

## 第5章 使用子程序和包

在上一章中，我们讨论了创建过程，函数和包的细节。在本章中，我们介绍这些部件的功能，存储子程序和本地子程序的区别，存储子程序与数据字典的交互方式及如何从 SQL 语句中调用存储子程序。除此之外，我们还要介绍 Oracle8i 存储子程序的新增特性。

### 5.1 子程序位置

我们已在前几章中演示了可以存储在数据字典中的子程序和包。子程序首次是用命令 CREATE OR REPLACE 创建的，接着，我们可以从其他 PL/SQL 块中调用已创建的子程序。除此之外，子程序可以在块的声明部分定义，以这种方式定义的子程序叫做本地子程序。包则必须存储在数据字典中，而不能在本地定义存储。

#### 5.1.1 存储子程序和数据字典

当使用命令 CREATE OR REPLACE 创建子程序时，该子程序就存储在数据字典中。除去子程序中的源文本外，该子程序是以编译后的中间代码形式存储的，这种中间代码叫做 p-code。中间代码中带有子程序中经计算得到的所有引用参数，子程序的源代码也被转换为 PL/SQL 引擎易读的格式。当调用子程序时，就将中间代码从磁盘读入并启动执行。一旦从磁盘读入中间代码，系统就将其存储在系统全局工作区（SGA）的共享缓冲区部分，以便由多个用户同时进行访问。与缓冲区的所有内容一样，根据系统采用的最近最少使用的算法，过期的中间代码将被从共享缓冲区中清除。

中间代码类似于由 3GL 语言生成的对象代码，或者类似于可由 Java 运行时使用的 Java 字节码。由于中间代码带有经计算得到的子程序中的所有对象引用（属于前联编的属性），所以，执行中间代码的效率非常高。

子程序的信息可以通过各种数据字典视图来访问。视图 user\_objects 包括了当前用户拥有的所有对象的信息。该信息包括了对象的创建以及最后修改的时间，对象类型（表，序列，函数等）和对象的有效性。视图 user\_source 包括了对象的源程序代码。而视图 user\_errors 则包括了编译错误信息。

请看下面的简单过程：

```
CREATE OR REPLACE PROCEDURE Simple AS
    v_Counter NUMBER;
BEGIN
    v_Counter := 7;
END Simple;
```

创建该过程后，视图 user\_objects 显示该过程是合法的，视图 user\_source 则包括了该过程的源代码。由于该过程已经编译成功，所以视图 user\_errors 没有显示错误。图 5-1 的窗口显示了上

述信息。

如果我们修改了过程 Simple 的代码，就会出现编译错误（源程序中缺少一个分号），修改过的该过程如下：

```
CREATE OR REPLACE PROCEDURE Simple AS
    v_counter NUMBER;
BEGIN
    v_counter := 7
END Simple;
```

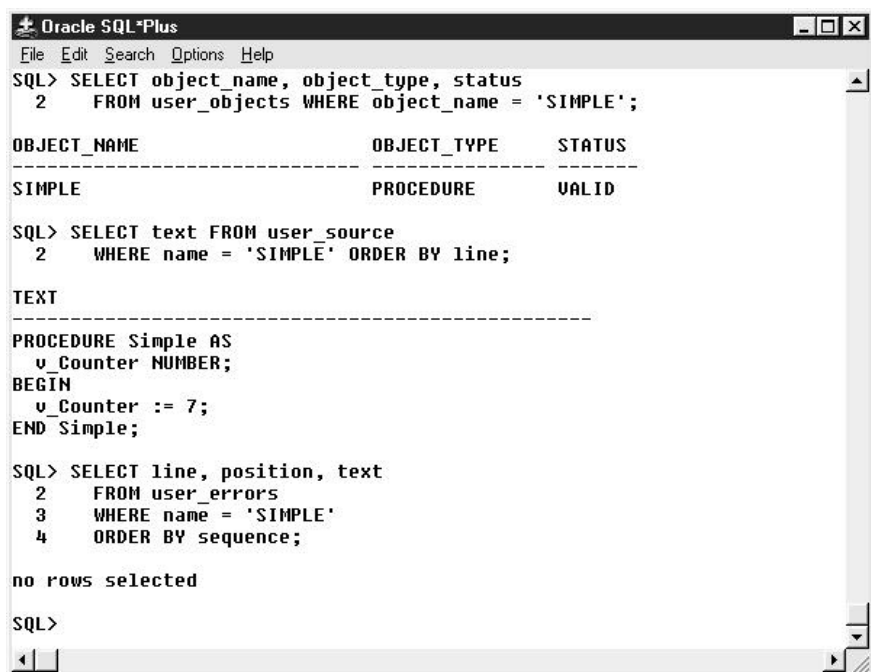


图5-1 成功编译后的数据字典视图

分析图5-2所示的相同的三个数据字典，我们不难看出有几个不同之处。首先，user\_source仍然显示了该过程的源代码。然而，user\_objects中的状态指示为非法，而不是前面例子的合法提示，user\_errors中有一条编译错误信息 PLS-103。

**提示** 在SQL \*Plus中，命令 SHOW ERRORS 将为用户查询 user\_errors 并将输出数据格式为用户可读的形式。该命令将返回最后创建的对象错误信息。如果编译程序出现了错误提示：‘Warning: Procedure created with compilation errors’时，就可以使用该命令查询错误的详细信息。有关 SQL \*Plus 的详细介绍，请看本书第2章中介绍 PL/SQL 开发工具的内容。

虽然非法的存储子程序仍然在数据字典中，但是该子程序只有在将编译错误修改后才能调用。如果调用了非法的过程，就会引发 PLS-905 错误，下面的例子演示了这种非法调用产生的错误：

```

Oracle SQL*Plus
File Edit Search Options Help
SQL> SELECT object_name, object_type, status
2   FROM user_objects WHERE object_name = 'SIMPLE';

OBJECT_NAME          OBJECT_TYPE          STATUS
-----
SIMPLE                PROCEDURE            INVALID

SQL> SELECT text FROM user_source
2   WHERE name = 'SIMPLE' ORDER BY line;

TEXT
-----
PROCEDURE Simple AS
  v_Counter NUMBER;
BEGIN
  v_Counter := 7
END Simple;

SQL> SELECT line, position, text
2   FROM user_errors
3   WHERE name = 'SIMPLE'
4   ORDER BY sequence;

LINE  POSITION TEXT
-----
5          1 PLS-00103: Encountered the symbol "END" when expecting one of
the following:

          * & = - + ; < / > in mod not rem an exponent (**)
          <> or != or ~= >= <= <> and or like between is null is not
          ||
          is dangling
          The symbol ";" was substituted for "END" to continue.
  
```

图5-2 编译出错后的数据字典

```

SQL> BEGIN Simple; END;
2 /
BEGIN Simple; END;
*
ERROR at line 1:
ORA-06550: line 1, column 7:
PLS-00905: object EXAMPLE.SIMPLE is invalid
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored
  
```

有关数据字典的内容将在本书的配套光盘中的附录 C 给予详细的介绍。

### 5.1.2 本地子程序

下面的程序是一个在 PL/SQL 块的声明部分声明的本地子程序的例子：

节选自在线代码 localSub.sql

```

DECLARE
  CURSOR c_AllStudents IS
    SELECT first_name, last_name
    FROM students;

  v_FormattedName VARCHAR2(50);
  
```

```

/* Function that will return the first and last name
concatenated together, separated by a space. */
FUNCTION FormatName(p_FirstName IN VARCHAR2,
                   p_LastName IN VARCHAR2)
RETURN VARCHAR2 IS
BEGIN
RETURN p_FirstName || ' ' || p_LastName;
END FormatName;

-- Begin main block.
BEGIN
FOR v_StudentRecord IN c_AllStudents LOOP
v_FormattedName :=
FormatName(v_StudentRecord.first_name,
           v_StudentRecord.last_name);
DBMS_OUTPUT.PUT_LINE(v_FormattedName);
END LOOP;
END;

```

函数ForamtName是在块的声明部分声明的。该函数名是一个 PL/SQL 的标识符，因此该函数名将遵循 PL/SQL 语言中标识符的作用域和可见性规则，该函数只在其声明的块中可见，其作用域从声明点开始到该块结束为止。其他块不能调用该函数，因为该函数对其他块来说是不可见的。

#### 1. 本地子程序作为存储子程序的一部分

请看下面的程序例子，本地子程序也可以声明为存储子程序声明部分的内容。在这种情况下，由于该函数的作用域的限制，也只能从过程 StoredProc 中调用函数 FormatName。

节选自在线代码 localStored.sql

```

CREATE OR REPLACE PROCEDURE StoredProc AS
/* Local declarations, which include a cursor, variable, and a
function. */
CURSOR c_AllStudents IS
SELECT first_name, last_name
FROM students;

v_FormattedName VARCHAR2(50);

/* Function that will return the first and last name
concatenated together, separated by a space. */
FUNCTION FormatName(p_FirstName IN VARCHAR2,
                   p_LastName IN VARCHAR2)
RETURN VARCHAR2 IS
BEGIN
RETURN p_FirstName || ' ' || p_LastName;
END FormatName;

-- Begin main block.
BEGIN
FOR v_StudentRecord IN c_AllStudents LOOP

```

```

v_FormattedName :=
    FormatName(v_StudentRecord.first_name,
              v_StudentRecord.last_name);
DBMS_OUTPUT.PUT_LINE(v_FormattedName);
END LOOP;
END StoredProc;

```

## 2. 本地子程序的位置

任何本地子程序都必须在声明部分的结尾处声明。如果我们把函数 `FormatName` 的声明移动到 `c_AllStudent` 声明的上面的话，如下面的 SQL \*Plus 声明部分所示，我们将得到编译错误。

节选自在线代码 `localError.sql`

```

SQL> DECLARE
2    /* Declare FormatName first. This will generate a compile
3       error, since all other declarations have to be before
4       any local subprograms. */
5    FUNCTION FormatName(p_FirstName IN VARCHAR2,
6                       p_LastName IN VARCHAR2)
7        RETURN VARCHAR2 IS
8    BEGIN
9        RETURN p_FirstName || ' ' || p_LastName;
10   END FormatName;
11
12   CURSOR c_AllStudents IS
13       SELECT first_name, last_name
14          FROM students;
15
16   v_FormattedName VARCHAR2(50);
17 -- Begin main block
18 BEGIN
19     NULL;
20 END;
21 /
22 CURSOR c_AllStudents IS
23 *
24
ERROR at line 11:
ORA-06550: line 11, column 3:
PLS-00103: Encountered the symbol "CURSOR" when expecting one of the
following:
begin function package pragma procedure form

```

## 3. 前向声明 (Forward Declarations)

由于本地 PL/SQL 子程序的名称是标识符，所以它们必须在引用前声明。一般来说，满足这种要求不是一件困难的事情。然而，在具有相互引用的子程序中，实现上述要求就有一定的难度。请看下面的例子：

节选自在线代码 `mutual.sql`

```

DECLARE
    v_TempVal BINARY_INTEGER := 5;
    -- Local procedure A. Note that the code of A calls procedure B.
    PROCEDURE A(p_Counter IN OUT BINARY_INTEGER) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('A(' || p_Counter || ')');
        IF p_Counter > 0 THEN
            B(p_Counter);
            p_Counter := p_Counter - 1;
        END IF;
    END A;

    -- Local procedure B. Note that the code of B calls procedure A.
    PROCEDURE B(p_Counter IN OUT BINARY_INTEGER) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('B(' || p_Counter || ')');
        p_Counter := p_Counter - 1;
        A(p_Counter);
    END B;
BEGIN
    B(v_TempVal);
END;
```

该例子无法进行编译，其原因是过程 A 调用了过程 B，因此过程 B 必须要在过程 A 之前声明以便可以确定对过程 B 的引用。同时，由于过程 B 要调用过程 A，要求过程 A 也要在过程 B 之前声明以便对确定对过程 B 的引用。在这种情况下，上述要求不能同时满足。为了协调这种需求，我们可以使用前向声明来解决该问题。前向声明只需要提供过程名和其形参就可以实现相互引用的过程的并存。除此之外，前向声明还可以用在包头中。下面就是一个使用前向声明的例子：

节选自在线代码 forwardDeclaration.sql

```

DECLARE
    v_TempVal BINARY_INTEGER := 5;

    -- Forward declaration of procedure B.
    PROCEDURE B(p_Counter IN OUT BINARY_INTEGER);

PROCEDURE A(p_Counter IN OUT BINARY_INTEGER) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('A(' || p_Counter || ')');
    IF p_Counter > 0 THEN
        B(p_Counter);
        p_Counter := p_Counter - 1;
    END IF;
END A;

PROCEDURE B(p_Counter IN OUT BINARY_INTEGER) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('B(' || p_Counter || ')');
```

```
p_Counter := p_Counter - 1;
A(p_Counter);
END B;
BEGIN
  B(v_TempVal);
END;
```

该块的输出如下：

```
B(5)
A(4)
B(4)
A(3)
B(3)
A(2)
B(2)
A(1)
B(1)
A(0)
```

#### 4. 重载本地子程序

我们在第4章中曾经介绍过，包中声明的子程序可以被重载。该规则对于本地子程序的情况也适用，下面就是一个对本地子程序进行重载的例子：

节选自在线代码overloadedlocal.sql

```
DECLARE
  -- Two overloaded local procedures
  PROCEDURE LocalProc(p_Parameter1 IN NUMBER) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('In version 1, p_Parameter1 = ' ||
                          p_Parameter1);

  END LocalProc;

  PROCEDURE LocalProc(p_Parameter1 IN VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('In version 2, p_Parameter1 = ' ||
                          p_Parameter1);

  END LocalProc;
BEGIN
  -- Call version 1
  LocalProc(12345);

  -- And version 2
  LocalProc('abcdef');
END;
```

该块的输出如下：

```
In version 1, p_Parameter1 = 12345
In version 2, p_Parameter1 = abcdef
```

### 5.1.3 存储子程序和本地子程序的比较

存储子程序和本地子程序的工作方式不同，它们具有不同的属性。它们在什么情况下使用呢？作者个人倾向于使用存储子程序，通常把存储子程序放在包里使用。如果我们开发了一个有用的子程序，通常，我们希望能够从一个以上的块中对其进行调用。为了实现这种功能，该子程序必须放在数据库中使用。除此之外，存储子程序的长度和复杂性与本地子程序相比也具有一定的优势。作者声明为本地子程序的过程和函数一般都是代码很少的程序段，并且也只是从程序（包含该程序的块）特定的部分进行调用。使用这类的本地子程序的主要考虑是为了避免单块中的代码重复问题。这种使用方法类似于 C 语言中的宏功能。表 5-1 总结了存储子程序和本地子程序的区别。

表5-1 存储子程序与本地子程序的比较

存储子程序	本地子程序
该类子程序以编译后生成的中间代码形式p-code存储在数据库中。当调用该类子程序时，不需进行编译即可运行	本地子程序被编译为该程序所在块的一部分。如果其所在块是匿名块并需要多次运行时，则该子程序就必须每次进行编译
存储子程序可以从由用户提交的具有子程序优先级 EXECUTE 属性的任何块中调用	本地子程序只能从包含子程序的块内调用
由于存储子程序与调用块的相互隔离，调用块具有代码少，易于理解的特点。除此之外，子程序和调用块还可以各自独立维护	本地子程序和调用块同处于一个块内，所以容易引起混淆。如果修改了调用块的话，则该块调用的子程序作为所属块的一部分也要重新编译
可以使用 DBMS_SHARED_POOL.KEEP 打包过程来把编译后 p-code 代码存储在共享缓冲区中*。这种方式可以改善程序性能	本地子程序自身不能存储在共享缓冲区中
不能对独立存储子程序进行重载，但同一包内的打包子程序可以重载	同一块中的本地子程序可以重载

\* 包DBMS\_SHARED\_POOL将在5.4.1节中介绍。

## 5.2 存储子程序和包的几个问题

作为数据字典对象的存储子程序和包具有自身独特的优势，例如，这类程序可以由多个数据库用户共享等。然而，在使用中，我们必须还要注意到有关存储子程序和包的几种特性。其中包括存储子程序间的相关性，包状态的处理方法，以及运行存储子程序和包所需的特权等。

### 5.2.1 子程序的相关性

当我们编译存储过程或函数时，该过程或函数引用的所有 Oracle 对象都将记录在数据字典中。



该过程就依赖于这些存储的对象。在前面几节中，我们已经看到了在数据字典中显示了标志为非法的有编译错误的子程序。同样，如果一个 DDL 操作运行在其所相关的对象上时，存储子程序也将是非法的。理解该问题的最好方式是通过例子进行说明。我们在第 4 章中定义的函数 AlmostFull 要查询表 classes。图 5-3 演示了函数 AlmostFull 的相关性。AlmostFull 只与一个对象相关，即表 classes。图 5-3 中的箭头标识了 AlmostFull 的相关对象。

现在，让我们假设创建了调用 AlmostFull 的过程并把结果插入到表 temp\_table 中。过程 RecordFullClasses 的代码如下：

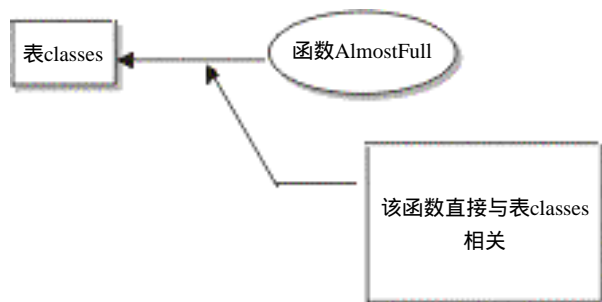


图 5-3 函数 AlmostFull 的相关图示

节选自在线代码 RecordFullClasses.sql

```
CREATE OR REPLACE PROCEDURE RecordFullClasses AS
  CURSOR c_Classes IS
    SELECT department, course
      FROM classes;
BEGIN
  FOR v_ClassRecord IN c_Classes LOOP
    -- Record all classes that don't have very much room left
    -- in temp_table.
    IF AlmostFull(v_ClassRecord.department, v_ClassRecord.course)
    THEN
      INSERT INTO temp_table (char_col) VALUES
        (v_ClassRecord.department || ' ' || v_ClassRecord.course ||
         ' is almost full!');
    END IF;
  END LOOP;
END RecordFullClasses;
```

过程 RecordFullClasses 的相关性如图 5-4 所示。过程 RecordFullClasses 与函数 AlmostFull 和表 temp\_table 相关。由于过程 RecordFullClasses 直接引用函数 AlmostFull 和表 temp\_table，这种相关就是直接相关。函数 AlmostFull 又与表 classes 相关，因此过程 RecordFullClasses 就与表 classes 间接相关。

如果一个 DLL 执行了对表 classes 的操作，则所有与表 classes 相关的对象（直接或间接）都将

处于无效状态。如下所示，假设我们在例子中的表 classes 中增加一列使其发生变更：

```
ALTER TABLE classes ADD (  
    student_rating NUMBER(2) -- Difficulty rating from 1 to 10  
);
```

该表的变更将导致与表 classes 相关的函数 AlmostFull 和过程 RecordFullClasses 变为非法。图 5-5 所示的 SQL \*Plus 会话中窗口显示了非法错误信息。

### 1. 自动重编译

如果相关的对象处于非法状态的话，PL/SQL 引擎将在该对象再次被调用时对其重新进行编译。由于函数 AlmostFull 和过程 RecordFullClasses 都没有引用表 classes 中新增的列，所以重编译可以顺利通过。继续图 5-5 会话的 SQL \*Plus 窗口图 5-6 演示了重编译的结果。

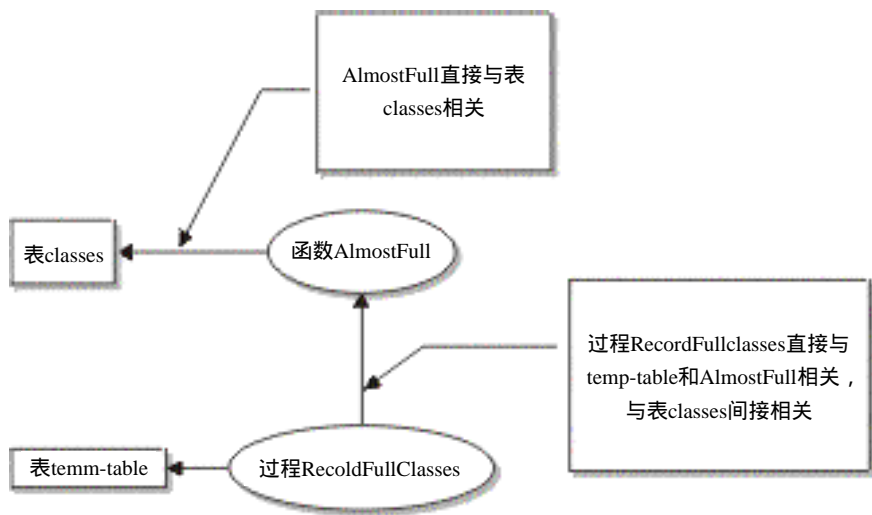


图5-4 RecordFullClasses 的相关图示

**警告** 自动重编也可能失败（特别是在修改了表说明的情况下）。这时，调用块将收到一条编译错误信息。但该错误是在运行时生成，而不在编译后给出。

### 2. 包和相关性

如同上面例子所演示的，存储子程序在其相关的对象变更时将会处于非法状态。然而，包的情况有所不同。请看图 5-7 所示的包 ClassPackage 的相关图。该包体与表 registered\_students 和包头有关。但是，该包头与包体无关，仅与表 registered\_students 有关。这就是包的一个优点，既包体的变化不会导致修改包头。因此，与该包头有关的其他对象也不需要重编。如果该包头有变化，则将自动地作废其包体，这是因为该包体与其包头有关。

**注意** 确实存在着在某种情况下包体的改变将导致包头做相应的修改。例如，如果某个过程在其包体和说明部分的参数在其包体中发生变化，则该包头也必须做相应的修改以

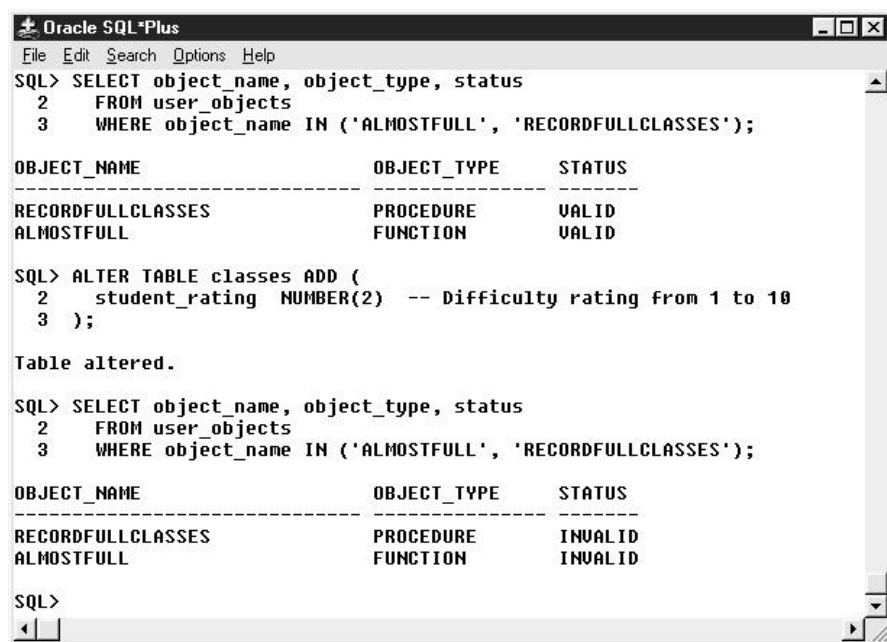


图5-5 DDL操作导致的非法信息

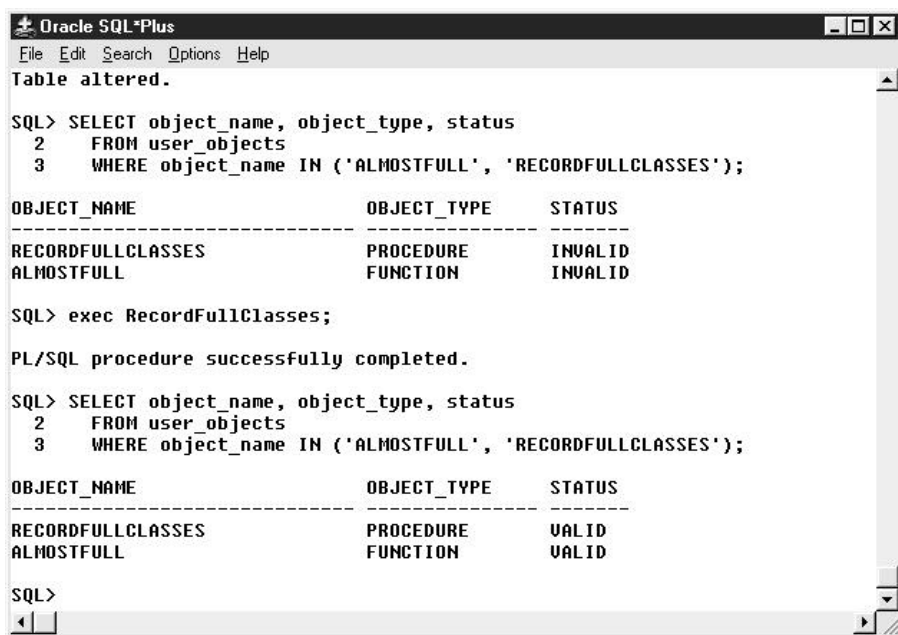


图5-6 出现非法错误后的自动编译

适应这种变化。如果只是对实现过程的包体做了修改而没有影响其声明部分，则该包头就可以不做修改。相类似，如果使用了特征相关模式（本章下文小节介绍），则仅对包说明部分的对象特征修的改变将会导致包体非法。同样，如果在包头中增加了一个对象

(如游标或变量), 则包体将被作废。

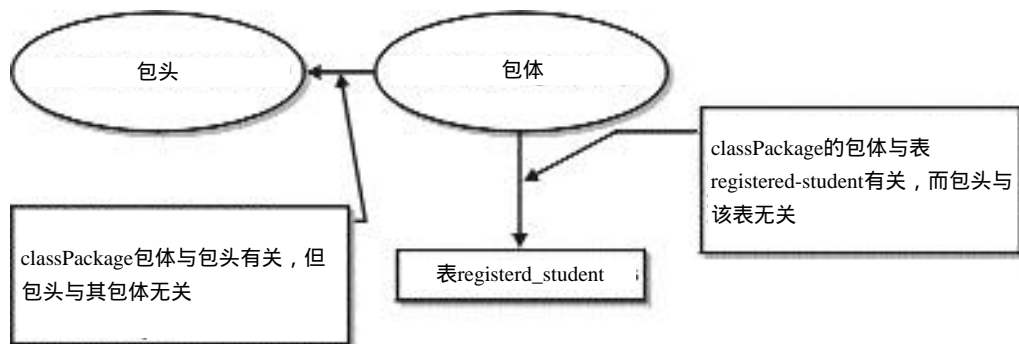


图5-7 包ClassesPackage的相关图

从下面的SQL \*Plus会话中我们也可以看出上述情况：

节选自在线代码dependencies.sql

```
SQL> -- First create a simple table.
```

```
SQL> CREATE TABLE simple_table (f1 NUMBER);
```

```
Table created.
```

```
SQL> -- Now create a packaged procedure that references the table.
```

```
SQL> CREATE OR REPLACE PACKAGE Dependee AS
```

```
2   PROCEDURE Example(p_Val IN NUMBER);
```

```
3 END Dependee;
```

```
4 /
```

```
Package created.
```

```
SQL> CREATE OR REPLACE PACKAGE BODY Dependee AS
```

```
2   PROCEDURE Example(p_Val IN NUMBER) IS
```

```
3   BEGIN
```

```
4       INSERT INTO simple_table VALUES (p_Val);
```

```
5   END Example;
```

```
6 END Dependee;
```

```
7 /
```

```
Package body created.
```

```
SQL> -- Now create a procedure that references Dependee.
```

```
SQL> CREATE OR REPLACE PROCEDURE Depender(p_Val IN NUMBER) AS
```

```
2 BEGIN
```

```
3   Dependee.Example(p_Val + 1);
```

```
4 END Depender;
```

```
5 /
```

```
Procedure created.
```

```
SQL> -- Query user_objects to see that all objects are valid.
```

```
SQL> SELECT object_name, object_type, status
```

```

2 FROM user_objects
3 WHERE object_name IN ('DEPENDER', 'DEPENDEE',
4                       'SIMPLE_TABLE');
                                OBJECT_NAME OBJECT_TYPE STATUS

```

```

-----
SIMPLE_TABLE          TABLE          VALID
DEPENDEE              PACKAGE          VALID
DEPENDEE              PACKAGE BODY   VALID
DEPENDER              PROCEDURE      VALID

```

SQL> -- Change the package body only. Note that the header is  
SQL> -- unchanged.

```

SQL> CREATE OR REPLACE PACKAGE BODY Dependee AS
2   PROCEDURE Example(p_Val IN NUMBER) IS
3   BEGIN
4       INSERT INTO simple_table VALUES (p_Val - 1);
5   END Example;
6 END Dependee;
7 /

```

Package body created.

SQL> -- Now user\_objects shows that Depender is still valid.

```

SQL> SELECT object_name, object_type, status
2 FROM user_objects
3 WHERE object_name IN ('DEPENDER', 'DEPENDEE',
4                       'SIMPLE_TABLE');

```

```

OBJECT_NAME          OBJECT_TYPE STATUS
-----
SIMPLE_TABLE          TABLE          VALID
DEPENDEE              PACKAGE          VALID
DEPENDEE              PACKAGE BODY   VALID
DEPENDER              PROCEDURE      VALID

```

SQL> -- Even if we drop the table, it only invalidates the  
SQL> -- package body.

```

SQL> DROP TABLE simple_table;
Table dropped.

```

SQL> SELECT object\_name, object\_type, status

```

2 FROM user_objects
3 WHERE object_name IN ('DEPENDER', 'DEPENDEE',
4 'SIMPLE_TABLE');

```

```

OBJECT_NAME OBJECT_TYPE STATUS
-----
DEPENDEE     PACKAGE          VALID
DEPENDEE     PACKAGE BODY   INVALID
DEPENDER     PROCEDURE      VALID

```

注意 数据字典视图user\_dependencies,all\_dependencies,和dba\_dependencies直接列出了模式对象间的关系。有关这些视图的介绍,请看本书 CD-ROM中的附录C部分。

图5-8演示了由脚本创建的对象的相关图。

### 3. 如何确认非法状态

当对象变更时,其相关的对象就会变成非法对象,我们在上面的例子中已经看到了这一点。如果所有的对象都在同一个数据库中的话,则相关的对象将会在底层对象变更的同时进入非法状态。由于数据字典在不断地跟踪对象间的相关,所以这种变化可以快速反应出来。假设我们创建了过程P1和P2,如图5-9所示,P1依赖于P2,也就是说,对P2进行重编译将会导致P1非法。下面的SQL \*Plus会话演示了这一过程:

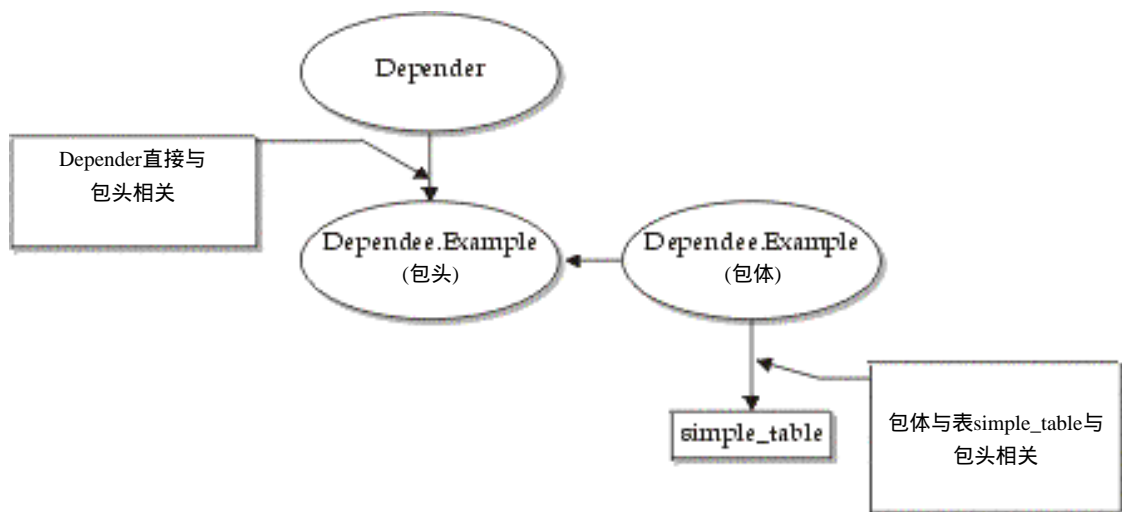


图5-8 多个包的相关图示

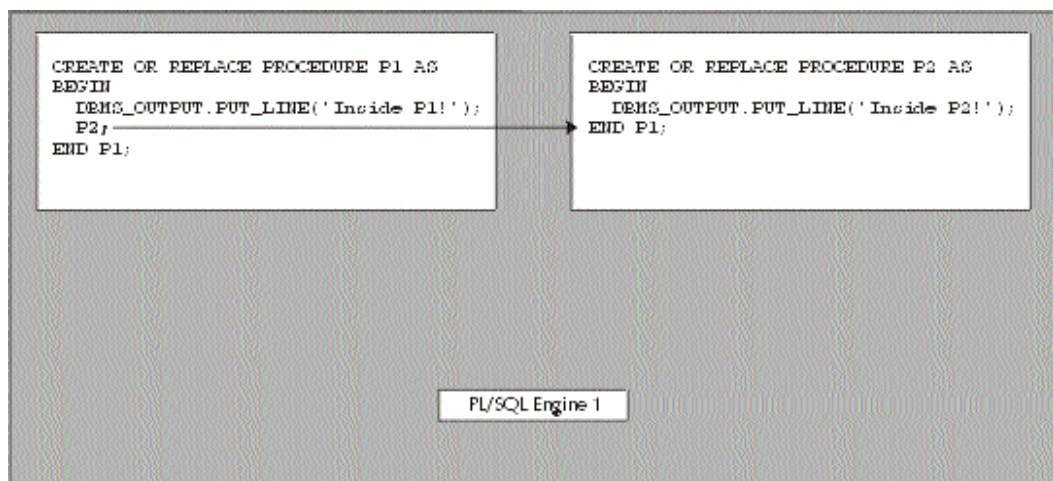


图5-9 在同一数据库中的过程P1和P2

节选自在线代码remoteDependencies.sql

```
SQL> -- Create two procedures. P1 depends on P2.
```

```
SQL> CREATE OR REPLACE PROCEDURE P2 AS
```

```
2 BEGIN
```

```
3   DBMS_OUTPUT.PUT_LINE('Inside P2!');
```

```
4 END P2;
```

```
5 /
```

Procedure created.

```
SQL> CREATE OR REPLACE PROCEDURE P1 AS
```

```
2 BEGIN
```

```
3   DBMS_OUTPUT.PUT_LINE('Inside P1!');
```

```
4   P2;
```

```
5 END P1;
```

```
6 /
```

Procedure created.

```
SQL> -- Verify that both procedures are valid.
```

```
SQL> SELECT object_name, object_type, status
```

```
2   FROM user_objects
```

```
3   WHERE object_name IN ('P1', 'P2');
```

OBJECT_NAME	OBJECT_TYPE	STATUS
P2	PROCEDURE	VALID
P1	PROCEDURE	VALID

```
SQL> -- Recompile P2, which invalidates P1 immediately.
```

```
SQL> ALTER PROCEDURE P2 COMPILE;
```

Procedure altered.

```
SQL> -- Query again to see this.
```

```
SQL> SELECT object_name, object_type, status
```

```
2 FROM user_objects
```

```
3 WHERE object_name IN ('P1', 'P2');
```

OBJECT_NAME	OBJECT_TYPE	STATUS
P2	PROCEDURE	VALID
P1	PROCEDURE	INVALID

假设，过程P1和P2位于不同的数据库，并且P1通过数据库连接来调用P2。图5-10所示的就是上述调用方式，在这种情况下，如果重编译P2将不会立即影响P1，下面的SQL \*Plus演示了这一过程：

节选自在线代码remoteDependencies.sql

```
SQL> -- Create a database link that points back to the current
```

```
SQL> -- instance.
```

```
SQL> CREATE DATABASE LINK loopback
```

```
2   USING 'connect_string';
```

Database link created.

SQL> -- Change P1 to call P2 over the link.

SQL> CREATE OR REPLACE PROCEDURE P1 AS

2 BEGIN

3 DBMS\_OUTPUT.PUT\_LINE('Inside P1!');

4 P2@loopback;

5 END P1;

6 /

Procedure created.

SQL> -- Verify that both are valid.

SQL> SELECT object\_name, object\_type, status

2 FROM user\_objects

3 WHERE object\_name IN ('P1', 'P2');

OBJECT_NAME	OBJECT_TYPE	STATUS
P2	PROCEDURE	VALID
P1	PROCEDURE	VALID

SQL> -- Now when we recompile P2, P1 is not invalidated immediately.

SQL> ALTER PROCEDURE P2 COMPILE;

Procedure altered.

SQL> SELECT object\_name, object\_type, status

2 FROM user\_objects

3 WHERE object\_name IN ('P1', 'P2');

OBJECT_NAME	OBJECT_TYPE	STATUS
P2	PROCEDURE	VALID
P1	PROCEDURE	VALID

注意 在这个例子中，数据库连接实际上是一个回送环（loopback），总是指向相同的数据库。然而，我们所看到的就象 P1 和 P2 各自处于不同的数据库一样。使用回送环可以使我们在一个 SELECT 语句中查询 P1 和 P2 的状态。

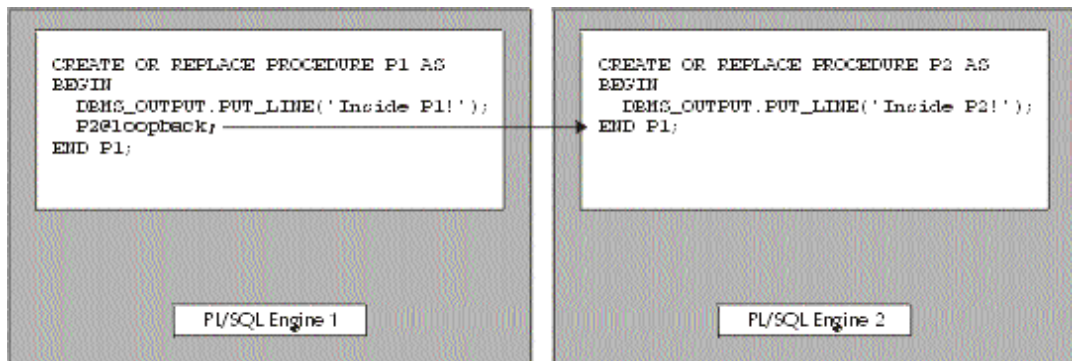


图5-10 处于不同数据库中的过程P1和P2



为什么在远程调用下的过程看起来有所不同呢？答案就在于数据字典并不跟踪远程相关对象。实际上，由于远程对象可能位于不同的数据库中，因此要将所有相关远程对象作废实际上是不可能的（如果远程对象处于无效期的话，数据字典可能无法对其进行访问）。

与上不同的是，远程对象的合法性要在运行时进行检查。当过程 P1 被调用时，就要访问远程数据字典来确定过程 P2 的状态（如果远程数据库不能访问的话，就会引发异常）。这时，要对 P1 和 P2 进行比较以决定是否对过程 P1 需要重新编译。比较过程的方法有两个，一种方法是时间戳法（Timestamp），另一种是标记法（Signature）。

**注意** 实际上没有必要使用数据库连接来进行运行合法检查。如果 P1 在客户端的 PL/SQL 引擎中（如在 Oracle Forms 中运行），而 P2 在服务器端，那么情况就会类似，上述两种比较方法都可以使用。有关 PL/SQL 运行环境的介绍，请看本书第 2 章内容。

**时间戳模型** 在这种模型下，将对最后修改的两个对象的时间戳进行比较。表 user\_objects 的 LAST\_DDL\_TIME 字段包含有对象的时间戳。如果底层对象的时间戳要比其相关的对象的时间戳新，则相关对象将进行重编译。然而，时间戳模型有下列问题需要注意：

- 时间的比较并没有把两个对象所在的 PL/SQL 引擎的位置考虑在内。如果两个引擎位于不同的时区的话，那么这种比较就没有意义。
- 即使上述两个引擎位于同一时区，时间戳法仍然会引起不必要的重编译。在上面的例子中，P2 实际上没有变更，但还是进行了重编译。这时，P1 没有必要重编译，但由于 P1 的时间戳老一些，故还要对其重编译。
- 稍微严重的问题是，当 P1 属于客户端 PL/SQL 引擎时，如运行在 Oracle Forms 软件下。在这种情况下，由于该过程的源代码可能不在 Forms 软件的运行版本中存储，所以无法对其重编译。

**PL/SQL 2.3 及更高版本** **标记法模型** 从 PL/SQL 2.3 版开始，PL/SQL 提供了另一种叫做标记法的比较算法用来克服时间戳法存在的问题。每当创建一个过程时，除去中间代码外，还把一个标记存储在该过程的数据字典中。该标记将过程的类型和参数顺序进行编码。使用这种方法，过程 P2 的标记将只在其参数变更时改变。当过程 P1 第一次编译时，P2 的标记就被加入（不是记录时间戳）。因此，过程 P1 只有在过程 P2 变更时才需要重编译。

为了使用标记法，要将系统参数 REMOTE\_DEPENDENCIES\_MODE 设置为 SIGNATURE。它是数据库初始化文件中的一个参数。（该初始化文件一般是 init.ora，其名称和位置与数据库系统有关。）该参数也可交互设置。下面是设置该参数的三种方法：

- 把命令行 REMOTE\_DEPENDENCIES\_MODE= SIGNATURE 加入到数据库初始文件中。当数据库再次启动时，将为所有会话设置 SIGNATURE。
- 提交下面的命令

```
ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE=SIGNATURE;
```

该命令将从其提交开始对所有数据库会话生效。发布该命令要具有 ALTER SYSTEM 的系统特权。

- 提交下面的命令

```
ALTER SESSION SET REMOTE_DEPENDENCIES_MODE=SIGNATURE;
```

该命令只对会话有效。在该命令后当前会话中创建的对象将使用标记法。

在以上的所有选择项中，参数 `TIMESTAMP` 可以用来取代参数 `SIGNATURE` 来适应 PL/SQL 2.2 版或更低版本的环境。参数 `TIMESTAMP` 是默认值。

下面是使用标记法的一些注意事项：

- 如果形参的默认值变更的话，标记将不被修改。假设过程 P2 的一个参数有默认值，而过程 P1 正在使用该默认值。如果在 P2 的说明部分修改了该默认值，则根据默认规则，P1 将不重编译。这样一来，除非人工重编过程 P1，否则该默认参数的旧值仍将被使用。以上规则仅适用具有 IN 属性的参数。
- 如果过程 P1 正在调用过程 P2，而 P2 的新的重载版本已经追加到远程包中，这时，标记不做变动。过程 P1 仍将使用旧的版本（不是新的重载版本）直到过程 P1 手工重编译为止。
- 要用手对过程进行重编，请使用下面的命令：

```
ALTER PROCEDURE procedure_name COMPILE;
```

其中，`procedure_name` 是要编译的过程名。对于函数，请使用下面的命令：

```
ALTER FUNCTION function_name COMPILE;
```

对于包，可以使用下面两个命令中的一个：

```
ALTER PACKAGE package_name COMPILE SPECIFICATION;
```

```
ALTER PACKAGE package_name COMPILE BODY;
```

有关标记方法的详细介绍，请参考 Oracle 服务器应用开发指南 7.3 版以上的文档资料。

### 5.2.2 包运行时状态

当对包进行第一次实例时，将从磁盘读入该包的中间代码代码并将其放入系统全局工作区 SGA 的共享缓冲区中。然而，包的运行状态，即打包的变量和游标，将存放在用户全局区 (UGA) 的会话存储区中。这就保证了每个会话都将有其自己包运行状态的副本。正如我们在第 4 章中看到的，包头中声明的变量的作用域为全局范围，这些变量对于具有 EXECUTE 特权的任何 PL/SQL 块都是可见的。由于包的运行状态是在 UGA 中存放的，所以它们具有与数据库会话相同的生存期。当包被实例时，其运行状态也得到初始化（包中的初始化代码将启动运行），并且这些状态直到会话结束才被释放。即使包本身由于超时被从共享缓冲区中清除，但该包的状态仍将持续。下面的例子演示了包运行状态：

节选自在线代码 `PersistPkg.sql`

```
CREATE OR REPLACE PACKAGE PersistPkg AS
-- Type which holds an array of student ID's.
TYPE t_StudentTable IS TABLE OF students.ID%TYPE
INDEX BY BINARY_INTEGER;

-- Maximum number of rows to return each time.
v_MaxRows NUMBER := 5;

-- Returns up to v_MaxRows student ID's.
```

```

PROCEDURE ReadStudents(p_StudTable OUT t_StudentTable,
                      p_NumRows   OUT NUMBER);

END PersistPkg;

CREATE OR REPLACE PACKAGE BODY PersistPkg AS
-- Query against students. Since this is global to the package
-- body, it will remain past a database call.
CURSOR StudentCursor IS
    SELECT ID
      FROM students
     ORDER BY last_name;

PROCEDURE ReadStudents(p_StudTable OUT t_StudentTable,
                      p_NumRows OUT NUMBER) IS
    v_Done BOOLEAN := FALSE;
    v_NumRows NUMBER := 1;
BEGIN
    IF NOT StudentCursor%ISOPEN THEN
        -- First open the cursor
        OPEN StudentCursor;
    END IF;

    -- Cursor is open, so we can fetch up to v_MaxRows
    WHILE NOT v_Done LOOP
        FETCH StudentCursor INTO p_StudTable(v_NumRows);
        IF StudentCursor%NOTFOUND THEN
            -- No more data, so we're finished.
            CLOSE StudentCursor;
            v_Done := TRUE;
        ELSE
            v_NumRows := v_NumRows + 1;
            IF v_NumRows > v_MaxRows THEN
                v_Done := TRUE;
            END IF;
        END IF;
    END LOOP;

    -- Return the actual number of rows fetched.
    p_NumRows := v_NumRows - 1;

END ReadStudents;

END PersistPkg;

```

过程PersistPkg.ReadStudents将从游标StudentCursor中进行选择。由于该游标是在包一级声明的（不是在过程ReadStudents中声明的），该游标仍将保持过去对ReadStudents的调用。我们可以使用下面的块来调用过程PersistPkg.ReadStudents：

节选自在线代码callRS.sql

```
DECLARE
    v_StudentTable PersistPkg.t_StudentTable;
    v_NumRows NUMBER := PersistPkg.v_MaxRows;
    v_FirstName students.first_name%TYPE;
    v_LastName students.last_name%TYPE;
BEGIN
    PersistPkg.ReadStudents(v_StudentTable, v_NumRows);
    DBMS_OUTPUT.PUT_LINE(' Fetched ' || v_NumRows || ' rows:');
    FOR v_Count IN 1..v_NumRows LOOP
        SELECT first_name, last_name
            INTO v_FirstName, v_LastName
            FROM students
            WHERE ID = v_StudentTable(v_Count);
        DBMS_OUTPUT.PUT_LINE(v_FirstName || ' ' || v_LastName);
    END LOOP;
END;
```

图5-11是执行该块三次的输出结果。对于每次调用，由于该游标一直在两次调用间保持打开状态，所以每次都返回不同的数据。

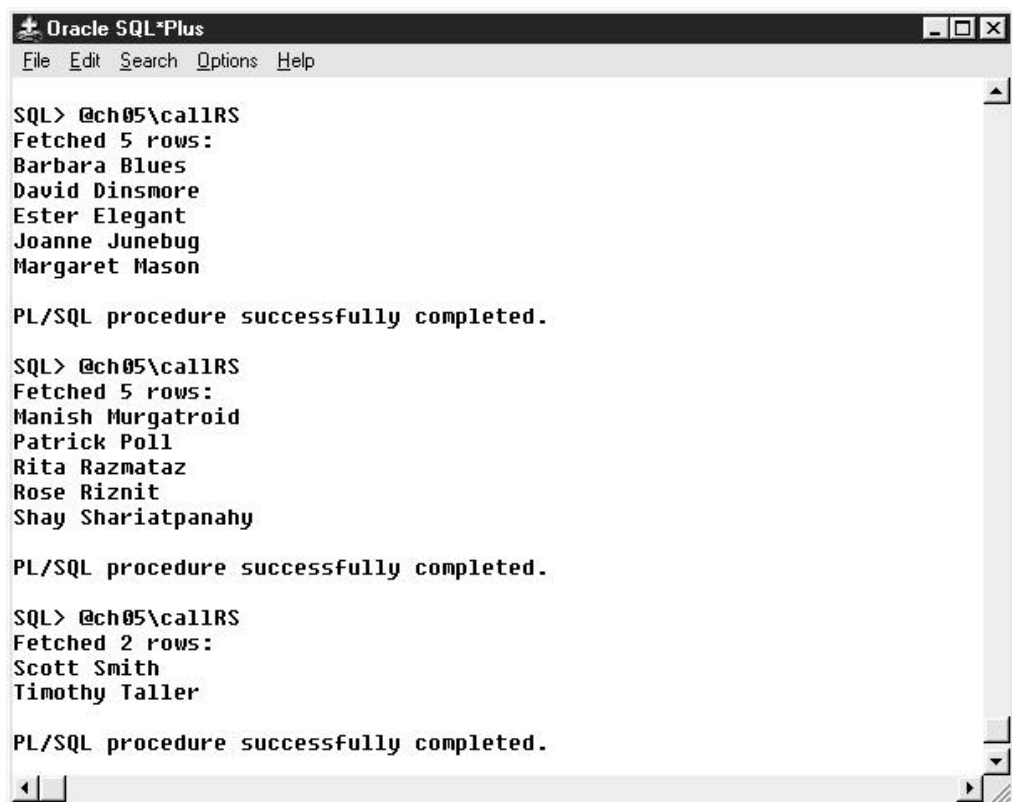


图5-11 调用过程ReadStudents的结果

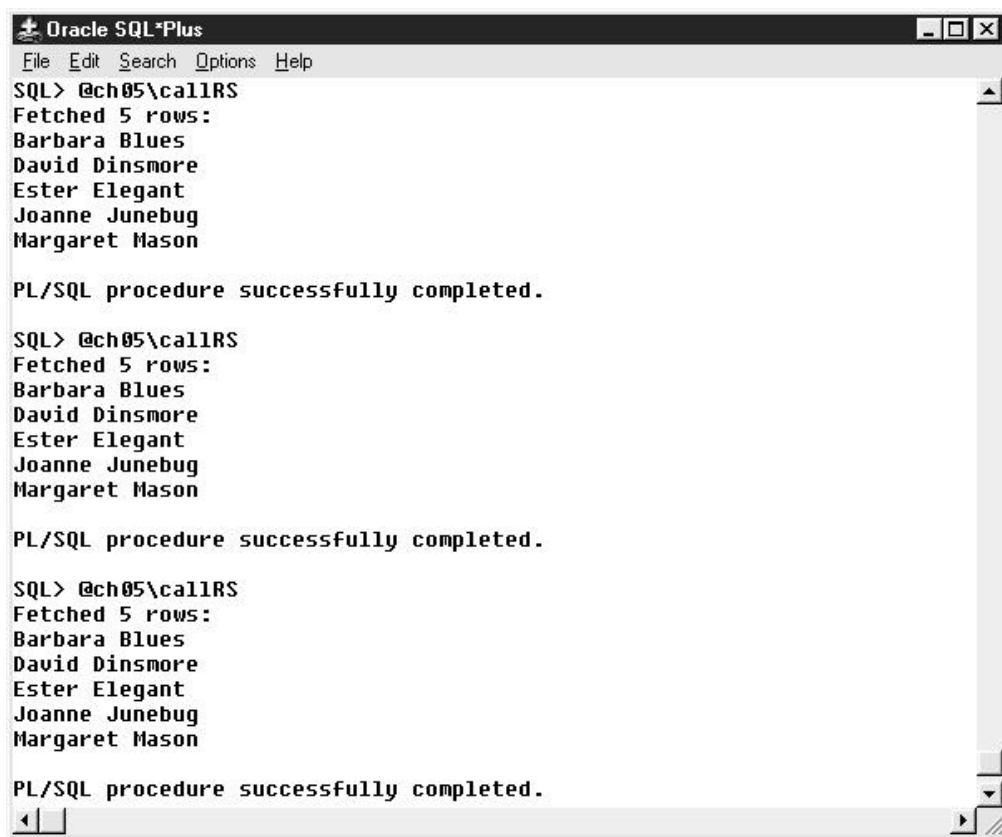
## 1. 可连续重用包

PL/SQL 2.3 及  
更高版本

PL/SQL 2.3 版及更高版本允许程序员对包做连续可重用标志。可连续重用包的运行状态将保存在 SGA 而不是 UGA 中，并仅在每个数据库调用期间有效。可连续重用包的语法是：

```
PRAGMA SERIALLY_REUSABLE;
```

该语句应在包头中（如果有包体的话，也在包体中）中使用。如果我们修改 PersistPkg 使其包括这个 PRAGMA，其输出也将发生变化。图 5-12 是该包的输出信息。下面的程序是修改过的包 PersistPkg：



```
Oracle SQL*Plus
File Edit Search Options Help
SQL> @ch05\callRS
Fetched 5 rows:
Barbara Blues
David Dinsmore
Ester Elegant
Joanne Junebug
Margaret Mason

PL/SQL procedure successfully completed.

SQL> @ch05\callRS
Fetched 5 rows:
Barbara Blues
David Dinsmore
Ester Elegant
Joanne Junebug
Margaret Mason

PL/SQL procedure successfully completed.

SQL> @ch05\callRS
Fetched 5 rows:
Barbara Blues
David Dinsmore
Ester Elegant
Joanne Junebug
Margaret Mason

PL/SQL procedure successfully completed.
```

图5-12 调用可连续重用包 ReadStudents

节选自在线代码 PersistPkg2.sql

```
CREATE OR REPLACE PACKAGE PersistPkg AS
```

```
    PRAGMA SERIALLY_REUSABLE;
```

```
-- Type that holds an array of student IDs.
```

```
TYPE t_StudentTable IS TABLE OF students.ID%TYPE
```

```
    INDEX BY BINARY_INTEGER;
```

```
-- Maximum number of rows to return each time.
v_MaxRows NUMBER := 5;

-- Returns up to v_MaxRows student IDs.
PROCEDURE ReadStudents(p_StudTable OUT t_StudentTable,
                      p_NumRows OUT NUMBER);

END PersistPkg;

CREATE OR REPLACE PACKAGE BODY PersistPkg AS
  PRAGMA SERIALLY_REUSABLE;

  -- Query against students. Even though this is global to the
  -- package body, it will be reset after each database call,
  -- because the package is now serially reusable.
  CURSOR StudentCursor IS
    SELECT ID
      FROM students
     ORDER BY last_name;
  ...
END PersistPkg;
```

值得注意的是包 PersistPkg 的两个版本之间的差别。非连续重用包版的程序将跨越数据库来维持游标状态，而可连续重用包版程序每次都要复位其状态（以及输出）。这两种版本的区别在表5-2中给出：

表5-2 包的两种版本的区别

可连续重用包	非连续重用包
运行状态保存在SGA中，每次数据库调用后都将该运行状态释放	运行状态保存在UGA中，其生存期与数据库会话相同
所用的最大内存与包的并发用户数量成正比	所用的最大内存与当前登录的用户数目成正比

注意 可连续重用包的语义在 MTS（多线程服务器）下仍保持不变。在 MTS 环境下，UGA 将存储在共享存储器中以便会话可以在数据库服务器进程间迁移。因为可连续重用包减少了对内存的需求，所以它们在 MTS 下具有优势。有关 MTS 的介绍，请参阅 Oracle 文档。

2. 包运行状态的相关

除了在存储对象之间的存在着相关外，包状态和匿名块之间也有相关特性。例如，请看下面的包：

```
节选自在线代码anonymousDependencies.sql
CREATE OR REPLACE PACKAGE SimplePkg AS
  v_GlobalVar NUMBER := 1;
  PROCEDURE UpdateVar;
END SimplePkg;
```

```
CREATE OR REPLACE PACKAGE BODY SimplePkg AS
  PROCEDURE UpdateVar IS
  BEGIN
    v_GlobalVar := 7;
  END UpdateVar;
END SimplePkg;
```

包SimplePkg包含了一个包全局量 v\_GlobalVar.假设我们从一个数据库会话创建包 SimplePkg.接着,在第二个会话中,我们使用下面的块来调用 SimplePkg.UpdateVar:

```
BEGIN
  SimplePkg.UpdateVar;
END;
```

现在返回第一个会话,我们运行创建脚本再次创建包 SimplePkg.最后,我们在第二个会话中提交同样的匿名块。下面是得到的输出信息:

```
BEGIN
*
ERROR at line 1:
ORA-04068: existing state of packages has been discarded
ORA-04061: existing state of package "EXAMPLE.SIMPLEPKG" has been
            invalidated
ORA-04065: not executed, altered or dropped package
            "EXAMPLE.SIMPLEPKG"
ORA-06508: PL/SQL: could not find program unit being called
ORA-06512: at line 2
```

上面的程序发生了什么问题?图 5-13是上述情况的相关图示。匿名块依赖于包 SimplePkg.这种相关是编译时的依赖性,也就是在匿名块首次编译时就确定的相关关系。然而,除此之外,由于每个会话都有其自己包变量的复本,所以运行时包变量之间也存在着依赖关系。因此,当重编SimplePkg时,运行时相关就紧随其后,引发了错误 ORA-4068并作废了该块。

运行时相关仅存在于包状态之上,它包括包中的变量和游标声明。如果包没有全局变量的话,则匿名块的第二次运行将会成功。

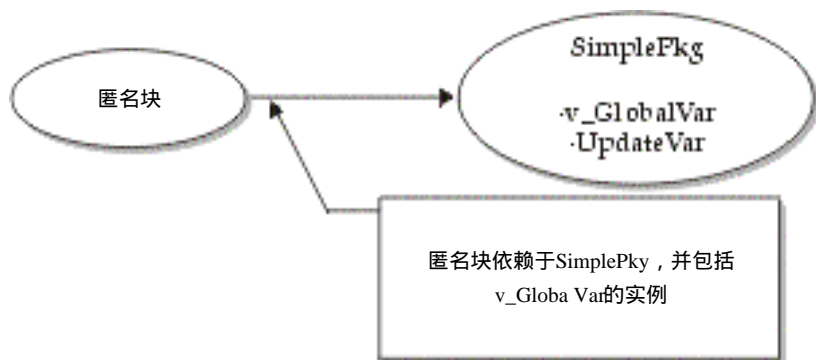


图5-13 包的全局相关图示

### 5.2.3 特权和存储子程序

存储子程序和包都是数据库字典中的对象，因而，它们属于特殊的数据库用户或模式。如果用户被授予了正确的特权，则它们就可以访问这些对象。特权和角色在创建存储对象时也与子程序内部可行的访问一起开始活动。

#### 1. EXECUTE特权

为了能够对表进行访问，必须使用 SELECT，INSERT，UPDATE和DELETE对象的特权。语句 GRANT把这些特权赋予数据库用户或角色。对于存储子程序和包来说，相关的特权是 EXECUTE。现在请看下面的过程 RecordFullClasses，该程序是5.2.1节中的案例：

节选自在线代码execute.sql

```
CREATE OR REPLACE PROCEDURE RecordFullClasses AS
    CURSOR c_Classes IS
        SELECT department, course
        FROM classes;
BEGIN
    FOR v_ClassRecord IN c_Classes LOOP
        -- Record all classes that don't have very much room left
        -- in temp_table.
        IF AlmostFull(v_ClassRecord.department, v_ClassRecord.course) THEN
            INSERT INTO temp_table (char_col) VALUES
                (v_ClassRecord.department || ' ' || v_ClassRecord.course ||
                 ' is almost full!');
        END IF;
    END LOOP;
END RecordFullClasses;
```

注意 在线案例execute.sql将首先创建用户UserA和用户B，接着再创建本节案例需要的对象。读者可以修改用于DBA帐户的口令，以便使该案例可以运行在读者所在的系统上。除此之外，我们还提供了演示该程序输出结果的程序 execute.out。

假设过程RecordFullClasses相关的对象（即函数AlmostFull,表classes和temp\_table）都属于数据库用户UserA。并且RecordFullClasses也属于UserA。如果我们将RecordFullClasses的特权赋予其他的数据库用户，如UserB,如下的命令所示：

节选自在线代码execute.sql

```
GRANT EXECUTE ON RecordFullClasses TO UserB;
```

这样一来，UserB就可以用下面的块来运行RecordFullClasses。下面语句中使用的点符号用于指示模式：

节选自在线代码execute.sql

```
BEGIN
    UserA.RecordFullClasses;
END;
```

在这种情况下，所有的数据库对象都属于 UserA。图5-14演示了UserA的所属的对象。该图



中的虚线表示从UserA到UserB的GRANT语句，而图中的实线则表示对象的相关。在运行了本节前面的该块代码后，其结果将插入表 UserA.temp\_table中。

现在假设 UserB有另外一个叫做 temp\_table的表，如图 5-15所示。如果 UserB调用 UserA.

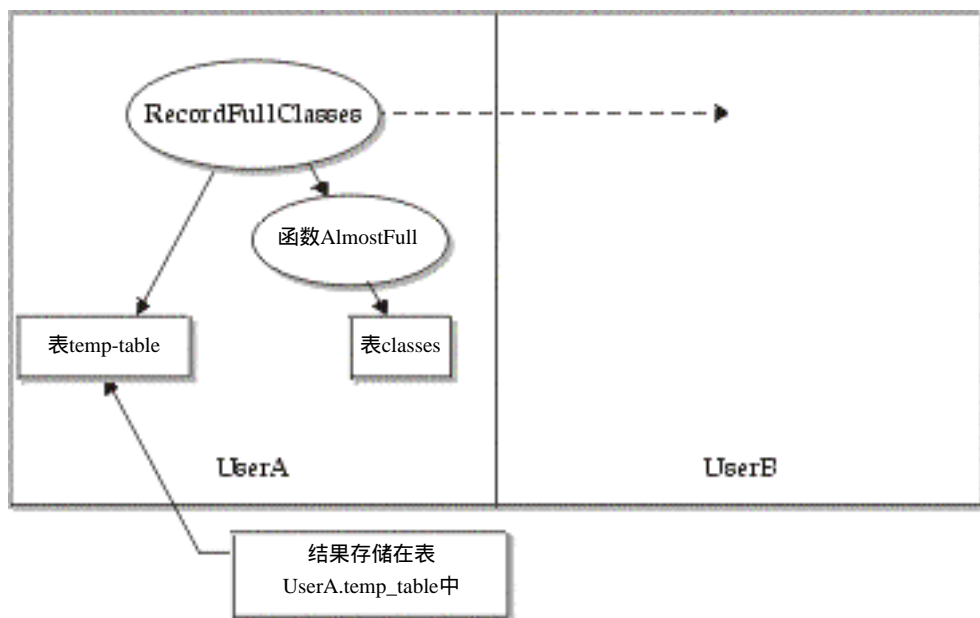


图5-14 UserA所属的数据库对象

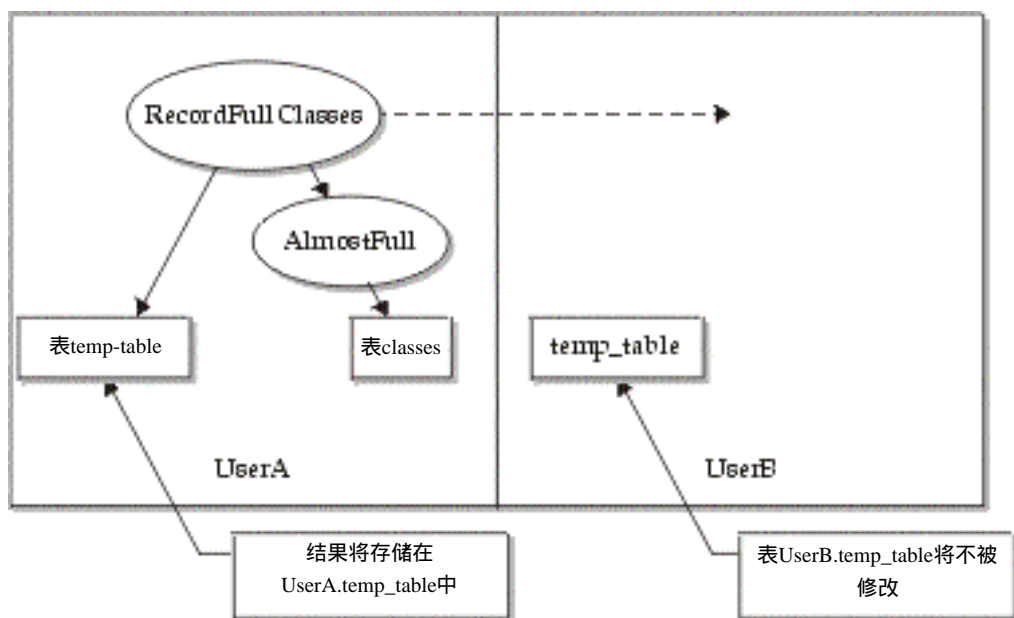


图5-15 UserA和UserB所属的表temp\_table

RecordFullClasses(通过运行前面介绍的匿名块), 哪个表将被修改呢? 答案是 UserA的表将被修改。上述概念可以如下描述:

子程序在其拥有者的优先集下运行

即使UserB正在调用属于 UserA的RecordFullClasses。但标识符 temp\_table经过求值也将指向属于 UserA, 而不是 UserB的表。

Oracle 8i 及  
更高版本

Oracle8i提供一个叫做调用权的新功能, 借助于调用权, 可以指定一个过程是否在其拥有者或其调用者的特权下运行。详细内容请参阅下面的“调用权与定义权的比较”

内容。

## 2. 存储子程序和角色

现在让我们对图 5-15 所示的情况做一点修改。假设 UserA不拥有表 temp\_table或 RecordFullClasses, 而UserB则拥有这两个对象。我们再进行假设已经对 RecordFullClasses做了修改使其显式地引用 UserA所属的对象。修改后的程序如下所示:

节选自在线代码execute.sql

```
CREATE OR REPLACE PROCEDURE RecordFullClasses AS
  CURSOR c_Classes IS
    SELECT department, course
      FROM UserA.classes;
BEGIN
  FOR v_ClassRecord IN c_Classes LOOP
    -- Record all classes that don't have very much room left
    -- in temp_table.
    IF UserA.AlmostFull(v_ClassRecord.department,
                       v_ClassRecord.course) THEN
      INSERT INTO temp_table (char_col) VALUES
        (v_ClassRecord.department || ' ' || v_ClassRecord.course ||
         ' is almost full!');
    END IF;
  END LOOP;
END RecordFullClasses;
```

为了能够正确地编译 RecordFullClasses, UserA必须把表 classes的 SELECT特权 and AlmostFull的EXECUTE特权赋予 UserB。图5-16中的虚线代表了这种授权。同时, 该授权必须显式地执行, 而不能通过角色来实现。由 UserA执行的下列授权将保证 UserB. RecordFullClassesd 的编译成功:

节选自在线代码execute.sql

```
GRANT SELECT ON classes TO UserB;
GRANT EXECUTE ON AlmostFull TO UserB;
```

而下面通过中间角色实现的授权将不起作用。图 5-17显示了该角色的作用。

节选自在线代码execute.sql

```
CREATE ROLE UserA_Role;
GRANT SELECT ON classes TO UserA_Role;
GRANT EXECUTE ON AlmostFull TO UserA_Role;
```

```
GRANT UserA_Role to UserB;
```

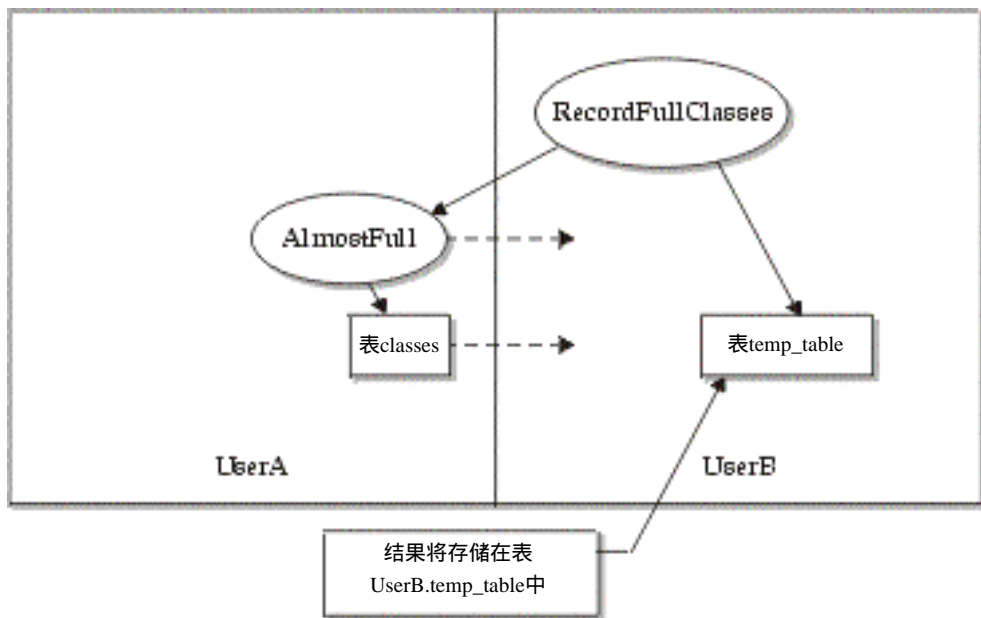


图5-16 UserB所属的RecordFullClasses

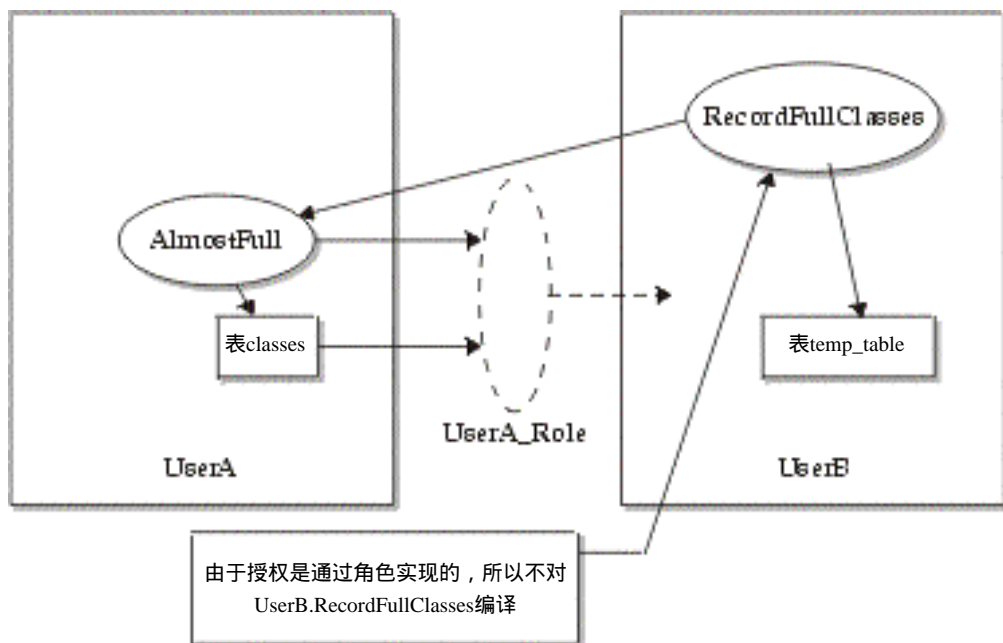


图5-17 通过角色实现授权

根据上面的例子，我们可以把上一节中的内容总结如下：

子程序运行在其被显示地授权的所有者的特权下。

如果通过角色实现授权，则当我们试图编译 RecordFullClasses时就会收到如下所示的 PL-201 错误提示：

```
PLS-201: identifier 'CLASSES' must be declared
PLS-201: identifier 'ALMOSTFULL' must be declared
```

该规则也适用于存储在数据库中的触发器和包。特别对于存储过程内部的函数，包，或触发器的对象（Oracle 7.3 及更高版本）都是子程序所有者所属的对象，或者是显式赋予拥有者的对象。

为什么有这种限制呢？为了回答这个问题，需要我们来介绍有关联编的概念。PL/SQL 使用了前联编技术，也就是在编译子程序时而不是在运行时就对引用求值。语句 GRANT 和 REVOKE 都是 DDL 命令，该命令运行时立即生效，其新的特权将记录在数据字典中。所有数据库会话都可以看到该新特权组。然而，对于角色来说，这种方法就没有必要。角色可以赋予一个用户，用户可以使用 SET ROLE 命令来选择取消角色。其区别是命令 SET ROLE 只能作用在一个数据库会话上，而 GRANT 和 REVOKE 可以对所有会话有效。我们可以在一个会话中禁止一个角色，而在另一个会话中启动该角色。

为了实现将通过角色授权的特权作用在子程序内部和触发器中，就必须在每次运行过程时对该特权进行检查。编译程序在其联编中对特权进行检查。但前联编是在编译时检查特权，而不是在运行时检查。为了保证前联编执行，存储过程和触发器内部的所有角色都将被禁止。

### 3. 调用权与定义权的比较

Oracle 8i 及  
更高版本

现在让我们来考虑本章 5.2.3 节中介绍的例子，以及图 5-15 演示的相关图示。在上述情况下，UserA 和 UserB 都有表 temp\_table 的副本，由于 RecordFullClasses 属于 UserA，所以插入 UserA.temp\_table.RecordFullClasses 就被叫做定义权过程，其原因就是其内部不合格的外部引用是在其所有者或定义者的特权下实现的。

Oracle 8i 提供了不同的外部引用解决方案。在调用权子程序中，外部引用是通过调用者而不是拥有者的特权组实现的。调用权程序是由 AUTHID 子句实现的，该语句只适用于独立子程序，包说明和对象类型说明。在包内部或对象类型中的独立子程序必须都是调用权或都是定义权，不能有混合类型。AUTHID 的语法如下：

```
CREATE [OR REPLACE] FUNCTION function_name
[ parameter_list ] RETURN return_type
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}
function_body;
```

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[ parameter_list ]
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}
function_body;
```

```
CREATE [OR REPLACE] PACKAGE package_spec_name
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}
```

```
package_spec;
```

```
CREATE [OR REPLACE] TYPE type_name
  [AUTHID {CURRENT_USER | DEFINER}] {IS | AS} OBJECT
  type_spec;
```

如果在子句AUTHID中说明了参数CURRENT\_USER，则该对象将具有调用权。如果说明了DEFINER，则该对象就具有定义权。如果没有使用AUTHID子句的话，其默认值将是定义权。

例如，下面版本的RecordFullClasses是调用权过程：

节选自在线代码invokers.sql

```
CREATE OR REPLACE PROCEDURE RecordFullClasses
  AUTHID CURRENT_USER AS

  -- Note that we have to preface classes and AlmostFull with
  -- UserA, since both of these are owned by UserA only.
  CURSOR c_Classes IS
    SELECT department, course
      FROM UserA.classes;
BEGIN
  FOR v_ClassRecord IN c_Classes LOOP
    -- Record all classes that don't have very much room left
    -- in temp_table.
    IF UserA.AlmostFull(v_ClassRecord.department,
                       v_ClassRecord.course) THEN
      INSERT INTO temp_table (char_col) VALUES
        (v_ClassRecord.department || ' ' || v_ClassRecord.course ||
         ' is almost full!');
    END IF;
  END LOOP;
END RecordFullClasses;
```

注意 在线案例invokers.sql 将首先创建用户UserA和UserB，接着再创建本节案例需要的对象。读者可以修改用于DBA帐户的口令，以便使该案例可以运行在读者所在的系统上。除此之外，我们还提供了演示该程序输出结果的程序 invokers.out。

如果UserB运行了RecordFullClasses，插入操作将在表UserB.temp\_table上执行。如果UserA运行该过程，则插入操作将在表UserA.temp\_table上执行。下面的SQL \*Plus会话和图5-18演示了上述执行过程：

节选自在线代码invokers.sql

```
SQL> connect UserA/UserA
Connected.
SQL> -- Call as UserA. This will insert into UserA.temp_table.
SQL> BEGIN
  2   RecordFullClasses;
  3   COMMIT;
  4 END;
  5 /
```

PL/SQL procedure successfully completed.

SQL> -- Query temp\_table. There should be 1 row.

SQL> SELECT \* FROM temp\_table;

NUM\_COL CHAR\_COL

MUS 410 is almost full!

SQL> -- Connect as UserB.

SQL> -- Now the call to RecordFullClasses will insert into

SQL> -- UserB.temp\_table.

SQL> BEGIN

2 UserA.RecordFullClasses;

3 COMMIT;

4 END;

5 /

PL/SQL procedure successfully completed.

SQL> -- So we should have one row here as well.

SQL> SELECT \* FROM temp\_table;

NUM\_COL CHAR\_COL

MUS 410 is almost full!

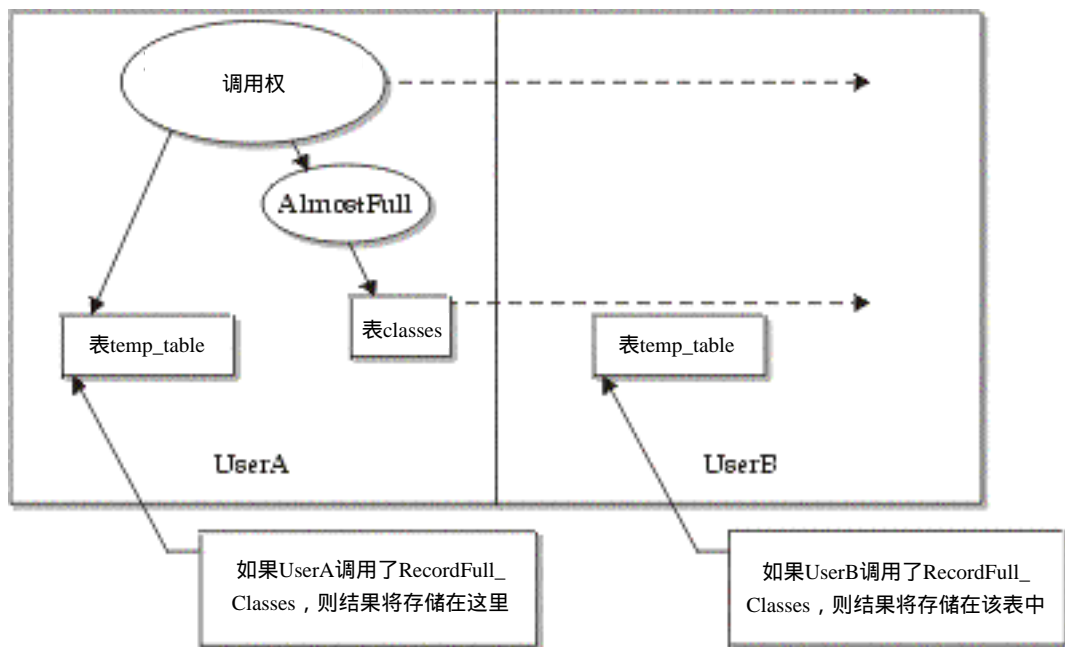


图5-18 具有调用权的过程 RecordFullClasses

使用调用权的解决方案 在调用权子程序中，SQL语句中的外部引用将通过调用者特权组求

值。然而，在PL/SQL语句中的引用（如赋值语句和过程调用语句）还是在其拥有者的特权组下求值。这是为什么呢？图 5-18中，GRANT语句仅作用在过程 RecordFullClasses和表classes上。由于对AlmostFull的调用是一个PL/SQL语句，所以该调用总是在UsreA特权组下实现，因此，不必对用户B使用GRANT语句。

然而，假设对表 classes的GRANT语句没有实现。这时，由于所有的 SQL对象都可以在 UserA的特权组下访问，所以 UserA可以成功地编译该过程。但是 UserB将会在调用过程 RecordFullClasses时收到ORA-942的错误信息。图 5-19和下面的SQL \*Plus会话演示了上述情况：

节选自在线代码invokers.sql

```
SQL> connect UserB/UserB
```

```
Connected.
```

```
SQL> BEGIN
```

```
2   UserA.RecordFullClasses;
```

```
3 END;
```

```
4 /
```

```
BEGIN
```

```
*
```

```
ERROR at line 1:
```

```
ORA-00942: table or view does not exist
```

```
ORA-06512: at "USERA.RECORDFULLCLASSES", line 7
```

```
ORA-06512: at "USERA.RECORDFULLCLASSES", line 10
```

```
ORA-06512: at line 2
```

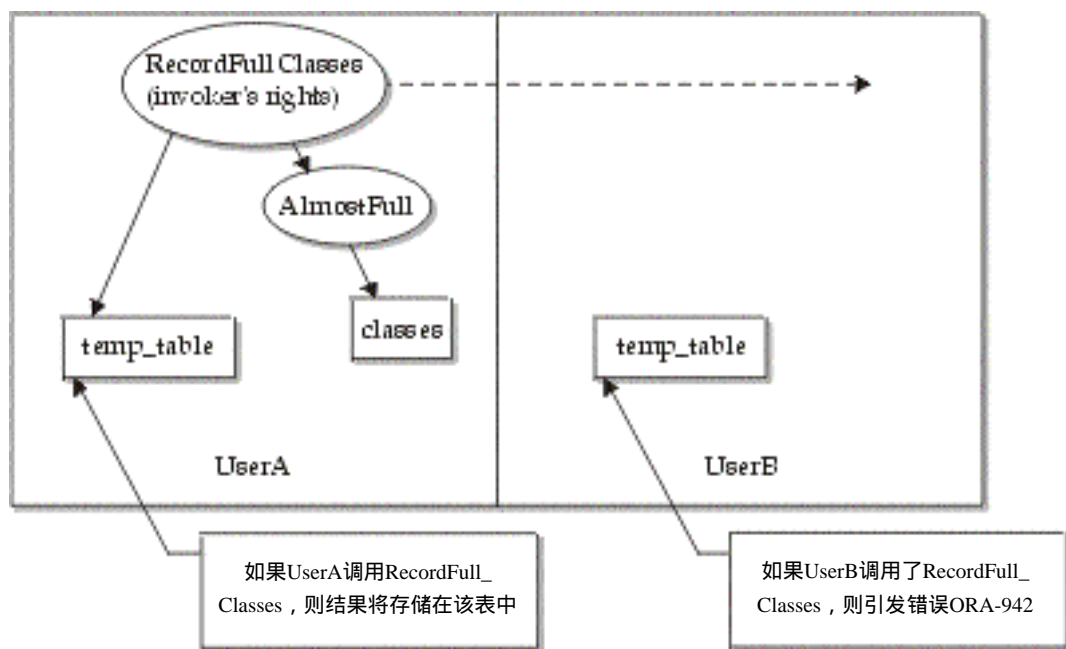


图5-19 撤消对表classes的SELECT操作

注意 这里收到的错误信息是 ORA-942，而不是 PLS-201。该错误是数据库编译错误，

但我们却在运行时接收了该信息。

**角色和调用权** 假设对表classes的授权语句GRANT是通过角色间接实现的。请回忆一下我们在图5-17中演示的定义权过程必须进行显式授权的规则。对于调用权程序来说，该规则不适用。由于对调用权程序的外部引用是在运行时实现的，所以当前的特权组是可用的。这就说明通过角色赋予调用者的特权将可以访问。图5-20和下面的SQL \*Plus会话演示了上述过程：

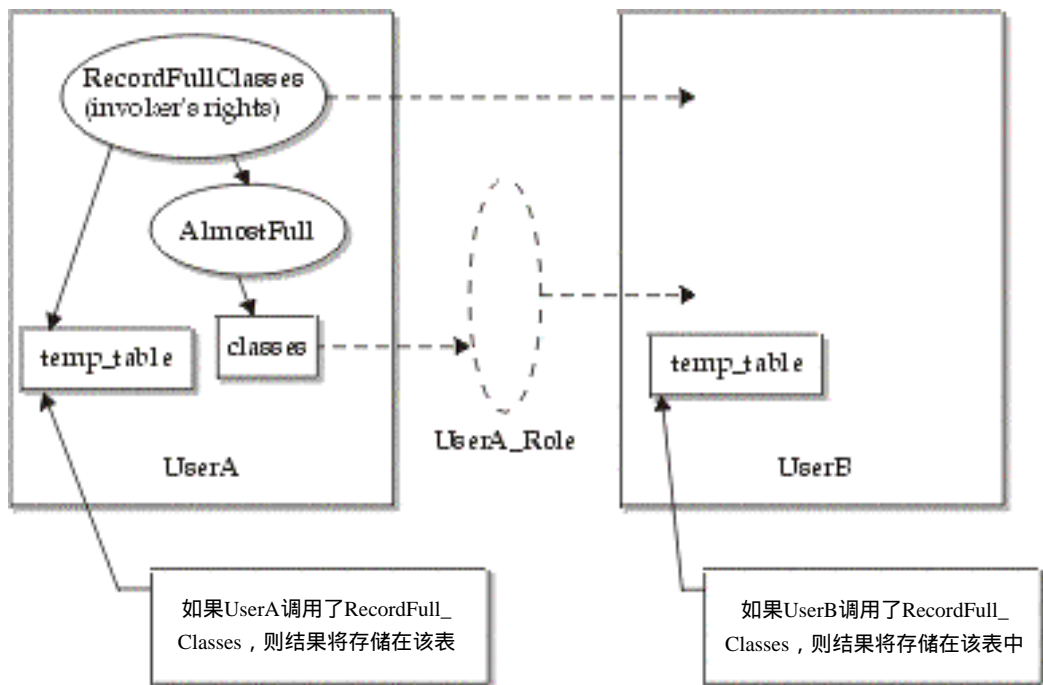


图5-20 角色和调用权图示

节选自在线代码invokers.sql

```
SQL> connect UserA/UserA
Connected.
CREATE ROLE UserA_Role;
Role created.
SQL> GRANT SELECT ON classes TO UserA_Role;
Grant succeeded.
SQL> GRANT UserA_Role TO UserB;
Grant succeeded.
SQL> -- Connect as UserB and call.
SQL> connect UserB/UserB
Connected.
SQL> -- Now the call to RecordFullClasses will succeed.
SQL> BEGIN
2   UserA.RecordFullClasses;
3   COMMIT;
```



```
4 END;  
5 /  
PL/SQL procedure successfully completed.
```

注意 在过程编译时求值的引用也必须直接授权。只有在运行时求值的引用可以通过角色间接授权。该规则也说明命令 SET ROLE（如果在动态SQL下执行）可以与运行时引用一同使用。

外部例程和调用权 按默认值，使用Java语言编制的外部例程（也叫做Java存储过程）将继承调用权，这一点与PL/SQL例程的默认值不同。该规则与Java方法调用模式保持一致。如果希望Java存储过程带有定义权运行的话，可以使用子句 AUTHID DEFINER来说明调用。本书的第10章将对这一规则做进一步介绍。

触发器、视图和调用权 数据库触发器总是带有定义权，并运行在其拥有触发器表模式的特权下。该规则也适用于从视图中调用的PL/SQL函数。这时，该函数将运行在其视图的拥有者的特权下。

## 5.3 在SQL语句中使用存储函数

总的来说，由于子程序调用是程序性命令，所以不能从SQL语句中调用过程。然而，从PL/SQL2.1版开始对存储函数取消了该限制。如果独立的或打包的函数满足某些条件的话，就可以从SQL语句的运行中对其进行调用。该功能只能用于PL/SQL2.1版（Oracle7的7.1版）及更高版本。Oracle8i则对该功能进行了进一步加强。

用户定义函数的调用方法与内置函数如 TO\_CHAR, UPPER, 或 ADD\_MONTHS 等使用方法相同。根据用户定义函数的使用位置以及程序使用的 Oracle 数据库版本，这种调用要满足不同的要求。调用限制按纯层定义。

### 5.3.1 纯层

函数有四个不同的纯层（purity levels），纯层定义了函数所读或修改的数据结构。表 5-3 列出了可用的纯层。根据函数所处的纯层，调用将受到如下限制：

- 从SQL语句中执行的任何函数调用不能修改任何数据库表（WNDS）。（在Oracle8i中，从非SELECT语句中执行的函数调用则可以修改数据库表。请看 5.3.3节。）
- 为了实现远程运行（通过数据库连接）或并行运行，函数不能读或写包变量的值（RNPS 和 WNPS）。
- 从SELECT, VALUES, 或SET子句执行的函数调用可以写包变量。在所有其他子句中的函数都必须是WNPS纯层。
- 函数要与它所调用的子程序具有同样的纯层。例如，如果函数调用了执行UPDATE功能的存储过程的话，该函数就不具有WNDS纯层，因此，该调用不能使用在选择语句的内部。
- 不管存储PL/SQL函数具有何种纯层，都不能从CREATE TABLE或ALTER TABLE命令的CHECK限制子句中调用该存储PL/SQL函数。也不能使用该存储PL/SQL函数来说明列的默认值，其原因是这几种操作都要求不变定义。

表5-3 函数的纯层

纯 层	含 义	说 明
WNDS	不能写数据库状态	函数不能修改任何数据库表（使用 DML 语句）
RNDS	不能读数据库状态	函数不能读数据库表（使用 SELECT 语句）
WNPS	不能写包状态	函数不能修改任何包变量（包变量不能出现在赋值语句的左边以及 FETCH 语句中）
RNPS	不能读包状态	函数不能使用任何包变量（包变量不能出现在赋值语句的右边或作为过程的一部分，以及 SQL 表方式中）

除了上述的限制外，用户定义的函数还必须满足下面的要求才能实现从 SQL 语句中的调用。除了用户定义的函数外，所有内置函数也必须遵循下列要求。

- 函数必须要存储在数据库中，其形式可以是独立的函数，或作为包的一部分。除此之外，函数之间不能具有本地的关系。
- 函数只能使用 IN 参数，不能使用 IN OUT 或 OUT 参数。
- 形参只能使用数据库类型，不能使用 PL/SQL 类型，如 BOOLEAN，或 RECORD 类型。数据库类型可以是 NUMBER，CHAR，VARCHAR2，ROWID，LONG，RAW，LONG RAW，以及 DATE 和 Oracle8i 引入的新数据类型。
- 函数的返回类型也必须是数据库类型。
- 函数不能使用 COMMIT 或 ROLLBACK 结束当前的事务，或返回到函数运行前的断点（Savepoint）。
- 函数也不能提交任何 ALTER SESSION 或 ALTER SYSTEM 命令。

作为函数调用的例子，下面的函数 FullName 以学生 ID 号为输入并返回学生的全名。

节选自在线代码 FullName.sql

```
CREATE OR REPLACE FUNCTION FullName (  
    p_StudentID students.ID%TYPE)  
    RETURN VARCHAR2 IS  
  
    v_Result VARCHAR2(100);  
BEGIN  
    SELECT first_name || ' ' || last_name  
        INTO v_Result  
        FROM students  
        WHERE ID = p_StudentID;  
  
    RETURN v_Result;  
END FullName;
```

函数 FullName 满足了上述所有的要求，因此我们可以从 SQL 语句中调用该函数，下面是调用该函数的 SQL \*Plus 会话：

节选自在线代码 FullName.sql

```
SQL> SELECT ID, FullName(ID) "Full Name"  
2      FROM students;
```

```

ID Full Name
-----
10000 Scott Smith
10001 Margaret Mason
10002 Joanne Junebug
10003 Manish Murgratroid
10004 Patrick Poll
10005 Timothy Taller
10006 Barbara Blues
10007 David Dinsmore
10008 Ester Elegant
10009 Rose Riznit
10010 Rita Razmataz
10011 Shay Shariatpanahy
12 rows selected.
SQL> INSERT INTO temp_table (char_col)
      2      VALUES (FullName(10010));
1 row created.

```

### RESTRICT\_REFERENCES

PL/SQL引擎可以确认独立函数的纯层。当从SQL语句中调用函数时，PL/SQL对函数的纯层进行检查。如果该函数没有满足限制条件，PL/SQL就返回一个错误。对于打包的函数，需要使用编译标识RESTRICT\_REFERENCES（Oracle8i之前的版本）。该编译标识说明了给定函数的纯层。其语法如下：

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

其中，subprogram\_or\_package\_name是包的名称，或打包的子程序的名称。（Oracle8以上的版本可以使用关键字DEFAULT。）由于WNDS是出现在SQL语句中所有函数必须指定的参数，因此上述编译标识中也必须使用该参数。（Oracle8i对该限制有所放松。）该语句中的纯层说明的顺序是任意的。该编译标识与函数说明都应在包头中出现。例如，下面的包 StudentOps两次使用了RESTRICT\_REFERENCES：

节选自在线代码StudentOps.sql

```

CREATE OR REPLACE PACKAGE StudentOps AS
  FUNCTION FullName(p_StudentID IN students.ID%TYPE)
    RETURN VARCHAR2;
  PRAGMA RESTRICT_REFERENCES(FullName, WNDS, WNPS, RNPS);

  /* Returns the number of History majors. */
  FUNCTION NumHistoryMajors
    RETURN NUMBER;
  PRAGMA RESTRICT_REFERENCES(NumHistoryMajors, WNDS);
END StudentOps;

```

```

CREATE OR REPLACE PACKAGE BODY StudentOps AS
  -- Packaged variable to hold the number of history majors.
  v_NumHist NUMBER;

```

```
FUNCTION FullName(p_StudentID IN students.ID%TYPE)
RETURN VARCHAR2 IS
    v_Result VARCHAR2(100);
BEGIN
    SELECT first_name || ' ' || last_name
        INTO v_Result
        FROM students
        WHERE ID = p_StudentID;

    RETURN v_Result;
END FullName;

FUNCTION NumHistoryMajors RETURN NUMBER IS
    v_Result NUMBER;
BEGIN
    IF v_NumHist IS NULL THEN
        /* Determine the answer. */
        SELECT COUNT(*)
            INTO v_Result
            FROM students
            WHERE major = 'History';
        /* And save it for future use. */
        v_NumHist := v_Result;
    ELSE
        v_Result := v_NumHist;
    END IF;

    RETURN v_Result;
END NumHistoryMajors;
END StudentOps;
```

注意 在Oracle8i中，不再需要使用该编译标识。PL/SQL引擎可以在运行时根据需要校验所有函数的纯层。有关详细信息，请参阅 5.3.3节内容。

**使用RESTRICT\_REFERENCES的合理性** 为什么打包的函数要使用这个编译标识参数，而对独立的函数却没有这种限制呢？该问题的答案与包头和包体的关系有关。我们以前讲过调用打包函数的PL/SQL块只与该包头有关，而与包体无关。进一步说，当我们创建调用块时，其包体甚至可以没有。其结果是，PL/SQL编译器需要这个编译标识参数来确认打包函数的纯层，以及验证该包体是否在其调用块中正确使用。此后，不管在什么时候包体被修改（或首次创建），该函数的代码都要按编译标识进行检查。该标识只在在编译期间检查。

严格地讲，PL/SQL引擎可以在运行时检验函数的纯层，就象 Oracle8i之前的版本对独立函数进行运行时验证那样。然而，使用编译标识就可以通知 PL/SQL引擎不在运行时进行纯层检查，这样做的好处是可以提高运行效率。同时，这样做也确保了给定的子程序不会因为从 SQL语句中调用过程而失败。

Oracle 8 及  
更高版本

**DEFAULT关键字** 如果没有使用编译标识 RESTRICT\_REFERENCES 与给定的打包函数相关联的话, 该函数就没有任何纯层可言。然而, 如果使用 Oracle8 或更高的版本, 我们就可以更改包的默认纯层。关键字 DEFAULT 可用来代替编译标识中的子程序名:

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

修改默认纯层后, 包中所有后续的子程序都必须与说明的纯层一致。例如, 请看下面的包 DefaultPragma:

节选自在线代码 DefaultPragma .sql

```
CREATE OR REPLACE PACKAGE DefaultPragma AS
    FUNCTION f1 RETURN NUMBER;
    PRAGMA RESTRICT_REFERENCES(f1, RNDS, RNPS);

    PRAGMA RESTRICT_REFERENCES(DEFAULT, WNDS, WNPS, RNDS, RNPS);
    FUNCTION f2 RETURN NUMBER;
    FUNCTION f3 RETURN NUMBER;
END DefaultPragma;

CREATE OR REPLACE PACKAGE BODY DefaultPragma AS
    FUNCTION f1 RETURN NUMBER IS
    BEGIN
        INSERT INTO temp_table (num_col, char_col)
            VALUES (1, 'f1!');
        RETURN 1;
    END f1;

    FUNCTION f2 RETURN NUMBER IS
    BEGIN
        RETURN 2;
    END f2;

    -- This function violates the default pragma.
    FUNCTION f3 RETURN NUMBER IS
    BEGIN
        INSERT INTO temp_table (num_col, char_col)
            VALUES (1, 'f3!');
        RETURN 3;
    END f3;
END DefaultPragma;
```

该默认的编译标识 (声明了所有的纯层) 将应用于函数 f2 和 f3。由于 f3 执行了插入表 temp\_table 的操作, 所以违反了编译标识声明, 编译该包时将出现下列错误:

```
PL/SQL: Compilation unit analysis terminated
PLS-00452: Subprogram 'F3' violates its associated pragma
```

**初始化部分** 包初始化部分的代码也可以带有纯层。第一次调用包中的任何函数将导致包初始化部分的启动运行。因此, 打包函数就具有与包括该包的初始话部分相同的纯层。包的纯层也是用 RESTRICT\_REFERENCES 实现的, 但要用包名称代替函数名称。下面是包初始化的语法:

```
CREATE OR REPLACE PACKAGE StudentOps AS
    PRAGMA RESTRICT_REFERENCES (StudentOps, WNDS);
...
END StudentOps;
```

**重载函数** RESTRICT\_REFERENCES可以出现在包说明部分中函数说明后的任何位置。然而，该说明只能对一个函数定义有效。对于重载函数，该编译标识可以对前一个编译标识后最近的函数定义有效。在下面的例子中，每个编译标识都对其前一个编译标识后的 TestFunc的版本有效。

节选自在线代码Overload .sql

```
CREATE OR REPLACE PACKAGE Overload AS
    FUNCTION TestFunc(p_Parameter1 IN NUMBER)
        RETURN VARCHAR2;
    PRAGMA RESTRICT_REFERENCES(TestFunc, WNDS, RNDS, WNPS, RNPS);

    FUNCTION TestFunc(p_ParameterA IN VARCHAR2,
        p_ParameterB IN DATE)
        RETURN VARCHAR2;
    PRAGMA RESTRICT_REFERENCES(TestFunc, WNDS, RNDS, WNPS, RNPS);
END Overload;

CREATE OR REPLACE PACKAGE BODY Overload AS
    FUNCTION TestFunc(p_Parameter1 IN NUMBER)
        RETURN VARCHAR2 IS
    BEGIN
        RETURN 'Version 1';
    END TestFunc;

    FUNCTION TestFunc(p_ParameterA IN VARCHAR2,
        p_ParameterB IN DATE)
        RETURN VARCHAR2 IS
    BEGIN
        RETURN 'Version 2';
    END TestFunc;
END Overload;
```

下面的SQL \*Plus会话演示了上面两个重载函数都可以从SQL中调用：

节选自在线代码Overload .sql

```
SQL> SELECT Overload.TestFunc(1) FROM dual;
OVERLOAD.TESTFUNC(1)
-----
Version 1

SQL> SELECT Overload.TestFunc('abc', SYSDATE) FROM dual;
OVERLOAD.TESTFUNC('ABC',SYSDATE)
-----
Version 2
```

**提示** 作者个人倾向于在每个函数的后面都加上 RESTRICT\_REFERENCES，这样可以清楚地表明该参数作用的函数。

**内置包** PL/SQL提供的内置包中的过程一般都不如 PL/SQL2.3版的过程纯。这些过程包括 DBMS\_OUTPUT,DBMS\_PIPE,DBMS\_ALERT,DBMS\_SQL,和 UTL\_FILE。然而，稍后的版本中在这些包中加入了必要的编译标识 PRAGMA。表5-3介绍了在常用包中加入编译标识 PRAGMA 的时间。由于编译标识没有必要从 Oracle8i开始使用，所以所有满足限制条件的内置包函数都可以向Oracle8i那样用在SQL语句中。如果在 Oracle8i中调用了内置包函数并且该函数没有满足限制条件，则该调用将在运行时发生错误。

**注意** 编译标识已经通过RDBMS修补程序加入到上述某些包中，因此某些比表 5-4列出的版本还要早的 PL/SQL也可以支持编译标识功能。可以通过检查包头的源代码来验证某个 PL/SQL 版本的纯层。（通常，这些文件存放在根 \$ORACLE\_HOME下的 rdbms/admin目录中）。

### 5.3.2 默认参数

从过程语句中调用一个函数时，如果该函数有形参的话，我们可以使用其默认值。然而，如果我们从 SQL语句调用函数时，则所有的参数都必须说明。除此之外，还要使用位置符号（Positinal Notation），而不能使用命名符号（Name Notation）。下面的对函数FullName的调用是非法的：

```
SELECT FullName (p_StudentID = > 10000) FROM dual;
```

表5-4 内置包的编译标识RESTRICT\_REFERENCES

包	加入编译标识的版本
DBMS_ALERT	不存在—REGISTER包括一个COMMIT命令
DBMS_JOB	不存在—作业运行在分离的进程中，因此不能从 SQL中调用
DBMS_OUTPUT	7.3.3
DBMS_PIPE	7.3.3
DBMS_SQL	不存在—EXECUTE和PARSE可以用来执行DDL语句，该语句将引发隐式地 COMMIT命令
STANDARD	7.3.3(包括过程RAISE_APPLICATION_ERROR)
UTL_FILE	8.0.6
UTL_HTTP	7.3.3

### 5.3.3 从Oracle8i的SQL语句中调用函数

正如我们在前几节看到的，编译标识 RESTRICT\_REFERENCES强化了编译时的纯层处理。对于Oracle8i之前的版本，打包函数需要设置编译标识来实现从 SQL语句中的函数调用。然而，从 Oracle8i开始放宽了这种限制，如果没有设置编译标识的话，数据库将在运行时验证纯层。

Oracle8i的这种新的规则对使用外部例程非常有利（使用 C或Java语言编制的外部例程）。在这种情况下，由于PL/SQL编译器并不对这些函数进行实际编译，所以 PL/SQL编译器也就无法实

施纯层检查。（请看本书第10章有关外部例程的介绍）因此对这类外部例程的纯层检查就只能在运行时进行。

对外部函数的纯层检查只在 PL/SQL 运行时发现从 SQL 语句中对该函数进行了调用时才实施。并且，如果在该函数中有编译标识的话，纯层检查将不执行。其结果是，使用纯层检查可以节省运行时间并且也可用来标识函数的状态。

例如，假设我们把下面函数 StudentOps 中的编译标识去掉：

节选自在线代码 StudentOps2.sql

```
CREATE OR REPLACE PACKAGE StudentOps AS
    FUNCTION FullName(p_StudentID IN students.ID%TYPE)
        RETURN VARCHAR2;

    /* Returns the number of History majors. */
    FUNCTION NumHistoryMajors
        RETURN NUMBER;
END StudentOps;
```

重新编译该函数后，如下面所示的 SQL \*Plus 会话，仍然可以从 SQL 语句中对函数进行调用。

```
SQL> SELECT StudentOps.FullName(ID)
      2     FROM students
      3     WHERE major = 'History';
STUDENTOPS.FULLNAME(ID)
-----
Margaret Mason
Patrick Poll
Timothy Taller
SQL> INSERT INTO temp_table (num_col)
      2     VALUES (StudentOps.NumHistoryMajors);
1 row created.
SQL> SELECT * FROM temp_table;
      NUM_COL CHAR_COL
-----
          3
```

如果企图从 SQL 语句中调用非法函数的话，Oracle8i 将发布一条 ‘ORA-14551: 不能在查询中执行 DML 操作’ 的错误信息。请看下面程序中的函数 InsertTemp:

节选自在线代码 InsertTemp.sql

```
CREATE OR REPLACE FUNCTION InsertTemp(
    p_Num IN temp_table.num_col%TYPE,
    p_Char IN temp_table.char_col%type)
    RETURN NUMBER AS
BEGIN
    INSERT INTO temp_table (num_col, char_col)
        VALUES (p_Num, p_Char);
    RETURN 0;
END InsertTemp;
```



如果我们从SELECT语句中调用该函数的话，下面是输出的调用结果：

节选自在线代码InsertTemp.sql

```
SQL> SELECT InsertTemp(1, 'Hello')
      2      FROM dual;
SELECT InsertTemp(1, 'Hello')
      *
ERROR at line 1:
ORA-14551: cannot perform a DML operation inside a query
ORA-06512: at "EXAMPLE.INSERTTEMP", line 6
ORA-06512: at line 1
```

### 1. TRUST关键字

尽管在Oracle8i中可以不再使用编译标识 RESTRICT\_REFERENCES（对外部函数已经不能使用该标识）但在Oracle8i之前的编制的代码仍可以使用该参数。我们在上一节中提到过，使用编译标识的好处是可以提高程序的运行效率。因此，也许存在着这种情况，即程序从一个没有使用编译标识的函数中调用一个声明了纯层的函数。为了实现这种程序设计要求，Oracle8i提供了一个可在编译标识中使用的新关键字，用来代替或辅助纯层参数，它就是 TRUST。

如果使用了TRUST的话，则在编译标识中列出的限制将失效。更确切的讲，编译程序视这些限制为真。这就允许我们来编制不再使用编译标识 RESTRICT\_REFERENCES的新代码，并实现从声明为纯的函数中调用这些新的函数。例如，请看下面的包：

节选自在线代码TrustPkg.sql

```
CREATE OR REPLACE PACKAGE TrustPkg AS
  FUNCTION ToUpper(p_a VARCHAR2) RETURN VARCHAR2 IS
    LANGUAGE JAVA
    NAME 'Test.Uppercase(char[]) return char[]';
    PRAGMA RESTRICT_REFERENCES(ToUpper, WNDS, TRUST);

  PROCEDURE Demo(p_in IN VARCHAR2, p_out OUT VARCHAR2);
  PRAGMA RESTRICT_REFERENCES(Demo, WNDS);
END TrustPkg;

CREATE OR REPLACE PACKAGE BODY TrustPkg AS
  PROCEDURE Demo(p_in IN VARCHAR2, p_out OUT VARCHAR2) IS
  BEGIN
    p_out := ToUpper(p_in);
  END Demo;
END TrustPkg;
```

正在被该DML语句修改的该包中TrustPkg.ToUpper是一个外部例程，它是一个用Java编制的函数体，其功能是将输入参数转换为大写返回。（我们将在第10章讨论参数的转换方法）。由于该函数体不在PL/SQL中，关键字TRUST就要与编译标识一起使用。这样一来，由于编译器承认函数ToUpper具有了WNDS纯层，所以我们可以从Demo中调用该函数ToUpper。

### 2. 从DML语句中调用函数

在Oracle8i之前，从DML语句中调用的函数不能更新数据库（也就是说，该函数必须声明为

RNDS纯层)。然而，在Oracle8i下，该限制有所放宽。现在，从DML语句中调用的函数即不能读正在被该DML语句修改的数据库表也不能修改正在被该DML语句修改的数据库表，但该函数可以更新其他的表。例如，请看下面的函数UpdateTemp：

节选自在线代码DMLUpdate .sql

```
CREATE OR REPLACE FUNCTION UpdateTemp(p_ID IN students.ID%TYPE)
RETURN students.ID%TYPE AS
BEGIN
    INSERT INTO temp_table (num_col, char_col)
    VALUES(p_ID, 'Updated!');
    RETURN p_ID;
END UpdateTemp;
```

在Oracle8i之前，执行下面的更新语句将导致错误：

节选自在线代码DMLUpdate .sql

```
UPDATE students
SET major = 'Nutrition'
WHERE UpdateTemp(ID) = ID;
```

而在Oracle8i下，上面的更新语句只对表temp\_table进行了修改，而没有更新表students。

注意 从并行DML语句调用的函数不能修改数据库以及当前处于修改状态的表。

## 5.4 包的辅助功能

我们将在本节讨论包的某些辅助功能，其中包括在共享缓冲区中锁定包以及讨论包体长度的问题。对于Oracle8i,我们将讨论优化选择项DETERMINISTIC和PARALLEL\_ENABLE。

### 5.4.1 共享缓冲区锁定

共享缓冲区是存放已编译子程序中间代码的SGA的一部分。当第一次调用存储子程序时，该子程序的中间代码将从磁盘转载到共享缓冲区中；一旦没有其他程序引用该子程序时，其中间代码将被从共享缓冲区中清除；共享缓冲区中的对象的清除是按LRU算法（最近使用最少的对象先清除）执行的。

包DBMS\_SHARED\_POOL允许程序员把一个对象锁定在共享缓冲区中。当该对象被锁定后，除非由程序申请对其清除，否则不管共享缓冲区是否已满，或是否有程序访问该对象，该对象将常驻在共享缓冲区中。这种处理方法有利于提高程序的运行效率，因为从系统的磁盘重新载入对象要进行大量读写操作。锁定对象还有助于将共享缓冲区的碎片效应减至最小。包DBMS\_SHARED\_POOL有三个过程，它们分别是：DBMS\_SHRAED\_POOL.KEEP,DBMS\_SHARED\_POOL.UNKEEP,以及DBMS\_SHARED\_POOL.SIZES。

#### 1. 过程KEEP

过程DBMS\_SHRAED\_POOL.KEEP用来在缓冲区中锁定对象。包、触发器（Oracle7.3及更高版本）、序列和SQL语句都可以被锁定。需要注意的是，独立过程和函数不能锁定。过程KEEP的语法如下：

```
PROCEDURE KEEP (name VARCHAR2,flag CHAR DEFAULT'P');
```

以上参数在下面的表 5-5 中说明。一旦对象被锁定，除非在数据库关闭或使用了过程 DBMS\_SHARED\_POOL.UNKEEP 外，该对象将永不退出共享缓冲区。注意，DBMS\_SHARED\_POOL.KEEP 并不立即将包载入缓冲区中；而是对在其后载入的包实施锁定。

表5-5 过程KEEP的参数说明

参 数	类 型	说 明
name	VARCHAR2	对象的名称。该参数可以是包名或与 SQL 语句关联的标识符。SQL 标识符是由视图 \$sqlarea 中的字段 hash_value 和 address 的连接组成（默认情况下，只能通过 SYS 选择）并由过程 SIZES 返回
flag	CHAR	决定对象的类型。如果该参数是 ' P '（默认值），则参数 name 就必须与包名匹配。如果该参数是 ' C '（游标）的话，则 name 就要带有 SQL 语句的文本。如果该参数是 ' S '，则 name 就是序列，如果它是 ' R '，则 name 就是触发器

## 2. 过程UNKEEP

过程 UNKEEP 是从共享缓冲区中删除锁定对象的唯一方法。锁定的对象不会自动退出共享缓冲区。UNKEEP 的语法是：

```
PROCEDURE UNKEEP(name VARCHAR2,flag CHAR DEFAULT 'P');
```

以上参数的含义与过程 KEEP 的参数相同。如果说明的对象没有在共享缓冲区中的话，将会引发错误。

## 3. 过程SIZES

该过程将把共享缓冲区的内容输出到屏幕。其语法如下：

```
PROCEDURE SIZES(minsize NUMBER);
```

执行该命令后，长度大于指定的 minsize 的对象将显示在屏幕上。过程 SIZES 使用包 DBMS\_OUTPUT 来返回数据，因此，在调用该过程前，一定要在 SQL \*Plus 或 SQL 服务管理器中使用 SET SEVEROUPUT ON。

### 5.4.2 包体长度的限制

PL/SQL 编译器内部的限制条件可能会影响 PL/SQL 的长度。一般来说，包体是包的最大部分，因此也就容易与 PL/SQL 的内部限制冲突。当包体长度达到了编译器的内部默认长度限制时，编译器将显示错误信息 ' PLS-123:程序太大，无法编译 '。编译器对包体长度的限制如下：

- Diana 树中的节点数。PL/SQL 编译器构造一种叫做 Diana 的内部树，该树反映了块的结构。Diana 树是在第一遍编译中生成的。在 Oracle8i 之前的版本中，Diana 节点的最大数目是 32K，Oracle8i 将该包和类型体的限制扩充到了 64 兆字节的容量。该容量是大多数块首次达到的极限。
- 编译器生成的临时中间变量。该类变量的最大数是 21K 字节。
- 入口点的数量。一个包体最多可以有 32K 个入口点，入口点可以是过程或函数。
- 字符串的数量。PL/SQL 对字符串的限制单位是  $2^{32}$ 。

在Oracle8i之前的版本中，最常达到的限制是 Diana节点数，因此，后来的版本对该限制进行了扩充。总的来说，节点数是与源程序行的数量成正比，因此，减少包体长度的最好方法是减少代码的行数。通常，我们可以把某些子程序从包体中去掉，转而放置在独立的包中。

提示 为了易于阅读和载入方便，应尽量将包的长度减少。

### 5.4.3 优化参数

对Oracle8i来说，还有两个可以用在函数声明中的辅助关键字——DETERMINISTIC 和 PARALLEL\_ENABLE。如果使用这两个关键字的话，PL/SQL编译优化器将会对调用PL/SQL函数进行优化。该关键字要放在函数的返回类型后以及语句IS或AS之前。其语法如下：

```
CREATE [OR REPLACE] FUNCTION function_name
[ parameter_list]
RETURN return_type
[ DETERMINISTIC]
[ PARALLEL_ENABLE]
IS | AS
function_body;
```

如果同时指定了这两个关键字，其出现顺序可以是任意的。关键字 DETERMINISTIC和 PARALLEL\_ENABLE可用于独立的函数，打包函数，或对象类型方法（请看本书第 13章有关对象类型的介绍）。如果函数是方法或打包函数的话，则该关键字就要在包头或类型头中使用，而不能出现在包体或类型体中。

我们在下面两节中将会看到这些关键字的使用方法。

#### 1. 关键字DETERMINISTIC

如果一个函数对于给定的相同输入值总是返回同一结果并不会引起任何副作用的话（如对打包变量进行修改），则该函数就被称为是确定函数。由于确定函数在其参数保持不变时可以免去多次调用，所以该类函数可广泛使用。例如，请看下面的函数 StudentStatus：

节选自在线代码determ .sql

```
CREATE OR REPLACE FUNCTION StudentStatus(
p_NumCredits IN NUMBER)
RETURN VARCHAR2 AS
BEGIN
IF p_NumCredits = 0 THEN
RETURN 'Inactive';
ELSIF p_NumCredits <= 12 THEN
RETURN 'Part Time';
ELSE
RETURN 'Full Time';
END IF;
END StudentStatus;
```

由于该函数对于给定的相同输入总是返回同一个结果并不修改任何包变量，所以该函数是一个确定函数。正是该函数的这种特点，我们可以使用关键字 DETERMINISTIC通知编译器该函

数是确定函数：

```

节选自在线代码determ .sql
CREATE OR REPLACE FUNCTION StudentStatus(p_NumCredits IN NUMBER)
RETURN VARCHAR2
DETERMINISTIC AS
BEGIN
    IF p_NumCredits = 0 THEN
        RETURN 'Inactive';
    ELSIF p_NumCredits <= 12 THEN
        RETURN 'Part Time';
    ELSE
        RETURN 'Full Time';
    END IF;
END StudentStatus;

```

PL/SQL编译器将不验证该函数是否是真的确定函数，它只是对该函数做确定函数的标记。确定函数可用于下列场合：

- 用在基于函数的索引上的任何函数必须是确定函数。非确定函数可用在 SQL 语句的 WHERE 子句中，但程序员不能基于该函数创建索引。
- 如果实际的视图（Materialized view）被标记为 ENABLE\_QUERY REWRITE，则该视图中使用的任何函数都必须是确定函数。
- 声明为 REFRESH FAST 的快照或实际视图中的函数也应该是确定函数。该声明的使用并不是必须的（在确定函数使用之前参数 REFRESH FAST 就已经被使用），它是一种推荐。
- 在 SQL 语句中的 WHERE，ORDER BY 或 GROUP BY 子句中使用的函数也是确定函数。这也适用于 SQL 类型的 ORDER 方法或 MAP 方法。总的来说，用于确定结果集内容的任何函数都必须是确定的。在这里需要再次说明的是，Oracle 语言只是推荐使用上述规则，并不是强制执行标准。

**基于函数索引的确定函数** 在 Oracle8i 中，可以根据调用 PL/SQL 存储函数的表达式来创建索引，这类索引叫做基于函数的索引。这就使我们在从 SQL 语句中调用函数时可以使用索引功能。例如，请看下面的例子：

```

节选自在线代码determ .sql
SELECT id
FROM students
WHERE SUBSTR(StudentStatus(current_credits), 1, 20) =
    'Part Time';

```

如果看一下 SELECT 语句的执行情况，可以看到：

```

Rows          Row Source Operation
-----
      12      TABLE ACCESS FULL STUDENTS
Rows          Execution Plan
-----

```

```
0      SELECT STATEMENT GOAL: CHOOSE
12     TABLE ACCESS (FULL) OF 'STUDENTS'
```

在上面的程序中，我们正在进行全表扫描。为了使上面的查询操作效率更高，我们可以创建使用函数值的索引。这样一来，该查询就可以使用索引了：

节选自在线代码determ.sql

```
CREATE INDEX students_index ON students
(SUBSTR(StudentStatus(current_credits), 1, 20))
COMPUTE STATISTICS;
```

查询的执行情况现在为：

Rows	Row	Source	Operation
------	-----	--------	-----------

	12	TABLE ACCESS BY INDEX ROWID STUDENTS	
	13	INDEX RANGE SCAN (object id 13271)	

Rows	Execution Plan
------	----------------

0	SELECT STATEMENT GOAL: FIRST_ROWS
12	TABLE ACCESS (FULL) OF 'STUDENTS'

注意 为了创建基于函数的索引，索引的拥有者必须具有 QUERY REWRITE的特权。系统特权GLOBAL REWEITE为在其他用户模式下创建基于函数的索引提供了保证。除此之外，在优化器使用索引之前，还必须满足其他一些要求。有关信息请看 Oracle文档资料。

## 2. 关键字PARALLEL\_ENABLE

在某些情况下，程序员可以使用 Oracle的并行处理功能来并行执行 SQL语句。如果有多个 SQL语句调用一个PL/SQL函数，则该函数将被独立的进程所调用，每个进程都运行一个该函数的副本来对表列的子集进行操作。

如果该函数引用了一个包变量的话，就将引发错误。该函数的副本将对其打包变量进行初始化，就像该函数是刚注册的函数。因此，该函数的副本将看不到函数早期对这些包变量的修改。其结果是，读或修改包变量的任何函数都不能同时运行。如果该语句是一个 DML语句的话（不是查询语句），则该函数就可以既不读也不修改数据库状态。

对于Oracle8i之前的版本，上述限制是唯一由编译标识 RESTRICT\_REFERENCES实施的。然而，在Oracle8i下，程序员可以使用子句 PARALLEL\_ENABLE标识来通知优化器该函数的并行属性。这就使程序员可以更灵活地控制函数运行在并行状态。

## 3. 非PL/SQL函数的优化

Oracle8i允许程序员创建外部例程，这些用 C语言或Java语言编制的外部例程是可以直接从 PL/SQL调用的函数。DETERMINISTIC和（或）PARALLEL\_EANBLE子句也可以用在PL/SQL调用外部例程的说明中。如果使用了这些关键字，则上面所述的强制规则就将实施。当然，由于 PL/SQL编译器不能运行在外部例程上，所以应由程序员来验证外部例程是否满足 DETERMINISTIC和PARALLEL\_EANBLE实现的要求。有关进一步信息，请看本书的第10章内容。

## 5.5 小结

我们在本章讨论了命名 PL/SQL 块的三种类型，过程，函数和包。我们讨论的内容包括本地和存储子程序的区别以及存储子程序之间的相关工作原理。除此之外，我们还研究了如何从 SQL 语句中调用存储子程序。在本章的最后，我们介绍了包的几种辅助功能。在下一章中，我们将学习命名 PL/SQL 块的第四种类型——数据库触发器。