

PL/SQL

ユーザーズ・ガイドおよびリファレンス

リリース 8.1

ORACLE®

PL/SQL ユーザーズ・ガイドおよびリファレンス リリース 8.1

部品番号 : A62738-1

第 1 版 : 1999 年 5 月 (第 1 刷)

原本名 : PL/SQL User's Guide and Reference, Release 8.1.5

原本部品番号 : A67842-01

原本著者 : Tom Portfolio

グラフィック・デザイナー : Valarie Moore

原本協力者 : Dave Alpern, Chandrasekharan Iyer, Ervan Darnell, Ken Jacobs, Sanjay Kaluskar, Sanjay Krishnamurthy, Janaki Krishnaswamy, Neil Le, Kannan Muthukkaruppan, Shirish Puranik, Chris Racicot, Ken Rudin, Usha Sangam, Ajay Sethi, Guhan Viswanathan

Copyright © 1999, 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 including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, 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.....	xvii
1 概要	
主な特徴	1-2
ブロック構造.....	1-2
変数と定数.....	1-3
カーソル.....	1-4
カーソル FOR ループ.....	1-5
カーソル変数.....	1-6
属性.....	1-6
制御構造.....	1-8
モジュール性.....	1-11
データの抽象化.....	1-12
情報隠ぺい.....	1-14
エラー処理.....	1-15
アーキテクチャ	1-16
Oracle Server.....	1-17
Oracle Tools.....	1-18
PL/SQL の利点	1-19
SQL のサポート.....	1-19
オブジェクト指向プログラミングのサポート.....	1-20
パフォーマンスの向上.....	1-20
高い生産性.....	1-21

完全な移植性.....	1-22
Oracle との緊密な統合	1-22
セキュリティ	1-22

2 基礎

キャラクタ・セット	2-2
字句単位	2-2
デリミタ.....	2-3
識別子.....	2-4
リテラル.....	2-7
コメント.....	2-9
データ型	2-10
数値型.....	2-11
文字型.....	2-14
NLS 文字型.....	2-18
LOB 型.....	2-19
その他の型.....	2-21
ユーザー定義のサブタイプ	2-22
サブタイプの定義.....	2-22
サブタイプの使用.....	2-23
データ型の変換	2-25
明示的な変換.....	2-25
暗黙的な変換.....	2-25
暗黙的な変換と明示的な変換.....	2-26
DATE の値.....	2-27
RAW および LONG RAW の値	2-27
NLS 値.....	2-27
宣言	2-28
DEFAULT の使用方法.....	2-29
NOT NULL の使用方法.....	2-29
%TYPE の使用方法	2-30
%ROWTYPE の使用方法.....	2-30
制限.....	2-33

命名規則	2-33
シノニム.....	2-34
有効範囲.....	2-34
大文字 / 小文字の区別	2-34
ネーム変換.....	2-34
有効範囲と可視性	2-35
代入	2-38
ブール値.....	2-38
データベース値.....	2-39
式と比較	2-39
演算子の優先順位.....	2-40
論理演算子.....	2-41
比較演算子.....	2-42
連結演算子.....	2-44
ブール式.....	2-44
NULL の扱い	2-46
組込みファンクション	2-49

3 制御構造

概要	3-2
条件制御: IF 文	3-2
IF-THEN	3-3
IF-THEN-ELSE.....	3-3
IF-THEN-ELSIF	3-4
指針.....	3-5
反復制御: LOOP 文と EXIT 文	3-6
LOOP.....	3-6
WHILE-LOOP.....	3-8
FOR-LOOP	3-9
順次制御: GOTO 文と NULL 文	3-13
GOTO 文.....	3-13
NULL 文	3-16

4 コレクションとレコード

コレクションとは?	4-2
ネストした表について.....	4-2
ネストした表と索引付き表との相違点.....	4-3
varray について.....	4-4
varray とネストした表との相違点.....	4-4
コレクションの定義と宣言	4-5
コレクションの宣言.....	4-7
コレクションの初期化と参照	4-8
コレクション要素の参照.....	4-10
コレクションの代入と比較	4-11
コレクション全体の比較.....	4-12
コレクションの操作	4-13
ネストした表の例.....	4-13
varray の例.....	4-15
個々の要素の操作.....	4-17
ローカル・コレクションの操作.....	4-19
コレクション・メソッドの使用	4-20
EXISTS の使用.....	4-20
COUNT の使用.....	4-21
LIMIT の使用.....	4-21
FIRST および LAST の使用.....	4-21
PRIOR および NEXT の使用.....	4-22
EXTEND の使用.....	4-22
TRIM の使用.....	4-23
DELETE の使用.....	4-24
メソッドをコレクション・パラメータに適用する.....	4-25
コレクション例外の回避	4-26
バルク・バインドの利用	4-27
バルク・バインドによるパフォーマンスの向上.....	4-28
FORALL 文の使用.....	4-30
BULK COLLECT 句の使用.....	4-32
FORALL と BULK COLLECT の使用.....	4-33
ホスト配列の使用.....	4-34
カーソル属性の使用.....	4-34

レコードとは？.....	4-35
レコードの定義と宣言.....	4-36
レコードの宣言.....	4-37
レコードの初期化と参照.....	4-38
レコードの参照.....	4-38
レコードの代入と比較.....	4-40
レコードの比較.....	4-42
レコードの操作.....	4-42

5 Oracle の操作

SQL のサポート.....	5-2
データ操作.....	5-2
トランザクション制御.....	5-2
SQL ファンクション.....	5-3
SQL 疑似列.....	5-3
SQL 演算子.....	5-5
カーソル管理.....	5-6
明示カーソル.....	5-6
暗黙カーソル.....	5-10
パッケージ・カーソル.....	5-11
カーソル FOR ループの使用.....	5-12
副問合せの使用.....	5-13
別名の使用.....	5-13
パラメータ渡し.....	5-13
カーソル変数の使用.....	5-14
カーソル変数とは？.....	5-14
変数を使う理由.....	5-15
REF CURSOR 型の定義.....	5-15
カーソル変数の宣言.....	5-16
カーソル変数の制御.....	5-17
例 1.....	5-22
例 2.....	5-22
例 3.....	5-23
例 4.....	5-25
ネットワーク通信量の削減.....	5-27

エラーの回避.....	5-28
カーソル変数の制限.....	5-30
カーソル属性の使用.....	5-31
明示カーソルの属性.....	5-31
暗黙カーソルの属性.....	5-35
トランザクション処理.....	5-37
トランザクションがデータベースを保護する方法.....	5-38
COMMIT の使用	5-38
ROLLBACK の使用.....	5-39
SAVEPOINT の使用.....	5-40
暗黙的なロールバック.....	5-41
トランザクションの終了.....	5-41
SET TRANSACTION の使用.....	5-42
デフォルトのロックの上書き	5-43
サイズ制限への対応.....	5-46
自律型トランザクションの使用.....	5-48
自律型トランザクションの利点.....	5-48
自律型トランザクションの定義.....	5-49
自律型トランザクションの制御.....	5-52
例 1: 自律型トリガーの使用.....	5-54
例 2: 自律型ファンクションの SQL からのコール.....	5-55
パフォーマンスの向上.....	5-56
オブジェクト型およびコレクションの使用.....	5-56
バルク・バインドの使用.....	5-57
システム固有の動的 SQL の使用	5-58
外部ルーチンの使用.....	5-58
NOCOPY コンパイラ・ヒントの使用	5-59
RETURNING 句の使用	5-59
逐次再使用可能パッケージの使用.....	5-60
PLS_INTEGER データ型の使用.....	5-61
NOT NULL 制約の回避.....	5-61
条件制御文の句の入替え.....	5-62
暗黙のデータ型変換の回避.....	5-63
下位互換性の保証.....	5-63

6 エラー処理

概要.....	6-2
例外の利点.....	6-3
事前定義の例外.....	6-4
ユーザー定義の例外.....	6-7
例外の宣言.....	6-7
有効範囲の規則.....	6-7
EXCEPTION_INIT の使用.....	6-8
raise_application_error の使用.....	6-9
事前定義の例外の再宣言.....	6-10
例外の呼び出し方.....	6-11
RAISE 文の使用.....	6-11
例外の遷移.....	6-12
例外の再呼出し.....	6-14
呼び出された例外の処理.....	6-15
宣言の中で呼び出された例外.....	6-16
ハンドラの中で呼び出された例外.....	6-16
例外ハンドラへの分岐と例外ハンドラからの分岐.....	6-17
SQLCODE と SQLERRM の使用.....	6-17
未処理例外.....	6-19
便利なテクニック.....	6-19
例外が呼び出された後に実行を続ける方法.....	6-19
トランザクションの再試行.....	6-20
ロケータ変数の使用.....	6-21

7 サブプログラム

サブプログラムについて.....	7-2
サブプログラムの利点.....	7-3
プロシージャ.....	7-3
ファンクション.....	7-5
副作用の制御.....	7-7
RETURN 文.....	7-8
サブプログラムの宣言.....	7-9
前方宣言.....	7-9
ストアド・サブプログラム.....	7-11

実パラメータと仮パラメータ	7-12
位置表記法と名前表記法	7-13
位置表記法.....	7-13
名前表記法.....	7-13
表記法の混在.....	7-13
パラメータのモード	7-14
IN モード.....	7-14
OUT モード.....	7-14
IN OUT モード.....	7-16
まとめ.....	7-16
NOCOPY コンパイラ・ヒント	7-17
パフォーマンス向上のトレードオフ.....	7-18
NOCOPY の制限.....	7-18
パラメータのデフォルト値	7-19
パラメータのエイリアシング	7-21
オーバーロード	7-23
制限.....	7-24
コールの判定	7-25
コール解決エラーの回避.....	7-27
外部ルーチンのコール	7-27
実行者権限と定義者権限	7-29
実行者権限の利点.....	7-29
AUTHID 句の使用方法.....	7-31
外部参照の解決方法.....	7-32
EXECUTE 権限の付与.....	7-34
ビューとデータベース・トリガーの使用.....	7-35
データベース・リンクの使用.....	7-36
インスタンス・メソッドの起動.....	7-36
再帰	7-37
再帰サブプログラム.....	7-37
相互再帰.....	7-40
再帰と反復.....	7-41

8 パッケージ

パッケージについて.....	8-2
パッケージの利点.....	8-4
パッケージの仕様部.....	8-5
パッケージの内容の参照.....	8-6
パッケージ本体.....	8-7
例.....	8-8
プライベート項目とパブリック項目.....	8-14
オーバーロード.....	8-14
パッケージ STANDARD.....	8-15
製品固有のパッケージ.....	8-16
DBMS_STANDARD	8-16
DBMS_ALERT	8-16
DBMS_OUTPUT.....	8-16
DBMS_PIPE.....	8-17
UTL_FILE	8-17
UTL_HTTP	8-17
指針.....	8-18

9 オブジェクト型

抽象化の役割.....	9-2
オブジェクト型とは?.....	9-3
なぜオブジェクト型を使うか.....	9-5
オブジェクト型の構造.....	9-5
オブジェクト型を構成するさまざまな要素.....	9-7
属性.....	9-7
メソッド.....	9-8
オブジェクト型の定義.....	9-12
オブジェクト型 <i>Stack</i>	9-13
オブジェクト型 <i>Ticket_Booth</i>	9-15
オブジェクト型 <i>Bank_Account</i>	9-17
オブジェクト型 <i>Rational</i>	9-19
オブジェクトの宣言と初期化.....	9-21
オブジェクトの宣言.....	9-21

オブジェクトの初期化.....	9-22
PL/SQL による未初期化オブジェクトの処理.....	9-22
属性へのアクセス	9-23
コンストラクタのコール	9-24
メソッドのコール	9-25
オブジェクトの共有	9-26
ref の使用方法.....	9-27
前方型定義.....	9-27
オブジェクトの操作	9-28
オブジェクトの選択.....	9-29
オブジェクトの挿入.....	9-34
オブジェクトの更新.....	9-35
オブジェクトの削除.....	9-35

10 システム固有の動的 SQL

動的 SQL とは？	10-2
動的 SQL の必要性	10-2
EXECUTE IMMEDIATE 文の使用法	10-3
例.....	10-4
OPEN-FOR、FETCH および CLOSE 文の使用法	10-5
カーソル変数のオープン.....	10-5
カーソル変数からの取出し.....	10-6
カーソル変数のクローズ.....	10-6
例.....	10-7
パラメータのモードの指定	10-9
ティップス	10-10
スキーマ・オブジェクトの名前を渡す.....	10-10
重複するプレースホルダの使用.....	10-10
カーソル属性の使用.....	10-11
NULL を渡す.....	10-12
リモート操作の実行.....	10-12
実行者権限の使用.....	10-13
ブラグマ RESTRICT_REFERENCES の使用.....	10-13
デッドロックの回避.....	10-14

11 言語要素

代入文.....	11-3
ブロック.....	11-7
CLOSE 文.....	11-14
コレクション・メソッド.....	11-16
コレクション.....	11-21
コメント.....	11-26
COMMIT 文.....	11-27
定数と変数.....	11-29
カーソルの属性.....	11-33
カーソル変数.....	11-38
カーソル.....	11-44
DELETE 文.....	11-48
EXCEPTION_INIT プラグマ.....	11-52
例外.....	11-54
EXECUTE IMMEDIATE 文.....	11-57
EXIT 文.....	11-60
式.....	11-62
FETCH 文.....	11-72
FORALL 文.....	11-76
ファンクション.....	11-78
GOTO 文.....	11-82
IF 文.....	11-84
INSERT 文.....	11-87
リテラル.....	11-90
LOCK TABLE 文.....	11-93
LOOP 文.....	11-95
NULL 文.....	11-101
オブジェクト型.....	11-102
OPEN 文.....	11-109
OPEN-FOR 文.....	11-111
OPEN-FOR-USING 文.....	11-115
パッケージ.....	11-118
プロシージャ.....	11-123
RAISE 文.....	11-128
レコード.....	11-130
RETURN 文.....	11-134
ROLLBACK 文.....	11-136

%ROWTYPE 属性.....	11-138
SAVEPOINT 文.....	11-140
SELECT INTO 文.....	11-141
SET TRANSACTION 文.....	11-145
SQL カーソル.....	11-147
SQLCODE ファンクション.....	11-149
SQLERRM ファンクション	11-151
%TYPE 属性.....	11-153
UPDATE 文.....	11-155

A サンプル・プログラム

プログラムの実行.....	A-2
サンプル 1. FOR ループ.....	A-3
入力表.....	A-3
PL/SQL ブロック.....	A-3
出力表.....	A-4
サンプル 2. カーソル.....	A-4
入力表.....	A-4
PL/SQL ブロック.....	A-5
出力表.....	A-5
サンプル 3. 有効範囲.....	A-5
入力表.....	A-5
PL/SQL ブロック.....	A-6
出力表.....	A-6
サンプル 4. バッチ・トランザクション処理.....	A-7
入力表.....	A-7
PL/SQL ブロック.....	A-8
出力表.....	A-9
サンプル 5. 埋込み PL/SQL	A-10
入力表.....	A-10
C プログラム中の PL/SQL ブロック.....	A-10
対話型セッション.....	A-13
出力表.....	A-13
サンプル 6. ストアド・プロシージャのコール.....	A-14
入力表.....	A-14

ストアド・プロシージャ.....	A-15
対話型セッション.....	A-17
B CHAR と VARCHAR2 の意味の比較	
文字値の割当て.....	B-2
文字値の比較.....	B-2
文字値の挿入.....	B-3
文字値の選択.....	B-4
C PL/SQL ラッパー	
ラッピングの利点.....	C-2
PL/SQL ラッパーの実行.....	C-2
入力ファイルと出力ファイル.....	C-3
エラー処理.....	C-4
D ネーム変換	
ネーム変換とは?.....	D-2
種々の参照.....	D-3
ネーム変換のアルゴリズム.....	D-5
ベースの検索.....	D-5
獲得を理解する.....	D-7
内部獲得.....	D-7
同一有効範囲獲得.....	D-9
外部獲得.....	D-9
獲得の防止.....	D-9
属性やメソッドへのアクセス.....	D-10
サブプログラムとメソッドのコール.....	D-11
例 1.....	D-11
例 2.....	D-12
SQL と PL/SQL の比較.....	D-12
E 予約語	
索引	

はじめに

SQL をプロシージャ型言語として機能拡張したオラクル社の製品である PL/SQL は、上級の第 4 世代プログラミング言語です。PL/SQL によって、データのカプセル化、オーバーロード、コレクション型、例外処理、情報隠ぺいといった機能が提供されます。また、シームレスな SQL アクセス、Oracle Server および Oracle Tools との緊密な統合、移植性、セキュリティが提供されます。

このマニュアルでは、PL/SQL の基盤になっているすべての概念を説明し、言語のあらゆる側面について解説します。マニュアル全体を通して数多くの例を示し、すぐれたプログラミング・スタイルを提示します。ユーザーはこのマニュアルを通して、短時間で効果的に PL/SQL について学びます。

主なトピック

[対象読者](#)

[このマニュアルの構成](#)

[表記法規約](#)

[サンプル・データベース表](#)

対象読者

このマニュアルは、Oracle8i 用の、PL/SQL を基盤としたアプリケーションを開発するすべての人を対象としています。特に、プログラマ向けに PL/SQL に関する記述を充実させているので、システム・アナリストやプロジェクト・マネージャ、またデータベース・アプリケーションに関係するその他のユーザーにも役に立ちます。このマニュアルを有効に活用するには、Oracle8i や SQL、および Ada、C、COBOL など 3GL で作業するための知識を持っている必要があります。

このマニュアルでは、インストレーションについての指示またはシステム固有の情報は記述していません。そのような情報については、使っているシステムに該当する Oracle のインストレーション・ガイドまたはユーザーズ・ガイドを参照してください。

このマニュアルの構成

『PL/SQL ユーザーズ・ガイドおよびリファレンス』は、11 の章と 5 つの付録で構成されています。第 1 ～ 10 章は、PL/SQL についての説明で、PL/SQL の多くの機能の使用方を示しています。第 11 章は、PL/SQL のコマンド、構文、およびその意味のリファレンスです。付録 A ～ E には、サンプル・プログラム、技術情報の補足、予約語のリストが含まれています。

第 1 章「概要」 この章では、PL/SQL の主な特徴を取り上げて、その利点を示します。また、PL/SQL の基本概念を説明し、代表的な PL/SQL プログラムの形式を示します。

第 2 章「基礎」 この章では、PL/SQL の細かい点を取り上げます。字句単位、スカラー・データ型、ユーザー定義のサブタイプ、データ変換、式、代入、ブロック構造、宣言、および有効範囲について説明します。

第 3 章「制御構造」 この章では、PL/SQL プログラムの制御の流れを構造化する方法を示します。条件制御、反復制御および順次制御の説明があります。IF-THEN-ELSE や WHILE-LOOP などの単純で強力な制御構造の使用方を学ぶことができます。

第 4 章「コレクションとレコード」 この章では、複合データ型 TABLE、VARRAY、および RECORD について説明します。データのコレクション全体を参照して操作する方法、および関連してはいるが異なるデータを 1 つの論理単位として扱う方法を説明します。また、コレクションをバルク・バインドしてパフォーマンスを改善する方法も説明します。

第 5 章「Oracle の操作」 この章では、Oracle データを操作するための SQL コマンド、ファンクション、および演算子が PL/SQL でどのようにサポートされているかを示します。また、カーソルの管理方法、トランザクションの処理方法、およびデータベースの整合性を保持する方法についても説明します。

第 6 章「エラー処理」 この章では、エラーの報告と回復について詳細に説明します。PL/SQL の例外を使用してエラーの検出と処理を行う方法を学ぶことができます。

第7章「サブプログラム」 この章では、サブプログラムを作成および使用する方法を示します。プロシージャ、ファンクション、前方宣言、実パラメータと仮パラメータ、位置表記法と名前表記法、パラメータ・モード、NOCOPY コンパイラ・ヒント、パラメータのデフォルト値、エイリアシング、オーバーロード、実行者権限、再帰について説明します。

第8章「パッケージ」 この章では、関連する PL/SQL の型、項目、およびサブプログラムを1つのパッケージにまとめる方法を示します。自分で作った汎用パッケージは、コンパイルされて Oracle データベースに格納され、複数のアプリケーションで内容を共用できます。

第9章「オブジェクト型」 この章では、オブジェクト型に基づいたオブジェクト指向プログラミングについて説明します。オブジェクト型は、現実の世界にみられるいろいろなオブジェクトに対応する抽象テンプレートとなるものです。オブジェクト型の定義方法とオブジェクトの操作方法が説明されています。

第10章「システム固有の動的 SQL」 この章では、アプリケーションをより柔軟で多目的に使用できるようにする、上級プログラミング技術の動的 SQL の使用方法を説明します。実行時に、SQL 文を即時に構築および処理できるプログラムを簡単に作成する方法を、2つ説明します。

第11章「言語要素」 この章では、コマンドおよびパラメータ、その他の言語要素を並べて PL/SQL 文を作る方法を示します。また、PL/SQL を早く使いこなすための使用方法と簡単な例を示します。

付録 A「サンプル・プログラム」 この付録では、独自のプログラムを作る上で参考になる PL/SQL プログラムをいくつか示します。サンプル・プログラムには重要な概念や機能が盛り込まれています。

付録 B「CHAR と VARCHAR2 の意味の比較」 この付録では、基本型 CHAR と VARCHAR2 の微妙だが重要な意味上の相違点を説明します。

付録 C「PL/SQL ラッパー」 この付録では、PL/SQL ラッパーの実行方法について説明します。PL/SQL ラッパーは、ソース・コードを隠したまま PL/SQL アプリケーションを配布することを可能にするスタンドアロン・ユーティリティです。

付録 D「ネーム変換」 この付録では、潜在的に意味の曖昧なプロシージャ文および SQL 文で、名前への参照を PL/SQL がどのように解決するかについて説明します。

付録 E「予約語」 この付録では、PL/SQL で使うために予約されている予約語のリストを示します。

表記法規約

このマニュアルでの表記は次の規則に従います。

規則	意味
クーリエ	PL/SQL コード、キーワード、プログラム名、ファイル名、パス名は、クーリエ・フォントで表記する。

PL/SQL のコード例の表記は次の規則に従います。

規則	意味
--	2 連続ハイフンは、その位置から行の終わりまでがコメントであることを示す。
/* */	スラッシュ - アスタリスクとアスタリスク - スラッシュは、複数行にまたがるコメントを区切る。
...	省略記号は、説明と関係のない文や句が省略されていることを示す。
小文字	定数および変数、カーソル、例外、サブプログラム、パッケージの名前には、小文字を使用する。
大文字	キーワード、および事前定義の例外の名前、提供される PL/SQL パッケージの名前には、大文字を使用する。
大小文字混合	ユーザー定義のデータ型およびサブタイプには、大文字と小文字を混在させて使用する。ユーザー定義型の名前は大文字で開始する。

構文定義には、バックス - ナウア記法 (BNF) の簡易形を使用します。BNF には次の記号が含まれます。

記号	意味
[]	オプションの項目を囲む。
{ }	中カッコで囲まれている項目は、そのうちの 1 つだけが必要であることを示す。
	縦線は、大カッコまたは中カッコの中の項目を区切る。
...	省略記号は、直前の構文要素を繰り返し使えることを示す。
デリミタ	大カッコおよび中カッコ、縦線、省略記号を除くデリミタは、示されているとおりに入力しなければならない。

サンプル・データベース表

このマニュアルのほとんどのプログラミング例では、dept と emp というサンプル・データベース表を使用しています。これらの表の定義を次に示します。

```
CREATE TABLE dept (  
    deptno NUMBER(2) NOT NULL,  
    dname  VARCHAR2(14),  
    loc    VARCHAR2(13));  
  
CREATE TABLE emp (  
    empno    NUMBER(4) NOT NULL,  
    ename    VARCHAR2(10),  
    job      VARCHAR2(9),  
    mgr      NUMBER(4),  
    hiredate DATE,  
    sal      NUMBER(7,2),  
    comm     NUMBER(7,2),  
    deptno   NUMBER(2));
```

サンプル・データ

dept 表と emp 表には、それぞれ次のデータが入っています。

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

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

The limits of my language mean the limits of my world.—Ludwig Wittgenstein

この章では、PL/SQL の主な特徴を取り上げて、その利点を示します。また、PL/SQL の基本概念を説明し、代表的な PL/SQL プログラムの形式を示します。PL/SQL が、データベース技術とプロシージャ型プログラミング言語の間のギャップをどのように埋めているのかが理解できます。

主なトピック

主な特徴

[アーキテクチャ](#)

[PL/SQL の利点](#)

主な特徴

PL/SQL を理解するための第一歩としては、サンプル・プログラムを見るのがよいでしょう。次に示すのは、テニス・ラケットの注文を処理するプログラムです。このプログラムは、まずテニス・ラケットの在庫数を格納する `NUMBER` 型の変数を宣言しています。次に、`inventory` という名前のデータベース表から在庫数を取り出します。在庫数がゼロよりも多ければ、プログラムは表を更新し、`purchase_record` という名前の別の表に購入レコードを挿入します。在庫数がゼロ以下の場合は、表 `purchase_record` に在庫切れレコードを挿入します。

```
-- available online in file 'exampl'
DECLARE
    qty_on_hand  NUMBER(5);
BEGIN
    SELECT quantity INTO qty_on_hand FROM inventory
        WHERE product = 'TENNIS RACKET'
        FOR UPDATE OF quantity;
    IF qty_on_hand > 0 THEN -- check quantity
        UPDATE inventory SET quantity = quantity - 1
            WHERE product = 'TENNIS RACKET';
        INSERT INTO purchase_record
            VALUES ('Tennis racket purchased', SYSDATE);
    ELSE
        INSERT INTO purchase_record
            VALUES ('Out of tennis rackets', SYSDATE);
    END IF;
    COMMIT;
END;
```

PL/SQL では、SQL 文を使った Oracle データの操作や、フロー制御文を使ったデータ処理ができます。また、定数や変数を宣言し、プロシージャとファンクションを定義し、実行時エラーを検出し、処理することもできます。このように、PL/SQL では、SQL のデータ操作機能とプロシージャ型言語のデータ処理機能の両方が利用できます。

ブロック構造

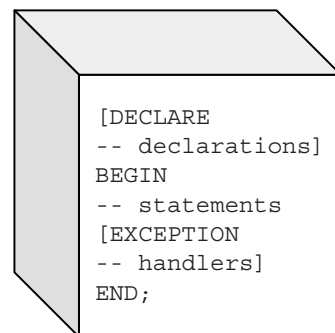
PL/SQL はブロック構造化言語です。つまり、PL/SQL プログラムを構成する基本単位（プロシージャ、ファンクション、無名ブロック）は、それぞれが任意の数のネストされたサブブロックを持つことができる論理ブロックです。一般に、個々の論理ブロックは、解決すべきそれぞれの問題または副問題に対応しています。このように、PL/SQL は、問題を分割して克服する「段階的詳細化」のアプローチをサポートしています。

ブロック（またはサブブロック）は、互いに関連する宣言や文を論理的にグループ化します。これを利用すると、それを使う場所に近い位置に宣言を置くことができます。宣言はブロックの中で局所的に有効で、そのブロックが終わると消滅します。

図 1-1 に示すように、PL/SQL ブロックは宣言部、実行部、例外処理部の 3 つの部分に分かれています。(PL/SQL では警告またはエラー条件のことを「例外」と呼びます。) このうち必ず存在しなければならないのは実行部だけです。

各部分は論理的に並べられています。最初にあるのは宣言部で、ここでは項目を宣言することができます。1 度宣言した項目は、実行部で操作できます。実行の間に起動された例外は、例外処理部で処理されます。

図 1-1 ブロック構造



PL/SQL のブロックまたはサブプログラムの実行部と例外処理部では、サブブロックをネストできます。宣言部ではネストできません。また、どのブロックの宣言部でも、ローカルのサブプログラムを定義できます。ただし、ローカルのサブプログラムは、それが定義されているブロックからしかコールされません。

変数と定数

PL/SQL では変数と定数を宣言し、SQL 文とプロシージャ文の中で、式が使える任意の場所で使えます。ただし、前方参照はできません。このため、宣言文などの他の文で定数または変数を参照するときは、事前に宣言しておく必要があります。

変数の宣言

変数は、CHAR、DATE、NUMBER などの任意の SQL データ型や、BOOLEAN、BINARY_INTEGER などの PL/SQL データ型を持つことができます。たとえば、4 桁の数字が入る part_no という名前の変数と、ブール値 TRUE または FALSE が入る in_stock という名前の変数を宣言します。この場合、変数の宣言は次のようにします。

```
part_no  NUMBER(4);
in_stock BOOLEAN;
```

複合データ型 TABLE、VARRAY、RECORD を使って、ネストした表、可変サイズの配列 (varray)、およびレコードも宣言できます。

変数への値の代入

変数に値を代入する方法は 2 つあります。1 つ目は、コロンに等号を付けた代入演算子 (:=) を使う方法です。変数は演算子の左に、式は演算子の右に置きます。次に例を示します。

```
tax := price * tax_rate;
bonus := current_salary * 0.10;
amount := TO_NUMBER(SUBSTR('750 dollars', 1, 3));
valid := FALSE;
```

変数に値を代入する 2 つ目の方法は、SELECT または FETCH を使ってデータベース値を代入する方法です。次の例では、従業員の給与を取り出して、10% のボーナスを計算しています。

```
SELECT sal * 0.10 INTO bonus FROM emp WHERE empno = emp_id;
```

変数 `bonus` を別の計算に使ったり、変数の値をデータベース表に挿入したりできます。

定数の宣言

定数の宣言は変数の宣言と似ていますが、キーワード `CONSTANT` を付ける点と、定数にただちに値を代入しなければならない点が異なります。その後、定数に値を代入できません。次の例では、`credit_limit` という名前の定数を宣言しています。

```
credit_limit CONSTANT REAL := 5000.00;
```

カーソル

Oracle は、作業領域を使って SQL 文の実行や処理情報を格納します。「カーソル」と呼ばれる PL/SQL の構成体を使うと、作業領域に名前を付けて、そこに格納されている情報にアクセスできます。カーソルには、暗黙カーソルと明示カーソルの 2 種類があります。PL/SQL は、1 行だけを戻す問合せなど、すべての SQL データ操作文に対して、カーソルを暗黙的に宣言します。複数の行を戻す問合せについては、行を 1 行ずつ処理するためにカーソルを明示的に宣言できます。たとえば、

```
DECLARE
    CURSOR c1 IS
        SELECT empno, ename, job FROM emp WHERE deptno = 20;
```

複数行の問合せが戻す行の集合は「結果セット」と呼ばれます。結果セットの大きさは、検索条件に合致した行の数です。[図 1-2](#) に示すように、明示カーソルは結果セットの中の現在の行を指しています。これを利用して、プログラムは行を 1 つずつ処理できます。

図 1-2 問い合わせ処理

結果セット		
7369	SMITH	CLERK
7566	JONES	MANAGER
7788	SCOTT	ANALYST
7876	ADAMS	CLERK
7902	FORD	ANALYST

カーソル → 現在の行

複数行の問い合わせの処理は、ある意味でファイル処理と似ています。たとえば、COBOL プログラムは、ファイルをオープンしてレコードを処理し、その後ファイルをクローズします。これと同じように、PL/SQL プログラムは、カーソルをオープンして、問い合わせによって戻された行を処理し、その後カーソルをクローズします。オープンされているファイルの現在位置をファイル・ポインタが指すのと同じように、カーソルは結果セット中の現在位置を指します。

カーソルの制御には、OPEN 文、FETCH 文、CLOSE 文を使います。OPEN 文は、カーソルに結び付けられている問い合わせを実行し、結果セットを識別し、カーソル位置を先頭行にします。FETCH 文は現在の行を取り出し、カーソルを次の行に進めます。最後の行の処理が終了すると、CLOSE 文によってカーソルを使用不能にします。

カーソル FOR ループ

明示カーソルが必要になる状況では、ほとんどの場合、OPEN 文や FETCH 文や CLOSE 文ではなくカーソル FOR ループを使って、コードを単純化できます。カーソル FOR ループは、データベースからフェッチされた行を表すレコードとして暗黙的にループ索引を宣言します。次にカーソルをオープンし、結果セットから行の値をフェッチしてレコード中のフィールドに入れるという作業を繰り返し、すべての行を処理したらカーソルをクローズします。次の例のカーソル FOR ループは、emp_rec をレコードとして暗黙的に宣言しています。

```
DECLARE
    CURSOR c1 IS
        SELECT ename, sal, hiredate, deptno FROM emp;
    ...
BEGIN
    FOR emp_rec IN c1 LOOP
        ...
        salary_total := salary_total + emp_rec.sal;
    END LOOP;
```

レコード内の個々のフィールドを参照するには、ドット表記法を使います。ドット (.) は、構成要素の選択子の役割を果たします。

カーソル変数

カーソルと同じように、カーソル変数は複数行の問合せの結果セットの中の現在行を指します。ただし、カーソル変数はカーソルとは異なり、型互換性のある任意の問合せ用にオープンできます。また、特定の問合せとは結合しません。カーソル変数は、新しい値を代入したり、Oracle データベースに格納されているサブプログラムに渡したりできる有効な PL/SQL 変数です。この変数を使うと、より柔軟に、また便利な方法でデータ検索を集中的に実行できます。

一般に、カーソル変数をオープンするときは、ストアド・プロシージャに渡し、そのストアド・プロシージャで仮パラメータの 1 つとして宣言します。次のプロシージャは、選択された問合せ用にカーソル変数 `generic_cv` をオープンします。

```
PROCEDURE open_cv (generic_cv IN OUT GenericCurTyp, choice NUMBER) IS
BEGIN
    IF choice = 1 THEN
        OPEN generic_cv FOR SELECT * FROM emp;
    ELSIF choice = 2 THEN
        OPEN generic_cv FOR SELECT * FROM dept;
    ELSIF choice = 3 THEN
        OPEN generic_cv FOR SELECT * FROM salgrade;
    END IF;
```

属性

PL/SQL 変数とカーソルには「属性」があり、これらのプロパティを使うと、定義を繰り返すことなく項目のデータ型や構造を参照できます。データベースの列や表も同様の属性を持ち、これによってメンテナンスが容易になります。パーセント符号 (%) は、属性の標識の役割を果たします。

%TYPE

%TYPE 属性は、変数またはデータベース列のデータ型を与えます。これはデータベース値を保持する変数を宣言する場合に、特に便利です。たとえば、`books` という名前の表に `title` という名前の列があるとします。列 `title` と同じデータ型の変数 `my_title` を宣言するには、ドット表記法と %TYPE 属性を次のように使います。

```
my_title books.title%TYPE;
```

%TYPE 属性を使って `my_title` を宣言することには 2 つの利点があります。第 1 に、ユーザーは `title` の正確なデータ型を知る必要がありません。第 2 に、`title` のデータベース定義を変更した場合（文字列の長さを増やすなど）でも、`my_title` のデータ型はそれに応じて実行時に変化します。

%ROWTYPE

PL/SQL ではデータのグループ化にレコードを使います。レコードは、データ値を格納できる複数の関連したフィールドから構成されます。%ROWTYPE 属性は、表の中の行を表すレコード型を与えます。レコードには、表から選択された行全体、あるいはカーソルまたはカーソル変数で取り出された行全体のデータを格納できます。

行の中の列と、それに対応するレコード中のフィールドは、同じ名前と同じデータ型を持ちます。次の例では、dept_rec という名前のレコードを宣言しています。このレコードのフィールドは、表 dept の列とは名前およびデータ型が同じです。

```
DECLARE
    dept_rec dept%ROWTYPE; -- declare record variable
```

次の例に示すように、フィールドの値にアクセスするにはドット表記法を使います。

```
my_deptno := dept_rec.deptno;
```

従業員の姓および給与、入社日、役職を取り出すカーソルを宣言する場合、%ROWTYPE を使って、これらの情報を格納するレコードを宣言できます。

```
DECLARE
    CURSOR c1 IS
        SELECT ename, sal, hiredate, job FROM emp;
    emp_rec c1%ROWTYPE; -- declare record variable that represents
                        -- a row fetched from the emp table
```

このとき、次の文を実行すると、

```
FETCH c1 INTO emp_rec;
```

表 emp の列 ename の値は emp_rec のフィールド ename に、列 sal の値はフィールド sal に、というように代入されます。図 1-3 に結果の例を示します。

図 1-3 %ROWTYPE レコード

	emp_rec
emp_rec.ename	JAMES
emp_rec.sal	950.00
emp_rec.hiredate	03-DEC-95
emp_rec.job	CLERK

制御構造

制御構造は、SQL に対して加えられた PL/SQL の最も重要な機能拡張です。PL/SQL を使うと、Oracle データを操作できるだけでなく、IF-THEN-ELSE、FOR-LOOP、WHILE-LOOP、EXIT-WHEN、GOTO などの条件制御文、反復制御文および順次制御文を使ってデータを処理できます。これらの文を組み合わせれば、どんな状況にも対応できます。

条件制御

状況に応じてアクションを選ばなければならない場面はよくあります。IF-THEN-ELSE 文を使うと、一連の文を条件に合わせて実行できます。IF 句で条件を検査します。THEN 句で条件が TRUE の場合のアクションを定義し、ELSE 句では条件が FALSE または NULL の場合のアクションを定義します。

次のような、銀行のトランザクションを処理するプログラムを考えます。口座 3 から \$500 を引き出す前に、プログラムは口座に引き出しが可能なだけの預金があるかどうかを確認します。預金があれば、プログラムは口座に出金を記入します。預金がない場合は、監査表にレコードを挿入します。

```
-- available online in file 'examp2'
DECLARE
    acct_balance NUMBER(11,2);
    acct          CONSTANT NUMBER(4) := 3;
    debit_amt     CONSTANT NUMBER(5,2) := 500.00;
BEGIN
    SELECT bal INTO acct_balance FROM accounts
        WHERE account_id = acct
        FOR UPDATE OF bal;
    IF acct_balance >= debit_amt THEN
        UPDATE accounts SET bal = bal - debit_amt
            WHERE account_id = acct;
    ELSE
        INSERT INTO temp VALUES
            (acct, acct_balance, 'Insufficient funds');
        -- insert account, current balance, and message
    END IF;
    COMMIT;
END;
```

問合せの結果を使ってアクションを選択するような一連の文が、データベース・アプリケーションではよく使われます。また、関連するエントリが別の表に見つかった場合にだけ、行の挿入や削除を実行するような一連の文もよく使われます。このようなよく使われる一連の文は、条件判断を使って 1 つの PL/SQL ブロックにまとめられます。これによって、パフォーマンスが向上し、Oracle Forms アプリケーションに組み込まれた整合性検査が単純化されます。

反復制御

LOOP 文を使うと、一連の文を繰り返して実行できます。1つの文の並びを構成している最初の文の前にキーワード LOOP を置き、同じ並びの最後の文の後にキーワード END LOOP を置きます。一連の文を連続的に繰り返す最も簡単な形式のループを次に示します。

```
LOOP
    -- sequence of statements
END LOOP;
```

FOR-LOOP 文では、整数の範囲を指定し、範囲中の整数 1 つについて一連の文を 1 回実行できます。たとえば、カスタムメイドの車を製造する会社で、すべての車に製造番号が付いているものとし、どの客がどの車を購入したのかを記録するには、次の FOR ループを使います。

```
FOR i IN 1..order_qty LOOP
    UPDATE sales SET custno = customer_id
        WHERE serial_num = serial_num_seq.NEXTVAL;
END LOOP;
```

WHILE-LOOP 文は、ある一連の文を条件付きで実行します。ループを反復する前に条件が評価されます。条件が TRUE ならば、一連の文が実行されてから、ループの先頭で制御が再開します。条件が FALSE または NULL ならば、ループは実行されず、制御は次の文に移ります。

次の例では、従業員 7902 よりも指揮系統内で上位にあり、給与が \$4000 よりも高い最初の従業員を探しています。

```
-- available online in file 'examp3'
DECLARE
    salary          emp.sal%TYPE;
    mgr_num         emp.mgr%TYPE;
    last_name       emp.ename%TYPE;
    starting_empno  CONSTANT NUMBER(4) := 7902;
BEGIN
    SELECT sal, mgr INTO salary, mgr_num FROM emp
        WHERE empno = starting_empno;
    WHILE salary < 4000 LOOP
        SELECT sal, mgr, ename INTO salary, mgr_num, last_name
            FROM emp WHERE empno = mgr_num;
        END LOOP;
    INSERT INTO temp VALUES (NULL, salary, last_name);
    COMMIT;
END;
```

これ以上の処理が望ましくない場合、あるいは不可能な場合は、EXIT-WHEN 文でループを終了できます。EXIT 文が見つかったら、WHEN 句の中の条件が評価されます。条件が TRUE ならば、ループは終了し、制御は次の文に移ります。次の例では、total の値が 25,000 を超えたときにループが終了します。

```
LOOP
    ...
    total := total + salary;
    EXIT WHEN total > 25000; -- exit loop if condition is true
END LOOP;
-- control resumes here
```

順次制御

GOTO 文を使うと、無条件にラベルへ分岐します。ラベルは、二重の山カッコで囲まれた未宣言の識別子で、実行可能文または PL/SQL ブロックの前に置かなければなりません。GOTO 文が実行されると、制御はラベルが付いた文またはブロックに移ります。次に例を示します。

```
IF rating > 90 THEN
    GOTO calc_raise; -- branch to label
END IF;
...
<<calc_raise>>
IF job_title = 'SALESMAN' THEN -- control resumes here
    amount := commission * 0.25;
ELSE
    amount := salary * 0.10;
END IF;
```


モジュール性

モジュール性を利用すると、アプリケーションを管理の容易な、正しく定義された論理モジュールに分けることができます。つまり、複雑な問題を次々と詳細化していくことによって、容易に解決できる単純な問題の集まりにすることができるのです。PL/SQL は、このニーズに応えるためにプログラム・ユニットを提供しています。PL/SQL では、ブロックやサブプログラムの他にパッケージも提供しており、これを使うと、関連するプログラム項目をまとめてより大きな単位にできます。

サブプログラム

PL/SQL にはプロシージャとファンクションの 2 種類のサブプログラムがあり、そのどちらもパラメータを取り、起動（コール）できます。次の例に示すように、サブプログラムはプログラムのミニチュアのようなもので、ヘッダーから始まって宣言部（オプション）、実行部、例外処理部（オプション）が続きます。

```
PROCEDURE award_bonus (emp_id NUMBER) IS
    bonus          REAL;
    comm_missing   EXCEPTION;
BEGIN
    SELECT comm * 0.15 INTO bonus FROM emp WHERE empno = emp_id;
    IF bonus IS NULL THEN
        RAISE comm_missing;
    ELSE
        UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
    END IF;
EXCEPTION
    WHEN comm_missing THEN
        ...
END award_bonus;
```

このプロシージャは、コール時に従業員番号を受け付けます。この番号を使ってデータベース表から従業員のコミッションを選択し、同時に 15% のボーナスを計算します。次に、ボーナスの金額を検査します。ボーナスが NULL なら例外が呼び出され、NULL でなければ従業員の給与台帳レコードが更新されます。

パッケージ

PL/SQL では、論理的に関連のある型および変数、カーソル、サブプログラムをパッケージにひとまとめにできます。パッケージはそれぞれ単純、明確に定義されているので、理解しやすくアプリケーションの開発効率を向上させます。これはアプリケーション開発に役立ちます。

通常、パッケージには仕様部と本体の 2 つの部分があります。仕様部はユーザー・アプリケーションとのインタフェースとなり、ここでデータ型や定数、変数、例外、カーソル、サブプログラムなどを宣言します。本体ではカーソルやサブプログラムを定義し、仕様を実際にインプリメントします。

次の例では、パッケージは2つの雇用処理を含んでいます。

```
CREATE PACKAGE emp_actions AS -- package specification
    PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...);
    PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

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

アプリケーションから参照およびアクセスできるのは、パッケージの仕様部の宣言だけです。パッケージ本体のインプリメンテーション詳細は隠ぺいされ、アクセスできません。

パッケージは、コンパイルして Oracle データベースに格納でき、内容を複数のアプリケーションで共有できます。パッケージ・サブプログラムを初めてコールすると、パッケージ全体がメモリーにロードされます。2 度目以降のコールでは、メモリー上のパッケージを使うため、ディスク I/O を必要としません。このことは、生産性と実行速度の向上を意味します。

データの抽象化

データの抽象化により、不必要な詳細を無視しながら、データの基本的な特性を抽出できます。データ構造を一度設計してしまえば、詳細な点を考えることなく、データ構造を操作するアルゴリズムの設計に集中できます。

コレクション

コレクション型 TABLE および VARRAY により、ネストした表および可変サイズの配列 (varray) を宣言できます。コレクションは、すべて同じ型の要素の順序付きグループです。各要素には一意の添字が付いています。その番号によって、コレクションの中での要素の位置が決まります。

要素を参照するには、標準的な添字構文を使います。たとえば、次のコールはファンクション new_hires が戻すネストした表の 5 番目の要素 (Staff 型) を参照します。

```
DECLARE
    TYPE Staff IS TABLE OF Employee;
    staffer Employee;
    FUNCTION new_hires (hiredate DATE) RETURN Staff IS
    BEGIN ... END;
```

```
BEGIN
  staffer := new_hires('10-NOV-98')(5);
  ...
END;
```

コレクションは、ほとんどの第3世代のプログラミング言語で見られる配列のような働きをします。また、コレクションはパラメータとして渡すこともできます。そのため、それらを使うことにより、データの列をデータベースの表に入れたり、データの列をデータベースの表から出したり、クライアント側アプリケーションとストアド・サブプログラムとの間でデータの列を移動したりできます。

レコード

%ROWTYPE 属性を使って、表の中の行またはカーソルからフェッチされた行を表すレコードを宣言できます。さらに、ユーザー定義のレコードを使うと、独自のフィールドを宣言できます。

レコード内のフィールドは一意的な名前を持ちますが、フィールドのデータ型は異なっていてかまいません。名前、給与、入社日など、従業員に関するさまざまなデータがあるとしめます。これらの項目はデータ型が異なりますが、論理的に関連しています。レコードには各項目を表すフィールドが入っています。レコードを使うと、データを1つの論理単位として扱うことができます。

次の例を考えてみます。

```
DECLARE
  TYPE TimeRec IS RECORD (hours SMALLINT, minutes SMALLINT);
  TYPE MeetingTyp IS RECORD (
    date_held DATE,
    duration TimeRec, -- nested record
    location VARCHAR2(20),
    purpose VARCHAR2(50));
```

レコードをネストできることに注意してください。つまり、レコードを他のレコードの構成要素にできます。

オブジェクト型

PL/SQLでのオブジェクト指向のプログラミングは、オブジェクト型をベースにしています。オブジェクト型は、データを操作するのに必要なファンクションおよびプロシージャとともにデータ構造をカプセル化します。データ構造を形成する変数は、属性と呼ばれます。オブジェクト型の動作を特徴付けるファンクションとプロシージャはメソッドと呼ばれます。

オブジェクト型により、大きなシステムが複数の論理エンティティに細分化されるため、複雑さが軽減されます。これにより、モジュール構造を持ち、維持および再利用が可能なソフトウェア・コンポーネントを作れます。

CREATE TYPE 文を使ってオブジェクト型を定義する場合（たとえば SQL*Plus を用いて）、実世界のオブジェクトに対応する抽象テンプレートを作ります。次の銀行口座の例が示すように、テンプレートでは、アプリケーション環境でオブジェクトで必要となる属性と動作だけを指定します。

```
CREATE TYPE Bank_Account AS OBJECT (  
    acct_number INTEGER(5),  
    balance      REAL,  
    status       VARCHAR2(10),  
    MEMBER PROCEDURE open (amount IN REAL),  
    MEMBER PROCEDURE verify_acct (num IN INTEGER),  
    MEMBER PROCEDURE close (num IN INTEGER, amount OUT REAL),  
    MEMBER PROCEDURE deposit (num IN INTEGER, amount IN REAL),  
    MEMBER PROCEDURE withdraw (num IN INTEGER, amount IN REAL),  
    MEMBER FUNCTION curr_bal (num IN INTEGER) RETURN REAL  
);
```

実行時には、データ構造体に値が入れられた時点で、抽象概念としての銀行口座の実際のインスタンスが作られることになります。インスタンス（オブジェクト）は、必要な数だけ作れます。各オブジェクトごとに、実際の銀行口座の口座番号、残高、および状態が含まれています。

情報隠ぺい

情報隠ぺいによって、アルゴリズム設計とデータ構造設計について、指定したレベルの細部しか見えないように制限できます。情報隠ぺいを実施すると、高いレベルでの設計作業と、変更される可能性が大きい低レベルでの設計作業とを分離できます。

アルゴリズム

アルゴリズムの情報隠ぺいはトップダウン設計によってインプリメントします。低レベルのプロシージャの目的とインタフェース仕様を定義すれば、インプリメンテーションの細部は無視できます。高レベルからはこれら細部の構造は見えません。たとえば、raise_salary という名前のプロシージャのインプリメンテーションは隠されています。ユーザーが知っていなければならないのは、このプロシージャが、特定の従業員の給与をある一定の金額だけ増やすという事実だけです。raise_salary の定義に対するすべての変更は、コール側のアプリケーションに透過的です。

データ構造

データ構造の情報隠ぺいは、データのカプセル化によってインプリメントします。特定のデータ構造に対応する一連のユーティリティ・サブプログラムを開発しておけば、データ構造をユーザーや他の開発者から隠ぺいできます。このとき、他の開発者は、データ構造がどのように表現されているかを知らなくても、そのデータ構造を操作するサブプログラムを使えます。

PL/SQL パッケージでは、サブプログラムをパブリックまたはプライベートとして指定できます。そのようにすることにより、パッケージは、サブプログラム定義をブラック・ボックス化してデータのカプセル化を実現します。プライベートな定義は隠されており、アクセスできません。プライベートの型の定義が変化しても、影響を受けるのはパッケージだけで、アプリケーションは影響を受けません。このため、メンテナンスや機能拡張が簡単に実施できます。

エラー処理

PL/SQL では、例外と呼ばれる、事前定義およびユーザー定義のエラー条件を、簡単に検出、処理できます。エラーが発生すると例外が呼び出されます。つまり、通常の実行は中止され、PL/SQL ブロックまたはサブプログラムの例外処理部に制御が移ります。呼び出された例外を処理するには、例外ハンドラと呼ばれる独立したルーチンを作ります。

事前定義の例外は、実行時システムによって暗黙的に呼び出されます。たとえば、数値をゼロで除算しようとする、事前定義の例外 `ZERO_DIVIDE` が自動的に呼び出されます。ユーザー定義の例外は `RAISE` 文によって明示的に呼び出さなければなりません。

ユーザーは、PL/SQL ブロックまたはサブプログラムの宣言部で独自の例外を定義できます。実行部では、特に注意が必要な条件がないかどうかを検査します。その条件がある場合は、`RAISE` 文を実行します。次の例では、営業マンに与えられるボーナスを計算しています。ボーナスは給与とコミッションに基づいて計算されます。このとき、コミッションが `NULL` の場合は例外 `comm_missing` が呼び出されます。

```
DECLARE
...
    comm_missing EXCEPTION; -- declare exception
BEGIN
...
    IF commission IS NULL THEN
        RAISE comm_missing; -- raise exception
    END IF;
    bonus := (salary * 0.10) + (commission * 0.15);
EXCEPTION
    WHEN comm_missing THEN ... -- process exception
```

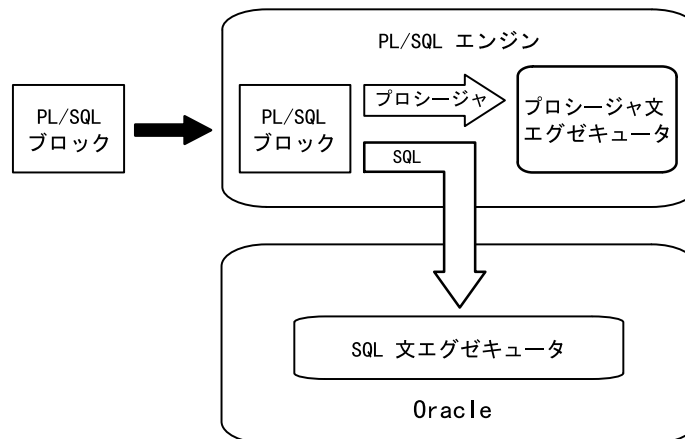
アーキテクチャ

PL/SQL コンパイルおよび実行時システムは、独立した製品ではなく、一種の技術を意味します。この技術は、PL/SQL ブロックとサブプログラムをコンパイルして実行するエンジンのようなものです。このエンジンは、Oracle Server にインストールすることも、Oracle Forms や Oracle Reports のようなアプリケーション開発ツールにインストールすることもできます。そのため、PL/SQL は次の 2 つの環境で使えます。

- Oracle Server
- Oracle Tools

この 2 つの環境は独立しています。PL/SQL は Oracle Server にまとめられていますが、いくつかの Tools では使用できません。どちらの環境でも、PL/SQL エンジンは任意の適切な PL/SQL ブロックまたはサブプログラムを入力として受け付けます。図 1-4 は、無名ブロックを処理する PL/SQL エンジンを示します。エンジンはプロシージャ文だけを実行し、SQL 文を Oracle Server の SQL 文エグゼキュータに送ります。

図 1-4 PL/SQL エンジン



Oracle Server

ローカルな PL/SQL エンジンを持たないアプリケーション開発ツールは、PL/SQL のブロックやサブプログラムの処理に Oracle を利用しなければなりません。PL/SQL エンジンがあれば、Oracle Server は単一の SQL 文だけでなく、PL/SQL のブロックやサブプログラムも処理できます。Oracle Server はブロックとサブプログラムをローカルな PL/SQL エンジンに渡します。

無名ブロック

Oracle プリコンパイラや OCI のプログラムには、名前を指定しないで PL/SQL ブロックを埋め込むことができます。ローカルな PL/SQL エンジンのないプログラムは、実行時にこれらのブロックを Oracle Server に送信します。Oracle Server でそのプログラムをコンパイルおよび実行します。同様に、ローカルな PL/SQL エンジンを持たない SQL*Plus や Enterprise Manager などの対話型ツールは、無名ブロックを Oracle に送信する必要があります。

ストアド・サブプログラム

サブプログラムは、別々にコンパイルして Oracle データベースに永続的に格納し、いつでも実行できるようにすることができます。Oracle Tool で CREATE を使って明示的に作られたサブプログラムは、ストアド・サブプログラムと呼ばれます。コンパイルされ、データ・ディクショナリに格納されたサブプログラムはスキーマ・オブジェクトになり、そのデータベースに接続されている任意の数のアプリケーションから参照できます。

パッケージ内で定義されたストアド・サブプログラムは、パッケージされたサブプログラムと呼ばれます。独立して定義されたものは、スタンドアロン・サブプログラムと呼ばれます。別のサブプログラムや PL/SQL ブロック内で定義されたものは、ローカル・サブプログラムと呼ばれます。これは他のアプリケーションからは参照できず、囲みブロック用に存在します。

ストアド・サブプログラムは、より優れた生産性、パフォーマンス、メモリーの節約、アプリケーションの整合性、セキュリティを実現します。たとえば、ストアド・プロシージャやストアド・ファンクションのライブラリを中心にアプリケーションを設計すると、不要なコーディングを避けて生産性を向上させることができます。

ストアド・サブプログラムは、データベース・トリガー、別のストアド・サブプログラム、Oracle プリコンパイラ・アプリケーション、OCI アプリケーション、または対話的に SQL*Plus や Enterprise Manager からコールできます。たとえば、スタンドアロン・プロシージャ create_dept は、次のようにして SQL*Plus からコールします。

```
SQL> CALL create_dept('FINANCE', 'NEW YORK');
```

サブプログラムは解析され、コンパイルされた形式で格納されます。このため、コールされたサブプログラムはただちにロードされ、PL/SQL エンジンに直接渡されます。また、ストアド・サブプログラムは共有メモリー機能を利用します。したがって、複数のユーザーが実行する場合でも、メモリーにはサブプログラムのコピーが 1 つだけロードされます。

データベース・トリガー

データベース・トリガーは、表と結び付けられているストアド・サブプログラムです。INSERT 文、UPDATE 文または DELETE 文が表に作用する前または後に、自動的にデータベース・トリガーを起動するように Oracle を設定できます。データベース・トリガーの様々な用途の 1 つに、データの変更の監査があります。たとえば、次のデータベース・トリガーは表 emp の給与が更新されるたびに起動されます。

```
CREATE TRIGGER audit_sal
  AFTER UPDATE OF sal ON emp
  FOR EACH ROW
BEGIN
  INSERT INTO emp_audit VALUES ...
END;
```

データベース・トリガーの実行部では、あらゆる SQL データ操作文とプロシージャ文を使えます。

Oracle Tools

PL/SQL エンジンを持つアプリケーション開発用 Oracle Tools は、PL/SQL ブロックとサブプログラムを処理できます。Oracle Tools はブロックをローカルな PL/SQL エンジンに渡します。エンジンはすべてのプロシージャ文をアプリケーション側で実行し、SQL 文だけを Oracle に送信します。このように、大部分の処理はサーバー側ではなくアプリケーション側で行われます。

さらに、ブロックに SQL 文がない場合、PL/SQL エンジンはブロック全体をアプリケーション側で実行します。アプリケーションが条件制御や反復制御を活用できる場合は、この機能が特に便利です。

Oracle Forms アプリケーションは、フィールド・エントリの値のテストや単純な計算のために SQL 文を頻繁に使います。PL/SQL を使うと、Oracle Server へのコールを避けることができます。さらに、PL/SQL ファンクションを使ってフィールド・エントリを操作することもできます。

PL/SQL の利点

PL/SQL は完全な移植性を持つ高性能のトランザクション処理言語で、次のような利点を持っています。

- SQL のサポート
- オブジェクト指向のプログラミングのサポート
- すぐれたパフォーマンス
- 高い生産性
- 完全な移植性
- Oracle との緊密な統合
- セキュリティ

SQL のサポート

SQL は、柔軟かつ強力で、しかも覚えやすいという特長のために、標準データベース言語になりました。SELECT、INSERT、UPDATE、DELETE などの、いくつかの英語に似たコマンドを使って、リレーショナル・データベースに格納されているデータを簡単に操作できます。

SQL は非プロシージャ型です。つまり、処理の方法でなく、要求内容だけを指定します。Oracle がユーザーの要求を実行する最良の方法を判定します。また、Oracle は複数の SQL 文を一度に実行するため、連続する文の間に結び付きがなくてもかまいません。

PL/SQL では、SQL のファンクションおよび演算子、疑似列すべてと同様に、SQL のデータ操作およびカーソル制御、トランザクション制御のすべてのコマンドを使えます。そのため、Oracle データを柔軟かつ安全に操作できます。また、PL/SQL は SQL のデータ型を完全にサポートしています。その結果、アプリケーションとデータベースの間でデータをやり取りする際、データを変換する必要が少なくなります。

PL/SQL は上級プログラミング技術の動的 SQL もサポートしており、これによってアプリケーションをより柔軟で多目的に使用できます。プログラムは SQL データ定義文、データ制御文、セッション制御文を、実行時に即時に作成して処理できます。

オブジェクト指向プログラミングのサポート

オブジェクト型は理想的なオブジェクト指向のモデル・ツールであり、それによって複雑なアプリケーションを構築するのに必要な費用と時間を節約できます。オブジェクト型を使うと、モジュール構造で維持および再利用が可能なソフトウェア構成要素を作れるだけでなく、複数の異なるチームのプログラマが同時にソフトウェア構成要素を開発できます。

データに対する操作をカプセル化することにより、オブジェクト型を使ってデータ・メンテナンスのためのコードを SQL スクリプトの外に出し、PL/SQL ブロックをメソッドに入れることができます。また、オブジェクト型を使えば、インプリメンテーションの細部が隠されるため、クライアント・プログラムに影響を及ぼすことなく細部を変更できます。

さらに、オブジェクト型を使うことにより、現実のデータをモデル化できます。複雑な実世界のエンティティと関連図は、オブジェクト型に直接対応付けることができます。このことは、プログラムがシミュレートしている世界をよりよく反映するのに役立ちます。

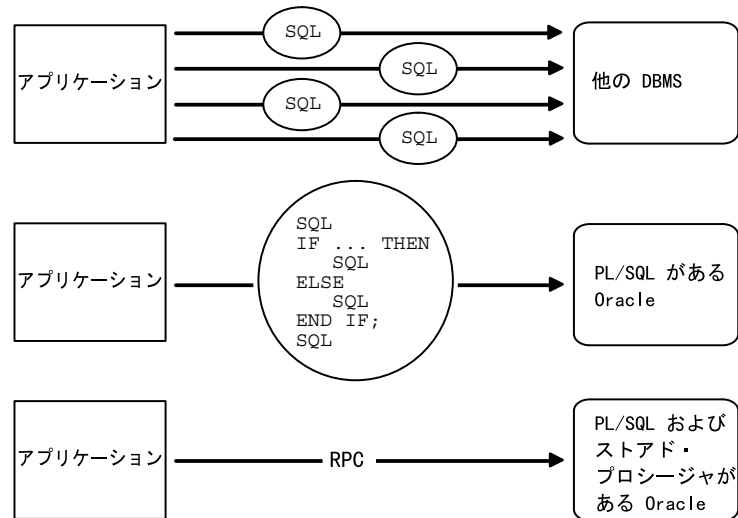
パフォーマンスの向上

PL/SQL がなければ、Oracle は SQL 文を 1 文ずつ処理しなければなりません。1 つの SQL 文は Oracle を 1 回コールするので、パフォーマンスのオーバーヘッドが増加します。ネットワーク環境では、このオーバーヘッドが非常に大きくなることもあります。SQL 文が発行されるたびにネットワーク上で送信しなければならず、通信量が増大します。

しかし、PL/SQL があれば、複数文のブロック全体を Oracle に一度に送信できます。そのため、アプリケーションと Oracle の間の通信量を大幅に削減できます。[図 1-5](#) に示すように、データベースの操作を頻繁に実行するアプリケーションの場合には、PL/SQL ブロックおよびサブプログラムを使って SQL 文をグループ化してから、Oracle に送信して実行させることができます。

PL/SQL ストアド・プロシージャは一度コンパイルされてから実行可能なフォームで格納されるので、プロシージャ・コールは迅速で効果的です。また、サーバーで実行されるストアド・プロシージャは、低速のネットワーク接続上で 1 度コールするだけで起動できます。これによりネットワーク通信量が軽減され、往復応答時間が改善されます。実行可能コードは自動的にキャッシュされ、ユーザー間で共有されます。これにより、必要なメモリー量と起動オーバーヘッドが減少します。

図 1-5 PL/SQL によるパフォーマンスの向上



また、Oracle Tools にプロシージャの処理能力を与えることで、PL/SQL はパフォーマンスを向上させます。PL/SQL を使うと、ツールは Oracle Server をコールすることなく、計算を素早く効果的に実行できます。これは時間の節約になり、ネットワーク通信量を減らすことにもつながります。

高い生産性

PL/SQL は、Oracle Forms や Oracle Reports のような非プロシージャ型のツールに機能を追加しています。これらのツールで PL/SQL を利用すると、使い慣れたプロシージャ型の構成体を使ってアプリケーションを構築できます。たとえば、Oracle Forms トリガーの中で PL/SQL ブロック全体を使えます。複数のトリガー・ステップまたはマクロ、ユーザー・イグジットを使う必要はありません。このように、PL/SQL は優れたツールを提供して生産性を向上させます。

また、PL/SQL はどの環境でも同じです。ある 1 つの Oracle ツールで習得した PL/SQL の知識は他のツールにも利用できるのも、生産性はさらに向上します。たとえば、1 つのツールを使って書いたスクリプトを他のツールでも使えます。

完全な移植性

PL/SQL で書かれたアプリケーションは、Oracle が動作する任意のオペレーティング・システムおよびプラットフォームに移植できます。つまり、PL/SQL プログラムは、Oracle が動作するすべての環境で、手直しをすることなく実行できます。このため、様々な環境で再利用できる、移植性の高いプログラム・ライブラリを作成できます。

Oracle との緊密な統合

PL/SQL と SQL 言語は緊密に統合されています。PL/SQL はすべての SQL データ型と値を持たない NULL をサポートします。これにより、Oracle データを簡単かつ効率的に操作できます。また、パフォーマンスの高いコードを作るのに役立ちます。

%TYPE 属性と %ROWTYPE 属性は、PL/SQL と SQL の統合をさらに進めます。たとえば、%TYPE 属性を使うと、データベース列の定義の宣言を基にして変数を宣言できます。定義が変更されると、次回にプログラムをコンパイルまたは実行したときに、変数宣言もそれに応じて変更されます。ユーザーが何もしなくても新しい定義の効果が得られます。これはデータの独立性を実現し、メンテナンス費を削減する効果があります。また、新しいビジネス・ニーズに合わせてデータベースが変更された場合でも、プログラムは変更に対応できます。

セキュリティ

PL/SQL ストアド・プロシージャによって、クライアントとサーバー間でアプリケーション・ロジックのパーティションが可能です。こうすると、クライアント・アプリケーションが、影響を受けやすい Oracle データを操作しないようにできます。PL/SQL で作成されたデータベース・トリガーはアプリケーションの更新を選択的に無効にし、ユーザーによる問い合わせの内容ベース監査を実行します。

さらに、ユーザーが、定義者の特権で実行されるストアド・プロシージャを通じてでなければ Oracle データを操作できないようにして、Oracle データへのアクセスを制限できます。たとえば、表を更新するプロシージャへのアクセスをユーザーに付与して、表そのもののへのアクセスは付与しないようにします。

2

基礎

There are six essentials in painting. The first is called spirit; the second, rhythm; the third, thought; the fourth, scenery; the fifth, the brush; and the last is the ink.—Ching Hao

前の章では、PL/SQL の概要を示しました。この章では、PL/SQL を詳細に説明します。他のプログラミング言語と同様に、PL/SQL にはキャラクタ・セット、予約語、デリミタ、データ型、厳密な構文および一定の使用規則と文の配置規則があります。PL/SQL のこれらの基本要素を使って、実世界のオブジェクトや演算を表現できます。

主なトピック

キャラクタ・セット

字句単位

データ型

ユーザー定義のサブタイプ

データ型の変換

宣言

命名規則

有効範囲と可視性

代入

式と比較

組込みファンクション

キャラクタ・セット

PL/SQL プログラムは、特定のキャラクタ・セットを使ったテキストとして作られます。PL/SQL キャラクタ・セットには、次のキャラクタが含まれます。

- 大文字と小文字の英字、A ~ Z および a ~ z
- 数字、0 ~ 9
- 記号、() + - * / < > = ! ~ ^ ; : . ' @ % , " # \$ & _ | { } ? []
- タブ、スペースおよび改行

PL/SQL は大文字と小文字を区別しないので、文字列リテラルと文字リテラルの中を除き、小文字の英字は対応する大文字の英字と等価です。

字句単位

PL/SQL テキストの行には字句単位と呼ばれる文字のグループがあります。字句単位は、次のように分類されます。

- デリミタ（単純記号と複合記号）
- 識別子（予約語を含む）
- リテラル
- コメント

たとえば、

```
bonus := salary * 0.10; -- compute bonus
```

この行には次のような字句単位が含まれています。

- 識別子 bonus および salary
- 複合記号 :=
- 単純記号 * および ;
- 数値リテラル 0.10
- コメント -- compute bonus

わかりやすくするために、字句単位は空白で区切ることができます。実際には、隣接する識別子は、空白またはデリミタで区切る必要があります。次の行は予約語の END と IF が結合されているため、不正です。

```
IF x > y THEN high := x; ENDIF; -- illegal
```

ただし、文字列リテラルとコメントの場合を除き、字句単位の中に空白を埋め込むことはできません。たとえば、次の行は代入を表す複合記号（:=）が分かれていますので、不正です。

```
count := count + 1; -- illegal
```

構造を示すために、改行で行を分けたり、空白またはタブで行にインデントを付けたりできます。次の IF 文の読みやすさを比較してみてください。

```
IF x>y THEN max:=x;ELSE max:=y;END IF; | IF x > y THEN
                                         |     max := x;
                                         | ELSE
                                         |     max := y;
                                         | END IF;
```

デリミタ

デリミタは、PL/SQL にとって特別な意味を持つ単純記号または複合記号です。たとえば、デリミタを使って加算や減算などの算術演算を表現できます。単純記号は 1 文字で構成されます。リストを示します。

記号	意味
+	加算演算子
%	属性の標識
'	文字列のデリミタ
.	構成要素の選択子
/	除算演算子
(式またはリストのデリミタ
)	式またはリストのデリミタ
:	ホスト変数の標識
,	項目の区切り子
*	乗算演算子
"	二重引用符で囲んだ識別子のデリミタ
=	関係演算子
<	関係演算子
>	関係演算子
@	リモート・アクセスの標識
;	文の終結子
-	減算 / 否定演算子

複合記号は 2 文字で構成されます。リストを示します。

記号	意味
:=	代入演算子
=>	結合演算子
	連結演算子
**	指数演算子
<<	ラベルのデリミタ（開始）
>>	ラベルのデリミタ（終了）
/*	複数行コメントのデリミタ（開始）
*/	複数行コメントのデリミタ（終了）
..	範囲演算子
<>	関係演算子
!=	関係演算子
~=	関係演算子
^=	関係演算子
<=	関係演算子
>=	関係演算子
--	1 行コメントの標識

識別子

識別子を使って、定数、変数、例外、カーソル、カーソル変数、サブプログラム、パッケージなどの PL/SQL プログラムに名前を付けることができます。次に識別子の例をいくつか示します。

```
X
t2
phone#
credit_limit
LastName
oracle$number
```

識別子は英字 1 文字でも構いませんが、後に英字、数字、ドル記号、アンダースコアおよび番号記号を続けることもできます。次の例のように、ハイフン、スラッシュ、空白などの文字は使えません。

```
mine&yours      -- illegal ampersand
debit-amount    -- illegal hyphen
on/off          -- illegal slash
user id         -- illegal space
```


次の例は、ドル記号、アンダースコアおよび番号記号を隣接して使ったり、先頭以外の位置で使ったりできることを示しています。

```
money$$$tree
SN##
try_again_
```

識別子では大文字と小文字が使える、両者の混用もできます。PL/SQL では、文字列リテラルと文字リテラルの中を除いて、大文字と小文字は区別されません。したがって、2 つの識別子の違いが英字の大文字と小文字の違いだけならば、PL/SQL は同じ識別子とみなします。次の例を参照してください。

```
lastname
LastName  -- same as lastname
LASTNAME  -- same as lastname and LastName
```

識別子の長さは 30 文字を超えてはなりません。なお、ドル記号、アンダースコアおよび番号記号を含むすべての文字が意味を持ちます。たとえば、PL/SQL は、次の 2 つの識別子を別のものとして扱います。

```
lastname
last_name
```

識別子は、内容がわかりやすいものにするべきです。cpm のようなあいまいな名前は避けま
す。かわりに、cost_per_thousand のように意味のわかりやすい名前を使ってください。

予約語

識別子の中には、PL/SQL に対して構文上の特別な意味を持ち、再定義が不可能な予約語があります。たとえば、BEGIN と END という語は、ブロックまたはサブプログラムの実行部を囲む予約語です。次の例に示すように、予約語を再定義しようとするとコンパイル・エラーが発生します。

```
DECLARE
    end BOOLEAN;  -- illegal; causes compilation error
```

ただし、次の例のように識別子の中に予約語を埋め込むことはできます。

```
DECLARE
    end_of_game BOOLEAN;  -- legal
```

一般に、予約語は区別しやすくするために大文字で書かれています。ただし、PL/SQL の他の識別子と同様に、予約語は小文字で書くことも、大文字と小文字を混在させることもできます。予約語のリストは、[付録 E](#)にあります。

事前定義の識別子

例外 `INVALID_NUMBER` など、パッケージ `STANDARD` でグローバルに宣言されている識別子は、再宣言できません。しかし、事前定義の識別子を再宣言すると、ローカルな宣言がグローバルな宣言を上書きするのでエラーが発生しやすくなります。

二重引用符で囲んだ識別子

柔軟性を高めるために、PL/SQL では識別子を二重引用符で囲むことができます。通常は、このようにする必要はありませんが、ときには便利な場合もあります。二重引用符で囲んだ識別子には、空白など、二重引用符を除くすべての印字可能文字を任意に並べて入れることができます。したがって、次の識別子は有効です。

```
"X+Y"
"last name"
"on/off switch"
"employee(s) "
"*** header info ***"
```

二重引用符で囲んだ識別子の最大長は、二重引用符を数えずに 30 字です。PL/SQL の予約語を二重引用符で囲んだ識別子として使用することもできますが、それはあまり好ましくないプログラミング習慣です。

PL/SQL の予約語の中には、SQL では予約されていないものがあります。たとえば、PL/SQL の予約語の `TYPE` は、`CREATE TABLE` 文の中でデータベースの列に名前を付けるために使えます。しかし、次の例のようにプログラム中の SQL 文がその列を参照すると、コンパイル・エラーが発生します。

```
SELECT acct, type, bal INTO ... -- causes compilation error
```

このエラーを防ぐには、次のように列名を大文字にして二重引用符で囲みます。

```
SELECT acct, "TYPE", bal INTO ...
```

列名は (`CREATE TABLE` 文でそのように定義された場合を除き) 小文字、または大文字と小文字を混在させては使えません。たとえば、次の文は無効です。

```
SELECT acct, "type", bal INTO ... -- causes compilation error
```

ビューを作って不正な列を改名し、SQL 文の中で実表のかわりにこのビューを使えます。

リテラル

リテラルは、識別子によって表現する必要がない明示的な数値または文字、文字列、ブール値です。例として、数値リテラル `147` やブール・リテラル `FALSE` があります。

数値リテラル

算術式では、整数と実数の 2 種類の数値リテラルを使えます。整数リテラルは、小数点を持たず、必要に応じて符号を付けた整数です。次に例を示します。

```
030    6    -14    0    +32767
```

実数リテラルとは、小数点を持ち、必要に応じて符号を付けた整数または小数です。次に例を示します。

```
6.6667    0.0    -12.0    3.14159    +8300.00    .5    25.
```

`12.0` や `25.` といった数値は整数値ですが、PL/SQL では実数とみなされます。

数値リテラルはドル記号やコンマを含むことはできませんが、科学表記法で書くことができます。数字の後に `E` (または `e`) を付けて、必要な場合は符号付き整数を続けます。次に例を示します。

```
2E5    1.0E-7    3.14159e0    -1E38    -9.5e-3
```

`E` は、"10 の累乗" を意味します。次の例で示すように、`E` の前の数に、`E` の後の数の 10 の累乗を掛けます (二重アスタリスク (`**`) は指数演算子です)。

```
5E3 = 5    10**3 = 5    1000 = 5000
```

`E` の後の数値は、小数点がずれる桁数にも対応しています。上の例では、暗黙の小数点が 3 桁右に移動しました。次の例では、3 桁左に移動します。

```
5E-3 = 5    10**-3 = 5    0.001 = 0.005
```

次の例に示すように、数値リテラルの値が `1E-130` ~ `10E125` の範囲外の場合、コンパイラ・エラーが発生します。

```
DECLARE
    n NUMBER;
BEGIN
    n := 10E127; -- causes a 'numeric overflow or underflow' error
```

文字リテラル

文字リテラルは引用符（アポストロフィ）で囲まれた 1 文字のことです。文字リテラルには PL/SQL キャラクタ・セットの表示可能な文字がすべて使えます。つまり、英字および数字、空白、特殊記号を使えます。次に例を示します。

```
'Z'   '7'   'z'   ' ('
```

文字リテラルの中で、PL/SQL は大文字と小文字を区別します。このため、PL/SQL はリテラル 'Z' と 'z' を違うものとして扱います。また文字リテラル literals '0' ~ '9' は、整数リテラルと等価ではありませんが、暗黙のうちに整数に変換されるため、算術式の中で使用できます。

文字列リテラル

文字値は、識別子によって表現することも、引用符（'）で囲まれたゼロ文字以上の並びである文字列リテラルとして明示的に書くこともできます。次に例を示します。

```
'Hello, world!'  
'XYZ Corporation'  
'10-NOV-91'  
'He said "Life is like licking honey from a thorn."  
'$1,000,000'
```

NULL 文字列（"）を除くすべての文字列リテラルは、CHAR データ型に属します。

アポストロフィ（引用符）で文字列リテラルを区切るとすると、文字列中でアポストロフィを表現する場合はどうすればいいのでしょうか。次の例に示すように、引用符（'）を 2 つ書きます。これは二重引用符を書く場合とは違う意味を持ちます。

```
'Don''t leave without saving your work.'
```

文字列リテラルの中で、PL/SQL は大文字と小文字を区別します。たとえば、PL/SQL は次の 2 つのリテラルを異なるものとして扱います。

```
'baker'  
'Baker'
```

ブール・リテラル

ブール・リテラルとは、事前定義の値 TRUE と FALSE、および NULL（存在しない値、未知の値または適用不可能な値を表す）のことです。ブール・リテラルは値であり、文字列ではないことに注意してください。たとえば、TRUE は数値 25 と同じように 1 つの値です。

コメント

PL/SQL コンパイラはコメントを無視しますが、ユーザーはコメントを無視するべきではありません。プログラムにコメントを付け加えると、わかりやすくなり理解に役立ちます。一般に、コメントは各コード・セグメントの目的や使用方法を説明するために使います。PL/SQL では、1 行コメントと複数行コメントの 2 種類のコメント・スタイルをサポートしています。

1 行コメント

1 行コメントは、行の中の任意の位置にある二重ハイフン (--) から始まり、その行の終わりまで続きます。次に例を示します。

```
-- begin processing
SELECT sal INTO salary FROM emp -- get current salary
    WHERE empno = emp_id;
bonus := salary * 0.15; -- compute bonus amount
```

コメントは、行の末尾ならば文の途中でも使えることに注意してください。

プログラムのテストやデバッグのときに、コード中の 1 つの行を使用不能にしたい場合があります。行を「コメントにする」方法を次の例に示します。

```
-- DELETE FROM emp WHERE comm IS NULL;
```

複数行

複数行コメントは、スラッシュ - アスタリスク (/*) で始まってアスタリスク - スラッシュ (*/) で終わり、複数行にまたがることができます。次に例を示します。

```
BEGIN
    ...
    /* Compute a 15% bonus for top-rated employees. */
    IF rating > 90 THEN
        bonus := salary * 0.15 /* bonus is based on salary */
    ELSE
        bonus := 0;
    END If;
    ...
    /* The following line computes the area of a
       circle using pi, which is the ratio between
       the circumference and diameter. */
    area := pi * radius**2;
END;
```

次の例に示すように、複数行コメントを使って、コードの一部をそのままコメントにすることができます。

```
/*  
LOOP  
    FETCH c1 INTO emp_rec;  
    EXIT WHEN c1%NOTFOUND;  
    ...  
END LOOP;  
*/
```

制限

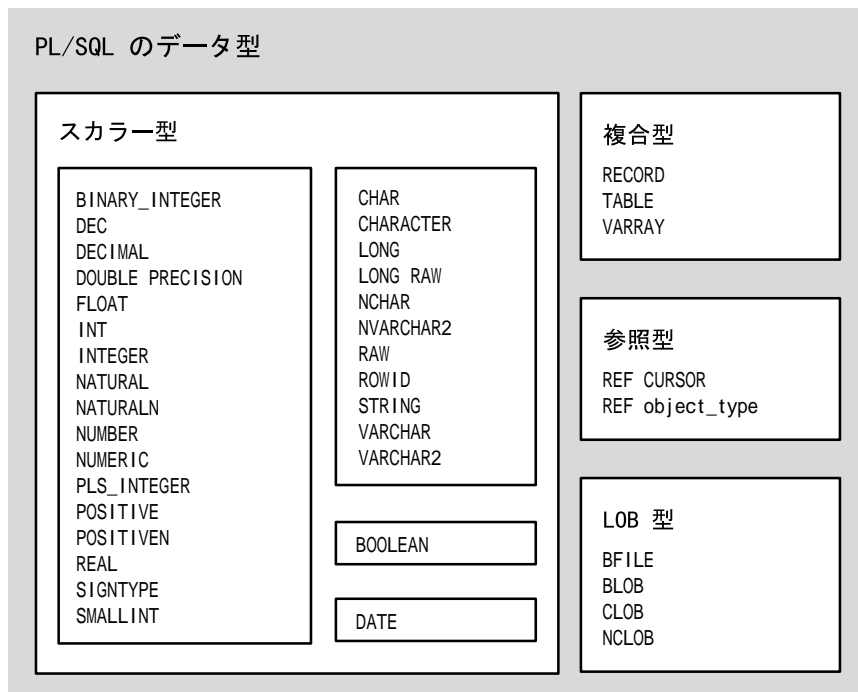
コメントはネストできません。また、Oracle プリコンパイラ・プログラムが動的に処理する PL/SQL ブロックの中では、1 行コメントは使えません。これは、行の終わりを示す文字が無視されるので、1 行コメントが行の終わりではなくブロックの終わりまで続いてしまうためです。この場合は複数行コメントを使ってください。

データ型

すべての定数と変数は、記憶形式および制約、値の有効範囲を指定する、データ型を持っています。PL/SQL には、様々な事前定義のデータ型が用意されています。スカラー型には内部コンポーネントがありません。複合型には、個別に操作できる内部コンポーネントがあります。参照型には、他のプログラム項目を指定するポインタという値があります。LOB 型には、LOB ロケータと呼ばれる値が入れられます。この値は、行外部に格納されている大きなオブジェクト（図形イメージなど）の位置を指定します。

[図 2-1](#) に、使用可能な事前定義済みのデータ型を示します。スカラー型は数値、文字、日付／時刻、真理値などを格納するデータの種類によって 4 つのグループに分類されます。

図 2-1 組み込みデータ型



ここでは、スカラー型および LOB 型について説明します。複合型については、[第 4 章](#)で説明します。参照型については、[第 5 章](#)および[第 9 章](#)で説明します。

数値型

数値型は、数値データ（整数、実数、浮動小数点数）を格納したり、量を表現したり、計算したりするのに使います。

BINARY_INTEGER

BINARY_INTEGER データ型は、符号付き整数を格納するために使います。値の大きさの範囲は、-2147483647 ~ 2147483647 です。BINARY_INTEGER の値は、PLS_INTEGER の値と同じように、NUMBER の値より少ない記憶域しか必要としません。ただし、ほとんどの BINARY_INTEGER 演算は PLS_INTEGER 演算より処理速度が遅くなります。

BINARY_INTEGER サブタイプ 基本型は、サブタイプの派生元となるデータ型です。サブタイプは基本型を制約と結び付け、値のサブセットを定義します。PL/SQL では、次のような BINARY_INTEGER サブタイプがあらかじめ定義されています。

```
NATURAL
NATURALN
POSITIVE
POSITIVEN
SIGNTYPE
```

サブタイプ NATURAL と POSITIVE は、整変数をそれぞれ負でない整数値、または正の整数値に制限したい場合に使います。NATURALN と POSITIVEN は、整変数に NULL を代入できないようにします。SIGNTYPE は、整変数を値 -1、0、1 に制限するときに使います。これは 3 値論理のプログラミングに役立ちます。

NUMBER

NUMBER データ型を使うと、事実上、任意のサイズの固定小数点数または浮動小数点数を格納できます。値の大きさの範囲は、 $1\text{E-}130 \sim 10\text{E}125$ です。式の値がこの範囲外にあると、数値オーバーフローまたはアンダーフローのエラーが発生します。全体の桁数を精度 (*precision*) として、小数点以下の桁数を位取り (*scale*) として指定できます。次に構文を示します。

```
NUMBER [ (precision, scale) ]
```

固定小数点数を宣言するには (位取り (*scale*) を指定する必要があります) 次のフォームを使います。

```
NUMBER (precision, scale)
```

浮動小数点数を宣言する場合は (小数点が任意の位置に浮動するので、精度 (*precision*) や位取り (*scale*) を指定できません) 次のフォームを使います。

```
NUMBER
```

整数を宣言する場合は (小数点がありません) 次のフォームを使います。

```
NUMBER (precision) -- same as NUMBER (precision, 0)
```

精度 (*precision*) と位取り (*scale*) の指定には、定数や変数を使えません。整数リテラルを使用する必要があります。NUMBER 型の値の場合、精度 (*precision*) の最大値は 10 進の 38 桁です。精度 (*precision*) を指定しないと、38 が、デフォルトでシステムがサポートしている最大値のどちらか小さいほうになります。

位取り (scale) の範囲は -84 ~ 127 で、この値によって四捨五入の位置が決まります。たとえば、位取り (scale) として 2 を指定すると小数点以下 2 桁に四捨五入されます (3.456 は 3.46 になります)。負の位取り (scale) を指定すると、小数点の左側で四捨五入されます。たとえば、位取り (scale) として -3 を指定すると、1000 の単位に四捨五入されます (3456 は 3000 になります)。位取り (scale) として 0 を指定すると、最も近い整数値に四捨五入されます。位取り (scale) を指定しないと、デフォルトでゼロになります。

NUMBER サブタイプ 次の NUMBER サブタイプは、名前の記述性を高め、ANSI/ISO および IBM 型に対して互換性を保つ目的で用意されたデータ型です。

```
DEC
DECIMAL
DOUBLE PRECISION
FLOAT
INTEGER
INT
NUMERIC
REAL
SMALLINT
```

サブタイプ DEC、DECIMAL、NUMERIC は、固定小数点数を宣言する場合に使います。その場合、精度 (precision) の最大値は 10 進数の 38 桁です。

サブタイプ DOUBLE PRECISION と FLOAT は、浮動小数点数を宣言する場合に使います。その場合、精度 (precision) の最大値は 2 進数の 126 桁であり、10 進数の 38 桁にほぼ等しくなります。サブタイプ REAL は、浮動小数点数を宣言する場合に使います。その場合、精度 (precision) の最大値は 2 進数で 63 桁であり、およそ 10 進数の 18 桁に等しくなります。

サブタイプ INTEGER、INT、SMALLINT は、整数を宣言する場合に使います。その場合、精度 (precision) の最大値は 10 進数の 38 桁です。

PLS_INTEGER

PLS_INTEGER データ型は、符号付き整数を格納するために使います。値の大きさの範囲は、-2147483647 ~ 2147483647 です。PLS_INTEGER 値は、NUMBER の値より少ない記憶域しか必要としません。また、PLS_INTEGER 演算はマシン算術計算を使うため、ライブラリ算術計算を使う NUMBER 演算や BINARY_INTEGER 演算よりも処理速度が速くなります。より高いパフォーマンスを得るために、PLS_INTEGER の大きさの範囲内でのすべての計算に PLS_INTEGER を使ってください。

PLS_INTEGER と BINARY_INTEGER は、値の大きさの範囲は同じですが、完全に互換ではありません。PLS_INTEGER 計算がオーバーフローすると、例外が発生します。しかし、BINARY_INTEGER 計算がオーバーフローしても、結果が NUMBER 変数に代入される場合には例外は発生しません。

このように、わずかですが意味上の違いがあるので、古いアプリケーションでは互換性を保つために従来どおり `BINARY_INTEGER` を使ってください。新しいアプリケーションでは、より高いパフォーマンスを得るために必ず `PLS_INTEGER` を使ってください。

文字型

文字型は、英数字のデータを格納したり、ワードとテキストを表現したり、文字ストリングを操作したりするのに使います。

CHAR

CHAR データ型は固定長の文字データを格納するのに使います。データの内部表現形式は、データベース・キャラクタ・セットによって異なります。CHAR データ型はオプション・パラメータを使って、その最大長を 32767 バイトまで指定できます。次に構文を示します。

```
CHAR[(maximum_length)]
```

最大長には 1 ~ 32767 の範囲の整数リテラルを指定します。定数や変数は指定できません。

最大長を指定しない場合のデフォルト値は 1 です。最大長を指定して CHAR(n) 変数とした場合、n は文字数ではなくバイト数となることに注意してください。したがって、CHAR(n) 変数にマルチバイト文字を格納する場合、最大長は n 文字より少なくなります。CHAR データベース列の最大幅は 2000 バイトです。このため、2000 バイトよりも長い CHAR 値を CHAR 型の列に挿入できません。

任意の CHAR(n) 値を LONG データベース列に格納できます。LONG 列の最大幅は 2147483647 バイト、すなわち 2GB だからです。ただし、LONG 列から 32767 バイトを超える値を取り出して CHAR(n) に入れることはできません。

注意：付録 B では、CHAR 基本型と VARCHAR2 基本型の意味上の相違点を説明しています。

CHAR サブタイプ CHAR サブタイプ CHARACTER の値は、その基本型と同じ範囲をとります。つまり、CHARACTER は CHAR の別名にすぎません。このサブタイプは、CHAR 型だけの場合よりも識別子の記述性を高め、ANSI/ISO および IBM 型に対して互換性を保つ目的で用意されたデータ型です。

LONG と LONG RAW

LONG データ型は、可変長の文字列を格納するために使います。LONG データ型は、この型の値の最大長が 32760 バイトであるという点を除けば、VARCHAR2 データ型と同じです。

LONG RAW データ型はバイナリ・データやバイト列を格納するために使います。LONG RAW データ型は、この型のデータが PL/SQL によって解釈されないという点を除けば、LONG データ型と同じです。LONG RAW 型の値の最大長は 32760 バイトです。

任意の LONG 値を LONG データベース列に格納できます。LONG 列の最大幅は 2147483647 バイトだからです。ただし、LONG 列から 32760 バイトを超える値を取り出して LONG 変数に入れることはできません。

同様に、任意の LONG RAW 値を LONG RAW データベース列に格納できます。LONG RAW 列の最大幅は 2147483647 バイトだからです。ただし、LONG RAW 列から 32760 バイトを超える値を取り出して LONG RAW 変数に入れることはできません。

LONG 型の列にはテキスト、文字の配列または短いドキュメントなどが格納できます。LONG 型の列は、UPDATE 文、INSERT 文、および（ほとんどの）SELECT 文で参照できますが、式や SQL ファンクション・コール、または WHERE、GROUP BY、CONNECT BY といった一部の SQL 句では参照できません。詳細は、『Oracle8i SQL リファレンス』を参照してください。

RAW

RAW データ型はバイナリ・データやバイト列を格納するために使います。たとえば、RAW 型変数には図形文字の並びやデジタル化された絵を格納できます。RAW データは VARCHAR2 データと似ていますが、PL/SQL によって解釈されない点が違います。また、RAW データをシステム間で送信する際に、Net8 はキャラクタ・セット変換を実行しません。

RAW データ型は必須パラメータを使って、その最大長を 32767 バイトまで指定できます。次に構文を示します。

```
RAW(maximum_length)
```

最大長には 1 ~ 32767 の範囲の整数リテラルを指定します。定数や変数は指定できません。

RAW データベース列の最大幅は 2000 バイトです。このため、2000 バイトよりも長い RAW 値を RAW 型の列に挿入できません。任意の RAW 値を LONG RAW データベース列に格納できます。LONG RAW 列の最大幅は 2147483647 バイトだからです。ただし、LONG RAW 列から 32767 バイトを超える値を取り出して RAW 変数に入れることはできません。

ROWID と UROWID

内部的に、すべてのデータベース表には、行 ID というバイナリ値を格納する ROWID 疑似列があります。各行 ID は、行の記憶領域アドレスを表します。物理行 ID は、通常の表の行を識別します。論理行 ID は、索引構成表の行を識別します。ROWID データ型には物理行 ID だけを格納できます。ただし、UROWID（汎用行 ID）データ型には物理行 ID、論理行 ID、外部（非 Oracle）行 ID を格納できます。

提案：ROWID データ型は、古いアプリケーションとの下位互換性のためだけに使用してください。新しいアプリケーションの場合には、UROWID データ型を使います。

行 ID を選択または取り出して UROWID 変数に入れる場合は、バイナリ値を 18 バイトの文字列に変換する組み込みファンクション ROWIDTOCHAR を使います。逆に、ファンクション CHARTOROWID は UROWID 文字列を行 ID に変換します。文字列が有効な行 ID を表していないために変換が失敗すると、PL/SQL は事前定義の例外 SYS_INVALID_ROWID を発生します。これは、暗黙の変換にも適用されます。

物理行 ID 物理行 ID を使うと、特定の行にすばやくアクセスできます。行 ID は、その行が存在するかぎり変わりません。行 ID は効率的で安定しているため、行の集合を選択し、集合全体を操作してサブセットを更新するのに便利です。たとえば、UPDATE 文または DELETE 文の WHERE 句の中で UROWID 変数と ROWID 疑似列を比較して、カーソルから取り出された最新の行を識別できます。5-44 ページの「[コミットにまたがる取出し](#)」を参照してください。

物理行 ID には 2 つの形式があります。10 バイトの拡張 ROWID 形式は相対表領域ブロック・アドレスをサポートしており、パーティション化表と非パーティション化表の行を識別できます。6 バイトの制限 ROWID 形式は、下位互換性のために用意されています。

拡張 ROWID は、選択された各行の物理アドレスの base-64 コード化を使用します。たとえば、行 ID を暗黙的に文字列に変換する SQL*Plus での次の問合せを考えてみます。

```
SQL> SELECT rowid, ename FROM emp WHERE empno = 7788;
```

これは、次の行を戻します。

```
ROWID          ENAME
-----
AAAAqcAABAAADFNAAH SCOTT
```

形式 OOOOOOFFFFBBBBBBRRR は、4 つの部分に分かれています。

- OOOOOO: データ・オブジェクト番号（上の例では AAAAqc）。データベース・セグメントを識別する。表のクラスタなど、同じセグメント中のスキーマ・オブジェクトのオブジェクト番号は同じです。
- FFF: ファイル番号（上の例では AAB）。行が入っているデータ・ファイルを識別する。ファイル番号はデータベース内で一意です。
- BBBB: ブロック番号（上の例では AAADFN）。行が入っているデータ・ブロックを識別する。ブロック番号は、その表領域ではなく、データ・ファイルに対応します。このため、同じ表領域内の異なるデータ・ファイルにある 2 つの行は、同じブロック番号を持つことができます。
- RRR: 行番号（上の例では AAH）。ブロック内の行を識別する。

論理行 ID 論理行 ID を使うと、特定の行に最も早くアクセスできます。Oracle は論理行 ID を使用して索引構成表の 2 次索引を組み立てます。論理行 ID は恒久物理アドレスを持たないので、新しい行が挿入されると複数のデータ・ブロックの間を移動できます。ただし、行の物理的な位置が変わった場合、その論理行 ID は有効なまま残ります。

論理行 ID には推測を含めることができます。これは推測が行われたときの行のブロック位置を識別します。Oracle は全体のキー検索をせずに、推測を使ってブロックを直接検索します。ただし新しい行が挿入されると、推測は古くなって行へのアクセスの処理速度が遅くなります。新しい推測を得るには、2 次索引を再作成します。

ROWID 疑似列を使用して、索引構成表から論理行 ID（これは不透明値です）を選択できます。また、最大サイズが 4000 バイトの UROWID 型の列に論理行 ID を挿入できます。

ANALYZE 文は、推測の古さをトラックするのに役立ちます。これは、推測を持つ行 ID を UROWID 列に格納し、その行 ID を使って行をフェッチするアプリケーションの場合に便利です。ただし、揮発性の高い表からフェッチする場合は、推測を持たない行 ID を使うことをお勧めします。

注意：行 ID を操作するには、提供されているパッケージ DBMS_ROWID を使います。詳細は、『Oracle8i パッケージ・プロシージャ リファレンス』を参照してください。

VARCHAR2

VARCHAR2 データ型は可変長の文字データを格納するのに使います。データの内部表現形式は、データベース・キャラクタ・セットによって異なります。VARCHAR2 データ型は必須パラメータを使って、その最大長を 32767 バイトまで指定できます。次に構文を示します。

VARCHAR2 (maximum_length)

最大長には 1 ~ 32767 の範囲の整数リテラルを指定します。定数や変数は指定できません。

VARCHAR2 データ型ではメモリー使用と効率の間のトレードオフが発生します。

VARCHAR2 (>= 2000) 変数の場合、PL/SQL は実際の値を保持するのに必要なだけのメモリーを動的に割り当てます。しかし、VARCHAR2 (<= 2000) 変数の場合、PL/SQL は最大サイズの値を保持するのに必要なメモリーを事前に割り当てます。このため、たとえば VARCHAR2 (2000) 変数と VARCHAR2 (1999) 変数に同じ 500 バイトの値を割り当てると、VARCHAR2 (1999) 変数の方が 1499 バイト多くメモリーを使うことになります。

VARCHAR2 (n) 変数の最大長は、文字数ではなくバイト数で指定することに注意してください。したがって、VARCHAR2 (n) 変数にマルチバイト文字を格納する場合、最大長は n 文字より少なくなります。VARCHAR2 データベース列の最大幅は 4000 バイトです。このため、4000 バイトよりも長い VARCHAR2 値を VARCHAR2 型の列に挿入できません。

任意の VARCHAR2 (n) 値を LONG データベース列に格納できます。LONG 列の最大幅は 2147483647 バイトだからです。ただし、LONG 列から 32767 バイトを超える値を取り出して VARCHAR2 (n) に入れることはできません。

VARCHAR2 サブタイプ 次の VARCHAR2 サブタイプの値は、その基本型と同じ範囲をとります。たとえば、VARCHAR は VARCHAR2 の別名です。

STRING
VARCHAR

このサブタイプは、ANSI/ISO および IBM 型に対して互換性を保つ目的で用意されたデータ型です。

注意：現在のところ、VARCHAR は VARCHAR2 と同義です。しかし PL/SQL の今後のリリースでの VARCHAR は、SQL 標準に従うために比較時の意味が違ふ別個のデータ型になる可能性があります。そのため、VARCHAR よりも VARCHAR2 を使うことをお勧めします。

NLS 文字型

よく使われる 7 ビットまたは 8 ビットの ASCII と EBCDIC のキャラクタ・セットはアルファベットを表示するには十分ですが、日本語などのアジアの言語には何千もの文字があります。これらの言語では、1 文字を表すのに 16 ビット (2 バイト) が必要です。Oracle はこれら異質の言語をどのように扱っているのでしょうか。

Oracle には各国語サポート (NLS) が用意されており、シングルバイト文字データとマルチバイト文字データを処理したり、キャラクタ・セット間で変換したりできます。また、アプリケーションを複数の異なる言語環境で実行できます。

NLS により、数字と日付の形式は、ユーザー・セッションで指定されている言語の規則に自動的に合わせられます。したがって、NLS により世界中のユーザーは Oracle を使って母国語で対話できます。NLS の詳細は、『Oracle8i NLS ガイド』を参照してください。

PL/SQL は、識別子およびソース・コードに使われるデータベースのキャラクタ・セットと、NLS データに使われる各国文字セットの 2 つのキャラクタ・セットをサポートしています。データ型 `NCHAR` と `NVARCHAR2` は、各国キャラクタ・セットから構成される文字列を格納できます。

NCHAR

`NCHAR` データ型は、固定長 (必要に応じて空白埋めされる) の NLS 文字データを格納するために使います。データの内部表現形式は、各国キャラクタ・セットで US7ASCII などの固定幅のコード化を使用するか、それとも JA16SJIS などの可変幅のコード化を使用するかによって異なります。

`NCHAR` データ型はオプション・パラメータを使って、その最大長を 32767 バイトまで指定できます。次に構文を示します。

```
NCHAR[(maximum_length)]
```

最大長には 1 ~ 32767 の範囲の整数リテラルを指定します。定数や変数は指定できません。

最大長を指定しない場合のデフォルト値は 1 です。最大長を指定する方法は、各国キャラクタ・セットごとに違います。固定幅のキャラクタ・セットの場合、最大長は文字数で指定します。可変幅のキャラクタ・セットは、バイト数で最大長を指定します。次の例では、キャラクタ・セットは JA16EUCFIXED (固定幅) なので、最大長を文字数で指定します。

```
my_string NCHAR(100); -- maximum length is 100 characters
```

`NCHAR` データベース列の最大幅は 2000 バイトです。このため、2000 バイトよりも長い `NCHAR` 値を `NCHAR` 型の列に挿入できません。固定幅、マルチバイト・キャラクタ・セットの場合、2000 バイトに入る文字数より長い `NCHAR` 値は挿入できないことに注意してください。

`NCHAR` 値が `NCHAR` 列の定義された幅より短ければ、Oracle は定義幅まで値を空白で埋めます。 `CHAR` 値は `NCHAR` 列に挿入できません。同様に、`NCHAR` 値は `CHAR` 列に挿入できません。

NVARCHAR2

NVARCHAR2 データ型は可変長の NLS 文字データを格納するのに使います。データの内部表現形式は、各国キャラクタ・セットで WE8EBCDIC37C などの固定幅のコード化を使用するか、それとも JA16DBCS などの可変幅のコード化を使用するかによって異なります。

NVARCHAR2 データ型は必須パラメータを使って、その最大長を 32767 バイトまで指定できます。次に構文を示します。

```
NVARCHAR2 (maximum_length)
```

最大長には 1 ~ 32767 の範囲の整数リテラルを指定します。定数や変数は指定できません。

最大長を指定する方法は、各国キャラクタ・セットごとに違います。固定幅のキャラクタ・セットの場合、最大長は文字数で指定します。可変幅のキャラクタ・セットは、バイト数で最大長を指定します。次の例では、キャラクタ・セットは JA16SJIS (可変幅) なので、最大長をバイト数で指定します。

```
my_string NVARCHAR2(200); -- maximum length is 200 bytes
```

NVARCHAR2 データベース列の最大幅は 4000 バイトです。このため、4000 バイトよりも長い NVARCHAR2 値を NVARCHAR2 型の列に挿入できません。固定幅、マルチバイト・キャラクタ・セットの場合、4000 バイトに入る文字数より長い NVARCHAR2 値は挿入できないことに注意してください。

VARCHAR2 値は NVARCHAR2 列に挿入できません。同様に、NVARCHAR2 値は VARCHAR2 列に挿入できません。

LOB 型

LOB (ラージ・オブジェクト) データ型 BFILE、BLOB、CLOB、および NCLOB は、構造化されていないデータ (テキスト、図形イメージ、ビデオ・クリップ、サウンド・ウェーブ形式など) のブロックを、4GB まで格納するために使います。さらに、効率的、かつランダムで、断片的なデータへのアクセスができます。

LOB 型は、いくつかの点で LONG 型と LONG RAW 型とは違います。たとえば、LOB (NCLOB を除く) はオブジェクト型の属性として使えますが、LONG は使えません。LOB の最大サイズは 4GB ですが、LONG の最大サイズは 2GB です。また、LOB ではデータのランダム・アクセスがサポートされていますが、LONG では順次アクセスしかサポートされていません。

LOB 型には LOB ロケータと呼ばれる値が入れられます。これは、外部ファイル、インライン (行内部) またはライン外 (行外部) に格納される大きなオブジェクトの位置を指定するものです。BLOB 型、CLOB 型、NCLOB 型、または BFILE 型のデータベース列にはロケータが格納されます。BLOB データ、CLOB データ、および NCLOB データは、データベース内の行内部または行外部に格納されます。BFILE データは、データベース外のオペレーティング・システム・ファイルに格納されます。

PL/SQL の LOB は、ロケータによって操作されます。たとえば、BLOB 列の値を取り出した場合、ロケータだけが戻されます。複数のトランザクションまたはセッションにまたがるロケータは使えません。そのため、あるトランザクションまたはセッション中に PL/SQL 変数でロケータを保管してからそれを別のトランザクションまたはセッションで使うことはできません。LOB を操作するには、提供されているパッケージ DBMS_LOB を使います。LOB およびパッケージ DBMS_LOB の詳細は、『Oracle8i アプリケーション開発者ガイド ラージ・オブジェクト』を参照してください。

BFILE

BFILE データ型は、データベース外のオペレーティング・システム・ファイルに大規模なバイナリ・オブジェクトを格納するのに使います。どの BFILE 変数にも、サーバー上の大規模なバイナリ・ファイルを指し示すファイル・ロケータが格納されています。ロケータには、フル・パス名を指定するディレクトリ別名が含まれています（論理パス名はサポートされていません）。

BFILE は読み専用です。修正を加えることはできません。BFILE のサイズはシステムに依存していますが、4GB (2³² - 1 バイト) を超えるものは使えません。指定された BFILE が存在し、Oracle にその読み許可があることは、DBA によって保証されます。基礎となるオペレーティング・システムがファイルの整合性を維持します。

BFILE はトランザクションには関与せず、回復可能ではなく、レプリケートできません。オープンする BFILE の最大数は、システムに依存する Oracle 初期化パラメータ SESSION_MAX_OPEN_FILES により設定されます。

BLOB

BLOB データ型は、データベース内の行内部または行外部に大規模なバイナリ・オブジェクトを格納するのに使用します。どの BLOB 変数にも、大規模なバイナリ・オブジェクトを指し示すロケータが格納されます。BLOB のサイズは 4GB 以下でなければなりません。

BLOB はトランザクションに完全に関与し、回復可能で、レプリケートできます。パッケージ DBMS_LOB または OCI が加える変更は、コミットすることもロールバックすることもできます。しかし、複数のトランザクションまたはセッションにまたがる BLOB ロケータは使えません。

CLOB

CLOB データ型は、シングルバイト文字データの大規模なブロックを、データベース内の行内部または行外部に格納するのに使います。固定幅と可変幅の、両方のキャラクタ・セットがサポートされています。どの CLOB 変数にも、文字データの大規模なブロックを指し示すロケータが格納されます。CLOB のサイズは 4GB 以下でなければなりません。

CLOB はトランザクションに完全に関与し、回復可能で、レプリケートできます。パッケージ DBMS_LOB または OCI が加える変更は、コミットすることもロールバックすることもできます。しかし、複数のトランザクションまたはセッションにまたがる CLOB ロケータは使えません。

NCLOB

NCLOB データ型は、マルチバイトの NCHAR データの大規模なブロックを、データベース内の行内部または行外部に格納するのに使います。固定幅と可変幅の、両方のキャラクタ・セットがサポートされています。どの NCLOB 変数にも、NCHAR データの大規模なブロックを指し示すロケータが格納されます。NCLOB のサイズは 4GB 以下でなければなりません。

NCLOB はトランザクションに完全に關与し、回復可能で、レプリケートできます。パッケージ DBMS_LOB または OCI が加える変更は、コミットすることもロールバックすることもできます。しかし、複数のトランザクションまたはセッションにまたがる NCLOB ロケータは使えません。

その他の型

次の型は、論理値および日付 / 時刻値を格納したり、操作したりするのに使います。

BOOLEAN

BOOLEAN データ型は、論理値 TRUE と FALSE、および NULL (存在しない値、未知の値、または適用不可能な値を表す) を格納するのに使います。BOOLEAN 変数で可能な操作は論理操作だけです。

BOOLEAN データ型はパラメータを取りません。BOOLEAN 変数に代入できるのは、値 TRUE と FALSE、および NULL だけです。データベース列に値 TRUE や FALSE を挿入できません。また、列の値を選択またはフェッチして BOOLEAN 変数に入れることはできません。

DATE

DATE データ型は固定長の日付 / 時刻値を格納するために使います。DATE 値には、時刻を表す午前 0 時からの秒数が入れられます。デフォルトでは、日付部分は現在の月の最初の日であり、時刻の部分は午前 0 時です。日付関数 SYSDATE は、現在の日付と時刻を戻します。

使える日付は、紀元前 4712 年 1 月 1 日から西暦 9999 年 12 月 31 日までです。ユリウス暦日付は、紀元前 4721 年 1 月 1 日からの日数です。ユリウス暦日付によって、共通の参照元からの連続した日付が可能になります。日付関数 TO_DATE と TO_CHAR で日付形式モデル 'J' を使うと、DATE 値とそれに対応するユリウス暦日付の値の間で変換できます。

日付式の中では、デフォルトの日付形式の文字値は自動的に DATE 値に変換されます。デフォルトのデータ形式は、Oracle 初期化パラメータ NLS_DATE_FORMAT によって設定されます。たとえば、デフォルトは 'DD-MON-YY' であり、これは 2 桁数字の日、月の省略名、年数の下 2 桁を含むものということです。

ユーザー定義のサブタイプ

日付に対しては加算および減算ができます。たとえば、次の文はある従業員が雇用されてからの日数を戻します。

```
SELECT SYSDATE - hiredate INTO days_worked FROM emp
WHERE empno = 7499;
```

算術式の中では、整数リテラルが日数として解釈されます。たとえば、SYSDATE + 1 は、次の日ということです。

日付ファンクションおよび書式モデルの詳細は、『Oracle8i SQL リファレンス』を参照してください。

ユーザー定義のサブタイプ

PL/SQL の基本型はそれぞれ、その型の項目に適用可能な値のセットや演算のセットを指定します。サブタイプは、その基本型と同じ演算のセットを指定しますが、指定する値は基本型のサブセットだけです。つまり、サブタイプは新しい型を導入するためのものではありません。単にその基本型に対してオプションの制約を定義するためのものです。

サブタイプを使うと、信頼性が向上し、ANSI/ISO 型との互換性が保たれ、さらに定数や変数の使用意図を示すことで読みやすさが向上します。PL/SQL では、パッケージ STANDARD の中にいくつかのサブタイプが前もって定義されています。たとえば、PL/SQL では、サブタイプ CHARACTER が次のようにあらかじめ定義されています。

```
SUBTYPE CHARACTER IS CHAR;
```

サブタイプ CHARACTER は、その基本型 CHAR と同じ値の集合を指定します。したがって、CHARACTER は無制約のサブタイプです。

サブタイプの定義

ユーザー独自のサブタイプは、次の構文を使って、任意の PL/SQL ブロックまたはサブプログラム、パッケージの宣言部で定義できます。

```
SUBTYPE subtype_name IS base_type [NOT NULL];
```

subtype_name は、それ以降の宣言の中で使われる型指定子です。base_type はスカラーまたはユーザー定義の PL/SQL 型です。base_type の指定では、%TYPE を使って、変数またはデータベース列のデータ型を指定できます。また、%ROWTYPE を使って、カーソル、カーソル変数、またはデータベース表の行の型を指定できます。次に例を示します。

```
DECLARE
    SUBTYPE BirthDate IS DATE NOT NULL; -- based on DATE type
    SUBTYPE Counter IS NATURAL;         -- based on NATURAL subtype
    TYPE NameList IS TABLE OF VARCHAR2(10);
    SUBTYPE DutyRoster IS NameList;      -- based on TABLE type
    TYPE TimeRec IS RECORD (minutes INTEGER, hours INTEGER);
```

```

SUBTYPE FinishTime IS TimeRec;           -- based on RECORD type
SUBTYPE ID_Num IS emp.empno%TYPE;        -- based on column type
CURSOR c1 IS SELECT * FROM dept;
SUBTYPE DeptFile IS c1%ROWTYPE;          -- based on cursor rowtype

```

しかし、基本型にはサイズ制約を指定できません。たとえば、次の定義は不正です。

```

DECLARE
  SUBTYPE Accumulator IS NUMBER(7,2); -- illegal; must be NUMBER
  SUBTYPE Delimiter IS CHAR(1);       -- illegal; must be CHAR

```

サイズ制約付きのサブタイプを直接定義できません。簡単な代替方法として、間接的に定義する方法があります。サイズ制約付きの変数を宣言し、次に %TYPE を使ってそのデータ型を指定します。次に例を示します。

```

DECLARE
  temp VARCHAR2(15);
  SUBTYPE Word IS temp%TYPE; -- maximum size of Word is 15

```

同様に、%TYPE を使ってサブタイプを定義し、データベース列のデータ型を指定すると、そのサブタイプは列のサイズ制約（定義されている場合）を継承します。しかし、NOT NULL のような他の種類の制約は継承しません。

サブタイプの使用

いったんサブタイプを定義すると、その型の項目を宣言できます。次の例では、Counter 型の変数を宣言しています。変数の使用意図がサブタイプ名によってどのように示されているかに注意してください。

```

DECLARE
  SUBTYPE Counter IS NATURAL;
  rows Counter;

```

ユーザー定義のサブタイプに属する変数を宣言する場合に、そのサブタイプについて制約を設定できます。次に例を示します。

```

DECLARE
  SUBTYPE Accumulator IS NUMBER;
  total Accumulator(7,2);

```

サブタイプを使うと、範囲外の値を検出できるため信頼性が向上します。次の例では、-9 ~ 9 の範囲の整数を格納するようにサブタイプ `Scale` を制限しています。プログラムで `Scale` 変数の範囲外の数値を格納しようとする、PL/SQL は例外を呼び出します。

```
DECLARE
    temp NUMBER(1,0);
    SUBTYPE Scale IS temp%TYPE;
    x_axis Scale; -- magnitude range is -9 .. 9
    y_axis Scale;
BEGIN
    x_axis := 10; -- raises VALUE_ERROR
```

データ型の互換性

無制約のサブタイプは、その基本型と互換性があります。たとえば、次のように宣言すると、`amount` の値を変換せずに `total` に代入できます。

```
DECLARE
    SUBTYPE Accumulator IS NUMBER;
    amount NUMBER(7,2);
    total Accumulator;
BEGIN
    ...
    total := amount;
```

異なるサブタイプでも、基本型が同じならば互換性があります。たとえば、次のように宣言すると、`finished` の値を `debugging` に代入できます。

```
DECLARE
    SUBTYPE Sentinel IS BOOLEAN;
    SUBTYPE Switch IS BOOLEAN;
    finished Sentinel;
    debugging Switch;
BEGIN
    ...
    debugging := finished;
```

また、異なるサブタイプの場合、基本型が同じデータ型の系列ならば互換性があります。たとえば、次のように宣言すると、`verb` の値を `sentence` に代入できます。

```
DECLARE
    SUBTYPE Word IS CHAR;
    SUBTYPE Text IS VARCHAR2;
    verb Word;
    sentence Text;
BEGIN
    ...
    sentence := verb;
```

データ型の変換

あるデータ型の値を別のデータ型に変換することが必要な場合があります。たとえば ROWID を調べるためには、これを文字列に変換しなければなりません。データ型の変換について PL/SQL では、明示的な変換と暗黙的（自動的）な変換がサポートされています。

明示的な変換

値のデータ型を別のデータ型に変換するには、組み込みファンクションを使います。たとえば、CHAR 値を DATE 値または NUMBER 値に変換するには、それぞれ TO_DATE ファンクションまたは TO_NUMBER ファンクションを使います。逆に、DATE 値または NUMBER 値を CHAR 値に変換するには、TO_CHAR ファンクションを使います。これらのファンクションの詳細は、『Oracle8i SQL リファレンス』を参照してください。

暗黙的な変換

変換して意味がある場合、PL/SQL は暗黙的にデータ型の変換を実行することがあります。この機能によって、ある型のリテラルや変数、パラメータなどを別の型が期待されている箇所で使用可能になります。次の例では、CHAR 型変数である start_time と finish_time に、午前 0 時からの秒数を表す文字列値が保持されています。これらの値の差を NUMBER 型変数の elapsed_time に代入する必要があります。ここで PL/SQL は CHAR 値を NUMBER 値に自動的に変換します。

```
DECLARE
    start_time    CHAR(5);
    finish_time   CHAR(5);
    elapsed_time  NUMBER(5);
BEGIN
    /* Get system time as seconds past midnight. */
    SELECT TO_CHAR(SYSDATE, 'SSSSS') INTO start_time FROM sys.dual;
    -- do something
    /* Get system time again. */
    SELECT TO_CHAR(SYSDATE, 'SSSSS') INTO finish_time FROM sys.dual;
    /* Compute elapsed time in seconds. */
    elapsed_time := finish_time - start_time;
    INSERT INTO results VALUES (elapsed_time, ...);
END;
```

選択された列値を変数に代入する前に、PL/SQL は必要に応じて変換元の列のデータ型を変数のデータ型に適合するよう変換します。DATE 列の値を選択して VARCHAR2 変数に入れる場合がこれに該当します。

同様に、変数の値をデータベース列に代入する前に、PL/SQL は必要に応じてその値を変数のデータ型からターゲット列のデータ型に変換します。PL/SQL では、どのような暗黙的変換が必要なのかが決定できない場合、コンパイル・エラーが発生します。このような場合、ユーザーはデータ型変換ファンクションを使う必要があります。表 2-1 は PL/SQL が実行できる暗黙的変換を示しています。

表 2-1 暗黙的な変換

	BIN_INT	CHAR	DATE	LONG	NUMBER	PLS_INT	RAW	UROWID	VARCHAR2
BIN_INT		X		X	X	X			X
CHAR	X		X	X	X	X	X	X	X
DATE		X		X					X
LONG		X					X		X
NUMBER	X	X		X		X			X
PLS_INT	X	X		X	X				X
RAW		X		X					X
UROWID		X							X
VARCHAR2	X	X	X	X	X	X	X	X	

値が実際に変換可能であるかどうかは、ユーザーが自分で確認する必要があります。たとえば、'02-JUN-92' という CHAR 型の値は DATE 型に変換できますが、'YESTERDAY' という CHAR 型の値は DATE 型に変換できません。同様に、英字を含んでいる VARCHAR2 型の値は NUMBER 値に変換できません。

暗黙的変換と明示的変換

一般的に、データ型の暗黙的変換に頼ってしまうようなプログラミング習慣は、パフォーマンスの悪化と将来ソフトウェアが仕様変更されるかもしれないことを考えると、あまり好ましくありません。また、暗黙的変換は実行されるときに条件に左右されやすく、結果を常に予測できるとは限りません。したがって、なるべくデータ型変換ファンクションを使うようにしてください。そうすることによって、アプリケーションの信頼性とメンテナンスの容易性を高めることができます。

DATE の値

DATE 列の値を選択して CHAR 型変数または VARCHAR2 型変数に入れる場合、PL/SQL は内部バイナリ値を文字値に変換しなければなりません。このとき、PL/SQL は、文字列をデフォルトの日付フォーマットで戻す関数 `TO_CHAR` をコールします。時刻やユリウス暦日付などの情報を得るには、書式マスクを使って `TO_CHAR` をコールする必要があります。

CHAR 型または VARCHAR2 型の値を DATE 型の列に挿入する場合にも変換が必要です。PL/SQL は、デフォルトの日付フォーマットを期待する関数 `TO_DATE` をコールします。他の書式の日付を挿入するには、書式マスクを使って `TO_DATE` を明示的にコールする必要があります。

RAW および LONG RAW の値

RAW 型または LONG RAW 型の列の値を選択して CHAR 型変数または VARCHAR2 型変数に入れる場合、PL/SQL は内部バイナリ値を文字値に変換しなければなりません。この場合、PL/SQL は RAW 型データまたは LONG RAW 型データの各バイナリ・バイトを文字のペアとして戻します。個々の文字はニブル（半バイト）の 16 進値を表します。たとえば、PL/SQL はバイナリ・バイト 11111111 を文字のペア 'FF' として戻します。関数 `RAWTOHEX` も同じ変換を実行します。

CHAR 型または VARCHAR2 型の値を RAW 型または LONG RAW 型の列に挿入する場合にも変換が必要です。変数中の文字のペアは、いずれもバイナリ・バイトの 16 進値を表していなければなりません。どちらかの文字がニブルの 16 進値を表していない場合は、PL/SQL は例外を呼び出します。

NLS 値

大文字のキャラクタ・セット名が渡されると、組み込み関数 `NLS_CHARSET_ID` は対応するキャラクタ・セットの ID 番号を戻します。逆に、キャラクタ・セットの ID 番号が渡されると、関数 `NLS_CHARSET_NAME` は対応するキャラクタ・セット名を戻します。

値 'CHAR_CS' または 'NCHAR_CS' を `NLS_CHARSET_ID` に渡すと、それぞれデータベースまたは各国キャラクタ・セットの ID 番号が戻されます。キャラクタ・セット名のリストは、『Oracle8i NLS ガイド』にあります。

宣言

プログラムは、変数と定数に値を格納します。プログラムの実行中に、変数の値を変更できますが、定数の値は変更できません。

変数および定数は、任意の PL/SQL ブロック、サブプログラム または パッケージの宣言部で宣言できます。宣言によって、値の記憶領域を割り当て、データ型を指定し、値を参照できるように記憶位置の名前を決めます。

次に例を示します。

```
birthday DATE;  
emp_count SMALLINT := 0;
```

1 つめの宣言で、DATE 型の変数の名前を決めています。2 つめの宣言で、SMALLINT 型の変数の名前を決め、代入演算子を使って変数に初期値 0 を代入しています。

代入演算子の後に続く式は複雑なものでもかまいません。また、事前に初期化されている変数を参照することもできます。

```
pi      REAL := 3.14159;  
radius REAL := 1;  
area    REAL := pi * radius**2;
```

デフォルトでは、変数は NULL に初期化されます。このため、これらの宣言は次のものと等価になります。

```
birthday DATE;  
birthday DATE := NULL;
```

定数の宣言では、型指定子の前にキーワード CONSTANT が必要です。次に例を示します。

```
credit_limit CONSTANT REAL := 5000.00;
```

この宣言では、REAL 型の定数の名前を決め、定数に初期値 5000 を代入しています（初期値は最終的な値でもあります）。定数は、宣言の中で初期化しなければなりません。そうしないと、コンパイラが宣言を処理するときにコンパイル・エラーが発生します。（PL/SQL コンパイラによる宣言の処理はエラボレーションと呼ばれます。）

DEFAULT の使用方法

変数の初期化には、代入演算子のかわりにキーワード `DEFAULT` も使えます。たとえば、次の宣言は、

```
blood_type CHAR := 'O';
```

次のように書き換えることができます。

```
blood_type CHAR DEFAULT 'O';
```

標準的な値を持つ変数には、`DEFAULT` を使用します。特殊な値を持つ変数（カウンタやアキュムレータ）には、代入演算子を使用します。次に例を示します。

```
hours_worked INTEGER DEFAULT 40;  
employee_count INTEGER := 0;
```

`DEFAULT` を使って、サブプログラム・パラメータおよびカーソル・パラメータ、ユーザー定義レコードのフィールドを初期化することもできます。

NOT NULL の使用方法

宣言によって、初期値を代入する以外に `NOT NULL` 制約を付けることもできます。次に例を示します。

```
acct_id INTEGER(4) NOT NULL := 9999;
```

`NOT NULL` と定義されている変数には `NULL` を代入できません。`NULL` を代入しようとすると、PL/SQL は事前定義の例外 `VALUE_ERROR` を呼び出します。`NOT NULL` 制約の後に初期化句が続かなければなりません。たとえば、次の宣言は誤りです。

```
acct_id INTEGER(5) NOT NULL; -- illegal; not initialized
```

サブタイプ `NATURALN` と `POSITIVEN` は、あらかじめ `NOT NULL` と定義されています。たとえば、次に示す宣言は等価です。

```
emp_count NATURAL NOT NULL := 0;  
emp_count NATURALN := 0;
```

`NATURALN` 宣言および `POSITIVEN` 宣言では、型指定子の後に初期化句が続かなければなりません。そうでない場合、コンパイル・エラーが発生します。たとえば、次の宣言は誤りです。

```
line_items POSITIVEN; -- illegal; not initialized
```

%TYPE の使用方法

%TYPE 属性は、変数またはデータベース列のデータ型を与えます。次の例では、%TYPE 属性は変数のデータ型を与えています。

```
credit REAL(7,2);
debit  credit%TYPE;
```

%TYPE 属性を使って宣言された変数は、データ型指定子を使って宣言された変数と同じように扱われます。たとえば上の宣言では、PL/SQL は debit を REAL(7,2) 型の変数として扱います。次の例に示すように、%TYPE 属性を使った宣言は初期化句を含むことができます。

```
balance          NUMBER(7,2);
minimum_balance balance%TYPE := 10.00;
```

%TYPE 属性は、データベース列を参照する変数を宣言する場合に特に便利です。次の例のように、表や列を参照したり、所有者、表、列を参照したりできます。

```
my_dname scott.dept.dname%TYPE;
```

%TYPE 属性を使って my_dname を宣言することには 2 つの利点があります。第一に、ユーザーは dname の正確なデータ型を知る必要がありません。次に、dname のデータベース定義が変更された場合でも、my_dname のデータ型は実行時にそれに対応して変更されます。

ただし、%TYPE 変数は NOT NULL 列制約を継承しません。次の例では、データベース列 empno が NOT NULL として定義されていても、変数 my_empno に NULL を代入できます。

```
DECLARE
    my_empno emp.empno%TYPE;
    ...
BEGIN
    my_empno := NULL;  -- this works
```

%ROWTYPE の使用方法

%ROWTYPE 属性は、表（またはビュー）の中の行を表すレコードを与えます。レコードには、表から選択された行全体のデータを格納することも、カーソルまたはタイプしたカーソル変数で取り出された行全体のデータを格納することもできます。次の例では、2 つのレコードを宣言しています。1 つめのレコードには表 emp から選択された行が格納されます。2 つめのレコードには、カーソル c1 で取り出された行が格納されます。

```
DECLARE
    emp_rec emp%ROWTYPE;
    CURSOR c1 IS SELECT deptno, dname, loc FROM dept;
    dept_rec c1%ROWTYPE;
```

行の中の列と、それに対応するレコード中のフィールドは、同じ名前と同じデータ型を持ちます。ただし、%ROWTYPE レコードのフィールドは NOT NULL 列制約を継承しません。

次の例では、列の値を選択して emp_rec という名前のレコードに入れています。

```
BEGIN
    SELECT * INTO emp_rec FROM emp WHERE ...
```

SELECT 文によって戻された列の値がフィールドに格納されます。フィールドを参照する場合は、ドット表記法を使います。たとえば、フィールド deptno は次のように参照できます。

```
IF emp_rec.deptno = 20 THEN ...
```

また、式の値を特定のフィールドに代入できます。次に例を示します。

```
emp_rec.ename := 'JOHNSON';
emp_rec.sal := emp_rec.sal * 1.15;
```

最後の例では、%ROWTYPE を使ってパッケージされたカーソルを定義します。

```
CREATE PACKAGE emp_actions AS
    CURSOR c1 RETURN emp%ROWTYPE; -- declare cursor specification
    ...
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
    CURSOR c1 RETURN emp%ROWTYPE IS -- define cursor body
        SELECT * FROM emp WHERE sal > 3000;
    ...
END emp_actions;
```

一括代入

%ROWTYPE 属性を使った宣言は、初期化句を含むことができません。しかし、レコード中のすべてのフィールドに一度に値を代入する方法が2つあります。1番目の方法として、PL/SQL では、2つのレコードの宣言が同じ表またはカーソルを参照している場合に、その2つのレコード全体の間での一括代入ができます。たとえば、次の代入は有効です。

```
DECLARE
    dept_rec1 dept%ROWTYPE;
    dept_rec2 dept%ROWTYPE;
    CURSOR c1 IS SELECT deptno, dname, loc FROM dept;
    dept_rec3 c1%ROWTYPE;
BEGIN
    ...
    dept_rec1 := dept_rec2;
```

次の代入は、dept_rec2 が表に基づき、dept_rec3 がカーソルに基づいているため、誤りになります。

```
dept_rec2 := dept_rec3; -- illegal
```

2 番目の方法として、次の例に示すように、SELECT 文または FETCH 文を使って列の値のリストをレコードに代入できます。列名の順番は、CREATE TABLE 文または CREATE VIEW 文で定義された順番でなければなりません。

```
DECLARE
    dept_rec dept%ROWTYPE;
    ...
BEGIN
    SELECT deptno, dname, loc INTO dept_rec FROM dept
        WHERE deptno = 30;
```

しかし、代入文を使って列の値のリストをレコードに代入できません。このため、次の構文は誤りです。

```
record_name := (value1, value2, value3, ...); -- illegal
```

レコード全体を取り出すことはできますが、挿入または更新はできません。たとえば、次の文は誤りです。

```
INSERT INTO dept VALUES (dept_rec); -- illegal
```

別名の使用

カーソルによって %ROWTYPE 属性を使って取り出された選択リスト項目は、単純名を持たなければなりません。また、選択リスト項目が式の場合は別名を持たなければなりません。次の例では、wages という別名を使っています。

```
-- available online in file 'examp4'
DECLARE
    CURSOR my_cursor IS
        SELECT sal + NVL(comm, 0) wages, ename FROM emp;
    my_rec my_cursor%ROWTYPE;
BEGIN
    OPEN my_cursor;
    LOOP
        FETCH my_cursor INTO my_rec;
        EXIT WHEN my_cursor%NOTFOUND;
        IF my_rec.wages > 2000 THEN
            INSERT INTO temp VALUES (NULL, my_rec.wages, my_rec.ename);
        END IF;
    END LOOP;
    CLOSE my_cursor;
END;
```

制限

PL/SQL では前方参照ができません。宣言文などの他の文で変数または定数を参照するときは、事前に宣言しておく必要があります。たとえば、次に示す `maxi` の宣言は誤りです。

```
maxi INTEGER := 2 * mini; -- illegal
mini INTEGER := 15;
```

しかし、PL/SQL ではサブプログラムの方前宣言ができます。詳細は、7-9 ページの「[前方宣言](#)」を参照してください。

言語によっては、同一のデータ型の複数の変数の並びを一度に宣言できます。しかし、PL/SQL ではそれができません。たとえば、次の宣言は誤りです。

```
i, j, k SMALLINT; -- illegal
```

正しい宣言を次に示します。

```
i SMALLINT;
j SMALLINT;
k SMALLINT;
```

命名規則

定数、変数、カーソル、カーソル変数、例外、プロシージャ、ファンクション、パッケージなどの PL/SQL プログラム項目には、いずれも同じ命名規則が適用されます。名前には単純名または修飾名、リモート名、修飾リモート名があります。たとえば、プロシージャ名 `raise_salary` は次のように使えます。

```
raise_salary(...);           -- simple
emp_actions.raise_salary(...); -- qualified
raise_salary@newyork(...);    -- remote
emp_actions.raise_salary@newyork(...); -- qualified and remote
```

1 番目の例ではプロシージャ名をそのまま使っています。2 番目の例では、プロシージャが `emp_actions` という名前のパッケージに格納されているため、ドット表記法を使って名前を修飾しなければなりません。3 番目の例では、プロシージャがリモート・データベースに格納されているため、リモート・アクセスの標識 (@) を使ってデータベース・リンク `newyork` を参照しています。4 番目の例では、プロシージャ名を修飾し、データベース・リンクの参照も行っています。

シノニム

表、シーケンス、ビュー、スタンドアロン・サブプログラム、パッケージなどのリモート・スキーマ・オブジェクトについて、位置を知らなくても指定できるように、シノニムを作成できます。ただし、サブプログラムやパッケージの中で宣言された項目については、シノニムを作成できません。これには、定数および変数、カーソル、カーソル変数、例外、パッケージ化されたサブプログラムが該当します。

有効範囲

同じ有効範囲の中では、宣言された識別子はすべて他と重複しないものでなければなりません。このため、変数と定数はデータ型が異なる場合でも同じ名前を共有できません。たとえば、次に示す2つの宣言は誤りです。

```
DECLARE
    valid_id BOOLEAN;
    valid_id VARCHAR2(5); -- illegal duplicate identifier
    FUNCTION bonus (valid_id IN INTEGER) RETURN REAL IS ...
                        -- illegal triplicate identifier
```

識別子に適用される有効範囲の規則については、2-35 ページの「[有効範囲と可視性](#)」を参照してください。

大文字 / 小文字の区別

定数および変数、パラメータの名前では、すべての識別子と同様に大文字と小文字が区別されません。たとえば、PL/SQL は次の名前を同じものとみなします。

```
DECLARE
    zip_code INTEGER;
    Zip_Code INTEGER; -- same as zip_code
```

ネーム変換

潜在的にあいまいな SQL 文では、データベース列の名前はローカル変数名および仮パラメータ名より優先されます。たとえば、次の例では、WHERE 句の2つの `ename` がいずれもデータベース列を参照しているとみなされるため、DELETE 文により 'KING' だけでなくすべての雇用者が `emp` 表から削除されてしまいます。

```
DECLARE
    ename VARCHAR2(10) := 'KING';
BEGIN
    DELETE FROM emp WHERE ename = ename;
```

このような場合は、重複を避けるため、次のように、ローカル変数と仮パラメータに `my_` という接頭辞を付けます。

```
DECLARE
    my_ename VARCHAR2(10);
```

あるいは、ブロック・ラベルを使って参照を修飾します。

```
<<main>>
DECLARE
    ename VARCHAR2(10) := 'KING';
BEGIN
    DELETE FROM emp WHERE ename = main.ename;
```

次の例では、ローカル変数と仮パラメータへの参照を、サブプログラム名を使って修飾しています。

```
FUNCTION bonus (deptno IN NUMBER, ...) RETURN REAL IS
    job CHAR(10);
BEGIN
    SELECT ... WHERE deptno = bonus.deptno AND job = bonus.job;
```

ネーム変換の詳細は、[付録 D](#) を参照してください。

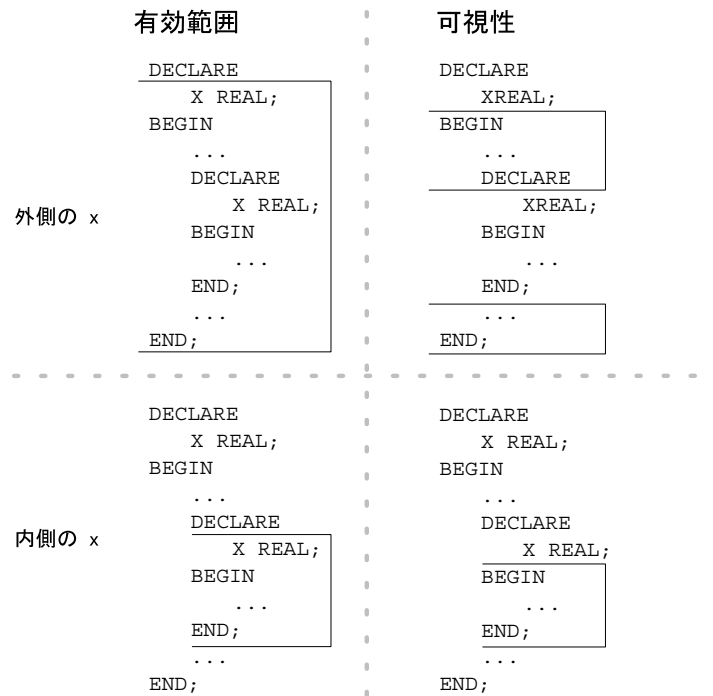
有効範囲と可視性

識別子に対する参照は、その有効範囲と可視性に従って解決されます。識別子の有効範囲とは、その識別子の参照が可能な、プログラム・ユニット（ブロック、サブプログラムまたはパッケージ）の領域です。識別子は、未修飾の名前で識別子を参照できる領域からだけ、可視の状態にあります。[図 2-2](#) は、`x` という名前の変数の有効範囲と可視性を示します。この変数は囲みブロックで宣言されてからサブブロックで再宣言されます。

ある PL/SQL ブロックで宣言された識別子は、そのブロックに対してはローカルであり、そのサブブロックすべてに対してはグローバルです。グローバル識別子がサブブロックの中で再定義されると、両方の識別子がある有効範囲内にあることになります。しかし、サブブロックの中でグローバル識別子を参照する場合は修飾名が必要なので、可視であるのはローカル識別子だけです。

同じブロックで識別子を 2 度宣言できませんが、同じ識別子を 2 つの異なるブロックで宣言できます。識別子が表す 2 つの項目は区別され、一方を変更しても他方には影響がありません。しかし、あるブロックから、同じレベルの他のブロックで宣言されている識別子への参照はできません。そのような識別子は、そのブロックに対してローカルでもグローバルでもないためです。

図 2-2 有効範囲と可視性



次の例は、有効範囲の規則を示すものです。あるサブブロックで宣言された識別子は、別のサブブロックで参照できないことに注意してください。これは、あるブロックと同じレベルでネストされた他のブロックで宣言された識別子を、そのブロックで参照できないためです。

```

DECLARE
  a CHAR;
  b REAL;
BEGIN
  -- identifiers available here: a (CHAR), b
  DECLARE
    a INTEGER;
    c REAL;
  BEGIN
    -- identifiers available here: a (INTEGER), b, c
  END;
  DECLARE
    d REAL;

```



```
BEGIN
  -- identifiers available here: a (CHAR), b, d
END;
  -- identifiers available here: a (CHAR), b
END;
```

グローバル識別子はサブブロックで再宣言でき、その場合はローカルな宣言が優先され、サブブロックでは、修飾名を使わないとグローバル識別子を参照できません。次の例に示すように、外側のブロックのラベルを修飾子として使えます。

```
<<outer>>
DECLARE
  birthdate DATE;
BEGIN
  DECLARE
    birthdate DATE;
  BEGIN
    ...
    IF birthdate = outer.birthdate THEN ...
```

また、次の例に示すように、外側のサブプログラムの名前を修飾子として使えます。

```
PROCEDURE check_credit (...) IS
  rating NUMBER;
  FUNCTION valid (...) RETURN BOOLEAN IS
    rating NUMBER;
  BEGIN
    ...
    IF check_credit.rating < 3 THEN ...
```

しかし、同一の有効範囲内でラベルとサブプログラムを同じ名前にすることはできません。

代入

変数と定数は、ブロックまたはサブプログラムに入るたびに初期化されます。デフォルトでは、変数は `NULL` に初期化されます。したがって、次の例に示すように、変数を明示的に初期化しない限り、変数の値は未定義です。

```
DECLARE
    count INTEGER;
    ...
BEGIN
    count := count + 1; -- assigns a null to count
```

`count` が `NULL` であるため、代入演算子の右辺の式は `NULL` になります。予期しない結果を避けるため、値を代入する前には変数を参照しないようにしてください。

変数に値を代入する場合は、代入文を使います。たとえば、次の文では変数 `bonus` の古い値を上書きして、新しい値を代入しています。

```
bonus := salary * 0.15;
```

代入演算子の後に続く式は、複雑なものでも構いませんが、そのデータ型は変数のデータ型と同じか、変数のデータ型に変換できるものでなければなりません。

ブール値

ブール変数に代入できるのは、値 `TRUE` と `FALSE`、および `NULL` だけです。たとえば、次のような宣言があるとします。

```
DECLARE
    done BOOLEAN;
```

次の文は有効です。

```
BEGIN
    done := FALSE;
    WHILE NOT done LOOP
        ...
    END LOOP;
```

式に関係演算子を適用するとブール値が戻されます。したがって、次の代入は有効です。

```
done := (count > 500);
```

データベース値

SELECT 文を使っても変数に値を代入できます。選択リスト中の項目ごとに、対応する型互換の変数が INTO リストに存在していなければなりません。たとえば、

```
DECLARE
    my_empno emp.empno%TYPE;
    my_ename emp.ename%TYPE;
    wages    NUMBER(7,2);
BEGIN
    ...
    SELECT ename, sal + comm
        INTO last_name, wages FROM emp
        WHERE empno = emp_id;
```

列の値を選択してブール変数に代入できません。

式と比較

式はオペランドと演算子を使って作ります。オペランドとは、変数または定数、リテラル、ファンクション・コールのことで、式の中の値はオペランドを使って表現します。単純な算術式の例を次に示します。

$-X / 2 + 3$

否定演算子 (-) のような単項演算子は、1 つのオペランドに対して作用します。除算演算子 (/) のような 2 項演算子は、2 つのオペランドに対して作用します。PL/SQL には 3 項演算子はありません。

最も単純な式は変数 1 つで構成され、その変数の値が式の値になります。PL/SQL は、演算子が指定する方法でオペランドの値を組み合わせ、式を評価します (現在の値を得ます)。この結果、常に 1 つの値と 1 つのデータ型が得られます。PL/SQL は、式の内容と、式が使われているコンテキストに基づいてデータ型を決定します。

演算子の優先順位

式の中の演算は、その位置と優先順位に応じて特定の順序で実行されます。表 2-2 は、デフォルトでの演算の順序を、上から順に示します。

表 2-2 演算の順序

演算子	演算
** , NOT	指数、論理否定
+ , -	恒等、否定
* , /	乗算、除算
+ , - , 	加算、減算、連結
= , < , > , <= , >= , <> , != , ~= , ^= ,	比較
IS NULL, LIKE, BETWEEN, IN	
AND	論理積
OR	論理和

優先順位が高い演算子が先に適用されます。たとえば、次の 2 つの式の結果はどちらも 8 になります。これは、除算が加算よりも優先順位が高いためです。同じ優先順位の演算子は、特に順序を考慮せずに適用されます。

```
5 + 12 / 4
12 / 4 + 5
```

カッコを使うと、評価の順序を制御できます。たとえば、次の式ではカッコが演算子のデフォルトの優先順位を上書きするので、式の結果は 11 ではなく 7 になります。

```
(8 + 6) / 2
```

次の例では、最も深くネストされた副式が必ず最初に評価されるので、除算の前に減算が実行されます。

```
100 + (20 / 5 + (7 - 3))
```

次の例のように、カッコが不要な場合でも、わかりやすくするために自由にかっこを使えます。

```
(salary * 0.05) + (commission * 0.25)
```

論理演算子

論理演算子 AND、OR、および NOT は、表 2-3 に示す 3 値論理に従います。AND と OR は 2 項演算子、NOT は単項演算子です。

表 2-3 論理真理値表

X	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	
TRUE	NULL	NULL	TRUE	
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	
FALSE	NULL	FALSE	NULL	
NULL	TRUE	NULL	TRUE	NULL
NULL	FALSE	FALSE	NULL	
NULL	NULL	NULL	NULL	

真理値表からわかるように、AND は、オペランドの両方が TRUE の場合に限り TRUE を戻します。一方、OR は、オペランドの片方が TRUE ならば TRUE を戻します。NOT はオペランドの反対の値（論理否定）を戻します。たとえば、NOT TRUE は FALSE を戻します。

NULL は値を持たないため、NOT NULL は NULL を戻します。つまり、NULL に NOT 演算子を適用しても、その結果は値を持ちません。ここでは注意が必要です。NULL によって予期しない結果になる場合があります。2-46 ページの「[NULL の扱い](#)」を参照してください。

評価の順序

カッコを使って評価の順序を指定しない場合は、演算子の優先順位によって順序が決定されます。次の式を比べてみてください。

NOT (valid AND done) | NOT valid AND done

ブール変数 valid と done がどちらも値 FALSE を持つ場合、1 番目の式の結果は TRUE になります。しかし、2 番目の式では NOT が AND より優先されるので、結果は FALSE になります。したがって、2 番目の式は次の式と等価になります。

(NOT valid) AND done

次の例では、valid の値が FALSE である場合、done の値とは関係なく式全体の結果が FALSE になることに注意してください。

valid AND done

同様に、次の例では、`valid` の値が `TRUE` である場合に、`done` の値とは関係なく式全体の結果が `TRUE` になります。

```
valid OR done
```

短絡評価 論理式を評価するとき、PL/SQL では短絡評価を使います。これにより、PL/SQL は結果が判別できた時点でただちに式の評価を停止します。そのため、評価を続ければエラーになるような式でも書くことができます。次の `OR` 式で考えてみます。

```
DECLARE
    ...
    on_hand  INTEGER;
    on_order INTEGER;
BEGIN
    ..
    IF (on_hand = 0) OR ((on_order / on_hand) < 5) THEN
        ...
    END IF;
END;
```

`on_hand` の値がゼロの場合、左のオペランドは `TRUE` になるので、PL/SQL は右のオペランドを評価する必要がありません。`OR` 演算子を適用する前に両方のオペランドを評価した場合には、右のオペランドはゼロによる除算エラーになります。いずれにせよ、短絡評価に頼るのはあまり好ましくないプログラミング習慣です。

比較演算子

比較演算子は式と式を比較します。結果は常に `TRUE`、`FALSE`、`NULL` のいずれかです。比較演算子は、一般に、SQL データ操作文の `WHERE` と、条件制御文の中で使います。次に例を示します。

```
IF quantity_on_hand > 0 THEN
    UPDATE inventory SET quantity = quantity - 1
    WHERE part_number = item_number;
ELSE
    ...
END IF;
```

関係演算子

関係演算子を使うと、複雑な式を比較できます。次の表に、各演算子の意味を示します。

演算子	意味
=	等しい
<>, !=, ~=, ^=	等しくない
<	より小さい
>	より大きい
<=	より小さいか、等しい
>=	より大きいか、等しい

IS NULL 演算子

IS NULL 演算子は、オペランドが NULL ならばブール値 TRUE を、NULL でなければ FALSE を戻します。NULL が関係する比較は、常に結果が NULL になります。したがって、NULL である (NULL である状態) かどうかをテストする場合は、次の文を使ってはなりません。

```
IF variable = NULL THEN ...
```

かわりに次の文を使います。

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

LIKE 演算子

LIKE 演算子を使うと、文字値をパターンと比較できます。大文字と小文字は区別されます。LIKE は、文字のパターンが一致すればブール値 TRUE を、一致しなければ FALSE を戻します。

LIKE を使ってパターンを比較するために、ワイルドカードと呼ばれる 2 つの特殊な目的の文字を使用できます。アンダースコア () は 1 つの文字を表します。パーセント記号 () はゼロ個以上の文字を表します。たとえば、ename の値が 'JOHNSON' ならば、次の式は TRUE になります。

```
ename LIKE 'J%SON'
```

BETWEEN 演算子

BETWEEN 演算子は、ある値が、指定された範囲に含まれているかどうかをテストします。つまり、" 下限以上、上限以下 " という意味を持ちます。たとえば、次の式は FALSE です。

```
45 BETWEEN 38 AND 44
```

IN 演算子

IN 演算子は、集合のメンバーであるかどうかを調べます。つまり、集合の「いずれかのメンバーと等しい」という意味を持ちます。集合には NULL が含まれていてもかまいませんが、NULL は無視されます。たとえば、次の文では、ename が NULL になっている行は削除されません。

```
DELETE FROM emp WHERE ename IN (NULL, 'KING', 'FORD');
```

さらに、次の形式の式では、

```
value NOT IN set
```

集合に NULL が含まれていると FALSE になります。たとえば次の文では、ename 列が NULL でも 'KING' でもない行が削除されるのではなく、行はいっさい削除されません。

```
DELETE FROM emp WHERE ename NOT IN (NULL, 'KING');
```

連結演算子

連結演算子 (||) は、文字列を他の文字列に連結します。たとえば、次の式は、

```
'suit' || 'case'
```

これは、次の行を戻します。

```
'suitcase'
```

CHAR 型のオペランドだけが渡された場合、連結演算子は CHAR 型の値を戻します。CHAR 型のオペランドだけではない場合は、VARCHAR2 型の値を戻します。

ブール式

PL/SQL では、SQL 文の中でもプロシージャ文の中でも、変数と定数を比較できます。これらの比較はブール式と呼ばれ、関係演算子で区切られた単純式または複合式で構成されます。ブール式は一般に論理演算子 AND、OR および NOT で結合されます。ブール式の結果は常に、TRUE、FALSE、NULL のいずれかになります。

SQL 文の中でブール式を使って、表の中の文が影響を与える列を指定できます。プロシージャ文では、条件制御の基盤としてブール式が使われます。ブール式には、算術式、文字式および日付式の 3 種類があります。

算術式

関係演算子を使って数値を比較し、等しいか等しくないかを判定できます。比較は量によるものです。つまり、片方の数値がより大きな量を表す場合、その数値はより大きいとみなされます。たとえば、次のような代入文があるとします。

```
number1 := 75;  
number2 := 70;
```

次の式は TRUE になります。

```
number1 > number2
```

文字式

文字値を比較して、等しいか等しくないかを判定できます。比較はデータベースのキャラクタ・セットの照合順番に基づいてなされます。照合順番とは、特定の範囲の数値コードが個々の文字に対応しているキャラクタ・セットの内部的な順序のことです。内部的な順序を表す数値が他方の文字より大きい場合、その文字値はより大きいとみなされます。たとえば、次のような代入文があるとします。

```
string1 := 'Kathy';  
string2 := 'Kathleen';
```

次の式は TRUE になります。

```
string1 > string2
```

ただし、文字値を比較する場合には、基本型 CHAR と VARCHAR2 の間にある意味上の違いを考慮しなければなりません。詳細は、[付録 B](#) を参照してください。

日付式

日付も比較できます。比較は時系列によってなされます。つまり、片方の日付がより新しければ、その日付はより大きいとみなされます。たとえば、次のような代入文があるとします。

```
date1 := '01-JAN-91';  
date2 := '31-DEC-90';
```

次の式は TRUE になります。

```
date1 > date2
```

指針

一般に、実数を比較して等しいかどうかを判定することはお薦めしません。実数は近似値として格納されます。このため、たとえば次のような IF 条件は TRUE に評価されない可能性があります。

```
count := 1;
IF count = 1.0 THEN ...
```

比較するときは、カッコを使うことをお薦めします。たとえば次の式で、`100 < tax` はブール値になりますが、これは数値 500 と比較できないため、この式は無効です。

```
100 < tax < 500 -- illegal
```

これをデバッグすれば、次の式になります。

```
(100 < tax) AND (tax < 500)
```

ブール変数はそれ自身が TRUE または FALSE です。つまり、ブール値 TRUE や FALSE と比較するのは冗長です。たとえば変数 `done` が BOOLEAN 型である場合、次の WHILE 文は、

```
WHILE NOT (done = TRUE) LOOP
    ...
END LOOP;
```

次のように単純化できます。

```
WHILE NOT done LOOP
    ...
END LOOP;
```

NULL の扱い

NULL を使う場合は、次の規則を念頭に置くことによって、問題の発生を未然に防ぐことができます。

- NULL が関係する比較は、常に結果が NULL になること
- 論理演算子 NOT を NULL 値に適用すると NULL が戻ること
- 条件制御文において条件が NULL になる場合、関連する一連の文は実行されないこと

次の例では、 x と y が等しくないために一連の文 (sequence_of_statements) が実行されるときと考えるかもしれませんが、NULL は予測不能です。そのため、 x と y が等しいかどうか分かりません。したがって、IF 条件は NULL に評価され、一連の文は実行されずにバイパスされます。

```
x := 5;
y := NULL;
...
IF x != y THEN -- yields NULL, not TRUE
    sequence_of_statements; -- not executed
END IF;
```

次の例では、 a と b が等しいように見えるために一連の文が実行されるときと考えるかもしれませんが、等号条件が成立するかどうかは不定であるため、IF 条件は NULL になり、一連の文は実行されずにバイパスされます。

```
a := NULL;
b := NULL;
...
IF a = b THEN -- yields NULL, not TRUE
    sequence_of_statements; -- not executed
END IF;
```

NOT 演算子

論理演算子 NOT を NULL 値に適用すると NULL が戻ることにご注意ください。このため、次の 2 つの文は必ずしも等価ではありません。

IF $x > y$ THEN		IF NOT $x > y$ THEN
$high := x$;		$high := y$;
ELSE		ELSE
$high := y$;		$high := x$;
END IF;		END IF;

IF 条件が FALSE または NULL になると、ELSE 句内の文の並びが実行されます。 x と y のどちらも NULL でなければ、両方の IF 文で同じ値が $high$ に代入されます。ただし、 x と y のどちらかが NULL の場合、1 番目の IF 文は y の値を $high$ に代入しますが、2 番目の IF 文は x の値を $high$ に代入します。

長さゼロの文字列

PL/SQL は長さがゼロの文字値をすべて NULL とみなします。これには文字ファンクションやブール式によって戻された値が含まれます。たとえば、次の文ではターゲットの変数に NULL を代入します。

```
null_string := TO_VARCHAR2('');
zip_code := SUBSTR(address, 25, 0);
valid := (name != '');
```

NULL 文字列かどうかをテストする場合は、次のように IS NULL 演算子を使ってください。

```
IF my_string IS NULL THEN ...
```

連結演算子

連結演算子は NULL オペランドを無視します。たとえば、次の式は、

```
'apple' || NULL || NULL || 'sauce'
```

これは、次の行を戻します。

```
'applesauce'
```

ファンクション

組み込みファンクションに引数 NULL が渡されると、次に示す場合を除いて NULL が戻されます。

ファンクション DECODE は、先頭の引数を 1 つまたは複数の検索式と比較します。検索式は結果式と対になっています。検索式や結果式は NULL の場合があります。検索に成功すると、対応する結果が戻されます。次の例で、列 rating が NULL ならば、DECODE は値 1000 を戻します。

```
SELECT DECODE(rating, NULL, 1000, 'C', 2000, 'B', 4000, 'A', 5000)
       INTO credit_limit FROM accts WHERE acctno = my_acctno;
```

先頭の引数が NULL の場合、ファンクション NVL は 2 番目の引数の値を戻します。次の例で、hire_date が NULL ならば、NVL は SYSDATE の値を戻します。NULL でなければ、NVL は hire_date の値を戻します。

```
start_date := NVL(hire_date, SYSDATE);
```

2 番目の引数が NULL の場合、ファンクション REPLACE はオプションの 3 番目の引数が存在するかどうかにかかわらず、1 番目の引数の値を戻します。たとえば、次に示す代入の後では、

```
new_string := REPLACE(old_string, NULL, my_string);
```

old_string と new_string の値は同じになります。

3 番目の引数が NULL ならば、REPLACE は、1 番目の引数から 2 番目の引数をすべて削除したものを戻します。たとえば、次の代入式の後では、

```
syllabified_name := 'Gold-i-locks';  
name := REPLACE(syllabified_name, '-', NULL);
```

name の値は 'goldilocks' になります。

2 番目の引数と 3 番目の引数が NULL ならば、REPLACE は単に 1 番目の引数を戻します。

組込みファンクション

PL/SQL には、データを操作するのに役立つ多くの強力なファンクションが用意されています。組込みファンクションは、次のカテゴリに分類できます。

- エラー報告
- 数値
- 文字
- データ型変換
- 日付
- オブジェクト参照
- その他

表 2-4 に、各カテゴリのファンクションを示します。エラー報告ファンクションの説明は、[第 11 章](#)を参照してください。その他のファンクションの説明は、『Oracle8i SQL リファレンス』を参照してください。

SQL 文の中では、エラー報告ファンクション SQLCODE と SQLERRM を除くすべてのファンクションを使えます。また、プロシージャ文の中では、ファンクション DECODE、DUMP、VSIZE を除くすべてのファンクションが使えます。

SQL 集計関数の AVG、COUNT、GROUPING、MIN、MAX、SUM、STDDEV、VARIANCE は、PL/SQL に組み込まれていません。ただし、これらのファンクションは SQL 文の中で使えます（プロシージャ文の中では使えません）。

表 2-4 組込みファンクション

エラー	数値	文字	変換	日付	オブ ジェク ト参照	その他
SQLCODE	ABS	ASCII	CHARTOROWID	ADD_MONTHS	DEREF	BFILENAME
SQLERRM	ACOS	CHR	CONVERT	LAST_DAY	REF	DECODE
	ASIN	CONCAT	HEXTORAW	MONTHS_BETWEEN	VALUE	DUMP
	ATAN	INITCAP	RAWTOHEX	NEW_TIME		EMPTY_BLOB
	ATAN2	INSTR	ROWIDTOCHAR	NEXT_DAY		EMPTY_CLOB
	CEIL	INSTRB	TO_CHAR	ROUND		GREATEST
	COS	LENGTH	TO_DATE	SYSDATE		LEAST
	COSH	LENGTHB	TO_MULTI_BYTE	TRUNC		NLS_CHARSET_DECL_LEN
	EXP	LOWER	TO_NUMBER			NLS_CHARSET_ID
	FLOOR	LPAD	TO_SINGLE_BYTE			NLS_CHARSET_NAME
	LN	LTRIM				NVL
	LOG	NLS_INITCAP				SYS_CONTEXT
	MOD	NLS_LOWER				SYS_GUID
	POWER	NLSSORT				UID
	ROUND	NLS_UPPER				USER
	SIGN	REPLACE				USERENV
	SIN	RPAD				VSIZE
	SINH	RTRIM				
	SQRT	SOUNDEX				
	TAN	SUBSTR				
	TANH	SUBSTRB				
	TRUNC	TRANSLATE				
		TRIM				
		UPPER				

*One ship drives east and another drives west
With the selfsame winds that blow.
'Tis the set of the sails and not the gales
Which tells us the way to go.*—Ella Wheeler Wilcox

この章では、PL/SQL プログラムの制御の流れを構造化する方法を示します。入口と出口を1つずつ持つ、単純かつ強力な制御構造によって文を結合する方法を説明します。これらの構造を使うと、どのような状況でも処理できます。また、これらの構造を適切に使うことで、優れた構造を持つプログラムが自然に作成できます。

主なトピック

概要

条件制御 : IF 文

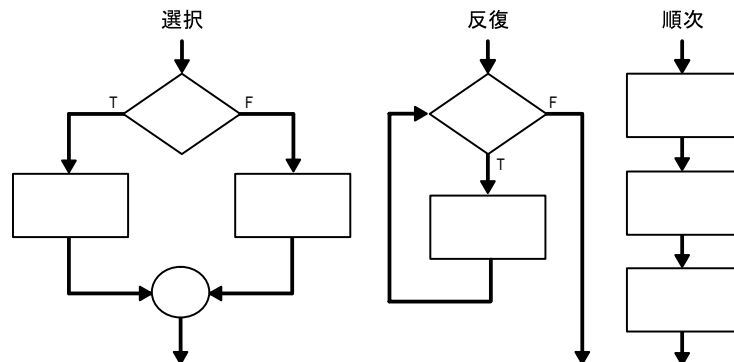
反復制御 : LOOP 文と EXIT 文

順次制御 : GOTO 文と NULL 文

概要

構造定理によれば、どのコンピュータ・プログラムも、[図 3-1](#) に示す基本的な制御構造を使って書くことができます。これらを適当に組み合わせれば、どのような問題でも取り扱うことができます。

図 3-1 制御構造



選択構造は、条件をテストし、条件の真偽に応じて一連の文を実行します。条件とは、ブール値 (TRUE または FALSE) を戻す任意の変数または式です。反復構造は、ある条件が真の間、一連の文を繰り返して実行します。順次構造は、一連の文を、出現する順番にそのまま実行します。

条件制御 : IF 文

状況に応じてアクションを選ばなければならない場面はよくあります。IF 文を使うと、一連の文を条件に合わせて実行できます。つまり、一連の文が実行されるかどうかは、条件の値に依存します。IF 文には、IF-THEN、IF-THEN-ELSE、IF-THEN-ELSIF の、3 つの形式があります。

IF-THEN

IF 文の最も単純な形式である IF-THEN は、キーワード THEN と END IF (ENDIF ではない) によって囲まれた一連の文に条件を関連付けます。次に例を示します。

```
IF condition THEN
    sequence_of_statements
END IF;
```

一連の文は、条件が TRUE に評価された場合にだけ実行されます。条件が FALSE または NULL に評価されると、IF 文は何も実行しません。いずれの場合も、制御は次の文に渡されます。たとえば、

```
IF sales > quota THEN
    compute_bonus(empid);
    UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
END IF;
```

短い IF 文は 1 つの行に書くことができます。

```
IF x > y THEN high := x; END IF;
```

IF-THEN-ELSE

IF 文の 2 つ目の形式では、キーワード ELSE が追加され、その後に THEN 句とは異なる処理をする一連の文を続けます。次に例を示します。

```
IF condition THEN
    sequence_of_statements1
ELSE
    sequence_of_statements2
END IF;
```

ELSE 句の中の一連の文は、条件が FALSE または NULL に評価された場合にだけ実行されます。このように、ELSE 句を使うと必ずなんらかの一連の文が実行されます。次の例では、条件が TRUE の場合に最初の UPDATE 文が実行され、条件が FALSE または NULL の場合に 2 番目の UPDATE 文が実行されます。

```
IF trans_type = 'CR' THEN
    UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
    UPDATE accounts SET balance = balance - debit WHERE ...
END IF;
```

THEN 句と ELSE 句に IF 文を入れることができます。つまり、次の例に示すように、IF 文はネストできます。

```
IF trans_type = 'CR' THEN
    UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
    IF new_balance >= minimum_balance THEN
        UPDATE accounts SET balance = balance - debit WHERE ...
    ELSE
        RAISE insufficient_funds;
    END IF;
END IF;
```

IF-THEN-ELSIF

いくつかの相互排他的なアクションから 1 つのアクションを選択したい場合があります。IF 文の 3 番目の形式では、キーワード `ELSIF` (`ELSEIF` ではない) を使って、条件を追加します。

```
IF condition1 THEN
    sequence_of_statements1
ELSIF condition2 THEN
    sequence_of_statements2
ELSE
    sequence_of_statements3
END IF;
```

条件が `FALSE` または `NULL` に評価されると、`ELSIF` 句は別の条件をテストします。IF 文は任意の数の `ELSIF` 句を持つことができます。最後の `ELSE` 句はオプションです。条件は上から下に向かって 1 つずつ評価されます。いずれかの条件が `TRUE` に評価されると、それに付随する一連の文が実行され、制御は次の文に移ります。すべての条件が `FALSE` または `NULL` に評価されると、`ELSE` 句の一連の文が実行されます。次の例を考えてみます。

```
BEGIN
    ...
    IF sales > 50000 THEN
        bonus := 1500;
    ELSIF sales > 35000 THEN
        bonus := 500;
    ELSE
        bonus := 100;
    END IF;
    INSERT INTO payroll VALUES (emp_id, bonus, ...);
END;
```

`sales` の値が 50000 よりも大きい場合は、1 番目と 2 番目の条件が `TRUE` になります。しかし、2 番目の条件はテストされないため、`bonus` には 1500 という正しい値が代入されます。1 番目の条件が `TRUE` に評価されると、それに付随する文が実行され、制御は `INSERT` 文に移ります。

指針

次の例のような IF 文の使用方法は避けてください。

```
DECLARE
    ...
    overdrawn BOOLEAN;
BEGIN
    ...
    IF new_balance < minimum_balance THEN
        overdrawn := TRUE;
    ELSE
        overdrawn := FALSE;
    END IF;
    ...
    IF overdrawn = TRUE THEN
        RAISE insufficient_funds;
    END IF;
END;
```

このコードでは 2 つの有用な事実が無視されています。第 1 に、論理式の値は論理変数に直接代入できます。つまり、1 番目の IF 文は、次のように単純な代入に置き換えることができます。

```
overdrawn := new_balance < minimum_balance;
```

第 2 に、論理変数はそれ自身が TRUE または FALSE です。つまり、2 番目の IF 文の条件は、次のように単純化できます。

```
IF overdrawn THEN ...
```

可能ならば、IF 文をネストするのではなく、ELSIF 句を使ってください。そうすると、わかりやすく、理解しやすいコードになります。次の IF 文を比較してみてください。

<pre>IF condition1 THEN statement1; ELSE IF condition2 THEN statement2; ELSE IF condition3 THEN statement3; END IF; END IF; END IF;</pre>	<pre>IF condition1 THEN statement1; ELSIF condition2 THEN statement2; ELSIF condition3 THEN statement3; END IF;</pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------

この 2 つの文は論理的に等価ですが、1 つ目の文では論理の流れがあいまいで、2 つ目の文では明解に示されています。

反復制御 : LOOP 文と EXIT 文

LOOP 文を使うと、一連の文を繰り返して実行できます。LOOP 文には、LOOP、WHILE-LOOP、FOR-LOOP の 3 つの形式があります。

LOOP

LOOP 文の最も単純な形式は、キーワード LOOP と END LOOP で一連の文を囲む基本（または無限）ループです。次に例を示します。

```
LOOP
    sequence_of_statements
END LOOP;
```

ループが繰り返されるたびに一連の文が実行され、制御がループの先頭に戻ります。処理の続行が望ましくない場合、または不可能になった場合は、EXIT 文を使ってループを終了できます。ループの中では、任意の場所に 1 つまたは複数の EXIT 文を置くことができます。ただし、ループの外には置くことができません。EXIT 文には、EXIT および EXIT-WHEN という 2 つの形式があります。

EXIT

EXIT 文はループを無条件に終了させます。EXIT 文が現れると、ループはただちに終了し、制御は次の文に移ります。たとえば、

```
LOOP
    ...
    IF credit_rating < 3 THEN
        ...
        EXIT; -- exit loop immediately
    END IF;
END LOOP;
-- control resumes here
```

次の例のように、EXIT 文を使って、PL/SQL のブロックを終了できません。

```
BEGIN
    ...
    IF credit_rating < 3 THEN
        ...
        EXIT; -- illegal
    END IF;
END;
```

EXIT 文はループの中に置くことに注意してください。PL/SQL ブロックを通常終了より前の段階で終了させる場合は、RETURN 文を使います。詳細は、7-8 ページの「[RETURN 文](#)」を参照してください。

EXIT-WHEN

EXIT-WHEN 文を使うと、ループを条件に合わせて終了できます。EXIT 文が見つかった、WHEN 句の中の条件が評価されます。条件の評価結果が TRUE ならば、ループは終了し、制御はそのループの後の文に移ります。たとえば、

```
LOOP
    FETCH c1 INTO ...
    EXIT WHEN c1%NOTFOUND; -- exit loop if condition is true
    ...
END LOOP;
CLOSE c1;
```

条件の評価結果が TRUE になるまで、ループは終了できません。このため、ループの中で条件の値を変更する必要があります。上の例で、FETCH 文が行を戻すと、条件は FALSE に評価されます。FETCH 文が行を戻すことに失敗した場合、条件は TRUE に評価され、ループは終了し、制御は CLOSE 文に移ります。

EXIT-WHEN 文は単純な IF 文のかわりとして使えます。たとえば、次の 2 つの文を比較してみてください。

IF count > 100 THEN		EXIT WHEN count > 100;
EXIT;		
END IF;		

この 2 つの文は論理的に等価ですが、EXIT-WHEN 文の方がわかりやすく、理解しやすくなっています。

ループ・ラベル

PL/SQL ブロックと同様に、ループにもラベルを付けることができます。ラベルは二重の山カッコで囲んだ未宣言の識別子で、次に示すように LOOP 文の先頭に置きます。

```
<<label_name>>
LOOP
    sequence_of_statements
END LOOP;
```

次の例のように、オプションとして、LOOP 文の末尾にもラベル名を付けることができます。

```
<<my_loop>>
LOOP
    ...
END LOOP my_loop;
```

ラベル付きのループをネストする場合は、末尾のラベルを使ってわかりやすくできます。

どちらの形式の EXIT 文でも、現在のループに限らず、任意の外側のループも終了させることができます。これを行うには、終了したい外側のループにラベルを付けます。次に示すように、EXIT 文でそのラベルを使用します。

```
<<outer>>
LOOP
    ...
    LOOP
        ...
        EXIT outer WHEN ... -- exit both loops
    END LOOP;
    ...
END LOOP outer;
```

ラベルを付けた外側のループが、内側のループを含めて終了します。

WHILE-LOOP

WHILE-LOOP 文は、キーワード LOOP と END LOOP で囲まれた一連の文に条件を結び付けます。次に例を示します。

```
WHILE condition LOOP
    sequence_of_statements
END LOOP;
```

ループを反復する前に条件が評価されます。条件が TRUE ならば、一連の文が実行されてから、ループの先頭で制御が再開します。条件が FALSE または NULL ならば、ループは実行されず、制御は次の文に移ります。たとえば、

```
WHILE total <= 25000 LOOP
    ...
    SELECT sal INTO salary FROM emp WHERE ...
    total := total + salary;
END LOOP;
```

反復の回数は条件に依存し、ループが終了するまでわかりません。条件はループの先頭でテストされるため、一連の文が一度も実行されない可能性もあります。上の例で total の初期値が 25000 よりも大きいと、条件が FALSE に評価されてループは実行されません。

いくつかの言語は、条件をループの先頭ではなく末尾でテストする LOOP UNTIL 構造または REPEAT UNTIL 構造を持っています。この場合、一連の文は少なくとも一度は実行されます。PL/SQL にはこうした構造はありませんが、次のようにすれば簡単に作ることができます。

```
LOOP
    sequence_of_statements
    EXIT WHEN boolean_expression;
END LOOP;
```

WHILE ループが少なくとも一度は実行されるようにするには、初期化済みのブール変数を条件の中で使います。

```
done := FALSE;
WHILE NOT done LOOP
    sequence_of_statements
    done := boolean_expression;
END LOOP;
```

ループの中の文で、論理変数に新しい値を代入しなければなりません。さもないと無限ループになってしまいます。たとえば、次の 2 つの LOOP 文は論理的に等価です。

WHILE TRUE LOOP		LOOP
...		...
END LOOP;		END LOOP;

FOR-LOOP

WHILE ループの反復回数はループが終了するまではわかりませんが、FOR ループの反復回数はループに入る前からわかっています。FOR ループは、指定された整数の範囲内でループを繰り返し実行します。(カーソルの結果セットの中で繰り返すカーソル FOR ループについては、[第 5 章](#)で説明します。) 繰り返しの範囲は、キーワード FOR と LOOP に囲まれた反復スキーマの一部です。二重ドット (..) は、範囲演算子です。次に構文を示します。

```
FOR counter IN [REVERSE] lower_bound..higher_bound LOOP
    sequence_of_statements
END LOOP;
```

繰り返しの範囲は FOR ループに入った段階で評価され、それ以降は評価されません。

次の例に示すように、一連の文は範囲中の整数 1 つについて 1 回実行されます。繰り返しが 1 回起こるたびに、ループ・カウンタが 1 つ増やされます。

```
FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i
    sequence_of_statements -- executes three times
END LOOP;
```

次の例のように、下限が上限と等しければ、一連の文は 1 回だけ実行されます。

```
FOR i IN 3..3 LOOP -- assign the value 3 to i
    sequence_of_statements -- executes one time
END LOOP;
```

デフォルトでは、反復は下限から上限の向きに進みます。しかし、次の例のように、キーワード REVERSE を使うと、反復は上限から下限の向きに進みます。繰り返しが 1 回起こるたびに、ループ・カウンタが 1 つ減らされます。

```
FOR i IN REVERSE 1..3 LOOP -- assign the values 3,2,1 to i
```

```
sequence_of_statements -- executes three times
END LOOP;
```

この場合でも、範囲の上限と下限は（降順ではなく）昇順に書きます。

FOR ループの中では、ループ・カウンタは定数のように参照できますが、値は代入できません。次に例を示します。

```
FOR ctr IN 1..10 LOOP
  IF NOT finished THEN
    INSERT INTO ... VALUES (ctr, ...); -- legal
    factor := ctr * 2; -- legal
  ELSE
    ctr := 10; -- illegal
  END IF;
END LOOP;
```

反復スキーム

ループ範囲の上限と下限にはリテラル、変数または式が使えますが、必ず整数に評価されるものでなければなりません。次に例を示します。このように、下限は 1 である必要はありません。ただし、ループ・カウンタの増分値（または減分値）は 1 でなければなりません。

```
j IN -5..5
k IN REVERSE first..last
step IN 0..TRUNC(high/low) * 2
```

言語によっては、STEP 句を使って異なる増分値を指定できるものがあります。次に BASIC で書かれた例を示します。

```
FOR J = 5 TO 15 STEP 5 :REM assign values 5,10,15 to J
  sequence_of_statements -- J has values 5,10,15
NEXT J
```

PL/SQL はこのような構造を持っていませんが、作るのは簡単です。次の例を考えてみます。

```
FOR j IN 5..15 LOOP -- assign values 5,6,7,... to j
  IF MOD(j, 5) = 0 THEN -- pass multiples of 5
    sequence_of_statements -- j has values 5,10,15
  END IF;
END LOOP;
```

このループは上の BASIC ループと論理的に等価です。一連の文の中では、値 5、10 および 15 だけが使われます。

あまりエレガントな手法とはいえませんが、効率的な次の手法を使えます。一連の文の中では、ループ・カウンタが参照されるたびに増分値が乗算されたものが使えます。

```
FOR j IN 1..3 LOOP -- assign values 1,2,3 to j
```



```
sequence_of_statements -- each j becomes j*5
END LOOP;
```

動的な範囲

次に示すように、PL/SQL ではループの繰返し範囲を実行時に動的に決定できます。

```
SELECT COUNT(empno) INTO emp_count FROM emp;
FOR i IN 1..emp_count LOOP
    ...
END LOOP;
```

emp_count の値はコンパイル時には未定で、SELECT 文が実行時に値を戻します。

ループの繰返し範囲の下限が上限よりも大きな整数に評価されると、どうなるでしょうか。次の例に示すように、ループ中の一連の文は実行されず、制御は次の文に移ります。

```
-- limit becomes 1
FOR i IN 2..limit LOOP
    sequence_of_statements -- executes zero times
END LOOP;
-- control passes here
```

有効範囲の規則

ループ・カウンタはループの中でしか定義されません。そのため、ループの外側からは参照できません。次に示すように、ループが終了すると、ループ・カウンタは未定義になります。

```
FOR ctr IN 1..10 LOOP
    ...
END LOOP;
sum := ctr - 1; -- illegal
```

ループ・カウンタは、INTEGER 型のローカル変数として暗黙的に宣言されているので、明示的に宣言する必要はありません。次の例では、ローカル宣言がグローバル宣言を隠しています。

```
DECLARE
    ctr INTEGER;
BEGIN
    ...
    FOR ctr IN 1..25 LOOP
        ...
        IF ctr > 10 THEN ... -- refers to loop counter
    END LOOP;
END;
```

この例でグローバル変数を使う場合は、次のようにラベルとドット表記法を使う必要があります。

```
<<main>>
DECLARE
    ctr  INTEGER;
    ...
BEGIN
    ...
    FOR ctr IN 1..25 LOOP
        ...
        IF main.ctr > 10 THEN -- refers to global variable
            ...
        END IF;
    END LOOP;
END main;
```

ネストされた FOR ループにも同じ有効範囲の規則が適用されます。次の例を考えてみてください。どちらのループ・カウンタも同じ名前を持っています。このため、内側のループから外側のループ・カウンタを参照する場合は、次のようにラベルとドット表記法を使います。

```
<<outer>>
FOR step IN 1..25 LOOP
    FOR step IN 1..10 LOOP
        ...
        IF outer.step > 15 THEN ...
    END LOOP;
END LOOP outer;
```

EXIT 文の使用方法

EXIT 文を使うと、FOR ループを途中で終了させることができます。たとえば、次のループは通常は 10 回実行されますが、FETCH 文が行を戻さなくなると、ループはそれまで何回実行されていてもただちに終了します。

```
FOR j IN 1..10 LOOP
    FETCH c1 INTO emp_rec;
    EXIT WHEN c1%NOTFOUND;
    ...
END LOOP;
```

ネストされた FOR ループから途中で出なければならなくなった場合でも、現在のループだけでなく、外側のループも終了できます。これを行うには、終了したい外側のループにラベルを付けます。次に示すように、EXIT 文でそのラベルを使用して、どの FOR ループを終了するかを指定します。

```
<<outer>>
FOR i IN 1..5 LOOP
```

```

...
FOR j IN 1..10 LOOP
    FETCH c1 INTO emp_rec;
    EXIT outer WHEN c1%NOTFOUND;  -- exit both FOR loops
...
END LOOP;
END LOOP outer;
-- control passes here

```

順次制御 : GOTO 文と NULL 文

IF 文や LOOP 文とは異なり、GOTO 文と NULL 文は、PL/SQL プログラミングにとって重要なものではありません。PL/SQL の構造では、通常は GOTO 文は不要です。ただし、GOTO 文を使うと論理を単純化できる場合もあります。NULL 文には、条件文の意味とアクションを明確にすることによって、コードをわかりやすくする効果があります。

GOTO 文を多用すると、構造化されていない複雑なコード（スパゲティ・コードと呼ばれることもある）になり、理解やメンテナンスが難しくなりがちです。GOTO 文はなるべく使わないようにしてください。たとえば、深くネストされた構造からエラー処理ルーチンに分岐する場合は、GOTO 文を使うのではなく、例外を呼び出してください。

GOTO 文

GOTO 文はラベルに無条件に分岐する場合に使います。ラベルは有効範囲の中で他と重複しないもので、実行可能文か PL/SQL ブロックの前に置かれている必要があります。GOTO 文が実行されると、ラベルが付けられた文またはブロックに制御が移ります。次の例では、一連の文の下の方にある実行可能文に制御が渡されています。

```

BEGIN
    ...
    GOTO insert_row;
    ...
    <<insert_row>>
    INSERT INTO emp VALUES ...
END;

```

次の例では、一連の文の上の方にある PL/SQL ブロックに制御が渡されています。

```

BEGIN
    ...
    <<update_row>>
    BEGIN
        UPDATE emp SET ...
    ...
END;
...

```

```
GOTO update_row;
...
END;
```

次の例に示すラベル `end_loop` は、実行可能文の前に置かれていないので不正です。

```
DECLARE
  done BOOLEAN;
BEGIN
  ...
  FOR i IN 1..50 LOOP
    IF done THEN
      GOTO end_loop;
    END IF;
    ...
    <<end_loop>> -- illegal
  END LOOP; -- not an executable statement
END;
```

上の例をデバッグするには、次のように NULL 文を追加してください。

```
FOR i IN 1..50 LOOP
  IF done THEN
    GOTO end_loop;
  END IF;
  ...
  <<end_loop>>
  NULL; -- an executable statement
END LOOP;
```

次の例に示すように、GOTO 文で現在のブロックから外側のブロックに分岐できます。

```
DECLARE
  my_ename CHAR(10);
BEGIN
  <<get_name>>
  SELECT ename INTO my_ename FROM emp WHERE ...
  BEGIN
    ...
    GOTO get_name; -- branch to enclosing block
  END;
END;
```

この GOTO 文では、参照されたラベルが置かれている 1 つ外側のブロックに分岐します。

制限

GOTO 文の宛先として使えないものがあります。特に、GOTO 文は IF 文または LOOP 文、サブブロックには分岐できません。たとえば、次の GOTO 文は誤りです。

```
BEGIN
  ...
  GOTO update_row; -- illegal branch into IF statement
  ...
  IF valid THEN
    ...
    <<update_row>>
    UPDATE emp SET ...
  END IF;
END;
```

また、次の例に示すように、GOTO 文では IF 文の句から句へ分岐できません。

```
BEGIN
  ...
  IF valid THEN
    ...
    GOTO update_row; -- illegal branch into ELSE clause
  ELSE
    ...
    <<update_row>>
    UPDATE emp SET ...
  END IF;
END;
```

次の例に示すように、GOTO 文では外側のブロックからサブブロックに分岐できません。

```
BEGIN
  ...
  IF status = 'OBSOLETE' THEN
    GOTO delete_part; -- illegal branch into sub-block
  END IF;
  ...
  BEGIN
    ...
    <<delete_part>>
    DELETE FROM parts WHERE ...
  END;
END;
```

また、次の例に示すように、GOTO 文ではサブプログラムの外に分岐できません。

```
DECLARE
  ...
```

```
PROCEDURE compute_bonus (emp_id NUMBER) IS
BEGIN
    ...
    GOTO update_row; -- illegal branch out of subprogram
END;
BEGIN
    ...
    <<update_row>>
    UPDATE emp SET ...
END;
```

最後に、GOTO 文では例外ハンドラから現在のブロックに分岐できません。たとえば、次の GOTO 文は誤りです。

```
DECLARE
    ...
    pe_ratio REAL;
BEGIN
    ...
    SELECT price / NVL(earnings, 0) INTO pe_ratio FROM ...
    <<insert_row>>
    INSERT INTO stats VALUES (pe_ratio, ...);
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        pe_ratio := 0;
        GOTO insert_row; -- illegal branch into current block
END;
```

ただし、例外ハンドラから外側のブロックに分岐できます。

NULL 文

NULL 文は、アクションを起こさないことを明示するために使います。NULL 文は制御を次の文に渡すことしかしません。しかし、これによってコードをわかりやすくできます。代替アクションが指定できる構成体では、NULL 文はプレースホルダとしての役割を果たします。読み手に対して、代替アクションを誤って見逃したのではなく、実際にアクションが不要であるということを伝えることができます。次の例では、NULL 文によって、名前のない例外ではアクションを起こさないことを明確にしています。

```
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        ROLLBACK;
    WHEN VALUE_ERROR THEN
        INSERT INTO errors VALUES ...
        COMMIT;
    WHEN OTHERS THEN
        NULL;
```

```
END;
```

IF 文のすべての句には、少なくとも 1 つの実行可能文がなければなりません。NULL 文は実行可能文なので、構文上では句が必要だがアクションは起こしたくないという場合に使えます。次の例では、NULL 文によって、成績優秀な従業員だけがボーナスを受け取ることがわかりやすくなっています。

```
IF rating > 90 THEN
    compute_bonus(emp_id);
ELSE
    NULL;
END IF;
```

また、NULL 文はアプリケーションをトップダウンで設計する際に、スタブを簡単に作るためにも使えます。スタブはダミーのサブプログラムです。スタブを使うと、メイン・プログラムのテストとデバッグが終了するまで、プロシージャまたは関数の定義をしないで済むことが可能になります。次の例では、NULL 文によって、サブプログラムの実行部に少なくとも 1 つの文が存在しなければならないという条件を解決しています。

```
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
BEGIN
    NULL;
END debit_account;
```

コレクションとレコード

Knowledge is that area of ignorance that we arrange and classify.—Ambrose Bierce

従来のデータベース・アプリケーションにおいて、プログラマが配列、バッグ、リスト、ネストした表、セット、ツリーなどのコレクション型を使うことがますます多くなっています。増加する要求を満たすために、PL/SQL では `TABLE` と `VARRAY` というデータ型が用意されており、索引付き表、ネストした表、可変サイズの配列を宣言できるようになっています。この章では、これらの型を使ってデータの集まり（コレクション）をオブジェクト全体として参照したり操作したりする方法を説明します。また、データ型 `RECORD` を使って、関連してはいるが異なるデータを 1 つの論理単位として扱う方法も説明します。

主なトピック

- コレクションとは？
- コレクションの初期化と参照
- コレクションの代入と比較
- コレクションの操作
- コレクション・メソッドの使用
- コレクション例外の回避
- バルク・バインドの利用
- レコードとは？
- レコードの定義と宣言
- レコードの初期化と参照
- レコードの代入と比較
- レコードの操作

コレクションとは？

コレクションは、すべて同じ型の要素の順序付きグループです（あるクラスの生徒の成績など）。各要素には一意の添字が付いています。その番号によって、集合の中での要素の位置が決まります。PL/SQL では 2 つのコレクション型を用意しています。TABLE 型の項目は、索引付き表（バージョン 2 PL/SQL 表）またはネストした表（索引付き表の機能を拡張）のいずれかです。VARRAY 型の項目は `varrays`（可変サイズの配列）です。

コレクションは、ほとんどの第 3 世代のプログラミング言語で見られる配列のような働きをします。ただし、コレクションには 1 次元しかないため、添字を整数にしなければなりません。（Ada および Pascal などの言語では、複数次元の配列が可能であり、添字を列挙型にすることができます）。

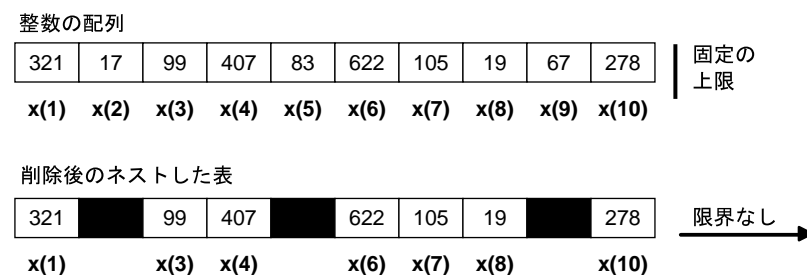
コレクションにオブジェクト型のインスタンスを格納したり、また、逆にコレクションがオブジェクト型の属性であったりします。また、コレクションはパラメータとして渡すこともできます。そのため、それらを使うことにより、データの列をデータベースの表に入れたり、データの列をデータベースの表から出したり、クライアント側アプリケーションとストアード・サブプログラムとの間でデータの列を移動したりできます。さらに、PL/SQL パッケージでコレクション型を定義して、アプリケーションでプログラムとして使用できます。

ネストした表について

データベース内では、ネストした表は 1 列のデータベース表と考えられます。Oracle では、ネストした表の行を特に順序付けずに格納します。ただし、ネストした表を PL/SQL 変数の中に取り出すと、それらの行に 1 から始まる連続した添字が付けられます。これによって、個々の行に配列のようにアクセスできるようになります。

PL/SQL 内では、ネストした表は 1 次元配列と類似しています。しかし、ネストした表と配列には重要な相違点が 2 つあります。第 1 に、配列には固定の上限がありますが、ネストした表には限界がありません（図 4-1 を参照）。したがって、ネストした表のサイズは動的に大きくできます。

図 4-1 配列とネストした表との相違点



第 2 に、配列は密でなければなりません（つまり添字は連続していなければなりません）。そのため、個々の要素を配列から削除することはできません。ネストした表も、最初は密で

すが、疎にすることができます（つまり添字が連続していなくてもかまいません）。したがって、組込みプロシージャ `DELETE` を使って、ネストした表から要素を削除できます。その場合は索引に欠番が生じることになりますが、組込みファンクション `NEXT` を使えば連続した添字に対する反復処理を実行できます。

ネストした表と索引付き表との相違点

ネストした表と索引付き表は似ています。たとえば、これらは同じ構造を持ち、個々の要素は同じ方法で（添字表記法を使用して）アクセスされます。主な相違点は、ネストした表はデータベース列に格納できる（そのために「ネストした表」と呼ばれる）のに対し、索引付き表はできないという点です。

ネストした表は、データベースに格納されたネストした表を `SELECT`、`INSERT`、`UPDATE` および `DELETE` することによって、索引付き表の機能を拡張します。（索引付き表はデータベースに格納できないことに注意してください。）また、コレクション・メソッドのなかには、ネストした表と `varrays` でしか使用できないものがあります。たとえば、組込みプロシージャ `TRIM` は、索引付き表には適用できません。

ネストした表のもうひとつの利点は、初期化するまではアトミック `NULL`（つまり、表の要素ではなく表自体が `NULL`）である点です。索引付き表は、初期化するまでは単に空です。そのため、ネストした表には `IS NULL` 比較演算子を適用できますが、索引付き表には適用できません。

しかし、索引付き表にも利点があります。たとえば、`PL/SQL` は、ホスト配列と索引付き表との間で暗黙の（自動）データ型変換をサポートしています（ネストした表ではサポートされていません）。このため、データベース・サーバーとの間でのコレクションの受け渡しには、無名 `PL/SQL` ブロックを使用してホスト配列を索引付き表にバルク・バインド入出力するのが、最も効果的な方法です。

また、索引付き表は最初は疎です。これは数値の主キー（口座番号や従業員番号など）を索引として使用して参照データを格納するのに便利です。

まれに、索引付き表の方が、ネストした表より柔軟性が高い場合があります。たとえば、ネストした表の添字は無制約です。索引付きの表では、添字が負場合があります（ネストした表にはありません）。また、索引付き表には使用できるがネストした表には使用できない要素型もいくつかあります（4-10 ページの「[コレクション要素の参照](#)」を参照）。最後に、ネストした表を拡張するには、組込みプロシージャ `EXTEND` を使用する必要がありますが、索引付き表を拡張するには、現在のものより大きな添字を指定するだけです。

varray について

VARRAY 型の項目は、*varray* と呼ばれます。*varray* を使うと、単一の識別子をコレクション全体に関連付けることができます。この関連付けによって、コレクションを全体として操作し、個々の要素の参照が簡単になります。要素を参照するには、標準的な添字構文を使います（[図 4-2](#) を参照）。たとえば、`Grade(3)` は、`Grades` という *varray* の 3 番目の要素を参照します。

図 4-2 サイズ 10 の *varray*

Varray *Grades*

B	C	A	A	C	D	B			
(1)	(2)	(3)	(4)	(5)	(6)	(7)			

最大サイズ
= 10

varray には最大サイズがあり、このサイズを型定義で指定する必要があります。*varray* の索引には、1 に固定されている下限と、拡張可能な上限があります。たとえば、`varrayGrades` の現在の上限は 7 ですが、これを 8、9 または 10 に拡張できます。したがって、*varray* に入れることのできる要素の数は、0（空の場合）個から、型定義で指定された最大値までいろいろに変えられます。

varray とネストした表との相違点

ネストした表は、以下の点で *varray* とは異なります。

- *varray* には最大サイズがあるが、ネストした表にはない。
- *varray* は常に密だが、ネストした表は疎にもできる。したがって、ネストした表からは個々の要素を削除できるが、*varray* からは削除できない。
- Oracle では、*varray* のデータは行内部（同じ表領域）に格納される。しかし、ネストした表のデータは記憶域表の行外部に格納される。記憶域表は、ネストした表に関連付けられたシステム生成データベース表である。
- データベースに格納されるとき、*varray* はその順序と添字が保たれるが、ネストした表は保たれない。

どのコレクション型を使うべきでしょうか。それは、使用目的とコレクションのサイズによります。*varray* は不透明オブジェクトとして格納されますが、ネストした表はすべての要素を記憶域表の行に対応付けた形で、記憶域表に格納されます。したがって、効率的な問合せが必要な場合は、ネストした表を使います。コレクション全体をまとめて取り出したい場合は、*varray* を使います。ただし、コレクションが非常に大きい場合、取り出すのはサブセットまでにしておいたほうが実用的です。つまり、*varray* は比較的小さいコレクションに向いています。

コレクションの定義と宣言

コレクションを作るには、コレクション型を定義してから、その型のコレクションを宣言します。TABLE 型および VARRAY 型は、任意の PL/SQL ブロック、サブプログラム、またはパッケージの宣言部で定義できます。ネストした表の場合には、次の構文を使います。

```
TYPE type_name IS TABLE OF element_type [NOT NULL];
```

varray の場合には、次の構文を使用します。

```
TYPE type_name IS {VARRAY | VARYING ARRAY} (size_limit)
  OF element_type [NOT NULL];
```

ここで、type_name はコレクションを宣言するために後で使われる型指定子、size_limit は正の整数リテラル、element_type は次にあげる型を除く PL/SQL のデータ型です。

```
BINARY_INTEGER, PLS_INTEGER
BOOLEAN
BLOB、CLOB ( varrays に対してのみ制約が適用される )
LONG、LONG RAW
NATURAL、NATURALN
NCHAR、NCLOB、NVARCHAR2
BLOB または CLOB 属性のオブジェクト型 ( varrays に対してのみ制約が適用される )
TABLE 属性または VARRAY 属性のオブジェクト型
POSITIVE、POSITIVEN
REF CURSOR
SIGNTYPE
STRING
TABLE
VARRAY
```

element_type がレコード型である場合、そのレコード中のすべてのフィールドはスカラー型かオブジェクト型でなければなりません。

索引付き表の場合には、次の構文を使います。

```
TYPE type_name IS TABLE OF element_type [NOT NULL]
  INDEX BY BINARY_INTEGER;
```

ネストした表や varrays とは異なり、索引付き表には、BINARY_INTEGER、BOOLEAN、LONG、LONG RAW、NATURAL、NATURALN、PLS_INTEGER、POSITIVE、POSITIVEN、SIGNTYPE、STRING の要素型を使用できます。これは、ネストした表および varray が、主にデータベース列として作られているためです。そのため、PL/SQL 固有の型を使用できません。理論的には、ローカルで宣言すれば、ネストした表および varray でもそうした型を使えますが、一貫性を保つために制限が加えられています。

索引付き表は最初は疎です。そのため、たとえば、数値の主キーを索引として使って、参照データを一時索引付き表に格納できます。次の例では、レコードの索引付き表を宣言しています。表の要素のそれぞれに、emp データベース表の行が格納されます。

```
DECLARE
    TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE
        INDEX BY BINARY_INTEGER;
    emp_tab EmpTabTyp;
BEGIN
    /* Retrieve employee record. */
    SELECT * INTO emp_tab(7468) FROM emp WHERE empno = 7788;
```

VARRAY 型の定義では、その最大サイズを指定する必要があります。次の例では、366 個までの日付を格納する型を定義します。

```
DECLARE
    TYPE Calendar IS VARRAY(366) OF DATE;
```

要素型を指定するには、%TYPE を使って変数またはデータベース列のデータ型を指定できます。また、%ROWTYPE を使って、カーソルまたはデータベース表の行の型を指定できます。2 つの例を次に示します。

```
DECLARE
    TYPE EmpList IS TABLE OF emp.ename%TYPE; -- based on column
    CURSOR c1 IS SELECT * FROM dept;
    TYPE DeptFile IS VARRAY(20) OF c1%ROWTYPE; -- based on cursor
```

次の例では、RECORD 型を使って、要素型を指定しています。

```
DECLARE
    TYPE AnEntry IS RECORD (
        term    VARCHAR2(20),
        meaning VARCHAR2(200));
    TYPE Glossary IS VARRAY(250) OF AnEntry;
```

次の最後の例では、NOT NULL 制約を要素の型に指定します。

```
DECLARE
    TYPE EmpList IS TABLE OF emp.empno%TYPE NOT NULL;
```

初期化の句は必要ありません (指定できません)。

コレクションの宣言

一度コレクション型を定義したなら、次の SQL*Plus スクリプトに示すようにしてその型のコレクションを宣言できます。

```
CREATE TYPE CourseList AS TABLE OF VARCHAR2(10)  -- define type
/
CREATE TYPE Student AS OBJECT (  -- create object
    id_num  INTEGER(4),
    name    VARCHAR2(25),
    address VARCHAR2(35),
    status  CHAR(2),
    courses CourseList)  -- declare nested table as attribute
/
```

識別子 `courses` はネストした表全体を表します。`courses` の各要素には、'Math 1020' などの大学のコースのコード名を入れます。

以下のスクリプトによって、`varray` を格納するデータベースの列が作成されます。`varray` の各要素には、`Project` オブジェクトが格納されます。

```
CREATE TYPE Project AS OBJECT(  --create object
    project_no NUMBER(2),
    title      VARCHAR2(35),
    cost       NUMBER(7,2))
/
CREATE TYPE ProjectList AS VARRAY(50) OF Project  -- define VARRAY type
/
CREATE TABLE department (  -- create database table
    dept_id  NUMBER(2),
    name     VARCHAR2(15),
    budget   NUMBER(11,2),
    projects ProjectList)  -- declare varray as column
/
```

次の例は、`%TYPE` を使って、あらかじめ宣言されたコレクションのデータ型を指定できることを示しています。

```
DECLARE
    TYPE Platoon IS VARRAY(20) OF Soldier;
    p1 Platoon;
    p2 p1%TYPE;
```

コレクションは、ファンクションおよびプロシージャの仮パラメータとして宣言できます。そのようにすると、コレクションをストアド・サブプログラムに渡したり、あるサブプログラムから別のサブプログラムに渡したりできます。次の例では、ネストした表をパッケージ・プロシージャの仮パラメータとして宣言しています。

```
CREATE PACKAGE personnel AS
```

```
TYPE Staff IS TABLE OF Employee;
...
PROCEDURE award_bonuses (members IN Staff);
END personnel;
```

また、次の例に示すように、ファンクション仕様部の RETURN 句の中にコレクション型を指定できます。

```
DECLARE
TYPE SalesForce IS VARRAY(25) OF Salesperson;
FUNCTION top_performers (n INTEGER) RETURN SalesForce IS ...
```

コレクションは、通常の有効範囲とインスタンス生成の規則に従います。ブロックまたはサブプログラムの中で、コレクションは、ブロックまたはサブプログラムに入ったときにインスタンス化され、ブロックまたはサブプログラムが終了した時点で消滅します。パッケージの中では、そのパッケージが初めて参照された時点でコレクションはインスタンス化され、データベース・セッションが終わった時点で消滅します。

コレクションの初期化と参照

ネストした表または varray は、初期化するまではアトミック NULL（つまりコレクションの要素ではなく、コレクション自体が NULL）です。ネストした表または varray を初期化するには、コンストラクタ（コレクション型と同じ名前のシステム定義ファンクション）を使います。このファンクションは、コレクションに渡される要素から、コレクションを「構成（コンストラクト）」します。次の例では、6 つの要素をコンストラクタ CourseList() に渡し、それらの要素を含むネストした表が戻されます。

```
DECLARE
my_courses CourseList;
BEGIN
my_courses := CourseList('Econ 2010', 'Acct 3401', 'Mgmt 3100',
    'PoSc 3141', 'Mktg 3312', 'Engl 2005');
...
END;
```

次の例では、3 つのオブジェクトをコンストラクタ ProjectList() に渡し、それらのオブジェクトを含む varray が戻されます。

```
DECLARE
accounting_projects ProjectList;
BEGIN
accounting_projects :=
    ProjectList(Project(1, 'Design New Expense Report', 3250),
        Project(2, 'Outsource Payroll', 12350),
        Project(3, 'Audit Accounts Payable', 1425));
...
END;
```


varray 全体を初期化する必要はありません。たとえば、varray の最大サイズが 50 の場合、コンストラクタに渡せる要素は 50 未満です。

要素に NOT NULL 制約を指定していない場合、またはレコード型を指定していない場合は、コンストラクタに NULL 要素を渡せます。たとえば、

```
BEGIN
    my_courses := CourseList('Math 3010', NULL, 'Stat 3202', ...);
```

次の例に示すように、宣言の中でコレクションを初期化できます。これはプログラミング上 好ましい習慣です。

```
DECLARE
    my_courses CourseList :=
        CourseList('Art 1111', 'Hist 3100', 'Engl 2005', ...);
```

引数を指定しないでコンストラクタをコールすると、次の例で示すように、空だが NULL でないコレクションを受け取ります。

```
DECLARE
    TYPE Clientele IS VARRAY(100) OF Customer;
    vips Clientele := Clientele(); -- initialize empty varray
BEGIN
    IF vips IS NOT NULL THEN -- condition yields TRUE
        ...
    END IF;
END;
```

索引付き表を除き、PL/SQL が暗黙のうちにコンストラクタをコールすることは決してないため、明示的にコールする必要があります。コンストラクタは、ファンクション・コールが許可されているところでコールできます。これには SELECT 句、VALUES 句、SET 句が含まれます。

次の例では、Student オブジェクトをオブジェクトの表 sophomores に挿入します。表コンストラクタ CourseList() により、属性 courses の値が指定されます。

```
BEGIN
    INSERT INTO sophomores
        VALUES (Student(5035, 'Janet Alvarez', '122 Broad St', 'FT',
            CourseList('Econ 2010', 'Acct 3401', 'Mgmt 3100', ...)));
    ...
```

次の最後の例では、データベースの表 department に行を挿入します。varray コンストラクタ ProjectList() により、列 projects の値が指定されます。

```
BEGIN
    INSERT INTO department
        VALUES(60, 'Security', 750400,
```

```
ProjectList (Project (1, 'Issue New Employee Badges', 9500),
               Project (2, 'Find Missing IC Chips', 2750),
               Project (3, 'Inspect Emergency Exits', 1900));
...
```

コレクション要素の参照

要素への参照はいずれも、コレクション名と添字をカッコで囲んで指定します。この添字によって、処理の対象となる要素が決まります。要素を参照するには、次の構文を使ってその添字を指定します。

```
collection_name(subscript)
```

ここで、`subscript` は結果が整数になる式です。索引付き表の場合、有効な添字の範囲は -2147483647 ~ 2147483647 です。ネストした表の場合、有効な添字の範囲は 1 ~ 2147483647 です。また、`varrays` の場合、有効な添字の範囲は 1 ~ `size_limit` です。

どの式のコンテキスト中でもコレクションを参照できます。次の例では、ネストした表の `names` の要素を参照しています。

```
DECLARE
    TYPE Roster IS TABLE OF VARCHAR2(15);
    names Roster := Roster('J Hamil', 'D Caruso', 'R Singh', ...);
    i BINARY_INTEGER;
BEGIN
    ...
    IF names(i) = 'J Hamil' THEN
        ...
    END IF;
END;
```

次の例に示すように、サブプログラム・コール中にコレクションの要素を参照できます。

```
DECLARE
    TYPE Roster IS TABLE OF VARCHAR2(15);
    names Roster := Roster('J Hamil', 'D Piro', 'R Singh', ...);
    i BINARY_INTEGER;
BEGIN
    ...
    verify_name(names(i)); -- call procedure
END;
```

コレクションを戻すファンクションをコールする場合、次の構文を使ってコレクション内の要素を参照します。

```
function_name(parameter_list)(subscript)
```

たとえば、次のコールはファンクション `new_hires` が戻す `varray` の 3 番目の要素を参照します。

```
DECLARE
  TYPE Staff IS VARRAY(20) OF Employee;
  staffer Employee;
  FUNCTION new_hires (hiredate DATE) RETURN Staff IS ...
BEGIN
  staffer := new_hires('16-OCT-96')(3);  -- call function
  ...
END;
```

コレクションの代入と比較

あるコレクションを、`INSERT` 文、`UPDATE` 文、`FETCH` 文、`SELECT` 文、代入文、またはサブプログラム・コールによって、別のコレクションに代入できます。以下の例に示すように、それらのコレクションは同じデータ型でなければなりません。要素型が同じだけでは不十分です。

```
DECLARE
  TYPE Clientele IS VARRAY(100) OF Customer;
  TYPE Vips IS VARRAY(100) OF Customer;
  group1 Clientele := Clientele(...);
  group2 Clientele := Clientele(...);
  group3 Vips := Vips(...);
BEGIN
  group2 := group1;
  group3 := group2;  -- illegal; different datatypes
```

アトミック `NULL` である `collection` を別の `collection` に代入すると、代入先の `collection` はアトミック `NULL` になります（そして再初期化が必要です）。次の例を考えてみます。

```
DECLARE
  TYPE Clientele IS TABLE OF Customer;
  group1 Clientele := Clientele(...);  -- initialized
  group2 Clientele;  -- atomically null
BEGIN
  IF group1 IS NULL THEN ...  -- condition yields FALSE
  group1 := group2;
  IF group1 IS NULL THEN ...  -- condition yields TRUE
  ...
END;
```

同じように、値のない `NULL` をコレクションに代入すると、そのコレクションはアトミック `NULL` になります。

コレクション要素の代入

次の構文を使うと、式の値をコレクションの特定の要素に代入できます。

```
collection_name(subscript) := expression;
```

ここで、`expression` は結果がコレクション型定義の要素に指定された型の値になる式です。`subscript` が `NULL` の場合、または整数に変換できない場合、PL/SQL は事前に定義された `VALUE_ERROR` 例外を呼び出します。コレクションがアトミック `NULL` である場合、PL/SQL は `COLLECTION_IS_NULL` を呼び出します。次に例を示します。

```
DECLARE
    TYPE NumList IS TABLE OF INTEGER;
    nums NumList := NumList(10,20,30);
    ints NumList;
    ...
BEGIN
    ...
    nums(1) := TRUNC(high/low);
    nums(3) := nums(1);
    nums(2) := ASCII('B');
    /* Assume execution continues despite the raised exception. */
    nums('A') := 40; -- raises VALUE_ERROR
    ints(1) := 15;   -- raises COLLECTION_IS_NULL
END;
```

コレクション全体の比較

ネストした表と `varrays` は、アトミック `NULL` の場合があるため、次の例のようにして `NULL` かどうかをテストできます。

```
DECLARE
    TYPE Staff IS TABLE OF Employee;
    members Staff;
BEGIN
    ...
    IF members IS NULL THEN ... -- condition yields TRUE;
END;
```

しかし、コレクションの等価の比較はできません。たとえば、次の `IF` 条件は誤りです。

```
DECLARE
    TYPE Clientele IS TABLE OF Customer;
    group1 Clientele := Clientele(...);
    group2 Clientele := Clientele(...);
BEGIN
    ...
    IF group1 = group2 THEN -- causes compilation error
    ...
```

```
END IF;  
END;
```

この制限は、暗黙の比較にも適用されます。たとえば、コレクションは `DISTINCT`、`GROUP BY`、または `ORDER BY` リストには使えません。

コレクションの操作

PL/SQL 内では、コレクションを使うと、柔軟性と処理能力が向上します。大きな利点は、特定の要素を処理するために、プログラムで添字を計算できるということです。より大きな利点は、メモリー内コレクションを操作するために、プログラムで SQL を使用できるということです。

ネストした表の例

SQL*Plus で、次のようにオブジェクト型 `Course` を定義したとします。

```
SQL> CREATE TYPE Course AS OBJECT (  
2   course_no NUMBER(4),  
3   title      VARCHAR2(35),  
4   credits   NUMBER(1));
```

次に、`Course` オブジェクトを格納する `TABLE` 型 `CourseList` を定義します。

```
SQL> CREATE TYPE CourseList AS TABLE OF Course;
```

最後に、型 `CourseList` の列を含むデータベース表 `department` を次のようにして作ります。

```
SQL> CREATE TABLE department (  
2   name      VARCHAR2(20),  
3   director  VARCHAR2(20),  
4   office    VARCHAR2(20),  
5   courses   CourseList)  
6   NESTED TABLE courses STORE AS courses_tab;
```

列 `courses` の各項目は、指定された学部 (`department`) が提供するコースを格納するネストした表です。`department` にネストした表の列があるため、`NESTED TABLE` 句が必要です。この句は、ネストした表を識別し、システム生成された記憶域表の名前を指定しますが、Oracle はこの記憶域表に行外部 (別の表領域) にデータを格納します。

これで、データベース表 `department` にデータを入れることができます。次の例で、表のコンストラクタ `CourseList()` によって列 `courses` の値を指定する方法に注目してください。

```
BEGIN  
    INSERT INTO department
```

```
VALUES('Psychology', 'Irene Friedman', 'Fulton Hall 133',
       CourseList(Course(1000, 'General Psychology', 5),
                  Course(2100, 'Experimental Psychology', 4),
                  Course(2200, 'Psychological Tests', 3),
                  Course(2250, 'Behavior Modification', 4),
                  Course(3540, 'Groups and Organizations', 3),
                  Course(3552, 'Human Factors in Business', 4),
                  Course(4210, 'Theories of Learning', 4),
                  Course(4320, 'Cognitive Processes', 4),
                  Course(4410, 'Abnormal Psychology', 4)));
INSERT INTO department
VALUES('History', 'John Whalen', 'Applegate Hall 142',
       CourseList(Course(1011, 'History of Europe I', 4),
                  Course(1012, 'History of Europe II', 4),
                  Course(1202, 'American History', 5),
                  Course(2130, 'The Renaissance', 3),
                  Course(2132, 'The Reformation', 3),
                  Course(3105, 'History of Ancient Greece', 4),
                  Course(3321, 'Early Japan', 4),
                  Course(3601, 'Latin America Since 1825', 4),
                  Course(3702, 'Medieval Islamic History', 4)));
INSERT INTO department
VALUES('English', 'Lynn Saunders', 'Breakstone Hall 205',
       CourseList(Course(1002, 'Expository Writing', 3),
                  Course(2020, 'Film and Literature', 4),
                  Course(2418, 'Modern Science Fiction', 3),
                  Course(2810, 'Discursive Writing', 4),
                  Course(3010, 'Modern English Grammar', 3),
                  Course(3720, 'Introduction to Shakespeare', 4),
                  Course(3760, 'Modern Drama', 4),
                  Course(3822, 'The Short Story', 4),
                  Course(3870, 'The American Novel', 5)));
END;
```

次の例は、英語学部が提供するコースのリストを改訂します。

```
DECLARE
new_courses CourseList :=
    CourseList(Course(1002, 'Expository Writing', 3),
              Course(2020, 'Film and Literature', 4),
              Course(2810, 'Discursive Writing', 4),
              Course(3010, 'Modern English Grammar', 3),
              Course(3550, 'Realism and Naturalism', 4),
              Course(3720, 'Introduction to Shakespeare', 4),
              Course(3760, 'Modern Drama', 4),
              Course(3822, 'The Short Story', 4),
              Course(3870, 'The American Novel', 4),
              Course(4210, '20th-Century Poetry', 4),
```

```

        Course(4725, 'Advanced Workshop in Poetry', 5));
BEGIN
    UPDATE department
        SET courses = new_courses WHERE name = 'English';
END;

```

次の例では、心理学部が提供するすべてのコースを、ローカルなネストした表に取り出します。

```

DECLARE
    psyc_courses CourseList;
BEGIN
    SELECT courses INTO psyc_courses FROM department
        WHERE name = 'Psychology';
    ...
END;

```

varray の例

SQL*Plus で、次のようにオブジェクト型 `Project` を定義したとします。

```

SQL> CREATE TYPE Project AS OBJECT (
2   project_no NUMBER(2),
3   title      VARCHAR2(35),
4   cost       NUMBER(7,2));

```

次に、`Project` オブジェクトを格納する `VARRAY` 型 `ProjectList` を定義します。

```

SQL> CREATE TYPE ProjectList AS VARRAY(50) OF Project;

```

最後に、型 `ProjectList` の列を含む関係表 `department` を次のようにして作ります。

```

SQL> CREATE TABLE department (
2   dept_id  NUMBER(2),
3   name     VARCHAR2(15),
4   budget   NUMBER(11,2),
5   projects ProjectList);

```

列 `projects` 中の各項目は、指定された `department` で計画されているプロジェクトを格納する `varray` です。

これで、`department` 表にデータを入れる用意ができました。次の例で、`varray` コンストラクタ `ProjectList()` によって列 `projects` の値を指定する方法に注目してください。

```

BEGIN
    INSERT INTO department
        VALUES(30, 'Accounting', 1205700,
            ProjectList(Project(1, 'Design New Expense Report', 3250),
                Project(2, 'Outsource Payroll', 12350),

```

コレクションの操作

```
        Project(3, 'Evaluate Merger Proposal', 2750),
        Project(4, 'Audit Accounts Payable', 1425)));
INSERT INTO department
VALUES(50, 'Maintenance', 925300,
       ProjectList(Project(1, 'Repair Leak in Roof', 2850),
                   Project(2, 'Install New Door Locks', 1700),
                   Project(3, 'Wash Front Windows', 975),
                   Project(4, 'Repair Faulty Wiring', 1350),
                   Project(5, 'Winterize Cooling System', 1125)));
INSERT INTO department
VALUES(60, 'Security', 750400,
       ProjectList(Project(1, 'Issue New Employee Badges', 13500),
                   Project(2, 'Find Missing IC Chips', 2750),
                   Project(3, 'Upgrade Alarm System', 3350),
                   Project(4, 'Inspect Emergency Exits', 1900)));
END;
```

次の例では、セキュリティ部門に割り当てられているプロジェクトのリストを更新します。

```
DECLARE
    new_projects ProjectList :=
        ProjectList(Project(1, 'Issue New Employee Badges', 13500),
                    Project(2, 'Develop New Patrol Plan', 1250),
                    Project(3, 'Inspect Emergency Exits', 1900),
                    Project(4, 'Upgrade Alarm System', 3350),
                    Project(5, 'Analyze Local Crime Stats', 825));
BEGIN
    UPDATE department
        SET projects = new_projects WHERE dept_id = 60;
END;
```

次の例では、会計部門のすべてのプロジェクトを取り出してローカル varray に入れます。

```
DECLARE
    my_projects ProjectList;
BEGIN
    SELECT projects INTO my_projects FROM department
        WHERE dept_id = 30;
    ...
END;
```

次の最後の例では、会計部門、およびそのプロジェクト・リストを表 department から削除します。

```
BEGIN
    DELETE FROM department WHERE dept_id = 30;
END;
```


個々の要素の操作

ここまでは、コレクション全体を操作しました。SQL 内で、コレクションの個々の要素を操作するには、演算子 `TABLE` を使用します。`TABLE` のオペランドは、操作対象の単一の列値を戻す副問合せです。この値はネストした表か `varray` です。

次の例では、列 `courses` に格納される歴史学部ネストした表に行を追加します。

```
BEGIN
  INSERT INTO
    TABLE(SELECT courses FROM department WHERE name = 'History')
    VALUES(3340, 'Modern China', 4);
END;
```

次の例では、心理学部が提供する 2 つのコースの履修単位の数を変更します。

```
DECLARE
  adjustment INTEGER DEFAULT 1;
BEGIN
  UPDATE TABLE(SELECT courses FROM department
    WHERE name = 'Psychology')
    SET credits = credits + adjustment
    WHERE course_no IN (2200, 3540);
END;
```

次の例では、歴史学部が提供する特定のコースの番号とタイトルを取り出します。

```
DECLARE
  my_course_no NUMBER(4);
  my_title VARCHAR2(35);
BEGIN
  SELECT course_no, title INTO my_course_no, my_title
    FROM TABLE(SELECT courses FROM department
      WHERE name = 'History')
      WHERE course_no = 3105;
  ...
END;
```

次の例では、英語学部が提供する 5 つの履修コースをすべて削除します。

```
BEGIN
  DELETE TABLE(SELECT courses FROM department
    WHERE name = 'English')
    WHERE credits = 5;
END;
```

次の例では、管理部の 4 番目のプロジェクトのタイトルとコストを `varray` 列 `projects` から取り出します。

```
DECLARE
    my_cost  NUMBER(7,2);
    my_title VARCHAR2(35);
BEGIN
    SELECT cost, title INTO my_cost, my_title
    FROM TABLE(SELECT projects FROM department
    WHERE dept_id = 50)
    WHERE project_no = 4;
    ...
END;
```

現在のところ、varray の個々の要素を INSERT 文、UPDATE 文、または DELETE 文内で参照できません。PL/SQL プロシージャ文を使用する必要があります。次の例では、ストアド・プロシージャ add_project は、department のプロジェクト・リストの指定された位置に新しいプロジェクトを挿入します。

```
CREATE PROCEDURE add_project (
    dept_no      IN NUMBER,
    new_project  IN Project,
    position     IN NUMBER) AS
    my_projects ProjectList;
BEGIN
    SELECT projects INTO my_projects FROM department
    WHERE dept_no = dept_id FOR UPDATE OF projects;
    my_projects.EXTEND; -- make room for new project
    /* Move varray elements forward. */
    FOR i IN REVERSE position..my_projects.LAST - 1 LOOP
        my_projects(i + 1) := my_projects(i);
    END LOOP;
    my_projects(position) := new_project; -- add new project
    UPDATE department SET projects = my_projects
    WHERE dept_no = dept_id;
END add_project;
```

次のストアド・プロシージャは、指定したプロジェクトを更新します。

```
CREATE PROCEDURE update_project (
    dept_no  IN NUMBER,
    proj_no  IN NUMBER,
    new_title IN VARCHAR2 DEFAULT NULL,
    new_cost  IN NUMBER DEFAULT NULL) AS
    my_project ProjectList;
BEGIN
    SELECT projects INTO my_projects FROM department
    WHERE dept_no = dept_id FOR UPDATE OF projects;
    /* Find project, update it, then exit loop immediately. */
    FOR i IN my_projects.FIRST..my_projects.LAST LOOP
        IF my_projects(i).project_no = proj_no THEN
```

```

        IF new_title IS NOT NULL THEN
            my_projects(i).title := new_title;
        END IF;
        IF new_cost IS NOT NULL THEN
            my_projects(i).cost := new_cost;
        END IF;
        EXIT;
    END IF;
END LOOP;
UPDATE department SET projects = my_projects
WHERE dept_no = dept_id;
END update_project;

```

ローカル・コレクションの操作

SQL 内で、ローカル・コレクションを操作するには、演算子 `TABLE` と `CAST` を使用します。`CAST` のオペランドは、ローカルに（たとえば PL/SQL の無名ブロックなどで）宣言されたコレクションと SQL コレクション型です。`CAST` はローカル・コレクションを指定した型に変換します。このようにして、コレクションを SQL データベース表と同様に操作できます。次の例では、変更されたコース・リストとオリジナルとの相違点を数えます（コース 3720 の履修単位の数が 4 から 3 に変更されていることに注意してください）。

```

DECLARE
    revised CourseList :=
        CourseList(Course(1002, 'Expository Writing',      3),
                   Course(2020, 'Film and Literature',     4),
                   Course(2810, 'Discursive Writing',      4),
                   Course(3010, 'Modern English Grammar ', 3),
                   Course(3550, 'Realism and Naturalism',   4),
                   Course(3720, 'Introduction to Shakespeare', 3),
                   Course(3760, 'Modern Drama',            4),
                   Course(3822, 'The Short Story',         4),
                   Course(3870, 'The American Novel',      5),
                   Course(4210, '20th-Century Poetry',      4),
                   Course(4725, 'Advanced Workshop in Poetry', 5));
    num_changed INTEGER;
BEGIN
    SELECT COUNT(*) INTO num_changed
    FROM TABLE(CAST(revised AS CourseList)) AS new,
    TABLE(SELECT courses FROM department
           WHERE name = 'English') AS old
    WHERE new.course_no = old.course_no AND
          (new.title != old.title OR new.credits != old.credits);
    DBMS_OUTPUT.PUT_LINE(num_changed);
END;

```

コレクション・メソッドの使用

次に示すコレクション・メソッドは、コードを一般化したり、コレクションを使いやすくしたり、アプリケーションを維持しやすくしたりするのに使います。

```
EXISTS
COUNT
LIMIT
FIRST および LAST
PRIOR および NEXT
EXTEND
TRIM
DELETE
```

コレクション・メソッドとは、コレクションに対する操作を実行するための、ドット表記法を使ってコールされる組込みファンクションまたはプロシージャです。次に構文を示します。

```
collection_name.method_name[(parameters)]
```

コレクション・メソッドは、プロシージャ文からはコールできますが、SQL 文からはコールできません。EXISTS、COUNT、LIMIT、FIRST、LAST、PRIOR、および NEXT は、式の一部として使われるファンクションです。EXTEND、TRIM、DELETE は、文として使われるプロシージャです。また、EXISTS、PRIOR、NEXT、TRIM、EXTEND、DELETE はパラメータを取ります。各パラメータは整数式でなければなりません。

アトミック NULL であるコレクションに適用されるのは EXISTS だけです。それ以外のメソッドをそのようなコレクションに適用すると、PL/SQL は COLLECTION_IS_NULL を呼び出します。

EXISTS の使用

EXISTS(n) は、コレクションに *n* 番目の要素が存在する場合に TRUE を戻します。ない場合、EXISTS(n) は FALSE を戻します。主に EXISTS は、DELETE とともに、疎であるネストした表のメンテナンスのために使います。また、EXISTS を使うことによって、存在しない要素を参照した場合に発生する例外を避けることができます。次の例では、要素 *i* が存在する場合にだけ代入文が実行されます。

```
IF courses.EXISTS(i) THEN courses(i) := new_course; END IF;
```

範囲外の添字を渡した場合、EXISTS は SUBSCRIPT_OUTSIDE_LIMIT を呼び出さずに、FALSE を戻します。

COUNT の使用

COUNT は、コレクションに現在含まれている要素の数を戻します。たとえば、`varrayprojects` に 15 個の要素が含まれる場合、次の IF 条件は TRUE です。

```
IF projects.COUNT = 15 THEN ...
```

コレクションの現在のサイズは不明の場合があるので、そのようなとき COUNT が役立ちます。たとえば、Oracle データの列を取り出してネストした表に入れると、表には何個の要素が入れられるでしょうか。COUNT で答えを出すことができます。

COUNT は、整数式が使える位置ならどこでも使えます。次の例では、COUNT を使って、ループ範囲の上限を指定しています。

```
FOR i IN 1..courses.COUNT LOOP ...
```

varray の場合、COUNT は常に LAST と同じです。ネストした表の場合、COUNT は通常、LAST と同じです。しかし、ネストした表の途中から要素を削除すると、COUNT は LAST より小さくなります。

要素を総計する際、COUNT は削除された要素を無視します。

LIMIT の使用

最大サイズがないネストした表の場合、LIMIT は NULL を戻します。varray の場合、LIMIT は varray に入れることのできる（型定義で指定する必要がある）要素の最大数を戻します。たとえば、`varrayprojects` の最大要素数が 25 個である場合、次の IF 条件は TRUE です。

```
IF projects.LIMIT = 25 THEN ...
```

LIMIT は、整数式が使える位置ならどこでも使えます。次の例では、LIMIT を使って、`varrayprojects` にさらに 20 の要素を追加できるかどうかを調べています。

```
IF (projects.COUNT + 20) < projects.LIMIT THEN ...
```

FIRST および LAST の使用

FIRST と LAST は、それぞれコレクションの最初と最後（最小と最大）の索引番号を戻します。コレクションが空なら、FIRST と LAST は NULL を戻します。コレクションに含まれる要素の数が 1 つだけの場合、次の例に示すように FIRST と LAST は同じ索引番号を戻します。

```
IF courses.FIRST = courses.LAST THEN ... -- only one element
```

次の例に示すように、FIRST と LAST を使って、ループ範囲の下限と上限を指定できます（ただし、その範囲内にそれぞれの要素が存在することが必要です）。

```
FOR i IN courses.FIRST..courses.LAST LOOP ...
```

実際には、FIRST または LAST は、整数式が使える位置ならどこでも使えます。次の例では、FIRST を使ってループ・カウンタを初期化しています。

```
i := courses.FIRST;
WHILE i IS NOT NULL LOOP ...
```

varray の場合、FIRST は常に 1 を返し、LAST は常に COUNT と同じです。ネストした表の場合、FIRST は通常は 1 を返します。ただし、ネストした表の先頭から要素を削除すると、FIRST は 1 よりも大きい数値を返します。また、ネストした表の場合、LAST は通常は COUNT と同じです。しかし、ネストした表の途中から要素を削除すると、LAST は COUNT より大きくなります。

要素を走査する際、FIRST および LAST は削除された要素を無視します。

PRIOR および NEXT の使用

PRIOR(n) は、コレクションの索引 n の前の索引番号を返します。NEXT(n) は、索引 n の後の索引番号を返します。n の前の番号がない場合、PRIOR(n) は NULL を返します。同様に、n の後の番号がない場合、NEXT(n) は NULL を返します。

PRIOR と NEXT は、コレクションの 1 つの端からもう一方の端に折り返すことはありません。たとえば次の文では、コレクションの第 1 要素には先行する要素がないため、n には NULL が代入されます。

```
n := courses.PRIOR(courses.FIRST); -- assigns NULL to n
```

PRIOR は NEXT の逆です。たとえば、要素 i が存在する場合、次の文によって要素 i がそれ自身に割り当てられます。

```
projects(i) := projects.PRIOR(projects.NEXT(i));
```

PRIOR または NEXT を使うことにより、任意の添字列を索引とするコレクション内を移動できます。次の例では、NEXT を使って、いくつかの要素が削除されたネストした表内を移動しています。

```
i := courses.FIRST; -- get subscript of first element
WHILE i IS NOT NULL LOOP
    -- do something with courses(i)
    i := courses.NEXT(i); -- get subscript of next element
END LOOP;
```

要素間を移動する際、PRIOR および NEXT は削除された要素を無視します。

EXTEND の使用

コレクションのサイズを大きくするには、EXTEND を使います。このプロシージャには 3 つの形式があります。EXTEND は、コレクションに 1 つの NULL 要素を追加します。

EXTEND(n) は、コレクションに n 個の NULL 要素を追加します。EXTEND(n,i) は、コレ

クションに i 番目の要素のコピーを n 個追加します。たとえば次の文は、要素 1 のコピーをネストした表の `courses` に 5 個追加します。

```
courses.EXTEND(5,1);
```

EXTEND を使って、アトミック NULL であるコレクションの初期化はできません。また、NOT NULL 制約を TABLE または VARRAY 型に指定した場合、EXTEND の最初の 2 つの形式はその型のコレクションに適用できません。

EXTEND は、削除された要素を含むコレクションの内部サイズに対して操作します。そのため、EXTEND は削除された要素を見つけると、それらの要素を数に含めます。PL/SQL は、必要なら削除された要素を置き換えることができるように、それらの要素のプレースホルダを保持します。次の例を考えてみます。

```
DECLARE
    TYPE CourseList IS TABLE OF VARCHAR2(10);
    courses CourseList;
BEGIN
    courses := CourseList('Biol 4412', 'Psyc 3112', 'Anth 3001');
    courses.DELETE(3); -- delete element 3
    /* PL/SQL keeps a placeholder for element 3. So, the
       next statement appends element 4, not element 3. */
    courses.EXTEND; -- append one null element
    /* Now element 4 exists, so the next statement does
       not raise SUBSCRIPT_BEYOND_COUNT. */
    courses(4) := 'Engl 2005';
```

削除された要素を含めると、ネストした表の内部サイズは、COUNT と LAST が戻す値とは異なります。たとえば、ネストした表を 5 つの要素で初期化してから、要素 2 と要素 5 を削除した場合、内部サイズは 5 で、COUNT は 3 を返し、LAST は 4 を返します。削除された要素（先頭、中間、末尾のいずれでも）はすべて同様に処理されます。

TRIM の使用

このプロシージャには 2 つの形式があります。TRIM は、コレクションの末尾から 1 つの要素を削除します。TRIM(n) は、コレクションの末尾から n 個の要素を削除します。たとえば次の文は、ネストした表の `courses` から最後の 3 つの要素を削除します。

```
courses.TRIM(3);
```

n が COUNT より大きいと、TRIM(n) は SUBSCRIPT_BEYOND_COUNT を呼び出します。

TRIM は、コレクションの内部サイズに対して操作します。そのため、TRIM は削除された要素を見つけると、それらの要素を数に含めます。次の例を考えてみます。

```
DECLARE
    TYPE CourseList IS TABLE OF VARCHAR2(10);
    courses CourseList;
```

```
BEGIN
  courses := CourseList('Biol 4412', 'Psyc 3112', 'Anth 3001');
  courses.DELETE(courses.LAST); -- delete element 3
  /* At this point, COUNT equals 2, the number of valid
     elements remaining. So, you might expect the next
     statement to empty the nested table by trimming
     elements 1 and 2. Instead, it trims valid element 2
     and deleted element 3 because TRIM includes deleted
     elements in its tally. */
  courses.TRIM(courses.COUNT);
  DBMS_OUTPUT.PUT_LINE(courses(1)); -- prints 'Biol 4412'
```

一般に、TRIM と DELETE の間の相互作用には依存しないようにしてください。ネストした表は、固定サイズの配列のように扱って DELETE だけを使うか、またはスタックのように扱って TRIM と EXTEND だけを使うとよいでしょう。

PL/SQL は切り捨てられた (TRIM) 要素のプレースホルダは保持しません。そのため、切り捨てられた要素に単に新しい値を代入するだけではその要素を置き換えることができません。

DELETE の使用

このプロシージャには 3 つの形式があります。DELETE は、コレクションからすべての要素を削除します。DELETE (n) は、ネストした表から n 番目の要素を削除します。n が NULL である場合、DELETE (n) は何も実行しません。DELETE (m, n) は、索引付き表またはネストした表から $m \sim n$ の範囲のすべての要素を削除します。m が n より大きい場合、または m が n が NULL である場合、DELETE (m, n) は何も実行しません。次に例を示します。

```
BEGIN
  ...
  courses.DELETE(2); -- deletes element 2
  courses.DELETE(7,7); -- deletes element 7
  courses.DELETE(6,3); -- does nothing
  courses.DELETE(3,6); -- deletes elements 3 through 6
  projects.DELETE; -- deletes all elements
END;
```

varray は密であるため、個々の要素は削除できません。

削除対象の要素が存在しない場合でも、DELETE は単にその要素をスキップするので、例外は呼び出されません。PL/SQL は、削除された要素のプレースホルダを保持します。そのため、削除された要素に単に新しい値を代入するだけでその要素を置き換えることができません。

DELETE を使うことにより、疎であるネストした表を維持できます。次の例では、ネストした表 prospects を一時表に入れ、データを削除してから、再びデータベースに格納します。


```

DECLARE
    my_prospects ProspectList;
    revenue      NUMBER;
BEGIN
    SELECT prospects INTO my_prospects FROM customers WHERE ...
    FOR i IN my_prospects.FIRST..my_prospects.LAST LOOP
        estimate_revenue(my_prospects(i), revenue); -- call procedure
        IF revenue < 25000 THEN
            my_prospects.DELETE(i);
        END IF;
    END LOOP;
    UPDATE customers SET prospects = my_prospects WHERE ...

```

ネストした表に割り当てられるメモリーの量は、動的に増減します。要素を削除すると、メモリーはページ単位で解放されます。表全体を削除した場合は、すべてのメモリーが解放されます。

メソッドをコレクション・パラメータに適用する

サブプログラム内で、コレクション・パラメータは引数のプロパティがバインドされていることを前提にしています。そのため、組み込みコレクション・メソッド（FIRST、LAST、COUNT など）をそのようなパラメータに適用できます。次の例では、ネストした表をパッケージ・プロシージャの仮パラメータとして宣言しています。

```

CREATE PACKAGE personnel AS
    TYPE Staff IS TABLE OF Employee;
    ...
    PROCEDURE award_bonuses (members IN Staff);
END personnel;
CREATE PACKAGE BODY personnel AS
    ...
    PROCEDURE award_bonuses (members IN Staff) IS
    BEGIN
        ...
        IF members.COUNT > 10 THEN -- apply method
            ...
        END IF;
    END;
END personnel;

```

注意：varray パラメータの場合、パラメータ・モードに関係なく、LIMIT の値は常にパラメータの型定義から派生します。

コレクション例外の回避

ほとんどの場合、存在しないコレクション要素を参照すると、PL/SQL は事前定義された例外を呼び出します。次の例を考えてみます。

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  nums NumList; -- atomically null
BEGIN
  /* Assume execution continues despite the raised exceptions. */
  nums(1) := 1; -- raises COLLECTION_IS_NULL (1)
  nums := NumList(1,2); -- initialize table
  nums(NULL) := 3 -- raises VALUE_ERROR (2)
  nums(0) := 3; -- raises SUBSCRIPT_OUTSIDE_LIMIT (3)
  nums(3) := 3; -- raises SUBSCRIPT_BEYOND_COUNT (4)
  nums.DELETE(1); -- delete element 1
  IF nums(1) = 1 THEN ... -- raises NO_DATA_FOUND (5)
```

最初のケースでは、ネストした表はアトミック NULL です。2 番目のケースでは、添字が NULL です。3 番目のケースでは、添字は有効範囲外です。4 番目のケースでは、添字は表の要素数を超えています。5 番目のケースでは、添字は削除された要素を指定しています。

次のリストは、指定された例外が呼び出される場合を示しています。

例外	呼び出される場合
COLLECTION_IS_NULL	アトミック NULL のコレクションに対して操作する
NO_DATA_FOUND	添字が削除された要素を指定している
SUBSCRIPT_BEYOND_COUNT	添字がコレクションの中の要素数を超えている
SUBSCRIPT_OUTSIDE_LIMIT	添字が有効範囲外である
VALUE_ERROR	添字が NULL、または整数に変換できない

場合によっては、例外を呼び出さずに「無効な」添字をメソッドに渡せます。たとえば、NULL 添字をプロシージャ DELETE に渡しても、何も実行されません。また、次の例のようにすれば、削除された要素を、NO_DATA_FOUND を呼び出さずに置き換えることができます。

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  nums NumList := NumList(10,20,30); -- initialize table
BEGIN
  ...
  nums.DELETE(-1); -- does not raise SUBSCRIPT_OUTSIDE_LIMIT
  nums.DELETE(3); -- delete 3rd element
  DBMS_OUTPUT.PUT_LINE(nums.COUNT); -- prints 2
```

```

      nums(3) := 30;      -- legal; does not raise NO_DATA_FOUND
      DBMS_OUTPUT.PUT_LINE(nums.COUNT); -- prints 3
END;
```

パッケージ・コレクション型とローカル・コレクション型は互換性がありません。たとえば、次のパッケージ・プロシージャをコールするとします。

```

CREATE PACKAGE pkg1 AS
  TYPE NumList IS VARRAY(25) OF NUMBER(4);
  PROCEDURE delete_emps (emp_list NumList);
  ...
END pkg1;

CREATE PACKAGE BODY pkg1 AS
  PROCEDURE delete_emps (emp_list NumList) IS ...
  ...
END pkg1;
```

次に示す PL/SQL ブロックを実行すると、2 番目のプロシージャ・コールは *wrong number or types of arguments* エラーで失敗します。これは、パッケージおよびローカルの VARRAY 型は定義が同一でも互換性がないためです。

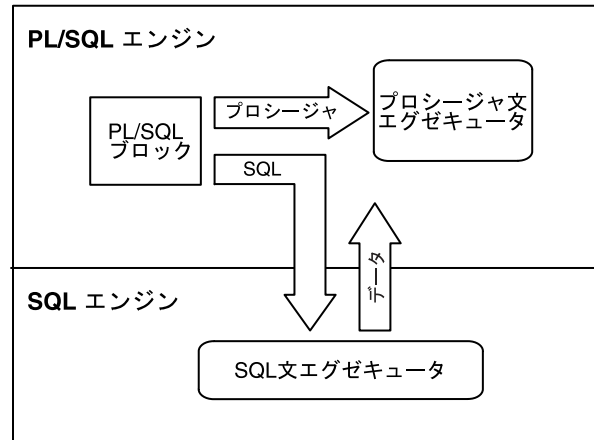
```

DECLARE
  TYPE NumList IS VARRAY(25) OF NUMBER(4);
  emps  pkg1.NumList := pkg1.NumList(7369, 7499);
  emps2 NumList := NumList(7521, 7566);
BEGIN
  pkg1.delete_emps(emps);
  pkg1.delete_emps(emps2); -- causes a compilation error
END;
```

バルク・バインドの利用

Oracle RDBMS に埋め込まれていると、PL/SQL エンジンでは任意の適切な PL/SQL ブロックまたはサブプログラムを受け付けます。図 4-3 に示すように、PL/SQL エンジンではプロシージャ文を実行しますが、SQL 文を SQL エンジンに送信します。SQL エンジンでは SQL 文を実行し、場合によってはデータを PL/SQL エンジンに戻します。

図 4-3 コンテキスト切替え



PL/SQL と SQL エンジンの間でのコンテキスト切替えを行うたびに、オーバーヘッドが増加します。このため、切替えが何度も必要な場合、パフォーマンスに影響します。これは、コレクション（索引付き表、ネストした表、varray またはホスト配列）の要素をバインド変数として使用してループ内で SQL 文が実行されると発生します。たとえば、次の DELETE 文は、FOR の反復ごとに SQL エンジンに送信されます。

```

DECLARE
    TYPE NumList IS VARRAY(20) OF NUMBER;
    depts NumList := NumList(10, 30, 70, ...); -- department numbers
BEGIN
    ...
    FOR i IN depts.FIRST..depts.LAST LOOP
        ...
        DELETE FROM emp WHERE deptno = depts(i);
    END LOOP;
END;

```

この場合、SQL 文が 5 つ以上のデータベース行に影響するのであれば、バルク・バインドを使用するとパフォーマンスが向上します。

バルク・バインドによるパフォーマンスの向上

SQL 文の PL/SQL 変数への値の代入は、バインドと呼ばれます。コレクション全体のバインドを一度に行うことを、バルク・バインドと呼びます。バルク・バインドは、PL/SQL エンジンと SQL エンジンの間でのコンテキスト切替えの回数を最小限に抑えることによって、パフォーマンスを向上します。バルク・バインドでは、個々の要素ではなくコレクション全体がやりとりされます。たとえば、次の DELETE 文は、ネストした表全体とともに、一度だけ SQL エンジンに送信されます。

```

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  mgrs NumList := NumList(7566, 7782, ...); -- manager numbers
BEGIN
  ...
  FORALL i IN mgrs.FIRST..mgrs.LAST
    DELETE FROM emp WHERE mgr = mgrs(i);
END;

```

次の例では、5000 の部品番号と名前を索引付き表にロードします。そして、すべての表要素をデータベース表に 2 度挿入します。まず、FOR ループを使用して挿入します。これは 38 秒で完了します。次に、FORALL 文を使用して一括挿入します。これはわずか 3 秒で完了します。

```

SQL> SET SERVEROUTPUT ON
SQL> CREATE TABLE parts (pnum NUMBER(4), pname CHAR(15));

Table created.

SQL> GET test.sql
1  DECLARE
2      TYPE NumTab IS TABLE OF NUMBER(4) INDEX BY BINARY_INTEGER;
3      TYPE NameTab IS TABLE OF CHAR(15) INDEX BY BINARY_INTEGER;
4      pnums NumTab;
5      pnames NameTab;
6      t1 CHAR(5);
7      t2 CHAR(5);
8      t3 CHAR(5);
9      PROCEDURE get_time (t OUT NUMBER) IS
10     BEGIN SELECT TO_CHAR(SYSDATE, 'SSSSS') INTO t FROM dual; END;
11 BEGIN
12     FOR j IN 1..5000 LOOP -- load index-by tables
13         pnums(j) := j;
14         pnames(j) := 'Part No. ' || TO_CHAR(j); 15     END LOOP;
16     get_time(t1);
17     FOR i IN 1..5000 LOOP -- use FOR loop
18         INSERT INTO parts VALUES (pnums(i), pnames(i));
19     END LOOP;
20     get_time(t2);
21     FORALL i IN 1..5000 -- use FORALL statement
22         INSERT INTO parts VALUES (pnums(i), pnames(i));
23     get_time(t3);
24     DBMS_OUTPUT.PUT_LINE('Execution Time (secs)');
25     DBMS_OUTPUT.PUT_LINE('-----');
26     DBMS_OUTPUT.PUT_LINE('FOR loop: ' || TO_CHAR(t2 - t1));
27     DBMS_OUTPUT.PUT_LINE('FORALL:  ' || TO_CHAR(t3 - t2));
28* END;

```

```
SQL> /
Execution Time (secs)
-----
FOR loop: 38
FORALL:    3

PL/SQL procedure successfully completed.
```

コレクションをバルク・バインド入力するには、FORALL 文を使用します。コレクションをバルク・バインド出力するには、BULK COLLECT 句を使用します。

FORALL 文の使用方法

キーワード FORALL は、コレクションを SQL エンジンに送信する前にバルク・バインド入力するよう、PL/SQL エンジンに指示を与えます。FORALL 文は反復スキームを含んでいますが、FOR ループではありません。次に構文を示します。

```
FORALL index IN lower_bound..upper_bound
    sql_statement;
```

索引は、コレクションの添字として、FORALL 文の中でだけ参照できます。SQL 文は、コレクション要素を参照する INSERT 文、UPDATE 文または DELETE 文でなければなりません。さらに、境界には連続した索引番号の有効な範囲を指定しなければなりません。SQL エン진은、範囲内の各索引番号に対して一度ずつ SQL 文を実行します。次の例に示すように、境界を使ってコレクションの任意のスライスをバルク・バインドできます。

```
DECLARE
    TYPE NumList IS VARRAY(15) OF NUMBER;
    depts NumList := NumList();
BEGIN
    -- fill varray here
    ...
    FORALL j IN 6..10 -- bulk-bind middle third of varray
        UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(j);
END;
```

SQL 文は複数のコレクションを参照できます。しかし、PL/SQL エンジンには添字付きコレクションだけをバルク・バインドします。このため、次の例では、ファンクション median に渡されるコレクション sals はバルク・バインドされません。

```
FORALL i IN 1..20
    INSERT INTO emp2 VALUES (enums(i), names(i), median(sals), ...);
```

次の例は、コレクションの添字を式には使用できないことを示します。

```
FORALL j IN mgrs.FIRST..mgrs.LAST
    DELETE FROM emp WHERE mgr = mgrs(j+1); -- illegal subscript
```

指定した範囲のコレクション要素がすべて存在しなければなりません。要素が足りなかったり削除されていた場合は、エラーが発生します。次に例を示します。

```
DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    depts NumList := NumList(10, 20, 30, 40);
BEGIN
    depts.DELETE(3); -- delete third element
    FORALL i IN depts.FIRST..depts.LAST
        DELETE FROM emp WHERE deptno = depts(i);
    -- raises an "element does not exist" exception
END;
```

ロールバックの動作

FORALL 文が失敗すると、データベースの変更は、各 SQL 文の実行の前にマークされた暗黙のセーブポイントまでロールバックされます。前の実行の間に行われた変更は、ロールバックされません。たとえば次に示すように、部門番号と肩書きを格納するデータベース表を作成するとします。

```
CREATE TABLE emp2 (deptno NUMBER(2), job VARCHAR2(15));
```

次に、表に行を挿入します。

```
INSERT INTO emp2 VALUES(10, 'Clerk');
INSERT INTO emp2 VALUES(10, 'Clerk');
INSERT INTO emp2 VALUES(20, 'Bookkeeper'); -- 10-char job title
INSERT INTO emp2 VALUES(30, 'Analyst');
INSERT INTO emp2 VALUES(30, 'Analyst');
```

次の UPDATE 文を使用して、7 文字の文字列 ' (temp)' を特定の肩書きに追加します。

```
DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    depts NumList := NumList(10, 20);
BEGIN
    FORALL j IN depts.FIRST..depts.LAST
        UPDATE emp2 SET job = job || ' (temp)'
        WHERE deptno = depts(j);
    -- raises a "value too large" exception
EXCEPTION
    WHEN OTHERS THEN
        COMMIT;
END;
```

SQL エンジンは、指定した範囲内の各索引番号に対して 1 回ずつ、UPDATE 文を 2 回実行します。つまり、depts(10) に対して 1 回、depts(20) に対して 1 回です。文字列値

'Bookkeeper (temp)' は、job 列には長すぎるため、最初の実行は成功しますが 2 回目の実行は失敗します。この場合、2 回目の実行だけがロールバックされます。

BULK COLLECT 句の使用法

キーワード BULK COLLECT は、コレクションを PL/SQL エンジンに戻す前にバルク・バインド出力するよう、SQL エンジンに指示を与えます。このキーワードは、SELECT INTO 句、FETCH INTO 句、RETURNING INTO 句で使用できます。次に構文を示します。

```
... BULK COLLECT INTO collection_name[, collection_name] ...
```

SQL エンジンでは、INTO リスト内で参照されるすべてのコレクションをバルク・バインドします。対応する列には、スカラー値（複合ではない）が格納されている必要があります。次の例では、SQL エンジンでは、ネストした表を PL/SQL エンジンに戻す前に、empno および ename データベース列全体をネストした表にロードします。

```
DECLARE
    TYPE NumTab IS TABLE OF emp.empno%TYPE;
    TYPE NameTab IS TABLE OF emp.ename%TYPE;
    enums NumTab; -- no need to initialize
    names NameTab;
BEGIN
    SELECT empno, ename BULK COLLECT INTO enums, names FROM emp;
    ...
END;
```

SQL エンジンはコレクションを初期化、拡張します。（ただし、最大サイズを超えては varray を拡張できません。）次に、索引 1 から順に要素を挿入し、既存の要素を上書きします。

SQL エンジンはデータベース列全体をバルク・バインドします。つまり、表に 50,000 行ある場合、エンジンは 50,000 列値をターゲットのコレクションにロードします。ただし、疑似列 ROWNUM 使用して、処理される行の数を制限できます。次の例では、行の数を 100 に制限します。

```
DECLARE
    TYPE NumTab IS TABLE OF emp.empno%TYPE;
    sals NumTab;
BEGIN
    SELECT sal BULK COLLECT INTO sals FROM emp WHERE ROWNUM <= 100;
    ...
END;
```

バルク・フェッチ

次の例では、1 つのカーソルから 1 つ以上のコレクションにバルク・フェッチを行います。

```
DECLARE
```



```
TYPE NameTab IS TABLE OF emp.ename%TYPE;
TYPE SalTab IS TABLE OF emp.sal%TYPE;
names NameTab;
sals SalTab;
CURSOR c1 IS SELECT ename, sal FROM emp WHERE sal > 1000;
BEGIN
  OPEN c1;
  FETCH c1 BULK COLLECT INTO names, sals;
  ...
END;
```

制限事項：次の例に示すように、1つのカーソルから複数のレコードのコレクション（1つ）にはバルク・フェッチできません。

```
DECLARE
TYPE EmpRecTab IS TABLE OF emp%ROWTYPE;
emp_recs EmpRecTab;
CURSOR c1 IS SELECT ename, sal FROM emp WHERE sal > 1000;
BEGIN
  OPEN c1;
  FETCH c1 BULK COLLECT INTO emp_recs; -- illegal
  ...
END;
```

FORALL と BULK COLLECT の使用

BULK COLLECT 句を FORALL 文と組み合わせることができます。この場合、SQL エンジン は列値を段階的にバルク・バインドします。次の例では、コレクション depts に 3 つの要素があり、それぞれによって 5 行ずつ削除される場合、コレクション enums は、文が完了すると 15 の要素を持ちます。

```
FORALL j IN depts.FIRST..depts.LAST
  DELETE FROM emp WHERE empno = depts(j)
  RETURNING empno BULK COLLECT INTO enums;
```

各実行によって戻された列の値は、前に戻された値に追加されます。（FOR ループでは、前の値は上書きされます。）

制限事項：SELECT ... BULK COLLECT 文は、FORALL 文では使用できません。

ホスト配列の使用

クライアント側のプログラムは、無名 PL/SQL ブロックを使用してホスト配列をバルク・バインド入出力できます。これは、コレクションをデータベース・サーバーとの間でやりとりするのに最も効率的な方法です。

ホスト配列は OCI または Pro*C プログラムなどのホスト環境で宣言され、PL/SQL コレクションと区別するためのコロンを接頭辞としてつける必要があります。次の例では、DELETE 文に入力ホスト配列が使われています。実行時に、無名 PL/SQL ブロックがデータベース・サーバーに送信されて、実行されます。

```
DECLARE
    ...
BEGIN
    -- assume that values were assigned to the host array
    -- and host variables in the host environment
    FORALL i IN :lower...:upper
        DELETE FROM emp WHERE deptno = :depts(i);
    ...
END;
```

FIRST や LAST などのコレクション・メソッドは、ホスト配列とともに使用できません。たとえば、次の文は誤りです。

```
FORALL i IN :depts.FIRST...:depts.LAST -- illegal
    DELETE FROM emp WHERE deptno = :depts(i);
```

カーソル属性の使用

SQL データ操作文を処理するには、SQL エンジンが SQL という名前の暗黙カーソルをオープンします。このカーソルの属性 (%FOUND、%ISOPEN、%NOTFOUND、%ROWCOUNT) は、直前に実行された SQL データ操作文についての有用な情報を戻します。

SQL カーソルには、複合属性 %BULK_ROWCOUNT が 1 つだけあります。これは索引付き表の意味を持っています。i 番目の要素には、SQL 文の i 番目の実行によって処理された行の数が格納されます。i 番目の実行によって影響を受ける行がない場合、%BULK_ROWCOUNT(i) はゼロを戻します。たとえば、

```
DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    depts NumList := NumList(10, 20, 50);
BEGIN
    FORALL j IN depts.FIRST...:depts.LAST
        UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(j);
    IF SQL%BULK_ROWCOUNT(3) = 0 THEN
        ...
    END IF
END;
```

%BULK_ROWCOUNT と FORALL 文は同じ添字を使用します。たとえば、FORALL 文が範囲 -5..10 を使用する場合、%BULK_ROWCOUNT もそれを使用します。

注意：索引付きの表だけが負の添字を使用できます。

バルク・バインドには、スカラー属性の %FOUND、%NOTFOUND、%ROWCOUNT も使用できません。たとえば、%ROWCOUNT は、SQL 文のすべての実行によって処理された行の総数を戻します。

%FOUND と %NOTFOUND は、SQL 文の最後の実行だけを参照します。ただし、%BULK_ROWCOUNT を使用して個々の実行に対する値を推論することができます。たとえば、%BULK_ROWCOUNT(i) がゼロの場合、%FOUND と %NOTFOUND はそれぞれ、FALSE および TRUE になります。

制限事項：%BULK_ROWCOUNT は他のコレクションには代入できません。また、パラメータとしてサブプログラムに渡すことはできません。

レコードとは？

レコードはフィールドに格納されるいくつかの関連したデータ項目のグループであり、それぞれに独自の名前とデータ型があります。名前、給与、入社日など、従業員に関するさまざまなデータがあるとします。これらの項目は論理的に関連していますが、データ型は異なります。レコードには各項目を表すフィールドが入っています。レコードを使うと、データを 1 つの論理単位として扱うことができます。レコードを使えば、情報の編成と表示が容易になります。

属性 %ROWTYPE を使用すれば、データベースの表の中の行を表すレコードを宣言できます。しかし、フィールドのデータ型をレコードの中で指定したり、独自のフィールドを宣言したりできません。データ型の RECORD を使うと、それらの制限を回避し、独自のレコードを定義できます。

レコードの定義と宣言

レコードを作るには、RECORD 型を定義してから、その型のレコードを宣言します。RECORD 型は、次の構文を使って、任意の PL/SQL ブロック、サブプログラム、またはパッケージの宣言部で定義できます。

```
TYPE type_name IS RECORD (field_declaration[,field_declaration]...);
```

ここで、field_declaration は次のことを意味します。

```
field_name field_type [[NOT NULL] {:= | DEFAULT} expression]
```

ここで、type_name はレコードを宣言するために後で使われる型指定子、field_type は REF CURSOR を除く PL/SQL データ型、expression は結果が field_type と同じ型の値となる式です。

注意：TABLE 型や VARRAY 型とは異なり、RECORD 型はデータベース内で CREATE を使って作成したり格納したりできません。

%TYPE および %ROWTYPE を使ってフィールド型を指定できます。次の例では、DeptRec という名前の RECORD 型を定義します。

```
DECLARE
    TYPE DeptRec IS RECORD (
        dept_id    dept.deptno%TYPE,
        dept_name  VARCHAR2(15),
        dept_loc   VARCHAR2(15));
```

フィールドの宣言は変数の宣言と似ていることに注意してください。個々のフィールドは一意の名前と特定のデータ型を持っています。そのため、レコードの値は、単純な型の値の集まりです。

次の例に示すように、PL/SQL を使用するとオブジェクト、コレクション、およびその他のレコード（ネストしたレコードと呼ぶ）を含むレコードを定義できます。しかし、オブジェクト型には RECORD 型の属性を指定できません。

```
DECLARE
    TYPE TimeRec IS RECORD (
        seconds SMALLINT,
        minutes SMALLINT,
        hours    SMALLINT);
    TYPE FlightRec IS RECORD (
        flight_no    INTEGER,
        plane_id     VARCHAR2(10),
        captain      Employee, -- declare object
        passengers   PassengerList, -- declare varray
        depart_time  TimeRec, -- declare nested record
        airport_code VARCHAR2(10));
```

次の例に示すように、ファンクション仕様部の RETURN 句の中に RECORD 型を指定できます。こうすると、ファンクションは同じ型のユーザー定義のレコードを戻します。

```
DECLARE
  TYPE EmpRec IS RECORD (
    emp_id    INTEGER
    last_name VARCHAR2(15),
    dept_num  INTEGER(2),
    job_title VARCHAR2(15),
    salary    REAL(7,2));
  ...
  FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRec IS ...
```

レコードの宣言

一度 RECORD 型を定義すれば、次の例に示すように、その型のレコードを宣言できます。

```
DECLARE
  TYPE StockItem IS RECORD (
    item_no    INTEGER(3),
    description VARCHAR2(50),
    quantity   INTEGER,
    price      REAL(7,2));
  item_info StockItem; -- declare record
```

識別子 item_info はレコード全体を表します。

スカラー変数と同様に、ユーザー定義のレコードもプロシージャやファンクションの仮パラメータとして宣言できます。たとえば、

```
DECLARE
  TYPE EmpRec IS RECORD (
    emp_id    emp.empno%TYPE,
    last_name VARCHAR2(10),
    job_title VARCHAR2(15),
    salary    NUMBER(7,2));
  ...
  PROCEDURE raise_salary (emp_info EmpRec);
```

レコードの初期化と参照

次の例に示すように、型の定義の中でレコードを初期化できます。型 `TimeRec` のレコードを宣言すると、その 3 つのフィールドは初期値が 0 であるとみなされます。

```
DECLARE
    TYPE TimeRec IS RECORD (
        secs SMALLINT := 0,
        mins SMALLINT := 0,
        hrs  SMALLINT := 0);
```

次の例で示すように、どのフィールドにも `NOT NULL` 制約を指定することにより、フィールドに `NULL` が代入されないようにすることができます。`NOT NULL` と宣言されたフィールドは、初期化されていなければなりません。

```
DECLARE
    TYPE StockItem IS RECORD (
        item_no      INTEGER(3) NOT NULL := 999,
        description  VARCHAR2(50),
        quantity     INTEGER,
        price        REAL(7,2));
```

レコードの参照

添字を使ってアクセスするコレクション中の要素とは異なり、レコード中のフィールドは名前前でアクセスします。個々のフィールドを参照するには、ドット表記法と次の構文を使います。

```
record_name.field_name
```

たとえば、レコード `emp_info` 中のフィールド `hire_date` は、次のように参照します。

```
emp_info.hire_date ...
```

ユーザー定義のレコードを戻すファンクションをコールする場合、次の構文を使ってレコード内のフィールドを参照します。

```
function_name(parameter_list).field_name
```

たとえば、次に示すファンクション `nth_highest_sal` のコールでは、レコード `emp_info` 中のフィールド `salary` を参照しています。

```
DECLARE
    TYPE EmpRec IS RECORD (
        emp_id      NUMBER(4),
        job_title   CHAR(14),
        salary      REAL(7,2));
    middle_sal REAL;
    FUNCTION nth_highest_sal (n INTEGER) RETURN EmpRec IS
```

```

        emp_info EmpRec;
BEGIN
    ...
    RETURN emp_info; -- return record
END;
BEGIN
    middle_sal := nth_highest_sal(10).salary; -- call function

```

パラメータなしでファンクションをコールする場合、次の構文を使います。

```
function_name().field_name -- note empty parameter list
```

ファンクションから戻されるレコード内のネストされたフィールドを参照するには、拡張されたドット表記法を使います。次に構文を示します。

```
function_name(parameter_list).field_name.nested_field_name
```

たとえば、次に示すファンクション `item` へのコールは、レコード `item_info` の中のネストしたフィールド `minutes` を参照します。

```

DECLARE
    TYPE TimeRec IS RECORD (minutes SMALLINT, hours SMALLINT);
    TYPE AgendaItem IS RECORD (
        priority INTEGER,
        subject VARCHAR2(100),
        duration TimeRec);
    FUNCTION item (n INTEGER) RETURN AgendaItem IS
        item_info AgendaItem;
    BEGIN
        ...
        RETURN item_info; -- return record
    END;
BEGIN
    ...
    IF item(3).duration.minutes > 30 THEN ... -- call function
END;

```

また、次の例に示すように、フィールドに格納されているオブジェクトの属性を参照するには、拡張したドット表記法を使います。

```

DECLARE
    TYPE FlightRec IS RECORD (
        flight_no    INTEGER,
        plane_id     VARCHAR2(10),
        captain      Employee, -- declare object
        passengers   PassengerList, -- declare varray
        depart_time  TimeRec, -- declare nested record
        airport_code VARCHAR2(10));

```

```
        flight FlightRec;
BEGIN
    ...
    IF flight.captain.name = 'H Rawlins' THEN ...
END;
```

レコードの代入と比較

式の値をレコード内の特定のフィールドに代入するには、次の構文を使います。

```
record_name.field_name := expression;
```

次の例では、従業員の名前を大文字に変換します。

```
emp_info.ename := UPPER(emp_info.ename);
```

レコード中の個々のフィールドに別々に値を代入するかわりに、すべてのフィールドに値を一度に代入できます。これには2つの方法があります。第1の方法として、2つのユーザー定義レコードのデータ型が同じであれば、一方のレコードをもう一方のレコードに代入できます。正確に一致するフィールドが含まれているだけでは不十分です。次の例を考えてみます。

```
DECLARE
    TYPE DeptRec IS RECORD (
        dept_num  NUMBER(2),
        dept_name VARCHAR2(14),
        location  VARCHAR2(13));
    TYPE DeptItem IS RECORD (
        dept_num  NUMBER(2),
        dept_name VARCHAR2(14),
        location  VARCHAR2(13));
    dept1_info DeptRec;
    dept2_info DeptItem;
BEGIN
    ...
    dept1_info := dept2_info; -- illegal; different datatypes
END;
```

次の例に示すように、フィールドの数と順序が同じで、対応するフィールドのデータ型に互換性があれば、%ROWTYPE レコードをユーザー定義のレコードに代入できます。

```
DECLARE
    TYPE DeptRec IS RECORD (
        dept_num  NUMBER(2),
        dept_name CHAR(14),
        location  CHAR(13));
    dept1_info DeptRec;
    dept2_info dept%ROWTYPE;
BEGIN
```



```
SELECT * INTO dept2_info FROM dept WHERE deptno = 10;
dept1_info := dept2_info;
```

第2の方法として、次の例のように SELECT 文または FETCH 文を使って列の値を取り出し、レコードに代入できます。選択リストの列が、レコード中のフィールドと同じ順序で並ぶようにしてください。

```
DECLARE
  TYPE DeptRec IS RECORD (
    dept_num  NUMBER(2),
    dept_name CHAR(14),
    location  CHAR(13));
  dept_info DeptRec;
BEGIN
  SELECT deptno, dname, loc INTO dept_info FROM dept
    WHERE deptno = 20;
  ...
END;
```

しかし、INSERT 文を使ってユーザー定義のレコードをデータベース表に挿入できません。そのため、次の文は誤りです。

```
INSERT INTO dept VALUES (dept_info); -- illegal
```

また、代入文を使ってレコードに値のリストを代入できません。このため、次の構文は誤りです。

```
record_name := (value1, value2, value3, ...); -- illegal
```

次の例に示すように、ネストされたレコードは、データ型が同じであれば互いに代入できます。このような代入は、親レコードのデータ型が違っている場合でもできます。

```
DECLARE
  TYPE TimeRec IS RECORD (mins SMALLINT, hrs SMALLINT);
  TYPE MeetingRec IS RECORD (
    day      DATE,
    time_of TimeRec, -- nested record
    room_no  INTEGER(4));
  TYPE PartyRec IS RECORD (
    day      DATE,
    time_of TimeRec, -- nested record
    place    VARCHAR2(25));
  seminar MeetingRec;
  party    PartyRec;
BEGIN
  ...
  party.time_of := seminar.time_of;
END;
```

レコードの比較

レコードが NULL であるかどうか、等しいかどうかはテストできません。たとえば、次の IF 条件は誤りです。

```
BEGIN
    ...
    IF emp_info IS NULL THEN ... -- illegal
    IF dept2_info > dept1_info THEN ... -- illegal
END;
```

レコードの操作

データ型 RECORD を使用すると、何かの属性に関する情報を収集することができます。集めた情報は、まとまった 1 つの集合として参照できるので取扱いが簡単です。次の例では、assets および liabilities というデータベース表から会計情報を集め、比率分析を利用して 2 つの子会社の業績を比較しています。

```
DECLARE
    TYPE FiguresRec IS RECORD (cash REAL, notes REAL, ...);
    sub1_figs FiguresRec;
    sub2_figs FiguresRec;
    FUNCTION acid_test (figs FiguresRec) RETURN REAL IS ...
BEGIN
    SELECT cash, notes, ... INTO sub1_figs FROM assets, liabilities
        WHERE assets.sub = 1 AND liabilities.sub = 1;
    SELECT cash, notes, ... INTO sub2_figs FROM assets, liabilities
        WHERE assets.sub = 2 AND liabilities.sub = 2;
    IF acid_test(sub1_figs) > acid_test(sub2_figs) THEN ...
    ...
END;
```

集めた数値を財務比率を計算する acid_test ファンクションに渡す処理が、きわめて簡単に実行できている点に注意してください。

SQL*Plus で、次のようにオブジェクト型 Passenger を定義したとします。

```
SQL> CREATE TYPE Passenger AS OBJECT(
2   flight_no NUMBER(3),
3   name      VARCHAR2(20),
4   seat      CHAR(5));
```

次に、Passenger オブジェクトを格納する VARRAY 型 PassengertList を定義します。

```
SQL> CREATE TYPE PassengerList AS VARRAY(300) OF Passenger;
```

最後に、型 PassengerList の列を含む関係表 flights を次のようにして作ります。

```
SQL> CREATE TABLE flights (
```

```

2 flight_no  NUMBER(3),
3 gate       CHAR(5),
4 departure  CHAR(15),
5 arrival    CHAR(15),
6 passengers PassengerList);

```

列 `passengers` の中の各項目は、指定されたフライト (flight) の乗客リスト (PassengerList) を格納する varray です。これで、次のようにして、データベース表 `flights` にデータを入れることができます。

```

BEGIN
  INSERT INTO flights
    VALUES(109, '80', 'DFW 6:35PM', 'HOU 7:40PM',
      PassengerList (Passenger(109, 'Paula Trusdale', '13C'),
        Passenger(109, 'Louis Jemenez', '22F'),
        Passenger(109, 'Joseph Braun', '11B'), ...));
  INSERT INTO flights
    VALUES(114, '12B', 'SFO 9:45AM', 'LAX 12:10PM',
      PassengerList (Passenger(114, 'Earl Benton', '23A'),
        Passenger(114, 'Alma Breckenridge', '10E'),
        Passenger(114, 'Mary Rizutto', '11C'), ...));
  INSERT INTO flights
    VALUES(27, '34', 'JFK 7:05AM', 'MIA 9:55AM',
      PassengerList (Passenger(27, 'Raymond Kiley', '34D'),
        Passenger(27, 'Beth Steinberg', '3A'),
        Passenger(27, 'Jean Lafevre', '19C'), ...));
END;

```

次の例では、データベース表 `flights` から行を取り出して、レコード `flight_info` に入れます。このように、飛行 (flight) 予定に関するすべての情報を、乗客リスト (passenger list) も含めて 1 つの論理単位として扱うことができます。

```

DECLARE
  TYPE FlightRec IS RECORD (
    flight_no  NUMBER(3),
    gate       CHAR(5),
    departure  CHAR(15),
    arrival    CHAR(15),
    passengers PassengerList);
  flight_info FlightRec;
  CURSOR c1 IS SELECT * FROM flights;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO flight_info;
    EXIT WHEN c1%NOTFOUND;
    FOR i IN 1..flight_info.passengers.LAST LOOP
      IF flight_info.passengers(i).seat = 'NA' THEN

```

```
        DBMS_OUTPUT.PUT_LINE(flight_info.passengers(i).name);
        RAISE seat_not_available;
    END IF;
    ...
END LOOP;
END LOOP;
CLOSE c1;
EXCEPTION
    WHEN seat_not_available THEN
        ...
END;
```

5

Oracle の操作

Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information upon it.—Samuel Johnson

この章では、Oracle の性能を引き出す方法を説明します。Oracle データを操作する SQL コマンド、ファンクション、演算子が PL/SQL でどのようにサポートされているのかを示します。また、カーソルの管理方法およびカーソル変数の使用方法、トランザクションの処理方法についても説明します。

主なトピック

SQL のサポート

カーソル管理

パッケージ・カーソル

カーソル FOR ループの使用

カーソル変数の使用

カーソル属性の使用

トランザクション処理

自律型トランザクションの使用

パフォーマンスの向上

下位互換性の保証

SQL のサポート

SQL を拡張した PL/SQL は、優れた性能に加えて、使いやすさも実現しています。PL/SQL は、EXPLAIN PLAN 以外のすべての SQL データ操作文、トランザクション制御文、ファンクション、疑似列、演算子を完全にサポートしているので、Oracle データを柔軟かつ安全に操作できます。PL/SQL は動的 SQL もサポートしているので、SQL データ定義文、データ制御文、セッション制御文を動的に実行できます（詳細は第 10 章の「システム固有の動的 SQL」を参照）。さらに、PL/SQL は、現行の ANSI/ISO SQL 標準に準拠しています。

データ操作

Oracle データを操作するには、INSERT コマンド、UPDATE コマンド、DELETE コマンド、SELECT コマンド、および LOCK TABLE コマンドを使用します。INSERT で、データベースの表に新たなデータ行を追加します。UPDATE で、行を変更します。DELETE で、不要な行を削除します。SELECT で、検索基準に合う行を取り出します。LOCK TABLE で、表へのアクセスを一時的に制限します。

トランザクション制御

Oracle では、トランザクションに基づいて作業します。つまり、トランザクションを使ってデータの整合性を確保します。トランザクションとは、論理作業単位を実行する一連の SQL のデータ操作文です。たとえば、2 つの UPDATE 文を使って、ある銀行口座に入金し、別の口座から出金します。

また、Oracle では、トランザクションがデータベースに加えた変更内容の確定または取消しができます。トランザクション中にプログラムに障害が発生すると、Oracle がエラーを検出して、トランザクションをロールバックします。この結果、データベースは自動的に元の状態に復元されます。

COMMIT コマンド、ROLLBACK コマンド、SAVEPOINT コマンド、および SET TRANSACTION コマンドを使用して、トランザクションを制御します。COMMIT は、現在のトランザクション中にデータベースに加えられた変更内容を確定します。ROLLBACK は、現在のトランザクションを終了させ、トランザクションの開始以降に加えられた変更をすべて取り消します。SAVEPOINT は、トランザクション処理内の現在位置にマークを付けます。ROLLBACK と SAVEPOINT を併用すると、トランザクションの一部だけを取り消すことができます。SET TRANSACTION は、読み書きアクセスや分離レベルなど、トランザクションのプロパティを設定します。

SQL ファンクション

PL/SQL では、SQL ファンクションをすべて使えます。その中には、Oracle データの列全体をサマリーする集計関数 AVG、COUNT、GROUPING、MAX、MIN、STDDEV、SUM、VARIANCE が含まれます。COUNT(*) を例外として、すべての集計関数は NULL を無視します。

集計関数は、SQL 文では使用できますが、プロシージャ文では使えません。集計関数では、戻された行を SELECT GROUP BY 文でソートしてサブグループに分けない限り、列全体が対象になります。GROUP BY 句を省略すると、戻されたすべての行が 1 つのグループとみなされます。

集計関数のコールには、次の構文を使います。

```
function_name([ALL | DISTINCT] expression)
```

expression は、1 つまたは複数のデータベースの列を参照します。ALL (デフォルト) を指定すると、重複するものも含めて、すべての列値が対象となります。DISTINCT を指定すると、集計関数は重複しない値だけを取り扱います。たとえば、次の文はデータベースの表 emp の中にある異なる肩書の数に戻します。

```
SELECT COUNT(DISTINCT job) INTO job_count FROM emp;
```

COUNT ファンクションでは、表の行数を戻すアスタリスク (*) オプションが指定できます。たとえば、次の文は表 emp の中にある行の数を戻します。

```
SELECT COUNT(*) INTO emp_count FROM emp;
```

SQL 疑似列

PL/SQL は SQL 疑似列 CURRVAL、LEVEL、NEXTVAL、ROWID、ROWNUM を認識します。これらの疑似列は、それぞれ特定のデータ項目を戻します。疑似列は、表の中の実際の列ではありませんが、列と同様に扱うことができます。たとえば、疑似列から値を取り出すことができます。しかし、疑似列に値を挿入したり、疑似列の値を更新または削除したりできません。また、疑似列は SQL 文では使用できますが、プロシージャ文では使えません。

CURRVAL と NEXTVAL

順序とは、連続的な数値を生成するスキーマ・オブジェクトのことです。順序を作る場合は、その初期値と増分を指定できます。CURRVAL は指定された順序の中での現在の値を戻します。

セッションの中で CURRVAL を参照する前に、NEXTVAL を使って数値を生成する必要があります。NEXTVAL を参照すると、現在の順序番号が CURRVAL に格納されます。NEXTVAL は順序に増分を加えて、次の値を戻します。順序の現在の値または次の値を得るには、次のようにドット表記法を使います。

```
sequence_name.CURRVAL  
sequence_name.NEXTVAL
```

順序を作ると、トランザクション処理の目的のために独自の順序番号を生成させることができます。ただし、CURRVAL および NEXTVAL は、選択リストおよび VALUES 句、SET 句の中でしか使えません。次の例では、順序を使って同じ従業員番号を 2 つの表に挿入しています。

```
INSERT INTO emp VALUES (empno_seq.NEXTVAL, my_ename, ...);
INSERT INTO sals VALUES (empno_seq.CURRVAL, my_sal, ...);
```

トランザクションが順序番号を生成する場合、トランザクションのコミットやロールバックがなされるかどうかにかかわらず、順序にはただちに増分が加えられます。

LEVEL

SELECT CONNECT BY 文で LEVEL を使うと、データベース表の行をツリー構造に整理できます。LEVEL はツリー構造の中のノードのレベル番号を戻します。ルートはレベル 1、ルートの子はレベル 2、孫はレベル 3、のように続きます。

PRIOR 演算子を使って、問合せがツリーの中をたどるときの向き（ルートから下へ、または枝から上へ）を指定します。ツリーのルートを識別する条件は、START WITH 句で指定します。

ROWID

ROWID はデータベース表の行の ROWID（バイナリ・アドレス）を戻します。UROWID 型の変数を使うと、ROWID を読取り可能な書式で格納できます。次の例では、この目的で row_id という変数を宣言しています。

```
DECLARE
    row_id UROWID;
```

ROWID を選択または取り出して UROWID 変数に入れる場合は、バイナリ値を 18 バイトの文字列に変換する関数 ROWIDTOCHAR を使います。すると、UPDATE 文または DELETE 文の WHERE 句の中で UROWID 変数と ROWID 疑似列を比較して、カーソルから取り出された最新の行を識別できます。例は、5-44 ページの「[コミットにまたがる取出し](#)」を参照してください。

ROWNUM

ROWNUM は、行が表から取り出された順番を示す番号を戻します。最初に取り出された行の ROWNUM は 1、2 番目に取り出された行の ROWNUM は 2、のように続きます。SELECT 文に ORDER BY 句が含まれている場合は、取り出された行がソートされる前に ROWNUM が割り当てられます。

UPDATE 文で ROWNUM を使って、表の中の個々の行に一意的な値を割り当てることができます。また、SELECT 文の WHERE 句で ROWNUM を使って、取り出される行の数を制限することもできます。次の例を参照してください。


```
DECLARE
  CURSOR c1 IS SELECT empno, sal FROM emp
    WHERE sal > 2000 AND ROWNUM < 10; -- returns 10 rows
```

ROWNUM の値が増えるのは、行が取り出されたときだけです。つまり、WHERE 句で ROWNUM を使う場合は、次のようにしなければなりません。

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

SQL 演算子

PL/SQL では、SQL 文の中で、SQL の比較演算子、集合演算子および行演算子を使えます。このセクションでは、これらの演算子について簡単に説明します。詳細は、『Oracle8i SQL リファレンス』を参照してください。

比較演算子

比較演算子は、通常、データ操作文の WHERE 句で述語を形成するために使います。述語は、2 つの式を比較して、必ず TRUE、FALSE、NULL のいずれかに評価します。下記の比較演算子はすべて、述語の形成に使用できます。さらに、述語は論理演算子 AND、OR および NOT を使用して結合できます。

演算子	説明
ALL	値をリストのすべての値、または副問合せが戻したすべての値と 1 つずつ比較して、結果がすべて TRUE であれば TRUE に評価する。
ANY、SOME	値をリストのすべての値、または副問合せが戻したすべての値と 1 つずつ比較して、結果のいずれかが TRUE であれば TRUE に評価する。
BETWEEN	値が指定範囲内かどうかをテストする。
EXISTS	副問合せが行を 1 つでも戻すと TRUE を戻す。
IN	集合のメンバーシップをテストする。
IS NULL	NULL かどうかをテストする。
LIKE	文字列が、指定したパターンと一致するかどうかをテストする。指定パターンにはワイルドカードを使用できる。

集合演算子

集合演算子は 2 つの問合せの結果を組み合わせる 1 つの結果を戻します。INTERSECT によって、2 つの問合せの両方で選択されたすべての行を、重複するものを除いて戻します。MINUS は、1 番目の問合せによって選択されたが、2 番目の問合せでは選択されなかったすべての行を、重複するものを除いて戻します。UNION によって、2 つの問合せのいずれかで

選択されたすべての行を、重複するものを除いて戻します。UNION ALL は、重複した行も含めて、どちらかの問合せによって選択されたすべての行を戻します。

行演算子

行演算子は特定の行を戻すか、または参照します。ALL によって、問合せの結果や集合式に含まれる重複した行をそのまま残します。DISTINCT は、問合せの結果や集合式に含まれる重複した行を削除します。PRIOR は、ツリー構造の問合せによって戻された現在行の親の行を参照します。

カーソル管理

PL/SQL は、暗黙カーソルと明示カーソルの、2 つの型のカーソルを使用します。PL/SQL では、1 つの行だけを戻す問合せを含むすべての SQL データ操作文で、暗黙的にカーソルを宣言します。しかし、複数の行を戻す問合せの場合は、カーソル FOR ループを使うか、カーソルを明示的に宣言する必要があります。

明示カーソル

問合せが戻す行の集合は、検索条件に合致する行がどれだけあったかに応じて、ゼロまたは 1 つ、複数の行で構成されます。問合せが複数の行を戻す場合は、行を処理するために明示的にカーソルを宣言できます。さらに、カーソルの宣言は、PL/SQL のブロックまたはサブプログラム、パッケージの宣言部の中でできます。

OPEN、FETCH、CLOSE という 3 つのコマンドを使ってカーソルを制御できます。まず、結果セットを識別する OPEN 文でカーソルを初期化します。次に、FETCH 文を使って最初の行を取り出します。すべての行が取り出されるまで、FETCH 文を繰り返して実行できます。最後の行の処理が終わったら、CLOSE 文でカーソルを解放します。複数のカーソルを宣言し、オープンすると、複数の問合せを並行して処理できます。

カーソルの宣言

PL/SQL では前方参照ができません。このため、他の文でカーソルを参照するときは、事前に宣言しておく必要があります。カーソルを宣言する場合は、次の構文を使ってカーソルに名前を与え、特定の問合せと結び付けます。

```
CURSOR cursor_name [(parameter[, parameter]...)]  
    [RETURN return_type] IS select_statement;
```

ここで、return_type はデータベースの表の中のレコードまたは行を表していなければなりません。また、parameter は次の構文を表します。

```
cursor_parameter_name [IN] datatype [{:= | DEFAULT} expression]
```

たとえば次の例では、c1 および c2 という名前のカーソルを宣言しています。

```
DECLARE
```

```
CURSOR c1 IS SELECT empno, ename, job, sal FROM emp
WHERE sal > 2000;
CURSOR c2 RETURN dept%ROWTYPE IS
SELECT * FROM dept WHERE deptno = 10;
```

カーソル名は未宣言の識別子であり、PL/SQL 変数の名前ではありません。カーソル名に値を代入したり、カーソル名を式の中で使ったりはできません。ただし、カーソルの有効範囲の規則は変数の有効範囲の規則と同じです。データベース表に基づいた名前をカーソルにつけることは、可能ですが、お薦めしません。

カーソルはパラメータを取ることができます。カーソルのパラメータは、カーソルに結び付けられた問合せの中で、定数が使える位置であればどこでも使えます。カーソルの仮パラメータは IN パラメータでなければなりません。このため、実パラメータに値を戻すことはできません。また、カーソル・パラメータに NOT NULL 制約を課することはできません。

次の例に示すように、カーソルのパラメータをデフォルト値に初期化できます。初期化しておけば、必要に応じてデフォルト値を受け入れたり上書きしたりすることで、カーソルの実パラメータにさまざまな数値を渡すことができます。また、カーソルへの参照を個々に変更しなくても、仮パラメータを新しく追加できます。

```
DECLARE
CURSOR c1 (low INTEGER DEFAULT 0,
           high INTEGER DEFAULT 99) IS SELECT ...
```

カーソルのパラメータの有効範囲は、カーソルに対してローカルです。つまり、カーソル宣言で指定されている問合せの範囲からしか参照できません。カーソルのパラメータ値は、カーソルがオープンされているときに、カーソルに結び付けられた問合せから使えます。

カーソルのオープン

カーソルをオープンすると、問合せが実行され、結果セットが識別されます（結果セットは、問合せの検索条件に合致するすべての行で構成されています）。FOR UPDATE 句を使って宣言されたカーソルの場合、OPEN 文はこれらの行のロックもします。OPEN 文の例を次に示します。

```
DECLARE
CURSOR c1 IS SELECT ename, job FROM emp WHERE sal < 3000;
...
BEGIN
OPEN c1;
...
END;
```

結果セットの中の行は、OPEN 文の実行時には取り出されません。行の取出しには FETCH 文を使います。

カーソル・パラメータ渡し

OPEN 文を使ってパラメータをカーソルに渡します。デフォルト値を受け入れるのであれば、カーソル宣言の中の仮パラメータは、すべて OPEN 文の中で対応する実パラメータを持たなければなりません。たとえば、次のようなカーソル宣言があると、

```
DECLARE
    emp_name emp.ename%TYPE;
    salary    emp.sal%TYPE;
    CURSOR c1 (name VARCHAR2, salary NUMBER) IS SELECT ...
```

次の文はいずれもカーソルをオープンします。

```
OPEN c1(emp_name, 3000);
OPEN c1('ATTLEY', 1500);
OPEN c1(emp_name, salary);
```

最後の例で、カーソル宣言の中で識別子 salary を使うと、この識別子は仮パラメータを参照します。しかし、OPEN 文の中で使うと、PL/SQL 変数を参照します。混乱を避けるため、他と重複しない識別子を使ってください。

デフォルト値で宣言された仮パラメータの詳細は、対応する実パラメータがなくてもかまいません。このような仮パラメータは、OPEN 文の実行時にデフォルト値を取ります。

位置表記法または名前表記法を使って、OPEN 文の実パラメータを、カーソル宣言の仮パラメータに結び付けることができます。(7-13 ページの「[位置表記法と名前表記法](#)」を参照。) 個々の実パラメータは、対応する仮パラメータのデータ型と互換性のあるデータ型に属していなければなりません。

カーソルを使った取出し

FETCH 文は、結果セットの行を 1 行ずつ取り出します。行を取り出すたびに、カーソルは結果セットの中の次の行に進みます。たとえば、

```
FETCH c1 INTO my_empno, my_ename, my_deptno;
```

カーソルと結び付けられた問合せが戻す列の値に対しては、INTO リストの中に、対応する、型互換の変数が存在しなければなりません。通常は、次のように FETCH 文を使用します。

```
LOOP
    FETCH c1 INTO my_record;
    EXIT WHEN c1%NOTFOUND;
    -- process data record
END LOOP;
```

問合せは、有効範囲内にある PL/SQL 変数を参照できます。ただし、問合せの中の変数はカーソルがオープンされたときにだけ評価されます。次の例では、FETCH 文が実行されるたびに factor に増分が加えられていますが、取り出された給与はそれぞれ 2 倍されます。

```
DECLARE
```

```

my_sal emp.sal%TYPE;
my_job emp.job%TYPE;
factor INTEGER := 2;
CURSOR c1 IS SELECT factor*sal FROM emp WHERE job = my_job;
BEGIN
...
OPEN c1; -- here factor equals 2
LOOP
    FETCH c1 INTO my_sal;
    EXIT WHEN c1%NOTFOUND;
    factor := factor + 1; -- does not affect FETCH
END LOOP;
END;
```

結果セットや問合せの中の変数の値を変更する場合は、カーソルをクローズし、入力変数を新しい値に設定して、もう一度オープンしなければなりません。

ただし、同じカーソルを使って、別々の FETCH 文で、異なる INTO リストを使えます。個々の FETCH 文で別の行を取り出し、ターゲット変数に値を代入します。次に例を示します。

```

DECLARE
CURSOR c1 IS SELECT ename FROM emp;
name1 emp.ename%TYPE;
name2 emp.ename%TYPE;
name3 emp.ename%TYPE;
BEGIN
OPEN c1;
FETCH c1 INTO name1; -- this fetches first row
FETCH c1 INTO name2; -- this fetches second row
FETCH c1 INTO name3; -- this fetches third row
...
CLOSE c1;
END;
```

FETCH 文を実行した時点で結果セットに行が残っていなかった場合、FETCH 文のリストの変数の値は不定になります。

注意：結果として FETCH 文は行を戻すことに失敗しますが、この状況が発生した場合に例外は呼び出されません。失敗を検出するには、カーソル属性 %FOUND または %NOTFOUND を使わなければなりません。詳細は、5-31 ページの「[カーソル属性の使用](#)」を参照してください。

カーソルのクローズ

CLOSE 文によってカーソルは使用禁止になり、結果セットは未定義になります。CLOSE 文の例を次に示します。

```
CLOSE c1;
```

クローズされたカーソルは、再度オープンできます。クローズされたカーソルに対してこれ以外の操作を実行すると、事前定義の例外 `INVALID_CURSOR` が呼び出されます。

副問合せの使用

副問合せは、別の SQL データ操作文内に出現する問合せであり、多くの場合カッコで囲まれています。副問合せを評価すると、文に対して 1 つの値または複数の値の集合が与えられます。通常、副問合せは `WHERE` 句の中で最もよく使われます。たとえば、次の問合せは、シカゴ在住ではない従業員を戻します。

```
DECLARE
  CURSOR c1 IS SELECT empno, ename FROM emp
    WHERE deptno IN (SELECT deptno FROM dept
      WHERE loc <> 'CHICAGO');
```

`FROM` 句の中で副問合せを使った次の問合せは、従業員 5 人以上の部門の番号と名称を戻します。

```
DECLARE
  CURSOR c1 IS SELECT t1.deptno, dname, "STAFF"
    FROM dept t1, (SELECT deptno, COUNT(*) "STAFF"
      FROM emp GROUP BY deptno) t2
    WHERE t1.deptno = t2.deptno AND "STAFF" >= 5;
```

副問合せが各表につき 1 回しか評価されないのに対し、関連副問合せは各行につき 1 回評価されます。次に示す問合せは、給与が部門平均を上回っている従業員の名前と給与を戻します。関連副問合せでは、`emp` 表の各行について、その行の部門の平均給与を計算します。行の給与が平均を上回っている場合、その行は戻されます。

```
DECLARE
  CURSOR c1 IS SELECT deptno, ename, sal FROM emp t
    WHERE sal > (SELECT AVG(sal) FROM emp WHERE t.deptno = deptno)
    ORDER BY deptno;
```

暗黙カーソル

明示的に宣言されたカーソルと結び付けられていない SQL 文を処理するために、Oracle は暗黙的にカーソルをオープンします。PL/SQL では直前の暗黙カーソルを `SQL` カーソルとして参照できます。

SQL カーソルの制御には、`OPEN` 文、`FETCH` 文、`CLOSE` 文は使えません。しかし、カーソルの属性を使って、直前に実行された SQL 文に関する情報を読み込むことはできます。5-31 ページの「[カーソル属性の使用](#)」を参照してください。

パッケージ・カーソル

パッケージの中で、カーソルの仕様部を本体と切り離して別の位置に置くことができます。こうしておくと、カーソルの仕様部を変更せずに、本体だけを変更できます。パッケージの仕様部の中でカーソルの仕様部をコーディングする場合は、次の構文を使います。

```
CURSOR cursor_name [(parameter[, parameter]...)] RETURN return_type;
```

次に示す例では、%ROWTYPE 属性を使って、データベース表 emp の中の行を表すレコード型を指定しています。

```
CREATE PACKAGE emp_actions AS
    /* Declare cursor spec. */
    CURSOR c1 RETURN emp%ROWTYPE;
    ...
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
    /* Define cursor body. */
    CURSOR c1 RETURN emp%ROWTYPE IS
        SELECT * FROM emp
            WHERE sal > 3000;
    ...
END emp_actions;
```

結果の値のデータ型を RETURN 句で定義しているので、カーソル仕様部には SELECT 文がありません。しかしカーソル本体には、SELECT 文と、カーソル仕様部と同じ RETURN 句が必要になります。また、SELECT リスト中の項目の数とデータ型は、RETURN 句と一致していなければなりません。

パッケージ・カーソルを使うと柔軟性が向上します。たとえば、次のようにすると、上記の例でカーソル仕様部を変更せずにカーソル本体を変更できます。

```
CREATE PACKAGE BODY emp_actions AS
    /* Define cursor body. */
    CURSOR c1 RETURN emp%ROWTYPE IS
        SELECT * FROM emp
            WHERE deptno = 20; -- new WHERE clause
    ...
END emp_actions;
```

カーソル FOR ループの使用

明示カーソルが必要になる状況では、ほとんどの場合、OPEN 文や FETCH 文や CLOSE 文ではなくカーソル FOR ループを使って、コードを単純化できます。カーソル FOR ループは、ループ索引を %ROWTYPE 属性のレコードとして暗黙的に宣言し、カーソルをオープンして、結果セットから行の値を取り出してレコード中のフィールドに入れるという作業を繰り返し、すべての行を処理したらカーソルをクローズします。

次に示す PL/SQL ブロックでは、実験データを使って結果を計算し、その結果を一時表に格納しています。FOR ループの索引 c1_rec は、レコードとして暗黙的に宣言されています。そのフィールドには、カーソル c1 から取り出されたすべての列の値を格納します。個々のフィールドはドット表記法を使って参照します。

```
-- available online in file 'examp7'
DECLARE
    result temp.col1%TYPE;
    CURSOR c1 IS
        SELECT n1, n2, n3 FROM data_table WHERE exper_num = 1;
BEGIN
    FOR c1_rec IN c1 LOOP
        /* calculate and store the results */
        result := c1_rec.n2 / (c1_rec.n1 + c1_rec.n3);
        INSERT INTO temp VALUES (result, NULL, NULL);
    END LOOP;
    COMMIT;
END;
```

カーソル FOR ループを入力する場合、カーソル名は、OPEN 文または 1 つ外側のカーソル FOR ループによってすでにオープンされているカーソルに属してはなりません。FOR ループを繰り返す前に、PL/SQL は値を取り出して暗黙的に宣言されたレコードに入れます。このレコードは、次のように明示的に宣言されたレコードと等価です。

```
c1_rec c1%ROWTYPE;
```

レコードはループの内側だけで定義されています。ループの外側からこのレコードのフィールドを参照できません。たとえば、次の参照は誤りです。

```
FOR c1_rec IN c1 LOOP
    ...
END LOOP;
result := c1_rec.n2 + 3; -- illegal
```

ループの中の一連の文は、カーソルと結び付けられた問合せを満たす行 1 つについて 1 回実行されます。ループを終了させると、カーソルは自動的にクローズされます。これは、EXIT 文または GOTO 文を使ってループを途中で終了させた場合やループの内側から例外が呼び出された場合にも当てはまります。

副問合せの使用

PL/SQL では副問合せが置き換えられるため、カーソルを宣言する必要はありません。次のカーソル FOR ループでは、ボーナスを計算し、その結果をデータベース表に挿入しています。

```
DECLARE
    bonus REAL;
BEGIN
    FOR emp_rec IN (SELECT empno, sal, comm FROM emp) LOOP
        bonus := (emp_rec.sal * 0.05) + (emp_rec.comm * 0.25);
        INSERT INTO bonuses VALUES (emp_rec.empno, bonus);
    END LOOP;
    COMMIT;
END;
```

別名の使用

暗黙的に宣言されたレコードのフィールドは、直前に取り出された行の列の値を保持しています。フィールドは、選択リスト中の対応する列と同じ名前を持ちます。しかし、選択項目が式の場合はどうなるのでしょうか。次の例を考えてみます。

```
CURSOR c1 IS
    SELECT empno, sal+NVL(comm,0), job FROM ...
```

このような場合は、選択項目の別名を指定する必要があります。次の例では、選択項目 `sal+NVL(comm,0)` の別名として `wages` を指定しています。

```
CURSOR c1 IS
    SELECT empno, sal+NVL(comm,0) wages, job FROM ...
```

対応するフィールドを参照する場合は、次のように、列名ではなく別名を使います。

```
IF emp_rec.wages < 1000 THEN ...
```

パラメータ渡し

カーソル FOR ループ内のカーソルには、パラメータを渡すことができます。次の例では、部門番号を渡しています。次に、その部門の従業員に支払われている賃金の合計を計算します。また、2000 ドルよりも高い給与または給与より高いコミッションを受け取っている従業員、あるいはその両方を満たす従業員の数も調べています。

```
-- available online in file 'examp8'
DECLARE
    CURSOR emp_cursor(dnum NUMBER) IS
        SELECT sal, comm FROM emp WHERE deptno = dnum;
    total_wages NUMBER(11,2) := 0;
    high_paid    NUMBER(4) := 0;
```

```
higher_comm NUMBER(4) := 0;
BEGIN
  /* The number of iterations will equal the number of rows
     returned by emp_cursor. */
  FOR emp_record IN emp_cursor(20) LOOP
    emp_record.comm := NVL(emp_record.comm, 0);
    total_wages := total_wages + emp_record.sal +
      emp_record.comm;
    IF emp_record.sal > 2000.00 THEN
      high_paid := high_paid + 1;
    END IF;
    IF emp_record.comm > emp_record.sal THEN
      higher_comm := higher_comm + 1;
    END IF;
  END LOOP;
  INSERT INTO temp VALUES (high_paid, higher_comm,
    'Total Wages: ' || TO_CHAR(total_wages));
  COMMIT;
END;
```

カーソル変数の使用

カーソルと同じように、カーソル変数は複数行の問合せの結果セットの中の現在行を指します。ただし、定数が変数と異なるように、カーソルとカーソル変数も異なります。カーソルが静的であるのに対し、カーソル変数は動的であり、特定の問合せに結び付けられていません。カーソル変数は、型互換性のある任意の問合せに対してオープンできます。そのため、柔軟性が向上します。

また、新しい値をカーソル変数に代入し、それをパラメータとしてローカル・サブプログラムおよびストアド・サブプログラムに渡すこともできます。このことによって、データ検索を集中化しやすくなります。

カーソル変数は、すべての PL/SQL クライアントで使えます。たとえば、OCI や Pro*C プログラムなどの PL/SQL ホスト環境の中でカーソル変数を宣言し、それを入力ホスト変数（バインド変数）として PL/SQL に渡すことができます。さらに、PL/SQL エンジンを備えた Oracle Forms や Oracle Reports などのアプリケーション開発ツールでは、クライアント側でカーソル変数を完全に使えます。

Oracle Server も PL/SQL エンジンを備えています。したがって、アプリケーションとサーバーの間で、リモート・プロシージャ・コール（RPC）を介してカーソル変数をやりとりできます。

カーソル変数とは？

カーソル変数は、C や Pascal のポインタに類似しており、項目そのもののかわりに項目のメモリー位置（アドレス）を保持します。したがって、カーソル変数を宣言すると、項目では

なくポインタが作られます。PL/SQL では、ポインタはデータ型 REF X に属します。REF は REFERENCE の略であり、X はオブジェクトのクラスを表します。したがって、カーソル変数はデータ型 REF CURSOR に属します。

複数行の問合せを実行するために、Oracle は処理情報を格納する名前の付けられていない作業域をオープンします。その情報にアクセスするには、その作業域の名を示す明示カーソルを使います。または、作業域を指すカーソル変数を使います。カーソルが常に同じ問合せ作業域を参照するのに対し、カーソル変数は異なる作業域を参照できます。そのため、カーソルとカーソル変数には相互操作性がありません。つまり、一方の値が期待されている場所で、もう一方を使用することはできません。

変数を使う理由

カーソル変数は、主に、PL/SQL のストアド・サブプログラムと各種クライアントとの間で問合せ結果を渡すために使います。PL/SQL およびクライアントはどちらも結果セットを所有してはならず、単に、結果セットが格納されている作業域を指すポインタを共有しているだけです。たとえば、OCI クライアントおよび Oracle Forms アプリケーション、Oracle Server がすべて同じ作業域を参照する場合があります。

問合せ作業域は、それを指すカーソル変数が存在するかぎりアクセスできます。したがって、カーソル変数の値は、1 つの有効範囲から別の有効範囲へ自由に渡すことができます。たとえば、Pro*C プログラムに組み込まれた PL/SQL ブロックにホスト・カーソル変数を渡す場合、カーソル変数が指す作業域は、そのブロックの終了後もアクセス可能な状態のままです。

クライアント側に PL/SQL エンジンがあれば、クライアントからサーバーへのコールに課される制限はありません。たとえば、クライアント側でカーソル変数を宣言し、それをサーバー側でオープンして取り出した後で、クライアント側で引き続き取り出すことができます。また、PL/SQL ブロックを使って複数のホスト・カーソル変数を 1 回の往復でオープンまたはクローズしておくことで、ネットワーク通信量を削減できます。

REF CURSOR 型の定義

カーソル変数を作る場合は、2 つのステップを実行します。まず、REF CURSOR 型を定義し、次にその型のカーソル変数を宣言します。REF CURSOR 型は、任意の PL/SQL ブロックまたはサブプログラム、パッケージの中で、次の構文を使って定義できます。

```
TYPE ref_type_name IS REF CURSOR RETURN return_type;
```

ref_type_name は、それ以降のカーソル変数の宣言の中で使う型指定子です。return_type は、データベース表の中のレコードまたは行を表していなければなりません。次の例では、データベース表 dept の中の行を表す戻り型を指定しています。

```
DECLARE
    TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
```

REF CURSOR 型は、強い（制限的な）ものと弱い（制限的でない）ものがあります。次の例に示すように、強い REF CURSOR 型定義では戻り型を指定しますが、弱い定義では戻り型を指定しません。

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE; -- strong
    TYPE GenericCurTyp IS REF CURSOR; -- weak
```

強い REF CURSOR 型の方が、エラー発生の可能性は少なくなります。その理由は、PL/SQL コンパイラの場合、強い型定義のカーソル変数は型互換性のある問合せにしか結び付けることができないからです。弱い REF CURSOR 型は、より柔軟です。弱い型定義のカーソル変数は、どの問合せにも結び付けることができます。

カーソル変数の宣言

REF CURSOR 型を宣言すると、PL/SQL ブロックまたはサブプログラムで、その型のカーソル変数を宣言できます。次の例では、カーソル変数 dept_cv を宣言しています。

```
DECLARE
    TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
    dept_cv DeptCurTyp; -- declare cursor variable
```

注意：パッケージの中ではカーソル変数を宣言できません。カーソル変数はパッケージ変数とは異なり、永続状態を持ちません。カーソル変数を宣言すると、項目ではなく ポインタが作られることを覚えておいてください。そのため、カーソル変数はデータベースに保存できません。

カーソル変数は、通常の有効範囲とインスタンス生成の規則に従います。ローカルな PL/SQL カーソルは、ブロックまたはサブプログラムに入ったときにインスタンスが作られ、ブロックまたはサブプログラムを出るときに消滅します。

次に示すように、REF CURSOR 型定義の RETURN 句では、%ROWTYPE を使って、強い型定義（弱い型定義ではなく）のカーソル変数によって戻される行を表すレコード型を指定できます。

```
DECLARE
    TYPE TmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
    tmp_cv TmpCurTyp; -- declare cursor variable
    TYPE EmpCurTyp IS REF CURSOR RETURN tmp_cv%ROWTYPE;
    emp_cv EmpCurTyp; -- declare cursor variable
```

同様に、%TYPE を使ってレコード変数のデータ型を指定できます。次に例を示します。

```
DECLARE
    dept_rec dept%ROWTYPE; -- declare record variable
    TYPE DeptCurTyp IS REF CURSOR RETURN dept_rec%TYPE;
    dept_cv DeptCurTyp; -- declare cursor variable
```

次の例では、RETURN 句の中でユーザー定義の RECORD 型を指定しています。

```
DECLARE
  TYPE EmpRecTyp IS RECORD (
    empno NUMBER(4),
    ename VARCHAR2(10),
    sal    NUMBER(7,2));
  TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
  emp_cv EmpCurTyp; -- declare cursor variable
```

パラメータとしてのカーソル変数

カーソル変数をファンクションおよびプロシージャの仮パラメータとして宣言できます。次の例では、REF CURSOR 型 EmpCurTyp を定義し、その型のカーソル変数をプロシージャの仮パラメータとして宣言しています。

```
DECLARE
  TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS ...
```

注意：すべてのポインタと同様、カーソル変数にはパラメータ・エイリアシングの可能性があります。詳細は、7-21 ページの「[パラメータのエイリアシング](#)」を参照してください。

カーソル変数の制御

カーソル変数を制御する場合は、OPEN-FOR、FETCH、CLOSE という 3 つの文を使います。まず、OPEN-FOR 文でカーソル変数を複数行問合せ用にオープンします。次に、FETCH 文で結果セットから一度に 1 行ずつ行を取り出します。すべての行が処理されたら、CLOSE 文でカーソル変数をクローズします。

カーソル変数のオープン

OPEN-FOR 文を使うと、カーソル変数を複数行の問合せに結び付けたり、問合せを実行したり、結果セットを識別したりできます。次に構文を示します。

```
OPEN {cursor_variable_name | :host_cursor_variable_name}
  FOR select_statement;
```

ここで、host_cursor_variable_name は、OCI や Pro*C プログラムなどの PL/SQL ホスト環境で宣言されたカーソル変数を識別します。

カーソルとは異なり、カーソル変数はパラメータをとりません。ただし、カーソル変数にはパラメータだけでなく問合せ全体を渡すことができるので、柔軟性があります。問合せでは、ホスト変数、PL/SQL 変数、パラメータ、ファンクションを参照できます。ただし、FOR UPDATE にはできません。

次に示す例では、カーソル変数 `emp_cv` をオープンしています。カーソルの属性（`%FOUND`、`%NOTFOUND`、`%ISOPEN`、`%ROWCOUNT`）をカーソル変数に適用できることに注意してください。

```
IF NOT emp_cv%ISOPEN THEN
    /* Open cursor variable. */
    OPEN emp_cv FOR SELECT * FROM emp;
END IF;
```

その他の `OPEN-FOR` 文は、異なる複数の問合せ用に同じカーソル変数をオープンできます。カーソル変数を再オープンする場合、その前にクローズする必要はありません。（静的カーソルを `OPEN` 文で連続してオープンすると、事前定義の例外 `CURSOR_ALREADY_OPEN` が呼び出されます。）別の問合せ用にカーソル変数を再オープンすると、前の問合せは失われます。

一般に、カーソル変数をオープンするときは、ストアード・プロシージャに渡し、そのストアード・プロシージャで仮パラメータの 1 つとして宣言します。たとえば、次のパッケージ・プロシージャは、カーソル変数 `emp_cv` をオープンします。

```
CREATE PACKAGE emp_data AS
    ...
    TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
    PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp);
END emp_data;

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

カーソル変数を、そのカーソル変数をオープンするサブプログラムの仮パラメータとして宣言する場合は、`IN OUT` モードを指定しなければなりません。そうすることで、サブプログラムはコール元にオープン・カーソルを渡すことができます。

あるいは、スタンドアロン・プロシージャを使ってカーソル変数をオープンする方法もあります。単に別々のパッケージの中で `REF CURSOR` 型を定義し、スタンドアロン・プロシージャの中でその型を参照します。たとえば、次のような本体部のないパッケージを作るなら、スタンドアロン・プロシージャを作り、その中でパッケージに定義した型を参照できます。

```
CREATE PACKAGE cv_types AS
    TYPE GenericCurTyp IS REF CURSOR;
    TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
    TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
    ...
```

```
END cv_types;
```

次の例では、パッケージ `cv_types` の中で定義された REF CURSOR 型 `EmpCurTyp` を参照するスタンドアロン・プロシージャを作っています。

```
CREATE PROCEDURE open_emp_cv (emp_cv IN OUT cv_types.EmpCurTyp) AS
BEGIN
    OPEN emp_cv FOR SELECT * FROM emp;
END open_emp_cv;
```

データ検索を集中的に実行するために、ストアド・プロシージャの中で型互換性のある問合せをグループにまとめることができます。次に示す例では、パッケージ・プロシージャは仮パラメータの 1 つとして選択子を宣言しています。(この場合、選択子とは、条件制御文の中でいくつかの代替項目の中から 1 つを選択するために使用する変数です。) コールされた場合、プロシージャは選択された問合せに対してカーソル変数 `emp_cv` をオープンします。

```
CREATE PACKAGE emp_data AS
    TYPE GenericCurTyp IS REF CURSOR;
    TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
    PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp, choice NUMBER);
END emp_data;
```

```
CREATE PACKAGE BODY emp_data AS
    PROCEDURE open_emp_cv (
        emp_cv IN OUT EmpCurTyp,
        choice NUMBER) IS
    BEGIN
        IF choice = 1 THEN
            OPEN emp_cv FOR SELECT * FROM emp WHERE comm IS NOT NULL;
        ELSIF choice = 2 THEN
            OPEN emp_cv FOR SELECT * FROM emp WHERE sal > 2500;
        ELSIF choice = 3 THEN
            OPEN emp_cv FOR SELECT * FROM emp WHERE deptno = 20;
        END IF;
    END open_emp_cv;
END emp_data;
```

さらに柔軟性を高めるために、カーソル変数と選択子を、異なる戻り値の型を指定した問合せを実行するストアド・プロシージャに渡すことができます。次の例を考えてみます。

```
CREATE PACKAGE BODY emp_data AS
    PROCEDURE open_cv (
        generic_cv IN OUT GenericCurTyp,
        choice     NUMBER) IS
    BEGIN
        IF choice = 1 THEN
            OPEN generic_cv FOR SELECT * FROM emp;
        ELSIF choice = 2 THEN
```

```
        OPEN generic_cv FOR SELECT * FROM dept;
    ELSIF choice = 3 THEN
        OPEN generic_cv FOR SELECT * FROM salgrade;
    END IF;
END open_cv;
END emp_data;
```

ホスト変数の使用

OCI や Pro*C プログラムなどの PL/SQL ホスト環境で、カーソル変数を宣言できます。カーソル変数を使う場合は、ホスト変数として PL/SQL に渡さなければなりません。次の Pro*C の例では、ホスト・カーソル変数と選択子を PL/SQL ブロックに渡すことで、選択した問合せ用のカーソル変数をオープンしています。

```
EXEC SQL BEGIN DECLARE SECTION;
...
/* Declare host cursor variable. */
SQL_CURSOR generic_cv;
int         choice;
EXEC SQL END DECLARE SECTION;
...
/* Initialize host cursor variable. */
EXEC SQL ALLOCATE :generic_cv;
...
/* Pass host cursor variable and selector to PL/SQL block. */
EXEC SQL EXECUTE
BEGIN
    IF :choice = 1 THEN
        OPEN :generic_cv FOR SELECT * FROM emp;
    ELSIF :choice = 2 THEN
        OPEN :generic_cv FOR SELECT * FROM dept;
    ELSIF :choice = 3 THEN
        OPEN :generic_cv FOR SELECT * FROM salgrade;
    END IF;
END;
END-EXEC;
```

ホスト・カーソル変数はすべての問合せの戻り型と互換性があります。ホスト・カーソル変数は、弱い型定義の PL/SQL カーソル変数と同じように動作します。

カーソル変数からの取出し

FETCH 文は、複数行の問合せの結果セットから、一度に 1 行ずつ行を取り出します。次に構文を示します。

```
FETCH {cursor_variable_name | :host_cursor_variable_name}
      INTO {variable_name[, variable_name]... | record_name};
```


次の例では、カーソル変数 `emp_cv` からユーザー定義のレコード `emp_rec` へ行を取り出します。

```
LOOP
  /* Fetch from cursor variable. */
  FETCH emp_cv INTO emp_rec;
  EXIT WHEN emp_cv%NOTFOUND; -- exit when last row is fetched
  -- process data record
END LOOP;
```

カーソル変数がオープンしている場合だけ、対応付けられた問合せの中の変数が評価されます。結果セットや問合せの中の変数の値を変更する場合は、カーソル変数を新しい値に設定して再オープンしなければなりません。ただし、同じカーソル変数を使って、別々の `FETCH` 文で、異なる `INTO` 句を使えます。各 `FETCH` 文で同じ結果セットから別の行を取り出します。

PL/SQL では、カーソル変数の戻り型が、必ず `FETCH` 文の `INTO` 句と互換性を持ちます。カーソル変数と結び付けられた問合せが戻す列の値に対して、`INTO` 句の中に、対応する型互換性のあるフィールドまたは変数が存在しなければなりません。また、フィールドまたは変数の数は、列の値の数と等しくなければなりません。そうでない場合はエラーになります。

カーソル変数が強い型定義の場合はコンパイル時に、弱い型定義の場合は実行時にエラーが発生します。実行時、PL/SQL は最初の取出しの前に、事前定義の例外 `ROWTYPE_MISMATCH` を呼び出します。したがって、エラーをトラップし、異なる `INTO` 句を使って `FETCH` 文を実行すると、行は失われません。

カーソル変数を、そのカーソル変数から取り出すサブプログラムの仮パラメータとして宣言する場合は、`IN` または `IN OUT` モードを指定しなければなりません。ただし、サブプログラムがカーソル変数もオープンする場合は、`IN OUT` モードを指定しなければなりません。

クローズしている、または一度もオープンされていないカーソル変数から取出しを実行すると、PL/SQL によって事前定義の例外 `INVALID_CURSOR` が呼び出されます。

カーソル変数のクローズ

カーソル変数は `CLOSE` 文によって使用禁止になります。その後、対応付けられた結果セットは未定義になります。次に構文を示します。

```
CLOSE {cursor_variable_name | :host_cursor_variable_name};
```

次の例では、最後の行が処理された時点でカーソル変数 `emp_cv` をクローズします。

```
LOOP
  FETCH emp_cv INTO emp_rec;
  EXIT WHEN emp_cv%NOTFOUND;
  -- process data record
END LOOP;
/* Close cursor variable. */
```

```
CLOSE emp_cv;
```

カーソル変数を、そのカーソル変数をクローズするサブプログラムの仮パラメータとして宣言する場合は、IN または IN OUT モードを指定しなければなりません。

すでにクローズされているか一度もオープンされたことのないカーソル変数をクローズすると、PL/SQL によって事前定義の例外 `INVALID_CURSOR` が呼び出されます。

例 1

次に示すストアド・プロシージャでは、メイン・ライブラリのデータベースで本、雑誌、テープを検索しています。マスター表には、各項目のタイトルとカテゴリ・コード（1= 本、2= 雑誌、3= テープ）が格納されています。3 つの細目表にはカテゴリ固有の情報が格納されています。コールされると、プロシージャはマスター表をタイトルで検索し、対応付けられたカテゴリ・コードを使って OPEN-FOR 文を選択し、適切な細目表の問合せ用にカーソル変数をオープンします。

```
CREATE PACKAGE cv_types AS
    TYPE LibCurTyp IS REF CURSOR;
    ...
END cv_types;

CREATE PROCEDURE find_item (title VARCHAR2(100),
                           lib_cv IN OUT cv_types.LibCurTyp) AS
    code BINARY_INTEGER;
BEGIN
    SELECT item_code FROM titles INTO code
    WHERE item_title = title;
    IF code = 1 THEN
        OPEN lib_cv FOR SELECT * FROM books
        WHERE book_title = title;
    ELSIF code = 2 THEN
        OPEN lib_cv FOR SELECT * FROM periodicals
        WHERE periodical_title = title;
    ELSIF code = 3 THEN
        OPEN lib_cv FOR SELECT * FROM tapes
        WHERE tape_title = title;
    END IF;
END find_item;
```

例 2

ブランチ・ライブラリのクライアント側アプリケーションは、次の PL/SQL ブロックを使って、検索した情報を表示できます。

```
DECLARE
    lib_cv          cv_types.LibCurTyp;
```

```

book_rec      books%ROWTYPE;
periodical_rec periodicals%ROWTYPE;
tape_rec      tapes%ROWTYPE;
BEGIN
  get_title(:title); -- title is a host variable
  find_item(:title, lib_cv);
  FETCH lib_cv INTO book_rec;
  display_book(book_rec);
EXCEPTION
  WHEN ROWTYPE_MISMATCH THEN
    BEGIN
      FETCH lib_cv INTO periodical_rec;
      display_periodical(periodical_rec);
    EXCEPTION
      WHEN ROWTYPE_MISMATCH THEN
        FETCH lib_cv INTO tape_rec;
        display_tape(tape_rec);
    END;
END;

```

例 3

次に示す Pro*C プログラムは、データベース表を選択するようユーザーに求め、その表の問合せ用にカーソル変数をオープンしてから、問合せによって戻された行を取り出します。

```

#include <stdio.h>
#include <sqlca.h>
void sql_error();
main()
{
  char temp[32];
  EXEC SQL BEGIN DECLARE SECTION;
  char * uid = "scott/tiger";
  SQL_CURSOR generic_cv; /* cursor variable */
  int table_num; /* selector */
  struct /* EMP record */
  {
    int emp_num;
    char emp_name[11];
    char job_title[10];
    int manager;
    char hire_date[10];
    float salary;
    float commission;
    int dept_num;
  } emp_rec;
  struct /* DEPT record */

```

```

    {
        int    dept_num;
        char   dept_name[15];
        char   location[14];
    } dept_rec;
    struct          /* BONUS record */
    {
        char   emp_name[11];
        char   job_title[10];
        float  salary;
    } bonus_rec;
EXEC SQL END DECLARE SECTION;
/* Handle Oracle errors. */
EXEC SQL WHENEVER SQLERROR DO sql_error();

/* Connect to Oracle. */
EXEC SQL CONNECT :uid;

/* Initialize cursor variable. */
EXEC SQL ALLOCATE :generic_cv;

/* Exit loop when done fetching. */
EXEC SQL WHENEVER NOT FOUND DO break;

for (;;)
{
    printf("\n1 = EMP, 2 = DEPT, 3 = BONUS");
    printf("\nEnter table number (0 to quit): ");
    gets(temp);
    table_num = atoi(temp);
    if (table_num <= 0) break;

    /* Open cursor variable. */
    EXEC SQL EXECUTE
        BEGIN
            IF :table_num = 1 THEN
                OPEN :generic_cv FOR SELECT * FROM emp;
            ELSIF :table_num = 2 THEN
                OPEN :generic_cv FOR SELECT * FROM dept;
            ELSIF :table_num = 3 THEN
                OPEN :generic_cv FOR SELECT * FROM bonus;
            END IF;
        END;
    END-EXEC;
    for (;;)
    {
        switch (table_num)
```

```

        {
            case 1: /* Fetch row into EMP record. */
                EXEC SQL FETCH :generic_cv INTO :emp_rec;
                break;
            case 2: /* Fetch row into DEPT record. */
                EXEC SQL FETCH :generic_cv INTO :dept_rec;
                break;
            case 3: /* Fetch row into BONUS record. */
                EXEC SQL FETCH :generic_cv INTO :bonus_rec;
                break;
        }
        /* Process data record here. */
    }
    /* Close cursor variable. */
    EXEC SQL CLOSE :generic_cv;
}
exit(0);
}
void sql_error()
{
    /* Handle SQL error here. */
}

```

例 4

ホスト変数はホスト環境で宣言する変数です。したがって、1つまたは複数の PL/SQL プログラムに渡します。PL/SQL プログラムではホスト変数を他の変数と同様に使用できます。SQL*Plus 環境では、コマンド VARIABLE を使用してホスト変数を宣言します。たとえば、次のように NUMBER 型の変数を宣言します。

```
VARIABLE return_code NUMBER
```

SQL*Plus と PL/SQL のどちらもホスト変数を参照できますが、SQL*Plus はその値を表示することもできます。

注意： PL/SQL プログラム変数と同じ名前でホスト変数を宣言した場合には、PL/SQL プログラム変数が優先されます。

PL/SQL でホスト変数を参照するには、次の例に示すように、その名前にコロン (:) を接頭辞として付ける必要があります。

```

BEGIN
    :return_code := 0;
    IF credit_check_ok(acct_no) THEN
        :return_code := 1;
    END IF;
    ...
END;

```

SQL*Plus でホスト変数の値を表示するには、次のように、PRINT コマンドを使用します。

```
SQL> PRINT return_code
```

```
RETURN_CODE
-----
1
```

SQL*Plus データ型の REF CURSOR を使用すると、カーソル変数を宣言することができ、さらにそのカーソル変数を使ってストアド・サブプログラムから照会結果を戻すことができます。次のスクリプトでは、REF CURSOR 型のホスト変数を宣言します。照会結果を自動的に表示するには、SQL*Plus コマンド SET AUTOPRINT ON を使います。

```
CREATE PACKAGE emp_data AS
  TYPE EmpRecTyp IS RECORD (
    emp_id    NUMBER(4),
    emp_name  CHAR(10),
    job_title CHAR(9),
    dept_name CHAR(14),
    dept_loc  CHAR(13));
  TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
  PROCEDURE get_staff (
    dept_no IN NUMBER,
    emp_cv  IN OUT EmpCurTyp);
END;
/
CREATE PACKAGE BODY emp_data AS
  PROCEDURE get_staff (
    dept_no IN NUMBER,
    emp_cv  IN OUT EmpCurTyp) IS
  BEGIN
    OPEN emp_cv FOR
      SELECT empno, ename, job, dname, loc FROM emp, dept
      WHERE emp.deptno = dept_no AND emp.deptno = dept.deptno
      ORDER BY empno;
  END;
END;
/
COLUMN EMPNO HEADING Number
COLUMN ENAME HEADING Name
COLUMN JOB HEADING JobTitle
COLUMN DNAME HEADING Department
COLUMN LOC HEADING Location
SET AUTOPRINT ON

VARIABLE cv REF CURSOR
EXECUTE emp_data.get_staff(20, :cv)
```

ネットワーク通信量の削減

ホスト・カーソル変数を PL/SQL に渡す場合、OPEN-FOR 文をグループ化することでネットワーク通信量を削減できます。たとえば、次の PL/SQL ブロックは、1 回の往復で 5 つのカーソル変数をオープンしています。

```
/* anonymous PL/SQL block in host environment */
BEGIN
  OPEN :emp_cv FOR SELECT * FROM emp;
  OPEN :dept_cv FOR SELECT * FROM dept;
  OPEN :grade_cv FOR SELECT * FROM salgrade;
  OPEN :pay_cv FOR SELECT * FROM payroll;
  OPEN :ins_cv FOR SELECT * FROM insurance;
END;
```

これは Oracle Forms で便利です（たとえば、マルチブロック・フォームに挿入する場合など）。

ホスト・カーソル変数を PL/SQL ブロックに渡してオープンする場合、ホスト・カーソル変数が指す問合せ作業域は、ブロックの終了後もアクセス可能な状態のままです。そのため、OCI や Pro*C プログラムで、通常のカーソル操作にその作業域を使えます。次の例では、1 回の往復でこのような作業域をいくつかオープンします。

```
BEGIN
  OPEN :c1 FOR SELECT 1 FROM dual;
  OPEN :c2 FOR SELECT 1 FROM dual;
  OPEN :c3 FOR SELECT 1 FROM dual;
  OPEN :c4 FOR SELECT 1 FROM dual;
  OPEN :c5 FOR SELECT 1 FROM dual;
  ...
END;
```

c1、c2、c3、c4、c5 に代入されたカーソルは通常どおり動作し、あらゆる用途に使えます。終了すると、次のように単にカーソルを解放します。

```
BEGIN
  CLOSE :c1;
  CLOSE :c2;
  CLOSE :c3;
  CLOSE :c4;
  CLOSE :c5;
  ...
END;
```

エラーの回避

代入に関係する両方のカーソル変数が強い型定義である場合は、両方が同じデータ型でなければなりません。次の例では、カーソル変数の戻り型は同じですがデータ型が異なるために、代入すると例外が呼び出されます。

```
DECLARE
  TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
  TYPE TmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
  PROCEDURE open_emp_cv (
    emp_cv IN OUT EmpCurTyp,
    tmp_cv IN OUT TmpCurTyp) IS
  BEGIN
    ...
    emp_cv := tmp_cv; -- causes 'wrong type' error
  END;
```

ただし、一方または両方のカーソル変数が弱い型定義である場合は、同じデータ型でなくてもかまいません。

問合せ作業域を指していないカーソル変数に対して取出しまたはクローズを実行するか、カーソルの属性を適用しようとする、PL/SQL によって `INVALID_CURSOR` が呼び出されます。カーソル変数（またはパラメータ）が問合せ作業域を指すようにするには、次の 2 通りの方法があります。

- `OPEN-FOR` 文でカーソル変数を問合せ用にオープンする
- `OPEN` 文ですでにオープンされたホスト・カーソル変数または PL/SQL カーソル変数の値を、カーソル変数に代入する

次の例は、この 2 つの方法がどのように互いに関係しているかを示しています。

```
DECLARE
  TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
  emp_cv1 EmpCurTyp;
  emp_cv2 EmpCurTyp;
  emp_rec emp%ROWTYPE;
BEGIN
  /* The following assignment is useless because emp_cv1
     does not point to a query work area yet. */
  emp_cv2 := emp_cv1; -- useless
  /* Make emp_cv1 point to a query work area. */
  OPEN emp_cv1 FOR SELECT * FROM emp;
  /* Use emp_cv1 to fetch first row from emp table. */
  FETCH emp_cv1 INTO emp_rec;
  /* The following fetch raises an exception because emp_cv2
     does not point to a query work area yet. */
  FETCH emp_cv2 INTO emp_rec; -- raises INVALID_CURSOR
EXCEPTION
```



```

WHEN INVALID_CURSOR THEN
    /* Make emp_cv1 and emp_cv2 point to same work area. */
    emp_cv2 := emp_cv1;
    /* Use emp_cv2 to fetch second row from emp table. */
    FETCH emp_cv2 INTO emp_rec;
    /* Reuse work area for another query. */
    OPEN emp_cv2 FOR SELECT * FROM old_emp;
    /* Use emp_cv1 to fetch first row from old_emp table.
       The following fetch succeeds because emp_cv1 and
       emp_cv2 point to the same query work area. */
    FETCH emp_cv1 INTO emp_rec; -- succeeds
END;

```

カーソル変数をパラメータとして渡す場合は注意が必要です。実パラメータと仮パラメータの戻り型に互換性がないと、実行時に PL/SQL によって ROWTYPE_MISMATCH が呼び出されます。

次に示す Pro*C の例では、パッケージ REF CURSOR 型を定義し、戻り値の型として emp%ROWTYPE を指定します。次に、新しい型を参照するスタンドアロン・プロシージャを作ります。そして、PL/SQL ブロック内で、dept 表への問合せ用にホスト・カーソル変数をオープンします。後で、オープンしたホスト・カーソル変数をストアド・プロシージャに渡す場合に、PL/SQL によって ROWTYPE_MISMATCH が呼び出されます（実パラメータと仮パラメータの戻り型に互換性がないため）。

```

CREATE PACKAGE cv_types AS
    TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
    ...
END cv_types;
/
CREATE PROCEDURE open_emp_cv (emp_cv IN OUT cv_types.EmpCurTyp) AS
BEGIN
    OPEN emp_cv FOR SELECT * FROM emp;
END open_emp_cv;
/
-- anonymous PL/SQL block in Pro*C program
EXEC SQL EXECUTE
    BEGIN
        OPEN :cv FOR SELECT * FROM dept;
        ...
        open_emp_cv(:cv); -- raises ROWTYPE_MISMATCH
    END;
END-EXEC;

```

カーソル変数の制限

現在のところ、カーソル変数には次の制限があります。

- パッケージの中ではカーソル変数を宣言できない。たとえば、次の宣言は誤りである。

```
CREATE PACKAGE emp_stuff AS
    TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
    emp_cv IN OUT EmpCurTyp; -- illegal
END emp_stuff;
```
- 別のサーバー上にあるリモート・サブプログラムは、カーソル変数の値を受け入れることができない。そのため、RPC を使って、あるサーバーから別のサーバーにカーソル変数を渡すことはできない。
- ホスト・カーソル変数を PL/SQL に渡す場合、サーバー側で同じサーバー・コールで変数をオープンしないかぎり、サーバー側で変数からの取出しはできない。
- OPEN-FOR 文の中でカーソル変数と対応付けられた問合せは、FOR UPDATE できない。
- 比較演算子を使って、カーソル変数が等しいかどうか、または NULL かどうかをテストできない。
- NULL をカーソル変数に代入できない。
- REF CURSOR 型を使用して、CREATE TABLE 文または CREATE VIEW 文に列を指定できない。そのため、データベースの列はカーソル変数の値を格納できない。
- REF CURSOR 型を使って、コレクションの要素の型を指定できない。そのため、ネストした表、索引付き表、または varray の中の要素は、カーソル変数の値を格納できない。
- カーソルとカーソル変数には相互操作性がない。つまり、一方の値が期待されている場所で、もう一方が使えない。たとえば、次のようなカーソル FOR ループは誤りである。

```
DECLARE
    CURSOR emp_cur IS SELECT * FROM emp; -- static cursor
    TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
    emp_cv EmpCurTyp; -- cursor variable
BEGIN
    ...
    FOR emp_rec IN emp_cv LOOP ... -- illegal
END;
```

カーソル属性の使用

カーソル、またはカーソル変数には、それぞれ、%FOUND、%ISOPEN、%NOTFOUND、%ROWCOUNT の 4 つの属性があります。これらの属性をカーソルまたはカーソル変数に付加すると、データ操作文の実行について役立つ情報が戻されます。カーソル属性は、プロシージャ文では使えますが、SQL 文では使えません。

明示カーソルの属性

明示カーソルの属性は、複数行の問合せの実行に関する情報を戻します。明示カーソルまたはカーソル変数をオープンすると、対応する問合せを満たす行が識別され、結果セットが形成されます。行は、結果セットから一度に 1 行ずつ取り出されます。

%FOUND

カーソルまたはカーソル変数のオープン後、最初の取出しが実行されるまでは、%FOUND の結果は NULL になります。その後、直前の取出しが行を戻した場合は TRUE に、直前の取出しが行を戻さなかった場合は FALSE になります。次の例では、%FOUND を使って、アクションを選択しています。

```
LOOP
  FETCH c1 INTO my_ename, my_sal, my_hiredate;
  IF c1%FOUND THEN -- fetch succeeded
    ...
  ELSE -- fetch failed, so exit loop
    EXIT;
  END IF;
END LOOP;
```

カーソルまたはカーソル変数をオープンしていない場合、%FOUND でカーソルまたはカーソル変数を参照すると、事前定義の例外 INVALID_CURSOR が呼び出されます。

%ISOPEN

%ISOPEN の結果は、カーソルまたはカーソル変数をオープンしている場合は TRUE に、その他の場合は FALSE になります。次の例では、%ISOPEN を使って、アクションを選択しています。

```
IF c1%ISOPEN THEN -- cursor is open
  ...
ELSE -- cursor is closed, so open it
  OPEN c1;
END IF;
```

%NOTFOUND

%NOTFOUND は、論理的に %FOUND の逆です。%NOTFOUND は、直前の取出しが行を戻した場合は FALSE に、直前の取出しが行を戻さなかった場合は TRUE になります。次の例では、%NOTFOUND を使って、FETCH が行を戻さなくなった場合に、ループが終了するようにしています。

```
LOOP
    FETCH c1 INTO my_ename, my_sal, my_hiredate;
    EXIT WHEN c1%NOTFOUND;
    ...
END LOOP;
```

最初のフェッチの前は、%NOTFOUND の評価結果は NULL です。したがって、FETCH が正常に実行されない場合には、ループは終了しません。これは、WHEN 条件が真である場合にのみ EXIT WHEN 文が実行されるからです。安全のために、次の EXIT 文をかわりに使用することができます。

```
EXIT WHEN c1%NOTFOUND OR ci%NOTFOUND IS NULL;
```

カーソルまたはカーソル変数をオープンしていない場合、%NOTFOUND でカーソルまたはカーソル変数を参照すると、INVALID_CURSOR が呼び出されます。

%ROWCOUNT

カーソルまたはカーソル変数をオープンしている場合、%ROWCOUNT はゼロになります。最初の取出しが行われるまでは、%ROWCOUNT の結果は 0（ゼロ）となります。その後は、現時点までに取り出された行数になります。取出しで行が戻されるたびに、数値が増えています。次の例では、%ROWCOUNT を使って、取出し行が 10 行を超えた場合にアクションを実行するようにしています。

```
LOOP
    FETCH c1 INTO my_ename, my_deptno;
    IF c1%ROWCOUNT > 10 THEN
        ...
    END IF;
    ...
END LOOP;
```

カーソルまたはカーソル変数をオープンしていない場合、%ROWCOUNT でカーソルまたはカーソル変数を参照すると、INVALID_CURSOR が呼び出されます。

表 5-1 に、OPEN 文、FETCH 文または CLOSE 文を実行する前後での、各カーソル属性の結果を示します。

表 5-1 カーソル属性値

		%FOUND	%ISOPEN	%NOTFOUND	%ROWCOUNT
OPEN	前	例外	FALSE	例外	例外
	後	NULL	TRUE	NULL	0
最初の FETCH	前	NULL	TRUE	NULL	0
	後	TRUE	TRUE	FALSE	1
以降の FETCH	前	TRUE	TRUE	FALSE	1
	後	TRUE	TRUE	FALSE	データに依存
最後の FETCH	前	TRUE	TRUE	FALSE	データに依存
	後	FALSE	TRUE	TRUE	データに依存
CLOSE	前	FALSE	TRUE	TRUE	データに依存
	後	例外	FALSE	例外	例外

注意：

1. カーソルをオープンする前またはカーソルをクローズした後で、%FOUND または %NOTFOUND、%ROWCOUNT を参照すると、INVALID_CURSOR が呼び出されます。
2. 最初の FETCH の後、結果セットが空の場合、%FOUND は FALSE、%NOTFOUND は TRUE、%ROWCOUNT は 0 になります。

例

実験で収集されたデータを保持する data_table という名前の表があり、実験 1 のデータを解析しようとしているとします。次の例では、結果を計算し、temp という名前のデータベース表に格納しています。

```
-- available online in file 'examp5'
DECLARE
    num1    data_table.n1%TYPE; -- Declare variables
    num2    data_table.n2%TYPE; -- having same types as
    num3    data_table.n3%TYPE; -- database columns
    result temp.col1%TYPE;
    CURSOR c1 IS
        SELECT n1, n2, n3 FROM data_table WHERE exper_num = 1;
BEGIN
```

```
OPEN c1;
LOOP
    FETCH c1 INTO num1, num2, num3;
    EXIT WHEN c1%NOTFOUND; -- TRUE when FETCH finds no more rows
    result := num2/(num1 + num3);
    INSERT INTO temp VALUES (result, NULL, NULL);
END LOOP;
CLOSE c1;
COMMIT;
END;
```

次の例では、部品番号 5469 が入っているすべての保管場所を検査し、合計で 1000 個になるまでその内容を取り出します。

```
-- available online in file 'examp6'
DECLARE
    CURSOR bin_cur(part_number NUMBER) IS
        SELECT amt_in_bin FROM bins
            WHERE part_num = part_number AND amt_in_bin > 0
            ORDER BY bin_num
            FOR UPDATE OF amt_in_bin;
    bin_amt          bins.amt_in_bin%TYPE;
    total_so_far     NUMBER(5) := 0;
    amount_needed    CONSTANT NUMBER(5) := 1000;
    bins_looked_at   NUMBER(3) := 0;
BEGIN
    OPEN bin_cur(5469);
    WHILE total_so_far < amount_needed LOOP
        FETCH bin_cur INTO bin_amt;
        EXIT WHEN bin_cur%NOTFOUND;
        -- if we exit, there's not enough to fill the order
        bins_looked_at := bins_looked_at + 1;
        IF total_so_far + bin_amt < amount_needed THEN
            UPDATE bins SET amt_in_bin = 0
                WHERE CURRENT OF bin_cur;
            -- take everything in the bin
            total_so_far := total_so_far + bin_amt;
        ELSE -- we finally have enough
            UPDATE bins SET amt_in_bin = amt_in_bin
                - (amount_needed - total_so_far)
                WHERE CURRENT OF bin_cur;
            total_so_far := amount_needed;
        END IF;
    END LOOP;

    CLOSE bin_cur;
    INSERT INTO temp
```

```
VALUES (NULL, bins_looked_at, '<- bins looked at');  
COMMIT;  
END;
```

暗黙カーソルの属性

暗黙カーソルの属性は、INSERT 文または UPDATE 文、DELETE 文、SELECT INTO 文の実行に関する情報を戻します。カーソル属性の値は、常に直前に実行された SQL 文を参照しています。Oracle が SQL カーソルをオープンするまでは、暗黙カーソルの属性の結果は NULL になります。

%FOUND

SQL のデータ操作文が実行されるまでは、%FOUND の結果は NULL になります。その後、INSERT 文、UPDATE 文、DELETE 文が 1 行または複数の行に作用するか、または SELECT INTO 文が 1 行または複数の行を戻すと、%FOUND の結果は TRUE になります。その他の場合、%FOUND の結果は FALSE になります。次の例では、%FOUND を使って、削除に成功した場合に行を挿入するようにしています。

```
DELETE FROM emp WHERE empno = my_empno;  
IF SQL%FOUND THEN -- delete succeeded  
    INSERT INTO new_emp VALUES (my_empno, my_ename, ...);
```

%ISOPEN

Oracle は、SQL カーソルに対応付けられた SQL 文の実行を終了すると、この SQL カーソルを自動的にクローズします。その結果、%ISOPEN の結果は常に FALSE になります。

%NOTFOUND

%NOTFOUND は、論理的に %FOUND の逆です。INSERT 文、UPDATE 文、DELETE 文がどの行にも作用しないか、または SELECT INTO 文がどの行も戻さないと、%NOTFOUND の結果は TRUE になります。その他の場合、%NOTFOUND の結果は FALSE になります。

%ROWCOUNT

%ROWCOUNT の結果は、INSERT 文、UPDATE 文、DELETE 文の影響を受けた行、または SELECT INTO 文に戻された行の数になります。INSERT 文、UPDATE 文、DELETE 文がどの行にも作用しないか、または SELECT INTO 文がどの行も戻さないと、%ROWCOUNT の結果は 0 になります。次の例では、%ROWCOUNT を使って、削除された行が 10 行を超えた場合にアクションを実行するようにしています。

```
DELETE FROM emp WHERE ...  
IF SQL%ROWCOUNT > 10 THEN -- more than 10 rows were deleted  
    ...  
END IF;
```

SELECT INTO 文が複数の行を戻した場合、PL/SQL によって事前定義の例外 TOO_MANY_ROWS が呼び出され、%ROWCOUNT は、問合せを満たす行の実数ではなく、1 になります。

指針

カーソル属性の値は、常に直前に実行された SQL 文を参照します（その文の場所とは無関係です）。文が別の有効範囲に存在する場合もあります（サブブロックなど）。したがって、属性の値を保存して後で使いたい場合は、ブール変数にただちに代入してください。プロセスが check_status は、%NOTFOUND の値を変更している可能性があるため、次の例のような IF 条件を信頼するのは危険です。

```
BEGIN
    ...
    UPDATE parts SET quantity = quantity - 1 WHERE partno = part_id;
    check_status(part_id); -- procedure call
    IF SQL%NOTFOUND THEN -- dangerous!
        ...
    END;
END;
```

このコードを、次のように改善できます。

```
BEGIN
    ...
    UPDATE parts SET quantity = quantity - 1 WHERE partno = part_id;
    sql_notfound := SQL%NOTFOUND; -- assign value to Boolean variable
    check_status(part_id);
    IF sql_notfound THEN ...
END;
```

SELECT INTO 文が行を戻せなかった場合は、次の行で %NOTFOUND をチェックしているかどうかにかかわらず、PL/SQL によって事前定義の例外 NO_DATA_FOUND が呼び出されます。次の例を考えてみます。

```
BEGIN
    ...
    SELECT sal INTO my_sal FROM emp WHERE empno = my_empno;
    -- might raise NO_DATA_FOUND
    IF SQL%NOTFOUND THEN -- condition tested only when false
        ... -- this action is never taken
    END IF;
```

IF 条件がテストされるのは %NOTFOUND が FALSE の場合だけなので、このチェックは役に立ちません。PL/SQL によって NO_DATA_FOUND が呼び出されると、通常の実行は停止され、ブロックの例外処理部に制御が移ります。

ただし、SQL 集計関数をコールする SELECT INTO 文が、NO_DATA_FOUND を呼び出すことはありません。グループ・ファンクションは、必ず値または NULL を戻すからです。このような場合、次の例で示すように、%NOTFOUND の結果は FALSE になります。


```
BEGIN
...
SELECT MAX(sal) INTO my_sal FROM emp WHERE deptno = my_deptno;
-- never raises NO_DATA_FOUND
IF SQL%NOTFOUND THEN -- always tested but never true
... -- this action is never taken
END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN ... -- never invoked
```

トランザクション処理

このセクションでは、トランザクション処理の方法を説明します。ここでは、データベースの一貫性を守るための基本的な手法を学びます。その中には、Oracle データの変更内容を永久的なものにするか取り消すかを制御する方法も含まれます。

Oracle が管理するジョブまたはタスクは、セッションと呼ばれます。アプリケーション・プログラムまたは Oracle Tool を実行し、Oracle に接続すると、ユーザー・セッションが開始されます。ユーザー・セッションを同時に実行してコンピュータのリソースを共有できるようにするためには、並行性を制御する必要があります。並行性とは、多くのユーザーが同一のデータにアクセスすることです。並行性の制御が十分でないと、データの整合性が失われる可能性があります。つまり、データへの変更が誤った順序で実行される可能性があるということです。

Oracle では、ロックを使ってデータへの同時アクセスを制御します。ロックを使うと、データの表または行のようなデータベース・リソースを一時的に所有できます。そのため、ロックを使っているユーザーが変更を終了するまで、他のユーザーはデータを変更できません。デフォルトのロッキング機構が Oracle のデータと構造体を保護するので、ロックは明示的にする必要はありません。ただし、デフォルトのロッキングを上書きした方がユーザーにとって有益な場合は、表または行に対するデータ・ロックを要求できます。行の共有および排他のような数種類のロッキングのモードから選択できます。

複数のユーザーが同じスキーマ・オブジェクトにアクセスしようとすると、デッドロックが発生することがあります。たとえば、同じ表の更新を行っている 2 人のユーザーがお互いに、現在もう一方のユーザーによってロックされている状態の行を更新しようとした場合、2 人とも待機となる可能性があります。それぞれのユーザーは、もう一方のユーザーによって保留状態となっているリソースを待っています。そのため、直前に関連していたトランザクションにエラーが返されてデッドロックが解かれないうちに、どちらのユーザーも作業を続行できません。

1 人のユーザーが問合せ中である表を、同時に別のユーザーが更新している場合、Oracle は、その問合せに対してデータの「読みみ一貫性」ビューを生成します。つまりある問合せが開始され、進行していく間、その問合せによって読み込まれたデータは変更されません。更新アクティビティが継続している間、Oracle は表のデータのスナップショットをとり、変更内容をロールバック・セグメントの中に記録します。Oracle は、ロールバック・セグメン

トを使って、読み込み一貫性のある問合せ結果を作成し、必要に応じて変更内容を取り消します。

トランザクションがデータベースを保護する方法

トランザクションとは、論理作業単位を実行する一連の SQL のデータ操作文です。Oracle では、一連の SQL 文を一単位として扱い、それらの文によって実行されたすべての変更が、同時にコミット（永久化）されるか、ロールバック（取消し）されます。トランザクション中にプログラムに障害が発生すると、データベースは自動的にその前の状態に復元されます。

プログラム中の最初の SQL 文でトランザクションが開始されます。1 つのトランザクションが終了すると、次の SQL 文で自動的に別のトランザクションが開始されます。このように、個々の SQL 文はトランザクションの一部になっています。分散トランザクションとは、分散データベースの複数のノードにあるデータを更新する SQL 文を少なくとも 1 つは含んでいるものです。

COMMIT 文と ROLLBACK 文を使うと、SQL 操作によってなされたデータベースの変更を、その時点で確定するか取り消すことができます。現在のトランザクションは、最後のコミットまたはロールバックよりも後に実行されたすべての SQL 文で構成されます。SAVEPOINT 文は、トランザクション処理の過程で、現在の位置に名前を付けてマークします。

COMMIT の使用

COMMIT 文は、現在のトランザクションを終了し、トランザクションの中でなされた変更をすべて永久的なものとしめます。変更内容をコミットするまで、他のユーザーは変更されたデータにアクセスできません。他のユーザーは変更前のデータを見ることになります。

銀行口座の間で振替えを実行する単純なトランザクションを考えます。このトランザクションでは、1 番目の口座から出金し、2 番目の口座に入金するため、2 つの更新が必要です。次の例では、2 番目の口座に入金した後にコミットを実行し、変更内容を確定しています。そうすることで、他のユーザーが変更内容を見ることができるようになります。

```
BEGIN
...
UPDATE accts SET bal = my_bal - debit
WHERE acctno = 7715;
...
UPDATE accts SET bal = my_bal + credit
WHERE acctno = 7720;
COMMIT WORK;
END;
```

COMMIT 文はすべての行と表のロックを解除します。また、最後のコミットまたはロールバック以降にマークされたセーブポイントをすべて消去します（セーブポイントの詳細は後述します）。オプションのキーワード WORK には、わかりやすくするという効果しかありません。キーワード END は、トランザクションの終わりではなく、PL/SQL ブロックの終わり

を示すキーワードです。ブロックが複数のトランザクションにまたがることができるように、トランザクションも複数のブロックにまたがることができます。

COMMENT 句を使うと、分散トランザクションに対応付けるコメントを指定できます。コミットを発行すると、分散トランザクションによって影響を受けた各データベースの変更内容は永久的なものとなります。しかし、コミットの際にネットワークやマシンが障害を起こして、分散トランザクションが未知の状態または疑わしい状態になる場合があります。このとき Oracle は、COMMENT で指定されたテキストを、トランザクション ID とともにデータ・ディクショナリに格納します。テキストは引用符で囲んだ 50 文字以内のリテラルでなければなりません。次に例を示します。

```
COMMIT COMMENT 'In-doubt order transaction; notify Order Entry';
```

PL/SQL は FORCE 句をサポートしていません (FORCE 句は、SQL で、インダウト分散トランザクションを手動でコミットする句です)。たとえば、次の COMMIT 文は誤りです。

```
COMMIT FORCE '23.51.54'; -- illegal
```

ROLLBACK の使用

ROLLBACK 文は、現在のトランザクションを終了し、トランザクションの中でなされた変更をすべて取り消します。ロールバックが使われる理由は 2 つあります。第 1 に、表から間違った行を削除したなどの誤りを犯した場合に、ロールバックは元のデータを復元できます。第 2 に、例外が呼び出されたり SQL 文が失敗したために終了できないトランザクションを開始してしまった場合、ロールバックを使うと、開始点まで戻って対処措置をし、実行し直すことができます。

次の例では、3 つの異なるデータベース表に従業員に関する情報を挿入しています。3 つの表には、従業員番号を保持するための、表ごとに固有の索引によって制約されている列があります。INSERT 文で重複する従業員番号を格納すると、事前定義の例外 DUP_VAL_ON_INDEX が呼び出されます。このような場合にはすべての変更内容を取り消したいので、例外ハンドラでロールバックを発行します。

```
DECLARE
    emp_id  INTEGER;
    ...
BEGIN
    SELECT empno, ... INTO emp_id, ... FROM new_emp WHERE ...
    ...
    INSERT INTO emp VALUES (emp_id, ...);
    INSERT INTO tax VALUES (emp_id, ...);
    INSERT INTO pay VALUES (emp_id, ...);
    ...
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK;
    ...
END;
```

文レベルのロールバック

SQL 文を実行する前に、Oracle は暗黙的なセーブポイントをマークします。その文が失敗すると、Oracle は自動的にロールバックします。たとえば、INSERT 文で一意的索引に重複する値を挿入しようとしたために例外が呼び出されると、その文はロールバックされます。失われるのは失敗した SQL 文による処理だけです。現在のトランザクション中のその文以前の処理は、保存されます。

Oracle では、デッドロックを解消するために SQL 文を 1 文だけロールバックすることもできます。Oracle は関係しているトランザクションの 1 つにエラーを返し、そのトランザクション中の現在の文をロールバックします。

SQL 文を実行する前に、Oracle はその文を解析しなければなりません。すなわち、その文が構文規則に従っているかどうかや、有効なスキーマ・オブジェクトを参照しているかどうかを確認しなければなりません。SQL 文の実行時に検出されたエラーはロールバックを引き起こしますが、文の解析の際に検出されたエラーはロールバックを引き起こしません。

SAVEPOINT の使用

SAVEPOINT は、トランザクション処理内の現在位置に名前とマークを付けます。セーブポイントを ROLLBACK TO 文と組み合わせると、トランザクション全体ではなく、トランザクションの一部を取り消すことができます。次の例では、挿入する前にセーブポイントをマークしています。INSERT 文で empno 列に重複した値を格納すると、事前定義の例外 DUP_VAL_ON_INDEX が呼び出されます。この場合は、セーブポイントまでロールバックして、その挿入だけを取り消すことができます。

```
DECLARE
    emp_id  emp.empno%TYPE;
BEGIN
    UPDATE emp SET ... WHERE empno = emp_id;
    DELETE FROM emp WHERE ...
    ...
    SAVEPOINT do_insert;
    INSERT INTO emp VALUES (emp_id, ...);
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK TO do_insert;
END;
```

あるセーブポイントまでロールバックすると、そのセーブポイント以降にマークされたセーブポイントはすべて消去されます。ただし、ロールバック先のセーブポイントは消去されません。たとえば、セーブポイントを 5 つマークし、3 番目のセーブポイントまでロールバックすると、4 番目と 5 番目のセーブポイントだけが消去されます。単に、ロールバックまたはコミットだけをするすべてのセーブポイントが消去されます。

再帰的なサブプログラムの中でセーブポイントをマークすると、再帰しながら進んでいく過程で、各レベルで SAVEPOINT 文の新しいインスタンスが実行されます。ただし、ロールバックできるのは直前にマークされたセーブポイントまでだけです。

セーブポイント名は未宣言の識別子で、トランザクションの中で再利用できます。再利用すると、セーブポイントはトランザクションの中の古い位置から現在の位置に移動します。つまり、セーブポイントへのロールバックは、トランザクションの現在の部分だけに影響を与えます。たとえば、

```
BEGIN
...
SAVEPOINT my_point;
UPDATE emp SET ... WHERE empno = emp_id;
...
SAVEPOINT my_point; -- move my_point to current point
INSERT INTO emp VALUES (emp_id, ...);
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK TO my_point;
END;
```

セッションごとのアクティブなセーブポイントの数には、制限がありません。アクティブなセーブポイントとは、最後のコミットまたはロールバック以降にマークされたセーブポイントのことです。

暗黙的なロールバック

INSERT 文、UPDATE 文または DELETE 文を実行する前に、Oracle は（ユーザーが利用できない）暗黙的なセーブポイントをマークします。文の実行に失敗すると、Oracle はこのセーブポイントまでロールバックします。通常は、トランザクション全体ではなく、失敗した SQL 文だけがロールバックされます。しかし、その文が原因で未処理例外が呼び出された場合は、ホスト環境によってロールバックの対象が決まります。

ストアド・サブプログラムを未処理例外で終了すると、PL/SQL は値を OUT パラメータに割り当てません。また、サブプログラムが行ったデータベース処理をロールバックしません。

トランザクションの終了

すべてのトランザクションを明示的にコミットまたはロールバックすることは、プログラミングの習慣として好ましいことです。PL/SQL プログラムまたはホスト環境のどちらで、コミットまたはロールバックのどちらを発行するかは、アプリケーションの論理の流れによって決まります。トランザクションを明示的にコミットまたはロールバックしなかった場合は、ホスト環境によって最終的な状態が決定されます。

たとえば、SQL*Plus 環境で、PL/SQL ブロックに COMMIT 文または ROLLBACK 文がない場合、トランザクションの最終状態はそのブロックの実行後に何をするか依存します。ユーザーがデータ定義文、データ制御文または COMMIT 文を実行するか、EXIT コマンド、DISCONNECT コマンドまたは QUIT コマンドを発行すると、Oracle はトランザクションをコミットします。ROLLBACK 文を実行するか SQL*Plus セッションを中断すると、Oracle はトランザクションをロールバックします。

Oracle プリコンパイラ環境では、プログラムが正常に終了しない場合、Oracle はトランザクションをロールバックします。次に示すように、作業を明示的にコミットまたはロールバックし、RELEASE パラメータを使って Oracle から切断すると、プログラムは正常に終了します。

```
EXEC SQL COMMIT WORK RELEASE;
```

SET TRANSACTION の使用

SET TRANSACTION 文を使うと、読取り専用または読取り書込みトランザクションを開始したり、分離レベルを確立したり、指定したロールバック・セグメントに現在のトランザクションを割り当てたりできます。読取り専用トランザクションは、他のユーザーが更新中である 1 つまたは複数の表に対して、複数の問合せを実行する場合に便利です。

読取り専用トランザクションでは、複数の表と複数の問合せで構成された読みみ一貫性のあるビューが作られ、すべての問合せがデータベースの同一のスナップショットを参照します。他のユーザーは、通常の方法でデータの問合せや更新ができます。コミットまたはロールバックするとトランザクションが終了します。次の例では、スーパーマーケットの店長が、読取り専用トランザクションを使って、当日、先週および先月の売上を調べています。トランザクションの途中で他のユーザーがデータベースを更新しても、売上の数値には影響がありません。

```
DECLARE
    daily_sales    REAL;
    weekly_sales   REAL;
    monthly_sales  REAL;
BEGIN
    ...
    COMMIT; -- ends previous transaction
    SET TRANSACTION READ ONLY;
    SELECT SUM(amt) INTO daily_sales FROM sales
        WHERE dte = SYSDATE;
    SELECT SUM(amt) INTO weekly_sales FROM sales
        WHERE dte > SYSDATE - 7;
    SELECT SUM(amt) INTO monthly_sales FROM sales
        WHERE dte > SYSDATE - 30;
    COMMIT; -- ends read-only transaction
    ...
END;
```

SET TRANSACTION 文は、読取り専用トランザクションの最初の SQL 文でなければならず、1 つのトランザクションで 1 回しか使えません。トランザクションを READ ONLY に設定すると、それ以降の問合せからはトランザクションの開始前にコミットされた変更内容しか見えません。READ ONLY を使っても、他のユーザーや他のトランザクションには影響がありません。

SET TRANSACTION の制限

読取り専用のトランザクションに使用できるのは、SELECT INTO、OPEN、FETCH、CLOSE、LOCK TABLE、COMMIT、ROLLBACK 文だけです。また、問合せは FOR UPDATE にはできません。

デフォルトのロックの上書き

デフォルトで、Oracle はデータ構造を自動的にロックします。しかし、デフォルトのロックを上書きした方がいい場合は、特定の行や表を対象とするデータ・ロックを要求できます。明示的なロックにより、トランザクションの途中で表に対するアクセスを共有または拒否できます。

LOCK TABLE 文を使うと、明示的に表全体をロックできます。SELECT FOR UPDATE 文を使うと、表の中の特定の行を明示的にロックすることで、更新や削除が実行される前に行が変更されることを防止できます。ただし、Oracle は更新または削除時に自動的に行レベル・ロックを取得します。そのため、FOR UPDATE 句は、更新または削除の前に行をロックした場合以外は使わないでください。

FOR UPDATE の使用

UPDATE 文または DELETE 文の CURRENT OF 句で参照されるカーソルを宣言する場合は、FOR UPDATE 句を使って排他的な行ロックを取得する必要があります。たとえば、

```
DECLARE
  CURSOR c1 IS SELECT empno, sal FROM emp
    WHERE job = 'SALESMAN' AND comm > sal
    FOR UPDATE NOWAIT;
```

FOR UPDATE 句は、これから更新または削除される行を識別し、結果セットの中で各行をロックします。これは、行の中の既存の値に基づいて更新する場合に便利です。この場合、更新の前に他のユーザーが行を変更しないようにする必要があります。

オプションのキーワード NOWAIT を指定すると、Oracle は他のユーザーが表をロックしていても待機しません。制御はただちにプログラムに戻されるので、他の処理を行ってから、改めてロックを試みてください。キーワード NOWAIT を省略すると、Oracle は表が利用できるようになるまで待ちます。

カーソルをオープンしたときにすべての行がロックされるのであり、行が取り出されるときにロックされるものではありません。また、トランザクションをコミットすると、行のロックは解除されます。つまり、コミットの後で FOR UPDATE カーソルからの取出しはできません。（対策の詳細は、5-44 ページの「[コミットにまたがる取出し](#)」を参照してください。）

複数の表に対して問合せを実行する場合は、FOR UPDATE 句を使って、ロックを特定の表に制限できます。表中の行は、FOR UPDATE OF 句でその表中の列を参照する場合に限ってロックされます。たとえば、次の問合せでは表 emp の行はロックされますが、表 dept の行はロックされません。

```
DECLARE
  CURSOR c1 IS SELECT ename, dname FROM emp, dept
    WHERE emp.deptno = dept.deptno AND job = 'MANAGER'
    FOR UPDATE OF sal;
```

カーソルから取り出された最新の行を参照するには、次に示すように UPDATE 文または DELETE 文で CURRENT OF 句を使います。

```
DECLARE
  CURSOR c1 IS SELECT empno, job, sal FROM emp FOR UPDATE;
  ...
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO ...
    ...
    UPDATE emp SET sal = new_sal WHERE CURRENT OF c1;
  END LOOP;
```

LOCK TABLE の使用

LOCK TABLE 文を使って、指定されたロック・モードでデータベース表全体をロックすると、表へのアクセスを共有または拒否できます。たとえば、次の文は行共有モードで表 emp をロックします。行共有ロックでは表に対する同時アクセスができます。つまり、他のユーザーが排他的使用のために表全体をロックしないようにします。表ロックは、トランザクションがコミットまたはロールバックを発行したときに解除されます。

```
LOCK TABLE emp IN ROW SHARE MODE NOWAIT;
```

ロック・モードによって、表に対して他にどのようなロックを使えるかが決まります。たとえば、1 つの表に対して多くのユーザーが同時に行共用ロックを取得できますが、排他ロックを取得できるのは一度に 1 人のユーザーだけです。あるユーザーが表に対して排他ロックをかけていると、他のユーザーはその表に対して行の挿入、更新、削除を実行できません。ロック・モードの詳細は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

表がロックされていても、他のユーザーは表に対して問合せできますが、問合せを実行しても表のロックを取得できません。2 つの異なるトランザクションが同じ行を変更した場合だけ、一方のトランザクションがもう一方のトランザクションの終了を待ちます。

コミットにまたがる取出し

FOR UPDATE 句によるロックは排他的な行ロックです。カーソルをオープンするとすべての行はロックされ、トランザクションをコミットすると、ロックは解除されます。つまり、コミットの後で FOR UPDATE カーソルからの取出しはできません。これを実行すると、PL/SQL によって例外が呼び出されます。次の例では、カーソル FOR ループは、10 回目の挿入を行った後に失敗します。


```
DECLARE
  CURSOR c1 IS SELECT ename FROM emp FOR UPDATE OF sal;
  ctr NUMBER := 0;
BEGIN
  FOR emp_rec IN c1 LOOP -- FETCHes implicitly
    ...
    ctr := ctr + 1;
    INSERT INTO temp VALUES (ctr, 'still going');
    IF ctr >= 10 THEN
      COMMIT; -- releases locks
    END IF;
  END LOOP;
END;
```

複数のコミットにまたがってフェッチしたい場合は、FOR UPDATE 句と CURRENT OF 句は使わないでください。そのかわりに、ROWID 疑似列を使って CURRENT OF 句と同じ処理を行います。各行の ROWID を取り出して、UROWID 変数に入れます。その後、更新や削除のときに、ROWID を使って現在行を識別します。たとえば、

```
DECLARE
  CURSOR c1 IS SELECT ename, job, rowid FROM emp;
  my_ename emp.ename%TYPE;
  my_job emp.job%TYPE;
  my_rowid UROWID;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_ename, my_job, my_rowid;
    EXIT WHEN c1%NOTFOUND;
    UPDATE emp SET sal = sal * 1.05 WHERE rowid = my_rowid;
    -- this mimics WHERE CURRENT OF c1
    COMMIT;
  END LOOP;
  CLOSE c1;
END;
```

ここでは注意が必要です。ここでは FOR UPDATE 句を使っていないので、取り出された行がロックされていません。そのため、他のユーザーが意識せずに変更内容を上書きしてしまう可能性があります。また、カーソルが読取り一貫性のあるデータのビューを持っていないければなりません。これは、更新に使われたロールバック・セグメントが、カーソルをクローズするまで解放されないようにするためです。更新する行が多い場合は、処理速度が低下する場合があります。

ROWID 疑似列を参照するカーソルで %ROWTYPE 属性を使う例を次に示します。

```
DECLARE
  CURSOR c1 IS SELECT ename, sal, rowid FROM emp;
  emp_rec c1%ROWTYPE;
```

```
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO emp_rec;
    EXIT WHEN c1%NOTFOUND;
    ...
    IF ... THEN
      DELETE FROM emp WHERE rowid = emp_rec.rowid;
    END IF;
  END LOOP;
  CLOSE c1;
END;
```

サイズ制限への対応

PL/SQL は高速トランザクション処理用に設計されています。その結果、コンパイラでは、プログラム・ユニット（ブロック、サブプログラム、またはパッケージ）に入るトークン（識別子、キーワード、演算子など）の数が制限されています。この制限を超える単位は、「プログラムが大きすぎる」というコンパイルのエラーの原因になります。一般に、32K より大きい単位仕様部および 64K より大きい単位本体は、トークンの制限を超えてしまいます。しかし、これより小さい単位でも、変数や複雑な SQL 文を多く含んでいる場合は、制限を超えてしまう可能性があります。

多くの場合、この問題はパッケージ本体または無名ブロックで発生します。パッケージの場合、最良の方法は複数個のもっと小さなパッケージに分けることです。ブロックの場合、最良の方法は一連のサブプログラムとして再定義し、データベースに格納できるようにすることです。詳細は、[第 7 章の「サブプログラム」](#)を参照してください。

もう 1 つの解決方法は、ブロックを 2 つのサブブロックに分けることです。次の SQL*Plus スクリプトの例で考えてみます。最初のブロックが終了する前に、2 番目のブロックが必要とするデータを、temp という名前のデータベース表に挿入します。2 番目のブロックが実行を開始したら、temp からデータを選択します。これはプロシージャ間でのパラメータの受け渡しに似ています。

```
DECLARE
  mode  NUMBER;
  median NUMBER;
BEGIN
  ...
  INSERT INTO temp (col1, col2, col3)
    VALUES (mode, median, 'blockA');
END;
/
DECLARE
  mode  NUMBER;
  median NUMBER;
BEGIN
```

```
        SELECT col1, col2 INTO mode, median FROM temp
        WHERE col3 = 'blockA';
    ...
END;
/
```

この例がうまく機能しない場合があります。2 番目のブロックの実行中に、1 番目のブロックを再実行しなければならない場合と、複数のユーザーがスクリプトを同時に実行しなければならない場合です。

あるいは、C や COBOL などのホスト言語の中にブロックを埋め込むこともできます。こうすることで、フロー制御文を使って最初のブロックを再実行できます。また、データベース表ではなく、グローバルなホスト変数にデータを格納することもできます。たとえば、次の 2 つのブロックを Pro*C プログラムの中に組み込みます。

```
/* The host variables 'my_sal', 'my_comm', and 'my_empno'
   are assigned values in the host environment. The host
   variable 'comm_ind' is an indicator variable. */
BEGIN
    SELECT sal, comm INTO :my_sal, :my_comm:comm_ind FROM emp
    WHERE empno = :my_empno;
    IF :my_comm:comm_ind IS NULL THEN
        ...
    END IF;
END;

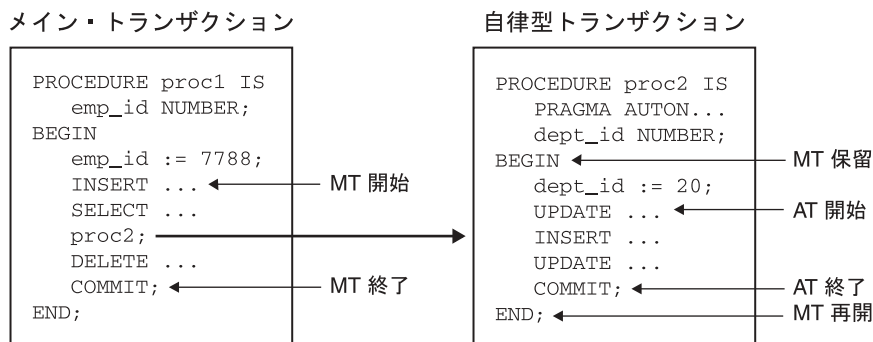
BEGIN
    ...
    IF :my_comm:comm_ind > 1000 THEN
        :my_sal := :my_sal * 1.10;
        UPDATE emp SET sal = :my_sal WHERE empno = :my_empno;
    END IF;
END;
```

自律型トランザクションの使用

トランザクションとは、論理作業単位を実行する一連の SQL 文です。通常は、1 つのトランザクションによって別のトランザクションが開始されます。アプリケーションによっては、あるトランザクションは、そのトランザクションを開始したトランザクションの有効範囲外で操作しなければなりません。これは、たとえば、トランザクションがデータ・カートリッジをコールする場合に発生します。

自律型トランザクションは、別の、メイン・トランザクションによって開始される独立したトランザクションです。自律型トランザクションを使用すると、メイン・トランザクションを停止し、SQL 操作を実行してその操作をコミットまたはロールバックしてから、メイン・トランザクションを再開できます。図 5-1 に、メイン・トランザクション (MT) から自律型トランザクション (AT) へ制御がどのように流れ、また戻るかを示します。

図 5-1 トランザクション制御の流れ



自律型トランザクションの利点

自律型トランザクションは、開始すると完全に独立します。ロック、リソース、コミット依存関係をメイン・トランザクションと共有することはありません。そのため、メイン・トランザクションがロールバックする場合でも、イベントや増分再試行カウンタなどのログを取ることができます。

さらに重要な点は、自律型トランザクションは再利用可能なソフトウェア・コンポーネントであるモジュール構造の作成に役立つということです。たとえば、ストアド・プロシージャは独自に自律型トランザクションを開始したり終了したりできます。コール側のアプリケーションはプロシージャの自律型操作について知る必要はなく、プロシージャはアプリケーションのトランザクション・コンテキストについて知る必要はありません。これにより、自律型トランザクションは通常のトランザクションよりエラー発生の可能性が少なくなり、使いやすくなります。

さらに、自律型トランザクションは通常のトランザクションの機能をすべて備えています。パラレル問合せ、分散処理、および SET TRANSACTION を含むすべてのトランザクション制御文を使用できます。

自律型トランザクションの定義

自律型トランザクションを定義するには、プラグマ（コンパイラ・ディレクティブ）AUTONOMOUS_TRANSACTION を使用します。プラグマはルーチンを自律型（独立型）としてマークするよう PL/SQL コンパイラに指示します。このコンテキストでは、ルーチンには次のものが含まれます。

- トップレベル（ネストしていない）の無名 PL/SQL ブロック
- ローカル、スタンドアロンおよびパッケージのファンクションとプロシージャ
- SQL オブジェクト型のメソッド
- データベース・トリガー

プラグマは、ルーチンの宣言部の任意の場所でコーディングできます。しかし、見やすくするためには、セクションの先頭にプラグマをコーディングしてください。次に構文を示します。

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

次の例では、パッケージ・ファンクションを自律型としてマークします。

```
CREATE PACKAGE banking AS
...
    FUNCTION balance (acct_id INTEGER) RETURN REAL;
END banking;

CREATE PACKAGE BODY banking AS
...
    FUNCTION balance (acct_id INTEGER) RETURN REAL IS
        PRAGMA AUTONOMOUS_TRANSACTION;
        my_bal REAL;
    BEGIN
        ...
    END;
END banking;
```

制限事項： プラグマを使用してパッケージ内のすべてのサブプログラム（またはあるオブジェクト型のすべてのメソッド）を自律型としてマークすることはできません。自律型としてマークできるのは、個々のルーチンだけです。たとえば、次のプログラムは誤りです。

```
CREATE PACKAGE banking AS
    PRAGMA AUTONOMOUS_TRANSACTION; -- illegal
...
```

```
FUNCTION balance (acct_id INTEGER) RETURN REAL;  
END banking;
```

次の例では、スタンドアロン・プロシージャを自律型としてマークします。

```
CREATE PROCEDURE close_account (acct_id INTEGER, OUT balance) AS  
  PRAGMA AUTONOMOUS_TRANSACTION;  
  my_bal REAL;  
BEGIN  
  ...  
END;
```

次の例では、PL/SQL ブロックを自律型としてマークします。

```
DECLARE  
  PRAGMA AUTONOMOUS_TRANSACTION;  
  my_empno NUMBER(4);  
BEGIN  
  ...  
END;
```

制限事項： ネストした PL/SQL ブロックは自律型としてマークできません。

次の例では、データベース・トリガーを自律型としてマークします。通常のトリガーとは異なり、自律型トリガーには、COMMIT および ROLLBACK などのトランザクション制御文を含めることができます。

```
CREATE TRIGGER parts_trigger  
BEFORE INSERT ON parts FOR EACH ROW  
DECLARE  
  PRAGMA AUTONOMOUS_TRANSACTION;  
BEGIN  
  INSERT INTO parts_log VALUES (:new.pnum, :new.pname);  
  COMMIT; -- allowed only in autonomous triggers  
END;
```

自律型トランザクションとネストしたトランザクションとの相違点

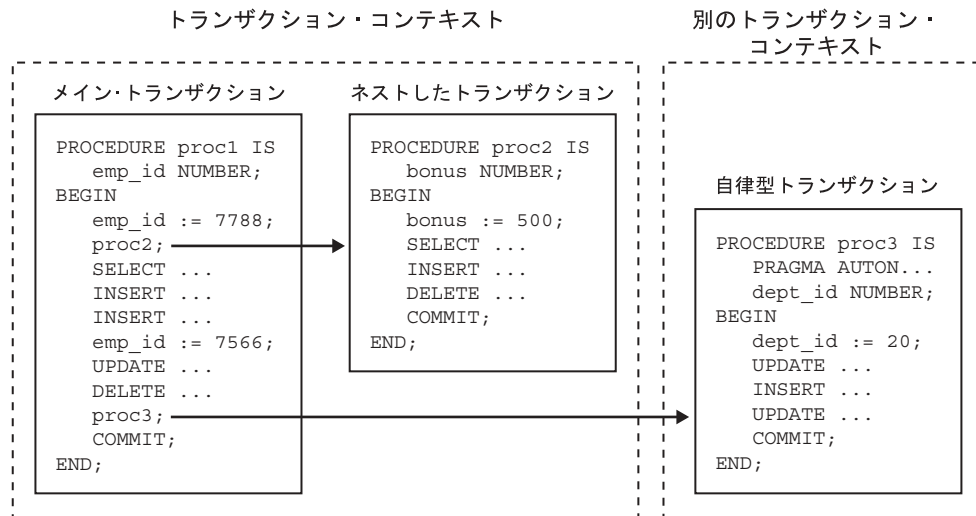
自律型トランザクションは別のトランザクションによって開始されますが、これはネストしたトランザクションではありません。その理由は次のとおりです。

- ロックなどのトランザクション・リソースをメイン・トランザクションと共有しない。
- メイン・トランザクションに依存しない。たとえば、メイン・トランザクションがロールバックする場合、ネストしたトランザクションがロールバックするのに対し、自律型トランザクションはロールバックしない。
- コミットされた変更を、他のトランザクションからすぐに参照できる。(ネストしたトランザクションのコミットされた変更は、メイン・トランザクションがコミットするまで他のトランザクションからは参照できない。)

トランザクション・コンテキスト

図 5-2 に示すように、メイン・トランザクションはそのコンテキストをネストしたトランザクションと共有しますが、自律型トランザクションとは共有しません。同様に、ある自律型ルーチンが別の自律型ルーチンを（または自身を再帰的に）コールする場合、ルーチンはトランザクション・コンテキストを共有しません。ただし、ある自律型ルーチンが自律型でないルーチンをコールする場合、ルーチンは同じトランザクション・コンテキストを共有します。

図 5-2 トランザクション・コンテキスト



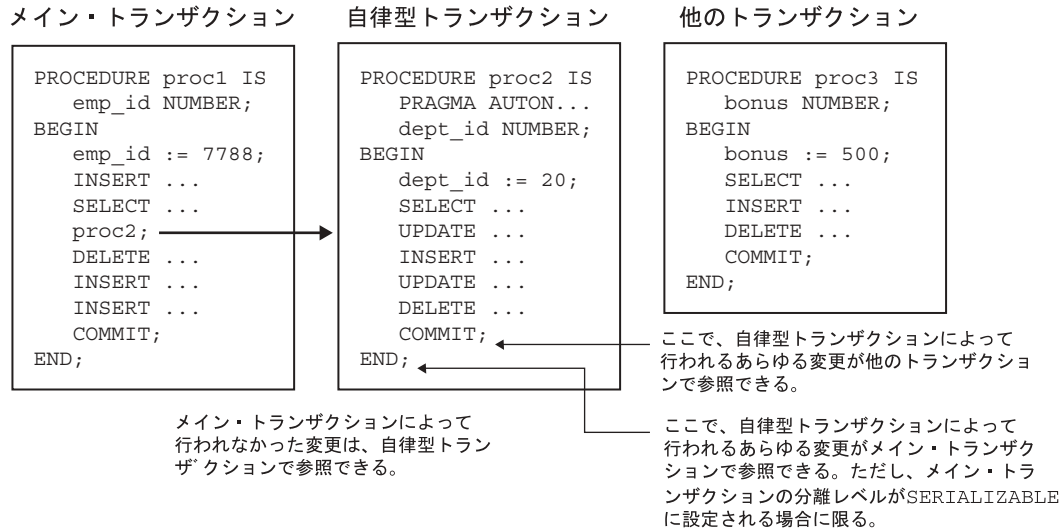
トランザクションの可視性

図 5-35-52 ページの示すように、自律型トランザクションによって行われた変更は、自律型トランザクションがコミットすると、他のトランザクションから参照できるようになります。変更は、メイン・トランザクションが再開するとメイン・トランザクションからも参照できるようになりますが、これは分離レベルが READ COMMITTED（デフォルト）に設定されている場合だけです。

メイン・トランザクションの分離レベルを、次に示すように SERIALIZABLE に設定すると、その自律型トランザクションによって行われた変更は、再開してもメイン・トランザクションからは参照できません。

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

図 5-3 トランザクションの可視性



自律型トランザクションの制御

自律型ルーチンの最初の SQL 文でトランザクションが開始されます。1 つのトランザクションが終了すると、次の SQL 文で別のトランザクションが開始されます。現在のトランザクションは、最後のコミットまたはロールバックよりも後に実行されたすべての SQL 文で構成されます。自律型トランザクションを制御するには、次の文を使用します。これは現在の（アクティブな）トランザクションだけに適用されます。

- COMMIT
- ROLLBACK [TO savepoint_name]
- SAVEPOINT savepoint_name
- SET TRANSACTION

COMMIT 文は、現在のトランザクションを終了し、トランザクションの中でなされた変更を永久的なものとして保存します。ROLLBACK 文は、現在のトランザクションを終了し、トランザクションの中でなされた変更を取り消します。ROLLBACK TO は、トランザクションの一部だけを取り消します。SAVEPOINT は、トランザクション内の現在位置に名前とマークを付けます。SET TRANSACTION は、読み書きアクセスや分離レベルなど、トランザクションのプロパティを設定します。

注意：メイン・トランザクションで設定されたトランザクションのプロパティは、そのトランザクションだけに適用され、自律型トランザクションには適用されません。逆も同じことが言えます。

開始と終了

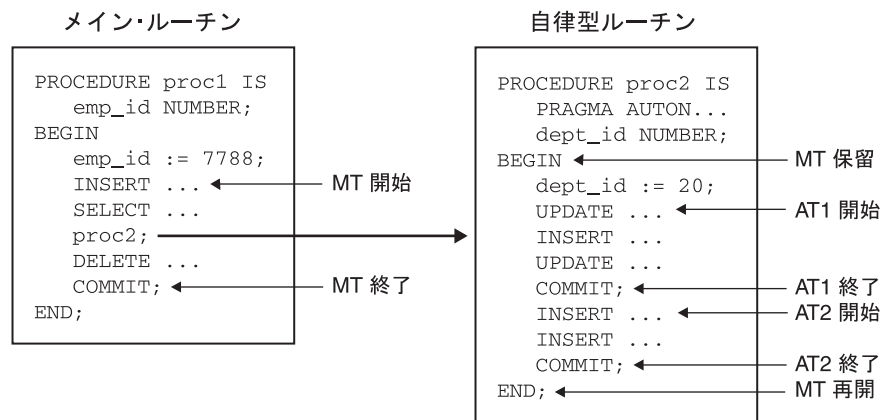
自律型ルーチンの実行部に入ると、メイン・トランザクションは停止します。ルーチンを終了すると、メイン・トランザクションは再開します。

正常に終了するには、すべての自律型トランザクションを明示的にコミットまたはロールバックする必要があります。ルーチン（またはそれによってコールされたルーチン）に保留中のトランザクションがある場合は、例外が呼び出され、保留中のトランザクションはロールバックされます。

コミットとロールバック

COMMIT と ROLLBACK はアクティブな自律型トランザクションを終了しますが、自律型ルーチンから抜けるわけではありません。図 5-4 に示すように、1 つのトランザクションが終了すると、次の SQL 文で別のトランザクションが開始されます。

図 5-4 複数の自律型トランザクション



セーブポイントの使用

セーブポイントの有効範囲は、それが定義されたトランザクションです。メイン・トランザクション内で定義されたセーブポイントは、その自律型トランザクション内で定義されたセーブポイントとは無関係です。実際、メイン・トランザクションと自律トランザクションのセーブポイントには、同じ名前を使用できます。

ロールバックできるのは、現在のトランザクション内でマークされたセーブポイントまでです。つまり、自律型トランザクション内では、メイン・トランザクション内でマークされたセーブポイントまではロールバックできません。メイン・トランザクションのセーブポイントまでロールバックするには、自律型ルーチンを抜けてメイン・トランザクションを再開する必要があります。

メイン・トランザクション内では、自律型トランザクションを開始する前にマークされたセーブポイントまでロールバックしても、自律型トランザクションはロールバックされません。自律型トランザクションは、メイン・トランザクションからは完全に独立していることに注意してください。

エラーの回避

一般的なエラーを回避するには、次の事項を守って自律型トランザクションを設計します。

- 自律型トランザクションが、メイン・トランザクション（自律型ルーチンを終了するまで再開できない）が保持するリソースにアクセスしようすると、デッドロックが発生する。
- Oracle 初期化パラメータ `TRANSACTIONS` は、同時トランザクションの最大数を指定する。メイン・トランザクションと同時に実行する自律型トランザクションを考慮に入れないと、この最大数を超過してしまう場合がある。

例 1: 自律型トリガーの使用

データベース・トリガーを使用してイベントのログを透過的に取ることができます。ある表に対するすべての挿入を、ロールバックするものも含めて、追跡したいとします。次の例では、トリガーを使って、重複する行をシャドー表に挿入します。トリガーは自律型であるため、メインの表への挿入をコミットするかどうかに関係なく、シャドー表への挿入をコミットできます。

```
-- create a main table and its shadow table
CREATE TABLE parts (pnum NUMBER(4), pname VARCHAR2(15));
CREATE TABLE parts_log (pnum NUMBER(4), pname VARCHAR2(15));

-- ceate an autonomous trigger that inserts into the
-- shadow table before each insert into the main table
CREATE TRIGGER parts_trig
BEFORE INSERT ON parts FOR EACH ROW
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO parts_log VALUES (:new.pnum, :new.pname);
    COMMIT;
END;

-- insert a row into the main table, and then commit the insert
INSERT INTO parts VALUES (1040, 'Head Gasket');
COMMIT;

-- insert another row, but then roll back the insert
INSERT INTO parts VALUES (2075, 'Oil Pan');
ROLLBACK;
```

```

-- show that only committed inserts add rows to the main table
SELECT * FROM parts ORDER BY pnum;
      PNUM PNAME
-----
      1040 Head Gasket

-- show that both committed and rolled-back inserts add rows
-- to the shadow table
SELECT * FROM parts_log ORDER BY pnum;
      PNUM PNAME
-----
      1040 Head Gasket
      2075 Oil Pan

```

例 2: 自律型ファンクションの SQL からのコール

SQL 文からコールされるファンクションは、副次作用を制御するための特定の規則に従う必要があります。(7-7 ページの「[副作用の制御](#)」を参照。) この規則に違反していないか確認するには、プラグマ `RESTRICT_REFERENCES` を使用できます。これはデータベース表またはパッケージ変数、あるいはその両方への読取りや書込みを報告するよう、コンパイラに指示します。(『Oracle8i アプリケーション開発者ガイド 基礎編』を参照。)

しかし、すべての自律型ルーチンにはデータベースへの読取り / 書込みアクセスがあります。このため、自律型ルーチンが " データベース読込み禁止状態 " および " データベース書込み禁止状態 " の規則に違反することはありません。次の例に示すように、これは便利な機能です。問合せからパッケージ・ファンクション `log_msg` をコールすると、" データベース書込み禁止状態 " の規則に違反することなく、データベース表 `debug_output` にメッセージが挿入されます。

```

-- create the debug table
CREATE TABLE debug_output (msg VARCHAR2(200));

-- create the package spec
CREATE PACKAGE debugging AS
  FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2;
  PRAGMA RESTRICT_REFERENCES(log_msg, WNDS, RNDS);
END debugging;

-- create the package body
CREATE PACKAGE BODY debugging AS
  FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2 IS
    PRAGMA AUTONOMOUS_TRANSACTION;
  BEGIN
    -- the following insert does not violate the constraint
    -- WNDS because this is an autonomous routine

```

```
        INSERT INTO debug_output VALUES (msg);
        COMMIT;
        RETURN msg;
    END;
END debugging;

-- call the packaged function from a query
DECLARE
    my_empno NUMBER(4);
    my_ename VARCHAR2(15);
BEGIN
    ...
    SELECT debugging.log_msg(ename) INTO my_ename FROM emp
        WHERE empno = my_empno;
    -- even if you roll back in this scope, the insert
    -- into 'debug_output' remains committed because
    -- it is part of an autonomous transaction
    IF ... THEN
        ROLLBACK;
    END IF;
END;
```

パフォーマンスの向上

このセクションでは、パフォーマンスを向上させるいくつかの方法と、それらをアプリケーションで使用方法を説明します。

オブジェクト型およびコレクションの使用

コレクション型（第4章を参照）およびオブジェクト型（第9章を参照）を使うと、現実のデータをモデル化できるので、生産性が上がります。複雑な実世界のエンティティと関連は、オブジェクト型に直接対応付けることができます。また、うまく作成されたオブジェクト・モデルは、表の結合をなくし、ネットワークの往復を減らすなど、アプリケーションのパフォーマンスを向上させます。

PL/SQL プログラムなどのクライアント・プログラムでは、オブジェクトおよびコレクションを宣言したり、パラメータとして渡したり、データベースに格納したり、取り出したりできます。また、オブジェクト型を使うと、データに対する操作をカプセル化することによって、データ・メンテナンスのためのコードを SQL スクリプトの外に出し、PL/SQL ブロックをメソッドに入れることができます。

オブジェクトおよびコレクションは、1つのかたまりとして操作できるので、格納や取出しの効率が良くなります。また、オブジェクト・サポートは、データベースと構造的に統合されているので、Oracle8i に組み込まれている拡張性およびパフォーマンス上の多くの改善点を利用できます。

バルク・バインドの使用

コレクション要素をバインド変数として使用するループ内で SQL 文を実行する場合、PL/SQL エンジンと SQL エンジンとの間のコンテキスト切替えによって、実行速度が遅くなる場合があります。たとえば、次の UPDATE 文は、FOR ループの反復ごとに SQL エンジンに送信されます。

```
DECLARE
  TYPE NumList IS VARRAY(20) OF NUMBER;
  depts NumList := NumList(10, 30, 70, ...); -- department numbers
BEGIN
  ...
  FOR i IN depts.FIRST..depts.LAST LOOP
    ...
    UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(i);
  END LOOP;
END;
```

この場合、SQL 文が 5 つ以上のデータベース行に影響するのであれば、バルク・バインドを使用するとパフォーマンスが向上します。たとえば、次の UPDATE 文は、ネストした表全体とともに、一度だけ SQL エンジンに送信されます。

```
FORALL i IN depts.FIRST..depts.LAST
  UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(i);
```

パフォーマンスを最適化するには、次のようにプログラムを再作成します。

- INSERT 文、UPDATE 文または DELETE 文がループ内で実行され、コレクション要素を参照する場合は、これを FORALL 文に移動する。
- SELECT INTO 句、FETCH INTO 句または RETURNING INTO 句がコレクション要素を参照する場合は、BULK COLLECT 句を組み込む。
- 可能であれば、ホスト配列を使用して、プログラムとデータベース・サーバーとの間でコレクションをやりとりする。

注意：これらは簡単な作業ではありません。これを行うにはプログラム制御フローと依存性の慎重な分析が必要です。一括変換の詳細は、4-27 ページの「[バルク・バインドの利用](#)」を参照してください。

システム固有の動的 SQL の使用

たとえば汎用目的の報告書作成プログラムなど、プログラムによっては、さまざまな SQL 文を実行時に構築および処理する必要があります。このため、このようなプログラムのフル・テキストは、その時までわかりません。多くの場合、このような文は実行ごとに変わります。そのため、このような文は動的 SQL 文と呼ばれます。

以前は、動的 SQL 文を実行するには、提供されているパッケージ DBMS_SQL を使用する必要がありました。現在、PL/SQL 内では、どの種類の動的 SQL 文でも、システム固有の動的 SQL と呼ばれるインタフェースを使用して実行できます。

システム固有の動的 SQL は、パッケージ DBMS_SQL より使いやすく、処理速度も高速です。次の例では、カーソル変数を宣言し、それをデータベース表 emp から行を戻す動的 SELECT 文と関連付けます。

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    emp_cv    EmpCurTyp;
    my_ename  VARCHAR2(15);
    my_sal    NUMBER := 1000;
BEGIN
    OPEN emp_cv FOR
        'SELECT ename, sal FROM emp
         WHERE sal > :s' USING my_sal;
    ...
END;
```

詳細は、[第 10 章の「システム固有の動的 SQL」](#)を参照してください。

外部ルーチンの使用

PL/SQL は、SQL トランザクション処理に特化されています。そのため、作業の中には、マシン精度の計算で効率のよい C などの低レベルの言語で実行するとさらに高速に処理できるものがあります。

PL/SQL には他の言語で書かれたルーチンをコールするためのインタフェースが用意されているので、PL/SQL を使うことにより、Oracle Server の機能を拡張できます。他の言語ですでに作成され利用できるようなっている標準ライブラリを PL/SQL プログラムからコールできます。それによって、再利用性、効率、モジュール性が高まります。

実行を高速化するには、計算専用プログラムを C で再作成できます。さらに、そのようなプログラムをクライアントからサーバーに移動できます。サーバーの方が計算能力が高く、ネットワーク上の通信が少ないので、プログラムをより高速に実行できます。

たとえば、イメージ・オブジェクト型のメソッドを C で作成して動的リンク・ライブラリ (DLL) に格納し、そのライブラリを PL/SQL に登録して、アプリケーションからコールできます。ライブラリは実行時に動的にロードされ、安全保護のため、(分離したプロセスとしてインプリメントされた) 別々のアドレス空間で実行されます。

詳細は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

NOCOPY コンパイラ・ヒントの使用

デフォルトでは、OUT パラメータと IN OUT パラメータは値によって渡されます。つまり、IN OUT 実パラメータの値は、対応する仮パラメータにコピーされます。サブプログラムが正常に終了すると、OUT および IN OUT 仮パラメータに代入された値は、対応する実パラメータにコピーされます。

パラメータが、コレクション、レコードおよびオブジェクト型のインスタンスなどの大きなデータ構造を保持している場合、このコピー作業によって実行速度が遅くなり、メモリーが消費されます。これを回避するには、NOCOPY ヒントを指定します。これによって、PL/SQL コンパイラは OUT および IN OUT パラメータを参照によって渡すことができます。次の例では、IN OUT パラメータ my_unit を、値ではなく参照によって渡すよう、コンパイラに指示します。

```
DECLARE
    TYPE Platoon IS VARRAY(200) OF Soldier;
    PROCEDURE reorganize (my_unit IN OUT NOCOPY Platoon) IS ...
BEGIN
    ...
END;
```

詳細は、7-17 ページの「[NOCOPY コンパイラ・ヒント](#)」を参照してください。

RETURNING 句の使用

レポートを生成する場合や後続のアクションをとる場合などに、アプリケーションでは、SQL 操作の影響が及ぶ行の情報が必要になることがよくあります。INSERT 文、UPDATE 文、DELETE 文では、RETURNING 句を使用できます。RETURNING 句は、影響が及ぶ行の列値を戻して、PL/SQL 変数またはホスト変数に入れます。これにより、挿入や更新の後、または削除の前に、行を SELECT で選択する必要がなくなります。その結果、ネットワークの往復、サーバー CPU タイム、カーソル、およびサーバー・メモリーが少なくて済みます。

次の例では、従業員の給与を更新し、同時に従業員の名前と新しい給与を取り出して PL/SQL 変数に入れます。

```
PROCEDURE update_salary (emp_id NUMBER) IS
    name    VARCHAR2(15);
    new_sal NUMBER;
BEGIN
    UPDATE emp SET sal = sal * 1.1
    WHERE empno = emp_id
    RETURNING ename, sal INTO name, new_sal;
```

逐次再使用可能パッケージの使用

メモリーの使用を管理しやすくするために、PL/SQL ではプラグマ `SERIALLY_REUSABLE` を用意しています。これを使用すると、いくつかのパッケージを逐次再使用可能としてマークできます。サーバーへの 1 コール（たとえば、サーバーへの OCI コールやサーバー間の RPC）の間だけパッケージ状態が必要な場合に、パッケージをこのようにマークできます。

このようなパッケージのグローバル・メモリーは、ユーザー・グローバル領域（UGA）で個々のユーザーに割り当てられるのではなく、システム・グローバル領域（SGA）にプールされます。それによって、パッケージ作業域の再使用が可能になります。サーバーへのコールが終わると、メモリーはプールに戻されます。パッケージが再使用されるたびに、そのパッケージのパブリック変数はデフォルト値か `NULL` に初期設定されます。

1 つのパッケージに必要な作業域の最大数は、そのパッケージを同時に使用するユーザーの数です。通常は、ログオン・ユーザーの数よりもかなり少ない数になります。SGA メモリーの使用量が増えた場合、UGA メモリーの使用量を減らしても埋め合わせはできません。また、Oracle では、SGA メモリーの再生が必要になると、使用されていない古い作業域が破棄されます。

本体のないパッケージの場合は、次の構文を使って、このプラグマをパッケージの仕様部に作成します。

```
PRAGMA SERIALLY_REUSABLE;
```

本体のあるパッケージの場合は、このプラグマを仕様部と本体の両方に作成する必要があります。このプラグマを本体にのみ作成することはできません。次の例で、逐次再使用可能パッケージのパブリック変数がコール境界を超えて作用する様子を示します。

```
CREATE OR REPLACE PACKAGE sr_pkg IS
    PRAGMA SERIALLY_REUSABLE;
    num NUMBER := 0;
    PROCEDURE init_pkg_state(n NUMBER);
    PROCEDURE print_pkg_state;
END sr_pkg;
/

CREATE OR REPLACE PACKAGE BODY sr_pkg IS
    PRAGMA SERIALLY_REUSABLE;
    /* Initialize package state. */
    PROCEDURE init_pkg_state (n NUMBER) IS
    BEGIN
        sr_pkg.num := n;
    END;
    /* Print package state. */
    PROCEDURE print_pkg_state IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Num is: ' || sr_pkg.num);
    END;
END sr_pkg;
/
```



```

BEGIN
    /* Initialize package state. */
    sr_pkg.init_pkg_state(4);
    /* On same server call, print package state. */
    sr_pkg.print_pkg_state; -- prints 4
END;
/
-- subsequent server call
BEGIN
    -- package's public variable will be initialized to its
    -- default value automatically
    sr_pkg.print_pkg_state; -- prints 0
END;

```

詳細は、『Oracle8i パッケージ・プロシージャ リファレンス』を参照してください。

PLS_INTEGER データ型の使用

整変数を宣言する必要がある場合は、PLS_INTEGER データ型を使います。PLS_INTEGER は、最も効率的な数値型です。それは、PLS_INTEGER 値では、内部的に 22 バイトの Oracle の数値で表される INTEGER 値や NUMBER 値に比べ、あまり記憶域を必要としないからです。また、PLS_INTEGER 演算はマシン算術計算を使うため、ライブラリ算術計算を使う BINARY_INTEGER、INTEGER、または NUMBER 演算よりも処理速度が速くなります。

さらに、INTEGER、NATURAL、NATURALN、POSITIVE、POSITIVEN、および SIGNTYPE は制約付きのサブタイプです。したがって、これらの型の変数は実行時に精度検査を必要とし、それがパフォーマンスに影響することがあります。

NOT NULL 制約の回避

PL/SQL では、NOT NULL 制約を使うと、パフォーマンス・コストがかかります。次の例を考えてみます。

```

PROCEDURE calc_m IS
    m NUMBER NOT NULL;
    a NUMBER;
    b NUMBER;
BEGIN
    ...
    m := a + b;

```

m には NOT NULL の制約があるので、式 $a + b$ の値は一時変数に代入されて、NULL かどうかの検査を受けます。変数が NULL でなければ、その値が m に代入されます。変数が NULL の場合は、例外が発生します。しかし、m に制約がなければ、値は直接 m に代入されます。

上の例をより効率的に作成する方法を次に示します。

```
PROCEDURE calc_m IS
  m NUMBER; -- no constraint
  a NUMBER;
  b NUMBER;
BEGIN
  ...
  m := a + b;
  IF m IS NULL THEN ... -- enforce constraint programmatically
END;
```

サブタイプ `NATURALN` と `POSITIVEN` は、`NOT NULL` と定義されています。したがって、それらを使うと、同じパフォーマンス・コストがかかります。

条件制御文の句の入替え

論理式を評価するとき、PL/SQL では短絡評価を使います。これにより、PL/SQL は結果が判別できた時点でただちに式の評価を停止します。たとえば、次の `OR` 式では、`sal` の値が 1500 より少ない場合、左のオペランドの結果が `TRUE` になるので、PL/SQL は右のオペランドを評価する必要はありません（いずれかのオペランドが `TRUE` であれば `OR` は `TRUE` を戻すため）。

```
IF (sal < 1500) OR (comm IS NULL) THEN
  ...
END IF;
```

次の `AND` 式で考えてみます。

```
IF credit_ok(cust_id) AND (loan < 5000) THEN
  ...
END IF;
```

ブール・ファンクション `credit_ok` が常にコールされます。しかし、次のように `AND` のオペランドを入れ替えたとします。

```
IF (loan < 5000) AND credit_ok(cust_id) THEN
  ...
END IF;
```

この場合、ファンクションは式 `loan < 5000` が `TRUE` の場合にだけコールされます（`AND` は、オペランドが両方とも `TRUE` の場合にだけ `TRUE` を戻すため）。

`EXIT-WHEN` 文も同じ考え方です。

暗黙のデータ型変換の回避

PL/SQL は実行時に、構造の異なるデータ型を暗黙的に変換します。たとえば、PLS_INTEGER 変数を NUMBER 変数に代入すると、両者の内部表現は異なるため、変換が実行されます。

暗黙の変換を回避することで、パフォーマンスが向上します。次の例を見てください。整数リテラル 15 は、内部的に符号付きの 4 バイトの数量で表されるので、加算する前に、PL/SQL で Oracle の数値に変換する必要があります。しかし、浮動小数点リテラル 15.0 は 22 バイトの Oracle の数値で表されるので、変換は必要ありません。

```
DECLARE
  n NUMBER;
  c CHAR(5);
BEGIN
  n := n + 15;    -- converted
  n := n + 15.0; -- not converted
```

次にもう 1 つ例を示します。

```
DECLARE
  c CHAR(5);
BEGIN
  c := 25;    -- converted
  c := '25'; -- not converted
```

下位互換性の保証

PL/SQL バージョン 2 では、バージョン 8 で禁止されている正常でない動作のいくつかが認められています。具体的には、バージョン 2 を使うと、次の処理が行えるようになります。

- 変数を宣言するときに、RECORD 型と TABLE 型の前参照が行える。
- ファンクション仕様部の RETURN 句に変数（データ型ではなく）の名前を指定できる。
- 索引付き表の IN パラメータの要素に値を代入できる。
- レコードの IN パラメータのフィールドを OUT パラメータとして別のサブプログラムに渡すことができる。
- 代入文の右側にあるレコードの OUT パラメータのフィールドを使用できる。
- SELECT 文の FROM リスト内の OUT パラメータを使用できる。

下位互換性に備えて、バージョン 2 のこの特定の動作を保持することができます。これは、PLSQL_V2_COMPATIBILITY フラグを設定することによって行えます。サーバー側では、次の 2 つの方法でこのフラグを設定できます。

- 次の行を Oracle 初期化ファイルに追加する。

```
PLSQL_V2_COMPATIBILITY=TRUE
```

- 次のいずれかの SQL 文を実行する。

```
ALTER SESSION SET PLSQL_V2_COMPATIBILITY = TRUE;  
ALTER SYSTEM SET PLSQL_V2_COMPATIBILITY = TRUE;
```

FALSE (デフォルト) を指定した場合には、バージョン 8 の動作しか認められません。

クライアント側では、コマンド行オプションによってこのフラグを設定します。たとえば、Oracle プリコンパイラ環境では、次のようにランタイム・オプション `DBMS` をコマンド行に指定します。

```
... DBMS=V7 ...
```

There is nothing more exhilarating than to be shot at without result.—Winston Churchill

実行時エラーは、設計の失敗、コーディングの間違い、ハードウェアの障害など、多くの原因で発生します。発生する可能性があるエラーをすべては予想できませんが、ユーザーの PL/SQL プログラムにとって重大なエラーに対しては、処理を準備しておくことはできません。

プログラミング言語では、通常、エラー・チェックを使用禁止にしていない限り、「スタック・オーバーフロー (*stack overflow*)」や「ゼロによる除算 (*division by zero*)」のような実行時エラーがあると、正常な処理が停止され、オペレーティング・システムに制御が戻ります。PL/SQL には「例外処理」というしくみがあり、エラーが発生しても処理を続けられるように、プログラムを保護しています。

主なトピック

[概要](#)

[例外の利点](#)

[事前定義の例外](#)

[ユーザー定義の例外](#)

[例外の呼び出し方](#)

[例外の遷移](#)

[例外の再呼出し](#)

[呼び出された例外の処理](#)

[便利なテクニック](#)

概要

PL/SQL では、警告またはエラー条件のことを例外と呼びます。例外には、（実行時システムによって）内部的に定義された例外と、ユーザーが定義した例外があります。一般的な内部例外の中には、「ゼロによる除算（*division by zero*）」や「メモリーの不足（*out of memory*）」などがあります。内部的に定義された例外には、`ZERO_DIVIDE` や `STORAGE_ERROR` といった事前定義の名前を持つものもあります。それ以外の内部例外にも名前を付けることができます。

PL/SQL ブロック、サブプログラムまたはパッケージの宣言部で、ユーザー独自の例外を定義できます。たとえば、残高がマイナスになっている銀行口座にフラグを付けるために、`insufficient_funds` という名前の例外を定義できます。内部例外とは異なり、ユーザー定義の例外には必ず名前を付けなければなりません。

エラーが発生すると例外が呼び出されます。つまり、通常の実行は中止され、PL/SQL ブロックまたはサブプログラムの例外処理部に制御が移ります。内部例外は実行時システムによって暗黙的（自動的）に呼び出されます。ユーザー定義の例外は `RAISE` 文によって明示的に呼び出さなければなりません（`RAISE` 文も事前定義の例外を呼び出します）。

呼び出された例外を処理するには、例外ハンドラと呼ばれる独立したルーチンを作ります。例外ハンドラが実行されると、現在のブロックの実行を中止し、外側のブロックの次の文から再開します。外側にブロックがなければ、制御はホスト環境に戻ります。

次の例では、ティッカー・シンボル XYZ の企業について、株価収益率を計算し、格納しています。企業の収益がゼロの場合は、事前定義の例外 `ZERO_DIVIDE` が呼び出されます。このとき、ブロックの通常の実行は中止され、制御が例外ハンドラに移ります。ブロックで特に名前を指定していないすべての例外は、オプションの `OTHERS` ハンドラで処理します。

```
DECLARE
    pe_ratio NUMBER(3,1);
BEGIN
    SELECT price / earnings INTO pe_ratio FROM stocks
        WHERE symbol = 'XYZ'; -- might cause division-by-zero error
    INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
    COMMIT;
EXCEPTION -- exception handlers begin
    WHEN ZERO_DIVIDE THEN -- handles 'division by zero' error
        INSERT INTO stats (symbol, ratio) VALUES ('XYZ', NULL);
        COMMIT;
    ...
    WHEN OTHERS THEN -- handles all other errors
        ROLLBACK;
END; -- exception handlers and block end here
```

上の例は例外処理の様子を示すためのもので、INSERT 文の使用方法としては効率的とはいえません。挿入する場合は、次のようにするとよいでしょう。

```
INSERT INTO stats (symbol, ratio)
  SELECT symbol, DECODE(earnings, 0, NULL, price / earnings)
  FROM stocks WHERE symbol = 'XYZ';
```

この例では、副問合せで INSERT 文に値を与えています。収益がゼロならば、ファンクション DECODE は NULL を戻します。ゼロでなければ、DECODE は株価収益率を戻します。

例外の利点

エラー処理に例外を使うと、次のような利点があります。例外処理がなければ、コマンドを発行するたびに実行エラーを検査しなければなりません。

```
BEGIN
  SELECT ...
    -- check for 'no data found' error
  SELECT ...
    -- check for 'no data found' error
  SELECT ...
    -- check for 'no data found' error
```

エラー処理は通常の処理から明確に分離されておらず、安全性が高いともいえません。検査コードを作っておかなければエラーは検出されず、一見して無関係なエラーが別に発生する可能性が高くなります。

例外を利用すると、複数の検査コードを作らなくても、次の例のようにエラーを処理できます。

```
BEGIN
  SELECT ...
  SELECT ...
  SELECT ...
  ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN -- catches all 'no data found' errors
```

例外を利用すると、エラー処理ルーチンが分離され、わかりやすくなります。エラー回復アルゴリズムのために主アルゴリズムが理解しにくくなることもありません。例外には信頼性を向上させるという効果もあります。エラーが発生する可能性のある場所で、いちいちエラーを検査する必要はありません。PL/SQL ブロックに例外ハンドラを追加するだけです。そうすると、そのブロック（またはサブブロック）で例外が呼び出されたときにその例外を確実に処理できます。

事前定義の例外

PL/SQL プログラムが Oracle の規則に違反するか、そのシステムの制限を超えると、暗黙的に内部例外が呼び出されます。すべての Oracle エラーは番号を持っていますが、例外は名前によって処理しなければなりません。そこで、PL/SQL では、いくつかの一般的な Oracle エラーが例外として事前定義されています。たとえば、SELECT INTO 文が行を返さなかった場合は、事前定義の例外 NO_DATA_FOUND が PL/SQL により呼び出されます。

その他の Oracle エラーを処理するには OTHERS ハンドラを使います。Oracle エラー・コードとメッセージ・テキストを戻すファンクション SQLCODE および SQLERRM は、特に OTHERS ハンドラで使うと便利です。あるいは EXCEPTION_INIT プラグマを使って、例外名を Oracle エラー・コードに結び付けることもできます。

PL/SQL は、PL/SQL 環境を定義するパッケージ STANDARD の中で、事前定義の例外をグローバルに宣言します。ユーザーが宣言する必要はありません。次の表に示す名前を使えば、事前定義の例外を処理するハンドラを作成できます。この表では、対応する Oracle エラー・コードと SQLCODE の戻り値も示しています。

例外	Oracle エラー	SQLCODE 値
ACCESS_INTO_NULL	ORA-06530	-6530
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

次に事前定義の例外を簡単に説明します。

例外	呼び出される場合
ACCESS_INTO_NULL	プログラムが未初期化（アトミック NULL）オブジェクトの属性に値を代入しようとしたとき。
COLLECTION_IS_NULL	プログラムが EXISTS 以外のコレクション・メソッドを未初期化（アトミック NULL）のネストした表または varray に適用しようとしたか、または未初期化のネストした表または varray の要素に値を代入しようとしたとき。
CURSOR_ALREADY_OPEN	すでにオープンされているカーソルをオープンしようとしたとき。カーソルをオープンするには、一度クローズしなければならない。 カーソル FOR ループは、参照するカーソルを自動的にオープンする。このため、ループの内側ではカーソルをオープンできない。
DUP_VAL_ON_INDEX	UNIQUE 索引によって制約されているデータベース列に、重複した値を格納しようとしたとき。
INVALID_CURSOR	オープンされていないカーソルをクローズするなど、不正なカーソル操作を実行しようとしたとき。
INVALID_NUMBER	SQL 文の中で、文字列が正しい数値を表していなかったために、文字列から数値への変換が失敗したとき。（プロシージャ文では、VALUE_ERROR が呼び出される。）
LOGIN_DENIED	不正なユーザー名 / パスワードで Oracle にログインしようとしたとき。
NO_DATA_FOUND	SELECT INTO 文が行を戻さなかったとき、ネストした表で削除された要素を参照したとき、または索引付き表で未初期化の要素を参照したとき。 AVG や SUM などの SQL 集計関数は必ず値または NULL を返す。したがって、集計関数をコールする SELECT INTO 文では、NO_DATA_FOUND が呼び出されることはない。 FETCH 文は最終的には行を戻さなくなると予想されるが、その場合は、例外は呼び出されない。
NOT_LOGGED_ON	Oracle に接続していないプログラムが、データベース・コールを発行した場合。
PROGRAM_ERROR	PL/SQL に内部的な問題点が発生した場合。

例外	呼び出される場合
ROWTYPE_MISMATCH	1 つの代入の中に含まれるホスト・カーソル変数と PL/SQL カーソル変数の戻り型に互換性がない場合。たとえば、オープン・ホスト・カーソル変数をストアド・サブプログラムに渡すとき、実パラメータの戻り型と仮パラメータの戻り型には互換性がなければならない。
SELF_IS_NULL	NULL インスタンスで MEMBER メソッドをコールしようとしたとき。つまり、組込みパラメータ SELF が NULL である場合。このパラメータは、常に MEMBER メソッドに最初に渡されるパラメータである。
STORAGE_ERROR	PL/SQL のメモリーが足りない場合、またはメモリーが破壊されている場合。
SUBSCRIPT_BEYOND_COUNT	コレクション中の要素数より大きい索引番号を使ってネストした表または varray の要素を参照した場合。
SUBSCRIPT_OUTSIDE_LIMIT	有効範囲外（たとえば -1）の索引番号を使ってネストした表または varray の要素を参照した場合。
SYS_INVALID_ROWID	文字列が正しい ROWID を表していなかったために、文字列から汎用 ROWID への変換が失敗した場合。
TIMEOUT_ON_RESOURCE	Oracle がリソースを求めて待機しているときにタイムアウトが発生した場合。
TOO_MANY_ROWS	SELECT INTO 文が複数の行を戻した場合。
VALUE_ERROR	算術エラー、変換エラー、切捨てエラー、またはサイズ制約エラーが発生した場合。たとえば、列値を選択し文字変数に代入するときに、その値が変数の宣言された長さよりも長い場合、PL/SQL はその割当てを異常終了させて VALUE_ERROR を呼び出す。 プロシージャ文では、文字列から数値への変換が失敗した場合に VALUE_ERROR が呼び出される。（SQL 文では、INVALID_NUMBER が呼び出される。）
ZERO_DIVIDE	数値をゼロで割ろうとしたとき。

ユーザー定義の例外

PL/SQL ではユーザー独自の例外を定義できます。事前定義の例外とは異なり、ユーザー定義の例外は、宣言しなければならず、RAISE 文を使って明示的に呼び出さなければなりません。

例外の宣言

例外は PL/SQL ブロック、サブプログラムまたはパッケージの宣言部でしか宣言できません。例外は、例外の名前にキーワード `EXCEPTION` を付けて宣言します。次の例では、`past_due` という名前の例外を宣言しています。

```
DECLARE
    past_due EXCEPTION;
```

例外の宣言と変数の宣言は似ています。ただし、例外はデータ項目ではなく、エラー条件であることを覚えておいてください。変数とは異なり、例外は代入文や SQL 文では使えません。ただし、変数と例外の有効範囲の規則は同じです。

有効範囲の規則

同じブロックでは 1 つの例外を 2 回宣言できません。しかし、2 つの異なるブロックであれば、同じ例外を宣言できます。

ブロックの中で宣言された例外は、そのブロックに対してローカルで、そのブロックのすべてのサブブロックに対してグローバルであるとみなされます。ブロックはローカルまたはグローバルな例外しか参照できないので、サブブロックで宣言された例外を外側のブロックから参照できません。

サブブロックでグローバルな例外を再宣言すると、ローカルの宣言が優先されます。このため、サブブロックからはグローバルな例外を参照できません。ただし、グローバルな例外がラベル付きのブロックで宣言されている場合は、次の構文を使うとグローバルな例外を参照できます。

```
block_label.exception_name
```

次の例に有効範囲の規則を示します。

```
DECLARE
    past_due EXCEPTION;
    acct_num NUMBER;
BEGIN
    DECLARE ----- sub-block begins
        past_due EXCEPTION; -- this declaration prevails
        acct_num NUMBER;
    BEGIN
        ...
```

```
        IF ... THEN
            RAISE past_due; -- this is not handled
        END IF;
    END; ----- sub-block ends
EXCEPTION
    WHEN past_due THEN -- does not handle RAISED exception
        ...
END;
```

サブブロックの `past_due` の宣言が優先されるため、外側のブロックは呼び出された例外を処理しません。この2つの例外は同じ `past_due` という名前を持っていますが、同じ名前の2つの `acct_num` 変数が別の変数であるのと同様に、別々の例外です。したがって、`RAISE` 文と `WHEN` 句は別々の例外を参照していることになります。呼び出された例外を外側のブロックで処理するためには、サブブロックから宣言を削除するか、`OTHERS` ハンドラを定義する必要があります。

EXCEPTION_INIT の使用

無名の内部例外を処理する場合は、`OTHERS` ハンドラか `EXCEPTION_INIT` プラグマを使う必要があります。プラグマとは、コンパイラに対する命令の1つで、コンパイラに挿入されたコメントとみなされます。プラグマ（疑似命令とも言う）は、実行時ではなくコンパイル時に処理されます。たとえば、Ada 言語では、次のようなプラグマでコンパイラに指示を与え、ユーザーの記憶領域の使用方法を最適化します。

```
pragma OPTIMIZE (SPACE);
```

PL/SQL では、プラグマ `EXCEPTION_INIT` でコンパイラに指示して、例外名と Oracle エラーの数値を対応付けます。この対応付けにより、内部例外を名前で参照し、専用のハンドラを作成できます。

`EXCEPTION_INIT` プラグマは、PL/SQL ブロック、サブプログラムまたはパッケージの宣言部で、次の構文を使って指定します。

```
PRAGMA EXCEPTION_INIT(exception_name, Oracle_error_number);
```

`exception_name` は事前に宣言した例外の名前です。次の例に示すとおり、プラグマは、例外宣言と同じ宣言部のうち、例外宣言より後の部分になければなりません。

```
DECLARE
    deadlock_detected EXCEPTION;
    PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
    ...
EXCEPTION
    WHEN deadlock_detected THEN
        -- handle the error
END;
```

raise_application_error の使用

Oracle が提供するパッケージ DBMS_STANDARD は、アプリケーションと Oracle とのやりとりを支援する言語機能を提供します。たとえば、プロシージャ raise_application_error を使うと、ストアド・サブプログラムからユーザー定義のエラー・メッセージを発行できます。これを利用すると、アプリケーションに対してエラーを報告し、処理されない例外が戻されるのを避けることができます。

raise_application_error を呼び出すには、次の構文を使います。

```
raise_application_error(error_number, message[, {TRUE | FALSE}]);
```

error_number は -20000 ~ -20999 の範囲内の負の整数で、message は長さが 2048 バイトまでの文字列です。オプションの 3 番目のパラメータが TRUE の場合、エラーは、以前のエラーのスタックに配置されます。そのパラメータが FALSE (デフォルト) の場合、エラーは以前のエラーをすべて置換します。パッケージ DBMS_STANDARD はパッケージ STANDARD を拡張したものであるため、その内容を参照する場合も修飾名にする必要はありません。

アプリケーションは、実行中のストアド・サブプログラム (またはメソッド) からだけ raise_application_error を呼び出せます。raise_application_error が呼び出されると、サブプログラムは終了され、ユーザー定義のエラー番号とメッセージがアプリケーションに戻されます。エラー番号とメッセージは、Oracle エラーのように検出させることができます。

次の例では、従業員の給与が見つからない場合に、raise_application_error を呼び出しています。

```
CREATE PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) AS
  curr_sal NUMBER;
BEGIN
  SELECT sal INTO curr_sal FROM emp WHERE empno = emp_id;
  IF curr_sal IS NULL THEN
    /* Issue user-defined error message. */
    raise_application_error(-20101, 'Salary is missing');
  ELSE
    UPDATE emp SET sal = curr_sal + amount WHERE empno = emp_id;
  END IF;
END raise_salary;
```

呼出し側のアプリケーションは、PL/SQL 例外を受け取り、エラー報告関数 SQLCODE および SQLERRM を使って OTHERS ハンドラで処理できます。また、プラグマ EXCEPTION_INIT を使うと、raise_application_error が戻す特定のエラー番号をアプリケーション独自の例外にマップできます。

```
EXEC SQL EXECUTE
/* Execute embedded PL/SQL block using host
   variables my_emp_id and my_amount, which were
   assigned values in the host environment. */
DECLARE
...
null_salary EXCEPTION;
/* Map error number returned by raise_application_error
   to user-defined exception. */
PRAGMA EXCEPTION_INIT(null_salary, -20101);
BEGIN
...
raise_salary(:my_emp_id, :my_amount);
EXCEPTION
WHEN null_salary THEN
INSERT INTO emp_audit VALUES (:my_emp_id, ...);
...
END;
END-EXEC;
```

この手法を使うと、呼出し側のアプリケーションは、エラーが発生している状態を特定の例外ハンドラで処理できます。

事前定義の例外の再宣言

PL/SQL は、事前定義の例外をパッケージ STANDARD でグローバルに宣言しているので、ユーザーが宣言する必要はありません。事前定義の例外を再宣言すると、ローカルな宣言がグローバルな宣言を上書きするために、エラーが発生しやすくなります。たとえば、*invalid_number* という名前の例外を宣言し、PL/SQL によって事前定義の例外 *INVALID_NUMBER* が内部的に呼び出された場合、*INVALID_NUMBER* 用に作成されたハンドラは内部例外を捕捉できません。この場合、事前定義の例外を指定するには、次のようにドット表記法を使わなければなりません。

```
EXCEPTION
WHEN invalid_number OR STANDARD.INVALID_NUMBER THEN
-- handle the error
END;
```

例外の呼び出し方

内部例外は実行時システムによって暗黙的に呼び出されます。これは、`EXCEPTION_INIT` を使って Oracle エラー番号と結び付けたユーザー定義の例外の場合も同じです。しかし、それ以外のユーザー定義の例外は、`RAISE` 文で明示的に呼び出す必要があります。

RAISE 文の使用方法

PL/SQL ブロックおよびサブプログラムから例外を呼び出すのは、エラーが原因で処理の完了が望ましくない場合または不可能な場合に限るべきです。指定した例外に対する `RAISE` 文は、その例外の有効範囲の中ならば任意の場所に置くことができます。次の例では、PL/SQL ブロックで `out_of_stock` という名前のユーザー定義の例外を指定しています。

```
DECLARE
    out_of_stock    EXCEPTION;
    number_on_hand NUMBER(4);
BEGIN
    ...
    IF number_on_hand < 1 THEN
        RAISE out_of_stock;
    END IF;
EXCEPTION
    WHEN out_of_stock THEN
        -- handle the error
END;
```

事前定義の例外を明示的に呼び出すこともできます。これを利用すると、事前定義の例外のために書かれた例外ハンドラで、それ以外のエラーを処理させることができます。次に例を示します。

```
DECLARE
    acct_type INTEGER;
BEGIN
    ...
    IF acct_type NOT IN (1, 2, 3) THEN
        RAISE INVALID_NUMBER; -- raise predefined exception
    END IF;
EXCEPTION
    WHEN INVALID_NUMBER THEN
        ROLLBACK;
    ...
END;
```

例外の遷移

例外が呼び出されたとき、PL/SQL がその例外のハンドラを現在のブロックまたはサブプログラムで発見できなければ、例外は遷移します。すなわち、例外は外側のブロックで再生され、ハンドラが見つかるまで、または検索するブロックがなくなるまで、1 つずつ外側のブロックに進んでいきます。検索するブロックがなくなった場合、PL/SQL はホスト環境に「未処理例外 (unhandled exception)」エラーを戻します。

ただし、例外はリモート・プロシージャ・コール (RPC) には遷移しません。そのため、PL/SQL ブロックは、リモート・サブプログラムによって呼び出された例外を処理できません。対策の詳細は、6-9 ページの「[raise_application_error の使用](#)」を参照してください。

図 6-1、図 6-2、および図 6-3 に、基本的な遷移規則を示します。

図 6-1 遷移規則 : 例 1

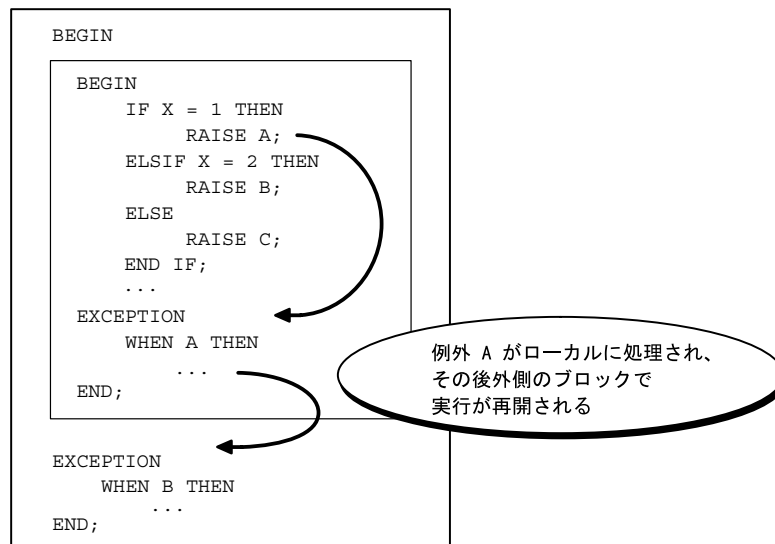


図 6-2 遷移規則 : 例 2

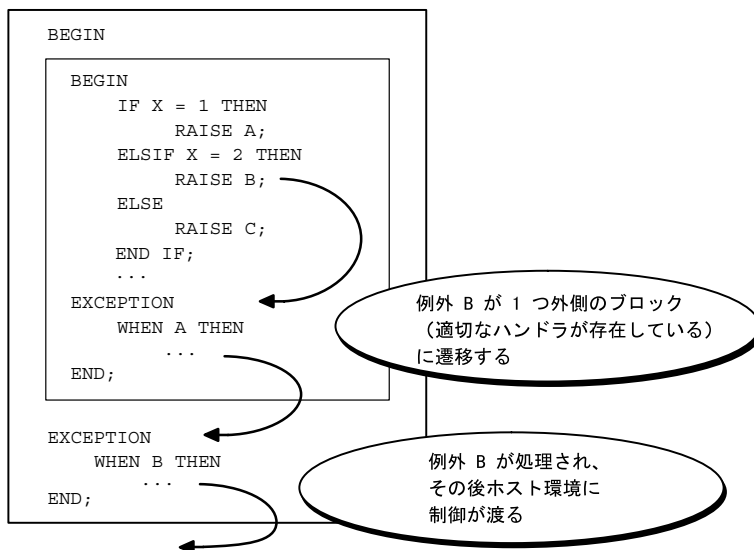
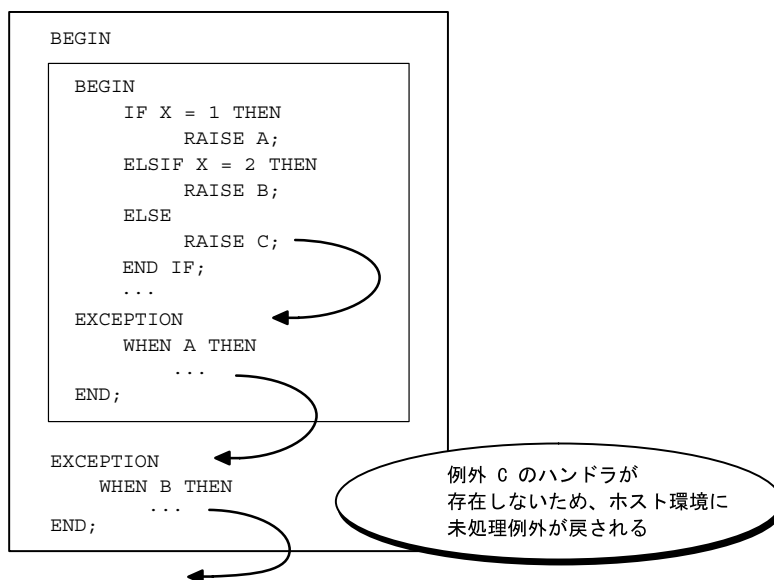


図 6-3 遷移規則 : 例 3



例外は有効範囲を超えて、つまり宣言されたブロックを超えたところまで 遷移することがあります。次の例を考えてみます。

```
BEGIN
    ...
    DECLARE ----- sub-block begins
        past_due EXCEPTION;
    BEGIN
        ...
        IF ... THEN
            RAISE past_due;
        END IF;
    END; ----- sub-block ends
EXCEPTION
    ...
    WHEN OTHERS THEN
        ROLLBACK;
END;
```

例外 `past_due` が宣言されたブロックに例外ハンドラが存在しないため、例外は外側のブロックに遷移します。しかし、有効範囲の規則によれば、外側のブロックはサブブロックで宣言された例外を参照できません。このため、この例外を捕捉できるのは `OTHERS` ハンドラに限られます。

例外の再呼出し

例外の再呼出しとは、ローカルに処理した例外を、外側のブロックに渡すことです。たとえば、現在のブロックでトランザクションをロールバックし、外側のブロックの中でエラーのログをとりたい場合があります。

例外の再呼出しをする場合は、次の例に示すようにローカルなハンドラで `RAISE` 文を使います。

```
DECLARE
    out_of_balance EXCEPTION;
BEGIN
    ...
    BEGIN ----- sub-block begins
        ...
        IF ... THEN
            RAISE out_of_balance; -- raise the exception
        END IF;
    EXCEPTION
        WHEN out_of_balance THEN
            -- handle the error
            RAISE; -- reraise the current exception
    END; ----- sub-block ends
```

```

EXCEPTION
  WHEN out_of_balance THEN
    -- handle the error differently
    ...
END;

```

RAISE 文で例外名を省略すると（これは例外ハンドラの中でしか許されていません）、現在の例外が再び呼び出されます。

呼び出された例外の処理

例外が呼び出されると、PL/SQL ブロックまたはサブプログラムの通常の実行は中止され、制御が例外処理部に移ります。例外処理部の書式を次に示します。

```

EXCEPTION
  WHEN exception_name1 THEN -- handler
    sequence_of_statements1
  WHEN exception_name2 THEN -- another handler
    sequence_of_statements2
    ...
  WHEN OTHERS THEN          -- optional handler
    sequence_of_statements3
END;

```

呼び出された例外を処理するには、例外ハンドラを作成します。個々のハンドラは、例外を指定する WHEN 句に、その例外が呼び出されたときに実行される一連の文を続けたものです。これらの文を最後に、ブロックまたはサブプログラムの実行は終わります。制御は例外が呼び出された箇所に戻りません。つまり、処理を中止した位置からは再開できません。

オプションの OTHERS 例外ハンドラは、必ずブロックまたはサブプログラムの最後のハンドラでなければなりません。OTHERS 例外ハンドラは、名前を付けなかったすべての例外のハンドラとして使われます。このため、ブロックまたはサブプログラムが持てる OTHERS ハンドラは 1 つだけです。

次の例で示すように、OTHERS ハンドラを使うと、すべての例外が処理されます。

```

EXCEPTION
  WHEN ... THEN
    -- handle the error
  WHEN ... THEN
    -- handle the error
  WHEN OTHERS THEN
    -- handle all other errors
END;

```

2 つ以上の例外で、同じ文の並びを実行したい場合は、`WHEN` 句の中でキーワード `OR` で区切って例外名を並べてください。次に例を示します。

```
EXCEPTION
  WHEN over_limit OR under_limit OR VALUE_ERROR THEN
    -- handle the error
```

リスト中の例外のいずれかが呼び出されると、それに対応する一連の文が実行されます。キーワード `OTHERS` は例外名のリストの中では使えず、単独で使わなければなりません。例外ハンドラの数に制限はなく、また、個々のハンドラは例外のリストを一連の文と結び付けることができます。ただし、1 つの例外名は PL/SQL ブロックまたはサブプログラムの例外処理部で一度しか使えません。

例外ハンドラについても、PL/SQL 変数の通常の有効範囲の規則が適用されるので、例外ハンドラの中ではローカル変数とグローバル変数が参照できます。しかし、カーソル `FOR` ループの内側で例外が呼び出されると、ハンドラに制御が移る前にカーソルは暗黙的にクローズされます。したがって、ハンドラでは明示カーソルは属性の値を参照できません。

宣言の中で呼び出された例外

宣言の中でも、初期化の式が間違っていると例外が呼び出される場合があります。たとえば、次の宣言では定数 `credit_limit` が 999 よりも大きい数値を格納できないので、例外が呼び出されます。

```
DECLARE
  credit_limit CONSTANT NUMBER(3) := 5000; -- raises an exception
BEGIN
  ...
EXCEPTION
  WHEN OTHERS THEN -- cannot catch the exception
    ...
END;
```

宣言の中で呼び出された例外は、ただちに外側のブロックに遷移するため、現在のブロックの中のハンドラは呼び出された例外を捕捉できません。

ハンドラの中で呼び出された例外

ブロックまたはサブプログラムの例外処理部の中で、有効になれる例外は一度に 1 つだけです。このため、ハンドラの内側で呼び出された例外はただちに外側のブロックに遷移し、そこで再び呼び出されて、その例外のハンドラが検索されます。それ以降の例外の遷移は通常どおりに起こります。次の例を考えてみます。

```

EXCEPTION
  WHEN INVALID_NUMBER THEN
    INSERT INTO ... -- might raise DUP_VAL_ON_INDEX
  WHEN DUP_VAL_ON_INDEX THEN ... -- cannot catch the exception
END;

```

例外ハンドラへの分岐と例外ハンドラからの分岐

GOTO 文では例外ハンドラに分岐できません。また、GOTO 文では例外ハンドラから現在のブロックに分岐できません。たとえば、次の GOTO 文は誤りです。

```

DECLARE
  pe_ratio NUMBER(3,1);
BEGIN
  DELETE FROM stats WHERE symbol = 'XYZ';
  SELECT price / NVL(earnings, 0) INTO pe_ratio FROM stocks
    WHERE symbol = 'XYZ';
  <<my_label>>
  INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    pe_ratio := 0;
    GOTO my_label; -- illegal branch into current block
END;

```

ただし、例外ハンドラから外側のブロックに分岐できます。

SQLCODE と SQLERRM の使用

例外ハンドラでは、組み込み関数 SQLCODE および SQLERRM を使って、発生したエラーを確認し、対応するエラー・メッセージを取得できます。内部例外の場合、SQLCODE は Oracle エラーの番号を戻します。SQLCODE が戻す番号は負の値ですが、Oracle エラー「データが見つからない (*no data found*)」の場合は例外で、+100 が戻されます。SQLERRM は対応するエラー・メッセージを戻します。メッセージの先頭には Oracle エラー・コードが示されています。

ユーザー定義の例外の場合、SQLCODE は +1 を戻し、SQLERRM は次のメッセージを戻します。

User-Defined Exception

ただし、EXCEPTION_INIT プラグマを使って例外名に Oracle エラー番号を結び付けた場合は、SQLCODE はエラー番号を戻し、SQLERRM は対応するエラー・メッセージを戻します。Oracle エラー・メッセージの長さは、エラー・コードおよびネストされたメッセージ、表や列の名前といったメッセージの挿入部分を入れて 512 文字までです。

これまで例外が呼び出されていない場合、SQLCODE はゼロを返し、SQLERRM は次のメッセージを返します。

```
ORA-0000: normal, successful completion
```

SQLERRM にエラー番号を渡すことができます。このとき、SQLERRM はそのエラー番号に結び付けられたメッセージを返します。SQLERRM に渡すエラー番号は負の値でなければなりません。次の例では、正の値を渡したため、予期しない結果となります。

```
DECLARE
    ...
    err_msg VARCHAR2(100);
BEGIN
    /* Get all Oracle error messages. */
    FOR err_num IN 1..9999 LOOP
        err_msg := SQLERRM(err_num); -- wrong; should be -err_num
        INSERT INTO errors VALUES (err_msg);
    END LOOP;
END;
```

SQLERRM に正の値を渡すと、必ず「ユーザー定義の例外 (*user-defined exception*)」というメッセージが返されます。+100 を渡した場合は例外で、この場合 SQLERRM は「データが見つからない (*no data found*)」というメッセージを返します。SQLERRM にゼロを渡すと、常にメッセージ「通常の正常終了 (*normal, successful completion*)」を返します。

SQLCODE または SQLERRM は、SQL 文では直接使えません。次の例に示すように、値をローカル変数に代入してから、その変数を SQL 文の中で使わなければなりません。

```
DECLARE
    err_num NUMBER;
    err_msg VARCHAR2(100);
BEGIN
    ...
EXCEPTION
    ...
    WHEN OTHERS THEN
        err_num := SQLCODE;
        err_msg := SUBSTR(SQLERRM, 1, 100);
        INSERT INTO errors VALUES (err_num, err_msg);
END;
```

文字列関数 SUBSTR を使っているため、SQLERRM の値を err_msg に代入しても、(切捨ての結果として起こる) VALUE_ERROR 例外は呼び出されません。どの内部例外が呼び出されるかを通知する関数 SQLCODE および SQLERRM は、特に OTHERS 例外ハンドラで使うと便利です。

未処理例外

呼び出された例外に対応するハンドラが発見できない場合、PL/SQL はホスト環境に「未処理例外 (unhandled exception)」エラーを戻します。その結果はホスト環境によって異なります。たとえば、Oracle プリコンパイラ環境では、失敗した SQL 文または PL/SQL ブロックがデータベースに加えた変更は、すべてロールバックされます。

未処理例外はサブプログラムにも影響を与えます。サブプログラムの実行が正常終了すると、PL/SQL は OUT パラメータに値を代入します。しかし、未処理例外が発生して実行が終了すると、PL/SQL は OUT パラメータに値を代入しません。また、ストアド・サブプログラムで未処理例外が発生して実行が失敗した場合、PL/SQL はそのサブプログラムが実行したデータベース処理をロールバックしません。

すべての PL/SQL プログラムの最上上のレベルに OTHERS ハンドラを置くと、未処理例外の発生を避けることができます。

便利なテクニック

ここでは、柔軟性が高くなる 3 つのテクニックについて説明します。

例外が呼び出された後に実行を続ける方法

例外ハンドラを使用すると、ブロックを終了する前に「致命的」なエラーから回復できます。しかしハンドラの実行が終了すると、ブロックの実行も終了します。例外ハンドラから現在のブロックに戻ることはできません。次の例で、SELECT INTO 文が ZERO_DIVIDE を呼び出した場合、INSERT 文の実行は再開できません。

```
DECLARE
    pe_ratio NUMBER(3,1);
BEGIN
    DELETE FROM stats WHERE symbol = 'XYZ';
    SELECT price / NVL(earnings, 0) INTO pe_ratio FROM stocks
        WHERE symbol = 'XYZ';
    INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        ...
END;
```

PL/SQL では、処理を続行できる例外はサポートされていませんが、1 つの文に対して例外を処理し、次の文から処理を続けることはできます。この場合、独立した例外ハンドラを持つ独立したサブブロックに文を入れます。サブブロックでエラーが発生すると、ローカルなハンドラが例外を処理します。サブブロックが終了すると、サブブロックが終わった位置から、外側のブロックが実行されます。次の例を考えてみます。

```
DECLARE
    pe_ratio NUMBER(3,1);
BEGIN
    DELETE FROM stats WHERE symbol = 'XYZ';
    BEGIN ----- sub-block begins
        SELECT price / NVL(earnings, 0) INTO pe_ratio FROM stocks
            WHERE symbol = 'XYZ';
    EXCEPTION
        WHEN ZERO_DIVIDE THEN
            pe_ratio := 0;
    END; ----- sub-block ends
    INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
EXCEPTION
    WHEN OTHERS THEN
        ...
END;
```

この例では、SELECT INTO 文が ZERO_DIVIDE 例外を呼び出すと、ローカル・ハンドラが例外を処理して pe_ratio をゼロに設定します。ハンドラの実行が終わり、サブブロックが終了すると、実行は INSERT 文から続けられます。

トランザクションの再試行

例外が呼び出された場合、トランザクションを中止するのではなく、再試行したい場合があります。ここで使うテクニックは単純なものです。まず、トランザクションをサブブロックに入れます。次に、そのサブブロックをループの中に入れ、トランザクションが繰り返して実行されるようにします。

トランザクションを開始する前にセーブポイントをマークします。トランザクションの実行に成功すると、コミットしてループを終了します。トランザクションの実行に失敗すると制御は例外ハンドラに移り、例外ハンドラはセーブポイントまでロールバックして変更をすべて取り消し、問題点を修正します。

次の例を考えてみてください。例外ハンドラが終了すると、サブブロックが終了し、制御は外側のブロックの LOOP 文に移り、サブブロックが再び実行されて、トランザクションが再試行されます。FOR ループまたは WHILE ループを使って、試行の回数を制限するとよいでしょう。

```
DECLARE
    name    VARCHAR2(20);
    ans1    VARCHAR2(3);
    ans2    VARCHAR2(3);
    ans3    VARCHAR2(3);
    suffix  NUMBER := 1;
```



```
BEGIN
...
LOOP -- could be FOR i IN 1..10 LOOP to allow ten tries
  BEGIN -- sub-block begins
    SAVEPOINT start_transaction; -- mark a savepoint
    /* Remove rows from a table of survey results. */
    DELETE FROM results WHERE answer1 = 'NO';
    /* Add a survey respondent's name and answers. */
    INSERT INTO results VALUES (name, ans1, ans2, ans3);
    -- raises DUP_VAL_ON_INDEX if two respondents
    -- have the same name
    COMMIT;
    EXIT;
  EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
      ROLLBACK TO start_transaction; -- undo changes
      suffix := suffix + 1;           -- try to fix problem
      name := name || TO_CHAR(suffix);
    END; -- sub-block ends
  END LOOP;
END;
```

ロケータ変数の使用

次の例に示すように、例外によってエラーの原因となった文がわからなくなる場合があります。

```
BEGIN
  SELECT ...
  SELECT ...
  SELECT ...
  ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN ...
    -- Which SELECT statement caused the error?
END;
```

通常は、これは問題ではありません。ただし、必要ならば、次のようにロケータ変数を使って文の実行を追跡できます。

```
DECLARE
  stmt INTEGER := 1; -- designates 1st SELECT statement
BEGIN
  SELECT ...
  stmt := 2; -- designates 2nd SELECT statement
```

```
SELECT ...  
stmt := 3; -- designates 3rd SELECT statement  
SELECT ...  
...  
EXCEPTION  
  WHEN NO_DATA_FOUND THEN  
    INSERT INTO errors VALUES ('Error in statement ' || stmt);  
END;
```

7

サブプログラム

Civilization advances by extending the number of important operations that we can perform without thinking about them.—Alfred North Whitehead

この章では、サブプログラムの使用方法を説明します。サブプログラムを使用すると、一連の文に名前を付けてカプセル化できます。サブプログラムを使って個々の操作を切り離すと、アプリケーションを開発しやすくなります。サブプログラムは基礎部品のようなもので、これを使用すると、メンテナンスしやすいモジュール構造のアプリケーションを組み立てることができます。

主なトピック

- サブプログラムについて
- サブプログラムの利点
- プロシージャ
- ファンクション
- RETURN 文
- サブプログラムの宣言
- 実パラメータと仮パラメータ
- 位置表記法と名前表記法
- パラメータのモード
- NOCOPY コンパイラ・ヒント
- パラメータのデフォルト値
- パラメータのエイリアシング
- オーバーロード
- 外部ルーチンのコール
- 実行者権限と定義者権限
- 再帰

サブプログラムについて

サブプログラムは、パラメータを取ったり起動したりできる、名前の付けられた PL/SQL ブロックです。PL/SQL にはプロシージャとファンクションの 2 種類のサブプログラムがあります。一般に、プロシージャはアクションを実行するために使い、ファンクションは値を計算するために使います。

名前を持たない無名 PL/SQL ブロックと同様に、サブプログラムには宣言部と実行部、およびオプションの例外処理部があります。宣言部には、型およびカーソル、定数、変数、例外と、ネストされたサブプログラムが含まれます。これらの項目はローカルで、サブプログラムを終了すると消去されます。実行部には、値の代入、実行の制御および Oracle データの操作を実行する文があります。例外処理部には、実行の途中で呼び出された例外を処理する例外ハンドラを入れます。

次に示す `debit_account` という名前のプロシージャは、銀行口座から出金します。

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

このプロシージャが起動またはコールされるときには、口座番号と出金の金額を受け取ります。口座番号はデータベース表 `accts` から口座残高を取り出すために使います。その後、出金の金額を使って新しい残高を計算します。新しい残高がゼロ未満の場合は例外を呼び出し、ゼロ以上の場合は銀行口座を更新します。

サブプログラムの利点

サブプログラムは拡張性をもたらします。つまり、ユーザーのニーズに合わせて PL/SQL 言語を拡張できるようになります。たとえば、新しい部門を作るプロシージャが必要な場合は、次のようなプロシージャを簡単に作成することができます。

```
PROCEDURE create_dept (new_dname VARCHAR2, new_loc VARCHAR2) IS
BEGIN
    INSERT INTO dept
        VALUES (deptno_seq.NEXTVAL, new_dname, new_loc);
END create_dept;
```

さらに、サブプログラムはモジュール性を実現します。つまり、プログラムを、管理が容易で、定義のしっかりした論理モジュールに分けられます。この特性を利用すると、トップダウン設計と「ステップごとの分割」のアプローチによって問題を解決できます。

また、サブプログラムは、再利用性とメンテナンス性を向上させます。検証の済んだサブプログラムは、いくつものアプリケーションで安心して使えます。定義が変わった場合でも、影響の範囲をサブプログラムだけに抑えられます。このため、メンテナンスや機能拡張が簡単に実施できます。

最後に、サブプログラムは抽象性を実現し、ユーザーを個々の詳細から精神的に解放します。サブプログラムを使用するには、サブプログラムがどのように働くかではなく、何をするかを知る必要があります。そうすれば、処理の詳細にとらわれることなく、トップダウン手法でアプリケーションを設計できます。ダミーのサブプログラム（スタブ）を使うと、メイン・プログラムのテストとデバッグが終了するまで、プロシージャまたはファンクションの定義をしないで済ませることができます。

プロシージャ

プロシージャとは、特定のアクションを実行するサブプログラムのことです。プロシージャは次の構文で作ります。

```
PROCEDURE name [(parameter[, parameter, ...])] IS
    [local declarations]
BEGIN
    executable statements
[EXCEPTION
    exception handlers]
END [name];
```

`parameter` の構文を次に示します。

```
parameter_name [IN | OUT [NOCOPY] | IN OUT [NOCOPY]] datatype_name
    [{:= | DEFAULT} expression]
```

パラメータのデータ型に、NOT NULL など制約を課することはできません。たとえば、次に示す acct_id の宣言は、データ型 CHAR にサイズの制約があるので誤りです。

```
PROCEDURE reconcile (acct_id CHAR(5)) IS ... -- illegal
```

ただし、次の代替方法を使って、パラメータ型に間接的にサイズ制約を課することができます。

```
DECLARE
    temp CHAR(5);
    SUBTYPE Char5 IS temp%TYPE;
    PROCEDURE reconcile (acct_id Char5) IS ...
```

プロシージャには、仕様部と本体の 2 つの部分があります。プロシージャの仕様部は、キーワード PROCEDURE で始め、プロシージャ名またはパラメータ・リストで終わります。パラメータ宣言はオプションです。パラメータを取らないプロシージャではカッコを書きません。

プロシージャの本体は、キーワード IS で始め、キーワード END で終わります。END の後には、オプションとしてプロシージャ名を続けることができます。プロシージャの本体には、宣言部、実行部、例外処理部（オプション）という 3 つの部分があります。

宣言部では、キーワード IS と BEGIN の間にローカル宣言を置きます。無名 PL/SQL ブロックでの宣言を指定するキーワード DECLARE は使いません。実行部では、キーワード BEGIN と EXCEPTION（または END）の間に文を置きます。プロシージャの実行部には、少なくとも 1 つの文が存在しなければなりません。NULL 文はこの条件を満たします。例外処理部では、キーワード EXCEPTION と END の間に例外ハンドラを置きます。

次に示すのは、従業員の給与を増やすプロシージャ raise_salary です。

```
PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS
    current_salary REAL;
    salary_missing EXCEPTION;
BEGIN
    SELECT sal INTO current_salary FROM emp
        WHERE empno = emp_id;
    IF current_salary IS NULL THEN
        RAISE salary_missing;
    ELSE
        UPDATE emp SET sal = sal + amount
            WHERE empno = emp_id;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        INSERT INTO emp_audit VALUES (emp_id, 'No such number');
    WHEN salary_missing THEN
        INSERT INTO emp_audit VALUES (emp_id, 'Salary is null');
END raise_salary;
```

このプロシージャは、コールされるときに従業員番号と給与の増額を受け取ります。従業員番号はデータベース表 `emp` から現在の給与を取り出すために使います。従業員番号が見つからないか、現在の給与が `NULL` の場合は、例外を呼び出します。それ以外の場合は、給与を更新します。

プロシージャは PL/SQL 文としてコールされます。たとえば、プロシージャ `raise_salary` は次のようにしてコールできます。

```
DECLARE
    emp_id NUMBER;
    amount REAL;
BEGIN
    ...
    raise_salary(emp_id, amount);
```

ファンクション

ファンクションとは、値を計算するサブプログラムのことです。ファンクションとプロシージャは同じような構造を持ちますが、ファンクションの方は `RETURN` 句を持っています。次にファンクションの構文を示します。

```
FUNCTION name [(parameter[, parameter, ...])] RETURN datatype IS
    [local declarations]
BEGIN
    executable statements
[EXCEPTION
    exception handlers]
END [name];
```

`parameter` の構文を次に示します。

```
parameter_name [IN | OUT [NOCOPY] | IN OUT [NOCOPY]] datatype_name
    [{:= | DEFAULT} expression]
```

パラメータまたはファンクションの結果のデータ型に、`NOT NULL` など制約を課することはできません。ただし、代替方法を使って、パラメータ型に間接的にサイズ制約を課することができます。7-5 ページの「[ファンクション](#)」を参照してください。

プロシージャと同様に、ファンクションも仕様部と本体の 2 つの部分を持ちます。ファンクションの仕様部はキーワード `FUNCTION` で始め、結果の値のデータ型を指定する `RETURN` 句で終わります。パラメータ宣言はオプションです。パラメータを取らないファンクションではカッコを書きません。

ファンクション本体は、キーワード `IS` で始め、キーワード `END` で終わります。 `END` の後には、オプションとしてファンクション名を続けることができます。ファンクション本体には、宣言部、実行部、例外処理部（オプション）という3つの部分があります。

宣言部では、キーワード `IS` と `BEGIN` の間にローカル宣言を置きます。キーワード `DECLARE` は使いません。実行部では、キーワード `BEGIN` と `EXCEPTION`（または `END`）の間に文を置きます。ファンクションの実行部には、1つまたは複数の `RETURN` 文が存在しなければなりません。例外処理部では、キーワード `EXCEPTION` と `END` の間に例外ハンドラを置きます。従業員の給与が範囲を超えているかどうかを判定するファンクション `sal_ok` を次に示します。

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

このファンクションは、コールされるときに従業員の給与と肩書を受け取ります。肩書はデータベース表 `sals` から範囲の制限を取り出すために使います。ファンクション識別子 `sal_ok` は、`RETURN` 文によってブール値に設定されます。給与が範囲を超えていれば `sal_ok` は `FALSE` に設定され、超えていなければ `TRUE` に設定されます。

ファンクションは、式の一部としてコールされます。たとえば、ファンクション `sal_ok` は次のようにしてコールできます。

```
DECLARE
    new_sal REAL;
    new_title VARCHAR2(15);
BEGIN
    ...
    IF sal_ok(new_sal, new_title) THEN ...
```

ファンクション識別子 `sal_ok` は、渡されるパラメータによって値が異なる変数のような働きをします。

副作用の制御

ファンクションは、次に示す副作用を制御するための " 純正 " レベルの規則に従っている場合に限り、SQL 文からコールできます。

- SELECT 文またはパラレル化 INSERT、UPDATE、DELETE 文からコールされた場合、ファンクションはデータベース表を変更できない。
- INSERT 文、UPDATE 文、または DELETE 文からコールされた場合、ファンクションは、その文によって変更されたデータベース表を問い合わせたり変更したりできない。
- SELECT 文、INSERT 文、UPDATE 文、または DELETE 文からコールされた場合、ファンクションは SQL トランザクション制御文 (COMMIT など)、セッション制御文 (SET ROLE など)、またはシステム制御文 (ALTER SYSTEM など) を実行できない。また、DDL 文 (CREATE など) には自動コミットが続くため、これも実行できない。

ファンクション本体内の SQL 文が規則に違反すると、実行時 (文が解析されるとき) にエラーが発生します。

規則に違反していないか確認するには、プラグマ (コンパイラ・ディレクティブ) `RESTRICT_REFERENCES` を使います。プラグマは、ファンクションがデータベース表またはパッケージ変数 (あるいはその両方) に対する読み込みや書き込み、またはそのいずれも行っていないことを示します。たとえば次のプラグマは、パッケージ・ファンクション `credit_ok` がデータベース書き込み禁止状態 (WNDS) およびパッケージ読み込み禁止状態 (RNPS) であることを示します。

```
CREATE PACKAGE loans AS
...
FUNCTION credit_ok RETURN BOOLEAN;
PRAGMA RESTRICT_REFERENCES (credit_ok, WNDS, RNPS);
END loans;
```

注意: 静的 INSERT 文、UPDATE 文、または DELETE 文は、常に WNDS に違反します。また、列を読み取る場合は RNDS (データベース読み込み禁止状態) にも違反します。動的 INSERT 文、UPDATE 文、または DELETE 文は、常に WNDS および RNDS に違反します。

純正レベルの規則とプラグマ `RESTRICT_REFERENCES` の詳細は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

RETURN 文

RETURN 文は、サブプログラムの実行を即座に完了させ、コールした側に制御を戻します。その後は、サブプログラム・コールの直後の文から、実行が再開されます。(ファンクションの仕様部の中で結果の値のデータ型を指定する RETURN 句と混同しないようにしてください。)

サブプログラムは複数の RETURN 文を持つことができます。そのいずれも、最後の文である必要はありません。どの RETURN 文を実行しても、サブプログラムは即座に終了します。しかし、サブプログラムに複数の終了点を作るのはプログラミングの習慣として好ましくありません。

プロシージャでは、RETURN 文に式を含めることはできません。RETURN 文には、プロシージャ本来の終了地点に達する前に、コールした側に制御を戻す役割しかありません。

ただし、ファンクションにおいて、RETURN 文には文の実行時に評価される式が含まれていなければなりません。結果として得られる値は、RETURN 句で指定された型の変数と同様の機能を持つファンクション識別子に代入されます。ファンクション `balance` が、指定した銀行口座の残高をどのように戻すのかに注目してください。

```
FUNCTION balance (acct_id INTEGER) RETURN REAL IS
    acct_bal REAL;
BEGIN
    SELECT bal INTO acct_bal FROM accts
        WHERE acct_no = acct_id;
    RETURN acct_bal;
END balance;
```

次の例で示すように、ファンクション RETURN 文では、複雑な式も使います。

```
FUNCTION compound (
    years  NUMBER,
    amount NUMBER,
    rate   NUMBER) RETURN NUMBER IS
BEGIN
    RETURN amount * POWER((rate / 100) + 1, years);
END compound;
```

ファンクションは少なくとも 1 つの RETURN 文を持っていなければなりません。RETURN 文が 1 つもない場合、PL/SQL は実行時に事前定義の例外 `PROGRAM_ERROR` をコールします。

サブプログラムの宣言

サブプログラムは、任意の PL/SQL ブロックまたはサブプログラム、パッケージの中で宣言できます。ただし、サブプログラムは、その他すべてのプログラム項目の後の宣言部の末尾で宣言しなければなりません。たとえば、次のプロシージャの宣言は、位置が間違っています。

```
DECLARE
  PROCEDURE award_bonus (...) IS -- misplaced; must come last
  BEGIN
    ...
  END;
  rating NUMBER;
  CURSOR c1 IS SELECT * FROM emp;
```

前方宣言

PL/SQL では、識別子は宣言してからでなければ使えません。したがって、サブプログラムも、コールする前に宣言しておかなければなりません。たとえば、プロシージャ `award_bonus` の次のような宣言は無効です。コールの時点では宣言されていないプロシージャ `calc_rating` をコールしているからです。

```
DECLARE
  ...
  PROCEDURE award_bonus ( ... ) IS
  BEGIN
    calc_rating( ... ); -- undeclared identifier
    ...
  END;
  PROCEDURE calc_rating ( ... ) IS
  BEGIN
    ...
  END;
```

この場合は、プロシージャ `calc_rating` をプロシージャ `award_bonus` の前に置けば、問題は解決します。しかし、いつも簡単に解決できるとは限りません。たとえば、プロシージャが相互再帰的になっている場合（互いにコールし合う場合）や、アルファベット順に定義したい場合などがそうです。PL/SQL は、この問題を解決するために、前方宣言というサブプログラムの特殊な宣言を備えています。前方宣言を使うのは、次のような場合です。

- サブプログラムを、論理の順またはアルファベット順に定義する場合
- 相互再帰的なサブプログラムを定義する場合（7-37 ページの「[再帰](#)」を参照）
- サブプログラムのグループをパッケージに入れる場合

前方宣言は、末尾にセミコロンを付けたサブプログラム仕様部で構成されています。次の例の前方宣言では、プロシージャ `calc_rating` の本体がブロックの後半にあることを PL/SQL に知らせています。

```
DECLARE
    PROCEDURE calc_rating ( ... ); -- forward declaration
    ...
    /* Define subprograms in alphabetical order. */
    PROCEDURE award_bonus ( ... ) IS
    BEGIN
        calc_rating( ... );
        ...
    END;
    PROCEDURE calc_rating ( ... ) IS
    BEGIN
        ...
    END;
```

仮パラメータのリストは前方宣言の中にありますが、サブプログラム本体にも必要です。サブプログラム本体の位置は、前方宣言の後ならどこでもかまいませんが、同じプログラム・ユニットの中でなければなりません。

パッケージ内

前方宣言を使うと、論理的に関連のあるサブプログラムのグループをパッケージに入れることもできます。サブプログラム仕様部はパッケージ仕様部に、サブプログラム本体はパッケージ本体に入れます。この場合、どちらもアプリケーションには認識できません。たとえば、

```
CREATE PACKAGE emp_actions AS -- package spec
    PROCEDURE hire_employee (emp_id INTEGER, name VARCHAR2, ...);
    PROCEDURE fire_employee (emp_id INTEGER);
    PROCEDURE raise_salary (emp_id INTEGER, amount REAL);
    ...
END emp_actions;

CREATE PACKAGE BODY emp_actions AS -- package body
    PROCEDURE hire_employee (emp_id INTEGER, name VARCHAR2, ...) IS
    BEGIN
        ...
        INSERT INTO emp VALUES (empno, ename, ...);
    END hire_employee;

    PROCEDURE fire_employee (emp_id INTEGER) IS
    BEGIN
        DELETE FROM emp
            WHERE empno = emp_id;
    END fire_employee;
```

```
PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS
    salary REAL;
BEGIN
    SELECT sal INTO salary FROM emp
        WHERE empno = emp_id;
    ...
END raise_salary;
...
END emp_actions;
```

パッケージ仕様部でサブプログラム仕様部を宣言せずに、パッケージ本体でサブプログラムを定義できます。しかし、このようなサブプログラムは、パッケージの中からしかコールできません。パッケージの詳細は、[第 8 章](#)を参照してください。

ストアド・サブプログラム

一般に、PL/SQL エンジンを組み込む (Oracle Forms などの) ツールは、サブプログラムを格納しておいて、後からローカルでだけ実行できます。ただし、一般的な用途に使うためには、サブプログラムを Oracle データベースに格納しておかなければなりません。

サブプログラムを作り、Oracle データベースの中に永続的に格納するには、SQL*Plus から対話式で実行できる CREATE PROCEDURE 文および CREATE FUNCTION 文を使います。たとえば次のように、SQL*Plus で、プロシージャ fire_employee を作成できます。

```
SQL> list
1 CREATE PROCEDURE fire_employee (emp_id NUMBER) AS
2 BEGIN
3     DELETE FROM emp WHERE empno = emp_id;
4* END;
SQL> /
```

Procedure created.

サブプログラムを作る場合は、読みやすくするために、仕様部でキーワード IS のかわりにキーワード AS を使えます。ストアド・サブプログラムの作成と使用の詳細は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

実パラメータと仮パラメータ

サブプログラムはパラメータを使って情報を渡します。サブプログラムをコールする場合のパラメータ・リストの中で参照される変数や式は、**実パラメータ**と呼ばれます。たとえば、次のプロシージャ・コールでは、`emp_num`と`amount`という2つの実パラメータが指定されています。

```
raise_salary(emp_num, amount);
```

次のプロシージャ・コールでは、**実パラメータ**として式を使っています。

```
raise_salary(emp_num, merit + cola);
```

サブプログラムの仕様部で宣言され、サブプログラム本体で参照される変数は、**仮パラメータ**と呼ばれます。たとえば、次のプロシージャは、`emp_id`と`amount`という2つの仮パラメータを宣言しています。

```
PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS
    current_salary REAL;
    ...
BEGIN
    SELECT sal INTO current_salary FROM emp WHERE empno = emp_id;
    ...
    UPDATE emp SET sal = sal + amount WHERE empno = emp_id;
END raise_salary;
```

実パラメータと仮パラメータに別々の名前を付けることは、プログラミングの習慣として好ましいことです。

プロシージャ `raise_salary` をコールすると、**実パラメータ**が評価され、対応する**仮パラメータ**にその値が代入されます。**実パラメータ**の値を**仮パラメータ**に代入する前に、PL/SQL は必要ならば値のデータ型を変換します。たとえば、次の例では、`raise_salary` が正しくコールされています。

```
raise_salary(emp_num, '2500');
```

実パラメータと、それに対応する仮パラメータは、互換性のあるデータ型を持っていないければなりません。たとえば、PL/SQL は `DATE` データ型と `REAL` データ型の間で変換できません。また、実パラメータの値は仮パラメータのデータ型に変換できなければなりません。次のプロシージャ・コールでは、PL/SQL が2番目の実パラメータを数値に変換できないために、事前定義の例外 `VALUE_ERROR` が呼び出されます。

```
raise_salary(emp_num, '$2500'); -- note the dollar sign
```

詳細は、2-25 ページの「[データ型の変換](#)」を参照してください。

位置表記法と名前表記法

サブプログラムをコールする場合の実パラメータの指定方法は、位置表記法と名前表記法の2つがあります。つまり、実パラメータと仮パラメータの間の結び付きは、位置と名前のどちらかで指示します。たとえば、次のような宣言を考えます。

```
DECLARE
    acct INTEGER;
    amt  REAL;
    PROCEDURE credit_acct (acct_no INTEGER, amount REAL) IS ...
```

次に示すプロシージャ credit_acct の4種類のコールは、いずれも論理的に等価です。

```
BEGIN
    credit_acct(acct, amt);           -- positional notation
    credit_acct(amount => amt, acct_no => acct); -- named notation
    credit_acct(acct_no => acct, amount => amt); -- named notation
    credit_acct(acct, amount => amt);  -- mixed notation
```

位置表記法

1つ目のプロシージャ・コールでは位置表記法を使っています。PL/SQL コンパイラは、1番目の実パラメータ acct を1番目の仮パラメータ acct_no と結び付けます。また、2番目の実パラメータ amt を2番目の仮パラメータ amount と結び付けます。

名前表記法

2つ目のプロシージャ・コールでは名前表記法を使っています。矢印(=>)は結合演算子の役割を果たし、矢印の左側の仮パラメータと矢印の右側の実パラメータを結び付けます。

3つ目のプロシージャ・コールでも名前指定対応を使っています。この例では、パラメータのペアを任意の順序で指定できることを示しています。つまり、ユーザーは仮パラメータの指定順序を知る必要がありません。

表記法の混在

4つ目のプロシージャ・コールでは、位置表記法と名前表記法を混在させています。この例では、1番目のパラメータで位置表記法を使い、2番目のパラメータで名前表記法を使っています。位置表記法は名前表記法よりも前で使わなければなりません。逆の順序で使うことはできません。たとえば、次のようなプロシージャ・コールは誤りです。

```
credit_acct(acct_no => acct, amt); -- illegal
```

パラメータのモード

パラメータ・モードは、仮パラメータの動作を定義する場合に使います。サブプログラムで使えるモードには、IN (デフォルト)、OUT、IN OUT の3つがあります。ただし、ファンクションでは、OUT モードと IN OUT モードを使わないようにしてください。ファンクションの目的は、0 (ゼロ) 個以上の引数 (実パラメータ) を取り、単一の値を戻すことです。ファンクションが複数の値を戻すようなプログラミングは、好ましくありません。また、サブプログラム専用でない変数の値を変更するなどの副作用にも注意してください。

IN モード

IN パラメータは、コールされるサブプログラムに値を渡すために使います。サブプログラムの中では、IN パラメータは定数のように扱われます。したがって、値を代入できません。たとえば、次の代入文ではコンパイル・エラーが発生します。

```
PROCEDURE debit_account (acct_id IN INTEGER, amount IN REAL) IS
    minimum_purchase CONSTANT REAL DEFAULT 10.0;
    service_charge     CONSTANT REAL DEFAULT 0.50;
BEGIN
    IF amount < minimum_purchase THEN
        amount := amount + service_charge; -- causes compilation error
    END IF;
    ...
END debit_account;
```

IN 仮パラメータに対応する実パラメータには、定数、リテラル、初期化された変数または式が使えます。OUT パラメータおよび IN OUT パラメータとは異なり、IN パラメータはデフォルト値に初期化できます。詳細は、7-19 ページの「[パラメータのデフォルト値](#)」を参照してください。

OUT モード

OUT パラメータは、サブプログラムのコール側に値を戻すために使います。サブプログラムの中では、OUT パラメータは変数のように扱われます。つまり、OUT 仮パラメータは、ローカル変数のように使えます。次の例に示すように、値の変更や参照ができます。

```
PROCEDURE calc_bonus (emp_id IN INTEGER, bonus OUT REAL) IS
    hire_date     DATE;
    bonus_missing EXCEPTION;
BEGIN
    SELECT sal * 0.10, hiredate INTO bonus, hire_date FROM emp
        WHERE empno = emp_id;
    IF bonus IS NULL THEN
        RAISE bonus_missing;
    END IF;
    IF MONTHS_BETWEEN(SYSDATE, hire_date) > 60 THEN
```



```

        bonus := bonus + 500;
    END IF;
    ...
EXCEPTION
    WHEN bonus_missing THEN
        ...
END calc_bonus;

```

OUT 仮パラメータに対応する実パラメータは、定数や式ではなく、変数でなければなりません。たとえば、次のようなプロシージャ・コールは誤りです。

```
calc_bonus(7499, salary + commission); -- causes compilation error
```

OUT 実パラメータには、サブプログラムがコールされる前にも値を入れることができます。しかし、サブプログラムをコールした時点で、値は失われます（未処理例外で終了しない場合）。

変数と同じように、OUT 仮パラメータは NULL に初期設定されます。このため、OUT 仮パラメータは NOT NULL として定義されたサブタイプ（組み込みサブタイプ NATURALN および POSITIVEN を含む）にはできません。そうしないと、サブプログラムをコールしたときに PL/SQL は VALUE_ERROR を呼び出します。たとえば、

```

DECLARE
    SUBTYPE Counter IS INTEGER NOT NULL;
    rows Counter := 0;
    PROCEDURE count_ems (n OUT Counter) IS
    BEGIN
        SELECT COUNT(*) INTO n FROM emp;
    END;
BEGIN
    count_ems(rows); -- raises VALUE_ERROR

```

サブプログラムを終了する前に、すべての OUT 仮パラメータに明示的に値を代入するようにしてください。そうしないと、対応する実パラメータの値は NULL になります。実行に成功して終了した場合、PL/SQL は実パラメータに値を代入します。しかし、未処理例外が発生して実行が終了すると、PL/SQL は実パラメータに値を代入しません。

IN OUT モード

IN OUT パラメータを使うと、コールされる側のサブプログラムに初期値を渡して、コールした側に更新された値を戻すことができます。サブプログラムの中では、IN OUT パラメータは初期化された変数のように取り扱われます。このため、IN OUT パラメータに値を代入したり、その値を他の変数に代入したりできます。

IN OUT 仮パラメータに対応する実パラメータは、定数や式ではなく、変数でなければなりません。

サブプログラムを正常に終了した場合、PL/SQL は実パラメータに値を代入します。しかし、未処理例外が発生して実行が終了すると、PL/SQL は実パラメータに値を代入しません。

まとめ

表 7-1 に、パラメータ・モードについて知っておくべきことをまとめて示します。

表 7-1 パラメータのモード

IN	OUT	IN OUT
デフォルト	指定する必要がある	指定する必要がある
サブプログラムに値を渡す	コールした側に値を戻す	サブプログラムに初期値を渡し、更新された値をコールした側に戻す
仮パラメータは定数のように取り扱われる	仮パラメータは変数のように取り扱われる	仮パラメータは初期化された変数のように取り扱われる
仮パラメータに値を代入できない	仮パラメータには値を代入しなければならない	仮パラメータには値を代入しなければならない
実パラメータには定数、リテラル、初期化された変数または式が使える	実パラメータは変数でなければならない	実パラメータは変数でなければならない
実パラメータは参照方式によって渡される（その値を指すポインタが渡される）	NOCOPY が指定されていないかぎり、実パラメータは値方式によって渡される（値のコピーが戻される）	NOCOPY が指定されていないかぎり、実パラメータは値方式によって渡される（値のコピーがやりとりされる）

注意：NOCOPY コンパイラ・ヒントによって、PL/SQL コンパイラは OUT パラメータおよび IN OUT パラメータを参照方式によって渡すことができる詳細は、7-17 ページの「[NOCOPY コンパイラ・ヒント](#)」を参照。

NOCOPY コンパイラ・ヒント

サブプログラムが、IN パラメータ、OUT パラメータ、および IN OUT パラメータを宣言するとします。サブプログラムをコールすると、IN パラメータが参照方式によって渡されます。つまり、IN 実パラメータへのポインタが、対応する仮パラメータに渡されます。このため、パラメータは両方とも、実パラメータの値を保持する同じメモリー位置を参照します。

デフォルトでは、OUT パラメータと IN OUT パラメータは値によって渡されます。つまり、IN OUT 実パラメータの値は、対応する仮パラメータにコピーされます。サブプログラムが正常に終了すると、OUT および IN OUT 仮パラメータに代入された値は、対応する実パラメータにコピーされます。

パラメータが、コレクション、レコードおよびオブジェクト型のインスタンスなどの大きなデータ構造を保持している場合、このコピー作業によって実行速度が遅くなり、メモリーが消費されます。これを回避するには、NOCOPY ヒントを指定します。これによって、PL/SQL コンパイラは OUT および IN OUT パラメータを参照によって渡すことができます。

次の例では、IN OUT パラメータ `my_staff` を、値ではなく参照によって渡すよう、コンパイラに指示します。

```
DECLARE
  TYPE Staff IS VARRAY(200) OF Employee;
  PROCEDURE reorganize (my_staff IN OUT NOCOPY Staff) IS ...
```

NOCOPY は、ディレクティブではなく、ヒントです。コンパイラは、ユーザーの要求に関係なく `my_staff` を値方式で渡します。ただし通常は、NOCOPY は成功します。これは大きなデータ構造を扱う PL/SQL アプリケーションで役立ちます。

次の例では、ローカルなネストした表に 5000 レコードがロードされます。この表は、NULL 文を実行するだけの 2 つのローカル・プロシージャに渡されます。しかし、プロシージャを 1 つコールするには、コピー作業のために 26 秒かかります。NOCOPY を使うと、もう片方のプロシージャのコールには 1 秒もかかりません。

```
SQL> SET SERVEROUTPUT ON
SQL> GET test.sql
1  DECLARE
2      TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE;
3      emp_tab EmpTabTyp := EmpTabTyp(NULL); -- initialize
4      t1 CHAR(5);
5      t2 CHAR(5);
6      t3 CHAR(5);
7      PROCEDURE get_time (t OUT NUMBER) IS
8      BEGIN SELECT TO_CHAR(SYSDATE, 'SSSSS') INTO t FROM dual; END;
9      PROCEDURE do_nothing1 (tab IN OUT EmpTabTyp) IS
10     BEGIN NULL; END;
11     PROCEDURE do_nothing2 (tab IN OUT NOCOPY EmpTabTyp) IS
12     BEGIN NULL; END;
13 BEGIN
14     SELECT * INTO emp_tab(1) FROM emp WHERE empno = 7788;
```

```
15     emp_tab.EXTEND(4999, 1); -- copy element 1 into 2..5000
16     get_time(t1);
17     do_nothing1(emp_tab); -- pass IN OUT parameter
18     get_time(t2);
19     do_nothing2(emp_tab); -- pass IN OUT NOCOPY parameter
20     get_time(t3);
21     DBMS_OUTPUT.PUT_LINE('Call Duration (secs)');
22     DBMS_OUTPUT.PUT_LINE('-----');
23     DBMS_OUTPUT.PUT_LINE('Just IN OUT: ' || TO_CHAR(t2 - t1));
24     DBMS_OUTPUT.PUT_LINE('With NOCOPY: ' || TO_CHAR(t3 - t2));
25* END;
SQL> /
Call Duration (secs)
-----
Just IN OUT: 26
With NOCOPY: 0
```

パフォーマンス向上のトレードオフ

NOCOPY を使用すると、定義のしっかりした例外処理方法と引換えに、パフォーマンスを向上できます。これを使用することによって、例外処理に次のような影響があります。

- NOCOPY はディレクティブではなくヒントなので、コンパイラは NOCOPY パラメータを、値方式が参照方式によってサブプログラムに渡すことができる。このため、未処理例外が発生してサブプログラムが終了した場合、NOCOPY 実パラメータの値は信頼できない。
- デフォルトでは、サブプログラムが未処理例外で終了すると、その OUT および IN OUT 仮パラメータに代入された値は、対応する実パラメータにコピーされず、変更はロールバックされる。しかし、NOCOPY を指定すると、仮パラメータへの代入値は直ちに実パラメータにも影響する。このため、そのサブプログラムが未処理例外で終了した場合、変更（完了していない可能性がある）は "ロールバック" されない。
- 現在のところ、RPC プロトコルを使用すると、パラメータは値方式によってのみ渡すことができる。このため、アプリケーションをパーティションするとき、例外処理方法が暗黙的に変更されることがある。たとえば、NOCOPY パラメータを指定したローカル・プロシージャをリモート・サイトに移動すると、これらのパラメータは参照方式では渡せなくなる。

また、NOCOPY を使用することによって、パラメータ・エイリアシングの可能性が高くなります。詳細は、7-21 ページの「[パラメータのエイリアシング](#)」を参照してください。

NOCOPY の制限

次の場合には、PL/SQL コンパイラは NOCOPY ヒントを無視して、値方式によってパラメータを受け渡します（エラーは生成されません）。

- 実パラメータは索引付き表の要素である。この制限は索引付き表全体には適用されない。
- 実パラメータには（たとえば位取りや NOT NULL などによって）制約がある。この制限は制約付きの要素または属性には拡張されない。また、サイズ制約付きの文字列にも適用されない。
- 実パラメータと仮パラメータはレコードである。いずれか、またはいずれのレコードも %ROWTYPE か %TYPE を使用して宣言されており、レコード内の対応するフィールドの制約は異なる。
- 実パラメータと仮パラメータはレコードである。実パラメータはカーソル FOR ループの索引として（暗黙的に）宣言されており、レコード内の対応するフィールドの制約は異なる。
- 実パラメータを渡すには、暗黙のデータ型変換が必要になる。
- サブプログラムは外部プロシージャ・コールまたはリモート・プロシージャ・コール（RPC）に関連する。

パラメータのデフォルト値

次の例に示すように、IN パラメータは、デフォルト値に初期化できます。初期化すると、サブプログラムに実パラメータとしてさまざまな数値を渡し、場合に応じてデフォルト値を受け入れることも上書きすることもできます。さらに、サブプログラムへのコールを個々に変更しなくても、仮パラメータを新しく追加できます。

```
PROCEDURE create_dept (
    new_dname CHAR DEFAULT 'TEMP',
    new_loc    CHAR DEFAULT 'TEMP') IS
BEGIN
    INSERT INTO dept
        VALUES (deptno_seq.NEXTVAL, new_dname, new_loc);
    ...
END;
```

実パラメータが渡されないと、対応する仮パラメータのデフォルト値が使われます。次に示すような create_dept のコールを考えてみます。

```
create_dept;
create_dept('MARKETING');
create_dept('MARKETING', 'NEW YORK');
```

1 番目のコールでは実パラメータを渡していないので、両方のデフォルト値が使われます。2 番目のコールでは実パラメータを 1 つ渡しているので、new_loc はデフォルト値が使われます。3 番目のコールでは実パラメータを 2 つ渡しているので、デフォルト値はどちらも使われません。

大部分の場合は、位置指定対応を使うと、仮パラメータのデフォルト値を上書きできます。ただし、実パラメータを省略して、仮パラメータを1つスキップするといったことはできません。たとえば、次のコールでは、実パラメータ 'NEW YORK' が、誤って仮パラメータ `new_dname` と結び付けられます。

```
create_dept('NEW YORK'); -- incorrect
```

実パラメータのプレースホルダを使っても、この問題点は解決できません。たとえば、次のコールは誤りです。

```
create_dept(, 'NEW YORK'); -- illegal
```

このような場合は、次のように名前表記法を使います。

```
create_dept(new_loc => 'NEW YORK');
```

また、実パラメータを省略して、初期化されていない仮パラメータに NULL を代入できません。たとえば、次のような宣言があるとします。

```
DECLARE
    FUNCTION gross_pay (
        emp_id    IN NUMBER,
        st_hours  IN NUMBER DEFAULT 40,
        ot_hours  IN NUMBER) RETURN REAL IS
    BEGIN
        ...
    END;
```

次のファンクション・コールは、NULL を `ot_hours` に代入しません。

```
IF gross_pay(emp_num) > max_pay THEN ... -- illegal
```

このため、次のように NULL を明示的に渡さなければなりません。

```
IF gross_pay(emp_num, ot_hour => NULL) > max_pay THEN ...
```

または、次のように `ot_hours` を初期化して NULL にできます。

```
ot_hours IN NUMBER DEFAULT NULL;
```

最後に、ストアド・サブプログラムを作る場合、DEFAULT 句の中ではホスト変数（バインド変数）を使えません。SQL*Plus の次の例では、「不適当なバインド変数（*bad bind variable*）」のエラーが呼び出されます。作成時に `num` が、その値が変更する可能性のあるプレースホルダとなっているためです。

```
SQL> VARIABLE num NUMBER
SQL> CREATE FUNCTION gross_pay (emp_id IN NUMBER DEFAULT :num, ...
```

パラメータのエイリアシング

サブプログラムのコールを最適化するために、PL/SQL コンパイラでは、2つのパラメータ受渡し方式のいずれかを選択できます。値方式では、実パラメータの値がサブプログラムに渡されます。参照方式では、値へのポインタだけが渡されます。この場合、実パラメータと仮パラメータとは同じ項目を参照します。

NOCOPY コンパイラ・ヒントによって、エイリアシングの可能性が高くなります（つまり、異なる2つの名前が同じメモリー位置を参照するようになります）。これは、グローバル変数がサブプログラムのコールの中で実パラメータとして使われ、そのサブプログラム内で参照されると発生します。結果はコンパイラが選ぶパラメータの受け渡し方式に依存するため、予測不能になります。

次の例では、プロシージャ `add_entry` は、`varray lexicon` を、パラメータとして参照する方法と、グローバル変数として参照する方法の、異なる2つの方法で参照しています。`add_entry` がコールされると、識別子 `word_list` および `lexicon` は同じ `varray` を指定します。

```
DECLARE
    TYPE Definition IS RECORD (
        word    VARCHAR2(20),
        meaning VARCHAR2(200));
    TYPE Dictionary IS VARRAY(2000) OF Definition;
    lexicon Dictionary := Dictionary();
    PROCEDURE add_entry (word_list IN OUT NOCOPY Dictionary) IS
    BEGIN
        word_list(1).word := 'aardvark';
        lexicon(1).word := 'aardwolf';
    END;
BEGIN
    lexicon.EXTEND;
    add_entry(lexicon);
    DEMS_OUTPUT.PUT_LINE(lexicon(1).word);
    -- prints 'aardvark' if parameter was passed by value
    -- prints 'aardwolf' if parameter was passed by reference
END;
```

結果は、コンパイラが選ぶパラメータの受け渡し方式によって異なります。コンパイラが値方式を選ぶ場合、`word_list` と `lexicon` は同じ `varray` からの別々のコピーとなります。そのため、一方を変更しても他方は変更されません。しかし、コンパイラが参照方式を選択する場合、`word_list` および `lexicon` は名前だけが異なる同じ `varray` にすぎません。（"エイリアシング（別名）"と呼ばれるのはこのためです。）そのため、`lexicon(1)` の値を変えると `word_list(1)` の値も変わります。

エイリアシングは、1回のサブプログラム・コールに、同じ実パラメータが2回以上現れる場合にも発生します。次の例では、`n2` が `IN OUT` のパラメータであるため、実パラメータの値は、プロシージャが終了するまで更新されません。そのため、最初の `PUT_LINE` は 10 (`n` の初期値) を出力し、3 番目の `PUT_LINE` は 20 を出力します。しかし `n3` は `NOCOPY` パラメータなので、実パラメータの値はただちに更新されます。2 番目の `PUT_LINE` が 30 を出力するのはこのためです。

```
DECLARE
  n NUMBER := 10;
  PROCEDURE do_something (
    n1 IN NUMBER,
    n2 IN OUT NUMBER,
    n3 IN OUT NOCOPY NUMBER) IS
  BEGIN
    n2 := 20;
    DBMS_OUTPUT.PUT_LINE(n1); -- prints 10
    n3 := 30;
    DBMS_OUTPUT.PUT_LINE(n1); -- prints 30
  END;
BEGIN
  do_something(n, n, n);
  DBMS_OUTPUT.PUT_LINE(n); -- prints 20
END;
```

これらはポインタであるため、カーソル変数にもエイリアシングの可能性がありま。次の例を考えてみてください。代入の後、`emp_cv2` は `emp_cv1` の別名になります。これは、両者が同じ問合せ作業域を指すためです。したがって、どちらも状態を変更できます。そのため、`emp_cv2` からの最初の取出しで 1 行目ではなく 3 行目を取り出され、`emp_cv1` をクローズした後の `emp_cv2` からの 2 回目の取出しは失敗します。

```
PROCEDURE get_emp_data (
  emp_cv1 IN OUT EmpCurTyp,
  emp_cv2 IN OUT EmpCurTyp) IS
  emp_rec emp%ROWTYPE;
BEGIN
  OPEN emp_cv1 FOR SELECT * FROM emp;
  emp_cv2 := emp_cv1;
  FETCH emp_cv1 INTO emp_rec; -- fetches first row
  FETCH emp_cv1 INTO emp_rec; -- fetches second row
  FETCH emp_cv2 INTO emp_rec; -- fetches third row
  CLOSE emp_cv1;
  FETCH emp_cv2 INTO emp_rec; -- raises INVALID_CURSOR
  ...
END;
```


オーバーロード

PL/SQL ではサブプログラム名をオーバーロードできます。つまり、仮パラメータの数または順序、データ型が異なりさえすれば、同じ名前を複数のサブプログラムで使えます。

次のように宣言された 2 つの索引付き表の最初の n 行を初期化するとします。

```
DECLARE
    TYPE DateTabTyp IS TABLE OF DATE INDEX BY BINARY_INTEGER;
    TYPE RealTabTyp IS TABLE OF REAL INDEX BY BINARY_INTEGER;
    hiredate_tab DateTabTyp;
    sal_tab RealTabTyp;
```

次のようなプロシージャを作ると、hiredate_tab という名前の索引付き表を初期化できます。

```
PROCEDURE initialize (tab OUT DateTabTyp, n INTEGER) IS
BEGIN
    FOR i IN 1..n LOOP
        tab(i) := SYSDATE;
    END LOOP;
END initialize;
```

次のようなプロシージャを作ると、sal_tab という名前の索引付き表を初期化できます。

```
PROCEDURE initialize (tab OUT RealTabTyp, n INTEGER) IS
BEGIN
    FOR i IN 1..n LOOP
        tab(i) := 0.0;
    END LOOP;
END initialize;
```

この 2 つのプロシージャは同じ処理を実行しているため、同じ名前を与えるのが論理的です。

オーバーロードされたこの 2 つの initialize プロシージャは、同じブロックまたはサブプログラム、パッケージの中に置くことができます。PL/SQL はパラメータを検査して、2 つのプロシージャのどちらがコールされるかを判定します。

次の例を考えてみてください。DateTabTyp パラメータで initialize をコールすると、PL/SQL は前者の initialize を使います。しかし、RealTabTyp パラメータで initialize をコールすると、PL/SQL は後者の initialize を使います。

```
DECLARE
    TYPE DateTabTyp IS TABLE OF DATE INDEX BY BINARY_INTEGER;
    TYPE RealTabTyp IS TABLE OF REAL INDEX BY BINARY_INTEGER;
    hiredate_tab DateTabTyp;
    comm_tab RealTabTyp;
    indx BINARY_INTEGER;
```

```
BEGIN
  indx := 50;
  initialize(hiredate_tab, indx); -- calls first version
  initialize(comm_tab, indx);    -- calls second version
```

制限

オーバーロードできるのは、ローカル・サブプログラムとパッケージ・サブプログラムだけです。このため、スタンドアロン・サブプログラムはオーバーロードできません。また、サブプログラムの仮パラメータの違いが名前とパラメータ・モードだけの場合は、2つのサブプログラムをオーバーロードできません。たとえば、次の2つのプロシージャはオーバーロードできません。

```
PROCEDURE reconcile (acct_no IN INTEGER) IS BEGIN ... END;
PROCEDURE reconcile (acct_no OUT INTEGER) IS BEGIN ... END;
```

また、サブプログラムの仮パラメータの違いがデータ型だけの場合、または違うデータ型でも同じファミリのものである場合は、2つのサブプログラムをオーバーロードできません。たとえば、次の例では、データ型 `INTEGER` と `REAL` が同じファミリであるため、この2つのプロシージャはオーバーロードできません。

```
PROCEDURE charge_back (amount INTEGER) IS BEGIN ... END;
PROCEDURE charge_back (amount REAL) IS BEGIN ... END;
```

同様に、2つのサブプログラムの仮パラメータの違いがサブタイプだけで、かつこれらのサブタイプが同じファミリ内の型を基礎型とする場合は、これらのサブプログラムをオーバーロードできません。たとえば、次の例では、基本型 `CHAR` と `LONG` が同じファミリであるため、この2つのプロシージャはオーバーロードできません。

```
DECLARE
  SUBTYPE Delimiter IS CHAR;
  SUBTYPE Text IS LONG;
  PROCEDURE scan (x Delimiter) IS BEGIN ... END;
  PROCEDURE scan (x Text) IS BEGIN ... END;
```

最後に、戻り型（結果の値のデータ型）にしか違いがない2つのファンクションは、その型のファミリが違っている場合でも、オーバーロードできません。たとえば、次の2つのファンクションはオーバーロードできません。

```
FUNCTION acct_ok (acct_id INTEGER) RETURN BOOLEAN IS BEGIN ... END;
FUNCTION acct_ok (acct_id INTEGER) RETURN INTEGER IS BEGIN ... END;
```

コールの判定

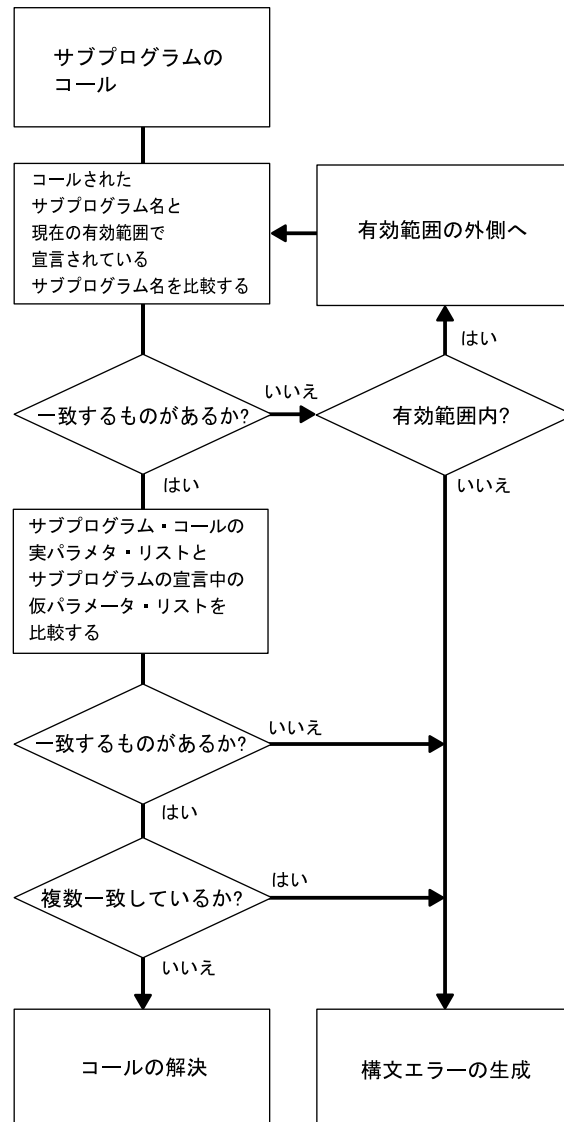
図 7-1 に、PL/SQL コンパイラがサブプログラム・コールを解決する方法を示します。コンパイラがプロシージャ・コールまたはファンクション・コールを発見すると、そのコールに合う宣言を探そうとします。コンパイラはまず現在の有効範囲を検索し、必要ならば外側の有効範囲を順に検索します。コールされたサブプログラムの名前と同じ名前のサブプログラム宣言が 1 つまたは複数見つかった段階で、コンパイラは検索を中止します。

同じ有効範囲のレベルに同じような名前のサブプログラムが存在する場合は、コールを解決するため、コンパイラは実パラメータと仮パラメータが正確に一致するものを発見しなければなりません。つまり、(いくつかの仮パラメータにデフォルト値が代入されている場合を除き) パラメータの数および順序、データ型が一致しなければなりません。一致するものが発見できなかった場合、または一致するものが複数発見された場合、コンパイラは構文エラーを生成します。

次の例では、ファンクション `valid` から外側のプロシージャ `swap` をコールしています。しかし、現在の有効範囲の中にある `swap` の宣言が、いずれもプロシージャ・コールと一致しないために、コンパイラはエラーを生成します。

```
PROCEDURE swap (d1 DATE, d2 DATE) IS
    date1 DATE;
    date2 DATE;
    FUNCTION valid (d DATE) RETURN BOOLEAN IS
        PROCEDURE swap (n1 INTEGER, n2 INTEGER) IS BEGIN ... END;
        PROCEDURE swap (n1 REAL, n2 REAL) IS BEGIN ... END;
    BEGIN
        ...
        swap(date1, date2);
    END valid;
BEGIN
    ...
END;
```

図 7-1 PL/SQL コンパイラによるコールの解決方法



コール解決エラーの回避

PL/SQL では、組み込みファンクションをパッケージ STANDARD の中でグローバルに宣言します。これらをローカルに再宣言すると、ローカル宣言がグローバル宣言よりも優先されるためエラーが発生しやすくなります。たとえば、次の例では `sign` という名前のファンクションを宣言し、その宣言の有効範囲の中で組み込みファンクション `SIGN` をコールしようとしています。

```
DECLARE
    x NUMBER;
    ...
BEGIN
    DECLARE
        FUNCTION sign (n NUMBER) RETURN NUMBER IS
        BEGIN
            IF n < 0 THEN
                RETURN -1;
            ELSE
                RETURN 1;
            END IF;
        END;
    BEGIN
        ...
        x := SIGN(0); -- assigns 1 to x
    END;
    ...
    x := SIGN(0); -- assigns 0 to x
END;
```

サブブロックの中で、PL/SQL は組み込み定義ではなくファンクション定義を使います。サブブロックの中から組み込みファンクションをコールする場合は、次のようにドット表記法を使わなければなりません。

```
x := STANDARD.SIGN(0); -- assigns 0 to x
```

外部ルーチンのコール

PL/SQL は強力な開発ツールです。ほとんどどんな用途にも使用できます。しかし、それは SQL トランザクション処理に特化されています。そのため、作業の中には、マシン精度の計算で効率のよい C などの低レベルの言語で実行するとさらに高速に処理できるものがあります。Java などの、完全なオブジェクト指向の標準化言語では、さらに簡単に処理できる作業もあります。

そのような特殊な目的の処理をサポートするには、PL/SQL コール仕様部を使用して外部ルーチン（他の言語で書かれたルーチン）を呼び出すことができます。これによって、それら他の言語の長所や機能を活用できます。固有の限界がある 1 つの言語だけに制限される必要はもうありません。

たとえば、Java ストアド・プロシージャは、任意の PL/SQL ブロックまたはサブプログラム、パッケージからコールできます。データベースに次の Java クラスを格納するとします。

```
import java.sql.*;
import oracle.jdbc.driver.*;

public class Adjuster {
    public static void raiseSalary (int empNo, float percent)
        throws SQLException {
        Connection conn = new OracleDriver().defaultConnection();
        String sql = "UPDATE emp SET sal = sal * ? WHERE empno = ?";
        try {
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setFloat(1, (1 + percent / 100));
            pstmt.setInt(2, empNo);
            pstmt.executeUpdate();
            pstmt.close();
        } catch (SQLException e) {System.err.println(e.getMessage());}
    }
}
```

クラス Adjuster には、従業員の給与を指定のパーセンテージ分だけ増やすメソッドがあります。raiseSalary は void メソッドなので、次のコール仕様部を使用してプロシージャとして発行します。

```
CREATE PROCEDURE raise_salary (empno NUMBER, pct NUMBER)
AS LANGUAGE JAVA
NAME 'Adjuster.raiseSalary(int, float)';
```

後で、プロシージャ raise_salary を無名 PL/SQL ブロックから次のようにコールできます。

```
DECLARE
    emp_id  NUMBER;
    percent NUMBER;
BEGIN
    -- get values for emp_id and percent
    raise_salary(emp_id, percent); -- call external routine
```

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

通常、外部 C ルーチンは、組込みシステムとのインタフェース、技術的な分野の問題解決、データの分析、リアルタイムのデバイスや処理の制御に使います。さらに、外部 C ルーチンを使用すると、データベース・サーバーの機能性を拡張し、計算専用プログラムをクライアントからサーバーに移動できます。サーバーの方が高速に処理できます。

外部 C ルーチンの詳細は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

実行者権限と定義者権限

デフォルトでは、ストアド・プロシージャおよび SQL メソッドは、実行者の権限ではなく定義者の権限で実行します。このような定義者権限ルーチンはスキーマにバインドされ、そこに常駐します。たとえば、表 dept の複製がスキーマ scott とスキーマ blake に常駐し、次のスタンドアロン・プロシージャがスキーマ scott に常駐するとします。

```
CREATE PROCEDURE create_dept (new_dname CHAR, new_loc CHAR) AS
BEGIN
    INSERT INTO dept
        VALUES (deptno_seq.NEXTVAL, new_dname, new_loc);
END;
```

さらに、ユーザー scott が、このプロシージャに対する EXECUTE 権限をユーザー blake に付与したとします。ユーザー blake がプロシージャをコールすると、INSERT 文がユーザー scott の権限で実行されます。また、表 dept への未修飾の参照は、スキーマ scott 内で解決されます。このため、プロシージャは、スキーマ blake ではなく、スキーマ scott 内で dept 表を更新します。

あるスキーマのルーチンは、どのようにして別のスキーマのオブジェクトを操作するのでしょうか。1 つには、次のようにオブジェクトへの参照を完全に修飾する方法があります。

```
INSERT INTO blake.dept ...
```

しかし、これは移植性の妨げになります。もう 1 つの方法は、ルーチンを別のスキーマにコピーするやり方です。しかし、これはメンテナンスの妨げになります。お勧めする方法として、AUTHID 句を使用します。これによって、ストアド・プロシージャおよび SQL メソッドをその実行者（現在のユーザー）の権限で実行できます。このような実行者権限ルーチンは、特定のスキーマにバインドされません。これはさまざまなユーザーが実行できます。定義者はどのユーザーが実行しているのかを知る必要はありません。

たとえば、次に示すバージョンのプロシージャ create_dept は、その実行者の権限で実行されます。

```
CREATE PROCEDURE create_dept (new_dname CHAR, new_loc CHAR)
    AUTHID CURRENT_USER AS
BEGIN
    INSERT INTO dept
        VALUES (deptno_seq.NEXTVAL, new_dname, new_loc);
END;
```

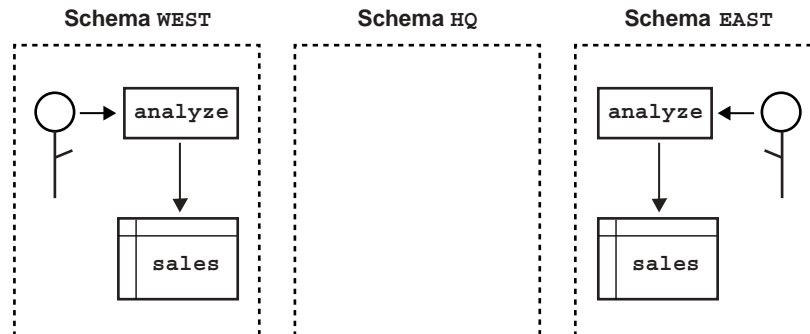
また、表 dept への未修飾の参照は、定義者ではなく、実行者のスキーマで解決されます。

実行者権限の利点

実行者権限ルーチンを使用すると、データ検索を集中化できます。これは、異なるスキーマにデータを格納するアプリケーションで特に便利です。このような場合、1 つのコード・ベースを使用して複数のユーザーが独自のデータを管理できます。

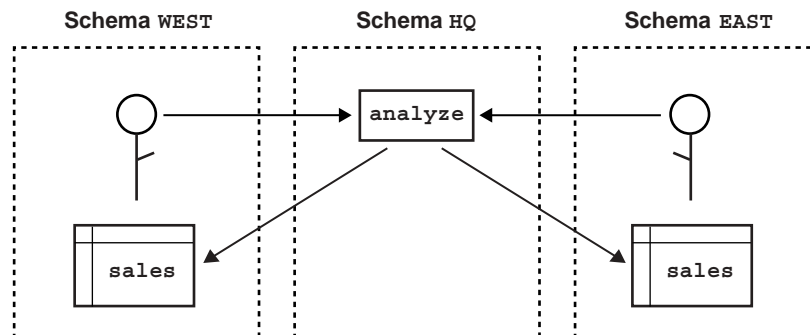
定義者権限プロシージャを使って売上を分析する会社の場合を考えてみます。ローカルの売上統計を出すには、プロシージャ `analyze` は各地のサイトに常駐する `sales` 表にアクセスする必要があります。このため、図 7-2 に示すように、プロシージャも各地のサイトに常駐する必要があります。これはメンテナンスの問題を引き起こします。

図 7-2 定義者権限の問題



この問題を解決するには、プロシージャ `analyze` の実行者権限バージョンを本社にインストールします。こうすると、図 7-3 に示すように、すべての地域のサイトで同じプロシージャを使用して、各自の `sales` 表を問い合わせることができます。

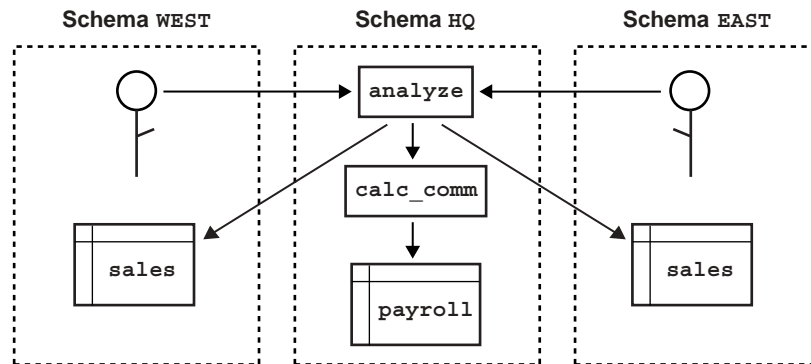
図 7-3 実行者権限による解決



実行者権限ルーチンを使用すると、機密データへのアクセスの制限もできます。本社が、プロシージャ `analyze` を使用して、売上のコミッションを計算し、中央の `payroll` 表を更新したいとします。

この場合、analyze の実行者には、従業員の給与やその他の機密データが格納されている payroll 表への直接アクセスを持たせるべきではないので、問題が発生します。図 7-4 に示すように、プロシージャ analyze に定義者権限プロシージャ calc_comm をコールさせて、この問題を解決します。これによって、payroll 表が更新されます。

図 7-4 間接アクセス



AUTHID 句の使用法

実行者権限をインプリメントするには、AUTHID 句を使って、ルーチンが定義者と実行者のどちらの権限で実行するかを指定します。またこの句は、外部参照（ルーチンの外側のオブジェクトへの参照）が定義者と実行者のどちらのスキーマで解決されるかも指定します。

AUTHID 句は、スタンドアロン・サブプログラム、パッケージ仕様部、またはオブジェクト型仕様部のヘッダーでだけ使用できます。次にヘッダー構文を示します。

```

-- stand-alone function
CREATE [OR REPLACE] FUNCTION [schema_name.]function_name
[(parameter_list)] RETURN datatype
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}

-- stand-alone procedure
CREATE [OR REPLACE] PROCEDURE [schema_name.]procedure_name
[(parameter_list)]
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}

-- package spec
CREATE [OR REPLACE] PACKAGE [schema_name.]package_name
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}
  
```

```
-- object type spec
CREATE [OR REPLACE] TYPE [schema_name.]object_type_name
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS} OBJECT
```

DEFINER はデフォルトのオプションです。パッケージまたはオブジェクト型では、AUTHID 句はすべてのルーチンに適用されます。

外部参照の解決方法

AUTHID CURRENT_USER を指定すると、実行時に実行者の権限がチェックされ、外部参照は実行者のスキーマで解決されます。ただし、これは次の文の外部参照だけに適用されます。

- SELECT、INSERT、UPDATE および DELETE データ操作文
- LOCK TABLE トランザクション制御文
- OPEN および OPEN-FOR カーソル制御文
- EXECUTE IMMEDIATE および OPEN-FOR-USING 動的 SQL 文
- DBMS_SQL.PARSE() を使用して解析された SQL 文

それ以外の文の場合は、コンパイル時に定義者の権限がチェックされ、外部参照は定義者のスキーマで解決されます。たとえば、次の代入文はパッケージ・ファンクションを参照します。この外部参照はプロシージャ reconcile の定義者のスキーマで解決されます。

```
CREATE PROCEDURE reconcile (acc_id IN INTEGER)
  AUTHID CURRENT_USER AS
  bal NUMBER;
BEGIN
  bal := bank_stuff.balance(acct_id);
  ...
END;
```

ネーム変換エラーの回避

実行者権限ルーチンの外部参照は、実行時に実行者のスキーマで解決されます。しかし、PL/SQL コンパイラはすべての参照をコンパイル時に解決する必要があります。そのため定義者は、自分のスキーマに、あらかじめテンプレート・オブジェクトを作成しておく必要があります。実行時に、テンプレート・オブジェクトと実オブジェクトは一致しなければなりません。そうでない場合はエラーか、または予期しない結果になります。

たとえば、ユーザー scott が、次のデータベース表とスタンドアロン・プロシージャを作成するとします。

```
CREATE TABLE emp (
  empno NUMBER(4),
  ename VARCHAR2(15));
/
CREATE PROCEDURE evaluate (my_empno NUMBER)
```

```

AUTHID CURRENT_USER AS
my_ename VARCHAR2(15);
BEGIN
  SELECT ename INTO my_ename FROM emp WHERE empno = my_empno;
  ...
END;
/

```

ここで、ユーザー `blake` が類似のデータベース表を作成し、プロシージャ `evaluate` をコールするとします。

```

CREATE TABLE emp (
  empno    NUMBER(4),
  ename     VARCHAR2(15),
  my_empno NUMBER(4)); -- note extra column
/
DECLARE
  ...
BEGIN
  ...
  scott.evaluate(7788);
END;
/

```

プロシージャ・コールはエラーを発生することなく実行されますが、ユーザー `blake` が作成した表の列 `my_empno` は無視します。これは、実行者のスキーマにある実際の表が、コンパイル時に使用されたテンプレート表と一致しないために発生します。

デフォルトのネーム変換の上書き

デフォルトの実行者定義の動作を変更したい場合があります。たとえばユーザー `scott` が、次のスタンドアロン・プロシージャを定義するとします。`SELECT` 文が外部ファンクションをコールするのに注意してください。この外部参照は、通常は実行者のスキーマで解決されます。

```

CREATE PROCEDURE calc_bonus (emp_id INTEGER)
AUTHID CURRENT_USER AS
  mid_sal REAL;
BEGIN
  SELECT median(sal) INTO mid_sal FROM emp;
  ...
END;

```

定義者のスキーマで参照を解決するには、次のように `CREATE SYNONYM` 文を使用して、ファンクションのパブリック・シノニムを作成します。

```

CREATE PUBLIC SYNONYM median FOR scott.median;

```

実行者が `median` という名前のファンクションまたはプライベート・シノニムを定義していないかぎり、これは有効です。

または、次のように参照を完全に修飾することができます。

```
BEGIN
    SELECT scott.median(sal) INTO mid_sal FROM emp;
    ...
END;
```

実行者が、`median` という名前のファンクションを含む `scott` というパッケージを定義していないかぎり、これは有効です。

EXECUTE 権限の付与

実行者権限ルーチンを直接コールするには、ユーザーはそのルーチンに対して `EXECUTE` 権限を持っている必要があります。権限を付与することによって、ユーザーに次のことを許可します。

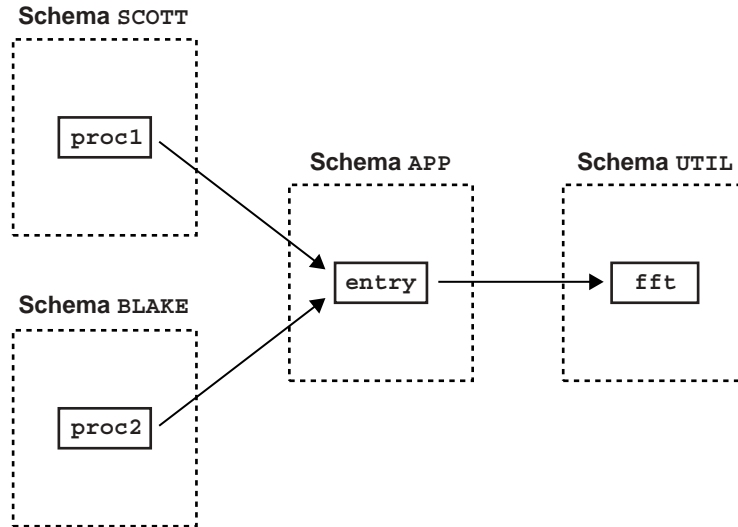
- ルーチンの直接のコール
- ルーチンをコールするファンクションおよびプロシージャのコンパイル

次に示すように、ユーザー `util` が、実行者権限ルーチン `fft` に対する `EXECUTE` 権限を、ユーザー `app` に付与するとします。

```
GRANT EXECUTE ON util.fft TO app;
```

ユーザー `app` は、ルーチン `fft` をコールするファンクションおよびプロシージャをコンパイルできるようになります。実行時には、このコールについての権限チェックは行われません。このため、[図 7-5](#) に示すように、ユーザー `util` は、`fft` を間接的にコールするユーザー 1 人ひとりに `EXECUTE` 権限を付与する必要はありません。

図 7-5 実行者権限ルーチンへの間接コール



ルーチン `util.fft` は、定義者権限ルーチン `app.entry` からだけ、直接コールされていることに注意してください。ユーザー `util` は、`EXECUTE` 権限をユーザー `app` だけに付与する必要があります。`util.fft` が実行されるとき、その実行者は、`app`、`scott`、または `blake` の可能性があります。

ビューとデータベース・トリガーの使用

ビュー式内で実行される実行者権限ルーチンの場合は、ビュー・ユーザーではなく、ビュー所有者が実行者とみなされます。たとえば次のように、ユーザー `scott` がビューを作成するとします。

```
CREATE VIEW payroll AS SELECT layout(3) FROM dual;
```

ファンクション `layout` は、常に、ビュー所有者であるユーザー `scott` の権限で実行されます。

この規則は、データベース・トリガーにも適用されます。

データベース・リンクの使用

データベース・リンクへの外部参照は通常の方法で解決されます。ただし、リンクはどのユーザーがリモート・データベースに接続されているかを判別します。

名前付きリンクでは、リンクで指定されているユーザー名を使用します。ユーザー `blake` が所有するルーチンが、次のデータベース・リンクを参照するとします。これは、どのユーザーがルーチンをコールするかに関係なく、ユーザー `scott` としてシカゴのデータベースに接続します。

```
CREATE DATABASE LINK chicago CONNECT TO scott
IDENTIFIED BY tiger USING connect_string;
```

無名リンクでは、セッション・ユーザー名を使用します。ユーザー `blake` が所有するルーチンが、次のデータベース・リンクを参照するとします。ユーザー `scott` がログインしていて、そのルーチンをコールしていれば、ユーザー `scott` としてアトランタのデータベースに接続します。

```
CREATE DATABASE LINK atlanta USING connect_string;
```

権限付きリンクでは、実行者のユーザー名を使用します。ユーザー `blake` が所有する実行者権限ルーチンが、次のデータベース・リンクを参照するとします。グローバルなユーザー `scott` がそのルーチンをコールしていれば、現在のユーザーであるユーザー `scott` として、ダラスのデータベースに接続します。

```
CREATE DATABASE LINK dallas CONNECT TO CURRENT_USER
USING connect_string;
```

これが定義者権限ルーチンだった場合、現在のユーザーは `blake` になります。このため、ルーチンはグローバルなユーザー `blake` としてダラスのデータベースに接続することになります。

インスタンス・メソッドの起動

実行者権限インスタンス・メソッドは、インスタンスの作成者ではなく、実行者の権限で実行します。Person が実行者権限オブジェクト型で、ユーザー `scott` が、型 Person のオブジェクトである `p1` を作成するとします。ユーザー `blake` が、オブジェクト `p1` で操作するためのインスタンス・メソッド `change_job` をコールした場合、メソッドの実行者は、`scott` ではなく、`blake` です。次の例を考えてみます。

```
-- user blake creates a definer-rights procedure
CREATE PROCEDURE reassign (p Person, new_job VARCHAR2) AS
BEGIN
    -- user blake calls method change_job, so the
    -- method executes with the privileges of blake
    p.change_job(new_job);
END;
/
```

```
-- user scott passes a Person object to the procedure
DECLARE
    p1 Person;
BEGIN
    p1 := Person(...);
    blake.reassign(p1, 'CLERK');
END;
/
```

再帰

再帰はアルゴリズムの設計を単純化する強力な手法です。一般に、再帰とは自己参照を意味します。再帰的な数列の個々の項は、それ以前の項に計算式を適用することで得られます。最初はウサギのコロニーの成長をモデル化するために使われたフィボナッチ数列（1, 1, 2, 3, 5, 8, 13, 21, ...）がその一例です。この数列では、2 番以降の各項が、すぐ前の 2 つの項の合計になっています。

再帰定義では、それ自身の単純なバージョンから新しいバージョンが定義されます。 n の階乗（ $n!$ 、 $1 \sim n$ のすべての整数の積）の定義を考えてみます。

```
n! = n * (n - 1)!
```

再帰サブプログラム

再帰サブプログラムとは、自分自身をコールするサブプログラムのことです。再帰的コールを、同じタスクを持つ他のサブプログラムへのコールと考えてみてください。再帰的コールが行われるたびに、パラメータ、変数、カーソル、および例外など、そのサブプログラムで宣言されているすべての項目の新しいインスタンスが作成されます。また、再帰を繰り返して進む過程の各レベルで、SQL 文の新しいインスタンスが作られます。

再帰的コールを入れる位置には注意してください。カーソル FOR ループの中や、OPEN 文と CLOSE 文の間に入れると、コールのたびに新しいカーソルがオープンされてしまいます。その結果、Oracle の初期化パラメータ OPEN_CURSORS で設定された上限を超えてしまう可能性があります。

再帰サブプログラムには、再帰的コールへ導くパスとそうでないパスの、少なくとも 2 つのパスが必要です。終了条件へ導くパスが少なくとも 1 つは必要だということです。終了条件へのパスがないと、理論上、再帰が永遠に続くことになります。実際に再帰サブプログラムが無限退行に入り込んだ場合は、最終的にメモリーが足りなくなり、PL/SQL は事前定義の例外 STORAGE_ERROR を呼び出します。

例 1

プログラミング上の必要から、ある条件が満たされるまで一連の文を繰り返さなければならぬ場合があります。これを解決するには、反復または再帰を使います。問題がそれ自身の単純なバージョンに分解できる場合は、再帰を使用します。たとえば、 $3!$ は次のようにして評価できます。

```
0! = 1  -- by definition
1! = 1 * 0! = 1
2! = 2 * 1! = 2
3! = 3 * 2! = 6
```

このアルゴリズムをインプリメントする場合は、次のようなファンクションを記述すると正の整数の階乗を戻すことができます。

```
FUNCTION fac (n POSITIVE) RETURN INTEGER IS -- returns n!
BEGIN
    IF n = 1 THEN -- terminating condition
        RETURN 1;
    ELSE
        RETURN n * fac(n - 1); -- recursive call
    END IF;
END fac;
```

再帰コールのたびに n の値は減分されます。 n が 1 になった時点で再帰は停止します。

例 2

次に示すプロシージャは、特定のマネージャに属するスタッフを探すものです。プロシージャは、マネージャの従業員番号を表す `mgr_no` と、マネージャの部門組織の職階を表す `tier` という 2 つの仮パラメータを宣言します。1 番目の職階は、このマネージャの直属のスタッフ・メンバーです。

```
PROCEDURE find_staff (mgr_no NUMBER, tier NUMBER := 1) IS
    boss_name CHAR(10);
    CURSOR c1 (boss_no NUMBER) IS
        SELECT empno, ename FROM emp WHERE mgr = boss_no;
BEGIN
    /* Get manager's name. */
    SELECT ename INTO boss_name FROM emp WHERE empno = mgr_no;
    IF tier = 1 THEN
        INSERT INTO staff -- single-column output table
            VALUES (boss_name || ' manages the staff');
    END IF;
    /* Find staff members who report directly to manager. */
    FOR ee IN c1 (mgr_no) LOOP
        INSERT INTO staff
```



```

VALUES (boss_name || ' manages ' || ee.ename
        || ' on tier ' || TO_CHAR(tier));
/* Drop to next tier in organization. */
find_staff(ee.empno, tier + 1); -- recursive call
END LOOP;
COMMIT;
END;
```

このプロシージャは、コールされたときに mgr_n の値を受け取りますが、tier の値としてはデフォルト値を使います。たとえば、このプロシージャは次のようにしてコールできます。

```
find_staff(7839);
```

プロシージャは mgr_no を、カーソル FOR ループの中のカーソルに渡します。このループは、組織の中の職階を順次下がっていった、スタッフ・メンバーを探します。再帰コールのたびに、FOR ループの新しいインスタンスが作られ、別のカーソルがオープンされます。ただし、以前のカーソルは結果セットの中の次の行にとどまっています。

行の取り出しができなかった場合、カーソルは自動的にクローズされ、FOR ループが終了します。再帰コールは FOR ループの中で起こっているため、再帰は終了します。最初のコールとは異なり、個々の再帰コールではプロシージャに 2 番目の実パラメータ（次の職階）が渡されます。

ここで示したのは再帰の例で、集合を扱う SQL 文の使用例とはいえません。この再帰プロシージャと、同じ処理を実行する次の SQL 文のパフォーマンスを比較してみてください。

```

INSERT INTO staff
SELECT PRIOR ename || ' manages ' || ename
       || ' on tier ' || TO_CHAR(LEVEL - 1)
FROM emp
START WITH empno = 7839
CONNECT BY PRIOR empno = mgr;
```

SQL 文の方がはるかに高速です。ただし、プロシージャの方が柔軟性は高くなります。たとえば、複数表の間合せは CONNECT BY 句を持つことができません。また、プロシージャとは異なり、SQL 文を変更して結合できません。（結合とは、複数のデータベース表の行を組み合せることです。）さらに、プロシージャでは、1 つの SQL 文ではできないような方法で、データを処理できます。

相互再帰

相互再帰とは、直接または間接に、サブプログラムが互いにコールし合うことです。たとえば、次の例では、数値が奇数か偶数かを判断するブール・ファンクション `odd` と `even` が、互いに直接コールし合っています。`even` が `odd` をコールする時点では `odd` はまだ宣言されていないので、`odd` を前方宣言します。(7-9 ページの「[前方宣言](#)」を参照。)

```
FUNCTION odd (n NATURAL) RETURN BOOLEAN; -- forward declaration

FUNCTION even (n NATURAL) RETURN BOOLEAN IS
BEGIN
    IF n = 0 THEN
        RETURN TRUE;
    ELSE
        RETURN odd(n - 1); -- mutually recursive call
    END IF;
END even;

FUNCTION odd (n NATURAL) RETURN BOOLEAN IS
BEGIN
    IF n = 0 THEN
        RETURN FALSE;
    ELSE
        RETURN even(n - 1); -- mutually recursive call
    END IF;
END odd;
```

正の整数 `n` が `odd` または `even` に渡されると、この 2 つのファンクションは交互にコール合います。コールのたびに `n` の値は減分されます。`n` は最終的にゼロになり、最後のコールは `TRUE` または `FALSE` を戻します。たとえば、数値 4 を `odd` に渡すと、次のような一連のコールがなされます。

```
odd(4)
even(3)
odd(2)
even(1)
odd(0) -- returns FALSE
```

また、数値 4 を `even` に渡すと、コールは次のようになります。

```
even(4)
odd(3)
even(2)
odd(1)
even(0) -- returns TRUE
```

再帰と反復

再帰は、反復とは異なり、PL/SQLでのプログラミングに必須ではありません。再帰を使って解決できる問題は、必ず反復でも解決できます。したがって、通常は、再帰バージョンのサブプログラムより、反復バージョンのサブプログラムの方が設計が簡単です。ただし、一般的に言って、再帰バージョンの方が構造が単純で小さいため、デバッグは簡単です。 n 番目のフィボナッチ数を求める次のファンクションを比較してみてください。

```
-- recursive version
FUNCTION fib (n POSITIVE) RETURN INTEGER IS
BEGIN
    IF (n = 1) OR (n = 2) THEN
        RETURN 1;
    ELSE
        RETURN fib(n - 1) + fib(n - 2);
    END IF;
END fib;

-- iterative version
FUNCTION fib (n POSITIVE) RETURN INTEGER IS
    pos1 INTEGER := 1;
    pos2 INTEGER := 0;
    cum INTEGER;
BEGIN
    IF (n = 1) OR (n = 2) THEN
        RETURN 1;
    ELSE
        cum := pos1 + pos2;
        FOR i IN 3..n LOOP
            pos2 := pos1;
            pos1 := cum;
            cum := pos1 + pos2;
        END LOOP;
        RETURN cum;
    END IF;
END fib;
```

fib の再帰バージョンは、反復バージョンよりも簡潔です。ところが、反復バージョンは速い上にメモリーの消費量が少ないので、効率では優れています。再帰的コールの方が、実行のたびに時間とメモリーを余分に使用するからです。再帰的コールの数が多くなるほど、効率に差がつくことになります。それでも、再帰的コールの数が少ない場合は、可読性の点から、再帰バージョンを選択します。

Goods which are not shared are not goods.—Fernando de Rojas

この章では、互いに関連する PL/SQL プログラミングの構成体を 1 つのパッケージにまとめる方法を示します。構成体には、プロシージャの集まりや、型定義と変数宣言の集まりなどが考えられます。たとえば、人事パッケージには入社用のプロシージャと解雇用のプロシージャを入れることができます。一度作られた汎用パッケージは、コンパイルされて Oracle データベースに格納され、複数のアプリケーションで内容を共用できます。

主なトピック

[パッケージについて](#)

[パッケージの利点](#)

[パッケージの仕様部](#)

[パッケージ本体](#)

[例](#)

[プライベート項目とパブリック項目](#)

[オーバーロード](#)

[パッケージ STANDARD](#)

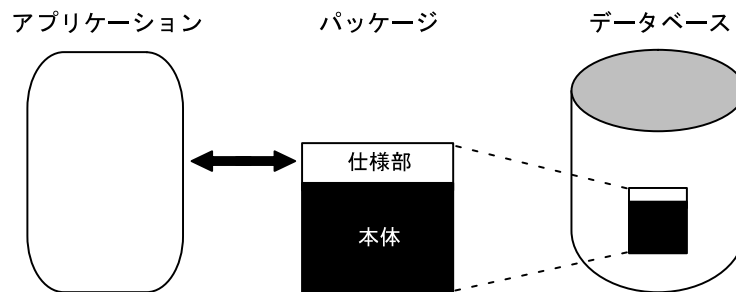
[製品固有のパッケージ](#)

パッケージについて

パッケージとは、論理的に関連する PL/SQL の型および項目、サブプログラムをグループにまとめたスキーマ・オブジェクトのことです。通常、パッケージは仕様部と本体の 2 つの部分で構成されますが、本体が不要な場合もあります。仕様部はアプリケーションへのインタフェースです。ここでは、使用する型および変数、定数、例外、カーソル、サブプログラムなどを宣言します。本体ではカーソルとサブプログラムを完全に定義し、仕様部をインプリメントします。

図 8-1 に示すように、仕様部は操作インタフェース、本体は "ブラック・ボックス" と考えることができます。パッケージ本体は、パッケージへのインタフェース (パッケージ仕様部) を変更せずに、デバッグ、拡張または置換ができます。

図 8-1 パッケージのインタフェース



パッケージを作成するには、CREATE PACKAGE 文を使用します。この文は SQL*Plus から対話式で実行できます。次に構文を示します。

```
CREATE [OR REPLACE] PACKAGE package_name
  [AUTHID {CURRENT_USER | DEFINER}] {IS | AS}
  [type_definition [type_definition] ...]
  [cursor_spec [cursor_spec] ...]
  [item_declaration [item_declaration] ...]
  [{subprogram_spec | call_spec} [{subprogram_spec | call_spec}]...]
END [package_name];

[CREATE [OR REPLACE] PACKAGE BODY package_name {IS | AS}
  [type_definition [type_definition] ...]
  [cursor_body [cursor_body] ...]
  [item_declaration [item_declaration] ...]
  [{subprogram_spec | call_spec} [{subprogram_spec | call_spec}]...]
[BEGIN
  sequence_of_statements]
END [package_name];]
```

仕様部には、アプリケーションから見えるパブリックな宣言を入れます。本体には、インプリメンテーションの細部と、アプリケーションからは隠されているプライベートな宣言を入れます。パッケージ本体の宣言部の後には、オプションの初期化部があります。ここには、一般にパッケージ変数を初期化する文が置かれています。

AUTHID 句は、すべてのパッケージ・サブプログラムがその定義者（デフォルト）と実行者のどちらの権限で実行するか、およびスキーマ・オブジェクトへの未修飾の参照が定義者と実行者のどちらのスキーマで解決されるかを決定します。詳細は、7-29 ページの「[実行者権限と定義者権限](#)」を参照してください。

コール仕様部を使用すると、Oracle データ・ディクショナリ内の外部 C ファンクションまたは Java メソッドを発行できます。コール仕様部は、対応する SQL 版に名前、パラメータ型および戻り型をマップすることによって、ルーチンを発行します。Java コール仕様部を作成する方法は、『Oracle8i Java ストアド・プロシージャ開発者ガイド』を参照してください。C コール仕様部を作成する方法は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

次の例では、1 つのレコード型とカーソルと 2 つの人事プロシージャをパッケージ化しています。プロシージャ hire_employee では、データベース順序 empno_seq とファンクション SYSDATE を利用して、それぞれ新しい従業員番号と入社日を挿入しています。

```
CREATE OR REPLACE PACKAGE emp_actions AS -- spec
    TYPE EmpRecTyp IS RECORD (emp_id INTEGER, salary REAL);
    CURSOR desc_salary RETURN EmpRecTyp;
    PROCEDURE hire_employee (
        ename VARCHAR2,
        job   VARCHAR2,
        mgr   NUMBER,
        sal   NUMBER,
        comm  NUMBER,
        deptno NUMBER);
    PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;
```

```
CREATE OR REPLACE PACKAGE BODY emp_actions AS -- body
    CURSOR desc_salary RETURN EmpRecTyp IS
        SELECT empno, sal FROM emp ORDER BY sal DESC;
    PROCEDURE hire_employee (
        ename VARCHAR2,
        job   VARCHAR2,
        mgr   NUMBER,
        sal   NUMBER,
        comm  NUMBER,
        deptno NUMBER) IS
```

```
BEGIN
    INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename, job,
        mgr, SYSDATE, sal, comm, deptno);
END hire_employee;
PROCEDURE fire_employee (emp_id NUMBER) IS
BEGIN
    DELETE FROM emp WHERE empno = emp_id;
END fire_employee;
END emp_actions;
```

アプリケーションから参照およびアクセスできるのは、パッケージの仕様部の宣言だけです。パッケージ本体のインプリメンテーション詳細は隠ぺいされ、アクセスできません。そのため、コールする側のプログラムを再コンパイルしなくても、本体（インプリメンテーション）を変更できます。

パッケージの利点

パッケージの利点には、モジュール性、アプリケーション設計の容易さ、情報隠ぺい、機能の追加、パフォーマンスの向上などがあります。

モジュール性

パッケージを使うと、論理的に関連した型、項目、およびサブプログラムを、名前付きの PL/SQL モジュールにカプセル化できます。個々のパッケージは理解しやすく、パッケージ間のインタフェースは単純かつ明快で、定義がしっかりしています。これはアプリケーション開発に役立ちます。

アプリケーションの設計の容易さ

アプリケーション設計の最初の段階では、パッケージの仕様部に含まれるインタフェース情報だけが必要です。仕様部は本体がなくてもコーディングし、コンパイルできます。仕様部のコンパイルが終了すると、そのパッケージを参照するストアド・サブプログラムもコンパイルできます。アプリケーション作成の最終段階になるまで、パッケージ本体を完全に定義する必要はありません。

情報隠ぺい

パッケージを使うと、個々の型、項目、およびサブプログラムについて、それがパブリック（可視でアクセス可能）なのか、またはプライベート（隠されていてアクセス不可）なのかを指定できます。たとえば、パッケージに含まれる 4 つのサブプログラムのうち、3 つをパブリック、1 つをプライベートにすることもできます。パッケージはプライベートなサブプログラムの定義を隠ぺいするので、定義が変更された場合も（アプリケーションではなく）パッケージだけが影響を受けます。このため、メンテナンスや機能拡張が簡単に実施できます。また、インプリメンテーション上の細部をユーザーから隠ぺいすることで、パッケージの整合性を維持できます。

機能の追加

パッケージ化されたパブリック変数およびカーソルは、セッションを通じて存続します。このため、同じ環境の中で実行するすべてのサブプログラムで共有できます。また、トランザクション間でデータを保持したい場合も、データベースにデータを格納する必要があります。

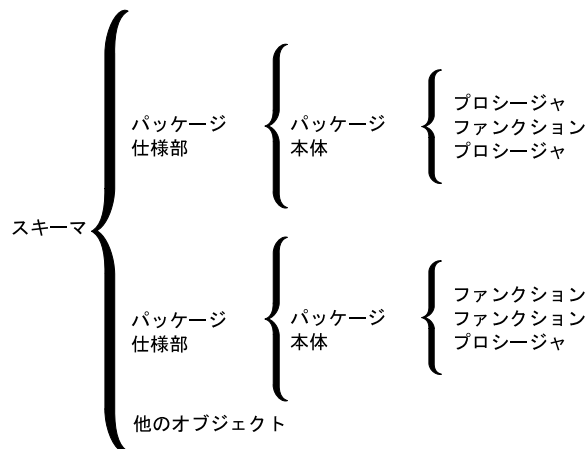
パフォーマンスの向上

パッケージ・サブプログラムを初めてコールすると、パッケージ全体がメモリーにロードされます。このため、それ以降にパッケージ中の関連するサブプログラムをコールしても、ディスクへの入出力はありません。さらに、パッケージ化すると互いに依存することがなくなるので、不要な再コンパイルを避けることができます。たとえば、パッケージ・ファンクションの定義を変更した場合でも、コールする側のサブプログラムはパッケージ本体に依存していないため、Oracle はコールする側のサブプログラムを再コンパイルする必要がありません。

パッケージの仕様部

パッケージの仕様部にはパブリック宣言が入っています。これらの宣言の有効範囲は、データベース・スキーマに対してローカルで、パッケージに対してグローバルです。したがって、宣言された項目は、ユーザーのアプリケーションからも、パッケージ内のどの場所からもアクセスできます。図 8-2 に有効範囲を示します。

図 8-2 パッケージの有効範囲



仕様部には、アプリケーションが利用できるパッケージ・リソースのリストがあります。アプリケーションがリソースを使うために必要な情報は、すべて仕様部の中にあります。たと

例えば、次の宣言は、`fac` という名前のファンクションが `INTEGER` 型の引数を 1 つ取り、`INTEGER` 型の値を戻すことを示しています。

```
FUNCTION fac (n INTEGER) RETURN INTEGER; -- returns n!
```

ファンクションのコールに必要な情報はこれだけです。ユーザーは `fac` の下位のインプリメンテーションのこと（それが反復を利用しているのか、再帰を利用しているのかなど）を考える必要がありません。

下位のインプリメンテーションすなわち定義を持つのは、サブプログラムとカーソルだけです。したがって、仕様部で宣言されているのが型、定数、変数、例外、およびコール仕様部だけならばパッケージ本体は不要です。このような本体なしのパッケージの例を次に示します。

```
-- a bodiless package
CREATE PACKAGE trans_data AS
    TYPE TimeRec IS RECORD (
        minutes SMALLINT,
        hours   SMALLINT);
    TYPE TransRec IS RECORD (
        category VARCHAR2,
        account  INTEGER,
        amount   REAL,
        time_of  TimeRec);
    minimum_balance  CONSTANT REAL := 10.00;
    number_processed INTEGER;
    insufficient_funds EXCEPTION;
END trans_data;
```

型および定数、変数、例外は下位のインプリメンテーションを持たないため、パッケージ `trans_data` は本体を必要としません。このようなパッケージを利用すると、セッションを通じて存続するグローバル変数（サブプログラムやデータベース・トリガーで利用できる）を定義できます。

パッケージの内容の参照

パッケージの仕様部で宣言された型、項目、サブプログラム、およびコール仕様部を参照するときには、次のようにドット表記法を使用します。

```
package_name.type_name
package_name.item_name
package_name.subprogram_name
package_name.call_spec_name
```

パッケージ内容は、データベース・トリガー、ストアド・サブプログラム、3GL アプリケーション・プログラム、およびさまざまな Oracle Tools から参照できます。たとえば、パッケージ・プロシージャ `hire_employee` は次のようにして `SQL*Plus` からコールします。

```
SQL> CALL emp.actions.hire_employee('TATE', 'CLERK', ...);
```

次の例では、Pro*C プログラムに組み込まれた 無名の PL/SQL ブロックから同じプロシージャをコールしています。実パラメータ emp_name と job_title はホスト変数（ホスト環境で宣言された変数）です。

```
EXEC SQL EXECUTE
  BEGIN
    emp_actions.hire_employee(:emp_name, :job_title, ...);
```

制限

リモート・パッケージ変数は、直接的にも間接的にも参照できません。たとえば、次のようなプロシージャは、パラメータの初期化句の中でパッケージ変数を参照するため、リモートでコールできません。

```
CREATE PACKAGE random AS
  seed NUMBER;
  PROCEDURE initialize (starter IN NUMBER := seed, ...);
```

また、パッケージ内ではホスト変数を参照できません。

パッケージ本体

パッケージ本体はパッケージの仕様部をインプリメントします。つまり、パッケージ本体には、パッケージの仕様部で宣言されているすべてのカーソルとサブプログラムの定義が含まれています。パッケージ本体で定義されたサブプログラムをパッケージの外側からアクセスするためには、その指定がパッケージの仕様部に存在しなければならないことに注意してください。

サブプログラムの仕様部と本体を一致させるために、PL/SQL は、それらのヘッダーをトークンごとに比較します。このため、空白を除いて、ヘッダーは一語一語が一致していなければなりません。一致していない場合は、PL/SQL は例外をコールします。次に例を示します。

```
CREATE PACKAGE emp_actions AS
  ...
  PROCEDURE calc_bonus (date_hired emp.hiredate%TYPE, ...);
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
  ...
  PROCEDURE calc_bunus (date_hired DATE, ...) IS
    -- parameter declaration raises an exception because 'DATE'
    -- does not match 'emp.hiredate%TYPE' word for word
  BEGIN ... END;
END emp_actions;
```

パッケージ本体には、パッケージの内部動作に必要な型や項目を定義するプライベート宣言を入れることもできます。これらの宣言の有効範囲は、パッケージ本体に対してローカルです。このため、宣言された型と項目はパッケージ本体の中からでなければアクセスできません。パッケージの仕様部とは異なり、パッケージ本体の宣言部にはサブプログラムの本体を置くことができます。

パッケージ本体の宣言部の後には、オプションの初期化部があります。ここには、一般にパッケージの中で宣言済みの変数を初期化する文がいくつか置かれています。

サブプログラムとは異なり、パッケージをコールすることもパッケージにパラメータを渡すこともできないので、パッケージの初期化部にはあまり意味がありません。このため、パッケージの初期化部は、パッケージが初めて参照されたときに一度だけ実行されます。

すでに説明したように、仕様部で宣言されているのが型、定数、変数、例外、およびコール仕様部だけならばパッケージ本体は不要です。ただしその場合でも、パッケージ本体を使って、パッケージ仕様部で宣言した項目を初期化できます。

例

次に示す `emp_actions` という名前のパッケージを例に取って考えます。パッケージの仕様部では、次のような型、項目、およびサブプログラムを宣言します。

- 型 `EmpRecTyp` および `DeptRecTyp`
- カーソル `desc_salary`
- 例外 `salary_missing`
- ファンクション `hire_employee`、`nth_highest_salary`、`rank`
- プロシージャ `fire_employee` および `raise_salary`

パッケージを作ると、そのパッケージの型を参照したり、サブプログラムをコールしたり、カーソルを利用したり、例外をコールしたりするアプリケーションを開発できます。パッケージを作ると、そのパッケージは Oracle データベースに格納され、さまざまな用途に利用されます。

```
CREATE PACKAGE emp_actions AS
    /* Declare externally visible types, cursor, exception. */
    TYPE EmpRecTyp IS RECORD (emp_id INTEGER, salary REAL);
    TYPE DeptRecTyp IS RECORD (dept_id INTEGER, location VARCHAR2);
    CURSOR desc_salary RETURN EmpRecTyp;
    salary_missing EXCEPTION;
```

```
/* Declare externally callable subprograms. */
FUNCTION hire_employee (
    ename VARCHAR2,
    job   VARCHAR2,
    mgr   NUMBER,
    sal   NUMBER,
    comm  NUMBER,
    deptno NUMBER) RETURN INTEGER;
PROCEDURE fire_employee (emp_id INTEGER);
PROCEDURE raise_salary (emp_id INTEGER, amount NUMBER);
FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRecTyp;
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
    number_hired INTEGER; -- visible only in this package

/* Fully define cursor specified in package. */
CURSOR desc_salary RETURN EmpRecTyp IS
    SELECT empno, sal FROM emp ORDER BY sal DESC;

/* Fully define subprograms specified in package. */
FUNCTION hire_employee (
    ename VARCHAR2,
    job   VARCHAR2,
    mgr   NUMBER,
    sal   NUMBER,
    comm  NUMBER,
    deptno NUMBER) RETURN INTEGER IS
    new_empno INTEGER;
BEGIN
    SELECT empno_seq.NEXTVAL INTO new_empno FROM dual;
    INSERT INTO emp VALUES (new_empno, ename, job,
        mgr, SYSDATE, sal, comm, deptno);
    number_hired := number_hired + 1;
    RETURN new_empno;
END hire_employee;

PROCEDURE fire_employee (emp_id INTEGER) IS
BEGIN
    DELETE FROM emp WHERE empno = emp_id;
END fire_employee;

PROCEDURE raise_salary (emp_id INTEGER, amount NUMBER) IS
    current_salary NUMBER;
```

```
BEGIN
    SELECT sal INTO current_salary FROM emp
        WHERE empno = emp_id;
    IF current_salary IS NULL THEN
        RAISE salary_missing;
    ELSE
        UPDATE emp SET sal = sal + amount WHERE empno = emp_id;
    END IF;
END raise_salary;

FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRecTyp IS
    emp_rec EmpRecTyp;
BEGIN
    OPEN desc_salary;
    FOR i IN 1..n LOOP
        FETCH desc_salary INTO emp_rec;
    END LOOP;
    CLOSE desc_salary;
    RETURN emp_rec;
END nth_highest_salary;

/* Define local function, available only in package. */
FUNCTION rank (emp_id INTEGER, job_title VARCHAR2)
    RETURN INTEGER IS
/* Return rank (highest = 1) of employee in a given
   job classification based on performance rating. */
    head_count INTEGER;
    score      NUMBER;
BEGIN
    SELECT COUNT(*) INTO head_count FROM emp
        WHERE job = job_title;
    SELECT rating INTO score FROM reviews
        WHERE empno = emp_id;
    score := score / 100; -- maximum score is 100
    RETURN (head_count + 1) - ROUND(head_count * score);
END rank;

BEGIN -- initialization part starts here
    INSERT INTO emp_audit VALUES (SYSDATE, USER, 'EMP_ACTIONS');
    number_hired := 0;
END emp_actions;
```

パッケージの初期化部は、パッケージが初めて参照されたときに一度だけ実行されることに注意してください。このため、上の例の INSERT 文では、データベース表 emp_audit に挿入される行は 1 行だけです。また、変数 number_hired は一度しか初期化されません。

プロシージャ `hire_employee` がコールされるたびに、変数 `number_hired` が更新されます。しかし、`number_hired` が保持しているカウントは各セッションによって異なります。つまり、カウントは全ユーザーが処理した数ではなく、1人のユーザーが処理した新しい従業員の数を反映します。

次の例では、いくつかの一般的な銀行トランザクションをパッケージ化しています。営業時間の終了後も、現金自動預け払い機で出金および入金のトランザクションが入力され、翌朝になってから口座に適用されると考えています。

```
CREATE PACKAGE bank_transactions AS
  /* Declare externally visible constant. */
  minimum_balance CONSTANT NUMBER := 100.00;
  /* Declare externally callable procedures. */
  PROCEDURE apply_transactions;
  PROCEDURE enter_transaction (
    acct NUMBER,
    kind CHAR,
    amount NUMBER);
END bank_transactions;

CREATE PACKAGE BODY bank_transactions AS
  /* Declare global variable to hold transaction status. */
  new_status VARCHAR2(70) := 'Unknown';

  /* Use forward declarations because apply_transactions
     calls credit_account and debit_account, which are not
     yet declared when the calls are made. */
  PROCEDURE credit_account (acct NUMBER, credit REAL);
  PROCEDURE debit_account (acct NUMBER, debit REAL);

  /* Fully define procedures specified in package. */
  PROCEDURE apply_transactions IS
    /* Apply pending transactions in transactions table
       to accounts table. Use cursor to fetch rows. */
    CURSOR trans_cursor IS
      SELECT acct_id, kind, amount FROM transactions
        WHERE status = 'Pending'
        ORDER BY time_tag
        FOR UPDATE OF status; -- to lock rows
  BEGIN
    FOR trans IN trans_cursor LOOP
      IF trans.kind = 'D' THEN
        debit_account(trans.acct_id, trans.amount);
      ELSIF trans.kind = 'C' THEN
        credit_account(trans.acct_id, trans.amount);
      END IF;
    END LOOP;
  END apply_transactions;
END bank_transactions;
```

```
ELSE
    new_status := 'Rejected';
END IF;
UPDATE transactions SET status = new_status
    WHERE CURRENT OF trans_cursor;
END LOOP;
END apply_transactions;

PROCEDURE enter_transaction (
/* Add a transaction to transactions table. */
    acct    NUMBER,
    kind    CHAR,
    amount  NUMBER) IS
BEGIN
    INSERT INTO transactions
        VALUES (acct, kind, amount, 'Pending', SYSDATE);
END enter_transaction;

/* Define local procedures, available only in package. */
PROCEDURE do_journal_entry (
/* Record transaction in journal. */
    acct    NUMBER,
    kind    CHAR,
    new_bal NUMBER) IS
BEGIN
    INSERT INTO journal
        VALUES (acct, kind, new_bal, sysdate);
    IF kind = 'D' THEN
        new_status := 'Debit applied';
    ELSE
        new_status := 'Credit applied';
    END IF;
END do_journal_entry;

PROCEDURE credit_account (acct NUMBER, credit REAL) IS
/* Credit account unless account number is bad. */
    old_balance NUMBER;
    new_balance NUMBER;
BEGIN
    SELECT balance INTO old_balance FROM accounts
        WHERE acct_id = acct
        FOR UPDATE OF balance; -- to lock the row
    new_balance := old_balance + credit;
    UPDATE accounts SET balance = new_balance
        WHERE acct_id = acct;
    do_journal_entry(acct, 'C', new_balance);
```



```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        new_status := 'Bad account number';
    WHEN OTHERS THEN
        new_status := SUBSTR(SQLERRM,1,70);
END credit_account;

PROCEDURE debit_account (acct NUMBER, debit REAL) IS
/* Debit account unless account number is bad or
account has insufficient funds. */
    old_balance NUMBER;
    new_balance NUMBER;
    insufficient_funds EXCEPTION;
BEGIN
    SELECT balance INTO old_balance FROM accounts
        WHERE acct_id = acct
        FOR UPDATE OF balance; -- to lock the row
    new_balance := old_balance - debit;
    IF new_balance >= minimum_balance THEN
        UPDATE accounts SET balance = new_balance
            WHERE acct_id = acct;
        do_journal_entry(acct, 'D', new_balance);
    ELSE
        RAISE insufficient_funds;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        new_status := 'Bad account number';
    WHEN insufficient_funds THEN
        new_status := 'Insufficient funds';
    WHEN OTHERS THEN
        new_status := SUBSTR(SQLERRM,1,70);
    END debit_account;
END bank_transactions;
```

このパッケージでは初期化部を使っていません。

プライベート項目とパブリック項目

パッケージ emp_actions を再び取り上げてみます。パッケージ本体では、ゼロに初期化される変数 number_hired が宣言されています。emp_actions の仕様部で宣言された項目とは異なり、本体で宣言された項目はパッケージの中でしか使えません。このため、パッケージの外側の PL/SQL コードからは変数 number_hired を参照できません。このような項目はプライベートと呼ばれます。

しかし、例外 salary_missing など、emp_actions の仕様部で宣言された項目は、パッケージの外からも見えます。このため、例外 salary_missing はどの PL/SQL コードからも参照できます。このような項目はパブリックと呼ばれます。

セッションを通じて、または複数のトランザクションの間で維持しなければならない項目は、パッケージ本体の宣言部に置くようにしてください。たとえば、number_hired の値は hire_employee への複数のコールの間も保持されています。しかし、number_hired の値が各セッションに固有の値になることに注意が必要です。

パブリックにしなければならない項目は、パッケージ仕様部の中に置いてください。たとえば、bank_transactions のパッケージ仕様部で宣言された定数 minimum_balance は、パブリックで使用可能です。

注意：パッケージ化されたサブプログラムをリモートでコールすると、パッケージ全体のインスタンスが再度生成され、前の状態は消滅します。

オーバーロード

PL/SQL では、パッケージ化された複数のサブプログラムに同じ名前を付けることができます。サブプログラムで、異なるデータ型を持つパラメータを受け取れるようにしたい場合は、この方法が便利です。たとえば、次のパッケージでは journalize という名前の2つのプロシージャを定義しています。

```
CREATE PACKAGE journal_entries AS
    ...
    PROCEDURE journalize (amount NUMBER, trans_date VARCHAR2);
    PROCEDURE journalize (amount NUMBER, trans_date NUMBER );
END journal_entries;

CREATE PACKAGE BODY journal_entries AS
    ...
    PROCEDURE journalize (amount NUMBER, trans_date VARCHAR2) IS
    BEGIN
        INSERT INTO journal
            VALUES (amount, TO_DATE(trans_date, 'DD-MON-YYYY'));
    END journalize;
```

```
PROCEDURE journalize (amount NUMBER, trans_date NUMBER) IS
BEGIN
    INSERT INTO journal
        VALUES (amount, TO_DATE(trans_date, 'J'));
END journalize;
END journal_entries;
```

1 番目のプロシージャは trans_date を文字列として受け取りますが、2 番目のプロシージャは数値（ユリウス暦日付）として受け取ります。しかし、どちらのプロシージャもデータを適切に処理します。オーバーロードされたサブプログラムに適用される規則については、7-23 ページの「[オーバーロード](#)」を参照してください。

パッケージ STANDARD

STANDARD という名前のパッケージでは PL/SQL 環境を定義しています。このパッケージの仕様部では、型、例外およびサブプログラムをグローバルに宣言します。それらは、自動的にすべての PL/SQL プログラムで使用可能になります。たとえば、パッケージ STANDARD では、引数の絶対値を戻す ABS という名前の組み込みファンクションを宣言しています。

```
FUNCTION ABS (n NUMBER) RETURN NUMBER;
```

パッケージ STANDARD の内容は、アプリケーションから直接見るできます。ABS は、データベース・トリガー、ストアド・サブプログラム、3GL アプリケーション・プログラム、およびさまざまな Oracle Tools からコールできます。

PL/SQL プログラムの中で ABS を再宣言すると、ローカル宣言がグローバル宣言をオーバーライドします。ただし次に示すように、ドット表記法を使って組み込みファンクションをコールすることもできます。

```
... STANDARD.ABS(x) ...
```

ほとんどの組み込みファンクションはオーバーロードされています。たとえば、パッケージ STANDARD には次のような宣言があります。

```
FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN VARCHAR2;
```

PL/SQL は、仮パラメータと実パラメータの数とデータ型を比較して、どの TO_CHAR のコールかを判定します。

製品固有のパッケージ

Oracle といくつかの Oracle Tools では、PL/SQL のアプリケーションの構築を支援するために、製品固有のパッケージを用意しています。たとえば、Oracle には多数のユーティリティ・パッケージがあります。そのうちのいくつかを次に示します。詳細は、『Oracle8i Java ストアド・プロシージャ開発者ガイド』を参照してください。

DBMS_STANDARD

DBMS_STANDARD パッケージでは、アプリケーションと Oracle の対話を支援する言語機能を提供します。たとえば、プロシージャ `raise_application_error` を使うと、ユーザー定義のエラー・メッセージを発行できます。これを利用すると、アプリケーションに対してエラーを報告し、処理されない例外が戻されるのを避けることができます。例は、6-9 ページの「[raise_application_error の使用](#)」を参照してください。

DBMS_ALERT

DBMS_ALERT パッケージでは、データベース内の特定の値が変更されたときに、データベース・トリガーを使ってアプリケーションに警告できます。その警告は、トランザクション・ベースで、非同期です（つまり、警告はタイミング・メカニズムとは無関係に作動します）。たとえば、会社ではこのパッケージを使って、株や債券の取り引き価格が更新されたときに、投資ポートフォリオの値を更新できます。

DBMS_OUTPUT

DBMS_OUTPUT パッケージを使うと、PL/SQL ブロックとサブプログラムからの出力を表示できます。これによって、テストとデバッグが簡単になります。`put_line` プロシージャは、情報を SGA のバッファに出力します。この情報は、プロシージャ `get_line` をコールするか、SQL*Plus に `SERVEROUTPUT ON` を設定することによって表示します。たとえば、次のストアド・プロシージャを作成するとします。

```
CREATE PROCEDURE calc_payroll (payroll OUT NUMBER) AS
  CURSOR c1 IS SELECT sal, comm FROM emp;
BEGIN
  payroll := 0;
  FOR c1rec IN c1 LOOP
    c1rec.comm := NVL(c1rec.comm, 0);
    payroll := payroll + c1rec.sal + c1rec.comm;
  END LOOP;
  /* Display debug info. */
  DBMS_OUTPUT.PUT_LINE('Value of payroll: ' || TO_CHAR(payroll));
END;
```

次のコマンドを発行すると、SQL*Plus はプロシージャによって payroll に代入された値を表示します。

```
SQL> SET SERVEROUTPUT ON
SQL> VARIABLE num NUMBER
SQL> CALL calc_payroll(:num);
Value of payroll: 31225
```

DBMS_PIPE

DBMS_PIPE パッケージを使うと、名前付きパイプを通じて異なるセッション間で通信できます。(パイプとは、あるプロセスから他のプロセスに情報を渡すために使用するメモリーの領域のことです。) プロシージャ pack_message と send_message を使ってパイプの中にメッセージをパックし、同じインスタンスの中の別のセッションに送信できます。

パイプのもう一端では、プロシージャ receive_message と unpack_message を使って、メッセージを受信し、アンパック（読み込み）できます。名前付きパイプは、あらゆる点で便利です。たとえば、外部サーバーが情報を収集するルーチンを C 言語で書き、次にそれをパイプを通じて Oracle データベースの中に格納されたプロシージャに送信できます。

UTL_FILE

UTL_FILE パッケージを使うと、PL/SQL プログラムでオペレーティング・システム（OS）のテキスト・ファイルを読み書きできます。このパッケージは、OS の標準ストリーム・ファイル I/O の制限されたバージョン（OPEN、PUT、GET、CLOSE の操作を含む）を提供します。

テキスト・ファイルの読み書きを実行する場合は、fopen ファンクションをコールします。このファンクションは、それ以降のプロシージャ・コールで使うためのファイル・ハンドルを戻します。たとえば、プロシージャ put_line は、テキスト文字列と行終了文字をオープン・ファイルに書き込みます。プロシージャ get_line は、オープン・ファイルから出力バッファにテキストの行を読み込みます。

PL/SQL ファイル I/O は、クライアント側およびサーバー側の両方で使えます。ただし、サーバー側でのファイル・アクセスは、アクセス可能なディレクトリ・リストに明示的にリストされたディレクトリだけに制限されています。このリストは Oracle 初期化ファイルに格納されています。

UTL_HTTP

UTL_HTTP パッケージを使用すると、PL/SQL プログラムでハイパー・テキスト転送プロトコル（HTTP）のコールアウトを実行できます。これによって、データをインターネットから取り出すことも、Oracle Web Server カートリッジを呼び出すこともできます。このパッケージには 2 つのエントリ・ポイントがあり、各ポイントで URL（汎用リソース・ロケータ）を受け取り、指定されたサイトに接続し、要求されたデータを戻します。通常このデータはハイパー・テキスト・マークアップ言語（HTML）形式のものです。

指針

パッケージを作る場合は、別のアプリケーションで再利用できるように、なるべく一般性を持たせるようにしてください。すでに Oracle が提供している機能と重複する機能を持つパッケージを作らないように注意してください。

パッケージの仕様部はアプリケーションの設計を反映します。したがって、パッケージ本体の前にパッケージの仕様部を定義してください。仕様部に入れるのは、パッケージのユーザーに見えなければならない型、項目、およびサブプログラムだけにします。こうすれば、他の開発者がインプリメンテーション上の細部に基づくコードを書いて、パッケージの不適当な使い方をするのを防ぐことができます。

コードの変更時に必要な再コンパイルを削減するために、パッケージ仕様部に置く項目はできるかぎり少なくしておきます。そうすれば、パッケージ本体を変更しても、Oracle は依存するプロシージャを再コンパイルする必要がありません。しかし、パッケージ仕様部を変更すると、Oracle はそのパッケージを参照するすべてのストアド・サブプログラムを再コンパイルしなければならなくなります。

オブジェクト型

*... It next will be right
To describe each particular batch:
Distinguishing those that have feathers, and bite,
From those that have whiskers, and scratch.*—Lewis Carroll

オブジェクト指向プログラミングは、複雑なアプリケーションを作るのに必要なコストおよび時間を縮小できるので、急速に普及してきました。PL/SQL でのオブジェクト指向のプログラミングは、オブジェクト型をベースにしています。オブジェクト型は実社会の抽象的テンプレートを提供するので、モデル化ツールとして理想的です。さらに、ブラックボックスの中にカプセル化する手段を提供します。それはちょうど、種々の電子機器に差し込める統合された部品のようなものです。オブジェクト型をプログラム内に差し込むためには、それが何をするものかを知っているだけでよく、その仕組みまで知っている必要はありません。

主なトピック

- 抽象化の役割
- オブジェクト型とは？
- なぜオブジェクト型を使うか
- オブジェクト型の構造
- オブジェクト型を構成するさまざまな要素
- オブジェクト型の定義
- オブジェクトの宣言と初期化
- 属性へのアクセス
- コンストラクタのコール
- メソッドのコール
- オブジェクトの共有
- オブジェクトの操作

抽象化の役割

抽象化とは、実社会のエンティティを高いレベルで記述したもの、つまりモデルです。日常生活を管理できるのは、抽象化のおかげです。関係のない細かいところを切り捨てることにより、オブジェクト、イベント、または関連について推論しやすくなります。たとえば、車を運転するのに、エンジンが作動する仕組みまで知る必要はありません。ギアシフトおよびハンドル、アクセル、ブレーキで構成される簡単なインタフェースについて知るだけで、効果的に車を運転できます。ボンネットの下でなされていることの詳細は、日常の運転では重要ではありません。

抽象化は、プログラミングのかなめとなる概念です。たとえば、手続き（プロシージャ）を記述してそれにパラメータを渡すことにより、複雑なアルゴリズムをいちいち細かく考えることなく済ませるとき、手続き抽象化を使っていることになります。その手続きをコールするという単純な操作により、その実現方法についての細かい事情は隠されます。別の方法で実現したものを試すときも、その手続きを、名前とパラメータが同じ別の手続きに置き換えるだけですみます。抽象化のおかげで、その手続きをコールするプログラムは修正する必要がありません。

変数のデータ型を指定するときは、データ抽象化を使っています。データ型は、値の一定の集合と、それらの値に適用される操作の集合とを決定するものです。たとえば、`POSITIVE` 型の変数は正の整数だけを入れることができ、加算、減算、乗算などだけを実行できます。その変数を使うのに、`PL/SQL` が整数を格納する方法や算術演算を実行する方法について知る必要はありません。必要なのは、そのプログラミング・インタフェースを受け入れることだけです。

オブジェクト型は、ほとんどのプログラム言語に含まれているデータ型を一般化したものです。`PL/SQL` には、さまざまなスカラー型や複合データ型が用意されており、各データ型には特定の操作の集合が対応付けられています。スカラー型（`CHAR` など）には、内部的な要素はありません。複合型（`RECORD` など）には、別々に操作できる内部的な要素があります。`RECORD` 型のように、オブジェクト型は複合型です。しかし、その演算はユーザーが定義するものであり、事前定義済みではありません。

現在のところ、`PL/SQL` ではオブジェクト型の定義はできません。それらは `CREATE` 文を使って作り、Oracle データベースに格納して、たくさんのプログラムで共有できるようにしなければなりません。オブジェクト型を使用するプログラムは、クライアント・プログラムと呼ばれます。そこでは、オブジェクト型がデータを表す方法や演算の実現方法を知らなくても、オブジェクトを宣言したり操作したりできます。これにより、プログラムとオブジェクト型とを別々に作ったり、プログラムを修正することなくオブジェクト型の実現方法を変えたりできます。このように、オブジェクト型は手続き抽象化とデータ抽象化の両方をサポートします。

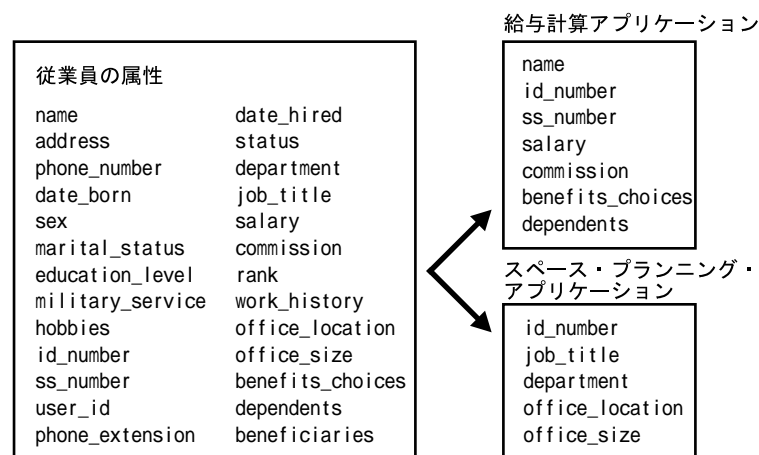
オブジェクト型とは？

オブジェクト型は、データを操作するのに必要なファンクションおよびプロシージャとともにデータ構造をカプセル化するユーザー定義の複合データ型です。データ構造を形成する変数は、属性と呼ばれます。オブジェクト型の動作を特徴付けるファンクションとプロシージャはメソッドと呼ばれます。

普通、オブジェクト（人、車、銀行口座など）のことを考えるとき、それにさまざまな属性や動作があるものとして考えます。たとえば、赤ちゃんには性、年齢、体重などの属性があり、食べる、飲む、寝るなどの動作があります。オブジェクト型でアプリケーションを記述するときは、このようなものの見方を取り入れることができます。

CREATE TYPE 文を使ってオブジェクト型を定義する場合、実世界のオブジェクトに対応する抽象テンプレートを作ります。テンプレートでは、アプリケーション環境でオブジェクトで必要となる属性と動作だけを指定します。たとえば、従業員には多くの属性がありますが、アプリケーションの必要を満たすのに必要な属性は、そのうちの少しだけです（図 9-1 を参照）。

図 9-1 形式はファンクションによって異なる



ボーナスを従業員に割り振るプログラムを作る必要があるとします。この問題の解決に、すべての社員属性は必要ありません。そこで、名前（name）、ID 番号（id_number）、部署（department）、肩書（job title）、給与（salary）、および等級（rank）という、この問題に関係のある特定の属性だけを取り出して抽象化した従業員を設計することにします。次に、その抽象化した社員を処理するのにどんな操作が必要かを考えます。たとえば、管理者が従業員の等級を変更するための操作が必要となります。

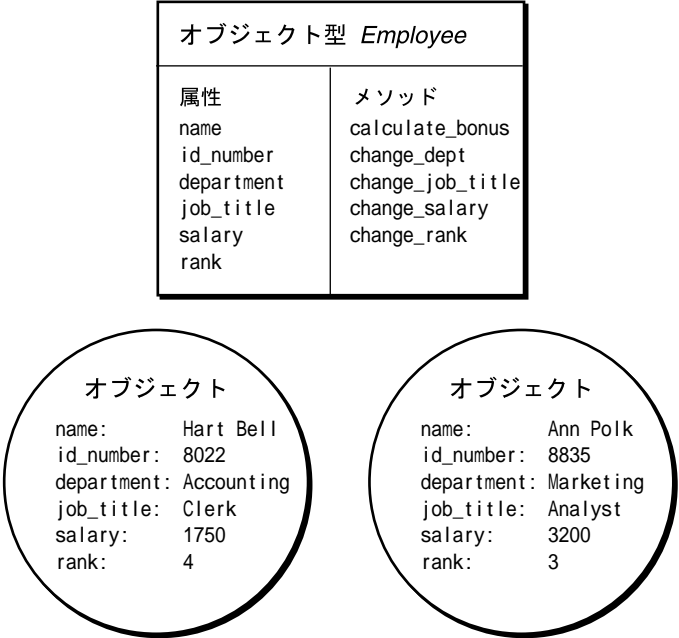
次に、データを表す変数（属性）の集合、および操作を実行するサブプログラム（メソッド）の集合を定義します。最後に、それらの属性およびメソッドをカプセル化して 1 つのオブジェクト型にします。

オブジェクト型とは？

属性の集合によって形成されるデータ構造体はパブリックです（クライアント・プログラムから参照できる）。しかし、正しいプログラムは、それを直接操作したりはしません。提供される一連のメソッドを使います。そのようにして、従業員データは常に適切な状態に保たれます。

実行時には、データ構造体に値が入れた時点で、抽象概念としての従業員のインスタンスが作られることになります。インスタンス（通常オブジェクトと呼ばれる）は、必要な数だけ作れます。それぞれのオブジェクトには、実際の従業員の名前、番号、肩書などが割り当てられます（図 9-2 を参照）。このデータは、それに対応付けられたメソッドによってしかアクセスまたは変更できません。このように、オブジェクト型を使えば、属性と動作がうまく定義されたオブジェクトを作成できます。

図 9-2 オブジェクト型とそのオブジェクト（インスタンス）



なぜオブジェクト型を使うか

オブジェクト型により、大きなシステムが複数の論理エンティティに細分化されるため、複雑さが軽減されます。これにより、モジュール構造を持ち、維持および再利用が可能なソフトウェア・コンポーネントを作れます。さらに、別々のチームの複数のプログラマがソフトウェア・コンポーネントを並行して開発できるようになります。

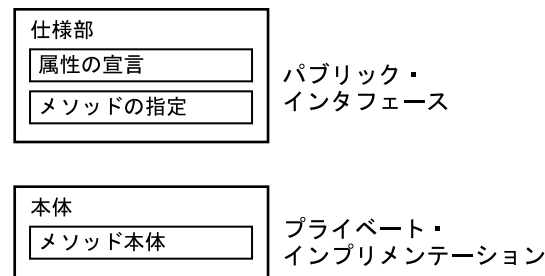
データに対する操作をカプセル化することにより、オブジェクト型を使ってデータ・メンテナンスのためのコードをSQL スクリプトや PL/SQL ブロックではなく、メソッドに入れることができます。オブジェクト型では、データへのアクセスが、そのための許可を受けた操作によってしかできないようにすることにより、副作用が最小限になります。また、オブジェクト型を使えば、インプリメンテーションの細部が隠されるため、クライアント・プログラムに影響を及ぼすことなく細部を変更できます。

オブジェクト型を使うことにより、現実のデータをモデル化できます。複雑な実世界のエンティティと関連は、オブジェクト型に直接対応付けることができます。さらにオブジェクト型は、Java および C++ などのオブジェクト指向言語で定義されたクラスに直接対応付けられます。このようにして、プログラムがシミュレートしようとしている世界をよりよく反映できるようになります。

オブジェクト型の構造

パッケージと同様に、オブジェクト型は仕様部と本体という2つの部分から構成されます（図 9-3 を参照）。仕様部はアプリケーションへのインタフェースです。ここでは、データ構造体（属性の集合）とデータ操作に必要な演算（メソッド）を宣言します。本体ではメソッドを完全に定義し、それによって仕様部をインプリメントします。

図 9-3 オブジェクト型構造



メソッドを使うためにクライアント・プログラムが必要とするすべての情報は、仕様部にあります。仕様部は操作インタフェース、そして本体はブラック・ボックスと考えてください。仕様部を変更しなくても、本体をデバッグ、拡張、または置換できます。クライアント・プログラムには影響を与えません。

オブジェクト型の仕様部では、メソッドより前にすべての属性を宣言しなければなりません。サブプログラムだけがインプリメンテーションを必要とします。そのため、オブジェクト型の仕様部に属性の宣言しかない場合、オブジェクト型の本体は不要です。本体では属性を宣言できません。オブジェクト型の仕様部のすべての宣言は、パブリックです（オブジェクト型の外側から参照できる）。

構造をさらに理解するために、下記の例をご覧ください。ここでは、複素数のオブジェクト型が定義されています。今のところ、複素数には実数部と虚数部の2つの部分があること、および複素数に対して数種類の算術演算が定義されていることを知るだけで十分です。

```
CREATE TYPE Complex AS OBJECT (  
    rpart REAL, -- attribute  
    ipart REAL,  
    MEMBER FUNCTION plus (x Complex) RETURN Complex, -- method  
    MEMBER FUNCTION less (x Complex) RETURN Complex,  
    MEMBER FUNCTION times (x Complex) RETURN Complex,  
    MEMBER FUNCTION divby (x Complex) RETURN Complex  
);  
  
CREATE TYPE BODY Complex AS  
    MEMBER FUNCTION plus (x Complex) RETURN Complex IS  
    BEGIN  
        RETURN Complex(rpart + x.rpart, ipart + x.ipart);  
    END plus;  
  
    MEMBER FUNCTION less (x Complex) RETURN Complex IS  
    BEGIN  
        RETURN Complex(rpart - x.rpart, ipart - x.ipart);  
    END less;  
  
    MEMBER FUNCTION times (x Complex) RETURN Complex IS  
    BEGIN  
        RETURN Complex(rpart * x.rpart - ipart * x.ipart,  
                        rpart * x.ipart + ipart * x.rpart);  
    END times;  
  
    MEMBER FUNCTION divby (x Complex) RETURN Complex IS  
        z REAL := x.rpart**2 + x.ipart**2;  
    BEGIN  
        RETURN Complex((rpart * x.rpart + ipart * x.ipart) / z,  
                        (ipart * x.rpart - rpart * x.ipart) / z);  
    END divby;  
END;
```

オブジェクト型を構成するさまざまな要素

オブジェクト型はデータと操作をカプセル化します。そのため、属性とメソッドはオブジェクト型仕様部で宣言できますが、定数、例外、カーソル、型は宣言できません。少なくとも 1 つの属性を宣言する必要があります（最大で 1000）。メソッドはオプションです。既存のオブジェクト型には属性を追加できません（型の拡張はサポートされていません）。

属性

変数と同じように、属性は名前とデータ型を指定して宣言します。名前はそのオブジェクト型の中で固有でなければなりません（他のオブジェクト型内では使えます）。データ型としては、任意の Oracle 型を使えますが、次のものは使えません。

- LONG、LONG RAW
- NCHAR、NCLOB、NVARCHAR2
- ROWID、UROWID
- PL/SQL 固有の型 BINARY_INTEGER（およびそのサブタイプ）、BOOLEAN、PLS_INTEGER、RECORD、REF CURSOR、%TYPE、%ROWTYPE
- PL/SQL パッケージ内で定義されている型

属性の宣言内では、代入演算子または DEFAULT 句を使っての属性の初期化はできません。また、属性に NOT NULL 制約を課することはできません。しかし、オブジェクトをデータベースの表に格納して、その表に制約を課することはできます。

属性の集合によって形成されるデータ構造体の種類は、モデル化される実社会のオブジェクトに依存します。たとえば、分子と分母からなる有理数を表すのに必要なのは、2 つの INTEGER 変数だけです。一方、大学の学生を表すには、名前、住所、電話番号、状況などを入れるための複数の VARCHAR2 変数、そしてコースおよび成績を入れるための 1 つの VARRAY 変数が必要です。

データ構造体は非常に複雑なものとなることがあります。たとえば、属性のデータ型を別のオブジェクト型とすることができます（ネストされたオブジェクト型と呼ばれます）。それにより、より単純なオブジェクト型からもっと複雑なオブジェクト型を作ることが可能になります。待ち行列、リスト、ツリーなどの一部のオブジェクト型は動的です。つまり、使われるときに拡張します。自分自身への直接または間接の参照を含む再帰的オブジェクト型は、高度に洗練されたデータ・モデルを可能にします。

メソッド

一般にメソッドとは、オブジェクト型仕様部でキーワード `MEMBER` または `STATIC` を使って宣言されるサブプログラムです。メソッドの名前には、オブジェクト型またはその属性のどれかと同じ名前は使えません。次に示すように、`MEMBER` メソッドはインスタンスで起動されます。

```
instance_expression.method()
```

しかし、次に示すように、`STATIC` メソッドはインスタンスではなくオブジェクト型で起動されます。

```
object_type_name.method()
```

パッケージ化されたサブプログラムと同様に、ほとんどのメソッドには仕様部と本体の2つの部分があります。仕様部は、メソッド名、オプションのパラメータ・リスト、そして関クションの場合は戻り型から構成されます。本体は、特定の操作を実行するためのコードです。

オブジェクト型仕様部内のそれぞれのメソッド仕様部に対応するメソッド本体が、オブジェクト型本体内に存在していなければなりません。メソッドの仕様部と本体を一致させるために、PL/SQL は、それらのヘッダーをトークンごとに比較します。このため、ヘッダーは一語一語が一致していなければなりません。

次の例が示すように、オブジェクト型の中でメソッドは、修飾子なしで属性および他のメソッドを参照できます。

```
CREATE TYPE Stack AS OBJECT (  
    top INTEGER,  
    MEMBER FUNCTION full RETURN BOOLEAN,  
    MEMBER PROCEDURE push (n IN INTEGER),  
    ...  
);  
  
CREATE TYPE BODY Stack AS  
    ...  
    MEMBER PROCEDURE push (n IN INTEGER) IS  
    BEGIN  
        IF NOT full THEN  
            top := top + 1;  
            ...  
        END push;  
    END;  
END;
```

パラメータ SELF

MEMBER メソッドは SELF という名前の組込みパラメータを受け入れます。これはオブジェクト型のインスタンスです。暗黙のうちに宣言されていても明示的に宣言されていても、SELF は常に MEMBER メソッドに渡される第 1 パラメータです。しかし、STATIC メソッドは SELF を受け入れたり、参照したりできません。

メソッドの本体では、SELF はメソッドが起動されたオブジェクトを示します。たとえば、メソッド transform は SELF を IN OUT パラメータとして宣言します。

```
CREATE TYPE Complex AS OBJECT (
    MEMBER FUNCTION transform (SELF IN OUT Complex) ...
```

SELF にそれ以外のデータ型は指定できません。MEMBER ファンクションでは、SELF が宣言されていない場合、そのパラメータ・モードはデフォルトで IN に設定されます。しかし、MEMBER プロシージャでは、SELF が宣言されていない場合、そのパラメータ・モードはデフォルトで IN OUT に設定されます。SELF には OUT パラメータを指定できません。

次の例で示されるように、メソッドでは修飾子なしで SELF の属性を参照できます。

```
CREATE FUNCTION gcd (x INTEGER, y INTEGER) RETURN INTEGER AS
-- find greatest common divisor of x and y
    ans INTEGER;
BEGIN
    IF (y <= x) AND (x MOD y = 0) THEN ans := y;
    ELSIF x < y THEN ans := gcd(y, x);
    ELSE ans := gcd(y, x MOD y);
    END IF;
    RETURN ans;
END;

CREATE TYPE Rational AS OBJECT (
    num INTEGER,
    den INTEGER,
    MEMBER PROCEDURE normalize,
    ...
);

CREATE TYPE BODY Rational AS
    MEMBER PROCEDURE normalize IS
        g INTEGER;
    BEGIN
        g := gcd(SELF.num, SELF.den);
        g := gcd(num, den); -- equivalent to previous statement
        num := num / g;
        den := den / g;
    END normalize;
    ...
END;
```

SQL 文からは、NULL インスタンス（SELF が NULL である状態）で MEMBER メソッドをコールすると、メソッドは起動されず NULL が戻されます。プロシージャ文からは、NULL インスタンスで MEMBER メソッドをコールすると、メソッドが起動される前に事前定義の例外 SELF_IS_NULL が呼び出されます。

オーバーロード

パッケージ化されたサブプログラムと同じく、同じ種類（ファンクションまたはプロシージャ）のメソッドはオーバーロードできます。つまり、仮パラメータの数、順序、またはデータ型の種類が違っていれば、同じ名前を複数の異なるメソッドで使えます。メソッドの 1 つをコールするとき、PL/SQL は実パラメータのリストと仮パラメータのリストとを比較して、メソッドを検索します。

2 つのメソッドの仮パラメータの違いがパラメータ・モードだけの場合、それらのメソッドはオーバーロードできません。また、戻り型しか違う 2 つのメンバー・ファンクションはオーバーロードできません。詳細は、7-23 ページの「[オーバーロード](#)」を参照してください。

マップ・メソッドと順序付けメソッド

CHAR または REAL などのスカラー・データ型の値には事前定義済みの順序があるので、比較できます。しかし、オブジェクト型のインスタンスには事前定義済みの順序がありません。それらを順序付けるために、PL/SQL はユーザー提供のマップ・メソッドをコールします。次の例で、キーワード MAP は、メソッド convert が有理オブジェクトを REAL 値にマップすることにより順序付けることを示します。

```
CREATE TYPE Rational AS OBJECT (  
    num INTEGER,  
    den INTEGER,  
    MAP MEMBER FUNCTION convert RETURN REAL,  
    ...  
);  
  
CREATE TYPE BODY Rational AS  
    MAP MEMBER FUNCTION convert RETURN REAL IS  
    BEGIN  
        RETURN num / den;  
    END convert;  
    ...  
END;
```

PL/SQL は順序付けを使って、 $x > y$ などのブール式を評価したり、DISTINCT、GROUP BY および ORDER BY 句によって暗黙のうちに必要となる比較を実行したりします。マップ・メソッド convert は、すべての有理オブジェクトを順序付けする際の、オブジェクトの相対的な位置を戻します。

1つのオブジェクト型のマップ・メソッドは1つだけです。このメソッドは、DATE、NUMBER、VARCHAR2、または CHARACTER や REAL などの ANSI SQL 型のいずれかをスカラーの戻り型として持つ、パラメータのないファンクションでなければなりません。

あるいは、順序付けメソッドを用意することもできます。1つのオブジェクト型の順序付けメソッドは1つだけです。このメソッドは、戻り型が数値のファンクションでなければなりません。次に示す例で、キーワード ORDER はメソッド match が2つのオブジェクトを比較するものであることを示しています。

```
CREATE TYPE Customer AS OBJECT (  
    id    NUMBER,  
    name  VARCHAR2(20),  
    addr  VARCHAR2(30),  
    ORDER MEMBER FUNCTION match (c Customer) RETURN INTEGER  
);  
  
CREATE TYPE BODY Customer AS  
    ORDER MEMBER FUNCTION match (c Customer) RETURN INTEGER IS  
    BEGIN  
        IF id < c.id THEN  
            RETURN -1; -- any negative number will do  
        ELSIF id > c.id THEN  
            RETURN 1;  -- any positive number will do  
        ELSE  
            RETURN 0;  
        END IF;  
    END;  
END;
```

順序付けメソッドは、いずれもパラメータを2つだけとります。組込みパラメータ SELF および同じ型の別のオブジェクトの2つです。c1 および c2 が Customer オブジェクトの場合、c1 > c2 などの比較操作を実行すると、自動的にメソッド match がコールされます。このメソッドの戻り値は、負数、0（ゼロ）、または正数であり、それぞれ SELF が他方のパラメータより小さい、等しい、大きいことを示しています。順序付けメソッドに渡されるパラメータのいずれかが NULL の場合、メソッドは NULL を戻します。

指針 map メソッドはハッシュ関数のように働いて、オブジェクト値をスカラー値にマップします（スカラー値の方が比較が容易です）。次に、このスカラー値どうしを比較します。order メソッドは、1つのオブジェクト値をもう1つのオブジェクト値と比較するだけです。

マップ・メソッドか順序付けメソッドを宣言できますが、その両方は宣言できません。どちらかのメソッドを宣言すれば、オブジェクトを SQL 文およびプロシージャ文によって比較できます。しかしどちらのメソッドも宣言しない場合、オブジェクトは SQL 文でしか比較できず、しかも等しいか等しくないかの比較しかできません。（同じ型の2つのオブジェクトが等しいとされるのは、それらの対応する属性の値が等しい場合だけです。）

大量のオブジェクトをソートまたはマージするときは、map メソッドを使います。1 回のコールで、すべてのオブジェクトをスカラーにマップして、そのスカラーをソートできます。order メソッドは、何回もコールしなければならないので効率が悪くなります（1 回に 2 つのオブジェクトどうしを比較することしかできません）。PL/SQL ではオブジェクト値に対してハッシュするため、ハッシュ結合のためには、map メソッドを使う必要があります。

コンストラクタ・メソッド

どのオブジェクト型にもコンストラクタ・メソッド（略してコンストラクタ）があります。それは、そのオブジェクト型と同じ名前のシステム定義のファンクションです。コンストラクタは、そのオブジェクト型のインスタンスを初期化したり、そのインスタンスを戻すために使います。

Oracle は、どのオブジェクト型についてもデフォルトのコンストラクタを生成します。コンストラクタの仮パラメータは、オブジェクト型の属性と一致します。つまり、パラメータと属性は同じ順序で宣言され、その名前とデータ型は同じです。PL/SQL は、暗黙のうちにコンストラクタをコールすることは決していないため、明示的にコールすることが必要です。詳細は、9-24 ページの「[コンストラクタのコール](#)」を参照してください。

オブジェクト型の定義

オブジェクト型は、任意の実社会のエンティティを表すことができます。たとえば、オブジェクト型により学生、銀行口座、コンピュータ・スクリーン、有理数、あるいは待ち行列またはスタック、リストなどのデータ構造体を表せます。ここでは、いくつかの完結した例を示して、オブジェクト型の設計の多くの面を示し、独自のオブジェクト型を作る準備をします。

現在のところ、PL/SQL のブロック、サブプログラム、またはパッケージ内ではオブジェクト型を定義できません。しかし、SQL*Plus では、次の構文を使って対話的に定義できます。

```
CREATE [OR REPLACE] TYPE type_name
  [AUTHID {CURRENT_USER | DEFINER}] {IS | AS} OBJECT (
    attribute_name datatype[, attribute_name datatype]...
    [{MAP | ORDER} MEMBER function_spec,]
    [{MEMBER | STATIC} {subprogram_spec | call_spec}
    [, {MEMBER | STATIC} {subprogram_spec | call_spec}]...]
  );

[CREATE [OR REPLACE] TYPE BODY type_name {IS | AS}
  { {MAP | ORDER} MEMBER function_body;
    | {MEMBER | STATIC} {subprogram_body | call_spec};}
  [{MEMBER | STATIC} {subprogram_body | call_spec};]...
END;]
```

AUTHID 句は、すべてのメンバー・メソッドがその定義者（デフォルト）と実行者のどちらの権限で実行するか、およびスキーマ・オブジェクトへの未修飾の参照が定義者と実行者のどちらのスキーマで解決されるかを決定します。詳細は、7-29 ページの「[実行者権限と定義者権限](#)」を参照してください。

コール仕様部は、Oracle データ・ディクショナリ内の外部 C ファンクションまたは Java メソッドを発行します。これは、対応する SQL 版に名前、パラメータ型および戻り型をマップすることによって、ルーチンを発行します。Java コール仕様部を作成する方法は、『Oracle8i Java ストアド・プロシージャ開発者ガイド』を参照してください。C コール仕様部を作成する方法は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

オブジェクト型 *Stack*

スタックは、データ項目の順序付き集合を入れるものです。その名前に示されているように、スタックには先頭と末尾があります。しかし、項目を追加または削除できるのは、先頭からだけです。そのため、スタックに最後に追加された項目が、最初に削除される項目となります。（カフェテリアで取り皿を積み重ねたところ（スタック）を想像してください。）スタックは、操作 *push* および *pop* により、後入れ先出し（LIFO）方式で更新されます。

スタックは、幅広く適用できます。たとえば、システム・プログラミングでは、割込みの優先順位を決定したり、再帰を管理するために使われます。スタックの最も簡単なインプリメンテーションは、整数の配列を使うものです。整数は配列要素として格納され、配列の片方の端がスタックの先頭を表します。

PL/SQL には VARRAY というデータ型があります。これを使うと、サイズが可変の配列（varray）を宣言できます。varray 属性を宣言するには、まずその型を定義する必要があります。しかし、オブジェクト型仕様部の中で型の宣言はできません。そこで、次のようにして最大サイズを指定し、スタンドアロンの varray 型を定義します。

```
CREATE TYPE IntArray AS VARRAY(25) OF INTEGER;
```

これで、オブジェクト型仕様部を記述できます。

```
CREATE TYPE Stack AS OBJECT (
    max_size INTEGER,
    top      INTEGER,
    position IntArray,
    MEMBER PROCEDURE initialize,
    MEMBER FUNCTION full RETURN BOOLEAN,
    MEMBER FUNCTION empty RETURN BOOLEAN,
    MEMBER PROCEDURE push (n IN INTEGER),
    MEMBER PROCEDURE pop (n OUT INTEGER)
);
```

最後に、オブジェクト型本体を記述できます。

```
CREATE TYPE BODY Stack AS
  MEMBER PROCEDURE initialize IS
  BEGIN
    top := 0;
    /* Call constructor for varray and set element 1 to NULL. */
    position := IntArray(NULL);
    max_size := position.LIMIT; -- get varray size constraint
    position.EXTEND(max_size - 1, 1); -- copy element 1 into 2..25
  END initialize;

  MEMBER FUNCTION full RETURN BOOLEAN IS
  BEGIN
    RETURN (top = max_size); -- return TRUE if stack is full
  END full;

  MEMBER FUNCTION empty RETURN BOOLEAN IS
  BEGIN
    RETURN (top = 0); -- return TRUE if stack is empty
  END empty;

  MEMBER PROCEDURE push (n IN INTEGER) IS
  BEGIN
    IF NOT full THEN
      top := top + 1; -- push integer onto stack
      position(top) := n;
    ELSE -- stack is full
      RAISE_APPLICATION_ERROR(-20101, 'stack overflow');
    END IF;
  END push;

  MEMBER PROCEDURE pop (n OUT INTEGER) IS
  BEGIN
    IF NOT empty THEN
      n := position(top);
      top := top - 1; -- pop integer off stack
    ELSE -- stack is empty
      RAISE_APPLICATION_ERROR(-20102, 'stack underflow');
    END IF;
  END pop;
END;
```

メンバー・プロシージャ push および pop では、組み込みプロシージャ raise_application_error を使ってユーザー定義のエラー・メッセージを発行します。このようにして、エラーをクライアント・プログラムに報告し、未処理の例外をホスト環境に戻さないようにしています。クライアント・プログラムは、PL/SQL 例外を受け取り、エラー報

告ファンクション `SQLCODE` および `SQLERRM` を使って `OTHERS` 例外ハンドラで処理できます。次に例を示します。

```
DECLARE
    err_num NUMBER;
    err_msg VARCHAR2(100);
BEGIN
    ...
EXCEPTION
    WHEN OTHERS THEN
        err_num := SQLCODE;
        err_msg := SUBSTR(SQLERRM, 1, 100);
        DBMS_OUTPUT.PUT_LINE(TO_CHAR(err_num) || ': ' || err_msg);
```

文字列ファンクション `SUBSTR` を使っているので、`SQLERRM` の値を `err_msg` に代入しても、(切捨ての結果として起こる) `VALUE_ERROR` 例外は呼び出されません。

または次の例に示すように、プログラムでプラグマ `EXCEPTION_INIT` を使用して、`raise_application_error` から戻されたエラー番号を名前付き例外にマップすることもできます。

```
DECLARE
    stack_overflow EXCEPTION;
    stack_underflow EXCEPTION;
    PRAGMA EXCEPTION_INIT(stack_overflow, -20101);
    PRAGMA EXCEPTION_INIT(stack_underflow, -20102);
BEGIN
    ...
EXCEPTION
    WHEN stack_overflow THEN ...
```

オブジェクト型 *Ticket_Booth*

3 つの部分からなる低価格の映画館を想像してください。各映画館にはチケット・ブースがあって、3 つの異なる映画のチケットが販売されています。すべてのチケットの価格は \$3.00 です。定期的に、チケット代金が収集されて、チケットの在庫が補充されます。

チケット・ブースを表すオブジェクト型を定義する前に、必要なデータおよび操作を考えなければなりません。単純なチケット・ブースの場合、オブジェクト型にはチケット価格、手元にあるチケットの数量、および受領額の属性が必要です。また、チケットの購入、在庫調べ、在庫の補充、および領収書の収集といった操作のためのメソッドも必要です。

受領額のために、要素 3 個の `varray` を使います。要素 1 および 2、3 には、それぞれ映画 1 および 2、3 のチケットの受領額を記録します。`varray` 属性を宣言するには、まずその型を定義する必要があります。

```
CREATE TYPE RealArray AS VARRAY(3) OF REAL;
```

これで、オブジェクト型仕様部を記述できます。

```
CREATE TYPE Ticket_Booth AS OBJECT (  
    price      REAL,  
    qty_on_hand INTEGER,  
    receipts   RealArray,  
    MEMBER PROCEDURE initialize,  
    MEMBER PROCEDURE purchase (  
        movie  INTEGER,  
        amount REAL,  
        change OUT REAL),  
    MEMBER FUNCTION inventory RETURN INTEGER,  
    MEMBER PROCEDURE replenish (quantity INTEGER),  
    MEMBER PROCEDURE collect (movie INTEGER, amount OUT REAL)  
);
```

最後に、オブジェクト型本体を記述できます。

```
CREATE TYPE BODY Ticket_Booth AS  
    MEMBER PROCEDURE initialize IS  
    BEGIN  
        price := 3.00;  
        qty_on_hand := 5000; -- provide initial stock of tickets  
        -- call constructor for varray and set elements 1..3 to zero  
        receipts := RealArray(0,0,0);  
    END initialize;  
  
    MEMBER PROCEDURE purchase (  
        movie  INTEGER,  
        amount REAL,  
        change OUT REAL) IS  
    BEGIN  
        IF qty_on_hand = 0 THEN  
            RAISE_APPLICATION_ERROR(-20103, 'out of stock');  
        END IF;  
        IF amount >= price THEN  
            qty_on_hand := qty_on_hand - 1;  
            receipts(movie) := receipts(movie) + price;  
            change := amount - price;  
        ELSE -- amount is not enough  
            change := amount; -- so return full amount  
        END IF;  
    END purchase;  
  
    MEMBER FUNCTION inventory RETURN INTEGER IS  
    BEGIN  
        RETURN qty_on_hand;  
    END inventory;
```

```

MEMBER PROCEDURE replenish (quantity INTEGER) IS
BEGIN
    qty_on_hand := qty_on_hand + quantity;
END replenish;

MEMBER PROCEDURE collect (movie INTEGER, amount OUT REAL) IS
BEGIN
    amount := receipts(movie); -- get receipts for a given movie
    receipts(movie) := 0; -- reset receipts to zero
END collect;
END;

```

オブジェクト型 *Bank_Account*

銀行口座を表すオブジェクト型を定義する前に、必要なデータおよび操作を考えなければなりません。簡単な銀行口座では、オブジェクト型に口座番号、残高、状態のための属性が必要です。また、口座の開設、口座番号の確認、口座の閉鎖、現金の預入れ、現金の引出し、および残高照会といった操作のためのメソッドが必要です。

まず、次のようにオブジェクト型仕様部を記述します。

```

CREATE TYPE Bank_Account AS OBJECT (
    acct_number INTEGER(5),
    balance      REAL,
    status       VARCHAR2(10),
    MEMBER PROCEDURE open (amount IN REAL),
    MEMBER PROCEDURE verify_acct (num IN INTEGER),
    MEMBER PROCEDURE close (num IN INTEGER, amount OUT REAL),
    MEMBER PROCEDURE deposit (num IN INTEGER, amount IN REAL),
    MEMBER PROCEDURE withdraw (num IN INTEGER, amount IN REAL),
    MEMBER FUNCTION curr_bal (SELF IN OUT Bank_Account,
        num IN INTEGER) RETURN REAL
);

```

次に、オブジェクト型本体を記述します。

```

CREATE TYPE BODY Bank_Account AS
MEMBER PROCEDURE open (amount IN REAL) IS
-- open account with initial deposit
BEGIN
    IF NOT amount > 0 THEN
        RAISE_APPLICATION_ERROR(-20104, 'bad amount');
    END IF;
    SELECT acct_sequence.NEXTVAL INTO acct_number FROM dual;
    status := 'open';
    balance := amount;
END open;

```

```
MEMBER PROCEDURE verify_acct (num IN INTEGER) IS
-- check for wrong account number or closed account
BEGIN
    IF (num <> acct_number) THEN
        RAISE_APPLICATION_ERROR(-20105, 'wrong number');
    ELSIF (status = 'closed') THEN
        RAISE_APPLICATION_ERROR(-20106, 'account closed');
    END IF;
END verify_acct;

MEMBER PROCEDURE close (num IN INTEGER, amount OUT REAL) IS
-- close account and return balance
BEGIN
    verify_acct(num);
    status := 'closed';
    amount := balance;
END close;

MEMBER PROCEDURE deposit (num IN INTEGER, amount IN REAL) IS
BEGIN
    verify_acct(num);
    IF NOT amount > 0 THEN
        RAISE_APPLICATION_ERROR(-20104, 'bad amount');
    END IF;
    balance := balance + amount;
END deposit;

MEMBER PROCEDURE withdraw (num IN INTEGER, amount IN REAL) IS
-- if account has enough funds, withdraw
-- given amount; else, raise an exception
BEGIN
    verify_acct(num);
    IF amount <= balance THEN
        balance := balance - amount;
    ELSE
        RAISE_APPLICATION_ERROR(-20107, 'insufficient funds');
    END IF;
END withdraw;

MEMBER FUNCTION curr_bal (SELF IN OUT Bank_Account, num IN INTEGER)
RETURN REAL IS
BEGIN
    verify_acct(num);
    RETURN balance;
END curr_bal;
END;
```


オブジェクト型 *Rational*

有理数 (rational number) は、分子と分母の 2 つの整数の商で表される数値です。ほとんどの言語と同じく、PL/SQL には有理数型がなく、有理数のための事前定義済みの操作もありません。オブジェクト型 *Rational* を定義することにより、それを補うことにします。まず、次のようにオブジェクト型仕様部を記述します。

```
CREATE TYPE Rational AS OBJECT (
    num INTEGER,
    den INTEGER,
    MAP MEMBER FUNCTION convert RETURN REAL,
    MEMBER PROCEDURE normalize,
    MEMBER FUNCTION reciprocal RETURN Rational,
    MEMBER FUNCTION plus (x Rational) RETURN Rational,
    MEMBER FUNCTION less (x Rational) RETURN Rational,
    MEMBER FUNCTION times (x Rational) RETURN Rational,
    MEMBER FUNCTION divby (x Rational) RETURN Rational,
    PRAGMA RESTRICT_REFERENCES (DEFAULT, RNDS, WNDS, RNPS, WNPS)
);
```

PL/SQL では、演算子のオーバーロードはできません。そのため、挿入演算子 +、-、*、および / をオーバーロードするかわりに、plus、less (minus という語は予約語です)、times、および divby という名前のメソッドを定義する必要があります。

次に、以下のスタンドアロン・ストアッド・ファンクションを作成します。このファンクションは、normalize メソッドからコールされます。

```
CREATE FUNCTION gcd (x INTEGER, y INTEGER) RETURN INTEGER AS
-- find greatest common divisor of x and y
    ans INTEGER;
BEGIN
    IF (y <= x) AND (x MOD y = 0) THEN
        ans := y;
    ELSIF x < y THEN
        ans := gcd(y, x); -- recursive call
    ELSE
        ans := gcd(y, x MOD y); -- recursive call
    END IF;
    RETURN ans;
END;
```

オブジェクト型本体は次のように記述します。

```
CREATE TYPE BODY Rational AS
    MAP MEMBER FUNCTION convert RETURN REAL IS
-- convert rational number to real number
    BEGIN
        RETURN num / den;
```

```
END convert;

MEMBER PROCEDURE normalize IS
-- reduce fraction num / den to lowest terms
  g INTEGER;
BEGIN
  g := gcd(num, den);
  num := num / g;
  den := den / g;
END normalize;

MEMBER FUNCTION reciprocal RETURN Rational IS
-- return reciprocal of num / den
BEGIN
  RETURN Rational(den, num); -- call constructor
END reciprocal;

MEMBER FUNCTION plus (x Rational) RETURN Rational IS
-- return sum of SELF + x
  r Rational;
BEGIN
  r := Rational(num * x.den + x.num * den, den * x.den);
  r.normalize;
  RETURN r;
END plus;

MEMBER FUNCTION less (x Rational) RETURN Rational IS
-- return difference of SELF - x
  r Rational;
BEGIN
  r := Rational(num * x.den - x.num * den, den * x.den);
  r.normalize;
  RETURN r;
END less;

MEMBER FUNCTION times (x Rational) RETURN Rational IS
-- return product of SELF * x
  r Rational;
BEGIN
  r := Rational(num * x.num, den * x.den);
  r.normalize;
  RETURN r;
END times;

MEMBER FUNCTION divby (x Rational) RETURN Rational IS
-- return quotient of SELF / x
  r Rational;
```

```

BEGIN
    r := Rational(num * x.den, den * x.num);
    r.normalize;
    RETURN r;
END divby;
END;

```

オブジェクトの宣言と初期化

オブジェクト型を宣言してスキーマにインストールしたなら、任意の PL/SQL ブロック、サブプログラム、またはパッケージの中で、それを使ってオブジェクトを宣言できます。たとえば、そのオブジェクト型を使って属性、列、変数、バインド変数、レコード・フィールド、表要素、仮パラメータ、またはファンクション結果のデータ型を指定できます。実行時には、そのオブジェクト型のインスタンスが作られます。つまり、その型のオブジェクトがインスタンス生成されます。オブジェクトごとに異なる値を保持できます。

そのようなオブジェクトは、通常の有効範囲およびインスタンス生成のルールに従います。ブロックまたはサブプログラムでは、ローカル・オブジェクトは、ブロックまたはサブプログラムに入ったときにインスタンス生成され、ブロックまたはサブプログラムが終了した時点で消滅します。パッケージでは、そのパッケージが初めて参照された時点でオブジェクトのインスタンスが生成され、データベース・セッションが終わった時点で消滅します。

オブジェクトの宣言

CHAR や NUMBER などの組み込み型を使用できるのであれば、どこでもオブジェクト型を使用できます。次のブロックでは、Rational 型のオブジェクト `r` を宣言しています。その後、オブジェクト型 Rational のコンストラクタをコールして、そのオブジェクトを初期化します。このコールでは、属性 `num` および `den` にそれぞれ値 6 および 8 を代入しています。

```

DECLARE
    r Rational;
BEGIN
    r := Rational(6, 8);
    DBMS_OUTPUT.PUT_LINE(r.num); -- prints 6

```

オブジェクトは、ファンクションおよびプロシージャの仮パラメータとして宣言できます。そのようにすると、オブジェクトをストアド・サブプログラムに渡したり、あるサブプログラムから別のサブプログラムに渡したりできます。次の例では、オブジェクト型 Account を使って仮パラメータのデータ型を指定しています。

```

DECLARE
    ...
    PROCEDURE open_acct (new_acct IN OUT Account) IS ...

```

次の例では、オブジェクト型 `Account` を使ってファンクションの戻り型を指定しています。

```
DECLARE
...
FUNCTION get_acct (acct_id IN INTEGER) RETURN Account IS ...
```

オブジェクトの初期化

オブジェクト型のためのコンストラクタをコールしてそのオブジェクトを初期化するまで、そのオブジェクトはアトミック `NULL` になっています。つまり、その属性だけではなくオブジェクトそのものが `NULL` になっています。次の例を考えてみます。

```
DECLARE
    r Rational; -- r becomes atomically null
BEGIN
    r := Rational(2,3); -- r becomes 2/3
```

`NULL` のオブジェクトが別のオブジェクトと等しくなることは決してありません。実際のところ、`NULL` のオブジェクトを別のオブジェクトと比較すると、常に `NULL` になります。また、アトミック `NULL` になっているオブジェクトを他のオブジェクトに代入すると、そのオブジェクトもアトミック `NULL` になります（したがって再度初期化しなければなりません）。同じように、次の例に示すように値のない `NULL` をオブジェクトに代入すると、そのオブジェクトはアトミック `NULL` になります。

```
DECLARE
    r Rational;
BEGIN
    r Rational := Rational(1,2); -- r becomes 1/2
    r := NULL; -- r becomes atomically null
    IF r IS NULL THEN ... -- condition yields TRUE
```

プログラミング上の習慣として、次の例に示すように、オブジェクトを宣言するときには初期化するようにしてください。

```
DECLARE
    r Rational := Rational(2,3); -- r becomes 2/3
```

PL/SQL による未初期化オブジェクトの処理

式の中で、未初期化オブジェクトの属性は `NULL` として評価されます。未初期化オブジェクトの属性に値を代入しようとすると、事前定義済みの例外 `ACCESS_INTO_NULL` が呼び出されます。`IS NULL` 比較演算子を未初期化オブジェクトまたはその属性に適用すると、結果は `TRUE` になります。

次の例は、NULL のオブジェクトと、属性が NULL であるオブジェクトの違いを示しています。

```
DECLARE
    r Rational; -- r is atomically null
BEGIN
    IF r IS NULL THEN ...      -- yields TRUE
    IF r.num IS NULL THEN ...  -- yields TRUE
    r := Rational(NULL, NULL); -- initializes r
    r.num := 4; -- succeeds because r is no longer atomically null
                    -- even though all its attributes are null
    r := NULL; -- r becomes atomically null again
    r.num := 4; -- raises ACCESS_INTO_NULL
EXCEPTION
    WHEN ACCESS_INTO_NULL THEN
        ...
END;
```

未初期化オブジェクトのメソッドをコールできます。その場合、SELF が NULL にバインドされます。未初期化オブジェクトの属性を IN パラメータへの引数として渡すと、それは NULL と評価されます。OUT または IN OUT パラメータへの引数として渡されると、書き込もうとしたときに例外が呼び出されます。

属性へのアクセス

属性は、オブジェクト型内の位置によってではなく、名前によってのみ参照できます。属性にアクセスしたり、その値を変更したりするには、ドット表記法を使います。次の例では、属性 den の値を変数 denominator に割り当てています。次に、変数 numerator に格納された値を属性 num に代入します。

```
DECLARE
    r Rational := Rational(NULL, NULL);
    numerator   INTEGER;
    denominator INTEGER;
BEGIN
    ...
    denominator := r.den;
    r.num := numerator;
END;
```

属性名を連鎖させて、ネストされたオブジェクト型の属性にアクセスできます。たとえば、オブジェクト型の Address および Student を次のように定義するとします。

```
CREATE TYPE Address AS OBJECT (
    street  VARCHAR2(30),
    city    VARCHAR2(20),
    state   CHAR(2),
```

コンストラクタのコール

```
        zip_code VARCHAR2(5)
    );

CREATE TYPE Student AS OBJECT (
    name          VARCHAR2(20),
    home_address  Address,
    phone_number  VARCHAR2(10),
    status        VARCAHR2(10),
    advisor_name  VARCHAR2(20),
    ...
);'
```

オブジェクト型 `Address` の 1 つの属性が `zip_code` であり、またオブジェクト型 `Student` の中の属性 `home_address` のデータ型は `Address` になっています。`s` が `Student` オブジェクトである場合、その `zip_code` 属性には次のようにしてアクセスします。

```
s.home_address.zip_code
```

コンストラクタのコール

コンストラクタは、ファンクション・コールが許可されているところでコールできます。次の例が示すように、すべてのファンクションと同じく、コンストラクタは式の一部としてコールされます。

```
DECLARE
    r1 Rational := Rational(2, 3);
    FUNCTION average (x Rational, y Rational) RETURN Rational IS
    BEGIN
        ...
    END;
BEGIN
    r1 := average(Rational(3, 4), Rational(7, 11));
    IF (Rational(5, 8) > r1) THEN
        ...
    END IF;
END;
```

パラメータをコンストラクタに渡してコンストラクタをコールすると、インスタンスを生成するオブジェクトの属性に初期値が代入されます。定数や変数とは違って、属性にはデフォルト句を指定できないので、すべての属性のためにパラメータを指定しなければなりません。次の例で示すように、 n 番目のパラメータは n 番目の属性に値を代入します。

```
DECLARE
    r Rational;
BEGIN
    r := Rational(5, 6); -- assign 5 to num, 6 to den
```

```
-- now r is 5/6
```

次の例で示すように、位置表記法ではなく名前表記法を使ってコンストラクタをコールすることもできます。

```
BEGIN
  r := Rational(den => 6, num => 5); -- assign 5 to num, 6 to den
```

メソッドのコール

パッケージされたサブプログラムと同じく、メソッドはドット表記法を使ってコールされます。次の例では、属性 `num` および `den` をそれらの最大公約数で割るメソッド `normalize` をコールします。

```
DECLARE
  r Rational;
BEGIN
  r := Rational(6, 8);
  r.normalize;
  DBMS_OUTPUT.PUT_LINE(r.num); -- prints 3
```

次の例でわかるように、メソッドのコールは連鎖することができます。実行は左から右へと進んでいきます。最初にメンバー・ファンクション `reciprocal` がコールされ、次にメンバー・プロシージャ `normalize` がコールされます。

```
DECLARE
  r Rational := Rational(6, 8);
BEGIN
  r.reciprocal().normalize;
  DBMS_OUTPUT.PUT_LINE(r.num); -- prints 4
```

SQL 文からパラメータのないメソッドをコールするには、空のパラメータ・リストが必要です。プロシージャ文では、コールを連鎖しないかぎり空のパラメータ・リストはなくてもかまいません。連鎖する場合は、最後のコール以外のすべてで空のパラメータ・リストが必要です。

プロシージャは式の一部としてではなく文としてコールされるので、追加のメソッド・コールをプロシージャの右側に連鎖させることはできません。たとえば、次の文は誤りです。

```
r.normalize().reciprocal; -- illegal
```

さらに、2 つのファンクション・コールの連鎖では、1 番目のファンクションは 2 番目のファンクションに渡せるオブジェクトを戻さなければなりません。

オブジェクトの共有

実社会にあるほとんどのオブジェクトは、型 `Rational` よりもずっと大きくて複雑です。次のオブジェクト型について考えてください。

```
CREATE TYPE Address AS OBJECT (  
    street_address VARCHAR2(35),  
    city            VARCHAR2(15),  
    state           CHAR(2),  
    zip_code        INTEGER  
);  
  
CREATE TYPE Person AS OBJECT (  
    first_name  VARCHAR2(15),  
    last_name   VARCHAR2(15),  
    birthday    DATE,  
    home_address Address, -- nested object type  
    phone_number VARCHAR2(15),  
    ss_number   INTEGER,  
    ...  
);
```

`Address` オブジェクトには `Rational` オブジェクトの2倍の数の属性があり、`Person` オブジェクトには型 `Address` も含めてさらに多くの属性があります。オブジェクトが大きい場合、それらのコピーをサブプログラムからサブプログラムに渡すことは非効率的です。それらを共有する方が便利です。これは、オブジェクトにオブジェクト識別子があれば可能です。オブジェクトを共有するには、参照 (`ref`) を使います。`ref` は、オブジェクトへのポインタです。

共有には、2つの重要な利点があります。まず、データが不必要にレプリケートされません。第2に、共有されているオブジェクトが更新される時、変更は1箇所だけでなされ、すべての `ref` では更新された値をすぐに取り出せます。

次の例では、オブジェクト型 `Home` を定義し、そのオブジェクト型のインスタンスを格納する表を作ることにより、共有の利点を活用しています。

```
CREATE TYPE Home AS OBJECT (  
    address  VARCHAR2(35),  
    owner    VARCHAR2(25),  
    age      INTEGER,  
    style    VARCHAR(15),  
    floor plan BLOB,  
    price    REAL(9,2),  
    ...  
);  
...  
CREATE TABLE homes OF Home;
```


ref の使用方法

オブジェクト型 `Person` を訂正することにより、複数の人が同じ家を共有する社会をモデル化できます。オブジェクトへのポインタを含む `ref` を宣言するために、型修飾子 `REF` を使います。

```
CREATE TYPE Person AS OBJECT (
    first_name  VARCHAR2(10),
    last_name   VARCHAR2(15),
    birthday    DATE,
    home_address REF Home, -- can be shared by family
    phone_number VARCHAR2(15),
    ss_number   INTEGER,
    mother      REF Person, -- family members refer to each other
    father      REF Person,
    ...
);
```

人から家への参照および人と人との間の参照によって、実社会に存在する関係をモデル化しているところに注目してください。

`ref` は、変数、パラメータ、フィールド、または属性として宣言できます。また、`ref` は SQL のデータ操作文の中で入力変数または出力変数として使えます。しかし、`ref` を通してはナビゲートできません。つまり、`x.attribute` などの式に対して (`x` は `ref`) 次の例が示すように、PL/SQL は参照されるオブジェクトが格納されている表にナビゲートできません。たとえば、次の割当ては誤りです。

```
DECLARE
    p_ref    REF Person;
    phone_no VARCHAR2(15);
BEGIN
    phone_no := p_ref.phone_number; -- illegal
```

そのかわりに、オブジェクトにアクセスするには演算子 `DEREF` を使う必要があります。例は、9-31 ページの「[演算子 DEREF を使う](#)」を参照してください。

前方型定義

参照できるスキーマ・オブジェクトは、すでに存在するものだけです。次の例の最初の `CREATE TYPE` 文は、まだ存在しないオブジェクト型 `Department` を参照するため誤りです。

```
CREATE TYPE Employee AS OBJECT (
    name VARCHAR2(20),
    dept REF Department, -- illegal
    ...
);
```

```
CREATE TYPE Department AS OBJECT (  
    number INTEGER,  
    manager Employee,  
    ...  
);
```

CREATE TYPE 文の順番を入れ替えても意味がありません。これは、オブジェクト型が相互に依存しているためです。つまり、あるオブジェクト型が ref を通じて他のオブジェクト型に依存しています。この問題を解決するには、互いに依存するオブジェクト型の定義が可能な、前方型定義と呼ばれる特別な CREATE TYPE 文を使います。

上の例をデバッグするには、その前に次の文を追加してください。

```
CREATE TYPE Department; -- forward type definition  
-- at this point, Department is an incomplete object type
```

前方型定義によって作られたオブジェクト型は、(完全に定義されるまで) 属性またはメソッドがないため、不完全なオブジェクト型と呼ばれます。

純粋ではない不完全なオブジェクト型は、属性は持っていますがコンパイルするとエラーが発生します。これは、そのオブジェクト型が未定義の型を参照しているためです。たとえば、次の CREATE TYPE 文は、オブジェクト型 Address が未定義のためにエラーを引き起こします。

```
CREATE TYPE Customer AS OBJECT (  
    id NUMBER,  
    name VARCHAR2(20),  
    addr Address, -- not yet defined  
    phone VARCHAR2(15)  
);
```

前方型定義により、オブジェクト型 Address の定義を遅らせることができます。さらに、不完全な型 Customer を他のアプリケーション開発者は ref で使えます。

オブジェクトの操作

列のデータ型を指定するには、CREATE TABLE 文の中でオブジェクト型を使います。表が作られたなら、SQL 文を使ってオブジェクトの挿入、属性の選択、メソッドのコール、および状態の更新ができます。

注意: リモート・オブジェクトまたは分散オブジェクトへはアクセスできません。

次の SQL*Plus スクリプトで、INSERT 文はオブジェクト型 Rational のコンストラクタをコールし、結果として生成されたオブジェクトを挿入 (INSERT) します。SELECT 文は属性 num の値を取り出します。UPDATE 文はメンバー・メソッド reciprocal をコールします。これは、属性 num と den を交換した後、Rational の値を戻します。属性やメソッドを参照するには、表の別名が必要であることに注意してください。(詳細は、[付録 D](#) を参照してください。)

```

CREATE TABLE numbers (rn Rational, ...)
/
INSERT INTO numbers (rn) VALUES (Rational(3, 62)) -- inserts 3/62
/
SELECT n.rn.num INTO my_num FROM numbers n ... -- returns 3
/
UPDATE numbers n SET n.rn = n.rn.reciprocal ... -- yields 62/3
/

```

この方法でオブジェクトをインスタンス生成するとき、それはデータベースの表の外部では認識されません。しかし、そのオブジェクト型はどの表からも独立して存在していて、他の方法でオブジェクトを作るために使えます。

次の例では、Rational 型のオブジェクトを行に格納する表を作ります。オブジェクトの行を含むそのような表は、オブジェクト表と呼ばれます。行の各列は、そのオブジェクト型の属性に対応します。行ごとに違う列の値を持つことが可能です。

```
CREATE TABLE rational_nums OF Rational;
```

オブジェクト表の各行ごとにオブジェクト識別子があり、それはその行に格納されているオブジェクトを固有に識別して、そのオブジェクトへの参照として機能します。

オブジェクトの選択

オブジェクト型 Person とオブジェクト表 persons を作る次の SQL*Plus スクリプトを実行して、その表にデータを入れたとします。

```

CREATE TYPE Person AS OBJECT (
    first_name  VARCHAR2(15),
    last_name   VARCHAR2(15),
    birthday    DATE,
    home_address Address,
    phone_number VARCHAR2(15))
/
CREATE TABLE persons OF Person
/

```

次の副問合せは、Person オブジェクトの属性しか含まない行からなる結果セットを生成します。

```

BEGIN
    INSERT INTO employees -- another object table of type Person
        SELECT * FROM persons p WHERE p.last_name LIKE '%Smith';

```

オブジェクトの結果セットを戻すには、次の項で説明する演算子 VALUE を使わなければなりません。

演算子 VALUE を使う

その名前のとおり、演算子 VALUE はオブジェクトの値を戻します。VALUE は相関変数を引数とします。(この文脈で相関変数とは、オブジェクト表の行に関連付けられている行変数または表別名のことです。)たとえば、Person オブジェクトから結果セットを戻すには、次のように VALUE を使います。

```
BEGIN
  INSERT INTO employees
    SELECT VALUE(p) FROM persons p
    WHERE p.last_name LIKE '%Smith';
```

次の例では、VALUE を使って特定の Person オブジェクトを戻しています。

```
DECLARE
  p1 Person;
  p2 Person;
  ...
BEGIN
  SELECT VALUE(p) INTO p1 FROM persons p
    WHERE p.last_name = 'Kroll';
  p2 := p1;
  ...
END;
```

この時点で、p1 は姓が 'Kroll' の格納オブジェクトのコピーであるローカル Person オブジェクト、p2 は p1 のコピーである別のローカル Person オブジェクトです。次の例に示すように、これらの変数を使ってそれらに含まれるオブジェクトにアクセスしたりそれを更新したりできます。

```
BEGIN
  p1.last_name := p1.last_name || 'Jr';
```

この時点で、ローカル Person オブジェクトである p1 は、姓が 'Kroll Jr' になっています。

演算子 REF を使う

演算子 REF を使うと、ref を取り出せます。この演算子は、VALUE と同様に相関変数を引数とします。次の例では、Person オブジェクトへの 1 つまたは複数の ref を取り出し、その ref を表 person_refs に挿入します。

```
BEGIN
  INSERT INTO person_refs
    SELECT REF(p) FROM persons p
    WHERE p.last_name LIKE '%Smith';
```

次の例では、ref と属性を同時に取り出します。

```
DECLARE
  p_ref      REF Person;
  taxpayer_id VARCHAR2(9);
BEGIN
  SELECT REF(p), p.ss_number INTO p_ref, taxpayer_id
    FROM persons p
   WHERE p.last_name = 'Parker'; -- must return one row
  ...
END;
```

最後の例で、Person オブジェクトの属性を更新します。

```
DECLARE
  p_ref      REF Person;
  my_last_name VARCHAR2(15);
  ...
BEGIN
  ...
  SELECT REF(p) INTO p_ref FROM persons p
    WHERE p.last_name = my_last_name;
  UPDATE persons p
    SET p = Person('Jill', 'Anders', '11-NOV-67', ...)
    WHERE REF(p) = p_ref;
END;
```

参照先がない REF のテスト

ref が指すオブジェクトが削除されると、その ref は参照先がない REF として（存在しないオブジェクトを指すものとして）残ります。この状態になっているかどうかをテストするには、SQL 述語の IS DANGLING を使います。たとえば、department という関係表の manager という列に、あるオブジェクト表に格納されている Employee というオブジェクトへの ref があるとします。次の UPDATE 文を使うと、参照先がない REF を NULL に変換できます。

```
BEGIN
  UPDATE department SET manager = NULL WHERE manager IS DANGLING;
```

演算子 Deref を使う

PL/SQL のプロシージャ文の中では、ref を通してナビゲートできません。そのかわりに、SQL 文で演算子 Deref を使います。(Deref は、参照解除 (dereference) の略です。ポインタを参照解除すると、そのポインタが指していた値が得られます。) Deref はオブジェクトへの参照を引数とし、そのオブジェクトの値を戻します。参照先にオブジェクトが存在しなければ、Deref は NULL オブジェクトを戻します。

次の例では、Person オブジェクトへの ref を参照解除します。ref を選択するのが、ダミーの表 dual からであることに注意してください。オブジェクト表を指定して基準を検索する必要はありません。これは、オブジェクト表に格納されているオブジェクトがすべて、一意の変化しない識別子を持っていて、その識別子は、オブジェクトへの ref の一部となっているためです。

```
DECLARE
    p1      Person;
    p_ref REF Person;
    name    VARCHAR2(15);
BEGIN
    ...
    /* Assume that p_ref holds a valid reference
       to an object stored in an object table. */
    SELECT Deref(p_ref) INTO p1 FROM dual;
    name := p1.last_name;
```

次の例に示すように、連続するいくつかの SQL 文の中で Deref を使って ref を参照解除できます。

```
CREATE TYPE PersonRef AS OBJECT (p_ref REF Person)
/
DECLARE
    name    VARCHAR2(15);
    pr_ref REF PersonRef;
    pr      PersonRef;
    p       Person;
BEGIN
    ...
    /* Assume pr_ref holds a valid reference. */
    SELECT Deref(pr_ref) INTO pr FROM dual;
    SELECT Deref(pr.p_ref) INTO p FROM dual;
    name := p.last_name;
    ...
END
/
```

次の例に示すように、演算子 Deref をプロシージャ文の中で使うことはできません。

```
BEGIN
    ...
    p1 := Deref(p_ref); -- illegal
```

SQL 文の中では、ドット表記法を使ってオブジェクト列の ref 属性までナビゲートしたり、ある ref 属性から他の ref 属性にナビゲートしたりできます。また、表の別名を使えば、ref 列から属性にナビゲートできます。たとえば、次の構文は有効です。

```
table_alias.object_column.ref_attribute
table_alias.object_column.ref_attribute.attribute
table_alias.ref_column.attribute
```

オブジェクト型 Address と Person、およびオブジェクト表 persons を作る次の SQL*Plus スクリプトを実行したとします。

```
CREATE TYPE Address AS OBJECT (
    street  VARCHAR2(35),
    city    VARCHAR2(15),
    state   CHAR(2),
    zip_code INTEGER)
/
CREATE TYPE Person AS OBJECT (
    first_name  VARCHAR2(15),
    last_name   VARCHAR2(15),
    birthday    DATE,
    home_address REF Address, -- shared with other Person objects
    phone_number VARCHAR2(15))
/
CREATE TABLE persons OF Person
/
```

ref 属性 home_address は、オブジェクト表 persons の列に対応しています。home_address は、他の表に格納されているオブジェクト Address への ref です。表の内容を入力したら、次のように ref を参照解除することによって特定のアドレスを選択できます。

```
DECLARE
    addr1 Address,
    addr2 Address,
    ...
BEGIN
    SELECT Deref(home_address) INTO addr1 FROM persons p
    WHERE p.last_name = 'Derringer';
```

次の例では、ref 列 home_address から属性 street にナビゲートします。この場合、表の別名が必要です。

```
DECLARE
    my_street VARCHAR2(25),
    ...
BEGIN
    SELECT p.home_address.street INTO my_street FROM persons p
    WHERE p.last_name = 'Lucas';
```

オブジェクトの挿入

オブジェクトをオブジェクト表に追加するには、INSERT 文を使います。次の例では、Person オブジェクトをオブジェクトの表 persons に挿入します。

```
BEGIN
  INSERT INTO persons
    VALUES ('Jenifer', 'Lapidus', ...);
```

または、オブジェクト型 Person のコンストラクタを使ってもオブジェクトをオブジェクト表 persons にオブジェクトを追加できます。

```
BEGIN
  INSERT INTO persons
    VALUES (Person('Albert', 'Brooker', ...));
```

次の例では、RETURNING 句を使って Person の ref をローカル変数に格納します。この句が、SELECT 文でどのように解釈されるかに注意してください。RETURNING 句は、UPDATE 文と DELETE 文でも使えます。

```
DECLARE
  p1_ref REF Person;
  p2_ref REF Person;
  ...
BEGIN
  INSERT INTO persons p
    VALUES (Person('Paul', 'Chang', ...))
    RETURNING REF(p) INTO p1_ref;
  INSERT INTO persons p
    VALUES (Person('Ana', 'Thorne', ...))
    RETURNING REF(p) INTO p2_ref;
```

オブジェクトをオブジェクト表に挿入するには、同じ型のオブジェクトを戻す副問合せを使います。たとえば、

```
BEGIN
  INSERT INTO persons2
    SELECT VALUE(p) FROM persons p
    WHERE p.last_name LIKE '%Jones';
```

オブジェクト表 persons2 にコピーされた行に、新しいオブジェクト識別子が付けられます。オブジェクト識別子はオブジェクト表 persons からはコピーされません。

次のスクリプトは、Person 型の列を持つ department という関係表を作成してから、その表に行を挿入します。コンストラクタ Person() によって、列 manager に値が与えられます。

```
CREATE TABLE department (  
    dept_name VARCHAR2(20),  
    manager   Person,  
    location  VARCHAR2(20))  
/  
INSERT INTO department  
    VALUES ('Payroll', Person('Alan', 'Tsai', ...), 'Los Angeles')  
/
```

列 manager に格納される新しい Person オブジェクトは、行ではなく 1 つの列に格納されるため、オブジェクト識別子はなく、参照はできません。

オブジェクトの更新

次の例に示すように、オブジェクト表内のオブジェクトの属性を変更するには、UPDATE 文を使います。

```
BEGIN  
    UPDATE persons p SET p.home_address = '341 Oakdene Ave'  
        WHERE p.last_name = 'Brody';  
    ...  
    UPDATE persons p SET p = Person('Beth', 'Steinberg', ...)  
        WHERE p.last_name = 'Steinway';  
    ...  
END;
```

オブジェクトの削除

オブジェクト（行）をオブジェクト表から削除するには、DELETE 文を使います。オブジェクトを選択的に削除するには、次の例に示すように、WHERE 句を使います。

```
BEGIN  
    DELETE FROM persons p  
        WHERE p.home_address = '108 Palm Dr';  
    ...  
END;
```

システム固有の動的 SQL

A happy and gracious flexibility ...—Matthew Arnold

この章では、システム固有の動的 SQL（略して動的 SQL）を使用する方法について説明します。これは、アプリケーションをより柔軟で多目的に使用できるようにする PL/SQL インタフェースです。実行時に、SQL 文を即時に構築および処理できるプログラムを簡単に作成する方法を説明します。

PL/SQL 内では、任意の種類の SQL 文（データ定義文およびデータ制御文を含む）を、面倒なプログラムによるアプローチに頼ることなく実行できます。動的 SQL はプログラムに自然に取り入れることができ、プログラムをより効率的で読みやすく、簡潔なものにします。

主なトピック

[動的 SQL とは？](#)

[動的 SQL の必要性](#)

[EXECUTE IMMEDIATE 文の使用方法](#)

[OPEN-FOR、FETCH および CLOSE 文の使用方法](#)

[パラメータのモードの指定](#)

[ティップス](#)

動的 SQL とは？

PL/SQL プログラムのほとんどは、特定の予測可能な作業を実行します。たとえば、あるストアド・プロシージャは従業員番号と給与の増額を受け入れ、emp 表の sal 列を更新します。この場合、UPDATE 文のフル・テキストは、コンパイル時に認識されます。このような文は実行ごとに変わるものではありません。そのため、このような文は静的 SQL 文と呼ばれます。

しかし、プログラムによっては、さまざまな SQL 文を実行時に構築および処理する必要があります。たとえば、汎用目的の報告書作成プログラムは、生成するさまざまな報告書用に、異なる SELECT 文を構築する必要があります。この場合、文のフル・テキストは、実行時まで不定です。多くの場合、このような文は実行ごとに変わります。そのため、このような文は動的 SQL 文と呼ばれます。

動的 SQL 文は、実行時にプログラムが構築する文字列に格納されます。このような文字列は、有効な SQL 文または PL/SQL ブロックを含んでいる必要があります。また、バインド引数のプレースホルダも含むことができます。プレースホルダは宣言されていない識別子なので、名前（接頭辞としてコロンを付ける必要がある）は関係ありません。たとえば、PL/SQL は次の文字列を区別しません。

```
'DELETE FROM emp WHERE sal > :my_sal AND comm < :my_comm'  
'DELETE FROM emp WHERE sal > :s AND comm < :c'
```

動的 SQL 文を処理するには、ほとんどの場合 EXECUTE IMMEDIATE 文を使用します。ただし、複数行の問合せ（SELECT 文）を処理する場合は、OPEN-FOR 文、FETCH 文、および CLOSE 文を使用する必要があります。

動的 SQL の必要性

動的 SQL は、次のような場合に必要になります。

- SQL データ定義文（CREATE など）、データ制御文（GRANT など）、またはセッション制御文（ALTER SESSION など）を実行したい場合。PL/SQL では、このような文を静的に実行できない。
- 柔軟性が必要な場合。たとえば、スキーマ・オブジェクトの選択を実行時まで延期したい場合。または、SELECT 文の WHERE 句に対して異なる検索条件を構築したい場合。より複雑なプログラムがさまざまな SQL 操作、句などから選択される可能性がある。
- パッケージ DBMS_SQL を使用して SQL 文を動的に実行するときに、より優れたパフォーマンスや使いやすさ、またはオブジェクトやコレクションのサポートなどの DBMS_SQL にはない機能性を求める場合。（DBMS_SQL との比較は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照。）

EXECUTE IMMEDIATE 文の使用法

EXECUTE IMMEDIATE 文は、動的 SQL 文または無名 PL/SQL ブロックを準備（解析）し、即時に実行します。次に構文を示します。

```
EXECUTE IMMEDIATE dynamic_string
[INTO {define_variable[, define_variable]... | record}]
[USING [IN | OUT | IN OUT] bind_argument
      [, [IN | OUT | IN OUT] bind_argument]...];
```

dynamic_string は SQL 文または PL/SQL ブロックを表す文字列式です。define_variable は SELECT 文によって選択された列値を格納する変数、record は SELECT 文によって選択された行を格納する %ROWTYPE レコードまたはユーザー定義レコードです。bind_argument は式で、その値は動的 SQL 文または PL/SQL ブロックに渡されます。（EXECUTE IMMEDIATE 文では、NOCOPY コンパイラ・ヒントは使用できません。）

複数行の問合せの場合を除いて、文字列には任意の SQL 文（終了記号なし）または任意の PL/SQL ブロック（終了記号付き）を含むことができます。また、バインド引数のプレースホルダも含むことができます。しかし、バインド引数を使用してスキーマ・オブジェクトの名前を動的 SQL 文に渡すことはできません。正しい用法は、10-10 ページの「[スキーマ・オブジェクトの名前を渡す](#)」を参照してください。

1 行の問合せの場合に便利な INTO 句は、取り出された列値を入れる変数またはレコードを指定します。問合せが戻す列の値それぞれに対して、INTO 句の中に、対応する型互換性のあるフィールドまたは変数が存在しなければなりません。

バインド引数は、いずれも USING 句に入れる必要があります。パラメータ・モードは、指定しないとデフォルトで IN に設定されます。USING 句内のすべてのバインド引数は、実行時に SQL 文内または PL/SQL ブロック内の対応するプレースホルダを置き換えます。このため、すべてのプレースホルダを USING 句内のバインド引数に対応付ける必要があります。USING 句内では数値リテラル、文字リテラル、および文字列リテラルは使用できますが、ブール・リテラル（TRUE、FALSE、NULL）は使用できません。動的文字列に NULL を渡すには、ほかの方法を使う必要があります（10-12 ページの「[NULL を渡す](#)」を参照してください）。

動的 SQL はすべての SQL データ型をサポートしています。たとえば、定義変数やバインド引数をコレクション、LOB、オブジェクト型のインスタンス、および REF とすることができます。通常、動的 SQL は PL/SQL 固有の型をサポートしていません。たとえば定義変数やバインド引数をブールまたは索引付き表にできません。例外として、PL/SQL レコードを INTO 句に入れることができます。

注意：動的 SQL 文は、バインド引数の新しい値を使用して繰り返し実行できます。ただし、EXECUTE IMMEDIATE は実行のたびに動的文を準備しなおすため、オーバーヘッドが発生します。

例

次の PL/SQL ブロックには動的文の例がいくつか含まれています。

```
DECLARE
    sql_stmt    VARCHAR2(100);
    plsql_block VARCHAR2(200);
    my_deptno   NUMBER(2) := 50;
    my_dname    VARCHAR2(15) := 'PERSONNEL';
    my_loc      VARCHAR2(15) := 'DALLAS';
    emp_rec     emp%ROWTYPE;
BEGIN
    sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
    EXECUTE IMMEDIATE sql_stmt USING my_deptno, my_dname, my_loc;

    sql_stmt := 'SELECT * FROM emp WHERE empno = :id';
    EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING 7788;

    EXECUTE IMMEDIATE 'DELETE FROM dept
        WHERE deptno = :n' USING my_deptno;

    plsql_block := 'BEGIN emp_stuff.raise_salary(:id, :amt); END;';
    EXECUTE IMMEDIATE plsql_block USING 7788, 500;

    EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt NUMBER)';

    sql_stmt := 'ALTER SESSION SET SQL_TRACE TRUE';
    EXECUTE IMMEDIATE sql_stmt;
END;
```

次の例では、スタンドアロン・プロシージャはデータベース表の名前（'emp' など）とオプションの WHERE 句条件（'sal > 2000' など）を受け入れます。条件を省略すると、プロシージャは表の中のすべての行を削除します。条件が省略されていなければ、プロシージャは条件を満たす行だけを削除します。

```
CREATE PROCEDURE delete_rows (
    table_name IN VARCHAR2,
    condition IN VARCHAR2 DEFAULT NULL) AS
    where_clause VARCHAR2(100) := ' WHERE ' || condition;
BEGIN
    IF condition IS NULL THEN where_clause := NULL; END IF;
    EXECUTE IMMEDIATE 'DELETE FROM ' || table_name || where_clause;
EXCEPTION
    ...
END;
```

OPEN-FOR、FETCH および CLOSE 文の使用法

動的な複数行の問合せを処理するには、OPEN-FOR、FETCH、CLOSE の 3 つの文を使用します。まず、OPEN-FOR 文でカーソル変数を複数行問合せ用にオープンします。次に、FETCH 文で結果セットから一度に 1 行ずつ行を取り出します。すべての行が処理されたら、CLOSE 文でカーソル変数をクローズします。(カーソル変数の詳細は、5-14 ページの「[カーソル変数の使用](#)」を参照してください。)

カーソル変数のオープン

OPEN-FOR 文はカーソル変数を複数行の問合せと対応付け、問合せを実行し、結果を識別してカーソルを結果セットの最初の行に配置してから、%ROWCOUNT によって保持される処理行カウントをゼロに設定します。

静的形式とは異なり、OPEN-FOR の動的形式にはオプションの USING 句があります。実行時に、USING 句のバインド引数は動的 SELECT 文内の対応するプレースホルダを置き換えます。次に構文を示します。

```
OPEN {cursor_variable | :host_cursor_variable} FOR dynamic_string
    [USING bind_argument[, bind_argument]...];
```

cursor_variable は弱い型定義のカーソル変数（戻り型を持たない）です。host_cursor_variable は OCI プログラムなどの PL/SQL ホスト環境で宣言されたカーソル変数で、dynamic_string は複数行の問合せを表す文字式です。

次の例では、カーソル変数を宣言し、それをデータベース表 emp から行を戻す動的 SELECT 文と関連付けます。

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR; -- define weak REF CURSOR type
    emp_cv EmpCurTyp; -- declare cursor variable
    my_ename VARCHAR2(15);
    my_sal NUMBER := 1000;
BEGIN
    OPEN emp_cv FOR -- open cursor variable
        'SELECT ename, sal FROM emp WHERE sal > :s' USING my_sal;
    ...
END;
```

カーソル変数がオープンしている場合だけ、問合せの中のバインド引数が評価されます。このため、異なるバインド値を使用してカーソルから取り出すには、新しい値に設定されたバインド引数でカーソル変数を再オープンする必要があります。

カーソル変数からの取出し

FETCH 文は複数行の問合せの結果セットから行を戻し、選択リスト項目の値を INTO 句内の対応する変数またはフィールドに代入し、%ROWCOUNT に保持されるカウントに増分を加えて、カーソルを次の行に進めます。次に構文を示します。

```
FETCH {cursor_variable | :host_cursor_variable}
      INTO {define_variable[, define_variable]... | record};
```

次の例では、カーソル変数 emp_cv から定義変数 my_ename および my_sal へ行を取り出します。

```
LOOP
    FETCH emp_cv INTO my_ename, my_sal; -- fetch next row
    EXIT WHEN emp_cv%NOTFOUND; -- exit loop when last row is fetched
    -- process row
END LOOP;
```

カーソル変数と結び付けられた問合せが戻す列の値に対して、INTO 句の中に、対応する型互換性のあるフィールドまたは変数が存在しなければなりません。同じカーソル変数を使って、別々の FETCH 文で、異なる INTO 句を使えます。各 FETCH 文で同じ結果セットから別の行を取り出します。

クローズしている、または一度もオープンされていないカーソル変数から取出しを実行すると、PL/SQL によって事前定義の例外 INVALID_CURSOR が呼び出されます。

カーソル変数のクローズ

カーソル変数は CLOSE 文によって使用禁止になります。その後、対応付けられた結果セットは未定義になります。次に構文を示します。

```
CLOSE {cursor_variable | :host_cursor_variable};
```

次の例では、最後の行が処理された時点でカーソル変数 emp_cv をクローズします。

```
LOOP
    FETCH emp_cv INTO my_ename, my_sal;
    EXIT WHEN emp_cv%NOTFOUND;
    -- process row
END LOOP;
CLOSE emp_cv; -- close cursor variable
```

すでにクローズされているか一度もオープンされたことのないカーソル変数をクローズすると、PL/SQL によって INVALID_CURSOR が呼び出されます。

例

次の例に示すように、動的な複数行の問合せの結果セットから行を取り出してレコードに入れることができます。

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    emp_cv    EmpCurTyp;
    emp_rec   emp%ROWTYPE;
    sql_stmt  VARCHAR2(100);
    my_job    VARCHAR2(15) := 'CLERK';
BEGIN
    sql_stmt := 'SELECT * FROM emp WHERE job = :j';
    OPEN emp_cv FOR sql_stmt USING my_job;
    LOOP
        FETCH emp_cv INTO emp_rec;
        EXIT WHEN emp_cv%NOTFOUND;
        -- process record
    END LOOP;
    CLOSE emp_cv;
END;
```

次の例は、オブジェクトとコレクションの使用法を示しています。たとえば、オブジェクト型 `Person` および `VARRAY` 型 `Hobbies` を、次のように定義するとします。

```
CREATE TYPE Person AS OBJECT (name VARCHAR2(25), age NUMBER);
CREATE TYPE Hobbies IS VARRAY(10) OF VARCHAR2(25);
```

次のように動的 SQL を使用して、これらの型を使用するプロシージャのパッケージを作成できます。

```
CREATE PACKAGE teams AS
    PROCEDURE create_table (tab_name VARCHAR2);
    PROCEDURE insert_row (tab_name VARCHAR2, p Person, h Hobbies);
    PROCEDURE print_table (tab_name VARCHAR2);
END;

CREATE PACKAGE BODY teams AS
    PROCEDURE create_table (tab_name VARCHAR2) IS
    BEGIN
        EXECUTE IMMEDIATE 'CREATE TABLE ' || tab_name ||
            ' (p Person, h Hobbies)';
    END;
```

```
PROCEDURE insert_row (  
    tab_name VARCHAR2,  
    p Person,  
    h Hobbies) IS  
BEGIN  
    EXECUTE IMMEDIATE 'INSERT INTO ' || tab_name ||  
        ' VALUES (:1, :2)' USING p, h;  
END;  
  
PROCEDURE print_table (tab_name VARCHAR2) IS  
    TYPE RefCurTyp IS REF CURSOR;  
    cv RefCurTyp;  
    p Person;  
    h Hobbies;  
BEGIN  
    OPEN cv FOR 'SELECT p, h FROM ' || tab_name;  
    LOOP  
        FETCH cv INTO p, h;  
        EXIT WHEN cv%NOTFOUND;  
        -- print attributes of 'p' and elements of 'h'  
    END LOOP;  
    CLOSE cv;  
END;  
END;
```

無名 PL/SQL ブロックから、パッケージ `teams` のプロシージャを次のようにコールできます。

```
DECLARE  
    team_name VARCHAR2(15);  
    ...  
BEGIN  
    ...  
    team_name := 'Notables';  
    teams.create_table(team_name);  
    teams.insert_row(team_name, Person('John', 31),  
        Hobbies('skiing', 'coin collecting', 'tennis'));  
    teams.insert_row(team_name, Person('Mary', 28),  
        Hobbies('golf', 'quilting', 'rock climbing'));  
    teams.print_table(team_name);  
END;
```

パラメータのモードの指定

たとえば WHERE 句内などで使用される入力バインド引数に、パラメータ・モードを指定する必要はありません。モードはデフォルトで IN に設定されます。しかし、INSERT 文、UPDATE 文、または DELETE 文の RETURNING 句内で使用される出力バインド引数には、OUT モードを指定する必要があります。たとえば、

```
DECLARE
    sql_stmt VARCHAR2(100);
    old_loc  VARCHAR2(15);
BEGIN
    sql_stmt :=
        'DELETE FROM dept WHERE deptno = 20 RETURNING loc INTO :x';
    EXECUTE IMMEDIATE sql_stmt USING OUT old_loc;
    ...
END;
```

同様に、パラメータとして渡されるバインド引数には、適宜 OUT または IN OUT モードを指定する必要があります。たとえば、次のスタンドアロン・プロシージャをコールするとします。

```
CREATE PROCEDURE create_dept (
    deptno IN OUT NUMBER,
    dname  IN VARCHAR2,
    loc    IN VARCHAR2) AS
BEGIN
    deptno := deptno_seq.NEXTVAL;
    INSERT INTO dept VALUES (deptno, dname, loc);
END;
```

動的 PL/SQL ブロックからプロシージャをコールするには、次のように、仮パラメータ deptno に対応付けられたバインド引数に、IN OUT モードを指定する必要があります。

```
DECLARE
    plsql_block VARCHAR2(200);
    new_deptno  NUMBER(2);
    new_dname    VARCHAR2(15) := 'ADVERTISING';
    new_loc      VARCHAR2(15) := 'NEW YORK';
BEGIN
    plsql_block := 'BEGIN create_dept(:a, :b, :c); END;';
    EXECUTE IMMEDIATE plsql_block
        USING IN OUT new_deptno, new_dname, new_loc;
    IF new_deptno > 90 THEN ...
END;
```

ティップス

この項では、動的 SQL を十分に活用する方法、およびよく起こる問題を回避する方法を説明します。

スキーマ・オブジェクトの名前を渡す

任意のデータベース表の名前を受け入れ、その表をスキーマから削除するプロシージャが必要だとします。動的 SQL を使用して、次のスタンドアロン・プロシージャを作成します。

```
CREATE PROCEDURE drop_table (table_name IN VARCHAR2) AS
BEGIN
    EXECUTE IMMEDIATE 'DROP TABLE :tab' USING table_name;
END;
```

しかし、このプロシージャは実行時に *invalid table name* というエラーで失敗します。これは、バインド引数を使用してスキーマ・オブジェクトの名前を動的 SQL 文に渡すことはできないためです。かわりに、動的文字列にパラメータを埋め込んで、スキーマ・オブジェクトの名前をこのパラメータに渡します。

上の例をデバッグするには、EXECUTE IMMEDIATE 文を変更する必要があります。プレースホルダとバインド引数のかわりに、連結演算子を使用して、動的文字列にパラメータ table_name を埋め込みます。次に例を示します。

```
CREATE PROCEDURE drop_table (table_name IN VARCHAR2) AS
BEGIN
    EXECUTE IMMEDIATE 'DROP TABLE ' || table_name;
END;
```

これで任意のデータベース表の名前を動的 SQL 文に渡せます。

重複するプレースホルダの使用

動的 SQL 文のプレースホルダは、USING 句内のバインド引数に、名前ではなく位置によって対応付けられます。このため、SQL 文内で同じプレースホルダが 2 回以上指定されている場合、それぞれが USING 句内のバインド引数に対応する必要があります。たとえば、次のような動的文字列があるとします。

```
sql_stmt := 'INSERT INTO payroll VALUES (:x, :x, :y, :x)';
```

対応する USING 句を次のようにコーディングできます。

```
EXECUTE IMMEDIATE sql_stmt USING a, a, b, a;
```

しかし、USING 句内のバインド引数に位置によって対応付けられるのは、動的 PL/SQL ブロック内で一意のプレースホルダだけです。このため、PL/SQL ブロック内で同じプレースホルダが 2 回以上指定されている場合、そのすべてが USING 句内の 1 つのバインド引数に対応します。次の例では、1 番目の一意のプレースホルダ (x) が 1 番目のバインド引数 (a) に対応付けられます。同様に、2 番目の一意のプレースホルダ (y) が 2 番目のバインド引数 (b) に対応付けられます。

```
DECLARE
  a NUMBER := 4;
  b NUMBER := 7;
BEGIN
  plsql_block := 'BEGIN calc_stats(:x, :x, :y, :x); END;';
  EXECUTE IMMEDIATE plsql_block USING a, b;
  ...
END;
```

カーソル属性の使用

カーソルには、%FOUND、%ISOPEN、%NOTFOUND、%ROWCOUNT の 4 つの属性があります。これらの属性をカーソル名に付加すると、静的および動的 SQL 文の実行について役立つ情報が戻されます。

SQL データ操作文を処理するには、Oracle は SQL という名前の暗黙カーソルをオープンします。暗黙カーソルの属性は、直前に実行された INSERT 文、UPDATE 文、DELETE 文、または 1 行の SELECT 文に関する情報を戻します。たとえば、次のスタンドアロン・ファンクションは、%ROWCOUNT を使用して、データベース表から削除された行数の数を戻します。

```
CREATE FUNCTION rows_deleted (
  table_name IN VARCHAR2,
  condition IN VARCHAR2) RETURN INTEGER AS
BEGIN
  EXECUTE IMMEDIATE
    'DELETE FROM ' || table_name || ' WHERE ' || condition;
  RETURN SQL%ROWCOUNT; -- return number of rows deleted
END;
```

同様に、カーソル変数名にカーソル属性を追加すると、複数行の問合せの実行に関する情報が戻されます。カーソル属性の詳細は、5-31 ページの「[カーソル属性の使用](#)」を参照してください。

NULL を渡す

動的 SQL 文に NULL を渡すとします。たとえば、次の EXECUTE IMMEDIATE 文を作成します。

```
EXECUTE IMMEDIATE 'UPDATE emp SET comm = :x' USING NULL;
```

しかし、この文は *bad expression* エラーで失敗します。USING 句ではリテラル NULL を使用できないためです。この制限を回避するには、次のように、キーワード NULL を初期化されていない変数で置き換えます。

```
DECLARE
    a_null CHAR(1); -- set to NULL automatically at run time
BEGIN
    EXECUTE IMMEDIATE 'UPDATE emp SET comm = :x' USING a_null;
END;
```

リモート操作の実行

次の例に示すように、PL/SQL サブプログラムは、リモート・データベースにあるオブジェクトを参照する動的 SQL 文を実行できます。

```
PROCEDURE delete_dept (db_link VARCHAR2, dept_num INTEGER) IS
BEGIN
    EXECUTE IMMEDIATE 'DELETE FROM dept@' || db_link ||
        ' WHERE deptno = :n' USING dept_num;
END;
```

また、リモート・プロシージャ・コール (RPC) のターゲットは、動的 SQL 文を含むことができます。たとえば、表内の行の数を戻す次のスタンドアロン・ファンクションが、シカゴのデータベースに常駐するとします。

```
CREATE FUNCTION row_count (tab_name CHAR) RETURN INT AS
    rows INT;
BEGIN
    EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM ' || tab_name INTO rows;
    RETURN rows;
END;
```

無名ブロックから、次のようにしてファンクションをリモートでコールできます。

```
DECLARE
    rows INTEGER;
BEGIN
    rows := row_count@chicago('emp');
```

実行者権限の使用

デフォルトでは、ストアド・プロシージャは、実行者の権限ではなく定義者の権限で実行します。このようなプロシージャはスキーマにバインドされ、そこに常駐します。たとえば、任意のデータベース・オブジェクトを削除できる次のスタンドアロン・プロシージャが、スキーマ `scott` に常駐するとします。

```
CREATE PROCEDURE drop_it (kind IN VARCHAR2, name IN VARCHAR2) AS
BEGIN
    EXECUTE IMMEDIATE 'DROP ' || kind || ' ' || name;
END;
```

さらに、ユーザー `jones` が、このプロシージャに対する `EXECUTE` 権限を付与されているとします。次のように、ユーザー `jones` が `drop_it` をコールすると、動的 `DROP` 文がユーザー `scott` の権限で実行されます。

```
SQL> CALL drop_it('TABLE', 'dept');
```

また、表 `dept` への未修飾の参照は、スキーマ `scott` 内で解決されます。このため、プロシージャは、スキーマ `jones` ではなく、スキーマ `scott` から表を削除します。

ただし、`AUTHID` 句を使用することによって、ストアド・プロシージャをその実行者（現在のユーザー）の権限で実行できます。このようなプロシージャは、特定のスキーマにバインドされません。たとえば、次に示すバージョンの `drop_it` は、その実行者の権限で実行されます。

```
CREATE PROCEDURE drop_it (kind IN VARCHAR2, name IN VARCHAR2)
    AUTHID CURRENT_USER AS
BEGIN
    EXECUTE IMMEDIATE 'DROP ' || kind || ' ' || name;
END;
```

また、データベース・オブジェクトへの未修飾の参照は、実行者のスキーマで解決されます。詳細は、7-29 ページの「[実行者権限と定義者権限](#)」を参照してください。

プラグマ RESTRICT_REFERENCES の使用

SQL 文からコールされるファンクションは、副次作用を制御するための特定の規則に従う必要があります。（7-7 ページの「[副作用の制御](#)」を参照。）この規則に違反していないか確認するには、プラグマ `RESTRICT_REFERENCES` を使用できます。これはデータベース表またはパッケージ変数、あるいはその両方への読取りや書込みを報告するよう、コンパイラに指示します。（『Oracle8i アプリケーション開発者ガイド 基礎編』を参照。）ただし、ファンクション本体に動的 `INSERT` 文、`UPDATE` 文、または `DELETE` 文が含まれている場合、そのファンクションは常に " データベース読込み禁止状態 " および " データベース書込み禁止状態 " の規則に違反します。

デッドロックの回避

まれに、SQL データ定義文の実行によってデッドロックが発生することがあります。たとえば、次のプロシージャは自身を削除しようとしているため、デッドロックが発生します。デッドロックを回避するには、サブプログラムまたはパッケージを、使用中に ALTER で変更したり、DROP で削除したりしないでください。

```
CREATE PROCEDURE calc_bonus (emp_id NUMBER) AS
BEGIN
    ...
    EXECUTE IMMEDIATE 'DROP PROCEDURE calc_bonus';
    -- causes "timeout occurred while waiting to lock object" error
END;
```


Grammar, which knows how to control even kings.—Molière

この章は、PL/SQL 構文および方法のクイック・リファレンス・ガイドです。コマンドおよびパラメータ、その他の言語要素を並べて PL/SQL 文を作る方法を示します。また、読者の時間と手間を軽減するために、使用方法と簡単な例も示します。

主なトピック

- 代入文
- ブロック
- CLOSE 文
- コレクション・メソッド
- コレクション
- コメント
- COMMIT 文
- 定数と変数
- カーソルの属性
- カーソル変数
- カーソル
- DELETE 文
- EXCEPTION_INIT プラグマ
- 例外
- EXECUTE IMMEDIATE 文
- EXIT 文
- 式
- FETCH 文
- FORALL 文
- ファンクション
- GOTO 文
- IF 文
- INSERT 文

リテラル
LOCK TABLE 文
LOOP 文
NULL 文
オブジェクト型
OPEN 文
OPEN-FOR 文
OPEN-FOR-USING 文
パッケージ
プロシージャ
RAISE 文
レコード
RETURN 文
ROLLBACK 文
%ROWTYPE 属性
SAVEPOINT 文
SELECT INTO 文
SET TRANSACTION 文
SQL カーソル
SQLCODE ファンクション
SQLERRM ファンクション
%TYPE 属性
UPDATE 文

構文図の読み方

PL/SQL 文の構文を確認したい場合は、この構文図を左から右、上から下の順に読んでください。どの PL/SQL 文でも、この方法で検証または作成できます。

このダイアグラムは、バックス - ナウア形 (BNF) の結果を図で表したものです。このダイアグラムでは、四角形の中はキーボード、円の中はデリミタ、楕円の中は識別子です。

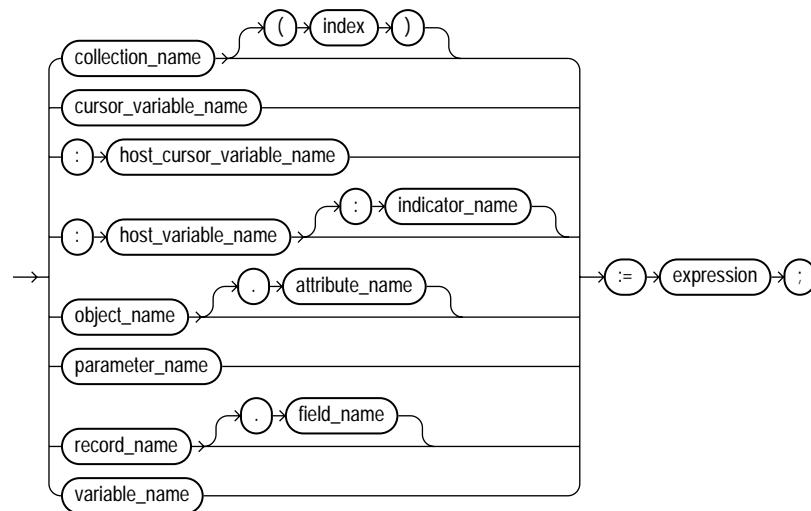
各ダイアグラムは、構文の要素を定義しています。ダイアグラムを通るパスは、それぞれがその要素のフォームとして考えられるものを表しています。矢印の向きに進んでください。線がループ状になっている場合は、そのループ内の要素を繰り返すことができます。

代入文

代入文は、変数またはフィールド、パラメータ、要素の現在の値を設定します。代入文は、代入のターゲットと、それに続く代入演算子および式で構成されています。代入文を実行すると、式が評価され、結果の値がターゲットに格納されます。詳細は、2-38 ページの「[代入](#)」を参照してください。

構文

assignment_statement



キーワードとパラメータの説明

collection_name

現在の有効範囲のうちこれより前の部分で宣言されている、ネストした表、索引付き表、または varray を指定します。

cursor_variable_name

現在の有効範囲の中で事前に宣言されている PL/SQL カーソル変数を識別します。カーソル変数に代入できるのは、別のカーソル変数の値だけです。

host_cursor_variable_name

PL/SQL ホスト環境で宣言され、バインド変数として PL/SQL に渡されるカーソル変数を識別します。ホスト・カーソル変数のデータ型は、PL/SQL カーソル変数の戻り型と互換性があります。ホスト変数には、接頭辞としてコロンをつけなければなりません。

host_variable_name

PL/SQL ホスト環境で宣言され、バインド変数として PL/SQL に渡される変数を識別します。ホスト変数には、接頭辞としてコロンをつけなければなりません。

object_name

現在の有効範囲のうちこれより前の部分で宣言されているオブジェクト（オブジェクト型のインスタンス）を指定します。

indicator_name

PL/SQL ホスト環境で宣言され、PL/SQL に渡される標識変数を識別します。標識変数には、接頭辞としてコロンをつけなければなりません。標識変数は、対応付けられたホスト変数の値または条件を示します。たとえば、Oracle プリコンパイラ環境では、標識変数を使って出力ホスト変数内の NULL や切り捨てられた値を検出できます。

parameter_name

代入文が使われているサブプログラムの仮パラメータ OUT または IN OUT を識別します。

index

結果が BINARY_INTEGER 型の値、またはその型に暗黙的に変換可能な値になる数値式です。

record_name.field_name

現在の有効範囲の中で事前に宣言されているユーザー定義のレコードまたは %ROWTYPE 属性のレコードのフィールドを識別します。

variable_name

現在の有効範囲の中で事前に宣言されている PL/SQL 変数を識別します。

expression

変数、定数、リテラル、演算子、ファンクション・コールの任意の組合せです。最も単純な式は、1 個の変数で構成されています。expression の構文は、11-62 ページの「式」を参照してください。代入文を実行すると、式が評価され、結果の値が代入のターゲットに格納されます。値とターゲットはデータ型に互換性がなければなりません。

使用方法

デフォルトでは、宣言で初期化されていない変数は、ブロックまたはサブプログラムに入るたびに `NULL` に初期化されます。したがって、値を代入する前に変数を参照しないようにしてください。

`NOT NULL` と定義されている変数には `NULL` を代入できません。`NULL` を代入しようとすると、`PL/SQL` は事前定義の例外 `VALUE_ERROR` を呼び出します。

ブール変数に代入できるのは、値 `TRUE` と `FALSE`、および `NULL` だけです。式に関係演算子を適用するとブール値が戻されます。したがって、次の代入は有効です。

```
DECLARE
    out_of_range BOOLEAN;
    ...
BEGIN
    ...
    out_of_range := (salary < minimum) OR (salary > maximum);
```

次の例に示すように、式の値は、レコードの特定のフィールドに代入できます。

```
DECLARE
    emp_rec emp%ROWTYPE;
BEGIN
    ...
    emp_rec.sal := current_salary + increase;
```

さらに、レコードの全フィールドに一度に値を代入できます。`PL/SQL` では、レコードの宣言で同じカーソルまたは表が参照されている場合は、レコード全体の間での一括代入ができます。たとえば、次の代入は有効です。

```
DECLARE
    emp_rec1 emp%ROWTYPE;
    emp_rec2 emp%ROWTYPE;
    dept_rec dept%ROWTYPE;
BEGIN
    ...
    emp_rec1 := emp_rec2;
```

代入文で値のリストをレコードに代入できません。このため、次の代入は誤りです。

```
dept_rec := (60, 'PUBLICITY', 'LOS ANGELES');
```

次の構文を使うと、コレクション中の特定の要素に式の値を代入できます。

```
collection_name(index) := expression;
```

次の例では、`last_name` を大文字に変換した値を、ネストした表 `ename_tab` の 3 番目の行に代入しています。

```
ename_tab(3) := UPPER(last_name);
```

例

代入文の例を次に示します。

```
wages := hours_worked * hourly_salary;  
country := 'France';  
costs := labor + supplies;  
done := (count > 100);  
dept_rec.loc := 'BOSTON';  
comm_tab(5) := sales * 0.15;
```

関連項目

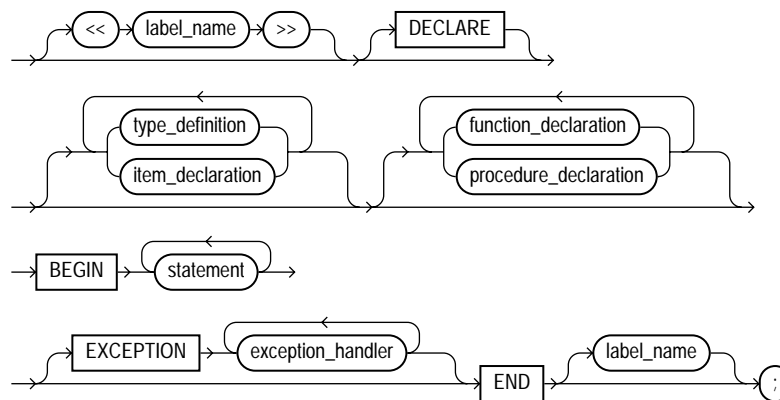
定数と変数、式、SELECT INTO 文

ブロック

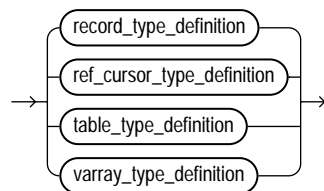
PL/SQL におけるプログラムの基本単位はブロックです。PL/SQL ブロックは、キーワード DECLARE および BEGIN、EXCEPTION、END で定義します。これらのキーワードは、ブロックを宣言部、実行部、例外処理部に分けます。このうち必ず存在しなければならないのは実行部だけです。ブロックの中では、実行可能文を置ける場所ならば別のブロックをネストできます。詳細は、1-2 ページの「[ブロック構造](#)」および 2-35 ページの「[有効範囲と可視性](#)」を参照してください。

構文

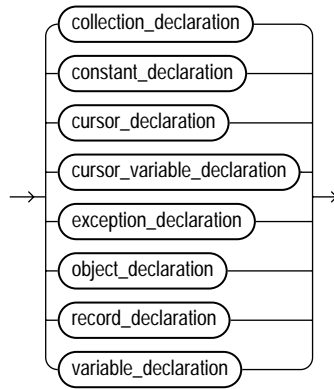
plsql_block



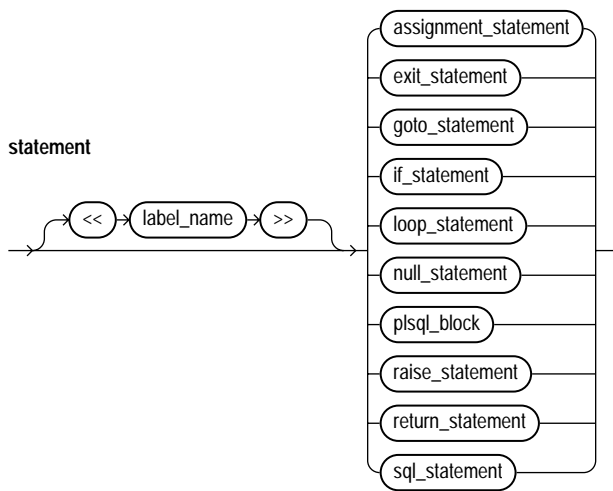
type_definition



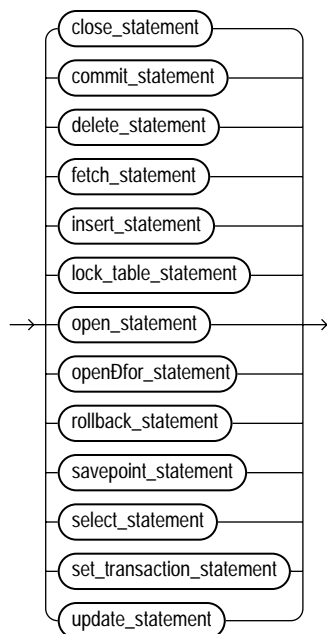
item_declaration



statement



sql_statement



キーワードとパラメータの説明

label_name

オプションとして PL/SQL ブロックに付けるラベル名で、未宣言の識別子です。label_name を使う場合は、二重の山カッコで囲み、ブロックの先頭に置かなければなりません。オプションとして、label_name を、山カッコで囲まずに、ブロックの最後に置くこともできます。

外側のブロックで宣言されたグローバル識別子を、サブブロックで再宣言できます。この場合、ローカルな宣言が優先され、サブブロックではグローバル識別子を参照できなくなります。グローバル識別子を参照する場合は、次の例に示すように、ブロック・ラベルを使って参照を修飾します。

```

<<outer>>
DECLARE
    x INTEGER;
BEGIN
    ...
    DECLARE
        x INTEGER;
  
```

```
BEGIN
    ...
    IF x = outer.x THEN -- refers to global x
        ...
    END IF;
END;
END outer;
```

DECLARE

このキーワードは、PL/SQL ブロックの宣言部の開始を示します。宣言部にはローカル宣言が置かれます。ローカルに宣言された項目は現在のブロックとその全サブブロックでしか存在せず、外側のブロックからは見えません。PL/SQL ブロックの宣言部はオプションです。宣言部は、ブロックの実行部の開始を示すキーワード `BEGIN` によって暗黙的に終了します。

PL/SQL では前方参照ができません。このため、宣言文などの他の文で項目を参照するときは、事前に宣言しておく必要があります。ただし、サブプログラムは、その他すべてのプログラム項目の後の宣言部の末尾で宣言しなければなりません。

collection_declaration

現在の有効範囲のうちこれより前の部分で宣言されている、索引付き表、ネストした表、または `varray` を指定します。`collection_declaration` の構文は、11-21 ページの「[コレクション](#)」を参照してください。

constant_declaration

定数の宣言です。`constant_declaration` の構文は、11-29 ページの「[定数と変数](#)」を参照してください。

cursor_declaration

明示カーソルの宣言です。`cursor_declaration` の構文は、11-44 ページの「[カーソル](#)」を参照してください。

cursor_variable_declaration

カーソル変数の宣言です。`cursor_variable_declaration` の構文は、11-38 ページの「[カーソル変数](#)」を参照してください。

exception_declaration

例外の宣言です。`exception_declaration` の構文は、11-54 ページの「[例外](#)」を参照してください。

object_declaration

現在の有効範囲のうちこれより前の部分で宣言されているオブジェクト（オブジェクト型のインスタンス）を指定します。object_declaration の構文は、11-102 ページの「[オブジェクト型](#)」を参照してください。

record_declaration

ユーザー定義のレコードの宣言です。record_declaration の構文は、11-130 ページの「[レコード](#)」を参照してください。

variable_declaration

変数の宣言です。variable_declaration の構文は、11-29 ページの「[定数と変数](#)」を参照してください。

function_declaration

ファンクションの宣言です。function_declaration の構文は、11-78 ページの「[ファンクション](#)」を参照してください。

procedure_declaration

プロシージャの宣言です。procedure_declaration の構文は、11-123 ページの「[プロシージャ](#)」を参照してください。

BEGIN

PL/SQL ブロックの実行部の開始を示すキーワードです。実行部には実行可能文が置かれます。ブロックの実行部は必ず存在しなければなりません。つまり、ブロックには少なくとも 1 つの実行可能文が含まれていなければなりません。NULL 文はこの条件を満たします。

statement

これは、アルゴリズムを作成するために使う実行可能文（宣言文ではなく）です。一連の文の中には、RAISE などのプロシージャ文、UPDATE などの SQL 文、および PL/SQL ブロック（ブロック文と呼ぶことがあります）を含めることができます。

PL/SQL 文は自由形式です。つまり、PL/SQL 文は、キーワード、デリミタ、リテラルが複数の行にまたがらない限り、何行でも続けることができます。文の終わりは、セミコロン（;）です。

PL/SQL は、データ操作およびカーソル制御、トランザクション制御文を含む SQL 文のサブセットをサポートしています。ただし、ALTER、CREATE、GRANT、REVOKE などのデータ定義およびデータ制御文はサポートしていません。

EXCEPTION

PL/SQL ブロックの例外処理部の開始を示すキーワードです。例外が呼び出されると、ブロックの通常の実行が停止され、制御が適切な例外ハンドラに移ります。例外ハンドラが終了すると、ブロック直後の文から実行が再開されます。

呼び出された例外の例外ハンドラが現在のブロックに存在しないと、制御は外側のブロックに渡されます。この過程が、例外ハンドラが見つかるまで、または外側にブロックがなくなるまで繰り返されます。PL/SQL が、例外を処理するための例外ハンドラを見つけられない場合、実行は停止され、「未処理例外 (*unhandled exception*)」エラーがホスト環境に戻されます。詳細は、[第 6 章](#)を参照してください。

exception_handler

例外ハンドラです。例外が呼び出されると、その例外に結び付けられた一連の文を実行します。exception_handler の構文は、11-54 ページの「[例外](#)」を参照してください。

END

PL/SQL ブロックの終わりを示すキーワードです。これはブロック中の最後のキーワードでなければなりません。IF 文の END IF、または LOOP 文の END LOOP のいずれも、キーワード END の代わりとしては使用できません。

END はトランザクションの終わりを通知しません。ブロックが複数のトランザクションにまたがることができるように、トランザクションも複数のブロックにまたがることができます。

例

次の PL/SQL ブロックでは、複数の変数と定数を宣言し、データベース表から選択した値を使って比率を計算しています。

```
-- available online in file 'examp11'
DECLARE
    numerator    NUMBER;
    denominator  NUMBER;
    the_ratio    NUMBER;
    lower_limit  CONSTANT NUMBER := 0.72;
    samp_num     CONSTANT NUMBER := 132;
BEGIN
    SELECT x, y INTO numerator, denominator FROM result_table
        WHERE sample_id = samp_num;
    the_ratio := numerator/denominator;
    IF the_ratio > lower_limit THEN
        INSERT INTO ratio VALUES (samp_num, the_ratio);
    ELSE
        INSERT INTO ratio VALUES (samp_num, -1);
    END IF;
    COMMIT;
```

```
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    INSERT INTO ratio VALUES (samp_num, 0);
    COMMIT;
  WHEN OTHERS THEN
    ROLLBACK;
END;
```

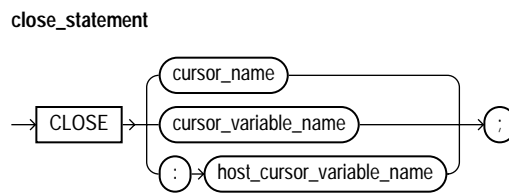
関連項目

定数と変数、例外、ファンクション、プロシージャ

CLOSE 文

CLOSE 文を使うと、オープンされているカーソルまたはカーソル変数が占有しているリソースを再利用できます。クローズされたカーソルまたはカーソル変数からは行を取り出すことができません。詳細は、5-6 ページの「[カーソル管理](#)」を参照してください。

構文



キーワードとパラメータの説明

cursor_name

現在の有効範囲の中で事前に宣言され、現在オープンされている明示カーソルを識別します。

cursor_variable_name

現在の有効範囲の中で事前に宣言され、現在オープンされている PL/SQL カーソル変数（またはパラメータ）を識別します。

host_cursor_variable_name

PL/SQL ホスト環境で宣言され、バインド変数として PL/SQL に渡されるカーソル変数を識別します。ホスト・カーソル変数のデータ型は、PL/SQL カーソル変数の戻り型と互換性があります。ホスト変数には、接頭辞としてコロンをつけなければなりません。

使用方法

いったんクローズしたカーソルまたはカーソル変数を再オープンする場合は、それぞれ OPEN 文または OPEN-FOR 文を使います。カーソルをいったんクローズせずに再オープンすると、PL/SQL によって事前定義の例外 `CURSOR_ALREADY_OPEN` が呼び出されます。ただし、カーソル変数を再オープンする場合、その前にクローズする必要はありません。

すでにクローズされているか一度もオープンされたことのないカーソルまたはカーソル変数をクローズしようとすると、PL/SQL によって事前定義の例外 `INVALID_CURSOR` が呼び出されます。

例

次の例では、最終行が取り出され、処理されてからカーソル変数 `emp_cv` をクローズします。

```
LOOP
    FETCH emp_cv INTO emp_rec;
    EXIT WHEN emp_cv%NOTFOUND;
    ... -- process data record
END LOOP;
/* Close cursor variable. */
CLOSE emp_cv;
```

関連項目

FETCH 文、OPEN 文、OPEN-FOR 文

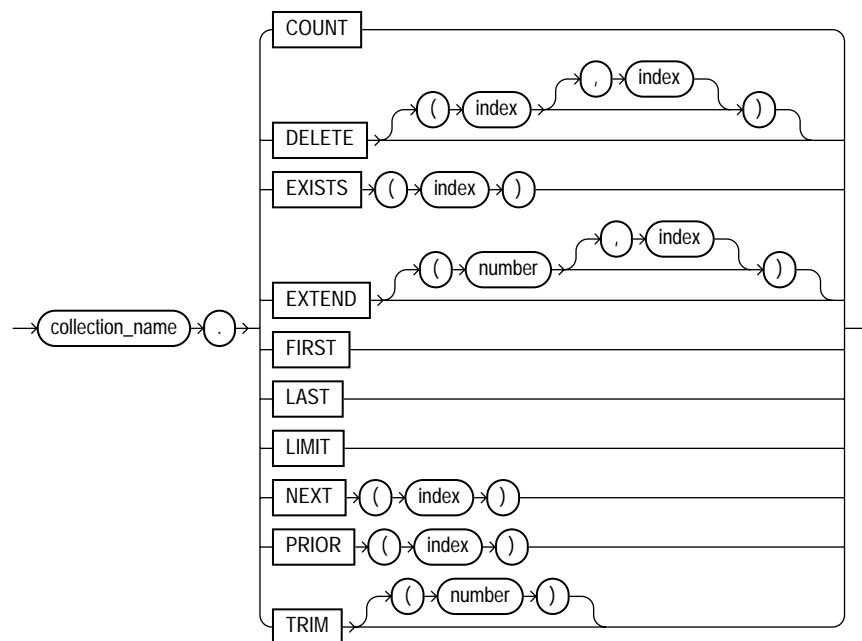
コレクション・メソッド

コレクション・メソッドとは、コレクションに対する操作を実行するための、ドット表記法を使ってコールされる組み込みファンクションまたはプロシージャです。メソッド EXISTS、COUNT、LIMIT、FIRST、LAST、PRIOR、NEXT、EXTEND、TRIM、DELETE は、コードを一般化し、コレクションを使いやすくして、アプリケーションをメンテナンスしやすくするのに役立ちます。

EXISTS、COUNT、LIMIT、FIRST、LAST、PRIOR、および NEXT は、式の一部として使われるファンクションです。EXTEND、TRIM、DELETE は、文として使われるプロシージャです。EXISTS、PRIOR、NEXT、TRIM、EXTEND、DELETE は整数パラメータを取ります。詳細は、4-20 ページの「[コレクション・メソッドの使用](#)」を参照してください。

構文

collection_method_call



キーワードとパラメータの説明

collection_name

現在の有効範囲のうちこれより前の部分で宣言されている、索引付き表、ネストした表、または varray を指定します。

COUNT

COUNT は、コレクションに現在含まれている要素の数を戻します。コレクションの現在のサイズは不明の場合があるので、そのようなときに役立ちます。COUNT は、整数式が使える位置ならどこでも使えます。

varray の場合、COUNT は常に LAST と同じです。ネストした表の場合、COUNT は通常、LAST と同じです。しかし、ネストした表の途中から要素を削除すると、COUNT は LAST より小さくなります。

DELETE

このプロシージャには 3 つの形式があります。DELETE は、コレクションからすべての要素を削除します。DELETE (n) は、ネストした表から n 番目の要素を削除します。n が NULL である場合、DELETE (n) は何も実行しません。DELETE (m,n) は、ネストした表から m ~ n の範囲のすべての要素を削除します。m が n より大きい場合、または m が n が NULL である場合、DELETE (m,n) は何も実行しません。

index

結果が整数になる（または暗黙のうちに整数に変換される）式。詳細は、2-25 ページの「[データ型の変換](#)」を参照してください。

EXISTS

EXISTS (n) は、コレクションに n 番目の要素が存在する場合に TRUE を戻します。ない場合、EXISTS (n) は FALSE を戻します。主に EXISTS は、DELETE とともに、疎であるネストした表のメンテナンスのために使います。また、EXISTS を使うことによって、存在しない要素を参照した場合に発生する例外を避けることができます。範囲外の添字を渡した場合、EXISTS は SUBSCRIPT_OUTSIDE_LIMIT を呼び出さずに、FALSE を戻します。

EXTEND

このプロシージャには 3 つの形式があります。EXTEND は、コレクションに 1 つの NULL 要素を追加します。EXTEND (n) は、コレクションに n 個の NULL 要素を追加します。EXTEND (n,i) は、コレクションに i 番目の要素のコピーを n 個追加します。EXTEND は、コレクションの内部サイズに対して操作します。そのため、EXTEND は削除された要素を見つけると、それらの要素を数に含めます。

FIRST、LAST

FIRST と LAST は、それぞれコレクションの最初と最後（最小と最大）の索引番号を戻します。コレクションが空なら、FIRST と LAST は NULL を戻します。コレクションに含まれる要素の数が 1 つだけの場合、FIRST と LAST は同じ索引番号を戻します。

varray の場合、FIRST は常に 1 を戻し、LAST は常に COUNT と同じです。ネストした表の場合、LAST は通常、COUNT と同じです。しかし、ネストした表の途中から要素を削除すると、LAST は COUNT より大きくなります。

LIMIT

最大サイズがないネストした表の場合、LIMIT は NULL を戻します。varray の場合、LIMIT は varray に入れることのできる（型定義で指定する必要がある）要素の最大数を戻します。

NEXT、PRIOR

PRIOR(n) は、コレクション内の索引 n の前の索引番号を戻します。NEXT(n) は、索引 n の後の索引番号を戻します。n の前の番号がない場合、PRIOR(n) は NULL を戻します。同様に、n の後の番号がない場合、NEXT(n) は NULL を戻します。

TRIM

このプロシージャには 2 つの形式があります。TRIM は、コレクションの末尾から 1 つの要素を削除します。TRIM(n) は、コレクションの末尾から n 個の要素を削除します。n が COUNT より大きいと、TRIM(n) は SUBSCRIPT_BEYOND_COUNT を呼び出します。

TRIM は、コレクションの内部サイズに対して操作します。そのため、TRIM は削除された要素を見つけると、それらの要素を数に含めます。

使用方法

コレクション・メソッドは SQL 文では使えません。使うと、コンパイル・エラーになります。

アトミック NULL であるコレクションに適用されるのは EXISTS だけです。それ以外のメソッドをそのようなコレクションに適用すると、PL/SQL は COLLECTION_IS_NULL を呼び出します。

PRIOR または NEXT を使うことにより、任意の添字列を索引とするコレクション内を移動できます。たとえば、PRIOR または NEXT を使うことにより、要素をいくつか削除したネストした表内を移動できます。

EXTEND は、削除された要素を含むコレクションの内部サイズに対して操作します。EXTEND を使って、アトミック NULL であるコレクションの初期化はできません。また、NOT NULL 制約を TABLE または VARRAY 型に指定した場合、EXTEND の最初の 2 つの形式はその型のコレクションに適用できません。

削除対象の要素が存在しない場合でも、DELETE は単にその要素をスキップするので、例外は呼び出されません。varray は密であるため、個々の要素は削除できません。

PL/SQL は、削除された要素のプレースホルダを保持します。そのため、削除された要素に単に新しい値を代入するだけでその要素を置き換えることができます。しかし、PL/SQL は切り捨てられた (TRIM) 要素のプレースホルダは保持しません。

ネストした表に割り当てられるメモリーの量は、動的に増減します。要素を削除すると、メモリーはページ単位で解放されます。表全体を削除した場合は、すべてのメモリーが解放されます。

一般に、TRIM と DELETE の間の相互作用には依存しないようにしてください。ネストした表は、固定サイズの配列のように扱って DELETE だけを使うか、またはスタックのように扱って TRIM と EXTEND だけを使うとよいでしょう。

サブプログラム内で、コレクション・パラメータは引数のプロパティがバインドされていることを前提にしています。そのため、メソッド FIRST、LAST、COUNT などを経験的なパラメータに適用できます。varray パラメータの場合、パラメータ・モードに関係なく、LIMIT の値は常にパラメータの型定義から派生します。

例

次の例では、NEXT を使って、いくつかの要素が削除されたネストした表内を移動しています。

```
i := courses.FIRST; -- get subscript of first element
WHILE i IS NOT NULL LOOP
    -- do something with courses(i)
    i := courses.NEXT(i); -- get subscript of next element
END LOOP;
```

次の例では、要素 i が存在する場合にだけ代入文が実行されます。

```
IF courses.EXISTS(i) THEN
    courses(i) := new_course;
END IF;
```

次の例に示すように、FIRST と LAST を使って、ループ範囲の下限と上限を指定できます (ただし、その範囲内にそれぞれの要素が存在することが必要です)。

```
FOR i IN courses.FIRST..courses.LAST LOOP ...
```

次の例では、ネストした表の要素 2 ~ 5 を削除します。

```
courses.DELETE(2, 5);
```

次の例では、LIMIT を使って、varrayprojects にさらに 20 の要素を追加できるかどうかを調べています。

```
IF (projects.COUNT + 20) < projects.LIMIT THEN  
    -- add 20 more elements
```

関連項目

コレクション

コレクション

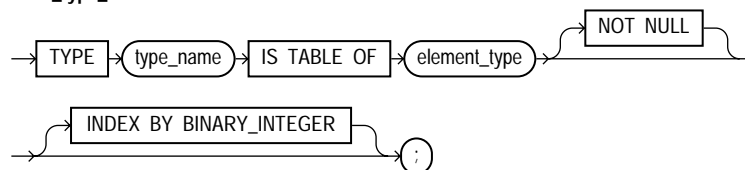
コレクションは、すべて同じ型の要素の順序付きグループです（あるクラスの生徒の成績など）。各要素には一意の添字が付いています。その番号によって、集合の中での要素の位置が決まります。PL/SQL には、索引付き表、ネストした表、varray（可変サイズの配列）の、3 種類のコレクションがあります。ネストした表は、索引付き表（以前の PL/SQL 表）の機能を拡張します。

コレクションは、ほとんどの第 3 世代のプログラミング言語で見られる配列のような働きをします。ただし、コレクションには 1 次元しかないため、添字を整数にしなければなりません。（Ada および Pascal などの言語では、複数次元の配列が可能であり、添字を列挙型にすることができます）。

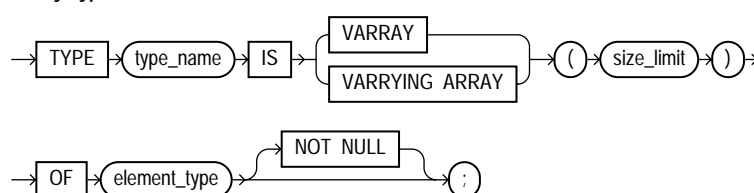
ネストした表や varray にオブジェクト型のインスタンスを格納したり、また、逆にネストした表や varray がオブジェクト型の属性であったりします。また、コレクションはパラメータとして渡すこともできます。そのため、それらを使うことにより、データの列をデータベースの表に入れたり、データの列をデータベースの表から出したり、クライアント側アプリケーションとストアード・サブプログラムとの間でデータの列を移動したりできます。詳細は、4-5 ページの「[コレクションの定義と宣言](#)」を参照してください。

構文

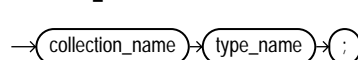
table_type_definition

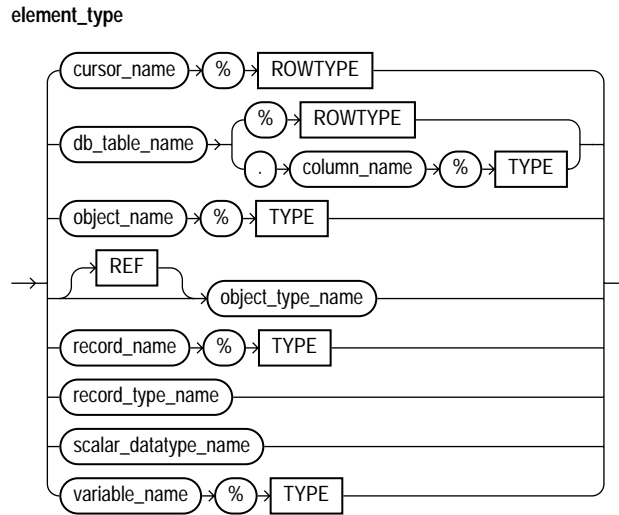


varray_type_definition



collection_declaration





キーワードとパラメータの説明

type_name

コレクションの宣言で使うユーザー定義の型指定子。

element_type

これは、BINARY_INTEGER、BOOLEAN、LONG、LONG RAW、NATURAL、NATURALN、NCHAR、NCLOB、NVARCHAR2、TABLE 属性か VARRAY 属性を持つオブジェクト型、PLS_INTEGER、POSITIVE、POSITIVEN、REF CURSOR、SIGNTYPE、STRING、TABLE、VARRAY 以外の PL/SQL データ型です。さらに varray では、element_type は BLOB、CLOB、または BLOB 属性か CLOB 属性を持つオブジェクト型にできません。element_type がレコード型である場合、そのレコード中のすべてのフィールドはスカラー型かオブジェクト型でなければなりません。

INDEX BY BINARY_INTEGER

これはオプションの句であり、バージョン 8 で索引付き表と呼ばれます。バージョン 2 では PL/SQL 表を定義するのに使います。

size_limit

これは正の整数のリテラルであり、varray の最大サイズ、つまり varray に格納できる要素数の最大値を指定します。

使用方法

ネストした表は、索引付き表の機能を拡張したもので、いくつかの点で異なります。4-3 ページの「[ネストした表と索引付き表との相違点](#)」を参照してください。

すべての要素参照は、コレクション名とカッコで囲んだ添字から構成されます。添字によって、処理される要素が決まります。索引付き表を除き、コレクションの添字の下限は 1（固定）です。索引付き表は負の添字を持つことができます。

3 つのすべてのコレクション型は、任意の PL/SQL ブロックまたはサブプログラム、パッケージの宣言部で定義できます。しかし、CREATE を使用して作成し、Oracle データベース内に格納できるのは、ネストした表と varray 型だけです。

索引付き表とネストした表は疎である（添字が連続していない）場合があります。しかし、varray は常に密です（添字が連続している）。ネストした表とは異なり、varray はデータベースに格納されるときにその順序と添字が保たれます。

索引付き表は最初は疎です。そのため、たとえば、数値の主キー（口座番号や従業員番号など）を索引として使って、参照データを一時索引付き表に格納できます。

コレクションは、通常の有効範囲とインスタンス生成の規則に従います。パッケージの中では、そのパッケージが初めて参照された時点でコレクションのインスタンスが生成され、データベース・セッションが終わった時点で消滅します。ブロックまたはサブプログラムの中で、ローカル・コレクションは、ブロックまたはサブプログラムに入ったときにインスタンス生成され、ブロックまたはサブプログラムが終了した時点で消滅します。

ネストした表または varray は、初期化するまではアトミック NULL（つまりコレクションの要素ではなく、コレクション自体が NULL）です。ネストした表または varray を初期化するには、コンストラクタ（コレクション型と同じ名前のシステム定義ファンクション）を使います。このファンクションは、コレクションに渡される要素から、コレクションを「構成（コンストラクト）」します。

ネストした表と varray は、アトミック NULL であることがあるため、NULL かどうかをテストできます。しかし、等価、不等価の比較はできません。この制限は、暗黙の比較にも適用されます。たとえば、コレクションは DISTINCT、GROUP BY、または ORDER BY リストには使えません。

コレクションにオブジェクト型のインスタンスを格納したり、また、逆にコレクションがオブジェクト型の属性であったりします。また、コレクションはパラメータとして渡すこともできます。そのため、それらを使うことにより、データの列をデータベースの表に入れたり、データの列をデータベースの表から出したり、クライアント側アプリケーションとストアド・サブプログラムとの間でデータの列を移動したりできます。

コレクションを戻すファンクションをコールする場合、次の構文を使ってコレクション内の要素を参照します。

```
collection_name (parameter_list) (subscript)
```

Oracle コール・インタフェース (OCI) または Oracle プリコンパイラを使うと、サブプログラムの仮パラメータとして宣言された コレクションにホスト配列をバインドできます。これによって、ホスト配列をストアド・ファンクションやプロシージャに渡すことができます。

例

コレクションの要素型を指定するには、次の例に示すようにして、%TYPE または %ROWTYPE を使います。

```
DECLARE
    TYPE JobList IS VARRAY(10) OF emp.job%TYPE;  -- based on column
    CURSOR c1 IS SELECT * FROM dept;
    TYPE DeptFile IS TABLE OF c1%ROWTYPE;  -- based on cursor
    TYPE EmpFile IS VARRAY(150) OF emp%ROWTYPE;  -- based on database table
```

次の例では、RECORD 型を使って、要素型を指定しています。

```
DECLARE
    TYPE Entry IS RECORD (
        term    VARCHAR2(20),
        meaning VARCHAR2(200));
    TYPE Glossary IS VARRAY(250) OF Entry;
```

次の例では、レコードの索引付き表を宣言しています。表の要素のそれぞれに、emp データベース表の行が格納されます。

```
DECLARE
    TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE
        INDEX BY BINARY_INTEGER;
    emp_tab EmpTabTyp;
BEGIN
    /* Retrieve employee record. */
    SELECT * INTO emp_tab(7468) FROM emp WHERE empno = 7788;
```

VARRAY 型の定義では、その最大サイズを指定する必要があります。次の例では、366 個までの日付を格納する型を定義します。

```
DECLARE
    TYPE Calendar IS VARRAY(366) OF DATE;
```

一度コレクション型を定義したなら、次の SQL*Plus スクリプトに示すようにしてその型のコレクションを宣言できます。

```
CREATE TYPE Project AS OBJECT(
    project_no NUMBER(2),
    title      VARCHAR2(35),
    cost       NUMBER(7,2))
/
CREATE TYPE ProjectList AS VARRAY(50) OF Project  -- VARRAY type
```



```

/
CREATE TABLE department (
    idnum    NUMBER(2),
    name     VARCHAR2(15),
    budget   NUMBER(11,2),
    projects ProjectList) -- declare varray
/

```

識別子 `projects` は varray 全体を表します。projects の各要素には、Project オブジェクトが格納されます。

次の例では、ネストした表をパッケージ・プロシージャの仮パラメータとして宣言しています。

```

CREATE PACKAGE personnel AS
    TYPE Staff IS TABLE OF Employee;
    ...
    PROCEDURE award_bonuses (members IN Staff);

```

次の例に示すように、ファンクション仕様部の RETURN 句の中にコレクション型を指定できます。

```

DECLARE
    TYPE SalesForce IS VARRAY(20) OF Salesperson;
    FUNCTION top_performers (n INTEGER) RETURN SalesForce IS ...

```

次の例では、セキュリティ部門に割り当てられているプロジェクトのリストを更新します。

```

DECLARE
    new_projects ProjectList :=
        ProjectList(Project(1, 'Issue New Employee Badges', 13500),
                     Project(2, 'Inspect Emergency Exits', 1900),
                     Project(3, 'Upgrade Alarm System', 3350),
                     Project(4, 'Analyze Local Crime Stats', 825));
BEGIN
    UPDATE department
        SET projects = new_projects WHERE name = 'Security';

```

次の例では、会計部門のすべてのプロジェクトを取り出してローカル varray に入れます。

```

DECLARE
    my_projects ProjectList;
BEGIN
    SELECT projects INTO my_projects FROM department
        WHERE name = 'Accounting';

```

関連項目

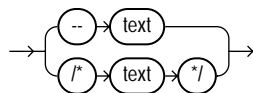
コレクション・メソッド、オブジェクト型、レコード

コメント

コメントは、コード・セグメントの目的と使用方法を明示して、わかりやすくするために使います。PL/SQL では、1 行コメントと複数行コメントの 2 種類のコメント・スタイルをサポートしています。1 行コメントは、行の中の任意の位置にある二重ハイフン (--) から始まり、その行の終わりまで続きます。複数行コメントは、スラッシュ - アスタリスク (/*) で始まってアスタリスク - スラッシュ (*/) で終わり、複数行にまたがることができます。詳細は、2-9 ページの「[コメント](#)」を参照してください。

構文

comment



使用方法

コメントは、行の末尾ならば、文の途中に置くこともできます。ただしコメントのネストはできません。

また、Oracle プリコンパイラ・プログラムが動的に処理する PL/SQL ブロックの中では、1 行コメントは使えません。これは、行の終わりを示す文字が無視されるので、1 行コメントが行の終わりではなくブロックの終わりまで続いてしまうためです。この場合、複数行コメントを使ってください。

プログラムのテストやデバッグのときに、コード中の 1 つの行を使用不能にしたい場合があります。行を「コメントにする」方法を次の例に示します。

```
-- UPDATE dept SET loc = my_loc WHERE deptno = my_deptno;
```

複数行コメントのデリミタを使うと、コードの一部をすべてコメントにできます。

例

様々なコメントのスタイルを次の例に示します。

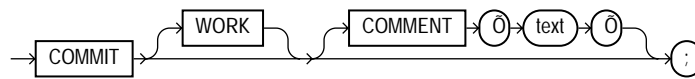
```
-- compute the area of a circle
area := pi * radius**2; -- pi equals 3.14159
/* Compute the area
   of a circle. */
area := pi * radius**2; /* pi equals 3.14159 */
```

COMMIT 文

COMMIT 文は、現在のトランザクションの間にデータベースに加えられた変更を、明示的に確定します。データベースに加えられた変更は、コミットされるまで確定されたとはみなされません。また、コミットは変更内容が他のユーザーからも見えるようにします。詳細は、5-37 ページの「[トランザクション処理](#)」を参照してください。

構文

commit_statement



キーワードとパラメータの説明

WORK

このキーワードはオプションで、コードをわかりやすくするという効果しか持ちません。

COMMENT

コメントを、現在のトランザクションに対応付けるキーワードです。分散トランザクションでよく使われます。このテキストは引用符で囲んだ 50 文字以内のリテラルでなければなりません。

使用方法

COMMIT 文はすべての行と表のロックを解除します。また、最後のコミットまたはロールバック以降にマークされたすべてのセーブポイントを消去します。変更がコミットされるまでは、次のような状況になっています。

- 自分で変更を加えた表に問合せを発行するとその変更内容が見えるが、他のユーザーから変更内容は見えない。
- 考えを変えた場合や間違いを修正する場合は、ROLLBACK 文を使って変更内容をロールバック（取消し）できる。

FOR UPDATE カーソルがオープンしているときにコミットした場合、そのカーソルでそれ以降フェッチすると例外が呼び出されます。ただし、カーソルはオープンしたままなので、クローズしてください。詳細は、5-43 ページの「[FOR UPDATE の使用](#)」を参照してください。

分散トランザクションの実行に失敗した場合は、COMMENT で指定されたテキストが問題点の診断に役立ちます。分散トランザクションが疑わしい状態（in doubt）になった場合、

COMMIT 文

Oracle はこのテキストをトランザクション ID とともにデータ・ディクショナリに格納します。分散トランザクションの詳細は、『Oracle8i 概要』を参照してください。

SQL では、FORCE 句はインダウト分散トランザクションを手動でコミットする句です。ただし、PL/SQL ではこの句はサポートされていません。たとえば、次の文は誤りです。

```
COMMIT WORK FORCE '23.51.54'; -- illegal
```

埋込み SQL では、RELEASE オプションは、プログラムに保持されるすべての Oracle リソース（ロックおよびカーソル）を解放し、データベースから切断します。ただし、PL/SQL ではこのオプションはサポートされていません。たとえば、次の文は誤りです。

```
COMMIT WORK RELEASE; -- illegal
```

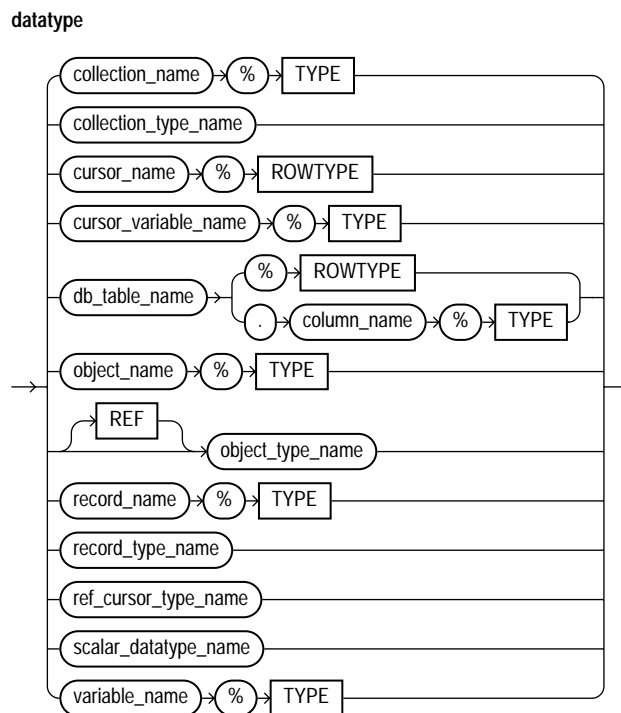
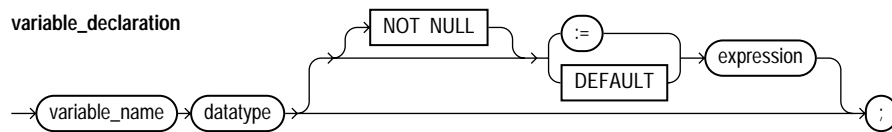
関連項目

ROLLBACK 文、SAVEPOINT 文

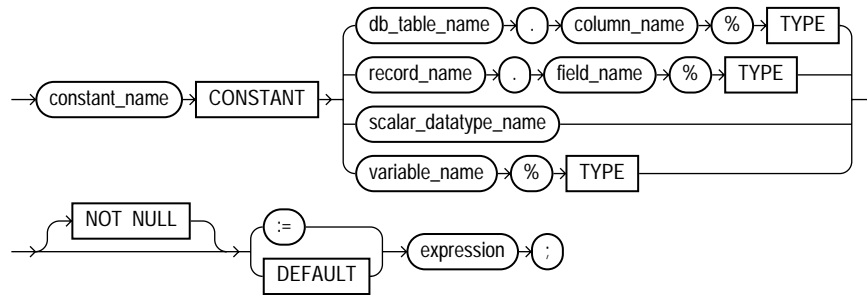
定数と変数

定数と変数は、任意の PL/SQL ブロックまたはサブプログラム、パッケージの宣言部で宣言できます。宣言によって、値の記憶領域を割り当て、データ型を指定し、値を参照できるように記憶位置の名前を決めます。また、初期値を代入したり、NOT NULL 制約を付けたりすることもできます。詳細は、2-28 ページの「[宣言](#)」を参照してください。

構文



constant_declaration



キーワードとパラメータの説明

constant_name

プログラム定数を識別します。命名規則の詳細は、2-4 ページの「[識別子](#)」を参照してください。

CONSTANT

定数の宣言であることを示すキーワードです。定数は宣言部で初期化しなければなりません。初期化された定数の値は変更できません。

record_name.field_name

現在の有効範囲の中で事前に宣言されているユーザー定義のレコードまたは %ROWTYPE 属性のレコードのフィールドを識別します。

scalar_type_name

事前定義済みのスカラー・データ型 (BOOLEAN、NUMBER、VARCHAR2 など) を識別します。詳細は、2-10 ページの「[データ型](#)」を参照してください。

db_table_name.column_name

宣言が PL/SQL コンパイラによって処理されるときにアクセスできないデータベースの表および列を識別します。

variable_name

プログラム変数を識別します。

collection_name

現在の有効範囲のうちこれより前の部分で宣言されている、ネストした表、索引付き表、または varray を指定します。

cursor_name

現在の有効範囲の中で、事前に宣言されている明示カーソルを識別します。

cursor_variable_name

現在の有効範囲の中で事前に宣言されている PL/SQL カーソル変数を識別します。

object_name

現在の有効範囲のうちこれより前の部分で宣言されているオブジェクト（オブジェクト型のインスタンス）を指定します。

record_name

現在の有効範囲の中で事前に宣言されているユーザー定義のレコードを識別します。

db_table_name

宣言が PL/SQL コンパイラによって処理されるときにアクセスできないデータベースの表（またはビュー）を識別します。

%ROWTYPE

この属性は、データベース表の中の行、または事前に宣言されたカーソルから取り出される行を表すレコード型を指定します。レコードの中のフィールドと、それに対応する行の中の列は、同じ名前とデータ型を持ちます。

%TYPE

この属性は、これより前に宣言されたコレクション、カーソル変数、フィールド、オブジェクト、データベース列、または変数のデータ型を指定します。

NOT NULL

変数または定数に NULL を代入できないようにする制約です。実行時に、NOT NULL として定義された変数に NULL を代入しようとすると、事前定義の例外 `VALUE_ERROR` が呼び出されます。NOT NULL 制約の後には初期化句が続かなければなりません。

expression

変数、定数、リテラル、演算子、ファンクション・コールの任意の組合せです。最も単純な式は、1 個の変数で構成されています。宣言が PL/SQL コンパイラによって処理されるとき、`expression` の値が定数または変数に代入されます。その値と定数または変数は、データ型に互換性がなければなりません。

使用方法

定数と変数は、ブロックまたはサブプログラムに入るたびに初期化されます。デフォルトでは、変数は `NULL` に初期化されます。つまり、変数を明示的に初期化しない限り、その値は未定義です。

パッケージの仕様部で宣言された定数と変数は、パブリックであるかプライベートであるかの区別なく、セッションごとに 1 回だけ初期化されます。

`NOT NULL` 変数を宣言する場合、および定数を宣言する場合には、必ず初期化の句が存在していなければなりません。

`%ROWTYPE` 属性を使って定数を宣言できません。`%ROWTYPE` を使って変数を宣言する場合は、初期化できません。

例

変数と定数の宣言の例をいくつか示します。

```
credit_limit CONSTANT NUMBER := 5000;
invalid      BOOLEAN := FALSE;
acct_id      INTEGER(4) NOT NULL DEFAULT 9999;
pi           CONSTANT REAL := 3.14159;
last_name    VARCHAR2(20);
my_ename     emp.ename%TYPE;
```

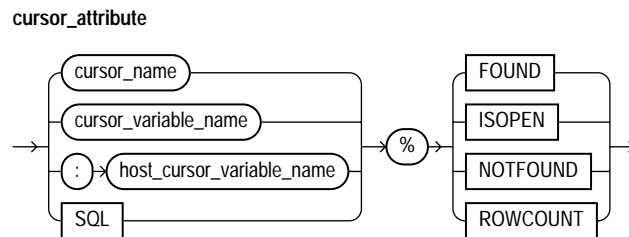
関連項目

代入文、式、`%ROWTYPE` 属性、`%TYPE` 属性

カーソルの属性

カーソルおよびカーソル変数には 4 つの属性があり、それらの属性を使うと、データ操作文の実行に役立つ情報を得ることができます。詳細は、5-31 ページの「[カーソル属性の使用](#)」を参照してください。

構文



キーワードとパラメータの説明

cursor_name

現在の有効範囲の中で、事前に宣言されている明示カーソルを識別します。

cursor_variable_name

現在の有効範囲の中で、事前に宣言されている PL/SQL カーソル変数（またはパラメータ）を識別します。

host_cursor_variable_name

PL/SQL ホスト環境で宣言され、バインド変数として PL/SQL に渡されるカーソル変数を識別します。ホスト・カーソル変数のデータ型は、PL/SQL カーソル変数の戻り型と互換性があります。ホスト変数には、接頭辞としてコロンをつけなければなりません。

SQL

暗黙 SQL カーソルの名前です。詳細は、11-147 ページの「[SQL カーソル](#)」を参照してください。

%FOUND

カーソルまたはカーソル変数の名前に追加できるカーソルの属性です。カーソルがオープンされてから最初の取出しまでの `cursor_name%FOUND` の結果は、`NULL` になります。その後、直前の取出しが行を戻した場合は `TRUE` に、直前の取出しが行を戻さなかった場合は `FALSE` になります。

SQL 文が実行されるまでは、`SQL%FOUND` の結果は `NULL` になります。その後、文がいずれかの行に作用した場合は `TRUE`、作用しなかった場合は `FALSE` になります。

%ISOPEN

カーソルまたはカーソル変数の名前に追加できるカーソルの属性です。カーソルがオープンされていると `cursor_name%ISOPEN` の結果は `TRUE` になり、それ以外の場合は `FALSE` になります。

Oracle は、暗黙 SQL カーソルに対応付けられた SQL 文の実行を終了すると、そのカーソルを自動的にクローズします。このため、`SQL%ISOPEN` の結果は常に `FALSE` になります。

%NOTFOUND

カーソルまたはカーソル変数の名前に追加できるカーソルの属性です。カーソルがオープンされてから最初の取出しまでの `cursor_name%NOTFOUND` の結果は、`NULL` になります。その後、直前の取出しが行を戻した場合は `FALSE` に、直前の取出しが行を戻さなかった場合は `TRUE` になります。

SQL 文が実行されるまでは、`SQL%NOTFOUND` の結果は `NULL` になります。その後、文がいずれかの行に作用した場合は `FALSE`、作用しなかった場合は `TRUE` になります。

%ROWCOUNT

カーソルまたはカーソル変数の名前に追加できるカーソルの属性です。カーソルがオープンされると `%ROWCOUNT` は 0 (ゼロ) になります。最初の取出しまでは、`cursor_name%ROWCOUNT` の結果は 0 (ゼロ) になります。その後は、現時点までに取り出された行数になります。取出しで行が戻されるたびに、数値は増加します。

SQL 文が実行されるまでは、`SQL%ROWCOUNT` の結果は `NULL` になります。その後は、文が作用した行の数になります。文がいずれの行にも作用しなかった場合、`SQL%ROWCOUNT` の結果は 0 になります。

使用方法

カーソルの属性は、すべてのカーソルおよびカーソル変数に適用されます。そのため、たとえば複数のカーソルをオープンし、`%FOUND` または `%NOTFOUND` を使って、まだ取り出していない行が残っているカーソルがどれかを判別できます。同様に、`%ROWCOUNT` を使って、これまでに取り出した行数を知ることができます。

カーソルまたはカーソル変数をオープンしていない場合、%FOUND、%NOTFOUND、あるいは%ROWCOUNT でカーソルやカーソル変数を参照すると、事前定義の例外 INVALID_CURSOR が呼び出されます。

カーソルまたはカーソル変数をオープンすると、対応する問合せを満たす行が識別され、結果セットが形成されます。行は、結果セットから一度に 1 行ずつ取り出されます。

SELECT INTO 文が複数の行を戻した場合、PL/SQL によって事前定義の例外 TOO_MANY_ROWS が呼び出され、%ROWCOUNT は、問合せを満たす行の実数ではなく、1 に設定されます。

最初のフェッチの前は、%NOTFOUND の評価結果は NULL です。したがって、FETCH が正常に実行されない場合には、ループは終了しません。これは、WHEN 条件が真である場合にのみ EXIT WHEN 文が実行されるからです。安全のために、次の EXIT 文をかわりに使用することができます。

```
EXIT WHEN c1%NOTFOUND OR ci%NOTFOUND IS NULL;
```

カーソルの属性は、プロシージャ文では使えますが、SQL 文では使えません。

例

次の PL/SQL ブロックでは、%FOUND を使ってアクションを選択しています。IF 文は、行を挿入するか、無条件にループを終了します。

```
-- available online in file 'examp12'
DECLARE
    CURSOR num1_cur IS SELECT num FROM num1_tab
        ORDER BY sequence;
    CURSOR num2_cur IS SELECT num FROM num2_tab
        ORDER BY sequence;
    num1      num1_tab.num%TYPE;
    num2      num2_tab.num%TYPE;
    pair_num NUMBER := 0;
BEGIN
    OPEN num1_cur;
    OPEN num2_cur;
    LOOP -- loop through the two tables and get pairs of numbers
        FETCH num1_cur INTO num1;
        FETCH num2_cur INTO num2;
        IF (num1_cur%FOUND) AND (num2_cur%FOUND) THEN
            pair_num := pair_num + 1;
            INSERT INTO sum_tab VALUES (pair_num, num1 + num2);
        ELSE
            EXIT;
        END IF;
    END LOOP;
    CLOSE num1_cur;
```

```
    CLOSE num2_cur;  
END;
```

次の例でも同じブロックを使っています。ただし、IF 文の中で %FOUND を使うかわりに、EXIT WHEN 文の中で %NOTFOUND を使っています。

```
-- available online in file 'examp13'  
DECLARE  
    CURSOR num1_cur IS SELECT num FROM num1_tab  
        ORDER BY sequence;  
    CURSOR num2_cur IS SELECT num FROM num2_tab  
        ORDER BY sequence;  
    num1      num1_tab.num%TYPE;  
    num2      num2_tab.num%TYPE;  
    pair_num  NUMBER := 0;  
BEGIN  
    OPEN num1_cur;  
    OPEN num2_cur;  
    LOOP -- loop through the two tables and get  
        -- pairs of numbers  
        FETCH num1_cur INTO num1;  
        FETCH num2_cur INTO num2;  
        EXIT WHEN (num1_cur%NOTFOUND) OR (num2_cur%NOTFOUND);  
        pair_num := pair_num + 1;  
        INSERT INTO sum_tab VALUES (pair_num, num1 + num2);  
    END LOOP;  
    CLOSE num1_cur;  
    CLOSE num2_cur;  
END;
```

次の例では、%ISOPEN を使って判別をしています。

```
IF NOT (emp_cur%ISOPEN) THEN  
    OPEN emp_cur;  
END IF;  
FETCH emp_cur INTO emp_rec;
```

次の PL/SQL ブロックでは、%ROWCOUNT を使って、給与が最も高い 5 人の従業員の名前と給与を取り出しています。

```
-- available online in file 'examp14'  
DECLARE  
    CURSOR c1 is  
    SELECT ename, empno, sal FROM emp  
        ORDER BY sal DESC; -- start with highest-paid employee  
    my_ename CHAR(10);  
    my_empno  NUMBER(4);  
    my_sal    NUMBER(7,2);
```

```
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_ename, my_empno, my_sal;
    EXIT WHEN (c1%ROWCOUNT > 5) OR (c1%NOTFOUND);
    INSERT INTO temp VALUES (my_sal, my_empno, my_ename);
    COMMIT;
  END LOOP;
  CLOSE c1;
END;
```

次の例では、%ROWCOUNT を使って、予期しない多数の行が削除される場合に例外を呼び出しています。

```
DELETE FROM accts WHERE status = 'BAD DEBT';
IF SQL%ROWCOUNT > 10 THEN
  RAISE out_of_bounds;
END IF;
```

関連項目

カーソル、カーソル変数

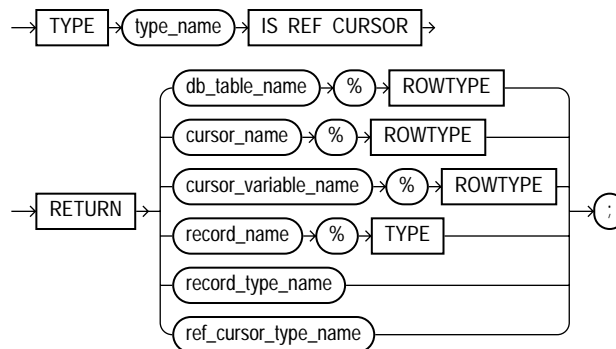
カーソル変数

複数行の問合せを実行するために、Oracle は処理情報を格納する名前の付けられていない作業域をオープンします。その情報にアクセスするには、その作業域の名を示す明示カーソルを使います。または、作業域を指すカーソル変数を使います。カーソルが常に同じ問合せ作業域を参照するのに対し、カーソル変数は異なる作業域を参照できます。カーソル変数を作るには、REF CURSOR 型を定義してから、その型のカーソル変数を宣言します。

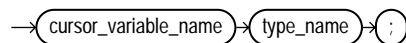
カーソル変数は、C や Pascal のポインタに類似しており、項目そのもののかわりに項目のメモリー位置（アドレス）を保持します。したがって、カーソル変数を宣言すると、項目ではなくポインタが作られます。詳細は、5-14 ページの「[カーソル変数の使用](#)」を参照してください。

構文

ref_cursor_type_definition



cursor_variable_declaration



キーワードとパラメータの説明

type_name

それ以降の PL/SQL カーソル変数の宣言で使うユーザー定義の型指定子です。

REF CURSOR

PL/SQL では、ポインタはデータ型 `REF X` に属します。`REF` は `REFERENCE` の略であり、`X` はオブジェクトのクラスを表します。したがって、カーソル変数はデータ型 `REF CURSOR` に属します。

RETURN

`RETURN` 句の開始を知らせるキーワードです。この句では、カーソル変数が戻す値のデータ型を定義します。`RETURN` 句で `%ROWTYPE` 属性を使うと、データベース表の行や、カーソルまたは強い型定義のカーソル変数によって戻される行を表すレコード型を与えることができます。また、`%TYPE` 属性を使って、事前に宣言されたレコードのデータ型を与えることもできます。

cursor_name

現在の有効範囲の中で、事前に宣言されている明示カーソルを識別します。

cursor_variable_name

現在の有効範囲の中で事前に宣言されている PL/SQL カーソル変数を識別します。

record_name

現在の有効範囲の中で事前に宣言されているユーザー定義のレコードを識別します。

record_type_name

現在の有効範囲の中で事前に宣言されている `RECORD` 型を識別します。

db_table_name

宣言が PL/SQL コンパイラによって処理されるときにアクセスできないデータベースの表（またはビュー）を識別します。

%ROWTYPE

この属性は、データベース表の中の行、カーソルまたは明示されたカーソル変数から取り出される行を表すレコード型を与えます。レコードの中のフィールドと、それに対応する行の中の列は、同じ名前とデータ型を持ちます。

%TYPE

この属性は、事前に宣言されているユーザー定義のレコードのデータ型を与えます。

使用方法

カーソル変数は、すべての PL/SQL クライアントで使えます。たとえば、OCI や Pro*C プログラムなどの PL/SQL ホスト環境の中でカーソル変数を宣言し、それをバインド変数として PL/SQL に渡すことができます。さらに、PL/SQL エンジン具备了 Oracle Forms や Oracle Reports などのアプリケーション開発ツールでは、クライアント側でカーソル変数を完全に使えます。

Oracle データベース・サーバーも PL/SQL エンジンを持っています。したがって、アプリケーションとサーバーの間で、リモート・プロシージャ・コール (RPC) を介してカーソル変数をやりとりできます。クライアント側に PL/SQL エンジンがあれば、クライアントからサーバーへのコールに課される制限はありません。たとえば、クライアント側でカーソル変数を宣言し、それをサーバー側でオープンして取り出した後で、クライアント側で引き続き取り出すことができます。

カーソル変数は、主に、PL/SQL のストアド・サブプログラムと各種クライアントとの間で問合せ結果を渡すために使います。PL/SQL およびクライアントはどちらも結果セットを所有してはならず、単に、結果セットが格納されている作業域を指すポインタを共有しているだけです。たとえば、OCI クライアントおよび Oracle Forms アプリケーション、Oracle Server がすべて同じ作業域を参照する場合があります。

REF CURSOR 型は、強い (制限的な) ものと弱い (制限的でない) ものがあります。強い REF CURSOR 型定義では戻り型を指定しますが、弱い型定義では戻り型を指定しません。強い REF CURSOR 型の方が、エラー発生の可能性は少なくなります。その理由は、PL/SQL コンパイラの場合、強い型定義のカーソル変数は型互換性のある問合せにしか結び付けることができないからです。弱い REF CURSOR 型は、より柔軟です。弱い型定義のカーソル変数は、どの問合せにも結び付けることができます。

REF CURSOR 型を 1 度定義すれば、その型のカーソル変数を宣言できます。%TYPE を使うと、レコード変数のデータ型を与えることができます。また、REF CURSOR 型定義の RETURN 句では、%ROWTYPE を使って、強い型定義 (弱い型定義ではなく) のカーソル変数によって戻される行を表すレコード型を指定できます。

カーソル変数を制御する場合は、OPEN-FOR、FETCH、CLOSE という 3 つの文を使います。まず、OPEN-FOR 文でカーソル変数を複数行問合せ用にオープンします。次に、FETCH 文で結果セットから一度に 1 行ずつ行を取り出します。すべての行が処理されたら、CLOSE 文でカーソル変数をクローズします。

その他の OPEN-FOR 文は、異なる複数の問合せ用に同じカーソル変数をオープンできます。カーソル変数を再オープンする場合、その前にクローズする必要はありません。別の問合せ用にカーソル変数を再オープンすると、前の問合せは失われます。

PL/SQL では、カーソル変数の戻り型が、必ず FETCH 文の INTO 句と互換性を持ちます。カーソル変数と結び付けられた問合せが戻す列の値に対して、INTO 句の中に、対応する型互換性のあるフィールドまたは変数が存在しなければなりません。また、フィールドまたは変数の数は、列の値の数と等しくなければなりません。そうでない場合はエラーになります。

代入に関係する両方のカーソル変数が強い型定義である場合は、両方が同じデータ型でなければなりません。ただし、一方または両方のカーソル変数が弱い型定義である場合は、同じデータ型でなくてもかまいません。

カーソル変数を、そのカーソル変数から取り出す、またはそのカーソル変数をクローズするサブプログラムの仮パラメータとして宣言する場合は、IN または IN OUT モードを指定しなければなりません。サブプログラムがカーソル変数をオープンする場合は、IN OUT モードを指定しなければなりません。

カーソル変数をパラメータとして渡す場合は注意が必要です。実パラメータと仮パラメータの戻り型に互換性がないと、実行時に PL/SQL によって ROWTYPE_MISMATCH が呼び出されます。

カーソル属性 %FOUND、%NOTFOUND、%ISOPEN、%ROWCOUNT をカーソル変数に適用できます。詳細は、5-31 ページの「[カーソル属性の使用](#)」を参照してください。

問合せ作業域を指していないカーソル変数に対して取出しまたはクローズを実行するか、カーソルの属性を適用しようとすると、PL/SQL によって事前定義の例外 INVALID_CURSOR が呼び出されます。カーソル変数（またはパラメータ）が問合せ作業域を指すようにするには、次の 2 通りの方法があります。

- OPEN-FOR 文でカーソル変数を問合せ用にオープンする
- OPEN 文ですでにオープンされたホスト・カーソル変数または PL/SQL カーソル変数の値を、カーソル変数に代入する

問合せ作業域は、それを指すカーソル変数が存在するかぎりアクセスできます。したがって、カーソル変数の値は、1 つの有効範囲から別の有効範囲へ自由に渡すことができます。たとえば、Pro*C プログラムに組み込まれた PL/SQL ブロックにホスト・カーソル変数を渡す場合、カーソル変数が指す作業域は、そのブロックの終了後もアクセス可能な状態のままです。

現在のところ、カーソル変数にはいくつかの制限があります。5-30 ページの「[カーソル変数の制限](#)」を参照してください。

例

OCI や Pro*C プログラムなどの PL/SQL ホスト環境で、カーソル変数を宣言できます。ホスト・カーソル変数を使う場合は、それをバインド変数として PL/SQL に渡さなければなりません。次の Pro*C の例では、ホスト・カーソル変数と選択子を PL/SQL ブロックに渡すことで、選択した問合せ用のカーソル変数をオープンしています。

```
EXEC SQL BEGIN DECLARE SECTION;
...
/* Declare host cursor variable. */
SQL_CURSOR generic_cv;
int choice;
EXEC SQL END DECLARE SECTION;
...
```

```
/* Initialize host cursor variable. */
EXEC SQL ALLOCATE :generic_cv;
...
/* Pass host cursor variable and selector to PL/SQL block. */
EXEC SQL EXECUTE
BEGIN
    IF :choice = 1 THEN
        OPEN :generic_cv FOR SELECT * FROM emp;
    ELSIF :choice = 2 THEN
        OPEN :generic_cv FOR SELECT * FROM dept;
    ELSIF :choice = 3 THEN
        OPEN :generic_cv FOR SELECT * FROM salgrade;
    END IF;
END;
END-EXEC;
```

ホスト・カーソル変数はすべての問合せの戻り型と互換性があります。ホスト・カーソル変数は、弱い型定義の PL/SQL カーソル変数と同じように動作します。

ホスト・カーソル変数を PL/SQL に渡す場合、OPEN-FOR 文をグループ化することでネットワーク通信量を削減できます。たとえば、次の PL/SQL ブロックは、1 回の往復で 3 つのカーソル変数をオープンしています。

```
/* anonymous PL/SQL block in host environment */
BEGIN
    OPEN :emp_cv FOR SELECT * FROM emp;
    OPEN :dept_cv FOR SELECT * FROM dept;
    OPEN :grade_cv FOR SELECT * FROM salgrade;
END;
```

また、カーソル変数を仮パラメータの 1 つとして宣言するストアド・プロシージャをコールしても、カーソル変数を PL/SQL に渡すことができます。データ検索を集中的にするために、次の例のように、型互換性のある問合せをパッケージ・プロシージャの中でグループにまとめることができます。

```
CREATE PACKAGE emp_data AS
    TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
    PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp,
                           choice IN NUMBER);
END emp_data;

CREATE PACKAGE BODY emp_data AS
    PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp,
                           choice IN NUMBER) IS
    BEGIN
        IF choice = 1 THEN
            OPEN emp_cv FOR SELECT * FROM emp WHERE comm IS NOT NULL;
        ELSIF choice = 2 THEN
```

```
        OPEN emp_cv FOR SELECT * FROM emp WHERE sal > 2500;
    ELSIF choice = 3 THEN
        OPEN emp_cv FOR SELECT * FROM emp WHERE deptno = 20;
    END IF;
END open_emp_cv;
END emp_data;
```

あるいは、スタンドアロン・プロシージャを使ってカーソル変数をオープンする方法もあります。単に別々のパッケージの中で REF CURSOR 型を定義し、スタンドアロン・プロシージャの中でその型を参照します。たとえば、次のような（本体部のない）パッケージを作るなら、スタンドアロン・プロシージャを作り、その中でパッケージに定義した型を参照できます。

```
CREATE PACKAGE cv_types AS
    TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
    TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
    TYPE BonusCurTyp IS REF CURSOR RETURN bonus%ROWTYPE;
    ...
END cv_types;
```

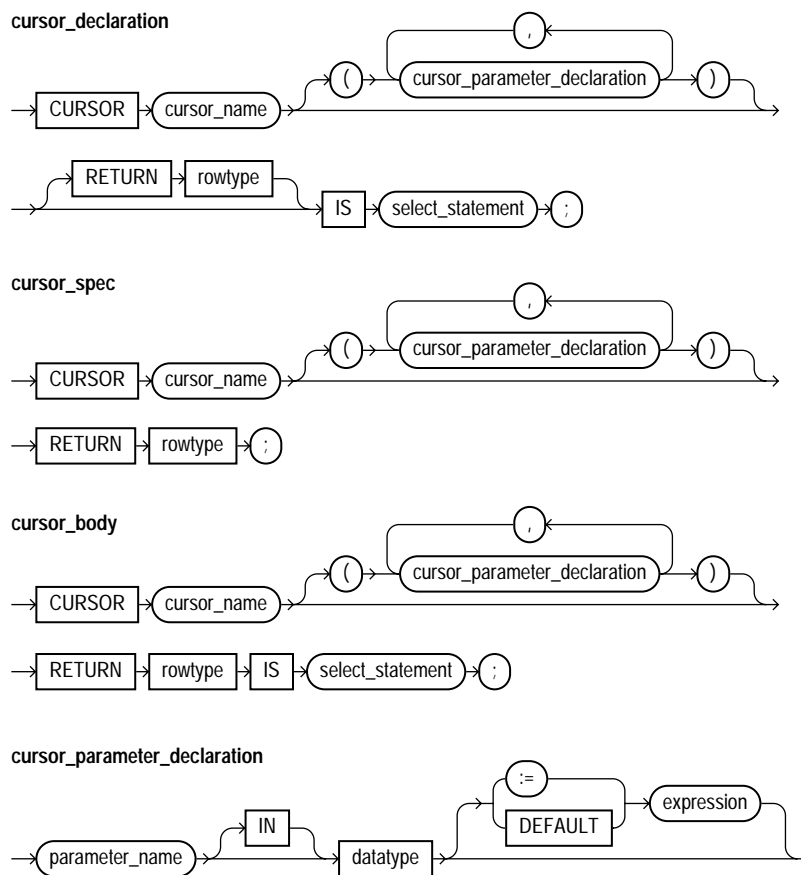
関連項目

CLOSE 文、カーソルの属性、カーソル、FETCH 文、OPEN-FOR 文

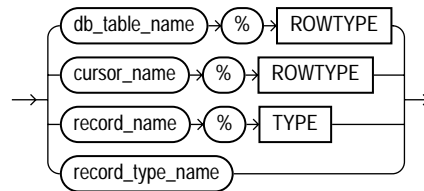
カーソル

複数行の問合せを実行するために、Oracle は処理情報を格納する名前の付けられていない作業域をオープンします。カーソルを使うと、作業域に名前をつけたり、情報にアクセスしたり、行を個別に処理したりできます。詳細は、5-6 ページの「[カーソル管理](#)」を参照してください。

構文



rowtype



キーワードとパラメータの説明

select_statement

行の結果セットを戻す問合せです。構文は `select_into_statement` の構文と似ていますが、`INTO` 句は使えません。11-141 ページの「[SELECT INTO 文](#)」を参照してください。カーソル宣言でパラメータを宣言した場合は、すべてのパラメータを問合せで使わなければなりません。

RETURN

`RETURN` 句の開始を知らせるキーワードです。この句では、カーソルが戻す値のデータ型を定義します。`RETURN` 句で `%ROWTYPE` 属性を使うと、データベースの表の行や、事前に宣言されたカーソルによって戻される行を表すレコード型を与えることができます。また、`%TYPE` 属性を使って、事前に宣言されたレコードのデータ型を与えることもできます。

カーソル本体には、`SELECT` 文と、対応するカーソル仕様部と同じ `RETURN` 句がなければなりません。さらに、`SELECT` 句の中の選択項目の数および順序、データ型は、`RETURN` 句と一致しなければなりません。

parameter_name

カーソルのパラメータ、すなわちカーソルの仮パラメータとして宣言された変数を識別します。カーソルのパラメータは、問合せの中で定数が使える場所ならばどこでも使えます。カーソルの仮パラメータは `IN` パラメータでなければなりません。問合せは、有効範囲の他の PL/SQL 変数を参照することもできます。

db_table_name

宣言が PL/SQL コンパイラによって処理されるときにアクセスできないデータベースの表（またはビュー）を識別します。

cursor_name

現在の有効範囲の中で、事前に宣言されている明示カーソルを識別します。

record_name

現在の有効範囲の中で事前に宣言されているユーザー定義のレコードを識別します。

record_type_name

現在の有効範囲の中で事前に宣言されている RECORD 型を識別します。

%ROWTYPE

この属性は、データベース表の中の行、または事前に宣言されたカーソルから取り出される行を表すレコード型を指定します。レコードの中のフィールドと、それに対応する行の中の列は、同じ名前とデータ型を持ちます。

%TYPE

この属性は、これより前に宣言されたコレクション、カーソル変数、フィールド、オブジェクト、データベース列、または変数のデータ型を指定します。

datatype

これは、型指定子です。datatype の構文は、11-29 ページの「[定数と変数](#)」を参照してください。

expression

変数、定数、リテラル、演算子、ファンクション・コールの任意の組合せです。最も単純な式は、1 個の変数で構成されています。宣言が PL/SQL コンパイラによって処理されるとき、expression の値がパラメータに代入されます。その値とパラメータは、データ型に互換性がなければなりません。

使用方法

OPEN 文、FETCH 文または CLOSE 文でカーソルを参照する前に、そのカーソルを宣言します。また、カーソル宣言で変数を参照する前に、その変数を宣言します。SQL という語は、暗黙カーソルのデフォルト名として PL/SQL によって予約されており、カーソル宣言の中で使えません。

カーソル名に値を代入したり、カーソル名を式の中で使ったりはできません。ただし、カーソルの有効範囲の規則は変数の有効範囲の規則と同じです。詳細は、2-35 ページの「[有効範囲と可視性](#)」を参照してください。

カーソルからデータを取り出す場合は、まずカーソルをオープンし、そこから取り出します。FETCH 文ではターゲットとなる変数を指定するので、cursor_declaration の SELECT 文で INTO 句を使うのは冗長かつ誤りです。

カーソルのパラメータの有効範囲は、カーソルに対してローカルです。つまり、カーソル宣言の中で使われている問合せの内側からしか参照できません。カーソルのパラメータ値は、カーソルがオープンされているときに、カーソルに結び付けられた問合せから使えます。問合せは、有効範囲の他の PL/SQL 変数を参照することもできます。

カーソルのパラメータのデータ型は、無制約で指定しなければなりません。たとえば、次のパラメータ宣言は誤りです。

```
CURSOR c1 (emp_id NUMBER NOT NULL, dept_no NUMBER(2)) -- illegal
```

例

カーソル宣言の例を示します。

```
CURSOR c1 IS SELECT empno, ename, job, sal FROM emp
    WHERE sal > 2000;
CURSOR c2 RETURN dept%ROWTYPE IS
    SELECT * FROM dept WHERE deptno = 10;
CURSOR c3 (start_date DATE) IS
    SELECT empno, sal FROM emp WHERE hiredate > start_date;
```

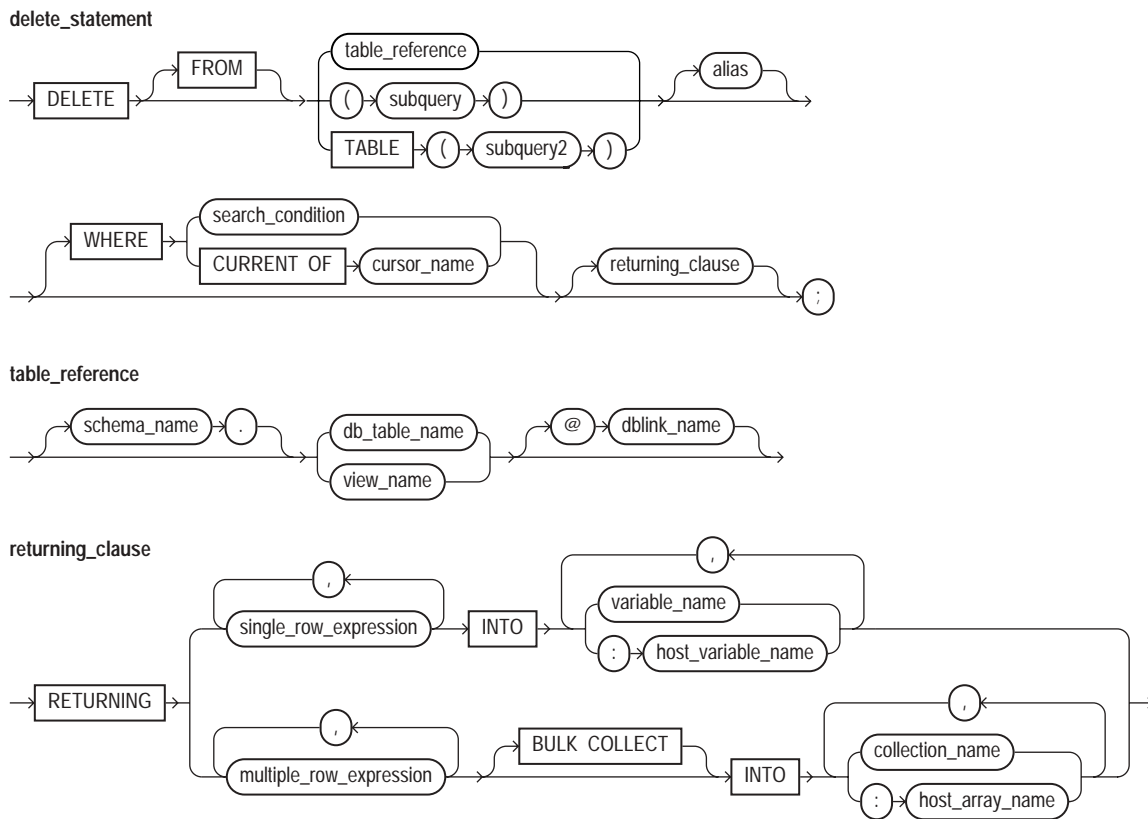
関連項目

CLOSE 文、FETCH 文、OPEN 文、SELECT INTO 文

DELETE 文

DELETE 文は、指定された表またはビューから、行のデータを削除します。DELETE 文の詳細は、『Oracle8i SQL リファレンス』を参照してください。

構文



キーワードとパラメータの説明

table_reference

表またはビューを指定します。指定された表またはビューは、DELETE 文の実行時にアクセスできなければならず、ユーザーが DELETE 権限を持っていなければなりません。

subquery

処理する行の集合を提供する SELECT 文です。構文は `select_into_statement` の構文と似ていますが、INTO 句は使えません。11-141 ページの「[SELECT INTO 文](#)」を参照してください。

TABLE (subquery2)

TABLE のオペランドは、1 つの列値を戻す SELECT 文です。これはネストした表、またはネストした表として割り当てられる varray である必要があります。演算子 TABLE は、値がスカラー値ではなくコレクションであることを Oracle に通知します。

alias

参照される表またはビューの別名（通常は短縮名）で、WHERE 句の中でよく使用されます。

WHERE search_condition

この句は、参照された表またはビューから削除する行を条件に従って選択します。検索条件を満たす行だけが削除されます。WHERE 句を省略すると、表またはビューのすべての行が削除されます。

WHERE CURRENT OF cursor_name

この句は、`cursor_name` で識別されるカーソルに結び付けられている FETCH 文によって処理された最後の行を参照します。カーソルは、FOR UPDATE であること、さらにオープンされていて行に置かれていることが必要です。カーソルがオープンされていないと、CURRENT OF 句でエラーが発生します。

カーソルがオープンされていても、取り出された行がないか、最後の取り出しで行が戻されなかった場合は、PL/SQL により事前定義の例外 `NO_DATA_FOUND` が呼び出されます。

returning_clause

この句を使うと、削除された行から値を戻せるので、あらかじめ行を SELECT で選択しておく必要がありません。取得した列値を、変数かホスト変数（またはその両方）あるいはコレクションかホスト変数（またはその両方）に代入できます。ただし、RETURNING 句はリモート、またはパラレルでの削除には使用できません。

BULK COLLECT

この句は、コレクションを PL/SQL エンジンに戻す前にバルク・バインド出力するよう、SQL エンジンに指示を与えます。SQL エンジンは、RETURNING INTO リスト内で参照されるすべてのコレクションをバルク・バインドします。対応する列には、スカラー値（複合ではない）が格納されている必要があります。詳細は、4-27 ページの「[バルク・バインドの利用](#)」を参照してください。

使用方法

DELETE WHERE CURRENT OF 文は、オープンされているカーソルからの取出し（カーソル FOR ループで実行される暗黙の取出しを含む）の後で使えます（ただしそのためには、対応付けられた問合せが FOR UPDATE でなければなりません）。この文は現在の行、すなわち直前に取り出された行を削除します。

暗黙的な SQL のカーソルとカーソルの属性 %NOTFOUND、%FOUND および %ROWCOUNT を使うと、DELETE 文の実行結果に関する有用な情報にアクセスできます。

DELETE 文では、1 つまたは複数の行が削除されるか、行の削除が実行されないかのどちらかです。1 つまたは複数の行が削除された場合は、属性の値は次のようになります。

- SQL%NOTFOUND は、FALSE になる。
- SQL%FOUND は、TRUE になる。
- SQL%ROWCOUNT は、削除された行数になる。

行の削除が実行されなかった場合は、属性の値は次のようになります。

- SQL%NOTFOUND は、TRUE になる。
- SQL%FOUND は、FALSE になる。
- SQL%ROWCOUNT は 0 になる。

例

次の文は、売上が目標を下回った従業員全員を表 bonus から削除します。

```
DELETE FROM bonus WHERE sales_amt < quota;
```

次の文は、削除された行から列 sal を戻し、その列値をホスト配列の要素に格納します。

```
DELETE FROM emp WHERE job = 'CLERK' AND sal > 3000
RETURNING sal INTO :clerk_sals;
```

BULK COLLECT 句を FORALL 文と組み合わせることができます。この場合、SQL エンジンでは列値を段階的にバルク・バインドします。次の例では、コレクション `depts` に 3 つの要素があり、それぞれによって 5 行ずつ削除される場合、コレクション `enums` は、文が完了すると 15 の要素を持ちます。

```
FORALL j IN depts.FIRST..depts.LAST
  DELETE FROM emp WHERE empno = depts(j)
  RETURNING empno BULK COLLECT INTO enums;
```

各実行によって戻された列の値は、前に戻された値に追加されます。

関連項目

FETCH 文、SELECT 文

EXCEPTION_INIT プラグマ

EXCEPTION_INIT プラグマは、例外名を Oracle のエラー番号に対応付けます。この対応付けにより、OTHERS ハンドラを使うかわりに、内部例外を名前で参照し、専用のハンドラを作成できます。詳細は、6-8 ページの「[EXCEPTION_INIT の使用](#)」を参照してください。

構文

exception_init_pragma

→ PRAGMA EXCEPTION_INIT ((exception_name , error_number) ;

キーワードとパラメータの説明

PRAGMA

文がプラグマ (コンパイラ・ディレクティブ) であることを表します。プラグマは、実行時ではなくコンパイル時に処理されます。プログラムの機能に影響を与えず、コンパイラに情報を提供する役割しかありません。

exception_name

現在の有効範囲の中で事前に宣言されているユーザー定義の例外を識別します。

error_number

任意の有効な ORACLE のエラー番号です。これはファンクション SQLCODE が戻すエラー番号と同じです。

使用方法

EXCEPTION_INIT は、任意の PL/SQL ブロック、サブプログラム、パッケージの宣言部で使えます。このプラグマは、対応付けられた例外と同じ宣言部の中で、例外宣言の後のどこかに指定しなければなりません。

1 つのエラー番号に割り当てる例外名は 1 つだけです。

例

次のプラグマは、例外 `deadlock_detected` を Oracle エラー 60 に対応付けています。

```
DECLARE
    deadlock_detected EXCEPTION;
    PRAGMA EXCEPTION_INIT (deadlock_detected, -60);
BEGIN
    ...
EXCEPTION
    WHEN deadlock_detected THEN
        -- handle the error
    ...
END;
```

関連項目

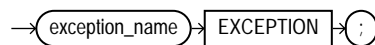
例外、SQLCODE ファンクション

例外

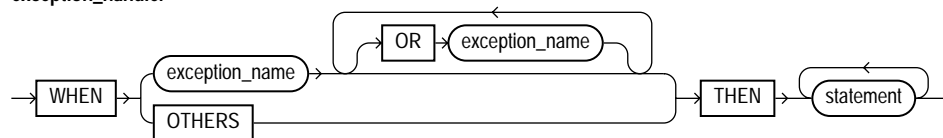
例外は、事前定義またはユーザー定義可能な、実行時エラーまたは警告状態です。事前定義の例外は実行時システムによって暗黙的（自動的）に呼び出されます。ユーザー定義の例外は `RAISE` 文によって明示的に呼び出さなければなりません。呼び出された例外を処理するには、「例外ハンドラ」と呼ばれる独立したルーチンを作ります。詳細は、[第6章の「エラー処理」](#)を参照してください。

構文

`exception_declaration`



`exception_handler`



キーワードとパラメータの説明

WHEN

例外ハンドラを結び付けるキーワードです。キーワード `WHEN` に続けて、キーワード `OR` で区切った例外のリストを指定すると、複数の例外で同一の一連の文を実行できます。リスト中のいずれかの例外が呼び出されると、それに結び付けられた文が実行されます。

exception_name

`ZERO_DIVIDE` のような事前定義の例外、または現在の有効範囲の中で事前に宣言されているユーザー定義の例外を識別します。

OTHERS

このキーワードは、ブロックの例外処理部で明示的に名前を指定していないすべての例外を表します。`OTHERS` の使用はオプションで、ブロックの最後の例外ハンドラとしてしか使えません。キーワード `WHEN` に続く例外のリストの中で、`OTHERS` を使えません。

statement

これは、実行可能文です。statement の構文は、11-7 ページの「[ブロック](#)」を参照してください。

使用方法

例外宣言はブロック、サブプログラム、またはパッケージの宣言部でしか使えません。例外の有効範囲の規則は変数と同じです。しかし、変数とは異なり、例外をパラメータとしてサブプログラムに渡すことができません。

例外のいくつかは PL/SQL によって事前に定義されています。これらの例外のリストは、6-4 ページの「[事前定義の例外](#)」を参照してください。PL/SQL は、事前定義の例外をパッケージ STANDARD でグローバルに宣言しているので、ユーザーが宣言する必要はありません。

事前定義の例外を再宣言すると、ローカルな宣言がグローバルな宣言を上書きするために、エラーが発生しやすくなります。この場合、事前定義の例外を指定するには、次のようにドット表記法を使わなければなりません。

```
EXCEPTION
  WHEN invalid_number OR STANDARD.INVALID_NUMBER THEN ...
```

PL/SQL ブロックの例外処理部はオプションです。例外ハンドラはブロックの末尾になければなりません。例外処理部はキーワード EXCEPTION で始まります。ブロックの例外処理部の終わりは、ブロックの終わりも示すキーワード END です。

例外を呼び出すのは、処理の続行が不可能、あるいは望ましくないようなエラーが発生した場合だけにしてください。呼び出された例外に対応する例外ハンドラが現ブロックに存在しない場合、例外は次の規則に従って遷移します。

- 現ブロックの外のブロックがある場合、例外はそのブロックに渡される。その後、その外のブロックが現ブロックになる。呼び出された例外に対応するハンドラが見つからない場合は、この過程が繰り返される。
- 現ブロックに外のブロックがない場合、「未処理例外 (unhandled exception)」エラーがホスト環境に戻される。

ただし、例外はリモート・プロシージャ・コール (RPC) には遷移しません。そのため、PL/SQL ブロックは、リモート・サブプログラムによって呼び出された例外を処理できません。対策の詳細は、6-9 ページの「[raise_application_error の使用](#)」を参照してください。

ブロックの例外処理部でアクティブになれる例外は一度に 1 つだけです。このため、ハンドラの内側で例外状況が発生すると、現ブロックの外側のブロックが、新しく呼び出された例外に対するハンドラを検索するための最初のブロックになります。それ以降の例外の遷移は通常どおりに起こります。

例外ハンドラから参照できる変数は、現ブロックから参照できる変数だけです。

例

次の PL/SQL ブロックには 2 つの例外ハンドラがあります。

```
DECLARE
    bad_emp_id  EXCEPTION;
    bad_acct_no EXCEPTION;
    ...
BEGIN
    ...
EXCEPTION
    WHEN bad_emp_id OR bad_acct_no THEN -- user-defined
        ROLLBACK;
    WHEN ZERO_DIVIDE THEN -- predefined
        INSERT INTO inventory VALUES (part_number, quantity);
        COMMIT;
END;
```

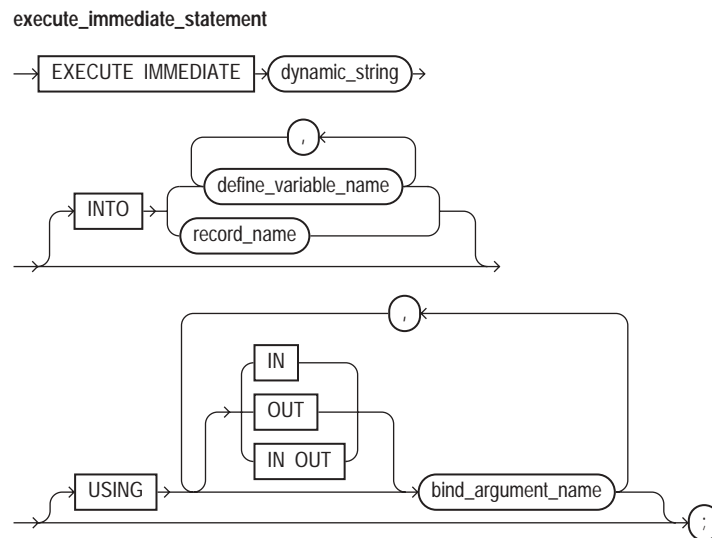
関連項目

ブロック、EXCEPTION_INIT プラグマ、RAISE 文

EXECUTE IMMEDIATE 文

EXECUTE IMMEDIATE 文は、動的 SQL 文または無名 PL/SQL ブロックを準備（解析）し、即時に実行します。詳細は、[第 10 章の「システム固有の動的 SQL」](#)を参照してください。

構文



キーワードとパラメータの説明

dynamic_string

SQL 文または PL/SQL ブロックを表す文字列リテラル、変数、または式です。

define_variable_name

SELECT によって選択された列値を格納する変数を識別します。

record_name

SELECT によって選択された行を格納する、ユーザー定義レコードまたは %ROWTYPE レコードを識別します。

bind_argument_name

動的 SQL 文または PL/SQL ブロックに渡される値を持つ式を識別します。

INTO ...

1 行の問合せの場合に便利なこの句は、取り出された列値を入れる変数またはレコードを指定します。

USING ...

バインド引数のリストを指定します。パラメータ・モードは、指定しないとデフォルトで IN に設定されます。USING 句内のすべてのバインド引数は、実行時に SQL 文内または PL/SQL ブロック内の対応するプレースホルダを置き換えます。

使用方法

複数行の問合せの場合を除いて、動的文字列には任意の SQL 文（終了記号なし）または任意の PL/SQL ブロック（終了記号付き）を含むことができます。また、バインド引数のプレースホルダも含むことができます。しかし、バインド引数を使用してスキーマ・オブジェクトの名前を動的 SQL 文に渡すことはできません。正しい用法は、10-10 ページの「[スキーマ・オブジェクトの名前を渡す](#)」を参照してください。

問合せが戻す列の値それぞれに対して、INTO 句の中に、対応する型互換性のあるフィールドまたは変数が存在しなければなりません。また、動的文のプレースホルダは、USING 句のバインド引数に対応付けられていなければなりません。

USING 句内では数値リテラル、文字リテラル、および文字列リテラルは使用できますが、ブール・リテラル（TRUE、FALSE、NULL）は使用できません。動的文に NULL を渡すには、代替方法を使う必要があります（10-12 ページの「[NULL を渡す](#)」を参照してください）。

動的 SQL はすべての SQL データ型をサポートしています。たとえば、定義変数やバインド引数をコレクション、LOB、オブジェクト型のインスタンス、および ref とすることができます。通常、動的 SQL は PL/SQL 固有の型をサポートしていません。たとえば定義変数やバインド引数をプールまたは索引付き表にできません。例外として、PL/SQL レコードを INTO 句に入れることができます。

動的 SQL 文は、バインド引数の新しい値を使用して繰り返し実行できます。ただし、EXECUTE IMMEDIATE は実行のたびに動的文を準備しなおすため、オーバーヘッドが発生します。

例

次の PL/SQL ブロックには動的文の例がいくつか含まれています。

```
DECLARE
    sql_stmt    VARCHAR2(100);
    plsql_block VARCHAR2(200);
    my_deptno   NUMBER(2) := 50;
    my_dname    VARCHAR2(15) := 'PERSONNEL';
    my_loc      VARCHAR2(15) := 'DALLAS';
    emp_rec     emp%ROWTYPE;
BEGIN
    sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
    EXECUTE IMMEDIATE sql_stmt USING my_deptno, my_dname, my_loc;

    sql_stmt := 'SELECT * FROM emp WHERE empno = :id';
    EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING 7788;

    EXECUTE IMMEDIATE 'DELETE FROM dept
        WHERE deptno = :n' USING my_deptno;

    plsql_block := 'BEGIN emp_stuff.raise_salary(:id, :amt); END;';
    EXECUTE IMMEDIATE plsql_block USING 7788, 500;

    EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt NUMBER)';

    sql_stmt := 'ALTER SESSION SET SQL_TRACE TRUE';
    EXECUTE IMMEDIATE sql_stmt;
END;
```

関連項目

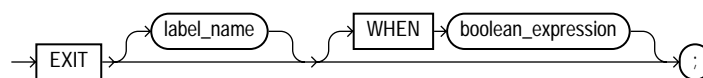
OPEN-FOR-USING 文

EXIT 文

EXIT 文は、ループを終了するために使います。EXIT 文には、無条件の EXIT と条件付きの EXIT WHEN という 2 つの形式があります。どちらの形式でも、終了するループの名前を指定できます。詳細は、3-6 ページの「[反復制御: LOOP 文と EXIT 文](#)」を参照してください。

構文

exit_statement



キーワードとパラメータの説明

EXIT

無条件の（つまり WHEN 句のない）EXIT 文は、現行のループをただちに終了します。実行はループの直後の文から再開されます。

label_name

終了するループを識別します。現在のループだけでなく、ラベルが付けられている外側のループも終了できます。

boolean_expression

結果が TRUE、FALSE、NULL のいずれかのブール値になる式です。この式は、EXIT WHEN 文を使ったループが反復されるたびに評価されます。式の結果が TRUE の場合、現行のループ（または label_name のラベルの付いたループ）はただちに終了します。boolean_expression の構文は、11-62 ページの「[式](#)」を参照してください。

使用方法

EXIT 文は、ループの内側でしか使えません。PL/SQL では無限ループをコーディングできません。たとえば、次のループは、通常の方法では決して終了しません。

```
WHILE TRUE LOOP ... END LOOP;
```

このループを終了させるには、EXIT 文を使います。

EXIT 文を使ってカーソル FOR ループを途中で終了させると、カーソルは自動的にクローズされます。ループの内側で例外状況が発生した場合も、カーソルは自動的にクローズされず。

例

EXIT 文ではブロックを直接終了できないので、次の例は誤りです。この文で終了できるのはループだけです。

```
DECLARE
  amount  NUMBER;
  maximum NUMBER;
BEGIN
  ...
  BEGIN
    ...
    IF amount >= maximum THEN
      EXIT; -- illegal
    END IF;
  END;
END;
```

次のループは本来なら 10 回実行されますが、フェッチする行が 10 行未満の場合は途中で終了します。

```
FOR i IN 1..10
  FETCH c1 INTO emp_rec;
  EXIT WHEN c1%NOTFOUND;
  total_comm := total_comm + emp_rec.comm;
END LOOP;
```

次の例は、ループ・ラベルの使用方法を示しています。

```
<<outer>>
FOR i IN 1..10 LOOP
  ...
  <<inner>>
  FOR j IN 1..100 LOOP
    ...
    EXIT outer WHEN ... -- exits both loops
  END LOOP inner;
END LOOP outer;
```

関連項目

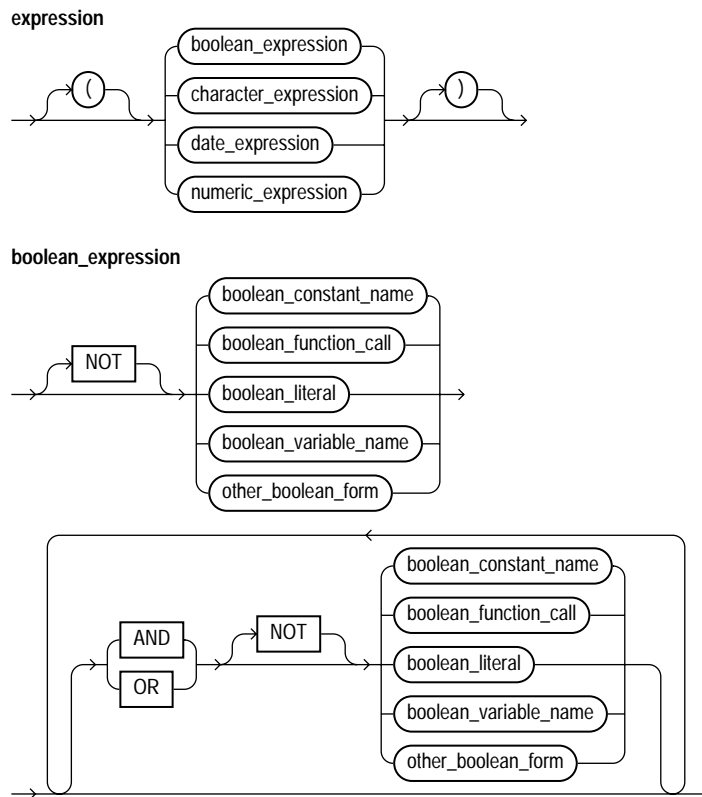
式、LOOP 文

式

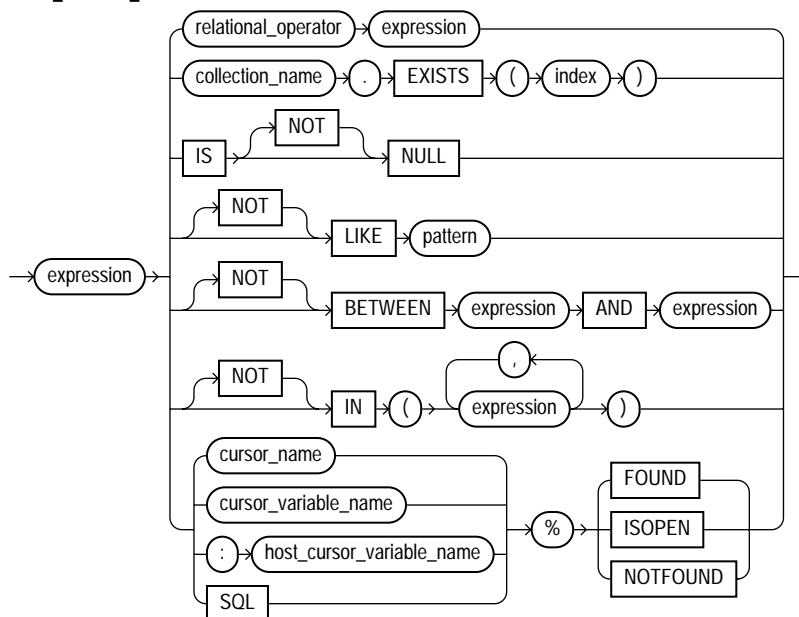
式は、変数、定数、リテラル、演算子、ファンクション・コールの任意の組合せです。最も単純な式は、1 個の変数で構成されています。

PL/SQL コンパイラは、式を構成する変数、定数、リテラル、および演算子の型から、式のデータ型を決定します。式が評価されたときは、その型の 1 つの値が結果として得られます。詳細は、2-39 ページの「[式と比較](#)」を参照してください。

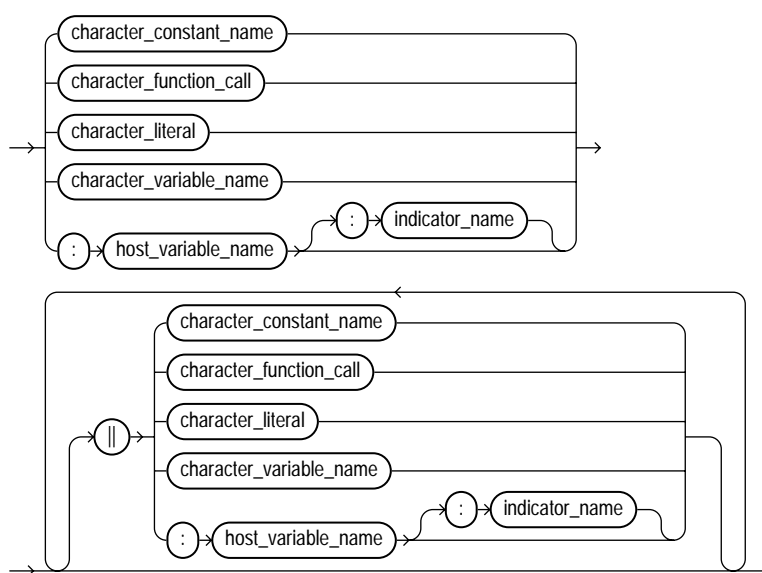
構文



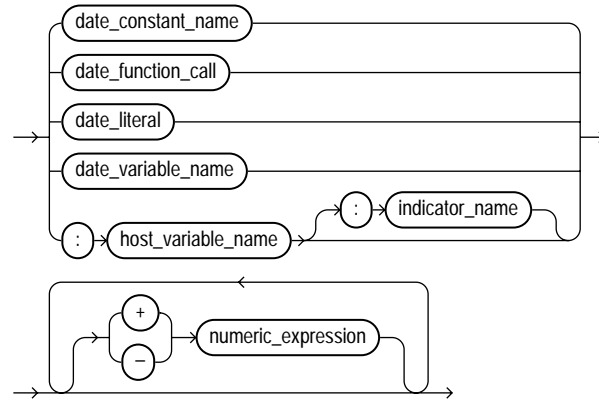
other_boolean_form



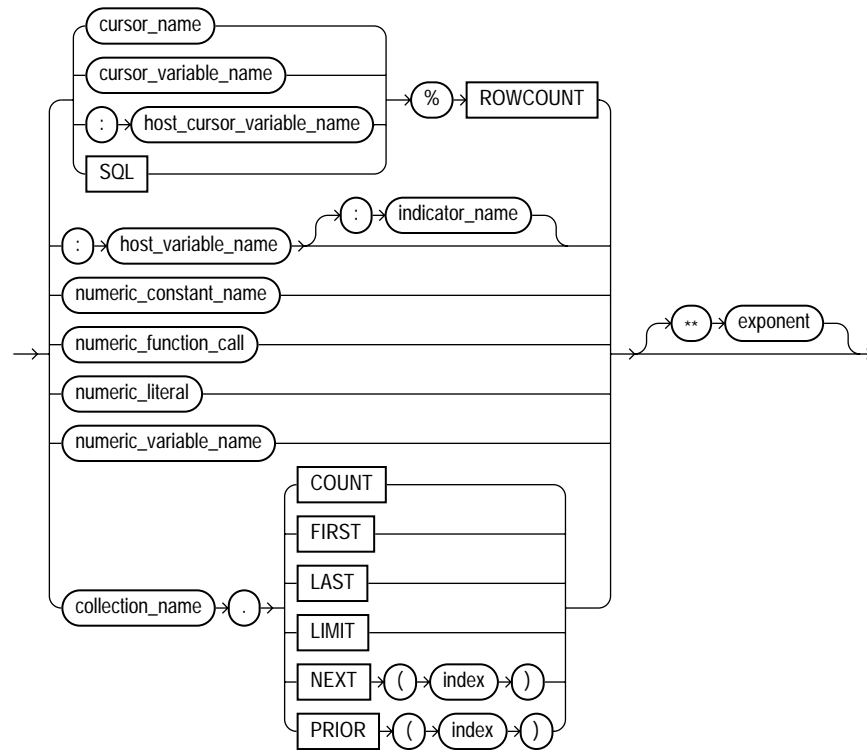
character_expression



date_expression



numeric_expression



numeric_expression

結果が、整数または実数になる式です。

NOT、AND、OR

これらは論理演算子であり、2-41 ページの表 2-3 の 3 値論理に従います。AND は両方のオペランドが真の場合に値 TRUE を戻します。OR はオペランドのいずれかが真の場合に値 TRUE を戻します。NOT はオペランドの反対の値（論理否定）を戻します。NULL は値を持たないため、NOT NULL は NULL を戻します。詳細は、2-41 ページの「[論理演算子](#)」を参照してください。

boolean_constant_name

TRUE、FALSE、または NULL に初期化されなければならない、BOOLEAN 型の定数を識別します。ブール定数に算術演算を実行できません。

boolean_function_call

ブール値を戻すファンクション・コールです。

boolean_literal

事前定義の値 TRUE、FALSE、または NULL（存在しない値、未知の値または適用不可能な値を表す）です。データベース列に値 TRUE や FALSE を挿入できません。

boolean_variable_name

BOOLEAN 型の変数を識別します。BOOLEAN 変数に代入できるのは、値 TRUE と FALSE、および NULL だけです。列の値を選択またはフェッチして BOOLEAN 変数に入れることはできません。また、BOOLEAN 変数に対しては算術演算を実行できません。

relational_operator

この演算子を使うと、式を比較できます。各演算子の意味は、2-42 ページの「[比較演算子](#)」を参照してください。

IS [NOT] NULL

この比較演算子は、オペランドが NULL の場合にブール値 TRUE を返し、オペランドが NULL でなければ FALSE を戻します。

[NOT] LIKE

この比較演算子は、文字値とパターンを比較します。大文字と小文字は区別されます。LIKE は、文字のパターンが一致すればブール値 TRUE を、一致しなければ FALSE を戻します。

pattern

LIKE 演算子によって、指定された文字列値と比較される文字列です。pattern には、ワイルドカードと呼ばれる特殊目的の文字を 2 種類使えます。アンダースコア (_) は 1 つの文字を表します。パーセント記号 (%) はゼロ個以上の文字を表します。

[NOT] BETWEEN

この比較演算子は、ある値が指定範囲の中にあるかどうかを調べます。つまり、下限以上、上限以下という意味を持ちます。

[NOT] IN

この比較演算子は、集合のメンバーであるかどうかを調べます。つまり、集合の「いずれかのメンバーと等しい」という意味を持ちます。集合には NULL が含まれていてもかまいませんが、NULL は無視されます。さらに、次の形式の式は、

```
value NOT IN set
```

集合に NULL が含まれていると FALSE になります。

cursor_name

現在の有効範囲の中で、事前に宣言されている明示カーソルを識別します。

cursor_variable_name

現在の有効範囲の中で事前に宣言されている PL/SQL カーソル変数を識別します。

host_cursor_variable_name

PL/SQL ホスト環境で宣言され、バインド変数として PL/SQL に渡されるカーソル変数を識別します。ホスト・カーソル変数には、接頭辞としてコロンをつけなければなりません。

SQL

SQL のデータ操作文を処理するために、Oracle によって暗黙的にオープンされるカーソルを識別します。暗黙 SQL カーソルは常に、直前に実行された SQL 文を参照します。

%FOUND、%ISOPEN、%NOTFOUND、%ROWCOUNT

カーソルの属性です。カーソル名またはカーソル変数名にこれらの属性を追加すると、複数行の問合せの実行に関する有用な情報が戻されます。これらの属性は暗黙 SQL カーソルにも追加できます。詳細は、5-31 ページの「[カーソル属性の使用](#)」を参照してください。

EXISTS、COUNT、FIRST、LAST、LIMIT、NEXT、PRIOR

コレクション・メソッドです。コレクションの名前にこれらを付加すると、有用な情報が戻されます。たとえば、EXISTS(*n*) は、コレクションに *n* 番目の要素が存在する場合に TRUE を戻します。ない場合、EXISTS(*n*) は FALSE を戻します。詳細は、11-16 ページの「[コレクション・メソッド](#)」を参照してください。

index

結果が BINARY_INTEGER 型の値、またはその型に暗黙的に変換可能な値になる数値式です。

host_variable_name

PL/SQL ホスト環境で宣言され、バインド変数として PL/SQL に渡される変数を識別します。ホスト変数のデータ型は、適切な PL/SQL のデータ型に暗黙的に変換できなければなりません。また、ホスト変数には、接頭辞としてコロンを付けなければなりません。

indicator_name

PL/SQL ホスト環境で宣言され、PL/SQL に渡される標識変数を識別します。標識変数には、接頭辞としてコロンを付けなければなりません。標識変数は、対応付けられたホスト変数の値または条件を示します。たとえば、Oracle プリコンパイラ環境では、標識変数を使って出力ホスト変数内の NULL や切り捨てられた値を検出できます。

numeric_constant_name

数値を格納する、事前に宣言された定数を識別します。この定数は、数値または暗黙的に数値に変換可能な値に初期化されなければなりません。

numeric_function_call

数値または暗黙的に数値に変換可能な値を戻すファンクション・コールです。

numeric_literal

数値または暗黙的に数値に変換可能な値を表すリテラルです。

collection_name

現在の有効範囲のうちこれより前の部分で宣言されている、ネストした表、索引付き表、または varray を指定します。

numeric_variable_name

数値を格納する、事前に宣言された変数を識別します。

NULL

このキーワードは NULL を表し、欠落している値、不明な値、または適用できない値を示します。数値式または日付式の中で NULL を使用すると、結果は NULL になります。

exponent

結果が数値式になる式です。

+, -, /, *, **

これらの記号はそれぞれ、加算、減算、除算、乗算、指数の演算子です。

character_constant_name

文字値を格納する、事前に宣言された定数を識別します。この定数は、文字値または暗黙的に文字値に変換可能な値に初期化されなければなりません。

character_function_call

文字値または暗黙的に文字値に変換可能な値を戻すファンクション・コールです。

character_literal

文字値または暗黙的に文字値に変換可能な値を表すリテラルです。

character_variable_name

文字値を格納する、事前に宣言された変数を識別します。

||

連結演算子です。次の例に示すように、*string1* と *string2* を連結した結果は、*string1* の後に *string2* が続く文字列になります。

```
'Good' || ' morning!' = 'Good morning!'
```

次の例に示すように、NULL は連結の結果に影響しません。

```
'suit' || NULL || 'case' = 'suitcase'
```

長さがゼロの文字列 ('') は NULL 文字列と呼ばれ、NULL と同じように扱われます。

date_constant_name

日付値を格納する、事前に宣言された定数を識別します。この定数は、日付値、または暗黙的に日付値に変換可能な値に初期化されなければなりません。

date_function_call

日付値、または暗黙的に日付値に変換可能な値を戻すファンクション・コールです。

date_literal

日付値、または暗黙的に日付値に変換可能な値を表すリテラルです。

date_variable_name

日付値を格納する、事前に宣言された変数を識別します。

使用方法

ブール式では、互換性のあるデータ型を持つ値しか比較できません。詳細は、2-25 ページの「[データ型の変換](#)」を参照してください。

条件制御文においてブール式が TRUE になると、関連する一連の文が実行されます。しかし、式が FALSE または NULL になると、対応する一連の文は実行されません。

関係演算子は、BOOLEAN 型のオペランドに適用できます。定義によれば、TRUE は FALSE よりも大きい値を持ちます。NULL の関係する比較は、常に結果も NULL になります。ブール式の値はブール変数にしか代入できず、ホスト変数やデータベースの列には代入できません。また、BOOLEAN 型からの、または BOOLEAN 型へのデータ型変換はできません。

次の例に示すように、加算演算子または減算演算子を使うと、日付値を増減できます。

```
hire_date := '10-MAY-95';  
hire_date := hire_date + 1;  -- makes hire_date '11-MAY-95'  
hire_date := hire_date - 5;  -- makes hire_date '06-MAY-95'
```

PL/SQL がブール式を評価する場合は、優先順位が最も高いのが NOT 演算子で、次が AND 演算子、最後が OR 演算子です。ただし、カッコを使うと、演算子のデフォルトの優先順位を変更できます。

式の中では、事前定義の優先順位に従って演算が実行されます。デフォルトの演算順序を、優先順位の高いものから順に示すと、次のようになります。

- カッコ
- 指数
- 単項演算子
- 乗算 および除算
- 加算、減算、および連結

PL/SQL では、優先順位の等しい演算子を評価する順序は特に決まっていません。ある式の一部にカッコで囲まれた別の式が含まれている場合、PL/SQL では、カッコで囲まれた式を先に評価し、その結果の値を外側の式で使います。カッコで囲まれた式がネストされている場合、PL/SQL では、最も内側にある式を 1 番目に評価し、最も外側にある式を最後に評価します。

例

式の例を次に示します。

<code>(a + b) > c</code>	-- Boolean expression
<code>NOT finished</code>	-- Boolean expression
<code>TO_CHAR(acct_no)</code>	-- character expression
<code>'Fat ' 'cats'</code>	-- character expression
<code>'15-NOV-95'</code>	-- date expression
<code>MONTHS_BETWEEN(d1, d2)</code>	-- date expression
<code>pi * r**2</code>	-- numeric expression
<code>emp_cv%ROWCOUNT</code>	-- numeric expression

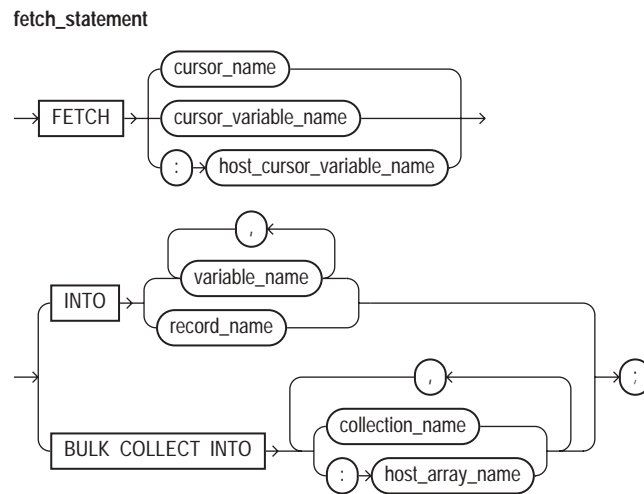
関連項目

代入文、定数と変数、EXIT 文、IF 文、LOOP 文

FETCH 文

FETCH 文は、複数行の問合せの結果セットから、一度に 1 行ずつ行を取り出します。データは問合せが選択した列に対応する変数またはフィールドに格納されます。詳細は、5-6 ページの「[カーソル管理](#)」を参照してください。

構文



キーワードとパラメータの説明

cursor_name

現在の有効範囲の中で、事前に宣言されている明示カーソルを識別します。

cursor_variable_name

現在の有効範囲の中で、事前に宣言されている PL/SQL カーソル変数（またはパラメータ）を識別します。

host_cursor_variable_name

PL/SQL ホスト環境で宣言され、バインド変数として PL/SQL に渡されるカーソル変数を識別します。ホスト・カーソル変数のデータ型は、PL/SQL カーソル変数の戻り型と互換性があります。ホスト変数には、接頭辞としてコロンをつけなければなりません。

BULK COLLECT

この句は、コレクションを PL/SQL エンジンに戻す前にバルク・バインド出力するよう、SQL エンジンに指示を与えます。SQL エンジンは、`INTO` リスト内で参照されるすべてのコレクションをバルク・バインドします。対応する列には、スカラー値（複合ではない）が格納されている必要があります。詳細は、4-27 ページの「[バルク・バインドの利用](#)」を参照してください。

variable_name

フェッチした列値を格納するための、事前に宣言されたスカラー変数を識別します。カーソルまたはカーソル変数と結び付けられた問合せが戻す列の値に対して、リストの中に、対応する型互換の変数が存在しなければなりません。

record_name

フェッチした行の値を格納する、ユーザー定義のレコードまたは `%ROWTYPE` のレコードを識別します。カーソルまたはカーソル変数と結び付けられた問合せが戻す列の値に対して、レコードの中に、対応する型互換のフィールドが存在しなければなりません。

collection_name

バルク・フェッチした列値を格納するための、宣言されたコレクションを識別します。問合せ `select_item` ごとに、リストの中に、対応する型互換のコレクションが存在しなければなりません。

host_array_name

バルク・フェッチした列値を格納するための配列を識別します。この配列は、PL/SQL ホスト環境で宣言され、バインド変数として PL/SQL に渡されます。問合せ `select_item` ごとに、リストの中に、対応する型互換の配列が存在しなければなりません。ホスト配列には、接頭辞としてコロンをつけなければなりません。

使用方法

複数行の問合せを処理するには、カーソル `FOR` ループか `FETCH` 文を使います。

問合せの `WHERE` 句に含まれる変数は、カーソルまたはカーソル変数がオープンされたときにだけ評価されます。結果セットや問合せの中の変数の値を変更するには、カーソルまたはカーソル変数を、新しい値に設定して再オープンしなければなりません。

カーソルをオープンし直す場合は、まずクローズしなければなりません。ただし、カーソル変数を再オープンする場合には、その前にクローズする必要はありません。

同じカーソルまたはカーソル変数を使用した別々のフェッチで、異なる `INTO` リストを使用できます。個々の `FETCH` 文で別の行を取り出し、目標変数に値を代入します。

結果セットの中に行が残っていない状態で FETCH 文を実行すると、ターゲット・フィールドの値またはターゲット変数の値は予測不能となり、%NOTFOUND 属性の結果は TRUE となります。

PL/SQL では、カーソル変数の戻り型が、必ず FETCH 文の INTO 句と互換性を持ちます。カーソル変数と結び付けられた問合せが戻す列の値に対して、INTO 句の中に、対応する型互換性のあるフィールドまたは変数が存在しなければなりません。また、フィールドまたは変数の数は、列の値の数と等しくなければなりません。

カーソル変数を、そのカーソル変数から取り出すサブプログラムの仮パラメータとして宣言する場合は、IN または IN OUT モードを指定しなければなりません。ただし、サブプログラムがカーソル変数もオープンする場合は、IN OUT モードを指定しなければなりません。

最終的に、FETCH 文は行を戻すことに失敗しますが、この状況が発生した場合に例外は呼び出されません。失敗を検出するには、カーソル属性 %FOUND または %NOTFOUND を使わなければなりません。詳細は、5-31 ページの「[カーソル属性の使用](#)」を参照してください。

クローズしている、または一度もオープンされていないカーソルまたはカーソル変数からフェッチを実行すると、PL/SQL によって事前定義の例外 INVALID_CURSOR が呼び出されます。

例

次の例に示されているとおり、カーソルに対応する問合せの中の変数はカーソルがオープンされたときにしか評価されません。

```
DECLARE
    my_sal NUMBER(7,2);
    n      INTEGER(2) := 2;
    CURSOR emp_cur IS SELECT  n*sal FROM emp;
BEGIN
    OPEN emp_cur;  -- n equals 2 here
    LOOP
        FETCH emp_cur INTO my_sal;
        EXIT WHEN emp_cur%NOTFOUND;
        -- process the data
        n := n + 1; -- does not affect next FETCH; sal will be multiplied by 2
    END LOOP;
```

次の Pro*C の例では、ホスト・カーソル変数から行をフェッチして、emp_rec という名前のホスト・レコード（構造体）に入れます。

```
/* Exit loop when done fetching. */
EXEC SQL WHENEVER NOTFOUND DO break;
for (;;)
{
    /* Fetch row into record. */
    EXEC SQL FETCH :emp_cur INTO :emp_rec;
}
```

次の例では、同じカーソル変数を使用した別々のフェッチで、異なる INTO 句を使用できることを示しています。各 FETCH 文で同じ結果セットから別の行を取り出します。

```
for (;;)
{
    /* Fetch row from result set. */
    EXEC SQL FETCH :emp_cur INTO :emp_rec1;
    /* Fetch next row from same result set. */
    EXEC SQL FETCH :emp_cur INTO :emp_rec2;
}
```

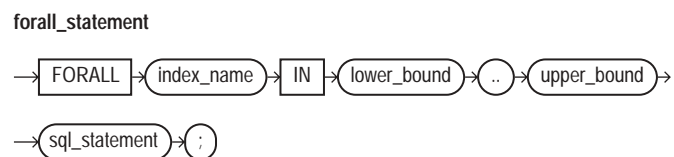
関連項目

CLOSE 文、カーソル、カーソル変数、LOOP 文、OPEN 文、OPEN-FOR 文

FORALL 文

FORALL 文は、コレクションを SQL エンジンに送信する前にバルク・バインド入力するよう、PL/SQL エンジンに指示を与えます。FORALL 文は反復スキームを含んでいますが、FOR ループではありません。詳細は、4-27 ページの「[バルク・バインドの利用](#)」を参照してください。

構文



キーワードとパラメータの説明

index_name

コレクションの添字として、FORALL 文の中でだけ参照できる、未宣言の識別子です。

index_name の暗黙的な宣言は、ループの外側での宣言を上書きします。そのため、文の中では同じ名前の別の変数を参照できません。FORALL 文の中では、index_name は、式に使用したり値を代入したりできません。

lower_bound .. upper_bound

結果が整数の値で、連続した索引番号の有効な範囲を指定する式です。式は、最初に FORALL 文に入ったときにだけ評価されます。

索引には lower_bound の値が割り当てられます。この値が upper_bound の値を超えていない場合、SQL 文が実行され、索引は増分されます。索引の値がまだ upper_bound の値を超えていない場合、SQL 文がもう一度実行されます。この処理は、索引の値が upper_bound の値を超えるまで繰り返されます。超えた時点で、FORALL 文が終了します。

sql_statement

これは、コレクション要素を参照する INSERT 文、UPDATE 文または DELETE 文でなければなりません。

使用方法

SQL 文は複数のコレクションを参照できます。しかし、PL/SQL エンジンでは添字付きコレクションだけをバルク・バインドします。

指定した範囲のコレクション要素がすべて存在しなければなりません。要素が足りなかったり削除されていた場合は、エラーが発生します。

FORALL 文が失敗すると、データベースの変更は、各 SQL 文の実行の前にマークされた暗黙のセーブポイントまでロールバックされます。前の実行の間に行われた変更は、ロールバックされません。

例

次の例に示すように、上限と下限を使ってコレクションの任意のスライスをバルク・バインドできます。

```
DECLARE
  TYPE NumList IS VARRAY(15) OF NUMBER;
  depts NumList := NumList();
BEGIN
  -- fill varray here
  ...
  FORALL j IN 6..10 -- bulk-bind middle third of varray
    UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(j);
END;
```

PL/SQL エンジンでは添字付きコレクションだけをバルク・バインドします。このため、次の例では、ファンクション median に渡されるコレクション sals はバルク・バインドされません。

```
FORALL i IN 1..20
  INSERT INTO emp2 VALUES (enums(i), names(i), median(sals), ...);
```

関連項目

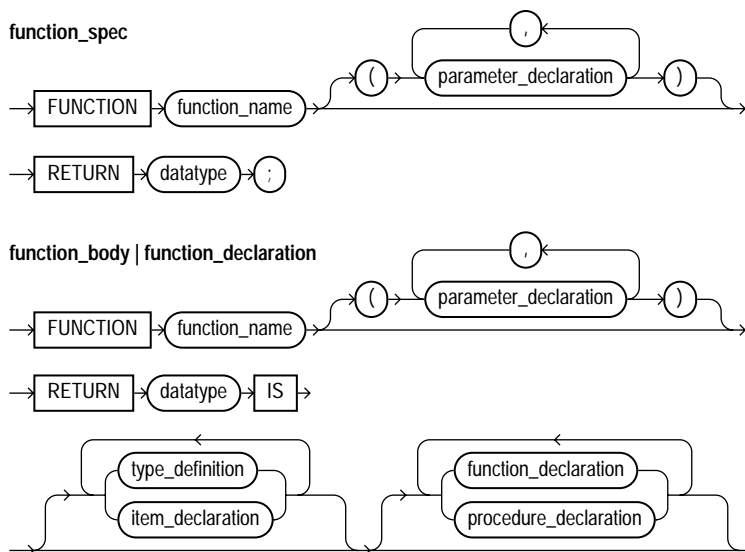
BULK COLLECT 句

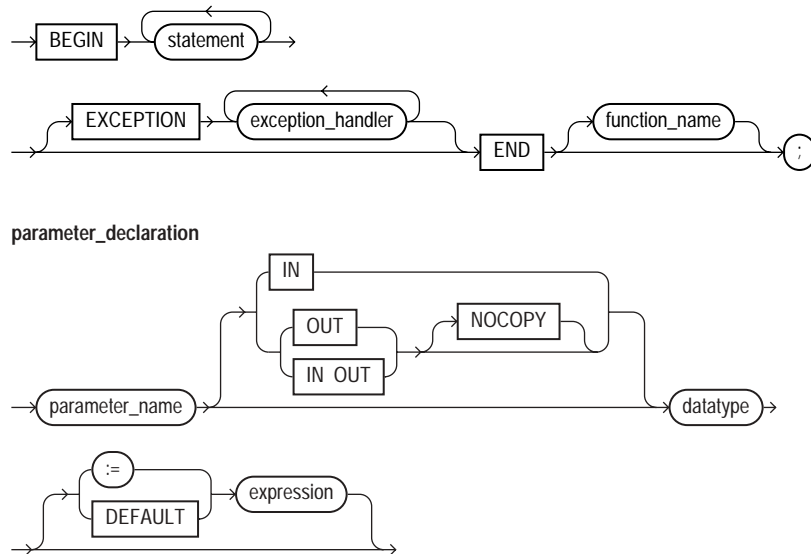
ファンクション

ファンクションとは、パラメータを指定して起動できるサブプログラムのことです。一般に、ファンクションは値を計算するために使います。ファンクションには、仕様部と本体の2つの部分があります。ファンクションの仕様部はキーワード `FUNCTION` で始め、結果の値のデータ型を指定する `RETURN` 句で終わります。パラメータ宣言はオプションです。パラメータを取らないファンクションではカッコを書きません。ファンクション本体は、キーワード `IS` で始め、キーワード `END` で終わります。END の後には、オプションとしてファンクション名を続けることができます。

ファンクション本体には、宣言部（オプション）、実行部、例外処理部（オプション）の3つの部分があります。宣言部には、型およびカーソル、定数、変数、例外と、サブプログラムが含まれます。これらの項目はローカルで、ファンクションを終了すると消去されます。実行部には、値の代入、実行の制御および Oracle データの操作を実行する文があります。例外処理部には、実行の途中で呼び出された例外を処理する例外ハンドラを入れます。詳細は、7-5 ページの「[ファンクション](#)」を参照してください。

構文





キーワードとパラメータの説明

function_name

ユーザー定義のファンクションを識別します。

parameter_name

仮パラメータを識別します。仮パラメータとは、ファンクションの仕様部で宣言され、ファンクション本体の中で参照される変数のことです。

IN、OUT、IN OUT

これらのパラメータ・モードは、仮パラメータの動作を定義します。IN パラメータは、コールされるサブプログラムに値を渡すために使います。OUT パラメータは、サブプログラムのコール側に値を戻すために使います。IN OUT パラメータを使うと、コールされる側のサブプログラムに初期値を渡して、コールした側に更新された値を戻すことができます。

NOCOPY

コンパイラ・ヒント（ディレクティブではない）です。これによって、PL/SQL コンパイラは OUT および IN OUT パラメータを、デフォルトの値方式ではなく、参照方式で渡すことができます。詳細は、7-17 ページの「[NOCOPY コンパイラ・ヒント](#)」を参照してください。

datatype

これは、型指定子です。datatype の構文は、11-29 ページの「[定数と変数](#)」を参照してください。

:= | DEFAULT

この演算子またはキーワードを使うと、IN パラメータをデフォルト値に初期化できます。

expression

変数、定数、リテラル、演算子、ファンクション・コールの任意の組合せです。最も単純な式は、1 個の変数で構成されています。宣言が PL/SQL コンパイラによって処理されるとき、expression の値がパラメータに代入されます。その値とパラメータは、データ型に互換性がなければなりません。

RETURN

RETURN 句の開始を知らせるキーワードです。この句では、結果値のデータ型を指定します。

type_definition

これは、ユーザー定義のデータ型を指定します。type_definition の構文は、11-7 ページの「[ブロック](#)」を参照してください。

item_declaration

これは、プログラム・オブジェクトを宣言します。item_declaration の構文は、11-7 ページの「[ブロック](#)」を参照してください。

function_declaration

ファンクションの宣言です。function_declaration の構文は、11-78 ページの「[ファンクション](#)」を参照してください。

procedure_declaration

プロシージャの宣言です。procedure_declaration の構文は、11-123 ページの「[プロシージャ](#)」を参照してください。

exception_handler

例外ハンドラです。例外が呼び出されると、その例外に結び付けられた一連の文を実行します。exception_handler の構文は、11-54 ページの「[例外](#)」を参照してください。

使用方法

ファンクションは、式の一部としてコールされます。たとえば、ファンクション `sal_ok` は次のようにしてコールできます。

```
promotable := sal_ok(new_sal, new_title) AND (rating > 3);
```

ファンクションは少なくとも 1 つの RETURN 文を持っていなければなりません。RETURN 文が 1 つもない場合、PL/SQL は実行時に事前定義の例外 `PROGRAM_ERROR` をコールします。

ストアド・ファンクションは、副作用を制御するための特定の規則に従っている場合に限り、SQL 文からコールできます。7-7 ページの「副作用の制御」を参照してください。

ファンクションの仕様部と本体を合わせて 1 つの単位として作成できます。また、ファンクションの仕様部と本体を別々にすることもできます。このように、ファンクションをパッケージに入れると、インプリメンテーション上の細部を隠ぺいできます。パッケージ仕様部でファンクション仕様部を宣言せずに、パッケージ本体でファンクションを定義できます。しかし、このようなファンクションは、パッケージの中からしかコールできません。

ファンクションの中では、IN パラメータは定数のように取り扱われます。したがって、値を代入できません。OUT パラメータはローカル変数のように取り扱われます。したがって、値を変更して参照できます。IN OUT パラメータは初期化された変数のように取り扱われます。したがって、値を代入したり、その値を他の変数に代入したりできます。パラメータ・モードの概要は、7-16 ページの表 7-1 を参照してください。

ファンクションでは、OUT モードと IN OUT モードを使わないようにしてください。ファンクションの目的は、0 (ゼロ) 個以上のパラメータを取り、単一の値を戻すことです。また、サブプログラム専用でない変数の値を変更するなどの副作用にも注意してください。

ファンクションは、PL/SQL をサポートするすべての Oracle Tools で定義できます。ただし、一般的な用途に使用するには、CREATE 文でファンクションを作成し、Oracle データベースに格納しておかなければなりません。CREATE FUNCTION 文は、SQL*Plus から対話式で実行できます。

例

次のファンクションは、指定された銀行口座の残高を戻します。

```
FUNCTION balance (acct_id INTEGER) RETURN REAL IS
    acct_bal REAL;
BEGIN
    SELECT bal INTO acct_bal FROM accts WHERE acctno = acct_id;
    RETURN acct_bal;
END balance;
```

関連項目

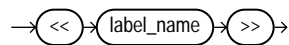
コレクション、パッケージ、プロシージャ、レコード

GOTO 文

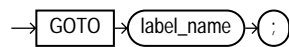
GOTO 文は、文ラベルまたはブロック・ラベルに無条件に分岐します。ラベルは有効範囲の中で他と重複しないもので、実行可能文か PL/SQL ブロックの前に置かれている必要があります。GOTO 文によって、制御はラベルの付いた文またはブロックに移ります。詳細は、3-13 ページの「[GOTO 文](#)」を参照してください。

構文

label_declaration



goto_statement



キーワードとパラメータの説明

label_name

実行可能文または PL/SQL ブロックに付けるラベル名で、未宣言の識別子です。GOTO 文を使うと、<<label_name>> の後に指定した文またはブロックに制御を移すことができます。

使用方法

GOTO 文の宛先として使えないものがあります。特に、GOTO 文は IF 文または LOOP 文、サブブロックには分岐できません。たとえば、次の GOTO 文は誤りです。

```
BEGIN
    ...
    GOTO update_row; -- illegal branch into IF statement
    ...
    IF valid THEN
        ...
        <<update_row>>
        UPDATE emp SET ...
    END IF;
```

GOTO 文では、現ブロックから同じブロックの別の場所、または囲みブロックには分岐できませんが、例外ハンドラには分岐できません。例外ハンドラの GOTO 文は、囲みブロックには分岐できますが、現ブロックには分岐できません。

GOTO 文を使ってカーソル FOR ループを途中で終了させると、カーソルは自動的にクローズされます。ループの内側で例外状況が発生した場合も、カーソルは自動的にクローズされます。

ある 1 つのブロックの中では、1 つのラベルは一度しか使えません。ただし、囲みブロックやサブブロックなどの他のブロックで使えます。ターゲット・ラベルが現ブロックにない場合、GOTO 文は囲みブロックのうちそのラベルが存在する最初のものに分岐します。

例

どのキーワードにも GOTO 文のラベルを付けられるわけではありません。GOTO 文のラベルは、実行可能文か PL/SQL ブロックの前でなければなりません。たとえば、次の GOTO 文は誤りです。

```
FOR ctr IN 1..50 LOOP
  DELETE FROM emp WHERE ...
  IF SQL%FOUND THEN
    GOTO end_loop; -- illegal
  END IF;
  ...
<<end_loop>>
END LOOP; -- not an executable statement
```

上の例をデバッグするには、次のように NULL 文を追加してください。

```
FOR ctr IN 1..50 LOOP
  DELETE FROM emp WHERE ...
  IF SQL%FOUND THEN
    GOTO end_loop;
  END IF;
  ...
<<end_loop>>
NULL; -- an executable statement that specifies inaction
END LOOP;
```

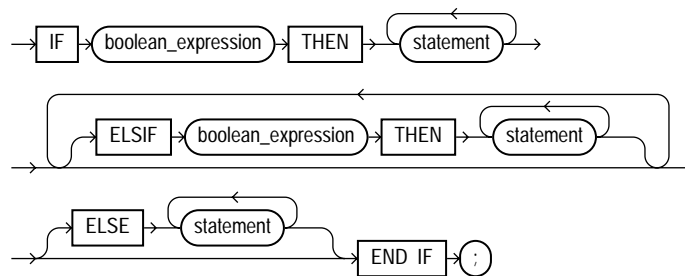
GOTO 文の無効な例と有効な例は、3-13 ページの「[GOTO 文](#)」を参照してください。

IF 文

IF 文を使うと、一連の文を条件に合わせて実行できます。一連の文が実行されるかどうかは、ブール式の値に依存します。詳細は、3-2 ページの「[条件制御: IF 文](#)」を参照してください。

構文

if_statement



キーワードとパラメータの説明

boolean_expression

結果が TRUE、FALSE、NULL のいずれかのブール値になる式です。式の結果が TRUE である場合だけに実行される一連の文が対応付けられています。

THEN

このキーワードは、これに先行するブール式と、後続の一連の文とを関連付けます。式の結果が TRUE の場合、関連付けられた一連の文が実行されます。

ELSIF

このキーワードを使用すると、IF に続く式、および先行する ELSIF に続くすべての式の結果が FALSE または NULL である場合に、ブール式が評価されます。

ELSE

制御がこのキーワードに達すると、その直後の一連の文が実行されます。

使用方法

IF 文には、IF-THEN、IF-THEN-ELSE、IF-THEN-ELSIF の 3 つの形式があります。IF 文の最も単純な形式では、ブール式を、キーワード THEN と END IF で囲まれた一連の文と対応付けます。一連の文は、式の結果が TRUE になった場合にだけ実行されます。式の結果が FALSE または NULL の場合、IF 文は何も実行しません。いずれの場合も、制御は次の文に渡されます。

IF 文の 2 つ目の形式では、キーワード ELSE が追加され、その後に THEN 句とは異なる処理をする一連の文を続けます。ブール式の結果が FALSE または NULL の場合にだけ、ELSE 句内の文の並びが実行されます。このように、ELSE 句を使うと必ずなんらかの一連の文が実行されます。

IF 文の 3 番目の形式では、キーワード ELSIF を使用して別のブール式を追加します。最初の式の結果が FALSE または NULL の場合、ELSIF 句は別の式を評価します。IF 文は任意の数の ELSIF 句を持つことができます。最後の ELSE 句はオプションです。ブール式は上から下に 1 つずつ評価されます。いずれかの式の結果が TRUE の場合、それに付随する一連の文が実行され、制御は次の文に移ります。すべての式の結果が FALSE または NULL の場合、ELSE 句の一連の文が実行されます。

一連の文が 1 つでも実行されると、IF 文の処理は終了します。このため、一連の文が複数実行されることはありません。ただし、THEN 句と ELSE 句には、さらに IF 文を入れることができます。つまり、IF 文はネストできます。

例

次の例で、shoe_count の値が 10 の場合、1 番目と 2 番目のブール式の結果はどちらも TRUE になります。しかし、1 つの式の結果が TRUE となり、それに対応付けられた一連の文が実行された時点で IF 文の処理は終了するため、order_quantity には正しい値 50 が代入されます。ELSIF に対応付けられた式は評価されず、制御は INSERT 文に移ります。

```
IF shoe_count < 20 THEN
    order_quantity := 50;
ELSIF shoe_count < 30 THEN
    order_quantity := 20;
ELSE
    order_quantity := 10;
END IF;

INSERT INTO purchase_order VALUES (shoe_type, order_quantity);
```

次の例では、`score` の値に応じて、2 つの状態メッセージのどちらかが表 `grades` に挿入されます。

```
IF score < 70 THEN
    fail := fail + 1;
    INSERT INTO grades VALUES (student_id, 'Failed');
ELSE
    pass := pass + 1;
    INSERT INTO grades VALUES (student_id, 'Passed');
END IF;
```

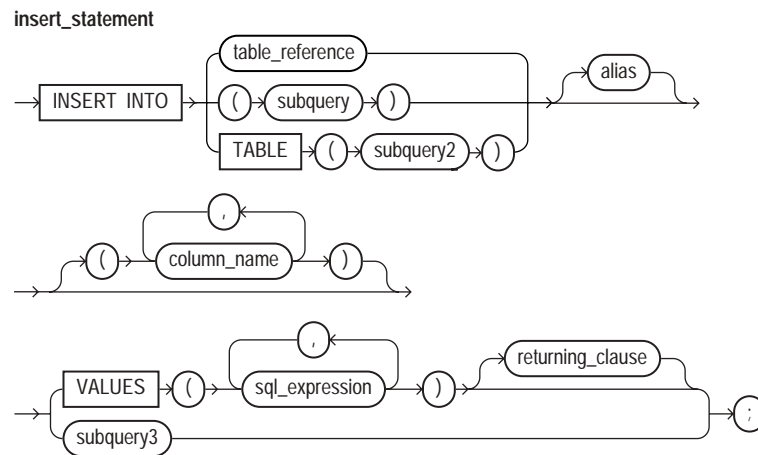
関連項目

式

INSERT 文

INSERT 文は、指定されたデータベースの表またはビューに、新しい行データを追加します。INSERT 文の詳細は、『Oracle8i SQL リファレンス』を参照してください。

構文



キーワードとパラメータの説明

table_reference

表またはビューを指定します。指定された表またはビューは、INSERT 文の実行時にアクセスできなければならない、ユーザーが INSERT 権限を持っている必要があります。table_reference の構文は、11-48 ページの「[DELETE 文](#)」を参照してください。

subquery

処理する行の集合を提供する SELECT 文です。構文は select_into_statement の構文と似ていますが、INTO 句は使えません。11-141 ページの「[SELECT INTO 文](#)」を参照してください。

TABLE (subquery2)

TABLE のオペランドは、1 つの列値を戻す SELECT 文です。これはネストした表、またはネストした表として割り当てられる varray である必要があります。演算子 TABLE は、値がスカラー値ではなくコレクションであることを Oracle に通知します。

alias

参照される表またはビューの別名（通常は短縮名）です。

column_name[, column_name]...

データベースの表またはビューの列のリストを識別します。列名の順番は、CREATE TABLE 文または CREATE VIEW 文で定義された順番である必要はありません。しかし、列の名前はリストの中で一度しか指定できません。表の列のうち、リストに含まれていないものがある場合、それらの列は NULL、または CREATE TABLE 文で指定されたデフォルト値に設定されます。

sql_expression

任意の有効な SQL の式です。詳細は、『Oracle8i SQL リファレンス』を参照してください。

VALUES (...)

この句は、式の値を、列リストの中の対応する列に代入します。列リストが指定されていない場合、最初の値は CREATE TABLE 文で定義された最初の列に、2 番目の値は 2 番目の列に、というように挿入されます。

列リストの中では、各列につき指定できる値は 1 つだけです。1 番目の値は 1 番目の列に、2 番目の値は 2 番目の列に、のように対応付けられます。列リストが存在しない場合は、表の中のすべての列について値を与えなければなりません。

挿入される値のデータ型は、列リストの対応する列のデータ型と互換性がなければなりません。

VALUES 句の中の副問合せが戻す値の数は、表に追加される行の数と同じです。この副問合せは、列リストのすべての列について値を戻さなければなりません。また、列リストが存在しない場合は、表の中のすべての列について値を戻さなければなりません。

subquery3

値または値の集合を VALUES 句に提供する SELECT 文です。この副問合せは、列リストのすべての列について、値を含む行を 1 つだけ戻さなければなりません。また、列リストが存在しない場合は、表の中のすべての列について値を含む行を 1 つだけ戻さなければなりません。

returning_clause

この句を使うと、挿入された行から値を戻せるので、後で行を SELECT で選択する必要がありません。取得した列値を、変数かホスト変数（またはその両方）あるいはコレクションかホスト変数（またはその両方）に代入できます。ただし、RETURNING 句はリモート、またはパラレルでの挿入には使用できません。returning_clause の構文は、11-48 ページの「[DELETE 文](#)」を参照してください。

使用方法

VALUES リストの中の文字リテラルと日付リテラルは、引用符 (') で囲まなければなりません。数値リテラルは引用符で囲みません。

暗黙的な SQL カーソルとカーソル属性 %NOTFOUND、%FOUND、%ROWCOUNT、および %ISOPEN を使うと、INSERT 文の実行に関する有用な情報にアクセスできます。

INSERT 文では、1 つまたは複数の行が挿入されるか、行の挿入が実行されないかのどちらかです。1 つまたは複数の行が挿入された場合、属性の値は次のようになります。

- SQL%NOTFOUND は、FALSE になる。
- SQL%FOUND は、TRUE になる。
- SQL%ROWCOUNT は、挿入された行数になる。

行が挿入されなかった場合、属性の値は次のようになります。

- SQL%NOTFOUND は、TRUE になる。
- SQL%FOUND は、FALSE になる。
- SQL%ROWCOUNT は 0 になる。

例

さまざまな形式の INSERT 文を次の例に示します。

```
INSERT INTO bonus SELECT ename, job, sal, comm FROM emp
  WHERE comm > sal * 0.25;
...
INSERT INTO emp (empno, ename, job, sal, comm, deptno)
  VALUES (4160, 'STURDEVIN', 'SECURITY GUARD', 2045, NULL, 30);
...
INSERT INTO dept
  VALUES (my_deptno, UPPER(my_dname), 'CHICAGO');
```

関連項目

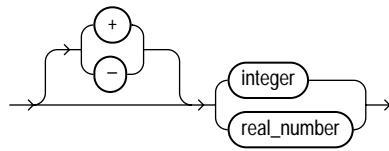
[SELECT 文](#)

リテラル

リテラルは、識別子によって表現する必要がない明示的な数値または文字、文字列、ブール値です。たとえば、数値リテラル 135 や文字列リテラル 'hello world' などです。詳細は、2-7 ページの「[リテラル](#)」を参照してください。

構文

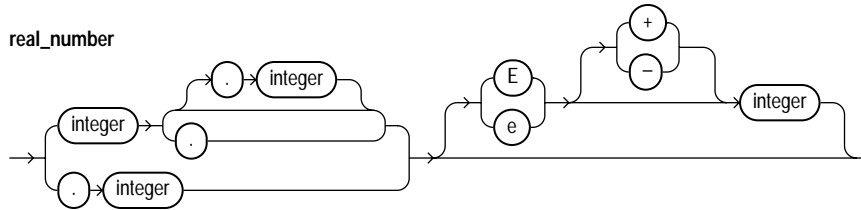
numeric_literal



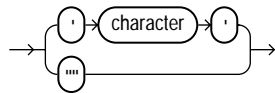
integer



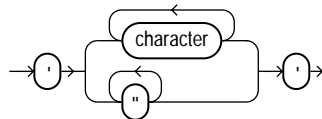
real_number



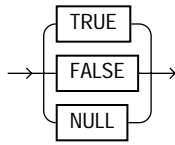
character_literal



string_literal



boolean_literal



キーワードとパラメータの説明

integer

小数点を持たない整数で、オプションとして符号を付けることができます。

real_number

小数点を持つ整数または小数で、オプションとして符号を付けることができます。

digit

数字 0 ~ 9 のうちのいずれかです。

char

PL/SQL キャラクタ・セットのメンバーです。詳細は、2-2 ページの「[キャラクタ・セット](#)」を参照してください。

TRUE、FALSE、NULL

事前定義のブール値です。

使用方法

算術式では、整数と実数の 2 種類の数値リテラルを使えます。数値リテラルは、句読点文字で区切らなければなりません。句読点以外に空白も使えます。

文字リテラルは引用符（アポストロフィ）で囲まれた 1 文字のことです。文字リテラルには PL/SQL キャラクタ・セットの印刷可能文字をすべて使用できます。つまり、英字および数字、空白、特殊記号を使用できます。

文字リテラルの中で、PL/SQL は大文字と小文字を区別します。たとえば、PL/SQL はリテラル 'Q' と 'q' を違うものとみなします。

リテラル

文字列リテラルは、引用符 (') で囲まれたゼロ個以上の文字の並びです。NULL 文字列 ('') はゼロ個の文字です。文字列の中でアポストロフィを表現したい場合は、引用符 (') を 2 つ書いてください。文字列リテラルの中で、PL/SQL は大文字と小文字を区別します。たとえば、PL/SQL はリテラル 'white' と 'White' を違うものとみなします。

また、文字列リテラルの中では、値に後続する空白が意味を持ちます。つまり、'abc' と 'abc ' は違います。リテラルの中の後続する空白は切り捨てられません。

ブール値 TRUE および FALSE はデータベース列に挿入できません。

例

次に数値リテラルの例を示します。

```
25    6.34    7E2    25e-03    .1    1.    +17    -4.4
```

次に文字リテラルの例を示します。

```
'H'    '&'    ' '    '9'    ']'    'g'
```

次に文字列リテラルの例を示します。

```
'$5,000'  
'02-AUG-87'  
'Don't leave without saving your work.'
```

関連項目

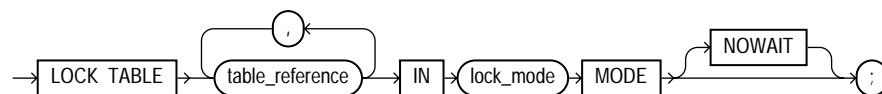
定数と変数、式

LOCK TABLE 文

LOCK TABLE 文を使用して、データベース表全体を指定のロック・モードでロックできます。これにより、表の整合性を維持したまま、表へのアクセスを共有したり、拒否したりできます。詳細は、5-44 ページの「[LOCK TABLE の使用](#)」を参照してください。

構文

lock_table_statement



キーワードとパラメータの説明

table_reference

ロックする対象の表またはビューです。LOCK TABLE 文を実行する場合は、この表またはビューがアクセス可能でなければなりません。table_reference の構文は、11-48 ページの「[DELETE 文](#)」を参照してください。

lock_mode

ロック・モードを指定するパラメータです。これは、ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE のいずれかです。

NOWAIT

これはオプションのキーワードであり、これを指定すると Oracle は他のユーザーが表をロックしていると待機しません。制御はただちにプログラムに戻されるので、他の処理を実行してから、改めてロックを試みてください。

使用方法

キーワード NOWAIT を省略すると、Oracle は表が利用できるようになるまで待ちます。待機時間に制限はありません。表ロックは、トランザクションがコミットまたはロールバックを発行したときに解除されます。

表がロックされていても、他のユーザーは表に対して問合せできますが、問合せを実行しても表のロックを取得できません。

プログラムに SQL ロッキング文が含まれている場合は、ロックを要求している Oracle ユーザーが、ロックのために必要な権限を持っていることを確認してください。DBA は、任意の表をロックできます。他のユーザーは、自分が所有する表か、SELECT、INSERT、UPDATE、または DELETE などの権限の付与されている表をロックできます。

例

次の文は、表 `accts` を共有モードでロックします。

```
LOCK TABLE accts IN SHARE MODE;
```

関連項目

COMMIT 文、ROLLBACK 文、UPDATE 文

LOOP 文

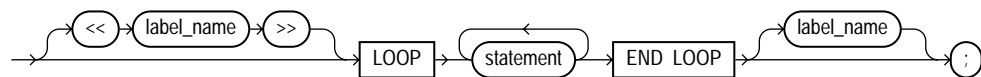
LOOP 文は一連の文を繰り返して実行します。ループとは、繰り返して実行する一連の文をキーワードで囲んだものです。PL/SQL では、次の種類のループ文がサポートされています。

- 基本ループ
- WHILE ループ
- FOR ループ
- カーソル FOR ループ

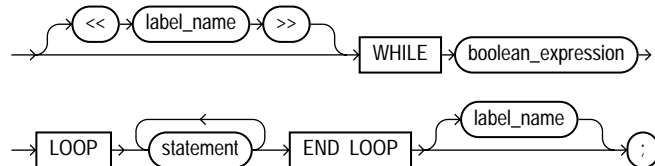
詳細は、3-6 ページの「[反復制御: LOOP 文と EXIT 文](#)」を参照してください。

構文

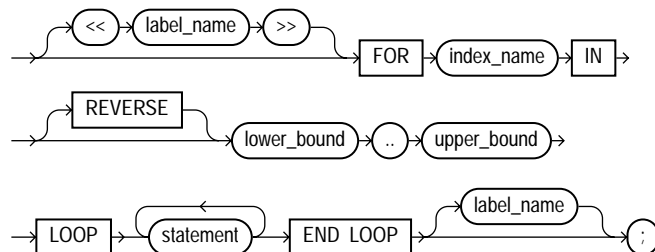
basic_loop_statement



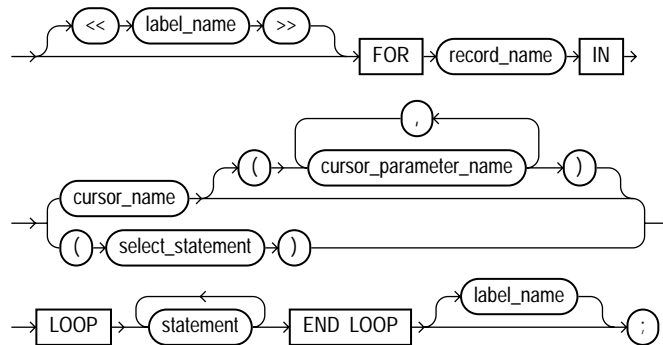
while_loop_statement



for_loop_statement



cursor_for_loop_statement



キーワードとパラメータの説明

label_name

オプションとしてループに付けるラベル名で、未宣言の識別子です。label_name を使う場合は、二重の山カッコで囲み、ループの先頭に置かなければなりません。オプションとして、label_name を、山カッコで囲まずに、ループの最後に置くこともできます。

label_name を EXIT 文の中で使用すると、label_name によってラベル付けされているループを終了できます。現在のループだけでなく、外側のループも終了できます。

外側の FOR ループと、その内側のネストされた FOR ループの索引が同じ名前である場合、ネストされたループから外側のループの索引を参照できません。ただし、外側のループが label_name でラベル付けされている場合は、次のようにドット表記法を使えば参照できます。

label_name.index_name

次の例では、外側のループとネストされたループで使われている同じ名前の 2 つのループ索引を比較しています。

```

<<outer>>
FOR ctr IN 1..20 LOOP
    ...
    <<inner>>
    FOR ctr IN 1..10 LOOP
        IF outer.ctr > ctr THEN ...
    END LOOP inner;
END LOOP outer;

```


basic_loop_statement

LOOP 文の最も単純な形式は、キーワード LOOP と END LOOP で一連の文を囲む基本（または無限）ループです。ループが繰り返されるたびに一連の文が実行され、制御がループの先頭に戻ります。処理の続行が望ましくない場合、または不可能になった場合は、EXIT、GOTO、または RAISE 文を使ってループを終了できます。例外が呼び出された場合もループは終了します。

while_loop_statement

WHILE-LOOP 文は、ブール式を、キーワード LOOP と END LOOP で囲まれた一連の文と対応付けます。ループを反復する前に条件が評価されます。式の結果が TRUE の場合、一連の文が実行されてから、ループの先頭で制御が再開します。式の結果が FALSE または NULL の場合、ループは実行されず、制御は次の文に渡されます。

boolean_expression

結果が TRUE、FALSE、NULL のいずれかのブール値になる式です。式の結果が TRUE である場合だけに実行される一連の文が対応付けられています。boolean_expression の構文は、11-62 ページの「式」を参照してください。

for_loop_statement

WHILE ループの反復回数はループが終了するまではわかりませんが、FOR ループの反復回数はループに入る前からわかっています。数値 FOR ループは、指定された整数の範囲内でループを繰り返し実行します。（カーソル FOR ループはカーソルの結果セットの中で繰り返し実行します。）繰り返しの範囲は、キーワード FOR と LOOP に囲まれた反復スキーマの一部です。

繰り返しの範囲は FOR ループに入った段階で評価され、それ以降は評価されません。ループの中の一連の文は、lower_bound..upper_bound によって定義された範囲の整数 1 つにつき 1 回実行されます。1 回の繰り返しが終わると、ループ索引に増分が加えられます。

index_name

ループ索引に名前をつける未宣言の識別子です（ループ・カウンタと呼ばれる場合もある）。有効範囲はループ自体になります。そのため、ループの外側では索引を参照できません。

index_name の暗黙的な宣言は、ループの外側での宣言を上書きします。このため、ループの内側から同じ名前の変数を参照できません。ただし、次のようにラベルを使うと参照できます。

```
<<main>>
DECLARE
    num NUMBER;
BEGIN
    ...
    FOR num IN 1..10 LOOP
        ...
        IF main.num > 5 THEN -- refers to the variable num,
```

```

        ...                -- not to the loop index
    END IF;
    END LOOP;
END main;

```

ループの内側では、索引は定数のように扱われます。索引は、式の中で使えますが、値を代入できません。

lower_bound .. upper_bound

結果が整数値になる式です。式は、最初にループに入ったときにだけ評価されます。

デフォルトでは、ループ索引には `lower_bound` の値が割り当てられます。この値が `upper_bound` の値を超えていない場合、ループの中の一連の文が実行され、索引が増分されます。索引の値がまだ `upper_bound` の値を超えていない場合、一連の文がもう一度実行されます。この処理は、索引の値が `upper_bound` の値を超えるまで繰り返されます。そうなった時点で、ループが終了します。

REVERSE

デフォルトでは、反復は、範囲の下限から上限に上向きに進みます。しかし、キーワード `REVERSE` を使用すると、反復は上限から下限に下向きに進みます。

たとえば、

```

FOR i IN REVERSE 1..10 LOOP -- i starts at 10, ends at 1
    -- statements here execute 10 times
END LOOP;

```

ループ索引には `upper_bound` の値が割り当てられます。この値が `lower_bound` の値を下回っていない場合、ループの中の一連の文が実行され、索引が減分されます。索引の値がまだ `lower_bound` の値を下回っていない場合、一連の文がもう一度実行されます。この処理は、索引の値が `lower_bound` の値を下回るまで繰り返されます。そうなった時点で、ループが終了します。

cursor_for_loop_statement

カーソル `FOR` ループは、ループ索引を `%ROWTYPE` 属性のレコードとして暗黙的に宣言し、カーソルをオープンして、結果セットから行の値を取り出してレコード中のフィールドに入れるという作業を繰り返し、すべての行を処理したらカーソルをクローズします。このようにして、ループの中の一連の文は、`cursor_name` と対応付けられた問合せを満たす行ごとに 1 回実行されます。

cursor_name

現在の有効範囲の中で、事前に宣言されている明示カーソルを識別します。カーソル `FOR` ループに入ると、`cursor_name` は、`OPEN` 文または外側のカーソル `FOR` ループによって、すでにオープンされたカーソルを参照できません。

record_name

暗黙的に宣言されたレコードを識別します。このレコードは `cursor_name` によって取り出された行と同じ構造を持ち、次のように宣言されたレコードと等価です。

```
record_name cursor_name%ROWTYPE;
```

レコードはループの内側だけで定義されています。ループの外側からこのレコードのフィールドを参照できません。`record_name` の暗黙的な宣言は、ループの外側での宣言を上書きします。そのため、同じ名前のレコードは、ラベルを使わない限り、ループの内側から参照できません。

レコード中のフィールドには、暗黙のうちにフェッチされた行の列値が格納されます。フィールドの名前とデータ型は、対応する列の名前とデータ型と同じです。フィールドの値にアクセスするには、次のようにドット表記法を使います。

```
record_name.field_name
```

FOR ループのカーソルによってフェッチされた選択項目の名前は、単純名でなければなりません。また、それらが式である場合は、別名を持っていなければなりません。次の例では、選択項目 `sal+NVL(comm,0)` の別名として `wages` を指定しています。

```
CURSOR c1 IS SELECT empno, sal+NVL(comm,0) wages, job ...
```

cursor_parameter_name

カーソルのパラメータ、すなわちカーソルの仮パラメータとして宣言された変数を識別します。(`cursor_parameter_declaration` の構文は、11-44 ページの「[カーソル](#)」を参照してください。) カーソルのパラメータは、問合せの中で定数が使える場所ならばどこでも使えます。カーソルの仮パラメータは IN パラメータでなければなりません。

select_statement

使用不可能な内部カーソルに対応付けられた問合せです。構文は `select_into_statement` の構文と似ていますが、`INTO` 句は使えません。11-141 ページの「[SELECT INTO 文](#)」を参照してください。PL/SQL は内部カーソルを自動的に宣言し、オープンし、データをフェッチしてクローズします。`select_statement` は独立した文ではないので、暗黙 SQL カーソルは適用されません。

使用方法

`EXIT WHEN` 文を使用すると、任意のループを途中で終了できます。`WHEN` 句の中のブール式の結果が `TRUE` の場合、ループはただちに終了します。

カーソル FOR ループを終了すると、`EXIT` 文または `GOTO` 文を使ってループを途中で終了した場合でも、カーソルは自動的にクローズされます。ループの内側で例外状況が発生した場合も、カーソルは自動的にクローズされます。

例

次のカーソル FOR ループでは、ボーナスを計算し、その結果をデータベース表に挿入しています。

```
DECLARE
    bonus REAL;
    CURSOR c1 IS SELECT empno, sal, comm FROM emp;
BEGIN
    FOR clrec IN c1 LOOP
        bonus := (clrec.sal * 0.05) + (clrec.comm * 0.25);
        INSERT INTO bonuses VALUES (clrec.empno, bonus);
    END LOOP;
    COMMIT;
END;
```

関連項目

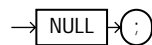
カーソル、EXIT 文、FETCH 文、OPEN 文、%ROWTYPE 属性

NULL 文

NULL 文は、アクションを起こさないことを明示するために使います。NULL 文は制御を次の文に渡すことしかしません。代替アクションが指定できる構成体では、NULL 文はプレースホルダとしての役割を果たします。詳細は、3-16 ページの「[NULL 文](#)」を参照してください。

構文

null_statement



使用方法

NULL 文には、条件文の意味とアクションを明確にすることによって、コードをわかりやすくする効果があります。読み手に対して、代替アクションを誤って見逃したのではなく、実際にアクションが不要であるということを伝えることができます。

IF 文のすべての句には、少なくとも 1 つの実行可能文がなければなりません。NULL 文はこの条件を満たします。つまり、構文上では句が必要だがアクションは起こしたくないという場合には、NULL 文を使えます。NULL 文とブール値 NULL は無関係です。

例

次の例では、NULL 文によって、販売員だけがコミッションを受け取れることを明確にしています。

```
IF job_title = 'SALESPERSON' THEN
    compute_commission(emp_id);
ELSE
    NULL;
END IF;
```

次の例では、NULL 文によって、名前のない例外ではアクションを起こさないことを明確にしています。

```
EXCEPTION
...
WHEN OTHERS THEN
    NULL;
```

オブジェクト型

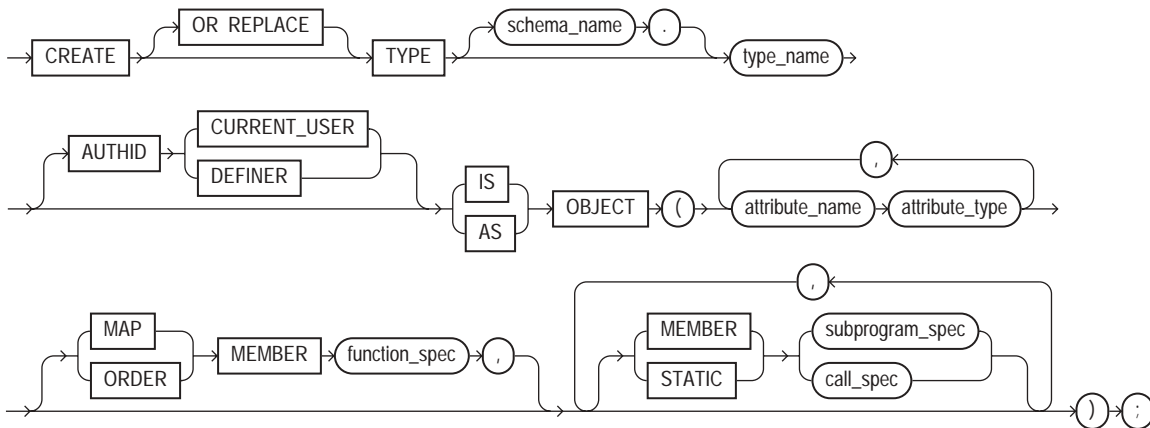
オブジェクト型は、データを操作するのに必要なファンクションおよびプロシージャとともにデータ構造をカプセル化するユーザー定義の複合データ型です。データ構造を形成する変数は、属性と呼ばれます。オブジェクト型の動作を特徴付けるファンクションとプロシージャはメソッドと呼ばれます。

現在のところ、PL/SQL ではオブジェクト型の定義はできません。それらは CREATE 文を使って作り、Oracle データベースに格納して、たくさんのプログラムで共有できるようにしなければなりません。CREATE TYPE 文を使用してオブジェクト型を（たとえば SQL*Plus など）定義する場合、実世界のオブジェクトに対応する抽象テンプレートを作ります。テンプレートでは、アプリケーション環境でオブジェクトで必要となる属性と動作だけを指定します。

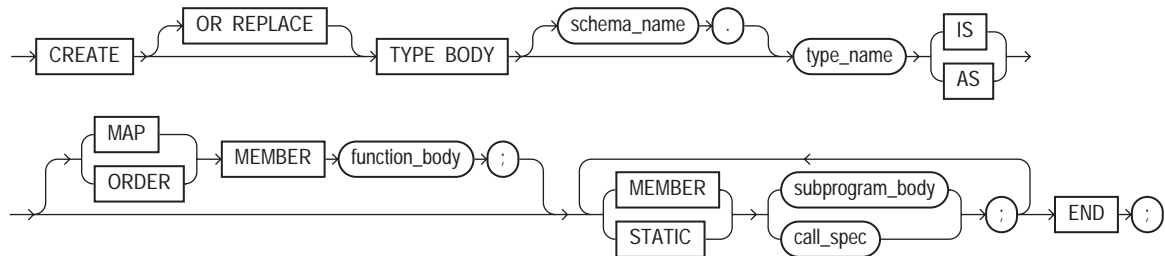
属性の集合によって形成されるデータ構造体はパブリックです（クライアント・プログラムから参照できる）。しかし、正しいプログラムは、それを直接操作したりはしません。提供される一連のメソッドを使います。そのようにして、データは常に適切な状態に保たれます。実行時には、データ構造体に値が入れられた時点で、オブジェクト型のインスタンスが作られることになります。インスタンス（通常オブジェクトと呼ばれる）は、必要な数だけ作れます。詳細は、[第 9 章の「オブジェクト型」](#)を参照してください。

構文

object_type_declaration | object_type_spec



object_type_body



キーワードとパラメータの説明

type_name

オブジェクトのそれ以降の宣言で使うユーザー定義の型指定子です。

AUTHID 句

すべてのメンバー・メソッドがその定義者（デフォルト）と実行者のどちらの権限で実行するか、およびスキーマ・オブジェクトへの未修飾の参照が定義者と実行者のどちらのスキーマで解決されるかを決定します。詳細は、7-29 ページの「[実行者権限と定義者権限](#)」を参照してください。

attribute_name

オブジェクト属性です。名前はそのオブジェクト型の中で固有でなければなりません（他のオブジェクト型内では使えます）。属性の宣言内では、代入演算子または DEFAULT 句を使っての属性の初期化はできません。また、属性に NOT NULL 制約を課することはできません。

attribute_datatype

これは、LONG、LONG RAW、NCHAR、NCLOB、NVARCHAR2、ROWID、PL/SQL 固有の BINARY_INTEGER 型とそのサブタイプ、BOOLEAN、PLS_INTEGER、RECORD、REF CURSOR、%TYPE、%ROWTYPE を除く任意の Oracle データ型と、PL/SQL パッケージ内で定義される型です。

MEMBER | STATIC

このキーワードは、オブジェクト型仕様部で、サブプログラムまたはコール仕様部をメソッドとして宣言するのに使用します。メソッドの名前には、オブジェクト型またはその属性のいずれかと同じ名前は使えません。次に示すように、MEMBER メソッドはインスタンスで起動されます。

```
instance_expression.method()
```

しかし、次に示すように、STATIC メソッドはインスタンスではなくオブジェクト型で起動されます。

```
object_type_name.method()
```

オブジェクト型仕様部内のそれぞれのサブプログラム仕様部に対応するサブプログラム本体が、オブジェクト型本体内に存在していなければなりません。仕様部と本体を一致させるために、コンパイラは、それらのヘッダーをトークンごとに比較します。このため、ヘッダーは一語一語が一致していなければなりません。

MEMBER メソッドは SELF という名前の組込みパラメータを受け入れます。これはオブジェクト型のインスタンスです。暗黙のうちに宣言されていても明示的に宣言されていても、SELF は常に MEMBER メソッドに渡される第 1 パラメータです。しかし、STATIC メソッドは SELF を受け入れたり、参照したりできません。

メソッドの本体では、SELF はメソッドが起動されたオブジェクトを示します。たとえば、メソッド transform は SELF を IN OUT パラメータとして宣言します。

```
CREATE TYPE Complex AS OBJECT (  
    MEMBER FUNCTION transform (SELF IN OUT Complex) ...
```

SELF にそれ以外のデータ型は指定できません。MEMBER ファンクションでは、SELF が宣言されていない場合、そのパラメータ・モードは IN にデフォルトで設定されます。しかし、MEMBER プロシージャでは、SELF が宣言されていない場合、そのパラメータ・モードは IN OUT にデフォルトで設定されます。SELF には OUT パラメータを指定できません。

MAP

このキーワードは、CHAR や REAL のような事前定義済みの順序を持つスカラー・データ型の値にオブジェクトをマップすることによって、オブジェクトを順序付けるメソッドであることを示します。PL/SQL は順序付けを使って、 $x > y$ などのブール式を評価したり、DISTINCT、GROUP BY および ORDER BY 句によって暗黙のうちに必要となる比較を実行したりします。マップ・メソッドは、すべてのオブジェクトを順序付けする際の、オブジェクトの相対的な位置を戻します。

1 つのオブジェクト型のマップ・メソッドは 1 つだけです。このメソッドは、DATE、NUMBER、VARCHAR2、または CHARACTER や INTEGER、REAL などの ANSI SQL 型のいずれかを戻り型として持つ、パラメータのないファンクションでなければなりません。

ORDER

このキーワードは、2つのオブジェクトを比較するメソッドであることを示します。1つのオブジェクト型の順序付けメソッドは1つだけです。このメソッドは、戻り型が数値のファンクションでなければなりません。

順序付けメソッドはいずれも、組み込みパラメータ `SELF` と、同じ型の別のオブジェクトの、2つのパラメータをとります。`c1` および `c2` が `Customer` オブジェクトの場合、`c1 > c2` などの比較操作を実行すると、自動的にメソッド `match` がコールされます。このメソッドの戻り値は、負数、0 (ゼロ)、または正数であり、それぞれ `SELF` が他方のパラメータより小さい、等しい、大きいことを示しています。順序付けメソッドに渡されるパラメータのいずれかが `NULL` の場合、メソッドは `NULL` を戻します。

subprogram_spec

メンバー・ファンクションまたはプロシージャへのインタフェースの宣言です。構文は、`function_spec` や `procedure_spec` の構文と似ていますが、終了記号は使えません。11-78 ページの「[ファンクション](#)」または 11-123 ページの「[プロシージャ](#)」を参照してください。

subprogram_body

メンバー・ファンクションまたはプロシージャの基盤となるインプリメンテーションの定義です。構文は、`function_body` や `procedure_body` の構文と似ていますが、終了記号は使えません。11-78 ページの「[ファンクション](#)」または 11-123 ページの「[プロシージャ](#)」を参照してください。

call_spec

Oracle データ・ディクショナリ内の外部 C ファンクションまたは Java メソッドを発行します。これは、対応する SQL に名前、パラメータ型および戻り型をマップすることによって、ルーチンを発行します。Java コール仕様部を作成する方法は、『Oracle8i Java ストアド・プロシージャ開発者ガイド』を参照してください。C コール仕様部を作成する方法は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

使用方法

オブジェクト型を宣言してスキーマにインストールしたなら、任意の PL/SQL ブロック、サブプログラム、またはパッケージの中で、それを使ってオブジェクトを宣言できます。たとえば、そのオブジェクト型を使って属性、列、変数、バインド変数、レコード・フィールド、表要素、仮パラメータ、またはファンクション結果のデータ型を指定できます。

パッケージと同様に、オブジェクト型は仕様部と本体という2つの部分から構成されます。仕様部はアプリケーションへのインタフェースです。ここでは、データ構造体（属性の集合）とデータ操作に必要な演算（メソッド）を宣言します。本体ではメソッドを完全に定義し、それによって仕様部をインプリメントします。

メソッドを使うためにクライアント・プログラムが必要とするすべての情報は、仕様部にあります。仕様部は操作インタフェース、そして本体はブラック・ボックスと考えてください。仕様部を変更しなくても、本体をデバッグ、拡張、または置換できます。

オブジェクト型はデータと操作をカプセル化します。そのため、属性とメソッドはオブジェクト型仕様部で宣言できますが、定数、例外、カーソル、型は宣言できません。少なくとも1つの属性が必要です (最大で 1000)。メソッドはオプションです。

オブジェクト型の仕様部では、メソッドより前にすべての属性を宣言しなければなりません。サブプログラムだけがインプリメンテーションを必要とします。そのため、オブジェクト型の仕様部に属性およびコール仕様部の宣言しかない場合、またはそのいずれかの宣言しかない場合、オブジェクト型の本体は不要です。本体では属性を宣言できません。オブジェクト型の仕様部のすべての宣言は、パブリックです (オブジェクト型の外側から参照できる)。

属性は、オブジェクト型内の位置によってではなく、名前によってのみ参照できます。属性にアクセスしたり、その値を変更したりするには、ドット表記法を使います。属性名を連鎖させて、ネストされたオブジェクト型の属性にアクセスできます。

オブジェクト型の中でメソッドは、修飾子なしで属性および他のメソッドを参照できます。SQL 文からパラメータのないメソッドをコールするには、空のパラメータ・リストが必要です。プロシージャ文では、コールを連鎖しないかぎり空のパラメータ・リストはなくてもかまいません。連鎖する場合は、最後のコール以外のすべてで空のパラメータ・リストが必要です。

SQL 文からは、NULL インスタンス (SELF が NULL である状態) で MEMBER メソッドをコールすると、メソッドは起動されず NULL が戻されます。プロシージャ文からは、NULL インスタンスで MEMBER メソッドをコールすると、メソッドが起動される前に事前定義の例外 SELF_IS_NULL が呼び出されます。

マップ・メソッドか順序付けメソッドを宣言できますが、その両方は宣言できません。どちらかのメソッドを宣言すれば、オブジェクトを SQL 文およびプロシージャ文によって比較できます。しかしどちらのメソッドも宣言しない場合、オブジェクトは SQL 文でしか比較できず、しかも等しいか等しくないかの比較しかできません。同じ型の2つのオブジェクトが等しいとされるのは、それらの対応する属性の値が等しい場合だけです。

パッケージ化されたサブプログラムと同じく、同じ種類 (ファンクションまたはプロシージャ) のメソッドはオーバーロードできます。つまり、仮パラメータの数、順序、またはデータ型の種類が違っていれば、同じ名前を複数の異なるメソッドで使えます。

どのオブジェクト型にもコンストラクタ・メソッド (略してコンストラクタ) があります。それは、そのオブジェクト型と同じ名前のシステム定義のファンクションです。コンストラクタは、そのオブジェクト型のインスタンスを初期化したり、そのインスタンスを戻すために使います。PL/SQL は、暗黙のうちにコンストラクタをコールすることは決してないため、明示的にコールすることが必要です。コンストラクタは、ファンクション・コールが許可されているところでコールできます。

例

次の SQL*Plus スクリプトでは、スタックのためのオブジェクト型を定義しています。スタックに最後に追加された項目が、最初に削除される項目となります。スタックは、操作 *push* および *pop* により、後入れ先出し (LIFO) 方式で更新されます。スタックの最も簡単なインプリメンテーションは、整数の配列を使うものです。整数は配列要素として格納され、配列の片方の端がスタックの先頭を表します。

```
CREATE TYPE IntArray AS VARRAY(25) OF INTEGER;

CREATE TYPE Stack AS OBJECT (
    max_size INTEGER,
    top      INTEGER,
    position IntArray,
    MEMBER PROCEDURE initialize,
    MEMBER FUNCTION full RETURN BOOLEAN,
    MEMBER FUNCTION empty RETURN BOOLEAN,
    MEMBER PROCEDURE push (n IN INTEGER),
    MEMBER PROCEDURE pop (n OUT INTEGER)
);

CREATE TYPE BODY Stack AS
    MEMBER PROCEDURE initialize IS
        -- fill stack with nulls
    BEGIN
        top := 0;
        -- call constructor for varray and set element 1 to NULL
        position := IntArray(NULL);
        max_size := position.LIMIT; -- use size constraint (25)
        position.EXTEND(max_size - 1, 1); -- copy element 1
    END initialize;

    MEMBER FUNCTION full RETURN BOOLEAN IS
        -- return TRUE if stack is full
    BEGIN
        RETURN (top = max_size);
    END full;

    MEMBER FUNCTION empty RETURN BOOLEAN IS
        -- return TRUE if stack is empty
    BEGIN
        RETURN (top = 0);
    END empty;

    MEMBER PROCEDURE push (n IN INTEGER) IS
        -- push integer onto stack
    BEGIN
        IF NOT full THEN
```

オブジェクト型

```
        top := top + 1;
        position(top) := n;
    ELSE -- stack is full
        RAISE_APPLICATION_ERROR(-20101, 'stack overflow');
    END IF;
END push;

MEMBER PROCEDURE pop (n OUT INTEGER) IS
-- pop integer off stack and return its value
BEGIN
    IF NOT empty THEN
        n := position(top);
        top := top - 1;
    ELSE -- stack is empty
        RAISE_APPLICATION_ERROR(-20102, 'stack underflow');
    END IF;
END pop;
END;
```

メンバー・プロシージャ `push` および `pop` では、組み込みプロシージャ `raise_application_error` を使ってユーザー定義のエラー・メッセージを発行していることに注意してください。このようにして、エラーをクライアント・プログラムに報告し、未処理の例外をホスト環境に戻さないようにしています。

次の例が示す内容は、オブジェクト型のネストです。

```
CREATE TYPE Address AS OBJECT (
    street_address VARCHAR2(35),
    city            VARCHAR2(15),
    state          CHAR(2),
    zip_code       INTEGER
);

CREATE TYPE Person AS OBJECT (
    first_name  VARCHAR2(15),
    last_name   VARCHAR2(15),
    birthday    DATE,
    home_address Address, -- nested object type
    phone_number VARCHAR2(15),
    ss_number   INTEGER,
);
```

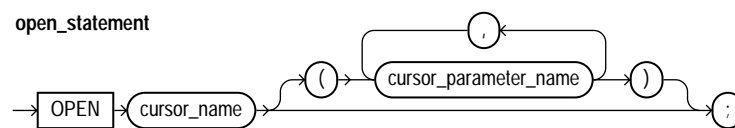
関連項目

ファンクション、パッケージ、プロシージャ

OPEN 文

OPEN 文は、明示カーソルに対応付けられた複数行の問合せを実行します。また、Oracle が問合せを処理するのに使うリソースを割り振り、結果セットを識別します（結果セットは、問合せの検索条件に合致するすべての行で構成されています）。カーソルは、結果セットの最初の行の前に置かれます。詳細は、5-6 ページの「[カーソル管理](#)」を参照してください。

構文



キーワードとパラメータの説明

cursor_name

現行の有効範囲のうちそれより前に宣言されていて、現在オープンされていない明示カーソルです。

cursor_parameter_name

カーソルのパラメータ、すなわちカーソルの仮パラメータとして宣言された変数を識別します。（`cursor_parameter_declaration` の構文は、11-44 ページの「[カーソル](#)」を参照してください。）カーソルのパラメータは、問合せの中で定数が見える場所ならばどこでも使えます。

使用方法

一般に、PL/SQL による明示カーソルは、それを最初にオープンするときにしか解析されません。また、SQL 文の解析（およびそれによる暗黙カーソルの作成）は、その文が初めて実行されるときにしか行われません。解析された SQL 文は、すべてキャッシュに入れられます。SQL 文を解析し直さなければならないのは、新しい SQL 文によってキャッシュから押し出された場合に限られます。

したがって、カーソルを再オープンするには、まずクローズしなければなりませんが、PL/SQL はカーソルに対応付けられた `SELECT` 文を再解析する必要はありません。カーソルをクローズしてからただちに再オープンした場合、再解析は不要です。

結果セットの中の行は、OPEN 文の実行時には取り出されません。行の取出しには `FETCH` 文を使います。FOR UPDATE カーソルでは、カーソルがオープンされるときに、行はロックされます。

仮パラメータが宣言されている場合は、カーソルに実パラメータを渡さなければなりません。カーソルの仮パラメータは IN パラメータでなければなりません。このため、実パラメータに値を戻すことはできません。実パラメータの値はカーソルをオープンする場合に使われます。仮パラメータと実パラメータのデータ型には、互換性がなければなりません。問合せでは、有効範囲の中で宣言されている他の PL/SQL 変数を参照することもできます。

デフォルト値を受け入れるのであれば、カーソル宣言の中の仮パラメータは、すべて OPEN 文の中で対応する実パラメータを持たなければなりません。デフォルト値で宣言された仮パラメータの詳細は、対応する実パラメータがなくてもかまいません。このような仮パラメータは、OPEN 文の実行時にデフォルト値を取ります。

位置表記法または名前表記法を使って、OPEN 文の実パラメータを、カーソル宣言の仮パラメータに結び付けることができます。詳細は、7-13 ページの「[位置表記法と名前表記法](#)」を参照してください。

カーソルがオープンされている場合は、そのカーソルの名前をカーソル FOR ループで使用できません。

例

次のようなカーソル宣言の場合、

```
CURSOR parts_cur IS SELECT part_num, part_price FROM parts;
```

次の文はカーソルをオープンします。

```
OPEN parts_cur;
```

次のようなカーソル宣言の場合、

```
CURSOR emp_cur(my_ename CHAR, my_comm NUMBER DEFAULT 0)
  IS SELECT * FROM emp WHERE ...
```

次の文はいずれもカーソルをオープンします。

```
OPEN emp_cur('LEE');
OPEN emp_cur('BLAKE', 300);
OPEN emp_cur(employee_name, 150);
```

関連項目

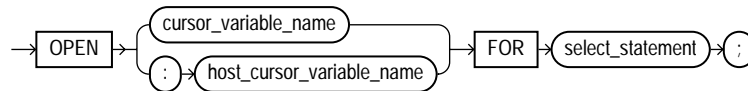
CLOSE 文、カーソル、FETCH 文、LOOP 文

OPEN-FOR 文

OPEN-FOR 文は、カーソル変数に対応付けられている複数行問合せを実行します。また、Oracle が問合せを処理するのに使うリソースを割り振り、結果セットを識別します（結果セットは、問合せの検索条件に合致するすべての行で構成されています）。カーソル変数は、結果セットの中の最初の行の前に配置します。詳細は、5-14 ページの「[カーソル変数の使用](#)」を参照してください。

構文

open_for_statement



キーワードとパラメータの説明

cursor_variable_name

現行の有効範囲のうちそれより前に宣言されているカーソル変数（またはパラメータ）です。

host_cursor_variable_name

PL/SQL ホスト環境で事前に宣言され、バインド変数として PL/SQL に渡されるカーソル変数を識別します。ホスト・カーソル変数のデータ型は、PL/SQL カーソル変数の戻り型と互換性があります。ホスト変数には、接頭辞としてコロンを付けなければなりません。

select_statement

cursor_variable と対応付けられた問合せです。一連の値を戻します。問合せでは、バインド変数、PL/SQL 変数、パラメータ、ファンクションを参照できます。ただし、FOR UPDATE にはできません。select_statement の構文は、11-141 ページの「[SELECT INTO 文](#)」で定義されている select_into_statement の構文に似ていますが、select_statement では INTO 句は使用できません。

使用方法

OCI や Pro*C プログラムなどの PL/SQL ホスト環境で、カーソル変数を宣言できます。ホスト・カーソル変数をオープンするには、バインド変数として無名 PL/SQL ブロックに渡します。OPEN-FOR 文をグループにまとめることによって、ネットワーク・トラフィックを削減できます。たとえば、次の PL/SQL ブロックは、1 回の往復で 5 つのカーソル変数をオープンしています。

```
/* anonymous PL/SQL block in host environment */
BEGIN
  OPEN :emp_cv FOR SELECT * FROM emp;
  OPEN :dept_cv FOR SELECT * FROM dept;
  OPEN :grade_cv FOR SELECT * FROM salgrade;
  OPEN :pay_cv FOR SELECT * FROM payroll;
  OPEN :ins_cv FOR SELECT * FROM insurance;
END;
```

その他の OPEN-FOR 文は、異なる複数の問合せ用に同じカーソル変数をオープンできます。カーソル変数を再オープンする場合、その前にクローズする必要はありません。別の問合せ用にカーソル変数を再オープンすると、前の問合せは失われます。

カーソルとは異なり、カーソル変数はパラメータをとりません。ただし、カーソル変数にはパラメータだけでなく問合せ全体を渡すことができるので、柔軟性があります。

カーソル変数は、それを仮パラメータの 1 つとして宣言するストアド・プロシージャをコールすることにより、PL/SQL に渡せます。ただし、別のサーバー上にあるリモート・サブプログラムは、カーソル変数の値を受け入れることができません。したがって、リモート・プロシージャ・コール (RPC) を使って、カーソル変数はオープンできません。

カーソル変数を、そのカーソル変数をオープンするサブプログラムの仮パラメータとして宣言する場合は、IN OUT モードを指定しなければなりません。そうすることで、サブプログラムはコール元にオープン・カーソルを渡すことができます。

例

次の Pro*C の例では、ホスト・カーソル変数と選択子を PL/SQL ブロックに渡すことで、選択した問合せ用のカーソル変数をオープンしています。

```
EXEC SQL BEGIN DECLARE SECTION;
...
/* Declare host cursor variable. */
SQL_CURSOR generic_cv;
int choice;
EXEC SQL END DECLARE SECTION;
...
/* Initialize host cursor variable. */
EXEC SQL ALLOCATE :generic_cv;
...
/* Pass host cursor variable and selector to PL/SQL block. */
```



```

EXEC SQL EXECUTE
BEGIN
  IF :choice = 1 THEN
    OPEN :generic_cv FOR SELECT * FROM emp;
  ELSIF :choice = 2 THEN
    OPEN :generic_cv FOR SELECT * FROM dept;
  ELSIF :choice = 3 THEN
    OPEN :generic_cv FOR SELECT * FROM salgrade;
  END IF;
END;
END-EXEC;

```

データ検索を集中的に実行するために、ストアド・プロシージャの中で型互換性のある問合せをグループにまとめることができます。次のパッケージ・プロシージャは、選択された問合せ用にカーソル変数 emp_cv をオープンします。

```

CREATE PACKAGE emp_data AS
  TYPE GenericCurTyp IS REF CURSOR;
  TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp,
                        choice IN NUMBER);
END emp_data;

CREATE PACKAGE BODY emp_data AS
  PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp,
                        choice IN NUMBER) IS
  BEGIN
    IF choice = 1 THEN
      OPEN emp_cv FOR SELECT * FROM emp WHERE comm IS NOT NULL;
    ELSIF choice = 2 THEN
      OPEN emp_cv FOR SELECT * FROM emp WHERE sal > 2500;
    ELSIF choice = 3 THEN
      OPEN emp_cv FOR SELECT * FROM emp WHERE deptno = 20;
    END IF;
  END open_emp_cv;
END emp_data;

```

さらに柔軟性を高めるために、次のように、異なる戻り値の型を指定した問合せを実行するストアド・プロシージャにカーソル変数を渡すことができます。

```

CREATE PACKAGE BODY emp_data AS
  PROCEDURE open_cv (generic_cv IN OUT GenericCurTyp,
                    choice IN NUMBER) IS
  BEGIN
    IF choice = 1 THEN
      OPEN generic_cv FOR SELECT * FROM emp;
    ELSIF choice = 2 THEN
      OPEN generic_cv FOR SELECT * FROM dept;

```

OPEN-FOR 文

```
        ELSIF choice = 3 THEN
            OPEN generic_cv FOR SELECT * FROM salgrade;
        END IF;
    END open_cv;
END emp_data;
```

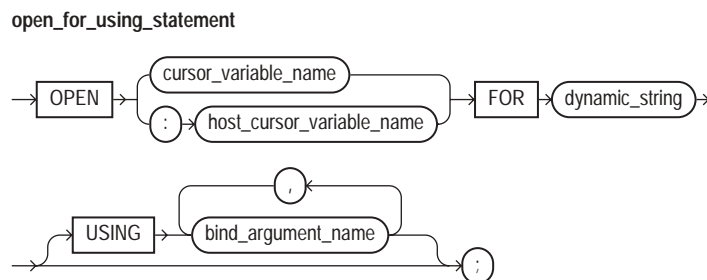
関連項目

CLOSE 文、カーソル変数、FETCH 文、LOOP 文

OPEN-FOR-USING 文

OPEN-FOR-USING 文はカーソル変数を複数行の問合せと対応付け、問合せを実行し、結果を識別してカーソルを結果セットの最初の行に配置してから、%ROWCOUNT によって保持される処理行カウントをゼロに設定します。詳細は、[第 10 章の「システム固有の動的 SQL」](#)を参照してください。

構文



キーワードとパラメータの説明

dynamic_string

複数行の SELECT 文を表す文字列リテラル、変数、または式です。

cursor_variable_name

現行の有効範囲内でそれより前に宣言されている、弱い型定義のカーソル変数（戻り型を持たない）です。

host_cursor_variable_name

PL/SQL ホスト環境で宣言され、バインド変数として PL/SQL に渡されるカーソル変数を識別します。ホスト・カーソル変数のデータ型は、PL/SQL カーソル変数の戻り型と互換性があります。ホスト変数には、接頭辞としてコロンをつけなければなりません。

USING ...

バインド引数のリストを指定するオプションの句です。実行時に、USING 句のバインド引数は動的 SELECT 文内の対応するプレースホルダを置き換えます。

bind_argument_name

動的 SELECT 文に渡される値を持つ式を識別します。

使用方法

動的な複数行の問合せを処理するには、OPEN-FOR-USING 文、FETCH 文、CLOSE 文の、3つの文を使用します。まず、OPEN-FOR 文でカーソル変数を複数行問合せ用にオープンします。次に、FETCH 文で結果セットから一度に 1 行ずつ行を取り出します。すべての行が処理されたら、CLOSE 文でカーソル変数をクローズします。(カーソル変数の詳細は、5-14 ページの「[カーソル変数の使用](#)」を参照してください。)

動的文字列には、任意の複数行 SELECT 文(終了記号を持たない)を含むことができます。また、バインド引数のプレースホルダも含むことができます。しかし、バインド引数を使用してスキーマ・オブジェクトの名前を動的 SQL 文に渡すことはできません。正しい用法は、10-10 ページの「[スキーマ・オブジェクトの名前を渡す](#)」を参照してください。

動的文字列内のすべてのプレースホルダは、USING 句内のバインド引数に対応付ける必要があります。USING 句内では数値リテラル、文字リテラル、および文字列リテラルは使用できますが、ブール・リテラル(TRUE、FALSE、NULL)は使用できません。動的文字列に NULL を渡すには、代替方法を使う必要があります。10-12 ページの「[NULL を渡す](#)」を参照してください。

カーソル変数がオープンしている場合だけ、問合せの中のバインド引数が評価されます。このため、異なるバインド値を使用してカーソルから取り出すには、新しい値に設定されたバインド引数でカーソル変数を再オープンする必要があります。

動的 SQL はすべての SQL データ型をサポートしています。たとえば、バインド引数をコレクション、LOB、オブジェクト型のインスタンス、および ref とすることができます。通常、動的 SQL は PL/SQL 固有の型をサポートしていません。たとえばバインド引数をブールまたは索引付き表にできません。

例

次の例では、カーソル変数を宣言し、それをデータベース表 `emp` から行を戻す動的 `SELECT` 文と関連付けます。

```
DECLARE
  TYPE EmpCurTyp IS REF CURSOR; -- define weak REF CURSOR type
  emp_cv   EmpCurTyp; -- declare cursor variable
  my_ename VARCHAR2(15);
  my_sal   NUMBER := 1000;
BEGIN
  OPEN emp_cv FOR -- open cursor variable
    'SELECT' ename, sal FROM emp WHERE sal > :s' USING my_sal;
  ...
END;
```

関連項目

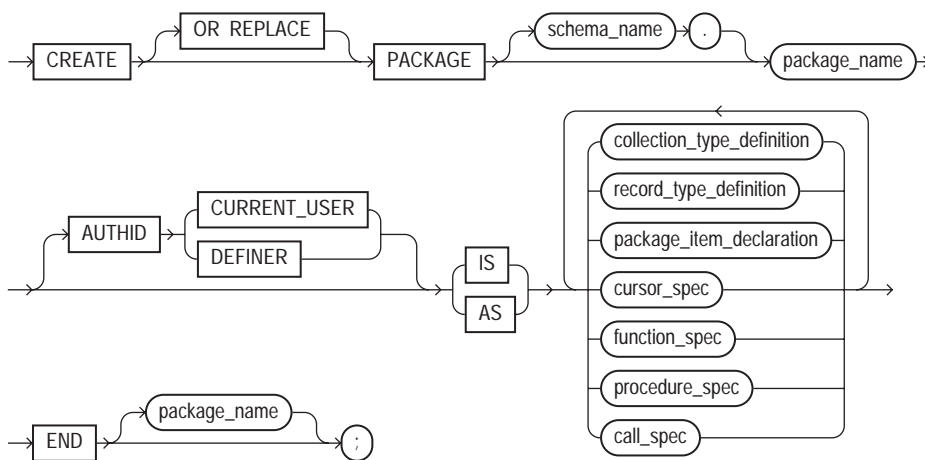
EXECUTE IMMEDIATE 文

パッケージ

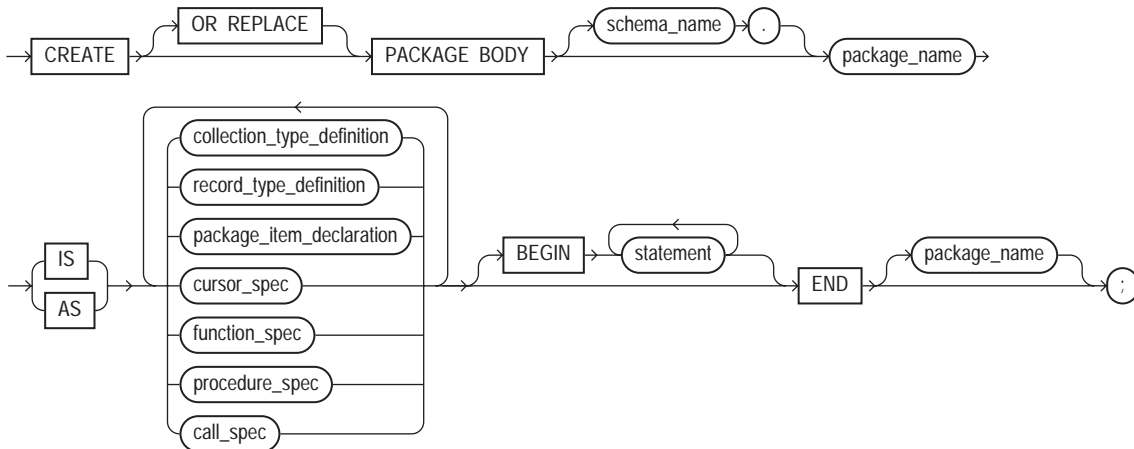
パッケージとは、論理的に関連する PL/SQL の型および項目、サブプログラムをグループにまとめたスキーマ・オブジェクトのことです。パッケージには、仕様部と本体の 2 つの部分があります。詳細は、[第 8 章の「パッケージ」](#)を参照してください。

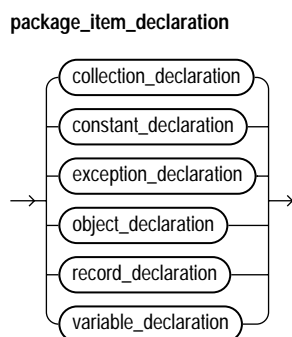
構文

package_declaration | package_spec



package_body





キーワードとパラメータの説明

package_name

パッケージを識別します。命名規則の詳細は、2-4 ページの「[識別子](#)」を参照してください。

AUTHID 句

すべてのパッケージ・サブプログラムがその定義者（デフォルト）と実行者のどちらの権限で実行するか、およびスキーマ・オブジェクトへの未修飾の参照が定義者と実行者のどちらのスキーマで解決されるかを決定します。詳細は、7-29 ページの「[実行者権限と定義者権限](#)」を参照してください。

collection_declaration

現在の有効範囲のうちこれより前の部分で宣言されている、ネストした表、索引付き表、または varray を指定します。collection_declaration の構文は、11-21 ページの「[コレクション](#)」を参照してください。

constant_declaration

定数の宣言です。constant_declaration の構文は、11-29 ページの「[定数と変数](#)」を参照してください。

exception_declaration

例外の宣言です。exception_declaration の構文は、11-54 ページの「[例外](#)」を参照してください。

object_declaration

現在の有効範囲のうちこれより前の部分で宣言されているオブジェクト（オブジェクト型のインスタンス）を指定します。object_declaration の構文は、11-102 ページの「[オブジェクト型](#)」を参照してください。

record_declaration

ユーザー定義のレコードの宣言です。record_declaration の構文は、11-130 ページの「[レコード](#)」を参照してください。

variable_declaration

変数の宣言です。variable_declaration の構文は、11-29 ページの「[定数と変数](#)」を参照してください。

cursor_spec

明示カーソルへのインタフェースの宣言です。cursor_spec の構文は、11-44 ページの「[カーソル](#)」を参照してください。

function_spec

ファンクションへのインタフェースの宣言です。function_spec の構文は、11-78 ページの「[ファンクション](#)」を参照してください。

procedure_spec

プロシージャへのインタフェースの宣言です。procedure_spec の構文は、11-123 ページの「[プロシージャ](#)」を参照してください。

call spec

Oracle データ・ディクショナリ内の外部 C ファンクションまたは Java メソッドを発行します。これは、対応する SQL に名前、パラメータ型および戻り型をマップすることによって、ルーチンを実行します。詳細は、『Oracle8i Java ストアド・プロシージャ開発者ガイド』または『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

cursor_body

明示カーソルの基盤となるインプリメンテーションの定義です。cursor_body の構文は、11-44 ページの「[カーソル](#)」を参照してください。

function_body

関クションの基盤となるインプリメンテーションの定義です。function_body の構文は、11-78 ページの「[ファンクション](#)」を参照してください。

procedure_body

プロシージャの基盤となるインプリメンテーションの定義です。procedure_body の構文は、11-123 ページの「[プロシージャ](#)」を参照してください。

使用方法

パッケージは、PL/SQL ブロックまたはサブプログラムの中では定義できません。しかし、PL/SQL をサポートする Oracle Tools を使うと、パッケージを作り、それを Oracle データベース内に格納できます。CREATE PACKAGE および CREATE PACKAGE BODY 文は、Oracle プリコンパイラか OCI ホスト・プログラムから、または対話的に SQL*Plus から発行できます。

通常、パッケージには仕様部と本体があります。仕様部はアプリケーションへのインタフェースです。ここでは、使用できる型および変数、定数、例外、カーソル、サブプログラムなどを宣言します。本体ではカーソルとサブプログラムを完全に定義し、仕様をインプリメントします。

下位のインプリメンテーション（定義）を持つのは、サブプログラムとカーソルだけです。したがって、仕様部で宣言されているのが型、定数、変数、例外、およびコール仕様部だけならばパッケージ本体は不要です。ただしその場合でも、次のようにパッケージ本体を使って、仕様部で宣言した項目を初期化できます。

```
CREATE PACKAGE emp_actions AS
...
    number_hired INTEGER;
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
BEGIN
    number_hired := 0;
END emp_actions;
```

仕様部は本体がなくてもコーディングし、コンパイルできます。仕様部のコンパイルが終了すると、そのパッケージを参照するストアド・サブプログラムもコンパイルできます。アプリケーション作成の最終段階になるまで、パッケージ本体を完全に定義する必要はありません。さらにパッケージ本体は、パッケージ本体へのインタフェース（パッケージの仕様部）を変更せずに、デバッグ、拡張または置換ができます。つまり、コールする側のプログラムを再コンパイルする必要はありません。

パッケージ仕様部で宣言したカーソルとサブプログラムは、パッケージ本体で定義しなければなりません。パッケージ仕様部で宣言したその他のプログラム項目は、パッケージ本体で再宣言できません。

サブプログラムの仕様部と本体を一致させるために、PL/SQL は、それらのヘッダーをトークンごとに比較します。このため、空白を除いて、ヘッダーは一語一語が一致していなければなりません。そうでない場合、PL/SQL により例外が呼び出されます。

関連項目

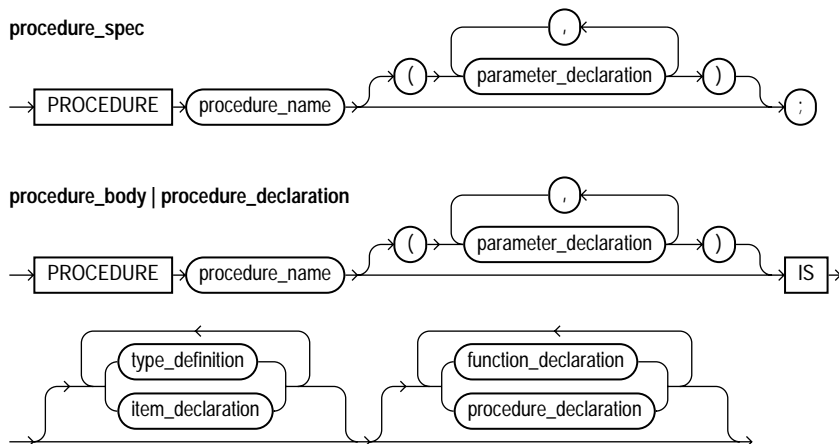
コレクション、カーソル、例外、ファンクション、プロシージャ、レコード

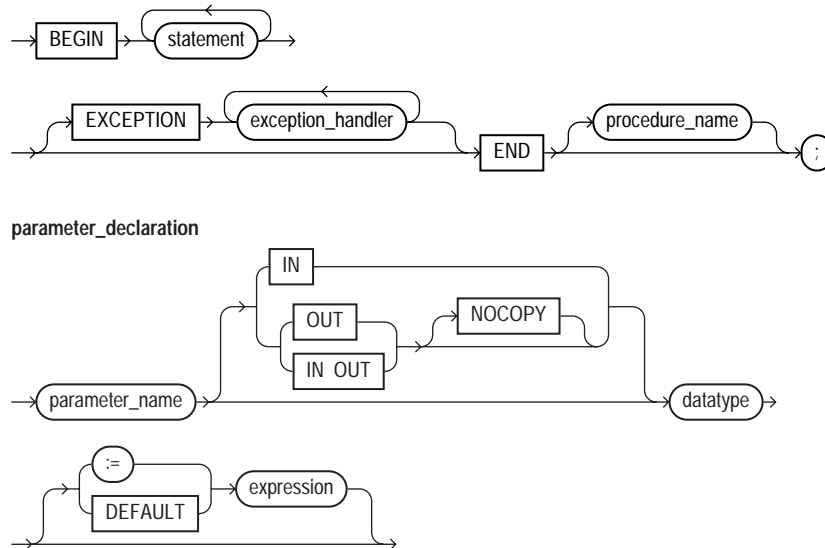
プロシージャ

プロシージャとは、パラメータを指定して起動できるサブプログラムのことです。一般に、プロシージャはアクションを実行するために使います。プロシージャには、仕様部と本体の2つの部分があります。仕様部は、キーワード `PROCEDURE` で始め、プロシージャ名またはパラメータ・リストで終わります。パラメータ宣言はオプションです。パラメータを取らないプロシージャではカッコを書きません。プロシージャの本体は、キーワード `IS` で始め、キーワード `END` で終わります。END の後には、オプションとしてプロシージャ名を続けることができます。

プロシージャ本体には、宣言部（オプション）、実行部、例外処理部（オプション）という3つの部分があります。宣言部には、型およびカーソル、定数、変数、例外と、サブプログラムが含まれます。これらの項目はローカルで、プロシージャを終了すると消去されます。実行部には、値の代入、実行の制御および Oracle データの操作を実行する文があります。例外処理部には、実行の途中で呼び出された例外を処理する例外ハンドラを入れます。詳細は、7-3 ページの「[プロシージャ](#)」を参照してください。

構文





キーワードとパラメータの説明

procedure_name

ユーザー定義のプロシージャです。

parameter_name

仮パラメータを識別します。仮パラメータとは、プロシージャの仕様部で宣言され、プロシージャ本体の中で参照される変数のことです。

IN、OUT、IN OUT

これらのパラメータ・モードは、仮パラメータの動作を定義します。IN パラメータは、コールされるサブプログラムに値を渡すために使います。OUT パラメータは、サブプログラムのコール側に値を戻すために使います。IN OUT パラメータを使うと、コールされる側のサブプログラムに初期値を渡して、コールした側に更新された値を戻すことができます。

NOCOPY

コンパイラ・ヒント（ディレクティブではない）です。これによって、PL/SQL コンパイラは OUT および IN OUT パラメータを、デフォルトの値方式ではなく、参照方式で渡すことができます。詳細は、7-17 ページの「[NOCOPY コンパイラ・ヒント](#)」を参照してください。

datatype

これは、型指定子です。datatype の構文は、11-29 ページの「[定数と変数](#)」を参照してください。

:= | DEFAULT

この演算子またはキーワードを使うと、IN パラメータをデフォルト値に初期化できます。

expression

変数、定数、リテラル、演算子、ファンクション・コールの任意の組合せです。最も単純な式は、1 個の変数で構成されています。宣言が PL/SQL コンパイラによって処理されるとき、expression の値がパラメータに代入されます。その値とパラメータは、データ型に互換性がなければなりません。

type_definition

これは、ユーザー定義のデータ型を指定します。type_definition の構文は、11-7 ページの「[ブロック](#)」を参照してください。

item_declaration

これは、プログラム・オブジェクトを宣言します。item_declaration の構文は、11-7 ページの「[ブロック](#)」を参照してください。

function_declaration

ファンクションの宣言です。function_declaration の構文は、11-78 ページの「[ファンクション](#)」を参照してください。

procedure_declaration

プロシージャの宣言です。procedure_declaration の構文は、11-123 ページの「[プロシージャ](#)」を参照してください。

exception_handler

例外ハンドラです。例外が呼び出されると、その例外に結び付けられた一連の文を実行します。exception_handler の構文は、11-54 ページの「[例外](#)」を参照してください。

使用方法

プロシージャは PL/SQL 文としてコールされます。たとえば、次のようにプロシージャ `raise_salary` をコールできます。

```
raise_salary(emp_num, amount);
```

プロシージャの中では、IN パラメータは定数のように取り扱われます。したがって、値を代入できません。OUT パラメータはローカル変数のように取り扱われます。したがって、値を変更して参照できます。IN OUT パラメータは初期化された変数のように取り扱われます。したがって、値を代入したり、その値を他の変数に代入したりできます。パラメータ・モードの概要は、7-16 ページの表 7-1 を参照してください。

OUT パラメータおよび IN OUT パラメータとは異なり、IN パラメータはデフォルト値に初期化できます。詳細は、7-19 ページの「[パラメータのデフォルト値](#)」を参照してください。

プロシージャを終了する前に、すべての OUT 仮パラメータに明示的に値を代入してください。そうしないと、対応する実パラメータの値は不定になります。実行に成功して終了した場合、PL/SQL は実パラメータに値を代入します。しかし、未処理例外が発生して実行が終了すると、PL/SQL は実パラメータに値を代入しません。

プロシージャの仕様部と本体を合わせて 1 つの単位として作成できます。また、プロシージャの仕様部と本体を別々にすることもできます。このように、プロシージャをパッケージに入れると、インプリメンテーション上の細部を隠ぺいできます。パッケージ仕様部でプロシージャ仕様部を宣言せずに、パッケージ本体でプロシージャを定義できます。ただし、このようなプロシージャはパッケージの内側からしかコールできません。

プロシージャは、PL/SQL をサポートするすべての Oracle Tools で定義できます。しかし、一般的な用途に使うには、CREATE 文でプロシージャを作成し、Oracle データベースに格納しておかなくてはなりません。CREATE PROCEDURE 文は、SQL*Plus から対話式で実行できます。

プロシージャの実行部には、少なくとも 1 つの文が存在しなければなりません。NULL 文はこの条件を満たします。

例

次のプロシージャは銀行口座から出金します。

```
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
    old_balance REAL;
    new_balance REAL;
    overdrawn    EXCEPTION;
BEGIN
    SELECT bal INTO old_balance FROM accts WHERE acctno = acct_id;
    new_balance := old_balance - amount;
    IF new_balance < 0 THEN
        RAISE overdrawn;
    ELSE
```

```

        UPDATE accts SET bal = new_balance WHERE acctno = acct_id;
    END IF;
EXCEPTION
    WHEN overdrawn THEN
        ...
END debit_account;
```

次の例では、名前表記法を使ってプロシージャをコールしています。

```
debit_account (amount => 500, acct_id => 10261);
```

関連項目

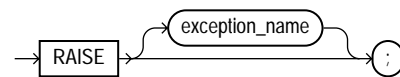
コレクション、ファンクション、パッケージ、レコード

RAISE 文

RAISE 文は、PL/SQL ブロックまたはサブプログラムの通常の実行を停止させ、適切な例外ハンドラに制御を移します。通常、事前定義の例外は実行時システムによって暗黙的に呼び出されます。しかし、事前定義の例外を RAISE 文で呼び出すこともできます。ユーザー定義の例外は RAISE 文によって明示的に呼び出さなければなりません。詳細は、6-7 ページの「[ユーザー定義の例外](#)」を参照してください。

構文

raise_statement



キーワードとパラメータの説明

exception_name

事前定義の例外またはユーザー定義の例外を識別します。事前定義の例外のリストは、6-4 ページの「[事前定義の例外](#)」を参照してください。

使用方法

PL/SQL のブロックとサブプログラムから RAISE 文で例外を呼び出すのは、エラーのために処理の続行が不可能になった場合、または望ましくない場合だけにしてください。指定した例外に対する RAISE 文は、その例外の有効範囲内であれば任意の場所にコーディングできます。

例外が呼び出されたとき、PL/SQL がその例外のハンドラを現在のブロックで発見できなければ、例外は遷移します。すなわち、例外は外側のブロックで再生され、ハンドラが見つかるまで、または検索するブロックがなくなるまで、1 つずつ外側のブロックに進んでいきます。検索するブロックがなくなった場合、PL/SQL はホスト環境に「未処理例外 (unhandled exception)」エラーを戻します。

RAISE 文で例外名を省略すると、現在の例外が再び呼び出されます。例外名の省略は、例外ハンドラの中でしか許されていません。パラメータのない RAISE 文が例外ハンドラの中で実行された場合、最初に検索されるブロックは、現行のブロックではなく囲みブロックです。

例

次の例では、在庫部品が在庫切れになった場合に例外を呼び出します。

```
IF quantity_on_hand = 0 THEN
  RAISE out_of_stock;
END IF;
```

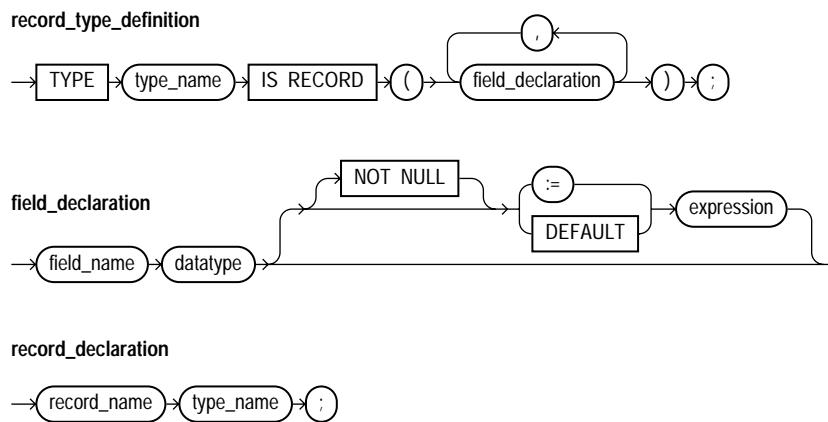
関連項目

例外

レコード

レコードは、RECORD 型の項目です。レコードには、様々な型のデータ値を格納できる、他と重複しない名前のフィールドがあります。このため、レコードによって、関連してはいるが異なるデータを 1 つの論理単位として扱うことができます。詳細は、4-35 ページの「[レコードとは?](#)」を参照してください。

構文



キーワードとパラメータの説明

record_type_name

それ以降のレコードの宣言で使うユーザー定義の型指定子を識別します。

NOT NULL

フィールドに NULL を代入できないようにするための制約です。実行時に、NOT NULL として定義されたフィールドに NULL を代入しようとすると、事前定義の例外 VALUE_ERROR が呼び出されます。NOT NULL 制約の後には初期化句が続かなければなりません。

datatype

これは、型指定子です。datatype の構文は、11-29 ページの「[定数と変数](#)」を参照してください。

:= | DEFAULT

この演算子またはキーワードを使うと、フィールドをデフォルト値に初期化できます。

expression

変数、定数、リテラル、演算子、ファンクション・コールの任意の組合せです。最も単純な式は、1 個の変数で構成されています。expression の構文は、11-62 ページの「式」を参照してください。宣言が PL/SQL コンパイラによって処理されるとき、expression の値がフィールドに代入されます。その値とフィールドは、データ型に互換性がなければなりません。

使用方法

RECORD 型定義とユーザー定義レコードの宣言は、任意のブロック、サブプログラム、またはパッケージの宣言部でできます。また、次の例に示すように、レコードを宣言の中で初期化できます。

```
DECLARE
    TYPE TimeTyp IS RECORD(
        seconds SMALLINT := 0,
        minutes SMALLINT := 0,
        hours    SMALLINT := 0);
```

次の例では、%TYPE 属性を使って、フィールドのデータ型を指定しています。また、この例では、フィールド宣言に NOT NULL 制約を加えて、フィールドに NULL が割り当てられないようにしています。

```
DECLARE
    TYPE DeptRecTyp IS RECORD(
        deptno NUMBER(2) NOT NULL,
        dname  dept.dname%TYPE,
        loc    dept.loc%TYPE);
    dept_rec DeptRecTyp;
```

レコード中の個々のフィールドを参照する場合は、ドット表記法を使います。たとえば、レコード dept_rec のフィールド dname に値を代入する場合は、次のようにします。

```
dept_rec.dname := 'PURCHASING';
```

レコード中の個々のフィールドに別々に値を代入するかわりに、すべてのフィールドに値を一度に代入できます。これには 2 つの方法があります。第 1 の方法として、2 つのユーザー定義レコードのデータ型が同じであれば、一方のレコードをもう一方のレコードに代入できます。(正確に一致するフィールドが含まれているだけでは不十分です。) フィールドの数と順序が同じで、対応するフィールドのデータ型に互換性があれば、%ROWTYPE レコードをユーザー定義のレコードに代入できます。

第2の方法として、SELECT 文または FETCH 文を使って列の値をフェッチし、レコードに代入できます。選択リストの列が、レコード中のフィールドと同じ順序で並ぶようにしてください。

ネストされたレコードを宣言し、参照できます。つまり、レコードは他のレコードの構成要素になることができます。次に例を示します。

```
DECLARE
  TYPE TimeTyp IS RECORD(
    minutes SMALLINT,
    hours   SMALLINT);
  TYPE MeetingTyp IS RECORD(
    day      DATE,
    time_of TimeTyp,  -- nested record
    place   CHAR(20),
    purpose CHAR(50));
  TYPE PartyTyp IS RECORD(
    day      DATE,
    time_of TimeTyp,  -- nested record
    place   CHAR(15));
  meeting MeetingTyp;
  seminar MeetingTyp;
  party   PartyTyp;
```

次の例では、ネストされたレコードを、同じデータ型を持つ別のネストされたレコードに代入しています。

```
seminar.time_of := meeting.time_of;
```

このような代入は、親レコードが異なるデータ型を持っている場合でもできます。

ユーザー定義のレコードは、通常の有効範囲の規則とインスタンス生成の規則に従います。パッケージでは、パッケージを最初に参照するときにインスタンス生成され、アプリケーションを終了するかデータベース・セッションを終了すると消去されます。ブロックまたはサブプログラムでは、ブロックまたはサブプログラムに入るときにインスタンス生成され、ブロックまたはサブプログラムを終了すると消去されます。

スカラー変数と同様に、ユーザー定義のレコードもプロシージャやファンクションの仮パラメータとして宣言できます。スカラー・パラメータに適用されるのと同じ制限が、ユーザー定義のレコードにも適用されます。

ファンクション仕様部の RETURN 句の中に RECORD 型を指定できます。こうすると、ファンクションは同じ型のユーザー定義のレコードを戻します。ユーザー定義のレコードを戻すファンクションをコールする場合、次の構文を使ってレコード内のフィールドを参照します。

```
function_name(parameter_list).field_name
```

ネストしたフィールドを参照するには、次の構文を使用します。

```
function_name(parameter_list).field_name.nested_field_name
```

ファンクションがパラメータをとらない場合は、空のパラメータ・リストをコーディングします。次に構文を示します。

```
function_name().field_name
```

例

次の例では、DeptRecTyp という名前の RECORD 型を定義し、dept_rec という名前のレコードを宣言した後、行の値を選択してレコードに入れています。

```
DECLARE
    TYPE DeptRecTyp IS RECORD(
        deptno NUMBER(2),
        dname  CHAR(14),
        loc    CHAR(13));
    dept_rec DeptRecTyp;
    ...
BEGIN
    SELECT deptno, dname, loc INTO dept_rec FROM dept
    WHERE deptno = 20;
```

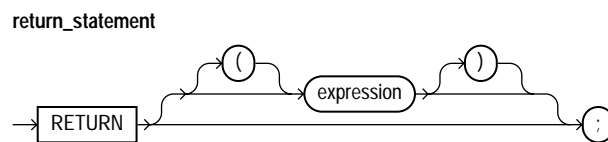
関連項目

代入文、コレクション、ファンクション、プロシージャ

RETURN 文

RETURN 文は、サブプログラムの実行を即座に完了させ、コールした側に制御を戻します。その後は、サブプログラム・コールの直後の文から、実行が再開されます。ファンクションの中の RETURN 文は、ファンクション識別子を結果の値に代入します。詳細は、7-8 ページの「[RETURN 文](#)」を参照してください。

構文



キーワードとパラメータの説明

expression

変数、定数、リテラル、演算子、ファンクション・コールの任意の組合せです。最も単純な式は、1 個の変数で構成されています。RETURN 文を実行すると、expression の値がファンクション識別子に代入されます。

使用方法

RETURN 文を、ファンクションの仕様部の中で結果値のデータ型を指定する RETURN 句と混同しないようにしてください。

サブプログラムは複数の RETURN 文を持つことができます。そのいずれも、最後の文である必要はありません。どの RETURN 文を実行しても、サブプログラムは即座に終了します。しかし、サブプログラムに複数の終了点を作るのはプログラミングの習慣として好ましくありません。

プロシージャでは、RETURN 文に式を含めることはできません。RETURN 文には、プロシージャ本来の終了地点に達する前に、コールした側に制御を戻す役割しかありません。

ただし、ファンクションにおいて、RETURN 文には、RETURN 文の実行時に評価される式が含まれていなければなりません。結果として得られる値がファンクション識別子に代入されます。そのため、ファンクションは少なくとも 1 つの RETURN 文を持っていなければなりません。RETURN 文が 1 つもない場合、PL/SQL は実行時に事前定義の例外 PROGRAM_ERROR をコールします。

RETURN 文を無名ブロックで使って、そのブロック（およびすべての囲みブロック）を即座に終了させることもできますが、RETURN 文は式を含むことはできません。

例

次の例のファンクション `balance` は、指定された銀行口座の残高を戻します。

```
FUNCTION balance (acct_id INTEGER) RETURN REAL IS
    acct_bal REAL;
BEGIN
    SELECT bal INTO acct_bal FROM accts WHERE acctno = acct_id;
    RETURN acct_bal;
END balance;
```

関連項目

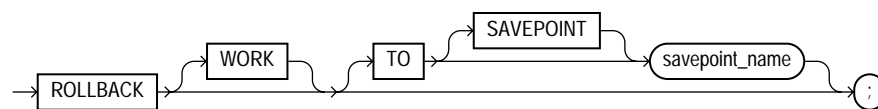
ファンクション、プロシージャ

ROLLBACK 文

ROLLBACK 文は COMMIT 文の逆です。これは、現行のトランザクションでデータベースに加えられたすべての変更または一部の変更を取り消します。詳細は、5-37 ページの「[トランザクション処理](#)」を参照してください。

構文

rollback_statement



キーワードとパラメータの説明

ROLLBACK

パラメータなしの ROLLBACK 文が実行されると、現行のトランザクションでデータベースに加えられた変更がすべて取り消されます。

WORK

このキーワードはオプションで、コードをわかりやすくするという効果しか持ちません。

ROLLBACK TO

この文は、**savepoint_name** で識別されるセーブポイントがマークされた後にデータベースに加えられた変更をすべて取り消します（また、マーク以降に取得されたロックをすべて解放します）。

SAVEPOINT

このキーワードはオプションで、コードをわかりやすくするという効果しか持ちません。

savepoint_name

トランザクション処理の中で現行の位置を識別するためのマークとなる未宣言の識別子です。命名規則の詳細は、2-4 ページの「[識別子](#)」を参照してください。

使用方法

ロールバック先のセーブポイント以降にマークされているセーブポイントはすべて消去されます。ただし、ロールバック先のセーブポイントは消去されません。たとえば、セーブポイントを A、B、C、D の順でマークしている場合、セーブポイント B までロールバックすると、セーブポイント C と D だけが消去されます。

INSERT 文、UPDATE 文、または DELETE 文を実行する前に、暗黙のセーブポイントがマークされます。文の実行が失敗すると、その暗黙のセーブポイントまでロールバックされます。通常は、トランザクション全体ではなく、失敗した SQL 文だけがロールバックされます。しかし、その文が原因で未処理例外が呼び出された場合は、ホスト環境によってロールバックの対象が決まります。

SQL では、FORCE 句はインダウト分散トランザクションを手動でロールバックする句です。ただし、PL/SQL ではこの句はサポートされていません。たとえば、次の文は誤りです。

```
ROLLBACK WORK FORCE '24.37.85'; -- illegal
```

埋込み SQL では、RELEASE オプションは、プログラムに保持されるすべての Oracle リソース（ロックおよびカーソル）を解放し、データベースから切断します。ただし、PL/SQL ではこのオプションはサポートされていません。たとえば、次の文は誤りです。

```
ROLLBACK WORK RELEASE; -- illegal
```

関連項目

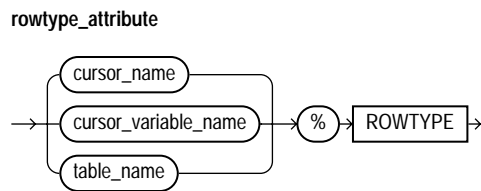
COMMIT 文、SAVEPOINT 文

%ROWTYPE 属性

%ROWTYPE 属性は、データベース表の中の行を表すレコード型を提供します。レコードには、表から選択された行全体、あるいはカーソルまたはカーソル変数で取り出された行全体のデータを格納できます。レコード中のフィールドと、それに対応する行の中の列は、同じ名前とデータ型を持ちます。

%ROWTYPE 属性は、変数宣言の中でデータ型指定子として使えます。%ROWTYPE 属性を使って宣言された変数は、データ型名を使って宣言された変数と同じように扱われます。詳細は、2-30 ページの「[%ROWTYPE の使用方法](#)」を参照してください。

構文



キーワードとパラメータの説明

cursor_name

現在の有効範囲の中で、事前に宣言されている明示カーソルを識別します。

cursor_variable_name

現在の有効範囲の中で事前に制限されている、強い型定義をされた PL/SQL カーソル変数を識別します。

table_name

宣言が PL/SQL コンパイラによって処理されるときにアクセスできないデータベースの表（またはビュー）を識別します。

使用方法

%ROWTYPE 属性を使用すると、データベース表のデータ行のような構造を持つレコードを宣言できます。レコード中のフィールドを参照するには、ドット表記法を使用します。たとえば、フィールド deptno は次のように参照できます。

```
IF emp_rec.deptno = 20 THEN ...
```

次の例に示すように、式の値を特定のフィールドに代入できます。

```
emp_rec.sal := average * 1.15;
```

レコードのすべてのフィールドに一度に値を代入する方法は2つあります。1番目の方法として、PL/SQLでは、2つのレコードの宣言が同じ表またはカーソルを参照している場合に、その2つのレコード全体の間での一括代入ができます。2番目の方法では、SELECT 文か FETCH 文を使って、レコードに列値のリストを代入します。列名の順番は、CREATE TABLE 文または CREATE VIEW 文で定義された順番でなければなりません。カーソルによって %ROWTYPE 属性を使って取り出された選択項目は、単純名を持たなければなりません。また、選択項目が式の場合は別名を持たなければなりません。

例

次の例では %ROWTYPE を使って2つのレコードを宣言しています。1つめのレコードには表 emp から選択された行が格納されます。2つめのレコードには、カーソル c1 で取り出された行が格納されます。

```
DECLARE
    emp_rec    emp%ROWTYPE;
    CURSOR c1 IS SELECT deptno, dname, loc FROM dept;
    dept_rec   c1%ROWTYPE;
```

次の例では、表 emp から選択した行を %ROWTYPE 属性のレコードに入れています。

```
DECLARE
    emp_rec    emp%ROWTYPE;
    ...
BEGIN
    SELECT * INTO emp_rec FROM emp WHERE empno = my_empno;
    IF (emp_rec.deptno = 20) AND (emp_rec.sal > 2000) THEN
        ...
    END IF;
END;
```

関連項目

定数と変数、カーソル、カーソル変数、FETCH 文

SAVEPOINT 文

SAVEPOINT 文は、トランザクション処理の過程で、現在の位置に名前を付けてマークします。セーブポイントを ROLLBACK TO 文と組み合わせると、トランザクション全体ではなく、トランザクションの一部を取り消すことができます。詳細は、5-37 ページの「[トランザクション処理](#)」を参照してください。

構文

savepoint_statement

→ **SAVEPOINT** (savepoint_name) ;

キーワードとパラメータの説明

savepoint_name

トランザクション処理の中で現行の位置を識別するためのマークとなる未宣言の識別子です。

使用方法

あるセーブポイントまでロールバックすると、そのセーブポイント以降にマークされたセーブポイントはすべて消去されます。ただし、ロールバック先のセーブポイントは消去されません。単に、ロールバックまたはコミットだけをするるとすべてのセーブポイントが消去されます。セーブポイント名は、トランザクション内で再利用できます。再利用すると、セーブポイントはトランザクションの中の古い位置から現在の位置に移動します。

再帰的なサブプログラムの中でセーブポイントをマークすると、再帰しながら進んでいく過程で、各レベルで SAVEPOINT 文の新しいインスタンスが実行されます。ただし、ロールバックできるのは直前にマークされたセーブポイントまでだけです。

INSERT 文、UPDATE 文、または DELETE 文を実行する前に、暗黙のセーブポイントがマークされます。文の実行が失敗すると、その暗黙のセーブポイントまでロールバックされます。通常は、トランザクション全体ではなく、失敗した SQL 文だけがロールバックされます。しかし、その文が原因で未処理例外が呼び出された場合は、ホスト環境によってロールバックの対象が決まります。

関連項目

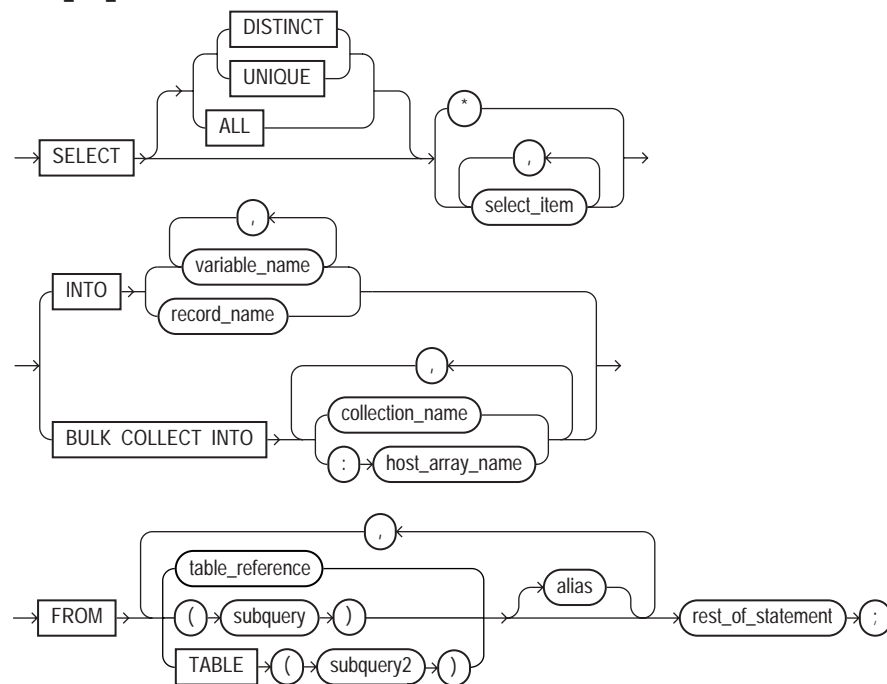
COMMIT 文、ROLLBACK 文

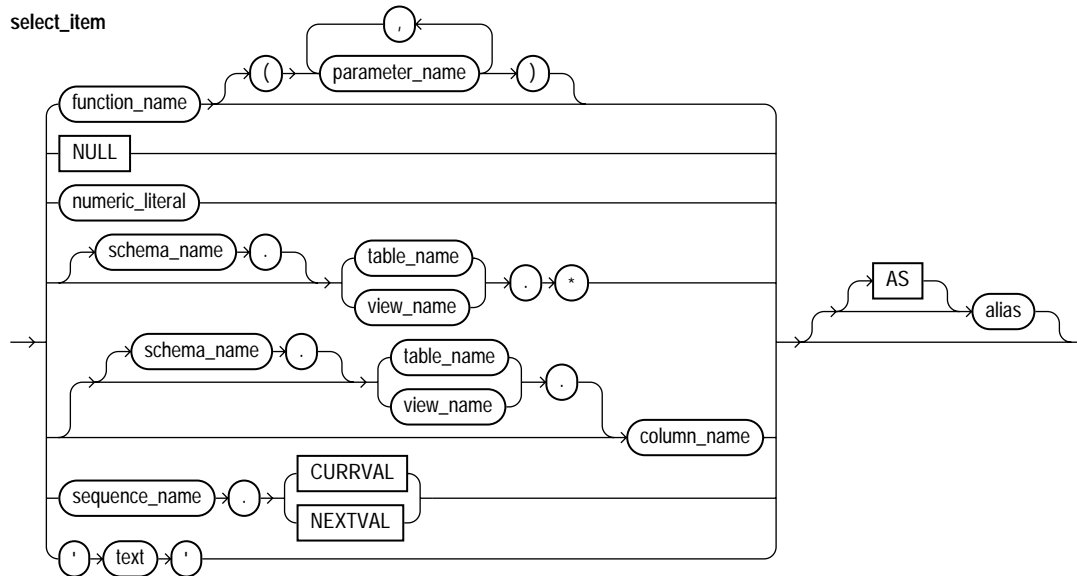
SELECT INTO 文

SELECT INTO 文は、データベースの 1 つ以上の表からデータを取り出して、選択した値を変数またはフィールドに代入します。SELECT 文の詳細は、『Oracle8i SQL リファレンス』を参照してください。

構文

select_into_statement





キーワードとパラメータの説明

select_item

SELECT 文によって戻される値です。この値は、INTO 句の中の対応する変数またはフィールドに代入されます。

BULK COLLECT

この句は、コレクションを PL/SQL エンジンに戻す前にバルク・バインド出力するよう、SQL エンジンに指示を与えます。SQL エンジンは、INTO リスト内で参照されるすべてのコレクションをバルク・バインドします。対応する列には、スカラー値（複合ではない）が格納されている必要があります。詳細は、4-27 ページの「[バルク・バインドの利用](#)」を参照してください。

variable_name

フェッチした select_item 値を格納するための、事前に宣言されたスカラー変数を識別します。問合せが戻す select_item 値に対して、リストの中に、対応する型互換の変数が存在しなければなりません。

record_name

フェッチした行の値を格納する、ユーザー定義のレコードまたは %ROWTYPE のレコードを識別します。問合せが戻す select_item 値に対して、レコードの中に、対応する型互換のフィールドが存在しなければなりません。

collection_name

バルク・フェッチした select_item 値を格納するための、宣言されたコレクションを識別します。select_item ごとに、リストの中に、対応する型互換のコレクションが存在しなければなりません。

host_array_name

一括フェッチした select_item 値を格納するための配列を識別します。この配列は、PL/SQL ホスト環境で宣言され、バインド変数として PL/SQL に渡されます。select_item ごとに、リストの中に、対応する型互換の配列が存在しなければなりません。ホスト配列には、接頭辞としてコロンを付けなければなりません。

table_reference

表またはビューを指定します。指定された表またはビューは、SELECT 文の実行時にアクセスできなければならない、ユーザーが SELECT 権限を持っていなければならない。table_reference の構文は、11-48 ページの「[DELETE 文](#)」を参照してください。

subquery

処理する行の集合を提供する SELECT 文です。構文は select_into_statement の構文と似ていますが、INTO 句は使えません。11-141 ページの「[SELECT INTO 文](#)」を参照してください。

TABLE (subquery2)

TABLE のオペランドは、1 つの列値を戻す SELECT 文です。これはネストした表、またはネストした表として割り当てられる varray である必要があります。演算子 TABLE は、値がスカラー値ではなくコレクションであることを Oracle に通知します。

alias

参照される列、表、またはビューの別名（多くの場合、短縮名）です。

rest_of_statement

SAMPLE 句を除く、SELECT 文の FROM 句に続けることができる任意の構成体です。

使用方法

暗黙的な SQL カーソルとカーソル属性 %NOTFOUND、%FOUND、%ROWCOUNT、および %ISOPEN を使うと、SELECT INTO 文の実行に関する有用な情報にアクセスできます。

SELECT INTO 文を使って変数に値を代入する場合、1 行だけしか戻されません。2 行以上が戻された場合は、次の結果になります。

- PL/SQL によって、事前定義の例外 TOO_MANY_ROWS が呼び出される。
- SQLCODE は、-1422 (Oracle エラー・コード ORA-01422) を戻す。
- SQLERRM は、Oracle エラー・メッセージ「単一行の問合せが複数の行を戻した (single-row query returns more than one row)」を戻す。
- SQL%NOTFOUND は、FALSE になる。
- SQL%FOUND は、TRUE になる。
- SQL%ROWCOUNT は 1 になる。

行が戻されなかった場合、属性の値は次のようになります。

- PL/SQL によって、事前定義の例外 NO_DATA_FOUND が呼び出される。ただし、SELECT 文が AVG や SUM などの SQL 集約関数をコールしていた場合を除く。(SQL 集約関数は必ず値または NULL を戻す。したがって、集約関数をコールする SELECT INTO 文では、NO_DATA_FOUND が呼び出されることはない。)
- SQLCODE は、+100 (Oracle エラー・コード ORA-01403) を戻す。
- SQLERRM は、Oracle エラー・メッセージ「データが見つからない (no data found)」を戻す。
- SQL%NOTFOUND は、TRUE になる。
- SQL%FOUND は、FALSE になる。
- SQL%ROWCOUNT は 0 になる。

例

次の SELECT 文は、データベースの表 emp から従業員の名前、肩書、給与を戻します。

```
SELECT ename, job, sal INTO my_ename, my_job, my_sal FROM emp
WHERE empno = my_empno;
```

関連項目

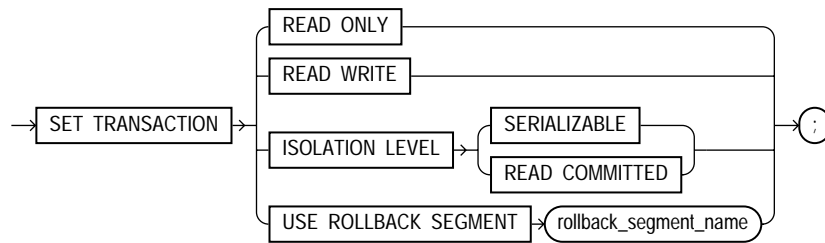
代入文、FETCH 文、%ROWTYPE 属性

SET TRANSACTION 文

SET TRANSACTION 文は、読取り専用または読取り / 書込みのトランザクションを開始するか、分離レベルを設定するか、指定したロールバック・セグメントに現在のトランザクションを代入します。読取り専用トランザクションは、他のユーザーが更新中である 1 つまたは複数の表に対して、複数の問合せを実行する場合に便利です。詳細は、5-42 ページの「[SET TRANSACTION の使用](#)」を参照してください。

構文

set_transaction_statement



キーワードとパラメータの説明

READ ONLY

現行のトランザクションを読取り専用に設定する句です。トランザクションを **READ ONLY** に設定すると、それ以降の問合せからはトランザクションの開始前にコミットされた変更内容しか見えません。**READ ONLY** を使っても、他のユーザーや他のトランザクションには影響がありません。

READ WRITE

現行のトランザクションを読取り / 書込みに設定する句です。**READ WRITE** を使っても、他のユーザーや他のトランザクションには影響がありません。トランザクションでデータ操作文が実行されると、Oracle はトランザクションをロールバック・セグメントに代入します。

ISOLATION LEVEL

この句は、データベースを変更するトランザクションがどのように処理されるかを指定します。**SERIALIZABLE** を指定すると、直列可能なトランザクションが、コミットされていない別のトランザクションですでに変更された表を変更する SQL データ操作文を実行しようとした場合、その文は失敗します。

SERIALIZABLE モードを使用可能にするには、DBA が、Oracle 初期化パラメータ COMPATIBLE を 7.3.0 以上に設定します。

READ COMMITTED を指定すると、トランザクションに含まれる SQL データ操作文が、別のトランザクションによって保持されている行ロックを必要とする場合に、その文は行ロックが解放されるまで待機します。

USE ROLLBACK SEGMENT

この句は、現行のトランザクションを、指定したロールバック・セグメントに代入し、トランザクションを読取り / 書込みに設定します。このパラメータは、同じトランザクションの中で READ ONLY パラメータとともに使用できません。読込み専用のトランザクションは、ロールバック情報を生成しないためです。

使用方法

SET TRANSACTION 文は、トランザクションの最初の SQL 文でなければならず、そのトランザクションで 1 回しか使えません。

例

次の例では、読取り専用トランザクションを確立しています。

```
COMMIT; -- end previous transaction
SET TRANSACTION READ ONLY;
SELECT ... FROM emp WHERE ...
SELECT ... FROM dept WHERE ...
SELECT ... FROM emp WHERE ...
COMMIT; -- end read-only transaction
```

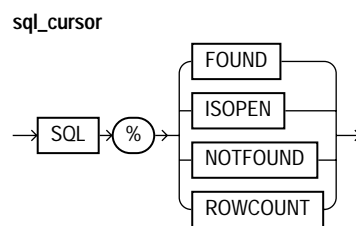
関連項目

COMMIT 文、ROLLBACK 文、SAVEPOINT 文

SQL カーソル

明示的なカーソルと結び付けられていない SQL 文を処理するために、Oracle は暗黙的にカーソルをオープンします。PL/SQL では直前の暗黙カーソルを SQL カーソルとして参照できます。このカーソルには、%FOUND、%ISOPEN、%NOTFOUND、%ROWCOUNT の、4 つの属性があります。これらの属性を使用すると、データ操作文の実行についての情報が得られます。詳細は、5-6 ページの「[カーソル管理](#)」を参照してください。

構文



キーワードとパラメータの説明

SQL

暗黙 SQL カーソルの名前です。

%FOUND

INSERT 文、UPDATE 文、DELETE 文が 1 行以上の行に作用するか、または SELECT INTO 文が 1 行以上の行を戻す場合、この属性の結果は TRUE になります。これ以外の場合、FALSE となります。

%ISOPEN

Oracle は、SQL カーソルに対応付けられた SQL 文の実行を終了すると、このカーソルを自動的にクローズするため、この属性の結果は常に FALSE になります。

%NOTFOUND

この属性は %FOUND とは論理的に反対の意味を持ちます。INSERT 文、UPDATE 文、DELETE 文がどの行にも作用しないか、または SELECT INTO 文がどの行も戻さない場合、この属性の結果は TRUE になります。これ以外の場合、FALSE となります。

%ROWCOUNT

この属性の結果は、INSERT 文、UPDATE 文、DELETE 文の影響を受けた行、または SELECT INTO 文に戻された行の数になります。

使用方法

カーソル属性は、プロシージャ文では使えますが、SQL 文では使えません。Oracle が SQL カーソルを自動的にオープンするまでは、暗黙カーソルの属性の結果は NULL になります。

カーソル属性の値は、常に直前に実行された SQL 文を参照します（その文の場所とは無関係です）。文が別の有効範囲に存在する場合があります。したがって、属性の値を保存して後で使いたい場合は、ブール変数にただちに代入してください。

SELECT INTO 文が行を戻せなかった場合は、次の行で SQL%NOTFOUND をチェックしているかどうかにかかわらず、PL/SQL によって事前定義の例外 NO_DATA_FOUND が呼び出されます。ただし、SQL 集約関数をコールする SELECT INTO 文では、NO_DATA_FOUND が呼び出されることはありません。これは、AVG や SUM などの集約関数は必ず値または NULL を戻すためです。このような場合、SQL%NOTFOUND の結果は FALSE になります。

例

次の例では、更新される行がない場合に、%NOTFOUND を使って行を挿入しています。

```
UPDATE emp SET sal = sal * 1.05 WHERE empno = my_empno;
IF SQL%NOTFOUND THEN
    INSERT INTO emp VALUES (my_empno, my_ename, ...);
END IF;
```

次の例では、100 を超える行数が削除された場合に、%ROWCOUNT で例外を呼び出しています。

```
DELETE FROM parts WHERE status = 'OBSOLETE';
IF SQL%ROWCOUNT > 100 THEN -- more than 100 rows were deleted
    RAISE large_deletion;
END IF;
```

関連項目

カーソル、カーソル属性

SQLCODE ファンクション

ファンクション SQLCODE は、直前に呼び出された例外に対応付けられている番号コードを戻します。SQLCODE は例外ハンドラでしか意味を持ちません。ハンドラの外側では、SQLCODE は常に 0 を戻します。

内部例外の場合、SQLCODE は対応付けられている Oracle エラーの番号を戻します。SQLCODE が戻す番号は負の値ですが、Oracle エラー「データが見つからない (*no data found*)」の場合は例外です。この場合、SQLCODE は +100 を戻します。

ユーザー定義の例外の場合、SQLCODE は +1 を戻します。ただし、EXCEPTION_INIT プラグマを使って例外を Oracle エラー番号に関連付けている場合例外です。この場合、SQLCODE はそのエラー番号を戻します。詳細は、6-17 ページの「[SQLCODE と SQLERRM の使用](#)」を参照してください。

構文

sqlcode_function
→ SQLCODE →

使用方法

SQLCODE は、SQL 文の中で直接使うことができません。まず、SQLCODE の値をローカル変数に代入する必要があります。たとえば、

```
DECLARE
    my_sqlcode NUMBER;
BEGIN
    ...
EXCEPTION
    WHEN OTHERS THEN
        my_sqlcode := SQLCODE;
        INSERT INTO errors VALUES (my_sqlcode, ...);
END;
```

SQLCODE は呼び出された内部例外の識別に使えるので、OTHERS 例外ハンドラの中で使うと特に便利です。

プラグマ RESTRICT_REFERENCES を使用してパッケージ・ファンクションの純正度を示す場合、ファンクションが SQLCODE をコールするのであれば、WNPS および RNPS 制約は指定できません。

関連項目

例外、SQLERRM ファンクション

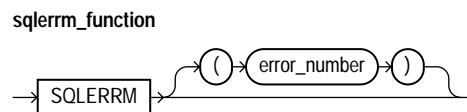
SQLERRM ファンクション

ファンクション SQLERRM は、エラー番号の引数に対応付けられているエラー・メッセージを返します。引数が省略されている場合は、現在の SQLCODE の値と対応付けられているエラー・メッセージを返します。引数なしの SQLERRM は、例外ハンドラの中でしか意味がありません。ハンドラの外側では、引数なしの SQLERRM は常にメッセージ「通常の正常終了 (normal, successful completion)」を返します。

内部例外の場合、SQLERRM は、発生した Oracle エラーに対応付けられているメッセージを返します。メッセージの先頭には Oracle エラー・コードが示されています。

ユーザー定義の例外の場合、SQLERRM はメッセージ「ユーザー定義の例外 (user-defined exception)」を返します。ただし、EXCEPTION_INIT プラグマを使って例外を Oracle エラー番号に対応付けている場合は例外です。この場合、SQLERRM は対応するエラー・メッセージを返します。詳細は、6-17 ページの「[SQLCODE と SQLERRM の使用](#)」を参照してください。

構文



キーワードとパラメータの説明

error_number

有効な Oracle エラー番号でなければなりません。Oracle エラーのリストは、『Oracle8i エラー・メッセージ』にあります。

使用方法

SQLERRM にエラー番号を渡すことができます。このとき、SQLERRM はそのエラー番号に結び付けられたメッセージを返します。SQLERRM に渡されるエラー番号は、負の値でなくてはなりません。ゼロを渡すと、SQLERRM は常に次のメッセージを返します。

ORA-0000: normal, successful completion

正の数値を渡すと、SQLERRM は常に次のメッセージを返します。

User-Defined Exception

ただし、+100 を渡した場合、SQLERRM は次のメッセージを戻します。

ORA-01403: no data found

SQLERRM は、SQL 文の中で直接使うことができません。まず、SQLERRM の値をローカル変数に代入する必要があります。たとえば、

```
DECLARE
    my_sqlerrm CHAR(150);
    ...
BEGIN
    ...
EXCEPTION
    ...
    WHEN OTHERS THEN
        my_sqlerrm := SUBSTR(SQLERRM, 1, 150);
        INSERT INTO errors VALUES (my_sqlerrm, ...);
END;
```

文字列ファンクション SUBSTR を使っているので、SQLERRM の値を my_sqlerrm に代入しても、(切捨ての結果として起こる) VALUE_ERROR 例外は呼び出されません。SQLERRM は呼び出された内部例外の識別に使えるので、OTHERS 例外ハンドラの中で使うと特に便利です。

プラグマ RESTRICT_REFERENCES を使用してパッケージ・ファンクションの純正度を示す場合、ファンクションが SQLCODE をコールするのであれば、WNPS および RNPS 制約は指定できません。

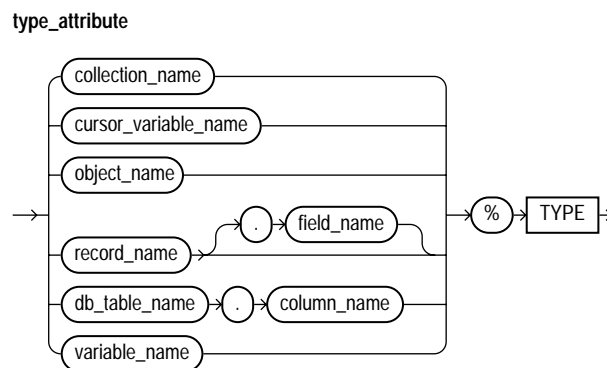
関連項目

例外、SQLCODE ファンクション

%TYPE 属性

%TYPE 属性は、フィールド、レコード、ネストした表、データベース列、または変数のデータ型を指定します。%TYPE 属性は、定数、変数、フィールド、またはパラメータを宣言するときにデータ型指定子として使えます。詳細は、2-30 ページの「[%TYPE の使用方法](#)」を参照してください。

構文



キーワードとパラメータの説明

collection_name

現在の有効範囲のうちこれより前の部分で宣言されている、ネストした表、索引付き表、または varray を指定します。

cursor_variable_name

現在の有効範囲の中で事前に宣言されている PL/SQL カーソル変数を識別します。カーソル変数に代入できるのは、別のカーソル変数の値だけです。

object_name

現在の有効範囲のうちこれより前の部分で宣言されているオブジェクト（オブジェクト型のインスタンス）を指定します。

record_name

現在の有効範囲のうちこれより前に宣言されているユーザー定義のレコードまたは %ROWTYPE 属性のレコードです。

record_name.field_name

現在の有効範囲の中で事前に宣言されているユーザー定義のレコードまたは %ROWTYPE 属性のレコードのフィールドを識別します。

table_name.column_name

宣言が PL/SQL コンパイラによって処理されるときにアクセスできない表および列を参照します。

variable_name

同じ有効範囲の中で事前に宣言されている変数を識別します。

使用方法

%TYPE 属性は、データベース列を参照する変数、フィールド、およびパラメータを宣言する場合に特に便利です。しかし、%TYPE を使って宣言した項目には NOT NULL 列制約は継承されません。

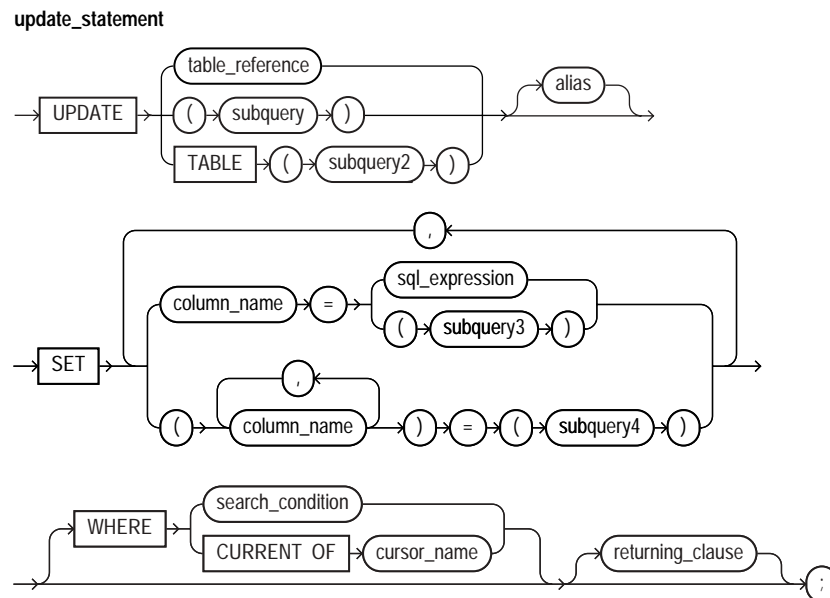
関連項目

定数と変数、%ROWTYPE 属性

UPDATE 文

UPDATE 文は、表またはビューの中の 1 行以上の行にある指定された列の値を変更します。
UPDATE 文の詳細は、『Oracle8i SQL リファレンス』を参照してください。

構文



キーワードとパラメータの説明

table_reference

表またはビューを指定します。指定された表またはビューは、UPDATE 文の実行時にアクセスできなければならない、ユーザーが UPDATE 権限を持っていないなりません。table_reference の構文は、11-48 ページの「[DELETE 文](#)」を参照してください。

subquery

処理する行の集合を提供する SELECT 文です。構文は select_into_statement の構文と似ていますが、INTO 句は使えません。11-141 ページの「[SELECT INTO 文](#)」を参照してください。

TABLE (subquery2)

TABLE のオペランドは、1 つの列値を戻す SELECT 文です。これはネストした表、またはネストした表として割り当てられる varray である必要があります。演算子 TABLE は、値がスカラー値ではなくコレクションであることを Oracle に通知します。

alias

参照される表またはビューの別名（通常は短縮名）で、WHERE 句の中でよく使用されます。

column_name

更新する列（または更新する複数の列の中の 1 つ）の名前です。これは参照される表またはビューの列の名前でなければなりません。column_name リストでは同じ列名を繰り返して使えません。UPDATE 文の列名は、表またはビューの中と同じ順序で指定する必要はありません。

sql_expression

任意の有効な SQL の式です。詳細は、『Oracle8i SQL リファレンス』を参照してください。

SET column_name = sql_expression

この句は sql_expression の値を、column_name によって識別される列に代入します。sql_expression の中で、更新される表の列が参照されている場合、参照は現在の行の列が対象になります。古い列の値は、等号の右辺で使われます。

次の例では、すべての従業員の給与を 10% だけ増加させています。sal 列の元の値が 1.10 倍され、その結果が同じ sal 列に代入されて元の値を上書きします。

```
UPDATE emp SET sal = sal * 1.10;
```

SET column_name = (subquery3)

この句は、subquery3 でデータベースから取り出した値を、column_name の列に代入します。この副問合せは、正確に 1 つの行と 1 つの列を戻さなければなりません。

SET (column_name, column_name, ...)= (subquery4)

この句は、subquery4 でデータベースから取り出した値を、column_name リストにある列に代入します。subquery は、リストされている列すべてを含む 1 つの行だけを戻さなければなりません。

subquery によって戻された列値は、列リストの列に順番に代入されます。1 番目の値はリストの 1 番目の列に、2 番目の値はリストの 2 番目の列に、というように代入されます。

次の相関問合せでは、item_num に格納されている値が item_id 列に代入され、item_price に格納されている値が price 列に代入されます。

```
UPDATE inventory inv -- alias
  SET (item_id, price) =
    (SELECT item_num, item_price FROM item_table
     WHERE item_name = inv.item_name);
```

WHERE search_condition

この句は、データベース表の中の更新対象行を選択します。検索条件を満たす行だけが更新されます。検索条件を省略すると、表の中のすべての行が更新されます。

WHERE CURRENT OF cursor_name

この句は、cursor_name で識別されるカーソルに結び付けられている FETCH 文によって処理された最後の行を参照します。カーソルは、FOR UPDATE であること、さらにオープンされていて行に置かれていることが必要です。

カーソルがオープンされていないと、CURRENT OF 句でエラーが発生します。カーソルがオープンされていても、取り出された行がないか、最後の取り出しで行が戻されなかった場合は、PL/SQL により事前定義の例外 NO_DATA_FOUND が呼び出されます。

returning_clause

この句を使うと、更新された行から値を戻せるので、後で行を SELECT で選択する必要がありません。取得した列値を、変数がホスト変数（またはその両方）あるいはコレクションかホスト変数（またはその両方）に代入できます。ただし、RETURNING 句はリモート、またはパラレルでの更新には使用できません。returning_clause の構文は、11-48 ページの「[DELETE 文](#)」を参照してください。

使用方法

UPDATE WHERE CURRENT OF 文は、オープンされているカーソルからの取出し（カーソル FOR ループで実行される暗黙の取出しを含む）の後で使えます（ただしそのためには、対応付けられた問合せが FOR UPDATE でなければなりません）。この文は現在の行、すなわち直前に取り出された行を更新します。

暗黙的な SQL カーソルとカーソル属性 %NOTFOUND、%FOUND、%ROWCOUNT、および %ISOPEN を使うと、UPDATE 文の実行に関する有用な情報にアクセスできます。

UPDATE 文では、1 つまたは複数の行が更新されるか、行の更新が実行されないかのどちらかです。1 つまたは複数の行が更新された場合、属性の値は次のようになります。

- SQL%NOTFOUND は、FALSE になる。
- SQL%FOUND は、TRUE になる。
- SQL%ROWCOUNT は、更新された行数になる。

更新された行がない場合、属性の値は次のようになります。

- SQL%NOTFOUND は、TRUE になる。
- SQL%FOUND は、FALSE になる。
- SQL%ROWCOUNT は 0 になる。

例

次の例では、部門 20 のアナリストを 10% 昇給しています。

```
UPDATE emp SET sal = sal * 1.10
WHERE job = 'ANALYST' AND DEPTNO = 20;
```

次の例では、Ford という名前の従業員が Analyst のポジションに昇進し、給与が 15% 上がっています。

```
UPDATE emp SET job = 'ANALYST', sal = sal * 1.15
WHERE ename = 'FORD';
```

最後の例では、更新される行から値を戻して変数に格納します。

```
UPDATE emp SET sal = sal + 500 WHERE ename = 'MILLER'
RETURNING sal, ename INTO my_sal, my_ename;
```

関連項目

DELETE 文、FETCH 文

サンプル・プログラム

この付録では、独自のプログラムを作る上で参考になる PL/SQL プログラムをいくつか示します。サンプル・プログラムには PL/SQL の重要な概念や機能が盛り込まれています。

主なトピック

[プログラムの実行](#)

[サンプル 1. FOR ループ](#)

[サンプル 2. カーソル](#)

[サンプル 3. 有効範囲](#)

[サンプル 4. バッチ・トランザクション処理](#)

[サンプル 5. 埋込み PL/SQL](#)

[サンプル 6. ストアド・プロシージャのコール](#)

プログラムの実行

この付録に掲載されたすべてのサンプル・プログラムおよびこのマニュアルに掲載されたその他のいくつかのサンプル・プログラムは、オンラインでアクセスできます。このようなプログラムには、次のようなコメントが付いています。

```
-- available online in file '<filename>'
```

オンライン・ファイルは、PL/SQL のデモ・ディレクトリにあります。デモ・ディレクトリの位置については、使っているシステムに該当する Oracle のインストレーション・ガイドまたはユーザーズ・ガイドを参照してください。ファイルと掲載ページを次の表に示します。

ファイル名	掲載ページ
例 1	「主な特徴」 (1-2 ページ)
例 2	「条件制御」 (1-8 ページ)
例 3	「反復制御」 (1-9 ページ)
例 4	「別名の使用」 (2-32 ページ)
例 7	「カーソル FOR ループの使用」 (5-12 ページ)
例 8	「パラメータ渡し」 (5-13 ページ)
例 5	「例」 (5-33 ページ)
例 6	「例」 (5-33 ページ)
例 11	「例」 (11-12 ページ)
例 12	「例」 (11-35 ページ)
例 13	「例」 (11-35 ページ)
例 14	「例」 (11-35 ページ)
サンプル 1	「サンプル 1. FOR ループ」 (A-3 ページ)
サンプル 2	「サンプル 2. カーソル」 (A-4 ページ)
サンプル 3	「サンプル 3. 有効範囲」 (A-5 ページ)
サンプル 4	「サンプル 4. バッチ・トランザクション処理」 (A-7 ページ)
サンプル 5	「サンプル 5. 埋込み PL/SQL」 (A-10 ページ)
サンプル 6	「サンプル 6. ストアド・プロシージャのコール」 (A-14 ページ)

SQL*Plus から対話形式で実行されるサンプルと、Pro*C プログラムから実行されるサンプルがあります。これらのサンプルは任意の Oracle アカウントから試すことができます。ただし、Pro*C のサンプルでは scott/tiger アカウントを使用します。

サンプルを試す前に、まずデータベースの表をいくつか作り、表にデータをロードしなければなりません。このためには、PL/SQL に付属する 2 つの SQL*Plus スクリプト、`exampbld` と `exemplod` を実行します。これらのスクリプトは、PL/SQL のデモ・ディレクトリにあります。

最初のスクリプトは、サンプル・プログラムが処理するデータベースの表を作成します。2 番目のスクリプトはデータベースの表をロード（または再ロード）します。スクリプトを実行する場合は、SQL*Plus を起動してから、次のコマンドを発行してください。

```
SQL> START exampbld
...
SQL> START exemplod
```

サンプル 1. FOR ループ

次の例では、単純な FOR ループを使ってデータベースの表に 10 個の行を挿入します。ループ索引、カウンタ変数、および 2 つの文字列のうちのどちらかの値が挿入されます。どちらの文字列が挿入されるかは、ループ索引の値に依存します。

入力表

使いません。

PL/SQL ブロック

```
-- available online in file 'sample1'
DECLARE
  x NUMBER := 100;
BEGIN
  FOR i IN 1..10 LOOP
    IF MOD(i,2) = 0 THEN      -- i is even
      INSERT INTO temp VALUES (i, x, 'i is even');
    ELSE
      INSERT INTO temp VALUES (i, x, 'i is odd');
    END IF;
    x := x + 100;
  END LOOP;
  COMMIT;
END;
```

出力表

```
SQL> SELECT * FROM temp ORDER BY col1;
```

COL1	COL2	MESSAGE
1	100	i is odd
2	200	i is even
3	300	i is odd
4	400	i is even
5	500	i is odd
6	600	i is even
7	700	i is odd
8	800	i is even
9	900	i is odd
10	1000	i is even

サンプル 2. カーソル

次の例では、カーソルを使って表 emp から給与の最も高い 5 人の従業員を選択しています。

入力表

```
SQL> SELECT ename, empno, sal FROM emp ORDER BY sal DESC;
```

ENAME	EMPNO	SAL
KING	7839	5000
SCOTT	7788	3000
FORD	7902	3000
JONES	7566	2975
BLAKE	7698	2850
CLARK	7782	2450
ALLEN	7499	1600
TURNER	7844	1500
MILLER	7934	1300
WARD	7521	1250
MARTIN	7654	1250
ADAMS	7876	1100
JAMES	7900	950
SMITH	7369	800

PL/SQL ブロック

```
-- available online in file 'sample2'
DECLARE
  CURSOR c1 IS
    SELECT ename, empno, sal FROM emp
    ORDER BY sal DESC;  -- start with highest paid employee
  my_ename CHAR(10);
  my_empno NUMBER(4);
  my_sal    NUMBER(7,2);
BEGIN
  OPEN c1;
  FOR i IN 1..5 LOOP
    FETCH c1 INTO my_ename, my_empno, my_sal;
    EXIT WHEN c1%NOTFOUND;  /* in case the number requested */
                                /* is more than the total      */
                                /* number of employees      */
    INSERT INTO temp VALUES (my_sal, my_empno, my_ename);
    COMMIT;
  END LOOP;
  CLOSE c1;
END;
```

出力表

```
SQL> SELECT * FROM temp ORDER BY col1 DESC;
```

COL1	COL2	MESSAGE
5000	7839	KING
3000	7902	FORD
3000	7788	SCOTT
2975	7566	JONES
2850	7698	BLAKE

サンプル 3. 有効範囲

次の例は、ブロック構造と有効範囲の規則を示すためのものです。外側のブロックで `x` と `counter` という 2 つの変数を宣言し、4 回ループします。ループの内側には、やはり `x` という名前の変数を宣言するサブブロックがあります。表 `temp` に挿入される値を見ると、2 つの `x` が異なる変数であることが確認できます。

入力表

使いません。

PL/SQL ブロック

```
-- available online in file 'sample3'
DECLARE
  x NUMBER := 0;
  counter NUMBER := 0;
BEGIN
  FOR i IN 1..4 LOOP
    x := x + 1000;
    counter := counter + 1;
    INSERT INTO temp VALUES (x, counter, 'outer loop');
    /* start an inner block */
    DECLARE
      x NUMBER := 0; -- this is a local version of x
    BEGIN
      FOR i IN 1..4 LOOP
        x := x + 1; -- this increments the local x
        counter := counter + 1;
        INSERT INTO temp VALUES (x, counter, 'inner loop');
      END LOOP;
    END;
  END LOOP;
  COMMIT;
END;
```

出力表

```
SQL> SELECT * FROM temp ORDER BY col2;
```

COL1	COL2	MESSAGE
-----	-----	-----
1000	1	OUTER loop
1	2	inner loop
2	3	inner loop
3	4	inner loop
4	5	inner loop
2000	6	OUTER loop
1	7	inner loop
2	8	inner loop
3	9	inner loop
4	10	inner loop
3000	11	OUTER loop
1	12	inner loop
2	13	inner loop
3	14	inner loop
4	15	inner loop

4000	16	OUTER loop
1	17	inner loop
2	18	inner loop
3	19	inner loop
4	20	inner loop

サンプル 4. バッチ・トランザクション処理

次の例では、表 `action` に格納されている指示に従って表 `accounts` が変更されます。表 `action` の各行には、口座番号、実行するアクション（挿入は I、更新は U、削除は D）、口座の更新金額、およびトランザクションを順番に並べるために使う時間タグが入っています。

挿入する場合、口座がすでに存在していればかわりに更新されます。更新する場合、口座が存在していなければ挿入によって作られます。削除する場合、行が存在しなければ何のアクションも起こりません。

入力表

```
SQL> SELECT * FROM accounts ORDER BY account_id;
```

ACCOUNT_ID	BAL
1	1000
2	2000
3	1500
4	6500
5	500

```
SQL> SELECT * FROM action ORDER BY time_tag;
```

ACCOUNT_ID	O	NEW_VALUE	STATUS	TIME_TAG
3	u	599		18-NOV-88
6	i	20099		18-NOV-88
5	d			18-NOV-88
7	u	1599		18-NOV-88
1	i	399		18-NOV-88
9	d			18-NOV-88
10	x			18-NOV-88

PL/SQL ブロック

```
-- available online in file 'sample4'
DECLARE
    CURSOR c1 IS
        SELECT account_id, oper_type, new_value FROM action
        ORDER BY time_tag
        FOR UPDATE OF status;
BEGIN
    FOR acct IN c1 LOOP -- process each row one at a time

        acct.oper_type := upper(acct.oper_type);

        /*-----*/
        /* Process an UPDATE.  If the account to */
        /* be updated doesn't exist, create a new */
        /* account.                                */
        /*-----*/
        IF acct.oper_type = 'U' THEN
            UPDATE accounts SET bal = acct.new_value
            WHERE account_id = acct.account_id;

            IF SQL%NOTFOUND THEN -- account didn't exist. Create it.
                INSERT INTO accounts
                    VALUES (acct.account_id, acct.new_value);
                UPDATE action SET status =
                    'Update: ID not found. Value inserted.'
                WHERE CURRENT OF c1;
            ELSE
                UPDATE action SET status = 'Update: Success.'
                WHERE CURRENT OF c1;
            END IF;

            /*-----*/
            /* Process an INSERT.  If the account already */
            /* exists, do an update of the account        */
            /* instead.                                    */
            /*-----*/
            ELIF acct.oper_type = 'I' THEN
                BEGIN
                    INSERT INTO accounts
                        VALUES (acct.account_id, acct.new_value);
                    UPDATE action set status = 'Insert: Success.'
                    WHERE CURRENT OF c1;
                EXCEPTION
                    WHEN DUP_VAL_ON_INDEX THEN -- account already exists
                        UPDATE accounts SET bal = acct.new_value
```

```

        WHERE account_id = acct.account_id;
        UPDATE action SET status =
            'Insert: Acct exists. Updated instead.'
        WHERE CURRENT OF c1;

    END;

/*-----*/
/* Process a DELETE.  If the account doesn't */
/* exist, set the status field to say that    */
/* the account wasn't found.                  */
/*-----*/
ELSIF acct.oper_type = 'D' THEN
    DELETE FROM accounts
        WHERE account_id = acct.account_id;

    IF SQL%NOTFOUND THEN -- account didn't exist.
        UPDATE action SET status = 'Delete: ID not found.'
        WHERE CURRENT OF c1;
    ELSE
        UPDATE action SET status = 'Delete: Success.'
        WHERE CURRENT OF c1;
    END IF;

/*-----*/
/* The requested operation is invalid.        */
/*-----*/
ELSE -- oper_type is invalid
    UPDATE action SET status =
        'Invalid operation. No action taken.'
    WHERE CURRENT OF c1;

END IF;

END LOOP;
COMMIT;
END;
```

出力表

```
SQL> SELECT * FROM accounts ORDER BY account_id;
```

ACCOUNT_ID	BAL
1	399
2	2000
3	599
4	6500

サンプル 5. 埋込み PL/SQL

```

        6      20099
        7      1599

SQL> SELECT * FROM action ORDER BY time_tag;

ACCOUNT_ID  O  NEW_VALUE STATUS                                TIME_TAG
-----
          3  u      599 Update: Success.                18-NOV-88
          6  i    20099 Insert: Success.                18-NOV-88
          5  d                Delete: Success.                18-NOV-88
          7  u      1599 Update: ID not found.                18-NOV-88
                        Value inserted.
          1  i      399 Insert: Acct exists.                18-NOV-88
                        Updated instead.
          9  d                Delete: ID not found.                18-NOV-88
         10  x                Invalid operation.                18-NOV-88
                        No action taken.
```

サンプル 5. 埋込み PL/SQL

次の例では、C などの高水準ホスト言語に PL/SQL を埋め込む方法を示します。また、銀行の出金トランザクションの実行例を示します。

入力表

```
SQL> SELECT * FROM accounts ORDER BY account_id;

ACCOUNT_ID  BAL
-----
          1    1000
          2    2000
          3    1500
          4    6500
          5     500
```

C プログラム中の PL/SQL ブロック

```
/* available online in file 'sample5' */
#include <stdio.h>
char buf[20];
EXEC SQL BEGIN DECLARE SECTION;
int acct;
double debit;
double new_bal;
VARCHAR status[65];
```



```

        VARCHAR uid[20];
        VARCHAR pwd[20];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;

main()
{
    extern double atof();

    strcpy (uid.arr,"scott");
    uid.len=strlen(uid.arr);
    strcpy (pwd.arr,"tiger");
    pwd.len=strlen(pwd.arr);

    printf("\n\n\tEmbedded PL/SQL Debit Transaction Demo\n\n");
    printf("Trying to connect...");
    EXEC SQL WHENEVER SQLERROR GOTO errprint;
    EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
    printf(" connected.\n");
for (;;)          /* Loop infinitely */
{
    printf("\n** Debit which account number? (-1 to end) ");
    gets(buf);
    acct = atoi(buf);
    if (acct == -1) /* Need to disconnect from Oracle */
    {
        /* and exit loop if account is -1 */
        EXEC SQL COMMIT RELEASE;
        exit(0);
    }

    printf("   What is the debit amount? ");
    gets(buf);
    debit = atof(buf);

    /* ----- */
    /* ----- Begin the PL/SQL block ----- */
    /* ----- */
    EXEC SQL EXECUTE

    DECLARE
        insufficient_funds    EXCEPTION;
        old_bal               NUMBER;
        min_bal               NUMBER := 500;
    BEGIN
        SELECT bal INTO old_bal FROM accounts
            WHERE account_id = :acct;

```

サンプル 5. 埋込み PL/SQL

```
-- If the account doesn't exist, the NO_DATA_FOUND
-- exception will be automatically raised.
:new_bal := old_bal - :debit;
IF :new_bal >= min_bal THEN
    UPDATE accounts SET bal = :new_bal
    WHERE account_id = :acct;
    INSERT INTO journal
    VALUES (:acct, 'Debit', :debit, SYSDATE);
    :status := 'Transaction completed.';
ELSE
    RAISE insufficient_funds;
END IF;
COMMIT;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        :status := 'Account not found.';
        :new_bal := -1;
    WHEN insufficient_funds THEN
        :status := 'Insufficient funds.';
        :new_bal := old_bal;
    WHEN OTHERS THEN
        ROLLBACK;
        :status := 'Error: ' || SQLERRM(SQLCODE);
        :new_bal := -1;
END;

END-EXEC;
/* ----- */
/* ----- End the PL/SQL block ----- */
/* ----- */

status.arr[status.len] = '\0'; /* null-terminate */
/* the string */
printf("\n\n Status: %s\n", status.arr);
if (new_bal >= 0)
    printf(" Balance is now: $%.2f\n", new_bal);
} /* End of loop */

errprint:
EXEC SQL WHENEVER SQLERROR CONTINUE;
printf("\n\n>>>> Error during execution:\n");
printf("%s\n", sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK RELEASE;
exit(1);
}
```

対話型セッション

```
Embedded PL/SQL Debit Transaction Demo

Trying to connect... connected.

** Debit which account number? (-1 to end) 1
What is the debit amount? 300

Status: Transaction completed.
Balance is now: $700.00

** Debit which account number? (-1 to end) 1
What is the debit amount? 900
Status: Insufficient funds.
Balance is now: $700.00

** Debit which account number? (-1 to end) 2
What is the debit amount? 500

Status: Transaction completed.
Balance is now: $1500.00

** Debit which account number? (-1 to end) 2
What is the debit amount? 100

Status: Transaction completed.
Balance is now: $1400.00

** Debit which account number? (-1 to end) 99
What is the debit amount? 100

Status: Account not found.

** Debit which account number? (-1 to end) -1
```

出力表

```
SQL> SELECT * FROM accounts ORDER BY account_id;

ACCOUNT_ID  BAL
-----
1          700
2         1400
3         1500
4         6500
5          500
```

サンプル 6. ストアド・プロシージャのコール

```
SQL> SELECT * FROM journal ORDER BY date_tag;
```

ACCOUNT_ID	ACTION	AMOUNT	DATE_TAG
1	Debit	300	28-NOV-88
2	Debit	500	28-NOV-88
2	Debit	100	28-NOV-88

サンプル 6. ストアド・プロシージャのコール

この Pro*C プログラムは Oracle に接続し、ユーザーに部門番号の入力を要求して、`personnel` というパッケージに格納されている `get_employees` というプロシージャをコールします。このプロシージャは、3 つの索引付き表を OUT 仮パラメータとして宣言し、一連の従業員データをフェッチしてその索引付き表に入れます。合致する実パラメータはホスト配列です。

プロシージャが終了すると、索引付き表のすべての行の値が、自動的にホスト配列中の対応する要素に割り当てられます。プログラムは、データがなくなるまで繰り返しプロシージャをコールして、取り出した従業員データを表示します。

入力表

```
SQL> SELECT ename, empno, sal FROM emp ORDER BY sal DESC;
```

ENAME	EMPNO	SAL
KING	7839	5000
SCOTT	7788	3000
FORD	7902	3000
JONES	7566	2975
BLAKE	7698	2850
CLARK	7782	2450
ALLEN	7499	1600
TURNER	7844	1500
MILLER	7934	1300
WARD	7521	1250
MARTIN	7654	1250
ADAMS	7876	1100
JAMES	7900	950
SMITH	7369	800

ストアド・プロシージャ

```
/* available online in file 'sample6' */
#include <stdio.h>
#include <string.h>

typedef char asciz;

EXEC SQL BEGIN DECLARE SECTION;
/* Define type for null-terminated strings. */
EXEC SQL TYPE asciz IS STRING(20);
asciz username[20];
asciz password[20];
int dept_no; /* which department to query */
char emp_name[10][21];
char job[10][21];
float salary[10];
int done_flag;
int array_size;
int num_ret; /* number of rows returned */
int SQLCODE;
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE sqlca;

int print_rows(); /* produces program output */
int sqlerror(); /* handles unrecoverable errors */

main()
{
    int i;

    /* Connect to Oracle. */
    strcpy(username, "SCOTT");
    strcpy(password, "TIGER");

    EXEC SQL WHENEVER SQLERROR DO sqlerror();

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to Oracle as user: %s\n\n", username);

    printf("Enter department number: ");
    scanf("%d", &dept_no);
    fflush(stdin);

    /* Print column headers. */
    printf("\n\n");
```

サンプル 6. ストアド・プロシージャのコール

```
printf("%-10.10s%-10.10s%\n", "Employee", "Job", "Salary");
printf("%-10.10s%-10.10s%\n", "-----", "----", "-----");

/* Set the array size. */
array_size = 10;
done_flag = 0;
num_ret = 0;

/* Array fetch loop - ends when NOT FOUND becomes true. */
for (;;)
{
    EXEC SQL EXECUTE
        BEGIN personnel.get_employees
            (:dept_no, :array_size, :num_ret, :done_flag,
             :emp_name, :job, :salary);
        END;
    END-EXEC;

    print_rows(num_ret);

    if (done_flag)
        break;
}

/* Disconnect from Oracle. */
EXEC SQL COMMIT WORK RELEASE;
exit(0);
}

print_rows(n)
int n;
{
    int i;

    if (n == 0)
    {
        printf("No rows retrieved.\n");
        return;
    }

    for (i = 0; i < n; i++)
        printf("%10.10s%10.10s%6.2f\n",
            emp_name[i], job[i], salary[i]);
}

sqlerror()
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
```

```

        printf("\nOracle error detected:");
        printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc);
        EXEC SQL ROLLBACK WORK RELEASE;
        exit(1);
    }

```

対話型セッション

Connected to Oracle as user: SCOTT

Enter department number: 20

Employee	Job	Salary
-----	---	-----
SMITH	CLERK	800.00
JONES	MANAGER	2975.00
SCOTT	ANALYST	3000.00
ADAMS	CLERK	1100.00
FORD	ANALYST	3000.00

CHAR と VARCHAR2 の意味の比較

この付録では、基本型 CHAR と VARCHAR2 の意味上の相違点を説明します。微妙ではあっても重要なこれらの相違点は、文字値の割当て、比較、挿入、更新、選択、またはフェッチに関係してきます。

主なトピック

[文字値の割当て](#)

[文字値の比較](#)

[文字値の挿入](#)

[文字値の選択](#)

文字値の割当て

文字値を CHAR 型変数に割り当てるとき、変数の宣言された長さよりも値が短い場合、PL/SQL は、その値が宣言された長さと同じ長さになるまで空白を埋めます。そのため、後続する空白に関する情報は失われます。たとえば、次の宣言では、last_name に割り当てられた値の後には、1 つではなく 6 つの空白ができます。

```
last_name CHAR(10) := 'CHEN '; -- note trailing blank
```

CHAR 型変数の宣言された長さよりも文字値が長い場合、PL/SQL は割当てを中止して事前定義の例外 VALUE_ERROR を呼び出します。PL/SQL は、値を切り捨てたり、後続する空白を切り捨てたりはしません。たとえば、次のような宣言があるとします。

```
acronym CHAR(4);
```

次のような割当てを試みると VALUE_ERROR が呼び出されます。

```
acronym := 'SPCA'; -- note trailing blank
```

文字値を VARCHAR2 型変数に割り当てるとき、変数の宣言された長さよりも値が短い場合に、値を空白で埋めたり、値に後続する空白を削除することはありません。文字値はそのまま割り当てられ、情報は何も失われません。CHAR 型変数の宣言された長さよりも文字値が長い場合、PL/SQL は割当てを中止して VALUE_ERROR を呼び出します。PL/SQL は、値を切り捨てたり、後続する空白を切り捨てたりはしません。

文字値の比較

関係演算子を使うと、2 つの文字値が等しいかどうかを比較できます。比較はデータベースのキャラクタ・セットの照合順番に基づいてなされます。文字値の比較では、照合順番で後にくる方が文字値が大きくなります。たとえば、次のような宣言があるとします。

```
last_name1 VARCHAR2(10) := 'COLES';  
last_name2 VARCHAR2(10) := 'COLEMAN';
```

次の IF 条件は TRUE です。

```
IF last_name1 > last_name2 THEN ...
```

ANSI/ISO SQL では、比較する 2 つの文字値が同じ長さでなければなりません。このため、比較対象の値がいずれもデータ型 CHAR を持つ場合は、空白埋め比較方法が使われます。つまり、長さが異なる文字値を比較する前に、短い方の値に、長い方の値と同じ長さになるまで空白埋めがなされます。たとえば、次のような宣言があるとします。

```
last_name1 CHAR(5) := 'BELLO';  
last_name2 CHAR(10) := 'BELLO  '; -- note trailing blanks
```

次の IF 条件は TRUE です。

```
IF last_name1 = last_name2 THEN ...
```

比較対象の値の一方がデータ型 VARCHAR2 の場合、非空白埋め比較方法が使われます。つまり、長さが異なる文字値を比較する場合に、PL/SQL は調整せず、そのままの長さを使います。たとえば、次のような宣言があるとします。

```
last_name1 VARCHAR2(10) := 'DOW';  
last_name2 VARCHAR2(10) := 'DOW   '; -- note trailing blanks
```

次の IF 条件は FALSE です。

```
IF last_name1 = last_name2 THEN ...
```

比較対象の値の一方がデータ型 VARCHAR2 で、もう一方の値がデータ型 CHAR の場合、非空白埋め比較方法が使われます。ただし、文字値を CHAR 変数に割り当てるときに、その値が変数の宣言された長さよりも短い場合、PL/SQL は、その値が宣言された長さになるまで空白を埋めます。たとえば、次のような宣言を考えます。

```
last_name1 VARCHAR2(10) := 'STAUB';  
last_name2 CHAR(10)      := 'STAUB'; -- PL/SQL blank-pads value
```

last_name2 の値の後に 5 つの空白が含まれるため、次の IF 条件は FALSE です。

```
IF last_name1 = last_name2 THEN ...
```

すべての文字列リテラルは、CHAR データ型を持っています。このため、比較対象の値の両方がリテラルの場合は、空白埋め比較方法が使われます。片方の値がリテラルの場合は、残りの値がデータ型 CHAR を持っている場合に限り空白埋め比較方法が使われます。

文字値の挿入

PL/SQL 文字変数の値を Oracle データベース列に挿入する場合、それが空白埋めされるかどうかは、変数の型ではなく列の型に依存します。

文字値を CHAR データベース列に挿入する場合、Oracle は値に後続する空白を削除しません。列の定義された幅よりも値が短ければ、Oracle は定義幅まで値を空白で埋めます。その結果、後続する空白に関する情報は失われます。文字値の長さが列の幅の定義より長いなら、Oracle は挿入を中止してエラーを発生させます。

文字値を VARCHAR2 データベース列に挿入する場合、Oracle は値に後続する空白を削除しません。列の定義された幅よりも値が短い場合も、Oracle は値の空白埋めをしません。文字値はそのまま格納されるので、情報は何も失われません。文字値の長さが列の幅の定義より長いなら、Oracle は挿入を中止してエラーを発生させます。

注意：更新の場合にも同じ規則が適用されます。

文字値を挿入するとき、RTRIM ファンクションを使用することにより、後続する空白が切り捨てられ、後続する空白が格納されないようにすることができます。たとえば、

```
DECLARE
    ...
    my_name VARCHAR2(15);
BEGIN
    ...
    my_ename := 'LEE  '; -- note trailing blanks
    INSERT INTO emp
        VALUES (my_empno, RTRIM(my_ename), ...); -- inserts 'LEE'
END;
```

文字値の選択

Oracle データベース列から値を選択して PL/SQL 文字変数に入れる場合、それが空白埋めされるかどうかは、列の型ではなく変数の型に依存します。

列値を選択して CHAR 変数に入れる場合、変数の宣言された長さよりも値が短ければ、PL/SQL は宣言された長さまで値を空白で埋めます。その結果、後続する空白に関する情報は失われます。文字値の長さが変数の長さの宣言より長い場合、PL/SQL は割当てを中止して例外 VALUE_ERROR を呼び出します。

列値を選択して VARCHAR2 変数に入れる場合、その値が変数の宣言された長さよりも短いと、PL/SQL は空白埋めも、後続する空白の削除もしません。文字値はそのまま格納されるので、情報は何も失われません。

たとえば、空白埋めの CHAR 列値を選択し VARCHAR2 変数に入れる場合、後続する空白は削除されません。CHAR 型変数の宣言された長さよりも文字値が長い場合、PL/SQL は割当てを中止して VALUE_ERROR を呼び出します。

注意： フェッチの場合にも同じ規則が適用されます。

PL/SQL ラッパー

この付録では、PL/SQL ラッパーの実行方法を示します。これは、PL/SQL ソース・コードを、移植性のあるオブジェクト・コードに変換するためのスタンドアロン・ユーティリティです。ラッパーを使うと、ソース・コードを隠したまま、PL/SQL のアプリケーションを配布できます。

主なトピック

[ラッピングの利点](#)

[PL/SQL ラッパーの実行](#)

ラッピングの利点

PL/SQL ラッパーは、PL/SQL ソース・コードをオブジェクト・コードの中間形式に変換します。ラッパーは、アプリケーションの内部を隠すことによって、次のことを防ぎます。

- 他の開発者によるアプリケーションの誤用。
- アルゴリズムの競合他社への公開。

ラップされたコードは、ソース・コードと同程度の移植性があります。PL/SQL コンパイラは、ラップされたコンパイル単位を自動的に認識し、ロードします。これ以外にも、次の利点があります。

- プラットフォームの独立性 同じコンパイル単位の複数のバージョンを配布する必要はない。
- 動的ロード ユーザーは、新しい機能を追加するためにシャット・ダウンおよび再リンクをする必要がない。
- 動的バインド 外部参照はロード時に解決される。
- 厳しい依存性検査 無効となったプログラム・ユニットは、自動的に再コンパイルされる。
- 正常なインポートとエクスポート インポート / エクスポート・ユーティリティで、ラップされたファイルを扱える。

PL/SQL ラッパーの実行

PL/SQL ラッパーを実行するには、次の構文を使って、オペレーティング・システム・プロンプトで `wrap` コマンドを入力します。

```
wrap iname=input_file [oname=output_file]
```

空白は個々の引数を区切るために使うため、等号の前後には空白を付けないでください。

`wrap` コマンドに必要な引数は次の 1 つだけです。

```
iname=input_file
```

ここで、`input_file` は、ラッパーの入力ファイルのパスと名前です。ファイル拡張子を指定する必要はありません。デフォルトで `sql` になります。たとえば、次のコマンドは同じ意味を持ちます。

```
wrap iname=/mydir/myfile  
wrap iname=/mydir/myfile.sql
```

ただし、次の例で示すように、異なるファイル拡張子を指定することもできます。

```
wrap iname=/mydir/myfile.src
```

wrap コマンドは、オプションで次のような 2 番目の引数を取ることもできます。

```
oname=output_file
```

ここで、output_file は、ラッパーの出力ファイルのパスと名前です。出力ファイルの名前はデフォルトで入力ファイルの名前となり、拡張子はデフォルトで plb (PL/SQL バイナリ) となるため、出力ファイルを指定する必要はありません。たとえば、次のコマンドは同じ意味を持ちます。

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql oname=/mydir/myfile.plb
```

ただし、oname オプションを使って、異なるファイル名と拡張子を指定できます。次に例を示します。

```
wrap iname=/mydir/myfile oname=/yourdir/yourfile.obj
```

入力ファイルと出力ファイル

入力ファイルでは、SQL 文を任意に組み合わせることができます。ただし、PL/SQL ラッパーは、オブジェクト型、パッケージまたはスタンドアロン・サブプログラムを定義する次の CREATE 文しかラップしません。

```
CREATE [OR REPLACE] TYPE type_name ... OBJECT
CREATE [OR REPLACE] TYPE BODY type_name
CREATE [OR REPLACE] PACKAGE package_name
CREATE [OR REPLACE] PACKAGE BODY package_name
CREATE [OR REPLACE] FUNCTION function_name
CREATE [OR REPLACE] PROCEDURE procedure_name
```

その他の SQL 文はすべて、そのままの形で出力ファイルに渡されます。コメント行は、オブジェクト型、パッケージまたはサブプログラム内にはいかり、削除されます。

ラップされると、オブジェクト型、パッケージまたはサブプログラムは次の形式になります。

```
<header> wrapped <body>
```

header は予約語 CREATE で始まり、オブジェクト型、パッケージまたはサブプログラムの名前で終わります。また、body はオブジェクト・コードの中間形式です。wrapped は、オブジェクト型、パッケージまたはサブプログラムがラッパーによって処理されたことを PL/SQL コンパイラに通知します。

ヘッダーにはコメントを含めることができます。たとえば、ラッパーは、次のソース・コードを

```
CREATE PACKAGE
-- Author: J. Hollings
-- Date: 12/15/98
```

```
banking AS
    minimum_balance CONSTANT REAL := 25.00;
    insufficient_funds EXCEPTION;
END banking;
```

次のオブジェクト・コードに変換します。

```
CREATE PACKAGE
-- Author: J. Hollings
-- Date: 12/15/98
banking wrapped
0
abcd ...
```

通常、出力ファイルは入力ファイルよりもかなり大きくなります。

提案：パッケージ（またはオブジェクト型）をラップする場合は、仕様部ではなく、本体だけをラップします。こうすると、パッケージの使用に必要な情報（サブプログラム仕様部）すべてを、パッケージのインプリメンテーションを隠したまま、他の開発者に提供できます。

エラー処理

入力ファイルに構文エラーが含まれている場合、PL/SQL ラッパーはそのエラーを検出し、報告します。ただし、ラッパーは外部参照を解決しないため、意味エラーは検出できません。たとえば、ラッパーは次のエラー（*table or view does not exist*）を報告しません。

```
CREATE PROCEDURE raise_salary (emp_id INTEGER, amount NUMBER) AS
BEGIN
    UPDATE emp -- should be emp
        SET sal = sal + amount WHERE empno = emp_id;
END;
```

PL/SQL コンパイラは外部参照を解決します。このため、ラッパー出力ファイル（.plb ファイル）がコンパイルされるときに、意味エラーが報告されます。

D

ネーム変換

この付録では、潜在的に意味の曖昧なプロシージャ文および SQL 文で、名前への参照を PL/SQL がどのように解決するかについて説明します。

主なトピック

[ネーム変換とは？](#)

[種々の参照](#)

[ネーム変換のアルゴリズム](#)

[獲得を理解する](#)

[獲得の防止](#)

[属性やメソッドへのアクセス](#)

[サブプログラムとメソッドのコール](#)

ネーム変換とは？

コンパイル中、PL/SQL コンパイラは変数名のような識別子を、アドレス（メモリー位置）、データ型、実際の値などに関連付けます。このプロセスをバインドといいます。関連付けは、リバインドをおこす再コンパイルが発生しない限り、一連のすべての動作に対して有効となります。

名前をバインドする前に、コンパイル単位で、これらの名前に対する参照がすべて解決されている必要があります。このプロセスをネーム変換といいます。PL/SQL では、名前はすべて同じ名前領域にあるものと見なされます。したがって、内部有効範囲における宣言または定義で、外部有効範囲における別の宣言または定義が隠されてしまう可能性があります。PL/SQL では文字列リテラルの場合を除いて、大 / 小文字が区別されないため、次の例では、変数 `client` の宣言によりデータ型 `Client` が隠されてしまっています。

```
BEGIN
  <<block1>>
  DECLARE
    TYPE Client IS RECORD (...);
    TYPE Customer IS RECORD (...);
  BEGIN
    DECLARE
      client Customer;          -- hides definition of type Client
                                -- in outer scope
      lead1 Client;             -- illegal; Client resolves to the
                                -- variable client
      lead2 block1.Client;      -- OK; refers to type Client
    BEGIN
      NULL;
    END;
  END;
END;
```

ただし、この場合でも、ブロック・ラベル `block1` への参照を修飾することにより、データ型 `Client` を参照することは可能です。

次に示す `CREATE TYPE person1` 文では、コンパイラは `manager` への 2 つ目の参照を、宣言しようとしている属性の名前として解決します。`CREATE TYPE person2` 文では、コンパイラは `manager` への 2 つ目の参照を、宣言したばかりの属性の名前として解決します。どちらの場合でも、コンパイラは型名を要求しているため、`manager` への参照はエラーとなります。

```
CREATE TYPE manager AS OBJECT (dept NUMBER);
CREATE TYPE person1 AS OBJECT (manager manager);
CREATE TYPE person2 AS OBJECT (manager NUMBER, mgr manager);
```

種々の参照

ネーム変換の際、コンパイラは、単なる未修飾の名前、ドットで区切られて連鎖した識別子、あるコレクションの索引付きのコンポーネントなど、様々な種類の参照に遭遇する可能性があります。以下に、有効な参照の例をいくつか示します。

```
CREATE PACKAGE pack1 AS
    m NUMBER;
    TYPE t1 IS RECORD (a NUMBER);
    v1 t1;
    TYPE t2 IS TABLE OF t1 INDEX BY BINARY_INTEGER;
    v2 t2;
    FUNCTION f1 (p1 NUMBER) RETURN t1;
    FUNCTION f2 (q1 NUMBER) RETURN t2;
END
/
CREATE PACKAGE BODY pack1 AS
    FUNCTION f1 (p1 NUMBER) RETURN t1 IS
        n NUMBER;
    BEGIN
        ...
        n := m;           -- (1) unqualified name
        n := pack1.m;      -- (2) dot-separated chain of identifiers
                           -- (package name used as scope
                           -- qualifier followed by variable name)
        n := pack1.f1.p1;  -- (3) dot-separated chain of identifiers
                           -- (package name used as scope
                           -- qualifier followed by function name
                           -- also used as scope qualifier
                           -- followed by parameter name)
        n := v1.a;        -- (4) dot-separated chain of identifiers
                           -- (variable name followed by
                           -- component selector)
        n := pack1.v1.a;   -- (5) dot-separated chain of identifiers
                           -- (package name used as scope
                           -- qualifier followed by
                           -- variable name followed by component
                           -- selector)
        n := v2(10).a;     -- (6) indexed name followed by component
                           -- selector
        n := f1(10).a;     -- (7) function call followed by component
                           -- selector
        n := f2(10)(10).a; -- (8) function call followed by indexing
                           -- followed by component selector
        n := scott.pack1.f2(10)(10).a;
                           -- (9) function call (which is a dot-
                           -- separated chain of identifiers,
```

```

--      including schema name used as
--      scope qualifier followed by package
--      name used as scope qualifier
--      followed by function name)
--      followed by component selector
--      of the returned result followed
--      by indexing followed by component
--      selector

END;
FUNCTION f2 (q1 NUMBER) RETURN t2 IS
BEGIN
    NULL;
END;
END;
/
CREATE OR REPLACE PACKAGE BODY pack1 AS
    FUNCTION f1 (p1 NUMBER) RETURN t1 IS
        n NUMBER;
    BEGIN
        n := scott.pack1.f1.n; -- (10) dot-separated chain of
                                --      identifiers (schema name
                                --      used as scope qualifier
                                --      followed by package name also
                                --      used as scope qualifier
                                --      followed by function name also
                                --      used as scope qualifier
                                --      followed by local
                                --      variable name)

    END;

    FUNCTION f2 (q1 NUMBER) RETURN t2 IS
    BEGIN
        NULL;
    END;
END;
/
```

ネーム変換のアルゴリズム

ネーム変換アルゴリズムについて説明します。

ネーム変換アルゴリズムの最初の部分では、ベースの検索が実行されます。ベースはドットで区切られて連鎖した識別子への最小の接頭辞で、現行の有効範囲で検索を行い、スキーマレベルの有効範囲へ向かって外側へ検索範囲を移動することにより、解決することができます。

上に示した例では、(3) `pack1.f1.p1` に対するベースは `pack1`、(4) `scott.pack1.f1.n` に対するベースは `scott.pack1`、(5) `v1.a` に対するベースは `v1` です。(5) では、`v1.a` の `a` はコンポーネント・セクタで、変数 `v1` のフィールド `a` として解決します。これは、`v1` が `t1` 型で、`a` というフィールドを持つためです。

ベースが見つからない場合、コンパイラは未宣言エラーを生成します。ベースが見つかった場合、コンパイラは参照の全体を解決しようとします。これがうまく行かない場合、コンパイラはエラーを生成します。

ベースの長さは、1、2、または3に限られます。3の値をとれるのは、SQLの有効範囲内で、コンパイラが3つの部分から構成される名前を次に示すように解決する場合に限られません。

`schema_name.table_name.column_name`

以下に、その他のベースの例を示します。

`variable_name`
`type_name`
`package_name`
`schema_name.package_name`
`schema_name.function_name`
`table_name`
`table_name.column_name`
`schema_name.table_name`
`schema_name.table_name.column_name`

ベースの検索

ここで、ベースを検索するためのアルゴリズムについて説明します。

コンパイラが、SQLの有効範囲で名前を解決している場合（これには、`INTO` 句の項目およびスキーマレベルの表名を除く DML 文のすべてが含まれます）まずその有効範囲内でベースの検索が実行されます。この有効範囲で見つからない場合、PL/SQL のローカル有効範囲で、非 SQL の有効範囲の名前に対する場合と同様にして、ベースの検索が実行されます。

次に、コンパイラが列名を検索しようとする場合に、SQLの有効範囲でベースを検索するためのルールを示します。

- 識別子が1つ与えられた場合、コンパイラは長さ1のベースの検索を実行する。この際、この識別子は有効範囲内の任意の FROM 句にリストされている表の1つに含まれる未修飾の列名として使用される。
- 連鎖した2つの識別子が与えられた場合、コンパイラは長さ2のベースの検索を実行する。この際、これらの識別子は表名または表の別名により修飾される列名として使用される。
- 連鎖した3つの識別子が与えられた場合、コンパイラは検索を実行する有効範囲ごとに、以下に示すいずれかの項目の検索を行う。検索は現行の有効範囲から開始され、外側に向かって実行される。
 - 長さ3のベース。ここで、3つの識別子はスキーマの名前によって修飾された表名により修飾された列名として使用される。
 - 長さ2のベース。ここで、最初の2つの識別子が、表の別名により修飾された任意のユーザー定義の列名として使用される。
- 連鎖した4つの識別子が与えられた場合、コンパイラは長さ2のベースの検索を実行する。この際、最初の2つの識別子が、表の別名として修飾されたユーザー定義の型の列名として使用される。

列名に使用するベースが見つかった場合、コンパイラは、ベースのコンポーネントなどを検索することにより（検索対象は列名の型により異なる）参照全体を解決しようとします。

次に、コンパイラが行の式が渡されると見込んでいる場合に、SQLの有効範囲でベースを検索するためのルールを示します。（行の式は、それだけで使用することのできる表の別名です。行の式は、オブジェクト表の演算子 REF または VALUE か、オブジェクト表の INSERT 文または UPDATE 文でしか使えません。）

- 識別子が1つ与えられた場合、コンパイラは表の別名として長さ1のベースの検索を実行する。検索は現行の有効範囲から開始され、外側に向かって実行される。表の別名に対応するオブジェクト表がない場合、コンパイラはエラーを生成する。
- 連鎖した2つ以上の識別子が与えられた場合、コンパイラはエラーを生成する。

解決中の名前について、次に示す2つのケースのいずれかに該当する場合を考えます。

- SQLの有効範囲に存在しない
- SQLの有効範囲に存在するが、これに対応するベースを有効範囲内に見つけることができない

これらの場合、ベースを見つけるため、コンパイラはコンパイル単位に対してローカルなすべての PL/SQL の有効範囲に対して検索を実行します。検索は現行の有効範囲から開始され、外側へ移動していきます。名前が発見された場合、ベースの長さは1になります。名前が見つからない場合、コンパイラは、次のルールに従いながらスキーマ・オブジェクトを検索することにより、ベースを見つけようとします。

1. まず、コンパイラは長さ 1 のベースを検索する。これは、連鎖した識別子の最初の識別子と名前的一致するスキーマ・オブジェクトのスキーマを検索することにより実行される。検索の結果として得られるスキーマ・オブジェクトは、パッケージ仕様部、ファンクション、プロシージャ、表、ビュー、順序、シノニム、スキーマレベルのデータ型のいずれかになる。シノニムの場合、ベースは、このシノニムによって指定された基本オブジェクトとして解決される。
2. 前の検索に失敗した場合、コンパイラは長さ 1 のベースの検索を実行する。この場合、連鎖した識別子の最初の識別子と名前的一致するパブリック・シノニムが検索される。これが成功した場合、ベースは、シノニムにより指定された基本オブジェクトとして解決される。
3. 前の検索に失敗し、連鎖した識別子が少なくとも 2 つの識別子を持つ場合、コンパイラは長さ 2 のベースの検索を実行する。この場合、連鎖した識別子の 2 つ目の識別子と名前が一致するスキーマ・オブジェクトで、連鎖の 1 つ目の識別子と名前的一致するスキーマにより所有されるものが検索される。
4. コンパイラがスキーマ・オブジェクトとしてベースを見つけた場合、基本オブジェクトに対する権限がチェックされる。基本オブジェクトが参照可能でない場合、コンパイラは未宣言エラーを生成する。不十分な権限エラーではなくこのエラーが生成される理由は、「不十分な権限」エラーを生成した場合、オブジェクトの存在が確認されてしまい、セキュリティ違反となるため。
5. スキーマ・オブジェクトを検索することによりベースを見つけることができなかった場合、コンパイラは未宣言エラーを生成する。
6. コンパイラがベースを見つけた場合、このベースがどのようにして解決されたかに依存して、参照を完全に解決しようとする。参照全体の解決に失敗した場合、コンパイラはエラーを生成する。

獲得を理解する

別の有効範囲における宣言または型の定義が参照の正常な解決の妨げになる場合、その宣言または定義が参照を「獲得する」と言います。通常、これは移行またはスキーマのアップグレードの結果として生じます。獲得には、内部獲得、同一有効範囲獲得、外部獲得の 3 種類があります。内部および同一有効範囲の獲得は SQL の有効範囲にのみ適用されます。

内部獲得

内部獲得が発生するのは、いったん、外部有効範囲のエンティティに解決した内部有効範囲に含まれる名前に、次のような状態が発生した場合です。

- 内部有効範囲に含まれるエンティティに解決された場合
- 識別子の連鎖が内部有効範囲に獲得され、参照を完全に解決することができなかったため、エラーが発生する場合

この状態が、内部有効範囲でエラーが発生することなく解決された場合、ユーザーの知らない間に獲得が発生している可能性があります。次の例を考えてみます。

```
CREATE TABLE tab1 (col1 NUMBER, col2 NUMBER)
/
CREATE TABLE tab2 (col1 NUMBER)
/
CREATE PROCEDURE proc AS
    CURSOR c1 IS SELECT * FROM tab1
        WHERE EXISTS (SELECT * FROM tab2 WHERE col2 = 10);
BEGIN
    ...
END
/
```

この例では、内側の SELECT 文における col2 への参照は、表 tab2 が col2 という名前の列を持たないため、表 tab1 の列 col2 にバインドされます。次に示すように、表 tab2 に列 col2 を追加した場合を考えます。

```
ALTER TABLE tab2 ADD (col2 NUMBER);
```

この場合には、プロシージャ proc は無効となり、次に使用する際に、自動的に再コンパイルされます。ただし、再コンパイルの際に、tab2 は内部有効範囲にあるため、内側の SELECT 文の col2 は tab2 の列 col2 にバインドされます。したがって、表 tab2 への列 col2 の追加により、col2 への参照は獲得されます。

コレクションやオブジェクト型を使用することにより、さらに多くの内部獲得を実現することができます。次の例を考えてみます。

```
CREATE TYPE type1 AS OBJECT (a NUMBER)
/
CREATE TABLE tab1 (tab2 type1)
/
CREATE TABLE tab2 (x NUMBER)
/
SELECT * FROM tab1 s -- alias with same name as schema name
    WHERE EXISTS (SELECT * FROM s.tab2 WHERE x = s.tab2.a)
        -- note lack of alias
/
```

この例では、s.tab2.a への参照は、問合せの外部有効範囲で参照することのできる表の別名 s を経由して、表 tab1 の列 tab2 の属性 a に解決されます。内側の副問合せに現れる表 s.tab2 に列名 a を追加することを考えます。問合せが処理されると、s.tab2.a への参照がスキーマ s 内の表 tab2 の列 a に解決されるため、内部獲得が発生します。

内部獲得を防止するには、D-9 ページの「[獲得の防止](#)」で説明されたルールに従います。これらのルールに従えば、上の問合せは、以下のように書き換えるべきです。

```
SELECT * FROM s.tab1 p1
WHERE EXISTS (SELECT * FROM s.tab2 p2 WHERE p2.x = p1.tab2.a);
```

同一有効範囲獲得

SQL の有効範囲で、同一有効範囲の獲得が発生するのは、同一有効範囲内の 2 つの表のどちらかに列が追加され、その列がもう一方の表の列と同じ名前を持つ場合です。次の問合せの例と上の例を比較検討してください。

```
PROCEDURE proc IS
  CURSOR c1 IS SELECT * FROM tab1, tab2 WHERE col2 = 10;
```

この例では、問合せ中の col2 への参照は、表 tab1 の列 col2 にバインドされています。col2 という名前の列を表 tab2 に追加した場合、問合せのコンパイルはエラーが発生します。したがって、col2 への参照はエラーにより獲得されます。

外部獲得

外部獲得が発生するのは、過去に内部有効範囲内のエントリに解決されていた内部 有効範囲内の名前が、外部有効範囲に解決された場合です。幸い、SQL と PL/SQL は、外部獲得を防止する設計になっています。

獲得の防止

次のルールを遵守することにより、DML 文における内部獲得を防止できます。

- DML 文内の各表に対して別名を指定する。
- DML 文の全体を通じて、表の別名を一意に保つ。
- 問合せ内で使用されているスキーマ名と合致する表の別名の使用を避ける。
- 列の参照を表の別名で修飾する。

DML 文がユーザー定義のオブジェクト型の列を持つ表を参照している場合には、<schema-name>.<table-name> への参照を修飾しても内部獲得は防止できません。

属性やメソッドへのアクセス

ユーザー定義のオブジェクト型の列により、より多くの内部獲得が発生する可能性があります。問題を最小限に抑えるため、以下のルールがネーム変換アルゴリズムに含まれています。

- 属性およびメソッドへのすべての参照が、表の別名により修飾されている必要がある。したがって、表を参照する場合、その表に保存されているオブジェクトの属性やメソッドを参照するときには、表名に別名を添付する必要がある。以下の例に示すとおり、属性またはメソッドへの列修飾された参照は、これらが接頭辞として表名（またはスキーマおよび表名）を持つ場合には無効となる。

```
CREATE TYPE t1 AS OBJECT (x NUMBER);
CREATE TABLE tb1 (col t1);
SELECT col.x FROM tb1;                -- illegal
SELECT tb1.col.x FROM tb1;           -- illegal
SELECT scott.tb1.col.x FROM scott.tb1; -- illegal
SELECT t.col.x FROM tb1 t;
UPDATE tb1 SET col.x = 10;            -- illegal
UPDATE scott.tb1 SET scott.tb1.col.x=10; -- illegal
UPDATE tb1 t set t.col.x = 1;
DELETE FROM tb1 WHERE tb1.col.x = 10; -- illegal
DELETE FROM tb1 t WHERE t.col.x = 10;
```

- 行の式は、表の別名への参照として解決する必要がある。行の式を REF および VALUE に受け渡したり、UPDATE 文の SET 句に行の式を使用することができる。次に例を示す。

```
CREATE TYPE t1 AS OBJECT (x number);
CREATE TABLE ot1 OF t1;                -- object table
SELECT REF(ot1) FROM ot1;              -- illegal
SELECT REF(o) FROM ot1 o;
SELECT VALUE(ot1) FROM ot1;            -- illegal
SELECT VALUE(o) FROM ot1 o;
DELETE FROM ot1 WHERE VALUE(ot1) = (t1(10)); -- illegal
DELETE FROM ot1 o WHERE VALUE(o) = (t1(10));
UPDATE ot1 SET ot1 = ...                -- illegal
UPDATE ot1 o SET o = ....
```

オブジェクト表に挿入するための以下に示す各方法は有効です。また、列のリストを持たないため、別名は必要とされません。

```
INSERT INTO ot1 VALUES (t1(10)); -- no row expression
INSERT INTO ot1 VALUES (10);     -- no row expression
```

サブプログラムとメソッドのコール

パラメータを持たないサブプログラムをコールする場合、空のパラメータ・リストは付けても付けなくても問題ありません。同様に、PL/SQL の有効範囲では、空のパラメータ・リストはオプションです。ただし、SQL の有効範囲では、パラメータ・リストは必須です。

例 1

```
CREATE FUNCTION func1 RETURN NUMBER AS
  BEGIN
    RETURN 10;
  END;

CREATE PACKAGE pkg1 AS
  FUNCTION func1 RETURN NUMBER;
  PRAGMA RESTRICT_REFERENCES(func1,WNDS,RNDS,WNPS,RNPS);
END;

CREATE PACKAGE BODY pkg1 AS
  FUNCTION func1 RETURN NUMBER IS
  BEGIN
    RETURN 20;
  END;
END;

SELECT func1 FROM dual;
SELECT func1() FROM dual;
SELECT pkg1.func1 FROM dual;
SELECT pkg1.func1() FROM dual;

DECLARE
  x NUMBER;
BEGIN
  x := func1;
  x := func1();
  SELECT func1 INTO x FROM dual;
  SELECT func1() INTO x FROM dual;
  SELECT pkg1.func1 INTO x FROM dual;
  SELECT pkg1.func1() INTO x FROM dual;
END;
```

例 2

```
CREATE OR REPLACE TYPE type1 AS OBJECT (  
    a NUMBER,  
    MEMBER FUNCTION f RETURN number,  
    PRAGMA RESTRICT_REFERENCES (f, WNDS, RNDS, WNPS, RNPS)  
);  
  
CREATE TYPE BODY type1 AS  
    MEMBER FUNCTION f RETURN number IS  
    BEGIN  
        RETURN 1;  
    END;  
END;  
  
CREATE TABLE tab1 (col1 type1);  
INSERT INTO tab1 VALUES (type1(10));  
  
SELECT x.col1.f FROM tab1 x; -- illegal  
SELECT x.col1.f() FROM tab1 x;  
  
DECLARE  
    n NUMBER;  
    y type1;  
BEGIN  
    /* In PL/SQL scopes, an empty parameter list is optional. */  
    n := y.f;  
    n := y.f();  
    /* In SQL scopes, an empty parameter list is required. */  
    SELECT x.col1.f INTO n FROM tab1 x; -- illegal  
    SELECT x.col1.f() INTO n FROM tab1 x;  
    SELECT y.f INTO n FROM tab1 x; -- illegal  
    SELECT y.f() INTO n FROM tab1 x; l  
END;
```

SQL と PL/SQL の比較

SQL と PL/SQL のネーム変換ルールはよく似ています。ただし、いくつかの小さな違いがあります。しかし、獲得防止ルールに従う限り、これらの違いは問題にはなりません。

互換性のため、SQL ルールは PL/SQL と比較して、より許容性の高いものになっています。つまり、そのほとんどがコンテキスト依存な SQL ルールでは、PL/SQL ルールで認識されるよりも多くの状況と DML 文が、有効なものとして認識されます。

E

予約語

この付録では、PL/SQL で使うために予約されている予約語のリストを示します。予約語は、PL/SQL に対して構文上の特別な意味を持ちます。そのため、定数、変数、カーソルなどのプログラム・オブジェクトの名前には使わないでください。これらの単語の中で、アスタリスクの付いたものは SQL の予約語でもあります。そのため、列、表、索引などのスキーマ・オブジェクトの名前には使わないでください。

ALL*	DESC*	ISOLATION	OUT	SQLERRM
ALTER*	DISTINCT*	JAVA	PACKAGE	START*
AND*	DO	LEVEL*	PARTITION	STDDEV
ANY*	DROP*	LIKE*	PCTFREE*	SUBTYPE
ARRAY	ELSE*	LIMITED	PLS_INTEGER	SUCCESSFUL*
AS*	ELSIF	LOCK*	POSITIVE	SUM
ASC*	END	LONG*	POSITIVEN	SYNONYM*
AUTHID	EXCEPTION	LOOP	PRAGMA	SYSDATE*
AVG	EXCLUSIVE*	MAX	PRIOR*	TABLE*
BEGIN	EXECUTE	MIN	PRIVATE	THEN*
BETWEEN*	EXISTS*	MINUS*	PROCEDURE	TIME
BINARY_INTEGER	EXIT	MINUTE	PUBLIC*	TIMESTAMP
BODY	EXTENDS	MLSLABEL*	RAISE	TO*
BOOLEAN	FALSE	MOD	RANGE	TRIGGER*
BULK	FETCH	MODE*	RAW*	TRUE
BY*	FLOAT*	MONTH	REAL	TYPE
CHAR*	FOR*	NATURAL	RECORD	UID*
CHAR_BASE	FORALL	NATURALN	REF	UNION*
CHECK*	FROM*	NEW	RELEASE	UNIQUE*
CLOSE	FUNCTION	NEXTVAL	RETURN	UPDATE*
CLUSTER*	GOTO	NOCOPY	REVERSE	USE
COLLECT	GROUP*	NOT*	ROLLBACK	USER*
COMMENT*	HAVING*	NOWAIT*	ROW*	VALIDATE*
COMMIT	HEAP	NULL*	ROWID*	VALUES*
COMPRESS*	HOURL	NUMBER*	ROWLABEL	VARCHAR*
CONNECT*	IF	NUMBER_BASE	ROWNUM*	VARCHAR2*
CONSTANT	IMMEDIATE*	OCIROWID	ROWTYPE	VARIANCE
CREATE*	IN*	OF*	SAVEPOINT	VIEW*
CURRENT*	INDEX*	ON*	SECOND	WHEN
CURRVAL	INDICATOR	OPAQUE	SELECT*	WHENEVER*
CURSOR	INSERT*	OPEN	SEPERATE	WHERE*
DATE*	INTEGER*	OPERATOR	SET*	WHILE
DAY	INTERFACE	OPTION*	SHARE*	WITH*
DECLARE	INTERSECT*	OR*	SMALLINT*	WORK
DECIMAL*	INTERVAL	ORDER*	SPACE	WRITE
DEFAULT*	INTO*	ORGANIZATION	SQL	YEAR
DELETE*	IS*	OTHERS	SQLCODE	ZONE

索引

記号

< 関係演算子, 2-3, 2-43
<= 関係演算子, 2-4, 2-43
<< ラベルのデリミタ, 2-4
!= 関係演算子, 2-4, 2-43
" 二重引用符で囲んだ識別子のデリミタ, 2-3
%BULK_ROWCOUNT カーソル属性, 4-34
%FOUND カーソル属性, 5-31, 5-35
%ISOPEN カーソル属性, 5-31, 5-35
%NOTFOUND カーソル属性, 5-32
%ROWCOUNT カーソル属性, 5-32, 5-35
%ROWTYPE 属性, 2-30
 構文, 11-138
%TYPE 属性, 2-30
 構文, 11-153
% 属性の標識, 1-6, 2-3
(式またはリストのデリミタ, 2-3
) 式またはリストのデリミタ, 2-3
** 指数演算子, 2-4
* 乗算演算子, 2-3
*/ 複数行コメントのデリミタ, 2-4
/* 複数行コメントのデリミタ, 2-4
+ 加算 / 一致演算子, 2-3
:= 代入演算子, 1-4, 2-4
: ホスト変数の標識, 2-3
; 終結子, 11-11
; 文の終結子, 2-3
=> 結合演算子, 2-4, 7-13
= 関係演算子, 2-3, 2-43
>= 関係演算子, 2-4, 2-43
>> ラベルのデリミタ, 2-4
> 関係演算子, 2-3, 2-43
@ リモート・アクセスの標識, 2-3, 2-33
^= 関係演算子, 2-4

' 文字列のデリミタ, 2-3
. 構成要素の選択子, 1-5, 2-3
/ 除算演算子, 2-3
. 項目の区切り子, 2-3
.. 範囲演算子, 2-4, 3-9
- 減算 / 否定演算子, 2-3
-- 1 行コメントのデリミタ, 2-4
|| 連結演算子, 2-4, 2-44
~= 関係演算子, 2-4, 2-43

数字

1 行コメント, 2-9
2 項演算子, 2-39
3 項演算子, 2-39

A

ACCESS INTO NULL 例外, 6-5
ALL オプション, 5-3
ALL 行演算子, 5-6
ALL 比較演算子, 5-5
ANY 比較演算子, 5-5
AUTHID 句, 7-31
AUTONOMOUS_TRANSACTION プラグマ, 5-49
AVG 集計関数, 5-3

B

BETWEEN 比較演算子, 2-43, 5-5
BFILE データ型, 2-20
BINARY_INTEGER データ型, 2-11
BLOB データ型, 2-20
BOOLEAN データ型, 2-21
BULK COLLECT 句, 4-32

C

CHAR_CS 値, 2-27
CHARACTER サブタイプ, 2-14
CHAR データ型, 2-14
CHAR の意味, B-1
CHAR 列
 最大幅, 2-14
CLOB データ型, 2-20
CLOSE 文, 5-9, 5-21
 構文, 11-14
COLLECTION_IS_NULL 例外, 6-5
COMMENT 句, 5-39
COMMIT 文, 5-38
 構文, 11-27
COUNT コレクション・メソッド, 4-21
COUNT 集計関数, 5-3
CURRENT OF 句, 5-44
CURRVAL 疑似列, 5-3
CURSOR_ALREADY_OPEN 例外, 6-5

D

DATE データ型, 2-21
DBMS_ALERT パッケージ, 8-16
DBMS_OUTPUT パッケージ, 8-16
DBMS_PIPE パッケージ, 8-17
DBMS_STANDARD パッケージ, 8-16
DECIMAL サブタイプ, 2-13
DECODE ファンクション
 NULL の扱い, 2-48
DEC サブタイプ, 2-13
DEFAULT キーワード, 2-29
DELETE コレクション・メソッド, 4-24
DELETE 文
 RETURNING 句, 5-59
 構文, 11-48
DEPT 表, xxi
DEREF 演算子, 9-31
DISTINCT オプション, 5-3
DISTINCT 行演算子, 5-6
DOUBLE PRECISION サブタイプ, 2-13
DUP_VAL_ON_INDEX 例外, 6-5

E

ELSE 句, 3-3
ELSIF 句, 3-4
EMP 表, xxi
END IF 予約語, 3-3
END LOOP 予約語, 3-8
EXAMPBLD スクリプト, A-3
EXAMPLD スクリプト, A-3
EXCEPTION_INIT プラグマ, 6-8
 raise_application_error での使用, 6-9
 構文, 11-52
EXECUTE IMMEDIATE 文, 10-3
EXECUTE 権限, 7-34
EXISTS コレクション・メソッド, 4-20
EXISTS 比較演算子, 5-5
EXIT 文, 3-6, 3-12
 WHEN 句, 3-7
 構文, 11-60
 使える場所, 3-6
EXTEND コレクション・メソッド, 4-23

F

FALSE 値, 2-8
FETCH 文, 5-8, 5-20
 構文, 11-72
FIRST コレクション・メソッド, 4-21
FLOAT サブタイプ, 2-13
FORALL 文, 4-30
 構文, 11-76
 使用、BULK COLLECT 句, 4-33
FOR UPDATE 句, 5-7
 使う場合, 5-43
 の制限, 5-18
FOR ループ, 3-9
 カーソル, 5-12
 動的な範囲, 3-11
 ネスト, 3-12
 反復スキーム, 3-9
 ループ・カウンタ, 3-9

G

GB, 2-20
GOTO 文, 3-13
 構文, 11-82
 制限, 6-17
 間違った使用, 3-15
 ラベル, 3-13
GROUP BY 句, 5-3
GROUPING 集計関数, 5-3

I

IF 文, 3-2
 ELSE 句, 3-3
 ELSIF 句, 3-4
 THEN 句, 3-3
 構文, 11-84
IN OUT パラメータ・モード, 7-16
INSERT 文
 RETURNING 句, 5-59
 構文, 11-87
INTEGER サブタイプ, 2-13
INTERSECT 集合演算子, 5-6
INTO 句, 5-21
INTO リスト, 5-8
INT サブタイプ, 2-13
INVALID_CURSOR 例外, 6-5
INVALID_NUMBER 例外, 6-5
IN パラメータ・モード, 7-14
IN 比較演算子, 2-44, 5-5
IS DANGLING 述語, 9-31
IS NULL 比較演算子, 2-43, 5-5

L

LAST コレクション・メソッド, 4-21
LEVEL 疑似列, 5-4
LIKE 比較演算子, 2-43, 5-5
LIMIT コレクション・メソッド, 4-21
LOB (大規模オブジェクト) データ型, 2-19
lob ロケータ, 2-19
LOCK TABLE 文, 5-44
 構文, 11-93
LOGIN_DENIED 例外, 6-5

LONG RAW データ型, 2-14
 最大長, 2-14
 変換, 2-27
LONG データ型, 2-14
 最大長, 2-14
 制限, 2-15
LOOP 文, 3-6
 形式, 3-6
 構文, 11-95

M

MAX 集計関数, 5-3
MINUS 集合演算子, 5-6
MIN 集計関数, 5-3

N

NATURALN サブタイプ, 2-12
NATURAL サブタイプ, 2-12
NCHAR_CS 値, 2-27
NCHAR データ型, 2-18
NCLOB データ型, 2-21
NEXTVAL 疑似列, 5-3
NEXT コレクション・メソッド, 4-22
NLS_CHARSET_ID ファンクション, 2-27
NLS_CHARSET_NAME ファンクション, 2-27
NLS (各国語サポート), 2-18
NLS データ型, 2-18
NO_DATA_FOUND 例外, 6-5
NOCOPY コンパイラ・ヒント, 7-17
 制限, 7-18
NOT_LOGGED_ON 例外, 6-5
NOT NULL 制約
 %TYPE 宣言の効果, 2-30
 コレクション宣言での使用, 4-6
 制限, 5-7, 7-4
 パフォーマンスへの影響, 5-61
 フィールド宣言での使用, 4-38
 変数宣言で使う, 2-29
NOT 論理演算子
 NULL の扱い, 2-47
NOWAIT パラメータ, 5-43
NULL かどうか, 2-43
NULL の扱い, 2-46
 動的 SQL, 10-12

- NULL 文, 3-16
 - 構文, 11-101
 - プロシージャで使う, 7-4
- NUMBER データ型, 2-12
- NUMERIC サブタイプ, 2-13
- NVARCHAR2 データ型, 2-19
- NVL ファンクション
 - NULL の扱い, 2-48

O

- OPEN-FOR-USING 文, 10-5
 - 構文, 11-115
- OPEN-FOR 文, 5-17
 - 構文, 11-111
- OPEN 文, 5-7
 - 構文, 11-109
- Oracle
 - Trusted, 2-10
- OR キーワード, 6-16
- OTHERS 例外ハンドラ, 6-2, 6-15
- OUT パラメータ・モード, 7-14

P

- PLS_INTEGER データ型, 2-13
- PL/SQL
 - SQL のサポート, 1-19
 - アーキテクチャ, 1-16
 - 移植性, 1-22
 - サンプル・プログラム, A-1
 - 実行環境, 1-16
 - パフォーマンス, 1-20
 - プロシージャ的な面, 1-2
 - ブロック構造, 1-2
 - 予約語, E-1
 - 利点, 1-19
- PLSQL_V2_COMPATIBILITY フラグ, 5-63
- PL/SQL エンジン, 1-16
 - Oracle Server, 1-17
 - Oracle Tools, 1-18
- PL/SQL 構文, 11-1
- PL/SQL コンパイラ
 - コールの解決方法, 7-25

- PL/SQL ブロック
 - 構文, 11-7
 - 最大サイズ, 5-46
 - 無名, 1-2, 7-2
- PL/SQL ラッパー, C-1
 - 実行, C-2
 - 入力ファイルと出力ファイル, C-3
- POSITIVEN サブタイプ, 2-12
- POSITIVE サブタイプ, 2-12
- PRIOR 行演算子, 5-4, 5-6
- PRIOR コレクション・メソッド, 4-22
- PROGRAM_ERROR 例外, 6-5

R

- raise_application_error プロシージャ, 6-9
- RAISE 文, 6-11
 - 構文, 11-128
 - 例外ハンドラでの使用, 6-15
- RAW データ型, 2-15
 - 最大長, 2-15
 - 変換, 2-27
- READ ONLY パラメータ, 5-42
- REAL サブタイプ, 2-13
- RECORD データ型, 4-35
- ref, 9-26
 - 参照解除, 9-31
 - 参照先がない, 9-31
 - 宣言, 9-27
- REF CURSOR データ型, 5-15
 - 定義, 5-15
- REF 演算子, 9-30
- REF 型修飾子, 9-27
- REPEAT UNTIL 構造
 - 疑似実行, 3-8
- REPLACE ファンクション
 - NULL の扱い, 2-48
- RESTRICT_REFERENCES プラグマ, 5-55, 7-7, 10-13
- RETURNING 句, 5-59, 9-34
- RETURN 句
 - カーソル, 5-11
 - ファンクション, 7-5
- RETURN 文, 7-8
 - 構文, 11-134
- REVERSE 予約語, 3-9

ROLLBACK 文, 5-39
 構文, 11-136
 セーブポイントへの影響, 5-40
ROWID, 2-15
 拡張, 2-16
 制限, 2-16
ROWIDTOCHAR ファンクション, 5-4
ROWID 疑似列, 5-4
ROWID データ型, 2-15
ROWNUM 疑似列, 5-4
ROWTYPE_MISMATCH 例外, 6-6
RPC (リモート・プロシージャ・コール), 6-12
RTRIM ファンクション
 データの挿入に使う, B-4

S

SAVEPOINT 文, 5-40
 構文, 11-140
SELECT INTO 文
 構文, 11-141
SELF パラメータ, 9-9
SERIALLY_REUSABLE プラグマ, 5-60
SET TRANSACTION 文, 5-42
 構文, 11-145
SIGNTYPE サブタイプ, 2-12
SMALLINT サブタイプ, 2-13
SOME 比較演算子, 5-5
SQL
 PL/SQL でのサポート, 1-19
 疑似列, 5-3
 行演算子, 5-6
 集合演算子, 5-6
 データ操作文, 5-2
 動的, 10-2
 比較演算子, 5-5
SQLCODE ファンクション, 6-17
 構文, 11-149
SQLERRM ファンクション, 6-17
 構文, 11-151
SQL カーソル
 構文, 11-147
SQL のサポート, 5-2
START WITH 句, 5-4
STDDEV 集計関数, 5-3

STEP 句
 疑似実行, 3-10
STORAGE_ERROR 例外, 6-6
 呼び出される場合, 7-37
STRING サブタイプ, 2-17
SUBSCRIPT_BEYOND_COUNT 例外, 6-6
SUBSCRIPT_OUTSIDE_LIMIT 例外, 6-6
SUBSTR ファンクション, 6-18
SUM 集計関数, 5-3

T

TABLE 演算子, 4-17
TABLE データ型, 4-2
THEN 句, 3-3
TIMEOUT_ON_RESOURCE 例外, 6-6
TOO_MANY_ROWS 例外, 6-6
TRIM コレクション・メソッド, 4-23
TRUE 値, 2-8
Trusted Oracle, 2-10

U

UNION ALL 集合演算子, 5-6
UNION 集合演算子, 5-6
UPDATE 文
 RETURNING 句, 5-59
 構文, 11-155
URL (汎用リソース・ロケータ), 8-17
UROWID データ型, 2-15
USING 句, 10-3
UTL_FILE パッケージ, 8-17
UTL_HTTP パッケージ, 8-17

V

VALUE_ERROR 例外, 6-6
VALUE 演算子, 9-30
VARCHAR2 データ型, 2-17
VARCHAR2 の意味, B-1
VARCHAR サブタイプ, 2-17
VARIANCE 集計関数, 5-3
varray
 サイズの制限, 4-5
VARRAY データ型, 4-4

W

WHEN 句, 3-7, 6-15
WHILE ループ, 3-8

Z

ZERO_DIVIDE 例外, 6-6

あ

アーキテクチャ, 1-16
アスタリスク (*) オプション, 5-3
値方式
 パラメータの受け渡し, 7-21
扱い
 NULL の, 2-46
アトミック NULL, 9-22
アドレス, 5-15
アポストロフィ, 2-8
アンダースコア, 2-4
暗黙カーソル, 5-10
 属性, 5-35
暗黙の宣言
 FOR ループ・カウンタ, 3-11
 カーソル FOR ループのレコード, 5-12
暗黙のデータ型変換, 2-25
 パフォーマンスへの影響, 5-63

い

移植性, 1-22
位置表記法, 7-13
一括代入, 2-31
意味のある文字数, 2-5
インスタンス, 9-4
隠べい
 情報, 1-14

え

エイリアシング, 7-21
エラー・メッセージ
 最大長, 6-17
エラーレクション, 2-28

演算子

DEREF, 9-31
REF, 9-30
TABLE, 4-17
VALUE, 9-30
関係, 2-43
比較, 2-42
優先順位, 2-40
連結, 2-44

お

オーバーロード, 7-23
 オブジェクト・メソッド, 9-10
 サブタイプを使う, 7-24
 制限, 7-24
 パッケージ・サブプログラム, 8-14
大文字 / 小文字の区別
 識別子, 2-5
 文字列リテラル, 2-8
オブジェクト, 9-4
 共有, 9-26
 初期化, 9-22
 宣言, 9-21
 操作, 9-28
オブジェクト型, 9-1, 9-3
 構造, 9-5
 構文, 11-102
 定義, 9-12
 利点, 9-5
 例, 9-12
オブジェクト・コンストラクタ
 コール, 9-24
 パラメータを渡す, 9-24
オブジェクト指向プログラミング, 9-1
オブジェクト属性, 9-3, 9-7
 アクセス, 9-23
 最大数, 9-7
 使えるデータ型, 9-7
オブジェクト表, 9-29
オブジェクト・メソッド, 9-3, 9-8
 コール, 9-25

か

- カーソル, 1-4, 5-6
 - RETURN 句, 5-11
 - 暗黙, 5-10
 - オープン, 5-7
 - クローズ, 5-9
 - 構文, 11-44
 - 宣言, 5-6
 - 取出し, 5-8
 - パッケージ, 5-11
 - パラメータ付き, 5-8
 - 明示, 5-6
 - 有効範囲の規則, 5-7
 - 類似点, 1-5
- カーソル FOR ループ, 5-12
 - パラメータを渡す, 5-13
- カーソル属性
 - %BULK_ROWCOUNT, 4-34
 - %FOUND, 5-31, 5-35
 - %ISOPEN, 5-31, 5-35
 - %NOTFOUND, 5-32
 - %ROWCOUNT, 5-32, 5-35
 - 値, 5-33
 - 暗黙, 5-35
 - 構文, 11-33
- カーソル変数, 5-14
 - オープン, 5-17
 - クローズ, 5-21
 - 構文, 11-38
 - 使用、動的 SQL での, 10-5
 - 制限, 5-30
 - 宣言, 5-16
 - 代入, 5-28
 - 取出し, 5-20
 - ネットワーク通信量を少なくするために使う, 5-27
- 改行, 2-3
- 外部参照, 7-31
 - 解決方法, 7-32
- 外部ルーチン, 7-27
- 科学表記法, 2-7
- 隠された宣言, 8-3
- 拡張 ROWID, 2-16
- 拡張性, 7-3
- 格納表, 4-4

- 可視性
 - トランザクション, 5-51
 - パッケージの内容, 8-3
 - 有効範囲, 2-35
- 型の拡張, 9-7
- 型の定義
 - RECORD, 4-36
 - REF CURSOR, 5-15
 - コレクション, 4-5
 - 前方, 9-28
- カッコ, 2-40
- 各国キャラクタ・セット, 2-18
- 各国語サポート (NLS), 2-18
- カプセル化
 - データ, 1-14
- 仮パラメータ, 5-8
- 関係演算子, 2-4, 2-43
- 間接参照, 9-31

き

- 記号
 - 単純, 2-3
 - 複合, 2-4
- 疑似命令, 6-8
- 疑似列, 5-3
 - CURRVAL, 5-3
 - LEVEL, 5-4
 - NEXTVAL, 5-3
 - ROWID, 5-4
 - ROWNUM, 5-4
- 規則
 - 純正, 7-7
 - 命名, 2-33
- 基本型, 2-12, 2-22
- 基本ループ, 3-6
- キャラクタ・セット, 2-2
- 行 ID
 - 推測, 2-16
 - 汎用, 2-15
 - 物理, 2-15
 - 論理, 2-15
- 行演算子, 5-6
- 行ロック, 5-43

く

句

- AUTHID, 7-31
- BULK COLLECT, 4-32

空白

- 使える場所, 2-2
- 空白埋めの方法, B-2
- 区切り子, 2-3
- 組込みファンクション, 2-49
- クライアント・プログラム, 9-2
- 位取り
 - 指定, 2-13

け

形式

- パッケージ・プロシージャ, 7-10
- ファンクション, 7-5
- プロシージャ, 7-3

桁数

- 精度, 2-12
- 結果セット, 1-4, 5-7
- 結果の値
 - ファンクション, 7-5
- 結合, 7-39
- 結合演算子, 7-13
- 言語間のコール, 7-27
- 現在行, 1-4

こ

語

- 予約, E-1
- 構成要素の選択子, 1-5
- 構造定理, 3-2
- 後続する空白
 - 処理方法, B-3
- 構文図
 - 読み方, 11-2
- 構文定義, 11-1
- コール
 - 言語間, 7-27
 - サブプログラム, 7-13

コール仕様部, 8-3

コメント, 2-9

構文, 11-26

制限, 2-10

コレクション, 4-2

構文, 11-21

コンストラクタ, 4-8

参照, 4-10

初期化, 4-8

宣言, 4-7

代入, 4-11

定義, 4-5

バルク・バインド, 4-27

比較, 4-12

有効範囲, 4-8

要素型, 4-5

コレクション型, 4-1

コレクションの例外

呼び出される場合, 4-26

コレクション・メソッド

COUNT, 4-21

DELETE, 4-24

EXISTS, 4-20

EXTEND, 4-23

FIRST, 4-21

LAST, 4-21

LIMIT, 4-21

NEXT, 4-22

PRIOR, 4-22

TRIM, 4-23

構文, 11-16

パラメータに適用, 4-25

混合表記法, 7-13

コンストラクタ

オブジェクト, 9-12

コレクション, 4-8

コンテキスト

トランザクション, 5-51

コンテキスト切替え, 4-28

コンパイラ・ヒント

NOCOPY, 7-17

さ

サーバー

PL/SQL の統合, 1-22

再帰

終了条件, 7-37

相互, 7-40

反復との比較, 7-41

無限, 7-37

再使用可能パッケージ, 5-60

サイズ制約

サブタイプ, 2-23

サイズの制限

varray, 4-5

最大サイズ

LOB, 2-19

最大精度, 2-12

最大長

CHAR 値, 2-14

LONG RAW 値, 2-14

LONG 値, 2-14

NCHAR 値, 2-18

NVARCHAR2 値, 2-19

Oracle エラー・メッセージ, 6-17

RAW 値, 2-15

VARCHAR2 値, 2-17

識別子, 2-5

再呼出し

例外, 6-14

再利用性, 7-3

作業域

問合せ, 5-15

索引

カーソル FOR ループ, 5-12

索引付き表

ネストした表との比較, 4-3

サブタイプ, 2-12, 2-22

CHARACTER, 2-14

DEC, 2-13

DECIMAL, 2-13

DOUBLE PRECISION, 2-13

FLOAT, 2-13

INT, 2-13

INTEGER, 2-13

NATURAL, 2-12

NATURALN, 2-12

NUMERIC, 2-13

POSITIVE, 2-12

POSITIVEN, 2-12

REAL, 2-13

SIGNTYPE, 2-12

SMALLINT, 2-13

STRING, 2-17

VARCHAR, 2-17

オーバーロード, 7-24

互換性, 2-24

定義, 2-22

サブプログラム, 7-2

オーバーロード, 7-23

コールの解決方法, 7-25

再帰, 7-37

スタンドアロン, 1-17

ストアド, 1-17, 7-11

宣言, 7-9

パッケージ, 1-17, 7-10

部分, 7-2

プロシージャとファンクションの比較, 7-5

利点, 7-3

ローカル, 1-17

参照

外部, 7-31

参照型, 2-10

参照先がない REF, 9-31

参照方式

パラメータの受け渡し, 7-21

サンプル・データベース表

DEPT 表, xxi

EMP 表, xxi

サンプル・プログラム, A-1

し

式

カッコ, 2-40

構文, 11-62

評価の方法, 2-39

ブール, 2-44

識別子

構成, 2-4

最大長, 2-5

二重引用符で囲んだ, 2-6

有効範囲の規則, 2-35

字句単位, 2-2

システム固有の動的 SQL、「動的 SQL」を参照

- 事前定義の例外
 - 再宣言, 6-10
 - 明示的な呼出し, 6-11
 - リスト, 6-4
- 実行可能部分
 - PL/SQL ブロック, 1-3
 - ファンクション, 7-6
 - プロシージャ, 7-4
- 実行環境, 1-16
- 実行時エラー, 6-1
- 実行者権限, 7-29
 - 定義者権限との比較, 7-29
 - 利点, 7-29
- 実パラメータ, 5-8
- 集計関数
 - AVG, 5-3
 - COUNT, 5-3
 - GROUPING, 5-3
 - MAX, 5-3
 - MIN, 5-3
 - NULL の扱い, 5-3
 - STDDEV, 5-3
 - SUM, 5-3
 - VARIANCE, 5-3
- 終結子
 - 文, 2-3
- 集合演算子, 5-6
- 修飾子
 - サブプログラム名を使う, 2-35
 - 必要な場合, 2-33, 2-37
- 終了条件, 7-37
- 述語, 5-5
- 順次制御, 3-13
- 順序, 5-3
 - 評価の, 2-40, 2-41
- 順序付けメソッド, 9-10
- 純正レベルの規則, 7-7
- 条件制御, 3-2
- 照合順番, 2-45
- 仕様部
 - オブジェクト, 9-5
 - カーソル, 5-11
 - パッケージ, 8-5
 - ファンクション, 7-6
 - プロシージャ, 7-4
 - メソッド, 9-8
- 情報隠ぺい, 1-14, 8-4

- 初期化
 - DEFAULT を使う, 2-29
 - オブジェクト, 9-22
 - コレクション, 4-8
 - パッケージ, 8-8
 - 必要な場合, 2-29
 - 変数, 2-38
 - レコード, 4-38
- 書式マスク
 - 必要な場合, 2-27
- 自律型トランザクション, 5-48
 - 制御, 5-52
 - 利点, 5-48

す

- 推測, 2-16
- 数値リテラル, 2-7
- スカラー型, 2-10
- スキーム
 - 反復, 3-9
- スタック, 9-13
- スタブ, 3-17, 7-3
- スタンドアロン・サブプログラム, 1-17
- ストアド・サブプログラム, 1-17, 7-11
- スナップショット, 5-38
- スバゲティ・コード, 3-13

せ

- 制御構造, 3-2
 - 順次, 3-13
 - 条件, 3-2
 - 反復, 3-6
- 制限 ROWID, 2-16
- 生産性, 1-21
- 精度桁数
 - 指定, 2-12
- 制約
 - NOT NULL, 2-29
 - 使用できない場合, 2-23, 7-4
- セーブポイント名
 - 再利用, 5-41
- セッション, 5-37
- セッション固有の変数, 8-11
- 遷移
 - 例外, 6-12

宣言

- オブジェクト, 9-21
- カーソル, 5-6
- カーソル変数, 5-16
- コレクション, 4-7
- サブプログラム, 7-9
- 前方, 7-9
- 定数, 2-28
- 変数, 2-28
- 例外, 6-7
- レコード, 4-37

宣言部

- PL/SQL ブロック, 1-3
- ファンクション, 7-6
- プロシージャ, 7-4

選択子, 5-19

前方型定義, 9-28

前方参照, 2-33

前方宣言, 7-9

- 必要な場合, 7-9, 7-40

そ

相関副問合せ, 5-10

相互再帰, 7-40

相互操作性

- カーソルの, 5-15

属性, 1-6

- %ROWTYPE, 2-30

- %TYPE, 2-30

- オブジェクト, 9-3, 9-7

- カーソル, 5-31

属性の標識, 1-6

疎コレクション, 4-3

た

大規模オブジェクト (LOB) データ型, 2-19

代入

- 一括, 2-31

- カーソル変数, 5-28

- コレクション, 4-11

- フィールド, 4-40

- レコード, 4-40

代入演算子, 1-4

代入文

- 構文, 11-3

タブ, 2-3

段階的詳細化, 1-2

単項演算子, 2-39

単純記号, 2-3

短絡評価, 2-42

ち

逐次再使用可能パッケージ, 5-60

抽象化, 7-3, 9-2

て

定義者権限, 7-29

- 実行者権限との比較, 7-29

定数

- 構文, 11-29

- 宣言, 2-28

データ型, 2-10

- BFILE, 2-20

- BINARY_INTEGER, 2-11

- BLOB, 2-20

- BOOLEAN, 2-21

- CHAR, 2-14

- CLOB, 2-20

- DATE, 2-21

- LONG, 2-14

- LONG RAW, 2-14

- NCHAR, 2-18

- NCLOB, 2-21

- NLS, 2-18

- NUMBER, 2-12

- NVARCHAR2, 2-19

- PLS_INTEGER, 2-13

- RAW, 2-15

- RECORD, 4-35

- REF CURSOR, 5-15

- ROWID, 2-15

- TABLE, 4-2

- UROWID, 2-15

- VARCHAR2, 2-17

- VARRAY, 4-4

- 暗黙の変換, 2-25

- グループ, 2-10

- スカラーと複合, 2-10

- 制約, 7-4

データ抽象化, 9-2

データのカプセル化, 1-14
データの整合性, 5-37
データベース・キャラクタ・セット, 2-18
データベースの変更
 永久的なものにする, 5-38
 やり直す, 5-39
データ・ロック, 5-37
手続き抽象化, 9-2
デッドロック, 5-37
 解消方法, 5-40
 トランザクションへの影響, 5-40
デフォルトのパラメータ値, 7-19
デリミタ, 2-3

と

問合せ作業域, 5-15
動的 SQL, 10-2
 扱い、NULL, 10-12
 システム固有の, 10-1
 使用、スキーマ・オブジェクトの名前, 10-10
 使用、重複するブレースホルダ, 10-10
 使用方法、EXECUTE IMMEDIATE 文, 10-3
 使用方法、OPEN-FOR-USING 文, 10-5
動的な範囲
 FOR ループ, 3-11
ドット表記法, 1-5, 1-6
 オブジェクト属性, 9-23
 オブジェクト・メソッド, 9-25
 グローバル変数, 3-12
 コレクション・メソッド, 4-20
 パッケージ内容, 8-6
 レコード・フィールド, 2-31
トップダウン設計, 1-14
トランザクション, 5-2
 コミット, 5-38
 コンテキスト, 5-51
 自律型, 5-48
 適切な終了, 5-41
 分散, 5-38
 読み込み専用, 5-42
 ロールバック, 5-39
トランザクション処理, 5-2, 5-37
トランザクションの可視性, 5-51

な

名前
 カーソル, 5-7
 修飾, 2-33
 セーブポイント, 5-41
 変数, 2-34
名前表記法, 7-13

に

二重引用符で囲んだ識別子, 2-6
ニブル, 2-27

ね

ネーム変換, 2-34, D-1
ネスト
 FOR ループ, 3-12
 オブジェクト, 9-7
 ブロック, 1-2
 レコード, 4-36
ネストした表
 索引付き表との比較, 4-3
 操作, 4-13
ネットワークの通信量
 少なくする, 1-21

は

ハイパー・テキスト転送プロトコル (HTTP), 8-17
ハイパー・テキスト・マークアップ言語 (HTML),
 8-17
パイプ, 8-17
バインド, 4-28
バインド変数, 5-14
パターン一致, 2-43
パッケージ, 8-1, 8-2
 構文, 11-118
 参照, 8-6
 仕様部, 8-2
 初期化, 8-8
 製品固有の, 8-16
 逐次再使用可能, 5-60
 プライベート・オブジェクトとパブリック・オブ
 ジェクトの比較, 8-14
 本体, 8-2

- 本体なし, 8-6
- 有効範囲, 8-5
- 利点, 8-4
- パッケージ・カーソル, 5-11
- パッケージ・サブプログラム, 1-17, 7-10
 - オーバーロード, 8-14
 - コール, 8-6
- パフォーマンス, 1-20
 - 向上, 5-56
- パブリック・オブジェクト, 8-14
- パラメータ
 - SELF, 9-9
 - カーソル, 5-8
 - 実～と仮～の対比, 7-12
 - デフォルト値, 7-19
 - モード, 7-14
- パラメータの受け渡し
 - 値方式, 7-21
 - 参照方式, 7-21
 - 動的 SQL, 10-9
- パラメータのエイリアシング, 7-21
- バルク・バインド, 4-27
- バルク・フェッチ, 4-32
- 範囲演算子, 3-9
- ハンドラ
 - 例外, 6-2
- 反復
 - 再帰との比較, 7-41
 - スキーム, 3-9
- 反復制御, 3-6
- 汎用行 ID, 2-15

ひ

- 比較
 - 演算子, 2-42, 5-5
 - コレクションの, 4-12
 - 式の比較, 2-44
 - 文字値, B-2
- 非空白埋めの方法, B-3
- 日付
 - TO_CHAR デフォルト書式, 2-27
 - 変換, 2-27
- 非同期操作, 8-16
- 評価, 2-39
 - 短絡, 2-42

- 表記法
 - 位置～と名前～の対比, 7-13
 - 混合, 7-13
- ヒント
 - NOCOPY, 7-17

ふ

- ファイル I/O, 8-17
- ファンクション, 7-1, 7-5
 - RETURN 句, 7-5
 - 組込み, 2-49
 - 構文, 11-78
 - コール, 7-6
 - 仕様部, 7-6
 - パラメータ, 7-5
 - 部分, 7-5
 - 本体, 7-6
- フィールド, 4-35
- フィールド型, 4-36
- フィボナッチ数列, 7-37
- ブール
 - 値, 2-44
 - 式, 2-44
 - リテラル, 2-8
- フェッチ
 - バルク, 4-32
 - 複数のコミット, 5-44
- 不完全なオブジェクト型, 9-28
- 複合型, 2-10
- 複合記号, 2-4
- 副作用, 7-14
 - 制御, 7-7
- 複数行コメント, 2-9
- 副問合せ, 5-10
- 物理行 ID, 2-15
- プライベート・オブジェクト, 8-14
- フラグ
 - PLSQL_V2_COMPATIBILITY, 5-63
- プラグマ, 6-8
 - AUTONOMOUS_TRANSACTION, 5-49
 - EXCEPTION_INIT, 6-8
 - RESTRICT_REFERENCES, 5-55, 7-7, 10-13
 - SERIALLY_REUSABLE, 5-60
- ブレースホルダ, 10-2
 - 重複, 10-10
- プログラム・ユニット, 1-11

プロシージャ , 7-1 , 7-3

構文 , 11-123

コール , 7-5

仕様部 , 7-4

パラメータ , 7-3

部分 , 7-4

本体 , 7-4

ブロック

PL/SQL , 11-7

構造 , 1-2

最大サイズ , 5-46

無名 , 7-2

ラベル , 2-37

文

CLOSE , 5-9 , 5-21 , 11-14

COMMIT , 11-27

DELETE , 11-48

EXECUTE IMMEDIATE , 10-3

EXIT , 11-60

FETCH , 5-8 , 5-20 , 11-72

FORALL , 4-30

GOTO , 11-82

IF , 11-84

INSERT , 11-87

LOCK TABLE , 11-93

LOOP , 11-95

NULL , 11-101

OPEN , 5-7 , 11-109

OPEN-FOR , 5-17 , 11-111

OPEN-FOR-USING , 10-5

RAISE , 11-128

RETURN , 11-134

ROLLBACK , 11-136

SAVEPOINT , 11-140

SELECT INTO , 11-141

SET TRANSACTION , 11-145

UPDATE , 11-155

代入 , 11-3

動的 SQL , 10-2

分散トランザクション , 5-38

文の終結子 , 11-11

文レベルのロールバック , 5-40

へ

並行性 , 5-37

変換

データ型 , 2-25

ネーム , 2-34 , D-1

変換ファンクション

必要な場合 , 2-26

変数

値の代入 , 2-38

構文 , 11-29

初期化 , 2-38

セッション固有 , 8-11

宣言 , 2-28

ほ

ポインタ , 5-15

方法

CHAR と VARCHAR2 の比較 , B-1

空白埋め , B-2

非空白埋め , B-3

文字列の比較 , B-2

割当て , B-2

ホスト配列

バルク・バインド , 4-34

ホスト変数 , 5-14

本体

オブジェクト , 9-5

カーソル , 5-11

パッケージ , 8-7

ファンクション , 7-6

プロシージャ , 7-4

メソッド , 9-8

ま

マップ・メソッド , 9-10

み

未初期化オブジェクト

処理方法 , 9-22

未処理例外 , 6-12 , 6-19

密コレクション , 4-3

見やすくする , 2-2 , 3-16

む

無限ループ, 3-6
無名 PL/SQL ブロック, 7-2

め

明示カーソル, 5-6
命名規則, 2-33
メソッド
 COUNT, 4-21
 DELETE, 4-24
 EXISTS, 4-20
 EXTEND, 4-23
 FIRST, 4-21
 LAST, 4-21
 LIMIT, 4-21
 NEXT, 4-22
 PRIOR, 4-22
 TRIM, 4-23
 オブジェクト, 9-3, 9-8
 コレクション, 4-20
 順序付け, 9-10
 マップ, 9-10
メソッドのコール
 連鎖, 9-25
メンテナンス性, 7-3
メンバーかどうかのテスト, 2-44

も

モード、パラメータ
 IN, 7-14
 IN OUT, 7-16
 OUT, 7-14
文字値
 選択, B-4
 挿入, B-3
 比較, B-2
 割当て, B-2
モジュール性, 1-11, 7-3, 8-4
文字リテラル, 2-8
文字列の比較方法, B-2
文字列リテラル, 2-8
戻り型, 5-15, 7-24

ゆ

有効範囲, 2-35
 カーソル, 5-7
 カーソル・パラメータ, 5-7
 コレクション, 4-8
 識別子, 2-35
 定義, 2-35
 パッケージ, 8-5
 ループ・カウンタ, 3-11
 例外, 6-7
ユーザー・セッション, 5-37
ユーザー定義のサブタイプ, 2-22
ユーザー定義の例外, 6-7
ユーザー定義のレコード, 4-35
 参照, 4-38
 宣言, 4-37
優先順位
 演算子, 2-40

よ

要素型
 コレクション, 4-5
呼出し
 例外, 6-11
読み込み一貫性, 5-38
読み込み専用トランザクション, 5-42
予約語, E-1
 二重引用符で囲んだ識別子として使う, 2-6
 間違った使用, 2-5

ら

ライブラリ, 8-1
ラッパー・ユーティリティ, C-1
ラベル
 GOTO 文, 3-13
 ブロック, 2-37
 ループ, 3-7

り

- リテラル, 2-7
 - 構文, 11-90
 - 数値, 2-7
 - ブール, 2-8
 - 文字, 2-8
 - 文字列, 2-8
- リモート・アクセスの標識, 2-33

る

- ルーチン
 - 外部, 7-27
- ループ
 - カウンタ, 3-9
 - 種類, 3-6
 - ラベル, 3-7

れ

- 例外, 6-2
 - RAISE 文での呼出し, 6-11
 - WHEN 句, 6-15
 - 構文, 11-54
 - 再呼出し, 6-14
 - 事前定義, 6-4
 - 遷移, 6-12
 - 宣言, 6-7
 - 宣言の中で呼び出される, 6-16
 - ハンドラで呼び出される, 6-16
 - 有効範囲の規則, 6-7
 - ユーザー定義, 6-7
- 例外処理, 6-1
 - OTHERS ハンドラの使用, 6-15
 - 宣言の中で呼び出される, 6-16
 - ハンドラで呼び出される, 6-16
- 例外処理部
 - PL/SQL ブロック, 1-3
 - ファンクション, 7-6
 - プロシージャ, 7-4
- 例外ハンドラ, 6-15
 - OTHERS ハンドラ, 6-2
 - RAISE 文の使用, 6-15
 - SQLCODE ファンクションの使用, 6-17
 - SQLERRM ファンクションの使用, 6-17
 - 分岐, 6-17

- レコード, 4-35
 - %ROWTYPE, 5-12
 - 暗黙の宣言, 5-12
 - 構文, 11-130
 - 参照, 4-38
 - 初期化, 4-38
 - 宣言, 4-37
 - 操作, 4-42
 - 代入, 4-40
 - 定義, 4-36
 - ネスト, 4-36
 - 比較, 4-42
- 列の別名, 5-13
 - 必要な場合, 2-32
- 連結演算子, 2-44
 - NULL の扱い, 2-48

ろ

- ローカル・サブプログラム, 1-17
- ロールバック
 - FORAL 文, 4-31
 - 暗黙, 5-41
 - 文レベル, 5-40
- ロールバック・セグメント, 5-38
- ロケータ変数, 6-21
- ロック, 5-37
 - FOR UPDATE 句の使用, 5-43
 - 上書き, 5-43
 - モード, 5-37
- 論理行 ID, 2-15

わ

- ワイルドカード, 2-43
- 割当て
 - 方法, B-2
 - 文字列, B-2