



EDB PostgresAdvanced Server

リリース **13**

Database Compatibility Stored Procedural Language Guide

2020 年 12 月 31 日

目次

第 1 章	Basic SPL Elements	1
1.1	Character Set	1
1.2	Case Sensitivity	2
1.3	Identifiers	2
1.4	Qualifiers	2
1.5	Constants	3
1.6	User-Defined PL/SQL Subtypes	4
第 2 章	SPL Programs	7
2.1	SPL Block Structure	7
2.2	Anonymous Blocks	9
2.3	Procedures Overview	11
2.3.1	Creating a Procedure	11
2.3.2	Calling a Procedure	15
2.3.3	Deleting a Procedure	16
2.4	Functions Overview	18
2.4.1	Creating a Function	18
2.4.2	Calling a Function	22
2.4.3	Deleting a Function	23
2.5	Procedure and Function Parameters	24
2.5.1	Positional vs. Named Parameter Notation	26
2.5.2	Parameter Modes	28
2.5.3	Using Default Values in Parameters	29
2.6	Subprograms – Subprocedures and Subfunctions	31
2.6.1	Creating a Subprocedure	33
2.6.2	Creating a Subfunction	35
2.6.3	Block Relationships	37
2.6.4	Invoking Subprograms	39
2.6.5	Using Forward Declarations	47
2.6.6	Overloading Subprograms	48
2.6.7	Accessing Subprogram Variables	52
2.7	Compilation Errors in Procedures and Functions	62

2.8	Program Security	64
2.8.1	EXECUTE Privilege	64
2.8.2	Database Object Name Resolution	65
2.8.3	Database Object Privileges	66
2.8.4	Definer' s vs. Invokers Rights	66
2.8.5	Security Example	67
第 3 章	Variable Declarations	75
3.1	Declaring a Variable	75
3.2	User-Defined Record Types and Record Variables	76
第 4 章	Basic Statements	79
4.1	Assignment	79
4.2	DELETE	80
4.3	INSERT	81
4.4	NULL	83
4.5	Using the RETURNING INTO Clause	83
4.6	SELECT INTO	86
4.7	UPDATE	88
4.8	Obtaining the Result Status	90
第 5 章	Control Structures	91
5.1	IF Statement	91
5.1.1	IF-THEN	91
5.1.2	IF-THEN-ELSE	92
5.1.3	IF-THEN-ELSE IF	93
5.1.4	IF-THEN-ELSIF-ELSE	95
5.2	RETURN Statement	97
5.3	GOTO Statement	97
5.4	CASE Expression	100
5.4.1	Selector CASE Expression	100
5.4.2	Searched CASE Expression	101
5.5	CASE Statement	103
5.5.1	Selector CASE Statement	103
5.5.2	Searched CASE Statement	105
5.6	Loops	107
5.6.1	LOOP	107
5.6.2	EXIT	107
5.6.3	CONTINUE	108
5.6.4	WHILE	109
5.6.5	FOR (integer variant)	109
5.7	Exception Handling	111

5.8	User-defined Exceptions	113
5.9	PRAGMA EXCEPTION_INIT	115
5.10	RAISE_APPLICATION_ERROR	117
第 6 章	Transaction Control	119
6.1	COMMIT	120
6.2	ROLLBACK	121
6.3	PRAGMA AUTONOMOUS_TRANSACTION	125
第 7 章	Dynamic SQL	135
第 8 章	Static Cursors	139
8.1	Declaring a Cursor	139
8.2	Opening a Cursor	140
8.3	Fetching Rows From a Cursor	140
8.4	Closing a Cursor	142
8.5	Cursor Attributes	143
8.5.1	Summary of Cursor States and Attributes	143
8.6	Cursor FOR Loop	143
8.7	Parameterized Cursors	145
第 9 章	REF CURSORS and Cursor Variables	147
9.1	REF CURSOR Overview	147
9.2	Declaring a Cursor Variable	147
9.2.1	Declaring a SYS_REFCURSOR Cursor Variable	148
9.2.2	Declaring a User Defined REF CURSOR Type Variable	148
9.3	Opening a Cursor Variable	148
9.4	Fetching Rows From a Cursor Variable	149
9.5	Closing a Cursor Variable	150
9.6	Usage Restrictions	151
9.7	Examples	152
9.7.1	Returning a REF CURSOR From a Function	152
9.7.2	Modularizing Cursor Operations	153
9.8	Dynamic Queries With REF CURSORS	157
第 10 章	Collections	161
10.1	Associative Arrays	162
10.2	Nested Tables	166
10.3	Varrays	171
第 11 章	Collection Methods	175
11.1	COUNT	175
11.2	DELETE	176

11.3	EXISTS	177
11.4	EXTEND	178
11.5	FIRST	180
11.6	LAST	181
11.7	LIMIT	181
11.8	NEXT	182
11.9	PRIOR	182
11.10	TRIM	183
第 12 章	Working with Collections	185
12.1	TABLE()	185
12.2	Using the MULTISSET UNION Operator	186
12.3	Using the FORALL Statement	188
12.4	Using the BULK COLLECT Clause	190
12.4.1	SELECT BULK COLLECT	190
12.4.2	FETCH BULK COLLECT	192
12.4.3	EXECUTE IMMEDIATE BULK COLLECT	194
12.4.4	RETURNING BULK COLLECT	194
12.5	Errors and Messages	198
第 13 章	Triggers	199
13.1	Overview	199
13.2	Types of Triggers	200
13.3	Creating Triggers	201
13.4	Trigger Variables	205
13.5	Transactions and Exceptions	206
13.6	Compound Triggers	206
13.7	Trigger Examples	209
13.7.1	Before Statement-Level Trigger	209
13.7.2	After Statement-Level Trigger	209
13.7.3	Before Row-Level Trigger	211
13.7.4	After Row-Level Trigger	211
13.7.5	INSTEAD OF Trigger	214
13.7.6	Compound Triggers	215
第 14 章	Packages	221
第 15 章	Object Types and Objects	223
15.1	Basic Object Concepts	223
15.1.1	Attributes	224
15.1.2	Methods	224
15.1.3	Overloading Methods	224

15.2	Object Type Components	225
15.2.1	Object Type Specification Syntax	225
15.2.2	Object Type Body Syntax	228
15.3	Creating Object Types	231
15.3.1	Member Methods	231
15.3.2	Static Methods	232
15.3.3	Constructor Methods	234
15.4	Creating Object Instances	236
15.5	Referencing an Object	237
15.6	Dropping an Object Type	239
第 16 章	Conclusion	241
索引		243

第 1 章

Basic SPL Elements

AdvancedServer のストアードプロシージャ言語（SPL）は、以下を提供する AdvancedServer のカスタムプロシージャ、関数、トリガー、およびパッケージを作成するための生産性の高いプロシージャプログラミング言語です。

- SQL 言語を補完する完全な手続き型プログラミング機能
- AdvancedServer データベースのストアードプロシージャ、関数、トリガー、およびパッケージを作成するための単一の共通言語
- シームレスな開発およびテスト環境
- 再利用可能なコードの使用
- 使いやすさ

このガイドでは、SPL プログラムの構成の概要と、それを使用してプロシージャまたは関数を作成する方法を説明する前に、SPL プログラムの基本要素について説明します。SPL 言語と組み合わせて使用される可能性のあるその他の互換性機能（トリガーや組み込み関数など）に関するガイドは、EDBWeb サイト<<https://www.enterprisedb.com/edb-docs>>で入手できます。

1.1 Character Set

SPL プログラムは、次の文字セットを使用して作成されます。

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

SPL 言語を構成する識別子、式、ステートメント、制御構造などは、これらの文字を使用して記述されます。

注釈: SPL プログラムで操作できるデータは、データベースエンコーディングでサポートされている文字セットによって決まります。

1.2 Case Sensitivity

SPL プログラムで使用するキーワードとユーザー定義識別子は、大文字と小文字を区別しません。たとえば、`DBMS_OUTPUT.PUT_LINE('Hello World');` という文は、`dbms_output.put_line('Hello World');` または `Dbms_Output.Put_Line('Hello World');` または `DBMS_output.Put_line('Hello World');` と同じものを意味すると解釈されます。

ただし、文字定数と文字列定数は大文字と小文字が区別され、AdvancedServer データベースから取得したデータや他の外部ソースから取得したデータも同様です。ステートメント `DBMS_OUTPUT.PUT_LINE('Hello World!');` は、次の出力を生成します。

```
Hello World!
```

ただし、ステートメント `DBMS_OUTPUT.PUT_LINE('HELLO WORLD!');` は出力を生成します。

```
HELLO WORLD!
```

1.3 Identifiers

識別子は、変数、カーソル、ラベル、プログラム、パラメーターなど、SPL プログラムのさまざまな要素を識別するために使用されるユーザー定義の名前です。有効な識別子の構文規則は、SQL 言語の識別子と同じです。

識別子は、SPL キーワードまたは SQL 言語のキーワードと同じであってはなりません。有効な識別子の例を示します。

```
x
last__name
a_$_Sign
Many$$$$$$signs_____
THIS_IS_AN_EXTREMELY_LONG_NAME
A1
```

1.4 Qualifiers

*qualifier**は、資格のオブジェクトであるエンティティの所有者またはコンテキストを指定する名前です。修飾オブジェクトは、修飾子名とそれに続く空白を含まないドット、その後に空白を含まないで修飾されるオブジェクトの

名前として指定されます。この構文は*ドット表記 と呼ばれます。

修飾されたオブジェクトの構文は次のとおりです。

```
<qualifier.> [ <qualifier.> ]... <object>
```

qualifier はオブジェクトの所有者の名前です。 object は qualifier に属するエンティティの名前です。先行する修飾子が後続の修飾子とオブジェクトによって識別されるエンティティを所有している場合、資格のチェーンを持つことができます。

ほとんどすべての識別子を修飾できます。識別子が修飾されるものは、識別子が表すものとその使用のコンテキストに依存します。

資格のいくつかの例を次に示します。

- 所属するスキーマによって修飾されたプロシージャ名と関数名-例:schema_name.procedure_name (..)
- 所属するスキーマによって修飾されたトリガー名-例:schema_name.trigger_name
- 所属するテーブルで修飾された列名-例:emp.empno
- 所属するスキーマによって修飾されたテーブル名-public.emp など
- テーブルとスキーマで修飾された列名-例:public.emp.empno

一般的な規則として、SPL ステートメントの構文で名前が表示される場合は常に、その修飾名も使用できます。通常、修飾名は、名前にあいまいさが関連付けられている場合にのみ使用されます。たとえば、2つの異なるスキーマに属する同じ名前の2つのプロシージャがプログラム内から呼び出される場合、または同じプログラム内のテーブル列と SPL 変数に同じ名前が使用される場合です。

可能な限り、修飾名を使用しないでください。この章では、命名の競合を回避するために、次の規則が採用されています。

- SPL プログラムの宣言セクションで宣言されたすべての変数には、 v_ という接頭辞が付きます。
例:v_empno
- プロシージャまたは関数の定義で宣言されたすべての仮パラメータには、 p_ という接頭辞が付きます。
例:p_empno
- 列名と表名には、特別な接頭辞規則はありません。たとえば、表 emp の列 empno

1.5 Constants

定数*または*リテラル は、SPL プログラムでさまざまなタイプ（数値、文字列、日付など）の値を表すために使用できる固定値です。定数には次のタイプがあります。

- 数値（整数および実数）
- 文字と文字列
- 日時

1.6 User-Defined PL/SQL Subtypes

AdvancedServer は、ユーザー定義の PL/SQL サブタイプおよび（サブタイプ）エイリアスをサポートしています。サブタイプは、そのタイプの列に格納できる値を制限する制約のオプションセットを持つデータタイプです。サブタイプのベースとなる型に適用されるルールは引き続き適用されますが、追加の制約を使用して、型に格納されている値の精度またはスケールを制限できます。

PL 関数、プロシージャ、匿名ブロックまたはパッケージの宣言でサブタイプを定義できます。構文は次のとおりです。

```
SUBTYPE <subtype_name> IS <type_name>[( <constraint>)] [NOT NULL]
```

constraint は次のとおりです。

```
{<precision> [, <scale>]} | <length>
```

どこで:

subtype_name

subtype_name はサブタイプの名前を指定します。

type_name

type_name は、サブタイプが基づく元のタイプの名前を指定します。type_name は:

- AdvancedServer がサポートするタイプの名前。
- 任意の複合タイプの名前。
- %TYPE 演算子によって固定された列。
- 別のサブタイプの名前。

constraint 句を含めて、精度またはスケールをサポートする型の制限を定義します。

precision

precision は、サブタイプの値で許可される合計桁数を指定します。

scale

scale は、サブタイプの値で許可される小数桁数を指定します。

length

length は、CHARACTER、VARCHAR または TEXT の基本タイプの値で許可される全長を指定します。

NOT NULL 句を含めて、指定されたサブタイプの列に NULL 値が格納されないように指定します。

列に基づくサブタイプは列サイズの制約を継承しますが、サブタイプは NOT NULL または CHECK の制約を継承しません。

Unconstrained Subtypes

制約のないサブタイプを作成するには、SUBTYPE コマンドを使用して、新しいサブタイプ名とサブタイプのベースとなるタイプの名前を指定します。たとえば、次のコマンドは、タイプ CHAR: のすべての属性を持つ address という名前のサブタイプを作成します

```
SUBTYPE address IS CHAR;
```

別のサブタイプのサブタイプであるサブタイプ（制約付きまたは制約なし）を作成することもできます。

```
SUBTYPE cust_address IS address NOT NULL;
```

このコマンドは、address サブタイプのすべての属性を共有する cust_address という名前のサブタイプを作成します。cust_address の値が NULL ではないことを指定するには、NOT NULL 句を含めます。

Constrained Subtypes

サブタイプの最大長を定義するために、文字タイプに基づくサブタイプを作成するときに length 値を含めます。例:

```
SUBTYPE acct_name IS VARCHAR (15);
```

この例では、acct_name という名前のサブタイプを作成します。これは VARCHAR データタイプに基づいていますが、長さは 15 文字に制限されています。

数値ベースタイプを制約する場合は、precision（サブタイプの値の最大桁数を指定する）およびオプションで scale（小数点の右側の桁数を指定する）の値を含めます。例:

```
SUBTYPE acct_balance IS NUMBER (5, 2);
```

この例は、NUMBER タイプのすべての属性を共有する acct_balance という名前のサブタイプを作成しますが、小数点の左側が 3 桁、小数点の右側が 2 桁を超えてはなりません。

（関数またはプロシージャヘッダー内の）引数宣言は、正式**引数*です。関数またはプロシージャに渡される値は、*実際**引数 です。関数またはプロシージャを呼び出すとき、呼び出し元は（0 個以上の）実際の引数を提供します。各実引数は、関数またはプロシージャの本体内に値を保持する仮引数に割り当てられます。

仮引数が制約付きサブタイプとして宣言されている場合:

- AdvancedServer は、関数を呼び出すときに実際の引数を仮引数に割り当てるときにサブタイプ制約を強制しません。

- AdvancedServer は、プロシージャを呼び出すときに実際の引数を仮引数に割り当てるときにサブタイプ制約を強制します。

Using the %TYPE Operator

%TYPE 表記を使用して、列に固定されたサブタイプを宣言できます。例:

```
SUBTYPE emp_type IS emp.empno%TYPE
```

このコマンドは、ベースタイプが emp テーブルの empno 列のタイプと一致する emp_type という名前のサブタイプを作成します。列に基づくサブタイプは、列サイズの制約を共有します。NOT NULL および CHECK 制約は継承されません。

Subtype Conversion

制約のないサブタイプは、ベースとなるタイプのエイリアスです。タイプ subtype（制約なし）の変数は、変換なしで基本タイプの変数と交換可能です。

制約付きサブタイプの変数は、変換せずに基本タイプの変数と交換できますが、基本タイプの変数は、サブタイプの制約に準拠している場合にのみ、制約付きサブタイプと交換できます。制約サブタイプの変数は、同じサブタイプに基づいており、制約値が変換先のサブタイプの値内にある場合、暗黙的に別のサブタイプに変換される場合があります。

第 2 章

SPL Programs

SPL は、手続き型のブロック構造化言語です。SPL を使用して作成できるプログラムには、*procedures*、*functions*、*triggers*、および**packages** の 4 種類があります。

さらに、SPL はサブプログラムの作成に使用されます。サブプログラム*は、*サブプロシージャ*または*サブファンクション*を指します。これらは外観はプロシージャおよび関数とほぼ同じですが、プロシージャおよび関数は*スタンドアロンプログラム*であり、データベースに個別に保存され、他の SPL プログラムまたは PSQL から呼び出されます。サブプログラムは、作成されたスタンドアロンプログラム内からのみ呼び出すことができます。

2.1 SPL Block Structure

プログラムがプロシージャ、関数、サブプログラム、またはトリガーであるかどうかに関係なく、SPL プログラムは同じ**block** 構造を持ちます。ブロックは最大 3 つのセクションで構成されます-オプションの宣言セクション、必須の実行可能セクション、およびオプションの例外セクション。少なくとも、ブロックには、キーワード BEGIN および END 内の 1 つ以上の SPL ステートメントで構成される実行可能セクションがあります。

オプションの宣言セクションは、実行可能セクションおよび例外セクション内のステートメントで使用される変数、カーソル、型、およびサブプログラムを宣言するために使用されます。宣言は、実行可能セクションの BEGIN キーワードの直前に表示されます。ブロックが使用される場所のコンテキストに応じて、宣言セクションはキーワード DECLARE で始まる場合があります。

BEGIN - END ブロック内に例外セクションを含めることができます。例外セクションは、キーワード EXCEPTION で始まり、表示されるブロックの終わりまで続きます。ブロック内のステートメントによって例外がスローされた場合、プログラム制御は例外セクションに移動し、例外と例外セクションの内容に応じて、スローされた例外が処理される場合と処理されない場合があります。

ブロックの一般的な構造は次のとおりです。

```
[ [ DECLARE ]
    <pragmas>
    <declarations> ]
```

(次のページに続く)

(前のページからの続き)

```

BEGIN
    <statements>
[ EXCEPTION
    WHEN <exception_condition> THEN
        <statements> [, ...] ]
END;
```

pragmas はディレクティブです (AUTONOMOUS_TRANSACTION は現在サポートされているプラグマです)。declarations は、ブロックに対してローカルな 1 つ以上の変数、カーソル、型、またはサブプログラムの宣言です。サブプログラム宣言が含まれる場合、他のすべての変数、カーソル、および型宣言の後に宣言する必要があります。各宣言はセミコロンで終了する必要があります。キーワード DECLARE の使用は、ブロックが表示されるコンテキストに依存します。

statements は 1 つ以上の SPL ステートメントです。各ステートメントはセミコロンで終了する必要があります。キーワード END で示されるブロックの終わりもセミコロンで終了する必要があります。

存在する場合、キーワード EXCEPTION は例外セクションの開始を示します。exception_condition は、1 つ以上のタイプの例外をテストする条件式です。例外が exception_condition の例外の 1 つと一致する場合、WHEN exception_condition 節に続く statements が実行されます。1 つ以上の WHEN exception_condition 節があり、それぞれに statements が続きます。

注釈: BEGIN/END ブロック自体は、ステートメントと見なされます。したがって、ブロックはネストできます。例外セクションには、ネストされたブロックも含まれる場合があります。

以下は、実行可能セクション内の NULL ステートメントで構成される最も単純なブロックです。NULL ステートメントは、何も実行しない実行可能ステートメントです。

```

BEGIN
    NULL;
END;
```

次のブロックには、宣言セクションと実行可能セクションが含まれています。

```

DECLARE
    v_numerator    NUMBER(2);
    v_denominator  NUMBER(2);
    v_result        NUMBER(5,2);
BEGIN
    v_numerator := 75;
    v_denominator := 14;
    v_result := v_numerator / v_denominator;
    DBMS_OUTPUT.PUT_LINE(v_numerator || ' divided by ' || v_denominator ||
        ' is ' || v_result);
```

(次のページに続く)

(前のページからの続き)

```
END;
```

この例では、データ型 NUMBER の 3 つの数値変数が宣言されています。2 つの変数に値が割り当てられ、1 つの数値が他の数値で除算され、結果が 3 番目の変数に格納されて表示されます。実行すると、出力は次のようになります。

```
75 divided by 14 is 5.36
```

次のブロックは、宣言、実行可能ファイル、および例外で構成されています。

```
DECLARE
    v_numerator    NUMBER(2);
    v_denominator   NUMBER(2);
    v_result        NUMBER(5,2);
BEGIN
    v_numerator := 75;
    v_denominator := 0;
    v_result := v_numerator / v_denominator;
    DBMS_OUTPUT.PUT_LINE(v_numerator || ' divided by ' || v_denominator ||
        ' is ' || v_result);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An exception occurred');
END;
```

次の出力は、例外セクション内のステートメントがゼロ除算の結果として実行されることを示しています。

```
An exception occurred
```

2.2 Anonymous Blocks

ブロックは通常、プロシージャ、関数、サブプログラム、またはトリガーの一部として記述されます。プロシージャ、関数、およびトリガープログラムには名前が付けられ、再利用のためにデータベースに保存されます。迅速な（1 回限りの）実行（テストなど）の場合、名前を指定したりデータベースに保存したりせずに、ブロックを入力するだけです。

このタイプのブロックは、匿名ブロックと呼ばれます。匿名ブロックは名前が付けられておらず、データベースに保存されていません。ブロックが実行され、アプリケーションバッファから消去されると、ブロックコードがアプリケーションに再入力されない限り、ブロックを再実行できません。

通常、同じコードブロックが何度も再実行されます。コードを毎回再入力する必要なく、コードブロックを繰り返し実行するには、いくつかの簡単な変更を加えて、匿名ブロックをプロシージャまたは関数に変換できます。以下のセクションでは、データベースに格納し、別のプロシージャ、関数、またはアプリケーションプログラムによ

て繰り返し呼び出すことができるプロシージャまたは関数を作成する方法について説明します。

2.3 Procedures Overview

プロシージャは、個々の SPL プログラムステートメントとして呼び出されるか呼び出されるスタンドアロンの SPL プログラムです。呼び出されると、プロシージャはオプションで入力パラメータの形式で呼び出し元から値を受け取り、オプションで出力パラメータの形式で呼び出し元に値を返すことができます。

2.3.1 Creating a Procedure

CREATE PROCEDURE コマンドは、データベースに保存されるスタンドアロンプロシージャを定義し、名前を付けます。

スキーマ名が含まれている場合、プロシージャは指定されたスキーマに作成されます。それ以外の場合は、現在のスキーマで作成されます。新しいプロシージャの名前は、同じスキーマ内の同じ入力引数タイプを持つ既存のプロシージャと一致してはなりません。ただし、異なる入力引数タイプのプロシージャは名前を共有する場合があります（これは*オーバーロード*と呼ばれます）。(プロシージャのオーバーロードは AdvancedServer の機能です。ストアドスタンドアロンプロシージャのオーバーロードは Oracle データベースと互換性がありません。)

既存のプロシージャの定義を更新するには、CREATE OR REPLACE PROCEDURE を使用します。この方法でプロシージャの名前または引数の型を変更することはできません（実際に試してみると、実際には新しい別個のプロシージャを作成することになります）。OUT パラメータを使用する場合、OUT パラメータのタイプを変更するには、プロシージャをドロップする必要があります。

```
CREATE [OR REPLACE] PROCEDURE <name> [ (<parameters> ) ]
[
    IMMUTABLE
  | STABLE
  | VOLATILE
  | DETERMINISTIC
  | [ NOT ] LEAKPROOF
  | CALLED ON NULL INPUT
  | RETURNS NULL ON NULL INPUT
  | STRICT
  | [ EXTERNAL ] SECURITY INVOKER
  | [ EXTERNAL ] SECURITY DEFINER
  | AUTHID DEFINER
  | AUTHID CURRENT_USER
  | PARALLEL { UNSAFE | RESTRICTED | SAFE }
  | COST <execution_cost>
  | ROWS <result_rows>
  | SET <configuration_parameter>
    { TO <value> | = <value> | FROM CURRENT }
... ]
{ IS | AS }
[ PRAGMA AUTONOMOUS_TRANSACTION; ]
[ <declarations> ]
```

(次のページに続く)

(前のページからの続き)

```
BEGIN
  <statements>
END [ <name> ];
```

Where:

name

name はプロシージャの識別子です。

parameters

parameters は仮パラメータのリストです。

declarations

declarations は、変数、カーソル、タイプ、またはサブプログラムの宣言です。サブプログラム宣言が含まれる場合、他のすべての変数、カーソル、および型宣言の後に宣言する必要があります。

statements

statements は SPL プログラムステートメントです (BEGIN - END ブロックには EXCEPTION セクションが含まれる場合があります)。

IMMUTABLE

STABLE

VOLATILE

これらの属性は、クエリオプティマイザーにプロシージャの動作を通知します。指定できる選択肢は 1 つだけです。VOLATILE がデフォルトの動作です。

IMMUTABLE は、プロシージャがデータベースを変更できず、同じ引数値が指定されたときに常に同じ結果に達することを示します。データベースの検索は行わず、引数リストに直接存在しない情報を使用しません。この句を含めると、すべて定数の引数を使用したプロシージャの呼び出しをすぐにプロシージャの値に置き換えることができます。

STABLE は、プロシージャがデータベースを変更できないこと、および単一のテーブルスキャン内で同じ引数値に対して一貫して同じ結果を返しますが、その結果は SQL ステートメント間で変わる可能性があることを示します。これは、データベースルックアップ、パラメータ変数 (現在のタイムゾーンなど) などに依存する手順に適した選択です。

VOLATILE は、1 回のテーブルスキャン内でもプロシージャ値が変更される可能性があるため、最適化できないことを示します。副作用のある関数は、結果がかなり予測可能である場合でも、呼び出しが最適化されてしまうのを防ぐために、揮発性に分類する必要があることに注意してください。

DETERMINISTIC

DETERMINISTIC は IMMUTABLE の同義語です。DETERMINISTIC プロシージャはデータベースを変更できず、同じ引数値を指定すると常に同じ結果になります。データベースの検索は行わず、引数リストに直接存在しない情報を使用しません。この句を含めると、すべて定数の引数を使用したプロシージャの呼び出しをすぐにプロシージャの値に置き換えることができます。

[NOT] LEAKPROOF

LEAKPROOF プロシージャには副作用がなく、プロシージャの呼び出しに使用される値に関する情報は表示されません。

CALLED ON NULL INPUT

RETURNS NULL ON NULL INPUT

STRICT

CALLED ON NULL INPUT（デフォルト）は、引数の一部が NULL の場合にプロシージャが正常に呼び出されることを示します。必要に応じて NULL の値を確認し、適切に対応するのは作成者の責任です。

RETURNS NULL ON NULL INPUT または STRICT は、その引数のいずれかが NULL である場合、常に NULL を返すことを示します。これらの句が指定されている場合、NULL 個の引数がある場合、プロシージャは実行されません。代わりに、NULL の結果が自動的に想定されます。

[EXTERNAL] SECURITY DEFINER

SECURITY DEFINER は、プロシージャを作成したユーザーの権限で実行することを指定します。これがデフォルトです。キーワード EXTERNAL は SQL 準拠に使用できますが、オプションです。

[EXTERNAL] SECURITY INVOKER

SECURITY INVOKER 句は、プロシージャを呼び出すユーザーの権限でプロシージャが実行されることを示します。キーワード EXTERNAL は SQL 準拠に使用できますが、オプションです。

AUTHID DEFINER

AUTHID CURRENT_USER

AUTHID DEFINER 節は [EXTERNAL] SECURITY DEFINER の同義語です。AUTHID 句が省略されている場合、または AUTHID DEFINER が指定されている場合、プロシージャ所有者の権限を使用して、データベースオブジェクトへのアクセス権限が決定されます。

AUTHID CURRENT_USER 節は [EXTERNAL] SECURITY INVOKER の同義語です。AUTHID CURRENT_USER を指定すると、プロシージャを実行している現在のユーザーの権限を使用して、アクセス権限が決定されます。

PARALLEL { UNSAFE | RESTRICTED | SAFE }

PARALLEL 句は、並列順次スキャン（並列モード）の使用を有効にします。パラレルシーケンシャルスキャンは、シリアルシーケンシャルスキャンとは対照的に、クエリ中に複数のワーカーを使用してリレーション

を並行してスキャンします。

UNSAFE に設定すると、プロシージャをパラレルモードで実行できません。このようなプロシージャが存在すると、シリアル実行プランが強制されます。これは、`PARALLEL` 句が省略された場合のデフォルト設定です。

RESTRICTED に設定すると、プロシージャはパラレルモードで実行できますが、実行はパラレルグループリーダーに制限されます。特定のリレーションの資格に並列制限のあるものがある場合、そのリレーションは並列処理のために選択されません。

SAFE に設定すると、プロシージャは制限なしでパラレルモードで実行できます。

`COST execution_cost`

`execution_cost` は、`cpu_operator_cost` を単位とする、プロシージャの推定実行コストを示す正数です。プロシージャがセットを返す場合、これは返される行ごとのコストです。値が大きいと、プランナーは必要以上に関数を評価することを避けようとしています。

`ROWS result_rows`

`result_rows` は正の数で、プランナがプロシージャが返すと予想される行の推定数を示します。これは、プロシージャがセットを返すように宣言されている場合にのみ許可されます。デフォルトの想定は 1000 行です。

`SET configuration_parameter { TO value | = value | FROM CURRENT }`

`SET` 句により、指定された構成パラメータは、プロシージャに入るときに指定された値に設定され、プロシージャが終了するときに以前の値に復元されます。`SET FROM CURRENT` は、パラメータのセッションの現在の値を、プロシージャに入るときに適用される値として保存します。

`SET` 句がプロシージャに付加されている場合、同じ変数に対してプロシージャ内で実行される `SET LOCAL` コマンドの効果はプロシージャに制限されます。構成パラメータの以前の値は、プロシージャの終了時に復元されます。通常の `SET` コマンド（`LOCAL` なし）は、以前の `SET LOCAL` コマンドの場合と同様に `SET` 句をオーバーライドし、現在のトランザクションがロールされない限り、プロシージャ終了後もそのようなコマンドの効果が持続します戻る。

`PRAGMA AUTONOMOUS_TRANSACTION`

`PRAGMA AUTONOMOUS_TRANSACTION` は、プロシージャを自律型トランザクションとして設定するディレクティブです。

注釈: `STRICT`, `LEAKPROOF`, `PARALLEL`, `COST`, `ROWS` および `SET` キーワードは AdvancedServer の拡張機能を提供し、Oracle ではサポートされていません。

注釈: デフォルトでは、ストアードプロシージャは `SECURITY DEFINERS` として作成されますが、`plpgsql` で記述

されている場合、ストアードプロシージャは SECURITY INVOKERS として作成されます。

Example

以下は、パラメーターを使用しない単純な手順の例です。

```
CREATE OR REPLACE PROCEDURE simple_procedure
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('That's all folks!');
END simple_procedure;
```

プロシージャは、AdvancedServer にプロシージャコードを入力することでデータベースに保存されます。

次の例は、プロシージャ宣言で AUTHID DEFINER および SET 句を使用する方法を示しています。update_salary プロシージャは、プロシージャを定義しているロールの特権を、プロシージャを呼び出しているロールに（プロシージャの実行中に）伝えます。

```
CREATE OR REPLACE PROCEDURE update_salary(id INT, new_salary NUMBER)
SET SEARCH_PATH = 'public' SET WORK_MEM = '1MB'
AUTHID DEFINER IS
BEGIN
    UPDATE emp SET salary = new_salary WHERE emp_id = id;
END;
```

SET 句を含めて、プロシージャの検索パスを public に、ワークメモリを 1MB に設定します。他の手順、関数、およびオブジェクトは、これらの設定の影響を受けません。

この例では、AUTHID DEFINER 句は、それ以外の場合はプロシージャ内のステートメントの実行を許可されないロールに一時的に特権を付与します。プロシージャを呼び出すロールに関連付けられた特権を使用するようにサーバーに指示するには、AUTHID DEFINER 句を AUTHID CURRENT_USER 句に置き換えます。

2.3.2 Calling a Procedure

プロシージャは、別の SPL プログラムから呼び出すことができます。それには、プロシージャ名の後にパラメータがあればセミコロンを続けて指定するだけです。

```
<name> [ ([ <parameters> ]) ];
```

Where:

name はプロシージャの識別子です。

parameters は実際のパラメータのリストです。

注釈: 渡される実際のパラメーターがない場合は、空のパラメーターリストを使用してプロシージャを呼び出すか、開き括弧と閉じ括弧を完全に省略できます。

注釈: プロシージャを呼び出すための構文は、**PSQL** または **EDB*Plus** で **EXEC** コマンドを使用して実行する場合の前述の構文図と同じです。EXEC コマンドの詳細については、*Oracle Developers Tools and Utilities Guide* のデータベース互換性を参照してください。

以下は、匿名ブロックからプロシージャを呼び出す例です。

```
BEGIN
    simple_procedure;
END;

That's all folks!
```

注釈: 各アプリケーションには、プロシージャを呼び出す独自の方法があります。たとえば、Java アプリケーションでは、アプリケーションプログラミングインターフェイスである **JDBC** が使用されます。

2.3.3 Deleting a Procedure

DROP PROCEDURE コマンドを使用して、データベースからプロシージャを削除できます。

```
DROP PROCEDURE [ IF EXISTS ] <name> [ (<parameters>) ]
    [ CASCADE | RESTRICT ];
```

ここで、**name** は削除するプロシージャの名前です。

注釈: パラメータリストの指定は、これがオーバーロードされたプロシージャである場合など、特定の状況下で **AdvancedServer** に必要です。**Oracle** では、パラメータリストを常に省略する必要があります。

注釈: **IF EXISTS**, **CASCADE** または **RESTRICT** の使用は、**Oracle** データベースと互換性がありません。これらのオプションの詳細については、『**Oracle** 開発者向けデータベース互換性リファレンスガイド』の **DROP PROCEDURE** コマンドを参照してください。

この例では、以前に作成されたプロシージャが削除されます。

```
DROP PROCEDURE simple_procedure;
```

2.4 Functions Overview

関数は、式として呼び出されるスタンドアロンの SPL プログラムです。評価されると、関数は、その関数が埋め込まれている式に代入される値を返します。関数は、オプションで、呼び出しプログラムから入力パラメーターの形式で値を取得できます。関数自体が値を返すという事実に加えて、関数はオプションで出力パラメーターの形式で追加の値を呼び出し元に返す場合があります。ただし、関数で出力パラメーターを使用することは、推奨されるプログラミング手法ではありません。

2.4.1 Creating a Function

CREATE FUNCTION コマンドは、データベースに保存されるスタンドアロン関数を定義して名前を付けます。

スキーマ名が含まれている場合、関数は指定されたスキーマに作成されます。それ以外の場合は、現在のスキーマで作成されます。新しい関数の名前は、同じスキーマ内の同じ入力引数タイプを持つ既存の関数と一致してはなりません。ただし、異なる入力引数タイプの関数は名前を共有する場合があります（これは*オーバーロード*と呼ばれます）。(関数のオーバーロードは AdvancedServer の機能です。保存されたスタンドアロン関数のオーバーロードは Oracle データベースと互換性がありません)。

既存の関数の定義を更新するには、CREATE OR REPLACE FUNCTION を使用します。この方法で関数の名前または引数の型を変更することはできません（試してみると、実際には新しい別個の関数を作成することになります）。また、CREATE OR REPLACE FUNCTION では、既存の関数の戻り値の型を変更できません。そのためには、関数をドロップして再作成する必要があります。また、OUT パラメータを使用する場合、関数をドロップしない限り、OUT パラメータのタイプを変更することはできません。

```
CREATE [ OR REPLACE ] FUNCTION <name> [ (<parameters>) ]
  RETURN <data_type>
  [
    IMMUTABLE
  | STABLE
  | VOLATILE
  | DETERMINISTIC
  | [ NOT ] LEAKPROOF
  | CALLED ON NULL INPUT
  | RETURNS NULL ON NULL INPUT
  | STRICT
  | [ EXTERNAL ] SECURITY INVOKER
  | [ EXTERNAL ] SECURITY DEFINER
  | AUTHID DEFINER
  | AUTHID CURRENT_USER
  | PARALLEL { UNSAFE | RESTRICTED | SAFE }
  | COST <execution_cost>
  | ROWS <result_rows>
  | SET <configuration_parameter>
    { TO <value> | = <value> | FROM CURRENT }
  ]
```

(次のページに続く)

(前のページからの続き)

```

...]
{ IS | AS }
  [ PRAGMA AUTONOMOUS_TRANSACTION; ]
  [ <declarations> ]
BEGIN
  <statements>
END [ <name> ];

```

Where:

name

name は関数の識別子です。

parameters

parameters は仮パラメータのリストです。

data_type

data_type は、関数の RETURN ステートメントによって返される値のデータ型です。

declarations

declarations は、変数、カーソル、タイプ、またはサブプログラムの宣言です。サブプログラム宣言が含まれる場合、他のすべての変数、カーソル、および型宣言の後に宣言する必要があります。

statements

statements は SPL プログラムステートメントです (BEGIN - END ブロックには EXCEPTION セクションが含まれる場合があります)。

IMMUTABLE

STABLE

VOLATILE

これらの属性は、クエリオプティマイザーに関数の動作を通知します。指定できる選択肢は 1 つだけです。VOLATILE がデフォルトの動作です。

IMMUTABLE は、関数がデータベースを変更できず、同じ引数値が指定されたときに常に同じ結果に達することを示します。データベースの検索は行わず、引数リストに直接存在しない情報を使用しません。この句を含めると、すべて定数の引数を使用した関数の呼び出しを、関数の値にすぐに置き換えることができます。

STABLE は、関数がデータベースを変更できないこと、および単一のテーブルスキャン内で、同じ引数値に対して一貫して同じ結果を返しますが、その結果は SQL ステートメント間で変わる可能性があることを示します。これは、データベースルックアップ、パラメーター変数 (現在のタイムゾーンなど) などに依存する機能に適した選択です。

VOLATILE は、1 回のテーブルスキャン内でも関数値が変更される可能性があるため、最適化できないことを示します。副作用のある関数は、結果がかなり予測可能である場合でも、呼び出しが最適化されてしまうのを防ぐために、揮発性に分類する必要があることに注意してください。

DETERMINISTIC

DETERMINISTIC は IMMUTABLE の同義語です。DETERMINISTIC 関数はデータベースを変更できず、同じ引数値が与えられたときに常に同じ結果に到達します。データベースの検索は行わず、引数リストに直接存在しない情報を使用しません。この句を含めると、すべて定数の引数を使用した関数の呼び出しを、関数の値にすぐに置き換えることができます。

[NOT] LEAKPROOF

LEAKPROOF 関数には副作用はなく、関数の呼び出しに使用される値に関する情報は表示されません。

CALLED ON NULL INPUT

RETURNS NULL ON NULL INPUT

STRICT

CALLED ON NULL INPUT（デフォルト）は、引数の一部が NULL の場合にプロシージャが正常に呼び出されることを示します。必要に応じて NULL の値を確認し、適切に対応するのは作成者の責任です。

RETURNS NULL ON NULL INPUT または STRICT は、その引数のいずれかが NULL である場合、常に NULL を返すことを示します。これらの句が指定されている場合、NULL 個の引数がある場合、プロシージャは実行されません。代わりに、NULL の結果が自動的に想定されます。

[EXTERNAL] SECURITY DEFINER

SECURITY DEFINER は、関数を作成したユーザーの権限で実行することを指定します。これがデフォルトです。キーワード EXTERNAL は SQL 準拠に使用できますが、オプションです。

[EXTERNAL] SECURITY INVOKER

SECURITY INVOKER 句は、関数がそれを呼び出すユーザーの権限で実行されることを示します。キーワード EXTERNAL は SQL 準拠に使用できますが、オプションです。

AUTHID DEFINER

AUTHID CURRENT_USER

AUTHID DEFINER 節は [EXTERNAL] SECURITY DEFINER の同義語です。AUTHID 句が省略されている場合、または AUTHID DEFINER が指定されている場合、関数所有者の権限を使用して、データベースオブジェクトへのアクセス権限が決定されます。

AUTHID CURRENT_USER 節は [EXTERNAL] SECURITY INVOKER の同義語です。AUTHID CURRENT_USER が指定されている場合、機能を実行している現在のユーザーの権限を使用してアクセス権限が決定されます。

PARALLEL { UNSAFE | RESTRICTED | SAFE }

PARALLEL 句は、並列順次スキャン（並列モード）の使用を有効にします。パラレルシーケンシャルスキャンは、シリアルシーケンシャルスキャンとは対照的に、クエリ中に複数のワーカーを使用してリレーションを並行してスキャンします。

UNSAFE に設定すると、関数をパラレルモードで実行できません。SQL ステートメントにこのような関数が存在すると、シリアル実行プランが強制されます。これは、PARALLEL 句が省略された場合のデフォルト設定です。

RESTRICTED に設定すると、関数はパラレルモードで実行できますが、実行はパラレルグループリーダーに制限されます。特定のリレーションの資格に並列制限のあるものがある場合、そのリレーションは並列処理のために選択されません。

SAFE に設定すると、関数は制限なしで並列モードで実行できます。

COST execution_cost

execution_cost は、cpu_operator_cost を単位とする、関数の推定実行コストを示す正の数です。関数がセットを返す場合、これは返される行ごとのコストです。値が大きいと、プランナーは必要以上に関数を評価することを避けようとしています。

ROWS result_rows

result_rows は正の数であり、プランナーが関数が返すと予想される行の推定数を示します。これは、関数がセットを返すように宣言されている場合にのみ許可されます。デフォルトの想定は 1000 行です。

SET configuration_parameter { TO value | = value | FROM CURRENT }

SET 句を使用すると、指定された構成パラメーターが、関数に入るときに指定された値に設定され、関数が終了するときに以前の値に復元されます。SET FROM CURRENT は、パラメータのセッションの現在の値を、関数が入力されたときに適用される値として保存します。

SET 句が関数に付加されている場合、同じ変数に対して関数内で実行される SET LOCAL コマンドの効果はその関数に制限されます。構成パラメーターの以前の値は、関数の終了時に復元されます。通常の SET コマンド（LOCAL なし）は、以前の SET LOCAL コマンドの場合と同様に SET 句をオーバーライドし、現在のトランザクションがロールされない限り、プロシージャ終了後もそのようなコマンドの効果が持続します。

PRAGMA AUTONOMOUS_TRANSACTION

PRAGMA AUTONOMOUS_TRANSACTION は、機能を自律型トランザクションとして設定するディレクティブです。

注釈: STRICT, LEAKPROOF, PARALLEL, COST, ROWS および SET キーワードは AdvancedServer の拡張機能を提供し、Oracle ではサポートされていません。

Examples

以下は、パラメーターを受け取らない単純な関数の例です。

```
CREATE OR REPLACE FUNCTION simple_function
    RETURN VARCHAR2
IS
BEGIN
    RETURN 'That''s All Folks!';
END simple_function;
```

次の関数は、2つの入力パラメーターを取ります。パラメーターについては、以降のセクションで詳しく説明します。

```
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal          NUMBER,
    p_comm         NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END emp_comp;
```

次の例は、関数宣言で AUTHID CURRENT_USER 句と STRICT キーワードを使用する方法を示しています。

```
CREATE OR REPLACE FUNCTION dept_salaries(dept_id int) RETURN NUMBER
    STRICT
    AUTHID CURRENT_USER
BEGIN
    RETURN QUERY (SELECT sum(salary) FROM emp WHERE deptno = id);
END;
```

渡された入力パラメーターが NULL の場合に NULL を返すようにサーバーに指示するには、STRICT キーワードを含めます。NULL 値が渡された場合、関数は実行されません。

dept_salaries 関数は、関数を呼び出しているロールの特権で実行されます。現在のユーザーに「従業員の給与を表示するために」emp テーブルを照会する SELECT ステートメントを実行するための十分な権限がない場合、関数はエラーを報告します。関数を定義したロールに関連付けられた特権を使用するようにサーバーに指示するには、AUTHID CURRENT_USER 句を AUTHID DEFINER 句に置き換えます。

2.4.2 Calling a Function

関数は、式が SPL ステートメント内で使用できる任意の場所で使用できます。関数は、名前を指定し、その後に括弧がある場合はそのパラメーターを指定するだけで呼び出されます。

```
<name> [ ([ <parameters> ]) ]
```

name は関数の名前です。parameters は実際のパラメータのリストです。

注釈: 渡される実際のパラメーターがない場合、関数は空のパラメーターリストで呼び出されるか、開き括弧と閉じ括弧が完全に省略されます。

以下に、別の SPL プログラムから関数を呼び出す方法を示します。

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(simple_function);
END;

That's All Folks!
```

関数は通常、次のように SQL ステートメント内で使用されます。

```
SELECT empno "EMPNO", ename "ENAME", sal "SAL", comm "COMM",
       emp_comp(sal, comm) "YEARLY COMPENSATION" FROM emp;
```

EMPNO	ENAME	SAL	COMM	YEARLY COMPENSATION
7369	SMITH	800.00		19200.00
7499	ALLEN	1600.00	300.00	45600.00
7521	WARD	1250.00	500.00	42000.00
7566	JONES	2975.00		71400.00
7654	MARTIN	1250.00	1400.00	63600.00
7698	BLAKE	2850.00		68400.00
7782	CLARK	2450.00		58800.00
7788	SCOTT	3000.00		72000.00
7839	KING	5000.00		120000.00
7844	TURNER	1500.00	0.00	36000.00
7876	ADAMS	1100.00		26400.00
7900	JAMES	950.00		22800.00
7902	FORD	3000.00		72000.00
7934	MILLER	1300.00		31200.00

(14 rows)

2.4.3 Deleting a Function

DROP FUNCTION コマンドを使用して、データベースから関数を削除できます。

```
DROP FUNCTION [ IF EXISTS ] <name> [ (<parameters>) ]
    [ CASCADE | RESTRICT ];
```

ここで、name は削除する関数の名前です。

注釈: パラメータリストの指定は、これがオーバーロードされた関数である場合など、特定の状況下で Advanced-Server に必要です。Oracle では、パラメータリストを常に省略する必要があります。

注釈: IF EXISTS, CASCADE または RESTRICT の使用は、Oracle データベースと互換性がありません。これらのオプションの詳細については、『Oracle 開発者向けデータベース互換性リファレンスガイド』の DROP FUNCTION コマンドを参照してください。

この例では、以前に作成された関数は削除されます。

```
DROP FUNCTION simple_function;
```

2.5 Procedure and Function Parameters

プロシージャと関数を使用する重要な側面は、呼び出し側プログラムからプロシージャまたは関数にデータを渡し、プロシージャまたは関数からデータを受信する機能です。これは、*parameters* を使用して実現されます。

パラメーターは、プロシージャまたは関数の名前の後に括弧で囲まれて、プロシージャまたは関数の定義で宣言されます。プロシージャまたは関数の定義で宣言されたパラメーターは、*仮パラメーター* と呼ばれます。プロシージャまたは関数が呼び出されると、呼び出し元プログラムは、呼び出されたプログラムの処理で使用される実際のデータと、呼び出されたプログラムの処理の結果を受け取る変数を提供します。プロシージャまたは関数が呼び出されたときに呼び出し元プログラムによって提供されるデータと変数は、*実際のパラメーター* と呼ばれます。

以下は、仮パラメータ宣言の一般的な形式です。

```
(<name> [ IN | OUT | IN OUT ] <data_type> [ DEFAULT <value> ])
```

name は仮パラメータに割り当てられた識別子です。指定されている場合、「12」は、プロシージャまたは関数への入力データを受信するためのパラメータを定義します。「12」パラメータもデフォルト値に初期化できます。指定した場合、OUT は、プロシージャまたは関数からデータを返すためのパラメータを定義します。指定した場合、IN OUT により、入力と出力の両方にパラメーターを使用できます。IN, OUT と IN OUT のすべてが省略されている場合、パラメータはデフォルトで「12」として定義されているように動作します。パラメータが IN, OUT であるか IN OUT であるかは、パラメータの **mode** と呼ばれます。「10」は、パラメータのデータ型を定義します。「11」は、呼び出しで実際のパラメーターが指定されていない場合に、呼び出されたプログラムの「12」パラメーターに割り当てられるデフォルト値です。

以下は、パラメーターをとるプロシージャの例です。

```
CREATE OR REPLACE PROCEDURE emp_query (
    p_deptno      IN      NUMBER,
    p_empno       IN OUT  NUMBER,
    p_ename       IN OUT  VARCHAR2,
    p_job         OUT     VARCHAR2,
    p_hiredate    OUT     DATE,
    p_sal         OUT     NUMBER
)
IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
        INTO p_empno, p_ename, p_job, p_hiredate, p_sal
        FROM emp
        WHERE deptno = p_deptno
           AND (empno = p_empno
              OR  ename = UPPER(p_ename));
END;
```

この例では、p_deptno は IN の仮パラメータ、p_empno と p_ename は IN OUT の仮パラメータ、p_job, p_hiredate と p_sal は OUT の仮パラメータです。

注釈: 前の例では、VARCHAR2 パラメーターで最大長が指定されておらず、NUMBER パラメーターで精度とスケールが指定されていませんでした。パラメーター宣言で長さ、精度、スケール、またはその他の制約を指定することは違法です。これらの制約は、プロシージャまたは関数が呼び出されるときに使用される実際のパラメータから自動的に継承されます。

emp_query プロシージャは、別のプログラムから呼び出して、実際のパラメータを渡すことができます。以下は、emp_query を呼び出す別の SPL プログラムの例です。

```
DECLARE
    v_deptno      NUMBER(2);
    v_empno       NUMBER(4);
    v_ename       VARCHAR2(10);
    v_job         VARCHAR2(9);
    v_hiredate    DATE;
    v_sal         NUMBER;
BEGIN
    v_deptno := 30;
    v_empno  := 7900;
    v_ename  := '';
    emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
```

(次のページに続く)

(前のページからの続き)

```

DBMS_OUTPUT.PUT_LINE('Name      : ' || v_ename);
DBMS_OUTPUT.PUT_LINE('Job       : ' || v_job);
DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
END;
```

この例では、v_deptno、v_empno、v_ename、v_job、v_hiredate と v_sal が実際のパラメーターです。

上記の例の出力は次のとおりです。

```

Department : 30
Employee No: 7900
Name       : JAMES
Job        : CLERK
Hire Date  : 03-DEC-81
Salary     : 950
```

2.5.1 Positional vs. Named Parameter Notation

パラメーターを関数またはプロシージャに渡す場合、*positional**または**named* パラメーター表記のいずれかを使用できます。位置表記法を使用してパラメーターを指定する場合、宣言されている順序でパラメーターをリストする必要があります。名前付き表記でパラメーターを指定する場合、パラメーターの順序は重要ではありません。

名前付き表記を使用してパラメーターを指定するには、各パラメーターの名前の後に矢印 (=>) とパラメーター値を指定します。名前付き表記はより冗長ですが、コードを読みやすく保守しやすくします。

位置および名前付きパラメーター表記法の使用を示す簡単な例は次のとおりです。

```

CREATE OR REPLACE PROCEDURE emp_info (
    p_deptno      IN      NUMBER,
    p_empno       IN OUT NUMBER,
    p_ename       IN OUT VARCHAR2,
)
IS
BEGIN
    dbms_output.put_line('Department Number =' || p_deptno);
    dbms_output.put_line('Employee Number =' || p_empno);
    dbms_output.put_line('Employee Name =' || p_ename);
END;
```

位置表記法を使用してプロシージャを呼び出すには、次を渡します。

```
emp_info(30, 7455, 'Clark');
```

名前付き表記を使用してプロシージャを呼び出すには、次を渡します。

```
emp_info(p_ename =>'Clark', p_empno=>7455, p_deptno=>30);
```

名前付き表記を使用すると、パラメーターリストが変更された場合、パラメーターの順序が変更された場合、または新しいオプションパラメーターが追加された場合に、プロシージャのパラメーターリストを再配置する必要性を軽減できます。

引数にデフォルト値があり、引数が末尾の引数ではない場合、名前付き表記を使用してプロシージャまたは関数を呼び出す必要があります。次の例は、2つの主要なデフォルト引数を持つプロシージャを示しています。

```
CREATE OR REPLACE PROCEDURE check_balance (
    p_customerID  IN NUMBER DEFAULT NULL,
    p_balance     IN NUMBER DEFAULT NULL,
    p_amount      IN NUMBER
)
IS
DECLARE
    balance NUMBER;
BEGIN
    IF (p_balance IS NULL AND p_customerID IS NULL) THEN
        RAISE_APPLICATION_ERROR
            (-20010, 'Must provide balance or customer');
    ELSEIF (p_balance IS NOT NULL AND p_customerID IS NOT NULL) THEN
        RAISE_APPLICATION_ERROR
            (-20020, 'Must provide balance or customer, not both');
    ELSEIF (p_balance IS NULL) THEN
        balance := getCustomerBalance(p_customerID);
    ELSE
        balance := p_balance;
    END IF;

    IF (amount > balance) THEN
        RAISE_APPLICATION_ERROR
            (-20030, 'Balance insufficient');
    END IF;
END;
```

名前付き表記を使用することで、末尾以外の引数値のみを省略できます（このプロシージャを呼び出す場合）。位置表記法を使用する場合、デフォルトでは末尾の引数のみが許可されます。次の引数を使用してこのプロシージャを呼び出すことができます。

```
check_balance(p_customerID => 10, p_amount = 500.00)
```

```
check_balance(p_balance => 1000.00, p_amount = 500.00)
```

位置表記法と名前表記法（混合表記法）の組み合わせを使用して、パラメーターを指定できます。混合パラメータ表記の使用を示す簡単な例は次のとおりです。

```
CREATE OR REPLACE PROCEDURE emp_info (  
    p_deptno      IN      NUMBER,  
    p_empno       IN OUT NUMBER,  
    p_ename       IN OUT VARCHAR2,  
)  
IS  
BEGIN  
    dbms_output.put_line('Department Number =' || p_deptno);  
    dbms_output.put_line('Employee Number =' || p_empno);  
    dbms_output.put_line('Employee Name =' || p_ename);  
END;
```

混合表記を使用してプロシージャを呼び出すことができます。

```
emp_info(30, p_ename =>'Clark', p_empno=>7455);
```

混合表記法を使用する場合、名前付き引数を位置引数の前に置くことはできないことに注意してください。

2.5.2 Parameter Modes

前述のように、パラメータには IN, OUT または IN OUT の 3 つのモードのいずれかがあります。仮パラメータの次の特性は、そのモードに依存しています。

- プロシージャまたは関数が呼び出されたときの初期値。
- 呼び出されたプロシージャまたは関数が仮パラメータを変更できるかどうか。
- 実際のパラメーター値が呼び出し元プログラムから呼び出し先プログラムに渡される方法。
- 呼び出されたプログラムで未処理の例外が発生すると、仮パラメーター値はどうなりますか。

次の表は、モードに応じたパラメーターの動作をまとめたものです。

モードプロパティ	IN	INOUT	OUT
以下に初期化される仮パラメーター:	実パラメータ値	実パラメータ値	実パラメータ値
呼び出されたプログラムによって変更可能な正式なパラメーター?	NO	はい	はい
実際のパラメーターには以下が含まれます (通常のプログラム終了の呼び出し後)	呼び出し前の元の 実 際 の パ ラ メ ー タ ー 値	仮パラメータの最 後の値	仮パラメータの最 後の値
実パラメータには以下が含まれます: (呼び出されたプログラムで処理された例外の後)	呼び出し前の元の 実 際 の パ ラ メ ー タ ー 値	仮パラメータの最 後の値	仮パラメータの最 後の値
実パラメータには次が含まれます: (呼び出されたプログラムの未処理の例外の後)	呼び出し前の元の 実 際 の パ ラ メ ー タ ー 値	呼び出し前の元の 実 際 の パ ラ メ ー タ ー 値	呼び出し前の元の 実 際 の パ ラ メ ー タ ー 値

表に示すように、IN 仮パラメータは、デフォルト値で明示的に初期化されていない限り、呼び出される実際のパラメータに初期化されます。IN パラメータは呼び出されたプログラム内で参照できますが、呼び出されたプログラムは IN パラメータに新しい値を割り当てない場合があります。制御が呼び出し側プログラムに戻った後、実際のパラメーターには常に、呼び出し前に設定された値と同じ値が含まれます。

OUT 仮パラメータは、呼び出される実際のパラメータに初期化されます。呼び出されたプログラムは、新しい値を参照して仮パラメーターに割り当てることができます。呼び出されたプログラムが例外なく終了した場合、実際のパラメーターは、仮パラメーターで最後に設定された値を取ります。処理された例外が発生した場合、実際のパラメーターの値は、仮パラメーターに割り当てられた最後の値を取ります。未処理の例外が発生した場合、実際のパラメーターの値は呼び出し前の状態のままです。

IN パラメータと同様に、IN OUT 仮パラメータは、呼び出される実際のパラメータに初期化されます。OUT パラメーターと同様に、IN OUT 仮パラメーターは呼び出されたプログラムによって変更可能であり、仮パラメーターの最後の値は、呼び出されたプログラムが例外なく終了した場合、呼び出しプログラムの実際のパラメーターに渡されます。処理された例外が発生した場合、実際のパラメーターの値は、仮パラメーターに割り当てられた最後の値を引き継ぎます。未処理の例外が発生した場合、実際のパラメーターの値は呼び出し前と同じままです。

2.5.3 Using Default Values in Parameters

DEFAULT 句を含めるか、代入演算子 (:=) を CREATE PROCEDURE または CREATE FUNCTION ステートメントに使用することにより、仮パラメータのデフォルト値を設定できます。

仮パラメータ宣言の一般的な形式は次のとおりです。

```
(<name> [ IN|OUT|IN OUT ] <data_type> [{DEFAULT | := } <expr> ])
```

name はパラメータに割り当てられた識別子です。

IN|OUT|IN OUT はパラメータモードを指定します。

data_type は変数に割り当てられたデータ型です。

expr はパラメータに割り当てられたデフォルト値です。DEFAULT 句を含めない場合、呼び出し元はパラメーターの値を提供する必要があります。

デフォルト値は、関数またはプロシージャが呼び出されるたびに評価されます。たとえば、タイプ DATE のパラメーターに SYSDATE を割り当てると、パラメーターには、プロシージャまたは関数が作成された時刻ではなく、現在の呼び出しの時刻が設定されます。

次の簡単な手順は、代入演算子を使用して、デフォルト値の SYSDATE をパラメーター hiredate: に設定する方法を示しています

```
CREATE OR REPLACE PROCEDURE hire_emp (  
    p_empno          NUMBER,  
    p_ename          VARCHAR2,  
    p_hiredate       DATE := SYSDATE  
)  
IS  
BEGIN  
    INSERT INTO emp(empno, ename, hiredate)  
        VALUES(p_empno, p_ename, p_hiredate);  
  
    DBMS_OUTPUT.PUT_LINE('Hired!');  
END hire_emp;
```

パラメータ宣言にデフォルト値が含まれる場合、プロシージャを呼び出すときに、実際のパラメータリストからパラメータを省略できます。サンプルプロシージャ (hire_emp) の呼び出しには、従業員番号 (p_empno) と従業員名 (p_ename) の 2 つの引数を含める必要があります。3 番目のパラメーター (p_hiredate) のデフォルトは SYSDATE: の値です

```
hire_emp (7575, Clark)
```

プロシージャを呼び出すときに実際のパラメータの値を含めると、その値がデフォルト値よりも優先されます。

```
hire_emp (7575, Clark, 15-FEB-2010)
```

SYSDATE の現在の値に関係なく、February 15, 2010 の雇用者を持つ新しい従業員を追加します。

代入演算子の代わりに DEFAULT キーワードを使用して、同じ手順を記述できます。

```
CREATE OR REPLACE PROCEDURE hire_emp (  
    p_empno          NUMBER,  
    p_ename          VARCHAR2,  
    p_hiredate       DATE DEFAULT SYSDATE  
)
```

(次のページに続く)

(前のページからの続き)

```
IS
BEGIN
    INSERT INTO emp(empno, ename, hiredate)
        VALUES(p_empno, p_ename, p_hiredate);

    DBMS_OUTPUT.PUT_LINE('Hired!');
END hire_emp;
```

2.6 Subprograms – Subprocedures and Subfunctions

SPL プロシージャおよび関数プログラムの機能と機能は、SPL コードをサブプロシージャおよびサブ関数に編成することにより、適切に構造化され保守可能なプログラムを構築するために有利に使用できます。

SPL プログラム内のサブプロシージャとサブ関数を宣言することにより、比較的大きな SPL プログラム内の異なる場所から同じ SPL コードを複数回呼び出すことができます。

サブプロシージャとサブ関数には、次の特性があります。

- サブプロシージャとサブ関数の構文、構造、機能は、スタンドアロンのプロシージャと関数と実質的に同じです。主な違いは、サブプログラムを宣言するために CREATE PROCEDURE または CREATE FUNCTION の代わりにキーワード PROCEDURE または FUNCTION を使用することです。
- サブプロシージャとサブ関数は、それ自体で宣言された識別子（つまり、変数、カーソル、型、およびその他のサブプログラム）を分離します。つまり、これらの識別子は、サブプロシージャまたはサブ関数の外部にある上位の親レベルの SPL プログラムまたはサブプログラムからアクセスまたは変更することはできません。これにより、サブプロシージャおよびサブ関数の結果が信頼性が高く予測可能になります。
- サブプロシージャとサブ関数の宣言セクションには、独自のサブプロシージャとサブ関数を含めることができます。したがって、サブプログラムのマルチレベル階層がスタンドアロンプログラムに存在する可能性があります。階層内で、サブプログラムは上位レベルの親サブプログラムの識別子にアクセスし、上位レベルの親サブプログラムを呼び出すこともできます。ただし、識別子および呼び出しへの同じアクセスは、階層内の下位レベルの子サブプログラムに対しては実行できません。

サブプロシージャおよびサブ関数は、次のタイプの SPL プログラムのいずれかから宣言および呼び出すことができます。

- スタンドアロンの手順と機能
- 匿名ブロック
- トリガー
- パッケージ
- オブジェクト型本体のプロシージャおよび関数メソッド

- 上記のプログラムのいずれかで宣言されたサブプロシージャとサブ関数

サブプロシージャおよびサブ関数の構造とアクセスに関するルールについては、次のセクションで詳しく説明します。

2.6.1 Creating a Subprocedure

宣言セクションで指定された PROCEDURE 句は、そのブロックにローカルなサブプロシージャを定義し、名前を付けます。

ブロックという用語は、オプションの宣言セクション、必須の実行可能セクション、およびオプションの例外セクションで構成される SPL ブロック構造を指します。ブロックは、スタンドアロンのプロシージャとファンクション、匿名ブロック、サブプログラム、トリガー、パッケージ、オブジェクト型メソッドの構造です。

フレーズ*識別子はブロックに対してローカルです* は、識別子（つまり、変数、カーソル、型、またはサブプログラム）がそのブロックの宣言セクション内で宣言されているため、実行可能セクション内の SPL コードからアクセスできることを意味しますそのブロックのオプションの例外セクション。

サブプロシージャは、宣言セクションに含まれる他のすべての変数、カーソル、および型宣言の後にのみ宣言できます。（つまり、サブプログラムは宣言の最後のセットでなければなりません。）

```
PROCEDURE <name> [ (<parameters>) ]
{ IS | AS }
    [ PRAGMA AUTONOMOUS_TRANSACTION; ]
    [ <declarations> ]
BEGIN
    <statements>
END [ <name> ];
```

Where:

name

name はサブプロシージャの識別子です。

parameters

parameters は仮パラメータのリストです。

PRAGMA AUTONOMOUS_TRANSACTION

PRAGMA AUTONOMOUS_TRANSACTION は、サブプロシージャを自律型トランザクションとして設定するディレクティブです。

declarations

declarations は、変数、カーソル、タイプ、またはサブプログラムの宣言です。サブプログラム宣言が含まれる場合、他のすべての変数、カーソル、および型宣言の後に宣言する必要があります。

statements

statements は SPL プログラムステートメントです（BEGIN - END ブロックには EXCEPTION セクションが含まれる場合があります）。

Examples

次の例は、匿名ブロック内のサブプロシージャです。

```
DECLARE
    PROCEDURE list_emp
    IS
        v_empno    NUMBER(4);
        v_ename    VARCHAR2(10);
        CURSOR emp_cur IS
            SELECT empno, ename FROM emp ORDER BY empno;
    BEGIN
        OPEN emp_cur;
        DBMS_OUTPUT.PUT_LINE('Subprocedure list_emp:');
        DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
        DBMS_OUTPUT.PUT_LINE('-----');
        LOOP
            FETCH emp_cur INTO v_empno, v_ename;
            EXIT WHEN emp_cur%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(v_empno || '    ' || v_ename);
        END LOOP;
        CLOSE emp_cur;
    END;
BEGIN
    list_emp;
END;
```

この匿名ブロックを呼び出すと、次の出力が生成されます。

```
Subprocedure list_emp:
EMPNO    ENAME
-----
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER
```

次の例は、トリガー内のサブプロシージャです。

```

CREATE OR REPLACE TRIGGER dept_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON dept
DECLARE
    v_action          VARCHAR2(24);
    PROCEDURE display_action (
        p_action      IN  VARCHAR2
    )
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('User ' || USER || ' ' || p_action ||
            ' dept on ' || TO_CHAR(SYSDATE, 'YYYY-MM-DD'));
    END display_action;
BEGIN
    IF INSERTING THEN
        v_action := 'added';
    ELSIF UPDATING THEN
        v_action := 'updated';
    ELSIF DELETING THEN
        v_action := 'deleted';
    END IF;
    display_action(v_action);
END;

```

このトリガーを呼び出すと、次の出力が生成されます。

```

INSERT INTO dept VALUES (50, 'HR', 'DENVER');

User enterprisedb added dept on 2016-07-26

```

2.6.2 Creating a Subfunction

宣言セクションで指定された FUNCTION 句は、そのブロックにローカルなサブ関数を定義して名前を付けます。

ブロック という用語は、オプションの宣言セクション、必須の実行可能セクション、およびオプションの例外セクションで構成される SPL ブロック構造を指します。ブロックは、スタンドアロンのプロシージャとファンクション、匿名ブロック、サブプログラム、トリガー、パッケージ、オブジェクト型メソッドの構造です。

フレーズ*識別子はブロックに対してローカルです* は、識別子（つまり、変数、カーソル、型、またはサブプログラム）がそのブロックの宣言セクション内で宣言されているため、実行可能セクション内の SPL コードからアクセスできることを意味しますそのブロックのオプションの例外セクション。

```

FUNCTION <name> [ (<parameters>) ]
RETURN <data_type>
{ IS | AS }
    [ PRAGMA AUTONOMOUS_TRANSACTION; ]

```

(次のページに続く)

(前のページからの続き)

```
[ <declarations> ]
BEGIN
  <statements>
END [ <name> ];
```

Where:

`name`

`name` はサブ機能の識別子です。

`parameters`

`parameters` は仮パラメータのリストです。

`data_type`

`data_type` は、関数の RETURN ステートメントによって返される値のデータ型です。

PRAGMA AUTONOMOUS_TRANSACTION

PRAGMA AUTONOMOUS_TRANSACTION は、サブ機能を自律型トランザクションとして設定するディレクティブです。

`declarations`

`declarations` は、変数、カーソル、タイプ、またはサブプログラムの宣言です。サブプログラム宣言が含まれる場合、他のすべての変数、カーソル、および型宣言の後に宣言する必要があります。

`statements`

`statements` は SPL プログラムステートメントです (BEGIN - END ブロックには EXCEPTION セクションが含まれる場合があります)。

Examples

次の例は、再帰サブ関数の使用法を示しています。

```
DECLARE
  FUNCTION factorial (
    n          BINARY_INTEGER
  ) RETURN BINARY_INTEGER
  IS
  BEGIN
    IF n = 1 THEN
      RETURN n;
    ELSE
      RETURN n * factorial(n-1);
    END IF;
```

(次のページに続く)

(前のページからの続き)

```

    END factorial;
BEGIN
    FOR i IN 1..5 LOOP
        DBMS_OUTPUT.PUT_LINE(i || '!' = ' || factorial(i));
    END LOOP;
END;
```

例からの出力は次のとおりです。

```

1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

2.6.3 Block Relationships

このセクションでは、SPL プログラムで宣言できるブロック間の関係の用語について説明します。サブプログラムを呼び出し、ブロック内で宣言された識別子にアクセスする機能は、この関係に依存します。

基本的な用語は次のとおりです。

- **ブロック** は、オプションの宣言セクション、必須の実行可能セクション、およびオプションの例外セクションで構成される基本的な SPL 構造です。ブロックは、スタンドアロンのプロシージャおよび関数プログラム、匿名ブロック、トリガー、パッケージ、およびサブプロシージャとサブ関数を実装します。
- ブロックにローカルな識別子（変数、カーソル、型、またはサブプログラム）は、特定のブロックの宣言セクション内で宣言されていることを意味します。このようなローカル識別子は、ブロックの実行可能セクションとオプションの例外セクションからアクセスできます。
- **親ブロック***には、別のブロック（*子ブロック）の宣言が含まれています。
- **子孫ブロック** は、特定の親ブロックから始まる子関係を形成するブロックのセットです。
- **祖先ブロック** は、特定の子ブロックから始まる親関係を形成するブロックのセットです。
- 子孫（または祖先）ブロックのセットは、**階層**を形成します。
- **level** は、最上位の祖先ブロックからの特定のブロックの序数です。たとえば、スタンドアロンプロシージャが与えられた場合、このプロシージャの宣言セクション内で宣言されたサブプログラムはすべて同じレベルにあり、たとえばレベル 1 と呼ばれます。スタンドアロンプロシージャで宣言されたサブプログラムの宣言セクション内の追加のサブプログラムは、レベル、レベル 2 です。
- **兄弟ブロック** は、同じ親ブロックを持つブロックのセットです（つまり、すべて同じブロックでローカルに宣言されます）。兄弟ブロックも常に相互に同じレベルにあります。

プロシージャ宣言セクションのセットの次の概略図は、ブロックのセットとそれらの周囲のブロックとの関係の例を示しています。

ブロックの左側にある 2 本の縦線は、兄弟ブロックのペアが 2 つあることを示しています。 block_1a と block_1b は 1 つのペアであり、 block_2a と block_2b は 2 番目のペアです。

各ブロックとその祖先の関係は、ブロックの右側に表示されます。最下位レベルの子ブロックから階層を上に進むと、3 つの階層パスが形成されます。最初は block_0, block_1a, block_2a と block_3 で構成されます。2 番目は block_0, block_1a と block_2b です。3 番目は block_0, block_1b と block_2b です。

```
CREATE PROCEDURE block_0
IS
    .
    +---- PROCEDURE block_1a      ----- Local to block_0
    |      IS
    |      .
    |      .
    |      .
    |      +--- PROCEDURE block_2a  ---- Local to block_1a and descendant
    |      |      IS                of block_0
    |      |      .
    |      |      .
    |      |      .
    |      |      PROCEDURE block_3  -- Local to block_2a and descendant
    |      |      IS                of block_1a, and block_0
    |      Siblings      .
    |      |      .
    |      |      .
    |      |      END block_3;
    |      |      END block_2a;
    |      +--- PROCEDURE block_2b  ---- Local to block_1a and descendant
    |      |      IS                of block_0
Siblings |      ,
    |      |      .
    |      |      .
    |      +--- END block_2b;
    |
    |      END block_1a;      -----+
+---- PROCEDURE block_1b;  ----- Local to block_0
    |      IS
    |      .
    |      .
    |      .
    |      PROCEDURE block_2b  ---- Local to block_1b and descendant
    |      IS                of block_0
    |      .
    |      .
    |      .
```

(次のページに続く)

(前のページからの続き)

```

|          END block_2b;          |
|                                |
+---- END block_1b;      -----+
BEGIN
    .
    .
    .
END block_0;

```

ブロックの位置に基づいてサブプログラムを呼び出すためのルールは、`InvokingSubprograms` で説明されています。

2.6.4 Invoking Subprograms

サブプログラムは、その名前と実際のパラメータを指定することにより、スタンドアロンのプロシージャまたはファンクションと同じ方法で起動されます。

サブプログラムは、なし、1 つ、または複数の修飾子で呼び出すことができます。これらの修飾子は、親サブプログラムの名前、またはサブプログラムが宣言された先祖階層を形成するラベル付き匿名ブロックです。

呼び出しは、次のように、サブプログラム名とその引数のいずれかで終わる修飾子のドット区切りリストとして指定されます。

```
[ [<qualifier_1.> ] [...] <qualifier_n.> ] <subprog> [ (<arguments> ) ]
```

指定されている場合、`qualifier_n` は `subprog` が宣言セクションで宣言されているサブプログラムです。前述の修飾子のリストは、階層の `qualifier_n` から `qualifier_1`、`qualifier_1` までの連続したパスに存在する必要があります。パス内の任意の祖先サブプログラムと次のいずれかを指定できます。

- サブプログラムを含むスタンドアロンプロシージャ名
- サブプログラムを含むスタンドアロン関数名
- サブプログラムを含むパッケージ名
- オブジェクト型メソッド内のサブプログラムを含むオブジェクト型名
- 宣言セクションが存在する場合は `DECLARE` キーワードの前、宣言セクションがない場合は `BEGIN` キーワードの前に含まれる匿名ブロックラベル。

注釈: `qualifier_1` はスキーマ名ではない場合があります。そうでない場合、サブプログラムの呼び出し時にエラーがスローされます。この `AdvancedServer` の制限は、スキーマ名を修飾子として使用できる `Oracle` データベースと互換性がありません。

`arguments` は、サブプロシージャまたはサブ関数に渡される実際のパラメータのリストです。

呼び出し時に、サブプログラムの検索は次のように行われます。

- 指定された順序の修飾子（呼び出しリストと呼ばれる）とともに、そのタイプ（つまり、サブプロシージャーまたはサブ関数）の呼び出されたサブプログラム名を使用して、同じ階層順序にある一致するブロックのセットを見つけます。検索は、最下位レベルがサブプログラムの呼び出し元のブロックであるブロック階層で開始されます。サブプログラムの宣言は、コードが上から下に向かって観察されるときに呼び出されるコード行の前の `SPL` コード内になければなりません。（この要件の例外は、前方宣言を使用して実現できます。前方宣言の詳細については、[UsingForwardDeclarations](#) を参照してください。）
- 呼び出しリストが、サブプログラムが呼び出されるブロックから始まるブロックの階層と一致しない場合は、前の開始ブロックの親で始まる呼び出しリストを一致させることにより比較が行われます。つまり、比較は階層を上に進んでいきます。
- 祖先の兄弟ブロックがある場合、呼び出しリスト比較には兄弟ブロックの階層も含まれますが、常に上位レベルで比較され、兄弟ブロックの子孫は比較されません。
- この比較プロセスは、最初の完全な一致が見つかるまで階層を上に進み、完全に一致する場合は、見つかったサブプログラムが呼び出されます。一致したサブプログラムの仮パラメータリストは、呼び出されたサブプログラムに指定された実際のパラメータリストに準拠している必要があります。そうでない場合、サブプログラムの呼び出し時にエラーが発生します。
- スタンドアロンプログラムまで検索した後に一致が見つからない場合、サブプログラムの呼び出し時にエラーがスローされます。

注釈: サブプログラム呼び出しの `AdvancedServer` 検索アルゴリズムは、`Oracle` データベースと完全に互換性がありません。`Oracle` の場合、検索は最初の修飾子の最初の一致（つまり `qualifier_1`）を探します。そのような一致が見つかった場合、残りのすべての修飾子、サブプログラム名、サブプログラムタイプ、および呼び出しの引数は、一致する最初の修飾子が見つかった階層コンテンツと一致する必要があります。一致しない場合、エラーがスローされます。`AdvancedServer` の場合、呼び出しのすべての修飾子、サブプログラム名、およびサブプログラムタイプが階層コンテンツと一致しない限り、一致は見つかりません。そのような完全に一致するものが最初に見つからない場合、`AdvancedServer` は検索を続けて階層を進めます。

呼び出し元のブロックに関連するサブプログラムの場所には、次のようにアクセスできます。

- ローカルブロックで宣言されたサブプログラムは、同じブロックの実行可能セクションまたは例外セクションから呼び出すことができます。
- 親または他の祖先ブロックで宣言されたサブプログラムは、親または他の祖先の子ブロックから呼び出すことができます。
- 兄弟ブロックで宣言されたサブプログラムは、兄弟ブロックまたは兄弟の子孫ブロックから呼び出すことができます。

ただし、サブプログラムの次の場所は、呼び出し元のブロックに関連してアクセスできません。

- 呼び出しが試行されるブロックの子孫であるブロックで宣言されたサブプログラム。
- 呼び出しが試行される兄弟ブロックの子孫であるブロックで宣言されたサブプログラム。

次の例は、前述のさまざまな条件を示しています。

Invoking Locally Declared Subprograms

次の例には、スタンドアロンプロシージャ `level_0` に含まれるブロックの単一階層が含まれています。プロシージャ `level_1a` の実行可能セクション内に、ローカルプロシージャ `level_2a` を呼び出す手段が、修飾子の有無にかかわらず表示されます。

また、ローカルプロシージャ `level_2a`（プロシージャ `level_3a`）の子孫へのアクセスは、修飾子の有無にかかわらず許可されないことに注意してください。これらの呼び出しは、例ではコメント化されています。

```
CREATE OR REPLACE PROCEDURE level_0
IS
    PROCEDURE level_1a
    IS
        PROCEDURE level_2a
        IS
            PROCEDURE level_3a
            IS
                BEGIN
                    DBMS_OUTPUT.PUT_LINE('..... BLOCK level_3a');
                    DBMS_OUTPUT.PUT_LINE('..... END BLOCK level_3a');
                END level_3a;
            BEGIN
                DBMS_OUTPUT.PUT_LINE('..... BLOCK level_2a');
                DBMS_OUTPUT.PUT_LINE('..... END BLOCK level_2a');
            END level_2a;
        BEGIN
            DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
            level_2a;                                -- Local block called
            level_1a.level_2a;                        -- Qualified local block
called
            level_0.level_1a.level_2a;                -- Double qualified local
block called
--            level_3a;                                -- Error - Descendant of
local block
--            level_2a.level_3a;                        -- Error - Descendant of
local block
            DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
        END level_1a;
    BEGIN
        DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
        level_1a;
```

(次のページに続く)

(前のページからの続き)

```
DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;
```

スタンドアロンプロシージャが呼び出されると、出力は次のようになります。これは、プロシージャ level_1a の実行可能セクションの呼び出しからプロシージャ level_2a が正常に呼び出されることを示しています。

```
BEGIN
    level_0;
END;

BLOCK level_0
.. BLOCK level_1a
..... BLOCK level_2a
..... END BLOCK level_2a
..... BLOCK level_2a
..... END BLOCK level_2a
..... BLOCK level_2a
..... END BLOCK level_2a
.. END BLOCK level_1a
END BLOCK level_0
```

子孫ブロックへの呼び出しのいずれかをコメント解除してプロシージャ level_0 を実行しようとする、エラーが発生します。

Invoking Subprograms Declared in Ancestor Blocks

次の例は、呼び出しが行われたブロックに関連する親および他の祖先ブロックで宣言されたサブプログラムを呼び出す方法を示しています。

この例では、プロシージャ level_3a の実行可能セクションは、親ブロックであるプロシージャ level_2a を呼び出します。(v_cnt は無限ループを避けるために使用されることに注意してください。)

```
CREATE OR REPLACE PROCEDURE level_0
IS
    v_cnt          NUMBER(2) := 0;
    PROCEDURE level_1a
    IS
        PROCEDURE level_2a
        IS
            PROCEDURE level_3a
            IS
                BEGIN
                    DBMS_OUTPUT.PUT_LINE('..... BLOCK level_3a');
                    v_cnt := v_cnt + 1;
                    IF v_cnt < 2 THEN
                        level_2a;          -- Parent block called
                    END IF;
                END;
            END;
        END;
    END;
END;
```

(次のページに続く)

(前のページからの続き)

```

        DBMS_OUTPUT.PUT_LINE('..... END BLOCK level_3a');
    END level_3a;
BEGIN
    DBMS_OUTPUT.PUT_LINE('..... BLOCK level_2a');
    level_3a;                                -- Local block called
    DBMS_OUTPUT.PUT_LINE('..... END BLOCK level_2a');
END level_2a;
BEGIN
    DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
    level_2a;                                -- Local block called
    DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
END level_1a;
BEGIN
    DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
    level_1a;
    DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;

```

結果の出力は次のとおりです。

```

BEGIN
    level_0;
END;

BLOCK level_0
.. BLOCK level_1a
..... BLOCK level_2a
..... BLOCK level_3a
..... BLOCK level_2a
..... BLOCK level_3a
..... END BLOCK level_3a
..... END BLOCK level_2a
..... END BLOCK level_3a
..... END BLOCK level_2a
.. END BLOCK level_1a
END BLOCK level_0

```

同様の例では、プロシージャ level_3a の実行可能セクションがプロシージャ level_1a を呼び出します。これは先祖階層のさらに上にあります。(v_cnt は無限ループを避けるために使用されることに注意してください。)

```

CREATE OR REPLACE PROCEDURE level_0
IS
    v_cnt          NUMBER(2) := 0;
    PROCEDURE level_1a
    IS
        PROCEDURE level_2a
        IS

```

(次のページに続く)

(前のページからの続き)

```

PROCEDURE level_3a
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('..... BLOCK level_3a');
    v_cnt := v_cnt + 1;
    IF v_cnt < 2 THEN
        level_1a;                -- Ancestor block called
    END IF;
    DBMS_OUTPUT.PUT_LINE('..... END BLOCK level_3a');
END level_3a;
BEGIN
    DBMS_OUTPUT.PUT_LINE('..... BLOCK level_2a');
    level_3a;                    -- Local block called
    DBMS_OUTPUT.PUT_LINE('..... END BLOCK level_2a');
END level_2a;
BEGIN
    DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
    level_2a;                    -- Local block called
    DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
END level_1a;
BEGIN
    DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
    level_1a;
    DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;

```

結果の出力は次のとおりです。

```

BEGIN
    level_0;
END;

BLOCK level_0
.. BLOCK level_1a
..... BLOCK level_2a
..... BLOCK level_3a
.. BLOCK level_1a
..... BLOCK level_2a
..... BLOCK level_3a
..... END BLOCK level_3a
..... END BLOCK level_2a
.. END BLOCK level_1a
..... END BLOCK level_3a
..... END BLOCK level_2a
.. END BLOCK level_1a
END BLOCK level_0

```

Invoking Subprograms Declared in Sibling Blocks

次の例は、サブプログラムの呼び出し元のローカルブロック、親ブロック、または他の祖先ブロックに関連する兄弟ブロックで宣言されているサブプログラムを呼び出す方法を示しています。

この例では、プロシージャ level_1b の実行可能セクションは、その兄弟ブロックであるプロシージャ level_1a を呼び出します。両方ともローカルからスタンドアロンの手順 level_0 です。

この呼び出しはエラーになるため、プロシージャ level_1b 内からの level_2a または同等の level_1a. level_2a の呼び出しはコメント化されていることに注意してください。兄弟ブロック (level_1a) の子孫サブプログラム (level_2a) の呼び出しは許可されていません。

```
CREATE OR REPLACE PROCEDURE level_0
IS
    v_cnt      NUMBER(2) := 0;
    PROCEDURE level_1a
    IS
        PROCEDURE level_2a
        IS
            BEGIN
                DBMS_OUTPUT.PUT_LINE('..... BLOCK level_2a');
                DBMS_OUTPUT.PUT_LINE('..... END BLOCK level_2a');
            END level_2a;
        BEGIN
            DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
            DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
        END level_1a;
        PROCEDURE level_1b
        IS
            BEGIN
                DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
                level_1a;                      -- Sibling block called
--      level_2a;                          -- Error - Descendant of
sibling block
--      level_1a.level_2a;                  -- Error - Descendant of
sibling block
                DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
            END level_1b;
        BEGIN
            DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
            level_1b;
            DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
        END level_0;
```

結果の出力は次のとおりです。

```
BEGIN
    level_0;
END;
```

(次のページに続く)

(前のページからの続き)

```

BLOCK level_0
.. BLOCK level_1b
.. BLOCK level_1a
.. END BLOCK level_1a
.. END BLOCK level_1b
END BLOCK level_0

```

次の例では、プロシージャ level_3b の先祖であるプロシージャ level_1b の兄弟であるプロシージャ level_1a が正常に呼び出されます。

```

CREATE OR REPLACE PROCEDURE level_0
IS
    PROCEDURE level_1a
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
    END level_1a;
    PROCEDURE level_1b
    IS
        PROCEDURE level_2b
        IS
            PROCEDURE level_3b
            IS
            BEGIN
                DBMS_OUTPUT.PUT_LINE('..... BLOCK level_3b');
                level_1a;                                -- Ancestor's sibling block
called
                level_0.level_1a;                        -- Qualified ancestor's
sibling block
                DBMS_OUTPUT.PUT_LINE('..... END BLOCK level_3b');
            END level_3b;
        BEGIN
            DBMS_OUTPUT.PUT_LINE('..... BLOCK level_2b');
            level_3b;                                    -- Local block called
            DBMS_OUTPUT.PUT_LINE('..... END BLOCK level_2b');
        END level_2b;
    BEGIN
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
        level_2b;                                        -- Local block called
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
    END level_1b;
BEGIN
    DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
    level_1b;
    DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;

```

結果の出力は次のとおりです。

```
BEGIN
    level_0;
END;

BLOCK level_0
.. BLOCK level_1b
..... BLOCK level_2b
..... BLOCK level_3b
.. BLOCK level_1a
.. END BLOCK level_1a
.. BLOCK level_1a
.. END BLOCK level_1a
..... END BLOCK level_3b
..... END BLOCK level_2b
.. END BLOCK level_1b
END BLOCK level_0
```

2.6.5 Using Forward Declarations

これまでに説明したように、サブプログラムが呼び出される場合、サブプログラムは、スタンドアロンプログラム内のブロック階層のどこかで、ただし呼び出される前に宣言されている必要があります。言い換えると、SPL コードを最初から最後までスキャンする場合、サブプログラム宣言はその呼び出しの前に検出されなければなりません。

ただし、サブプログラムの完全な宣言（つまり、オプションの宣言セクション、必須の実行可能セクション、およびオプションの例外セクション）が、コード内のポイントの後の SPL コードに表示されるように、SPL コードを構築する方法があります。呼び出されます。

これは、呼び出しの前に SPL コードに **forward 宣言** を挿入することで実現されます。前方宣言は、サブプロシージャまたはサブ関数の名前、仮パラメータ、およびサブ関数の場合の戻り値の型の指定です。

オプションの宣言セクション、実行可能セクション、およびオプションの例外セクションで構成される完全なサブプログラム仕様は、前方宣言と同じ宣言セクションで指定する必要がありますが、前方宣言でこのサブプログラムを呼び出す他のサブプログラム宣言の後に表示される場合があります。

前方宣言の一般的な使用法は、次のように 2 つのサブプログラムが相互に呼び出す場合です。

```
DECLARE
    FUNCTION add_one (
        p_add      IN NUMBER
    ) RETURN NUMBER;
    FUNCTION test_max (
        p_test      IN NUMBER)
    RETURN NUMBER
```

(次のページに続く)

(前のページからの続き)

```

IS
BEGIN
    IF p_test < 5 THEN
        RETURN add_one(p_test);
    END IF;
    DBMS_OUTPUT.PUT('Final value is ');
    RETURN p_test;
END;
FUNCTION add_one (
    p_add      IN NUMBER)
RETURN NUMBER
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Increase by 1');
    RETURN test_max(p_add + 1);
END;
BEGIN
    DBMS_OUTPUT.PUT_LINE(test_max(3));
END;

```

サブ関数 `test_max` はサブ関数 `add_one` を呼び出し、サブ関数 `test_max` も呼び出します。そのため、サブブロックの 1 つに前方宣言が必要です。これは、匿名ブロック宣言セクションの先頭の `add_one` に実装されます。

無名ブロックからの結果の出力は次のとおりです。

```

Increase by 1
Increase by 1
Final value is 5

```

2.6.6 Overloading Subprograms

一般に、同じ名前の同じタイプ（サブプロシージャまたはサブ関数）のサブプログラムは、兄弟ブロックでない限り（つまり、サブプログラムが同じローカルブロック）。

前のセクションで説明したように、修飾子の使用とサブプログラムの呼び出しが行われる場所に応じて、各サブプログラムを個別に呼び出すことができます。

ただし、仮パラメータの特定の側面が異なる限り、同じサブプログラムのタイプと名前のサブプログラムを兄弟として宣言することは可能です。これらの特性（サブプログラムのタイプ、名前、および仮パラメーターの仕様）は、一般にプログラムの*署名*として知られています。

仮パラメータ仕様の特定の側面を除いてシグネチャが同一である複数のサブプログラムの宣言は、サブプログラム*オーバーロード*と呼ばれます。

したがって、どの特定のオーバーロードされたサブプログラムを呼び出すかは、サブプログラムの呼び出しで指定

された実際のパラメーターとオーバーロードされたサブプログラムの正式なパラメーターリストの一致によって決まります。

次の違いにより、オーバーロードされたサブプログラムが許可されます。

- 仮パラメータの数は異なります。
- 対応する仮パラメータのデータ型の少なくとも 1 組（つまり、仮パラメーターリストの同じ順序で比較）は異なりますが、エイリアスではありません。データ型エイリアスについては、このセクションの後半で説明します。

次の違いだけでは、オーバーロードされたサブプログラムは許可されないことに注意してください。

- 異なる仮パラメーター名
- 対応する仮パラメータの異なるパラメータモード (IN, IN OUT, OUT)
- サブ関数の場合、RETURN 句の異なるデータ型

前述のように、オーバーロードされたサブプログラムを許可する違いの 1 つは異なるデータ型です。

ただし、特定のデータタイプには、エイリアスと呼ばれる代替名があり、テーブル定義に使用できます。

たとえば、CHAR または CHARACTER として指定できる固定長の文字データ型があります。CHAR VARYING, CHARACTER VARYING, VARCHAR, または VARCHAR2 として指定できる可変長文字データ型があります。整数には、BINARY_INTEGER, PLS_INTEGER, と INTEGER のデータ型があります。数字には、NUMBER, NUMERIC, DEC, と DECIMAL のデータ型があります。

AdvancedServer がサポートするデータ型の詳細については、次の EDB から入手可能な『Oracle Developers のデータベース互換性リファレンスガイド』を参照してください。

<https://www.enterprisedb.com/edb-docs>

したがって、オーバーロードされたサブプログラムを作成するときに、指定されたデータ型が相互のエイリアスである場合、仮パラメータのデータ型は異なるとは見なされません。

問題のデータ型を含むテーブル定義を表示することにより、特定のデータ型が他の型のエイリアスであるかどうかを判断できます。

たとえば、次のテーブル定義には、いくつかのデータ型とそのエイリアスが含まれています。

```
CREATE TABLE data_type_aliases (
  dt_BLOB          BLOB,
  dt_LONG_RAW      LONG RAW,
  dt_RAW           RAW(4),
  dt_BYTEA        BYTEA,
  dt_INTEGER       INTEGER,
  dt_BINARY_INTEGER BINARY_INTEGER,
  dt_PLS_INTEGER   PLS_INTEGER,
```

(次のページに続く)

(前のページからの続き)

```

dt_REAL          REAL,
dt_DOUBLE_PRECISION DOUBLE PRECISION,
dt_FLOAT         FLOAT,
dt_NUMBER        NUMBER,
dt_DECIMAL       DECIMAL,
dt_NUMERIC       NUMERIC,
dt_CHAR          CHAR,
dt_CHARACTER     CHARACTER,
dt_VARCHAR2      VARCHAR2(4),
dt_CHAR_VARYING  CHAR VARYING(4),
dt_VARCHAR       VARCHAR(4)
);

```

PSQL の /d コマンドを使用してテーブル定義を表示すると、タイプ列には、テーブル定義のデータ型に基づいて各列に内部的に割り当てられたデータ型が表示されます。

```

\d data_type_aliases

```

Column	Type	Modifiers
dt_blob	bytea	
dt_long_raw	bytea	
dt_raw	bytea(4)	
dt_bytea	bytea	
dt_integer	integer	
dt_binary_integer	integer	
dt_pls_integer	integer	
dt_real	real	
dt_double_precision	double precision	
dt_float	double precision	
dt_number	numeric	
dt_decimal	numeric	
dt_numeric	numeric	
dt_char	character(1)	
dt_character	character(1)	
dt_varchar2	character varying(4)	
dt_char_varying	character varying(4)	
dt_varchar	character varying(4)	

この例では、データ型の基本セットは bytea, integer, real, double precision, numeric, character と character varying です。

オーバーロードされたサブプログラムを宣言しようとする、エイリアスである正式なパラメータデータ型のペアでは、サブプログラムのオーバーロードを許可するには不十分です。したがって、データ型が INTEGER と PLS_INTEGER のパラメーターはサブプログラムのペアをオーバーロードできませんが、データ型 INTEGER と REAL、または INTEGER と FLOAT、または INTEGER と NUMBER はオーバーロードできますサブプログラム。

注釈: 正式なパラメータデータ型に基づくオーバーロードルールは、Oracle データベースと互換性がありません。一般に、AdvancedServer ルールはより柔軟であり、Oracle データベースでプロシージャまたは関数を作成しようとするとエラーが発生する特定の組み合わせが AdvancedServer で許可されています。

オーバーロードに使用されるデータ型の特定のペアでは、サブプログラムの実行中に発生したエラーを回避するために、サブプログラムの呼び出しで指定された引数のキャストが必要になる場合があります。サブプログラムの呼び出しには、データ型を具体的に識別できる実際のパラメーターリストを含める必要があります。オーバーロードされたデータ型の特定のペアでは、データ型を明示的に識別するために CAST 関数が必要になる場合があります。たとえば、呼び出し中にキャストが必要なオーバーロードされたデータ型のペアは、CHAR と VARCHAR2、または NUMBER と REAL です。

次の例は、匿名ブロック内から呼び出されるオーバーロードされたサブ関数のグループを示しています。匿名ブロックの実行可能セクションには、特定のデータ型でオーバーロードされた関数を呼び出すための CAST 関数の使用が含まれています。

```
DECLARE
    FUNCTION add_it (
        p_add_1      IN BINARY_INTEGER,
        p_add_2      IN BINARY_INTEGER
    ) RETURN VARCHAR2
    IS
    BEGIN
        RETURN 'add_it BINARY_INTEGER: ' || TO_CHAR(p_add_1 +
p_add_2,9999.9999);
    END add_it;
    FUNCTION add_it (
        p_add_1      IN NUMBER,
        p_add_2      IN NUMBER
    ) RETURN VARCHAR2
    IS
    BEGIN
        RETURN 'add_it NUMBER: ' || TO_CHAR(p_add_1 + p_add_2,999.9999);
    END add_it;
    FUNCTION add_it (
        p_add_1      IN REAL,
        p_add_2      IN REAL
    ) RETURN VARCHAR2
    IS
    BEGIN
        RETURN 'add_it REAL: ' || TO_CHAR(p_add_1 + p_add_2,9999.9999);
    END add_it;
    FUNCTION add_it (
        p_add_1      IN DOUBLE PRECISION,
        p_add_2      IN DOUBLE PRECISION
    ) RETURN VARCHAR2
```

(次のページに続く)

(前のページからの続き)

```

IS
BEGIN
    RETURN 'add_it DOUBLE PRECISION: ' || TO_CHAR(p_add_1 +
p_add_2,9999.9999);
END add_it;
BEGIN
    DBMS_OUTPUT.PUT_LINE(add_it (25, 50));
    DBMS_OUTPUT.PUT_LINE(add_it (25.3333, 50.3333));
    DBMS_OUTPUT.PUT_LINE(add_it (TO_NUMBER(25.3333), TO_NUMBER(50.3333)));
    DBMS_OUTPUT.PUT_LINE(add_it (CAST('25.3333' AS REAL), CAST('50.3333' AS
REAL)));
    DBMS_OUTPUT.PUT_LINE(add_it (CAST('25.3333' AS DOUBLE PRECISION),
    CAST('50.3333' AS DOUBLE PRECISION)));
END;

```

以下は、匿名ブロックから表示される出力です。

```

add_it BINARY_INTEGER:      75.0000
add_it NUMBER:             75.6666
add_it NUMBER:             75.6666
add_it REAL:               75.6666
add_it DOUBLE PRECISION:   75.6666

```

2.6.7 Accessing Subprogram Variables

サブプログラムや匿名ブロックなどのブロックで宣言された変数は、相対的な場所に応じて、他のブロックの実行可能セクションまたは例外セクションからアクセスできます。

変数にアクセスすると、ローカル変数で行われるように、SQL ステートメントまたは SPL ステートメント内で変数を参照できるようになります。

注釈: サブプログラムの署名に正式なパラメータが含まれている場合、サブプログラムのローカル変数と同じ方法でこれらにアクセスできます。このセクションでは、サブプログラムの変数に関するすべての説明は、サブプログラムの仮パラメータにも適用されます。

変数のアクセスには、データ型として定義されたものだけでなく、レコード型、コレクション型、カーソルなどの他の要素も含まれます。

変数は、サブプログラムの名前または変数がローカルに宣言されているラベル付き匿名ブロックである、最大 1 つの修飾子によってアクセスできます。

変数を参照する構文は次のとおりです。

```
[<qualifier.>]<variable>
```

指定されている場合、`qualifier` は、`variable` が宣言セクションで宣言されているサブプログラムまたはラベル付き匿名ブロックです（つまり、ローカル変数です）。

注釈: `AdvancedServer` では、2 つの修飾子が許可される状況は 1 つだけです。このシナリオは、次の形式で参照を指定できるパッケージのパブリック変数にアクセスするためのものです。

```
schema_name.package_name.public_variable_name
```

サポートされているパッケージ構文の詳細については、*OracleDevelopersBuilt-InPackageGuide* のデータベース互換性を参照してください。

以下は、変数にアクセスする方法をまとめたものです。

- 変数がローカルに宣言されているブロックが、変数への参照を含むブロックから始まる先祖階層パス内にある限り、変数にアクセスできます。先祖ブロックで宣言されたこのような変数は、**グローバル変数** と呼ばれます。
- 修飾されていない変数への参照が行われた場合、最初の試みはその名前のローカル変数を見つけることです。そのようなローカル変数が存在しない場合、変数の検索は現在のブロックの親などで行われ、祖先の階層を上に進みます。そのような変数が見つからない場合、サブプログラムの呼び出し時にエラーが発生します。
- 修飾変数への参照が行われた場合、前の箇条書きで説明したのと同じ検索プロセスが実行されますが、ローカル変数を含むサブプログラムまたはラベル付き匿名ブロックの最初の一致が検索されます。検索は、一致が見つかるまで祖先階層を上に進みます。そのような一致が見つからない場合、サブプログラムの起動時にエラーが発生します。

次の変数の場所は、変数への参照が行われるブロックに関連してアクセスできません。

- 子孫ブロックで宣言された変数にはアクセスできません。
- 兄弟ブロック、祖先ブロックの兄弟ブロック、または兄弟ブロック内の子孫で宣言された変数にはアクセスできません。

注釈: 変数にアクセスするための `AdvancedServer` プロセスは、`Oracle` データベースと互換性がありません。`Oracle` の場合、任意の数の修飾子を指定でき、検索は、サブプログラムを呼び出す `Oracle` マッチングアルゴリズムと同様の方法で、最初の修飾子の最初の一致に基づきます。

以下の例は、さまざまなブロック内の変数が、修飾子付きと修飾子なしでアクセスされる方法を示しています。コメントアウトされている行は、エラーになる変数にアクセスする試みを示しています。

```

CREATE OR REPLACE PROCEDURE level_0
IS
    v_level_0      VARCHAR2(20) := 'Value from level_0';
    PROCEDURE level_1a
    IS
        v_level_1a VARCHAR2(20) := 'Value from level_1a';
        PROCEDURE level_2a
        IS
            v_level_2a      VARCHAR2(20) := 'Value from level_2a';
            BEGIN
                DBMS_OUTPUT.PUT_LINE('..... BLOCK level_2a');
                DBMS_OUTPUT.PUT_LINE('..... v_level_2a: ' || v_level_2a);
                DBMS_OUTPUT.PUT_LINE('..... v_level_1a: ' || v_level_1a);
                DBMS_OUTPUT.PUT_LINE('..... level_1a.v_level_1a: ' ||
                                     level_1a.v_level_1a);
                DBMS_OUTPUT.PUT_LINE('..... v_level_0: ' || v_level_0);
                DBMS_OUTPUT.PUT_LINE('..... level_0.v_level_0: ' ||
level_0.v_level_0);
                DBMS_OUTPUT.PUT_LINE('..... END BLOCK level_2a');
            END level_2a;
        BEGIN
            DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
            level_2a;
--            DBMS_OUTPUT.PUT_LINE('.... v_level_2a: ' || v_level_2a);
--            Error - Descendent block ----^
--            DBMS_OUTPUT.PUT_LINE('.... level_2a.v_level_2a: ' ||
level_2a.v_level_2a);
--            Error - Descendent block -----^
            DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
        END level_1a;
    PROCEDURE level_1b
    IS
        v_level_1b VARCHAR2(20) := 'Value from level_1b';
        BEGIN
            DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
            DBMS_OUTPUT.PUT_LINE('.... v_level_1b: ' || v_level_1b);
            DBMS_OUTPUT.PUT_LINE('.... v_level_0 : ' || v_level_0);
--            DBMS_OUTPUT.PUT_LINE('.... level_1a.v_level_1a: ' ||
level_1a.v_level_1a);
--            Error - Sibling block -----^
--            DBMS_OUTPUT.PUT_LINE('.... level_2a.v_level_2a: ' ||
level_2a.v_level_2a);
--            Error - Sibling block descendant -----^
            DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
        END level_1b;
    BEGIN
        DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
        DBMS_OUTPUT.PUT_LINE('.. v_level_0: ' || v_level_0);

```

(次のページに続く)

(前のページからの続き)

```

    level_1a;
    level_1b;
    DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;

```

以下は、プロシージャが呼び出されたときの各変数の内容を示す出力です。

```

BEGIN
    level_0;
END;

BLOCK level_0
.. v_level_0: Value from level_0
.. BLOCK level_1a
..... BLOCK level_2a
..... v_level_2a: Value from level_2a
..... v_level_1a: Value from level_1a
..... level_1a.v_level_1a: Value from level_1a
..... v_level_0: Value from level_0
..... level_0.v_level_0: Value from level_0
..... END BLOCK level_2a
.. END BLOCK level_1a
.. BLOCK level_1b
.... v_level_1b: Value from level_1b
.... v_level_0 : Value from level_0
.. END BLOCK level_1b
END BLOCK level_0

```

次の例は、すべてのブロック内のすべての変数が同じ名前を持つ場合の同様のアクセス試行を示しています。

```

CREATE OR REPLACE PROCEDURE level_0
IS
    v_common          VARCHAR2(20) := 'Value from level_0';
    PROCEDURE level_1a
    IS
        v_common      VARCHAR2(20) := 'Value from level_1a';
        PROCEDURE level_2a
        IS
            v_common    VARCHAR2(20) := 'Value from level_2a';
            BEGIN
                DBMS_OUTPUT.PUT_LINE('..... BLOCK level_2a');
                DBMS_OUTPUT.PUT_LINE('..... v_common: ' || v_common);
                DBMS_OUTPUT.PUT_LINE('..... level_2a.v_common: ' ||
level_2a.v_common);
                DBMS_OUTPUT.PUT_LINE('..... level_1a.v_common: ' ||
level_1a.v_common);
                DBMS_OUTPUT.PUT_LINE('..... level_0.v_common: ' ||

```

(次のページに続く)

(前のページからの続き)

```

level_0.v_common);
    DBMS_OUTPUT.PUT_LINE('..... END BLOCK level_2a');
END level_2a;
BEGIN
    DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
    DBMS_OUTPUT.PUT_LINE('.... v_common: ' || v_common);
    DBMS_OUTPUT.PUT_LINE('.... level_0.v_common: ' || level_0.v_common);
    level_2a;
    DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
END level_1a;
PROCEDURE level_1b
IS
    v_common    VARCHAR2(20) := 'Value from level_1b';
BEGIN
    DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
    DBMS_OUTPUT.PUT_LINE('.... v_common: ' || v_common);
    DBMS_OUTPUT.PUT_LINE('.... level_0.v_common : ' || level_0.v_common);
    DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
END level_1b;
BEGIN
    DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
    DBMS_OUTPUT.PUT_LINE('.. v_common: ' || v_common);
    level_1a;
    level_1b;
    DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;

```

以下は、プロシージャが呼び出されたときの各変数の内容を示す出力です。

```

BEGIN
    level_0;
END;

BLOCK level_0
.. v_common: Value from level_0
.. BLOCK level_1a
.... v_common: Value from level_1a
.... level_0.v_common: Value from level_0
..... BLOCK level_2a
..... v_common: Value from level_2a
..... level_2a.v_common: Value from level_2a
..... level_1a.v_common: Value from level_1a
..... level_0.v_common: Value from level_0
..... END BLOCK level_2a
.. END BLOCK level_1a
.. BLOCK level_1b
.... v_common: Value from level_1b

```

(次のページに続く)

(前のページからの続き)

```
.... level_0.v_common : Value from level_0
.. END BLOCK level_1b
END BLOCK level_0
```

前述のように、匿名ブロックのラベルは変数へのアクセスを限定するためにも使用できます。次の例は、ネストされた匿名ブロックのセット内の変数アクセスを示しています。

```
DECLARE
    v_common          VARCHAR2(20) := 'Value from level_0';
BEGIN
    DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
    DBMS_OUTPUT.PUT_LINE('.. v_common: ' || v_common);
    <<level_1a>>
    DECLARE
        v_common       VARCHAR2(20) := 'Value from level_1a';
    BEGIN
        DBMS_OUTPUT.PUT_LINE('... BLOCK level_1a');
        DBMS_OUTPUT.PUT_LINE('.... v_common: ' || v_common);
        <<level_2a>>
        DECLARE
            v_common    VARCHAR2(20) := 'Value from level_2a';
        BEGIN
            DBMS_OUTPUT.PUT_LINE('..... BLOCK level_2a');
            DBMS_OUTPUT.PUT_LINE('..... v_common: ' || v_common);
            DBMS_OUTPUT.PUT_LINE('..... level_1a.v_common: ' ||
level_1a.v_common);
            DBMS_OUTPUT.PUT_LINE('..... END BLOCK level_2a');
        END;
        DBMS_OUTPUT.PUT_LINE('... END BLOCK level_1a');
    END;
    <<level_1b>>
    DECLARE
        v_common       VARCHAR2(20) := 'Value from level_1b';
    BEGIN
        DBMS_OUTPUT.PUT_LINE('... BLOCK level_1b');
        DBMS_OUTPUT.PUT_LINE('.... v_common: ' || v_common);
        DBMS_OUTPUT.PUT_LINE('.... level_1b.v_common: ' ||
level_1b.v_common);
        DBMS_OUTPUT.PUT_LINE('... END BLOCK level_1b');
    END;
    DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END;
```

以下は、匿名ブロックが呼び出されたときの各変数の内容を示す出力です。

```
BLOCK level_0
.. v_common: Value from level_0
```

(次のページに続く)

(前のページからの続き)

```

.. BLOCK level_1a
.... v_common: Value from level_1a
..... BLOCK level_2a
..... v_common: Value from level_2a
..... level_1a.v_common: Value from level_1a
..... END BLOCK level_2a
.. END BLOCK level_1a
.. BLOCK level_1b
.... v_common: Value from level_1b
.... level_1b.v_common: Value from level_1b
.. END BLOCK level_1b
END BLOCK level_0

```

次の例は、オブジェクトタイプメソッド `display_emp` にレコードタイプ `emp_typ` とサブプロシージャ `emp_sal_query` が含まれるオブジェクトタイプです。 `emp_sal_query` にローカルで宣言されたレコード変数 `r_emp` は、親ブロック `display_emp` で宣言されたレコードタイプ `emp_typ` にアクセスできます。

```

CREATE OR REPLACE TYPE emp_pay_obj_typ AS OBJECT
(
    empno          NUMBER(4),
    MEMBER PROCEDURE display_emp(SELF IN OUT emp_pay_obj_typ)
);

CREATE OR REPLACE TYPE BODY emp_pay_obj_typ AS
    MEMBER PROCEDURE display_emp (SELF IN OUT emp_pay_obj_typ)
    IS
        TYPE emp_typ IS RECORD (
            ename          emp.ename%TYPE,
            job            emp.job%TYPE,
            hiredate       emp.hiredate%TYPE,
            sal             emp.sal%TYPE,
            deptno         emp.deptno%TYPE
        );
        PROCEDURE emp_sal_query (
            p_empno        IN emp.empno%TYPE
        )
        IS
            r_emp          emp_typ;
            v_avgsal       emp.sal%TYPE;
        BEGIN
            SELECT ename, job, hiredate, sal, deptno
            INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal,
r_emp.deptno
            FROM emp WHERE empno = p_empno;
            DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
            DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
            DBMS_OUTPUT.PUT_LINE('Job       : ' || r_emp.job);

```

(次のページに続く)

(前のページからの続き)

```

        DBMS_OUTPUT.PUT_LINE('Hire Date   : ' || r_emp.hiredate);
        DBMS_OUTPUT.PUT_LINE('Salary      : ' || r_emp.sal);
        DBMS_OUTPUT.PUT_LINE('Dept #      : ' || r_emp.deptno);

        SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
        IF r_emp.sal > v_avgsal THEN
            DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the '
                                || 'department average of ' || v_avgsal);
        ELSE
            DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the
                                '
                                || 'department average of ' || v_avgsal);
        END IF;
    END;
BEGIN
    emp_sal_query(SELF.empno);
END;
END;
```

以下は、オブジェクトタイプのインスタンスが作成され、プロシージャ `display_emp` が呼び出されたときに表示される出力です。

```

DECLARE
    v_emp          EMP_PAY_OBJ_TYP;
BEGIN
    v_emp := emp_pay_obj_typ(7900);
    v_emp.display_emp;
END;
```

```

Employee # : 7900
Name       : JAMES
Job        : CLERK
Hire Date  : 03-DEC-81 00:00:00
Salary     : 950.00
Dept #     : 30
Employee's salary does not exceed the department average of 1566.67
```

次の例は、3つのレベルのサブプロシージャを持つパッケージです。上位レベルのプロシージャで宣言されたレコードタイプ、コレクションタイプ、カーソルタイプには、子孫サブプロシージャからアクセスできます。

```

CREATE OR REPLACE PACKAGE emp_dept_pkg
IS
    PROCEDURE display_emp (
        p_deptno          NUMBER
    );
END;
```

(次のページに続く)

(前のページからの続き)

```

CREATE OR REPLACE PACKAGE BODY emp_dept_pkg
IS
    PROCEDURE display_emp (
        p_deptno      NUMBER
    )
    IS
        TYPE emp_rec_typ IS RECORD (
            empno      emp.empno%TYPE,
            ename      emp.ename%TYPE
        );
        TYPE emp_arr_typ IS TABLE OF emp_rec_typ INDEX BY BINARY_INTEGER;
        TYPE emp_cur_type IS REF CURSOR RETURN emp_rec_typ;
        PROCEDURE emp_by_dept (
            p_deptno      emp.deptno%TYPE
        )
        IS
            emp_arr      emp_arr_typ;
            emp_refcur    emp_cur_type;
            i              BINARY_INTEGER := 0;
            PROCEDURE display_emp_arr
            IS
                BEGIN
                    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
                    DBMS_OUTPUT.PUT_LINE('-----      -----');
                    FOR j IN emp_arr.FIRST .. emp_arr.LAST LOOP
                        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '      ' ||
                                                emp_arr(j).ename);
                    END LOOP;
                END display_emp_arr;
            BEGIN
                OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno =
p_deptno;
                LOOP
                    i := i + 1;
                    FETCH emp_refcur INTO emp_arr(i).empno, emp_arr(i).ename;
                    EXIT WHEN emp_refcur%NOTFOUND;
                END LOOP;
                CLOSE emp_refcur;
                display_emp_arr;
            END emp_by_dept;
        BEGIN
            emp_by_dept(p_deptno);
        END;
    END;
END;

```

以下は、最上位のパッケージプロシージャが呼び出されたときに表示される出力です。

```
BEGIN
    emp_dept_pkg.display_emp(20);
END;
```

EMPNO	ENAME
-----	-----
7369	SMITH
7566	JONES
7788	SCOTT
7876	ADAMS
7902	FORD

2.7 Compilation Errors in Procedures and Functions

AdvancedServer パーサーは、プロシージャまたは関数をコンパイルするときに、CREATE ステートメントとプログラム本体（AS キーワードに続くプログラムの部分）の両方が、SPL および SQL 構文の文法規則に準拠していることを確認します。デフォルトでは、パーサーがエラーを検出した場合、サーバーはコンパイルプロセスを終了します。パーサーは式の構文エラーを検出しますが、セマンティックエラー（つまり、存在しない列、テーブル、関数を参照する式、または不正な型の値）は検出しません。

spl.max_error_count は、SPL コードで指定された数のエラーが発生した場合、または SQL コードでエラーが発生した場合、解析を停止するようサーバーに指示します。spl.max_error_count のデフォルト値は 10 です。最大値は 1000 です。spl.max_error_count を 1 の値に設定すると、SPL または SQL コードで最初のエラーが発生したときに解析を停止するようサーバーに指示します。

SET コマンドを使用して、現在のセッションの spl.max_error_count の値を指定できます。構文は次のとおりです。

```
SET spl.max_error_count = <number_of_errors>
```

number_of_errors は、サーバーがコンパイルプロセスを停止する前に発生する可能性のある SPL エラーの数を指定します。例:

```
SET spl.max_error_count = 6
```

この例では、サーバーに、発生した最初の 5 つの SPL エラーを過ぎて続行するように指示します。サーバーが 6 番目のエラーを検出すると、検証を停止し、6 つの詳細なエラーメッセージと 1 つのエラーの概要を出力します。

新しいコードを開発するとき、または別のソースから既存のコードをインポートするときに時間を節約するために、spl.max_error_count 構成パラメーターを比較的多くのエラーに設定することができます。

プログラム本体の SPL コードでエラーが発生しても解析を続行するようにサーバーに指示し、パーサーが SQL コードのセグメントでエラーを検出した場合、後続の SPL または SQL コードでエラーが発生する可能性があることに注意してくださいエラーのある SQL コード。たとえば、次の関数では 2 つのエラーが発生します。

```
CREATE FUNCTION computeBonus(baseSalary number) RETURN number AS
BEGIN

    bonus := baseSalary * 1.10;
    total := bonus + 100;

    RETURN bonus;
END;

ERROR:  "bonus" is not a known variable
LINE 4:      bonus := baseSalary * 1.10;
           ^
ERROR:  "total" is not a known variable
```

(次のページに続く)

(前のページからの続き)

```
LINE 5:      total := bonus + 100;
          ^
ERROR:  compilation of SPL function/procedure "computebonus" failed due to 2
errors
```

次の例は、前の例に SELECT ステートメントを追加します。SELECT ステートメントのエラーは、後続のその他のエラーをマスクします。

```
CREATE FUNCTION computeBonus(employeeName number) RETURN number AS
BEGIN
    SELECT salary INTO baseSalary FROM emp
        WHERE ename = employeeName;

    bonus := baseSalary * 1.10;
    total := bonus + 100;

    RETURN bonus;

END;

ERROR:  "basesalary" is not a known variable
LINE 3:      SELECT salary INTO baseSalary FROM emp WHERE ename = emp...
```

2.8 Program Security

SPL プログラムを実行するユーザーと、プログラムを実行する特定のユーザーに対して SPL プログラムがアクセスできるデータベースオブジェクトに対するセキュリティは、以下によって制御されます。

- プログラムを実行する特権。
- プログラムがアクセスを試みるデータベースオブジェクト（他の SPL プログラムを含む）に付与された特権。
- プログラムが定義者の権利で定義されているか、実行者の権利で定義されているか。

これらの側面については、次のセクションで説明します。

2.8.1 EXECUTE Privilege

SPL プログラム（関数、プロシージャ、またはパッケージ）は、次のいずれかに該当する場合にのみ実行を開始できます。

- 現在のユーザーはスーパーユーザー、または
- 現在のユーザーに SPL プログラムの EXECUTE 特権が付与されている、または
- 現在のユーザーは、そのような特権を持つグループのメンバーであるため、SPL プログラムの EXECUTE 特権を継承します。または
- PUBLIC グループに EXECUTE 特権が付与されました。

AdvancedServer で SPL プログラムが作成されるたびに、デフォルトで EXECUTE 特権が PUBLIC グループに自動的に付与されるため、どのユーザーでもすぐにプログラムを実行できます。

このデフォルトの特権は、REVOKE EXECUTE コマンドを使用して削除できます。次に例を示します。

```
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
```

その後、プログラムの明示的な EXECUTE 特権を個々のユーザーまたはグループに付与できます。

```
GRANT EXECUTE ON PROCEDURE list_emp TO john;
```

これで、ユーザー john は list_emp プログラムを実行できます。このセクションの最初にリストされている条件のいずれにも当てはまらない他のユーザーはできません。

プログラムの実行が開始されると、セキュリティの次の側面は、プログラムが以下を含むデータベースオブジェクトに対してアクションを実行しようとした場合に特権チェックが行われることです。

- テーブルまたはビューのデータの読み取りまたは変更。

- テーブル、ビュー、インデックス、シーケンスなどのデータベースオブジェクトの作成、変更、または削除。
- シーケンスから現在または次の値を取得します。
- 別のプログラム（関数、プロシージャ、またはパッケージ）の呼び出し。

このような各アクションは、ユーザーに対して許可または禁止されているデータベースオブジェクトに対する特権によって保護できます。

データベースには、同じ名前の同じタイプのオブジェクトを複数持つことができますが、そのような各オブジェクトはデータベース内の異なるスキーマに属することに注意してください。この場合、どのオブジェクトが SPL プログラムによって参照されていますか？これは次のセクションのトピックです。

2.8.2 Database Object Name Resolution

SPL プログラム内のデータベースオブジェクトは、修飾名または非修飾名で参照できます。修飾名は `schema.name` の形式で、`schema` は識別子 `name` を持つデータベースオブジェクトが存在するスキーマの名前です。非修飾名には `schema.` 部分はありません。修飾名への参照が行われた場合、どのデータベースオブジェクトが意図されているかに関しては明確にあいまいさはありません。指定されたスキーマに存在するか、存在しないかのどちらかです。

ただし、非修飾名のオブジェクトを見つけるには、現在のユーザーの検索パスを使用する必要があります。ユーザーがセッションの現在のユーザーになると、デフォルトの検索パスが常にそのユーザーに関連付けられます。検索パスは、非修飾データベースオブジェクト参照を見つけるために左から右の順序で検索されるスキーマのリストで構成されます。検索パス内のどのスキーマでもオブジェクトが見つからない場合、オブジェクトは存在しないと見なされます。デフォルトの検索パスは、`SHOW search_path` コマンドを使用して `PSQL` に表示できます。

```
edb=# SHOW search_path;
      search_path
-----
 "$user", public
(1 row)
```

上記の検索パスの `$user` は現在のユーザーを参照する一般的なプレースホルダーです。したがって、上記のセッションの現在のユーザーが `enterprisedb` の場合、非修飾データベースオブジェクトは次のスキーマでこの順序で検索されます。 `enterprisedb`、次に `public`。

修飾されていない名前が検索パスで解決されると、現在のユーザーがその特定のオブジェクトに対して目的のアクションを実行するための適切な特権を持っているかどうかを判断できます。

注釈：検索パスの概念は、Oracle データベースと互換性がありません。修飾されていない参照の場合、Oracle は単に指定されたデータベースオブジェクトの現在のユーザーのスキーマを検索します。また、Oracle では、ユーザーとそのスキーマは同じエンティティであり、AdvancedServer では、ユーザーとスキーマは 2 つの別個のオブジェ

クトであることに注意することも重要です。

2.8.3 Database Object Privileges

SPL プログラムの実行が開始されると、プログラム内からデータベースオブジェクトにアクセスしようとする、現在のユーザーが参照オブジェクトに対して意図したアクションを実行する権限を持っていることを確認するチェックが行われます。データベースオブジェクトの特権は、それぞれ GRANT および REVOKE コマンドを使用して付与および削除されます。現在のユーザーがデータベースオブジェクトへの不正アクセスを試みると、プログラムは例外をスローします。例外処理の詳細については、[ExceptionHandling](#) を参照してください。

2.8.4 Definer' s vs. Invokers Rights

SPL プログラムが実行を開始しようとする、どのユーザーがこのプロセスに関連付けられるかが決定されます。このユーザーは、**現在のユーザー** と呼ばれます。現在のユーザーのデータベースオブジェクト権限を使用して、プログラムで参照されているデータベースオブジェクトへのアクセスを許可するかどうかを決定します。プログラムが呼び出されたときに有効な現在の一般的な検索パスは、非修飾オブジェクト参照を解決するために使用されます。

現在のユーザーの選択は、SPL プログラムが定義者の権利で作成されたか、実行者の権利で作成されたかによって影響を受けます。AUTHID 節がその選択を決定します。AUTHID DEFINER という句の出現は、プログラム定義者の権利を与えます。これは、AUTHID 句が省略された場合のデフォルトでもあります。AUTHID CURRENT_USER という句を使用すると、プログラムの実行者の権利が付与されます。2つの違いは次のように要約されます。

- プログラムに「定義者の権限」がある場合、プログラムの実行が開始されると、プログラムの所有者が現在のユーザーになります。プログラム所有者のデータベースオブジェクト権限を使用して、参照オブジェクトへのアクセスが許可されているかどうかを判断します。定義者の権利プログラムでは、どのユーザーが実際にプログラムを呼び出したかは関係ありません。
- プログラムに*呼び出し側の権限*がある場合、プログラムが呼び出された時点での現在のユーザーは、プログラムの実行中は現在のユーザーのままです（ただし、呼び出されたサブプログラム内である必要はありません。次の箇条書きを参照）。実行者の権利プログラムが呼び出されると、現在のユーザーは通常、セッションを開始した（つまり、データベース接続を行った）ユーザーですが、SETROLE コマンドを使用してセッションを開始した後に現在のユーザーを変更することもできます実行者の権利プログラムでは、どのユーザーが実際にプログラムを所有しているかは関係ありません。

前の定義から、次の観察を行うことができます。

- 定義者の権利プログラムが定義者の権利プログラムを呼び出すと、現在のユーザーは、呼び出されたプログラムの実行中に、呼び出し元プログラムの所有者から呼び出されたプログラムの所有者に変わります。

- 定義者の権利プログラムが呼び出し元の権利プログラムを呼び出す場合、呼び出し元プログラムと呼び出し先プログラムの両方の実行中、呼び出し元プログラムの所有者は現在のユーザーのままです。
- 呼び出し元の権利プログラムが呼び出し元の権利プログラムを呼び出す場合、呼び出し元プログラムの現在のユーザーは、呼び出されたプログラムの実行中は現在のユーザーのままです。
- 実行者の権利プログラムが定義者の権利プログラムを呼び出す場合、現在のユーザーは、呼び出されたプログラムの実行中に定義者の権利プログラムの所有者に切り替えます。

上記のケースで、呼び出されたプログラムが別のプログラムを呼び出す場合、同じ原則が適用されます。

2.8.5 Security Example

次の例では、スキーマ内のサンプルアプリケーション `hr_mgr` 全体のコピーを所有する 2 人のユーザー – `hr_mgr` とともに新しいデータベースが作成されます。 `sales_mgr` という名前のスキーマを所有する `sales_mgr` は、営業に従事する従業員のみを含む `emp` テーブルのみのコピーを持ちます。

この例では、プロシージャ `list_emp`、関数 `hire_clerk`、およびパッケージ `emp_admin` が使用されます。サンプルアプリケーションのインストール時に付与されるデフォルトの特権はすべて削除され、この例でより安全な環境を提供するために明示的に再付与されます。

プログラム `list_emp` および `hire_clerk` は、定義者の権限のデフォルトから起動者の権限に変更されます。`sales_mgr` がこれらのプログラムを実行すると、`sales_mgr` の検索パスと権限が名前解決と承認チェックに使用されるため、`sales_mgr` のスキーマの `emp` テーブルに基づいて動作することが示されます。

`emp_admin` パッケージの `get_dept_name` と `hire_emp` のプログラムは、`sales_mgr` によって実行されます。この場合、「10」は定義者の権限を使用している `emp_admin` パッケージの所有者であるため、「10」のスキーマの `dept` テーブルと `emp` テーブルにアクセスします。デフォルトの検索パスは `$user` プレースホルダーで有効であるため、ユーザーに一致するスキーマ（この場合は「10」）を使用してテーブルが検索されます。

Step 1 – Create Database and Users

ユーザー `enterprisedb` として、`hr` データベースを作成します。

```
CREATE DATABASE hr;
```

`hr` データベースに切り替えて、ユーザーを作成します。

```
\c hr enterprisedb
CREATE USER hr_mgr IDENTIFIED BY password;
CREATE USER sales_mgr IDENTIFIED BY password;
```

Step 2 – Create the Sample Application

`hr_mgr` のスキーマに、`hr_mgr` が所有するサンプルアプリケーション全体を作成します。

```
\c - hr_mgr
\i /usr/edb/as11/share/edb-sample.sql

BEGIN
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE VIEW
CREATE SEQUENCE
      .
      .
      .
CREATE PACKAGE
CREATE PACKAGE BODY
COMMIT
```

Step 3 – Create the emp Table in Schema sales_mgr

sales_mgr のスキーマで sales_mgr が所有する emp テーブルのサブセットを作成します。

```
\c - hr_mgr
GRANT USAGE ON SCHEMA hr_mgr TO sales_mgr;
\c - sales_mgr
CREATE TABLE emp AS SELECT * FROM hr_mgr.emp WHERE job = 'SALESMAN';
```

上記の例では、hr_mgr のスキーマに sales_mgr から hr_mgr の emp テーブルのコピーを作成できる GRANT USAGE ON SCHEMA コマンドが与えられています。Oracle にはユーザーとは異なるスキーマの概念がないため、この手順は AdvancedServer で必要であり、Oracle データベースと互換性がありません。

Step 4 – Remove Default Privileges

すべての特権を削除して、必要な最小限の特権を後で示します。

```
\c - hr_mgr
REVOKE USAGE ON SCHEMA hr_mgr FROM sales_mgr;
REVOKE ALL ON dept FROM PUBLIC;
REVOKE ALL ON emp FROM PUBLIC;
REVOKE ALL ON next_empno FROM PUBLIC;
REVOKE EXECUTE ON FUNCTION new_empno() FROM PUBLIC;
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
REVOKE EXECUTE ON FUNCTION hire_clerk(VARCHAR2,NUMBER) FROM PUBLIC;
REVOKE EXECUTE ON PACKAGE emp_admin FROM PUBLIC;
```

Step 5 – Change list_emp to Invoker's Rights

ユーザー hr_mgr として接続しているときに、list_emp プログラムに AUTHID CURRENT_USER 句を追加し、AdvancedServer に保存し直します。この手順を実行するときは、hr_mgr としてログオンしていることを確認してください。ログオンしていないと、変更されたプログラムが hr_mgr のスキーマではなく public スキーマ

マに表示される可能性があります。

```
CREATE OR REPLACE PROCEDURE list_emp
AUTHID CURRENT_USER
IS
    v_empno      NUMBER(4);
    v_ename      VARCHAR2(10);
    CURSOR emp_cur IS
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
    DBMS_OUTPUT.PUT_LINE('-----      -----');
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '      ' || v_ename);
    END LOOP;
    CLOSE emp_cur;
END;
```

Step 6 – Change hire_clerk to Invoker's Rights and Qualify Call to new_empno

ユーザー hr_mgr として接続しているときに、AUTHID CURRENT_USER 句を hire_clerk プログラムに追加します。

また、BEGIN ステートメントの後、new_empno 関数への hire_clerk 関数呼び出しが hr_mgr スキーマに解決されるように、参照 new_empno を hr_mgr.new_empno に完全修飾します。

プログラムを再保存するときは、hr_mgr としてログオンしていることを確認してください。そうしないと、変更されたプログラムが hr_mgr のスキーマではなく public スキーマに表示される可能性があります。

```
CREATE OR REPLACE FUNCTION hire_clerk (
    p_ename      VARCHAR2,
    p_deptno     NUMBER
) RETURN NUMBER
AUTHID CURRENT_USER
IS
    v_empno      NUMBER(4);
    v_ename      VARCHAR2(10);
    v_job        VARCHAR2(9);
    v_mgr        NUMBER(4);
    v_hiredate    DATE;
    v_sal        NUMBER(7,2);
    v_comm       NUMBER(7,2);
    v_deptno     NUMBER(2);
BEGIN
    v_empno := hr_mgr.new_empno;
```

(次のページに続く)

(前のページからの続き)

```

INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
    TRUNC(SYSDATE), 950.00, NULL, p_deptno);
SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO
    v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
    FROM emp WHERE empno = v_empno;
DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
DBMS_OUTPUT.PUT_LINE('Name      : ' || v_ename);
DBMS_OUTPUT.PUT_LINE('Job       : ' || v_job);
DBMS_OUTPUT.PUT_LINE('Manager   : ' || v_mgr);
DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
DBMS_OUTPUT.PUT_LINE('Commission : ' || v_comm);
RETURN v_empno;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
        RETURN -1;
END;
```

Step 7 – Grant Required Privileges

ユーザー `hr_mgr` として接続している間に、`sales_mgr` が `list_emp` 手順、`hire_clerk` 機能、および `emp_admin` パッケージを実行できるように、必要な特権を付与します。`sales_mgr` にアクセスできる唯一のデータオブジェクト `sales_mgr` は、`sales_mgr` スキーマの `emp` テーブルです。`sales_mgr` には、`hr_mgr` スキーマ内のテーブルに対する権限がありません。

```

GRANT USAGE ON SCHEMA hr_mgr TO sales_mgr;
GRANT EXECUTE ON PROCEDURE list_emp TO sales_mgr;
GRANT EXECUTE ON FUNCTION hire_clerk(VARCHAR2,NUMBER) TO sales_mgr;
GRANT EXECUTE ON FUNCTION new_empno() TO sales_mgr;
GRANT EXECUTE ON PACKAGE emp_admin TO sales_mgr;
```

Step 8 – Run Programs `list_emp` and `hire_clerk`

ユーザー `sales_mgr` として接続し、次の匿名ブロックを実行します。

```

\c - sales_mgr
DECLARE
    v_empno      NUMBER(4);
BEGIN
    hr_mgr.list_emp;
    DBMS_OUTPUT.PUT_LINE('*** Adding new employee ***');
    v_empno := hr_mgr.hire_clerk('JONES',40);
```

(次のページに続く)

(前のページからの続き)

```

DBMS_OUTPUT.PUT_LINE('*** After new employee added ***');
hr_mgr.list_emp;
END;

```

```

EMPNO      ENAME
-----
7499      ALLEN
7521      WARD
7654      MARTIN
7844      TURNER
*** Adding new employee ***
Department : 40
Employee No: 8000
Name       : JONES
Job        : CLERK
Manager    : 7782
Hire Date  : 08-NOV-07 00:00:00
Salary     : 950.00
*** After new employee added ***
EMPNO      ENAME
-----
7499      ALLEN
7521      WARD
7654      MARTIN
7844      TURNER
8000      JONES

```

無名ブロックのプログラムがアクセスするテーブルとシーケンスを次の図に示します。灰色の楕円は、sales_mgr と hr_mgr のスキーマを表します。各プログラム実行中の現在のユーザーは、括弧内に太字の赤いフォントで表示されます。

sales_mgr の emp テーブルから選択すると、このテーブルで更新が行われたことがわかります。

```

SELECT empno, ename, hiredate, sal, deptno,
hr_mgr.emp_admin.get_dept_name(deptno) FROM sales_mgr.emp;

```

empno	ename	hiredate	sal	deptno	get_dept_name
7499	ALLEN	20-FEB-81 00:00:00	1600.00	30	SALES
7521	WARD	22-FEB-81 00:00:00	1250.00	30	SALES
7654	MARTIN	28-SEP-81 00:00:00	1250.00	30	SALES
7844	TURNER	08-SEP-81 00:00:00	1500.00	30	SALES
8000	JONES	08-NOV-07 00:00:00	950.00	40	OPERATIONS

(5 rows)

次の図は、SELECT コマンドが sales_mgr スキーマの emp テーブルを参照しますが、emp_admin パッケージの get_dept_name 関数によって参照される「11」テーブルは「12」スキーマからのものであることを示して

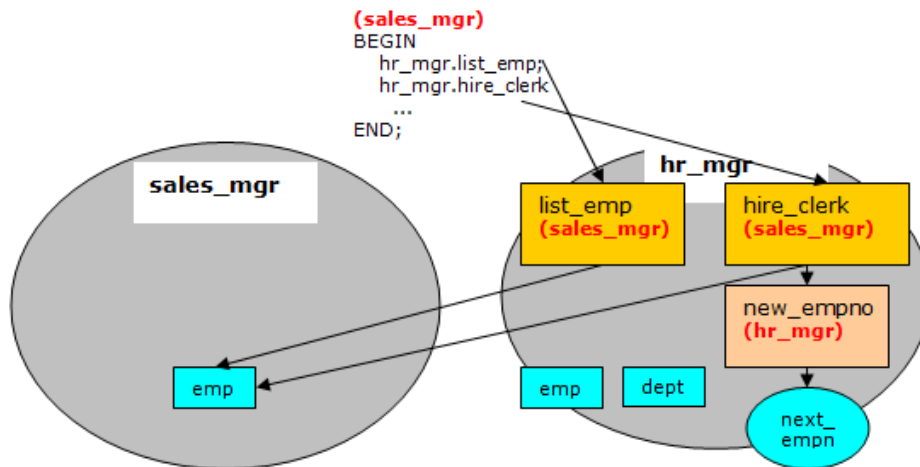


図 1 呼び出し者の権利プログラム

います emp_admin パッケージには定義者の権利があり、「12」が所有しています。\$user プレースホルダーを使用したデフォルトの検索パス設定は、ユーザー「12」によるアクセスを「12」スキーマの「11」テーブルに解決します。

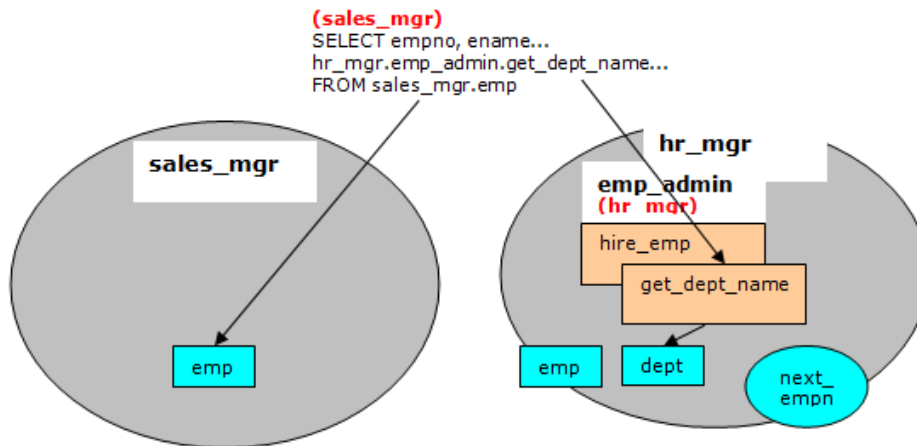


図 2 定義者の権利パッケージ

Step 9 – Run Program hire_emp in the emp_admin Package

ユーザー sales_mgr として接続した状態で、emp_admin パッケージの hire_emp 手順を実行します。

```
EXEC hr_mgr.emp_admin.hire_emp(9001,
'ALICE', 'SALESMAN', 8000, TRUNC(SYSDATE), 1000, 7369, 40);
```

この図は、hr_mgr のオブジェクト権限が使用されているため、emp_admin 定義者の権限パッケージの

hire_emp プロシージャが hr_mgr に属する emp テーブルを更新し、デフォルトの検索パス設定が \$user であることを示していますプレースホルダーは hr_mgr のスキーマに解決します。

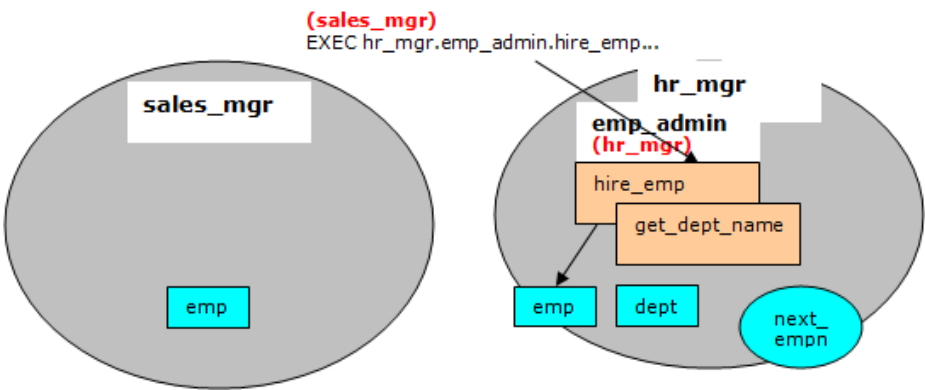


図 3 定義者の権利パッケージ

次に、ユーザー hr_mgr として接続します。次の SELECT コマンドは、emp_admin パッケージに定義者の権利があり、hr_mgr が emp_admin の所有者であるため、hr_mgr の emp テーブルに新しい従業員が追加されたことを確認します。

```
\c - hr_mgr
SELECT empno, ename, hiredate, sal, deptno,
hr_mgr.emp_admin.get_dept_name(deptno) FROM hr_mgr.emp;
```

empno	ename	hiredate	sal	deptno	get_dept_name
7369	SMITH	17-DEC-80 00:00:00	800.00	20	RESEARCH
7499	ALLEN	20-FEB-81 00:00:00	1600.00	30	SALES
7521	WARD	22-FEB-81 00:00:00	1250.00	30	SALES
7566	JONES	02-APR-81 00:00:00	2975.00	20	RESEARCH
7654	MARTIN	28-SEP-81 00:00:00	1250.00	30	SALES
7698	BLAKE	01-MAY-81 00:00:00	2850.00	30	SALES
7782	CLARK	09-JUN-81 00:00:00	2450.00	10	ACCOUNTING
7788	SCOTT	19-APR-87 00:00:00	3000.00	20	RESEARCH
7839	KING	17-NOV-81 00:00:00	5000.00	10	ACCOUNTING
7844	TURNER	08-SEP-81 00:00:00	1500.00	30	SALES
7876	ADAMS	23-MAY-87 00:00:00	1100.00	20	RESEARCH
7900	JAMES	03-DEC-81 00:00:00	950.00	30	SALES
7902	FORD	03-DEC-81 00:00:00	3000.00	20	RESEARCH
7934	MILLER	23-JAN-82 00:00:00	1300.00	10	ACCOUNTING
9001	ALICE	08-NOV-07 00:00:00	8000.00	40	OPERATIONS

(15 rows)

第 3 章

Variable Declarations

SPL はブロック構造化言語です。ブロックに表示できる最初のセクションは宣言です。宣言には、ブロックに含まれる SPL ステートメントで使える変数、カーソル、およびその他のタイプの定義が含まれます。

3.1 Declaring a Variable

通常、ブロックで使用されるすべての変数は、ブロックの宣言セクションで宣言する必要があります。変数宣言は、変数に割り当てられた名前とそのデータ型で構成されます。オプションで、変数は変数宣言でデフォルト値に初期化できます。

変数宣言の一般的な構文は次のとおりです。

```
<name> <type> [ { := | DEFAULT } { <expression> | NULL } ];
```

name は変数に割り当てられた識別子です。

type は変数に割り当てられたデータ型です。

[:= expression] は、与えられた場合、ブロックに入るときに変数に割り当てられる初期値を指定します。句が指定されていない場合、変数は SQL の NULL 値に初期化されます。

デフォルト値は、ブロックに入るたびに評価されます。したがって、たとえば、DATE 型の変数に SYSDATE を割り当てると、変数には、プロシージャまたは関数がプリコンパイルされた時刻ではなく、現在の呼び出しの時刻が設定されます。

次の手順は、文字列式と数値式で構成されるデフォルトを利用するいくつかの変数宣言を示しています。

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno          NUMBER
)
IS
    todays_date       DATE := SYSDATE;
```

(次のページに続く)

(前のページからの続き)

```

    rpt_title      VARCHAR2(60) := 'Report For Department # ' || p_deptno
                    || ' on ' || todays_date;
    base_sal       INTEGER := 35525;
    base_comm_rate NUMBER := 1.33333;
    base_annual    NUMBER := ROUND(base_sal * base_comm_rate, 2);
BEGIN
    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```

上記の手順の次の出力は、変数宣言のデフォルト値が実際に変数に割り当てられていることを示しています。

```
EXEC dept_salary_rpt(20);

Report For Department # 20 on 10-JUL-07 16:44:45
Base Annual Salary: 47366.55
```

3.2 User-Defined Record Types and Record Variables

レコード宣言<using_%_row_type_in_record_declarations>‘での% ROWTYPE の使用に示すように、レコードは ‘%ROWTYPE’ 属性を使用してテーブル定義に基づいて宣言できます。このセクションでは、特定のテーブル定義に関連付けられていない新しいレコード構造を定義する方法について説明します。

TYPE IS RECORD ステートメントは、レコードタイプの定義を作成するために使用されます。レコードタイプは、1 つ以上の識別子とそれに対応するデータタイプで構成されるレコードの定義です。レコードタイプを単独で使用してデータを操作することはできません。

TYPE IS RECORD ステートメントの構文は次のとおりです。

```
TYPE <rec_type> IS RECORD ( <fields> )
```

fields は、次の形式の 1 つ以上のフィールド定義のコンマ区切りリストです。

```
<field_name data_type> [NOT NULL] [{:= | DEFAULT} <default_value>]
```

どこで:

rec_type

rec_type は、レコードタイプに割り当てられた識別子です。

field_name

field_name は、レコードタイプのフィールドに割り当てられた識別子です。

data_type

data_type は field_name のデータ型を指定します。

DEFAULT default_value

DEFAULT 句は、対応するフィールドにデフォルトのデータ値を割り当てます。デフォルト式のデータ型は、列のデータ型と一致する必要があります。デフォルトが指定されていない場合、デフォルトは NULL です。

レコード変数*または単に置く*レコード は、レコードタイプのインスタンスです。レコードは、レコードタイプから宣言されます。フィールド名やタイプなどのレコードのプロパティは、レコードタイプから継承されます。

以下は、レコード宣言の構文です。

```
<record rectype>
```

record は、レコード変数に割り当てられた識別子です。rectype は、以前に定義されたレコードタイプの識別子です。宣言すると、レコードを使用してデータを保持できます。

レコード内のフィールドを参照するには、ドット表記法が使用されます。

```
<record.field>
```

record は以前に宣言されたレコード変数であり、field は record が定義されているレコードタイプに属するフィールドの識別子です。

emp_sal_query が再び変更されました-今回はユーザー定義のレコードタイプとレコード変数を使用しています。

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno          IN emp.empno%TYPE
)
IS
    TYPE emp_typ IS RECORD (
        ename        emp.ename%TYPE,
        job           emp.job%TYPE,
        hiredate      emp.hiredate%TYPE,
        sal           emp.sal%TYPE,
        deptno        emp.deptno%TYPE
    );
    r_emp            emp_typ;
    v_avgsal         emp.sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
```

(次のページに続く)

(前のページからの続き)

```
DBMS_OUTPUT.PUT_LINE('Job          : ' || r_emp.job);
DBMS_OUTPUT.PUT_LINE('Hire Date   : ' || r_emp.hiredate);
DBMS_OUTPUT.PUT_LINE('Salary      : ' || r_emp.sal);
DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);

SELECT AVG(sal) INTO v_avgsal
  FROM emp WHERE deptno = r_emp.deptno;
IF r_emp.sal > v_avgsal THEN
  DBMS_OUTPUT.PUT_LINE('Employee's salary is more than the '
    || 'department average of ' || v_avgsal);
ELSE
  DBMS_OUTPUT.PUT_LINE('Employee's salary does not exceed the '
    || 'department average of ' || v_avgsal);
END IF;
END;
```

データ型名を指定する代わりに、%TYPE 属性をレコード型定義のフィールドデータ型に使用できることに注意してください。

以下は、このストアードプロシージャの実行からの出力です。

```
EXEC emp_sal_query(7698);

Employee # : 7698
Name       : BLAKE
Job        : MANAGER
Hire Date  : 01-MAY-81 00:00:00
Salary     : 2850.00
Dept #     : 30
Employee's salary is more than the department average of 1566.67
```

第 4 章

Basic Statements

このセクションでは、SPL プログラムで使用できるプログラミングステートメントの説明を開始します。

4.1 Assignment

割り当てステートメントは、割り当ての左側に指定されたモード OUT または IN OUT の変数または仮パラメーター `:=` を、割り当ての右側に指定された評価された式に設定します。

```
<variable> := <expression>;
```

`variable` は、以前に宣言された変数、OUT の仮パラメーター、または IN OUT の仮パラメーターの識別子です。

`expression` は、単一の値を生成する式です。式によって生成される値には、`variable` のデータ型と互換性のあるデータ型が必要です。

次の例は、プロシージャの実行可能セクションでの割り当てステートメントの一般的な使用法を示しています。

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno      NUMBER
)
IS
    todays_date    DATE;
    rpt_title      VARCHAR2(60);
    base_sal       INTEGER;
    base_comm_rate  NUMBER;
    base_annual    NUMBER;
BEGIN
    todays_date := SYSDATE;
    rpt_title := 'Report For Department # ' || p_deptno || ' on '
        || todays_date;
    base_sal := 35525;
    base_comm_rate := 1.33333;
```

(次のページに続く)

(前のページからの続き)

```

base_annual := ROUND(base_sal * base_comm_rate, 2);

DBMS_OUTPUT.PUT_LINE(rpt_title);
DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```

4.2 DELETE

DELETE コマンド (SQL 言語で使用可能) は、SPL プログラムでも使用できます。

SPL 言語の式は、SQL の DELETE コマンドで式を使用できる場所であればどこでも使用できます。したがって、SPL 変数およびパラメーターを使用して、削除操作に値を提供できます。

```

CREATE OR REPLACE PROCEDURE emp_delete (
    p_empno          IN emp.empno%TYPE
)
IS
BEGIN
    DELETE FROM emp WHERE empno = p_empno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || p_empno);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

SQL%FOUND 条件式は、行が削除されると TRUE を返し、そうでない場合は FALSE を返します。SQL%FOUND および他の同様の式の説明については、:ref:「結果ステータスの取得<obtaining_the_result_status>」を参照してください。

以下は、この手順を使用した従業員の削除を示しています。

```

EXEC emp_delete(9503);

Deleted Employee # : 9503

SELECT * FROM emp WHERE empno = 9503;

 empno |  ename  | job   | mgr | hiredate | sal | comm | deptno
-----+-----+-----+-----+-----+-----+-----+-----
(0 rows)
```

注釈: DELETE コマンドは FORALL ステートメントに含めることができます。FORALL ステートメントを使用す

ると、1つの DELETE コマンドで 1つ以上のコレクションで指定された値から複数の行を削除できます。FORALL ステートメントの詳細については、:ref:「FORALL ステートメントの使用<using_the_forall_statement>」を参照してください。

4.3 INSERT

SQL 言語で使用可能な INSERT コマンドは、SPL プログラムでも使用できます。

SPL 言語の式は、SQL の INSERT コマンドで式を使用できる場所であればどこでも使用できます。したがって、SPL 変数およびパラメーターを使用して、挿入操作に値を提供できます。

以下は、呼び出し側プログラムから渡されたデータを使用して、新しい従業員の挿入を実行する手順の例です。

```
CREATE OR REPLACE PROCEDURE emp_insert (
    p_empno      IN emp.empno%TYPE,
    p_ename      IN emp.ename%TYPE,
    p_job        IN emp.job%TYPE,
    p_mgr        IN emp.mgr%TYPE,
    p_hiredate    IN emp.hiredate%TYPE,
    p_sal        IN emp.sal%TYPE,
    p_comm       IN emp.comm%TYPE,
    p_deptno     IN emp.deptno%TYPE
)
IS
BEGIN
    INSERT INTO emp VALUES (
        p_empno,
        p_ename,
        p_job,
        p_mgr,
        p_hiredate,
        p_sal,
        p_comm,
        p_deptno);

    DBMS_OUTPUT.PUT_LINE('Added employee...');
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name      : ' || p_ename);
    DBMS_OUTPUT.PUT_LINE('Job      : ' || p_job);
    DBMS_OUTPUT.PUT_LINE('Manager  : ' || p_mgr);
    DBMS_OUTPUT.PUT_LINE('Hire Date : ' || p_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary   : ' || p_sal);
    DBMS_OUTPUT.PUT_LINE('Commission : ' || p_comm);
    DBMS_OUTPUT.PUT_LINE('Dept #   : ' || p_deptno);
    DBMS_OUTPUT.PUT_LINE('-----');
```

(次のページに続く)

(前のページからの続き)

```

EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('OTHERS exception on INSERT of employee # '
                               || p_empno);
        DBMS_OUTPUT.PUT_LINE('SQLCODE : ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE('SQLERRM : ' || SQLERRM);
END;

```

例外が発生した場合、プロシージャで行われたすべてのデータベース変更は自動的にロールバックされます。この例では、WHEN OTHERS 句のある EXCEPTION セクションがすべての例外をキャッチします。2 つの変数が表示されます。SQLCODE は、発生した特定の例外を識別する番号です。SQLERRM はエラーを説明するテキストメッセージです。例外処理の詳細については、[ExceptionHandling](#) を参照してください。

以下は、この手順が実行されたときの出力を示しています。

```

EXEC emp_insert(9503,'PETERSON','ANALYST',7902,'31-MAR-05',5000,NULL,40);

Added employee...
Employee # : 9503
Name       : PETERSON
Job        : ANALYST
Manager    : 7902
Hire Date  : 31-MAR-05 00:00:00
Salary     : 5000
Dept #     : 40
-----

SELECT * FROM emp WHERE empno = 9503;

 empno | ename   | job      | mgr   | hiredate       | sal    | comm | deptno
-----+-----+-----+-----+-----+-----+-----+-----
  9503 | PETERSON | ANALYST | 7902  | 31-MAR-05 00:00:00 | 5000.00 |      | 40
(1 row)

```

注釈: INSERT コマンドは FORALL ステートメントに含めることができます。FORALL ステートメントを使用すると、1 つの INSERT コマンドで、1 つ以上のコレクションで提供される値から複数の行を挿入できます。FORALL ステートメントの詳細については、:ref:「FORALL ステートメントの使用<using_the_forall_statement>」を参照してください。

4.4 NULL

最も単純なステートメントは NULL ステートメントです。このステートメントは、何も実行しない実行可能なステートメントです。

```
NULL;
```

以下は、最も簡単で有効な SPL プログラムです。

```
BEGIN
    NULL;
END;
```

NULL ステートメントは、IF-THEN-ELSE ステートメントのブランチなど、実行可能なステートメントが必要なプレースホルダーとして機能します。

例:

```
CREATE OR REPLACE PROCEDURE divide_it (
    p_numerator    IN  NUMBER,
    p_denominator  IN  NUMBER,
    p_result       OUT NUMBER
)
IS
BEGIN
    IF p_denominator = 0 THEN
        NULL;
    ELSE
        p_result := p_numerator / p_denominator;
    END IF;
END;
```

4.5 Using the RETURNING INTO Clause

INSERT, UPDATE および DELETE コマンドには、オプションの RETURNING INTO 節を追加できます。この句により、SPL プログラムは、INSERT, UPDATE または DELETE コマンドの結果から、それぞれ新しく追加、変更、または削除された値をキャプチャできます。

以下は構文です。

```
{ <insert> | <update> | <delete> }
    RETURNING { * | <expr_1> [, <expr_2> ] ...}
    INTO { <record> | <field_1> [, <field_2> ] ...};
```

insert は有効な INSERT コマンドです。update は有効な UPDATE コマンドです。delete は有効な「13」

コマンドです。「10」を指定すると、「12」または「13」コマンドの影響を受ける行の値が、「14」キーワードの右側のレコードまたはフィールドへの割り当てに使用可能になります。（「10」の使用は **AdvancedServer** 拡張機能であり、**Oracle** データベースと互換性がないことに注意してください。）「11」は「12」または「13」コマンドの影響を受ける行で評価される式です。評価結果は、「14」キーワードの右側のレコードまたはフィールドに割り当てられます。「15」は、番号と順序が一致するフィールドを含む必要があるレコードの識別子であり、「18」句の値と互換性のあるデータ型です。「17」は、数と順序が一致する必要がある変数であり、「18」句の値のセットと互換性のあるデータ型です。

INSERT, UPDATE または DELETE コマンドが複数行の結果セットを返す場合、SQLCODE 01422, query returned more than one row で例外がスローされます。結果セットに行がない場合、INTO キーワードに続く変数は null に設定されます。

注釈: コレクションに返される複数の行の結果セットを許可する BULK COLLECT 句を使用する RETURNING INTO のバリエーションがあります。BULK COLLECT 句の詳細については、[BULKCOLLECT 句の使用](#) を参照してください。

次の例は、UPDATE<update>‘で導入された ‘emp_comp_update‘ プロシージャの変更で、RETURNING INTO 句が追加されています。

```
CREATE OR REPLACE PROCEDURE emp_comp_update (
    p_empno      IN emp.empno%TYPE,
    p_sal        IN emp.sal%TYPE,
    p_comm       IN emp.comm%TYPE
)
IS
    v_empno      emp.empno%TYPE;
    v_ename      emp.ename%TYPE;
    v_job        emp.job%TYPE;
    v_sal        emp.sal%TYPE;
    v_comm       emp.comm%TYPE;
    v_deptno     emp.deptno%TYPE;
BEGIN
    UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno
    RETURNING
        empno,
        ename,
        job,
        sal,
        comm,
        deptno
    INTO
        v_empno,
        v_ename,
        v_job,
```

(次のページに続く)

(前のページからの続き)

```

        v_sal,
        v_comm,
        v_deptno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || v_empno);
        DBMS_OUTPUT.PUT_LINE('Name                : ' || v_ename);
        DBMS_OUTPUT.PUT_LINE('Job                : ' || v_job);
        DBMS_OUTPUT.PUT_LINE('Department       : ' || v_deptno);
        DBMS_OUTPUT.PUT_LINE('New Salary        : ' || v_sal);
        DBMS_OUTPUT.PUT_LINE('New Commission    : ' || v_comm);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;

```

次に、この手順の出力を示します（emp_insert 手順で作成された従業員 9503 がテーブル内にまだ存在すると仮定します）。

```

EXEC emp_comp_update(9503, 6540, 1200);

Updated Employee # : 9503
Name                : PETERSON
Job                 : ANALYST
Department         : 40
New Salary          : 6540.00
New Commission      : 1200.00

```

次の例は、レコードタイプを使用して RETURNING INTO 句を追加した emp_delete プロシージャの変更です。

```

CREATE OR REPLACE PROCEDURE emp_delete (
    p_empno          IN emp.empno%TYPE
)
IS
    r_emp            emp%ROWTYPE;
BEGIN
    DELETE FROM emp WHERE empno = p_empno
    RETURNING
        *
    INTO
        r_emp;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || r_emp.empno);
        DBMS_OUTPUT.PUT_LINE('Name                : ' || r_emp.ename);
        DBMS_OUTPUT.PUT_LINE('Job                : ' || r_emp.job);
        DBMS_OUTPUT.PUT_LINE('Manager             : ' || r_emp.mgr);
    END IF;
END;

```

(次のページに続く)

(前のページからの続き)

```

        DBMS_OUTPUT.PUT_LINE('Hire Date           : ' || r_emp.hiredate);
        DBMS_OUTPUT.PUT_LINE('Salary             : ' || r_emp.sal);
        DBMS_OUTPUT.PUT_LINE('Commission        : ' || r_emp.comm);
        DBMS_OUTPUT.PUT_LINE('Department       : ' || r_emp.deptno);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

以下は、この手順の出力です。

```

EXEC emp_delete(9503);

Deleted Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Manager            : 7902
Hire Date          : 31-MAR-05 00:00:00
Salary             : 6540.00
Commission         : 1200.00
Department        : 40
```

4.6 SELECT INTO

SELECT INTO ステートメントは、SQL の SELECT コマンドの **SPL** バリエーションです。違いは次のとおりです。

- その SELECT INTO は、結果を変数またはレコードに割り当てて、**SPL** プログラムステートメントで使用できるように設計されています。
- アクセス可能な SELECT INTO の結果セットは、最大で 1 行です。

上記以外に、SELECT コマンドの WHERE, ORDER BY, GROUP BY, HAVING などのすべての句は SELECT INTO に対して有効です。以下は、SELECT INTO の 2 つのバリエーションです。

```
SELECT <select_expressions> INTO <target> FROM ...;
```

target は、カンマで区切られた単純変数のリストです。select_expressions およびステートメントの残りの部分は、SELECT コマンドの場合と同じです。選択した値は、ターゲットの構造のデータ型、数、順序が完全に一致する必要があります。一致しない場合、ランタイムエラーが発生します。

```
SELECT * INTO <record> FROM <table> ...;
```

record は、以前に宣言されたレコード変数です。

クエリがゼロ行を返す場合、**null** 値がターゲットに割り当てられます。クエリが複数の行を返す場合、最初の行がターゲットに割り当てられ、残りは破棄されます。(ORDER BY. を使用しない限り、「最初の行」は明確に定義されていないことに注意してください)

注釈: どちらの場合でも、行が返されないか、複数の行が返されると、SPL は例外をスローします。

注釈: コレクションに返される複数の行の結果セットを許可する BULK COLLECT 句を使用する SELECT INTO のバリエーションがあります。SELECT INTO ステートメントで BULK COLLECT 節を使用する方法の詳細については、[SELECTBULKCOLLECT](#) を参照してください。

EXCEPTION ブロックで WHEN NO_DATA_FOUND 句を使用して、割り当てが成功したかどうかを判断できます (つまり、少なくとも 1 つの行がクエリによって返されました)。

このバージョンの emp_sal_query プロシージャは、結果セットをレコードに返す SELECT INTO のバリエーションを使用します。WHEN NO_DATA_FOUND 条件式を含む EXCEPTION ブロックの追加にも注意してください。

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno          IN emp.empno%TYPE
)
IS
    r_emp            emp%ROWTYPE;
    v_avgsal         emp.sal%TYPE;
BEGIN
    SELECT * INTO r_emp
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee's salary is more than the '
            || 'department average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee's salary does not exceed the '
            || 'department average of ' || v_avgsal);
    END IF;
EXCEPTION
```

(次のページに続く)

(前のページからの続き)

```

    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
END;
```

存在しない従業員番号を使用してクエリを実行すると、結果は次のように表示されます。

```

EXEC emp_sal_query(0);

Employee # 0 not found
```

SELECT INTO を含む EXCEPTION セクションで使用される別の条件節は、TOO_MANY_ROWS 例外です。SELECT INTO ステートメントによって複数の行が選択された場合、SPL によって例外がスローされます。

次のブロックが実行されると、指定された部門に多くの従業員がいるため、TOO_MANY_ROWS 例外がスローされます。

```

DECLARE
    v_ename          emp.ename%TYPE;
BEGIN
    SELECT ename INTO v_ename FROM emp WHERE deptno = 20 ORDER BY ename;
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('More than one employee found');
        DBMS_OUTPUT.PUT_LINE('First employee returned is ' || v_ename);
END;
```

More than one employee found
First employee returned is ADAMS

注釈: 例外処理の詳細については、[ExceptionHandling](#) を参照してください。

4.7 UPDATE

SQL 言語で使用可能な UPDATE コマンドは、SPL プログラムでも使用できます。

SPL 言語の式は、SQL の UPDATE コマンドで式を使用できる場所であればどこでも使用できます。したがって、SPL 変数およびパラメーターを使用して、更新操作に値を提供できます。

```

CREATE OR REPLACE PROCEDURE emp_comp_update (
    p_empno          IN emp.empno%TYPE,
    p_sal            IN emp.sal%TYPE,
    p_comm           IN emp.comm%TYPE
```

(次のページに続く)

(前のページからの続き)

```

)
IS
BEGIN
    UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || p_empno);
        DBMS_OUTPUT.PUT_LINE('New Salary          : ' || p_sal);
        DBMS_OUTPUT.PUT_LINE('New Commission       : ' || p_comm);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;

```

SQL%FOUND 条件式は、行が更新されると TRUE を返し、そうでない場合は FALSE を返します。SQL%FOUND および他の同様の式の説明については、:ref:「結果ステータスの取得<obtaining_the_result_status>」を参照してください。

以下は、この手順を使用した従業員の更新を示しています。

```

EXEC emp_comp_update(9503, 6540, 1200);

Updated Employee # : 9503
New Salary          : 6540
New Commission       : 1200

SELECT * FROM emp WHERE empno = 9503;

empno | ename   | job      | mgr  | hiredate       | sal   | comm  | deptno
-----+-----+-----+-----+-----+-----+-----+-----
 9503 | PETERSON | ANALYST | 7902 | 31-MAR-05 00:00:00 | 6540.00 | 1200.00 | 40
(1 row)

```

注釈: UPDATE コマンドは FORALL ステートメントに含めることができます。FORALL ステートメントを使用すると、1 つの UPDATE コマンドで 1 つ以上のコレクションで提供された値から複数の行を更新できます。FORALL ステートメントの詳細については、:ref:「FORALL ステートメントの使用<using_the_forall_statement>」を参照してください。

4.8 Obtaining the Result Status

コマンドの効果を判断するために使用できる属性がいくつかあります。SQL%FOUND は、少なくとも 1 つの行が INSERT, UPDATE または DELETE コマンドの影響を受けた場合、または SELECT INTO コマンドが 1 つ以上の行を取得した場合に TRUE を返すブール値です。

次の匿名ブロックは行を挿入し、行が挿入されたという事実を表示します。

```
BEGIN
    INSERT INTO emp (empno,ename,job,sal,deptno) VALUES (
        9001, 'JONES', 'CLERK', 850.00, 40);
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Row has been inserted');
    END IF;
END;

Row has been inserted
```

SQL%ROWCOUNT は、INSERT, UPDATE, DELETE または SELECT INTO コマンドの影響を受ける行数を提供します。SQL%ROWCOUNT 値は BIGINT データ型として返されます。次の例は、挿入されたばかりの行を更新し、SQL%ROWCOUNT を表示します。

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9001;
    DBMS_OUTPUT.PUT_LINE('# rows updated: ' || SQL%ROWCOUNT);
END;

# rows updated: 1
```

SQL%NOTFOUND は SQL%FOUND の反対です。INSERT, UPDATE または DELETE コマンドの影響を受けた行がない場合、または SELECT INTO コマンドが行を取得しなかった場合、SQL%NOTFOUND は TRUE を返します。

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9000;
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('No rows were updated');
    END IF;
END;

No rows were updated
```

第 5 章

Control Structures

このセクションでは、SPL を SQL の完全な手続き型補完にする SPL のプログラミングステートメントについて説明します。

5.1 IF Statement

IF ステートメントを使用すると、特定の条件に基づいてコマンドを実行できます。SPL には IF: の 4 つの形式があります

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSE IF
- IF ... THEN ... ELSIF ... THEN ... ELSE

5.1.1 IF-THEN

```
IF boolean-expression THEN  
  <statements>  
END IF;
```

IF-THEN ステートメントは、IF の最も単純な形式です。条件が TRUE の場合、THEN と END IF の間のステートメントが実行されます。それ以外の場合、それらはスキップされます。

次の例では、IF-THEN ステートメントを使用して、コミッションを持つ従業員をテストおよび表示しています。

```
DECLARE  
  v_empno          emp.empno%TYPE;  
  v_comm           emp.comm%TYPE;
```

(次のページに続く)

(前のページからの続き)

```

    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO      COMM');
    DBMS_OUTPUT.PUT_LINE('-----      -----');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
-- Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || ' ' ||
                TO_CHAR(v_comm, '$99999.99'));
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

以下は、このプログラムからの出力です。

EMPNO	COMM
-----	-----
7499	\$300.00
7521	\$500.00
7654	\$1400.00

5.1.2 IF-THEN-ELSE

```

IF boolean-expression THEN
    <statements>
ELSE
    <statements>
END IF;
```

IF-THEN-ELSE ステートメントは、条件が **false** と評価された場合に実行するステートメントの代替セットを指定できるようにすることで、IF-THEN に追加されます。

前の例は変更されているため、従業員が手数料を受け取らなかった場合に IF-THEN-ELSE ステートメントを使用してテキスト **Non-commission** が表示されます。

```

DECLARE
    v_empno          emp.empno%TYPE;
    v_comm           emp.comm%TYPE;
    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
```

(次のページに続く)

(前のページからの続き)

```

BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO      COMM');
    DBMS_OUTPUT.PUT_LINE('-----      -----');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
-- Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || ' ' ||
                TO_CHAR(v_comm, '$99999.99'));
        ELSE
            DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || 'Non-commission');
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;

```

以下は、このプログラムからの出力です。

EMPNO	COMM
-----	-----
7369	Non-commission
7499	\$ 300.00
7521	\$ 500.00
7566	Non-commission
7654	\$ 1400.00
7698	Non-commission
7782	Non-commission
7788	Non-commission
7839	Non-commission
7844	Non-commission
7876	Non-commission
7900	Non-commission
7902	Non-commission
7934	Non-commission

5.1.3 IF-THEN-ELSE IF

IF ステートメントをネストして、外側の IF ステートメントの条件が TRUE または FALSE であるかどうかを判断した後、代替の IF ステートメントを呼び出すことができます。

次の例では、外側の IF-THEN-ELSE ステートメントは、従業員に手数料があるかどうかをテストします。内側の IF-THEN-ELSE ステートメントは、従業員の報酬総額が会社の平均を上回るか下回るかをテストします。

```

DECLARE
    v_empno      emp.empno%TYPE;
    v_sal        emp.sal%TYPE;
    v_comm       emp.comm%TYPE;
    v_avg        NUMBER(7,2);
    CURSOR emp_cursor IS SELECT empno, sal, comm FROM emp;
BEGIN
--
-- Calculate the average yearly compensation in the company
--
SELECT AVG((sal + NVL(comm,0)) * 24) INTO v_avg FROM emp;
DBMS_OUTPUT.PUT_LINE('Average Yearly Compensation: ' ||
    TO_CHAR(v_avg, '$999,999.99'));
OPEN emp_cursor;
DBMS_OUTPUT.PUT_LINE('EMPNO      YEARLY COMP');
DBMS_OUTPUT.PUT_LINE('-----      -----');
LOOP
    FETCH emp_cursor INTO v_empno, v_sal, v_comm;
    EXIT WHEN emp_cursor%NOTFOUND;
--
-- Test whether or not the employee gets a commission
--
    IF v_comm IS NOT NULL AND v_comm > 0 THEN
--
-- Test if the employee's compensation with commission exceeds the average
--
        IF (v_sal + v_comm) * 24 > v_avg THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || ' ' ||
                TO_CHAR((v_sal + v_comm) * 24, '$999,999.99') ||
                ' Exceeds Average');
        ELSE
            DBMS_OUTPUT.PUT_LINE(v_empno || ' ' ||
                TO_CHAR((v_sal + v_comm) * 24, '$999,999.99') ||
                ' Below Average');
        END IF;
    ELSE
--
-- Test if the employee's compensation without commission exceeds the
average
--
        IF v_sal * 24 > v_avg THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || ' ' ||
                TO_CHAR(v_sal * 24, '$999,999.99') || ' Exceeds Average');
        ELSE
            DBMS_OUTPUT.PUT_LINE(v_empno || ' ' ||
                TO_CHAR(v_sal * 24, '$999,999.99') || ' Below Average');
        END IF;
    END IF;
END IF;

```

(次のページに続く)

(前のページからの続き)

```

END LOOP;
CLOSE emp_cursor;
END;

```

注釈: このプログラムのロジックは、カーソル宣言の SELECT コマンド内で NVL 関数を使用して従業員の年間報酬を計算することで大幅に簡素化できますが、この例の目的は IF ステートメントの使用方法を示すことです。

以下は、このプログラムからの出力です。

```

Average Yearly Compensation: $ 53,528.57
EMPNO      YEARLY COMP
-----
7369 $ 19,200.00 Below Average
7499 $ 45,600.00 Below Average
7521 $ 42,000.00 Below Average
7566 $ 71,400.00 Exceeds Average
7654 $ 63,600.00 Exceeds Average
7698 $ 68,400.00 Exceeds Average
7782 $ 58,800.00 Exceeds Average
7788 $ 72,000.00 Exceeds Average
7839 $ 120,000.00 Exceeds Average
7844 $ 36,000.00 Below Average
7876 $ 26,400.00 Below Average
7900 $ 22,800.00 Below Average
7902 $ 72,000.00 Exceeds Average
7934 $ 31,200.00 Below Average

```

このフォームを使用すると、実際には、外側の IF ステートメントの ELSE 部分内に IF ステートメントをネストします。したがって、ネストされた IF ごとに 1 つの END IF ステートメントと、親 IF-ELSE に対して 1 つのステートメントが必要です。

5.1.4 IF-THEN-ELSIF-ELSE

```

IF boolean-expression THEN
    <statements>
[ ELSEIF boolean-expression THEN
    <statements>
[ ELSEIF boolean-expression THEN
    <statements> ] ...]
[ ELSE
    <statements> ]
END IF;

```

IF-THEN-ELSIF-ELSE は、1 つのステートメントで多くの選択肢をチェックする方法を提供します。正式には、ネストされた IF-THEN-ELSE-IF-THEN コマンドと同等ですが、必要な END IF は 1 つだけです。

次の例では、IF-THEN-ELSIF-ELSE ステートメントを使用して、従業員数を 25,000 ドルの報酬範囲でカウントします。

```
DECLARE
    v_empno          emp.empno%TYPE;
    v_comp           NUMBER(8,2);
    v_lt_25K         SMALLINT := 0;
    v_25K_50K        SMALLINT := 0;
    v_50K_75K        SMALLINT := 0;
    v_75K_100K       SMALLINT := 0;
    v_ge_100K        SMALLINT := 0;
    CURSOR emp_cursor IS SELECT empno, (sal + NVL(comm,0)) * 24 FROM emp;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO v_empno, v_comp;
        EXIT WHEN emp_cursor%NOTFOUND;
        IF v_comp < 25000 THEN
            v_lt_25K := v_lt_25K + 1;
        ELSIF v_comp < 50000 THEN
            v_25K_50K := v_25K_50K + 1;
        ELSIF v_comp < 75000 THEN
            v_50K_75K := v_50K_75K + 1;
        ELSIF v_comp < 100000 THEN
            v_75K_100K := v_75K_100K + 1;
        ELSE
            v_ge_100K := v_ge_100K + 1;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
    DBMS_OUTPUT.PUT_LINE('Number of employees by yearly compensation');
    DBMS_OUTPUT.PUT_LINE('Less than 25,000 : ' || v_lt_25K);
    DBMS_OUTPUT.PUT_LINE('25,000 - 49,9999 : ' || v_25K_50K);
    DBMS_OUTPUT.PUT_LINE('50,000 - 74,9999 : ' || v_50K_75K);
    DBMS_OUTPUT.PUT_LINE('75,000 - 99,9999 : ' || v_75K_100K);
    DBMS_OUTPUT.PUT_LINE('100,000 and over : ' || v_ge_100K);
END;
```

以下は、このプログラムからの出力です。

```
Number of employees by yearly compensation
Less than 25,000 : 2
25,000 - 49,9999 : 5
50,000 - 74,9999 : 6
75,000 - 99,9999 : 0
100,000 and over : 1
```

5.2 RETURN Statement

RETURN ステートメントは、現在の関数、プロシージャ、または匿名ブロックを終了し、呼び出し元に制御を返します。

RETURN ステートメントには 2 つの形式があります。RETURN ステートメントの最初の形式は、void を返すプロシージャまたは関数を終了するために使用されます。最初の形式の構文は次のとおりです。

```
RETURN;
```

RETURN の 2 番目の形式は、呼び出し元に値を返します。RETURN ステートメントの 2 番目の形式の構文は次のとおりです。

```
RETURN <expression>;
```

expression は、関数の戻り値の型と同じデータ型に評価される必要があります。

次の例では、RETURN ステートメントを使用して、呼び出し元に値を返します。

```
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal          NUMBER,
    p_comm         NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END emp_comp;
```

5.3 GOTO Statement

GOTO ステートメントにより、実行ポイントは指定されたラベルを持つステートメントにジャンプします。GOTO ステートメントの構文は次のとおりです。

```
GOTO <label>
```

label は、実行可能なステートメントに割り当てられた名前です。label は、関数、プロシージャ、または匿名ブロックのスコープ内で一意でなければなりません。

ステートメントにラベルを付けるには、次の構文を使用します。

```
<<label>> <statement>
```

statement は、プログラムがジャンプする実行ポイントです。

割り当てステートメント、任意の SQL ステートメント（INSERT, UPDATE, CREATE など）および選択した手続き言語ステートメントにラベルを付けることができます。ラベル付けできる手続き言語ステートメントは次のとおりです。

- IF
- EXIT
- RETURN
- RAISE
- EXECUTE
- PERFORM
- GET DIAGNOSTICS
- OPEN
- FETCH
- MOVE
- CLOSE
- NULL
- COMMIT
- ROLLBACK
- GOTO
- CASE
- LOOP
- WHILE
- FOR

exit はキーワードと見なされ、ラベルの名前として使用できないことに注意してください。

GOTO ステートメントは、制御を条件付きブロックまたはサブブロックに転送できませんが、制御を条件付きブロックまたはサブブロックから転送できます。

次の例では、従業員レコードに名前、職務説明、従業員の雇用日が含まれていることを確認します。情報の一部が欠落している場合、GOTO ステートメントは実行ポイントを、従業員が無効であるというメッセージを出力するステートメントに転送します。

```
CREATE OR REPLACE PROCEDURE verify_emp (
    p_empno          NUMBER
)
IS
    v_ename          emp.ename%TYPE;
    v_job            emp.job%TYPE;
    v_hiredate       emp.hiredate%TYPE;
BEGIN
    SELECT ename, job, hiredate
        INTO v_ename, v_job, v_hiredate FROM emp
        WHERE empno = p_empno;
    IF v_ename IS NULL THEN
        GOTO invalid_emp;
    END IF;
    IF v_job IS NULL THEN
        GOTO invalid_emp;
    END IF;
    IF v_hiredate IS NULL THEN
        GOTO invalid_emp;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||
        ' validated without errors.');
```

```
RETURN;
<<invalid_emp>> DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||
    ' is not a valid employee.');
```

```
END;
```

GOTO ステートメントには次の制限があります。

- GOTO ステートメントは宣言にジャンプできません。
- GOTO ステートメントは、制御を別の関数またはプロシージャに転送できません。
- label は、ブロック、関数、またはプロシージャの最後に配置しないでください。

5.4 CASE Expression

CASE 式は、CASE 式が式内にある場合に置換される値を返します。

CASE 式には 2 つの形式があります。1 つは **searched** CASE と呼ばれ、もう 1 つは **selector** を使用します。

5.4.1 Selector CASE Expression

セレクトー CASE 式は、セレクトーと呼ばれる式を 1 つ以上の WHEN 句で指定された式に一致させようとしします。result は、CASE 式が使用されるコンテキストで型互換性がある式です。一致が見つかった場合、対応する THEN 句で指定された値が CASE 式によって返されます。一致するものがない場合、ELSE に続く値が返されます。ELSE を省略すると、CASE 式は null を返します。

```
CASE <selector-expression>
  WHEN <match-expression> THEN
    <result>
[ WHEN <match-expression> THEN
  <result>
[ WHEN <match-expression> THEN
  <result> ] ...]
[ ELSE
  <result> ]
END;
```

match-expression は「14」式内に現れる順序で評価されます。「11」は、「14」式が使用されるコンテキストで型互換性がある式です。「10」に等しい最初の match-expression が検出されると、対応する THEN 節の「11」が「14」式の値として返されます。match-expression が「10」に等しくない場合、「13」に続く「11」が返されます。「13」が指定されていない場合、「14」式は null を返します。

次の例では、セレクトー CASE 式を使用して、部門番号に基づいて変数に部門名を割り当てています。

```
DECLARE
  v_empno      emp.empno%TYPE;
  v_ename      emp.ename%TYPE;
  v_deptno     emp.deptno%TYPE;
  v_dname      dept.dname%TYPE;
  CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
  OPEN emp_cursor;
  DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME      DEPTNO      DNAME');
  DBMS_OUTPUT.PUT_LINE('-----      -');
  LOOP
    FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
    EXIT WHEN emp_cursor%NOTFOUND;
    v_dname :=
```

(次のページに続く)

(前のページからの続き)

```

        CASE v_deptno
            WHEN 10 THEN 'Accounting'
            WHEN 20 THEN 'Research'
            WHEN 30 THEN 'Sales'
            WHEN 40 THEN 'Operations'
            ELSE 'unknown'
        END;
        DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || RPAD(v_ename, 10) ||
            ' ' || v_deptno || ' ' || v_dname);
    END LOOP;
    CLOSE emp_cursor;
END;
```

以下は、このプログラムからの出力です。

EMPNO	ENAME	DEPTNO	DNAME
-----	-----	-----	-----
7369	SMITH	20	Research
7499	ALLEN	30	Sales
7521	WARD	30	Sales
7566	JONES	20	Research
7654	MARTIN	30	Sales
7698	BLAKE	30	Sales
7782	CLARK	10	Accounting
7788	SCOTT	20	Research
7839	KING	10	Accounting
7844	TURNER	30	Sales
7876	ADAMS	20	Research
7900	JAMES	30	Sales
7902	FORD	20	Research
7934	MILLER	10	Accounting

5.4.2 Searched CASE Expression

検索された CASE 式は、1 つ以上のブール式を使用して、返される結果の値を決定します。

```

CASE WHEN <boolean-expression> THEN
    <result>
[ WHEN <boolean-expression> THEN
    <result>
[ WHEN <boolean-expression> THEN
    <result> ] ...]
[ ELSE
    <result> ]
END;
```

boolean-expression は、「13」式内に現れる順序で評価されます。「10」は、「13」式が使用されるコンテキストで型互換性がある式です。TRUE と評価される最初の boolean-expression が検出されると、対応する THEN 節の「10」が「13」式の値として返されます。boolean-expression のいずれも true と評価されない場合、「12」に続く「10」が返されます。「12」が指定されていない場合、「13」式は null を返します。

次の例では、検索された CASE 式を使用して、部門番号に基づいて部門名を変数に割り当てます。

```
DECLARE
    v_empno      emp.empno%TYPE;
    v_ename      emp.ename%TYPE;
    v_deptno     emp.deptno%TYPE;
    v_dname      dept.dname%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME      DEPTNO      DNAME');
    DBMS_OUTPUT.PUT_LINE('-----      -');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        v_dname :=
            CASE
                WHEN v_deptno = 10 THEN 'Accounting'
                WHEN v_deptno = 20 THEN 'Research'
                WHEN v_deptno = 30 THEN 'Sales'
                WHEN v_deptno = 40 THEN 'Operations'
                ELSE 'unknown'
            END;
        DBMS_OUTPUT.PUT_LINE(v_empno || '      ' || RPAD(v_ename, 10) ||
            '      ' || v_deptno || '      ' || v_dname);
    END LOOP;
    CLOSE emp_cursor;
END;
```

以下は、このプログラムからの出力です。

EMPNO	ENAME	DEPTNO	DNAME
-----	-----	-----	-----
7369	SMITH	20	Research
7499	ALLEN	30	Sales
7521	WARD	30	Sales
7566	JONES	20	Research
7654	MARTIN	30	Sales
7698	BLAKE	30	Sales
7782	CLARK	10	Accounting
7788	SCOTT	20	Research
7839	KING	10	Accounting
7844	TURNER	30	Sales

(次のページに続く)

(前のページからの続き)

7876	ADAMS	20	Research
7900	JAMES	30	Sales
7902	FORD	20	Research
7934	MILLER	10	Accounting

5.5 CASE Statement

CASE ステートメントは、指定された検索条件が TRUE の場合、1 つ以上のステートメントのセットを実行します。CASE ステートメントはそれ自体がスタンドアロンのステートメントですが、前述の CASE 式は式の一部として表示する必要があります。

CASE ステートメントには 2 つの形式があります。1 つは **searched** CASE と呼ばれ、もう 1 つは **selector** を使用します。

5.5.1 Selector CASE Statement

セレクトー CASE ステートメントは、セレクトーと呼ばれる式を 1 つ以上の WHEN 句で指定された式に一致させようとします。一致が見つかったら、1 つ以上の対応するステートメントが実行されます。

```

CASE <selector-expression>
WHEN <match-expression> THEN
    <statements>
[ WHEN <match-expression> THEN
    <statements>
[ WHEN <match-expression> THEN
    <statements> ] ... ]
[ ELSE
    <statements> ]
END CASE;
```

selector-expression は、各 match-expression と互換性のある値を返します。match-expression は、CASE ステートメント内に現れる順序で評価されます。statements は、セミコロンで終了する 1 つ以上の SPL ステートメントです。selector-expression の値が最初の match-expression と等しい場合、対応する THEN 節のステートメントが実行され、END CASE キーワードに従って制御が続行されます。一致するものがない場合、「10」に続くステートメントが実行されます。一致するものがなく、「10」句がない場合、例外がスローされます。

次の例では、セレクトー CASE ステートメントを使用して、部門名と場所を部門番号に基づいて変数に割り当てます。

```

DECLARE
    v_empno      emp.empno%TYPE;
    v_ename      emp.ename%TYPE;
    v_deptno     emp.deptno%TYPE;
    v_dname      dept.dname%TYPE;
    v_loc        dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME      DEPTNO      DNAME      '
        || '      LOC');
    DBMS_OUTPUT.PUT_LINE('-----      -      -      -      -');
    || '      -');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE v_deptno
            WHEN 10 THEN v_dname := 'Accounting';
                        v_loc    := 'New York';
            WHEN 20 THEN v_dname := 'Research';
                        v_loc    := 'Dallas';
            WHEN 30 THEN v_dname := 'Sales';
                        v_loc    := 'Chicago';
            WHEN 40 THEN v_dname := 'Operations';
                        v_loc    := 'Boston';
            ELSE v_dname := 'unknown';
                v_loc    := '';
        END CASE;
        DBMS_OUTPUT.PUT_LINE(v_empno || '      ' || RPAD(v_ename, 10) ||
            '      ' || v_deptno || '      ' || RPAD(v_dname, 14) || '      ' ||
            v_loc);
    END LOOP;
    CLOSE emp_cursor;
END;

```

以下は、このプログラムからの出力です。

EMPNO	ENAME	DEPTNO	DNAME	LOC
-----	-----	-----	-----	-----
7369	SMITH	20	Research	Dallas
7499	ALLEN	30	Sales	Chicago
7521	WARD	30	Sales	Chicago
7566	JONES	20	Research	Dallas
7654	MARTIN	30	Sales	Chicago
7698	BLAKE	30	Sales	Chicago
7782	CLARK	10	Accounting	New York
7788	SCOTT	20	Research	Dallas
7839	KING	10	Accounting	New York

(次のページに続く)

(前のページからの続き)

7844	TURNER	30	Sales	Chicago
7876	ADAMS	20	Research	Dallas
7900	JAMES	30	Sales	Chicago
7902	FORD	20	Research	Dallas
7934	MILLER	10	Accounting	New York

5.5.2 Searched CASE Statement

検索された CASE ステートメントは、1 つ以上のブール式を使用して、実行するステートメントの結果セットを決定します。

```

CASE WHEN <boolean-expression> THEN
    <statements>
[ WHEN <boolean-expression> THEN
    <statements>
[ WHEN <boolean-expression> THEN
    <statements> ] ... ]
[ ELSE
    <statements> ]
END CASE;
```

boolean-expression は、CASE ステートメント内に現れる順序で評価されます。「10」と評価される最初の boolean-expression が検出されると、対応する THEN 節のステートメントが実行され、END CASE キーワードに従って制御が継続されます。boolean-expression のいずれも「10」と評価されない場合、「11」に続くステートメントが実行されます。boolean-expression のいずれも「10」と評価されず、「11」節がない場合、例外がスローされます。

次の例では、検索された CASE ステートメントを使用して、部門名と場所を部門番号に基づいて変数に割り当てます。

```

DECLARE
    v_empno      emp.empno%TYPE;
    v_ename      emp.ename%TYPE;
    v_deptno     emp.deptno%TYPE;
    v_dname      dept.dname%TYPE;
    v_loc        dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME      DEPTNO      DNAME      '
        || '      LOC');
    DBMS_OUTPUT.PUT_LINE('-----      -      -      -      -      -');
    || '      -      -      -      -      -';
    LOOP
```

(次のページに続く)

(前のページからの続き)

```

    FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
    EXIT WHEN emp_cursor%NOTFOUND;
    CASE
        WHEN v_deptno = 10 THEN v_dname := 'Accounting';
                                v_loc   := 'New York';
        WHEN v_deptno = 20 THEN v_dname := 'Research';
                                v_loc   := 'Dallas';
        WHEN v_deptno = 30 THEN v_dname := 'Sales';
                                v_loc   := 'Chicago';
        WHEN v_deptno = 40 THEN v_dname := 'Operations';
                                v_loc   := 'Boston';
        ELSE v_dname := 'unknown';
            v_loc    := '';
    END CASE;
    DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || RPAD(v_ename, 10) ||
        ' ' || v_deptno || ' ' || RPAD(v_dname, 14) || ' ' ||
        v_loc);
END LOOP;
CLOSE emp_cursor;
END;
```

以下は、このプログラムからの出力です。

EMPNO	ENAME	DEPTNO	DNAME	LOC
-----	-----	-----	-----	-----
7369	SMITH	20	Research	Dallas
7499	ALLEN	30	Sales	Chicago
7521	WARD	30	Sales	Chicago
7566	JONES	20	Research	Dallas
7654	MARTIN	30	Sales	Chicago
7698	BLAKE	30	Sales	Chicago
7782	CLARK	10	Accounting	New York
7788	SCOTT	20	Research	Dallas
7839	KING	10	Accounting	New York
7844	TURNER	30	Sales	Chicago
7876	ADAMS	20	Research	Dallas
7900	JAMES	30	Sales	Chicago
7902	FORD	20	Research	Dallas
7934	MILLER	10	Accounting	New York

5.6 Loops

LOOP, EXIT, CONTINUE, WHILE および FOR ステートメントを使用すると、SPL プログラムが一連のコマンドを繰り返すように調整できます。

5.6.1 LOOP

```
LOOP
    <statements>
END LOOP;
```

LOOP は、EXIT または RETURN ステートメントで終了するまで無期限に繰り返される無条件ループを定義します。

5.6.2 EXIT

```
EXIT [ WHEN <expression> ];
```

最も内側のループが終了し、END LOOP に続くステートメントが次に実行されます。

WHEN が存在する場合、指定された条件が TRUE の場合にのみループ終了が発生し、そうでない場合は EXIT の後のステートメントに制御が渡されます。

EXIT を使用すると、すべてのタイプのループを早期に終了できます。無条件ループでの使用に限定されません。

以下は、10 回反復し、EXIT ステートメントを使用して終了するループの簡単な例です。

```
DECLARE
    v_counter      NUMBER(2);
BEGIN
    v_counter := 1;
    LOOP
        EXIT WHEN v_counter > 10;
        DBMS_OUTPUT.PUT_LINE('Iteration # ' || v_counter);
        v_counter := v_counter + 1;
    END LOOP;
END;
```

以下は、このプログラムからの出力です。

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
```

(次のページに続く)

(前のページからの続き)

```
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

5.6.3 CONTINUE

CONTINUE ステートメントは、間にあるステートメントをスキップしながら、ループの次の反復に進む方法を提供します。

CONTINUE ステートメントが見つかると、最も内側のループの次の反復が開始され、CONTINUE ステートメントに続くすべてのステートメントがループの終わりまでスキップされます。つまり、制御はループ制御式（存在する場合）に戻され、ループの本体が再評価されます。

WHEN 句が使用される場合、WHEN 句で指定された式が TRUE に評価される場合にのみ、ループの次の反復が開始されます。それ以外の場合、制御は CONTINUE ステートメントに続く次のステートメントに渡されます。

CONTINUE ステートメントはループ外では使用できません。

以下は、CONTINUE ステートメントを使用して奇数の表示をスキップする前の例のバリエーションです。

```
DECLARE
    v_counter      NUMBER(2);
BEGIN
    v_counter := 0;
    LOOP
        v_counter := v_counter + 1;
        EXIT WHEN v_counter > 10;
        CONTINUE WHEN MOD(v_counter,2) = 1;
        DBMS_OUTPUT.PUT_LINE('Iteration # ' || v_counter);
    END LOOP;
END;
```

以下は、上記のプログラムからの出力です。

```
Iteration # 2
Iteration # 4
Iteration # 6
Iteration # 8
Iteration # 10
```

5.6.4 WHILE

```
WHILE <expression> LOOP
    <statements>
END LOOP;
```

WHILE ステートメントは、条件式が TRUE と評価される限り、一連のステートメントを繰り返します。条件は、ループ本体への各エントリの直前にチェックされます。

次の例には、前の例と同じロジックが含まれていますが、WHILE ステートメントを使用して EXIT ステートメントの代わりにループを終了するタイミングを決定します。

注釈: ループをいつ終了するかを決定するために使用される条件式を変更する必要があります。EXIT ステートメントは、条件式が真の場合にループを終了します。条件式が偽の場合、WHILE ステートメントは終了します（またはループを開始しません）。

```
DECLARE
    v_counter      NUMBER(2);
BEGIN
    v_counter := 1;
    WHILE v_counter <= 10 LOOP
        DBMS_OUTPUT.PUT_LINE('Iteration # ' || v_counter);
        v_counter := v_counter + 1;
    END LOOP;
END;
```

この例では、前の例と同じ結果が生成されます。

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

5.6.5 FOR (integer variant)

```
FOR <name> IN [REVERSE] <expression .. expression> LOOP
    <statements>
END LOOP;
```

この FOR の形式は、整数値の範囲で反復するループを作成します。変数 name はタイプ INTEGER として自動的に定義され、ループ内にのみ存在します。ループに入る 2 つの式は、ループに入るときに 1 回評価されます。反復ステップは +1 であり、name は.. の左側の expression の値で始まり、name が右側の expression の値を超えると終了します。したがって、2 つの式が引き継がれます。次の役割: start-value .. end-value.

オプションの REVERSE 句は、ループを逆の順序で繰り返すことを指定します。最初のループでは、name は右端の expression の値に設定されます。name が一番左の expression より小さい場合、ループは終了します。

次の例では、1 から 10 まで反復する FOR ループを使用して、WHILE ループの例をさらに簡素化します。

```
BEGIN
    FOR i IN 1 .. 10 LOOP
        DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
    END LOOP;
END;
```

FOR ステートメントを使用した出力は次のとおりです。

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

開始値が終了値より大きい場合、ループ本体はまったく実行されません。次の例に示すように、エラーは発生しません。

```
BEGIN
    FOR i IN 10 .. 1 LOOP
        DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
    END LOOP;
END;
```

ループ本体は実行されないため、この例からの出力はありません。

注釈: SPL は CURSOR FOR ループもサポートしています ([カーソル FOR ループ](#) を参照)。

5.7 Exception Handling

デフォルトでは、SPL プログラムでエラーが発生すると、プログラムの実行が中止されます。BEGIN ブロックと EXCEPTION セクションを使用して、エラーをトラップして回復できます。この構文は、BEGIN ブロックの通常の構文の拡張です。

```
[ DECLARE
    <declarations> ]
BEGIN
    <statements>
EXCEPTION
    WHEN <condition> [ OR <condition> ]... THEN
        <handler_statements>
[ WHEN <condition> [ OR <condition> ]... THEN
    <handler_statements> ]...
END;
```

エラーが発生しない場合、この形式のブロックはすべての statements を実行し、END の後の次のステートメントに制御が渡されます。statements 内でエラーが発生した場合、statements のさらなる処理は中止され、制御は EXCEPTION リストに渡されます。リストは、発生したエラーに一致する最初の condition を検索します。一致が見つかった場合、対応する handler_statements が実行され、END の後の次のステートメントに制御が渡されます。一致するものが見つからない場合、EXCEPTION 句がまったくないかのようにエラーが伝播します。エラーは、EXCEPTION で囲まれたブロックでキャッチできます。囲んでいるブロックがない場合、サブプログラムの処理を中止します。

特別な条件名 OTHERS は、すべてのエラータイプに一致します。条件名では大文字と小文字は区別されません。

選択された handler_statements 内で新しいエラーが発生した場合、この EXCEPTION 節でキャッチすることとはできませんが、伝播されます。周囲の EXCEPTION 句でキャッチできます。

次の表に、使用できる条件名を示します。

条件名	説明
CASE_NOT_FOUND	アプリケーションは、CASE ステートメントのどのケースも TRUE と評価されず、ELSE 条件がないという状況に遭遇しました。
COLLECTION_IS_NULL	アプリケーションは、初期化されていない入れ子になったテーブルなどの NULL コレクションでコレクションメソッドを呼び出そうとしました。
CURSOR_ALREADY_OPEN	アプリケーションは、すでに開いているカーソルを開こうとしました。
DUP_VAL_ON_INDEX	アプリケーションは、制約された列内に現在存在する重複値を保存しようとしてしました。
INVALID_CURSOR	アプリケーションは、開かれていないカーソルにアクセスしようとしてしました。
INVALID_NUMBER	アプリケーションでデータ例外が発生しました (SQLSTATE クラスコード 22 に相当)。INVALID_NUMBER は VALUE_ERROR のエイリアスです。
NO_DATA_FOUND	選択基準を満たす行はありません。
OTHERS	アプリケーションは、例外セクションの前の条件に捕捉されていない例外を検出しました。
SUBSCRIPT_BEYOND_COUNT	アプリケーションは、初期化または拡張サイズを超えて、ネストした表または VARRAY の添字を参照しようとしてしました。
SUBSCRIPT_OUTSIDE_LIMIT	アプリケーションは、添え字を参照しようとしたか、VARRAY を最大サイズ制限を超えて拡張しようとしてしました。
TOO_MANY_ROWS	アプリケーションは、選択条件を満たす複数の行を検出しました (1 行のみを返すことが許可されています)。
VALUE_ERROR	アプリケーションでデータ例外が発生しました (SQLSTATE クラスコード 22 に相当)。VALUE_ERROR は INVALID_NUMBER のエイリアスです。
ZERO_DIVIDE	アプリケーションはゼロで除算しようとしてしました。
User-defined Exception	ユーザー定義の例外 を参照してください。

注釈: 条件名 INVALID_NUMBER および VALUE_ERROR は、文字列の数値リテラルへの変換の失敗のみに起因する例外用のこれらの条件名がある Oracle データベースと互換性がありません。さらに、Oracle データベースの場合、INVALID_NUMBER 例外は SQL ステートメントにのみ適用され、VALUE_ERROR 例外は手続き型ステートメントにのみ適用されます。

5.8 User-defined Exceptions

プログラムの実行中に、任意の数のエラー（PL/SQL では*excepts*と呼ばれる）が発生する可能性があります。例外が「スロー」されると、プログラムの通常の実行は停止し、プログラムの制御はプログラムのエラー処理部分に移ります。例外は、サーバーによって生成される事前定義のエラーの場合もあれば、ユーザー定義の例外を発生させる論理エラーの場合もあります。

ユーザー定義の例外がサーバーによって発生することはありません。それらは RAISE ステートメントによって明示的に発生します。開発者が定義した論理規則が破られると、ユーザー定義の例外が発生します。資金不足の口座に対して小切手が提示されると、論理ルールが破られる一般的な例が発生します。資金不足の口座に対して小切手を換金しようとする、ユーザー定義の例外が発生します。

関数、プロシージャ、パッケージまたは匿名ブロックで例外を定義できます。同じブロックで同じ例外を 2 回宣言することはできませんが、2 つの異なるブロックで同じ例外を宣言できます。

ユーザー定義の例外を実装する前に、関数、プロシージャ、パッケージまたは匿名ブロックの宣言セクションで例外を宣言する必要があります。その後、RAISE ステートメントを使用して例外を発生させることができます。

```
DECLARE
    <exception_name> EXCEPTION;

BEGIN
    ...
    RAISE <exception_name>;
    ...
END;
```

exception_name は例外の名前です。

未処理の例外は、コールスタックを通じて伝播します。例外が未処理のままの場合、例外は最終的にクライアントアプリケーションに報告されます。

ブロックで宣言されたユーザー定義の例外は、そのブロックに対してローカルであり、ブロック内のネストされたブロックに対してグローバルであると見なされます。外側のブロックにある例外を参照するには、外側のブロックにラベルを割り当てる必要があります。次に、例外の名前の前にブロック名を付けます。

```
block_name.exception_name
```

逆に、外側のブロックは、ネストされたブロックで宣言された例外を参照できません。

宣言のスコープは、パッケージ内で作成されている場合を除き、宣言されているブロックに制限され、参照されるとパッケージ名で修飾されます。たとえば、inventory_control という名前のパッケージにある out_of_stock という名前の例外を発生させるには、プログラムは次の名前のエラーを発生させる必要があります。

```
inventory_control.out_of_stock
```

次の例は、パッケージでユーザー定義の例外を宣言する方法を示しています。ユーザー定義の例外は、例外と同じパッケージに存在するため、`check_balance` で発生した場合にパッケージ修飾子を必要としません。

```
CREATE OR REPLACE PACKAGE ar AS
    overdrawn EXCEPTION;
    PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
    PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER)
    IS
    BEGIN
        IF (p_amount > p_balance) THEN
            RAISE overdrawn;
        END IF;
    END;
END;
```

次のプロシージャ (`purchase`) は `check_balance` プロシージャを呼び出します。 `p_amount` が `p_balance`, `check_balance` よりも大きい場合、例外が発生します。 `purchase` は `ar.overdrawn` 例外をキャッチします。 `purchase` は `ar` パッケージ内で定義されていないため、`purchase` はパッケージ修飾名 (`ar.overdrawn`) の例外を参照する必要があります。

```
CREATE PROCEDURE purchase(customerID INT, amount NUMERIC)
AS
BEGIN
    ar.check_balance(getcustomerbalance(customerid), amount);
    record_purchase(customerid, amount);
EXCEPTION
    WHEN ar.overdrawn THEN
        raise_credit_limit(customerid, amount*1.5);
END;
```

`ar.check_balance` が例外が発生させると、実行は `purchase:` で定義された例外ハンドラにジャンプします

```
EXCEPTION
    WHEN ar.overdrawn THEN
        raise_credit_limit(customerid, amount*1.5);
```

例外ハンドラーは、顧客の信用限度を引き上げて終了します。例外ハンドラーが終了すると、`ar.check_balance` に続くステートメントから実行が再開されます。

5.9 PRAGMA EXCEPTION_INIT

PRAGMA EXCEPTION_INIT は、ユーザー定義のエラーコードを例外に関連付けます。PRAGMA EXCEPTION_INIT 宣言は、ブロック、サブブロック、またはパッケージに含めることができます。例外を宣言した後にのみ、エラーコードを例外に割り当てることができます (PRAGMA EXCEPTION_INIT を使用)。PRAGMA EXCEPTION_INIT 宣言の形式は次のとおりです。

```
PRAGMA EXCEPTION_INIT (<exception_name>,
                        {<exception_number> | <exception_code>})
```

どこで:

exception_name は関連する例外の名前です。

exception_number は、プラグマに関連付けられたユーザー定義のエラーコードです。マッピングされていない exception_number を指定すると、サーバーは警告を返します。

exception_code は事前定義された例外の名前です。有効な例外の完全なリストについては、次の場所にある Postgres のコアドキュメントを参照してください。

<https://www.postgresql.org/docs/current/static/errcodes-appendix.html>

前のセクション (ユーザー定義の例外) には、パッケージでユーザー定義の例外を宣言する例を示す例が含まれていました。次の例では、同じ基本構造を使用していますが、PRAGMA EXCEPTION_INIT 宣言を追加しています。

```
CREATE OR REPLACE PACKAGE ar AS
    overdrawn EXCEPTION;
    PRAGMA EXCEPTION_INIT (overdrawn, -20100);
    PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
    PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER)
    IS
    BEGIN
        IF (p_amount > p_balance) THEN
            RAISE overdrawn;
        END IF;
    END;
END;
```

次のプロシージャ (purchase) は check_balance プロシージャを呼び出します。p_amount が p_balance, check_balance よりも大きい場合、例外が発生します。purchase は ar.overdrawn 例外をキャッチします。

```
CREATE PROCEDURE purchase(customerID int, amount NUMERIC)
AS
BEGIN
    ar.check_balance(getcustomerbalance(customerid), amount);
    record_purchase(customerid, amount);
EXCEPTION
    WHEN ar.overdrawn THEN
        DBMS_OUTPUT.PUT_LINE ('This account is overdrawn. ');
        DBMS_OUTPUT.PUT_LINE ('SQLCode :'||SQLCODE||' '||SQLERRM );
END;
```

ar.check_balance が例外を発生させると、実行は purchase で定義された例外ハンドラにジャンプします。

```
EXCEPTION
    WHEN ar.overdrawn THEN
        DBMS_OUTPUT.PUT_LINE ('This account is overdrawn. ');
        DBMS_OUTPUT.PUT_LINE ('SQLCode :'||SQLCODE||' '||SQLERRM );
```

例外ハンドラーはエラーメッセージを返し、その後に SQLCODE 情報が続きます。

```
This account is overdrawn.
SQLCODE: -20100 User-Defined Exception
```

次の例は、事前定義の例外の使用方法を示しています。コードは、no_data_found exception のより意味のある名前を作成します。指定された顧客が存在しない場合、コードは例外をキャッチし、DBMS_OUTPUT.PUT_LINE を呼び出してエラーを報告し、元の例外を再度発生させます。

```
CREATE OR REPLACE PACKAGE ar AS
    overdrawn EXCEPTION;
    PRAGMA EXCEPTION_INIT (unknown_customer, no_data_found);
    PROCEDURE check_balance(p_customer_id NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
    PROCEDURE check_balance(p_customer_id NUMBER)
    IS
    DECLARE
        v_balance NUMBER;
    BEGIN
        SELECT balance INTO v_balance FROM customer
            WHERE cust_id = p_customer_id;
    EXCEPTION WHEN unknown_customer THEN
        DBMS_OUTPUT.PUT_LINE('invalid customer id');
        RAISE;
    END;
```

5.10 RAISE_APPLICATION_ERROR

プロシージャ RAISE_APPLICATION_ERROR を使用すると、開発者は、例外を発生させることで、呼び出し元の SPL プログラム内の処理を意図的に中止できます。例外は、ExceptionHandling<exception_handling>で説明されているのと同じ方法で処理されます。さらに、‘RAISE_APPLICATION_ERROR‘ プロシージャは、ユーザー定義のコードとエラーメッセージをプログラムで使用できるようにし、それを使用して例外を識別することができます。

```
RAISE_APPLICATION_ERROR(<error_number>, <message>);
```

どこで:

error_number は、プロシージャの実行時に SQLCODE という名前の変数に返される整数値または式です。error_number は -20000 と -20999 の間の値でなければなりません。

message は、SQLERRM という名前の変数で返される文字列リテラルまたは式です。

SQLCODE および SQLERRM 変数の詳細については、[ErrorsandMessages](#)、*ErrorsandMessages* を参照してください。

次の例では、RAISE_APPLICATION_ERROR プロシージャを使用して、従業員から欠落している情報に応じて異なるコードとメッセージを表示します。

```
CREATE OR REPLACE PROCEDURE verify_emp (
    p_empno          NUMBER
)
IS
    v_ename          emp.ename%TYPE;
    v_job            emp.job%TYPE;
    v_mgr            emp.mgr%TYPE;
    v_hiredate        emp.hiredate%TYPE;
BEGIN
    SELECT ename, job, mgr, hiredate
        INTO v_ename, v_job, v_mgr, v_hiredate FROM emp
        WHERE empno = p_empno;
    IF v_ename IS NULL THEN
        RAISE_APPLICATION_ERROR(-20010, 'No name for ' || p_empno);
    END IF;
    IF v_job IS NULL THEN
        RAISE_APPLICATION_ERROR(-20020, 'No job for' || p_empno);
    END IF;
    IF v_mgr IS NULL THEN
        RAISE_APPLICATION_ERROR(-20030, 'No manager for ' || p_empno);
    END IF;
    IF v_hiredate IS NULL THEN
        RAISE_APPLICATION_ERROR(-20040, 'No hire date for ' || p_empno);
    END IF;
```

(次のページに続く)

(前のページからの続き)

```
        DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||  
        ' validated without errors');  
EXCEPTION  
    WHEN OTHERS THEN  
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);  
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);  
END;
```

次の例は、従業員レコードにマネージャー番号が欠落している場合の出力を示しています。

```
EXEC verify_emp(7839);  
  
SQLCODE: -20030  
SQLERRM: EDB-20030: No manager for 7839
```

第 6 章

Transaction Control

データベースへのすべての更新が正常に行われることが望ましい場合や、エラーが発生した場合はまったく更新されないことが望ましい場合があります。すべてが単一のユニットとして正常に発生する、またはまったく発生しないデータベース更新のセットは、トランザクションと呼ばれます。

銀行業務の一般的な例は、2つの口座間の資金移動です。トランザクションの2つの部分は、1つの口座からの資金の引き出しと、別の口座への資金の預け入れです。このトランザクションの両方の部分が発生しないと、銀行の帳簿の残高が不足します。入出金は1つのトランザクションです。

次の条件が満たされている場合、Oracle データベースと互換性のあるスタイルのトランザクション制御を使用する SPL アプリケーションを作成できます。

- `edb_stmt_level_tx` パラメータを `TRUE` に設定する必要があります。これにより、例外が発生した場合に `BEGIN/END` ブロック内のすべてのデータベース更新を無条件にロールバックするアクションが防止されます。
- アプリケーションを自動コミットモードで実行してはなりません。自動コミットモードがオンの場合、成功した各データベース更新はすぐにコミットされ、元に戻すことはできません。自動コミットモードをオンまたはオフにする方法は、アプリケーションによって異なります。

SPL プログラムで最初の SQL コマンドが検出されると、トランザクションが開始されます。後続のすべての SQL コマンドは、そのトランザクションの一部として含まれます。次のいずれかが発生すると、トランザクションは終了します。

- 未処理の例外が発生します。この場合、トランザクション中に行われたすべてのデータベース更新の影響がロールバックされ、トランザクションが中止されます。
- `COMMIT` コマンドが検出された場合、トランザクション中に行われたすべてのデータベース更新の効果は永続的になります。
- `ROLLBACK` コマンドが検出された場合、トランザクション中に行われたすべてのデータベース更新の効果がロールバックされ、トランザクションが中止されます。新しい SQL コマンドが検出されると、新しいトランザクションが開始されます。

- 呼び出し元のアプリケーション（Java、PSQL など）に制御が戻ります。この場合、アプリケーションのアクションは、トランザクションが `PRAGMA AUTONOMOUS_TRANSACTION` が宣言されているブロック内
にない限り、トランザクションをコミットまたはロールバックするかどうかを決定します。呼び出しプログラ
ムとは無関係にトランザクションのコミットまたはロールバックが発生する場合。

注釈: Oracle とは異なり、`CREATE TABLE` などの DDL コマンドは、独自のトランザクション内で暗黙的に発
生しません。したがって、DDL コマンドは Oracle のように自動的にデータベースを即座にコミットすることとはな
く、DDL コマンドは DML コマンドのようにロールバックされる場合があります。

トランザクションは 1 つ以上の `BEGIN/END` ブロックにまたがるか、単一の `BEGIN/END` ブロックに 1 つ以上の
トランザクションが含まれる場合があります。

次のセクションでは、`COMMIT` および `ROLLBACK` コマンドについて詳しく説明します。

6.1 COMMIT

`COMMIT` コマンドは、現在のトランザクション中に行われたすべてのデータベース更新を永続的にし、現在のトラ
ンザクションを終了します。

```
COMMIT [ WORK ];
```

`COMMIT` コマンドは、匿名ブロック、ストアードプロシージャ、または関数内で使用できます。SPL プログラム内
では、実行可能セクションおよび/または例外セクションに表示される場合があります。

次の例では、匿名ブロックの 3 番目の `INSERT` コマンドはエラーになります。最初の 2 つの `INSERT` コマンドの
効果は、最初の `SELECT` コマンドが示すように保持されます。`ROLLBACK` コマンドを発行した後でも、2 行が実
際にコミットされたことを確認する 2 番目の `SELECT` コマンドが示すように、テーブルに残ります。

注釈: 以下の例に示す `edb_stmt_level_tx` 構成パラメータは、`ALTER DATABASE` コマンドを使用してデー
タベース全体に設定するか、`postgresql.conf` ファイルで変更してデータベースサーバー全体に設定でき
ます。

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

BEGIN
    INSERT INTO dept VALUES (50, 'FINANCE', 'DALLAS');
    INSERT INTO dept VALUES (60, 'MARKETING', 'CHICAGO');
    COMMIT;
    INSERT INTO dept VALUES (70, 'HUMAN RESOURCES', 'CHICAGO');
```

(次のページに続く)

(前のページからの続き)

```
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;
```

```
SQLERRM: value too long for type character varying(14)
SQLCODE: 22001
```

```
SELECT * FROM dept;
```

deptno	dname	loc
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
50	FINANCE	DALLAS
60	MARKETING	CHICAGO

(6 rows)

```
ROLLBACK;
```

```
SELECT * FROM dept;
```

deptno	dname	loc
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
50	FINANCE	DALLAS
60	MARKETING	CHICAGO

(6 rows)

6.2 ROLLBACK

ROLLBACK コマンドは、現在のトランザクション中に行われたすべてのデータベース更新を取り消し、現在のトランザクションを終了します。

```
ROLLBACK [ WORK ];
```

ROLLBACK コマンドは、匿名ブロック、ストアードプロシージャ、または関数内で使用できます。SPL プログラム内では、実行可能セクションおよび/または例外セクションに表示される場合があります。

次の例では、例外セクションに ROLLBACK コマンドが含まれています。最初の 2 つの INSERT コマンドは正常に

実行されますが、3 番目のコマンドでは例外が発生し、匿名ブロック内のすべての INSERT コマンドがロールバックされます。

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

BEGIN
    INSERT INTO dept VALUES (50, 'FINANCE', 'DALLAS');
    INSERT INTO dept VALUES (60, 'MARKETING', 'CHICAGO');
    INSERT INTO dept VALUES (70, 'HUMAN RESOURCES', 'CHICAGO');
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;
```

SQLERRM: value too long for type character varying(14)
SQLCODE: 22001

```
SELECT * FROM dept;
```

deptno	dname	loc
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

(4 rows)

以下は、COMMIT と ROLLBACK の両方を使用したより複雑な例です。最初に、新しい従業員を挿入する次のストアプロシージャが作成されます。

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

CREATE OR REPLACE PROCEDURE emp_insert (
    p_empno          IN emp.empno%TYPE,
    p_ename          IN emp.ename%TYPE,
    p_job            IN emp.job%TYPE,
    p_mgr            IN emp.mgr%TYPE,
    p_hiredate       IN emp.hiredate%TYPE,
    p_sal            IN emp.sal%TYPE,
    p_comm           IN emp.comm%TYPE,
    p_deptno         IN emp.deptno%TYPE
)
IS
BEGIN
```

(次のページに続く)

(前のページからの続き)

```

INSERT INTO emp VALUES (
    p_empno,
    p_ename,
    p_job,
    p_mgr,
    p_hiredate,
    p_sal,
    p_comm,
    p_deptno);

DBMS_OUTPUT.PUT_LINE('Added employee...');
DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
DBMS_OUTPUT.PUT_LINE('Name      : ' || p_ename);
DBMS_OUTPUT.PUT_LINE('Job      : ' || p_job);
DBMS_OUTPUT.PUT_LINE('Manager  : ' || p_mgr);
DBMS_OUTPUT.PUT_LINE('Hire Date : ' || p_hiredate);
DBMS_OUTPUT.PUT_LINE('Salary   : ' || p_sal);
DBMS_OUTPUT.PUT_LINE('Commission : ' || p_comm);
DBMS_OUTPUT.PUT_LINE('Dept #    : ' || p_deptno);
DBMS_OUTPUT.PUT_LINE('-----');
END;

```

このプロシージャには例外セクションがないため、発生する可能性のあるエラーは呼び出し元プログラムまで伝播されることに注意してください。

次の匿名ブロックが実行されます。例外セクションの `emp_insert` プロシージャと `ROLLBACK` コマンドへのすべての呼び出しの後、`COMMIT` コマンドの使用に注意してください。

```

BEGIN
    emp_insert(9601, 'FARRELL', 'ANALYST', 7902, '03-MAR-08', 5000, NULL, 40);
    emp_insert(9602, 'TYLER', 'ANALYST', 7900, '25-JAN-08', 4800, NULL, 40);
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('An error occurred - roll back inserts');
        ROLLBACK;
END;

Added employee...
Employee # : 9601
Name      : FARRELL
Job      : ANALYST
Manager  : 7902
Hire Date : 03-MAR-08 00:00:00
Salary   : 5000
Commission :

```

(次のページに続く)

(前のページからの続き)

```

Dept #      : 40
-----
Added employee...
Employee #  : 9602
Name       : TYLER
Job        : ANALYST
Manager    : 7900
Hire Date  : 25-JAN-08 00:00:00
Salary     : 4800
Commission :
Dept #     : 40
-----

```

次の SELECT コマンドは、従業員 Farrell と Tyler が正常に追加されたことを示しています。

```
SELECT * FROM emp WHERE empno > 9600;
```

empno	ename	job	mgr	hiredate	sal	comm	deptno
9601	FARRELL	ANALYST	7902	03-MAR-08 00:00:00	5000.00		40
9602	TYLER	ANALYST	7900	25-JAN-08 00:00:00	4800.00		40

(2 rows)

ここで、次の匿名ブロックを実行します。

```

BEGIN
    emp_insert(9603, 'HARRISON', 'SALESMAN', 7902, '13-DEC-07', 5000, 3000, 20);
    emp_insert(9604, 'JARVIS', 'SALESMAN', 7902, '05-MAY-08', 4800, 4100, 11);
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('An error occurred - roll back inserts');
        ROLLBACK;
END;

Added employee...
Employee #  : 9603
Name       : HARRISON
Job        : SALESMAN
Manager    : 7902
Hire Date  : 13-DEC-07 00:00:00
Salary     : 5000
Commission : 3000
Dept #     : 20
-----
SQLERRM: insert or update on table "emp" violates foreign key constraint

```

(次のページに続く)

(前のページからの続き)

```
"emp_ref_dept_fk"
An error occurred - roll back inserts
```

テーブルに対して SELECT コマンドを実行すると、次の結果が得られます。

```
SELECT * FROM emp WHERE empno > 9600;
```

empno	ename	job	mgr	hiredate	sal	comm	deptno
9601	FARRELL	ANALYST	7902	03-MAR-08 00:00:00	5000.00		40
9602	TYLER	ANALYST	7900	25-JAN-08 00:00:00	4800.00		40

(2 rows)

例外セクションの ROLLBACK コマンドは、従業員 Harrison の挿入を正常に取り消します。また、従業員 Farrell と Tyler は、最初の匿名ブロックの COMMIT コマンドによって挿入が永続化されたため、まだテーブルにいることに注意してください。

注釈: ランタイムスタックに Oracle スタイルの SPL プロシージャがある場合、plpgsql プロシージャで COMMIT または ROLLBACK を実行するとエラーがスローされます。

6.3 PRAGMA AUTONOMOUS_TRANSACTION

SPL プログラムは、SPL ブロックの宣言セクションで次のディレクティブを指定することにより、自律型トランザクションとして宣言できます。

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

自律型トランザクション は、呼び出しプログラム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内の SQL コマンドのコミットまたはロールバックは、呼び出し元プログラムのトランザクションのコミットまたはロールバックには影響しません。呼び出し側プログラムでのコミットまたはロールバックは、自律型トランザクションでの SQL コマンドのコミットまたはロールバックには影響しません。

次の SPL プログラムには PRAGMA AUTONOMOUS_TRANSACTION: を含めることができます

- スタンドアロンの手順と機能
- 匿名ブロック
- パッケージ内のサブプログラムおよび他の呼び出しプロシージャ、関数、および匿名ブロックとして宣言されたプロシージャと関数
- トリガー

- オブジェクト型のメソッド

以下は、自律型トランザクションに関連する問題と制限です。

- 各自律型トランザクションは、進行中の接続スロットを消費します。場合によっては、これは `postgresql.conf` ファイルの `max_connections` パラメータを上げる必要があることを意味する場合があります。
- ほとんどの点で、自律型トランザクションは完全に別個のセッションであるかのように動作しますが、GUC（つまり、SET で設定された設定）は意図的な例外です。自律型トランザクションは周囲の値を吸収し、コミットした値を外部トランザクションに伝播できます。
- 自律型トランザクションはネストできますが、単一セッション内の自律型トランザクションには 16 レベルの制限があります。
- 自律型トランザクション内では、並列クエリはサポートされていません。
- AdvancedServer の自律型トランザクションの実装は、SPL ブロックの最後にコミットされていないトランザクションがある場合、AdvancedServer 自律型トランザクションがエラーを生成しないため、Oracle データベースと完全に互換性がありません。

次の一連の例は、自律型トランザクションの使用法を示しています。このシナリオの最初のセットは、自律型トランザクションがない場合のデフォルトの動作を示しています。

各シナリオの前に、dept テーブルは次の初期値にリセットされます。

```
SELECT * FROM dept;
```

deptno	dname	loc
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

(4 rows)

Scenario 1a – No autonomous transactions with only a final COMMIT

このシナリオの最初のセットは、トランザクションの最初の BEGIN コマンドの直後から開始し、開始トランザクション内の匿名ブロック内から、最後に匿名ブロック内から実行されたストアードプロシージャからの 3 行の挿入を示しています。

ストアードプロシージャは次のとおりです。

```
CREATE OR REPLACE PROCEDURE insert_dept_70 IS
BEGIN
    INSERT INTO dept VALUES (70, 'MARKETING', 'LOS ANGELES');
END;
```

PSQL セッションは次のとおりです。

```

BEGIN;
INSERT INTO dept VALUES (50,'HR','DENVER');
BEGIN
    INSERT INTO dept VALUES (60,'FINANCE','CHICAGO');
    insert_dept_70;
END;
COMMIT;

```

最終コミット後、3 行すべてが挿入されます。

```

SELECT * FROM dept ORDER BY 1;

 deptno |    dname    |    loc
-----+-----+-----
      10 | ACCOUNTING  | NEW YORK
      20 | RESEARCH    | DALLAS
      30 | SALES       | CHICAGO
      40 | OPERATIONS  | BOSTON
      50 | HR          | DENVER
      60 | FINANCE     | CHICAGO
      70 | MARKETING   | LOS ANGELES
(7 rows)

```

Scenario 1b – No autonomous transactions, but a final ROLLBACK

次のシナリオは、すべての挿入後に最後の ROLLBACK コマンドを実行すると、3 つの挿入すべてがロールバックされることを示しています。

```

BEGIN;
INSERT INTO dept VALUES (50,'HR','DENVER');
BEGIN
    INSERT INTO dept VALUES (60,'FINANCE','CHICAGO');
    insert_dept_70;
END;
ROLLBACK;

SELECT * FROM dept ORDER BY 1;

 deptno |    dname    |    loc
-----+-----+-----
      10 | ACCOUNTING  | NEW YORK
      20 | RESEARCH    | DALLAS
      30 | SALES       | CHICAGO
      40 | OPERATIONS  | BOSTON
(4 rows)

```

Scenario 1c – No autonomous transactions, but anonymous block ROLLBACK

匿名ブロックの最後に ROLLBACK コマンドを指定すると、以前の 3 つの挿入すべてが削除されます。

```
BEGIN;
INSERT INTO dept VALUES (50, 'HR', 'DENVER');
BEGIN
    INSERT INTO dept VALUES (60, 'FINANCE', 'CHICAGO');
    insert_dept_70;
    ROLLBACK;
END;
COMMIT;

SELECT * FROM dept ORDER BY 1;
```

deptno	dname	loc
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

(4 rows)

次の一連のシナリオは、さまざまな場所で PRAGMA AUTONOMOUS_TRANSACTION を使用して自律型トランザクションを使用した場合の効果を示しています。

Scenario 2a – Autonomous transaction of anonymous block with COMMIT

手順は最初に作成されたままです。

```
CREATE OR REPLACE PROCEDURE insert_dept_70 IS
BEGIN
    INSERT INTO dept VALUES (70, 'MARKETING', 'LOS ANGELES');
END;
```

これで、匿名ブロックの最後に COMMIT コマンドとともに PRAGMA AUTONOMOUS_TRANSACTION が匿名ブロックとともに表示されます。

```
BEGIN;
INSERT INTO dept VALUES (50, 'HR', 'DENVER');
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO dept VALUES (60, 'FINANCE', 'CHICAGO');
    insert_dept_70;
    COMMIT;
END;
ROLLBACK;
```

トランザクションの最後の ROLLBACK の後、トランザクションの最初の最初の行の挿入のみが破棄されます。

PRAGMA AUTONOMOUS_TRANSACTION の匿名ブロック内の他の 2 行の挿入は、個別にコミットされています。

```
SELECT * FROM dept ORDER BY 1;
```

deptno	dname	loc
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
60	FINANCE	CHICAGO
70	MARKETING	LOS ANGELES

(6 rows)

Scenario 2b – Autonomous transaction anonymous block with COMMIT including procedure with ROLLBACK, but not an autonomous transaction procedure

現在、プロシージャには ROLLBACK コマンドが最後にあります。ただし、PRAGMA ANONYMOUS_TRANSACTION はこの手順に含まれていないことに注意してください。

```
CREATE OR REPLACE PROCEDURE insert_dept_70 IS
BEGIN
    INSERT INTO dept VALUES (70, 'MARKETING', 'LOS ANGELES');
    ROLLBACK;
END;
```

これで、プロシージャ内のロールバックにより、匿名ブロック内の最後の COMMIT コマンドの前に匿名ブロック内に挿入された 2 つの行 (deptno 60 および 70) が削除されます。

```
BEGIN;
INSERT INTO dept VALUES (50, 'HR', 'DENVER');
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO dept VALUES (60, 'FINANCE', 'CHICAGO');
    insert_dept_70;
    COMMIT;
END;
COMMIT;
```

トランザクション終了時の最終コミットの後、挿入される行はトランザクション開始から最初の行のみです。匿名ブロックは自律型トランザクションであるため、囲まれたプロシージャ内のロールバックは、匿名ブロックが実行される前に発生する挿入には影響しません。

```
SELECT * FROM dept ORDER by 1;
```

deptno	dname	loc
--------	-------	-----

(次のページに続く)

(前のページからの続き)

```

-----+-----+-----
 10 | ACCOUNTING | NEW YORK
 20 | RESEARCH   | DALLAS
 30 | SALES      | CHICAGO
 40 | OPERATIONS | BOSTON
 50 | HR         | DENVER
(5 rows)

```

Scenario 2c – Autonomous transaction anonymous block with COMMIT including procedure with ROLLBACK that is also an autonomous transaction procedure

現在、最後に ROLLBACK コマンドを使用した手順にも PRAGMA ANONYMOUS_TRANSACTION が含まれています。これにより、プロシージャ内の ROLLBACK コマンドの効果が分離されます。

```

CREATE OR REPLACE PROCEDURE insert_dept_70 IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO dept VALUES (70,'MARKETING','LOS ANGELES');
    ROLLBACK;
END;

```

現在、プロシージャ内のロールバックは、プロシージャによって挿入された行を削除しますが、匿名ブロック内に挿入された他の行は削除しません。

```

BEGIN;
INSERT INTO dept VALUES (50,'HR','DENVER');
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO dept VALUES (60,'FINANCE','CHICAGO');
    insert_dept_70;
    COMMIT;
END;
COMMIT;

```

トランザクションの最後の最後のコミットの後、挿入される行は、トランザクションの最初から最初の行であり、匿名ブロックの最初に挿入される行です。ロールバックされる唯一の挿入は、プロシージャ内の挿入です。

```

SELECT * FROM dept ORDER by 1;

deptno |  dname   |  loc
-----+-----+-----
 10 | ACCOUNTING | NEW YORK
 20 | RESEARCH   | DALLAS
 30 | SALES      | CHICAGO
 40 | OPERATIONS | BOSTON

```

(次のページに続く)

(前のページからの続き)

```

50 | HR          | DENVER
60 | FINANCE     | CHICAGO
(6 rows)

```

以下のセクションでは、他のいくつかの SPL プログラムタイプの PRAGMA AUTONOMOUS_TRANSACTION の例を示します。

Autonomous Transaction Trigger

次の例は、PRAGMA AUTONOMOUS_TRANSACTION でトリガーを宣言した結果を示しています。

emp テーブルへの変更を記録するために、次のテーブルが作成されます。

```

CREATE TABLE empauditlog (
    audit_date      DATE,
    audit_user      VARCHAR2(20),
    audit_desc      VARCHAR2(20)
);

```

これらの変更を empauditlog テーブルに挿入する emp テーブルに接続されているトリガーは次のとおりです。宣言セクションに PRAGMA AUTONOMOUS_TRANSACTION が含まれていることに注意してください。

```

CREATE OR REPLACE TRIGGER emp_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
    v_action      VARCHAR2(20);
BEGIN
    IF INSERTING THEN
        v_action := 'Added employee(s)';
    ELSIF UPDATING THEN
        v_action := 'Updated employee(s)';
    ELSIF DELETING THEN
        v_action := 'Deleted employee(s)';
    END IF;
    INSERT INTO empauditlog VALUES (SYSDATE, USER,
        v_action);
END;

```

次の 2 つの挿入は、BEGIN コマンドによって開始されたトランザクション内の emp テーブルに行われます。

```

BEGIN;
INSERT INTO emp VALUES (9001, 'SMITH', 'ANALYST', 7782, SYSDATE, NULL, NULL, 10);
INSERT INTO emp VALUES (9002, 'JONES', 'CLERK', 7782, SYSDATE, NULL, NULL, 10);

```

以下に、emp テーブルの 2 つの新しい行と empauditlog テーブルの 2 つのエントリを示します。

```
SELECT * FROM emp WHERE empno > 9000;
```

empno	ename	job	mgr	hiredate	sal	comm	deptno
9001	SMITH	ANALYST	7782	23-AUG-18 07:12:27			10
9002	JONES	CLERK	7782	23-AUG-18 07:12:27			10

(2 rows)

```
SELECT TO_CHAR(AUDIT_DATE, 'DD-MON-YY HH24:MI:SS') AS "audit date",
       audit_user, audit_desc FROM empauditlog ORDER BY 1 ASC;
```

audit date	audit_user	audit_desc
23-AUG-18 07:12:27	enterprisedb	Added employee(s)
23-AUG-18 07:12:27	enterprisedb	Added employee(s)

(2 rows)

ただし、このセッション中に ROLLBACK コマンドが表示されます。emp テーブルには 2 つの行が含まれなくなりましたが、トリガーが暗黙的にコミットを実行し、PRAGMA AUTONOMOUS_TRANSACTION が呼び出しトランザクションで指定されたロールバックとは無関係にこれらの変更をコミットするため、empauditlog テーブルにはまだ 2 つのエントリが含まれます。

```
ROLLBACK;
```

```
SELECT * FROM emp WHERE empno > 9000;
```

empno	ename	job	mgr	hiredate	sal	comm	deptno
-------	-------	-----	-----	----------	-----	------	--------

(0 rows)

```
SELECT TO_CHAR(AUDIT_DATE, 'DD-MON-YY HH24:MI:SS') AS "audit date",
       audit_user, audit_desc FROM empauditlog ORDER BY 1 ASC;
```

audit date	audit_user	audit_desc
23-AUG-18 07:12:27	enterprisedb	Added employee(s)
23-AUG-18 07:12:27	enterprisedb	Added employee(s)

(2 rows)

Autonomous Transaction Object Type Method

次の例は、PRAGMA AUTONOMOUS_TRANSACTION. でオブジェクトメソッドを宣言した結果を示しています

次のオブジェクトタイプとオブジェクトタイプボディが作成されます。オブジェクト型本体内のメンバープロシージャには、宣言セクションに PRAGMA AUTONOMOUS_TRANSACTION が含まれ、プロシージャの最後に COMMIT が含まれています。

```

CREATE OR REPLACE TYPE insert_dept_typ AS OBJECT (
    deptno          NUMBER(2),
    dname           VARCHAR2(14),
    loc             VARCHAR2(13),
    MEMBER PROCEDURE insert_dept
);

CREATE OR REPLACE TYPE BODY insert_dept_typ AS
    MEMBER PROCEDURE insert_dept
    IS
        PRAGMA AUTONOMOUS_TRANSACTION;
    BEGIN
        INSERT INTO dept VALUES (SELF.deptno, SELF.dname, SELF.loc);
        COMMIT;
    END;
END;

```

次の匿名ブロックでは、dept テーブルへの挿入が実行され、その後にオブジェクトの insert_dept メソッドが呼び出され、匿名ブロックで ROLLBACK コマンドで終了します。

```

BEGIN;
DECLARE
    v_dept          INSERT_DEPT_TYP :=
                    insert_dept_typ(60, 'FINANCE', 'CHICAGO');
BEGIN
    INSERT INTO dept VALUES (50, 'HR', 'DENVER');
    v_dept.insert_dept;
    ROLLBACK;
END;

```

insert_dept は自律型トランザクションとして宣言されているため、部門番号 60 の挿入はテーブルに残りますが、ロールバックは部門 50 の挿入を削除します。

```

SELECT * FROM dept ORDER BY 1;

 deptno |    dname    |    loc
-----+-----+-----
      10 | ACCOUNTING | NEW YORK
      20 | RESEARCH   | DALLAS
      30 | SALES       | CHICAGO
      40 | OPERATIONS | BOSTON
      60 | FINANCE     | CHICAGO
(5 rows)

```


第 7 章

Dynamic SQL

動的 SQL は、コマンドが実行されようとするまで知られていない SQL コマンドを実行する機能を提供する手法です。これまで、SPL プログラムで説明された SQL コマンドは静的 SQL でした-プログラム自体が実行を開始する前に、完全なコマンド（変数を除く）を認識してプログラムにコーディングする必要があります。したがって、動的 SQL を使用すると、実行された SQL はプログラムの実行中に変更される可能性があります。

さらに、動的 SQL は、CREATE TABLE などのデータ定義コマンドを SPL プログラム内から実行できる唯一の方法です。

ただし、動的 SQL の実行時パフォーマンスは静的 SQL よりも遅くなることに注意してください。

EXECUTE IMMEDIATE コマンドは、SQL コマンドを動的に実行するために使用されます。

```
EXECUTE IMMEDIATE '<sql_expression>;'
  [ INTO { <variable> [, ...] | <record> } ]
  [ USING <expression> [, ...] ]
```

「11」は、動的に実行される SQL コマンドを含む文字列式です。variable は、通常「11」で SQL コマンドを実行した結果として作成された SELECT コマンドから結果セットの出力を受け取ります。変数の数、順序、およびタイプは、数、順序と一致し、結果セットのフィールドとタイプ互換性がある必要があります。または、レコードのフィールドが番号、順序に一致し、結果セットと型互換性がある限り、record を指定できます。INTO 句を使用する場合、結果セットで正確に 1 行を返す必要があります。そうしないと、例外が発生します。USING 句を使用すると、expression の値が*プレースホルダー*に渡されます。プレースホルダーは、変数を使用できる「11」の SQL コマンド内に埋め込まれて表示されます。プレースホルダーは、コロン (:) プレフィックス-:name の付いた識別子で示されます。評価された式の数、順序、および結果のデータ型は、番号、順序と一致し、「11」のプレースホルダーと型互換性がある必要があります。プレースホルダーは SPL プログラムのどこにも宣言されていないことに注意してください。「11」にのみ表示されます。

次の例は、基本的な動的 SQL コマンドを文字列リテラルとして示しています。

```
DECLARE
  v_sql          VARCHAR2 (50);
```

(次のページに続く)

(前のページからの続き)

```

BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE job (jobno NUMBER(3), ' ||
        ' jname VARCHAR2(9))';
    v_sql := 'INSERT INTO job VALUES (100, 'ANALYST')';
    EXECUTE IMMEDIATE v_sql;
    v_sql := 'INSERT INTO job VALUES (200, 'CLERK')';
    EXECUTE IMMEDIATE v_sql;
END;

```

次の例は、SQL 文字列のプレースホルダーに値を渡すための USING 句を示しています。

```

DECLARE
    v_sql          VARCHAR2(50) := 'INSERT INTO job VALUES ' ||
        ' (:p_jobno, :p_jname)';
    v_jobno        job.jobno%TYPE;
    v_jname        job.jname%TYPE;
BEGIN
    v_jobno := 300;
    v_jname := 'MANAGER';
    EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
    v_jobno := 400;
    v_jname := 'SALESMAN';
    EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
    v_jobno := 500;
    v_jname := 'PRESIDENT';
    EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
END;

```

次の例は、INSERT 句と USING 句の両方を示しています。SELECT コマンドを最後に実行すると、個々の変数ではなくレコードに結果が返されることに注意してください。

```

DECLARE
    v_sql          VARCHAR2(60);
    v_jobno        job.jobno%TYPE;
    v_jname        job.jname%TYPE;
    r_job          job%ROWTYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('JOBNO      JNAME');
    DBMS_OUTPUT.PUT_LINE('-----');
    v_sql := 'SELECT jobno, jname FROM job WHERE jobno = :p_jobno';
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 100;
    DBMS_OUTPUT.PUT_LINE(v_jobno || ' ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 200;
    DBMS_OUTPUT.PUT_LINE(v_jobno || ' ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 300;
    DBMS_OUTPUT.PUT_LINE(v_jobno || ' ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 400;

```

(次のページに続く)

(前のページからの続き)

```
DBMS_OUTPUT.PUT_LINE(v_jobno || ' ' || v_jname);  
EXECUTE IMMEDIATE v_sql INTO r_job USING 500;  
DBMS_OUTPUT.PUT_LINE(r_job.jobno || ' ' || r_job.jname);  
END;
```

以下は、前の匿名ブロックからの出力です。

JOBNO	JNAME
100	ANALYST
200	CLERK
300	MANAGER
400	SALESMAN
500	PRESIDENT

BULK COLLECT 句を使用して、EXECUTE IMMEDIATE ステートメントの結果セットを名前付きコレクションにアセンブルできます。BULK COLLECT 句の使用については、[:ref:「BULKCOLLECT 句の使用<using_the_bulk_collect_clause>」](#)、EXECUTE IMMEDIATE BULK COLLECT を参照してください。

第 8 章

Static Cursors

クエリ全体を一度に実行するのではなく、クエリをカプセル化する*カーソル*を設定し、クエリ結果セットを一度に 1 行ずつ読み取ることができます。これにより、結果セットから行を取得し、その行のデータを処理してから次の行を取得してプロセスを繰り返す SPL プログラムロジックを作成できます。

カーソルは、FOR または WHILE ループのコンテキストで最もよく使用されます。プログラムがループを終了できるように、結果セットの最後に到達したことを検出する条件付きテストを SPL ロジックに含める必要があります。

8.1 Declaring a Cursor

カーソルを使用するには、最初に SPL プログラムの宣言セクションで宣言する必要があります。カーソル宣言は次のように表示されます。

```
CURSOR <name> IS <query>;
```

name は、後のプログラムでカーソルとその結果セットを参照するために使用される識別子です。query は、カーソルで取得可能な結果セットを決定する SQL SELECT コマンドです。

注釈: この構文を拡張すると、パラメーターを使用できます。これについては、[ParameterizedCursors](#) で詳しく説明します。

以下は、カーソル宣言の例です。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_1 IS SELECT * FROM emp;
    CURSOR emp_cur_2 IS SELECT empno, ename FROM emp;
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
```

(次のページに続く)

(前のページからの続き)

```
BEGIN
    ...
END;
```

8.2 Opening a Cursor

カーソルを使用して行を取得するには、まずカーソルを開く必要があります。これは OPEN ステートメントで実現されます。

```
OPEN <name>;
```

name は、SPL プログラムの宣言セクションで以前に宣言されたカーソルの識別子です。OPEN ステートメントは、すでに開いていてまだ開いているカーソルに対して実行しないでください。

以下は、OPEN ステートメントとそれに対応するカーソル宣言を示しています。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    ...
END;
```

8.3 Fetching Rows From a Cursor

カーソルを開くと、FETCH ステートメントを使用して、カーソルの結果セットから行を取得できます。

```
FETCH <name> INTO { <record> | <variable> [, <variable_2> ]... };
```

name は、以前に開いたカーソルの識別子です。record は、以前に定義されたレコードの識別子です（たとえば、table%ROWTYPE を使用）。variable, variable_2... は、フェッチされた行からフィールドデータを受け取る SPL 変数です。record または variable, variable_2... のフィールドは、カーソル宣言で指定されたクエリの SELECT リストに返されるフィールドの数と順序で一致する必要があります。SELECT リストのフィールドのデータ型は、record のフィールドのデータ型または variable, variable_2... のデータ型と一致するか、暗黙的に変換可能である必要があります

注釈: 一度に複数の行をコレクションに返すことができる BULK COLLECT 句を使用する FETCH INTO のバリエーションがあります。FETCH INTO ステートメントで BULK COLLECT 句を使用する方法の詳細については、

セクション:ref:「BULKCOLLECT 句の使用<using_the_bulk_collect_clause>」を参照してください。

以下に FETCH ステートメントを示します。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno          NUMBER(4);
    v_ename          VARCHAR2(10);
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
    ...
END;
```

ターゲット変数のデータ型を明示的に宣言する代わりに、%TYPE を代わりに使用できます。このようにして、データベース列のデータ型が変更された場合、SPL プログラムのターゲット変数宣言を変更する必要はありません。%TYPE は、指定された列の新しいデータ型を自動的に取得します。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno          emp.empno%TYPE;
    v_ename          emp.ename%TYPE;
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
    ...
END;
```

テーブル内のすべての列がテーブルで定義された順序で取得される場合、%ROWTYPE を使用して、FETCH ステートメントが取得データを配置するレコードを定義できます。レコード内の各フィールドには、ドット表記を使用してアクセスできます。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec        emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    FETCH emp_cur_1 INTO v_emp_rec;
    DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
    DBMS_OUTPUT.PUT_LINE('Employee Name   : ' || v_emp_rec.ename);
    ...
END;
```

8.4 Closing a Cursor

カーソル結果セットから目的の行をすべて取得したら、カーソルを閉じる必要があります。一度閉じると、結果セットにはアクセスできなくなります。CLOSE ステートメントは次のように表示されます。

```
CLOSE <name>;
```

name は現在開いているカーソルの識別子です。カーソルを一度閉じたら、再度閉じてはいけません。ただし、カーソルが閉じられると、閉じられたカーソルに対して OPEN ステートメントが再度発行され、クエリ結果セットが再構築されます。その後、FETCH ステートメントを使用して新しい結果セットの行を取得できます。

次の例は、CLOSE ステートメントの使用法を示しています。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec      emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    FETCH emp_cur_1 INTO v_emp_rec;
    DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
    DBMS_OUTPUT.PUT_LINE('Employee Name   : ' || v_emp_rec.ename);
    CLOSE emp_cur_1;
END;
```

このプロシージャを呼び出すと、次の出力が生成されます。従業員番号 7369, SMITH は、結果セットの最初の行です。

```
EXEC cursor_example;

Employee Number: 7369
Employee Name   : SMITH
```

8.5 Cursor Attributes

各カーソルには、プログラムがカーソルの状態をテストできる一連の属性が関連付けられています。これらの属性は %ISOPEN, %FOUND, %NOTFOUND および %ROWCOUNT です。これらの属性については、次のセクションで説明します。

8.5.1 Summary of Cursor States and Attributes

次の表は、カーソルの状態とカーソル属性によって返される値をまとめたものです。

カーソル状態	% ISOPEN	% FOUND	% NOTFOUND	% ROWCOUNT
OPEN の前	偽	INVALID_CURSOR 例外	INVALID_CURSOR 例外	INVALID_CURSOR 例外
OPEN の後 と 1 日前の FETCH	本当だ	ヌル	ヌル	0
最初の成功後 FETCH	本当だ	本当だ	偽	1
nth 成功後 FETCH (最後 の行)	本当だ	本当だ	偽	n
n+1 番目の FETCH の後 (最後の行の 後)	本当だ	偽	本当だ	n
CLOSE の後	偽	INVALID_CURSOR 例外	INVALID_CURSOR 例外	INVALID_CURSOR 例外

8.6 Cursor FOR Loop

これまでに示したカーソルの例では、カーソルの結果セットを処理するために必要なプログラミングロジックには、カーソルを開くステートメント、結果セットの各行を取得するループ構造、結果セットの最後のテスト、最後に、カーソルを閉じるステートメント。*cursorFOR* ループは、リストされたステートメントを個別にコーディングする必要をなくすループ構造です。

カーソル FOR ループは、以前に宣言されたカーソルを開き、カーソル結果セット内のすべての行をフェッチしてから、カーソルを閉じます。

カーソル FOR ループを作成するための構文は次のとおりです。

```
FOR <record> IN <cursor>
LOOP
    <statements>
END LOOP;
```

record は定義付きで暗黙的に宣言されたレコードに割り当てられた識別子であり、cursor%ROWTYPE. cursor は以前に宣言されたカーソルの名前です。statements は 1 つ以上の SPL ステートメントです。少なくとも 1 つのステートメントが必要です。

次の例は、カーソル FOR ループを使用するように変更された % NOTFOUND の例を示しています。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
    DBMS_OUTPUT.PUT_LINE('-----      -');
    FOR v_emp_rec IN emp_cur_1 LOOP
        DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '      ' || v_emp_rec.ename);
    END LOOP;
END;
```

以下の出力に示すように、同じ結果が得られます。

```
EXEC cursor_example;
```

```
EMPNO      ENAME
-----      -
7369      SMITH
7499      ALLEN
7521      WARD
7566      JONES
7654      MARTIN
7698      BLAKE
7782      CLARK
7788      SCOTT
7839      KING
7844      TURNER
7876      ADAMS
7900      JAMES
7902      FORD
7934      MILLER
```

8.7 Parameterized Cursors

ユーザーは、パラメーターを受け入れる静的カーソルを宣言し、そのカーソルを開くときにそれらのパラメーターの値を渡すこともできます。次の例では、emp テーブルのすべての従業員の名前と給与を表示するパラメータ化されたカーソルを作成しました。従業員の給与は、パラメータとして渡される指定値未満です。

```
DECLARE
    my_record      emp%ROWTYPE;
    CURSOR c1 (max_wage NUMBER) IS
        SELECT * FROM emp WHERE sal < max_wage;
BEGIN
    OPEN c1(2000);
    LOOP
        FETCH c1 INTO my_record;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Name = ' || my_record.ename || ', salary = '
                               || my_record.sal);
    END LOOP;
    CLOSE c1;
END;
```

たとえば、値 2000 を max_wage として渡すと、給与が 2000 未満のすべての従業員の名前と給与のみが表示されます。上記のクエリの結果は次のとおりです。

```
Name = SMITH, salary = 800.00
Name = ALLEN, salary = 1600.00
Name = WARD, salary = 1250.00
Name = MARTIN, salary = 1250.00
Name = TURNER, salary = 1500.00
Name = ADAMS, salary = 1100.00
Name = JAMES, salary = 950.00
Name = MILLER, salary = 1300.00
```


第 9 章

REF CURSORS and Cursor Variables

このセクションでは、前述の静的カーソルよりもはるかに高い柔軟性を提供する別のタイプのカーソルについて説明します。

9.1 REF CURSOR Overview

カーソル変数は、クエリ結果セットへのポインタを実際に含むカーソルです。結果セットは、カーソル変数を使用した `OPEN FOR` ステートメントの実行によって決定されます。

カーソル変数は、静的カーソルのような単一の特定のクエリに関連付けられていません。同じカーソル変数は、異なるクエリを含む `OPEN FOR` ステートメントで何度も開くことができます。そのたびに、そのクエリから新しい結果セットが作成され、カーソル変数を介して使用可能になります。

`REF CURSOR` 型は、ストアドプロシージャおよび関数との間でパラメータとして渡すことができます。関数の戻り値の型は `REF CURSOR` 型の場合もあります。これにより、プログラム間でカーソル変数を渡すことにより、カーソルの操作を個別のプログラムにモジュール化する機能が提供されます。

9.2 Declaring a Cursor Variable

`SPL` は、`SYS_REFCURSOR` 組み込みデータ型の使用と `REF CURSOR` 型の作成とその型の変数の宣言の両方を使用したカーソル変数の宣言をサポートしています。`SYS_REFCURSOR` は `REF CURSOR` タイプで、任意の結果セットをそれに関連付けることができます。これは **weakly-typed** `REF CURSOR` として知られています。

`SYS_REFCURSOR` 変数とユーザー定義 `REF CURSOR` 変数の宣言のみが異なります。カーソルを開く、カーソルを選択する、カーソルを閉じるなどの残りの使用法は、両方のカーソルタイプで同じです。この章の残りの部分では、例では主に `SYS_REFCURSOR` カーソルを使用します。ユーザー定義の `REF CURSORs` で機能するように例を変更する必要があるのは、宣言セクションだけです。

注釈：厳密に型指定された `REF CURSORs` は、互換性のあるデータ型を持つフィールドの宣言された数と順序に

準拠する結果セットを必要とし、オプションで結果セットを返すこともできます。

9.2.1 Declaring a SYS_REFCURSOR Cursor Variable

以下は、SYS_REFCURSOR カーソル変数を宣言するための構文です。

```
<name> SYS_REFCURSOR;
```

name はカーソル変数に割り当てられた識別子です。

以下は、SYS_REFCURSOR 変数宣言の例です。

```
DECLARE
    emp_refcur      SYS_REFCURSOR;
    ...
```

9.2.2 Declaring a User Defined REF CURSOR Type Variable

ユーザー定義の REF CURSOR 変数を使用するには、2 つの異なる宣言手順を実行する必要があります。

- 参照カーソル TYPE を作成します
- その TYPE に基づいて実際のカーソル変数を宣言します

ユーザー定義の REF CURSOR タイプを作成するための構文は次のとおりです。

```
TYPE <cursor_type_name> IS REF CURSOR [RETURN <return_type>];
```

以下は、カーソル変数宣言の例です。

```
DECLARE
    TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
    my_rec emp_cur_type;
    ...
```

9.3 Opening a Cursor Variable

カーソル変数が宣言されたら、関連する SELECT コマンドで開く必要があります。OPEN FOR ステートメントは、結果セットの作成に使用される SELECT コマンドを指定します。

```
OPEN <name> FOR query;
```

name は、以前に宣言されたカーソル変数の識別子です。query は、ステートメントの実行時に結果セットを決定する SELECT コマンドです。OPEN FOR ステートメントが実行された後のカーソル変数の値は、結果セットを識別します。

次の例では、結果セットは、選択した部門の従業員番号と名前のリストです。式が通常表示される場所であればどこでも、変数またはパラメーターを SELECT コマンドで使用できることに注意してください。この場合、部門番号の同等性テストでパラメーターが使用されます。

```
CREATE OR REPLACE PROCEDURE emp_by_dept (
    p_deptno          emp.deptno%TYPE
)
IS
    emp_refcur        SYS_REFCURSOR;
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
    ...
```

9.4 Fetching Rows From a Cursor Variable

カーソル変数を開いた後、FETCH ステートメントを使用して結果セットから行を取得できます。FETCH ステートメントを使用して結果セットから行を取得する方法の詳細については、[カーソルからの行のフェッチ](#) を参照してください。

次の例では、前の例に FETCH ステートメントが追加されているため、結果セットが2つの変数に返されて表示されます。静的カーソルのカーソル状態を決定するために使用されるカーソル属性は、カーソル変数でも使用できることに注意してください。カーソル属性の詳細については、[カーソル属性](#) を参照してください。

```
CREATE OR REPLACE PROCEDURE emp_by_dept (
    p_deptno          emp.deptno%TYPE
)
IS
    emp_refcur        SYS_REFCURSOR;
    v_empno           emp.empno%TYPE;
    v_ename           emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
    DBMS_OUTPUT.PUT_LINE('-----');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '      ' || v_ename);
    END LOOP;
```

(次のページに続く)

(前のページからの続き)

```
END LOOP;
...
```

9.5 Closing a Cursor Variable

カーソルを閉じる<closing_a_cursor>‘で説明されている ‘CLOSE‘ ステートメントを使用して、結果セットを解放します。

注釈: 静的カーソルとは異なり、カーソル変数を再度開く前に閉じる必要はありません。前回のオープンからの結果セットは失われます。

この例は、CLOSE ステートメントを追加して完了します。

```
CREATE OR REPLACE PROCEDURE emp_by_dept (
    p_deptno          emp.deptno%TYPE
)
IS
    emp_refcur        SYS_REFCURSOR;
    v_empno           emp.empno%TYPE;
    v_ename           emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
    DBMS_OUTPUT.PUT_LINE('-----      -');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '      ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

以下は、この手順を実行したときの出力です。

```
EXEC emp_by_dept(20)
```

```
EMPNO      ENAME
-----      -
7369       SMITH
7566       JONES
7788       SCOTT
7876       ADAMS
7902       FORD
```

9.6 Usage Restrictions

以下は、カーソル変数の使用に関する制限です。

- 比較演算子を使用して、カーソル変数の等価性、不等価性、NULL、または非 NULL をテストすることはできません
- カーソル変数に NULL を割り当てることはできません
- カーソル変数の値はデータベース列に保存できません
- 静的カーソルとカーソル変数は交換できません。たとえば、静的カーソルは OPEN FOR ステートメントでは使用できません。

さらに、次の表は、プロシージャまたは関数内のカーソル変数の操作に応じて、プロシージャまたは関数のパラメータとして使用されるカーソル変数に許可されるパラメータモードを示しています。

操作	で	INOUT	アウト
OPEN	No	Yes	No
FETCH	Yes	Yes	No
CLOSE	Yes	Yes	No

したがって、たとえば、プロシージャの仮パラメータとして宣言されたカーソル変数に対して、プロシージャが OPEN FOR, FETCH および CLOSE の 3 つの操作をすべて実行する場合、そのパラメータは IN OUT モードで宣言する必要があります。

9.7 Examples

次の例は、カーソル変数の使用法を示しています。

9.7.1 Returning a REF CURSOR From a Function

次の例では、特定のジョブを持つ従業員を選択するクエリでカーソル変数が開かれます。カーソル変数はこの関数の RETURN ステートメントで指定されているため、関数の呼び出し元が結果セットを利用できることに注意してください。

```
CREATE OR REPLACE FUNCTION emp_by_job (p_job VARCHAR2)
RETURN SYS_REFCURSOR
IS
    emp_refcur      SYS_REFCURSOR;
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE job = p_job;
    RETURN emp_refcur;
END;
```

この関数は、関数の戻り値を匿名ブロックの宣言セクションで宣言されたカーソル変数に割り当てることにより、次の匿名ブロックで呼び出されます。結果セットはこのカーソル変数を使用してフェッチされ、閉じられます。

```
DECLARE
    v_empno          emp.empno%TYPE;
    v_ename          emp.ename%TYPE;
    v_job            emp.job%TYPE := 'SALESMAN';
    v_emp_refcur     SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES WITH JOB ' || v_job);
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
    DBMS_OUTPUT.PUT_LINE('-----      -');
    v_emp_refcur := emp_by_job(v_job);
    LOOP
        FETCH v_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN v_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '      ' || v_ename);
    END LOOP;
    CLOSE v_emp_refcur;
END;
```

以下は、匿名ブロックが実行されたときの出力です。

```
EMPLOYEES WITH JOB SALESMAN
EMPNO      ENAME
-----      -
```

(次のページに続く)

(前のページからの続き)

7499	ALLEN
7521	WARD
7654	MARTIN
7844	TURNER

9.7.2 Modularizing Cursor Operations

次の例は、カーソル変数のさまざまな操作を個別のプログラムにモジュール化する方法を示しています。

次の手順では、すべての行を取得する SELECT コマンドを使用して、指定されたカーソル変数を開きます。

```
CREATE OR REPLACE PROCEDURE open_all_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp;
END;
```

このバリエーションは、指定された部門のすべての行を取得する SELECT コマンドで、指定されたカーソル変数を開きます。

```
CREATE OR REPLACE PROCEDURE open_emp_by_dept (
    p_emp_refcur    IN OUT SYS_REFCURSOR,
    p_deptno        emp.deptno%TYPE
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp
        WHERE deptno = p_deptno;
END;
```

この 3 番目のバリエーションは、すべての行を取得する SELECT コマンドを使用して、指定されたカーソル変数を開きますが、別のテーブルからです。また、関数の戻り値は開かれたカーソル変数であることに注意してください。

```
CREATE OR REPLACE FUNCTION open_dept (
    p_dept_refcur    IN OUT SYS_REFCURSOR
) RETURN SYS_REFCURSOR
IS
    v_dept_refcur    SYS_REFCURSOR;
BEGIN
    v_dept_refcur := p_dept_refcur;
    OPEN v_dept_refcur FOR SELECT deptno, dname FROM dept;
```

(次のページに続く)

(前のページからの続き)

```

    RETURN v_dept_refcur;
END;
```

このプロシージャは、従業員番号と名前で構成されるカーソル変数の結果セットをフェッチして表示します。

```

CREATE OR REPLACE PROCEDURE fetch_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
    DBMS_OUTPUT.PUT_LINE('-----      -');
    LOOP
        FETCH p_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN p_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '      ' || v_ename);
    END LOOP;
END;
```

このプロシージャは、部門番号と名前で構成されるカーソル変数の結果セットをフェッチして表示します。

```

CREATE OR REPLACE PROCEDURE fetch_dept (
    p_dept_refcur   IN SYS_REFCURSOR
)
IS
    v_deptno        dept.deptno%TYPE;
    v_dname         dept.dname%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('DEPT      DNAME');
    DBMS_OUTPUT.PUT_LINE('-----      -');
    LOOP
        FETCH p_dept_refcur INTO v_deptno, v_dname;
        EXIT WHEN p_dept_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_deptno || '      ' || v_dname);
    END LOOP;
END;
```

このプロシージャは、指定されたカーソル変数を閉じます。

```

CREATE OR REPLACE PROCEDURE close_refcur (
    p_refcur        IN OUT SYS_REFCURSOR
)
IS
BEGIN
    CLOSE p_refcur;
```

(次のページに続く)

(前のページからの続き)

END;

次の匿名ブロックは、前述のすべてのプログラムを実行します。

```
DECLARE
    gen_refcur      SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('ALL EMPLOYEES');
    open_all_emp(gen_refcur);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('*****');

    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #10');
    open_emp_by_dept(gen_refcur, 10);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('*****');

    DBMS_OUTPUT.PUT_LINE('DEPARTMENTS');
    fetch_dept(open_dept(gen_refcur));
    DBMS_OUTPUT.PUT_LINE('*****');

    close_refcur(gen_refcur);
END;
```

以下は、匿名ブロックからの出力です。

```
ALL EMPLOYEES
EMPNO      ENAME
-----
7369       SMITH
7499       ALLEN
7521       WARD
7566       JONES
7654       MARTIN
7698       BLAKE
7782       CLARK
7788       SCOTT
7839       KING
7844       TURNER
7876       ADAMS
7900       JAMES
7902       FORD
7934       MILLER
*****
EMPLOYEES IN DEPT #10
EMPNO      ENAME
-----
```

(次のページに続く)

(前のページからの続き)

```
7782      CLARK
7839      KING
7934      MILLER
*****
DEPARTMENTS
DEPT      DNAME
-----
10        ACCOUNTING
20        RESEARCH
30        SALES
40        OPERATIONS
*****
```

9.8 Dynamic Queries With REF CURSORS

AdvancedServer は、OPEN FOR USING ステートメントによる動的クエリもサポートしています。文字列リテラルまたは文字列変数は、SELECT コマンドの OPEN FOR USING ステートメントで提供されます。

```
OPEN <name> FOR <dynamic_string>
  [ USING <bind_arg> [, <bind_arg_2> ] ...];
```

name は、以前に宣言されたカーソル変数の識別子です。dynamic_string は SELECT コマンドを含む文字列リテラルまたは文字列変数です（終了セミコロンなし）。bind_arg, bind_arg_2... は、カーソル変数が開かれたときに SELECT コマンドで対応するプレースホルダーに変数を渡すために使用されるバインド引数です。プレースホルダーは、コロン文字で始まる識別子です。

以下は、文字列リテラルを使用した動的クエリの例です。

```
CREATE OR REPLACE PROCEDURE dept_query
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = 30' ||
        ' AND sal >= 1500';
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
    DBMS_OUTPUT.PUT_LINE('-----      -----');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '      ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

以下は、プロシージャが実行されたときの出力です。

```
EXEC dept_query;

EMPNO      ENAME
-----      -----
7499       ALLEN
7698       BLAKE
7844       TURNER
```

次の例では、前のクエリを変更して、バインド引数を使用してクエリパラメータを渡します。

```
CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno         emp.deptno%TYPE,
```

(次のページに続く)

(前のページからの続き)

```

    p_sal          emp.sal%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = :dept'
        || ' AND sal >= :sal' USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
    DBMS_OUTPUT.PUT_LINE('-----      -');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '      ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

結果の出力は次のとおりです。

```
EXEC dept_query(30, 1500);
```

```

EMPNO      ENAME
-----      -
7499      ALLEN
7698      BLAKE
7844      TURNER
```

最後に、文字列変数を使用して SELECT を渡し、最も柔軟性を高めます。

```

CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno         emp.deptno%TYPE,
    p_sal            emp.sal%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    p_query_string  VARCHAR2(100);
BEGIN
    p_query_string := 'SELECT empno, ename FROM emp WHERE ' ||
        'deptno = :dept AND sal >= :sal';
    OPEN emp_refcur FOR p_query_string USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
    DBMS_OUTPUT.PUT_LINE('-----      -');
    LOOP
```

(次のページに続く)

(前のページからの続き)

```
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
EXEC dept_query(20, 1500);
```

EMPNO	ENAME
-----	-----
7566	JONES
7788	SCOTT
7902	FORD

第 10 章

Collections

コレクション は、同じデータ型の順序付けられたデータ項目のセットです。一般的に、データ項目はスカラーフィールドですが、ユーザー定義型の各フィールドを構成する構造とデータ型が同じである限り、レコード型やオブジェクト型などのユーザー定義型でもかまいません。セット内の各要素に対して。セット内の特定の各データ項目は、括弧のペア内で添え字表記を使用して参照されます。

注釈: マルチレベルコレクション（つまり、コレクションのデータ項目が別のコレクションである場合）はサポートされていません。

最も一般的に知られているコレクションのタイプは配列です。AdvancedServer では、サポートされるコレクションタイプは、連想配列*（以前は Oracle では **index-by-tables** と呼ばれていました）、**nestedtables**、および **varrays** です。

コレクションを使用するための一般的な手順は次のとおりです。

- 目的のタイプのコレクションを定義する必要があります。これは、SPL プログラムの宣言セクションで実行できます。これにより、そのプログラム内でのみアクセス可能な*ローカルタイプ*が作成されます。ネストした表および VARRAY 型の場合、これは CREATE TYPE コマンドを使用して行うこともできます。このコマンドは、データベース内の任意の SPL プログラムで参照できる永続的な*スタンドアロン型*を作成します。
- コレクション型の変数が宣言されています。変数宣言の一部として値の割り当てが行われていない場合、宣言された変数に関連付けられたコレクションは、この時点で*未初期化*と呼ばれます。
- ネストした表および VARRAY の初期化されていないコレクションは NULL です。*null* コレクションはまだ存在しません。通常、*null* コレクションでコレクションメソッドが呼び出されると、COLLECTION_IS_NULL 例外がスローされます。
- 連想配列の初期化されていないコレクションは存在しますが、要素はありません。要素のない既存のコレクションは、空のコレクションと呼ばれます。

- null コレクションを初期化するには、空のコレクションにするか、null 以外の値を割り当てる必要があります。一般に、null コレクションはその*constructor* を使用して初期化されます。
- 空の連想配列に要素を追加するには、キーに値を割り当てるだけです。ネストした表および VARRAY の場合、通常、コンストラクタを使用して、ネストした表または VARRAY に初期値を割り当てます。ネストした表および VARRAY の場合、EXTEND メソッドを使用して、コンストラクタによって設定された初期サイズを超えてコレクションを拡大します。

各コレクションタイプの特定のプロセスについては、次のセクションで説明します。

10.1 Associative Arrays

連想配列 は、一意のキーを値に関連付けるコレクションのタイプです。キーは数値である必要はありませんが、文字データでもかまいません。

連想配列には次の特性があります。

- 連想配列タイプ*を定義した後、その配列タイプの*配列変数 を宣言できます。データ操作は、配列変数を使用して行われます。
- 配列変数が宣言されると、連想配列が作成されますが、空です-キー値への値の割り当てを開始するだけです。
- キーは、任意の負の整数、正の整数、または INDEX BY BINARY_INTEGER または PLS_INTEGER が指定されている場合はゼロです。
- INDEX BY VARCHAR2 を指定すると、キーは文字データになります。
- 配列内の要素の数に事前定義された制限はありません-要素が追加されると動的に増加します。
- 配列はまばらである可能性があります-キーへの値の割り当てにギャップがある場合があります。
- 値が割り当てられていない配列要素を参照しようとすると、例外が発生します。

TYPE IS TABLE OF ... INDEX BY ステートメントは、連想配列タイプを定義するために使用されます。

```
TYPE <assoctype> IS TABLE OF { <datatype> | <rectype> | <objtype> }
    INDEX BY { BINARY_INTEGER | PLS_INTEGER | VARCHAR2(<n>) };
```

assoctype は、配列タイプに割り当てられた識別子です。datatype は、VARCHAR2 や NUMBER などのスカラーデータ型です。rectype は以前に定義されたレコードタイプです。objtype は以前に定義されたオブジェクトタイプです。n は文字キーの最大長です。

配列を使用するには、その配列タイプで*変数* を宣言する必要があります。以下は、配列変数を宣言するための構文です。

```
<array assoctype>
```

array は連想配列に割り当てられた識別子です。assoctype は、以前に定義された配列タイプの識別子です。

配列の要素は、次の構文を使用して参照されます。

```
<array>(<n>)[.<field> ]
```

array は、以前に宣言された配列の識別子です。n はキー値であり、INDEX BY 句で指定されたデータ型と型互換性があります。array の配列タイプがレコードタイプまたはオブジェクトタイプから定義されている場合、[.field] はレコードタイプ内の個々のフィールドまたは配列タイプが定義されているオブジェクトタイプ内の属性を参照する必要があります。または、[.field] を省略することでレコード全体を参照できます。

次の例では、emp テーブルから最初の 10 人の従業員名を読み取り、それらを配列に保存してから、配列の結果を表示します。

```
DECLARE
    TYPE emp_arr_typ IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
    emp_arr          emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 10;
    i                INTEGER := 0;
BEGIN
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;
END;
```

上記の例では、次の出力が生成されます。

```
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
```

前の例は、配列定義でレコードタイプを使用するように変更されました。

```

DECLARE
    TYPE emp_rec_typ IS RECORD (
        empno          NUMBER(4),
        ename          VARCHAR2(10)
    );
    TYPE emp_arr_typ IS TABLE OF emp_rec_typ INDEX BY BINARY_INTEGER;
    emp_arr            emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i                  INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
    DBMS_OUTPUT.PUT_LINE('-----      -');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '      ' ||
                               emp_arr(j).ename);
    END LOOP;
END;

```

以下は、この匿名ブロックからの出力です。

EMPNO	ENAME
-----	-----
7369	SMITH
7499	ALLEN
7521	WARD
7566	JONES
7654	MARTIN
7698	BLAKE
7782	CLARK
7788	SCOTT
7839	KING
7844	TURNER

次のように emp_rec_typ レコードタイプを使用する代わりに、emp%ROWTYPE 属性を使用して emp_arr_typ を定義できます。

```

DECLARE
    TYPE emp_arr_typ IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    emp_arr            emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i                  INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');

```

(次のページに続く)

(前のページからの続き)

```

DBMS_OUTPUT.PUT_LINE('-----      -----');
FOR r_emp IN emp_cur LOOP
    i := i + 1;
    emp_arr(i).empno := r_emp.empno;
    emp_arr(i).ename := r_emp.ename;
END LOOP;
FOR j IN 1..10 LOOP
    DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '      ' ||
        emp_arr(j).ename);
END LOOP;
END;

```

結果は前の例と同じです。

レコードの各フィールドを個別に割り当てる代わりに、レコードレベルの割り当てを `r_emp` から `emp_arr` に行うことができます。

```

DECLARE
    TYPE emp_rec_ttyp IS RECORD (
        empno      NUMBER(4),
        ename      VARCHAR2(10)
    );
    TYPE emp_arr_ttyp IS TABLE OF emp_rec_ttyp INDEX BY BINARY_INTEGER;
    emp_arr      emp_arr_ttyp;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i            INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
    DBMS_OUTPUT.PUT_LINE('-----      -----');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '      ' ||
            emp_arr(j).ename);
    END LOOP;
END;

```

連想配列のキーは、次の例に示すように文字データにすることができます。

```

DECLARE
    TYPE job_arr_ttyp IS TABLE OF NUMBER INDEX BY VARCHAR2(9);
    job_arr      job_arr_ttyp;
BEGIN
    job_arr('ANALYST') := 100;
    job_arr('CLERK')   := 200;

```

(次のページに続く)

(前のページからの続き)

```

job_arr('MANAGER')      := 300;
job_arr('SALESMAN')     := 400;
job_arr('PRESIDENT')    := 500;
DBMS_OUTPUT.PUT_LINE('ANALYST   : ' || job_arr('ANALYST'));
DBMS_OUTPUT.PUT_LINE('CLERK     : ' || job_arr('CLERK'));
DBMS_OUTPUT.PUT_LINE('MANAGER   : ' || job_arr('MANAGER'));
DBMS_OUTPUT.PUT_LINE('SALESMAN  : ' || job_arr('SALESMAN'));
DBMS_OUTPUT.PUT_LINE('PRESIDENT: ' || job_arr('PRESIDENT'));
END;

ANALYST   : 100
CLERK     : 200
MANAGER   : 300
SALESMAN  : 400
PRESIDENT : 500

```

10.2 Nested Tables

ネストされたテーブルは、正の整数を値に関連付けるコレクションのタイプです。ネストした表には、次の特性があります。

- ネストされたテーブルタイプ*を定義した後、そのネストされたテーブルタイプの*ネストされたテーブル変数*を宣言できます。データ操作は、ネストされたテーブル変数、または単に「テーブル」を使用して行われます。
- ネストされたテーブル変数が宣言されると、ネストされたテーブルは最初には存在しません（null コレクションです）。null テーブルは*constructor*で初期化する必要があります。また、割り当ての右側が同じタイプの初期化されたテーブルである割り当てステートメントを使用して、テーブルを初期化することもできます。**Note:** ネストしたテーブルの初期化は Oracle では必須ですが、SPL ではオプションです。
- キーは正の整数です。
- コンストラクターは、テーブル内の要素の数を確立します。EXTEND メソッドは、テーブルに追加の要素を追加します。コレクションメソッドの詳細については、コレクションメソッド<collection_methods>‘を参照してください。‘EXTEND‘ テーブル内の要素数を確立するためのコンストラクターの使用と、テーブルに要素を追加するための EXTEND メソッドの使用は Oracle では必須ですが、SPL ではオプションです。
- テーブルはまばらになる可能性があります-キーへの値の割り当てにギャップがある場合があります。
- 初期化または拡張されたサイズを超えてテーブル要素を参照しようとする、SUBSCRIPT_BEYOND_COUNT 例外が発生します。

TYPE IS TABLE ステートメントは、SPL プログラムの宣言セクション内でネストされたテーブルタイプを定義するために使用されます。

```
TYPE <tbltype> IS TABLE OF { <datatype> | <rectype> | <objtype> };
```

tbltype は、ネストされたテーブルタイプに割り当てられた識別子です。datatype は、VARCHAR2 や NUMBER などのスカラーデータ型です。rectype は以前に定義されたレコードタイプです。objtype は以前に定義されたオブジェクトタイプです。

注釈: CREATE TYPE コマンドを使用して、データベース内のすべての SPL プログラムで利用できるネストされたテーブルタイプを定義できます。CREATE TYPE コマンドの詳細については、『Oracle 開発者向けデータベース互換性リファレンスガイド』を参照してください。

テーブルを使用するには、そのネストされたテーブルタイプの*変数*を宣言する必要があります。以下は、テーブル変数を宣言するための構文です。

```
<table tbltype>
```

table は、ネストされたテーブルに割り当てられた識別子です。tbltype は、事前に定義されたネストされたテーブルタイプの識別子です。

ネストしたテーブルは、ネストしたテーブルタイプのコンストラクタを使用して初期化されます。

```
<tbltype> ([ { <expr1> | NULL } [, { <expr2> | NULL } ] [, ...] ])
```

tbltype はネストしたテーブルタイプのコンストラクターの識別子で、ネストしたテーブルタイプと同じ名前を持っています。expr1, expr2, ... は、テーブルの要素タイプとタイプ互換性のある式です。NULL が指定されている場合、対応する要素は null に設定されます。パラメータリストが空の場合、空のネストされたテーブルが返されます。これは、テーブルに要素がないことを意味します。テーブルがオブジェクトタイプから定義されている場合、exprn はそのオブジェクトタイプのオブジェクトを返す必要があります。オブジェクトは、関数の戻り値またはオブジェクト型のコンストラクターにすることができます。または、オブジェクトは、同じ型の別のネストしたテーブルの要素にすることができます。

EXISTS 以外のコレクションメソッドが初期化されていないネストされたテーブルに適用されると、COLLECTION_IS_NULL 例外がスローされます。コレクションメソッドの詳細については、[コレクションメソッド](#)を参照してください。

以下は、ネストしたテーブルのコンストラクターの例です。

```
DECLARE
    TYPE nested_typ IS TABLE OF CHAR(1);
    v_nested        nested_typ := nested_typ('A','B');
```

テーブルの要素は、次の構文を使用して参照されます。

```
<table>(<n>)[<.element> ]
```

table は、以前に宣言されたテーブルの識別子です。n は正の整数です。table のテーブルタイプがレコードタイプまたはオブジェクトタイプから定義されている場合、[.element] はレコードタイプ内の個々のフィールドまたはネストされたテーブルタイプが定義されているオブジェクトタイプ内の属性を参照する必要があります。または、[.element] を省略することにより、レコードまたはオブジェクト全体を参照できます。

以下は、4 つの要素があることがわかっているネストされたテーブルの例です。

```
DECLARE
    TYPE dname_tbl_typ IS TABLE OF VARCHAR2(14);
    dname_tbl          dname_tbl_typ;
    CURSOR dept_cur IS SELECT dname FROM dept ORDER BY dname;
    i                  INTEGER := 0;
BEGIN
    dname_tbl := dname_tbl_typ(NULL, NULL, NULL, NULL);
    FOR r_dept IN dept_cur LOOP
        i := i + 1;
        dname_tbl(i) := r_dept.dname;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('DNAME');
    DBMS_OUTPUT.PUT_LINE('-----');
    FOR j IN 1..i LOOP
        DBMS_OUTPUT.PUT_LINE(dname_tbl(j));
    END LOOP;
END;
```

上記の例では、次の出力が生成されます。

```
DNAME
-----
ACCOUNTING
OPERATIONS
RESEARCH
SALES
```

次の例では、最初の 10 人の従業員名を emp テーブルから読み取り、ネストしたテーブルに保存してから、テーブルの結果を表示します。SPL コードは、返される従業員の数事前にわからないことを前提として書かれています。

```
DECLARE
    TYPE emp_rec_typ IS RECORD (
        empno      NUMBER(4),
        ename       VARCHAR2(10)
    );
    TYPE emp_tbl_typ IS TABLE OF emp_rec_typ;
```

(次のページに続く)

(前のページからの続き)

```

emp_tbl          emp_tbl_typ;
CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
i                INTEGER := 0;
BEGIN
  emp_tbl := emp_tbl_typ();
  DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
  DBMS_OUTPUT.PUT_LINE('-----      -----');
  FOR r_emp IN emp_cur LOOP
    i := i + 1;
    emp_tbl.EXTEND;
    emp_tbl(i) := r_emp;
  END LOOP;
  FOR j IN 1..10 LOOP
    DBMS_OUTPUT.PUT_LINE(emp_tbl(j).empno || '      ' ||
                          emp_tbl(j).ename);
  END LOOP;
END;

```

匿名ブロックの実行可能セクションの最初のステートメントとしてコンストラクター `emp_tbl_typ()` を使用した空のテーブルの作成に注意してください。次に、`EXTEND` コレクションメソッドを使用して、結果セットから返された各従業員のテーブルに要素を追加します。`EXTEND` の詳細については、[Extend](#) を参照してください。

出力は次のとおりです。

EMPNO	ENAME
-----	-----
7369	SMITH
7499	ALLEN
7521	WARD
7566	JONES
7654	MARTIN
7698	BLAKE
7782	CLARK
7788	SCOTT
7839	KING
7844	TURNER

次の例は、オブジェクト型のネストされたテーブルの使用方を示しています。最初に、部門名と場所の属性を持つオブジェクトタイプが作成されます。

```

CREATE TYPE dept_obj_typ AS OBJECT (
  dname      VARCHAR2(14),
  loc        VARCHAR2(13)
);

```

次の匿名ブロックは、要素が `dept_obj_typ` オブジェクトタイプで構成されるネストされたテーブルタイプを定義します。ネストされたテーブル変数は宣言され、初期化され、`dept` テーブルから読み込まれます。最後に、

ネストされたテーブルの要素が表示されます。

```

DECLARE
    TYPE dept_tbl_typ IS TABLE OF dept_obj_typ;
    dept_tbl          dept_tbl_typ;
    CURSOR dept_cur IS SELECT dname, loc FROM dept ORDER BY dname;
    i                  INTEGER := 0;
BEGIN
    dept_tbl := dept_tbl_typ(
        dept_obj_typ(NULL, NULL),
        dept_obj_typ(NULL, NULL),
        dept_obj_typ(NULL, NULL),
        dept_obj_typ(NULL, NULL)
    );
    FOR r_dept IN dept_cur LOOP
        i := i + 1;
        dept_tbl(i).dname := r_dept.dname;
        dept_tbl(i).loc   := r_dept.loc;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('DNAME          LOC');
    DBMS_OUTPUT.PUT_LINE('-----          -----');
    FOR j IN 1..i LOOP
        DBMS_OUTPUT.PUT_LINE(RPAD(dept_tbl(j).dname, 14) || ' ' ||
                               dept_tbl(j).loc);
    END LOOP;
END;
```

ネストされたテーブルのコンストラクター dept_tbl_typ を構成するパラメーターは、オブジェクトタイプのコンストラクター dept_obj_typ. の呼び出しです。匿名ブロックからの出力は次のとおりです。

DNAME	LOC
-----	-----
ACCOUNTING	NEW YORK
OPERATIONS	BOSTON
RESEARCH	DALLAS
SALES	CHICAGO

10.3 Varrays

`varray*`または`*variable-sizearray` は、正の整数を値に関連付けるコレクションのタイプです。多くの点で、ネストされたテーブルに似ています。

VARRAY には次の特性があります。

- `varraytype*`は、最大サイズ制限とともに定義する必要があります。VARRAY 型が定義された後、その VARRAY 型の`*varray` 変数を宣言できます。データ操作は、VARRAY 変数、または単に「`varray`」を使用しで行われます。VARRAY の要素の数は、VARRAY 型定義で設定された最大サイズ制限を超えることはできません。
- VARRAY 変数が宣言されると、VARRAY は最初は存在しません (NULL コレクションです)。NULLVARRAY は、コンストラクタで初期化する必要があります。また、代入文を使用して VARRAY を初期化することもできます。代入文の右側は、同じ型の初期化された VARRAY です。
- キーは正の整数です。
- コンストラクターは、VARRAY 内の要素の数を設定します。これは、最大サイズ制限を超えてはなりません。EXTEND メソッドは、最大サイズ制限まで追加の要素を `varray` に追加できます。コレクションメソッドの詳細については、[コレクションメソッド](#) を参照してください。
- ネストした表とは異なり、VARRAY はスパースにすることはできません-キーへの値の割り当てにギャップはありません。
- 初期化または拡張サイズを超えて、最大サイズ制限内で `varray` 要素を参照しようとする、SUBSCRIPT_BEYOND_COUNT 例外が発生します。
- 最大サイズ制限を超えて VARRAY 要素を参照しようとしたり、最大サイズ制限を超えて VARRAY を拡張しようとする、SUBSCRIPT_OUTSIDE_LIMIT 例外が発生します。

TYPE IS VARRAY ステートメントは、SPL プログラムの宣言セクション内で `varray` タイプを定義するために使用されます。

```
TYPE <varraytype> IS { VARRAY | VARYING ARRAY }(<maxsize>)
  OF { <datatype> | <objtype> };
```

`varraytype` は、VARRAY タイプに割り当てられた識別子です。 `datatype` は、VARCHAR2 や NUMBER などのスカラーデータ型です。 `maxsize` は、そのタイプの VARRAY で許可される要素の最大数です。 `objtype` は以前に定義されたオブジェクトタイプです。

CREATE TYPE コマンドを使用して、データベース内のすべての SPL プログラムで使用可能な VARRAY タイプを定義できます。VARRAY を使用するには、その VARRAY 型の`*変数*`を宣言する必要があります。以下は、VARRAY 変数を宣言するための構文です。

```
<varray varraytype>
```

varray は VARRAY に割り当てられた識別子です。varraytype は、以前に定義された VARRAY タイプの識別子です。

VARRAY は、VARRAY 型のコンストラクタを使用して初期化されます。

```
<varraytype> ([ { <expr1> | NULL } [, { <expr2> | NULL } ]
  [, ...] ])
```

varraytype は VARRAY 型のコンストラクターの識別子であり、VARRAY 型と同じ名前を持ちます。expr1, expr2, … は、VARRAY の要素型と型互換性のある式です。NULL が指定されている場合、対応する要素は null に設定されます。パラメータリストが空の場合、空の VARRAY が返されます。これは、VARRAY に要素がないことを意味します。VARRAY がオブジェクト型から定義されている場合、exprn はそのオブジェクト型のオブジェクトを返す必要があります。オブジェクトは、関数の戻り値またはオブジェクト型のコンストラクターの戻り値になります。オブジェクトは、同じ VARRAY タイプの別の VARRAY の要素にすることもできます。

EXISTS 以外の収集メソッドが初期化されていない VARRAY に適用されると、COLLECTION_IS_NULL 例外がスローされます。コレクションメソッドの詳細については、[コレクションメソッド](#)を参照してください。

以下は、VARRAY のコンストラクターの例です。

```
DECLARE
    TYPE varray_ttyp IS VARRAY(2) OF CHAR(1);
    v_varray          varray_ttyp := varray_ttyp('A','B');
```

VARRAY の要素は、次の構文を使用して参照されます。

```
<varray>(<n>)[<.element> ]
```

varray は、事前に宣言された VARRAY の識別子です。n は正の整数です。varray の VARRAY 型がオブジェクト型から定義されている場合、[.element] は VARRAY 型の定義元のオブジェクト型内の属性を参照する必要があります。または、[.element]. を省略してオブジェクト全体を参照できます

以下は、4 つの要素があることがわかっている VARRAY の例です。

```
DECLARE
    TYPE dname_varray_ttyp IS VARRAY(4) OF VARCHAR2(14);
    dname_varray          dname_varray_ttyp;
    CURSOR dept_cur IS SELECT dname FROM dept ORDER BY dname;
    i                      INTEGER := 0;
BEGIN
    dname_varray := dname_varray_ttyp(NULL, NULL, NULL, NULL);
    FOR r_dept IN dept_cur LOOP
        i := i + 1;
        dname_varray(i) := r_dept.dname;
```

(次のページに続く)

(前のページからの続き)

```
END LOOP;
DBMS_OUTPUT.PUT_LINE('DNAME');
DBMS_OUTPUT.PUT_LINE('-----');
FOR j IN 1..i LOOP
    DBMS_OUTPUT.PUT_LINE(dname_varray(j));
END LOOP;
END;
```

上記の例では、次の出力が生成されます。

```
DNAME
-----
ACCOUNTING
OPERATIONS
RESEARCH
SALES
```


第 11 章

Collection Methods

コレクションメソッドは、コレクション内のデータの処理に役立つコレクションに関する有用な情報を提供する関数とプロシージャです。次のセクションでは、AdvancedServer がサポートする収集方法について説明します。

11.1 COUNT

COUNT は、コレクション内の要素の数を返すメソッドです。COUNT を使用するための構文は次のとおりです。

```
<collection>.COUNT
```

collection はコレクションの名前です。

VARRAY の場合、COUNT は常に LAST に等しくなります。

次の例は、連想配列にまばらに入力できることを示しています（つまり、割り当てられた要素のシーケンスに「ギャップ」があります）。COUNT には、値が割り当てられている要素のみが含まれます。

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr          sparse_arr_typ;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10)  := -10;
    sparse_arr(0)    := 0;
    sparse_arr(10)   := 10;
    sparse_arr(100)  := 100;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
END;
```

次の出力は、COUNT に含まれる 5 つの要素が含まれていることを示しています。

```
COUNT: 5
```

11.2 DELETE

DELETE メソッドは、コレクションからエントリを削除します。DELETE メソッドは、3 つの異なる方法で呼び出すことができます。

DELETE メソッドの最初の形式を使用して、コレクションからすべてのエントリを削除します。

```
<collection>.DELETE
```

DELETE メソッドの 2 番目の形式を使用して、指定したエントリをコレクションから削除します。

```
<collection>.DELETE(<subscript>)
```

DELETE メソッドの 3 番目の形式を使用して、first_subscript と last_subscript で指定された範囲内のエントリ（first_subscript と last_subscript のエントリを含む）をコレクションから削除します。

```
<collection>.DELETE(<first_subscript>, <last_subscript>)
```

first_subscript および last_subscript が存在しない要素を参照する場合、指定された添え字の間の範囲にある要素は削除されます。first_subscript が last_subscript より大きい場合、または引数の 1 つに NULL の値を指定した場合、DELETE は効果がありません。

エントリを削除しても、添え字はコレクションに残ることに注意してください。代替エントリで添え字を再利用できます。DELETE メソッドの呼び出しに存在しない添え字を指定した場合、DELETE は例外を発生させません。

次の例は、DELETE メソッドを使用して、コレクションから添え字 0 を持つ要素を削除する方法を示しています。

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr          sparse_arr_typ;
    v_results           VARCHAR2(50);
    v_sub               NUMBER;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10)  := -10;
    sparse_arr(0)    := 0;
    sparse_arr(10)   := 10;
    sparse_arr(100)  := 100;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.DELETE(0);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    v_sub := sparse_arr.FIRST;
```

(次のページに続く)

(前のページからの続き)

```

WHILE v_sub IS NOT NULL LOOP
    IF sparse_arr(v_sub) IS NULL THEN
        v_results := v_results || 'NULL ';
    ELSE
        v_results := v_results || sparse_arr(v_sub) || ' ';
    END IF;
    v_sub := sparse_arr.NEXT(v_sub);
END LOOP;
DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
COUNT: 4
Results: -100 -10 10 100

```

COUNT は、DELETE メソッドの前に、コレクションに 5 要素があったことを示します。DELETE メソッドが呼び出された後、コレクションには 4 要素が含まれます。

11.3 EXISTS

EXISTS メソッドは、コレクション内に添え字が存在することを確認します。下付き文字が存在する場合、EXISTS は TRUE を返します。下付き文字が存在しない場合、EXISTS は FALSE を返します。このメソッドは単一の引数を取ります。テスト対象の subscript。構文は次のとおりです。

```
<collection>.EXISTS(<subscript>)
```

collection はコレクションの名前です。

subscript は、テストする値です。NULL, EXISTS の値を指定すると、false が返されます。

次の例は、添え字番号 10 が連想配列内に存在することを確認します。

```

DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr          sparse_arr_typ;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10)  := -10;
    sparse_arr(0)    := 0;
    sparse_arr(10)   := 10;
    sparse_arr(100)  := 100;
    DBMS_OUTPUT.PUT_LINE('The index exists: ' ||
        CASE WHEN sparse_arr.exists(10) = TRUE THEN 'true' ELSE 'false' END);
END;

```

(次のページに続く)

(前のページからの続き)

```
The index exists: true
```

一部のコレクションメソッドは、指定されたコレクション内に存在しない添え字で呼び出した場合に例外を発生させます。エラーを発生させるのではなく、EXISTS メソッドは FALSE の値を返します。

11.4 EXTEND

EXTEND メソッドは、コレクションのサイズを増やします。EXTEND メソッドには 3 つのバリエーションがあります。最初のバリエーションは、単一の NULL 要素をコレクションに追加します。最初のバリエーションの構文は次のとおりです。

```
<collection>.EXTEND
```

collection はコレクションの名前です。

次の例は、EXTEND メソッドを使用して単一のヌル要素をコレクションに追加する方法を示しています。

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr          sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results           VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
COUNT: 6
Results: -100 -10 0 10 100 NULL
```

COUNT は、EXTEND メソッドの前に、コレクションに 5 要素があったことを示します。EXTEND メソッドが呼び出された後、コレクションには 6 要素が含まれます。

EXTEND メソッドの 2 番目のバリエーションは、指定された数の要素をコレクションの末尾に追加します。

```
<collection>.EXTEND(<count>)
```

collection はコレクションの名前です。

count は、コレクションの最後に追加されるヌル要素の数です。

次の例は、EXTEND メソッドを使用して複数の null 要素をコレクションに追加する方法を示しています。

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr          sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results           VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND(3);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
```

COUNT: 5
COUNT: 8
Results: -100 -10 0 10 100 NULL NULL NULL

COUNT は、EXTEND メソッドの前に、コレクションに 5 要素があったことを示します。EXTEND メソッドが呼び出された後、コレクションには 8 要素が含まれます。

EXTEND メソッドの 3 番目のバリエーションは、特定の要素の指定された数のコピーをコレクションの末尾に追加します。

```
<collection>.EXTEND(<count>, <index_number>)
```

collection はコレクションの名前です。

count は、コレクションの最後に追加される要素の数です。

index_number は、コレクションにコピーされる要素の添え字です。

次の例は、EXTEND メソッドを使用して、2 番目の要素の複数のコピーをコレクションに追加する方法を示しています。

```

DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr          sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results           VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND(3, 2);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
COUNT: 8
Results: -100 -10 0 10 100 -10 -10 -10

```

COUNT は、EXTEND メソッドの前に、コレクションに 5 要素があったことを示します。EXTEND メソッドが呼び出された後、コレクションには 8 要素が含まれます。

注釈: EXTEND メソッドは、null または空のコレクションでは使用できません。

11.5 FIRST

FIRST は、コレクションの最初の要素の添え字を返すメソッドです。FIRST を使用するための構文は次のとおりです。

```
<collection>.FIRST
```

collection はコレクションの名前です。

次の例は、連想配列の最初の要素を表示します。

```

DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr          sparse_arr_typ;
BEGIN
    sparse_arr(-100) := -100;

```

(次のページに続く)

(前のページからの続き)

```

sparse_arr(-10)    := -10;
sparse_arr(0)      := 0;
sparse_arr(10)     := 10;
sparse_arr(100)    := 100;
DBMS_OUTPUT.PUT_LINE('FIRST element: ' || sparse_arr(sparse_arr.FIRST));
END;

FIRST element: -100

```

11.6 LAST

LAST は、コレクションの最後の要素の添え字を返すメソッドです。LAST を使用するための構文は次のとおりです。

```
<collection>.LAST
```

collection はコレクションの名前です。

次の例は、連想配列の最後の要素を表示します。

```

DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr          sparse_arr_typ;
BEGIN
    sparse_arr(-100)    := -100;
    sparse_arr(-10)     := -10;
    sparse_arr(0)       := 0;
    sparse_arr(10)      := 10;
    sparse_arr(100)     := 100;
    DBMS_OUTPUT.PUT_LINE('LAST element: ' || sparse_arr(sparse_arr.LAST));
END;

LAST element: 100

```

11.7 LIMIT

LIMIT は、コレクションで許可される要素の最大数を返すメソッドです。LIMIT は VARRAY にのみ適用されます。LIMIT を使用するための構文は次のとおりです。

```
<collection>.LIMIT
```

collection はコレクションの名前です。

初期化された **VARRAY** の場合、**LIMIT** は **VARRAY** 型定義で決定された最大サイズ制限を返します。**VARRAY** が初期化されていない場合（つまり、**NULLVARRAY** である場合）、例外がスローされます。

連想配列または初期化されたネストされたテーブルの場合、**LIMIT** は **NULL** を返します。ネストしたテーブルが初期化されていない場合（つまり、ネストしたテーブルが **null** の場合）、例外がスローされます。

11.8 NEXT

NEXT は、指定された添え字に続く添え字を返すメソッドです。このメソッドは単一の引数を取ります。テストする subscript。

```
<collection>.NEXT(<subscript>)
```

collection はコレクションの名前です。

指定した添え字がコレクション内の最初の添え字より小さい場合、関数は最初の添え字を返します。添え字に後継がない場合、**NEXT** は **NULL** を返します。**NULL** 添え字を指定すると、**PRIOR** は値を返しません。

次の例では、**NEXT** を使用して、連想配列 **sparse_arr** の添え字 10 に続く添え字を返します。

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr          sparse_arr_typ;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10)  := -10;
    sparse_arr(0)    := 0;
    sparse_arr(10)   := 10;
    sparse_arr(100)  := 100;
    DBMS_OUTPUT.PUT_LINE('NEXT element: ' || sparse_arr.next(10));
END;

NEXT element: 100
```

11.9 PRIOR

PRIOR メソッドは、コレクション内の指定された添え字の前の添え字を返します。このメソッドは単一の引数を取ります。テストする subscript。構文は次のとおりです。

```
<collection>.PRIOR(<subscript>)
```

collection はコレクションの名前です。

指定された添え字に先行タスクがない場合、**PRIOR** は **NULL** を返します。指定された添え字がコレクション内の

最後の添え字より大きい場合、メソッドは最後の添え字を返します。NULL 添え字を指定すると、PRIOR は値を返しません。

次の例では、連想配列 `sparse_arr`: の添え字 100 に先行する添え字を返します

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr          sparse_arr_typ;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10)  := -10;
    sparse_arr(0)    := 0;
    sparse_arr(10)   := 10;
    sparse_arr(100)  := 100;
    DBMS_OUTPUT.PUT_LINE('PRIOR element: ' || sparse_arr.prior(100));
END;

PRIOR element: 10
```

11.10 TRIM

TRIM メソッドは、コレクションの最後から要素を削除します。TRIM メソッドの構文は次のとおりです。

```
<collection>.TRIM[(<count>)]
```

`collection` はコレクションの名前です。

`count` は、コレクションの最後から削除された要素の数です。 `count` が 0 より小さいか、コレクション内の要素の数より大きい場合、AdvancedServer はエラーを返します。

次の例は、TRIM メソッドを使用してコレクションの末尾から要素を削除する方法を示しています。

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr          sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.TRIM;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
END;

COUNT: 5
COUNT: 4
```

COUNT は、TRIM メソッドの前に、コレクションに 5 要素があったことを示します。TRIM メソッドが呼び出された後、コレクションには 4 要素が含まれます。

TRIM メソッドを使用して、コレクションの末尾から削除する要素の数を指定することもできます。

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr          sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results           VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.TRIM(2);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
```

COUNT: 5
COUNT: 3
Results: -100 -10 0

COUNT は、TRIM メソッドの前に、コレクションに 5 要素があったことを示します。TRIM メソッドが呼び出された後、コレクションには 3 要素が含まれます。

第 12 章

Working with Collections

コレクション演算子を使用すると、コレクションのコンテンツを変換、クエリ、および操作できます。

12.1 TABLE()

TABLE () 関数を使用して、配列のメンバーを行のセットに変換します。署名は次のとおりです。

```
TABLE (<collection_value>)
```

どこで:

collection_value

collection_value は、コレクションタイプの値に評価される式です。

TABLE () 関数は、コレクションのネストされたコンテンツをテーブル形式に展開します。TABLE () 関数は、正規のテーブル式を使用する任意の場所で使用できます。

TABLE () 関数は SETOF ANYELEMENT (任意のタイプの値のセット) を返します。たとえば、この関数に渡される引数が dates, TABLE () の配列の場合、SETOF dates が返されます。この関数に渡される引数が paths, TABLE () の配列の場合、SETOF paths を返します。

TABLE () 関数を使用して、コレクションの内容を表形式に展開できます。

```
postgres=# SELECT * FROM TABLE(monthly_balance(445.00, 980.20, 552.00));

monthly_balance
-----
 445.00
 980.20
 552.00
(3 rows)
```

12.2 Using the MULTiset UNION Operator

MULTiset UNION 演算子は、2つのコレクションを組み合わせて3番目のコレクションを形成します。署名は次のとおりです。

```
<coll_1> MULTiset UNION [ ALL | DISTINCT | UNIQUE ] <coll_2>
```

coll_1 と coll_2 は、結合するコレクションの名前を指定します。

ALL キーワードを含めて、元のコレクションに存在するたびに1回ずつ、重複する要素（coll_1 と coll_2 の両方に存在する要素）を結果に表示するように指定します。これは MULTiset UNION のデフォルトの動作です。

DISTINCT または UNIQUE キーワードを含めて、重複する要素を結果に1回だけ含めるように指定します。DISTINCT と UNIQUE のキーワードは同義語です。

次の例は、MULTiset UNION 演算子を使用して2つのコレクション（collection_1 と collection_2）を3番目のコレクション（collection_3）に結合する方法を示しています。

```
DECLARE
    TYPE int_arr_typ IS TABLE OF NUMBER(2);
    collection_1    int_arr_typ;
    collection_2    int_arr_typ;
    collection_3    int_arr_typ;
    v_results       VARCHAR2(50);
BEGIN
    collection_1 := int_arr_typ(10,20,30);
    collection_2 := int_arr_typ(30,40);
    collection_3 := collection_1 MULTiset UNION ALL collection_2;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || collection_3.COUNT);
    FOR i IN collection_3.FIRST .. collection_3.LAST LOOP
        IF collection_3(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || collection_3(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
```

COUNT: 5
Results: 10 20 30 30 40

結果のコレクションには、collection_1 と collection_2 の要素ごとに1つのエントリが含まれます。DISTINCT キーワードを使用すると、結果は次のようになります。

```
DECLARE
    TYPE int_arr_typ IS TABLE OF NUMBER(2);
```

(次のページに続く)

(前のページからの続き)

```

collection_1    int_arr_typ;
collection_2    int_arr_typ;
collection_3    int_arr_typ;
v_results       VARCHAR2(50);
BEGIN
    collection_1 := int_arr_typ(10,20,30);
    collection_2 := int_arr_typ(30,40);
    collection_3 := collection_1 MULTISSET UNION DISTINCT collection_2;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || collection_3.COUNT);
    FOR i IN collection_3.FIRST .. collection_3.LAST LOOP
        IF collection_3(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || collection_3(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 4
Results: 10 20 30 40

```

結果のコレクションには、異なる値を持つメンバーのみが含まれます。次の例では、MULTISSET UNION DISTINCT 演算子は同じコレクション内に保存されている重複エントリも削除することに注意してください。

```

DECLARE
    TYPE int_arr_typ IS TABLE OF NUMBER(2);
    collection_1    int_arr_typ;
    collection_2    int_arr_typ;
    collection_3    int_arr_typ;
    v_results       VARCHAR2(50);
BEGIN
    collection_1 := int_arr_typ(10,20,30,30);
    collection_2 := int_arr_typ(40,50);
    collection_3 := collection_1 MULTISSET UNION DISTINCT collection_2;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || collection_3.COUNT);
    FOR i IN collection_3.FIRST .. collection_3.LAST LOOP
        IF collection_3(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || collection_3(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5

```

(次のページに続く)

(前のページからの続き)

Results: 10 20 30 40 50

12.3 Using the FORALL Statement

コレクションを使用して DML コマンドをより効率的に処理するには、新しい「値。このような方法で処理される DML コマンドは、FORALL ステートメントで指定されます。さらに、1 つ以上のコレクションが DML コマンドで提供され、コマンドが実行されるたびに異なる値が置換されます。

```
FORALL <index> IN <lower_bound> .. <upper_bound>
  { <insert_stmt> | <update_stmt> | <delete_stmt> };
```

index は、lower_bound として指定された整数値から upper_bound. までの反復する insert_stmt, update_stmt または delete_stmt DML コマンドで指定されたコレクション内の位置です。

FORALL ステートメントの反復中に例外が発生すると、FORALL ステートメントの実行開始以降に発生したすべての更新が自動的にロールバックされます。この動作は、Oracle データベースと互換性がありません。Oracle では、COMMIT または ROLLBACK コマンドを明示的に使用して、例外の前に発生した更新をコミットまたはロールバックするかどうかを制御できます。

FORALL ステートメントはループを作成します-ループの各反復は index 変数を増分します（通常、ループ内の index を使用してコレクションのメンバーを選択します）。反復回数は lower_bound .. upper_bound 節によって制御されます。ループは lower_bound から upper_bound までの整数ごとに 1 回実行され、インデックスは反復ごとに 1 つずつ増加します。例:

```
FORALL i IN 2 .. 5
```

4 回実行するループを作成します。最初の反復では、index (i) が値 2 に設定されます。2 回目の反復では、インデックスは値 3 に設定されます。ループは値 5 に対して実行され、その後終了します。

次の例では、emp テーブルの空のコピーである (emp_copy) テーブルを作成します。この例では、配列 (emp_tbl) を宣言しています。これは、配列の各要素が複合型であり、表 emp の作成に使用される列定義で構成されています。この例では、emp_tbl タイプのインデックスも作成されます。

t_emp は emp_tbl 型の連想配列です。SELECT ステートメントは、BULK COLLECT INTO コマンドを使用して t_emp 配列に入力します。t_emp 配列が読み込まれると、FORALL ステートメントは t_emp 配列インデックスの値 (i) を反復処理し、各レコードの行を emp_copy に挿入します。

```
CREATE TABLE emp_copy(LIKE emp);

DECLARE

    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
```

(次のページに続く)

(前のページからの続き)

```

t_emp emp_tbl;

BEGIN
    SELECT * FROM emp BULK COLLECT INTO t_emp;

    FORALL i IN t_emp.FIRST .. t_emp.LAST
        INSERT INTO emp_copy VALUES t_emp(i);

END;
```

次の例では、FORALL ステートメントを使用して 3 人の従業員の給与を更新します。

```

DECLARE
    TYPE empno_tbl IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
    TYPE sal_tbl IS TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;
    t_empno EMPNO_TBL;
    t_sal SAL_TBL;
BEGIN
    t_empno(1) := 9001;
    t_sal(1) := 3350.00;
    t_empno(2) := 9002;
    t_sal(2) := 2000.00;
    t_empno(3) := 9003;
    t_sal(3) := 4100.00;
    FORALL i IN t_empno.FIRST..t_empno.LAST
        UPDATE emp SET sal = t_sal(i) WHERE empno = t_empno(i);
END;
```

```

SELECT * FROM emp WHERE empno > 9000;
```

empno	ename	job	mgr	hiredate	sal	comm	deptno
9001	JONES	ANALYST			3350.00		40
9002	LARSEN	CLERK			2000.00		40
9003	WILSON	MANAGER			4100.00		40

(3 rows)

次の例では、FORALL ステートメントで 3 人の従業員を削除します。

```

DECLARE
    TYPE empno_tbl IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
    t_empno EMPNO_TBL;
BEGIN
    t_empno(1) := 9001;
    t_empno(2) := 9002;
    t_empno(3) := 9003;
    FORALL i IN t_empno.FIRST..t_empno.LAST
```

(次のページに続く)

(前のページからの続き)

```

DELETE FROM emp WHERE empno = t_empno(i);
END;

SELECT * FROM emp WHERE empno > 9000;

 empno |  ename  |  job  | mgr | hiredate | sal | comm | deptno
-----+-----+-----+-----+-----+-----+-----+-----
(0 rows)

```

12.4 Using the BULK COLLECT Clause

多数の行で構成される結果セットを返す SQL コマンドは、結果セット全体を転送するためにデータベースサーバーとクライアント間で絶え間ないコンテキストスイッチングが発生するため、可能な限り効率的に動作しない場合があります。この非効率性は、コレクションを使用してメモリに結果セット全体を収集し、クライアントがアクセスできるようにすることで軽減できます。BULK COLLECT 句を使用して、結果セットのコレクションへの集約を指定します。

BULK COLLECT 節は、SELECT INTO、FETCH INTO および EXECUTE IMMEDIATE コマンド、および DELETE、INSERT および UPDATE コマンドの RETURNING INTO 節で使用できます。これらのそれぞれについて、次のセクションで説明します。

12.4.1 SELECT BULK COLLECT

BULK COLLECT 句は、次のように SELECT INTO ステートメントで使用できます。(SELECT INTO ステートメントの詳細については、[SELECT INTO](#) を参照してください。)

```

SELECT <select_expressions> BULK COLLECT INTO <collection>
    [, ...] FROM ...;

```

単一のコレクションが指定されている場合、collection は単一のフィールドのコレクション、またはレコードタイプのコレクションです。複数のコレクションを指定する場合、各 collection は単一のフィールドで構成する必要があります。select_expressions は、ターゲットコレクションのすべてのフィールドの数、順序、および型互換性が一致する必要があります。

次の例は、ターゲットコレクションが単一のフィールドで構成される連想配列である BULK COLLECT 句の使用方法を示しています。

```

DECLARE
    TYPE empno_ttbl    IS TABLE OF emp.empno%TYPE    INDEX BY BINARY_INTEGER;
    TYPE ename_ttbl    IS TABLE OF emp.ename%TYPE    INDEX BY BINARY_INTEGER;
    TYPE job_ttbl      IS TABLE OF emp.job%TYPE      INDEX BY BINARY_INTEGER;

```

(次のページに続く)

(前のページからの続き)

```

TYPE hiredate_tbl IS TABLE OF emp.hiredate%TYPE INDEX BY BINARY_INTEGER;
TYPE sal_tbl      IS TABLE OF emp.sal%TYPE      INDEX BY BINARY_INTEGER;
TYPE comm_tbl     IS TABLE OF emp.comm%TYPE     INDEX BY BINARY_INTEGER;
TYPE deptno_tbl   IS TABLE OF emp.deptno%TYPE   INDEX BY BINARY_INTEGER;
t_empno           EMPNO_TBL;
t_ename           ENAME_TBL;
t_job             JOB_TBL;
t_hiredate        HIREDATE_TBL;
t_sal             SAL_TBL;
t_comm            COMM_TBL;
t_deptno          DEPTNO_TBL;
BEGIN
  SELECT empno, ename, job, hiredate, sal, comm, deptno BULK COLLECT
    INTO t_empno, t_ename, t_job, t_hiredate, t_sal, t_comm, t_deptno
    FROM emp;
  DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      JOB            HIREDATE      ' ||
    'SAL          ' || 'COMM      DEPTNO');
  DBMS_OUTPUT.PUT_LINE('-----  -----  -----  -----  ' ||
    '-----  ' || '-----  -----');
  FOR i IN 1..t_empno.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(t_empno(i) || ' ' ||
      RPAD(t_ename(i),8) || ' ' ||
      RPAD(t_job(i),10) || ' ' ||
      TO_CHAR(t_hiredate(i),'DD-MON-YY') || ' ' ||
      TO_CHAR(t_sal(i),'99,999.99') || ' ' ||
      TO_CHAR(NVL(t_comm(i),0),'99,999.99') || ' ' ||
      t_deptno(i));
  END LOOP;
END;
```

EMPNO	ENAME	JOB	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	17-DEC-80	800.00	.00	20
7499	ALLEN	SALESMAN	20-FEB-81	1,600.00	300.00	30
7521	WARD	SALESMAN	22-FEB-81	1,250.00	500.00	30
7566	JONES	MANAGER	02-APR-81	2,975.00	.00	20
7654	MARTIN	SALESMAN	28-SEP-81	1,250.00	1,400.00	30
7698	BLAKE	MANAGER	01-MAY-81	2,850.00	.00	30
7782	CLARK	MANAGER	09-JUN-81	2,450.00	.00	10
7788	SCOTT	ANALYST	19-APR-87	3,000.00	.00	20
7839	KING	PRESIDENT	17-NOV-81	5,000.00	.00	10
7844	TURNER	SALESMAN	08-SEP-81	1,500.00	.00	30
7876	ADAMS	CLERK	23-MAY-87	1,100.00	.00	20
7900	JAMES	CLERK	03-DEC-81	950.00	.00	30
7902	FORD	ANALYST	03-DEC-81	3,000.00	.00	20
7934	MILLER	CLERK	23-JAN-82	1,300.00	.00	10

次の例は同じ結果を生成しますが、%ROWTYPE 属性で定義されたレコードタイプで連想配列を使用します。

```

DECLARE
    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    t_emp          EMP_TBL;
BEGIN
    SELECT * BULK COLLECT INTO t_emp FROM emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      JOB              HIREDATE      ' ||
        'SAL          ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -----  -----  -----  ' ||
        '-----  ' || '-----  -----');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || ' ' ||
            RPAD(t_emp(i).ename,8) || ' ' ||
            RPAD(t_emp(i).job,10) || ' ' ||
            TO_CHAR(t_emp(i).hiredate,'DD-MON-YY') || ' ' ||
            TO_CHAR(t_emp(i).sal,'99,999.99') || ' ' ||
            TO_CHAR(NVL(t_emp(i).comm,0),'99,999.99') || ' ' ||
            t_emp(i).deptno);
    END LOOP;
END;

```

EMPNO	ENAME	JOB	HIREDATE	SAL	COMM	DEPTNO
-----	-----	-----	-----	-----	-----	-----
7369	SMITH	CLERK	17-DEC-80	800.00	.00	20
7499	ALLEN	SALESMAN	20-FEB-81	1,600.00	300.00	30
7521	WARD	SALESMAN	22-FEB-81	1,250.00	500.00	30
7566	JONES	MANAGER	02-APR-81	2,975.00	.00	20
7654	MARTIN	SALESMAN	28-SEP-81	1,250.00	1,400.00	30
7698	BLAKE	MANAGER	01-MAY-81	2,850.00	.00	30
7782	CLARK	MANAGER	09-JUN-81	2,450.00	.00	10
7788	SCOTT	ANALYST	19-APR-87	3,000.00	.00	20
7839	KING	PRESIDENT	17-NOV-81	5,000.00	.00	10
7844	TURNER	SALESMAN	08-SEP-81	1,500.00	.00	30
7876	ADAMS	CLERK	23-MAY-87	1,100.00	.00	20
7900	JAMES	CLERK	03-DEC-81	950.00	.00	30
7902	FORD	ANALYST	03-DEC-81	3,000.00	.00	20
7934	MILLER	CLERK	23-JAN-82	1,300.00	.00	10

12.4.2 FETCH BULK COLLECT

BULK COLLECT 節は FETCH ステートメントで使用できます。（FETCH ステートメントの詳細については、「[カーソルからの行のフェッチ](#)」を参照してください。）結果セットから一度に 1 行を返す代わりに、FETCH BULK COLLECT は LIMIT 句によって制限されていない限り、指定されたコレクションに結果セット。

```
FETCH <name> BULK COLLECT INTO <collection> [, ...] [ LIMIT <n> ];
```

単一のコレクションが指定されている場合、collection は単一のフィールドのコレクション、またはレコード

タイプのコレクションです。複数のコレクションを指定する場合、各 collection は単一のフィールドで構成する必要があります。name で識別されるカーソルの SELECT リストの式は、ターゲットコレクションのすべてのフィールドの数、順序、および型互換性が一致する必要があります。LIMIT n が指定されている場合、各 FETCH でコレクションに返される行の数は n を超えません。

次の例では、FETCH BULK COLLECT ステートメントを使用して、連想配列に行を取得します。

```
DECLARE
    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    t_emp          EMP_TBL;
    CURSOR emp_cur IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur;
    FETCH emp_cur BULK COLLECT INTO t_emp;
    CLOSE emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      JOB            HIREDATE      ' ||
                          'SAL          ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -----  -----  -----  ' ||
                          '-----  ' || '-----  -----');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || ' ' ||
                              RPAD(t_emp(i).ename,8) || ' ' ||
                              RPAD(t_emp(i).job,10) || ' ' ||
                              TO_CHAR(t_emp(i).hiredate,'DD-MON-YY') || ' ' ||
                              TO_CHAR(t_emp(i).sal,'99,999.99') || ' ' ||
                              TO_CHAR(NVL(t_emp(i).comm,0),'99,999.99') || ' ' ||
                              t_emp(i).deptno);
    END LOOP;
END;
```

EMPNO	ENAME	JOB	HIREDATE	SAL	COMM	DEPTNO
-----	-----	-----	-----	-----	-----	-----
7369	SMITH	CLERK	17-DEC-80	800.00	.00	20
7499	ALLEN	SALESMAN	20-FEB-81	1,600.00	300.00	30
7521	WARD	SALESMAN	22-FEB-81	1,250.00	500.00	30
7566	JONES	MANAGER	02-APR-81	2,975.00	.00	20
7654	MARTIN	SALESMAN	28-SEP-81	1,250.00	1,400.00	30
7698	BLAKE	MANAGER	01-MAY-81	2,850.00	.00	30
7782	CLARK	MANAGER	09-JUN-81	2,450.00	.00	10
7788	SCOTT	ANALYST	19-APR-87	3,000.00	.00	20
7839	KING	PRESIDENT	17-NOV-81	5,000.00	.00	10
7844	TURNER	SALESMAN	08-SEP-81	1,500.00	.00	30
7876	ADAMS	CLERK	23-MAY-87	1,100.00	.00	20
7900	JAMES	CLERK	03-DEC-81	950.00	.00	30
7902	FORD	ANALYST	03-DEC-81	3,000.00	.00	20
7934	MILLER	CLERK	23-JAN-82	1,300.00	.00	10

12.4.3 EXECUTE IMMEDIATE BULK COLLECT

BULK COLLECT 句を EXECUTE IMMEDIATE ステートメントとともに使用して、返される行を受け取るコレクションを指定できます。

```
EXECUTE IMMEDIATE '<sql_expression>;'
    BULK COLLECT INTO <collection> [,...]
    [USING {[<bind_type>] <bind_argument>} [, ...]]];
```

collection はコレクションの名前を指定します。

bind_type は bind_argument のパラメータモードを指定します。

- IN の bind_type は、bind_argument に sql_expression に渡される値が含まれることを指定します。
- OUT の bind_type は、bind_argument が sql_expression から値を受け取ることを指定します。
- IN OUT の bind_type は、bind_argument が sql_expression に渡されることを指定し、sql_expression によって返される値を格納します。

bind_argument は、sql_expression (IN の bind_type で指定) に渡される値、または sql_expression (bind_type で指定の「OUT」、またはその両方 (IN OUT の bind_type で指定)。

単一のコレクションが指定されている場合、collection は単一のフィールドのコレクション、またはレコードタイプのコレクションです。複数のコレクションを指定する場合、各 collection は単一のフィールドで構成する必要があります。

12.4.4 RETURNING BULK COLLECT

BULK COLLECT 句は、DELETE、INSERT または UPDATE コマンドの RETURNING INTO 句に追加できます。(RETURNING INTO 節の詳細については、[:ref:「RETURNING INTO 節の使用<using_the_returning_into_clause>」](#)を参照してください。)

```
{ <insert> | <update> | <delete> }
    RETURNING { * | <expr_1> [, <expr_2> ] ...}
    BULK COLLECT INTO <collection> [, ...];
```

insert、update、delete は「INSERT、UPDATE、および DELETE<削除>」で説明されている「INSERT、UPDATE」、DELETE コマンドです。>、それぞれ。単一のコレクションが指定されている場合、collection は単一のフィールドのコレクション、またはレコードタイプのコレクションです。複数のコレクションを指定する場合、各 collection は単一のフィールドで構成する必要があります。RETURNING キーワードに続く式は、ターゲットコレクションのすべてのフィールドの数、順序、および型互換性が一致する必要があります。* を指定すると、影響を受けるテーブルのすべての列が返されます。(* の使用は AdvancedServer 拡張機能であり、Oracle データベースと互換性がないことに注意してください。)

emp テーブルをコピーして作成された clerkemp テーブルは、以下に示すように、このセクションの残りの例で使用されます。

```
CREATE TABLE clerkemp AS SELECT * FROM emp WHERE job = 'CLERK';

SELECT * FROM clerkemp;
```

empno	ename	job	mgr	hiredate	sal	comm	deptno
7369	SMITH	CLERK	7902	17-DEC-80 00:00:00	800.00		20
7876	ADAMS	CLERK	7788	23-MAY-87 00:00:00	1100.00		20
7900	JAMES	CLERK	7698	03-DEC-81 00:00:00	950.00		30
7934	MILLER	CLERK	7782	23-JAN-82 00:00:00	1300.00		10

(4 rows)

次の例では、全員の給与を 1.5 増やし、従業員の番号、名前、新しい給与を 3 つの連想配列に保存し、最後にこれらの配列の内容を表示します。

```
DECLARE
    TYPE empno_tbl IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
    TYPE ename_tbl IS TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;
    TYPE sal_tbl IS TABLE OF emp.sal%TYPE INDEX BY BINARY_INTEGER;
    t_empno      empno_tbl;
    t_ename      ename_tbl;
    t_sal        sal_tbl;
BEGIN
    UPDATE clerkemp SET sal = sal * 1.5 RETURNING empno, ename, sal
        BULK COLLECT INTO t_empno, t_ename, t_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      SAL          ');
    DBMS_OUTPUT.PUT_LINE('-----  -----  -----  ');
    FOR i IN 1..t_empno.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_empno(i) || ' ' || RPAD(t_ename(i), 8) ||
            ' ' || TO_CHAR(t_sal(i), '99,999.99'));
    END LOOP;
END;
```

EMPNO	ENAME	SAL
7369	SMITH	1,200.00
7876	ADAMS	1,650.00
7900	JAMES	1,425.00
7934	MILLER	1,950.00

次の例では、前の例と同じ機能を実行しますが、レコードタイプで定義された単一のコレクションを使用して、従業員の番号、名前、新しい給与を保存します。

```

DECLARE
    TYPE emp_rec IS RECORD (
        empno      emp.empno%TYPE,
        ename       emp.ename%TYPE,
        sal         emp.sal%TYPE
    );
    TYPE emp_tbl IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
    t_emp         EMP_TBL;
BEGIN
    UPDATE clerkemp SET sal = sal * 1.5 RETURNING empno, ename, sal
        BULK COLLECT INTO t_emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      SAL          ');
    DBMS_OUTPUT.PUT_LINE('-----  -----  -----  ');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || ' ' ||
            RPAD(t_emp(i).ename,8) || ' ' ||
            TO_CHAR(t_emp(i).sal, '99,999.99'));
    END LOOP;
END;

```

EMPNO	ENAME	SAL
-----	-----	-----
7369	SMITH	1,200.00
7876	ADAMS	1,650.00
7900	JAMES	1,425.00
7934	MILLER	1,950.00

次の例では、clerkemp テーブルからすべての行を削除し、削除された行に関する情報を連想配列に返します。その後、連想配列が表示されます。

```

DECLARE
    TYPE emp_rec IS RECORD (
        empno      emp.empno%TYPE,
        ename       emp.ename%TYPE,
        job         emp.job%TYPE,
        hiredate    emp.hiredate%TYPE,
        sal         emp.sal%TYPE,
        comm        emp.comm%TYPE,
        deptno      emp.deptno%TYPE
    );
    TYPE emp_tbl IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
    r_emp         EMP_TBL;
BEGIN
    DELETE FROM clerkemp RETURNING empno, ename, job, hiredate, sal,
        comm, deptno BULK COLLECT INTO r_emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB          HIREDATE    ' ||
        'SAL      ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -----  -----  -----  ' ||

```

(次のページに続く)

(前のページからの続き)

```
'-----' || '-----');
FOR i IN 1..r_emp.COUNT LOOP
  DBMS_OUTPUT.PUT_LINE(r_emp(i).empno || ' ' ||
    RPAD(r_emp(i).ename,8) || ' ' ||
    RPAD(r_emp(i).job,10) || ' ' ||
    TO_CHAR(r_emp(i).hiredate,'DD-MON-YY') || ' ' ||
    TO_CHAR(r_emp(i).sal,'99,999.99') || ' ' ||
    TO_CHAR(NVL(r_emp(i).comm,0),'99,999.99') || ' ' ||
    r_emp(i).deptno);
END LOOP;
END;
```

EMPNO	ENAME	JOB	HIREDATE	SAL	COMM	DEPTNO
-----	-----	-----	-----	-----	-----	-----
7369	SMITH	CLERK	17-DEC-80	1,200.00	.00	20
7876	ADAMS	CLERK	23-MAY-87	1,650.00	.00	20
7900	JAMES	CLERK	03-DEC-81	1,425.00	.00	30
7934	MILLER	CLERK	23-JAN-82	1,950.00	.00	10

12.5 Errors and Messages

メッセージを報告するには、`DBMS_OUTPUT.PUT_LINE` ステートメントを使用します。

```
DBMS_OUTPUT.PUT_LINE ( <message> );
```

`message` は文字列に評価される式です。

この例では、ユーザーの出力ディスプレイにメッセージを表示します。

```
DBMS_OUTPUT.PUT_LINE('My name is John');
```

特別な変数 `SQLCODE` と `SQLERRM` には、それぞれ最後に発行された **SQL** コマンドの結果を説明する数値コードとテキストメッセージが含まれています。プログラムでゼロ除算などの他のエラーが発生した場合、これらの変数にはエラーに関する情報が含まれます。

第 13 章

Triggers

この章では、AdvancedServer トリガーについて説明します。プロシージャおよび関数と同様に、トリガーは SPL 言語で記述されています。

13.1 Overview

トリガーは、テーブルに関連付けられ、データベースに保存される名前付きの **SPL** コードブロックです。関連付けられたテーブルで指定されたイベントが発生すると、**SPL** コードブロックが実行されます。コードブロックが実行されると、トリガーは「発動」したと言われます。

トリガーを起動するイベントは、直接または間接的にテーブルで実行される挿入、更新、または削除の任意の組み合わせです。テーブルが **SQL** の **INSERT**, **UPDATE**, **DELETE** または **TRUNCATE** コマンドのオブジェクトである場合、対応する挿入、更新、削除、または切り捨てイベントが*トリガーイベント*として定義されていると仮定して、トリガーが直接起動されます。トリガーを起動するイベントは、**CREATE TRIGGER** コマンドで定義されます。

別のテーブルで開始されたイベントの結果としてテーブルでトリガーイベントが発生した場合、トリガーを間接的に起動できます。たとえば、**ON DELETE CASCADE** 句で定義された外部キーを含むテーブルでトリガーが定義され、親テーブルの行が削除されると、親のすべての子も削除されます。削除が子テーブルのトリガーイベントである場合、子を削除するとトリガーが起動します。

13.2 Types of Triggers

AdvancedServer は、*row-level* および *statement-level* トリガーの両方をサポートしています。行レベルのトリガーは、トリガーイベントの影響を受ける行ごとに 1 回発生します。たとえば、削除がテーブルのトリガーイベントとして定義され、テーブルから 5 行を削除する単一の DELETE コマンドが発行されると、トリガーは各行に 1 回ずつ 5 回起動します。

対照的に、ステートメントレベルのトリガーは、トリガーイベントによって影響を受ける行の数に関係なく、トリガーステートメントごとに 1 回起動します。5 行を削除する単一の DELETE コマンドの前の例では、ステートメントレベルのトリガーは 1 回だけ起動します。

ステートメントレベルのトリガーの場合、トリガーコードブロックがトリガーステートメント自体の前または後に実行されるかどうかに関して、アクションのシーケンスを定義できます。または、行レベルトリガーの場合、各行がトリガーステートメントの影響を受ける前後。

before 行レベルトリガーでは、影響を受ける各行でトリガーアクションが実行される前にトリガーコードブロックが実行されます。**before* ステートメントレベルトリガーでは、トリガーステートメントのアクションが実行される前に、トリガーコードブロックが実行されます。

after 行レベルトリガーでは、影響を受ける各行でトリガーアクションが実行された後にトリガーコードブロックが実行されます。**after* ステートメントレベルトリガーでは、トリガーステートメントのアクションが実行された後にトリガーコードブロックが実行されます。

複合トリガーでは、ステートメントレベルと行レベルのトリガーを単一のトリガーで定義でき、複数のタイミングで起動できます。複合トリガーの詳細については、[CompoundTriggers](#) を参照してください。

13.3 Creating Triggers

CREATE TRIGGER コマンドは、データベースに保存されるトリガーを定義して名前を付けます。

Name

CREATE TRIGGER -- は単純なトリガーを定義します

Synopsis

```
CREATE [ OR REPLACE ] TRIGGER <name>
{ BEFORE | AFTER | INSTEAD OF }
{ INSERT | UPDATE | DELETE | TRUNCATE }
[ OR { INSERT | UPDATE | DELETE | TRUNCATE } ] [, ...]
ON <table>
[ REFERENCING { OLD AS <old> | NEW AS <new> } ...]
[ FOR EACH ROW ]
[ WHEN <condition> ]
[ DECLARE
  [ PRAGMA AUTONOMOUS_TRANSACTION; ]
  <declaration>; [, ...] ]
BEGIN
  <statement>; [, ...]
[ EXCEPTION
  { WHEN <exception> [ OR <exception> ] [...] THEN
    <statement>; [, ...] } [, ...]
]
END
```

Name

CREATE TRIGGER -- は複合トリガーを定義します

Synopsis

```
CREATE [ OR REPLACE ] TRIGGER <name>
FOR { INSERT | UPDATE | DELETE | TRUNCATE }
[ OR { INSERT | UPDATE | DELETE | TRUNCATE } ] [, ...]
ON <table>
[ REFERENCING { OLD AS <old> | NEW AS <new. > } ...]
[ WHEN <condition> ]
COMPOUND TRIGGER
[ <private_declaration>; ] ...
[ <procedure_or_function_definition> ] ...
<compound_trigger_definition>
END
```

private_declaration は、任意のプロシージャまたは関数からアクセスできるプライベート変数の識別子です。0、1、またはそれ以上のプライベート変数があります。private_declaration は次のいずれかになり

ます。

- 可変宣言
- 記録宣言
- コレクション宣言
- REF CURSOR とカーソル変数宣言
- TYPE のレコード、コレクション、および REF CURSORs の定義
- 例外
- オブジェクト変数宣言

ここで `procedure_or_function_definition:=`

`procedure_definition | function_definition`

ここで `procedure_definition:=`

```
PROCEDURE proc_name[ argument_list ]
  [ options_list ]
  { IS | AS }
  procedure_body
END [ proc_name ] ;
```

ここで `procedure_body:=`

```
[ declaration; ] [, ...]
BEGIN
  statement; [...]
[ EXCEPTION
  { WHEN exception [OR exception] [...] ] THEN statement; }
  [...]
]
```

ここで `function_definition:=`

```
FUNCTION func_name [ argument_list ]
  RETURN rettype [ DETERMINISTIC ]
  [ options_list ]
  { IS | AS }
  function_body
END [ func_name ] ;
```

ここで `function_body:=`

```
[ declaration; ] [, ...]
BEGIN
    statement; [...]
[ EXCEPTION
    { WHEN exception [ OR exception ] [...] THEN statement; }
    [...]
]
```

compound_trigger_definition は次のとおりです。

```
{ compound_trigger_event } { IS | AS }
    compound_trigger_body
END [ compound_trigger_event ] [ ... ]
```

ここで compound_trigger_event :=

```
[ BEFORE STATEMENT | BEFORE EACH ROW | AFTER EACH ROW |
    AFTER STATEMENT | INSTEAD OF EACH ROW ]
```

ここで compound_trigger_body :=

```
[ declaration; ] [, ...]
BEGIN
    statement; [...]
[ EXCEPTION
    { WHEN exception [OR exception] [...] THEN statement; }
    [...]
]
```

Description

CREATE TRIGGER は新しいトリガーを定義します。CREATE OR REPLACE TRIGGER は、新しいトリガーを作成するか、既存の定義を置き換えます。

CREATE TRIGGER キーワードを使用して新しいトリガーを作成する場合、新しいトリガーの名前は、同じテーブルで定義されている既存のトリガーと一致してはなりません。新しいトリガーは、トリガーイベントが定義されているテーブルと同じスキーマに作成されます。

既存のトリガーの定義を更新する場合は、CREATE OR REPLACE TRIGGER キーワードを使用します。

Oracle データベースと互換性のある構文を使用してトリガーを作成すると、トリガーは SECURITY DEFINER 関数として実行されます。

Parameters

name

作成するトリガーの名前。

BEFORE | AFTER

トリガーイベントの前または後にトリガーを起動するかどうかを決定します。

INSTEAD OF

INSTEAD OF トリガーは更新可能なビューを変更します。トリガーが実行され、基になるテーブルが適切に更新されます。INSTEAD OF トリガーは、更新または変更されたビューの行ごとに実行されます。

INSERT | UPDATE | DELETE | TRUNCATE

トリガーイベントを定義します。

table

トリガーイベントが発生するテーブルまたはビューの名前。

condition

condition は、トリガーが実際に実行されるかどうかを決定するブール式です。condition が TRUE と評価されると、トリガーが起動します。

単純なトリガー定義に FOR EACH ROW キーワードが含まれる場合、WHEN 句は OLD.column_name または NEW.column_name をそれぞれ書き込むことにより、古い行値または新しい行値、あるいはその両方の列を参照できます。INSERT トリガーは OLD を参照できず、DELETE トリガーは NEW を参照できません。

複合トリガー定義に WHEN 句を持つ文レベルのトリガーが含まれている場合、WHEN 句の式を評価せずにトリガーが実行されます。同様に、複合トリガー定義に WHEN 句を持つ行レベルのトリガーが含まれている場合、式が TRUE と評価されるとトリガーが実行されます。

トリガーに INSTEAD OF キーワードが含まれる場合、WHEN 句が含まれない場合があります。WHEN 句にサブクエリを含めることはできません。

REFERENCING { OLD AS old | NEW AS new } ...

古い行と新しい行を参照するための REFERENCING 節。ただし、old は old という名前の識別子またはすべて小文字で保存される同等のもの（REFERENCING OLD AS old, REFERENCING OLD AS OLD や REFERENCING OLD AS /"old/"）など）また、new は、new という名前の識別子、またはすべて小文字で保存された同等のもの（REFERENCING NEW AS new, REFERENCING NEW AS NEW、REFERENCING NEW AS /"new/"）などにのみ置き換えることができます。

REFERENCING 句で OLD AS old と NEW AS new のいずれか、または両方のフレーズを指定できます（REFERENCING NEW AS New OLD AS Old）など。この句は、古いまたは新しい以外の識別子は使用できないため、Oracle データベースと互換性がありません。

FOR EACH ROW

トリガーイベントによって影響を受けるすべての行に対してトリガーを 1 回起動するか、SQL ステートメントごとに 1 回だけ起動するかを決定します。指定した場合、トリガーは影響を受ける行ごとに 1 回起動されます（行レベルトリガー）。それ以外の場合、トリガーはステートメントレベルのトリガーです。

PRAGMA AUTONOMOUS_TRANSACTION

PRAGMA AUTONOMOUS_TRANSACTION は、トリガーを自律型トランザクションとして設定するディレクティブです。

declaration

変数、タイプ、REF CURSOR、またはサブプログラム宣言。サブプログラム宣言が含まれる場合、他のすべての変数、型、および REF CURSOR 宣言の後に宣言する必要があります。

statement

SPL プログラムステートメント。DECLARE - BEGIN - END ブロックは、それ自体が SPL ステートメントと見なされることに注意してください。したがって、トリガー本体にはネストされたブロックが含まれる場合があります。

exception

NO_DATA_FOUND, OTHERS などの例外条件名

13.4 Trigger Variables

トリガーコードブロックでは、いくつかの特別な変数を使用できます。

NEW

NEW は、行レベルトリガーでの挿入および更新操作の新しいテーブル行を参照する擬似レコード名です。この変数は、文レベルのトリガーおよび行レベルのトリガーの削除操作には適用されません。

その使用法は、:NEW.column です。column は、トリガーが定義されているテーブル内の列の名前です。

:NEW.column の初期コンテンツは、挿入される新しい行の名前付き列の値、または前の行レベルトリガーで使用されたときに古い行を置き換える新しい行の値です。影響を受ける行でアクションが既に発生しているため、後行レベルのトリガーで使用する場合、この値はすでにテーブルに格納されています。

トリガーコードブロックでは、:NEW.column を他の変数と同様に使用できます。値が :NEW.column に割り当てられている場合、前の行レベルトリガーのコードブロックで、割り当てられた値が新しい挿入または更新された行で使用されます。

OLD

OLD は、行レベルトリガーでの更新および削除操作のために古いテーブル行を参照する擬似レコード名です。この変数は、文レベルのトリガーおよび行レベルのトリガーの挿入操作には適用できません。

その使用法は、`:OLD.column` です。`column` は、トリガーが定義されているテーブル内の列の名前です。

`:OLD.column` の初期コンテンツは、削除される行の名前付き列の値、または前行レベルのトリガーで使用されたときに新しい行に置き換えられる古い行の値です。後行レベルのトリガーで使用すると、影響を受ける行でアクションが既に発生しているため、この値はテーブルに保存されなくなります。

トリガーコードブロックでは、`:OLD.column` を他の変数と同様に使用できます。`:OLD.column` に値を割り当てても、トリガーのアクションには影響しません。

INSERTING

INSERTING は、挿入操作がトリガーを起動した場合に `TRUE` を返す条件式であり、そうでない場合は `FALSE` を返します。

UPDATING

UPDATING は、更新操作がトリガーを起動した場合に `TRUE` を返す条件式であり、そうでない場合は `FALSE` を返します。

DELETING

DELETING は、削除操作がトリガーを起動した場合に `TRUE` を返す条件式であり、そうでない場合は `FALSE` を返します。

13.5 Transactions and Exceptions

トリガーは常に、トリガーを起動する文が実行されている同じトランザクションの一部として実行されます。トリガーコードブロック内で例外が発生しない場合、トリガーステートメントを含むトランザクションがコミットされた場合にのみ、トリガー内のトリガーコマンドの効果がコミットされます。したがって、トランザクションがロールバックされると、トリガー内のトリガーコマンドの効果もロールバックされます。

トリガーコードブロック内で例外が発生したが、例外セクションでキャッチされて処理された場合でも、トリガー内のトリガーコマンドの効果は引き続きロールバックされます。ただし、アプリケーションがカプセル化トランザクションのロールバックを強制しない限り、トリガーを実行するステートメント自体はロールバックされません。

未処理の例外がトリガーコードブロック内で発生した場合、トリガーをカプセル化するトランザクションは中止され、ロールバックされます。したがって、トリガー内のトリガーコマンドおよびトリガーステートメント自体の効果自体はすべてロールバックされます。

13.6 Compound Triggers

AdvancedServer は、複合トリガーをサポートする互換性のある構文を追加しました。複合トリガーは、1 つ以上の*タイミングポイント* で呼び出すことができる 1 つのトリガー本体の下で、すべてのトリガータイミングを組み

合わせます。タイミングポイントは、トリガーステートメント（データを変更する INSERT, UPDATE, DELETE または TRUNCATE ステートメント）に関連する時点です。サポートされるタイミングポイントは次のとおりです。

- BEFORE STATEMENT: トリガーを起動する文が実行される前。
- BEFORE EACH ROW: トリガーステートメントが影響する各行の前。
- AFTER EACH ROW: トリガーをかけるステートメントが影響する各行の後。
- AFTER STATEMENT: トリガーを起動する文の実行後。
- INSTEAD OF EACH ROW: トリガーステートメントによって影響を受けるすべての行に対してトリガーが 1 回起動します。

複合トリガーには、単一のトリガーで定義されたタイミングポイントの任意の組み合わせを含めることができます。

複合トリガーのオプションの宣言セクションでは、トリガーレベルの変数とサブプログラムを宣言できます。宣言の内容は、トリガー定義によって参照されるすべてのタイミングポイントにアクセスできます。宣言によって作成された変数およびサブプログラムは、トリガーを起動する文の間のみ持続します。

複合トリガーには宣言が含まれ、その後に各タイミングポイントの PL ブロックが続きます。

```
CREATE OR REPLACE TRIGGER compound_trigger_name
FOR INSERT OR UPDATE OR DELETE ON table_name
COMPOUND TRIGGER
    -- Global Declaration Section (optional)
    -- Variables declared here can be used inside any timing-point blocks.

    BEFORE STATEMENT IS
    BEGIN
        NULL;
    END BEFORE STATEMENT;

    BEFORE EACH ROW IS
    BEGIN
        NULL;
    END BEFORE EACH ROW;

    AFTER EACH ROW IS
    BEGIN
        NULL;
    END AFTER EACH ROW;

    AFTER STATEMENT IS
    BEGIN
        NULL;
    END AFTER STATEMENT;
```

(次のページに続く)

(前のページからの続き)

```
END compound_trigger_name;  
/  
Trigger created.
```

注釈: 4つのタイミングブロックをすべて持つことは必須ではありません。必要なタイミングポイントの複合トリガーを作成できます。

複合トリガーには次の制限があります。

- 複合トリガー本体は、複合トリガーブロックで構成されます。
- 複合トリガーは、テーブルまたはビューで定義できます。
- 例外は他のタイミングポイントセクションに転送できないため、各複合トリガーブロックによってのみそのセクションで個別に処理する必要があります。
- GOTO ステートメントがタイミングポイントセクションで指定されている場合、GOTO ステートメントのターゲットも同じタイミングポイントセクションで指定する必要があります。
- :OLD および :NEW 変数識別子は、宣言セクション、BEFORE STATEMENT セクション、または AFTER STATEMENT セクションには存在できません。
- :NEW の値は、BEFORE EACH ROW ブロックによってのみ変更できます。
- 複合トリガーのタイミングポイント実行のシーケンスは特定ですが、同じタイミングポイント内に単純トリガーが存在する場合、単純トリガーが最初に起動され、次に複合トリガーの起動が続きます。

13.7 Trigger Examples

次のセクションでは、各タイプのトリガーの例を示します。

13.7.1 Before Statement-Level Trigger

以下は、emp テーブルでの挿入操作の前にメッセージを表示する単純な before ステートメントレベルトリガーの例です。

```
CREATE OR REPLACE TRIGGER emp_alert_trig
  BEFORE INSERT ON emp
BEGIN
  DBMS_OUTPUT.PUT_LINE('New employees are about to be added');
END;
```

次の INSERT は、コマンドの 1 回の実行で複数の新しい行が挿入されるように構成されています。7900～7999 の従業員 ID を持つ行ごとに、従業員 ID を 1000 増やした新しい行が挿入されます。次は、3 つの新しい行が挿入されたときにコマンドを実行した結果です。

```
INSERT INTO emp (empno, ename, deptno) SELECT empno + 1000, ename, 40
  FROM emp WHERE empno BETWEEN 7900 AND 7999;
New employees are about to be added

SELECT empno, ename, deptno FROM emp WHERE empno BETWEEN 8900 AND 8999;
```

EMPNO	ENAME	DEPTNO
8900	JAMES	40
8902	FORD	40
8934	MILLER	40

メッセージ New employees are about to be added は、結果が 3 つの新しい行の追加であっても、トリガーの起動によって 1 回表示されます。

13.7.2 After Statement-Level Trigger

以下は、ステートメント後レベルのトリガーの例です。emp テーブルで挿入、更新、または削除の操作が発生すると、empauditlog テーブルに行が追加され、日付、ユーザー、およびアクションが記録されます。

```
CREATE TABLE empauditlog (
  audit_date      DATE,
  audit_user      VARCHAR2(20),
  audit_desc      VARCHAR2(20)
```

(次のページに続く)

(前のページからの続き)

```
);
CREATE OR REPLACE TRIGGER emp_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    v_action          VARCHAR2(20);
BEGIN
    IF INSERTING THEN
        v_action := 'Added employee(s)';
    ELSIF UPDATING THEN
        v_action := 'Updated employee(s)';
    ELSIF DELETING THEN
        v_action := 'Deleted employee(s)';
    END IF;
    INSERT INTO empauditlog VALUES (SYSDATE, USER,
        v_action);
END;
```

次の一連のコマンドでは、2つの INSERT コマンドを使用して emp テーブルに 2 つの行が挿入されます。両方の行の sal 列と comm 列は、1 つの UPDATE コマンドで更新されます。最後に、1 つの DELETE コマンドで両方の行が削除されます。

```
INSERT INTO emp VALUES (9001,'SMITH','ANALYST',7782,SYSDATE,NULL,NULL,10);

INSERT INTO emp VALUES (9002,'JONES','CLERK',7782,SYSDATE,NULL,NULL,10);

UPDATE emp SET sal = 4000.00, comm = 1200.00 WHERE empno IN (9001, 9002);

DELETE FROM emp WHERE empno IN (9001, 9002);

SELECT TO_CHAR(AUDIT_DATE,'DD-MON-YY HH24:MI:SS') AS "AUDIT DATE",
       audit_user, audit_desc FROM empauditlog ORDER BY 1 ASC;
```

AUDIT DATE	AUDIT_USER	AUDIT_DESC
31-MAR-05 14:59:48	SYSTEM	Added employee(s)
31-MAR-05 15:00:07	SYSTEM	Added employee(s)
31-MAR-05 15:00:19	SYSTEM	Updated employee(s)
31-MAR-05 15:00:34	SYSTEM	Deleted employee(s)

empauditlog テーブルの内容は、トリガーが起動された回数を示します-2 回の挿入ごとに 1 回、更新（2 行が変更された場合でも）、削除（2 行が削除された場合でも）に 1 回

13.7.3 Before Row-Level Trigger

次の例は、emp テーブルに挿入される部門 30 に属するすべての新しい従業員の歩合を計算する前行レベルのトリガーです。

```
CREATE OR REPLACE TRIGGER emp_comm_trig
  BEFORE INSERT ON emp
  FOR EACH ROW
BEGIN
  IF :NEW.deptno = 30 THEN
    :NEW.comm := :NEW.sal * .4;
  END IF;
END;
```

2 人の従業員を追加した後のリストは、トリガーがコミッションを計算し、それを新しい従業員行の一部として挿入したことを示しています。

```
INSERT INTO emp VALUES
(9005, 'ROBERS', 'SALESMAN', 7782, SYSDATE, 3000.00, NULL, 30);

INSERT INTO emp VALUES
(9006, 'ALLEN', 'SALESMAN', 7782, SYSDATE, 4500.00, NULL, 30);

SELECT * FROM emp WHERE empno IN (9005, 9006);
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
9005	ROBERS	SALESMAN	7782	01-APR-05	3000	1200	30
9006	ALLEN	SALESMAN	7782	01-APR-05	4500	1800	30

13.7.4 After Row-Level Trigger

次の例は、行レベル後のトリガーです。新しい従業員の行が挿入されると、トリガーはその従業員の jobhist テーブルに新しい行を追加します。既存の従業員が更新されると、トリガーは最新の jobhist 行の enddate 列（enddate が null であると仮定）を現在の日付に設定し、従業員の jobhist 行を挿入します新しい情報。

最後に、トリガーはアクションの説明とともに empchglog テーブルに行を追加します。

```
CREATE TABLE empchglog (
  chg_date      DATE,
  chg_desc      VARCHAR2(30)
);
CREATE OR REPLACE TRIGGER emp_chg_trig
  AFTER INSERT OR UPDATE OR DELETE ON emp
  FOR EACH ROW
```

(次のページに続く)

(前のページからの続き)

```

DECLARE
    v_empno          emp.empno%TYPE;
    v_deptno         emp.deptno%TYPE;
    v_dname          dept.dname%TYPE;
    v_action         VARCHAR2(7);
    v_chgdesc        jobhist.chgdesc%TYPE;
BEGIN
    IF INSERTING THEN
        v_action := 'Added';
        v_empno := :NEW.empno;
        v_deptno := :NEW.deptno;
        INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
                                     :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, 'New Hire');
    ELSIF UPDATING THEN
        v_action := 'Updated';
        v_empno := :NEW.empno;
        v_deptno := :NEW.deptno;
        v_chgdesc := '';
        IF NVL(:OLD.ename, '-null-') != NVL(:NEW.ename, '-null-') THEN
            v_chgdesc := v_chgdesc || 'name, ';
        END IF;
        IF NVL(:OLD.job, '-null-') != NVL(:NEW.job, '-null-') THEN
            v_chgdesc := v_chgdesc || 'job, ';
        END IF;
        IF NVL(:OLD.sal, -1) != NVL(:NEW.sal, -1) THEN
            v_chgdesc := v_chgdesc || 'salary, ';
        END IF;
        IF NVL(:OLD.comm, -1) != NVL(:NEW.comm, -1) THEN
            v_chgdesc := v_chgdesc || 'commission, ';
        END IF;
        IF NVL(:OLD.deptno, -1) != NVL(:NEW.deptno, -1) THEN
            v_chgdesc := v_chgdesc || 'department, ';
        END IF;
        v_chgdesc := 'Changed ' || RTRIM(v_chgdesc, ', ');
        UPDATE jobhist SET enddate = SYSDATE WHERE empno = :OLD.empno
            AND enddate IS NULL;
        INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
                                     :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, v_chgdesc);
    ELSIF DELETING THEN
        v_action := 'Deleted';
        v_empno := :OLD.empno;
        v_deptno := :OLD.deptno;
    END IF;

    INSERT INTO empchglog VALUES (SYSDATE,
                                   v_action || ' employee # ' || v_empno);
END;

```

以下に示す最初のコマンドシーケンスでは、2 人の従業員が 2 つの個別の INSERT コマンドを使用して追加され、その後両方が単一の UPDATE コマンドを使用して更新されます。jobhist テーブルの内容は、影響を受ける各行のトリガーのアクションを示しています。2 人の新しい従業員の 2 つの新しい採用エントリと、2 人の従業員の更新されたコミッションの 2 つの変更されたコミッションレコードです。empchglog テーブルには、トリガーが 2 行のアクションごとに 1 回、合計 4 回起動されたことが示されています。

```
INSERT INTO emp VALUES (9003, 'PETERS', 'ANALYST', 7782, SYSDATE, 5000.00, NULL, 40);

INSERT INTO emp VALUES (9004, 'AIKENS', 'ANALYST', 7782, SYSDATE, 4500.00, NULL, 40);

UPDATE emp SET comm = sal * 1.1 WHERE empno IN (9003, 9004);

SELECT * FROM jobhist WHERE empno IN (9003, 9004);
```

EMPNO	STARTDATE	ENDDATE	JOB	SAL	COMM	DEPTNO	CHGDESC
9003	31-MAR-05	31-MAR-05	ANALYST	5000		40	New Hire
9004	31-MAR-05	31-MAR-05	ANALYST	4500		40	New Hire
9003	31-MAR-05		ANALYST	5000	5500	40	Changed commission
9004	31-MAR-05		ANALYST	4500	4950	40	Changed commission

```
SELECT * FROM empchglog;
```

CHG_DATE	CHG_DESC
31-MAR-05	Added employee # 9003
31-MAR-05	Added employee # 9004
31-MAR-05	Updated employee # 9003
31-MAR-05	Updated employee # 9004

最後に、両方の従業員が単一の DELETE コマンドで削除されます。empchglog テーブルには、削除された従業員ごとに 1 回ずつ、トリガーが 2 回起動されたことが表示されます。

```
DELETE FROM emp WHERE empno IN (9003, 9004);

SELECT * FROM empchglog;
```

CHG_DATE	CHG_DESC
31-MAR-05	Added employee # 9003
31-MAR-05	Added employee # 9004
31-MAR-05	Updated employee # 9003
31-MAR-05	Updated employee # 9004

(次のページに続く)

(前のページからの続き)

```
31-MAR-05 Deleted employee # 9003
31-MAR-05 Deleted employee # 9004
```

13.7.5 INSTEAD OF Trigger

次の例は、新しい従業員の行を emp_vw ビューに挿入するための INSTEAD OF トリガーを示しています。CREATE VIEW ステートメントは、2つのテーブルを結合して emp_vw ビューを作成します。トリガーは、特定の従業員の emp および dept テーブルにそれぞれ対応する新しい行を追加します。

```
CREATE VIEW emp_vw AS SELECT * FROM emp e JOIN dept d USING(deptno);
CREATE VIEW

CREATE OR REPLACE TRIGGER empvw_instead_of_trig
    INSTEAD OF INSERT ON emp_vw
    FOR EACH ROW
DECLARE
    v_empno      emp.empno%TYPE;
    v_ename      emp.ename%TYPE;
    v_deptno     emp.deptno%TYPE;
    v_dname      dept.dname%TYPE;
    v_loc        dept.loc%TYPE;
    v_action     VARCHAR2(7);
BEGIN
    v_empno      := :NEW.empno;
    v_ename      := :New.ename;
    v_deptno     := :NEW.deptno;
    v_dname      := :NEW.dname;
    v_loc        := :NEW.loc;
    INSERT INTO emp(empno, ename, deptno) VALUES(v_empno, v_ename,
v_deptno);
    INSERT INTO dept(deptno, dname, loc) VALUES(v_deptno, v_dname, v_loc);
END;
CREATE TRIGGER
```

次に、値を emp_vw ビューに挿入します。挿入アクションは新しい行を挿入し、次の出力を生成します。

```
INSERT INTO emp_vw (empno, ename, deptno, dname, loc ) VALUES(1234,
'ASHTON', 50, 'IT', 'NEW JERSEY');
INSERT 0 1
```

```
SELECT empno, ename, deptno FROM emp  WHERE deptno = 50;
 empno | ename   | deptno
-----+-----+-----
  1234 | ASHTON |     50
(1 row)
```

```
SELECT * FROM dept WHERE deptno = 50;
 deptno |  dname  |    loc
-----+-----+-----
      50 |  IT     | NEW JERSEY
(1 row)
```

同様に、UPDATE または DELETE ステートメントを指定すると、トリガーは UPDATE または DELETE イベントに対して適切なアクションを実行します。

13.7.6 Compound Triggers

次の複合トリガーの例では、emp テーブルに複合トリガー（hr_trigger という名前）を定義することにより、従業員の給与の変更を記録します。

まず、emp という名前のテーブルを作成します。

```
CREATE TABLE emp(EMPNO INT, ENAME TEXT, SAL INT, DEPTNO INT);
CREATE TABLE
```

次に、hr_trigger という名前の複合トリガーを作成します。トリガーは、4 つのタイミングポイントのそれぞれを使用して、INSERT、UPDATE、または DELETE ステートメントで給与を変更します。グローバル宣言セクションでは、初期給与は 10,000 と宣言されています。

```
CREATE OR REPLACE TRIGGER hr_trigger
FOR INSERT OR UPDATE OR DELETE ON emp
COMPOUND TRIGGER
-- Global declaration.
var_sal NUMBER := 10000;

BEFORE STATEMENT IS
BEGIN
    var_sal := var_sal + 1000;
    DBMS_OUTPUT.PUT_LINE('Before Statement: ' || var_sal);
END BEFORE STATEMENT;

BEFORE EACH ROW IS
BEGIN
    var_sal := var_sal + 1000;
    DBMS_OUTPUT.PUT_LINE('Before Each Row: ' || var_sal);
END BEFORE EACH ROW;

AFTER EACH ROW IS
BEGIN
    var_sal := var_sal + 1000;
    DBMS_OUTPUT.PUT_LINE('After Each Row: ' || var_sal);
```

(次のページに続く)

(前のページからの続き)

```

END AFTER EACH ROW;

AFTER STATEMENT IS
BEGIN
    var_sal := var_sal + 1000;
    DBMS_OUTPUT.PUT_LINE('After Statement: ' || var_sal);
END AFTER STATEMENT;

END hr_trigger;

Output: Trigger created.

```

INSERT レコードをテーブル emp に入れます。

```
INSERT INTO emp (EMPNO, ENAME, SAL, DEPTNO) VALUES (1111, 'SMITH', 10000, 20);
```

INSERT ステートメントは、次の出力を生成します。

```

Before Statement: 11000
Before each row: 12000
After each row: 13000
After statement: 14000
INSERT 0 1

```

UPDATE ステートメントは従業員の給与レコードを更新し、特定の従業員番号の給与を 15000 に設定します。

```
UPDATE emp SET SAL = 15000 where EMPNO = 1111;
```

UPDATE ステートメントは、次の出力を生成します。

```

Before Statement: 11000
Before each row: 12000
After each row: 13000
After statement: 14000
UPDATE 1

SELECT * FROM emp;
  EMPNO | ENAME |   SAL   | DEPTNO
-----+-----+-----+-----
   1111 | SMITH | 15000   |      20
(1 row)

```

DELETE ステートメントは、従業員の給与レコードを削除します。

```
DELETE from emp where EMPNO = 1111;
```

DELETE ステートメントは、次の出力を生成します。

```

Before Statement: 11000
Before each row: 12000
After each row: 13000
After statement: 14000
DELETE 1

SELECT * FROM emp;
  EMPNO | ENAME | SAL | DEPTNO
-----+-----+-----+-----
(0 rows)

```

TRUNCATE ステートメントは、emp テーブルからすべてのレコードを削除します。

```

CREATE OR REPLACE TRIGGER hr_trigger
FOR TRUNCATE ON emp
  COMPOUND TRIGGER
-- Global declaration.
var_sal NUMBER := 10000;
BEFORE STATEMENT IS
BEGIN
  var_sal := var_sal + 1000;
  DBMS_OUTPUT.PUT_LINE('Before Statement: ' || var_sal);
END BEFORE STATEMENT;

AFTER STATEMENT IS
BEGIN
  var_sal := var_sal + 1000;
  DBMS_OUTPUT.PUT_LINE('After Statement: ' || var_sal);
END AFTER STATEMENT;

END hr_trigger;

Output: Trigger created.

```

TRUNCATE ステートメントは、次の出力を生成します。

```

TRUNCATE emp;
Before Statement: 11000
After statement: 12000
TRUNCATE TABLE

```

注釈: TRUNCATE ステートメントは、BEFORE STATEMENT または AFTER STATEMENT のタイミングポイントでのみ使用できます。

次の例では、emp テーブルに hr_trigger という名前の複合トリガーを作成し、INSERT、UPDATE、また

は DELETE ステートメントが emp テーブルに影響するたびに従業員の給与をチェックおよび出力する「10」条件を設定します。データベースは、行レベルトリガーの「10」条件を評価し、「10」条件が TRUE と評価された場合、トリガーは行ごとに 1 回実行されます。ステートメントレベルのトリガーは、「10」条件に関係なく実行されます。

```
CREATE OR REPLACE TRIGGER hr_trigger
FOR INSERT OR UPDATE OR DELETE ON emp
REFERENCING NEW AS new OLD AS old
WHEN (old.sal > 5000 OR new.sal < 8000)
    COMPOUND TRIGGER

BEFORE STATEMENT IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Before Statement');
END BEFORE STATEMENT;

BEFORE EACH ROW IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Before Each Row: ' || :OLD.sal || ' ' || :NEW.sal);
END BEFORE EACH ROW;

AFTER EACH ROW IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('After Each Row: ' || :OLD.sal || ' ' || :NEW.sal);
END AFTER EACH ROW;

AFTER STATEMENT IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('After Statement');
END AFTER STATEMENT;

END hr_trigger;
```

レコードをテーブル emp に挿入します。

```
INSERT INTO emp(EMPNO, ENAME, SAL, DEPTNO) VALUES(1111, 'SMITH', 1600, 20);
```

INSERT ステートメントは、次の出力を生成します。

```
Before Statement
Before Each Row: 1600
After Each Row: 1600
After Statement
INSERT 0 1
```

UPDATE ステートメントは、従業員の給与レコードを更新し、給与を 7500 に設定します。

```
UPDATE emp SET SAL = 7500 where EMPNO = 1111;
```

UPDATE ステートメントは、次の出力を生成します。

```
Before Statement
Before Each Row: 1600 7500
After Each Row: 1600 7500
After Statement
UPDATE 1

SELECT * from emp;
 empno |  ename  | sal  | deptno
-----+-----+-----+-----
    1111 | SMITH   | 7500 |      20
(1 row)
```

DELETE ステートメントは、従業員の給与レコードを削除します。

```
DELETE from emp where EMPNO = 1111;
```

DELETE ステートメントは、次の出力を生成します。

```
Before Statement
Before Each Row: 7500
After Each Row: 7500
After Statement
DELETE 1

SELECT * from emp;
 empno |  ename  | sal  | deptno
-----+-----+-----+-----
(0 rows)
```


第 14 章

Packages

AdvancedServer は、Oracle パッケージとの互換性を提供するパッケージのコレクションを提供します。

パッケージ は、関数、プロシージャ、変数、カーソル、ユーザー定義のレコードタイプ、および共通の修飾子（パッケージ識別子）を使用して参照されるレコードの名前付きコレクションです。パッケージには次の特徴があります。

- パッケージは、関連する目的を実行する機能と手順を整理する便利な手段を提供します。パッケージの関数とプロシージャを使用する権限は、パッケージ全体に付与された 1 つの特権に依存します。すべてのパッケージプログラムは、共通名で参照する必要があります。
- パッケージ内の特定の関数、プロシージャ、変数、型などは、*public* として宣言できます。パブリックエンティティは表示されており、パッケージに対する EXECUTE 特権が付与されている他のプログラムから参照できます。パブリック関数およびパブリックプロシージャの場合、署名のみが表示されます-プログラム名、パラメータ（存在する場合）、関数の戻り値の型。これらの関数とプロシージャの SPL コードは他のユーザーがアクセスできないため、パッケージを使用するアプリケーションは、プロシージャロジック自体ではなく、署名で使用可能な情報のみに依存します。
- パッケージ内の他の関数、プロシージャ、変数、型などは、*private* として宣言できます。プライベートエンティティは、パッケージ内の関数とプロシージャで参照および使用できますが、他の外部アプリケーションでは使用できません。プライベートエンティティは、パッケージ内のプログラムでのみ使用されます。
- 関数およびプロシージャ名は、パッケージ内でオーバーロードできます。1 つ以上の関数/プロシージャを同じ名前で作成できますが、署名は異なります。これにより、同じジョブを実行するが異なるタイプの入力である同じ名前のプログラムを作成する機能が提供されます。

AdvancedServer が提供するパッケージサポートの詳細については、次の Web サイトから入手できる *OracleDevelopersBuilt-inPackageGuide のデータベース互換性* を参照してください。

<https://www.enterprisedb.com/edb-docs>

ビルトインパッケージのリストについては、*OracleDevelopers* ビルトインパッケージガイドのデータベース互換性の"/"ビルトインパッケージ/"の目次を参照してください。

第 15 章

Object Types and Objects

この章では、SPL でオブジェクト指向プログラミング手法を活用する方法について説明します。Java や C++ などのプログラミング言語で見られるオブジェクト指向プログラミングは、オブジェクトの概念を中心にしています。オブジェクト*は、人、場所、物などの実世界のエンティティの表現です。たとえば、人物などの特定のオブジェクトの一般的な説明または定義は、*オブジェクトタイプ*と呼ばれます。「Joe」や「Sally」などの特定の人々は、*オブジェクトタイプのオブジェクト、人物、または同等のオブジェクトタイプ、人物、または単に人物オブジェクトの*インスタンス*と呼ばれます。

注釈: これまでこのドキュメントで使用されていた「データベースオブジェクト」および「オブジェクト」という用語を、この章で使用されているオブジェクトタイプおよびオブジェクトと混同しないでください。これらの用語の以前の使用法は、テーブル、ビュー、インデックス、ユーザーなどのデータベースで作成できるエンティティに関連しています。この章のコンテキスト内で、オブジェクトタイプとオブジェクトは、SPL プログラミングでサポートされる特定のデータ構造を指しますオブジェクト指向の概念を実装する言語。

注釈: Oracle では、*abstractdatatype* (ADT) という用語は、PL/SQL でオブジェクト型を記述するために使用されます。オブジェクト型の SPL 実装は、Oracle の抽象データ型と互換性があることを目的としています。

注釈: AdvancedServer は、オブジェクト指向プログラミング言語の一部の機能のサポートをまだ実装していません。この章では、実装されている機能のみを文書化します。

15.1 Basic Object Concepts

オブジェクトタイプは、エンティティの説明または定義です。オブジェクトタイプのこの定義は、2つのコンポーネントによって特徴付けられます。

- **属性** –オブジェクトインスタンスの特定の特性を記述するフィールド。人物オブジェクトの例には、名前、住所、性別、生年月日、身長、体重、目の色、職業などがあります。
- **メソッド** –オブジェクトに対して、またはオブジェクトに関連する何らかのタイプの機能または操作を実行するプログラム。人物オブジェクトの例には、人物の年齢の計算、人物の属性の表示、人物の属性に割り当てられた値の変更などがあります。

次のセクションでは、オブジェクトの基本的な概念について詳しく説明します。

15.1.1 Attributes

すべてのオブジェクトタイプには、少なくとも 1 つの属性が含まれている必要があります。属性のデータ型は次のいずれかです。

- NUMBER, VARCHAR2 などの基本データ型
- 別のオブジェクトタイプ
- ネストされたテーブルや VARRAY などのグローバルに定義されたコレクションタイプ (CREATE TYPE コマンドで作成)

属性は、オブジェクトインスタンスが最初に作成されるときに初期値 (null の場合もあります) を取得します。各オブジェクトインスタンスには、独自の属性値のセットがあります。

15.1.2 Methods

メソッドは、オブジェクトタイプ内で定義された SPL プロシージャまたは関数です。メソッドは、次の 3 つの一般的なタイプに分類できます。

- **メンバーメソッド** –オブジェクトインスタンスのコンテキスト内で動作するプロシージャまたは関数。メンバーメソッドは、アクセスしているオブジェクトメソッドの属性を変更できます。
- **静的メソッド** –特定のオブジェクトインスタンスから独立して動作するプロシージャまたは関数。静的メソッドにはアクセスできず、オブジェクトインスタンスの属性を変更できません。
- **コンストラクタメソッド** –オブジェクトタイプのインスタンスを作成するために使用される関数。オブジェクトタイプが定義されると、デフォルトのコンストラクターメソッドが常に提供されます。

15.1.3 Overloading Methods

オブジェクト型では、同じ型 (これは、手続きまたは関数) の 2 つ以上の同じ名前のメソッドを定義できますが、署名は異なります。このようなメソッドは*オーバーロード*メソッドと呼ばれます。

メソッドの署名は、仮パラメーターの数、その仮パラメーターのデータ型、およびその順序で構成されます。

15.2 Object Type Components

オブジェクトタイプは、SPL 言語の次の 2 つの構造を使用して作成され、データベースに保存されます。

- オブジェクトタイプ仕様-これは、オブジェクトタイプの属性とメソッドシグネチャを指定するパブリックインターフェイスです。
- オブジェクト型の本体-これには、オブジェクト型の仕様で指定されたメソッドの実装が含まれます。

次のセクションでは、オブジェクトタイプ仕様とオブジェクトタイプ本体を作成するために使用されるコマンドについて説明します。

15.2.1 Object Type Specification Syntax

次に、オブジェクトタイプの指定の構文を示します。

```
CREATE [ OR REPLACE ] TYPE <name>
  [ AUTHID { DEFINER | CURRENT_USER } ]
  { IS | AS } OBJECT
  ( { <attribute> { <datatype> | <objtype> | <collecttype> } }
    [, ...]
    [ <method_spec> ] [, ...]
    [ <constructor> ] [, ...]
  ) [ [ NOT ] { FINAL | INSTANTIABLE } ] ...;
```

method_spec は次のとおりです。

```
[ [ NOT ] { FINAL | INSTANTIABLE } ] ...
[ OVERRIDING ]
  <subprogram_spec>
```

subprogram_spec は次のとおりです。

```
{ MEMBER | STATIC }
{ PROCEDURE <proc_name>
  [ ( [ SELF [ IN | IN OUT ] <name> ]
    [, <parm1> [ IN | IN OUT | OUT ] <datatype1>
      [ DEFAULT <value1> ] ]
    [, <parm2> [ IN | IN OUT | OUT ] <datatype2>
      [ DEFAULT <value2> ]
    ] ...)
  ]
}
|
FUNCTION <func_name>
  [ ( [ SELF [ IN | IN OUT ] <name> ]
    [, <parm1> [ IN | IN OUT | OUT ] <datatype1>
```

(次のページに続く)

(前のページからの続き)

```

        [ DEFAULT <value1> ] ]
    [, <parm2> [ IN | IN OUT | OUT ] <datatype2>
        [ DEFAULT <value2> ]
    ] ...)
]
RETURN <return_type>
}

```

constructor は次のとおりです。

```

CONSTRUCTOR <func_name>
[ ( [ SELF [ IN | IN OUT ] <name> ]
    [, <parm1> [ IN | IN OUT | OUT ] <datatype1>
        [ DEFAULT <value1> ] ]
    [, <parm2> [ IN | IN OUT | OUT ] <datatype2>
        [ DEFAULT <value2> ]
    ] ...)
]
RETURN self AS RESULT

```

注釈: 現在、OR REPLACE オプションを使用して、既存のオブジェクトタイプの属性を追加、削除、または変更することはできません。DROP TYPE コマンドを使用して、最初に既存のオブジェクトタイプを削除します。OR REPLACE オプションを使用して、既存のオブジェクトタイプのメソッドを追加、削除、または変更できます。

注釈: ALTER TYPE ALTER ATTRIBUTE コマンドの PostgreSQL 形式を使用して、既存のオブジェクトタイプの属性のデータタイプを変更できます。ただし、ALTER TYPE コマンドはオブジェクトタイプの属性を追加または削除できません。

name は、オブジェクトタイプに割り当てられた識別子（オプションでスキーマ修飾）です。

AUTHID 句を省略するか DEFINER を指定すると、オブジェクトタイプの所有者の権限を使用して、データベースオブジェクトへのアクセス権限が決定されます。CURRENT_USER を指定すると、オブジェクト内のメソッドを実行している現在のユーザーの権限を使用して、アクセス権限が決定されます。

attribute は、オブジェクトタイプの属性に割り当てられた識別子です。

datatype は基本データ型です。

objtype は以前に定義されたオブジェクトタイプです。

collecttype は以前に定義されたコレクションタイプです。

CREATE TYPE 定義の閉じ括弧に続いて、[NOT] FINAL はサブタイプをこのオブジェクトタイプから派生

できるかどうかを指定します。デフォルトの `FINAL` は、このオブジェクトタイプからサブタイプを派生できないことを意味します。このオブジェクトタイプでサブタイプを定義できるようにする場合は、`NOT FINAL` を指定します。

注釈: `NOT FINAL` の指定は `CREATE TYPE` コマンドで受け入れられますが、**SPL** は現在サブタイプの作成をサポートしていません。

`CREATE TYPE` 定義の閉じ括弧に続いて、`[ NOT ] INSTANTIABLE` は、このオブジェクトタイプのオブジェクトインスタンスを作成できるかどうかを指定します。デフォルトの `INSTANTIABLE` は、このオブジェクトタイプのインスタンスを作成できることを意味します。このオブジェクトタイプを、他の特殊なサブタイプを定義する親「テンプレート」としてのみ使用する場合は、`NOT INSTANTIABLE` を指定します。`NOT INSTANTIABLE` を指定する場合、`NOT FINAL` も指定する必要があります。オブジェクトタイプのメソッドに `NOT INSTANTIABLE` 修飾子が含まれる場合、オブジェクトタイプ自体を `NOT INSTANTIABLE` と `NOT FINAL` で定義する必要があります。

注釈: `NOT INSTANTIABLE` の指定は `CREATE TYPE` コマンドで受け入れられますが、**SPL** は現在サブタイプの作成をサポートしていません。

`method_spec` は、メンバーメソッドまたは静的メソッドの仕様を示します。

メソッドを定義する前に、`[ NOT ] FINAL` はメソッドをサブタイプでオーバーライドできるかどうかを指定します。`NOT FINAL` はデフォルトで、サブタイプでメソッドをオーバーライドできます。

メソッドがスーパータイプの同じ名前のメソッドをオーバーライドする場合、メソッドの定義の前に `OVERRIDING` を指定します。オーバーライドするメソッドは、スーパータイプで定義されているのと同じデータ型とパラメーターモード、同じ順序、同じ戻り値型（メソッドの場合）を持つ同じ名前のメソッドパラメーターを同じ数持つ必要があります。

メソッドの定義の前に、`[ NOT ] INSTANTIABLE` はオブジェクトタイプ定義がメソッドの実装を提供するかどうかを指定します。`INSTANTIABLE` が指定されている場合、オブジェクトタイプの `CREATE TYPE BODY` コマンドでメソッドの実装を指定する必要があります。`NOT INSTANTIABLE` が指定されている場合、オブジェクトタイプの `CREATE TYPE BODY` コマンドにメソッドの実装を含めることはできません。後者の場合、サブタイプにはメソッドの実装が含まれ、このオブジェクトタイプのメソッドをオーバーライドすると想定されています。オブジェクトタイプに `NOT INSTANTIABLE` メソッドがある場合、オブジェクトタイプ定義自体は、オブジェクトタイプ仕様の閉じ括弧の後に `NOT INSTANTIABLE` と `NOT FINAL` を指定する必要があります。デフォルトは `INSTANTIABLE` です。

`subprogram_spec` はプロシージャまたは関数の指定を示し、`MEMBER` または `STATIC` の指定で始まります。メンバーサブプログラムは特定のオブジェクトインスタンスに関して呼び出される必要がありますが、静的サブプログラムはオブジェクトインスタンスに関して呼び出されません。

proc_name は手順の識別子です。SELF パラメータが指定されている場合、name は CREATE TYPE コマンドで指定されたオブジェクトタイプ名です。指定されている場合、parm1, parm2, ... は手順の正式なパラメータです。datatype1, datatype2, ... はそれぞれ parm1, parm2, ... のデータ型です。IN, IN OUT および OUT は、各仮パラメータの可能なパラメータモードです。何も指定されていない場合、デフォルトは「11」です。「10」は「11」パラメータに指定できるデフォルト値です。

CONSTRUCTOR キーワードと関数定義を含めて、コンストラクター関数を定義します。

func_name は関数の識別子です。SELF パラメータが指定されている場合、name は CREATE TYPE コマンドで指定されたオブジェクトタイプ名です。指定されている場合、parm1, parm2, ... は関数の正式なパラメーターです。datatype1, datatype2, ... はそれぞれ parm1, parm2, ... のデータ型です。IN, IN OUT および OUT は、各仮パラメータの可能なパラメータモードです。何も指定されていない場合、デフォルトは「11」です。「10」は「11」パラメータに指定できるデフォルト値です。「12」は、関数が返す値のデータ型です。

オブジェクトタイプの指定について、次の点に注意する必要があります。

- オブジェクトタイプには少なくとも 1 つの属性が定義されている必要があります。
- オブジェクトタイプに定義されているメソッドはないか、1 つまたは複数あります。
- 各メンバーメソッドには、SELF という名前の暗黙の組み込みパラメーターがあり、そのデータ型は定義されているオブジェクト型のデータ型です。

SELF は、現在メソッドを呼び出しているオブジェクトインスタンスを指します。SELF は、パラメーターリストで IN または IN OUT パラメーターとして明示的に宣言できます（たとえば、MEMBER FUNCTION (SELF IN OUT object_type ...))。

SELF が明示的に宣言されている場合、SELF はパラメーターリストの最初のパラメーターである必要があります。SELF が明示的に宣言されていない場合、そのパラメータモードのデフォルトは、メンバープロシージャの場合は IN OUT、メンバー関数の場合は IN です。

- 静的メソッドはオーバーライドできません（OVERRIDING と STATIC を method_spec で一緒に指定することはできません）。
- 静的メソッドはインスタンス化可能でなければなりません（method_spec で NOT INSTANTIABLE と STATIC を一緒に指定することはできません）。

15.2.2 Object Type Body Syntax

以下は、オブジェクトタイプ本体の構文です。

```
CREATE [ OR REPLACE ] TYPE BODY <name>
{ IS | AS }
<method_spec> [...]
[<constructor>] [...]
END;
```

method_spec は次のとおりです。

```
subprogram_spec
```

subprogram_spec は次のとおりです。

```
{ MEMBER | STATIC }
{ PROCEDURE <proc_name>
  [ ( [ SELF [ IN | IN OUT ] <name> ]
    [, parm1 [ IN | IN OUT | OUT ] datatype1
      [ DEFAULT value1 ] ]
    [, parm2 [ IN | IN OUT | OUT ] datatype2
      [ DEFAULT value2 ]
    ] ...)
  ]
}
{ IS | AS }
[ PRAGMA AUTONOMOUS_TRANSACTION; ]
[ <declarations> ]
BEGIN
  <statement>; ...
[ EXCEPTION
  WHEN ... THEN
    <statement>; ...]
END;
|
FUNCTION <func_name>
  [ ( [ SELF [ IN | IN OUT ] <name> ]
    [, parm1 [ IN | IN OUT | OUT ] datatype1
      [ DEFAULT value1 ] ]
    [, parm2 [ IN | IN OUT | OUT ] datatype2
      [ DEFAULT value2 ]
    ] ...)
  ]
  RETURN <return_type>
{ IS | AS }
[ PRAGMA AUTONOMOUS_TRANSACTION; ]
[ <declarations> ]
BEGIN
  <statement>; ...
[ EXCEPTION
  WHEN ... THEN
    <statement>; ...]
END;
```

ここで、constructor は次のとおりです。

```
CONSTRUCTOR <func_name>
  [ ( [ SELF [ IN | IN OUT ] <name> ]
    [, parm1 [ IN | IN OUT | OUT ] datatype1
```

(次のページに続く)

(前のページからの続き)

```

        [ DEFAULT value1 ] ]
    [, parm2 [ IN | IN OUT | OUT ] datatype2
        [ DEFAULT value2 ]
    ] ...)
]
RETURN self AS RESULT
{ IS | AS }
[ <declarations> ]
BEGIN
    <statement>; ...
[ EXCEPTION
    WHEN ... THEN
        <statement>; ...]
END;
```

name は、オブジェクトタイプに割り当てられた識別子（オプションでスキーマ修飾）です。

method_spec は、CREATE TYPE コマンドで指定されたインスタンス化可能なメソッドの実装を示します。

CREATE TYPE コマンドの method_spec で INSTANTIABLE を指定または省略した場合、CREATE TYPE BODY コマンドのこのメソッドには method_spec が必要です。

CREATE TYPE コマンドの method_spec で NOT INSTANTIABLE が指定されている場合、CREATE TYPE BODY コマンドのこのメソッドには method_spec が存在しない必要があります。

subprogram_spec はプロシージャまたは関数の仕様を示し、MEMBER または STATIC の仕様で始まります。CREATE TYPE コマンドの subprogram_spec で指定されたものと同じ修飾子を使用する必要があります。

proc_name は CREATE TYPE コマンドで指定されたプロシージャの識別子です。パラメータ宣言は CREATE TYPE コマンドで説明したのと同じ意味を持ち、CREATE TYPE コマンドで指定したのと同じ方法で CREATE TYPE BODY コマンドで指定する必要があります。

CONSTRUCTOR キーワードと関数定義を含めて、コンストラクター関数を定義します。

func_name は CREATE TYPE コマンドで指定された機能の識別子です。パラメータ宣言は CREATE TYPE コマンドで説明したのと同じ意味を持ち、CREATE TYPE コマンドで指定したのと同じ方法で CREATE TYPE BODY コマンドで指定する必要があります。return_type は関数が返す値のデータ型であり、CREATE TYPE コマンドで指定された return_type と一致する必要があります。

PRAGMA AUTONOMOUS_TRANSACTION は、プロシージャまたは関数を自律型トランザクションとして設定するディレクティブです。

declarations は、変数、カーソル、タイプ、またはサブプログラムの宣言です。サブプログラム宣言が含まれる場合、他のすべての変数、カーソル、および型宣言の後に宣言する必要があります。

statement は SPL プログラムのステートメントです。

15.3 Creating Object Types

CREATE TYPE コマンドを使用してオブジェクトタイプ仕様を作成し、CREATE TYPE BODY コマンドを使用してオブジェクトタイプ本体を作成できます。このセクションでは、CREATE TYPE および CREATE TYPE BODY コマンドを使用したいいくつかの例を示します。

最初の例では、属性のみを含みメソッドを含まない addr_object_type オブジェクトタイプを作成します。

```
CREATE OR REPLACE TYPE addr_object_type AS OBJECT
(
    street          VARCHAR2(30),
    city            VARCHAR2(20),
    state           CHAR(2),
    zip             NUMBER(5)
);
```

このオブジェクトタイプにはメソッドがないため、オブジェクトタイプの本体は必要ありません。この例では、複合型を作成します。これにより、関連するオブジェクトを単一の属性として扱うことができます。

15.3.1 Member Methods

メンバーメソッドは、オブジェクト型内で定義された関数またはプロシージャであり、その型のインスタンスを通じてのみ呼び出すことができます。メンバーメソッドは、操作しているオブジェクトインスタンスにアクセスし、その属性を変更できます。

次のオブジェクトタイプ仕様は、emp_obj_typ オブジェクトタイプを作成します。

```
CREATE OR REPLACE TYPE emp_obj_typ AS OBJECT
(
    empno          NUMBER(4),
    ename          VARCHAR2(20),
    addr           ADDR_OBJ_TYP,
    MEMBER PROCEDURE display_emp(SELF IN OUT emp_obj_typ)
);
```

オブジェクトタイプ emp_obj_typ には、display_emp という名前のメンバーメソッドが含まれています。display_emp は SELF パラメーターを使用し、メソッドが呼び出されるオブジェクトインスタンスを渡します。

SELF パラメータは、定義されているオブジェクトタイプのデータタイプであるパラメータです。SELF は常にメソッドを呼び出しているインスタンスを指します。SELF パラメーターは、パラメーターリストで明示的に宣言されているかどうかに関係なく、メンバープロシージャまたは関数の最初のパラメーターです。

次のコードスニペットは、emp_obj_typ: のオブジェクトタイプボディを定義しています

```
CREATE OR REPLACE TYPE BODY emp_obj_typ AS
  MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Employee No      : ' || empno);
    DBMS_OUTPUT.PUT_LINE('Name              : ' || ename);
    DBMS_OUTPUT.PUT_LINE('Street           : ' || addr.street);
    DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || addr.city || ', ' ||
      addr.state || ' ' || LPAD(addr.zip,5,'0'));
  END;
END;
```

オブジェクトタイプの本体で SELF パラメータを使用することもできます。CREATE TYPE BODY コマンドで SELF パラメータがどのように使用されるかを示すために、前述のオブジェクトタイプの本体は次のように記述できます。

```
CREATE OR REPLACE TYPE BODY emp_obj_typ AS
  MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Employee No      : ' || SELF.empno);
    DBMS_OUTPUT.PUT_LINE('Name              : ' || SELF.ename);
    DBMS_OUTPUT.PUT_LINE('Street           : ' || SELF.addr.street);
    DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || SELF.addr.city || ', ' ||
      SELF.addr.state || ' ' || LPAD(SELF.addr.zip,5,'0'));
  END;
END;
```

emp_obj_typ 本体の両方のバージョンは完全に同等です。

15.3.2 Static Methods

メンバーメソッドのように、静的メソッドは型に属します。ただし、静的メソッドは、型の*インスタンス*ではなく、型の*名前*を使用して呼び出されます。たとえば、emp_obj_type type 内で定義された get_count という名前の静的関数を呼び出すには、次のように記述できます。

```
emp_obj_type.get_count();
```

静的メソッドは、オブジェクトインスタンスへのアクセス権を持たず、オブジェクトインスタンスの属性を変更できず、通常、そのタイプのインスタンスでは機能しません。

次のオブジェクトタイプの仕様には、静的関数 get_dname とメンバープロシージャ display_dept: が含まれます。

```
CREATE OR REPLACE TYPE dept_obj_typ AS OBJECT (
    deptno          NUMBER(2),
    STATIC FUNCTION get_dname(p_deptno IN NUMBER) RETURN VARCHAR2,
    MEMBER PROCEDURE display_dept
);
```

dept_obj_typ のオブジェクト型本体は、get_dname という名前の静的関数と display_dept: という名前のメンバープロシージャを定義します

```
CREATE OR REPLACE TYPE BODY dept_obj_typ AS
    STATIC FUNCTION get_dname(p_deptno IN NUMBER) RETURN VARCHAR2
    IS
        v_dname      VARCHAR2(14);
    BEGIN
        CASE p_deptno
            WHEN 10 THEN v_dname := 'ACCOUNTING';
            WHEN 20 THEN v_dname := 'RESEARCH';
            WHEN 30 THEN v_dname := 'SALES';
            WHEN 40 THEN v_dname := 'OPERATIONS';
            ELSE v_dname := 'UNKNOWN';
        END CASE;
        RETURN v_dname;
    END;

    MEMBER PROCEDURE display_dept
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Dept No      : ' || SELF.deptno);
        DBMS_OUTPUT.PUT_LINE('Dept Name   : ' ||
            dept_obj_typ.get_dname(SELF.deptno));
    END;
END;
```

静的関数 get_dname 内では、SELF への参照は存在できません。静的関数はオブジェクトインスタンスとは無関係に呼び出されるため、オブジェクト属性には暗黙的にアクセスできません。

メンバープロシージャ display_dept は、SELF パラメータで渡されたオブジェクトインスタンスの deptno 属性にアクセスできます。display_dept パラメータリストで SELF パラメータを明示的に宣言する必要はありません。

display_dept プロシージャの最後の DBMS_OUTPUT.PUT_LINE ステートメントには、静的関数 get_dname (オブジェクトタイプ名 dept_obj_typ で修飾) の呼び出しが含まれます。

15.3.3 Constructor Methods

コンストラクターメソッドは、通常、オブジェクトのメンバーに値を割り当てることにより、オブジェクト型のインスタンスを作成する関数です。オブジェクトタイプは、さまざまなタスクを実行するために複数のコンストラクタを定義できます。コンストラクターメソッドは、名前が型の名前と一致するメンバー関数（SELF パラメーターで呼び出されます）です。

たとえば、address という名前のタイプを定義すると、各コンストラクターは address という名前になります。同じ名前でも異なる引数タイプを持つ 1 つ以上の異なるコンストラクター関数を作成することにより、コンストラクターをオーバーロードできます。

SPL コンパイラは、オブジェクトタイプごとにデフォルトのコンストラクタを提供します。既定のコンストラクターは、名前が型の名前と一致し、引数リストが型メンバーに（順番に）一致するメンバー関数です。たとえば、次のようなオブジェクトタイプがあるとします。

```
CREATE TYPE address AS OBJECT
(
  street_address VARCHAR2(40),
  postal_code     VARCHAR2(10),
  city            VARCHAR2(40),
  state           VARCHAR2(2)
)
```

SPL コンパイラーは、次のシグネチャを持つデフォルトコンストラクターを提供します。

```
CONSTRUCTOR FUNCTION address
(
  street_address VARCHAR2(40),
  postal_code     VARCHAR2(10),
  city            VARCHAR2(40),
  state           VARCHAR2(2)
)
```

デフォルトのコンストラクターの本体は、各メンバーを NULL に設定するだけです。

カスタムコンストラクターを作成するには、CREATE TYPE コマンドで（キーワードコンストラクターを使用して）コンストラクター関数を宣言し、CREATE TYPE BODY コマンドで構築関数を定義します。たとえば、street_address と postal_code: が与えられた都市と州を計算する address タイプのカスタムコンストラクターを作成できます。

```
CREATE TYPE address AS OBJECT
(
  street_address VARCHAR2(40),
  postal_code     VARCHAR2(10),
  city            VARCHAR2(40),
  state           VARCHAR2(2),
```

(次のページに続く)

(前のページからの続き)

```

CONSTRUCTOR FUNCTION address
(
    street_address VARCHAR2,
    postal_code VARCHAR2
) RETURN self AS RESULT
)
CREATE TYPE BODY address AS
CONSTRUCTOR FUNCTION address
(
    street_address VARCHAR2,
    postal_code VARCHAR2
) RETURN self AS RESULT
IS
BEGIN
    self.street_address := street_address;
    self.postal_code := postal_code;
    self.city := postal_code_to_city(postal_code);
    self.state := postal_code_to_state(postal_code);
    RETURN;
END;
END;

```

オブジェクト型のインスタンスを作成するには、その型のコンストラクターメソッドの 1 つを呼び出します。例:

```

DECLARE
    cust_addr address := address('100 Main Street', 02203);
BEGIN
    DBMS_OUTPUT.PUT_LINE(cust_addr.city); -- displays Boston
    DBMS_OUTPUT.PUT_LINE(cust_addr.state); -- displays MA
END;

```

通常、カスタムコンストラクター関数は、不完全な情報が与えられたときにメンバー値を計算するために使用されます。上記の例では、郵便番号が与えられたときに city と state の値を計算します。

カスタムコンストラクター関数は、オブジェクトの状態を制限するビジネスルールを適用するためにも使用されます。たとえば、payment を表すオブジェクトタイプを定義する場合、カスタムコンストラクターを使用して、NULL、負、またはゼロの amount で payment タイプのオブジェクトを作成できないようにすることができます。デフォルトのコンストラクターは payment.amount を NULL に設定するため、カスタムコンストラクター（署名がデフォルトコンストラクターと一致する）を作成して NULL の量を禁止する必要があります。

15.4 Creating Object Instances

オブジェクト型のインスタンスを作成するには、まずオブジェクト型の変数を宣言してから、宣言されたオブジェクト変数を初期化する必要があります。オブジェクト変数を宣言するための構文は次のとおりです。

```
<object obj_type>
```

`object` は、オブジェクト変数に割り当てられた識別子です。

`obj_type` は、以前に定義されたオブジェクトタイプの識別子です。

オブジェクト変数を宣言した後、*constructor**method* を呼び出して、値でオブジェクトを初期化する必要があります。次の構文を使用して、コンストラクターメソッドを呼び出します。

```
[NEW] <obj_type> ({expr1 | NULL} [, {expr2 | NULL} ] [, ...])
```

`obj_type` は、オブジェクトタイプのコンストラクターメソッドの識別子です。コンストラクターメソッドの名前は、以前に宣言されたオブジェクトタイプと同じです。

`expr1`, `expr2`, ... は、オブジェクトタイプの最初の属性、オブジェクトタイプの 2 番目の属性などと型互換性がある式です。属性がオブジェクトタイプの場合、対応する式は `NULL`、オブジェクト初期化式、またはそのオブジェクトタイプを返す式。

次の匿名ブロックは、変数を宣言して初期化します。

```
DECLARE
    v_emp          EMP_OBJ_TYP;
BEGIN
    v_emp := emp_obj_typ (9001, 'JONES',
        addr_obj_typ('123 MAIN STREET', 'EDISON', 'NJ', 08817));
END;
```

変数 (`v_emp`) は、`EMP_OBJ_TYPE` という名前の以前に定義されたオブジェクトタイプで宣言されます。ブロックの本体は、`emp_obj_typ` および `addr_obj_type` コンストラクターを使用して変数を初期化します。

ブロックの本体にオブジェクトの新しいインスタンスを作成するときに、`NEW` キーワードを含めることができます。`NEW` キーワードは、指定された引数に一致するシグネチャを持つオブジェクトコンストラクターを呼び出します。

次の例では、`mgr` と `emp` という 2 つの変数を宣言しています。変数は両方とも `EMP_OBJ_TYPE` です。`mgr` オブジェクトは宣言で初期化され、`emp` オブジェクトは宣言で `NULL` に初期化され、本文に値が割り当てられます。

```
DECLARE
    mgr EMP_OBJ_TYPE := (9002, 'SMITH');
    emp EMP_OBJ_TYPE;
BEGIN
```

(次のページに続く)

(前のページからの続き)

```
emp := NEW EMP_OBJ_TYPE (9003, 'RAY');
END;
```

注釈: AdvancedServer では、コンストラクターメソッドの代わりに次の代替構文を使用できます。

```
[ ROW ] ( { expr1 | NULL } [, { expr2 | NULL } ] [, ...])
```

ROW は、括弧で囲まれたコンマ区切りリスト内で 2 つ以上の用語が指定されている場合のオプションのキーワードです。1 つの用語のみが指定されている場合、ROW キーワードの指定は必須です。

15.5 Referencing an Object

オブジェクト変数が作成および初期化されると、フォームのドット表記を使用して個々の属性を参照できます。

```
<object.attribute>
```

object は、オブジェクト変数に割り当てられた識別子です。attribute は、オブジェクトタイプ属性の識別子です。

attribute 自体がオブジェクト型の場合、参照は次の形式をとる必要があります。

```
<object.attribute.attribute_inner>
```

attribute_inner は、object の定義で attribute が参照するオブジェクトタイプに属する識別子です。

次の例は、前の匿名ブロックを拡張して、emp_obj_typ オブジェクトに割り当てられた値を表示します。

```
DECLARE
    v_emp      EMP_OBJ_TYP;
BEGIN
    v_emp := emp_obj_typ(9001, 'JONES',
        addr_obj_typ('123 MAIN STREET', 'EDISON', 'NJ', 08817));
    DBMS_OUTPUT.PUT_LINE('Employee No   : ' || v_emp.empno);
    DBMS_OUTPUT.PUT_LINE('Name          : ' || v_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Street       : ' || v_emp.addr.street);
    DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || v_emp.addr.city || ', ' ||
        v_emp.addr.state || ' ' || LPAD(v_emp.addr.zip, 5, '0'));
END;
```

以下は、この匿名ブロックからの出力です。

```
Employee No   : 9001
Name          : JONES
Street        : 123 MAIN STREET
City/State/Zip: EDISON, NJ 08817
```

メソッドは、属性と同様の方法で呼び出されます。

オブジェクト変数が作成および初期化されると、次の形式のドット表記を使用してメンバープロシージャまたは関数が呼び出されます。

```
<object.prog_name>
```

`object` は、オブジェクト変数に割り当てられた識別子です。 `prog_name` は、プロシージャまたは関数の識別子です。

静的プロシージャまたは関数は、オブジェクト変数を使用して呼び出されません。代わりに、オブジェクトタイプ名を使用してプロシージャまたは関数が呼び出されます。

```
<object_type.prog_name>
```

`object_type` は、オブジェクトタイプに割り当てられた識別子です。 `prog_name` は、プロシージャまたは関数の識別子です。

前の匿名ブロックの結果は、メンバープロシージャ `display_emp:` を呼び出すことで複製できます。

```
DECLARE
    v_emp          EMP_OBJ_TYP;
BEGIN
    v_emp := emp_obj_typ(9001, 'JONES',
        addr_obj_typ('123 MAIN STREET', 'EDISON', 'NJ', 08817));
    v_emp.display_emp;
END;
```

以下は、この匿名ブロックからの出力です。

```
Employee No   : 9001
Name          : JONES
Street        : 123 MAIN STREET
City/State/Zip: EDISON, NJ 08817
```

次の匿名ブロックは `dept_obj_typ` のインスタンスを作成し、メンバープロシージャ `display_dept:` を呼び出します

```
DECLARE
    v_dept          DEPT_OBJ_TYP := dept_obj_typ (20);
BEGIN
```

(次のページに続く)

(前のページからの続き)

```
v_dept.display_dept;
END;
```

以下は、この匿名ブロックからの出力です。

```
Dept No      : 20
Dept Name    : RESEARCH
```

dept_obj_typ で定義された静的関数は、次のようにオブジェクトタイプ名で修飾することで直接呼び出すことができます。

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(dept_obj_typ.get_dname(20));
END;

RESEARCH
```

15.6 Dropping an Object Type

オブジェクトタイプを削除するための構文は次のとおりです。

```
DROP TYPE <objtype>;
```

objtype は、ドロップするオブジェクトタイプの識別子です。objtype の定義にそれ自体がオブジェクトタイプまたはコレクションタイプである属性が含まれる場合、これらのネストされたオブジェクトタイプまたはコレクションタイプは最後にドロップする必要があります。

オブジェクトタイプのボディがオブジェクトタイプに定義されている場合、DROP TYPE コマンドは、オブジェクトタイプの仕様とオブジェクトタイプのボディを削除します。完全なオブジェクトタイプを再作成するには、CREATE TYPE コマンドと CREATE TYPE BODY コマンドの両方を再発行する必要があります。

次の例では、この章で前述した emp_obj_typ および addr_obj_typ オブジェクトタイプを削除します。emp_obj_typ は、定義として addr_obj_typ が属性として含まれているため、最初に削除する必要があります。

```
DROP TYPE emp_obj_typ;
DROP TYPE addr_obj_typ;
```

オブジェクト型の仕様を削除せずに、オブジェクト型の本体を削除する構文は次のとおりです。

```
DROP TYPE BODY <objtype>;
```

オブジェクトタイプの本体は、CREATE TYPE BODY コマンドを発行して再作成できます。

次の例では、dept_obj_typ. のオブジェクトタイプ本体のみを削除します。

```
DROP TYPE BODY dept_obj_typ;
```

第 16 章

Conclusion

EDBPostgres™AdvancedServer データベース互換性ストアードプロシージャ言語ガイド

著作権 ©2007-2020EnterpriseDBCorporation。

無断複写・転載を禁じます。

EnterpriseDB®Corporation

34CrosbyDrive、Suite201、Bedford、MA01730、アメリカ

T+17813573390F+19784671307E

info@enterprisedb.com

www.enterprisedb.com

- EnterpriseDB および PostgresEnterpriseManager は、EnterpriseDBCorporation の登録商標です。EDB および EDBPostgres は、EnterpriseDBCorporation の商標です。Oracle は、Oracle、Inc の登録商標です。その他の商標は、それぞれの所有者の商標です。
- EDB は、存在する場合、EDB 製品のログオン UI の設計、コーディングのベストプラクティスの確立、レビュー、および入力検証の検証を行います。EDB は追加の入力コンポーネントに対して同じアプローチに従いますが、製品の性質上、限定された検証が可能な信頼できるユーザーによって入力および送信されるフリーフォーム SQL、WMI またはその他の文字列を受け入れる必要がある場合があります。このような場合、ユーザーが誤った入力や危険な入力を禁止することはできません。
- EDB は、将来、認証された信頼できるユーザーからのフリーフォーム SQL、WMI またはその他の潜在的に危険な入力を受け入れる製品に機能を追加する権利を留保しますが、そのような機能はすべて、可能な限り最小のリスクを提供するように設計およびテストされます可能であれば、スーパーユーザーまたは同等の特権が必要です。
- EDB は、すべての潜在的な脅威を予測できることを保証しないため、すべての潜在的な脆弱性が対処または検討されたことを完全に保証することはできません。

索引

- Accessing Subprogram Variables, 52
- After Row-Level Trigger, 211
- After Statement-Level Trigger, 209
- Anonymous Blocks, 9
- Assignment, 79
- Associative Arrays, 162
- Attributes, 224
- Basic Object Concepts, 223
- Basic SPL Elements, 1
- Basic Statements, 79
- Before Row-Level Trigger, 211
- Before Statement-Level Trigger, 209
- Block Relationships, 37
- Calling a Function, 22
- Calling a Procedure, 15
- CASE Expression, 100
- Case Sensitivity, 2
- CASE Statement, 103
- Character Set, 1
- Closing a Cursor, 142
- Closing a Cursor Variable, 150
- Collection Methods, 175
- Collections, 161
- COMMIT, 120
- Compilation Errors in Procedures and Functions, 62
- Compound Triggers, 206, 215
- Conclusion, 241
- Constants, 3
- Constructor Methods, 234
- CONTINUE, 108
- Control Structures, 91
- COUNT, 175
- Creating a Function, 18
- Creating a Procedure, 11
- Creating a Subfunction, 35
- Creating a Subprocedure, 33
- Creating Object Instances, 236
- Creating Object Types, 231
- Creating Triggers, 201
- Cursor Attributes, 143
- Cursor FOR Loop, 143
- Database Object Name Resolution, 65
- Database Object Privileges, 66
- Declaring a Cursor, 139
- Declaring a Cursor Variable, 147
- Declaring a SYS_REFCURSOR Cursor Variable, 148
- Declaring a User Defined REF CURSOR Type Variable, 148
- Declaring a Variable, 75
- Definer's vs. Invokers Rights, 66
- DELETE, 80, 176
- Deleting a Function, 23
- Deleting a Procedure, 16
- Dropping an Object Type, 239
- Dynamic Queries With REF CURSORS, 157
- Dynamic SQL, 135
- Errors and Messages, 198
- Examples, 152
- Exception Handling, 111
- EXECUTE IMMEDIATE BULK COLLECT, 194
- EXECUTE Privilege, 64
- EXISTS, 177
- EXIT, 107
- EXTEND, 178
- FETCH BULK COLLECT, 192
- Fetching Rows From a Cursor, 140
- Fetching Rows From a Cursor Variable, 149
- FIRST, 180
- FOR (*integer variant*), 109
- Functions Overview, 18
- GOTO Statement, 97
- Identifiers, 2
- IF Statement, 91
- IF-THEN, 91
- IF-THEN-ELSE, 92
- IF-THEN-ELSE IF, 93
- IF-THEN-ELSIF-ELSE, 95
- INSERT, 81
- INSTEAD OF Trigger, 214
- Invoking Subprograms, 39
- LAST, 181
- LIMIT, 181
- LOOP, 107
- Loops, 107
- Member Methods, 231
- Methods, 224
- Modularizing Cursor Operations, 153
- Nested Tables, 166
- NEXT, 182
- NULL, 83
- Object Type Body Syntax, 228
- Object Type Components, 225
- Object Type Specification Syntax, 225
- Object Types and Objects, 223
- Obtaining the Result Status, 90
- Opening a Cursor, 140
- Opening a Cursor Variable, 148
- Overloading Methods, 224
- Overloading Subprograms, 48
- Overview, 199
- Packages, 221

- Parameter Modes, 28
- Parameterized Cursors, 145
- Positional vs. Named Parameter Notation, 26
- PRAGMA AUTONOMOUS_TRANSACTION, 125
- PRAGMA EXCEPTION_INIT, 115
- PRIOR, 182
- Procedure and Function Parameters, 24
- Procedures Overview, 11
- Program Security, 64

- Qualifiers, 2

- RAISE_APPLICATION_ERROR, 117
- REF CURSOR Overview, 147
- REF CURSORs and Cursor Variables, 147
- Referencing an Object, 237
- RETURN Statement, 97
- Returning a REF CURSOR From a Function, 152
- RETURNING BULK COLLECT, 194
- ROLLBACK, 121

- Searched CASE Expression, 101
- Searched CASE Statement, 105
- Security Example, 67
- SELECT BULK COLLECT, 190
- SELECT INTO, 86
- Selector CASE Expression, 100
- Selector CASE Statement, 103
- SPL Block Structure, 7
- SPL Programs, 7
- Static Cursors, 139
- Static Methods, 232
- Subprograms - Subprocedures and Subfunctions, 31
- Summary of Cursor States and Attributes, 143

- TABLE (), 185
- Transaction Control, 119
- Transactions and Exceptions, 206
- Trigger Examples, 209
- Trigger Variables, 205
- Triggers, 199
- TRIM, 183
- Types of Triggers, 200

- UPDATE, 88
- Usage Restrictions, 151
- User-defined Exceptions, 113
- User-Defined PL/SQL Subtypes, 4
- User-Defined Record Types and Record Variables, 76
- Using Default Values in Parameters, 29
- Using Forward Declarations, 47
- Using the BULK COLLECT Clause, 190
- Using the FORALL Statement, 188
- Using the MULTISSET UNION Operator, 186
- Using the RETURNING INTO Clause, 83

- Variable Declarations, 75
- Varrays, 171

- WHILE, 109
- Working with Collections, 185