PL/SQL

ユーザーズ・ガイドおよびリファレンス

リリース 2（9.2）

2002 年 7 月

部品番号：J06255-01

ORACLE®

PL/SQL ユーザーズ・ガイドおよびリファレンス , リリース 2（9.2）

部品番号：J06255-01

原本名：PL/SQL User's Guide and Reference, Release 2 (9.2)

原本部品番号：A96624-01

原本著者：John Russell

原本協力者：Tom Portfolio, Shashaanka Agrawal, Cailein Barclay, Dmitri Bronnikov, Sharon Castledine, Thomas Chang, Ravindra Dani, Chandrasekharan Iyer, Susan Kotsovolos, Neil Le, Warren Li, Chris Racicot, Murali Vemulapati, Guhan Viswanathan, Minghui Yang

目次

2 PL/SQL の基礎

3 PL/SQL のデータ型

4 PL/SQL の制御構造

5 PL/SQL のコレクションとレコード

6 PL/SQL と Oracle の相互作用

7 PL/SQL エラーの処理

8 PL/SQL のサブプログラム

9 PL/SQL パッケージ

10 PL/SQL のオブジェクト型

11 システム固有の動的 SQL

12 PL/SQL アプリケーションのチューニング

13 PL/SQL の言語要素

A PL/SQL のサンプル・プログラム

B CHAR と VARCHAR2 の意味の比較

C PL/SQL ラップ・ユーティリティ

D PL/SQL の名前解決

E PL/SQL のプログラム上の制限

F PL/SQL の予約語のリスト

索引

はじめに

SQL を手続き型言語として機能拡張した、オラクル社の 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、情報隠ぺいなどの機能が提供されます。また、シームレスな SQL アクセス、Oracle サーバーおよび Oracle のツール製品との緊密な統合、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 について学ぶことができます。

この章の項目は、次のとおりです。

対象読者

このマニュアルは、Oracle の PL/SQL ベースのアプリケーション開発者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ため、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。このマニュアルを効果的に使用するには、Oracle、SQL および Ada、C、COBOL などの 3GL 言語について理解し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、使用し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』の構成は、次のとおりです。

第 1 章「PL/SQL の概要」

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「PL/SQL の基礎」

この章では、PL/SQL の詳細について説明します。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「PL/SQL のデータ型」

この章では、整数、浮動小数点、文字、ブール、日付、コレクション、参照および LOB など、PL/SQL の事前定義のデータ型について説明します。またユーザー定義のサブタイプおよびデータ変換についても説明します。

第 4 章「PL/SQL の制御構造」

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`、`CASE` および `WHILE-LOOP` などの単純で強力な制御構造の使用方法を学ぶことができます。

第 5 章「PL/SQL のコレクションとレコード」

この章では、コンポジット・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 6 章「PL/SQL と Oracle の相互作用」

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 7 章「PL/SQL エラーの処理」

この章では、エラーの報告とリカバリ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 8 章「PL/SQL のサブプログラム」

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 9 章「PL/SQL 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。作成し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有できます。

第 10 章「PL/SQL のオブジェクト型」

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 11 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 12 章「PL/SQL アプリケーションのチューニング」

この章では、PL/SQL ベースのアプリケーションのチューニング方法と、小規模な調整を行ってパフォーマンスを改善する方法について説明します。

第 13 章「PL/SQL の言語要素」

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 文を作成する方法を示します。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「PL/SQL のサンプル・プログラム」

この付録では、独自のプログラムを作成する上で参考になる 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、ベース型 CHAR と VARCHAR2 の微妙ではありますが重要な意味上の相違点を説明します。

付録 C「PL/SQL ラップ・ユーティリティ」

この付録では、ラップ・ユーティリティの実行方法について説明します。ラップ・ユーティリティは、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できるようにするスタンドアロン・プログラミング・ユーティリティです。

付録 D「PL/SQL の名前解決」

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「PL/SQL のプログラム上の制限」

この付録では、PL/SQL コンパイルおよび実行時システムによって課されるプログラム上の制限に対応する方法を説明します。

付録 F「PL/SQL の予約語のリスト」

この付録では、PL/SQL で使用するために予約されている予約語のリストを示します。

関連文書

詳細は、次の Oracle マニュアルを参照してください。

PL/SQL プログラミングの様々な側面、特にトリガーとストアド・プロシージャの詳細は、『Oracle9*i* 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。

PL/SQL 機能と SQL 機能の両方を使用するオブジェクト指向のプログラミングの詳細は、『Oracle9*i* アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』を参照してください。

ラージ・オブジェクト（Large Object: LOB）を使用したプログラミングについては、『Oracle9*i* アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。

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

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

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

OTN-J のユーザー名とパスワードを取得済みであれば、次の OTN-J Web サイトの文書セクションに直接接続できます。

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

表記規則

このマニュアル・セットの本文とコード例に使用されている表記規則について説明します。

- 本文の表記規則
- コード例の表記規則

本文の表記規則

本文中には、特別な用語が一目でわかるように様々な表記規則が使用されています。次の表は、本文の表記規則と使用例を示しています。

規則	意味	例
太字	太字は、本文中に定義されている用語または用語集に含まれている用語、あるいはその両方を示します。	この句を指定する場合は、**索引構成表**を作成します。
固定幅フォントの大文字	固定幅フォントの大文字は、システムにより指定される要素を示します。パラメータ、権限、データ型、Recovery Manager キーワード、SQL キーワード、SQL*Plus またはユーティリティ・コマンド、パッケージとメソッド、システム指定の列名、データベース・オブジェクトと構造体、ユーザー名、およびロールがあります。	この句は、NUMBER 列に対してのみ指定できます。 BACKUP コマンドを使用すると、データベースのバックアップを作成できます。 USER_TABLES データ・ディクショナリ・ビューの TABLE_NAME 列を問い合せます。 DBMS_STATS.GENERATE_STATS プロシージャを使用します。

規則	意味	例
固定幅フォントの小文字	固定幅フォントの小文字は、実行可能ファイル、ファイル名、ディレクトリ名およびサンプルのユーザー指定要素を示します。この要素には、コンピュータ名とデータベース名、ネット・サービス名、接続識別子の他、ユーザー指定のデータベース・オブジェクトと構造体、列名、パッケージとクラス、ユーザー名とロール、プログラム・ユニットおよびパラメータ値があります。 **注意 :** 一部のプログラム要素には、大文字と小文字の両方が使用されます。この場合は、記載されているとおりに入力してください。	sqlplus と入力して SQL*Plus を開きます。 パスワードは orapwd ファイルに指定されています。 データ・ファイルと制御ファイルのバックアップを /disk1/oracle/dbs ディレクトリに作成します。 department_id、department_name および location_id の各列は、hr.departments 表にあります。 初期化パラメータ QUERY_REWRITE_ENABLED を true に設定します。 oe ユーザーで接続します。 これらのメソッドは JRepUtil クラスに実装されます。
固定幅フォントの小文字のイタリック	固定幅フォントの小文字のイタリックは、プレースホルダまたは変数を示します。	*parallel_clause* を指定できます。 U*old_release*.SQL を実行します。*old_release* は、アップグレード前にインストールしたリリースです。

コード例の表記規則

コード例は、SQL、PL/SQL、SQL*Plus または他のコマンドラインを示します。次のように、固定幅フォントで表され、通常の本文とは区別して記載されています。

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

次の表は、コード例の記載上の表記規則と使用例を示しています。

規則	意味	例
[ ]	大カッコで囲まれている項目は、1 つ以上のオプション項目を示します。大カッコ自体は入力しないでください。	DECIMAL (*digits* [, *precision*])
{ }	中カッコで囲まれている項目は、そのうちの 1 つのみが必要であることを示します。中カッコ自体は入力しないでください。	{ENABLE | DISABLE}
|	縦線は、大カッコまたは中カッコ内の複数の選択肢を区切るために使用します。オプションのうち 1 つを入力します。縦線自体は入力しないでください。	{ENABLE | DISABLE} [COMPRESS | NOCOMPRESS]

規則	意味	例
. . .	水平の省略記号は、次のどちらかを示します。	
	■ 例に直接関係のないコード部分が省略されていること。	`CREATE TABLE ... AS subquery;`
	■ コードの一部が繰返し可能であること。	`SELECT col1, col2, ... , coln FROM employees;`
. . .	垂直の省略記号は、例に直接関係のない数行のコードが省略されていることを示します。	
その他の表記	大カッコ、中カッコ、縦線および省略記号以外の記号は、示されているとおりに入力してください。	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
イタリック	イタリックの文字は、特定の値を指定する必要のあるプレースホルダまたは変数を示します。	`CONNECT SYSTEM/system_password` `DB_NAME = database_name`

大文字	大文字は、システムにより提供される要素を示します。これらの用語は、ユーザー定義用語と区別するために大文字で記載されています。大カッコで囲まれている場合を除き、記載されているとおりの順序とスペルで入力してください。ただし、この種の用語は大 / 小文字区別がないため、小文字でも入力できます。	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

小文字	小文字は、ユーザー指定のプログラム要素を示します。たとえば、表名、列名またはファイル名を示します。
	注意：一部のプログラム要素には、大文字と小文字の両方が使用されます。この場合は、記載されているとおりに入力してください。
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

--	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します。	--
/* */	スラッシュ ー アスタリスクとアスタリスク ー スラッシュは、複数行にまたがるコメントを区切ります。	/* */

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
```

PL/SQL の新機能

この項では、PL/SQL リリース 1（9.0.1）およびリリース 2（9.2）の新機能について説明し、追加情報の参照先も示します。

次の PL/SQL の新機能について説明します。

- Oracle9i の PL/SQL の新機能

Oracle9*i* の PL/SQL の新機能

リリース 2（9.2）

- **PL/SQL レコード全体の挿入、更新および選択**

  レコードの各属性を個別に指定せず、1 つの PL/SQL レコード変数を指定しレコードを SQL 表に挿入したり、更新できるようになりました。また、各 SQL 列に対して個別の PL/SQL 表を使用せずに、行全体を PL/SQL のレコード表に選択することもできます。

  関連項目：

  - 5-61 ページの「PL/SQL レコードのデータベースへの挿入」
  - 5-62 ページの「PL/SQL レコード値を使用したデータベースの更新」
  - 5-65 ページの「レコードのコレクションへのデータの問合せ」

- **結合配列**

  VARCHAR2 値で索引付けされたコレクションを作成して、Perl や他の言語のハッシュ表に類似した機能を提供できます。

  関連項目：

  - 5-4 ページの「結合配列（索引付き表）」

- **ユーザー定義のコンストラクタ**

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

  関連項目：

  - 10-28 ページの「コンストラクタの定義」

- **UTL_FILE パッケージの拡張**

  UTL_FILE には、PL/SQL で一般的なファイル管理操作を実行する様々な新しいファンクションが含まれています。

  関連項目：

  - 『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』

- **オブジェクト型のための TREAT ファンクション**

  オブジェクト・メソッドをコールする時に使用する型継承のレベルを動的に選択できます。つまり、親の型の様々なレベルから、継承するオブジェクト型を参照して、特定の親の型からメソッドをコールできます。このファンクションは、同じ名前の SQL ファンクションと似ています。

  関連項目：

  - 『Oracle9*i* SQL リファレンス』

- **オンライン・マニュアルでのリンク**

  このマニュアルから他のマニュアルへのほとんどの相互参照が、より具体的になりました。相互参照は、別のマニュアルの目次ではなく、特定の位置にリンクします。この変更は現在進行中のため、すべてのリンクがこの版で改良されているとはかぎりません。

リリース 1（9.0.1）

- **SQL パーサーと PL/SQL パーサーの統合**

  PL/SQL では、INSERT、UPDATE、DELETE などの SQL 文の構文が全面的にサポートされるようになりました。従来は PL/SQL プログラムで有効な SQL 構文に対してエラーが戻されていた場合がありますが、このリリースからは動作します。

  関連項目： エラー・チェックの一貫性向上により、実行時にエラーになるかわりにコンパイル時に無効なコードが検出されたり、その逆になる場合があります。移行手続きの一部としてソース・コードの変更を必要とする場合があります。移行手続きの詳細は、『Oracle9*i* データベース移行ガイド』を参照してください。

- **CASE 文および式**

  CASE 文および式は、IF/THEN の選択肢を複数の代替で表す簡単な方法です。

  関連項目：

  - 2-31 ページの「CASE 式」
  - 4-6 ページの「CASE 文」
  - 13-17 ページの「CASE 文」

- **継承および動的メソッド・ディスパッチ**

  型をスーパータイプ / サブタイプの階層内で宣言できます。サブタイプはスーパータイプから属性とメソッドを継承します。また、サブタイプで新規の属性とメソッドを追加し、既存のメソッドをオーバーライドできます。オブジェクト・メソッドのコールにより、オブジェクトの型に基づいて適切なバージョンのメソッドが実行されます。

  関連項目：

  - 10-14 ページの「PL/SQL の型の継承の概要」
  - 8-28 ページの「オーバーロードと継承の相互作用」

- **型の発展**

  オブジェクト型の属性とメソッドを追加または削除でき、その際に型および対応するデータを再作成する必要はありません。この機能により、型の階層を事前に全面的に計画するのではなく、アプリケーションの変化に合せて調整できます。

  関連項目： 10-12 ページの「既存のオブジェクト型の属性およびメソッドの変更（型の発展）」

- **新しい日付 / 時刻型**

  新しいデータ型 `TIMESTAMP` では、秒の小数部を含む時刻値が記録されます。新しいデータ型である `TIMESTAMP WITH TIME ZONE` および `TIMESTAMP WITH LOCAL TIME ZONE` を使用すると、タイム・ゾーンの違いを考慮して日付値と時刻値を調整できます。クロックが正方向または逆方向にシフトするときの時差を考慮して、タイム・ゾーンに夏時間が適用されるかどうかを指定できます。新しいデータ型である `INTERVAL DAY TO SECOND` および `INTERVAL YEAR TO MONTH` は、2 つの日付値および時刻値の時差を表し、日付演算を簡素化します。

  関連項目：

  - 3-14 ページの「日時および時間隔型」
  - 3-19 ページの「日時および時間隔の演算」
  - 2-9 ページの「日時リテラル」

- **PL/SQL コードのシステム固有のコンパイル**

  オラクル社が提供するストアド・プロシージャやユーザーが記述するストアド・プロシージャを、代表的な C の開発ツールを使用してシステム固有の実行可能ファイルにコンパイルして、パフォーマンスを改善します。この設定は保存されるため、プロシージャは後で無効になった場合も同じ方法でコンパイルされます。

  関連項目： 12-14 ページの「システム固有の実行のための PL/SQL コードのコンパイル」

- **改善されたグローバリゼーションおよび各国語サポート**

  固定幅または可変幅のキャラクタ・セットを使用し、データを Unicode 形式で格納できます。文字列の処理とストレージの宣言は、バイト長または文字長で指定できます。文字長で指定すると、バイト数が計算されます。データベース全体を、文字列に同じ長さセマンティクスを使用するように設定したり、個々のプロシージャの設定を指定できます。プロシージャが無効になった場合は、この設定は記憶されます。

  関連項目：

  - 3-5 ページの「キャラクタ・タイプ」
  - 3-10 ページの「各国語キャラクタ・タイプ」

- **表関数とカーソル式**

  戻される行のセットを表のように問合せできます。ある関数から別の関数に結果セットを渡すことができるため、表に中間結果を保持せずに一連の変換を設定できます。結果セットの行を一度に少しずつ戻し、1 つの関数で多数の結果セットが生成された場合のメモリー・オーバーヘッドを削減します。

  関連項目：

  - 8-30 ページの「表関数を使用した複数行の受入れと戻し」
  - 6-40 ページの「カーソル式の使用」

- **マルチレベル・コレクション**

  PL/SQL 表の VARRAY、VARRAY の VARRAY または PL/SQL 表の PL/SQL 表などを作成するために、コレクション型をネストできます。マルチディメンション配列など、複雑なデータ構造のモデルを通常の方法で作成できます。

  関連項目： 5-25 ページの「マルチレベル・コレクションの使用」

- **LOB データ型の適切な統合**

  LOB 型を類似する他の型と同様に操作できます。CLOB および NCLOB 型に文字関数を使用できます。BLOB 型を RAW として扱うことができます。LOB と他の型との変換は、特に LONG 型から LOB 型に変換する場合には、はるかに単純化されます。

  関連項目： 3-12 ページの「LOB 型」

- **バルク操作の強化**

  システム固有の動的 SQL（EXECUTE IMMEDIATE 文）を使用して、バルク・フェッチなどのバルク SQL 操作を実行できるようになりました。一部の行にエラーがあっても継続するバルク挿入操作やバルク更新操作を実行し、操作の完了後に問題を検査できます。

  関連項目：

  - 5-38 ページの「バルク・バインドを使用したコレクションのループ・オーバーヘッドの削減」
  - 11-11 ページの「バルク動的 SQL の使用」
  - 13-63 ページの「EXECUTE IMMEDIATE 文」

- **MERGE 文**

  この特化された文により、挿入と更新が組み合されて 1 つの操作になります。これは、特定パターンの挿入と更新を実行するデータ・ウェアハウス・アプリケーションを意図しています。

  関連項目：

  - 概要と例は、13-113 ページの「MERGE 文」を参照してください。
  - 詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

1

PL/SQL の概要

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。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。

この章の項目は、次のとおりです。

関連項目： PL/SQL に関する追加情報とサンプル・コードについては、次の URL で OTN-J（Oracle Technology Network Japan）にアクセスしてください。`http://otn.oracle.co.jp/tech/pl_sql/`

PL/SQL の主な特長

PL/SQL を理解するための第一歩としては、サンプル・プログラムを見ることをお薦めします。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 'examp1'
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 文を使用した Oracle データの操作や、フロー制御文を使用したデータ処理ができます。また、定数と変数の宣言、プロシージャとファンクションの定義、およびランタイム・エラーのトラップもできます。このように、PL/SQL では、SQL のデータ操作機能と手続き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

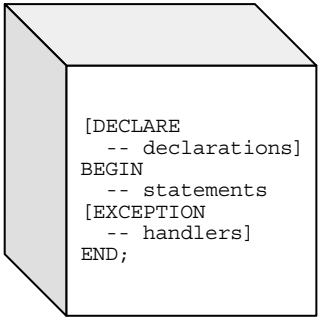
PL/SQL はブロック構造化言語です。つまり、PL/SQL プログラムを構成する基本単位（プロシージャ、ファンクション、無名ブロック）は、それぞれが任意の数のネストされたサブブロックを持つことができる論理ブロックです。一般に、個々の論理ブロックは、解決する必要のあるそれぞれの問題または副問題に対応しています。このように、PL/SQL は、問題を分割して克服する「段階的詳細化」のアプローチをサポートしています。

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、それを使用する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 のように、PL/SQL ブロックは 3 つの部分に分かれています。宣言部、実行部、例外処理部です。（PL/SQL では、警告またはエラー条件が「例外」と呼ばれます。）このうち必ず存在する必要があるのは実行部のみです。

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、式を使用可能な任意の場所で使用できます。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、事前に宣言する必要があります。

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;
```

コンポジット・データ型 TABLE、VARRAY、RECORD を使用して、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 3 つあります。1 つ目は、コロンに等号を付けた代入演算子（:=）を使用する方法です。変数は演算子の左に、ファンクション・コールを含む式は演算子の右に置きます。次に例を示します。

```
tax := price * tax_rate;
valid_id := FALSE;
bonus := current_salary * 0.10;
wages := gross_pay(emp_id, st_hrs, ot_hrs) - deductions;
```

変数に値を代入する 2 つ目の方法は、データベース値を選択またはフェッチして代入する方法です。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。変数 bonus を別の計算に使用したり、変数の値をデータベース表に挿入できます。

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

変数を値に代入する 3 つ目の方法は、値を OUT または IN OUT パラメータとしてサブプログラムに渡して代入する方法です。次の例に示すように、IN OUT パラメータを使用すると、コール先のサブプログラムに初期値を渡して、コール元に更新された値を戻すことができます。

```
DECLARE
   my_sal REAL(7,2);
   PROCEDURE adjust_salary (emp_id INT, salary IN OUT REAL) IS ...
BEGIN
   SELECT AVG(sal) INTO my_sal FROM emp;
   adjust_salary(7788, my_sal); -- assigns a new value to my_sal
```

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する必要がある点が異なります。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;
```

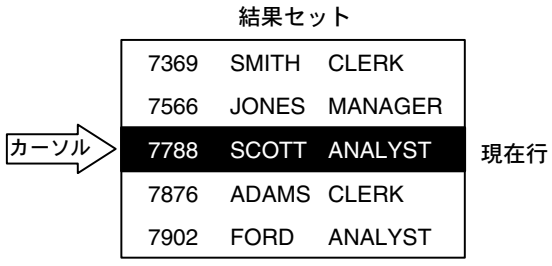
カーソル

Oracle は、作業領域を使用して 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 の構造体を使用すると、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 行のみを戻す問合せなど、すべての SQL DML 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。次に例を示します。

```
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　問合せの処理



複数行の問合せの処理は、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、問合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 文および CLOSE 文を使用します。OPEN 文は、カーソルに関連付けられている問合せを実行し、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禁止にします。

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 ループを使用して、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後に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、ドット表記法を使用します。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すことができる有効な PL/SQL 変数です。この変数を使用すると、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;
   ...
END;
```

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用すると、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、属性のインジケータ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用します。

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

%TYPE 属性を使用して 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更されます。

%ROWTYPE

PL/SQL ではデータのグループ化にレコードを使用します。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フェッチ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には、dept 表の列と同じ名前およびデータ型があります。

```
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用します。

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

従業員の姓、給与、入社日および役職を取り出すカーソルを宣言する場合、次のように %ROWTYPE を使用して、同じ情報を格納するレコードを宣言できます。

```
DECLARE
```

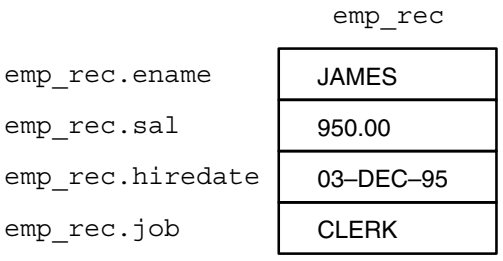
```
   CURSOR c1 IS
      SELECT ename, sal, hiredate, job FROM emp;
   emp_rec c1%ROWTYPE;  -- declare record variable that represents
                        -- a row fetched from the emp table
```

このとき、次の文を実行すると、

```
FETCH c1 INTO emp_rec;
```

emp 表の ename 列の値は emp_rec の ename フィールドに、sal 列の値は sal フィールドに、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



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 を使用すると、Oracle データを操作できるのみでなく、IF-THEN-ELSE、CASE、FOR-LOOP、WHILE-LOOP、EXIT-WHEN および GOTO などの条件制御文、反復制御文および順次制御文を使用してデータを処理できます。これらの文を組み合せれば、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ぶ必要のある場面はよくあります。IF-THEN-ELSE 文を使用すると、一連の文を条件に合せて実行できます。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できる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;
```

複数の値から、またはアクションの途中で選択するには、CASE 構造体を使用できます。CASE 式では条件が評価され、各ケースの値が戻されます。CASE 文では条件が評価され、ケースごとにアクションが実行されます。アクションは、PL/SQL ブロック全体の場合もあります。

```
-- This CASE statement performs different actions based
-- on a set of conditional tests.
CASE
  WHEN shape = 'square' THEN area := side * side;
  WHEN shape = 'circle' THEN
    BEGIN
      area := pi * (radius * radius);
      DBMS_OUTPUT.PUT_LINE('Value is not exact because pi is irrational.');
    END;
  WHEN shape = 'rectangle' THEN area := length * width;
  ELSE
    BEGIN
      DBMS_OUTPUT.PUT_LINE('No formula to calculate area of a' || shape);
      RAISE PROGRAM_ERROR;
    END;
END CASE;
```

問合せの結果を使用してアクションを選択する一連の文が、データベース・アプリケーションではよく使用されます。また、関連するエントリが別の表に見つかった場合にのみ、行の挿入や削除を実行する一連の文もよく使用されます。このようなよく使用される一連の文は、条件論理を使用して 1 つの PL/SQL ブロックにまとめることができます。

反復制御

LOOP 文を使用すると、一連の文を複数回実行できます。一連の文の最初の文の前にキーワード LOOP を置き、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、範囲中のそれぞれの整数に対して一連の文を 1 回実行できます。たとえば、次のループでは、500 個の数とその平方根がデータベース表に挿入されます。

```
FOR num IN 1..500 LOOP
   INSERT INTO roots VALUES (num, SQRT(num));
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 7499 よりも指揮系統内で上位にあり、給与が $2500 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 := 0;
   mgr_num        emp.mgr%TYPE;
   last_name      emp.ename%TYPE;
   starting_empno emp.empno%TYPE := 7499;
BEGIN
   SELECT mgr INTO mgr_num FROM emp
      WHERE empno = starting_empno;
   WHILE salary <= 2500 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;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      INSERT INTO temp VALUES (NULL, NULL, 'Not found');
      COMMIT;
END;
```

これ以上の処理を望まない場合、または不可能な場合は、EXIT-WHEN 文でループを終了できます。EXIT 文が見つかると、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 文を使用すると、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 ブロックの前に置く必要があります。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、正しく定義されたモジュールに分けることができます。つまり、次々と詳細化していくことで、複雑な問題を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ます。PL/SQL は、このニーズに応えるために、ブロック、サブプログラムおよびパッケージなどのプログラム・ユニットを提供しています。

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  -- executable part starts here
```

```
   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  -- exception-handling part starts here
   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。この番号を使用し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 の場合は例外が呼び出され、NULL でない場合は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、変数、カーソルおよび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ため、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、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、仕様を実際に実装します。

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、パッケージ仕様部の宣言のみです。パッケージ本体の実装の詳細は隠ぺいされ、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 度目以降は、パッケージ内の関連サブプログラムをコールするため、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 により索引付き表、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、集合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用します。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と同様の働きをします。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、それらを使用して、データの列をデータベースの表に出し入れしたり、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できます。

レコード

%ROWTYPE 属性を使用して、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用すると、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の名前を持ちますが、フィールドのデータ型は異なっていてもかまいません。名前、給与、入社日など、従業員に関する様々なデータがあるとします。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。レコードを使用すると、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長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成できます。

CREATE TYPE 文を使用し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成します。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に必要な属性と動作のみを指定します。

```
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成されることになります。インスタンス（オブジェクト）は、必要な数のみ

作成できます。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。

情報の隠ぺい

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実装します。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ると、実装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を見ることはできません。たとえば、`raise_salary` という名前のプロシージャの実装は隠されています。ユーザーが知っておく必要があるのは、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のみです。`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実装します。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すると、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用できます。

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施行します。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のみで、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、簡単に検出して処理できます。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成します。

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と、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 文で明示的に呼び出す必要があります。

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 the exception
```

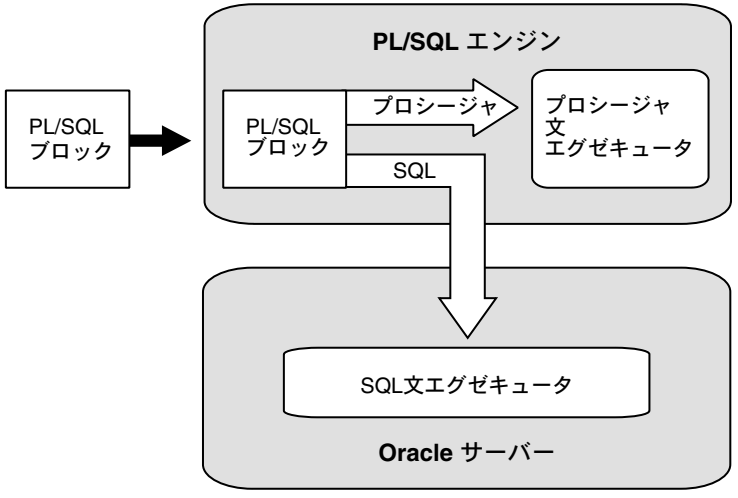
PL/SQL アーキテクチャ

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 サーバー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 つの環境で使用できます。

- Oracle データベース・サーバー
- Oracle のツール製品

この 2 つの環境は独立しています。PL/SQL は Oracle サーバーにまとめられていますが、いくつかの Oracle のツール製品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のみを実行し、SQL 文を Oracle サーバーの SQL 文エグゼキュータに送ります。

図 1-4　PL/SQL エンジン



Oracle データベース・サーバー

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 を利用する必要があります。PL/SQL エンジンがあれば、Oracle サーバーは単一の SQL 文のみでなく、PL/SQL のブロックやサブプログラムも処理できます。Oracle サーバー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 サーバーに送信します。Oracle サーバー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` を使用して明示的に作成さ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・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、または対話形式で 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 つのみロードされます。

データベース・トリガー

データベース・トリガーは、データベースの表、ビューまたはイベントに対応付けられているストアド・サブプログラムです。たとえば、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、次の表レベルのトリガーは emp 表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;
```

トリガーの実行部にはプロシージャ文のみでなく、SQL DML 文も入れることができます。表レベルのトリガー以外にも、ビュー用の INSTEAD OF トリガーとスキーマ用のシステム・イベントのトリガーがあります。詳細は、『Oracle9*i* 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。

Oracle のツール製品

PL/SQL エンジンを持つアプリケーション開発用 Oracle のツール製品は、PL/SQL ブロックとサブプログラムを処理できます。Oracle のツール製品はブロックをローカルの 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 文のみを 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 文を頻繁に使用します。PL/SQL を使用すると、Oracle サーバーへのコールを回避できます。さらに、PL/SQL ファンクションを使用してフィールド・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
- 優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
- 優れたセキュリティ

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用して、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、要求内容のみを指定します。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用できます。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の構築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用すると、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成できるのみでなく、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と、オブジェクト型を使用してデータ・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。また、オブジェクト型を使用すれば、実装の細部が隠されるため、クライアント・プログラムに影響する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用することで、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。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。

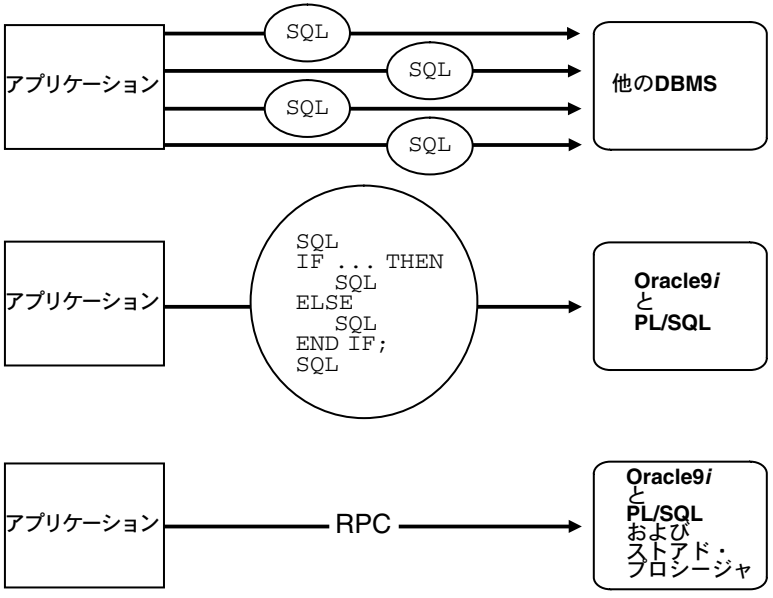
パフォーマンスの向上

PL/SQL がなければ、Oracle は SQL 文を 1 文ずつ処理する必要があります。1 つの SQL 文は Oracle を 1 回コールするため、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する必要があり、通信量が増大します。

ただし、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用して 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ため、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 度コールするのみで起動できます。これによりネットワークの通信量が軽減され、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 のツール製品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 を使用すると、ツールは Oracle サーバー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、ネットワークの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用し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 ブロック全体を使用できます。複数のトリガー・ステップ、マクロまたはユーザー・イグジットを使用する必要はありません。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ため、生産性はさらに向上します。たとえば、1 つのツールを使用して書いたスクリプトを他のツールでも使用可能です。

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。

SQL 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成するのに役立ちます。

%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 属性を使用すると、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するときに、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は有効です。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。

優れたセキュリティ

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化が可能です。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無効化し、ユーザーによる問合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、表そのものへのアクセスは付与しないようにします。

2

PL/SQL の基礎

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用して、実世界のオブジェクトや演算を表現できます。

この章の項目は、次のとおりです。

キャラクタ・セット

PL/SQL プログラムは、特定のキャラクタ・セットを使用したテキストとして作成されます。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ため、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（単純記号とコンパウンド記号）
識別子（予約語を含む）
リテラル
コメント

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;  -- not allowed
```

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コンパウンド記号（:=）が分かれているため、不正です。

```
count : = count + 1;  -- not allowed
```

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ることができます。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コンパウンド記号です。たとえば、デリミタを使用し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
%	属性のインジケータ
'	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
:	ホスト変数のインジケータ
,	項目のセパレータ
*	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・アクセスのインジケータ
;	文の終了記号
-	減算 / 否定演算子

コンパウンド記号は 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

記号	意味
<<	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
--	単一行コメントのインジケータ

識別子

識別子を使用して、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シャープ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用できません。

```
mine&yours    -- not allowed because of ampersand
debit-amount  -- not allowed because of hyphen
on/off        -- not allowed because of slash
user id       -- not allowed because of space
```

次の例は、ドル記号、アンダースコアおよびシャープ記号を隣接して使用したり、先頭以外の位置で使用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用でき、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のみであれば、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
```

識別子のサイズは 30 文字以内にする必要があります。ただし、ドル記号、アンダースコアおよびシャープ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してください。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用してください。

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、予約語を再定義するとコンパイル・エラーが発生します。

```
DECLARE
   end BOOLEAN;  -- not allowed; 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;  -- allowed
```

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 F を参照してください。

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、再宣言できます。ただし、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ため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サイズは、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用できます。ただし、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用できません。たとえば、次の文は無効です。

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

ビューを作成して不正な列を改名し、SQL 文の中でベース表のかわりにこのビューを使用できます。

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、文字、文字列またはブール値です。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用できます。整数と実数整数リテラルは、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.
```

PL/SQL では、12.0 および 25. などの数字は、整数値がある場合でも実数とみなします。

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、小数点が移動する桁数にも対応しています。上の例では、暗黙的な小数点が 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、コンパイル・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・セットの印刷可能文字をすべて使用できます。つまり、英字、数字、空白および特殊記号を使用できます。次に例を示します。

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

文字リテラルの中で、PL/SQL は大 / 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 '0' ～ '9' は、整数リテラルと同じではありませんが、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場合に、文字列中でアポストロフィを表現するにはどうすればよいかを考えます。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 / 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。

日時リテラル

日時リテラルには、データ型に応じて様々な形式があります。次に例を示します。

```
DECLARE
d1 DATE := DATE '1998-12-25';
t1 TIMESTAMP := TIMESTAMP '1997-10-22 13:01:01';
t2 TIMESTAMP WITH TIME ZONE := TIMESTAMP '1997-01-31 09:26:56.66 +02:00';
-- Three years and two months
-- (For greater precision, we would use the day-to-second interval)
i1 INTERVAL YEAR TO MONTH := INTERVAL '3-2' YEAR TO MONTH;
-- Five days, four hours, three minutes, two and 1/100 seconds
i2 INTERVAL DAY TO SECOND := INTERVAL '5 04:03:02.01' DAY TO SECOND;
...
```

特定の時間隔値が YEAR TO MONTH であるか DAY TO SECOND であるかも指定できます。たとえば、current_timestamp - current_timestamp では、デフォルトで INTERVAL DAY TO SECOND 型の値が生成されます。時間隔の型は、次の形式で指定できます。

- (*interval_expression*) DAY TO SECOND
- (*interval_expression*) YEAR TO MONTH

日付および時刻型の構文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。日付 / 時刻算術の実行例は、『Oracle9*i* 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。

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しないでください。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用します。PL/SQL では、単一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。

単一行コメント

単一行コメントは、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、行の末尾であれば文の途中でも使用でき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 行を使用禁止にする場合があります。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;
```

複数行コメント

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用して、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;
*/
```

コメントの制限

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、単一行コメントは使用できません。これは、行の終わりを示す文字が無視され、単一行コメントが行の終わりで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用してください。

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格納場所の名前を決めます。

次に例を示します。

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

1 つ目の宣言で、DATE 型の変数の名前を決めています。2 つ目の宣言で、SMALLINT 型の変数の名前を決め、代入演算子を使用して変数に初期値 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、宣言の中で初期化する必要があります。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

変数の初期化には、代入演算子のかわりにキーワード DEFAULT も使用できます。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 を使用して、サブプログラム・パラメータ、カーソル・パラメータおよび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

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を続ける必要があります。たとえば、次の宣言は誤りです。

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

PL/SQL では、サブタイプ NATURALN と POSITIVEN は、あらかじめ NOT NULL として定義されています。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を続ける必要があります。それ以外の場合は、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;  -- not allowed; not initialized
```

%TYPE の使用

%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 属性を使用して宣言された変数は、データ型指定子を使用して宣言された変数と同じように扱われます。たとえば前述の宣言では、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 属性を使用し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、列を参照できます。

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

%TYPE 属性を使用して 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

%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、カーソルまたは強い型指定のカーソル変数でフェッチ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 つ目のレコードには表 emp から選択された行が格納されます。2 つ目のレコードには、カーソル c1 でフェッチ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 レコードに入れます。

```
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 ...
```

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 を使用し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;
```

集計代入

%ROWTYPE 属性を使用した宣言は、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、レコード全体の宣言が同じ表またはカーソルを参照している場合には、そのレコード全体の間で集計代入できます。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;
```

ただし、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;  -- not allowed
```

2 番目の方法として、次の例に示すように、SELECT 文または FETCH 文を使用し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ある必要があります。

```
DECLARE
   dept_rec dept%ROWTYPE;
   ...
BEGIN
   SELECT * INTO dept_rec FROM dept WHERE deptno = 30;
   ...
END;
```

しかし、代入文を使用し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, ...);  -- not allowed
```

エイリアシングの使用

%ROWTYPE 属性と関連のあるカーソルからフェッチされた選択リスト項目は、単純名を持つ必要があります。また、選択リスト項目が式の場合は別名を持つ必要があります。次の例では、wages という別名を使用しています。

```
-- 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;
```

宣言の制限

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、事前に宣言する必要があります。たとえば、次に示す maxi の宣言は誤りです。

```
maxi INTEGER := 2 * mini;  -- not allowed
mini INTEGER := 15;
```

しかし、PL/SQL ではサブプログラムの前方宣言ができます。詳細は、8-10 ページの「PL/SQL のサブプログラムの宣言」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。ただし、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;  -- not allowed
```

各変数を次のように別々に宣言する必要があります。

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

PL/SQL の命名規則

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、修飾名、リモート名または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 は次のように使用できます。

```
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用しています。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、ドット表記法を使用して名前を修飾する必要があります。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・アクセスのインジケータ（@）を使用してデータベース・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

シノニムを作成し、表、順序、ビュー、スタンドアロン・サブプログラム、パッケージ、オブジェクト型などのリモート・スキーマ・オブジェクトに関する位置の透過性を提供できます。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、変数、カーソル、カーソル変数、例外および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ある必要があります。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。次の例では、2 番目の宣言は誤りです。

```
DECLARE
   valid_id BOOLEAN;
   valid_id VARCHAR2(5);  -- not allowed duplicate identifier
```

識別子に適用される有効範囲規則については、2-19 ページの「PL/SQL の識別子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、変数および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
```

名前解決

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' のみでなくすべての雇用者が emp 表から削除されます。

```
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・ラベルを使用し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用して修飾しています。

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

名前解決の詳細は、付録 D を参照してください。

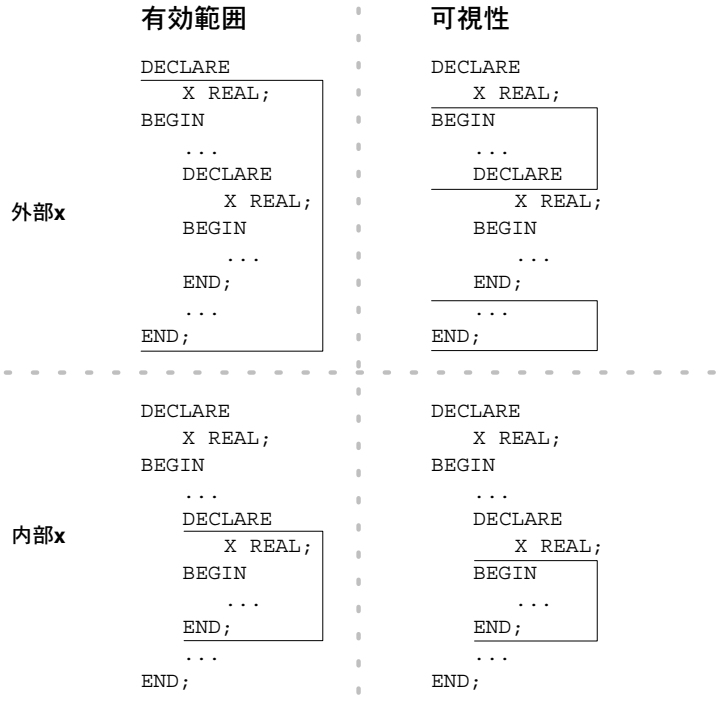
PL/SQL の識別子の有効範囲と可視性

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のみ、可視の状態になっています。図 2-1 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宣言されると、両方の識別子が有効範囲内にあることになります。ただし、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になるため、可視であるのはローカル識別子のみです。

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 2-1　有効範囲と可視性



次の例は、有効範囲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、修飾名を使用し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用できます。

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

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用できます。

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

ただし、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。

変数の代入

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。変数の値は、明示的に初期化しないかぎり未定義のままです。

```
DECLARE
   count INTEGER;
BEGIN
-- COUNT began with a value of NULL.
-- Thus the expression 'COUNT + 1' is also null.
-- So after this assignment, COUNT is still NULL.
   count := count + 1;
```

予期しない結果を避けるため、値を代入する前に変数を参照しないでください。

変数に値を代入する場合は、代入文を使用します。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あることが必要です。

ブール値の代入

ブール変数に代入できるのは、値 TRUE、FALSE および NULL のみです。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);
```

SQL 問合せ結果の PL/SQL 変数への代入

SELECT 文を使用しても変数に値を代入できます。選択リストの項目ごとに、対応する型互換の変数が INTO リストに存在している必要があります。次に例を示します。

```
DECLARE
   emp_id   emp.empno%TYPE;
   emp_name emp.ename%TYPE;
   wages    NUMBER(7,2);
BEGIN
   -- assign a value to emp_id here
   SELECT ename, sal + comm
      INTO emp_name, wages FROM emp
      WHERE empno = emp_id;
   ...
END;
```

列の値を選択してブール変数に代入できません。

PL/SQL の式および比較

式はオペランドと演算子を使用して作成します。オペランドとは、変数、定数、リテラルまたはファンクション・コールのことで、式の中の値はオペランドを使用して表現します。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（/）のようなバイナリ演算子は、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せて、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、式が使用さ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、優先順位に応じて特定の順序で実行されます。表 2-1 に、デフォルトでの演算の順序を上から順に示します。

表 2-1　演算の順序

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

カッコを使用すると、評価の順序を制御できます。たとえば、次の式ではカッコで演算子のデフォルトの優先順位が上書きされるため、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ため、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カッコを使用できます。

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

論理演算子

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

表 2-2　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	FALSE
TRUE	NULL	NULL	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	NULL	FALSE	NULL	TRUE
NULL	TRUE	NULL	TRUE	NULL
NULL	FALSE	FALSE	NULL	NULL
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 の場合にかぎり 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 は予想不可能な結果を生じる場合があります。2-33 ページの「比較文と条件文での NULL の扱い」を参照してください。

評価の順序

カッコを使用し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。ただし、2 番目の式では NOT が AND より優先されるため、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に、PL/SQL では短絡評価を使用します。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 になるため、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 DML 文の WHERE と、条件制御文の中で使用します。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

関係演算子を使用すると、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 の場合はブール値 TRUE を、NULL ではない場合は 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 かどうか**（NULL になっている状態かどうか）は、次のようにテストします。

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

LIKE 演算子

LIKE 演算子を使用すると、文字、文字列または CLOB 値をパターンと比較できます。大 / 小文字が区別されます。LIKE は、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 を使用し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' の場合、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、セット・メンバーシップを調べます。つまり、集合のいずれかのメンバーと等しいかどうかがテストされます。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、どの行も削除されません。

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

連結演算子

連結演算子（||）は、文字列（CHAR、VARCHAR2、CLOB またはそれと同等の Unicode で使用可能な型）を他の文字列に連結します。使用例を次に示します。

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

これは、次の行を戻します。

```
'suitcase'
```

両方のオペランドがデータ型 CHAR を持つ場合、連結演算子は CHAR 型の値を戻します。一方のオペランドが CLOB 値を持つ場合、連結演算子は一時的な CLOB を戻します。それ以外の場合は、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コンポジット式で構成されます。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用して、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用されます。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。

ブール算術式

関係演算子を使用して数値を比較し、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
```

ブール文字式

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。デフォルトでは、比較は文字列の各バイトのバイナリ値に基づいて行われます。

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
```

初期化パラメータ NLS_COMP=ANSI を設定すると、NLS_SORT 初期化パラメータで識別される照合順番を比較に使用できます。**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。内部的な順番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。この種の文字が照合順番に使用される場所については、言語ごとに規則が異なる場合があります。たとえば、アクセント記号が付いた文字のソート順序は、バイナリ値が同じであってもデータベース・キャラクタ・セットに応じて異なることがあります。

文字値を比較する場合には、ベース型 CHAR と VARCHAR2 の間にある意味上の違いを考慮する必要があります。詳細は、付録 B を参照してください。

多くの型は文字型に変換できます。たとえば、CLOB 変数を使用して比較、代入および他の文字操作ができます。可能な変換の詳細は、3-5 ページの「キャラクタ・タイプ」を参照してください。

ブール日付式

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
```

PL/SQL ブール式のガイドライン

- 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 にならない可能性があります。

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

- 比較する場合は、カッコを使用する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  -- not allowed
  ```

  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;
```

- CLOB 値を比較演算子または LIKE や BETWEEN などのファンクションとともに使用すると、一時的な LOB が作成されます。一時表領域がこのような一時的な LOB を処理できる大きさかどうかを確認する必要があります。詳細は、『Oracle9*i* アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。

CASE 式

CASE 式は、1 つ以上の選択肢から結果を選択して戻します。このために、CASE 式は**選択子**を使用します。この選択子は、戻す選択肢を判断するために値が使用される式です。CASE 式の書式は、次のとおりです。

```
CASE selector
   WHEN expression1 THEN result1
   WHEN expression2 THEN result2
   ...
   WHEN expressionN THEN resultN
  [ELSE resultN+1]
END;
```

選択子の後に 1 つ以上の WHEN 句があり、各句が順番にチェックされます。選択子の値によって、どの句が実行されるかが決定されます。選択子の値と最初に一致した WHEN 句によって結果値が決定され、後続の WHEN 句は評価されません。次に例を示します。

```
DECLARE
   grade CHAR(1) := 'B';
   appraisal VARCHAR2(20);
BEGIN
   appraisal :=
      CASE grade
         WHEN 'A' THEN 'Excellent'
         WHEN 'B' THEN 'Very Good'
         WHEN 'C' THEN 'Good'
         WHEN 'D' THEN 'Fair'
         WHEN 'F' THEN 'Poor'
         ELSE 'No such grade'
      END;
END;
```

オプションの ELSE 句の機能は、IF 文の ELSE 句に似ています。選択子の値が WHEN 句のオプションの 1 つでなければ、ELSE 句が実行されます。ELSE 句が指定されておらず、一致する WHEN 句がなければ、式は NULL を戻します。

CASE 式のかわりに、CASE 文を使用して WHEN 句に PL/SQL ブロック全体を使用できます。詳細は、4-6 ページの「CASE 文」を参照してください。

検索 CASE 式

PL/SQL には、次の書式の検索 CASE 式も用意されています。

```
CASE
   WHEN search_condition1 THEN result1
   WHEN search_condition2 THEN result2
   ...
   WHEN search_conditionN THEN resultN
  [ELSE resultN+1]
END;
```

検索 CASE 式には選択子はありません。各 WHEN 句にはブール値を生成する検索条件が含まれており、単一の WHEN 句で異なる変数または複数の条件をテストできます。次に例を示します。

```
DECLARE
   grade CHAR(1);
   appraisal VARCHAR2(20);
BEGIN
   ...
   appraisal :=
      CASE
         WHEN grade = 'A' THEN 'Excellent'
         WHEN grade = 'B' THEN 'Very Good'
         WHEN grade = 'C' THEN 'Good'
         WHEN grade = 'D' THEN 'Fair'
         WHEN grade = 'F' THEN 'Poor'
         ELSE 'No such grade'
      END;
   ...
END;
```

検索条件は順番に評価されます。各検索条件のブール値によって、どの WHEN 句が実行されるかが決定されます。検索条件が TRUE になると、その WHEN 句が実行されます。WHEN 句が 1 つでも実行された後は、後続の検索条件は評価されません。TRUE になる検索条件がなければ、オプションの ELSE 句が実行されます。WHEN 句が実行されず、ELSE 句が指定されていなければ、式の値は NULL となります。

比較文と条件文での NULL の扱い

NULL を使用する場合は、次の規則を念頭に置くことで、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 になります。
- 論理演算子 NOT を NULL 値に適用すると NULL が戻ります
- 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ません
- 単純な CASE 文中の式または CASE 式が NULL になる場合は、WHEN NULL を使用して一致させることができません。この場合は、検索 CASE 構文を使用して WHEN *expression* IS NULL をテストする必要があります。

次の例では、x と y が等しくないために一連の文（sequence_of_statements）が実行されることが予測されます。しかし、NULL は予測不能です。そのため、x と y が等しいかどうかは不明です。したがって、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。

```
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 が等しいために一連の文が実行されると予測されます。ただし、等号条件が成立するかどうかは不明であるため、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 が戻ることに注意してください。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 句内の一連の文が実行されます。x と y のどちらも NULL ではない場合、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。これには文字関数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_CHAR('');
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 演算子を使用してください。

```
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。使用例を次に示します。

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

これは、次の行を戻します。

```
'applesauce'
```

ファンクション

組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 の場合、NVL は SYSDATE の値を戻します。NULL ではない場合、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 の場合、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・レポート
数値
文字
データ型変換
日付
オブジェクト参照
その他

表 2-3 に、各カテゴリのファンクションを示します。エラー・レポート・ファンクションの説明は、第 13 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、『Oracle9*i* 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用できます。また、オブジェクト参照ファンクション DEREF、REF および VALUE とファンクション DECODE、DUMP および VSIZE 以外であれば、すべてのファンクションをプロシージャ文で使用できます。

SQL 集計関数（AVG や COUNT など）と SQL 分析関数（CORR や LAG など）は PL/SQL に組み込まれていませんが、SQL 文に使用できます（ただし、プロシージャ文には使用できません）。

表 2-3 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	CURRENT_DATE	REF	DECODE
	ASIN	CONCAT	HEXTORAW	CURRENT_TIMESTAMP	VALUE	DUMP
	ATAN	INITCAP	RAWTOHEX	DBTIMEZONE	TREAT	EMPTY_BLOB
	ATAN2	INSTR	ROWIDTOCHAR	EXTRACT		EMPTY_CLOB
	BITAND	INSTRB	TO_BLOB	FROM_TZ		GREATEST
	CEIL	LENGTH	TO_CHAR	LAST_DAY		LEAST
	COS	LENGTHB	TO_CLOB	LOCALTIMESTAMP		NLS_CHARSET_DECL_LEN
	COSH	LOWER	TO_DATE	MONTHS_BETWEEN		NLS_CHARSET_ID
	EXP	LPAD	TO_MULTI_BYTE	NEW_TIME		NLS_CHARSET_NAME
	FLOOR	LTRIM	TO_NCLOB	NEXT_DAY		NVL
	LN	NLS_INITCAP	TO_NUMBER	NUMTODSINTERVAL		SYS_CONTEXT
	LOG	NLS_LOWER	TO_SINGLE_BYTE	NUMTOYMINTERVAL		SYS_GUID
	MOD	NLSSORT		ROUND		UID
	POWER	NLS_UPPER		SESSIONTIMEZONE		USER
	ROUND	REPLACE		SYSDATE		USERENV
	SIGN	RPAD		SYSTIMESTAMP		VSIZE
	SIN	RTRIM		TO_DSINTERVAL		
	SINH	SOUNDEX		TO_TIMESTAMP		
	SQRT	SUBSTR		TO_TIMESTAMP_LTZ		
	TAN	SUBSTRB		TO_TIMESTAMP_TZ		
	TANH	TRANSLATE		TO_YMINTERVAL		
	TRUNC	TRIM		TZ_OFFSET		
		UPPER		TRUNC		

3

PL/SQL のデータ型

すべての定数、変数およびパラメータは、記憶形式、制約および値の有効範囲を指定するデータ型（または型）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。たとえば、整数、浮動小数点、文字、ブール、日付、コレクション、参照および LOB の各型から選択できます。また、PL/SQL では独自のサブタイプを定義できます。この章では、PL/SQL プログラムで頻繁に使用する基本的な型について説明します。より特化された型については、以降の章を参照してください。

この章の項目は、次のとおりです。

事前定義されたデータ型

スカラー型には内部コンポーネントがありません。コンポジット型には、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 3-1 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 3-1　組込みデータ型

数値型

数値型は、数値データ（整数、実数および浮動小数点数）の格納、量の表現および計算に使用します。

BINARY_INTEGER

BINARY_INTEGER データ型は、符号付き整数を格納するために使用します。値の大きさの範囲は、-2**31 ～ 2**31 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 ベース型は、サブタイプの導出元となるデータ型です。サブタイプはベース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 および POSITIVE は、整変数をそれぞれ負ではない整数値、または正の整数値に制限する場合に使用します。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用します。これは 3 値論理のプログラミングに役立ちます。

NUMBER

NUMBER データ型を使用すると、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、1E-130 ～ 10E125 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。次のフォームを使用します。

```
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ため、精度（*precision*）や位取り（*scale*）を指定できません。次のフォームを使用します。

```
NUMBER
```

整数を宣言する場合は（小数点がありません）次のフォームを使用します。

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

精度（*precision*）と位取り（*scale*）の指定には、定数や変数を使用できません。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 か、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用します。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用します。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用します。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用します。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用します。値の大きさの範囲は、-2**31 ～ 2**31 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用するため、ライブラリ算術計算を使用する NUMBER 演算や BINARY_INTEGER 演算よりも処理速度が速くなります。効率のために、PLS_INTEGER の大きさの範囲内でのすべての計算に PLS_INTEGER を使用してください。

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ため、古いアプリケーションでは互換性を保てるように従来どおり BINARY_INTEGER を使用してください。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 を使用してください。

キャラクタ・タイプ

キャラクタ・タイプは、英数字データの格納、ワードとテキストの表現および文字列の操作に使用します。

CHAR

CHAR データ型は固定長の文字データを格納するのに使用します。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・パラメータを使用して、その最大サイズを 32767 バイトまで指定できます。サイズはバイト数または文字数で指定できます。各文字には、キャラクタ・セットのエンコーディングに応じて 1 バイト以上が含まれます。次に構文を示します。

```
CHAR[(maximum_size [CHAR | BYTE] )]
```

最大サイズには 1 ～ 32767 の範囲の整数リテラルを指定します。シンボリック定数や変数は指定できません。

最大サイズを指定しなければ、デフォルトで 1 に設定されます。最大サイズを文字数ではなくバイト数で指定すると、CHAR(n) 変数が小さすぎて n 個のマルチバイト文字を保持できなくなる場合があります。この可能性を回避するには、データベース・キャラクタ・セットの一部の文字に複数のバイトが含まれていても、変数で n 個の文字を保持できるように、表記法 CHAR(n CHAR) を使用します。文字数の長さを指定する場合の上限は、32767 バイトです。ダブルバイトおよびマルチバイトのキャラクタ・セットの場合、指定できるのは、シングルバイト・キャラクタ・セットの 1/2 または 1/3 の文字のみです。

PL/SQL の文字変数は比較的長くなりがちですが、CHAR 型のデータベース列の最大幅は 2000 バイトです。このため、2001 バイト以上の CHAR 値は 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 2**31 バイト、つまり 2GB であるためです。ただし、LONG 列から 32768 バイト以上の値を取り出して CHAR(n) に入れることはできません。

CHAR または BYTE 修飾子を使用しなければ、デフォルトは NLS_LENGTH_SEMANTICS 初期化パラメータの設定によって決定されます。PL/SQL プロシージャのコンパイル時には、このパラメータの設定が記録されるため、プロシージャが無効になった後に再コンパイルするときにも同じ設定が使用されます。

注意 : 付録 B では、CHAR ベース型と VARCHAR2 ベース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、そのベース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 型のみ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用します。LONG データ型は、LONG 値の最大サイズが 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用します。LONG RAW データは、LONG RAW 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 データと同じです。LONG RAW 値の最大サイズは 32760 バイトです。

Oracle9*i* 以上では、LOB 変数を LONG および LONG RAW 変数のかわりに使用できます。LONG データを CLOB 型に、LONG RAW データを BLOB 型に移行することをお薦めします。詳細は、3-12 ページの「LOB 型」を参照してください。

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 2**31 バイトであるためです。ただし、LONG 列から 32761 バイト以上の値を取り出して 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 2**31 バイトであるためです。ただし、LONG RAW 列から 32761 バイト以上の値を取り出して 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 関数コール、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、『Oracle9*i* SQL リファレンス』を参照してください。

注意 : SQL 文では、LONG 値が LONG ではなく VARCHAR2 としてバインドされます。ただし、バインドされた VARCHAR2 の長さが VARCHAR2 列の最大幅（4000 バイト）を超える場合は、バインド型が LONG に自動的に変換され、エラー・メッセージが発行されます。これは、SQL ファンクションには LONG 値を渡すことができないためです。

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用します。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異なります。また、RAW データをシステム間で送信する際に、Oracle Net 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用して、その最大サイズを 32767 バイトまで指定できます。次に構文を示します。

```
RAW(maximum_size)
```

最大サイズには 1 ～ 32767 の範囲の整数リテラルを指定します。シンボリック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、長さ 2001 バイト以上の RAW 値は 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 2**31 バイトであるためです。ただし、LONG RAW 列から 32768 バイト以上の値を取り出して 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、**ROWID** というバイナリ値を格納する ROWID 疑似列があります。各 ROWID は、行の記憶域アドレスを表します。**物理 ROWID** は、通常の表の行を識別します。**論理 ROWID** は、索引構成表の行を識別します。ROWID データ型には物理 ROWID のみを格納できます。ただし、UROWID（ユニバーサル ROWID）データ型には物理 ROWID、論理 ROWID、外部（非 Oracle）ROWID 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のみ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 データ型を使用します。

ROWID を選択またはフェッチして ROWID 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込みファンクション ROWIDTOCHAR を使用します。逆に、ファンクション CHARTOROWID は ROWID 文字列を ROWID に変換します。文字列が有効な ROWID 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、暗黙的な変換にも適用されます。

UROWID 変数と文字列間で変換するには、ファンクション・コールなしで通常の代入文を使用します。値は、UROWID とキャラクタ・タイプの間で暗黙的に変換されます。

物理 ROWID 物理 ROWID を使用すると、特定の行にすばやくアクセスできます。物理 ROWID は、その行が存在するかぎり変わりません。ROWID 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、カーソルからフェッチされた最新の行を識別できます。6-50 ページの「コミットにまたがるフェッチ」を参照してください。

物理 ROWID 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 エンコーディングを使用します。たとえば、ROWID 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 OOOOOOFFFBBBBBBRRR 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・セグメントを識別します。表のクラスタなど、同じセグメント中のスキーマ・オブジェクトのデータ・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・ファイルを識別します。ファイル番号はデータベース内で一意です。
- BBBBBB: ブロック番号（上の例では AAADFN）。行が入っているデータ・ブロックを識別します。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します。

論理 ROWID 論理 ROWID を使用すると、特定の行に最も早くアクセスできます。Oracle は論理 ROWID を使用して索引構成表の 2 次索引を組み立てます。論理 ROWID は恒久物理アドレスを持たないため、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 ROWID は有効なまま残ります。

論理 ROWID には不確定要素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、不確定要素を使用し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、不確定要素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。新しい不確定要素を得るには、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 ROWID（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 ROWID を挿入できます。

ANALYZE 文は、推測の古さを追跡するのに役立ちます。これは、推測を持つ ROWID を UROWID 列に格納し、その ROWID を使用し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。

注意： ROWID を操作するには、提供されているパッケージ DBMS_ROWID を使用します。詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

VARCHAR2

VARCHAR2 データ型は可変長の文字データを格納するのに使用します。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用して、その最大サイズを 32767 バイトまで指定できます。次に構文を示します。

```
VARCHAR2(maximum_size [CHAR | BYTE])
```

最大サイズには 1 ～ 32767 の範囲の整数リテラルを指定します。シンボリック定数や変数は指定できません。

VARCHAR2 変数が小さい場合はパフォーマンスに合せて最適化され、大きい場合は効率的なメモリー使用に合せて最適化されます。分離点は 2000 バイトです。長さ 2000 バイト以上の VARCHAR2 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。長さ 2000 バイト未満の VARCHAR2 変数の場合は、変数の宣言済みの長さ全体が事前に割り当てられます。たとえば VARCHAR2(2000 BYTE) 変数と VARCHAR2(1999 BYTE) 変数に同じ 500 バイトの値を割り当てると、前者の変数は 500 バイト、後者は 1999 バイトのメモリーを使用することになります。

最大サイズを文字数ではなくバイト数で指定すると、VARCHAR2(n) 変数が小さすぎて n 個のマルチバイト文字を保持できないことがあります。この可能性を回避するには、データベース・キャラクタ・セットの一部の文字に複数のバイトが含まれていても、変数で n 個の文字を保持できるように、表記法 VARCHAR2(n CHAR) を使用します。文字数の長さを指定する場合の上限は、32767 バイトです。ダブルバイトおよびマルチバイトのキャラクタ・セットの場合、指定できるのは、シングルバイト・キャラクタ・セットの 1/2 または 1/3 の文字のみです。

PL/SQL の文字変数は比較的長くなりがちですが、VARCHAR2 型のデータベース列の最大幅は 4000 バイトです。このため、長さ 4001 バイト以上の VARCHAR2 値は 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 2**31 バイトであるためです。ただし、LONG 列から 32768 バイト以上の値を取り出して VARCHAR2(n) に入れることはできません。

CHAR または BYTE 修飾子を使用しなければ、デフォルトは NLS_LENGTH_SEMANTICS 初期化パラメータの設定によって決定されます。PL/SQL プロシージャのコンパイル時には、このパラメータの設定が記録されるため、プロシージャが無効になった後に再コンパイルするときにも同じ設定が使用されます。

VARCHAR2 サブタイプ 次の VARCHAR2 サブタイプの値は、そのベース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 型と IBM 型に対する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。ただし、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異なる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 を使用することをお薦めします。

各国語キャラクタ・タイプ

よく使用される 1 バイトの 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 2 バイトまたは 3 バイトが必要です。Oracle にはグローバリゼーション・サポートが用意されており、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。

グローバリゼーション・サポートにより、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せられます。したがって、世界中のユーザーは Oracle を使用して母国語で対話できます。

PL/SQL は、識別子およびソース・コードに使用されるデータベース・キャラクタ・セットと、各国語データに使用される各国語キャラクタ・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、各国語キャラクタ・セットから構成される文字列を格納できます。

注意 : CHAR データまたは VARCHAR2 データをキャラクタ・セットが異なるデータベース間で変換する場合は、データが適切な形式の文字列で構成されていることを確認してください。詳細は、『Oracle9*i* グローバリゼーション・サポート・ガイド』を参照してください。

UTF8 および AL16UTF16 エンコーディング

各国語キャラクタ・セットでは、データは UTF8 または AL16UTF16 エンコーディングを使用して Unicode として表されます。

AL16UTF16 エンコーディングの場合、各文字は 2 バイトです。このため、異なるプログラミング言語が混在する場合に、文字列長を計算して切捨てエラーを回避するのは簡単ですが、ほとんどが ASCII 文字で構成される文字列を格納するために余分なストレージ・オーバーヘッドを必要とします。

UTF8 エンコーディングの場合、各文字は 1、2 または 3 バイトです。このため、ほとんどの文字を 1 バイトで表せる場合にかぎり、変数や表の列に、より多数の文字を入れることがで

きます。データをバイト単位のバッファに転送すると、切捨てエラーになる可能性があります。

実行時に最大限の信頼性が得られるように、できるかぎりデフォルトの AL16UTF16 エンコーディングを使用することをお薦めします。Unicode 文字列の保持に必要なバイト数を判断する場合は、LENGTH ではなく LENGTHB ファンクションを使用してください。

NCHAR

NCHAR データ型は、固定長の（必要に応じて空白埋めされる）各国語キャラクタ・データを格納するために使用します。データの内部表現形式は、データベースの作成時に指定した各国語キャラクタ・セット、つまり、可変幅のエンコーディング（UTF8）を使用するか、それとも固定幅のエンコーディング（AL16UTF16）を使用するかによって異なります。この型は常にマルチバイト文字に対処できるため、Unicode データの保持に使用できます。

NCHAR データ型はオプション・パラメータを使用して、その最大サイズを文字数で指定できます。次に構文を示します。

```
NCHAR[(maximum_size)]
```

物理的な上限は 32767 バイトのため、長さに指定できる最大値は、AL16UTF16 エンコーディングの場合は 32767/2、UTF8 エンコーティングの場合は 32767/3 です。

最大サイズには整数リテラルを指定します。シンボリック定数や変数は指定できません。

最大サイズを指定しなければ、デフォルトの 1 が使用されます。CHAR では文字数またはバイト数で指定できますが、この値は常に文字数を表します。

```
my_string NCHAR(100);  -- maximum size 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、長さ 2001 バイト以上の NCHAR 値は NCHAR 型の列に挿入できません。

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 および NCHAR 値は、文と式の中で交換できます。常に安全に CHAR 値が NCHAR 値になりますが、CHAR 値のキャラクタ・セットで NCHAR 値のすべての文字を表すことができない場合は、NCHAR 値を CHAR 値にするとデータが失われる場合があります。このようなデータの消失が発生すると、各文字では通常は疑問符（?）のようになります。

NVARCHAR2

NVARCHAR2 データ型は、可変長の Unicode 文字データの格納に使用します。データの内部表現形式は、データベースの作成時に指定した各国語キャラクタ・セット、つまり、可変幅のエンコーディング（UTF8）を使用するか、それとも固定幅のエンコーディング（AL16UTF16）を使用するかによって異なります。この型は常にマルチバイト文字に対処できるため、Unicode データの保持に使用できます。

NVARCHAR2 データ型は必須パラメータを使用して、その最大サイズを文字数で指定できます。次に構文を示します。

```
NVARCHAR2(maximum_size)
```

物理的な上限は 32767 バイトのため、長さに指定できる最大値は、AL16UTF16 エンコーディングの場合は 32767/2、UTF8 エンコーティングの場合は 32767/3 です。

最大サイズには整数リテラルを指定します。シンボリック定数や変数は指定できません。

最大サイズを文字数またはバイト数で指定できる VARCHAR2 とは異なり、最大サイズは常に文字数を表します。

```
my_string NVARCHAR2(200);  -- maximum size is 200 characters
```

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。

VARCHAR2 および NVARCHAR2 値は、文と式の中で交換できます。VARCHAR2 値を NVARCHAR2 値にするのは常に問題ありませんが、NVARCHAR2 値のすべての文字が VARCHAR2 値のキャラクタ・セットで表せるとは限らない場合は、NVARCHAR2 値を VARCHAR2 値にするとデータが失われる場合があります。このようなデータの消失が発生すると、各文字では通常は疑問符（?）のようになります。

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用します。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 型とは異なります。たとえば、LOB（NCLOB を除く）はオブジェクト型の属性として使用できますが、LONG は使用できません。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 ロケータが格納されます。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 列の値を選択した場合、ロケータのみが戻されます。トランザクション中に取得した場合、LOB ロケータにはトランザクション ID が含まれるため、別のトランザクションの LOB の更新にはその LOB ロケータを使用できません。同様にあるセッション中に LOB ロケータを保管してから、それを別のセッションで使用することはできません。

Oracle9*i* 以上では、CLOB と CHAR および VARCHAR2 型との間の変換や、BLOB と RAW との間の変換も可能です。このため、LOB 型はほとんどの SQL および PL/SQL の文とファンクションに使用できます。LOB の読込み、書込みおよびピース単位の操作を行うには、オラクル社が提供するパッケージ DBMS_LOB を使用できます。詳細は、『Oracle9*i* アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。

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用します。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 は読取り専用です。変更はできません。BFILE のサイズはシステムに依存していますが、4GB（2**32 - 1 バイト）を超えるものは使用できません。指定された BFILE が存在し、Oracle にその読込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、リカバリ可能ではなく、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・オブジェクトを指すロケータが格納されます。BLOB のサイズは 4GB 以下にしてください。

BLOB はトランザクションに完全に関与し、リカバリ可能で、レプリケートできます。パッケージ DBMS_LOB によって行われた変更は、コミットまたはロールバックできます。BLOB ロケータは複数の（読取り専用）トランザクションにまたがることはできますが、複数のセッションにはまたがることができません。

CLOB

CLOB データ型は、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用します。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すロケータが格納されます。CLOB のサイズは 4GB 以下にしてください。

CLOB はトランザクションに完全に関与し、リカバリ可能で、レプリケートできます。パッケージ DBMS_LOB によって行われた変更は、コミットまたはロールバックできます。CLOB ロケータは複数の（読取り専用）トランザクションにまたがることはできますが、複数のセッションにはまたがることができません。

NCLOB

NCLOB データ型は、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用します。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すロケータが格納されます。NCLOB のサイズは 4GB 以下にしてください。

NCLOB はトランザクションに完全に関与し、リカバリ可能で、レプリケートできます。パッケージ DBMS_LOB によって行われた変更は、コミットまたはロールバックできます。NCLOB ロケータは複数の（読取り専用）トランザクションにまたがることはできますが、複数のセッションにはまたがることができません。

ブール型

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）を格納するのに使用します。BOOLEAN 変数で可能な操作は論理操作のみです。

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、FALSE および NULL のみです。データベース列には値 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。

日時および時間隔型

この項で説明するデータ型を使用すると、日付、時刻および時間隔（期間）を格納し、操作できます。日付 / 時刻型の変数には日時と呼ばれる値が保持され、時間隔型の変数には時間隔と呼ばれる値が保持されます。日時または時間隔は、その値を決定するフィールドで構成されます。次のリストに、各フィールドの有効値を示します。

フィールド名	有効な日時値	有効な時間隔値
YEAR	-4712 ～ 9999（年 0 を除く）	0 以外の任意の整数
MONTH	01 ～ 12	0 ～ 11
DAY	01 ～ 31（ロケールのカレンダの規則に従って MONTH および YEAR の値による制限付き）	0 以外の任意の整数
HOUR	00 ～ 23	0 ～ 23
MINUTE	00 ～ 59	0 ～ 59
SECOND	00 ～ 59.9(n)、9(n) は時刻の秒の精度	0 ～ 59.9(n)、9(n) は時間隔の秒の精度

フィールド名	有効な日時値	有効な時間隔値
TIMEZONE_HOUR	-12 ～ 14（範囲は夏時間の変更に対応）	該当なし
TIMEZONE_MINUTE	00 ～ 59	該当なし
TIMEZONE_REGION	ビュー V$TIMEZONE_NAMES に表示	該当なし
TIMEZONE_ABBR	ビュー V$TIMEZONE_NAMES に表示	該当なし

TIMESTAMP WITH LOCAL TIMEZONE を除き、前述の型はすべて SQL92 規格の一部です。日時および時間隔のフォーマット・モデル、リテラル、タイムゾーン名および SQL ファンクションの詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

DATE

DATE データ型は、午前 0 時からの経過秒数を含む固定長の日時の格納に使用します。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。

ヒント: 日付の等価性を各日付の時刻の部分に関係なく比較するには、比較、GROUP BY 操作などにファンクションの結果 TRUNC(*date_variable*) を使用します。

使用できる日付は、紀元前 4712 年 1 月 1 日から西暦 9999 年 12 月 31 日までです。ユリウス日付は、紀元前 4712 年 1 月 1 日からの日数です。ユリウス日付によって、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 で日付書式モデル 'J' を使用すると、DATE 値とそれに対応するユリウス日付の値の間で変換できます。

日付式の中では、デフォルトの日付書式の文字値は自動的に DATE 値に変換されます。デフォルトの日付書式は、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。

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、PL/SQL は整数リテラルを日数として解釈します。たとえば、SYSDATE + 1 は、次の日ということです。

TIMESTAMP

TIMESTAMP データ型は DATE データ型への拡張であり、年、月、日、時間、分および秒が格納されます。次に構文を示します。

```
TIMESTAMP[(precision)]
```

オプションの precision パラメータでは、秒のフィールドの小数部の桁数を指定します。精度には 0 ～ 9 の範囲の整数リテラルを指定します。シンボリック定数や変数は指定できません。デフォルトは 6 です。

デフォルトのタイムスタンプ日付書式は、Oracle 初期化パラメータ NLS_TIMESTAMP_FORMAT によって設定されます。

次の例では、TIMESTAMP 型の変数を宣言してからリテラル値を代入しています。

```
DECLARE
   checkout TIMESTAMP(3);
BEGIN
   checkout := '1999-06-22 07:48:53.275';
   ...
END;
```

この例では、秒フィールドの小数部は 0.275 です。

TIMESTAMP WITH TIME ZONE

TIMESTAMP WITH TIME ZONE データ型は、TIMESTAMP データ型を拡張したものであり、**タイムゾーンによる時差**を含みます。タイムゾーンによる時差とは、現地時間と協定世界時（UTC、旧称はグリニッジ標準時）の時差（時間および分単位）です。次に構文を示します。

```
TIMESTAMP[(precision)] WITH TIME ZONE
```

オプションの precision パラメータでは、秒のフィールドの小数部の桁数を指定します。精度には 0 ～ 9 の範囲の整数リテラルを指定します。シンボリック定数や変数は指定できません。デフォルトは 6 です。

デフォルトのタイムゾーン付きタイムスタンプの書式は、Oracle 初期化パラメータ NLS_TIMESTAMP_TZ_FORMAT によって設定されます。

次の例では、TIMESTAMP WITH TIME ZONE 型の変数を宣言してからリテラル値を代入しています。

```
DECLARE
   logoff TIMESTAMP(3) WITH TIME ZONE;
BEGIN
   logoff := '1999-10-31 09:42:37.114 +02:00';
   ...
END;
```

この例では、タイムゾーンによる時差は +02:00 です。

また、シンボリック名でタイムゾーンを指定することもできます。この指定には、'US/Pacific' などの長い形式、'PDT' などの省略形またはその組合せを使用できます。たとえば、次のリテラルはいずれも同じ時刻を表します。3 番目は、夏時間に切り替わった時点で適用される規則を指定しているため、最も信頼性の高い書式です。

```
TIMESTAMP '1999-04-15 8:00:00 -8:00'
TIMESTAMP '1999-04-15 8:00:00 US/Pacific'
TIMESTAMP '1999-10-31 01:30:00 US/Pacific PDT'
```

タイムゾーンに使用可能な名前は、V$TIMEZONE_NAMES データ・ディクショナリ・ビューの TIMEZONE_REGION および TIMEZONE_ABBR 列にあります。

2 つの TIMESTAMP WITH TIME ZONE 値は、UTC で同じ時間を表していれば、タイムゾーンによる時差に関係なく同一とみなされます。たとえば、UTC の太平洋標準時で午前 8:00 は東部標準時の午前 11:00 と同じであるため、次の 2 つの値は同一とみなされます。

```
'1999-08-29 08:00:00 -8:00'
'1999-08-29 11:00:00 -5:00'
```

TIMESTAMP WITH LOCAL TIME ZONE

TIMESTAMP WITH LOCAL TIME ZONE データ型は、TIMESTAMP データ型への拡張であり、**タイムゾーンによる時差**を含みます。タイムゾーンによる時差とは、現地時間と協定世界時（UTC、旧称はグリニッジ標準時）の時差（時間および分単位）です。TIMESTAMP WITH TIME ZONE と同様に、名前付きのタイム・ゾーンも使用できます。

次に構文を示します。

```
TIMESTAMP[(precision)] WITH LOCAL TIME ZONE
```

オプションの precision パラメータでは、秒のフィールドの小数部の桁数を指定します。精度には 0 ～ 9 の範囲の整数リテラルを指定します。シンボリック定数や変数は指定できません。デフォルトは 6 です。

このデータ型は、TIMESTAMP WITH TIME ZONE とは異なり、データベース列に値を挿入すると、値はデータベースのタイム・ゾーンに正規化され、タイムゾーンによる時差は列に格納されません。値を取り出すときには、ローカル・セッションのタイム・ゾーンで戻されます。

次の例では、TIMESTAMP WITH LOCAL TIME ZONE 型の変数を宣言しています。

```
DECLARE
   logoff TIMESTAMP(3) WITH LOCAL TIME ZONE;
BEGIN
   ...
END;
```

この型の変数には、リテラル値を代入できません。

INTERVAL YEAR TO MONTH

INTERVAL YEAR TO MONTH データ型は、年および月の時間隔を格納し、操作するために使用します。次に構文を示します。

```
INTERVAL YEAR[(precision)] TO MONTH
```

precision では、年フィールドの桁数を指定します。精度には 0 ～ 4 の範囲の整数リテラルを指定します。シンボリック定数や変数は指定できません。デフォルトは 2 です。

次の例では、INTERVAL YEAR TO MONTH 型の変数を宣言してから、101 年および 3 月の値を代入しています。

```
DECLARE
   lifetime INTERVAL YEAR(3) TO MONTH;
BEGIN
   lifetime := INTERVAL '101-3' YEAR TO MONTH; -- interval literal
   lifetime := '101-3'; -- implicit conversion from character type
   lifetime := INTERVAL '101' YEAR; -- Can specify just the years
   lifetime := INTERVAL '3' MONTH; -- Can specify just the months
   ...
END;
```

INTERVAL DAY TO SECOND

INTERVAL DAY TO SECOND データ型は、日、時間、分および秒の時間隔を格納し、操作するために使用します。次に構文を示します。

```
INTERVAL DAY[(leading_precision)] TO SECOND[(fractional_seconds_precision)]
```

leading_precision および *fractional_seconds_precision* では、それぞれ日付フィールドと秒フィールドの桁数を指定します。どちらの場合も、精度には 0 ～ 9 の範囲の整数リテラルを指定します。シンボリック定数や変数は指定できません。デフォルトはそれぞれ 2 および 6 です。

次の例では、INTERVAL DAY TO SECOND 型の変数を宣言しています。

```
DECLARE
   lag_time INTERVAL DAY(3) TO SECOND(3);
BEGIN
   IF lag_time > INTERVAL '6' DAY THEN ...
   ...
END;
```

日時および時間隔の演算

PL/SQL では、日時式と時間隔式を作成できます。この種の式に使用できる演算子を次のリストに示します。

オペランド 1	演算子	オペランド 2	結果タイプ
日時	+	時間隔	日時
日時	-	時間隔	日時
時間隔	+	日時	日時
日時	-	日時	時間隔
時間隔	+	時間隔	時間隔
時間隔	-	時間隔	時間隔
時間隔	*	数値	時間隔
数値	*	時間隔	時間隔
時間隔	/	数値	時間隔

また、EXTRACT などの各種ファンクションを使用して日時値を操作することもできます。この種のファンクションのリストは、2-36 ページの図 2-3「組込みファンクション」を参照してください。

日時の演算の詳細は、『Oracle9*i* SQL リファレンス』および『Oracle9*i* 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。

日付および時刻のサブタイプを使用する場合の切捨て問題の回避

日時の型のデフォルト精度は、最大精度より低い場合があります。たとえば、DAY TO SECOND のデフォルトは DAY(2) TO SECOND(6) ですが、最大精度は DAY(9) TO SECOND(9) です。この種の型の変数を代入し、プロシージャのパラメータを渡す場合に切り捨てを防ぐためには、次のサブタイプの変数とプロシージャのパラメータを宣言できます。各サブタイプでは、精度に最大値が使用されます。

```
TIMESTAMP_UNCONSTRAINED
TIMESTAMP_TZ_UNCONSTRAINED
TIMESTAMP_LTZ_UNCONSTRAINED
YMINTERVAL_UNCONSTRAINED
DSINTERVAL_UNCONSTRAINED
```

ユーザー定義のサブタイプ

PL/SQL のベース型はそれぞれ、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、そのベース型と同じ演算のセットを指定しますが、指定する値はベース型のサブセットのみです。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。単にそのベース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。

サブタイプを使用すると、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事前に定義されています。たとえば、PL/SQL では、サブタイプ CHARACTER および INTEGER が次のようにあらかじめ定義されています。

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0);  -- allows only whole numbers
```

サブタイプ CHARACTER は、そのベース型 CHAR と同じ値の集合を指定します。したがって、CHARACTER は無制約サブタイプです。ただし、サブタイプ INTEGER は、そのベース・タイプ NUMBER の値のサブセットのみを指定するため、INTEGER は制約付きサブタイプです。
.

サブタイプの定義

ユーザー独自のサブタイプは、次の構文を使用して、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。

```
SUBTYPE subtype_name IS base_type[(constraint)] [NOT NULL];
```

ここで subtype_name は後続の宣言で使用される型指定子です。base_type は任意のスカラーまたはユーザー定義の PL/SQL データ型で、constraint は精度およびスケールまたは最大サイズを指定できるベース型にのみ適用されます。

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
```

%TYPE または %ROWTYPE を使用してベース型を指定できます。%TYPE がデータベース列のデータ型を提供する場合、サブタイプはその列のサイズ制約（存在する場合）を継承します。ただし、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);
```

サブタイプを使用すると、範囲外の値を検出でき、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 Numeral を制限しています。プログラムで Numeral 変数の範囲外の数値を格納すると、PL/SQL は例外を呼び出します。

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

データ型の互換性

無制約のサブタイプは、そのベース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;
   ...
END;
```

異なるサブタイプでも、ベース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;
   ...
END;
```

また、異なるサブタイプの場合、ベース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(15);
   SUBTYPE Text IS VARCHAR2(1500);
   verb     Word;
   sentence Text(150);
BEGIN
   ...
   sentence := verb;
   ...
END;
```

データ型変換

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する必要があります。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、組込みファンクションを使用します。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用します。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用します。これらのファンクションの詳細は、『Oracle9*i* 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、変数またはパラメータを、別の型が期待されている箇所で使用できます。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。このような場合は、データ型変換ファンクションを使用する必要があります。表 3-1 は PL/SQL が実行できる暗黙的な変換を示しています。

注意：

- この表は、異なる表現を持つ型のみを示しています。CLOB と NCLOB、CHAR と NCHAR、VARCHAR と NVARCHAR2 など、同じ表現を持つ型は、相互に代用できます。
- CLOB と NCLOB の間で変換するには、変換ファンクション TO_CLOB および TO_NCLOB を使用する必要があります。
- TIMESTAMP、TIMESTAMP WITH TIME ZONE、TIMESTAMP WITH LOCAL TIME ZONE、INTERVAL DAY TO SECOND および INTERVAL YEAR TO MONTH は、いずれも DATE 型と同じ規則を使用して変換できます。ただし、これらの型は内部表現が異なるため、常に相互に変換できるとは限りません。様々な日付および時刻型間の暗黙的な変換の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

表 3-1 暗黙的な変換

	BIN_INT	BLOB	CHAR	CLOB	DATE	LONG	NUMBER	PLS_INT	RAW	UROWID	VARCHAR2
BIN_INT			X			X	X	X			X
BLOB									X		
CHAR	X			X	X	X	X	X	X	X	X
CLOB			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					X
UROWID			X								X
VARCHAR2	X		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、暗黙的変換は状況依存であるため、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用する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する必要があります。このとき、PL/SQL は、文字列をデフォルトの日付書式で戻すファンクション TO_CHAR をコールします。時刻やユリウス日付などの情報を得るには、書式マスクを使用して 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、デフォルトの日付書式を期待するファンクション TO_DATE をコールします。他の書式の日付を挿入するには、書式マスクを使用して 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する必要があります。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。ファンクション 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 進値を表している必要があります。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。

4

PL/SQL の制御構造

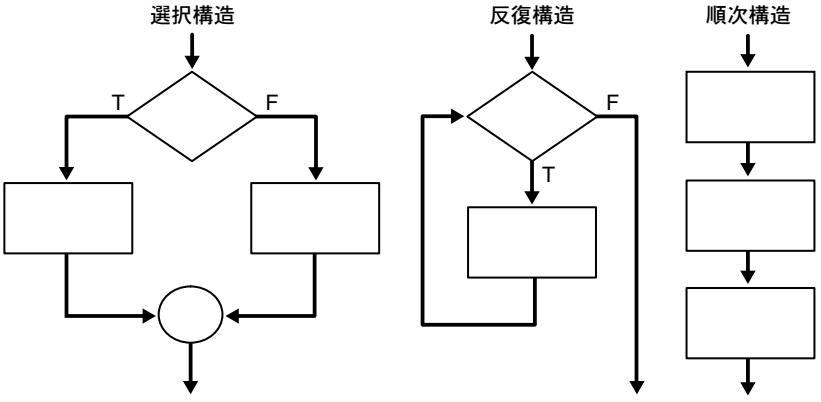
この章では、PL/SQL プログラムの制御の流れを構造化する方法を示します。入口と出口を 1 つずつ持つ、単純かつ強力な制御構造によって文を結合する方法を説明します。これらの構造を使用すると、どのような状況でも処理できます。また、これらの構造を適切に使用することで、優れた構造を持つプログラムを自然に作成できます。

この章の項目は、次のとおりです。

PL/SQL の制御構造の概要

構造定理によれば、どのコンピュータ・プログラムも、図 4-1 に示す基本的な制御構造を使用して書くことができます。これらを適当に組み合せれば、どのような問題でも取り扱うことができます。

図 4-1　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 文および CASE 文

状況に応じてアクションを選ぶ必要のある場面はよくあります。IF 文を使用すると、一連の文を条件に合せて実行できます。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 および IF-THEN-ELSIF の 3 つの形式があります。CASE 文は、単一の条件を評価して多数の代替アクションから選択するコンパクトな手段です。

IF-THEN 文

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のみ実行されます。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。次に例を示します。

```
IF sales > quota THEN
   bonus := 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 文

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のみ実行されます。このように、ELSE 句では必ず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 文

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する場合があります。IF 文の 3 番目の形式では、キーワード ELSIF（ELSEIF ではなく）を使用して、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。

CASE 文

IF 文と同様に、CASE 文では一連の文を選択して実行できます。ただし、CASE 文では、順序を選択するために複数のブール式ではなく選択子を使用します。（選択子は複数の選択肢から 1 つ選択するために値が使用される式です。第 2 章を参照してください。）IF 文と CASE 文を比較するために、学業成績の説明を出力する次のコードを考えます。

```
IF grade = 'A' THEN
   dbms_output.put_line('Excellent');
ELSIF grade = 'B' THEN
   dbms_output.put_line('Very Good');
ELSIF grade = 'C' THEN
   dbms_output.put_line('Good');
ELSIF grade = 'D' THEN
   dbms_output. put_line('Fair');
ELSIF grade = 'F' THEN
   dbms_output.put_line('Poor');
ELSE
   dbms_output.put_line('No such grade');
END IF;
```

5 つのブール式があることに注意してください。各インスタンスで、同じ変数 grade が 5 つの値 'A'、'B'、'C'、'D' または 'F' のどれかをテストしています。CASE 文を使用して、前述のコードを次のように書き直します。

```
CASE grade
   WHEN 'A' THEN dbms_output.put_line('Excellent');
   WHEN 'B' THEN dbms_output.put_line('Very Good');
   WHEN 'C' THEN dbms_output.put_line('Good');
   WHEN 'D' THEN dbms_output.put_line('Fair');
   WHEN 'F' THEN dbms_output.put_line('Poor');
   ELSE dbms_output.put_line('No such grade');
END CASE;
```

CASE 文の方が読みやすく効率的です。したがって、長い IF-THEN-ELSIF 文はできるかぎり CASE 文として書き直してください。

CASE 文は、キーワード CASE で始まります。キーワードの後に選択子（前述の例では変数 grade）があります。選択子式は、どんなに複雑でもかまいません。たとえば、ファンクション・コールを含めることができます。ただし、通常は、1 個の変数で構成されています。選択子式が評価されるのは 1 度のみです。生成される値は、BLOB、BFILE、オブジェクト型、PL/SQL レコード、索引付き表または VARRAY、ネストした表以外であれば、どんな PL/SQL データ型でもかまいません。

選択子の後に 1 つ以上の WHEN 句があり、各句が順番にチェックされます。選択子の値によって、どの句が実行されるかが決定されます。選択子の値が WHEN 句の式の値と等しければ、その WHEN 句が実行されます。たとえば、最後の例では、grade が 'C' であれば、

'Good' が出力されます。実行が失敗することはなく、WHEN 句が 1 つでも実行されると、制御が次の文に渡されます。

ELSE 句の機能は、IF 文の ELSE 句に似ています。前述の例では、学年が WHEN 句のオプションの 1 つでなければ、ELSE 句が選択され、'No such grade' という句が出力されます。ELSE 句はオプションです。ただし、ELSE 句を省略すると、PL/SQL では次の暗黙的な ELSE 句が追加されます。

```
ELSE RAISE CASE_NOT_FOUND;
```

CASE 文で暗黙的な ELSE 句が選択されると、PL/SQL は事前に定義された例外 CASE_NOT_FOUND を呼び出します。したがって、ELSE 句を省略しても、常にデフォルト・アクションがあることになります。

CASE 文は、キーワード END CASE で終了します。この 2 つのキーワードは、空白で区切る必要があります。CASE 文の書式は、次のとおりです。

```
[<<label_name>>]
CASE selector
   WHEN expression1 THEN sequence_of_statements1;
   WHEN expression2 THEN sequence_of_statements2;
   ...
   WHEN expressionN THEN sequence_of_statementsN;
  [ELSE sequence_of_statementsN+1;]
END CASE [label_name];
```

PL/SQL ブロックと同様に、CASE 文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、CASE 文の先頭に置きます。オプションとして、CASE 文の末尾にもラベル名を付けることができます。

CASE 文の実行中に呼び出された例外は、通常の方法で処理されます。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。

CASE 文は CASE 式の代替であり、各 WHEN 句が式になっています。詳細は、2-31 ページの「CASE 式」を参照してください。

検索 CASE 文

PL/SQL には、次の書式の検索 CASE 文も用意されています。

```
[<<label_name>>]
CASE
   WHEN search_condition1 THEN sequence_of_statements1;
   WHEN search_condition2 THEN sequence_of_statements2;
   ...
   WHEN search_conditionN THEN sequence_of_statementsN;
  [ELSE sequence_of_statementsN+1;]
END CASE [label_name];
```

検索 CASE 文には選択子はありません。また、WHEN 句には、任意の型の値になる式ではなく、ブール値になる検索条件が含まれています。次に例を示します。

```
CASE
   WHEN grade = 'A' THEN dbms_output.put_line('Excellent');
   WHEN grade = 'B' THEN dbms_output.put_line('Very Good');
   WHEN grade = 'C' THEN dbms_output.put_line('Good');
   WHEN grade = 'D' THEN dbms_output.put_line('Fair');
   WHEN grade = 'F' THEN dbms_output.put_line('Poor');
   ELSE dbms_output.put_line('No such grade');
END CASE;
```

検索条件は順番に評価されます。各検索条件のブール値によって、どの WHEN 句が実行されるかが決定されます。検索条件が TRUE になると、その WHEN 句が実行されます。WHEN 句が 1 つでも実行されると、制御が次の文に渡されるため、後続の検索条件は評価されません。

TRUE になる検索条件がなければ、ELSE 句が実行されます。ELSE 句はオプションです。ただし、ELSE 句を省略すると、PL/SQL では次の暗黙的な ELSE 句が追加されます。

```
ELSE RAISE CASE_NOT_FOUND;
```

検索 CASE 文の実行中に呼び出された例外は、通常の方法で処理されます。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。

PL/SQL 条件文のガイドライン

次の例のような IF 文の使用は避けてください。

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

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 句を使用してください。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。

```
IF condition1 THEN            |     IF condition1 THEN
   statement1;                |        statement1;
ELSE                          |     ELSIF condition2 THEN
   IF condition2 THEN         |        statement2;
      statement2;             |     ELSIF condition3 THEN
   ELSE                       |        statement3;
      IF condition3 THEN      |     END IF;
         statement3;          |
      END IF;                 |
   END IF;                    |
END IF;                       |
```

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。

単一の式を複数の値と比較する場合は、IF と ELSIF 句の組合せのかわりに単一の CASE 文を使用すると、論理を簡素化できます。

反復制御 : LOOP 文と EXIT 文

LOOP 文を使用すると、一連の文を複数回実行できます。LOOP 文には LOOP、WHILE-LOOP および 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。処理の続行を望まない場合、または不可能になった場合は、EXIT 文を使用し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 の 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。次に例を示します。

```
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 文を使用して PL/SQL のブロックを終了することはできません。

```
BEGIN
   ...
   IF credit_rating < 3 THEN
      ...
      EXIT;  -- not allowed
   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 文を使用します。詳細は、8-8 ページの「RETURN 文の使用」を参照してください。

EXIT-WHEN

EXIT-WHEN 文を使用すると、ループを条件に合せて終了できます。EXIT 文が見つかると、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。次に例を示します。

```
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用できます。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、末尾のラベルを使用してわかりやすくします。

どちらの形式の EXIT 文でも、カレント・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、終了する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。次に例を示します。

```
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 よりも大きい場合、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成できます。

```
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用します。

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

ループの中の文で、ブール変数に新しい値を代入してください。代入しないと無限ループになります。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。繰返しの範囲は、キーワード FOR と LOOP に囲まれた反復スキームの一部です。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 回のみ実行されます。

```
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 を使用すると、反復は上限から下限の向きに進みます。繰返しが 1 回起こるたびに、ループ・カウンタが 1 つ減らされます。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 i IN REVERSE 1..3 LOOP  -- assign the values 3,2,1 to i
   sequence_of_statements  -- executes three times
END LOOP;
```

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;  -- not allowed
   END IF;
END LOOP;
```

反復スキーム

ループ範囲の境界にはリテラル、変数または式を使用できますが、必ず整数に評価されるものにしてください。それ以外の場合、PL/SQL は事前定義の例外 VALUE_ERROR を呼び出します。次の例に示すように、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 である必要があります。

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

内部的に、PL/SQL は PLS_INTEGER 一時変数に境界の値を代入します。さらに、必要に応じてその値を最も近い整数に四捨五入します。PLS_INTEGER の大きさの範囲は、-2**31 ～ 2**31 です。このため、範囲外の数値を評価した場合、PL/SQL が代入をすると、次に示す数値オーバーフローのエラーが発生します。

```
DECLARE
   hi NUMBER := 2**32;
BEGIN
   FOR j IN 1..hi LOOP  -- causes a 'numeric overflow' error
      ...
   END LOOP;
END;
```

言語によっては、STEP 句を使用して異なる増分値（たとえば、1 ではなく 5）を指定できるものがあります。PL/SQL はこのような構造を持っていませんが、作成するのは簡単です。FOR ループの内部で、ループ・カウンタへの各参照に新しい増分値を乗じます。次の例では、本日の日付を索引付き表の要素 5、10、および 15 に代入します。

```
DECLARE
   TYPE DateList IS TABLE OF DATE INDEX BY BINARY_INTEGER;
   dates DateList;
   k CONSTANT INTEGER := 5;  -- set new increment
BEGIN
   FOR j IN 1..3 LOOP
      dates(j*k) := SYSDATE;  -- multiply loop counter by increment
   END LOOP;
   ...
END;
```

動的な範囲

次に示すように、PL/SQL ではループの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 emp_count の値を戻します。

ループの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;  -- not allowed
```

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ため、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用する場合は、次のようにラベルとドット表記法を使用する必要があります。

```
<<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用します。

```
<<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

EXIT 文を使用すると、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る必要があった場合でも、カレント・ループのみでなく、外側のループも終了できます。これを行うには、終了する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

GOTO 文と NULL 文は、PL/SQL プログラミングにとって IF 文や LOOP 文ほど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 文を使用する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ります）になり、理解やメンテナンスが難しくなりがちです。GOTO 文はなるべく使用し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 文を使用するのではなく、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用します。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ため、使用できません。

```
DECLARE
   done  BOOLEAN;
BEGIN
   ...
   FOR i IN 1..50 LOOP
      IF done THEN
         GOTO end_loop;
      END IF;
      ...
   <<end_loop>>  -- not allowed
   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 文でカレント・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最初の外側のブロックに分岐します。

制限

GOTO 文の宛先として使用できないものがあります。特に、GOTO 文は IF 文、CASE 文、LOOP 文またはサブブロックには分岐できません。たとえば、次の GOTO 文は許可されません。

```
BEGIN
   ...
   GOTO update_row;  -- can't 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。同様に、GOTO 文では、ある CASE 文の WHEN 句から別の句へ分岐できません。

```
BEGIN
   ...
   IF valid THEN
      ...
      GOTO update_row;  -- can't 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;  -- can't 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;  -- can't 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カレント・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;  -- can't 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、制御を次の文に渡す以外はなにもしません。条件構造内での NULL 文は、可能性を考慮した結果、アクションが不要であることを読み手に伝えます。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 文または少なくとも 1 つの実行可能文が必要な他の場所では、NULL 文が構文の条件を満たします。次の例では、NULL 文によって、成績優秀な従業員のみがボーナスを受け取ることが強調されています。

```
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、スタブを簡単に作成するためにも使用できます。**スタブ**はダミーのサブプログラムです。スタブを使用すると、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を定義せずにすみます。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 つの文が存在している必要がある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;
```

5

PL/SQL のコレクションとレコード

多くのプログラミング・テクニックでは、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用します。このようなテクニックをデータベース・アプリケーションでサポートするために、PL/SQL には TABLE および VARRAY というデータ型が用意されており、索引付き表、ネストした表および可変サイズの配列を宣言できます。この章では、これらの型を使用してデータの集まり（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する方法を説明します。また、異なるデータ型が関連している場合、データ型 RECORD を使用して、それらを 1 つの論理単位として扱う方法も説明します。

この章の項目は、次のとおりです。

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。コレクションは、リスト、配列および他のよく知られているデータ型を包含した一般的な概念です。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、集合の中での要素の位置が決まります。

PL/SQL には、次のコレクション型が用意されています。

- **索引付き表**は、**結合配列**とも呼ばれ、添字の値に任意の数字または文字列を使用して要素を参照できます。（他のプログラミング言語でのハッシュ表に似ています。）
- **ネストした表**は、任意の数の要素を保持します。ネストした表では、添字に連番が使用されます。等価の SQL の型を定義すると、ネストした表をデータベース表に格納し、SQL を介して操作できます。
- **VARRAY**（可変サイズの配列）は、固定数の要素を保持します（ただし、要素の数は実行時に変更できます）。VARRAY では、添字に連番を使用します。等価の SQL の型を定義すると、データベース表に VARRAY を格納できます。VARRAY は SQL を介して格納および取得できますが、ネストした表に比べると柔軟性は低くなります。

コレクションは 1 次元のみですが、コレクションを要素に持つコレクションを作成すると、多次元配列のモデルを作成できます。

アプリケーションでコレクションを使用するには、1 つ以上の PL/SQL の型を定義し、それらの型の変数を定義します。コレクション型は、プロシージャ、ファンクションまたはパッケージで定義できます。コレクションの変数をパラメータとして渡し、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を移動できます。

単一の値よりも複雑なデータを参照するために、PL/SQL レコードまたは SQL オブジェクト型をコレクションに格納できます。ネストした表と VARRAY は、オブジェクト型の属性にすることもできます。

ネストした表

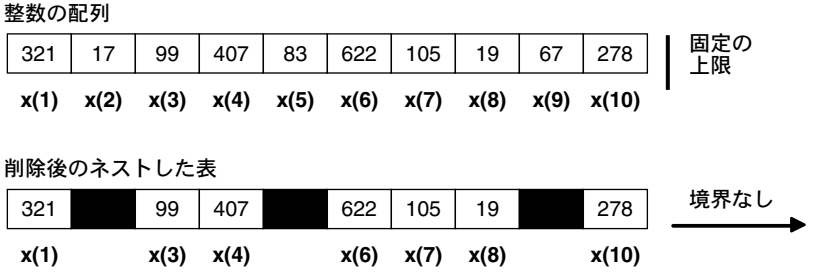
データベース内では、ネストした表は 1 列のデータベース表と考えられます。Oracle では、ネストした表の行を特に順序付けずに格納します。ただし、ネストした表を PL/SQL 変数の中に取り出すと、それらの行に 1 から始まる連続した添字が付けられます。これによって、個々の行に配列のようにアクセスできるようになります。

PL/SQL のネストした表は、1 次元配列と類似しています。要素にネストした表を持つネストした表を作成すると、多次元配列のモデルを作成できます。

ネストした表と配列には重要な相違点が 2 つあります。

1. 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 5-1 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 5-1　配列とネストした表との相違点

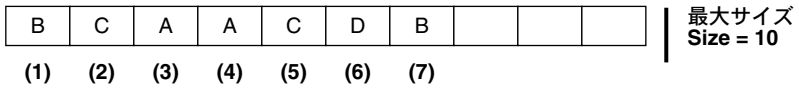


2. 配列は密である必要があります（つまり添字は連続している必要があります）。そのため、個々の要素を配列から削除することはできません。ネストした表も最初は密ですが、疎にすることができます（つまり添字が連続していなくてもかまいません）。したがって、組込みプロシージャ `DELETE` を使用して、ネストした表から要素を削除できます。その場合、索引に欠番が生じますが、組込みファンクション `NEXT` を使用すると、連続した添字に対する反復処理を実行できます。

VARRAY

VARRAY 型の項目は、VARRAY と呼ばれます。VARRAY を使用すると、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用します（図 5-2 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 5-2 サイズ 10 の VARRAY



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、VARRAY Grades の現在の上限は 7 ですが、8、9、10 などに拡張できます。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変更できます。

結合配列（索引付き表）

結合配列は、キーと値のペアのセットです。各キーは一意で、配列内の対応する値を検索するために使用されます。キーは、整数または文字列にできます。

初めてキーを使用して値を代入すると、そのキーが結合配列に追加されます。その後、同じキーを使用して値を代入すると、同じエントリが更新されます。SQL 表の主キーを使用するか、または複数の文字列を連結して一意の値を形成することで、一意であるキーを選択することが重要です。

次に、文字列のキーを使用した結合配列型の宣言と、この型の 2 つの配列の例を示します。

```
DECLARE
  TYPE population_type IS TABLE OF NUMBER INDEX BY VARCHAR2(64);
  country_population population_type;
  howmany NUMBER;
  which VARCHAR2(64);

BEGIN
  country_population('Greenland') := 100000;
  country_population('Iceland') := 750000;
  howmany := country_population('Greenland');

  continent_population('Australia') := 30000000;
  continent_population('Antarctica') := 1000; -- Creates new entry
  continent_population('Antarctica') := 1001; -- Replaces previous value
```

```
  which := continent_population.FIRST; -- Returns 'Antarctica'
-- as that comes first alphabetically.
  which := continent_population.LAST; -- Returns 'Australia'
  howmany := continent_population(continent_population.LAST);
-- Returns the value corresponding to the last key, in this
-- case the population of Australia.
END;
/
```

結合配列は、任意のサイズのデータ・セットを表すために役立ち、配列内での要素の位置が不明でも、配列の全要素をループせずに個々の要素をすばやく参照します。単純な SQL 表のように、主キーに基づいて値を取り出すことができます。単純な参照データの一時記憶域では、結合配列によって、SQL 表に必要なディスク領域の使用やネットワーク操作を回避できます。

結合配列は、永続的なデータの格納ではなく一時的なデータを意図しているため、INSERT や SELECT INTO などの SQL 文では使用できません。結合配列は、パッケージで型を宣言し、パッケージ本体に値を代入することで、データベース・セッションの間を永続的に維持できます。

グローバリゼーション設定が結合配列の VARCHAR2 キーに与える影響

VARCHAR2 のキー値を持つ結合配列を使用したセッション中に、各国語またはグローバリゼーションの設定が変わると、プログラムでランタイム・エラーとなる可能性があります。たとえば、セッション中に NLS_COMP 初期化パラメータや NLS_SORT 初期化パラメータを変更すると、NEXT や PRIOR などのメソッドで例外が発生する可能性があります。セッション中にこれらの設定の変更が必要な場合は、必ず元の値に戻してから、結合配列での操作を実行してください。

キーに文字列を使用して結合配列を宣言する場合、その宣言では VARCHAR2 型、STRING 型または LONG 型を使用する必要があります。結合配列を参照するためのキー値には、NCHAR 型や NVARCHAR2 型などの異なる型を使用できます。TO_CHAR ファンクションで VARCHAR2 に変換できる場合は、DATE などの型を使用することもできます。

ただし、他の型を使用する場合は、キーに使用する値の一貫性と一意性に注意してください。たとえば、NLS_DATE_FORMAT 初期化パラメータが変更された場合は、SYSDATE の文字列値が変更され、その結果、array_element(SYSDATE) では、以前と異なる結果が生じます。異なる 2 つの NVARCHAR2 値が、(特定の各国語キャラクタのかわりに疑問符が使用されて) 同一の VARCHAR2 値に変わる可能性もあります。この場合、array_element(national_string1) と array_element(national_string2) は、同一の要素を参照します。

データベース・リンクを使用して、リモート・データベースへのパラメータとして結合配列を渡すと、2 つのデータベースで、グローバリゼーション設定が異なる可能性があります。リモート・データベースで FIRST や NEXT などの操作を実行すると、文字順序が元のコレクションとは異なる場合でも、リモート・データベース自体の文字順序が使用されます。

キャラクタ・セットの相違によって、一意であった 2 つのキーがリモート・データベース上では一意でない場合、プログラムは VALUE_ERROR 例外を受け取ります。

使用する PL/SQL コレクション型の選択

他の言語を使用するコードまたはビジネス・ロジックがすでに存在する場合、通常は、その言語の配列を変換して、型を PL/SQL のコレクション型に直接設定できます。

- 他の言語の配列は、PL/SQL の VARRAY になります。
- 他の言語の設定およびバッグは、PL/SQL のネストした表になります。
- 他の言語のハッシュ表や他の無秩序な参照表は、PL/SQL の結合配列になります。

オリジナルのコードを記述していたり、最初からビジネス・ロジックを設計している場合は、各状況に適切なコレクション型を判断するために各コレクション型の長所を考慮してください。

ネストした表と結合配列の選択

ネストした表と結合配列（以前の索引付き表）はともに、同じような添字表記法を使用しますが、パラメータ受け渡しの永続性と容易性の点で、異なる特性があります。

ネストした表はデータベースの列に格納できますが、結合配列をデータベースの列に格納することはできません。ネストした表は、永続的な格納が必要な重要なデータ関連の格納に適しています。

結合配列は、プロシージャのコールやパッケージの初期化のたびに、コレクションをメモリー内に構成される、比較的小規模な参照表に適しています。結合配列のサイズには固定制限がないため、量が事前にわからない情報を収集する場合に効果的です。結合配列の添字には、負数や非連続の数字を指定したり、場合によっては数字ではなく文字列の値を使用できるため、その索引値には柔軟性があります。

PL/SQL は、数値のキー値を使用するホスト配列と結合配列との間で自動的に変換を行います。データベース・サーバーとの間でのコレクションの受渡しには、無名 PL/SQL ブロックを使用してホスト配列を結合配列にバルク・バインド入出力するのが、最も効果的な方法です。

ネストした表と VARRAY との使い分け

要素の数が事前に判明しており、すべての要素が通常、順序どおりにアクセスされる場合は、VARRAY が適しています。データベースに格納されている間、VARRAY はその順序と添字を保持しています。

各 VARRAY は、それが列である表の内部（VARRAY が 4KB 未満の場合）か、同じ表領域内の表の外部（VARRAY が 4KB を超える場合）のいずれかに、単一のオブジェクトとして格納されます。VARRAY のすべての要素は、同時に更新または取得する必要があります。これは、すべての要素に対してなんらかの操作を同時に実行する場合に最適です。しかし、この方法で多数の要素を格納および取得することは、現実的ではありません。

ネストした表は疎密な場合があります。最後から順に項目を削除せずに、任意の要素を削除できます。ネストした表のデータは、**記憶域表**の行外部に格納されます。この記憶域表は、ネストした表に対応付けられたシステム生成によるデータベース表です。この記憶域表によって、ネストした表は、コレクションの一部の要素にのみ影響を与える問合せと更新に適した内容になります。ネストした表ではデータベースへの格納時に順序と添字が保持されないため、表の格納と取得では信頼できた順序と添字は、ネストした表では信頼できません。

コレクション型の定義

コレクションを作成するには、コレクション型を定義した後、その型の変数を宣言します。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。

コレクションは、他の型や変数と同じ有効範囲とインスタンス化の規則に従います。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。

ネストした表

ネストした表の場合には、次の構文を使用します。

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

type_name は、コレクションを宣言するために後で使用する型指定子です。PL/SQL 内で宣言されたネストした表の場合、*element_type* は、次の型を除く PL/SQL のデータ型です。

```
REF CURSOR
```

SQL でグローバルに宣言されたネストした表には、要素型に追加制約があります。次の要素型は使用できません。

```
BINARY_INTEGER, PLS_INTEGER
BOOLEAN
LONG、LONG RAW
```

```
NATURAL、NATURALN
POSITIVE、POSITIVEN
REF CURSOR
SIGNTYPE
STRING
```

VARRAY

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];
```

type_name と *element_type* の意味は、ネストした表の場合と同じです。

size_limit は、配列内の最大要素数を表す正の整数リテラルです。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 個以内の日付を格納する型を定義します。

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

結合配列

結合配列（索引付き表とも呼ばれます）の場合には、次の構文を使用します。

```
TYPE type_name IS TABLE OF element_type [NOT NULL]
   INDEX BY [BINARY_INTEGER | PLS_INTEGER | VARCHAR2(size_limit)];
   INDEX BY key_type;
```

key_type には、BINARY_INTEGER または PLS_INTEGER の数値を指定できます。VARCHAR2 またはそのサブタイプ VARCHAR、STRING または LONG のいずれかを指定することもできます。VARCHAR2 ベースのキーを使用するには、VARCHAR2(32760) のキーの型を宣言することになる LONG の場合を除いて、キーの長さを指定する必要があります。RAW、LONG RAW、ROWID、CHAR および CHARACTER の各型は、結合配列のキーとしては使用できません。

初期化の句は必要ありません（指定できません）。

VARCHAR2 ベースのキーを使用する結合配列の要素を参照する場合は、TO_CHAR ファンクションで VARCHAR2 に変換できるかぎり、DATE または TIMESTAMP などの別の型を使用できます。

索引付き表は、主キー値を索引として使用してデータを格納できます。この場合、連続したキー値とはなりません。次の例では、添字が 1 ではなく 7468 である単一のレコードを索引付き表に格納します。

```
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 = 7468;
END;
```

PL/SQL コレクション型に相当する SQL の型の定義

ネストした表および VARRAY をデータベース表の内部に格納するには、CREATE TYPE 文を使用して SQL の型を宣言することも必要です。SQL の型は、列としてまたは SQL オブジェクト型の属性として使用できます。

PL/SQL 内で相当する型を宣言するか、PL/SQL の変数宣言で SQL の型名を使用できます。

ネストした表の例

次の SQL*Plus スクリプトは、SQL で宣言したネストした表をオブジェクト型の属性として使用する方法を示しています。

```
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。

VARRAY の例

次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 VARCHAR2 型が格納されます。

```
-- Each project has a 16-character code name.
-- We will store up to 50 projects at a time in a database column.
CREATE TYPE ProjectList AS VARRAY(50) OF VARCHAR2(16);
/
CREATE TABLE department (  -- create database table
   dept_id  NUMBER(2),
   name     VARCHAR2(15),
   budget   NUMBER(11,2),
-- Each department can have up to 50 projects.
   projects ProjectList)
/
```

PL/SQL のコレクション変数の宣言

コレクション型を定義した後は、その型の変数を宣言できます。NUMBER や INTEGER などの事前定義の型と同様に、宣言では新しい型名を使用します。

例 : ネストした表、VARRAY および結合配列の宣言

```
DECLARE
   TYPE nested_type IS TABLE OF VARCHAR2(20);
   TYPE varray_type IS VARRAY(50) OF INTEGER;
   TYPE associative_array_type IS TABLE OF NUMBER
      INDEXED BY BINARY_INTEGER;
   v1 nested_type;
   v2 varray_type;
   v3 associative_array_type;
```

%TYPE の例

%TYPE を使用すると、事前に宣言したコレクションのデータ型を指定できます。この指定によって、コレクションの定義を変更すると、要素の数または要素の型に依存している他の変数が自動的に更新されます。

```
DECLARE
   TYPE Platoon IS VARRAY(20) OF Soldier;
   p1 Platoon;
-- If we change the number of soldiers in a platoon, p2 will
-- reflect that change when this block is recompiled.
   p2 p1%TYPE;
```

例 : プロシージャのパラメータをネストした表として宣言する

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。それにより、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すことができます。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;
```

パッケージ外部から PERSONNEL.AWARD_BONUSES をコールするには、PERSONNEL.STAFF 型の変数を宣言し、その変数をパラメータとして渡します。

コレクション型は、ファンクション仕様部の RETURN 句にも指定できます。

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

例 : %TYPE と %ROWTYPE によるコレクション要素型の指定

要素型を指定するには、%TYPE を使用し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 を使用して、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
```

例 : レコードとしての VARRAY

次の例では、RECORD 型を使用して、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;
```

例 : コレクション要素の NOT NULL 制約

NOT NULL 制約は、要素型に対しても指定できます。

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

コレクションの初期化と参照

ネストした表または VARRAY は、初期化されるまでは基本構造的に NULL です。つまり、コレクション自体が NULL で、コレクションの要素ではありません。ネストした表または VARRAY を初期化するには、コンストラクタを使用します。このコンストラクタは、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です。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します。

VARRAY やネストした表の変数に対しては、コンストラクタを明示的にコールする必要があります （第 3 のコレクションである結合配列は、コンストラクタを使用しません）。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場合にコールできます。

例 : ネストした表のコンストラクタ

次の例では、複数の要素をコンストラクタ CourseList() に渡します。このコンストラクタは、それらの要素が含まれたネストした表を戻します。

```
DECLARE
   TYPE CourseList IS TABLE OF VARCHAR2(16);
   my_courses CourseList;
BEGIN
   my_courses :=
      CourseList('Econ 2010', 'Acct 3401', 'Mgmt 3100');
END;
```

ネストした表では最大サイズが宣言されていないため、コンストラクタには必要な数だけ要素を配置できます。

例 : VARRAY のコンストラクタ

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
   TYPE ProjectList IS VARRAY(50) OF VARCHAR2(16);
   accounting_projects ProjectList;
BEGIN
   accounting_projects :=
      ProjectList('Expense Report', 'Outsourcing', 'Auditing');
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。

例 : NULL の要素を含むコンストラクタのコレクション

NULL の要素は、NOT NULL 制約を指定しなかぎり、コンストラクタに渡すことができます。次に例を示します。

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

例 : コレクション宣言とコンストラクタの組合せ

コレクションは、そのコレクションの宣言で初期化できます。これはプログラミング上好ましい習慣です。

```
DECLARE
   TYPE CourseList IS TABLE OF VARCHAR2(16);
   my_courses CourseList :=
      CourseList('Art 1111', 'Hist 3100', 'Engl 2005');
```

例 : 空の VARRAY コンストラクタ

引数を指定しないでコンストラクタをコールすると、空（NULL ではない）の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;
```

この場合は、そのコレクションの EXTEND メソッドをコールして、後で要素を追加できます。

例 : SQL 文の中でのネストした表のコンストラクタ

この例では、複数のスカラー値およびネストした表 CourseList を SOPHOMORES 表に挿入しています。

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

例 : SQL 文の中での VARRAY コンストラクタ

この例では、行をデータベース表 DEPARTMENT に挿入しています。VARRAY コンストラクタ ProjectList() によって、列 PROJECTS の値が指定されます。

```
BEGIN
  INSERT INTO department
    VALUES(60, 'Security', 750400,
      ProjectList('New Badges', 'Track Computers', 'Check Exits'));
```

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、次の構文を使用してその添字を指定します。

collection_name(*subscript*)

subscript は、ほとんどの場合、結果が整数になる式か、または文字列キーで宣言した結合配列の場合は VARCHAR2 です。

使用できる添字範囲は、次のとおりです。

- ネストした表の場合は、1 ～ 2**31 です。
- VARRAY の場合は、1 ～ *size_limit*（宣言に指定した制限）です。
- 数値キーの結合配列の場合は、-2**31 ～ 2**31 です。
- 文字列キーの結合配列の場合、キーの長さおよび使用可能な値の数は、型宣言に指定した VARCHAR2 の長さ制限およびデータベース・キャラクタ・セットによって異なります。

例 : ネストした表の要素の添字による参照

この例は、ネストした表 NAMES の要素を参照する方法を示しています。

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

例 : ネストした表の要素をパラメータとして渡す

この例は、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ることを示しています。

```
DECLARE
   TYPE Roster IS TABLE OF VARCHAR2(15);
   names Roster := Roster('J Hamil', 'D Piro', 'R Singh');
   i BINARY_INTEGER := 2;
BEGIN
   verify_name(names(i));  -- call procedure
END;
```

コレクションの代入

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・コールよって、別のコレクションに代入できます。

次の構文を使用すると、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です。

例 : データ型の互換性

この例は、代入の操作では、コレクションに同じデータ型が必要であることを示しています。要素型が同じであることのみでは不十分です。

```
DECLARE
   TYPE Clientele IS VARRAY(100) OF Customer;
   TYPE Vips IS VARRAY(100) OF Customer;
-- These first two variables have the same datatype.
   group1 Clientele := Clientele(...);
   group2 Clientele := Clientele(...);
-- This third variable has a similar declaration,
-- but is not the same type.
   group3 Vips := Vips(...);
BEGIN
-- Allowed because they have the same datatype
   group2 := group1;
-- Not allowed because they have different datatypes
   group3 := group2;
END;
```

例 : NULL 値のネストした表への代入

基本構造的に NULL のネストした表または VARRAY を、第 2 のネストした表または VARRAY に代入します。この場合、第 2 のコレクションはに再初期化が必要です。

```
DECLARE
   TYPE Clientele IS TABLE OF VARCHAR2(64);
  -- This nested table has some values.
   group1 Clientele := Clientele('Customer 1','Customer 2');
-- This nested table is not initialized ("atomically null").
   group2 Clientele;
BEGIN
-- At first, the test IF group1 IS NULL yields FALSE.
-- Then we assign a null nested table to group1.
   group1 := group2;
-- Now the test IF group1 IS NULL yields TRUE.
-- We must use another constructor to give it some values.
END;
```

同様に、コレクションに NULL 値を代入することは、コレクションを基本構造的に NULL にします。

例 : コレクション代入で予想される例外

値のコレクション要素への代入では、例外が発生する可能性があります。

- 添字が NULL であったり、正しいデータ型に変換できない場合、PL/SQL は事前定義の例外 VALUE_ERROR を呼び出します。通常、添字は整数である必要があります。結合配列では、VARCHAR2 の添字を使用するように宣言することもできます。
- 添字が初期化されていない要素を参照した場合、PL/SQL は SUBSCRIPT_BEYOND_COUNT を呼び出します。
- コレクションが基本構造的に NULL の場合、PL/SQL は COLLECTION_IS_NULL を呼び出します。

```
DECLARE
   TYPE WordList IS TABLE OF VARCHAR2(5);
   words WordList;
BEGIN
   /* Assume execution continues despite the raised exceptions. */
-- Raises COLLECTION_IS_NULL. We haven't used a constructor yet.
-- This exception applies to varrays and nested tables, but not
-- associative arrays which don't need a constructor.
   words(1) := 10;
-- After using a constructor, we can assign values to the elements.
   words := WordList(10,20,30);
-- Any expression that returns a VARCHAR2(5) is OK.
   words(1) := 'yes';
   words(2) := words(1) || 'no';
```

```
-- Raises VALUE_ERROR because the assigned value is too long.
   words(3) := 'longer than 5 characters';
-- Raises VALUE_ERROR because the subscript of a nested table must
-- be an integer.
   words('B') := 'dunno';
-- Raises SUBSCRIPT_BEYOND_COUNT because we only made 3 elements
-- in the constructor. To add new ones, we must call the EXTEND
-- method first.
   words(4) := 'maybe';
END;
```

コレクションの比較

コレクションが NULL かどうかはチェックできますが、2 つのコレクションが同一かどうかのテストはできません。以上、未満などの条件も使用できません。

例 : コレクションが NULL かどうかのチェック

ネストした表と VARRAY は、基本構造的に NULL の場合があるため、NULL かどうかをテストできます。

```
DECLARE
   TYPE Staff IS TABLE OF Employee;
   members Staff;
BEGIN
  -- Condition yields TRUE because we haven't used a constructor.
   IF members IS NULL THEN ...
END;
```

例 : 2 つのコレクションの比較

コレクションは、その等価性を直接比較することはできません。たとえば、次の IF 条件は誤りです。

```
DECLARE
   TYPE Clientele IS TABLE OF VARCHAR2(64);
   group1 Clientele := Clientele('Customer 1', 'Customer 2');
   group2 Clientele := Clientele('Customer 1', 'Customer 3');
BEGIN
-- Equality test causes compilation error.
   IF group1 = group2 THEN
      ...
   END IF;
END;
```

この制限は、暗黙的な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 リストには使用できません。

このような比較操作を行う場合は、2 つのコレクションが等しい、大きい、小さいなどを判断する手段について、ユーザー独自の概念を定義し、コレクションとその要素の調査結果を TRUE または FALSE の値で戻すような、1 つ以上のファンクションを記述する必要があります。

SQL 文での PL/SQL コレクションの使用

コレクションを使用すると、PL/SQL 内で複雑なデータ型を操作できます。添字を計算してメモリー内の特定の要素を処理し、SQL を使用してその結果をデータベース表に格納するようにプログラムを記述できます。

例 : PL/SQL のネストした表に対応した SQL の型の作成

SQL*Plus では、定義が PL/SQL のネストした表と VARRAY に対応した SQL の型を作成できます。

```
SQL> CREATE TYPE CourseList AS TABLE OF VARCHAR2(64);
```

次の SQL の型は、データベース表の列として使用できます。

```
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 内の各項目は、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。データベース表にネストした表の列がある場合は常に、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に名前を指定します。Oracle はネストした表のデータをこの記憶域表に格納します。

例 : ネストした表のデータベース表への挿入

データベース表には、データを入れることができます。表のコンストラクタによって、単一の列 COURSES に入るすべての値が指定されます。

```
BEGIN
   INSERT INTO department
      VALUES('English', 'Lynn Saunders', 'Breakstone Hall 205',
         CourseList('Expository Writing',
                    'Film and Literature',
                    'Modern Science Fiction',
                    'Discursive Writing',
                    'Modern English Grammar',
```

```
                    'Introduction to Shakespeare',
                    'Modern Drama',
                    'The Short Story',
                    'The American Novel'));
END;
```

例 : PL/SQL のネストした表をデータベース表から取得する

英語学部が提供するすべてのコースを、PL/SQL のネストした表に取り出すことができます。

```
DECLARE
   english_courses CourseList;
BEGIN
   SELECT courses INTO english_courses FROM department
      WHERE name = 'English';
END;
```

PL/SQL 内では、ネストした表の要素をループし、TRIM や EXTEND などのメソッドを使用し、要素の一部またはすべてを更新することによって、ネストした表を操作できます。その後、更新した表をデータベースに再度格納できます。

例 : データベース表でのネストした表の更新

英語学部が提供するコースのリストは、改訂することができます。

```
DECLARE
   new_courses CourseList :=
      CourseList('Expository Writing',
                 'Film and Literature',
                 'Discursive Writing',
                 'Modern English Grammar',
                 'Realism and Naturalism',
                 'Introduction to Shakespeare',
                 'Modern Drama',
                 'The Short Story',
                 'The American Novel',
                 '20th-Century Poetry',
                 'Advanced Workshop in Poetry');
BEGIN
   UPDATE department
      SET courses = new_courses WHERE name = 'English';
END;
```

VARRAY の例

SQL*Plus で、次のようにオブジェクト型 Project を定義します。

```
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 の列を含むリレーショナル表 department を次のように作成します。

```
SQL> CREATE TABLE department (
  2  dept_id  NUMBER(2),
  3  name     VARCHAR2(15),
  4  budget   NUMBER(11,2),
  5  projects ProjectList);
```

列 projects の中の各項目は、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),
                     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;
```

SQL での個々のコレクション要素の操作

デフォルトの SQL 操作では、個々の要素ではなく、コレクション全体が格納および取得されます。SQL で、コレクションの個々の要素を操作するには、TABLE 演算子を使用します。TABLE 演算子は、副問合せを使用して VARRAY またはネストした表を抽出します。その結果、INSERT 文、UPDATE 文または DELETE 文が、トップレベルの表ではなく、ネストした表に適用されます。

例 : SQL による要素のネストした表への挿入

次の例では、列 COURSES に格納されている歴史学部のネストした表に行を追加します。

```
BEGIN
-- The TABLE operator makes the statement apply to the nested
-- table from the 'History' row of the DEPARTMENT table.
   INSERT INTO
      TABLE(SELECT courses FROM department WHERE name = 'History')
      VALUES('Modern China');
END;
```

例 : SQL によるネストした表内の要素の更新

次の例は、心理学部が提供するいくつかのコース名を短縮します。

```
BEGIN
   UPDATE TABLE(SELECT courses FROM department
         WHERE name = 'Psychology')
      SET credits = credits + adjustment
      WHERE course_no IN (2200, 3540);
END;
```

例 : SQL によるネストした表からの単一要素の取得

次の例では、歴史学部が提供する特定のコースのタイトルを取り出します。

```
DECLARE
   my_title VARCHAR2(64);
BEGIN
-- We know that there is one history course with 'Etruscan'
-- in the title. This query retrieves the complete title
-- from the nested table of courses for the History department.
  SELECT title INTO my_title
    FROM
      TABLE(SELECT courses FROM department WHERE name = 'History')
```

```
    WHERE name LIKE '%Etruscan%';
END;
```

例 : SQL によるネストした表からの要素の削除

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;
```

例 : SQL による VARRAY からの要素の取得

次の例では、管理部の第 4 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;
```

例 : SQL による VARRAY の INSERT、UPDATE および DELETE 操作の実行

現在のところ、VARRAY の個々の要素は INSERT 文、UPDATE 文または DELETE 文中で参照できません。VARRAY 全体を取り出し、PL/SQL プロシージャ文を使用してその要素の追加、削除または更新を行い、次に、変更した VARRAY をデータベース表に戻す必要があります。

次の例で、ストアド・プロシージャ ADD_PROJECT は、指定された位置にある department のプロジェクト・リスト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_projects 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;
```

例 : PL/SQL のネストした表での INSERT、UPDATE および DELETE 操作の実行

PL/SQL のネストした表での DML 操作には、TABLE 演算子と CAST 演算子を使用します。この方法によって、ネストした表をデータベースに実際に格納せずに、SQL 表記法を使用してネストした表で集合演算を実行できます。

CAST のオペランドは、PL/SQL コレクション変数と SQL コレクション型（CREATE TYPE 文で作成）です。CAST によって、PL/SQL コレクションが SQL の型に変換されます。

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)) 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;
```

マルチレベル・コレクションの使用

スカラー型やオブジェクト型のコレクションの他に、コレクションを要素に持つコレクションも作成できます。たとえば、VARRAY のネストした表、VARRAY の VARRAY、ネストした表の VARRAY などを作成できます。

ネストした表のネストした表を SQL の列として作成する場合は、CREATE TABLE 文の構文をチェックして、記憶表の定義方法を確認します。

マルチレベル・コレクションの構文と可能性を示す例を次に示します。

マルチレベル VARRAY の例

```
declare
 type t1 is varray(10) of integer;
 type nt1 is varray(10) of t1; -- multilevel varray type
 va t1 := t1(2,3,5);
-- initialize multilevel varray
 nva nt1 := nt1(va, t1(55,6,73), t1(2,4), va);
 i integer;
```

```
 va1 t1;
begin
 -- multilevel access
 i := nva(2)(3); -- i will get value 73
 dbms_output.put_line(i);
 -- add a new varray element to nva
 nva.extend;

 nva(5) := t1(56, 32);
-- replace an inner varray element
 nva(4) := t1(45,43,67,43345);
-- replace an inner integer element
 nva(4)(4) := 1; -- replaces 43345 with 1
-- add a new element to the 4th varray element
-- and store integer 89 into it.
 nva(4).extend;
 nva(4)(5) := 89;
end;
/
```

マルチレベルのネストした表の例

```
declare
 type tb1 is table of varchar2(20);
 type ntb1 is table of tb1; -- table of table elements
 type tv1 is varray(10) of integer;
 type ntb2 is table of tv1; -- table of varray elements

 vtb1 tb1 := tb1('one', 'three');
 vntb1 ntb1 := ntb1(vtb1);
 vntb2 ntb2 := ntb2(tv1(3,5), tv1(5,7,3));  -- table of varray elements
begin
 vntb1.extend;
 vntb1(2) := vntb1(1);
 -- delete the first element in vntb1
 vntb1.delete(1);
 -- delete the first string from the second table in the nested table
 vntb1(2).delete(1);
end;
/
```

マルチレベルの結合配列の例

```
declare
 type tb1 is table of integer index by binary_integer;
 -- the following is index-by table of index-by tables
 type ntb1 is table of tb1 index by binary_integer;
 type va1 is varray(10) of varchar2(20);
 -- the following is index-by table of varray elements
 type ntb2 is table of va1 index by binary_integer;

 v1 va1 := va1('hello', 'world');
 v2 ntb1;
 v3 ntb2;
 v4 tb1;
 v5 tb1; -- empty table
begin
 v4(1) := 34;
 v4(2) := 46456;
 v4(456) := 343;
 v2(23) := v4;
 v3(34) := va1(33, 456, 656, 343);
-- assign an empty table to v2(35) and try again
   v2(35) := v5;
   v2(35)(2) := 78; -- it works now
end;
/
```

マルチレベル・コレクションおよびバルク SQL の例

```
create type t1 is varray(10) of integer;
/
create table tab1 (c1 t1);

insert into tab1 values (t1(2,3,5));
insert into tab1 values (t1(9345, 5634, 432453));

declare
 type t2 is table of t1;
 v2 t2;
begin
 select c1  BULK COLLECT INTO v2 from tab1;
 dbms_output.put_line(v2.count); -- prints 2
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、コレクションを使用しやすくしたり、アプリケーションを維持しやすくしたりするのに使用します。

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、ドット表記法を使用し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、SQL 文からはコールできません。また、EXTEND と TRIM は結合配列で使用できません。EXISTS、COUNT、LIMIT、FIRST、LAST、PRIOR および NEXT は、ファンクションです。EXTEND, TRIM および DELETE はプロシージャです。EXISTS、PRIOR、NEXT、TRIM、EXTEND および DELETE は、コレクションの添字に対応するパラメータを取ります。通常、この添字は整数ですが、結合配列の場合は文字列も使用できます。

基本構造的に NULL であるコレクションに適用されるのは EXISTS のみです。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。

コレクション要素の存在のチェック（EXISTS メソッド）

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。それ以外の場合、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用します。また、EXISTS を使用すると、存在しない要素を参照した場合に発生する例外を回避できます。次の例では、要素 i が存在する場合にのみ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。

コレクション内の要素数のカウント（COUNT メソッド）

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、VARRAY projects に 25 個の要素が含まれる場合、次の IF 条件は TRUE です。

```
IF projects.COUNT = 25 THEN ...
```

コレクションの現在のサイズは不明の場合があるため、そのような場合に COUNT が役立ちます。たとえば、Oracle データの列をフェッチ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かを考えます。この場合は、COUNT で答えを出すことができます。

COUNT は、整数式が使用できる位置ならどこでも使用できます。次の例では、COUNT を使用して、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。ただし、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。

要素を総計するときに、COUNT は削除された要素を無視します。

コレクションの最大サイズのチェック（LIMIT メソッド）

最大サイズがないネストした表や結合配列の場合、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要素の最大数を戻します（この最大数は、型定義で指定する必要があり、TRIM メソッドや EXTEND メソッドを使用して後で変更できます）。たとえば、VARRAY PROJECTS 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、整数式が使用できる位置ならどこでも使用できます。次の例では、LIMIT を使用して、VARRAY projects にさらに 15 の要素を追加できるかどうかを調べています。

```
IF (projects.COUNT + 15) < projects.LIMIT THEN ...
```

最初または最後のコレクション要素の検索（FIRST メソッドと LAST メソッド）

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。キー値が VARCHAR2 の結合配列の場合は、最小および最大のキー値が戻ります。NLS_COMP 初期化パラメータが ANSI に設定されていないかぎり、順序付けは文字列内の文字のバイナリ値に従います。ANSI の場合の順序付けは、NLS_SORT 初期化パラメータで指定したロケール固有のソート順に従います。

コレクションが空の場合、FIRST および 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 つのみの場合、FIRST と LAST は同じ索引値を戻します。

```
IF courses.FIRST = courses.LAST THEN ...  -- only one element
```

次の例に示すように、FIRST と LAST を使用して、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、整数式が使用できる位置ならどこでも使用できます。次の例では、FIRST を使用してループ・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 を戻し、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 を戻します。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戻します。また、ネストした表の場合、LAST は通常は COUNT と同じです。ただし、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。

要素を走査するときに、FIRST および LAST は削除された要素を無視します。

コレクションの各要素のループ（PRIOR メソッドと NEXT メソッド）

PRIOR(n) は、コレクションの索引 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。

キーが VARCHAR2 型の結合配列の場合は、これらのメソッドは適切なキー値を戻します。NLS_COMP 初期化パラメータが ANSI に設定されていないかぎり、順序付けは文字列内の文字のバイナリ値に従います。この場合、ANSI の場合の順序付けは、NLS_SORT 初期化パラメータで指定したロケール固有のソート順に従います。

これらのメソッドは、ループ中に、コレクションの要素が挿入または削除される可能性があるため、添字の値の固定セットを使用したループに比べて高い信頼性があります。特に結合配列の場合、添字は連続した順序ではないため、添字の順序が（1、2、4、8、16）や（'A'、'E'、'I'、'O'、'U'）となっている可能性があります。

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、次の文は要素 i をそれ自身に割り当てます。

```
projects(i) := projects.PRIOR(projects.NEXT(i));
```

PRIOR または NEXT を使用すると、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 を使用して、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;
```

要素間を横断するときに、PRIOR および NEXT は削除された要素を無視します。

コレクションのサイズの拡大（EXTEND メソッド）

ネストした表または VARRAY のサイズを大きくするには、EXTEND を使用します。索引付き表で EXTEND を使用することはできません。

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 を使用して、基本構造的に 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、削除された要素のプレースホルダを保持するため、必要に応じてそれらの要素を置き換えることができます。次の例を考えます。

```
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 を戻し、LAST は 4 を戻します。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。

コレクションのサイズの縮小（TRIM メソッド）

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、次の文では、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 が大きすぎる場合、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でください。ネストした表は、固定サイズの配列のように扱って DELETE のみを使用するか、またはスタックのように扱って TRIM と EXTEND のみを使用することをお薦めします。

PL/SQL は切り捨てられた（TRIM）要素のプレースホルダを保持しません。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のみではその要素を置き換えることができません。

コレクション要素の削除（DELETE メソッド）

このプロシージャには、様々な形式があります。

- 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、数値キーの結合配列またはネストした表から、*n* 番目の要素を削除します。結合配列のキーが文字列の場合は、そのキー値に対応する要素が削除されます。n が NULL の場合、DELETE(n) は何も実行しません。
- DELETE(m,n) は、結合配列またはネストした表から、m ～ n の範囲にある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 か 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

   nicknames.DELETE('Chip'); -- deletes element denoted by this key
   nicknames.DELETE('Buffy','Fluffy'); -- deletes elements with keys
                                       -- in this alphabetic range
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ため、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のみでその要素を置き換えることができます。

DELETE を使用すると、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。

コレクション・パラメータへのメソッドの適用

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導出されます。

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、ネストした表は基本構造的に 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。

コレクションに関する例外	呼び出される場合
COLLECTION_IS_NULL	基本構造的に NULL のコレクションに対して操作を試みた場合。
NO_DATA_FOUND	添字で、削除されている要素や結合配列の存在していない要素が指定された場合。
SUBSCRIPT_BEYOND_COUNT	添字がコレクションの中の要素数を超えている場合。
SUBSCRIPT_OUTSIDE_LIMIT	添字が有効範囲外である場合。
VALUE_ERROR	添字が NULL か、またはキーの型に変換できない場合。この例外は、キーが PLS_INTEGER の範囲として定義され、添字がこの範囲外の場合に発生する可能性があります。

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すことができます。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;    -- allowed; 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・コレクション型に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;
```

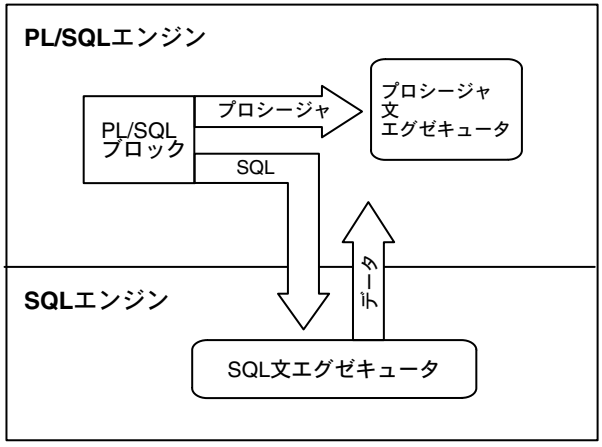
次に示す PL/SQL ブロックを実行すると、2 番目のプロシージャ・コールは「引数の数または型が正しくありません」というエラーで失敗します。これは、パッケージおよびローカルの VARRAY 型は定義が同一でも互換性がないためです。

```
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・バインドを使用したコレクションのループ・オーバーヘッドの削減

図 5-3 に示すように、PL/SQL エンジンはプロシージャ文を実行し、SQL 文を SQL エンジンに送信します。SQL エンジンは 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 5-3　コンテキスト切替え



PL/SQL と SQL エンジンの間でコンテキスト切替えを頻繁に行うと、パフォーマンスに悪影響を与える場合があります。この切替えが発生するのは、コレクションの各要素に対して個別の SQL 文をループで実行し、コレクション要素をバインド変数として指定した場合です。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 4 つ以上のデータベース行に影響する場合は、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

値を PL/SQL 変数に SQL 文で代入することを、**バインド**と呼びます。PL/SQL バインド操作は、3 つのカテゴリに分類されます。

- **インバインド** PL/SQL 変数またはホスト変数が、INSERT 文または UPDATE 文によってデータベースに格納される場合。
- **アウトバインド** データベースの値が、INSERT 文、UPDATE 文または DELETE 文の RETURNING 句によって PL/SQL 変数またはホスト変数に代入される場合。
- **定義** データベースの値が、SELECT 文または FETCH 文によって PL/SQL 変数またはホスト変数に代入される場合。

DML 文は単一の操作でコレクションのすべての要素を送信できます。この処理は、**バルク・バインド**と呼ばれます。コレクションに 20 個の要素がある場合、バルク・バインドは単一の操作で、20 回分の SELECT 文、INSERT 文、UPDATE 文または DELETE 文に相当する処理を実行できます。バルク・バインドの手法は、PL/SQL エンジンと SQL エンジンの間のコンテキスト切替えの回数を最小限に抑えることで、パフォーマンスを向上させます。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。

INSERT 文、UPDATE 文および DELETE 文でバルク・バインドを行うには、PL/SQL の FORALL 文に、SQL 文を記述します。

SELECT 文でバルク・バインドを行うには、SELECT 文で、INTO を使用するかわりに BULK COLLECT 句を指定します。

これらの文に関する構文と制限の詳細は、13-84 ページの「FORALL 文」および 13-163 ページの「SELECT INTO 文」を参照してください。

例 : DELETE を使用したバルク・バインドの実行

次の DELETE 文は DELETE 操作を 3 回実行しますが、SQL エンジンに対しては 1 回のみ送信されます。

```
DECLARE
   TYPE NumList IS VARRAY(20) OF NUMBER;
   depts NumList := NumList(10, 30, 70);  -- department numbers
BEGIN
   FORALL i IN depts.FIRST..depts.LAST
      DELETE FROM emp WHERE deptno = depts(i);
END;
```

例 : INSERT を使用したバルク・バインドの実行

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。すべての表要素がデータベース表に 2 度挿入されます。最初に FOR ループを使用し、次に FORALL 文を使用します。FORALL のほうが高速です。

```
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 NUMBER(5);
 7     t2 NUMBER(5);
 8     t3 NUMBER(5);
 9
10
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     t1 := dbms_utility.get_time;
17     FOR i IN 1..5000 LOOP  -- use FOR loop
18        INSERT INTO parts VALUES (pnums(i), pnames(i));
19     END LOOP;
20     t2 := dbms_utility.get_time;
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: 32
FORALL:   3

PL/SQL procedure successfully completed.
```

FORALL 文の使用

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に、PL/SQL エンジンに指示します。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 文中でのみ参照できます。SQL 文は、コレクション要素を参照する INSERT 文、UPDATE 文または DELETE 文であることが必要です。さらに、境界には連続した索引番号の有効範囲を指定する必要があります。SQL エンジンは、範囲内の各索引番号に対して 1 度ずつ SQL 文を実行します。

例 : コレクションの一部での FORALL の使用

次の例のように、FORALL ループの境界は、必ずしも全要素に対してではなく、コレクションの一部に適用できます。

```
DECLARE
   TYPE NumList IS VARRAY(10) OF NUMBER;
   depts NumList := NumList(20,30,50,55,57,60,70,75,90,92);
BEGIN
   FORALL j IN 4..7  -- bulk-bind only part of varray
      UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(j);
END;
```

例 : 添字付きコレクションが必要なバルク・バインド

SQL 文は複数のコレクションを参照できます。ただし、PL/SQL エンジンは添字付きコレクションのみをバルク・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), ...);
```

例 : FORALL を使用したオブジェクト表への挿入

リレーショナル表以外に、FORALL 文は次の例に示すように、オブジェクト表を操作できます。

```
CREATE TYPE PNum AS OBJECT (n NUMBER);
/
CREATE TABLE partno OF PNum;

DECLARE
   TYPE NumTab IS TABLE OF NUMBER;
   nums NumTab := NumTab(1, 2, 3, 4);
   TYPE PNumTab IS TABLE OF PNum;
   pnums PNumTab := PNumTab(PNum(1), PNum(2), PNum(3), PNum(4));
BEGIN
   FORALL i IN pnums.FIRST..pnums.LAST
      INSERT INTO partno VALUES(pnums(i));
   FORALL i IN nums.FIRST..nums.LAST
      DELETE FROM partno WHERE n = 2 * nums(i);
   FORALL i IN nums.FIRST..nums.LAST
      INSERT INTO partno VALUES(100 + nums(i));
END;
```

FORALL がロールバックに与える影響

FORALL 文では、SQL 文の実行によって未処理例外が発生した場合、前回の実行中に行われたすべてのデータベース変更はロールバックされます。しかし、呼び出された例外が捕捉され処理されると、変更は、各 SQL 文の実行の前にマークされた暗黙的な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、次の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, 30);
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 3 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、そして depts(30) 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 回目の実行のみがロールバックされます。

SQL 文の実行で例外が呼び出されると、FORALL 文が停止します。この例では、UPDATE 文を 2 回実行すると例外が発生するため、3 度目は実行できません。

%BULK_ROWCOUNT 属性を持つ FORALL の反復による影響をうける行カウント

SQL DML 文を処理するには、SQL エンジンは SQL という名前の暗黙カーソルをオープンします。このカーソルのスカラー属性、%FOUND、%ISOPEN、%NOTFOUND および %ROWCOUNT は直前に実行された SQL DML 文についての有用な情報を戻します。

SQL カーソルには、FORALL 文での使用に設計された複合属性 %BULK_ROWCOUNT のみがあります。この属性は索引付き表の意味を持っています。*i* 番目の要素には、INSERT 文、UPDATE 文または DELETE 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。次に例を示します。

```
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);
-- Did the 3rd UPDATE statement affect any rows?
   IF SQL%BULK_ROWCOUNT(3) = 0 THEN ...
END;
```

FORALL 文と %BULK_ROWCOUNT 属性は同じ添字を使用します。たとえば、FORALL が 5 ～ 10 の範囲を使用した場合は、%BULK_ROWCOUNT でも同じ範囲が使用されます。

典型的な挿入操作は 1 行にのみ影響するため、通常、挿入の場合の %BULK_ROWCOUNT は 1 です。ただし、INSERT ... SELECT 構造体の場合は、%BULK_ROWCOUNT が 2 以上になる場合があります。たとえば、次の FORALL 文は反復のたびに任意の数の行を挿入します。それぞれの反復後に、%BULK_ROWCOUNT は挿入された行数を戻します。

```
SET SERVEROUTPUT ON;
DECLARE
  TYPE num_tab IS TABLE OF NUMBER;
  deptnums num_tab;
BEGIN
  SELECT deptno BULK COLLECT INTO deptnums FROM DEPT;

  FORALL i IN 1..deptnums.COUNT
     INSERT INTO emp_by_dept
        SELECT empno, deptno FROM emp WHERE deptno =
deptnums(i);

  FOR i IN 1..deptnums.COUNT LOOP
-- Count how many rows were inserted for each department; that is,
-- how many employees are in each department.
     dbms_output.put_line('Dept '||deptnums(i)||': inserted '||
                          SQL%BULK_ROWCOUNT(i)||' records');
  END LOOP;

  dbms_output.put_line('Total records inserted =' || SQL%ROWCOUNT);
END;
/
```

バルク・バインドには、スカラー属性の %FOUND、%NOTFOUND および %ROWCOUNT も使用できます。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 文の最後の実行のみを参照します。ただし、%BULK_ROWCOUNT を使用すると、個々の実行に対する値を推論できます。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。

%BULK_EXCEPTIONS 属性を持つ FORALL 例外の処理

PL/SQL には、FORALL 文の実行中に呼び出される例外を処理するメカニズムが用意されています。このメカニズムにより、バルク・バインド操作では、例外に関する情報を保存して処理を継続できます。

エラーが発生した場合もバルク・バインドを完了させるには、FORALL 文にキーワード SAVE EXCEPTIONS を追加します。次に構文を示します。

```
FORALL index IN lower_bound..upper_bound SAVE EXCEPTIONS
   {insert_stmt | update_stmt | delete_stmt}
```

実行中に呼び出されたすべての例外は、レコードのコレクションを格納する新規のカーソル属性 %BULK_EXCEPTIONS に保存されます。各レコードには 2 つのフィールドがあります。一方のフィールド %BULK_EXCEPTIONS(i).ERROR_INDEX には、例外が呼び出されたとき

に実行中だった FORALL 文の「反復」が保持されます。他方のフィールド %BULK_EXCEPTIONS(i).ERROR_CODE には、対応する Oracle エラー・コードが保持されます。

%BULK_EXCEPTIONS により格納された値は常に、直前に実行された FORALL 文を参照します。例外の数は、%BULK_EXCEPTIONS のカウント属性、つまり %BULK_EXCEPTIONS.COUNT に保存されます。添字の範囲は 1 ～ COUNT です。

キーワード SAVE EXCEPTIONS を省略すると、例外が呼び出された時点で FORALL 文の実行が停止します。その場合、SQL%BULK_EXCEPTIONS.COUNT は 1 を戻し、SQL%BULK_EXCEPTIONS にはレコードが 1 つのみ含まれます。実行中に例外が呼び出されなければ、SQL%BULK_EXCEPTIONS.COUNT は 0 を戻します。

カーソル属性 %BULK_EXCEPTIONS の用途を次の例に示します。

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   num_tab NumList := NumList(10,0,11,12,30,0,20,199,2,0,9,1);
   errors  NUMBER;
   dml_errors EXCEPTION;
   PRAGMA exception_init(dml_errors, -24381);
BEGIN
   FORALL i IN num_tab.FIRST..num_tab.LAST SAVE EXCEPTIONS
      DELETE FROM emp WHERE sal > 500000/num_tab(i);
EXCEPTION
  WHEN dml_errors THEN
   errors := SQL%BULK_EXCEPTIONS.COUNT;
   dbms_output.put_line('Number of errors is ' || errors);
   FOR i IN 1..errors LOOP
      dbms_output.put_line('Error ' || i || ' occurred during '||
         'iteration ' || SQL%BULK_EXCEPTIONS(i).ERROR_INDEX);
      dbms_output.put_line('Oracle error is ' ||
         SQLERRM(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
   END LOOP;
END;
```

この例では、i が 2、6、10 のときに、事前定義の例外 ZERO_DIVIDE が呼び出されています。バルク・バインドが完了すると、SQL%BULK_EXCEPTIONS.COUNT は 3 を戻し、SQL%BULK_EXCEPTIONS の内容は (2,1476)、(6,1476) および (10,1476) となっています。Oracle エラー・メッセージを（コードを含めて）取得するために、SQL%BULK_EXCEPTIONS(i).ERROR_CODE の値を無効にし、その結果をエラー・レポート・ファンクション SQLERRM に渡しています。このファンクションでは負の数値が予測されています。次に出力を示します。

```
Number of errors is 3
Error 1 occurred during iteration 2
Oracle error is ORA-01476: 除数がゼロです。
Error 2 occurred during iteration 6
```

```
Oracle error is ORA-01476: 除数がゼロです。
Error 3 occurred during iteration 10
Oracle error is ORA-01476: 除数がゼロです。
```

BULK COLLECT 句を使用した、問合せ結果のコレクションへの取出し

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に、SQL エンジンに指示します。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、スカラー値またはオブジェクトなどの複合値が格納できます。次の例では、SQL エンジンは、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;
```

次の例では、SQL エンジンはネストした表を PL/SQL エンジンに戻す前に、オブジェクト列のすべての値をネストした表にロードします。

```
CREATE TYPE Coords AS OBJECT (x NUMBER, y NUMBER);
CREATE TABLE grid (num NUMBER, loc Coords);
INSERT INTO grid VALUES(10, Coords(1,2));
INSERT INTO grid VALUES(20, Coords(3,4));

DECLARE
   TYPE CoordsTab IS TABLE OF Coords;
   pairs CoordsTab;
BEGIN
   SELECT loc BULK COLLECT INTO pairs FROM grid;
   -- now pairs contains (1,2) and (3,4)
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 を使用して、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 SalList IS TABLE OF emp.sal%TYPE;
   sals SalList;
BEGIN
   SELECT sal BULK COLLECT INTO sals FROM emp
      WHERE ROWNUM <= 100;
   ...
END;
```

カーソルからのバルク・フェッチの例

1 つ以上のコレクションへのバルク・フェッチ

1 つのカーソルから 1 つ以上のコレクションにバルク・フェッチできます。

```
DECLARE
   TYPE NameList IS TABLE OF emp.ename%TYPE;
   TYPE SalList IS TABLE OF emp.sal%TYPE;
   CURSOR c1 IS SELECT ename, sal FROM emp WHERE sal > 1000;
   names NameList;
   sals  SalList;
BEGIN
   OPEN c1;
   FETCH c1 BULK COLLECT INTO names, sals;
END;
```

レコードのコレクションへのバルク・フェッチ

1 つのカーソルから複数レコードのコレクションにバルク・フェッチできます。

```
DECLARE
   TYPE DeptRecTab IS TABLE OF dept%ROWTYPE;
   dept_recs DeptRecTab;
   CURSOR c1 IS
      SELECT deptno, dname, loc FROM dept WHERE deptno > 10;
BEGIN
   OPEN c1;
```

```
   FETCH c1 BULK COLLECT INTO dept_recs;
END;
```

LIMIT 句を使用したバルク・フェッチ操作の対象行の制限

バルク（スカラーではない）FETCH 文の中でのみ許されるオプションの LIMIT 句で、データベースからフェッチされる行の数を制限します。次に構文を示します。

```
FETCH ... BULK COLLECT INTO ... [LIMIT rows];
```

行にはリテラル、変数または式を使用できますが、必ず整数に評価されるものであることが必要です。それ以外の場合、PL/SQL は事前定義の例外 VALUE_ERROR を呼び出します。正の整数値ではない場合は、PL/SQL は INVALID_NUMBER を呼び出します。必要に応じて、PL/SQL は数値を最も近い整数に四捨五入します。

次の例では、ループが繰り返されるたびに、FETCH 文によって 10（またはそれ以下の）行が索引付き表 empnos にフェッチされます。前の値は上書きされます。

```
DECLARE
   TYPE NumTab IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
   CURSOR c1 IS SELECT empno FROM emp;
   empnos NumTab;
   rows   NATURAL := 10;
BEGIN
   OPEN c1;
   LOOP
      /* The following statement fetches 10 rows (or less). */
      FETCH c1 BULK COLLECT INTO empnos LIMIT rows;
      EXIT WHEN c1%NOTFOUND;
      ...
   END LOOP;
   CLOSE c1;
END;
```

RETURNING INTO 句を使用したコレクションへの DML 結果の取出し

次の例に示すように、INSERT、UPDATE または DELETE 文の RETURNING INTO 句に BULK COLLECT 句を使用できます。

```
DECLARE
   TYPE NumList IS TABLE OF emp.empno%TYPE;
   enums NumList;
BEGIN
   DELETE FROM emp WHERE deptno = 20
      RETURNING empno BULK COLLECT INTO enums;
      -- if there were five employees in department 20,
      -- then enums contains five employee numbers
END;
```

BULK COLLECT の制限

BULK COLLECT 句には、次の制限が適用されます。

- キーが文字列型の結合配列に対しては BULK COLLECT を使用できません。
- BULK COLLECT 句はサーバー側（クライアント側ではなく）のプログラムの中でしか使用できません。クライアント側で使用すると、「この機能はクライアント側のプログラムではサポートされていません。」というエラーが表示されます。
- BULK COLLECT INTO 句のすべてのターゲットは次の例に示すようにコレクションである必要があります。

  ```
  DECLARE
     TYPE NameList IS TABLE OF emp.ename%TYPE;
     names  NameList;
     salary emp.sal%TYPE;
  BEGIN
     SELECT ename, sal BULK COLLECT INTO names, salary  -- illegal target
        FROM emp WHERE ROWNUM < 50;
     ...
  END;
  ```

- コンポジット・ターゲット（オブジェクトなど）を RETURNING INTO 句で使用することはできません。使用すると、「*RETURNING* 句ではサポートされていない機能です。」というエラーが発生します。
- 暗黙的なデータ型変換が必要な場合、複数のコンポジット・ターゲットを BULK COLLECT INTO 句で使用することはできません。
- 暗黙的なデータ型変換が必要な場合、コンポジット・ターゲットのコレクション（オブジェクトのコレクションなど）を BULK COLLECT INTO 句で使用することはできません。

FORALL と BULK COLLECT の併用

BULK COLLECT 句を FORALL 文と組み合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。)

FORALL 文では、SELECT ...BULK COLLECT 文は使用できません。使用すると、「実装上の制約 *SELECT* 文では *FORALL* と *BULK COLLECT INTO* を一緒に使用できません。」というエラーが表示されます。

バルク・バインドとホスト配列の併用

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付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用されています。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;
```

レコード

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、従業員に関する様々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。レコードを使用すると、データを 1 つの論理単位として扱うことができます。レコードを使用すれば、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。ただし、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することはできません。データ型の RECORD を使用すると、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

レコードを作成するには、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、次の構文を使用して、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]...);
```

field_declaration は次のことを意味します。

```
field_name field_type [[NOT NULL] {:= | DEFAULT} expression]
```

type_name はレコードを宣言するために後で使用される型指定子、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 : VARRAY や（NESTED）TABLE 型とは異なり、RECORD 型はデータベース内で CREATE を使用して作成したり格納したりできません。

%TYPE および %ROWTYPE を使用し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(14),
      dept_loc  VARCHAR2(13));
BEGIN
   ...
END;
```

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）を含むレコードを定義できます。ただし、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));
BEGIN
   ...
END;
```

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    NUMBER(4)
      last_name VARCHAR2(10),
      dept_num  NUMBER(2),
      job_title VARCHAR2(9),
      salary    NUMBER(7,2));
   ...
   FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRec IS ...
BEGIN
   ...
END;
```

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。識別子 item_info はレコード全体を表します。

```
DECLARE
   TYPE StockItem IS RECORD (
      item_no     INTEGER(3),
      description VARCHAR2(50),
      quantity    INTEGER,
      price       REAL(7,2));
   item_info StockItem;  -- declare record
BEGIN
   ...
END;
```

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。次に例を示します。

```
DECLARE
   TYPE EmpRec IS RECORD (
      emp_id    emp.empno%TYPE,
      last_name VARCHAR2(10),
      job_title VARCHAR2(9),
      salary    NUMBER(7,2));
   ...
   PROCEDURE raise_salary (emp_info EmpRec);
BEGIN
   ...
END;
```

レコードの初期化

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);
BEGIN
   ...
END;
```

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で、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、初期化されている必要があります。

```
DECLARE
   TYPE StockItem IS RECORD (
      item_no     INTEGER(3) NOT NULL := 999,
      description VARCHAR2(50),
      quantity    INTEGER,
      price       REAL(7,2));
BEGIN
   ...
END;
```

レコードの参照

添字を使用し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用します。

```
record_name.field_name
```

たとえば、レコード emp_info の中のフィールド 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、次の構文を使用し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 の中のフィールド salary を参照しています。

```
DECLARE
   TYPE EmpRec IS RECORD (
      emp_id    NUMBER(4),
      job_title VARCHAR2(9),
      salary    NUMBER(7,2));
   middle_sal NUMBER(7,2);
   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
   ...
END;
```

パラメータなしでファンクションをコールする場合、次の構文を使用します。

```
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用します。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 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用します。

```
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;
```

NULL 値のレコードへの代入

レコード内のすべてのフィールドを NULL に設定するには、次の例のように、同じ型の初期化されていないレコードを代入します。

```
DECLARE
   TYPE EmpRec IS RECORD (
      emp_id    emp.empno%TYPE,
      job_title VARCHAR2(9),
      salary    NUMBER(7,2));
   emp_info EmpRec;
   emp_null EmpRec;
BEGIN
   emp_info.emp_id := 7788;
   emp_info.job_title := 'ANALYST';
   emp_info.salary := 3500;
   emp_info := emp_null;  -- nulls all fields in emp_info
   ...
END;
```

レコードの代入

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、次の構文を使用します。

```
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のみでは不十分です。次の例を考えます。

```
DECLARE
   TYPE DeptRec IS RECORD (
      dept_num  NUMBER(2),
      dept_name VARCHAR2(14));
   TYPE DeptItem IS RECORD (
      dept_num  NUMBER(2),
      dept_name VARCHAR2(14));
   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 VARCHAR2(14),
      location  VARCHAR2(13));
   dept1_info DeptRec;
   dept2_info dept%ROWTYPE;
BEGIN
   SELECT * INTO dept2_info FROM dept WHERE deptno = 10;
   dept1_info := dept2_info;
   ...
END;
```

第 2 の方法として、次の例のように SELECT 文または FETCH 文を使用し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。

```
DECLARE
   TYPE DeptRec IS RECORD (
      dept_num  NUMBER(2),
      dept_name VARCHAR2(14),
      location  VARCHAR2(13));
   dept_info DeptRec;
BEGIN
   SELECT * INTO dept_info FROM dept WHERE deptno = 20;
   ...
END;
```

ただし、代入文を使用してレコードに値のリストを代入することはできません。次の構文は誤りです。

```
record_name := (value1, value2, value3, ...);  -- not allowed
```

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できます。収集された情報は、まとまった 1 つの集合として参照できるため取扱いが簡単です。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 PassengerList を定義します。

```
SQL> CREATE TYPE PassengerList AS VARRAY(300) OF Passenger;
```

最後に、型 PassengerList の列を含むリレーショナル表 flights を次のようにして作成します。

```
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、指定された飛行（flights）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 から行をフェッチして、レコード flight_info に入れます。このように、飛行（flights）予定に関するすべての情報を、乗客リスト（PassengerList）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;
   seat_not_available EXCEPTION;
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;
```

PL/SQL レコードのデータベースへの挿入

INSERT 文の PL/SQL のみの拡張機能によって、フィールドのリストではなく、RECORD 型または %ROWTYPE 型の単一の変数を使用して、レコードをデータベース行に挿入できます。その結果、コードが読みやすくなり、メンテナンスが容易になります。

レコード内のフィールドの数は、INTO 句にリストされている列数と等しい必要があります。また、対応するフィールドと列のデータ型には互換性が必要です。レコードと表との互換性を確実に保持するには、変数を *table_name*%ROWTYPE 型として宣言することが最も便利です。

%ROWTYPE を使用した PL/SQL レコードの挿入 : 例

この例では、%ROWTYPE 修飾子を使用してレコード変数を宣言しています。この変数は、列リストを指定せずに挿入できます。%ROWTYPE 宣言によって、表の列と同じ名前と型をレコードの属性に設定できます。

```
DECLARE
   dept_info dept%ROWTYPE;
BEGIN
-- deptno, dname, and loc are the table columns.
-- The record picks up these names from the %ROWTYPE.
  dept_info.deptno := 70;
```

```
  dept_info.dname := 'PERSONNEL';
  dept_info.loc := 'DALLAS';
-- Using the %ROWTYPE means we can leave out the column list
-- (deptno, dname, loc) from the INSERT statement.
   INSERT INTO dept VALUES dept_info;
END;
```

PL/SQL レコード値を使用したデータベースの更新

UPDATE 文の PL/SQL のみの拡張機能によって、フィールドのリストではなく、RECORD 型または %ROWTYPE 型の単一の変数を使用して、データベース行を更新できます。

レコード内のフィールドの数は、SET 句にリストされている列数と等しい必要があります。また、対応するフィールドと列のデータ型には互換性が必要です。

レコードを使用した行の更新 : 例

キーワード ROW を使用すると、行全体を表現できます。

```
DECLARE
   dept_info dept%ROWTYPE;
BEGIN
   dept_info.deptno := 30;
   dept_info.dname := 'MARKETING';
   dept_info.loc := 'ATLANTA';
-- The row will have values for the filled-in columns, and null
-- for any other columns.
   UPDATE dept SET ROW = dept_info WHERE deptno = 30;
END;
```

キーワード ROW を指定できる位置は、SET 句の左側のみです。

副問合せと一緒に使用できない SET ROW: 例

ROW と副問合せは一緒に使用できません。たとえば、次の UPDATE 文は誤りです。

```
UPDATE emp SET ROW = (SELECT * FROM mgrs);  -- not allowed
```

オブジェクトを含むレコードを使用した行の更新 : 例

オブジェクト型が含まれるレコードを使用できます。

```
CREATE TYPE Worker AS OBJECT (name VARCHAR2(25), dept VARCHAR2(15));
/
CREATE TABLE teams (team_no NUMBER, team_member Worker);

DECLARE
```

```
   team_rec teams%ROWTYPE;
BEGIN
   team_rec.team_no := 5;
   team_rec.team_member := Worker('Paul Ocker', 'Accounting');
   UPDATE teams SET ROW = team_rec;
END;
/
```

コレクションを含むレコードを使用した行の更新 : 例

レコードには、コレクションも含めることができます。

```
CREATE TYPE Worker AS OBJECT (name VARCHAR2(25), dept VARCHAR2(15));
/
CREATE TYPE Roster AS TABLE OF Worker;
/
CREATE TABLE teams (team_no NUMBER, members Roster)
   NESTED TABLE members STORE AS teams_store;

INSERT INTO teams VALUES (1, Roster(
   Worker('Paul Ocker', 'Accounting'),
   Worker('Gail Chan', 'Sales')
   Worker('Marie Bello', 'Operations')
   Worker('Alan Conwright', 'Research')));

DECLARE
   team_rec teams%ROWTYPE;
BEGIN
   team_rec.team_no := 3;
   team_rec.members := Roster(
      Worker('William Bliss', 'Sales'),
      Worker('Ana Lopez', 'Sales')
      Worker('Bridget Towner', 'Operations')
      Worker('Ajay Singh', 'Accounting'));
   UPDATE teams SET ROW = team_rec;
END;
/
```

レコードを使用した RETURNING 句の使用 : 例

INSERT 文、UPDATE 文および DELETE 文には、RETURNING 句を含めることができます。この句は、影響のある行の列値を PL/SQL レコード変数に戻します。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。この句が使用できるのは、厳密に 1 つの行で操作する場合のみです。

次の例では、従業員の給与の更新と同時に、従業員の名前、肩書きおよび新しい給与をレコード変数に取り出しています。

```
DECLARE
   TYPE EmpRec IS RECORD (
      emp_name  VARCHAR2(10),
      job_title VARCHAR2(9),
      salary    NUMBER(7,2));
   emp_info EmpRec;
   emp_id NUMBER(4);
BEGIN
   emp_id := 7782;
   UPDATE emp SET sal = sal * 1.1
      WHERE empno = emp_id
      RETURNING ename, job, sal INTO emp_info;
END;
```

レコードの挿入 / 更新に関する制約

現在、レコードの挿入 / 更新には、次の制約があります。

- レコード変数が使用できるのは、次の位置に限定されます。
  - UPDATE 文の SET 句の右側
  - INSERT 文の VALUES 句の中
  - RETURNING 句の INTO 副次句の中

  レコード変数は、SELECT リスト、WHERE 句、GROUP BY 句または ORDER BY 句では使用できません。
- キーワード ROW を指定できる位置は、SET 句の左側のみです。また、ROW と副問合せは一緒に使用できません。
- UPDATE 文では、ROW が使用されている場合、許可される SET 句は 1 つのみです。
- INSERT 文の VALUES 句にレコード変数が含まれている場合は、その句の中で他の変数または値を使用することはできません。
- RETURNING 句の INTO 副次句にレコード変数が含まれている場合は、その副次句の中で他の変数または値を使用することはできません。

- 次の内容はサポートされません。
  - ネストしたレコード型
  - レコードを戻すファンクション
  - EXECUTE IMMEDIATE 文を使用したレコードの挿入および更新

レコードのコレクションへのデータの問合せ

PL/SQL バインド操作は、3 つのカテゴリに分類されます。

- **定義** SELECT 文または FETCH 文によって、PL/SQL 変数またはホスト変数に取り出されたデータベース値を意味します。
- **インバインド** INSERT 文によって挿入、または UPDATE 文によって変更されたデータベース値を意味します。
- **アウトバインド** INSERT 文、UPDATE 文または DELETE 文の RETURNING 句によって、PL/SQL 変数またはホスト変数に戻されたデータベース値を意味します。

PL/SQL では、DML 文にあるレコードのコレクションに関するバルク・バインドをサポートしています。特に、定義やアウトバインドの変数はレコードのコレクションにでき、インバインドの値はレコードのコレクションに格納できます。次に構文を示します。

```
SELECT select_items BULK COLLECT INTO record_variable_name
   FROM rest_of_select_stmt

FETCH {  cursor_name
       | cursor_variable_name
       | :host_cursor_variable_name}
   BULK COLLECT INTO record_variable_name
   [LIMIT numeric_expression];

FORALL index IN lower_bound..upper_bound
   INSERT INTO {  table_reference
                | THE_subquery} [{column_name[, column_name]...}]
   VALUES (record_variable_name(index)) rest_of_insert_stmt

FORALL index IN lower_bound..upper_bound
   UPDATE {table_reference | THE_subquery} [alias]
   SET (column_name[, column_name]...) = record_variable_name(index)
   rest_of_update_stmt

RETURNING row_expression[, row_expression]...
   BULK COLLECT INTO record_variable_name;
```

前述の各文と句では、レコード変数にレコードのコレクションが格納されます。レコード内のフィールドの数は、SELECT リスト内の項目数、INSERT INTO 句内の列数、

UPDATE ... SET 句内の列数または RETURNING 句にある行の式数と一致している必要があります。対応するフィールドと列のデータ型には互換性が必要です。次に例を示します。

```
CREATE TABLE tab1 (col1 NUMBER, col2 VARCHAR2(20));
/
CREATE TABLE tab2 (col1 NUMBER, col2 VARCHAR2(20));
/
DECLARE
   TYPE RecTabTyp IS TABLE OF tab1%ROWTYPE
      INDEX BY BINARY_INTEGER;
   TYPE NumTabTyp IS TABLE OF NUMBER
      INDEX BY BINARY_INTEGER;
   TYPE CharTabTyp IS TABLE OF VARCHAR2(20)
      INDEX BY BINARY_INTEGER;
   CURSOR c1 IS SELECT col1, col2 FROM tab2;
   rec_tab  RecTabTyp;
   num_tab  NumTabTyp := NumTabTyp(2,5,8,9);
   char_tab CharTabTyp := CharTabTyp('Tim', 'Jon', 'Beth', 'Jenny');
BEGIN
   FORALL i IN 1..4
      INSERT INTO tab1 VALUES(num_tab(i), char_tab(i));

   SELECT col1, col2 BULK COLLECT INTO rec_tab FROM tab1
      WHERE col1 < 9;

   FORALL i IN rec_tab.FIRST..rec_tab.LAST
      INSERT INTO tab2 VALUES rec_tab(i);

   FOR i IN rec_tab.FIRST..rec_tab.LAST LOOP
      rec_tab(i).col1 := rec_tab(i).col1 + 100;
   END LOOP;

   FORALL i IN rec_tab.FIRST..rec_tab.LAST
      UPDATE tab1 SET (col1, col2) = rec_tab(i) WHERE col1 < 8;

   OPEN c1;
   FETCH c1 BULK COLLECT INTO rec_tab;
   CLOSE c1;
END;
```

6

PL/SQL と Oracle の相互作用

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。

この章の項目は、次のとおりです。

PL/SQL における SQL サポートの概要

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、疑似列および演算子を完全にサポートしているため、Oracle データを柔軟かつ安全に操作できます。また、動的 SQL もサポートしているため、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。さらに、PL/SQL は、現行の ANSI/ISO SQL 規格に準拠しています。

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用し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 DML 文です。たとえば、2 つの UPDATE 文を使用して、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リストアされます。

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、カレント・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、カレント・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のみを取り消すことができます。SET TRANSACTION は、読取り / 書込み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用できます。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 および 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用できません。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かぎり、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、次の構文を使用します。

```
function_name([ALL | DISTINCT] expression)
```

expression は、1 つ以上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のみを取り扱います。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を戻します。

```
SELECT COUNT(DISTINCT job) INTO job_count FROM emp;
```

ファンクション COUNT では、表の行数を戻すアスタリスク（*）行演算子を使用できます。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 では、特定のデータ項目を戻す SQL 疑似例 CURRVAL、LEVEL、NEXTVAL、ROWID および ROWNUM を認識します。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。ただし、疑似列に値を挿入したり、疑似列の値を更新または削除することはできません。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用できません。

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。順序を作成する場合は、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 を使用し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用します。

```
sequence_name.CURRVAL
sequence_name.NEXTVAL
```

順序を作成すると、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、SELECT リスト、VALUES 句および SET 句の中でしか使用できません。次の例では、順序を使用し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 を使用すると、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。

ツリーのルートを識別する条件は、START WITH 句で指定します。PRIOR 演算子を使用して、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。

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 型の変数を使用すると、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;
```

物理 ROWID を選択またはフェッチして 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ファンクション ROWIDTOCHAR を使用します。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、カーソルからフェッチされた最新の行を識別できます。例は、6-50 ページの「コミットにまたがるフェッチ」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 を使用して、表の中の個々の行に一意の値を代入できます。また、SELECT 文の WHERE 句で ROWNUM を使用して、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のみです。つまり、WHERE 句で ROWNUM を使用する場合は、次のようにする必要があります。

```
... WHERE ROWNUM < constant;
... 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用します。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、『Oracle9*i* 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、DML 文の WHERE 句で述語を形成するために使用します。述語は、2 つの式を比較して、常に 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 に評価します。
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 に評価します。
BETWEEN	値が指定範囲内かどうかをテストします。
EXISTS	副問合せが行を 1 つでも戻すと TRUE を戻します。
IN	セット・メンバーシップをテストします。
IS NULL	NULL かどうかをテストします。
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します。指定パターンにはワイルドカードを使用できます。

集合演算子

集合演算子は 2 つの問合せの結果を組み合せて 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 では、暗黙カーソルと明示カーソルの 2 種類があります。PL/SQL では、1 つの行のみ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。ただし、複数の行を戻す問合せの場合は、カーソル FOR ループを使用するか、BULK COLLECT 句を使用して、明示カーソルを宣言する必要があります。

明示カーソルの概要

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、0 行、1 行または複数行で構成されます。問合せが複数行を戻す場合は、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、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の中でカーソルを宣言できます。

カーソルの制御には、OPEN、FETCH および CLOSE の 3 つのコマンドを使用します。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 を繰り返し実行して、すべての行を取り出します。または BULK COLLECT 句を使用して、すべての行を一度にフェッチします。最後の行の処理が終わってから、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、事前に宣言する必要があります。カーソルを宣言する場合は、次の構文を使用し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;
```

return_type はデータベースの表の中のレコードまたは行を表す必要があります。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用することはできません。ただし、カーソルと変数は、同じ有効範囲規則に従います。データベース表に基づいてカーソルを命名できますが、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、定数が使用可能な位置であればどこででも使用できます。カーソルの仮パラメータは IN パラメータにしてください。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。初期化すると、必要に応じてデフォルト値を受け入れたり上書きすることで、カーソルの実パラメータに様々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用できます。

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 句を使用し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。行のフェッチには FETCH 文を使用します。

カーソル・パラメータを渡す方法

OPEN 文を使用し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なければ、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つ必要があります。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 を使用すると、この識別子は仮パラメータを参照します。ただし、OPEN 文の中で使用すると、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用してください。

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用して、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。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る必要があります。

カーソルを使用したフェッチ

BULK COLLECT 句（次の項で説明）を使用していない場合は、FETCH 文によって結果セットの行が一度に 1 行づつ取り出されます。各 FETCH 文は現在の行を取り出してから、カーソルを結果セットの次の行に進めます。次に例を示します。

```
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ている必要があります。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のみ評価されます。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、再オープンする必要があります。

ただし、同じカーソルを使用して、別々の FETCH 文で、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、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 を使用する必要があります。詳細は、6-33 ページの「カーソル属性の使用」を参照してください。

カーソルを使用したバルクデータのフェッチ

BULK COLLECT 句を使用すると、Oracle データの列全体をバルク・バインドできます（5-46 ページの「BULK COLLECT 句を使用した、問合せ結果のコレクションへの取出し」を参照）。そのようにすると、結果セットからすべての行を一度に取り出せます。次の例では、1 つのカーソルから 2 つのコレクションにバルク・フェッチを行います。

```
DECLARE
   TYPE NumTab IS TABLE OF emp.empno%TYPE;
   TYPE NameTab IS TABLE OF emp.ename%TYPE;
   nums  NumTab;
   names NameTab;
   CURSOR c1 IS SELECT empno, ename FROM emp WHERE job = 'CLERK';
BEGIN
   OPEN c1;
   FETCH c1 BULK COLLECT INTO nums, names;
   ...
   CLOSE c1;
END;
```

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。クローズされたカーソルは、再オープンできます。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。

カーソルでの副問合せの使用

副問合せは、別の SQL DML 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用されます。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用した次の問合せは、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、相関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。相関副問合せでは、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;
```

暗黙カーソルの概要

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。直前の暗黙カーソルを SQL カーソルとして参照できます。OPEN、FETCH および CLOSE 文を使用して SQL カーソルを制御することはできませんが、カーソル属性を使用して直前に実行された SQL 文に関する情報を入手できます。6-33 ページの「カーソル属性の使用」を参照してください。

パッケージでのカーソル仕様部と本体の分離

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配置できます。これにより、カーソルの仕様部を変更せずに、本体のみを変更できます。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、この構文を使用します。

```
CURSOR cursor_name [(parameter[, parameter]...)] RETURN return_type;
```

次に示す例では、%ROWTYPE 属性を使用して、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_stuff AS
   CURSOR c1 RETURN emp%ROWTYPE;  -- declare cursor spec
   ...
END emp_stuff;

CREATE PACKAGE BODY emp_stuff AS
   CURSOR c1 RETURN emp%ROWTYPE IS
      SELECT * FROM emp WHERE sal > 2500;  -- define cursor body
   ...
END emp_stuff;
```

戻り値のデータ型を RETURN 句で指定しているため、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 句が必要です。また、SELECT リスト中の項目の数とデータ型は、RETURN 句と一致する必要があります。

パッケージ・カーソルを使用すると柔軟性が向上します。たとえば、次のようにすると、前述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_stuff AS
   CURSOR c1 RETURN emp%ROWTYPE IS
      SELECT * FROM emp WHERE deptno = 20;  -- new WHERE clause
   ...
END emp_stuff;
```

次の例に示すように、PL/SQL ブロックまたはサブプログラムからパッケージ・カーソルを参照するドット表記法を使用します。

```
DECLARE
   emp_rec emp%ROWTYPE;
   ...
BEGIN
   ...
   OPEN emp_stuff.c1;
   LOOP
      FETCH emp_stuff.c1 INTO emp_rec;
      EXIT WHEN emp_suff.c1%NOTFOUND;
      ...
```

```
   END LOOP;
   CLOSE emp_stuff.c1;
END;
```

パッケージ・カーソルの有効範囲は特定の PL/SQL ブロックに制限されません。したがって、パッケージ・カーソルをオープンすると、クローズするか、Oracle セッションを切断するまでオープンしたままになります。

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 ループを使用して、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一連の作業を繰り返し、すべての行を処理した後にカーソルをクローズします。

次に示す PL/SQL ブロックでは、実験データを使用して結果を計算し、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用して参照します。

```
-- 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すことはできません。FOR ループを反復する前に、PL/SQL は暗黙的に宣言したレコードに取り出した値を格納します。レコードはループの内側のみ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。

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 文を使用し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。

明示カーソルに対する副問合せの代用

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;
```

カーソル副問合せの使用

カーソル副問合せはカーソル式とも呼ばれ、ファンクションにパラメータとして行セットを渡すことができます。たとえば、次の文では、StockPivot ファンクションにパラメータを渡しており、このファンクションはカーソル副問合せから戻される行を表す REF CURSOR で構成されています。

```
SELECT * FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable)));
```

カーソル副問合せは、通常は表関数と併用されます。8-30 ページの「表関数を使用した複数行の受入れと戻し」を参照してください。

カーソル FOR ループ内の式の別名の定義

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、SELECT リストの中の対応する列と同じ名前を持ちます。ただし、選択項目が式の場合はどうなるかを考える必要があります。次の例を考えます。

```
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用します。

```
IF emp_rec.wages < 1000 THEN ...
```

カーソル FOR ループへのパラメータの受渡し

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 クライアントで使用します。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用できます。

Oracle サーバーも 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）を通じてカーソル変数をやりとりできます。

カーソル変数

カーソル変数は、C や Pascal のポインタに類似しており、項目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成されます。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、作業域の名前を示す明示カーソルを使用します。または、作業域を指すカーソル変数を使用します。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、他方を使用することはできません。

変数を使用する理由

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用します。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おらず、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のみです。たとえば、OCI クライアント、Oracle Forms アプリケーションおよび Oracle サーバー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 ブロックを使用して複数のホスト・カーソル変数を 1 回の往復でオープンまたはクローズすることで、ネットワークの通信量を削減できます。

REF CURSOR 型の定義

カーソル変数を作成する場合は、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、サブプログラムまたはパッケージの中で、次の構文を使用して定義できます。

```
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用する型指定子です。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ることが必要です。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 型には、強い（限定的）ものと弱い（限定的ではない）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。これは、PL/SQL コンパイラの場合、強い型指定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ためです。弱い REF CURSOR 型は、より柔軟です。弱い型指定のカーソル変数は、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成さ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化の規則に従います。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成され、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 を使用して、強い型指定（弱い型指定ではなく）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 を使用し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に、カーソル変数にはパラメータのエイリアシングの可能性があります。8-21 ページの「サブプログラムのパラメータのエイリアシングの理解」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 文、FETCH 文および 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行を取り出します。すべての行が処理された後に、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 文を使用すると、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 | :host_cursor_variable} FOR
{   select_statement
 | dynamic_string [USING bind_argument[, bind_argument]...] };
```

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列式です。

注意 : この項では静的 SQL の場合について説明します。ここでは select_statement を使用します。動的 SQL の場合は dynamic_string が使用されます。11-7 ページの「カーソル変数のオープン」を参照してください。

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のみでなく問合せ全体を渡すことができるため、柔軟性はあります。問合せでは、ホスト変数、PL/SQL 変数、パラメータおよびファンクションを参照できます。

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。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 モードを指定する必要があります。この指定によって、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・プロシージャを使用してカーソル変数をオープンする方法もあります。単に別個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成する場合は、スタンドアロン・プロシージャを作成し、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・プロシージャを作成しています。

```
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。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 EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp, choice INT);
END emp_data;

CREATE PACKAGE BODY emp_data AS
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp, choice INT) 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;
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。次に例を示します。

```
CREATE PACKAGE admin_data AS
   TYPE GenCurTyp IS REF CURSOR;
   PROCEDURE open_cv (generic_cv IN OUT GenCurTyp, choice INT);
END admin_data;

CREATE PACKAGE BODY admin_data AS
   PROCEDURE open_cv (generic_cv IN OUT GenCurTyp, choice INT) 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;
END admin_data;
```

ホスト変数としてのカーソル変数の使用

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。カーソル変数を使用する場合は、ホスト変数として PL/SQL に渡す必要があります。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、弱い型指定の 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フェッチ

FETCH 文は、複数行の問合せの結果セットから、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}
[BULK COLLECT]
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 へ一度に 1 行ずつ行をフェッチします。

```
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;
```

BULK COLLECT 句を使用して、1 つのカーソル変数から 1 つ以上のコレクションに行のバルク・フェッチを行います。次に例を示します。

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   TYPE NameList IS TABLE OF emp.ename%TYPE;
   TYPE SalList IS TABLE OF emp.sal%TYPE;
   emp_cv EmpCurTyp;
   names  NameList;
   sals   SalList;
BEGIN
   OPEN emp_cv FOR SELECT ename, sal FROM emp;
   FETCH emp_cv BULK COLLECT INTO names, sals;
   ...
END;
```

カーソル変数がオープンしている場合のみ、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する必要があります。ただし、同じカーソル変数を使用して、別々の FETCH 文で異なる INTO 句を使用できます。各 FETCH 文で同じ結果セットから別の行をフェッチします。

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ている必要があります。また、フィールドまたは変数の数は、列の値の数と一致している必要があります。それ以外の場合はエラーになります。カーソル変数が強い型指定の場合はコンパイル時に、弱い型指定の場合は実行時にエラーが発生します。実行時に、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 句を使用して 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 モードを指定する必要があります。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 モードを指定する必要があります。

クローズしているか、または一度もオープンされていない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。

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 モードを指定する必要があります。

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。

カーソル変数の例 : マスター表とディテール表

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 つのディテール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・コードを使用して OPEN-FOR 文を選択し、適切なディテール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;
```

カーソル変数の例 : クライアント側の PL/SQL ブロック

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 ブロックを使用して、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;
```

カーソル変数の例 : Pro*C プログラム

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. */
}
```

カーソル変数の例 : SQL*Plus でのホスト変数の操作

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。ただし、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。

```
DECLARE
   ...
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 REFCURSOR を使用すると、カーソル変数を宣言でき、さらにそのカーソル変数を使用してストアド・サブプログラムから問合せ結果を戻すことができます。次のスクリプトでは、REFCURSOR 型のホスト変数を宣言します。問合せ結果を自動的に表示するには、SQL*Plus コマンド SET AUTOPRINT ON を使用します。

```
CREATE PACKAGE emp_data AS
   TYPE EmpRecTyp IS RECORD (
      emp_id    NUMBER(4),
      emp_name  VARCHAR2(10),
      job_title VARCHAR2(9),
      dept_name VARCHAR2(14),
      dept_loc  VARCHAR2(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 REFCURSOR
EXECUTE emp_data.get_staff(20, :cv)
```

ホスト・カーソル変数を PL/SQL に渡すときのネットワークの通信量の削減

ホスト・カーソル変数を PL/SQL に渡す場合は、OPEN-FOR 文をグループ化することでネットワークの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用にその作業域を使用できます。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、あらゆる用途に使用できます。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;
```

カーソル変数でのエラーの回避

代入に関係する両方のカーソル変数が強い型指定である場合は、両方が同じデータ型であることが必要です。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指定である場合は、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すると、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します。
- 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します。

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・プロシージャを作成します。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ません。たとえば、次の宣言は誤りです。

  ```
  CREATE PACKAGE emp_stuff AS
     TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
     emp_cv EmpCurTyp;  -- not allowed
  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ません。そのため、RPC を使用して、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ません。
- ホスト・カーソル変数を PL/SQL に渡す場合は、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取り出すことはできません。
- 比較演算子を使用して、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ません。
- NULL をカーソル変数に代入できません。
- REF CURSOR 型を使用して、CREATE TABLE 文または CREATE VIEW 文に列を指定できません。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ません。

- REF CURSOR 型を使用して、コレクションの要素型を指定できません。そのため、索引付き表、ネストした表または VARRAY の中の要素は、カーソル変数の値を格納できません。
- 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、もう一方が使用できません。たとえば、次のカーソル FOR ループはカーソル変数から取出しを実行しようとしているため、不正です。

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   emp_cv EmpCurTyp;
   ...
BEGIN
   ...
   FOR emp_rec IN emp_cv LOOP ...  -- not allowed
END;
```

カーソル属性の使用

どの明示カーソルおよびカーソル変数にも %FOUND、%ISOPEN、%NOTFOUND および %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、DML 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用できますが、SQL 文では使用できません。

明示カーソルの属性の概要

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、結果セットから取り出されます。

%FOUND カーソル属性 : 1 行がフェッチされたかどうか

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、直前のフェッチが行を戻した場合は TRUE に、直前のフェッチが行を戻さなかった場合は FALSE になります。次の例では、%FOUND を使用して、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 カーソル属性 : カーソルがオープンしているかどうか

%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 を使用して、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 カーソル属性 : フェッチが失敗したかどうか

%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 を使用して、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 TRUE である場合にのみ EXIT WHEN 文が実行されるためです。安全のために、次の EXIT 文をかわりに使用できます。

```
EXIT WHEN c1%NOTFOUND OR c1%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 カーソル属性 : これまでにフェッチされた行数

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。最初のフェッチの前は、%ROWCOUNT の評価結果は 0 です。その後は、これまでにフェッチした行の数になります。取出しで行が戻されるたびに、数値が増えていきます。次の例では、%ROWCOUNT を使用して、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 6-1 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 6-1 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、%NOTFOUND または %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。

カーソル属性の例

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の概要

暗黙カーソルの属性は、INSERT 文、UPDATE 文、DELETE 文または 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。

注意：SQL カーソルには、FORALL 文での使用に設計された別の属性 %BULK_ROWCOUNT があります。詳細は、5-43 ページの「%BULK_ROWCOUNT 属性を持つ FORALL の反復による影響をうける行カウント」を参照してください。

%FOUND カーソル属性 : DML 文で行が変更されたかどうか

SQL の DML 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 文または 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。それ例外の場合、%FOUND の結果は FALSE になります。次の例では、%FOUND を使用して、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 カーソル属性 : 暗黙カーソルの場合は常に FALSE

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 属性 : DML 文で行の変更が失敗したかどうか

%NOTFOUND は、論理的に %FOUND の逆です。INSERT 文、UPDATE 文または 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、%NOTFOUND の結果は TRUE になります。それ以外の場合、%NOTFOUND の結果は FALSE になります。

%ROWCOUNT 属性 : これまでに影響を受けた行数

%ROWCOUNT の結果は、INSERT 文、UPDATE 文または 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 文または 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 を使用して、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。

暗黙カーソルの属性を使用する場合のガイドライン

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用する場合は、ブール変数にただちに代入してください。プロシージャ 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りなく、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 の場合のみであるため、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。集計関数は、必ず値または NULL を戻すためです。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
```

カーソル式の使用

カーソル式はネストしたカーソルを戻します。結果セットの各行には、通常の値の他に、行内の他の値に関係する副問合せで生成されるカーソルも含まれることがあります。したがって、1 つの問合せにより、複数の表から取り出された関連値の、大きな集合を戻すことができます。結果セットは、最初にその行から、次に各行でネストしたカーソルからフェッチするネステッド・ループで処理できます。

PL/SQL では、カーソルの宣言、REF CURSOR の宣言および REF CURSOR 変数の一部として、カーソル式を持つ問合せがサポートされます。また、カーソル式は動的 SQL 問合せにも使用できます。次に構文を示します。

```
CURSOR ( subquery )
```

ネストしたカーソルは、それを含んでいる行が親カーソルからフェッチされるときに暗黙的にオープンされます。ネストしたカーソルがクローズされるのは、次の場合のみです。

- ネストしたカーソルをユーザーが明示的にクローズしたとき。
- 親カーソルが再実行されるとき。
- 親カーソルがクローズされるとき。
- 親カーソルが取り消されるとき。
- 親カーソルの 1 つのフェッチ中にエラーが呼び出されるとき。ネストしたカーソルはクリーン・アップの一部としてクローズされます。

カーソル式の制限

- カーソル式は、暗黙カーソルとは併用できません。
- カーソル式を使用できるのは、次の場合のみです。
  - カーソル式自体が副問合せである場合を除き、他の問合せの式の中でネストされていない SELECT 文中。
  - SELECT 文の FROM 句で、表関数の引数として。
- カーソル式を使用できるのは、問合せ仕様部の最も外側の SELECT リスト内のみです。
- カーソル式はビュー宣言には使用できません。
- カーソル式の BIND および EXECUTE 操作は実行できません。

カーソル式の例

この例では、指定した所在地 ID と、その所在地にある全部門をフェッチできるカーソルを検索します。各部門の名前をフェッチすると、別の表から関連する従業員詳細をフェッチできる、もう 1 つのカーソルも取得します。

```
CREATE OR REPLACE procedure emp_report(p_locid number) is
TYPE refcursor is ref cursor;
-- The query returns only 2 columns, but the second column is
-- a cursor that lets us traverse a set of related information.

CURSOR c1 is
SELECT l.city,
CURSOR(SELECT d.department_name,
CURSOR(SELECT e.last_name
FROM employees e
WHERE e.department_id = d.department_id) as ename
FROM departments d where l.location_id = d.location_id) dname
FROM locations l
WHERE l.location_id = p_locid;

loccur refcursor;
deptcur refcursor;
empcur refcursor;

V_city locations.city%type;
V_dname departments.department_name%type;
V_ename employees.last_name%type;

BEGIN
OPEN c1;
LOOP
FETCH C1 INTO v_city, loccur;
EXIT WHEN c1%notfound;
-- We can access the column C1.city, then process the results of
-- the nested cursor.

LOOP
FETCH loccur INTO v_dname, deptcur; -- No need to open
EXIT WHEN loccur%notfound;

LOOP
FETCH deptcur into v_ename; -- No need to open
EXIT WHEN deptcur%notfound;
DBMS_OUTPUT.PUT_LINE(v_city ||' '||v_dname||' '||v_ename);
END LOOP;
END LOOP;
END LOOP;
```

```
close c1;
END;
/
```

PL/SQL におけるトランザクション処理の概要

この項では、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続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。並行処理制御が十分ではないと、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、ロックを使用してデータへの同時アクセスを制御します。ロックを使用すると、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、ロックを使用し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ため、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かぎり、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「読込み一貫性」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・セグメントに記録します。Oracle は、ロールバック・セグメントを使用して、読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 DML 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（永続化）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リストアされます。

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 文を使用すると、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。カレント・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、現行の位置に名前を付けてマークします。

COMMIT による変更の永続化

COMMIT 文は、カレント・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続的なものとします。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。

オプションの COMMENT 句を使用すると、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続的なものとなります。ただし、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あることが必要です。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';  -- not allowed
```

ROLLBACK による変更の取消し

ROLLBACK 文は、カレント・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。ロールバックが使用される理由は 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リストアできます。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は、ロールバックを使用すると、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。このようにすべての変更内容を取り消す場合には、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 文で一意の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 文による処理のみです。カレント・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 文のみ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 つにエラーを戻し、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 はその文を解析する必要があります。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する必要があります。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 による変更の一部取消し

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、その挿入のみ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のみが消去されます。単純なロールバックまたはコミットではすべてのセーブポイントが消去されます。

再帰的サブプログラムの中でセーブポイントをマークすると、再帰しながら進む過程で、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のみです。

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のみに影響を与えます。次に例を示します。

```
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。

Oracle による暗黙的なロールバックの実行方法

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 文のみ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 パラメータに代入しません。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。コミットを発行するか、あるいは PL/SQL プログラムまたはホスト環境でロールバックするかは、アプリケーション・ロジック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行うことによって決まります。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 パラメータを使用して 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 を使用したトランザクション・プロパティの設定

SET TRANSACTION 文を使用すると、読取り専用または読取り / 書込みトランザクションの開始、分離レベルの確立、指定したロールバック・セグメントへのカレント・トランザクションの割当てができます。読取り専用トランザクションは、他のユーザーが更新している 1 つ以上の表に対して、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込み一貫性のあるビューが作成され、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用して、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 NAME 'Calculate sales figures';
   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 文である必要があり、1 つのトランザクションで 1 回しか使用できません。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 を使用しても、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

読取り専用トランザクションに使用できるのは、SELECT INTO、OPEN、FETCH、CLOSE、LOCK TABLE、COMMIT および ROLLBACK 文のみです。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。ただし、デフォルトのロックを上書きする方がよい場合は、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 文を使用すると、明示的に表全体をロックできます。SELECT FOR UPDATE 文を使用すると、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する場合以外は使用しないでください。

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 句を使用して排他的な行ロックを取得する必要があります。次に例を示します。

```
DECLARE
   CURSOR c1 IS SELECT empno, sal FROM emp
      WHERE job = 'SALESMAN' AND comm > sal
      FOR UPDATE NOWAIT;
```

SELECT ... FOR UPDATE 文は、更新または削除する行を識別し、結果セット内の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 は他のユーザーが要求された行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ため、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 は行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のではありません。また、トランザクションをコミットまたはロールバックすると、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。（回避策の詳細は、6-50 ページの「コミットにまたがるフェッチ」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 句を使用して、ロックを特定の表に制限できます。表の行は、FOR UPDATE OF 句でその表の列を参照する場合にのみロックされます。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 句を使用します。

```
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 文を使用して、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用できるかが決まります。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 人のユーザーのみです。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、『Oracle9*i* 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のみ、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フェッチ

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がロックされ、トランザクションをコミットするとロックが解除されます。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する場合は、FOR UPDATE 句と CURRENT OF 句は使用しないでください。そのかわりに、ROWID 疑似列を使用して CURRENT OF 句と同じ処理を実行します。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 を使用して現在行を識別します。次に例を示します。

```
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 句を使用していないため、取り出された行は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、カーソルが読込み一貫性のあるデータのビューを持つ必要があります。これは、更新に使用されたロールバック・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 属性を使用する例を次に示します。

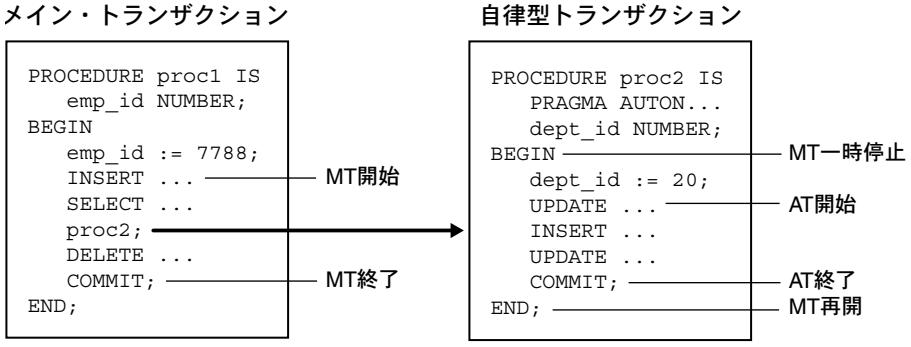
```
DECLARE
   CURSOR c1 IS SELECT ename, sal, rowid FROM emp;
   emp_rec c1%ROWTYPE;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO emp_rec;
      EXIT WHEN c1%NOTFOUND;
      ...
      IF ... THEN
         DELETE FROM emp WHERE rowid = emp_rec.rowid;
      END IF;
   END LOOP;
   CLOSE c1;
END;
```

自律型トランザクションによる独立した作業単位の実行

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する必要があります。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 6-1 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 6-1 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、リソースまたは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使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、使用しやすくなります。

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、AUTONOMOUS_TRANSACTION プラグマ（コンパイラ・ディレクティブ）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に 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： パッケージのすべてのサブプログラム（またはオブジェクト型のすべてのメソッド）を自律型としてマークするためにプラグマを使用することはできません。自律型としてマークできるのは、個々のルーチンのみです。たとえば、次のプラグマは誤りです。

```
CREATE PACKAGE banking AS
   PRAGMA AUTONOMOUS_TRANSACTION; -- not allowed
   ...
   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： 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

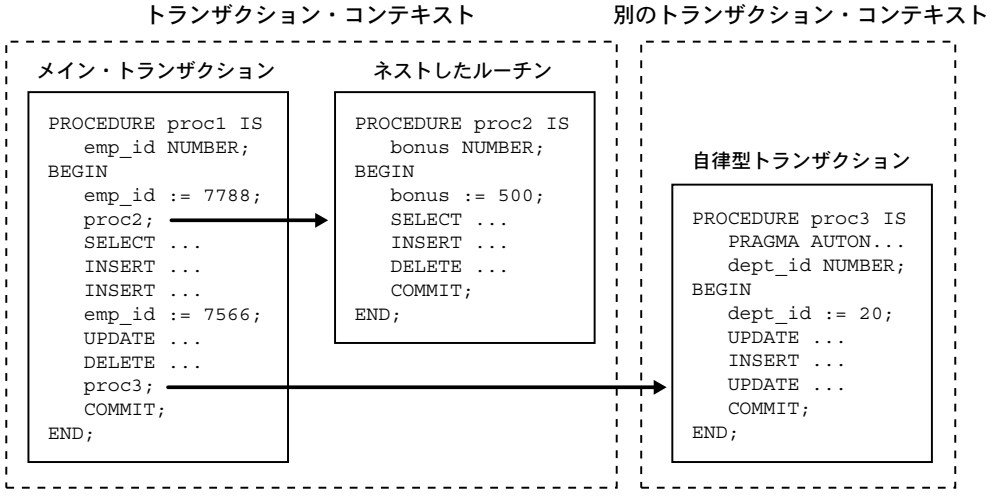
自律型トランザクションは別のトランザクションによって開始されますが、これはネストしたトランザクションではありません。その理由は次のとおりです。

- 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ません。
- メイン・トランザクションに依存しません。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は、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ません。
- 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ます。（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ません。）
- 自律型トランザクションで例外が呼び出されると、文レベルのロールバックではなくトランザクション・レベルのロールバックが発生します。

トランザクション・コンテキスト

図 6-2 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ルーチンと共有しますが、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は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 6-2 トランザクション・コンテキスト



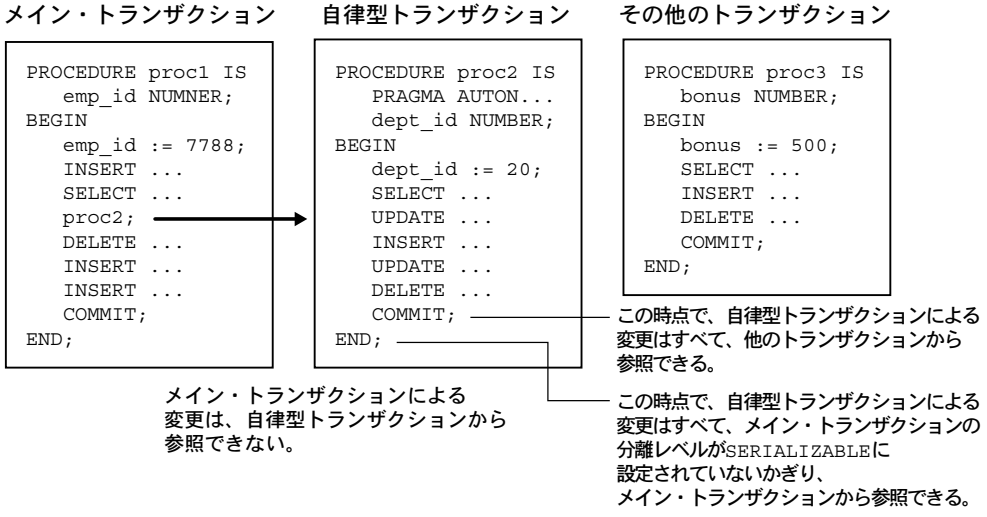
トランザクションの可視性

6-56 ページの図 6-3 のように、自律型トランザクションによって行われた変更は、その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のみです。

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 6-3　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。カレント・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）トランザクションのみに適用されます。

- `COMMIT`
- `ROLLBACK [TO savepoint_name]`
- `SAVEPOINT savepoint_name`
- `SET TRANSACTION`

`COMMIT` 文は、カレント・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続的なものとします。`ROLLBACK` 文は、カレント・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のみを取り消します。`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、読取り / 書込み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のみに適用され、自律型トランザクションには適用されません。逆の場合も同じです。

開始と終了

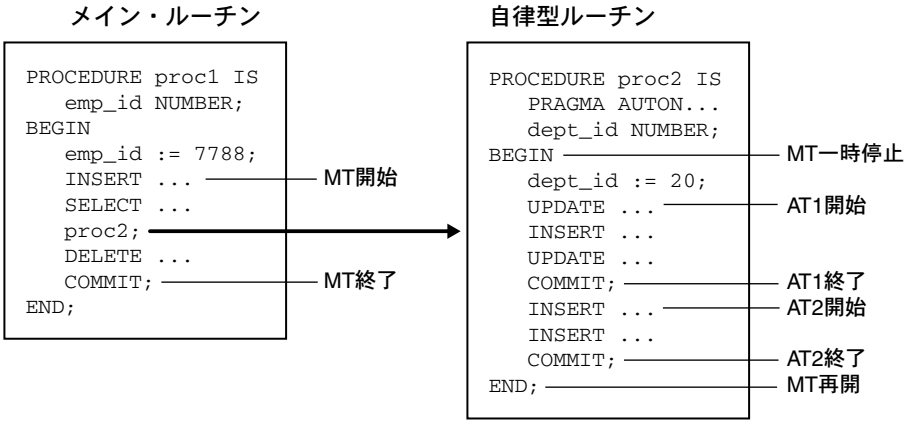
自律型ルーチンの実行部に入ると、メイン・トランザクションは停止します。ルーチンを終了すると、メイン・トランザクションは再開します。

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 6-4 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 6-4 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型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、カレント・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。

- メイン・トランザクション（自律型ルーチンの終了まで再開できない）が保持するリソースに、自律型トランザクションがアクセスしようとすると、デッドロックが発生します。この場合、Oracle は自律型トランザクションで例外を呼び出します。例外が未処理になった場合、自律型トランザクションはロールバックされます。
- 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します。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、この最大数を超えてしまう場合があります。

- コミットまたはロールバックせずにアクティブな自律型トランザクションを終了しようとすると、Oracle は例外を呼び出します。例外が未処理になった場合、トランザクションはロールバックされます。

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するとします。次の例では、トリガーを使用して、重複する行をシャドウ表に挿入します。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、シャドウ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));

-- create 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
```

通常のトリガーとは異なり、自律型トリガーはシステム固有の動的 SQL を使用して、DDL 文を実行できます（第 11 章を参照）。次の例では、表 bonus が更新された後にトリガー bonus_trig が一時データベース表を削除します。

```
CREATE TRIGGER bonus_trig
AFTER UPDATE ON bonus
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;  -- enables trigger to perform DDL
BEGIN
   EXECUTE IMMEDIATE 'DROP TABLE temp_bonus';
END;
```

データベース・トリガーの詳細は、『Oracle9*i* 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。

SQL からの自律型ファンクションのコール

SQL 文からコールされるファンクションは、副作用を制御するための特定の規則に従う必要があります。（8-9 ページの「PL/SQL サブプログラムの副作用の制御」を参照。）この規則に違反していないかどうかを確認するには、RESTRICT_REFERENCES プラグマを使用できます。プラグマは、関数によるデータベース表またはパッケージ変数に対する読込みまたは書込みが行われていないことを示します。（詳細は、『Oracle9*i* 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。）

ただし、自律型ルーチンの動作に関係なく、「データベース読込み禁止状態」（RNDS）および「データベース書込み禁止状態」（WNDS）の規則に違反しないように定義できます。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;
```

PL/SQL プログラムの下位互換性の保証

PL/SQL バージョン 2 では、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 を使用すると、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 型の前方参照が行えます。
- ファンクション仕様部の RETURN 句に変数（データ型ではなく）の名前を指定できます。
- 索引付き表の IN パラメータの要素に値を代入できます。
- 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ます。
- 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ます。
- SELECT 文の FROM リスト内の OUT パラメータを使用できます。

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します。

```
PLSQL_V2_COMPATIBILITY=TRUE
```

- 次のいずれかの SQL 文を実行します。

```
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、現行のデフォルト動作しか認められません。

クライアント側では、コマンドライン・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、ランタイム・オプション DBMS をコマンドラインに指定します。

7

PL/SQL エラーの処理

ランタイム・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す。

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かぎり、「スタック・オーバーフロー（*stack overflow*）」や「ゼロによる除算（*division by zero*）」のようなランタイム・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。

この章の項目は、次のとおりです。

PL/SQL のエラー処理の概要

PL/SQL では、警告またはエラー条件が例外と呼ばれます。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名前を付ける必要があります。

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。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す必要があります（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成します。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い場合は、制御はホスト環境に戻ります。

次の例では、ティッカ・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、次のようにすることをお薦めします。

```
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。収益がゼロの場合、ファンクション DECODE は NULL を戻します。ゼロではない場合、DECODE は株価収益率を戻します。

PL/SQL 例外の利点

エラー処理に例外を使用すると、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する必要があります。

```
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。検査コードを作成し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成しなくても、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・リカバリ・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のみです。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 PL/SQL 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する必要があります。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 文が行を戻さなかった場合は、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 ハンドラを使用します。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 ハンドラで使用すると便利です。あるいは EXCEPTION_INIT プラグマを使用して、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用すれば、事前定義の例外を処理するハンドラを作成できます。

例外	Oracle エラー	SQLCODE 値
ACCESS_INTO_NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
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（基本構造的に NULL）オブジェクトの属性に値を代入しようとしたとき。
CASE_NOT_FOUND	CASE 文の WHEN 句で何も選択されておらず、ELSE 句もない場合。
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（基本構造的に 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、一度クローズする必要があります。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します。このため、ループの内側ではカーソルをオープンできません。
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 が呼び出されます。）この例外は、バルク FETCH 文の LIMIT 句の式が正数に評価されない場合にも呼び出されます。
LOGIN_DENIED	不正なユーザー名 / パスワードで Oracle にログオ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 を戻します。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ありません。FETCH 文は最終的には行を戻さないと予想されますが、その場合は、例外は呼び出されません。
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必要です。
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す。
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用し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）の索引番号を使用し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、文字列からユニバーサル 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 を呼び出します。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 が呼び出されます。（SQL 文では、INVALID_NUMBER が呼び出されます。）
ZERO_DIVIDE	数値をゼロで割ろうとしたとき。

独自の PL/SQL 例外の定義

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する必要があり、RAISE 文を使用して明示的に呼び出す必要があります。

PL/SQL 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 文では使用できません。ただし、変数と例外の有効範囲規則は同じです。

PL/SQL 例外の有効範囲規則

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ため、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、次の構文を使用する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ます。呼び出された例外を外側のブロックで処理するには、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。

PL/SQL 例外と番号の対応付け : EXCEPTION_INIT プラグマ

事前定義の名前がないエラー状態（通常は ORA- メッセージ）を処理するには、OTHERS ハンドラまたは EXCEPTION_INIT プラグマを使用する必要があります。**プラグマ**は、実行時ではなくコンパイル時に処理されるコンパイラ・ディレクティブです。

PL/SQL では、EXCEPTION_INIT プラグマでコンパイラに指示して、例外名と Oracle エラー番号を対応付けます。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。**エラー・スタック**または一連のエラー・メッセージを確認する場合、一番上のエラーがトラップおよび処理できるエラーです。

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、次の構文を使用して指定します。

```
PRAGMA EXCEPTION_INIT(exception_name, -Oracle_error_number);
```

exception_name は事前に宣言されている例外の名前で、番号は ORA- エラー番号に対応する負の値です。次の例に示すとおり、プラグマは、同じ宣言部内の例外宣言より後に表示されます。

```
DECLARE
   deadlock_detected EXCEPTION;
   PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
   ... -- Some operation that causes an ORA-00060 error
EXCEPTION
   WHEN deadlock_detected THEN
      -- handle the error
END;
```

独自のエラー・メッセージの定義 : プロシージャ RAISE_APPLICATION_ERROR

プロシージャ RAISE_APPLICATION_ERROR を使用すると、ストアド・サブプログラムからユーザー定義の ORA- 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回避できます。

RAISE_APPLICATION_ERROR をコールするには、次の構文を使用します。

```
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 バイト以内の文字列です。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。RAISE_APPLICATION_ERROR はパッケージ DBMS_STANDARD の一部で、パッケージ STANDARD と同様に、参照する際に名前を修飾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）からのみ raise_application_error をコールできます。raise_application_error が呼び出されると、サブプログラムは終了し、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 エラーのようにトラップ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 をコールしています。

```
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・レポート・ファンクション SQLCODE および SQLERRM を使用して OTHERS ハンドラで処理できます。また、EXCEPTION_INIT プラグマを使用すると、次の Pro*C 例が示すように 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;
```

この手法を使用すると、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ため、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。この場合は、ドット表記法を使用して、次のように事前定義の例外を指定する必要があります。

```
EXCEPTION
   WHEN invalid_number OR STANDARD.INVALID_NUMBER THEN
      -- handle the error
END;
```

PL/SQL の例外の呼出し

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 を使用して 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 文を使用した例外の呼出し

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のみにする必要があります。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 := 7;
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;
```

PL/SQL 例外の伝播

例外が呼び出されたときに、PL/SQL がその例外のハンドラをカレント・ブロックまたはサブプログラムで発見できない場合、例外は伝播します。つまり、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）には伝播しません。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。回避策の詳細は、7-9 ページの「独自のエラー・メッセージの定義 : プロシージャ RAISE_APPLICATION_ERROR」を参照してください。

図 7-1、図 7-2 および図 7-3 に、基本的な伝播規則を示します。

図 7-1 伝播規則 : 例 1

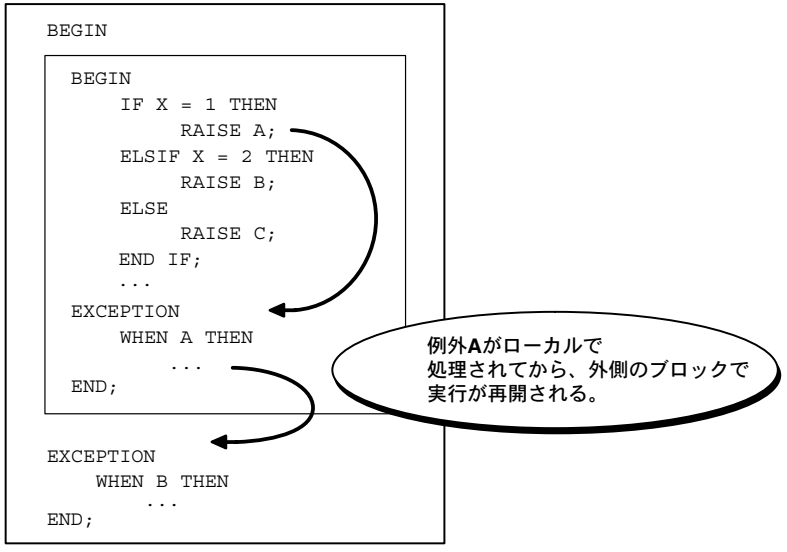


図 7-2 伝播規則 : 例 2

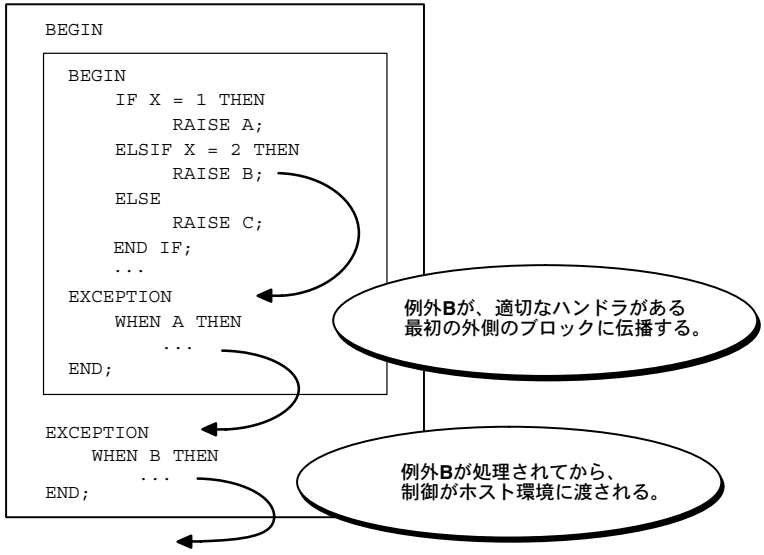
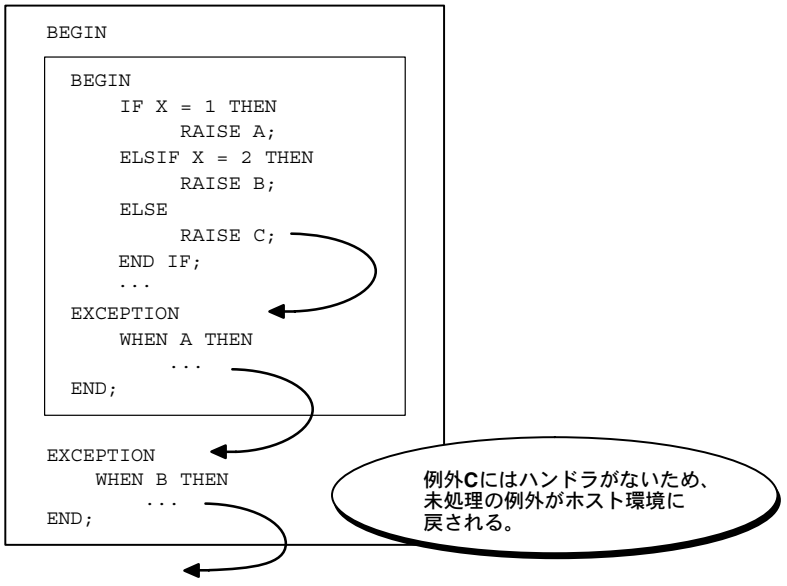

BEGIN
BEGIN
IF X = 1 THEN
RAISE A;
ELSIF X = 2 THEN
RAISE B;
ELSE
RAISE C;
END IF;
...
EXCEPTION
WHEN A THEN
...
END;
EXCEPTION
WHEN B THEN
...
END;
例外Bが、適切なハンドラがある
最初の外側のブロックに伝播する。
例外Bが処理されてから、
制御がホスト環境に渡される。

図 7-3 伝播規則 : 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伝播することがあります。次の例を考えます。

```
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伝播します。しかし、有効範囲規則によれば、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。ユーザー定義の例外のハンドラがない場合は、呼出し側のアプリケーションは次のエラーを受け取ります。

```
ORA-06510: PL/SQL: ユーザー定義の例外が発生しましたが、処理されませんでした
```

PL/SQL 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、エラーを外側のブロックの中でログする場合があります。

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 文を使用します。

```
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）、現在の例外が再度呼び出されます。

呼び出された PL/SQL 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にする必要があります。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用されます。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 つのみです。

次の例で示すように、OTHERS ハンドラを使用すると、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、同じ一連の文を実行する場合は、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用できず、単独で使用する必要があります。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、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用できません。

PL/SQL 変数の通常の有効範囲規則が適用されるため、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。ただし、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の処理

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ため、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伝播するため、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の処理

ブロックまたはサブプログラムの例外処理部の中で、アクティブになる例外は一度に 1 つのみです。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伝播し、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。それ以降の例外の伝播は通常どおりに起こります。次の例を考えます。

```
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。

エラー・コードとエラー・メッセージの取得 : SQLCODE および SQLERRM

例外ハンドラでは、組込みファンクション SQLCODE および SQLERRM を使用して、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「データが見つかりません。」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 は対応するエラー・メッセージを戻します。

ただし、EXCEPTION_INIT プラグマを使用して例外名に 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含めて 512 文字以内です。

例外が発生していない場合、SQLCODE はゼロを戻し、SQLERRM は「ORA-0000: 正常に完了しました。」のメッセージを戻します。

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戻します。SQLERRM に渡すエラー番号は負の値にしてください。次の例では、正の値を渡したため、予期しない結果が得られます。

```
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 に正数を渡すと、必ず「ユーザー定義の例外」というメッセージが戻されます。+100 を渡した場合は例外で、この場合 SQLERRM は「データが見つかりません。」というメッセージを戻します。SQLERRM にゼロを渡すと、常にメッセージ「正常に完了しました。」を戻します。

SQLCODE または SQLERRM は、SQL 文では直接使用できません。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 文の中で使用する必要があります。

```
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;
```

文字列ファンクション SUBSTR を使用しているため、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ファンクション SQLCODE および SQLERRM は、特に OTHERS 例外ハンドラで使用すると便利です。

注意：RESTRICT_REFERENCES プラグマを使用してストアド・ファンクションの純正度を示すとき、ファンクションが SQLCODE または SQLERRM をコールする場合は、WNPS および RNPS 制約は指定できません。

未処理例外の捕捉

発生し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「例外は処理されませんでした。」というエラーを戻します。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（NOCOPY パラメータではない場合）。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 プログラムの最も上のレベルに 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。

PL/SQL エラーの処理のヒント

ここでは、柔軟性が高い 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リカバリできます。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;
```

ある文の例外を処理してから、次の文に進むことができます。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、外側のブロックは、そのサブブロックの終了位置から実行を継続します。次の例を考えます。

```
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。

また、一部で失敗した可能性がある一連の DML 操作を実行し、操作全体が完了した後で例外を処理する方法もあります。5-44 ページの「%BULK_EXCEPTIONS 属性を持つ FORALL 例外の処理」を参照してください。

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せずに、再試行する場合があります。そのテクニックは、次のとおりです。

1. トランザクションをサブブロックに入れます。
2. 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。
3. 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 ループを使用して、試行の回数を制限することをお薦めします。

```
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;
```

ロケータ変数を使用した例外の位置の識別

一連の文に対して 1 つの例外ハンドラを使用すると、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 ...
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、必要な場合は、次のようにロケータ変数を使用し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 ...
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      INSERT INTO errors VALUES ('Error in statement ' || stmt);
END;
```

8

PL/SQL のサブプログラム

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用し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。サブプログラムはビルディング・ブロックのようなもので、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。

この章の項目は、次のとおりです。

サブプログラム

サブプログラムは、パラメータを取得し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用し、ファンクションは値を計算するために使用します。

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、カーソル、定数、変数、例外および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用します。その後、出金の金額を使用し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、新しい部門を作成す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、管理が容易な、正しく定義されたモジュールに分けます。この特性を利用すると、トップダウン設計と段階的詳細化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用できます。定義が変わった場合でも、影響の範囲をサブプログラムのみに抑えられます。このため、メンテナンスが簡単に実施できます。

最後に、サブプログラムは抽象化を実現し、ユーザーを個々の詳細な処理から解放します。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）を使用すると、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を定義しないで済ますことができます。

PL/SQL 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成します。

```
[CREATE [OR REPLACE]]
PROCEDURE procedure_name[(parameter[, parameter]...)]
   [AUTHID {DEFINER | CURRENT_USER}] {IS | AS}
   [PRAGMA AUTONOMOUS_TRANSACTION;]
   [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
   [{:= | DEFAULT} expression]
```

CREATE 句を使用するとスタンドアロン・プロシージャを作成できます。これは Oracle データベース内に格納されます。CREATE PROCEDURE 文は、SQL*Plus またはシステム固有の動的 SQL を使用したプログラムから対話式で実行できます（第 11 章を参照してください）。

AUTHID 句は、ストアド・プロシージャをその所有者（デフォルト）と現行ユーザーのどちらの権限で実行するかという点、およびスキーマ・オブジェクトへの未修飾の参照を所有者と現行ユーザーのどちらのスキーマで解決するかという点を決定します。CURRENT_USER を指定すると、デフォルトの動作を変更できます。詳細は、8-50 ページの「実行者権限と定義者権限」を参照してください。

AUTONOMOUS_TRANSACTION プラグマはプロシージャを自律型（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。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。詳細は、6-52 ページの「自律型トランザクションによる独立した作業単位の実行」を参照してください。

パラメータのデータ型に制約を課すことはできません。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ため誤りです。

```
PROCEDURE reconcile (acct_id CHAR(5)) IS ...  -- illegal
```

ただし、次の回避策を使用して、パラメータ型に間接的にサイズ制約を課すことができます。

```
DECLARE
   SUBTYPE Char5 IS CHAR(5);
   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（または AS）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）の 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 は使用しません。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 つの文が存在している必要があります。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用します。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。

```
raise_salary(emp_id, amount);
```

PL/SQL 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（ローカル）ファンクションの構文を示します。

```
[CREATE  [OR REPLACE ] ]
FUNCTION function_name [ ( parameter [ , parameter ]... ) ] RETURN datatype
   [ AUTHID { DEFINER  | CURRENT_USER } ]
   [ PARALLEL_ENABLE
    [ { [CLUSTER parameter BY (column_name [, column_name ]... ) ] |
     [ORDER parameter BY (column_name [ , column_name ]... ) ] } ]
     [ ( PARTITION parameter BY
       { [ {RANGE | HASH } (column_name [, column_name]...)] | ANY } ) ]
   ]
   [DETERMINISTIC]    [ PIPELINED  [ USING implementation_type ] ]
   [ AGGREGATE  [UPDATE VALUE]  [WITH EXTERNAL CONTEXT]
USING  implementation_type ]  {IS | AS}
   [ PRAGMA AUTONOMOUS_TRANSACTION; ]
   [ local declarations ]
BEGIN
   executable statements
[ EXCEPTION
   exception handlers ]
END [ name ];
```

CREATE 句を使用するとスタンドアロン・ファンクションを作成できます。これは Oracle データベース内に格納されます。CREATE FUNCTION 文は、SQL*Plus またはシステム固有の動的 SQL を使用したプログラムから対話式で実行できます。

AUTHID 句は、ストアド・ファンクションをその所有者（デフォルト）と現行ユーザーのどちらの権限で実行するかという点、およびスキーマ・オブジェクトへの未修飾の参照を所有者と現行ユーザーのどちらのスキーマで解決するかという点を決定します。CURRENT_USER を指定すると、デフォルトの動作を変更できます。

PARALLEL_ENABLE オプションは、ストアド・ファンクションがパラレル DML 評価のスレーブ・セッションで安全に使用されることを宣言します。メイン（ログオン）・セッションの状態が、スレーブ・セッションと共有されることはありません。スレーブ・セッションごとに固有の状態があり、セッション開始時に初期化されます。ファンクションの結果がセッション（static）変数の状態に依存しないようにしてください。さもないと、セッションごとに結果が異なる可能性があります。

ヒント DETERMINISTIC は、オプティマイザが冗長なファンクション・コールを回避するのに役立ちます。ストアド・ファンクションが同じ引数で事前にコールされた場合は、オプティマイザは前の結果を使用できます。ファンクションの結果をセッション変数の状態またはスキーマ・オブジェクトに依存させないでください。さもないと、コールごとに結果が異なる可能性があります。DETERMINISTIC ファンクションのみが、ファンクションベースの

索引または問合せ再作成を使用可能にしたマテリアライズド・ビューからコールできます。詳細は、Oracle9*i*SQL リファレンスを参照してください。

AUTONOMOUS_TRANSACTION プラグマはファンクションを自律型（独立型）としてマークするように PL/SQL コンパイラに指示します。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。

パラメータまたはファンクション戻り値のデータ型に、NOT NULL などで制約を課すことはできません。ただし、回避策を使用して、パラメータ型に間接的にサイズ制約を課すことができます。8-4 ページの「PL/SQL プロシージャ」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、戻り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（または AS）で始め、キーワード END で終わります。END の後には、オプションのファンクション名を続けます。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）の 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 は使用しません。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 文が存在している必要があります。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。

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 VARCHAR2) 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 から範囲の制限を選択するために使用します。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る場合は、sal_ok は FALSE に設定され、超えていない場合は TRUE に設定されます。

次の例に示すように、ファンクションは、式の一部としてコールされます。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。

```
IF sal_ok(new_sal, new_title) THEN ...
```

RETURN 文の使用

RETURN 文は、サブプログラムの実行を即座に完了させ、コール元に制御を戻します。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。(この RETURN 文を、ファンクションの仕様部の中で戻り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。)

サブプログラムでは、複数の RETURN 文を使用できます。RETURN 文を最後の文にす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。ただし、サブプログラムに複数の終了点を作成す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 文に戻り値を含めることはできず、したがって式を含めることもできません。RETURN 文には、プロシージャ本来の終了地点に達する前に、コール元に制御を戻す役割のみがあります。

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る必要があります。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、複雑な式も使用します。

```
FUNCTION compound (
   years  NUMBER,
   amount NUMBER,
   rate   NUMBER) RETURN NUMBER IS
BEGIN
   RETURN amount * POWER((rate / 100) + 1, years);
END compound;
```

ファンクションには、RETURN 文へ導く少なくとも 1 つの実行パスが必要です。実行パスがない場合は、実行時に「ファンクションが値なしで戻されました。」というエラーが発生します。

PL/SQL サブプログラムの副作用の制御

ストアド・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」規則に従っている場合にかぎり、SQL 文からコールできます。

- SELECT 文またはパラレル化 INSERT 文､UPDATE 文または DELETE 文からコールされた場合、ファンクションはデータベース表を変更できません。
- 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の問合せや変更はできません。
- 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）を実行できません。また、DDL 文（CREATE など）には自動確定が続くため、これも実行できません。

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。

この規則に違反していないかどうかを確認するには、RESTRICT_REFERENCES プラグマ（コンパイラ・ディレクティブ）を使用します。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）に対する読込みや書込み、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 がデータベース書込み禁止状態（WNDS）、およびパッケージ読込み禁止状態（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。

純正規則と RESTRICT_REFERENCES プラグマの詳細は、『Oracle9*i* 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。

PL/SQL の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、サブプログラムまたは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する必要があります。

PL/SQL では、使用する前に識別子を宣言する必要があります。したがって、サブプログラムも、コールする前に宣言する必要があります。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。award_bonus コールの時点では宣言されていないプロシージャ calc_rating をコールしているためです。

```
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。ただし、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である場合（互いにコールし合う場合）や、論理順またはアルファベット順に定義する場合などがそうです。

このような問題を解決するには、前方宣言と呼ばれる特殊なサブプログラム宣言を使用します。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
   ...
```

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、前方宣言の後であればどこでもかまいませんが、同じプログラム・ユニットの中に置く必要があります。

複数の PL/SQL サブプログラムのパッケージ化

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ができます。これはデータベースに格納されます。それによって、サブプログラムは複数のアプリケーションで共用できます。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。次に例を示します。

```
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 (emp_id, name, ...);
   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
   BEGIN
      UPDATE emp SET sal = sal + amount 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。ただし、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 9 章「PL/SQL パッケージ」を参照してください。

サブプログラムの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用して情報を渡します。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 および 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用しています。

```
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
BEGIN
   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結果が代入されます。実パラメータの値を仮パラメータに代入する前に、必要に応じて PL/SQL は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る必要があります。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、結果は新規のデータ型に変換可能である必要があります。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
```

サブプログラムのパラメータの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には、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関連は、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
```

位置表記法の使用

1 つ目のプロシージャ・コールでは位置表記法を使用しています。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。

名前表記法の使用

2 つ目のプロシージャ・コールでは名前表記法を使用しています。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・コールでも名前表記法を使用しています。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。

混合表記法の使用

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用し、2 番目のパラメータで名前表記法を使用しています。位置表記法は名前表記法よりも前で使用する必要があります。逆の順序で使用することはできません。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
```

サブプログラムのパラメータのモードの指定

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用します。サブプログラムで使用できるモードには、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 モードを使用しないでください。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は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 モードの使用

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用します。サブプログラムの中では、IN パラメータは定数のように取り扱われます。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を使用できます。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、8-19 ページの「サブプログラムのパラメータのデフォルト値の使用」を参照してください。

OUT モードの使用

OUT パラメータは、サブプログラムのコール元に値を戻すために使用します。サブプログラムの中では、OUT パラメータは変数のように取り扱われます。つまり、OUT 仮パラメータは、ローカル変数のように使用できます。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、変数である必要があります。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。ただし、サブプログラムをコールすると、コンパイラ・ヒント NOCOPY（8-17 ページの「NOCOPY コンパイラ・ヒントを使用した大型データ構造の受渡し」を参照）を指定していない場合は値が失われ、あるいはサブプログラムが未処理例外を発生して終了します。

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。次に例を示します。

```
DECLARE
   SUBTYPE Counter IS INTEGER NOT NULL;
   rows Counter := 0;
   PROCEDURE count_emps (n OUT Counter) IS
   BEGIN
      SELECT COUNT(*) INTO n FROM emp;
   END;
```

```
BEGIN
   count_emps(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。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 モードの使用

IN OUT パラメータを使用すると、コール先のサブプログラムに初期値を渡して、コール元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、変数である必要があります。

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終了すると、PL/SQL は実パラメータに値を代入しません。

サブプログラムのパラメータのモードの概要

表 8-1 に、パラメータ・モードの詳細をまとめます。

表 8-1 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	コール元に値を戻す	サブプログラムに初期値を渡し、更新された値をコール元に戻す
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する必要がある	仮パラメータには値を代入する必要がある
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用できる	実パラメータは変数である必要がある	実パラメータは変数である必要がある
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場合、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場合、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）

NOCOPY コンパイラ・ヒントを使用した大型データ構造の受渡し

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に、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 25000 レコードがロードされます。この表は、NULL 文を実行するのみの 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 21 秒かかります。NOCOPY を使用すると、他のプロシージャへのコールはすぐに完了します。

```
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 NUMBER(5);
  5     t2 NUMBER(5);
  6     t3 NUMBER(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(24999, 1);  -- copy element 1 into 2..25000
 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: 21
With NOCOPY: 0
```

NOCOPY の使用によるパフォーマンス向上のトレードオフ

NOCOPY を使用すると、正しく定義された例外処理方法と引き換えに、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であるため、コンパイラは NOCOPY パラメータを、値方式か参照方式によってサブプログラムに渡すことができます。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ません。
- 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ます。しかし、NOCOPY を指定すると、仮パラメータへの代入値はただちに実パラメータにも影響します。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」されません。
- 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ます。このため、アプリケーションをパーティション化するときに、例外処理方法が暗黙的に変更されることがあります。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を参照方式で渡すことはできません。

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、8-21 ページの「サブプログラムのパラメータのエイリアシングの理解」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す。この制限は索引付き表全体には適用されません。
- 実パラメータには（たとえば位取りや NOT NULL などによって）制約があります。この制限は制約付きの要素または属性には拡張されません。また、サイズ制約付きの文字列にも適用されません。
- 実パラメータと仮パラメータはレコードです。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ります。
- 実パラメータと仮パラメータはレコードです。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ります。
- 実パラメータを渡すには、暗黙的なデータ型変換が必要になります。
- 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（remote procedure call: RPC）に関連します。

サブプログラムのパラメータのデフォルト値の使用

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様々な数値を渡し、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 VARCHAR2 DEFAULT 'TEMP',
    new_loc   VARCHAR2 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用されます。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ため、両方のデフォルト値が使用されます。2 番目のコールでは実パラメータを 1 つ渡しているため、new_loc はデフォルト値が使用されます。3 番目のコールでは実パラメータを 2 つ渡しているため、デフォルト値はどちらも使用されません。

通常は、位置表記法を使用すると、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用しても、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');  -- not allowed
```

このような場合は、次のように名前表記法を使用します。

```
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 ...  -- not allowed
```

このため、次のように NULL を明示的に渡す必要があります。

```
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成する場合、DEFAULT 句の中ではホスト変数（バインド変数）を使用できません。SQL*Plus の次の例では、「バインド変数が正しくありません。（*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, ...
```

サブプログラムのパラメータのエイリアシングの理解

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、値へのポインタのみが渡されます。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用され、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択するパラメータの受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);
   dbms_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渡し方式によって異なります。コンパイラが値方式を選択する場合、word_list と lexicon は同じ VARRAY の別個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。ただし、コンパイラが参照方式を選択する場合、word_list および lexicon は名前のみが異なる同じ VARRAY となります。「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。ただし、n3 は NOCOPY パラメータであるため、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す。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 行目が取り出され、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;
```

サブプログラム名のオーバーロード

PL/SQL ではサブプログラム名と型の順序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用できます。

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;
BEGIN
   ...
END;
```

次のようなプロシージャを作成すると、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成すると、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 は仮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断します。次の例で PL/SQL が使用する initialize のバージョンは、プロシージャを DateTabTyp パラメータまたは RealTabTyp パラメータのどちらでコールするかによって異なります。

```
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;
   PROCEDURE initialize (tab OUT DateTabTyp, n INTEGER) IS
   BEGIN
      ...
   END;
   PROCEDURE initialize (tab OUT RealTabTyp, n INTEGER) IS
   BEGIN
      ...
   END;
BEGIN
   indx := 50;
   initialize(hiredate_tab, indx);  -- calls first version
   initialize(comm_tab, indx);      -- calls second version
   ...
END;
```

オーバーロードの制限

オーバーロードできるのは、ローカル・サブプログラム、パッケージ・サブプログラムまたは型の順序のみです。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・モードのみの場合は、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。

```
DECLARE
   ...
   PROCEDURE reconcile (acct_no IN INTEGER) IS
   BEGIN ... END;
   PROCEDURE reconcile (acct_no OUT INTEGER) IS
   BEGIN ... END;
```

サブプログラムの仮パラメータの違いがデータ型のみの場合、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。

```
DECLARE
   ...
   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のみで、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、ベース型 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（戻り値のデータ型）のみが異なる 2 つのファンクションは、その型のファミリが異なっている場合でも、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。

```
DECLARE
   ...
   FUNCTION acct_ok (acct_id INTEGER) RETURN BOOLEAN IS
   BEGIN ... END;
   FUNCTION acct_ok (acct_id INTEGER) RETURN INTEGER IS
   BEGIN ... END;
```

サブプログラムのコールの解決方法

図 8-1 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。

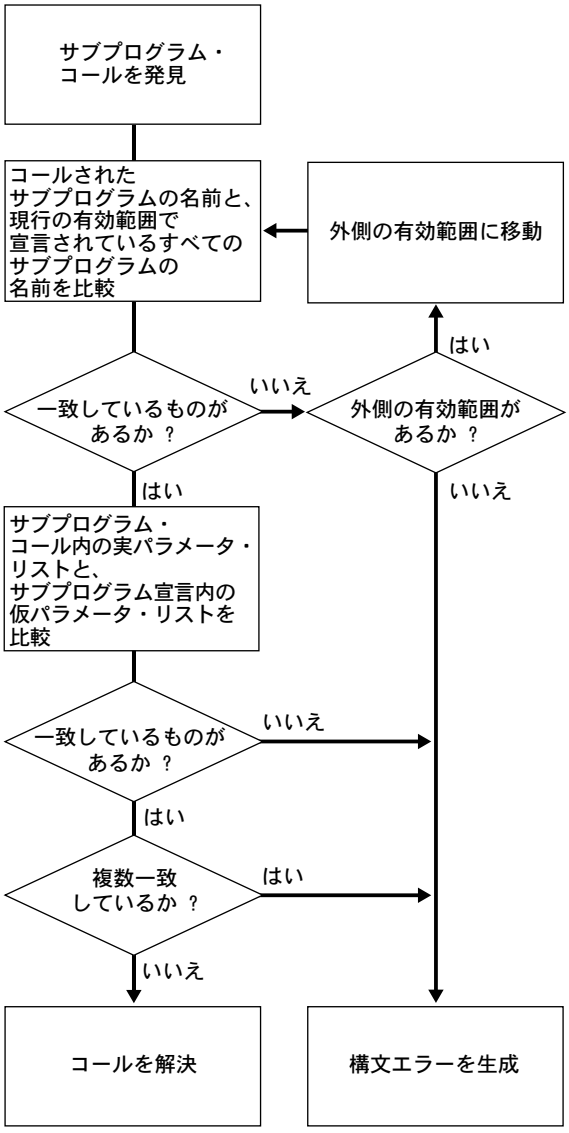
同じ有効範囲のレベルに同じような名前のサブプログラムが存在する場合は、コールを解決するために、コンパイラは実パラメータと仮パラメータが正確に一致するものを発見する必要があります。つまり、(いくつかの仮パラメータにデフォルト値が代入されている場合を除き）パラメータの数、順序およびデータ型が一致している必要があります。一致するものが見つからなかった場合、または一致するものが複数見つかった場合、コンパイラは意味エラーを生成します。

次の例では、ファンクション reconcile 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 (n1 NUMBER, n2 NUMBER) IS
   num1 NUMBER;
   num2 NUMBER;
   FUNCTION balance (...) RETURN REAL IS
      PROCEDURE swap (d1 DATE, d2 DATE) IS
```

```
      BEGIN
         ...
      END;
      PROCEDURE swap (b1 BOOLEAN, b2 BOOLEAN) IS
      BEGIN
         ...
      END;
   BEGIN
      ...
      swap(num1, num2);
      RETURN ...
   END balance;
BEGIN
   ...
END;
```

図 8-1 PL/SQL コンパイラによるコールの解決方法



オーバーロードと継承の相互作用

オーバーロード・アルゴリズムでは、スーパータイプである仮パラメータのかわりにサブタイプの値を使用できます。この機能は**代替性**と呼ばれます。オーバーロードされるプロシージャの複数のインスタンスがプロシージャ・コールと一致する場合は、どのプロシージャがコールされるかを判断するために次の規則が適用されます。

オーバーロードされる各プロシージャのシグネチャの違いが、一部のパラメータが同じスーパータイプ / サブタイプ階層のオブジェクト型であるということのみの場合は、最大一致のものが使用されます。最大一致とは、すべてのパラメータがオーバーロードされる他のインスタンスと少なくとも同程度に近く、少なくとも 1 つのパラメータが近い場合を指します。この近さは、サブタイプとスーパータイプ間の継承の深さによって決まります。

オーバーロードされる 2 つのインスタンスが一致し、オーバーロードされる一方のプロシージャ内の一部の引数の型が他のインスタンス内よりも実引数に近いと、意味エラーが発生します。

また、一部のパラメータの位置がオブジェクト型階層内とは異なり、他のパラメータのデータ型が異なる場合も、暗黙的な変換が必要になるため、意味エラーが発生します。

たとえば、3 つのレベルを持つ型の階層を作成するとします。

```
CREATE TYPE super_t AS object
  (n NUMBER) NOT final;
CREATE OR replace TYPE sub_t under super_t
  (n2 NUMBER) NOT final;
CREATE OR replace TYPE final_t under sub_t
  (n3 NUMBER);
```

あるファンクションについて、オーバーロードされるインスタンスを 2 つ宣言します。両者の引数の型の違いは、この型階層内での位置のみです。

```
CREATE PACKAGE p IS
   FUNCTION foo (arg super_t) RETURN NUMBER;
   FUNCTION foo (arg sub_t) RETURN NUMBER;
END;
/
CREATE PACKAGE BODY p IS
   FUNCTION foo (arg super_t) RETURN NUMBER IS BEGIN RETURN 1; END;
   FUNCTION foo (arg sub_t) RETURN NUMBER  IS BEGIN RETURN 2; END;
END;
/
```

final_t 型の変数を宣言してから、オーバーロードされるファンクションをコールします。ファンクションのインスタンスのうち実行されるのは、sub_t パラメータを受け入れるインスタンスです。これは、その型が階層内で super_t よりも final_t に近いためです。

```
SQL> set serveroutput on
SQL> declare
```

```
v final_t := final_t(1,2,3);
begin
  dbms_output.put_line(p.foo(v));
end;
/
2
```

前述の例では、コールするインスタンスはコンパイル時に選択されます。次の例では、この選択が動的に行われます。

```
CREATE TYPE super_t2 AS object
  (n NUMBER, MEMBER FUNCTION foo RETURN NUMBER) NOT final;
/
CREATE TYPE BODY super_t2 AS
 MEMBER FUNCTION foo RETURN NUMBER IS BEGIN RETURN 1; END; END;
/
CREATE OR replace TYPE sub_t2 under super_t2
  (n2 NUMBER,
   OVERRIDING MEMBER FUNCTION foo RETURN NUMBER) NOT final;
/
CREATE TYPE BODY sub_t2 AS
 OVERRIDING MEMBER FUNCTION foo RETURN NUMBER IS BEGIN RETURN 2; END;
END;
/
CREATE OR replace TYPE final_t2 under sub_t2
  (n3 NUMBER);
/
```

v を super_t2 のインスタンスとして宣言しますが、そこに sub_t2 の値を代入するために、ファンクションの適切なインスタンスがコールされます。この機能は**動的ディスパッチ**と呼ばれます。

```
SQL> set serveroutput on

declare
  v super_t2 := final_t2(1,2,3);
begin
  dbms_output.put_line(v.foo);
end;
/
2
```

表関数を使用した複数行の受入れと戻し

この項では、表関数と、表関数でよく使用される汎用データ型 ANYTYPE、ANYDATA および ANYDATASET について説明します。

主な項目は、次のとおりです。

表関数の概要

表関数は、物理データベース表と同様に問合せできるように、または PL/SQL コレクション変数に代入できるように、行のコレクション（ネストした表または VARRAY）を生成する関数です。表関数は、問合せの FROM 句にあるデータベース表の名前のように、または問合せの SELECT リストにある列名のように使用できます。

表関数は、入力として行のコレクションを使用することができます。入力コレクション・パラメータには、コレクション型（VARRAY や PL/SQL 表など）または REF CURSOR を使用できます。

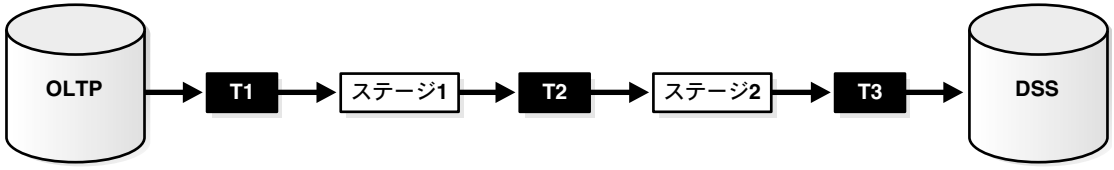
表関数の実行はパラレル化でき、戻される行は中間のステージングなしで次のプロセスに直接送ることができます。表関数から戻されるコレクションの行は、**パイプライン化**することもできます。つまり、表関数の入力の処理がすべて完了してからバッチで戻されるのではなく、生成された時点で反復的に戻されます。

表関数のストリーム、パイプラインおよびパラレル実行により、パフォーマンスを改善できます。

- マルチスレッドが可能になり、表関数を同時に実行します。
- プロセス間の中間的なステージングを排除します。
- 問合せの応答時間が短縮されます。パイプラインでない表関数の場合は、表関数から戻されるコレクション全体を組み立てて、サーバーに戻してからでなければ、問合せでは 1 つの結果行を戻すことができません。パイプライン化により、行を生成時に反復的に戻すことができます。これにより、オブジェクト・キャッシュではコレクション全体をマテリアライズする必要がないため、表関数のメモリー所要量も減少します。
- コレクション全体が表やメモリーにステージングされるまで待ってから、コレクション全体を戻すのではなく、各行の生成時に表関数から戻されるコレクションの結果行を反復的に提供します。

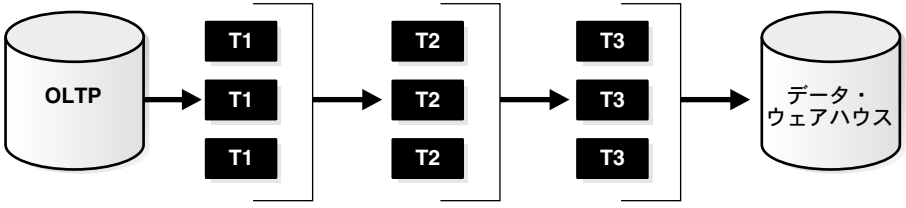
図 8-2 に、データが表関数により実装される複数（この場合は 3 つ）の変換を通過し、最後にデータベースにロードされる、典型的な処理の使用例を示します。この使用例では、表関数はパラレル化されておらず、各変換の後に、一時表を使用して結果のコレクション全体をステージングする必要があります。

図 8-2 パラレル化もパイプライン化もされていない表関数による典型的なデータ処理



これに対して、次の図 8-3 は、ストリームとパラレル実行により同じ使用例をどのように簡素化できるかを示しています。

図 8-3 パイプライン化とパラレル実行を使用したデータ処理



表関数

表関数は、表の各行を表すコレクション型のインスタンスを戻します。この表関数は、問合せの FROM 句内のキーワード TABLE で囲まれた関数をコールすることによって、表と同様に問合せを実行できます。また、問合せの SELECT リストにある関数をコールすることで、PL/SQL コレクション変数に代入できます。

例 : 表関数の問合せ

次の例の表関数 GetBooks は、入力として CLOB を取り、コレクション型 BookSet_t のインスタンスを戻します。CLOB 列には、なんらかの（独自または次のように XML などの標準）形式で書籍のカタログ・リストが格納されています。この表関数は、すべてのカタログとそれに対応する書籍リストを戻します。

コレクション型 BookSet_t の定義は、次のとおりです。

```
CREATE TYPE Book_t AS OBJECT
( name VARCHAR2(100),
  author VARCHAR2(30),
  abstract VARCHAR2(1000));

CREATE TYPE BookSet_t AS TABLE OF Book_t;
```

CLOB は、表 Catalogs に格納されています。

```
CREATE TABLE Catalogs
( name VARCHAR2(30),
  cat CLOB);
```

関数 GetBooks の定義は、次のとおりです。

```
CREATE FUNCTION GetBooks(a CLOB) RETURN BookSet_t;
```

次の問合せは、すべてのカタログとそれに対応する書籍リストを戻します。

```
SELECT c.name, Book.name, Book.author, Book.abstract
  FROM Catalogs c, TABLE(GetBooks(c.cat)) Book;
```

例 : 表関数の結果の代入

次の例は、表関数の結果を PL/SQL コレクション変数に代入する方法を示しています。表関数は問合せの SELECT リストからコールされるため、TABLE キーワードは不要です。

```
create type numset_t as table of number;
/

create function f1(x number) return numset_t pipelined is
begin
  for i in 1..x loop
    pipe row(i);
  end loop;
  return;
end;
/

-- pipelined function in from clause
select * from table(f1(3));

COLUMN_VALUE
------------
           1
           2
           3

3 rows selected.

-- pipelined function in select list

select f1(3) from dual;

F1(3)
---------------------------------
NUMSET_T(1, 2, 3)

-- Since the function returns a collection, we can assign
-- the result to a PL/SQL variable.
declare
  func_result numset_t;
begin
  select f1(3) into func_result from dual;
end;
/
```

パイプライン表関数

データが**パイプライン化**されたと呼ばれるのは、次の変換への入力になる前に表やキャッシュにステージングされず、生成側（変換）で生成された直後にコンシューマ（変換）で使用される場合です。

パイプライン化により、表関数は行をすばやく戻すことができ、表関数の結果をキャッシュするためのメモリーを削減できます。

パイプライン表関数は、その結果のコレクションをサブセットで戻すことができます。戻されたコレクションは、必要に応じてフェッチできるストリームのように動作します。このため、表関数を仮想表のように使用できるようになります。

パイプライン表関数を実装するには、次の 2 つの方法があります。

- **システム固有の PL/SQL アプローチ :** コンシューマと生成側は別個の実行スレッド（同じまたは異なるプロセス・コンテキスト内）で動作し、パイプまたはキューイング・メカニズムを通じてやりとりできます。このアプローチは、コルーチンの実行に似ています。
- **インタフェースによるアプローチ :** コンシューマと生成側は同じ実行スレッドで動作します。生成側は、結果セットを生成後に制御をコンシューマに明示的に戻します。また、生成側は、コンシューマにより再びコールされた時に前回の状態から再開できるように、現在の状態をキャッシュします。インタフェースによるアプローチの場合は、手続き型言語で適切に定義されたインタフェース・セットを実装する必要があります。インタフェースによるアプローチの詳細は、『Oracle9*i* Data Cartridge Developer's Guide』を参照してください。

この章の残りの部分では、**表関数**という用語はパイプライン表関数を指します。つまり、パイプライン化された反復的な方法でコレクションを戻す表関数です。

変換へのパイプライン表関数の使用

パイプライン表関数には、通常の関数に使用できる引数であれば、すべて使用できます。引数として REF CURSOR を受け入れる表関数は、変換関数として使用できます。つまり、REF CURSOR を使用して入力行をフェッチし、その変換を実行し、結果をパイプラインで（インタフェース・アプローチまたは固有の PL/SQL アプローチを使用して）出力できます。

たとえば、次のコードは、StockPivot 関数を定義する宣言を示しています。この関数は、(Ticker, OpenPrice, ClosePrice) 型の 1 行を (Ticker, PriceType, Price) 形式の 2 行に変換します。行 ("ORCL", 41, 42) について StockPivot をコールすると、("ORCL", "O", 41) および ("ORCL", "C", 42) の 2 行が生成されます。

この表関数の入力データを次の表 StockTable のようなソースから取り込むとします。

```
CREATE TABLE StockTable (
  ticker VARCHAR(4),
  open_price NUMBER,
  close_price NUMBER
);
```

これは宣言です。関数本体については、8-37 ページの「表関数からの結果の戻し」を参照してください。

```
-- Create the types for the table function's output collection
-- and collection elements

CREATE TYPE TickerType AS OBJECT
(
  ticker VARCHAR2(4),
  PriceType VARCHAR2(1),
  price NUMBER
);
/
CREATE TYPE TickerTypeSet AS TABLE OF TickerType;
/
-- Define the ref cursor type

CREATE PACKAGE refcur_pkg IS
  TYPE refcur_t IS REF CURSOR RETURN StockTable%ROWTYPE;
END refcur_pkg;
/

-- Create the table function

CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet
PIPELINED ... ;
/
```

StockPivot 表関数を使用する問合せの例を次に示します。

```
SELECT * FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable)));
```

この問合せでは、パイプライン表関数 StockPivot は CURSOR 副問合せ SELECT * FROM StockTable から行をフェッチし、変換を実行して、結果をパイプラインでユーザーに表として戻します。この関数では、入力行ごとに出力行（コレクション要素）が 2 行ずつ生成されます。

CURSOR 副問合せが前述の例のように SQL から REF CURSOR 関数の引数に渡される場合、関数の実行中には参照先のカーソルがすでに開いています。

パイプライン表関数の記述

パイプライン表関数を宣言するには、PIPELINED キーワードを指定します。このキーワードは、関数が行を反復的に戻すことを示します。パイプライン表関数の戻り型は、ネストした表や VARRAY のようなコレクション型である必要があります。このコレクションはスキーマ・レベルまたはパッケージ内で宣言できます。関数内では、コレクション型の個々の要素を戻します。

たとえば、次のような 2 つのパイプライン表関数の宣言を考えてみます。（関数本体については、後の例を参照。）

```
CREATE FUNCTION GetBooks(cat CLOB) RETURN BookSet_t
  PIPELINED IS ...;

CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet
  PIPELINED IS...;
```

表関数からの結果の戻し

PL/SQL では、PIPE ROW 文によって表関数で行がパイプされ、処理が継続します。この文を使用すると、PL/SQL の表関数で生成直後に行を戻すことができます。（パフォーマンス上、PL/SQL ランタイム・システムでは、行はコンシューマにバッチで与えられます。）次に例を示します。

```
CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet
PIPELINED IS
  out_rec TickerType := TickerType(NULL,NULL,NULL);
  in_rec p%ROWTYPE;
BEGIN
  LOOP
    FETCH p INTO in_rec;
    EXIT WHEN p%NOTFOUND;
    -- first row
    out_rec.ticker := in_rec.Ticker;
    out_rec.PriceType := 'O';
    out_rec.price := in_rec.OpenPrice;
    PIPE ROW(out_rec);
    -- second row
    out_rec.PriceType := 'C';
    out_rec.Price := in_rec.ClosePrice;
    PIPE ROW(out_rec);
  END LOOP;
  CLOSE p;
  RETURN;
END;
/
```

この例で、PIPE ROW(out_rec) 文はデータをパイプラインで PL/SQL 表関数から戻しています。out_rec はレコードで、その型は出力コレクションの要素の型と一致します。

PIPE ROW 文を使用できるのはパイプライン表関数の本体内のみで、他の場所で使用するとエラーが呼び出されます。行を戻さないパイプライン表関数の場合は、PIPE ROW 文を省略できます。

パイプライン表関数には、値を戻さない RETURN 文が必要です。この RETURN 文は、制御をコンシューマに移し、次回のフェッチで NO_DATA_FOUND 例外が確実に呼び出されるようにします。

Oracle には、オブジェクトやコレクション型など、他の SQL 型の型記述、データ・インスタンスおよびデータ・インスタンス・セットを動的にカプセル化してアクセスできるように、3 つの特別なデータ型が用意されています。また、この 3 つの型を使用すると、匿名コレクション型のように**匿名**の（つまり、名前を持たない）型を作成できます。この 3 つの型は、SYS.ANYTYPE、SYS.ANYDATA および SYS.ANYDATASET です。SYS.ANYDATA 型は、表関数からの戻り値として役立つ場合があります。

関連項目： ANYTYPE、ANYDATA および ANYDATASET 型へのインタフェースと、この 3 つの型で使用する DBMS_TYPES パッケージについては、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

PL/SQL 表関数間のデータのパイプライン

シリアル実行では、コルーチン実行に似たアプローチを使用して、結果がある PL/SQL 表関数から別の PL/SQL 表関数へとパイプラインされます。たとえば、次の文では、関数 g から関数 f へと結果がパイプラインされます。

```
SELECT * FROM TABLE(f(CURSOR(SELECT * FROM TABLE(g()))));
```

パラレル実行の場合も同様ですが、各関数は異なるプロセス（またはプロセス・セット）で実行されます。

表関数の問合せ

パイプライン表関数は、SELECT 文の FROM 句で使用されます。結果行は Oracle によって表関数の実装から反復的に取り出されます。次に例を示します。

```
SELECT x.Ticker, x.Price
FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable))) x
WHERE x.PriceType='C';
```

注意： 表関数はコレクションを戻します。トップレベルの問合せで SELECT * が使用され、その問合せが PL/SQL 変数またはバインド変数を参照するような場合は、正確な戻り型を指定するために、表変数の周囲に CAST 演算子が必要なことがあります。

表関数に対する複数コールの最適化

表関数を同じ問合せまたは別の問合せで複数回コールすると、基礎となる実装が複数回実行されます。デフォルトでは、行のバッファリングや再利用は行われません。

次に例を示します。

```
SELECT * FROM TABLE(f(...)) t1, TABLE(f(...)) t2
  WHERE t1.id = t2.id;

SELECT * FROM TABLE(f());

SELECT * FROM TABLE(f());
```

ただし、表関数の出力が引数として渡された値でのみ決定される場合、関数は渡された値の組合せごとに常にまったく同じ結果を生成します。この場合は、関数 DETERMINISTIC を宣言すると、行が自動的にバッファリングされます。ただし、DETERMINISTIC でマークされている関数が実際に DETERMINISTIC であるかどうかをデータベースが認識する方法はなく、実際には異なっている場合は予測できない結果になるため注意してください。

表関数の結果からのフェッチ

PL/SQL の CURSOR と REF CURSOR は、表関数に対する問合せ用に定義できます。次に例を示します。

```
OPEN c FOR SELECT * FROM TABLE(f(...));
```

表関数のカーソルと通常のカーソルでは、フェッチの意味は同じです。表関数に基づく REF CURSOR の代入に特別な意味はありません。

ただし、SQL オプティマイザでは、PL/SQL 文にまたがる最適化は行われません。次に例を示します。

```
BEGIN
    OPEN r FOR SELECT * FROM TABLE(f(CURSOR(SELECT * FROM tab)));
    SELECT * BULK COLLECT INTO rec_tab FROM TABLE(g(r));
END;
```

前述の例は、次の例と同様には実行されません。

```
SELECT * FROM TABLE(g(CURSOR(SELECT * FROM
  TABLE(f(CURSOR(SELECT * FROM tab))))));
```

このため、2 つの SQL 文の実行に関連したオーバーヘッドは無視され、2 つの文の間で結果をパイプラインできるとみなされます。

カーソル変数によるデータの受渡し

PL/SQL 関数に REF CURSOR パラメータで行セットを渡すことができます。たとえば、この関数が事前定義された弱い型指定の REF CURSOR を持つ SYS_REFCURSOR 型の引数を受け入れるように宣言されているとします。

```
FUNCTION f(p1 IN SYS_REFCURSOR) RETURN ... ;
```

副問合せの結果を関数に直接渡すことができます。

```
SELECT * FROM TABLE(f(CURSOR(SELECT empno FROM tab)));
```

この例では、副問合せの結果を REF CURSOR パラメータとして渡す必要があることを示すために、CURSOR キーワードが必要です。

事前定義の弱い REF CURSOR 型の SYS_REFCURSOR もサポートされます。SYS_REFCURSOR を使用すると、パッケージ内で REF CURSOR 型を使用前に作成する必要はありません。

強い REF CURSOR 型を使用する場合は、PL/SQL パッケージを作成し、その中で宣言する必要があります。また、強い REF CURSOR 型を表関数の引数として使用する場合は、REF CURSOR 引数の実際の型が列の型と一致する必要があります。一致しない場合は、エラーが生成されます。表関数の弱い REF CURSOR 引数をパーティション化するには、PARTITION BY ANY 句を使用する必要があります。弱い REF CURSOR 引数には、レンジ・パーティション化もハッシュ・パーティション化も使用できません。

例 : 複数の REF CURSOR 入力変数の使用

PL/SQL 関数は、複数の REF CURSOR 入力変数を受け入れることができます。

```
CREATE FUNCTION g(p1 pkg.refcur_t1, p2 pkg.refcur_t2) RETURN...
  PIPELINED ... ;
```

関数 g は、次のようにコールできます。

```
SELECT * FROM TABLE(g(CURSOR(SELECT empno FROM tab),
  CURSOR(SELECT * FROM emp));
```

戻されたデータにまたがって反復する REF CURSOR を作成すると、表関数の戻り値を他の表関数に渡すことができます。

```
SELECT * FROM TABLE(f(CURSOR(SELECT * FROM TABLE(g(...)))));
```

例 : 問合せに対する REF CURSOR の明示的オープン

問合せに対して REF CURSOR を明示的にオープンし、それを表関数にパラメータとして渡すことができます。

```
BEGIN
  OPEN r FOR SELECT * FROM TABLE(f(...));
  -- Must return a single row result set.
  SELECT * INTO rec FROM TABLE(g(r));
END;
```

この場合、表関数は完了時にカーソルをクローズするため、プログラムではカーソルを明示的にクローズしないようにする必要があります。

例 : 集計関数としてのパイプライン表関数の使用

表関数では、入力 REF CURSOR を使用して集計結果を計算できます。次の例では、一連の入力行を反復することで、加重平均を計算しています。

```
DROP TABLE gradereport;
CREATE TABLE gradereport (student VARCHAR2(30), subject VARCHAR2(30), weight NUMBER,
grade NUMBER);

INSERT INTO gradereport  VALUES('Mark', 'Physics', 4, 4);
INSERT INTO  gradereport VALUES('Mark','Chemistry', 4,3);
INSERT INTO gradereport VALUES('Mark','Maths', 3,3);
INSERT INTO gradereport VALUES('Mark','Economics', 3,4);

CREATE OR replace TYPE gpa AS TABLE OF NUMBER;
/

CREATE OR replace FUNCTION weighted_average(input_values sys_refcursor)
RETURN gpa PIPELINED IS
  grade NUMBER;
  total NUMBER := 0;
  total_weight NUMBER := 0;
  weight NUMBER := 0;
BEGIN
-- The function accepts a ref cursor and loops through all the input rows.
  LOOP
     FETCH input_values INTO weight, grade;
     EXIT WHEN input_values%NOTFOUND;
-- Accumulate the weighted average.
     total_weight := total_weight + weight;
     total := total + grade*weight;
  END LOOP;
  PIPE ROW (total / total_weight);
-- The function returns a single result.
```

```
  RETURN;
END;
/
show errors;

-- The result comes back as a nested table with a single row.
-- COLUMN_VALUE is a keyword that returns the contents of a nested table.
select weighted_result.column_value from table(weighted_average(cursor(select
weight,grade from gradereport))) weighted_result;

COLUMN_VALUE
------------
       3.5
```

表関数内での DML 操作の実行

DML 文を実行するには、自律型トランザクション・プラグマで表関数を宣言する必要があります。このプラグマにより、関数は他のプロセスと共有されない自律型トランザクションで実行されます。

次の構文を使用して、表関数を自律型トランザクション・プラグマで宣言します。

```
CREATE FUNCTION f(p SYS_REFCURSOR) return CollType PIPELINED IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN ... END;
```

パラレル実行中に、表関数の各インスタンスが独立したトランザクションを作成します。

表関数への DML 操作の実行

表関数を UPDATE、INSERT または DELETE 文のターゲット表にすることはできません。たとえば、次の文ではエラーが呼び出されます。

```
UPDATE F(CURSOR(SELECT * FROM tab)) SET col = value;
INSERT INTO f(...) VALUES ('any', 'thing');
```

ただし、表関数のビューを作成し、INSTEAD OF トリガーを使用して更新できます。次に例を示します。

```
CREATE VIEW BookTable AS
  SELECT x.Name, x.Author
  FROM TABLE(GetBooks('data.txt')) x;
```

次の INSTEAD OF トリガーは、ユーザーが BookTable ビューに行を挿入すると起動します。

```
CREATE TRIGGER BookTable_insert
INSTEAD OF INSERT ON BookTable
REFERENCING NEW AS n
FOR EACH ROW
BEGIN
  ...
END;
INSERT INTO BookTable VALUES (...);
```

INSTEAD OF トリガーは、表関数に作成されたビューでのすべての DML 操作について定義できます。

表関数の例外処理

表関数の例外処理は、通常のユーザー定義関数の場合と同じです。

C や Java など、一部の言語には、ユーザー指定の例外処理のためのメカニズムが用意されています。表関数内で呼び出された例外が処理される場合に、表関数は例外ハンドラを実行して処理を継続します。例外ハンドラを終了すると、制御が外側の有効範囲に移ります。例外が解消されると、実行は通常どおり進行します。

表関数に未処理の例外があると、親トランザクションがロールバックされます。

表関数のパラレル化

パラレルに実行される表関数には、パーティション化された入力パラメータが必要です。表関数のパラレル化がオンになるのは、次の条件が満たされている場合のみです。

- 関数の宣言に PARALLEL_ENABLE 句がある場合
- PARTITION BY 句で REF CURSOR 引数が 1 つのみ指定されている場合

  PARALLEL_ENABLE 句の一部として入力の REF CURSOR に PARTITION BY 句が指定されていなければ、SQL コンパイラはデータを適切にパーティション化する方法を判断できません。PARTITION BY ANY 句を使用しない場合、PARTITION BY 句で指定できるのは強い型指定を持つ REF CURSOR 引数のみです。

表関数のパラレル実行

SELECT リストに指定されている関数のパラレル実行では、関数の実行はプッシュ・ダウンされ、複数のスレーブ・**スキャン**・プロセスにより実行されます。これらは、それぞれ関数の入力データの 1 セグメントに対して関数を実行します。

たとえば、

```
SELECT f(col1) FROM tab;
```

この問合せは、f が純粋な関数の場合にパラレル化されます。ひとつのスレーブ・スキャン・プロセスでは、次のような SQL が実行されます。

```
SELECT f(col1) FROM tab WHERE ROWID BETWEEN :b1 AND :b2;
```

各スレーブ・スキャンは ROWID の範囲を操作し、範囲内の各行に関数 f を適用します。次に、関数 f がスキャン・プロセスにより実行されます。独立して実行されることはありません。

SELECT リストに指定されている関数とは異なり、表関数は FROM 句の中でコールされ、コレクションを戻します。パーティション化アプローチは表関数で実行される操作に適したものである必要があるため、これは表関数の入力データをスレーブ・スキャン・プロセス間でパーティション化する方法に影響します。(たとえば、ORDER BY 操作の場合は入力をレンジ・パーティション化する必要がありますが、GROUP BY 操作の場合は入力をハッシュ・パーティション化する必要があります。)

表関数自体は、適切なパーティション化アプローチを宣言内で指定します。(8-45 ページの「入力データのパーティション化」を参照。)次に、関数が 2 段階の操作で実行されます。最初に、1 組目のスレーブ・プロセスにより、データが関数の宣言で指定されたとおりにパーティション化されます。次に、2 組目のスレーブ・プロセスにより表関数がパーティション化されたデータに対してパラレルに実行されます。

たとえば、次の問合せの表関数には、REF CURSOR パラメータがあります。

```
SELECT * FROM TABLE(f(CURSOR(SELECT * FROM tab)));
```

スキャンは 1 組目のスレーブ・プロセスで実行され、行は（関数宣言で指定したパーティション化方法に基づいて）実際に関数 f をパラレルに実行する 2 組目のスレーブ・プロセスに再配布されます。

入力データのパーティション化

表関数の宣言で指定できるのは、1 つの REF CURSOR パラメータのデータのパーティション化のみです。そのための構文を次に示します。

```
CREATE FUNCTION f(p ref cursor type) RETURN rec_tab_type PIPELINED
  PARALLEL_ENABLE(PARTITION p BY [{HASH | RANGE} (column list) | ANY ]) IS
BEGIN ... END;
```

PARALLEL_ENABLE 句の PARTITION…BY 句では、入力カーソルのうちパーティション化するものと、パーティション化に使用する列を指定します。

列のリストで明示的な列名が指定されていれば、パーティション化方法として RANGE または HASH を使用できます。入力行は、指定した列でハッシュ・パーティション化またはレンジ・パーティション化されます。

ANY キーワードは、関数の動作が入力データのパーティション化から独立していることを示します。このキーワードを使用すると、ランタイム・システムではスレーブ間でデータがランダムにパーティション化されます。このキーワードは、1 行を取得してその列を操作し、この行の列のみに基づいて出力行を生成する関数に適しています。

たとえば、次に示すピボットのような StockPivot 関数は、入力行として次の型の 1 行を取得し、

```
(Ticker varchar(4), OpenPrice number, ClosePrice number)
```

次の型の行を生成します。

```
(Ticker varchar(4), PriceType varchar(1), Price number).
```

したがって、行 ("ORCL", 41, 42) では ("ORCL", "O", 41) および ("ORCL", "C", 42) の 2 行が生成されます。

```
CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN rec_tab_type PIPELINED
    PARALLEL_ENABLE(PARTITION p BY ANY) IS
  ret_rec rec_type;
BEGIN
  FOR rec IN p LOOP
    ret_rec.Ticker := rec.Ticker;
    ret_rec.PriceType := "O";
    ret_rec.Price := rec.OpenPrice;
    PIPE ROW(ret_rec);

    ret_rec.Ticker := rec.Ticker;   -- Redundant; not required
    ret_rec.PriceType := "C";
```

```
    ret_rec.Price := rec.ClosePrice;
    push ret_rec;
  END LOOP;
  RETURN;
END;
```

関数 f を使用すると、次の方法で Stocks 表から別の表を生成できます。

```
INSERT INTO AlternateStockTable
  SELECT * FROM
  TABLE(StockPivot(CURSOR(SELECT * FROM StockTable)));
```

StockTable がパラレルにスキャンされ、OpenPrice でパーティション化されると、関数 StockPivot は StockTable のスキャンを実行するデータ・フロー演算子と結合され、同じパーティション化を示します。

これに対して、StockTable がパーティション化されず、スキャンがパラレルに実行されなければ、AlternateStockTable への挿入も逐次実行されます。やや複雑な例を次に示します。

```
INSERT INTO AlternateStockTable
  SELECT *
    FROM TABLE(f(CURSOR(SELECT * FROM Stocks))),
         TABLE(g(CURSOR( ... )))
    WHERE join_condition;
```

g の定義は次のとおりです。

```
CREATE FUNCTION g(p refcur_pkg.refcur_t) RETURN ... PIPELINED
  PARALLEL_ENABLE (PARTITION p BY ANY)
BEGIN ... END;
```

関数 g がパラレルに実行され、ANY でパーティション化される場合、パラレル INSERT は g と同じデータ・フロー操作に属すことができます。

キーワード ANY が指定されると、データはスレーブ間でランダムにパーティション化されます。このため、関数は入力パラメータに関連したスキャンを実行する同じスレーブ・セットで実行されることになります。

この場合、データの再配布や再パーティション化は不要です。カーソル p 自体がパラレル化されていない場合、受け取るデータは列リストにある列でランダムにパーティション化されます。このパーティション化には、ラウンドロビン法によるテーブル・キューが使用されます。

リーフ・レベル表関数のパラレル実行

複数行を入力として受け入れる必要がないため、REF CURSOR を必要としないが、複数行を生成する関数でパラレル実行を使用するには、REF CURSOR が必要になるように配置します。これにより、関数では処理がなんらかの方法でパーティション化されます。

たとえば、外部ファイル・セットをパラレルに読み込んで、中のレコードを戻す関数が必要であるとします。REF CURSOR の処理を与えるために、最初に表を作成してから、ファイル名を移入できます。この表の REF CURSOR を表関数にパラメータとして渡すことができます（readfiles）。この方法を次のコードに示します。

```
CREATE TABLE filetab(filename VARCHAR(20));

INSERT INTO filetab VALUES('file0');
INSERT INTO filetab VALUES('file1');
...
INSERT INTO filetab VALUES('fileN');

SELECT * FROM
    TABLE(readfiles(CURSOR(SELECT filename FROM filetab)));

CREATE FUNCTION readfiles(p pkg.rc_t) RETURN coll_type
  PARALLEL_ENABLE(PARTITION p BY ANY) IS
  ret_rec rec_type;
BEGIN
  FOR rec IN p LOOP
    done := FALSE;
    WHILE (done = FALSE) LOOP
         done := readfilerecord(rec.filename, ret_rec);
         PIPE ROW(ret_rec);
    END LOOP;
  END LOOP;
  RETURN;
END;
```

表関数による入力データのストリーム

表関数がカーソル引数からフェッチする行を順序付けまたはクラスタ化する方法は、**データのストリーム化**と呼ばれます。関数は、入力データを次の方法でストリーム化します。

- 入力される行の順序付けに制限を課しません。
- 特定のキー列または列で順序付けします。
- 特定のキーでクラスタ化します。

クラスタ化により、同じキー値を持つ行がまとめられるように見えますが、それ以外に、行の順序付けは行われません。

関数の定義時には、ORDER BY 句または CLUSTER BY 句を使用して、入力ストリームの動作を制御します。

入力ストリームは、関数の順次実行またはパラレル実行について指定できます。

ORDER BY または CLUSTER BY 句を指定しなければ、行は（ランダムな）順番に入力されます。

注意： パラレル実行の ORDER BY のセマンティクスは、SQL 文の ORDER BY 句のセマンティクスとは異なる意味を持ちます。SQL 文の ORDER BY 句は、データ・セット全体をグローバルに順序付けします。表関数の ORDER BY 句は、スレーブで実行中の表関数の各インスタンスに対してローカルに、それぞれの行を順序付けします。

入力ストリームを順序付けする構文の例を次に示します。この例で、関数 f は種類が (Region, Sales) の行を取り、各地域の平均売上高を示す (Region, AvgSales) 形式の行を戻します。

```
CREATE FUNCTION f(p ref_cursor_type) RETURN tab_rec_type PIPELINED
  CLUSTER p BY Region
  PARALLEL_ENABLE(PARTITION p BY Region) IS
  ret_rec rec_type;
  cnt number;
  sum number;
BEGIN
  FOR rec IN p LOOP
    IF (first rec in the group) THEN
        cnt := 1;
        sum := rec.Sales;
    ELSIF (last rec in the group) THEN
      IF (cnt <> 0) THEN
        ret_rec.Region := rec.Region;
          ret_rec.AvgSales := sum/cnt;
          PIPE ROW(ret_rec);
```

```
        END IF;
    ELSE
       cnt := cnt + 1;
      sum := sum + rec.Sales;
    END IF;
  END LOOP;
  RETURN;
END;
```

パラレル実行のためのパーティション化またはクラスタ化の選択

パーティション化とクラスタ化は混同されやすいですが、両者の動作は異なります。たとえば、パラレル実行では、クラスタ化しなくてもパーティション化で十分な場合があります。

各 department_id の給与のメモリー内集計を実行する関数 SmallAggr を考えてみます。department_id は 1、2、3 のいずれかです。この関数への入力行は、department_id が 1 のすべての行があるスレーブに送られ、department_id が 2 のすべての行が別のスレーブに送られるように、department_id で HASH パーティション化できます。

この関数内で集計を実行するために入力行を department_id でクラスタ化する必要はありません。各スレーブは、1 × 3 の配列 SmallSum[1..3] を持つことができ、その配列内で各 department_id の総計がメモリー内の SmallSum[department_id] に追加されます。これに対して、department_id の一意の値の個数が極端に大きい場合は、クラスタ化を使用して部門の集計を計算し、それを一度にディスクの 1 つの department_id に書き込む必要があります。

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、現行ユーザーの権限ではなく所有者の権限で実行します。このような定義者権限サブプログラム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 がスキーマ scott と 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 (
   my_deptno NUMBER,
   my_dname  VARCHAR2,
   my_loc    VARCHAR2) AS
BEGIN
   INSERT INTO dept VALUES (my_deptno, my_dname, my_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。したがって、ユーザー blake がプロシージャをコールした場合でも、スキーマ scott 内の dept 表は更新されます。

あるスキーマのサブプログラムが、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かを考えてみます。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。回避策として、スキーマ名を変数としてSQL*Plusで定義できます。

もう1つの方法は、サブプログラム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。

そこで、AUTHID 句を使用することをお薦めします。これによって、ストアド・プロシージャおよびSQLメソッドを現行ユーザーの権限とスキーマ・コンテキストで実行できます。

このような実行者権限サブプログラムは、特定のスキーマにバインドされません。様々なユーザーが実行できます。たとえば、次に示すプロシージャ create_dept は、その現行ユーザーの権限で実行され、そのユーザーのスキーマ内の dept 表に行を挿入します。

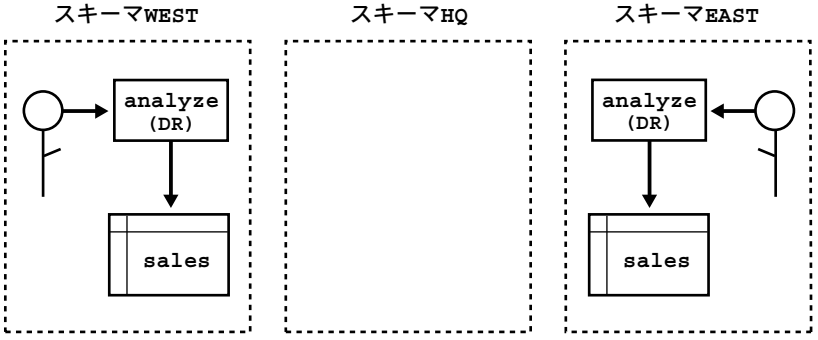
```
CREATE PROCEDURE create_dept (
   my_deptno NUMBER,
   my_dname  VARCHAR2,
   my_loc    VARCHAR2) AUTHID CURRENT_USER AS
BEGIN
   INSERT INTO dept VALUES (my_deptno, my_dname, my_loc);
END;
```

実行者権限の利点

実行者権限サブプログラムを使用すると、コードを再利用し、アプリケーション・ロジックを集中化できます。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。

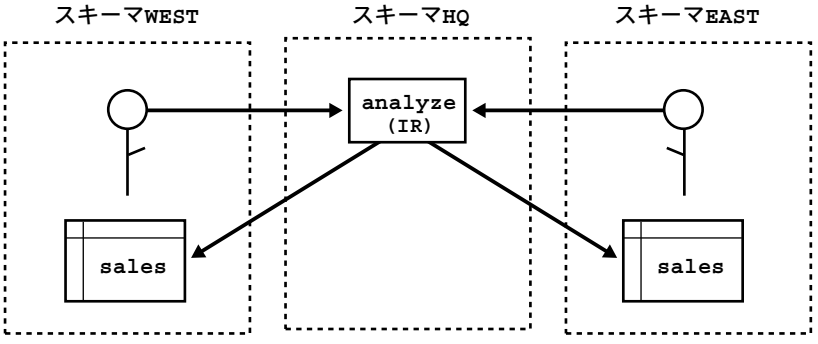
定義者権限（DR）プロシージャを使用して売上を分析する会社の場合を考えてみます。ローカルの売上統計を出すには、プロシージャ analyze は各地のサイトに常駐する sales 表にアクセスする必要があります。このため、図 8-4 に示すように、プロシージャも各地のサイトに常駐する必要があります。これはメンテナンスの問題を引き起こします。

図 8-4 定義者権限の問題 : 同じプロシージャの複数のコピー



この問題を解決するには、プロシージャ analyze の実行者権限（IR）バージョンを本社にインストールします。こうすると、図 8-5 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 表を問い合せることができます。

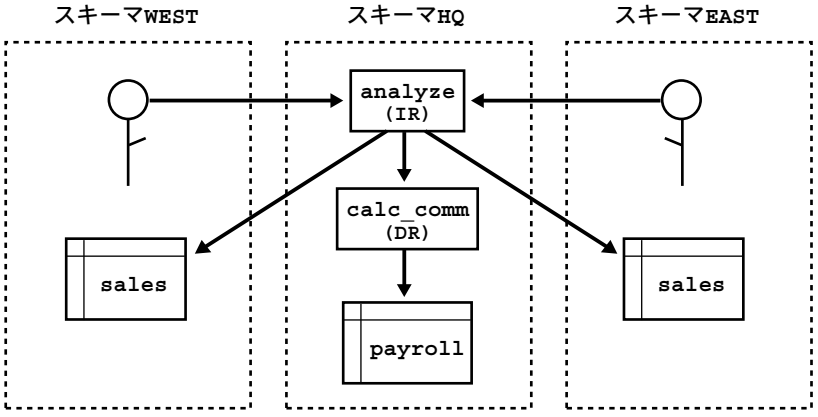
図 8-5 実行者権限による解決 : 複数のスキーマを操作する単一のプロシージャ



機密データへのアクセスを制限するには、実行者権限サブプログラムに定義者権限サブプログラムをコールさせます。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 表を更新するとします。

この場合、analyze の現行ユーザーには、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必要はないため、問題が発生します。図 8-6 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 8-6 実行者権限サブプログラムへの制御されたアクセス



AUTHID 句によるサブプログラムの権限の指定

実行者権限を実装するには、AUTHID 句を使用して、サブプログラムを所有者と現行ユーザー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（サブプログラムの外側のオブジェクトへの参照）が所有者と現行ユーザー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のみ使用できます。次にヘッダーの構文を示します。

```
-- standalone function
CREATE [OR REPLACE] FUNCTION [schema_name.]function_name
[(parameter_list)] RETURN datatype
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}

-- standalone 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 句はすべてのサブプログラムに適用されます。

注意 : 提供されている PL/SQL パッケージの大部分（DBMS_LOB、DBMS_PIPE、DBMS_ROWID、DBMS_SQL および UTL_REF など）は、実行者権限パッケージです。

サブプログラム実行中の現行ユーザー

一連のコールで実行者権限サブプログラムが最初にコールされた場合、現行ユーザーとはセッション・ユーザーを指します。これは定義者権限サブプログラムがコールされるまで変わりません。定義者権限サブプログラムがコールされた場合は、そのサブプログラムの所有者が現行ユーザーになります。定義者権限サブプログラムが任意の実行者権限サブプログラムをコールした場合は、所有者の権限によって実行されます。定義者権限サブプログラムが終了した場合は、直前の現行ユーザーに制御が復帰します。

実行者権限サブプログラムでの外部参照の解決

AUTHID CURRENT_USER を指定すると、実行時に現行ユーザーの権限がチェックされ、外部参照は現行ユーザー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にのみ適用されます。

- SELECT、INSERT、UPDATE および DELETE DML 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、コンパイル時に所有者の権限がチェックされ、外部参照は所有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 balance を参照します。この外部参照はプロシージャ reconcile の所有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_ops.balance(acct_id);
```

```
   ...
END;
```

実行者権限サブプログラムでのテンプレート・オブジェクトの必要性

実行者権限サブプログラムの外部参照は、実行時に現行ユーザー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。そのため所有者は、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する必要があります。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ている必要があります。そうではない場合は、エラー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、現行ユーザー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。

実行者権限サブプログラムでのデフォルトの名前解決のオーバーライド

デフォルトの実行者定義の動作変更が必要な場合があります。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ことに注意してください。この外部参照は、通常は現行ユーザー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;
```

所有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;
```

現行ユーザーが 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できます。

```
BEGIN
   SELECT scott.median(sal) INTO mid_sal FROM emp;
   ...
END;
```

現行ユーザーが、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。

実行者権限サブプログラムに対する権限の付与

サブプログラムを直接コールするには、ユーザーはそのサブプログラムに対して 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で、ユーザーに次のことを許可します。

- サブプログラムの直接のコール
- サブプログラム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

現行ユーザーのスキーマで解決される外部参照の場合（DML 文内など）、現行ユーザーはサブプログラムが参照するスキーマ・オブジェクトへのアクセスに必要な権限を持っている必要があります。その他すべての外部参照（ファンクション・コールなど）の場合は、所有者の権限がコンパイル時にチェックされ、実行時にはチェックされません。

定義者権限サブプログラムは、実行者にかかわらず、その所有者のセキュリティ・ドメインで作動します。したがって、所有者はサブプログラムが参照するスキーマ・オブジェクトへのアクセスに必要な権限を持っている必要があります。

複数のサブプログラムからなるプログラムを作成できます。定義者権限を持つものと実行者権限を持つものとを混在させることもできます。次に、EXECUTE 権限を使用して、プログラムのエントリ・ポイント（*controlled step-in*）を制限します。これによって、エントリ・ポイント・サブプログラムのユーザーは他のサブプログラムを直接ではなく間接的に実行できます。

実行者権限サブプログラムに対する権限の付与 : 例

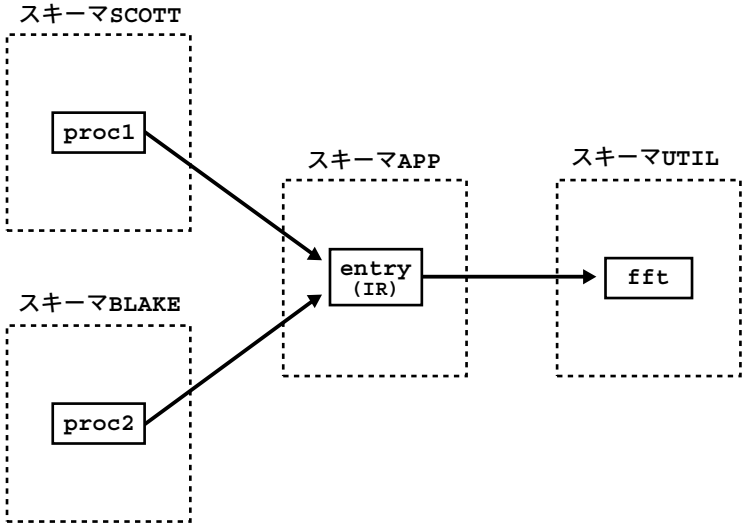
次に示すように、ユーザー util が、サブプログラム fft に対する EXECUTE 権限を、ユーザー app に付与するとします。

```
GRANT EXECUTE ON util.fft TO app;
```

ユーザー app は、サブプログラム 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、8-57 ページの図 8-7 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に 1 人ずつ EXECUTE 権限を付与する必要はありません。

サブプログラム util.fft は、実行者権限サブプログラム app.entry からのみ直接コールされることに注意してください。したがって、ユーザー util はユーザー app にのみ EXECUTE 権限を付与する必要があります。util.fft を実行すると、その現行ユーザーは app、scott または blake になります。scott および blake が EXECUTE 権限を付与されていない場合も同様です。

図 8-7 実行者権限サブプログラムへの間接コール



実行者権限サブプログラムでのロールの使用

サブプログラムでのロールの使用は、定義者権限と実行者権限のどちらで実行するかによって異なります。定義者権限サブプログラムでは、すべてのロールが使用禁止になります。ロールは権限チェックには使用されないため、ロールを設定することはできません。

実行者権限サブプログラムでは、ロールは使用可能になります（サブプログラムが定義者権限サブプログラムによって直接または間接的にコールされた場合を除きます）。ロールは権限チェックに使用されるため、システム固有の動的 SQL を使用してセッションにロールを設定できます。しかし、ロールはコンパイル時ではなく実行時に適用されるため、ロールを使用してテンプレート・オブジェクトに権限を付与することはできません。

実行者権限サブプログラムでのビューおよびデータベース・トリガーの使用

ビュー式内で実行される実行者権限サブプログラムの場合は、ビュー・ユーザーではなく、ビューの所有者が現行ユーザーとみなされます。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。実行者権限ファンクション layout は、常に、ビューの所有者であるユーザー scott の権限で実行されます。

```
CREATE VIEW payroll AS SELECT layout(3) FROM dual;
```

この規則は、データベース・トリガーにも適用されます。

実行者権限サブプログラムでのデータベース・リンクの使用

実行者権限は、1 種類のデータベース・リンク、つまり現行ユーザー・リンクにのみ影響します。これは、次のように作成されます。

```
CREATE DATABASE LINK link_name CONNECT TO CURRENT_USER
   USING connect_string;
```

現行ユーザー・リンクでは、そのユーザー権限を持つ別のユーザーとしてリモート・データベースに接続できます。接続するために、Oracle では現行ユーザーのユーザー名を使用します（実行者はグローバル・ユーザーである必要があります）。ユーザー blake が所有する実行者権限サブプログラムが、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・ユーザー scott がそのサブプログラムをコールしていれば、現行ユーザーであるユーザー scott でデータベース dallas に接続します。

```
CREATE DATABASE LINK dallas CONNECT TO CURRENT_USER USING ...
```

これが定義者権限サブプログラムであった場合、現行ユーザーは blake になります。このため、サブプログラムはグローバル・ユーザー blake でダラスのデータベースに接続することになります。

実行者権限サブプログラムでのオブジェクト型の使用

任意のスキーマで使用するオブジェクト型を定義するために、AUTHID CURRENT_USER 句を指定します。（オブジェクト型の詳細は、第 10 章を参照。）ユーザー blake が次のようにオブジェクト型を作成するとします。

```
CREATE TYPE Num AUTHID CURRENT_USER AS OBJECT (
   x NUMBER,
   STATIC PROCEDURE new_num (
      n NUMBER, schema_name VARCHAR2, table_name VARCHAR2)
);

CREATE TYPE BODY Num AS
   STATIC PROCEDURE new_num (
      n NUMBER, schema_name VARCHAR2, table_name VARCHAR2) IS
      sql_stmt VARCHAR2(200);
   BEGIN
      sql_stmt := 'INSERT INTO ' || schema_name || '.'
         || table_name || ' VALUES (blake.Num(:1))';
      EXECUTE IMMEDIATE sql_stmt USING n;
   END;
END;
```

次にユーザー blake は、オブジェクト型 Num に対する EXECUTE 権限を、ユーザー scott に付与します。

```
GRANT EXECUTE ON Num TO scott;
```

最後に、ユーザー scott は、Num 型のオブジェクトを格納するためにオブジェクト表を作成します。次に、プロシージャ new_num をコールして、その表にデータを入れます。

```
CONNECT scott/tiger;
CREATE TABLE num_tab OF blake.Num;
/
BEGIN
   blake.Num.new_num(1001, 'scott', 'num_tab');
   blake.Num.new_num(1002, 'scott', 'num_tab');
   blake.Num.new_num(1003, 'scott', 'num_tab');
END;
/
```

コールは成功しました。これはプロシージャがその所有者（blake）の権限ではなく現行ユーザー（scott）の権限で実行されたためです。

オブジェクト型階層内のサブタイプには、次の規則が適用されます。

- サブタイプで AUTHID 句が明示的に指定されていない場合は、スーパータイプの AUTHID を継承します。
- サブタイプで AUTHID 句が指定されている場合、その AUTHID がスーパータイプの AUTHID と一致する必要があります。また、AUTHID が DEFINER の場合は、スーパータイプとサブタイプの両方が同じスキーマに作成されている必要があります。

実行者権限のインスタンス・メソッドのコール

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 をコールする場合、メソッドの現行ユーザーは、scott ではなく、blake です。次の例を考えます。

```
-- 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;
```

再帰の理解と使用

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用されたフィボナッチ数列（0, 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、それ自身をさらに単純なバージョンに定義します。*n* の階乗（*n!*、1 ～ *n* のすべての整数の積）の定義を考えてみます。

```
n! = n * (n - 1)!
```

再帰的サブプログラム

再帰的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成されます。

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る可能性があります。

再帰的サブプログラムには、再帰的コールへ導くパスとそうではないパスの、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。実際に再帰的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。

再帰の例 : 階乗の計算

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す必要がある場合があります。これを解決するには、反復または再帰を使用します。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実装する場合は、次のような再帰的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;
```

再帰的コールのたびに 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。

再帰の例 : ツリー構造を持つデータの横断

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 VARCHAR2(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_no 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用します。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て、スタッフ・メンバーを探します。再帰的コールのたびに、FOR ループの新しいインスタンスが作成され、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。再帰的コールは 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、個々の再帰的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。

ヒント : CONNECT BY 句を使用した再帰問合せの実行

ここで示したのは再帰の例で、集合指向の 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、複数表の問合せは 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。

相互再帰の使用

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ため、odd を前方宣言します。

```
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し合います。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。再帰を使用し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;
   accumulator  INTEGER;
BEGIN
   IF (n = 1) OR (n = 2) THEN
      RETURN 1;
   ELSE
      accumulator := pos1 + pos2;
      FOR i IN 3..n LOOP
         pos2 := pos1;
         pos1 := accumulator;
         accumulator := pos1 + pos2;
      END LOOP;
      RETURN accumulator;
   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記憶域の消費量が少ないため、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ためです。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。

外部サブプログラムのコール

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。ただし、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サブプログラム（他の言語で書かれたサブプログラム）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 PL/SQL で活用できます。

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、サブプログラムまたは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のみ増やすメソッドがあります。`raiseSalary` は `void` メソッドであるため、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 subprogram
```

通常、外部 C サブプログラムは、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用します。外部 C サブプログラムを使用すると、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。

Java ストアド・プロシージャの詳細は、『Oracle9*i* Java ストアド・プロシージャ開発者ガイド』を参照してください。外部 C サブプログラムの詳細は、『Oracle9*i* 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。

PL/SQL Server Pages（PSP）を使用した動的 Web ページの作成

PL/SQL Server Pages（PSP）を使用すると、Web ページを動的コンテンツで開発できます。これは、Web ページ用に HTML コードを一度に 1 行ずつ書き出すストアド・プロシージャをコーディングするための代替手段です。

特殊なタグを使用すると、PL/SQL スクリプトを HTML ソース・コードに埋め込むことができます。スクリプトは、ページがブラウザなどの Web クライアントによって要求された時に実行されます。スクリプトは、パラメータ、データベースへの問合せまたは更新を受け入れる事が可能で、結果を示すカスタマイズ済みのページを表示できます。

開発中、PSP はページ・レイアウト用の静的部分およびコンテンツ用の動的部分を持つテンプレートのように動作できます。任意の HTML オーサリング・ツールを使用してレイアウトを設計できます。プレースホルダは動的コンテンツ用に残します。次に、コンテンツを生成する PL/SQL スクリプトを作成できます。終了した後、作成した PSP ファイルをストアド・プロシージャとして単にデータベースにロードします。

PSP の作成と使用の詳細は、『Oracle9*i* 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。

9

PL/SQL パッケージ

この章では、互いに関連する PL/SQL プログラミング・リソースを 1 つのパッケージにまとめる方法を示します。リソースには、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。作成し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有できます。

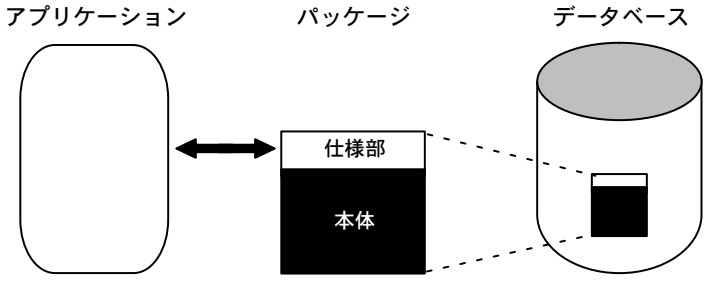
この章の項目は、次のとおりです。

PL/SQL パッケージ

パッケージとは、論理的に関連する PL/SQL の型、項目およびサブプログラムをグループ化したスキーマ・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、仕様部を実装します。

図 9-1 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 9-1 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 から対話形式で実行できます。次に構文を示します。

```
CREATE [OR REPLACE] PACKAGE package_name
   [AUTHID {CURRENT_USER | DEFINER}]
   {IS | AS}
   [PRAGMA SERIALLY_REUSABLE;]
   [collection_type_definition ...]
   [record_type_definition ...]
   [subtype_definition ...]
   [collection_declaration ...]
   [constant_declaration ...]
   [exception_declaration ...]
   [object_declaration ...]
   [record_declaration ...]
   [variable_declaration ...]
   [cursor_spec ...]
   [function_spec ...]
   [procedure_spec ...]
   [call_spec ...]
   [PRAGMA RESTRICT_REFERENCES(assertions) ...]
```

```
END [package_name];

[CREATE [OR REPLACE] PACKAGE BODY package_name {IS | AS}
   [PRAGMA SERIALLY_REUSABLE;]
   [collection_type_definition ...]
   [record_type_definition ...]
   [subtype_definition ...]
   [collection_declaration ...]
   [constant_declaration ...]
   [exception_declaration ...]
   [object_declaration ...]
   [record_declaration ...]
   [variable_declaration ...]
   [cursor_body ...]
   [function_spec ...]
   [procedure_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。サブプログラムは、仕様部で他のすべての項目の後で最後に宣言する必要があります（ただし、特定のファンクションの名前を指定するプラグマは、ファンクション仕様部の後に宣言する必要があります）。

本体には、実装の細部と、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、8-50 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で、ルーチンを発行します。Java コール仕様を作成する方法は、『Oracle9*i* 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 コール仕様を作成する方法は、『Oracle9*i* 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。

PL/SQL パッケージの例

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 INT, 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、パッケージ仕様部の宣言のみです。パッケージ本体の実装の詳細は隠ぺいされ、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（実装）を変更できます。

PL/SQL 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、情報の隠ぺい、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

パッケージを使用すると、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、明確に定義されています。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のみが必要です。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。

情報の隠ぺい

パッケージを使用すると、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実装を隠ぺいするため、実装が変更された場合も（アプリケーションではなく）パッケージのみが影響を受けます。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、実装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する場合も、データベースにデータを格納する必要がありません。

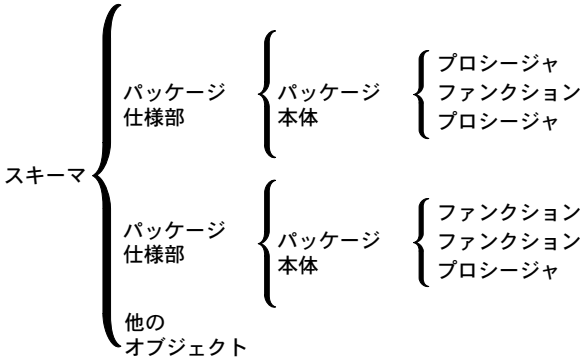
パフォーマンスの向上

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 I/O 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ため、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・ファンクションの実装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。

パッケージ仕様部の理解

パッケージ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 9-2 に有効範囲を示します。

図 9-2 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用するために必要な情報は、すべて仕様部の中にあります。たとえば、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のみです。ユーザーは fac の下位の実装のこと（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。

下位の実装を持つのは、サブプログラムとカーソルのみです。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のみであれ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。

```
CREATE PACKAGE trans_data AS  -- bodiless package
   TYPE TimeRec IS RECORD (
      minutes SMALLINT,
      hours   SMALLINT);
   TYPE TransRec IS RECORD (
      category VARCHAR2,
      account  INT,
      amount   REAL,
      time_of  TimeRec);
   minimum_balance    CONSTANT REAL := 10.00;
```

```
   number_processed   INT;
   insufficient_funds EXCEPTION;
END trans_data;
```

型、定数、変数および例外は下位の実装を持たないため、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・トリガーで使用できる）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・プログラムおよび様々な Oracle のツール製品から参照できます。たとえば、パッケージ・プロシージャ hire_employee は次のように 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 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。

パッケージ本体の理解

パッケージ本体はパッケージ仕様部を実装します。つまり、パッケージ本体には、パッケージ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実装が含まれています。パッケージ本体で定義されたサブプログラムにパッケージの外側からアクセスするには、その指定がパッケージ仕様部に存在している必要がある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る必要があります。一致していない場合、PL/SQL は例外を呼び出します。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_bonus (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ため、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のみ実行されます。

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のみであれ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用して、パッケージ仕様部で宣言した項目を初期化できます。

パッケージ機能の例

次に示す emp_actions という名前のパッケージの例を考えます。パッケージ仕様部では、次のような型、項目およびサブプログラムを宣言します。

- EmpRecTyp 型および DeptRecTyp 型
- カーソル desc_salary
- 例外 invalid_salary
- 関数 hire_employee および nth_highest_salary
- プロシージャ fire_employee および raise_salary

パッケージを作成すると、そのパッケージの型の参照、サブプログラムのコール、カーソルの使用、例外のコールなどを行うアプリケーションを開発できます。パッケージを作成すると、そのパッケージは Oracle データベースに格納され、様々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 INT, salary REAL);
   TYPE DeptRecTyp IS RECORD (dept_id INT, location VARCHAR2);
   CURSOR desc_salary RETURN EmpRecTyp;
   invalid_salary EXCEPTION;

   /* Declare externally callable subprograms. */
   FUNCTION hire_employee (
      ename  VARCHAR2,
      job    VARCHAR2,
      mgr    REAL,
      sal    REAL,
      comm   REAL,
      deptno REAL) RETURN INT;
   PROCEDURE fire_employee (emp_id INT);
   PROCEDURE raise_salary (emp_id INT, grade INT, amount REAL);
   FUNCTION nth_highest_salary (n INT) RETURN EmpRecTyp;
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
   number_hired INT;  -- 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    REAL,
   sal    REAL,
   comm   REAL,
   deptno REAL) RETURN INT IS
   new_empno INT;
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 INT) IS
BEGIN
   DELETE FROM emp WHERE empno = emp_id;
END fire_employee;

/* Define local function, available only inside package. */
FUNCTION sal_ok (rank INT, salary REAL) RETURN BOOLEAN IS
   min_sal REAL;
   max_sal REAL;
BEGIN
   SELECT losal, hisal INTO min_sal, max_sal FROM salgrade
      WHERE grade = rank;
   RETURN (salary >= min_sal) AND (salary <= max_sal);
END sal_ok;

PROCEDURE raise_salary (emp_id INT, grade INT, amount REAL) IS
   salary REAL;
BEGIN
   SELECT sal INTO salary FROM emp WHERE empno = emp_id;
   IF sal_ok(grade, salary + amount) THEN
      UPDATE emp SET sal = sal + amount WHERE empno = emp_id;
   ELSE
      RAISE invalid_salary;
   END IF;
END raise_salary;

FUNCTION nth_highest_salary (n INT) 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;

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のみ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 行のみです。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仮定します。

```
CREATE PACKAGE bank_transactions AS
   /* Declare externally visible constant. */
   minimum_balance  CONSTANT REAL := 100.00;
   /* Declare externally callable procedures. */
   PROCEDURE apply_transactions;
   PROCEDURE enter_transaction (
      acct   INT,
      kind   CHAR,
      amount REAL);
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 INT, credit REAL);
   PROCEDURE debit_account (acct INT, 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);
      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   INT,
   kind   CHAR,
   amount REAL) 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    INT,
   kind    CHAR,
   new_bal REAL) 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 INT, credit REAL) IS
/* Credit account unless account number is bad. */
   old_balance REAL;
   new_balance REAL;
```

```
   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 INT, debit REAL) IS
   /* Debit account unless account number is bad or
      account has insufficient funds. */
      old_balance REAL;
      new_balance REAL;
      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用していません。

パッケージの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用できません。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。

ただし、例外 invalid_salary 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 invalid_salary 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する必要がある項目は、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。セッションが終了すると、値が失われます。

パブリックにする必要がある項目は、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。

パッケージ・サブプログラムのオーバーロード

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、データ型が異なるパラメータからなる類似したパラメータのセットを受け取れるようにする場合は、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 REAL, trans_date VARCHAR2);
   PROCEDURE journalize (amount REAL, trans_date INT);
END journal_entries;

CREATE PACKAGE BODY journal_entries AS
   ...
   PROCEDURE journalize (amount REAL, trans_date VARCHAR2) IS
   BEGIN
      INSERT INTO journal
         VALUES (amount, TO_DATE(trans_date, 'DD-MON-YYYY'));
   END journalize;

   PROCEDURE journalize (amount REAL, trans_date INT) 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、8-23 ページの「サブプログラム名のオーバーロード」を参照してください。

パッケージ STANDARD による PL/SQL 環境の定義

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 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ファンクション ABS を次のように宣言します。

```
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ことができます。その内容を参照する場合もパッケージ名に接頭辞を付けて修飾名にする必要はありません。たとえば、ABS はデータベース・トリガー、ストアド・サブプログラム、Oracle のツール製品または 3GL アプリケーションから次のようにコールできます。

```
abs_diff := ABS(x - y);
```

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、ABS への参照を修飾することで組込みファンクションをコールすることもできます。

```
abs_diff := STANDARD.ABS(x - y);
```

ほとんどの組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の概要

Oracle といくつかの Oracle のツール製品では、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、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

DBMS_ALERT パッケージ

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・トリガーを使用し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用して、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 パッケージ

DBMS_OUTPUT パッケージを使用すると、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パラメータ 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 パッケージ

パッケージ DBMS_PIPE を使用すると、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 を使用し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 を使用して、メッセージを受信し、アンパック（読込み）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、外部プログラム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 パッケージ

パッケージ UTL_FILE を使用すると、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、ファンクション fopen 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・コールで使用するためのファイル・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。また、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。

UTL_HTTP パッケージ

UTL_HTTP パッケージを使用すると、PL/SQL プログラムで HTTP（Hypertext Transfer Protocol）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 カートリッジをコールすることもできます。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（Uniform Resource Locator）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 HTML（Hypertext Markup Language）形式のものです。

パッケージ作成のガイドライン

パッケージを作成する場合は、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成しないように注意してください。

パッケージ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ることが必要な型、項目、およびサブプログラムのみにします。こうすれば、他の開発者が実装上の細部に基づくコードを書いて、パッケージを誤って使用すること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。ただし、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する必要があります。

10

PL/SQL のオブジェクト型

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成するのに必要なコストおよび時間を縮小できるため、急速に普及してきました。PL/SQL 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に対応する抽象的なテンプレートを提供するため、モデル化ツールとして理想的です。オブジェクト型をプログラム内に組み込むためには、その使用方法を理解するのみで、その仕組みまで理解する必要はありません。

この章の項目は、次のとおりです。

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。抽象化では、無関係な詳細を切り捨てることで、日常生活を維持管理できます。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。ボンネットの下で行わ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、**プロシージャ抽象化**は、手続き（プロシージャ）を記述してそれにパラメータを渡し、複雑なアルゴリズムの詳細を削除するときに使用します。異なる実装の試行に必要なのは、プロシージャ本体の単純な置換のみです。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**を使用しています。データ型が表す内容は、それらの値に対する適切な一連の値と一連の操作です。たとえば、POSITIVE 型の変数には正の整数のみを入れることができ、加算、減算、乗算などのみを実行できます。変数を使用するために、PL/SQL による整数の格納方法や算術演算の実行方法を知る必要はありません。

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、様々なスカラー型やコンポジット・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。**コンポジット型**（RECORD など）には、別個に操作できる内部的な要素があります。RECORD 型のように、オブジェクト型はコンポジット型です。ただし、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 文を使用して作成し、Oracle データベースに格納して、多くのプログラムで共有できるようにする必要があります。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成したり、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更できます。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。

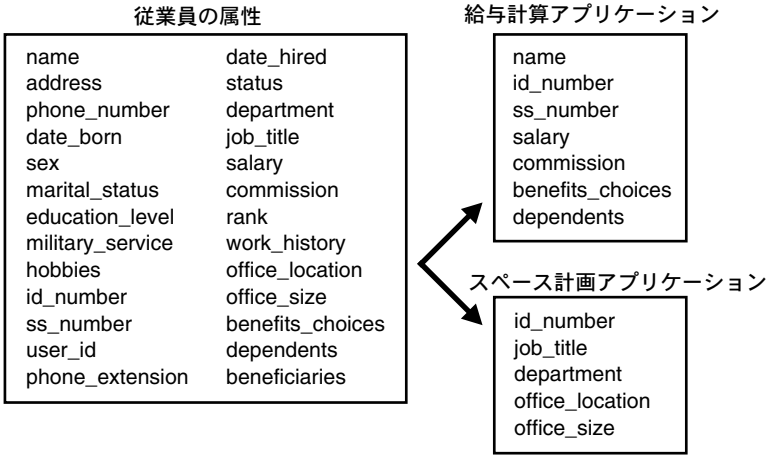
オブジェクト型

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、それに様々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 文を使用してオブジェクト型を定義する場合は、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成します。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に必要な属性と動作のみを指定します。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要件を満たすのに必要な属性は、そのうちの少数のみです（図 10-1 を参照）。

図 10-1 各アプリケーションでのオブジェクト属性のサブセットの使用



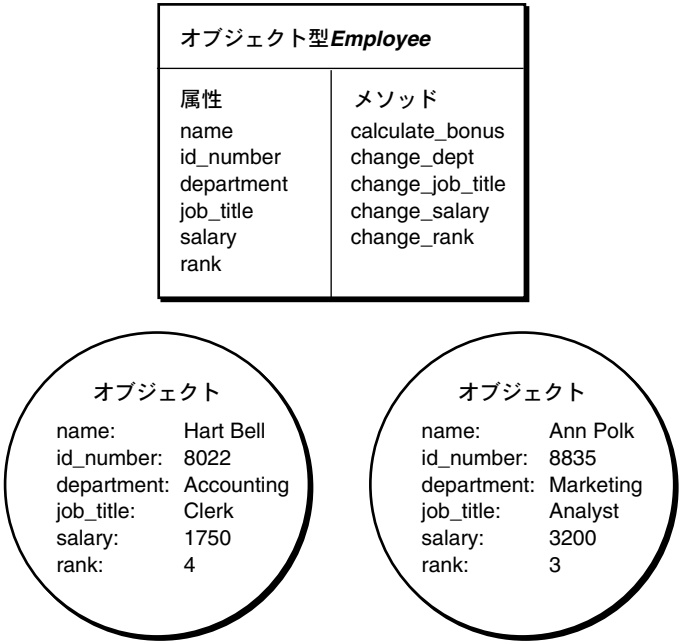
ボーナスを従業員に割り振るプログラムを作成する必要があるとします。この問題の解決に、すべての社員属性は必要ありません。そこで、名前（name）、ID 番号（id_number）、部署（department）、肩書（job_title）、給与（salary）および等級（rank）という、この問題に関係のある特定の属性のみを取り出して抽象化した従業員を設計することにします。次に、その抽象化した社員の処理に必要な操作について考えます。たとえば、管理者が従業員の等級を変更するための操作が必要となります。

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。

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（クライアント・プログラムから参照可能）です。しかし、正しいプログラムは、それを直接操作しません。提供される一連のメソッドを使用します。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られた時点で、抽象概念としての従業員の**インスタンス** が作成されることになります。インスタンス（通常は**オブジェクト**）は、必要な数のみ作成します。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 10-2 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用すれば、属性と動作が適切に定義されたオブジェクトを作成できます。

図 10-2 オブジェクト型とそのオブジェクト（インスタンス）



オブジェクト型を使用する理由

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成できます。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。

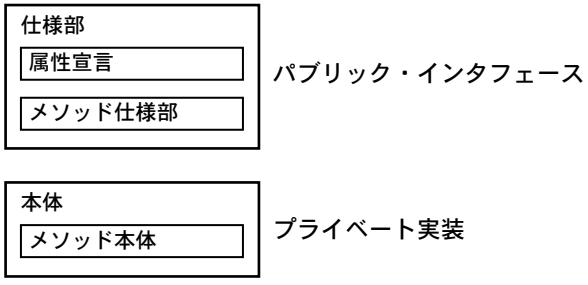
データに対する操作をカプセル化することで、オブジェクト型を使用してデータ・メンテナンスのためのコードを SQL スクリプトや PL/SQL ブロックではなく、メソッドに入れることができます。オブジェクト型では、データへのアクセスが、そのための許可を受けた操作によってしかできないようにすることで、副作用が最小限になります。また、オブジェクト型を使用すれば、インプリメンテーションの細部が隠されるため、クライアント・プログラムに影響を及ぼすことなく細部を変更できます。

オブジェクト型を使用することにより、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 10-3 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実装します。

図 10-3 オブジェクト型構造



メソッドを使用する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する必要があります。サブプログラムのみが実装を必要とします。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（オブジェクト型の外側から参照可能）です。

構造をさらに理解するために、次の例を考えてみます。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っていれば十分です。

```
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;
```

オブジェクト型の構成要素

オブジェクト型はデータと操作をカプセル化します。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。

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一意である必要があります（他のオブジェクト型内では使用できます）。データ型には、任意の Oracle 型を使用できますが、次の型は使用できません。

- LONG、LONG RAW
- 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 句を使用しての属性の初期化はできません。また、属性に NOT NULL 制約を課すことはできません。ただし、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 変数のみです。一方、大学の学生を表すには、名前、住所、電話番号、状態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複雑なオブジェクト型を作成することが可能になります。待ち行列、リストおよびツリーなどの一部のオブジェクト型は動的です。つまり、使用さ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 を使用し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いずれかと同じ名前は使用できません。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()
```

ただし、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、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、およびファンクションの場合は戻り型から構成されます。**本体**は、特定の作業を実行するためのコードです。

オブジェクト型仕様部のメソッド仕様部ごとに、対応するメソッド本体がオブジェクト型本体に存在している必要があります。または、メソッドに NOT INSTANTIABLE を宣言し、メソッド本体がこの型のサブタイプにのみ存在することを示す必要があります。メソッドの仕様部と本体を一致させるために、PL/SQL 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。ヘッダーは正確に一致する必要があります。

属性と同じように、仮パラメータは名前とデータ型を指定して宣言します。ただし、パラメータのデータ型をサイズ制約することはできません。データ型としては、任意の Oracle 型を使用できますが、属性に認められていないものは使用できません。（10-7 ページの「属性」を参照。）同じ制限が戻り値の型にも適用されます。

メソッドに使用可能な言語

Oracle では、オブジェクトのメソッドを PL/SQL、Java または C 言語で実装できます。型でコール仕様部を指定すると、型のメソッドを Java または C 言語で実装できます。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 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 コール仕様を作成する方法は、『Oracle9*i* 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 コール仕様を作成する方法は、『Oracle9*i* 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。

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。暗黙的宣言か明示的宣言かにかかわらず、SELF は常に MEMBER メソッドに渡される第 1 パラメータです。ただし、STATIC メソッドは SELF の受入れや参照ができません。

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。ただし、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・モードを指定できません。

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用できます。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。

サブタイプも、スーパータイプから継承するメソッドをオーバーロードできます。この場合、メソッドの仮パラメータは正確に一致します。

2 つのメソッドの仮パラメータの違いがパラメータ・モードのみの場合、それらのメソッドはオーバーロードできません。また、戻り型しか違わない 2 つのメンバー関数はオーバーロード できません。詳細は、8-23 ページの「サブプログラム名のオーバーロード」を参照してください。

MAP メソッドと ORDER 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ため、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 **MAP 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 Rational 型のオブジェクトを 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 は順序付けを使用して、x > y などのブール式を評価したり、DISTINCT、GROUP BY および ORDER BY 句によって暗黙的に必要となる比較を実行します。MAP 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に含めることができる MAP メソッドは 1 つのみです。組込みパラメータ SELF を受け入れて、スカラー型 DATE、NUMBER、VARCHAR2 のうちの 1 つ、あるいは CHARACTER や REAL などの ANSI SQL 型を戻します。

あるいは、**ORDER メソッド**を使用することもできます。1 つのオブジェクト型に含めることができる ORDER メソッドは 1 つのみです。このメソッドは、戻り型が数値のファンクションである必要があります。次の例のキーワード ORDER は、メソッド match() によって、2 つのオブジェクトが比較され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;
```

ORDER メソッドはいずれも、組込みパラメータ 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、あるいは大きいことを示しています。ORDER 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な働きをし、オブジェクト値をスカラー値にマップし、< や = などの演算子を使用して比較します。ORDER メソッドは、単に 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します。

MAP メソッドまたは ORDER 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。ただし、どちらのメソッドも宣言しなければ、オブジェクトは SQL 文でのみ比較し、しかも比較できるのは等しいか等しくないかについてのみです。（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のみです。）

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 メソッドを使用します。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、何回もコールする必要があるため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 メソッドを使用する必要があります。

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。このコンストラクタ・メソッドは、そのオブジェクト型と同じ名前のファンクションで、そのオブジェクト型の新しいインスタンスを初期化して戻します。

Oracle では、すべてのオブジェクト型に対して、そのオブジェクト型の属性と一致した仮パラメータ付きのデフォルト・コンストラクタが生成されます。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。

システム定義のコンストラクタを上書きするか、または異なるシグネチャで新規ファンクションを定義して、ユーザー独自のコンストラクタ・メソッドを定義できます。

PL/SQL は、暗黙的にコンストラクタをコールすることはないため、明示的にコールする必要があります。

詳細は、10-28 ページの「コンストラクタの定義」を参照してください。

既存のオブジェクト型の属性およびメソッドの変更（型の発展）

ALTER TYPE 文を使用すると、既存のオブジェクト型の属性の追加、変更または削除やメソッドの追加または削除ができます。

```
CREATE TYPE Person_typ AS OBJECT
( name CHAR(20),
ssn CHAR(12),
address VARCHAR2(100));
CREATE TYPE Person_nt IS TABLE OF Person_typ;
CREATE TYPE dept_typ AS OBJECT
( mgr Person_typ,
emps Person_nt);
CREATE TABLE dept OF dept_typ;

-- Add new attributes to Person_typ and propagate the change
-- to Person_nt and dept_typ
ALTER TYPE Person_typ ADD ATTRIBUTE (picture BLOB, dob DATE)
CASCADE NOT INCLUDING TABLE DATA;

CREATE TYPE mytype AS OBJECT (attr1 NUMBER, attr2 NUMBER);
ALTER TYPE mytype ADD ATTRIBUTE (attr3 NUMBER),
DROP ATTRIBUTE attr2,
ADD ATTRIBUTE attr4 NUMBER CASCADE;
```

プロシージャのコンパイル時には常に、参照先のオブジェクト型の現行バージョンが使用されます。サーバー上でオブジェクト型を参照している既存のプロシージャは、その型が変更

されると無効になり、次回のコール時に自動的に再コンパイルされます。変更があった型を参照するクライアント側のプロシージャは、すべて手動で再コンパイルする必要があります。

スーパータイプからメソッドを削除すると、そのメソッドをオーバーライドするサブタイプにも変更が必要になる場合があります。サブタイプが影響を受けるかどうかは、ALTER TYPE 文の CASCADE オプションを使用して検索できます。この文は、メソッドをオーバーライドするサブタイプがあるとロールバックされます。メソッドをスーパータイプから正常に削除するには、次の方法があります。

- 最初にメソッドをサブタイプから永続的に削除します。
- サブタイプのメソッドを削除してから、後で OVERRIDING キーワードを指定せずに ALTER TYPE 文を使用して追加します。

ALTER TYPE 文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。アプリケーションに型の発展を使用する際の指針と、他の型とその依存データを変更するためのオプションについては、『Oracle9*i* アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』を参照してください。

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成する準備をします。

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。ただし、SQL*Plus では、次の構文を使用して対話形式で定義できます。

```
CREATE [OR REPLACE] TYPE type_name
  [AUTHID {CURRENT_USER | DEFINER}]
  { {IS | AS} OBJECT | UNDER supertype_name }
(
  attribute_name datatype[, attribute_name datatype]...
  [{MAP | ORDER} MEMBER function_spec,]
  [{FINAL| NOT FINAL} MEMBER function_spec,]
  [{INSTANTIABLE| NOT INSTANTIABLE} MEMBER function_spec,]
  [{MEMBER | STATIC} {subprogram_spec | call_spec}
  [, {MEMBER | STATIC} {subprogram_spec | call_spec}]...]
) [{FINAL| NOT FINAL}] [ {INSTANTIABLE| NOT INSTANTIABLE}];

[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、8-50 ページの「実行者権限と定義者権限」を参照してください。

PL/SQL の型の継承の概要

PL/SQL では、単一継承モデルがサポートされます。オブジェクト型のサブタイプを定義できます。このサブタイプには、親の型（スーパータイプ）の属性とメソッドがすべて含まれます。また、サブタイプには他の属性やメソッドを含めたり、スーパータイプからのメソッドをオーバーライドできます。

サブタイプを特定の型から導出できるかどうかを定義できます。また、直接はインスタンス化できず、インスタンス化するサブタイプを宣言する必要がある型とメソッドも定義できます。

型のプロパティの一部は、ALTER TYPE 文を使用して動的に変更できます。ALTER TYPE 文を使用または再定義して、スーパータイプに変更を加えると、そのサブタイプには変更結果が自動的に反映されます。

TREAT 演算子を使用すると、指定したサブタイプのオブジェクトのみを戻すことができます。

REF および DEREF ファンクションからの値では、表またはビューの宣言された型、あるいはその 1 つ以上のサブタイプを表すことができます。

これらのオブジェクト・リレーショナル機能の詳細は、『Oracle9*i* アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』を参照してください。

PL/SQL の型の継承の例

```
-- Create a supertype from which several subtypes will be derived.
CREATE TYPE Person_typ AS OBJECT ( ssn NUMBER, name VARCHAR2(30), address
VARCHAR2(100)) NOT FINAL;

-- Derive a subtype that has all the attributes of the supertype,
-- plus some additional attributes.
CREATE TYPE Student_typ UNDER Person_typ ( deptid NUMBER, major VARCHAR2(30)) NOT
FINAL;

-- Because Student_typ is declared NOT FINAL, you can derive
-- further subtypes from it.
CREATE TYPE PartTimeStudent_typ UNDER Student_typ( numhours NUMBER);

-- Derive another subtype. Because it has the default attribute
```

```
-- FINAL, you cannot use Employee_typ as a supertype and derive
-- subtypes from it.
CREATE TYPE Employee_typ UNDER Person_typ( empid NUMBER, mgr VARCHAR2(30));


-- Define an object type that can be a supertype. Because the
-- member function is FINAL, it cannot be overridden in any
-- subtypes.

CREATE TYPE T AS OBJECT (..., MEMBER PROCEDURE Print(), FINAL MEMBER
FUNCTION foo(x NUMBER)...) NOT FINAL;

-- We never want to create an object of this supertype. By using
-- NOT INSTANTIABLE, we force all objects to use one of the subtypes
-- instead, with specific implementations for the member functions.
CREATE TYPE Address_typ AS OBJECT(...) NOT INSTANTIABLE NOT FINAL;

-- These subtypes can provide their own implementations of
-- member functions, such as for validating phone numbers and
-- postal codes. Because there is no "generic" way of doing these
-- things, only objects of these subtypes can be instantiated.
CREATE TYPE USAddress_typ UNDER Address_typ(...);
CREATE TYPE IntlAddress_typ UNDER Address_typ(...);

-- Return REFs for those Person_typ objects that are instances of
-- the Student_typ subtype, and NULL REFs otherwise.
SELECT TREAT(REF(p) AS REF Student_typ) FROM Person_v p;

-- Example of using TREAT for assignment...

-- Return REFs for those Person_type objects that are instances of
-- Employee_type or Student_typ, or any of their subtypes.
SELECT REF(p) FROM Person_v P WHERE VALUE(p) IS OF (Employee_typ, Student_typ);

-- Similar to above, but do not allow any subtypes of Student_typ.
SELECT REF(p) FROM Person_v p WHERE VALUE(p) IS OF(ONLY Student_typ);

-- The results of REF and DEREF can include objects of Person_typ
-- and its subtypes such as Employee_typ and Student_typ.
SELECT REF(p) FROM Person_v p;
SELECT DEREF(REF(p)) FROM Person_v p;
```

オブジェクト型の例 : Stack

スタックは、データ項目の順序付き集合を保持します。スタックには先頭と末尾があります。項目を追加または削除できるのは、先頭からのみです。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用されます。スタックの最も単純な実装では整数の配列が使用され、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。これを使用すると、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。ただし、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、組込みプロシージャ raise_application_error を使用し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・レポート・ファンクション SQLCODE および SQLERRM を使用して OTHERS 例外ハンドラで処理できます。次の例では、例外が呼び出されたとき、対応する Oracle エラー・メッセージが出力されます。

```
DECLARE
   ...
BEGIN
   ...
EXCEPTION
   WHEN OTHERS THEN
      dbms_output.put_line(SQLERRM);
END;
```

または次の例に示すように、プログラムで EXCEPTION_INIT プラグマを使用して、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
      ...
END;
```

オブジェクト型の例 : Ticket_Booth

各映画館が 3 つの会場を持つ映画館のチェーンについて考えてみます。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る必要があります。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の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 を使用します。要素 1、2 および 3 には、それぞれ映画 1、2 および 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;
```

オブジェクト型の例 : 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る必要があります。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の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 (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 (num IN INTEGER)
      RETURN REAL IS
   BEGIN
      verify_acct(num);
      RETURN balance;
   END curr_bal;
END;
```

オブジェクト型の例 : Rational Numbers（有理数）

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 を定義することで、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 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。

ここで、次のスタンドアロン・ストアド・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すると、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、それを使用し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用して属性、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成されます。つまり、その型のオブジェクトがインスタンス化されます。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、通常の有効範囲規則とインスタンス化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 などの組込み型を使用できるところであれば、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すことができます。次の例では、オブジェクト型 Account を使用し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 を使用し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基本構造的に *NULL* になっています。つまり、その属性のみ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、基本構造的に 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基本構造的に NULL になります（したがって再度初期化する必要があります）。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基本構造的に 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のコールによって、事前定義済みの例外 NULL_SELF_DISPATCH が呼び出されます。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、書き込む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、その値を変更するには、ドット表記法を使用します。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;
```

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),
   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アクセスします。

```
s.home_address.zip_code
```

コンストラクタの定義

デフォルトでは、オブジェクト型に対するコンストラクタの定義は不要です。システムによって、各属性に対応するパラメータを受け入れるデフォルト・コンストラクタが指定されます。

次のように、独自のコンストラクタを定義することもできます。

- 一部の属性にデフォルト値を指定する場合。コール元に頼らずに、すべての属性値に値が正しく供給されるようにできます。
- 多数の、オブジェクトの異なる部分を初期化するのみの、特殊目的のプロシージャを回避する場合。
- 新規属性を型へ追加する時に、コンストラクタをコールするアプリケーションでのコード変更を回避する場合。たとえば、属性は NULL に設定するが、コンストラクタへの既存のコールが継続して動作できるように、そのシグネチャは変更しない場合、コンストラクタには新しいコードが必要となります。

次に例を示します。

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(
-- The type has 3 attributes.
  length NUMBER,
  width NUMBER,
  area NUMBER,
-- Define a constructor that has only 2 parameters.
  CONSTRUCTOR FUNCTION rectangle(length NUMBER, width NUMBER)
    RETURN SELF AS RESULT
);
/

CREATE OR REPLACE TYPE BODY rectangle AS
  CONSTRUCTOR FUNCTION rectangle(length NUMBER, width NUMBER)
    RETURN SELF AS RESULT
  AS
  BEGIN
    SELF.length := length;
    SELF.width := width;
-- We compute the area rather than accepting it as a parameter.
    SELF.area := length * width;
    RETURN;
  END;
END;
/

DECLARE
  r1 rectangle;
```

```
  r2 rectangle;
BEGIN
-- We can still call the default constructor, with all 3 parameters.
  r1 := NEW rectangle(10,20,200);
-- But it is more robust to call our constructor, which computes
-- the AREA attribute. This guarantees that the initial value is OK.
  r2 := NEW rectangle(10,20);
END;
/
```

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。すべての属性に値を入れるためにデフォルト・コンストラクタをコールする場合、属性には定数や変数とは異なりデフォルト値がないため、すべての属性にパラメータを指定する必要があります。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用し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用してコールされます。次の例では、属性 num および den（分子と分母）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
END;
```

次の例でわかるように、メソッドのコールは連鎖させることができます。実行は左から右へと進んでいきます。最初にメンバー関数 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
END;
```

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・リストは必要ありません。連鎖する場合は、最後のコール以外のすべてのコールで空のパラメータ・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ため、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、次の宣言は誤りです。

```
r.normalize().reciprocal;  -- not allowed
```

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す必要があります。

静的メソッドの場合、コールでは型のインスタンスを指定するかわりに、表記法 *type_name.method_name* を使用します。

サブタイプのインスタンスを使用してメソッドをコールする場合、実際に実行されるメソッドは型の継承での正確な宣言によって決まります。サブタイプがスーパータイプから継承するメソッドをオーバーライドする場合、コールではサブタイプの実装が使用されます。また、サブタイプがメソッドをオーバーライドしない場合、コールではスーパータイプの実装が使用されます。この機能は動的メソッド・ディスパッチと呼ばれます。

REF 修飾子による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、オブジェクト型リレーショナルよりもさらに大きく複雑です。オブジェクトが大きい場合、そのコピーのサブプログラム間での受渡しは非効率的です。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）を使用します。*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とき、変更は 1 箇所のみで行われ、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成することで、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;
```

オブジェクト型 Person を訂正することで、複数の人が同じ家を共有する家族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 を使用します。

```
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 DML 文の中で入力変数または出力変数として使用できます。ただし、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）、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;  -- not allowed
```

オブジェクトにアクセスするにはファンクション DEREF を使用するか、パッケージ UTL_REF をコールする必要があります。例については、10-36 ページの「ファンクション DEREF の使用」を参照してください。

先送り型定義

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のみです。次の例の最初の CREATE TYPE 文は、まだ存在していないオブジェクト型 Department を参照しているため、誤りです。

```
CREATE TYPE Employee AS OBJECT (
   name VARCHAR2(20),
   dept REF Department,  -- not allowed
   ...
);

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。オブジェクト型 Employee は、オブジェクト型 Department を参照する属性を持ち、オブジェクト型 Department は型 Employee の属性を持ちます。この問題を解決するには、相互に依存するオブジェクト型の定義が可能な、先送り型定義と呼ばれる特別な CREATE TYPE 文を使用します。

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
```

先送り型定義によって作成さ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 が未定義な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)
);
```

先送り型定義により、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 で使用できます。

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用します。表の作成後は、SQL 文を使用し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化するとき、それはデータベースの表の外部では認識されません。ただし、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成するために使用できます。

次の例では、Rational 型のオブジェクトを行に格納する表を作成します。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。行ごとに異なる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 を作成する次の 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、次の項で説明するファンクション VALUE を使用する必要があります。

ファンクション VALUE の使用

その名前のとおり、ファンクション 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対応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、VALUE を使用します。

```
BEGIN
   INSERT INTO employees
      SELECT VALUE(p) FROM persons p
          WHERE p.last_name LIKE '%Smith';
```

次の例では、VALUE を使用して特定の 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' のストアド・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、これらの変数を使用して、それらに含まれるオブジェクトにアクセスしたり更新できます。

```
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。

ファンクション REF の使用

ファンクション REF を使用すると、ref を取り出せます。このファンクションは、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 を使用します。たとえば、department というリレーショナル表の 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 文を使用すると、参照先がない REF を NULL に変換できます。

```
UPDATE department SET manager = NULL WHERE manager IS DANGLING;
```

ファンクション DEREF の使用

PL/SQL 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。SQL 文の中でファンクション DEREF を使用する必要があります。（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、一意の不変な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 を使用して 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、ファンクション DEREF をプロシージャ文の中で使用することはできません。

```
BEGIN
   ...
   p1 := DEREF(p_ref);  -- not allowed
```

SQL 文の中では、ドット表記法を使用し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 属性にナビゲートできます。また、表の別名を使用すれば、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 を作成する次の 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後に、次のように ref を参照解除することで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 文を使用します。次の例では、Person オブジェクトをオブジェクト表 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用してもオブジェクトをオブジェクト表 persons に追加できます。

```
BEGIN
   INSERT INTO persons
      VALUES (Person('Albert', 'Brooker', ...));
```

次の例では、RETURNING 句を使用して 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 文でも使用できます。

```
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用します。次に例を示します。

```
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 というリレーショナル表を作成してから、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 文を使用します。

```
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 文を使用します。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 句を使用します。

```
BEGIN
   DELETE FROM persons p
      WHERE p.home_address = '108 Palm Dr';
END;
```

11

システム固有の動的 SQL

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。

この章の項目は、次のとおりです。

動的 SQL

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、様々な 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、生成する様々な報告書用に、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、実行時まで不明です。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であるため、名前（接頭辞としてコロンが必要）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）を実行する場合。PL/SQL では、このような文を静的に実行できません。
- 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する場合。または、SELECT 文の WHERE 句に対して異なる検索条件を構築する場合。より複雑なプログラムが様々な SQL 操作や句などから選択される可能性があります。
- 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、『Oracle9*i* 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。）

EXECUTE IMMEDIATE 文の使用

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]...]
[{RETURNING | RETURN} INTO bind_argument[, 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 は選択された列値を格納する変数、record は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。入力 bind_argument は動的 SQL 文または PL/SQL ブロックに渡される値を持つ式です。出力 bind_argument は動的 SQL 文または PL/SQL ブロックによって戻される値を格納する変数です。

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。ただし、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。正しい方法は、11-14 ページの「任意の名前を持つスキーマ・オブジェクトをプロシージャで処理する方法」を参照してください。

INTO 句は単一行の問合せの場合に使用され、検索された列の値を入れる変数またはレコードを指定します。問合せによって取り出された値それぞれに対して、INTO 句の中に、対応する型互換性変数が存在している必要があります。

RETURNING INTO 句は、RETURNING 句のある（BULK COLLECT 句のない）DML 文の場合にのみ使用され、列の値が戻される変数を指定します。DML 文によって戻された値それぞれに対して、RETURNING INTO 句の中に、対応する型互換性変数が存在している必要があります。

バインド引数は、すべて USING 句に入れることができます。デフォルトのパラメータ・モードは IN です。RETURNING 句を持つ DML 文の場合は、パラメータ・モード OUT を定義して指定しなくても、OUT 引数を RETURNING INTO 句に入れることができます。USING 句と RETURNING INTO 句の両方を使用する場合、USING 句には IN 引数のみを含めることができます。

実行時に、バインド引数は動的文字列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 句内または RETURNING INTO 句内（あるいはその両方）のバインド引数に対応付ける必要があります。数値リテラル、文字リテラルおよび文字列リテラルはバインド引数として使用できますが、ブール・リテラル（TRUE、FALSE および NULL）は使用できません。動的文字列に NULL を渡すには、回避策を使用する必要があります。11-16 ページの「NULL を渡す方法」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。

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字列を準備するため、オーバーヘッドが発生します。

動的 SQL の例

次の PL/SQL ブロックには動的 SQL の例がいくつか含まれています。

```
DECLARE
   sql_stmt    VARCHAR2(200);
   plsql_block VARCHAR2(500);
   emp_id      NUMBER(4) := 7566;
   salary      NUMBER(7,2);
   dept_id     NUMBER(2) := 50;
   dept_name   VARCHAR2(14) := 'PERSONNEL';
   location    VARCHAR2(13) := 'DALLAS';
   emp_rec     emp%ROWTYPE;
BEGIN
   EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt NUMBER)';
   sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
   EXECUTE IMMEDIATE sql_stmt USING dept_id, dept_name, location;
   sql_stmt := 'SELECT * FROM emp WHERE empno = :id';
   EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING emp_id;
   plsql_block := 'BEGIN emp_pkg.raise_salary(:id, :amt); END;';
   EXECUTE IMMEDIATE plsql_block USING 7788, 500;
   sql_stmt := 'UPDATE emp SET sal = 2000 WHERE empno = :1
      RETURNING sal INTO :2';
   EXECUTE IMMEDIATE sql_stmt USING emp_id RETURNING INTO salary;
   EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = :num'
      USING dept_id;
   EXECUTE IMMEDIATE 'ALTER SESSION SET SQL_TRACE TRUE';
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のみを削除します。

```
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;
```

USING 句の下位互換性

動的 INSERT 文、UPDATE 文または DELETE 文に RETURNING 句がある場合、出力バインド引数を RETURNING INTO 句または USING 句に入れることができます。新しいアプリケーションでは、RETURNING INTO 句を使用します。古いアプリケーションでは、USING 句を引き続き使用できます。たとえば、次の EXECUTE IMMEDIATE 文はいずれも使用可能です。

```
DECLARE
   sql_stmt VARCHAR2(200);
   my_empno NUMBER(4) := 7902;
   my_ename VARCHAR2(10);
   my_job   VARCHAR2(9);
   my_sal   NUMBER(7,2) := 3250.00;
BEGIN
   sql_stmt := 'UPDATE emp SET sal = :1 WHERE empno = :2
      RETURNING ename, job INTO :3, :4';

   /* Bind returned values through USING clause. */
   EXECUTE IMMEDIATE sql_stmt
      USING my_sal, my_empno, OUT my_ename, OUT my_job;

   /* Bind returned values through RETURNING INTO clause. */
   EXECUTE IMMEDIATE sql_stmt
      USING my_sal, my_empno RETURNING INTO my_ename, my_job;
   ...
END;
```

パラメータのモードの指定

USING 句を使用すると、入力バインド引数のパラメータ・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。RETURNING INTO 句を使用すると、出力バインド引数のパラメータ・モードを指定できません。これは、定義ではパラメータ・モードが OUT であるためです。次に例を示します。

```
DECLARE
   sql_stmt VARCHAR2(200);
   dept_id  NUMBER(2) := 30;
   old_loc  VARCHAR2(13);
BEGIN
   sql_stmt :=
      'DELETE FROM dept WHERE deptno = :1 RETURNING loc INTO :2';
   EXECUTE IMMEDIATE sql_stmt USING dept_id RETURNING INTO old_loc;
   ...
END;
```

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
   SELECT deptno_seq.NEXTVAL INTO deptno FROM dual;
   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(500);
   new_deptno NUMBER(2);
   new_dname  VARCHAR2(14) := 'ADVERTISING';
   new_loc    VARCHAR2(13) := '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;
```

OPEN-FOR、FETCH および CLOSE 文の使用

動的な複数行の問合せを処理するには、OPEN-FOR 文、FETCH 文および 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後に、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、6-16 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 は弱い型指定のカーソル変数（戻り型を持ちません）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 emp 表から行を戻す動的 SELECT 文と対応付けます。

```
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;
```

問合せの中のバインド引数が評価されるのは、カーソル変数がオープンしている場合のみです。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フェッチ

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 に行を取り出します。

```
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;
```

カーソル変数と対応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ている必要があります。同じカーソル変数を使用して、別々の FETCH 文で、異なる INTO 句を使用できます。各 FETCH 文で同じ結果セットから別の行をフェッチします。

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。

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、または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。

レコード、オブジェクトおよびコレクションの動的 SQL の例

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フェッチ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(200);
   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方法を示しています。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 ||
         ' (pers Person, hobbs 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 pers, hobbs 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;
```

バルク動的 SQL の使用

この項では、動的 SQL にバルク・バインド機能を追加する方法について説明します。バルク・バインドは、PL/SQL エンジンと SQL エンジンの間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。

次のコマンド、句およびカーソル属性を使用すると、アプリケーションでバルク SQL 文を作成して、実行時に動的に実行できます。

- BULK FETCH 文
- BULK EXECUTE IMMEDIATE 文
- FORALL 文
- COLLECT INTO 句
- RETURNING INTO 句
- %BULK_ROWCOUNT カーソル属性

前述の文、句およびカーソル属性の静的バージョンについては、5-38 ページの「バルク・バインドを使用したコレクションのループ・オーバーヘッドの削減」を参照してください。背景情報も記載されています。

バルク動的バインドの構文

バルク・バインドにより、SQL 文の変数を値のコレクションにバインドできます。コレクション型には、任意の PL/SQL コレクション型（索引付き表、ネストした表または VARRAY）を使用できます。ただし、コレクション要素には CHAR、DATE または NUMBER などの SQL データ型が必要です。バルク動的バインドがサポートされる文は、EXECUTE IMMEDIATE、FETCH および FORALL です。

バルク EXECUTE IMMEDIATE

この文では、動的 SQL 文にパラメータとして渡される定義変数または OUT バインド引数をバルク・バインドできます。次に構文を示します。

```
EXECUTE IMMEDIATE dynamic_string
   [[BULK COLLECT] INTO define_variable[, define_variable ...]]
   [USING bind_argument[, bind_argument ...]]
   [{RETURNING | RETURN}
   BULK COLLECT INTO bind_argument[, bind_argument ...]];
```

動的な複数行の問合せでは、BULK COLLECT INTO 句を使用して定義変数をバインドできます。各列の値はコレクションに格納されます。

複数行を戻す動的 INSERT 文、UPDATE 文または DELETE 文では、RETURNING BULK COLLECT INTO 句を使用して出力変数をバルク・バインドできます。戻される行の値は、コレクション・セットに格納されます。

バルク FETCH

この文を使用すると、静的カーソルからフェッチする場合と同様に、動的カーソルからフェッチできます。次に構文を示します。

```
FETCH dynamic_cursor
   BULK COLLECT INTO define_variable[, define_variable ...];
```

BULK COLLECT INTO リストにある定義変数の数が問合せの選択リストの列数を超えている場合は、エラーが生成されます。

バルク FORALL

この文を使用すると、動的 SQL 文中で入力変数をバインドできます。また、FORALL ループの内側で EXECUTE IMMEDIATE 文を使用できます。次に構文を示します。

```
FORALL index IN lower bound..upper bound
   EXECUTE IMMEDIATE dynamic_string
   USING bind_argument | bind_argument(index)
      [, bind_argument | bind_argument(index)] ...
   [{RETURNING | RETURN} BULK COLLECT
      INTO bind_argument[, bind_argument ... ]];
```

動的文字列は、SELECT 文ではなく INSERT 文、UPDATE 文または DELETE 文を表す必要があります。

バルク動的バインドの例

動的な問合せでは、BULK COLLECT INTO 句を使用して定義変数をバインドできます。次の例のように、この句をバルク FETCH 文またはバルク EXECUTE IMMEDIATE 文に使用できます。

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR;
   TYPE NumList IS TABLE OF NUMBER;
   TYPE NameList IS TABLE OF VARCHAR2(15);
   emp_cv EmpCurTyp;
   empnos NumList;
   enames NameList;
   sals   NumList;
BEGIN
   OPEN emp_cv FOR 'SELECT empno, ename FROM emp';
   FETCH emp_cv BULK COLLECT INTO empnos, enames;
   CLOSE emp_cv;

   EXECUTE IMMEDIATE 'SELECT sal FROM emp'
      BULK COLLECT INTO sals;
END;
```

出力バインド変数を使用できるのは、INSERT、UPDATE および DELETE 文のみです。これらの文をバルク・バインドするには、BULK RETURNING INTO 句を使用します。この句は、EXECUTE IMMEDIATE でのみ使用できます。次に例を示します。

```
DECLARE
   TYPE NameList IS TABLE OF VARCHAR2(15);
   enames    NameList;
   bonus_amt NUMBER := 500;
   sql_stmt  VARCHAR(200);
BEGIN
   sql_stmt := 'UPDATE emp SET bonus = :1 RETURNING ename INTO :2';
   EXECUTE IMMEDIATE sql_stmt
      USING bonus_amt RETURNING BULK COLLECT INTO enames;
END;
```

SQL 文中で入力変数をバインドするには、次のように FORALL 文と USING 句を使用できます。ただし、SQL 文を問合せにすることはできません。

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   TYPE NameList IS TABLE OF VARCHAR2(15);
   empnos NumList;
   enames NameList;
BEGIN
   empnos := NumList(1,2,3,4,5);
   FORALL i IN 1..5
      EXECUTE IMMEDIATE
        'UPDATE emp SET sal = sal * 1.1 WHERE empno = :1
         RETURNING ename INTO :2'
         USING empnos(i) RETURNING BULK COLLECT INTO enames;
   ...
END;
```

動的 SQL 活用時のヒントと注意点

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。

パフォーマンスの向上

次の例では、Oracle は emp_id. の個別値ごとに異なるカーソルをオープンしています。これはリソースの競合とパフォーマンスの低下を起こすことがあります。

```
CREATE PROCEDURE fire_employee (emp_id NUMBER) AS
BEGIN
   EXECUTE IMMEDIATE
      'DELETE FROM emp WHERE empno = ' || TO_CHAR(emp_id);
END;
```

次に示すように、パフォーマンスはバインド変数を使用することで向上させることができます。これにより、Oracle は emp_id の異なる値に対して同じカーソルを再利用できるようになります。

```
CREATE PROCEDURE fire_employee (emp_id NUMBER) AS
BEGIN
   EXECUTE IMMEDIATE
      'DELETE FROM emp WHERE empno = :num' USING emp_id;
END;
```

任意の名前を持つスキーマ・オブジェクトをプロシージャで処理する方法

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「表名が無効です。*(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、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のみです。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

明示カーソルには、%FOUND、%ISOPEN、%NOTFOUND および %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 DML 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、6-33 ページの「カーソル属性の使用」を参照してください。

NULL を渡す方法

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 を未初期化の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_id INTEGER) IS
BEGIN
   EXECUTE IMMEDIATE 'DELETE FROM dept@' || db_link ||
      ' WHERE deptno = :num' USING dept_id;
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 VARCHAR2) RETURN INTEGER AS
   rows INTEGER;
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
   emp_count INTEGER;
BEGIN
   emp_count := 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 句を使用することで、ストアド・プロシージャをその実行者（現行ユーザー）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、8-50 ページの「実行者権限と定義者権限」を参照してください。

RESTRICT_REFERENCES プラグマの使用

SQL 文からコールされるファンクションは、副作用を制御するための特定の規則に従う必要があります。（8-9 ページの「PL/SQL サブプログラムの副作用の制御」を参照。）この規則に違反していないかどうかを確認するには、RESTRICT_REFERENCES プラグマを使用できます。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）に対する読込みや書込み、またはそのいずれも行っていないことを示します。（詳細は、『Oracle9*i* 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「データベース書込み禁止状態」（WNDS）および「データベース読込み禁止状態」（RNDS）の規則に違反します。これは、動的 SQL 文はコンパイル時ではなく実行時にチェックされるためです。EXECUTE IMMEDIATE 文内では、INTO 句のみがコンパイル時に RNDS の規則に違反していないかどうかがチェックされます。

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 で削除しないでください。

```
CREATE PROCEDURE calc_bonus (emp_id NUMBER) AS
BEGIN
   ...
   EXECUTE IMMEDIATE 'DROP PROCEDURE calc_bonus';
```

12

PL/SQL アプリケーションのチューニング

適切に設計されたアプリケーションでも、時間の経過によってパフォーマンスが低下することがあります。そのため、アプリケーションのメンテナンスの中でも、定期的なチューニングが重要な部分になります。この章では、パフォーマンス改善を目的とした小規模な調整方法について説明します。アプリケーションをチューニングすることで、要件に合った応答時間とスループットを継続して提供できます。

この章の項目は、次のとおりです。

PL/SQL のパフォーマンス問題の原因

PL/SQL ベースのアプリケーションのパフォーマンスが十分でない場合、その原因は通常、不適切な SQL 文の記述、プログラミング慣習の不徹底、PL/SQL の基本に対する不注意、共有メモリーの間違った使用などが考えられます。

PL/SQL プログラムでの不適切な SQL 文の記述

PL/SQL プログラムは比較的単純に見えますが、これは、ほとんどの処理を行う SQL 文にその複雑さが隠されているためです。このため、不適切な SQL 文の記述は、実行が低速になる主な原因となります。プログラムに不適切な SQL 文の記述が多く含まれている場合は、適切に記述された PL/SQL 文がいくつあっても役に立ちません。

このような SQL 文があるためにプログラムが低速になっている場合は、次の方法を使用して、その実行計画とパフォーマンスを分析してください。分析後に SQL 文を記述し直します。たとえば、問合せオプティマイザにヒントを使用すると、不要な全表スキャンなどの問題を排除できます。

- `EXPLAIN PLAN` 文
- TKPROF ユーティリティを持つ SQL トレース機能
- Oracle Trace 機能

これらの方法の詳細は、『Oracle9*i* データベース・パフォーマンス・プランニング』を参照してください。

プログラミング慣習の不徹底

プログラミング慣習の不徹底は、しばしばスケジュールの混乱から生じることがあります。このような状況では、経験豊富なプログラマでもパフォーマンスの悪いコードを記述してしまう場合があります。

特定のタスクにどれほど適したプログラミング言語であっても、不適切に記述されたサブプログラム（低速のソートまたは検索機能など）があると、パフォーマンスが低下する場合があります。アプリケーションからきわめて頻繁にコールされるサブプログラムが、膨大なターゲットを持つ検索ファンクションであるとします。そのファンクションは、ハッシュまたはバイナリ検索として記述できますが、そうではなく線形検索であるとパフォーマンス全体が低下します。

また、プログラミング慣習の不徹底には、使用されない変数の宣言、ファンクションとプロシージャへの不要なパラメータの受渡し、ループ内への不要な初期化や計算の配置などもあります。

組込みファンクションの重複

PL/SQL には、REPLACE、TRANSLATE、SUBSTR、INSTR、RPAD および LTRIM など、高度に最適化されたファンクションが多数用意されています。独自のバージョンのコードを記述しないでください。組込みファンクションの方が効率的です。組込みファンクションが必要以上の機能を持っていても、それらの機能の一部をコーディングせずにそのまま使用してください。

不十分な条件制御文

論理式を評価するときに、PL/SQL では短絡評価を使用します。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 になるため、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。ただし、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 の場合にのみコールされます（AND は、オペランドが両方とも TRUE の場合にのみ TRUE を戻すため）。

EXIT-WHEN 文も同じ考え方です。

暗黙的なデータ型変換

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。

暗黙的な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ため、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。ただし、浮動小数点リテラル 15.0 は 22 バイトの Oracle の数値で表されるため、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
   ...
END;
```

次にもう 1 つ例を示します。

```
DECLARE
   c CHAR(5);
BEGIN
   c := 25;    -- converted
   c := '25';  -- not converted
   ...
END;
```

数値データ型の不適切な宣言

NUMBER 型とそのサブタイプは 22 バイトのデータベース形式の数値で、パフォーマンスよりも移植性と任意の位取り / 精度に重点を置いて設計されています。整変数を宣言する必要がある場合は、PLS_INTEGER データ型を使用します。PLS_INTEGER は、最も効率的な数値型です。これは、PLS_INTEGER 型の値の方が INTEGER 型または NUMBER 型の値よりもメモリー要件が小さいためです。また、PLS_INTEGER 演算はマシン算術計算を使用するため、ライブラリ算術計算を使用する 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。

不要な NOT NULL 制約

PL/SQL では、NOT NULL 制約を使用すると、パフォーマンス・コストがかかります。次の例を考えます。

```
PROCEDURE calc_m IS
   m NUMBER NOT NULL := 0;
   a NUMBER;
   b NUMBER;
BEGIN
   ...
   m := a + b;
   ...
END;
```

m には NOT NULL の制約があるため、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 ではない場合、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。ただし、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 IF;
END;
```

サブタイプ NATURALN と POSTIVEN は、NOT NULL と定義されています。したがって、それらを使用すると、同じパフォーマンス・コストがかかります。

VARCHAR2 変数の宣言のサイズ

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。ただし、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用することになります。

PL/SQL プログラムでの共有メモリーの間違った使用

パッケージ・サブプログラムを初めてコールすると、パッケージ全体が共有メモリー・プールにロードされます。パッケージ内の関連するサブプログラムに対する 2 度目以降のコールでは、ディスク I/O が必要ないため実行速度が向上します。ただし、パッケージがメモリーからエージ・アウトされた場合は、再参照する前に再ロードする必要があります。

共有メモリー・プールのサイズを適切に設定すると、パフォーマンスを改善できます。頻繁に使用するパッケージを十分に保持でき、しかもメモリーが浪費されないサイズになるように設定してください。

確保されたパッケージ

パフォーマンスを改善するもう 1 つの方法は、使用頻度の高いパッケージを共有メモリー・プールに確保することです。パッケージを確保すると、Oracle で通常使用される LRU（Least Recently Used）アルゴリズムでもエージ・アウトされることはありません。パッ

ケージは、プールの占有状態やパッケージへのアクセス頻度に関係なく、メモリーに残ります。

オラクル社が提供するパッケージ DBMS_SHARED_POOL を使用すると、パッケージを確保できます。詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

逐次再使用可能パッケージ

メモリー使用を管理しやすくするために、PL/SQL では SERIALLY_REUSABLE プラグマを用意しています。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）の間のみその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。

本体なしのパッケージの場合は、次の構文を使用して、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 PACKAGE pkg1 IS
   PRAGMA SERIALLY_REUSABLE;
   num NUMBER := 0;
   PROCEDURE init_pkg_state(n NUMBER);
   PROCEDURE print_pkg_state;
END pkg1;
/
CREATE PACKAGE BODY pkg1 IS
   PRAGMA SERIALLY_REUSABLE;
   PROCEDURE init_pkg_state (n NUMBER) IS
   BEGIN
      pkg1.num := n;
   END;
   PROCEDURE print_pkg_state IS
```

```
   BEGIN
      dbms_output.put_line('Num: ' || pkg1.num);
   END;
END pkg1;
/
BEGIN
   /* Initialize package state. */
   pkg1.init_pkg_state(4);
   /* On same server call, print package state. */
   pkg1.print_pkg_state;  -- prints 4
END;
/
-- subsequent server call
BEGIN
   -- the package's public variable is initialized
   -- to the default value automatically
   pkg1.print_pkg_state;  -- prints 0
END;
```

詳細は、『Oracle9*i* 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。

PL/SQL のパフォーマンス問題の識別

開発する PL/SQL アプリケーションが大きくなるほど、パフォーマンスの問題を切り出すのは困難になります。そのため、PL/SQL には、実行時の動作をプロファイルし、パフォーマンスのボトルネックを識別できるように、プロファイラ API が用意されています。また、サーバー上でプログラムの実行をトレースするためのトレース API もあります。Oracle Trace を使用すると、サブプログラム別または例外ごとに実行をトレースできます。

プロファイラ API: パッケージ DBMS_PROFILER

プロファイラ API は、PL/SQL パッケージ DBMS_PROFILER として実装され、ランタイム統計を収集して保存するサービスを提供します。情報はデータベース表に格納され、後で問合せできます。たとえば、PL/SQL の各行とサブプログラムの実行にかかる所要時間を知ることができます。

プロファイラを使用するには、プロファイル・セッションを開始し、十分な範囲のコードを取得できるまでアプリケーションを実行し、収集されたデータをデータベースにフラッシュし、プロファイル・セッションを停止します。通常のセッションでは、次の手順で操作します。

1. パッケージ DBMS_PROFILER 内のプロシージャ start_profiler をコールして、プロファイラ・セッションにコメントを対応付けします。
2. プロファイル対象のアプリケーションを実行します。
3. プロシージャ flush_data を繰り返しコールして増分データを保存し、データ構造に割り当てられたメモリーを解放します。
4. プロシージャ stop_profiler をコールして停止します。

プロファイラにより、プログラムの実行がトレースされ、各行および各サブプログラムの所要時間が計算されます。収集されたデータを使用してパフォーマンスを改善できます。たとえば、低速のサブプログラムに改善の重点を置くことができます。

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

収集されたパフォーマンス・データの分析

次の手順は、特定のコード・セグメントの実行や特定のデータ構造へのアクセスに、他よりも時間がかかっている原因を判断することです。パフォーマンス・データを問い合せて問題を探します。ほとんどの実行時間を占めているサブプログラムやパッケージに重点を置いて、SQL 文、ループおよび再帰関数など、考えられるパフォーマンスのボトルネックを調べてください。

トレース・データを使用したパフォーマンス改善

分析結果に基づいて低速のアルゴリズムを書き直します。たとえば、データが急増したために、線形検索をバイナリ検索に置き換える必要が出てくることがあります。また、不適切な

データ構造によって生じた非効率な部分を探し、必要に応じてそのデータ構造を置き換えてください。

トレース API: パッケージ DBMS_TRACE

大型で複雑なアプリケーションの場合は、サブプログラム間のコールを追跡するのは困難です。トレース API でコードをトレースすると、サブプログラムの実行順序を確認できます。トレース API は PL/SQL パッケージ DBMS_TRACE として実装され、サブプログラムまたは例外ごとに実行をトレースするサービスを提供します。

トレースを使用するには、トレース・セッションを開始し、アプリケーションを実行してから、トレース・セッションを停止します。プログラムを実行すると、トレース・データが収集され、データベース表に格納されます。通常のセッションでは、次の手順で操作します。

1. オプションで、トレース・データの収集対象となる特定のサブプログラムを選択します。
2. パッケージ DBMS_TRACE 内のプロシージャ set_plsql_trace をコールします。
3. トレースするアプリケーションを実行します。
4. プロシージャ clear_plsql_trace をコールして停止します。

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

トレースの制御

大きいアプリケーションをトレースすると、対象のデータが生成されて管理が困難になることがあります。必要であれば、トレースを起動する前に、トレース・データ収集用の特定のサブプログラムを選択し、収集されるデータ量を制限できます。

また、トレース・レベルも選択できます。たとえば、すべてのサブプログラムと例外をトレースするか、選択したサブプログラムと例外をトレースするように指定できます。

PL/SQL のパフォーマンス・チューニング機能

アプリケーションを低速にしていた問題を解決した後に、次の PL/SQL 機能およびテクニックを使用できます。

- システム固有の動的 SQL を使用した PL/SQL パフォーマンスのチューニング
- バルク・バインドを使用した PL/SQL パフォーマンスのチューニング
- NOCOPY コンパイラ・ヒントを使用した PL/SQL パフォーマンスのチューニング
- RETURNING 句を使用した PL/SQL パフォーマンスのチューニング
- 外部ルーチンを使用した PL/SQL パフォーマンスのチューニング
- オブジェクト型とコレクションを使用した PL/SQL パフォーマンスの改善
- システム固有の実行のための PL/SQL コードのコンパイル

これらの使用しやすい機能により、アプリケーションを大幅にスピードアップできます。

システム固有の動的 SQL を使用した PL/SQL パフォーマンスのチューニング

たとえば汎用目的のレポート・ライターなど、プログラムによっては、様々な 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、実行時まで不明です。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。

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、DBMS_SQL パッケージより使用しやすく、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 文と対応付けます。

```
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 11 章を参照してください。

バルク・バインドを使用した PL/SQL パフォーマンスのチューニング

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 文を実行する場合は、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 4 つ以上のデータベース行に影響する場合は、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、一度のみ 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 文に移動します。
- 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 句を組み込みます。
- 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します。
- 特定の行の DML 操作が失敗しても重大な問題とならない場合は、FORALL 文にキーワード SAVE EXCEPTIONS を組み込み、%BULK_EXCEPTIONS 属性を使用して後続のループでエラーをレポートまたはクリーン・アップします。

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。

バルク・バインドの詳細は、5-38 ページの「バルク・バインドを使用したコレクションのループ・オーバーヘッドの削減」を参照してください。

NOCOPY コンパイラ・ヒントを使用した PL/SQL パフォーマンスのチューニング

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に、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、8-17 ページの「NOCOPY コンパイラ・ヒントを使用した大型データ構造の受渡し」を参照してください。

RETURNING 句を使用した PL/SQL パフォーマンスのチューニング

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 文および 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すみます。

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;
-- Now do computations involving name and new_sal
END;
```

外部ルーチンを使用した PL/SQL パフォーマンスのチューニング

PL/SQL には、他の言語で記述さ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ます。他の言語ですでに作成され利用できるように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、再使用性、効率、モジュール性が高まります。

PL/SQL は、SQL トランザクション処理に特化されています。作業の中には、マシン精度の計算に適した 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。

実行を高速化するには、計算専用プログラムを C 言語で再作成できます。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ため、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 言語で作成して Dynamic Link Library（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実装された）別個のアドレス空間で実行されます。

詳細は、『Oracle9*i* 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。

オブジェクト型とコレクションを使用した PL/SQL パフォーマンスの改善

コレクション型（第 5 章を参照）およびオブジェクト型（第 10 章を参照）を使用すると、現実のデータをモデル化できるため、生産性が向上します。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、適切に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の宣言、パラメータとしての受渡し、データベースへの格納や取出しができます。また、オブジェクト型を使用すると、データに対する操作をカプセル化することで、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ため、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・アーキテクチャと統合されているため、Oracle リリース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。

システム固有の実行のための PL/SQL コードのコンパイル

PL/SQL プロシージャをコンパイルして、共有ライブラリに常駐するシステム固有のコードにすると、プロシージャをスピードアップできます。プロシージャは C 言語のコードに変換されてから、通常の C コンパイラでコンパイルされ、Oracle プロセスにリンクされます。このテクニックは、提供されている Oracle パッケージと独自に記述したプロシージャの両方に使用できます。この方法でコンパイルしたプロシージャは、共有サーバー構成（以前はマルチスレッド・サーバーと呼ばれていた）などのサーバー環境で動作します。

このテクニックでは、PL/SQL からコールされる SQL 文は高速にならないため、SQL の実行に費やす時間の少ない計算集中型の PL/SQL プロシージャで最も効率的です。

このテクニックを使用して 1 つ以上のプロシージャをスピードアップする手順は、次のとおりです。

1. 提供されている Make ファイルを更新し、システムの適切なパスや他の値を入力します。この Make ファイルのパスは `$ORACLE_HOME/plsql/spnc_makefile.mk` です。
2. `ALTER SYSTEM` または `ALTER SESSION` コマンドを使用するか、初期化ファイルを更新して、値 `NATIVE` を組み込むようにパラメータ `PLSQL_COMPILER_FLAGS` を設定します。デフォルト設定では値 `INTERPRETED` が組み込まれます。このキーワードをパラメータ値から削除する必要があります。
3. 次のいずれかの方法を使用して 1 つ以上のプロシージャをコンパイルします。
   - `ALTER PROCEDURE` または `ALTER PACKAGE` コマンドを使用して、プロシージャまたはパッケージ全体を再コンパイルします。
   - プロシージャを削除して再作成します。
   - `CREATE OR REPLACE` を使用してプロシージャを再コンパイルします。
   - オラクル社が提供するパッケージ・セットを設定する SQL*Plus スクリプトの 1 つを実行します。
   - `PLSQL_COMPILER_FLAGS=NATIVE` を指定し、構成済みの初期化ファイルを使用してデータベースを作成します。データベースの作成中に、`UTLIRP` スクリプトが実行され、オラクル社が提供するパッケージがすべてコンパイルされます。
4. プロセスが機能するかどうかを確認するために、データ・ディクショナリを問い合せて、プロシージャがシステム固有の実行用にコンパイルされたかどうかを調べることができます。既存のプロシージャがシステム固有の実行用にコンパイルされたかどうかをチェックするには、データ・ディクショナリ・ビュー `USER_STORED_SETTINGS`、`DBA_STORED_SETTINGS` および `ALL_STORED_SETTINGS` を問い合せます。たとえば、プロシージャ `MY_PROC` のステータスをチェックするには、次のように入力します。

```
SELECT param_value FROM user_stored_settings WHERE
  param_name = 'PLSQL_COMPILER_FLAGS'
  and object_name = 'MY_PROC';
```

PARAM_VALUE 列の値は、プロシージャがシステム固有の実行用にコンパイルされていれば NATIVE、それ以外の場合は INTERPRETED となります。

プロシージャをコンパイルして共有ライブラリに入れると、Oracle プロセスに自動的にリンクされます。データベースを再起動したり、共有ライブラリを別の場所に移動する必要はありません。ストアド・プロシージャがすべてデフォルトの方法（INTERPRETED）でコンパイルされたか、すべてシステム固有の実行用にコンパイルされたか、または両者を組み合わせてコンパイルされたかにかかわらず、相互の間でコールできます。

PLSQL_COMPILER_FLAGS の設定はプロシージャごとにライブラリ・ユニット内に格納されるため、システム固有の実行用にコンパイルされたプロシージャは、依存する表が再作成された場合など、無効になった後の再コンパイル時にも同じ方法でコンパイルされます。

PL/SQL のシステム固有のコンパイルの動作を制御するには、ALTER SYSTEM または ALTER SESSION コマンドを使用する方法と、これらのパラメータを初期化ファイル内で変更する方法があります。

- PLSQL_COMPILER_FLAGS
- PLSQL_NATIVE_LIBRARY_DIR（セキュリティ上の理由で、ALTER SESSION での設定は不可）
- PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT
- PLSQL_NATIVE_MAKE_UTILITY
- PLSQL_NATIVE_MAKE_FILE_NAME

> **関連項目：** 初期化パラメータとデータ・ディクショナリ・ビューの詳細は、『Oracle9*i* データベース・リファレンス』を参照してください。

システム固有の実行のための PL/SQL プロシージャのコンパイル例

```
connect scott/tiger;
set serveroutput on;
alter session set plsql_native_library_dir='/home/orauser/lib';
alter session set plsql_native_make_utility='gmake';
alter session set plsql_native_make_file_name='/home/orauser/spnc_makefile.mk';
alter session set plsql_compiler_flags='NATIVE';
create or replace procedure hello_native_compilation
as
begin
  dbms_output.put_line('Hello world');
  select sysdate from dual;
end;
```

プロシージャのコンパイル時に、各種のコンパイルおよびリンク・コマンドが実行されることがわかります。プロシージャはただちにコールできるようになり、Oracle プロセス内で共有ライブラリとして直接実行されます。

システム固有のコンパイルの制限

- パッケージ仕様部をシステム固有の実行用にコンパイルする場合は、対応する本体も同じ設定を使用してコンパイルする必要があります。
- PL/SQL 用のデバッグ・ツールでは、システム固有の実行用にコンパイルされたプロシージャは処理されません。
- 多数のプロシージャとパッケージ（通常は 5000 以上）をシステム固有の実行用にコンパイルすると、単一のディレクトリに多数の共有オブジェクトができるため、システムのパフォーマンスに影響することがあります。この場合は、データベースの作成や PL/SQL パッケージまたはプロシージャのコンパイルを行う前に、初期化ファイル内で初期化パラメータ `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` を設定できます。このパラメータを適切な値（通常は 1000 前後）に設定してください。その後、`PLSQL_NATIVE_LIBRARY_DIR` パラメータで指定したディレクトリの下にサブディレクトリを作成します。サブディレクトリ名には、`d0`、`d1`、`d2 ... d999` など、サブディレクトリ数に指定した値の範囲内の名前を使用します。プロシージャをシステム固有の実行用にコンパイルすると、PL/SQL コンパイラにより DLL がこれらのサブディレクトリ間で自動的に分散されます。

13

PL/SQL の言語要素

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、パラメータおよびその他の言語要素を並べて PL/SQL 文を作成する方法を示します。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。

この章の項目は、次のとおりです。

構文図の読み方

PL/SQL 文の構文を確認する場合は、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。

この図は、バッカス正規形（BNF）の結果を図で表したものです。この図では、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。

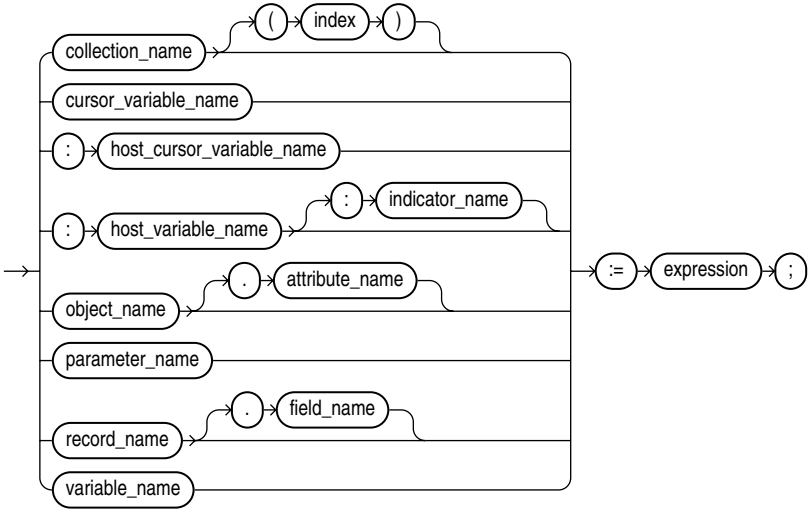
各図は、構文の要素を定義しています。図を通るパスは、それぞれがその要素のフォームとして考えられ るものを表しています。矢印の向きに進んでください。線がループ状になっている場合は、そのループ内の要素を繰り返すことができます。

代入文

代入文は、変数、フィールド、パラメータまたは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、2-22 ページの「変数の代入」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

attribute_name

オブジェクト型の属性を指定します。名前はそのオブジェクト型の中で固有である必要があります（他のオブジェクト型内では使用できます）。属性の宣言内では、代入演算子または DEFAULT 句を使用しての属性の初期化はできません。また、属性に NOT NULL 制約を課すことはできません。

collection_name

現行の有効範囲のうち、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

現行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のみです。

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、13-69 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。値とターゲットのデータ型には互換性が必要です。

field_name

ユーザー定義のレコードまたは %ROWTYPE レコード内のフィールドを指定します。

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付けてください。

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付けてください。

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。

indicator_name

PL/SQL ホスト環境で宣言され、PL/SQL に渡されるインジケータ変数を識別します。インジケータ変数には、接頭辞としてコロンを付ける必要があります。インジケータ変数は、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、インジケータ変数を使用し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。

object_name

現行の有効範囲のうち、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。

parameter_name

代入文が使用さ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。

record_name

現行の有効範囲のうち、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。

variable_name

現行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。

使用上の注意

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でください。

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、FALSE および NULL のみです。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集計代入ができます。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;
```

次の構文を使用すると、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

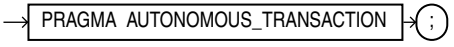
定数と変数、式、SELECT INTO 文

AUTONOMOUS_TRANSACTION プラグマ

AUTONOMOUS_TRANSACTION 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に PL/SQL コンパイラに指示します。自律型トランザクションは、別のトランザクション、メイン・トランザクション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。詳細は、6-52 ページの「自律型トランザクションによる独立した作業単位の実行」を参照してください。

構文

autonomous_transaction_pragma



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のみです。

使用上の注意

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

パッケージのすべてのサブプログラム（またはオブジェクト型のすべてのメソッド）を自律型としてマークするためにプラグマを使用することはできません。自律型としてマークできるのは、個々のルーチンのみです。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。

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、リソースまたは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。

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。また、通常のトリガーとは異なり、自律型トリガーはシステム固有の動的 SQL を使用して、DDL 文（CREATE および DROP など）を実行できます

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のみです。

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。

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。

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、自律型トランザクションがアクセスしようとすると、デッドロックが発生します。この場合、Oracle は自律型トランザクションで例外を呼び出します。例外が未処理になった場合、自律型トランザクションはロールバックされます。

コミットまたはロールバックせずにアクティブな自律型トランザクションを終了しようとすると、Oracle は例外を呼び出します。例外が未処理になった場合、トランザクションはロールバックされます。

例

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;
```

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、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;
```

関連項目

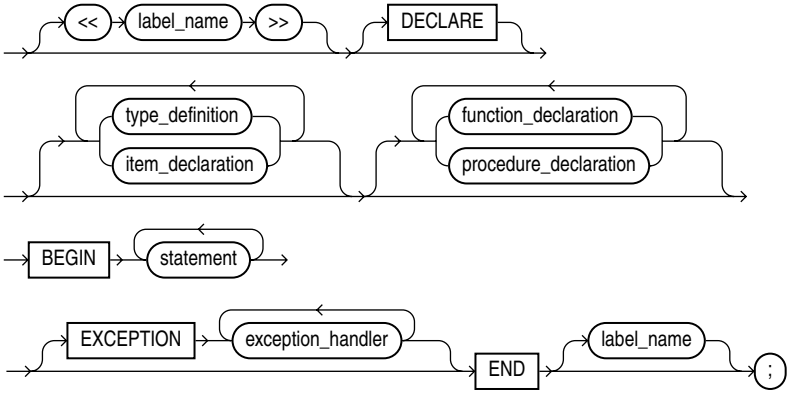
EXCEPTION_INIT プラグマ、RESTRICT_REFERENCES プラグマ、SERIALLY_REUSABLE プラグマ

ブロック

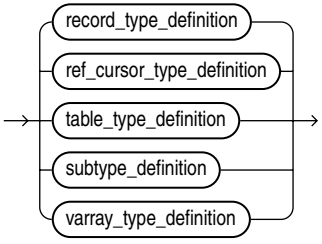
PL/SQL におけるプログラムの基本単位はブロックです。PL/SQL ブロックは、キーワード DECLARE、BEGIN、EXCEPTION および END で定義します。これらのキーワードは、ブロックを宣言部、実行部、例外処理部に分けます。このうち必ず存在する必要があるのは実行部のみです。ブロックの中では、実行可能文を置ける場所ならば別のブロックをネストできます。詳細は、1-3 ページの「ブロック構造」および 2-19 ページの「PL/SQL の識別子の有効範囲と可視性」を参照してください。

構文

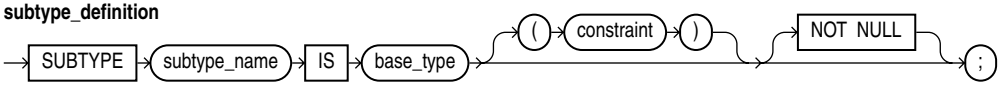
plsql_block



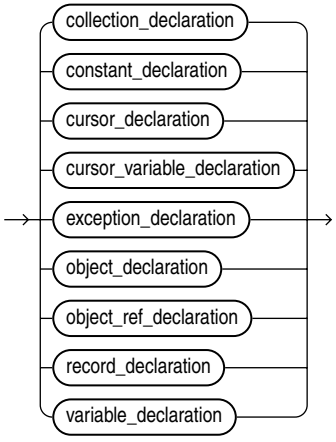
type_definition



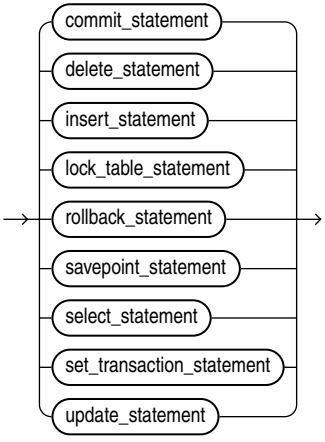
subtype_definition

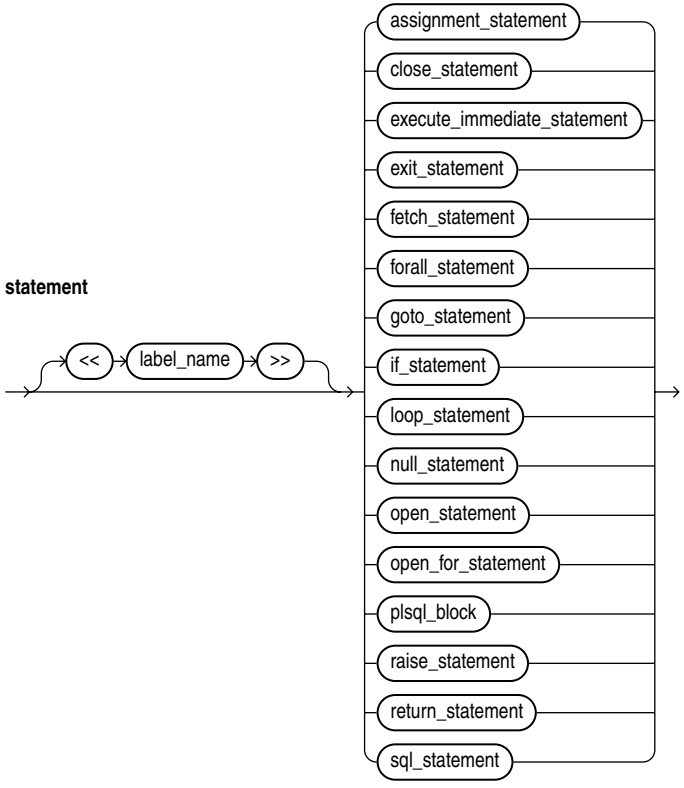


item_declaration



sql_statement




statement
<<
label_name
>>
assignment_statement
close_statement
execute_immediate_statement
exit_statement
fetch_statement
forall_statement
goto_statement
if_statement
loop_statement
null_statement
open_statement
open_for_statement
plsql_block
raise_statement
return_statement
sql_statement

キーワードとパラメータの説明

base_type
これは任意のスカラーまたは CHAR、DATE または RECORD などのユーザー定義の PL/SQL データ型指定子です。

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ている必要があります。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ることが必要です。NULL 文はこの条件を満たします。

collection_declaration
コレクションを宣言します（索引付き表、ネストした表または VARRAY）。collection_declaration の構文は、13-27 ページの「コレクション」を参照してください。

constant_declaration
定数を宣言します。constant_declaration の構文は、13-36 ページの「定数と変数」を参照してください。

constraint
これは CHAR や NUMBER などの制約できるデータ型にのみ適用されます。文字データ型の場合は、最大サイズのバイト数を指定します。数値データ型の場合は、精度と位取りの最大値を指定します。

cursor_declaration
明示カーソルを宣言します。cursor_declaration の構文は、13-51 ページの「カーソル」を参照してください。

cursor_variable_declaration
カーソル変数を宣言します。cursor_variable_declaration の構文は、13-45 ページの「カーソル変数」を参照してください。

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行のブロックとそのすべてのサブブロックにのみ存在し、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にする必要があります。IF 文の END IF、または LOOP 文の END LOOP のいずれも、キーワード END のか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ないことに注意してください。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。

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行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 7 章を参照してください。

exception_declaration

例外を宣言します。exception_declaration の構文は、13-60 ページの「例外」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、13-60 ページの「例外」を参照してください。

function_declaration

ファンクションを宣言します。function_declaration の構文は、13-88 ページの「ファンクション」を参照してください。

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 を使用する場合は、二重の山カッコで囲み、ブロックの先頭に置いてください。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、サブブロック内での宣言が優先され、ブロック・ラベルを使用して参照を修飾しなければ、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。次に例を示します。

```
<<outer>>
DECLARE
   x INTEGER;
BEGIN
   ...
   DECLARE
      x INTEGER;
   BEGIN
      ...
      IF x = outer.x THEN  -- refers to global x
         ...
      END IF;
   END;
END outer;
```

object_declaration

オブジェクト（オブジェクト型のインスタンス）を宣言します。object_declaration の構文は、13-116 ページの「オブジェクト型」を参照してください。

procedure_declaration

プロシージャを宣言します。procedure_declaration の構文は、13-140 ページの「プロシージャ」を参照してください。

record_declaration

ユーザー定義のレコードを宣言します。record_declaration の構文は、13-148 ページの「レコード」を参照してください。

statement

これは、アルゴリズムを作成するために使用する（宣言文ではなく）実行可能文です。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かぎり、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。

subtype_name

任意のスカラーまたは CHAR、DATE または RECORD などのユーザー定義の PL/SQL データ型指定子を使用して定義したユーザー定義のサブタイプを識別します。

variable_declaration

変数を宣言します。variable_declaration の構文は、13-36 ページの「定数と変数」を参照してください。

PL/SQL は、データ操作、カーソル制御および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 および 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。

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用し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

CASE 文

CASE 文を使用すると、一連の文を選択して実行できます。順序を選択するために、CASE 文は選択子（複数の選択肢から 1 つを選択するために値が使用される式）を使用します。検索 CASE 文の場合は、複数の検索条件が使用されます。

構文

searched_case_statement ::=

```
[ <<label_name>> ]
CASE { WHEN boolean_expression THEN {statement;} ... }...
[ ELSE {statement;}... ]
END CASE [ label_name ];
```

simple_case_statement ::=

```
[ <<label_name>> ]
CASE case_operand
{ WHEN when_operand THEN {statement;} ... }...
[ ELSE {statement;}... ]
END CASE [ label_name ];
```

キーワードとパラメータの説明

単純な CASE 文の場合、CASE オペランドと WHEN オペランドの値は、BLOB、BFILE、オブジェクト型、PL/SQL レコード、索引付き表、VARRAY またはネストした表以外であれば、任意の PL/SQL データ型にできます。

ELSE 句を省略すると、デフォルトのアクションが代用されます。CASE 文の場合、条件が一致しなければ、デフォルトで CASE_NOT_FOUND 例外が呼び出されます。CASE 式の場合は、デフォルトで NULL が戻されます。

使用上の注意

各 WHEN 句はそれぞれ 1 度のみ実行されます。

WHEN 句は順番に実行されます。

一致する WHEN 句が 1 つでも見つかると、後続の WHEN 句は実行されません。

WHEN 句の実行順序は前述の規則によって定義されるため、WHEN 句の文でデータベースを変更し、非決定的な関数を実行できます。

C 言語の switch 文のような失敗はありません。WHEN 句が一致し、その文が実行されると、CASE 文は終了します。

CASE 文は、オプションごとにアクションが異なる場合に適しています。複数の値から選択して変数に代入すれば、かわりに CASE 式を使用して代入文を記述できます。

例

単純な CASE 文の例を次に示します。WHEN 句の後に複数の文を使用できることに注意してください。

```
DECLARE
  n number;
BEGIN
  CASE n
    WHEN 1 THEN dbms_output.put_line('n = 1');
    WHEN 2 THEN
      dbms_output.put_line('n = 2');
      dbms_output.put_line('That implies n > 1');
    ELSE dbms_output.put_line('n is some other value.');
  END CASE;
END;
```

検索 CASE 文の例を次に示します。WHEN 句では、すべての条件に同じ変数をテストしたり同じ演算子を使用するのではなく、それぞれ異なる条件を使用できることに注意してください。この例では ELSE 句を使用していないため、WHEN 条件が 1 つも満たされない場合は例外が呼び出されます。

```
DECLARE
  quantity NUMBER;
  projected NUMBER;
  needed NUMBER;
BEGIN
<<here>>
  CASE
    WHEN quantity is null THEN
      dbms_output.put_line('Quantity not available');
    WHEN quantity + projected >= needed THEN
      dbms_output.put_line('Quantity ' || quantity ||
```

```
        ' should be enough if projections are met.');
    WHEN quantity >= 0 THEN
      dbms_output.put_line('Quantity ' || quantity || ' is probably not enough.');
  END CASE here;
  EXCEPTION
    WHEN CASE_NOT_FOUND THEN
      dbms_output.put_line('Somehow quantity must be less than 0.')
END;
```

関連項目

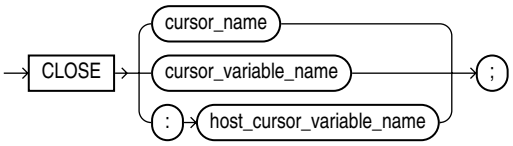
4-3 ページの「条件制御 : IF 文および CASE 文」、2-31 ページの「CASE 式」および『Oracle9*i* SQL リファレンス』の NULLIF 式および COALESCE 式

CLOSE 文

CLOSE 文を使用すると、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フェッチすることができません。詳細は、6-6 ページの「カーソル管理」を参照してください。

構文

close_statement



キーワードとパラメータの説明

cursor_name

現行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

現行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付けてください。

使用上の注意

一度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 文を使用します。カーソルを一度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には、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、最終行がフェッチされ、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

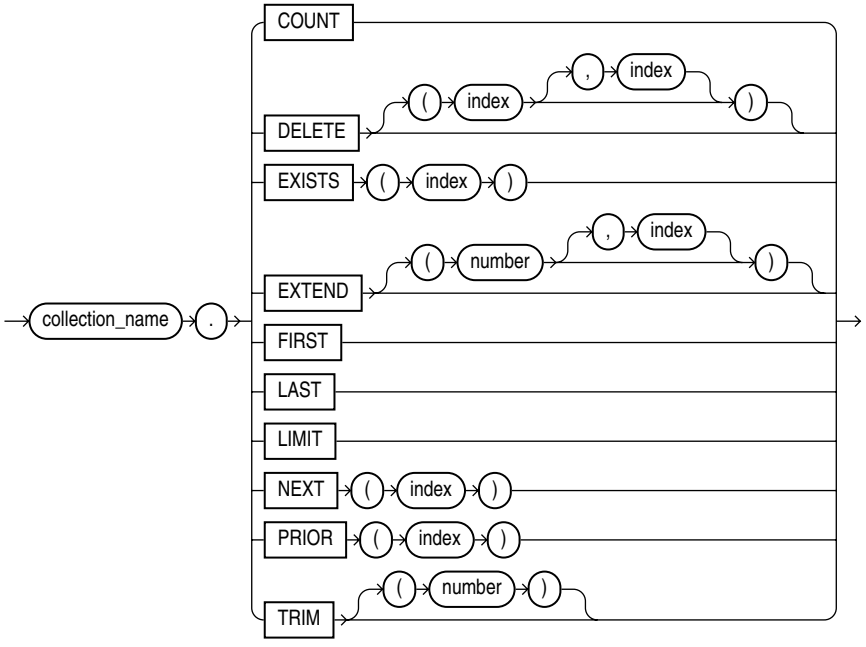
コレクション・メソッドとは、コレクションに対する操作を実行するための、ドット表記法を使用してコールされる組込みファンクションまたはプロシージャです。メソッド EXISTS、COUNT、LIMIT、FIRST、LAST、PRIOR、NEXT、EXTEND、TRIM および DELETE は、コードを一般化し、使用しやすくして、アプリケーションをメンテナンスしやすくするのに役立ちます。

EXISTS、COUNT、LIMIT、FIRST、LAST、PRIOR および NEXT は、式の一部として使用されるファンクションです。EXTEND、TRIM および DELETE は、文として使用されるプロシージャです。EXISTS、PRIOR、NEXT、TRIM、EXTEND および DELETE は整数パラメータを取ります。文字列キーを持つ結合配列の場合、EXISTS、PRIOR、NEXT および DELETE は、VARCHAR2 パラメータも取ります。EXTEND と TRIM は、索引付き表では使用できません。

詳細は、5-28 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

現行の有効範囲のうち、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。コレクションの現行のサイズは不明の場合があるため、そのようなときに役立ちます。COUNT は、整数式が使用できる位置ならどこでも使用できます。

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。ただし、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、索引付き表また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 ～ 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 か n が NULL である場合、DELETE(m,n) は何も実行しません。

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。それ以外の場合、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用します。また、EXISTS を使用すると、存在しない要素を参照した場合に発生する例外を回避できます。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。索引付き表で EXTEND を使用することはできません。

FIRST、LAST

FIRST と LAST は、それぞれコレクション内の最初と最後（最小と最大）の添字値を戻します。通常、添字値は整数ですが、結合配列では文字列の場合もあります。コレクションが空の場合、FIRST および 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 つのみの場合、FIRST と LAST は同じ添字値を戻します。

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。ただし、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。

index

ほとんどの場合は結果が整数になる（または暗黙的に整数に変換される）式、文字列キーを使用して宣言した結合配列の場合は文字列です。

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、コレクションの索引 n の前の添字を戻します。NEXT(n) は、索引 n の後の添字を戻します。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。索引付き表で TRIM を使用することはできません。

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。

使用上の注意

コレクション・メソッドは SQL 文では使用できません。使用すると、コンパイル・エラーになります。

基本構造的に NULL であるコレクションに適用されるのは EXISTS のみです。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 を使用すると、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 を使用すると、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 を使用して、基本構造的に 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ため、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のみでその要素を置き換えることができます。ただし、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でください。ネストした表は、固定サイズの配列のように扱って DELETE のみを使用するか、またはスタックのように扱って TRIM と EXTEND のみを使用することをお薦めします。

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 などをそのよう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導出されます。

例

次の例では、NEXT を使用して、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 が存在する場合にのみ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 を使用して、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 を使用して、VARRAY projects にさらに 20 の要素を追加できるかどうかを判断します。

```
IF (projects.COUNT + 20) < projects.LIMIT THEN
   -- add 20 more elements
```

関連項目

コレクション、ファンクション、プロシージャ

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、結合配列、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、結合配列（以前の「PL/SQL 表」または「索引付き表」）の機能を拡張します。

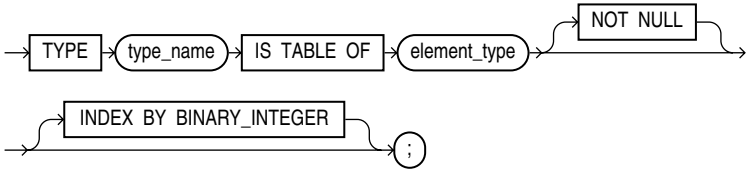
コレクションは、ほとんどの第 3 世代のプログラミング言語で見られる配列と同様の働きをします。コレクションにあるのは、1 次元のみです。ほとんどのコレクションは整数で索引付けされますが、結合配列では文字列も使用できます。マルチディメンション配列のモデルを作成する場合には、項目として他のコレクションを持つコレクションを宣言できます。

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、それらを使用して、データの列をデータベースの表に出し入れしたり、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できます。

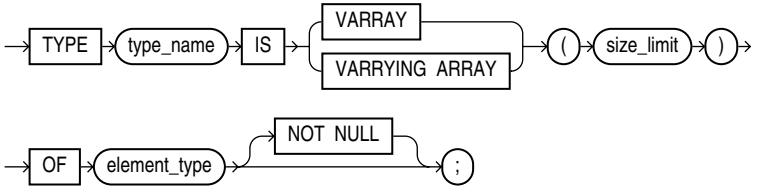
詳細は、5-7 ページの「コレクション型の定義」を参照してください。

構文

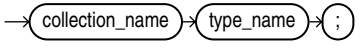
table_type_definition



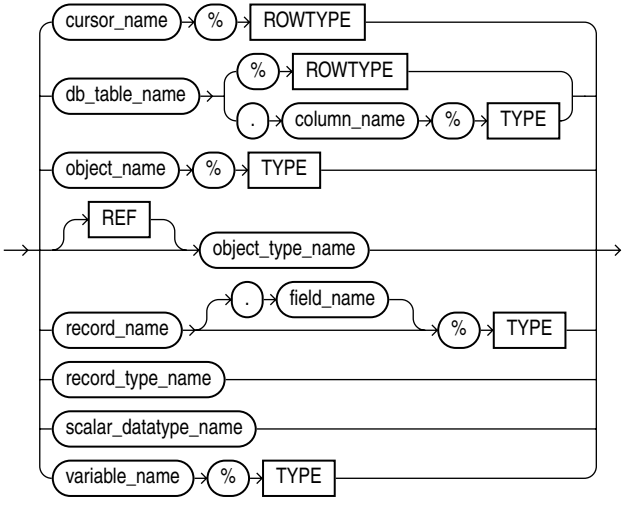
varray_type_definition



collection_declaration



element_type



キーワードとパラメータの説明

element_type

これは、BINARY_INTEGER、BOOLEAN、LONG、LONG RAW、NATURAL、NATURALN、PLS_INTEGER、POSITIVE、POSITIVEN、REF CURSOR、SIGNTYPE または STRING 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。

INDEX BY type_name

このオプションの句によって結合配列を定義します。システムで添字値を順に定義するのではなく、ユーザーが使用する添字値を指定します。

type_name には、BINARY_INTEGER、PLS_INTEGER または文字列型（VARCHAR2 など）を指定できます。

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。

type_name

データ型指定子 TABLE または VARRAY を使用して定義されたユーザー定義のコレクション型を識別します。

使用上の注意

ネストした表は、索引付き表の機能を拡張したものであるため、いくつかの点で異なります。5-6 ページの「ネストした表と結合配列の選択」を参照してください。

すべての要素参照には、コレクション名およびカッコで囲まれた 1 つ以上の添字が含まれており、この添字によって、処理される要素が決まります。負の添字を持つことができる結合配列を除き、コレクションの添字の下限は 1（固定）です。マルチレベル・コレクションの添字は、任意の順序で評価されます。添字に、別の添字の値を変更する式が含まれている場合の結果は、未定義になります。

3 つのすべてのコレクション型は、任意の PL/SQL ブロック、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。ただし、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 型のみです。

結合配列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（添字が連続しています）。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。

最初、結合配列は疎です。そのため、たとえば、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、参照データを一時変数に格納できます。

コレクションは、通常の有効範囲とインスタンス化の規則に従います。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。

ネストした表または VARRAY は、初期化するまでは基本構造的に 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）を使用します。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、基本構造的に NULL である場合があるため、NULL かどうかをテストできます。ただし、等価、不等価の比較はできません。この制限は、暗黙的な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 リストには使用できません。

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、それらを使用して、データの列をデータベースの表に出し入れしたり、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、次の構文を使用してコレクション内の要素を参照します。

```
collection_name(parameter_list)(subscript)
```

Oracle Call Interface（OCI）または Oracle プリコンパイラを使用すると、サブプログラムの仮パラメータとして宣言された索引付き表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 を使用します。

```
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 型を使用して、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 = 7468;
```

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 個以内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すると、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

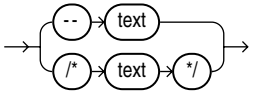
コレクション・メソッド、オブジェクト型、レコード

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用します。PL/SQL では、単一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。単一行コメントは、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



使用上の注意

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、単一行コメントは使用できません。これは、行の終わりを示す文字が無視され、単一行コメントが行の終わりで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用してください。

プログラムのテストやデバッグのときに、コード中の 1 行を使用禁止にする場合があります。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用すると、コードの一部分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。

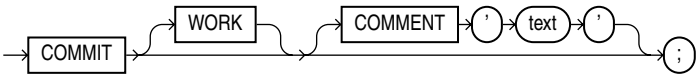
```
-- compute the area of a circle
area := pi * radius**2;  -- pi equals 3.14159
/* Compute the area
   of a circle. */
area := pi * radius**2;  /* pi equals 3.14159 */
```

COMMIT 文

COMMIT 文は、カレント・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、6-42 ページの「PL/SQL におけるトランザクション処理の概要」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

COMMENT

コメントを、カレント・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用されます。このテキストは引用符で囲んだ 50 文字以内のリテラルにしてください。

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目的にのみ使用します。

使用上の注意

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ますが、他のユーザーから変更内容は見えません。
- 考えを変えた場合や間違いを修正する場合は、ROLLBACK 文を使用して変更内容をロールバック（取消し）できます。

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のため、クローズしてください。詳細は、6-48 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インダウトになった場合、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、『Oracle9*i* データベース概要』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';  -- not allowed
```

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、次の文は許可されません。

```
COMMIT WORK RELEASE;  -- not allowed
```

関連項目

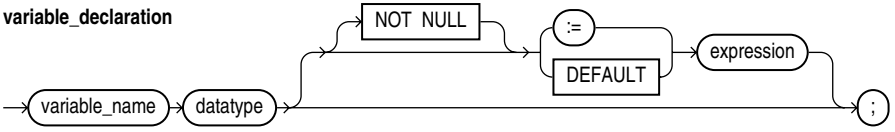
ROLLBACK 文、SAVEPOINT 文

定数と変数

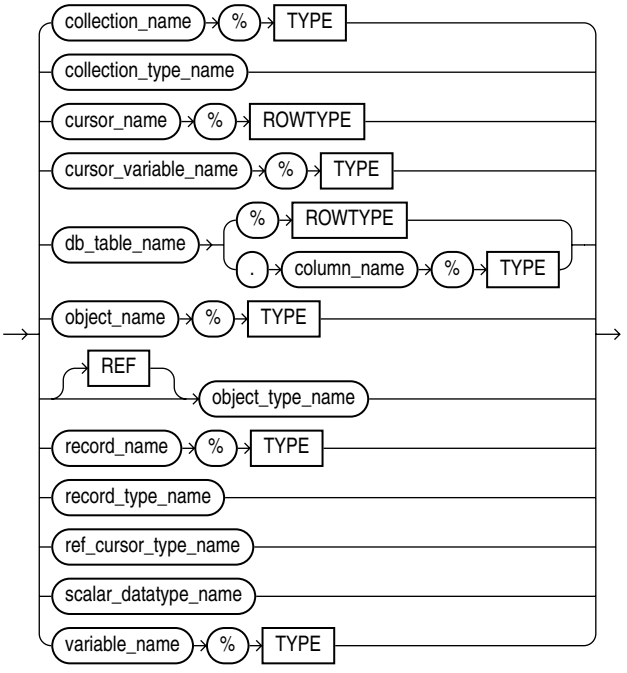
定数と変数は、任意の PL/SQL ブロック、サブプログラムまたはパッケージの宣言部で宣言できます。宣言によって、値の記憶領域を割り当て、データ型を指定し、値を参照できるように格納場所の名前を決めます。また、初期値を代入したり、NOT NULL 制約を付けることもできます。詳細は、2-11 ページの「宣言」を参照してください。

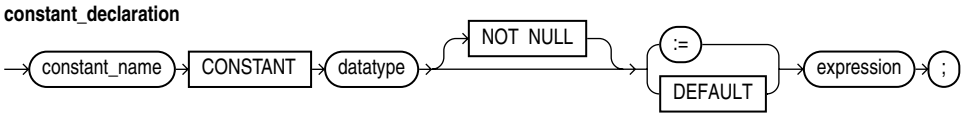
構文

variable_declaration



datatype





キーワードとパラメータの説明

collection_name
現行の有効範囲の中で、事前に宣言されているコレクション（索引付き表、ネストした表または VARRAY）を識別します。

collection_type_name
データ型指定子 TABLE または VARRAY を使用して定義されたユーザー定義のコレクション型を識別します。

CONSTANT
定数の宣言であることを示します。定数は宣言部で初期化してください。初期化された定数の値は変更できません。

constant_name
プログラム定数を識別します。命名規則は、2-4 ページの「識別子」を参照してください。

cursor_name
現行の有効範囲の中で、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
現行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。

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る必要があるデータベースの表（またはビュー）を識別します。

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る必要があるデータベースの表および列を識別します。

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。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のデータ型には互換性が必要です。

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く必要があります。

object_name
現行の有効範囲のうち、これより前の部分で宣言されているオブジェクト型のインスタンスを識別します。

record_name
現行の有効範囲のうち、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
現行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。

record_type_name
データ型指定子 RECORD を使用して定義するユーザー定義のレコード型を識別します。

ref_cursor_type_name
データ型指定子 REF CURSOR を使用して定義されたユーザー定義のカーソル変数型を識別します。

%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。

scalar_datatype_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。サイズ、精度、または文字とバイトのセマンティクスの修飾子が含まれます。

%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。

variable_name

プログラム変数を識別します。

使用上の注意

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かぎり、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にかかわらず、セッションごとに 1 回のみ初期化されます。

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、必ず初期化の句が必要です。%ROWTYPE を使用し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;
postal_code  VARCHAR2(20);
last_name    VARCHAR2(20 CHAR);
my_ename     emp.ename%TYPE;
```

関連項目

2-11 ページの「宣言」、3-2 ページの「事前定義されたデータ型」、代入文、式、%ROWTYPE 属性、%TYPE 属性

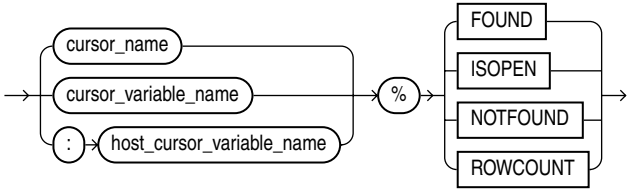
カーソル属性

どの明示カーソルおよびカーソル変数にも %FOUND、%ISOPEN、%NOTFOUND および %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、DML 文の実行について役立つ情報が戻されます。詳細は、6-33 ページの「カーソル属性の使用」を参照してください。

暗黙カーソル SQL にはさらに属性 %BULK_ROWCOUNT および %BULK_EXCEPTIONS があります。詳細は、13-171 ページの「SQL カーソル」を参照してください。

構文

cursor_attribute



キーワードとパラメータの説明

cursor_name

現行の有効範囲の中で、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

現行の有効範囲の中で、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。

%FOUND 属性

カーソル属性で、カーソルまたはカーソル変数の名前に追加できます。カーソルがオープンされてから最初のフェッチまでの cursor_name%FOUND の結果は、NULL になります。その後、直前のフェッチが行を戻した場合は TRUE に、直前のフェッチが行を戻さなかった場合は FALSE になります。

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付けてください。

%ISOPEN 属性

カーソル属性で、カーソルまたはカーソル変数の名前に追加できます。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。

%NOTFOUND 属性

カーソル属性で、カーソルまたはカーソル変数の名前に追加できます。カーソルがオープンされてから最初のフェッチまでの cursor_name%NOTFOUND の結果は、NULL になります。その後、直前のフェッチが行を戻した場合は FALSE に、直前のフェッチが行を戻さなかった場合は TRUE になります。

%ROWCOUNT 属性

カーソル属性で、カーソルまたはカーソル変数の名前に追加できます。カーソルがオープンされると %ROWCOUNT は 0（ゼロ）になります。最初のフェッチまでは、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。フェッチで行が戻されるたびに、数値は増加します。

使用上の注意

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 を使用して、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 を使用して、これまでに取り出した行の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 TRUE である場合にのみ EXIT WHEN 文が実行されるためです。安全のために、次の EXIT 文をかわりに使用できます。

```
EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
```

カーソルの属性は、プロシージャ文では使用できますが、SQL 文では使用できません。

例

次の PL/SQL ブロックでは、%FOUND を使用し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用しています。ただし、IF 文の中で %FOUND を使用するかわりに、EXIT WHEN 文の中で %NOTFOUND を使用しています。

```
-- 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 を使用し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 を使用して、給与が最も高い 5 人の従業員の名前と給与をフェッチしています。

```
-- 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 を使用して、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

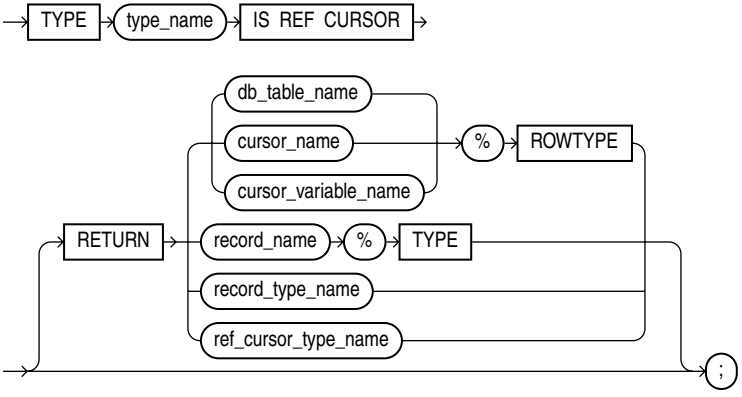
複数行の問合せを実行するために、Oracle は処理情報を格納する名前の付けられていない作業域をオープンします。その情報にアクセスするには、作業域の名前を示す明示カーソルを使用します。または、作業域を指すカーソル変数を使用します。カーソルが常に同じ問合せ作業域を参照するのに対し、カーソル変数は異なる作業域を参照できます。カーソル変数を作成するには、REF CURSOR 型を定義してから、その型のカーソル変数を宣言します。

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成されます。

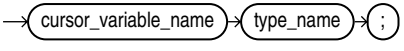
詳細は、6-16 ページの「カーソル変数の使用」を参照してください。

構文

ref_cursor_type_definition



cursor_variable_declaration



キーワードとパラメータの説明

cursor_name
現行の有効範囲の中で、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
現行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。

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る必要があるデータベースの表（またはビュー）を識別します。

record_name
現行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
データ型指定子 RECORD を使用して定義したユーザー定義のレコード型を識別します。

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、カーソル変数の戻り値のデータ型を定義します。RETURN 句で %ROWTYPE 属性を使用すると、データベース表の行や、カーソルまたは強い型指定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 属性を使用して、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。

%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、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指定します。

type_name

データ型指定子 REF CURSOR を使用して定義されたユーザー定義のカーソル変数型です。

使用上の注意

カーソル変数は、すべての PL/SQL クライアントで使用します。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用できます。

Oracle データベース・サーバーも 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）を通じ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用します。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おらず、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のみです。たとえば、OCI クライアント、Oracle Forms アプリケーションおよび Oracle サーバー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（限定的）ものと弱い（限定的ではない）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、強い型指定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ためです。弱い REF CURSOR 型は、より柔軟です。弱い型指定のカーソル変数は、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 を使用すると、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 を使用して、強い型指定（弱い型指定ではなく）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。

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。6-32 ページの「カーソル変数の制限」を参照してください。

カーソル変数を制御する場合は、OPEN-FOR、FETCH および CLOSE という 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行を取り出します。すべての行が処理された後に、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ている必要があります。また、フィールドまたは変数の数は、列の値の数と一致する必要があります。それ以外の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指定である場合は、両方が同じデータ型であることが必要です。ただし、一方または両方のカーソル変数が弱い型指定である場合は、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 モードを指定する必要があります。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 モードを指定する必要があります。

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。

問合せ作業域を指していないカーソル変数に対してフェッチ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します。
- 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します。

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。

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・カーソル変数を使用する場合は、それをバインド変数として PL/SQL に渡す必要があります。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、弱い型指定の 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の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・プロシージャを使用してカーソル変数をオープンする方法もあります。単に別個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、次のような本体なしのパッケージを作成する場合は、スタンドアロン・プロシージャを作成し、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

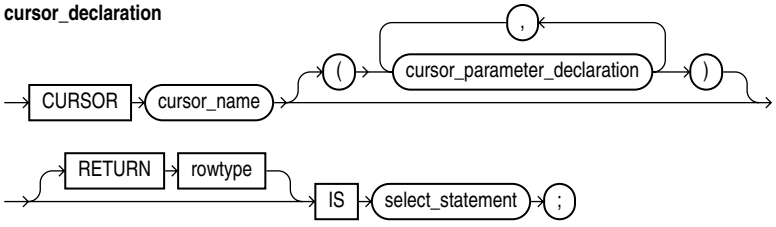
CLOSE 文、カーソル属性、カーソル、FETCH 文、OPEN-FOR 文

カーソル

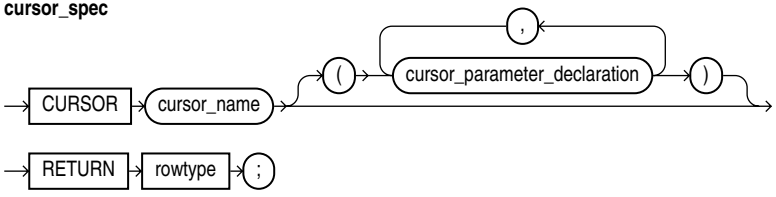
複数行の問合せを実行するために、Oracle は処理情報を格納する名前の付けられていない作業域をオープンします。明示カーソルを使用すると、作業域の名前付け、情報へのアクセス、行の個別処理が可能です。詳細は、6-6 ページの「カーソル管理」を参照してください。

構文

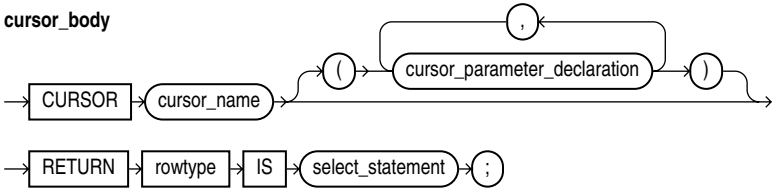
cursor_declaration



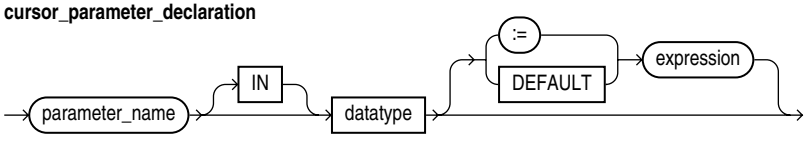
cursor_spec



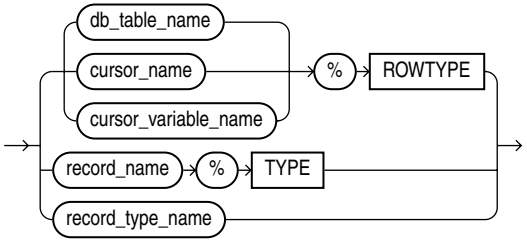
cursor_body



cursor_parameter_declaration



rowtype



キーワードとパラメータの説明

cursor_name
現行の有効範囲の中で、事前に宣言されている明示カーソルを識別します。

datatype
これは、型指定子です。`datatype` の構文は、13-36 ページの「定数と変数」を参照してください。

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る必要があるデータベースの表（またはビュー）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。宣言が PL/SQL コンパイラによって処理されると、`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のデータ型には互換性が必要です。

parameter_name
カーソルのパラメータ、つまり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用できる場所であれば、どこででも使用できます。カーソルの仮パラメータは `IN` パラメータにしてください。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。

record_name
現行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

データ型指定子 RECORD を使用して定義したユーザー定義のレコード型を識別します。

RETURN

RETURN 句の開始を知らせるキーワードです。この句では、カーソルの戻り値のデータ型を定義します。RETURN 句で %ROWTYPE 属性を使用すると、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 属性を使用して、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 句が必要です。さらに、SELECT 句の中の選択項目の数、順序およびデータ型は、RETURN 句と一致している必要があります。

%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またはカーソル変数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。

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 句は使用できません。13-163 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用する必要があります。

%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指定します。

使用上の注意

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は使用できません。

カーソル名に値を代入したり、カーソル名を式の中で使用することはできません。ただし、カーソルの有効範囲規則は変数の有効範囲規則と同じです。詳細は、2-19 ページの「PL/SQL の識別子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ため、cursor_declaration の SELECT 文で INTO 句を使用する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用さ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用できます。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、無制約で指定する必要があります。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))  -- not allowed
```

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

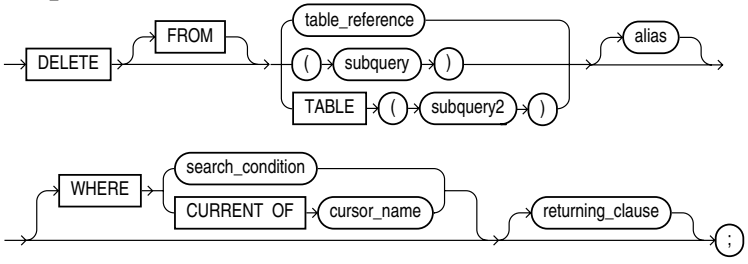
CLOSE 文、FETCH 文、OPEN 文、SELECT INTO 文

DELETE 文

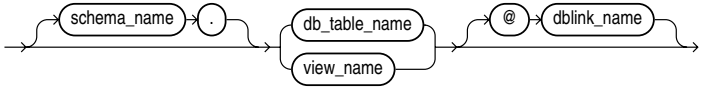
DELETE 文は、指定された表またはビューから、行のデータを削除します。DELETE 文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

構文

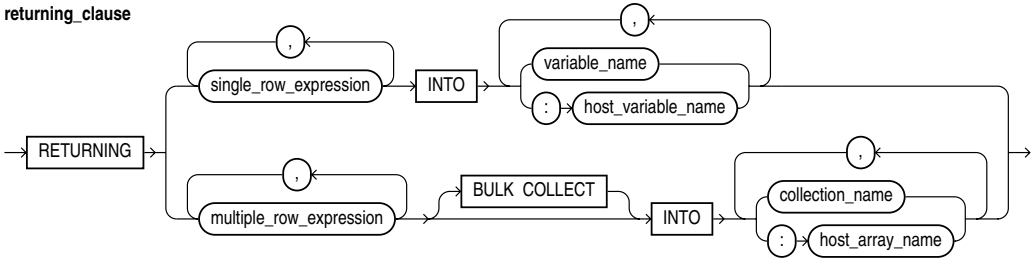
delete_statement



table_reference



returning_clause



キーワードとパラメータの説明

alias

参照される表またはビューの別名（通常は短縮名）で、WHERE 句の中で頻繁に使用されます。

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に、SQL エンジンに指示します。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（コンパウンドではない）が格納されている必要があります。詳細は、5-38 ページの「バルク・バインドを使用したコレクションのループ・オーバーヘッドの削減」を参照してください。

returning_clause

この句を使用すると、削除された行から値を戻せるため、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、変数かホスト配列（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。

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 句は使用できません。13-163 ページの「SELECT INTO 文」を参照してください。

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る必要があり、ユーザーは DELETE 権限を持つ必要があります。

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。

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。

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のみが削除されます。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。

使用上の注意

DELETE WHERE CURRENT OF 文は、オープンされているカーソルからのフェッチ（カーソル FOR ループで実行される暗黙的なフェッチを含む）の後で使用できます（ただし、そのためには、対応付けられた問合せが FOR UPDATE であることが必要です）。この文は現在行、つまり直前にフェッチされた行を削除します。

暗黙カーソル SQL と、カーソル属性 %NOTFOUND、%FOUND および %ROWCOUNT を使用すると、DELETE 文の実行に関する有用な情報にアクセスできます。

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、削除された行からローカル変数に 2 つの列値を戻します。

```
DECLARE
   my_empno emp.empno%TYPE;
   my_ename emp.ename%TYPE;
   my_job   emp.job%TYPE;
BEGIN
   ...
   DELETE FROM emp WHERE empno = my_empno
      RETURNING ename, job INTO my_ename, my_job;
END;
```

BULK COLLECT 句を FORALL 文と組み合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 deptno = 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

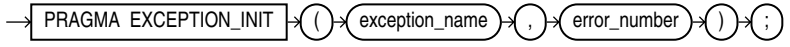
FETCH 文、SELECT INTO 文

EXCEPTION_INIT プラグマ

EXCEPTION_INIT プラグマは、例外名を Oracle エラー番号に対応付けます。この対応付けにより、OTHERS ハンドラを使用するかわりに、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、7-8 ページの「PL/SQL 例外と番号の対応付け : EXCEPTION_INIT プラグマ」を参照してください。

構文

exception_init_pragma



キーワードとパラメータの説明

error_number

任意の有効な ORACLE 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。

exception_name

現行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。

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のみです。

使用上の注意

EXCEPTION_INIT 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使用できます。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する必要があります。

1 つのエラー番号に割り当てる例外名は 1 つのみです。

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

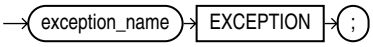
AUTONOMOUS_TRANSACTION プラグマ、例外、SQLCODE ファンクション

例外

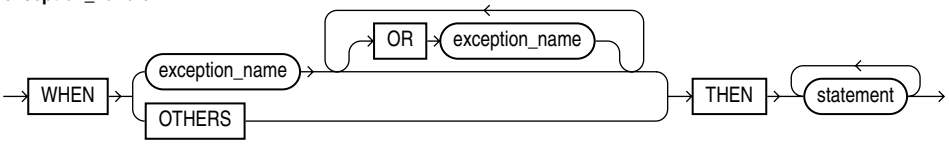
例外は、事前定義またはユーザー定義可能な、ランタイム・エラーまたは警告状態です。事前定義の例外は実行時システムによって暗黙的（自動的）に呼び出されます。ユーザー定義の例外は RAISE 文によって明示的に呼び出す必要があります。呼び出された例外を処理するには、例外ハンドラと呼ばれる独立したルーチンを作成します。詳細は、第 7 章を参照してください。

構文

exception_declaration



exception_handler



キーワードとパラメータの説明

exception_name

ZERO_DIVIDE のような事前定義の例外、または現行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のみ使用できます。キーワード WHEN に続く例外のリストの中では、OTHERS を使用できません。

statement

これは、実行可能文です。statement の構文は、13-10 ページの「ブロック」を参照してください。

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、複数の例外で一連の同一文を実行できます。リスト中のいずれかの例外が呼び出されると、それに関連付けられた文が実行されます。

使用上の注意

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のみ使用できます。例外の有効範囲の規則は変数と同じです。ただし、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、7-4 ページの「事前定義の PL/SQL 例外」を参照してください。PL/SQL は、事前定義済みの例外をパッケージ STANDARD でグローバルに宣言しているため、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用する必要があります。

```
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置く必要があります。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。例外ハンドラから参照できる変数は、カレント・ブロックから参照できる変数のみです。

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のみにしてください。呼び出された例外に対応する例外ハンドラがカレント・ブロックに存在しない場合、例外は次の規則に従って伝播します。

- カレント・ブロックの外側にブロックがある場合、例外はそのブロックに渡されます。その後、その外のブロックがカレント・ブロックになります。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ます。
- カレント・ブロックの外側に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ます。

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 つのみです。このため、ハンドラの内側で例外が発生すると、カレント・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。それ以降の例外の伝播は通常どおりに起こります。

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

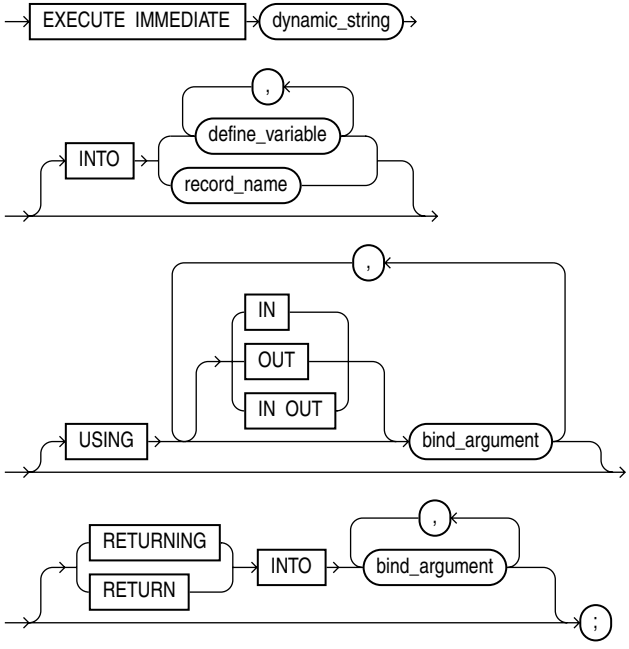
ブロック、EXCEPTION_INIT プラグマ、RAISE 文

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 11 章を参照してください。

構文

execute_immediate_statement



キーワードとパラメータの説明

bind_argument
動的 SQL 文または PL/SQL ブロックに渡される値を持つ式か、動的 SQL 文または PL/SQL ブロックから戻された値を格納する変数です。

define_variable_name
選択された列の値を格納する変数を識別します。

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。

INTO ...
単一行の問合せの場合に使用され、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。問合せによって取り出された値それぞれに対して、INTO 句の中に、対応する型互換性変数が存在している必要があります。

record_name
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。

RETURNING INTO ...
RETURNING 句のある（BULK COLLECT 句のない）DML 文の場合に使用され、列の値が戻されるバインド変数を指定します。DML 文によって戻された値それぞれに対して、RETURNING INTO 句の中に、対応する型互換性変数が存在している必要があります。

USING ...
入力または出力バインド引数（あるいはその両方）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の IN になります。

使用上の注意

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。

バインド引数は、すべて USING 句に入れることができます。デフォルトのパラメータ・モードは IN です。RETURNING 句を持つ DML 文の場合は、パラメータ・モード OUT を定義して指定しなくても、OUT 引数を RETURNING INTO 句に入れることができます。USING 句と RETURNING INTO 句の両方を使用する場合、USING 句には IN 引数のみを含めることができます。

実行時に、バインド引数は動的文字列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 句内または RETURNING INTO 句内（あるいはその両方）のバインド引数に対応付ける必要があります。数値リテラル、文字リテラルおよび文字列リテラルはバインド引数として使用できますが、ブール・リテラル（TRUE、FALSE および NULL）は使用できません。動的文字列に NULL を渡すには、回避策を使用する必要があります。11-16 ページの「NULL を渡す方法」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。

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字列を準備するため、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 SQL の例がいくつか含まれています。

```
DECLARE
   sql_stmt    VARCHAR2(200);
   plsql_block VARCHAR2(500);
   emp_id      NUMBER(4) := 7566;
   salary      NUMBER(7,2);
   dept_id     NUMBER(2) := 50;
   dept_name   VARCHAR2(14) := 'PERSONNEL';
   location    VARCHAR2(13) := 'DALLAS';
   emp_rec     emp%ROWTYPE;
BEGIN
   EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt NUMBER)';

   sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
   EXECUTE IMMEDIATE sql_stmt USING dept_id, dept_name, location;

   sql_stmt := 'SELECT * FROM emp WHERE empno = :id';
   EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING emp_id;

   plsql_block := 'BEGIN emp_pkg.raise_salary(:id, :amt); END;';
   EXECUTE IMMEDIATE plsql_block USING 7788, 500;

   sql_stmt := 'UPDATE emp SET sal = 2000 WHERE empno = :1
      RETURNING sal INTO :2';
   EXECUTE IMMEDIATE sql_stmt USING emp_id RETURNING INTO salary;

   EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = :num'
      USING dept_id;

   EXECUTE IMMEDIATE 'ALTER SESSION SET SQL_TRACE TRUE';
END;
```

関連項目

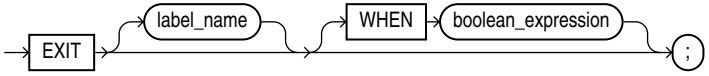
OPEN-FOR-USING 文

EXIT 文

EXIT 文は、ループを終了するために使用します。EXIT 文には、無条件 EXIT と、条件付き 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、4-9 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

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 文を使用し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、カレント・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、13-69 ページの「式」を参照してください。

EXIT

無条件の（つまり WHEN 句のない）EXIT 文は、カレント・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。カレント・ループのみでなく、ラベルが付けられている外側のループも終了できます。

使用上の注意

EXIT 文は、ループの内側でのみ使用できます。PL/SQL では無限ループをコーディングできます。たとえば、次のループは、通常の方法では永久に終了しません。

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

このループを終了させるには、EXIT 文を使用します。

EXIT 文を使用してカーソル 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

次の例にある EXIT 文は、ブロックから直接終了できないため誤りです。この文が終了できるのはループからのみです。

```
DECLARE
   amount  NUMBER;
   maximum NUMBER;
BEGIN
   ...
   BEGIN
      ...
      IF amount >= maximum THEN
         EXIT;  -- not allowed; use RETURN instead
      END IF;
   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 LOOP
   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

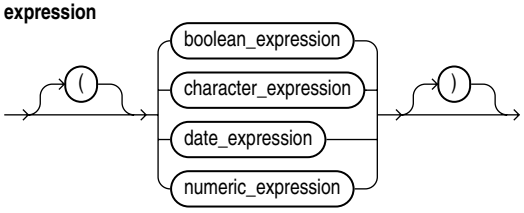
式

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。

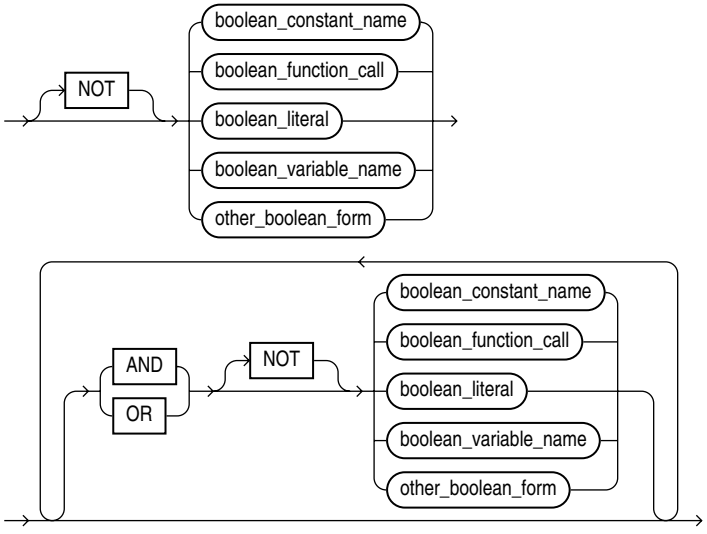
PL/SQL コンパイラは、式を構成する変数、定数、リテラルおよび演算子の型から、式のデータ型を決定します。式が評価されたときは、その型の 1 つの値が結果として得られます。詳細は、2-23 ページの「PL/SQL の式および比較」を参照してください。

構文

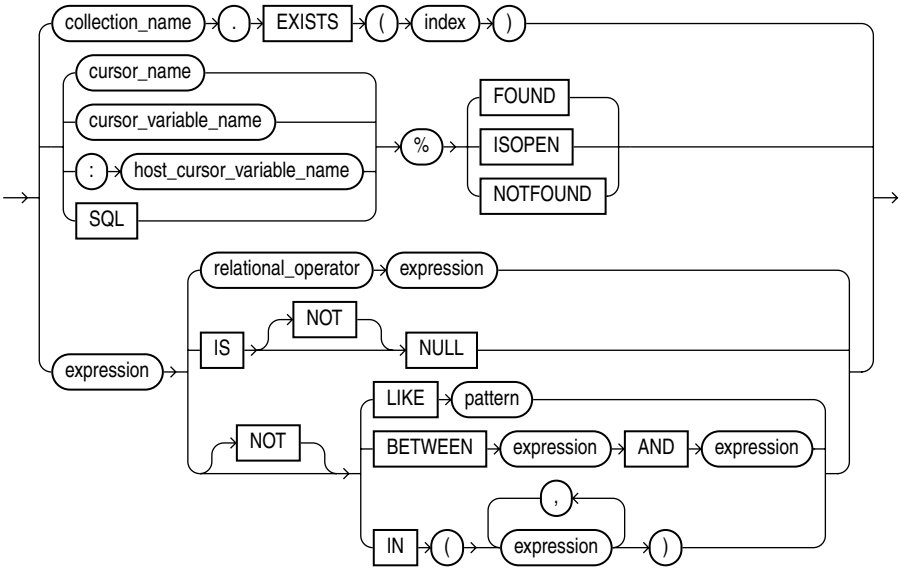
expression



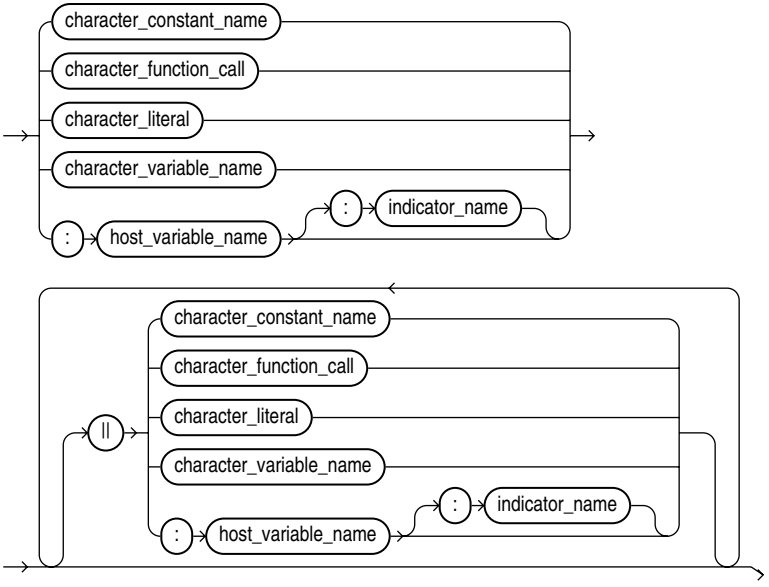
boolean_expression



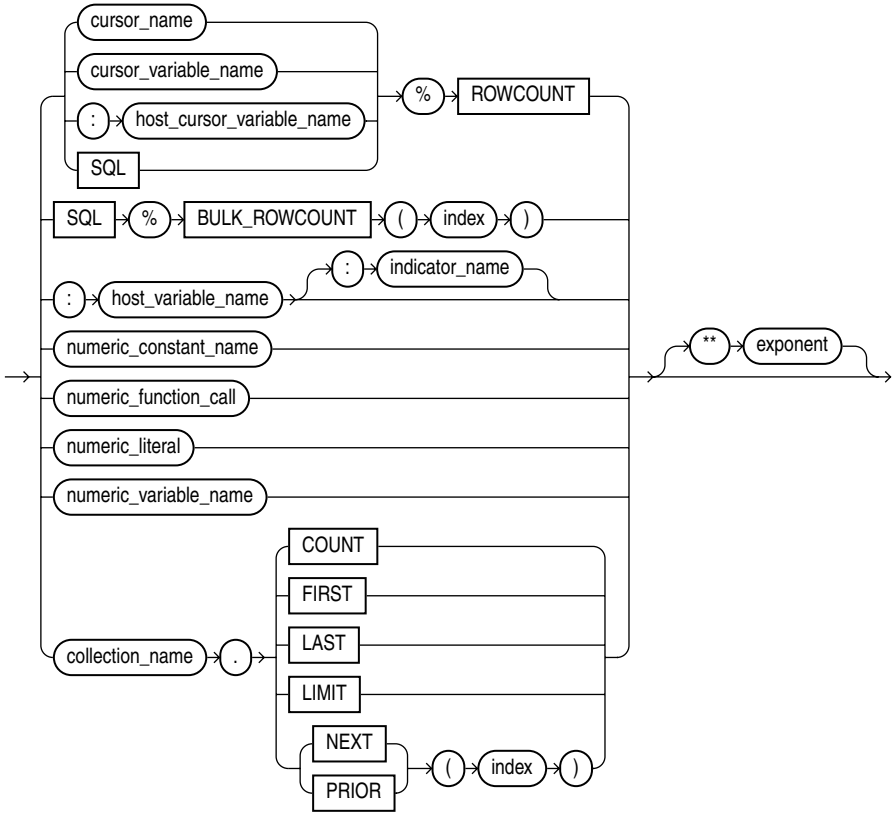
other_boolean_form



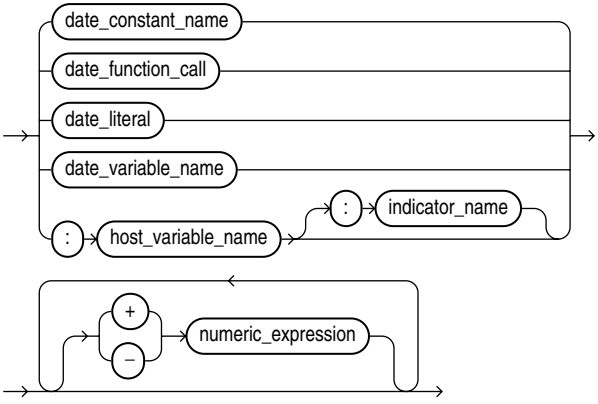
character_expression



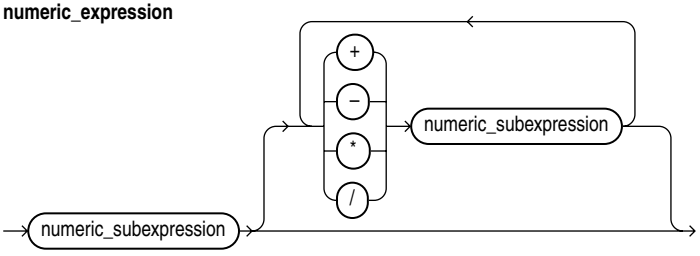
numeric_subexpression



date_expression



numeric_expression



キーワードとパラメータの説明

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テストします。つまり、下限以上、上限以下にあるかどうかがテストされます。

boolean_constant_name
BOOLEAN 型の定数を指定します。このような定数は、TRUE、FALSE、または NULL に初期化される必要があります。ブール定数に対する算術演算は許可されていません。

boolean_expression
結果が TRUE、FALSE、NULL のいずれかのブール値になる式です。

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、FALSE および NULL のみです。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。BOOLEAN 変数に対する算術演算も許可されていません。

%BULK_ROWCOUNT
FORALL 文で使用するように設計された、暗黙カーソル SQL の複合属性です。詳細は、13-171 ページの「SQL カーソル」を参照してください。

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る必要があります。

character_expression
結果が、文字または文字列になる式です。

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。

collection_name
現行の有効範囲のうち、これより前の部分で宣言されているコレクション（ネストした表、索引付き表または VARRAY）を指定します。

cursor_name
現行の有効範囲の中で、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
現行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。

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る必要があります。

date_expression
結果が、日付 / 時刻値になる式です。

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。

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。それ以外の場合、`EXISTS(n)` は `FALSE` を戻します。詳細は、13-22 ページの「コレクション・メソッド」を参照してください。

exponent
結果が数値になる式です。

%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カーソル SQL にも追加できます。

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付ける必要があります。

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る必要があります。また、ホスト変数には、接頭辞としてコロンを付ける必要があります。

IN
この比較演算子は、セット・メンバーシップをテストします。つまり、集合の「いずれかのメンバーと等しい」かどうかがテストされます。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、次の形式の式の場合、

```
value NOT IN set
```

集合に NULL が含まれていると FALSE になります。

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。

indicator_name
PL/SQL ホスト環境で宣言され、PL/SQL に渡されるインジケータ変数を識別します。インジケータ変数には、接頭辞としてコロンを付ける必要があります。インジケータ変数は、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、インジケータ変数を使用し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。

IS 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 を戻し、オペランドが NULL でなければ FALSE を戻します。

LIKE
この比較演算子は、文字値とパターンを比較します。大 / 小文字が区別されます。LIKE は、文字のパターンが一致した場合はブール値 TRUE、一致しない場合は FALSE を戻します。

NOT、AND、OR
2-25 ページの表 2-2 の 3 値論理に従う論理演算子です。AND は、オペランドの両方が TRUE の場合にのみ TRUE を戻します。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。詳細は、2-25 ページの「論理演算子」を参照してください。

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。

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る必要があります。

numeric_expression
結果が、整数または実数になる式です。

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。

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。

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 種類使用できます。アンダースコア（_）は 1 つの文字を表し、パーセント記号（%）はゼロ個以上の文字を表します。

relational_operator
この演算子を使用すると、式を比較できます。各演算子の意味は、2-26 ページの「比較演算子」を参照してください。

SQL

SQL の DML 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。暗黙カーソル SQL は常に、直前に実行された SQL 文を参照します。

+、-、/、*、**

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。

||

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。

使用上の注意

ブール式では、互換性のあるデータ型を持つ値のみを比較できます。詳細は、3-22 ページの「データ型変換」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。ただし、式が FALSE または NULL になると、関連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用すると、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、カッコを使用すると、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用します。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。

```
(a + b) > c             -- Boolean expression
NOT finished            -- Boolean expression
TO_CHAR(acct_no)        -- character expression
'Fat ' || 'cats'        -- character expression
'15-NOV-95'             -- date expression
MONTHS_BETWEEN(d1, d2)  -- date expression
pi * r**2               -- numeric expression
emp_cv%ROWCOUNT         -- numeric expression
```

関連項目

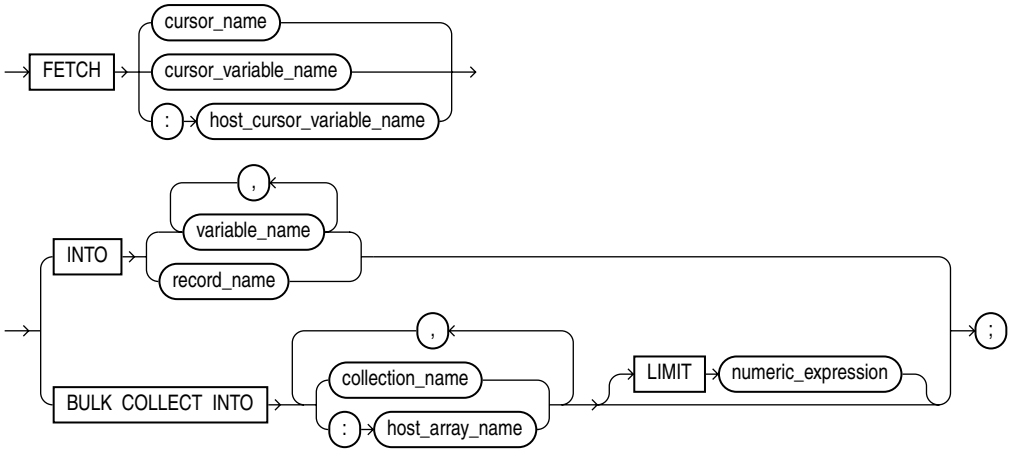
代入文、定数と変数、EXIT 文、IF 文、LOOP 文

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、6-6 ページの「カーソル管理」を参照してください。

構文

fetch_statement



キーワードとパラメータの説明

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に、SQL エンジンに指示します。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。

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ている必要があります。

cursor_name

現行の有効範囲の中で、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

現行の有効範囲の中で、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_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ている必要があります。ホスト配列には、接頭辞としてコロンが必要です。

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付けてください。

LIMIT

バルク（スカラーではない）FETCH 文の中でのみ許されるオプションの句で、データベースから取り出される行の数を制限します。

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ている必要があります。

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ている必要があります。

使用上の注意

複数行の問合せを処理するには、カーソル FOR ループか FETCH 文を使用します。

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のみ評価されます。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する必要があります。

カーソルを再オープンするには、まずクローズしてください。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、ターゲット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ている必要があります。また、フィールドまたは変数の数は、列の値の数と一致する必要があります。

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 モードを指定する必要があります。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 モードを指定する必要があります。

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 を使用する必要があります。

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のみ評価されます。

```
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 へ一度に 1 行ずつ行をフェッチします。

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   emp_cv  EmpCurTyp;
   emp_rec emp%ROWTYPE;
BEGIN
   LOOP
      FETCH emp_cv INTO emp_rec;
      EXIT WHEN emp_cv%NOTFOUND;
      ...
   END LOOP;
END;
```

BULK COLLECT 句では、Oracle データの列全体をバルク・バインドできます。そのようにすると、結果セットからすべての行を一度に取り出せます。次の例では、1 つのカーソルから 1 つのコレクションにバルク・フェッチを行います。

```
DECLARE
   TYPE NameList IS TABLE OF emp.ename%TYPE;
   names NameList;
   CURSOR c1 IS SELECT ename FROM emp WHERE job = 'CLERK';
BEGIN
   OPEN c1;
   FETCH c1 BULK COLLECT INTO names;
   ...
   CLOSE c1;
END;
```

次の例では、LIMIT 句を使用します。ループが繰り返されるたびに、FETCH 文によって 100（またはそれ以下の）行が索引付き表 acct_ids に取り出されます。前の値は上書きされます。

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
   CURSOR c1 IS SELECT acct_id FROM accounts;
   acct_ids NumList;
   rows NATURAL := 100;  -- set limit
BEGIN
   OPEN c1;
   LOOP
      /* The following statement fetches 100 rows (or less). */
      FETCH c1 BULK COLLECT INTO acct_ids LIMIT rows;
      EXIT WHEN c1%NOTFOUND;
      ...
   END LOOP;
   CLOSE c1;
END;
```

関連項目

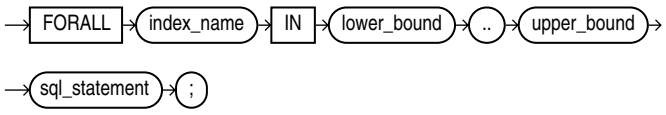
CLOSE 文、カーソル、カーソル変数、LOOP 文、OPEN 文、OPEN-FOR 文

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に、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、5-38 ページの「バルク・バインドを使用したコレクションのループ・オーバーヘッドの削減」を参照してください。

構文

forall_statement



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 文の中でのみ参照できる、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オーバーライドします。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できません。

lower_bound .. upper_bound

結果が整数値になる式です。必要に応じ、PL/SQL が最も近い整数に四捨五入します。整数には有効範囲内の連続した索引番号を指定する必要があります。SQL エンジンは、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。式は、最初に FORALL 文に入ったときにのみ評価されます。

SAVE EXCEPTIONS

これらのオプションのキーワードを使用すると、FORALL ループは一部の DML 操作が失敗しても継続します。エラーの詳細は、SQL%BULK_EXCEPTIONS でループの後に取得できます。プログラムは、発生するたびに例外を個別に処理するのではなく、FORALL ループの後ですべてのエラーをレポートまたはクリーン・アップできます。

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 文にしてください。

使用上の注意

SQL 文は複数のコレクションを参照できます。ただし、パフォーマンスに関わる利点は、添字付きコレクションのみに適用されます。

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的なセーブポイントまでロールバックされます。前回の FORALL ループの反復中に行われた変更はロールバックされません。

制限

FORALL 文には、次の制限が適用されます。

- キーが文字列型の結合配列の要素は、ループできません。
- FORALL ループ内では、UPDATE 文の SET 句と WHERE 句の両方で、同じコレクションを参照することはできません。この場合は、そのコレクションの 2 つ目のコピーを作成し、WHERE 句では新しい名前を参照する必要があります。
- FORALL 文を使用できるのは、サーバー側（クライアント側ではなく）のプログラム内のみです。クライアント側で使用すると、「この機能はクライアント側のプログラムではサポートされていません。」というエラーが表示されます。
- INSERT 文、UPDATE 文または DELETE 文では少なくとも 1 つのコレクションを参照する必要があります。たとえば、ループで一連の定数値を挿入する FORALL 文では、例外が発生します。
- 指定した範囲のコレクション要素がすべて存在している必要があります。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);  -- causes an error
END;
```

- 次に示すように、複合値の入力コレクションは分解してデータベース列にバインドすることはできません。

```
CREATE TABLE coords (x NUMBER, y NUMBER);
CREATE TYPE Pair AS OBJECT (m NUMBER, n NUMBER);

DECLARE
   TYPE PairTab IS TABLE OF Pair;
   pairs PairTab := PairTab(Pair(1,2), Pair(3,4), Pair(5,6));
   TYPE NumTab IS TABLE OF NUMBER;
   nums NumTab := NumTab(1, 2, 3);
```

```
BEGIN
   /* The following statement fails. */
   FORALL i IN 1..3
      UPDATE coords SET (x, y) = pairs(i)
         WHERE x = nums(i);
END;
```

回避策としては、複合値を手動で分解する方法があります。

```
DECLARE
   TYPE PairTab IS TABLE OF Pair;
   pairs PairTab := PairTab(Pair(1,2), Pair(3,4), Pair(5,6));
   TYPE NumTab IS TABLE OF NUMBER;
   nums NumTab := NumTab(1, 2, 3);
BEGIN
   /* The following statement succeeds. */
   FORALL i in 1..3
      UPDATE coords SET (x, y) = (pairs(i).m, pairs(i).n)
         WHERE x = nums(i);
END;
```

- 次の例に示すように、コレクションの添字は式にはなりません。

```
FORALL j IN mgrs.FIRST..mgrs.LAST
   DELETE FROM emp WHERE mgr = mgrs(j+1);  -- invalid subscript
```

- カーソル属性 %BULK_ROWCOUNT は他のコレクションには代入できません。パラメータとしてサブプログラムに渡すこともできません。

例

次の例に示すように、上限と下限を使用し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 エンジンは添字付きコレクションのみをバルク・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

5-46 ページの「BULK COLLECT 句を使用した、問合せ結果のコレクションへの取出し」

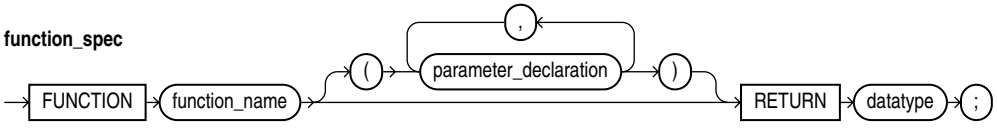
ファンクション

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用します。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、戻り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（または AS）で始め、キーワード END で終わります。END の後には、オプションのファンクション名を続けます。

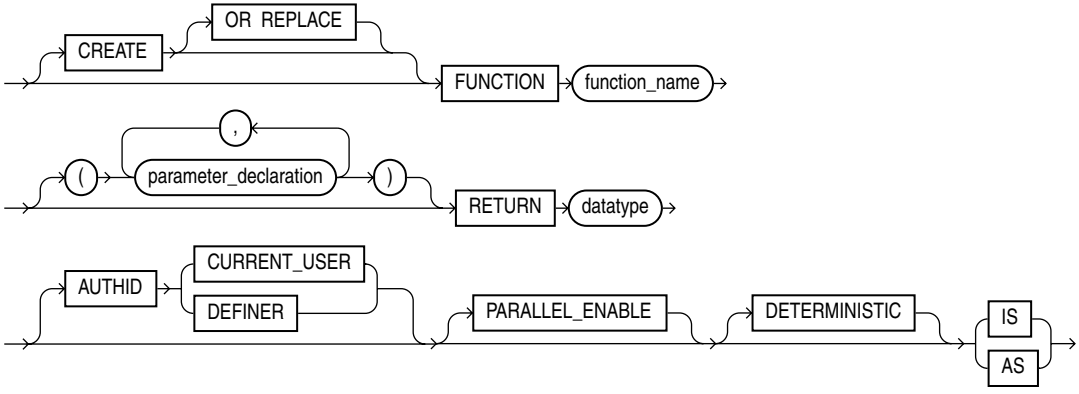
ファンクション本体には、宣言部（オプション）、実行部および例外処理部（オプション）の 3 つの部分があります。宣言部には、型、カーソル、定数、変数、例外およびサブプログラムが含まれています。これらの項目はローカルで、ファンクションを終了すると消去されます。実行部には、値の代入、実行の制御および Oracle データの操作を実行する文があります。例外処理部には、実行の途中で呼び出された例外を処理する例外ハンドラがあります。詳細は、8-6 ページの「PL/SQL ファンクション」を参照してください。

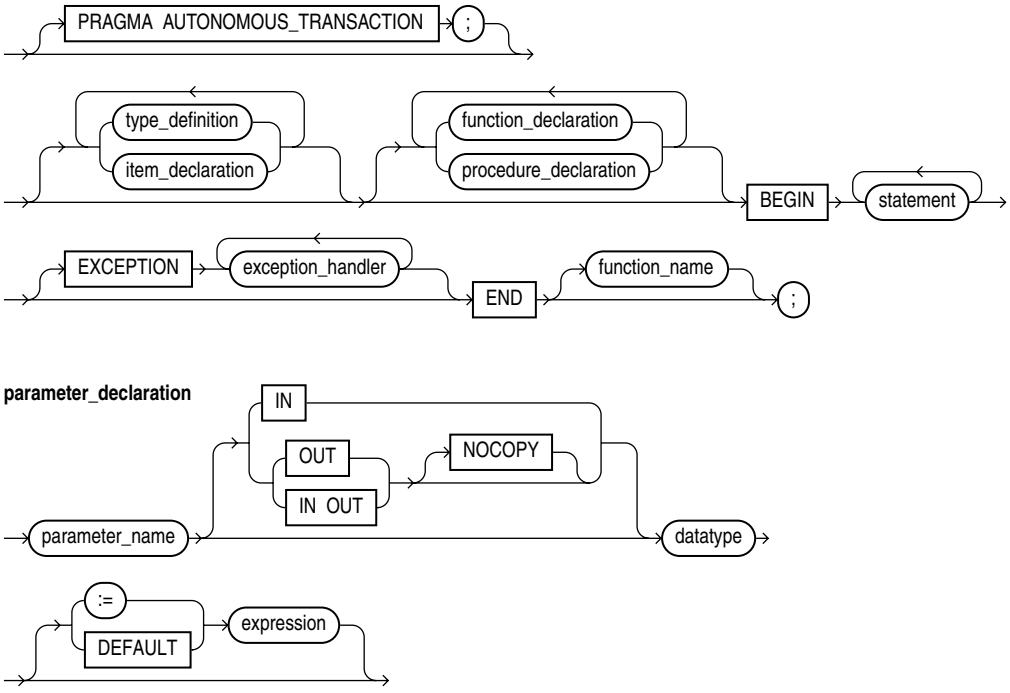
構文

function_spec



ffunction_declaration | function body





キーワードとパラメータの説明

AUTHID

ストアド・ファンクションをその所有者（デフォルト）と現行ユーザー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所有者と現行ユーザーのどちらのスキーマで解決されるかを決定します。CURRENT_USER を指定すると、デフォルトの動作を変更できます。詳細は、8-50 ページの「実行者権限と定義者権限」を参照してください。

CREATE

オプションの CREATE 句を使用するとスタンドアロン・ファンクションを作成できます。これは Oracle データベースに格納できます。CREATE 文は、SQL*Plus またはシステム固有の動的 SQL を使用したプログラムから対話形式で実行できます。

datatype

これは、型指定子です。datatype の構文は、13-36 ページの「定数と変数」を参照してください。

DETERMINISTIC

このヒントは、オプティマイザが冗長なファンクション・コールを回避するのに役立ちます。ストアド・ファンクションが同じ引数で事前にコールされた場合は、オプティマイザは前の結果を使用できます。ファンクションの結果をセッション変数の状態またはスキーマ・オブジェクトに依存させないでください。さもないと、コールごとに結果が異なる可能性があります。DETERMINISTIC ファンクションのみが、ファンクションベースの索引または問合せ再作成を使用可能にしたマテリアライズド・ビューからコールできます。詳細は、『Oracle9*i* SQL リファレンス』の「CREATE INDEX」および「CREATE MATERIALIZED VIEW」の各文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、13-60 ページの「例外」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に、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のデータ型には互換性が必要です。

function_name

ユーザー定義ファンクションを識別します。

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、コール先のサブプログラムに値を渡すために使用します。OUT パラメータは、サブプログラムのコール元に値を戻すために使用します。IN OUT パラメータを使用すると、コール先のサブプログラムに初期値を渡して、コール元に更新された値を戻すことができます。

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、13-10 ページの「ブロック」を参照してください。

NOCOPY

コンパイラ・ヒント（ディレクティブではなく）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、8-17 ページの「NOCOPY コンパイラ・ヒントを使用した大型データ構造の受渡し」を参照してください。

PARALLEL_ENABLE

ストアド・ファンクションがパラレル DML 評価のスレーブ・セッションで安全に使用されることを宣言します。メイン（ログオン）・セッションの状態が、スレーブ・セッションと共有されることはありません。スレーブ・セッションごとに固有の状態があり、セッション開始時に初期化されます。ファンクションの結果がセッション（static）変数の状態に依存しないようにしてください。さもないと、セッションごとに結果が異なる可能性があります。

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。

AUTONOMOUS_TRANSACTION プラグマ

このプラグマはファンクションを自律型（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。自律型トランザクションは、別のトランザクション、メイン・トランザクション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。詳細は、6-52 ページの「自律型トランザクションによる独立した作業単位の実行」を参照してください。

procedure_declaration

プロシージャを宣言します。procedure_declaration の構文は、13-140 ページの「プロシージャ」を参照してください。

RETURN

RETURN 句の開始を知らせるキーワードです。この句では、戻り値のデータ型を定義します。

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、13-10 ページの「ブロック」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用すると、IN パラメータをデフォルト値に初期化できます。

使用上の注意

次の例が示すように、ファンクションは式の一部としてコールされます。

```
promotable := sal_ok(new_sal, new_title) AND (rating > 3);
```

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かぎり、SQL 文からコールできます。8-9 ページの「PL/SQL サブプログラムの副作用の制御」を参照してください。

ファンクションには、RETURN 文へ導く少なくとも 1 つの実行パスが必要です。実行パスがないと、実行時に「ファンクションが値なしで返されました。(*function returned without value*)」というエラーが発生します。また、RETURN 文には、RETURN 文の実行時に評価される式が含まれている必要があります。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。ファンクション識別子は変数のように取り扱われます。

ファンクションの仕様部と本体を合せて 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、実装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。ただし、このようなファンクションは、パッケージの中からのみコールできます。

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できます。パラメータ・モードの概要は、8-16 ページの表 8-1 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 モードを使用しないでください。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はない変数の値を変更するという副作用も避ける必要があります。

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

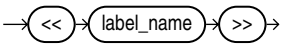
コレクション・メソッド、パッケージ、プロシージャ

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、4-17 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 文を使用すると、<<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。

使用上の注意

GOTO 文の宛先として使用できないものがあります。特に、GOTO 文は IF 文、LOOP 文またはサブブロックには分岐できません。たとえば、次の GOTO 文は許可されません。

```
BEGIN
   ...
   GOTO update_row;  -- can't branch into IF statement
   ...
   IF valid THEN
      ...
      <<update_row>>
      UPDATE emp SET ...
   END IF;
```

GOTO 文では、カレント・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すが、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、カレント・ブロックには分岐できません。

GOTO 文を使用してカーソル 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 つのラベルは一度のみ使用できます。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そのラベルを使用できます。ターゲット・ラベルがカレント・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 ブロックの前に付けてください。たとえば、次の GOTO 文は許可されません。

```
FOR ctr IN 1..50 LOOP
   DELETE FROM emp WHERE ...
   IF SQL%FOUND THEN
      GOTO end_loop;  -- not allowed
   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。

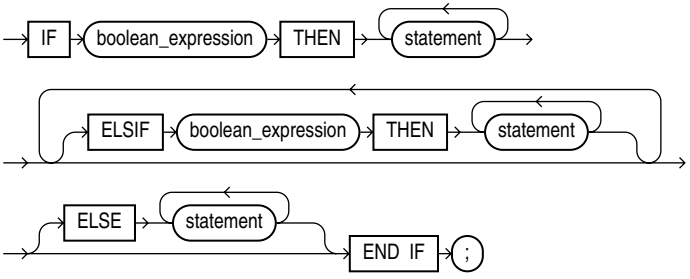
```
FOR ctr IN 1..50 LOOP
   DELETE FROM emp WHERE ...
   IF SQL%FOUND THEN
      GOTO end_loop;
   END IF;
   ...
<<end_loop>>
NULL;   -- an executable statement that specifies inaction
END LOOP;
```

IF 文

IF 文を使用すると、一連の文を条件に合せて実行できます。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、4-3 ページの「条件制御 : IF 文および CASE 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 である場合にのみ実行される一連の文が対応付けられています。

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。

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。

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対応付けます。式の結果が TRUE の場合、対応付けられた一連の文が実行されます。

使用上の注意

IF 文には、IF-THEN、IF-THEN-ELSE および 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 になった場合にのみ実行されます。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、その後に一連の代替文を続けます。ブール式の結果が FALSE または NULL の場合にのみ、ELSE 句内の文の並びが実行されます。このように、ELSE 句では必ず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 の場合は、それに関連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 の場合は、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、一連の文が複数回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。ただし、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

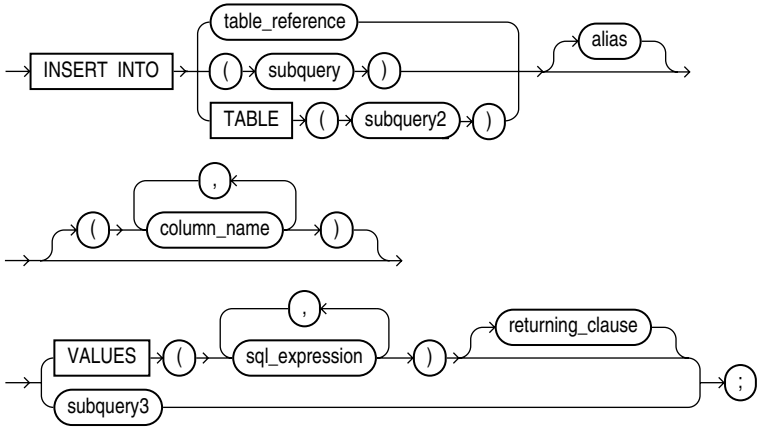
CASE 文、式

INSERT 文

INSERT 文は、指定されたデータベースの表に、新しい行データを追加します。INSERT 文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

構文

insert_statement



キーワードとパラメータの説明

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。ただし、列の名前をリストの中で指定できるのは一度のみです。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。

returning_clause

この句を使用すると、挿入された行から値を戻せるため、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、変数かホスト配列（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、13-55 ページの「DELETE 文」を参照してください。

sql_expression
任意の有効な SQL の式です。詳細は、『Oracle9*i* SQL リファレンス』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 句は使用できません。13-163 ページの「SELECT INTO 文」を参照してください。

subquery3
値または値の集合を戻す SELECT 文です。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す必要があります。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す必要があります。

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る必要があり、ユーザーは INSERT 権限を持つ必要があります。table_reference の構文は、13-55 ページの「DELETE 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。

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 つのみです。また、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の互換性が必要です。

使用上の注意
VALUES リストの中の文字リテラルと日付リテラルは、引用符（'）で囲む必要があります。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 を使用すると、INSERT 文の実行に関する有用な情報にアクセスできます。

例
様々な形式の 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

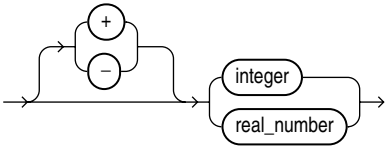
SELECT INTO 文

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。

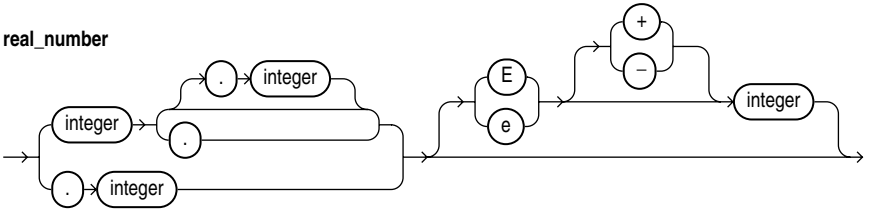
構文

numeric_literal

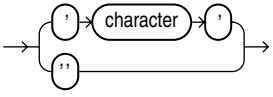


integer

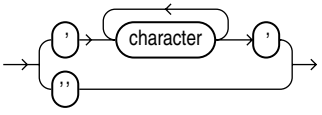
real_number



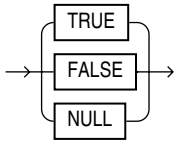
character_literal



string_literal



boolean_literal



キーワードとパラメータの説明

character

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。

digit

数字 0 ～ 9 のうちのいずれかです。

TRUE、FALSE、NULL

事前定義のブール値です。

使用上の注意

算術式では、整数と実数の 2 種類の数値リテラルを使用できます。整数と実数数値リテラルは、句読点文字で区切る必要があります。句読点以外に空白も使用できます。

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、数字、空白および特殊記号を使用できます。

文字リテラルの中で、PL/SQL は大 / 小文字を区別します。たとえば、PL/SQL はリテラル `'Q'` と `'q'` を異なるものとみなします。

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する場合は、引用符（'）を 2 つ入力します。文字列リテラルの中で、PL/SQL は大 / 小文字を区別します。たとえば、PL/SQL はリテラル `'white'` と `'White'` を異なるものとみなします。

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 '` は異なります。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

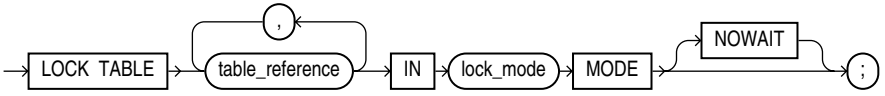
定数と変数、式

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、表アクセスの共有や拒否ができます。詳細は、6-49 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ある必要があります。table_reference の構文は、13-55 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ため、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。

使用上の注意

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

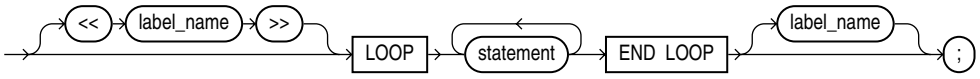
COMMIT 文、ROLLBACK 文

LOOP 文

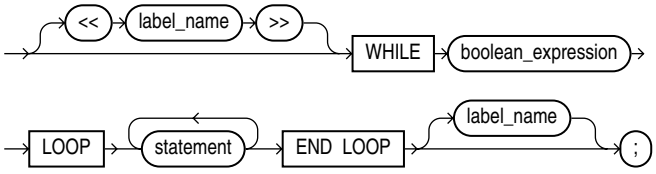
LOOP 文は一連の文を複数回実行します。ループとは、繰り返して実行する一連の文をキーワードで囲んだものです。PL/SQL では、基本ループ、WHILE ループ、FOR ループ、カーソル FOR ループの 4 種類がサポートされています。詳細は、4-9 ページの「反復制御 : LOOP 文と EXIT 文」を参照してください。

構文

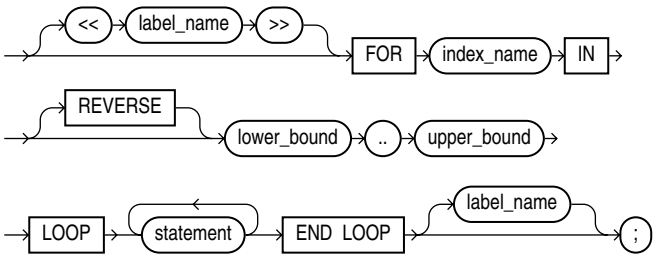
basic_loop_statement



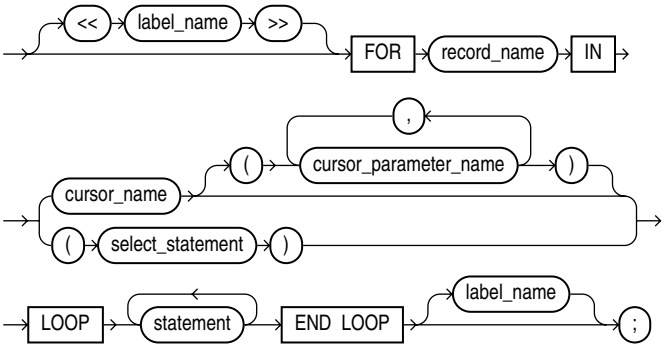
while_loop_statement



for_loop_statement



cursor_for_loop_statement



キーワードとパラメータの説明

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 文を使用し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。

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 である場合にのみ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、13-69 ページの「式」を参照してください。

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一連の作業を繰り返し、すべての行を処理した後にカーソルをクローズします。

cursor_name

現行の有効範囲の中で、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。

cursor_parameter_name

カーソルのパラメータ、つまり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、13-51 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用できる場所であれば、どこででも使用できます。カーソルの仮パラメータは IN パラメータにしてください。

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。繰返しの範囲は、キーワード FOR と LOOP に囲まれた反復スキームの一部です。

繰返しの範囲は 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付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ります）。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オーバーライドします。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用すると参照できます。

```
<<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、式の中で使用できますが、値を代入できません。

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 を使用する場合は、二重の山カッコで囲み、ループの先頭に置く必要があります。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。カレント・ループのみでなく、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用すれば参照できます。

```
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用さ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;
```

lower_bound .. upper_bound

結果が整数値になる式です。それ以外の場合、PL/SQL は事前定義の例外 VALUE_ERROR を呼び出します。式は、最初にループに入ったときにのみ評価されます。次の例に示すように、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 である必要があります。

```
FOR i IN -5..10 LOOP
   ...
END LOOP;
```

内部的に、PL/SQL は PLS_INTEGER 一時変数に境界の値を代入します。さらに、必要に応じてその値を最も近い整数に四捨五入します。PLS_INTEGER の大きさの範囲は、+/-2**31 です。このため、範囲外の数値を評価した場合、PL/SQL が代入をすると、数値オーバーフローのエラーが発生します。

デフォルトでは、ループ索引には lower_bound の値が代入されます。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 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。超えた時点で、ループが終了します。

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 または select_statement によって取り出された行と同じ構造を持ちます。

レコードはループの内側のみ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オーバーライドします。そのため、同じ名前のレコードは、ラベルを使用しないかぎ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用します。

```
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、単純名にしてください。また、それらが式である場合は、別名を持つ必要があります。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+comm wages, job ...
```

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。

次に例を示します。

```
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。下回った時点で、ループが終了します。

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 句は使用できません。13-163 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ため、暗黙カーソル SQL は適用されません。

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。

使用上の注意

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 文を使用し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 c1rec IN c1 LOOP
      bonus := (c1rec.sal * 0.05) + (c1rec.comm * 0.25);
      INSERT INTO bonuses VALUES (c1rec.empno, bonus);
   END LOOP;
   COMMIT;
END;
```

関連項目

カーソル、EXIT 文、FETCH 文、OPEN 文、%ROWTYPE 属性

MERGE 文

MERGE 文は、一部の行の挿入と他の行の更新を 1 回の操作で行います。ターゲット表に対して更新と挿入のどちらを行うかは、結合条件に基づいて決定されます。つまり、ターゲット表のうち結合条件と一致する既存の行は更新され、それ以外の場合は別個の副問合せからの値を使用して 1 行が挿入されます。

この文の構文と詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

使用上の注意

この文は、主として大量のデータが頻繁に挿入および更新されるデータ・ウェアハウスの場合に有効です。

例

次の例では、デフォルトのボーナスを 100 として、サンプル・スキーマ oe にボーナス表を作成しています。次に、売上があった全従業員を（oe.orders 表の sales_rep_id column に基づいて）ボーナス表に挿入します。最後に、人事管理マネージャが、全従業員がボーナスの対象となるように決定します。

- 売上がなかった従業員の場合、ボーナスは給与の 1% です。
- 売上があった従業員の場合、ボーナスは給与の 1% 相当がデフォルトのボーナスより増額されます。

MERGE 文はこのような変更を 1 度に実装します。

```
CREATE TABLE bonuses (employee_id NUMBER, bonus NUMBER DEFAULT 100);
INSERT INTO bonuses(employee_id)
(SELECT e.employee_id FROM employees e, orders o
WHERE e.employee_id = o.sales_rep_id
GROUP BY e.employee_id);
SELECT * FROM bonuses;
EMPLOYEE_ID BONUS
----------- ----------
153 100
154 100
155 100
156 100
158 100
159 100
160 100
161 100
163 100
MERGE INTO bonuses D
  USING (SELECT employee_id, salary, department_id FROM employees
         WHERE department_id = 80) S
  ON (D.employee_id = S.employee_id)
```

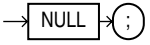
```
  WHEN MATCHED THEN UPDATE SET D.bonus = D.bonus + S.salary*.01
  WHEN NOT MATCHED THEN INSERT (D.employee_id, D.bonus)
    VALUES (S.employee_id, S.salary*0.1);

EMPLOYEE_ID BONUS
----------- ----------
153         180
154         175
155         170
156         200
158         190
159         180
160         175
161         170
163         195
157         950
145         1400
170         960
179         620
152         900
169         1000
...
```

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用します。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造体では、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、4-21 ページの「NULL 文」を参照してください。

構文

null_statement



使用上の注意

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 つの実行可能文が必要です。NULL 文はこの条件を満たします。つまり、構文上は句が必要でもアクションは必要ない場合には、NULL 文を使用できます。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、販売員のみ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、データの操作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。**コンストラクタ**と呼ばれる特殊な種類のメソッドは、オブジェクト型の新規インスタンスを作成し、その属性を記入します。

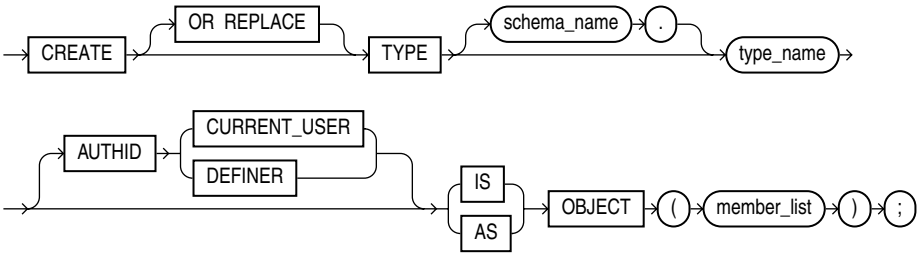
オブジェクト型は SQL を介して作成し、Oracle データベースに格納する必要があります。多くのプログラムがデータベース内のオブジェクト型を共有できます。CREATE TYPE 文を使用してオブジェクト型を定義する場合は、実世界のオブジェクトに対応する抽象テンプレートを作成します。テンプレートでは、アプリケーション環境でオブジェクトに必要な属性と動作を指定します。

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（クライアント・プログラムから参照可能）です。しかし、正しいプログラムは、それを直接操作しません。かわりに、提供される一連のメソッドを使用します。その結果、データは常に適切な状態に保たれます。

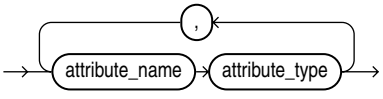
オブジェクト型の使用の詳細は、第 10 章を参照してください。

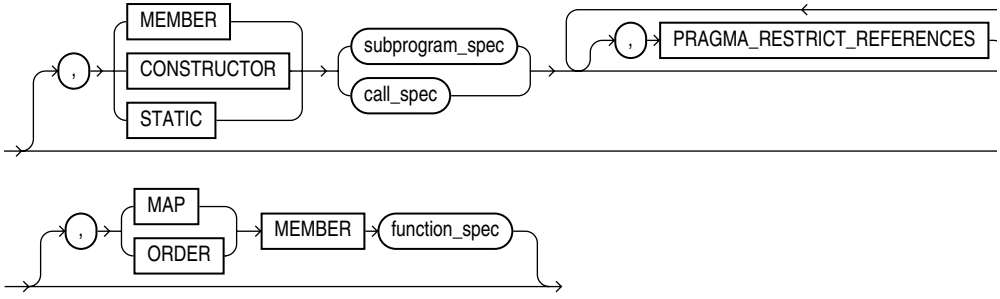
構文

object_type_declaration | object_type_spec

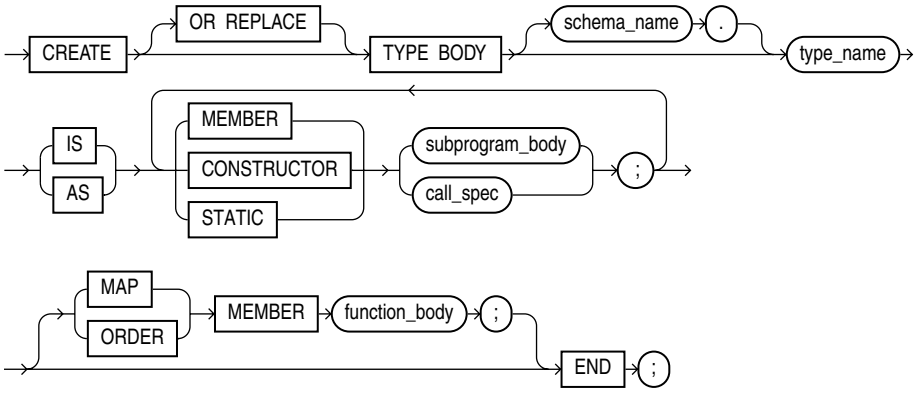


member_list





object_type_body



キーワードとパラメータの説明

attribute_datatype

これは、LONG、LONG RAW、ROWID、UROWID、PL/SQL 固有の BINARY_INTEGER 型とそのサブタイプ、BOOLEAN、PLS_INTEGER、RECORD、REF CURSOR、%TYPE および %ROWTYPE を除く任意の Oracle データ型と、PL/SQL パッケージ内に定義されている型です。

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ある必要があります（他のオブジェクト型内では使用できます）。属性の宣言内では、代入演算子または DEFAULT 句を使用しての属性の初期化はできません。また、属性に NOT NULL 制約を課すことはできません。

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、8-50 ページの「実行者権限と定義者権限」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 コール仕様を作成する方法は、『Oracle9*i* 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 コール仕様を作成する方法は『Oracle9*i* 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。

function_body

CONSTRUCTOR、MEMBER または STATIC ファンクションの基盤となる実装を定義します。function_body の構文は、13-88 ページの「ファンクション」を参照してください。

MAP

このキーワードは、メソッドによるオブジェクトの順序付けが、オブジェクトを事前定義の順序を持つ CHAR や REAL などのスカラー・データ型の値にマップすることで行われることを示します。PL/SQL は、この順序付けを使用して、x > y などのブール式を評価し、DISTINCT、GROUP BY および ORDER BY 句によって暗黙的に必要となる比較を実行します。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 つのみです。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ある必要があります。

MEMBER | CONSTRUCTOR | 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。コンストラクタ・メソッドには、オブジェクト型と同じ名前が必要です。一方、メンバー・メソッドと静的メソッドには、オブジェクト型またはその属性とは異なる名前が必要です。

MEMBER メソッドはオブジェクトのインスタンスで起動され、その特定のインスタンスの属性の読込みや変更を行います。

```
object_instance.method();
```

CONSTRUCTOR メソッドはオブジェクトの新規インスタンスを作成し、一部またはすべての属性を記入します。

```
object_instance := new object_type_name(attr1 => attr1_value,
  attr2 => attr2_value);
```

各オブジェクトの属性に対して 1 つのパラメータを持つデフォルト・コンストラクタ・メソッドは、システムによって定義されます。したがって、独自のコンストラクタ・メソッドの定義が必要なのは、異なるパラメータのセットに基づいてオブジェクトを構成する場合のみです。

STATIC メソッドはオブジェクト型で（特定のオブジェクト・インスタンスではなく）起動されます。したがって、このメソッドはオブジェクトの属性に関係のないグローバルな操作に制限する必要があります。

```
object_type.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る必要があります。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る必要があります。

CONSTRUCTOR メソッドと 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。このパラメータはオブジェクト型のインスタンスです。暗黙的宣言か明示的宣言かにかかわらず、このパラメータは常に MEMBER メソッドに渡される最初のパラメータです。ただし、STATIC メソッドは SELF の受入れや参照ができません。

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。コンストラクタ・ファンクションの場合、SELF のパラメータは、常に IN OUT モードです。MEMBER ファンクションで 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。MEMBER プロシージャで 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・モードを指定できません。

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オーダー・メソッドは 1 つのみです。このメソッドは、戻り型が数値のファンクションである必要があります。

オーダー・メソッドはいずれも、組込みパラメータ 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。オーダー・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。

pragma_restrict_refs

この RESTRICT_REFERENCES プラグマを使用して、純正規則に違反していないか確認できます。メンバー関数は、副作用を制御するための規則に従っている場合にのみ、SQL 文からコールできます。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。プラグマの構文は、13-153 ページの「RESTRICT_REFERENCES プラグマ」を参照してください（この場合、プラグマの終了記号は省略してください。）

プラグマは、メンバー関数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）に対する読込みや書込み、またはそのいずれも行っていないことを示します。純正規則と RESTRICT_REFERENCES プラグマの詳細は、『Oracle9*i* 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。

schema_name

この修飾子はオブジェクト型の入ったスキーマを識別します。schema_name を省略すると、オブジェクト型はスキーマに入っているとみなされます。

subprogram_body

MEMBER または STATIC ファンクションあるいはプロシージャの基盤となる実装を定義します。構文は、function_body や procedure_body の構文と似ていますが、終了記号は使用できません。13-88 ページの「ファンクション」または 13-140 ページの「プロシージャ」を参照してください。

subprogram_spec

CONSTRUCTOR、MEMBER または STATIC 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を宣言します。構文は、function_spec や procedure_spec の構文と似ていますが、終了記号は使用できません。13-88 ページの「ファンクション」または 13-140 ページの「プロシージャ」（あるいはその両方）を参照してください。

type_name

データ型指定子 OBJECT を使用して定義したユーザー定義のオブジェクト型を識別します。

使用上の注意

オブジェクト型を宣言してスキーマにインストールすると、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、それを使用し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用すると、オブジェクト属性、表の列、PL/SQL 変数、バインド変数、レコード・フィールド、コレクション要素、プロシージャの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の 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実装します。

メソッドを使用する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する必要があります。サブプログラムのみが実装を必要とします。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のみがある場合、またはそのいずれかの宣言のみがある場合、オブジェクト型本体は不要です。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（オブジェクト型の外側から参照可能）です。

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、その値を変更するには、ドット表記法を使用します。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のコールで空のパラメータ・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・メソッドかオーダー・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。ただし、どちらのメソッドも宣言しなければ、オブジェクトは SQL 文でのみ

比較し、しかも比較できるのは等しいか等しくないかについてのみです。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のみです。

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用できます。

どのオブジェクト型にもデフォルト・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。このコンストラクタは、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用します。異なるパラメータのセットを受け入れる独自のコンストラクタ・メソッドを定義することもできます。PL/SQL は、暗黙的にコンストラクタをコールすることはないため、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。

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実装は、整数の配列を使用します。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
         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;
```

メソッド push および pop では、組込みプロシージャ raise_application_error によ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できます。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;
```

次の例は、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,
);
```

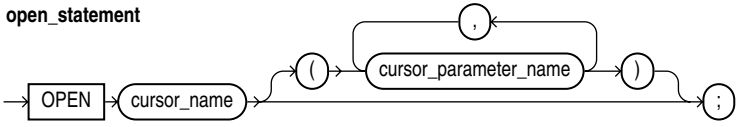
関連項目

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用するリソースを割り当て、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、6-6 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、つまり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、13-51 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用できる場所であれば、どこででも使用できます。

使用上の注意

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のみ解析されます。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のみ行われます。解析された SQL 文は、すべてキャッシュに入れられます。SQL 文を再度解析する必要があるのは、新しい SQL 文によってキャッシュから押し出された場合のみです。

したがって、カーソルを再オープンするには、まずクローズする必要がありますが、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。行のフェッチには FETCH 文を使用します。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す必要があります。カーソルの仮パラメータは IN パラメータにしてください。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用されます。仮パラメータと実パラメータのデータ型には、互換性が必要です。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なければ、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つ必要があります。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用して、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。

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 VARCHAR2, 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

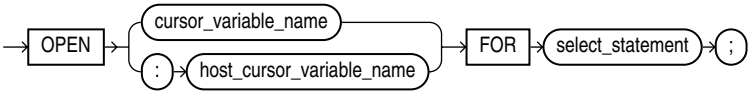
CLOSE 文、カーソル、FETCH 文、LOOP 文

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用するリソースを割り当て、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、6-16 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てください。

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。

select_statement の構文は、13-163 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。

使用上の注意

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で、ネットワークの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;
```

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のみでなく問合せ全体を渡すことができるため、柔軟性はあります。

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で、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）を使用して、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 モードを指定する必要があります。この指定によって、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

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 EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp, choice IN INT);
END emp_data;

CREATE PACKAGE BODY emp_data AS
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp, choice IN INT) 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;
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。次に例を示します。

```
CREATE PACKAGE admin_data AS
   TYPE GenCurTyp IS REF CURSOR;
   PROCEDURE open_cv (generic_cv IN OUT GenCurTyp, choice INT);
END admin_data;

CREATE PACKAGE BODY admin_data AS
   PROCEDURE open_cv (generic_cv IN OUT GenCurTyp, choice INT) 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;
END admin_data;
```

関連項目

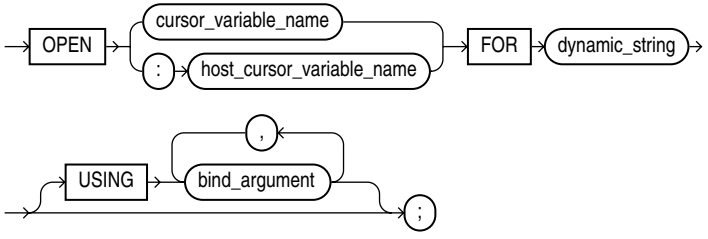
CLOSE 文、カーソル変数、FETCH 文、LOOP 文

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て、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の前に配置してから、%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、11-7 ページの「OPEN-FOR、FETCH および CLOSE 文の使用」を参照してください。

構文

open_for_using_statement



キーワードとパラメータの説明

cursor_variable_name
現行の有効範囲内でそれより前に宣言されている、弱い型指定の（戻り型を持たない）カーソル変数です。

bind_argument
動的 SELECT 文に渡される値を持つ式です。

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。

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付けてください。

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。

使用上の注意

動的な複数行の問合せを処理するには、OPEN-FOR-USING 文、FETCH 文および 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行を取り出します。すべての行が処理された後に、CLOSE 文でカーソル変数をクローズします。

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。

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、回避策を使用する必要があります。11-16 ページの「NULL を渡す方法」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にのみ、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、それを表 emp から行を戻す動的 SELECT 文と対応付けます。

```
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

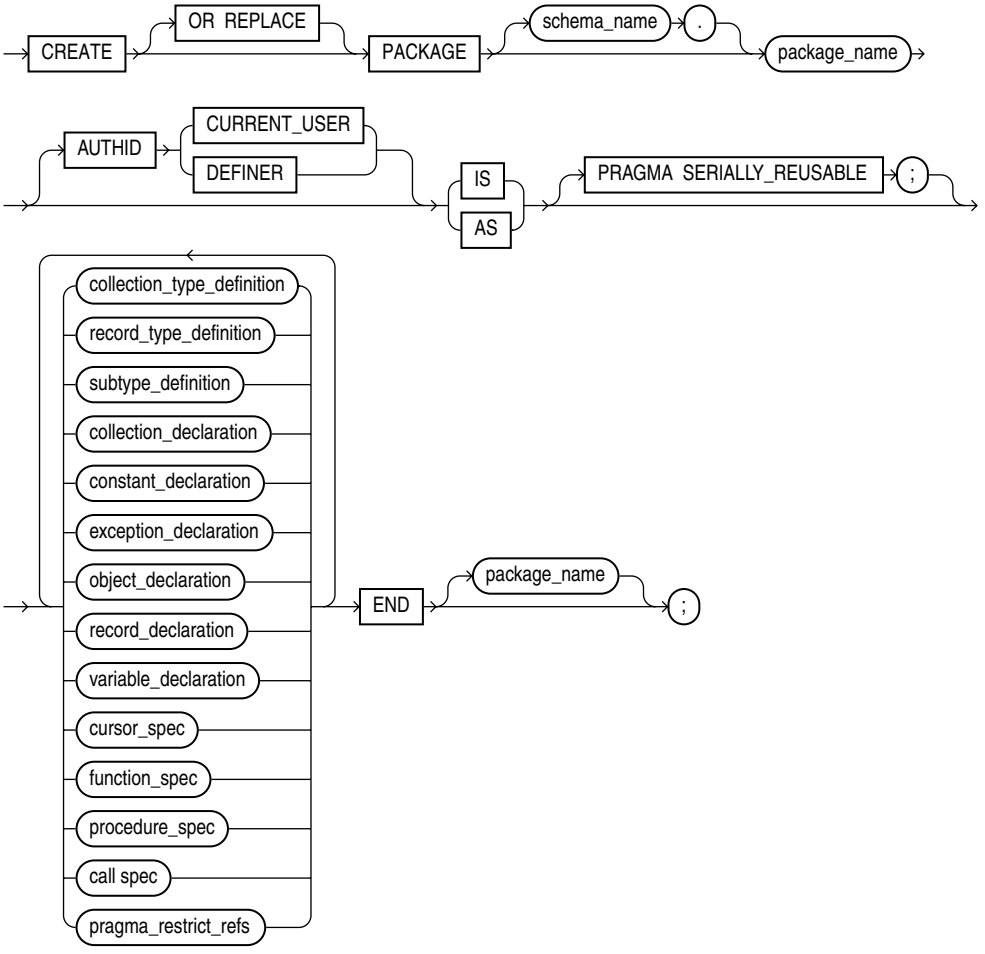
EXECUTE IMMEDIATE 文

パッケージ

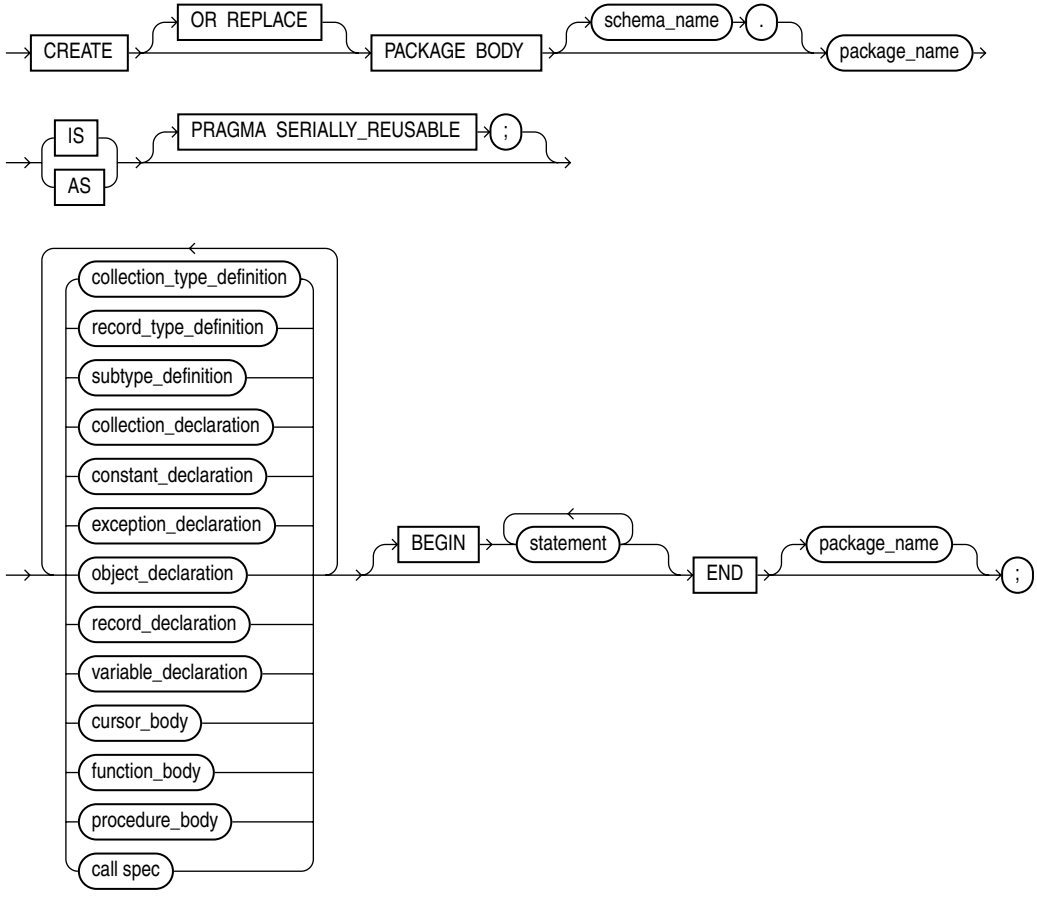
パッケージとは、論理的に関連する PL/SQL の型、項目およびサブプログラムをグループにまとめたスキーマ・オブジェクトのことです。パッケージには、仕様部と本体の 2 つの部分があります。詳細は、第 9 章を参照してください。

構文

package_declaration | package_spec



package_body



キーワードとパラメータの説明

AUTHID
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、8-50 ページの「実行者権限と定義者権限」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。詳細は、『Oracle9*i* Java ストアド・プロシージャ開発者ガイド』または『Oracle9*i* アプリケーション開発者ガイド - 基礎編』（あるいはその両方）を参照してください。

collection_declaration
コレクション（ネストした表、索引付き表または VARRAY）を宣言します。`collection_declaration` の構文は、13-27 ページの「コレクション」を参照してください。

collection_type_definition
データ型指定子 `TABLE` または `VARRAY` を使用してコレクション型を定義します。

constant_declaration
定数を宣言します。`constant_declaration` の構文は、13-36 ページの「定数と変数」を参照してください。

cursor_body
明示カーソルの基盤となる実装を定義します。`cursor_body` の構文は、13-51 ページの「カーソル」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を宣言します。`cursor_spec` の構文は、13-51 ページの「カーソル」を参照してください。

exception_declaration
例外を宣言します。`exception_declaration` の構文は、13-60 ページの「例外」を参照してください。

function_body

ファンクションの基盤となる実装を定義します。function_body の構文は、13-88 ページの「ファンクション」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を宣言します。function_spec の構文は、13-88 ページの「ファンクション」を参照してください。

object_declaration

オブジェクト（オブジェクト型のインスタンス）を宣言します。object_declaration の構文は、13-116 ページの「オブジェクト型」を参照してください。

package_name

データベースに格納されたパッケージを識別します。命名規則は、2-4 ページの「識別子」を参照してください。

pragma_restrict_refs

この RESTRICT_REFERENCES プラグマを使用して、純正規則に違反していないか確認できます。ファンクションは、副作用を制御するための規則に従っている場合にのみ、SQL 文からコールできます。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。プラグマの構文は、13-153 ページの「RESTRICT_REFERENCES プラグマ」を参照してください。

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）に対する読込みや書込み、またはそのいずれも行っていないことを示します。純正規則と RESTRICT_REFERENCES プラグマの詳細は、『Oracle9*i* 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。

PRAGMA SERIALLY_REUSABLE

このプラグマを使用すると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）の間のみその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。詳細は、Oracle9*i* 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。

procedure_body

プロシージャの基盤となる実装を定義します。procedure_body の構文は、13-140 ページの「プロシージャ」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を宣言します。procedure_spec の構文は、13-140 ページの「プロシージャ」を参照してください。

record_declaration
ユーザー定義のレコードを宣言します。record_declaration の構文は、13-148 ページの「レコード」を参照してください。

record_type_definition
データ型指定子 RECORD または属性 %ROWTYPE を使用してレコード型を定義します。

schema_name
この修飾子はパッケージの入ったスキーマを識別します。schema_name を省略すると、パッケージはスキーマに入っているとみなされます。

variable_declaration
変数を宣言します。variable_declaration の構文は、13-36 ページの「定数と変数」を参照してください。

使用上の注意
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。だたし、PL/SQL をサポートする Oracle のツール製品を使用すると、パッケージを作成し、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 SQL*Plus から対話形式で発行できます。

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、変数、定数、例外、カーソルおよび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、仕様を実装します。

下位の実装（定義）を持つのは、サブプログラムとカーソルのみです。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のみであれ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用して、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する必要があります。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る必要があります。一致していない場合は、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

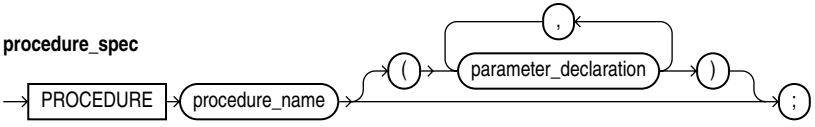
プロシージャ

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用します。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（または AS）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。

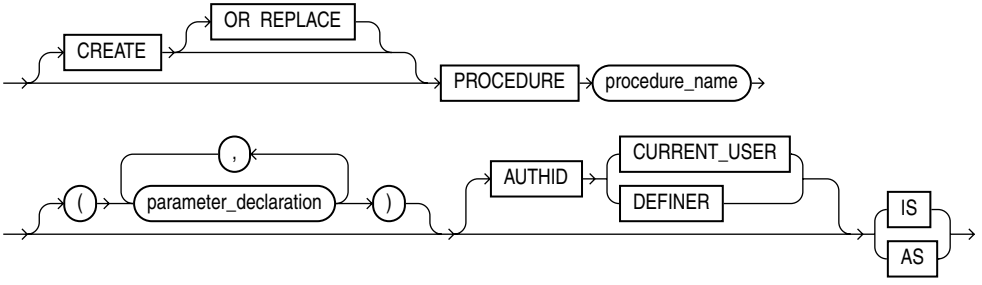
プロシージャ本体には、宣言部（オプション）、実行部、例外処理部（オプション）の 3 つの部分があります。宣言部には、型、カーソル、定数、変数、例外およびサブプログラムが含まれます。これらの項目はローカルで、プロシージャ を終了すると消去されます。実行部には、値の代入、実行の制御および Oracle データの操作を実行する文があります。例外処理部には、実行の途中で呼び出された例外を処理する例外ハンドラがあります。詳細は、8-4 ページの「PL/SQL プロシージャ」を参照してください。

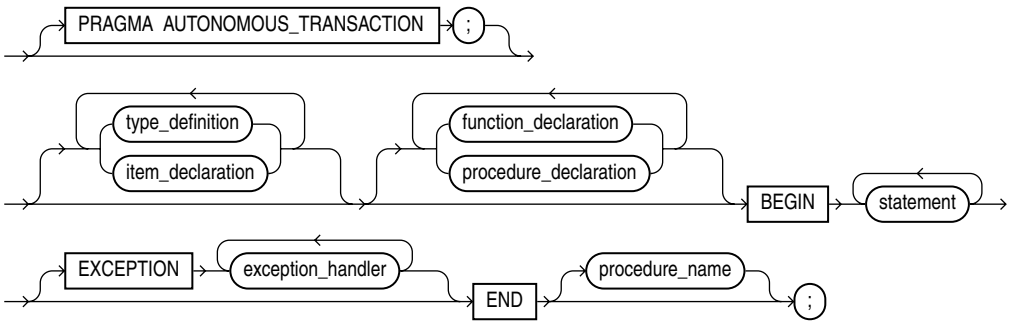
構文

procedure_spec

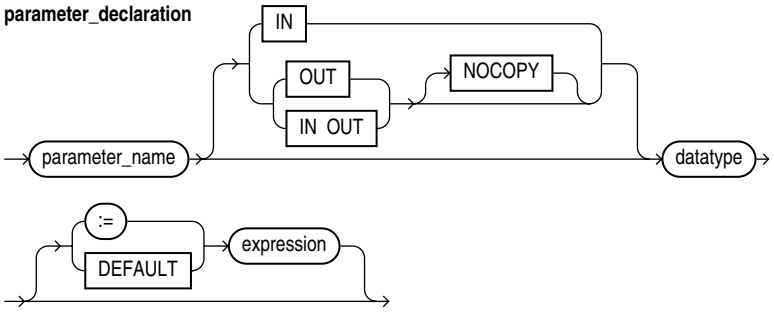


procedure_declaration | procedure body





parameter_declaration



キーワードとパラメータの説明

AUTHID

ストアド・プロシージャをその所有者（デフォルト）と現行ユーザー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所有者と現行ユーザーのどちらのスキーマで解決されるかを決定します。CURRENT_USER を指定すると、デフォルトの動作を変更できます。詳細は、8-50 ページの「実行者権限と定義者権限」を参照してください。

CREATE

オプションの CREATE 句を使用すると、Oracle データベースに格納できるスタンドアロン・プロシージャを作成できます。CREATE 文は、SQL*Plus またはシステム固有の動的 SQL を使用したプログラムから対話形式で実行できます。

datatype

これは、型指定子です。datatype の構文は、13-36 ページの「定数と変数」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、13-60 ページの「例外」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に、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のデータ型には互換性が必要です。

function_declaration

ファンクションを宣言します。function_declaration の構文は、13-88 ページの「ファンクション」を参照してください。

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用します。OUT パラメータは、サブプログラムのコール元に値を戻すために使用します。IN OUT パラメータを使用すると、コール先のサブプログラムに初期値を渡して、コール元に更新された値を戻すことができます。

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、13-10 ページの「ブロック」を参照してください。

NOCOPY
コンパイラ・ヒント（ディレクティブではなく）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、8-17 ページの「NOCOPY コンパイラ・ヒントを使用した大型データ構造の受渡し」を参照してください。

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。

AUTONOMOUS_TRANSACTION プラグマ
このプラグマはファンクションを自律型（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。自律型トランザクションは、別のトランザクション、メイン・トランザクション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。詳細は、6-52 ページの「自律型トランザクションによる独立した作業単位の実行」を参照してください。

procedure_name
ユーザー定義のプロシージャです。

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、13-10 ページの「ブロック」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用すると、IN パラメータをデフォルト値に初期化できます。

使用上の注意

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できます。パラメータ・モードの概要は、8-16 ページの表 8-1 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、8-19 ページの「サブプログラムのパラメータのデフォルト値の使用」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。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。ただし、サブプログラムをコールした時点で、値は失われます（コンパイラ・ヒント NOCOPY を指定しない場合、またはサブプログラムを未処理例外で終了しない場合）。

プロシージャの仕様部と本体を合せて 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、実装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のみコールできます。

プロシージャの実行部には、少なくとも 1 つの文が存在している必要があります。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、名前表記法を使用し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

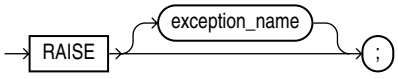
コレクション・メソッド、ファンクション、パッケージ

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、事前定義済み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。ただし、事前定義済みの例外を 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す必要があります。詳細は、7-7 ページの「独自の PL/SQL 例外の定義」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

事前定義済み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、7-4 ページの「事前定義の PL/SQL 例外」を参照してください。

使用上の注意

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のみにしてください。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カレント・ブロックで発見できない場合は、例外が伝播します。つまり、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、現行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のみ許されます。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、カレント・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

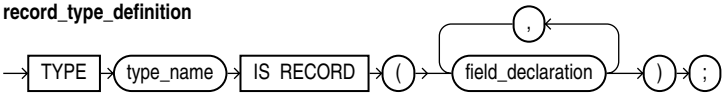
例外

レコード

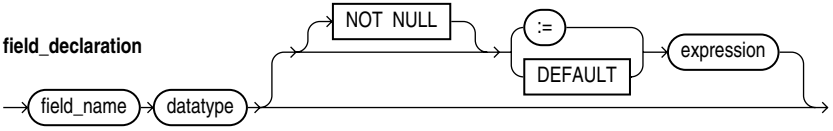
レコードは、RECORD 型の項目です。レコードには、様々な型のデータ値を格納できる、他と重複しない名前のフィールドがあります。このため、レコードによって、関連してはいるが異なるデータを 1 つの論理単位として扱うことができます。詳細は、5-51 ページの「レコード」を参照してください。

構文

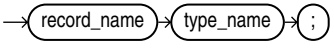
record_type_definition



field_declaration



record_declaration



キーワードとパラメータの説明

datatype
型指定子です。datatype の構文は、13-36 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。expression の構文は、13-69 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のデータ型には互換性が必要です。

field_name
ユーザー定義のレコード内のフィールドを識別します。

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く必要があります。

record_name
ユーザー定義のレコードを識別します。

type_name
データ型指定子 RECORD を使用して定義したユーザー定義のレコード型を識別します。

:= | DEFAULT
この演算子またはキーワードを使用すると、フィールドをデフォルト値に初期化できます。

使用上の注意

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 属性を使用して、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 を代入できないようにしています。NOT NULL と宣言されたフィールドは、初期化されている必要があります。

```
DECLARE
   TYPE DeptRecTyp IS RECORD(
      deptno NUMBER(2) NOT NULL := 99,
      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、ドット表記法を使用します。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のみでは不十分です。）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 文を使用し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   VARCHAR2(20),
      purpose VARCHAR2(50));
   TYPE PartyTyp IS RECORD(
      day     DATE,
      time_of TimeTyp,   -- nested record
      place   VARCHAR2(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、通常の有効範囲規則とインスタンス化の規則に従います。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化され、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、次の構文を使用し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  VARCHAR2(14),
      loc    VARCHAR2(13));
   dept_rec DeptRecTyp;
   ...
BEGIN
   SELECT deptno, dname, loc INTO dept_rec FROM dept
      WHERE deptno = 20;
```

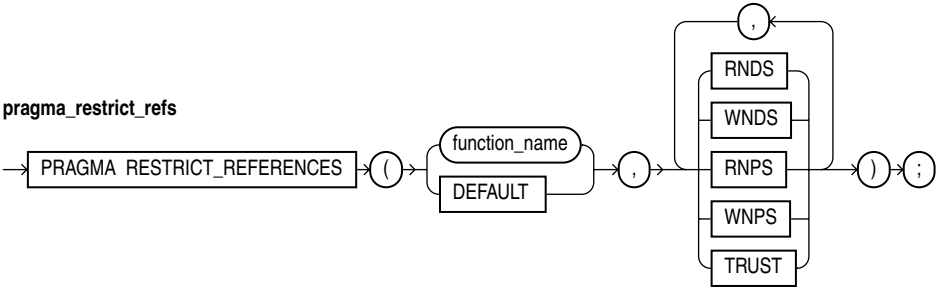
関連項目

コレクション、ファンクション、パッケージ、プロシージャ

RESTRICT_REFERENCES プラグマ

ストアド・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規則に従っている場合にのみ、SQL 文からコールできます。（8-9 ページの「PL/SQL サブプログラムの副作用の制御」を参照。）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。この規則に違反していないかどうかを確認するには、RESTRICT_REFERENCES プラグマ（コンパイラ・ディレクティブ）を使用します。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）に対する読込みや書込み、またはそのいずれも行っていないことを示します。詳細は、『Oracle9*i* 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。

構文



キーワードとパラメータの説明

DEFAULT
プラグマがパッケージ仕様部またはオブジェクト型の仕様部の中のすべてのファンクションに適用されるように指定します。その場合でも、ファンクションごとにプラグマを宣言できます。これらのプラグマは、デフォルトのプラグマをオーバーライドします。

function_name
ユーザー定義ファンクションを識別します。

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のみです。

RNDS
ファンクションがデータベース読込み禁止状態である（データベース表を問合せできない）ことを示します。

RNPS
ファンクションがパッケージ読込み禁止状態である（パッケージ変数の値を参照できない）ことを示します。

TRUST
ファンクションが 1 つ以上の規則に違反しないと信頼されていることを示します。

WNDS
ファンクションがデータベース書込み禁止状態である（データベース表を変更できない）ことを示します。

WNPS
ファンクションがパッケージ書込み禁止状態である（パッケージ変数の値を変更できない）ことを示します。

使用上の注意

RESTRICT_REFERENCES プラグマは、パッケージ仕様部またはオブジェクト型仕様部でのみ宣言できます。制約は 4 つまで（RNDS、RNPS、WNDS、WNPS）任意の順序で指定できます。パラレル問合せからファンクションをコールするには、4 つの制約をすべて指定します。他の制約を暗黙的に指定する制約はありません。たとえば、WNPS は RNPS を暗黙的に指定しません。

TRUST を指定すると、ファンクション本体はプラグマにリストされた制約に違反しているかどうかチェックされません。ファンクションは制約に違反していないことを承認されます。

関数名でなく DEFAULT を指定した場合には、このプラグマはパッケージ仕様部またはオブジェクト型の仕様部（オブジェクト型の場合は、システム定義のコンストラクタも含む）の中のすべてのファンクションに適用されます。その場合でも、ファンクションごとにプラグマを宣言できます。これらのプラグマは、デフォルトのプラグマをオーバーライドします。

RESTRICT_REFERENCES プラグマは、1 つのファンクション宣言にのみ適用できます。このため、オーバーロードされたファンクションの名前を参照するプラグマは、最も近い先行のファンクション宣言に必ず適用されます。

例

たとえば次のプラグマは、パッケージ・ファンクション balance がデータベース書込み禁止状態（WNDS）およびパッケージ読込み禁止状態（RNPS）であることを示します。

```
CREATE PACKAGE loans AS
   ...
   FUNCTION balance RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES (balance, WNDS, RNPS);
END loans;
```

オーバーロードされたファンクションの名前を参照するプラグマは、最も近い先行のファンクション宣言に必ず適用されます。したがって、次の例ではプラグマは credit_ok の 2 番目の宣言に適用されます。

```
CREATE PACKAGE loans AS
   FUNCTION credit_ok (amount_limit NUMBER) RETURN BOOLEAN;
   FUNCTION credit_ok (time_limit DATE) RETURN BOOLEAN;
   PRAGMA RESTRICT_REFERENCES (credit_ok, WNDS);
   ...
END loans;
```

関連項目

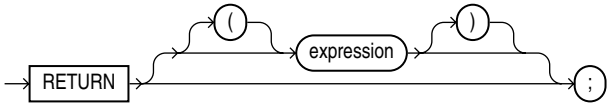
AUTONOMOUS_TRANSACTION プラグマ、EXCEPTION_INIT プラグマ、SERIALLY_REUSABLE プラグマ

RETURN 文

RETURN 文は、サブプログラムの実行を即座に完了させ、コール元に制御を戻します。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戻り値に代入します。詳細は、8-8 ページの「RETURN 文の使用」を参照してください。

構文

return_statement



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。

使用上の注意

RETURN 文を、ファンクションの仕様部の中で戻り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。ただし、サブプログラムに複数の終了点を作成す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、コール元に制御を戻す役割のみがあります。

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る必要があります。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。したがって、ファンクションには、RETURN 文へ導く少なくとも 1 つの実行パスが必要です。そうではない場合、PL/SQL により実行時に例外が呼び出されます。

RETURN 文を無名ブロックで使用して、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

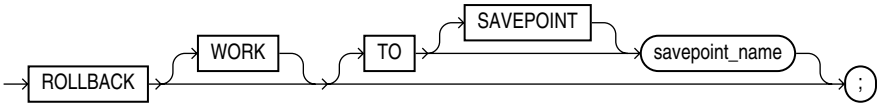
ファンクション、プロシージャ

ROLLBACK 文

ROLLBACK 文は COMMIT 文の逆です。これは、カレント・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、6-42 ページの「PL/SQL におけるトランザクション処理の概要」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、カレント・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。

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目的にのみ使用します。

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。命名規則は、2-4 ページの「識別子」を参照してください。

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目的にのみ使用します。

使用上の注意

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 のみが消去されます。

INSERT 文、UPDATE 文または DELETE 文を実行する前に、暗黙的な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、その暗黙的な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 文のみ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、次の文は許可されません。

```
ROLLBACK WORK FORCE '24.37.85';  -- not allowed
```

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、次の文は許可されません。

```
ROLLBACK WORK RELEASE;  -- not allowed
```

関連項目

COMMIT 文、SAVEPOINT 文

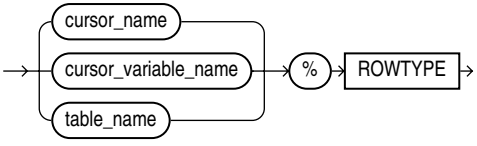
%ROWTYPE 属性

%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フェッチ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用できます。%ROWTYPE 属性を使用して宣言された変数は、データ型名を使用して宣言された変数と同じように扱われます。詳細は、2-14 ページの「%ROWTYPE の使用」を参照してください。

構文

rowtype_attribute



キーワードとパラメータの説明

cursor_name

現行の有効範囲の中で、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

現行の有効範囲の中で事前に宣言されている、（弱い型指定ではなく）強い型指定を持つ 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る必要があるデータベースの表（またはビュー）を識別します。

使用上の注意

%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、レコード全体の宣言が同じ表またはカーソルを参照している場合には、そのレコード全体の間で集計代入できます。2 番目の方法では、SELECT 文か FETCH 文を使用して、レコードに列値のリストを代入します。列名の順番は宣言された順番です。カーソルによって %ROWTYPE 属性を使用して取り出された選択項目は、単純名を持つ必要があります。また、選択項目が式の場合は別名を持つ必要があります。

例

次の例では %ROWTYPE を使用して 2 つのレコードを宣言しています。1 つ目のレコードには表 emp から選択された行が格納されます。2 つ目のレコードには、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

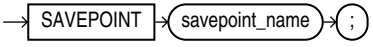
定数と変数、カーソル、カーソル変数、FETCH 文

SAVEPOINT 文

SAVEPOINT 文は、トランザクション処理の過程で、現行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、6-42 ページの「PL/SQL におけるトランザクション処理の概要」を参照してください。

構文

savepoint_statement



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。

使用上の注意

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。単純なロールバックまたはコミットでは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、再帰しながら進む過程で、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のみです。

INSERT 文、UPDATE 文または DELETE 文を実行する前に、暗黙的な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、その暗黙的な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 文のみ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

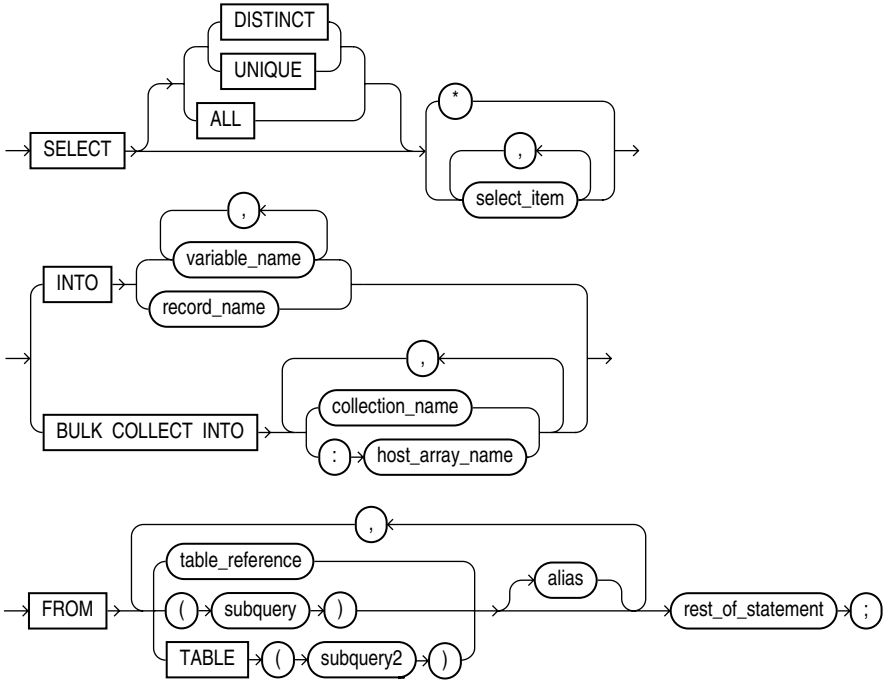
COMMIT 文、ROLLBACK 文

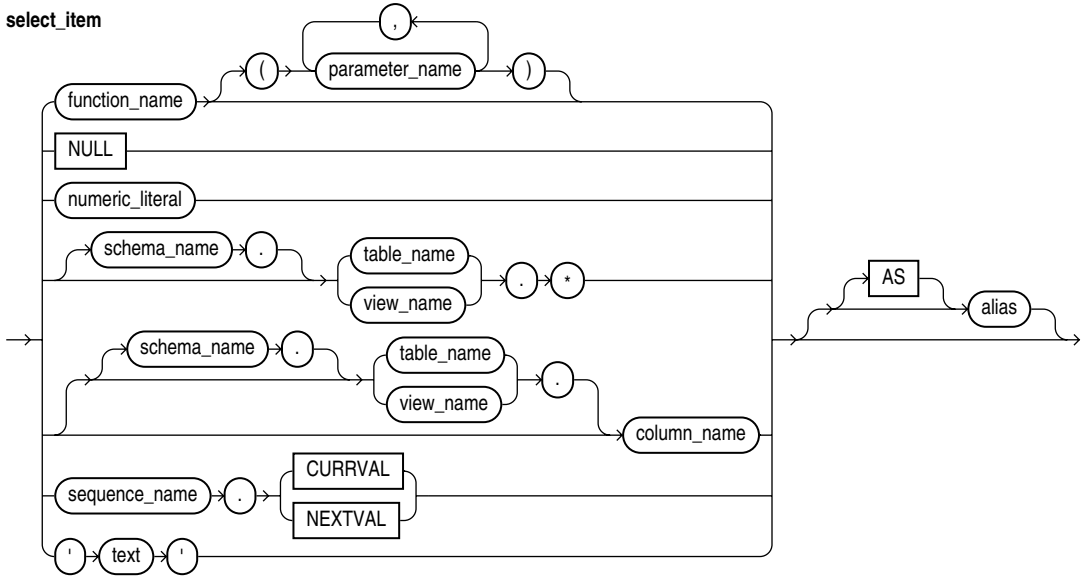
SELECT INTO 文

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、『Oracle9*i* 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

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。

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に、SQL エンジンに指示します。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。詳細は、5-38 ページの「バルク・バインドを使用したコレクションのループ・オーバーヘッドの削減」を参照してください。

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ている必要があります。

function_name
ユーザー定義ファンクションを識別します。

host_array_name
バルク・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ている必要があります。ホスト配列には、接頭辞としてコロンが必要です。

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。

parameter_name
ユーザー定義ファンクションの仮パラメータを識別します。

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ている必要があります。

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造体です。

schema_name
この修飾子は表またはビューの入ったスキーマを識別します。schema_name を省略すると、表またはビューはスキーマに入っているとみなされます。

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 句は使用できません。13-163 ページの「SELECT INTO 文」を参照してください。

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る必要があり、ユーザーが SELECT 権限を持っている必要があります。table_reference の構文は、13-55 ページの「DELETE 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 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。

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ている必要があります。

使用上の注意

BULK COLLECT 句は、コレクションを戻す前にバルク・バインド出力するように、SQL エンジンに指示します。これは、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、スカラー値またはオブジェクトなどの複合値が格納できます。

BULK COLLECT 句を使用せずに SELECT INTO 文を使用すると、1 行のみ戻されます。複数の行が戻った場合、PL/SQL によって事前定義の例外 TOO_MANY_ROWS が呼び出されます。

ただし、行が戻されなかった場合、PL/SQL によって NO_DATA_FOUND が呼び出されます。ただし、SELECT 文が AVG または SUM などの SQL 集計関数をコールした場合を除きます。(SQL 集計関数は必ず値または NULL を戻します。このため、集計関数をコールする SELECT INTO 文は NO_DATA_FOUND を呼び出しません。)

暗黙カーソル SQL とカーソル属性 %NOTFOUND、%FOUND、%ROWCOUNT および %ISOPEN を使用すると、SELECT INTO 文の実行に関する有用な情報にアクセスできます。

例

次の SELECT 文は、データベースの表 emp から従業員の名前、肩書および給与を戻します。

```
SELECT ename, job, sal INTO my_ename, my_job, my_sal FROM emp
   WHERE empno = my_empno;
```

次の例では、SQL エンジンは、ネストした表を PL/SQL 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;
```

関連項目

代入文、FETCH 文、%ROWTYPE 属性

SERIALLY_REUSABLE プラグマ

SERIALLY_REUSABLE プラグマを使用すると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）の間のみその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。詳細は、『Oracle9*i* 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。

構文

serially_reusable_pragma

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のみです。

使用上の注意

本体部のないパッケージを逐次再使用可能としてマークできます。パッケージに仕様部と本体がある場合は、両方ともマークする必要があります。本体のみをマークすることはできません。

逐次再使用可能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。

逐次再使用可能パッケージにはデータベース・トリガーからアクセスすることはできません。データベース・トリガーからアクセスすると、Oracle はエラーを生成します。

例

次の例では、逐次再使用可能パッケージを作成します。

```
CREATE PACKAGE pkg1 IS
   PRAGMA SERIALLY_REUSABLE;
   num NUMBER := 0;
   PROCEDURE init_pkg_state(n NUMBER);
   PROCEDURE print_pkg_state;
END pkg1;

CREATE PACKAGE BODY pkg1 IS
   PRAGMA SERIALLY_REUSABLE;
   PROCEDURE init_pkg_state (n NUMBER) IS
   BEGIN
      pkg1.num := n;
   END;
   PROCEDURE print_pkg_state IS
   BEGIN
      dbms_output.put_line('Num: ' || pkg1.num);
   END;
END pkg1;
```

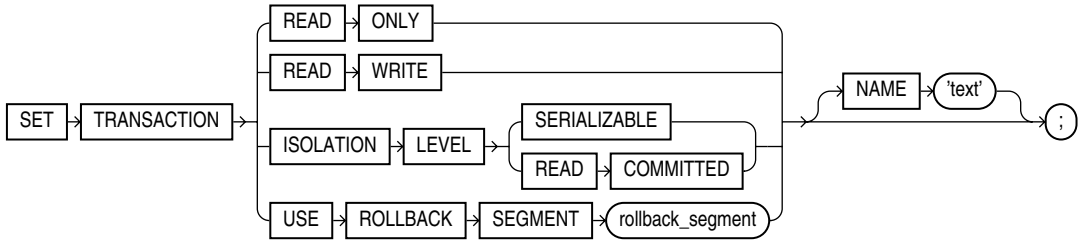
関連項目

AUTONOMOUS_TRANSACTION プラグマ、EXCEPTION_INIT プラグマ、RESTRICT_REFERENCES プラグマ

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・セグメントにカレント・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 つ以上の表に対して、複数の問合せを実行する場合に便利です。詳細は、6-47 ページの「SET TRANSACTION を使用したトランザクション・プロパティの設定」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

READ ONLY

カレント・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のみが見えます。READ ONLY を使用しても、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

カレント・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 を使用しても、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 DML 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 DML 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 DML 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、カレント・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は使用できません。読取り専用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。

NAME

トランザクションの名前またはコメント・テキストを指定できます。この指定した名前やコメント・テキストはトランザクションの実行中に使用可能で、長時間実行のインダウト・トランザクションをモニターしやすくなるため、COMMIT COMMENT 機能を使用するより適切です。

使用上の注意

SET TRANSACTION 文は、トランザクションの最初の SQL 文にする必要があり、そのトランザクションで 1 回しか使用できません。

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

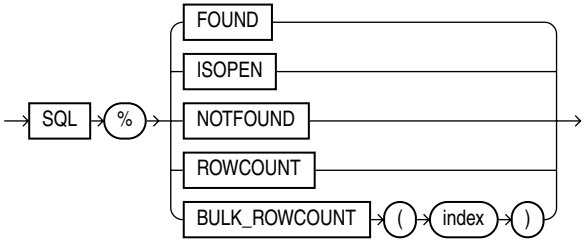
COMMIT 文、ROLLBACK 文、SAVEPOINT 文

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 カーソルには %FOUND、%ISOPEN、%NOTFOUND および %ROWCOUNT の 4 つの属性があります。これらの属性を使用すると、DML 文の実行についての情報が得られます。SQL カーソルには、FORALL 文で使用できるように設計された追加の属性 %BULK_ROWCOUNT および %BULK_EXCEPTIONS もあります。詳細は、6-6 ページの「カーソル管理」を参照してください。

構文

sql_cursor



キーワードとパラメータの説明

%BULK_ROWCOUNT

これは、FORALL 文での使用に設計された複合属性です。この属性は索引付き表の意味を持っています。*i* 番目の要素には、UPDATE 文または DELETE 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。

%BULK_EXCEPTIONS

%FOUND

INSERT 文、UPDATE 文または DELETE 文が 1 行以上の行に作用するか、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。それ以外の場合、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。それ以外の場合、FALSE となります。

%ROWCOUNT

この属性の結果は、INSERT 文、UPDATE 文または DELETE 文の影響を受けた行、あるいは SELECT INTO 文に戻された行の数になります。

SQL

Oracle 暗黙カーソルの名前です。

使用上の注意

カーソル属性は、プロシージャ文では使用できますが、SQL 文では使用できません。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もあります。したがって、属性の値を保存して後で使用する場合は、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りなく、PL/SQL によって事前定義済みの例外 NO_DATA_FOUND が呼び出されます。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。そのようなファンクションは、必ず値または NULL を戻します。このような場合、SQL%NOTFOUND の結果は FALSE になります。

%BULK_ROWCOUNT は、冗長になるのを防ぐためバルク挿入用に保持されません。たとえば、次の FORALL 文は反復ごとに 1 つの行を挿入します。つまり、反復するたびに %BULK_ROWCOUNT は 1 を戻します。

```
FORALL i IN 1..15
   INSERT INTO emp (sal) VALUES (sals(i));
```

バルク・バインドには、スカラー属性の %FOUND、%NOTFOUND および %ROWCOUNT を使用できます。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 文の最後の実行のみを参照します。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。

例

次の例では、更新される行がない場合に、%NOTFOUND を使用し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;
```

次に %BULK_ROWCOUNT を使用する例を示します。

```
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;
END;
```

関連項目

カーソル、カーソル属性

SQLCODE ファンクション

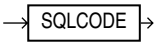
ファンクション SQLCODE は、直前に呼び出された例外に対応付けられている番号コードを戻します。SQLCODE は例外ハンドラでしか意味を持ちません。ハンドラの外側では、SQLCODE は常に 0 を戻します。

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「データが見つかりません。(*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 プラグマを使用して例外を Oracle エラー番号に関連付けている場合は例外です。この場合、SQLCODE はそのエラー番号を戻します。詳細は、7-18 ページの「エラー・コードとエラー・メッセージの取得 : SQLCODE および SQLERRM」を参照してください。

構文

sqlcode_function



使用上の注意

SQLCODE は呼び出された内部例外の識別に使用できるため、OTHERS 例外ハンドラの中で使用すると特に便利です。

SQLCODE は、SQL 文の中で直接使用することができません。まず、次のように SQLCODE の値をローカル変数に代入する必要があります。

```
my_sqlcode := SQLCODE;
...
INSERT INTO errors VALUES (my_sqlcode, ...);
```

RESTRICT_REFERENCES プラグマを使用してストアド・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 をコールする場合、WNPS および RNPS 制約は指定できません。

例

次の例では、SQLCODE の値を監査表に挿入します。

```
DECLARE
   my_sqlcode NUMBER;
BEGIN
   ...
EXCEPTION
   WHEN OTHERS THEN
      my_sqlcode := SQLCODE;
      INSERT INTO audits VALUES (my_sqlcode, ...);
END;
```

関連項目

例外、SQLERRM ファンクション

SQLERRM ファンクション

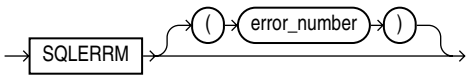
ファンクション SQLERRM は、エラー番号の引数に対応付けられているエラー・メッセージを戻します。引数が省略されている場合は、SQLCODE のカレント値と対応付けられているエラー・メッセージを戻します。引数なしの SQLERRM は、例外ハンドラの中でのみ意味があります。ハンドラの外側では、引数なしの SQLERRM は常にメッセージ「正常に完了しました（*normal, successful completion*）」を戻します。

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、SQLERRM はメッセージ「ユーザー定義の例外（*user-defined exception*）」を戻します。ただし、EXCEPTION_INIT プラグマを使用して例外を 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・メッセージを戻します。詳細は、7-18 ページの「エラー・コードとエラー・メッセージの取得：SQLCODE および SQLERRM」を参照してください。

構文

sqlerrm_function



キーワードとパラメータの説明

error_number

有効な Oracle エラー番号である必要があります。Oracle エラーのリストは、『Oracle9*i* データベース・エラー・メッセージ』を参照してください。

使用上の注意

SQLERRM は呼び出された内部例外の識別に使用できるため、OTHERS 例外ハンドラの中で使用すると特に便利です。

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戻します。SQLERRM に渡されるエラー番号は、負の値です。ゼロを渡すと、SQLERRM は常に次のメッセージを戻します。

```
ORA-0000: 正常に完了しました。
```

正数を渡すと、SQLERRM は常に次のメッセージを戻します。

```
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: データが見つかりません。
```

SQLERRM は、SQL 文の中で直接使用することができません。まず、次のように SQLERRM の値をローカル変数に代入する必要があります。

```
my_sqlerrm := SQLERRM;
...
INSERT INTO errors VALUES (my_sqlerrm, ...);
```

RESTRICT_REFERENCES プラグマを使用してストアド・ファンクションの純正度を示すときに、ファンクションが SQLERRM をコールする場合は WNPS および RNPS 制約を指定できません。

例

次の例では、文字列ファンクション SUBSTR を使用しているため、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。

```
DECLARE
   my_sqlerrm VARCHAR2(150);
   ...
BEGIN
   ...
EXCEPTION
   ...
   WHEN OTHERS THEN
   my_sqlerrm := SUBSTR(SQLERRM, 1, 150);
   INSERT INTO audits VALUES (my_sqlerrm, ...);
END;
```

関連項目

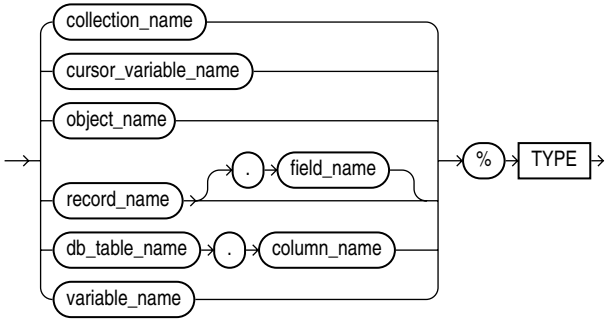
例外、SQLCODE ファンクション

%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用できます。詳細は、2-13 ページの「%TYPE の使用」を参照してください。

構文

type_attribute



キーワードとパラメータの説明

collection_name

現行の有効範囲のうち、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

現行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のみです。

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可能な表および列を参照します。

object_name

現行の有効範囲のうち、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

現行の有効範囲のうち、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

現行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

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。

使用上の注意

%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。ただし、%TYPE を使用し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

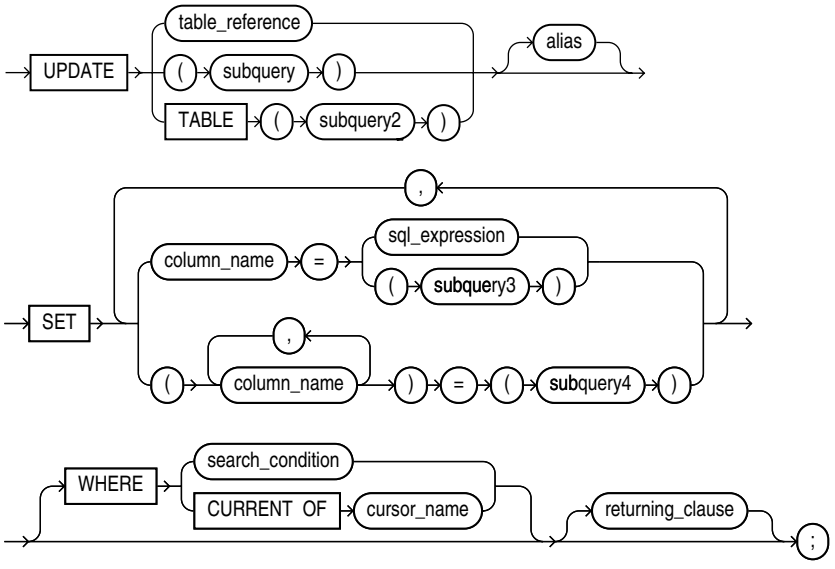
定数と変数、%ROWTYPE 属性

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、『Oracle9*i* SQL リファレンス』を参照してください。

構文

update_statement



キーワードとパラメータの説明

alias

参照される表またはビューの別名（通常は短縮名）で、WHERE 句の中で頻繁に使用されます。

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にしてください。column_name リストでは同じ列名を繰り返して使用できません。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。

returning_clause

この句を使用すると、更新された行から値を戻せるため、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、変数かホスト配列（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、13-55 ページの「DELETE 文」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、参照は現在行の列が対象になります。古い列の値は、等号の右辺で使用されます。

次の例では、すべての従業員の給与を 10% ずつ増加させています。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 つの列を戻す必要があります。

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。副問合せは、リストされている列すべてを含む 1 つの行のみを戻す必要があります。

副問合せ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列 item_id に代入され、item_price に格納されている値が列 price 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);
```

sql_expression

任意の有効な SQL の式です。詳細は、『Oracle9*i* SQL リファレンス』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 句は使用できません。13-163 ページの「SELECT INTO 文」を参照してください。

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る必要があり、ユーザーが UPDATE 権限を持つ必要があります。table_reference の構文は、13-55 ページの「DELETE 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 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。

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。

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のみが更新されます。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。

使用上の注意

UPDATE WHERE CURRENT OF 文は、オープンされているカーソルからのフェッチ（カーソル FOR ループで実行される暗黙的なフェッチを含む）の後で使用できます（ただしそのためには、対応付けられた問合せが FOR UPDATE である必要があります）。この文は現在行、つまり直前にフェッチ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 を使用すると、UPDATE 文の実行に関する有用な情報にアクセスできます。

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アナリスト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

A

PL/SQL のサンプル・プログラム

この付録では、独自のプログラムを作成する上で参考になる 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。

この付録の項目は、次のとおりです。

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・ディレクトリの場所については、使用し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	「PL/SQL の主な特長」1-2 ページ
例 2	「条件制御」1-8 ページ
例 3	「反復制御」1-10 ページ
例 4	「エイリアシングの使用」2-15 ページ
例 7	「カーソル FOR ループの使用」6-13 ページ
例 8	「カーソル FOR ループへのパラメータの受渡し」6-15 ページ
例 5	「カーソル属性の例」6-36 ページ
例 6	「カーソル属性の例」6-36 ページ
例 11	「例」13-16 ページ
例 12	「例」13-42 ページ
例 13	「例」13-42 ページ
例 14	「例」13-42 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-5 ページ
サンプル 3	「サンプル 3. 有効範囲」A-7 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-9 ページ
サンプル 5	「サンプル 5. 埋込み PL/SQL」A-13 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-17 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成し、表にデータをロードする必要があります。そのためには、PL/SQL に付属する 2 つの SQL*Plus スクリプト、

exampbld と examplod 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、次のコマンドを発行してください。

```
SQL> START exampbld
...
SQL> START examplod
```

サンプル 1. FOR ループ

次の例では、単純な FOR ループを使用し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

使用しません。

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;

NUM_COL1 NUM_COL2  CHAR_COL
-------- --------  ---------
       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、カーソルを使用して表 emp から給与の最も高い 5 人の従業員を選択します。

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
```

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 VARCHAR2(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;

NUM_COL1 NUM_COL2  CHAR_COL
-------- --------  --------
    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、同様に 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。temp 表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を確認できます。

入力表

使用しません。

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, 'in 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;

NUM_COL1 NUM_COL2  CHAR_COL
-------- --------  -------------
    1000         1  in OUTER loop
       1         2  inner loop
       2         3  inner loop
       3         4  inner loop
       4         5  inner loop
    2000         6  in OUTER loop
       1         7  inner loop
       2         8  inner loop
       3         9  inner loop
       4        10  inner loop
    3000        11  in OUTER loop
       1        12  inner loop
       2        13  inner loop
       3        14  inner loop
       4        15  inner loop
    4000        16  in 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用する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挿入する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成されます。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.                                   */
   /*-----------------------------------------*/
   ELSIF 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
         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 言語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            CONSTANT NUMBER := 500;
   BEGIN
      SELECT bal INTO old_bal FROM accounts
         WHERE account_id = :acct;
         -- 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

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。一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、フェッチ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];
   EXEC SQL VAR emp_name is STRING (21);
   EXEC SQL VAR job is STRING (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");
   printf("%-10.10s%-10.10s%s\n", "Employee", "Job", "Salary");
   printf("%-10.10s%-10.10s%s\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
```

B

CHAR と VARCHAR2 の意味の比較

この付録では、ベース型 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、文字値の代入、比較、挿入、更新、選択、またはフェッチに関係してきます。

この付録の項目は、次のとおりです。

文字値の代入

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 は代入を中止して事前定義済みの例外 VALUE_ERROR を呼び出します。PL/SQL が値を切り捨てたり、後続する空白を切り捨てることはありません。たとえば、次のような宣言があるとします。

```
acronym CHAR(4);
```

次のような代入を試みると 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、情報は失われません。VARCHAR2 型変数の宣言された長さよりも文字値が長い場合、PL/SQL は代入を中止して VALUE_ERROR を呼び出します。PL/SQL が値を切り捨てたり、後続する空白を切り捨てることはありません。

文字値の比較

関係演算子を使用すると、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・キャラクタ・セットの照合順番に基づいて行われます。文字値の比較では、照合順番で後の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 つの文字値は同じ長さにしてください。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、空白埋め方法が使用されます。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、非空白埋め方法が使用されます。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用します。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用されます。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用されます。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って空白埋め比較方法が使用されます。

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。文字の値が定義されている列幅より長い場合、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ため、情報は失われません。文字値の長さが列の幅の定義より長ければ、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、後続する空白を格納しないようにできます。次に例を示します。

```
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。文字値が変数の宣言された長さより長い場合、PL/SQL は代入を中止して例外 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ため、情報は失われません。

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。VARCHAR2 型変数の宣言された長さよりも文字値が長い場合、PL/SQL は代入を中止して 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。

C

PL/SQL ラップ・ユーティリティ

この付録では、ラップ・ユーティリティの実行方法について説明します。ラップ・ユーティリティは、PL/SQL ソース・コードを暗号化するスタンドアロン・プログラミング・ユーティリティです。ラップ・ユーティリティを使用すると、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。

この付録の項目は、次のとおりです。

PL/SQL プロシージャのラッピングの利点

ラップ・ユーティリティは、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ありません。
- 動的ロードーユーザーは、新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ありません。
- 動的バインドー外部参照はロード時に解決されます。
- 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ます。
- 正常なインポートとエクスポートーインポート / 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ます。

ラップ・ユーティリティの制限

文字列リテラル、数値リテラル、および変数、表、列の名前は、プレーン・テキストでラップされたファイル内に残ります。プロシージャをラップすると、アルゴリズムが隠蔽され、リバース・エンジニアリングの防止に役立ちますが、公表を望まないパスワードや表名の隠蔽にはなりません。

最近の一部の SQL 構文は、ラップ・ユーティリティによるデフォルトでのサポートがありません。すべての SQL 構文に対してサポートを有効にするには、オプションの `edebug=wrap_new_sql`（ダッシュなし）を指定します。このオプションはデフォルトではありません。これは、このオプションによって、ラップされたファイル内のすべての SQL 文がプレーン・テキストで表示されるためです。

ラップ・ユーティリティの実行

ラップ・ユーティリティを実行するには、次の構文を使用して、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用するため、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 つのみです。

```
iname=input_file
```

ここで、input_file は、ラップ・ユーティリティの入力ファイルの名前です。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
```

output_file は、ラップ・ユーティリティの出力ファイルの名前です。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 オプションを使用して、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
```

ラップ・ユーティリティの入力ファイルと出力ファイル

入力ファイルでは、SQL 文を任意に組み合せることができます。ただし、ラップ・ユーティリティは、サブプログラム、パッケージまたはオブジェクト型を定義する次の CREATE 文しか暗号化しません。

```
CREATE [OR REPLACE] FUNCTION function_name
CREATE [OR REPLACE] PROCEDURE procedure_name
CREATE [OR REPLACE] PACKAGE package_name
CREATE [OR REPLACE] PACKAGE BODY package_name
CREATE [OR REPLACE] TYPE type_name ... OBJECT
CREATE [OR REPLACE] TYPE BODY type_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、サブプログラム、パッケージまたはオブジェクト型内にないかぎり、削除されます。

暗号化されると、サブプログラム、パッケージまたはオブジェクト型は次の形式になります。

```
<header> wrapped <body>
```

header は予約語 CREATE で始まり、サブプログラム、パッケージまたはオブジェクト型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、サブプログラム、パッケージまたはオブジェクト型がラップ・ユーティリティによって暗号化されたことを 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、ラップ・ユーティリティは、次のソース・コードを

```
CREATE PACKAGE
-- Author: J. Hollings
-- Date:   10/15/99
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:   10/15/99
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。

ラップ・ユーティリティでのエラー処理

入力ファイルに構文エラーが含まれている場合、ラップ・ユーティリティはそのエラーを検出し、レポートします。ただし、ラップ・ユーティリティ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、ラップ・ユーティリティは次のエラー「表またはビューが存在しません。(*table or view does not exist*)」をレポートしません。

```
CREATE PROCEDURE raise_salary (emp_id INTEGER, amount NUMBER) AS
BEGIN
   UPDATE amp  -- 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、ラップ・ユーティリティ出力ファイル（.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。

バージョン間の互換性

ラップ・ユーティリティは Oracle と上位互換性を持ちます。そのため、たとえば V8.1.5 のラップ・ユーティリティで処理されたファイルを、V8.1.6 の Oracle データベースにロードできます。ただし、ラップ・ユーティリティは Oracle と下位互換性は持ちません。そのため、たとえば V8.1.6 のラップ・ユーティリティで処理されたファイルは、V8.1.5 の Oracle データベースにロードできません。

指針

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、本体のみをラップします。こうすると、他の開発者は、パッケージまたは型を使用するのに必要な情報を見ることができますが、その実装は見えません。

すべての暗号化されたファイルと同じく、ラップされたファイルは編集できません。ラップされたファイルを変更するには、基となるソース・コードを変更および再ラップする必要があります。そのため、サブプログラム、パッケージまたはオブジェクト型は、エンド・ユーザーに出荷する用意ができるまでラップしないでください。

D

PL/SQL の名前解決

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。

この付録の項目は、次のとおりです。

名前解決

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、実際の値などと関連付けます。このプロセスはバインドと呼ばれます。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かぎり、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。このプロセスは名前解決と呼ばれます。PL/SQL では、名前はすべて同じ名前空間にあると見なされます。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る可能性があります。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 の定義が隠されています。

```
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;        -- not allowed; 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で、データ型 Client を参照できます。

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

名前解決の際、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、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。次に、有効な参照の例をいくつか示します。

```
CREATE PACKAGE pkg1 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 pkg1;

CREATE PACKAGE BODY pkg1 AS
   FUNCTION f1 (p1 NUMBER) RETURN t1 IS
      n NUMBER;
   BEGIN
      n := m;             -- (1) unqualified name
      n := pkg1.m;        -- (2) dot-separated chain of identifiers
                          --     (package name used as scope
                          --     qualifier followed by variable name)
      n := pkg1.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 := pkg1.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.pkg1.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
      n := scott.pkg1.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 f1;

   FUNCTION f2 (q1 NUMBER) RETURN t2 IS
   BEGIN
      ...
   END f2;
END pkg1;
```

名前解決のアルゴリズム

名前解決アルゴリズムについて説明します。

名前解決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、解決できます。

上に示した例では、(3) `pkg1.f1.p1` に対するベースは `pkg1`、(4) `scott.pkg1.f1.n` に対するベースは `scott.pkg1`、(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・セレクタで、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（not declared）」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。これが失敗した場合、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 スコープ内で、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す。

```
schema_name.table_name.column_name
```

次に、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 スコープ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 スコープ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 スコープ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します。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ます。
- 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します。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ます。
- 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、次に示すいずれかの項目の検索を行います。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ます。
  - 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ます。
  - 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ます。
- 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します。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ます。

列名に使用するベースが検出された場合、コンパイラは、ベースのコンポーネントなどを検索して参照全体を解決しようとします（検索対象は列名の型により異なります）。

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 スコープでベースを検索するためのルールを示します。（行の式は、単独で使用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 文でしか使用できません。）

- 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します。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ます。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します。
- 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します。

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 スコープに存在しない場合

- SQL スコープに存在するが、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場合

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・ユニットに対してローカルなすべての PL/SQL スコープ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 のベースを検索します。これは、連鎖した識別子の最初の識別子と名前の一致するスキーマ・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ます。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、シノニムまたはスキーマ・レベルのデータ型のいずれかになります。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ます。
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します。この場合、連鎖した識別子の最初の識別子と名前の一致するパブリック ・シノニムが検索されます。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ます。
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します。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の一致するスキーマにより所有されるものが検索されます。
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ます。基本オブジェクトが参照可能ではない場合、コンパイラは「未宣言（not declared）」エラーを生成します。「不十分な権限（insufficient privileges）」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（insufficient privileges）」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です。
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（not declared）」エラーを生成します。
6. コンパイラがベースを検出した場合、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します。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します。

取得の理解

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「取得する」と呼びます。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。取得には、inner、same-scope および outer の 3 種類があります。内部および同一有効範囲の取得は SQL スコープにのみ適用されます。

内部取得

内部取得が発生するのは、一度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取得され、参照を完全に解決することができなかったため、エラーを発生する場合

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取得が発生している可能性があります。次の例では、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。

```
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;
```

上の例で、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 への参照は取得されます。

コレクションやオブジェクト型を使用することにより、さらに多くの内部取得状況を実現することができます。次の例では、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。

```
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
```

上の例で、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、内部取得が発生します。内部取得を防止するには、D-9 ページの「取得の防止」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、この問合せは、次のように書き換わります。

```
SELECT * FROM s.tab1 p1
   WHERE EXISTS (SELECT * FROM s.tab2 p2 WHERE p2.x = p1.tab2.a);
```

同一有効範囲の取得

SQL スコープで、同一有効範囲の取得が発生するのは、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を発生します。したがって、col2 への参照はエラーにより取得されます。

外部取得

外部取得が発生するのは、過去に内部有効範囲内の実体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。SQL と PL/SQL は、外部取得を防止する設計になっています。

取得の防止

次のルールを遵守することにより、DML 文における内部取得を防止できます。

- DML 文内の各表に対して別名を指定します。
- DML 文の全体を通じて、表の別名を一意に保ちます。
- 問合せ内で使用されているスキーマ名と一致する表の別名の使用を避けます。
- 列の参照を表の別名で修飾します。

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取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、より多くの内部取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、次のルールが名前解決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ります。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ります。次の例に示すとおり、属性またはメソッドへの列修飾された参照は、その参照に接頭辞として表名が使用されている場合は使用できません。

```
CREATE TYPE t1 AS OBJECT (x NUMBER);
CREATE TABLE tb1 (col t1);
SELECT col.x FROM tb1;                    -- not allowed
SELECT tb1.col.x FROM tb1;                -- not allowed
SELECT scott.tb1.col.x FROM scott.tb1;    -- not allowed
SELECT t.col.x FROM tb1 t;
UPDATE tb1 SET col.x = 10;                -- not allowed
UPDATE scott.tb1 SET scott.tb1.col.x=10;  -- not allowed
UPDATE tb1 t set t.col.x = 1;
DELETE FROM tb1 WHERE tb1.col.x = 10;     -- not allowed
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ります。行の式を REF および VALUE に受け渡したり、UPDATE 文の SET 句に行の式を使用できます。次に例を示します。

```
CREATE TYPE t1 AS OBJECT (x number);
CREATE TABLE ot1 OF t1;                       -- object table
SELECT REF(ot1) FROM ot1;                     -- not allowed
SELECT REF(o) FROM ot1 o;
SELECT VALUE(ot1) FROM ot1;                   -- not allowed
SELECT VALUE(o) FROM ot1 o;
DELETE FROM ot1 WHERE VALUE(ot1) = (t1(10));  -- not allowed
DELETE FROM ot1 o WHERE VALUE(o) = (t1(10));
UPDATE ot1 SET ot1 = ...                      -- not allowed
UPDATE ot1 o SET o = ....
```

  オブジェクト表に挿入するための次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した場合、次の例に示すように、空のパラメータ・リストはオプションです。

```
CREATE FUNCTION func1 RETURN NUMBER AS
BEGIN
   RETURN 10;
END;

CREATE PACKAGE pkg2 AS
   FUNCTION func1 RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES(func1,WNDS,RNDS,WNPS,RNPS);
END pkg2;

CREATE PACKAGE BODY pkg2 AS
   FUNCTION func1 RETURN NUMBER IS
   BEGIN
      RETURN 20;
   END;
END pkg2;

SELECT func1 FROM dual;
SELECT func1() FROM dual;

SELECT pkg2.func1 FROM dual;
SELECT pkg2.func1() FROM dual;

DECLARE
   x NUMBER;
BEGIN
   x := func1;
   x := func1();
   SELECT func1 INTO x FROM dual;
   SELECT func1() INTO x FROM dual;
   SELECT pkg2.func1 INTO x FROM dual;
   SELECT pkg2.func1() INTO x FROM dual;
END;
```

パラメータを持たないメソッドをコールした場合、PL/SQL スコープでは、空のパラメータ・リストはオプションです。ただし、SQL スコープでは、空のパラメータ・リストは必須です。次に例を示します。

```
CREATE 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;   -- not allowed
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;    -- not allowed
   SELECT x.col1.f() INTO n FROM tab1 x;
   SELECT y.f INTO n FROM dual;           -- not allowed
   SELECT y.f() INTO n FROM dual;
END;
```

SQL と PL/SQL の名前解決の比較

SQL と PL/SQL の名前解決ルールはよく似ています。いくつかの小さな違いはありますが、取得回避規則に従うかぎり、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、より許容性が高くなっています。つまり、そのほとんどが状況依存な 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、有効なものと認識されます。

E

PL/SQL のプログラム上の制限

PL/SQL は主として高速トランザクション処理用に設計されています。そのような設計では、この付録で説明するように、いくつかのプログラム上の制限が課されます。

PL/SQL はプログラミング言語 Ada をベースにしています。したがって PL/SQL では、ツリー構造の中間言語、DIANA（Descriptive Intermediate Attributed Notation for Ada）を使用しています。この中間言語は、インタフェース定義言語（IDL）と呼ばれるメタ表記を使用して定義されます。コンパイラなどのツール内部の通信は DIANA により提供されます。

PL/SQL のソース・コードはコンパイル時に機械可読の m コードに変換されます。1 つのプロシージャまたはパッケージに対応する DIANA および m コードの両方がデータベースに格納されます。実行時に DIANA と m コードは共有メモリー・プール内にロードされます。DIANA は依存プロシージャのコンパイルに使用され、m コードはそのまま実行されます。

共有メモリー・プールの場合、パッケージ仕様部、オブジェクト型仕様部、スタンドアロン・サブプログラムまたは無名ブロックは、2**26 個の DIANA ノードに制限されています（ノードは、識別子、キーワード、演算子などのトークンに対応します）。PL/SQL コンパイラによる制限を超えないかぎり、6,000,000 行以下のコードが許されます。PL/SQL コンパイラによる一部の制限を表 E-1 に示します。

表 E-1 PL/SQL コンパイラの制限

項目	制限
プログラム・ユニットに渡されるバインド変数	32K
プログラム・ユニット内の例外ハンドラ	64K
レコード内のフィールド	64K
ブロック・ネストのレベル	255
レコード・ネストのレベル	32
副問合せネストのレベル	254
ラベル・ネストのレベル	98
`BINARY_INTEGER` 値の絶対値	2G
`PLS_INTEGER` 値の絶対値	2G
プログラム・ユニットから参照されるオブジェクト	64K

表 E-1 PL/SQL コンパイラの制限（続き）

項目	制限
明示カーソルに渡されるパラメータ	64K
ファンクションまたはプロシージャに渡されるパラメータ	64K
FLOAT 値（2 進数）の精度	126
NUMBER 値（10 進数）の精度	38
REAL 値（2 進数）の精度	63
識別子のサイズ（文字）	30
文字列リテラルのサイズ（バイト）	32K
CHAR 値（バイト）のサイズ	32K
LONG 値（バイト）のサイズ	32K-7
LONG RAW 値（バイト）のサイズ	32K-7
RAW 値（バイト）のサイズ	32K
VARCHAR2 値（バイト）のサイズ	32K
NCHAR 値（バイト）のサイズ	32K
NVARCHAR2 値（バイト）のサイズ	32K
BFILE 値（バイト）のサイズ	4G
BLOB 値（バイト）のサイズ	4G
CLOB 値（バイト）のサイズ	4G
NCLOB 値（バイト）のサイズ	4G

プログラム・ユニットに必要なメモリー量を見積るときには、データ・ディクショナリ・ビュー user_object_size を使用して問い合せることができます。parsed_size という列に、フラット化された DIANA のバイト・サイズが戻ります。次の例では、パッケージ仕様部の行に表示されたパッケージのサイズの解析結果が得られます。

```
CREATE PACKAGE pkg1 AS
   PROCEDURE proc1;
END pkg1;
/

CREATE PACKAGE BODY pkg1 AS
   PROCEDURE proc1 IS
   BEGIN
      NULL;
   END;
END pkg1;
/
```

```
SQL> SELECT * FROM user_object_size WHERE name = 'PKG1';

NAME    TYPE         SOURCE_SIZE  PARSED_SIZE  CODE_SIZE  ERROR_SIZE
------------------------------------------------------------------
PKG1    PACKAGE               46          165        119           0
PKG1    PACKAGE BODY          82            0        139           0
```

しかし、サイズの解析結果から DIANA ノードの数を見積ることはできません。解析後のサイズが同じプログラム・ユニットが 2 つあり、一方に必要な DIANA ノードの数が 1500 に対して、他方に必要なのが 2000 である（2 番目のユニットに含まれる SQL 文が複雑であるなどの理由で）場合もあるためです。

PL/SQL ブロック、サブプログラム、パッケージまたはオブジェクト型のいずれかがサイズの限度を超えると、「プログラムが大きすぎます（as program too large）」などのエラーとなります。多くの場合、この問題はパッケージ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。無名ブロックの場合、最良の方法はサブプログラムのグループとして再定義し、データベースに格納できるようにすることです。

F

PL/SQL の予約語のリスト

この付録では、PL/SQL で使用する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用しないでください。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用しないでください。

ALL*
ALTER*
AND*
ANY*
ARRAY
AS*
ASC*
AT
AUTHID
AVG
BEGIN
BETWEEN*
BINARY_INTEGER
BODY
BOOLEAN
BULK
BY*
CASE
CHAR*
CHAR_BASE
CHECK*
CLOSE
CLUSTER*
COALESCE
COLLECT
COMMENT*
COMMIT
COMPRESS*
CONNECT*
CONSTANT
CREATE*
CURRENT*
CURRVAL
CURSOR
DATE*
DAY
DECLARE
DECIMAL*
DEFAULT*
DELETE*
DESC*
DISTINCT*
DO
DROP*
ELSE*
ELSIF
END
EXCEPTION
EXCLUSIVE*
EXECUTE
EXISTS*
EXIT
EXTENDS
EXTRACT
FALSE
FETCH
FLOAT*
FOR*
FORALL
FROM*
FUNCTION
GOTO
GROUP*
HAVING*
HEAP
HOUR
IF
IMMEDIATE*
IN*
INDEX*
INDICATOR
INSERT*
INTEGER*
INTERFACE
INTERSECT*
INTERVAL
INTO*
IS*
ISOLATION
JAVA
LEVEL*
LIKE*
LIMITED
LOCK*
LONG*
LOOP
MAX
MIN
MINUS*
MINUTE
MLSLABEL*
MOD
MODE*
MONTH
NATURAL
NATURALN
NEW
NEXTVAL
NOCOPY
NOT*
NOWAIT*
NULL*
NULLIF
NUMBER*
NUMBER_BASE
OCIROWID
OF*
ON*
OPAQUE
OPEN
OPERATOR
OPTION*
OR*
ORDER*
ORGANIZATION
OTHERS
OUT
PACKAGE
PARTITION
PCTFREE*
PLS_INTEGER
POSITIVE
POSITIVEN
PRAGMA
PRIOR*
PRIVATE
PROCEDURE
PUBLIC*
RAISE
RANGE
RAW*
REAL
RECORD
REF
RELEASE
RETURN
REVERSE
ROLLBACK
ROW*
ROWID*
ROWNUM*
ROWTYPE
SAVEPOINT
SECOND
SELECT*
SEPARATE
SET*
SHARE*
SMALLINT*
SPACE
SQL
SQLCODE
SQLERRM
START*
STDDEV
SUBTYPE
SUCCESSFUL*
SUM
SYNONYM*
SYSDATE*
TABLE*
THEN*
TIME
TIMESTAMP
TIMEZONE_REGION
TIMEZONE_ABBR
TIMEZONE_MINUTE
TIMEZONE_HOUR
TO*
TRIGGER*
TRUE
TYPE
UID*
UNION*
UNIQUE*
UPDATE*
USE
USER*
VALIDATE*
VALUES*

```
VARCHAR*
VARCHAR2*
VARIANCE
VIEW*
WHEN
WHENEVER*
WHERE*
WHILE
WITH*
WORK
WRITE
YEAR
ZONE
```

索引

記号

数字

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

て

と

な

に

ね

は

ひ

ふ

ゆ

よ

ら

り

る

れ

ろ

わ