1;
                status = check_doc_occ( nxtidx, doc_id, nxtoff );
            } else {
                GET_DOC_BY_ID( &doc_id, doc, &status );
                if (status != 0) return status;
                fndlen = occoff + toksiz[tokidx] - begoff;
                printf( "Found doc %d, offset %d: %*.s\n", doc_id, begoff,
                        fndlen, fndlen, &doc[begoff] );
            }
        }
    }
    if (tokidx != 0 && occoff >= curoff) break;
}

```

次の例に、前述の C プログラムがコールする SQL モジュール・プロシージャを示します。
このプロシージャは、データベース内の文字列を検索します。

```

PROCEDURE GET_DOC_BY_ID
    (:ID INTEGER, :TEXT CHAR(500), SQLCODE);
    SELECT DOC_TEXT INTO :TEXT
    FROM DOC WHERE DOC_ID = :ID;
PROCEDURE GET_DOC_INDEX_BY_TOKEN_ID
    (:TOK CHAR (32), :ID INTEGER, :USED INTEGER, :OCC CHAR (100), SQLCODE);
    SELECT TOKEN_OCC_USED, TOKEN_OCC INTO :USED, :OCC
    FROM DOC_INDEX WHERE :TOK = TOKEN AND :ID = DOC_ID;
PROCEDURE OPEN_FIND_DOC_INDEX_BY_TOKEN
    (:TOK CHAR (32), SQLCODE);
    OPEN FIND_DOC_INDEX_BY_TOKEN;

```



```
PROCEDURE FETCH_ID_FROM_DOC_INDEX
(:ID INTEGER, SQLCODE);
  FETCH FIND_DOC_INDEX_BY_TOKEN INTO :ID;
PROCEDURE CLOSE_FIND_DOC_INDEX_BY_TOKEN
(SQLCODE);
  CLOSE FIND_DOC_INDEX_BY_TOKEN;
```

14.5.4 外部ルーチンをコールするジャケッ ト・ルーチンの記述

一部の外部ルーチンでは、必要な情報を返すために付加的な処理が必要になります。サブプログラム（この場合は外部関数）上でこのような付加的処理を実行するアプリケーション・プログラムを、ジャケッ ト・ルーチンと呼びます。ジャケッ ト・ルーチンを実行すると、事実上、外包プログラムの中で間接的に外部関数を実行することになります。たとえば、OpenVMS システム・サービ ス・ルーチンと一部の OpenVMS RTL ルーチンはジャケッ ト・ルーチンからコールする必要があります。

乱数ルーチンなど、一部の外部ルーチンは、外包のジャケッ ト・ルーチンから元になるなんらかの値（あるいは開始値）を渡さないと計算できません。たとえば、MYRANDOM ジャケッ ト・ルーチンは元になるなんらかの値を OpenVMS RTL MTH\$RANDOM ルーチンに渡し、RTL MTH\$RANDOM ルーチンがデータベースに挿入する一連の乱数を計算します。

ジャケッ ト・ルーチンが実行する付加的な処理は、非常に単純なものからきわめて複雑なものまで様々です。この項に示す MYRANDOM ジャケッ ト・ルーチンは、最も単純な例です。

表 14-2 (14-23) に、ジャケッ ト・ルーチンを介して OpenVMS RTL のコールに必要なコンポーネントを示します。

表 14-2 外部関数をコールするジャケッ ト・ルーチンを構成するコンポーネント

コンポーネント	説明
myrandom_c.c	OpenVMS RTL MTH\$RANDOM ルーチンをコールするジャケッ ト・ルーチン MYRANDOM を記述した C プログラム
myrandom_c.opt	ユニバーサル・シンボルを定義するオプション・ファイル
sql_myrandom_c.sqlmod	ジャケッ ト・ルーチンを介して OpenVMS RTL MTH\$RANDOM ルーチンをコールするプロシージャを記述した SQL モジュール・ファイル
sql_myrandom_mod.c	関連付けられた SQL モジュール・ファイルをコールする C プログラム

例 14-16 (14-24) に、MTH\$RANDOM ルーチンに元になる値を渡すジャケッ ト・ルーチン myrandom_c.c を示します。

例 14-16 C によるジャケッ・ルーチンの記述

```
static seed = 0;
float MYRANDOM()
{
    return (MTH$RANDOM( &seed));
}
```

次の例に、OpenVMS VAX システム用に宣言された myrandom_c.opt オプション・ファイルの内容を示します。

```
UNIVERSAL = MYRANDOM
```

オプション・ファイル内で使用する名前は、CREATE FUNCTION 文で関数を定義するときに指定した名前と一致する必要があります。例 14-17 (14-24) に、ジャケッ・ルーチンをコンパイルし、MTH\$RANDOM ルーチンとオプション・ファイルをリンクして共有可能イメージを作成する方法を示します。

例 14-17 ジャケッ・ルーチンのコンパイルとリンク

```
$ CC/G_FLOAT myrandom_c.c
$ LINK/SHAREABLE=myrandom_c_image.exe myrandom_c, -
_$ myrandom_c.opt/OPT
```

例 14-18 (14-24) に、SQL で関数定義を作成する方法を示します。

例 14-18 ジャケッ・ルーチンをコールする外部関数の定義

```
CREATE FUNCTION MYRANDOM() RETURNS REAL;
    EXTERNAL LOCATION 'DB_DISK:[DB_FUNCT]MYRANDOM_C_IMAGE.EXE'
    LANGUAGE C
    GENERAL PARAMETER STYLE;
```

例 14-19 (14-24) に、MTH\$RANDOM ルーチンをコールする一連のプロシージャを定義する SQL モジュール・ファイル sql_myrandom_c.sqlmod を示します。

例 14-19 SQL モジュールからのジャケッ・ルーチンのコール

```

.
.
.
-----
-- Procedure to insert random numbers
-----
PROCEDURE INSERT_RANDOM_NUMBER
    (SQLCODE);
    INSERT INTO RANDOM_NUMBERS VALUES (MYRANDOM());
```

例 14-20 (14-25) に、SQL モジュール・プロシージャをコールする C プログラム sql_myrandom_mod.c を示します。

例 14-20 C プログラムからのジャケット・ルーチンのコール

```

#include <stdio.h>
typedef int *SQLCODE;
/* Declarations of entry points in the SQL module. */
extern void INSERT_RANDOM_NUMBER(int *sqlcode);
.
.
.
main()
{
    int sqlcode = 0;
    int i;
    float r;
/* Invoke the external function. */
    for(i=0;i<10;i++)
        INSERT_RANDOM_NUMBER(&sqlcode);
    OPEN_E1(&sqlcode);
    do{
        FETCH_E1(&sqlcode, &r);
        if (sqlcode==0)
            printf("Random number = %f\n", r);
    } while (sqlcode==0);
.
.
.
}

```

ジャケット・ルーチンと外部ルーチンを使用したその他の例は、14.5.1 項 (14-9) を参照してください。

14.6 外部ルーチンの共有可能イメージの作成

OpenVMS
VAX ≡≡≡

OpenVMS
Alpha ≡≡≡

OpenVMS では、外部ルーチンを使用するには、共有可能イメージ内に既存のルーチンがすでに存在する場合を除いて、共有可能イメージを作成する必要があります。

データベースで指定するルーチンの存在位置は、共有可能イメージの実際の存在位置と同じでなければなりません。CREATE FUNCTION または CREATE PROCEDURE 文で論理名を使用する場合、論理値が正しいイメージを指していることを確認してください。

サーバー・サイト・バインディングを使用する場合、データベースで定義されている論理名は、エグゼキュータ・プロセスでも適切な設定である必要があり、これは、データベース・アクセスで使用されます。

イメージをリンクして共有可能イメージを作成するには、まず、外部ルーチンを含むプログラムをコンパイルします。

データベースのコールがルーチンに含まれない場合は、ホスト言語プログラムをコンパイルします。たとえば、次のコマンド行は、C 言語で記述された外部ルーチンをコンパイルします。

```
$ CC soundex_c.c
```

ルーチンが埋込み SQL を使って Oracle Rdb データベースをコールする場合は、SQL プリコンパイラでソース・コードをコンパイルします。たとえば、次のコマンド行は、埋込み SQL 文を含む C 言語プログラム `sql_test.sc` をコンパイルします。

```
$ SQL$PRE
SQLPRE> sql_test/CC/SQLOPTIONS=ROLLBACK_ON_EXIT
```

ルーチンが SQL モジュール・プロシージャを参照し、これが Oracle Rdb データベースをコールする場合、ホスト言語コンパイラでルーチンのソース・コードをコンパイルし、SQL モジュール・プロセッサで SQL モジュール・プロシージャをコンパイルします。たとえば、次のコマンド行は、SQL モジュール・プロセッサと C 言語コンパイラを使って外部ルーチンをコンパイルします。

```
$ SQLMOD
INPUT FILE> sql_add_doc_mod
$
$ CC sql_add_doc
```

次の項では、OpenVMS VAX と OpenVMS Alpha システム上で共有可能イメージを作成する方法を説明します。◆

14.6.1 OpenVMS VAX における外部ルーチンの共有可能イメージの作成

OpenVMS
VAX

OpenVMS VAX で、データベースを参照しない外部ルーチンを含む共有可能イメージをリンクするには、次の手順に従ってください。

1. オプション・ファイルを作成し、各外部ルーチンの外部ルーチン名を、ユニバーサル・シンボルとして次の形式で指定します。

```
UNIVERSAL = routine-name
```

グローバル変数を含む場合、オプションを指定しないで共有可能イメージを作成すると、グローバル変数に対応するプログラム・セクション (PSECTS) は、すべて共有可能として作成されます。したがって、イメージをインストールする必要があります。イメージをインストールしないためには、グローバル変数を共有不可 (NOSHR) として定義してください。

たとえば、外部ルーチン `SOUNDEX` の場合、オプション・ファイル `SOUNDEX_C.OPT` には、次のような行が格納されています。

```
UNIVERSAL = SOUNDEX
```

2. プログラムのオブジェクト・ファイルとオプション・ファイルをリンクし、共有可能イメージを作成します。

```
$ LINK/SHAREABLE=soundex_c_image.exe soundex_c,-
_$ soundex_c.opt/OPT
```

データベースをコールする外部ルーチンを含む共有可能イメージをリンクするには、次の手順に従ってください。

1. オプション・ファイルを作成し、各外部ルーチンの外部ルーチン名を、ユニバーサル・シンボルとして次の形式で指定します。

```
UNIVERSAL = routine-name
```

外部ルーチンが通知ルーチンを含む場合、外部ルーチンに含まれる各通知ルーチンについて、ユニバーサル・シンボルを追加してください。

グローバル変数を含む場合、オプションを指定しないで共有可能イメージを作成すると、グローバル変数に対応するプログラム・セクション (PSECTS) は、すべて共有可能として作成されます。したがって、イメージをインストールする必要があります。イメージをインストールしないためには、グローバル変数を共有不可 (NOSHR) として定義してください。

ADDDOC 外部ルーチンは、ADD_DOC_NOTIFY という通知ルーチンを含むため、オプション・ファイル sql_add_doc.opt には、それ自体のユニバーサル・シンボルとルーチン用のユニバーサル・シンボルが格納されています。オプション・ファイルには、次のような行が含まれます。

```
UNIVERSAL = ADD_DOC
UNIVERSAL = ADD_DOC_NOTIFY
PSECT_ATTR=RDB$MESSAGE_VECTOR,NOSHR
PSECT_ATTR=RDB$DBHANDLE,NOSHR
PSECT_ATTR=RDB$TRANSACTION_HANDLE,NOSHR
```

2. プログラムのオブジェクト・ファイルとオプション・ファイルをリンクし、共有可能イメージを作成します。

```
$ LINK/SHARE=sys$login:adddoc.exe sql_add_doc.obj, sql_add_doc_mod.obj, -
_$ sql_add_doc.opt/OPT, -
_$ SQL$USER/LIB ◆
```

14.6.2 OpenVMS Alpha における外部ルーチンの共有可能イメージの作成

OpenVMS
Alpha

OpenVMS Alpha で、データベースを参照しない外部ルーチンを含む共有可能イメージをリンクするには、次の手順に従ってください。

1. オプション・ファイルを作成し、各外部ルーチンの外部ルーチン名を、ユニバーサル・シンボルとして次の形式で指定します。

```
SYMBOL_VECTOR = (routine-name = PROCEDURE )
```

グローバル変数を含む場合、オプションを指定しないで共有可能イメージを作成すると、グローバル変数に対応するプログラム・セクション (PSECTS) は、すべて共有可能として作成されます。したがって、イメージをインストールする必要があります。イメージをインストールしないためには、グローバル変数を共有不可 (NOSHR) として定義してください。

たとえば、外部ルーチン SOUNDEX の場合、オプション・ファイル SOUNDEX_C.OPT には、次のような行が格納されています。

```
SYMBOL_VECTOR = (SOUNDEX = PROCEDURE)
```

2. プログラムのオブジェクト・ファイルとオプション・ファイルをリンクし、共有可能イメージを作成します。

```
$ LINK/SHAREABLE=soundex_c_image.exe soundex_c,-  
_ $ soundex_c.opt/OPT
```

データベースをコールする外部ルーチンを含む共有可能イメージをリンクするには、次の手順に従ってください。

1. オプション・ファイルを作成し、各外部ルーチンの外部ルーチン名を、ユニバーサル・シンボルとして次の形式で指定します。

```
SYMBOL_VECTOR = (routine-name = PROCEDURE)
```

外部ルーチンが通知ルーチンを含む場合、外部ルーチンに含まれる各通知ルーチンについて、シンボル・ベクトルを追加してください。

```
SYMBOL_VECTOR = (notify-routine-name = PROCEDURE)
```

グローバル変数を含む場合、オプションを指定しないで共有可能イメージを作成すると、グローバル変数に対応するプログラム・セクション (PSECTS) は、すべて共有可能として作成されます。したがって、イメージをインストールする必要があります。イメージをインストールしないためには、グローバル変数を共有不可 (NOSHR) として定義してください。

ADDDOC 外部ルーチンは、ADD_DOC_NOTIFY という通知ルーチンを含むため、オプション・ファイル sql_add_doc.opt には、それ自体のシンボル・ベクトルとルーチン用のシンボル・ベクトルが格納されています。オプション・ファイルには、次のような行が含まれます。

```
SYMBOL_VECTOR = (ADD_DOC = PROCEDURE)  
SYMBOL_VECTOR = (ADD_DOC_NOTIFY = PROCEDURE)  
PSECT_ATTR=RDB$MESSAGE_VECTOR,NOSHR  
PSECT_ATTR=RDB$DBHANDLE,NOSHR  
PSECT_ATTR=RDB$TRANSACTION_HANDLE,NOSHR
```

2. プログラムのオブジェクト・ファイルとオプション・ファイルをリンクし、共有可能イメージを作成します。

```
$ LINK/SHARE=sys$login:adddoc.exe sql_add_doc.obj, sql_add_doc_mod.obj, -  
_ $ sql_add_doc.opt/OPT, -  
_ $ SQL$USER/LIB ◆
```

14.7 Compaq Tru64 UNIX における共有オブジェクトの作成

Compaq Tru64 UNIX

Compaq Tru64 UNIX では、外部ルーチンを使用するには、共有オブジェクト内に既存のルーチンがすでに存在する場合を除いて、共有オブジェクトを作成する必要があります。

データベースのコールがルーチンに含まれない場合は、次の手順に従って、共有オブジェクトを作成してください。

1. イメージをリンクして共有オブジェクトを作成するには、まず、外部ルーチンを含むプログラムをコンパイルします。たとえば、次のコマンド行は、C 言語で記述された外部ルーチンをコンパイルします。

```
$ cc -c -o soundex_c.o soundex_c.c
```

2. コマンド行で `-shared` オプションを指定して、モジュールをリンクします。次の例では、SOUNDEX 外部ルーチン用の共有オブジェクトを作成します。

```
$ ld -shared -soname ${HOME}/soundex_c_image.exe -o ¥
>    ${HOME}/soundex_c_image.exe soundex_c.o ¥
>    -no_archive -nocount -lots -lc
```

データベースをコールする外部ルーチンを含む共有オブジェクトを作成するには、次の手順に従ってください。

1. ルーチンが埋込み SQL を使って Oracle Rdb データベースをコールする場合は、SQL プリコンパイラでソース・コードをコンパイルします。たとえば、次のコマンド行は、埋込み SQL 文を含む C 言語プログラム `sql_test.sc` を、SQL プリコンパイラを使ってコンパイルします。

```
$ sqlpre -l cc sql_test.sc
```

ルーチンが SQL モジュール・プロシージャを参照し、これが Oracle Rdb データベースをコールする場合、ホスト言語コンパイラでルーチンのソース・コードをコンパイルし、SQL モジュール・プロセッサで SQL モジュール・プロシージャをコンパイルします。たとえば、次のコマンド行は、SQL モジュール・プロセッサと C 言語コンパイラを使って、ADD_DOC 外部ルーチンをコンパイルします。

```
$ sqlmod -int32 sql_add_doc_mod.sqlmod
$
$ cc -c -o sql_add_doc.o sql_add_doc.c
```

2. コマンド行で `-shared` オプションを指定して、モジュールをリンクします。次の例では、リンクによって、ADDDOC 外部ルーチン用の共有オブジェクトを作成します。

```
$ ld -hidden_symbol 'RDB$TRANSACTION_HANDLE' -hidden_symbol 'RDB$DBHANDLE' ¥
>    -hidden_symbol 'RDB$MESSAGE_VECTOR' -shared ¥
>    -soname ${HOME}/adddoc.exe -o ${HOME}/adddoc.exe ¥
>    sql_add_doc.o sql_add_doc_mod.o -no_archive -nocount ¥
>    -lsql -lrdbshr -lcosi -lots -lc ◆
```

14.8 外部ルーチンの起動

外部ルーチンを起動するには、外部ルーチンへの明示的な実行アクセスが必要です。OpenVMS SYSPRV や Compaq Tru64 UNIX のスーパー・ユーザー・アカウントのようなシステム権限には、上書きとしての機能はありません。

14.8.1 外部関数の起動

外部関数は、SQL 文で値式を指定できる任意の場所から起動できます。

例 14-21 (14-30) の SQL 文は、様々な値式のロケーションから外部関数 SOUNDEX を起動する方法を示しています。次の例に、SOUNDEX 外部関数の定義を示します。

```
CREATE FUNCTION SOUNDEX(IN CHAR(32))
  RETURNS CHAR(4);
  EXTERNAL LOCATION 'SOUNDEX$EXE' LANGUAGE C
  GENERAL PARAMETER STYLE;
```

例 14-21 SQL 文での外部関数の起動

```
-- Invoke the external function in the select list of a SELECT statement
-- to display an employee's last name and the last name generated by the
-- SOUNDEX external function.
SELECT  LAST_NAME, SOUNDEX(LAST_NAME)
  FROM EMPLOYEES;

-- Invoke the external function in a WHERE clause to find employees
-- whose name sounds like "Barns".
SELECT LAST_NAME
  FROM EMPLOYEES
 WHERE SOUNDEX(LAST_NAME) = SOUNDEX('Barns');

-- Invoke the external function to insert column values into rows of
-- a table.
INSERT INTO EMPLOYEES
  (LAST_NAME, SOUNDEX_LAST_NAME)
  VALUES
    ('Barnes', SOUNDEX('Barnes'));

-- Invoke the external function twice in a column constraint definition.
-- You can also use an external function in a table constraint definition.
CREATE TABLE EMPLOYEES
  (LAST_NAME CHAR(32),
   FIRST_NAME CHAR(32),
   CHECK(SOUNDEX(LAST_NAME) <> SOUNDEX(FIRST_NAME))
  );

-- Invoke an external function in a COMPUTED BY clause.
CREATE TABLE EMPLOYEES_CACHE
  (LAST_NAME,
   SOUNDEX_LAST_NAME COMPUTED BY SOUNDEX(LAST_NAME));
```


外部関数に渡すパラメータの数、データ型、長さは、仮パラメータと一致していなければなりません。

14.8.2 トリガー内での外部関数の起動

外部関数は、トリガーされるアクションの述語またはトリガーされる文（INSERT、DELETE、UPDATE）のアクションとして、トリガー内で起動できます。

例 14-22 (14-31) では、SOUNDEX 外部関数を、トリガーされる INSERT 文内で起動しています。

例 14-22 トリガー定義内での外部関数の起動

```
CREATE TRIGGER LOG_UNUSUAL_NAME_CHANGES
  AFTER UPDATE OF LAST_NAME
  ON EMPLOYEES
  REFERENCING OLD AS OEMP
               NEW AS NEMP
  WHEN (SOUNDEX(OEMP.LAST_NAME) = SOUNDEX(NEMP.LAST_NAME))
  (INSERT INTO UNUSUAL_NAME_CHANGES_LIST
    (OEMP.LAST_NAME,
     NEMP.LAST_NAME,
     SOUNDEX(OEMP.LAST_NAME))
  FOR EACH ROW;
```

従業員データベースから従業員レコードを削除するたびに、メール・メッセージを受け取りたいとします。メール・メッセージには、従業員のレコードと従業員の識別番号に対する両方のタイプの操作（DELETE）が含まれます。次の OpenVMS の例では、トリガー定義で外部関数を実行し、メール・メッセージを送信する方法の 1 つを示しています。トリガー定義での外部関数の使用方法の詳細は、『Oracle Rdb7 Guide to Database Design and Definition』を参照してください。

例 14-23 (14-31) に、外部関数の定義を示します。

例 14-23 外部関数でのデータベース操作の追跡

```
CREATE FUNCTION SEND_MAIL
  (IN CHAR(32),
   IN CHAR(32),
   IN CHAR(256))
  RETURNS INTEGER;
  EXTERNAL NAME SEND_MAIL
  LOCATION 'SUPPORT_FUNCTIONS'
  LANGUAGE C
  GENERAL PARAMETER STYLE;
```

最初の引数ではメールの送信先となるユーザーの名前、2 番目の引数ではメールの件名、3 番目の引数ではメッセージの内容を渡します。

例 14-24 (14-32) に、トリガー定義を示します。

例 14-24 トリガーと外部関数を使ったデータベース操作の追跡

```
CREATE TRIGGER EMPLOYEE_ID_CASCADE_DELETE
  BEFORE DELETE ON EMPLOYEES
  WHEN SEND_MAIL ('godfrind',
    'Employee ' || EMPLOYEES.EMPLOYEE_ID || ' deleted',
    'User ' || CURRENT_USER || ' just deleted employee ' ||
    EMPLOYEES.EMPLOYEE_ID ) <> 0
  ( ERROR )
  FOR EACH ROW;
```

SEND_MAIL 外部関数は、戻り値を使用して、メール操作の成否を判断します。従業員レコードを削除する前に、SEND_MAIL 外部関数をコールします。トリガーは、データベース操作に関する情報を、外部関数に渡します。

14.8.3 外部プロシージャの起動

例 14-25 (14-32) では、CALL 文を使用して外部プロシージャを起動します。

例 14-25 外部プロシージャの起動

```
BEGIN
  SET :ERROR = 0;
  CALL ADD_SOUNDINDEX_NAME (:ERROR);
END;
```

外部プロシージャに渡すパラメータは、数、データ型、長さが、仮パラメータと一致していなければなりません。

14.9 ルーチンの実行特性の指定

すべての外部ルーチンは、エグゼキュータ・マネージャによって制御されます。エグゼキュータ・マネージャやすべての外部ルーチンは、OpenVMS ユーザー・モードなど特権なしの実行モードで処理されます。

外部ルーチンの定義の作成では、BIND ON CLIENT SITE 句や BIND SERVER SITE 句の指定によって、ルーチンの実行特性の多くを制御できます。

OpenVMS VAX  OpenVMS Alpha 

CLIENT SITE バインディングは、OpenVMS でのみ指定できます。このバインディングを指定すると、Oracle Rdb は、サーバーと同じプロセスで、外部ルーチンをアクティブ化して実行します。このバインディングは、最も効率的な実行特性を提供します。このバインディングを指定すると、サーバーがクライアント・アプリケーションと同じプロセスにある場合、クライアント・リソースの共有を可能にし、外部ルーチンを、クライアント・アプリケーションの一部であるかのようにデバッグすることができます。ただし、アドレス空間に制限があり、SYSTEM_USER 実行環境に制限されます。

さらに、アプリケーション・プロセスの特権が、アプリケーションを実行しているユーザーよりも高い場合、Oracle Rdb はエラーを返します。これは、ユーザーには付与されていない特権を使用してインストールしたアプリケーション・イメージが存在するときに発生します。Oracle Rdb は、例外を生成することで、通常はアクセス権のないレベルの高い特権を使用するランダム外部ルーチンのセキュリティ上の問題を回避します。◆

SERVER SITE バインディングは、いずれのプラットフォームでも指定できます。SERVER SITE バインディングを指定すると、Oracle Rdb は、Oracle Rdb サーバーと同じノード上の、別のエグゼキュータ・プロセスで外部ルーチンをアクティブ化し、実行します。SERVER SITE バインディングは、適正な実行特性、クライアント・サイト・バインディングよりも大きなアドレス空間、本格的な SESSION_USER 実行環境などを提供し、クライアント・プロセスへの特権付与に関連する制限はありません。ただし、このバインディングでは、クライアント・リソースを共有することはできず、ルーチンのデバッグも困難な場合があります。

SERVER SITE バインディングを指定すると、外部ルーチンを起動した SESSION_USER のために、サーバー・サイト・エグゼキュータ・プロセスは初期化されます。つまり、エグゼキュータ・プロセスは、ユーザー名とユーザー識別、デフォルトのログイン・デバイスとディレクトリ、さらに OpenVMS 上では、セッション・ユーザーの特権とクォータを使用します。

プロセス環境は、ログイン・コマンド・スクリプトのないログインで設定される環境に類似しています。つまり、このプロセス環境は、DCL LOGIN/NOCOMMAND コマンドを使用して OpenVMS システムにログインしたときに提供される環境や、ログイン、プロファイル、リソース構成（*rc）などのファイルなしで Compaq Tru64 UNIX システムにログインしたときに提供される環境に類似しています。プロセスは、非対話型（つまり、OpenVMS で切断）であり、コマンド行インタープリタやコマンド・シェルはなく、すべての標準 I/O デバイスは NULL デバイスに接続され、ファイル作成の保護やその他マスクにはシステム・デフォルトを使用します。

これまでに説明した通り、サーバー・サイト・バインディングを指定すると、外部ルーチンは、セッション・ユーザーの環境を使用します。セッション・ユーザーは、現在の SQL セッションの認可された識別子で指定されているユーザーです。次の例を使用して、セッション・ユーザーが、どのような場合にどのような方法で使用されるかを理解してください。

- 名前が USERA というユーザーがいます。このユーザーが OpenVMS システムにログインすると、USERA というプロセス・システム・ユーザーが作成されます。
- このユーザーは、USER 句を指定しないで、データベース D1 に接続します。その結果、データベース D1 を照会すると、セッション・ユーザーは、SYSTEM_USER と同じ USERA となっています。クライアント・サイトとサーバー・サイトのいずれの外部ルーチンも、USERA のシステム環境で実行されます。
- 次に、USER 'USERX' 句を指定して、データベース D2 に接続します。その結果、データベース D2 を照会すると、セッション・ユーザーは USERX ですが、SYSTEM_USER は USERA です。サーバー・サイトのエグゼキュータ・プロセスを作成するときに、セッ

セッション・ユーザーを USERX で宣言したので、サーバー・サイトの外部ルーチンはシステム環境 USERX で実行されます。OpenVMS プロセスのシステム・ユーザーは USERA なので、クライアント・サイトの外部ルーチンは、システム環境 USERA で実行されます。

サーバー・サイト・バインディングのエグゼキュータ・マネージャで予期しないエラーが発生すると、現在のルーチンは強制終了され、例外が返されます。この後、このエグゼキュータ・プロセスを使用しようとする、同じ例外が発生し、このセッション・ユーザーのデータベースをすべて切断するまで、この状態は続きます。この時点で、エグゼキュータ・プロセスは破棄され、次にサーバー・サイト・バインディングでルーチンを実行するときに、新しいエグゼキュータ・プロセスが開始します。

Compaq Tru64 UNIX

Compaq Tru64 UNIX では、サーバー・サイト・バインディングを初期化することにより、通常はプロセスを停止または終了するトラップ可能なシグナルは、発行時に例外へ変換されます。◆

データベース・サーバーとサーバー・サイト・バインディングのエグゼキュータ・マネージャは、共有メモリーを使用して、ルーチンのパラメータ・データの操作と情報の制御を行います。共有メモリーのサイズは、デフォルトで 31 ページです。共有メモリーに異なるサイズを選択するには、RDMS\$RTX_SHRMEM_PAGE_CNT 論理名または RDB_RTX_SHRMEM_PAGE_CNT 構成パラメータに異なる値を設定してください。

14.10 ルーチンのアクティブ化と非アクティブ化

外部ルーチン定義の作成では、適切な外部ルーチン・イメージがアクティブになっていることを確認してください。アクティブ化は、次のいずれかの方法で実行できます。

- イメージ・ファイルを指定します。デフォルトのイメージ名は使用しないでください。
- 正しいファイル名を使用します。イメージ・ファイルの指定では、デフォルト・コンポーネントを使用しないでください。OpenVMS では、デバイスとディレクトリ、または両方に変換される論理名を指定します。Compaq Tru64 UNIX では、絶対パスを指定します。
- OpenVMS では、次のような規則に従ってください。
 - ファイル指定では、論理名を使用しないでください。物理デバイス名を指定してください。
 - /SYSTEM/EXECUTIVE_MODE/CONCEALED で定義されている論理名のみを使用してください。
 - ルーチンの定義で SYSTEM LOGICAL_NAME TRANSLATION を指定し、すべての論理名が /SYSTEM/EXECUTIVE_MODE で定義され、変換は 1 つのみであることを確認してください。
WITH SYSTEM LOGICAL_NAME TRANSLATION 句を使用する場合、論理名は展開されます。展開されたファイル指定は、255 文字以下にしてください。

OpenVMS VAX
OpenVMS Alpha

- 論理値は、イメージを使用するすべてのノード上で定義されていることを確認してください。

ユーザー・モード論理名は、テスト期間中にのみ使用してください。ユーザー・モード論理名は、論理名を再定義することにより、異なるルーチン・イメージを柔軟に使用できるので、本番でない環境のみで使用してください。◆

Oracle Rdb では、本番環境で使用するルーチン・イメージは、可能であれば、保護付きライブラリに配置することをお勧めします。

OpenVMS
VAX ≡≡≡
Alpha ≡≡≡

OpenVMS では、システム・レベルのエグゼクティブ・モード論理名 RDB\$LIBRARY が参照するディレクトリにイメージを配置し、ルーチン定義で SYSTEM LOGICAL_NAME TRANSLATION を指定してください。

例 14-26 (14-35) では、保護付き外部関数の定義を Oracle Rdb データベースに追加しています。

例 14-26 外部関数の定義の保護

```
CREATE FUNCTION SECURE_FUNC (IN CHAR(32) BY REFERENCE)
  RETURNS TINYINT;
  EXTERNAL LOCATION 'RDB$LIBRARY:SECURE_FUNC.EXE'
  WITH SYSTEM LOGICAL_NAME TRANSLATION
  LANGUAGE C
  GENERAL PARAMETER STYLE;
```

OpenVMS では、サーバー・サイト・バインディングでは特に、外部ルーチン・イメージをインストールする必要はありません。クライアント・サイトのバインディングを使用し、アプリケーション・イメージが権限付きイメージである場合、ルーチン・イメージをインストールする必要があり、ロケーション・ファイルの指定には、システム・レベルのエグゼクティブ・モード論理名のみを含む必要があります。◆

Compaq Tru64 UNIX
≡≡≡

Compaq Tru64 UNIX では、次のディレクトリに共有オブジェクトを配置してください。

/usr/local/dbs/shlib ◆

Oracle Rdb は、ルーチンを最初に起動するときに、外部ルーチン・コードを含む共有可能イメージを動的にアクティブ化します。ルーチンを含むイメージがアクティブになっている間、各ルーチンに 1 回のみ、動的なアクティブ化によるオーバーヘッドが発生します。

外部ルーチン・イメージを物理的に置換しても、これに対応付けられているアクティブな外部ルーチンは無効化されません。このように、すでにアクティブになっているルーチン・イメージで定義されているルーチンを起動すると、古いイメージのコードが実行されます。コードの置換後、起動によって物理的なアクティブ化が発生した場合にのみ、新しいイメージとルーチン・コードが使用されます。

エグゼキュータ・プロセスからは、論理的なルーチンの非アクティブ化が発生し、プラットフォームによっては、物理的な非アクティブ化も発生します。非アクティブ化は、バインド・スコープの最後で発生し、外部ルーチンがアクティブ化されている状態でのみ発生します。

OpenVMS VAX  OpenVMS Alpha 

OpenVMS では、物理的な非アクティブ化（アドレス空間からの削除）は、SERVER SITE バインディングで定義したルーチンのみで発生します。すべてのデータベースが切断され、SERVER SITE エグゼキュータ・プロセスが破壊されたときに、非アクティブ化が大量に発生します。◆

外部ルーチンを非アクティブ化しても、ルーチンが1つ以上のデータベースに接続されている場合、Oracle Rdb は、すべてのアクティブなトランザクションをロールバックして、すべてのデータベースを切断します。Oracle Rdb では、クリーンアップ操作の間に例外が発生しても、無視します。

14.11 パラメータの宣言および引渡しと戻り値

パラメータや関数の戻り値の定義の方法やパラメータの引渡し方法は、外部ルーチンで使用する言語によって異なります。14.12 項 (14-38) では、各言語でのガイドラインを説明しています。次に示すガイドラインは、すべての言語にあてはまります。

- VALUE によって INOUT または OUT パラメータを渡すことはできません。
- 関数の定義で宣言する関数値の引渡し方法が、REFERENCE または LENGTH、または OpenVMS 上では DESCRIPTOR の場合、対応する 3GL ルーチン・コードでは、出力関数値パラメータにアクセスするプロシージャを、最初の引数として示さなくてはなりません。FORTRAN などの一部の言語では、これを自動的に処理する関数固有の構文が用意されています。
- 文字列データ項目が、対応するパラメータよりも長い場合、SQL は、入力ではデータを切り捨て、出力では空白を追加します。文字列データ項目が、対応するパラメータよりも長い場合、SQL は、入力ではデータを切り詰め、出力では空白を追加します。
- 外部関数がサポートするのは、OpenVMS の F 浮動小数点と G 浮動小数点のデータ型のみです。IEEE 浮動小数点データ型（S 浮動小数点と T 浮動小数点）が必要なルーチンは、入力および出力の浮動小数点データを、使用する前と後に変換する必要があります。したがって、Compaq Tru64 UNIX では、サポート対象外の浮動小数点データを、浮動小数点ではないパラメータとして渡す必要があります。
- 入力または出力パラメータや関数値に関連する値が初期化されていない場合、結果は不定です。
- ルーチンの起動で例外が発生した場合、OUT および INOUT アクセス・モード・パラメータの出力値（関数値を含む）は不定です。
- 出力データ（関数値データを含む）は、アドレスを示すことはできません。

パラメータの引渡し方法を選択および使用する際は、次の点に注意してください。

- BY REFERENCE

- 入力および出力データは、実際のデータを参照するアドレス引数として渡されます。アドレス引数は、OpenVMS では 32 ビット、Compaq Tru64 UNIX では 64 ビットです。
- ルーチンのホスト言語を C と指定した場合、SQL は、文字データ型 CHAR(n)、NCHAR(n)、VARCHAR(n)、NCHAR VARYING(n) および LONG VARCHAR を NULL 終了文字列として渡し、実際の VARCHAR 文字列は、16 ビット長のフィールドを含みません。

■ BY VALUE

入力データと関数値データは、値として渡されます。OpenVMS で有効な入力データ型は、SMALLINT、TINYINT、INTEGER、REAL (32 ビット) です。Compaq Tru64 UNIX と、関数値として有効なデータ型は、SMALLINT、TINYINT、INTEGER、REAL (32 ビット)、BIGINT、DOUBLE PRECISION、DATE VMS、DATE ANSI、TIME、TIMESTAMP、INTERVAL (64 ビット) です。

OpenVMS OpenVMS
VAX Alpha

■ BY DESCRIPTOR

- 入力および出力文字データは、OpenVMS の固定長 (クラス S) 文字列 (タイプ T または VT) 記述子を参照するアドレス引数として渡され、これが実際の文字データを参照します。記述子は読取り専用であり、アドレスは両方とも 32 ビットです。
- 固定長の文字列 (タイプ T) 記述子では、記述子の長さの値が、実際の引数の長さと宣言したパラメータの長さの最小値です。可変長文字列 (タイプ VT) 記述子では、記述子の長さの値は、宣言したパラメータの長さです。◆

■ BY LENGTH

OpenVMS OpenVMS
VAX Alpha

Compaq Tru64 UNIX

- OpenVMS では、BY LENGTH は BY DESCRIPTOR と同じように機能します。◆
- Compaq Tru64 UNIX では、入力および出力データは、実際の文字データを直接または間接的に参照するアドレス引数として渡されます。
Compaq Tru64 UNIX Pascal では、パラメータ引数は、文字列と長さを参照するアドレスで構成されるプライベート記述子を参照するアドレスです。Compaq Tru64 UNIX FORTRAN では、パラメータ引数は、文字列を参照するアドレスであり、さらに隠し引数が、文字列の長さを渡します。アドレスはすべて 64 ビットです。
- Compaq Tru64 UNIX では、固定長文字列の場合、渡す値の長さが、実際の引数の長さと宣言したパラメータの長さの最小値となります。他の文字データ型については、渡す長さの値は、宣言したパラメータの長さになります。◆

14.12 外部ルーチンの記述に関する言語固有のガイドライン

Oracle Rdb は、あらゆるプログラミング言語を使用した外部ルーチンの記述をサポートしています。ただし、Ada、C、COBOL、FORTRAN、Pascal 以外の言語で外部ルーチンを記述する場合は、CREATE FUNCTION または CREATE PROCEDURE 文の LANGUAGE 句で GENERAL キーワードを指定する必要があります。GENERAL を指定すれば、あらゆる言語で記述した外部関数をコールできますが、外部関数で使用可能なデータ型とパラメータの引渡し方法は、Oracle Rdb がサポートするものに限定されます。

サポートされるデータ型とパラメータの引渡し方法は、次の項を参照してください。

14.12.1 Ada での外部ルーチンの使用方法

Ada 言語で外部ルーチンを使用する場合には、次のガイドラインに従ってください。

- 次の SQL データ型は、Ada に直接対応しています。TINYINT、SMALLINT、INTEGER、REAL、DOUBLE PRECISION、CHAR(n)、VARCHAR(n)
- パラメータの引渡し方法は、パラメータのデータ型によって異なります。
 - 数値の引数は、REFERENCE で渡します。
 - OpenVMS では、固定長文字列は DESCRIPTOR または LENGTH で渡します。
 - Compaq Tru64 UNIX では、固定長文字列は REFERENCE で渡します。
 - 可変長文字列は、REFERENCE で渡します。
- 外部関数の戻り値の引渡し方法は、戻り値のデータ型によって異なります。
 - 数値のデータ型は、VALUE で返されます。
 - OpenVMS では、固定長文字列は DESCRIPTOR または LENGTH で返されます。
 - Compaq Tru64 UNIX では、固定長文字列は、8 文字以下の場合は VALUE、9 文字以上の場合は REFERENCE で返されます。
 - 可変長文字列は、REFERENCE で返されます。
- DEC Ada V3.0 以下では、パラメータの引渡し方法はコンパイラによって異なります。コンパイルでオプション 'warnings=compilation_notes' を使用すると、選択されるパラメータの引渡し方法がわかります。CREATE ルーチン文で指定するパラメータの引渡し方法は、Ada コンパイラで選択した方法と一致していなければなりません。
- 一覧の引渡し方法に加えて、MECHANISM プラグマを使用すると、Ada コンパイラで、エクスポート・プロシージャの引渡し方法を指定できます。

- Ada 関数以外にも、最初の引数が OUT モードになっている Ada プロシージャは、Ada 関数として扱うことができます。このようなプロシージャは、値付きのプロシージャとしてエクスポートする必要があります。

14.12.2 C での外部ルーチンの使用方法

C 言語で外部ルーチンを使用する場合には、次のガイドラインに従ってください。

- 次の SQL データ型は、C に直接対応しています。TINYINT、SMALLINT、INTEGER、BIGINT（OpenVMS Alpha および Compaq Tru64 UNIX のみ）、REAL、DOUBLE PRECISION、CHAR(n)、VARCHAR(n)
- パラメータの引渡し方法は、パラメータのデータ型によって異なります。
 - 数値のデータ型は、VALUE または REFERENCE で渡します。
 - 文字列のデータ型は、REFERENCE で渡します。
- 外部関数の戻り値の引渡し方法は、戻り値のデータ型によって異なります。
 - 数値のデータ型は、VALUE または REFERENCE で返されます。
 - 文字列のデータ型は、REFERENCE で返されます。
- CHAR(n) および VARCHAR(n) の文字列を REFERENCE で渡すと、SQL は、入力パラメータの場合は NULL で終了するテキストに変換し、出力パラメータの場合は NULL で終了するテキストから CHAR(n) または VARCHAR(n) に変換します。
- 次の SQL 文は、C 言語で記述した外部関数を指す外部関数の定義を作成します。

```
CREATE FUNCTION CONCAT_STRING (IN CHAR(20), IN CHAR(20)) RETURNS CHAR(40);
EXTERNAL LOCATION 'concat_location.exe' LANGUAGE C
GENERAL PARAMETER STYLE;
```

C 関数では、戻り値は、最初の引数として記述します。

```
void concat_string (char *result_str, char *first_input, char *second_input);
```

VAX C (V3.2-044) などの一部のコンパイラは、アドレスを R0 に返すことができるので、戻り値を最初のパラメータとして扱いません。次の例では、C ルーチンを、この条件に合うように修正して、書き直す必要があります。次に例を示します。

```
char *concat_string (char *first_input, char *second_input); ◆
```

- 固定長と可変長の文字列は、DESCRIPTOR（OpenVMS）または LENGTH で渡せますが、結果のデータ型は、C 言語のデータ型ではありません。

OpenVMS
VAX

14.12.3 COBOL での外部ルーチンの使用方法

COBOL 言語で外部ルーチンを使用する場合には、次のガイドラインに従ってください。

- 次の SQL データ型は、COBOL に直接対応しています。TINYINT、SMALLINT、INTEGER、BIGINT、REAL、CHAR(n)、VARCHAR(n)
- COBOL での数値型は、COMP (TINYINT、SMALLINT、INTEGER、BIGINT) と COMP-1 (REAL) です。COMP-2 (D 倍精度浮動小数点) はサポートされていません。
- パラメータは、データ型に関係なく、すべて REFERENCE で渡さなければなりません。
- 外部関数の戻り値の引渡し方法は、戻り値のデータ型によって異なります。
 - 数値のデータ型は、VALUE で返されます。
 - 文字列のデータ型は、REFERENCE で返されます。

14.12.4 FORTRAN での外部ルーチンの使用方法

FORTRAN 言語で外部ルーチンを使用する場合には、次のガイドラインに従ってください。

- 次の SQL データ型は、FORTRAN に直接対応しています。TINYINT、SMALLINT、INTEGER、BIGINT (OpenVMS Alpha および Compaq Tru64 UNIX のみ)、REAL、DOUBLE PRECISION、CHAR(n)、VARCHAR(n)
- パラメータの引渡し方法は、パラメータのデータ型によって異なります。
 - 数値のデータ型は、REFERENCE で渡します。
 - 文字列は LENGTH、または OpenVMS では DESCRIPTOR で渡します。
- 外部関数の戻り値の引渡し方法は、戻り値のデータ型によって異なります。
 - 数値のデータ型は、VALUE で返されます。
 - 文字列のデータ型は LENGTH、または OpenVMS では DESCRIPTOR で返されます。
- 固定長と可変長の文字列は、REFERENCE で渡せますが、結果のデータ型は、FORTRAN 言語のデータ型ではありません。

14.12.5 Pascal での外部ルーチンの使用方法

Pascal 言語で外部ルーチンを使用する場合には、次のガイドラインに従ってください。

- 次の SQL データ型は、Pascal に直接対応しています。SMALLINT、INTEGER、REAL、DOUBLE PRECISION、CHAR(n)、VARCHAR(n)
- パラメータは、データ型に関係なく、すべて REFERENCE で渡すことができます。

- 文字列を REFERENCE で渡す場合は、次のように宣言してください。

```
VAR name : PACKED ARRAY [1..n] OF CHAR
```

- 固定長の文字列は LENGTH、または OpenVMS では DESCRIPTOR で渡します。そのためには、次の例のように、結果の引数を宣言する必要があります。

```
[CLASS_S] PACKED ARRAY [LOW..HIGH :INTEGER] OF CHAR
```

- 外部関数の戻り値の引渡し方法は、戻り値のデータ型によって異なります。
 - 数値のデータ型は、VALUE で返されます。
 - 文字列のデータ型は、REFERENCE で返されます。

次の例は、CONCAT_STRING という名前の外部関数の定義です。

```
CREATE FUNCTION CONCAT_STRING (IN CHAR(20) BY REFERENCE,
                               IN CHAR(20) BY REFERENCE)
  RETURNS CHAR(40);
  EXTERNAL LOCATION 'CONCAT_LOCATION' LANGUAGE Pascal
  GENERAL PARAMETER STYLE;
```

Pascal では、次のように外部関数を指定してください。

```
function concat_string (var input1_str : packed array[1..20] of char;
                        var input2_str : packed array[1..20] of char)
  : packed array [1..40];
```

14.13 通知ルーチンの使用方法

変数の初期化やデータベースへの接続や切断など、初期化やクリーンアップ操作には通知ルーチンを使用できます。通知ルーチンを使用すると、データベース関連のイベント情報を、外部ルーチンの本体と共有することもできます。たとえば、外部ルーチンの本体がデータベース以外のファイルを変更する場合、この変更内容のジャーナルを管理することによって、コール側のトランザクション・スコープを外部ルーチンの本体の関数にまで拡張できます。

通知ルーチンは、外部ルーチンを含む共有イメージまたは共有オブジェクトに格納してください。

NOTIFY 句を使用して、次のイベントを指定できます。

- BIND--- 外部ルーチンがアクティブ化および非アクティブ化されたときに、通知ルーチンをコールします。
- CONNECT--- 外部ルーチンを含むデータベースに接続および切断したときに、通知ルーチンをコールします。
- TRANSACTION--- トランザクションが開始および終了したときに、通知ルーチンをコールします。

Notify 句を使用すると、Oracle Rdb は、選択したイベントの開始時（ルーチン本体を次に実行するまで、キューに格納）と、イベントの終了時（イベントが完了する直前。開始イベントの通知が発生したという前提）に、通知エントリ・ポイントをコールします。

通知ルーチンには、4 つの 32 ビットの引数を参照で渡します。次の表のように、最初の引数は通知のタイプを示し、残り 3 つの引数は、将来的な用途のために予約されています。

値	意味	定数
1	ルーチンのアクティブ化	RDB\$K_RTX_NOTIFY_ACTV_START
2	ルーチンの非アクティブ化	RDB\$K_RTX_NOTIFY_ACTV_END
3	データベースの接続	RDB\$K_RTX_NOTIFY_CONN_START
4	データベースの切断	RDB\$K_RTX_NOTIFY_CONN_END
5	トランザクションの開始	RDB\$K_RTX_NOTIFY_TRAN_START
6	トランザクションのコミット	RDB\$K_RTX_NOTIFY_TRAN_COMMIT
7	トランザクションのロールバック	RDB\$K_RTX_NOTIFY_TRAN_ROLLBACK

通知イベントは、固有のスコープ階層を持っています。BIND イベント（アクティブ化と非アクティブ化）は最上位のスコープであり、TRANSACTION イベントは、最下位のスコープです。3 つのイベントすべてを通知として選択し、トランザクションがアクティブな間に少なくとも 1 回外部ルーチンを起動する場合、ルーチンのバインド、データベースの接続、トランザクションの開始、トランザクションの終了、データベースの切断、ルーチンのバインド解除の合計 6 回以上、外部ルーチンがコールされます。

BIND SCOPE 句で指定したバインド・スコープにより、受け取る通知イベントを指定できます。たとえば、バインド・スコープが TRANSACTION の場合、データベース切断の通知は受け取れません。

通知イベントと通知ルーチンの使用は、データベース・リカバリ・プロセスにはまったくリンクしていません。

14.14 外部ルーチンでの例外処理

外部ルーチンでは、外部ルーチン名で識別されるルーチンまたは通知ルーチンが例外を生成し、ルーチンのコール側に返されます。

スコープ・イベントの終了が必要な通知ルーチンで例外が発生すると、対応付けられている SQL 文は失敗します。ただし、通知イベントは完了したと見なされるので、もう一度 SQL 文を実行しても、通知ルーチンの例外は発生しません。

Oracle Rdb は、外部ルーチンの起動時に、次のような例外条件を検出できます。

- 次のような場合に発生する共有可能イメージまたはオブジェクトの動的アクティブ化の失敗
 - 共有可能イメージまたはオブジェクトのロケーション指定に誤りがある
 - 共有可能イメージまたはオブジェクトにコンパイルの警告がある
 - ルーチンのエントリ・ポイントを検出できない
- 次のような場合に発生する、外部ルーチンの実行の失敗
 - 外部ルーチンの引数の 1 つが NULL
Oracle Rdb は、要求を強制終了し、メッセージを表示します。
 - 0 での除算など、不正な命令を実行
Oracle Rdb は、要求を強制終了し、メッセージを表示します。
 - 外部ルーチンが無限ループに入る
Oracle Rdb は、このような要求の強制終了はできません。

14.15 外部ルーチンの制限について

Oracle Rdb は、次のような処理の実行は不可能であり、サポートしません。

- エグゼキュータ・マネージャの機能を禁止する外部ルーチン処理。これには、SYS\$EXIT() または exit(), SYS\$DELPRC() または kill() など、エグゼキュータ・プロセスに影響を与えるシステム・コールが含まれます。
- 変換されたイメージである OpenVMS Alpha ルーチンの直接コール。
このような場合は、ジャケット・ルーチンを作成し、変換されたイメージでルーチンを起動します。次に、/TIE 修飾子でジャケット・ルーチンをコンパイルし、/NONATIVE_ONLY 修飾子でロードした共有可能イメージにジャケット・ルーチンを入れ、ジャケット・ルーチンと共有可能イメージをルーチン定義で参照します。◆

データベースをコールする外部関数の作成では、次のガイドラインに従ってください。

- データ定義は使用できません。
- リモート・データベースにはアクセスできません。
- 分散トランザクションは使用できません。
- SQL 文で使用するために、データベースまたはトランザクション・ハンドルをルーチンに渡すことはできません。これを行うと、例外が発生するか、データベース・サーバーで無限の待機状態が発生します。
- クエリー・オプティマイザは、SQL 統計関数や、EXISTS や IN などの一部の述語を、実行回数の少ない外部レベルに移動しようとします。このような最適化は、統計関数や述語に VARIANT で宣言した外部関数が存在するかどうかに関係なく実行されます。

14.16 外部ルーチンのトラブルシューティング

この項では、外部ルーチンの実行で発生する可能性のある問題についてまとめ、さらにその原因と解決策を示します。

外部ルーチンでは、次のような問題が発生することがあります。

- 予期しない結果
通常の場合、不良なパラメータ・データが原因です。このエラーを修正するには、次の内容を確認してください。
 - すべての入力パラメータは、ルーチンをコールする前に値が定義されている。
 - すべての出力パラメータと関数の結果には、ルーチンで定義した値が与えられている。
 - 引数は、宣言したパラメータのデータ型や引渡し方法に合致した方法で使用している。宣言した言語で特別なパラメータの引渡し方法がある場合は、これに注意する。
- アクセス違反
一般的に、引数の使用方法が、宣言したパラメータのデータ型や引渡し方法に適合していないことが原因です。
データベース環境の外でルーチンをテストしてください。データ型やパラメータの引渡し方法での問題を検出するには、別のソース・モジュールでテスト・プログラムを記述し、予想されるデータを外部ルーチンに渡します。コンパイラは、ソース間のルーチン・コールをグローバルに最適化するため、問題がわかりにくくなることがありますが、別のソース・モジュールを使用することによって、問題を的確に特定できます。
- RTNSBC_INITERR または RTNSBC_TASKERR の例外、理由コード 1 ～ 9
一般的に、サーバー・サイト・バインディングを持つルーチンに関連するクォータの問題が原因です。
ユーザーに必要なクォータや権限を割り当て、ルーチンのエグゼキュータ・プロセスの作成、共有メモリーの作成、OpenVMS ではロックの確立やメールボックスの作成が可能な状態にしてください。
- 例外 RTNSBC_TASKERR、理由コード 10 ～ 19
一般的に、サーバー・サイト・バインディングで宣言したルーチンのパラメータに、十分な共有メモリーが割り当てられていないことが原因です。さらに、ルーチンがデータベース操作を実行する場合、データベース操作に必要な共有メモリーの、パラメータに対する割当てが不足していることも考えられます。
論理名 RDMS\$RTX_SHRMEM_PAGE_CNT または構成パラメータ
RDB_RTX_SHRMEM_PAGE_CNT で定義した共有メモリー・ページ・カウントを増やしてください。
共有メモリー・サイズの変更では、実行制御領域に 2 ページ、サーバーからエグゼキュータ（コールアウト）の各レベルに 2 ページを確保し、コールアウトとデータベース操作（コールバック）の引数リスト、記述子、すべてのアクティブなコールア

ウトとコールバックの引数データなどを格納できるだけのページ数を十分に確保してください。

- 例外 INVRTNUSE、イメージがアクティブ化されていない
サーバー・サイト・バインディングのルーチンで発生しやすい問題であり、ファイル指定やロケーション句が、未定義の OpenVMS の論理名を参照しているか、カレント・ディレクトリがユーザーのログイン・ディレクトリ以外を想定していることが原因です。見つからなかったイメージ・ファイルの指定での、もう 1 つの例外を調べてください。
- 例外またはバグチェック・ダンプが報告するロックの競合やデッドロック
外部ルーチンが、ユーザー・アプリケーションやその他の外部ルーチンがすでに使用しているデータベースに接続すると、ロックの競合が発生します。多くの場合、このようなロックの競合は、内部的に解決されます。ただし、競合やデッドロックが解決不可能だと、致命的なエラーを返します。また、重大なロックの競合エラーが発生する場合もあり、これは内部エラーのように見えるので、バグチェック・ダンプが作成されます。アプリケーションや外部ルーチンの調整によってロックの競合を回避する方法については、『Oracle Rdb7 Guide to Database Performance and Tuning』を参照してください。

14.17 外部ルーチンの移植性と効率の向上

Oracle Rdb では、外部ルーチンを記述する際、次のような処理は行わないでください。

- プラットフォームに依存するコードでルーチンを記述し、複数のオペレーティング・システムで使用する。論理名、構成パラメータ、環境変数、対話型 I/O など。
- システム・サービス・コールやその他の非同期的なイベント・サービス。
- 外部ルーチンが接続しているデータベースに対して、操作を実行する終了ハンドラ。
- 同じルーチンを異なる名前でも複数登録すること。
- ユーザーが同時に使用する複数のデータベースで、同じルーチンを登録すること。
- 外部ルーチンを使用する再帰。

データベースでのルーチン定義には、共有可能イメージや共有オブジェクトのロケーションやパラメータの引渡し方法などの情報が格納されており、この情報はプラットフォーム固有であることが多いので、通常の場合、ルーチン定義に移植性はありません。

第V部

プログラムのコンテキスト

第5部では、プログラムが動作する環境を説明します。

- データベース・コンテキスト
- トランザクション・コンテキスト
- SQL 接続

データベースへの接続

SQL 文を実行する前に、その文を実行するデータベース・コンテキストに関する情報を提供する必要があります。この章では、ユーザーが記述する SQL 文に対してデータベース・コンテキストを正しく宣言する方法を説明します。この章では、次の方法について解説します。

- データベースの指定と接続
- リモート・データベースの指定と接続
- 分散トランザクションにおけるデータベースへの接続
- エイリアスによる複数データベースへの接続
- データベースの接続解除

15.1 データベースの指定と接続

データベース・コンテキストによって、アクセスするデータベースを識別します。ユーザー・プログラムは、データベースに接続してからでないと、SQL 文を実行できません。

データベースに接続する作業の大部分は、ユーザー・プロセスで使用するメモリーにデータベース（データベース定義のセット）をロードすることです。データベースへの接続時に、SQL がデータベース定義にアクセスする方法を指定します。データベース定義には、リポジトリを介して、またはデータベース・ファイルから直接アクセスできます。

15.1.1 データベース接続に対するファイルまたはリポジトリ・アクセスの指定

作業対象のデータベースは、ファイル名とリポジトリ・パス名の 2 通りの方法で指定できます。

- FILENAME file-specification
file-specification は、データベース定義のソースを指定します。Oracle Rdb データベースの絶対ファイル指定または相対ファイル指定を入力します。

- **PATHNAME path-name**

path-name は、データベース定義が格納されたりポジトリ内のノードを表す文字列です。プログラム内では、論理名またはプログラム・パラメータを使用して、この値を間接的に提供する必要があります。

ファイル名では常にデータベースにアクセスできますが、リポジトリ・パス名を使用してデータベースにアクセスできるのは、データベース定義がリポジトリとデータベース・ファイルの両方に格納されている場合に限りです。

15.1.2 データベース名の指定

特定のデータベースに対して作業することを SQL に通知するには、通常は対話型 SQL で ATTACH 文を使用するか、SQL モジュール言語とプリコンパイル SQL で DECLARE ALIAS 文を使用します。次のいずれかの方法を使用してください。

- ATTACH 文か DECLARE ALIAS 文にデータベースのファイル指定か、リポジトリのパス名を指定して入力します。
- ATTACH 文か DECLARE ALIAS 文を入力し、論理名か構成パラメータを使用してデータベースのファイル名かリポジトリ・パス名を参照します。
論理名と構成パラメータを使用すると、データベースやリポジトリを他のディスクやノードに移行する場合に、プログラムの保守が容易になります。構成パラメータは、Compaq Tru64 UNIX 上の .dbsrc ファイルで指定します。
- ファイル名かリポジトリ・パス名を含む変数を渡します。
- 論理名 SQL\$DATABASE または構成パラメータ SQL_DATABASE を定義して、データベースのファイル仕様を指定します。
SQL コンテキスト・ファイル内やプログラム内でデータベースへのエイリアスや接続が明示的に宣言されていない場合は、SQL プリコンパイラ、SQL モジュール・プロセッサ、データベース・ランタイム・システムによって、SQL\$DATABASE または SQL_DATABASE を変換します。
SQL\$DATABASE または SQL_DATABASE を使用してリポジトリ・アクセスを指定することはできません。したがって、CREATE 文、ALTER 文、DROP 文のいずれかに DICTIONARY IS REQUIRED 句を指定して実行する場合は、SQL\$DATABASE か SQL_DATABASE を使用してデータベースを接続することはできません。

ホスト言語プログラム内では、論理名またはプログラム・パラメータを使用してファイル指定を間接的に提供する必要があります。様々なノード上で実行するプログラムでは、ユーザーに対してファイル指定値の入力を要求し、プログラム・パラメータを介して SQL に情報を提供することもできます。

『Oracle Rdb7 Introduction to SQL』では、対話型 SQL ではどのようにデータベースに接続するか詳しく説明しています。

ATTACH 文や DECLARE ALIAS 文を使用して、どのデータベースに接続するのか、またリポジトリ・アクセスを要求するかどうかの情報を SQL に通知しても、Oracle Rdb は、その情報が提供された時点でデータベースへの接続またはリポジトリへのアクセスが可能な場合とそうでない場合があります。Oracle Rdb がデータベースに接続するタイミングは、ユーザーが作業する環境やデータベース情報を提供する方法によって変わります。SQL プログラムでは次のようになります。

- 通常、Oracle Rdb は実行可能な最初の SQL 文を処理するときに初めてデータベースに接続します。
- ATTACH 文か CONNECT TO 文を使用すると、Oracle Rdb はただちにデータベースに接続します。
- プログラムで複数のデータベースを宣言すると、Oracle Rdb は（実行可能な最初の SQL 文を処理するときに）宣言されたすべてのデータベースに同時に接続します。このことは、プログラムで開始する最初のトランザクションが何個のデータベースにアクセスするかには関係ありません。

SQL で最初にデータベース接続が発生する文では、データベース接続障害を処理する必要があります。SQL プログラムでは、トランザクション開始時にデータベース接続が発生します。プログラムで最初にどんな操作を実行するかにより、ALTER、CREATE、DELETE、DROP、GRANT、INSERT、OPEN、REVOKE、SELECT、SET TRANSACTION、UPDATE のいずれかの文を使用してデータベースに接続します。

15.1.3 様々なデータベースのコンパイル時と実行時の指定

この項では、プログラム開発サイクルのいくつかの時点で、なぜパス名またはファイル名を使い分け、データベースへ接続する必要があるかを説明します。

データベースにアクセスするプログラムの作成では、コンパイル時に使用するデータベース定義と実行時に使用するデータベース定義を1つずつ指定できます。次の例は、2つの異なるソースからのデータベースを宣言しています。

```
EXEC SQL
  DECLARE LOCAL personnel ALIAS
    COMPILETIME PATHNAME PERS_CDD
    RUNTIME FILENAME PERS;
```

コンパイル時には、プリコンパイラがリポジトリ内のデータベース定義を使用します。DECLARE ALIAS 文の PATHNAME 句は、論理名 PERS_CDD を使用してリポジトリを指します。

実行時には、SQL がデータベース・ファイル内のデータベース定義を使用します。DECLARE ALIAS 文の FILENAME 句は、論理名または構成パラメータ PERS を使用して personnel データベースを指します。

実行時オプションを指定しないと、SQL はコンパイル時にリポジトリから抽出したファイル名か COMPILETIME FILENAME 句で指定したファイル名を使用します。

指定するソースは、ファイル指定かリポジトリ・パス名です。しかし、コンパイル時にリポジトリ・パス名によるアクセスを指定し、RUNTIME 句を指定しないと、SQL は、コンパイル時にリポジトリから抽出したファイル名を使用します。

プログラムがデータベース定義でなくデータのみを操作または更新する場合は、ファイル名アクセスが適しています。しかし、データベース定義がリポジトリに格納されており、プログラムでデータ定義を変更する場合は、必ず PATHNAME 修飾子を指定してください。しかし、データベース定義がリポジトリに格納されており、プログラムでデータ定義を変更する場合は、必ず PATHNAME 修飾子を指定してください。他のユーザーがプログラムでこれらのリポジトリ定義をパラメータ宣言として使用する場合は、このことが特に重要になります。

コンパイル時と実行時で別のデータベースを指定するのは、次の理由によります。

- 実行時にユーザーがデータベースを指定できるようにするため
実行時にファイル名またはパス名の値をプログラム・パラメータに渡す場合は、プログラムで明示的に指定する以外のなんらかの方法でこの値を取得する必要があります。コンパイル時に、両方の SQL プロセッサは、COMPILETIME 句内のリテラル、論理名、構成パラメータを処理してリポジトリまたはデータベース・ファイルに必要なデータベース定義を検出します。しかし、SQL は、プログラムを実行するまで RUNTIME 句のパラメータを評価しません。プログラムを実行すると、SQL は RUNTIME 句で指定されたプログラム・パラメータを評価して、適切なデータベースに接続します。
- リモート・データベースを処理する場合に、SQL プリコンパイラまたは SQL モジュール・プロセッサのパフォーマンスを改善するためのリモート・データベースへの接続については、15.2 項 (15-4) を参照してください。

15.2 リモート・ノードでのデータベースの指定

他のシステム上にある Oracle Rdb データベースにアクセスするには、両方のシステムで同じ通信プロトコルを使用する必要があります。つまり、両方のシステムで、TCP/IP または DECnet を使用してください。さらに、リモート・サーバー・アカウントとして使用するアカウントを正しく設定する必要があります。

ネットワーク・プロトコルの内容と、リモート・サーバー・アカウントの設定方法は、『Oracle Rdb7 Installation and Configuration Guide』を参照してください。

リモート・システム上のデータベースを指定するには、FILENAME 句でノード指定を行います。たとえば、リモートの OpenVMS システム上のデータベースに接続するには、次の SQL 文を使用します。

```
SQL> ATTACH 'FILENAME speedy::disk3:[dept3]personnel';
```

ローカル・データベースと同様に、ファイル指定のすべてまたは一部として、論理名や構成パラメータを指定できます。

リモートの Oracle Rdb データベースへのアクセスには、Oracle Rdb リモート・サーバー・アカウントを使用します。**リモート・サーバー・アカウント**は、リモート・ノード上で Oracle Rdb を実行するために、Oracle Rdb がログインするリモート・ノード上のアカウントです。

また、リモート・システム上のアカウントには、有効なユーザー名とパスワードを暗黙的または明示的に提供する必要があります。このよう**リモート・ユーザー認証**は、リモート・ノード上のユーザー情報であり、これによって、Oracle Rdb はユーザーのデータベース・アクセス権限を判断します。

リモートシステム上のユーザー認証用に、ユーザー名とパスワードを提供する方法には、次のようなものがあります。

- 次のいずれかの方法で明示的に指定
 - ATTACH 文や DECLARE ALIAS 文などの SQL 文で、USER 句および USING 句を使用します。詳細は、15.2.1 項 (15-6) を参照してください。
 - モジュール言語やプリコンパイラのコマンド行で、修飾子を使用します。修飾子は、DECLARE ALIAS 文の USER DEFAULT 句および USING DEFAULT 句と組み合わせて使用します。詳細は、15.2.2 項 (15-7) を参照してください。
 - ローカル・システム上のクライアント構成ファイルを使用します。構成ファイルは、Compaq Tru64 UNIX では .dbsrc ファイル、OpenVMS では RDB\$CLIENT_DEFAULTS.DAT ファイルです。詳細は、15.2.3 項 (15-8) を参照してください。
- 暗黙的な指定
Compaq Tru64 UNIX システムから Compaq Tru64 UNIX システムのデータベースに接続する場合などのように、ユーザー名とパスワードが両方のシステム上に存在するならば、ユーザー名やパスワードを明示的に指定する必要はありません。Oracle Rdb は、ユーザーがデータベースに接続するたびに、暗黙的なユーザー認証を行います。表 15-1 (15-6) は、どのような場合に、Oracle Rdb が暗黙的にユーザー認証を行うかを示しています。

リモート・サーバー・アカウントへのアクセスでは、次のいずれかの方法を使用します。

OpenVMS
VAX ≡≡≡ OpenVMS
Alpha ≡≡≡

- リモート・ノード上に設定したプロキシ・アカウントを使用します (OpenVMS で推奨)。15.2.4 項 (15-8) を参照してください。◆

OpenVMS
VAX ≡≡≡ OpenVMS
Alpha ≡≡≡

- リモート・アクセス用の OpenVMS 上の RDB\$REMOTE デフォルト・アカウントを使用します。15.2.5 項 (15-9) を参照してください。◆

Compaq Tru64 UNIX
≡≡≡≡≡

- Compaq Tru64 UNIX の dbsmgr アカウントを使用して、インストール中に dbsmgr アカウントを自動設定します。インストール中やインストールの後にアカウントを変更する必要はありません。または、リモート・アクセス中にアカウント名を指定する必要もありません。◆

リモート・データベースへの接続で利用できるオプションは、表 15-1 (15-6) で示すように、オペレーティング・システムやネットワーク・プロトコルによって異なります。

表 15-1 リモート・アクセスのオプション

	OpenVMS から OpenVMS	OpenVMS から UNIX	UNIX から UNIX	UNIX から OpenVMS
リモート・ユーザー認証				
暗黙的	T* ¹ 、D* ¹	N/A	T* ¹	N/A
USER/USING 句	T、D	T、D	T、D	T、D
SQL コマンド行修飾子	T、D	T、D	T、D	T、D
構成ファイル	T、D	T、D	T、D	T、D
リモート・サーバー・アカウント				
プロキシ・アカウント	D	N/A	N/A	D
RDB\$REMOTE[nn] アカウント	T、D	N/A	N/A	T、D
dbsmgr アカウント	N/A	T、D	T、D	N/A

*¹ ユーザー名とパスワードが同じノード上にある場合

ネットワーク・プロトコル

- T - TCP / IP
- D - DECnet
- N/A - 適用なし

次の項では、各オプションについて説明します。

参考: リモート・アカウントの設定の詳細は、『Oracle Rdb7 Installation and Configuration Guide』を参照してください。また、使用しているその他のデータベース・アクセス製品については、必要に応じて、インストールおよび構成ガイドを参照してください。

15.2.1 リモート・ユーザー認証での USER 句および USING 句の使用方法

リモート・ノード上のデータベースへのアクセスでは、データベースに接続する SQL 文で、ユーザー名とパスワード情報を明示的に指定することができます。

ATTACH 文または DECLARE ALIAS 文や、データベースに接続するその他の SQL 文（ALTER DATABASE 文や CONNECT 文など）では、USER 句と USING 句を使用して、ユーザー名とパスワードを埋め込みます。

たとえば、リモートの Compaq Tru64 UNIX ノード上にある mf_personnel データベースに接続するには、ATTACH 文の USER 句と USING 句を使用できます。

```
SQL> ATTACH 'FILENAME osfrem::/usr/users/heleng/mf_personnel
1>          USER 'heleng' USING 'MYpassword''';
```

ユーザー名とパスワードは一重引用符 (') で囲む必要がありますが、この例では、リテラルに引用符が使用されているので、ユーザー名とパスワードを 2 つの一重引用符で囲んでください。

ユーザー名とパスワードをプログラムのソース・コードで記述しないでよいように、両方の句に DEFAULT キーワードを使用できます。これは、SQL モジュール・プロセッサと SQL プリコンパイラのコマンド行修飾子を使用して、ユーザー名とパスワードを受け渡してください。

SQL 文で USER 句と USING 句を指定しないと、Oracle Rdb は、構成ファイル内の情報を使用します。

15.2.2 コマンド行修飾子によるリモート・ユーザー認証

ユーザー名とパスワードの指定には、次のようなコマンド行修飾子を使用できます。

Compaq Tru64 UNIX

- Compaq Tru64 UNIX :
 - SQL モジュール・プロセッサには、-user および -pass オプション
 - SQL プリコンパイラには、-s -user および -s -pass オプション ◆

OpenVMS VAX OpenVMS Alpha

- OpenVMS :
 - SQL モジュール・プロセッサでは、USER_DEFAULT および PASSWORD_DEFAULT 修飾子
 - SQL プリコンパイラでは、SQL_OPTIONS 修飾子に USER_DEFAULT および PASSWORD_DEFAULT オプション ◆

この修飾子は、DECLARE ALIAS 文の USER DEFAULT 句および PASSWORD DEFAULT 句と一緒に使用してください。修飾子は、コンパイル時のユーザー名とパスワードをプログラムに渡します。

次の例は、上記の修飾子を使用して、Compaq Tru64 UNIX 上の SQL モジュールを処理する方法を示しています。

```
$ sqlmod myprog -user heleng -pass MYpassword
```

15.2.3 構成ファイルによるリモート・ユーザー認証

SQL_USERNAME および SQL_PASSWORD 構成パラメータを使用して、ローカルの構成ファイルに格納されているリモート・システムのユーザー名とパスワードを明示的に提供できます。

Compaq Tru64 UNIX

OpenVMS VAX OpenVMS Alpha

Compaq Tru64 UNIX では、ローカルの .dbsrc 構成ファイルの情報を提供します。◆

OpenVMS では、ローカルの構成ファイル RDB\$CLIENT_DEFAULTS.DAT の情報を提供します。◆

たとえば、リモートの Compaq Tru64 UNIX ノード上のユーザー名が "heleng"、パスワードが "MYpassword" である場合、次の例のように、これをローカル・ノード上の構成ファイルに追加します。

SQL_USERNAME	heleng
SQL_PASSWORD	MYpassword

15.2.4 リモート・サーバー・アカウントとしてのプロキシ・アカウントの使用

OpenVMS VAX OpenVMS Alpha

リモートの OpenVMS データベースへのアクセスでは、Oracle Rdb では、**プロキシ・アカウント**の使用を推奨しています。

プロキシ・アカウントは、リモート・ノードにプライベート・アカウントを持たないユーザーに対して、リモート・ノードへのアクセス権限を与えます。プロキシ・アカウントを使用すると、一部の選択されたユーザーはネットワークにアクセスできますが、ネットワークへのログイン要求に、ユーザー名とパスワードを提供するアクセス制御文字列を含める必要はありません。したがって、プロキシ・アカウントには、次のようなセキュリティ上の利点があります。

- 要求を発行した端末上で、パスワードがエコーされません。
- システム間でのパスワードの受渡しがないので、暗号化されていない形式でパスワードが傍受されることがありません。
- ユーザーは、リモート・アクセス手順を実行するコマンド・ファイルにパスワードを格納しなくなります。

システムのセキュリティを維持するために、プロキシ・アカウントには、権限を付与するべきではありません。プロキシ・アカウントを設定すると、ローカル・アカウントの特権を継承します。

プロキシ・アカウントを使用してリモート・データベースに接続するには、ATTACH 文でユーザー名を指定してください。たとえば、NODEB 上の JONES アカウントをリモート・サーバー・アカウントとして使用するには、次のように指定します。

```
SQL> ATTACH 'FILENAME NODEB"JONES"::USER2:[JONES] PERSONNEL';
```

プロキシ・アカウントには、データベース・アクセスに必要なプロセス・クォータの値と権限を割り当ててください。『Oracle Rdb7 Installation and Configuration Guide』では、OpenVMS ユーザーに通常割り当てられる値が、データベース操作には不十分な場合に、推奨されるクォータの最小値を掲載しています。さらに、リモート・データベースの管理担当者は、プロキシ・アカウントによるデータベース・アクセスを実行するために、データベース保護を変更します。

プロキシ・アカウントに十分なプロセス・クォータが割り当てられていないと、プロセスは、リモート・システム上で必要なデータベース・イメージをアクティブ化できません。この場合、プロセスは、Oracle Rdb を使用できないというメッセージを受信します。プロキシ・アカウントによるアクセスができるようにリモート・データベースの保護を変更していないと、プロセスは、実行したい操作を許可するデータベース権限がない、というメッセージを受信します。

RDB\$REMOTE デフォルト・アカウントで指定した以外の Oracle Rdb バージョンにアクセスするには、プロキシ・アカウントの LOGIN.COM で、論理名 RDBSERVER と RDMS\$VERSION_VARIANT を定義してください。たとえば、Oracle Rdb バージョン 6.1 がデフォルトのバージョンであり、アクセスしたいのはバージョン 7.0 だとすると、プロキシ・アカウントの LOGIN.COM に次の内容を追加します。

```
$ DEFINE RDBSERVER SYS$SYSTEM:RDBSERVER70.EXE
$ DEFINE RDMS$VERSION_VARIANT 70 ◆
```

15.2.5 リモート・サーバー・アカウントとしての RDB\$REMOTE アカウントの使用

OpenVMS VAX ≡≡≡ OpenVMS Alpha ≡≡≡

RDB\$REMOTE デフォルト・アカウントを使用してリモート・サーバーに接続するには、リモート・ノード上に RDB\$REMOTE アカウントを正しく設定する必要があります。アカウントの設定方法は、『Oracle Rdb7 Installation and Configuration Guide』を参照してください。次に、データベースが格納されているノード名、データベースのディレクトリとファイル指定を、次の例のように指定します。

```
SQL> ATTACH 'FILENAME speedy::disk3:[dept3]personnel';
```

ただし、RDB\$REMOTE デフォルト・アカウントで指定した以外の Oracle Rdb バージョンを使用する場合は、リモート・システム上で、対応するアカウントのユーザー名とパスワードを指定する必要があります。たとえば、リモート・ノード上の RDB\$REMOTE アカウントの設定は Oracle Rdb バージョン 6.1 ですが、アクセスしたいのはバージョン 7.0 だとすると、次のように、アクセス制御文字列で、RDB\$REMOTE70 アカウントとユーザー名を指定します。

```
SQL> ATTACH 'FILENAME SPEEDY"RDB$REMOTE70 password":DISK3:[DEPT3]PERSONNEL';
```

注意: TCP/IP ネットワーク・プロトコルでは、アクセス制御文字列は使用できません。使用できるのは、DECnet のみです。TCP/IP を使用する場合に、RDB\$REMOTE デフォルト・アカウント以外アカウントにアクセスする方法の詳細は、15.2.6 項 (15-10) を参照してください。

SQL 文でパスワードを指定しないですむように、選択したユーザー用に、リモート・ノードにプロキシ・アカウントを設定してください。プロキシ・アカウントを、バージョン専用の RDB\$REMOTE アカウントにマッピングします。たとえば、次の例では、ユーザー HELENG が、ノード FASTR からリモート・ノード SPEEDY のバージョン 7.0 データベースにアクセスできるように設定しています。

```
UAF> ADD/PROXY FASTR::HELENG RDB$REMOTE70/DEFAULT ◆
```

15.2.6 代替 UCX またはインターネット・サービスの使用方法

TCP/IP ネットワーク・プロトコルを使用する場合、代替サービスを使えば、リモート OpenVMX システム上で異なるバージョンの Oracle Rdb へのアクセスや、リモートの Compaq Tru64 UNIX システム上で異なるデフォルト構成を使用できます。代替サービスは、Compaq Tru64 UNIX システムから、OpenVMS システム上にあるバージョンが異なる Oracle Rdb にアクセスするときに特に便利です。

TCP/IP ネットワーク・プロトコルを使用してリモート・データベースにアクセスする場合、デフォルトでは、サービス RdbServer を使用します。

OpenVMS
VAX ≡≡≡ OpenVMS
Alpha ≡≡≡

OpenVMS では、サービスは、ユーザー名 RDB\$REMOTE をデフォルトで使用します。異なるユーザー名で新しい UCX サービスを作成できます。クライアントの構成ファイルに、SQL_ALTERNATE_SERVICE_NAME 構成パラメータを追加すると、Oracle Rdb で新しいサービスを使用します。たとえば、myservice という名前の新しいサービスを作成し、サービスの名前を RDB\$REMOTE61 に設定したとします。クライアント構成ファイルに次の行を追加すると、Oracle Rdb バージョン 6.1 を使用してデータベースにアクセスできます。

```
SQL_ALTERNATE_SERVICE_NAME myservice ◆
```

Compaq Tru64 UNIX
≡≡≡

Compaq Tru64 UNIX では、サービスはユーザー名 dbsmgr をデフォルトで使用します。ユーザー名は変更できませんが、代替サービスを作成することは可能です。構成パラメータ SQL_ALTERNATE_SERVICE_NAME をクライアントの構成ファイルに追加すると、Oracle Rdb は、新しいサービスを使用します。たとえば、myservice という名前の新しいサービスを作成し、サービスの名前を dbsmgr (指定可能な名前はこれのみです) に設定したとします。クライアント構成ファイルに、次の行を追加してください。

```
SQL_ALTERNATE_SERVICE_NAME myservice ◆
```

代替サービスの設定と構成ファイルの詳細は、『Oracle Rdb7 Installation and Configuration Guide』を参照してください。

15.3 非同期システム・トラップの回避

Oracle Rdb データベースにアクセスするアプリケーションでは、非同期システム・トラップ (AST) サービス・ルーチンを使用できません。このルーチンを使用した場合、Oracle Rdb ではアプリケーションの動作を保証できません。

いくつかの Oracle Rdb コンポーネントで AST サービスを使用していること、および AST サービス・ルーチンが同じモードかさらに上の権限をもつモードで実行されている場合は AST サービスを提供できないことから、アプリケーション内で AST サービス・ルーチンを使用すると、Oracle Rdb のコンポーネントに AST を提供できなくなる可能性があります。

15.4 分散トランザクションにおけるデータベース接続

分散トランザクションを使用してリモート・システム上のデータベースにアクセスすると、分散トランザクション固有の問題に直面することがあります。これ以降の各項では、このような問題を回避する方法を説明します。

15.4.1 分散トランザクションによる検出不可能なデッドロックの回避

分散トランザクションを使用してリモート・システム上のデータベースにアクセスすると、検出不可能なデッドロックが発生する可能性があります。**デッドロック**は、2 人のユーザーが共に必要な複数のリソースをロックしようとして、いずれのユーザーも、相手がトランザクションを終了するまで作業を継続できなくなった場合に発生します。同一ノードまたは同一クラスタ上でデッドロックが発生すると、ロック・マネージャはデッドロックを検出し、一方のユーザーにデッドロック・エラー状態を通知します。しかし、トランザクションがリモート・システム上のデータベースにアクセスする場合、ロック・マネージャはデッドロックを検出できません。

Oracle Rdb は、分散デッドロックを回避するために、トランザクションのロック解放待時間を設定する方法をいくつか準備しています。16.2.5 項 (16-12) では、これらのオプションの使用方法を説明します。

分散トランザクションのデッドロック回避の詳細は、『Oracle Rdb7 Guide to Distributed Transactions』を参照してください。

15.4.2 分散トランザクションにおける権限エラーの回避

リモート・ノード上のデータベースに接続する分散トランザクションを開始すると、Oracle Rdb はリモート・ノード上のアカウントに DISTRIBTRAN 権限があるかどうかを確認します。たとえば、リモート・ノード上のプロキシ・アカウントを使用する場合、そのプロキシ・アカウントにはデータベースに関する DISTRIBTRAN 権限が必要です。

DISTRIBTRAN 権限のないユーザーが分散トランザクションを開始すると、Oracle Rdb はエラーを返し、トランザクションは開始されません。このことは、SQL を使用する上で特に注

れてはならない重要な問題です。SQL では、複数のデータベースに接続するトランザクションを開始すると、デフォルトで分散トランザクションが開始されます。

データベース権限の詳細は、『Oracle Rdb7 Guide to Database Design and Definition』と『Oracle Rdb7 SQL Reference Manual』を参照してください。

15.5 エイリアスによる複数の接続

エイリアスは、データベースへの特定の接続を表すためにユーザーが指定する名前です。エイリアスは、ファイル名やリポジトリ・パス名とは異なり、データベースの宣言ごとに変化してもかまいません。言い換えれば、エイリアスは、セッションごとまたはプログラムごとにユーザーがデータベースに付ける固有の名前です。いったんデータベースに接続（ATTACH 文を使用）した後、またはエイリアスを宣言（DECLARE ALIAS 文を使用）した後には、それ以降のすべての文でデータベースをエイリアスで参照することになります。

次のデータベース接続で、SP はエイリアスを表します。

```
SQL> ATTACH 'ALIAS SP FILENAME personnel';
```

複数のデータベースに接続して、データベースごとに異なるエイリアスを指定することで、同時に複数のデータベースに対し作業できます。また、同じデータベースに対して複数の ATTACH 文を使用して、それぞれに異なるエイリアスを指定すれば、同じデータベースに対して複数の接続を保持できます。

プリコンパイル SQL では、いったんデータベースをエイリアスで識別すると、そのデータベースへのすべての参照をそのエイリアス名で修飾する必要があります。SQL プリコンパイラは、修飾されていない参照は RDB\$DBHANDLE で指定したデフォルトのデータベースと見なします。RDB\$DBHANDLE が未定義の場合はエラーが発生します。RDB\$DBHANDLE が定義済みの場合は、意図しないデータベースにアクセスすることになります。

SQL モジュール・プロセッサがサポートするプログラム内でエイリアスを取り扱う場合は、SQL モジュール・ヘッダーの ALIAS 句でデフォルトのデータベースに対するエイリアスを指定します。モジュール・ヘッダーでエイリアスを宣言した場合には、モジュール・プロシージャでは、データベース・エンティティへの情報を定義する必要はありません。

一般に、一度に 1 つのデータベースを対象とする場合は、データベースのエイリアスを明示的に指定する必要はありません。エイリアスを明示的に入力しないと、SQL プリコンパイラは、デフォルトのエイリアス RDB\$DBHANDLE を提供します。SQL モジュール言語では、デフォルトはモジュール・ヘッダーで指定されたエイリアスです。モジュール・ヘッダーにエイリアスが指定されていない場合、SQL はエイリアスとして認可識別子を割り当てます。認可識別子が割り当てられていない場合、SQL はデフォルトとして RDB\$DBHANDLE を提供します。

固有のエイリアスを指定しても、SQL はそのエイリアスをデフォルトとは見なしません。テーブル参照は、必ずエイリアスで修飾する必要があります（例 15-1 (15-13)）。

例 15-1 テーブル参照をエイリアスで修飾

```
SQL> ATTACH 'ALIAS SP FILENAME personnel';
SQL> SELECT JOB_CODE, JOB_TITLE, MINIMUM_SALARY, MAXIMUM_SALARY
1>      FROM JOBS LIMIT TO 5 ROWS;
%SQL-F-NODEFDB, There is no default database
SQL> --
SQL> SELECT JOB_CODE, JOB_TITLE, MINIMUM_SALARY, MAXIMUM_SALARY
      FROM SP.JOBS LIMIT TO 5 ROWS;;
```

JOB_CODE	JOB_TITLE	MINIMUM_SALARY	MAXIMUM_SALARY
APGM	Associate Programmer	\$15,000.00	\$24,000.00
CLRK	Clerk	\$12,000.00	\$20,000.00
ASCK	Assistant Clerk	\$7,000.00	\$15,000.00
DMGR	Department Manager	\$50,000.00	\$100,000.00
DSUP	Dept. Supervisor	\$36,000.00	\$60,000.00

```
5 rows selected
SQL>
```

1つのデータベースに接続する場合は、デフォルトのエイリアスを使用すると処理が簡単です。しかし、マルチデータベース環境では、すべてのデータベースに対してエイリアスを明示的に宣言し、固有の名前を付ける必要があります。複数のデータベースで作業する場合には、論理名 SQL\$DATABASE または構成パラメータ SQL_DATABASE による、データベースに対するデフォルト・エイリアスの自動的な宣言に依存しないでください。

例 15-2 (15-13) に、1つのデータベースに対してエイリアスを宣言し、別のデータベースにはデフォルトの RDB\$DBHANDLE を使用する SQL プリコンパイル・プログラムの一部を示します。

例 15-2 エイリアスの使用方法

```
.
.
.
EXEC SQL DECLARE ALIAS FILENAME personnel;
EXEC SQL DECLARE alias_1 ALIAS FILENAME mf_personnel;
EXEC SQL SELECT EMPLOYEE_ID, LAST_NAME
      INTO :employee_id1, :last_name1
      FROM EMPLOYEES WHERE EMPLOYEE_ID = '00301';
EXEC SQL INSERT INTO alias_1.EMPLOYEES
      (EMPLOYEE_ID, LAST_NAME)
      VALUES (:employee_id1, :last_name1 );
.
.
.
```

2つのデータベースを同時に使用するには、少なくともその一方にはエイリアスを使用して接続する必要があります（デフォルトのエイリアス RDB\$DBHANDLE で特定できるデータ

ベースは1つだけです)。すべてのテーブル参照は、データベース接続で指定したエイリアスで修飾する必要があります。

例 15-3 (15-14) に、同時に2つのデータベースに接続し、それぞれにエイリアスを宣言するプログラムの一部を示します。

例 15-3 2 つ以上のデータベースの処理

```

/*****
* This routine transfers an employee from the EAST database to the WEST
* database. It expects the distributed context structure as a parameter.
* The employee record is selected first, deleted from the EAST database
* and inserted into the WEST database.
*****/

.
.
.
EXEC SQL DECLARE east ALIAS FOR FILENAME 2pceast;
EXEC SQL DECLARE west ALIAS FOR FILENAME 2pcwest;
.
.
.
EXEC SQL USING CONTEXT :local_context
      SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
             ADDRESS_DATA_1, CITY, STATE, POSTAL_CODE, SEX, BIRTHDAY
      INTO
        :local_employee.emp_id, :local_employee.emp_last_name,
        :local_employee.emp_first_name, :local_employee.emp_middle_initial,
        :local_employee.emp_address1, :local_employee.emp_city,
        :local_employee.emp_state, :local_employee.emp_zip,
        :local_employee.emp_sex, :local_employee.emp_birth
      FROM east.EMPLOYEES
      WHERE east.EMPLOYEES.EMPLOYEE_ID = :emp_id;
printf("Deleting the record from the EAST database %n");
EXEC SQL USING CONTEXT :local_context
      DELETE FROM east.EMPLOYEES E
      WHERE E.EMPLOYEE_ID = :emp_id;
printf("%nInserting the employee record into WEST %n");
EXEC SQL USING CONTEXT :local_context
      INSERT INTO west.EMPLOYEES
        (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
         ADDRESS_DATA_1, CITY, STATE, POSTAL_CODE, SEX, BIRTHDAY)
      VALUES
        (:local_employee.emp_id, :local_employee.emp_last_name,
        :local_employee.emp_first_name,
        :local_employee.emp_middle_initial,
        :local_employee.emp_address1, :local_employee.emp_city,
        :local_employee.emp_state, :local_employee.emp_zip,

```



```
:local_employee.emp_sex, :local_employee.emp_birth);
```

```
·  
·  
·
```

例 15-3 (15-14) に示すように、同一のデータ操作で2つの異なるデータベースからのデータを組み合わせることはできません。ただし、データベースごとに異なるデータ操作を実行する限り、同一のトランザクションで2つのデータベースを処理することができます。この場合は、DECLARE TRANSACTION 文か SET TRANSACTION 文でエイリアスを指定します。DECLARE TRANSACTION 文と SET TRANSACTION 文の詳細は、16.2 項 (16-5) を参照してください。

15.6 データベースの接続解除

プログラムにおいて、Oracle Rdb は、次の場合にデータベース接続を解除されます。

- ユーザーが DISCONNECT 文を入力した場合
DISCONNECT 文には次の機能があります。
 - 実行状態のトランザクションをロールバックします。
 - DISCONNECT 文が発行されたモジュール内で宣言が適用されるすべてのデータベースから接続を解除します。GLOBAL 引数で宣言したデータベースも接続解除されます。
 - モジュールで動的 SQL を使用する場合、DISCONNECT 文が発行されると、そのモジュール内で動的に実行されていた SQL 文が解放され、そのモジュール内で動的に宣言されたエイリアスが切断され、さらに動的に実行された DECLARE TRANSACTION 文で指定した特性が取り消されます。
- プログラムが実行を停止した場合
プログラムが終了すると、DISCONNECT 文と同じ処理が行われます。

プログラムで新しく発行した ATTACH 文ですでに使用されているエイリアスを指定しても、前の接続が終了することはありませんが、エラー・メッセージが表示されます。

トランザクション・コンテキストの管理

この章では、トランザクションの詳細と、SQL プログラムを正しく効率的に実行できるように、トランザクションの特性を指定する方法について説明します。この章の各項では、次の内容について説明します。

- トランザクションの概念とトランザクションの特性
- トランザクションで必要となる特性を指定する方法
- トランザクションのスコープ
- 分散トランザクションを使用する基準
- データベース・リソースのロックを制御する方法
- 入力または出力を短縮するトランザクションの設計方法
- 制約の評価と評価のタイミングを制御する方法
- トランザクションのコミットとロールバック

16.1 トランザクションについて

Oracle Rdb データベースに含まれるデータへのアクセスは、**トランザクション**として発生します。Oracle Rdb は、トランザクション内のすべての SQL 文を、1 つの単位として扱います。つまり、単位のすべての文が完了するか、まったく完了しないかのいずれかになります。(ただし ATOMIC 複合文に含まれる文は、それだけ個別に失敗することがあります)。したがって、トランザクションを部分的に実行し、データベースが変更されることはありません。

たとえば、従業員が、給与の高い新しい仕事に転属になったとします。サンプルの `personnel` データベースでは、`JOB_HISTORY` テーブルと `SALARY_HISTORY` テーブルに新しく行を追加することになります。各テーブルを個別に更新する場合、トランザクションのすべての操作が完了する前に、エラーやハードウェア障害が発生すると、データベースでは、その従業員が 2 つの部門に所属して 2 種類の給与をもらっている状態になってしまうか、仕事は新しくなっても給与が古いままの状態になってしまい、データベースの一貫性が失われます。このような状況を回避するためには、すべての更新を 1 つのトランザクションで実行します。その後、エラーをモニターし、エラーが発生したら、全体の操作を明示的にロールバックします。

トランザクションの開始と終了は、プログラムで決定します。トランザクションの実行中、他のユーザーにどのようなデータベース・アクセスを許可するか、トランザクションにどのような特性を持たせるか、Oracle Rdb は、変更内容を永続的にするか (コミット)、終了時にデータベースから削除するか (ロールバック)、などを指定します。

16.1.1 トランザクションの特性

トランザクションの特性には、次のように様々な内容を指定できます。

- 1 つまたは複数のデータベースへのアクセス・タイプ
読取り専用、読取り / 書込み、バッチ更新などのデータベース・アクセスを要求できます。デフォルトは、すべての接続データベースに対する読取り / 書込みアクセスです。読取り専用トランザクションは、データを読み取りますが、変更はしません。古いデータの削除や更新、新しい行の挿入などはできません。読取り / 書込みトランザクションは、取出しや更新などを実行できます。ここで一覧するトランザクション特性の多くは、読取り / 書込みトランザクションのみに適用されます。
バッチ更新トランザクションは、汎用でない特殊用途のトランザクションです。バッチ更新トランザクションは、`Recovery-unit` ジャーナル・ファイルに書込みを行わないので、ロール・バックできません。したがって、バッチ更新トランザクションでエラーが発生すると、異常終了し、データは破損してしまいます。16.2.3 項 (16-8) では、バッチ更新トランザクションの利点と危険性について説明しています。
- アクセスするデータベース
トランザクションは、1 つ以上のデータベースに対応付けられています。アクセスするデータベースが 1 つのみの場合は、指定する必要はありません。同様に、同じ方法です

すべてのデータベースにアクセスする場合は、データベースを指定する必要はありません。同じ方法でアクセスする場合、SQL は、各データベースに、同じトランザクション特性を自動的に適用します。

異なるトランザクション特性を使用して、1つ以上のデータベースを処理する場合（たとえば、読取り専用と読取り / 書込みが混在する場合）、DECLARE ALIAS 文や ATTACH 文で、エイリアスを定義する必要があります。また、トランザクションの開始では、エイリアスを指定する必要があります。16.2.8 項 (16-21) では、エイリアスを使用してトランザクションを開始する方法を説明します。16.4 項 (16-24) では、一部のデータベースが異なるデータベース上にある場合について説明しています。

- 使用するテーブル
処理するテーブルの名前は、RESERVING 句で明示的に指定できます。RESERVING 句でテーブルのリストを指定すると、リストに含まれないテーブルにはアクセスできません。明示的にテーブルを指定しない場合、Oracle Rdb は、トランザクションの操作に必要なテーブルを自動的に予約します。
- ロック・タイプ
ロック・タイプによって、Oracle Rdb による各テーブルの使用方法がわかります。読取りアクセスでは、情報の取出しは可能ですが、変更、削除、挿入はできません。書込みアクセスでは、データの読取り、変更、追加、削除が可能です。データ定義アクセスでは、指定したテーブルのインデックスの作成と削除が可能です。その他の操作は実行しません。
デフォルトでは、Oracle Rdb は、トランザクションが操作を行うとき、必要に応じて行をロックします。
- 共有モード
共有モードは、トランザクションの実行中、他のユーザーが各テーブルに対して実行できるアクセスの内容を示します。トランザクションで使用しているテーブルに対して、他のトランザクションに共有アクセスを許可したり、読取りは許可するが書込みを禁止したり、排他的にすべて禁止できます。
- 待機オプション
待機オプションは、他のユーザーがテーブルや行のロックを解放するまで待機するか (WAIT)、ロックの競合メッセージをすぐに受け取るか (NOWAIT) を指定します。デフォルトは WAIT です。
- 分離レベル
分離レベルは、他のトランザクションが書き込んだデータを、どの程度参照できるかを指定します。
直列化可能な分離レベルでは、トランザクションの処理中にある行を何回か取り出すと、読取りのたびに同じバージョンのデータが取り出されます（取出しの間に、行は更新または削除されていないものとします）。あまり厳しくない分離レベルを使用すると、他のユーザーが変更したデータを変更でき、トランザクションで以前に参照した内容と変更内容が一致していなくても、参照が可能です。デフォルトは、直列化可能な分離レベルです。
分離レベルの選択の詳細は、16.2.6 項 (16-14) を参照してください。

- 制約を評価するタイミング
16.7 項 (16-39) を参照してください。

データベースにアクセスするプログラムを記述するときは、まず、実行するタスクを決めます。データベースに関するタスクには、次のようなものがあります。

- 1 つ以上のデータベースのテーブルからのデータ読取り
- 既存のテーブル行の変更
- データベース定義の変更

タスク間に依存関係がある場合は、同じトランザクションで実行してください。依存関係がないタスクは、別のトランザクションで実行してください。

16.1.2 トランザクション特性を変更する基準

トランザクション特性のデフォルト値は、多くのトランザクションに適切な値です。特に、データベースを更新し、他のユーザーとデータベース・アクセスを共有しなければならない場合、このようなユーザーが開始するトランザクションには適正な値です。また、デフォルト値は、スナップショット・ファイルに書き込まれた古いバージョンの行ではなく、最新のデータを必要とするレポート・アプリケーションでも適切に機能します。

データベースが、複数ユーザーによる同時アクセスをサポートすることを前提とすると、次のような場合にデフォルト値を変更してください。

- データベース・クエリーの入力やレポートの作成で、完全に最新ではないデータが含まれる可能性がある場合
トランザクション文に `READ ONLY` を指定してください。データベースでスナップショット・ファイルが有効化されていると、他のデータベース・ユーザーは、テーブルにアクセスできます。また、分離レベルを、シリアライズ可能から、反復可能読取りや読取りコミットにリセットした方がよい場合もあります。トランザクションの分離レベルを下げると、アプリケーションのパフォーマンスは改善されます。
- 他のユーザーよりも高い優先順位をアプリケーションに割り当てることにより、トランザクションの開始後に他のユーザーが割り込んでこないようにする場合
テーブルを明示的に予約し、テーブルの共有モードを保護または排他にアップグレードします。(テーブルの共有モードについては、16.2.4 項 (16-10) と 16.5 項 (16-25) を参照してください)。
- データベース定義を変更する場合 (CREATE、DELETE、ALTER、GRANT または REVOKE 文)
SHARED DATA DEFINITION オプションを指定すると、インデックスの変更中でも、他のユーザーはデータベースにアクセスできます。データベースの一貫性を維持するために、Oracle Rdb は、データベースのテーブルへのアクセスが終了するまでの間、ほとんどのデータ定義への変更を許可していません。Oracle Rdb は、アクセスを明示的に指定しなくても、他のトランザクションによるテーブル・アクセスを明示的に制限したときと同じように動作します。

Oracle Rdb は、このように自動保護機能を提供しますが、定義の変更によって影響を受けるテーブルには、拘束性が高いアクセスを明示的に要求してください。これにより、トランザクションを開始するとすぐに、変更作業の実行に必要なアクセスがあるかどうかわかります。

インデックス定義のみを変更するときは、SHARED または EXCLUSIVE DATA DEFINITION でテーブルを予約できます。SHARED DATA DEFINITION を指定すると、他のトランザクションはテーブルにクエリーを実行できませんが、同時にインデックスを定義できます。EXCLUSIVE DATA DEFINITION でテーブルを要求すると、他のユーザーはテーブルにアクセスできなくなります。この場合、テーブルを要求しているユーザーも含め、すべてのユーザーは、他の目的のためにテーブルにアクセスできなくなります。

- 使用しているデータベース製品が、デフォルトのトランザクション特性の中で、1 つまたは複数をサポートしていない場合
他の製品で必要なトランザクション特性を明示的に指定します。

16.2 項 (16-5) では、トランザクション特性の指定方法や基準について説明しています。

16.2 SQL プログラムにおけるトランザクション特性の指定

複数のユーザーが同時にデータベースにアクセスする場合は、トランザクション特性とロックの関係を理解することが重要です。たとえば、他のユーザーを必要以上に制限するトランザクション特性を使用すると、データから他のユーザーを必要以上にロックアウトすることになります。さらに、予約するテーブルに他のユーザーがすでにアクセスしている場合は、トランザクションを開始できないこともあります。

逆に、一部のタスクで十分な制限のないトランザクション特性を明示的に指定すると、次に示す 2 つのイベントのいずれかが発生する場合があります。

- Oracle Rdb によって明示的な指定が変更されます (データベースの保護に必要な場合)。
- トランザクションで、他のユーザーの妨害によるロックの競合または許容範囲を超えた遅延が発生します。

トランザクション特性を明示的に制御するには、DECLARE TRANSACTION 文か SET TRANSACTION 文を使用します。

DECLARE TRANSACTION 文は実行文ではないので、トランザクションは開始されません。DECLARE TRANSACTION 文を使用する場合、SQL では宣言文、または COMMIT 文か ROLLBACK 文に続く最初の SQL 実行文によってトランザクションが開始されます。後者の場合 (COMMIT 文か ROLLBACK 文に続く場合)、SQL では次に開始するトランザクションに対してトランザクション特性の宣言を適用します。

DECLARE TRANSACTION 文には、次の利点があります。

- 対話型 SQL セッションまたはプログラムに関して、トランザクションのデフォルトを確立します。

- SQL コンテキスト・ファイルに記述できます。
データベースとトランザクションのコンテキストを管理するには、標準の SQL 言語にはない Oracle Rdb に固有の情報を提供する必要があります。このために、Oracle Rdb への SQL インタフェースを使用すると、上記の情報をプログラムのソース・ファイルに記述するか、他のなんらかの方法で記述するかを選択できます。SQL モジュール言語と埋込みの SQL プログラムで可能な方法は、データベースとトランザクションに関する情報を SQL コンテキスト・ファイルに記述することです。SQL モジュール・プロセッサでコンテキスト・ファイルを使用する方法については 5.5 項 (5-5) を、SQL プリコンパイラでコンテキスト・ファイルを使用する方法については 6.6.2 項 (6-12) を参照してください。

DECLARE TRANSACTION 文を使用すると、明示的な SET TRANSACTION 文でトランザクションを開始しない限り、すべてのトランザクションの特性が同じになります。

プログラムでは、SQL モジュールまたはプリコンパイル・ソース・ファイルごとに DECLARE TRANSACTION 文は 1 つに限定されます。DECLARE TRANSACTION 文を使用するプログラムで様々なデフォルトのトランザクション特性を確立するには、プログラム内で別々のファイルに SQL 文を分散し、ファイルごとに異なる DECLARE TRANSACTION 文を適用します。ソース・ファイル内でトランザクション特性を変更する場合は、SET TRANSACTION 文を使用して個々のトランザクションを開始します。

SET TRANSACTION 文は実行文であり、トランザクションの指定と開始の両方を行います。プリコンパイル・ソース・ファイルまたは SQL モジュールには、複数の SET TRANSACTION 文を記述できます。SET TRANSACTION 文には、次の利点があります。

- トランザクションを開始するタイミングを明示的に制御できます。
- プログラムのソース・ファイル内でトランザクション特性を柔軟に変更できます。

上記の 2 つの文によって、トランザクション特性を明示的に指定できます。このためのオプションについては、後続の各項で詳しく説明します。

16.3 項 (16-21) では、SET TRANSACTION 文と DECLARE TRANSACTION 文を併用する方法について説明します。

16.2.1 読取り専用トランザクションの使用方法

読取り専用トランザクションを指定すると、トランザクションはデータの検索に限定されますが、読取り / 書込みトランザクションを指定した場合よりも多くのユーザーが同時にデータベースにアクセスできます。

次の場合は、DECLARE TRANSACTION 文か SET TRANSACTION 文で READ ONLY を指定します。

- データベースに対して行の追加や既存の値の変更をする予定がない場合

- データ定義の作成、変更、削除、またはデータベース権限の許可や取消しをする予定がない場合
- 絶対的な最新の正確性が要求されない場合
読取り専用トランザクションは、トランザクションの開始時点での瞬間的なデータベース、すなわち事実上のデータベースのスナップショットを参照します。アプリケーションで絶対的な最新イメージを必要としない場合は、読取り専用トランザクションを使用してください。
- 読取り専用記憶領域にアクセスする場合
読取り専用記憶領域のデータは更新できないので、読取り専用記憶領域にスナップショットはありません（この章におけるスナップショット・ファイルとロックの説明は、読取り専用記憶領域には適用されません）。
- Oracle Rdb を読取り専用アクセスが必要なデータベース製品と併用する場合
他の製品で使用するロックに関する規則や方法は、Oracle Rdb とはまったく異なる場合があるので注意してください。

読取り専用トランザクションを開始すると、トランザクションはデータベース・ファイルから最新バージョンの行を読み取り、スナップショット・ファイルから前バージョンの行を読み取ります。スナップショット・ファイルの行は多くのトランザクションで共有できるので、他のトランザクションと競合することはありません。

RESERVING 句を指定しないで読取り専用トランザクションを開始すると、トランザクションはデータベース内の各テーブルへのアクセスを要求します。データベース操作は検索のみに制限されます。読取り専用トランザクション・モードでは行のロックを使用しないので、他のトランザクションも同じデータにアクセスできます。他のトランザクションで変更を行っても、引き続きスナップショット・ファイルから古いデータを取得できます。読取り専用トランザクションはスナップショット・ファイルのデータを読み取るので、他のトランザクションは待機もアクセスの競合もなく任意のテーブルを自由に更新できます。

RESERVING 句を読取り専用トランザクションと組み合わせて指定することもできますが、この句を記述してもトランザクション実行中に処理できるテーブルを指定（限定）するのみです。共有読取りモードは、共有モードと読取り専用トランザクションに指定できるロック・タイプのキーワードの組合せにすぎません。

ご使用のデータベースでスナップショット・ファイルが有効でない場合は（さらに読取り専用記憶領域にアクセスしていない場合）、読取り専用トランザクションには共有読取り予約オプションを使用してテーブルを予約する読取り / 書込みトランザクションと同じ効果があります。遅延スナップショットが有効な場合は、読取り専用トランザクションはスナップショット・ファイルが更新されるまで待機し、それから通常の処理を行います。

スナップショット・ファイルに格納された前のバージョンの行は数秒前あるいは数分前のものなので、ほとんどの場合は問題ありません。ただし、検索のトランザクションでデータベースに関する絶対的に最新のイメージが必要な場合は、スナップショット・ファイルを使用しないように読取り / 書込みトランザクションを開始します。

サイト内で優先順位が高い複雑なレポートを検索トランザクションで生成する場合は、保護共有モードの読取りでテーブルを予約することもできます。他のトランザクションの妨害があると、競合性の高いトランザクションを開始するのはますます困難になりますが、いったんトランザクションが開始されれば他のトランザクションによる妨害は最小になり、より短い時間で完了します。

16.2.2 読取り / 書込みトランザクションの使用方法

データベースに関する絶対的な最新イメージが必要な検索のタスクを実行する場合は、読取り / 書込みトランザクションを指定してください。さらに、INSERT、DELETE、UPDATE、CREATE、ALTER、DROP、GRANT、REVOKE の各文では、読取り / 書込みトランザクションを使用してスキーマの定義を変更する必要があります。

次の文では、他のトランザクションと競合するモードで予約されていないテーブルまたはビューに関する操作を実行できます。

```
SET TRANSACTION READ WRITE;
```

Oracle Rdb では、SQL 文でテーブル名を指定したときにテーブルを予約し、トランザクションで実行する操作の種類によって、検索のタスクを完了する場合は選択された行に読取りのロックを適用し、更新のタスクを完了する場合は選択された行に書込みのロックを適用します。

たとえば、トランザクションで最初にデータ操作を行う SQL 文で、EMPLOYEES テーブルの行を検索するとします。Oracle Rdb は、共有読取り予約オプションで指定するのと同じ方法でテーブルと行をロックします。同じトランザクションの後半で、EMPLOYEES テーブルの選択された行を変更するとします。Oracle Rdb は、トランザクションの完了に必要なロックのみを使用し、共有読取りロックを共有書込みに昇格します。

データ定義文を使用すると、システム・テーブルがロックされます。システム・テーブルには、データベースの定義が記述されています。変更の内容によって、このトランザクションが終了するまでの間、他のユーザーはデータベース内の 1 つまたは複数、あるいはすべてのテーブルのデータにアクセスできなくなります（つまり、これらのテーブルへの EXCLUSIVE WRITE アクセスを指定するのと同じ効果があります）。

一部のデータベース操作では、SQL 文の指定よりも高水準の保護が必要になります。このような場合、Oracle Rdb はモードを自動的に読取り保護または書込み保護に昇格します。

Oracle Rdb では、常にトランザクションとデータベースを保護するために必要な最高の共有モードとロック・タイプを保証します。共有モードが明示的な指定より高い場合も、データの処理を開始してからはそのレベルより下がることはありません。

16.2.3. バッチ更新トランザクションの使用方法

バッチ更新トランザクションでは、読取り / 書込みトランザクションよりパフォーマンスは向上しますが、リカバリと同時アクセスはできません。バッチ更新トランザクションは、排他的予約オプションを使用する読取り / 書込みトランザクションよりも高速に機能します。

バッチ更新トランザクションでは、スナップショット・ファイルや Recovery-unit ジャーナル・ファイルへの書き込みを行わないからです。結果として、ディスク I/O 操作が絶対的の最小に保持されます。しかし、バッチ更新トランザクションではロールバックに必要な情報が入手できないので、必ず COMMIT 文で終了することになります。バッチ更新トランザクションがエラーになると、データベースは破損します。唯一の方法はデータベース全体を再構築し、バックアップ・ファイルからリストアすることです。

バッチ更新トランザクション実行中は、他のユーザーはデータベースにアクセスできません。

バッチ更新トランザクションを使用すると、データベースを迅速にロードできます。たとえば、抽出したデータベース内のすべてのデータをリフレッシュするとします（抽出したデータベースは別のデータベースの一部のデータまたはすべてのデータのコピーであり、通常は即席のクエリーまたはレポートをサポートするために作成されます）。トランザクションをコミットする前にシステム障害などの異常なイベントが発生した場合は第 2 のデータベースが破損するという可能性を考慮しても、バッチ更新トランザクションによるパフォーマンスの向上に価値があると判断したとします。バッチ更新トランザクションを開始する直前に、必ず第 2 のデータベースのバックアップ・コピーを作成し、データベースが破損した場合にリストアできるようにしてください。次の例に、1 つのデータベースでバッチ更新トランザクションを開始し、別のデータベースで読取り専用トランザクションを開始する方法を示します。

```
SET TRANSACTION
  ON FIRST_DB USING (READ ONLY)
  AND ON SECOND_DB USING (BATCH UPDATE);
```

最初のトランザクションは、SECOND_DB として指定されたデータベースにアクセスするユーザーが他にいない場合にのみ開始されます。

Recovery-unit ジャーナルのサポートがないと、トランザクションはロールバックできません。したがって、バッチ更新トランザクションは必ず COMMIT 文で終了します。エラーの状態が COMMIT 文では処理されない場合、またはシステムなどの障害によってコミット操作が実行できない場合、データベースはバッチ更新トランザクションでアクセスしたデータベースは破損とマーク付けされるので、2 度とアクセスできなくなります。

明示的な SET TRANSACTION 文で BATCH UPDATE を指定します。この方法は、更新トランザクションのパフォーマンスが向上すれば対話型セッションまたはプログラムが異常終了した場合にデータベースが破損するというリスクがあってもよいと判断する場合にのみ使用してください。このリスクは次の場合に妥当と判断されます。

- 最初にデータベースをロードする場合
ロード操作が失敗した場合にデータベース定義の作成に使用できるプログラムまたは SQL コマンド・プロシージャを用意する必要があります。バッチ更新トランザクションがコミットされる前に失敗した場合は、破損したデータベースを削除してデータベースを再定義し、ロード操作を再開してください。
- データベース内のほとんど（すべてではない場合でも）のテーブルに対して大幅な更新操作を実行する場合

バッチ更新トランザクションを開始する直前に RMU Backup コマンドを使用してデータベースをバックアップする必要があります。バッチ更新トランザクションがコミットされる前に失敗した場合は、破損したデータベース・ファイル（1 つ以上）を削除し、RMU Restore コマンドを使用してバックアップ・コピーからデータベースをリストアしてから、更新操作を再開してください。

16.2.4 RESERVING 句の使用方法

SET TRANSACTION 文か DECLARE TRANSACTION 文を入力しないと、SQL では接続または宣言されたすべてのデータベースを使用するトランザクションを開始すると仮定します。したがって、次の文を入力したのと同様に動作します。

```
DECLARE TRANSACTION READ WRITE WAIT;
```

SQL は、SQL 文でテーブルを参照するときにテーブルを予約します。さらに、行にアクセスするときに必要に応じて読取りまたは書き込みの行ロックを適用します。

トランザクションを開始するタイミングとアクセスするテーブルがわかる場合は、明示的にテーブルを予約してください。Oracle Rdb では、テーブル全体に対して集中的なロックを適用します。プログラムは、他のトランザクションがテーブルに非互換のロックを適用している場合は解放するまで待つ必要があります。逆に、トランザクションが完了するまで他のトランザクションを待たせる場合もあります。

読取り / 書き込みトランザクションでは、RESERVING 句で共有モードとロック・タイプを次の組合せで指定できます。

- SHARED READ
- SHARED WRITE
- SHARED DATA DEFINITION
- PROTECTED READ
- PROTECTED WRITE
- EXCLUSIVE READ
- EXCLUSIVE WRITE
- EXCLUSIVE DATA DEFINITION

図 16-1 (16-11) に、SET TRANSACTION RESERVING の構文を示し、読取り / 書き込みトランザクションに指定できる共有モードとロック・タイプを要約します。

図 16-1 読取り / 書き込みトランザクションの共有モードとロック・タイプのオプション

```
SQL> SET TRANSACTION READ WRITE
```

```
RESERVING table_name FOR share mode lock type
```

READ	テーブルから行を検索するが、行の変更や新しい行の格納はしない。
WRITE	行を検索し、行の変更または新しい行の格納を行う。
DATA DEFINITION	インデックスを追加または削除する。

SHARED	他のユーザーが同じテーブルに対して作業を行うことができる。他のユーザーは、自分が選択したオプションに従って、テーブルに対して読取り専用アクセスまたは読取り/書き込みアクセスができる。
PROTECTED	他のユーザーが、同じテーブルから行を読み取ることができるが、書き込みアクセスはできない。
EXCLUSIVE	他のユーザーは、テーブルから行を読み取ることもできない。他のユーザーが同じテーブルにアクセスすると、SQLはその要求を拒否する。

NU-2111A-RA

RESERVING 句は、読取り専用トランザクションでも使用できます。次に例を示します。

```
SET TRANSACTION READ ONLY
```

```
RESERVING EMPLOYEES FOR SHARED READ;
```

この場合は、RESERVING 句で指定しないテーブルにはアクセスできません。さらに、Oracle Rdb で READ ONLY と RESERVING 句を組み合わせると、保護共有モードと排他的共有モードまたは書き込みのロック・モードは評価されません。

トランザクションで複数のテーブルを予約するには、各テーブルを明示的に指定します。次の例では、3 つのテーブルを予約して読取り専用アクセスを行います。

```
SET TRANSACTION READ ONLY RESERVING
```

```
EMPLOYEES FOR SHARED READ,
```

```
JOBS FOR SHARED READ,
```

```
DEPARTMENTS FOR SHARED READ;
```

この文で開始したトランザクションは、指定のないテーブルからの読取りを実行できません。

次の SET TRANSACTION 文で名前を指定した各テーブルは、データベースに関する別々のタスクに使用されます (EMPLOYEES は検索、COLLEGES と DEGREES は更新)。

```
SET TRANSACTION READ WRITE RESERVING
    EMPLOYEES FOR SHARED READ,
    COLLEGES FOR SHARED WRITE,
    DEGREES FOR EXCLUSIVE WRITE;
```

次の文は 2 つのテーブルを予約します。この文は、1 つのテーブルからデータを読み取って別のテーブルに格納するが、行はロックしない場合に使用してください。この文では、他のユーザーはいずれのテーブルにもアクセスできません。

```
SET TRANSACTION READ WRITE RESERVING
    EMPLOYEES FOR EXCLUSIVE READ,
    EMPLOYEES_TEMP FOR EXCLUSIVE WRITE;
```

いずれかのテーブルにアクセスする他のトランザクションがすべて完了しないと、前述のトランザクションは開始できません。

明示的に予約していないテーブルもロックされる場合があることを忘れないでください。この状況は、予約したテーブルのビュー、制約、トリガーが他のテーブルを参照する場合に発生します。次の点に注意してください。

- トリガーか制約が最初にテーブルにアクセスするときに、Oracle Rdb は共有読取りモードで自動的にテーブルを予約します。
- トリガーがテーブルを更新する場合は、Oracle Rdb は共有書込みモードで自動的にテーブルを予約します。
- ビューを明示的に予約すると、Oracle Rdb はビューがテーブルを参照する場合にそのテーブルを同じ共有モードで自動的に予約します。

必要なテーブルのいずれかに別のユーザーがアクセスしている場合は、そのテーブルがロックされている可能性があります。ロックの問題を回避するには、アクセスするすべてのテーブルを明示的に予約してください。

Oracle Rdb では、computed-by 列が参照するテーブルを自動的に予約することはありません。このようなテーブルは、明示的に予約する必要があります。

16.2.5 ロックの取得を待つかどうかの選択

他のユーザーがテーブルまたは行のロックを解放するまで待機する場合は、WAIT オプションを指定します。アクセスするテーブルを他のユーザーがロックしているときにただちにロック競合のメッセージを表示する場合は、NOWAIT オプションを指定します。デフォルトは WAIT オプションです。たとえば、次のトランザクションは必要なロックが得られるまで待機します。

```
SET TRANSACTION READ ONLY;
```

ロックの競合をプログラムで直接処理する場合は、SET TRANSACTION 文か DECLARE TRANSACTION 文で NOWAIT オプションを指定します。別のトランザクションで行をロックしている場合、NOWAIT 特性を持つトランザクションではロック競合エラーが発生し、行を使用できないことが通知されます。現行トランザクションを終了し、行にアクセスする文を再び実行すれば、行の取得を再試行できます。この文が正常に実行されるのは、その行のロックを保持するトランザクションが終了した場合だけです。

ロックの競合をただちに通知させる場合にのみ、NOWAIT オプションを指定してください。次に例を示します。

```
SET TRANSACTION READ WRITE RESERVING
    EMPLOYEES FOR PROTECTED WRITE,
    JOB_HISTORY FOR PROTECTED WRITE,
    SALARY_HISTORY FOR SHARED READ NOWAIT;
```

NOWAIT オプションを指定すると、プログラムでロックの競合を処理する方法を指定できます。プログラムでは、待機するかしないかをユーザーに選択させたり、特定の回数だけ再試行してからレコードを使用できないことをユーザーに通知したりします。たとえば、ユーザーが待機するかしないかを選択できるようにするには、プログラムに次のロジックを記述します。

1. NOWAIT を指定してトランザクションを開始します。
2. データ操作を開始します。
3. データ操作文でロックの競合が発生した場合は、NOWAIT トランザクションをロールバックし、"The record you want is in use. Do you want to wait?" というメッセージを表示します。
4. ユーザーが Y または y ユーザーがで答えた場合は、WAIT を指定するトランザクションを開始します。ユーザーが N または n で答えた場合は、アプリケーションごとに適切な処理を行います。

待機間隔を指定し、トランザクションがロックの解放を待機する時間を設定します。

アプリケーションに固有の待機時間を指定するには、SET TRANSACTION 文か DECLARE TRANSACTION 文の WAIT 句を使用します。次の例では、SET TRANSACTION 文を使用して待機間隔を 15 秒に指定しています。

```
SET TRANSACTION READ WRITE WAIT 15;
```

Oracle Rd は、ロックの解放を 15 秒待ってからロック競合エラーを返します（待機間隔は秒単位で表しますが、正確な間隔の近似値です）。

Oracle Rdb には、待機間隔を指定するいくつかの方法が用意されており、プロセス内の各トランザクション、データベース全体、またはシステム全体に適用できます。待機間隔に関する複数のオプションをトランザクションに適用すると、Oracle Rdb は最小値を採用します。すなわち、任意の待機間隔オプションで指定した間隔の中で最も短い時間だけ待機します。

待機間隔オプションと Oracle Rdb がどの待機間隔を使用するかを決定する方法の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

待機間隔の指定に SET TRANSACTION 文の WAIT 句を使用するか別の方法を使用するかにかかわらず、トランザクションが指定の時間だけ待機すると、Oracle Rdb はロック競合エラーを返します。

16.2.6 分離レベルの選択

トランザクションの**分離レベル**は、1つのトランザクションの読取り操作が同時に実行する別のトランザクションの更新操作によって影響される程度を定義します。

Oracle Rdb には、トランザクションに関して次に示す3つの分離レベルがあります。

- **SERIALIZABLE** (直列可能)。同時に実行するトランザクションの操作が他のどのトランザクションからも影響されないことが保証されます。直列可能なトランザクションを同時に実行しても、必ず同じトランザクションを1つずつ順番に実行したのと同じ結果になります。
デフォルトでは、Oracle Rdb トランザクションはすべて ISOLATION LEVEL SERIALIZABLE で実行されます。
- **REPEATABLE READ** (反復可能読取り)。同じクエリーを再実行した場合に初回の読取りと同じ行が得られることが保証されます。ただし、他のトランザクションで挿入され、コミットされた行も表示される場合があります。このような行は**ファントム**と呼ばれており、プログラムで COUNT、AVERAGE など、範囲の集計プロパティに基づく操作を実行する場合に、データの非一貫性を招くことがあります。
- **READ COMMITTED** (コミット読込み)。トランザクションでは他のトランザクションでコミットされたデータをすべて参照できます。Oracle Rdb は、カーソルが次の行に進んだ場合かカーソルがクローズした場合に、読取りのロックを解放します。したがって、トランザクションで読み取るデータ項目はそのトランザクションが終了する前に別のトランザクションで更新され、コミットされることがあります。したがって、アプリケーションは変更されない状態のデータを信頼することができません。ただし、表示中にデータ項目が変更されることはありません。

分離レベルは、SET TRANSACTION 文と DECLARE TRANSACTION 文で明示的に指定します。

分離レベルが影響を及ぼすのは、読取り / 書込みトランザクションだけです。読取り専用トランザクションは、常にスナップショット・ファイルから読取りを行います (スナップショットが有効な場合)。したがって、他のトランザクションの変更は参照されません。さらに、読取り / 書込みトランザクションの読取り操作のみが、同じトランザクション内で同じクエリーが1度以上実行される場合にのみ影響を受けます。

Oracle Rdb は、トランザクション実行中に様々なデータベース・リソースを長期間にわたってロックします。直列可能トランザクションによってデータ整合性は保証されますが、デー

データベース・オブジェクトへの複数ユーザーによる同時アクセスは制限されます。分離レベルを下げると、厳密な定義によるデータ整合性は犠牲になりますが、トランザクションの並行性は向上します。トランザクションが低い保護レベルでも適切に動作すれば、データベース・システムはデータベースの並行性を向上できます。整合性と保護に関するプログラムのニーズとデータベース全体のパフォーマンスとのバランスをとる必要があります。

表 16-1 (16-15) に、各分離レベルで許可される現象を示します。

表 16-1 各分離レベルで許可される現象

分離レベル	反復不可能な読取り が許可されるか	ファントム が許可されるか
READ COMMITTED	可	可
REPEATABLE READ	不可	可
SERIALIZABLE	不可	不可

注意: RESERVING 句を指定してテーブルを予約すると、その RESERVING 句は指定した分離レベルの動作より優先されます。たとえば、Oracle Rdb は明示的に保護検索を予約したテーブルでは常にファントムを回避します。一部のテーブルを保護検索で予約し、他のテーブルを同時検索で予約すると、Oracle Rdb は同時検索で予約したテーブルで発生するファントムを回避できません

すべての分離レベルについて、UPDATE ONLY カーソルで行を読み取ると、Oracle Rdb が行を排他的にロックし、COMMIT 文か ROLLBACK 文が発行されるまでロックを保持します。

16.2.6.1 直列可能トランザクションの使用方法

直列可能トランザクションを指定すると、Oracle Rdb では同じトランザクション内で行を変更しない限り、そのトランザクション実行中に行を繰り返し検索しても同じ行が得られることが保証されます。同じトランザクション内で行を変更した場合は、同じ行を再び要求すると変更された値が表示されます。トランザクションを開始してから初回の検索の前に、他のトランザクションが変更をコミットすることもあります。

しかし、初回の行検索の後には他のトランザクションによる変更は参照できません。初回の行検索でその行に読取りのロックが適用されるので、トランザクションが終了してロックが解放されるまで他のトランザクションはその行の更新も削除もできません。

16.2.6.2 反復可能読取りトランザクションの使用方法

反復可能読取りトランザクションを指定すると、Oracle Rdb は 1 つのトランザクション実行中に繰り返し実行されるクエリーが（クエリーを実行するたびに）クエリーの初回の実行で

16.2 SQL プログラムにおけるトランザクション特性の指定

返した行と同じ行を返すことが保証されます。反復可能読取りトランザクションでは同じ行セットが保証されますが、ファントムと呼ばれる余分な行が表示されないことは保証できません。

次に、反復可能読取りトランザクションでファントム行を検出する例を示します。反復可能読取りトランザクションでは、データベースのクエリーを実行して行セットを選択します。この例のクエリーは6行を返します。

```
ATTACH 'FILENAME mf_personnel';
SET TRANSACTION READ WRITE
    ISOLATION LEVEL REPEATABLE READ;
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL
    FROM EMPLOYEES WHERE EMPLOYEE_ID < '00170';
EMPLOYEE_ID  LAST_NAME      FIRST_NAME  MIDDLE_INITIAL
00164        Toliver      Alvin       A.
00165        Smith       Terry       D.
00166        Dietrich    Rick        NULL
00167        Kilpatrick  Janet       NULL
00168        Nash        Norman      NULL
00169        Gray        Susan       O.
6 rows selected
```

反復可能読取りトランザクションが実行する間に、別のトランザクションが開始され、mf_personnel データベースに1行を挿入し、トランザクションをコミットします（次を参照）。

```
ATTACH 'FILENAME mf_personnel';
SET TRANSACTION READ WRITE;
INSERT INTO EMPLOYEES
    (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL)
VALUES ('00161', 'Muggs', 'Fred', 'J');
1 row inserted
COMMIT;
```

最初のトランザクションで同じクエリーを繰り返し実行した場合にクエリーは前と同じ6行を返しますが、次の例に示すように2番目のトランザクションで挿入されたファントム行（Fred Muggs）も返します。

```
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL
    FROM EMPLOYEES WHERE EMPLOYEE_ID < '00170';
EMPLOYEE_ID  LAST_NAME      FIRST_NAME  MIDDLE_INITIAL
00161        Muggs       Fred        J.
00164        Toliver      Alvin       A.
00165        Smith       Terry       D.
00166        Dietrich    Rick        NULL
00167        Kilpatrick  Janet       NULL
00168        Nash        Norman      NULL
00169        Gray        Susan       O.
7 rows selected
```

Oracle Rdb では、読取りが反復可能なこと、すなわち Oracle Rdb はクエリーを実行するたびに少なくとも最初に選択された行セットは返すことが保証されます。ただし、他の行（ファントム）が表示されないことは保証できません。

反復可能読取りトランザクションはインデックスに対して短期間のロックを保持し、テーブルに対してトランザクション終了までロックを保持します。

分離レベル REPEATABLE READ を使用すると、Oracle Rdb は実際には反復不可能な読取りの発生を回避する必要のない行について長期間の読取りロックを保持することがあります。REPEATABLE READ 分離レベルでは、インデックスの競合は減りますが、データの競合は減りません。

16.2.6.3 コミット読込みトランザクションの使用方法

コミット読込みトランザクションを指定すると、Oracle Rdb は1つのトランザクション実行中に繰り返し実行されるクエリーが（クエリーを実行するたびに）クエリーの初回の実行で返した行と同じ行を返すことは保証できません。さらに、ファントムと呼ばれる余分な行が表示されないことも保証できません。コミット読込みトランザクションでは、行の数も行のデータも変わる場合もあります。また、現在実行中の他のトランザクションでコミットされたデータの読取りが許可されます。

多くのアプリケーションは、反復不可能な読取り / 書込みトランザクションを許可すると（READ COMMITTED 分離レベルでトランザクションを実行する場合など）、一貫性のない結果を返します。実行されていない更新（buried update anomaly）としてよく知られている例で、このことを示します。

- 2つのトランザクションで同じ行を読み取り、在庫の2品目を検出します。
- 最初のトランザクションでは、在庫総数から1品目を減らし、更新して結果をコミットします。
- 2番目のトランザクションでも在庫総数（最初の読取りによる2品目）から1品目を減らします。更新して結果をコミットします。

2つのトランザクションで実際には品目数を2つ減らしましたが、ディスク上の行は1品目の在庫を示しています。2番目の更新トランザクションによって、最初のトランザクションが上書きされています。後続のトランザクションは、誤った品目数が記載された行を読み取り、実際には存在しない在庫の1品目の検出します。これは**誤って埋没した更新**と呼ばれます。

誤って埋没した更新を回避するには、在庫の品目に関するトランザクションを分離レベル READ COMMITTED でなく REPEATABLE READ または SERIALIZABLE で実行する必要があります。

次の例に、データベースのクエリーを実行して行セットを選択するコミット読込みトランザクションを示します。この例のクエリーは6行を返します。

```
SET TRANSACTION READ WRITE
ISOLATION LEVEL READ COMMITTED;
```

16.2 SQL プログラムにおけるトランザクション特性の指定

```
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL
      FROM EMPLOYEES WHERE EMPLOYEE_ID < '00170';
EMPLOYEE_ID  LAST_NAME      FIRST_NAME     MIDDLE_INITIAL
00164        Toliver      Alvin          A.
00165        Smith       Terry          D.
00166        Dietrich    Rick           NULL
00167        Kilpatrick  Janet          NULL
00168        Nash        Norman         NULL
00169        Gray        Susan          O.
6 rows selected
```

同時に、別のトランザクションが開始され、mf_personnel データベースに 1 行を挿入し、別の行を更新してトランザクションをコミットします（次を参照）。

```
SET TRANSACTION READ WRITE;
INSERT INTO EMPLOYEES
      (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL)
VALUES  ('00161','Muggs','Fred','J');
1 row inserted
UPDATE EMPLOYEES
      SET FIRST_NAME = 'David',
          LAST_NAME = 'Garroway',
          MIDDLE_INITIAL = 'E'
      WHERE EMPLOYEE_ID = '00164';
1 row updated
COMMIT;
```

最初のトランザクションで前と同じ ID 番号の範囲を要求すると、2 番目のトランザクションで行った変更を返します。つまり、ファントム行（Fred Muggs）が追加され、Alvin Toliver の代わりに David Garroway の行を返します。

```
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL
      FROM EMPLOYEES WHERE EMPLOYEE_ID < '00170';
EMPLOYEE_ID  LAST_NAME      FIRST_NAME     MIDDLE_INITIAL
00161        Muggs       Fred           J.
00164        Garroway    David          E.
00165        Smith       Terry          D.
00166        Dietrich    Rick           NULL
00167        Kilpatrick  Janet          NULL
00168        Nash        Norman         NULL
00169        Gray        Susan          O.
7 rows selected
COMMIT;
```

反復可能読取りトランザクションでは、Oracle Rdb が Alvin Toliver の行を再び表示し（最初にクエリーを実行したときと同じ）、現在実行中の他のトランザクションによる更新は待機します。

コミット読み込みトランザクションはインデックスとテーブルに対して短期間のロックを保持します。

順次スキャンを分離レベル READ COMMITTED で実行すると、ロック操作の回数が増加します。ALG (Adjustable Lock Granularity) が無効な場合は、ロック操作の回数（およびロックの総数）が増加し、許容レベルを超える場合があります。

16.2.7 様々な分離レベルを使用する利点

SERIALIZABLE 分離レベルを使用すると、特殊なプログラミングまたはアプリケーション設計を使用しない場合、同時に実行するトランザクション間の関係においてすべてのトランザクションが適正に実行されます。誤って埋没した更新のケースに固有の不適切なインベントリの数字を追跡し、タスクをデバッグするには時間がかかります。非一貫性の原因をレポートで検出するにも、かなりの時間が必要です。SERIALIZABLE 分離レベルを使用すると、この種の時間のかかる作業から解放されます。

しかし、直列可能トランザクションの利点にはコストという代償があります。有害な状況をなくすために、データベース・システムはデータ・レコードやインデックス・ノードなど、アクセスするオブジェクトに対する長期の読取りロックを使用します。長期のロックを保持する結果として、並行性が低下します。ファントムを回避しなくても適切に実行できるトランザクションが分離レベル SERIALIZABLE で実行されている場合は、関連のない行を挿入するトランザクションをブロックしている可能性があります。この結果、アプリケーション全体のパフォーマンスが低下する恐れがあります。このトランザクションを分離レベル REPEATABLE READ で実行すると、データベース・システムは長期間の読取りロックを保持する必要がないので、アプリケーション全体のスループットが向上します。

トランザクション実行中に対話を行うアプリケーションでは、データ・レコードやインデックス・ノードに関する長期の読取りロックを保持しないコミット読み込みトランザクションを大いに活用してください。長期間のレポート・トランザクションも更新トランザクションをほとんど妨害しないので、コミット読み込みトランザクションを使用できます。多くのレポートの実行が必要な場合、またはレポートで直列可能保護が必要な場合は、分離レベル READ COMMITTED よりも読取り専用トランザクションの方がロックが少ないので適しています。

ユーザーが分離レベルを選択できるので個々のトランザクションに最適な並行性と一貫性に合った設定が可能になりますが、一方、様々な分離レベルを柔軟に使用することによって、アプリケーション設計によるデータの非一貫性やプログラミング・エラーが発生する恐れもあります。一部のアプリケーションは、最高でデフォルトの SERIALIZABLE 分離レベルより下の分離レベルで実行しても予測可能な結果が得られます。反復不可能な読取りまたはファントムが発生した場合に、これらのアプリケーションの動作によって一貫性が保持されるかを慎重に検討してください。一般に、同じクエリーの式を 2 回以上実行するトランザクションのないアプリケーションは、分離レベルを下げて実行できます。

分離レベルの低いアプリケーションは、レポートと更新の 2 つに分類されます。

レポート・アプリケーションで低い分離レベルを使用する方法

多くの便利なレポートや独自の問合せは、反復不可能な読取りとファントムを許可するアプリケーションから生成されます。たとえば見積りを計算する場合は、高い精度が必要な期末レポートなど、他の環境では許容できない非一貫性を許容できます。

本質的に、1 回限りのレポートまたはクエリーは反復不可能な読取りやファントムの影響を受けません。たとえば、販売データに関する 1 つのパスを作成する（先週販売された各品目のパーセンテージを計算する）クエリーには、直列可能トランザクションが提供する保護は不要です。このようなレポートまたはクエリーは、分離レベル `READ COMMITTED` または `REPEATABLE READ` で実行しても目的には適合します。

更新アプリケーションで低い分離レベルを使用する方法

慎重に構成された更新トランザクションは、多くの場合、反復不可能な読取りやファントム発生の可能性があっても正しく実行されます。ただし、このためにはアプリケーションへのコードの追加が必要な場合があります。

顧客が様々な色の T シャツを 100 枚発注したとします。この注文を処理するトランザクションでは、各色の在庫の枚数が表示されます。ここで、顧客の発注書に追加する各色の枚数を入力します。このトランザクションを実行し、反復不可能な読取りとファントムを許可し（`READ COMMITTED` 分離レベルを使用し）、しかもプログラムに特定のロジックを組み込むと、誤って埋没した更新を回避できます。

アプリケーションでブルーの T シャツの行を読み取り、25 枚の在庫を検出したとします。顧客は 10 枚の発送を要求しています。トランザクションは分離レベル `READ COMMITTED` で実行されるので、行を更新して在庫に 15 枚の青の T シャツがあるように記録することはできません。したがって、このアプリケーションで行を読み取ってから更新するまでの間に別のトランザクションで行を変更する必要があります。

そこで、再び行を読み取り、他に変更できる者がいないことを確認します。このために、`UPDATE ONLY` 句を指定して宣言した第 2 のカーソルをオープンして行をもう一度読み込みます。ここで、アプリケーションは再び読み込んだ行の値から 10 を引いて残りのインベントリを再計算し、更新を発行する必要があります。

これは、同じトランザクションで読取りと更新を行う代わりであり、トランザクションがレコードを処理する間に他のユーザーがそれを読み込めないことが保証されます。

Oracle Rdb 以外のデータベースで分離レベルを使用する方法

Oracle Rdb データベース以外のデータベースでは、`ISOLATION LEVEL READ COMMITTED` など、特定の分離レベルの指定がたびたび必要になります。Oracle Rdb 以外のデータベース製品で SQL を使用する場合は、その製品に必要な分離レベルを指定します。

16.2.8 エイリアスを使用して 1 つのトランザクション内で複数のデータベースにアクセスする方法

1 つの Oracle Rdb データベース内の 1 つのテーブルからデータを読み込んで別の Oracle Rdb データベースに格納する場合は、エイリアスを使用する必要があります。次の例で、FIRST_DB と SECOND_DB は ATTACH 文または DECLARE ALIAS 文で指定したエイリアスです。

```
SET TRANSACTION
  ON FIRST_DB USING (READ ONLY
    RESERVING FIRST_DB.EMPLOYEES FOR SHARED READ)
  AND ON SECOND_DB USING (READ WRITE
    RESERVING SECOND_DB.EMPLOYEES FOR EXCLUSIVE WRITE);
```

複数のエイリアスを使用してすべてのデータベースを修飾する必要があります。それ以外の場合、Oracle Rdb はオブジェクトがデフォルトのデータベースにあると見なされ、そのデータベースが検索されます。デフォルトのデータベースを確立していないと、Oracle Rdb はエラーを返します。デフォルトのデータベースが必要なデータベースでない場合は、デフォルトのデータベースに指定した名前のオブジェクトが存在するかどうかによって、エラーが発生したり予期せぬ結果が得られたりすることがあります。

エイリアスの詳細は、17 章 (17-1) を参照してください。

16.3 トランザクション・スコープについての理解

トランザクションの開始を示す実行文と、データベースへの変更をコミットまたはロールバックする（トランザクションを終了する）文によって、トランザクションの**スコープ**が特定されます。Oracle Rdb は、トランザクション・スコープ内の文をすべて実行するか、または 1 つも実行しません。

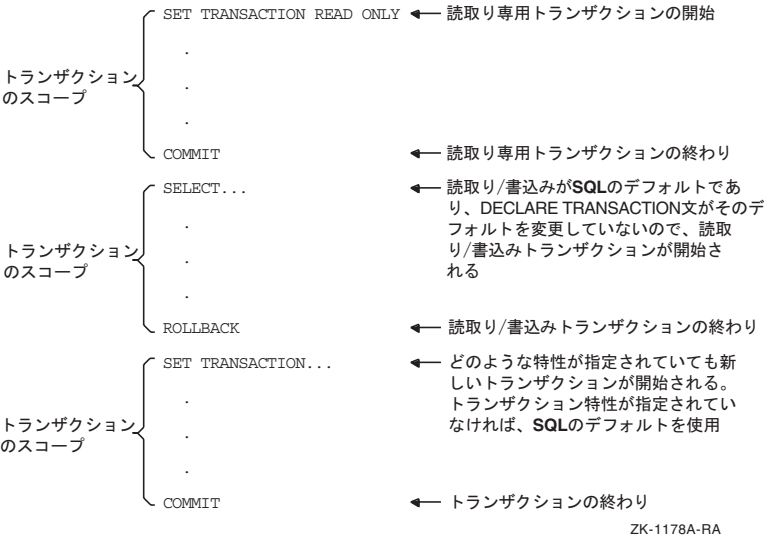
SET TRANSACTION 文を使用すると、プログラムで明示的にトランザクションを開始できます。それ以外の場合は、データベース・アクセスを必要とする実行文で暗黙的に開始します。SQL では、対話型の SQL セッションまたはプログラム内で、COMMIT 文か ROLLBACK 文に続く最初の SQL 実行文によってトランザクションが開始されます。

暗黙的なトランザクションを開始すると、Oracle Rdb はトランザクション特性に関する現在のデフォルトの値を使用します。このデフォルト値は、DECLARE TRANSACTION 文で設定したデフォルト値です。DECLARE TRANSACTION 文を指定しない場合は、Oracle Rdb のデフォルト値です。Oracle Rdb のデフォルトのトランザクションは、必要なロックを待機し、ISOLATION LEVEL SERIALIZABLE で動作する読取り / 書込みトランザクションです。

プログラムでは、Oracle Rdb によって暗黙的にトランザクションを終了できますが、すべての変更を永続的なものにする場合は COMMIT 文を発行し、すべての変更を取り消す場合は ROLLBACK 文を発行して明示的にトランザクションを終了することもできます。16.8 項 (16-42) では、トランザクションの終了に関するオプションについて説明します。

図 16-2 (16-22) に、SET TRANSACTION 文によるトランザクション・スコープを示します。SET TRANSACTION 文は、プログラム内でも対話型 SQL と同様に動作します。つまり、トランザクションの指定と開始を行います。

図 16-2 SET TRANSACTION 文によるトランザクション・スコープ

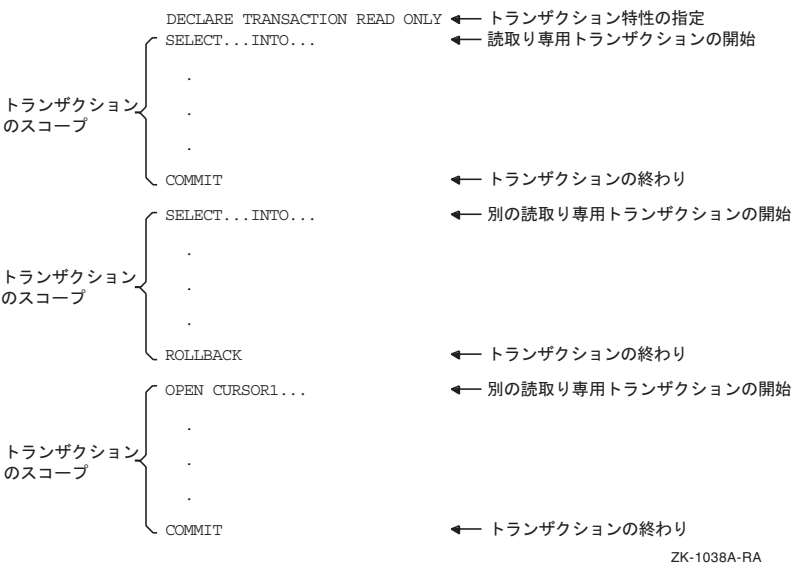


DECLARE TRANSACTION 文は SET TRANSACTION 文以外の SQL 文で暗黙的に開始されるトランザクションのデフォルト値を確立します。たとえば、OPEN 文でトランザクションを開始すると、このトランザクションは DECLARE TRANSACTION 文の特性を継承します。しかし、SET TRANSACTION 文でトランザクションを開始すると、SQL は DECLARE TRANSACTION 文を無視します。未指定のトランザクション特性は、Oracle Rdb のデフォルト値になります。

DECLARE TRANSACTION 文は、セッションまたはプログラムで Oracle Rdb のデフォルト値とは異なるデフォルト値を使用する場合には便利です。たとえば、データベースに対するクエリーを実行し、すべてのクエリーを読取り専用にして行ロックを回避する場合は、セッションまたはプログラムの先頭で 1 度だけ DECLARE TRANSACTION READ ONLY を指定すれば、クエリーで行ロックが回避されます。一方、SET TRANSACTION 文で READ ONLY を指定する場合は、COMMIT 文か ROLLBACK 文の後にこの文を挿入すると、後続のクエリーが確実に読取り専用トランザクションとして処理されます。

図 16-3 (16-23) に、SQL プリコンパイラまたは SQL モジュール・プロセッサが受け取ったファイル内の DECLARE TRANSACTION 文によるトランザクション・スコープを示します。

図 16-3 DECLARE TRANSACTION 文によるトランザクション・スコープ

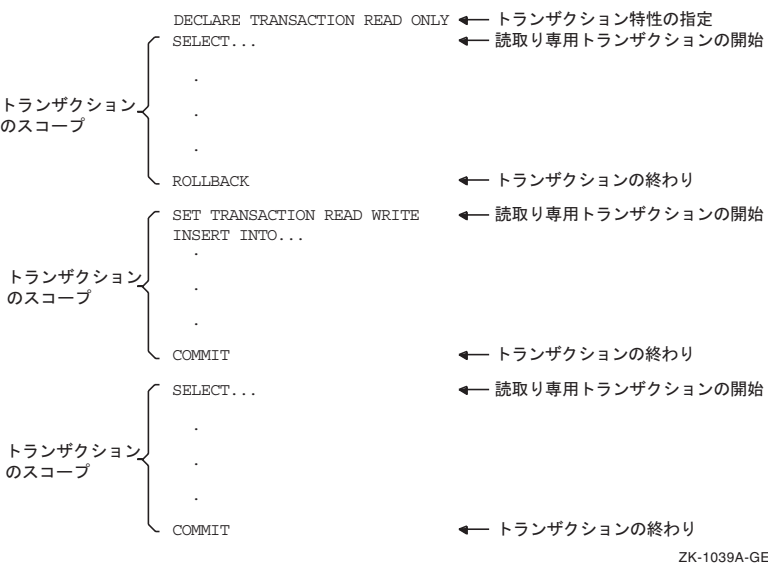


1 つの `DECLARE TRANSACTION` 文と複数の `SET TRANSACTION` 文の組合せにも注目してください。ただし、トランザクション管理のアプローチには豊富なバリエーションがあるので、プログラム内の様々なポイントで不適切なトランザクションを開始する可能性も高まります。一般に、必要なアクセスの種類、すなわち使用するテーブル、共有モード、ロック・タイプ、分離レベルを明示的に指定してください。限定的に指定するほど、データベース操作の効率と保守性は向上します。

参考： `DECLARE TRANSACTION` 文と `SET TRANSACTION` 文の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

図 16-4 (16-24) に、`DECLARE TRANSACTION` 文と `SET TRANSACTION` 文の組合せを示します。

図 16-4 SET TRANSACTION 文と DECLARE TRANSACTION 文によるトランザクション・スコープ



16.4 分散トランザクションの使用法

分散トランザクションでは、複数のデータベースまたは複数のデータベース接続が1つのトランザクションに統合されます。データベースは同一のノード上にあっても別々のノード上にあってもかまいません。トランザクションでは、トランザクション内の1つの操作が完了しない場合は他のどの操作も完了しないことが保証されます。しかし、通常のマルチデータベース・トランザクションでは、1つのデータベースは正常にコミットされ、別のデータベースは失敗する場合があります。

SQL では、アプリケーションで複数のノード上にある複数のデータベースに接続してもデータの整合性を保証するために2フェーズ・コミット・プロトコルを使用しており、分散トランザクションの場合にもトランザクションの個々の操作がすべて完了しないとトランザクションをコミットしないことが保証されます。

OpenVMS VAX
OpenVMS Alpha

OpenVMS では、トランザクションが複数のノード上にある複数のデータベースにアクセスし、システム上で DECdtm サービスがアクティブな場合は、Oracle Rdb はデフォルトで2フェーズ・コミット・トランザクションを開始します。◆

2フェーズ・コミット・トランザクションは、まずトランザクション内の各操作がコミットできるかどうかについてクエリーを実行します。1つの操作が失敗すると、トランザクシ

ン全体が失敗します。個々の操作がすべてコミットできる場合にのみ、最終的にコミットが行われます。

2 フェーズ・コミット・トランザクションと分散トランザクションについては、『Oracle Rdb7 Guide to Distributed Transactions』を参照してください。

アプリケーションで分散トランザクションを使用しない場合にハードウェアまたはソフトウェアの問題でシステムに障害が発生すると、不完全なトランザクションはシステムを再起動したときに自動的にロールバックされます。トランザクションがロールバックされる場合とコミットされる場合については、16.8 項 (16-42) に詳しい説明があります。

16.5 データベース・リソースのロック

ロックは、並列処理の制御とデータベースの論理的および物理的な整合性を強化するために、Oracle Rdb が使用するメカニズムです。ロックには、次のような目的があります。

- 物理的なデータベースへのアクセスを同期化します。たとえば、テーブルやインデックスなどのページや論理領域を、使用時にロックします。
- イベントを追跡します。たとえば、モニターは、プロセス終了ロックを使って、ユーザーがデータベースへのアクセスを終了したことを判断します。
- データベースのビューの論理的な整合性を維持します。

プログラムは、明示的にロックを制御しません。ロックの制御は行いませんが、必要な保護とアクセスを指定します。Oracle Rdb は、ユーザーのニーズに対応できるロック構造を決定すると同時に、ほとんどのユーザーによる同時アクセスを可能にし、データベースの整合性を維持します。Oracle Rdb が実際に取得するロックは、ユーザーが想定しているロックや、ユーザーが要求した通りの内容ではない場合があります。

次の項では、読取り / 書込みトランザクションでのロックについて説明します。読取り専用トランザクションは、スナップショットを有効化している場合、テーブルや行をロックしません。スナップショットを無効化すると、読取り専用トランザクションでも、読取り / 書込みトランザクションと同じようにロックが発生します。16.5.4 項 (16-30) では、読取り専用トランザクションとスナップショット・ファイルについて説明しています。

16.5.1 ロックの方法

Oracle Rdb は、次のように、簡単な方法でオブジェクトをロックします。

- オブジェクトをロックします。
- オブジェクトの操作を行います。たとえば、ディスクからの物理ページの取出しや行の更新などがあります。

- その後、トランザクションの終了時や、他の条件でロックを解放できる時点で、オブジェクトのロックを解放します。たとえば、他のトランザクションが同じ物理ページから別の行を読み取る必要がある場合などに、物理ページのロックを解放します。

一般的に、Oracle Rdb は、できる限り高いレベルでロックしようとします。たとえば、500,000 行のテーブルから 1 行を選択する場合、テーブルを使用しているトランザクションが他になければ、Oracle Rdb は、テーブル全体をロックすることもあります。したがって、他の行を選択する場合に、新しくロックを取得する必要がありません。他のトランザクションがテーブルに互換性のあるアクセスを要求している場合にのみ、インデックス・ノードや個々の行など、テーブルの一部をロックします。これは、**Adjustable Lock Granularity (ALG)** と呼ばれます。

16.5.2 明示的ロック

デフォルト・トランザクションを開始する場合、または特殊なアクセスや共有モード用にテーブルを明示的に予約すると、Oracle Rdb は、テーブル単位でロックを取得します。RESERVING 句を指定しないトランザクションでは、Oracle Rdb は、SQL 文でテーブルをアクセスするまでは、テーブルをロックしません。テーブルへのアクセス時に、Oracle Rdb は、実行中の処理内容に応じてロックを選択します。明示的にテーブルを予約すると、Oracle Rdb は、トランザクションの開始時に各テーブルをロックし、テーブルに**明示的ロック**を設定します。

表 16-2 (16-26) に、明示的ロック・モードとその意味、および実行可能な操作を示します。

表 16-2 明示的ロック

ロックのタイプ 内容		実行可能な操作
CR	同時読取り (Concurrent Read)	現在のトランザクションは、読取りを実行する各レコードを明示的にロックします。他のトランザクションは、同じテーブルへの読取りや書込みが可能です。
CW	同時書込み (Concurrent Write)	現在のトランザクションは、読取りまたは書込みを実行する各レコードを明示的にロックします。他のトランザクションは、同じテーブルへの読取りや書込みが可能です。
PR	読取り保護 (Protected Read)	現在のトランザクションは、読み取るレコードをロックする必要はありません。他のトランザクションは、同じテーブルの読取りは可能ですが、書込みはできません。

表 16-2 明示的ロック（続き）

ロックのタイプ	内容	実行可能な操作
PW	書込み保護（Protected Write）	現在のトランザクションは、読み取るレコードをロックする必要はありませんが、変更や追加ではレコードをロックします。他のユーザーは、CR モードでの読取りが可能です。
EX	排他（Exclusive）	他のトランザクションによるテーブルへのアクセスは許可されないで、現在のトランザクションは、レコードをロックしないで読取りや変更を実行します。他のトランザクションは、このトランザクションが処理を終了し、テーブルにアクセス可能になるまで待機しなくてはなりません。

デフォルトでは、各トランザクションは、トランザクションの終了まですべてのロックを保持します。ロックされた行を、他のトランザクションが変更することはできません。多くの場合、同じ行の読取りも許可されません。さらに、テーブルの構造や Oracle Rdb によるクエリーの処理方法によっては、トランザクションが直接要求していないレコードをロックすることも可能です。たとえば、テーブルを 1 行ずつ検索して、ある行を探す場合、テーブル全体を一度に 1 行ずつロックします。また、トランザクションは、制約やトリガーの参照先のテーブルをロックすることもあります。このようなロックは、トランザクションが終了するまで保持されます。トランザクションがレコードをロックしている間、他のトランザクションは、使用方法に整合性がないとレコードの使用は許可されません。

同様に、他のトランザクションが、行、テーブル、インデックスをすでに使用している場合、そのリソースが解放されるまで待機するか、ロックの競合エラーを処理する必要があります (16.5.3 項 (16-28) では、ロックの競合について説明しています。10.5 項 (10-32) では、ロックの競合やデッドロックをプログラム内で処理する方法について説明しています)。

予約するすべてのテーブルに対してアクセスを取得できないと、トランザクションは開始しません。

行に設定するロックの種類（読取りまたは書込み）は、行を処理する文の特性によって決まります。たとえば、OPEN 文は、RESERVING 句でテーブルに書込みのロック・タイプを指定しても、カーソルの行に読取りロックだけを設定します。オープンしたカーソルから行をフェッチしたら、Oracle Rdb が、読取りロックを書込みロックにアップグレードする前に、UPDATE または DELETE 文を実行する必要があります。

保護モードや排他モードでテーブルを予約すると、他のユーザーによるデータの使用を制限できるので、トランザクションが完了するまでの時間は短縮されます。たとえば、排他的共有モードでは、他のユーザーはテーブルにアクセスできません。他のユーザーがテーブルを

使用できないと、これから処理しようとしている行を、他のユーザーのトランザクションがロックするのを防止できます。

共有モードとロック・タイプの指定が重要な意味を持つのは、読取り / 書込みトランザクションの開始時のみです。読取り専用やバッチ更新トランザクションでは、あまり重要ではありません。

16.5.3 ロックの競合

行をロックすると、図 16-5 (16-29) で示すように、他のユーザーとの競合が発生する場合としない場合があります。競合が発生した場合、2 番目のトランザクションで NOWAIT オプションが指定されていると、このトランザクションはすぐにエラーを受け取ります。2 番目のトランザクションで WAIT オプションが指定されていると、次のいずれかの状態まで待機します。

- 最初のトランザクションがロックを解放。この時点で、2 番目のトランザクションが処理可能になります。
- デッドロックの検出。この時点で、いずれかのユーザーがデッドロック・エラーを受け取り、他のユーザーは処理を続行します。
- タイムアウト間隔の経過。この時点で、2 番目のトランザクションはエラーを受け取ります。

図 16-5 (16-29) は、どのような場合に、行ロックの解放まで待機するか、またはロックの競合エラーが発生するかを示しています。他のユーザーのトランザクションには、WAIT 特性（デフォルト）が指定されていると仮定します。NOWAIT を指定した場合は、表中の "a wait" は、"a conflict" になります（テーブルでは、いつユーザーにデッドロックが発生するかを認識できない点に注意してください。ロックの競合とは異なり、デッドロックは簡単に予測できません）。

他のユーザーは、トランザクションが終了してロックが解除されるまでの間、待機する必要があります。テーブルへの共有読取りアクセスを指定した読取り / 書込みトランザクションは、処理する行をすべて読取りロックします。したがって、一意のインデックス値を使用したアクセスではなく、テーブルの行を順次検索する場合、テーブルのすべての行がロックされます。他のユーザーは、トランザクションが読取りロックを設定している行の更新はできないので、トランザクションが終了するまでは、どの行の更新もできなくなります。

図 16-5 データベース・アクセスの競合

トランザクション モードを使用して 行にアクセスする 場合	誰かが次のように使用している場合						
	READ ONLY の場合	READ WRITE SHARED READの場合	READ WRITE SHARED WRITEの場合	READ WRITE PROTECTED READの場合	READ WRITE PROTECTED WRITEの場合	READ WRITE EXCLUSIVE READの場合	READ WRITE EXCLUSIVE WRITEの場合
READ ONLY	競合なし	競合なし	競合なし	競合なし	競合なし	待ち	待ち
READ WRITE SHARED READ	競合なし	競合なし	読込みの競合 なし、更新待 ち	競合なし	読込みの競合 なし、更新待 ち	待ち	待ち
READ WRITE SHARED WRITE	競合なし	競合なし	更新されていない 行の読込みに は競合なし、そ れ以外は待ち	待ち	待ち	待ち	待ち
READ WRITE PROTECTED READ	競合なし	競合なし	待ち	競合なし	待ち	待ち	待ち
READ WRITE PROTECTED WRITE	競合なし	競合なし	待ち	待ち	待ち	待ち	待ち
READ WRITE EXCLUSIVE READ	待ち	待ち	待ち	待ち	待ち	待ち	待ち
READ WRITE EXCLUSIVE WRITE	待ちの後、 競合	待ち	待ち	待ち	待ち	待ち	待ち

ZK-1484A-GE

更新の場合、Oracle Rdb は、他の読取り / 書込みトランザクションに対して、更新トランザクションがコミットまたはロールバックするまでは、変更された行の読取りを許可しません。Oracle Rdb は、行をロックして、他のトランザクションによるアクセスを許可しないので、更新トランザクションは、行の変更内容を表示できます。このような行のロック方式は、データベース行に一貫性と整合性があることを前提としています。

テーブルへの同時アクセスを改善するには、カーソル宣言において、行の指定をできるだけ限定してください。カーソルをオープンするとすぐに、トランザクションは、カーソルに関連付けられているテーブルのすべての行（インデックスによるテーブル・アクセスでない場合）と、他の行に対して、読取りロックを設定します。読取り / 書込みトランザクションでは、多数の行にアクセスするカーソルをオープンすると、トランザクションのコミットまたはロールバックが完了するまでは、他のユーザーはデータベースを更新できなくなります。たとえば、あるアプリケーションを記述し、このアプリケーションでは、トランザクションの開始、カーソルのオープン、データ入力者による行のチェックと更新および削除を実行し、すべての変更内容を一度にコミットする場合、このアプリケーションの起動後に開始した読取り / 書込みトランザクションは、検索操作でさえ実行できなくなります。一般的に、カーソルをオープンして対話型更新を実行する場合は、1 行から数行に限定し、COMMIT 文または ROLLBACK 文でできるだけ早くクローズしてください。（CLOSE 文を実行しても、ロックは解放されません）。

テーブルを共有アクセスで要求しても、データベースの一貫性や整合性を保つために必要であれば、Oracle Rdb は、排他アクセスでテーブルをロックすることがあるので注意してください。これは、データ定義文などの文で発生します。

16.5.4 読取り専用トランザクションとスナップショット・ファイル

スナップショット・ファイルを有効化すると（デフォルト）、読取り専用トランザクションは、読取りで行をロックしません。更新では、Oracle Rdb は、古いバージョンである **Before-image** をスナップショットに書き込みます。読取り専用トランザクションは、**Before-image** をスナップショット・ファイルから直接読み取ります。

行が何回か更新されると、複数の **Before-image** がスナップショット・ファイルに書き込まれます。**Before-image** はすべて保持されるので、他のトランザクションは、この更新トランザクションが開始した時点でのレコードを読み取ることができます。

しかし、スナップショット・ファイルを無効化した状態で、読取り専用トランザクションが Oracle Rdb データベースにアクセスすると、トランザクションを実行中、処理する行を読取りロックします。スナップショット・ファイルが存在しない場合（しかも、読取り専用の記憶領域にアクセスしていない場合）、読取り専用トランザクションは、読取り / 書込みトランザクションと同じように処理されます。

読取り専用の記憶領域では、データを更新できないので、スナップショット・ファイルは存在しません。

16.5.5 読取り専用トランザクションでのロックの競合エラー

排他的共有モードでテーブルを予約するトランザクションは、Recovery-unit ジャーナル・ファイルを更新しますが、スナップショット・ファイルは更新しません。スナップショット・ファイルの更新処理が不要なので、更新に伴うディスク I/O 操作が減り、読取り / 書込みトランザクションのパフォーマンスは改善されます。ただし、読取り専用トランザクションを実行するには、スナップショット・ファイルの更新が必要なので、排他的共有モードでテーブルを予約するトランザクションは、同じテーブルにアクセスする読取り専用トランザクションを無効化します。トランザクションの優先順位が高い場合、他の読取り / 書込みトランザクションとの競合を最小限にし、読取り専用トランザクションとの競合が発生しないようにするには、書込み保護でテーブルを予約してください。

スナップショット・ファイルにアクセスする読取り専用トランザクション（通常は、まったくロックは発生しません）にも、ロックの競合エラーが発生することがあります。これは、読取り / 書込みトランザクションが排他的書込みオプションを使って予約しているテーブルに、読取り専用トランザクションがアクセスした場合や、バッチ更新トランザクションがデータベースのテーブルにアクセスした場合に発生します。（バッチ更新トランザクションは、排他的書込みオプションを使って、データベース内のすべてのテーブルを予約します）。読取り専用トランザクションでこのような競合が発生すると、読取り専用トランザクションは、NOWAIT オプションを指定した状態でロックの競合エラーが発生したトランザクションとして処理されます。

次に、読取り専用トランザクションと、排他的書込みモードでテーブルを予約した読取り / 書込みトランザクションの間に競合が発生するしくみを示します。

1. ユーザー A が、EXCLUSIVE WRITE でテーブルを予約し、いくつかの行をフェッチします。
2. ユーザー B は、読取り専用の WAIT トランザクションを、RESERVING 句を指定しないで開始します。
3. ユーザー B が、ユーザー A が予約したテーブルにアクセスしようとしています。
4. ユーザー B は、すでに開始している排他的書込みトランザクションがコミットまたはロールバックするまで待機し、その後ロックの競合エラーを受け取ります。
5. ユーザー A が現在のトランザクションを終了し、すべてのロックを解放して SQL を終了した後も、ユーザー B のトランザクションは、同じトランザクション内でテーブルにアクセスすると、エラーが発生します。

トランザクションで EXCLUSIVE WRITE を指定すると、スナップショット・ファイルにはデータを書き込みません。ただし、読取り専用トランザクションが開始する前に、データベースにコミットされたすべてのデータを、読取り専用トランザクションでも使用可能な状態にしておく必要があります。Oracle Rdb は、排他的共有モードのトランザクションがスナップショット・ファイルにデータを書き込んだかどうかを判断できないため、読取り専用トランザクションの要件は満たされません。したがって、ユーザー B の WAIT オプションは拒否され、ユーザー B は、永久に待機するのではなく、エラーを受け取ります。

16.5.6 同時アクセスの改善

値が一意のインデックスを使用する場合や、重複値があっても適切に設計されているインデックスを使用する場合には、次のような条件が満たされれば、1つのテーブルに対するマルチユーザー・アクセスを改善できます。

- テーブルのサイズが大きい
- ユーザーはテーブルに同時アクセスするが、異なるインデックス値を使用する

テーブルへの同時アクセスを向上するには、順次（インデックスを使用しない）検索ではなく、一意のインデックスでテーブルを検索します。インデックスを使用しない検索では、テーブルの先頭から最後まで行を読み、各行に読取りロックを設置します。インデックスが一意であれば、同じテーブルを検索する場合でも、1つの値を使用した検索が他の値を使用した検索と競合することはありません。一意のインデックス値で行を検索するときには、必要なテーブル行にのみ読取りロックを設定します。

ただし、インデックスはデータベース構造であり、テーブルの行がユーザー・アクセスでロックされるのと同様に、エントリ・ポイント（ノード）がユーザー・アクセスでロックされます。したがって、ソート・インデックスのノードがロックされているために、トランザクションでロックの競合エラーが発生したり、他のトランザクションが終了するまで待機し

なければならないことがあります。このようなロックは、デッドロック・エラーの一因になります。

ソート・インデックスは、このような競合が発生しやすいオブジェクトです。一意のインデックス値には、ID 番号、タイムスタンプ、注文番号などが格納され、値は1ずつ増加することが多く、最新行へのアクセスが頻繁に発生します。同じインデックス・ノードに、必要なレコードが複数格納されている可能性も非常に高くなります。ハッシュ・インデックスは、データベースへ均等に分散されるので、競合が発生する可能性は低くなります。ただし、ハッシュ・インデックスは、マルチファイル・データベースでしか定義できません。インデックスを分割し、アプリケーションが別々のノード群にアクセスできるようにする方法でも、アクセスを向上できます。

重複値を持つソート・インデックスを使用したテーブル検索では、テーブルの順次検索と同様に、他のユーザーによるアクセスを効率的に排除できます。ただし、ソート・インデックスの構築方法によっては、インデックス値の数が少ないと問題が発生します（部門コードをベースにしたインデックスで、部門コードが3つしかない場合など）。このような場合、ソート・インデックス構造のノード数が少ないため、1人のユーザーがすべてのノードをロックしてしまう可能性があります（1つのインデックス値のみを検索する場合）。

このような問題を回避するには、ソート・インデックスの重複を少なくしてください。この方法の1つとして、ベースにする列を複数に増やします。たとえば、部門コードの値の数が少ない場合、部門コードと管理者識別番号（Identification Number : ID）の両方をベースにすると、ソート・インデックスの値の数が増えるので、インデックスのノード数も増加します。他にも、ソート・インデックスの他に、ハッシュ・インデックスを作成する方法があります。Oracle Rdb は、クエリーが範囲検索か完全一致検索かによって、使用するインデックスを選択することに注意してください。

特に、クエリーに結合操作が含まれる場合や、複数のテーブルを使用したビューにアクセスする場合は、テーブルが順次検索されるかインデックス検索されるかを予測できないので、注意が必要です。Oracle Rdb クエリー・オブティマイザは、どのテーブルを最初に検索するか、順次検索かインデックス検索のどちらを実行するか、特殊なインデックスを使用するか（ハッシュ・インデックスとソート・インデックスのどちらも使用可能な場合）、などを判断します。クエリー・オブティマイザは、様々な判断基準を使いますが、結合するテーブルの相対的なサイズもこの基準の1つです。

あるテーブルを最初に検索すれば、結合操作を最も速く実行でき、そのためにはテーブルのすべての行をロックする必要がある、とクエリー・オブティマイザが判断した場合、トランザクションは、そのテーブルのすべての行に読取りロックを設定します。他のユーザーは、トランザクションが終了するまで、テーブルの行を更新できません。逆に、他のトランザクションがすでにテーブルのロックを取得しているときは、ロックの競合エラーが発生したり、待機が必要になることがあります。

たとえば、クエリー・オブティマイザが次のクエリーを評価するのに、1つのテーブルの各行を読み取り、開始値を取得し、この値で他のテーブルをチェックする場合などがあります。クエリー・オブティマイザは、EMPLOYEES テーブルが一番小さいので、EMPLOYEES テーブルを使う方法が最も速いと判断するかもしれません。逆に、最初に大きなテーブルを

検索し、終了日の値が NULL でない行を除外する方法が、全体的なクエリー・パフォーマンスの点で最適だと判断するかもしれません。いずれの場合も、次のクエリーは、1 つ以上のテーブルの各行に読取りロックを設定することを理解しておいてください。

```
SELECT LAST_NAME, FIRST_NAME, JOB_CODE, JOB_START, SALARY_AMOUNT
FROM EMPLOYEES E, JOB_HISTORY JH, SALARY_HISTORY SH
WHERE (
    (JH.EMPLOYEE_ID = E.EMPLOYEE_ID)
    AND
    (SH.EMPLOYEE_ID = JH.EMPLOYEE_ID)
)
AND (JOB_END IS NULL)
AND (SALARY_END IS NULL);
```

このように、どのオブジェクトにどのようなロックが設定されるかという点では、アプリケーションは間接的な制御しかできません。すべてのテーブルをインデックスを使ってアクセスする方法が、最適な方法です。すべてのテーブルに存在するインデックス列でテーブルを結合すると、クエリー・オプティマイザは、アクセス方法を選択できるので、強制的に特定のテーブルを順次検索することはなくなります。

インデックスの詳細は、『Oracle Rdb7 Guide to Database Design and Definition』を参照してください。ロックの問題の特定とデバッグ方法、クエリー・オプティマイザで可能な処理方法の詳細、問題を解消する方法については、『Oracle Rdb7 Guide to Database Performance and Tuning』を参照してください。

16.6 端末 I/O 操作のないトランザクションの設計

多くのプログラムでは、対話型ユーザーがデータベースの検索と更新に使用する値を指定します。さらに、多くの対話型ユーザーが1つのプログラム・イメージを同時に実行している場合があります。このような場合、ユーザーがトランザクション時間を使用してエラー・メッセージを読み取り、入力を修正するのは望ましくありません。したがって、1人のユーザーがデータベース操作実行中に他のユーザーが変更を行わないことを保証するよりも、すべてのユーザーのトランザクションをできるだけ短く保持するよう判断することがあります。この選択は、ユーザーが、他のユーザーの干渉によって許容不可能な待機時間やロック・エラーに直面しないための最適な方法です。また、このようリスクがあるので、多くのユーザーが同じテーブル内のデータを同時に更新する可能性が高い場合に、対話型のユーザーが許可されるのは、トランザクションごとにテーブル内の1行の追加、変更、または削除に限定されます。

データ更新のタスクには、多くの場合、テーブル検索を伴うので、行とインデックスに対して読取りのロックをかけてから書込みのロックをかけます。1人のユーザーのために別のユーザーが長時間の待機かロック競合エラーやデッドロック・エラーに直面する場合、一般に次の状況で問題が発生しています。

- 1人のユーザーが行かインデックス・ノードに関する書込みのロックを保持しており、別のユーザーが読取りのアクセスを要求する場合

- 1 人のユーザーが行かインデックス・ノードに関する読取りのロックを保持しており、別のユーザーが書き込みのアクセスを要求する場合

対話型のユーザーにトランザクションごとにテーブル内の唯一の行の追加、変更、または削除を許可すれば、データの検索を妨害するロックの回数削減と時間短縮の両方を実現できます。唯一の行のトランザクションでは、動詞実行時の制約評価を使用すると不要な書き込みのロックが削減されます。動詞実行時の制約評価を指定すると、プログラムの要素には処理がない場合も、制約違反となる行の変更は自動的に取り消されます。

ユーザーが検証のために行を取得した後、または致命的エラーの後は、トランザクションをロールバックすることで、端末 I/O 操作時間をトランザクションに含めないようにします。この結果、行またはインデックス・ノードの読取りロックの時間が短縮されるので、書き込み操作とのロック競合が発生しにくくなります。

端末 I/O 操作時間をトランザクション外に保持するのは困難な場合があります。多くの場合、ユーザーに対して、ロールバック操作の後の行のすべての入力値を再入力するよう要求するのは困難だからです。ユーザー・エラーの後にトランザクションをロールバックする場合、最初のユーザーが正しい値を入力する前に、ユーザーが処理している行を別のユーザーが削除または更新するという可能性（リモートで）に対処する必要もあります。

この項では、2 つの疑似コードの例を示し、他のユーザーが行う操作による悪影響というリスクと、トランザクションを短時間に保持するというニーズとのかね合いをどう調整するかを説明します。

例 16-1 (16-34) に、現在多くのユーザーがアクセスしているテーブル (EMPLOYEES) の行を更新する方法を示します。例のロジックでは、行検索が 2 回実行されます。最初がインデックス値による検索、次がデータベース・キー (dbkey) による検索です (インデックス値でなく dbkey の値で検索すると、インデックス・ノードが更新されるので、書き込みロックに直面することはありません)。例では、2 人のユーザーが同時に同じ行を更新することはないと仮定しています。アプリケーションで実行する操作に関してこの条件を仮定できない場合は、ユーザーのタスクにマルチ・トランザクションのアプローチを使用しないでください。それ以外の場合は、大幅な変更が必要です。

例 16-1 マルチユーザー環境における行の更新

```
Set the flag to indicate that EMPLOYEE_ID is invalid and initialize
other parameters.
Perform VERIFY_EMPLOYEE section until a valid EMPLOYEE_ID is obtained.
Perform UPDATE_EMPLOYEES section.
VERIFY_EMPLOYEE section:
Prompt the user for EMPLOYEE_ID value to determine which employee row
needs an update.
Verify that input characters represent a valid string for an employee
identification number; prompt again if necessary.
SET TRANSACTION READ WRITE RESERVING EMPLOYEES FOR SHARED READ
Check the status parameter to ensure that the transaction started
successfully; if not:
Handle deadlock and lock conflict with timed retry of the
```

```
SET TRANSACTION statement, or
Stop the program with appropriate messages for other errors.
Select the row in the EMPLOYEES table that has the input ID number.
Check the status parameter to ensure that the operation was successful;
    if not:
        Handle deadlock and lock conflict with the ROLLBACK statement and
        the timed retry of the SET TRANSACTION statement and row retrieval,
        or display messages, roll back, and stop the program for
        unexpected errors.
If no row is found:
    Roll back.
    Tell the user that the entered employee number is not assigned
    to any employee and to enter a new value.
If row is found:
    Set a predefined flag to indicate the EMPLOYEE_ID exists.
    Store the dbkey value for the row in a parameter.
    Store the row in the first of three identical sets of program
    parameters.
    Roll back.
    Display the row values stored in the parameters on the terminal.
UPDATE_EMPLOYEES section:
Prompt for change values and store them in the second set of parameters
for the row.
Verify that valid characters are entered for all entries and prompt
for corrections if necessary.
SET TRANSACTION READ WRITE RESERVING EMPLOYEES FOR SHARED WRITE
Check the status parameter to ensure that the transaction started
successfully; if not:
    Handle deadlock or lock conflict with timed retry of the
    SET TRANSACTION statement, or
    Stop the program with appropriate messages for other errors.
Retrieve the row by the dbkey value and store it in the third set
of parameters for the row.
If no row found:
    Roll back.
    Tell the user that the row was unexpectedly deleted by
    another user.
    Exit the program or loop.
If row is found:
    Compare the values in the first and third sets of parameters
    to determine if another user has changed row.
    If yes:
        Roll back.
        Tell the user that an unexpected change occurred to the row.
        Stop the program.
    If no:
        Update the row, setting column values to those input by the
```

```
user.  
Check the status parameter to ensure that the operation was  
successful; if not:  
    Handle lock conflict or deadlock with ROLLBACK and  
    timed retry of the entire transaction.  
    Stop the program for unexpected errors.  
COMMIT
```

例 16-2 (16-36) では、JOB_HISTORY テーブルに従業員の新しい仕事に関する情報を記述した行を挿入し、従業員の最新の仕事に関する情報を格納した行を更新することで、従業員の職務経歴情報を更新します。この問題は、テーブル間の制約があるので、例 16-1 (16-34) に示す問題とは異なります。この例では、対話型のユーザーが多く、多くの制約に違反する入力エラーを犯した場合に、多くの短い読取り / 書込みトランザクションが発生します。しかし、端末 I/O 操作を行うトランザクションがないので、ユーザーは無効な値についてのみ入力を再要求されます。

例 16-2 制約のあるテーブルの更新

```
Perform INITIALIZE_FLAGS section.  
Perform VERIFY_EMPLOYEE section until a valid EMPLOYEE_ID is obtained.  
Perform VERIFY_ROW_VALUES section until valid SUPERVISOR_ID, JOB_CODE,  
    and DEPARTMENT_CODE values are obtained.  
Perform UPDATE_JOB_HISTORY section.  
INITIALIZE_FLAGS section:  
Initialize all flags to indicate "invalid". (Flags are set to "valid"  
    when values input by the user have passed program checks.)  
VERIFY_EMPLOYEE section:  
Prompt the user for the EMPLOYEE_ID value to determine for which  
    employee the job history information needs an update.  
Verify that input characters represent a valid string for an  
    employee identification number; prompt again if necessary.  
SET TRANSACTION READ WRITE RESERVING EMPLOYEES FOR SHARED READ  
Check the status parameter to ensure that the transaction started  
    successfully; if not:  
    Handle deadlock and lock conflict with timed retry.  
    Stop the program for unexpected errors.  
Select the row in the EMPLOYEES table that has the input ID number.  
Check the status parameter to ensure that the operation was successful;  
    if not:  
    Handle deadlock and lock conflict with ROLLBACK and timed  
    retry of the statement.  
    Roll back.  
    Stop the program for unexpected errors.  
If no row is found:  
    Roll back.  
    Tell the user that the entered employee ID number is not yet  
    assigned to any employee in the database; therefore, no update
```

```

        to the JOB_HISTORY table using that number is allowed.
    If a row is found:
        Set a predefined flag to indicate EMPLOYEE_ID exists.
        Perhaps display the employee ID number and name for the user.
        Roll back.
    VERIFY_ROW_VALUES section:
    Prompt for the JOB_START date and for any of the following values
        whose associated flag is set to "invalid":
        JOB_CODE, DEPARTMENT_CODE, SUPERVISOR_ID.
    Verify that valid characters have been entered for first or revised
        entries.
    Prompt for corrections if necessary.
    If the flag for SUPERVISOR_ID is set to "invalid":
        SET TRANSACTION READ WRITE
        RESERVING EMPLOYEES FOR SHARED READ
        Check the status parameter to ensure that the transaction
        started successfully; if not:
            Handle deadlock and lock conflict with timed retry of
            the SET TRANSACTION statement.
            Stop the program for unexpected errors.
            Select the row in the EMPLOYEES table that has the
            input supervisor ID.
        Check the status parameter to ensure that the operation was
        successful; if not:
            Handle deadlock and lock conflict with ROLLBACK and
            timed retry of the statement.
            Stop the program for unexpected errors.
        If a row is found, roll back and set the predefined flag
        to indicate that the SUPERVISOR_ID is valid.
        If no row is found, roll back.
    If the flag for DEPARTMENT_CODE is set to "invalid":
        SET TRANSACTION READ WRITE
        RESERVING DEPARTMENTS FOR SHARED READ
        Check the status parameter to ensure that the transaction
        started successfully; if not:
            Handle deadlock and lock conflict with timed retry of
            the SET TRANSACTION statement.
            Stop the program for unexpected errors.
        Select the row in the DEPARTMENTS table that has the input
        department code.
        Check the status parameter to ensure that the operation was
        successful; if not:
            Handle deadlock or lock conflict with ROLLBACK and
            timed retry of the statement.
            Stop the program for unexpected errors.
        If a row is found:
            Roll back.

```

```
        Set the predefined flag to indicate that DEPARTMENT_CODE
        is valid.
    If no row is found, roll back.
If the flag for JOB_CODE is set to "invalid":
    SET TRANSACTION READ WRITE RESERVING JOBS FOR SHARED READ
    Check the status parameter to ensure that the transaction
    started successfully; if not:
        Handle deadlock and lock conflict with timed retry of
        the SET TRANSACTION statement.
        Stop the program for unexpected errors.
    Select the row in the JOBS table that has the input job code
    value.
    Check the status parameter to ensure that the operation was
    successful; if not:
        Handle deadlock and lock conflict with ROLLBACK and
        timed retry of the statement.
        Stop the program for unexpected errors.
    If a row is found:
        Roll back.
        Set a predefined flag to indicate that JOB_CODE is
        valid.
    If no row is found, roll back.
If any flag indicates an input code or ID is still "invalid":
    Display messages that specify input values that failed
    verification checks.
UPDATE_JOB_HISTORY section:
SET TRANSACTION READ WRITE
        RESERVING EMPLOYEES FOR SHARED READ,
        JOBS FOR SHARED READ,
        DEPARTMENTS FOR SHARED READ,
        JOB_HISTORY FOR SHARED WRITE
Check the status parameter to ensure that the transaction started
successfullu; if not:
    Handle deadlock and lock conflict with timed retry of
    the SET TRANSACTION statement.
    Stop the program for unexpected errors.
Update the current JOB_HISTORY row where EMPLOYEE_ID is equal to
the input value for EMPLOYEE_ID and where JOB_END is null,
setting JOB_END equal to the value input for JOB_START.
Check the status parameter to ensure that the operation was successful
or a row was not found (an acceptable condition for newly hired
employees); if not:
    Handle lock conflict and deadlock with ROLLBACK and timed
    retry of the statement.
    Roll back and stop the program for unexpected errors.
Insert into the JOB_HISTORY table a new row using all input values
and setting JOB_END to null.
```



```

Check the status parameter to ensure that the operation was successful;
if not:
    Handle lock conflict and deadlock with ROLLBACK and timed
    retry of the statement.
    Roll back and stop the program for unexpected errors.

COMMIT
Check the status parameter to ensure that the COMMIT was successful;
if not:
    Display a message saying an unexpected error occurred.
    Roll back and stop the program.

```

例 16-2 (16-36) は、マルチ・トランザクションの方法では対話型ユーザーが他のユーザーによるロックの解放を長時間待つというリスクを最小にすることを示しています。同じテーブル・セットへのアクセスを必要とするすべてのトランザクションは、共有共有モードで各テーブルにアクセスし、アクセス時間をできるだけ短く保持する必要があります。この方法では、プログラムで値の存在を確認した後にだれかがリモートでそれを削除する可能性だけが想定されています。(データ入力担当者がある従業員に関する新しい仕事の情報をファイルに追加しようとしたときにコードの値が古くなる場合は、異常があります)。

データベース内の情報の種類、同時ユーザーの数、サイトで要求されるデータ・アクセス・パターンが指定された場合は、このアプリケーションに同じ方法を適用できません。たとえば、制約違反が発生した場合は、ユーザーに強制的に初めからやり直しをさせるのが適切です。また、待機時間が長くなってもデッドロックの回数を減らすことが重要な場合は、テーブル共有モードを保護モードにアップグレードするのが適切です。さらに、プログラム・ロジックではすべての問題を解決できません。たとえば、データを更新するユーザーが同じテーブルに関するレポート作成プログラムの長時間の読取り / 書き込みトランザクションと競合する場合は、大量のデータが更新アクセス不可能になるので、待機時間が許容不可能な長さになる恐れがあります。このような場合、プログラムでは対話型ユーザーが長時間の読取り / 書き込みトランザクションに妨害される機会を確実に減らすデータベース操作スケジュールのサポートが必要になります。

16.7 制約を評価するタイミングの指定

データベース内のテーブルに制約が定義されている場合は、制約のあるテーブルを更新するすべてのトランザクションで制約を評価する必要があります。一部の更新トランザクションについて制約を完全に無効化し、他のトランザクションについて制約評価を再び有効化することはできません。ただし、Oracle Rdb ではユーザーは次のタイミングで制約を評価することを指定できます。

- 動詞実行時。該当するすべての制約は、データベース・ファイルに書き込まれる行ごとに書き込みが行われるときに評価されます。
- 即時。制約評価の原因となる SQL 文の終了時に制約が評価されます。
- コミット時。すべての書き込み操作がデータベース・ファイルに適用されるときに、該当する制約がトランザクション全体を通じて 1 度のみ評価されます。

16.7.1 項 (16-40) では、制約評価の様々なタイミングを指定する方法を説明します。

16.7.1 制約評価のタイミングの指定

制約を評価するタイミングは、次の方法で指定できます。

- テーブル作成時にデフォルトのタイミングを定義します。
- 動詞実行時またはコミット時の評価は、DECLARE TRANSACTION 文と SET TRANSACTION 文の EVALUATE 句で指定します。

さらに、コミット時の制約が実際に評価されるタイミングを次の方法で指定できます。

- SQL プリコンパイラを使用する場合は、
/SQLOPTIONS=(CONSTRAINT=IMMEDIATE|DEFERRED) または -s -'cm immediate|deferred' コマンド行修飾子を指定します。
SQLOPTIONS=(CONSTRAINT=IMMEDIATE) または -s -'cm immediate' 修飾子を指定すると、該当する個々の制約はただちに評価され、SET ALL CONSTRAINTS DEFERRED 文が発行されるまで、またはコミット操作かロールバック操作でトランザクションが完了するまで、各文の終了時にも評価されます。
- SQL モジュール・プロセッサを使用する場合は、
CONSTRAINT=IMMEDIATE|DEFERRED または -cm immediate|deferred コマンド行修飾子を指定します。
CONSTRAINT=IMMEDIATE または -cm immediate' 修飾子を指定すると、該当する個々の制約はただちに評価され、SET ALL CONSTRAINTS DEFERRED 文が発行されるまで、またはコミット操作かロールバック操作でトランザクションが完了するまで、各文の終了時にも評価されます。
- SET ALL CONSTRAINTS 文を指定します。
SET ALL CONSTRAINTS IMMEDIATE 文を指定すると、コミット時の制約のすべてがただちに評価され、SET ALL CONSTRAINTS DEFERRED 文が発行されるまで、またはトランザクションが完了するまで、各文の終了時とコミット時にも評価されます。

次の各項目は、制約を評価するタイミングと制約評価時の動作について説明しています。

- NOT DEFERRABLE 制約は、SET DEFAULT CONSTRAINT MODE 文または SET ALL CONSTRAINTS 文の指定にかかわらず、常に SQL 文終了時に評価されます。SET TRANSACTION 文の EVALUATING 句で NOT DEFERRABLE 制約をコミット時に評価するように指定すると、Oracle Rdb では例外が発生します。
- DEFERRABLE 制約は、制約モードが ON でない場合も、SET TRANSACTION 文で EVALUATE AT VERB TIME を指定しない場合も、COMMIT 文までの任意のタイミングで評価できます。
- 動詞実行時に評価される NOT DEFERRABLE 制約または DEFERRABLE 制約の違反がトランザクション実行中に発生すると、Oracle Rdb では COMMIT 文を発行できます。

COMMIT 文によって、制約に違反しない行を格納できます。制約に違反する行は格納されません。動詞実行時に評価される DEFERRABLE 制約の動作は、NOT DEFERRABLE 制約と同じです。

- コミット時に評価される DEFERRABLE 制約に違反する場合、Oracle Rdb では制約に違反するレコードを検出して制約に違反しない値を指定するまで COMMIT 文を発行できません。制約に違反したレコードの検出と修正ができない場合は、制約に違反しない更新も含めてトランザクションで試行した変更をすべてロールバックする必要があります。
- COMMIT 文実行時に制約違反が生成されるのは、コミット時の制約評価が有効な場合だけです。文に固有のエラー処理セクションを設計する場合は、COMMIT に関するエラー処理で制約違反を考慮する必要はありません。データベースに書き込みを行う SQL 文でエラーが返された場合も COMMIT 文でエラーが返された場合も同じエラー処理セクションに制御を渡す場合は、予期されるすべてのエラーを監視して処理する汎用のエラー処理セクションを記述してください。
コミット時に制約が評価されると、Oracle Rdb は何行処理されたかにかかわらず、最初に違反が発生したときにエラーを返します。制約に違反した行の検出と修正ができない場合は、制約に違反しない更新も含めてトランザクション全体をロールバックする必要があります。

言語が SQLV40 に設定された場合、デフォルトの制約属性は DEFERRABLE です。ただし、制約属性を指定しないと、推奨されない機能のメッセージが表示されます。言語が SQL92 に設定された場合、デフォルトの制約属性は NOT DEFERRABLE です。

16.7.2 制約を評価するタイミングに関する推奨事項

一般に、多くの更新操作を実行するトランザクションでは、Oracle Rdb は、次の理由で動詞実行時の制約評価を推奨しています。

- 更新が制約に違反するたびにただちにエラー・メッセージが表示されるので、制約違反のレコードの検出と修正が容易になります。
一方、コミット時に評価する制約に 1 つ以上のレコードが違反した場合、Oracle Rdb では唯一のエラー・メッセージが生成されます。この場合は、制約違反のレコードの検出が難しくなります。
- 更新するレコードの一部だけが制約に違反する場合は、制約違反のない他のレコードをすべてそのままコミットできます。
一方、コミット時に評価する制約では唯一のレコードが違反した場合にも、更新トランザクションで変更したすべてのレコードをロールバックする必要があります（制約に違反したレコードの検出と修正ができない場合）。

トランザクションで実行する変更が動詞実行時のコンテキスト（外部キーなど）を使用して参照テーブルのレコードを直接参照できる場合も、動詞実行時に制約を評価するのは好都合です。このコンテキストはコミット時には使用できません。したがって、場合によっては

Oracle Rdb が参照テーブルの各行をスキャンする必要があります。結果として、少数の更新が行われた場合にも、動詞実行時の制約評価はコミット時の評価よりも短時間で済みます。

Oracle Rdb では、次の場合にはコミット時の制約評価をお勧めします。

- 制約によって 2 つ以上のテーブルをチェックする可能性がある場合。たとえば、制約が次の関係に基づくと仮定します。

`MIN (A.NUMBER) > MAX (B.NUMBER)`

最小値を求めるために、Oracle Rdb はテーブル A とテーブル B の各レコードを読み込む必要があります。

- トランザクション実行中に 1 つ以上のレコードを繰り返し更新する場合。
- 相互依存性がある場合 (A が B に依存し、B が A に依存する)。相互依存性があると動詞実行時には制約を評価できません。この場合は、コミット時でなければ制約が機能しません。

データベースはコミット時に制約を評価するが、アプリケーションは ANSI/ISO SQL に準拠してソース・コードを変更したくない場合は、プリコンパイラまたはモジュール・プロセッサのコマンド行修飾子を使用してください。コマンド行修飾子を使用すると、コミット時の制約を各文の終了時に評価するように指定できます。

プリコンパイラまたはモジュール・プロセッサのコマンド行修飾子を使用してコミット時の制約を各文の終了時に評価するように指定する場合は、制約チェックを遅延させると制約に違反するデータを挿入することができます。この場合は、SET ALL CONSTRAINTS OFF 文を使用してください。

16.8 トランザクションのコミットとロールバック

トランザクションを終了するときは、データベースへの変更をコミットまたはロールバックする必要があります。COMMIT 文を使用するとデータベースへの変更が確定され、ROLLBACK 文を使用すると変更が取り消されます。

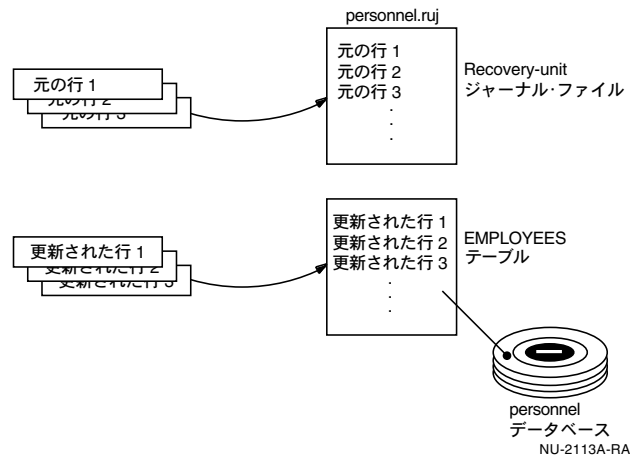
Oracle Rdb データベースのみを処理するトランザクションの場合は、COMMIT 文か ROLLBACK 文でトランザクションが終了します。トランザクション・マネージャを明示的にコールして分散トランザクションを開始した場合は、トランザクション・マネージャを直接コールしてトランザクションを終了する必要があります。もしトランザクション・マネージャを直接コールしなかった場合は、COMMIT 文または ROLLBACK 文でエラーが返されます。

サンプルの `personnel` データベースのエイリアスを宣言し、SET TRANSACTION 文を使って `EMPLOYEES` テーブルの書込みアクセスを予約する読取り / 書込みトランザクションを開始するとします。トランザクション内で、カーソルをオープンした後カーソルから行をフェッチし、行を更新します。

Oracle Rdb は、ロールバック操作をサポートするために **Recovery-unit ジャーナル・ファイル**（ファイル形式は .ruj）を作成します。このファイルには、トランザクションで変更または削除する前の行（または定義）が書き込まれます。Recovery-unit ジャーナル・ファイルには、トランザクションでどの行（または定義）が追加されたかも記録されます。トランザクションがロールバックされると、Oracle Rdb は、Recovery-unit ジャーナル・ファイルの情報を使用してトランザクションによるデータベース変更を取り消します。

各行を更新すると、Oracle Rdb は、更新前の行を Recovery-unit ジャーナル・ファイルに書き込みます。すべての UPDATE 文を実行すると、EMPLOYEES テーブルには更新済みの行が入り、リカバリ・ユニットジャーナル・ファイルには元の行が入ります。図 16-6 (16-43) に、データベースの更新による影響を示します。

図 16-6 更新トランザクション実行中のトランザクション Recovery-unit ジャーナル (.ruj) ファイル



COMMIT 文を入力して EMPLOYEES テーブルへの変更を恒久的なものにすると、Oracle Rdb は、Recovery-unit ジャーナル・ファイルを空にして、後続のトランザクションに備えます。

ROLLBACK 文を入力してトランザクションで行った EMPLOYEES テーブルへの変更を取り消すと、SQL は、Recovery-unit ジャーナル・ファイルを使用してデータベースをトランザクション開始前の状態に戻します。SQL は、EMPLOYEES テーブル内の変更された行を、Recovery-unit ジャーナル・ファイルに格納された元のバージョンの行で置き換えます。

トランザクションでは SQL 文がユニットにグループ化されるので、たとえば 1 つの文がエラーになった場合に、ユニットを構成するすべての文で行ったデータベース変更を取り消すことができます（ロールバック）。したがって、トランザクションではユニットを構成する

すべての文が正常に実行され、予定の作業が完了するまで、データベース変更の確定（コミット）を待つことができます。

対話型 SQL と OpenVMS で動くプログラムでは、ROLLBACK 文を指定しない場合、SQL は（デフォルトで）プログラムが正常に終了するとデータベース変更をコミットします。SQL は、トランザクション内のすべての更新操作が正常に実行されたかどうかにかかわらず、変更をコミットします。したがって、プログラム内で標準的なエラーを監視し、処理することが特に重要です。

対話型 SQL では、トランザクションがアクティブでない場合、完了ステータスは EXIT 文の直前の文のステータスとなります。この場合、SQL はトランザクションをコミットしようとするため、完了ステータスはその試みられた COMMIT 文のステータスになります。

Compaq Tru64 UNIX

Compaq Tru64 UNIX では、プログラムで SQL を終了すると、デフォルトではアクティブなトランザクションがロールバックされます。アクティブなトランザクションは、その終了ステータスを Oracle Rdb が認識できないため、コミットできません（OpenVMS ではできます）。トランザクションのコミットが必要な場合は、終了する前に必ず実行してください。



ただし、プログラムがリカバリできない異常な状態のトランザクションは SQL がロールバックします。たとえば、次の場合はトランザクションがロールバックされます。

- OpenVMS で、[Ctrl]+[Y] を押してから DCL STOP コマンドを入力し、プログラムの実行を終了した場合
- ハードウェア障害が発生した場合
- プログラム・イメージが障害ステータスで終了した場合

バッチ更新トランザクションは、必ずロールバックではなくコミットされなければなりません。バッチ更新トランザクションによる変更の書込みは、次の場所に限定されます。

- データベース・ストレージ・ファイル（1 つ以上）
- After-image ジャーナル・ファイル（ALTER DATABASE 文で有効化した場合）

バッチ更新トランザクションは、スナップショットまたは Recovery-unit ジャーナル・ファイルに書込みをしないため、ディスク I/O 操作が最小限化されます。しかし、そのためにトランザクションのロールバックに必要な情報が入手できないことになります。バッチ更新トランザクションがエラーになると、データベースは破損します。データベース再生の唯一の方法は、データベース全体を再構築するか、バックアップからリストアすることです。

プログラムにおける複数接続の管理

この章では、SQL 接続を対話的に使用してクエリー、テスト、プログラムのプロトタイプを実行する方法、および SQL 接続をプログラム内で使用する方法を説明します。接続を、基本的に静的および動的な SQL プログラムを開発するためのプログラミング・ツールとして捉えます。**接続**によって、Oracle Rdb データベースの明示的な接続と切断、1 つ以上のデータベースへの同時アクセス、SQL トランザクションの同時処理が可能になります。

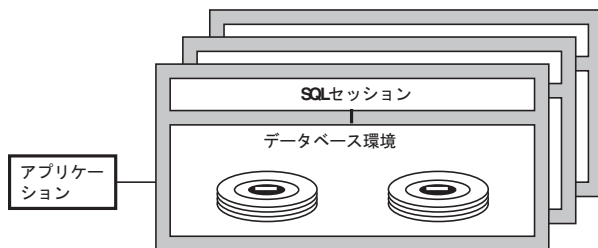
この章では、次の方法について解説します。

- SQL 接続のコンポーネントの定義
- 接続の作成、切替え、終了
- 接続を伴うトランザクションの使用
- プログラム内での接続の有効化と無効化
- アプリケーションでの接続の使用方法

17.1 接続について

SQL 接続は、2つの独立した関連する要素、SQL セッションとデータベース環境を関連付けて構成されます。接続によって、セッションとデータベース環境を関連付け、それらはアプリケーションと関連付けられた SQL と対をなします。アプリケーションは、図 17-1 (17-2) に示すように、1つまたは複数の接続を保持できます。

図 17-1 SQL 接続のコンポーネント



SQL セッションは、一連の SQL 文、およびその実行とリソースのコンテキストを意味しています。

データベース環境は、プログラム（または対話型 SQL）が接続するすべてのデータベースで構成されます。データベース環境は、1つの単位として接続し、切断する Oracle Rdb データベース群を表します。また、接続は、現行の接続において一意のエイリアスを持つすべてのデータベース群を指します。

17.1.1 セッションの定義

接続のコンテキストでは、SQL セッションは、連続する一連の SQL 文の実行、およびこれらの文の任意の時刻における実行状態の両方を表します。一連の SQL 文は、時間の経過に伴って定期的に変化します。次のようなコンテキストで変化が発生します。

- 文、クエリー、カーソル、動的にプリコンパイルされた SQL 文のすべての状態
- 要求ハンドルと動的にプリコンパイルされた SQL 文で使用するリソース

SQL では、プログラムが最初の SQL 文を実行すると暗黙的にデフォルトのセッションが作成されます。デフォルトのセッションとは、デフォルトのデータベース環境を構成するすべてのデータベース接続に関連する文とリソースのコンテキストです (17.1.2 項 (17-3) を参照)。デフォルトのセッションとデフォルトのデータベース環境とは、デフォルトの接続における対応コンポーネントです (17.1.3 項 (17-4) を参照)。

対話型 SQL では、セッションは一連の実行対象 SQL 文で構成されます。接続には、広義では SQL 文のコンテキストが含まれます。このようにセッションの捉え方を拡張することは、接続を考える上で特に重要な意味があります。

1つのアプリケーションで複数の接続を保持し、必要に応じて接続を切り替えることができるので、接続が切り替わる前に、SQL は、現在の接続に関する SQL 文のコンテキストを一時停止する必要があります。SQL 文の実行を一時停止すると、プログラムは別の接続に切り替えてから元の接続に戻り、中断している実行を再開できます。類例で考えるとこの点が明らかになります。

複数の接続を保持するアプリケーションは、ビデオカセット・レコーダのグループにたとえて考えることができます。一時停止したり、再開したりしながら、完了までの様々な段階でビデオ・テープを見ることができます。一時停止すると、もう一度テープを表示するまで処理が中断します。テープを一時停止した後、中断したところから表示を再開できるのとまったく同様に、SQL 文のコンテキストを一時停止すると、プログラム（または対話型 SQL）は接続を切り替えてからセッションに戻り、SQL 文のコンテキストが一時停止したところから処理を継続できます。

SQL セッションは、次のように休止（休止中）とカレント（動作中）の2つの状態があります。

- 休止中の SQL セッションは、現行の SQL 環境以外の SQL 環境に結合された SQL セッションです。
- 現行の SQL セッションは、現行の SQL データベース環境に結合された SQL セッションです。

SQL は、ユーザーの接続を自動的に処理します。ユーザーが認識する必要があるのは、各接続が1つのセッションを保持することと、セッションはカレント（動作中）か休止（休止中）かのいずれかの状態であることのみです。

17.1.2 データベース環境の定義

有効な接続には、セッションに対応するデータベース環境が必要です。セッションは、SQL 文の実行と、接続のリソース・コンテキストを表しますが、**データベース環境**は次の条件を満たす1つ以上のデータベースを表します。

- 特定のセッションに結合されている。
- 1つの単位として接続または切断する。
- 現行の接続で一意のエイリアスが割り当てられる。

埋込みおよびモジュール言語プログラムでは、アプリケーションは少なくとも1つ以上の**デフォルトのデータベース環境**を保持しています。コンパイル時に DECLARE ALIAS 文で宣言したすべてのデータベースは、アプリケーションのデフォルトの環境を構成します。

例 17-1 (17-4) のコードは、埋込み SQL による2つのエイリアス宣言で作成されたデフォルトのデータベース環境を示します。

例 17-1 埋込み SQL によるデータベース宣言で作成されたデフォルトのデータベース環境

```
EXEC SQL DECLARE ALIAS FILENAME personnel;  
EXEC SQL DECLARE ALIAS _CORP ALIAS FILENAME corporate;
```

このプログラムのデフォルトのデータベース環境は、personnel と corporate の 2 つのデータベースで構成されます。SQL では複数のエイリアスを宣言できますが、デフォルトのエイリアスは唯一と見なされます。DECLARE ALIAS 文でエイリアスを省略すると、SQL はデフォルトのエイリアス名 RDB\$DBHANDLE を使用します。DECLARE ALIAS 文が有効なのは、プリコンパイル SQL と SQL モジュール言語だけです。対話型 SQL 内で、または動的に実行可能な SQL 文としては使用できません。

DECLARE ALIAS 文では、モジュールのコンパイルとプリコンパイルで使用するデータベース定義の名前とソースを指定し、アプリケーションのデフォルト・データベース環境の名前付きエイリアス部分を作成します。DECLARE ALIAS 文は、コンパイル時にプログラムにデータベースを宣言する非実行文です。SQL は、プログラムまたは SQL モジュール内で実行可能な最初の SQL 文を実行するときに、初めてデータベースに接続します。

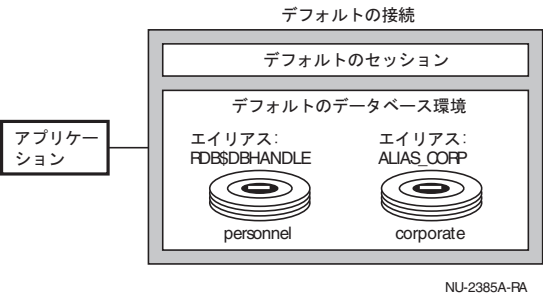
17.1.3 接続の定義

接続によって、セッションとデータベース環境を関連付け、対応するセッションと環境のペアをアプリケーションに関連付けます。アプリケーションは少なくとも 1 つの接続（デフォルトの接続と呼ばれる）を保持しますが、アプリケーションからのニーズに応じて複数の接続が可能です。デフォルトの接続の構成要素は次のとおりです。

- デフォルトのデータベース環境。プログラムで宣言したすべてのデータベースで構成されます。
- デフォルトのセッション。デフォルトのデータベース環境に関連するすべての SQL 文のコンテキストとリソースのコンテキストを表します。

図 17-2 (17-4) に示すアプリケーションのデフォルトの接続は、デフォルトのセッションと宣言済みの 2 つのデータベースによるデフォルトのデータベース環境で構成されます。

図 17-2 デフォルトの接続



personnel データベースではデフォルトの RDB\$DBHANDLE エイリアスを使用し、corporate データベースではユーザー定義のエイリアス ALIAS_CORP を使用します。

プログラムで接続を明示的に指定することなくモジュール内のプロシージャを呼び出すと、SQL はデフォルトの接続を確立し、デフォルトの SQL セッションとデフォルトのデータベース環境に接続します。SQL は、すべてのデータベースを同時に宣言し、プログラムまたは SQL モジュール内で実行可能な最初の SQL 文を実行するときに初めて個々のデータベースに接続します。

17.2 項 (17-5) で説明する CONNECT 文を使用すると、デフォルト以外の明示的な接続を作成できます。

17.2 接続の作成、切替え、終了

SQL インタフェースは、プログラム内で接続を制御する 3 つの文を用意しています。表 17-1 (17-5) では、CONNECT、SET CONNECT、DISCONNECT の各文について説明します。これらの文は、それぞれ接続の作成、切替え、終了を行います。

表 17-1 接続を制御する SQL 文

SQL 文	説明
CONNECT	セッションとデータベース環境のペアを作成し、この接続ペアをアプリケーションに関連付けます。接続に名前を付け、SET CONNECT 文で接続を変更するとき、および DISCONNECT 文で接続を解除するときに、この名前を使用します。
SET CONNECT	有効な接続のいずれかを選択し、接続を切り替えます。後続の SQL 文のデフォルトのデータベース環境を変更します。
DISCONNECT	宣言されたデータベースの接続を解除し、プログラムの宣言で指定したエイリアスを解放します。

接続の作成については、17.2.1 項 (17-5) を参照してください。接続の切替えについては、17.2.5 項 (17-9) を参照してください。接続の終了については、17.2.6 項 (17-10) を参照してください。

17.2.1 接続の作成

プログラムで CONNECT 文を実行すると、SQL は接続を作成し、データベース環境に接続し、CONNECT 文で指定された名前を付けます。また、暗黙的な SET CONNECT 文を実行して接続を確立するので、CONNECT 文の後に明示的な SET CONNECT 文を実行する必要はありません。

17.2.2 項 (17-6)、17.2.3 項 (17-7)、17.2.4 項 (17-8) に、CONNECT 文を使用する方法の例を示します。17.2.5 項 (17-9) では、明示的な接続を作成する方法とタイミングを示します。

17.2.2 デフォルトのデータベース環境の複製

CONNECT 文を使用すると、デフォルト接続内のすべてのエイリアスを別の接続内に複製できます。これを明示的に行うには、DEFAULT 引数と AS 句、または AS 句のみを使用します。3 つの CONNECT 文のいずれかを使用してデフォルトのデータベース環境を複製できます。

- デフォルト接続内のすべてのエイリアスを明示的に指定します。
このように CONNECT 文を使用してデフォルトのデータベース環境を複製する作業は、3 つの方法の中では最も大きな労力を必要とします。他の 2 つの方法のいずれかを使用すれば、入力に要する時間を節約できます。しかし、CONNECT 文で明示的にエイリアスを指定する方法では、新しい接続で使用するエイリアスだけを指定することで、デフォルトの接続で指定したエイリアスの一部を複製することができます。
明示的にすべてのエイリアスを指定する文は、次のようになります。

```
CONNECT TO 'ALIAS RDB$DBHANDLE, ALIAS ALIAS_CORP' AS 'OTHER';
```

- DEFAULT 引数と AS 句を使用します。
次のように、(DEFAULT 引数を使用することで) デフォルト接続内のエイリアスを複製することを明確に示すことができます。

```
CONNECT TO 'DEFAULT' AS 'OTHER';
```

- AS 句のみを使用します。
CONNECT 文に AS 句と名前のみを指定します。次に例を示します。

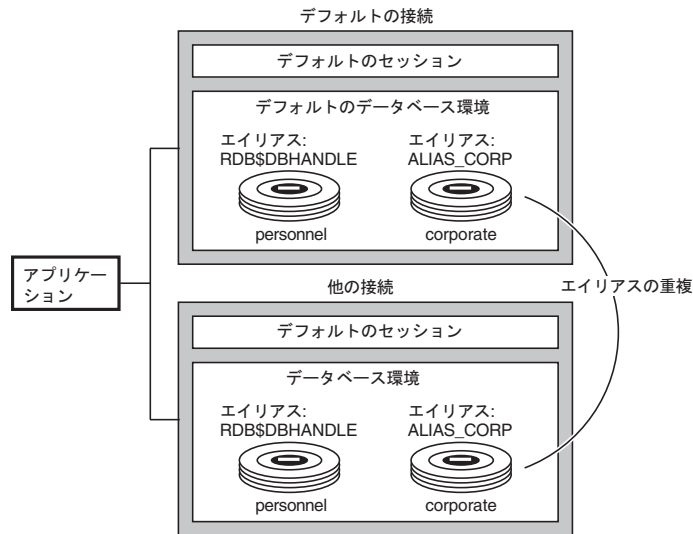
```
CONNECT AS 'OTHER';
```

SQL は、デフォルトの接続内のエイリアスを、指定した名前で複製します。これは、デフォルトのデータベース環境を複製するための最も容易な方法です。

上記の CONNECT 文のいずれかで作成されたデータベース環境を、図 17-3 (17-7) に示します。

新しい接続（上の例で作成した OTHER など）で複製できるのは、デフォルトの接続のエイリアスに限ります。それ以前に作成した接続からは複製できません。

図 17-3 デフォルトの接続の複製



NU-2386A-RA

17.2.3 異なるデータベースに対する同じエイリアスの指定

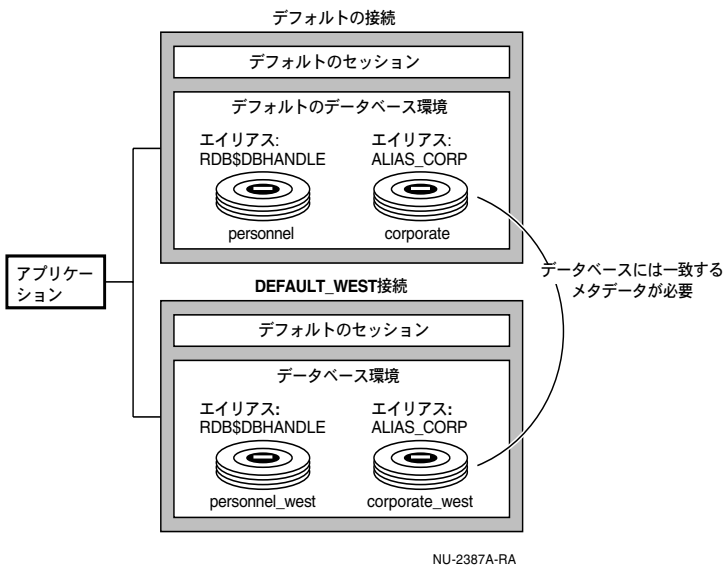
データベースは異なるものの同じエイリアスを持つ2つの接続を保持しなければならない場合があります。次の例に、この方法を示します。

```
CONNECT TO 'FILENAME personnel_west,
           ALIAS ALIAS_CORP FILENAME corporate_west' AS 'DEFAULT_WEST';
```

異なる接続でエイリアスを共有するデータベースは、同じメタデータを保持していなければなりません。したがって、personnel データベースと personnel_west データベース、および corporate データベースと corporate_west データベースはメタデータが一致する必要があります。

上記の CONNECT 文で作成されたデータベースを図 17-4 (17-8) に示します。

図 17-4 異なるデータベースに対する同じエイリアスの指定

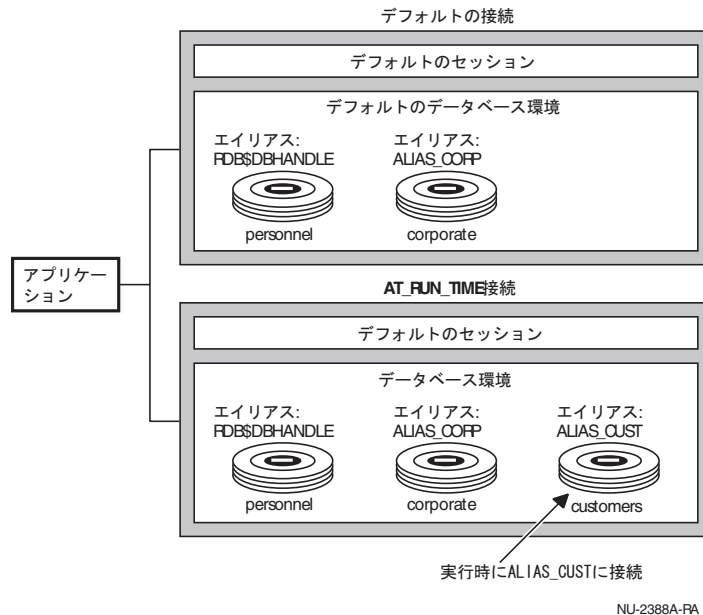


17.2.4 実行時の接続の指定

CONNECT 文で作成し、実行時に ATTACH 文で接続するデータベース環境を図 17-5 (17-9) に示します。この場合は、次の例に示すように CONNECT 文の中に ATTACH 文を指定します。

```
CONNECT TO 'ALIAS RDB$DBHANDLE, ALIAS ALIAS_CORP,
          ATTACH ALIAS ALIAS_CUST FILENAME customers' AS 'AT_RUN_TIME';
```

図 17-5 実行時の追加接続の指定



17.2.5 接続の切替え

CONNECT 文で接続を作成すると、プログラムは、それに接続する方法を持っていないかもしれません。SQL の埋込みおよびモジュール言語プログラムでは、次の 2 種類の SET CONNECT 文を使用してこれを実現します。

- SET CONNECT connection-name
プログラムで使用可能な接続名から、指定された接続にプログラムを接続します。
- SET CONNECT DEFAULT
デフォルトの接続に接続します。デフォルトの接続は、デフォルトのセッションとデフォルトのデータベース環境で構成されます。デフォルトのデータベース環境は、コンパイル時にプログラムで宣言したすべてのデータベースで構成されます。

SET CONNECT 文によって、現行の接続が一時停止し、この文で指定された接続に切り替わります。接続を切り替えると、後続の SQL 文はその接続のコンテキストの中で実行されます。

もう一度 SET CONNECT 文を発行し、元の接続に戻って処理を再開することもできます。

17.2.6 接続の終了

プログラムで接続が不要になった場合は、DISCONNECT 文を発行します。DISCONNECT 文は次の処理を行います。

- 1 つ以上の接続を削除します。
- 各接続に伴うセッションを閉じます。
- 各接続に伴うリソースをすべて解放します。
- コンパイルされ用意された SQL 文をすべて解放します。
- すべてのデータベースを切断します。
- セッション・コンテキストのメモリーを解放します。

DISCONNECT 文は、一部またはすべての接続の切断に柔軟に対応します。DISCONNECT 文には以下の文があります。

- DISCONNECT ALL は有効な接続をすべて終了します。
- DISCONNECT DEFAULT は、デフォルトの接続のみを終了します。
- DISCONNECT CURRENT は、プログラムの現行の接続を終了します。
- DISCONNECT connection-name は、名前を指定した接続を終了します。

接続を明示的に切断することなくプログラムを終了すると、SQL は次の処理を行います。

- デフォルトの接続に関するトランザクションをすべてコミットし、暗黙的な DISCONNECT ALL 文を実行します。
- 同じプログラム内で有効な他のすべてトランザクションをロールバックし、暗黙的な DISCONNECT ALL 文を実行します。

ANSI/ISO SQL 標準に準拠するには、明示的な DISCONNECT ALL 文を使用してください。これにより、すべてのトランザクションがロールバックされます。現行の接続を切断するには、COMMIT 文と DISCONNECT CURRENT 文を使用してください。

17.3 接続を伴うトランザクションの使用法

各セッションには、SQL が最初に接続に入るときに起動する独自のトランザクションがあります。例 17-2 (17-11) の SQL モジュールに示すように、接続におけるトランザクションの特性は SET TRANSACTION 文で変更できます。あるトランザクションがアクティブな間はデータベースに接続できないので、トランザクションをコミットまたはロールバックしなくても、複数の接続によって複数のトランザクションを同時にアクティブにすることができます。

17.4 プログラムにおける接続の有効化と無効化

SQL モジュールまたはプリコンパイル・プログラムで SQL 接続を使用する前に、SQL 接続を有効化する必要があります。SQL では、接続がデフォルトで無効になっています。この項では、SQL モジュール・プロセッサまたは SQL プリコンパイラを使用する場合に接続を有効化し、無効化するプロセスについて説明します。

17.4.1 モジュール・プログラミングにおける接続の有効化と無効化

SQL モジュール・プロセッサで接続を有効化するには、プログラムをコンパイルし、CONNECT または `-conn` 修飾子でモジュールを処理し、プログラムと処理済みのモジュールをリンクしてから実行します。

接続はデフォルトでは無効化されています。ただし、NOCONNECT または `-noconn` 修飾子を使用すると明示的に接続を無効化できます。

SQL モジュール・プロセッサの接続を有効化する場合は、接続を有効化してコンパイルしたモジュールと接続を無効化してコンパイルしたモジュールを 1 つのプログラムの中で組み合わせることはできません。接続を有効化してすべてのモジュールをコンパイルするか、それ以外の場合はすべてのモジュールを接続なしでコンパイルしてください。両方の併用はできません。

17.4.2 プリコンパイル・プログラムにおける接続の有効化と無効化

SQL プリコンパイラで接続を有効化するには、SQLOPTIONS=CONNECT または `-s' -conn` 修飾子を使用し、アプリケーションをリンクしてから実行します。

接続はデフォルトでは無効化されています。ただし、/SQLOPTIONS=NOCONNECT または `-s' -noconn` 修飾子を使用すると明示的に無効化できます。

17.5 アプリケーションでの接続の使用法

この項では、サンプル SQL モジュール（例 17-2 (17-11)）と対応する C プログラム（例 17-3 (17-13)）を示します。これは接続の使用例の 1 つです。このプログラムでは、同じデータベースへの 2 つの異なる接続を作成します。最初の接続では、読取り専用トランザクションによってカーソルをオープンし、カーソルを進めながらフェッチを実行します。2 番目の接続では、読取り / 書き込みトランザクションによってデータベースを更新します。

例 17-2 接続を使用する SQL モジュール

```
-- SQL module that uses connections for two concurrent read/write
-- transactions.
MODULE CURSOR_LOOP
LANGUAGE C
```

```
AUTHORIZATION JONES
PARAMETER COLONS
DECLARE ALIAS FILENAME test
DECLARE A CURSOR FOR SELECT CITY FROM S
        WHERE CITY IS NOT NULL
-- Connection management routines.
-- Create an explicit connection that duplicates the default database
-- environment.
PROCEDURE CONNECT
    (SQLCODE,
     :CONNECT_2 CHAR(31));
    CONNECT AS :CONNECT_2;
-- Switch to the nondefault connection called CONNECT_2.
PROCEDURE SET_CONNECT
    (SQLCODE,
     :CONNECT_2 CHAR(31));
    SET CONNECT :CONNECT_2;
-- Switch to the default connection.
PROCEDURE SET_CONNECT_DEFAULT
    (SQLCODE);
    SET CONNECT DEFAULT;
-- Disconnect all available connections (CONNECT_2 and DEFAULT).
PROCEDURE DISCONNECT
    (SQLCODE);
    DISCONNECT ALL;
-- Open and fetch row information from cursor A in a read-only transaction.
PROCEDURE OPEN_A
    (SQLCODE);
    OPEN A;
PROCEDURE FETCH_A
    (SQLCODE,
     :RET_CITY CHAR(15));
    FETCH A INTO :RET_CITY;
PROCEDURE CLOSE_A
    (SQLCODE);
    CLOSE A;
-- Update rows that meet a certain program criteria.
PROCEDURE UPDATE_P
    (SQLCODE,
     :PCITY CHAR(15));
    UPDATE MYP SET CITY_CODE = 'XXXXX' WHERE CITY = :PCITY;
-- Start a read-only transaction for fetching rows.
PROCEDURE RO_TXN
    (SQLCODE);
    SET TRANSACTION READ ONLY NOWAIT;
-- Start a read/write transaction for updating rows.
PROCEDURE RW_TXN
```

```

        (SQLCODE);
        SET TRANSACTION READ WRITE NOWAIT;
-- Commit the update transaction.
PROCEDURE COMMIT
        (SQLCODE);
        COMMIT;

```

例 17-3 (17-13) に、例 17-2 (17-11) のモジュールを使用するプログラムを示します。

例 17-3 接続を使用する C プログラム

```

.
.
.
/* Create the read-only transaction in the default connection. SQL
implicitly creates a default connection when a program executes its
first SQL statement.*/
RO_TXN( &sqlcode );
if (sqlcode != SQLCODE_SUCCESS)
    goto err;
/* Create a second connection to handle update operations. */
CONNECT( &sqlcode, updater );
if (sqlcode != SQLCODE_SUCCESS)
    goto err;
/* Return to the default connection to read rows. */
SET_CONNECT_DEFAULT( &sqlcode );
if (sqlcode != SQLCODE_SUCCESS)
    goto err;
/* Open a cursor and read the first row. */
OPEN_A( &sqlcode );
if (sqlcode != SQLCODE_SUCCESS)
    goto err;
FETCH_A( &sqlcode, city );
while (sqlcode == SQLCODE_SUCCESS) {
/* Test whether or not values in the row meet program criteria. If criteria
is not met, the program loops back to read another row; otherwise it
continues. */
    if (strcmp( city, "London          ") == 0) {
        /* Switch to the nondefault connection. */
        SET_CONNECT( &sqlcode, updater );
        if (sqlcode != SQLCODE_SUCCESS)
            goto err;
        /* Create the read/write transaction for an update operation. */
        RW_TXN( &sqlcode );
        if (sqlcode != SQLCODE_SUCCESS)
            goto err;
        /* Update the rows that meet the criteria. */
        UPDATE_P( &sqlcode, city );
    }
}

```

```
        if (sqlcode != SQLCODE_SUCCESS)
            goto err;
        /* Commit the change. */
        COMMIT( &sqlcode );
        if (sqlcode != SQLCODE_SUCCESS)
            goto err;
        /* Return to the default connection to read rows. */
        SET_CONNECT_DEFAULT( &sqlcode );
        if (sqlcode != SQLCODE_SUCCESS)
            goto err;
    }
    /* Read another row. */
    FETCH_A( &sqlcode, city );
}
/* Close cursor A. */
CLOSE_A( &sqlcode );
if (sqlcode != SQLCODE_SUCCESS)
    goto err;
return;
/* Error handling. */
err:
    sql_signal();
    /* Disconnect all active connections. */
    DISCONNECT(&sqlcode);
    EXIT(0);
}
```

このプログラムでは、2つのアクティブなデータベース接続、2つの並行トランザクション、両方の接続への同時アクセスを行います。

第Ⅵ部

プログラムにおけるデータ操作

第6部では、次の項目について説明します。

- カーソルの宣言方法と使用方法
- SQL プログラムからデータをロードして更新する方法
- マルチスキーマ・データベース内のデータを操作する方法

カーソルの使用方法

この章では、SQL カーソルを定義し、SQL カーソルのクラスとタイプについて説明します。
この章の各項では、次の内容について解説します。

- カーソルについて
- カーソルの分類
- カーソルのオープンとクローズの制御方法
- テーブル・カーソルの使用方法
- 保持可能なテーブル・カーソルの使用方法
- リスト・カーソルの使用方法
- スクロール可能なリスト・カーソルの使用方法
- 動的カーソルの使用方法
- 拡張動的カーソルの使用方法

参考: 11 章 (11-1) では、動的カーソルと拡張動的カーソルを使用して実行時に SELECT 文を処理する方法について説明します。『Oracle Rdb7 SQL Reference Manual』に、カーソルの機能に関する詳しい説明があります。

18.1 カーソルについて

多くの場合、SQL を使用してデータを検索すると、選択式の結果として複数の行が取得されます。しかし、アプリケーション・プログラムは複数の行を同時には処理できません。したがって、一度に 1 行ずつ処理する必要があります。よって、SQL で 1 行しか取得しない場合（1 行形式の選択）を除いて、プログラム内でカーソルを使用してデータベース内のデータを順に読取り、挿入、更新する必要があります。

カーソルを使用すると、クエリーを実行してから複数の行を一度に 1 行ずつ処理できます。カーソルを使用した場合に作成されるものは次のとおりです。

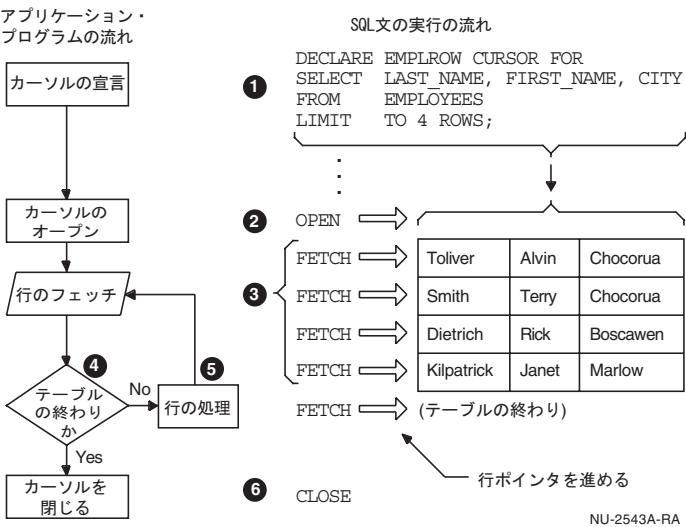
- 順序付きの結果テーブル。1 つ以上のテーブルまたはビューの行と列を一時的に収集したものです。
- 行ポインタ。次にフェッチされる行または結果テーブルの終了を決定します。

カーソルは、データ・ファイルのレコードを処理する一般的な方法に基づいてモデル化されています。複数の行を処理するには、プログラムで一連の基本的なカーソル操作（カーソルの宣言、カーソルのオープン、行のフェッチ、カーソルのクローズ）を行う必要があります。

18.1.1 カーソルの機能

基本的なカーソル操作には、カーソルの宣言、カーソルのオープン、行のフェッチ、カーソルのクローズによる結果テーブルの削除があります。図 18-1 (18-2) に、この基本的なアルゴリズムとテーブル・カーソルを使用して複数行の取得を行う手順を示します。

図 18-1 カーソルと関連する文の処理



カーソルを使用するには、次の手順を実行します。手順は図 18-1 (18-2) に対応します。

1. カーソルを宣言します。
 DECLARE CURSOR 文で結果テーブルの名前を指定し、テーブルのソース、行、列を指定します。カーソルの宣言は次の要素で構成されます。
 - カーソルの名前。この場合は EMPLROW です。
 - 結果テーブルを生成する選択式。この場合は、3 列 4 行の結果テーブルを生成するクエリーです。

次の例に、カーソル EMPLROW を宣言する方法を示します。

```
DECLARE EMPLROW CURSOR FOR
    SELECT LAST_NAME, FIRST_NAME, CITY FROM EMPLOYEES
```

2. カーソルをオープンします。
 カーソルをオープンすると、クエリーが実行され、カーソルの宣言で指定した結果テーブルが作成され、結果テーブルの最初の行の前に行ポインタが設定されます。
 OPEN 文を実行すると（カーソルを宣言するときでなく）、結果テーブルが後続の処理で使用可能になります。次の例に、カーソルをオープンする方法を示します。

```
OPEN EMPLROW;
```

結果テーブルは、カーソルをクローズするまで存在します。
 カーソルをクローズしてからオープンすると、次のフェッチ・データ取得操作は、結果テーブルの最初の行に戻って位置付けられます。カーソルがすでにオープンしている場合、SQL はエラーを返し、次の FETCH 文で次の行を取得します。

3. 行をフェッチします。
 FETCH 文は、行ポインタが指す行（**現在行**）を取得し、結果テーブルの次の行に行ポインタを進めます。行ポインタが結果テーブルの最初の行の前であれば、SQL は最初の行を取得します。行ポインタが結果テーブルの最終行の後にあれば、SQL はすべての行がフェッチされたことを通知します。
 したがって、行ポインタは最初の行の前、特定の行、最終行の後のいずれかの位置にあります。結果テーブルが空の場合は、行ポインタは最初の行の前か最終行の後のいずれかの位置にあります。
 次の例に、対話型 SQL を使用して EMPLROW カーソルの行をフェッチする方法を示します。

```
FETCH EMPLROW;
```

SQL モジュール言語またはプリコンパイル SQL では、INTO 句を使用してデータの値をプログラム・パラメータに格納します。次の例では、FETCH 文を使用して EMPLROW カーソルの値を SQL モジュール言語プロシージャのパラメータに格納する方法を示します。

```
FETCH EMPLROW INTO :L_NAME_P, :F_NAME_P, CITY_P;
```

FETCH 文を実行するたびに、行ポインタは結果テーブル内の新しい行に移動します。結果テーブルが 5 行の場合は、すべての行を取得するのに FETCH 文を 5 回実行する必要があります。SQL プログラムでは、通常、FETCH 文を含むループを実行してすべての行を処理します。

SQL では、行のフェッチに関してかなり柔軟な操作が可能です。行は、結果テーブル内で発生する順序で取得できます。リスト・カーソルを使用する場合は、スクロール可能なリスト・カーソルを使用して行をランダムに取得することもできます。スクロール可能なリスト・カーソルの詳細は、18.7 項 (18-15) を参照してください。

スクロール可能なリスト・カーソルを宣言しないと、FETCH 文によって行は必ず発生する順序で取得されます。行をスキップしたり戻したりはできません。前にフェッチした行を取得するには、カーソルをクローズしてから再オープンし、新しい結果テーブルを作成してから目的の行が得られるまで行をフェッチする必要があります。カーソルをクローズしてから行をフェッチしようとする、カーソルの宣言で指定した結果テーブルは存在しないのでフェッチ操作はエラーになります。

4. "No data" の状態かどうかをテストします。
データ・ファイルの処理と同様に、結果テーブルがいつ終了するかを認識し、正しく処理を終了する必要があります。ステータス・パラメータ `SQLCODE` か `SQLSTATE` をチェックして、データがない状態を検知します。
5. 行を処理します。
データがある場合は、現在フェッチされた行はプログラムで使用できます。取得したデータは、プログラム内での使用、位置を指定した UPDATE 文による行の変更、位置を指定した DELETE 文による行の削除が可能です。
位置を指定した DELETE 文がカーソル操作で正常に実行されると、行ポインタは次の行に移動します。検出された行が結果テーブルの最後の行の場合、行ポインタは最終行の後に移動します。
6. カーソルをクローズします。
カーソルをクローズすると、後続の処理で結果テーブルを使用できなくなり、カーソルの宣言で指定した結果テーブルは削除されます。カーソルは、何度もオープンしたりクローズしたりできます。カーソルをクローズしてから再オープンすると、クエリーを再実行し、行ポインタは新しい結果テーブルの最初の行の前に戻ります。次の例に、カーソルをクローズする方法を示します。

```
CLOSE EMPLOYROW;
```

オープンしているカーソルをすべてクローズするには、`sql_close_cursors()` ルーチンを使用します。

カーソルのオープンとクローズを制御する方法の詳細は、18.3 項 (18-8) を参照してください。

カーソルをクローズした後でも、カーソルの宣言はプログラム実行サイクル内の接続全体を通じて有効です。接続を切断 (DISCONNECT 文を使用または同じエイリアスで別のデータベースに接続) した後は、カーソルの宣言は無効になります。

カーソルをオープンすると新しい結果テーブルが作成されるので、同じカーソルを別の

トランザクションでオープンすると、結果テーブルは他のプロシージャで実行したデータベースの更新を反映する場合があります。トランザクションにおけるデータの一貫した状態の維持については、16 章 (16-1) を参照してください。

アプリケーション・プログラムでカーソルをオープンすると、Oracle Rdb はユーザーが特定のタイプのカーソル（更新カーソルなど）をオープンするために必要な権限を持っているかどうか、要求を実行するのに必要な権限があるかどうかを確認します。すなわち、アプリケーションでカーソルを使用して OPEN 文と同じモジュール内で行を挿入、更新、削除する場合、アプリケーションが実行時にこの処理を実際には実行しない場合でも、ユーザーはこの処理を実行するのに十分な権限を持っている必要があります。

参考: FETCH 文は SQL 通信領域 (SQLCA) 内の行カウント情報を更新します。FETCH 文が正常に実行されると、SQLCA 内の SQLERRD 配列の 3 番目の要素はこの文で取得した行のカーソルの位置を示しています。詳細は、『Oracle Rdb7 SQL Reference Manual』の SHOW SQLCA 文に関する項と、『Oracle Rdb7 SQL Reference Manual』の SQLCA に関する付録を参照してください。

18.1.2 カーソルとビューの比較

カーソルはビューと同様の論理構造です。ビューと同様に、カーソルは 1 つ以上のテーブルまたはビューの行と列を一時的に収集します。この行と列の一時的なコレクションを、結果テーブルと呼びます。

ビューの結果テーブルは、文がそのビューにアクセスするときに初めて作成されます。カーソルの結果テーブルは、そのカーソルをオープンするときに初めて作成されます。

SELECT、INSERT、UPDATE 文でビューを参照する場合とは異なり、カーソルの結果テーブルが存在する時間は変えることが可能です。ビューに関連して作成されたテーブルは、ビューを参照する文を実行する間だけ存在しますが、カーソルに関連して作成されたテーブルは、明示的または暗黙的にカーソルをクローズするまで存在します。

カーソルは論理エンティティであり、特定の結果テーブルではありません。ただし、説明を簡単にするために、"カーソル" という用語は、しばしば OPEN 文で作成された結果テーブルを表すために使用されます。他の SQL のマニュアルやこのマニュアルの他の章を読む場合は、このことに留意してください。

18.1.3 カーソルが必要な場合

プログラムで結果テーブルに複数の行が存在する場合は、カーソルを使用する必要があります。ただし、カーソルを必要としないケースもあります。たとえば、次のような場合です。

- プログラム変数の値を使用して複数の行を削除する条件を決定し、該当するすべての文を削除する DELETE 文を実行します。

- プログラム変数の値を使用して更新する行を選択する条件を決定し、該当するすべての行の特定の列を同じ値に変更します。

唯一の行に対して複数の操作を実行する場合は、カーソルを使用する必要はありません。1 行の処理では、**1 行形式の選択**を使用した方が簡単です。その場合は、SELECT 文に INTO 句を指定してデータベースの行を取得し、その値をプログラム変数に格納します。実際に、SELECT 文の INTO 句はプログラムで唯一の行を取得する場合にのみ使用されます。次に、SQL モジュールにおける 1 行形式の選択の文を示します。

```
SELECT EMPLOYEE_ID INTO :ID_NUMBER:ID_NUMBER_IND
      FROM DBH1.EMPLOYEES
      WHERE EMPLOYEE_ID = :INPUT_ID
SELECT LAST_NAME, FIRST_NAME INTO :FULL_NAME
      FROM DBH1.EMPLOYEES
      WHERE EMPLOYEE_ID = :INPUT_ID
```

プログラムで NULL 値をとり得る列のデータを取得または格納する場合は、前の 2 つの例の先頭に示すインジケータ・パラメータを使用する必要があります。インジケータ・パラメータをどんな場合にどのように使用するかについては、8 章 (8-1) を参照してください。

しかし、多くの場合、各行の値をプログラム変数で取得して、複数の行を一度に 1 つずつ処理する必要があります。このためには、カーソルの宣言とそのカーソルに伴う結果テーブルを作成する文が必要です。

プログラム全体の中でカーソルがどう機能するかについては、サンプル・ディレクトリに格納されたプリコンパイル・プログラム sql_report を参照してください。

18.2 カーソルの分類

SQL のカーソルは、タイプ、クラス、モードによって分類されます。

SQL では、次に示す 2 つのタイプのカーソルを作成できます。

- テーブル・カーソル
テーブル・カーソルを使用すると、結果テーブルの各行にアクセスできます。図 18-1 (18-2) など、この章では、多くの例でテーブル・カーソルの使用方法を説明します。18.4 項 (18-9) では、テーブル・カーソルを使用した作業方法を説明します。
- リスト・カーソル
リスト・カーソルを使用すると、リスト内の個々の要素にアクセスできます。リストは、セグメントとも呼ばれる要素の順序付きの集合です。リストは LIST OF BYTE VARYING データ型、リストを構成する個々のセグメントは BYTE VARYING データ型です。リスト・カーソルによって、大きすぎて CHAR または VARCHAR フィールドには格納できないオブジェクトを操作できるようになります（これらのフィールドには、多くの場合、構造化バイナリ・データが入ります）。18.6 項 (18-13) では、リスト・カーソルの使用方法について説明します。

カーソルのクラスは、コンパイル時に指定する情報によって区別されます。カーソルには、次の3つのクラスがあります。

- 静的カーソル。このマニュアルで使用する "カーソル" という用語は、一般に静的カーソルを表します。
 DECLARE CURSOR 文でカーソル名と SELECT 文の両方を明示的に指定した場合のカーソルは静的カーソルです。つまり、カーソル名と SELECT 文はコンパイル時に認識されます。
 静的カーソルは、対話型 SQL とプログラムで使⽤します。この章に示す多くの例では、静的カーソルの使用⽅法を説明します。
- 動的カーソル
 DECLARE CURSOR 文で SELECT 文のかわりにプリコンパイルされた SQL 文の名前を指定した場合は動的カーソルになります。
 動的カーソルは、動的 SQL プログラムで使⽤します。コンパイル時にカーソル名を明示的に指定しますが、SELECT 文は明示的に指定しません。つまり、カーソル名はコンパイル時に認識されますが、SELECT 文は実行時に初めて認識されます。18.8 項 (18-17) では、動的カーソルの使用⽅法について説明します。
- 拡張動的カーソル
 DECLARE CURSOR 文でカーソル名と SELECT 文の両方にパラメータを指定した場合は拡張動的カーソルになります。カーソル名も SELECT 文も、実行時に初めて認識されます。拡張動的カーソルを使用すると、一連のカーソル関連の文で動的に生成された任意の数の文を処理できます。18.9 項 (18-18) では、拡張動的カーソルの使用⽅法について説明します。

SQL では、カーソルのタイプごとにカーソルを次の操作モードに分類します。

- テーブル・カーソルには、次の操作モードがあります。
 - 更新
 結果テーブルの行を変更する場合は、更新カーソルを使用します。更新カーソルは、デフォルトのテーブル・カーソルです。更新カーソルを使用すると、Oracle Rdb はまず行を読み取り、SHARED (または PROTECTED) READ アクセスを使用します。行を更新する場合、Oracle Rdb は EXCLUSIVE アクセスを使用します。
 - 更新専用
 更新専用カーソルを使用すると、Oracle Rdb はより強力なロック・モデルを使用して行をロックし、EXCLUSIVE アクセスを使用して行を読み取ります。この結果、更新専用カーソルを使用するとロックの昇格時に通常発生するデッドロックを回避できます。フェッチしたすべて (またはほとんど) の行を更新する場合は、更新専用カーソルを使用してください。
 - 読取り専用
 結果テーブル内の情報を変更したくない場合は、読取り専用カーソルを使用します。

- 挿入専用
カーソルの位置に行を挿入し、その行にリストをロードする場合は、挿入専用カーソルを使用します。
 - リスト・カーソルには、次の操作モードがあります。
 - 読取り専用
読取り専用カーソルは、既存のリストを読み取る場合に使用します。
 - 挿入専用
挿入専用カーソルは、データをリストに挿入する場合に使用します。
- リスト内のデータは更新できません。

18.3 カーソルのオープンとクローズの制御

18.1.1 項 (18-2) では、OPEN 文を使用してカーソルをオープンする方法と、CLOSE 文を使用してカーソルをクローズする方法を説明しています。この項では、カーソルのオープンとクローズを制御する方法を詳しく説明します。

デフォルトでは、テーブル・カーソルまたはリスト・カーソルの結果テーブルは、OPEN 文が実行された時点からカーソルをクローズするまで存在します。カーソルは次の方法でクローズできます。

- CLOSE 文を使用（明示的）
- COMMIT 文または ROLLBACK 文を使用（暗黙的）
- プログラムの実行を停止（暗黙的）
- 対話型 SQL を終了（暗黙的）

カーソルをクローズすると、SQL は結果テーブルを削除します。カーソルを再オープンすると、REPEATABLE READ 分離レベルを使用しない限り、新しいデータや変更済みのデータを含む新しい結果テーブルが読み込まれます。

データベースを切断すると、SQL はカーソルをクローズするだけでなく、カーソルの宣言を認識しなくなります。

テーブル・カーソルでは、DECLARE CURSOR 文の WITH HOLD 句を使用すると、全トランザクションを通じてカーソルをオープンした状態に保持できます。このようなカーソルは**保持可能カーソル**と呼ばれ、トランザクションが終了してもオープンな状態で残り、新しいトランザクションが開始されたときに行ポインタの位置を保持しています。

保持可能カーソルの説明は、18.5 項 (18-11) を参照してください。

18.4 テーブル・カーソルの使用方法

テーブル・カーソルを使用すると、結果テーブルの各行にアクセスし、1度に1行ずつ行を検索、更新、削除できます。

行を処理するために実行する文は、通常、SQL文とプログラミング言語の文が組み合わされています。プログラムが結果テーブル内の複数の文を処理する場合、これらの文はループになります。初回のループの実行では、FETCH文は結果テーブルの最初の行を指し、プログラムで指定した1つ以上のパラメータにその値を格納します。後続の文は、たとえば次の目的でパラメータ値を操作します。

- プログラムで指定した値とあわせて計算を実行
- レポートに記載
- 端末画面に表示

2回目以降のループの実行では、FETCH文によって次の行の値をプログラム・パラメータに格納します。新しい値のセットを使用して、プログラムの操作を繰り返します。

FETCH文で"no data"の完了条件が確認されると、プログラムは行処理のループを終了します。"no data"の完了条件は、RDB\$_STREAM_EOFとしてデータベース・システムから返されます。対話型SQLでは、このエラーは"%RDB-E-STREAM_EOF, attempt to fetch past end of record stream."と表示されます。プログラムでは、このエラーはSQLCODE ステータス・パラメータの値が100、SQLSTATE ステータス・パラメータの値が02000で返されます。10章(10-1)では、完了例外条件とプログラム内でエラーを処理する方法について説明します。

カーソルがオープンでその結果テーブルに行が存在しない場合は、最初に行を実行するFETCH文(ループの初回)によって"no data"の完了条件が発生します。"no data"の完了条件が発生すると、カーソルに対応する最新の行が存在しないので、列からFETCH文のINTO句で指定したプログラム変数に値が転送されません。

例 18-1 (18-9) に、部門コード MBMF に関する最新の職務経歴データを保存したテーブル・カーソルを宣言する方法と、カーソルをオープンし、行をフェッチし、カーソルをクローズする方法を示します。

例 18-1 テーブル・カーソルの使用方法

```
SQL> DECLARE CURRENT_MBMF TABLE CURSOR FOR
1>     SELECT JOB_CODE, EMPLOYEE_ID, JOB_START FROM JOB_HISTORY
2>     WHERE (DEPARTMENT_CODE = 'MBMF') AND
3>           (JOB_END IS NULL)
4>     ORDER BY JOB_CODE, JOB_START;
SQL> --
SQL> -- The OPEN statement creates the result table.
SQL> --
SQL> OPEN CURRENT_MBMF;
```

```
SQL> --
SQL> -- The first FETCH statement retrieves the first row.
SQL> --
SQL> FETCH CURRENT_MBMF;
  JOB_CODE  EMPLOYEE_ID  JOB_START
  AFGM      00174        22-Sep-1981
SQL> --
SQL> -- Each subsequent FETCH statement retrieves the
SQL> -- row positioned next in the result table.
SQL> --
SQL> FETCH CURRENT_MBMF;
  JOB_CODE  EMPLOYEE_ID  JOB_START
  ASCK      00165        8-Mar-1981
SQL> FETCH CURRENT_MBMF;
  JOB_CODE  EMPLOYEE_ID  JOB_START
  DMGR      00227        25-Nov-1981
SQL> --
SQL> -- The preceding row was the last row in the result table.
SQL> -- Another FETCH statement reveals this fact.
SQL> --
SQL> FETCH CURRENT_MBMF;
%RDB-E-STREAM_EOF, attempt to fetch past end of record stream
SQL> --
SQL> -- The CLOSE statement deletes the result table.
SQL> --
SQL> CLOSE CURRENT_MBMF;
SQL> --
SQL> -- However, the cursor declaration remains in effect during
SQL> -- an entire interactive session or program execution cycle
SQL> -- unless you disconnect from the session using the DISCONNECT
SQL> -- statement or attach to another database using the same alias.
SQL> --
SQL> -- Another OPEN statement creates the result table again.
SQL> --
SQL> OPEN CURRENT_MBMF;
SQL> --
SQL> -- After the cursor is opened again, the FETCH statement starts
SQL> -- data retrieval at the first row in the result table.
SQL> --
SQL> FETCH CURRENT_MBMF;
  JOB_CODE  EMPLOYEE_ID  JOB_START
  AFGM      00174        22-Sep-1981
SQL> --
SQL> -- If a cursor is already open, an OPEN statement generates an
SQL> -- error message.
SQL> --
SQL> OPEN CURRENT_MBMF;
```



```
%SQL-F-CURALROPE, Cursor CURRENT_MBMF was already open
SQL> --
SQL> -- The next FETCH statement retrieves the next row.
SQL> FETCH CURRENT_MBMF;
      JOB_CODE      EMPLOYEE_ID      JOB_START
      ASCK          00165            8-Mar-1981
SQL> --
SQL> -- A COMMIT or ROLLBACK statement implicitly executes a CLOSE
SQL> -- statement, which deletes the result table. An attempt to
SQL> -- fetch a row after ending the transaction fails because no
SQL> -- result table exists for the cursor declaration.
SQL> --
SQL> COMMIT;
SQL> FETCH CURRENT_MBMF;
%SQL-F-CURNOTOPE, Cursor CURRENT_MBMF is not opened
SQL>
```

カーソルの宣言を上書きすることはできません。カーソルをオープンしてから、非常に多くの行が選択されていることがわかった場合は（変更する行を取得するには多くの FETCH 文の入力が必要）、次の 2 通りの方法を選択できます。

- カーソルをクローズしてから、行の選択専用の別のカーソルをオープンします（別の名前を使用）。
読取り / 書込みトランザクションを開始し、アクセスしている行に他のユーザーが更新を要求した場合は、トランザクションを終了し、使用する予定のない行のロックを解放します（カーソルをクローズしても行のロックは解放されません。COMMIT 文か ROLLBACK 文を指定するとロックが解放されます）。マルチユーザー・データベース環境で読取り / 書込みトランザクションを開始した場合は、必要以上に多くの行を含むカーソルの作成は避け、トランザクションをできるだけ短くしてください。
- DISCONNECT 文を発行し、データベースに再接続します。それから、名前の同じカーソルを別の選択式で宣言するか、カーソルを指定したデータベースと同じエイリアスでデータベースに接続します。DISCONNECT 文の詳細は、15.6 項 (15-15) を参照してください。

参考：『Oracle Rdb7 Introduction to SQL』にも DISCONNECT 文に関する詳しい説明があります。

18.5 保持可能カーソルの使用方法

保持可能カーソルを使用すると、すべてのトランザクションを通じてカーソルをオープンした状態に保持できるので、テーブル・カーソルの使用と配置に関する制御機能が拡張されます。保持可能カーソルを宣言すると、トランザクションが終了してもそのカーソルはオープンな状態で残り、結果テーブル内の行ポインタの位置を保持します。

アプリケーションで、各行をフェッチした後、それを変更する前にユーザーのクエリーが必要な場合は、保持可能カーソルの使用を検討してください。保持可能カーソルを使用すると、トランザクションのコミットまたはロールバックによってロックは解放されますが、カーソルの位置はそのまま保持されます。

保持可能カーソルには、次のオプションがあります。

- **WITH HOLD PRESERVE ON COMMIT** 句を使用すると、トランザクションをコミットしてもカーソルはオープンな状態を保持するように指定されます。トランザクションをロールバックすると、SQL ではカーソルを暗黙的にクローズします。
PRESERVE 句は SQL 標準の拡張であり、**PRESERVE ON COMMIT** がデフォルトなので、Oracle Rdb では **WITH HOLD** 句のみ使用してください。
- **WITH HOLD PRESERVE ON ROLLBACK** 句を使用すると、トランザクションをロールバックしてもカーソルはオープンな状態を保持するように指定されます。トランザクションをコミットした場合は、SQL が暗黙的にカーソルをクローズします。
- **WITH HOLD PRESERVE ALL** 句を使用すると、トランザクションをコミットまたはロールバックしてもカーソルはオープンな状態を保持するように指定します。
- **WITH HOLD PRESERVE NONE** 句は、**SET HOLD CURSOR** 句より優先されます。

例 18-2 (18-12) に、保持可能カーソルの動作を示します。例 18-1 (18-9) に示す基本的なテーブル・カーソルと比較してください。

例 18-2 保持可能テーブル・カーソルの使用方法

```
SQL> DECLARE CURRENT_MBMF TABLE CURSOR WITH HOLD PRESERVE ON COMMIT FOR
1>   SELECT JOB_CODE, EMPLOYEE_ID, JOB_START FROM JOB_HISTORY
2>   WHERE (DEPARTMENT_CODE = 'MBMF') AND
3>         (JOB_END IS NULL)
4>   ORDER BY JOB_CODE, JOB_START;
SQL> --
SQL> OPEN CURRENT_MBMF;
SQL> --
SQL> -- The first FETCH statement retrieves the first row.
SQL> FETCH CURRENT_MBMF;
   JOB_CODE  EMPLOYEE_ID  JOB_START
   -----
   APMG      00174        22-Sep-1981
SQL> FETCH CURRENT_MBMF;
   JOB_CODE  EMPLOYEE_ID  JOB_START
   -----
   ASCK      00165         8-Mar-1981
SQL> --
SQL> -- Commit the transaction.
SQL> COMMIT;
SQL> --
SQL> -- Because this is a holdable cursor using the WITH HOLD
SQL> -- PRESERVE ON COMMIT clause, the cursor remains open and
SQL> -- the row pointer retains its position. The next FETCH
```

```
SQL> -- statement reads the next record.
SQL> FETCH CURRENT_MBMF;
      JOB_CODE      EMPLOYEE_ID      JOB_START
      -----
      DMGR          00227            25-Nov-1981
```

SET HOLD CURSOR 文を使用すると、WITH HOLD 句の指定のないすべてのカーソルに対して、デフォルトの保持属性を指定できます。SET HOLD CURSORS 文は、SET HOLD CURSORS 文を指定した後に宣言したすべてのカーソルの保持属性を制御します。たとえば、WITH HOLD 句の指定のないすべてのカーソルをコミットで強制的にオープン状態にしておくには、次の文を使用します。

```
SQL> SET HOLD CURSORS 'ON COMMIT';
```

18.6 リスト・カーソルの使用方法

リスト・カーソルを使用すると、大きなデータ構造（リスト）の行からの読出しや行への挿入を実行できます。リストとは、データ型 LIST OF BYTE VARYING の要素やセグメントを順序通りに並べたコレクションです。リストは、ビデオ・ディスプレイ、長いテキスト、オーディオ録音など、マルチメディア・オブジェクトの格納によく使用されます。

リストは、テーブル行の要素群の集合であり、テーブル・カーソルを使用してテーブル内の各行の位置を指定するため、リスト・カーソルは、テーブル・カーソルを参照する必要があります。

リスト・カーソルを使用したリストの読取りや挿入は可能ですが、リスト・カーソルを使用してリストを更新することはできません。

リストの読取りや挿入を行うには、まずテーブル・カーソルを宣言し、テーブル列とリスト列の名前の両方を選択リストで指定してください。テーブル・カーソルは、リスト・カーソルの行コンテキストを提供します。言い換えると、行 1 のリストの読取りまたは挿入を実行する前に、テーブル・カーソルを行 1 に位置付けなければなりません。

テーブル・カーソルを宣言したら、次にリスト・カーソルを宣言し、DECLARE CURSOR 文の WHERE CURRENT OF 句でテーブル・カーソルを指定します。WHERE CURRENT OF 句は、リスト・カーソルとテーブル・カーソルを関連付けます。

リスト・カーソルをオープンする前に、テーブル・カーソルをオープンし、FETCH 文で、カーソルを目的の行に移動してください。例 18-3 (18-13) では、リスト・カーソルと読取りリストを宣言しています。この例では、従業員 ID を読み取り、RESUMES テーブルに戻ります。RESUME 列は、データ型 LIST OF BYTE VARYING で定義されています。

例 18-3 リスト・カーソルの使用方法

```
SQL> -- Declare the table cursor.
SQL> DECLARE RESUME_CURS TABLE CURSOR FOR
      1>   SELECT EMPLOYEE_ID, RESUME FROM RESUMES;
SQL> --
```

```
SQL> -- Open the table cursor.
SQL> OPEN RESUME_CURS;
SQL> --
SQL> -- Fetch the first row. The table cursor returns the employee ID
SQL> -- and segmented string identifier for the RESUME column.
SQL> --
SQL> FETCH RESUME_CURS;
      EMPLOYEE_ID  RESUME
      00164              86:2:4
SQL> --
SQL> -- To read the contents of the RESUME column, you must declare
SQL> -- a list cursor and position it on the row.
SQL> --
SQL> DECLARE LIST_CURS LIST CURSOR FOR
      1> SELECT RESUME WHERE CURRENT OF RESUME_CURS;
SQL> --
SQL> -- Open the list cursor.
SQL> OPEN LIST_CURS;
SQL> --
SQL> -- Fetch the first segment of the list. Remember that
SQL> -- the table cursor is already opened and positioned on
SQL> -- the row containing employee ID 00164.
SQL> FETCH LIST_CURS;
      RESUME
      This is the resume for Alvin Toliver
SQL> --
SQL> -- Fetch the remaining segments of the list.
SQL> FETCH LIST_CURS;
      RESUME
      Boston, MA
SQL> FETCH LIST_CURS;
      RESUME
      Oracle Corporation
SQL> FETCH LIST_CURS;
      RESUME
      %RDB-E-STREAM_EOF, attempt to fetch past end of record stream
SQL> --
SQL> -- To move to the next row, first close the list cursor, then
SQL> -- fetch the next table row.
SQL> --
SQL> CLOSE LIST_CURS;
SQL> FETCH RESUME_CURS;
      EMPLOYEE_ID  RESUME
      00165              86:2:8
SQL> --
SQL> -- Open the list cursor.
SQL> OPEN LIST_CURS;
```

```
SQL> -- Fetch the first segment of the list for employee 00165.
SQL> FETCH LIST_CURS;
      RESUME
      This is the resume for Terry Smith
SQL> --
SQL> -- Close the cursors.
SQL> CLOSE LIST_CURS;
SQL> CLOSE RESUME_CURS;
```

18.7 スクロール可能なリスト・カーソルの使用方法

スクロール可能なリスト・カーソルと、一方向の（スクロール不可能な）リスト・カーソルの違いは、スクロール可能なリスト・カーソルを使用すると、アプリケーションが結果テーブルをランダムにスキャンできる点です。アプリケーションは、上下方向のフェッチ、最初と最後の行の直接フェッチ、ランダムな任意の1行のフェッチなどが可能になります。次の例では、スクロール可能なリスト・カーソルを宣言します。

```
DECLARE BCURSOR SCROLL LIST CURSOR
      FOR SELECT RESUME WHERE CURRENT OF ACURSOR;
```

FETCH 文は、SCROLL リスト・カーソルで、次のオプションを実行できます。

- FETCH NEXT
- FETCH PRIOR
- FETCH FIRST
- FETCH LAST
- FETCH RELATIVE simple-value-expression
- FETCH ABSOLUTE simple-value-expression

simple-value-expression は、正または負の整数、モジュール言語またはホスト言語の数値パラメータである必要があります。これらのオプションの詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

例 18-4 (18-15) は、スクロール可能なリスト・カーソルを使用する埋込み SQL 文を使用した C プログラムの抜粋です。

例 18-4 スクロール可能なリスト・カーソルの使用方法

```
.
.
.
/* Declare a table cursor. */
EXEC SQL DECLARE ONE TABLE CURSOR
      FOR SELECT EMPLOYEE_ID, RESUME FROM RESUMES
      WHERE EMPLOYEE_ID = :sel_emp_id ;
```

```
EXEC SQL DECLARE TWO READ ONLY SCROLL LIST CURSOR
          FOR SELECT RESUME
          WHERE CURRENT OF ONE;

EXEC SQL OPEN ONE;
      dump_error();
EXEC SQL FETCH ONE INTO :emp_id, blob;
      dump_error();
EXEC SQL OPEN TWO;
      dump_error();
EXEC SQL FETCH LAST FROM TWO INTO :seg2;
      printf("Segment returned: %s\n", seg2 );
      dump_error();
EXEC SQL FETCH NEXT FROM TWO INTO :seg2;
      printf("Segment returned: %s\n", seg2 );
      dump_error();
EXEC SQL FETCH FIRST FROM TWO INTO :seg2;
      printf("Segment returned: %s\n", seg2 );
      dump_error();
EXEC SQL FETCH NEXT FROM TWO INTO :seg2;
      printf("Segment returned: %s\n", seg2 );
      dump_error();
EXEC SQL FETCH NEXT FROM TWO INTO :seg2;
      printf("Segment returned: %s\n", seg2 );
      dump_error();
EXEC SQL FETCH NEXT FROM TWO INTO :seg2;
      printf("Segment returned: %s\n", seg2 );
      dump_error();
EXEC SQL FETCH FIRST FROM TWO INTO :seg2;
      printf("Segment returned: %s\n", seg2 );
      dump_error();
EXEC SQL FETCH LAST FROM TWO INTO :seg2;
      printf("Segment returned: %s\n", seg2 );
      dump_error();
EXEC SQL FETCH PRIOR FROM TWO INTO :seg2;
      printf("Segment returned: %s\n", seg2 );
      dump_error();
EXEC SQL FETCH ABSOLUTE 1 FROM TWO INTO :seg2;
      printf("Segment returned: %s\n", seg2 );
      dump_error();
two_s = 2;
EXEC SQL FETCH ABSOLUTE :two_s FROM TWO INTO :seg2;
      printf("Segment returned: %s\n", seg2 );
      dump_error();
two_l = 2;
```

```
EXEC SQL FETCH ABSOLUTE :two_1 FROM TWO INTO :seg2;
printf("Segment returned: %s\n", seg2 );
dump_error();
.
.
.
```

18.8 動的カーソルの使用方法

動的カーソルは、動的 SQL プログラムで使⽤します。動的カーソルは通常のカーソル（静的カーソル）とは異なり、SELECT 文をソース・コードで明示的には指定しません。かわりに、プリコンパイルされた SQL 文の名前を指定します。カーソル名はコンパイル時に認識されますが、SELECT 文は実行時に初めて認識されます。ユーザーがどんなクエリーを⼊力するかわからない場合は、動的カーソルを使⽤してください。

動的カーソルを使⽤するには、まず動的な DECLARE CURSOR 文でカーソルを宣言します。それから、動的に実行する SELECT 文を PREPARE 文でプリコンパイルします。

例 18-5 (18-17) に、動的カーソルを使⽤してテーブルから⾏をフェッチする SQL モジュールを示します。

例 18-5 動的カーソルの使用方法

```
-- This module uses dynamic cursors to fetch rows one at a time.
--
MODULE          C_MOD_DYN_CURS
LANGUAGE        GENERAL
AUTHORIZATION   RDB$DBHANDLE
PARAMETER COLONS
DECLARE ALIAS FILENAME personnel
-- Declare the dynamic cursor. Use a statement name to identify
-- a prepared SELECT statement.
DECLARE CURSOR1 CURSOR FOR STMT_NAME
-- Prepare the statement from a statement entered at run time.
-- Specify that SQL write information about the number and
-- data type of select list items to the SQLDA.
PROCEDURE PREP_STMT
    (SQLCODE,
     :COMMAND_STRING CHAR (256));
    PREPARE STMT_NAME FROM :COMMAND_STRING;
PROCEDURE DESCRIBE
    (SQLCODE,
     SQLDA);
    DESCRIBE STMT_NAME SELECT LIST INTO SQLDA;
PROCEDURE OPEN_CURSOR
    (SQLCODE);
```

```
OPEN CURSOR1;
PROCEDURE FETCH_CURSOR
  (SQLCODE,
   SQLDA);
  FETCH CURSOR1 USING DESCRIPTOR SQLDA;
PROCEDURE CLOSE_CURSOR
  (SQLCODE);
  CLOSE CURSOR1;
PROCEDURE ROLLBACK
  (SQLCODE);
  ROLLBACK;
```

動的 SQL と動的カーソルの詳細は、11 章 (11-1) を参照してください。

18.9 拡張動的カーソルの使用方法

拡張動的カーソルは、動的 SQL プログラムで使用します。拡張動的カーソルを使用する場合は、カーソル名と SELECT 文の両方にパラメータを指定します。カーソル名も SELECT 文も、実行時に初めて指定されます。拡張動的カーソルを使用すると、一セットのカーソル関連の文で動的に生成された任意の数の文を処理できます。

拡張動的カーソルを使用するには、まず、拡張動的 DECLARE CURSOR 文でカーソルを宣言します。それから、動的に実行する SELECT 文を PREPARE 文でプリコンパイルします。

例 18-6 (18-18) に、拡張動的カーソルを使用してテーブルから行をフェッチする SQL モジュールを示します。

例 18-6 拡張動的カーソルの使用方法

```
--
-- This module uses extended dynamic cursors to fetch rows one at a time.
--
MODULE          MOD_C_EXTDYN_CURS
LANGUAGE        C
AUTHORIZATION   RDB$DBHANDLE
PARAMETER COLONS
DECLARE ALIAS FILENAME personnel
-- Prepare the statement from a statement entered at run time.
PROCEDURE PREP_STMT
  (SQLCODE,
   :STMTID INTEGER,
   :COMMAND_STRING CHAR (256),
   SQLDA);
  PREPARE :STMTID FROM :COMMAND_STRING;
-- Specify that SQL write information about the number and
-- data type of select list items to the SQLDA.
PROCEDURE DESCRIBE
```



```
(SQLCODE,
:STMTID INTEGER,
SQLDA);
    DESCRIBE :STMTID SELECT LIST INTO SQLDA;
-- Declare an extended dynamic cursor.
PROCEDURE DEC_CUR_FROM_STMT
    (SQLCODE,
     :CURSOR_NAME CHAR(32),
     :STMTID INTEGER );
    DECLARE :CURSOR_NAME CURSOR FOR :STMTID;
PROCEDURE OPEN_CURSOR_NAME
    (SQLCODE,
     :CURSOR_NAME CHAR(32));
    OPEN :CURSOR_NAME;
PROCEDURE FETCH_CURSOR_NAME
    (SQLCODE,
     :CURSOR_NAME CHAR(32),
     SQLDA);
    FETCH :CURSOR_NAME USING DESCRIPTOR SQLDA;
PROCEDURE CLOSE_CURSOR_NAME
    (SQLCODE,
     :CURSOR_NAME CHAR(32));
    CLOSE :CURSOR_NAME;
PROCEDURE ROLLBACK
    (SQLCODE);
    ROLLBACK;
```

動的 SQL と拡張動的カーソルの詳細は、11 章 [\(11-1\)](#) を参照してください。

データの挿入、更新、削除

この章では、プログラムを使用してテーブル行を挿入、更新、削除する方法を説明します。リスト・カーソルを使用して、大規模なデータ構造をデータベースに挿入する方法も示します。

この章の各項では、次の作業の方法を説明します。

- データベースのロード
- 行の挿入
- リスト・カーソルによる大規模なデータ構造の挿入
- 行の更新
- 行の削除
- トリガーによる挿入、更新、削除操作

19.1 データベースのロード

SQL でデータをテーブルにロードするには、INSERT 文を使用します。INSERT 文については、19.2 項 (19-2) で説明します。

大量のデータをファイルまたは別のテーブルからロードする場合は、『Oracle Rdb7 Guide to Database Design and Definition』のデータのロードに関する章を参照してください。その章では、データをロードする様々な方法を示し、ロード操作の効率を上げる方法を説明しています。

"sql_load" で始まる複数のオンライン・プログラムは、すべてサンプル・データベースのテーブルに行を格納する方法の例を示しています。サンプル・ディレクトリ内のプログラムは、複数のホスト言語で使用できます。

19.2 行の挿入

行を格納するには、INSERT 文を使用します。INSERT 文の一般的な書式は次の 2 つです。

- INSERT...VALUES
- INSERT...SELECT

両方の書式によって、テーブルまたはビューと値を格納する列が特定されます。ただし、INSERT...VALUES の書式では 1 行のみの値を指定するのに対して、INSERT...SELECT の書式では 1 行以上に対して値を指定します。

19.2.1 INSERT...VALUES 文の使用法

INSERT 文の最初の書式、INSERT...VALUES では、VALUES 句に指定したデータを挿入します。データは VALUES 句で明示的に指定できますが、ユーザーがデータ・ファイルからロードするデータをパラメータで表すこともできます。

例 19-1 (19-2) に示す sql_load_employees.sc プログラムの抜粋では、データ・ファイルから EMPLOYEES テーブルをロードしています。

例 19-1 プリコンパイル C プログラムにおけるデータ・ファイルからのテーブルのロード

```
.
.
.
/*
In the INSERT statement, the list of names in the VALUES clause corresponds
to the host variables containing the values. The list of names that follows
the INSERT clause names the columns in the table that are to be inserted.
*/
EXEC SQL
    -- Insert the row.
```

```

INSERT INTO EMPLOYEES
    (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
     ADDRESS_DATA_1, CITY, STATE, POSTAL_CODE, SEX, BIRTHDAY,
     STATUS_CODE)
VALUES
    (:emp_id, :last, :first, :middle:middle_ind,
     :address, :city, :state, :zip, :sex,
     :date_out, :status);
.
.
.

```

行を挿入するプログラムを作成する場合は、特定の列に実際の値か NULL 値を格納します。この場合、すべての行について実際の値が格納されているか NULL 値が格納されているかを予測することはできません。たとえば、例 19-1 (19-2) では MIDDLE_INITIAL 列に NULL 値がロードされています。NULL 値を処理するために、SQL に標識パラメータを渡します。

プログラムにおける格納された値と NULL 値の処理の詳細は、8.10 項 (8-18) を参照してください。

19.2.2 INSERT...SELECT 文の使用法

INSERT 文の 2 番目の書式、INSERT...SELECT では、別のソースからテーブルに行を格納します。この書式は、複製データベースのデータを抽出する場合に便利です。データを格納済みのテーブルを再構築する場合にも便利です。

この INSERT 文の書式は、対話型 SQL でもプログラムでも使用できます。テーブルに挿入する行が他のテーブルまたはビューから転送される場合、プログラムには列の値を格納する変数の宣言がないことに注意してください。選択式でデータを指定すると、SQL はプログラムによる中間処理なしでデータを転送できます。

INSERT 文を使用するには、次の処理を行います。

- 通知機能を実行し（ビューや複雑なクエリーなど）、実際に値を格納するテーブルのデータを更新します。ユーザーにこのテーブルの読取り専用の権限を付与します。ビューを使用すると、多くのテーブルに複雑な方法でアクセスできます。多くのユーザーとプログラムがビューを介して頻繁に情報を要求する場合は、データベース・パフォーマンスが低下する恐れがあります。ビューを介してデータを検索する一部のユーザーが絶対的に最新の情報でなくても満足する場合は、別のテーブルを作成してデータを格納します。この場合、この新たなテーブルの行を更新する INSERT 文はまれにしかビューを読み込みません。この方法では、データベース・システムが複雑な操作を実行する回数が減るので、データベース・パフォーマンスが向上します。
- 複数のクエリーによるデータを唯一のテーブルに格納します。UNION 演算子を使用して 1 つのテーブルの列の値を別のテーブルの列の値と組み合わせると、その結果をテーブルに格納できます。

- リモート・データベースのデータを独自のシステムにコピーします。
たとえば、Oracle Rdb のリモート・データベースのデータまたは Oracle Rdb データベース以外のデータベースのデータを頻繁に使用するクエリーかレポートを計画している場合は、リモート・データをローカル・データベースに定期的にコピーします。これで、ローカル・データベースに関するクエリーとレポートを生成できます。ローカル・データベースに関するクエリーの応答時間は、リモート・データベースに関するクエリーの応答時間より短くなります。
- 複雑なレポートや自発的なレポート、また "what if" の計算の基礎として一時的に作成された 1 つまたは複数のテーブルにデータを格納します。
複雑な計算やクエリーを処理することがわかっている場合、特にそれが多くのテーブルにアクセスする場合は、テーブルを作成して作業の早期に一時的に使用できます（これは、同じ種類のクエリーを繰り返し入力して、クエリーを改良する時期にあたります）。これらのテーブルには、オリジナル・テーブルを代表するサブセットが含まれます。このテーブルを使用する間はより高速に結果が得られ、永久的なテーブルにアクセスするユーザーを妨害することはありません。クエリーまたは計算が精製された場合は、作成したテーブルを削除し、永久的なテーブルを指定するようにクエリーを修正します。

テーブルを再構築する作業の一環として、INSERT 文に選択式を指定することもできます。

サイトで使用するデータベース内のテーブルを削除して再作成する場合は、次のガイドラインに従ってください。

- 必要な保護対策をとり、元の形式のデータを保護します（データベースのバックアップ）。
- 終了するまで、処理するテーブルへの他のユーザーによる読取りと書き込みを回避します（このために、テーブルを予約し、SET TRANSACTION 文で EXCLUSIVE WRITE を指定します）。
ピボット・テーブルを変更する場合は、唯一のユーザーがデータベースにアクセスするようにしてください。それ以外の場合、他のユーザーの操作を混乱させたり、他のユーザーの操作と競合したりすることがあります。

19.3 リスト・カーソルを使用した大きなデータ構造の挿入

テーブルにリストが含まれる場合、データをリストに挿入するには、リスト・カーソルを使用する必要があります。リストとは、データ型 LIST OF BYTE VARYING の要素群を順序通りに並べたものです。リストは、セグメント文字列と呼ばれることもあります。リスト・カーソルを使うと、大規模なオブジェクトをサポートできない言語からでも、非常に大きなデータ構造をスキャンできます。

まず、DECLARE CURSOR 文の FROM 句で、テーブル・カーソルを宣言し、テーブル列とリスト列の名前を指定します。テーブル・カーソルを宣言したら、DECLARE CURSOR 文の WHERE CURRENT OF 句でリスト・カーソルを宣言し、テーブル・カーソルを指定します。WHERE CURRENT OF 句は、リスト・カーソルとテーブル・カーソルを関連付けます。18.6 項 (18-13) では、リスト・カーソルをさらに詳細に説明しています。

リストを既存の行に挿入するには、挿入専用モードでリスト・カーソルを使用する必要があります。リスト・カーソルを開く前に、テーブル・カーソルを開き、FETCH 文で、カーソルを目的の行に移動します。

リストを含む行を新しく挿入する場合は、テーブル・カーソルとリスト・カーソルの両方を挿入専用モードで使用してください。SQL は、前のリストを新しく挿入するデータで置き換えるので、既存のリストには、データを追加しないでください。

サンプル・ディレクトリにある `sql_resumes` プログラムは、リスト・カーソルを使用して行にリストを挿入する方法を示しています。

19.4 行の更新

既存の行の値を変更するには、UPDATE 文を使用します。行の挿入とは異なり、行の更新では、変更対象となる行（または複数の行）を選択する必要があります。操作はさらに複雑になります。UPDATE 文には、いくつかの使用方法があります。

- UPDATE 文で、データ選択条件を指定する方法
- UPDATE 文とカーソルを使用する方法
- UPDATE...RETURNING 文を使用する方法

参考: 行の更新の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。UPDATE 文の項では、更新操作を詳細に説明しています。DECLARE CURSOR 文、CREATE VIEW 文（ビューの更新の例）、FETCH 文（INTO 句）の項では、データ変さらに関するガイドラインを説明しています。

19.4.1 UPDATE 文でのデータの選択

1 行のみの値を変更する場合は、UPDATE 文と SET 句を使用します。この UPDATE 文は、次のような書式を持ちます。

```
UPDATE table-or-view-name SET list-of-column-assignments
WHERE condition(s)-identifying-row(s)-to-be-modified ;
```

この方法で、複数の行を更新できますが、すべての列オカレンスに同じ値の式を設定している場合のみ更新できます。たとえば、UPDATE 文の WHERE 句で複数の JOB_HISTORY 行を指定して、すべての行について上司 ID に同じ値を格納します。

次は、プリコンパイルした SQL プログラムの抜粋であり、このような更新文を示しています。

```
EXEC SQL UPDATE JOB_HISTORY SET JOB_END = :BINTIMOUT
WHERE EMPLOYEE_ID = :ID-NUMBER
AND JOB_END IS NULL;
```

```

EXEC SQL UPDATE SALARY_HISTORY SET SALARY_END = :BINTIMOUT
WHERE EMPLOYEE_ID = :ID-NUMBER
AND SALARY_END IS NULL;

```

19.4.2 カーソルを使用した UPDATE 文

一度に複数の行を変更するには、UPDATE 文でカーソルを使用します。この UPDATE 文は、次のような書式を持ちます。

```

DECLARE cursor-name CURSOR FOR
    select-expression
    FOR UPDATE OF list-of-columns-to-be-modified ;
OPEN cursor-name;

--
FETCH cursor name;          | Repeat FETCH and
UPDATE table-or-view-name   | UPDATE statements
    SET list-of-column-assignments | to change each
    WHERE CURRENT OF cursor-name ; | row in the cursor.
--

```

この場合、行を個々に処理するので、各行の更新列には、異なる値を指定できます。

カーソル宣言で FOR UPDATE 句を指定すると、更新したい列を指定できます。カーソルの宣言に FOR UPDATE 句を含めるかどうかに関係なく、カーソルから行を更新できます。ただし、FOR UPDATE 句を指定し、この句で指定したものの以外の列を更新すると、更新対象以外の列を更新したことになるので、SQL は警告メッセージを返します。

行を更新するためにカーソルを宣言する場合、DECLARE CURSOR 文には、行の更新を許可する SELECT 文の構文を指定してください。SELECT 文による行の更新の許可に関するルールは、ANSI/ISO SQL 規格と Oracle Rdb で異なります。DECLARE CURSOR 文とオプションの WHERE 句で 1 つのテーブルを参照する場合、カーソルは行の更新を許可すると考えれば安全です。

例 19-2 (19-6) は、プリコンパイルした SQL プログラムでの、カーソルを使用した UPDATE 文を示しています。この例は、プリコンパイルした C プログラム `sql_terminate.sc` のソース・コードからの抜粋です。プログラムの完全なソース・ファイルは、サンプル・ディレクトリに格納されているので、オンラインで参照できます（他の言語のソース・ファイルも収録）。

例 19-2 プリコンパイルした C プログラムでの行の更新


```

/* ABSTRACT:
*
*      This program demonstrates the use of a cursor to fetch and update
*      rows in the database using the C precompiler.
*
*      This program prompts the operator for the employee ID and termination
*      date of an employee, until the operator asks to exit. It uses
*      the employee ID to check for the employee by opening a cursor
*      and fetching the employee row. If the employee is found in the
*      database, it updates three tables with the employee's status and
*      termination date. Then, it commits the transaction and prompts
*      the operator for the next request.
*/
.
.
.
/* Declare the cursor to be used for fetching employee records. */
EXEC SQL DECLARE EMPLROW CURSOR FOR
        SELECT EMPLOYEE_ID, STATUS_CODE FROM EMPLOYEES
        WHERE EMPLOYEE_ID = :employee_id;
.
.
.
/* Open the cursor that had been previously declared. */
EXEC SQL OPEN EMPLROW;
.
.
.
/* Fetch a row from the opened cursor. */
EXEC SQL FETCH EMPLROW INTO :e_id, :status_code;
.
.
.
/* Update the EMPLOYEES table. */
EXEC SQL UPDATE EMPLOYEES
        SET STATUS_CODE = '0'
        WHERE CURRENT OF EMPLROW;
.
.
.

```

19.4.3 UPDATE...RETURNING 文の使用方法

UPDATE...RETURNING 文は、更新した値をプログラムに返すことを要求できます。使用できるのは、1 行のみの更新です。

行を更新した後、データベース・システムが、行の dbkey を返すように要求できます。その後続くクエリーは、この dbkey の値を使用してレコードに直接アクセスできます。

次の例は、UPDATE 文を使用して、dbkey を返す方法を示しています。

```
EXEC SQL UPDATE JOB_HISTORY
      SET JOB_END = :BINTIMOUT
      WHERE EMPLOYEE_ID = :ID-NUMBER
      AND JOB_END IS NULL
      RETURNING DBKEY INTO :TERM_DBKEY;
```

これに続くクエリーは、dbkey の値を使用して、終了便益率の計算やステート・エージェンシへのレポート発行などの作業のためにジョブ履歴レコードを検索できます。

DBKEY キーワードのかわりに、ROWID キーワードを使用することも可能です。

次のように、RETURNING 句は複数の値を返すことができ、値は dbkey である必要はありません。

```
EXEC SQL UPDATE ACCOUNT
      SET AMOUNT = AMOUNT + :TXN
      WHERE ACCOUNT_NUMBER = :ACCNUM
      RETURNING AMOUNT INTO :NEW_AMOUNT;
```

19.5 行の削除

DELETE 文を使用すると、テーブルから行を削除できます。DELETE 文は、単独でもカーソルと組み合わせても使用できます。

DELETE 文を単独で使用する場合は、テーブル名と削除する行を特定する条件を指定します。

```
DELETE FROM EMPLOYEES
      WHERE EMPLOYEE_ID = '00164';
```

DELETE 文をカーソルと組み合わせて使用する場合は、テーブル名と CURRENT OF 句を使用して、カーソルが指す行を削除するよう指定します。次の例に、カーソルの宣言、関連する各カーソル文、DELETE 文を示します。

```
-- Declare the cursor.
DECLARE DEL_CURSOR CURSOR FOR
      SELECT DEPARTMENT_CODE, DEPARTMENT_NAME
      FROM DEPARTMENTS;

OPEN DEL_CURSOR;
-- Fetch the row.
FETCH DEL_CURSOR;
-- Delete the current row.
DELETE FROM DEPARTMENTS WHERE CURRENT OF DEL_CURSOR;
```

カーソルを宣言するときにオプションで FOR UPDATE 句を指定すると、更新する列を特定できます。

行を削除するためにカーソルを宣言する場合は、DECLARE CURSOR 文に行を削除できる SELECT 文の構文を指定する必要があります。SELECT 文で行を削除できるかどうかを決定する規則は、ANSI/ISO SQL 標準と Oracle Rdb で様々な定義されています。DECLARE CURSOR 文と任意指定の WHERE 句で 1 つのテーブルを参照する場合は、一般にカーソルによって行が削除されることを想定するのが安全です。

参考: DELETE 文の詳細は、『Oracle Rdb7 SQL Reference Manual』の SQL 文に関する章を参照してください。

19.6 リスト・データの削除

データ型 LIST の列は、行全体を削除しない限り削除できません。ただし、例 19-3 (19-9) に示すように、列に NULL を設定することは許可されます。

例 19-3 行からのリスト・データの削除

```
SQL> -- Set the list column, RESUME, to NULL
SQL> UPDATE RESUMES
  1>       SET RESUME = NULL
  2>       WHERE EMPLOYEE_ID = '00164';
1 row updated
SQL> select * from resumes;
EMPLOYEE_ID  RESUME
00164        NULL
00165                        86:2:8
00166                        86:2:12
3 rows selected
```

19.7 挿入、更新、削除操作でトリガーを使用する方法

定義済みのトリガーを使用してデータベース内のデータの挿入、更新、削除を実行すると、データベースに関する他の処理が自動的に実行されます。トリガーを使用すると、特定の書き込み操作を実行する前と後に、1 つ以上の処理が実行されます。たとえば、データベースを変更するユーザーと変更の日付と時刻を追跡するトリガーを定義できます。『Oracle Rdb7 Guide to Database Design and Definition』では、トリガーを定義する方法を説明し、CURRENT_TIMESTAMP 関数を使用してデータの変更を追跡するトリガー定義の例を示します。

複数のスキーマ・オプションの使用法

Oracle Rdb データベースには、複数のスキーマを含めることができます。Oracle Rdb への SQL インタフェースを使用すると、データベース内のカタログおよび複数のカタログ内にある複数スキーマ（マルチスキーマ）の ANSI/ISO 標準命名法を使用できるようになります。マルチスキーマ SQL オプションは、SQL がこれらのオブジェクトに名前を付ける方法を変更するだけで、Oracle Rdb がオブジェクトを保管する方法は変更しません。

この項では、次の方法について説明します。

- マルチスキーマ・データベースに関連した用語と概念を使用
- SQL モジュール言語で記述されたプログラムのマルチスキーマ・オブジェクトの命名
- プリコンパイル SQL プログラムのマルチスキーマ・オブジェクトの命名

参考: マルチスキーマ SQL オプションを完全に理解するには、次のマニュアルを参照してください。

- 『Oracle Rdb7 SQL Reference Manual』は、マルチスキーマの用語、マルチスキーマ・データベースの作成と変更を行う構文、マルチスキーマ命名規約および SQL モジュール言語と SQL プリコンパイラでマルチスキーマ・オプションを使用する方法についての参照情報を説明しています。
 - 『Oracle Rdb7 Guide to Database Design and Definition』は、マルチスキーマ・データベースとオブジェクトの変更、作成および削除方法の説明と、この章を通して例として使用しているサンプルのマルチスキーマ `corporate_data` データベースのデータ定義を示しています。
-

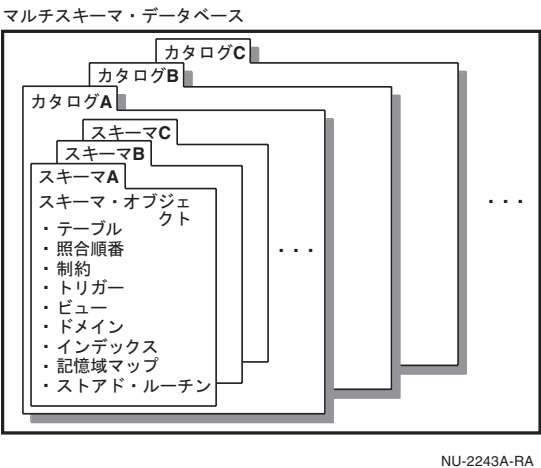
20.1 マルチスキーマ・データベースについて

SQL は、次に示すようにマルチスキーマ Oracle Rdb データベースでスキーマとカタログ・オブジェクトを定義します。

- **スキーマ**とは、カタログ内のオブジェクトのグループです。スキーマには、テーブル、ビュー、制約、トリガー、ドメイン、照合順番、ストレージ・マップ、インデックスおよびこれらのオブジェクトの権限が含まれています。マルチスキーマ・データベースのカタログで1つ以上のスキーマを作成できます。
各マルチスキーマ・データベースには、RDB\$SCHEMA と呼ばれる少なくとも1つのスキーマが含まれており、これはデフォルトのデータベース・カタログである RDB\$CATALOG に含まれています。
- **カタログ**とは、マルチスキーマ・データベース内のスキーマのグループを識別するマルチスキーマ・オブジェクトです。マルチスキーマ・データベースで1つ以上のカタログを作成できます。
各マルチスキーマ・データベースには、RDB\$CATALOG と呼ばれるカタログが少なくとも1つ含まれており、これはデータベース・スキーマである RDB\$SCHEMA を含んでいます。

したがって、**マルチスキーマ・データベース**には、それぞれが1つ以上のスキーマを含むことができる1つ以上のカタログが含まれています。図 20-1 (20-2) は、マルチスキーマ・データベースを示します。

図 20-1 マルチスキーマ SQL データベースの構造



CREATE DATABASE 文または ALTER DATABASE 文の MULTISCHEMA IS ON 句を指定することで Oracle Rdb データベース上のマルチスキーマ属性を有効にできます。デフォルトでは、SQL はただ1つのスキーマでデータベースを作成し、そのデータベースにスキーマを追加することはできません。

20.2 SQL モジュール・プロセッサでのマルチスキーマ・データベースの使用法

この項では、SQL モジュールの処理を行う場合に適用するデフォルトの設定を説明します。ここでは、マルチスキーマ名を使用するサンプル・モジュール・ファイルを示します。

20.2.1 SQL モジュールのデフォルトの設定

マルチスキーマ属性を持つ Oracle Rdb データベースを使用する場合、モジュール・ヘッダーでデフォルトのカatalogとデフォルトのスキーマを設定できます。デフォルトのスキーマとCatalogを設定すれば、参照する各オブジェクトを修飾するためのCatalog名とスキーマ名を含める必要はありません。デフォルト・スキーマおよびデフォルト・Catalog内のオブジェクトは、スキーマおよびCatalog機能がなくても参照できます。したがって、1つのスキーマおよび1つのCatalogで主に作業を行う予定であれば、モジュールのヘッダーにデフォルトを設定します。

コンテキスト・ファイルでデフォルトのCatalogおよびスキーマを指定することもできます。

スキーマ・オブジェクトを修飾せず、SQL コンテキストがあいまいである場合、SQL は、表 20-1 (20-3) で示しているデフォルトを使用します。

表 20-1 マルチスキーマ SQL のモジュールのデフォルト

コンパイラの属性	暗黙のデフォルト
エイリアス	認可識別子
認可識別子	ユーザー名
Catalog	RDB\$CATALOG
スキーマ	認可識別子

マルチスキーマ・オプションを選択する場合、各スキーマに対するデフォルトの認可識別子は、モジュールをコンパイルしたユーザーの名前です。モジュール・ヘッダーで AUTHORIZATION 句を使用すると、異なる認可識別子を指定できます。

この認可識別子は、デフォルトのエイリアスとスキーマを定義します。スキーマ要素の名前を修飾しなければ、SQL は現行の認可識別子を使用して暗黙的に修飾します。ALIAS 句および SCHEMA 句を使用して、デフォルトの設定を上書きできます。モジュール・ヘッダーでエイリアスまたはスキーマを指定せず、モジュール・ヘッダーで認可識別子を明示的に指定している場合、SQL は、モジュール・ヘッダーで定義した認可識別子をデフォルトのエイリアスとデフォルトのスキーマとして使用します。モジュール・ヘッダーでデフォルトのスキーマ名を指定しなければ、認可識別子を指定する必要があります。RDB\$CATALOG Catalog名は、CATALOG 句で上書きされるまでデフォルトのまま残ります。

認可識別子は、権限の確認に使用されます。モジュール・ヘッダーの RESTRICT オプションで RIGHTS 句を指定すると、SQL は、ANSI/ISO SQL 標準に準拠したデフォルトの識別子に基づいて権限チェックを行い、コンパイル時の認可識別子が実行時の認可識別子に一致していなければ、プログラムは実行できません。RIGHTS INVOKER（デフォルト）を指定すると、SQL は権限チェックをこのモジュールを実行する要員のユーザー名に基づいて行います。

20.2.2 SQL モジュール・ファイルと C プログラムにおけるマルチスキーマ命名の使用法

例 20-1 (20-4) では、SQL モジュール sql_msdb_mod.sqlmod を説明します。このモジュールとこれをコールするプログラムのコピーは、samples ディレクトリにあります。

例 20-1 SQL モジュール・ファイルでのマルチスキーマ名の使用法

```
-----
-- This module shows how to use SQL module language when working with a
-- multischema database and using multischema naming conventions. The
-- C program sql_msdb.c with which this module is linked uses the sample
-- multischema database corporate_data.
-----

-- Header Information Section
-----
MODULE          sql_msdb_mod      -- Module name
LANGUAGE        C                -- Language of calling program
CATALOG         ADMINISTRATION   -- Set default catalog (1) (20-5)
SCHEMA         PERSONNEL        -- Set default schema (2) (20-5)
PARAMETER      COLONS           -- Require colons before parameter names
-----

-- DECLARE Statements Section
-----
DECLARE ALIAS FILENAME corporate_data  -- Declare the database.
-- Declare table cursor for the DEPARTMENTS table in the ACCOUNTING schema.
DECLARE DEPT_CURSOR TABLE CURSOR FOR
    SELECT DEPARTMENT_CODE, DEPARTMENT_NAME, MANAGER_ID
    FROM ACCOUNTING.DEPARTMENTS (3) (20-6)
    ORDER BY DEPARTMENT_CODE
-- Declare table cursor for the EMPLOYEES table in the PERSONNEL schema.
DECLARE EMP_CURSOR TABLE CURSOR FOR
    SELECT E.EMPLOYEE_ID, E.LAST_NAME, E.FIRST_NAME
    FROM EMPLOYEES E, JOB_HISTORY JH (4) (20-6)
    WHERE E.EMPLOYEE_ID = JH.EMPLOYEE_ID
        AND JH.DEPARTMENT_CODE = :DEPT_CODE
        AND JH.JOB_END IS NULL
    ORDER BY E.LAST_NAME
-----

-- Procedure Section
```


(5) (20-6)

```

-- Opens the cursor declared for the DEPARTMENTS table.
PROCEDURE OPEN_DEPT_CURSOR
    (SQLCODE);
    OPEN DEPT_CURSOR;
-- Opens the cursor declared for the EMPLOYEES table.
PROCEDURE OPEN_EMP_CURSOR
    (SQLCODE,
     :DEPT_CODE CHAR(4));
    OPEN EMP_CURSOR;
-- Fetches data from the opened cursor for the DEPARTMENTS table.
PROCEDURE FETCH_DEPT_DATA
    (SQLCODE,
     :DEPT_CODE    CHAR(4),
     :DEPT_NAME    CHAR(20),
     :DEPT_MAN_ID  CHAR(5));
    FETCH DEPT_CURSOR INTO
        :DEPT_CODE, :DEPT_NAME, :DEPT_MAN_ID;
-- Fetches data from the opened cursor for the EMPLOYEES table.
PROCEDURE FETCH_EMP_DATA
    (SQLCODE,
     :EMP_ID        CHAR(5),
     :EMP_LAST_NAME CHAR(20),
     :EMP_FIRST_NAME CHAR(20));
    FETCH EMP_CURSOR INTO
        :EMP_ID, :EMP_LAST_NAME, :EMP_FIRST_NAME;
-- Closes the DEPT_CURSOR cursor for the DEPARTMENTS table.
PROCEDURE CLOSE_DEPT_CURSOR
    (SQLCODE);
    CLOSE DEPT_CURSOR;
-- Closes the EMP_CURSOR cursor for the EMPLOYEES table.
PROCEDURE CLOSE_EMP_CURSOR
    (SQLCODE);
    CLOSE EMP_CURSOR;

```

次の説明は、例 20-1 (20-4) で番号を付けた項目に対応しています。

(1) CATALOG 句

モジュール・ヘッダー内のカタログ名を指定して、カタログ名で修飾することなくスキーマ・オブジェクトを参照します。CATALOG 句は、デフォルトのカタログを RDB\$CATALOG から ADMINISTRATION カタログへ変更します。これは、モジュールで 사용되는 ACCOUNTING と PERSONNEL スキーマの両方を含みます。

(2) SCHEMA 句

モジュール・ヘッダー内のスキーマ名を指定して、スキーマ名で修飾することなくスキーマ・オブジェクトを参照します。デフォルトのスキーマ名を指定しなければ、

AUTHORIZATION 句で認可識別子を指定する必要があります。指定しない場合、SQL は、"%SQL-F-SCHNOTDEF, Schema SCHWARTZ is not defined." などのエラー・メッセージを発行します。このメッセージは、SQL はモジュールをコンパイルしたユーザーのユーザー名と同じ名前のスキーマを探していることを示しています。

(3) ACCOUNTING スキーマ・オブジェクト修飾

DEPARTMENTS テーブルは、ACCOUNTING スキーマに属しており、デフォルトの PERSONNEL スキーマには属していないので、ACCOUNTING スキーマ名で DEPARTMENTS テーブルを修飾する必要があります。DEPARTMENTS テーブルを修飾しなければ、SQL はデフォルトの PERSONNEL スキーマの DEPARTMENTS テーブルから部門データを収集しようとします。しかし、これは MANAGER_ID 列を検索できないので、エラーを生成することになります。

(4) PERSONNEL スキーマ・オブジェクト修飾

EMPLOYEES と JOB_HISTORY テーブルは、デフォルトの PERSONNEL スキーマに属しているので、スキーマ名でスキーマ・オブジェクト名を修飾する必要がありません。

(5) プロシージャ・セクション

マルチスキーマ・ファイルの処理は、シングルスキーマ・ファイルの処理とまったく同じです。使用している名前のみが異なります。

samples ディレクトリには、例 20-1 (20-4) で示すモジュール・プロシージャをコールする C プログラムが含まれています。コンパイルされ、SQL モジュール・オブジェクト・ファイルとリンクされたら、プログラムは、corporate_data マルチスキーマ・データベースの各部門に属する従業員のリストを表示します。プログラムは、シングルスキーマ・データベースを処理する場合と同じ文と論理を使用します。

SQL モジュールと C プログラムをコンパイルし、リンクして、実行可能イメージを作成します。そして、イメージを実行して、他の SQL モジュール言語プログラムで行った場合と同じように、各企業部門に割り当てられた従業員を表示します。

モジュール内のどの SQL 構文が ANSI/ISO SQL 標準に従っていないかを知るには、SQL モジュール・ファイルをコンパイルするときに、FLAG_NONSTANDARD または -std 修飾子を指定します。デフォルトでは、SQL は、非標準構文のフラグを設定しません。SQL モジュール・プロセッサ・コマンドラインで利用できるその他の修飾子は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

20.3 SQL プリコンパイラでのマルチスキーマ・データベースの使用法

この項では、SQL プリコンパイラでプログラムを処理する場合に適用されるデフォルトの設定を説明します。マルチスキーマ名を使用するプリコンパイル・プログラムを示します。

samples ディレクトリには、sql_msdb_pre.sc プログラムのコピーがあります。

20.3.1 SQL プリコンパイラに対するデフォルトの設定

SQL プリコンパイラを使用する場合、DECLARE MODULE 文を使用してデフォルトのエイリアス、認可識別子、カタログおよびスキーマの設定を変更できます。この文を使用しなければ、埋込みプログラム内で属性を明示的に指定する必要があります。表 20-2 (20-7) は、SQL コンパイラ属性に対する暗黙的なコンパイル時のデフォルトを示しています。

表 20-2 プリコンパイル・プログラムのコンパイラ属性に対する SQL のデフォルト

コンパイラの属性	暗黙のデフォルト
エイリアス	RDB\$DBHANDLE
認可識別子	ユーザー名
カタログ	RDB\$CATALOG
スキーマ	認可識別子

次に、埋込みプログラムにおけるコンパイラ属性のデフォルトの設定を説明します。

- デフォルトのエイリアス
プリコンパイル・プログラムのデフォルトのエイリアスは、RDB\$DBHANDLE です。デフォルトのデータベースの指定は、デフォルトのデータベースへの参照を行う文がエイリアスを使用する必要があることを示しています。デフォルトで RDB\$DBHANDLE エイリアスを持つことができるデータベースは 1 つに限られるので、プログラムが 1 つ以上のデータベースに接続する場合に明示的なエイリアスを割り当てる必要があります。
- デフォルトの認可識別子
デフォルトの認可識別子は、埋込みプログラムをコンパイルするユーザーの認可識別子（ユーザー名）です。SQL は、権限を確認するために認可識別子を使用します。
DECLARE MODULE 文またはコンテキスト・ファイルで RIGHTS 句を指定する場合、SQL は、ANSI/ISO SQL 標準に準拠しているデフォルトの認可識別子に基づいて権限の確認を行います。
RIGHTS RESTRICTED を指定すると、SQL はプログラムがコンパイルされた認可識別子を使用して埋込みプログラムを実行するユーザーの名前を比較します。SQL は、プログラムをコンパイルしたユーザー以外のユーザーがプログラムを起動することを防ぎます。デフォルトでは、SQL は RIGHTS INVOKER を使用して、プログラムを起動したユーザーのユーザー名に基づいて権限の確認を行います。
- デフォルトのカタログ
プリコンパイル・プログラムに対するデフォルトのカタログである RDB\$CATALOG は、SQL モジュール言語のデフォルトと同じです。デフォルトでは、RDB\$CATALOG には、RDB\$SCHEMA スキーマが含まれます。これには、すべての Oracle Rdb システム・テーブルが含まれます。

- デフォルトのスキーマ
プリコンパイル・プログラムで名付けられたスキーマ・オブジェクトのデフォルト・スキーマは、最後にプログラムをコンパイルしたユーザーの認可識別子（ユーザー名）から取られます。RDB\$SCHEMA は、デフォルトのスキーマではありません。プリコンパイル・プログラムをコンパイルしているユーザーの認可識別子は、デフォルトのスキーマです。

コンテキスト・ファイルで、デフォルトのカatalogおよびスキーマを指定することもできます。

DECLARE MODULE 文で SCHEMA 句および CATALOG 句を指定しない場合、デフォルトのカatalogとスキーマのみを使用しているのでなければ、プリコンパイル・プログラムのスキーマ・オブジェクト名をそれらのカatalog、スキーマおよびオプションのエイリアス名で修飾する必要があります。

20.3.2 プリコンパイル・プログラムにおけるマルチスキーマ命名の使用法

例 20-2 (20-8) は、プリコンパイル C プログラムでスキーマ・オブジェクトに名前を付ける方法を示しています。

例 20-2 プリコンパイル C プログラムにおけるマルチスキーマ命名の使用法

```
/* ABSTRACT:
   This sample precompiled C program lists the employees assigned to
   each department defined in the sample multischema corporate_data
   database. */
#include <stdio.h>
main()
{
/* Declare return status variable for error handling. */
   int   SQLCODE;
/* Declare module to specify SQL standard dialect, including quoting rules. */
   EXEC SQL DECLARE MODULE SQL_MODULE
      DIALECT SQL92; (1) (20-9)
/* Variables for program use. */
   char  emp_id[6],
         emp_last_name[21],
         emp_first_name[21],
         dept_name[21],
         dept_code[5],
         dept_man_id[6];
/* Declare the corporate_data database. */
   EXEC SQL DECLARE ALIAS FILENAME corporate_data; (2) (20-9)
/* Declare the cursor for the DEPARTMENTS table. */
   EXEC SQL DECLARE DEPT_CURSOR TABLE CURSOR
      FOR SELECT
         DEPARTMENT_CODE, DEPARTMENT_NAME, MANAGER_ID FROM
```

```

        "RDB$DBHANDLE.ADMINISTRATION".ACCOUNTING.DEPARTMENTS (3) (20-9)
ORDER BY DEPARTMENT_CODE;
/* Declare the cursor for the EMPLOYEES table. */
EXEC SQL DECLARE EMP_CURSOR TABLE CURSOR
FOR SELECT
    E.EMPLOYEE_ID, E.LAST_NAME,E.FIRST_NAME FROM
    "RDB$DBHANDLE.ADMINISTRATION".PERSONNEL.EMPLOYEES E,
    "RDB$DBHANDLE.ADMINISTRATION".PERSONNEL.JOB_HISTORY JH
WHERE E.EMPLOYEE_ID = JH.EMPLOYEE_ID
    AND JH.DEPARTMENT_CODE = :dept_code
    AND JH.JOB_END IS NULL
ORDER BY E.LAST_NAME;
/* Print the report title. */
printf("%20s DEPARTMENT EMPLOYEE LISTING\n\n", " ");
/* Open the DEPARTMENTS cursor. */
EXEC SQL OPEN DEPT_CURSOR;
while (1)
{
    /* Fetch the department data. */
    EXEC SQL FETCH DEPT_CURSOR INTO
        :dept_code,
        :dept_name,
        :dept_man_id;
    .
    .
    .
}

```

次の説明は、例 20-2 (20-8) で番号を付けた項目に対応しています。

(1) DECLARE MODULE 文

DIALECT 句は、中でも ANSI/ISO 引用規則を指定します。このプログラムは、デリミタ付き識別子を使用しているため、ANSI/ISO SQL (SQL92) 引用規則を使用することを指定する必要があります。指定しない場合、SQL は二重引用符内の文字列を文字列リテラルとして正しく解釈しないので、文字列リテラルに対する二重引用符 (") の使用は、推奨されない機能であることを示すエラーを表示します。dialect または引用規則のデフォルトは、SQLV40 であり、引用文字を文字列リテラルとして解釈します。

(2) エイリアスの宣言

接続したデータベースがマルチスキーマ・データベースとして作成されている場合、MULTISCHEMA IS ON 句を指定する必要はありません。シングルスキーマ・モードでマルチスキーマ・データベースを表示する場合、DECLARE ALIAS 文で MULTISCHEMA IS OFF 句を使用します。マルチスキーマ・モードが無効であれば、外部名を使用して SQL 文内でスキーマ・オブジェクトを参照する必要があります。

(3) マルチスキーマ・オブジェクトの命名

プリコンパイル・プログラム内で SCHEMA 句と CATALOG 句を指定していなければ、

スキーマ・オブジェクトをカタログとスキーマ名で修飾します。エイリアス名は、オプションです。名前の

"RDB\$HANDLE.ADMINISTRATION".ACCOUNTING.DEPARTMENTS の中で RDB\$HANDLE はエイリアスであり、ADMINISTRATION はカタログ名、ACCOUNTING は、DEPARTMENTS オブジェクトがあるスキーマの名前です。3 レベルの命名の制限を超えないため、エイリアスとカタログ名は、いっしょに二重引用符 (") で囲んで、1 レベルの名前として表されます。

この状況で引用符を使用するには、DECLARE MODULE 文で DIALECT SQL92 句または QUOTING RULES SQL92 句を使用する必要があります。

例のプログラムでは、1 つのみが接続されているので、デフォルトのエイリアス RDB\$DBHANDLE を使用してオブジェクト名を修飾する必要はありません。

マルチスキーマ名を使用していなければ、希望どおりにプログラムをリンクできます。

SQL 国際言語化オプションの使用方法

Oracle Rdb には、英語以外のデータを格納するデータベースや、第一言語が英語以外のユーザー向けに、便利な機能がいくつも搭載されています。

この付録では、次の内容について説明します。

- 入力と表示において、データの書式を制御する文
- ソートと比較を制御する照合順番の仕様
- 個々の照合順番の動作
- すべての Oracle Rdb キャラクタ・セットで使用する照合順番

A.1 入力および表示形式の制御

次の内容については、入力形式（ほとんどの場合、表示形式も対象）を変更できます。

- 基数文字
SET RADIX POINT と SHOW RADIX POINT 文を使用します。
- 桁区切り文字
SET DIGIT SEPARATOR と SHOW DIGIT SEPARATOR 文を使用します。
- 通貨文字
SET CURRENCY SIGN と SHOW CURRENCY SIGN 文を使用します。
- DATE VMS データ型の日付と時間の書式
OpenVMS では、SET DATE FORMAT と SHOW DATE FORMAT 文を使用します。◆
Compaq Tru64 UNIX では、日付と時間の書式は変更できません。◆
- 日の名前や月の名前など、入力や表示に使用する言語
OpenVMS では、SET LANGUAGE と SHOW LANGUAGE 文を使用します。◆
Compaq Tru64 UNIX では、ロケール設定を使用します。詳細は、A.1.1 項 (A-2) を参照してください。◆

上記の SET および SHOW 文の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

A.1.1 Compaq Tru64 UNIX でのロケール設定の使用方法

Compaq Tru64 UNIX

ロケールとは、プログラムの国際環境を指します。ロケールは、実行時のプログラムの地域ごとの動作を定義します。ロケールは、1 つ以上のロケール・データベースから構築できます。

Oracle Rdb は、POSIX 規格に準拠したロケール・サポートを実装しています。言語フィールドと地域フィールドを使用して名前を付けたサブディレクトリに、メッセージ・ファイルが存在します。次の例は、言語が日本語（ja）で地域が日本（JP）のメッセージ・サブディレクトリを示しています。

```
/usr/lib/dbs/sql/vmn/lib/ja_JP/
```

言語フィールド名は POSIX iso639lang フィールド、地域フィールド名は POSIX iso3166terr フィールドに基づいています。

デフォルト・メッセージのサブディレクトリの名前は C_C です。次の例では、ロケールが指定されていない状態で、SQL メッセージ・ファイルのファイルを指定しています。

```
/usr/lib/dbs/sql/vmn/lib/C_C/sqlmsg.mdf
```

Compaq Tru64 UNIX でのロケールの設定は、システム管理者に問い合せてください。◆

A.2 照合順番の指定

Oracle Rdb は、あらゆるソートおよびブール操作において、デフォルトで ASCII 照合順番を使用します。ただし、このデフォルトは、次のような指定により、上書きできます。

- OpenVMS National character set (NCS) ユーティリティが提供する言語固有の照合順番
- NSC を使ったユーザー定義の照合順番

照合順番は、データベース全体または特定のドメインで指定できますが、テーブルの列では指定できません（ただし、ドメインを使用する行を定義すると、その行は、ドメインに指定した照合順番を継承します）。照合順番は、行をソート・キーとして使用する場合、行を格納する順序を決定します。また、照合順番によって、2つの行やリテラル値を含む行を比較するときのブール操作の動作も決まります。

照合順番に関する SQL 文の詳細は、『Oracle Rdb7 SQL Reference Manual』を参照してください。

A.3 照合順番の使用法

述語は、真、偽、不明と SQL が評価する条件を指定します。次に、CONTAINING、STARTING WITH、LIKE などの述語による照合順番の動作を示します。

- CONTAINING
CONTAINING 述語は、複数国のキャラクタ・セットで使用する発音区別符号を区別しません。したがって、a は、A、á、à、ä、Á、À、Â などと一致します（ノルウェー語では、ä は ae として処理されます）。
スペイン語では、ch と ll は、1 文字として扱います。したがって、CONTAINING 'C' では、C、c、ç、Ç が検出されますが、CH、ch、Ch、cH は検出されません。
- STARTING WITH
STARTING WITH 演算子は、大文字と小文字を区別するので、大文字の検索では小文字は除外され、小文字の検索では大文字は除外されます。たとえば、STARTING WITH 'C' と STARTING WITH 'c' では、異なる行が検索されます。
スペイン語では、ch と ll は、1 文字として扱われます。たとえば、照合順番 SPANISH でドメインを定義すると、STARTING WITH 'c' では、ワード char は検索されませんが、ワード cat は検索されます。
- LIKE
LIKE 演算子の動作は、STARTING WITH 演算子と同じです。LIKE 演算子は、大文字と小文字を区別するので、大文字の検索では小文字は除外され、小文字の検索では大文字は除外されます。たとえば、LIKE Ç と、LIKE ç では、異なるレコードが検索されます。
スペイン語では、LIKE 演算子は、ch と ll を 1 文字として認識します。たとえば、照合

順番 SPANISH でドメインを定義すると、LIKE c ではワード char は検索されませんが、ワード cat は検索されます。

次に、すべての述語に適用される複数国用のキャラクタ・セットの特徴を示します。

- スペイン語の慣例に基づいて、文字 ñ は必ず n と区別します。同様に、フランス語の慣例に基づいて、文字 ç は c と同様に扱います。
- 文字 ü は、多くの言語で文字 u と同じものとして扱いますが、デンマーク語、ノルウェー語、フィンランド語では、ソート順が x と z の間 (y と同じ) になります。

述語の制限については、『Oracle Rdb7 SQL Reference Manual』の言語と構文の項を参照してください。

A.4 Oracle Rdb キャラクタ・セットでの照合順番

標準の照合順番は、文字はオクテットごとに比較されます。Oracle Rdb のキャラクタ・セットでは、標準の照合順番のみを使用できます。

注意：行にマルチオクテット・キャラクタ・セットが使用されている場合、ユーザー定義の照合順番は使用できません。たとえば、データベースのデフォルト・キャラクタ・セットとして DEC_KANJI を定義し、すべてのデータ値が ASCII 文字である場合、DEC Multinational Character Set (MCS) の照合順番を使用できます。ただし、データに漢字が使用されている場合、MCS 照合順番を使用すると、照合結果はランダムのように見えるので非効率的です。

表 A-1 (A-4) は、Oracle Rdb がサポートするキャラクタ・セットで利用できる照合順番です。

表 A-1 Oracle Rdb キャラクタ・セットでの照合順番

キャラクタ・セットのタイプ	照合順番の定義
シングル・オクテット・キャラクタ・セット ^{*1}	照合順番は、ユーザー定義またはデフォルトの MCS 照合順番
固定のマルチオクテット・キャラクタ・セット ^{*1}	照合順番は、KANJI, HANZI, HANYU, SICGCC, KOREAN などのキャラクタ・セットで指定される数値に基づきます。

表 A-1 Oracle Rdb キャラクタ・セットでの照合順番（続き）

キャラクタ・セットのタイプ	照合順番の定義
複合のマルチオクテット・キャラクタ・セット ^{*1}	キャラクタ・セットの照合順番は、次のようになります（最小値から）。 1. ASCII 文字 ASCII キャラクタ・セットの照合順番は、MCS 照合順番テーブルで定義されている数値を基準にします。 2. ユーザー定義の文字 ユーザー定義の文字の照合順番は、各オクテットの数値か、ユーザー定義の照合順番テーブルで指定された値を基準にします。 3. DEC_HANZI、DEC_KOREAN、DEC_HANYU、DEC_SICGCC キャラクタ・セット キャラクタ・セットの照合順番は、各オクテットの数値に基づきます。

^{*1} シングルオクテットおよびマルチオクテット（固定と複合）キャラクタ・セットについては、『Oracle Rdb7 SQL Reference Manual』の言語と構文の章を参照してください。

DEC_KANJI キャラクタ・セットには、半角（ナロー）カタカナ・キャラクタ・セットも含まれるため、他の複合マルチオクテット・キャラクタ・セットとは異なります。このキャラクタ・セットの照合順番は、次のようになります（最小値から）。

1. ASCII キャラクタ・セット
ASCII キャラクタ・セットの照合順番は、MCS 照合順番テーブルで定義されている数値を基準にします。
2. カタカナ・キャラクタ・セット
カタカナ・キャラクタ・セットの照合順番は、JIS X0201 の数値を基準にします。
3. ユーザー定義の文字
ユーザー定義の文字の照合順番は、各オクテットの数値か、ユーザー定義の照合順番テーブルで指定された値を基準にします。
4. 漢字キャラクタ・セット
KANJI キャラクタ・セットの照合順番は、JIS X0208 の数値を基準にします。

記号

- ! (感嘆符)
 - コメントを参照
- " (引用符)
 - 疑問符 (?) を参照
- \$SQL_VARCHAR データ型, 6-17, 8-28
- \$ (ドル記号)
 - ドル記号 (\$) を参照
- : (コロン)
 - コロン (:) を参照
- ; (セミコロン)
 - セミコロン (;) を参照
- = (等号)
 - 等号 (=) を参照
- ? (疑問符)
 - パラメータ・マーカーを参照
- @ (アット記号)
 - 実行文 (@) を参照
- _ (アンダースコア)
 - アンダースコア () を参照

数字

- 2 フェーズ・コミット・プロトコル, 16-24, 16-25

A

- Ada 言語
 - SQLDA, 11-9
 - SQL プリコンパイラ, 6-14
 - 出力ファイル, 6-7
 - 使用したファイル, 6-15
 - 入力ファイル, 6-7
 - SQL モジュールでのプロシージャのコール, 4-13

- SQL モジュール内の言語識別子, 3-8
- 外部ルーチンのガイドライン, 14-38
- 実行可能イメージの作成, 7-4
- 宣言
 - シンボリック・エラー・コード, 10-16
 - パラメータ, 8-27
- ドル記号のかわりにアンダースコアを使用, 8-27
- RDB\$ 名, 10-18
- パッケージ, 8-27
- パラメータを使用, 8-27
- プリコンパイル
 - 使用したファイル, 6-15
- プリコンパイル・プログラム, 6-14
 - SQL 文の終了, 6-6
 - ファイル名の長さ, 6-14
 - プログラムのデバッグ, 7-13
 - ライブラリ, 6-15
- プログラムを参照
- Ada プログラム・ライブラリ・マネージャ (ACS)
 - Ada または LINK コマンド (ACS) を参照
- Ada プログラム内のファイル名の長さ, 6-14
- Ada 言語
 - SQL プリコンパイラ
 - プリコンパイル SQL
- ALIAS 句, 20-3
 - SQL モジュール, 3-10
- ALIGN_RECORDS 修飾子, 8-28
- ansi SQL プリコンパイラ・コマンドライン修飾子, 6-8
- ANSI_FORMAT SQL プリコンパイラ・コマンドライン修飾子, 6-18
- ANSI/ISO SQL 規格, 1-1
 - COBOL プログラムのフォーマット, 6-18
 - SQL モジュール言語, 1-4, 3-14
 - 非標準 SQL 構文の検出, 20-6

- マルチスキーマ・データベース用, 20-1

AST

- 非同期システム・トラップ (AST) を参照

ATOMIC キーワード, 12-15

ATTACH 文, 15-1, 15-2, 15-3

- FILENAME オプション, 15-1
- リモート・データベース, 15-4

PATHNAME オプション, 15-2

USER 句, 15-5, 15-6

エイリアス, 15-13

リモート・アクセス, 15-9

ロック競合エラー, 10-33

AUTHORIZATION 句, 20-4, 20-5

- SQL モジュール, 3-10
- ストアド・ルーチン, 13-3, 13-4

AVG 関数

- NULL 結果, 8-18

B

BEGIN キーワード

- ストアド・ファンクション, 13-8
- ストアド・プロシージャ, 13-7
- 複合文, 12-3

BIND ON 句

- CREATE ルーチン文, 14-7, 14-32

BIND SCOPE 句

- CREATE ルーチン文, 14-7
- 通知ルーチン, 14-42

BY DESCRIPTOR 句

- SQL モジュール言語, 4-13

C

CALL 文, 12-4, 12-14

- 外部プロシージャを起動, 14-2, 14-32
- ストアド・プロシージャを起動, 13-10, 13-11
- 引数の受渡し, 13-11

CASCADE キーワード

- DROP FUNCTION 文, 13-13, 14-8
- DROP PROCEDURE 文, 13-13, 14-8

CASE 式, 13-7

CASE 文, 12-3, 12-8

CAST 関数, 9-4, 9-6, 9-8, 9-12, 9-13, 9-14, 9-15, 9-17

- パラメータ・マーカー, 9-17

CATALOG 句, 20-3, 20-5

SQL モジュール, 3-9

CDD/Plus

- リポジトリを参照

CHAR データ型, 8-26

- SQL モジュール言語, 4-17
- 動的 SQL, 4-5

CHECK オプション

- プロシージャ・パラメータ, 4-14

CLOSE 文, 18-4, 18-8

- 動的 SQL でのパラメータの使用, 11-35
- パラメータの使用, 11-35
- ロック, 16-29, 18-11

-cm 修飾子

- SQL プリコンパイラ・コマンドライン, 16-40
- SQL モジュール・プロセッサ・コマンドライン, 16-40

COBOL 言語

- COPY 文の制限, 8-8
- EXECUTE IMMEDIATE 文, 11-6
- SQLCODE, 6-18
- SQL プリコンパイラ
 - 出力ファイル, 6-7
 - 入力ファイル, 6-7
- SQL プリコンパイラ、プログラムを参照
- SQL モジュール内の言語識別子, 3-8
- 外部ルーチンのガイドライン, 14-40

宣言

- シンボリック・エラー・コード, 10-16
- パラメータ, 8-30

パラメータを使用, 8-30

プリコンパイル SQL, 6-18

- ANSI/ISO フォーマットの使用, 6-18
- SQL 文の終了, 6-6
- 制限, 6-18
- 端末フォーマットの使用, 6-18
- パラメータ名の処理, 6-5
- 複数行リテラルの継続, 6-18
- プリコンパイル SQL、プログラムを参照
- プリコンパイル・プログラム, 6-18
- プログラムのデバッグ, 7-13

COBOL に含まれるプログラム, 6-18

COBOL の END PROGRAM 文, 6-18

COMMIT 文, 16-42, 16-44

- Recovery-unit ジャーナル・ファイルへの影響,
16-43
- SQL モジュール
 - プロシージャ・パラメータ, 4-7
- カーソル, 18-8, 18-11, 18-12
- ストアド・ファンクション, 13-9
- 制約違反, 16-41
- トランザクションが開始しなかった場合のエラー,
10-35
- 複合文, 12-17
- Compaq Tru64 C コンパイラ
 - リンク, 7-3
- COMPILETIME オプション, 15-3
- COMPUTED BY 句
 - 外部関数, 14-30
- CONNECT 修飾子
 - SQL プリコンパイラ・コマンドライン, 17-11
 - SQL モジュール・プロセッサ・コマンドライン,
17-11
- CONNECT 文, 15-3, 17-5
 - 接続の作成, 17-5
 - デフォルトのデータベース環境の複製, 17-6
- conn 修飾子
 - SQL プリコンパイラ・コマンドライン, 17-11
 - SQL モジュール・プロセッサ・コマンドライン,
17-11
- CONSTRAINT 修飾子
 - SQL プリコンパイラ・コマンドライン, 16-40
 - SQL モジュール・プロセッサ・コマンドライン,
16-40
- COUNT 関数
 - インジケータ・パラメータ, 8-18
- CREATE DATABASE 文
 - SQL モジュール, 4-16
 - プリコンパイル・プログラム, 6-5
- CREATE FUNCTION 文, 14-4, 14-9, 14-13, 14-24
- CREATE MODULE 文, 13-3, 13-4
 - ストアド・ルーチンの定義, 13-3
- CREATE PROCEDURE 文, 14-6
- CREATE ルーチン文の VARIANT 句, 14-5
- CURRENT_TIMESTAMP 関数, 19-9
 - 環境の設定, 9-3
- C 言語
 - Compaq Tru64C のリンク, 7-3
 - INCLUDE SQLCA 文, 8-10
 - INTEGER データ型, 8-29
 - NULL 終了文字列, 4-17, 6-16, 8-9, 8-28, 11-20

- リポジトリ, 8-12
- SMALLINT データ型, 8-29
- SQLDA, 11-9
- SQL プリコンパイラ、プログラムを参照
- SQL モジュール言語, 8-12
- SQL モジュールでマルチスキーマ名を使用, 20-4
- SQL モジュール内の言語識別子, 3-8
- VAX C 拡張, 6-17
- インクルード・ファイル, 6-17, 10-30
- 外部ルーチンのガイドライン, 14-39
- 宣言
 - SQLCA, 8-28
 - SQLCODE, 8-27, 8-28
 - シンボリック・エラー・コード, 10-16
 - パラメータ, 8-12, 8-27
- データの整理, 8-28
- バイナリ・データ, 4-17, 6-17, 8-28
- パラメータの使用, 8-27
- パラメータの宣言, 8-27
- パラメータのポインタ, 6-17
- プリコンパイル SQL, 6-16
 - SQL 文の終了, 6-6
 - 出力ファイル, 6-7
 - 入力ファイル, 6-7
 - プリコンパイル・プログラム, 6-16
 - プログラムのデバッグ, 7-13
- C 言語の NULL 終了文字列, 8-9, 8-12, 8-28, 11-20
- C の SMALLINT データ型, 8-29
- C の整数データ型, 8-29

D

- Danish 照合順番条件演算子, A-4
- .dbsrc 構成ファイル, 15-2, 15-5, 15-8
- DEC C コンパイラ
 - Compaq Tru64 C コンパイラを参照
- DECIMAL データ型, 8-26
- DECLARE ALIAS 文, 15-2, 15-3
 - COMPILETIME オプション, 15-3
- EXTERNAL キーワード, 7-7, 7-12
- GLOBAL キーワード, 7-7, 7-12
- RUNTIME オプション, 15-4
- SQL で使用, 17-4
- SQL プリコンパイラ, 20-9
- SQL モジュール, 3-11, 4-5, 15-12
 - パラメータを含む, 4-5, 4-6
- USER 句, 15-5, 15-6

- 共有可能イメージ, 7-7
- コンテキスト・ファイルに含まれる, 6-13
- 複数のモジュール, 7-12
- プリコンパイル SQL, 6-5, 11-7
- リモート・データベース用の FILENAME オプション, 15-4
- DECLARE CURSOR 文, 18-3
 - FOR UPDATE 句, 19-6, 19-9
 - SCROLL キーワード, 18-4, 18-15
 - SQL モジュール, 3-11, 4-7
 - プロシージャ・パラメータ, 4-7
 - WHERE CURRENT OF 句, 18-13, 19-4
 - WITH HOLD 句, 18-8, 18-11
 - 上書きできない, 18-11
 - エラー処理, 10-5
 - 拡張動的カーソル, 18-7
 - スクロール可能リスト・カーソル, 18-15
 - 静的カーソル, 18-7
 - 動的 DECLARE CURSOR 文, 拡張動的 DECLARE CURSOR 文を参照
 - 動的カーソル, 18-7
 - プリコンパイル・プログラム, 6-5
 - 保持可能カーソル, 18-11
 - 読取り専用カーソル, 19-6, 19-9
- 拡張動的 DECLARE CURSOR 文を参照
- DECLARE CURSOR 文
 - 動的 SQL
 - 動的 DECLARE CURSOR 文
- DECLARE MODULE 文, 6-11
 - DEFAULT DATE FORMAT 句, 9-3
 - マルチスキーマ命名, 20-7
- DECLARE STATEMENT 文
 - SQL モジュール, 3-12
 - 移植性, 6-13
- DECLARE TABLE 文, 6-11
 - SQL モジュール, 3-12, 4-16
 - 移植性, 6-13
 - パフォーマンスの改善に使用, 5-5
 - プリコンパイル・プログラム, 6-5
- DECLARE TRANSACTION 文, 16-5, 16-10, 16-22
- RESERVING 句, 16-10
 - ストアド・ルーチン, 13-17
- SET TRANSACTION 文との対比, 16-5
- SQL モジュール, 3-12
- 共有モード・オプション, 16-10
- コンテキスト・ファイルに含まれる, 6-13
- デフォルトの設定, 16-22

- 複合文, 12-18
- プリコンパイル・プログラム, 6-5
- 分離レベル, 16-15
- 読取り / 書込みの指定, 16-8
- 読取り専用の指定, 16-6
- ロック・タイプ・オプション, 16-10
- DECLARE 文
 - SQL モジュール内, 3-11
- DECLARE 変数句
 - 複合文, 12-4
- DECLARE 変数句の CONSTANT キーワード, 12-5
- DECnet ネットワーク・プロトコル, 15-9
- DEFAULT DATE FORMAT 句
 - DECLARE MODULE 文, 9-3
 - モジュール・ヘッダー, 9-3
- DEFAULT 句
 - DECLARE 変数句, 12-5
- DEFERRABLE 制約, 16-40
- DELETE 文, 19-8
 - SQL モジュール, 4-8
 - プロシージャ・パラメータ, 4-8
 - WHERE 句, 19-8
 - カーソル, 19-8
 - 処理する行の数, 8-10
 - 制限, 15-14
 - 複数行に影響を与える, 18-5
- DESCRIBE 文, 11-4
 - MARKERS 句, 11-8, 11-23
 - SELECT LIST 句, 11-8, 11-19
- 動的 SQL
 - パラメータの使用, 11-33
 - パラメータの使用, 11-33, 11-35
 - 複合文, 12-18
- DESCRIBE 文の MARKERS 句, 11-8, 11-23
- DESCRIBE 文の SELECT LIST 句, 11-8, 11-19
- DIALECT 句
 - DECLARE MODULE 文, 20-9
- DISCONNECT 文, 15-15, 17-5, 17-10
 - カーソル, 18-4, 18-11
 - データベースの接続解除, 15-15
- DISTRIBTRAN 権限, 15-11
- DROP FUNCTION 文, 13-12, 14-8
- DROP MODULE 文
 - ストアド・ファンクションの削除, 13-12
 - ストアド・プロシージャの削除, 13-12
 - ストアド・モジュールの削除, 13-12
- DROP PROCEDURE 文, 13-12, 14-8

DROP TABLE 文
構造化テーブルの場合, 19-4

E

END MODULE 句, 13-5
END-EXEC フラグ, 6-5, 6-6
END キーワード
 ストアド・ファンクション, 13-8
 ストアド・プロシージャ, 13-7
 複合文, 12-3
EXCLUSIVE 共有モード, 16-27, 16-28
 読取り専用トランザクション, 16-30
 読取り専用トランザクションでの競合, 16-30
EXEC SQL フラグ, 6-4
EXECUTE IMMEDIATE 文, 11-4
 複合文, 12-18
EXECUTE 文, 11-4
 動的 SQL
 パラメータの使用, 11-32, 11-33
 パラメータの使用, 11-32, 11-33
 複合文, 12-18
Extend_source 修飾子, SQL プリコンパイラ・コマンド
 ライン, 6-20
EXTERNAL NAME 句
 外部関数の定義, 14-5
 外部プロシージャ定義, 14-7
EXTERNAL_GLOBAL コマンドライン修飾子
 SQL プリコンパイラ, 7-7, 7-9
 SQL モジュール・プロセッサ, 7-7, 7-9
EXTERNAL キーワード
 DECLARE ALIAS 文, 7-7, 7-12
-extern オプション, 7-11, 7-12
EXTRACT 関数, 9-8, 9-12

F

FETCH 文, 18-3
 INTO 句, 18-3
 SQL モジュール, 4-8
 プロシージャ・パラメータ, 4-8
 エラー処理, 10-35
 オプション, 18-15
 カーソル, 18-15
 カーソル内の行の序数位置, 8-10
 行検索の順序, 18-3, 18-4
 行ポインタ, 18-3

動的 SQL, 11-4
 パラメータの使用, 11-35
 パラメータの使用, 11-35
 リスト・カーソル, 19-5
 ループ, 18-9
FETCH 文の SCROLL キーワード, 18-15
FILENAME 句
 ATTACH 文
 エイリアスの指定, 15-12
 ノード仕様を含む, 15-4
 DECLARE ALIAS 文
 データベース定義の検索, 15-3
Finnish 照合順番条件演算子, A-4
FLAG_NONSTANDARD 修飾子, 3-14
 非標準 ANSI/ISO SQL 構文の表示, 20-6
FOR UPDATE OF 句, 19-6
FOR UPDATE 句
 行を削除する場合, 19-9
-form ansi SQL プリコンパイラ・コマンドライン修飾
 子, 6-8, 6-18
FORTRAN 言語
 END 文, 6-21
 SQLCODE, 6-21
 SQL プリコンパイラ
 出力ファイル, 6-7
 入力ファイル, 6-7
 SQL プリコンパイラ; プログラムも参照
 SQL モジュール内の言語識別子, 3-8
 外部ルーチン・ガイドライン, 14-37, 14-40
 シンボリック・エラー・コードの宣言, 10-16
 パラメータの使用, 8-30
 パラメータの宣言, 8-30
 プリコンパイル・プログラム, 6-20
 DO ループの制限, 6-21
 IF 文の制限, 6-21
 SQLCA フィールド名, 6-20
 SQL 文の終了, 6-6
 行あたりの文字数, 6-20
 継続行の行数, 6-20
 ブロック構造, 6-21
 プログラムのデバッグ, 7-13
 プログラムも参照
FORTRAN の DO 文, 6-21
FORTRAN の END 文, 6-21
FORTRAN のラベル付き文, 6-21
FORTRAN 言語
 プリコンパイル SQL

FOR 文, 12-3, 12-10, 13-12

 現在行, 12-21

 終了, 12-13

 ラベル, 12-13

French 照合順番条件演算子, A-4

FROM パス名句

 SQL モジュール言語, 8-3, 8-11

G

G_FLOAT 修飾子

 プリコンパイル・プログラムへの指定, 6-8

GENERAL キーワード

 CREATE FUNCTION 文, 14-5

 CREATE PROCEDURE ルーチン文, 14-7

 LANGUAGE 句, 14-38

GENERAL 言語識別子

 SQL モジュール, 3-8

 解釈, 4-12

GET DIAGNOSTICS 文, 10-3, 12-4, 12-21, 12-22, 12-23

GLOBAL キーワード

 DECLARE ALIAS 文, 7-7, 7-12

H

HAVING 句

 SQL モジュール

 プロシージャ・パラメータ, 4-7

 インジケータ・パラメータを含めない, 8-24

I

IF 文, 12-4, 12-6, 12-17, 13-12

 ネスト, 12-7

 FOR 文, 12-10

 プリコンパイルされた FORTRAN, 6-21

IF 文の ELSEIF 句, 12-7

IF 文の THEN 句, 12-7

INCLUDE SQLCA 文, 10-7

INCLUDE 文, 8-3, 8-8

 FROM DICTIONARY 句, 8-8, 8-9

 Pascal, 6-22

 SQLCA キーワード, 8-4, 8-8

 EXTERNAL キーワード, 8-10

 定義のソース, 8-10

 SQLCA キーワード 1, 10-7

SQLDA2 キーワード, 8-5, 8-8

 定義のソース, 8-11

SQLDA キーワード, 8-5, 8-8

 定義のソース, 8-11

SQL モジュール, 4-17

 制限, 8-8

 ファイル仕様オプション, 8-11

 プリコンパイル・プログラム, 6-4

INOUT パラメータ・モード, 13-6

INSERT 文, 19-2

 SELECT 句, 19-3

 SQL モジュール, 4-8

 プロシージャ・パラメータ, 4-8

 VALUES 句, 19-2

 インジケータ・パラメータ, 8-22

 外部関数, 14-30

 処理する行の数, 8-10

 データの更新, 19-3

 データのコピー, 19-4

 データの保管, 19-3

 データベースのロード, 19-2

 テーブルの組合せ, 19-3

 テーブルを構造化する場合, 19-4

 パフォーマンスの改善, 19-4

 日付時刻データ型, 9-2, 9-3

 プログラムで NULL を指定, 19-3

 プログラムで値を指定, 19-3

INTERVAL データ型, 9-2, 9-4, 9-12, 9-18

 SQL プリコンパイラ, 9-5

 精度, 9-2

INTO 句, 18-3

 FETCH 文

 ループ内, 18-9

 SELECT 文, 18-6

IN パラメータ・モード, 13-6, 13-8

ISO 規格

 ANSI/ISO SQL 規格を参照

L

LANGUAGE 句

 CREATE FUNCTION 文, 14-5

 CREATE PROCEDURE ルーチン文, 14-7

 SQL モジュール, 3-8, 4-11, 4-12

 C の使用, 4-17

 GENERAL の使用, 3-8, 4-17

- ストアド・ルーチン, 13-4
- lc_proc 修飾子
 - SQL モジュール・コマンドライン, 4-4
- lcosi ライブラリ, 5-3, 7-11
- LEAVE 文, 12-4, 12-13
- LINK コマンド (ACS)
 - Ada での使用, 7-4
 - 実行可能イメージの作成, 7-4, 7-5
- LINK コマンド (DCL)
 - 実行可能イメージの作成, 7-2
- LINK コマンドの SQL\$USER ライブラリ・ファイル, 7-2
- LIST OF BYTE VARYING データ型, 18-6
- list SQL プリコンパイラ・コマンドライン修飾子, 6-8
- LNK\$LIBRARY 論理名, 7-2
- LONG_SQLCODE コマンドライン修飾子, 8-28
- LOOP 文, 12-4, 12-9
 - 終了, 12-13
 - ラベル, 12-13
- lots ライブラリ, 5-3, 7-11
- lrdbshr ライブラリ, 5-3, 7-11
- lrdbsql ライブラリ, 5-3, 7-11
- lsqlcode コマンドライン修飾子, 8-28
- l SQL プリコンパイラ・コマンドライン修飾子, 6-6

M

- Make ファイル
 - sqllibs.make ファイル, 7-11
- match SQL プリコンパイラ・コマンドライン修飾子, 6-8
- MAX 関数
 - NULL 結果, 8-18
- MIN 関数
 - NULL 結果, 8-18
- MODULE 句
 - SQL モジュール, 3-6
- MULTISchema IS ON 句, 20-2
- Multivendor 統合アーキテクチャ (MIA)
 - 例外処理のサポート, 10-18

N

- NOCONNECT 修飾子
 - SQL プリコンパイラ・コマンドライン, 17-11
 - SQL モジュール・プロセッサ・コマンドライン, 17-11

- NOEXTERNAL_GLOBAL コマンドライン修飾子
 - SQL プリコンパイラ, 7-7, 7-9
 - SQL モジュール・プロセッサ, 7-7, 7-9
- noextern オプション, 7-11, 7-12
- NOG_FLOAT コマンドライン修飾子, 6-17
- NOOPTIMIZE 修飾子, 7-13
- NOPARAMETER_CHECK 修飾子
 - SQL モジュール・プロセッサ・コマンドライン, 5-4
- Norwegian 照合順番条件演算子, A-3
- NOT ATOMIC キーワード, 12-15
- NOT DEFERRABLE 制約, 16-40
- Not found 条件, 18-9
 - 検出
 - SQLCODE の使用, 10-8
 - WHENEVER 文の使用, 10-11
- Not found 条件, 検出, 10-11
- NOTIFY 句
 - CREATE ルーチン文, 14-8, 14-41
- NOWAIT トランザクション・オプション, 16-13, 16-28
- NULL 値
 - インジケータ・パラメータ, 18-6
 - 値, 8-21
 - 行を挿入する場合
 - プログラム, 19-3
 - 識別, 8-18
 - ストアド・ファンクション, 13-8
 - ストアド・プロシージャ, 13-6, 13-11
 - ストアド・プロシージャのコール, 13-11
 - 設定, 8-17
 - 割当て, 8-18

O

- OBJECT 修飾子
 - プリコンパイル・プログラムへの指定, 6-8
- OpenVMS Linker ユーティリティ
 - LIBRARY 修飾子, 7-2
 - SHAREABLE 修飾子, 7-3
 - 共有可能なイメージの要求を決定, 7-5
 - プログラム・セクションの使用, 7-6
- OPEN 文, 18-3, 18-8
 - SQL モジュール, 4-7
 - プロシージャ・パラメータ, 4-7
- 動的 SQL, 11-4
 - パラメータの使用, 11-35

パラメータの使用, 11-35
プリコンパイル・プログラム, 6-5
ロッキング, 16-27

Oracle Rally

Rally を参照

OUT パラメータ・モード, 13-6

P

PARAMETER COLONS 句, 3-11, 8-12

PARAMETER STYLE 句

CREATE FUNCTION 文, 14-5

CREATE PROCEDURE ルーチン文, 14-7

PARAMETER_CHECK 修飾子

SQL モジュール・プロセッサ・コマンドライン,
5-4

Pascal 言語

SQL プリコンパイラ

出力ファイル, 6-7

入力ファイル, 6-7

SQL プリコンパイラ、プログラムを参照

SQL モジュール内の言語識別子, 3-8

外部ルーチン・ガイドライン, 14-37, 14-40

シンボリック・エラー・コードの宣言, 10-16

パラメータの宣言, 8-31

パラメータを使用, 8-31

プリコンパイル SQL、プログラムを参照

プリコンパイル・プログラム, 6-22

SQL 文の終了, 6-6

制限, 6-22

プログラムのデバッグ, 7-13

変数の宣言, 6-23

Pascal の文の順序

SQL プリコンパイル・プログラム, 6-23

PASSWORD_DEFAULT 修飾子, 15-7

-pass オプション

SQL プリコンパイラ・コマンドライン, 15-7

SQL モジュール・プロセッサ・コマンドライン,
15-7

PATHNAME 句

DECLARE ALIAS 文

データベース定義の検索, 15-3

PL/I 言語

SQLDA, 11-9

SQL プリコンパイラ

出力ファイル, 6-7

入力ファイル, 6-7

SQL プリコンパイラ、プログラムを参照

SQL モジュール内の言語識別子, 3-8

宣言

シンボリック・エラー・コード, 10-16

パラメータ, 8-31

パラメータの使用, 8-31

パラメータの宣言, 8-31

プリコンパイル SQL、プログラムを参照

プリコンパイル・プログラム, 6-23

SQL 文の終了, 6-6

プログラムのデバッグ, 7-13

PREPARE 文, 11-4, 18-17, 18-18

パラメータの使用, 11-32, 11-33, 11-35

複合文, 12-18

文識別子の初期化, 11-35

PRESERVE 句

DECLARE CURSOR 文, 18-12

PROTECTED 共有モード, 16-7, 16-27, 16-28

PSECT 属性, 7-3, 7-6

Q

QUOTING RULES 句

DECLARE MODULE 文, 20-9

R

RDB\$_DEADLOCK コード

RDB\$LU_STATUS フィールド, 10-34

RDB\$_INTEG_FAIL コード

RDB\$LU_STATUS フィールド用, 10-31

RDB\$_LOCK_CONFLICT コード

RDB\$LU_STATUS フィールド用, 10-34

RDB\$_NO_DUP コード

RDB\$LU_STATUS フィールド用, 10-31

RDB\$_NOT_VALID コード

RDB\$LU_STATUS フィールド用, 10-31

RDB\$CATALOG カタログ, 20-2

デフォルト

SQL プリコンパイラ用, 20-7

SQL モジュール・プロセッサ用, 20-3

内容, 20-7

RDB\$CLIENT_DEFAULTS.DAT 構成ファイル, 15-5,
15-8

RDB\$DBHANDLE エイリアス, 15-12, 15-13

AUTHORIZATION 句内, 3-11

- SQL プリコンパイラのデフォルト・エイリアス, 20-7
- RDB\$DBHANDLE デフォルト・エイリアス, 3-11
- RDB\$INTERRELATIONS システム・テーブル
 - 依存関係の追跡, 13-14
- RDB\$LU_STATUS フィールド, 10-13, 10-15
 - シンボリック・エラー・コードを宣言, 10-16
 - シンボリック・コード
 - "valid if" 違反, 10-31
 - 一意なインデックスの重複値, 10-31
 - 制約違反, 10-31
 - デッドロック, 10-34
 - ロックの競合, 10-34
- RDB\$MESSAGE_VECTOR 配列, 10-3, 10-5, 10-13, 10-14
 - 宣言, 10-15
 - メッセージの表示に sql_signal を使用, 10-27
- RDB\$REMOTE アカウント, 15-9
- RDB\$ROUTINES イメージ名, 14-5, 14-7
- RDB\$SCHEMA スキーマ, 20-2
 - 場所, 20-8
- RDB_DEBUG_FLAGS_OUTPUT 構成パラメータ, 12-19
- RDB_DEBUG_FLAGS 構成パラメータ, 12-18
- RDB_RTX_SHRMEM_PAGE_CNT 構成パラメータ, 14-34, 14-44
- RDB_VALIDATE_ROUTINE 構成パラメータ, 13-21
- RDBSERVER 論理名, 15-9
- RDMS\$DEBUG_FLAGS_OUTPUT 論理名, 12-19
- RDMS\$DEBUG_FLAGS 論理名, 12-18
- RDMS\$RTX_SHRMEM_PAGE_CNT 論理名, 14-34, 14-44
- RDMS\$VALIDATE_ROUTINE 論理名, 13-21
- RDMS\$VERSION_VARIANT 論理名, 15-9
- READ COMMITTED 分離レベル, 16-14, 16-17, 16-19
 - 使用, 16-20
 - 利点, 16-19
- READ ロック型, 16-28
- Recovery-unit ジャーナル (.ruj) ファイル, 16-43
 - 行が書き込まれた場合, 16-27
- RELEASE 文, 11-4
- REPEATABLE READ 分離レベル, 16-14, 16-15, 16-17
 - 使用, 16-20
 - 利点, 16-19
- Reserve モード
 - アクセスの競合, 16-28

- RESERVING 句, 16-3
 - DECLARE TRANSACTION 文, 16-10
 - EXCLUSIVE 共有モード, 16-12
 - SET TRANSACTION 文, 16-10
 - 分離レベル, 16-15
 - 読取り / 書込みトランザクション, 16-12
 - 読取り専用トランザクション, 16-11
- RESTRICT キーワード
 - DROP FUNCTION 文, 13-13, 14-8
 - DROP PROCEDURE 文, 13-13, 14-8
- RETURNS 句, 13-8
 - CREATE FUNCTION 文, 14-5
- RETURN 文, 13-7, 13-9
- RIGHTS 句
 - AUTHORIZATION 句, 3-10
 - DECLARE MODULE 文, 20-7
 - 権限の確認, 20-4
- ROLLBACK 文, 16-42, 16-44
 - Recovery-unit ジャーナル・ファイルへの影響, 16-43
- SQL モジュール
 - プロシージャ・パラメータ, 4-7
- エラー
 - 監視, 10-12
 - トランザクションが開始されなかった場合, 10-35
 - カーソル, 18-8, 18-11, 18-12
 - ストアド・ファンクション, 13-9
 - 複合文, 12-17
- ROWID キーワード, 19-8
- .ruj ファイル
 - Recovery-unit ジャーナル (.ruj) ファイルを参照

S

- SCHEMA 句, 20-3, 20-5
 - SQL モジュール, 3-9
- SELECT 文
 - INTO 句, 18-6
 - カーソルのかわり, 18-6
 - SQL モジュール
 - プロシージャ・パラメータ, 4-7
 - 処理する行の数, 8-10
 - 制限, 15-14
 - 動的 SQL, 11-3, 11-19, 11-22, 11-32, 11-35
 - パラメータ・マーカーの使用, 11-3, 11-22
- SELECT 文でない

- 動的 SQL, 11-2, 11-32, 11-33
 - 実行, 11-18
 - パラメータ・マーカー, 11-8
 - パラメータ・マーカーなし, 11-5
- SERIALIZABLE 分離レベル, 16-3, 16-14, 16-15
 - 利点, 16-19
- SET ALL CONSTRAINTS 文, 16-40
- SET CONNECT 文, 17-5, 17-9
 - 暗黙的, 17-5
- SET DEFAULT CHARACTER SET 文, 3-7
- SET DEFAULT CONSTRAINT MODE 文, 16-40
- SET DIALECT 文, 3-7
- SET FLAGS 文
 - NOPREFIX キーワード, 12-20
 - TRACE キーワード, 12-18
- SET HOLD CURSOR 文, 18-13
- SET IDENTIFIER CHARACTER SET 文, 3-7
- SET LITERAL CHARACTER SET 文, 3-7
- SET NATIONAL CHARACTER SET 文, 3-7
- SET NOEXECUTE 文, 13-22
- SET TRANSACTION 文, 16-21
 - DECLARE TRANSACTION, 16-22
 - DECLARE TRANSACTION 文との対比, 16-6
 - RESERVING 句, 16-10
 - ストアド・ルーチン, 13-17
- SQL モジュール, 3-12
- WAIT または NOWAIT オプション, 16-12
- 共有モード・オプション, 16-10
- スコープ, 16-22
- ストアド・ファンクション, 13-9
- 制約, 16-12, 16-27
- 接続, 17-10
- トリガー, 16-12, 16-27
- 複合文, 12-17
- プリコンパイル・プログラム, 6-5
- 分離レベル, 16-15
- 読取り / 書込みの指定, 16-8
- 読取り専用の指定, 16-6
- 利点, 16-6
- ロック・タイプ・オプション, 16-10
- SET 句, 19-6
- SET 代入文
 - 複合文, 12-4
- SHARED 共有モード, 16-28
- SHOW FUNCTION 文, 13-20
- SHOW PROCEDURE 文, 13-20

- SIGNAL 文, 10-3, 12-4, 12-22
- Spanish 照合順番条件演算子, A-3
- SQL
 - DECLARE TRANSACTION 文, 16-10
 - SET TRANSACTION, 16-5, 16-22
 - DISCONNECT 文, 17-5
 - 動的 DECLARE CURSOR 文, 11-4, 11-19, 18-7, 18-17
- SQL INCLUDE 文でサポートされていないライブラリ・ファイル, 8-8
- SQL\$DATABASE 論理名, 15-2
 - 複数のデータベースへ接続する場合, 15-13
- SQL\$GET_ERROR_TEXT ルーチン, 10-30
 - sql_get_error_text ルーチンを参照
- sql\$persmsg ファイル, 10-30
- SQL\$SAMPLE ディレクトリ
 - サンプル・プログラム, 1-5
- SQL\$SIGNAL ルーチン
 - sql_signal ルーチンを参照
- sql_all_datatypes サンプル・プログラム, 8-27
- SQL モジュール, 8-31
- SQL_ALTERNATE_SERVICE_NAME 構成パラメータ, 15-10
- sql_close_cursors ルーチン, 18-4
- SQL_DATABASE 構成パラメータ, 15-2
 - 複数のデータベースへ接続する場合, 15-13
- sql_deregister_error_handler ルーチン, 10-4, 10-19, 10-20
 - コール, 10-24
 - 宣言, 10-25
- sql_dynamic サンプル・プログラム, 11-4
- sql_get_error_handler ルーチン, 10-4, 10-19, 10-20
 - コール, 10-24
 - 宣言, 10-25
- sql_get_error_text ルーチン, 10-28
- sql_get_message_vector ルーチン, 10-3, 10-13, 10-14
- sql_load_jobhist サンプル・プログラム, 19-2
- sql_rdb_headers.h インクルード・ファイル, 6-17
- sql_register_error_handler ルーチン, 10-4, 10-18, 10-20
 - 共有可能イメージ, 10-23
 - コール, 10-24
 - 宣言, 10-25
- sql_report サンプル・プログラム, 18-6
- sql_signal ルーチン, 10-4, 10-27
- sql_sqlda.h ヘッダー・ファイル, 8-5, 11-10
- sql_terminate サンプル・プログラム, 19-6

- エラー処理技術, 10-4
- ユーザー指定のメッセージ・ファイルの使用, 10-30
- SQLCA ステータス・パラメータ, 8-4, 10-5, 10-7
 - C での宣言, 8-28
 - FORTRAN での名前, 6-20
 - SQL モジュール, 4-4
 - ストアド・ファンクション, 13-8
 - ストアド・プロシージャ, 13-6
 - プリコンパイル SQL, 6-5
 - FORTRAN での制限, 6-21
- SQLCA データ構造, 10-7
- SQLCODE_DEADLOCK コード, 10-34
- SQLCODE_EOS コード, 10-8
- SQLCODE_LOCK_CONFLICT コード, 10-34
- SQLCODE_NOT_VALID コード, 10-31
- SQLCODE_SUCCESS コード, 10-8
- SQLCODE ステータス・パラメータ, 8-4, 10-3, 10-6, 10-7
 - C 言語, 8-28
- SQL モジュール, 4-4
- 値
 - "valid if" 違反, 10-31
 - 一意なインデックスの重複値, 10-31
 - 警告, 10-8
 - 成功, 10-8
 - 正常な実行, 10-8
 - 制約違反, 10-31
 - 致命的エラー, 10-8
 - データなし条件, 10-8
 - デッドロック, 10-34
 - ロックの競合, 10-34
 - ストアド・ファンクション, 13-8
 - ストアド・プロシージャ, 13-6
 - 宣言, 10-6, 10-7
 - C 言語, 8-27
 - FORTRAN, 6-20
 - 複合文からの検索値, 12-23
 - プリコンパイル SQL, 6-5
 - COBOL での制限, 6-18
 - FORTRAN での制限, 6-21
 - プログラムの移植性, 10-36
- SQLDA, 8-5, 11-9
 - 宣言, 11-10
 - パラメータ・マーカー
 - 戻されるデータ型, 11-9
 - ヘッダー・ファイル, 11-10
 - モジュール・プロシージャでの宣言, 4-5
- SQLDA2, 8-5, 11-9
 - 宣言, 11-10
 - モジュール・プロシージャでの宣言, 4-5
- SQLDA データ構造, 11-9
- SQLDA データ構造, 11-9
- SQLERRD 配列, 8-10
 - FORTRAN での宣言, 6-20
 - 処理した行の数が格納される要素, 10-7
- sqllibs.make ファイル, 7-11
- SQLLIBS グローバル・シンボル, 5-3, 7-11
- SQLSTATE ステータス・パラメータ, 8-4, 10-3, 10-5
 - SQL モジュール, 4-4
 - 値
 - エラー, 10-6
 - 警告, 10-6
 - 成功, 10-6
 - データなし条件, 10-6
 - デッドロック, 10-34
 - ロックの競合, 10-34
 - ストアド・ファンクション, 13-8
 - ストアド・プロシージャ, 13-6
 - 宣言, 10-5
 - C 言語, 8-27
 - 複合文からの検索値, 12-23
 - プリコンパイル SQL, 6-5
 - プログラムの移植性, 10-36
- SQL 記述子領域
 - SQLDA および SQLDA2 を参照
- SQL 通信領域
 - SQLCA ステータス・パラメータを参照
 - SQLCA データ構造を参照
- SQL で構築するプログラム
 - フロー制御文を参照
- SQL プリコンパイラ, 6-1
 - Ada 固有の要件, 6-14
 - デバッグ, 7-13
 - Ada プログラム, 6-14
 - COBOL 固有の要件, 6-18
 - COBOL の例, 19-6
 - COMPILETIME オプション, 15-3
 - C 固有の要件, 6-16
 - EXEC SQL フラグ, 6-4
 - FETCH 文, 18-3
 - FORTRAN 固有の要件, 6-20
 - デバッグ, 7-13
 - INCLUDE 文を使用してパラメータを宣言, 8-8

- Pascal 固有の要件, 6-22
 - デバッグ, 7-13
- Pascal プログラム
 - 文の順序, 6-23
- PL/I 固有の要件, 6-23
- RDB\$DBHANDLE デフォルト・エイリアス, 20-7
- SQL 文の区切り, 6-4
- アンダースコア, 6-5
- エラー・ログ・ファイル, 6-10
- エラーを修正, 6-9
- 大文字と小文字の区別, 6-5
- キーワード, 6-5
- 起動, 6-6, 6-7
- 句の埋込み, 6-11
- コマンドライン修飾子, 6-9
 - ansi, 6-8
 - ANSI_FORMAT, 6-18
 - cm deferred, 16-40
 - cm immediate, 10-37, 16-40
 - conn, 17-11
 - CONNECT, 17-11
 - CONSTRAINTS=IMMEDIATE, 10-37
 - extern, 7-11
 - EXTERNAL_GLOBAL, 7-7, 7-9
 - form ansi, 6-8, 6-18
 - l, 6-6
 - LIST, 6-10
 - list, 6-8, 6-10
 - match, 6-8
 - noconn, 17-11
 - NOCONNECT, 17-11
 - noextern, 7-11
 - NOEXTERNAL_GLOBAL, 7-7, 7-9
 - pass, 15-7
 - PASSWORD_DEFAULT, 15-7
 - user, 15-7
 - USER_DEFAULT, 15-7
- コロン (:), 8-12
- コンパイル
 - 接続用, 17-11
- 出力ファイル, 6-7
- 処理されるパラメータ名, 8-12
- 接続の無効化, 17-11
- 接続の有効化, 17-11
- ソース・ファイルの処理, 6-6
- 定義, 1-2
- デフォルトの日付書式の設定, 9-3
- 入力ファイル, 6-7
- ハイフン (-), 6-5
- パラメータ名, 6-5, 8-12
- 日付時刻データ型, 9-5
- 複合文, 12-7
- 複合文のラベル, 12-12
- プログラム開発, 1-3
- マルチスキーマ
 - コンパイル, 20-9
 - スキーマ・オブジェクトの命名, 20-8
 - デフォルトの設定, 20-7
 - リンク, 20-10
- よくある使用上の誤り, 8-24, 8-26
- リポトリ定義, 15-3
- SQL プログラムの開発
 - ガイドライン, 2-1
- SQL プログラムのリンク, 7-1
- SQL 文
 - ATTACH, 15-1
 - CALL, 12-4, 12-14, 13-11, 14-32
 - CASE, 12-3, 12-8
 - CLOSE, 18-4
 - 動的 SQL, 11-35
 - COMMIT, 16-42, 16-44
 - 複合文, 12-17
 - CREATE FUNCTION, 14-4, 14-9, 14-13, 14-24
 - CREATE PROCEDURE, 14-6
 - DECLARE CURSOR, 18-3
 - DECLARE TRANSACTION, 16-5, 16-8
 - 複合文, 12-18
 - DELETE, 19-8
 - DESCRIBE, 11-4, 11-33, 11-35
 - 複合文, 12-18
 - DROP FUNCTION, 13-12
 - DROP PROCEDURE, 13-12
 - EXECUTE, 11-4, 11-32, 11-33
 - 複合文, 12-18
 - EXECUTE IMMEDIATE, 11-4
 - 複合文, 12-18
 - FETCH, 18-3
 - 動的 SQL, 11-4, 11-35
 - プログラム, 18-3
 - FOR, 12-3, 12-10
 - GET DIAGNOSTICS, 12-4, 12-21, 12-22
 - IF, 12-4, 12-6
 - INCLUDE SQLCA, 10-7
 - INSERT, 19-2

- LEAVE, 12-4, 12-13
- LOOP, 12-9
- OPEN, 18-3
 - 動的 SQL, 11-4, 11-35
- PREPARE, 11-4, 11-32, 11-33, 11-35
 - 複合文, 12-18
- RELEASE, 11-4
- RETURN, 13-7, 13-9
- ROLLBACK, 16-42, 16-44
 - 複合文, 12-17
- SELECT
 - 動的 SQL, 11-3, 11-22, 11-32
- SELECT 文以外の SQL 文
 - 動的 SQL, 11-2, 11-5, 11-8
- SET ALL CONSTRAINTS, 16-40
- SET DEFAULT CONSTRAINT MODE, 16-40
- SET FLAGS
 - NOPREFIX キーワード, 12-20
 - TRACE キーワード, 12-18
- SET HOLD CURSOR, 18-13
- SET TRANSACTION, 16-6, 16-8
 - 複合文, 12-17
- SET 代入文, 12-4
- SIGNAL, 12-4, 12-22
- SQL モジュール・プロシージャ, 4-2
- TRACE, 12-4, 12-18
- UPDATE, 19-5
- WHENEVER, 10-10
- 拡張動的 DECLARE CURSOR, 11-4, 11-19, 11-23, 11-32, 11-35
- 言語のソースに宣言をコピー, 8-8
- 実行可能
 - SQL モジュール・プロシージャ, 4-14, 4-15
 - エラー処理, 10-5
- 処理された行の数を検索, 10-7
- チェックと処理
 - SQL モジュール, 5-2
- テストする
 - 対話的, 2-4
 - プログラム開発のために, 2-3
- 動的 DECLARE CURSOR, 11-4, 11-19, 11-23
- 動的に実行される文のカテゴリ, 11-2
- 複合, 12-1
- プリコンパイル・プログラムの複数行文, 6-4
- モジュール・プロシージャを作成, 3-4
- モニター, 10-4
 - RDB\$LU_STATUS の使用, 10-15
 - SQLCODE の使用, 10-7
 - WHENEVER 文の使用, 10-10
- SQL 文の終了
 - Ada プログラム, 6-6
 - COBOL プログラム, 6-6
 - C プログラム, 6-6
 - FORTRAN プログラム, 6-6
 - Pascal プログラム, 6-6
 - PL/I プログラム, 6-6
- SQL モジュール, 3-1, 4-1, 4-6, 5-1
 - COMPILETIME 句, 15-3
 - DECLARE 文, 3-11
 - FROM DICTIONARY 句, 8-11
 - FROM 句を使用してパラメータを宣言, 8-11
 - FROM パス名句, 8-3, 8-11
 - PARAMETER COLONS 句, 3-11
 - エイリアス句, 3-10, 15-12
 - エラー処理, 10-7
 - カタログ句, 3-9
 - 言語句, 3-8
 - パラメータ・データ型の効果, 4-11
 - パラメータの受渡しメカニズムの効果, 4-12
 - コロン (:), 8-12
 - コンパイル, 5-2
 - 実行可能文の指定, 4-14
 - 接続用, 17-11
 - 他のモジュールとのリンク, 3-11
 - 認可句, 3-10
 - ファイル, 3-4
 - ファイルの作成, 1-2
 - 複合文のラベル, 12-12
 - 部分, 3-4
 - DECLARE 文, 3-11
 - PARAMETER COLONS 句, 3-11
 - エイリアス句, 3-10
 - カタログ句, 3-9
 - 言語識別子, 3-8
 - コメント, 3-6
 - スキーマ句, 3-9
 - 名前, 3-6
 - 認可句, 3-10
- プロシージャ, 4-2
 - 受渡しメカニズムの確認, 4-14
 - 終了, 12-13
 - 複合文, 12-7, 12-11, 12-17
 - ホスト言語モジュールからのコール, 3-12
 - 命名, 4-3

- 大文字と小文字の区別, 4-4
- プロシージャ・パラメータの順序, 4-10
- プロシージャ・パラメータの宣言, 4-4
- ホスト言語プログラムからのコール, 1-2
- マルチスキーマ, 20-4
 - サンプル, 20-6
 - デフォルト設定, 20-3
- 命名, 3-6
 - よくある使用の誤り, 8-24, 8-26
- SQL モジュール言語, 3-1, 3-2, 3-4, 4-1, 5-1
 - C 言語, 8-28
 - パラメータの宣言, 8-12
 - INTEGER データ型, 8-29
 - SMALLINT データ型, 8-29
 - SQL 文, 4-16
 - SQL モジュールを参照
 - コンテキスト・ファイル, 5-5
 - 接続を使用したサンプル, 17-11
 - パフォーマンスの改善, 5-4
- SQL モジュール言語と Rally, 3-2
- SQL モジュール言語の FROM DICTIONARY 句, 8-11
- SQL モジュール言語の FROM パス名句, 8-3
- SQL モジュール内の BASIC 言語識別子, 3-8
- SQL モジュール内の DECLARE 文, 3-11
- SQL モジュール内の GENERAL 言語識別子, 4-12
- SQL モジュール内の PLI 言語識別子, 3-8
- SQL モジュールの LANGUAGE 句, 3-8
- SQL モジュールの MODULE 句, 3-6
- SQL モジュールの言語識別子, 3-8, 4-11, 4-12
- SQL モジュールの処理, 5-3
- SQL モジュールのプロシージャ, 4-2
 - 実行可能文, 4-14
 - 単純, 4-2
 - パラメータ, 4-9
 - パラメータの順序, 4-10
 - パラメータの宣言, 4-4
 - 必要なパラメータ, 4-5
 - 複合文, 4-15
 - 複数文, 4-2
 - プロシージャ・パラメータのデータ型, 4-10
 - ホスト言語モジュールからのコール, 3-12
 - 命名, 4-3
 - 大文字と小文字の区別, 4-4
- SQL モジュール・プロシージャ
 - 複合文, 12-7, 12-11, 12-17
- SQL モジュール・プロシージャの実行可能 SQL 文, 4-2, 4-14

- SQL モジュール・プロシージャへのコール
 - パラメータ, 4-9
- SQL モジュール・プロセッサ, 5-2
 - コマンドライン修飾子
 - cm deferred, 16-40
 - cm immediate, 10-37, 16-40
 - conn, 17-11
 - CONNECT, 17-11
 - CONSTRAINTS=IMMEDIATE, 10-37
 - extern, 7-11
 - EXTERNAL_GLOBAL, 7-7, 7-9
 - FLAG_NONSTANDARD, 3-14, 20-6
 - lc_proc, 4-4
 - コマンドライン・パラメータ, 5-2
 - 起動, 5-2
 - コマンドライン修飾子
 - LONG_SQLCODE, 8-28
 - lsqlcode, 8-28
 - noconn, 17-11
 - NOCONNECT, 17-11
 - noextern, 7-11
 - NOEXTERNAL_GLOBAL, 7-7, 7-9
 - NOPARAMETER_CHECK, 5-4
 - PARAMETER_CHECK, 5-4
 - pass, 15-7
 - PASSWORD_DEFAULT, 15-7
 - std, 3-14, 20-6
 - user, 15-7
 - USER_DEFAULT, 15-7
 - コンパイル, 5-2
 - 接続用, 17-11
 - 接続の無効化, 17-11
 - 接続の有効化, 17-11
 - 接続の有効化と無効化, 17-11
 - 定義, 1-2
 - データ書式の設定, 9-3
 - 日付時刻データ型, 9-8
 - プログラム開発, 1-2
 - マルチスキーマ, 20-6
- SQL ライブラリ
 - リンク, 5-3, 7-11
- SQL ルーチン
 - sql_deregister_error_handler, 10-19
 - sql_get_error_handler, 10-19
 - sql_get_error_text, 10-28
 - sql_get_message_vector, 10-13, 10-14
 - sql_register_error_handler, 10-18

- sql_signal, 10-27
- エラー処理用, 10-4, 10-18
- プログラムで宣言, 6-17, 10-30
- ルーチンを参照
- std コマンドライン修飾子, 3-14
 - 非標準 ANSI/ISO SQL 構文の表示, 20-6
- SUM 関数
 - NULL 結果, 8-18
- SYSTEM LOGICAL_NAME TRANSLATION 句
 - CREATE ルーチン文, 14-34

T

- TCP/IP ネットワーク・プロトコル, 15-9, 15-10
- TIMESTAMP データ型, 9-10
 - SQL プリコンパイラ, 9-5
 - 精度, 9-2
- TIME データ型および SQL プリコンパイラ, 9-5
- TRACE 文, 12-4, 12-18

U

- UCX サービス
 - 代替, 15-10
- UNION 演算子, 19-3
- UNIVERSAL シンボル, 14-26, 14-27
- UPDATABLE キーワード
 - DECLARE 変数句, 12-5
- UPDATE...RETURNING を使用する dbkey 値, 19-8
- UPDATE 文, 19-5
 - dbkey 値を返す, 19-8
 - FOR 文, 12-10
 - RETURNING 句, 19-7
 - SQL モジュール, 4-8
 - プロシージャ・パラメータ, 4-8
 - 値を返す, 19-8
 - インジケータ・パラメータ, 8-22
 - カーソル, 18-4, 19-6
 - 検索条件を含む, 19-5
 - 処理する行の数, 8-10
 - 制限, 15-14
 - 複数行に影響を与える, 18-6
 - プログラム, 19-6
- UPDATE 文の RETURNING 句, 19-7, 19-8
- UPDATE 文の SET 句の等号 (=), 19-6
- USER_DEFAULT 修飾子, 15-7
- user オプション

- SQL プリコンパイラ・コマンドライン, 15-7
- SQL モジュール・プロセッサ・コマンドライン, 15-7

- USER 句, 15-6

- USING 句, 15-6

V

- VARCHAR データ型, 8-28
 - C 言語, 6-17
 - 動的 SQL, 4-5

W

- WAIT オプション, 16-12
- WAIT トランザクション・オプション, 16-12, 16-28
- WHENEVER 文, 10-3, 10-10
 - CONTINUE オプション, 10-12
 - FORTRAN DO ループへの影響, 6-21
 - NOT FOUND オプション, 10-11
 - SQLERROR オプション, 10-11
 - SQLWARNING オプション, 10-11
 - SQL モジュールでは使用されない, 4-16
 - オフにする, 10-12
 - 使用時のエラーを回避するチェックリスト, 10-12
 - プリコンパイル中の処理, 10-11
 - ループの回避, 10-12
- WHERE CURRENT OF 句
 - DECLARE CURSOR 文, 19-4
 - UPDATE 文, 19-6
- WHERE 句, 19-5
 - DELETE 文, 19-8
 - SQL モジュール
 - プロシージャ・パラメータ, 4-7
 - UPDATE 文, 19-5
 - インジケータ・パラメータを含めない, 8-24
 - 外部関数, 14-30
- WHILE 句
 - LOOP 文, 12-9
- WHILE 述語句, 12-4
- WITH HOLD 句
 - DECLARE CURSOR 文, 18-8, 18-12
- WRITE ロック・タイプ, 16-28

あ

- アクセス

- 競合, 16-28
- 値式
 - 外部プロシージャを起動, 14-3
 - パラメータ用, 8-2
- 値の割当て
 - プロシージャ・パラメータの使用, 8-12
 - ホスト言語パラメータの使用, 8-12
- アトム性
 - 複合文, 12-15
- アンダースコア (_)
 - Ada プログラム, 8-27
 - ハイフン (-) との関連, 6-5

い

- 移植性
 - DATE VMS, 9-15
 - SQL モジュール言語, 3-14
 - エラー処理技術, 10-36
 - 外部ルーチン, 14-45
 - コンテキスト・ファイルの使用, 6-8, 6-12
 - 日付時刻データ型, 9-12
- 依存オブジェクト, 13-13
- 依存関係のタイプ, 13-13, 13-14
 - 言語セマンティクスの依存関係, 13-16
 - デフォルトの評価, 13-17
 - デフォルトの予約, 13-17
 - トランザクションの依存関係, 13-17
 - プロシージャの依存関係, 13-15
- 依存関係の追跡, 13-13, 13-18
 - RDB\$INTERRELATIONS で保存できる情報, 13-14
 - 依存オブジェクト, 13-13
 - 参照オブジェクト, 13-13
- 一行形式の選択文, 18-6
- 一貫性レベル
 - 分離レベルを参照
- イメージ・セクション, 7-6
- インクルード・ファイル
 - sql_rdb_headers.h, 6-17, 10-30
 - sql_sqlda.h, 8-5, 11-10
- インジケータ配列, 8-4, 8-18, 8-25
 - SQL モジュール, 8-20
 - プリコンパイル SQL, 8-20
- インジケータ・パラメータ, 8-4, 8-18, 18-6
 - C, 8-27
 - FETCH 文, 8-21
 - INSERT 文, 8-20, 8-22, 8-23

- SQL モジュール, 8-20, 8-31
- UPDATE 文, 8-20
- WHERE 句, 8-20
- WHERE 句または HAVING 句で使わない, 8-24
- 値の解釈, 8-21
- 値の保管, 8-22
- 検索値, 8-21
- 個々に名前をつけたパラメータ, 8-20
- 使用した共通の誤り, 8-24
- ストアド・ファンクション, 13-8
- ストアド・プロシージャ, 13-6
- ストアド・プロシージャのコール, 13-11
- 宣言, 8-18
- 配列, 8-18
- 配列要素, 8-18
 - 予期しない割当て, 8-25
- 必要, 8-18
- プリコンパイル SQL, 8-20
- ホスト言語構造用, 8-18
- インジケータ・パラメータを使用して値を格納, 8-22
- インジケータ変数
 - インジケータ・パラメータを参照
- インターネット・サービス
 - 代替, 15-10
- インデックス
 - ロック, 16-28
- インデックスのロック, 16-31, 16-33
 - インデックスを使用しない場合, 16-28
- 引用符 ("), 3-7
- 引用符、一重 (') > INCLUDE FROM DICTIONARY 句, 8-9

う

- 失われた値
 - NULL 値を参照
- 埋込み SQL
 - SQL プリコンパイラを参照
- 複合文, 12-7
- 埋もれた更新
 - 定義, 16-17

え

- 英語以外の照合順番の CONTAINING 演算子, A-3
- 英語以外の照合順番の LIKE 演算子, A-3

英語以外の照合順番の STARTING WITH 演算子, A-3

エイリアス, 15-12, 15-13

RDB\$DBHANDLE, 15-13

SQL モジュール, 3-10

共有可能なイメージ, 7-7, 7-9

指定する場合, 16-3

宣言, 15-2

デフォルト

SQL プリコンパイラ, 20-7

SQL モジュール, 20-3

デフォルトのデータベース, 15-13

SQL モジュール, 15-12

トランザクション文, 16-21

複数のデータベースへの接続, 15-13

プリコンパイラのデフォルト, 20-7

モジュールに渡る共有, 7-12

モジュールのデフォルト, 20-3

エラー

SQLCODE に返された値, 10-8

確認, 10-31

コード

RDB\$LU_STATUS, 10-31

SQLCODE, 10-31

実行時, 7-13

実行時の処理

データベース接続, 15-3

処理, 10-1

データの確認, 10-31

データベース

整合性, 10-31

接続, 10-35

デッドロック, 10-32, 10-34

トランザクションの開始, 10-35

プログラムのデバッグ, 7-13

無限ループ, 10-12

メッセージ, 10-27

メッセージの表示, 10-27

ロギング

SQL プリコンパイル・エラー, 6-10

コンパイル時エラー, 6-10

ロックの競合, 10-32

エラー

エラー・コード、エラー処理、例外条件を参照

エラー・コード

RDB\$LU_STATUS

RDB\$_NOT_VALID, 10-31

RDB\$_DEADLOCK, 10-34

RDB\$_INTEG_FAIL, 10-31

RDB\$_LOCK_CONFLICT, 10-34

RDB\$_NO_DUP, 10-31

SQLCODE, 10-31

SQLCODE_DEADLOCK, 10-34

SQLCODE_EOS, 10-8

SQLCODE_LOCK_CONFLICT, 10-34

SQLCODE_NOT_VALID, 10-31

SQLCODE_SUCCESS, 10-8

SQLSTATE, 10-31

エラー処理

RDB\$MESSAGE_VECTOR の使用, 10-13, 10-14

sql_get_error_text ルーチンの使用, 10-28

sql_get_message_vector の使用, 10-13, 10-14

sql_signal ルーチンの使用, 10-27

SQL ルーチンの使用, 10-4, 10-18

オプション, 10-2

外部ルーチン, 14-42

警告

SQLCODE の使用, 10-8

WHENEVER 文の使用, 10-11

実行時, 10-2

sql_get_error_text ルーチンの使用, 10-28

sql_signal ルーチンの使用, 10-27

SQLCODE の使用, 10-11

WHENEVER 文の使用, 10-11, 10-12

オプションの要約, 10-2

オンラインの例, 10-4

条件文の使用, 10-12

メッセージの表示, 10-27

シンボリック・エラー・コードの宣言, 10-16

複合文, 12-22, 12-23

メッセージの表示

sql_get_error_text ルーチン, 10-28

sql_signal ルーチン, 10-27

ユーザー指定のメッセージ・ファイル, 10-30

エラーの場合有効, 10-31

エラー処理

お

オープンしたカーソルの保持

保持可能カーソルを参照

大文字と小文字の区別

C 言語

SQLCODE, 8-27

- SQLSTATE, 8-27
- SQL モジュール・プロシージャ名, 4-4
- プリコンパイル・プログラム, 6-5
- オブジェクト・ファイル
 - アーカイブへの挿入, 7-12
 - プリコンパイル・プログラム
 - 位置の指定, 6-8
 - ライブラリへの挿入, 7-12
 - リンク
 - CompaqTru64 UNIX, 7-10
 - OpenVMS, 7-2
- オプション・ファイル
 - OpenVMS Linker ユーティリティ, 7-3
 - 外部ルーチンの作成, 14-26, 14-27, 14-28

か

- カーソル, 18-1
 - CLOSE 文, 18-4
 - CLOSE 文による削除, 18-4
 - COMMIT 文, 18-8, 18-11, 18-12
 - DECLARE 文, 4-7, 18-3
 - FETCH 文, 18-3
 - FOR 文, 12-10
 - OPEN 文, 4-7, 18-3
 - OPEN 文による削除, 18-3
 - ROLLBACK 文, 18-8, 18-11, 18-12
 - エラー処理, 10-8, 10-11
 - オープン, 18-3, 18-8
 - 異なるトランザクション, 18-4
 - 拡張動的, 18-7, 18-18
 - 拡張動的宣言, 11-32
 - カテゴリ, 18-6
 - かわりに SELECT 文を使用, 18-6
 - 関連した結果テーブル, 18-5
 - 基本操作, 18-2
 - 行の更新, 19-6
 - 行の削除, 19-8
 - 制限, 19-9
 - 行ポインタ, 18-3
 - 定義, 18-2
 - 空白, 18-9
 - クラス, 18-6
 - 現在行, 18-3
 - 検索, 18-3
 - 再オープン, 18-3, 18-4
 - 作成, 18-3
 - 使用したロック, 16-27, 18-11, 18-12
 - 使用の基準, 18-5, 18-9
 - スクロール可能リスト, 18-4, 18-15
 - 静的, 18-7
 - 静的および動的の間の相違, 18-17
 - 宣言, 18-3
 - SQL モジュール, 4-7
 - 上書きできない, 18-11
 - 選択式, 18-3
 - 定義, 18-2
 - テーブル, 18-6, 18-9
 - 更新, 18-7
 - 更新専用, 18-7
 - 挿入専用, 18-8
 - 読取り専用, 18-7
 - 動的, 11-23, 18-7, 18-17
 - 拡張, 11-35, 18-7, 18-18
 - 閉じる, 18-4, 18-8
 - 名前, 18-3
 - ビューとの比較, 18-5
 - 複合文, 12-10
 - 分類, 18-6
 - 保持可能, 18-8, 18-11
 - モード, 18-6
 - 読取り専用, 19-9
 - リスト, 18-6, 18-13, 19-4
 - 位置付け, 18-13
 - 挿入のみ, 18-8
 - 読取り専用, 18-8
 - 列の更新, 19-6
 - ロック, 16-29
- カーソルのオープン, 18-3
- 外部関数, 14-45
 - LANGUAGE 句
 - GENERAL キーワード, 14-38
 - 受渡しメカニズム, 14-4
 - 外部ルーチンを参照
 - 起動, 14-3, 14-11, 14-13, 14-30, 14-31
 - 言語特有のコーディングのガイドライン, 14-38
 - 作成, 14-9
 - 定義, 14-4
 - サポートしている言語, 14-5, 14-38
 - 事前定義, 14-9, 14-23
 - トリガー, 14-31
 - パラメータ・データ型, 14-4
 - 戻り値, 14-5
 - ユーザー定義, 14-12

- 外部プロシージャ, 14-45
 - 受渡しメカニズム, 14-6
 - 外部ルーチンを参照
 - 起動, 12-14, 14-2, 14-3, 14-32
 - 複合文, 12-14
 - 定義, 14-6
 - 定義の作成, 14-6
 - データ型, 14-6
 - 外部ルーチン, 14-45
 - Ada, 14-38
 - COBOL, 14-40
 - C 言語, 14-39
 - FORTRAN, 14-37, 14-40
 - LANGUAGE 句, 14-5, 14-7
 - Pascal, 14-37, 14-40
 - アクティブ化, 14-32, 14-34, 14-35
 - 移植性, 14-45
 - 受渡しメカニズム, 14-36
 - 開発, 14-3
 - 外部関数, 外部プロシージャを参照
 - 起動, 14-11, 14-13, 14-24, 14-30, 14-32
 - 共有メモリー, 14-34, 14-44
 - 作成, 14-9
 - オプション・ファイル, 14-26, 14-27, 14-28
 - 共有オブジェクト, 14-29
 - 共有可能なイメージ, 14-25, 14-26, 14-27
 - 定義, 14-3
 - サポートしている言語, 14-7
 - 実行状態のテスト, 14-3
 - 実行特性, 14-32
 - ジャケット・ルーチン, 14-10, 14-23
 - スコープ, 14-3, 14-7
 - 制限, 14-43
 - セキュリティ, 14-34
 - 通知, 14-41
 - 定義, 14-2
 - 削除, 14-8
 - 作成, 14-3
 - 変更, 14-8
 - トラブルシューティング, 14-44
 - パラメータ・データ型, 14-36
 - 非アクティブ化, 14-35
 - 例外処理, 14-42
 - 拡張動的 DECLARE CURSOR 文, 11-4, 11-19, 11-23, 18-7, 18-18
 - パラメータの使用, 11-32, 11-35
 - 拡張動的カーソル, 11-4, 11-19, 11-23, 11-32, 11-35, 18-7, 18-18
 - 確認
 - インデックスに対する UNIQUE オプション, 10-31
 - ストアド・ルーチン, 13-18
 - 制約, 10-31
 - チェック, 10-31
 - 列に対する NOT NULL 制約, 10-31
 - 列に対する UNIQUE 制約, 10-31
 - カタログ, 20-2
 - RDB\$CATALOG システム・データベース・デフォルト, 20-2, 20-7
 - SQL モジュールでの命名, 3-9
 - 定義, 20-2
 - デフォルト, 20-3
 - 設定する
 - SQL プリコンパイラ, 20-7
 - SQL モジュール, 20-3
 - カレント・レコード
 - 現在行を参照
 - 関数
 - CAST, 9-4, 9-8
 - EXTRACT, 9-8
 - NULL 結果の扱い, 8-18
 - 外部, 14-45
 - ストアド, 13-1
 - ストアド・ファンクション、ストアド・ルーチンを参照
 - 完了条件, 10-2
 - 検出, 10-4
 - SQLCODE の使用, 10-6
 - SQLSTATE の使用, 10-5
 - 成功
 - SQLCODE の使用を検出, 10-8
 - SQLSTATE の使用を検出, 10-6
 - データなし, 18-9
 - SQLCODE の使用を検出, 10-8
 - SQLSTATE の使用を検出, 10-6
 - WHENEVER 文の使用を検出, 10-11
 - 複合文, 12-22, 12-23
 - 完了ステータス
 - 対話型 SQL, 16-44
- ## き
-
- 規格
 - ANSI/ISO SQL, 20-1

基数文字, A-2

起動

 ストアド・ファンクション, 13-12

 ストアド・プロシージャ, 12-15, 13-10

キャラクタ・セット

 照合順番, A-4

 セッション用, 3-7

 デフォルト, 3-8

行

 SQL 文が処理した行の数, 10-7

 現在, 18-3, 18-9

 検索, 18-2

 更新, 19-5

 プログラムの使用, 19-6

 削除, 19-8

 挿入, 19-2

 行がリストを含む場合, 19-5

 データの検索を参照

 ポインタ, 18-3

 ロック, 16-25

行数

 SQL 文による処理, 8-10

共通データ・ディクショナリ

 リポジトリを参照

行の格納

 INSERT 文を参照

行ポインタ, 18-3

 位置, 18-3

 カーソルを再オープンした後の位置, 18-4

 行の削除, 18-4

 設定, 18-3

 定義, 18-2

共有オブジェクト

 外部ルーチン

 CompaqTru64 UNIX システム, 14-29

共有可能なイメージ, 7-2

 sql_register_error_handler ルーチン, 10-23

 インストールする, 7-10

 オプション・ファイルを使用して指定, 7-3

 外部ルーチン, 14-25

 OpenVMS Alpha システム上, 14-27

 OpenVMS VAX システム上, 14-26

 共有ハンドル, 7-6, 7-7

 なしに, 7-6

 作成, 7-5

 転送ベクトル, 7-9

 リンク, 7-3

共有メモリー

 外部ルーチン, 14-34, 14-44

共有モード, 16-3, 16-10, 16-27, 16-28

 EXCLUSIVE, 16-10, 16-27, 16-28

 PROTECTED, 16-7, 16-10, 16-27, 16-28

 SHARED, 16-28

 アクセスの競合, 16-28

 テーブルの予約, 16-28

<

空白行

 SQL モジュールを含む, 3-6

クエリー・オプティマイザ, 16-32

クライアント / サーバー・システム

 ストアド・ルーチン, 13-3

クライアントサイト・バインディング

 外部ルーチン, 14-7, 14-32

グローバル・シンボル

 SQLLIBS, 5-3, 7-11

け

計画ファイル

 コンテキスト・ファイルを参照

警告

 SQLCODE を使用して検出, 10-11

継続行

 FORTRAN の場合, 6-20

桁区切り文字, A-2

結果テーブル, 18-2

 FETCH 文, 18-3

 カーソル, 18-5

 プログラム, 18-5

 カーソルの再オープン, 18-4

 更新カーソル, 18-7

 更新専用カーソル, 18-7

 異なるトランザクション, 18-4

 削除, 18-4, 18-8, 18-11, 18-12

 作成, 18-3

 式の選択, 18-3

 スクロール可能なリスト・カーソル, 18-15

 挿入専用カーソル, 18-8

 定義, 18-2, 18-5

 ビューとの関連, 18-5

 読取り専用カーソル, 18-7

 ランダム・アクセス, 18-15

権限

- DISTRIBTRAN, 15-11
- 外部ルーチン, 14-33
- ストアド・ルーチン, 13-3, 13-4
- 削除, 13-12
- チェック
 - SQL プリコンパイラ, 20-7
 - SQL モジュール・プロセッサ, 20-4

権限の確認

- カーソル, 18-5

言語

- SQL プリコンパイラによるサポート, 6-13
- SQL モジュール言語に対するサポート, 3-2
- SQL モジュール処理の識別子, 4-11, 4-12
- 外部ルーチンに対するサポート, 14-38
- 入力 / 出力用の指定, A-2

言語セマンティクスの依存関係, 13-16

現在行, 18-9

こ

更新

- データ, 19-5
- 更新カーソル, 18-7
- 更新専用カーソル, 16-20, 18-7
- 分離レベル, 16-15

構成パラメータ

- ATTACH 文, 15-13
- RDB_DEBUG_FLAGS, 12-18
- RDB_DEBUG_FLAGS_OUTPUT, 12-19
- RDB_RTX_SHRMEM_PAGE_CNT, 14-34, 14-44
- RDB_VALIDATE_ROUTINE, 13-21
- SQL_ALTERNATE_SERVICE_NAME, 15-10
- SQL_DATABASE, 15-2

構成ファイル

- .dbsrc, 15-2, 15-5, 15-8
- RDB\$CLIENT_DEFAULTS.DAT, 15-5, 15-8

構造, 8-15

- COBOL での宣言, 6-19
- FORTTRAN での宣言, 6-22
- SQL で
 - 制限, 8-5
- SQL プリコンパイラによる拡張, 8-25
- SQL モジュール・プロセッサによる拡張, 8-25
- 定義, 8-24
- ブロック
 - Ada, 6-14

COBOL, 6-18

C 言語, 6-17

FORTTRAN, 6-21

Pascal, 6-22

PL/I, 6-23

レコードとの比較, 8-3

構造化

テーブル, 19-4

構造化テーブル, 19-4

構造、レコードとの比較, 8-3

コールバック

データベース

外部ルーチン, 14-16

コール・パラメータ

実パラメータを参照

国際化, A-1

Oracle Rdb サポート, A-1

コピー

リポジトリ定義

FROM 句の使用, 8-11

INCLUDE 文の使用, 8-8, 8-9

コメント

SQL モジュールに含む, 3-6

小文字の名前

SQL モジュール, 4-4

コロロン (:)

SQL モジュール, 3-11, 8-12

プリコンパイル・プログラム, 8-12

コンテキスト・ファイル, 5-5, 6-8, 6-12

SQL プリコンパイラ, 6-12

DECLARE MODULE 文の挿入, 6-11

SQL モジュール言語, 5-5

宣言, 6-12

デフォルトのカatalogおよびスキーマ, 20-3, 20-8

プリコンパイラ・コマンドライン・パラメータ,

6-8

プログラムの移植性, 6-8, 6-12

コンパイル

SQL プリコンパイラ、SQL モジュール・プロセッサを参照

SQL モジュール

言語ソース・ファイル, 5-2

SQL モジュール・プロセッサ

接続用, 17-11

SQL モジュール・プロセッサの起動, 5-2

接続用 SQL プリコンパイラ, 17-11

ホスト言語

出力ファイル, 6-7
入力ファイル, 6-7

さ

サーバーサイト・バインディング
外部ルーチン, 14-7, 14-32
削除
外部ルーチン定義, 14-8
行, 19-8
制限, 19-9
ストアド・ファンクション, 13-12
ストアド・プロシージャ, 13-12
ストアド・モジュール, 13-12
リスト・データ, 19-9
算術式
NULL 結果の扱い, 8-18
参照されたオブジェクト, 13-13
サンプル・ディレクトリ, 3-14
サンプル・データベース
位置, 1-5
サンプル・プログラム
位置, 1-5
プログラム、サンプルを参照

し

時間書式
変更, A-2
システム・サービス・コール
外部ルーチン, 14-45
事前定義の外部関数
直接, 14-9
実行可能イメージ, 7-2
作成, 7-1
実行可能イメージ、作成, 7-1
実行可能イメージを作成する LINK コマンド (DCL),
7-2
実行者の権利モジュール, 13-4
実パラメータ, 4-9
パラメータを参照
ジャケット・ルーチン, 14-10
書込み, 14-23
修飾子
SQL プリコンパイラ・コマンドライン, 6-9
SQL モジュール・プロセッサ・コマンドライン,
5-2

重複値
エラー, 10-31
重複値エラー, 10-31
終了
複合文, 12-13
プログラム, 16-44
プロシージャ, 12-13
終了ラベル
FOR 文用, 12-13
LOOP 文用, 12-13
複合文, 12-12
述語
複数国用のキャラクタ・セットの動作, A-4
順序番号
ストアド・ファンクションで生成, 13-9
条件付き演算子
CONTAINING 演算子
英語以外の照合, A-3
LIKE 演算子
英語以外の照合, A-3
STARTING WITH 演算子
英語以外の照合, A-3
動的 SQL での使用, 11-9
照合順番
キャラクタ・セットの比較, A-4
指定, A-3
述語における動作, A-4
順序, A-4
処理, 5-3
シンボリック・エラー・コード, 10-16
宣言および使用方法, 10-16
シンボル
グローバル
SQLLIBS, 5-3, 7-11
シンボル (DCL)
起動の定義
SQL モジュール・プロセッサ, 5-2
プリコンパイラの起動, 6-6
制限, 6-7
シンボル・ベクトル, 7-5, 7-9
外部ルーチン, 14-28

す

スキーマ, 20-2
RDB\$SCHEMA, 20-2, 20-8
SQL モジュールでの名前, 3-9

- オブジェクト, 20-2
- 定義, 20-2
- デフォルト, 20-3
 - SQL プリコンパイラの設定, 20-7
 - SQL モジュールの設定, 20-3
- スキーマ句, 3-9
- スクロール可能リスト・カーソル, 18-4, 18-15
- スコープ, 5-6
 - SQL モジュール, 5-6
 - 外部ルーチン, 14-3, 14-7
 - トランザクション, 16-21, 16-22, 16-23
- ステータス・コード
 - エラー・コード、ステータス・パラメータを参照
- ステータス・パラメータ
 - SQLCODE, 10-6
 - SQLCODE ステータス・パラメータ, SQLCA,
SQLSTATE ステータス・パラメータも参照
 - SQLSTATE, 10-5
 - ストアド・ファンクション, 13-8
 - ストアド・プロシージャ, 13-6
 - 宣言
 - ストアド・プロシージャ, 13-11
 - ノン・ストアド・プロシージャ, 13-11
- ストアド・ファンクション, 13-1
 - NULL 値, 13-8
 - インジケータ・パラメータ, 13-8
 - 起動, 13-12
 - 削除, 13-12
 - 作成, 13-7, 13-9
 - データ型, 13-8
 - パラメータ, 13-8
 - パラメータ・モード, 13-8
- ストアド・プロシージャ, 13-1
 - NULL 値, 13-6, 13-11
 - 起動, 12-14, 13-10
 - 複合文, 12-14
 - 起動 (CALL), 13-11
 - 禁止されているインジケータ・パラメータ, 13-6
 - コール, 13-10
 - インジケータ・パラメータの使用, 13-6, 13-11
 - 削除, 13-12
 - 作成, 13-5
 - 使用されているデータ型, 13-6
 - 使用されるステータス・パラメータ, 13-6
 - ステータス・パラメータの宣言, 13-11
 - ネスト, 12-15
 - ネストする, 13-10

- パラメータ・モード, 13-6
- モジュールの名前, 3-6
- ストアド・モジュール, 13-2
 - 削除, 13-12
 - 定義, 13-3
- ストアド・ルーチン, 13-1
 - アクセスの特権, 13-3
 - 依存性のタイプ, 13-13, 13-14
 - 依存性の追跡, 13-13, 13-18
 - クライアント / サーバー処理, 13-3
 - 言語セマンティクスの依存関係, 13-16
 - 言語の起動, 13-4
 - 再有効化, 13-20, 13-21
 - 実行者の権限モジュール, 13-4
 - 実行の特権, 13-4
 - スキーマ・オブジェクトにアクセス, 13-4
 - 定義, 13-3
 - 定義者の権限モジュール, 13-4
 - トランザクションの依存関係, 13-17
 - プロシージャの依存関係, 13-15
 - 無効化, 13-14, 13-18
 - 無効な SQL 文, 13-18
 - 利点, 13-2
- ストアド・ルーチンの有効化, 13-20, 13-21
- スナップショット (.snp)・ファイル
 - EXCLUSIVE 共有, 16-30
 - 読取り専用トランザクション, 16-30
 - ロッキング, 16-27
 - ロック競合エラー, 16-30

せ

制御文

- フロー制御文、複合文を参照

制限, 4-16

SQL プリコンパイラ

- DO ループ (FORTRAN), 6-21
- IF 文 (FORTRAN), 6-21
- Pascal, 6-22
- SQLCODE フィールド (COBOL), 6-18
- SQLCODE フィールド (FORTRAN), 6-21
- 埋込み SQL 文 (COBOL), 6-18
- 起動, 6-7
- 行あたりの文字数 (FORTRAN), 6-20
- 継続行の行数 (FORTRAN), 6-20
- パラメータ名 (FORTRAN), 6-20
- 日付時刻, 9-19

- 静的カーソル, 18-7
- 精度
 - TIMESTAMP データ型, 9-2
 - 先頭
 - 日付時刻データ型, 9-2
 - 分数
 - 日付時刻データ型, 9-2
- 制約
 - 違反, 10-31
 - 値, 10-31
 - 外部関数, 14-30
 - 妥当性エラー, 10-31
 - 遅延可能, 16-40
 - 遅延可能ではない, 16-40
 - トランザクション, 16-12, 16-27
 - 評価, 10-32, 10-37, 16-39, 16-41
 - Verb 時間, 16-39
 - コミット時間, 16-39
 - 制御, 16-40
- セキュリティ
 - Oracle Rdb リモート, 15-11
 - 外部ルーチン, 14-34
 - システム
 - リモート・アクセス, 15-8
 - リモート・アクセス, 15-5
- セグメント文字列
 - リストを参照
- セッション
 - 各国語キャラクタ・セット, 3-7
 - キャラクタ・セット, 3-7
 - 識別子キャラクタ・セット, 3-7
 - 接続, 17-2
 - カレント, 17-3
 - 休止中, 17-3
 - 対話型との比較, 17-2
 - 定義, 17-2
 - デフォルト, 17-2
 - 対話型
 - 接続との比較, 17-2
 - デフォルトのキャラクタ・セット, 3-7
 - リテラル・キャラクタ・セット, 3-7
- セッションの各国語キャラクタ・セット, 3-7
- セッションのキャラクタ・セットの識別, 3-7
- セッションのデフォルトのキャラクタ・セット, 3-7
- セッションのリテラル・キャラクタ・セット, 3-7
- 接続, 17-1
 - CONNECT 文, 17-5
 - デフォルトのデータベース環境の複製, 17-6
 - DISCONNECT 文, 17-5
 - SET CONNECT 文, 17-5
 - SQL プリコンパイラ
 - 無効化, 17-11
 - 有効化, 17-11
 - SQL モジュール, 17-11
 - SQL モジュール・プロセッサ, 17-11
 - 無効化, 17-11
 - 有効化, 17-11
 - 暗黙的, 17-2
 - カーソル宣言, 18-4
 - 休止中のセッション, 17-3
 - 切替え, 17-9
 - 現行セッション, 17-3
 - 構成要素, 17-2
 - 作成, 17-5
 - サンプル
 - 使用する C プログラム, 17-11, 17-13
 - 使用する SQL モジュール, 17-11
 - 終了, 17-10
 - 制御, 17-5
 - セッション, 17-2
 - 切断, 17-10
 - 定義, 17-2
 - データベース環境, 17-2
 - デフォルト, 17-4
 - セッション, 17-2
 - データベース環境, 17-2
 - 動的 SQL, 17-8
 - トランザクション, 17-10
 - 複合文, 12-21
 - 明示的, 17-5
 - 接続での休止中のセッション, 17-3
 - 接続での現行セッション, 17-3
 - セミコロン (;)
 - SQL モジュール, 3-12, 4-14, 4-15
 - 複合文, 6-4
 - プリコンパイル SQL, 6-6
- 宣言
 - カーソル, 18-3
 - ホスト言語パラメータ, 8-2
- 宣言のコピー
 - FROM パス名句を参照
 - INCLUDE 文、FROM パス名句を参照
 - ホスト言語文の使用, 8-3
- 選択式

FOR 文, 12-10
カーソル用, 18-3
選択リスト
外部関数, 14-30

そ

挿入専用カーソル, 18-8
list, 18-8

た

第三世代言語 (3GL), 3-2
タイム・スタンプ, 19-9
対話型 SQL
EXIT 文, 16-44
SET FLAGS 文, 12-18
完了ステータス, 16-44
複合文, 12-3
複合文のラベル, 12-12
プログラム内の文のテスト, 2-4
単純文のプロシージャ, 4-2, 4-14
ストアド・ファンクションを起動, 13-12
ストアド・プロシージャを起動, 13-10

ち

抽出されたデータベース, 16-9

つ

通貨文字, A-2
通信領域
SQLCA ステータス・パラメータを参照
通知ルーチン, 14-41

て

定義者権限モジュール, 13-4
定数
リテラルを参照
データ型
\$SQL_VARCHAR, 6-17, 8-28
CHAR, 4-17, 8-26
C 内の整数, 8-29
C の SMALLINT, 8-29
DATE, 8-7, 8-22

DATE ANSI, 9-1, 9-3, 9-4, 9-6, 9-8, 9-13
SQL プリコンパイラ, 9-5
フォーマット, 9-3
DATE VMS, 9-13, 9-15
フォーマット, 9-14
INTERVAL, 9-2, 9-4, 9-12, 9-18
SQL プリコンパイラ, 9-5
精度, 9-2
LIST OF BYTE VARYING, 18-6
SQL モジュール, 4-10
SQL モジュール言語でのパラメータの受渡し, 8-26
TIME, 9-5
TIMESTAMP, 9-2, 9-5, 9-10
VARCHAR, 6-17
ストアド・ファンクション定義, 13-8
ストアド・プロシージャ定義, 13-6
テキスト文字列, 8-22
パラメータ用, 8-5
SQL モジュール, 4-10, 4-11
日付時刻データ型を参照
含む
ソース・プログラムへのテキスト・ファイル,
8-11
リポジトリの RDB\$RELATIONS ディレクトリ内
のテーブル, 8-9
浮動小数点, 6-17, 8-6, 8-28
プログラムの処理
Ada, 8-27
C, 8-27
SQL モジュール, 8-31
変換, 8-7, 8-26
10 進, 8-26
データ構造
SQLCA (SQL 通信領域), 10-7
SQLDA2 (SQL 記述子領域 2), 11-9
SQLDA (SQL 記述子領域), 11-9
データ操作言語 (DML)
SQL, 1-1
データ定義言語 (DDL)
EXECUTE IMMEDIATE 文, 11-8
SQL, 1-1
SQL モジュール
プロシージャ・パラメータ, 4-7
埋込み SQL, 6-5
モジュール言語, 4-16
データなし条件
SQLCODE を使用して検出, 10-8

- SQLSTATE を使用して検出, 10-6
- データの格納, 19-2
 - 日付時刻データ型, 9-2
- データの検索
 - FETCH 文
 - 行検索の順序, 18-4
 - 使い方
 - 一意なインデックス, 16-31
 - インジケータ・パラメータ, 8-21
 - カーソル, 18-1
 - 重複を許すインデックス, 16-32
 - ホスト言語パラメータ, 8-12
 - プログラム, 18-6
 - ロッキング, 16-28
- データの更新, 19-5, 19-6
 - 間で競合, 16-28
 - 同じ列値セットを持つ 2 行以上の行, 19-5
 - 行のセットから一度に 1 行, 19-6
 - 単一行, 19-5
 - 複数行, 18-5, 19-5
 - カーソルの使用, 18-6
 - プログラム, 19-6
- データベース
 - 一度に複数を使用, 15-12, 15-15
 - トランザクションを宣言する場合, 16-3
 - 一貫性, 16-2
 - エイリアス, 15-12
 - 行の挿入, 19-2
 - コンテキスト, 15-1
 - コンテキスト・ファイルの宣言を含む, 6-12
 - 整合性エラー, 10-31
 - 接続, 17-2
 - 接続解除, 15-15
 - 接続する, 15-1, 15-2
 - 2 つ以上, 15-12
 - データベース・ファイルのみへのアクセスを使用, 15-1
 - リポジトリ・アクセスの使用, 15-2
 - リモート, 15-4
 - 接続の失敗, 10-35
 - 抽出, 16-9
 - 定義の指定, 15-3
 - データのロード, 19-2
 - データをロード
 - データ・ファイルの使用, 19-2
 - デフォルト
 - Oracle Rdb での命名, 20-1
 - 非一貫性, 16-25
 - マルチスキーマ, 20-1, 20-10
 - プログラミング, 20-3, 20-10
 - 論理名, 15-2
- データベース環境
 - 埋込み言語プログラム, 17-3
 - エイリアスの重複, 17-7
 - 作成, 17-4
 - 重複, 17-6
 - 接続, 17-2
 - 定義, 17-2, 17-3
 - デフォルト, 17-2, 17-3
 - モジュール言語プログラム, 17-3
- データベース・コールバック
 - 外部ルーチン, 14-16
- データベースの整合性
 - エラー, 10-31
- データベースの接続解除, 15-15
 - DISCONNECT 文, 15-15
- データベースのロード, 19-2
 - サンプルのオンライン・プログラム, 19-2
- データベースへの接続, 15-2
 - 2 つ以上, 15-12
 - ATTACH 文を参照
 - SQL\$DATABASE 論理名, 15-2, 15-13
 - SQL_DATABASE 構成パラメータ, 15-2, 15-13
 - 障害, 10-35
 - 接続, 17-5
 - プログラム, 17-4
- データベースへのバインド
 - データベースへの接続を参照
- テーブル
 - アクセスの競合, 16-28
 - 行の挿入, 19-2
 - 更新, 19-5
 - プログラムの使用, 19-6
 - 構造化, 19-4
 - 削除, 19-4
 - 日付時刻データ型, 9-2
 - 予約
 - 暗黙的, 16-10
 - 明示的, 16-10
 - 列に NULL 値を割り当てる, 8-18
 - 列の NULL 値を識別する, 8-18
 - 列を削除, 19-8
 - ロック, 16-25
- テーブル・カーソル, 18-6

- テーブルの組合せ
 - テーブルに値を保管, 19-3
- テーブルの宣言
 - DECLARE TABLE 文を参照
- デッドロック
 - エラー, 10-32, 10-34
 - 外部ルーチン, 14-45
- デッドロック・エラー, 10-32, 10-34
 - 分散トランザクションの回避, 15-11
 - リカバリ, 10-34
- デバッグ
 - 複合文, 12-18
- デフォルトの評価依存タイプ, 13-17
- デフォルトの予約依存タイプ, 13-17
- デリミタ付き識別子, 20-9
- 転送機能
 - 移植性を参照
- 転送ベクトル, 7-5, 7-9

と

- 動的 DECLARE CURSOR 文, 11-4, 11-19, 11-23,
 - 18-7, 18-17
 - SQL モジュール, 3-11
- 動的 SQL, 11-1
 - CAST およびパラメータ・マーカー, 9-17
 - CLOSE 文, 11-35
 - DESCRIBE 文, 11-4, 11-33, 11-35
 - EXECUTE IMMEDIATE 文, 11-4, 11-6
 - EXECUTE 文, 11-4, 11-32, 11-33
 - FETCH 文, 11-4, 11-35
 - OPEN 文, 11-4
 - PREPARE 文, 11-4, 11-32, 11-33, 11-35
 - RELEASE 文, 11-4
 - SELECT 文, 11-3, 11-19, 11-22, 11-32, 11-35
 - SQLDA, 11-9
 - SQLDA2, 11-9
 - 概要, 11-2
 - 拡張動的カーソル, 11-4, 11-19, 11-23, 11-32,
 - 11-35
 - 処理, 11-5
 - 接続, 17-8
 - 使い方
 - パラメータ, 11-32, 11-35
 - 動的カーソル, 11-4, 11-19, 11-23
 - 動的に実行される文のカテゴリ, 11-2
 - パラメータ
 - カーソル名, 11-32, 11-35
 - 実行時に提供, 11-32
 - 文名, 11-32, 11-35
 - パラメータの宣言
 - カーソル名, 11-33
 - 動的に実行された文, 8-4
 - 文に対する名前, 11-33
 - パラメータ・マーカー, 11-2, 11-8
 - 非 SELECT 文, 11-2, 11-32, 11-33
 - パラメータ・マーカー, 11-8
 - パラメータ・マーカーなし, 11-5
 - 日付時刻データ型の注意事項, 9-17, 9-19
 - 複合文, 12-18
 - 複数文の手続き, 12-18
 - 文, 11-2
 - 動的に実行, 11-2
 - パラメータ・マーカー, 11-13
 - リスト項目の選択, 11-3, 11-4, 11-19
- 動的 SQL の選択リスト項目, 11-3, 11-4, 11-19
 - DESCRIBE 文, 11-4, 11-8, 11-19
- 動的カーソル, 11-4, 11-19, 11-23, 18-7, 18-17
- トラブルシューティング
 - DISTRIBTRAN 権限, 15-11
 - 外部ルーチン, 14-44
 - デッドロック, 15-11
 - リモート・アクセス, 15-11
- トランザクション, 16-2
 - 2 フェーズ・コミット・プロトコルの使用, 16-24,
 - 16-25
 - Recovery-time ジャーナル・ファイルの更新, 16-43
 - 開始, 16-2
 - 開始の失敗, 10-35
 - 外部ルーチン, 14-3
 - 管理, 16-33
 - コミット, 16-42, 16-43, 16-44
 - コンテキスト, 16-2
 - コンテキスト・ファイルの宣言を含む, 6-12
 - 終了, 16-42, 16-44
 - CompaqTru64 UNIX, 16-44
 - 異常条件下, 16-44
 - スコープ, 16-21, 16-22, 16-23
 - 制約, 16-12, 16-27
 - 接続とともに使用, 17-10
 - 定義, 16-2
 - データベースの非一貫性を回避, 16-2
 - デフォルト, 16-10
 - デフォルトの特性, 16-21

特性

- 指定, 16-2, 16-5
- 設定, 16-22

特性を指定, 16-2

- トリガー, 16-12, 16-27
- バッチ更新, 16-8, 16-44

ビュー, 16-12

不完全, 16-2

複合文, 12-17, 12-21

複数のデータベースを含む, 16-21

分散

- 分散トランザクションを参照

前の特性を上書きする, 16-22

読取り / 書込み, 16-8, 16-12

読取り専用, 16-6, 16-30

- ロック競合エラー, 16-30

列による計算, 16-12

ロールバック, 16-42, 16-43, 16-44

ロックの解放を待つ, 16-12

トランザクションの依存関係, 13-17

トランザクションの終了

COMMIT 文, 16-42

ROLLBACK 文, 16-42

トランザクションの宣言

DECLARE TRANSACTION 文 DECLARE

TRANSACTION 文, SET TRANSACTION 文,

コンテキスト・ファイルを参照

トランザクション文の ON 句, 16-21

トリガー, 19-9

外部関数, 14-31

トランザクション, 16-12, 16-27

ドル記号 (\$)

- Ada プログラムでは、かわりにアンダースコアを使用, 8-27

ドル記号の代わりにアンダースコアを使用, 8-27

な

名前

Ada で一意, 6-14

SQL モジュール, 3-6

SQL モジュールのプロシージャ, 4-3

- 大文字と小文字の区別, 4-4

SQL モジュールのプロシージャ・パラメータ, 4-10

複合文, 12-12

に

二重ハイフン (--), 3-6

認可識別子, 15-12

SQL プリコンパイラでのデフォルト, 20-7

SQL モジュール, 3-10

SQL モジュールでのデフォルト設定, 20-3

デフォルト・スキーマ, 20-3

認証

- リモート・アクセス, 15-5, 15-6, 15-7, 15-8

ね

ネストした複合文, 12-4, 12-6, 12-11

ラベル, 12-12

の

ノン・ストアド・モジュール, 13-2

は

倍精度浮動小数, 6-17

配置

SQLCA ステータス・パラメータ

C 言語, 8-28

SQLDA ステータス・パラメータ, 11-11

バイナリ・データ

C 言語, 4-17, 6-17, 8-28

- (ハイフン)

ハイフン (-) を参照

ハイフン (-)

アンダースコア (_) との関連, 6-5

二重ハイフン (--)-SQL モジュール内のコメント・フ ラグ, 3-6

パスワード

リモート・アクセスへの提供, 15-6

バッチ更新トランザクション, 16-2, 16-8, 16-9, 16-44

パフォーマンス

改善

INSERT の間, 19-4

SQL プリコンパイラ, 6-10

SQL モジュール・プロセッサ, 5-4

SQL モジュール・プロセッサでのコンパイル時 間, 5-4

冗長なテーブルの作成, 19-3

- ストアド・ルーチン, 13-2
- 複合文, 12-2
- 複合文の使用, 12-2
- パフォーマンスの低下
 - 他がインデックスを使用しなければ, 16-28
- パラメータ, 8-1
 - NULL 値の処理, 8-4, 8-18
 - sql_get_error_text ルーチン用, 10-28
 - SQL プリコンパイラの取扱い, 6-5
 - SQL モジュールのプロシージャへのコール, 4-12, 4-13
 - SQL モジュール・プロシージャ, 4-4, 4-5, 4-13
 - SQL モジュールへの受渡し, 4-5
 - インジケータ, 8-4, 13-6, 13-8, 18-6
 - 受渡しメカニズム, 4-13
 - エラー処理, 8-4
 - 検索行, 8-12
 - 宣言, 8-3, 8-5, 8-27
 - Ada, 8-27
 - COBOL, 8-30
 - C 言語, 8-27, 8-28
 - FORTRAN, 8-30
 - FROM 句の使用, 8-11
 - INCLUDE 文の使用, 8-8
 - Pascal, 8-31
 - PL/I, 8-31
 - SQL モジュール・プロシージャ, 4-5, 4-6, 8-31
 - ホスト言語, 8-2
 - ホスト言語プログラム, 8-2, 8-12
- データ型, 8-5
 - SQL モジュール, 4-10
- データベース接続への使用, 15-4
- 動的 SQL, 11-32, 11-35
 - カーソル名, 11-32, 11-35
 - 宣言, 11-33
 - 文名, 11-32, 11-35
- 動的に実行された文, 8-4
- 名前, 8-12
 - SQL モジュール・プロシージャ, 8-26
- 浮動小数点, 8-6
- プリコンパイル・プログラム, 8-12
 - アンダースコア, 6-5
 - 大文字と小文字の区別, 6-5
 - ハイフン (-), 6-5
- 変数と比較, 8-3
- ホスト言語変数、インジケータ・パラメータ、構造体を参照

- マーカー, 11-2
- メイン, 8-3, 8-12, 8-13
 - 使用, 8-15
 - 宣言, 8-13
 - よくある使用誤り, 8-24
 - 列値の保存, 8-3
- パラメータの受渡しメカニズム
 - SQL モジュール, 4-13, 8-26
 - BY DESCRIPTOR CHECK, 4-14
 - CHECK オプション, 4-14
 - デフォルトの上書き, 4-13
 - プロシージャへのコール, 4-12
- パラメータの宣言
 - 実行時エラーの原因, 4-12
 - モジュール・プロシージャ, 4-4, 4-5, 4-8, 4-10
- パラメータ・マーカー, 11-2, 11-8, 11-17
 - CAST 関数, 9-17
 - DESCRIBE 文, 11-8, 11-19
 - データ型, 11-9, 11-10
 - 日付時刻データ型, 9-17
 - ホスト言語変数との比較, 11-2, 11-8
- パラメータ・モード
 - ストアド・ファンクション, 13-8
 - ストアド・プロシージャ, 13-6

ひ

- 日付
 - カレントの保存, 19-9
 - フォーマットの変更, A-2
- 日付時刻データ型
 - ANSI/ISO SQL 規格, 9-1
 - DATE, 9-3
 - DATE ANSI, 9-3, 9-4, 9-6, 9-8, 9-13
 - SQL プリコンパイラ, 9-5
 - フォーマット, 9-3
 - DATE VMS, 9-13, 9-14, 9-15
 - CompaqTru64 UNIX, 9-15
 - 動的 SQL での制限, 9-19
 - フォーマット, 9-14
 - INSERT 文, 9-3
 - INTERVAL, 9-2, 9-4, 9-12, 9-18
 - SQL プリコンパイラ, 9-5
 - 精度, 9-2
 - SQL 定義, 9-5
 - SQL プリコンパイラ, 9-5

- SQL モジュール・プロセッサ, 9-8
- TIME
 - SQL プリコンパイラ, 9-5
- TIMESTAMP, 9-2, 9-10
 - SQL プリコンパイラ, 9-5
- 移植性について, 9-12
- 埋込み SQL での宣言, 9-6
- 代入エラー, 9-2
- データの保管, 9-2
- 動的 SQL, 9-17
- トリガー, 19-9
- パラメータとしての受渡し, 8-7
- フォーマット, 16 文字, 9-14
- プログラムで使用, 9-5
- プログラムで日付時刻書式を変換, 9-4
- プログラムで文字列を変換, 9-4
- プログラムの移植性, 9-12
- 変換, 9-3, 9-4, 9-13
- 日付時刻データ型での移植性, 9-12
- 非同期システム・トラップ (AST), 15-11
- 一つのデータベースからのデータに制限された結合, 15-14
- ビュー
 - カーソルとの比較, 18-5
 - トランザクション, 16-12
 - 比較
 - カーソル, 18-5
 - クエリー用に作成されたテーブル, 19-3
- 表現形式
 - 設定, 3-7, 6-12

ふ

- ファイル・アクセス
 - 指定, 15-1
- ファントム行, 16-14
 - READ COMMITTED, 16-17
 - REPEATABLE READ, 16-15
- 複合文, 12-1, 12-2, 12-12
 - CALL 文, 13-10
 - SQL モジュール, 4-15, 12-7, 12-11, 12-17
 - アトム性, 12-15
 - 埋込み SQL, 6-4, 12-7
 - エラー処理, 10-6
 - カーソル, 12-10
 - 完了条件, 12-23
 - 作成, 12-3
 - 終了, 12-13
 - ストアド・ファンクション, 13-7, 13-8
 - ストアド・ファンクションを起動, 13-12
 - ストアド・プロシージャ, 13-5, 13-7
 - ストアド・プロシージャを起動, 13-10
 - 対話型 SQL, 12-3
 - 動的 SQL, 12-18
 - ネスト, 12-4, 12-6, 12-11
 - アトム性, 12-15
 - 変数宣言, 12-6
 - ラベル, 12-12
 - パフォーマンス, 12-2
 - 例外処理, 12-22
 - 複合文内の SET 代入文, 12-4
 - 複合文のデバッグ, 12-18
 - 複数行の文
 - プリコンパイル・プログラム, 6-4
 - 複数行文の継続
 - プリコンパイル・プログラム, 6-4
 - 複数行リテラル
 - COBOL, 6-18
 - SQL モジュール, 4-16
 - プログラム, 6-4
 - 複数行リテラルの継続
 - SQL モジュール, 4-16
 - プログラム, 6-4
 - 複数国のキャラクタ・セット, A-1
 - 述語における動作, A-4
 - 複数のオブジェクト・ファイル
 - リンク, 7-11
 - 複数バージョン
 - 接続, 15-9, 15-10
 - リモート・アクセス, 15-9
 - 複数文プロシージャ, 4-2, 4-15, 12-1
 - パフォーマンス, 12-2
 - 浮動小数点データ, 8-6
 - C 言語, 6-17
 - 倍精度, 6-17
 - プリコンパイラ
 - SQL プリコンパイラを参照
 - プリコンパイラを起動するシンボル (DCL), 6-6
 - プリコンパイル SQL
 - Ada 固有の要件, 6-14
 - COBOL 固有の要件, 6-18
 - FORTTRAN 固有の要件, 6-20
 - Pascal 固有の要件, 6-22
 - PL/I 固有の要件, 6-23

- SQL プリコンパイラを参照
- プリコンパイル・プログラムのキーワード, 6-5
- プリコンパイル・プログラムの複数行文, 6-4
- フロー制御文, 12-2
 - CASE, 12-8
 - FOR, 12-10
 - IF, 12-6
 - LEAVE, 12-13
 - LOOP, 12-9
 - WHILE ループ, 12-9
- プロキシ・アカウント, 15-8
 - 複数バージョン, 15-10
- プログラム
 - DEBUG 修飾子とのコンパイル, 7-13
 - SQL 文の埋込み
 - Ada, 6-14
 - COBOL, 6-18
 - C 言語, 6-16
 - FORTRAN, 6-20
 - Pascal, 6-22
 - PL/I, 6-23
 - SQL 文を含める, 1-3
 - SQL モジュール, 3-1, 3-4, 4-1, 5-1
 - インジケータ・パラメータ, 8-18
 - 埋込み UPDATE 文, 19-6
 - エラー・メッセージの表示, 10-27
 - エラー・リカバリ
 - デッドロック, 10-34
 - ロックの競合, 10-32
 - カーソル, 18-5
 - カーソルの使用, 18-5
 - 開発, 2-1
 - 開発のガイドライン, 2-1
 - コールされる SQL モジュールの作成, 3-4
 - サンプル, 3-14
 - sql_all_datatypes, 8-27
 - sql_all_datatypes_ada.sqlmod, 8-31
 - sql_all_datatypes.sc, 8-27
 - sql_dynamic, 11-39
 - sql_multi_stmt_dyn.sqlada, 11-39
 - sql_report, 18-6
 - sql_terminate, 10-30, 19-6
 - エラー処理の説明, 10-4
 - カーソルの使用, 19-6
 - カーソルの使用の説明, 18-6
 - 拡張動的カーソル, 18-18
 - スクロール可能なリスト・カーソルの説明, 18-15
 - データ・ファイルからのデータベースのロード, 19-3
 - 動的カーソルの説明, 18-17
 - 日付時刻データ型, 9-5
 - リスト・カーソルの説明, 19-5
- 実行, 6-2, 7-12
- 実行可能イメージの作成, 7-1
- 実行時エラー, 10-2
- 終了, 16-44
- 使用する SQL モジュールの処理, 5-3
- 処理, 6-2
- 接続, 17-1, 17-11, 17-13
- データベース
 - 接続, 15-3
 - 接続解除, 15-15
- デバッグ, 2-4, 7-13
- パラメータ宣言
 - Ada, 8-27
 - COBOL, 8-30
 - C 言語, 8-27
 - FORTRAN, 8-30
 - Pascal, 8-31
 - PL/I, 8-31
 - SQL モジュール, 4-5, 8-31
- 日付時刻データ型, 9-3, 9-19
- ファイルまたはライブラリを含める場合のホスト言語制限, 8-8
- 複数のモジュールの結合, 3-11
- プリコンパイル, 6-2, 6-4
 - SQL 文, 6-5
 - SQL 文の検出, 6-4
 - ハイフンとアンダースコアの取扱い, 6-5
- プリコンパイル SQL の処理, 6-2
- ホスト言語による妥当性チェック, 10-31
- マルチスキーマ・データベース, 20-1
- メイン・パラメータ, 8-12
- リンク, 7-1
 - 複数のモジュール, 7-11
- プログラムの移植性
 - 移植性を参照
- プログラムの開発
 - ガイドライン, 2-1
- プログラムのセクション
 - 属性, 7-6
 - 定義, 7-6

- データベース・エイリアス, 7-7, 7-9
- データベース・オブジェクト, 7-6
- 長さが変わった場合, 7-9
- プログラムのデバッグ, 7-13
 - DEBUG 修飾子, 7-13
 - EDIT コマンドの使用, 7-14
 - エラーも参照
- 対話型 SQL の使用, 2-4
- プロシージャ
 - SQL モジュール, 3-12, 4-2, 4-3, 4-4, 4-10
 - SQL モジュールのプロシージャを参照
 - 外部, 14-2, 14-45
 - 外部プロシージャ、外部ルーチンを参照
 - ストアド, 13-1
 - ストアド・プロシージャ、ストアド・ルーチンを参照
- プロシージャの依存関係, 13-15
- プロシージャ・パラメータ
 - コール・パラメータへの関連, 4-9
- プロシージャ・パラメータのデータ型の指定, 4-10
- ブロック構造
 - Ada, 6-14
 - COBOL, 6-18
 - C 言語, 6-17
 - FORTRAN, 6-21
 - Pascal, 6-22
 - PL/I, 6-23
- 分散トランザクション
 - 終了, 16-42
 - 定義, 16-24
 - リモート・ノード上のデータベースへのアクセス, 15-11
- 分離レベル, 16-3
 - Oracle Rdb データベース以外の指定, 16-20
 - READ COMMITTED, 16-17
 - REPEATABLE READ, 16-15
 - RESERVING 句, 16-15
 - SERIALIZABLE, 16-3, 16-15
 - 型, 16-14
 - 許可されている現象, 16-15
 - 更新専用カーソル, 16-15
 - 更新用, 16-20
 - 利点, 16-19
 - レポート用, 16-20

へ

- ヘッダー・ファイル
 - sql_rdb_headers.h, 6-17, 10-30
 - sql_sqlda.h, 8-5, 11-10
- 変換イメージ, 14-43
- 変数
 - Pascal での宣言, 6-23
 - パラメータとの比較, 8-3
 - 複合文での宣言, 12-4
 - 複合文内への割当て, 12-4
- 変数の割当て
 - 複合文複合文, 12-4

ほ

- ポインタ変数, C プリコンパイラでの使用, 6-17
- 保護
 - ロック・タイプ、共有モード、テーブルを参照
- 保持可能カーソル, 18-8, 18-11
- ホスト言語
 - SQL プリコンパイラによるサポート, 6-13
 - SQL モジュール言語に対するサポート, 3-2
 - 外部ルーチンに対するサポート, 14-38
- ホスト言語パラメータ
 - 検索行, 8-12
 - ファイルからのコピー, 8-8, 8-12
 - 割当て値, 8-12
- ホスト言語パラメータの宣言, 8-2
 - NULL 値の扱い, 8-4
 - エラー処理, 8-4
 - 動的に実行された文, 8-4
 - 列値の保存, 8-3
- ホスト言語プログラム
 - 開発, 2-1
- ホスト言語プログラムの埋込み SQL 文, 1-3
- ホスト言語変数
 - 言語文による操作, 18-9
 - パラメータ、SQL モジュール・プロシージャ内を参照
 - パラメータ・マーカーとの比較, 11-2, 11-8
 - 1 行形式の選択文との使用, 18-6

ま

- マルチスキーマ
 - 属性, 20-2

- プログラミング
 - SQL プリコンパイラ, 20-6, 20-10
 - SQL モジュール・プロセッサ, 20-3, 20-6
- マルチスキーマ・データベース, 20-1
- SQL プリコンパイラで無効にする, 20-9
- カタログ内, 20-2
- 構造, 20-2
- スキーマ・オブジェクト, 20-2
- 属性, 20-2
- 定義, 20-2
- デフォルト
 - カタログ, 20-2, 20-7
 - スキーマ, 20-2, 20-8
- 内のスキーマ, 20-2
- プログラミング, 20-6
 - SQL プリコンパイラ, 20-6, 20-10
 - SQL モジュール・プロセッサ, 20-3, 20-6
 - コンパイル, 20-9
 - サンプル C プログラム, 20-4, 20-6
 - サンプル SQL モジュール, 20-4
 - リンク, 20-10
- 無効化
 - SQL プリコンパイラ, 20-9
- 命名, 20-1
 - SQL モジュール, 20-6
 - プリコンパイル SQL, 20-8
- 有効化, 20-2
- マルチユーザーの競合, 10-32
- アクセスの競合, 16-28
- プログラムの処理, 10-34

む

- 無限ループ
 - WHENEVER 文とエラー・ハンドラ間, 10-12
- 無効性
 - ストアド・ルーチン, 13-18
 - SQL 文が発生, 13-18

め

- 構造を参照
- メイン・パラメータ
 - パラメータ
- メッセージの表示
 - sql_get_error_text ルーチン, 10-28
 - sql_signal ルーチン, 10-27

- ユーザー指定のメッセージ・ファイル, 10-30
- メッセージ・ファイル, ユーザー定義, 10-30
- Multivendor 統合アーキテクチャ (MIA) を参照
- メッセージ・ファイル
 - ユーザー定義
 - sql\$persmsg の例, 10-30
 - 実行時, 10-30
- メモリー
 - 共有
 - 外部ルーチン, 14-34, 14-44

も

- モード
 - 共有, 16-28
 - EXCLUSIVE, 16-10, 16-27, 16-28
 - PROTECTED, 16-10, 16-27, 16-28
 - SHARED, 16-10, 16-28
 - トランザクション
 - バッチ更新, 16-44
 - 読取り / 書込み, 16-8
 - 読取り専用, 16-6
 - 予約, 16-28
- モジュール
 - SQL モジュールを参照
 - アプリケーションの構築, 7-11
 - 保存
 - 作成, 13-5, 13-7
- SQL モジュールを参照
- モジュール言語
 - SQL モジュール言語を参照
- モジュール・プロセッサ
 - SQL モジュール・プロセッサを参照
- モジュール言語
 - SQL モジュール言語
- 文字列
 - C の NULL 終了, 4-17, 6-16, 8-9, 8-12, 8-28, 11-20
- 文字列の切捨て, 8-22

ゆ

- ユーザー定義外部関数
 - 記述, 14-12
- ユーザー認証, 15-5, 15-6, 15-7, 15-8
- ユーザー名
 - リモート・アクセスへの提供, 15-6

よ

読取り / 書込みトランザクション, 16-2, 16-8, 16-12
RESERVING 句, 16-12
読取り専用カーソル
テーブル, 18-7
リスト, 18-8
読取り専用記憶領域
アクセス, 16-7, 16-30
読取り専用トランザクション, 16-2, 16-6
EXCLUSIVE 共有モード, 16-30
スナップショットが有効な場合行ロック, 16-28
スナップショット・ファイル, 16-30
ロック競合エラー, 16-30

ら

ライブラリ
Ada, 6-15
SQL, 7-2, 7-11
ラベル
FOR 文, 12-13
LOOP 文用, 12-13
複合文, 12-12
ラベルの開始
LOOP 文用, 12-13
FOR 文用, 12-13
複合文, 12-12

り

リスト
カーソル, 18-13
削除, 19-9
スクロール可能, 18-15
挿入, 18-13, 19-4, 19-5
定義, 18-6
読込み, 18-13
リスト・カーソル, 18-6, 18-13, 19-4
位置付け, 18-13
行の挿入, 19-5
スクロール可能, 18-15
挿入専用, 18-8, 19-5
読取り専用, 18-8
リテラル
Ada, 6-14
SQL モジュール, 4-16

プログラムの複数行リテラル, 6-4
リポジトリ
NULL 終了文字列および C の使用, 8-12
定義のコピー
FROM 句の使用, 8-11
INCLUDE 文の使用, 8-8, 8-9
データベース接続用に指定したアクセス, 15-2
リポジトリ定義, 15-3
リポジトリパス名の RDB\$RELATIONS ノード, 8-9
リモート・アクセス
DECnet, 15-6
TCP/IP, 15-6, 15-10
デフォルトのアカウントの使用, 15-9
トラブルシューティング, 15-11
認証ユーザー, 15-5
プロキシ・アカウント, 15-8
方法, 15-4, 15-9
リモート・サーバー・アカウント, 15-5
リモート・データベース
ブリコンパイラのパフォーマンスを改善, 6-10
リモート・ノード
データベースの宣言, 15-4
リモート・ユーザー認証, 15-5, 15-6, 15-7, 15-8
リンク, 7-1
LNK\$LIBRARY 論理名, 7-3
SQL\$USER ライブラリ・ファイル, 7-3
SQL モジュール, 5-3
SQL ライブラリ, 5-3, 7-11
オブジェクト・ファイル
CompaqTru64 UNIX, 7-10
OpenVMS, 7-2
共有可能なイメージ, 7-5
発生するエラー
一意でないプロシージャ名, 4-3
一意でないモジュール名, 3-6
複数のオブジェクト・ファイル, 7-11
複数のモジュール, 7-11
ユーザー指定のメッセージ・ファイル, 7-2

る

ルーチン
sql_close_cursors, 18-4
sql_deregister_error_handler, 10-19
sql_get_error_handler, 10-19
sql_get_error_text, 10-28
sql_get_message_vector, 10-13, 10-14

- sql_register_error_handler, 10-18
- SQL ルーチンを参照
 - 外部, 14-2, 14-45
 - ストアド, 13-1
- ループ
 - データを検索, 18-9

れ

- 例
 - オンライン・プログラム, 1-5
 - サンプル・データベース, 1-5
- 例外条件, 10-2, 10-4
 - エラー・コード、エラー処理を参照
 - 検出, 10-4
 - SQLCODE の使用, 10-8
 - SQLSTATE の使用, 10-5
 - WHENEVER 文の使用, 10-10
 - 処理, 10-2
 - アトム性, 12-15
 - 複合文, 12-22
 - データなし, 18-9
- 例外処理
 - エラー処理、例外処理を参照
- レコード
 - 行、構造体を参照
 - 構造体との比較, 8-3
- 列
 - インジケータ・パラメータを使用, 8-18
 - 行、テーブルを参照
- 列制約
 - 制約を参照
- 列による計算
 - トランザクション, 16-12
- レポートの記述
 - PROTECTED 共有モードの使用, 16-7
 - テーブルの作成, 19-4
 - トランザクション・タイプの選択, 16-7
- 連結
 - 日付時刻データ型, 9-17

ろ

- ログ・ファイル
 - コンパイル時エラー

- SQL プリコンパイラを使用する場合, 6-10
- プリコンパイル・エラー, 6-10
- ロケール設定, A-2
- ロッキング
 - READ COMMITTED 分離レベル, 16-19
 - REPEATABLE READ 分離レベル, 16-17
 - SERIALIZABLE 分離レベル, 16-15
 - 意図的ロック, 16-26
 - インデックス, 16-31
 - 重複可能, 16-32
 - 使用しない, 16-28
 - オプション
 - アクセスの競合, 16-28
 - カーソル, 16-29, 18-11, 18-12
 - 行, 16-25
 - 競合の削減, 16-31
 - ストアド・インデックス, 16-31
 - ストラテジ, 16-25
 - 定義, 16-25
 - テーブル, 16-10, 16-25
 - リソースを待機, 16-27
 - ロックの解放を待機, 16-12
 - ロックの競合
 - no-wait 特性, 16-28
 - 定義, 16-28
- ロック, 16-12
 - 解放, 16-29
 - 行, 16-25
 - 競合エラー, 10-32
 - 読取り専用トランザクション・モード, 16-30
 - テーブル, 16-25
- ロック競合エラー, 10-32
 - 読取り専用トランザクション, 16-30
 - リカバリ, 10-32
- ロック・タイプ, 16-10
 - READ, 16-28
 - WRITE, 16-28
 - 他のユーザーの影響, 16-27
 - どのように決定されたか, 16-27
- ロックの競合
 - no-wait 特性, 16-28
 - アクセスの競合, 16-28
 - 外部ルーチン, 14-45
 - 定義, 16-28
- 論理名
 - ATTACH 文, 15-13
 - LNK\$LIBRARY, 7-2

RDB\$ROUTINES, 14-5, 14-7
RDBSERVER, 15-9
RDMS\$DEBUG_FLAGS, 12-18
RDMS\$DEBUG_FLAGS_OUTPUT, 12-19
RDMS\$RTX_SHRMEM_PAGE_CNT, 14-34, 14-44
RDMS\$VALIDATE_ROUTINE, 13-21
RDMS\$VERSION_VARIANT, 15-9
SQL\$DATABASE, 15-2
外部ルーチン, 14-5, 14-7, 14-25, 14-34
データベース, 15-2