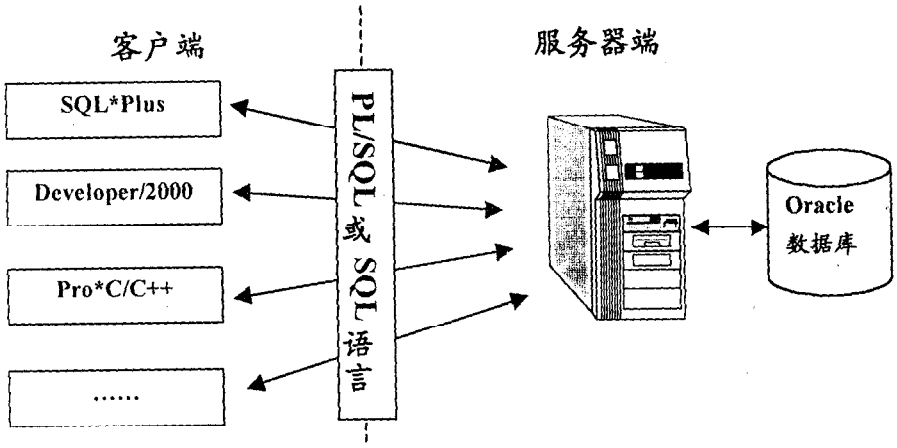
第一章 PL/SQL 简介

PL/SQL（Procedural Language/SQL）是在标准 SQL 的基础上增加了过程化处理的语言，是 Oracle 对 SQL 的扩充。与标准 SQL 语言相同，PL/SQL 也是 Oracle 客户端工具（如 SQL*Plus、Developer/2000 等）访问服务器的操作语言，如下图所示：



1.1 PL/SQL 概述

1.1.1 SQL 与 PL/SQL

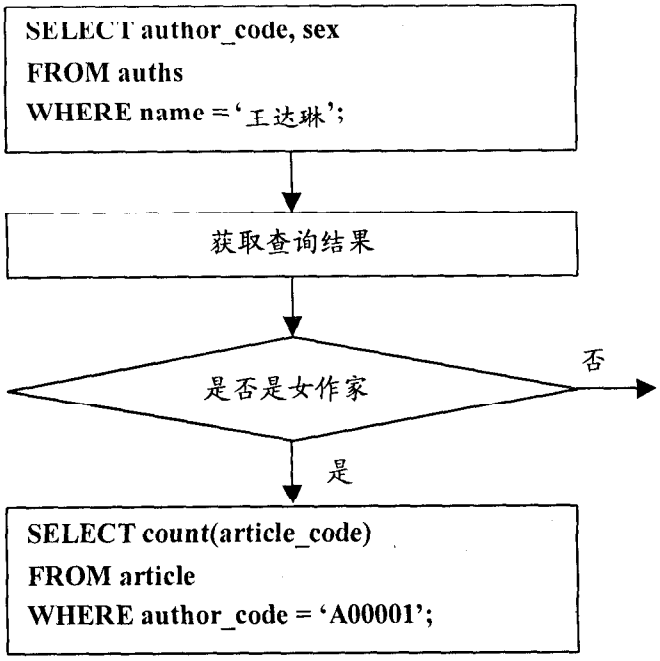
SQL (Structured Query Language) 是一种对关系数据库管理系统（RDBMS）进行操作的标准结构化语言，它用来设置、使用和维护关系数据库。例如，下面是一条查询表中数据的 SQL 语句：

```
SELECT author_code,sex
FROM auths
WHERE name = '王达琳';
```

当用户执行了上面的查询语句后，服务器会给出如下的查询结果：

```
AUTHOR        SEX
------ ----------
A00001         0
```

但是，SQL 语言本身不支持对结果的进一步处理。例如，如果希望进一步对查询结果进行判断，当该作家为女作家时查询该作家发表文章的数量。SQL 语言本身是做不到这一点的，而要做到这一点，需要下述的处理过程：



PL/SQL 正是一种增加了过程化概念的 SQL 语言，它有以下一些标准 SQL 所没有的特征：

- 变量（包括预先定义的和自定义的）
- 控制结构（如 IF-THEN-EISE 等流控制语句）
- 自定义的存储过程和函数
- 对象类型

由于 PL/SQL 融合了 SQL 语言的灵活性和过程化的概念，使得 PL/SQL 成为了一种功能强大的结构化语言，可以设计复杂的应用。例如，下面的 PL/SQL 程序用来实现上面标准 SQL 所不能实现的功能：

```
DECLARE
  /*定义SQL语句中使用的变量 */
  v_Sex NUMBER ;
  v_AuthorCode CHAR(6);
  v_Arcount NUMBER;
BEGIN
  /* 查询表中数据*/
  SELECT author_code,sex
    INTO v_AuthorCode, v_Sex
    FROM auths
    WHERE name = '干达琳';

  /* 检查该作家是否为女作家。如果是女作家，那么查询其发表文章的数量*/
  IF v_Sex = 0  THEN
    SELECT count(article_code)
      INTO v_Arcount
      FROM article
      WHERE author_code = 'A00001';
  END IF;
END;
```

在上面的PL/SQL程序中，定义了三个变量（v_Sex、v_AuthorCode和v_Arcount）、两条SQL语句（UPDATE、INSERT）和一条IF 语句。

1.1.2 PL/SQL 的特性

块结构、变量与常量、循环结构以及游标是 PL/SQL 最重要的特性。本节将依次介绍这些特性。

1.1.2.1 块结构

块（block）是 PL/SQL 程序的基本执行单元，所有的 PL/SQL 程序都是由块组成的。每个块完成程序中的部分工作，这样就可将程序分成多个快。一个标准的块结构如下：

```
DECLARE
    /*块的定义部分，这里可以定义 PL/SQL 变量、自定义类型、游标和局部子程序。
```

```
    这部分是块中可选部分。*/
BEGIN
    /*块的执行部分，这里放置一些可执行的 SQL 或 PL/SQL 语句。
      这是块中最重要的部分，并且也是块中必须有的部分。
      这里必须至少包含一条可执行语句。*/
EXCEPTION
    /* 异常处理部分，这里放置对错误进行处理的语句。这部分是块中可选部分。*/
END;
```

其中，异常处理部分是用来报告和处理程序运行时遇到的错误。Oracle 的这种将错误处理代码从程序的主体中分离出来的结构，使得程序的结构显得非常清晰。

1.1.2.2 变量与常量

PL/SQL 和数据库之间是通过变量来进行数据交换的。变量是一个存储空间，它是在程序中取出数据或传入数据给程序。

变量是在定义部分声明的。每个变量都有一个类型，这个类型决定变量应该存储什么样的数据。例如，下面是一些 PL/SQL 变量的声明：

```
DECLARE
  v_Name varchar2(10);   --声明为字符型的变量。
  v_CurrentDate DATE;   --声明为日期型的变量。
  v_Salary NUMBER(8,2);   --声明为数值型的变量。
  v_LoopCounter BINARY_INTEGER;   --声明为整数型的变量。
  v_CurrentlyRegistered BOOLEAN;   --声明为布尔型的变量。
```

上面变量对应的类型是 Oracle 预定义的类型，PL/SQL 也支持用户自定义类型（例如，PL/SQL 表和记录）。自定义类型是指用户定义程序中要进行处理的数据的结构，如下面的例子：

```
DECLARE
  TYPE Au_Record IS RECORD(    -- 创建一个记录类型。
  Author_code char(6),
  Name varchar2(10),
  sex NUMCBER(1)
```

```
);
v_Author Au_Record ;  --声明一个 Au_Record 记录类型的变量。
```

PL/SQL8 以上版本支持对象类型，它可以存储在数据库表中。例如，下面的例子用来创建一个对象类型：

```
CREATE OR REPLACE TYPE AuthorObj AS OBJECT (
  --创建一个对象类型
  Author_code   char(6),
  Name          VARCHAR2(10),
  Sex           NUMBER(1),
  birthdate     DATE,
  address       VARCHAR2(30)
);
```

常量的定义与变量的定义类似，只是在定义时必须增加一个关键字CONSTANT，并且同时给它一个值，这以后就不能再给常量进行赋值了，如下面的定义：

```
DECLARE
    sex_male CONSTANT INT := 1;
    sex_female CONSTANT INT := 0;
```

1.1.2.3 循环结构

PL/SQL 支持多种类型的循环结构。一个循环结构用来重复地执行块中的一些语句。

例如，有下面一张数据库表：

```
create table Table_A(
  num_col     NUMBER,
  char_col    VARCHAR2(60));
```

下面通过一个简单的循环将数字 1 到 100 插入到 Table_A 表中：

```
DECLARE
  v_LoopCounter BINARY_INTEGER := 1;
```

```
BEGIN
  LOOP
    INSERT INTO Table_A(num_col)
      VALUES (v_LoopCounter);
    v_LoopCounter := v_LoopCounter + 1;
    EXIT WHEN v_LoopCounter > 100;
  END LOOP;
END;
```

还可以使用其它类型的循环结构，例如 FOR 结构。下面的例子是用 FOR 循环实现与上例相同的功能：

```
DECLARE
  v_LoopCounter BINARY_INTEGER := 1;
BEGIN
  FOR v_LoopCounter IN 1..50 LOOP
    INSERT INTO Table_A(num_col)
      VALUES (v_LoopCounter);
  END LOOP;
END;
```

1.1.2.4 游标

游标用来查询数据库中的数据（例如 SELECT 语句返回的记录），并对查询的结果进行处理。游标分为显式游标和隐式游标两种。通过游标，可以对查询结果中的数据一条一条进行处理。例如，在下面的例子中，通过游标查询数据库表 AUTHS 中所有作家的姓名和工资，并将每条记录的值依次传给变量 v_Name 和 v_Salary：

```
DECLARE
  v_Name VARCHAR2(10);
  v_Salary  NUMBER(8,2);

  -- 游标的定义。
  CURSOR c_Auths IS
    SELECT name, salary
      FROM auths;
```

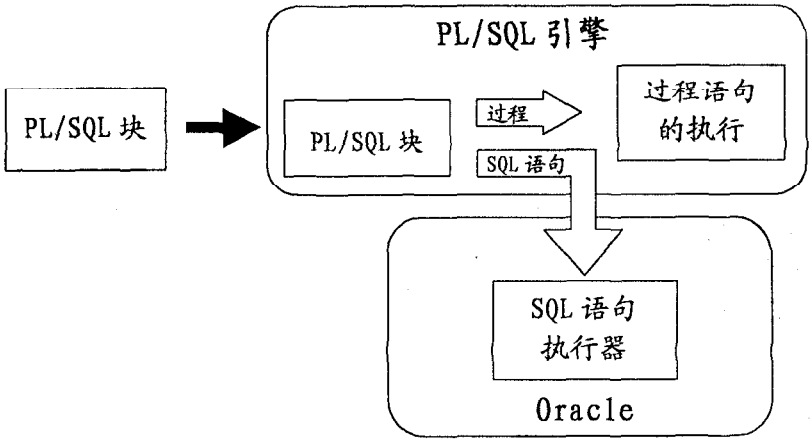
```
BEGIN
  OPEN c_Auths;  --打开游标。

  LOOP
    -- 检索一条记录
    FETCH c_Auths INTO v_Name, v_Salary;
    -- 当所有的记录都被检索出后退出循环。
    EXIT WHEN c_Auths%NOTFOUND;
    /* 对检索出的数据进行处理。*/
  END LOOP;

  -- 结束处理
  CLOSE c_Auths;
END;
```

1.2 PL/SQL 的运行

PL/SQL程序是通过一个引擎来执行的。这个引擎安装在Oracle的数据库服务器或一些客户端的应用开发工具（如Oracle Forms或Oracle Reports）中。PL/SQL程序可以在下面两个环境下运行：

- Oracle 服务器
- Oracle 的一些应用开发工具

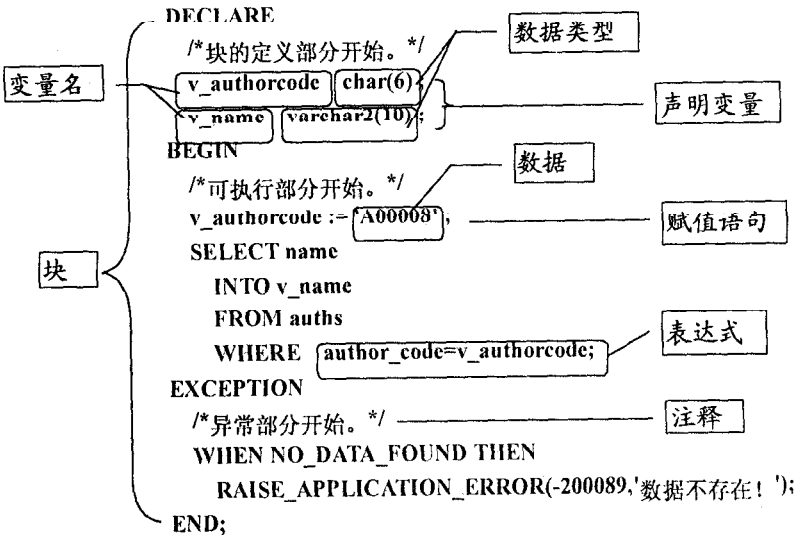
这两个环境是独立的。PL/SQL程序可以是在Oracle服务器上有效，但在应用开发工具中无效，也可以在两个环境中都有效。在任意一个环境中，PL/SQL引擎都可接收有效的PL/SQL块或子程序。这个引擎执行PL/SQL程序中过程性的语句，而将SQL语句送给服务器端的SQL语句执行器运行，如下图所示：



如果客户端的工具中有 **PL/SQL** 引擎，则在执行 **PL/SQL** 块时，该引擎仅执行处理块中的过程性语句，而将 **SQL** 语句送给服务器由 **SQL** 语句执行器运行；如果客户端的工具中没有 **PL/SQL** 引擎，则客户工具将 **PL/SQL** 块传给服务器端的 **PL/SQL** 引擎，在该引擎中处理块中的过程性语句，而将块中的 **SQL** 语句送给 **SQL** 语句执行器运行。

第二章 PL/SQL 的组成元素

同其它高级语言程序相同，**PL/SQL** 程序也是由一些基本元素组成的。例如，下面是一段简单的 **PL/SQL** 程序：



上面的程序用于从表 **auths** 中检索代码为‘**A00008**’的作家姓名。如果表 **auths** 中存在该作家，则将该作家的名字赋给变量 **v_name**，否则，显示错误信息。在上面的程序中，包含了如下的组成元素：

- 块
- 变量名
- 数据类型
- 数据
- 变量声明
- 赋值语句
- 表达式
- 注释

下面分别介绍。

2.1 块

块是组成PL/SQL程序的基本单元，一个块可以嵌套多个子块。一般地，一个块用来解决一个问题或一个子问题。下面介绍块的结构和分类。

2.1.1 块的基本结构

标准的块由如下三个独立的部分组成：

- 定义部分：声明块中所用到的变量、游标和类型，也可以声明局部的存储过程和函数。注意，这些过程和函数只在定义它的块中有效。
- 执行部分：指定块所要完成的工作，由 SQL 语句和过程性语句构成。
- 异常处理部分：用于处理程序中的错误，当错误发生时执行其中的程序代码。在 PL/SQL 中，警告或错误信息被称为异常。

其中，可执行部分是必须有的，定义部分和异常处理部分是可有可无的。块的三部分是用不同的关键字来分隔的。下面介绍块的各种结构：

1. 当块的三部分都存在时，块的结构如下所示：

```
DECLARE
  定义部分
BEGIN
  可执行部分，在块中必不可少
EXCEPTION
  异常部分
END;
```

2. 当只有可执行部分的块时，结构如下所示：

```
BEGIN
  可执行部分
END;
```

3. 当块只有定义部分和可执行部分（没有异常部分）时，其结构如下所示 ：

```
DECLARE
  定义部分
BEGIN
  可执行部分
END;
```

4. 当块中只有可执行部分和异常处理部分（没有定义部分）时，其结构如下所示：

```
BEGIN
  可执行部分
EXCEPTION
  异常处理部分
END;
```

2.1.2 块的分类

块可以分为以下四类：

- 无名块：动态构造并只能执行一次。
- 命名块：前面加了标号的无名块。
- 子程序：包括存储在数据库中的存储过程、函数和包等。这些块一旦被定义后，便可随时调用。
- 触发器：它是存储在数据库中的块。这些块一旦被构造后，就可以多次执行。当触发它的事件发生时调用该触发器。触发事件是指对表中数据的操作，如插入、删除和修改。

例 1，下面是一个无名块的例子：

```
SET SERVEROUTPUT ON SIZE 10000   -- 设置存储缓存区的大小。
DECLARE
  /*声明在块中要用到的变量。*/
  v_TypeCode1 VARCHAR2(6):='CC';
  v_TypeCode2 VARCHAR2(6):='DD';
  v_TypeRemark1 VARCHAR2(15):='Computer';
  v_TypeRemark2 VARCHAR2(15):='C++ Language';
  v_OutPut VARCHAR2(15);
BEGIN
```

```
  /*利用变量的值向表中插入两行。*/
  INSERT INTO type
    VALUES(v_TypeCode1,v_TypeRemark1);
  INSERT INTO type
    VALUES(v_TypeCode2,v_TypeRemark2);

  /*查询刚插入表中的两行，并使用 DBMS_OUTPUT 包输出结果。*/
  SELECT type_remark
    INTO v_OutPut
    FROM type
    WHERE type_code=v_TypeCode1;
  DBMS_OUTPUT.PUT_LINE(v_OutPut); /*将结果显示在屏幕上。*/

  SELECT type_remark
    INTO v_OutPut
    FROM type
    WHERE type_code=v_TypeCode2;
  DBMS_OUTPUT.PUT_LINE(v_OutPut);
END inserttype;
```

上面的块用来向 type 表插入两行记录，然后查询这两行，并将结果显示在屏幕上。

例 2，下面是一个命名块的例子：

```
<<inserttype>>
DECLARE
  /*声明在块中要用到的变量*/
  v_TypeCode1 VARCHAR2(6):='CC';
  v_TypeCode2 VARCHAR2(6):='DD';
  v_TypeRemark1 VARCHAR2(15):='Computer';
  v_TypeRemark2 VARCHAR2(15):='C++ Language';
  v_OutPut VARCHAR2(15);
BEGIN
  /*利用变量的值向表中插入两行*/
  INSERT INTO type
    VALUES(v_TypeCode1,v_TypeRemark1);
  INSERT INTO type
```

```
    VALUES(v_TypeCode2,v_TypeRemark2);
  /*查询刚插入表中的两行，并使用 DBMS_OUTPUT 包输出结果。*/

  SELECT type_remark
    INTO v_OutPut
    FROM type
    WHERE type_code=v_TypeCode1;
  DBMS_OUTPUT.PUT_LINE(v_OutPut); /*将结果显示在屏幕上。*/

  SELECT type_remark
    INTO v_OutPut
    FROM type
    WHERE type_code=v_TypeCode2;
  DBMS_OUTPUT.PUT_LINE(v_OutPut);
END inserttype;
```

在例 1 中无名块的 DECLARE 关键字前加一个标号后，就变成了有名块。在 END 关键字后也可以加标号名，表示该命名块的结束。块前的标号必须用符号“<<”和“>>”括起来。块末的标号可有可无，也无须用符号“<<”和“>>”括起来。

例 3，下面是一个定义子程序的例子：

```
--创建存储过程 inserttype
CREATE OR REPLACE PROCEDURE inserttype AS
  /*声明在块中要用到的变量。*/
  v_TypeCode1          VARCHAR2(6):='CC';
  v_TypeCode2          VARCHAR2(6):='DD';
  v_TypeRemark1        VARCHAR2(15):='Computer';
  v_TypeRemark2        VARCHAR2(15):='C++ Language';
  v_OutPut             VARCHAR2(15);

BEGIN
  /*利用变量的值向表中插入两行。*/
  INSERT INTO type
    VALUES(v_TypeCode1,v_TypeRemark1);
  INSERT INTO type
    VALUES(v_TypeCode2,v_TypeRemark2);
```

```
  /*查询刚插入表中的两行，并使用 DBMS_OUTPUT 包将结果显示在屏幕上。*/
  SELECT type_remark
    INTO v_OutPut
    FROM type
    WHERE type_code=v_TypeCode1;
  DBMS_OUTPUT.PUT_LINE(v_OutPut);

  SELECT type_remark
    INTO v_OutPut
    FROM type
    WHERE type_code=v_TypeCode2;
  DBMS_OUTPUT.PUT_LINE(v_OutPut);
END inserttype;
```

当将例 1 中块的关键字 DECLARE 改变成关键字 CREATE OR REPLACE PROCEDURE 时，该无名块就变成了存储过程。注意，在 END 关键字后必须加存储过程名。

例 4，下面是一个触发器的例子：

```
--创建触发器 salary_trigger
CREATE OR REPLACE TRIGGER salary_trigger
  --当向 auths 表中插入一行或修改一行后激活触发器。
  BEFORE INSERT OR UPDATE of salary
  ON auths
  FOR EACH ROW
BEGIN
  --如果插入值或修改值大于 10000 时，报告应插入的工资值，
  --并产生异常，退出该触发器。
  IF :new.salary >10000 THEN
     RAISE_APPLICATION_ERROR(-200060,'插入的工资应小于 10000！ ');
  END IF;
END salary_trigger;
```

上面的触发器被创建在 auths 表上，该触发器确保只有小于 10000 的值才能插入到 salary 列中。当向表中插入或修改一条记录时，触发该触发器。

2.2 变量名

变量用于传递 PL/SQL 块中的信息，其名称必须是合法的标识符。下面首先介绍标识符，然后介绍如何定义变量名。

2.2.1 标识符

标识符用来命名 PL/SQL 对象，例如变量、光标、子程序等。标识符的书写规则如下：

- 标识符必须以字母开头。
- 标识符可以由一个或多个字母、数字（0-9）或特殊字符（$、#、_）组成。
- 标识符长度不超过 30 个字符。
- 标识符名内不能有空格。

例如，下面是合法的标识符：

v_AuthorCode
AuthorCode
salary_$

下面是一些不合法的标识符：

$salary	必须以字母开头
Author Code	标识符中不能有空格
Articles_written_by_female_authors	标识符超过了 30 个字符
1_type	不能以数字开头
Author's_name	在标识符中不能包含单引号（'）

因为 PL/SQL 不区分大小写，所以下面的几个标识符代表同一个对象：

author_code
AUTHOR_code
AUTHOR_CODE
Author_Code

如果希望标识符能够区分大小写或包含其它的字符，则可以使用带双引号的标识符。

除了双引号，任何可打印字符都可作为标识符的一部分。但是，该标识符的最大长度不能是 30 个字符（不包括双引号）。

例如，下面是一些带引号的标识符：

```
"author's_name "
"DECLARE"
"001"
"author code"
```

例如，下面的块中使用了一个带引号的标识符：

```
DECLARE
  "author's age" NUMBER;
BEGIN
  "author's age" :=21;
   DBMS_OUTPUT.PUT_LINE("author's age");
END;
```

2.2.2 定义变量名

PL/SQL 程序中的变量名必须是合法的标识符，但不可以是保留字。在 PL/SQL 中，保留字是有特定意义的标识符。例如，BEGIN 和 END 是两个保留字，它们用来分隔 PL/SQL 块，用户不能将它们作为变量名。

例 1，下面声明的变量是非法的，因为变量名 LOOP 是保留字：

```
DECLARE
LOOP   NUMBER;
```

例 2，保留字可以和其它的标识符（也可以是保留字）共同构成一个合法的变量名：

```
DECLARE
v_LoopDate DATE;
```

其中，变量名 v_LoopDate 中的“Date”和“Loop”是两个关键字，但 v_LoopDate 却是一个合法的关键字。

例 3，如果使用带引号的标识符，则可以使用保留字作为变量名：

```
DECLARE
"LOOP"   NUMBER;
```

其中，尽管 LOOP 为保留字，但由于使用了双引号，所以是合法的。

定义的变量名最好是有意义的。例如，下面的变量名 x 没有任何意义：

```
x   INT;
```

而下面的变量名：

```
V_AuthorName VARCHAR2(10);
```

尽管我们没有加任何注释，但根据上面的变量名就可以估计这个变量是用来存储作家姓名的。PL/SQL 标识符的最大长度是 30 个字符，而用这 30 个字符足可以来描述一个变量的意义了。

下表介绍命名一些 PL/SQL 变量名的习惯格式，其中，下划线前面的字母用来表示特定的意义，下划线后面的部分用来描述变量：

变量名格式	意　　义
V_VariableName	程序变量
E_ExceptionName	用户自定义异常
T_TypeName	用户自定义类型变量
P_ParameterName	存储过程或函数参数变量
C_ConstantValue	用 CONSTANT 关键字限制的变量

2.3 数据类型

在 PL/SQL 版本 8 中，有系统提供的数据类型，也有用户自定义的数据类型。系统提供了四种数据类型：标量类型、复合类型、引用类型和 LOB 类型。下面分别介绍这几种数据类型。

2.3.1 标量类型

标量类型包括以下七类：

- NUMERIC
- CHARACTER
- RAW
- ROWID
- DATE
- BOOLEAN
- TRUSTED

下面介绍这几种标量数据类型以及数据类型间的转换。

2.3.1.1 NUMERIC 类

NUMERIC 类用来存放整数、实数和浮点数。它包含如下三种类型：

- BINARY INTEGER
- NUMBER
- PLS_INTEGER

1. BINARY_INTEGER

BINARY_INTEGER类型用于存储带符号的整数，它的取值范围为-2147483647到2147483647。PLS_INTEGER值和BINARY_INTEGER值比NUMBER值占用较少的存储空间。但是，BINARY_INTEGER操作较PLS_INTEGER操作慢。

基础类型是派生子类型的数据类型。子类型要受到基础类型的约束。子类型主要用于和其它的数据库兼容或便于理解和操作。PL/SQL提供了如下的BINARY_INTEGER子类型：

- NATURAL：存储非负的整数。
- NATURALN：存储非负的整数，但不可以为空。
- POSITIVE：存储正整数。
- POSITIVEN：存储正整数，但不可以为空。

- **SIGNTYPE：存储-1, 0和1。**

2. NUMBER

NUMBER 类型可以存储定点数和浮点数，它的取值范围为从 **1.0E-130** 到 **9.99E125**。

下面分别介绍定义定点数、整型数和浮点数格式。

1) 定点数

定义定点数的格式如下：

NUMBER(precision,scale)

其中，参数 **precision** 指定精度；**scale** 指定标度。标度指定从输入数值的什么位置进行四舍五入，当为 **n** 时，从小数点右边第 **n** 位后进行四舍五入；为 **0** 时，从小数点处进行四舍五入；为 **-n** 时，从小数点左边第 **n** 位前进行四舍五入。而精度指定从四舍五入的位置向左计算，有效数字可有的最多位数，如果输入数值超出精度限制，**Oracle** 将返回出错信息。

对于指定标度的定点数，小数点在整个数字串中从右到左的位置是固定的。其中 precision 的取值范围为 **1~38**，**scale** 的取值范围为 **-84~127**。注意，不能使用常量或变量来指定标度和精度，必须使用整型数据。

下面列表举例说明了 **NUMBER** 数据类型的特点：

数据类型	输入数值	存　　储　　值
NUMBER(4,2)	12.345	12.35
NUMBER(3,2)	12.345	超出精度，出错。
NUMBER(4,-2)	123456	123500
NUMBER(3,-2)	123456	超出精度，出错。
NUMBER(4,5)	0.0012345	0.00123
NUMBER(2,5)	0.0012345	超出精度，出错。
NUMBER(3,-2)	123045	超出精度，出错。(注意：经四舍五入后数值变为 123000，但该数值的有效数字并不只是 1、2、3 三位，3 后面的第一个 0 也为有效数字，所以有效数字共 4 位，超出精度 3 的限制。)

2) 整型数

定义整型数的格式如下：

NUMBER(precision)

其中 **precision** 为精度。

3) 定义浮点数

定义浮点数的语法如下：

NUMBER

这是一个精度为 **38** 位的十进制浮点数。对浮点数来说没有标度的概念，因为小数点可以出现在整个数字串中的任意位置，或者干脆没有小数点。

NUMBER 类型的子类型有：

- **DEC**
- **DECMAL**
- **DOUBLE PRECISION**
- **FLOAT**
- **INTEGER**
- **INT**
- **NUMERIC**
- **REAL**
- **SMALLINT**

其中，子类型**DEC**、**DECIMAL**和**NUMERIC**用于定义定点数，精度的最大值为**38**；子类型**DOUBLE PRECISION**和**FLOAT**用于定义浮点数，精度的最大值为**38**；**REAL**也用于定义浮点数，精度的最大值为**18**；**INTEGER**、**INT**和**SMALLINT**用于定义整数，精度的最大值为**38**。

3. **PLS_INTEGER**

PLS_INTEGER 类型用于存储带符号的整数，取值范围与 **BINARY_INTEGER** 的取值范围相同，都是从**-2147483647** 到**+2147483647**。该类型的操作速度比 **NUMBER** 和

BINARY_INTEGER 类型的要快。因此，为了提高操作性能，在相同的取值范围内，最好使用该数据类型。尽管 PLS_INTEGER 和 BINARY_INTEGER 的取值范围相同，但两者不能完全兼容。当 PLS_INTEGER 类型的值算术运算溢出时，会出现错误。而 BINARY_INTEGER 类型值的算术运算溢出时，如果将该结果赋给 NUMBER 类型的变量，不会出现错误。

2.3.1.2 CHARACTER 类

CHARACTER 类的变量用来存放字符串。CHARACTER 类中的类型有：

- CHAR
- VARCHAR2
- LONG
- NCHAR
- NVARCHAR2

1. CHAR

该类型用来存放固定长度的字符数据，数据的内部表示取决于当前数据库的字符集。定义该类型的格式如下：

CHAR[(maximum_length)];

其中，参数 maximum_length 用于指定最大长度（以字节为单位），可指定的最大值为 32767，缺省值为 1。注意，不能使用常量或变量来指定该值，只能使用 1 到 32767 之间的整数。

如果 CHAR(n)类型的变量用于存储多字节字符，则它的最大长度小于 n 个字符。如果数据不满 n 个字符，则末尾用空格填充。数据库 CHAR 类型列的最大长度为 2000 个字节。所以不能把长度大于 2000 的 CHAR 类型的值插入到数据库列中。但是可以插入到类型为 VARCHAR2 或 LONG 的表列中。

CHAR 的子类型为 CHARACTER，它具有和基础类型相同的取值范围。也可以说，CHARACTER 是 CHAR 的别名。使用该子类型可以与 ANSI/ISO 和 IBM 相兼容。

2. VARCHAR2

VARCHAR2 是变长字符类型，数据的内部表示取决于当前数据库的字符集。定义该类型的格式如下：

VARCHAR2(maximum_length);

其中，参数 **maximum_length** 用于指定最大长度（以字节为单位），可指定的最大值为 **32767**。不能使用常量或变量来指定该值，只能使用 **1** 到 **32767** 之间的整数。

如果 **VARCHAR2 (n)**类型的变量用于存储多字节字符，则它的最大长度小于 **n** 个字符。数据库 **VARCHAR2** 类型列的最大长度为 **4000** 个字节。所以不能把长度大于 **4000** 的 **VARCHAR** 类型的值插入到数据库列中。但是可以插入到类型为 **LONG** 的表列中，因为数据库类型 **LONG** 的最大长度是 **2G** 字节。但是，如果数据库中 **LONG** 类型数据的长度大于 **32767** 字节，则不能赋给 **VARCHAR2** 类型的变量。

3. LONG

LONG 类型用于存储可变长字符数据，最大长度为 **32760** 字节。可以将任意的 **LONG** 类型数据插入到 **LONG** 列中，因为 **LONG** 数据库列的最大长度为 **2G**。但是，不能将数据长度大于 **32760** 的 **LONG** 列值插入到 **LONG** 类型的变量中。

4. NCHAR

NCHAR 类型用于存储固定长度的 **NLS** 字符数据，数据的内部表示取决于当前系统的多字节字符集。定义该类型的格式如下：

NCHAR[(maximum_length)];

其中，参数 **maximum_length** 用于指定最大长度，可指定的最大值为 **32767** 个字节，缺省值为 **1**。注意，不能使用常量或变量来指定该值，只能使用 **1** 到 **32767** 之间的整数。

当多字节字符集的宽度固定时，类型的长度按照字符指定；当多字节字符集的宽度不固定时，则类型的长度按照字节指定。所以，对于长度不固定的多字节字符集，不能把长度大于 **2000** 的 **NCHAR** 类型的值插入到 **NCHAR** 类型的表列中，因为数据库 **NCHAR** 类型列的最大长度为 **2000** 个字节。

要特别注意的是，不能将 **VCHAR** 类型的数据赋给 **CHAR** 类型的变量，反之，也不

能将 CHAR 类型的数据赋给 VCHAR 类型的变量。

5. NVARCHAR2

NVARCHAR2 类型用于存储可变长度的 NLS 字符数据，数据的内部表示取决于当前系统的多字节字符集。定义该类型的格式如下：

NVARCHAR2(maximum_length);

其中，参数 maximum_length 用于指定最大长度，可指定的最大值为 32767。注意，不能使用常量或变量来指定该值，只能使用 1 到 32767 之间的整数。

当多字节字符集的宽度固定时，类型的长度按照字符指定；当多字节字符集的宽度可变时，则类型的长度按照字节指定。所以，对于长度可变的多字节字符集，不能把长度大于 4000 的 NVARCHAR2 类型的值插入到 NVARCHAR2 类型的表列中，因为数据库 NVARCHAR2 类型列的最大长度为 4000 个字节。

要特别注意的是，不能将 NVARCHAR2 类型的数据赋给 VARCHAR2 类型的变量，反之，也不能将 VARCHAR2 类型的数据赋给 NVARCHAR2 类型的变量。

2.3.1.3 RAW 类

RAW 类包括下面两种类型：

- RAW
- LONG RAW

1. RAW

RAW 用来存放固定长度的二进制数据。RAW 类型类似于 CHARACTER 类型，不同点在于将 RAW 类型从一个系统转换到另一个系统时，Net8 不能进行相应的字符集转换。RAW 类型的格式如下：

RAW(maximum_length);

其中，参数 maximum_length 用于指定最大长度，可指定的最大值为 32767。注意，

不能使用常量或变量来指定该值，只能使用 1 到 32767 之间的整数。

数据库类型 RAW 的最大长度是 2000 字节，所以不能将长度大于 2000 字节的 RAW 类型数据插入到 RAW 类型的表列中，但能插入到 LONG RAW 表列（最大长度是 2G 字节）中。如果 LONG RAW 表列值的数据长度大于 32767 字节，则不能赋给 RAW 类型的变量。

2. LONG RAW

LONG RAW 类型用于二进制数据。该类型类似于 LONG 类型，不同点在于 PL/SQL 不能在字符集之间进行转换。

LONG RAW 变量的最大长度是 32760 字节。因为 LONG RAW 表列的最大长度是 2G 字节，所以如果 LONG RAW 表列值的最大长度超过了 32760 字节，则不能赋给 PL/SQL 中的 LONG RAW 变量。但是，可以将 PL/SQL 中任意的 LONG RAW 类型值插入到 LONG RAW 表列中。

2.3.1.4 ROWID 类

每个数据库表都有一个 ROWID 伪列，用于存储行标识符（rowid）。行标识符是固定长度的十六进制字符串，用来表示一条记录的存储地址。在行标识符中包含如下的信息：

- 数据对象号
- 数据文件（第一个文件为 1）
- 数据文件中的数据块
- 数据块中的记录（第一条记录为 0）

使用行标识符可以快速访问特定的记录。通常，行标识符唯一确定一条记录。但是，如果不同表中的记录存储在相同的簇中，则这些记录具有相同的行标识符。行标识符不能由 PL/SQL 程序生成，它可以从一个表的 ROWID 伪列中得到。

2.3.1.5 DATE 类

DATE 类型用于存储长度固定的日期和时间值。可存储的日期范围是公元前 4712 年 1 月 1 日到公元后 4712 年 12 月 31 日。日期的缺省值为当月的第一天；时间的缺省值为午夜 12:00。

PL/SQL 中的 DATE 变量与数据库表中 DATE 类型的列相同，包括世纪、年、月、日、小时、分钟、秒。一个 DATE 变量占七个字节，这七个字节分别表示世纪、年、月、日、小时、分钟、秒。

在表达式中，PL/SQL 自动将字符值转换为缺省日期格式的 DATA 值。缺省的日期格式是由 Oracle 的初始化参数 NLS_DATA_FORMAT 决定的。使用 TO_DATE 内嵌函数可以将 CHARACTER 类型的值转换成 DATE 类型的值；同时，也可以使用 TO_CHAR 内嵌函数将 DATE 类型的值转换成 CHARACTER 类型的值。但要注意，被转换的数据必须是可转换的，例如，不能将字符串‘abc’转换为日期。

2.3.1.6 BOOLEAN 类

BOOLEAN 类型用于存储逻辑值：TRUE、FALSE 和 NULL。NULL 用来表示一个错误的、不合适的或不确定的值。

BOOLEAN 类型的变量只能用在逻辑操作中，且只能将 TRUE、FALSE 或 NULL 赋给该类型的变量。不能将 BOOLEAN 类型的值插入到数据库中，也不能将数据库中的数据赋给 BOOLEAN 类型的变量。

2.3.1.7 TRUSTED 类

MLSLABEL 类型的变量被用在 Trusted Oracle 中存储变长的二进制标签。

2.3.1.8 数据类型之间的转换

PL/SQL 能够处理标量数据类型中不同类型之间的转换。数据类型转换有两种方式：

- 强制类型转换
- 自动类型转换

1. 强制类型转换

强制类型转换是指通过使用函数来完成不同类型之间的转换。这些函数也可以是标准 SQL 中的数据类型转换函数。下表是这些函数的简单描述：

函　　数	描　　述
TO_CHAR	将 NUMERIC 或 DATE 类型值转换成 VARCHAR2 类型值。
TO_DATE	将 CHARACTER 类型值转换成 DATE 类型值。
TO_NUMBER	将 CHARACTER 类型值转换成 NUMBER 类型值。
RAWTOHEX	将 RAW 类型值转换成十六进制数值。
HEXTORAW	将用 CHARACTER 类型描述的十六进制数转换成二进制数。
CHARTOROWID	将用 CHARACTER 类型描述的 ROWID 转换成二进制数。
ROWIDTOCHAR	将二进制形式的 ROWID 转换成 18 个字符的行标识符。

例如，下面的语句用于检索作家王达琳的出生日期：

```
DECLARE
  v_birthdate DATE;
  v_show VARCHAR2(20);
BEGIN
  SELECT birthdate
  INTO v_birthdate
  FROM auths
  WHERE name = '王达琳';
  v_show := TO_CHAR(v_birthdate);
  DBMS_OUTPUT.PUT_LINE('王达琳的出生日期为'|| v_show);
END;
```

在上面的语句中，使用了类型转换函数 TO_CHAR，用来将 DATE 类型的数据转换为 VARCHAR2 类型的数据。

2. 自动类型转换

PL/SQL 能够自动将一些类型的数据转换成另外一种类型的数据。例如：

```
DECLARE
  v_salary VARCHAR2(15);
BEGIN
  SELECT salary
    INTO v_salary
    FROM auths
    where author_code='A00003';
```

END;

数据库中 salary 列的数据类型是 NUMBER(8,2)，但是 v_salary 变量的类型是 VARCHAR2(15)，PL/SQL 能将 NUMERIC 类数据自动转换成字符串并赋给 CHARACTER 类的变量。

在自动转换时，要特别注意的是进行自动转换的数据必须是可转换的。例如，可以将 CHAR 类型的“123”自动转换为 NUMBER 类型的 123。但不能将 CHAR 类型的“ABC”转换为 NUMBER 类型的值。

下面列表给出了能够实现自动转换的数据类型（X 表示能够实现自动转换）：

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

尽管 PL/SQL 能够自动转换数据类型，尤其在算术表达式中，但最好还是用强制数据类型转换。强制数据类型转换能够提高程序的清晰度，并使程序更便于阅读和理解。

2.3.2 复合类型

复合类型是指具有内部成员的类型，该成员可以被单独操作。通常，该类型的变量由一个或多个标量类型组成。在 PL/SQL 中有两种复合类型：

- 记录
- 表

2.3.2.1 记录

与预定义的标量类型不同，在使用复合类型之前用户必须先定义该复合类型。定义记录类型的语法如下：

```
TYPE record_type IS RECORD(
  field1   type1 [NOT NULL] [:=expr1],
  field2   type2 [NOT NULL] [:=expr2],
  ...
  fieldn   typen [NOT NULL] [:=exprn]);
```

其中 record_type 表示记录类型的名字，field 表示记录里域的名字，Type 表示记录里域的类型，expr 表示为域赋的初始值。

在定义记录类型时，对域的声明要注意以下几点：

- 域的声明与记录外的变量声明类似，可以有 NOT NULL 约束和初始值。
- 如果一个域没有被赋值，则系统自动将其赋值为 NULL。
- 在赋初值时，“:=”可以用关键字 DEFAULT 来代替。

当定义了一个记录类型后，就可以声明记录类型的变量了。

例 1，下面首先定义一个关于作家文章信息的记录类型 t_ArticleRec，然后声明了两个该类型的变量 v_Article1 和 v_article2：

```
DECLARE
  TYPE t_ArticleRec IS RECORD(
    ArticleCode        VARCHAR2(10) ,

  --在下面的域声明中，用关键字DEFAULT代替了“:=”
  Secrate_level    CHAR(1)   DEFAULT   '1',

  --下面的变量有NOT NULL约束，所以必须赋初值
  Title     VARCHAR2(120) NOT NULL:='not know');

  --声明t_ArticleRec记录类型的变量v_Article1和v_article2
  v_Article1   t_ArticleRec;
  v_article2   t_ArticleRec;
```

当定义了一个记录类型变量后，可按照下面的语法引用记录中的域：

```
record_name.field_name
```

其中 record_name 是记录名，field_name 是域名。

例 2，继续前面的例子：

```
BEGIN
  v_Article1.ArticleCode:='AS6';
  v_Article2.Title:='C++ Programing';
END;
```

我们不仅可以为记录的域赋值，具有相同类型的两个记录也可以相互赋值。

例 3，前面已定义的记录类型为 t_ArticleRec 的两个变量 v_Article1、v_Article2 可以互相赋值：

```
v_Article1:=v_Article2;
```

记录的赋值类似于记录之间的复制，v_Article2 中域的值赋给了 v_Article1 中对应的域。但值得注意的是两个具有相同域定义的不同记录类型是不能互相赋值的。

例 4，也可以用 SELECT 语句来为记录赋值，这种方法是从数据库表中检索出数据，然后将它们存入记录中。在这里要注意，记录中域的类型一定要和 SELECT 列表中域的类型一致：

```
DECLARE

  --定义一个记录，它的域和auths表中的一些域类型相同。
  TYPE t_AuthRec IS RECORD(
    AuthorCode        auths.author_code%type,
    Name              auths.name%type,
    Birthdate         auths.birthdate%type,
    EntryDateTime          auths.entry_date_time%type);

  --声明一个记录类型变量来接收数据。
  v_Author   t_AuthRec;

BEGIN
```

```
--检索作家代码是A00009的作家信息，并将其存入v_Author记录变量中。
 SELECT      author_code,name,birthdate,entry_date_time
    INTO     v_Author
    FROM     auths
    WHERE    author_code='A00009';
END;
```

例 5，在 PL/SQL 中经常会遇到这样的声明——声明一个与数据库表中各列类型相同的记录。为了简化这种记录的声明，PL/SQL 提供了%ROWTYPE 操作符。类似于%TYPE，%ROWTYPE 返回一个记录类型，这个记录类型中域类型与定义该记录的表中各列的类型相同。下面定义了一个与表 article 相同的记录：

```
DECLARE
  v_Article   article%ROWTYPE;
```

使用 %ROWTYPE 声明的记录不包括数据库表中列的 NOT NULL 约束，但包括 VARCHAR2 列和 CHAR 列的长度，NUMBER 列的精度和标度。当表定义发生改变时，由 %ROWTYPE 声明的记录也发生相应的改变。

2.3.2.2 表

定义一个表类型的语法如下所示：

```
TYPE tabletype IS TABLE OF type INDEX BY BINARY_INTEGER
```

其中 Tabletype 是被定义的表类型的名，Type 是预定义的标量类型或通过%TYPE 来引用的标量类型。表的索引是 BINARY_INTEGER 类型。当定义了表类型后，就可以声明该类型的变量了。

例 1，下面先定义两个表类型 t_NameTable 和 t_Address，然后分别声明这两种类型的变量：

```
DECLARE

  --定义表类型t_NameTable，其元素类型是auths数据库表中name列的类型。
  TYPE t_NameTable IS TABLE OF auths.name%TYPE
```

```
    INDEX BY BINARY_INTEGER;

--定义表类型t_Address，其元素类型是auths表中的address列的类型。
TYPE t_AddressTable IS TABLE OF auths.address%TYPE
    INDEX BY BINARY_INTEGER;

--声明两个表类型变量。
v_name t_NameTable;
v_Address t_AddressTable;
```

当定义了表类型，并声明了这个类型的变量后，就可以通过下面的语法引用 PL/SQL 表中的元素了：

```
tablename(index)
```

其中 tablename 是表名，index 是表的索引，它用 BINARY_INTEGER 类型的变量或是能够转换成 BINARY_INTEGER 类型的表达式表示。

例 2，使用例 1 中声明的表，为表元素赋值：

```
BEGIN
  v_Name(2):='Wang';
  v_Address(-3):='street 1';
END;
```

例 3，下面定义一个记录表 t_AuthorRecord，表元素类型是与数据库表 auths 匹配的记录类型，然后检索数据库表 auths 中作家代码是“A00009”的作家记录，并将其存放到创建的 t_AuthorRecord 中：

```
DECLARE
  TYPE t_AuthorRecord IS TABLE OF auths%ROWTYPE
     INDEX BY  BINARY_INTEGER;
  --每一个元素都是一条记录。
  v_Auths t_AuthorRecord;
BEGIN
  --检索作家代码是A00009的作家，并存放到v_Auths(00009)中。
  SELECT *
```

```
        INTO v_Auths(00009)
        FROM auths
        WHERE author_code='A00009';
    END;
```

因为 v_Auths 表中的每一个元素都是记录，所以可以通过下面的语法引用记录中的域：

table(index).field

继续前面的例子：

```
v_Auths(00009).name:='Join';
--将v_Auths(00009).name输出到屏幕上
DBMS_OUTPUT.PUT_LINE(v_Auths(00009).name);
```

v_Auths(00009)是一个 Auths%ROWTYPE 类型的记录，name 是这条记录中的一个域。

记录表在相当大程度上提高了 PL/SQL 表的功能。因为一个记录表就可以存放一个数据库表的所有信息。

我们在前面已介绍了 PL/SQL 表的定义和引用。从定义 PL/SQL 表类型的语法看，PL/SQL 表与数组一样，但表的实际结构与数组是不一样的。它与 C 中数组有以下不同：

- PL/SQL 表中的元素没有特定的顺序。因为表中元素不象数组一样被连续地存储在一个空间内。
- 在 PL/SQL 表中的 KEY 不必是连续的，任意 BINARY_INTEGER 类型的值或是表达式都可用作表的索引（KEY 列）。
- 当向一个 PL/SQL 表中插入一个元素时，就为该表分配了用以存放该元素存储空间。PL/SQL 表的大小是没有限制的。

2.3.2.2.1 表属性

PL/SQL 中，不仅引入了表，而且通过表属性扩展了 PL/SQL 表的功能。使用表属性的语法是：

table.attribute

其中 table 表示一个表类型变量，attribute 表示要使用的表属性。PL/SQL 中的表属性如下表所示：

表属性	语　　法	返回类型	描　　述
COUNT	Table. COUNT	NUMBER	返回 PL/SQL 表中元素个数。
DELETE	Table.DELETE	N/A	删除 PL/SQL 表中所有的元素。
	Table.DELETE(i)	N/A	删除 PL/SQL 表中索引号“i”指定的元素。
	Table.DELETE(i,j)	N/A	删除 PL/SQL 表中索引号“i”到“j”之间（包括“i”和“”j）指定的元素。
EXISTS	Table.EXISTS(i)	BOOLEAN	如果 PL/SQL 表中索引号为“i”的元素存在，则返回 TRUE。
FIRST	Table. FIRST	BINARY _INTEGER	返回 PL/SQL 表中第一个元素（索引号最小）的索引。
LAST	Table. LAST	BINARY _INTEGER	返回 PL/SQL 表中最后一个元素（索引号最大）的索引。
NEXT	Table. NEXT(i)	BINARY _INTEGER	返回 PL/SQL 表中索引号为“i”的元素的后继一个元素的索引。
PRIOR	Table. PRIOR (i)	BINARY _INTEGER	返回 PL/SQL 表中索引号“i”的元素的前一个元素的索引。

例 1，下例通过使用 PL/SQL 表的 DELETE 属性来删除表 t_CharRecord 中的元素。在删除的过程中，使用 COUNT 属性查看表 t_CharCount 中的元素个数：

```
DECLARE
  TYPE t_CharRecord IS TABLE OF VARCHAR2(20)
    INDEX BY BINARY_INTEGER;
  v_Chars t_CharRecord;
  v_Counter NUMBER;
BEGIN
  --向表中插入元素。
  v_Chars(-6):='negative';
  v_Chars(0):='zero';
  v_Chars(4):='Plus';
  v_Chars(90):='ninety';
```

```
  --删除前，PL/SQL表中元素的个数。
  v_Counter:=v_Chars.COUNT;

  --删除索引号为90的元素，并得到删除后的元素个数。
  v_Chars.DELETE(90);
  v_Counter:=v_Chars.COUNT;

  v_Chars.DELETE(0,4); --删除索引号从0到4的元素。
  v_Counter:=v_Chars.COUNT;

  v_Chars.DELETE;  --删除PL/SQL表中的所有元素。
  v_Counter:=v_Chars.COUNT;
END;
```

注意，DELETE 只能作为一条完整的语句使用，它不能作为表达式中的一部分被调用，这与其它的表属性不同。

例 2，通过 FIRST 和 LAST 属性查找 PL/SQL 表 t_AuthCodeRec 中第一个元素和最后一个元素的索引号，通过 NEXT 属性查找索引号为“00001”的元素后的元素索引号：

```
DECLARE
  /*定义一个PL/SQL表，表元素与数据库表auths的author_code列的类型一致。*/
  TYPE t_AuthCodeRec IS TABLE OF auths.author_code%TYPE
    INDEX BY BINARY_INTEGER;
  v_AuthCode t_AuthCodeRec;
  v_Index BINARY_INTEGER;
BEGIN
  v_AuthCode(00001):='A00001';
  v_AuthCode(00006):='A00006';
  v_AuthCode(00009):='A00009';

  --得到第一个元素的索引号。
  v_Index:=v_AuthCode.FIRST;

  --得到索引号为“00001”的元素后的元素索引号。
  v_Index:=v_AuthCode.NEXT(00001);
```

```
  --得到最后一个元素的索引号。
  v_Index:=v_AuthCode.LAST;
END;
```

最后讲讲在使用 PL/SQL 表时要注意的几点：

- 尽管在 PL/SQL2.3 以上版本可以使用 COUNT 属性得到 PL/SQL 表中的元素个数，但最好还是专门用一个变量来连续记录表的元素个数。因为表的大小（元素个数）是没有限制的，而在程序中需要确定表中的不断增加的元素个数。

- 表中的索引号从 1 开始，每增加一个元素，索引号随之加 1。也就是说索引号为 1 的下一个元素的索引号是 2，再下一个元素的索引号为 3，以此类推。使用这样的方法建立的 PL/SQL 表，其表元素在控制结构中的循环（循环控制结构将在第三章讲解）实现起来更简单。当 PL/SQL 块在 Pro*C 或 OCI 程序中被调用或被嵌入时，使用上述索引方法建立的 PL/SQL 表能够绑定到 C 语言中的数组上。

- 一个表元素在被赋值之前不能被引用。如果表元素在赋值之前的引用，就会出现 ORA-1403 错误。

- 除了使用 DELETE 属性删除一个表中的所有元素以外，还可以将刚建立的表（与准备删除元素的表的类型一致）赋给它来删除表中所有元素。这是因为一个表刚建立时是没有元素的。

例如：

```
DECLARE
  TYPE t_NameTable IS TABLE OF auths.name%TYPE
    INDEX BY BINARY_INTEGER;
  v_Name t_NameTable;
  v_Empty t_NameTable;
BEGIN
  --为表v_Name赋值。
  v_Name(1):='Zhang';
  v_Name(2):='Wang';
  v_Name(3):='Li';
  --删除表v_Name中的所有元素。
```

```
  v_Name:=v_Empty;
END;
```

2.3.3 引用类型

PL/SQL 中的引用类型变量与 C 中的指针类似，该类型用于存储指向存储空间的指针。在 PL/SQL 中，引用类型的变量包括游标变量（REF CURSOR）（有关游标变量的详细内容请参看“游标”一章）和对象引用类型（REF object type）（有关对象引用类型请参看“对象类型”一章）。

2.3.4 LOB 类型

LOB 类型用来存储大对象。大对象包括不超过 4G 字节的二进制数据或字符数据。我们通过 DBMS_LOB 包来操作 LOB 类型。有关大对象的详细内容，请参看本书大“对象”一章。

2.3.5 用户自定义子类型

PL/SQL的每个基础类型都有其对应的值和操作的集合。子类型和基础类型的操作相同，但值的集合为基础类型的子集。PL/SQL在STANDARD包中已预定义了几种子类型，例如DEC。除了预定义的子类型外，用户也可以在PL/SQL块的定义部分定义自己的子类型，其语法如下：

```
SUBTYPE subtype_name IS base_type;
```

其中：subtype_name 是定义的子类型名；base_type 为定义子类型的基本类型。基本类型可以是基础类型或子类型，也可以是%TYPE 引用或%ROWTYPE。

例如：下面的语句定义了基础类型为 NUMBER 的子类型 t_num 和基础类型为 auths.author_code%type 的子类型 t_authorcode：

```
DECLARE
  SUBTYPE t_num IS NUMBER;
  v_num t_num;
  SUBTYPE t_authorcode IS auths.author_code%type;
```

```
v_authorcode t_authorcode;
```

在定义子类型时，不能指定基本类型的长度（例如 VARCHAR2（20））、精度和标度（例如 NUMBER（2））。

例如，下面的子类型定义是错误的：

```
DECLARE
  SUBTYPE t_num IS NUMBER(4);
```

但下面的子类型定义是合法的：

```
DECLARE
  --v_temp变量不被引用，只是通过它定义一个受限制的子类型
  v_temp NUMBER(4);
  SUBTYPE t_num IS v_temp%type;
  v_num t_num;
```

我们可以限制子类型变量的长度、精度和标度。

例如，下面使用子类型来声明变量：

```
DECLARE
  SUBTYPE t_num IS NUMBER;   --定义一个NUMBER子类型
  v_num t_num(4);      --v_num是一个NUMBER(4)类型的变量
```

2.4 数据

PL/SQL 中的数据有字符型数据、数值型数据和布尔型数据。

2.4.1 字符型数据

字符型数据是指括在单引号中的字符串。这些字符串可以赋给 CHAR 或 VARCHAR 类型的变量。例如，下面是一些合法的字符型数据：

```
'A00001'
```

'10-10-1998'
'$10,123,500'

PL/SQL 字符集中的任意可打印字符都可以是字符串的一部分，包括单引号。因为单引号是字符串的分隔符，为了在字符串中引用单引号，则必须使用两个连续的单引号。

例 **1**：

'Author''s salary'

上面的字符串表示：**Author's salary** 。

例 **2**：

''''

上面是四个单引号，第一个单引号是字符串分隔符的开始符，中间连续两个单引号是代表字符串中的一个单引号，第四个单引号是字符串分隔符的结束符。

例 **3**：

''

上面的字符串是空串，它等同于 **NULL**。

2.4.2 数值型数据

数值型数据分为整数或实数，可以将其赋给 **NUMBER** 类型的变量。只有数值型数据能作为算术表达式的一部分。

整数是指没有小数点的完整数值，可以带有正负号。例如，下面是一些合法的整数：

4567，-78，，+34，0

实数是指带有小数点的数值。例如，下面是一些合法的实数：

-37.8，45.0，23.

在上面的例子中，尽管 23.和 45.0 没有小数部分，但它们依然是实数。

实数还可以用科学记数法来表示。例如，下面是几个用科学记数法表示的实数：

7.34E7，5.345E-3，-6.67E+10

用科学记数法表示的实数中，“E”之后的数字必须是整数。上面用科学记数法表示的实数的大小如下所示：

7.34E7 =7.34 乘以 10 的 7 次方 =73,400,000
5.345E-3 = 5.345 乘以 10 的-3 次方 = 0.005345
-6.67E+10= -6.67 乘以 10 的 10 次方 = -66,700,000,000

2.4.3 布尔型数据

布尔型数据有三个值：**TRUE**、**FALSE** 和 **NULL**。要特别注意的是，**TRUE**、**FALSE** 和 **NULL** 是数据，而不是字符串。布尔型数据可赋给 **BOOLEAN** 类型的变量，但不能将布尔型数据插入到数据库表中。**BOOLEAN** 类型的变量通常用来判断条件的真假，所以一般被用在 **IF** 和 **LOOP** 语句中。

2.5 变量声明

变量是存储数据的空间，用于存储程序中的值。声明变量时，为变量分配存储空间、指定数据类型和存储空间的名称（也即变量名）。下面介绍声明变量的语法以及变量的作用域和可视域。

2.5.1 声明变量的语法

变量在块的定义部分被声明，声明变量的一般语法为：

variable_name [CONSTANT] type [NOT NULL] [:=value];

其中：

- variable_name：变量名。
- type：变量类型。
- CONSTANT：表示声明的是一个常量，且必须为这个常量赋初值。
- NOT NULL：表示该变量不能为空值，即必须赋初值。
- value：变量的初始值。

使用上面的语法可以定义变量和常量。在程序的运行过程中，变量的值是可变的，而常量的值是不可变的。变量声明时应注意以下几点：

- 任何合法的标识符（包括被引用标识符）都可以作为变量名。
- 如果一个变量定义为 NOT NULL，则这个变量必须被赋初值。
- 一个定义为 NOT NULL 的变量在块中的可执行部分或其它部分中不能被赋值为 NULL。
- 如果没有给变量赋初值，则系统自动将这个变量赋值为 NULL。
- 在声明变量时，可以用“:=”或“default”来赋初值，两种方式是等价的。

例 1，下面的变量声明是合法的：

```
DECLARE
  v_Name        VARCHAR2(20);
  v_Sex         NUMBER := 1;
```

例 2，下面的变量声明将会出错，因为 v_Temp 被定义为 NOT NULL，但没有赋初值：

```
DECLARE
  v_Sex NUMBER NOT NULL;
```

正确写法为：

```
DECLARE
  V_Sex NUMBER NOT NULL := 1;
```

例 3，常量声明：

```
DECLARE
  v_AuthorCode CONSTANT CHAR(6) := 'A00009';
```

例 4，当为一个变量赋初值时，也可以用 DEFAULT 来代替“:=”，如下面的变量声明：

```
DECLARE
  v_Sex      NUMBER      DEFAULT 1;
  v_Name     VARHAR(20)  DEFAULT '王达琳';
```

例 5. 在定义部分每一行只能有一个变量被声明，下面的变量声明是非法的：

```
DECLARE
  v_AuthorName   v_ArticleCode   VARVHAR2(30);
```

正确的变量声明应当为：

```
DECLARE
  v_AuthorName VARVHAR2(30);
  v_ArticleCode VARCHAR2(30);
```

2.5.2 变量的作用域和可视域

在程序中使用变量时，要受到变量的作用域和可视域的限制。

变量的作用域是指变量在程序中的有效范围。对于一个 PL/SQL 变量，它的作用域是从该变量被声明开始到变量所在块结束。当变量超过了这个范围，则 PL/SQL 引擎将释放存放该变量的空间，这个变量就不存在了，也就无法再访问了。

例如，下图中的变量 v_num 和 v_char 分别是在块及其子块中定义的变量，它们的作用域分别在各自所在的块中：

```
                    DECLARE
                      v_num NUMBER(3,4);
                    BEGIN
                      DECLARE
                        v_char CHAR(10);
v_num 的作用域        BEGIN                  v_char 的作用域
                        ...
                      END;
                    END;
```

变量的可视域是指在变量前不加以限定就能够直接访问该变量的那一段程序。变量的可视域总在作用域之内，如果变量超出了作用域，则这个变量也将不可视。

例如，下图中的变量 v_num 和 v_char 分别是在块及其子块中定义的变量，它们的可视域分别在各自的作用域中：

```
                    DECLARE
                      v_num NUMBER(3,4);
                    BEGIN
                      DECLARE
                        v_char CHAR(10);
v_num 的可视域        BEGIN                  v_char 的可视域
                        ...
                      END;
                    END;
```

当在块中声明一个变量时，如果在子块中也声明了同名的变量，则它们的作用域没变。但是，在子块中，只有子块中声明的变量是可视的，这时要引用父块中的变量名时，必须加以限定。

例如，在块及其子块中都定义了变量 v_num，它们的作用域及可视域如下图所示：

全局变量 v_num 的作用域和可视域

```
DECLARE
  v_num NUMBER(3,4);
BEGIN
  ...
  DECLARE
    v_num CHAR(10);
  BEGIN
    ...
  END;
  ...
END;
```

作用域

可视域

可视域

局部变量 v_num 的作用域和可视域

```
DECLARE
  v_num NUMBER(3,4);
BEGIN
  ...
  DECLARE
    v_num CHAR(10);
  BEGIN
    ...
  END;
  ...
END;
```

作用域

可视域

这时，如果要在子块中引用父块中的同名变量时，必须对该变量加以限定。可以使用标识号来进行限定。

例如，下面在子块中引用父块中的变量 v_num 时，使用父块的标号 l_global 限定该变量：

```
<<l_global>>
DECLARE
  v_num NUMBER(3,4);
  ...
BEGIN
  ...
  DECLARE
    v_num CHAR(10);
    ...
  BEGIN
    ...
    l_global. v_num := v_num
  END;
  ...
END;
```

注意，不能在块中的定义部分同时声明两个同名的变量，但可以在不同的块中声明同名的变量。这两个变量可以存储不同的数据，且修改其中的一个变量也不会影响另一个变

量。如果这两个块的级别相同，则不能在一个块中引用另一个块中的变量。

例如，在下面的语句中，块l_local1和块l_local2是两个同级的块。由于在块l_local2中引用了块l_local1中的变量v_num，所以，该语句是不合法的：

```
BEGIN
  <<l_local1>>
  DECLARE
    v_num 1 INT;
  BEGIN
    ...
  END;
  <<l_local2>>
  DECLARE
    v_num1 INT;
  BEGIN
    ...
    l_local1.v_num1 := 2;
    DBMS_OUTPUT.PUT_LINE(l_local1.v_num1);
  END;
END;
```

2.6 赋值语句

每当进入一个块或子程序时，都要初始化变量和常量。缺省情况下，变量被初始化为NULL。因此，可以使用赋值语句为该变量赋初值。赋值是最基本的操作，其语法如下：

```
variable := expression;
```

其中：

- variable：变量名。
- expression：赋给变量的值。

可以在块的各个部分使用赋值语句。例如：

```
DECLARE
```

```
  v_authorcode VARCHAR2(10);
  v_error VARCHAR2(30);
  v_salary NUMBER :=120;
BEGIN
  v_authorcode:= 'A00001';
EXCEPTION
  v_error := 'Data are not found! ';
END;
```

在上例中，分别在块的定义部分、执行部分和异常处理部分使用了赋值语句。

注意，赋值操作符（:=）右边可以是任意的表达式，但表达式值的数据类型必须和变量的数据类型相同或相兼容，或者能够自动转换。

2.7 表达式

表达式是由运算符和操作数组成，操作数可以是变量、常量、数据或函数。PL/SQL 包含一元运算符和二元运算符，例如负号“-”为一元运算符；除号“/”为二元运算符。

优先级是指在同一表达式中对不同运算符求值的顺序。当在表达式中包含多个运算符时，先对优先级高的运算符求值，后对优先级低的运算符求值。对于优先级相同的运算符，按照从左到右的顺序求值。

下表由高到低列出了 PL/SQL 运算符的优先级，同一行的运算符优先级相同。

运算符	运算
**,NOT	求幂，逻辑非
+, -	正、负
*, /	乘、除
+, -, ‖	加、减、连接
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	比较
AND	与
OR	或

例如，在下面的表达式中，乘法的优先级高于加法，所以首先计算 2 乘以 3，然后在此结果上加 1：

1+2*3

因此，表达式的值为 7。

可以在表达式中使用括号来改变运算符的优先级。先对括号内的表达式求值，后对括号外的表达式求值。例如：表达式(1+2)*3 的值为 9。

按照表达式值的数据类型，可以将表达式分为算术表达式、字符表达式和布尔表达式。

2.7.1 算术表达式

算术表达式是指值为NUMERIC类的表达式。算术表达式中使用的运算符有加（+）、减（-）、乘（*）、除（/）和幂（**）。

例如，下面是一个算术表达式：

v_Salary + v_Salary * 70%

其中，变量v_Salary的数据类型为NUMBER(5,1)。

要特别注意的是，如果算术表达式中的一个操作数为NULL，则表达式的值为NULL。

例如，下面表达式的值为NULL：

（v_Salary + v_Salary * 70% ）+ NULL

2.7.2 字符表达式

字符表达式是指值为 CHARACTER 类的表达式。表达式中使用的运算符为连接运算符（||），该操作符用来将两个或多个字符串（或是可自动转换成字符串的其它数据）连接起来。

例如，下面是一个字符表达式：

'The author''name is '||v_name||'!'

如果字符表达式中的所有操作数都是 CHAR 类型，则表达式的结果也是 CHAR 类型。但如果其中一个操作数的类型是 VARCHAR2，则表达式的结果就是 VARCHAR2 类型。

例 2，在下面的块中将字符表达式的值赋给变量 v_info：

```
DECLARE
  v_name VARCHAR2(10);
  v_info VARCHAR2(30);
BEGIN
  SELECT name
    INTO v_name
    FROM auths
    WHERE author_code = 'A00001';
  v_info :='作家姓名为：'||v_name||'!';
  DBMS_OUTPUT.PUT_LINE(v_info);
END;
```

注意，PL/SQL将长度为0的字符串当作NULL来处理。例如，下面的两条赋值语句：

```
v_string1 := 'author''s name'||'';
v_string2 := 'author''s name'||NULL;
```

其中，变量v_string1和变量v_string2的值相同，都是'author's name'。

2.7.3 逻辑表达式

逻辑表达式是指值为布尔类型的表达式。在该表达式中，通常使用比较运算符和逻辑运算符。下面分别介绍比较运算符和逻辑运算符。

2.7.3.1 比较运算符

比较运算符用于将一个表达式同另一个表达式进行比较，该表达式的值总是 TRUE、FALSE 或 NULL。下表列出了 PL/SQL 中用到的比较运算符：

运算符	用　　途
=	比较两个操作数是否相等。如果相等，则返回 TRUE；如果不相等，则返回 FALSE；如果操作数中包含 NULL，则返回 NULL。
<>, !=, ~=	比较两个操作数是否不等。如果不相等，则返回 TRUE；如果相等，则返回 FALSE；如果操作数中包含 NULL，则返回 NULL。
>	比较操作数 1 是否大于操作数 2。如果大于，则返回 TRUE；如果不大于，则返回 FALSE；如果操作数中包含 NULL，则返回 NULL。
>=	比较操作数 1 是否大于等于操作数 2。如果大于等于，则返回 TRUE；如果小于，则返回 FALSE；如果操作数中包含 NULL，则返回 NULL。
<	比较操作数 1 是否小于操作数 2。如果小于，则返回 TRUE；如果不小于，则返回 FALSE；如果操作数中包含 NULL，则返回 NULL。
<=	比较操作数 1 是否小于等于操作数 2。如果大于等于，则返回 TRUE；如果大于，则返回 FALSE；如果操作数中包含 NULL，则返回 NULL。
IN	检测操作数是否等于其中任意成员。如果是，则返回 TRUE；否则，返回 FALSE。如果包含 NULL，则返回 NULL。
NOT IN	检测操作数是否不等于其中任意一个成员。如果不等于，则返回 TRUE；否则，返回 FALSE。如果包含 NULL，则返回 NULL。
[NOT] ETWEEN x AND y	检测操作数是否大于等于 x 且小于等于 y。如果是，则返回 TRUE；如果否，返回 FALSE；如果操作数中包含 NULL，则返回 NULL。
X [NOT] LIKE y	检测 x 与匹配模式 y 是否匹配。如果是，则返回 TRUE；如果否，返回 FALSE；如果操作数中包含 NULL，则返回 NULL。
IS [NOT] NULL	检测操作数是否为 NULL。如果是，则返回 TRUE；如果否，返回 FALSE。

例1．在下面语句的WHERE子句中使用了布尔表达式：

```
DECLARE
  v_name VARCHAR2(10);
BEGIN
  SELECT name
    INTO v_name
    FROM auths
    WHERE author_code = 'A00001';
END;
```

上面的语句用于检索代码为'A00001'的作家名，并将该作家名赋给变量v_name。

例2，下面是使用关系操作符IS NULL的例子：

```
BEGIN
  UPDATE auths
    SET salary = 200
    WHERE salary IS NULL;
END;
```

上面的语句用于将表auths中未定的工资设置为200。其中，WHERE子句中的布尔表达式用于检测salary列的值是否为NULL。注意，千万不能将WHERE子句写成如下的形式：

```
WHERE salary = NULL;
```

例3，下面是使用关系操作符LIKE的例子：

```
type_code LIKE 'A%'
```

其中，'%'（百分号）为通配符，用于同0个或多个字符进行匹配。例如，当type_code的值为 'A'、'AA' 或 'ABC' 时，该布尔表达式的值都为TRUE。

在使用LIKE运算符时，还可以使用通配符 '_'（下划线），用于同另一个字符精确匹配。例如，下面的布尔表达式：

```
type_code LIKE 'A_'
```

当type_code的值为 'AA' 或 'AB'时，该表达式的值都为TRUE；如果type_code的值为 'A' 或 'ABC' ，则该表达式的值为FALSE。

2.7.3.2 逻辑运算符

逻辑运算符把两个或两个以上的单个布尔表达式组合起来，产生一个逻辑值；或者将单一布尔表达式的结果取反。在 PL/SQL 中共用到三种逻辑运算符：NOT、AND 和 OR，下面分别介绍。

1. NOT 运算符

使用 NOT 运算符时，如果后面表达式的值为 FALSE 则返回 TRUE；如果值为 TRUE 则返回 FALSE；如果值为 NULL，则结果仍为 NULL。如下表所示：

NOT	操作数		
	TRUE	FALSE	NULL
结　果	FALSE	TRUE	NULL

2. AND 运算符

使用 AND 运算符时，如果被操作的两个表达式的结果均为 TRUE，则返回 TRUE；如果其中任意一个表达式的结果为 FALSE，则返回 FALSE；否则返回 NULL。如下表所示：

AND		操作数		
		TRUE	FALSE	NULL
操作数	TRUE	TRUE	FALSE	NULL
	FALSE	FALSE	FALSE	FALSE
	NULL	NULL	FALSE	NULL

3. OR 运算符

当在条件中使用运算符 OR 时，如果任意一个条件的结果为 TRUE 则返回 TRUE；如果两个均为 FALSE 则返回 FALSE；否则返回 NULL。如下表所示：

OR		操作数		
		TRUE	FALSE	NULL
操作数	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	NULL
	NULL	TRUE	NULL	NULL

例如，

(v_salary > 200 AND v_salary < 500) OR v_salary = 1000

上面的表达式使用了逻辑操作符AND和OR，当变量v_salary的值大于200且小于500，或等于1000时，表达式的值为TRUE。

要特别注意的是，当计算逻辑表达式时，采取短路计算。也就是说，当能够确定表达式的值时就终止表达式的计算。例如，下面的表达式：

(v_salary > 200 AND v_salary < 500) OR v_salary = 1000

当变量v_salary的值为300时，表达式(v_salary > 200 AND v_salary < 500)的值为TRUE，这时，就确定整个表达式的值为TRUE。就不再对表达式 v_salary = 1000 进行判别了。

2.8 注释

注释是用来告诉用户一个程序要完成什么样的工作或是如何来完成的。在编写PL/SQL程序时，可以使用注释来提高程序的可读性。在执行该程序时，PL/SQL编译器忽略注释。

注释一般写在下面几个位置：

- 写在块的前面：用来解释块或存储过程要做的工作。
- 写在变量的声明旁：用来说明这个变量的作用。
- 写在每一个块的主要部分之前：用来解释一组语句完成什么样的功能。

注释分单行注释和多行注释，下面分别介绍。

1. 单行注释

单行注释以两条短线“--”开始，以行结束符（回车键）结束。顾名思义，单行注释只能写在一行中，且只能写在行的末端

例如，

```
--向表 type 中插入数据
DECLARE

  --块的定义部分
  v_TypeCode VARCHAR2(12):='BAB';
  v_TypeRemark VARCHAR2(30):='纺织工业';
BEGIN
```

```
  --块的执行部分
  INSERT INTO type
    VALUES(v_TypeCode,v_TypeRemark);
END;
```

其中，第一条注释语句放在块的前面，用来说明 PL/SQL 块要做的工作；第二条和第三条注释语句放在块的定义部分和执行部分，用来说明这两组语句的作用。

注意，如果某个PL/SQL块是由Oracle的预编译程序动态地处理，则不能在块中使用单行注释，因为行结束符被忽略了。此时，最好使用多行注释。

2. 多行注释

多行注释以符号“/*”开始，并以符号“*/”结束，PL/SQL 编译器将中间的所有内容都认为是注释内容。多行注释可以写在一行中，也可以写在多行中。

例如，

```
DECLARE
  v_TypeCode VARCHAR2(12):= 'BAB';   /*用来存放文章类
                                       型代码的变量。*/
  v_TypeRemark VARCHAR2(30):= '纺织工业'; /*用来存放文章
                                       内容的变量。*/
BEGIN
  /*向 type 表中插入文章类型
    代码和文章内容。*/
  INSERT INTO type
    VALUES(v_TypeCode,v_TypeRemark);
END;
```

其中，第一条和第二条注释放在变量声明旁，用来说明变量的用途；第三条注释放在块的声明部分，用来说明这组语句的作用。

3. 注释的嵌套

多行注释可使注释延伸多行。但在多行注释中不能嵌套多行注释。例如，下面语句中

的第一条注释语句是非法的：

```
/*下面的块用于向表 type 中插入数据，
      其中/*存储文章类型编码。*/为多行注释*/
DECLARE
  v_TypeCode VARCHAR2(12):='BAB'; /*存储文章类型编码。*/
  v_TypeRemark VARCHAR2(30):='纺织工业';
BEGIN
  INSERT INTO type
    VALUES(v_TypeCode,v_TypeRemark);
END;
```

但是，可以在多行注释中嵌套单行注释，

例如，下面的注释是合法的：

```
/*下面的块用于向表 type 中插入数据，
    其中--块的定义部分为单行注释 */
DECLARE
  --块的定义部分
  v_TypeCode VARCHAR2(12):='BAB';
  v_TypeRemark VARCHAR2(30):='纺织工业';
BEGIN
  INSERT INTO type
    VALUES(v_TypeCode,v_TypeRemark);
END;
```

此外，在单行注释中可以嵌套单行注释，也可以嵌套多行注释（注意，被嵌套的多行注释不能折行）。

例如，下面的注释是合法的：

```
--向表 type 中插入数据，程序中的/*存储文章类型编码。*/为多行注释
DECLARE

  --块的定义部分，与注释--块的执行部分的作用相同
  v_TypeCode VARCHAR2(12):='BAB'; /*存储文章类型编码。*/
```

```
  v_TypeRemark VARCHAR2(30):='纺织工业';
BEGIN

  --块的执行部分
  INSERT INTO type
    VALUES(v_TypeCode,v_TypeRemark);
END;
```

其中，第一条单行注释中嵌套了多行注释，第二条单行注释中嵌套了单行注释。

第三章 PL/SQL 中的流控制语句

流控制语句是指具有一个入口和一个出口的语句块，用于控制PL/SQL程序的执行流向。合理地使用流控制语句，可以使程序具有良好的结构。本章将介绍下面三种流控制语句：

- 选择控制语句（IF语句）
- 循环控制语句（LOOP语句和EXIT语句）
- 顺序控制语句（GOTO语句和NULL语句）

3.1 选择控制语句

IF语句用于根据不同的情况来执行不同的语句，也就是说，根据条件的值来决定语句的执行顺序。共有三种格式的IF语句：

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

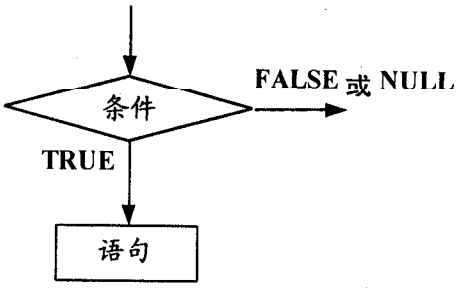
下面分别介绍这三种格式的IF语句。

3.1.1 IF...THEN

这是格式最简单的IF语句，其语法如下：

```
IF 条件 THEN
  语句;
END IF;
```

语法示意图如下：



由上面的语法可知，如果条件为TURE，则执行语句；如果条件为FALSE或NULL，则什么也不做。

例如，下面的语句用于查询作家王达琳的工资，并判断其工资是否为空，如果工资不为空，则显示该工资：

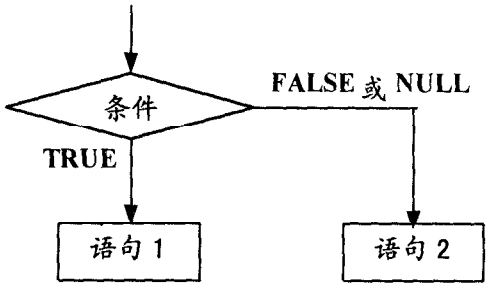
```
DECLARE
  v_Salary NUMBER(8,2);
BEGIN
  SELECT salary
    INTO v_Salary
    FROM auths
    WHERE name = '王达琳';
  IF v_Salary IS NOT NULL THEN
    DBMS_OUTPUT.PUT_LINE(v_Salary);
  END IF;
END;
```

3.1.2 IF…THEN…ELSE

这种格式的IF语句增加了关键字ELSE，格式如下：

```
IF 条件 THEN
  语句;
ELSE
  语句2;
END IF;
```

语法示意图如下：



由上面的语法可知，如果条件为TURE，则执行语句1；如果条件为FALSE或NULL，则执行语句2。

例如，下面的语句用于查询作家王达琳的工资，并判断其工资是否为空，如果工资不为空，则显示该工资；否则，显示提示信息：

```
DECLARE
  v_Salary NUMBER(8,2);
BEGIN
  SELECT salary
    INTO v_Salary
    FROM auths
    WHERE name = '王达琳';
  IF v_Salary IS NOT NULL THEN
    DBMS_OUTPUT.PUT_LINE(v_Salary);
  ELSE
    DBMS_OUTPUT.PUT_LINE('工资未定');
  END IF;
END;
```

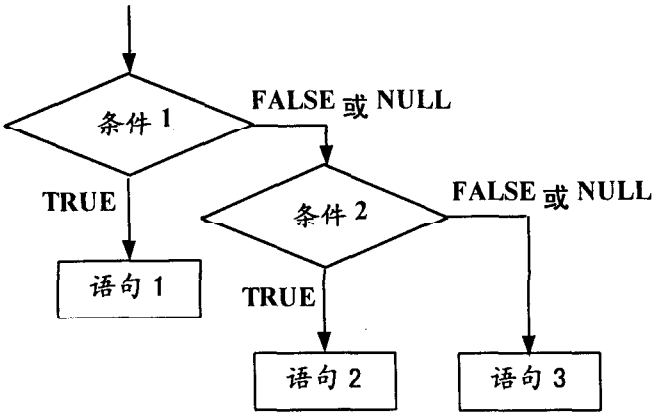
3.1.3 IF…THEN…ELSIF

这种格式的IF语句增加了关键字ELSIF，用于引入其它的条件。其语法如下：

```
IF 条件1 THEN
  语句1;
ELSIF 条件2 THEN
  语句2;
```

```
[ ELSE
  语句3;]
END IF;
```

语法示意图如下：



由上面的语法可知，当第一个条件的值为FALSE或NULL时，则检测ELSIF子句中第二个条件的值。一个IF语句可以由多个ELSIF子句组成，最后一个ELSE子句是可选的。在执行该格式的IF语句时，从上到下挨个检测其中的条件，如果某个条件的值为TRUE，则执行与之相关的语句；如果所有条件的值为FALSE或NULL，则执行ELSE子句。

例如，下面的 IF...THEN...ELSE 语句中有两个 ELSIF 和一个 ELSE 子句：

```
DECLARE
  v_Salary NUMBER(8,2);
  v_Comment varchar2(40);
BEGIN
  SELECT salary
    INTO v_Salary
    FROM auths
    WHERE author_code='A00009';
  IF v_Salary = 120 then
     v_Comment:='最低工资';
  ELSIF (v_Salary > 120) AND (v_Salary<=300) THEN
     v_Comment:='低工资';
  ELSIF (v_Salary>300) AND (v_Salary<=800) THEN
```

```
    v_Comment:='普通工资';
  ELSE
    v_Comment:='高工资';
  END IF;
END;
```

在上面的例子中，如果第一个条件 v_Salary = 120 的值为 TURE，则执行

```
v_Comment:='最低工资';
```

如果第一个条件的值为 FALSE，则检测第二个条件 (v_Salary>120)AND (v_Salary<=300)的值，当值为 TRUE 时，执行

```
v_Comment:='低工资';
```

如果第二个条件的值为 FALSE，则检测第三个条件 (v_Salary>300)AND (v_Salary<=800)的值，当值为 TRUE 时，执行

```
v_Comment:='普通工资';
```

当第三个条件也不满足时，执行

```
v_Comment:='高工资';
```

3.2 循环控制语句

LOOP 语句用于重复执行语句，共有三种格式的 LOOP 语句：

- LOOP
- WHILE...LOOP
- FOR...LOOP

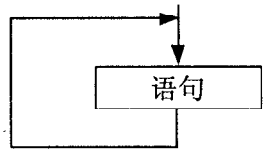
下面分别介绍这三种格式的LOOP语句。

3.2.1 LOOP

这是格式最简单的 **LOOP** 语句，其语法如下：

```
LOOP
  语句;
END LOOP;
```

语法示意图如下：



该语法用于无限制地循环执行语句，在实际应用中，这种语句没有意义。如果要退出循环，则必须使用 **EXIT** 语句来终止循环。这种格式的 **LOOP** 语句还可以带标号。下面分别介绍 **EXIT** 语句和带标号的 **LOOP** 语句。

3.2.1.1 EXIT 语句

EXIT 语句可以在 **LOOP** 语句中出现多次，但不可以在 **LOOP** 语句外使用。共有两种格式的 **EXIT** 语句：

- **EXIT**：该格式的语句用于无条件地强迫终止循环，当遇到一个 **EXIT** 语句时，则立即终止 **LOOP** 循环。
- **EXIT...WHEN**：该格式的语句用于根据条件的值来终止循环。当遇到 **EXIT** 语句时，首先检测 **WHEN** 子句中的条件，如果条件为 **TRUE**，则终止循环。

例1，下面是使用**EXIT**语句终止循环的例子：

```
DECLARE
  v_name VARCHAR2(10);
  v_num INTEGER:=1;
BEGIN
  LOOP
```

```
      SELECT name
        INTO v_name
        FROM auths
        WHERE author_code='A0000'||TO_CHAR(v_num);
      v_num:=v_num+1;
      IF v_num>=10 THEN
       EXIT;
      END IF;
    END LOOP;
  END;
```

上面的语句用于从auths表中检索代码小于“A00010”的作家姓名。

例2，下面是使用EXIT...WHEN语句终止循环的例子，该语句的功能同上面语句的功能相同：

```
  DECLARE
    v_name VARCHAR2(10);
    v_num INTEGER:=1;
  BEGIN
    LOOP

      --利用v_num的当前值检索表auths。
      SELECT name
        INTO v_name
        FROM auths
        WHERE author_code='A0000'||TO_CHAR(v_num);
      v_num:=v_num+1;

      --退出循环条件。当v_num>=10时，退出循环。
      EXIT WHEN v_num>=10;
    END LOOP;
  END;
```

3.2.1.2 带标号的循环语句

同 PL/SQL 块一样，LOOP 语句可以带标号。标号是指用“<<”和“>>”括起来的标识符，且必须放在 LOOP 语句的开始。可选地，也可在 LOOP 语句的结尾放标号。其格式如下：

```
<<标号名>>
LOOP
  语句;
END LOOP [标号名];
```

在嵌套的LOOP循环中，带标号的循环语句可以提高程序的可读性。同时，使用EXIT语句不仅可以终止当前的循环，还可以终止任意带标号的循环语句。

例如，在下面的语句中，通过使用标号first_loop，可以在内层循环中使用EXIT语句来终止外层的循环：

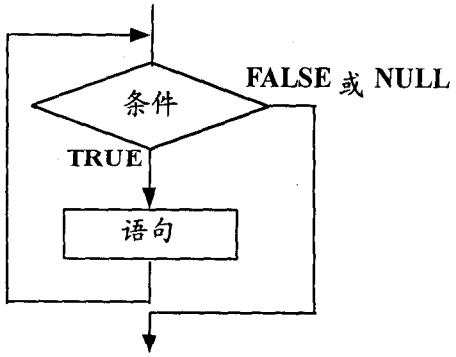
```
DECLARE
  ...
BEGIN
  <<first_loop>>
  LOOP
    LOOP
      ...
      EXIT first_loop WHEN ...;
    END LOOP;
  END LOOP first_loop;
END;
```

3.2.2 WHILE...LOOP

WHILE...LOOP语句根据条件执行循环语句。语法如下：

```
WHILE 条件 LOOP
  语句;
END LOOP;
```

语法示意图如下：



在执行语句前，首先检测条件的值。如果条件的值为TRUE，则执行语句；如果条件为FALSE或NULL，则退出循环。

例如，下面的语句用于从表auths中检索作家代码小于“A00010”的作家姓名：

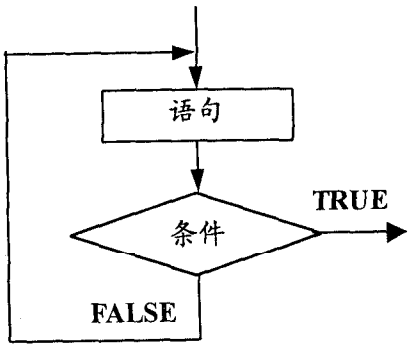
```
DECLARE
  v_name VARCHAR2(10);
  v_num INTEGER:=1;
BEGIN
  WHILE v_num<10 LOOP
    SELECT name
      INTO v_name
      FROM auths
      WHERE author_code='A0000'||TO_CHAR(v_num);
    v_num:=v_num+1;
  END LOOP;
END;
```

执行WHILE...LOOP语句时，如果语句中的条件为FALSE，则退出循环。由于是在执行循环语句前检测条件，所以，有可能不执行循环语句就退出循环。如果希望最少执行一次循环语句才退出循环，则可以使用如下的语法：

```
LOOP
  语句;
```

```
  EXIT WHEN 条件;
END LOOP;
```

语法示意图如下：



该语法保证循环中语句至少被执行一次，当满足WHEN子句中的条件时退出循环。

例如，下面的语句用于检索表auths中的作家姓名：

```
DECLARE
  v_name VARCHAR2(10);
  v_num INTEGER:=10;
BEGIN
  LOOP
    SELECT name
      INTO v_name
      FROM auths
      WHERE author_code='A000'||TO_CHAR(v_num);
    v_num:=v_num+1;
    EXIT WHEN v_num>10;
  END LOOP;
END;
```

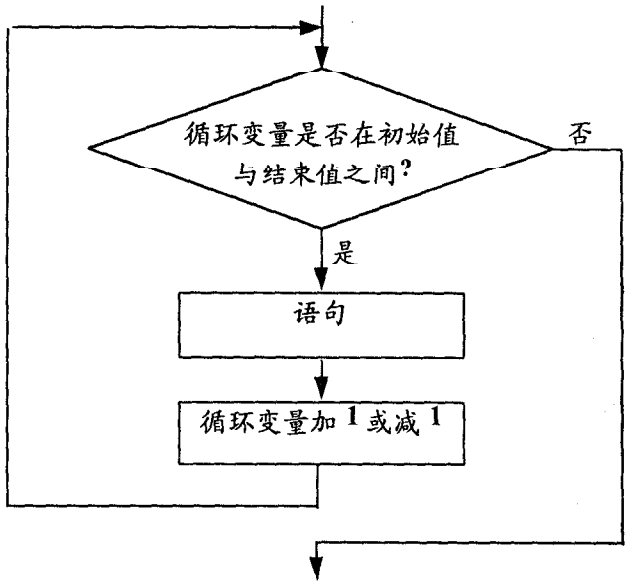
上面的语句首先执行循环语句，然后才判断是否满足循环条件。

3.2.3 FOR…LOOP

前面介绍的循环语句在执行前都不能预定循环的次数，如果希望在执行前预定循环次数，则可以使用**FOR…LOOP**循环语句。语法如下：

```
FOR 循环变量 IN [REVERSE] 初始值..结束值 LOOP
  语句;
END LOOP;
```

语法示意图如下：



由上面的语法可知，在执行 **FOR…LOOP** 语句时，每循环一次，循环变量自动加 **1** 或减 **1**，然后再判断循环变量的值是否在初始值和结束值之间，如果不在这两个值之间，则退出循环。

例如，下面是一个 **FOR…LOOP** 循环语句：

```
DECLARE
  v_name VARCHAR2(10);
BEGIN
  --执行 9 次循环语句，每执行一次，v_num 自动加 1
```

```
  FOR v_num IN 1..9 LOOP
    SELECT name
      INTO v_name
      FROM auths
      WHERE author_code='A0000'||TO_CHAR(v_num);
    DBMS_OUTPUT.PUT_LINE(v_name);
  END LOOP;
END;
```

上面的例子用于从表auths中检索作家代码从“A00001”到“A00009”的作家姓名。

在FOR...LOOP语句中，循环变量预定循环次数，如果希望提前退出循环，则必须使用EXIT语句。下面介绍循环变量和EXIT语句的使用。

3.2.3.1 循环变量

在执行FOR...LOOP语句时，循环变量被隐式地定义为INTEGER类型的局部变量，其值为初始值。所以，不必显式地定义循环变量。

缺省情况下，循环变量从初始值递增到结束值。但是，如果使用关键字REVERSE，则循环变量从结束值递减到初始值。但是，无论循环变量是递增还是递减，初始值都必须小于结束值，且增量（或减量）必须是1。

下面介绍循环变量的使用。

1. 在FOR...LOOP语句中，可以像引用常量一样引用循环变量。因此，循环变量可以出现表达式中，但不可以对其赋值。

   例如，下面的语句用于计算10、20...100的累加和：

```
DECLARE
  v_total NUMBER := 0 ;
BEGIN
  FOR v_num IN 1..100 LOOP
    IF MOD(v_num,10) = 0 THEN
      v_total := v_total + v_num;
```

```
    END IF;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(v_total);
END;
```

在上面的语句中，函数MOD()引用了循环变量v_num。

循环变量的初始值和结束值可以是数据、变量或表达式，但值必须是整数。

例如，下面是一些合法的初始值和结束值：

```
v_num IN 10..15
v_num IN REVERSE i*5..i*6
v_num IN 1..MOD(v_value/10)
```

例如：下面的语句用于计算2到20的累加和：

```
DECLARE
  v_total NUMBER := 0 ;
  v_value NUMBER :=2;
BEGIN
  FOR v_num IN v_value.. v_value *10 LOOP
    v_total := v_total+ v_num ;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(v_total);
END;
```

在上面的语句中，循环变量的初始值和结束值分别是变量v_value和表达式v_value*10。

3. PL/SQL允许在执行时动态指定循环变量的初始值和结束值。例如，下面的语句用于检索表auths中最后5位作家的姓名：

```
DECLARE
  v_name VARCHAR2(100);
  v_count NUMBER ;
BEGIN
```

```
    SELECT COUNT(name)
      INTO v_count
      FROM auths;
    FOR v_num IN v_count-5.. v_count LOOP
      SELECT name
        INTO v_name
        FROM auths
        WHERE author_code='A000'||TO_CHAR(v_num);
      DBMS  OUTPUT.PUT_LINE(v_name);
    END LOOP;
  END;
```

在上面的例子中，循环变量的初始值和结束是由变量v_count决定的，而变量v_count的值是SELECT语句的返回值。

4. 循环变量仅在循环语句中有效，当退出循环时，循环变量无效。例如，下面的语句用于计算从1到10的累加和：

```
DECLARE
  v_total NUMBER := 0 ;
BEGIN
  FOR v_num IN 1..10 LOOP
    v_total := v_total+ v_num ;
    DBMS_OUTPUT.PUT_LINE(v_num);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(v_total);
END;
```

上面的例子在循环语句中使用了过程DBMS_OUTPUT.PUT_LINE()，用于显示循环变量的值。如果在循环语句外使用引用循环变量v_num，将显示错误信息。例如，执行如下的语句：

```
DECLARE
  v_total NUMBER := 0 ;
BEGIN
  FOR v_num IN 1..10 LOOP
    v_total := v_total+ v_num ;
```

```
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(v_num);
  DBMS_OUTPUT.PUT_LINE(v_total);
END;
```

则会显示如下的错误信息：

```
DECLARE
*
ERROR at line 1:
ORA-06550: line 7, column 22:
PLS-00201: identifier 'V_NUM' must be declared
ORA-06550: line 7, column 1:
PL/SQL: Statement ignored
```

5. 因为循环变量被隐式地定义为局部变量，所以，任何同名的全局变量都将被该变量覆盖。如果要引用全局变量，则必须使用标号和点号。例如，下面的语句用于查询作家代码小于“A00010”的作家姓名：

```
<<global>>
DECLARE
  v_name VARCHAR2(10);
  v_num INTEGER := 8;
  v_count INTEGER ;
BEGIN
  SELECT COUNT(name)
    INTO v_count
    FROM auths;
  FOR v_num IN 1..v_count LOOP
    SELECT name
      INTO v_name
      FROM auths
      WHERE author_code='A0000'||TO_CHAR(v_num);
    DBMS_OUTPUT.PUT_LINE(v_name);
    IF v_num>global.v_num THEN
      EXIT;
    END IF;
```

```
  END LOOP;
END global;
```

上面循环语句中的变量v_num为循环变量，变量global.v_num为全局变量。

6. 如果在嵌套的循环语句中，内层和外层的循环变量名相同，这时，要在内部循环中使用外部循环的循环变量，必须使用标号和点号。例如，下面的语句：

```
DECLARE
  ...
BEGIN
  <<out_loop>>
  FOR v_num IN 1..100 LOOP
    ...
    FOR v_num IN 1..100 LOOP
      ...
      IF out_loop.v_num = v_num THEN
        ...
      ENDIF
    END LOOP;
  END LOOP;
END;
```

上面的语句在内层循环中引用外层循环的循环变量时，在循环变量v_num的前面加了标号out_loop和点号。

3.2.3.2 EXIT 语句

在FOR...LOOP循环语句中使用EXIT语句主要有两个用途：

- 提前退出当前循环
- 提前退出嵌套循环

例1，下面的循环语句预定循环10次，但是，如果FETCH语句不能返回记录，则不管循环了几次，都从退出循环：

```
DECLARE
  v_name VARCHAR2(10);
  v_num INTEGER:=1;
BEGIN
  LOOP
    SELECT name
      INTO v_name
      FROM auths
      WHERE author_code='A0000'||TO_CHAR(v_num);
    v_num:=v_num+1;
    IF v_num>=10 THEN
      GOTO print;
    END IF;
  END LOOP;
  << print >>
  DBMS_OUTPUT.PUT_LINE('检索到10条记录');
END;
```

例2，下面的语句在外层循环块前加标号，然后在内层循环块的EXIT语句中使用该标号，可以提前退出嵌套循环：

```
BEGIN
  ...
  <<first_loop>>
  FOR v_num1 IN 1..10 LOOP
    ...
    FOR v_num2 IN 2..10 LOOP
      ...
      EXIT first_loop WHEN ...
      ...
    END LOOP;
  END LOOP first_loop;
END;
```

3.3 顺序控制语句

和前面介绍的两种语句相比，GOTO语句和NULL语句在PL/SQL编程中并不重要。

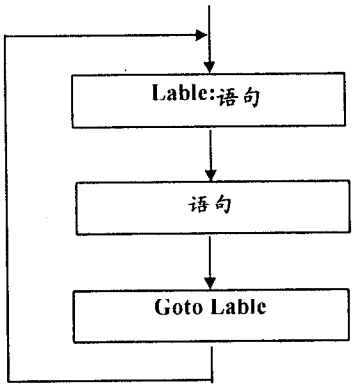
因为GOTO语句是非结构化语句，所以在PL/SQL程序中很少用到GOTO语句。NULL语句能够使条件语句的意义更加清晰，并且能够提高程序的可读性。

3.3.1 GOTO 语句

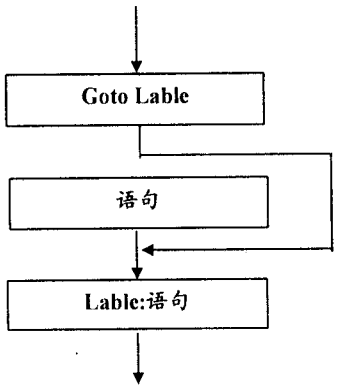
GOTO语句用于无条件地跳转到某个标号。在程序中，标号必须是唯一的，且标号后必须是一个可执行的语句或PL/SQL块。GOTO语句的语法如下：

GOTO label;

其语法示意图如下所示：



或为：



例 1，下面的 GOTO 语句用于向下跳转：

```
DECLARE
  v_name VARCHAR2(10);
  v_num INTEGER:=1;
BEGIN
  LOOP
    SELECT name
      INTO v_name
      FROM auths
      WHERE author_code='A0000'||TO_CHAR(v_num);
    v_num:=v_num+1;
    IF v_num>=10 THEN
      GOTO print;
    END IF;
  END LOOP;
  << print >>
  DBMS_OUTPUT.PUT_LINE('检索到10条记录');
END;
```

例2，下面的GOTO语句用于向上跳转：

```
DECLARE
  v_name VARCHAR2(10);
  v_num INTEGER:=1;
BEGIN
    << select_row>>
    SELECT name
      INTO v_name
      FROM auths
      WHERE author_code='A0000'||TO_CHAR(v_num);
    v_num:=v_num+1;
    IF v_num<10 THEN
      GOTO select_row;
    END IF;
  DBMS_OUTPUT.PUT_LINE('检索到10条记录');
END;
```

例3，下面语句中的标号<< print >>后面没有可执行的语句或语句块，所以该标号是无效的，该语句不合法：

```
DECLARE
  v_name VARCHAR2(10);
  v_num INTEGER:=1;
BEGIN
  LOOP
    SELECT name
      INTO v_name
      FROM auths
      WHERE author_code='A0000'||TO_CHAR(v_num);
    v_num:=v_num+1;
    IF v_num>=10 THEN
      GOTO print;
    END IF;
  END LOOP;
  << print >>
END;
```

3.3.1.1 GOTO 语句的限制

使用GOTO语句时，要受到如下的限制：

- 不能跳转到IF语句、LOOP语句或子块中
- 不能从子程序中跳出
- 不能从异常处理跳转到当前块

下面分别举例介绍。

例1，因为下面的GOTO语句跳转到了一个IF语句中，所以是不合法的：

```
BEGIN
  ...
  GOTO femal_auths;
  ...
```

```
  IF sex = 0 THEN
    << femal_auths >>
    UPDATE auths
    SET salary := salary + 100;
  END IF;
END;
```

例2，因为下面的GOTO语句跳转到了子块中，所以是不合法的：

```
BEGIN
  ...
  IF 条件 THEN
    GOTO select_name;
  END IF;
  ...
  BEGIN
    ...
    << select_name >>
    SELECT name
      FROM auths
      WHERE ...;
  END;
END;
```

例3，因为下面的GOTO语句从子程序中跳出，所以是不合法的：

```
DECLARE
  ...
PROCEDURE avg_salary IS
BEGIN
  ...
  GOTO delete_row;
END;
BEGIN
  ...
  <<delete_row>>
  DELETE FROM auths
```

```
    WHERE ...;
END;
```

例 4，因为下面的语句从异常处理部分跳转到可执行部分，所以是不合法的：

```
DECLARE
  v_name VARCHAR2(10);
  v_artcode varchar2(6);
BEGIN
  --检索auths表中的一行。
  SELECT name
    INTO v_name
    FROM auths
    WHERE author_code='A00075';
  <<l_select>>
  SELECT article_code
    INTO v_artcode
    FROM article
    WHERE author_code='A00075';
EXCEPTION
  WHEN NO_DATA_FOUND THEN
   GOTO l_select;
END;
```

3.3.2 NULL 语句

NULL语句用于显式地指定一个什么也不做的操作，相当于一个占位符。尽管NULL语句不执行任何操作，但是可以使某些语句变得有意义，同时可以提高程序的可读性。

例1，在前面的GOTO语句中已经介绍过，由于标号后面没有可执行的语句，下面的语句是不合法的：

```
DECLARE
  v_name VARCHAR2(10);
  v_num BINARY_INTEGER:=1;
BEGIN
  LOOP
```

```
    SELECT name
      INTO v_name
      FROM auths
      WHERE author_code='A0000'||TO_CHAR(v_num);
    v_num:=v_num+1;
    IF v_num>=10 THEN
      GOTO print;
    END IF;
  END LOOP;
  << print >>
END;
```

但是，如果在标号后面写入NULL语句，则该语句是合法的：

```
DECLARE
  v_name VARCHAR2(10);
  v_num BINARY_INTEGER:=1;
BEGIN
  LOOP
    SELECT name
      INTO v_name
      FROM auths
      WHERE author_code='A0000'||TO_CHAR(v_num);
    v_num:=v_num+1;
    IF v_num>=10 THEN
      GOTO print;
    END IF;
  END LOOP;
  << print >>
  NULL;
END;
```

例2，在下面的IF语句中使用了NULL语句，可以使IF语句中条件的意义更加清晰：

```
DECLARE
  v_salary NUMBER(8,2);
BEGIN
```

```
  SELECT salary
    INTO v_salary
    FROM auths
    WHERE author_code='A00008';
  IF v_salary IS NULL THEN
    UPDATE auths
      SET salary=600
      WHERE author_code='A00008';
  ELSIF v_salary<=500 THEN
    UPDATE auths
      SET salary=800
      WHERE author_code='A00008';
  ELSE
    NULL;  --不做任何处理。
  END IF;
END;
```

第四章 游标

环境区域是用来处理 SQL 语句的一个 Oracle 存储区域。游标是指向它的指针或句柄。通过游标，PL/SQL 程序可以控制这个环境区域中被处理的语句。Oracle 中的游标有两种：

- 显式游标
- 隐式游标

显式游标是用 CURSOR...IS 命令定义的游标，它可以对查询语句（SELECT）返回的多条记录进行处理。而隐式游标是在执行插入（INSERT）、删除（DELETE）、修改（UPDATE）和返回单条记录的查询（SELECT）语句时由 PL/SQL 自动定义的。

4.1 显式游标操作

显式游标在块定义部分、包或子程序（包和子程序将在以后章节中具体介绍）中声明。当声明了显式游标后，可以通过下面三条命令控制显式游标的操作：

1) 打开游标
2) 推进游标
3) 关闭游标

下面就详细介绍显式游标的声明和对显式游标的操作。

4.1.1 声明显式游标

声明显式游标就是指定游标名和与它关联的 SELECT 语句，其语法如下：

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

其中，Parameter子句的语法如下：

```
cursor_parameter_name [IN] datatype [{:= | DEFAULT} expr]
```

在上面的语法中 cursor_name 是游标名，该游标名必须符合 PL/SQL 标识符的命名规则。return_type 是游标返回的查询结果集类型，它只能是一个 PL/SQL 记录类型或数据库表的某一列的类型。SELECT_statement 是游标中的 SELECT 语句。在游标中指定的 SELECT 语句可以带有 UNION、MINUS 或 INTERSECT 子句。注意，SELECT_statement 不能含有 INTO 子句，查询结果是用 FETCH 语句中的 INTO 子句送给变量的。

例 1，下面的块定义部分中声明了一个显式游标：

```
DECLARE
  V_auths auths%ROWTYPE;
  v_code    auths.author_code%TYPE;
  CURSOR c_auths IS      --声明一个名为 c_auths 的游标
    SELECT * from auths
    Where   author_code=v_code;
```

注意，因为声明游标时将 PL/SQL 变量绑定在 WHERE 子句中，所以游标的声明必须在这些 PL/SQL 变量的作用域内，所以要先声明变量再声明显式游标。

例 2，上例是将 PL/SQL 变量绑定在游标的 WHERE 子句中，本例则是将游标参数绑定在游标的 WHERE 子句中：

```
DECLARE
  CURSOR c_Auths(P_code auths.author_code%TYPE) IS
    SELECT * FROM auths
      WHERE author_code = p_code;
```

注意，游标参数的作用域只局限在游标声明所指定的查询中，并且游标可以赋缺省值。

4.1.2 打开显式游标

当定义好显式游标后，就可以对该游标进行操作了。游标操作的第一步是打开游标。打开游标所做的处理是先查找绑定在游标中的变量，然后按照该变量的值确定查询结果集。

打开显式游标的语法如下：

OPEN cursor_name [(parameter[, parameter]...)];

其中 **cursor_name** 是已声明的游标的名，**parameter** 是绑定的实参。

例 1，下面的语句是打开上节例 1 中声明的显式游标 **c_Auths**：

```
BEGIN
  --在打开游标前为绑定变量赋值。
  v_Code:='A00001';

  --打开游标。
  OPEN c_Auths;
```

注意，在打开游标前，应先为绑定在游标中的变量（如果有该变量的话）赋值。如果打开游标后，再为该变量赋值，查询结果集不发生任何改变。这是因为结果集在游标打开时被确定，同时指向结果集的指针也被确定。

例2，如果对于一个带参数的游标，**OPEN**命令按照游标实参值确定查询结果集。下面的语句是打开上节例2中声明的带参数的显式游标 **c_Auths:**

```
BEGIN
--打开游标时将参数传入。
OPEN c_Auths('A00001');
```

打开一个已打开的游标也是合法的。当第二次打开游标时，**PL/SQL**先自动关闭游标，然后再打开。一次打开多个游标也是**PL/SQL**所允许的。

4.1.3 推进显式游标

当打开显式游标后，就可以使用 **FETCH** 语句来推进游标，返回查询结果集中的一行。每执行完一条 **FETCH** 语句后，显式游标会自动指向查询结果集的下一行。

FETCH 语句有下面两种形式：

FETCH cursor_name INTO list_of_variables;

或

FETCH cursor_name INTO PL/SQL_record;

其中 **cursor_name** 是已经声明的游标的名字，**list_of_variables** 是已经声明的变量列表，**PL/SQL_record** 是已经定义好的 **PL/SQL** 记录。在两种情况下，**INTO** 子句中的变量必须与游标 **SELECT** 语句中相应表列的类型兼容（一致或可以自动转换）。

例如，使用 **FETCH** 语句推进显式游标 **c_Auths**（**c_Auths** 已在“声明显式游标”、“打开显式游标”两节中声明并打开）：

--v_Auths已在声明游标c_Auths的同时被声明，具体看“声明显式游标”那节的例1。
FETCH c_Auths INTO v_Auths;

4.1.4 关闭显式游标

当整个结果集都检索完以后，应当关闭游标。关闭游标用来通知 **PL/SQL** 游标操作已经结束，并且释放游标所占用的资源（结果集所使用的资源空间）。

关闭游标的语法如下：

CLOSE cursor_name;

其中 **cursor_name** 为已经打开的游标名。

游标处于关闭状态时，推进游标是错误的，错误信息如下：

ORA-1001: Invalid Cursor

或

ORA-1002: Fetch out of Sequence

同样，如果关闭游标后又执行了关闭游标的命令，还会提示 **ORA-1001** 错误。
前面是对游标的基本操作，下面讲解游标在数据处理中的应用。

4.2 游标的属性

游标有四个属性，它们分别是%FOUND、%NOTFOUND、%ISOPEN 和%ROWCOUNT。这些属性将返回游标操作的一些有用信息。但要注意这些属性只能使用在过程性语句中，而不能使用在 SQL 语句中。下面就详细介绍这些属性。

1. %FOUND

%FOUND 是一个返回布尔值的属性，如果 FETCH 语句返回了一行，则该属性返回 TRUE，否则返回 FALSE。如果在打开游标之前或关闭游标之后使用该属性，会提示 ORA-1001(不可用的游标)错误。当打开游标后没有使用 FETCH 语句推进，则返回 NULL，因为这时属性值无法确定。该属性通常用在循环结构中。

2. %NOTFOUND

%NOTFOUND 与%FOUND 意思相反，如果 FETCH 语句返回了一行，则该属性返回 FALSE，否则返回 TRUE。如果在打开游标之前或关闭游标之后使用该属性，会提示 ORA-1001(不可用的游标)错误。当打开游标后没有使用 FETCH 语句推进，则返回 NULL，因为这时属性值无法确定。该属性通常用在 FETCH 循环结构中。

3. %ISOPEN

该属性用来确定相关的游标是否打开。如果游标已经打开，则%ISOPEN 返回 TRUE；否则返回 FALSE。

4. %ROWCOUNT

返回游标推进的行数。如果游标打开之前或关闭游标之后使用该属性，则产生 ORA-1001 错误。

下面举例介绍在游标操作过程的不同位置使用上述四个属性所返回的信息。

首先创建一个数据库表 TableAttribute，表中有两列 column1（NUMBER 类型）和 column2（VARCHAR2 类型），现在向表中插入两条记录：

```
INSERT INTO TableAttribute VALUES(10,'first');
```

```
INSERT INTO TableAttribute VALUES(20,'second');
```

有下面一个块：

```
DECLARE
    --声明游标
    CURSOR c_Attribute IS
      SELECT * FROM TableAttribute;
    --将从游标中取到的行放到v_AttributeRecord记录中。
    v_AttributeRecord c_Attribute%ROWTYPE;
BEGIN
  --位置1
  OPEN c_Attribute;
  --位置2
  FETCH c_Attribute INTO v_AttributeRecord;
  --位置3
  FETCH c_Attribute INTO v_AttributeRecord;
  --位置4
  FETCH c_Attribute INTO v_AttributeRecord;
  --位置5
  CLOSE c_Attribute;
  --位置6
END;
```

分别在上面块结构中六个位置书写 c_Attribut%FOUND、c_Attribut%OTFOUND、c_ Attribut%ISOPEN 或 c_ Attribut%ROWCOUNT，返回值如下表所示：

位置	%FOUND 的返回值	%NOTFOUND 的返回值	%ISOPEN 的返回值	%ROWCOUNT 的返回值
1	错误：ORA-1001	错误：ORA-1001	FALSE	错误：ORA-1001
2	NULL	NULL	TRUE	0
3	TRUE	FALSE	TRUE	1
4	TRUE	TRUE	TRUE	2
5	FALSE	TRUE	TRUE	2
6	错误：ORA-1001	错误：ORA-1001	FALSE	错误：ORA-1001

4.3 显式游标的推进循环

显式游标的操作是推进结果集内的行。我们可以使用 LOOP...END LOOP 语句、WHILE...LOOP 语句执行显式游标的推进循环。

例如，定义显式游标 c_salary，通过使用 LOOP...END LOOP 循环来逐条推进游标的结果集，通过判断游标的结果集来修改 auths 表的 salary 列：

```
DECLARE
  -- 声明一个变量，这个变量用来接收游标返回的结果集。
  v_Salary   Auths.salary%TYPE;
  v_Code   Auths.author_code%TYPE;

  /* 声明游标，该游标的查询结果集是作家代码为
  "A00001"到"A00006"的工资值。*/
  CURSOR c_salary IS
    SELECT salary,author_code
      FROM auths
      WHERE author_code<='A00006';
BEGIN
  -- 打开游标，并初始化结果集
  OPEN c_salary;
  LOOP
    -- 推进游标，将游标的查询结果集中的一行存到变量v_salary中。
    FETCH c_salary INTO v_Salary, v_Code;

    -- 当结果集中没有行时退出循环。
    EXIT WHEN c_salary%NOTFOUND;

    -- 如果查询到的作家工资小于或等于200，则增加该作家的工资值。
    IF v_salary<=200 THEN
       UPDATE auths SET salary=salary+50
        WHERE author_code = v_Code;
    END IF;
  END LOOP;
```

```
  -- 关闭游标，释放游标占用的资源。
  CLOSE c_salary;

  -- 提交所做的修改。
  COMMIT;
END;
```

注意，EXIT WHEN 语句在 FETCH 语句之后，这样在最后一行被检索后，c_Auths%NOTFOUND 变为 TRUE，循环结束。EXIT WHEN 语句同时放在数据处理语句前，这样做能确保不会重复处理同一行（检索到的最后一行）。

上面的游标循环也可以使用 WHILE...LOOP 语句来完成，这里就不再举例了。

从上面的例子中可以看出，无论是使用 LOOP...END LOOP 语句还是使用 WHILE...LOOP 语句来完成游标的推进循环，都必须使用 OPEN、FETCH 和 CLOSE 语句来控制游标的打开、推进和关闭。PL/SQL 还提供了一种简单类型的循环，可以自动控制游标的打开、推进和关闭，而不必再使用 OPEN、FETCH 和 CLOSE 语句。这叫做游标的 FOR 循环。

例如，使用游标的 FOR 循环完成与前面的例子相同的功能：

```
DECLARE
  CURSOR c_salary IS
    SELECT salary
      FROM auths
      WHERE author_code<='A00006';

BEGIN
  -- 开始游标FOR循环，隐含地打开c_salary 游标。
  FOR v_salary IN c_salary LOOP
   -- 一个隐含的FETCH语句在这里被执行。

    IF v_salary.salary<=200 THEN
      UPDATE auths SET salary=salary+50 WHERE salary=v_salary.salary;
    END IF;
    -- 在循环继续前，一个隐含的c_Auths%NOTFOUND被检测。
  END LOOP;
```

```
  -- 现在循环已经结束，c_Auths游标的一个隐含的CLOSE操作被执行。
  COMMIT;
END;
```

上例中 v_salary 没有在块的定义部分声明，该变量被 PL/SQL 编译器隐含地声明了，该变量的类型为 c_salary%ROWTYPE, 其作用域只在循环内部。而且在循环开始，游标 c_salary 被自动打开。在每一次自动推进后，%FOUND 属性用来检查在结果集中是否还有行。当结果集中没有行时，游标被自动关闭。

上面例子的 UPDATE 语句中还可以使用 CURRENT OF cursor 子句作为条件，仅用来改变该作家的工资，其中 cursor 是游标名。该子句指定游标结果集当前行，但需要注意的是，在 UPDATE 语句中使用 CURRENT OF cursor 子句时，必须先在声明游标的 SELECT 语句中有 FOR UPDATE OF 语句：

```
DECLARE
  --声明游标时在SELECT语句中必须加FOR UPDATE OF子句。
  CURSOR c_salary IS
    SELECT salary
      FROM auths
      WHERE author_code<='A00006' FOR UPDATE OF salary;
BEGIN
  FOR v_salary IN c_salary LOOP
    IF v_salary.salary<=200 THEN
      --下面的UPDATE语句中的CURRENT OF子句用来表明结果集的当前行。
      UPDATE auths SET salary=salary+50 WHERE CURRENT OF c_salary;
    END IF;
  END LOOP;
  COMMIT;
END;
```

如果我们在游标的 FOR 循环中使用子查询，则不用在块定义部分声明显式游标，在 FOR 循环中的子查询隐含声明了一个显式游标。

例如，在下面的块中，使用子查询的 FOR 循环来完成上例的功能：

```
BEGIN
  --在下面的 FOR 循环中隐含地声明了一个游标 c_salary。
```

```
    FOR c_salary IN
    (SELECT salary FROM auths WHERE author_code<='A00006' ) LOOP
      IF c_salary.salary<=200 THEN
        UPDATE auths SET salary=salary+50 where salary=c_salary.salary;
      END IF;
    END LOOP;
    COMMIT;
  END;
```

4.4 隐式游标处理

显式游标仅仅是用来控制返回多行的 SELECT 语句，而隐式游标是指向处理所有的 SQL 语句的环境区域的指针，隐式游标也叫 SQL 游标。与显式游标不同的是，SQL 游标不能通过专门的命令打开或关闭。PL/SQL 隐式地打开 SQL 游标，并在它内部处理 SQL 语句，然后关闭它。

SQL 游标用来处理 INSERT、UPDATE、DELETE 以及返回一行的 SELECT...INTO 语句。一个 SQL 游标不管是打开还是关闭，OPEN、FETCH 和 CLOSE 命令都不能操作它。

SQL 游标与显式游标类似，也有%FOUND、%NOTFOUND、%ISOPEN 和%ROWCOUNT 属性。SQL 游标的属性通常是返回执行 INSERT、DELETE、UPDATE 或 SELECT...INTO 语句时的信息。当打开 SQL 游标之前，SQL 游标的属性都为 NULL。

下面我们来具体介绍 SQL 游标的属性。

1. %FOUND

当使用 INSERT、DELETE 或 UPDATE 语句处理一行或多行，或执行 SELECT INTO 语句返回一行时，%FOUND 属性返回 TRUE，否则返回 FALSE。

注意，如果执行 SELECT INTO 语句时返回多行，则会产生 TOO_MANY_ROWS 异常，并将控制权转到异常处理部分（关于“异常”的具体介绍请看“异常处理”一章），%FOUND 属性并不返回 TRUE。如果执行 SELECT INTO 语句时返回 0 行，会产生 NO_DATA_FOUND 异常，%FOUND 属性并不返回 FALSE。

2. %NOTFOUND

%NOTFOUND 恰与%FOUND 属性相反，当使用 INSERT、DELETE 或 UPDATE 语句处理的行数为 0 时，%NOTFOUND 属性返回 TRUE，否则返回 FALSE。

3. %ISOPEN

因为在执行了 DML 语句后，Oracle 会自动关闭 SQL 游标，所以%ISOPEN 总为 FALSE。

4. %ROWCOUNT

该属性返回执行 INSERT、DELETE 或 UPDATE 语句返回的行数，或返回执行 SELECT INTO 语句时查询出的行数。如果 INSERT、DELETE、UPDATE 或 SELECT INTO 语句返回的行数为 0，则%ROWCOUNT 属性返回 0。但如果 SELECT INTO 语句返回的行数为多行，则产生 TOO_MANY_ROWS 异常，并将控制权转到异常处理部分，而不是去判断%ROWCOUNT 属性。

例如，在下面的块中使用了%NOTFOUND 属性：

```
BEGIN
  UPDATE auths
    SET entry_date_time = SYSDATE
    WHERE author_code = 'A00017';
  --如果UPDATE语句中修改的行不存在（SQL%NOTFOUND返回值为TRUE），
  --则向auths表中插入一行。
  IF SQL%NOTFOUND THEN
    INSERT INTO auths(author_code, name, sex, birthdate, salary,
                      entry_date_time)
      VALUES('A00017','赵浩',1, '30-APR-40',98.5, SYSDATE);
  END IF;
END;
```

我们使用 SQL%ROWCOUNT 完成与上例相同的功能：

```
BEGIN
  UPDATE auths
    SET entry_date_time = SYSDATE
```

```
    WHERE author_code = 'A00017';
  --如果UPDATE语句中修改的行不存在（SQL%ROWCOUNT=0），
  --则向auths表中插入一行。
  IF SQL%ROWCOUNT=0 THEN
    INSERT INTO auths(author_code, name, sex, birthdate, salary,
                      entry_date_time)
      VALUES('A00017','赵浩',1, '30-APR-40',98.5, SYSDATE);
  END IF;
END;
```

执行 SELECT INTO 语句返回多行或 0 行时，都会产生异常。

例如，下面的块中查询 auths 表中名字为“王达琳”的作家，如果查找到一条记录则删除它；如果没有查找到相应记录，则产生 NO_DATA_FOUND 异常，并转到异常处理部分去执行相应语句，而不判断%FOUND 属性值；如果查找到的记录不止一条，则产生 TOO_MANY_ROWS 异常，并且也转到异常处理部分去执行相应语句：

```
SET SERVEROUTPUT ON
DECLARE
  v_birthdate DATE;
BEGIN
  SELECT birthdate INTO v_birthdate FROM auths
    WHERE name='王达琳';

  --如果查询到一条记录，则删除该记录。
  IF SQL%FOUND THEN
    DELETE FROM auths where name='王达琳';
  END IF;

EXCEPTION
  --如果没有查找到该记录，则返回相应信息。
  WHEN NO_DATA_FOUND THEN
   DBMS_OUTPUT.PUT_LINE('该记录不存在');

  --如果查找到的记录不止一条，则表明有同名作家。
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('存在同名的作家');
```

```
END;
```

4.5 游标变量

到目前为止前面所有显式游标的例子都是静态游标——即游标与一个 SQL 语句关联，并且该 SQL 语句在编译时已经确定。而游标变量是一个引用类型（REF）的变量，它与 C 或 Pascal 中的指针类似。当程序运行时使用游标变量可以指定不同的查询，所以游标变量的使用比静态游标更灵活。游标变量的打开、推进以及关闭类似于静态游标。

4.5.1 游标变量的声明

使用游标变量前必须先定义一个游标变量类型，定义游标变量类型的语法如下所示：

```
TYPE type_name IS REF CURSOR RETURN return_type;
```

其中 type_name 是游标变量类型名，return_type 是一个记录类型，这个记录类型指定了游标变量最终返回的查询结果集类型。这个记录类型可以是一个用户自定义的记录类型，或使用%ROWTYPE 定义的记录类型。一旦游标变量类型被定义，就可以声明这种类型的变量了。

例如，下面的定义部分定义了两个游标变量类型，其结果集分别是自定义记录类型和%ROWTYPE 定义的记录类型：

```
DECLARE
  --使用%ROWTYPE定义一个游标变量类型。
  TYPE t_AuthsRef IS REF CURSOR
    RETURN auths%ROWTYPE;

  --定义一个记录类型。
  TYPE t_CodeRecord IS RECORD(
    author_code article.author_code%TYPE,
    article_code article.article_code%TYPE);

  --声明一个记录类型的变量。
  v_Code t_CodeRecord;
```

```
--使用t_CodeRecord作为一个游标变量类型的结果集类型。
TYPE t_CodeRef IS REF CURSOR
  RETURN t_CodeRecord;

-- 使用v_Code作为一个游标变量类型的结果集类型。
TYPE t_CodeRef2 IS REF CURSOR
  RETURN v_Code%TYPE;

-- 使用上面的类型声明两个游标变量。
v_authCV t_authsRef;
v_CodeCV t_CodeRef;
```

前面讲的游标变量指定了结果集的类型，所以当打开该变量时，必须指定一个与游标返回的类型匹配的查询。否则，会产生一个预定义异常 **ROWTYPE_MISMATCH**。

还有一种游标变量类型，它不用指定结果集类型（在定义游标变量类型时不用指定 **RETURN** 子句）。这种游标变量只能在 **PL/SQL2.3** 以上版本中使用。我们可以使用一个没有指定结果集类型的游标变量来指定多个不同类型的查询。

例如，下面的定义部分声明了一个没有指定结果集类型的游标变量：

```
DECLARE
  --定义一个没有指定结果集类型的游标变量类型。
  TYPE t_AuthsRef IS REF CURSOR;

  --声明一个该类型的变量。
 v_CursorVar t_AuthsRef;
```

4.5.2 打开游标变量

为了将一个游标变量与一个具体的 **SELECT** 语句联系起来，**OPEN** 的语法中增加了一个 **SELECT** 语句。下面是打开一个游标变量的语法：

```
OPEN cursor_variable FOR select_statement;
```

其中 **cursor_variable** 是一个已经声明了的游标变量的名字，**select_statement** 是所要求的查询。如果游标变量是指定了结果集类型的，那么查询列表必须与游标的返回类型匹

配。

注意，打开游标变量时指定的 SELECT 语句中不能使用 FOR UPDATE 子句，所以在 DML 语句中也不能使用 WHERE CURRENT OF 子句，这与静态游标不同。

例如，首先声明一个游标变量：

```
DECLARE
  TYPE t_AuthorsRef IS REF CURSOR RETURN auths%ROWTYPE;
  v_AuthsCV t_AuthorsRef;
```

然后用下面的语句打开 v_AuthsCV：

```
OPEN v_AuthsCV FOR
   SELECT * FROM auths;
```

我们可以试着用下面的方法打开 v_AuthsCV：

```
OPEN v_AuthsCV FOR
   SELECT name, salary FROM auths;
```

将会产生 ORA-6504 错误，因为查询中的选择列表与游标变量的结果集类型不匹配。

我们可以为不同的查询多次打开一个游标变量，在重新打开这个游标变量之前不必先关闭它，这一点与静态游标不同。当我们为一个不同的查询而再次打开该游标时，前面的查询就丢失了。

4.5.3 推进游标变量

当声明了游标变量后，就可以使用 FETCH 语句推进游标变量了。使用 FETCH 语句推进游标变量的语法与推进静态游标的语法一样，这里就不多介绍了。

在推进游标时要注意，由于一个游标变量可以多次打开，所以推进游标时用来接收查询结果集的 PL/SQL 变量类型一定要与结果集类型一致。

4.5.4 关闭游标变量

游标变量的关闭与静态游标的关闭一样，也是使用 CLOSE 语句。该操作用来释放查询所占用的资源。但没有释放游标变量占用的存储空间。当变量超出作用域的时候，它所占用的空间才被释放掉。

例如，在下面的块中定义了一个没有指定结果集的游标变量，这样我们就可以使用这个游标变量指向不同的查询，并能够返回不同的记录类型：

```
set serveroutput on size 100000   --设置存储缓冲区大小。
DECLARE

  /*定义游标变量类型 t_CurRef，该游标变量类型没有指定结果集类型，
    所以该游标变量类型的变量可以返回不同的 PL/SQL 记录类型。*/
  TYPE t_CurRef IS REF CURSOR;

  --声明一个游标变量类型的变量。
  c_CursorRef t_curref;

 --定义 PL/SQL 记录类型 t_AuthorRec，该类型的变量用来接收游标变量的返回值。
  TYPE t_AuthorRec IS RECORD(
    AuthorCode auths.author_code%TYPE,
    Name auths.name%TYPE);

  --定义 PL/SQL 记录类型 t_ArticleRec，
  --该类型的变量也用来接收游标变量的返回值。
  TYPE t_ArticleRec IS RECORD(
    AuthorCode article.author_code%TYPE,
    Title article.Title%TYPE);

  --声明两个记录类型变量。
  v_Author t_AuthorRec;
  v_Article t_ArticleRec;

BEGIN
```

```
--打开游标变量 c_CursorRef，返回 t_AuthorRec 类型的记录。
OPEN c_CursorRef FOR
SELECT author_code,name FROM auths
      WHERE author_code IN ('A00001','A00002','A00003','A00004','A00005');

--推进游标变量。
FETCH c_CursorRef INTO v_Author;

--游标变量的推进循环。
WHILE c_CursorRef%found LOOP
   --将作家代码和相应的作家名字输出到屏幕上。
   DBMS_OUTPUT.PUT(v_Author.AuthorCode||':'||v_Author.Name||'    ');
   FETCH c_CursorRef INTO v_Author;
END LOOP;

DBMS_OUTPUT.NEW_LINE;  --向屏幕上输出一个回车行。

--关闭游标变量，仅仅将游标变量指定的资源释放掉，
--游标变量本身的存储空间没有释放掉。
CLOSE c_CursorRef;

--再次打开游标变量，返回 t_ArticleRec 类型的记录。
OPEN c_CursorRef FOR
      SELECT author_code,title FROM article
      WHERE author_code IN ('A00001','A00002','A00003','A00004','A00005');

  FETCH c_CursorRef INTO v_Article;

  WHILE c_CursorRef%found LOOP
    --将作家代码和他的文章输出到屏幕上。
    DBMS_OUTPUT.PUT_LINE(v_Article.AUthorCode||':'||v_Article.Title);
    FETCH c_CursorRef INTO v_Article;
  END LOOP;

  CLOSE c_CUrsorRef;
END;
```

该例的执行结果如下所：

A00005:张启迪　**A00004:**许灵延　**A00003:**赵封昭　**A00002:**李井元　**A00001:**王达琳
A00001:Intermet 将成为多语系网
A00003:PowerPC AS 芯片将用于 **RS/6000** 系统
A00004:VESA 标准获得通过
A00005:CA 推出两种网络新软件
A00002:

注意，在上例中，第一次关闭游标变量（**CLOSE c_CursorRef**）是可以省略的，因为在第二次打开游标变量时，就将第一次的查询丢失掉了。而且游标变量也有游标属性，通常在推进游标变量时使用这些游标属性，例如上例使用了**%FOUND** 属性。

第五章 存储过程与函数

PL/SQL 中的存储过程和函数是用来完成特定功能的子程序，与 C 语言的函数与过程类似，都是由用户定义的。

5.1 创建存储过程

存储过程的创建包括参数的定义与过程体的定义两部分。

5.1.1 创建存储过程的语法

下面是创建存储过程的语法：

```
CREATE [OR REPLACE] PROCEDURE [schema.] procedure_name
    [(argument [{ IN | OUT | IN OUT }] datatype [,...] )]
    { IS | AS }
    pl/sql_body;
```

其中，procedure_name 是存储过程的名称，argument 是参数名，datatype 是对应参数的数据类型。pl/sql_body 是该存储过程真正进行处理操作的 PL/SQL 块。OR REPLACE 是可选项，如果已经存在一个同名的过程，则首先删除已有过程，然后创建。关键字 IS 和 AS 是等价的，用来引出过程体。

1. 存储过程的参数

参数在存储过程的定义中较复杂，下面我们首先给出两个概念——形式参数和实际参数。例如，有如下一个存储过程，该过程接收一个作家代码和一个工资值，将该作家的工资改为接收到的工资值：

```
CREATE OR REPLACE PROCEDURE UpdateAuths(
    p_AuthsCode auths.author_code%TYPE,
    p_AuthsSalary   auths.salary%TYPE)
AS
```

```
BEGIN

  UPDATE auths
  SET salary = p_AuthsSalary
  WHERE author_code = p_AuthsCode;

  COMMIT;
END  UpdateAuths;
```

下面的 PL/SQL 块调用 UpdateAuths 存储过程，将代码为 A00011 的作家的工资改为 350 元：

```
DECLARE
    v_authorcode   auths.author_code%TYPE:='A00011';
    v_salary   auths.salary%type := 350;
BEGIN
    updateauths(v_authorcode,v_salary);
END;
```

上面块中声明的变量 v_authorcode 和 v_salary 被作为参数传递到存储过程 UpdateAuths 中，这些参数就是实际参数，简称实参。而在存储过程中声明的参数 p_AuthsCode、p_AuthsSalary 是形式参数，简称形参。

在参数的定义中，IN、OUT 和 IN OUT 关键字代表参数的三种不同模式，对其说明如下：

- IN：当调用存储过程的时候，该模式的形参接收对应实参的值，并且该形参是只读的，即不能被修改。如果在创建存储过程时没有指定参数的模式，则默认为 IN。
- OUT：在存储过程中，该形参被认为是只能写，既只能为其赋值。在存储过程中不能读它的值。返回时，将该形参值传给相应的实参。
- IN OUT：该模式是前两种模式的合并。

例如，下面是一个含有三种参数模式的存储过程：

```
CREATE OR REPLACE PROCEDURE UpdateAuthsSalary (
  p_Author_code  IN OUT auths.author_code%TYPE,
  p_Salary  IN NUMBER,
```

```
  p_Name OUT auths.name%TYPE ) IS
  v_Salary_temp  NUMBER; --定义存储过程中的局部变量
BEGIN
  SELECT salary INTO v_Salary_temp
    FROM AUTHS
    WHERE author_code = p_Author_code;

  /* 如果该作家的工资小于300元，则修改其工资
  IF v_Salary_temp < 300 THEN
    UPDATE auths
      SET salary = p_Salary
      WHERE author_code = p_Author_code;
  END IF;

  SELECT name
    INTO p_name
    FROM auths
    WHERE author_code = p_Author_code;
END UpdateAuthsSalary;
```

在创建存储过程时，使用上述三种模式应注意以下几点：

- 存储过程中的 IN 参数只能是 rvalue（赋值语句的左值），OUT 参数只能是 lvalue（赋值语句的右值），IN OUT 参数既可以是 rvalue 也可以是 lvalue。
- 当存储过程中的形参是 OUT 或 IN OUT 模式时，则相应的实参必须是 lvalue。
- 在存储过程中的 OUT 和 IN OUT 形参可以作为存储过程中 SELECT...INTO 或 FETCH...INTO 语句中 INTO 子句内的变量使用。

2. 存储过程的处理部分

存储过程的处理部分是一个 PL/SQL 块，该块包含定义部分、可执行部分以及异常处理部分，其中可执行部分是必须有的。定义部分位于 IS 或 AS 与 BEGIN 之间，用来声明存储过程体中要用到的变量。可执行部分位于 BEGIN 和 EXCEPTION 关键字之间，是完成存储过程功能的部分。异常处理部分位于 EXCEPTION 和 END 关键字之间，用来处理块中出现的错误。各部分的位置如下图所示：

```
CREATE OR REPLACE 过程名
 (
  …
 )
 IS
  …
 BEGIN
  …
 EXCEPTION
  …
END 过程名;
```

参数定义部分

局部变量定义部分

可执行部分

异常处理部分

在 END 后可以加上存储过程名，也可以不加。如果要在 END 后加名字，则必须要和前面的存储过程名相同。在 END 后加存储过程名只是为了增加程序的可读性。

5.1.2 存储过程的参数

在存储过程的定义和调用时，参数都是比较重要的部分。下面我们讨论有关参数的问题。

1. 参数的数据类型

在定义一个存储过程参数时，不能指定 CHAR 类型和 VARCHAR2 类型形参的长度，也不能指定 NUMBER 形参的精度和标度。这些约束由实参来传递。

例如，下面的存储过程定义是不合法的，将产生一个编译错误：

```
CREATE OR REPLACE PROCEDURE proc_auths(
    --参数定义了类型长度，这将产生编译错误。
    p_code   IN OUT VARCHAR2(6),
    p_salary   OUT NUMBER(8,2)) AS
BEGIN
    SELECT salary INTO p_salary
    FROM auths
    WHERE author_code = p_code;
END proc_auths;
```

修改上面存储过程的定义为：

```
CREATE OR REPLACE PROCEDURE proc_auths(
   --去掉了 VARCHAR2 类型的长度，去掉了 NUMBER 类型的精度和标度。
    p_code   IN OUT VARCHAR2,
    p_salary   OUT NUMBER) AS
BEGIN
    SELECT salary INTO p_salary
    FROM auths
    WHERE author_code = p_code;
END proc_auths;
```

p_code 的长度约束和 p_salary 的精度、标度约束由实参传递。例如，我们调用 proc_auths：

```
DECLARE
  v_code VARCHAR2(6);
  v_salary   NUMBER(8,2);
BEGIN
  v_code := 'A00001';
  proc_auths(v_code,v_salary); -- 查询代码为A00001的作家的工资。
END;
```

上面的存储过程调用将实参的值和约束都传递给了形参，所以 p_code 的最大长度是 6，p_salary 的精度是 8，标度是 2。

注意，如果使用%TYPE 为参数定义类型，那么该参数将具有定义在形参上而不是通过实参传递的数据长度。例如，按下面的方式声明：

```
CREATE OR REPLACE PROCEDURE query_salary(
    p_code IN OUT auths.author_code%TYPE,
    p_salary   OUT auths.salary%TYPE)   AS
BEGIN
    SELECT salary INTO p_salary
    FROM auths
    WHERE author_code = p_code;
```

```
END query_salary;
```

如上面的存储过程，由于作家表中的作家代码author_code的长度为6，因此p_code的长度也为6。

2. 参数的传值方式

在调用存储过程时，有下面两种参数传值方式：

- 位置表示法
- 名称表示法

如果实参通过位置与形参进行联系，就叫作位置表示法；如果实参是与形参的名称进行联系，就叫做名称表示法。通常我们使用的是位置表示法。

例如，定义一个存储过程：

```
CREATE OR REPLACE PROCEDURE insert_auths (
  p_code auths.author_code%TYPE,
  p_name auths.name%TYPE,
  p_sex auths.sex%TYPE,
  p_birthdate auths.birthdate%TYPE) AS
BEGIN
  INSERT INTO auths (author_code, name, sex, birthdate, entry_date_time)
    VALUES(p_code, p_name, p_sex, p_birthdate, SYSDATE);
END insert_auths;
```

使用位置表示法调用上面定义的存储过程：

```
DECLARE
  v_code VARCHAR2(6);
  v_name VARCHAR2(12);
  v_sex NUMBER(1);
  v_birthdate DATE;
BEGIN
  v_code := 'A00021';
  v_name := '张志强';
```

```
  v_sex := 1;
  v_birthdate := '5-seq-70';

  -- 实参的位置顺序与形参的位置顺序相对应。
  insert_auths (v_code, v_name, v_sex, v_birthdate);
END;
```

使用命名表示法来调用上面的存储过程：

```
DECLARE
  v_code VARCHAR2(6);
  v_name VARCHAR2(12);
  v_sex NUMBER(1);
  v_birthdate DATE;
BEGIN
  v_code := 'A00021';
  v_name := '张志强';
  v_sex := 1;
  v_birthdate := '5-seq-70';

  -- 实参名和形参名对应，这样就可以重新排列参数的先后顺序。
  insert_auths (p_name => v_name,
                p_sex => v_sex,
                p_code => v_code,
                p_birthdate => v_birthdate);

END;
```

位置表示法和命名表示法在一些调用中也可以混合使用。但是，当在调用存储过程中出现了第一个命名表示法的参数时，后面的参数也必须使用命名表示法传值。

例如：

```
DECLARE
  v_code VARCHAR2(6);
  v_name VARCHAR2(12);
  v_sex NUMBER(1);
```

```
  v_birthdate DATE;
BEGIN
  v_code := 'A00021';
  v_name := '张志强';
  v_sex := 1;
  v_birthdate := '5-seq-70';

  -- 前两个参数通过位置传值，后两个参数通过命名法传值。
  insert_auths (v_code, v_name,
                p_birthdate => v_birthdate,
                p_sex => v_sex );

END;
```

存储过程的参数越多，调用就越困难，为了确保调用的准确性，可将存储过程的参数定义到一个PL/SQL记录中，这样，在调用时传递一个记录型的参数就可以了。PL/SQL中没有限制存储过程的参数个数。

3. 参数的缺省值

类似于变量的声明，一个过程或函数的形参可以有缺省值。如果参数有缺省值，那么在调用时就可以不用给它传值，只使用缺省值。如果给它传值，则实参的值代替缺省的值。参数缺省值的声明如下：

```
parameter [mode] datatype {:=|DEFAULT} initial_value
```

其中 parameter 是形参的名称，mode 是参数的模式（IN、OUT 和 IN OUT），datatype 是参数的类型（可以是预定义类型也可以是用户自定义类型），initial_value 用来为形参指定缺省值。使用关键字 DEFAULT 或“:=”来指定一个缺省值。

例如，定义 InsertAuths 存储过程：

```
CREATE OR REPLACE PROCEDURE InsertAuths (
  p_Author_code   auths.author_code%TYPE,
  p_Name    auths.name%TYPE,
  p_Birthdate  auths.birthdate%TYPE,
```

```
  /*设置P_Entry_date_time 参数的缺省值为系统当前的日期。*/
  P_Entry_date_time auths.entry_date_time%TYPE := SYSDATE,
  p_Sex       auths.sex%TYPE DEFAULT 1   --指定p_Sex的缺省值为1。
) AS

BEGIN
  -- 向auths表插入一条记录。
  INSERT INTO auths(author_code, name, sex, birthdate,
                    entry_date_time)
    VALUES (p_Author_code, p_Name, p_Sex, p_Birthdate,
            p_Entry_date_time);

  COMMIT;
END InsertAuths;
```

如果在该存储过程的调用中，没有实参与形参 p_Sex 关联，则系统会自动使用这个缺省值。我们可以用位置表示法来验证：

```
BEGIN
    InsertAuths ('A00021', '王达林', '6-May-71', '2-May-98');
END;
```

或者是用命名表示法调用：

```
BEGIN
    InsertAuths (p_name => '王达林',
                 p_author_code => 'A00021',
                 p_birthdate => '6-May-71',
                 p_entry_date_time => '2-May-98');
END;
```

在调用存储过程时，如果使用缺省值的参数不是存储过程中最后一个参数，则不能使用位置表示法。

5.2 创建函数

函数与存储过程非常类似，都有三种模式（IN，OUT 和 IN OUT）的参数。它们都可

以被存储在数据库中（当然过程与函数也可以不存于数据库中），并且在块中调用。

但存储过程和存储函数也有不同之处，存储过程只能作为一个 PL/SQL 语句调用，而函数作为表达式的一部分调用。并且它们的定义部分、可执行部分和异常处理部分是不同的。

5.2.1 创建函数的语法

创建函数的语法如下所示：

```
CREATE [OR REPLACE] FUNCTION schema.function
  [(argument [{IN|OUT|IN OUT}] datatype [,...] )]
RETURN return_datatype {IS | AS}
PL/SQL_body;
```

其中，function 是函数名，argument 是参数名，datatype 是参数的类型，return_datatype 是函数返回值的类型，PL/SQL_body 是函数的处理部分。

与存储过程一样，参数列表是可选的。在没有参数的情况下，函数的定义与调用都没有圆括号。但返回值类型是必需的，因为函数是作为表达式的一部分调用，必须返回一个值。

例如，如果作家表中男作家或女作家的工资在 200 元以上的人数大于百分之七十，则下面的函数返回 TRUE，否则返回 FALSE：

```
CREATE OR REPLACE FUNCTION SalaryStat (
  p_Sex auths.sex%TYPE )
  RETURN BOOLEAN IS

  v_CurrentSexAuthors NUMBER;
  v_MaxAuthors      NUMBER;
  v_ReturnValue       BOOLEAN;
  v_Percent       CONSTANT NUMBER := 70;
BEGIN
  -- 获得满足条件的作家的最大数。
  SELECT count(author_code)
    INTO v_MaxAuthors
```

```
    FROM auths
    WHERE sex = p_Sex
    AND salary >= 200 ;

  -- 统计表中指定性别的作家人数。
  SELECT count(author_code)
    INTO v_CurrentSexAuthors
    FROM auths
    WHERE sex = p_Sex ;

  -- 如果满足条件的作家人数所占的比例大于给出的百分比，
  -- 则返回TRUE，否则FALSE。
  IF (v_MaxAuthors / v_CurrentSexAuthors * 100) > v_Percent THEN
    v_ReturnValue := TRUE;
  ELSE
    v_ReturnValue := FALSE;
  END IF;

  RETURN v_ReturnValue;
END SalaryStat ;
```

SalaryStat 函数返回一个布尔值。可用下面的 PL/SQL 块来调用：

```
DECLARE
  CURSOR c_Auths IS
    SELECT distinct sex
      FROM auths;
BEGIN
  FOR v_AuthsRecord IN c_Auths LOOP
    IF SalaryStat(v_AuthsRecord.sex) THEN
      UPDATE auths
        SET salary = salary - 50
        WHERE sex = v_AuthsRecord.sex;
    END IF;
  END LOOP;
END;
```

例如，下面创建的函数将返回一个作家的文章标题信息，这个函数的参数是作家的姓名：

```
CREATE OR REPLACE FUNCTION Title(
 --参数是作家的姓名。
 p_name auths.name%TYPE)
 --将返回作家的文章标题信息。
 RETURN VARCHAR2 AS
   --这个变量用来存放p_name对应的作家代码值。
   v_AuthorCode CHAR(6);

   --这个游标用来查找p_name对应的作家的文章标题信息。
   CURSOR c_title IS
   SELECT title
     FROM article
     WHERE author_code=v_AuthorCode;

   --这个变量用来存放作家的一篇文章的标题。
   v_Title article.title%TYPE;

   --这个变量用来存放作家的所有文章标题信息。
   v_titles VARCHAR2(1000);
 BEGIN

   --查找名字为p_name的作家代码，如果不存在对应的作家代码，
   --将转到异常处理部分进行处理。
   SELECT author_code
     INTO v_AuthorCode
     FROM auths
     WHERE name=p_name;

   --打开游标。
   OPEN c_title;

   --推进游标。
   FETCH c_Title INTO v_Title;
   --如果没有查找到作家的标题，将返回没有查找到的信息，并返回。
```

```
  IF c_Title%NOTFOUND THEN
     RETURN '没有收录作家代码为'||v_AuthorCode||'的作家的文章！';
  END IF;

  --如果查找到作家的文章标题，则将该标题存放到v_Titles变量中。
  v_Titles:='作家代码为'||v_AuthorCode||'的作家的文章的标题为：';
  v_Titles:=v_Titles||v_Title||'、 ';

  --游标的推进循环，将作家的文章标题信息都存放到v_Titles变量中。
  FETCH c_title INTO v_Title;
  WHILE c_Title%FOUND LOOP
    v_Titles:=v_Titles||v_Title||'、 ';
    FETCH c_title INTO v_Title;
    END LOOP;
  v_Titles:=RTRIM(v_Titles,'、 ');

  --返回作家的文章标题信息。
  RETURN v_titles;
EXCEPTION

  --如果没有查找到p_name对应的作家代码，则返回没有找到信息，并返回。
  WHEN NO_DATA_FOUND THEN
    RETURN '没有收录名字为'||p_name||'的作家信息！';
END;
```

利用下面的块调用上面的函数：

```
DECLARE
  v_titles VARCHAR2(400);
BEGIN
  --“&name”是SQL*Plus中宏替换变量。
  v_titles:=title(&name);
  DBMS_OUTPUT.PUT_LINE(v_titles);
END;
```

在SQL*Plus中运行上面的块，执行结果如下所示：

1. 查找巴金的文章标题信息

```
Enter value for name: '巴金'
old   4:   v_titles:=title(&name);
new   4:   v_titles:=title('巴金');
没有收录名字为巴金的作家信息！
```

2. 查找王达琳的文章标题信息

```
Enter value for name: '王达琳'
old   4:   v_titles:=title(&name);
new   4:   v_titles:=title('王达琳');
作家代码为A00001的作家的文章的标题为：Intermet 将成为多语系网
```

3. 查找许灵延的文章标题信息

```
Enter value for name: '许灵延'
old   4:   v_titles:=title(&name);
new   4:   v_titles:=title('许灵延');
作家代码为A00004的作家的文章的标题为：VESA标准获得通过 、索尼和冲电气合
力开发芯片
```

4. 查找王达林的文章标题信息

```
Enter value for name: '王达林'
old   4:   v_titles:=title(&name);
new   4:   v_titles:=title('王达林');
没有收录作家代码为A00021的作家的文章！
```

5.2.2 函数的返回值

在函数内，是通过 RETURN 语句来返回值的。RETURN 语句的语法如下：

```
RETURN expression;
```

其中 expression 是准备返回的值。如果 expression 的类型与函数头中指定的类型不符，expression 类型会自动转换。RETURN 语句被执行后，控制权立刻返回给调用环境。

在函数体中可以有多条 RETURN 语句，但是只能有一条被执行。在函数结束的时候，如果没有执行 RETURN 语句会产生错误。

例如：

```
CREATE OR REPLACE FUNCTION SecrateInfo (
  /* 如果保密级别为“1”，则返回“一般”;
     如果保密级别为“2”，则返回“保密”;
     如果保密级别为“3”，则返回“密秘”;
     如果保密级别为“4”，则返回“绝密”;
     如果保密级别为其它数字，则返回“非法级别”。*/
  p_Article_code article.article_code%TYPE )
  RETURN VARCHAR2 IS

  v_secrate_level article.secrate_level%TYPE;
BEGIN
  --查询保密级别。
  SELECT secrate_level
    INTO v_secrate_level
    FROM article
    WHERE article_code = p_article_code;

  IF  v_secrate_level  = '1' THEN
    RETURN '一般';
  ELSIF v_secrate_level  = '2' THEN
    RETURN '保密';
  ELSIF v_secrate_level  = '3' THEN
    RETURN '密秘';
  ELSIF v_secrate_level  = '4' THEN
    RETURN '绝密';
  ELSE
    RETURN '非法级别';
  END IF;
END SecrateInfo;
```

RETURN 也可以用在存储过程中。在这种情况下，它没有参数。当执行了不带参数

的 RETURN 语句后，立刻将控制权返回到调用环境，并将 OUT 和 IN OUT 模式的形参的当前值传给实参，然后继续执行调用存储过程后的语句。

函数与存储过程的相同点：

- 都可以通过 OUT 模式的参数返回一个或多个值。
- 代码都有定义部分、可执行部分和异常处理部分。
- 都可以使用缺省值。
- 都可以用位置表示法和命名表示法调用。

在使用存储函数与存储过程时，一般情况下，如果只有一个返回值，则使用函数；如果有多个返回值则使用存储过程。尽管函数的参数可以是 OUT 模式，但是一般不这样使用。

5.3 删除过程与函数

删除存储过程的语法如下：

DROP PROCEDURE procedure;

删除函数的语法如下：

DROP FUNCTION function;

其中 procedure 是一个已创建的过程名，function 是一个已创建的函数名。

例如，下面的语句用来删除前面创建的 InsertAuths 存储过程：

DROP PROCEDURE InsertAuths;

5.4 库存子程序和局部子程序

我们前面所讲的子程序都是存储在数据库中的子程序，即库存子程序。这些子程序是由 CREATE 命令创建的，并可在其它的 PL/SQL 块中调用。它们在创建时要进行编译，并将编译后的代码存储在数据库中。当子程序被调用时，编译后的代码从数据库中读出并执行。

一个子程序也可以在块的定义部分创建，这样的子程序被叫作局部子程序。

例如，在下面的块中创建了一个局部函数 FormatName：

```
DECLARE
  CURSOR c_AllAuthors IS
    SELECT name, sex
      FROM auths;

  v_FormattedName VARCHAR2(60);

  /* 函数将返回带性别的作家名，性别用括号括起放在名字的后面。*/
  FUNCTION FormatName(p_Name IN VARCHAR2,
                      p_Sex IN NUMBER)
    RETURN VARCHAR2 IS
    v_Sex varchar2(16);
  BEGIN
    IF p_Sex = 1 THEN
       v_Sex := '男';
    ELSE
       v_Sex := '女';
    END IF;
    RETURN p_Name || '（' || v_Sex || '）';
  END FormatName;

-- 块的执行部分开始。
BEGIN
  FOR v_AuthsRecord IN c_AllAuthors LOOP
    v_FormattedName :=
      FormatName(v_AuthsRecord.name,
                 v_AuthsRecord.sex);
    DBMS_OUTPUT.PUT_LINE(v_FormattedName); -- 显示结果。
  END LOOP;

END;
```

如上例，在无名块的定义部分创建了 **FormatName** 函数。这个函数只在创建它的块中可用，它的作用域从创建它开始到块结束。因为它的作用域只在本块中，所以其它的块不能调用它。

局部子程序只能在定义部分的最后被创建，如果我们将 **FormatName** 函数移到 **c_AllAuthors** 的上面，就会出现编译错误。

子程序必须先定义再引用。在通常情况下，子程序的先定义再引用是比较容易办到的，但如果子程序间互相调用，子程序的先定义再引用就不好办到了。

第六章 包

包可将一些有联系的对象放在其内部。任何能在块定义部分出现的对象都可以在包中出现。这些对象包括存储过程、函数、游标、自定义的类型（例如 PL/SQL 表和记录）和变量。我们可以在其它的 PL/SQL 块中引用包中的这些对象，也就是说，包为 PL/SQL 提供了全局变量。

6.1 包的定义

一个包由两个独立的部分组成——包头和包体。各部分被单独地存放在数据字典中。定义一个包，要分别定义包头和包体。

6.1.1 定义包头

定义包头的语法如下所示：

```
CREATE [OR REPLACE] PACKAGE [schema.] package
{IS|AS}
   pl/sql_package;
```

其中 package 是包的名称，pl/sql_package 可以是存储过程、函数、变量、类型、异常以及游标的定义。

存储过程或函数必须在包头中预定义。也就是说，在包头中仅定义存储过程名或函数名以及它们的参数，存储过程或函数的执行代码将在包体中定义。这不同于无名块中定义存储过程和函数。

例如：

```
CREATE OR REPLACE PACKAGE AuthsPack AS

  --获得auths表中作家的工资。
  PROCEDURE QuerySalary(p_code auths.author_code%TYPE,
                        p_Salary OUT auths.salary%TYPE );
```

```
--向auths表插入记录。
PROCEDURE InsertAuthor( p_Code auths.author_code%TYPE,
                        p_Name auths.name%TYPE,
                        p_Sex auths.sex%TYPE,
                        p_Birthdate auths.birthdate%TYPE,
                        p_entry_date_time auths.entry_date_time%TYPE);

END AuthsPack;
```

AuthsPack 包中含有两个存储过程，QuerySalary 用来查询指定作家的工资，InsertAuthor 用来向 AUTHS 表插入记录。

6.1.2 定义包体

包体是一个数据字典对象。只有在包头成功编译后，包体才能被编译。包体只包含包头中已预定义的子程序的代码。在包头中定义（不是预定义）的对象可以直接在包体中使用，不必再在包体中定义。

```
CREATE [OR REPLACE] PACKAGE BODY [schema.] package
{IS | AS}
  pl/sql_body;
```

其中，package 为包名，该包名必须与定义包头时的名称一样，pl/sql_body 为过程与函数的定义。

例如，下面是 AuthsPack 包的包体：

```
CREATE OR REPLACE PACKAGE BODY AuthsPack IS

  --获得auths表中作家的工资。

  PROCEDURE QuerySalary(p_code auths.author_code%TYPE,
          p_Salary OUT auths.salary%TYPE )
  AS
   BEGIN

      SELECT salary INTO p_Salary
```

```
    . FROM auths
      WHERE author_code = p_code;

  END QuerySalary;

  --向auths表插入记录。
  PROCEDURE InsertAuthor( p_Code auths.author_code%TYPE,
                          p_Name auths.name%TYPE,
                          p_Sex auths.sex%TYPE,
                          p_Birthdate auths.birthdate%TYPE,
                          p_entry_date_time auths.entry_date_time%TYPE)
AS
  BEGIN

  INSERT INTO auths (author_code, name, sex, birthdate,
                     Entry_date_time)
              VALUES(p_code, p_name, p_sex, p_birthdate,
                     P_entry_date_time);
  END InsertAuthor;

END AuthsPack;
```

如果包头不包含存储过程和函数，则不必定义包体。如果包头中有预定义的子程序，则在包体中必须编写其子程序代码，而且包头和包体两部分指定的子程序必须一致，这包括相同的子程序名、参数名和参数类型。

6.1.3 包的初始化

与变量类似，包也可被初始化。只是初始化部分在包体的最后部分被定义，语法如下所示：

```
CREATE OR REPLACE PACKAGE BODY package {IS|AS}
  --包体中过程与函数的定义部分。
BEGIN
  Initialization_code;
END[package];
```

这里的 package 是包名，Initialization_code 是初始化代码。

如果在包体中定义了初始化部分，则每次调用包时，这个包都被初始化。例如，下面的包：

```
CREATE OR REPLACE PACKAGE   AuthorInfoPack AS
  v_auths_sex VARCHAR2(2);
  PROCEDURE Author_sex( p_Author_code auths.author_code%TYPE,
                        p_Sex OUT VARCHAR2 );
END AuthorInfoPack;
```

定义 AuthorInfo 包的包体：

```
CREATE OR REPLACE PACKAGE BODY AuthorInfoPack AS
  PROCEDURE Author_sex( p_Author_code auths.author_code%TYPE,
                        p_Sex OUT VARCHAR2 )
  AS
   v_sex NUMBER;
   BEGIN
     -- 获取作家的性别代码。
     SELECT sex INTO v_sex
     FROM auths
     WHERE author_code = p_author_code;
     -- 将性别代码转换为性别。
     IF v_sex = 0 THEN
        p_Sex := '女';
     ELSE
        p_Sex := '男';
     END IF;
   END Author_sex;
    -- 初始化部分。
BEGIN
  Author_sex('A00001', v_auths_sex);
END AuthorInfoPack ;
```

6.2 包的使用

定义好包头和包体后，就可以使用包了。下面我们讲解包的使用。

6.2.1 包中对象的引用

在包中定义的任意对象都可在包外使用，只是在引用该对象前用包名做前缀。

例如，我们可通过如下 **PL/SQL** 语句调用 **AuthsPack** 包中的存储过程 **QuerySalary**：

```
--设置存储缓冲区的大小。
SET SERVEROUTPUT ON SIZE 100000
DECLARE
  v_Salary auths.salary%TYPE;
BEGIN
  AuthsPack. QuerySalary ('A00010',v_Salary);

  -- 显示过程QuerySalary的查询结果。
  DBMS_OUTPUT.PUT_LINE('A00010作家的工资为：');
  DBMS_OUTPUT.PUT_LINE(v_Salary);
END;
```

执行结果如下：

```
A00010 作家的工资为：
120
```

这个存储过程的调用与单独调用存储过程是类似的。唯一不同的是此存储过程的调用有包名作前缀。与标准的存储过程一样，包中存储过程的参数也可以有缺省值，这些参数可通过位置表示法或命名表示法来调用。在包体内部不必带包名前缀即可直接引用包头中的对象。

6.2.2 重载包中的子程序

在包的内部，存储过程和函数都可被重载，这意味着有多个存储过程或函数可以使用同一个名称，但是参数不能相同。这样就允许用不同的参数调用同一个名字的过程或函数。

例如，通过指定文章编码和作家代码来增加一篇文章，或通过指定作家姓名和文章编码来增加一篇文章。下面定义一个包 ArticlePack 来实现这个功能：

```
CREATE OR REPLACE PACKAGE ArticlePack is
  PROCEDURE AddArticle(p_ArticleCode article.article_code%TYPE,
                       p_AuthorCode article.author_code%TYPE,
                       p_SecrateLevel article.secrate_level%TYPE,
                       p_PubDate article.pub_date%TYPE);
  PROCEDURE AddArticle(p_Birthdate auths.birthdate%TYPE,
                       p_ArticleCode article.article_code%TYPE,
                       p_SecrateLevel article.secrate_level%TYPE,
                       p_PubDate article.pub_date%TYPE);
END ArticlePack;

CREATE OR REPLACE PACKAGE BODY ArticlePack IS
  PROCEDURE AddArticle (p_ArticleCode article.article_code%TYPE,
                       p_AuthorCode article.author_code%TYPE,
                       p_SecrateLevel article.secrate_level%TYPE,
                       p_PubDate article.pub_date%TYPE) AS
  BEGIN
    INSERT INTO Article (Article_code,Author_code,
                         secrate_level, pub_date)
      VALUES(p_ArticleCode,p_AuthorCode,p_Secratelevel,p_PubDate);
    COMMIT;
  END AddArticle;

  PROCEDURE AddArticle(p_Birthdate auths.birthdate%TYPE,
                       p_ArticleCode article.article_code%TYPE,
                       p_SecrateLevel article.secrate_level%TYPE,
                       p_PubDate article.pub_date%TYPE) AS
  v_AuthorCode auths.author_code%TYPE;
  BEGIN
    SELECT author_code
      INTO v_AuthorCode
      FROM auths
      WHERE birthdate=p_Birthdate;
    INSERT INTO article
```

```
        (article_code,author_code,secrate_level,pub_date)
        VALUES(p_ArticleCode,v_AUthorCode,p_SecrateLevel,p_PubDate);
  END AddArticle;
END ArticlePack;
```

我们可以通过如下两种方法增加一篇文章：

方法一：

```
BEGIN
    ArticlePack.AddArticle('www','A00001','1',TO_DATE('12-MAY-1980'));
END;
```

方法二：

```
BEGIN
  ArticlePack.AddArticle(TO_DATE('12-NOV-58'),'www','1',
  TO_DATE('12-MAY-80'));
END;
```

重载是非常有用的技术，但是，它也有一些约束，这些约束如下所示：

- 当仅仅参数名不同或者是模式（IN、OUT、IN OUT）不同时，不能重载了程序。例如，下面的两个存储过程不能被重载：

  ```
  PROCEDURE Overload(p_Par IN CHAR);
  PROCEDURE Overload(p_Par OUT CHAR);
  ```

  上面的重载仅仅参数模式不同（一个为 IN，一个为 OUT），不能这样重载。

- 不能对仅有返回类型的函数进行重载。例如，下面的函数不能被重载：

  ```
  FUNCTION Overload FUN RETURN CHAR;
  FUNCTION Overload FUN RETURN BINARY_INTEGER;
  ```

- 重载函数的参数必须是数据类型不同或其类型间不可自动转换。例如，由于 CHAR 和 VARCHAR2 的变量类型可以自动转换，因此不能重载下面的存储过程：

  ```
  PROCEDURE OverloadChar(p_TheParameter IN CHAR);
  ```

PROCEDURE OverloadChar(p_TheParameter IN VARCHAR2);

尽管在定义包含违反上述限制的子程序的包时不会报编译错误。但是，运行时 PL/SQL 引擎不能调用该子程序，会出现错误“**PLS-307 : too many declarations of 'subprogram' match this call.**”。

6.3 在 SQL 语句中使用的函数

通常在 SQL 语句中不能调用 PL/SQL 函数（无论是单独存储在数据库中的函数还是包中的函数），因为 PL/SQL 函数是过程性的语句。在 PL/SQL 2.1 以上版本中放宽了这个限制，但函数必须满足特定的约束才能在 SQL 语句中使用。

PL/SQL 为函数指定了四种基本约束，这四种基本约束如下表所示：

基本约束	含　　义	描　　述
WNDS	Writes no database state	在函数内不能用 DML 语句修改数据库中的表。
RNDS	Reads no database state	在函数内不能通过 SELECT 语句来读取数据库中的表。
WNPS	Writes no package state	在函数内不能修改包变量（包变量不能在赋值语句的左边或一个 FETCH 语句的 INTO 语句中）。
RNPS	Reads no package state	在函数内不能查询包变量（包变量不能在赋值语句的右边，或不能是 SQL 表达式的一部分）。

根据对函数的四种基本约束，满足下面约束的函数可以被 SQL 语句调用：

- 当函数满足 WNDS 时，能够被 SQL 语句调用。
- 当函数满足 RNPS 和 WNPS 约束时，这个函数（通过数据库链接）能被远程或并行调用。
- 在 SELECT、VALULES 或 SET 子句中调用的函数可以没有 WNPS 约束。但在其它的子句中就必须满足 WNPS 约束。
- 一个函数所调用的子程序与该函数的约束级别相同。例如，假设一个函数调用一个执行 UPDATE 操作的存储过程，如果该函数没有 WNDS 约束，则这个存储过程也没有 WNDS 约束，所以也不能在 SQL 语句中使用。
- 在含有 CREATE TABLE 或 ALTER TABLE 命令的 CHECK 子句中，不能调用存

储在数据库中的 PL/SQL 函数，因为这些语句中的定义不能变化。

另外，用户自定义函数必须符合如下约束才能被 SQL 语句调用，这些约束对于内嵌函数（系统提供的函数）也同样适用：

- 函数必须单独的或作为包的一部分存储在数据库中，不能作为块的一部分。
- 函数只能定义 IN 参数，不能定义 IN OUT 或 OUT 参数。
- 形参类型必须是数据库中的数据类型，不能是 PL/SQL 中的数据类型（如 BOOLEAN 或 RECORD 类型）。数据库类型包括 NUMBER、CHAR、VARCHAR2、ROWID、LONG、LONG RAW 和 DATE。
- 函数返回的数据类型也必须是数据库中的数据类型。

例如，函数 nameandsex 将作家代码作为输入，返回作家的名字和性别：

```
CREATE OR REPLACE FUNCTION nameandsex(
   p_authorcode auths.author_code%TYPE)
   RETURN VARCHAR2 AS
   v_nameandsex varchar2(100);
 BEGIN
   SELECT name||' '||replace(replace(sex,0,'女'),1,'男')
        INTO v_nameandsex
        FROM auths
        WHERE author_code=p_authorcode;
   RETURN v_nameandsex;
 END nameandsex;
```

函数 nameandsex 符合上面所有的约束，所以我们可以用 SQL 语句调用它，如下所示：

```
SELECT seqno,nameandsex(author_code) "姓名和性别"
  FROM auths
  WHERE author_code < 'A00004';
```

显示结果如下：

```
SEQNO
---------
```

```
姓名和性别
-----------
         1
王达琳 女

         2
李井元 男

         3
赵封昭 男
```

当将单独存储在数据库中的函数用在 SQL 语句中时，PL/SQL 引擎自动确定该函数有哪些约束，这些约束能否保证函数在 SQL 语句中的调用。对于包函数，则先在包内部使用 RESTRICT_REFERENCES 编译指令来指定包函数的约束。在 SQL 语句中调用这个包函数时，PL/SQL 根据编译指令指定的约束来判断包函数能否在 SQL 语句中调用。

RESTRICT_REFERENCES 编译指令通过下面的语法指定函数的约束：

```
PRAGMA RESTRICT_REFERENCES(function_name,
    WNDS[,WNPS][,RNDS][,RNPS]);
```

这里的 function_name 是包函数名。由于能在 SQL 语句中调用的函数都有 WNDS 基本约束，因此编译指令同样也要求这样。其它的基本约束都可以以任意次序指定。编译指令须在函数所在的包头中指定。

例如，包 AuthorPack 使用两次 RESTRICT_REFERENCES 编译指令：

```
CREATE OR REPLACE PACKAGE AuthorPack AS

FUNCTION fun(p_AuthorCode auths.author_code%TYPE)
  RETURN VARCHAR2;
--使用RESTRICT_REFERENCES编译指令为函数fun指定三种约束
--这三种约束是WNDS、WNPS、RNPS。
PRAGMA RESTRICT_REFERENCES(fun,WNDS,WNPS,RNPS);

FUNCTION AuthorCount RETURN NUMBER;
--使用RESTRICT_REFERENCES编译指令为函数AuthorCount指定三种约束。
```

```
PRAGMA RESTRICT_REFERENCES(AuthorCount,WNDS,WNPS,RNPS);
END AuthorPack;

CREATE OR REPLACE PACKAGE BODY AuthorPack AS

--v_num是包变量。
v_num NUMBER;

--函数fun的函数体满足WNPS、WNDS和RNPS约束。
FUNCTION fun(p_AuthorCode auths.author_code%TYPE)
  RETURN VARCHAR2 AS
  v_return VARCHAR2(16);
BEGIN
  SELECT author_code||name
    INTO v_return
    FROM auths
    WHERE author_code=p_AuthorCode;
  RETURN v_return;
END fun;

--函数AuthorCount的函数体不满足WNPS和RNPS约束。
FUNCTION AuthorCount RETURN NUMBER AS
   v_return NUMBER;
 BEGIN
   IF v_num IS NULL THEN  --包变量v_num被读，这将不满足RNPS约束。
     SELECT COUNT(*)
       INTO v_return
       FROM AUTHS
       WHERE author_code LIKE 'A%';
     v_num:=v_return;  --包变量v_num被修改，这将不满足WNPS约束。
   ELSE
     v_return:=v_num;
   END IF;
   RETURN v_return;
 END AuthorCount;
 END AuthorPack;
```

在包头的 fun 函数中使用了 RESTRICT_REFERENCES 编译指令指定约束，包体中的 fun 函数代码显然符合编译指令指定的约束——没有修改数据库表、没有读包中变量、没有修改包中变量。

在包头的 AuthorCount 函数中也使用了 RESTRICT_REFERENCES 编译指令，但在包体中的 AuthorCount 函数代码并不符合指定的约束，包体中定义的变量 v_num 不仅被该函数读出（不满足 RNPS 约束），而且被修改（不满足 WNPS 约束），所以 PL/SQL 引擎编译到函数 AuthorCount 时，报“PLS-00452: Subprogram 'AUTHORCOUNT' violates its associated pragma”错误。

在使用 RESTRICT_REFERENCES 时应注意的几点：

- PL/SQL 编译程序根据编译指令来确定包函数的基本约束，从而确定这个包函数是否能在 SQL 语句中使用。只要随后修改了包体（或第一次创建），就要按照编译指令校验函数代码。

- 包初始化部分的代码同样也可以有基本约束。包的基本约束也用 RESTRICT_REFERENCE 来指定，但应以包名为参数而不是以函数名为参数。如下面的结构：

```
CREATE OR REPLACE PACKAGE AuthorPack AS
PRAGMA RESTRICT_REFERENCES(AuthorPack,WNDS,WNPS,RNPS);
...
END AuthorPack;
```

- RESTRICT_REFERENCES 可在包中函数定义后的任何位置出现，但它只能约束一个函数的定义。因此，对于函数的重载，pragma 只约束最近定义的函数。如下面的例子，编译指令只约束第二次定义的 F 函数：

```
CREATE OR REPLACE PACKAGE Testpackage AS
FUNCTION   F(p_ParameterOne IN NUMBER) RETURN VARCHAR2;
FUNCTION   F   RETURN   DATA;
PRAGMA REATRICT_REFERENCES(F,WNDS,RNDS);
END TestPackage;
```

注意，如果函数中用到了 DBMS_OUTPUT、DBMS_PIPE、DBMS_ALTER、DBMS_SQL、UTL_FILE 等系统包，则该函数不能用在 SQL 语句中。

当在过程性语句中调用一个函数时，可以使用参数缺省值。而通过 SQL 语句调用一个函数时，所有的参数都必须指定。另外，必须使用位置表示法，而不能使用命名表示法。例如，下面 fun 的调用是非法的：

```
SELECT fun(p_AuthorCode => 'A00001') FROM   auths;
```

6.4 系统提供的包 DBMS_OUTPUT

Oracle8 中，系统提供了一些包，这些包是在安装 Oracle 时自动安装的，用户可以直接使用。在内嵌包中有一个包经常用到——DBMS_OUTPUT，下面我们就简单地介绍这个系统包。

内嵌包 DBMS_OUTPUT 用来输出 PL/SQL 变量的值。DBMS_OUTPUT 包和其它系统包一样，都属于 Oracle 系统用户 SYS 内的对象。

DBMS_OUTPUT 包中有如下一些存储过程：

- PUT 和 PUT_LINE：将数据存放在缓冲区中。PUT_LINE 同时要换行。
- NEW LINE：是在缓冲区中加换行符，表明一行结束。
- GET_LINE：返回一个字符串。
- GET_LINES：返回一个 PL/SQL 表。
- ENABLE 和 DISABLE：用于控制缓冲区的大小。

下面给出这些过程的定义与说明：

定　义	说　明	举　例
PROCEDURE PUT(param);	其中 param 是 PUT 的参数，该参数可以是 VARCHAR2、NUMBER 和 DATE 类型的变量或数据。	DBMS_OUTPUT.PUT(v_name); DBMS_OUTPUT.PUT('abcde');
PROCEDURE PUT_LINE (param);	该过程的用法与 PUT 相同，只是它在最后会加一个换行符。	DBMS_OUTPUT.PUT_LINE(v_ address); DBMS_OUTPUT.PUT_LINE ('fehigk');
PROCEDURE	NEW_LINE 是在缓冲区中加	DBMS_OUTPUT.

定 义	说 明	举 例
NEW_LINE;	换行符，表明一行结束。在缓冲区中，行数是没有限制的，而整个缓冲区的大小由 ENABLE 来指定。	NEW_LINE;
PROCEDURE GET_LINE(line, status);	它将返回一个字符串和一个用来判断是否检索成功的状态值，如果检索成功，则返回 0，否则返回 1。	DBMS_OUTPUT. GET_LINE(v_line,v_stat);
PROCEDURE GET_LINES(line, tatus);	其中 param 是返回的行数，该过程的用法同 GET_LINE 相同，只是返回的是一个 PL/SQL 表。该表的定义为： TYPE CHARARR IS TABLE OF VARCHAR2(255) INDEX BY BINARY_INTEGER;	DBMS_OUTPUT. GET_LINES(v_chardata, v_lines);
PROCEDURE ENABLE (buffer_size);	buffer_size 是缓冲区的大小，缺省值为 20000，缓冲区的最大范围是 1,000,000 字节。	DBMS_OUTPUT. ENABLE(100000);
PROCEDURE DISABLE;	清除缓冲区中的内容，再次调用 PUT 或 PUT_LINE，其参数不会再存入缓冲区中。	DBMS_OUTPUT. DISABLE;

第七章 异常处理

在 PL/SQL 中出现的警告或错误叫做异常，对异常的处理称为异常处理。

7.1 异常

异常分为预定义异常和用户自定义异常。预定义异常是由系统定义的异常。由于它们已在 STANDARD 包中预定义了，因此，这些预定义异常可以直接在程序中使用，而不必在定义部分声明。而用户自定义异常则需要在定义部分声明后才能在可执行部分使用。用户自定义异常对应的错误不一定是 Oracle 错误，例如，它可能是一个数据错误。

7.1.1 预定义异常

常见的预定义异常如下表所示：

Oracle 错误	异　常	产生异常的时机
ORA-0001	DUP_VAL_ON_INDEX	违反唯一性约束。
ORA-0051	TIMEOUT_ON_RESOURCE	当等待资源时超时。
ORA-0061	TRANSACTION_BACKED_OUT	由于死锁使事务回退。
ORA-1001	INVALID_CURSOR	执行了非法的游标操作。例如，试图关闭一个已经关闭的游标。
ORA-1012	NOT_LOGGED_ON	没有连接到 Oracle。
ORA-1017	LOGIN_DENIED	错误的用户名或口令。
ORA-1403	NO_DATA_FOUND	SELECT...INTO 语句没有返回任何一行，或试图引用一个还没有被赋值的 PL/SQL 表元素。
ORA-1422	TOO_MANY_ROWS	一条 SELECT...INTO 语句查到多行。
ORA-1476	ZERO_DIVIDE	被零除。
ORA-1722	INVALID_NUMBER	未能将 SQL 语句中的字符串转换成数字就会产生这个异常。
ORA-6500	STORAGE_ERROR	由于 PL/SQL 运行时内存溢出，导致内部 PL/SQL 错误发生。
ORA-6501	PROGRAM_ERROR	内部的 PL/SQL 错误，这属于系统软件的问题。

Oracle 错误	异　常	产生异常的时机
ORA-6502	VALUE_ERROR	由于在过程性语句中出现转换、截断、算术错误而产生的异常。
ORA-6504	ROWTYPE_MISMATCH	一个主游标变量和 PL/SQL 游标变量的类型不匹配。
ORA-6511	CURSOR_ALREADY_OPEN	试图打开一个已打开的游标。
ORA-6530	ACCESS_INTO_NULL	试图给一个 NULL 对象的属性赋值。
ORA-6531	COLLECTION_IS_NULL	试图对一个 NULL 值的 PL/SQL 表或变长数组执行除 EXISTS 以外的操作。
ORA-6532	SUBSCRIPT_OUTSIDE_LIMIT	引用的嵌套表或变长数组索引超出了其声明范围（该异常出现在 PL/SQL2.2 以上版本中）。
ORA-6533	SUBSCRIPT_BEYOND_COUNT	引用的嵌套表或变长数组索引大于了嵌套表或嵌套表中的元素个数（该异常出现在 PL/SQL8.0 版本中）。

例 1，下面的块中产生一个 NO_DATA_FOUND 异常：

```
DECLARE
  TYPE t_NumberTableType IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
  v_NumberTable t_NumberTableType;
  v_TempVar NUMBER;
BEGIN
  v_TempVar:=v_NumberTable(1);
END;
```

例 2，下面的语句会产生 INVALID_NUMBER 异常，因为“'123a'”不是一个合法的工资值：

```
insert into auths (author_code,name,birthdate,entry_date_time,salary)
    values('A00022','王',TO_DATE('02-may-60'),TO_DATE('04-MAR-90')
            ,'123a');
```

例 3，下面的两个块中将产生 VALUE_ERROR 异常：

1.
```
DECLARE
  V_TempVar varchar2(3);
BEGIN
  v_TempVar:='ABCD';
END;
```

2.
```
DECLARE
    v_TempVar NUMBER(1);
BEGIN
   SELECT seqno
     INTO v_TempVar
     FROM auths
     WHERE author_code='A00010';
END;
```

7.1.2 异常的声明

用户自定义异常必须先在定义部分声明，然后再使用，这一点与变量声明类似。用户自定义异常声明的语法如下：

```
EXCEPTION exception
```

其中 EXCEPTION 用来指定声明的是异常。exception 是一个自定义异常名。

例如，下面的语句声明了一个名为 e_TooManyAuthors 的异常：

```
DECLARE
  e_TooManyAuthors EXCEPTION;
```

注意，尽管异常的声明与变量的声明类似，但异常是一个错误状态，而不是一个数据项，所以异常不能出现在赋值语句或 SQL 语句中。但异常的作用域与定义部分其它变量的作用域相同。如果一个用户自定义异常被传递到作用域外，则不能再通过原来的名字引用它。为了解决这个问题，我们可以在包中声明异常，这个异常就可以在任何块中使用，使用时在异常前加包名前缀即可。

例如，下面的包中声明了一个异常 e_ UserDefinedException，这个异常可以在任何块中使用：

```
1
CREATE OR REPLACE PACKAGE Globals
  /* 这个包中声明的对象可在任意块中引用。注意这个包没有包体。*/

    e_ UserDefinedException EXCEPTION;
    ...
END Globals;
```

由于预定义异常只是与一部分 Oracle 错误相连的异常，所以如果要处理没有与预定义异常对应的 Oracle 的错误时，则需要为这些 Oracle 错误声明相应的用户自定义异常。声明这样的异常需要使用 EXCEPTION_INIT 编译指令。EXCEPTION_INIT 编译指令的定义如下所示：

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

exception_name 是预先被声明的异常名，Oracle_error_number 是错误号。这条命令必须写在定义部分。

例如，在块定义部分声明了一个与错误“ORA-01401: inserted value too large for column”对应的用户自定义异常 e_TooLarge。当在块的可执行部分向表中插入的列值超出指定的列长度时，则产生 e_TooLarge 异常：

```
DECLARE
    V_code auths.name%TYPE;
    e_ TooLarge EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_ TooLarge, -1401);
```

注意，通过 EXCEPTION_INIT，一个用户自定义异常只能和一个 Oracle 错误相连。在异常处理语句中，SQLCODE 和 SQLERRM 将返回这个 Oracle 错误的代码和消息文本，而不是返回用户自定义消息。

7.1.3 异常的产生

当与预定义异常对应的错误出现时，则该预定义异常就会自动产生。而一个用户自定义异常通常由 RAISE 语句来产生（由 EXCEPTION_INIT 编译指令声明的用户自定义异常也可通过对应 Oracle 错误的出现而产生）。当然如果需要的话，预定义异常也可以使用 RAISE 语句来产生。

例 1，在下面块中使用 RAISE 语句产生用户自定义异常 e_TooSmallSalary：

```
DECLARE
  -- 声明用户自定义异常e_TooSmallSalary。
  e_TooSmallSalary EXCEPTION ;
  -- v_CurrentSalary是当前作家的工资。
  v_CurrentSalary NUMBER(8,2);
  -- v_SmallSalary是作家工资的最低限。
  v_SmallSalary NUMBER(8,2) DEFAULT 100;
BEGIN
  /* 查找“A00002”作家的工资*/
  SELECT salary
    INTO v_CurrentSalary
    FROM auths
    WHERE author_code = 'A00002';
  /* 检查作家的工资。*/
  IF v_CurrentSalary < v_SmallSalary THEN
    /* 该作家的工资太少。*/
    --产生e_TooSmallSalary异常。
    RAISE e_TooSmallSalary;
  END IF;
END;
```

当一个异常产生时，控制权立即转交给块的异常处理部分。如果该块没有异常处理部分，则向该块的外一层块传递。一旦控制权交给了异常处理部分，则再没有办法返回到块的可执行部分。

预定义异常通常是在与之相对应的错误发生时产生。

例 2，下面的 PL/SQL 块产生 NO_DATA_FOUND 异常，该异常是当没有查询（SELECT 语句）到表中的对应记录时产生：

```
DECLARE
  V_name VARCHAR2(10);
BEGIN
   SELECT name INTO v_name FROM auths WHERE author_code='B00006';
```

```
END;
```

由于没有作家代码号为“B00006”的作家，所以会产生“ORA-01403: not data found”错误，它就对应于 NO_DATA_FOUND 异常。

由 EXCEPTION_INIT 编译指令声明的用户自定义异常与一个 Oracle 错误相连，所以这样的用户自定义异常也是在一个 Oracle 错误发生时产生。

例 3，在上节中已声明了一个用户自定义异常 e_ TooLarge（使用 EXCEPTION_INIT 编译指令声明），当向表中插入一个列值超出了该列指定的长度时，会自动产生该异常：

```
BEGIN
  INSERT INTO auths (author_code,name,birthdate,entry_date_time)
    VALUES('A000001', 'WANG',TO_DATE('11-3 月-50'),
      TO_DATE('12-1 月-97'));
END;
```

由于插入的作家代码值“A000001”超出了列 aurhor_code 的长度，所以产生错误“ORA-01401: inserted value too large for column”，它对应用户自定义异常“e_ TooLarge”，同时控制权转到块外的调用环境。

当然用户自定义异常 e_TooLarge 也可以使用 RAISE 语句来产生，这里就不再举例了。

通常用户自定义异常是在声明后才能产生，但如果我们使用 RAISE_APPLICATION_ERROR 函数就可以直接产生异常，并且能为异常定义用户自己指定的错误消息。执行完 RAISE_APPLICATION_ERROR 函数后，控制权转到块外的调用环境。

RAISE_APPLICATION_ERROR 的定义如下所示：

RAISE_APPLICATION_ERROR（error_number,error_message,[keep_errors]）；

其中，error_number 是一个错误号，值在-20,000 到-20,999 之间，error_message 是与该错误相连的错误消息文本，它最大不能超过 512 个字符。keep_errors 是一个 boolean 值。该参数是可选的。如果 keep_errors 为 TRUE，则这个新的错误将加在已产生的错误列表之后。如果 keep_errors 为 FALSE，则这个新错误将代替当前的错误列表。

例 4，下面的存储过程用来为作家长工资，如果要长工资的作家根本不存在，则使用 RAISE_APPLICATION_ERROR 语句产生一个异常，并提示出错信息，然后将控制权转到存储过程外：

```
CREATE OR REPLACE PROCEDURE SalaryAdd (
  /* 参数 p_Author_code 用来确定准备长工资的作家，参数 p_AddSalary
     用来传入增加的工资。 */
  p_Author_code IN auths.author_code%TYPE,
  p_AddSalary IN auths.salary%TYPE ) AS
  v_CurrentSalary auths.salary%TYPE;  -- 作家工资增加以后的值。
  v_MaxSalary auths.salary%TYPE default 1000 ; -- 最高工资 1000 元。
BEGIN
  -- 修改作家的工资。
  UPDATE auths
    SET salary = salary + p_AddSalary
    WHERE author_code = p_Author_code
     RETURNING salary INTO v_currentsalary;

  -- 如果作家 p_Author_code 不存在，则提示一个错误信息，并退出该存储过程。
  IF SQL%NOTFOUND THEN
     /* 准备长工资的作家不存在，提示一个错误信息。*/
     RAISE_APPLICATION_ERROR(-20001, '没有代码为' || p_Author_code
                                  || '的作家存在');
  END IF;

  /* 提交所做的修改。*/
  COMMIT;
END SalaryAdd;
```

7.1.4 异常处理

异常处理部分包含着对异常的处理语句。当一个异常相应的错误发生导致这个异常产生时，异常处理语句被执行。异常处理部分的语法如下所示：

```
EXCEPTION
  WHEN exception_name THEN
     Sequence_of_statements1;
```

```
  WHEN exception_name THEN
    Sequence_of_statements2;
  WHEN OTHERS THEN
    Sequence_of-staements3;
END;
```

其中 exception_name 是异常的名字。OTHERS 语句对应前面 WHEN 子句中未出现的所有错误。Sequence_of_statements 是对 exception_name 异常或其它的所有异常（OTHERS）进行处理的语句。

例如，我们将上节中例 1 再重新编写（加上异常处理部分），在异常处理部分对产生的异常 e_TooSmall 进行如下所示的处理：

```
DECLARE
    e_TooSmallSalary EXCEPTION;
    v_CurrentSalary NUMBER(8,2);
    v_SmallSalary NUMBER(8,2) DEFAULT 100;
BEGIN
    SELECT salary
      INTO v_CurrentSalary
      FROM auths
      WHERE author_code = 'A00002';
    IF v_CurrentSalary < v_SmallSalary THEN
       RAISE e_TooSmallSalary;
    END IF;
EXCEPTION
    WHEN e_TooSmallSalary THEN
       UPDATE auths SET salary=500 WHERE author_code='A00002';
END;
```

一条异常处理语句可以处理多个异常。只要在 WHEN 子句中加由 OR 分隔的多个异常名即可，这里就不多介绍了。

如果块中的异常没有被处理，则该块会带着未处理的异常返回调用它的程序，这会导致调用它的程序出错。如果在存储过程中出现异常，则存储过程的 OUT 参数将得不到返回值。为了避免未处理异常带来的弊病，我们最好在块的最外层使用 OTHERS 子句处理块中所有未处理的异常。这样就可以确保所有的错误都能被发现和处理。

例如，可为上例的异常处理部分的结尾加一条语句，表明当出现其它异常时，则取消事务的提交：

```
WHEN OTHERS THEN
  /* 如果是其它的错误就执行该处理。*/
  ROLLBACK;
```

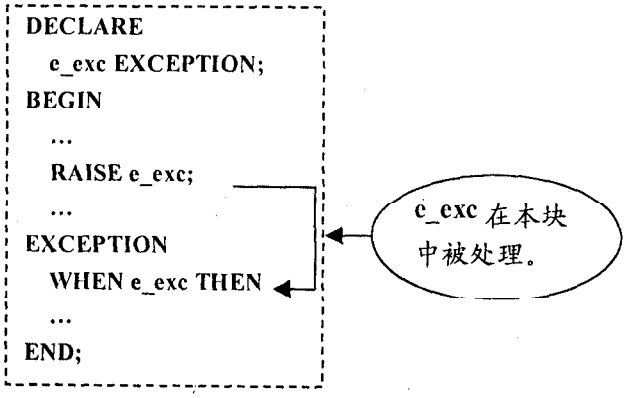
7.1.4.1 处理可执行部分产生的异常

尽管在块的三个部分（定义部分、可执行部分和异常处理部分）都可能产生异常，但大多数情况下是在可执行部分产生异常。前面的 **e_TooSmallSalary** 就是在可执行部分产生的。

当块的可执行部分中产生了异常时，如果块中有该异常对应的处理语句，则执行这条处理语句，该块也就被成功地完成，控制权转到该块外的调用环境中；如果这个块中没有该异常对应的处理语句，则这个异常被传递到该块外的调用环境中。

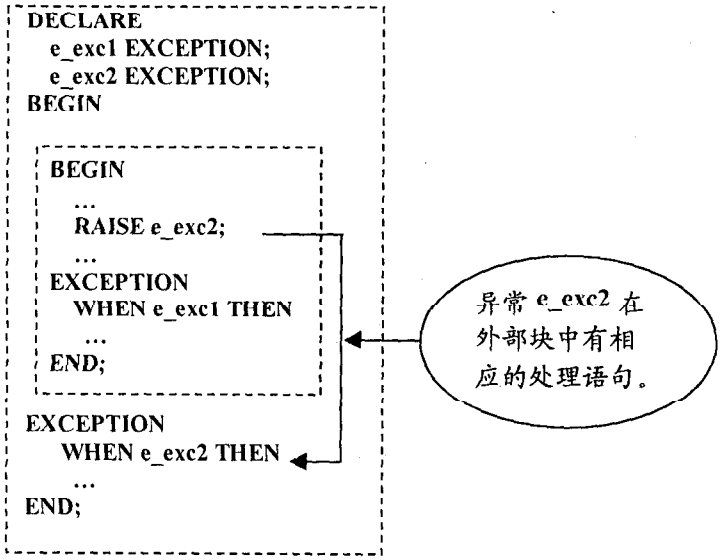
下面举例说明可执行部分产生的异常在不同情况下的处理方式。

例 1，下图中的异常 **e_exce** 在内部块中产生并被处理，然后控制权转到块外调用环境中：

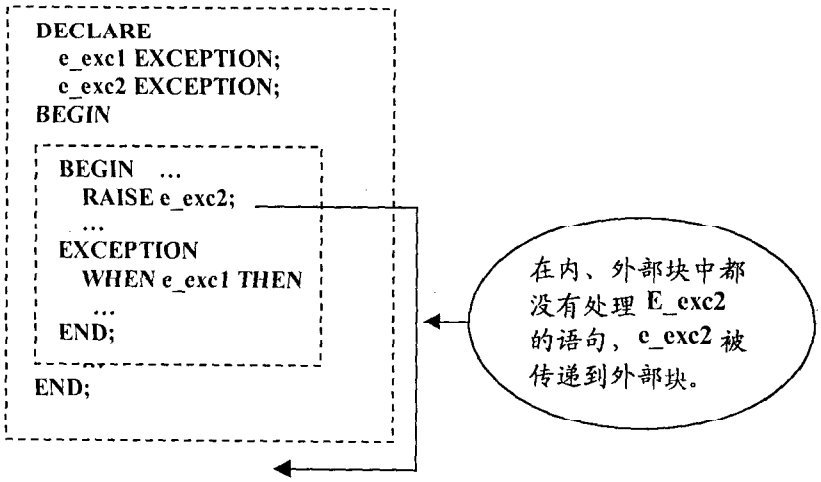


例 2：下图中，在内部块中产生的异常 **e_exc2**（注意该异常是在外部块中被声明）在

本块中没有对应的处理语句，而在外部块中有 e_exc2 的处理语句，则 e_exc2 被传递到外部块中处理：



例 3：下图中，在内部块中产生的异常 e_exc2（该异常在外部块中被声明）在本块和外部块中都没有对应的处理语句，则异常 e_exc2 被传递到外部块外：

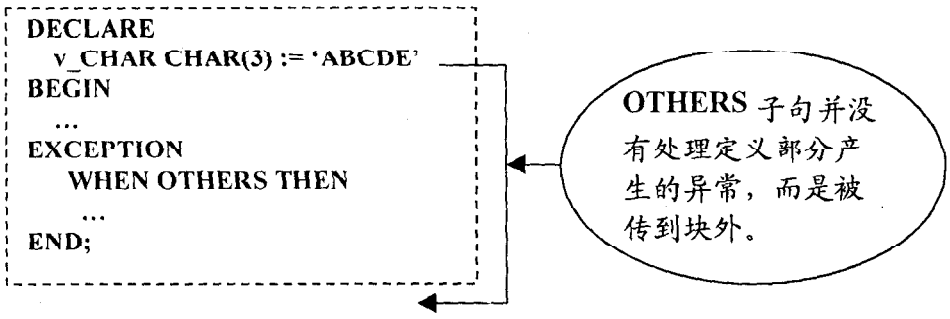


7.1.4.2 处理定义部分产生的异常

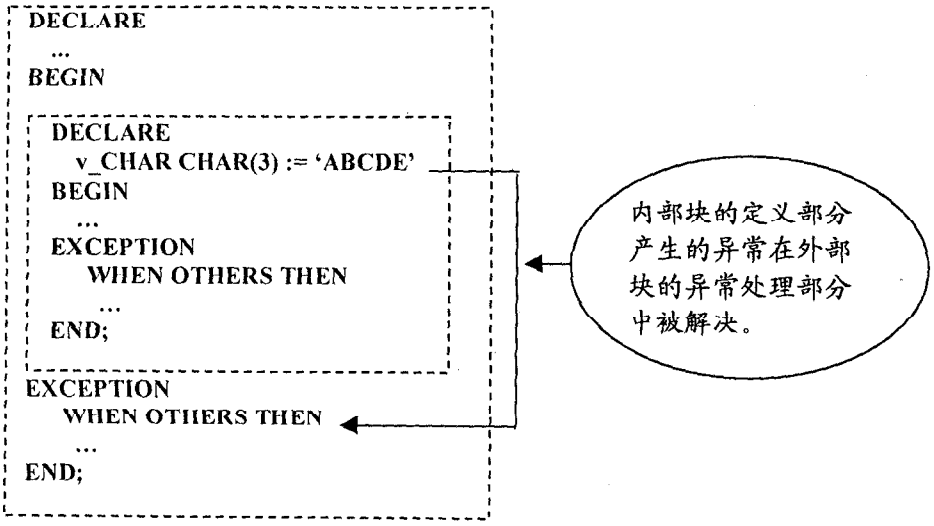
如果是定义部分的一个赋值语句产生了异常，即使在当前块的异常处理部分中有处理该异常的处理语句，也不去执行，而是立刻被传递到外部块中。当异常传递到外部块中以后，按照处理可执行部分中产生的异常一样去处理该异常。

下面举例说明在定义部分产生的异常的处理。

例 1，在定义部分产生了一个 **VALUE_ERROR** 异常，块中的 **OTHERS** 处理语句并没有处理该异常，该异常被传递到块外：



例 2，在内部块的定义部分产生了异常 **VALUE_ERROR**，内部块中的 **WHEN OTHERS** 语句并没有处理该异常，这个异常在外部块的 **WHEN OTHERS** 语句中处理，并且外部块成功结束：

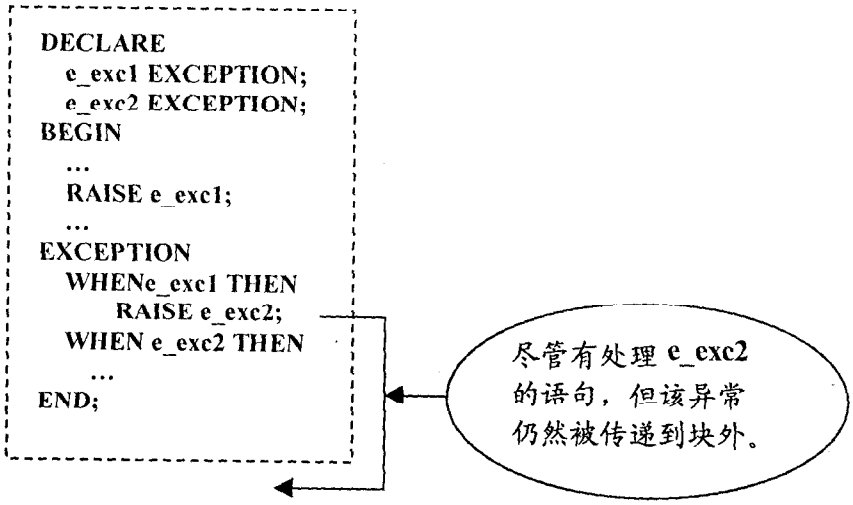


7.1.4.3 处理异常处理部分产生的异常

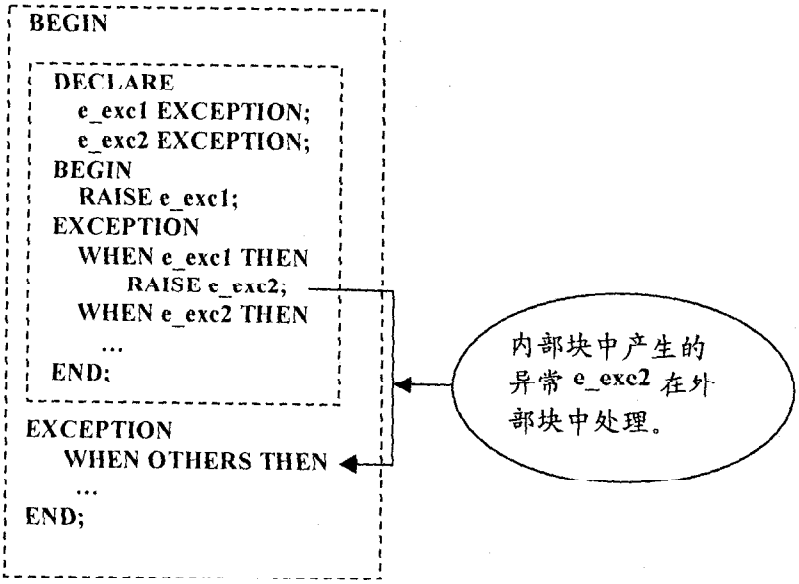
在异常处理语句中也可以产生异常，这个异常可以通过 **RAISE** 语句产生，或是由于出现一个运行错误而产生。这两种情况下产生的异常都被立刻传递到块外，这与定义部分产生的异常一样。为什么这样处理呢？因为异常部分每一次只能有一个异常被处理，当一个异常被处理时，产生了另一个异常，而一次不能同时处理多个异常，所以将异常处理部分产生的异常传递到块外。

下面举例说明异常处理部分产生的异常的处理。

例 **1**，下图中，在异常处理部分中产生了异常 **e_exc2**，同时有处理异常 **e_exc2** 的语句，但该语句没有处理异常 **e_exc2**，异常 **e_exc2** 被传递到块外：



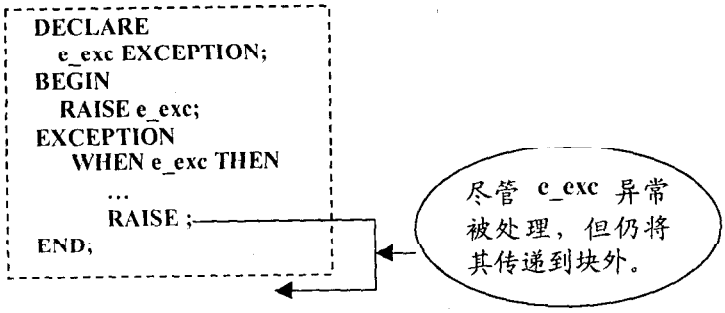
例 2，下图中，在内部块的异常处理部分产生了异常 e_exc2，同时在内部块中有处理异常 e_exc2 的语句，但异常 e_exc2 被传递到外部块中，在外部块中被处理，而且外部块成功结束：



在异常处理语句中，RAISE 语句还可以不带参数地使用。如果 RAISE 语句不带参数，则当前的异常被传递到块外。

例 3，下面块中的异常 e_exc 执行完处理它的语句后，由不带参数的 RAISE 语句将其

传递到块外：



7.1.4.4 SQLCODE 和 SQLERRM 函数

由于 **OTHERS** 子句处理 **WHEN** 子句没有处理的异常，所以在 **OTHERS** 子句中处理的异常是未知的。我们可以使用 **SQLCODE** 和 **SQLERRM** 函数来确定异常对应的错误代码和信息。

SQLCODE 返回异常对应的错误代码，**SQLERRM** 返回的是对应的错误信息。下表是异常和对应的 **SQLCODE**、**SQLERRM** 值：

异常种类	SQLCODE	SQLERRM
Oracle 错误对应的异常	负数	Oracle 错误
NO_DATA_FOUND	+100	No data found
用户自定义异常	+1	User-Defined Exception
没有产生异常	0	ORA-0000 : normal , successful completion

注意，如果使用 **EXCEPTION_INIT** 预编译指令声明与 **Oracle** 错误相连的自定义异常，则 **SQLCODE** 和 **SQLERRM** 返回对应的 **Oracle** 错误代码和相应的错误信息，而不是返回“+1”和“User-Defined Exception”。

调用 **SQLERRM** 时可以带一个数字参数，返回值是与这个数字参数相关的文本。例如，如果 **SQLERRM** 的参数是 **0**，则返回的消息是“ORA-0000 : normal , successful completion”。

例如，下面是一个带有完整的 **OTHERS** 异常处理语句的 **PL/SQL** 块：

```
DECLARE
  -- 用来指出一个错误条件的异常。
  e_TooSmallSalary EXCEPTION ;
  -- 当前作家的工资。
  v_CurrentSalary NUMBER(8,2);
  -- 作家工资的最低限。
  v_SmallSalary NUMBER(8,2) DEFAULT 100;

  v_ErrorCode NUMBER;    -- 获得错误消息代码的变量。
  v_ErrorText VARCHAR2(200); -- 获得错误消息文本的变量。

BEGIN
  /* 查找“A00002”作家的工资。*/
  SELECT salary
    INTO v_CurrentSalary
    FROM auths
    WHERE author_code = 'A00002';
  /* 检查已查出的作家的工资。*/
  IF v_CurrentSalary < v_SmallSalary THEN
    /* 该作家的工资太少。*/
    RAISE e_TooSmallSalary;
  END IF;
EXCEPTION
      WHEN e_TooSmallSalary THEN
        /* 如果查出的作家的工资小于工资的最小限，则执行删除
           作家记录的语句。*/
         DELETE auths
         where author_code = 'A00002';
      WHEN OTHERS THEN
        /* 如果是其它的错误就执行该处理。*/
         v_ErrorCode := SQLCODE;
         v_ErrorText := SUBSTR(SQLERRM, 1, 200);
         DBMS_OUTPUT.PUT_LINE(v_ErrorCode);  -- 显示错误代码。
         DBMS_OUTPUT.PUT_LINE(v_ErrorText);   -- 显示错误文本。
END;
```

前面的例子中，如果要在 SQL 语句中使用 SQLCODE 和 SQLERRM，则一定要先

把它们的值赋给局部变量，然后再将这些局部变量用在 SQL 语句中。因为这些函数是过程性的，不能直接用在 SQL 语句中。