第二部分　非对象功能

第4章　创建子程序和包

PL/SQL块主要有两种类型，即命名块和匿名块。匿名块（以 DECLARE或BEGIN开始）每次使用时都要进行编译，除此之外，该类块不在数据库中存储并且不能直接从其他的 PL/SQL块中调用。我们在本章以及下面两章中介绍的块结构，如过程，函数，包和触发器都属于命名块，这类构造没有匿名块的限制，它们可以存储在数据库中并在适当的时候运行。我们将在本章探讨创建过程，函数，以及包的语法。在第 5章，我们将介绍如何使用这些块结构和这些构造的实现，而在第6章集中介绍数据库触发器的功能。

4.1　过程和函数

PL/SQL的过程和函数的运行方式非常类似于其他 3GL(第3代程序设计语言)使用的过程和函数。它们之间具有许多共同的特征属性。总起来说，过程和函数统称为子程序。下面的代码就是一个在数据库中创建一个过程的例子：

节选自在线代码AddNewStudent.sql

```
CREATE OR REPLACE PROCEDURE AddNewStudent (
  p_FirstName students.first_name%TYPE,
  p_LastName students.last_name%TYPE,
  p_Major students.major%TYPE) AS
BEGIN
  -- Insert a new row in the students table. Use
  -- student_sequence to generate the new student ID, and
  -- 0 for current_credits.
INSERT INTO students (ID, first_name, last_name,
                      major, current_credits)
  VALUES (student_sequence.nextval, p_FirstName, p_LastName,
          p_Major, 0);
END AddNewStudent;
```

注意　表students以及我们在第 1章中描述的其他关系表，都可以用在线文档中的脚本relTables.sql来创建。

一旦创建了该过程，我们就可以从其他的 PL/SQL块中对其进行调用：

节选自在线代码AddNewStudent.sql

```
BEGIN
  AddNewStudent('Zelda', 'Zudnik', 'Computer Science');
```

```
END;
```

从该例中，我们可以总结如下要点：

- 过程AddNewStudent首先是用语句CREATE OR REPLACE PROCEDURE创建的。当该过程创建后，首先对其进行编译，接着将其按编译后的格式存储在数据库中。这种编译后生成的代码可以从另外一个PL/SQL块中运行。（该过程的源码也可以存储。有关过程源码的详细介绍，请参阅5.1.1节。）
- 当调用该过程时，可以向该过程传递参数。在上面的例子中，新生的名和姓，以及专业都在运行时作为参数传递给该过程。在该过程内部，参数 p_FirstName将具有值‘ Zelda ’，p_LastName的值是‘ Zudnik ’，而p_Major的值为‘ Computer Science ’，这些字符串都是在调用时传递给该过程的。
- 过程调用本身也是一个PL/SQL语句。过程不能作为表达式的一部分进行调用。当过程被调用时，系统就把控制交给该过程的第一个可执行语句。当过程结束时，控制就将返回到调用语句的下一个语句。在这点上，PL/SQL过程非常类似于其他3GL语言的过程调用方式。函数可以作为表达式的一部分进行调用，我们将在本节的稍后部分介绍函数的特点。
- 过程也是PL/SQL块，它由声明部分，可执行代码部分和异常处理部分组成。对于匿名块，只需要可执行部分。上面的过程AddNewStudent只有可执行部分。

4.1.1 创建子程序

类似于数据字典中的其他类型的对象，子程序是使用 CREATE语句创建的。如过程是用语句CREATE PROCEDURE创建的，函数的创建语句则是CREATE FUNCTION。下面我们分别介绍这些创建语句。

1. 创建过程

创建过程语句的语法如下所示：

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [ ( argument[{IN | OUT | IN OUT}] type,
   ...
   argument[{IN | OUT | IN OUT}] type) ] {IS | AS}
procedure_body
```

其中procedure_name是要创建的过程名，argument是过程的参数名，type是关联参数的类型，proedure_body是构成该过程代码的 PL/SQL块。有关过程和函数的参数和关键字 IN,OUT,和IN OUT的含义，请参见4.1.3节的内容。

Oracle 8i 及更高版本

Oracle8i给每个过程参数都增加了一个附加选择项关键字 NOCOPY。有关该关键字的讨论请参见4.1.3节中的“ 按引用或按值传递参数 ”。

为了修改过程的代码，首先必须将该过程撤消，然后再重建。由于这种操作已经是开发过程的标准方式，所以关键字OR REPLACE 允许将撤消和重建这两步操作合并为一个操作。如果过程存在，首先撤消该过程，而不给出任何警告提示。（可以使用命令DROP PROCEDURE来撤消一个过程，该命令在本章4.1.2节中介绍。）如果该过程已经不存在，就可以直接创建它。如果

该过程已存在而没有关键字 OR REPLACE,则CREATE语句将返回一条Oracle错误信息“ ORA-955:该名称已被当前对象使用 ”。

和其他的CREATE语句一样，创建过程是一种DDL操作，因此，在过程创建前和创建后，都要执行一条隐式的COMMIT命令。这种操作可以通过使用关键字 IS 或AS来实现，这两个关键字是等价的。

过程体　过程体是一种带有声明部分，可执行语句部分和异常部分的 PL/SQL块。该声明部分是位于关键字IS 或AS 和关键字BEGIN之间的语句。可执行部分（该部分是必须要有的）是位于关键字BEGIN和EXCEPTION之间的语句。最后，异常部分位于关键字 EXCEPTION和关键字END之间的语句。

提示　在过程和函数中没有使用关键字DECLARE。取而代之的是关键字IS或AS。这种语法风格是PL/SQL从Ada语言中继承下来的。

综上所述，过程的结构应具有下面所示的特征：

```
CREATE OR REPLACE PROCEDURE procedure_name [ parameter_list] AS
  /* Declarative section is here */
BEGIN
  /* Executable section is here */
EXCEPTION
  /* Exception section is here */
END [ procedure_name];
```

过程名可以写在过程声明中最后一个 END语句之后。如果在该END语句之后有标识符的话，该标识符一定要与该过程名匹配。

提示　在过程的最后一个END语句的后面写上过程名是一种良好的编程风格，这样做的好处是强调了END语句和CREATE语句的匹配，同时，也使 PL/SQL编译程序能够尽早地提示BEGING-END不匹配错误。

2. 创建函数

函数类似于过程。两者都带有参数，而参数具有模式（参数和模式在4.1.3节介绍），两者都不同于带有声明、可执行以及异常处理部分的 PL/SQL块。两者都可以存储在数据库中或在块中声明。（不能存储在数据库中的过程和函数在5.1节讨论。）两者不同的是，过程调用本身是一个PL/SQL语句，而函数调用是作为表达式的一部分执行的。例如，下面的函数在指定的班级有百分之80以上满员时返回真值TRUE，否则返回假值FALSE：

节选自在线代码AlmostFull.sql

```
CREATE OR REPLACE FUNCTION AlmostFull (
  p_Department classes.department%TYPE,
  p_Course     classes.course%TYPE)
  RETURN BOOLEAN IS

  v_CurrentStudents NUMBER;
  v_MaxStudents     NUMBER;
  v_ReturnValue     BOOLEAN;
```

```
  v_FullPercent CONSTANT NUMBER := 80;
BEGIN
  -- Get the current and maximum students for the requested
  -- course.
SELECT current_students, max_students
  INTO v_CurrentStudents, v_MaxStudents
  FROM classes
  WHERE department = p_Department
  AND course = p_Course;

-- If the class is more full than the percentage given by
-- v_FullPercent, return TRUE. Otherwise, return FALSE.
IF (v_CurrentStudents / v_MaxStudents * 100) >= v_FullPercent THEN
  v_ReturnValue := TRUE;
ELSE
  v_ReturnValue := FALSE;
END IF;

  RETURN v_ReturnValue;
END AlmostFull;
```

函数AlmostFull**返回的是逻辑值。该函数可以从下面的** PL/S QL**块中调用。值得注意的是，该调用不是一个独立的语句，而只是循环中作为** IF**语句表达式的一项。**

节选自在线代码callFunction.sql

```
SQL> DECLARE
  2    CURSOR c_Classes IS
  3      SELECT department, course
  4        FROM classes;
  5 BEGIN
  6   FOR v_ClassRecord IN c_Classes LOOP
  7    -- Output all the classes which don't have very much room
  8    IF AlmostFull(v_ClassRecord.department,
  9                  v_ClassRecord.course) THEN
 10     DBMS_OUTPUT.PUT_LINE(
 11       v_ClassRecord.department || ' ' ||
 12       v_ClassRecord.course || ' is almost full!');
 13   END IF;
 14  END LOOP;
 15 END;
 16 /
MUS 410 is almost full!
PL/SQL procedure successfully completed.
```

函数的语法　创建存储函数的语法非常类似于过程的语法。其定义如下：

```
CREATE [OR REPLACE] FUNCTION function_name
 [( argument[{IN | OUT | IN OUT}] type,
  ...
argument[{IN | OUT | IN OUT}] type)]
```

```
RETURN return_type{IS | AS}
function_body
```

其中function_name是函数的名称，参数argument和type的含义与过程相同，return_type是函数返回值的类型，function_body是包括函数体的PL/SQL块。

与过程的参数类似，函数的参数表是可选的，并且函数声明部分和函数调用都没有使用括弧。然而，由于函数调用是表达式的一部分，所以函数返回类型是必须要有的。函数的类型可以用来确定包含函数调用的表达式的类型。

Oracle 8i 及更高版本

像过程一样，Oracle8i为函数的参数提供了关键字 NOCOPY。本章“按引用和按值传递参数”一节将介绍该关键字。

返回语句　在函数体内，返回语句用来把控制返回到到调用环境中。该语句的通用语法如下：

```
RETURN expression;
```

其中expression是返回值。当该语句执行时，如果表达式的类型与定义不符，该表达式将被转换为函数定义子句RETURN中指定的类型。同时，控制将立即返回到调用环境。

尽管函数每次只有一个返回语句被执行，但是，函数中可以有一个以上的返回语句。如果函数结束时还没有遇到返回语句，就会发生错误。下面的例子介绍了一个函数中有多个返回语句的情况。尽管该函数中有五个不同的返回语句，但只有一个被执行，而执行哪个返回语句则取决于变量p_Department和p_Course的取值情况。

节选自在线代码ClassInfo.sql

```
CREATE OR REPLACE FUNCTION ClassInfo(
  /* Returns 'Full' if the class is completely full,
     'Some Room' if the class is over 80% full,
     'More Room' if the class is over 60% full,
     'Lots of Room' if the class is less than 60% full, and
     'Empty' if there are no students registered. */
  p_Department classes.department%TYPE,
  p_Course     classes.course%TYPE)
  RETURN VARCHAR2 IS

  v_CurrentStudents NUMBER;
  v_MaxStudents     NUMBER;
  v_PercentFull     NUMBER;
BEGIN
  -- Get the current and maximum students for the requested
  -- course.
SELECT current_students, max_students
  INTO v_CurrentStudents, v_MaxStudents
  FROM classes
  WHERE department = p_Department
  AND course = p_Course;

  -- Calculate the current percentage.
```

```
    v_PercentFull := v_CurrentStudents / v_MaxStudents * 100;

   IF v_PercentFull = 100 THEN
    RETURN 'Full';
   ELSIF v_PercentFull > 80 THEN
    RETURN 'Some Room';
   ELSIF v_PercentFull > 60 THEN
    RETURN 'More Room';
   ELSIF v_PercentFull > 0 THEN
    RETURN 'Lots of Room';
   ELSE
    RETURN 'Empty';
   END IF;
  END ClassInfo;
```

当在函数中使用返回语句时，返回语句必须带有表达式。同样，返回语句也可以用在过程中。但在过程中使用的返回语句没有参数，它只是立即把控制返回到调用环境中。这时，声明为OUT或IN OUT的形式参数的当前值将被传递回对应的实参，程序从调用过程语句的下一行继续执行（本章4.1.3节将介绍参数的详细内容。）

4.1.2 过程和函数的撤消

与表的撤消相类似，过程和函数也可以撤消。撤消操作是将过程或函数从数据字典中删除。撤消过程的语法如下：

```
DROP PROCEDURE procedure_name;
```

撤消函数的语法是：

```
DROP FUNCTION function_name;
```

其中procedure_name是现行的过程名，function_name则是现行函数名。例如，下面的语句将撤消过程AddNewStudent:

```
DROP PROCEDURE AddNewStudent;
```

如果要撤消的对象是函数的话，就必须使用语句 DROP FUNCITON,如果是过程，就使用DROP PROCEDURE。象语句CREATE一样，DROP语句也是DDL命令，因此在该语句执行前后都要隐式地执行 COMMIT命令。如果指定的子程序不存在的话，则 DROP语句将引发错误“ ORA-4043: 对象不存在. ”。

4.1.3 子程序参数

与其他类型的3GL语言一样，我们可以创建带参数的过程和函数。这些参数可以是不同的模式，并可以按值或按引用传递。下面我们来介绍这类参数的特性。

1. 参数模式

以上面使用的过程AddNewStudent为例，我们可以从下面的PL/SQL匿名块中调用该过程：

节选自在线代码callANS.sql

```
DECLARE
  -- Variables describing the new student
  v_NewFirstName students.first_name%TYPE := 'Cynthia';
  v_NewLastName  students.last_name%TYPE := 'Camino';
  v_NewMajor     students.major%TYPE := 'History';
BEGIN
  -- Add Cynthia Camino to the database.
  AddNewStudent(v_NewFirstName, v_NewLastName, v_NewMajor);
END;
```

该块声明的变量（v_NewFirstName,v_NewLastName,v_NewMajor）作为参数传递给过程AddNewStudent。在这种上下文中，我们把这些参数称为实参，而在过程声明部分中的参数（p_FirstName，p_LastName,p_Major）则称为形参。实参包含了过程被调用时传递过来的值，并且实参还接收过程返回时的结果（与返回模式有关）。实参的值是过程中将要使用的值。当调用过程时，形参被赋予实参的值。对于过程内部而言，实参是由形参引用的。当过程结束时，实参被赋予形参的值。上述的赋值操作（包括类型转换）必要时将遵循 PL/SQL的一般赋值规则。

形参可以有三种模式，IN，OUT或IN OUT。（Oracle8i增加了NOCOPY限定符，该参数在本章“使用NOCOPY”一节介绍。）如果没有为形参指定模式，其默认模式为 IN。表4-1说明了模式间的区别，下面的例子也使用了不同的参数模式：

表4-1 参数模式

模 式	说 明
IN	当过程被调用时，实参的值将传入该过程。在该过程内部，形参类似 PL/SQL使用的常数，即该值具有只读属性不能对其修改。当该过程结束时，控制将返回到调用环境，这时，对应的实参没有改变。
OUT	当过程被调用时，实参具有的任何值将被忽略不计。在该过程内部，形参的作用类似没有初始化的PL/SQL变量，其值为空（NULL）。该变量具有读写属性。当该过程结束时，控制将返回调用环境，形参的内容将赋予对应的实参。（在Oracle8i中，该操作可由NOCOPY变更。有关NOCOPY的详细内容，请看本章“按引用和按值传递参数”一节。）
IN OUT	该模式是模式IN 和OUT的组合。当调用过程时，实参的值将被传递到该过程中。在该过程内部，形参相当于初始化的变量，并具有读写属性。当该过程结束时，控制将返回到调用环境中，形参的内容将赋予实参（在 Oracle8i中与参数NOCOPY有关）。

注意 例子ModeTest演示了合法与不合法的PL/SQL赋值操作。如果将注释为非法语句的注释符删除，该程序将出现编译错误。

节选自在线代码ModeTest.sql

```
CREATE OR REPLACE PROCEDURE ModeTest (
  p_InParameter    IN NUMBER,
  p_OutParameter   OUT NUMBER,
  p_InOutParameter IN OUT NUMBER) IS

  v_LocalVariable NUMBER := 0;
BEGIN
```

```
  DBMS_OUTPUT.PUT_LINE('Inside ModeTest:');
  IF (p_InParameter IS NULL) THEN
    DBMS_OUTPUT.PUT('p_InParameter is NULL');
  ELSE
    DBMS_OUTPUT.PUT('p_InParameter = ' || p_InParameter);
  END IF;

  IF (p_OutParameter IS NULL) THEN
    DBMS_OUTPUT.PUT(' p_OutParameter is NULL');
  ELSE
    DBMS_OUTPUT.PUT(' p_OutParameter = ' || p_OutParameter);
  END IF;

  IF (p_InOutParameter IS NULL) THEN
    DBMS_OUTPUT.PUT_LINE(' p_InOutParameter is NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE(' p_InOutParameter = ' ||
                         p_InOutParameter);
  END IF;

  /* Assign p_InParameter to v_LocalVariable. This is legal,
     since we are reading from an IN parameter and not writing
     to it. */
v_LocalVariable := p_InParameter; -- Legal

/* Assign 7 to p_InParameter. This is ILLEGAL, since we
   are writing to an IN parameter. */
-- p_InParameter := 7; -- Illegal

/* Assign 7 to p_OutParameter. This is legal, since we
   are writing to an OUT parameter. */
p_OutParameter := 7; -- Legal

/* Assign p_OutParameter to v_LocalVariable. In Oracle7 version
   7.3.4, and Oracle8 version 8.0.4 or higher (including 8i),
   this is legal. Prior to 7.3.4, it is illegal to read from an
   OUT parameter. */
v_LocalVariable := p_OutParameter; -- Possibly illegal

/* Assign p_InOutParameter to v_LocalVariable. This is legal,
   since we are reading from an IN OUT parameter. */
v_LocalVariable := p_InOutParameter; -- Legal

/* Assign 8 to p_InOutParameter. This is legal, since we
   are writing to an IN OUT parameter. */
p_InOutParameter := 8; -- Legal

DBMS_OUTPUT.PUT_LINE('At end of ModeTest:');
IF (p_InParameter IS NULL) THEN
  DBMS_OUTPUT.PUT('p_InParameter is NULL');
ELSE
```

```
 DBMS_OUTPUT.PUT('p_InParameter = ' || p_InParameter);
END IF;

IF (p_OutParameter IS NULL) THEN
 DBMS_OUTPUT.PUT(' p_OutParameter is NULL');
ELSE
 DBMS_OUTPUT.PUT(' p_OutParameter = ' || p_OutParameter);
END IF;

IF (p_InOutParameter IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE(' p_InOutParameter is NULL');
ELSE
 DBMS_OUTPUT.PUT_LINE(' p_InOutParameter = ' ||
                      p_InOutParameter);
END IF;

END ModeTest;
```

注意 在Oracle7.3.4版之前，以及 8.0.3版中，从参数 OUT读取是非法操作，但在Oracle8的8.0.4版及更高版本中，该操作是合法的。详细介绍请看下文“从参数 OUT读取”。)

2. 在形参和实参之间传递值

我们使用下面的块来调用过程 ModeTest：

节选自在线代码callMT.sql

```
DECLARE
  v_In NUMBER := 1;
  v_Out NUMBER := 2;
  v_InOut NUMBER := 3;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before calling ModeTest:');
  DBMS_OUTPUT.PUT_LINE('v_In = ' || v_In ||
                       ' v_Out = ' || v_Out ||
                       ' v_InOut = ' || v_InOut);

ModeTest(v_In, v_Out, v_InOut);

DBMS_OUTPUT.PUT_LINE('After calling ModeTest:');
DBMS_OUTPUT.PUT_LINE(' v_In = ' || v_In ||
                     ' v_Out = ' || v_Out ||
                     ' v_InOut = ' || v_InOut);
END;
```

该调用生成的输出信息如下：

```
Before calling ModeTest:
v_In = 1 v_Out = 2 v_InOut = 3
Inside ModeTest:
p_InParameter = 1 p_OutParameter is NULL p_InOutParameter = 3
At end of ModeTest:
p_InParameter = 1 p_OutParameter = 7 p_InOutParameter = 8
```

```
After calling ModeTest:
v_In = 1 v_Out = 7 v_InOut = 8
```

该输出信息显示了在该过程内部OUT参数已经被初始化为NULL。同样，当该过程结束运行时，在该过程结尾处的形参IN和IN OUT的值也被复制给了对应的实参。

注意 如果该过程引发了异常，则形参IN OUT和OUT的值不会被复制到对应的实参中（在Oracle8i中，该功能与NOCOPY参数有关）。请读者看下文“子程序内部引发的异常”内容。

直接量或常数作为实参 因为复制功能的使用，对应于参数IN OUT或OUT的实参必须是变量，而不能是常数或表达式。也就是说，程序必须提供返回的变量的存储位置。例如，我们可以在调用过程ModeTest时用直接量来取代变量v_In.:

节选自在线代码callMT.sql

```
DECLARE
  v_Out NUMBER := 2;
  v_InOut NUMBER := 3;
BEGIN
  ModeTest(1, v_Out, v_InOut);
END;
```

但是如果用直接量来取代变量v_Out，就会发生下列错误：

节选自在线代码callMT.sql

```
SQL> DECLARE
  2   v_InOut NUMBER := 3;
  3 BEGIN
  4   ModeTest(1, 2, v_InOut);
  5 END;
  6 /
DECLARE
*
ERROR at line 1:
ORA-06550: line 4, column 15:
PLS-00363: expression '2' cannot be used as an assignment target
ORA-06550: line 4, column 3:
PL/SQL: Statement ignored
```

编译检查 PL/SQL编译器在创建过程时将对合法的赋值进行检查。例如，如果我们把对p_InParameter的赋值语句的注释去掉的话，编译器将报告过程ModeTest有下列错误：

```
PLS-363: expression 'P_INPARAMETER' cannot be used as an
         assignment target
```

从参数OUT读取 在Oracle7.3.4以前的版本以及8.0.3版下，对过程中OUT参数进行读操作是非法的。如果我们在8.0.3版数据库下编译过程ModeTest的话，编译器将报告下列错误：

```
PLS-00365: 'P_OUTPARAMETER' is an OUT parameter and cannot be read
```

解决该问题的方法是声明OUT参数为IN OUT模式。表4-2列出了允许对OUT参数进行读操作的Oracle版本。

表4-2 允许读操作的版本

Oracle 版本	读OUT参数
7.3.4之前的版本	不能
7.3.4版	可以
8.0.3	不能
8.0.4及更高版本	可以

3. 对形参的限制

调用过程时，实参的值将被传入该过程，这些实参在该过程内部以引用的方式使用形参。同时，作为参数传递机制一部分，对变量的限制也传递给该过程。在过程的声明中，强制指定参数CHAR和VARCHAR2的长度，以及指定NUMBER参数的精度或小数点后位数都是非法的，这是因为这些限制可以从实参中获得。例如，下面的过程声明就是非法的并将引发编译错误：

节选自在线代码ParameterLenght.sql

```
CREATE OR REPLACE PROCEDURE ParameterLength (
  p_Parameter1 IN OUT VARCHAR2(10),
  p_Parameter2 IN OUT NUMBER(3,1)) AS
BEGIN
  p_Parameter1 := 'abcdefghijklm';
  p_Parameter2 := 12.3;
END ParameterLength;
```

正确的声明应该是下面的代码：

节选自在线代码ParameterLenght.sql

```
CREATE OR REPLACE PROCEDURE ParameterLength (
  p_Parameter1 IN OUT VARCHAR2,
  p_Parameter2 IN OUT NUMBER) AS
BEGIN
  p_Parameter1 := 'abcdefghijklmno';
  p_Parameter2 := 12.3;
END ParameterLength;
```

因此，是谁对形参p_Parameter1和p_Parameter2有限制呢?从上面的例子中，我们可以看出正是实参对形参施加了限制。如果我们使用下面的代码调用过程ParameterLength的话：

节选自在线代码ParameterLength.sql

```
DECLARE
  v_Variable1 VARCHAR2(40);
  v_Variable2 NUMBER(7,3);
BEGIN
  ParameterLength(v_Variable1, v_Variable2);
  END;
```

则p_Parameter1的最大长度为40(该长度从实参v_Variable1继承而来)，而p_Parameter2的精度为7，小数点后位数为3(从实参v_Varible继承而来)。在进行参数声明时，一定要注意上述特点。现在，请考虑下面的程序块，该块也调用ParameterLenght:

节选自在线代码ParameterLenght.sql

```
DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,3);
BEGIN
  ParameterLength(v_Variable1, v_Variable2);
END;
```

上述两个块的唯一不同之处是第一块的参数 v_Variable1,也就是p_Parameter1的长度为10,而不是40。由于过程ParameterLenght要把长度为15的字符串赋给p_Parameter1(即v_Variable1),而该参数没有足够的长度。当调用该过程时，将导致下面的错误：

```
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "EXAMPLE.PARAMETERLENGTH", line 5
ORA-06512: at line 5
```

该错误的根源不在该过程中，而是与调用该过程的代码有关。除此之外，错误 ORA-6502是运行错误，而不是编译错误。因此，该块已经通过了编译，该错误是在该过程返回时发生的，错误的原因是PL/SQL引擎企图把字符串‘ adcdefghijklmno ’复制到形参中。

提示 为了避免如ORA-6502之类的错误，应在创建过程时，在文档中记录实参的任何限制要求。该文档应由存储在过程中的注释组成，并要标明该过程的功能，以及所有参数的定义。

%类型和过程参数 尽管对形参不能进行强制声明，但我们可以使用 %类型来说明形参。如果一个形参是用%类型声明的，并且说明%类型的变量也是强制类型的话，则该强制说明将作用在形参上而不是实参上。如果我们在过程 ParameterLength中使用了下面的说明：

节选自在线代码ParameterLength.sql

```
CREATE OR REPLACE PROCEDURE ParameterLength (
  p_Parameter1 IN OUT VARCHAR2,
  p_Parameter2 IN OUT students.current_credits%TYPE) AS
BEGIN
  p_Parameter2 := 12345;
  END ParameterLength;
```

由于列current_credits的精度是3,所以，上面程序中的 P_Parameter2的精度也被限制为3位。即使我们用具有足够精度的实参来调用该过程，其形参的精度也是 3位。因此下列的过程将生成ORA-6502错误：

节选自在线代码ParameterLenght.sql

```
SQL> DECLARE
  2    v_Variable1 VARCHAR2(1);
  3    -- Declare v_Variable2 with no constraints
  4    v_Variable2 NUMBER;
  5 BEGIN
  6    -- Even though the actual parameter has room for 12345, the
  7    -- constraint on the formal parameter is taken and we get
  8    -- ORA-6502 on this procedure call.
```

```
  9    ParameterLength(v_Variable1, v_Variable2);
  10 END;
  11 /
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: number precision too large
ORA-06512: at "EXAMPLE.PARAMETERLENGTH", line 5
ORA-06512: at line 9
```

4. 子程序内部引发的异常

如果错误发生在子程序的内部，就会引发异常。该异常即可以是由用户定义的，也可以是程序中予定义的。如果引发异常的过程中没有处理该错误的异常处理程序（或该异常发生在该异常处理程序的内部。），根据异常的传播规则（请看 PL/SQL用户指南的有关章节），控制将立即转出该过程返回其调用环境。然而，在本例的情况下，形参 OUT和IN OUT的值并没有返回到实参。这些实参仍将被设置为调用前的值。例如，假设我们创建下面的过程：

节选自在线代码RaiseError.sql

```
/* Illustrates the behavior of unhandled exceptions and
 * OUT variables. If p_Raise is TRUE, then an unhandled
 * error is raised. If p_Raise is FALSE, the procedure
 * completes successfully.
 */
 CREATE OR REPLACE PROCEDURE RaiseError (
  p_Raise IN BOOLEAN,
  p_ParameterA OUT NUMBER) AS
 BEGIN
  p_ParameterA := 7;

 IF p_Raise THEN
  /* Even though we have assigned 7 to p_ParameterA, this
   * unhandled exception causes control to return immediately
   * without returning 7 to the actual parameter associated
   * with p_ParameterA.
   */
  RAISE DUP_VAL_ON_INDEX;
 ELSE
  -- Simply return with no error. This will return 7 to the
  -- actual parameter.
  RETURN;
 END IF;
END RaiseError;
```

现在，如果我们用下面的块调用 RaiseError的话：

节选自在线代码RaiseError.sql

```
DECLARE
  v_TempVar NUMBER := 1;
BEGIN
```

```
  DBMS_OUTPUT.PUT_LINE('Initial value: ' || v_TempVar);
  RaiseError(FALSE, v_TempVar);
  DBMS_OUTPUT.PUT_LINE('Value after successful call: ' ||
                       v_TempVar);

  v_TempVar := 2;
  DBMS_OUTPUT.PUT_LINE('Value before 2nd call: ' || v_TempVar);
RaiseError(TRUE, v_TempVar);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Value after unsuccessful call: ' ||
                         v_TempVar);
END;
```

将得到下面所示的输出：

```
 Initial value: 1
Value after successful call: 7
Value before 2nd call: 2
Value after unsuccessful call: 2
```

在第一次调用RaiseError**之前，变量**v_TempVar**的值为**1.**在第一次调用成功结束后，该变量的值为**7。**接着，在第二次调用**RaiseError**前，该块将变量**v_TempVar**的值修改为**2。**由于第二次调用没有成功，变量**v_TempVar**的值仍然是**2(**而不是被再次赋予值**7)。**

Oracle 8i 及更高版本

当声明参数OUT**或**IN OUT**时使用了**NOCOPY**选项时，异常处理的语义将发生变化。有关该选项的使用方法，请看下文“使用**NOCOPY**时的异常语义”内容。**

5. 按引用和按值传递参数

子程序参数可以按下列两种方式传递，即按引用，或按值传递。当参数是按引用传递时，一个指向实参的指针将被传递到对应的形参。当参数是按值传递时，实参的值将被赋予对应的形参。按引用传递的效率要比按值传递的效率要高，这是因为按引用传递不涉及复制操作。按引用传递特别适合于处理集合类型参数，如表，数组等。 PL/SQL**的默认方式是对参数** IN**进行按引用传递，而对参数**OUT**，**IN OUT**执行按值传递。采用这种参数传递规则是为了与我们在前一节讨论的异常语义保持一致，以便可以对实参的强制说明进行验证。** Oracle8i**以前的版本不支持对引用方式的修改。**

6. 使用NOCOPY参数

Oracle 8i 及更高版本

Oracle8i**提供了一个叫做**NOCOPY**的编译选项。使用该项声明参数的语法如下：**

```
parameter_name [mode] NOCOPY datatype
```

其中parameter_name**是参数名，**mode**是参数的模式（**IN **，**OU **，**IN OUT**），**datatype**是参数的数据类型。如果使用了**NOCOPY**，则**PL/SQL**编译器将按引用传递参数，而不按值传递。由于**NOCOPY**是一个编译选项，而非指令，所以该选项不会大量使用。下面的例子介绍了** NOCOPY **的使用方法：**

节选自在线代码NoCopyTest.sql

```
CREATE OR REPLACE PROCEDURE NoCopyTest (
  p_InParameter    IN NUMBER,
  p_OutParameter   OUT NOCOPY VARCHAR2,
  p_InOutParameter IN OUT NOCOPY CHAR) IS
BEGIN
  NULL;
END NoCopyTest;
```

对参数IN使用NOCOPY将会产生编译错误，这是因为参数 IN总是按引用传递，NOCOPY不能更改其引用方式。

使用NOCOPY时的异常语义 当参数按引用传递时，任何对实参的修改也将引起对其对应形参的修改，这是因为该实参和形参同时位于相同的存储单元的缘故。换句话说，如果过程退出时没有处理异常而形参已被修改的话，则该形参对应的实参的原始值也将被修改。假设我们在过程RaiseError中使用NOCOPY选项，该代码如下：

节选自在线代码NoCopyTest.sql

```
CREATE OR REPLACE PROCEDURE RaiseError (
  p_Raise IN BOOLEAN,
  p_ParameterA OUT NOCOPY NUMBER) AS
BEGIN
  p_ParameterA := 7;
  IF p_Raise THEN
    RAISE DUP_VAL_ON_INDEX;
  ELSE
    RETURN;
  END IF;
END RaiseError;
```

对该过程的唯一修改是指定参数 p_ParameterA**按引用传递。假设我们用下面的代码调用该** RaiseError**过程：**

节选自在线代码NoCopyTest.sql

```
DECLARE
  v_TempVar NUMBER := 1;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Initial value: ' || v_TempVar);
  RaiseError(FALSE, v_TempVar);
  DBMS_OUTPUT.PUT_LINE('Value after successful call: ' ||
                       v_TempVar);

  v_TempVar := 2;
  DBMS_OUTPUT.PUT_LINE('Value before 2nd call: ' || v_TempVar);
  RaiseError(TRUE, v_TempVar);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Value after unsuccessful call: ' ||
                         v_TempVar);
END;
```

（上面的代码块与我们在4.1.3节中使用的代码一样。）

如下所示，现在该块的输出与前面使用的块的输出有所不同：

```
Initial value: 1
Value after successful call: 7
Value before 2nd call: 2
Value after unsuccessful call: 7
```

可以看出，即使该块中引发了异常，其实参也被修改了两次。

使用NOCOPY的限制 在某些情况下，NOCOPY将被编译器忽略，这时的参数仍将按值传递。这时，编译器不会报告编译错误。由于NOCOPY是一个提示项（Hint），编译器可以决定是否执行该项。在下列情况下，编译器将忽略NOCOPY项：

- 实参是索引表（index-by table）的成员时。如果该实参是全表，则该限制不起作用。
- 实参被强制指定精度，比例或NOT NULL时。该限制将不适用按最大长度强制的字符串参数。
- 实参和形参都是记录类型，二者是以隐含方式或使用了%ROWTYPE类型声明时，作用在对应字段的强制说明不一致。
- 传递实参需要隐式类型转换时。
- 子程序涉及到远程过程调用（PRC）。远程过程调用就是跨越数据库对远程服务器的过程调用。

提示 对于最后一条来说，如果子程序是PRC的一部分，则NOCOPY将被忽略。如果对现存的应用进行修改使其具有远程调用命令的话，则异常处理的语义将随之改变。

使用NOCOPY的优点 NOCOPY的主要优点是可以提高程序的效率。当我们传递大型PL/SQL表时，其优越性特别显著。请看下面的例子：

节选自在线代码CopyFast.sql

```
CREATE OR REPLACE PACKAGE CopyFast AS
  -- PL/SQL table of students.
  TYPE StudentArray IS
    TABLE OF students%ROWTYPE;

  -- Three procedures which take a parameter of StudentArray, in
  -- different ways. They each do nothing.
  PROCEDURE PassStudents1(p_Parameter IN StudentArray);
  PROCEDURE PassStudents2(p_Parameter IN OUT StudentArray);
  PROCEDURE PassStudents3(p_Parameter IN OUT NOCOPY StudentArray);

  -- Test procedure.
  PROCEDURE Go;
END CopyFast;

CREATE OR REPLACE PACKAGE BODY CopyFast AS
  PROCEDURE PassStudents1(p_Parameter IN StudentArray) IS
  BEGIN
```

```
    NULL;
  END PassStudents1;
PROCEDURE PassStudents2(p_Parameter IN OUT StudentArray) IS
BEGIN
  NULL;
END PassStudents2;

PROCEDURE PassStudents3(p_Parameter IN OUT NOCOPY StudentArray) IS
BEGIN
  NULL;
END PassStudents3;

PROCEDURE Go IS
  v_StudentArray StudentArray := StudentArray(NULL);
  v_StudentRec students%ROWTYPE;
  v_Time1 NUMBER;
  v_Time2 NUMBER;
  v_Time3 NUMBER;
  v_Time4 NUMBER;
BEGIN
  -- Fill up the array with 50,001 copies of David Dinsmore's
  -- record.
  SELECT *
    INTO v_StudentArray(1)
    FROM students
    WHERE ID = 10007;
  v_StudentArray.EXTEND(50000, 1);

  -- Call each version of PassStudents, and time them.
  -- DBMS_UTILITY.GET_TIME will return the current time, in
  -- hundredths of a second.
  v_Time1 := DBMS_UTILITY.GET_TIME;
  PassStudents1(v_StudentArray);
  v_Time2 := DBMS_UTILITY.GET_TIME;
  PassStudents2(v_StudentArray);
  v_Time3 := DBMS_UTILITY.GET_TIME;
  PassStudents3(v_StudentArray);
  v_Time4 := DBMS_UTILITY.GET_TIME;

  -- Output the results.
  DBMS_OUTPUT.PUT_LINE('Time to pass IN: ' ||
                       TO_CHAR((v_Time2 - v_Time1) / 100));
  DBMS_OUTPUT.PUT_LINE('Time to pass IN OUT: ' ||
                       TO_CHAR((v_Time3 - v_Time2) / 100));
  DBMS_OUTPUT.PUT_LINE('Time to pass IN OUT NOCOPY: ' ||
                       TO_CHAR((v_Time4 - v_Time3) / 100));
 END Go;
END CopyFast;
```

注意 该例使用了一个包来把相关的过程编为一组。本书的第14章介绍了集合以及方法EXTEND的用途。

过程组PassStudents中的每个过程除了接收PL/SQL表student的一个参数外没有任何其他的功能。该参数有 50 001个记录，是一个相当大的表。该过程组各过程之间的不同之处是PassStudents1是以IN模式接收其参数，而PassStudents2以IN OUT模式接收参数，PassStudents3则以IN OUT、NOCOPY的模式接收参数。因此，PassStudents2是按值传递参数，而其他两个过程则执行按引用传递参数。我们可以从下面调用CopyFast.Go的结果看到这种区别：

```
Time to pass IN: 0
Time to pass IN OUT: 4.28
Time to pass IN OUT NOCOPY: 0
```

尽管在不同的操作系统平台上，实际结果可能有所不同，但我们可以看出，按值传递 IN OUT 模式的参数所使用的时间远远大于按引用传递 IN 和IN OUT NOCOPY参数所使用的时间。

7. 不带参数的子程序

如果过程没有参数的话，就不需要在该过程调用声明中或在其过程调用中使用括弧。函数的也具有类似的情况。下面的过程代码演示了使用不带参数的过程。

节选自在线代码noparams.sql

```
CREATE OR REPLACE PROCEDURE NoParamsP AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('No Parameters!');
END NoParamsP;

CREATE OR REPLACE FUNCTION NoParamsF
  RETURN DATE AS
BEGIN
  RETURN SYSDATE;
END NoParamsF;

BEGIN
  NoParamsP;
  DBMS_OUTPUT.PUT_LINE('Calling NoParamsF on ' ||
    TO_CHAR(NoParamsF, 'DD-MON-YYYY'));
END;
```

Oracle 8i 及更高版本 在Oracle8i 的CALL调用语法下，括弧是选择项。

8. 按位置对应法和按名称对应法

到目前为止本章所举案例中，实参都与其位置对应的形参相关联。假设有下面的过程声明代码：

节选自在线代码CallMe.sql

```
CREATE OR REPLACE PROCEDURE CallMe(
  p_ParameterA VARCHAR2,
  p_ParameterB NUMBER,
  p_ParameterC BOOLEAN,
```

```
  p_ParameterD DATE) AS
BEGIN
  NULL;
END CallMe;
```

以及如下所示的过程调用：

节选自在线代码CallMe.sql

```
DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,6);
  v_Variable3 BOOLEAN;
  v_Variable4 DATE;
BEGIN
  CallMe(v_Variable1, v_Variable2, v_Variable3, v_Variable4);
END;
```

实参是按位置与形参相关联的。也就是说，v_Variable1是与P_ParameterA相关联的，v_Variable2是与p_ParameterB相关联的，依此类推，参数间的这种对应法称为按位置对应法（Positional Notation）.这种定位关系在3GL语言，以及C语言中都是常用的表达方式。

除此之外，我们还可以使用如下所示的按名称对应法（Named Notation）来调用过程：

节选自在线代码CallMe.sql

```
DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,6);
  v_Variable3 BOOLEAN;
  v_Variable4 DATE;
BEGIN
  CallMe(p_ParameterA => v_Variable1,
         p_ParameterB => v_Variable2,
         p_ParameterC => v_Variable3,
         p_ParameterD => v_Variable4);
END;
```

在按名称对应法下，形参和实参同时出现在参数的位置上。这种方法允许我们按程序的要求重新安排参数的顺序。例如，下面的块使用同样的参数调用过程 CallMe:

节选自在线代码CallMe.sql

```
DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,6);
  v_Variable3 BOOLEAN;
  v_Variable4 DATE;
BEGIN
  CallMe(p_ParameterB => v_Variable2,
         p_ParameterC => v_Variable3,
         p_ParameterD => v_Variable4,
         p_ParameterA => v_Variable1);
END;
```

如果有必要的话，按位置对应法和按名称对应法可以同时用在一个调用中。但是，该类调

用的第一个参数必须是按位置匹配的，其余的参数可以按名称指定。下面的程序中使用了这种混合调用方法：

节选自在线代码CallMe.sql

```
DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,6);
  v_Variable3 BOOLEAN;
  v_Variable4 DATE;
BEGIN
  -- First 2 parameters passed by position, the second 2 are
  -- passed by name.
  CallMe(v_Variable1, v_Variable2,
         p_ParameterC => v_Variable3,
         p_ParameterD => v_Variable4);
END;
```

按名称对应法也是PL/SQL从Ada 语言继承的一个特性。至于什么时候应该使用按名称对应，什么时候应使用按名称对应，从效率上讲，二者没有明显区别。唯一的区别是它们的风格不一样。表4-3介绍了这两种方式的不同风格。

就作者来说，我倾向于使用按位置对应法。因为我喜欢编制风格简明的代码。我认为给实参命名有意义的名称是非常重要的。但从另一方面来说，如果过程带有大量的参数（超过 10个），那么就应该考虑使用按名称对应法，因为这样做可以较好的匹配实参和形参。实际上，带有超过10个参数的过程是非常少的。

表4-3 按位置对应法与按名称对应法的对比

按位置对应法	按名称对应法
实参使用有意义的名称来说明每个参数的用途	清楚地指明了实参与形参的对应关系
用于实参和形参的参数名可以相互独立；当任意一个参数的名称修改时，不会影响程序的使用	该方式的维护工作比较多，因为如果某个形参的名称改变的话，则所有对该过程的调用中实参使用的名称都要做相应的修改
如果形参的顺序发生变化时，所有对该过程的调用的位置符号必须也做相应的调整，因此维护工作量大	形参和实参的使用顺序是独立的；参数出现的位置可以随意修改
程序的风格比命名方式更简洁	由于形参和实参都要写在调用中，所以代码的编制工作量要比按位置对应方式大一些
使用默认值的参数必须出现在参数表的最后一个	允许形参使用默认值，与参数的本身的默认值无关

提示 过程带有的参数越多，调用该过程并确认其所有参数都可用的难度就越大。如果过程带有大量的参数要传递或要接收的话，可以考虑定义记录类型将参数作为记录中的字段使用。这样一来，就可以使用一个记录类型的参数进行调用。（注意，如果调用环

境不是在PL/SQL下，可能无法对记录类型赋值。）PL/SQL对参数没有限制。

9. 参数默认值

类似于变量的声明，过程或函数的形参可以具有默认值。如果一个参数有默认值的话，该参数就可以不从调用环境中传递。如果传递了参数，则实参的值将取代默认值。参数使用默认值的语法如下：

```
parameter_name [mode] parameter_type{:= |DEFAULT} initial_value
```

其中，parameter_name是形参的名称，mode是参数使用的模式（IN，OUT，IN OUT）,parameter_type是参数类型（予定义的或用户定义的），initial_value是赋予形参的默认值。关键字:=和DEFAULT可以任选使用。例如，除非由显式说明来覆盖默认值外，我们可以重编过程AddNewStudent来把默认的经济专业值赋给所有的新生：

节选自在线代码default.sql

```
CREATE OR REPLACE PROCEDURE AddNewStudent (
  p_FirstName students.first_name%TYPE,
  p_LastName  students.last_name%TYPE,
  p_Major     students.major%TYPE DEFAULT 'Economics') AS
BEGIN
  -- Insert a new row in the students table. Use
  -- student_sequence to generate the new student ID, and
  -- 0 for current_credits.
  INSERT INTO students VALUES (student_sequence.nextval,
    p_FirstName, p_LastName, p_Major, 0);
END AddNewStudent;
```

如上所示，如果形参p_Major在过程调用中没有实参与其关联的话，该参数就使用其默认值。我们也可以使用按位置对应法来实现上述功能：

节选自在线代码default.sql

```
BEGIN
  AddNewStudent('Simon', 'Salovitz');
END;
```

或使用如下按名称对应法：

节选自在线代码default.sql

```
BEGIN
  AddNewStudent(p_FirstName => 'Veronica',
                p_LastName => 'Vassily');
END;
```

如果使用了按位置对应法，则所有带有默认值的没有与实参相关联的参数就必须位于该参数表的尾端。请看下面的代码：

节选自在线代码DefaultTtest.sql

```
CREATE OR REPLACE PROCEDURE DefaultTest (
  p_ParameterA NUMBER DEFAULT 10,
  p_ParameterB VARCHAR2 DEFAULT 'abcdef',
  p_ParameterC DATE DEFAULT SYSDATE) AS
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(
   'A: ' || p_ParameterA ||
   ' B: ' || p_ParameterB ||
   ' C: ' || TO_CHAR(p_ParameterC, 'DD-MON-YYYY'));
END DefaultTest;
```

上面代码中，过程 DefaultTest的三个参数都使用了默认值。如果我们只希望参数p_ParameterB使用默认值，而参数p_ParameterA和p_ParameterC都使用指定的值，我们最好使用按名称对应法，其代码如下：

节选自在线代码DefaultTtest.sql

```
SQL> BEGIN
  2 DefaultTest(p_ParameterA => 7, p_ParameterC => '30-DEC-95');
  3 END;
  4 /
A: 7 B: abcdef C: 30-DEC-1995
PL/SQL procedure successfully completed.
```

在使用按位置对应法的情况下，如果我们要 p_ParameterB使用默认值，我们就必须使p_ParameterC也用默认值，没有与实参关联的所有的默认参数都必须位于参数表的最后，其代码如下所示：

节选自在线代码DefaultTest.sql

```
SQL> BEGIN
  2    -- Uses the default value for both p_ParameterB and
  3    -- p_ParameterC.
  4    DefaultTest(7);
  5 END;
  6 /
A: 7 B: abcdef C: 17-OCT-1999
PL/SQL procedure successfully completed.
```

提示 在使用默认值时，尽量把它们放置在参数表的最后位置。在这种方式下，其他参数即可以使用按位置对应法也可以使用按名称对应法。

4.1.4 过程与函数的比较

过程和函数有许多相同的特点：

- 通过设置OUT参数，过程和函数都可以返回一个以上的值。
- 过程和函数都可以具有声明部分，执行部分，以及异常处理部分。
- 过程和函数都可以接收默认值。
- 都可以使用位置或名称对应法调用过程和函数。
- 过程和函数都可以接收参数NOCOPY（仅Oracle8i支持）。

综上所述，对于什么时候使用函数更合适，什么时候使用过程更有效这一问题，一般来说，过程和函数的使用与子程序将要返回的值的数量以及这些值的使用方法有关。通常，如果返回

值在一个以上时，用过程为好。如果只有一个返回值，使用函数就可以满足要求。尽管函数可以合法地使用参数OUT（因此可以返回一个以上的值），但这种做法通常不予考虑。除此之外，函数还可以从SQL语句中调用。（请看第5章的有关内容。）

4.2 包

集成在PL/SQL语言中的另一个Ada语言特性是包的概念。包是由存储在一起的相关对象组成的PL/SQL结构。包有两个独立的部分，即说明部分和包体，这两部分独立地存储在数据字典中。与可以位于本地块或数据库中的过程和函数不同，包只能存储；并且不能在本地存储。除了允许相关的对象结为组之外，包与依赖性较强的存储子程序相比，其所受的限制较少。除此之外，包的效率比较高（我们将在第5章讨论包的效率问题。）

从本质上讲，包就是一个命名的声明部分。任何可以出现在块声明中的语句都可以在包中使用，这些语句包括过程，函数，游标，类型以及变量。把上述内容放入包中的好处是我们可以从其他PL/SQL块中对其进行引用，因此包为PL/SQL提供了全程变量。

4.2.1 包的说明

包的说明（也叫做包头）包含了有关包内容的信息。然而，该部分中不包括包的代码部分。下面是一个包的说明部分：

节选自在线代码ClassPackage.sql

```
CREATE OR REPLACE PACKAGE ClassPackage AS
  -- Add a new student into the specified class.
  PROCEDURE AddStudent(p_StudentID  IN students.id%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course     IN classes.course%TYPE);

  -- Removes the specified student from the specified class.
  PROCEDURE RemoveStudent(p_StudentID  IN students.id%TYPE,
                          p_Department IN classes.department%TYPE,
                          p_Course     IN classes.course%TYPE);

  -- Exception raised by RemoveStudent.
  e_StudentNotRegistered EXCEPTION;

  -- Table type used to hold student info.
  TYPE t_StudentIDTable IS TABLE OF students.id%TYPE
    INDEX BY BINARY_INTEGER;

  -- Returns a PL/SQL table containing the students currently
  -- in the specified class.
  PROCEDURE ClassList(p_Department IN classes.department%TYPE,
                      p_Course     IN classes.course%TYPE,
                      p_IDs        OUT t_StudentIDTable,
                      p_NumStudents IN OUT BINARY_INTEGER);
END ClassPackage;
```

上面的包说明ClassPackage包括三个过程，一个类型说明和一个异常说明。创建包头的语法

如下所示：

```
CREATE [OR REPLACE] PACKAGE package_name{IS | AS}
  type_definition|
  procedure_specification|
  function_specification|
  variable_declaration|
  exception_declaration|
  cursor_declaration|
  pragma_declaration
END [ package_name];
```

其中，package_name是包的名称。该包内的各种元素的说明语法（即过程说明，函数说明，变量说明等）与匿名块中的同类元素的的说明使用的语法完全相同。也就是说，除去过程和函数的声明以外，我们在前面介绍的用于过程声明部分的语法也适用于包头的说明部分。下面是这类语法的规则：

- 包元素的位置可以任意安排，然而，在声明部分，对象必须在引用前进行声明。例如，如果一个游标使用了作为其WHERE子句一部分的变量，则该变量必须在声明游标之前声明。
- 包头可以不对任何类型的元素进行说明。例如，包可以只带有过程和函数说明语句，而不声明任何异常和类型。
- 对过程和函数的任何声明都必须是前向说明。所谓前向说明就是只对子程序和其参数（如果有的话）进行描述，但不带有任何代码的说明。本书的第 5章的‘前向声明’一节专门介绍该类声明的具体使用方法。该声明的规则不同于块声明语法，在块声明中，过程或函数的前向声明和代码同时出现在其声明部分，而实现包所说明的过程或函数的代码则只能出现在包体中。

4.2.2 包体

包体是一个独立于包头的数据字典对象。包体只能在包头完成编译后才能进行编译。包体中带有实现包头中描述的前向子程序的代码段。除此之外，包体还可以包括具有包体全局属性的附加声明部分，但这些附加说明对于说明部分是不可见的。下面的例子演示了包 ClassPackage的包体部分：

节选自在线代码ClassPackage.sql

```
CREATE OR REPLACE PACKAGE BODY ClassPackage AS
  -- Add a new student for the specified class.
  PROCEDURE AddStudent(p_StudentID  IN students.id%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course     IN classes.course%TYPE) IS
  BEGIN
    INSERT INTO registered_students (student_id, department, course)
      VALUES (p_StudentID, p_Department, p_Course);
  END AddStudent;
  -- Removes the specified student from the specified class.
  PROCEDURE RemoveStudent(p_StudentID IN students.id%TYPE,
```

```
                          p_Department IN classes.department%TYPE,
                          p_Course IN classes.course%TYPE) IS
  BEGIN
    DELETE FROM registered_students
      WHERE student_id = p_StudentID
      AND department = p_Department
      AND course = p_Course;

    -- Check to see if the DELETE operation was successful. If
    -- it didn't match any rows, raise an error.
    IF SQL%NOTFOUND THEN
      RAISE e_StudentNotRegistered;
    END IF;
END RemoveStudent;

  -- Returns a PL/SQL table containing the students currently
  -- in the specified class.
  PROCEDURE ClassList( p_Department  IN classes.department%TYPE,
                       p_Course      IN classes.course%TYPE,
                       p_IDs         OUT t_StudentIDTable,
                       p_NumStudents IN OUT BINARY_INTEGER) IS
v_StudentID registered_students.student_id%TYPE;

  -- Local cursor to fetch the registered students.
  CURSOR c_RegisteredStudents IS
    SELECT student_id
      FROM registered_students
      WHERE department = p_Department
      AND course = p_Course;
  BEGIN
   /* p_NumStudents will be the table index. It will start at
    * 0, and be incremented each time through the fetch loop.
    * At the end of the loop, it will have the number of rows
    * fetched, and therefore the number of rows returned in
    * p_IDs.
    */
   p_NumStudents := 0;

   OPEN c_RegisteredStudents;
   LOOP
     FETCH c_RegisteredStudents INTO v_StudentID;
     EXIT WHEN c_RegisteredStudents%NOTFOUND;
     p_NumStudents := p_NumStudents + 1;
     p_IDs(p_NumStudents) := v_StudentID;
   END LOOP;
  END ClassList;
END ClassPackage;
```

该包体部分包括了实现包头中过程的前向说明的代码。在包头中没有进行前向说明的对象

（如异常e_StudentNotRegistered）可以在包体中直接引用。

包体是可选的。如果包头中没有说明任何过程或函数的话（只有变量声明，游标，类型等），则该包体就不必存在。由于包中的所有对象在包外都是可见的，所以，这种说明方法可用来声明全局变量。（有关包元素的作用域和可见性将在下一节讨论。）

包头中的任何前向说明不能出现在包体中。包头和包体中的过程和函数的说明必须一致，其中包括子程序名和其参数名，以及参数的模式。例如，由于下面的包体对函数 FunctionA使用了不同的参数表，因此其包头与其包体不匹配。

节选自在线代码packageError.sql

```
CREATE OR REPLACE PACKAGE PackageA AS
  FUNCTION FunctionA(p_Parameter1 IN NUMBER,
                     p_Parameter2 IN DATE)
    RETURN VARCHAR2;
END PackageA;

CREATE OR REPLACE PACKAGE BODY PackageA AS
  FUNCTION FunctionA(p_Parameter1 IN CHAR)
    RETURN VARCHAR2;
END PackageA;
```

如果我们企图按上面的说明来创建包 PacgageA的话，编译程序将给包体提出下列错误警告：

```
PLS-00328: A subprogram body must be defined for the forward
           declaration of FUNCTIONA.
PLS-00323: subprogram or cursor 'FUNCTIONA' is declared in a
           package specification and must be defined in the package
           body.
```

4.2.3 包和作用域

包头中声明的任何对象都是在其作用域中，并且可在其外部使用包名作为前缀对其进行引用。例如，我们可以从下面的 PL/SQL块中调用对象ClassPackage.RemoveStudent:

```
BEGIN
  ClassPackage.RemoveStudent(10006, 'HIS', 101);
END;
```

上面的过程调用的格式与调用独立过程的格式完全一致，其唯一不同的地方是在被调用的过程名的前面使用了包名作为其前缀。打包的过程可以具有默认参数，并且这些参数可以通过按位置或按名称对应的方式进行调用，就象独立过程的参数的调用方式一样。

上述调用方法还可以适用于包中用户定义的类型。例如，为了调用过程 ClassList，我们需要声明一个类型为 ClassPackage.t_StudentIDTable的变量（请看本书的第 14章有关声明和使用 PL/SQL集合类型的介绍）：

节选自在线代码callCL.sql

```
DECLARE
  v_HistoryStudents ClassPackage.t_StudentIDTable;
  v_NumStudents     BINARY_INTEGER := 20;
```

```
BEGIN
  -- Fill the PL/SQL table with the first 20 History 101
  -- students.
  ClassPackage.ClassList('HIS', 101, v_HistoryStudents,
                         v_NumStudents);

  -- Insert these students into temp_table.
  FOR v_LoopCounter IN 1..v_NumStudents LOOP
    INSERT INTO temp_table (num_col, char_col)
      VALUES (v_HistoryStudents(v_LoopCounter),
              'In History 101');
  END LOOP;
END;
```

在包体内，包头中的对象可以直接引用，可以不用包名为其前缀。例如，过程 RemoveStuent **可以简单地使用** e_StudentNotRegistered**来引用异常，而不是用** ClassPackage. e_StudentNot Registered**来引用。当然，如果需要的话，也可以使用全名进行引用。**

包体中对象的作用域

按照目前的程序 ，过程 ClassPackage.AddStudent**和**ClassPackage.RemoveStudent**只是简单地对表**registered_student **进行更新。实际上，该操作还不完整。这两个过程还要更新表** students**和** classes**以反映新增或删除的学生情况。如下所示，我们可以在包体中增加一个过程来实现上述操作：**

节选自在线代码ClassPackage2.sql

```
CREATE OR REPLACE PACKAGE BODY ClassPackage AS
  -- Utility procedure that updates students and classes to reflect
  -- the change. If p_Add is TRUE, then the tables are updated for
  -- the addition of the student to the class. If it is FALSE,
  -- then they are updated for the removal of the student.
  PROCEDURE UpdateStudentsAndClasses(
     p_Add       IN BOOLEAN,
     p_StudentID IN students.id%TYPE,
     p_Department IN classes.department%TYPE,
     p_Course     IN classes.course%TYPE) IS

   -- Number of credits for the requested class
   v_NumCredits classes.num_credits%TYPE;
BEGIN
    -- First determine NumCredits.
    SELECT num_credits
      INTO v_NumCredits
      FROM classes
      WHERE department = p_Department
      AND course = p_Course;

    IF (p_Add) THEN
     -- Add NumCredits to the student's course load
```

```
      UPDATE STUDENTS
        SET current_credits = current_credits + v_NumCredits
        WHERE ID = p_StudentID;

      -- And increase current_students
      UPDATE classes
        SET current_students = current_students + 1
        WHERE department = p_Department
        AND course = p_Course;
      ELSE
        -- Remove NumCredits from the students course load
        UPDATE STUDENTS
          SET current_credits = current_credits - v_NumCredits
          WHERE ID = p_StudentID;

      -- And decrease current_students
      UPDATE classes
          SET current_students = current_students - 1
          WHERE department = p_Department
          AND course = p_Course;
      END IF;
    END UpdateStudentsAndClasses;

    -- Add a new student for the specified class.
    PROCEDURE AddStudent(p_StudentID IN students.id%TYPE,
                         p_Department IN classes.department%TYPE,
                         p_Course     IN classes.course%TYPE) IS
    BEGIN
      INSERT INTO registered_students (student_id, department, course)
        VALUES (p_StudentID, p_Department, p_Course);

  UpdateStudentsAndClasses(TRUE, p_StudentID, p_Department,
                           p_Course);
    END AddStudent;

    -- Removes the specified student from the specified class.
    PROCEDURE RemoveStudent(p_StudentID IN students.id%TYPE,
                            p_Department IN classes.department%TYPE,
                            p_Course IN classes.course%TYPE) IS
  BEGIN
    DELETE FROM registered_students
      WHERE student_id = p_StudentID
      AND department = p_Department
      AND course = p_Course;

    -- Check to see if the DELETE operation was successful. If
    -- it didn't match any rows, raise an error.
    IF SQL%NOTFOUND THEN
      RAISE e_StudentNotRegistered;
    END IF;
```

```
    UpdateStudentsAndClasses(FALSE, p_StudentID, p_Department,
                             p_Course);

  END RemoveStudent;

  ...
END ClassPackage;
```

过程UpdateStudentAndclasses声明为包体的全局量，其作用域是包体本身。该过程可以由该包中的其他过程调用（如AddStudent和RemoveStudent）,但是该过程在包体外是不可见的。

4.2.4 重载打包子程序

在包的内部，过程和函数可以被重载（Overloading）。也就是说，可以有一个以上的名称相同，但参数不同的过程或函数。由于重载允许将相同的操作施加在不同类型的对象上，因此，它是PL/SQL语言的一个重要特点。例如，假设我们要使用学生ID或该学生的姓和名来把一个学生加入到班级中。我们可以对包ClassPackage修改如下：

节选自在线代码overload.sql

```
CREATE OR REPLACE PACKAGE ClassPackage AS
  -- Add a new student into the specified class.
  PROCEDURE AddStudent(p_StudentID IN students.id%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course IN classes.course%TYPE);
  -- Also adds a new student, by specifying the first and last
  -- names, rather than ID number.
  PROCEDURE AddStudent(p_FirstName IN students.first_name%TYPE,
                       p_LastName  IN students.last_name%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course     IN classes.course%TYPE);
  ...
END ClassPackage;

CREATE OR REPLACE PACKAGE BODY ClassPackage AS
  -- Add a new student for the specified class.
  PROCEDURE AddStudent(p_StudentID IN students.id%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course IN classes.course%TYPE) IS
  BEGIN
    INSERT INTO registered_students (student_id, department, course)
      VALUES (p_StudentID, p_Department, p_Course);
END AddStudent;

  -- Add a new student by name, rather than ID.
  PROCEDURE AddStudent(p_FirstName IN students.first_name%TYPE,
                       p_LastName  IN students.last_name%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course     IN classes.course%TYPE) IS
```

```
  v_StudentID students.ID%TYPE;
BEGIN
  /* First we need to get the ID from the students table. */
  SELECT ID
    INTO v_StudentID
    FROM students
    WHERE first_name = p_FirstName
   AND last_name = p_LastName;

  -- Now we can add the student by ID.
  INSERT INTO registered_students (student_id, department, course)
    VALUES (v_StudentID, p_Department, p_Course);
  END AddStudent;
 ...
END ClassPackage;
```

现在，我们可以对 Music410**增加一名学生如下：**

```
BEGIN
  ClassPackage.AddStudent(10000, 'MUS', 410);
END;
```

或

```
BEGIN
  ClassPackage.AddStudent('Rita', 'Razmataz', 'MUS', 410);
END;
```

我们可以看到同样的操作可以通过不同类型的参数实现，这就说明重载是非常有用的技术。虽然如此，但重载仍要受到下列限制：

- **如果两个子程序的参数仅在名称和模式上不同的话，这两个过程不能重载。例如下面的两个过程是不能重载的：**

```
PROCEDURE overloadMe(p_TheParameter IN NUMBER);
PROCEDURE overloadMe(p_TheParameter OUT NUMBER);
```

- **不能仅根据两个过程不同的返回类型对其进行重载。例如，下面的函数是不能进行重载的：**

```
FUNCTION overloadMeToo RETURN DATE;
FUNCTION overloadMeToo RETURN NUMBER;
```

- **最后，重载函数的参数的类族（**type family**）必须不同，也就是说，不能对同类族的过程进行重载。例如，由于**CHAR**和**VARCHAR2**属于同一类族，所以不能重载下面的过程：**

```
PROCEDURE OverloadChar(p_TheParameter IN CHAR);
PROCEDURE OverloadChar(p_TheParameter IN VARCHAR2);
```

注意 PL/SQL**编译器实际上允许程序员创建违反上述限制的带有子程序的包。然而，**PL/SQL**运行时系统将无法解决引用问题并将引发“** PLS-307:too many declaration of ‘subprogram’ match this call **”的运行错误。**

对象类型和重载

Oracle 8i 及更高版本

根据用户定义的对象类型，打包子程序也可以重载。例如，假设我们要创建下面的两个对象类型：

节选自在线代码objectOverload.sql

```
CREATE OR REPLACE TYPE t1 AS OBJECT (
  f NUMBER
);

CREATE OR REPLACE TYPE t2 AS OBJECT (
  f NUMBER
);
```

现在，我们可以创建一个包和一个带有根据其参数的对象类型重载的两个过程的包体：

节选自在线代码objectOverload.sql

```
CREATE OR REPLACE PACKAGE Overload AS
  PROCEDURE Proc(p_Parameter1 IN t1);
  PROCEDURE Proc(p_Parameter1 IN t2);
END Overload;

CREATE OR REPLACE PACKAGE BODY Overload AS
  PROCEDURE Proc(p_Parameter1 IN t1) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Proc(t1): ' || p_Parameter1.f);
  END Proc;

  PROCEDURE Proc(p_Parameter1 IN t2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Proc(t2): ' || p_Parameter1.f);
  END Proc;
END Overload;
```

如下例所示，根据参数的类型对过程进行正确的调用：

节选自在线代码objectOverload.sql

```
SQL> DECLARE
  2   v_Obj1 t1 := t1(1);
  3   v_OBj2 t2 := t2(2);
  4 BEGIN
  5   Overload.Proc(v_Obj1);
  6   Overload.proc(v_Obj2);
  7 END;
  8 /
Proc(t1): 1
Proc(t2): 2
PL/SQL procedure successfully completed.
```

有关对象类型的详细内容及使用方法，请看本书的第 12,13 两章。

4.2.5 包的初始化

当第一次调用打包子程序时，该包将进行初始化。也就是说将该包从硬盘中读入到内存并

启动调用的子程序的编译代码开始运行。这时，系统为该包中定义的所有变量分配内存单元。每个会话都有其打包变量的副本，以确保执行同一包子程序的两个对话使用不同的内存单元。

在大多数情况下，初始化代码要在包第一次初始化时运行。为了实现这种功能，我们可以在包体中所有对象之后加入一个初始化部分，其语法如下：

```
CREATE OR REPLACE PACKAGE BODY package_name{IS | AS}
 ...
BEGIN
 initialization_code;
END [ package_name];
```

其中，package_name是包的名称，initialization_code是要运行的初始化代码。例如，下面的包实现了一个随机数函数：

节选自在线代码Random.sql

```
CREATE OR REPLACE PACKAGE Random AS
  -- Random number generator. Uses the same algorithm as the
  -- rand() function in C.

  -- Used to change the seed. From a given seed, the same
  -- sequence of random numbers will be generated.
  PROCEDURE ChangeSeed(p_NewSeed IN NUMBER);

  -- Returns a random integer between 1 and 32767.
  FUNCTION Rand RETURN NUMBER;

  -- Same as Rand, but with a procedural interface.
  PROCEDURE GetRand(p_RandomNumber OUT NUMBER);

  -- Returns a random integer between 1 and p_MaxVal.
  FUNCTION RandMax(p_MaxVal IN NUMBER) RETURN NUMBER;

  -- Same as RandMax, but with a procedural interface.
  PROCEDURE GetRandMax(p_RandomNumber OUT NUMBER,
                       p_MaxVal IN NUMBER);
END Random;

CREATE OR REPLACE PACKAGE BODY Random AS

  /* Used for calculating the next number. */
  v_Multiplier CONSTANT NUMBER := 22695477;
  v_Increment  CONSTANT NUMBER := 1;

  /* Seed used to generate random sequence. */
  v_Seed number := 1;

  PROCEDURE ChangeSeed(p_NewSeed IN NUMBER) IS
  BEGIN
    v_Seed := p_NewSeed;
  END ChangeSeed;

  FUNCTION Rand RETURN NUMBER IS
```

```
  BEGIN
    v_Seed := MOD(v_Multiplier * v_Seed + v_Increment,
                  (2 ** 32));
  RETURN BITAND(v_Seed/(2 ** 16), 32767);
END Rand;

PROCEDURE GetRand(p_RandomNumber OUT NUMBER) IS
BEGIN
  -- Simply call RandMax and return the value.
  p_RandomNumber := Rand;
END GetRand;

FUNCTION RandMax(p_MaxVal IN NUMBER) RETURN NUMBER IS
BEGIN
  RETURN MOD(Rand, p_MaxVal) + 1;
END RandMax;

PROCEDURE GetRandMax(p_RandomNumber OUT NUMBER,
                     p_MaxVal IN NUMBER) IS
BEGIN
  -- Simply call RandMax and return the value.
  p_RandomNumber := RandMax(p_MaxVal);
  END GetRandMax;

BEGIN
  /* Package initialization. Initialize the seed to the current
     time in seconds. */
  ChangeSeed(TO_NUMBER(TO_CHAR(SYSDATE, 'SSSSS')));
END Random;
```

为了检索随机数，我们可以直接调用函数Random.Rand。随机数序列是由其初始种子控制的，对于给定的种子可以生成相应的随机数序列。因此，为了提供更多的随机数值，我们要在每次实例化该包时，把随机数种子初始化为不同的值。为了实现上述功能，我们从包的初始部分调用过程ChangeSeed。

Oracle 8及更高版本

Oracle8中提供了内置包 DBMS_RANDOM,该包可以用于提供随机数。请看本书CD-ROM中附录A对内置包的介绍。

4.3 小结

我们在本章分析了三种类型的命名 PL/SQL块：过程，函数，和包。我们还讨论了创建这些块的语法，特别是着重分析了各种参数的类型及传递方式。在下一章，我们将更多地使用过程，函数和包。第5章的重点是子程序的类型，它们在数据字典中的存储方式，以及从 SQL语句调用存储子程序的方法。第5章的最后将介绍Oracle8i新增的功能。在本书的第6章，我们将介绍命名块的第四种类型，数据库触发器类型。