

第一部分 PL/SQL介绍及开发环境

第1章 PL/SQL介绍

PL/SQL是一种高级数据库程序设计语言，该语言专门用于在各种环境下对 Oracle数据库进行访问。由于该语言集成于数据库服务器中，所以 PL/SQL代码可以对数据进行快速高效的处理。除此之外，在 Oracle数据库的某些客户端工具中，使用 PL/SQL语言也是该语言的一个特点。本章的主要内容是讨论引入 PL/SQL语言的必要性和该语言的主要特点，以及了解 PL/SQL语言的重要性和数据库版本问题。还要介绍一些贯穿全书的更详细的高级概念，并在本章的最后就我们在本书案例中使用的数据库表的若干约定做一说明。

1.1 为什么要引入PL/SQL语言

Oracle数据库是一种关系型数据库。通常我们把用于访问这种关系型数据库的程序设计语言叫做结构化查询语言，即 SQL语言。SQL是一种灵活高效的查询语言，其主要功能是对关系数据库中的数据进行操作和处理。例如，下面的 SQL语句可以从数据库中将学习营养专业的全部学生一次删除：

```
DELETE FROM students  
WHERE major = 'Nutrition';
```

(本章的最后一节将对本书中使用的包括 students表在内的各种数据库表进行说明。)

SQL是先进的第四代程序设计语言，使用这种语言只需对要完成的任务进行描述，而不必指定实现任务的具体方法。以上面例子中的 DELETE语句为例，我们并不知道 SQL语言是如何找到学习营养专业的学生的。虽然按一般语言的做法推测，数据库服务器要按某种顺序逐个访问数据库表中的所有学生记录以决定删除满足条件的学生记录。但实际上，我们无法知道这些删除操作的细节。

第三代程序设计语言如 C语言和COBOL语言等是面向过程的语言。用第三代语言（3GL）编制的程序是一步一步地实现程序功能的。例如，我们可以用下面的程序段来实现上述的删除操作：

```
LOOP over each student record  
  IF this record has major = 'Nutrition' THEN  
    DELETE this record;  
  END IF;  
END LOOP;
```

面向对象的程序设计语言如 C++或Java也属于第三代程序设计语言。虽然这类语言采用了面向对象的程序结构，但程序中算法的实现还是要用各种语句逐步指定。

各种语言都有其自身的优缺点。相对于第三代程序设计语言来说，SQL一类的第四代程序语言使用起来非常简单，语言中语句的种类也比较少。但这类语言将用户与实际的数据结构和算法隔离开来，对数据的具体处理完全由该类语言的运行时系统实现。而在某些情况下，第三代语言使用的过程结构在表达某些程序过程来说是非常有用的。这也就是引入PL/SQL语言的原因，即PL/SQL语言将第四代语言的强大功能和灵活性与第三代语言的过程结构的优势融为一体。

PL/SQL代表面向过程化的语言与SQL语言的结合。我们可以从该语言的名称中看出，PL/SQL是在SQL语言中扩充了面向过程语言中使用的程序结构，如：

- 变量和类型（即可以预定义也可以由用户定义）
- 控制语句（如IF-THEN-ELSE）和循环
- 过程和函数
- 对象类型和方法（PL/SQL8.0版本以上）

PL/SQL语言实现了将过程结构与Oracle SQL的无缝集成，从而为用户提供了一种功能强大的结构化程序设计语言。例如，我们假设要修改一个学生的专业。如果没有该学生的记录的话，我们就为该学生创建一个新的记录。用PL/SQL编制的程序代码可以实现我们的要求，如下所示：

```
节选自在线代码3gl_4gL.sql
DECLARE
    /* Declare variables that will be used in SQL statements */
    v_NewMajor VARCHAR2(10) := 'History';
    v_FirstName VARCHAR2(10) := 'Scott';
    v_LastName VARCHAR2(10) := 'Urman';
BEGIN
    /* Update the students table. */
    UPDATE students
        SET major = v_NewMajor
        WHERE first_name = v_FirstName
        AND last_name = v_LastName;
    /* Check to see if the record was found. If not, then we need
       to insert this record. */
    IF SQL%NOTFOUND THEN
        INSERT INTO students (ID, first_name, last_name, major)
            VALUES(student_sequence.NEXTVAL, v_FirstName, v_LastName,
                   v_NewMajor);
    END IF;
END;
```

上面的例子中有两个不同的SQL语句（UPDATE和INSERT），这两个语句是第四代程序结构，同时该段程序中还使用了第三代语言的结构（变量声明和IF条件语句）。

注意 为了运行上面的程序例子，先要创建程序中引用的数据库对象（即表students和序列student_sequence）。可以使用本书CD-ROM中提供的脚本文件relTable.sql来实现上述工作。有关创建上述对象的进一步信息，请参见1.3.3节的内容。

PL/SQL语言在将SQL语言的灵活性及功能与第三代语言的可配置能力相结合方面是独一无二的。

二的。该语言集成了面向过程语言的过程结构和强大的数据库操作，为设计复杂的数据库应用提供了功能强大、健壮可靠的程序设计语言。

1.1.1 PL/SQL与网络传输

目前的大多数数据库应用一般都采用客户/服务器或三层模式。在客户/服务器模式下，驻留在客户机上的应用程序使用 SQL语句向数据库服务器发送服务请求。通常，发送这种请求将导致过多的网络传输，如图 1-1 左半部分的框图所示，每个 SQL语句将引起一次网络传输。现在将该图的左面与右面相比较，我们可以发现使用 PL/SQL语言可以将几个 SQL语句合并为一个PL/SQL块发送给服务器，从而减少网络通信流量并提高应用程序的执行速度。

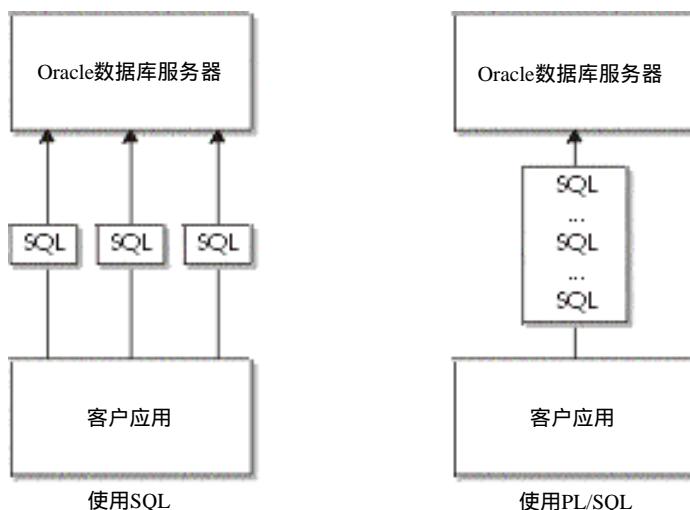


图1-1 在客户/服务器环境下使用SQL和PL/SQL的比较

即使在客户机与服务器都运行在同一台设备上时，使用 PL/SQL 编制的应用程序的性能也会有提高。在这种情况下，虽然不需要进行网络传输，但将 SQL语句打包传送将减少对数据库的访问次数。

PL/SQL语言的打包功能也适用于三层结构应用。在这种模式下，客户机（通常运行在 HTML浏览器下）与应用服务器进行通信，而应用服务器再与数据库交互操作，PL/SQL有利于应用服务器与数据库的通信。有关 PL/SQL的这种应用环境，我们将在本书的第 2 章中讨论。

1.1.2 PL/SQL标准

Oracle数据库支持 ANSI 标准的 SQL语言，即 ANSI X3.135-1992 文档中“数据库语言 SQL”定义的 SQL语言。该标准通常称为 SQL92 标准，只定义了 SQL语言本身。该标准并没有定义 PL/SQL 所提供语言的 3GL 扩充内容。SQL92 指定了三个实现层次：初等级别、中等级别和最高级别。Oracle7 的 7.0 版（包括所有高版本的 Oracle8 和 Oracle8i）实现了由美国国家标准技术研究所认证的 SQL92 标准的初等级别。现在 Oracle 公司正在与美国国家标准协会 ANSI 共同努力以确保 Oracle 数据库和 PL/SQL 语言的最新版本将实现 SQL92 的最高级别。

1.2 PL/SQL的特点

介绍PL/SQL的不同特点与功能的最好方式是通过实际程序案例演示。本章下面几节将描述PL/SQL语言的若干主要特点。

1.2.1 PL/SQL的基本特点

本书的主要内容是介绍PL/SQL语言的高级功能。在下面几节中，我们将介绍该语言的基本特点。如果读者要进一步了解有关信息，请参考PL/SQL用户指南及索引手册，或参考《Oracle8 PL/SQL程序设计》一书。

1. PL/SQL的块结构

PL/SQL程序的基本结构是块。所有的PL/SQL程序都是由块组成的，这些块之间还可以相互嵌套。通常，程序中的每一块都实现一个逻辑操作，从而把不同的任务进行分割，由不同的块来实现。PL/SQL的块结构如下所示：

```
DECLARE
    /* Declarative section - PL/SQL variables, types, cursors,
       and local subprograms go here. */

BEGIN
    /* Executable section - procedural and SQL statements go here.
       This is the main section of the block and the only one
       that is required. */

EXCEPTION
    /* Exception-handling section - error-handling statements go
       here. */

END;
```

在上面演示的块结构中，只有执行部分是必须的，声明部分和异常处理部分都是可选的。块结构中的执行部分至少要有一个可执行语句。PL/SQL块采用的这种分段结构将PL/SQL程序的不同功能各自独立出来。

PL/SQL的这种特点是仿效第三代程序设计语言Ada采用的程序结构。在Ada语言中使用的很多程序结构包括Ada使用的块结构都适用于PL/SQL语言。除此之外，在PL/SQL语言中还可以发现Ada语言使用的异常处理方法、过程和函数声明以及包的定义的语法等特征。

2. 错误处理

PL/SQL块中的异常处理部分是用来响应应用程序运行中遇到的错误。把程序的主体部分与错误处理部分代码相互隔离，这样，程序的结构看起来十分清晰。例如，下面的PL/SQL块演示了将异常发生的时间及将遇到该异常错误的用户名记录在日志表的处理过程。

```
节选自在线代码Error.sql
DECLARE
    v_ErrorCode NUMBER;          -- Code for the error
    v_ErrorMsg VARCHAR2(200);    -- Message text for the error
    v_CurrentUser VARCHAR2(8);   -- Current database user
    v_Information VARCHAR2(100); -- Information about the error
BEGIN
```

```
/* Code that processes some data here */
EXCEPTION
WHEN OTHERS THEN
-- Assign values to the log variables, using built-in
-- functions.
v_ErrorCode := SQLCODE;
v_ErrorMsg := SQLERRM;
v_CurrentUser := USER;
v_Information := 'Error encountered on ' ||
    TO_CHAR(SYSDATE) || ' by database user ' || v_CurrentUser;
-- Insert the log message into log_table.
INSERT INTO log_table (code, message, info)
VALUES (v_ErrorCode, v_ErrorMsg, v_Information);
END;
```

注意 上面的例子和许多其他例程都在本书的联机发布信息中。读者如要了解有关详细信息，请参阅1.3.3节的内容。

3. 变量和类型

信息在数据库与PL/SQL程序之间是通过变量进行传递的。所谓变量就是可以由程序读取或赋值的存储单元。在上面的例子中，v_CurrentUser,v_ErrorCode,和v_Information都是变量。通常，变量是在PL/SQL块的声明部分定义的。

每个变量都有一个特定的类型与其关联。变量的类型定义了变量可以存放的信息类别。如下所示，PL/SQL变量可以与数据库列具有同样的类型：

```
DECLARE
    v_StudentName VARCHAR2(20);
    v_CurrentDate DATE;
    v_NumberCredits NUMBER(3);
```

PL/SQL变量也可以是其他类型：

```
DECLARE
    v_LoopCounter BINARY_INTEGER;
    v_CurrentlyRegistered BOOLEAN;
```

除此之外，PL/SQL还支持用户自定义的数据类型，如记录类型、表类型等。使用用户自定义的数据类型可以让你定制程序中使用的数据类型结构。下面是一个用户自定义的数据类型例子：

```
DECLARE
    TYPE t_StudentRecord IS RECORD (
        FirstName VARCHAR2(10),
        LastName VARCHAR2(10),
        CurrentCredits NUMBER(3)
    );
    v_Student t_StudentRecord;
```

4. 循环结构

PL/SQL支持多种循环结构。所谓循环就是指可以重复执行的同代码段。例如，下面的程序块使用一个简单的循环来把数字 1 ~ 50 插入到表temp_table中：

```
节选自在线代码SimpleLoop.sql
DECLARE
    v_LoopCounter BINARY_INTEGER := 1;
BEGIN
    LOOP
        INSERT INTO temp_table (num_col)
        VALUES (v_LoopCounter);
        v_LoopCounter := v_LoopCounter + 1;
        EXIT WHEN v_LoopCounter > 50;
    END LOOP;
END;
```

PL/SQL中还提供了一种使用FOR的循环结构，这种循环的结构更简单。如下所示，我们可以使用这种FOR循环来实现上面的循环操作：

```
节选自在线代码NumricLoop.sql
BEGIN
    FOR v_LoopCounter IN 1..50 LOOP
        INSERT INTO temp_table (num_col)
        VALUES (v_LoopCounter);
    END LOOP;
END;
```

5. 游标

游标是用来处理使用SELECT语句从数据库中检索到的多行记录的工具。借助于游标的功能，数据库应用程序可以对一组记录逐个进行处理，每次处理一行。例如，下面的程序块可以检索到数据库中所有学生的名和姓：

```
节选自在线代码CursorLoop.sql
DECLARE
    v_FirstName VARCHAR2(20);
    v_LastName VARCHAR2(20);
    -- Cursor declaration. This defines the SQL statement to
    -- return the rows.
    CURSOR c_Students IS
        SELECT first_name, last_name
        FROM students;
BEGIN
    -- Begin cursor processing.
    OPEN c_Students;
    LOOP
        -- Retrieve one row.
        FETCH c_Students INTO v_FirstName, v_LastName;
        -- Exit the loop after all rows have been retrieved.
        EXIT WHEN c_Students%NOTFOUND;
        /* Process data here */
    END LOOP;
    CLOSE c_Students;
END;
```

```
END LOOP;
-- End processing.
CLOSE c_Students;
END;
```

1.2.2 PL/SQL的高级功能

下面将介绍几个有关PL/SQL高级功能的程序例子，这些高级功能将在本书的后几章做详细讲解。PL/SQL的高级功能是在其基本功能上实现的。

1. 过程和函数

PL/SQL中的过程和函数（通称为子程序）是PL/SQL块的一种特殊类型，这种类型的子程序可以以编译的形式存放在数据库中，并为后续的程序块调用。例如，下面的语句创建了一个叫做PrintStudents的过程，该过程使用包DBMS_OUTPUT将所有学生的姓名以定制的格式显示在屏幕上：

```
节选自在线代码PrintStudents.sql
CREATE OR REPLACE PROCEDURE PrintStudents(
    p_Major IN students.major%TYPE) AS

    CURSOR c_Students IS
        SELECT first_name, last_name
        FROM students
        WHERE major = p_Major;

    BEGIN
        FOR v_StudentRec IN c_Students LOOP
            DBMS_OUTPUT.PUT_LINE(v_StudentRec.first_name || ' ' ||
                v_StudentRec.last_name);
        END LOOP;
    END;
```

一旦创建了该过程并将其存储在数据库中，我们就可以用如下所示的程序块来调用该过程：

```
节选自在线代码PrintStudents.sql
```

```
SQL> BEGIN
  2 PrintStudents('Computer Science');
  3 END;
  4 /
Scott Smith
Joanne Junebug
Shay Shariatpanah
```

注意 使用SET SERVEROUTPUT ON命令可以设置DBMS_OUTPUT的输出功能。有关该命令的详细信息，请参见本书第2章的内容。

2. 包

PL/SQL子程序可以和变量与类型共同组成包。PL/SQL的包由两部分组成，即说明部分和包体。一个包可以带有多个相关的过程。例如，下面的包RoomsPkg就有两个过程，一个是插入新

教室信息的过程，另一个是从表 rooms 中删除一个教室的过程：

节选自在线代码 RoomPkg.sql

```

CREATE OR REPLACE PACKAGE RoomsPkg AS
    PROCEDURE NewRoom(p_Building rooms.building%TYPE,
                      p_RoomNum rooms.room_number%TYPE,
                      p_NumSeats rooms.number_seats%TYPE,
                      p_Description rooms.description%TYPE);

    PROCEDURE DeleteRoom(p_RoomID IN rooms.room_id%TYPE);
END RoomsPkg;

CREATE OR REPLACE PACKAGE BODY RoomsPkg AS
    PROCEDURE NewRoom(p_Building rooms.building%TYPE,
                      p_RoomNum rooms.room_number%TYPE,
                      p_NumSeats rooms.number_seats%TYPE,
                      p_Description rooms.description%TYPE) IS
        BEGIN
            INSERT INTO rooms
                (room_id, building, room_number, number_seats, description)
            VALUES
                (room_sequence.NEXTVAL, p_Building, p_RoomNum, p_NumSeats,
                 p_Description);
        END NewRoom;
    PROCEDURE DeleteRoom(p_RoomID IN rooms.room_id%TYPE) IS
        BEGIN
            DELETE FROM rooms
            WHERE room_id = p_RoomID;
        END DeleteRoom;
END RoomsPkg;

```

3. 动态SQL

借助于动态 SQL，一个PL/SQL应用程序可以在运行期间构造并执行 SQL语句。使用动态SQL方法有两种，一种是使用PL/SQL2.1版及以上支持的DBMS_SQL包，另一种是使用Oracle8i或更高版本支持的本地动态SQL。下面所示的DropTable就是使用DBMS_SQL实现的过程，其功能是释放指定的工作表：

节选自在线代码 DropTable.sql

```

CREATE OR REPLACE PROCEDURE DropTable(p_Table IN VARCHAR2) AS
    v_SQLString VARCHAR2(100);
    v_Cursor BINARY_INTEGER;
    v_ReturnCode BINARY_INTEGER;
BEGIN
    -- Build the string based on the input parameter.
    v_SQLString := 'DROP TABLE ' || p_Table;

```

-- Open the cursor.

```
v_Cursor := DBMS_SQL.OPEN_CURSOR;

-- Parse and execute the statement.
DBMS_SQLPARSE(v_Cursor, v_SQLString, DBMS_SQL.NATIVE);
v_ReturnCode := DBMS_SQL.EXECUTE(v_Cursor);

-- Close the cursor.
DBMS_SQLCLOSE_CURSOR(v_Cursor);
END DropTable;
```

如果使用Oracle8i或更高的版本，该过程可以使用本地动态SQL重写如下：

节选自在线代码PrintStudents.sql

```
CREATE OR REPLACE PROCEDURE DropTable(p_Table IN VARCHAR2) AS
  v_SQLString VARCHAR2(100);
BEGIN
  -- Build the string based on the input parameter.
  v_SQLString := 'DROP TABLE ' || p_Table;

  EXECUTE IMMEDIATE v_SQLString;
END DropTable;
```

4. 对象类型（Oracle8及以上版本）

Oracle8(包括PL/SQL 8)支持对象类型。Oracle中的对象类型由属性和方法组成并可以存储在数据库表中。下面的例子演示了创建对象类型的方法：

节选自在线代码ch12/objTypes.sql

```
CREATE OR REPLACE TYPE Student AS OBJECT (
  ID          NUMBER(5),
  first_name  VARCHAR2(20),
  last_name   VARCHAR2(20),
  major       VARCHAR2(30),
  current_credits NUMBER(3),

  -- Returns the first and last names, separated by a space.
  MEMBER FUNCTION FormattedName
    RETURN VARCHAR2,
  PRAGMA RESTRICT_REFERENCES(FormattedName, RNDS, WNDS, RNPS, WNPS),

  -- Updates the major to the specified value in p_NewMajor.
  MEMBER PROCEDURE ChangeMajor(p_NewMajor IN VARCHAR2),
  PRAGMA RESTRICT_REFERENCES(ChangeMajor, RNDS, WNDS, RNPS, WNPS),

  -- Updates the current_credits by adding the number of
  -- credits in p_CompletedClass to the current value.
  MEMBER PROCEDURE UpdateCredits(p_CompletedClass IN Class),
  PRAGMA RESTRICT_REFERENCES(UpdateCredits, RNDS, WNDS, RNPS, WNPS),
```

```
-- ORDER function used to sort students.  
ORDER MEMBER FUNCTION CompareStudent(p_Student IN Student)  
    RETURN NUMBER  
);  
  
CREATE OR REPLACE TYPE BODY Student AS  
    MEMBER FUNCTION FormattedName  
        RETURN VARCHAR2 IS  
    BEGIN  
        RETURN first_name || ' ' || last_name;  
    END FormattedName;  
  
    MEMBER PROCEDURE ChangeMajor(p_NewMajor IN VARCHAR2) IS  
    BEGIN  
        major := p_NewMajor;  
    END ChangeMajor;  
  
    MEMBER PROCEDURE UpdateCredits(p_CompletedClass IN Class) IS  
    BEGIN  
        current_credits := current_credits +  
            p_CompletedClass.num_credits;  
    END UpdateCredits;  
    ORDER MEMBER FUNCTION CompareStudent(p_Student IN Student)  
        RETURN NUMBER IS  
    BEGIN  
        -- First compare by last names  
        IF p_Student.last_name = SELF.last_name THEN  
            -- If the last names are the same, then compare first names.  
            IF p_Student.first_name < SELF.first_name THEN  
                RETURN 1;  
            ELSIF p_Student.first_name > SELF.first_name THEN  
                RETURN -1;  
            ELSE  
                RETURN 0;  
            END IF;  
        ELSE  
            IF p_Student.last_name < SELF.last_name THEN  
                RETURN 1;  
            ELSE  
                RETURN -1;  
            END IF;  
        END IF;  
    END CompareStudent;  
END;
```

5. 集合

PL/SQL的集合类似于其他3GL中使用的数组。PL/SQL提供了三种不同的集合类型：按表索

引 (PL/SQL 2.0及更高版本)、嵌套表 (PL/SQL 8.0及更高版本)、数组 (PL/SQL 8.0及更高版本)。下面的程序例子介绍了上述三种集合类型的使用方法：

```
节选自在线代码Collections.sql
DECLARE
  TYPE t_IndexBy IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
  TYPE t_Nested IS TABLE OF NUMBER;
  TYPE t_Varray IS VARRAY(10) OF NUMBER;

  v_IndexBy t_IndexBy;
  v_Nested t_Nested;
  v_Varray t_Varray;
BEGIN
  v_IndexBy(1) := 1;
  v_IndexBy(2) := 2;
  v_Nested := t_Nested(1, 2, 3, 4, 5);
  v_Varray := t_Varray(1, 2);
END;
```

1.2.3 PL/SQL内置包

除了上述PL/SQL语言提供的功能外，Oracle也提供了若干具有特殊功能的内置包。本书自始至终将贯穿讨论这些功能的细节，并在本书的附录A中给予总结。下面列出了本书将重点讨论的内置包：

包	描述
DBMS_ALERT	数据库报警，允许会话间通信。
DBMS_JOB	任务调度服务。
DBMS_LOB	大型对象操作。
DBMS_PIPE	数据库管道，允许会话间通信。
DBMS_SQL	动态SQL。
UTL_FILE	文本文件的输入与输出。

总的来说，所有DBMS_*包都存储在服务器中，只有包UTL_FILE既存储在服务器端又存储在客户端（某些客户环境，如Oracle表还提供了额外的包）。

1.3 本书的约定

本节介绍作者在本书中使用的几个约定。这些约定中包括用来标识PL/SQL版本的图标、引用Oracle文档资料的方法和在线案例所在的文件目录。

1.3.1 PL/SQL和Oracle 数据库版本说明

PL/SQL是随Oracle服务器一同提供的。PL/SQL的1.0版是与Oracle6.0版一起发行的。随后发

行的Oracle7提供了PL/SQL2.x系列版本。当Oracle8发布后，PL/SQL的版本号也升级到8系列。现在使用的Oracle8i（版本号为8.1）中的PL/SQL的版本号是8.1。将来发行的Oracle新版本数据库的PL/SQL版本号将与该数据库版本号保持一致。如表1-1所示，该表列出了每版Oracle数据库与对应的PL/SQL语言新增的功能。本书的重点是讨论PL/SQL2.0-8.1版的内容。本书使用以下的图标来表示某一版本具有的特殊功能：

- | | |
|----------------------|----------------------------------------------|
| PL/SQL 2.1 及
更高版本 | 该图标所在的段落将讨论PL/SQL 2.1版和更高版本具有的功能，如包DBMS_SQL。 |
| PL/SQL 2.2 及
更高版本 | 该图标所在的段落将讨论PL/SQL 2.2版和更高版本具有的功能，如游标变量。 |
| PL/SQL 2.3 及
更高版本 | 该图标所在的段落将讨论PL/SQL 2.3版和更高版本具有的功能，如包UTL_FILE。 |
| Oracle 8 及
更高版本 | 该图标所在的段落将讨论PL/SQL 8.0版和更高版本具有的功能，如对象类型。 |
| Oracle 8i 及
更高版本 | 该图标所在的段落将讨论PL/SQL 8.1版和更高版本具有的功能，如本地动态SQL。 |

表1-1 Oracle和PL/SQL版本号与功能对照表

Oracle版本	PL/SQL版本	新增或变更的功能
6	1.0	第一版
7.0	2.0	数据类型CHAR变为定长 存储子程序（包括过程、函数、包和触发器） 用户定义的复合类型——表和记录 使用DBMS_PIPE和DBMS_ALERT包进行会话间通信 在SQL*PLUS或服务器管理器中使用DBMS_OUTPUT包进行输出
7.1	2.1	用户定义的子类型 在SQL语句中使用用户定义函数的功能 使用DBMS_SQL包的动态PL/SQL
7.2	2.2	游标变量 用户定义的约束子类型 使用DBMS_JOB包调度PL/SQL批处理的功能
7.3	2.3	增强了游标变量的功能（扩充了对服务器的fetch操作和弱类（weakly typed）） 使用UTL_FILE进行文件输入输出操作 PL/SQL表属性和记录表 以编译格式存储的触发器
8.0	8.0	对象类型和方法 集合类型——嵌套表和数组 高级队列功能选项 外部过程 增强的LOB
8.1	8.1	本地动态SQL Java外部例程

(续)

Oracle版本	PL/SQL版本	新增或变更的功能
		调用权利
		NOCOPY参数
		自动事务
		大容量处理

在使用PL/SQL语言编程时，确认PL/SQL的版本号将有助于使用该版本提供的相应先进功能。当与Oracle数据库连接时，初始化字符串中带有该数据库的版本号。请看下面两个连接显示字符串：

```
Connected to:  
Oracle8 Enterprise Edition Release 8.0.6.0.0 - Production  
With the Objects option  
PL/SQL Release 8.0.6.0.0 - Production
```

和

```
Connected to:  
Oracle8i Enterprise Edition Release 8.1.5.0.0 - Production  
With the Partitioning and Java options  
PL/SQL Release 8.1.5.0.0 - Production
```

这两个字符串都是合法的初始字符串。请注意在上面的显示中，PL/SQL的版本号是与数据库的版本号完全对应的。

本书的大部分案例程序是用运行在SUN公司的Solaris操作系统上的Oracle8.0.6实现的。本书的Oracle8i案例程序也是用运行在SUN公司的Solaris操作系统上的Oracle8i，8.1.5版编制的。本书所用的所有的计算机屏幕图形都是与服务器上的Oracle数据库相连接的Windows 95或Windows NT操作系统的窗口图形。

1.3.2 Oracle数据库文档

在本书的有关章节中，我向读者推荐了包含详细信息的Oracle文档资料。由于这些文档手册的名称因数据库的版本不同而不同，我通常使用缩写的版本号来区别不同的手册。例如，Oracle服务器参考资料指的是Oracle7、Oracle8或Oracle8i服务器参考资料，具体是指哪一本资料，这取决于你正在使用的Oracle数据库版本。

1.3.3 本书提供的CD-ROM内容简介

本书附加的CD-ROM中包括下列几类信息：

- 本书使用的程序案例的代码。读者也可以在<http://www.osborne.com>站点浏览本书使用的这些程序案例。
- PL/SQL开发工具的五个试用版程序。有关这些开发工具的详细信息，请参见本书的第2章

及第3章的内容。

- 本书第15章和第16章及附录A~C电子版内容。除此之外，本书第2章使用的屏幕图形也在CD-ROM中。

有关本书CD-ROM内容的详细信息，请读者阅读CD-ROM根目录下的超文本文件readme.html。本书使用的程序案例在CD-ROM中的目录说明

本书中使用的程序案例的第一个注释行指出了该程序的名称。本书使用的所有程序案例的代码都存放在CD-ROM中CODE目录下以章节命名的子目录中。例如，请看下面我们在本章中使用的循环结构示范程序：

节选自在线代码SimpleLoop.sql

```
DECLARE
    v_LoopCounter BINARY_INTEGER := 1;
BEGIN
    LOOP
        INSERT INTO temp_table (num_col)
        VALUES (v_LoopCounter);
        v_LoopCounter := v_LoopCounter + 1;
        EXIT WHEN v_LoopCounter > 50;
    END LOOP;
END;
```

根据第一行的注释行信息，我们可以在 CD-ROM中的目录code/ch01中找到该源程序文件simple.sql。CD-ROM中CODE目录中的readme.html文件对所有的程序案例都有说明。

1.4 本书案例使用的通用数据库表

本书中使用的程序案例是一个大学入学注册系统中的数据库表，这组表中最常用的表有三个：students、classes和rooms。这三个表中带有注册系统所需的记录。除了这三个主表之外，表registered_students带有已注册课程学生的信息。下面，我们来详细介绍这些表的结构。

注意 可以使用在线文档中的脚本relTable.sql来创建上述这些数据库表。Oracle8使用的对象类型表可由脚本objTables.sql来实现。上述两个脚本都存储在CODE子目录中。本节只介绍关系型表，有关对象表，请看objTables.sql的内容。

1. 序列

student_sequence序列用来生成表students主键的唯一健值，而room_sequence序列则是为表rooms的主健生成的唯一健值。

```
CREATE SEQUENCE student_sequence
START WITH 10000
INCREMENT BY 1;
```

```
CREATE SEQUENCE room_sequence
START WITH 20000
```

```
INCREMENT BY 1;
```

2. 表students

表students中包括了入学新生的有关信息。

```
CREATE TABLE students (
    id          NUMBER(5) PRIMARY KEY,
    first_name  VARCHAR2(20),
    last_name   VARCHAR2(20),
    major       VARCHAR2(30),
    current_credits NUMBER(3)
);

INSERT INTO students (id, first_name, last_name, major,
                     current_credits)
VALUES (student_sequence.NEXTVAL, 'Scott', 'Smith',
        'Computer Science', 11);

INSERT INTO students (id, first_name, last_name, major,
                     current_credits)
VALUES (student_sequence.NEXTVAL, 'Margaret', 'Mason',
        'History', 4);

INSERT INTO students (id, first_name, last_name, major,
                     current_credits)
VALUES (student_sequence.NEXTVAL, 'Joanne', 'Junebug',
        'Computer Science', 8);

INSERT INTO students (id, first_name, last_name, major,
                     current_credits)
VALUES (student_sequence.NEXTVAL, 'Manish', 'Murgratroid',
        'Economics', 8);

INSERT INTO students(id, first_name, last_name, major,
                     current_credits)
VALUES(student_sequence.NEXTVAL, 'Patrick', 'Poll',
      'History', 4);

INSERT INTO students(id, first_name, last_name, major,
                     current_credits)
VALUES (student_sequence.NEXTVAL, 'Timothy', 'Taller',
        'History', 4);

INSERT INTO students(id, first_name, last_name, major,
                     current_credits)
VALUES (student_sequence.NEXTVAL, 'Barbara', 'Blues',
        'Economics', 7);

INSERT INTO students(id, first_name, last_name, major,
                     current_credits)
VALUES (student_sequence.NEXTVAL, 'David', 'Dinsmore',
        'Music', 4);

INSERT INTO students(id, first_name, last_name, major,
```

```

        current_credits)
VALUES (student_sequence.NEXTVAL, 'Ester', 'Elegant',
'Nutrition', 8);

INSERT INTO students(id, first_name, last_name, major,
        current_credits)
VALUES (student_sequence.NEXTVAL, 'Rose', 'Riznit',
'Music', 7);

INSERT INTO STUDENTS(id, first_name, last_name, major,
        current_credits)
VALUES (student_sequence.NEXTVAL, 'Rita', 'Razmataz',
'Nutrition', 8);

INSERT INTO students(id, first_name, last_name, major,
        current_credits)

VALUES (student_sequence.NEXTVAL, 'Shay', 'Shariatpanahy',
'Computer Science', 3);

```

3. 表major_stats

该表中保存不同专业的统计信息。

```

CREATE TABLE major_stats (
    major          VARCHAR2(30),
    total_credits  NUMBER,
    total_students NUMBER);

INSERT INTO major_stats (major, total_credits, total_students)
VALUES ('Computer Science', 22, 3);

INSERT INTO major_stats (major, total_credits, total_students)
VALUES ('History', 12, 3);

INSERT INTO major_stats (major, total_credits, total_students)
VALUES ('Economics', 15, 2);

INSERT INTO major_stats (major, total_credits, total_students)
VALUES ('Music', 11, 2);

INSERT INTO major_stats (major, total_credits, total_students)
VALUES ('Nutrition', 16, 2);

```

4. 表rooms

该表存储可用的教室有关信息。

```

CREATE TABLE rooms (
    room_id      NUMBER(5) PRIMARY KEY,
    building     VARCHAR2(15),
    room_number  NUMBER(4),
    number_seats NUMBER(4),
    description   VARCHAR2(50)
);

INSERT INTO rooms (room_id, building, room_number, number_seats,

```

```
        description)
VALUES (room_sequence.NEXTVAL, 'Building 7', 201, 1000,
       'Large Lecture Hall');

INSERT INTO rooms (room_id, building, room_number, number_seats,
                   description)
VALUES (room_sequence.NEXTVAL, 'Building 6', 101, 500,
       'Small Lecture Hall');

INSERT INTO rooms (room_id, building, room_number, number_seats,
                   description)
VALUES (room_sequence.NEXTVAL, 'Building 6', 150, 50,
       'Discussion Room A');

INSERT INTO rooms (room_id, building, room_number, number_seats,
                   description)
VALUES (room_sequence.NEXTVAL, 'Building 6', 160, 50,
       'Discussion Room B');

INSERT INTO rooms (room_id, building, room_number, number_seats,
                   description)
VALUES (room_sequence.NEXTVAL, 'Building 6', 170, 50,
       'Discussion Room C');

INSERT INTO rooms (room_id, building, room_number, number_seats,
                   description)
VALUES (room_sequence.NEXTVAL, 'Music Building', 100, 10,
       'Music Practice Room');

INSERT INTO rooms (room_id, building, room_number, number_seats,
                   description)
VALUES (room_sequence.NEXTVAL, 'Music Building', 200, 1000,
       'Concert Room');

INSERT INTO rooms (room_id, building, room_number, number_seats,
                   description)
VALUES (room_sequence.NEXTVAL, 'Building 7', 300, 75,
       'Discussion Room D');

INSERT INTO rooms (room_id, building, room_number, number_seats,
                   description)
VALUES (room_sequence.NEXTVAL, 'Building 7', 310, 50,
       'Discussion Room E');
```

5. 表classes

该表描述了学生可以选择的课程。

```
CREATE TABLE classes (
    department      CHAR(3),
    course         NUMBER(3),
    description     VARCHAR2(2000),
    max_students   NUMBER(3),
    current_students NUMBER(3),
```

```

num_credits NUMBER(1),
room_id NUMBER(5),
CONSTRAINT classes_department_course
    PRIMARY KEY (department, course),
CONSTRAINT classes_room_id
    FOREIGN KEY (room_id) REFERENCES rooms (room_id)
);

INSERT INTO classes(department, course, description, max_students,
                    current_students, num_credits, room_id)
VALUES ('HIS', 101, 'History 101', 30, 11, 4, 20000);

INSERT INTO classes(department, course, description, max_students,
                    current_students, num_credits, room_id)
VALUES ('HIS', 301, 'History 301', 30, 0, 4, 20004);

INSERT INTO classes(department, course, description, max_students,
                    current_students, num_credits, room_id)
VALUES ('CS', 101, 'Computer Science 101', 50, 0, 4, 20001);

INSERT INTO classes(department, course, description, max_students,
                    current_students, num_credits, room_id)
VALUES ('ECN', 203, 'Economics 203', 15, 0, 3, 20002);

INSERT INTO classes(department, course, description, max_students,
                    current_students, num_credits, room_id)
VALUES ('CS', 102, 'Computer Science 102', 35, 3, 4, 20003);

INSERT INTO classes(department, course, description, max_students,
                    current_students, num_credits, room_id)
VALUES ('MUS', 410, 'Music 410', 5, 4, 3, 20005);

INSERT INTO classes(department, course, description, max_students,
                    current_students, num_credits, room_id)
VALUES ('ECN', 101, 'Economics 101', 50, 0, 4, 20007);

INSERT INTO classes(department, course, description, max_students,
                    current_students, num_credits, room_id)
VALUES ('NUT', 307, 'Nutrition 307', 20, 2, 4, 20008);

INSERT INTO classes(department, course, description, max_students,
                    current_students, num_credits, room_id)
VALUES ('MUS', 100, 'Music 100', 100, 0, 3, NULL);

```

6. 表registered_students

该表保存学生目前参加的课程信息。

```

CREATE TABLE registered_students (
    student_id NUMBER(5) NOT NULL,
    department CHAR(3) NOT NULL,
    course      NUMBER(3) NOT NULL,
    grade       CHAR(1),
CONSTRAINT rs_grade
    CHECK (grade IN ('A', 'B', 'C', 'D', 'E')),

```

```
CONSTRAINT rs_student_id
    FOREIGN KEY (student_id) REFERENCES students (id),
CONSTRAINT rs_department_course
    FOREIGN KEY (department, course)
        REFERENCES classes (department, course)
);
INSERT INTO registered_students (student_id, department, course,
                                 grade)
VALUES (10000, 'CS', 102, 'A');

INSERT INTO registered_students (student_id, department, course,
                                 grade)
VALUES (10002, 'CS', 102, 'B');

INSERT INTO registered_students (student_id, department, course,
                                 grade)
VALUES (10003, 'CS', 102, 'C');

INSERT INTO registered_students (student_id, department, course,
                                 grade)
VALUES (10000, 'HIS', 101, 'A');

INSERT INTO registered_students (student_id, department, course,
                                 grade)
VALUES (10001, 'HIS', 101, 'B');

INSERT INTO registered_students (student_id, department, course,
                                 grade)
VALUES (10002, 'HIS', 101, 'B');

INSERT INTO registered_students (student_id, department, course,
                                 grade)
VALUES (10003, 'HIS', 101, 'A');

INSERT INTO registered_students (student_id, department, course,
                                 grade)
VALUES (10004, 'HIS', 101, 'C');

INSERT INTO registered_students (student_id, department, course,
                                 grade)
VALUES (10005, 'HIS', 101, 'C');

INSERT INTO registered_students (student_id, department, course,
                                 grade)
VALUES (10006, 'HIS', 101, 'E');

INSERT INTO registered_students (student_id, department, course,
                                 grade)
VALUES (10007, 'HIS', 101, 'B');

INSERT INTO registered_students (student_id, department, course,
                                 grade)
VALUES (10008, 'HIS', 101, 'A');
```

```

INSERT INTO registered_students (student_id, department, course,
                                grade)
VALUES (10009, 'HIS', 101, 'D');

INSERT INTO registered_students (student_id, department, course,
                                grade)
VALUES (10010, 'HIS', 101, 'A');

INSERT INTO registered_students (student_id, department, course,
                                grade)
VALUES (10008, 'NUT', 307, 'A');

INSERT INTO registered_students (student_id, department, course,
                                grade)
VALUES (10010, 'NUT', 307, 'A');

INSERT INTO registered_students (student_id, department, course,
                                grade)
VALUES (10009, 'MUS', 410, 'B');

INSERT INTO registered_students (student_id, department, course,
                                grade)
VALUES (10006, 'MUS', 410, 'E');

INSERT INTO registered_students (student_id, department, course,
                                grade)
VALUES (10011, 'MUS', 410, 'B');

INSERT INTO registered_students (student_id, department, course,
                                grade)
VALUES (10000, 'MUS', 410, 'B');

```

7. 表RS_audit

该表用来记录对表registered_students所做的修改。

```

CREATE TABLE RS_audit (
    change_type      CHAR(1)      NOT NULL,
    changed_by      VARCHAR2(8)   NOT NULL,
    timestamp       DATE        NOT NULL,
    old_student_id NUMBER(5),
    old_department  CHAR(3),
    old_course      NUMBER(3),
    old_grade       CHAR(1),
    new_student_id NUMBER(5),
    new_department  CHAR(3),
    new_course      NUMBER(3),
    new_grade       CHAR(1)
);

```

8. 表log_table

该表用来记录Oracle数据库发生的错误信息。

```

CREATE TABLE log_table (
    code NUMBER,

```

```
message VARCHAR2(200),  
info      VARCHAR2(100)  
);
```

9. 表temp_table

该表用来存放临时数据。

```
CREATE TABLE temp_table (  
    num_col NUMBER,  
    char_col VARCHAR2(60)  
);
```

10. 表connect_audit

该表由第6章的程序案例用来记录与数据库的连接和断开信息。

```
CREATE TABLE connect_audit (  
    user_name VARCHAR2(30),  
    operation VARCHAR2(30),  
    timestamp DATE);
```

11. 表debug_table

该表由本书第3章的Debug包用来保存PL/SQL调试信息。

```
CREATE TABLE debug_table (  
    linecount NUMBER,  
    debug_str VARCHAR2(100)  
);
```

12. 表source和destination

这两个表是由第3章中调试程序案例使用的。

```
CREATE TABLE source (  
    key NUMBER(5),  
    value VARCHAR2(50) );  
  
CREATE TABLE destination (  
    key NUMBER(5),  
    value NUMBER);
```

1.5 小结

我们在本章概述了引入PL/SQL语言的目的及该语言的主要特点。除此之外，我们还讨论了PL/SQL和数据库版本号的对应规则，介绍了本书CD-ROM中的内容和程序案例使用的数据库表。在下面的两章中，我们将讨论PL/SQL的各种开发、调试以及运行环境。

第2章 PL/SQL开发和运行环境

PL/SQL块可以在多种不同特点及功能的环境下运行。我们在本章主要讨论定位 PL/SQL引擎的有关问题。除此之外，我们还将讨论各种可用来开发 PL/SQL应用的环境，其中包括 Oracle本身提供的开发工具和由第三方开发商提供的开发工具。

2.1 应用模式和PL/SQL

一般的数据库应用可以分为三个部分：

- 用户界面，负责提供应用的外观和使用方式。该部分负责处理用户的输入信息和显示处理结果。
- 应用逻辑，这层的主要功能是控制应用的处理流程。
- 数据库，该层负责可靠地存储应用数据。

目前可以将上述三部分功能分配到不同位置的数据库应用设计模式主要有两类。

为了编译并运行一个 PL/SQL块，程序员必须将该块提交给 PL/SQL引擎来处理。与 Java语言的虚拟机相类似，PL/SQL引擎也是由编译器和运行时系统组成。借助于 Oracle公司和其他开发商提供的开发工具，PL/SQL可以用于应用的各个层次，并且 PL/SQL引擎也可以宿主在不同的系统中。

2.1.1 两层模式

两层模式，即客户 / 服务器模式，是传统的应用设计模式。在这种模式中，应用由客户端程序和服务器端程序两部分组成。客户端负责处理用户界面，而服务器端管理数据库。这种模式的应用逻辑分为客户端和服务器端两部分。通常，PL/SQL引擎驻留在服务器端，在个别情况下，PL/SQL引擎也可以驻留在客户端。

1. 服务器端的PL/SQL

从Oracle6.0版开始，PL/SQL就驻留在数据库服务器端，同时，该服务器也是 PL/SQL引擎的默认位置。由于数据库服务器可以处理 SQL语句，所以 SQL语句和PL/SQL块都可以送到该服务器进行处理。一个客户应用，不管是用 Oracle开发工具实现的或使用其他开发工具编制的，都可以向数据库服务器提交 SQL语句和PL/SQL块。SQL *Plus就是一个这种客户应用的案例，该程序可以在SQL提示符下接收交互输入的SQL语句和PL/SQL命令并将其送往服务器执行。

例如，我们可以假设在SQL *Plus与服务器建立了连接的情况下输入下列的SQL，PL/SQL命令：

节选自在线代码SQL_PLSQL.SQL

```
SQL> CREATE OR REPLACE PROCEDURE ServerProcedure AS
 2 BEGIN
 3 NULL;
 4 END ServerProcedure;
```

```
5 /
Procedure created.

SQL> DECLARE
 2   v_StudentRecord students%ROWTYPE;
 3   v_Counter BINARY_INTEGER;
 4   BEGIN
 5     v_Counter := 7;
 6
 7   SELECT *
 8     INTO v_StudentRecord
 9     FROM students
10    WHERE id = 10001;
11
12  ServerProcedure;
13
14 END;
15 /
PL/SQL procedure successfully completed.
```

```
SQL> UPDATE classes
 2   SET max_students = 70
 3   WHERE department = 'HIS'
 4   AND course = 101;
1 row updated.
```

注意 可以在本书的CD-ROM中找到上面的例子的代码。本书中存储在CD-ROM中的程序案例的文件名称在该例子的第一行注释中给予了说明，读者可以根据注释中提供的XXX.sql在CD-ROM中找到对应的程序文件。CD-ROM中CODE目录下的readme.html文件提供了对这些程序案例的说明。

图2-1演示了在服务器端的PL/SQL引擎对PL/SQL块的处理过程。客户应用可以向服务器提交PL/SQL块（该块可以带有过程和包括调用服务器端存储过程的SQL语句），以及单独的SQL语句。如图所示，PL/SQL块和SQL语句通过网络送往服务器。一旦服务器收到了这些内容，SQL语句将直接进入服务器内含的SQL语句执行器，而PL/SQL块则送往PL/SQL引擎进行语法分析。在该块的运行期间，PL/SQL引擎负责执行过程语句（如赋值语句和存储过程调用）。对于该块中出现的SQL语句（如SELECT语句等），PL/SQL引擎将它们送往SQL语句执行器执行。

2. 客户端的PL/SQL

除了在服务器端的PL/SQL引擎外，几种Oracle开发工具也带有PL/SQL引擎。由于这些开发工具是运行在客户端的，所以它们的PL/SQL引擎也可以运行在客户端。这样一来，借助于客户端的PL/SQL支持，PL/SQL块中的过程语句就可以在本地运行，而没有必要送到服务器端。例如开发工具Oracle Forms（该工具是Oracle Developer的一部分）自身带有PL/SQL引擎；在Oracle Developer工具包中，如Oracle Reports也带有PL/SQL引擎。需要指出的是这种引擎与PL/SQL服务器端的引擎有所不同。PL/SQL块只能出现在客户端应用中，并且该块必须用开发工具来编制。假设一个Oracle Forms应用包括了触发器和过程，这些语句都在客户端执行。只有该程序中的

SQL语句和调用服务器端存储子程序的语句被送往服务器进行处理。如图 2-2所示，客户端的PL/SQL引擎可以处理过程语句。

与服务器端的PL/SQL一样，应用程序提交的单独的SQL语句（如UPDATE语句）直接通过网络送往服务器端的SQL语句执行器。不同的是，PL/SQL块是在客户端直接处理。任何过程语句（如赋值语句）的处理都不会引起网络传输。PL/SQL块中的SQL语句要提交给SQL语句执行器，对服务器端的存储子程序的调用则是送到服务器端的PL/SQL引擎执行。

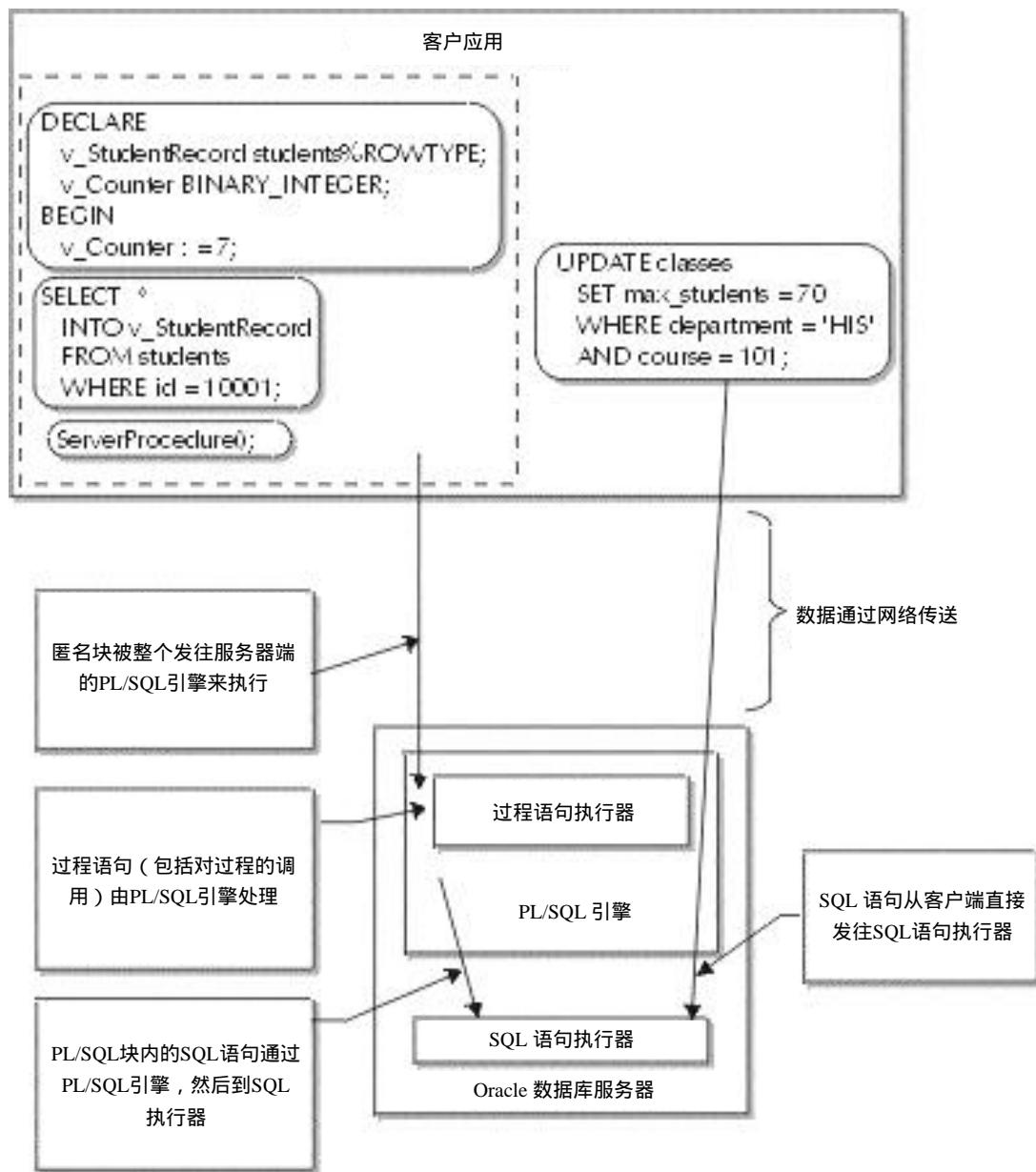


图2-1 服务器端PL/SQL引擎

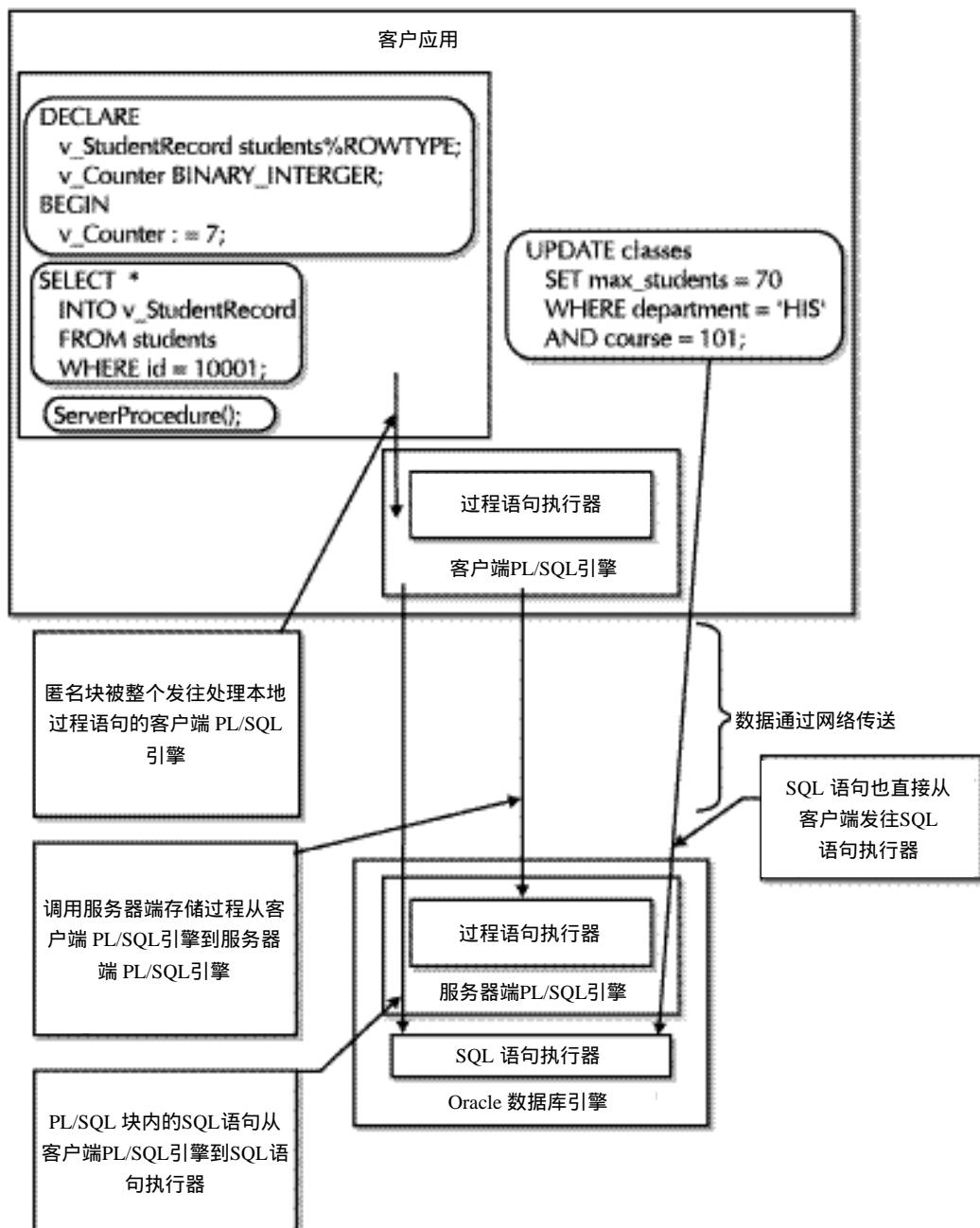


图2-2 客户端PL/SQL引擎

3. Oracle 预编译器

程序员可以使用 Pro *C/C++ 和 Pro *COBOL 一类的 Oracle 预编译器创建运行在服务器端的应用程序。用这种方法实现的应用将不包括 PL/SQL 引擎，因此，由这种应用提交的 SQL 和 PL/SQL 语句都要送到服务器进行处理。

预编译器本身包括了PL/SQL引擎，该引擎在预编译中用来校验应用代码中匿名块的语法和语义是否正确。预编译功能是Oracle开发工具的一大特点。

4. 引擎间的通信

在图2-2所示的流程图中，有两个各自独立但又相互通信的PL/SQL引擎。例如，一个运行在客户端PL/SQL下的报表（Form）中的触发器可以调用在服务器端PL/SQL下运行的存储过程，这一类的网络通信是通过远程过程调用实现的。借助于类似的机制，通过数据库连接可以实现不同PL/SQL引擎之间的通信。

在上述情况下，不同引擎中的PL/SQL对象可以相互依赖。这种存在依赖关系的类型工作方式与在同一个数据库中的PL/SQL对象几乎完全一样，不同之处是程序中必须提供一些防止产生误解的声明。本书的第5章提供了进一步的信息。

总的来说，两个PL/SQL引擎可以是不同的版本。例如，Oracle开发器1.2版使用了PL/SQL第1版，而服务器使用了PL/SQL版本2（如果使用了Oracle8，就需要使用PL/SQL版本8），这就意味着PL/SQL版本2或更高的版本提供的功能，如用户自定义表和记录，定长的CHAR数据类型和其他一些功能可能不会出现在客户端引擎中。尽管Oracle开发器第二版以上的工具中包括了PL/SQL8.0版，但在引擎之间仍然存在着版本差别，因此，每一个版本可用的功能也不一样。

2.1.2 三层模式

在三层模式中，用户界面，应用逻辑和数据库是三个各自独立的部分。该模式下的客户是典型的瘦客户类型，如浏览器一类的客户软件。应用层逻辑全部位于称为应用服务器的独立层中。在这种环境下，PL/SQL引擎通常只放置在服务器中。

Oracle应用服务器（OAS）具有一般应用服务器的全部功能。通过PL/SQL盒式磁带机，读者可以运行服务器上的存储过程并返回HTML页面形式的处理结果，这项功能可以借助于OAS提供的PL/SQL Web工具箱实现。图2-3演示了三层模式的内部结构。有关PL/SQL盒式磁带机和Web工具箱的进一步介绍，请读者参考Oracle文档。

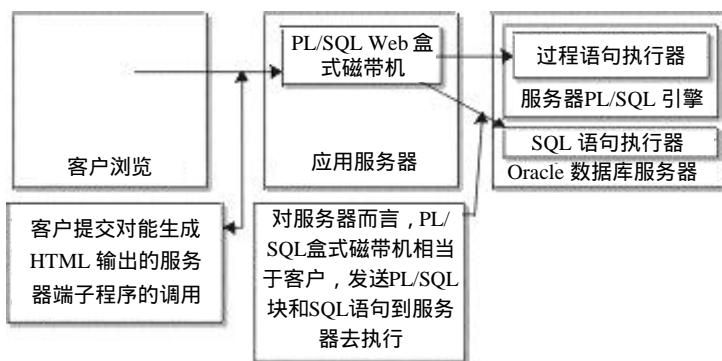


图2-3 三层模型示意图

2.2 PL/SQL开发工具介绍

开发调试PL/SQL应用可以使用多种不同的开发工具，每种开发工具都有其优点与不足。表

2-1介绍了我们在本章详细讨论的开发工具的基本情况。从表中我们可以看出，SQL *Plus是Oracle公司与服务器一起提供的开发工具，其他的开发工具则是由第三方开发商提供的。本书的CD-ROM中带有第三方开发工具的试用版，我们将在下面几节中将介绍这些开发工具的使用方法。在第3章，我们将详细介绍这些开发工具的调试功能。

注意 CD-ROM中的开发工具存放在Development Tools目录中，CD-ROM中根目录下的readme.html文件中有对开发工具的说明。

表2-1 PL/SQL开发环境一览

工具名称	开发商	Web站点地址	是否随CD-ROM提供
SQL *Plus	Oracle公司	www.oracle.com	不
Rapid SQL	Embarcadero技术公司	www.embarcadero.com	是
XPEDITER/SQL	Compuware	www.compuware.com	是
SQL Navigator	Quest Software	www.quest.com	是
TOAD	Quest Software	www.quest.com www.toadsoft.com	是
SQL-Programmer	Sylvain Faust International	www.sfi-software.com	是

为了保持一致性，下面讨论的每一种开发工具都采用相同的模式，即使用几种不同类型的PL/SQL对象案例来进行说明，以便了解每一种工具在同一个环境下的优缺点。表2-2描述了样本模式的具体内容，该样本模式的每一项都可以在执行安装脚本relTables.sql后运行表中指示的脚本文件来自动创建。本书的在线文件Setup.sql也可以用来运行这些脚本文件。需要指出的是，在创建外部过程和函数时，需要提供系统特权CREATE LIBRARY。

注意 在下面几节中，每种开发工具都带有图形窗口。由于受到本书空间的限制，某些工具的图形窗口只能存放在本书的CD-ROM中，有关这些图形窗口的说明，请看CD-ROM中目录online Chapters中的文件ch02ScreenShots.html。

表2-2 样本模式一览表

对象名称	对象类型	可运行的脚本
AddNewStudent	过程	ch04/AddNewStudent.sql
AlmostFull	函数	ch04/AlmostFull.sql
ModeTest	过程	ch04/ModeTest.sql
ClassPackage	包和包体	ch04/ClassPackage.sql
Point	对象类型和类型体	ch12/Point.sql
OutputString	外部过程和函数	ch10/OutputString.sql

2.2.1 SQL*Plus

SQL *Plus可能是最简单的PL/SQL开发工具。该工具允许用户交互式地从输入提示符输入SQL语句和PL/SQL块，这些输入的语句直接送到数据库执行，并将执行结果显示在终端屏幕上。该工具支持字符环境，没有内置的本地PL/SQL引擎。

一般，SQL *Plus是与Oracle服务器捆绑销售，并作为标准 Oracle安装过程的一部分执行。读者可以参阅SQL *Plus用户指南和参考手册来了解该工具的详细说明和有关命令。

SQL *Plus的命令不区分大小写字母。例如，下面三个命令都可以声明连接变量：

```
SQL> VARIABLE v_Num NUMBER
SQL> variable v_Char char(3)
SQL> vaRIAbLe v_Varchar VarCHAR2(5)
```

1. 连接数据库

在SQL *Plus下输入SQL或PL/SQL命令之前，必须先实现与数据库服务器的连接。下面是实现数据库连接的两种常用方法：

- 在SQL *Plus命令行输入用户标识（Userid）和口令，或输入连接字串。
- 进入SQL *Plus后使用CONNECT语句。

在下面的连接例子中，读者可以看到，如果没有定义口令的话，SQL *Plus将不会为用户提示口令内容并且不将输入字符回送到屏幕显示。

```
$ sqlplus example/example
SQL*Plus: Release 8.0.6.0.0 - Production on Wed Nov 3 10:29:11 1999
(c) Copyright 1999 Oracle Corporation. All rights reserved.

Connected to:
Oracle8 Enterprise Edition Release 8.0.6.0.0 - Production
With the Objects option
PL/SQL Release 8.0.6.0.0 - Production

SQL> exit
Disconnected from Oracle8 Enterprise Edition Release 8.0.6.0.0 -
Production
With the Objects option
PL/SQL Release 8.0.6.0.0 - Production
$ sqlplus example
SQL*Plus: Release 8.0.6.0.0 - Production on Wed Nov 3 10:29:15 1999
(c) Copyright 1999 Oracle Corporation. All rights reserved.

Enter password:
```

```
Connected to:
Oracle8 Enterprise Edition Release 8.0.6.0.0 - Production
With the Objects option
PL/SQL Release 8.0.6.0.0 - Production

SQL> connect example/example
Connected.
SQL> connect example/example@v806_tcp
Connected.
```

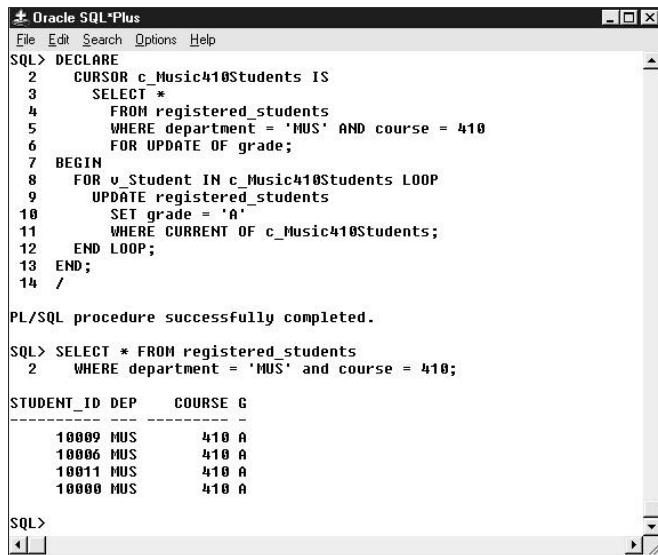
2. SQL *Plus中的块操作

当在SQL *Plus中执行一个SQL语句时，语句末尾的分号标识了该语句的结束。该分号不属于语句本身，它是一个独立的语句终止符。当SQL *Plus读到一个分号时，它就知道该语句已经

下载

结束并将其发送到数据库执行。另一方面，对于 PL/SQL块来说，分号则是块本身的语法部分，而不是语句终止符。当输入关键字 DELCARE或BEGIN时，SQL *Plus就能够检测到该关键字并确认正在输入的是 PL/SQL块，不是SQL语句。在这种情况下，SQL *Plus仍然需要确认输入的PL/SQL块何时结束，我们使用一个正斜杠来表示块结束，这种正斜杠是 SQL *Plus RUN命令的缩写形式。

请注意，图 2-4 中更新表 registered_students的PL/SQL块之后有一个正斜杠。该块后面的 SELECT语句由于使用了分号而不用再输入斜杠（如果需要的话，读者也可以在 SQL语句中使用斜杠来代替分号）。



```

* Oracle SQL*Plus
File Edit Search Options Help
SQL> DECLARE
  2   CURSOR c_Music410Students IS
  3     SELECT *
  4       FROM registered_students
  5     WHERE department = 'MUS' AND course = 410
  6     FOR UPDATE OF grade;
  7 BEGIN
  8   FOR v_Student IN c_Music410Students LOOP
  9     UPDATE registered_students
10       SET grade = 'A'
11     WHERE CURRENT OF c_Music410Students;
12   END LOOP;
13 END;
14 /

PL/SQL procedure successfully completed.

SQL> SELECT * FROM registered_students
  2   WHERE department = 'MUS' and course = 410;

STUDENT_ID DEP      COURSE G
-----
10009 MUS      410 A
10006 MUS      410 A
10011 MUS      410 A
10000 MUS      410 A

SQL>

```

图2-4 在SQL *Plus中输入PL/SQL块

3. 替换和连接 (bind) 变量

由于PL/SQL是服务器端的专用语言，所以该语言不支持用户输入和输出。文件输入输出操作是通过包 UTL_FILE实现的（需PL/SQL 2.3版和更高版本支持，本书第7章有专门介绍），借助于BFILE接口，Oracle8可以读入外部文件（第 15 ~ 16章有专文论述）。除此之外，与 SQL *Plus一样，使用DBMS_OUTPUT包可以实现有限的屏幕输出操作（详细内容请参阅第 3章）。

上述的各种方法都不支持接收用户的输入，其主要原因是 PL/SQL所在的运行环境所致。例如，Pro *C 程序可以使用C语言的 I/O库函数功能来接收用户输入并将其传入 PL/SQL中。

SQL *Plus提供了两种不同类型的变量来接收用户的输入并通过多个运行存储信息，这两种变量就是替换和连接变量。

替换变量 替换变量在 PL/SQL块或 SQL语句中是用字符符号“ & ”描述的（使用 SET DEFINE命令可以指定替换“ & ”的字符）。SQL *Plus在将PL/SQL块或SQL语句发送到服务器前要对变量进行彻底的原文替换，类似于 C语言对宏的处理。

图2-5中的PL/SQL块中使用了替换变量。该块运行了两次，每次都对变量 v_StudentID给予了不同的初始值。用户分别输入了数值 10004和10005，这两个数值在该块中被 &student-id替换。

```

SQL> DECLARE
 2   v_StudentID students.ID%TYPE := &student_ID;
 3 BEGIN
 4   ClassPackage.AddStudent(v_StudentID, 'CS', 102);
 5 END;
 6 /
Enter value for student_id: 10004
old  2:  v_StudentID students.ID%TYPE := &student_ID;
new  2:  v_StudentID students.ID%TYPE := 10004;

PL/SQL procedure successfully completed.

SQL> /
Enter value for student_id: 10005
old  2:  v_StudentID students.ID%TYPE := &student_ID;
new  2:  v_StudentID students.ID%TYPE := 10005;

PL/SQL procedure successfully completed.

SQL>

```

图2-5 SQL*Plus替换变量

需要注意的是替换变量实际上不会占用内存空间。SQL *Plus在把该块发送到数据库执行前使用用户输入的数值来置换替换变量。由于这种原因，替换变量只能用于输入。下面讨论的连接变量则可以用于输入和输出双向操作。

提示 假设现在从提示符SQL>下输入下列SQL语句：

```
SQL> SELECT *
  FROM students
 WHERE first_name = &first_name;
```

输入该语句后，当SQL *Plus提示输入时，用户必须要在输入的字符串两端使用单引号，如‘SCOTT’。现在请比较下面的语句：

```
SQL> SELECT *
  FROM students
 WHERE first_name = '&first_name';
```

在这种情况下，用户不必使用单引号，因为该语句中已经有了单引号。单引号的使用与否取决于要替换的内容。

连接变量 我们在上面讲过，替换变量不会占用内存空间，然而，SQL *Plus可以分配内存空间供PL/SQL块和SQL语句内部使用。因为这种存储空间位于块的外部，所以它可以作为一个以上的连续的块和SQL语句共用，并且该存储空间的内容还可以在块结束退出后打印显示，我们描述的这种变量就是连接变量。该类变量的图解说明在CD-ROM中提供的图CD2-1中。连接变量v_Count是用SQL*Plus的VARIABLE命令为其分配内存的。请注意，VARIABLE命令只能在SQL提示符下使用，它不能用在PL/SQL块中。在块内部，连接变量是用起始处的冒号来定界的。在该块结束后，命令PRINT将显示该变量的值。SQL*Plus允许使用的合法连接变量类型包括VARCHAR2、CHAR和NUMBER。连接变量REFCURSOR只能用于SQL *Plus3.2版和更高版本。

下载

连接变量NCHAR、NVARCHAR2、CLOB和NCLOB只能用于SQL *Plus8.0版及更高版本。如果没有特殊指定连接变量类型VARCHAR2、CHAR、NCHAR或NVARCHAR2的长度，这些变量的默认长度都为1，NUMBER类型的连接变量不受精度和比例的限制。

4. 变量和数据库对象

由于替换变量在PL/SQL块被发送到服务器之前就被进行了替换，所以这类变量可以作为SQL语句和数据库对象的语法部分使用，而连接变量则不能在这种情况下使用。下面的SQL *Plus对话演示了这种规则。

节选自在线代码Variables.sql

```
SQL> SELECT &columns
  2 FROM classes;
Enter value for columns: department, course
old 1: SELECT &columns
new 1: SELECT department, course
```

```
DEP      COURSE
```

```
-----
```

DEP	COURSE
HIS	101
HIS	301
CS	101
ECN	203
CS	102
MUS	410
ECN	101
NUT	307
MUS	100

9 rows selected.

```
SQL> SELECT first_name, last_name
  2 FROM students
  3 WHERE &where_clause;
Enter value for where_clause: ID = 10001
old 3: WHERE &where_clause
new 3: WHERE ID = 10001
```

```
FIRST_NAME      LAST_NAME
```

```
-----
```

FIRST_NAME	LAST_NAME
Margaret	Mason

```
SQL> -- But a bind variable cannot be used in this manner:
```

```
SQL> VARIABLE where_clause VARCHAR2(100)
SQL>
SQL> BEGIN
 2   :where_clause := 'WHERE ID = 10001';
 3 END;
4 /
```

```
PL/SQL procedure successfully completed.
SQL> PRINT :where_clause
```

```
WHERE_CLAUSE
-----
WHERE ID = 10001

SQL> SELECT first_name, last_name
  2   FROM students
  3 WHERE :where_clause;
WHERE :where_clause
*
ERROR at line 3:
ORA-00920: invalid relational operator
```

读者可以使用动态SQL在运行期间建立SQL语句和PL/SQL块，并使用PL/SQL变量来构造这些语句。有关详细内容，请看本书的第8章。

5. 使用EXECUTE命令调用存储过程

存储过程调用只能从PL/SQL块的可执行语句部分或其异常处理部分中执行。SQL *Plus为这种调用提供了一个有用的简写方式，即命令EXECUTE。该命令实现的功能是在其命令参数之前加上BEGIN，而在参数之后加上END。并同时在调用语句后加入一个分号，表示调用语句结束。经过这种处理后形成的块就被提交给数据库执行。例如，假设我们在SQL命令提示符下输入下列命令：

```
SQL> EXECUTE ClassPackage.AddStudent(10006, 'CS', 102)
```

则SQL *Plus就可以生成下面的PL/SQL块：

```
BEGIN ClassPackage.AddStudent(10006, 'CS', 102); END;
```

该块被发送到数据库执行。关键字EXECUTE之后的分号是该命令的选择项，这对所有的SQL *Plus命令都是适用的。并且如果使用了该分号的话，它将被忽略不记，就像命令PRINT，VARIABLE一样，EXECUTE是SQL *Plus的命令，它不能用在PL/SQL块内部。

提示 EXECUTE命令并不在SQL缓冲区中存储生成的匿名块，但如果匿名块是直接输入的话，该命令可以实现这种功能。

6. 使用文件

SQL *Plus可以把当前的PL/SQL块或SQL语句保存在文件中，并在需要时，将该文件读入系统执行。SQL *Plus的这种功能对于实现先开发PL/SQL程序后运行调试来说非常有用。例如，读者可以把命令CREATE或REPLACE存储在一个文件中。利用这种方法，任何对该过程的修改都可以通过该文件实现。为了保存数据库的修改，我们可以简单地把该文件读入到SQL *Plus中。文件中可以存放一个以上的命令。

SQL *Plus的GET命令将文件从磁盘读入到本地缓冲区中存放。输入一个正斜杠就可以运行读入的文件，就像该文件中的命令是从键盘直接输入的一样。如果文件的结尾处有一个正斜杠的话，就可以使用GET命令的缩写形式@将该文件读入并执行。例如，假设文件file.sql包含下列

命令行：

```
节选自在线代码File.sql
BEGIN
  FOR v_Count IN 1..10 LOOP
    INSERT INTO temp_Table (num_col, char_col)
      VALUES (v_Count, 'Hello World!');
  END LOOP;
END;
/
SELECT * FROM temp_table;
```

现在，我们可以从SQL提示符模式下使用下面的命令运行该文件。

```
SQL> @file
```

执行该文件的输出结果在本书 CD-ROM 中的图 CD2-2 所示。命令 SET ECHO ON (该图中的第一条命令) 告诉 SQL *Plus 在读入该文件时把其内容送到屏幕显示。

7. 使用 SHOW ERRORS 命令

在创建存储过程时，有关该过程的信息存储在数据字典中。所有的编译错误都存储在 user_errors 的数据字典中。SQL *Plus 提供了一个可以查询该数据字典并报告错误的命令 SHOW ERRORS。本书的 CD-ROM 上的图 CD2-3 演示了该命令的使用方法。命令 SHOW ERRORS 可以用在 SQL *Plus 报告了下列错误信息后：

```
Warning: Procedure created with compilation errors.
```

2.2.2 Rapid SQL

由 Embarcadero Technologies 公司开发的 Rapid SQL (快速 SQL) 提供了图形用户界面的开发环境，其主要功能如下：

- PL/SQL 和 SQL 语句的自动格式化
- PL/SQL 调试器
- 支持 Oracle8 的对象类型和表分区
- SQL 任务调度
- 工程管理
- 支持 Windows 活动脚本 (active scripting)
- 支持第三方版本控制系统
- 集成数据库开发和 Web 程序设计

有关该工具的安装和使用方法，请参考有关在线帮助。

1. 连接数据库

当第一次启动运行 Rapid SQL 时，该程序将显示如图 2-6 所示的窗口。该窗口左面的窗格中显示了按数据源分类的可浏览数据库对象。右面的窗格是工作窗口，用来显示正在浏览的不同类型的对象，该窗口中的数据源记录了远程数据库的所有相关信息，如数据库类型、用户标识、

口令和连接信息。当 Rapid SQL第一次运行时，它将通过检索当前计算机上的 SQL *Net或Net8配置自动地来发现可用的数据源。双击某个特殊的数据源将会启动一个对话框来接收连接使用的用户标识和口令，读者可以参考CD-ROM中的图CD2-4来了解该操作过程。除此之外，也可以通过数据源菜单增加新的数据源。

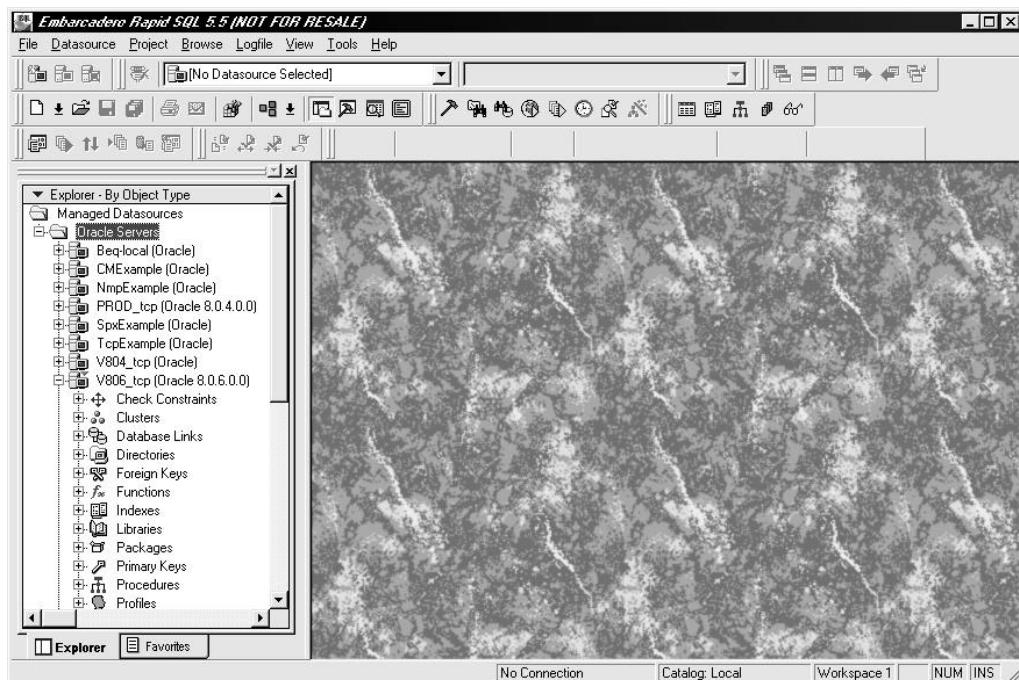


图2-6 Rapid SQL的主窗口

如果选择已建立的数据源的话，Rapid SQL将自动使用用户标识和口令，而不需要再次输入。Rapid SQL允许同时打开多个数据库连接。

2. 浏览数据库对象

一旦建立了数据库连接，我们就可以浏览左窗格中的对象。单击对象类型将使显示的目录中增加所选对象，这时，我们就可以进一步观察列表目录中某个单独的对象。双击某个对象将会启动创建该对象的SQL或PL/SQL。例如，图2-7中显示了对象浏览器中的包和表以及在右面的工作区中显示了表classes的DDL。

3. 编辑器类型

Rapid SQL提供了两类不同的窗口供用户输入 SQL语句和PL/SQL块，每种窗口适用于不同的任务：

- SQL编辑器是通用的编辑器，适用于输入 SQL DML和DDL语句，以及匿名PL/SQL块和CREATE语句。
- DDL编辑器允许用户选择待创建的对象类型并自动地用结构对其进行填充。

这两种编辑器可以从主窗口的 File | New 的子菜单中选择。CD-ROM中的图CD2-5图示了使

下载

用匿名块的PL/SQL编辑器。单击绿色的三角形将把该块发送到服务器运行。

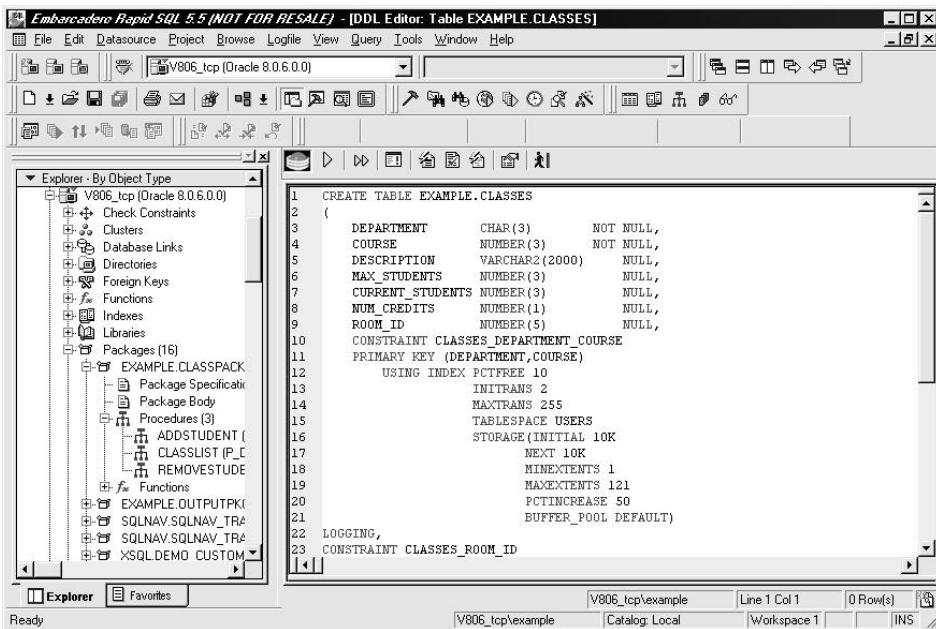


图2-7 浏览表classes

4. 自动格式

任何编辑器都可以对SQL和PL/SQL语句进行自动格式化。自动格式操作包括缩进和各种语法元素的识别。这种功能非常有助于根据程序员自己的程序设计风格来格式化PL/SQL块。CD-ROM中的图CD2-6中列举了自动格式的选择项。

5. 观察错误

把PL/SQL块提交给服务器后，只要单击Errors工具条就可以显示任何错误信息。这时浏览器中与错误对应的对象上也标有X标记，表示该对象有语法错误。CD-ROM中图2-7演示了显示错误的窗口图形。错误将显示在PL/SQL文本下面的窗格中。单击某个错误将使光标移动到产生错误的语句所在的位置。

6. 作业调度

Rapid SQL将Windows98和NT提供的作业调度合并入该软件中，这种功能的引入可以使用户在确定的时间独立于Rapid SQL Pro运行作业。

7. 集成Web开发功能

借助于附加的编辑器，Rapid SQL还可以编辑Java代码和HTML页面。有关这方面的信息，请参阅在线文档。

2.2.3 XPEDITER/SQL

XPEDITER/SQL是由Compuware公司发行的开发工具，它也提供图形用户界面开发环境。

该工具的主要功能如下：

- 自动格式PL/SQL和SQL语句
- 提供PL/SQL调试器
- 支持Oracle8对象类型
- 工程管理
- 对任何应用提供调试和跟踪功能
- 支持第三方版本控制系统

1. 连接数据库

当XPEDITOR/SQL首次运行时，该软件将提示用户建立数据库连接。为了方便建立数据库连接，我们可以事先存储不同数据库连接所需的配置信息，这些信息包括用户标识和数据库连接字符串。可以将用户标识的格式指定为“用户标识 / 口令”的格式，这样一来，用户口令也可以存储在配置文件中。建立数据库连接的对话框的窗口在本书 CD-ROM 中的图 CD2-8 显示。XPEDITOR/SQL 允许同时建立多个连接。

2. 服务器端的安装

为了正确的使用XPEDITOR/SQL，必须在服务器端进行安装操作。该安装将创建一个包括存储XPEDITOR/SQL所需信息的表的数据库用户。该数据库用户的创建可以在安装过程中完成，也可以在安装结束后，使用数据库安装向导来实现。该向导的窗口在 CD-ROM 中的图 CD2-9 中演示。数据库安装向导允许用户安装，卸载或为多个数据库更新服务器端对象。

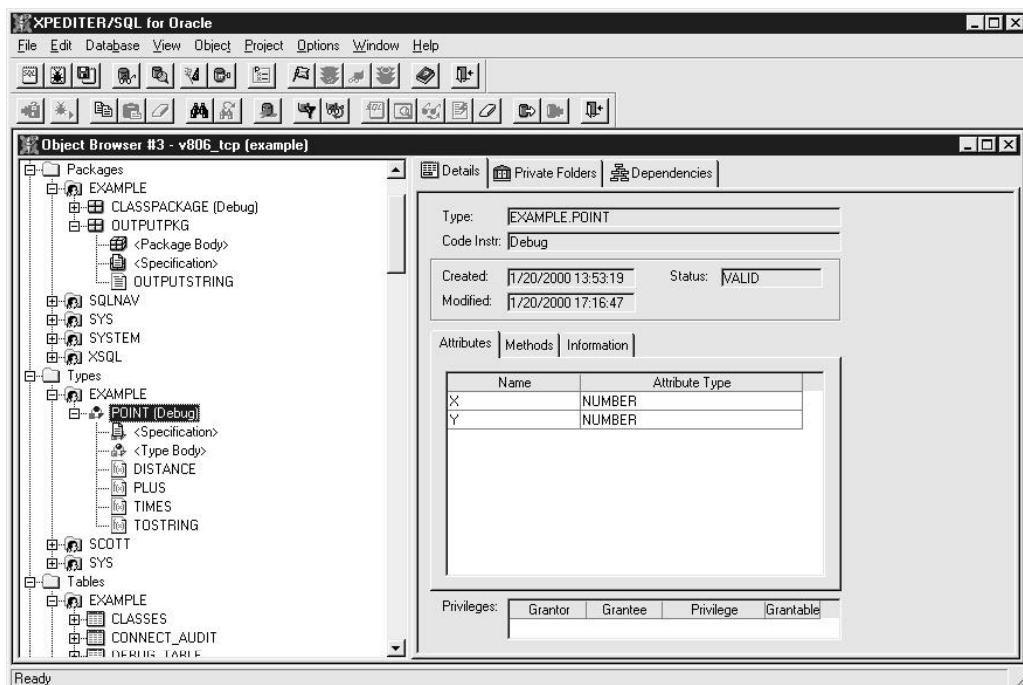


图2-8 浏览对象Point类型的窗口

3. 浏览数据库对象

XPEDITER/SQL的对象浏览器窗口是用来观察数据库对象信息和对话特权的专用窗口。XPEDITER/SQL允许用户为同一个数据库连接打开多个浏览器，但每个浏览器只能显示一个连接的信息。对象浏览器的窗口如图 2-8所示。该窗口的左窗格显示了对象的树形目录，用户可以查看希望的对象的类型和拥有者。该窗口右面的窗格则显示选择的对象的有关信息，包括该对象是否带有经过编译的调试信息和其他相关信息。(用户可以通过选择菜单项 Object | Debug On 或 Objetc | Debug Off 来编译带有或不带有调试信息的对象。)

4. 编辑数据库对象

双击对象浏览器中的某个对象将会在 SQL的Notepad编辑器窗口中打开该对象文件。SQL的Notepad编辑器窗口可以用来编辑所有类型的数据库对象，及 PL/SQL存储过程和匿名块。例如，图2-9中所示的窗口显示了在 Notepad编辑器中对象 Point的类型说明。编辑器对 PL/SQL代码自动进行了格式设置。除此之外，单击该窗口中的红色的三角形按钮可以运行编辑器中的 SQL语句，如果单击窗口中的绿色三角形按钮可以对 SQL语句进行调试。

5. 显示错误

单击位于 SQL Notepad编辑器底部窗格中的 SQL Errors标签，用户可以查看编译后报告的对象错误信息。这时显示的是当前对象或块的错误信息。如果单击某个错误，该编辑器将把光标定位到发生错误所在过程的语句行上。CD-ROM中图CD2-10显示了过程TooMamyErrors的错误。除此之外，非法的对象也在对象浏览中被标明。

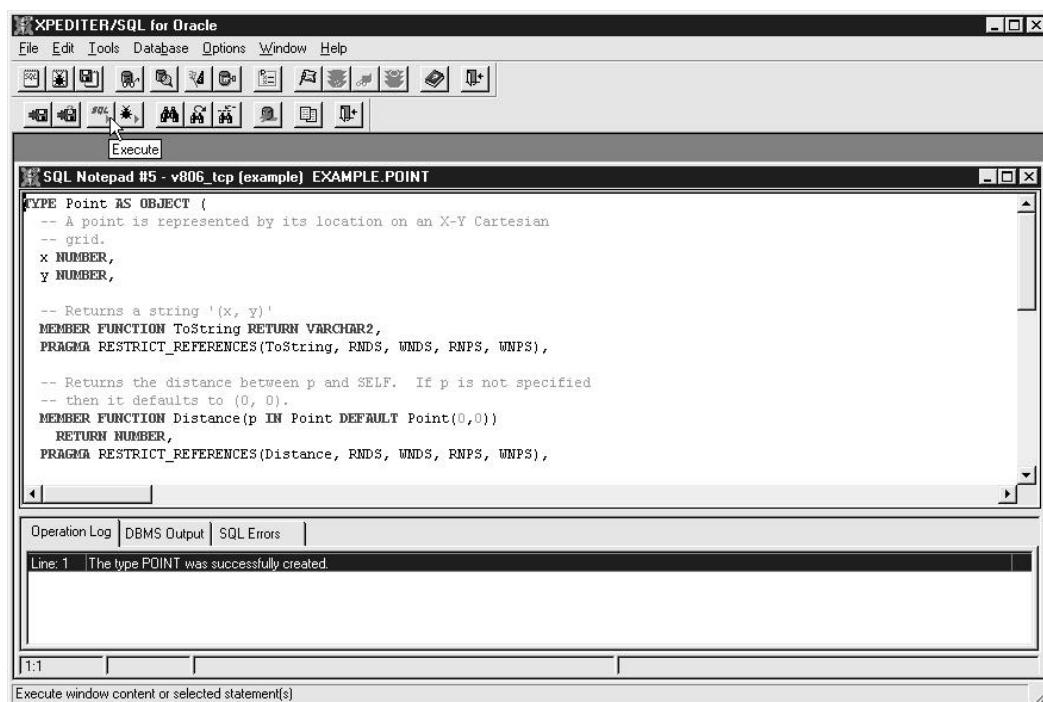


图2-9 编辑对象Point的窗口

6. 使用模板

XPEDITOR/SQL的另一个常用的工具是模板编辑器。该编辑窗口可以在 SQL Notepad编辑器窗口中通过选择Tools | Templates Editor子菜单激活。在该窗口中，用户可以把预定义的PL/SQL代码段插入到激活的Notepad中。除了可以使用创建新过程，函数，触发器，包和类型外的语法外，模板编辑器还包括对公用内置包（如DBMS_SQL，DBMS_OUTPUT等）的调用语句。本书CD-ROM中的图CD2-11有模板编辑器的图形窗口的样板。除此之外，在该窗口中双击某个模板将把该模板的文本复制到Notepad窗口中。

7. DBPartner

我们讨论的XPEDITOR/SQL的最后一个功能是工具DBPartner。该实用工具允许用户从任何程序中捕捉SQL语句并在XPEDITOR/SQL中对其进行编辑和调试。利用这种功能，用户可以在没有源程序的情况下调整或调试程序。

2.2.4 SQL Navigator

SQL Navigator是由Quest Software公司提供的开发工具。它也提供了图形用户界面，其主要功能如下：

- 自动格式PL/SQL和SQL语句
- 提供PL/SQL调试器
- 数据库浏览器
- 支持Oracle8对象类型和Oracle8i类型
- 代码模板
- 支持第三方版本控制系统

1. 连接数据库

与上述XPEDITOR/SQL开发工具一样，SQL Navigator在其第一次启动运行时也要请求用户建立数据库连接。SQL Navigator可以自动保存连接配置文件，但用户的口令不能与该文件一起保存。用来建立数据库连接的程序窗口如CD-ROM中图CD2-12所示。如果用户在初次启动时没有建立数据库连接，该工具将在一个编辑窗口或浏览窗口中用同一个对话框请求用户建立连接。SQL Navigator支持同时对不同数据库的多连接操作。

2. 安装服务器端的对象

SQL Navigator的多个选择项要求在服务器中创建一个名为SQLNAV的用户。CD-ROM中图CD2-13所示的服务器端安装向导可以帮助建立必要的用户和对象。该向导可以作为安装程序的一部分运行，也可以在安装完成后，选择菜单项Tools | Server Side Installation Wizard启动该程序运行。服务器端的安装是实现SQL Navigator的程序员组编程、第三方版本控制、SQL Navigator Tuner等功能的必要步骤。

3. 浏览数据库对象

SQL Navigator的DB浏览器窗口是用来浏览数据库对象的工具。用户可以从该窗口提供的树形视图中选择要查看的对象的类型和所有者。该浏览器的一个独特的功能是提供了过滤器。使用过滤器，用户可以选择希望显示的那些对象类型。该浏览器提供了多个预定义的过滤器，

用户也可以创建自己的过滤器。图 2-10 所示的 DB 浏览器窗口中使用了三个过滤器。

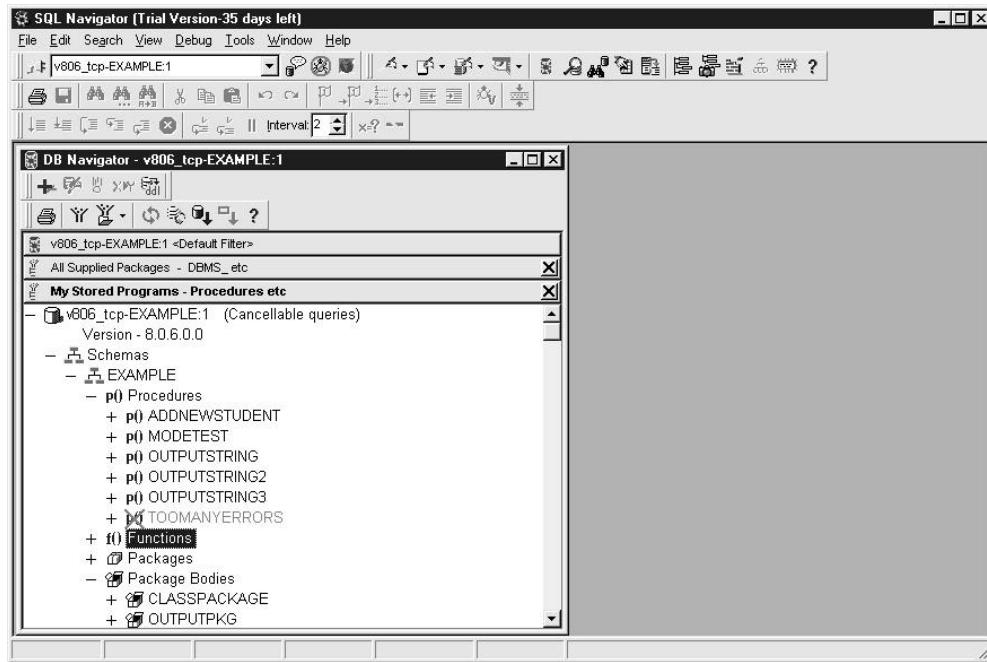


图2-10 带有三个过滤器的DB 浏览器窗口

如果用户建立了与 Oracle8i 数据库的连接，SQL Navigator 还在其数据库浏览器中支持 Oracle8i Java 存储过程 (JSP)。这时，在浏览器的树形目录中将增加有关 Java 类、Java 数据源和资源的目录入口。虽然用户不能直接在 SQL Navigator 中对 Java 代码进行编辑，但可以对已在数据库中的 Java 数据源进行编译。有关在 Oracle8i 中使用 JSP 的方法，请看本书第 10 章的内容。

4. 编辑数据库对象

SQL Navigator 提供了三种编辑窗口：

- SQL 编辑器，该编辑器只能编辑一个 SQL 语句或脚本。
- 触发器编辑器，该编辑器用来创建或编辑数据库触发器。除此之外，该编辑器还支持 Oracle8 的 Instead-of 触发器类型。
- 存储程序编辑器，该编辑器用来创建或编辑存储的 PL/SQL 代码。

当用户单击 DB 浏览器中的对象时，相应的编辑器就会开始运行。用户也可以使用 SQL Navigator 的工具条直接打开各种编辑器。图 2-11 显示了在存储程序编辑器中的函数 AlmostFull。该窗口突出显示了该函数中的 PL/SQL 代码。

5. 显示错误

如果在编译 PL/SQL 对象的过程中出现错误的话，这些错误将在用户在存储程序编辑器中编译对象时显示出来。单击突出显示错误的源程序行，以及双击一个错误时都将启动一个来自于 Oracle 文档资料中指示错误原因和方式的窗口。CD-ROM 中的图 CD2-14 显示了过程 TooManyErrors 的错误信息。

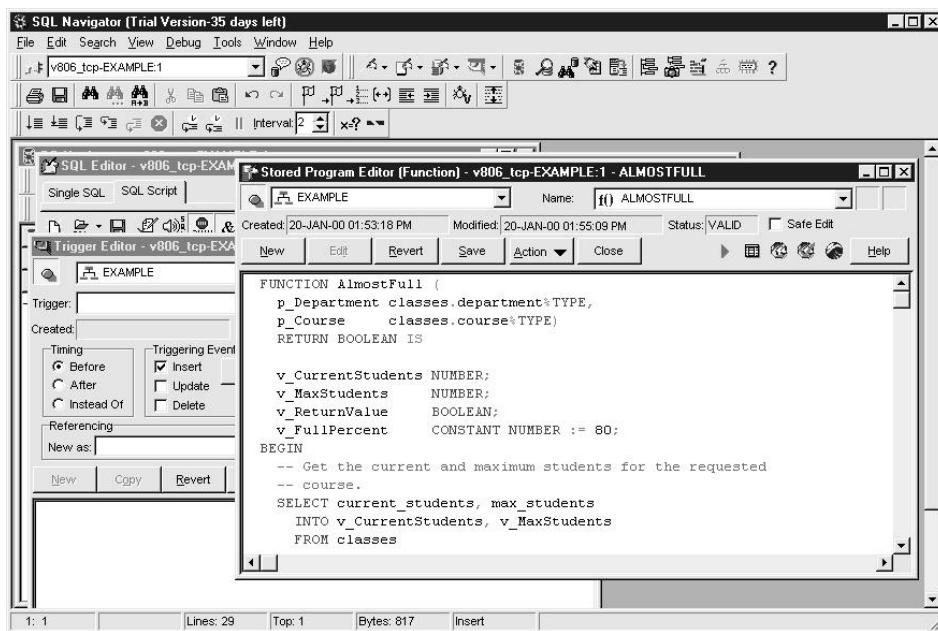


图2-11 编辑函数AlmostFull的窗口

6. 代码模板

可以在SQL Navigator工具菜单中启动的代码助理（Code Assistant）提供了PL/SQL和SQL结构中通用库。加亮显示某个结构将在代码助理信息窗口显示该结构的描述信息，双击该描述区将把该信息复制到现行的编辑窗口中并可由用户进行定制。代码助理的窗口图形显示在 CD-ROM中图CD2-15中。

2.2.5 TOAD

TOAD是Oracle应用开发者工具的缩写。该工具最早是从SQL Navigator分离出来的。现在，该工具是由Quest Software和SQL Navigator共同开发销售。这样做的结果导致这两个工具在某些方面（如许可机制）具有相同的功能，但是，我们马上也会看到它们之间的不同。TOAD提供了下述功能：

- 自动格式PL/SQL和SQL语句
- 提供PL/SQL调试器
- 数据库浏览器
- 支持Oracle8对象类型
- 代码模板
- 支持第三方版本控制系统

TOAD是一种功能完善的轻量级开发工具。该工具所需的磁盘和内存空间都比其他工具要小的多。

1. 连接数据库

TOAD可支持多数据库连接。当该应用首次启动运行时，它使用一个对话框提示用户建立数

下载

数据库连接（该对话框在CD-ROM中的图CD2-16中）。如果需要建立更多的连接的话，用户可以通过菜单命令File | New Connection来实现。一旦建立了连接，该连接就将保持到用户通过执行菜单命令File | Close Connection来将其关闭为止。连接使用的口令可以存储在连接配置文件中。

TOAD管理连接的一种特殊功能是与任何窗口关联的连接可以动态地进行变更。该功能使应用程序可以工作在多数据库环境下时将其他不用的窗口最小化。但对于一个给定的工作窗口来说，它只能与一个对话相关联。

2. 浏览数据库对象

TOAD提供了两种不同的数据库浏览器，它们是模式浏览器（Schema Browser）和对象浏览器（Object Browser）。图2-12所示的模式浏览器允许用户选择Oracle7类型的表、过程和包。该浏览器提供的目录结构与我们在上面介绍的其他三种开发工具所提供的树形目录结构不同，模式浏览器中有选择对象类型的标签，这些标签显示在窗口中左面的窗格中。该窗口的右窗格显示对象的详细内容，用户可以从该浏览器中编译或删除对象。

对象浏览器只能用来查看Oracle8的对象类型和类型体。类似于模式浏览器，对象浏览器允许用户查看和修改对象类型和类型体的属性。显示对象 Point的对象浏览器的图形窗口在CD-ROM中的图CD2-17中。

3. 编辑数据库对象

TOAD提供了两种编辑窗口：SQL编辑窗口和存储过程编辑窗口。正如这两个窗口的名称所表示的含意那样，SQL编辑窗口是用于编辑单个SQL语句或SQL脚本的，而存储过程编辑窗口则用来编辑存储过程、函数、包和触发器。在存储过程编辑窗口下，用户可以编译，运行，或调试过程。图2-13所示的是在存储过程编辑窗口中显示的过程OutputString，该过程是一个用C语言

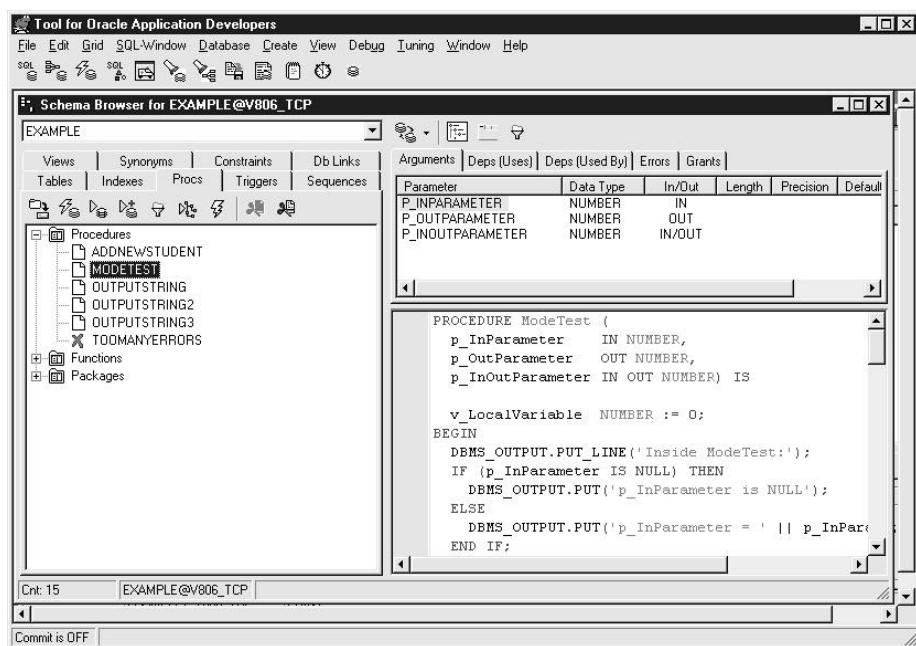


图2-12 模式浏览器窗口

言实现的外部过程。存储过程编辑窗口可以从数据库对象或文件中载入并用来创建新的对象。

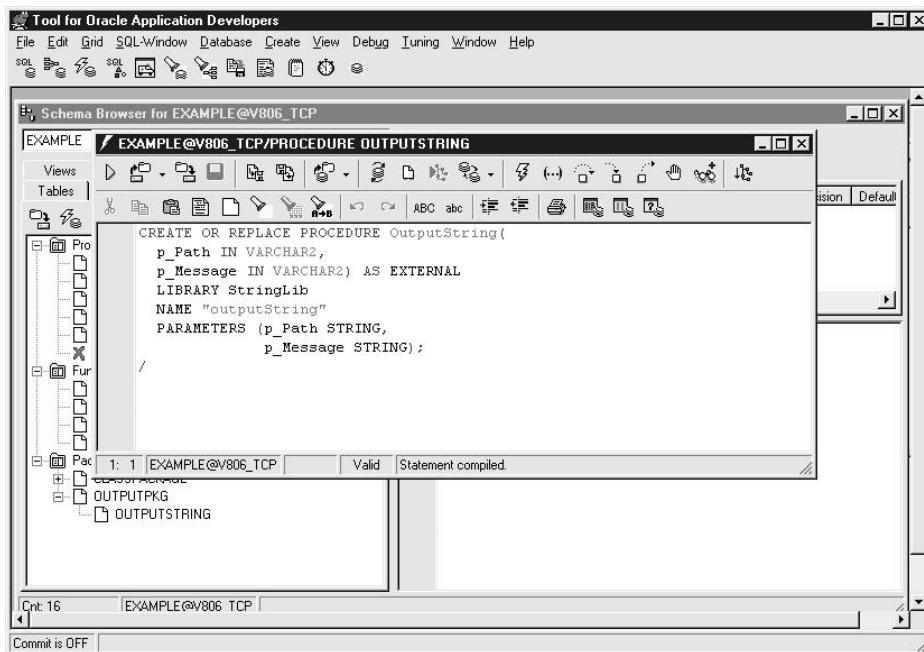


图2-13 存储过程编辑窗口

4. 显示错误信息

如果在编译过程中出现了编译错误，这些错误将显示在存储过程编辑窗口中下面的错误显示窗格中。当使用蓝色三角形按钮单击错误时，与产生这些错误有关的源程序行将突出显示，本书CD-ROM中的图CD2-18演示了这种情况。除此之外，也可以使用模式浏览器来查看非法对象的错误。

5. 代码模板

TOAD支持通用PL/SQL和SQL结构使用的代码模板。在该代码模板下，可以不用输入所需的结构，用户只要使用键盘快捷键就可以实现。其具体操作是，先在编辑窗口中输入键盘快捷键，然后再按CTRL-SPACEBAR，这时该快捷键将被一个完整的结构替代。代码模板可以在编辑选择项窗口中查看，如CD-ROM中的图CD2-19所示。除此之外，用户还可以编辑现存的模板，或加入自己建立的模板。

2.2.6 SQL-Programmer

最后一个介绍的图形用户界面开发工具是SQL-Programmer，该工具是由Sylvain Faust International开发的。它支持下列功能：

- 自动格式PL/SQL和SQL语句
- 提供PL/SQL调试器
- 数据库浏览器

下载

- 支持Oracle8对象类型
- 代码模板
- 支持第三方版本控制系统
- 数据库对象脚本编制

1. 连接数据库

SQL-Programmer支持对不同数据库的多点同时连接。CD-ROM中的图CD2-20是连接对话框的窗口显示。虽然连接口令不能存储在连接配置文件中，但系统可以为不同的服务器存储不同的连接配置文件。连接对话框还可以显示已打开的连接。

2. 浏览数据库对象

SQL-Programmer的SQL浏览窗口提供了查看数据库对象的功能，该窗口中显示的对象按模式和对象类型进行排序。如果当前有一个以上的连接处于活动状态的话，则单个SQL浏览窗口就可以浏览几个连接中的对象。图2-14中的窗口演示了正在显示函数AlmostFull的SQL浏览窗口。从该浏览器中，用户可以观察对象的属性，编译对象，或者将对象删除。

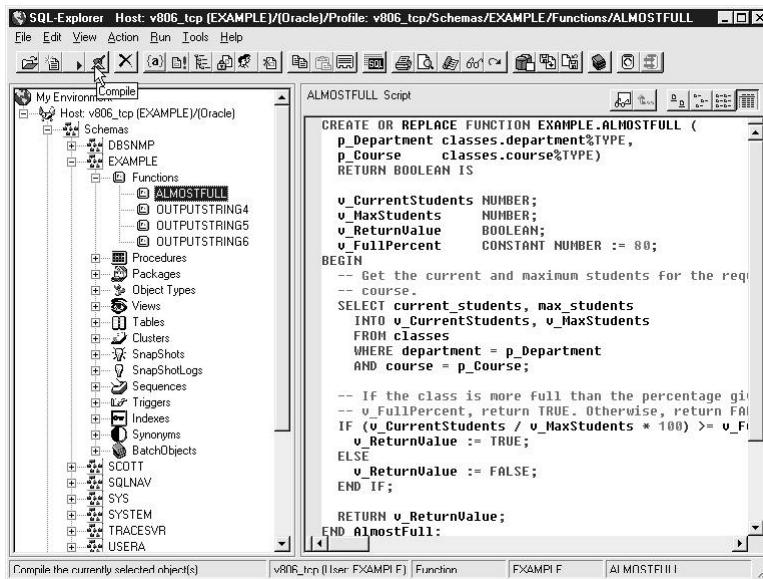


图2-14 SQL浏览窗口

3. 编辑数据库对象

双击SQL浏览窗口中显示的某个对象将启动SQL-Programmer的开发窗口（SPDW）对其进行编辑。SPDW提供查询或修改所显示对象信息的功能。图2-15是显示函数AlmostFull的SPDW窗口，如图所示，该窗口中还显示了变量。用户可以直接从SPDW中执行过程，这时，该窗口将显示对话框等待用户输入匿名块中使用的输入参数的值。

4. 显示错误信息

如果过程在编译出现错误的话，SPDW将在文本窗口下的错误窗格中显示这些错误。CD-ROM中图CD2-21就是演示SPDW该功能的图形窗口。双击该错误窗格中的某个错误将使与该错

误有关的代码段突出显示。同时，SPDW还将过程标识为非法。

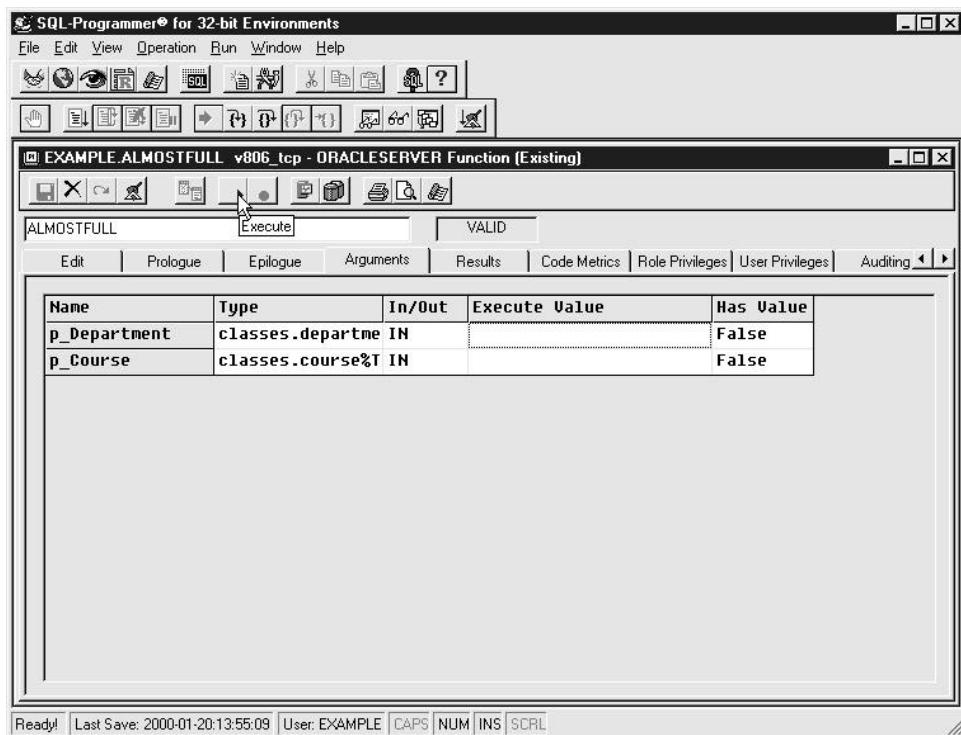


图2-15 显示AlmostFull的SPDW窗口

5. 代码模板

SQL-Programmer可以为创建新的对象提供模板。当用户在SQL浏览窗口创建新的对象时，该浏览器将自动为创建的对象填充相应的模板。用户可以在窗口中的选择菜单中指定模板。CD-ROM中的图CD2-22显示了过程使用的默认模板。在该窗口中，除了过程代码外还有代码注解。

6. 脚本

SQL-Programmer的脚本界面允许用户为任何数据库对象的组合创建脚本。这种脚本中包括了可以自动地重建任何其他服务器所需的代码。因此，这种脚本提供了一种可以在数据库之间复制单独的对象，或全部模式的机制。CD-ROM中图CD2-23显示了这种脚本的窗口界面。

2.2.7 PL/SQL开发工具小结

到目前为止，我们所讨论的所有开发工具都没有提供客户端的PL/SQL引擎。这些开发工具都可以用于开发和调试PL/SQL应用。表2-3给出了这些工具之间主要性能的比较。有关这些开发工具的详细信息，请看本书CD-ROM中提供的联机帮助和访问本章表2-1中列出的开发商站点。

支持版本控制的开发工具通常借助于第三方数据源控制系统实现开发功能。不同的开发工具支持不同的数据源控制系统。有关进一步的信息，请参阅CD-ROM中的联机帮助。

表2-3 PL/SQL开发工具性能比较

功能	SQL*PLUS	Rapid SQL	XPEDITOR/SQL	SQL Navigator	TOAD	SQL-Programmer
提供方式	随Oracle提供	单独出售	单独出售	单独出售	单独出售	单独出售
GUI界面环境	无	有	有	有	有	有
对象浏览器	无	有	有	有	有	有
代码模板	无	有	有	有	有	有
工程管理	无	有	有	无	无	无
代码格式化	无	有	有	有	有	有
作业调度	无	有	无	无	无	无
版本控制	无	有	有	有	有	有
支持Oracle8 类型	有	有	有	有	有	有
支持Oracle8i 类型	有	有	无	有	无	无
安装服务器端要求	无	无	有	有	无	无
支持同时多个连接	无	有	无	有	有	有

2.3 小结

我们在本章介绍了六个用于创建PL/SQL应用的开发工具。它们分别是Oracle的SQL *Plus，Embarcadero Technologies公司的Rapid SQL，Compuware公司的XPEDITOR/SQL，Qest Software公司的TOAD和SQL Navigator以及Sylvain-Faust International的SQL-Programmer。这些工具中，除了SQL *Plus是随Oracle数据库一起提供的外，其他工具的试用版程序都在本书的CD-ROM中。我们在下一章将介绍这些开发工具提供的调试功能。

第3章 跟踪和调试

没有几个程序能够不经过调试就可以实现其设计功能。这是因为除了程序本身不可能一次就可以完全书写正确外，而且在开发过程中，对程序的要求也可能发生变化，需要对程序进行重新编制。综上所述，程序需要经过彻底的测试才可以正常工作并实现预期的要求。我们将在本章讨论几种调试和测试PL/SQL程序的技术，其中即包括图形和非图形的PL/SQL调试技术。除此之外，我们还要讨论Oracle8i提供的跟踪和配置工具。

3.1 问题分析

每个程序错误都有其特殊之处，这就使得程序的调试和测试技术面临挑战。虽然，我们在开发过程中可以借助于测试和质量分析（QA）来减少程序错误的数量，但是，如果你具有使用开发工具的经验，毫无疑问，你将会在自己或其他人编制的程序中发现新的错误和问题。

3.1.1 调试指导原则

尽管每个程序错误都是不同的，并且对于每个给定错误的修改方法也有多种形式，但发现和修改错误的过程是可以确切定义的。经过前几年作者在调试自己和其他程序员编制的程序中所获得的经验，我总结了几条指导确定程序错误的原则。这些调试指导原则适用于所有的程序设计语言，而不仅仅是适用于PL/SQL。

1. 发现错误发生的位置

发现错误发生的位置是修改代码问题的关键一步。如果一个大型的复杂程序一开始就不能运行，诊断问题的第一步就是确认该程序出现错误的准确位置。实现错误定位的复杂程度取决于程序代码的复杂性。发现错误发生点的最简单方法是在程序运行时进行动态跟踪，查看数据结构的值以确定发生错误的原因。

2. 判定错误原因

一旦知道了程序中错误发生的位置，我们就需要进一步来判定发生错误的原因，是否是返回的Oracle错误？或是程序中计算部分返回了错误结果？还是把错误数据插入到数据库引发的错误？总之，为了修改错误，必须要搞清错误发生的原因。

3. 简化程序以便调试

当我们不能确认错误发生的位置时，一种有效的方法是把程序简化为简单的可测试的程序段。其方法是先删去程序的一部分代码，然后再返回到程序运行。如果错误还存在的话，就说明临时删除的程序部分不会导致错误。如果错误没有出现就要检查所删除的程序段中出现的错误。

请记住程序中某段代码的错误可能会显现在程序中的另一部分。例如，某个过程可能会返回一个不正确的值，但是该返回值可能在当前的程序中没有使用，而在其后的主程序段才会影

响程序执行。虽然问题看起来出现在主程序中，但是实际上是该过程的代码错误。如果我们将该过程调用去掉并直接把正确的返回值赋值给返回变量将揭示问题所在。我们在本章的后几节讨论这种特殊的情况。

4. 建立测试环境

在理想的情况下，测试和调试都不应在实用系统中进行，最好是通过尽可能多地复制实用系统来维持一个测试环境，例如，可以使用带有少量数据的同一个应用的数据库结构来进行测试。通过这种方法，在不影响正在运行的应用系统的前提下来开发测试应用的新版本。如果在应用系统中出现了问题，首先应在测试中尽可能地重现该问题。其方法就是把问题缩小在一个较小的测试案例中。测试案例可能不仅仅只包括程序代码，通常 PL/SQL语句的执行需要使用数据库结构和表中的数据，因此，与代码相关的部分也要复制并缩小。

3.1.2 调试程序包

PL/SQL的主要功能是处理存储在 Oracle数据库中的数据。该语言的结构就是基于数据处理的并具有优良的性能。但是，对于某些特殊的用途，我们需要一些工具来帮助我们编制和调试程序。

在下面的几节中，我们将详细分析调试 PL/SQL代码的不同方法。每一节都将集中讨论一种不同的程序错误并遵循上面的调试指导原则使用不同的方法来将程序问题孤立出来解决。每一节中将先描述通用的调试方法，然后给出要解决的问题的描述。我们将同时讨论非图形和图形调试技术。在讨论解决非图形问题的内容中，我们将开发不同版本的调试程序包， Debug，该包将可以用于程序员自己程序的调试。根据每个程序员所使用的环境和要求，本章提供的每个程序包将具有不同的用途。

3.2 非图形调试技术

尽管市场上有多种用于 PL/SQL的图形调试器（我们将在本章的后面讨论这些调试器），但是在很多的情况下简单的基于字符的调试器还是很有用的。基于图形用户界面的调试工具并不是随处可用的，并且在某些复杂的 PL/SQL运行环境下，这种基于图形用户界面的调试工具可能无法进行安装。我们在本章要讨论的两种简单的调试技术，即插入调试表和在屏幕上打印数据是最简单并且又不需要专门调试工具的实用技术。

3.2.1 在程序中插入调试用表

最简单的调试方法是把局部变量的值插入到程序维持的临时表中，当该程序运行结束时，我们可以查询该临时表中的变量数据值。这种调试方法实现起来最容易并且不需特定的运行环境，其原因是我们在程序中插入 INSERT语句。

1. 问题1

假设我们要编制一个根据当前已注册学生计算并返回每个班级平均分数值的函数。该函数的代码如下：

节选自在线代码AverageGrade1.sql

```
CREATE OR REPLACE FUNCTION AverageGrade (
/* Determines the average grade for the class specified. Grades are
 stored in the registered_students table as single characters
 A through E. This function will return the average grade, again,
 as a single letter. If there are no students registered for
 the class, an error is raised. */
p_Department IN VARCHAR2,
p_Course IN NUMBER) RETURN VARCHAR2 AS

v_AverageGrade VARCHAR2(1);
v_NumericGrade NUMBER;
v_NumberStudents NUMBER;

CURSOR c_Grades IS
    SELECT grade
        FROM registered_students
       WHERE department = p_Department
         AND course = p_Course;
BEGIN
/* First we need to see how many students there are for
 this class. If there aren't any, we need to raise an error. */
SELECT COUNT(*)
    INTO v_NumberStudents
    FROM registered_students
   WHERE department = p_Department
     AND course = p_Course;

IF v_NumberStudents = 0 THEN
    RAISE_APPLICATION_ERROR(-20001, 'No students registered for ' ||
p_Department || ' ' || p_Course);
END IF;

/* Since grades are stored as letters, we can't use the AVG
 function directly on them. Instead, we can use the DECODE
 function to convert the letter grades to numeric values,
 and take the average of those. */
SELECT AVG(DECODE(grade, 'A', 5,
                   'B', 4,
                   'C', 3,
                   'D', 2,
                   'E', 1))
    INTO v_NumericGrade
    FROM registered_students
   WHERE department = p_Department
     AND course = p_Course;

/* v_NumericGrade now contains the average grade, as a number from
 1 to 5. We need to convert this back into a letter. The DECODE
 function can be used here as well. Note that we are selecting
 the result into v_AverageGrade rather than assigning to it,
```

```

because the DECODE function is only legal in a SQL statement. */
SELECT DECODE(ROUND(v_NumericGrade),5, 'A',
               4, 'B',
               3, 'C',
               2, 'D',
               1, 'E')

INTO v_AverageGrade
FROM dual;

RETURN v_AverageGrade;
END AverageGrade;

```

假设表registered_students的内容如下：

```

SQL> select * from registered_students;
STUDENT_ID DEP COURSE G
-----
10000CS    102 A
10002 CS   102 B
10003 CS   102 C
10000 HIS  101 A
10001 HIS  101 B
10002 HIS  101 B
10003 HIS  101 A
10004 HIS  101 C
10005 HIS  101 C
10006 HIS  101 E
10007 HIS  101 B
10008 HIS  101 A
10009 HIS  101 D
10010 HIS 101 A
10008 NUT  307 A
10010 NUT  307 A
10009 MUS  410 B
10006 MUS  410 E
10011 MUS  410 B
10000 MUS  410 B
20 rows selected.

```

注意 表registered_students由联机代码relTables.sql脚本中上面的20行数据填充。有关该表的结构情况，请看第1章的内容。

该表中已有四个班级学生进行了注册。它们分别是计算机科学 102、历史 101、营养 307 和音乐 410。现在我们可以用这四个班级来调用函数AverageGrade。如果我们使用了其他的班级，该函数将会引发“无学生注册”的错误。下面是开发工具SQL*Plus的输出样本：

节选自在线代码callAG.sql

```

SQL> VARIABLE v_AveGrade VARCHAR2(1)
SQL> exec :v_AveGrade := AverageGrade('HIS', 101)
PL/SQL procedure successfully completed.

```

```

SQL> print v_AveGrade
V_AVEGRADE
-----
B

SQL> exec :v_AveGrade := AverageGrade('NUT', 307)
PL/SQL procedure successfully completed.

SQL> print v_AveGrade
V_AVEGRADE
-----
A

SQL> exec :v_AveGrade := AverageGrade('MUS', 410)
PL/SQL procedure successfully completed.

SQL> print v_AveGrade
V_AVEGRADE
-----
C

SQL> exec :v_AveGrade := AverageGrade('CS', 102)
begin :v_AveGrade := AverageGrade('CS', 102); end;
*
ERROR at line 1:
ORA-20001: No students registered for CS 102
ORA-06512: at "EXAMPLE.AVERAGEGRADE", line 29

```

对该函数的最后一次调用产生了错误，返回了错误信息 ORA-2001，尽管我们会发现实际上计算机科学已经有学生注册。

2. 问题1的调试程序包

我们用来发现上述错误的调试程序如下所示。该程序中的过程 Debug.Debug 是该程序包的主过程，它带有两个参数，一个描述和一个变量。这两个参数是连接在一起的并被插入到表 debug_table 中存储。过程 Debug.Reset 要在程序的开始就调用执行，以便初始化表 debug_table 和内部行记数器（该行记数器过程也被包的初始化代码调用）。行记数器的作用是保证表 debug_table 中的行可以按它们插入时的顺序进行选择。

```

节选自在线代码Debug1.sql
CREATE OR REPLACE PACKAGE Debug AS
  /* First version of the debug package. This package works
   by inserting into the debug_table table. In order to see
   the output, select from debug_table in SQL*Plus with:
  SELECT debug_str FROM debug_table ORDER BY linecount; */

  /* This is the main debug procedure. p_Description will be
   concatenated with p_Value, and inserted into debug_table. */

```

```
PROCEDURE Debug(p_Description IN VARCHAR2, p_Value IN VARCHAR2);

/* Resets the Debug environment. Reset is called when the
package is instantiated for the first time, and should be
called to delete the contents of debug_table for a new
session. */
PROCEDURE Reset;
END Debug;

CREATE OR REPLACE PACKAGE BODY Debug AS
/* v_LineCount is used to order the rows in debug_table. */
v_LineCount NUMBER;

PROCEDURE Debug(p_Description IN VARCHAR2, p_Value IN VARCHAR2) IS
BEGIN
INSERT INTO debug_table (linecount, debug_str)
VALUES (v_LineCount, p_Description || ':' || p_Value);
COMMIT;
v_LineCount := v_LineCount + 1;
END Debug;

PROCEDURE Reset IS
BEGIN
v_LineCount := 1;
DELETE FROM debug_table;
END Reset;

BEGIN /* Package initialization code */
Reset;
END Debug;
```

注意 表debug_table是用脚本relTables.sql中的下列语句创建的：

```
CREATE TABLE debug_table (
linecount NUMBER PRIMARY KEY,
debug_str VARCHAR2(200));
```

3. 使用问题1的调试程序包

为了发现程序AverageGrade中的错误，我们需要观察该过程使用变量的值的情况。我们可以在该程序中加入调试语句来实现。为了使用上面的调试程序包，我们需要在程序AverageGrade的开始处调用过程Debug.Reset，并在凡是我们要查看变量的地方调用过程Debug.Debug。下面是已经加入调试语句的AverageGrade程序。为了便于查看程序，我们删除了该程序的某些注释行。

节选自在线代码AverageGrade2.sql

```
CREATE OR REPLACE FUNCTION AverageGrade (
p_Department IN VARCHAR2,
p_Course IN NUMBER) RETURN VARCHAR2 AS
```

```
v_AverageGrade VARCHAR2(1);
v_NumericGrade NUMBER;
v_NumberStudents NUMBER;

CURSOR c_Grades IS
  SELECT grade
    FROM registered_students
   WHERE department = p_Department
     AND course = p_Course;

BEGIN
  Debug.Reset;
  Debug.Debug('p_Department', p_Department);
  Debug.Debug('p_Course', p_Course);

/* First we need to see how many students there are for
   this class. If there aren't any, we need to raise an
   error. */
  SELECT COUNT(*)
    INTO v_NumberStudents
   FROM registered_students
  WHERE department = p_Department
    AND course = p_Course;

  Debug.Debug('After select, v_NumberStudents', v_NumberStudents);

  IF v_NumberStudents = 0 THEN
    RAISE_APPLICATION_ERROR(-20001, 'No students registered for ' ||
      p_Department || ' ' || p_Course);
  END IF;

  SELECT AVG(DECODE(grade, 'A', 5,
                     'B', 4,
                     'C', 3,
                     'D', 2,
                     'E', 1))
    INTO v_NumericGrade
   FROM registered_students
  WHERE department = p_Department
    AND course = p_Course;

  SELECT DECODE(ROUND(v_NumericGrade), 5, 'A',
                 4, 'B',
                 3, 'C',
                 2, 'D',
                 1, 'E')
    INTO v_AverageGrade
   FROM dual;
```

```
    RETURN v_AverageGrade;
END AverageGrade;
```

现在我们可以再次调用程序 AverageGrade 并从表 debug_table 中选择下列的结果来观察：

```
SQL> EXEC :v_AveGrade := AverageGrade('CS', 102)
begin :v_AveGrade := AverageGrade('CS', 102); end;
*
ERROR at line 1:
ORA-20001: No students registered for CS 102
ORA-06512: at "EXAMPLE.AVERAGEGRADE", line 25
ORA-06512: at line 1
```

```
SQL> SELECT debug_str FROM debug_table ORDER BY linecount;
DEBUG_STR
-----
p_Department: CS
p_Course: 102
After select, v_NumberStudents: 0
```

从上面的代码中，我们可以看到变量 v_NumberStudents 的值是 0，这就解释了为什么程序提示出现 ORA-20001 错误的原因。这样一来，我们就可以把错误的范围缩小到 SELECT 语句来检查，因为 SELECT 语句在没有与表中的任何行相匹配时会返回 0。现在我们可以进一步来详细检查下面所示的 SELECT 语句中的 WHERE 子句是否有问题：

```
SELECT COUNT(*)
  INTO v_NumberStudents
  FROM registered_students
 WHERE department = p_Department
   AND course = p_Course;
```

根据调试程序的输出结果来看，变量 p_Department 和 p_Course 的值似乎没有问题。但实际上，在这些变量的字符串的最后可能存在空格字符，因此使看到的字符串与实际的字符串不符。现在让我们来调用 Debug.Debug 把变量 p_Department 和 p_Course 用引号扩号起来。这样一来我们就可以发现是否在这些变量的前面或后面存在着空格。

节选自在线代码 AverageGrade3.sql

```
CREATE OR REPLACE FUNCTION AverageGrade
  ...
BEGIN
  Debug.Reset;
  Debug.Debug('p_Department', ' ' || p_Department || ' ');
  Debug.Debug('p_Course', ' ' || p_Course || ' ');

  /* First we need to see how many students there are for
   * this class. If there aren't any, we need to raise an error. */
  SELECT COUNT(*)
    INTO v_NumberStudents
    FROM registered_students
   WHERE department = p_Department
```

```

AND course = p_Course;

Debug.Debug('After select, v_NumberStudents', v_NumberStudents);
...

```

现在当我们再次运行AverageGrade并查询表debug_table时，我们就可以得到下面的结果：

```

SQL> EXEC :v_AveGrade := AverageGrade('CS', 102)
begin :v_AveGrade := AverageGrade('CS', 102); end;
*
ERROR at line 1:
ORA-20001: No students registered for CS 102
ORA-06512: at "EXAMPLE.AVERAGEGRADE", line 25
ORA-06512: at line 1

```

```

SQL> SELECT debug_str FROM debug_table ORDER BY linecount;
DEBUG_STR
-----
p_Department: 'CS'
p_Course: '102'
After select, v_NumberStudents: 0

```

我们可以看到，变量p_NumberStudents的尾部没有多余的空格。这就是问题所在。表registered_students的列department定义为CHAR(3)，但变量p_Departement是VARCHAR2。这就意味着带有‘CS’的数据库列后面有一个空格，从而也说明了为什么SELECT语句没有发现匹配的数据库行，因此给变量p_NumberStudents赋值为0的原因。

提示 内置函数DUMP可以用来检查数据库列的准确内容。例如，我们可以用下面的命令在表registered_students中确认列department的内容。

```

SQL> SELECT DISTINCT DUMP(department)
  2 FROM registered_students
  3 WHERE department = 'CS';

DUMP(DEPARTMENT)
-----
Typ=96 Len=3: 67,83,32

```

如上所示，该列的类型是96，代表CHAR类型。该列的最后一个字符是32，它就是ASCII码的空格（实际的二进制数值取决于数据库系统采用的字符集），这就说明该列是用空格填补的。有关数据类型代码请参阅Oracle SQL参考资料。

修改该错误的一种方法是把变量p_Department的类型改变为如下所示的CHAR：

```

CREATE OR REPLACE FUNCTION AverageGrade (
  p_Department IN CHAR,
  p_Course IN NUMBER) RETURN VARCHAR2 AS
  ...
BEGIN
  ...
END AverageGrade;

```

在做了上述修改后，程序 AverageGrade 就可以正确运行了：

```
SQL> EXEC :v_AveGrade := AverageGrade('CS', 102)
PL/SQL procedure successfully completed.
```

```
SQL> print v_AveGrade
V_AVEGRADE
-----
B
```

上述的修改能够起作用的原因就是在 WHERE 子句中的两个值都是 CHAR 类型并且使用了空格填充比较语法。

提示 如果我们在函数中使用了属性 %TYPE 的话，变量 p_Department 的类型就是正确的类型了，这就是我推荐使用 %TYPE 的原因。同样，由于程序 AverageGrade 的返回值是长度为 1 的字符串，并且永远为 1，所以我们可以将 RETURN 子句定义为定长类型的 CHAR。现在，程序 AverageGrade 的声明部分如下所示：

```
节选自在线代码AverageGrade4.sql
CREATE OR REPLACE FUNCTION AverageGrade (
    p_Department IN registered_students.department%TYPE,
    p_Course IN registered_students.course%TYPE) RETURN CHAR AS

    v_AverageGrade CHAR(1);
    v_NumericGrade NUMBER;
    v_NumberStudents NUMBER;

    ...
BEGIN
    ...
END AverageGrade;
```

4. 问题1解决方案的评价

该版本的 Debug 非常简单。虽然它的功能只是插入一个调试表 debug_table，但是我们可以借助于该程序发现程序 AverageGrade 中的错误。这种调试技术的主要优点如下：

- 由于该程序只使用 SQL，所以它可以在任何环境下使用。显示输出的 SELECT 语句可以在 SQL *Plus 或在其他的开发工具中运行。
- 由于调试程序 Debug 代码很少，所以它不会对正在调试的程序增加太多的开销。

但是这种调试技术也有下面的缺点：

- 程序 AverageGrade 在处理过程中引发了异常，这就使该程序中已执行的 SQL 语句返回到开始状态。因此，就要在程序 Debug.Debug 中使用命令 COMMIT 来保证插入调试表的操作不会返回到起点。这时，如果该过程中的其他处理不需要执行提交的话，该提交命令的执行将引起错误。同时，该提交命令也将会使所有打开的 SELECT FOR UPDATE 游标无效。（在 Oracle8i 中，Debug.Debug 可以编制为自动事务，从而避免上述问题。有关内容请看第

11章。)

- 按照现在的编制方法，如果程序中同时有一个以上的会话时，程序 Debug将不能正常工作。SELECT语句将从多个会话中返回结果。该问题可以通过在 Debug包和表debug_table 中加入唯一标识会话的数据列来解决。

我们将在下面的一节中通过使用 DBMS_OUTPUT包来解决Debug程序存在的问题。

3.2.2 将结果打印到屏幕

我们在上一节看到的 Debug程序的第一个版本初步实现了有限的 I/O操作(对数据库的I/O操作,而不是输出到屏幕)。PL/SQL没有提供内置的输入输出功能，这是因为 PL/SQL是一种专门对存储在数据库中的数据进行操作的专用语言,它不需要具有打印变量和数据结构的功能。然而,输入输出的确是非常有用的调试工具，所以从 PL/SQL2.0版开始,通过内置包DBMS_OUTPUT,扩充了输出功能。我们将在本节使用 DBMS_OUTPUT来实现Debug程序的第二个版本。

提示 PL/SQL直到现在仍然没有内置的输入功能,但SQL *Plus中的替换变量(我们在第2章使用过)可以用来解决该问题。PL/SQL2.3及以上的版本提供的包 UTL_FILE可以用来读写系统文件。本书的第 7章将介绍包 UTL_FILE和可以把输出写入到文件中的 Debug程序。

1. DBMS_OUTPUT包

在我们开始讨论本节的调试程序之前，我们需要对包 DBMS_OUTPUT做一些分析介绍。就象PL/SQL的其他包一样，DBMS_OUTPUT是属于Oracle user SYS的。创建DBMS_OUTPUT的脚本授权该包的 EXECUTE命令具有 PUBLIC的属性并为其创建了公用命令。这就是说任何 Oracle用户都可以不在该包名的前面冠以前缀 SYS 就直接调用DBMS_OUTPUT中的子程序。

DBMS_OUTPUT是如何工作的呢？该包的两个基本操作，GET和PUT是借助于该包中的过程实现的。PUT操作带有参数并将其放入内部缓冲区存储。GET操作执行从该缓冲区的读操作并把读入的内容作为参数返回给该过程。DBMS_OUTPUT还提供一个叫做ENABLE的过程用来设置缓冲区的容量。

该包中的PUT例程有PUT、PUT_LINE和NEW_LINE。GET例程有GET_LINE和GET_LINES。而程序ENABLE和DISABLE则用于控制缓冲区。

例程PUT和PUT-LINE PUT和PUT_LINE的调用语法如下：

```
PROCEDURE PUT( a VARCHAR2);
PROCEDURE PUT( a NUMBER);
PROCEDURE PUT( a DATE);

PROCEDURE PUT_LINE( a VARCHAR2);
PROCEDURE PUT_LINE( a NUMBER);
PROCEDURE PUT_LINE( a DATE);
```

命令中的字母a是存放在缓冲区中的参数。请注意上面的过程是由参数的类型重载的(我们在第4章讨论重载)。由于有三种不同版本的 PUT和PUT_LINE例程,缓冲区可以存储类型为

VARCHAR2, NUMBER和DATE的值。这些类型的数据都以其原始格式存储。

缓冲区是按行使用的，每一行最多可以存储 255个字节。PUT_LINE 把一个换行符追加到其参数的尾部。PUT命令则没有这种功能。PUT_LINE命令的操作等价于在执行PUT命令后再执行NEW_LINE操作。

例程NEW_LINE 调用NEW_LINE的语法是：

```
PROCEDURE NEW_LINE;
```

NEW_LINE把换行字符放入缓冲区中，标识一行的结束。缓冲区对其中的字符行数没有限制。缓冲区的大小是在ENABLE命令中的说明的。

例程GET_LINE 调用GET_LINE的语法是：

```
PROCEDURE GET_LINE(line OUT VARCHAR2, status OUT INTEGER);
```

其中line是缓冲区内一行的字符串，status指示line是否检索成功。一行的最大长度是 255个字符。如果检索到指定行的话，则status为0；如果缓冲区为空的话，则status为1。

注意 尽管缓冲区行的最大长度是 255个字符，但是输出变量行可以大于 255个字符。例如，缓冲区行可以由日期量组成，日期值占用 7个存储字节，但通常我们把日期量转换为长度大于7个字符的字符串存储。

例程GET_LINES 该过程有一个按表索引的参数。其表类型和该过程的语法如下：

```
TYPE CHARARR IS TABLE OF VARCHAR2(255)
INDEX BY BINARY_INTEGER;
PROCEDURE GET_LINES( lines OUT CHARARR,
                     numlines IN OUT INTEGER);
```

其中参数line是包括来自缓冲区的多行表索引，numlines指出所需的行数。当指定GET_LINES为输入时，numlines说明所需的行数。当为输出方式时，numlines为实际返回的行数，该行数要小于或等于指定的行数。GET_LINES相当于执行多次对GET_LINE的调用。

DBMS_OUIPUT中还定义了类型 CHARARR。如果要在自己的代码中显示地调用GET_LINES时，就必须声明一个类型为DBMS_OUIPUT.CHARARR的变量。例如：

节选自在线代码DBMSOutput.sql

```
DECLARE
  /* Demonstrates using PUT_LINE and GET_LINE. */
  v_Data      DBMS_OUTPUT.CHARARR;
  v_NumLines NUMBER;
BEGIN
  -- Enable the buffer first.
  DBMS_OUTPUT.ENABLE(1000000);

  -- Put some data in the buffer first, so GET_LINES will
  -- retrieve something.
  DBMS_OUTPUT.PUT_LINE('Line One');
  DBMS_OUTPUT.PUT_LINE('Line Two');
  DBMS_OUTPUT.PUT_LINE('Line Three');
```

```
-- Set the maximum number of lines that we want to retrieve.
v_NumLines := 3;

/* Get the contents of the buffer back. Note that v_Data is
   declared of type DBMS_OUTPUT.CHARARR, so that it matches
   the declaration of DBMS_OUTPUT.GET_LINES. */
DBMS_OUTPUT.GET_LINES(v_Data, v_NumLines);

/* Loop through the returned buffer, and insert the contents
   into temp_table. */
FOR v_Counter IN 1..v_NumLines LOOP
  INSERT INTO temp_table (char_col)
    VALUES (v_Data(v_Counter));
END LOOP;
END;
```

注意 GET_LINE和GET_LINES都只从缓冲区中进行检索并返回字符串。当执行 GET操作时，缓冲区的内容将根据默认的数据类型转换规则被转换为字符串。如果要为转换说明格式的话，使用显式地的 TO_CHAR来调用PUT，而不是调用GET。

过程ENABLE和DISABLE ENABLE和DISABLE的调用语法如下：

```
PROCEDURE ENABLE(buffer_size IN INTEGER DEFAULT 20000);
PROCEDURE DISABLE;
```

其中buffer_size是内部缓冲区初始字节数，默认的缓冲区是 20 000个字节，该缓冲区的最大容量是1 000 000个字节。执行该过程后，PUT和PUT_LINE的参数将被放入到该缓冲区中，这些参数是以内部格式存储的，其所占用的空间由这些参数的结构决定。如果调用 DISABLE的话，缓冲区的内容将被清除，对PUT和PUT_LINE的后续调用将无效。

2. 使用DBMS_OUTPUT

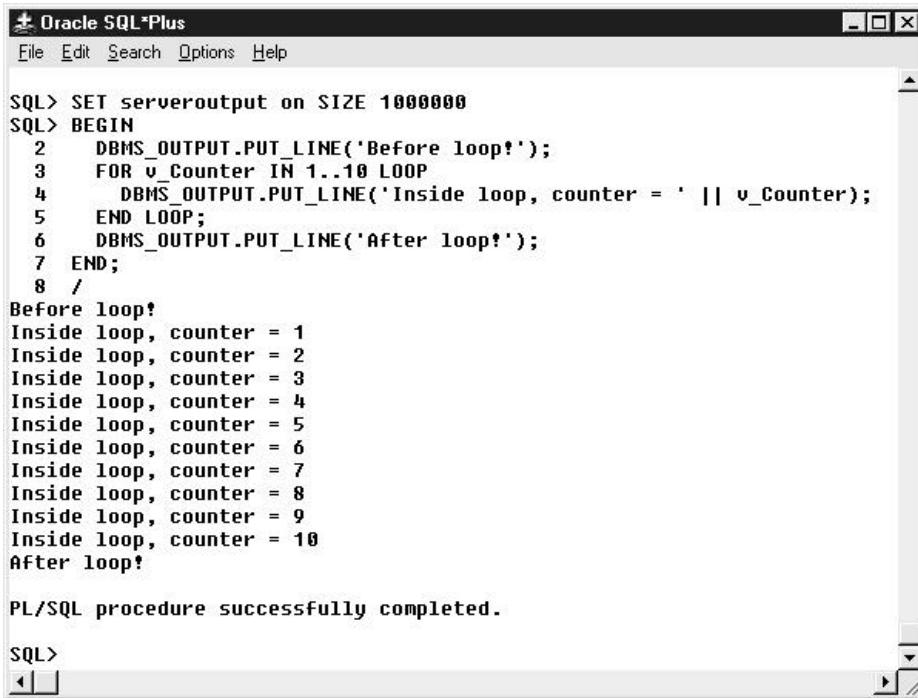
包DBMS_OUTPUT自身并不具有任何打印机制,该包只是简单地实现了先入先出（FIFO）的数据结构。但这样如何实现打印呢？SQL *Plus和服务器管理器都提供了叫做 SERVEROUTPUT的选择项。另外，许多第三方提供的开发工具（包括图形开发和调试工具）提供了显示DBMS_OUTPUT数据的功能。如果指定 SERVEROUTPUT选项，SQL *Plus将在PL/SQL块结束时自动调用DBMS_OUTPUT.GET_LINES并把结果打印到屏幕。图 3-1是打印窗口。

SQL *Plus的命令SET SERVEROUTPUT ON隐式地调用DBMS_OUTPUT.ENABLE来建立内部缓冲区。可以用下面的命令指定缓冲区的容量：

```
SET SERVEROUTPUT ON SIZE buffer_size
```

其中buffer_size用来作为缓冲区的初始长度（即DBMS_OUTPUT.ENABLE的参数）。在设置 SERVEROUTPUT ON时,SQL *Plus将在PL/SQL块结束运行后调用DBMS_OUTPUT. GET_LINES。这就意味着输出将在块结束时和该块不运行时送往屏幕显示。通常这样在DBMS_OUTPUT用于调试时，该功能不会带来其他问题。

警告 DBMS_OUTPUT是专门用于调试的，它不适于用来生成报表。如果程序员需要为查询指定输出格式的话，我们推荐使用专用工具 Oracle Reports，尽量不要用DBMS_DUTPUT和SQL*Plus。



The screenshot shows the Oracle SQL*Plus interface. The command line displays:

```
SQL> SET serveroutput on SIZE 1000000
SQL> BEGIN
 2   DBMS_OUTPUT.PUT_LINE('Before loop!');
 3   FOR v_Counter IN 1..10 LOOP
 4     DBMS_OUTPUT.PUT_LINE('Inside loop, counter = ' || v_Counter);
 5   END LOOP;
 6   DBMS_OUTPUT.PUT_LINE('After loop!');
 7 END;
 8 /
```

The output window shows the execution of the procedure:

```
Before loop!
Inside loop, counter = 1
Inside loop, counter = 2
Inside loop, counter = 3
Inside loop, counter = 4
Inside loop, counter = 5
Inside loop, counter = 6
Inside loop, counter = 7
Inside loop, counter = 8
Inside loop, counter = 9
Inside loop, counter = 10
After loop!
```

At the bottom, it says:

```
PL/SQL procedure successfully completed.
```

图3-1 使用SERVEROUTPUT和PUT_LINE选项时的输出窗口

内部缓冲区受到其最大容量限制（由 DBMS_OUTPUT.ENABLE说明），并且每一行的最大长度为 255个字节。因此，调用 DBMS_OUTPUT.PUT，DBMS_OUTPUT.PUT_LINE和DBMS_OUTPUT.NEW_LINE时可能会引发如下两种异常：

```
ORA-20000: ORU-10027: buffer overflow,
           limit of <buf_limit> bytes.
```

或

```
ORA-20000: ORU-10028: line length overflow,
           limit of 255 bytes per line.
```

错误信息的内容取决于所超出的限制类型。

提示 养成应用SET SERVEROUTPUT ON命令来说明缓冲区长度的习惯是非常好的。尽管DBMS_OUTPUT.ENABLE的默认值是20 000个字节，但是如果在SET SERVEROUTPUT ON中没有显式地说明缓冲区长度的话，SQL *Plus调用DBMS_OUTPUT.ENABLE时用的缓冲区长度将是2 000个字符。

3. 问题2

表students中有一个记录已注册学生的学分的列，表 registered_student则包含学生所在班级的信息。如果学生变更注册的话（因此表 registered_student中的信息要变更），我们也要更新表 students中的列 current_credits。我们可以编制一个叫做 CountCredits的函数来实现更新，该函数将计算注册学生的总学分。该函数如下：

```
节选自在线代码CountCredits1.sql
CREATE OR REPLACE FUNCTION CountCredits (
    /* Returns the number of credits for which the student
       identified by p_StudentID is currently registered */
    p_StudentID IN students.ID%TYPE)
RETURN NUMBER AS

v_TotalCredits NUMBER; -- Total number of credits
v_CourseCredits NUMBER; -- Credits for one course
CURSOR c_RegisteredCourses IS
    SELECT department, course
    FROM registered_students
    WHERE student_id = p_StudentID;

BEGIN
    FOR v_CourseRec IN c_RegisteredCourses LOOP
        -- Determine the credits for this class.
        SELECT num_credits
        INTO v_CourseCredits
        FROM classes
        WHERE department = v_CourseRec.department
        AND course = v_CourseRec.course;
        -- Add it to the total so far.
        v_TotalCredits := v_TotalCredits + v_CourseCredits;
    END LOOP;

    RETURN v_TotalCredits;
END CountCredits;
```

由于函数 CountCredits不修改数据库和包的状态，所以我们可以从 SQL语句中直接调用它（适用PL/SQL2.1及更高版本）。（SQL语句调用功能将在第5章详细讨论）这样，我们就可以通过表students的选择来得到所有学生的总学分。其命令如下：

```
SQL> SELECT ID, CountCredits(ID) "Total Credits"
  2 FROM students;
      ID Total Credits
-----
10000
10001
10002
10003
10004
10005
10006
10007
```

```

10008
10009
10010
10011
12 rows selected.

```

从上面的输出可以看出函数Count Credits没有输出结果，这就说明该函数的返回值为空，没有得到正确的处理结果。

4. 问题2的调试程序包

现在我们使用包DBMS_OUTPUT来定位函数CountCredits的错误。下面的程序段是一个解决该问题的调试程序，该程序具有与上一节使用的第一版调试程序相同的接口，因此我们只需要修改该包的包体就可以了。

节选自在线代码Debug2.sql

```

CREATE OR REPLACE PACKAGE BODY Debug AS
    PROCEDURE Debug(p_Description IN VARCHAR2,
                    p_Value IN VARCHAR2) IS
        BEGIN
            DBMS_OUTPUT.PUT_LINE(p_Description || ':' || p_Value);
        END Debug;

    PROCEDURE Reset IS
        BEGIN
            /* Disable the buffer first, then enable it with the
             * maximum size. Since DISABLE purges the buffer, this
             * ensures that we will have a fresh buffer whenever
             * Reset is called.
            */
            DBMS_OUTPUT.DISABLE;
            DBMS_OUTPUT.ENABLE(1000000);
        END Reset;

    BEGIN /* Package initialization code */
        Reset;
    END Debug;

```

在该程序中，我们不在使用表debug_table；取而代之的是包DBMS_OUTPUT。这样一来，该版本的Debug将只能工作在能够自动调用DBMS_OUTPUT.GET_LINES和打印缓冲区内容（如SQL*Plus）的工具中。除此之外，在使用Debug之前，SERVEROUTPUT也要设置为ON。

5. 使用问题2的调试程序包

函数CountCredits现在返回的是空的结果。现在，先让我们来验证上述的现象并看一下在循环中把什么数值赋予变量v_TotalCredits。我们在下面的程序中加入对Debug的调用：

节选自在线代码CountCredits2.sql

```

CREATE OR REPLACE FUNCTION CountCredits (
    /* Returns the number of credits for which the student
     * identified by p_StudentID is currently registered */
    p_StudentID IN students.ID%TYPE)
RETURN NUMBER AS

```

```

v_TotalCredits NUMBER; -- Total number of credits
v_CourseCredits NUMBER; -- Credits for one course
CURSOR c_RegisteredCourses IS
    SELECT department, course
        FROM registered_students
        WHERE student_id = p_StudentID;
BEGIN
    Debug.Reset;
    FOR v_CourseRec IN c_RegisteredCourses LOOP
        -- Determine the credits for this class.
        SELECT num_credits
            INTO v_CourseCredits
            FROM classes
            WHERE department = v_CourseRec.department
            AND course = v_CourseRec.course;

        Debug.Debug('Inside loop, v_CourseCredits', v_CourseCredits);
        -- Add it to the total so far.
        v_TotalCredits := v_TotalCredits + v_CourseCredits;
    END LOOP;

    Debug.Debug('After loop, returning', v_TotalCredits);
    RETURN v_TotalCredits;
END CountCredits;

```

现在该函数的输出结果是：

```

SQL> VARIABLE v_Total NUMBER
SQL> SET SERVEROUTPUT ON
SQL> exec :v_Total := CountCredits(10006);
Inside loop, v_CourseCredits: 4
Inside loop, v_CourseCredits: 3
After loop, returning:
PL/SQL procedure successfully completed.

SQL> print v_Total
      V_TOTAL
-----

```

SQL>

注意 我们是用SQL *Plus的连接变量来代替从表students中选择该函数的值来测试函数CountCredits的。这样做的原因是函数CountCredits现在调用了非纯函数DBMS_OUTPUT的缘故。如果我们在SQL语句内部使用函数CountCredits，我们将得到错误信息“ORA-6571:函数可能会更新数据库”(Oracle8i中取消了这条限制)。请看本书第5章的有关该错误和从SQL语句中调用存储函数的介绍。

基于该调试程序的输出看起来为每一个班级计算的分数是正确的。程序中的循环执行了两次，每次分别返回学分4和3。但实际上，该程序没有把学分加到总分中。现在让我们来增加几个调试语句：

节选自在线代码CountCredits3.sql

```
CREATE OR REPLACE FUNCTION CountCredits (
    /* Returns the number of credits for which the student
       identified by p_StudentID is currently registered */
    p_StudentID IN students.ID%TYPE)
RETURN NUMBER AS

v_TotalCredits NUMBER; -- Total number of credits
v_CourseCredits NUMBER; -- Credits for one course
CURSOR c_RegisteredCourses IS
    SELECT department, course
    FROM registered_students
    WHERE student_id = p_StudentID;

BEGIN
    Debug.Reset;
    Debug.Debug('Before loop, v_TotalCredits', v_TotalCredits);
    FOR v_CourseRec IN c_RegisteredCourses LOOP
        -- Determine the credits for this class.
        SELECT num_credits
        INTO v_CourseCredits
        FROM classes
        WHERE department = v_CourseRec.department
        AND course = v_CourseRec.course;

        Debug.Debug('Inside loop, v_CourseCredits', v_CourseCredits);
        -- Add it to the total so far.
        v_TotalCredits := v_TotalCredits + v_CourseCredits;
        Debug.Debug('Inside loop, v_TotalCredits', v_TotalCredits);
    END LOOP;

    Debug.Debug('After loop, returning', v_TotalCredits);
    RETURN v_TotalCredits;
END CountCredits;
```

新修改的函数CountCredits的输出如下所示：

```
SQL> EXEC :v_Total := CountCredits(10006);
Before loop, v_TotalCredits:
Inside loop, v_CourseCredits: 4
Inside loop, v_TotalCredits:
Inside loop, v_CourseCredits: 3
Inside loop, v_TotalCredits:
After loop, returning:
PL/SQL procedure successfully completed.
```

我们可以从该输出中发现程序的错误。我们可以发现变量 v_TotalCredits的值在循环开始前为空，在循环中也一直为空，这是因为我们在该变量的声明中没有对其进行初始化。该问题可以用下面的最终版本来解决，我们在该版本中删除了调试语句。

节选自在线代码CountCredits4.sql

```
CREATE OR REPLACE FUNCTION CountCredits (
    /* Returns the number of credits for which the student
```

```

identified by p_StudentID is currently registered */
p_StudentID IN students.ID%TYPE)
RETURN NUMBER AS

v_TotalCredits NUMBER := 0; -- Total number of credits
v_CourseCredits NUMBER; -- Credits for one course
CURSOR c_RegisteredCourses IS
SELECT department, course
FROM registered_students
WHERE student_id = p_StudentID;

BEGIN
FOR v_CourseRec IN c_RegisteredCourses LOOP
-- Determine the credits for this class.
SELECT num_credits
INTO v_CourseCredits
FROM classes
WHERE department = v_CourseRec.department
AND course = v_CourseRec.course;

-- Add it to the total so far.
v_TotalCredits := v_TotalCredits + v_CourseCredits;
END LOOP;

RETURN v_TotalCredits;
END CountCredits;

```

该版本的输出如下：

```

SQL> EXEC: V_Total:=CountCredits(10006);
PL/SQL procedure successfully completed.
SQL> print v_Total
V_TOTAL
-----
7
SQL> SELECT ID, CountCredits(ID) "Total Credits"
2   FROM students;
      ID Total Credits
-----
10000          11
10001           4
10002           8
10003           8
10004           4
10005           4
10006           7
10007           4
10008           8
10009           7
10010           8

```

10011

3

12 rows selected.

我们可以看到函数CountCredits在计算单个学生和全表学生的分数时都可以正常工作了。如果某个变量在其声明时没有进行初始化，该变量就被赋予空值 NULL。根据计算NULL表达式的规则，该空值将在加法操作中保持为空。

6. 问题2解决方案的评价

该Debug版本的功能与上节的第一个版本有所不同。也就是说，我们取消了该程序对表debug_table的依赖。这样做的优点如下：

- 由于每个会话都有其自己的DBMS_OUTPUT内部缓冲区，从而解决了多数据库会话之间的相互干扰问题。
- 我们不需要在程序Debug.Debug中再发送提交命令COMMIT。
- 只要SERVEROUTPUT处于ON的状态，就不需要额外的SELECT语句来查看输出。同样，我们可以将SERVEROUTPUT设置为OFF状态来关闭调试状态。如果 SERVEROUTPUT为OFF的话，调试信息仍然将被写入DBMS_OUTPUT缓冲区中，但不再输出到屏幕。

从另一方面来说，该版本的调试程序还要注意下列问题：

- 如果我们不使用象SQL*Plus或服务器管理器一类的工具，则调试输出将不能自动地发送到屏幕显示。该程序包仍然可以从其他的PL/SQL运行环境下使用（如Pro*C或Oracle Forms），但你必须显式地调用DBMS_OUTPUT.GET_LINE或DBMS_OUTPUT.GET_LINES，自己实现显示输出结果。
- 调试输出的数量受到了DBMS_OUTPUT缓冲区容量的限制。该问题会影响每行的长度和缓冲区的总长度。如果发现输出的内容太多并且缓冲区不能满足需要，这时使用上节第一个版本的Debug可能会更好一些。

3.3 PL/SQL调试器

目前有几种集成了调试器的PL/SQL开发工具可以使用。带有调试器的开发工具是非常有用的，这是因为这种工具在程序处于运行的状态下提供了逐行查看PL/SQL源码以及分析各个变量的数值的功能。除此之外，我们还可以在不同的点设置程序断点并观察特殊变量的值。

3.3.1 PL/SQL调试器功能概述

图形调试工具与非图形调试工具相比有下述几个优点：

- 不需要我们在应用程序中增加任何调试代码；我们只要在受控的调试环境下运行该应用程序就可以实现调试。
- 由于应用程序的代码没有任何变动，也不需要重新编译就可以运行并观察不同的变量，所以调试过程非常方便。
- 所有的工具都提供了集成的开发环境，具有浏览和编辑功能。

然而，在某些情况下，图形调试工具环境无法满足调试要求。例如，并非所有的操作系统平台都支持GUI（图形用户界面）工具。在这些情况下，非图形调试技术，或者我们在本章后面要讨论的跟踪和配置工具可能会满足特殊的要求。

在下面的几节中，我们将分析在第2章中提到的GUI工具的调试功能，这些工具都存储在本书的CD-ROM中。每种工具的调试功能在表3-1中给予了总结。有关这些工具的详细信息，请看本书CD-ROM中的联机文档。

表3-1 PL/SQL调试器功能比较

功能	Rapid SQL	XPEDITOR/SQL	SQL Navigator	TOAD	SQL Programmer
动画	否	是	否	否	否
观察点	是	是	是	是	是
自动显示当前变量	是	是	是	否	否
异常结束	否	是	否	是	是
动态修改变量值	是，只有观察点	是	是	否	是
服务器端必须安装调试器	否	是	是	否	否
调试匿名块	否	是	否	否	否
连接外部进程	否	是（通过DBPartner）	是	否	否

3.3.2 问题3

请看下面的过程：

节选自在线代码CreditLoop1.sql

```
CREATE OR REPLACE PROCEDURE CreditLoop AS
/* Inserts the student ID numbers and their current credit
   values into temp_table. */
v_StudentID students.ID%TYPE;
v_Credits students.current_credits%TYPE;
CURSOR c_Students IS
  SELECT ID
    FROM students;
BEGIN
  OPEN c_Students;
LOOP
  FETCH c_Students INTO v_StudentID;
  v_Credits := CountCredits(v_StudentID);
  INSERT INTO temp_table (num_col, char_col)
    VALUES (v_StudentID, 'Credits = ' || TO_CHAR(v_Credits));
  EXIT WHEN c_Students%NOTFOUND;
END LOOP;
CLOSE c_Students;
END CreditLoop;
```

CreditLoop只是在表temp_table中记录每个学生的分数。该过程调用前一节使用的函数CountCredits来计算每个学生的分数。当开始运行函数CreditLoop并在SQL *Plus中查询表temp_table时，我们得到下列输出结果：

节选自在线代码callCL.sql

```
SQL> EXEC CreditLoop;
```

```
PL/SQL procedure successfully completed.  
SQL> SELECT *  
  2 FROM temp_table  
  3 ORDER BY num_col;  
  
NUM_COL CHAR_COL  
-----  
10000 Credits = 11  
10001 Credits = 4  
10002 Credits = 8  
10003 Credits = 8  
10004 Credits = 4  
10005 Credits = 4  
10006 Credits = 7  
10007 Credits = 4  
10008 Credits = 8  
10009 Credits = 7  
10010 Credits = 8  
10011 Credits = 3  
10011 Credits = 3  
13 rows selected.
```

该输出的问题是最后两行被插入了两次，也就是说，学生 10011有两行分数，而其他所有学生只有一行。

1. 问题3:使用Rapid SQL进行调试

在Rapid SQL中调试存储过程的第一步是在 PL/SQL编辑器窗口中打开该过程。接着，我们可以通过选择菜项 Debug | Start Debugging,或单击如图3-2所示的图标Debug PL/SQL开始调试。调试开始时，Rapid SQL将初始化一个调试会话并建立如图 3-3所示的环境。

调试会话包括四个附加的窗口，它们由 Rapid SQL自动打开并放置在屏幕的底部：

- 监视窗口，该窗口显示用户指定变量的值，不管变量是否在其作用域中，你都可以在编辑窗口中将任何变量选中并将其拖动到监视窗口中进行观察。
- 变量窗口，该窗口显示处于作用域中的变量的数值。
- 调用栈窗口，该窗口显示当前运行的源程序行，以及完整的 PL/SQL调用栈。
- 相关树窗口（Dependency tree window），该窗口显示当前对象之间的相关树形结构。如图 3-3 所示，对象 CreditLoop依赖于对象 CountCredits。

除此之外，编辑窗口中有一个箭头指向当前行的下面。我们可以单击工具条中的 Setp into 按钮来启动该过程开始运行。在此之前，我们要设置一个断点来观察程序的运行情况，当启动程序运行到断点处来观察变量窗口中本地变量的值。设置断点的方法是单击所希望的源程序行，接着再单击 Insert或Remove断点按钮来设置或取消断点。对于过程 CreditLoop来说，我们希望断点设置在程序的第 13 行，在FETCH语句后停止。如图 3-4所示，一旦建立了断点，我们就可以单击Go按钮运行程序到断点为止。图 3-5显示了程序运行到断点处的窗口。如该窗口所示，我们已经取出了第一行，变量 v_StudentID的值是10000,变量v_Credits的值是空（NULL），这表明程序运行正常。

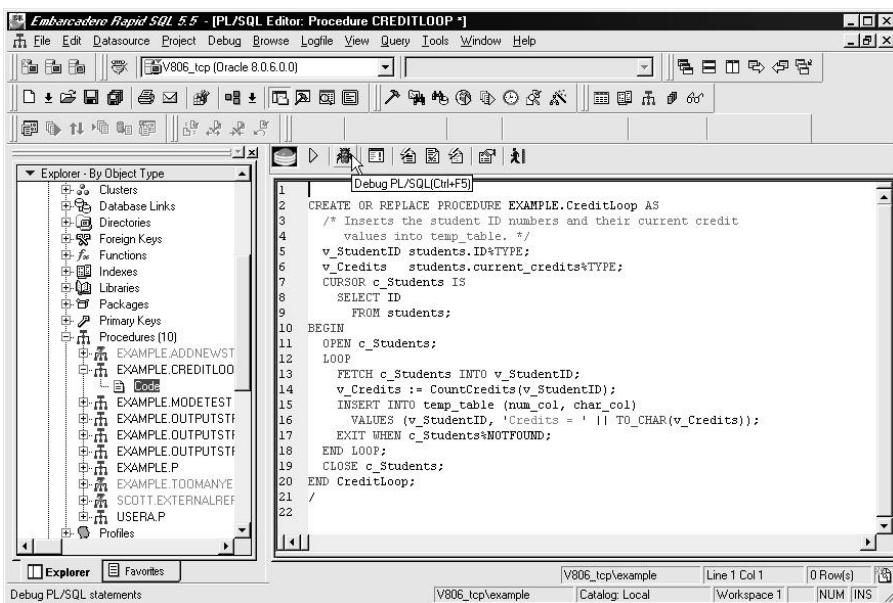


图3-2 准备开始调试过程 CreditLoop的窗口

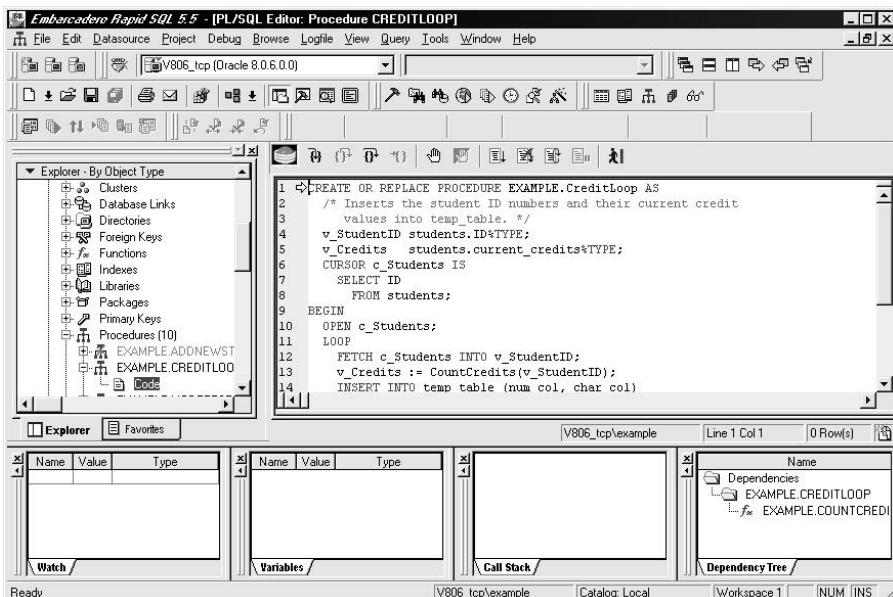


图3-3 初始化的CreditLoop调试会话窗口

从该点开始，我们可以单击按钮 Go进行单步调试，观察每个 FETCH语句执行后变量 v_StudentID的值。在执行过程中，我们可以看到v_StudentID的值是在正常范围内变化。当程序执行到最后一个FETCH语句时，返回的v_StudentID值是10011。现在，我们把该值插入到表temp_table中并再次开始执行循环，而发现下一个FETCH语句没有改变v_StudentID的值，它仍然是10011，因

此，该值被执行了两次插入。在第二个插入语句INSERT后，该循环由于c_Students%NOTFOUND为真而结束循环。这就是问题所在。也就是说，退出语句EXIT应该在FETCH语句后立即执行才对。现在，我们在PL/SQL编辑器中来修改CreditLoop并再次对其进行测试以确保其功能正确。图3-6显示的是正确运行的CreditLoop程序。该程序的在线代码名称为CreditLoop2.sql。

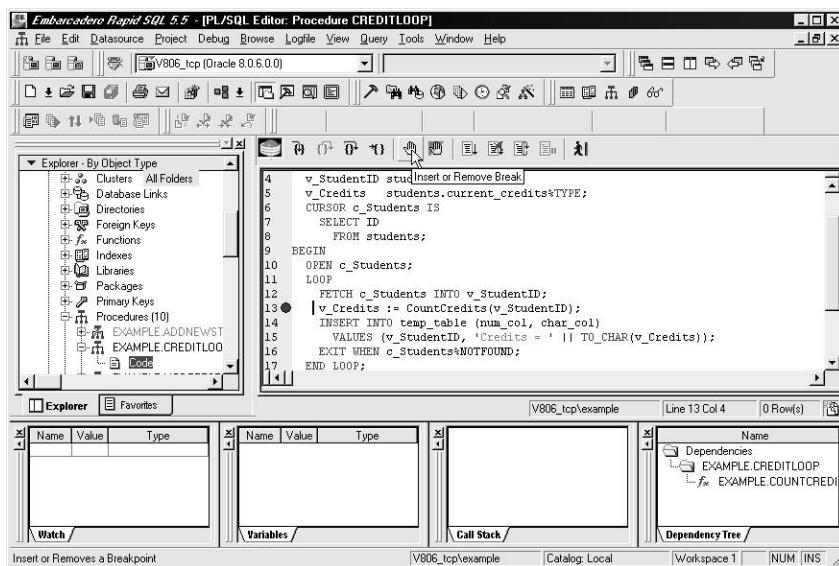


图3-4 在窗口中设置断点

解决了CreditLoop存在的问题后，我们可以从SQL *Plus中再次运行该程序，其输出结果如下：

节选自在线代码callCL.sql

```
SQL> EXEC CreditLoop
PL/SQL procedure successfully completed.
```

```
SQL> SELECT * FROM temp_table
  2 ORDER BY num_col;
  NUM_COL CHAR_COL
  -----
 10000 Credits = 11
 10001 Credits = 4
 10002 Credits = 8
 10003 Credits = 8
 10004 Credits = 4
 10005 Credits = 4
 10006 Credits = 7
 10007 Credits = 4
 10008 Credits = 8
 10009 Credits = 7
 10010 Credits = 8
 10011 Credits = 3
12 rows selected.
```

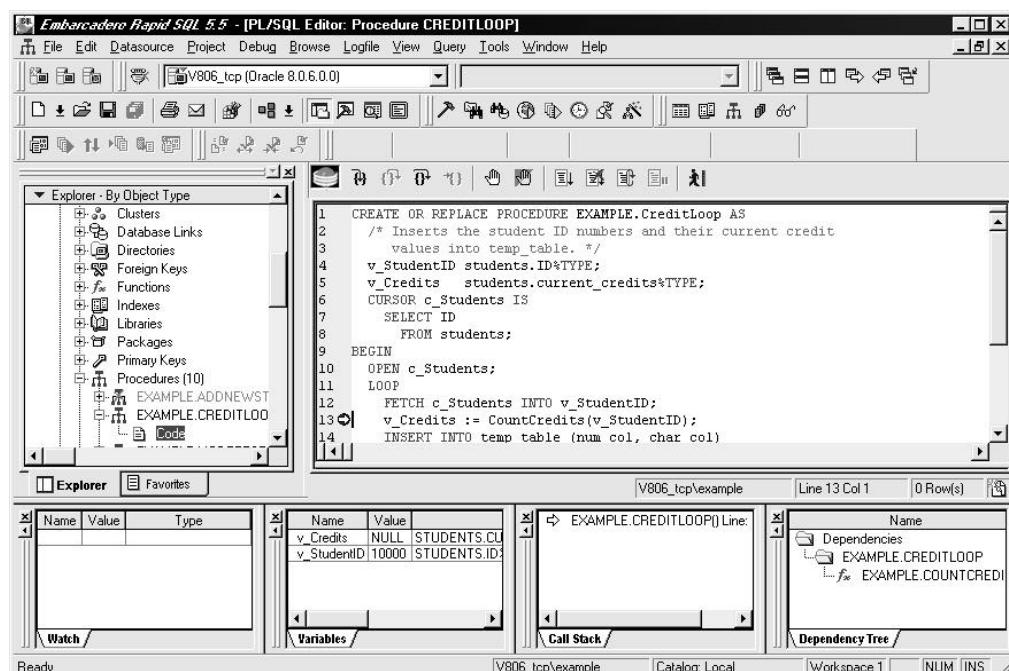


图3-5 停止在断点的运行窗口

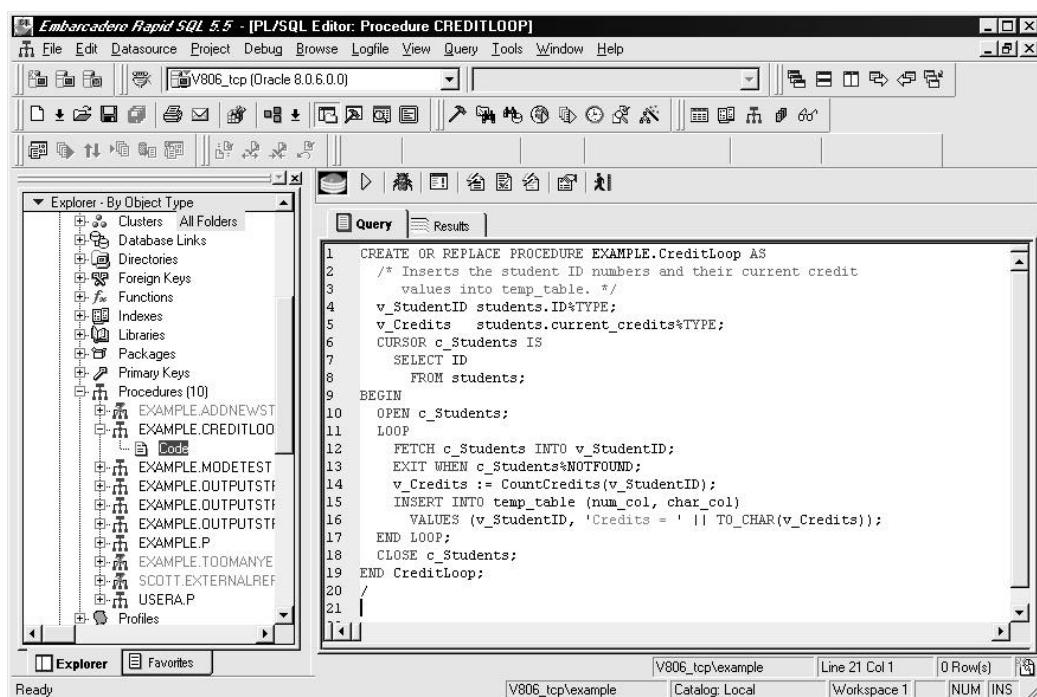


图3-6 CreditLoop的正确版本

2. 问题3: 评论

游标提取循环一直运行到返回 NO_DATA_FOUND为止(用属性%NOTFOUND进行检测),一旦检测到了%NOTFOUND为真,由于已经没有数据可取,该循环就停止运行。输出变量v_StudentID将保持前一次循环迭代的值。

提示 读者也可以使用游标FOR循环,该类循环可以隐式地打开游标,在循环中每次执行一次取操作,并在结束循环结束时关闭该游标。

3.3.3 问题4

如果游标存储在包中的话,该游标将一直维持到某个包的子程序调用生存期之外。这样以来就使我们可以编制一个从该游标每次取出若干行的子程序。这也就是包 StudentFetch的实际功能。

```
节选自在线代码StudentFetch1.sql
CREATE OR REPLACE PACKAGE StudentFetch AS
    TYPE t_Students IS TABLE OF students%ROWTYPE
        INDEX BY BINARY_INTEGER;

    -- Opens the cursor for processing.
    PROCEDURE OpenCursor;

    -- Closes the cursor.
    PROCEDURE CloseCursor;

    -- Returns up to p_BatchSize rows in p_Students, and returns
    -- TRUE as long as there are still rows to be fetched.
    FUNCTION FetchRows(p_BatchSize IN NUMBER := 5,
                       p_Students OUT t_Students)
        RETURN BOOLEAN;

    -- Prints p_BatchSize rows from p_Students.
    PROCEDURE PrintRows(p_BatchSize IN NUMBER,
                        p_Students IN t_Students);

END StudentFetch;

CREATE OR REPLACE PACKAGE BODY StudentFetch AS
CURSOR c_AllStudents IS
    SELECT *
        FROM students
        ORDER BY ID;

    -- Opens the cursor for processing.
    PROCEDURE OpenCursor IS
    BEGIN
        OPEN c_AllStudents;
```

```

END OpenCursor;

-- Closes the cursor.
PROCEDURE CloseCursor IS
BEGIN
    CLOSE c_AllStudents;
END CloseCursor;

-- Returns up to p_BatchSize rows in p_Students, and returns
-- TRUE as long as there are still rows to be fetched.
FUNCTION FetchRows(p_BatchSize IN NUMBER := 5,
                    p_Students OUT t_Students)
RETURN BOOLEAN IS
    v_Finished BOOLEAN := TRUE;
BEGIN
    FOR v_Count IN 1..p_BatchSize LOOP
        FETCH c_AllStudents INTO p_Students(v_Count);
        IF c_AllStudents%NOTFOUND THEN
            v_Finished := FALSE;
            EXIT;
        END IF;
    END LOOP;
    RETURN v_Finished;
END FetchRows;

-- Prints p_BatchSize rows from p_Students.
PROCEDURE PrintRows(p_BatchSize IN NUMBER,
                     p_Students IN t_Students) IS
BEGIN
    FOR v_Count IN 1..p_BatchSize LOOP
        DBMS_OUTPUT.PUT('ID: ' || p_Students(v_Count).ID);
        DBMS_OUTPUT.PUT(' Name: ' || p_Students(v_Count).first_name);
        DBMS_OUTPUT.PUT_LINE(' ' || p_Students(v_Count).last_name);
    END LOOP;
END PrintRows;
END StudentFetch;

```

每次我们调用StudentFecth.FetchRows时，该程序应返回下一批行。下面的 SQL *Plus会话演示了该程序的执行过程。

节选自在线代码callSF1.sql

```

SQL> DECLARE
2   v_BatchSize NUMBER := 5;
3   v_Students StudentFetch.t_Students;
4 BEGIN
5   StudentFetch.OpenCursor;
6   WHILE StudentFetch.FetchRows(v_BatchSize, v_Students) LOOP
7       StudentFetch.PrintRows(v_BatchSize, v_Students);
8   END LOOP;

```

```
9 StudentFetch.CloseCursor;
10 END;
11 /
ID: 10000 Name: Scott Smith
ID: 10001 Name: Margaret Mason
ID: 10002 Name: Joanne Junebug
ID: 10003 Name: Manish Murgatroid
ID: 10004 Name: Patrick Poll
ID: 10005 Name: Timothy Taller
ID: 10006 Name: Barbara Blues
ID: 10007 Name: David Dinsmore
ID: 10008 Name: Ester Elegant
ID: 10009 Name: Rose Riznit
PL/SQL procedure successfully completed.
```

然而，该过程有一个问题，即我们没有取出表 students 所有的记录。该程序只是返回了 10 行，但实际上，该表有 12 行。

1. 问题4:使用XPEDITER/SQL进行调试

下面我们将使用 XPEDITER/SQL 调试器来解决上述问题。首先，我们要在该调试器窗口中载入调用块。这可以通过在记事本窗口中打开该调用块实现，然后单击图 3-7 所示的 Debug 按钮。最后的调试窗口如图 3-8 所示。

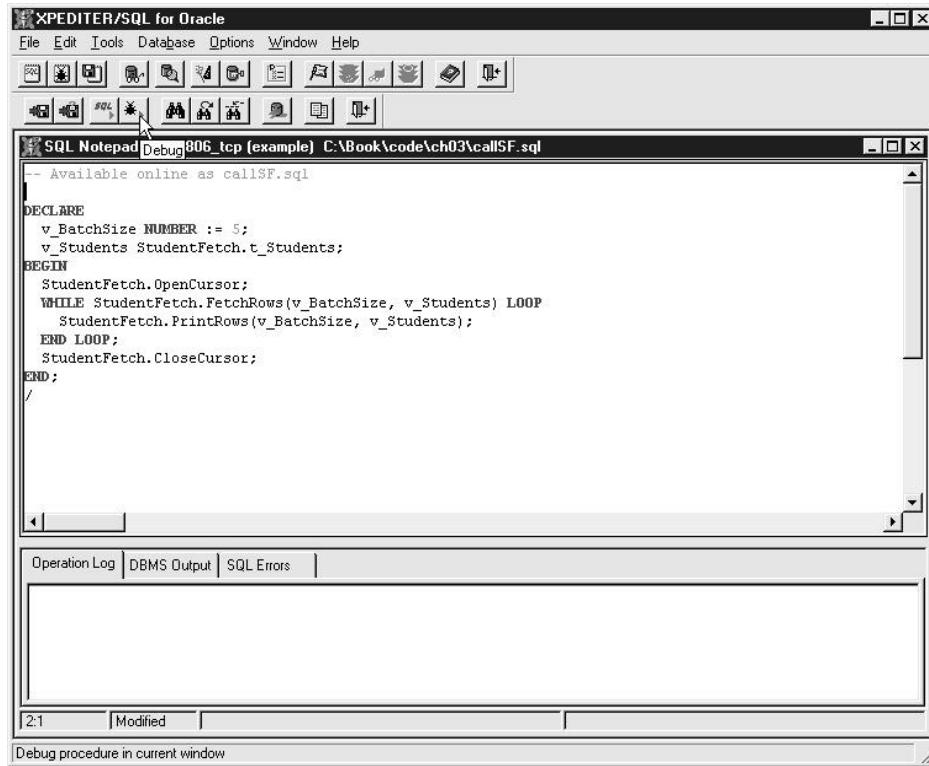


图3-7 准备调试调用块的窗口

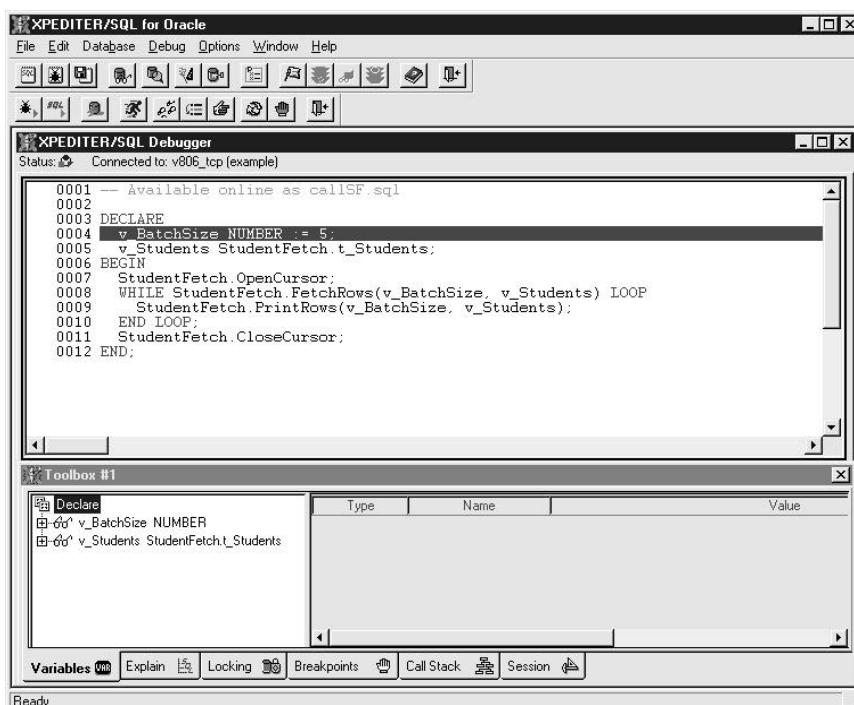


图3-8 调试窗口显示的调用块

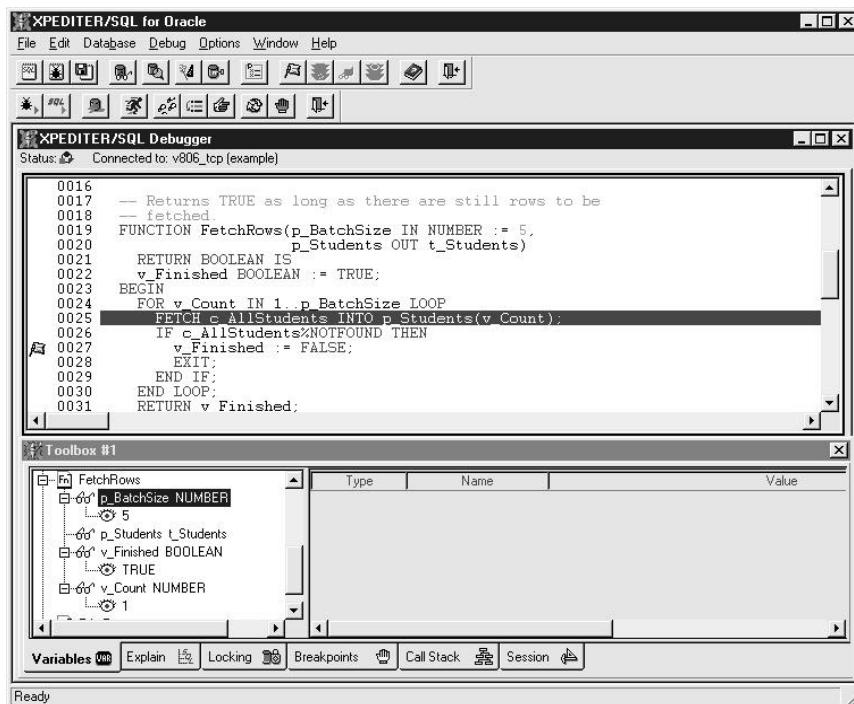


图3-9 设置断点

现在我们可以单步运行该代码到 FetchRows这一行。我们要在第 27行设置一个断点来观察变量v_Finished何时被置为FALSE。设置该断点的方法是双击该语句所在的行号。图 3-9是设置了该断点的窗口。现在我们可以运行该程序直到设置的断点为止。在运行过程中，除了运行之外，我们还可以使用XPEDITOR/SQL的动画功能，该功能自动地按用户设置的执行速度单步运行到断点。执行动画功能后到达断点的窗口如图 3-10所示。

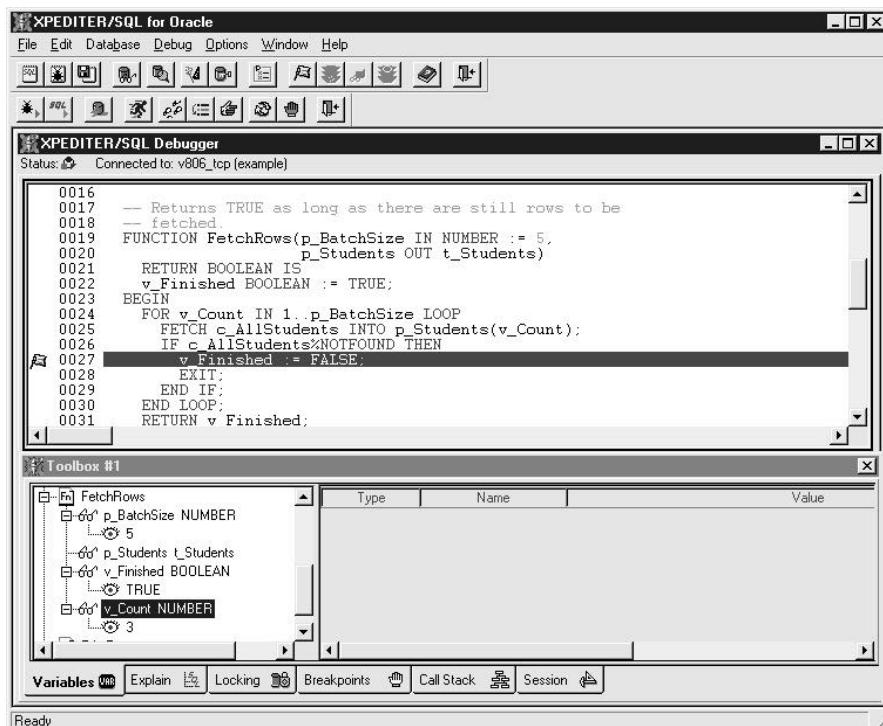


图3-10 停在断点的调试窗口

问题现在搞清楚了。我们可以从前面的显示内容以及从变量 v_Count的当前值是3（本地变量显示在图3-10的窗口的工具箱中）的情况下确认对 FetchRows的调用取回了两行记录。在该断点，我们将变量v_finished设置为FALSE并立即返回该值。但是调用块不会立即打印这些额外的行记录，这是因为该调用块只在 FetchRows返回TRUE时才调用打印语句PrintRows。

解决上面的问题需要做两处修改。首先，函数 FectchRows要能够返回检索到的实际行数，改动的代码如下所示：

节选自在线代码StudentFetch2.sql

```
-- Returns up to p_BatchSize rows in p_Students, and returns
-- TRUE as long as there are still rows to be fetched.
-- The actual number of rows fetched is returned in p_BatchSize.
FUNCTION FetchRows(p_BatchSize IN OUT NUMBER,
                    p_Students OUT t_Students)

RETURN BOOLEAN IS
    v_Finished BOOLEAN := TRUE;
```

```

BEGIN
  FOR v_Count IN 1..p_BatchSize LOOP
    FETCH c_AllStudents INTO p_Students(v_Count);
    IF c_AllStudents%NOTFOUND THEN
      v_Finished := FALSE;
      p_BatchSize := v_Count - 1;
      EXIT;
    END IF;
  END LOOP;
  RETURN v_Finished;
END FetchRows;

```

第二，我们要在最后一个 fetch操作后调用打印函数 PrintRows，修改的程序如下：

```

节选自在线代码callSF2.sql
SQL> DECLARE
 2   v_BatchSize NUMBER := 5;
 3   v_Students StudentFetch.t_Students;
 4 BEGIN
 5   StudentFetch.OpenCursor;
 6   WHILE StudentFetch.FetchRows(v_BatchSize, v_Students) LOOP
 7     StudentFetch.PrintRows(v_BatchSize, v_Students);
 8   END LOOP;
 9   -- Print any extra rows from the last batch.
10  IF v_BatchSize != 0 THEN
11    StudentFetch.PrintRows(v_BatchSize, v_Students);
12  END IF;
13  StudentFetch.CloseCursor;
14 END;
15 /
ID: 10000 Name: Scott Smith
ID: 10001 Name: Margaret Mason
ID: 10002 Name: Joanne Junebug
ID: 10003 Name: Manish Murgatroid
ID: 10004 Name: Patrick Poll
ID: 10005 Name: Timothy Taller
ID: 10006 Name: Barbara Blues
ID: 10007 Name: David Dinsmore
ID: 10008 Name: Ester Elegant
ID: 10009 Name: Rose Riznit
ID: 10010 Name: Rita Razmataz
ID: 10011 Name: Shay Shariatpanahy
PL/SQL procedure successfully completed.

```

2. 问题4:评论

一开始，该 fetch循环看起来非常象问题 3中分析过的循环。然而，该循环的每个 fectch操作可以返回多达 5行记录，而不是一个记录。该循环类似于成组循环操作，需要注意的是在结束 fetch的条件出现后，我们必须继续处理剩余的行。

3.3.4 问题5

斐波纳契序列是由如 $Fib(n)=Fib(n-1)+Fib(n-2)$ 一系列数组成的。 $Fib(0)$ 的值为 0 , $Fib(1)$ 的值为 1。由此定义了下面的序列 :

0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , ...

我们可以用 PL/SQL 写一个函数来计算第 n 个斐波纳契数。该函数如下 :

节选自在线代码 Fibonacci.sql

```
CREATE OR REPLACE FUNCTION Fib(n IN BINARY_INTEGER)
  RETURN BINARY_INTEGER AS
BEGIN
  RETURN Fib(n - 1) + Fib(n - 2);
END Fib;
```

然而 , 当我们从 SQL *Plus 中运行该函数时 , 该函数没有返回结果并出现内存溢出错误。

1. 问题5 : 使用 SQL Navigator 进行调试

下面 , 我们来使用 SQL Navigator 调试器来解决上述问题。类似于其他的 PL/SQL 调试器 , 我们可以直接在 SQL Navigator 内部调试函数 Fib 的全部代码 , 如图 3-11 所示 , 我们只要在存程序编辑器中打开该函数并执行单步操作就可以进入调试。这时 , 该窗口内将出现一个会话框来接收计算需要的初值 n , 并构造一个匿名块来调用该函数。

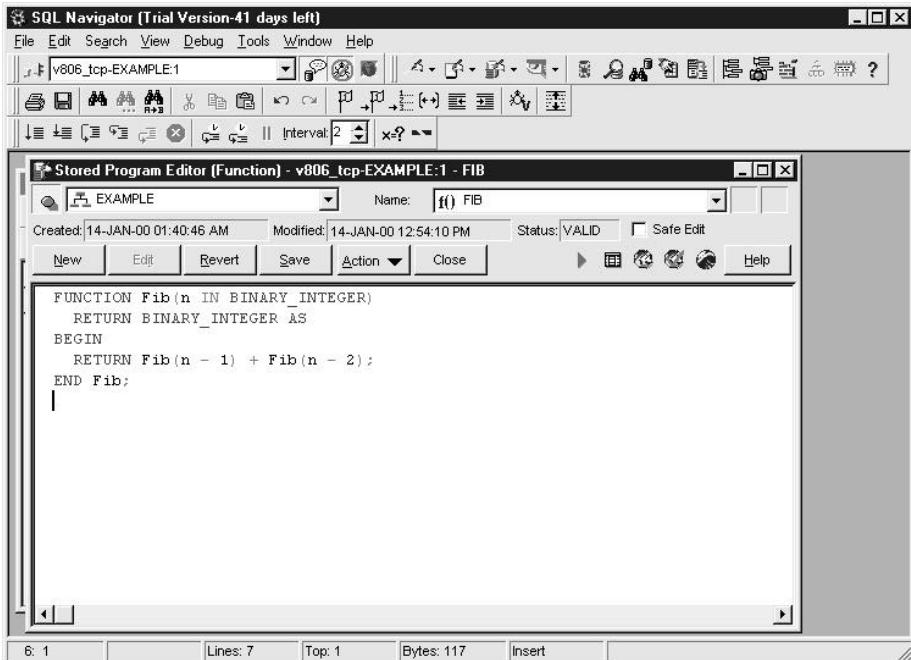


图3-11 准备直接调试函数 Fib

但现在我们要从 SQL *Plus 的窗口中调用函数 Fib 并且把 SQL Navigator 的会话与 SQL *Plus 的窗口连接。这种功能类似某些 C 调试器具有的可以把运行的进程与其连接调试的能力。为了使用

PL/SQL实现这种调试方法，调用程序必须首先调用包 DBMS_DEBUG中的两个子程序来启动调试器与其连接，下面是实现该功能的命令：

```
节选自在线代码attachFib.sql
SQL> -- First initialize the debugger
SQL> ALTER SESSION SET PLSQL_DEBUG = TRUE;
Session altered.
SQL> DECLARE
2   v_DebugID VARCHAR2(30);
3 BEGIN
4   v_DebugID := DBMS_DEBUG.INITIALIZE('DebugMe');
5   DBMS_DEBUG.DEBUG_ON;
6 END;
7 /
PL/SQL procedure successfully completed.
```

ALTER SESSION语句命令PL/SQL编译器来编译带有 DEBUG选项的未来匿名块（Future anonymous block）。DBMS_DEBUG.INITIALIZE带有供服务器标识的字符串，DBMS_DEBUG.DEBUG_ON告诉服务器下面的块包括了要调试的调用。如下所示，我们通过发布一个对Fib的调用来实现该功能。

```
SQL> -- And now execute the call to Fib. This will hang until you
SQL> -- attach to this session in SQL Navigator.
SQL> exec DBMS_OUTPUT.PUT_LINE(Fib(3));
```

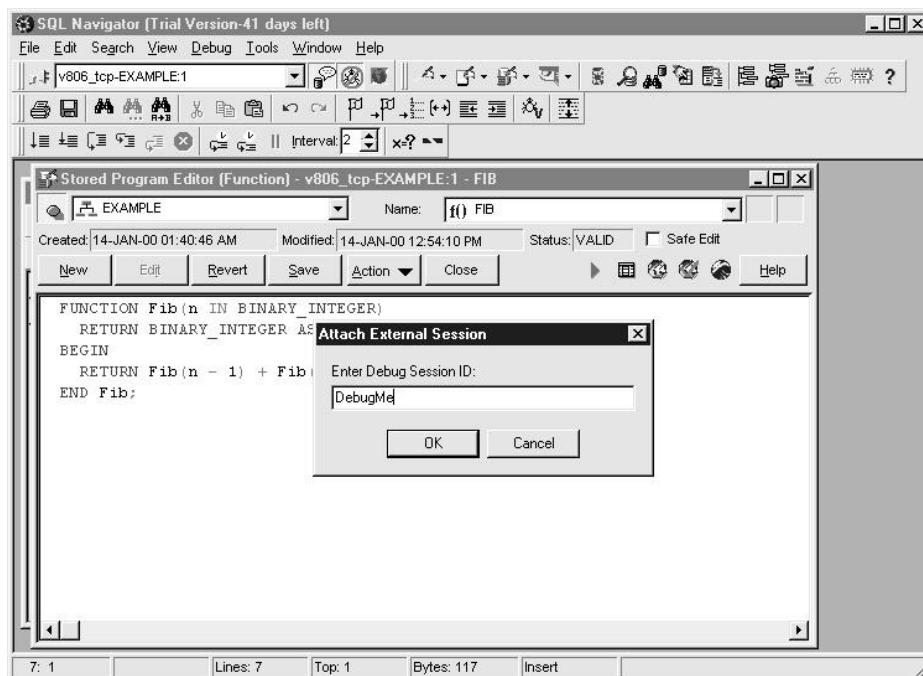


图3-12 连接会话对话框

现在，我们通过选择菜单项 Debug | Attach External Session 来连接SQL Navigator的会话。执行该选择后将启动一个对话框，请求输入在调用 INITIALIZE中使用的标识符，该对话框如图3-12所示。一旦我们输入了会话标识符，屏幕上就出现一个运行状态窗口，同时还有一个显示我们从SQL *Plus中运行的匿名块的调用栈的子窗口。现在，我们可以进入该块对函数 Fib进行调试了。Fib的调试窗口如图3-13所示。

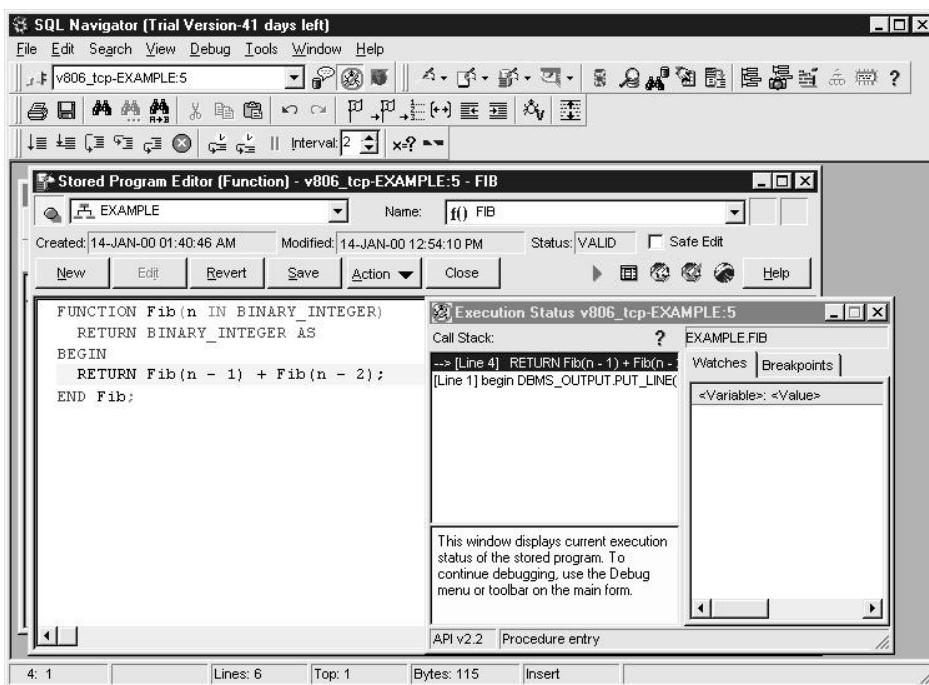


图3-13 调试函数Fib的窗口

我们现在可以为n设置一个观察点并继续单步执行程序。设置变量的观察点的方法是先选中变量，接着右键单击该变量，最后从上下文菜单中选择 Add Watchpoint来建立观察点。我们可以继续单步运行该程序来观察当函数返回时 n有什么变化。运行该程序几次后，n的值如图3-14所示，从该图中，我们可以发现出现问题的原因。

可以看出，我们在函数 Fib中没有设置结束条件。第一个调用将从 n中减去1，接着循环调用 Fib函数。这次调用也进行减1操作，并再反复调用。该过程继续下去，不断地从 n中减去数值，同时把另外的调用加入到堆栈，直到发生存储器溢出错位。解决该问题的方法是在 Fib中加入停止条件，下面是经过改动的程序：

节选自在线代码Fibonacci.sql

```
CREATE OR REPLACE FUNCTION Fib(n IN BINARY_INTEGER)
  RETURN BINARY_INTEGER AS
BEGIN
  IF n = 0 OR n = 1 THEN
    RETURN n;
  ELSE
    RETURN Fib(n - 1) + Fib(n - 2);
  END IF;
END;
```

```

ELSE
    RETURN Fib(n - 1) + Fib(n - 2);
END IF;
END Fib;

```

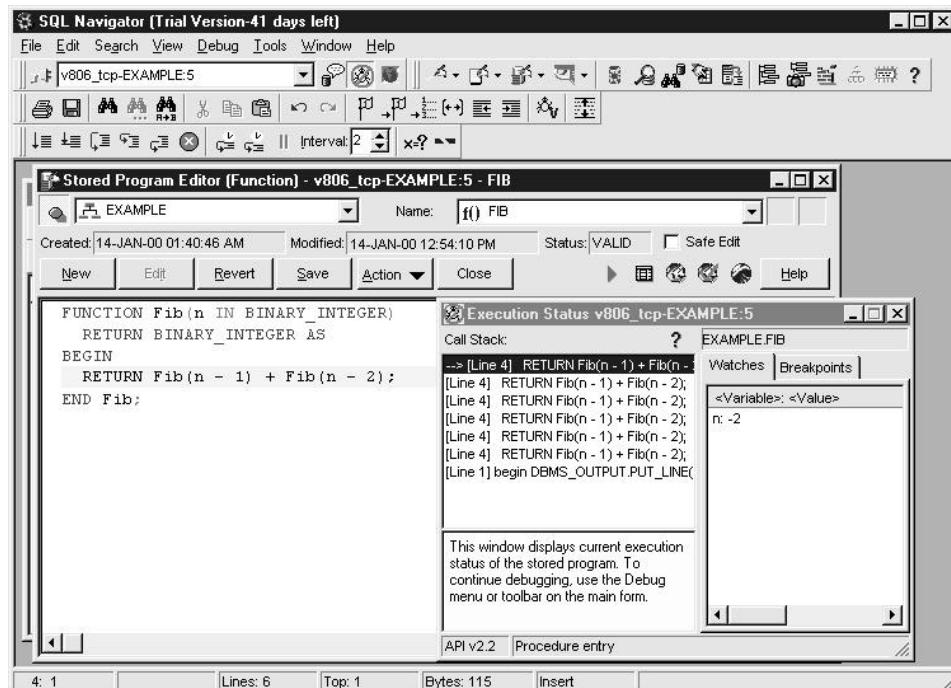


图3-14 变量的负值

上面的版本确实可以返回正确的结果，下面是 SQL *Plus的会话：

节选自在线代码Fibonacci.sql

```

SQL> BEGIN
 2      -- Some calls to Fib.
 3      FOR v_Count IN 1..10 LOOP
 4          DBMS_OUTPUT.PUT_LINE(
 5              'Fib(' || v_Count || ') is ' || Fib(v_Count));
 6      END LOOP;
 7  END;
 8 /
Fib(1) is 1
Fib(2) is 1
Fib(3) is 2
Fib(4) is 3
Fib(5) is 5
Fib(6) is 8
Fib(7) is 13
Fib(8) is 21

```

```
Fib(9) is 34
Fib(10) is 55
PL/SQL procedure successfully completed.
```

2. 问题5:评论

递归函数可以提供简单而精致的解决方案，但其前提是它们必须具有正确的停止条件。如果没有设置这种条件，这种函数将会导致内存溢出错误。

即使设置了停止条件，然而，递归函数并不是解决某类特殊问题的最有效方式，这是因为递归函数涉及了重复的函数调用。我们将在本章的基于 PL/SQL的配置一节中介绍函数 Fib的迭代版本程序。

3.3.5 问题6

在很多情况下，PL/SQL程序的问题并不是在程序的本身，而是与程序处理的数据有关。例如，请看下面的SQL脚本，该脚本完成从表source向表destination复制数据的任务：

```
节选自在线代码CopyTables.sql
CREATE OR REPLACE PROCEDURE CopyTables AS
    v_Key    source.key%TYPE;
    v_Value  source.value%TYPE;

    CURSOR c_AllData IS
        SELECT *
        FROM source;
    BEGIN
        OPEN c_AllData;
        LOOP
            FETCH c_AllData INTO v_Key, v_Value;
            EXIT WHEN c_AllData%NOTFOUND;

            INSERT INTO destination (key, value)
                VALUES (v_Key, TO_NUMBER(v_Value));
        END LOOP;

        CLOSE c_AllData;
    END CopyTables;
```

上面的表source和表destination都是用下面的语句创建的：

```
节选自在线代码relTables.sql
```

```
CREATE TABLE source (
    key NUMBER(5),
    value VARCHAR2(50));

CREATE TABLE destination (
    key NUMBER(5),
    value NUMBER);
```

请注意表source的列值是VARCHAR2,但表destination的列值是NUMBER类型。假设我们使用下面的PL/SQL块向表source填充500行的话，对于这500行来说，其中的499行是合法的字符串（可以转换为NUMBER类型）。然而，其中一行（使用第4章中的Random程序包随机生成）具有非法值。

```
节选自在线代码 populate.sql
DECLARE
    v_RandomKey source.key%TYPE;
BEGIN
    -- First fill up the source table with legal values.
    FOR v_Key IN 1..500 LOOP
        INSERT INTO source (key, value)
        VALUES (v_Key, TO_CHAR(v_Key));
    END LOOP;

    -- Now, pick a random number between 1 and 500, and update that
    -- row to an illegal value.
    v_RandomKey := Random.RandMax(500);
    UPDATE source
    SET value = 'Oops, not a number!'
    WHERE key = v_RandomKey;

    COMMIT;
END;
```

如果我们现在调用CopyTables，我们将得到编号为ORA-1722的错误信息：

```
SQL> exec CopyTables
begin CopyTables; end;
*
ERROR at line 1:
ORA-01722: invalid number
ORA-06512: at "EXAMPLE.COPYTABLES", line 15
ORA-06512: at line 1
```

可以看出，该错误发生在执行INSERT语句时，当我们不知道哪个值有问题，为了找出该非法值，我们可以使用调试器来确认错误发生的时间。

1. 问题6：使用TOAD调试器

为了在TOAD调试器下运行PL/SQL过程，首先必须从“ Stored Procedure Edit/Compile ”窗口下运行该程序，如图3-15所示，CopyTables出现在打开的窗口中。接着，我们可以通过选择菜单项 Debug | Trace Into，或单击调试器窗口中工具条中的 Trace Into按钮开始单步调试该过程。如图3-16所示，该过程被启动执行，并停在第一行的位置。

下一步是为两个本地变量 v_Key和v_Value，设置观察点。我们只要选中这两个变量并单击 Add Watch按钮就可以将它们加入到变量观察窗口中。这时的窗口如图 3-17所示。现在，这两个变量的值都为空（NULL），这是因为我们还没有对它们赋值的原因。我们可以继续运行该过程，TOAD将在有错误发生时自动停止运行，发生错误的窗口如图 3-18所示。当我们继续运行该程序

时，本地变量的值将显示在观察窗口中。该窗口中显示的内容告诉我们有问题的数据是在变量 v_Key 的值为 434 的地方，显示该错误数据的窗口如图 3-19 所示。

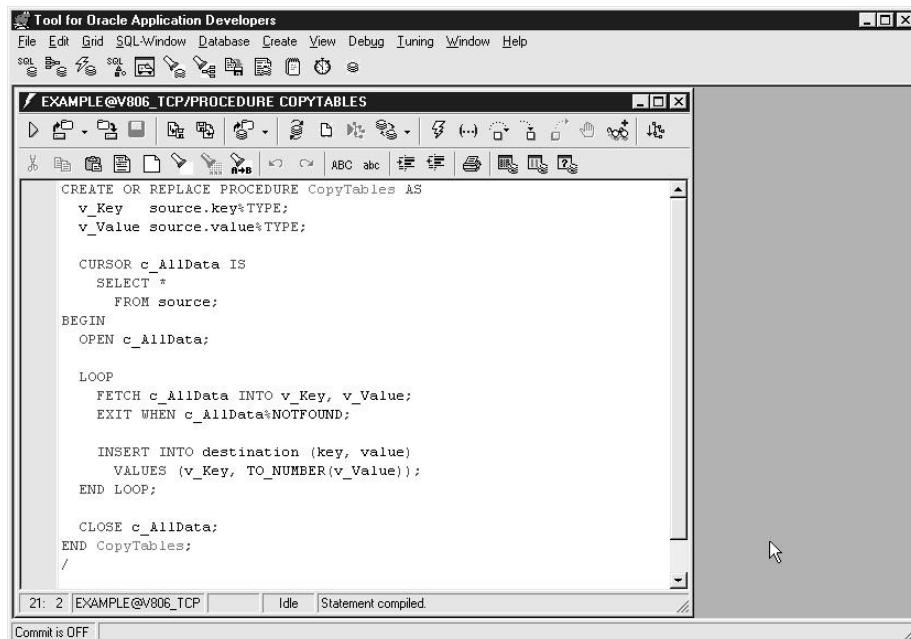


图3-15 显示Edit/Compile窗口中的CopyTables

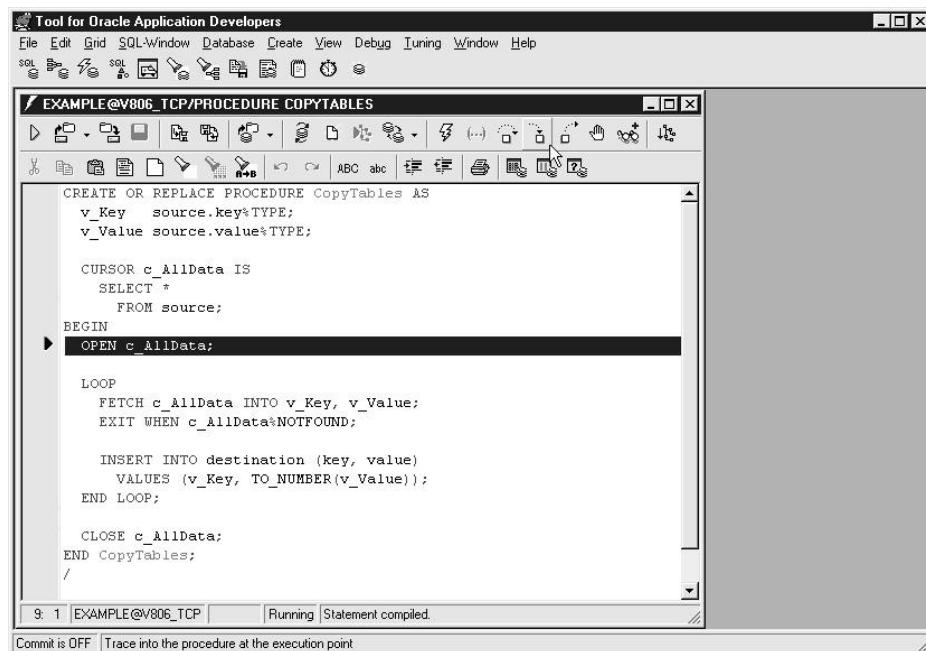


图3-16 停止在过程第一行的窗口

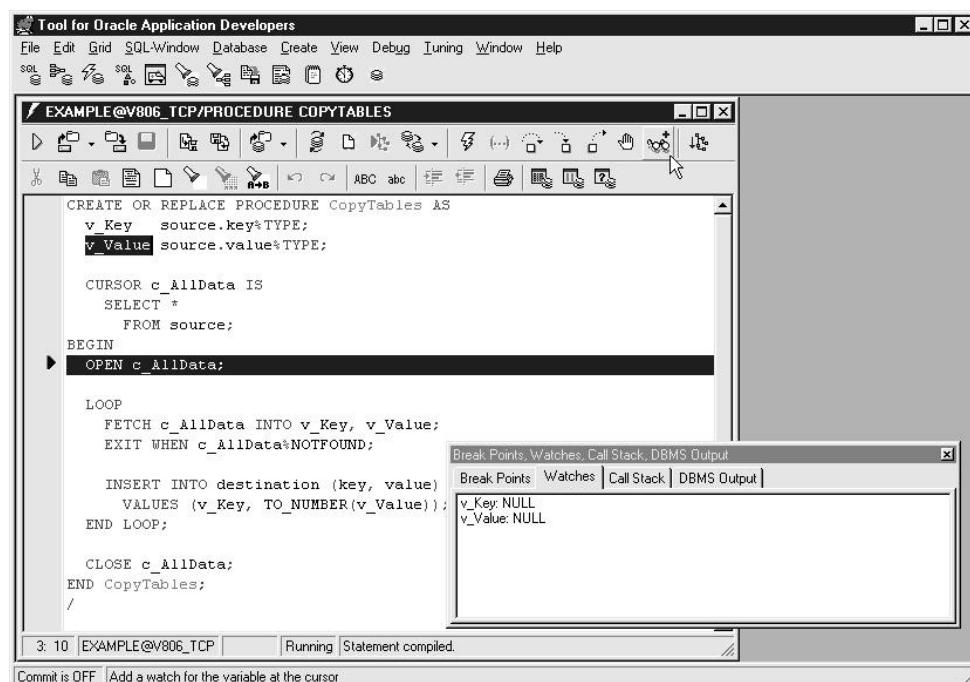


图3-17 设置观察点

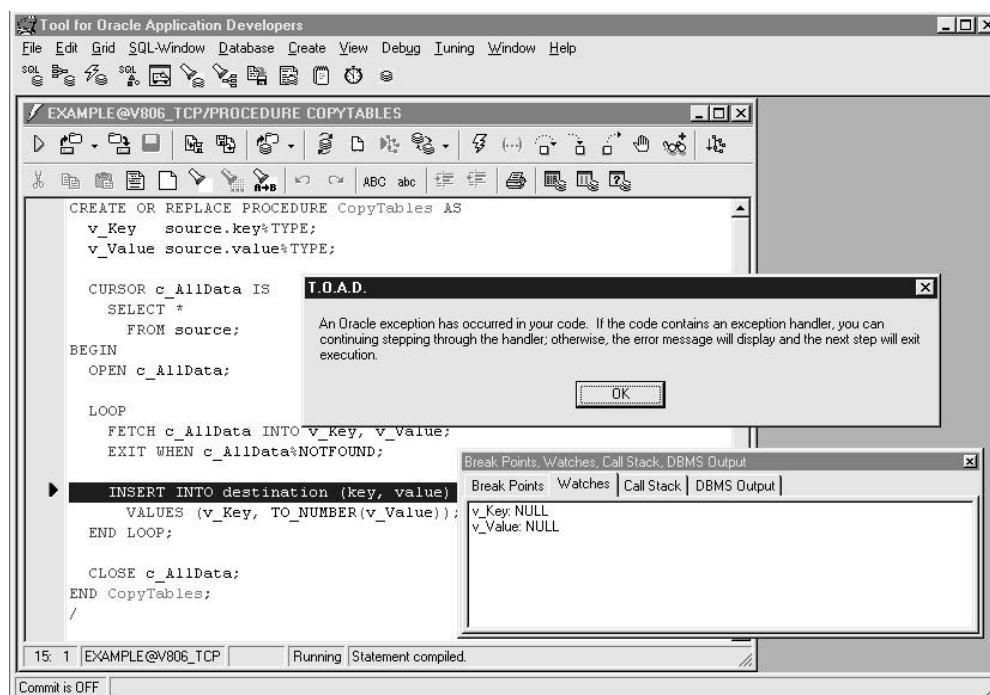


图3-18 停止在错误处的调试窗口

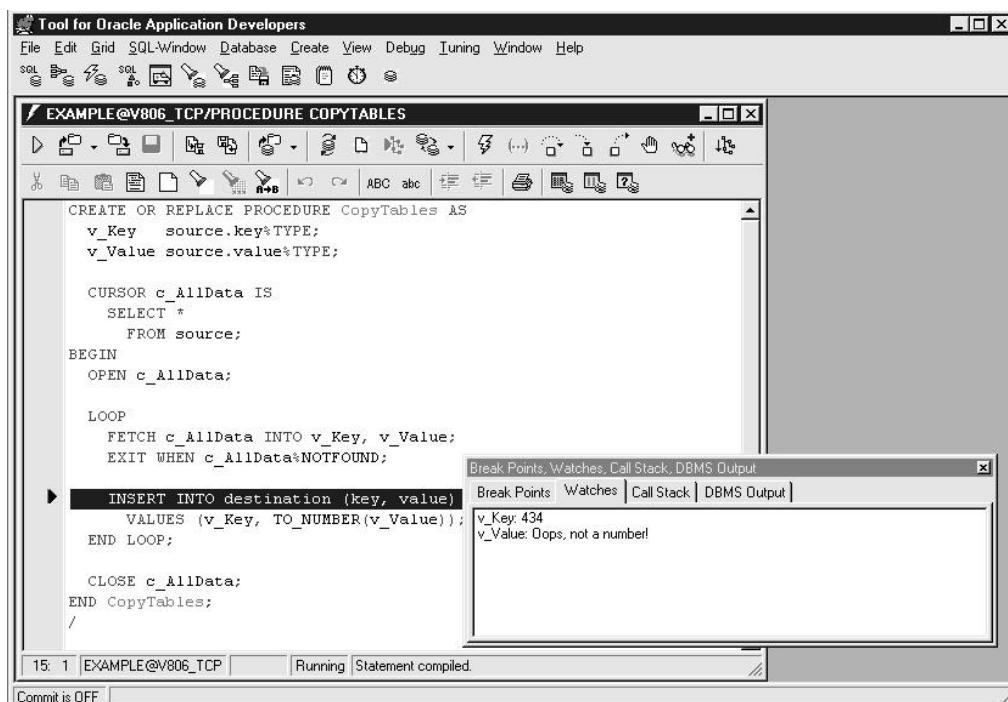


图3-19 有错误的数据

2. 问题6:评论

尽管上述只是一个简单的问题，但是，数据可能在更复杂、更微妙的方式下处于非法状态。在上述情况下，数据的类型有错误，因此不能进行类型转换。另外，也存在着数据的值超出取值范围的问题。请注意非法数据并不是每次都会引发异常，异常的引发取决于程序中的错误处理机制和数据不相容的性质，不同的情况会出现不同的结果。

值得注意的是，在上述情况下，PL/SQL代码本身并没有问题。问题的存在与代码无关，它只与代码处理的数据有关。因此，我们可以通过查询数据库表来发现错误数据，而不必对程序进行调试。

3.3.6 问题7

请看下面的过程：

节选自在线代码RSLoop1.sql

```

CREATE OR REPLACE PROCEDURE RSLoop AS
  v_RSRec registered_students%ROWTYPE;
  CURSOR c_RSGrades IS
    SELECT *
      FROM registered_students
     ORDER BY grade;
BEGIN
  -- Loop over the cursor to determine the last row.

```

```

FOR v_RSRec IN c_RSGrades LOOP
  NULL;
END LOOP;

-- And print it out.
DBMS_OUTPUT.PUT_LINE(
  'Last row selected has ID ' || v_RSRec.student_id);
END RSLoop;

```

过程RSLoop查询表registered_students(该表按分数排序)，并打印从最后一行取出的ID。然而，当我们运行该程序时，其执行结果为空 (NULL)：

```

SQL> exec RSLoop;
Last row selected has ID
PL/SQL procedure successfully completed.

```

1. 问题7: 使用SQL-Programmer进行调试

调试该程序的第一步是在如图 3-20 中所示的开发窗口中打开过程 RSLoop。接着，我们可以使用单步按钮来调试该过程代码，图 3-21 演示了处于调试状态的窗口。

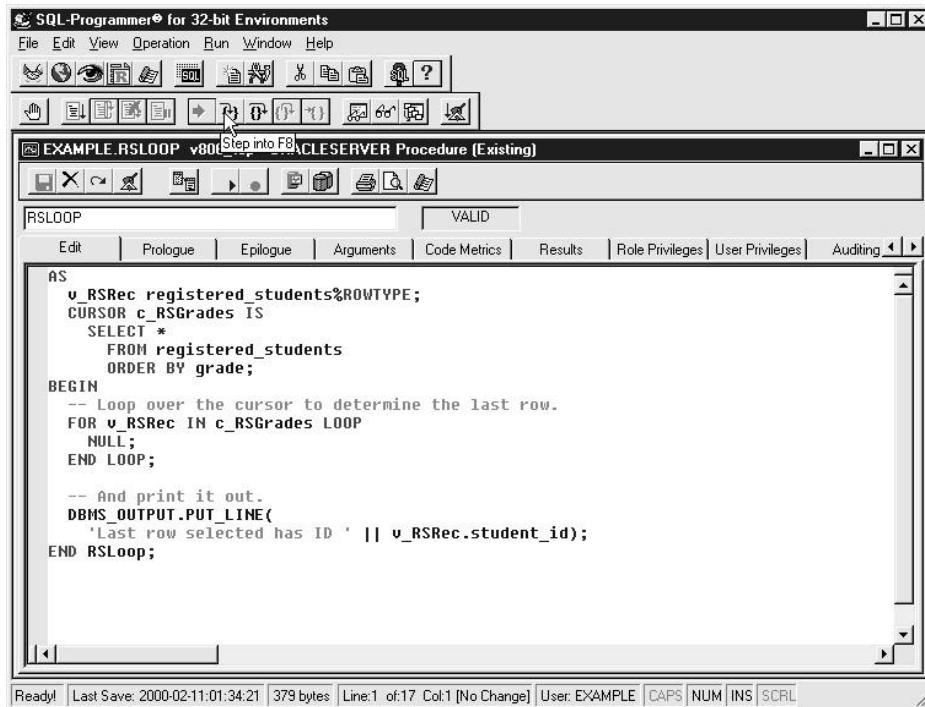


图3-20 准备调试过程RSLoop的窗口

下一步是为变量v_RSRec设置观察点。其做法是该代码中选中该变量，然后将其拖入观察窗口中，如图3-22所示。我们可以看到，由于我们刚启动该过程，该变量的值为空 (NULL)。当单步调试代码时，我们应该能够看到在循环过程中变量v_RSRec的值每次都在变化。但实际上，

如图3-23所示的观察窗口所显示的，该变量始终为空值。从该窗口中，我们还可以看到程序处理到第八行，但变量v_RSRec的值仍然为空。

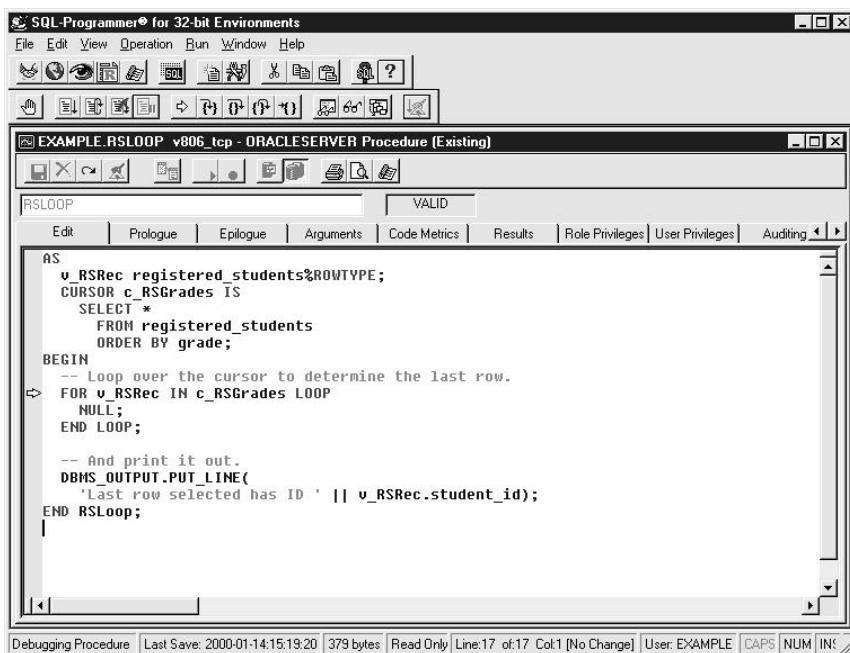


图3-21 停在过程代码第一行的调试窗口

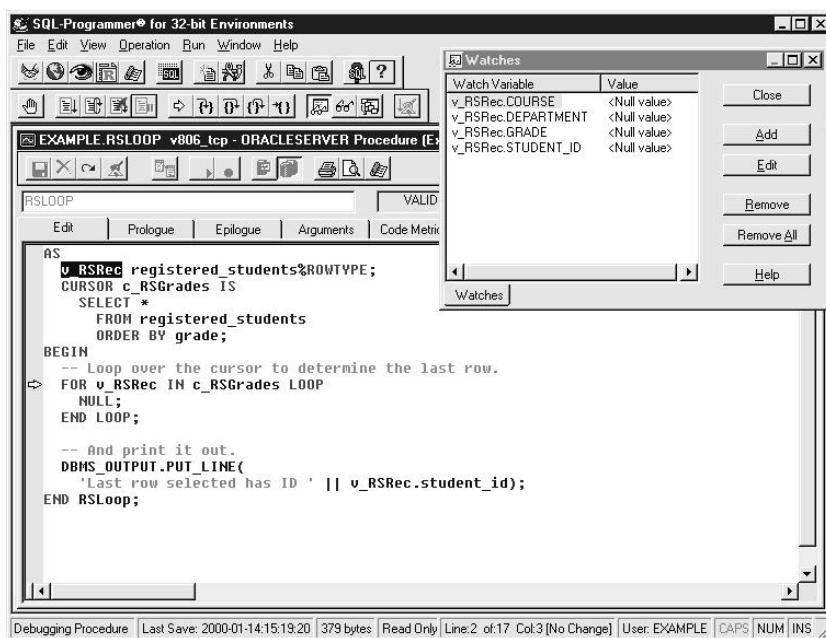


图3-22 观察变量v_RSRec的调试窗口

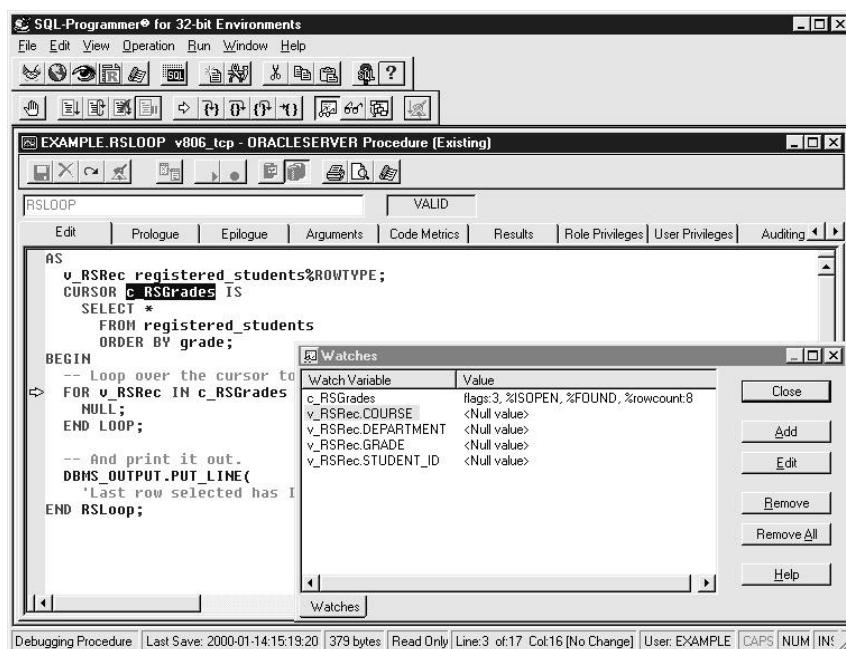


图3-23 显示v_RSRec仍然为空的观察窗口

为什么该程序的循环没有对v_RSRec进行赋值呢？问题出在该循环声明了一个隐式变量，其名称也叫v_RSRec。在循环体内部，隐式声明的变量把显式声明的变量隐藏起来，因此该变量不能赋值。一种修改的方法是把该循环用显式FETCH循环取代，如下所示，这种循环将使用显示声明的变量：

```
节选自在线代码RSLoop2.sql
CREATE OR REPLACE PROCEDURE RSLoop AS
    v_RSRec registered_students%ROWTYPE;
    CURSOR c_RSGrades IS
        SELECT *
        FROM registered_students
        ORDER BY grade;
    BEGIN
        -- Loop over the cursor to determine the last row.
        OPEN c_RSGrades;
        LOOP
            FETCH c_RSGrades INTO v_RSRec;
            EXIT WHEN c_RSGrades%NOTFOUND;
        END LOOP;
        CLOSE c_RSGrades;
        -- And print it out.
        DBMS_OUTPUT.PUT_LINE(
            'Last row selected has ID ' || v_RSRec.student_id);
    END RSLoop;
```

```
SQL> exec RSLoop
Last row selected has ID 10006
PL/SQL procedure successfully completed.
```

2. 问题7：评论

隐式声明的循环变量（包括游标 For 循环的记数器和数值 FOR 循环的循环记数器）的作用域仅限制在循环期间，这些变量在循环体内可以将同名变量隐藏起来。我们还可以通过将隐式变量换名，并在循环内将其值赋予显式声明的变量来解决该问题。

3.4 跟踪和配置

到目前为止，我们所讨论的调试技术可以用来指出应用程序中特殊问题的原因，然而，这些技术还不能处理所有可能遇到的程序问题，如程序的性能问题。为了弥补这种缺点，PL/SQL提供了几种不同的跟踪和配置工具。跟踪应用程序的结果是生成一份显示应用中调用子程序和所发生异常的清单。配置功能使在跟踪应用程序产生的报告中增加了时序信息。

基于事件的PL/SQL跟踪功能是在Oracle7.3.4版本中首次提供的；Oracle数据库的Oracle8i第一版（8.1.5）提供了跟踪PL/SQL API功能；而提供配置功能的是Oracle8i第二版（8.1.6）。我们将在下面讨论这些调试方法。

我们在下面几节使用的所有案例都将使用如下的过程和包。其中包 Random在本书的第4章中。

```
节选自在线代码traceDemo.sql
-- Returns fib(n), equivalent to fib(n-1) + fib(n-2).
CREATE OR REPLACE FUNCTION RecursiveFib(n IN BINARY_INTEGER)
  RETURN BINARY_INTEGER AS
BEGIN
  IF n = 0 OR n = 1 THEN
    RETURN n;
  ELSE
    RETURN RecursiveFib(n - 1) + RecursiveFib(n - 2);
  END IF;
END RecursiveFib;

CREATE OR REPLACE FUNCTION IterativeFib(n IN BINARY_INTEGER)
  RETURN BINARY_INTEGER AS
  v_Result BINARY_INTEGER;
  v_Sum1 BINARY_INTEGER := 1;
  v_Sum2 BINARY_INTEGER := 1;
BEGIN
  IF n = 1 OR n = 2 THEN
    RETURN 1;
  ELSE
    FOR v_Count IN 2..n - 1 LOOP
      v_Result := v_Sum1 + v_Sum2;
      v_Sum2 := v_Sum1;
      v_Sum1 := v_Result;
    END LOOP;
    RETURN v_Result;
  END IF;
END IterativeFib;
```

```
END LOOP;
RETURN v_Result;
END IF;
END IterativeFib;

CREATE OR REPLACE PROCEDURE RaiseIt(p_Exception IN NUMBER) AS
  e_MyException EXCEPTION;
BEGIN
  IF p_Exception = 0 THEN
    NULL;
  ELSIF p_Exception < 0 THEN
    RAISE e_MyException;
  ELSIF p_Exception = 1001 THEN
    RAISE INVALID_CURSOR;
  ELSIF p_Exception = 1403 THEN
    RAISE NO_DATA_FOUND;
  ELSIF p_Exception = 6502 THEN
    RAISE VALUE_ERROR;
  ELSE
    RAISE_APPLICATION_ERROR(-20001, 'Exception ' || p_Exception);
  END IF;
END RaiseIt;

CREATE OR REPLACE PROCEDURE CallRaise(p_Exception IN NUMBER := 0) AS
BEGIN
  RaiseIt(p_Exception);
EXCEPTION
  WHEN OTHERS THEN
    NULL;
END CallRaise;

CREATE OR REPLACE PROCEDURE RandomRaise(p_NumCalls IN NUMBER := 1) AS
  v_Case NUMBER;
BEGIN
  FOR v_Count IN 1..p_NumCalls LOOP
    v_Case := Random.RandMax(6);
    IF v_Case = 1 THEN
      CallRaise(-1);
    ELSIF v_Case = 2 THEN
      CallRaise(0);
    ELSIF v_Case = 3 THEN
      CallRaise(1001);
    ELSIF v_Case = 4 THEN
      CallRaise(1403);
    ELSIF v_Case = 5 THEN
      CallRaise(6502);
    ELSE
      CallRaise(v_Case);
    END IF;
  END LOOP;
END RandomRaise;
```

```
END LOOP;
END RandomRaise;

CREATE OR REPLACE PROCEDURE CallMe1 AS
BEGIN
    NULL;
END CallMe1;

CREATE OR REPLACE PROCEDURE CallMe2 AS
BEGIN
    NULL;
END CallMe2;

CREATE OR REPLACE PROCEDURE CallMe3 AS
BEGIN
    NULL;
END CallMe3;

CREATE OR REPLACE PACKAGE CallMe AS
    PROCEDURE One;
    PROCEDURE Two;
    PROCEDURE Three;
END CallMe;

CREATE OR REPLACE PACKAGE BODY CallMe AS
    PROCEDURE One IS
        BEGIN
            NULL;
        END One;

    PROCEDURE Two IS
        BEGIN
            NULL;
        END Two;

    PROCEDURE Three IS
        BEGIN
            NULL;
        END Three;
    END CallMe;

CREATE OR REPLACE PROCEDURE RandomCalls(p_NumCalls IN NUMBER := 1) AS
    v_Case NUMBER;
BEGIN
    FOR v_Count IN 1..p_NumCalls LOOP
        v_Case := Random.RandMax(6);
        IF v_Case = 1 THEN
            CallMe1;
```

```

ELSIF v_Case = 2 THEN
    CallMe2;
ELSIF v_Case = 3 THEN
    CallMe3;
ELSIF v_Case = 4 THEN
    CallMe.One;
ELSIF v_Case = 5 THEN
    CallMe.Two;
ELSE
    CallMe.Three;
END IF;
END LOOP;
END RandomCalls;

```

3.4.1 基于事件的跟踪

这种类型的跟踪功能需要设置数据库事件的允许或禁止标志。PL/SQL和RDBMS都提供了叫做数据库事件的调试工具。设置事件有以下两种方式：

- 在特别的会话中，使用 ALTER SESSION语句。其语法是：

```
ALTER SESSION SET EVENTS 'event event_string';
```

其中event是事件号，event_string描述了设置该事件的方式。使用上述命令，该事件只与这个特别的会话建立连接，并不会影响其他的数据库会话。

- 对于全部数据库的设置，在数据库初始化文件中（init.ora）使用如下所示的参数：

```
event="event event_string"
```

其中，event是事件号，event_string用于描述该事件的设置方式。使用上述命令后，该事件在数据库被关闭并重启动后，将与所有数据库会话建立连接。

不同的事件可以启动不同种类的事件跟踪功能。在各种跟踪中，跟踪信息都被写入会话跟踪文件中，该文件存储在由数据库初始文件参数USER_DUMP_DEST指示的目录中。

提示 如果不知道USER_DUMP_DEST的值，你可以通过查询v\$database_parameters数据字典窗口来找到该值，也可以使用服务器管理器（Server Manager）的命令SHOW PARAMETERS，或使用SQL *Plus 8i及更高版本的工具来确认该值。

由于附加信息都将被写入到跟踪文件中，设置事件连接将不可避免地影响程序性能。数据库事件除了有在本节中描述的用途外，还可用于多种其他目的。设置额外事件将可能会带来的影响，因此事件的设置要在Oracle支持服务的指导下实施。

1. 跟踪特殊的错误

设置等价于某个特殊错误号的事件将导致数据库把该错误发生时的信息转储到跟踪文件中，特别是，跟踪文件将包括发生错误时的当前SQL语句和调用队栈。为了设置具有这种功能的事件，应使用下列事件字符串：

```
event_num trace name errorstack
```

其中，event_num是指定的错误号。例如，假设我们提交下列匿名块：

节选自在线代码event6502.sql

```
SQL> -- First set the events in the session
SQL> ALTER SESSION SET EVENTS '6502 trace name errorstack';
Session altered.
SQL> -- And then raise ORA-6502
SQL> BEGIN
 2   RaiseIt(6502);
 3 END;
 4 /
BEGIN
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "EXAMPLE.RAISEIT", line 13
ORA-06512: at line 2
```

该块将生成包括类似下面内容的跟踪文件：

```
*** SESSION ID:(7.4) 2000-01-10 17:54:08.710
*** 2000-01-10 17:54:08.710
ksedmp: internal or fatal error
ORA-06502: PL/SQL: numeric or value error
Current SQL statement for this session:
BEGIN
  RaiseIt(6502);
END;
----- PL/SQL Call Stack -----
 object    line    object
 handle    number   name
 802da2a0     2      anonymous block
```

该跟踪文件指出当前的SQL语句是调用RaiseIt的匿名块。同时，该文件还包括了PL/SQL调用栈，该调用栈也指出了匿名块是可能出错的地方。然而，真正的错误发生在RaiseIt内部，而不是在匿名块中。为什么该调用栈不能提供发生错误时完整的PL/SQL栈呢？为了回答这个问题，我们要进一步介绍跟踪文件生成的过程。

当一个PL/SQL块发送到服务器运行时，服务器的影子进程（Shadow process）将其作为PL/SQL块接收。这时，该块被送往PL/SQL引擎执行，而不是提交给SQL语句执行器。当PL/SQL引擎返回时，所有的处理结果都将送回到客户端。如果PL/SQL引擎返回错误（如上面叙述的情况），服务器将在事件建立了连接的情况下，把这些错误信息写入跟踪文件。在该服务器记录错误信息时，该错误已经从内部过程传播到了匿名块中。因此，调用栈仅指示该匿名块有错。

注意 以这种方式生成的跟踪文件还包括了其他信息，如C的调用栈和CPU寄存器的转储信息，虽然，这类信息对确定源代码中出现的错误没有什么作用。但是它们对定位Oracle代码中发生的错误非常有用。

2. 调用和异常跟踪

这一级别的跟踪功能只能在 Oracle7 版本的 7.3.4 以及 Oracle8 版的 8.0.5 和更高版本中使用。低于 Oracle8 版 8.0.5 的版本都不具备该功能。

基于事件的调用和异常的跟踪允许用户跟踪 PL/SQL 的三类事件：调用存储子程序，引发异常和连接变量的值。该类跟踪可以在指定的事件发生时把输出记录到跟踪文件中，或者把跟踪数据存储在循环缓冲区中，并在一定的条件下进行转储。使用循环缓冲区的好处是可以限制跟踪文件的容量过大。

事件级别 为了启动调用和异常跟踪，可以使用 event 10938 来实现。该事件字符串的语法是：

```
10938 trace name context level level_num
```

其中，level_num 是下面列出的值按位“或”(OR)。

跟踪名称	十六进制值	十进制值	注释
TRACE_ACALL	0x0001	1	跟踪所有调用
TRACE_ECALL	0x0002	2	跟踪允许的调用
TRACE_AEXCP	0x0004	4	跟踪所有的异常
TRACE_EEXCP	0x0008	8	跟踪允许的异常
TRACE_CIRCULAR	0x0010	16	使用环形缓冲区
TRACE_BIND_VARS	0x0020	32	跟踪连接变量

表中所述的允许调用是已经用 DEBUG 选项进行编译的子程序，允许异常则是从允许的子程序中引发的异常（详细介绍请看下文的“允许调用和异常”）。为了计算所希望的级别，可以取需要跟踪类型的按位或的值作为跟踪级别（等于各个跟踪类型的十进制数的和）。例如：

- 级别 17(ACALL | CIRCULAR) 可以跟踪所有的调用，并使用环形缓冲区。
- 级别 22(ECALL | AEXCP) 将跟踪允许的调用和所有的异常，使用环形缓冲区。
- 级别 32(BIND_VARS) 将跟踪连接变量，不使用缓冲区。
- 级别 53(ACALL | AEXCP | CIRCULAR | BIND_VARS) 将是最高级别的跟踪，并使用缓冲区。
- 级别 37(ACALL | AEXCP | BIND_VARS) 是最高级别的跟踪，不使用缓冲区。

假设，我们把下列 SQL 语句提交给数据库：

节选自在线代码 AllCallsExceptions.sql

```
SQL> -- Enable tracing of all calls and all exceptions. 5 is the
SQL> -- bitwise OR of 0x01 and 0x04.
SQL> ALTER SESSION SET EVENTS '10938 trace name context level 5';
Session altered.

SQL> -- Anonymous block which raises some exceptions.
SQL> BEGIN
 2   CallRaise(1001);
 3   RaiseIt(-1);
 4 END;
 5 /
BEGIN
```

```
*  
ERROR at line 1:  
ORA-06510: PL/SQL: unhandled user-defined exception  
ORA-06512: at "EXAMPLE.RAISEIT", line 7  
ORA-06512: at line 3
```

该命令将在跟踪文件中生成下列输出：

```
----- PL/SQL TRACE INFORMATION -----  
Levels set : 1 4  
Trace: ANONYMOUS BLOCK: Stack depth = 1  
Trace: PROCEDURE EXAMPLE.CALLRAISE: Call to entry at line 3 Stack depth = 2  
Trace: PROCEDURE EXAMPLE.RAISEIT: RAISEIT Stack depth = 3  
Trace: Pre-defined exception - OER 1001 at line 9 of PROCEDURE EXAMPLE.RAISEIT:  
Trace: PROCEDURE EXAMPLE.RAISEIT: RAISEIT Stack depth = 2  
Trace: User defined exception at line 7 of PROCEDURE EXAMPLE.RAISEIT:
```

对服务器的每个调用都将生成与上面类似的输出，这些输出具有下列特点：

- 在位于跟踪文件的开始处，打印“PL/SQL TRACE INFORMATION”，接着是当前设置的跟踪级别。如上所示的跟踪文件中指出的级别 1 和 4 表示允许 TRACE_ACALL 和 TRACE_AEXCP 跟踪。
- 在开始行之后，当跟踪事件发生时，就在该文件中增加一行信息。在上面的例子中，由于允许 TRACE_ACALL 和 TRACE_AEXCP 跟踪，因此，每当调用一个子程序或发生异常时，就在跟踪文件登录一项。
- 对于子程序调用，该文件中打印了堆栈的长度。其中的文本信息也显示了堆栈的长度。随着堆栈的延伸，跟踪文件的每一行的长度也在增加。跟踪文件的每行的最大长度为 512 个字符，这就限制了堆栈的最大长度。
- 每当异常发生时，跟踪文件中都有记录。异常的记录与处理异常的程序所在的块无关。

在下面几节中，我们将会看到不同类型的跟踪文件的例子。

允许调用和异常 我们在上面提到过允许子程序是用 DEBUG 选项进行编译的子程序。指定调用和异常的方法有两个，第一个是提交下面的语句：

```
ALTER SESSION SET PLSQL_DEBUG= TRUE;
```

执行该语句后，任何 PL/SQL 块或子程序都将用 DEBUG 参数进行编译。也可以使用下面的语句重编存储子程序：

```
ALTER [PROCEDURE | FUNCTION | PACKAGE BODY | TYPE BODY ]      object_name COMPILE  
DEBUG;
```

匿名块只可以通过提交 ALTER SESSION 语句使用 DEBUG 项进行编译。例如，我们可以向数据库提交下列 SQL 语句：

节选自在线代码 EnabledCallsExceptions.sql

```
SQL> -- Enable tracing of enabled calls and exceptions. 10 is the  
SQL> -- bitwise OR of 0x02 and 0x08.  
SQL> ALTER SESSION SET EVENTS '10938 trace name context level 10';  
Session altered.
```

```
SQL> ALTER PROCEDURE RaiseIt COMPILE DEBUG;
Procedure altered.

SQL> -- Anonymous block which raises some exceptions.
SQL> BEGIN
 2   CallRaise(1001);
 3   RaiseIt(-1);
 4 END;
 5 /
BEGIN
*
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at "EXAMPLE.RAISEIT", line 7
ORA-06512: at line 3
```

RaiseIt是唯一允许的块。因此，跟踪文件只显示下列内容：

```
----- PL/SQL TRACE INFORMATION -----
Levels set : 2 8
Trace:      PROCEDURE EXAMPLE.RAISEIT: RAISEIT Stack depth = 3
Trace:      Pre-defined exception - OER 1001 at line 9 of PROCEDURE EXAMPLE.RAISEIT:
Trace:      PROCEDURE EXAMPLE.RAISEIT: RAISEIT Stack depth = 2
Trace:      User defined exception at line 7 of PROCEDURE EXAMPLE.RAISEIT:
```

该文件中登录的入口都来自于RaiseIt。如果在非允许的块中（没有使用DEBUG选项编译过的）引发了异常，该异常不在跟踪文件中记录。

跟踪对打包的子程序调用 当调用一个打包子程序时，该包内指定的子程序将不记录在跟踪文件中。例如，假设我们在SQL *Plus下提交下列匿名块：

节选自在线代码PackgeCalls.sql

```
SQL> -- Enable tracing of all calls and all exceptions. 5 is the
SQL> -- bitwise OR of 0x01 and 0x04.
SQL> ALTER SESSION SET EVENTS '10938 trace name context level 5';
Session altered.
```

```
SQL> -- Anonymous block which calls packaged procedures.
SQL> BEGIN
 2 CallMe.One;
 3 CallMe.Two;
 4 CallMe.Three;
 5 END;
 6 /
PL/SQL procedure successfully completed.
```

该块将在跟踪文件中生成下列的内容。

```
----- PL/SQL TRACE INFORMATION -----
Levels set : 1 4
Trace:  ANONYMOUS BLOCK: Stack depth = 1
Trace:  PACKAGE BODY EXAMPLE.CALLME: Call to entry at line 4 Stack depth = 2
```

```
Trace: PACKAGE BODY EXAMPLE.CALLME: Call to entry at line 9 Stack depth = 2
Trace: PACKAGE BODY EXAMPLE.CALLME: Call to entry at line 14 Stack depth = 2
```

从该跟踪文件中可以看出，除了每个登录行有包体的名称外，还提供了有关该行的信息，利用这些信息，我们可以识别指定的打包子程序。

连接变量 如果设置了调试级别 TRACE_BIND_VARS的话，连接变量的信息就会被记录在跟踪文件中。其实现方法是在 PL/SQL得到连接变量信息时，为每个连接变量发生的事件在跟踪文件中记录一行。连接变量信息的打印与它所在的块是否已建立了事件连接无关。下面的 SQL *Plus的会话将显示包含连接变量的PL/SQL块。

```
节选自在线代码BindVariables.sql
SQL> -- First set up the variables
SQL> VARIABLE v_String1 VARCHAR2(20);
SQL> VARIABLE v_String2 VARCHAR2(20);
SQL> BEGIN
2   :v_String1 := 'Hello';
3   :v_String2 := ' World!';
4 END;
5 /
PL/SQL procedure successfully completed.
```

```
SQL> -- Enable tracing for all calls and bind variables.
SQL> ALTER SESSION SET EVENTS '10938 trace name context level 33';
Session altered.
SQL> BEGIN
2   DBMS_OUTPUT.PUT_LINE(:v_String1 || :v_String2);
3 END;
4 /
Hello World!
PL/SQL procedure successfully completed.
```

该块生成类似于下面的跟踪文件。

```
----- PL/SQL TRACE INFORMATION -----
Levels set : 1 32
Trace: ANONYMOUS BLOCK: Stack depth = 1
op: GBVAR; pos: 1; buf: 10bd7b8; len: 5; ind: 0; bfl: 20;
op: GBVAR; pos: 2; buf: 10bd7d8; len: 7; ind: 0; bfl: 20;
Trace: PACKAGE BODY SYS.DBMS_OUTPUT: Call to entry at line 1 Stack depth = 2
Trace: PACKAGE BODY SYS.DBMS_OUTPUT: Call to entry at line 1 Stack depth = 3
Trace: PACKAGE BODY SYS.STANDARD: Call to entry at line 793 Stack depth = 4
Trace: PACKAGE BODY SYS.STANDARD: ICD vector index = 45 Stack depth = 4
Trace: PACKAGE BODY SYS.STANDARD: Call to entry at line 564 Stack depth = 5
```

除了调用 DBMS_OUTPUT（该包依次再调用包 STANDARD），上面的跟踪信息还显示 PL/SQL接收了使用伪操作码 GBVAR的两个连接变量。

提示 上述类型的跟踪不显示连接变量的值，而只是报告连接变量的类型和长度。我们可以通过把 event 10046的跟踪级别设置为 4来显示连接变量的值。这类事件还将生成

SQL_TRACE信息。我们将在3.4.1节中专门介绍这种跟踪方法。

使用环形缓冲区 当跟踪调用时，特别是跟踪长时间运行的程序时，跟踪文件有可能变的非常巨大。一般来说，只有这种文件的最后部分对调试有用，这是因为该部分包括了程序运行的最后信息。为了保留这部分信息，PL/SQL提供了环形结构的缓冲区。在这种记录方式下，调试信息不再直接送往跟踪文件，而是先发送到该缓冲区中存放。当该缓冲区装满时，后续的输出信息将覆盖该缓冲区的开始部分。因此，该缓冲区中将保留跟踪数据的最后一部分。最后，该缓冲区的内容可以按不同的条件转储到跟踪文件中。该环形缓冲区受到下列三个参数和事件的控制：

- TRACE_CIRCULAR位必须与event 10938事件一起设置。该设置将把跟踪信息送到缓冲区，而不直接写入跟踪文件。
- 环形缓冲区的容量要使用下面的事件字符串与event 10940一同设置：

```
10940 trace name context level buffer_size
```

其中，buffer_size是该缓冲区的以K为单位的容量（1KB=1024字节），默认值是8KB。该事件既可以使用ALTER SESSION语句设置，也可在数据库初始文件中设置。

- 将缓冲区的内容送往跟踪文件的条件由数据库初始参数 _PLSQL_DUMP_BUFFER_EVENTS指定（请注意开始的下划线）。可以设置的事件如下所示，请注意转储事件由逗号分隔，所有的事件都要使用大写字母，中间没有空格：

转储事件	说明
ON_EXIT	只要PL/SQL解释程序退出，如调用结束，缓冲区的内容将被转储。
错误号	只要发生指定的错误，缓冲区的内容将转储。该错误必须是一个运行时错误，而不是编译错误。
ALL_EXCEPTIONS	只要发生错误，缓冲区就进行转储。

下面的列表演示了某些合法的转储参数设置：

- _PLSQL_DUMP_BUFFER_EVENTS="1,6502,1001"将在引发ORA_1,ORA_1001,ORA_6502错误时将缓冲区的内容转储到跟踪文件。
- _PLSQL_DUMP_BUFFER_EVENTS="ON_EXIT,6502"在解释程序退出并且引发ORA_6502错误时，转储缓冲区的内容。
- _PLSQL_DUMP_BUFFER_EVENTS="ALL_EXCEPTIONS,ON_EXIT"在解释程序退出并且引发任何错误时，转储缓冲区内容。

3. 伪指令跟踪

该级别的跟踪适用于所有版本的PL/SQL。这种跟踪将把所有的PL/SQL伪指令操作的输出以及源代码的行号（如果带有行号的话）在其运行时送往跟踪文件。伪指令操作类似于汇编语言指令，它们是由PL/SQL编译器生成的机器代码并由PL/SQL的运行引擎执行。尽管伪指令本身并没有记录下来，但这种类型的跟踪对确认PL/SQL执行的行信息很有帮助。除此之外，伪指令也对Oracle系统自身调试非常有益。

伪指令跟踪的启动要求把event 10928的级别设置为0级以上，并使用下面的事件字符串语法：

```
10928 trace name context level 1
```

该事件可以使用ALTER SESSION语句来设置会话级别，或者在数据库初始化文件中设置全数据库通用。该类跟踪不使用环形缓冲区，因此，所有的跟踪信息都将写入跟踪文件。这将导致该文件过大，这时要求主机提供足够的硬盘空间。

注意 跟踪文件的最大容量可以使用初始化参数MAX_DUMP_FILE_SIZE来设置。当跟踪文件达到该容量时，就不在对该文件进行写入操作。

例如，请考虑下面的匿名块的情况：

节选自在线代码PseudoCode.sql

```
ALTER SESSION SET EVENTS '10928 trace name context level 1';

BEGIN
    CallMe1;
    CallMe2;
    CallRaise(100);
END;
```

在SQL *Plus中运行该块将在跟踪文件中记录下列输出信息：

```
*** SESSION ID:(12.4415) 2000-03-03 13:45:11.164
Entry #1
00001: ENTER 44, 0, 1, 1
00009: INFR DS[0]+36
    Frame Desc Version = 1, Size = 19
    # of locals = 1
    TC_SSCLAR: FP+8, d=FP+16, n=FP+40
<source not available>
00014: INSTB 1, STPROC
00018: XCAL 1, 1
Entry #1
EXAMPLE.CALLME1: 00001: ENTER 4, 0, 1, 1
[Line 4]
[Line 4] END CallMe1;
EXAMPLE.CALLME1: 00009: RET
<source not available>
00023: INSTB 2, STPROC
00027: XCAL 2, 1
Entry #1
EXAMPLE.CALLME2: 00001: ENTER 4, 0, 1, 1
[Line 3] NULL;
EXAMPLE.CALLME2: 00009: RET
<source not available>
00032: CVTIN HS+0 =100=, FP+8
00037: INSTB 4, STPROC
00041: MOVA FP+8, FP+4
00046: XCAL 4, 1
Entry #1
EXAMPLE.CALLRAISE: 00001: ENTER 8, 0, 1, 1
```

```
[Line 3] RaiseIt(p_Exception);
EXAMPLE.CALLRAISE: 00009: INSTB 2, STPROC
EXAMPLE.CALLRAISE: 00013: MOVA AP[4], FP+4
EXAMPLE.CALLRAISE: 00018: XCAL 2, 1
Entry #1
EXAMPLE.RAISEIT: 00001: ENTER 228, 0, 1, 1
EXAMPLE.RAISEIT: 00009: INFR DS[0]+120
Frame Desc Version = 1, Size = 52
# of locals = 6
TC_SSCLAR: FP+16, d=FP+80, n=FP+104
TC_SSCLAR: FP+24, d=FP+108, n=FP+132
TC_SSCLAR: FP+32, d=FP+136, n=FP+160
TC_SSCLAR: FP+40, d=FP+164, n=FP+188
TC_SSCLAR: FP+48, d=FP+192, n=FP+216
TC_VCHAR: FP+60, d=FP+220, n=FP+224, mxl=4000, CS_IMPLICIT
[Line 4] IF p_Exception = 0 THEN
EXAMPLE.RAISEIT: 00014: CVTIN HS+0 =0=, FP+16
EXAMPLE.RAISEIT: 00019: CMP3N AP[4], FP+16, PC+22 =00041:=
EXAMPLE.RAISEIT: 00029: BRNE PC+12 =00041:=
[Line 6] ELSIF p_Exception < 0 THEN
EXAMPLE.RAISEIT: 00041: CVTIN HS+0 =0=, FP+24
EXAMPLE.RAISEIT: 00046: CMP3N AP[4], FP+24, PC+27 =00073:=
EXAMPLE.RAISEIT: 00056: BRGE PC+17 =00073:=
[Line 8] ELSIF p_Exception = 1001 THEN
EXAMPLE.RAISEIT: 00073: CVTIN HS+8 =1001=, FP+32
EXAMPLE.RAISEIT: 00078: CMP3N AP[4], FP+32, PC+27 =00105:=
EXAMPLE.RAISEIT: 00088: BRNE PC+17 =00105:=
[Line 10] ELSIF p_Exception = 1403 THEN
EXAMPLE.RAISEIT: 00105: CVTIN HS+16 =1403=, FP+40
EXAMPLE.RAISEIT: 00110: CMP3N AP[4], FP+40, PC+27 =00137:=
EXAMPLE.RAISEIT: 00120: BRNE PC+17 =00137:=
[Line 12] ELSIF p_Exception = 6502 THEN
EXAMPLE.RAISEIT: 00137: CVTIN HS+24 =6502=, FP+48
EXAMPLE.RAISEIT: 00142: CMP3N AP[4], FP+48, PC+27 =00169:=
EXAMPLE.RAISEIT: 00152: BRNE PC+17 =00169:=
[Line 15] RAISE_APPLICATION_ERROR(-20001, 'Exception ' || p_Exception);
EXAMPLE.RAISEIT: 00169: CVTNC AP[4], FP+60
EXAMPLE.RAISEIT: 00174: CONC3 HS+32='Exception '=, FP+60, FP+56
EXAMPLE.RAISEIT: 00181: INSTS 2
EXAMPLE.RAISEIT: 00184: INSTB 2, SPEC
EXAMPLE.RAISEIT: 00188: MOVA HS+56, FP+4
EXAMPLE.RAISEIT: 00193: MOVA FP[56], FP+8
EXAMPLE.RAISEIT: 00198: MOVA HS+0, FP+12
EXAMPLE.RAISEIT: 00203: ICAL 2, 1, 1, 3
Exception handler: OTHER Line 3-3. PC 9-28.
[Line 6] NULL;
EXAMPLE.CALLRAISE: 00029: CLREX
```

```
EXAMPLE.CALLRAISE: 00030: BRNCH PC+6 =00036:=
EXAMPLE.CALLRAISE: 00036: RET
00051: RET
Entry #1
00001: ENTER 212, 0, 1, 1
00009: INFR DS[0]+32
    Frame Desc Version = 1, Size = 29
    # of locals = 1
    _TC_iVCHAR: FP+32, d=FP+196, n=FP+208, mxl=0, CS_IMPLICIT
    # of bind proxies = 1
    _TC_iVCHAR: FP+12, d=FP+52, n=FP+192, ubn(mxl)=128, CS_IMPLICIT
<source not available>
00014: GBVAR SQLT_CHR(1), 1, FP+12
00021: INSTS 4
00024: INSTB 4, SPEC_BODY
00028: MOVA FP+12, FP+4
00033: MOVA FP+32, FP+8
00038: XCAL 4, 1
Entry #1
SYS.DBMS_APPLICATION_INFO: 00001: ENTER 12, 1, 1, 1
[shrink-wrapped frame]
SYS.DBMS_APPLICATION_INFO: 00009: MOVA AP[4], FP+4
SYS.DBMS_APPLICATION_INFO: 00014: MOVA AP[8], FP+8
SYS.DBMS_APPLICATION_INFO: 00019: ICAL 0, 7, 1, 2
SYS.DBMS_APPLICATION_INFO: 00028: RET
00043: RET
```

连同伪指令本身，上面的输出信息显示了被编译入伪指令的源程序行。该跟踪文件描述了下述事件序列：

- 1) 匿名块的入口。该事件由指令 ENTER 指示。该块的代码不在显示之列，因为该代码没有存储在数据库中，这时跟踪信息中将显示 <source not available>。
- 2) CallMe1入口和立即返回。由于 CallMe1存储在数据库中，所以，我们在跟踪文件中可以看到该源代码的行。在源程序行之后，显示了另外一个 <source not available>，指示将返回到匿名块中。
- 3) CallMe2入口和立即返回。在这里，我们再次看到了源程序行，并返回到匿名块中。
- 4) 进入CallRaise的入口和下面的RaiseIt入口。每次调用都显示源代码。
- 5) 单步进入 RaiseIt的IF THEN 语句，使用相关联的伪指令进行每次测试。测试在调用 RAISE_APPLICATION_ERROR处停止并进入异常处理程序。
- 6) 从异常处理和匿名块退出。
- 7) 现在我们看到了 SQL *Plus 自身提交了另外一个 PL/SQL块，由该块调用 DBMS_APPLICATION_INFO。由于该包被覆盖，所以其源代码无法显示。但是，我们可以在跟踪信息中看到伪指令。（并不是所有版本的 SQL *Plus都可以提供该调用，在这种情况下，跟踪文件没有此项内容。）

其他伪指令的含义在表 3-2 中说明。即使不知道每个伪指令的确切含义，然而，我们也可以从该级别的跟踪信息中了解 PL/SQL 程序的处理过程。

表3-2 PL/SQL伪指令

伪 指 令	功 能 描 述
BR*	以BR开始的（如BRNE）伪指令是分枝指令
CALL,XCAL,SCAL,ICALL	调用当前块或块外部的过程。根据所调用的过程的位置，使用不同的伪指令
ENTER	栈帧入口（如匿名块或过程的入口）
GBVAR , SBVAR , GBCR	处理PL/SQL连接变量
MOV*	以MOV开始的伪指令表示数据从一个位置移动到另一个位置，类似赋值语句
RET	从栈帧中返回

4. SQL跟踪

通过在会话中使用下面的语句设置 SQL_TRACE 为真值（TRUE）：

```
ALTER SESSION SET SQL_TRACE = TRUE;
```

所有SQL语句的信息和被送往服务器的PL/SQL的块都被转储到跟踪文件中。（接着，实用程序tkprof可以用来把这些信息转换为更可读的格式）该类信息还可以通过设置 event 10046 来实现，其事件字符串如下：

```
10046 trace name context forever,level level_num
```

可为该事件设置的跟踪级别如下所示：

跟 踪 级 别	说 明
1	与SQL_TRACE相同。
4	1级+连接变量信息。
8	1级+等待信息（可用于观察锁存等待，也可用于检测全表扫描。）
12	1级+连接变量+等待信息。

与我们在前面作为环形缓冲区跟踪的一部分介绍的连接变量跟踪不同，这类跟踪将显示送往服务器的所有连接变量的信息，而非只是显示 PL/SQL 块中的信息。除此之外，这种跟踪还可以显示变量自身的值。

例如，假设我们从 SQL *Plus 中发布下列 PL/SQL 块：

```
节选自在线代码SQLTrace.sql
SQL> -- First set up the variables
SQL> VARIABLE v_String1 VARCHAR2(20);
SQL> VARIABLE v_String2 VARCHAR2(20);
SQL>
SQL> BEGIN
2      :v_String1 := 'Hello';
3      :v_String2 := ' World!';
4 END;
5 /
PL/SQL procedure successfully completed.
```

```
SQL> -- Turn on SQL tracing (including bind variable information)
SQL> ALTER SESSION SET EVENTS '10046 trace name context forever, level 4';
Session altered.
```

```
SQL> BEGIN
 2   DBMS_OUTPUT.PUT_LINE(:v_String1 || :v_String2);
 3 END;
 4 /
Hello World!
PL/SQL procedure successfully completed.
```

该块将生成下列跟踪信息。

```
PARSING IN CURSOR #1 len=61 dep=0 uid=28 oct=47 lid=28 tim=0 hv=2809072883
ad='801d40b4'
BEGIN
  DBMS_OUTPUT.PUT_LINE(:v_String1 || :v_String2);
END;
END OF STMT
PARSE #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=0
BINDS #1:
bind 0: dty=1 mxl=32(20) mal=00 scl=00 pre=00 oacflg=03 oacf12=10 size=64
      offset=0 bfp=010bf728 bln=32 avl=05 flg=05
      value="Hello"
bind 1: dty=1 mxl=32(20) mal=00 scl=00 pre=00 oacflg=03 oacf12=10 size=0
      offset=32 bfp=010bf748 bln=32 avl=07 flg=01
      value=" World!"
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=1,dep=0,og=4,tim=0
=====
PARSING IN CURSOR #2 len=52 dep=0 uid=28 oct=47 lid=28 tim=0 hv=4201917273
ad='8010fdac'
begin dbms_output.get_lines(:lines, :numlines); end;
END OF STMT
PARSE #2:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=0
BINDS #2:
bind 0: dty=1 mxl=2000(255) mal=25 scl=00 pre=00 oacflg=43 oacf12=10 size=2000
      offset=0 bfp=010c63a0 bln=255 avl=00 flg=05
bind 1: dty=2 mxl=22(02) mal=00 scl=00 pre=00 oacflg=01 oacf12=0 size=24
      offset=0 bfp=010bf750 bln=22 avl=02 flg=05
      value=25
EXEC #2:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=1,dep=0,og=4,tim=0
=====
PARSING IN CURSOR #1 len=53 dep=0 uid=28 oct=47 lid=28 tim=0 hv=583813323
ad='80355540'
begin DBMS_APPLICATION_INFO.SET_MODULE(:1,NULL); end;
END OF STMT
PARSE #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=0
BINDS #1:
bind 0: dty=1 mxl=128(08) mal=00 scl=00 pre=00 oacflg=21 oacf12=0 size=128
```

```

offset=0 bfp=010bf6e8 bln=128 avl=08 flg=05
value="SQL*Plus"
APPNAME mod='SQL*Plus' mh=3669949024 act='' ah=4029777240
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=1,dep=0,og=4,tim=0

```

除了由脚本发布的匿名块外，该跟踪还显示由 SQL *Plus 自身提交的两个额外的块（一个用来检索和显示 DBMS_OUTPUT 信息，另一个来检索 DBMS_APPLICATION_INFO 的信息）。这三个块中都包括连接变量。该类跟踪中显示的连接变量的值如表 3-3 所示。这里所显示的大多数的字段是供内部使用的，其中，dty、mxi 和 value 字段是最有用的。

表3-3 连接变量跟踪中的字段说明

字 段	说 明
bind	界标位置（从 0 开始）
dty	数据类型，该类型对应在头文件 ocidfn.h 中发现的 OCI 数据类型
mxi	连接变量的最大长度。需要注意的是，该值可能会比字符串所需的长度要大一些，以便在共享缓冲池中共享游标数据。所需长度要在圆括号中
mal	数据连接的数组长度
scl	比例
pre	精度
oacflg.oacf12	内部标志
size	成组连接缓冲区
offset	成组连接缓冲区内部偏移量的总长度
bfp	连接缓冲区的地址
bln	连接缓冲区的长度
avl	实际变量长度
flg	内部标志
value	可见的变量值

5. 组合跟踪事件

我们在前几节介绍的所有基于事件的跟踪都在同一个跟踪文件中生成输出信息。因此，如果设置了一个以上的事件，则每个事件的输出将会交错出现在跟踪文件中。例如，如下所示，我们可以把调用跟踪和伪指令跟踪结合使用：

```

节选自在线代码CombinedTracing.sql
SQL> -- Set both the pseudo-code and call tracing events.
SQL> ALTER SESSION SET EVENTS '10928 trace name context level 1';
Session altered.

SQL> ALTER SESSION SET EVENTS '10938 trace name context level 1';
Session altered.

SQL> -- Make some random calls.
SQL> BEGIN
 2      RandomCalls(3);
 3  END;
 4 /
PL/SQL procedure successfully completed.

```

下面是跟踪文件中的一部分输出信息。该文件显示了两类跟踪信息。

```
----- PL/SQL TRACE INFORMATION -----
Levels set : 1
Entry #1
00001: ENTER 44, 0, 1, 1
Trace: ANONYMOUS BLOCK: Stack depth = 1
00009: INFR DS[0]+36
    Frame Desc Version = 1, Size = 19
    # of locals = 1
    TC_SSCLAR: FP+8, d=FP+16, n=FP+40
<source not available>
00014: CVTIN HS+0 =3=, FP+8
00019: INSTB 2, STPROC
00023: MOVA FP+8, FP+4
00028: XCAL 2, 1
Entry #1
EXAMPLE.RANDOMCALLS: 00001: ENTER 312, 0, 1, 1
Trace: PROCEDURE EXAMPLE.RANDOMCALLS: Call to entry at line 4 Stack depth = 2
...
Trace: PACKAGE BODY EXAMPLE.RANDOM: Call to entry at line 3 Stack depth = 3
EXAMPLE.RANDOM: 00009: INFR DS[0]+92
...
[Line 6]      IF v_Case = 1 THEN
EXAMPLE.RANDOMCALLS: 00072: CVTIN HS+0 =1=, FP+52
EXAMPLE.RANDOMCALLS: 00077: CMP3N FP+12, FP+52, PC+31 =00108:=
EXAMPLE.RANDOMCALLS: 00087: BRNE PC+21 =00108:=
[Line 7]      CallMe1;
EXAMPLE.RANDOMCALLS: 00093: INSTB 3, STPROC
EXAMPLE.RANDOMCALLS: 00097: XCAL 3, 1
Entry #1
EXAMPLE.CALLME1: 00001: ENTER 4, 0, 1, 1
Trace: PROCEDURE EXAMPLE.CALLME1: Call to entry at line 3 Stack depth = 3
[Line 3] NULL;
EXAMPLE.CALLME1: 00009: RET
```

3.4.2 基于PL/SQL的跟踪

Oracle 8 及
更高版本

事件10938中的调用和异常跟踪可以使用 Oracle8 PL/SQL中的包DBMS_TRACE实现。该类跟踪功能的进一步完善也将通过该包，而不是通过事件实现。Oracle8i第一版(8.1.5)中的包DBMS_TRACE提供了与基于事件跟踪同类的跟踪功能，而Oracle8i第二版(8.1.6)则显著地增强了跟踪功能。

1. Oracle8i第1版(8.1.5)的DBMS_TRACE

Oracle8i第1版(8.1.5)中的DBMS_TRACE提供了三个过程，它们是SET_PLSQL_TRACE、CLEAR_PLSQL_TRACE和PLSQL_TRACE_VERSION。下面分别介绍这三个过程：

过程SET_PLSQL_TRACE 该过程启动当前会话中的跟踪，并开始把转储信息直接地送到

跟踪文件中。该过程用下面的语句定义：

```
PROCEDURE SET_PLSQL_TRACE (trace_level IN INTEGER);
```

其中trace_level说明要跟踪的对象。对事件跟踪级别的计算方法与我们在前面介绍的 event 10938 相同，也就是计算想要的跟踪功能值的和。下面列出了可以使用的跟踪功能（其中的值与 event 10938 的值相同）。

跟踪名称	值	说 明
TRACE_ALL_CALLS	1	跟踪所有的子程序调用。
TRACE_ENABLE_CALLS	2	只跟踪允许的调用（使用 DEBUG 编译的调用）。
TRACE_ALL_EXCEPTION	4	跟踪所有的异常。
TRACE_ENABLED_EXCEPTIONS	8	仅跟踪在允许的子程序中引发的异常。

该包的头文件中定义了跟踪名称的常数值，因此上表中的跟踪名称可以直接用在对 SET_PLSQL_TRACE 的调用中。例如，假设我们向数据库提交下面的语句：

```
节选自在线代码AllCallsExceptions815.sql
SQL> -- Enable tracing of all calls and all exceptions.
SQL> BEGIN
 2   DBMS_TRACE.SET_PLSQL_TRACE(
 3     DBMS_TRACE.TRACE_ALL_CALLS +
 4     DBMS_TRACE.TRACE_ALL_EXCEPTIONS);
 5 END;
 6 /
PL/SQL procedure successfully completed.

SQL> -- Anonymous block which raises some exceptions.
SQL> BEGIN
 2   CallRaise(1001);
 3   RaiseIt(-1);
 4 END;
 5 /
BEGIN
*
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at "EXAMPLE.RAISEIT", line 7
ORA-06512: at line 3
```

该块将在跟踪文件中生成下面的输出信息。这些信息与 event 10938 跟踪所生成的输出完全一致。

```
----- PL/SQL TRACE INFORMATION -----
Levels set : 1 4
Trace: ANONYMOUS BLOCK: Stack depth = 1
Trace: PROCEDURE EXAMPLE.CALLRAISE: Call to entry at line 3 Stack depth = 2
Trace: PROCEDURE EXAMPLE.RAISEIT: Call to entry at line 4 Stack depth = 3
Trace: Pre-defined exception - OER 1001 at line 9 of PROCEDURE EXAMPLE.RAISEIT:
```

```
Trace: PROCEDURE EXAMPLE.RAISEIT: Call to entry at line 4 Stack depth = 2
Trace: User defined exception at line 7 of PROCEDURE EXAMPLE.RAISEIT:
```

过程CLEAR_PLSQL_TRACE 该过程用来关闭会话的跟踪。该调用后执行的语句将不再被跟踪。该过程用下面的语句定义,不带参数:

```
PROCEDURE CLEAR_PLSQL_TRACE;
```

过程PLSQL_TRACE_VERSION 该过程返回包DBMS_TRACE的主版本和子版本。该包的头文件中也定义了主版本和子版本的常数。其格式如下:

```
PROCEDURE PLSQL_TRACE_VERSION(major OUT BINARY_INTEGER,
minor OUT BINARY_INTEGER);
```

其中*major*是主版本, *minor*是子版本。该过程在Oracle8i版本8.1.5下运行时返回下列输出:

节选自在线代码traceVersion.sql

```
SQL> DECLARE
 2   v_MajorVersion BINARY_INTEGER;
 3   v_MinorVersion BINARY_INTEGER;
 4 BEGIN
 5   DBMS_TRACE.PLSQL_TRACE_VERSION(v_MajorVersion, v_MinorVersion);
 6   DBMS_OUTPUT.PUT_LINE(
 7     'Trace major version: ' || v_MajorVersion);
 8   DBMS_OUTPUT.PUT_LINE(
 9     'Trace minor version: ' || v_MinorVersion);
10 END;
11 /
Trace major version: 1
Trace minor version: 0
PL/SQL procedure successfully completed.
```

2. Oracle8i版本8.1.6的DBMS_TRACE

与Oracle8i版本8.1.5的DBMS_TRACE和基于事件的跟踪不同, Oracle8i版本8.1.6的DBMS_TRACE生成的输出信息是存储在数据库表中的, 而不是转储在文件中。这种记录方式提供了更为可靠的存储方案。该包中提供的跟踪事件类型也要多一些, 并且还有额外的子程序可使用。

跟踪事件 Oracle8i版本8.1.6提供的可跟踪事件如表3-4所示。这些事件中包括了Oracle8i版本8.1.5中所有的事件。这些事件的使用与SET_PLSQL_TRACE中的使用方式完全一样。

表3-4 Oracle8i版本8.1.6的DBMS_TRACE提供的事件

跟踪事件	值	说 明
TRACE_ALL_CALLS	1	跟踪所有的子程序调用
TRACE_ENABLED_CALLS	2	仅跟踪允许的调用(使用DEBUG编译的调用)
TRACE_ALL_EXCEPTIONS	4	跟踪所有的异常
TRACE_ENABLED_EXCEPTIONS	8	仅跟踪在允许的子程序中引发的异常
TRACE_ALL_SQL	32	跟踪所有执行的SQL语句
TRACE_ENABLED_SQL	64	跟踪允许的子程序中执行的SQL语句
TRACE_ALL_LINES	128	跟踪所有代码行(包括调用和从过程返回的代码)
TRACE_ENABLED_LINES	256	跟踪允许子程序中的代码行

(续)

跟踪事件	值	说明
TRACE_STOP	16384	停止跟踪
TRACE_PAUSE	4096	暂停跟踪
TRACE_RESUME	8192	继续跟踪
TRACE_LIMIT	16	限制跟踪数量

暂停和继续跟踪 使用过程 PAUSE_PLSQL_TRACE可以暂停跟踪数据的收集，如果在其后调用过程RESUME_PLSQL_TRACE的话，可恢复跟踪功能。这两个命令都没有参数。

带有trace_level参数的SET_PLSQL_TRACE命令等价于TRACE_PAUSE命令，也可以暂停跟踪，类似地，也可以通过设置trace_level来恢复跟踪，等价于TRACE_RESUME。

限制跟踪数据 类似基于事件跟踪使用的环形缓冲区，基于PL/SQL的跟踪也可以保留最近的跟踪记录。实现这种功能的方法有两个。第一个是使用过程LIMIT_PLSQL_TRACE，该过程定义如下：

```
PROCEDURE LIMIT_PLSQL_TRACE(limit IN BINARY_INTEGER:=8192);
```

由参数limit指定的记录将被保留下（最近的记录），在此以前的记录将被覆盖。需要指出的是，系统保留的记录数是limit指定的记录数的近似值，这是因为系统并不对每个跟踪的发生进行核对。但是，超出指定记录数量的记录最多不会超出1000个。

第二个方法是在SET_PLSQL_TRACE中将TRACE_LIMIT作为trace_level使用，并设置event 10940(该命令用来限制基于事件跟踪使用的环行缓冲区的容量)。在这种情况下，跟踪限制将被设置为1023×event 10940设置的事件级别。

跟踪注释 每个跟踪操作都可以带有一个与其关联的注释，该功能可通过过程COMMENT_PLSQL_TRACE设置，其格式如下：

```
PROCEDURE COMMENT_PLSQL_TRACE(comment IN VARCHAR2);
```

其中，comment是指定的注释，其长度在2047个字符内。

版本核对 如果数据库版本已经升级或由于没有载入包DBMS_TRACE的相应版本而降级的话，用户程序就可能出现版本不兼容的问题。数据库版本可以由函数INTERNAL_VERSIAON_CHECK来核对。该函数的定义如下：

```
FUNCTION INTERNAL_VERSION_CHECK
RETURN BINARY_INTEGER;
```

如果版本匹配的话，该函数的返回值为0。如果不匹配，则该函数返回1。这时就要重新载入包DBMS_TRACE并重新运行程序。

操作号 每当开始一个跟踪操作时，系统就为该操作生成一个唯一的操作号。该操作号可由函数GET_PLSQL_TRACE_RUNNUMBER返回。其定义如下：

```
FUNCTION GET_PLSQL_TRACE_RUNNUMBER
RETURN BINARY_INTEGER;
```

跟踪表 保存跟踪数据的表有两个，它们是plsql_trace_runs和plsql_trace_events。这两个表是由脚本tracetab.sql生成的，在Unix操作系统中，存储在\$ORACLE_HOME/rdbms/admin中。这

两个表都隶属于SYS，因此对它们的访问必须经过授权。

有关每个跟踪操作的一般信息存储在表 plsql_trace_runs中，该表结构如下所示：

列	数据类型	说 明
runid	NUMBER	唯一的操作ID。
run_data	DATE	操作开始的时间。
run_owner	VARCHAR2(31)	已开始运行的数量。
run_comment	VARCHAR2(2047)	用户提供的注释。
run_commentl	VARCHAR2(2047)	附加的注释。注意，COMMENT_PLSQL_TRACE将只修改run_comment，因此，该列必须由手工修改。
run_end	DATE	操作结束的时间。
run_flags	VARCHAR2(2047)	操作使用的标志。
related_run	NUMBER	用于客户或服务器端相关操作。
run_system_info	VARCHAR2(2047)	没有启用。
spare1	VARCHAR2(256)	没有启用。

表plsql_trace_events记录了详细的跟踪数据，该表的每一行对应一个跟踪事件。其结构如下：

列	数据类型	说 明
runid	NUMBER	运行ID。
event_seq	NUMBER	事件ID。
event_time	DATE	该事件的时间。
related_event	NUMBER	相关事件的ID。
event_kind	VARCHAR2(31)	事件类型。
event_unit_dblink	VARCHAR2(31)	当前库单元的数据库连接。
event_unit_owner	VARCHAR2(31)	当前库单元的拥有者。
event_unit	VARCHAR2(31)	当前库单元的名称。
event_unit_kind	VARCHAR2(31)	当前库单元的类型。
event_line	NUMBER	当前行。
event_proc_name	VARCHAR2(31)	当前存在的过程名。
stack_depth	NUMBER	当前栈深度。
proc_name	VARCHAR2(31)	被调用过程的名称。
proc_dblink	VARCHAR2(31)	被调用过程的数据库连接。
proc_owner	VARCHAR2(31)	被调用过程的拥有者。
proc_unit	VARCHAR2(31)	调用的库单元。
proc_unit_kind	VARCHAR2(31)	调用过程的类型。
proc_line	NUMBER	调用过程的行。
proc_params	VARCHAR2 (2047)	过程参数。
icd_index	NUMBER	调用PL/SQL内部程序的ICD索引。
user_excp	NUMBER	用户定义的异常号。
excp	NUMBER	预定义异常号。
event_comment	VARCHAR2 (2047)	事件的注释。

跟踪事件的种类在列 event_kind给出，该列的值的含义在下面的表中说明。该表中的符号名称是定义在包DBMS_TRACE头文件中的常数。

符号名称	值	说 明
PLSQL_TRACE_START	38	开始跟踪。
PLSQL_TRACE_STOP	39	结束跟踪。
PLSQL_TRACE_SET_FLAGS	40	跟踪选项变更。
PLSQL_TRACE_PAUSE	41	跟踪暂停。
PLSQL_TRACE_RESUME	42	恢复跟踪。
PLSQL_TRACE_ENTER_VM	43	运行引擎入口。
PLSQL_TRACE_EXIT_VM	44	从运行引擎退出。
PLSQL_TRACE_BEGIN_CALL	45	调用独立过程。
PLSQL_TRACE_ELAB_SPEC	46	调用包说明。
PLSQL_TRACE_ELAB_BODY	47	调用包体。
PLSQL_TRACE_ICD	48	调用内部PL/SQL程序。
PLSQL_TRACE_PRC	49	调用远程过程。
PLSQL_TRACE_END_CALL	50	结束调用，返回调用块。
PLSQL_TRACE_NEW_LINE	51	PL/SQL代码新行。
PLSQL_TRACE_EXCP_RAISED	52	引发异常。
PLSQL_TRACE_EXCP_HANDLED	53	异常处理。
PLSQL_TRACE_SQL	54	运行SQL语句。
PLSQL_TRACE_BIND	55	处理连接变量。
PLSQL_TRACE_USER	56	用户定义的跟踪事件。
PLSQL_TRACE_NODEBUG	57	模块未用DEBUG编译，跳过该事件。

案例 有关DBMS_TRACE的案例，请访问专为本书设置的Web站点www.osborne.com。

3.4.3 基于PL/SQL的配置

Oracle 8i 及
更高版本

连同我们在上一节讨论的基于PL/SQL的跟踪一起，Oracle8i第二版(8.1.6)通过包DBMS_PROFILER提供了剖析程序(Profiler)。跟踪功能，如我们在上面所看到的，提供了PL/SQL程序运行期间所发生事件的有关信息，如对过程的调用，或所引发的异常等信息。而剖析程序则是用于记录程序运行时间的工具。借助于该工具，我们可以收集有关PL/SQL代码的每一行运行所用的最大，最小，和全部时间。这类时间信息还可以用于累计库单元一级或应用级的运行所用时间。

注意 我们在本章前面介绍的几种开发工具，如Rapid SQL、SQL Navigator以及SQL Programmer，都提供了剖析程序的图形界面。有关详细资料，请看在线文档。

1. DBMS_PROFILER子程序

DBMS_PROFILER中提供的子程序如表3-5所示。其有关功能在下面几节将详细说明。每个可用的子程序即可以作为函数，也可以作为过程使用。作为函数使用时，返回的BINAR_INTEGER的值用来说明成功或失败，而过程将在失败时引发异常。这些子程序的返回值在该包的头文件中定义。下面是部分返回码的说明。

返 回 码	值	说 明
SUCCESS	0	成功返回。
ERROR_PARAM	1	调用参数不对。

ERROR_IO	2	写入Profiler表有错。
ERROR_VERSION	-1	包版本与数据库版本不符。

表3-5 DBMS_PROFILER子程序

子 程 序	说 明
START_PROFILER	启动Profiler运行
STOP_PROFILER	停止Profiler运行并把数据写入表
PAUSE_PROFILER	暂停收集数据
RESUME_PROFILER	暂停后继续
FLUSH_DATA	把收集的数据写入表中
GET_VERSION	返回包Profiler的版本
INTERNAL_VERSION_CHECK	核对软件与数据库版本是否匹配
ROLLUP_UNIT	为指定的库单元累计数据
ROLLUP_RUN	为全部运行的程序累计数据

类似于Oracle8i的8.1.6版的DBMS_TRACE，剖析程序数据被写入数据库表中存放。我们将在下面“DBMS_PROFILER表”中介绍这些表的结构。

START_PROFILER 该程序运行后开始收集Profiling数据并返回当前运行号。该程序的定义如下：

```

FUNCTION START_PROFILER( run_comment IN VARCHAR2 := SYSDATE,
                           run_comment1 IN VARCHAR2 := "",
                           run_number OUT BINARY_INTEGER)

RETURN BINARY_INTEGER;

PROCEDURE START_PROFILER( run_comment IN VARCHAR2 := SYSDATE,
                           run_comment1 IN VARCHAR2 := "",
                           run_number OUT BINARY_INTEGER);

FUNCTION START_PROFILER( run_comment IN VARCHAR2 := SYSDATE,
                           run_comment1 IN VARCHAR2 := "")

RETURN BINARY_INTEGER;

PROCEDURE START_PROFILER( run_comment IN VARCHAR2 := SYSDATE,
                           run_comment1 IN VARCHAR2 := "");
```

其中的参数run_comment和run_comment1可以用 来说明该剖析程序的运行并被存储在该Profiler表中。当前运行号将在run_number中返回。需要注意的是，如果你使用了不返回运行号的版本，则确认运行号的唯一方法是去查询该表。

STOP_PROFILER 该程序将停止收集Profiler数据，并把已收集的数据写入表中。该程序定义如下：

```

FUNCTION STOP_PROFILER RETURN BINARY_INTEGER;
PROCEDURE STOP_PROFILER
```

该程序没有参数。

PAUSE_PROFILER 该程序将临时停止收集Profiler数据。这时，它不清除缓冲区。定义如下：

```
FUNCTION PAUSE_PROFILER RETURN BINARY_INTEGER;
```

```
PROCEDURE PAUSE_PROFILER;
```

RESUME_PROFILER 该程序将在暂停后重新开始收集 Profiler数据。定义如下：

```
FUNCTION RESUME_PROFILER RETURN BINARY_INTEGER;
```

```
PROCEDURE RESUME_PROFILER;
```

FLUSH_DATA 该程序将把收集到的数据写入 Profiler表中。除非使用 FLUSH_DATA或 STOP_PROFILER存储数据，否则数据不予保存。该程序定义如下：

```
FUNCTION FLUSH_DATA RETURN BINARY_INTEGER;
```

```
PROCEDURE FLUSH_DATA;
```

GET_VERSION 该过程将返回包 Profiler的主版本和子版本。主版本和子版本的定义在该包的头文件中。该过程定义如下：

```
PROCEDURE GET_VERSION(major OUT BINARY_INTEGER,
minor OUT BINARY_INTEGER);
```

主版本将在参数major中返回，子版本在minor中返回。

INTERNAL_VERSION_CHECK 类似于包 DBMS_TRACE,如果数据库版本已经升级，或因没有载入相应版本的包 DBMS_PROFILER而降级的话，程序运行时将会引发错误。版本匹配工作可由该函数实现，其定义如下：

```
FUNCTION INTERNAL_VERSION_CHECK
RETURN BINARY_INTEGER;
```

如果版本匹配的话，则其返回值为 0,否则为1。如果该函数返回 1,用户应再次载入 DBMS_PROFILER并重新运行程序。

ROLLUP_UNIT 该过程将计算运行某个程序单元的总运行时间。该过程的定义如下：

```
PROCEDURE ROLLUP_UNIT(run IN NUMBER,unit IN NUMBER);
```

其中，参数run是运行号，unit是单元号。每个程序单元都被赋予一个唯一的运行号，该运行号也可以通过查询 Profiler表获得。

ROLLUP_RUN 该过程将计算给定运行的总执行时间。其定义如下：

```
PROCEDURE ROLLUP_RUN(run IN NUMBER);
```

其中，参数run是运行号。

2. DBMS_PROFILER 表

Profiler数据存储在三个数据库表中，这些表可以用文件 proftab.sql创建。在 Unix系统下，该文件存储在\$ORACLE_HOME/rdbms/admin中。下面将描述这些表。

表PLSQL_PROFILER_RUNS 该表存储每个Profiler操作的信息。其结构如下：

列	数 据 类 型	说 明
runid	NUMBER	运行的唯一ID。
related_run	NUMBER	相关运行的ID。该ID用于客户和服务器的相互关联的运行。
run_owner	VARCHAR2(32)	启动该运行的用户。
run_date	DATE	运行的开始时间。
run_comment	VARCHAR2(2047)	用户为该运行指定注释，该注释传递到 START_PROFILER中。
run_total_time	NUMBER	该运行的总执行时间。

run_system_info	VARCHAR2(2047)	未启用。
run_comment1	VARCHAR2(2047)	附加的注释，该注释也传递到 START_PROFILER中。
spare1	VARCHAR2(256)	未启用。

表PLSQL_PROFILER_UNITS 该表存储了运行期间每个程序单元的有关信息,该表的结构如下：

列	数据类型	说 明
runid	NUMBER	运行的唯一ID。
unit_number	NUMBER	程序单元的唯一ID。
unit_type	VARCHAR2(32)	程序单元的类型。
unit_owner	VARCHAR2(32)	程序单元的所有者。
unit_name	VARCHAR2(32)	程序单元的名称。
unit_timestamp	DATE	程序单元的时间戳，可用来检测变更。
total_time	NUMBER	程序单元的运行所用的总时间。
spare1	NUMBER	未启用。
spare2	NUMBER	未启用。

表PLSQL_PROFILER_DATA 该表提供了最低级别的Profiling信息,也就是说,只提供每行PL/SQL代码的有关信息。该表的结构如下：

列	数据类型	说 明
runid	NUMBER	唯一的运行ID。
unit_number	NUMBER	唯一的单元ID。
line#	NUMBER	单元中的行号。
total_occur	NUMBER	运行中该行运行的次数。
total_time	NUMBER	该行所用的总运行时间。
min_time	NUMBER	该行运行所需的最长时间。
max_time	NUMBER	该行运行所需的最大时间。
spare1	NUMBER	未用。
spare2	NUMBER	未用。
spare3	NUMBER	未用。
spare4	NUMBER	未用。

案例 有关DBMS_PROFILER的案例 , 请访问为本书专门提供的 Web站点 www.osborne.com.

3.5 小结

我们在本章分析了调试 PL/SQL代码的不同技术 , 其涉及范围从基于字符的调试技术如 DBMS_OUTPUT到插入调试表的全图形 GUI调试器。使用那种调试方法取决于程序所在的环境和要求。我们在本章除了逐个介绍了每种调试方法外 , 还讨论了七个常见的 PL/SQL错误以及避免这些错误的方法。我们在本章的最后几节讨论了 PL/SQL不同版本提供的跟踪和配置工具。

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，procedure_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) ]

```

下载

```
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

下载

```

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。

下载

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ParameterLength.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ParameterLength.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_Variable2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ParameterLength:

节选自在线代码ParameterLength.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ParameterLength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ParameterLength.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.
```

下载

```
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。Oracle 8i 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** Oracle 8i 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，OUT，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

下载

```
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(PL/SQL)。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PL/SQL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_UTLILITY.GET_TIME will return the current time, in
    -- hundredths of a second.
    v_Time1 := DBMS_UTLILITY.GET_TIME;
    PassStudents1(v_StudentArray);
    v_Time2 := DBMS_UTLILITY.GET_TIME;
    PassStudents2(v_StudentArray);
    v_Time3 := DBMS_UTLILITY.GET_TIME;
    PassStudents3(v_StudentArray);
    v_Time4 := DBMS_UTLILITY.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 Oracle 8i 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 DefaultTest.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及其参数（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,
```

下载

```
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PackageA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;
```

下载

```

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;
```

下载

```
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
```

下载

```
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, 'SSSS')));
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。

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;
```

The screenshot shows the Oracle SQL*Plus interface with the following session history:

```
SQL> SELECT object_name, object_type, status
  2   FROM user_objects WHERE object_name = 'SIMPLE';
OBJECT_NAME          OBJECT_TYPE      STATUS
SIMPLE                PROCEDURE      VALID

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;
no rows selected

SQL>
```

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的错误信息。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FormatName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 StoredProcedure;

```

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
 11  CURSOR c_AllStudents IS
 12    SELECT first_name, last_name
 13    FROM students;
 14
 14  v_FormattedName VARCHAR2(50);
 15 -- Begin main block
 16 BEGIN
 17   NULL;
 18 END;
 19 /
CURSOR c_AllStudents IS
*
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

下载

```
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：

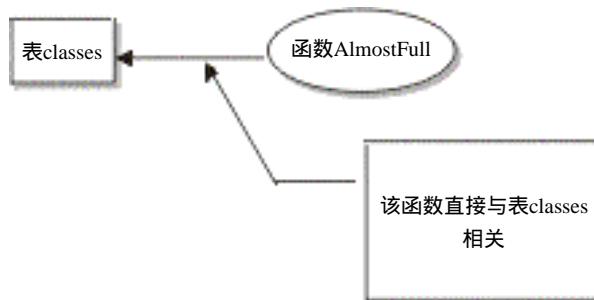


图5-3 函数AlmostFull的相关图示

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calsses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。

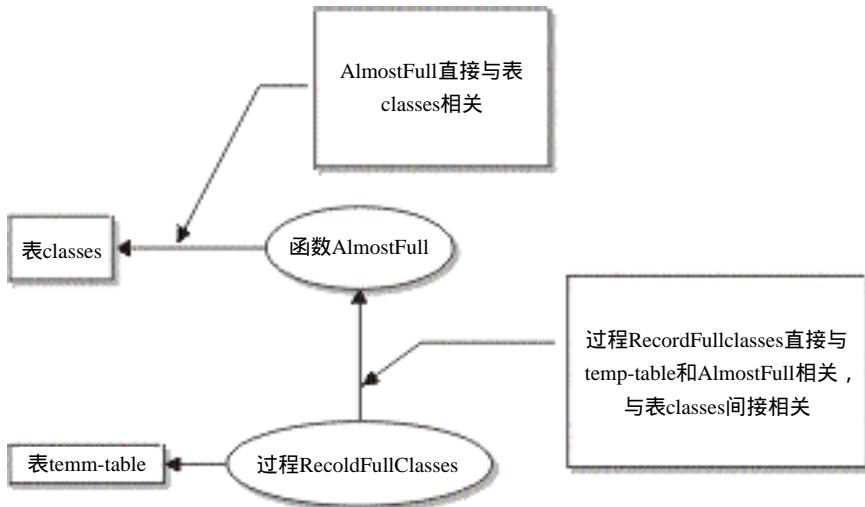


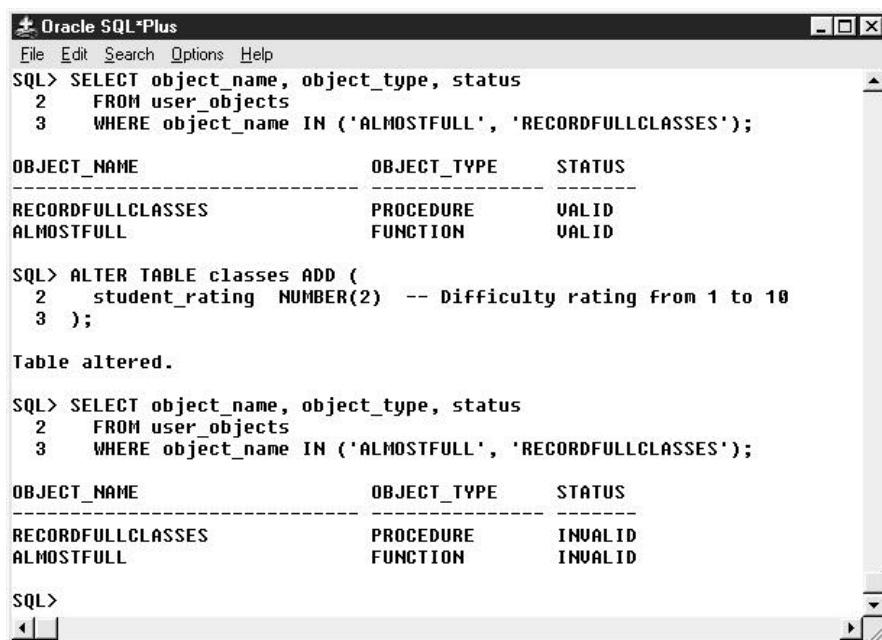
图 5-4 RecordFullClasses 的相关图示

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，而不是在编译后给出。

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进行重编。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



The screenshot shows an Oracle SQL*Plus session. The user runs a query to select objects from the user_objects view, filtering by object_name ('ALMOSTFULL' and 'RECORDFULLCLASSES'). The results show two objects: RECORDFULLCLASSES (PROCEDURE) and ALMOSTFULL (FUNCTION), both marked as VALID.

```

SQL> SELECT object_name, object_type, status
  2  FROM user_objects
  3  WHERE object_name IN ('ALMOSTFULL', 'RECORDFULLCLASSES');

OBJECT_NAME          OBJECT_TYPE      STATUS
-----              -----          -----
RECORDFULLCLASSES    PROCEDURE        VALID
ALMOSTFULL           FUNCTION         VALID

SQL> ALTER TABLE classes ADD (
  2   student_rating NUMBER(2) -- Difficulty rating from 1 to 10
  3 );

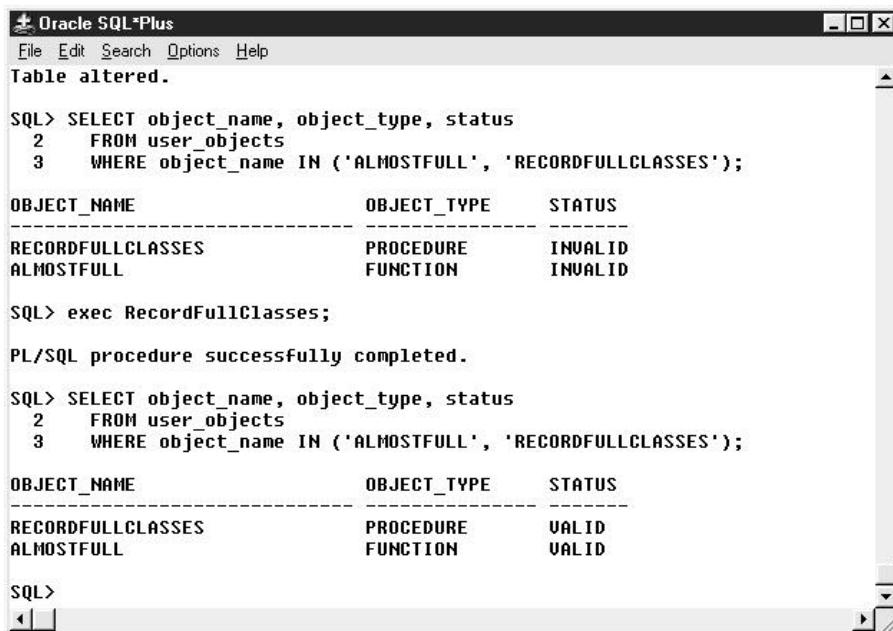
Table altered.

SQL> SELECT object_name, object_type, status
  2  FROM user_objects
  3  WHERE object_name IN ('ALMOSTFULL', 'RECORDFULLCLASSES');

OBJECT_NAME          OBJECT_TYPE      STATUS
-----              -----          -----
RECORDFULLCLASSES    PROCEDURE        INVALID
ALMOSTFULL           FUNCTION         INVALID

```

图5-5 DDL操作导致的非法信息



The screenshot shows an Oracle SQL*Plus session. The user alters a table and then runs a query to select objects from the user_objects view, filtering by object_name ('ALMOSTFULL' and 'RECORDFULLCLASSES'). The results show two objects: RECORDFULLCLASSES (PROCEDURE) and ALMOSTFULL (FUNCTION), both marked as INVALID. The user then executes a procedure named RecordFullClasses, which is successfully completed. Finally, the user runs the same query again, and the results show the objects are now marked as VALID.

```

Table altered.

SQL> SELECT object_name, object_type, status
  2  FROM user_objects
  3  WHERE object_name IN ('ALMOSTFULL', 'RECORDFULLCLASSES');

OBJECT_NAME          OBJECT_TYPE      STATUS
-----              -----          -----
RECORDFULLCLASSES    PROCEDURE        INVALID
ALMOSTFULL           FUNCTION         INVALID

SQL> exec RecordFullClasses;

PL/SQL procedure successfully completed.

SQL> SELECT object_name, object_type, status
  2  FROM user_objects
  3  WHERE object_name IN ('ALMOSTFULL', 'RECORDFULLCLASSES');

OBJECT_NAME          OBJECT_TYPE      STATUS
-----              -----          -----
RECORDFULLCLASSES    PROCEDURE        VALID
ALMOSTFULL           FUNCTION         VALID

```

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将会导致包体非法。同样，如果在包头中增加了一个对象

下载

(如游标或变量), 则包体将被作废。

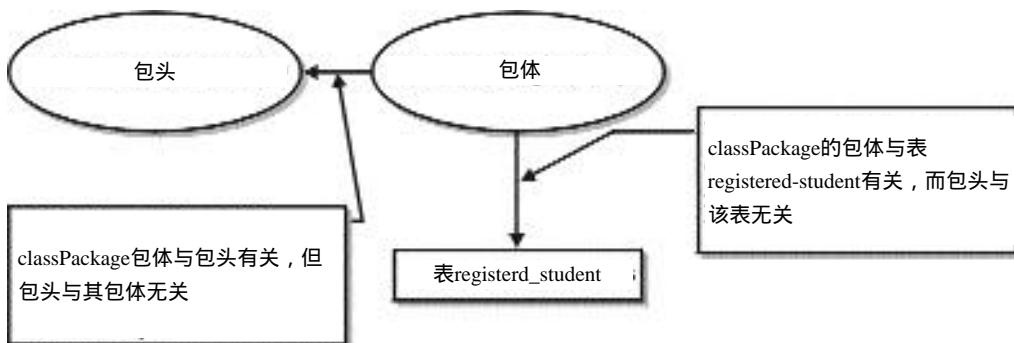


图5-7 包ClassesPackage的相关图

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关系，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：

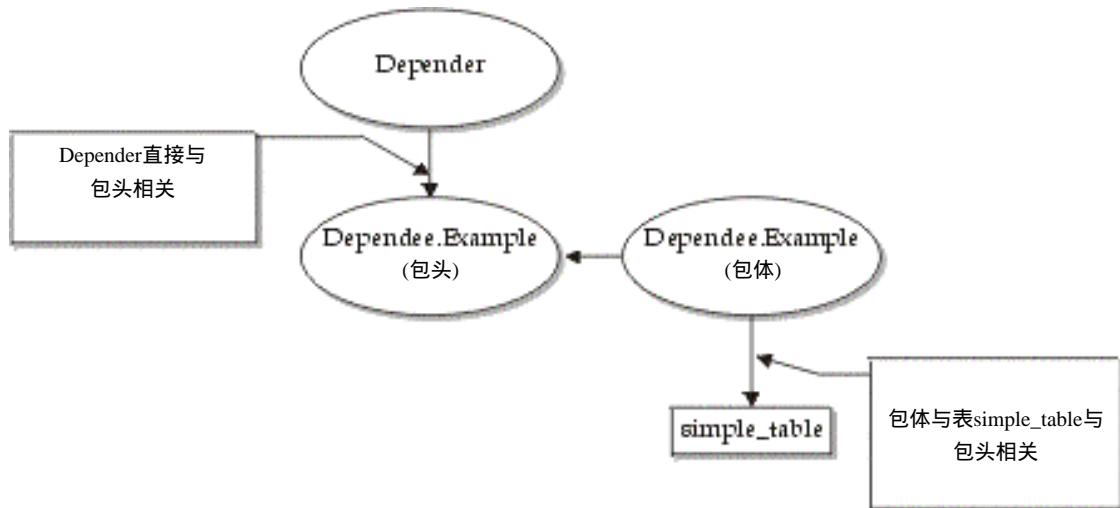


图5-8 多个包的相关图示

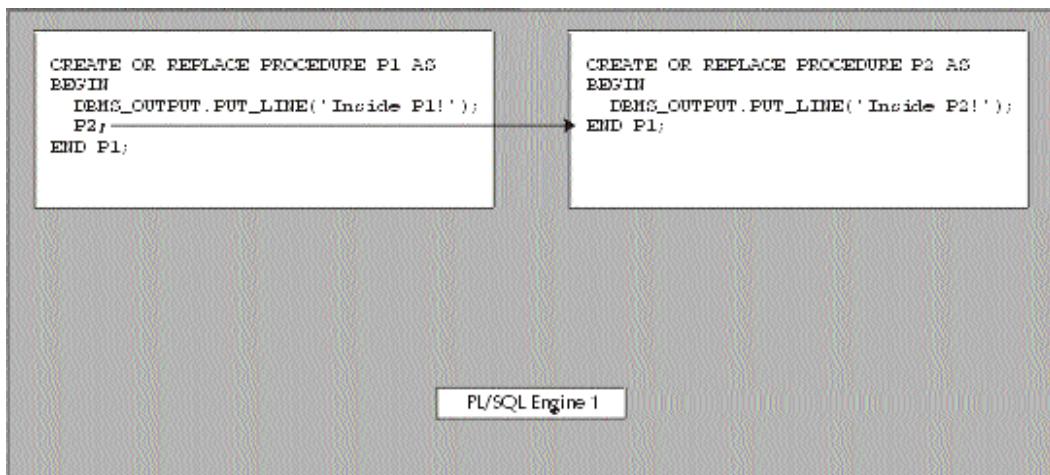


图5-9 在同一数据库中的过程P1和P2

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';
```

下载

```
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。

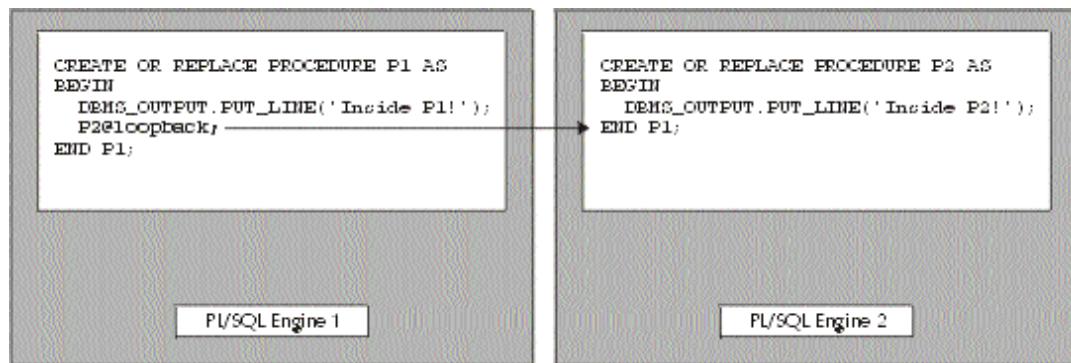


图5-10 处于不同数据库中的过程 P1 和 P2

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/SQL2.2 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。
- 要用手工对过程进行重编，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 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：

下载

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。

The screenshot shows the Oracle SQL*Plus interface with the title bar "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays three separate executions of a PL/SQL procedure named @ch05\callRS. Each execution prints the message "Fetched [number] rows:" followed by a list of student names from the "students" table. The results are as follows:

```
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:
Manish Murgatroid
Patrick Poll
Rita Razmataz
Rose Riznit
Shay Shariatpanahy

PL/SQL procedure successfully completed.

SQL> @ch05\callRS
Fetched 2 rows:
Scott Smith
Timothy Taller

PL/SQL procedure successfully completed.
```

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_REUSEABLE;
```

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_REUSEABLE;

  -- 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。

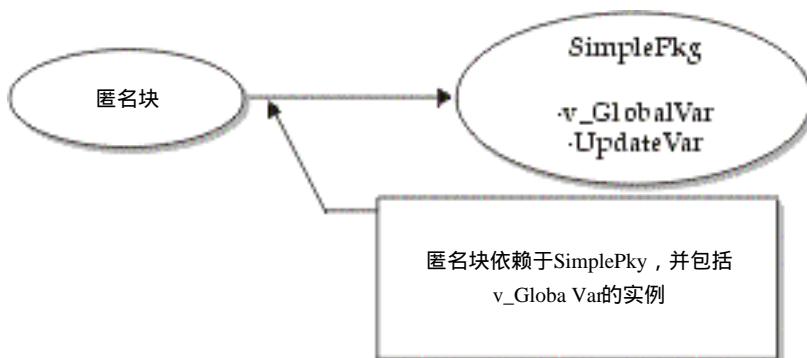


图5-13 包的全局相关图示

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 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 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。如果我们把 RecordFullClasses 的特权赋予其他的数据库用户，如 UserB，如下面的命令所示：

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.

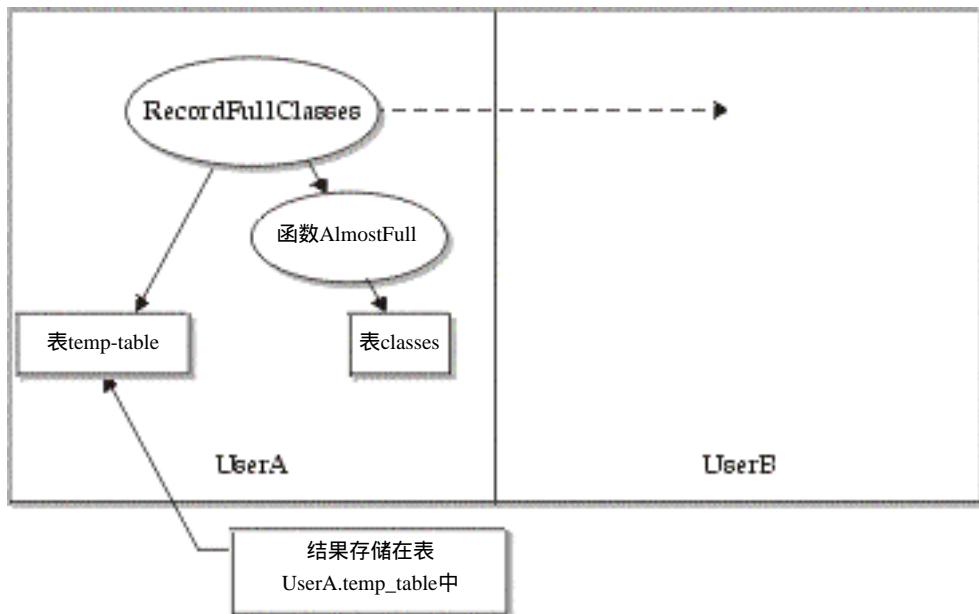


图5-14 UserA所属的数据库对象

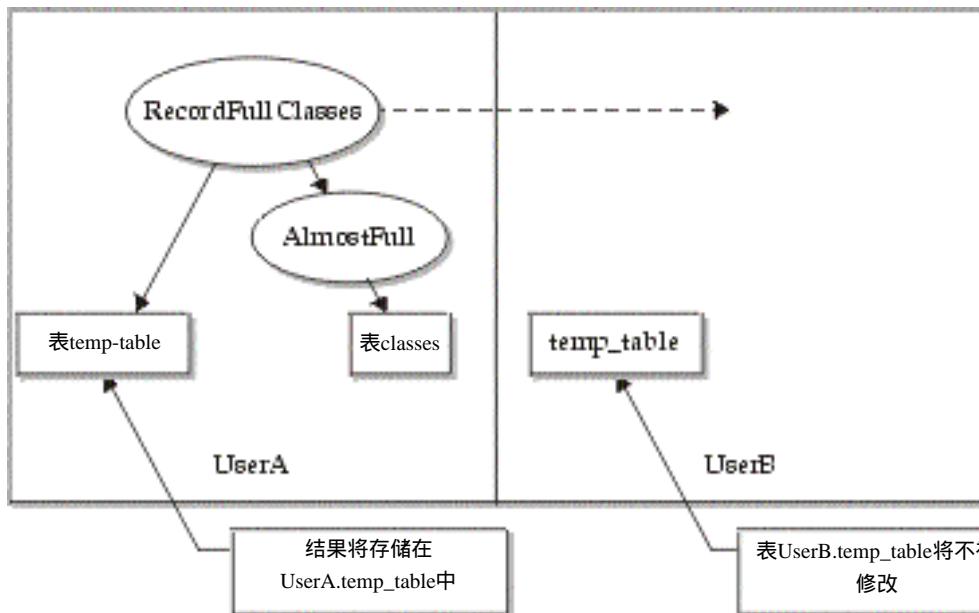


图5-15 UserA和UserB所属的表temp_table

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。我们再进一步假设已经对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特权和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. RecordFullClasses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;
```

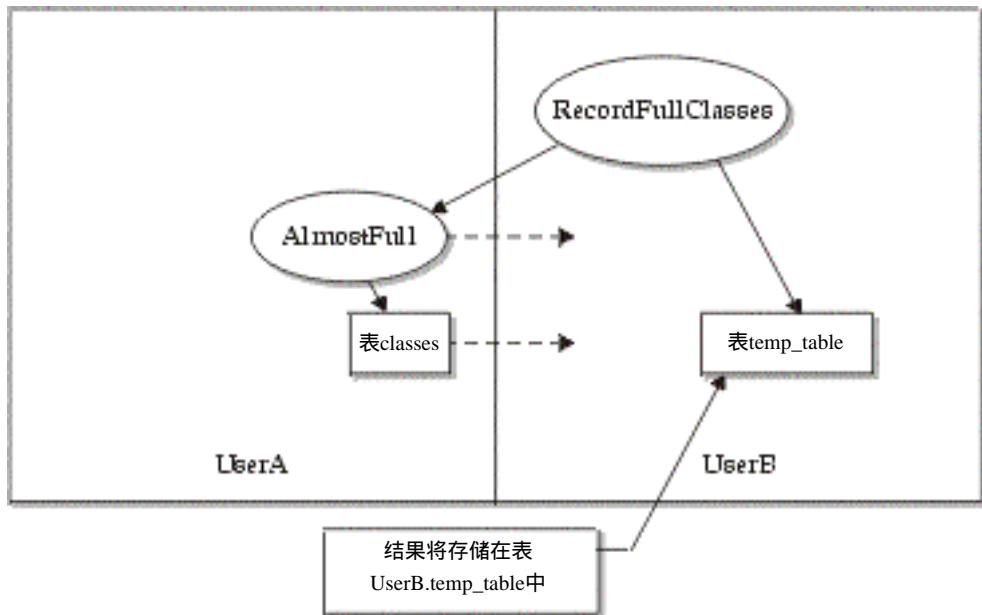


图5-16 UserB所属的RecordFullClasses

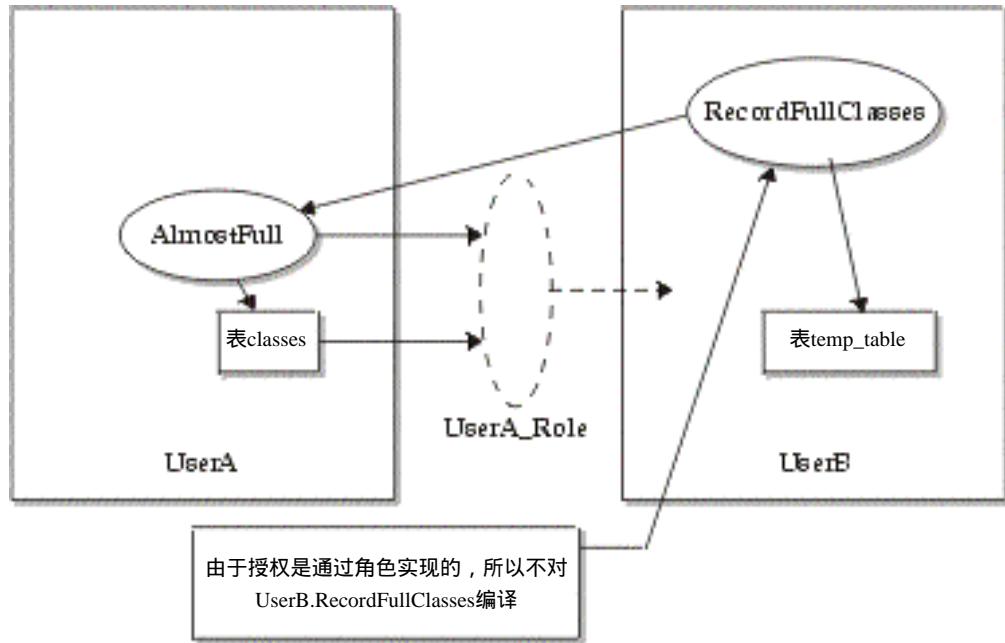


图5-17 通过角色实现授权

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（Oracle7.3及更高版本）都是子程序拥有者所属的对象，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.tmepl_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。

Oracle8i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  /
```

下载

```

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!

```

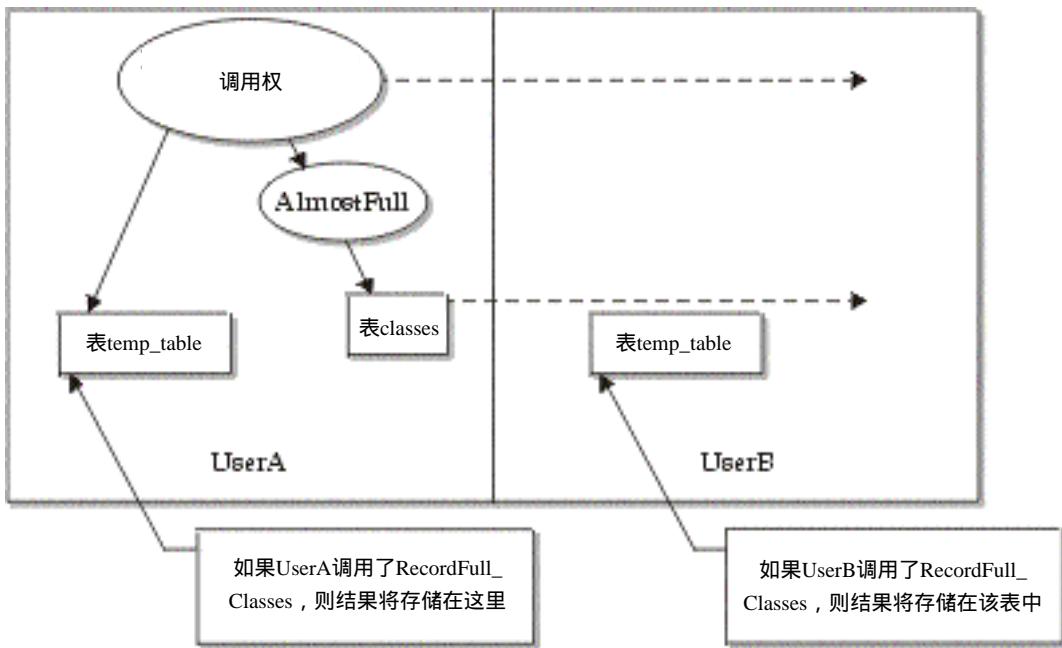


图5-18 具有调用权的过程 RecordFullClasses

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UserA特权组下实现，因此，不必对UserB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
```

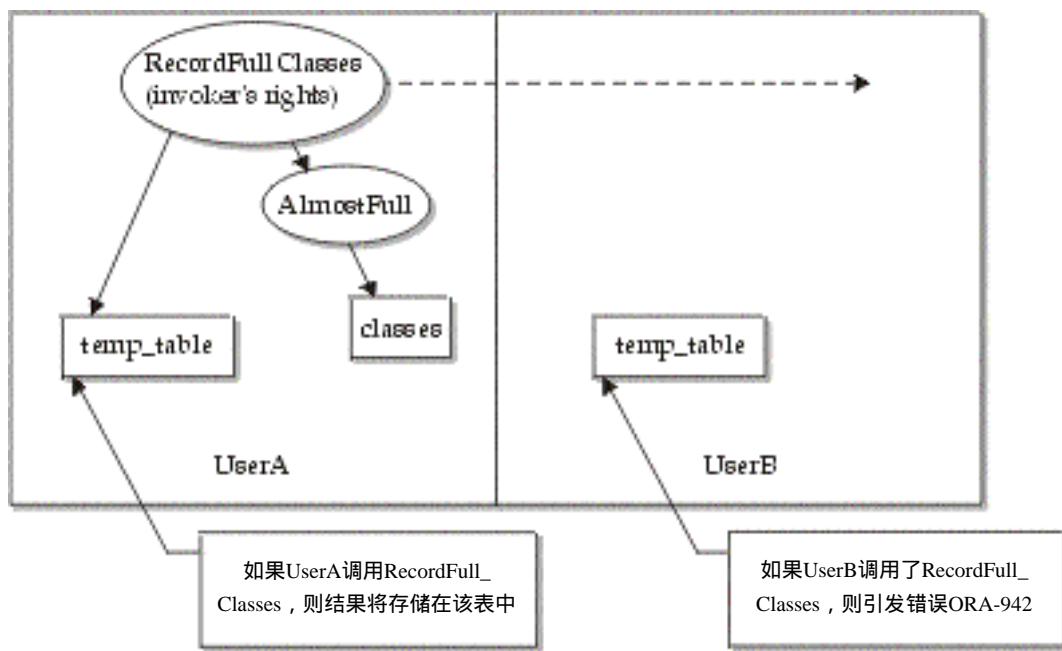


图5-19 撤消对表classes的SELECT操作

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，

下载

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：

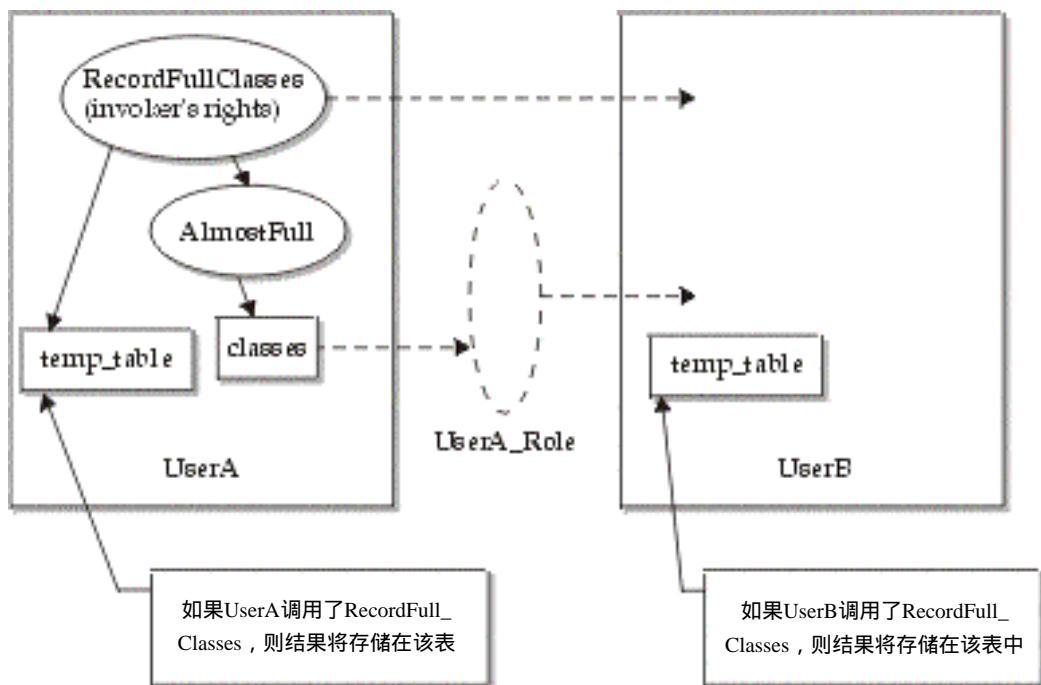


图 5-20 角色和调用权图示

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（Positional 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

Oracle 8i 及更高版本 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To Upper。

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中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_SHRAED_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。其语录法如下：

```
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 SERVEROUTPUT 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

Oracle 8i 及
更高版本

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 / S Q L 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。

第6章 数据库触发器

命名PL/SQL块的第四种类型是触发器。触发器类在某些方面类似于子程序，但它们之间也有明显地区别。我们在本章将介绍如何创建不同类型的触发器以及讨论触发器的一些应用。

6.1 触发器的类型

触发器类似于函数和过程，它们都是具有声明部分、执行部分和异常处理部分的命名PL/SQL块。像包一样，触发器必须在数据库中以独立对象的身份存储，并且不能与包和块具有本地关系。我们在前两章中已经讲过，过程是显式地通过过程调用从其他块中执行的，同时，过程调用可以传递参数。与之相反，触发器是在事件发生时隐式地运行的，并且触发器不能接收参数。运行触发器的方式叫做激发（firing）触发器，触发事件可以是对数据库表的DML（INSERT、UPDATE或DELETE）操作或某种视图的操作（View）。Oracle8i把触发器功能扩展到了可以激发系统事件，如数据库的启动和关闭，以及某种DDL操作。我们将在本章的后几节讨论触发事件的详细内容。

触发器可以用于下列情况：

- 维护在表创建阶段通过声明限制无法实现的复杂完整性限制。
- 通过记录修改内容和修改者来审计表中的信息。
- 在表内容发生变更时，自动通知其他程序采取相应的处理。
- 在订阅发布环境下，发布有关各种事件的信息。

有三种主要的触发器类型：DML、替代触发器和系统触发器。在下面几节中，我们将逐一介绍这些触发器类型。在本章后面“创建触发器”一节中还将详细讨论这些触发器。

注意 Oracle8i允许使用PL/SQL语言或可以作为外部例程调用的其他语言来编制触发器。有关触发器的详细介绍，请参阅6.2.4节和第10章的内容。

6.1.1 DML触发器

DML触发器可以由DML语句激发，并且由该语句的类型决定DML触发器的类型。可以定义DML触发器进行INSERT、UPDATE、DELETE操作。这类触发器可以在上述操作之前或之后激发，除此之外，它们也可以在行或语句操作上激发。

作为例子，让我们假设要跟踪不同专业的统计信息，其中包括已注册学生的数量和已得到的总分。我们要把这些结果存储在表major_stats中：

节选自在线代码relTables.sql

```
CREATE TABLE major_stats (
    major          VARCHAR2(30),
    total_credits NUMBER,
```

```
total_students NUMBER);
```

为了保持表major_stats中的数据处于更新状态，我们可以创建一个每次表students被修改时自动更新表major_stats的触发器。下面所示的UpdateMajorStats就是实现上述功能的触发器。在表students上进行任何DML操作之后，该触发器将启动运行。该触发器的代码要查询表students并使用当前的统计信息更新表major_stats。

节选自在线代码UpdateMajorStats.sql

```
CREATE OR REPLACE TRIGGER UpdateMajorStats
/* Keeps the major_stats table up-to-date with changes made
   to the students table. */
AFTER INSERT OR DELETE OR UPDATE ON students
DECLARE
  CURSOR c_Statistics IS
    SELECT major, COUNT(*) total_students,
           SUM(current_credits) total_credits
      FROM students
     GROUP BY major;
BEGIN
  /* First delete from major_stats. This will clear the
     statistics, and is necessary to account for the deletion
     of all students in a given major. */
  DELETE FROM major_stats;
  /* Now loop through each major, and insert the appropriate row into
     major_stats. */
  FOR v_StatsRecord in c_Statistics LOOP
    INSERT INTO major_stats (major, total_credits, total_students)
      VALUES (v_StatsRecord.major, v_StatsRecord.total_credits,
              v_StatsRecord.total_students);
  END LOOP;
END UpdateMajorStats;
```

语句触发器可以激发多种类型的触发语句。例如，UpdateMajorStats可以激发INSERT，UPDATE，DELETE语句。触发事件说明了一个或多个激发触发器的DML操作。

6.1.2 替代触发器

**Oracle 8 及
更高版本** Oracle8提供的这种替代触发器(Instead-of trigger)只能定义在视图上(可以是关系或对象)。与DML触发器不同，DML触发器是在DML操作之外运行的，而替代触发器则代替激发它的DML语句运行。替代触发器是行一级的。例如，请看下面的视图classes_rooms:

节选自在线代码insteadOf.sql

```
CREATE OR REPLACE VIEW classes_rooms AS
  SELECT department, course, building, room_number
    FROM rooms, classes
   WHERE rooms.room_id = classes.room_id;
```

下载

如下所示，直接执行对该视图的插入操作是非法的。这是因为该视图是两个表的联合，而插入操作要求对两个现行表进行修改，下面的SQL *Plus会话显示了插入操作过程：

节选自在线代码insteadOf.sql

```
SQL> INSERT INTO classes_rooms (department, course, building,
   room_number)
2      VALUES ('MUS', 100, 'Music Building', 200);
INSERT INTO classes_rooms (department, course, building, room_number)
*
ERROR at line 1:
ORA-01732: data manipulation operation not legal on this view
```

然而，我们可以创建一个替代触发器来实现正确的插入操作，也就是来更新现行表：

节选自在线代码insteadOf.sql

```
CREATE TRIGGER ClassesRoomsInsert
INSTEAD OF INSERT ON classes_rooms
DECLARE
  v_roomID rooms.room_id%TYPE;
BEGIN
  -- First determine the room ID
  SELECT room_id
    INTO v_roomID
   FROM rooms
  WHERE building = :new.building
    AND room_number = :new.room_number;
  -- And now update the class
  UPDATE CLASSES
    SET room_id = v_roomID
   WHERE department = :new.department
     AND course = :new.course;
END ClassesRoomsInsert;
```

有了触发器ClassesRoomsInsert，INSERT语句就可以执行正确的更新操作。

注意 在上面的程序中，触发器ClassesRoomsInsert没有做任何错误检查。我们将在本章后的程序中加入错误检查和处理代码。

6.1.3 系统触发器

Oracle 8i 及更高版本 Oracle8i提供了第三种触发器，这种系统触发器在发生如数据库启动或关闭等系统事件时激发，而不是在执行DML语句时激发。系统触发器也可以在DDL操作时，如表的创建中激发。例如，假设我们要记录对象创建的时间，我们可以通过创建下面的表来实现上述记录功能：

节选自在线代码LogCreations.sql

```
CREATE TABLE    ddl_creations (
  user_id          VARCHAR2(30),
  object_type      VARCHAR2(20),
```

```
object_name      VARCHAR2(30),
object_owner    VARCHAR2(30),
creation_date   DATE);
```

一旦该表可以使用，我们就可以创建一个系统触发器来记录相关信息。在每次 CREATE语句对当前模式进行操作之后，触发器 LogCreations就记录在 ddl_creations中创建的对象的有关信息。

```
节选自在线代码LogCreations.sql
CREATE OR REPLACE TRIGGER LogCreations
  AFTER CREATE ON SCHEMA
BEGIN
  INSERT INTO ddl_creations (user_id, object_type, object_name,
                             object_owner, creation_date)
  VALUES (USER, SYS.DICTIONARY_OBJ_TYPE, SYS.DICTIONARY_OBJ_NAME,
          SYS.DICTIONARY_OBJ_OWNER, SYSDATE);
END LogCreations;
```

6.2 创建触发器

所有触发器，不管其类型如何，都可以使用相同的语法创建。下面是创建触发器的通用语法：

```
CREATE [OR REPLACE] TRIGGER trigger_name
  {BEFORE | AFTER | INSTEAD OF} triggering_event
  referencing_clause
  [WHEN trigger_condition]
  [FOR EACH ROW]
  trigger_body;
```

其中，trigger_name是触发器的名称，triggering_event说明了激发触发器的事件（也可能包括特殊的表或视图），trigger_body是触发器的代码。referencing_clause用来引用正在处于修改状态下的行中的数据，如果在 WHEN子句中指定 trigger_condition的话，则首先对该条件求值。触发器主体只有在该条件为真值时才运行。我们在下面几节中将演示不同类型的触发器案例。

注意 触发器主体不能超过32K。如果触发器长度超过了该限制，就要把该体内的某些代码放到单独编译的包或存储子程序中，并从触发器主体中调用这些代码。

6.2.1 创建DML触发器

DML触发器是由对数据库表进行 INSERT、UPDATE、DELETE操作而激发的触发器。该类触发器可以在上述操作之前或之后激发运行，也可以按每个变更行激发一次，或每个语句激发一次进行。这些条件的组合形成了触发器的类型。总共有 12种可能的触发类型：3种语句 × 2种定时 × 2级。例如，下面所有的说明都是合法的 DML触发器类型：

- 更新语句之前
- 插入行之后
- 删除行之前

表6-1总结了DML触发器的各种选择项。除此之外，触发器也可以由给定表中的一个以上的DML语句，如INSERT和UPDAE而激发。触发器中的任何代码将随触发语句一起作为同一事务的一部分运行。

可以对一个表定义任意数量的触发器，其中可以包括一个以上的给定 DML类型。例如，可以定义两个删除语句之后的触发器。所有同类型的触发器将按顺序激发。（下一节讨论触发器的激发顺序。）

注意 在PL/SQL2.1版（Oracle7的7.1版）之前的版本下，每种类型的触发器只能在表上定义一个。也就是说，最多有 12个触发器。因此，初始化参数 COMPATIBLE就必须是7.1或更高以便复制一个表上的同类触发器。

DML触发器的触发事件说明了激发触发器的表的名称（以及列）。在Oracle8i下，触发器可以在嵌套表的列上激发。本书的第14章提供了更多的触发器内容。

表6-1 DML触发器类型

类 别	值	说 明
语句	INSERT、DELETE、 UPDATE	定义何种DML语句激发触发器
定时 级	之前或之后 行或语句	定义触发器是在语句运行前或运行后激发 如果触发器是行级触发器，该触发器就对由触发语句变更的每一行激发一次。如果触发器是语句级的触发器，则该触发器就在语句之前或之后激发一次。行级触发器是按触发器定义中的FOR EACH ROW子句表示的

1. DML 触发器激发顺序

触发器是在DML语句运行时激发的。下面是执行 DML语句的算法步骤：

- 1) 如果有语句之前级触发器的话，先运行该触发器。
- 2) 对于受语句影响每一行：
 - a. 如果有行之前级触发器的话，运行该触发器。
 - b. 执行该语句本身。
 - c. 如果有行之后级触发器的话，运行该触发器。
- 3) 如果有语句之后级触发器的话，运行该触发器。

为了说明上面的算法，假设我们在表 classes上创建了所有四种UPDATE触发器，即之前，之后，语句和行级。我们将创建三个行前触发器和两个语句后触发器，其代码如下：

```
节选自在线代码firingOrder .sql
CREATE SEQUENCE trig_seq
START WITH 1
INCREMENT BY 1;

CREATE OR REPLACE PACKAGE TrigPackage AS
-- Global counter for use in the triggers
v_Counter NUMBER;
END TrigPackage;
```

```
CREATE OR REPLACE TRIGGER ClassesBStatement
  BEFORE UPDATE ON classes
BEGIN
  -- Reset the counter first.
  TrigPackage.v_Counter := 0;
  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
            'Before Statement: counter = ' || TrigPackage.v_Counter);

  -- And now increment it for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesBStatement;

CREATE OR REPLACE TRIGGER ClassesAStatement1
  AFTER UPDATE ON classes
BEGIN
  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
            'After Statement 1: counter = ' || TrigPackage.v_Counter);

  -- Increment for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesAStatement1;

CREATE OR REPLACE TRIGGER ClassesAStatement2
  AFTER UPDATE ON classes
BEGIN
  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
            'After Statement 2: counter = ' || TrigPackage.v_Counter);

  -- Increment for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesAStatement2;

CREATE OR REPLACE TRIGGER ClassesBRow1
  BEFORE UPDATE ON classes
  FOR EACH ROW
BEGIN
  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
            'Before Row 1: counter = ' || TrigPackage.v_Counter);

  -- Increment for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesBRow1;

CREATE OR REPLACE TRIGGER ClassesBRow2
```

```
BEFORE UPDATE ON classes
FOR EACH ROW
BEGIN
  INSERT INTO temp_table (num_col, char_col)
  VALUES (trig_seq.NEXTVAL,
    'Before Row 2: counter = ' || TrigPackage.v_Counter);
  -- Increment for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesBRow2;

CREATE OR REPLACE TRIGGER ClassesBRow3
  BEFORE UPDATE ON classes
  FOR EACH ROW
BEGIN
  INSERT INTO temp_table (num_col, char_col)
  VALUES (trig_seq.NEXTVAL,
    'Before Row 3: counter = ' || TrigPackage.v_Counter);
  -- Increment for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesBRow3;

CREATE OR REPLACE TRIGGER ClassesARow
  AFTER UPDATE ON classes
  FOR EACH ROW
BEGIN
  INSERT INTO temp_table (num_col, char_col)
  VALUES (trig_seq.NEXTVAL,
    'After Row: counter = ' || TrigPackage.v_Counter);

  -- Increment for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesARow;
```

假设我们现在提交下列UPDATE语句：

```
UPDATE classes
  SET num_credits = 4
 WHERE department IN ('HIS', 'CS');
```

该语句对四行有影响。语句之前级和之后触发器将各自运行一次，而行之前级和之后触发器则每个运行四次。如果在上述操作之后，我们从表 temp_table进行选择的话，下面就是得到的结果：

节选自在线代码firingOrder .sql

```
SQL> SELECT * FROM temp_table
  2 ORDER BY num_col;
NUM_COL CHAR_COL
-----
1 Before Statement: counter = 0
```

```

2 Before Row 3: counter = 1
3 Before Row 2: counter = 2
4 Before Row 1: counter = 3
5 After Row: counter = 4
6 Before Row 3: counter = 5
7 Before Row 2: counter = 6
8 Before Row 1: counter = 7
9 After Row: counter = 8
10 Before Row 3: counter = 9
11 Before Row 2: counter = 10
12 Before Row 1: counter = 11
13 After Row: counter = 12
14 Before Row 3: counter = 13
15 Before Row 2: counter = 14
16 Before Row 1: counter = 15
17 After Row: counter = 16
18 After Statement 2: counter = 17
19 After Statement 1: counter = 18

```

当每个触发器被激发时，该触发器将可以看到由其前期触发器实现的变更，以及到目前为止由DML语句对数据实现的变更。触发器的激发可以通过由每个触发器打印的记数器值来判断。（请看第5章中有关使用包变量的说明。）

同类触发器的激发顺序没有明确的定义。如前面所示的例子中，每个触发器都将可以看到其前面的触发器实施的变更。如果顺序非常重要的话，可以把所有的操作组合在一个触发器中。

注意 当你为表创建快照日志时，Oracle将自动地为该表创建一个 AFTER ROW触发器，该触发器在每个DML语句后更新日志文件。如果要创建额外的 AFTER ROW触发器的话，你一定要避免与系统创建的触发器发生冲突。除此之外，数据库系统还对触发器和快照有其他的限制。

2. 行级触发器的相关标识

行级触发器是按触发语句所处理的行激发的。在触发器内部，我们可以访问正在处理中的行的数据。这种访问是通过两个相关的标识符：:old和:new实现的。相关标识符是一种特殊的PL/SQL连接变量（bind variable），在该标识符前面的冒号说明它们是使用在嵌套PL/SQL中的宿主变量意义上的连接变量，而不是一般的PL/SQL变量。PL/SQL编译器将把这种变量按记录类型处理。

```
triggering_table%ROWTYPE
```

其中，triggering_table是定义触发器所在的表。因此，下面的引用：

```
:new.field
```

将只有在该字段位于触发表中时才是合法的。表 6-2总结了标识符:old和:new的含义。尽管从语法上讲，这两个标识符都按记录类型处理，但实际上不是这样的（该问题将在下文“伪记录”中介绍。）。正是这个原因，标识符:old和:new也被称为伪记录。

下载

表6-2 :old和:new相关标识符

触发语句	标识符:old	标识符:new
INSERT	无定义-所有字段为空 NULL	该语句结束时将插入的值
UPDAE	更新前行的原始值	该语句结束时将更新的值
DELETE	行删除前的原始值	无定义-所有字段为空 NULL

注意 标识符 :old对INSERT语句无定义，而标识符 :new对DELETE语句无定义。

PL/SQL编译器不会对在 INSERT语句中使用的:old和在DELETE语句中使用的:new标识符报错，编译的结果将使该字段为空。

**Oracle 8i 及
更高版本** Oracle8i定义了另外一个相关标识符 :parent。如果触发器定义在嵌套表中的话，标识符:old和:new就引用嵌套表中的行，而 :parent则引用其父表的当前行。有关详细信息，请参阅Oracle文档资料。

使用:old和:new相关标识符 下面所示的触发器 GenerateStudentID使用了标识符:new。该触发器是一个INSERT之前触发器，其目的是使用从序列 studengt_seuence中生成的值来填写 ID字段。

节选自在线代码GenerateStudentID .sql

```
CREATE OR REPLACE TRIGGER GenerateStudentID
  BEFORE INSERT OR UPDATE ON students
  FOR EACH ROW
BEGIN
  /* Fill in the ID field of students with the next value from
  student_sequence. Since ID is a column in students, :new.ID
  is a valid reference. */
  SELECT student_sequence.NEXTVAL
    INTO :new.ID
    FROM dual;
END GenerateStudentID;
```

触发器GenerateStudentID实际上是修改:new.ID的值，这就是:new标识符的用途之一，即当该语句实际运行时，在:new中的任何值都将被使用。使用触发器 GenerateStudentID，我们可以发布如下所示的INSERT语句：

节选自在线代码GenerateStudentID .sql

```
INSERT INTO students (first_name, last_name)
VALUES ('Lolita', 'Lazarus');
```

该语句可以正确执行。尽管我们没有为主键列 ID指定值（该值是语句需要的），但触发器可以提供它。事实上，即使我们为该 ID指定了一个值，该值也将被忽略不记，因为触发器将改变该值。如果我们执行下面的命令：

节选自在线代码GenerateStudentID .sql

```
INSERT INTO students (ID, first_name, last_name)
VALUES (-7, 'Zelda', 'Zoom');
```

该ID列将被来自于student_sequence.NEXTVAL的值填充，而不是值-7.

按上面的操作结果，我们不能在行级触发器之后改变 :new，其原因是该语句已被处理了。总的来说，对 :new 的修改只能在行级触发器之前修改。:old 具有只读属性，只能读入。

记录：new 和 :old 只在行级触发器内部合法。如果企图引用在语句级触发器之内的： new 或 :old 的话，编译器将报错。由于语句级的触发器只运行一次，即使存在很多被语句处理过的行的话，new 和 :old 也没有定义。编译器不知道该引用那一行。

伪记录 尽管 :new 和 :old 从语法上讲按 triggering_table%ROWTYPE 的记录来处理，但实际处理却不一样。这样一来，应该是合法的记录操作对于 :new 和 :old 来说就变成了非法操作。例如，这两个记录不能按全记录赋值。而只用在其内部的个别字段可以赋值。下面的程序可以说明上述问题：

```
节选自在线代码pseudoRecords .sql
CREATE OR REPLACE TRIGGER TempDelete
  BEFORE DELETE ON temp_table
  FOR EACH ROW
DECLARE
  v_TempRec temp_table%ROWTYPE;
BEGIN
  /* This is not a legal assignment, since :old is not truly
   a record. */
  v_TempRec := :old;

  /* We can accomplish the same thing, however, by assigning
   the fields individually. */
  v_TempRec.char_col := :old.char_col;
  v_TempRec.num_col := :old.num_col;
END TempDelete;
```

除此之外，:old 和 :new 记录不能传递到接收 triggering_table%ROWTYPE 的过程或函数中。

引用子句 我们可以根据需要使用子句 REFERENCING 来为 :new 和 :old 指定一个不同的名称。该子句可以在触发事件之后，WHEN 子句之前使用，其语法如下：

```
REFERENCING [OLD AS :old_name] [NEW AS :new_name]
```

在触发器体内，我们可以使用 :old_name 和 :new_name 来代替 :old 和 :new。下面是触发器 GenerateStudentID 的另一种版本，该版本使用 REFERENCING 来把 :new 作为 :new_student 引用。

```
节选自在线代码GenerateStudentID .sql
CREATE OR REPLACE TRIGGER GenerateStudentID
  BEFORE INSERT OR UPDATE ON students
  REFERENCING new AS new_student
  FOR EACH ROW
BEGIN
  /* Fill in the ID field of students with the next value from
   student_sequence. Since ID is a column in students,
   :new_student.ID is a valid reference. */
  SELECT student_sequence.nextval
  INTO :new_student.ID
  FROM dual;
```

```
END GenerateStudentID;
```

3. WHEN子句

WHEN子句只适用于行级触发器。如果使用该子句的话，触发器体将只对满足由 WHEN子句说明的条件的行执行。WHEN子句的语法是：

```
WHEN trigger_condition
```

其中，trigger_condition是逻辑表达式。该表达式将为每行求值。:new和:old记录可以在trigger_condition内部引用，但不需使用冒号。该冒号只在触发器体内有效。例如，触发器CheckCredits的体只在当前的学生得到的学分超出20时才运行：

```
CREATE OR REPLACE TRIGGER CheckCredits
  BEFORE INSERT OR UPDATE OF current_credits ON students
  FOR EACH ROW
  WHEN (new.current_credits > 20)
  BEGIN
    /* Trigger body goes here. */
  END;
```

触发器CheckCredits也可写为下列代码：

```
CREATE OR REPLACE TRIGGER CheckCredits
  BEFORE INSERT OR UPDATE OF current_credits ON students
  FOR EACH ROW
  BEGIN
    IF :new.current_credits > 20 THEN
      /* Trigger body goes here. */
    END IF;
  END;
```

4. 触发器谓语：INSERTING、UPDATING和DELETING

6.1.1节中讨论的触发器UpdateMajorStats就是INSERT、UPDATE和DELETE触发器。在这种触发器的内部（为不同的DML语句激发的触发器）有三个可用来确认执行何种操作的逻辑表达式。这些表达式的谓语是INSERTING、UPDATING、DELETING。下面说明了每个谓词的含义。

表达式谓语

状 态

INSERTING

如果触发语句是INSERT的话，则为真值（TRUE），否则为FALSE

UPDATING

如果触发语句是UPDATE的话，则为真值（TRUE），否则为FALSE

DELETING

如果触发语句是DELETE的话，则为真值（TRUE），否则为FALSE

注意 Oracle8i定义了额外的可以从触发器体内调用的函数，这种函数类似于触发器表达式谓语。有关详细内容，请看6.2.3节中的“事件属性函数”。

触发器LogRSChanges使用上述表达式的谓语来记录表registered_students发生的所有变化。除了记录这些信息外，它还记录对表进行变更的用户名。该触发器的记录存放在表RS_audit中，其结构如下：

节选自在线代码relTables .sql

```
CREATE TABLE RS_audit (
```

```

change_type      CHAR(1)          NOT NULL,
changed_by       VARCHAR2(8)      NOT NULL,
timestamp        DATE NOT        NULL,
old_student_id   NUMBER(5),
old_department   CHAR(3),
old_course       NUMBER(3),
old_grade        CHAR(1),
new_student_id   NUMBER(5),
new_department   CHAR(3),
new_course       NUMBER(3),
new_grade        CHAR(1)
);

```

触发器LogRSChanges的创建语句如下：

节选自在线代码LogRSChanges.sql

```

CREATE OR REPLACE TRIGGER LogRSChanges
  BEFORE INSERT OR DELETE OR UPDATE ON registered_students
  FOR EACH ROW
DECLARE
  v_ChangeType CHAR(1);
BEGIN
  /* Use 'I' for an INSERT, 'D' for DELETE, and 'U' for UPDATE. */
  IF INSERTING THEN
    v_ChangeType := 'I';
  ELSIF UPDATING THEN
    v_ChangeType := 'U';
  ELSE
    v_ChangeType := 'D';
  END IF;
  /* Record all the changes made to registered_students in
  RS_audit. Use SYSDATE to generate the timestamp, and
  USER to return the userid of the current user. */
  INSERT INTO RS_audit
  (change_type, changed_by, timestamp,
  old_student_id, old_department, old_course, old_grade,
  new_student_id, new_department, new_course, new_grade)
  VALUES
  (v_ChangeType, USER, SYSDATE,
  :old.student_id, :old.department, :old.course, :old.grade,
  :new.student_id, :new.department, :new.course, :new.grade);
END LogRSChanges;

```

触发器常用于进行数据检查，就象触发器 LogRSChanges的功能那样。然而，检查还只是数据库的一部分用途，触发器还可用于更灵活和更用户化的记录。我们还可以对触发器 LogRSChanges进行修改，例如，使用它来记录仅由某些人做的修改。我们还可以使用该触发器来检查是否用户有权变更数据并在没有授权的情况下引发异常（使用 RAISE_APPLICATION_ERROR）。

下载

6.2.2 创建替代触发器

Oracle 8 及
更高版本

DML触发器是除去执行INSERT, UPDATE或DELETE操作外（在这些语句之前或之后）还要被激活运行的触发器，而替代触发器则被激发来代替执行 DML语句。除此之外，替代触发器还可以定义在视图上，而 DML触发器只能定义在表上。替代触发器的用途有两类：

- 允许对无法变更的视图进行修改。
- 修改视图中嵌套表列的列。

我们将在本节讨论第一种应用，其他信息请看本书第 14 章。

注意 在Oracle8i的8.1.5版中，替代触发器功能只能在其企业版中使用。在将来的版本中，有可能在其他的版本中也提供该功能。

1. 可变更的与不可变更的视图

可变更视图是可以发布 DML命令的视图。一般来说，视图如果不包括下列命令参数的话就是一个可变更视图：

- Set operators(UNION,UNION ALL,MINUS)
- Aggregate functions(SUM,AVG,etc.)
- GROUP BY,CONNECT BY,或START WITH clauses
- 操作数DISTINCT
- 联合

然而，也确实有包括联合的视图是可以变更的。总的来说，如果对该视图的 DML操作每次只变更基表，并且DML操作满足了表 6-3 的条件，那么，该联合视图也是可变更的。如果一个视图是不可变更的，则我们可以在其上写一个替代触发器来执行正确的操作，从而使该视图可变更。如果需要进行额外处理的话，替代触发器也可以写在可变更视图上。

表6-3引用了保留字表。如果一个表与另一个表联合后，其原始表中的关键字在联合后的表中仍然是关键字的话，该表就是一个关键字保留表（key-preserved）。

表6-3 可变更的联合视图

DML操作	可执行条件
INSERT	该语句没有显式或隐式地引用非保留字表的列
UPDATE	更新的列映射到保留字表的列中
DELETE	在联合中仅有一个保留字表

2. 替代触发器案例

请考虑我们在 6.1.2 节中介绍过的的视图 classes_rooms：

节选自在线代码 insteadOf.sql

```
CREATE OR REPLACE VIEW classes_rooms AS
  SELECT department, course, building, room_number
  FROM rooms, classes
 WHERE rooms.room_id = classes.room_id;
```

如上所见，对该视图的 INSERT操作是合法的，尽管可以合法地对该视图执行 UPDATE或

DELETE操作，但这些命令不能实现正确的操作。例如，从 classes_rooms发布的DELETE命令将删除表classes中对应的行，什么是classes_rooms的正确DML操作呢？这与程序的逻辑要求有关。假设这些命令有下列含义：

操作	含义
INSERT	把新近插入的班级赋予新插入的教室。该操作将导致对表 classes的更新
UPDATE	变更赋予班级的教室。该操作将导致 classes或rooms的更新，取决于表 classes_rooms的哪一列有变更
DELETE	从删除的班级中清除教室的ID。该操作将导致表 classes的更新，将ID置为空NULL

下面所示的触发器 ClassesRoomsInstead实施了上述规则并允许对表 classes_rooms执行正确的DML操作。

```
节选自在线代码ClassesRoomInstead.sql
CREATE OR REPLACE TRIGGER ClassesRoomsInstead
  INSTEAD OF INSERT OR UPDATE OR DELETE ON classes_rooms
  FOR EACH ROW
DECLARE
  v_roomID rooms.room_id%TYPE;
  v_UpdatingClasses BOOLEAN := FALSE;
  v_UpdatingRooms BOOLEAN := FALSE;

  -- Local function that returns the room ID, given a building
  -- and room number. This function will raise ORA-20000 if the
  -- building and room number are not found.
  FUNCTION GetRoomID(p_Building IN rooms.building%TYPE,
                     p_Room IN rooms.room_number%TYPE)
    RETURN rooms.room_id%TYPE IS

    v_RoomID rooms.room_id%TYPE;
BEGIN
  SELECT room_id
    INTO v_RoomID
   FROM rooms
  WHERE building = p_Building
    AND room_number = p_Room;
  RETURN v_RoomID;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20000, 'No matching room');
END getRoomID;

  -- Local procedure that checks whether the class identified by
  -- p_Department and p_Course exists. If not, it raises
  -- ORA-20001.
```

下载

```
PROCEDURE VerifyClass(p_Department IN classes.department%TYPE,
                      p_Course IN classes.course%TYPE) IS
    v_Dummy NUMBER;
BEGIN
    SELECT 0
    INTO v_Dummy
    FROM classes
    WHERE department = p_Department
      AND course = p_Course;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20001,
            p_Department || ' ' || p_Course || ' doesn''t exist');
END verifyClass;

BEGIN
    IF INSERTING THEN
        -- This essentially assigns a class to a given room. The logic
        -- here is the same as the "updating rooms" case below: First,
        -- determine the room ID:
        v_RoomID := GetRoomID(:new.building, :new.room_number);
        -- And then update classes with the new ID.
        UPDATE CLASSES
            SET room_id = v_RoomID
            WHERE department = :new.department
              AND course = :new.course;

    ELSIF UPDATING THEN
        -- Determine if we are updating classes, or updating rooms.
        v_UpdatingClasses := (:new.department != :old.department) OR
                            (:new.course != :old.course);
        v_UpdatingRooms := (:new.building != :old.building) OR
                           (:new.room_number != :old.room_number);
        IF (v_UpdatingClasses) THEN
            -- In this case, we are changing the class assigned for a
            -- given room. First make sure the new class exists.
            VerifyClass(:new.department, :new.course);

            -- Get the room ID,
            v_RoomID := GetRoomID(:old.building, :old.room_number);

            -- Then clear the room for the old class,
            UPDATE classes
                SET room_ID = NULL
                WHERE department = :old.department
                  AND course = :old.course;
```

```
-- And finally assign the old room to the new class.
UPDATE classes
    SET room_ID = v_RoomID
    WHERE department = :new.department
    AND course = :new.course;
END IF;

IF v_UpdatingRooms THEN
    -- Here, we are changing the room for a given class. This
    -- logic is the same as the "inserting" case above, except
    -- that classes is updated with :old instead of :new.
    -- First, determine the new room ID.
    v_RoomID := GetRoomID(:new.building, :new.room_number);

    -- And then update classes with the new ID.
    UPDATE CLASSES
        SET room_id = v_RoomID
        WHERE department = :old.department
        AND course = :old.course;
    END IF;

ELSE
    -- Here, we want to clear the class assigned to the room,
    -- without actually removing rows from the underlying tables.
    UPDATE classes
        SET room_ID = NULL
        WHERE department = :old.department
        AND course = :old.course;
    END IF;
END ClassesRoomsInstead;
```

注意 子句FOR EACH ROW是替代触发器的选择项。不管该子句是否存在，所有的替代触发器都是行级的。

触发器ClassesRoomsInstead使用触发器判定来决定将要执行的 DML操作，并且采取相应的动作。图6-1演示了表classes、rooms和classes_rooms的原始内容。假设我们发布了下列 INSERT命令：

节选自在线代码ClassesRoomInstead.sql

```
INSERT INTO classes_rooms
    VALUES ('MUS', 100, 'Music Building', 200);
```

该触发器对表 classes进行更新以便反映新的教室。新的 classes表如图 6-2所示。现在我们假设发布了下列UPDATE命令：

节选自在线代码ClassesRoomInstead.sql

```
UPDATE classes_rooms
```

下载

```
SET department = 'NUT', course = 307
WHERE building = 'Building 7' AND room_number = 201;
```

我们看到表classes又一次被更新，更新后的表反映了新的变更。从该表中我们可以看到，历史系101课程没有分配教室，营养系307课程分配到了原历史系101课程的教室。这种变化反映在图6-3所示的表中。最后，假设我们发布如下的DELETE命令：

节选自在线代码ClassesRoomInstead.sql

```
DELETE FROM classes_rooms
WHERE building = 'Building 6';
```

classes			rooms			classes_rooms			
Dept	Course	Room ID	Room ID	Building	Room Number	Dept	Course	Building	Room Number
HIS	101	20000	20000	Building 7	301	HIS	101	Building 7	301
CS	101	20001	20001	Building 6	101	CS	101	Building 6	101
ECN	203	20002	20002	Building 6	150	ECN	203	Building 6	150
CS	102	20003	20003	Building 6	160	CS	102	Building 6	160
HIS	301	20004	20004	Building 6	170	HIS	301	Building 6	170
MUS	410	20005	20005	Music Building	100	MUS	410	Music Building	100
MUS	100		20006	Music Building	200				
ECN	101	20007	20007	Building 7	300	ECN	101	Building 7	300
NUT	307	20008	20008	Building 7	310	NUT	307	Building 7	310

图6-1 表classesrooms和classes_rooms的原始内容

classes			rooms			classes_rooms			
Dept	Course	Room ID	Room ID	Building	Room Number	Dept	Course	Building	Room Number
HIS	101	20000	20000	Building 7	301	HIS	101	Building 7	301
CS	101	20001	20001	Building 6	101	CS	101	Building 6	101
ECN	203	20002	20002	Building 6	150	ECN	203	Building 6	150
CS	102	20003	20003	Building 6	160	CS	102	Building 6	160
HIS	301	20004	20004	Building 6	170	HIS	301	Building 6	170
MUS	410	20005	20005	Music Building	100	MUS	410	Music Building	100
MUS	100	20006	20006	Music Building	200				
ECN	101	20007	20007	Building 7	300	ECN	101	Building 7	300
NUT	307	20008	20008	Building 7	310	NUT	307	Building 7	310

图6-2 执行插入操作后的各表的内容

对表classes的更新操作把原在6楼教室的room_ID设置为空。更新后的表如图6-4所示。请注

意，经过前面所有的DML语句，表rooms保持没有变更，只有表classes进行了更新。

classes			rooms			classes_rooms			
Dept	Course	Room ID	Room ID	Building	Room Number	Dept	Course	Building	Room Number
NUT	307	20000	20000	Building 7	201	HIS	101	Building 7	201
CS	101	20001	20001	Building 6	101	CS	101	Building 6	101
ECN	203	20002	20002	Building 6	150	ECN	203	Building 6	150
CS	102	20003	20003	Building 6	160	CS	102	Building 6	160
HIS	301	20004	20004	Building 6	170	HIS	301	Building 6	170
MIS	410	20005	20005	Music Building	100	MIS	410	Music Building	100
MIS	100	20006	20006	Music Building	200	ECN	101	Building 7	300
ECN	101	20007	20007	Building 7	300	NUT	307	Building 7	201
HIS	101		20000	Building 7	310	MIS	100	Music Building	200

图6-3 更新后的表的内容

classes			rooms			classes_rooms			
Dept	Course	Room ID	Room ID	Building	Room Number	Dept	Course	Building	Room Number
NUT	307	20000	20000	Building 7	201	MIS	410	Music Building	100
CS	101		20001	Building 6	101	ECN	101	Building 7	300
ECN	203		20002	Building 6	150	NUT	307	Building 7	201
CS	102		20003	Building 6	160	MIS	100	Music Building	200
HIS	301		20004	Building 6	170				
MIS	410	20005	20005	Music Building	100				
MIS	100	20006	20006	Music Building	200				
ECN	101	20007	20007	Building 7	300				
HIS	101		20000	Building 7	310				

图6-4 删除操作后表的内容

6.2.3 创建系统触发器

正如我们在前几节所看到的，DML和替代触发器都在（或代替）DML事件，即INSERT、UPDATE、DELETE语句上激活。而系统触发器可以在两种不同的事件即DDL或数据库上激活。DDL事件包括CREATE、ALTER或DROP语句，而数据库事件包括服务器的启动或关闭，用户的登录或退出，以及服务器错误。创建系统触发器的语法如下：

```
CREATE [OR REPLACE] TRIGGER [ schema.] trigger_name
{BEFORE | AFTER}
{ ddl_event_list / database_event_list }
ON {DATABASE | [ schema.]SCHEMA}
[ when_clause ]
```

Oracle 8i 及
更高版本

```
trigger_body;
```

其中，`ddl_event_list`是一个或多个DDL事件（事件之间用OR分隔），`database_event_list`是一个或多个数据库事件（事件之间用‘OR’分隔）。

表6-4说明了DDL和数据库事件的种类以及这些事件出现的时机（之前或之后）。系统不支持替代系统触发器，`TRUNCATE`没有对应的数据库事件。

注意 创建系统触发器必须具有系统权限 `ADMINISTER DATABASE TRIGGER`。本章6.2.4节中的“触发器权限”提供了详细信息。

表6-4 系统DLL和数据库事件

事 件	允 许 时 机	说 明
启动	之后	实例启动时激活
关闭	之前	实例关闭时激活。如果数据库非正常关闭（如关闭故障），则该事件不激活
服务器错误	之后	只要有该类错误就激活
登录	之后	在用户成功连接数据库后激活
注销	之前	在用户注销开始时激活
创建	之前,之后	在创建模式对象之前或之后激活
撤消	之前,之后	在创建模式对象撤消之前或之后激活
变更	之前,之后	在创建模式对象变更之前或之后激活

1. 数据库与模式触发器的比较

系统触发器可以在数据库级或模式级定义。数据库级的触发器不管触发事件何时发生都将激活，而模式级触发器只有在指定的模式的触发事件发生时才会激活。关键字 `DATABASE` 和 `SCHEMA` 决定了给定系统触发器的等级。如果没有用关键字 `SCHEMA` 来说明模式，则以触发器所属的模式为默认模式。例如，假设我们在作为数据库的联机用户 UserA 时，创建了下列的触发器：

```
节选自在线代码DatabaseSchema.sql
CREATE OR REPLACE TRIGGER LogUserAConnects
  AFTER LOGON ON SCHEMA
BEGIN
  INSERT INTO example.temp_table
    VALUES (1, 'LogUserAConnects fired!');
END LogUserAConnects;
```

字符串 `LogUserAConnects` 将在 UserA 与数据库建立连接时记录在表 `temp_table` 中。我们可以通过创建下面的触发器在用户 UserB 与数据库建立连接时为用户 UserB 做相同的记录：

```
节选自在线代码DatabaseSchema.sql
CREATE OR REPLACE TRIGGER LogUserBConnects
  AFTER LOGON ON SCHEMA
BEGIN
  INSERT INTO example.temp_table
    VALUES (2, 'LogUserBConnects fired!');
END LogUserBConnects;
```

最后，我们可以在以 example 的身份与数据库建立连接时创建下面的触发器。由于 LogAllConnects 触发器是数据库级触发器，所以它可以把所有的数据库连接都记录在数据库中。

节选自在线代码 DatabaseSchema.sql

```
CREATE OR REPLACE TRIGGER LogAllConnects
  AFTER LOGON ON DATABASE
BEGIN
  INSERT INTO example.temp_table
    VALUES (3, 'LogAllConnects fired!');
END LogAllConnects;
```

注意 我们必须首先创建 UserA 和 UserB，并在运行上面的例子前把相应的权限赋予这些用户。请看程序 DatabaseSchema.sql 中的实现代码。

现在我们可以在 SQL *Plus 会话中看到不同触发器的影响。

节选自在线代码 DatabaseSchema.sql

```
SQL> connect UserA/UserA
Connected.
SQL> connect UserB/UserB
Connected.
SQL> connect example/example
Connected.
SQL>
SQL> SELECT * FROM temp_table;
  NUM_COL CHAR_COL
-----
          3 LogAllConnects fired!
          2 LogUserBConnects fired!
          3 LogAllConnects fired!
          3 LogAllConnects fired!
          1 LogUserAConnects fired!
```

LogAllConnects 触发器被激活了三次（每个连接激活一次），而 LogUserAConnects 和 LogUserBConnects 按预期只激活了一次。

注意 STARTUP 和 SHUTDOWN 触发器只与数据库级有关。虽然在模式级创建它们是合法的，但它们不会被激活。

2. 事件属性函数

系统触发器有几个内部的属性函数可供使用。这些函数类似于我们在前一节介绍的触发器参数（INSETTING, UPDATING, DELETING），这些参数允许触发器体获得有关触发事件的信息。尽管从其他的 PL/SQL 块中调用这些函数是合法的（在系统触发器中不必这样调用），但有时这些函数也会返回我们不希望的结果。表 6-5 对这些事件属性函数做了说明。

我们在本章的开始部分介绍的触发器 LogCreations 中使用了这些属性函数。与触发器参数不同，事件属性函数是 SYS 拥有的独立 PL/SQL 函数。系统没有为这些函数指定默认的替代名称，所以为了识别这些函数，在程序中必须在它们的前面加上前缀 SYS。

节选自在线代码 LogCreations.sql

下载

```

CREATE OR REPLACE TRIGGER LogCreations
AFTER CREATE ON SCHEMA
BEGIN
  INSERT INTO ddl_creations (user_id, object_type, object_name,
                             object_owner, creation_date)
  VALUES (USER, SYS.DICTIONARY_OBJ_TYPE, SYS.DICTIONARY_OBJ_NAME,
          SYS.DICTIONARY_OBJ_OWNER, SYSDATE);
END LogCreations;

```

表6-5 事件属性函数

属性函数	数据类型	可应用的系统事件	说明
SYSEVENT	VARCHAR2(20)	所有事件	返回激活触发器的系统事件
INSTANCE_NUM	NUMBER	所有事件	返回当前实例号。在不运行OPS的情况下，该号为1
DATABASE_NAME	VARCHAR2(50)	所有事件	返回当前数据库名
SERVER_ERROR	NUMBER	SERVERERROR	接收一个NUMBER类型的参数，返回由该参数所指示的错误堆栈中相应位置的错误。错误堆栈的顶部对应于位置1
IS_SEVERERROR	BOOLEAN	SERVERERROR	接收一个错误号作为参数，如果所指示的Oracle错误返回在堆栈中，则返回真值(TRUE)
LOGIN_USER	VARCHAR2(30)	所有事件	返回激活触发器的用户的userid
DICTIONARY_OB_J_TYPE	VARCHAR2(20)	CREATE, DROP, ALTER	返回激活触发器的DDL操作使用的字典对象的类型
DICTIONARY_OB_J_NAME	VARCHAR2(30)	CREATE, DROP, ALTER	返回激活触发器的DDL操作使用的字典对象的名称
DICTIONARY_OB_J_OWNER	VARCHAR2(30)	CREATE, DROP, ALTER	返回激活触发器的DDL操作使用的字典对象的拥有者
DES_ENCRYPTED_PASSWORD	VARCHAR2(30)	CREATE用户或ALTER	返回正在创建或变更用户的使用DES加密的口令

3. 使用 SERVERERROR事件

事件 SERVERERROR可以用于跟踪数据库中发生的错误。其错误代码可以使用触发器内部的 SERVER_ERROR属性函数取出。该函数可以让用户确定堆栈中的错误码。然而，该函数不能返回与该错误码相关的错误信息。

上述缺点可以通过使用过程 DBMS_UTILITY.FORMAT_ERROR_STACK来解决。尽管触发器本身不会引发错误，但借助于该过程，我们可以使用 PL/SQL来访问错误堆栈。下面是演示上述过程的例子，该程序将错误记录在下面的表中：

```

节选自在线代码LogErrors.sql
CREATE TABLE error_log (
  timestamp      DATE,
  username       VARCHAR2(30),
  instance       NUMBER,
  database_name VARCHAR2(50),
  error_stack    VARCHAR2(2000)
);

```

我们可以创建一个如下所示的插入表 error_log 的触发器：

节选自在线代码 LogErrors.sql

```
CREATE OR REPLACE TRIGGER LogErrors
    AFTER SERVERERROR ON DATABASE
BEGIN
    INSERT INTO error_log
        VALUES (SYSDATE, SYS.LOGIN_USER, SYS.INSTANCE_NUM, SYS.
DATABASE_NAME, DBMSUTILITYFORMAT_ERROR_STACK);
END LogErrors;
```

最后，我们可以生成几个错误并来看过程 LogErrors 怎样来记录这些错误信息。请注意可以捕捉 SQL 中的错误、运行时 PL/SQL 错误和编译时的 PL/SQL 错误。

节选自在线代码 LogErrors.sql

```
SQL> SELECT * FROM non_existent_table;
SELECT * FROM non_existent_table
*
ERROR at line 1:
ORA-00942: table or view does not exist
SQL> BEGIN
 2 INSERT INTO non_existent_table VALUES ('Hello!');
 3 END;
 4 /
INSERT INTO non_existent_table VALUES ('Hello!');
*
ERROR at line 2:
ORA-06550: line 2, column 15:
PLS-00201: identifier 'NON_EXISTENT_TABLE' must be declared
ORA-06550: line 2, column 3:
PL/SQL: SQL Statement ignored
SQL> BEGIN
 2 -- This is a syntax error!
 3 DELETE FROM students
 4 END;
 5 /
END;
*
ERROR at line 4:
ORA-06550: line 4, column 1:
PLS-00103: Encountered the symbol "END" when expecting one of the
following:
. @ ; return RETURNING_ <an identifier>
<a double-quoted delimited-identifier> partition where
The symbol ";" was substituted for "END" to continue.
SQL> SELECT *
 2 FROM error_log;

TIMESTAMP USERNAME INSTANCE DATABASE
```

```
-----  
ERROR_STACK  
-----  
30-AUG-99 EXAMPLE      1 V815  
ORA-00942: table or view does not exist  
  
30-AUG-99 EXAMPLE      1 V815  
ORA-06550: line 2, column 15:  
PLS-00201: identifier 'NON_EXISTENT_TABLE' must be declared  
ORA-06550: line 2, column 3:  
PL/SQL: SQL Statement ignored  
  
30-AUG-99 EXAMPLE      1 V815  
ORA-06550: line 4, column 1:  
PLS-00103: Encountered the symbol "END" when expecting one of  
the following:  
  
. @ ; return RETURNING_ <an identifier>  
<a double-quoted delimited-identifier> partition where  
The symbol ";" was substituted for "END" to continue.
```

4. 系统触发器和事务

系统触发器的事务特性与触发事件有关。系统触发器可以作为基于触发器正常结束时提交的独立事务激活，也可以作为当前用户事务的一部分激活。触发器 STARTUP, SHUTDOWN, SERVERERROR 和 LOGON 都是由独立事务激活的，而 LOGOFF 和 DDL 触发器则作为当前事务的一部分被激活。

需要注意的是，触发器实现的任务将被提交处理。在使用 DDL 触发器的情况下，当前事务（也就是 CREATE、ALTER 或 DROP 语句）将自动提交。触发器 LOGOFF 的操作也将作为会话中最后事务的一部分提交。

注意 由于系统触发器一般都要提交，因此把这类触发器声明为自主事务是没有意义的。请看本书第 11 章介绍自主事务的内容。

5. 系统触发器和 WHEN 子句

就象 DML 触发器一样，系统触发器可以使用 WHEN 子句来指定触发器激活条件。然而，对每一种系统触发器所指定的条件类型有如下限制：

- STARTUP 和 SHUTDOWN 触发器不能带有任何条件。
- SERVERERROR 触发器可以使用 ERRNO 测试来检查特定的错误。
- LOGON 和 LOGOFF 触发器可以使用 USERID 或 USERNAME 测试来检查用户标识或用户名。
- DDL 触发器可以检查正在修改对象的名称和类型。

6.2.4 其他触发器问题

我们在本节将讨论有关触发器的最后一些问题。其中包括触发器名称的命名空间（ Name-

space), 使用触发器的各种限制, 和不同的触发器体。本节的最后将讨论与触发器有关的权限问题。

1. 触发器名称

触发器的命名空间不同于其他子程序的命名空间。所谓命名空间就是一组合法的可供对象作为名字使用的标识符。过程, 包和表都共享同一个命名空间, 这就是说, 在一个数据库模式范围内, 同一命名空间内的所有的对象必须具有唯一的名称。例如, 把同一个名字同时赋予一个过程和包就是非法的。

然而, 触发器隶属于一个独立的命名空间。也就是说, 触发器可以有与表或过程相同的名称。在一个模式范围内, 然而, 给定的名称只能用于一个触发器。例如, 我们可以创建一个建立在表major_stats上叫做major_stats触发器, 当但是, 如果再创建一个叫做major_stats的过程就是非法操作, 下面的SQL *Plus会话演示了上述规则:

节选自在线代码Samename.sql

```
SQL> -- Legal, since triggers and tables are in different namespaces.
SQL> CREATE OR REPLACE TRIGGER major_stats
  2    BEFORE INSERT ON major_stats
  3 BEGIN
  4    INSERT INTO temp_table (char_col)
  5      VALUES ('Trigger fired!');
  6 END major_stats;
  7 /
Trigger created.

SQL> -- Illegal, since procedures and tables are in the same namespace.
SQL> CREATE OR REPLACE PROCEDURE major_stats AS
  2 BEGIN
  3    INSERT INTO temp_table (char_col)
  4      VALUES ('Procedure called!');
  5 END major_stats;
  6 /
CREATE OR REPLACE PROCEDURE major_stats AS
*
ERROR at line 1:
ORA-00955: name is already used by an existing object
```

提示 尽管触发器和表可以共用一个名称, 但我们不推荐使用这种命名方法。最好是给每个触发器和表都赋予一个标识其功能的唯一的名称, 也可以在触发器的名称加上如TRG_之类的前缀。

1. 对触发器的限制

触发器的体是一个PL/SQL块。(Oracle8i允许其他类型的触发器体, 下节将讨论。)除去下面的限制外, 在PL/SQL块中可以使用的语句也可以使用在触发器的体中:

- 触发器不能发布任何事务控制语句, 如 COMMIT、ROLLBACK、SAVEPOINT或SET TRANSACTION。PL/SQL编译器允许触发器体中出现上述控制语句, 但当该触发器激活时, 将出现错误提示。这是因为该触发器是作为触发语句的执行部分被激活, 并且与触发语句位于同一个事务中。当触发语句被提交或重新运行时, 则该触发器所做的工作也将被

提交或重新开始。(在Oracle8i下，我们可以创建作为自动事务运行的触发器，这时，触发器所做的工作就可以独立于触发语句的状态而独立提交或返回起始点。本书第11章对自动事务做了详细介绍。)

- 与上一条类似，由触发器体调用的任何过程或函数都不能发布任何事务控制命令(除非在Oracle8i下把它们声明为自动类型)。
- 触发器体不能声明任何LONG或LONG RAW变量。同样，:new和:old也不能引用定义触发器所用表的LONG或LONG RAW类型的列。
- 在Oracle8及更高版本中，触发器体内的代码可以引用和使用LOB(大型对象)的列，但不能修改该列的值。上述限制也适用于对象列。

除此之外，还有对触发器访问的表的操作限制。根据触发器类型和对该表的限制，有时，表可能要进行调整。

3. 触发器体

**Oracle 8i 及
更高版本** 在Oracle8i之前的版本中，触发器的体必须是PL/SQL块。而在Oracle8i下，触发器的体可以由调用语句组成，所调用的过程子程序可以是PL/SQL存储子程序，或是C语言使用的包(wrapper)，以及Java程序。借助于这种选择，我们可以创建一个其基础代码是用Java编制的触发器。例如，假设我们要把数据库的连接和断开信息记录在下面的表中：

节选自在线代码relGTables.sql

```
CREATE TABLE connect_audit (
    user_name VARCHAR2(30),
    operation VARCHAR2(30),
    timestamp DATE);
```

现在，我们用下面的包来记录连接和断开信息：

节选自在线代码LogPkg.sql

```
CREATE OR REPLACE PACKAGE LogPkg AS
    PROCEDURE LogConnect(p_UserID IN VARCHAR2);
    PROCEDURE LogDisconnect(p_UserID IN VARCHAR2);
END LogPkg;
CREATE OR REPLACE PACKAGE BODY LogPkg AS
    PROCEDURE LogConnect(p_UserID IN VARCHAR2) IS
        BEGIN
            INSERT INTO connect_audit (user_name, operation, timestamp)
                VALUES (p_UserID, 'CONNECT', SYSDATE);
    END LogConnect;
    PROCEDURE LogDisconnect(p_UserID IN VARCHAR2) IS
        BEGIN
            INSERT INTO connect_audit (user_name, operation, timestamp)
                VALUES (p_UserID, 'DISCONNECT', SYSDATE);
    END LogDisconnect;
END LogPkg;
```

包LongPkg.LogConnect和LogPkg.LogDisconnect都以用户名作为其入口参数，并在表connect_audit中插入一行。最后，我们可以从触发器LOGON和LOGOFF中按下列代码调用这两

个触发器：

```
节选自在线代码LogConnects.sql
CREATE OR REPLACE TRIGGER LogConnects
    AFTER LOGON ON DATABASE
    CALL LogPkg.LogConnect(SYS.LOGIN_USER)
/

CREATE OR REPLACE TRIGGER LogDisconnects
    BEFORE LOGOFF ON DATABASE
    CALL LogPkg.LogDisconnect(SYS.LOGIN_USER)
/
```

注意 由于LogConnect和LogDisconnect都是系统数据库触发器，(与模式相反)，创建上述触发器必须具有系统权限ADMINISTER DATABASE TRIGGER。

LogConnect和LogDisconnect的触发器体只是简单的调用语句，用来指出待执行的过程，当前用户是它们的唯一入口参数。在前面的例子中，语句CALL的目标是标准的PL/SQL打包过程。然而，语句CALL的目标也可以是简单的C语言使用的包或Java外部例程使用的已包装。例如，假设我们把下面的Java类载入到数据库中：

```
节选自在线代码Loger.sql
import java.sql.*;
import oracle.jdbc.driver.*;
public class Logger {
    public static void LogConnect(String userID)
        throws SQLException {
        // Get default JDBC connection
        Connection conn = new OracleDriver().defaultConnection();

        String insertString =
            "INSERT INTO connect_audit (user_name, operation, timestamp)" +
            " VALUES (?, 'CONNECT', SYSDATE)";

        // Prepare and execute a statement that does the insert
        PreparedStatement insertStatement =
            conn.prepareStatement(insertString);
        insertStatement.setString(1, userID);
        insertStatement.execute();
    }

    public static void LogDisconnect(String userID)
        throws SQLException {
        // Get default JDBC connection
        Connection conn = new OracleDriver().defaultConnection();

        String insertString =
            "INSERT INTO connect_audit (user_name, operation, timestamp)" +
```

```

    " VALUES (?, 'DISCONNECT', SYSDATE)";

// Prepare and execute a statement that does the insert
PreparedStatement insertStatement =
    conn.prepareStatement(insertString);
insertStatement.setString(1, userID);
insertStatement.execute();
}

}
}

```

如果我们接着再创建一个如下所示的作为该类包装的包 LogPkg:

节选自在线代码LogPkg2.sql

```

CREATE OR REPLACE PACKAGE LogPkg AS
    PROCEDURE LogConnect(p.UserID IN VARCHAR2);
    PROCEDURE LogDisconnect(p.UserID IN VARCHAR2);
END LogPkg;

CREATE OR REPLACE PACKAGE BODY LogPkg AS
    PROCEDURE LogConnect(p.UserID IN VARCHAR2) IS
        LANGUAGE JAVA
        NAME 'Logger.LogConnect(java.lang.String)';

    PROCEDURE LogDisconnect(p.UserID IN VARCHAR2) IS
        LANGUAGE JAVA
        NAME 'Logger.LogDisconnect(java.lang.String)';
END LogPkg;

```

我们可以使用相同的触发器来实现上述要求。读者请看本书第 10 章有关外部例程及如何把 Java 过程载入到数据库中的内容的介绍。

注意 触发器参数如INSERTING、UPDATING和DELETING，以及:new和:old相关标识符（还有:parent），只有在触发器体是完整的 PL/SQL 块而不是 CALL 语句的情况下才可以使用。

4. 触发器权限

下面的表 6-6 说明了五个适用于触发器的系统权限。除此之外，触发器的拥有者必须还具有对其引用的对象的权限。由于触发器是已经编译的对象（从 Oracle7 的 7.3 版本开始），所有权限都必须直接授权，不能通过角色授权。

表 6-6 与触发器有关的系统权限

系统权限	说 明
CREATE TRIGGER	授予在其自己的模式下创建触发器的权限
CREATE ANY TRIGGER	授予在任何模式（除了 SYS 外）下创建触发器的权限。在数据字典表中不推荐使用创建触发器
ALTER ANY TRIGGER	授予在任何模式（除了 SYS 外）下启用、禁用或编译数据库触发器的权限。注意，如果没有授予 CREATE ANY TRIGGER 权限，用户不能改变触发器代码

(续)

系统权限	说 明
DROP ANY TRIGGER	授予在任何模式下(除去SYS)撤消数据库触发器的权限
ADMINISTER DATABASE TRIGGER	授予创建或变更数据库系统触发器的权限(与当前模式相反)。所授的权限必须具有CREATE TRIGGER或CREATE ANY TRIGGER

6.2.5 触发器与数据字典

与存储子程序类似，数据字典视图包括了有关触发器和其执行状态的信息。这些视图必须在触发器创建或撤消时进行更新。

1. 数据字典视图

当创建了一个触发器时，其源程序代码存储在数据库视图 user_triggers中。该视图包括了触发器体，WHEN子句，触发表，和触发器类型。例如，下面的查询返回有关 UpdateMajorStats的信息：

```
SQL> SELECT trigger_type, table_name, triggering_event
  2      FROM user_triggers
  3     WHERE trigger_name = 'UPDATEMAJORSTATS';
TRIGGER_TYPE    TABLE_NAME      TRIGGERING_EVENT
-----
AFTER STATEMENT STUDENTS INSERT OR UPDATE OR DELETE
```

有关数据字典视图的详细介绍，请看附录C。

2. 撤消和禁止触发器

与过程和包相类似，触发器也可以被撤消。实现撤消功能的命令如下：

```
DROP TRIGGER triggername;
```

其中，triggername是触发器的名称。该命令把指定的触发器从数据字典中永久性地删除。类似于子程序，子句OR REPLACE可用在触发器的CREATE语句中。在这种情况下，要创建的触发器已存在的话，则先将其删除。

与过程和包不同的是，触发器可以被禁止使用。当触发器被禁止时，它仍将存储在数据字典中，但不再激活。禁止触发器的语句如下：

```
ALTER TRIGGER triggername{DISABLE | ENABLE};
```

其中，triggername是触发器的名称。当创建触发器时，所有触发器的默认值都是允许状态(ENABLED)。语句ALTER TRIGGER可以禁止，或再允许任何触发器。例如，下面的代码先禁止接着再允许激活触发器UpdateMajorStats：

```
SQL> ALTER TRIGGER UpdateMajorStats DISABLE;
Trigger altered.
```

```
SQL> ALTER TRIGGER UpdateMajorStats ENABLE;
Trigger altered.
```

在使用命令ALTER TABLE的同时加入ENABLE ALL TRIGGERS或DISABLE ALL

TRIGGERS子句可以将指定表的所有触发器禁止或允许。例如：

```
SQL> ALTER TABLE students
  2    ENABLE ALL TRIGGERS;
Table altered.

SQL> ALTER TABLE students
  2    DISABLE ALL TRIGGERS;
Table altered.
```

视图user_triggers的status列包括有ENABLED或DISABLED两个字符串用来指示触发器的当前状态。禁止一个触发器将不从其数据字典中删除。

3. p-code触发器

当包或子程序存储在数据字典中时，其编译后的中间代码将同该对象的源代码一起存储，对于触发器来说情况也一样。这就是说触发器不需重新编译就可调用，并且其相关信息也得到保存。因此，触发器也可以向包和子程序那样自动无效。当触发器处于无效状态时，该触发器将在下次激活时自动重编译。

在Oracle7的7.3版前，系统对触发器的处理是不同的。在数据字典中唯一存储的是触发器的源代码，而没有中间代码。结果，触发器每次从数据字典中载入时都要进行编译。虽然这样处理并不影响触发器的定义和使用，但降低了触发器的运行效率。

6.3 变异表

系统对触发器将要访问的表和列有一些限制。为了定义这些限制，有必要来理解什么是变异表（mutating table）和约束表（constraining table），变异表就是当前被DML语句修改的表。对触发器来说，变异表就是触发器在其上进行定义的表；由于执行DELETE CASCADE引用完整性约束而需要更新的表也是变异表。（有关引用完整性约束的详细信息，请看Oracle服务器参考手册。）约束表是一种需要实施引用完整性约束而读入的表。为了说明这些定义，请看表registered_students，该表的结构如下：

节选自在线代码relTables.sql

```
CREATE TABLE registered_students (
  student_id NUMBER(5) NOT NULL,
  department CHAR(3) NOT NULL,
  course NUMBER(3) NOT NULL,
  grade CHAR(1),
  CONSTRAINT rs_grade
    CHECK (grade IN ('A', 'B', 'C', 'D', 'E')),
  CONSTRAINT rs_student_id
    FOREIGN KEY (student_id) REFERENCES students (id),
  CONSTRAINT rs_department_course
    FOREIGN KEY (department, course)
      REFERENCES classes (department, course)
);
```

表registered_students有两个声明的引用完整性约束。在这种约束下，对表registered_students来说，表students和classes都是约束表。由于这些限制，表classes和studengts也需要由

DML语句修改或查询。同样，表 registered_students自身在DML语句的对其操作期间也是变异表。

触发器体中的SQL语句不能进行下列操作：

- 读或修改触发语句的任何变异表，其中包括触发表本身。
- 读或修改触发表的约束表中的主关键字，唯一关键字和外部关键字列。除此之外的其他列可以修改。

上述限制适用于所有的行级触发器，这些限制只在语句触发器作为 DELETE CASCADE操作的结果激活时适用于语句触发器。

注意 如果INSERT语句只影响一行的话，则在该行的之前和之后触发器将不把触发表作为变异表对待，这是在行级触发器可能载入或修改触发表时的唯一案例。下面一类的语句

```
INSERT INTO table SELECT ...
```

总是把触发表作为变异表对待，即使其子查询仅返回一行也是如此。

作为例子，请考虑下面的触发器 CascadeRSInserts。尽管该触发器修改表 students和classes，但由于修改的表students和classes中的列不是关键字列，所以修改是合法的。下面，我们将分析一个非法的触发器案例。

```
节选自在线代码CascadeRSInserts .sql
CREATE OR REPLACE TRIGGER CascadeRSInserts
/* Keep the registered_students, students, and classes
tables in synch when an INSERT is done to registered_students. */
BEFORE INSERT ON registered_students
FOR EACH ROW
DECLARE
  v_Credits classes.num_credits%TYPE;
BEGIN
  -- Determine the number of credits for this class.
  SELECT num_credits
    INTO v_Credits
   FROM classes
  WHERE department = :new.department
    AND course = :new.course;

  -- Modify the current credits for this student.
  UPDATE students
    SET current_credits = current_credits + v_Credits
   WHERE ID = :new.student_id;

  -- Add one to the number of students in the class.
  UPDATE classes
    SET current_students = current_students + 1
   WHERE department = :new.department
    AND course = :new.course;
```

下载

```
END CascadeRSInserts;
```

6.3.1 变异表案例介绍

假设我们要把每个专业的学生名额限制在五个。我们可以通过使用表 students上的之前插入或更新行级触发器来实现这种限制，下面是该触发器的代码：

```
节选自在线代码LimitMajors .sql
CREATE OR REPLACE TRIGGER LimitMajors
/* Limits the number of students in each major to 5.
If this limit is exceeded, an error is raised through
raise_application_error. */
BEFORE INSERT OR UPDATE OF major ON students
FOR EACH ROW
DECLARE
    v_MaxStudents CONSTANT NUMBER := 5;
    v_CurrentStudents NUMBER;
BEGIN
    -- Determine the current number of students in this
    -- major.
    SELECT COUNT(*)
        INTO v_CurrentStudents
        FROM students
        WHERE major = :new.major;

    -- If there isn't room, raise an error.
    IF v_CurrentStudents + 1 > v_MaxStudents THEN
        RAISE_APPLICATION_ERROR(-20000,
            'Too many students in major ' || :new.major);
    END IF;
END LimitMajors;
```

初看，该触发器好象可以实现需要的功能。然而，如果我们更新表 students并激活该触发器，我们会得到下面的输出：

```
节选自在线代码LimitMajors .sql
SQL> UPDATE students
2      SET major = 'History'
3      WHERE ID = 10003;
UPDATE students
*
ERROR at line 1:
ORA-04091: table EXAMPLE.STUDENTS is mutating, trigger/function
      may not see it
ORA-06512: at line 7
ORA-04088: error during execution of trigger 'EXAMPLE.LIMITMAJORS'
```

由于触发器LimitMajor查询其自己的触发表（该表是变异表），所以导致了错误ORA-4091。另外，错误ORA-4091是在该触发器激活时引发的，而不是在创建时引发的。

6.3.2 变异表错误的处理

表students由于使用了行级触发器而成了变异表，这就使我们不能在该行级触发器中对该表进行查询，而只能在语句级触发器使用查询语句。然而，我们还不能只是简单地把触发器LimitMajor修改为语句级触发器，这是因为我们需要在触发器体中使用 :new.major的值。解决该问题的方法是创建两个触发器，即一个行级触发器和一个语句级触发器。在行级触发器中，我们记录: new.major的值，但我们不查询表 students；而查询任务由语句级触发器来实现并使用行触发器记录的值。

至于如何记录 :new.major的值。一种方法是在包的内部使用 PL/SQL表来记录。用这种方法，我们可以在每次更新时保存多个值。同样，每个会话也得到其自己打包变量的实例，因此，我们就不必担心由于不同会话进行同时更新操作而引起的麻烦。实现上述方案的包 student_data,以及改进的触发器RLimitMajors和SLimiMajors的程序如下：

```
节选自在线代码mutating .sql
CREATE OR REPLACE PACKAGE StudentData AS
    TYPE t_Majors IS TABLE OF students.major%TYPE
        INDEX BY BINARY_INTEGER;
    TYPE t_IDs IS TABLE OF students.ID%TYPE
        INDEX BY BINARY_INTEGER;

    v_StudentMajors t_Majors;
    v_StudentIDs t_IDs;
    v_NumEntries BINARY_INTEGER := 0;
END StudentData;

CREATE OR REPLACE TRIGGER RLimitMajors
    BEFORE INSERT OR UPDATE OF major ON students
    FOR EACH ROW
BEGIN
    /* Record the new data in StudentData. We don't make any
       changes to students, to avoid the ORA-4091 error. */
    StudentData.v_NumEntries := StudentData.v_NumEntries + 1;
    StudentData.v_StudentMajors(StudentData.v_NumEntries) :=
        :new.major;
    StudentData.v_StudentIDs(StudentData.v_NumEntries) := :new.id;
END RLimitMajors;

CREATE OR REPLACE TRIGGER SLimitMajors
    AFTER INSERT OR UPDATE OF major ON students
DECLARE
    v_MaxStudents CONSTANT NUMBER := 5;
    v_CurrentStudents NUMBER;
    v_StudentID students.ID%TYPE;
    v_Major students.major%TYPE;
BEGIN
    /* Loop through each student inserted or updated, and verify
```

```
that we are still within the limit. */
FOR v_LoopIndex IN 1..StudentData.v_NumEntries LOOP
    v_StudentID := StudentData.v_StudentIDs(v_LoopIndex);
    v_Major := StudentData.v_StudentMajors(v_LoopIndex);

    -- Determine the current number of students in this major.
    SELECT COUNT(*)
        INTO v_CurrentStudents
        FROM students
        WHERE major = v_Major;

    -- If there isn't room, raise an error.
    IF v_CurrentStudents > v_MaxStudents THEN
        RAISE_APPLICATION_ERROR(-20000,
            'Too many students for major ' || v_Major ||
            ' because of student ' || v_StudentID);
    END IF;
END LOOP;

-- Reset the counter so the next execution will use new data.
StudentData.v_NumEntries := 0;
END LimitMajors;
```

注意 请注意要在运行前面的脚本之前，一定要撤消不正确的 LimitMajors 触发器。

现在，我们可以通过更新表 student 来测试这一系列触发器直到我们的程序中出现了过多的历史专业为止

```
节选自在线代码 mutating .sql
SQL> UPDATE students
  2      SET major = 'History'
  3      WHERE ID = 10003;
1 row updated.

SQL> UPDATE students
  2      SET major = 'History'
  3      WHERE ID = 10002;
1 row updated.

SQL> UPDATE students
  2      SET major = 'History'
  3      WHERE ID = 10009;
UPDATE students
*
ERROR at line 1:
ORA-20000: Too many students for major History because of student 10009
ORA-06512: at "EXAMPLE.SLIMITMAJORS", line 19
ORA-04088: error during execution of trigger 'EXAMPLE.SLIMITMAJORS'
```

上面的结果就是我们期望的功能。当行级触发器读入或修改变异表（ mutating table ）时，

上述技术可以引发错误信息 ORA-4091。为了避免在行级触发器中进行非法处理，我们把该处理推迟到一个之后的语句级触发器实现，这样以来该处理就是合法的了。打包的 PL/SQL表是用来存储被修改的行的。

下面是应用该技术时要注意的几个问题：

- 由于PL/SQL表是位于包中，所以这些表对于行级触发器和语句级触发器都是可见的，确保变量的全局性的唯一方法是把要定义的全局变量放在包中。
- 我们在该程序中使用了计数器变量 StudentsData.v_NumEntries，当其所在的包首次创建时，该变量的初始值为 0。然后，该变量的值由行级触发器修改。语句级触发器对该变量进行引用并在处理结束后将该变量设置为 0。上述的步骤是必须的，因为只有这样该会话发布的UPDATE语句将实现正确的结果。
- 对触发器 SLimitMajors 进行最多学生数量的检查将稍有变化。由于该触发器是语句之后触发器，所以变量 v_CurrentStudents 将在执行插入或更新命令后保存某专业的学生数量。因此，在触发器 LimitMajor 中对变量 v_CurrentStudents+1 的检查也被变量 v_CurrentStudents 替代。
- 我们也可以使用数据库表来代替 PL/SQL 表。我个人不喜欢这种技术，这是由于发布 UPDATE 的多个同时会话之间可能有相互用（在 Oracle8i 中，我们可以使用临时表）。打包的PL/SQL表是众多会话之间唯一的，因此解决了上面的问题。

6.4 小结

正如我们在本章所看到的，触发器是继 PL / SQL 与 Oracle 数据库之后的重要工具。我们可以使用该工具来加强比正常的引用完整性约束条件更复杂的数据约束。Oracle8i 把触发器扩展到了除了在表或视图上进行 DML 操作以外的事件。本章介绍的触发器结束了我们在前三章中对命名PL/SQL块的讨论。我们将在第 13 章中看到的另一种命名 PL/SQL 块是对象类型体。下一章将讨论PL/SQL中的内置包问题。

第7章 数据库作业和文件输入输出

本章将讨论 PL/SQL内置包 DBMS_JOB和UTL_FILE的特性。PL/SQL2.2以上版本提供的包 DBMS_JOB支持存储过程在系统的管理下周期性自动运行而无须用户的介入。而 PL/SQL2.3以上版本提供的包 UTL_FILE则扩充了读写系统文件的功能。这两个包提供的功能使 PL/SQL具备了与其他第三代程序设计语言相同的处理能力。

7.1 数据库作业

**PL/SQL 2.2 及
更高版本** 在PL/SQL2.2及更高版本下，我们可以指定 PL/SQL程序在指定时间运行，支持这种定时运行功能的是包 DBMS_JOB提供的作业序列。Oracle中作业的运行是通过将该作业和说明该作业运行方式的参数共同提交给作业序列实现的。有关当前运行的作业，上一个提交的作业的成功或失败状态等信息可以在数据字典中找到（请参阅 7.1.4节）。

**Oracle 8 及
更高版本** 读者需要注意的是，Oracle8及更高版本支持的高级查询功能提供了比包 DBMS_JOB功能更强大的PL/SQL查询功能。有关该功能的细节，请参考 Oracle文档资料。

7.1.1 后台进程

一个Oracle实例是由运行在系统上的各种进程所组成，不同的进程负责实现不同的数据库操作，如把数据库记录读入内存，把数据库记录写回磁盘，以及把数据归档到脱机存储器中。除了管理数据库的处理外，数据库系统还具有叫做 SNP的进程。这些后台进程除了要处理快照（Snapshot）的自动刷新外，还要通过DBMS_JOB来管理访问作业队列的访问通道。

象其他的数据库进程运行方式一样，SNP进程也运行在后台。然而，与数据库进程不同的是，如果SNP进程失败的话，Oracle就将重新启动该进程并不影响数据库的其他进程。如果其他的数据库进程失败的话，这些失败的进程将会使数据库停止运行。SNP周期性地激活来检查作业序列。如果论到某个作业运行，则 SNP进程就在启动该进程运行后再进入睡眠状态。一个给定的进程只能一次运行一个作业。在 Oracle7系统下，系统限定的最大 SNP进程的数目是 10个（从SNP0-SNP9），因此，系统中可同时运行的最大作业的数目也是 10个。在Oracle8系统中，SNP进程最大数目增加到了36个，其进程号是SNP0-SNP9,SNPA-SNPZ。

数据库初始化文件（init.ora）中有两个参数用来控制 SNP进程的属性。这两个参数是 JOB_QUEUE_PROCESSES和JOB_QUEUE_INTERVAL，下面的表7-1是该参数的说明。值得注意的是，如果将JOB_QUEUE_PROCESSES设置为0的话，系统将禁止作业运行。由于每个进程都将在查询新的作业之前按 JOB_QUEUE_INTERVAL指定的时间（以秒为单位）进行睡眠，所以参数JOB_QUEUE_INTERVAL就指定了运行两个作业之间的最短时间间隔。上述两个参数都不能使用 ALTER SYSTEM或ALTER SESSION进行动态修改，因此用户必须先对数据库初始化

文件进行修改并重启数据库才能使修改的参数生效。

注意 Oracle7.3以及更高版本中已经不在使用 Oracle7.2版使用的初始化参数 JOB_QUEUE_KEEP_CONNECTIONS。从Oracle7.3版开始数据库的物理连接完全由系统自动实施控制。

表7-1 作业初始化参数

参 数	默 认 值	范 围	说 明
JOB_QUEUE_PROCESSES	0	0 ~ 10(在Oracle8中为0 ~ 36)	可同时运行的进程数目
JOB_QUEUE_INTERVAL	60	1 ~ 3600s	两个进程间唤醒的间隔。进程按指定的时间进行睡眠

7.1.2 运行作业

运行作业的方法有两种，一种是将作业提交给作业队列，另一种是强制作业立即运行。当作业被提交给作业队列时，SNP就在该作业的启动时刻运行该作业。如果指定了作业的运行间隔的话，该作业将自动地周期运行。如果作业是立即启动的，该作业就运行一次。

1. 提交：SUBMIT

使用下面的SUBMIT过程可以将作业提交到作业队列中。该过程的语法是：

节选自在线代码TempInsert.sql

```
PROCEDURE SUBMIT( job OUT BINARY_INTEGER,
                   what IN VARCHAR2,
                   next_date IN DATE DEFAULT SYSDATE,
                   interval IN VARCHAR2 DEFAULT NULL,
                   no_parse IN BOOLEAN DEFAULT FALSE);
```

下面说明了SUBMIT语句中使用的参数：

参 数	类 型	说 明
job	BINARY_INTEGER	作业号。创建作业时，作业将被赋予一个作业号。只要该作业存在，其作业号将保持不变。作业号在实例的范围内是唯一的
what	VARCHAR2	组成作业的PL/SQL代码。通常，该代码是存储过程的调用
next_date	DATE	作业下一次运行的日期
interval	VARCHAR2	计算作业再次运行时间的函数。该函数的值必须是一个时间值或为空 NULL
no_parse	BOOLEAN	如果该参数为真的话，作业将在其第一次运行时才进行语法分析。如果该参数为假值的话（默认值），则作业在提交时就对其进行语法分析。如果作业引用的数据库对象不存在但又必须提交该作业时，可以将该参数设置为真。（被引用的对象在运行时必须存在。）

Oracle 8i 及
更高版本

Oracle8i允许作业运行在 Oracle Parallel Server(OPS)环境下的指定实例中。DBMS_JOB.SUBMIT提供的两个参数——instance和force以实现上述功能。这两个参数的意义7.1.3节的“实例仿射性”中讨论。

例如，假设我们用下面的程序来创建过程TempInsert:

节选自在线代码TempInsert.sql

```
CREATE SEQUENCE temp_seq
    START WITH 1
    INCREMENT BY 1;

CREATE OR REPLACE PROCEDURE TempInsert AS
BEGIN
    INSERT INTO temp_table (num_col, char_col)
        VALUES (temp_seq.nextval,
                TO_CHAR(SYSDATE, 'DD-MON-YYYY HH24:MI:SS')); COMMIT;
END TempInsert;
```

我们可以使用下面的SQL *Plus脚本指定过程TempInsert每90秒钟运行一次：

节选自在线代码TempInsert.sql

```
SQL> VARIABLE v_JobNum NUMBER
SQL> BEGIN
 2     DBMS_JOB.SUBMIT(:v_JobNum, 'TempInsert;', SYSDATE,
 3                         'sysdate + (90/(24*60*60))');
 4     COMMIT;
 5 END;
 6 /
PL/SQL procedure successfully completed.
```

```
SQL> print v_JobNum
V_JOBNUM
-----
2
```

注意 要启动作业运行必须提交包括调用DBMS_JOB.SUBMIT语句的事务。其原因是SUBMIT语句通过在数据字典表中插入一行来记录作业信息，SNP进程将查询该表以决定是否有作业需要运行。在这种情况下，INSERT和SELECT语句是由不同的会话实现的，所以必须提交事务。

如果把初始化参数JOB_QUEUE_PROCESSES设置为本0，我们仍然可以发布SUBMIT命令而不会出错。但在这种情况下，作业无法运行。这样一来，为了观察过程TempInsert对表temp_table的插入操作，我们至少要把JOB_QUEUE_PROCESSES的值设置为1。如果JOB_QUEUE_INTERVAL是其默认值(60s)，则作业可能不会在90s后开始运行。在运行该作业所需时间之后的90s内，该作业将被标记为就绪运行状态。然而，该作业的真正运行时间是在SNP进程唤醒后才可能开始，(唤醒SNP需要60s时间)。因此，在该作业再次运行前，有可能要等待长达150s。例如，下面是作业运行三次后表Temp_tbale的输出样本：

```
SQL> SELECT * FROM temp_table;
  NUM_COL CHAR_COL
-----
 1 25-APR-1999 18:18:59
 2 25-APR-1999 18:21:02
 3 25-APR-1999 18:23:05
```

在上例中，在作业第一次和第二次运行期间，以及第二次和第三次运行之间的间隔都是123s。

注意 上面SUBMIT调用将启动过程TempInsert周期性地运行直到数据库关闭，或该作业被REMOVE调用删除为止。7.1.3节中的“**删除作业**”将介绍 REMOVE语句。

作业号 当一个作业初次被提交时，该作业就被赋予一个作业号，该作业号是由序列SYS.JOBEQ生成的。一旦给作业赋予了作业号，该作业号将在作业被删除或再次提交前保持不变。

警告 可以象处理数据库对象那样对作业进行导入或导出操作，这种操作不会变更作业的作业号。如果企图导入一个其作业号已经存在的作业的话，系统将给出错误信息，并将禁止将该作业导入。在这种情况下，只须再次提交该作业，就可以生成新的作业号。我们也可以使用过程USER_EXPORT来导出作业。

作业定义 参数what指定了作业的代码。通常，作业由存储过程组成，因此参数 what应是调用存储过程的字符串。我们将在本节的后面介绍该字符串的完整格式。参数 what调用的过程可以带有任何数量的参数。但所有参数都必须是 IN类型参数，这是由于该过程没有实参来接收OUT或IN OUT形参的值。该规则的唯一例外是特殊标识符next_date和broken(将在后面介绍)。

警告 一旦提交了作业，该作业将由后台的SNP进程控制运行。为了了解运行结果，一定要在作业过程末尾写上提交 COMMIT代码。如果作业不发布 COMMIT命令，当运行该作业的会话结束时，则事务将自动地重新开始。

如表7-2所示，在作业定义中有三个可以合法使用的标识符。其中，参数 job是IN类型的参数，因此作业只能读该参数的值。参数 next_date和broken都是IN OUT参数，所以，该作业可以对它们进行修改。

表7-2 作业控制标识符

标示符	类 型	说 明
job	BINARY_INTEGER	计算当前作业的作业号
next_date	DATE	计算作业下一次运行的日期。如果作业将该参数设置为空的话，则该作业将被从队列中删除
broken	BOOLEAN	计算作业的状态，如果作业被中断，则为真值，否则为假值。如果作业自身将该参数设置为真的話，该作业将被标记为执行中断，但不会从队列中删除

假设我们对TempInsert修改如下：

```
节选自在线代码TempInsert.sql
CREATE OR REPLACE PROCEDURE TempInsert
  (p_NextDate IN OUT DATE) AS
    v_SeqNum    NUMBER;
    v_StartNum NUMBER;
    v_SQLErr   VARCHAR2(60);
BEGIN
```

```
SELECT temp_seq.NEXTVAL
  INTO v_SeqNum
  FROM dual;

-- See if this is the first time we're called.
BEGIN
  SELECT num_col
    INTO v_StartNum
    FROM temp_table
   WHERE char_col = 'TempInsert Start';

-- We've been called before, so insert a new value.
INSERT INTO temp_table (num_col, char_col)
  VALUES (v_SeqNum,
          TO_CHAR(SYSDATE, 'DD-MON-YYYY HH24:MI:SS')));

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    -- First time we're called. First clear out the table.
    DELETE FROM temp_table;

    -- And now insert.
    INSERT INTO temp_table (num_col, char_col)
      VALUES (v_SeqNum, 'TempInsert Start');
END;

-- If we've been called more than 5 times, exit.
IF v_SeqNum - V_StartNum > 5 THEN
  p_NextDate := NULL;
  INSERT INTO temp_table (num_col, char_col)
    VALUES (v_SeqNum, 'TempInsert End');

  END IF;

  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    -- Record the error in temp_table.
    v_SQLErr := SUBSTR(SQLERRM, 1, 60);
    INSERT INTO temp_table (num_col, char_col)
      VALUES (temp_seq.NEXTVAL, v_SQLErr);
    -- Exit the job.
    p_NextDate := NULL;
    COMMIT;
END TempInsert;
```

提交如下：

节选自在线代码TempInsert1.sql
BEGIN

```

DBMS_JOB.SUBMIT(:v_JobNum, 'TempInsert(next_date);', SYSDATE,
                  'SYSDATE + (60/(24*60*60))');

COMMIT;

END;

```

现在，该作业将每 60秒运行一次，并在被调用五次之后自动地将自己从作业队列中删除（通过把P_NextDate改置为Null）。该作业的输出样本如下：

```

SQL> SELECT * FROM temp_table ORDER BY num_col;
NUM_COL CHAR_COL
-----
1 TempInsert Start
2 25-APR-1999 18:45:37
3 25-APR-1999 18:46:38
4 25-APR-1999 18:47:40
5 25-APR-1999 18:48:41
6 25-APR-1999 18:49:43
7 25-APR-1999 18:50:44
7 TempInsert End

```

注意 在运行该上面的例子之前，一定要使用 DBMS_JOB.REMOVE来删除前面调用的TempInsert作业。

通过参数next_date(与参数broken一起或单独使用)的返回值，作业可以把自己从作业队列中删除。

参数what是一个VARCHAR2类型的字符串。这样做的结果使得用于调用作业过程中使用的任何字符都必须用两个单引号括起来，而过程调用也必须以分号结束。例如，如果我们使用下面的声明创建一个过程：

```
CREATE OR REPLACE PROCEDURE Test(p_InParameter In VARCHAR2);
```

这时，我们可以使用下面的what字符串来提交该过程：

```
'Test("This is a character string");'
```

运行间隔 参数next_date指定了作业在提交后的运行时间，该参数的值是在该作业第一次运行时计算的。就在该作业运行之前，对由参数interval给定的函数进行求值。如果该作业执行成功，则由interval返回的结果就变成了新的next_date参数（假设该作业没有说明next_date参数）。如果作业执行成功并且对interval的求值结果为空，则该业就被从作业队列中删除。参数interval给出的表达式是按字符串传递的，但其结果应是日期值。下面是对某些常用的表达式和它们的作用的说明：

间 隔 值	执 行 结 果
'SYSDATE+7'	从上次执行后的整一个星期。如果作业在星期二初次提交，则下一次运行将在下一个星期二。如果第二次运行失败的话，而在星期三获得成功，则后续的运行将在星期三开始
'NEXT_DAY(TRUNC(SYSDATE),"FRIDAY"+12/24'	每星期五中午运行。请在字符串内使用两个单引号把FRIDAY括起来
SYSIZE+1/24	每小时

2. 运行命令RUN

过程DBMS_JOB.RUN可以立即启动作业运行。其语法如下：

```
RUN ( job IN BINARY_INTEGER );
```

其中，作业必须是通过调用SUBMIT已创建的作业。该命令不管指定作业的当前状况如何，立即在当前进程中运行该作业。需要提醒的是，SNP后台进程与该作业无关。

警告 DBMS_JOB.RUN将会初始化当前会话包，这样做的原因是为了给作业提供一个连续的环境来运行，就象SNP进程对作业的处理一样。

7.1.3 其他的DBMS_JOB子程序

下面，我们来讨论过程DBMS_JOB中其他子程序的功能。表7-3对这些子程序做了说明。在DBMS_JOB的包头中有一个额外的程序——ISUBMIT。该程序是由其他过程使用指定的作业号进入作业的接口程序。

表7-3 DBMS_JOB子程序说明

子程序名	功能说明
SUBMIT	把一个新的作业提交给队列，由SNP进程控制运行
ISUBMIT	由其他过程使用指定的作业号进入作业的接口程序。程序员不能在其程序代码中直接使用该程序，只能使用SUBMIT
RUN	强迫指定的作业运行在当前进程中
REMOVE	从作业队列中删除一个作业
BROKEN	标记作业中断或没有中断
CHANGE	变更作业中任何可配置的字段
WHAT	变更what字段
NEXT_DATE	变更next_date字段
INTERVAL	变更interval字段
INSTANCE	在Oracle8i及更高版本下，变更instance字段
USER_EXPORT	返回需要重新创建作业调用的文本
CHECK_PRIVS	确认给定的作业号可访问

1. 删除作业

我们在本章前几节中已经看到，通过把参数next_date设置为空，作业可以把自己从作业队列中删除。我们也可以使用下面的过程显式地把作业从队列中删除：

```
REMOVE ( job IN BINARY_INTEGER );
```

上面唯一的参数是作业号。如果该作业的next_date的值为空的话（该作业所设置的值或参数间隔的值为空），则该作业在其结束运行后将被删除。如果调用删除过程时，该作业正在运行期间，则该作业将在其运行结束后从队列中删除。

2. 中断作业

对于运行失败的作业，Oracle系统将自动重新运行该作业。该作业将在其第一次运行失败一分钟后再次运行。如果上述努力再次失败，下一次运行将在两分钟后开始。每次运行的间隔将逐次加倍，从四分钟，到八分钟，一直持续下去。如果重试间隔超过了该作业的执行间隔，

则使用执行间隔。一旦该作业失败了 16 次，它就被标识为中断的作业。中断的作业将不能再次运行。

我们可以使用 RUN 命令来运行中断的作业。如果调用成功的话，则该作业的失败记数器将重置为 0，并且将该作业的中断标志取消。过程 BROKEN 也可以用来修改作业的状态。该过程的语法如下：

```
BROKEN( job IN BINARY_INTEGER,
        broken IN BOOLEAN,
        next_date IN DATE DEFAULT SYSDATE);
```

该过程参数的说明如下：

参 数	类 型	说 明
job	BIANRY_INTEGER	作业的状态将要发生变更的作业号
broken	BOOLEAN	作业的新状态。如果该值为真，则该作业就被标识为中断的作业。如果为假值，则该作业没有被中断并将在由 next_date 指定的时间到来时再次运行。
next_date	DATE	作业下次运行的日期。其默认值是 SYSDATE

3. 变更作业

在作业被提交后，该作业的参数也可以变更。实现这种功能的过程的语法如下：

```
PROCEDURE CHANGE( job IN BINARY_INTEGER,
                  what IN VARCHAR2 DEFAULT NULL,
                  next_date IN DATE DEFAULT NULL,
                  interval IN VARCHAR2 DEFAULT NULL);

PROCEDURE WHAT( job IN BINARY_INTEGER,
                what IN VARCHAR2);

PROCEDURE NEXT_DATE( job IN BINARY_INTEGER,
                     next_date IN DATE);

PROCEDURE INTERVAL( job IN BINARY_INTEGER,
                   interval IN VARCHAR2);
```

过程 CHANGE 用来一次变更一个以上作业的属性，过程 WHAT、NEXT_DATE、INTERVAL 则用来改变与它们相关参数标识的属性。

以上所有参数的作用都与过程 SUBMIT 中参数的功能一样。如果我们使用 CHANGE 或 WHAT 来修改 what 参数，则当前环境就将变为适应该作业的新的执行环境。有关作业环境的进一步介绍，请参阅 7.1.5 节。

4. 实例仿射性

Oracle 8i 及
更高版本

当我们使用 OPS (Oracle Parallel Server) 时，我们可以指定一个实例来运行给定的作业。这就叫做实例仿射性 (Instance Affinity)。实例可以由过程 INSTANCE 指定，其语法是：

```
PROCEDURE INSTANCE( job IN BINARY_INTEGER,
                    instance IN BINARY_INTEGER,
                    force IN BOOLEAN DEFAULT FALSE);
```

其中，instance是运行作业的实例号。如果实例是DBMS_JOB.ANY_INSTANCE(0)，则作业的仿射性将变更并且任何可用的实例都可以运行该作业，此时与参数force的值无关。如果instance是正值并且参数force为假值，则作业仿射性只有在指定的实例处于运行状态才被变更。如果指定的实例没有运行，或该实例不合法，这时Oracle8i将返回错误：“ORA-23428：与实例号字符串相关联的作业非法。”

其他DBMS_JOB子程序也支持实例仿射性。例如，如下所示的过程SUBMIT就带有参数instance和force：

```
PROCEDURE SUBMIT( job OUT BINARY_INTEGER,
                  what IN VARCHAR2,
                  next_date IN DATE DEFAULT SYSDATE,
                  interval IN VARCHAR2 DEFAULT NULL,
                  no_parse IN BOOLEAN DEFAULT FALSE,
                  instance IN BINARY_INTEGER DEFAULT ANY_INSTANCE,
                  force IN BOOLEAN DEFAULT FALSE);
```

参数instance和force的含义和使用方法都与过程DBMS_JOB.INSTANCE的同类参数一样。

过程CHANGE在Oracle8i下得到了增强：

```
PROCEDURE CHANGE( job IN BINARY_INTEGER,
                  what IN VARCHAR2 DEFAULT NULL,
                  next_date IN DATE DEFAULT NULL,
                  interval IN VARCHAR2 DEFAULT NULL,
                  instance IN BINARY_INTEGER DEFAULT NULL,
                  force IN BOOLEAN DEFAULT FALSE);
```

其中参数instance和force与上面的参数一样。

最后，DBMS_JOB.RUN也带有force参数：

```
PROCEDURE RUN( job IN BINARY_INTEGER,
               force IN BOOLEAN DEFAULT FALSE);
```

如果参数force为真，则该作业仅可以在从指定实例内部调用DBMS_JOB.RUN的情况下运行。

5. 导出作业

过程USER_EXPORT返回重新创建给定作业所需的文本：

```
PROCEDURE USER_EXPORT( job in BINARY_INTEGER,
                      mycall IN OUT VARCHAR2);
```

例如，如果我们通过TempInsert的第二版提交的作业调用过程USER_EXPORT的话，该过程将返回下面的文本：

节选自在线代码jobExport.sql

```
SQL> DECLARE
  2    v_JobText VARCHAR2(2000);
  3  BEGIN
  4    DBMS_JOB.USER_EXPORT(:v_JobNum, v_JobText);
  5    DBMS_OUTPUT.PUT_LINE(v_JobText);
  6  END;
```

7 /

```
dbms_job.isubmit(job=>10,what=>'TempInsert(next_date)',next
_date=>to_date('1999-03-29:00:07:37','YYYY-MM-DD:HH24:MI:SS'
),interval=>'sysdate + (5/(24*60*60))',no_parse=>TRUE);
```

该作业必须在当前作业队列中，否则过程 USER_EXPORT将返回错误：“ORA-23421：作业号job_num不在作业队列中”。

提示 USER_EXPORT返回对DBMS_JOB.ISUBMIT的调用，而不是DBMS_JOB.SUBMIT的调用。如果要重新创建具有相同作业号的作业的话，我们可以使用ISUBMIT。然而，这种方法不太常用。我们推荐使用过程SUBMIT。该命令将重新提交作业并生成一个新的作业号。USER_EXPORT是由导出工具程序自动调用并生成后面重新导入时所使用的代码。

6. 检查作业的权限

过程CHECK_PRIVS有两个功能：其一是校验给定的作业号是否存在，其二是锁定数据字典中的相应的行以便程序对其进行修改。该过程的语法定义如下：

```
PROCEDURE CHECK_PRIVS(job IN BINARY_INTEGER);
```

其中，job是作业号，如果该作业号不存在，或该作业号没有被当前用户提交，则Oracle将提示错误信息：“ORA-23421：作业号job_num不在作业队列中”。

7.1.4 在数据库视图中观察作业

Oracle有几个数据字典视图专门用来记录作业信息。其中视图dba_jobs和user_jobs返回作业的what、next_date、interval等有关信息。除此之外，这些视图还提供运行环境的信息。视图dba_jobs_running描述了当前运行的作业。有关这些视图的介绍，请看本书附录C的内容。

7.1.5 作业运行环境

当我们把作业提交给队列时，当前的环境就被记录下来。记录信息中包括如NLS_DATE_FORMAT之类的NLS参数的设置。这些在作业创建时记录下来的信息将在该作业运行时使用。如果我们使用CHANGE或WHAT过程修改what的属性时，上述设置将随之改变。

注意 作业可以通过发布DBMS_SQL包的ALTER SESSION命令（或Oracle8i中的本地动态SQL命令）来修改其运行环境。如果运行环境发生了变更，它将仅影响当前作业的运行，而不会对将来的运行有影响。包DBMS_SQL和本地动态SQL的内容在本书的第8章中介绍。

作业将运行在其自己提交器所属的权限组之下，不允许任何角色运行。如果我们需要在作业中允许角色执行，我们可以使用动态SQL来发布SET ROLE命令。当然，如果该作业是由存储过程组成的话，则所有的角色都将被禁止。

作业仅可以由提交器（Submitter）控制运行。作业的执行权限与作业没有任何关联；包自身的EXECUTE权限是唯一必要的数据库权限。

7.2 文件输入输出

如上所述，PL/SQL语言并没有把输入、输出功能配置在该语言内部，PL/SQL下的输入输出是通过所提供的包所支持的功能实现的。PL/SQL输出到屏幕的功能是通过本书第3章介绍的包DBMS_OUTPUT所提供的功能实现的。PL/SQL2.3版通过包UTL_FILE的功能把文件I/O扩充到了文本文件。在该版本下无法借助其UTL_FILE包来把输出直接转换为二进制文件。

Oracle8允许通过使用类型BFILE来读入二进制文件，这里BFILE是特殊形式的外部LOB。本书的第15，16将专门讨论BFILE类型以及LOB的其他类型。然而，即使使用Oracle8i，包UTL_FILE也不能用来处理二进制文件。

本节将描述UTL_FILE的工作原理。本章结尾处将提供三个完整的案例来介绍该包的使用方法。

7.2.1 安全

客户端PL/SQL有一个类似于UTL_FILE的包叫做TEXT_IO的包。值得注意的是，客户端要处理的安全问题要比服务器端要多，使用客户端包TEXT_IO首次创建的文件在其操作系统权限的限制下，可以放置在客户端的任何地方。PL/SQL或数据库本身没有与其关联的用于客户端I/O操作的权限。我们将在本节讨论UTL_FILE的安全实现。

1. 数据库安全

数据库服务器需要更可靠的安全机制来保证系统安全运行。Oracle数据库通过限制包UTL_FILE可以访问目录的范围来实现安全运行。系统允许访问的目录由数据库初始化文件中的参数UTL_FILE_DIR说明。每个可以访问的目录都在初始化文件中用下面的参数行指定：

```
UTL_FILE_DIR = directory_name
```

参数directory_name所指定的目录在很大程度上与操作系统有关。如果操作系统对大小写敏感，则directory_name也区分大小写字母。例如，下面的目录入口在Unix系统下是合法的（假设所指定的目录是存在的）：

```
UTL_FILE_DIR=/tmp  
UTL_FILE_DIR=/home/oracle/output_files
```

为了能够使用UTL_FILE来访问文件，目录名和文件名都要作为单独的参数传递给函数FOPEN。该函数将接收的目录名与可访问的文件清单进行比较。如果清单中有该文件，则允许对给文件进行操作，其具体操作方式还与下一节将介绍的操作系统的安全约束有关。如果函数FOPEN所接收的文件不可访问，该函数就返回错误。除此之外，可访问目录的子目录一般也是不可访问的，除非该子目录也显式地出现在可访问目录表中。假设上面的目录是可访问的，表7-4对合法与非法的目录/文件名给予了说明。

注意 即使操作系统对大小写不敏感，在指定目录与可访问目录之间的比较也是大小写

敏感的。

如果初始化文件中有下面一行：

```
UTL_FILE_DIR = *
```

则数据库访问权限将被禁止。该命令将使所有的目录都可由 UTL_FILE访问。

表7-4 合法与非法文件的说明

目 录 名	文 件 名	说 明
/tmp	myfile.out	合法
/home/oracle/output_files	studengts.list	合法
/tmp/1995	january.results	非法，子目录/tmp/1995不能访问
/home/oralce	output_files/classes.list	非法，子目录名不能作为文件名的一部分
/TMP	myfile.out	非法，大写字母不对

警告 执行关闭数据库访问权限的操作必须十分小心。Oracle并不推荐用户在产生式系统中使用该选择项，其原因是该选择项的使用可能会限制操作系统的访问权限。除此之外，也不要使用‘.’来代表要访问目录的当前目录部分（ Unix使用该符号表示当前目录），目录的指定一定要使用目录的全名。

2. 操作系统的安全性

由UTL_FILE执行的文件操作是作为 Oracle用户实现的。（ Oracle用户是用于运行数据库所需文件的拥有者，同时它也是组成数据库实例的进程的拥有者。）这样一来，Oracle用户就必须具有操作系统读写所有可访问文件的权限。如果 Oracle用户没有访问权限，则任何对该目录的访问将被操作系统禁止。

由UTL_FILE创建的任何文件将归属于 Oracle用户所有，这些文件在创建时已具有操作系统为 Oracle用户配置的权限。如果有其他用户要在 UTL_FILE之外访问这些文件的话，就需要操作系统变更这些文件的访问权限。

警告 禁止对所有可访问目录的写操作也是一种安全措施。在这种情况下，写操作的权限只赋予 Oracle用户。

7.2.2 UTL_FILE引发的异常

如果UTL_FILE中的过程或函数遇到了错误，它们将引发异常。这些可能发生的异常在表 7-5中给予了说明，这些异常中有八个在 UTL_FILE中有定义，其余的两个是予定义的异常（NO_DATA_FOUND和VALUE_ERROR）。UTL_FILE异常可以按名字或由异常处理程序 OTHERS来捕捉处理。予定义的异常还可以由 SQLCODE的值（1403或6502）标识。

7.2.3 打开和关闭文件

UTL_FILE中的所有操作都使用文件句柄实现。所谓文件句柄就是 PL/SQL中用来标识文件的值，它类似于DBMS_SQL中使用的游标ID。所有的文件句柄都是 UTL_FILE.FILE_TYPE类型。

文件句柄由 FOPEN 返回并作为 IN 类型参数传递给 UTL_FILE 中其他子程序使用。

1. FOPEN

FOPEN 可以打开用于输入或输出的文件。在任何时候，给定的文件每次只能打开用于输入或用于输出。打开的文件不能同时用于输入和输出操作。FOPEN 的语法如下：

```
FUNCTION FOPEN( location IN VARCHAR2,
                filename IN VARCHAR2,
                open_mode IN VARCHAR2)

RETURN FILE_TYPE;
```

要打开文件的目录路径必须是已经存在的，否则 FOPEN 不会创建新的目录。如果打开方式为‘w’，则 FOPEN 将覆盖现有的文件。FOPEN 的返回值和参数的说明如下所示：

参 数	类 型	说 明
location	VARCHAR2	文件位于的目录路径。如果该目录与可访问目录表中的目录不匹配，则将引发异常 UTL_FILE.INVALID_PATH。
filename	VARCHAR2	要打开的文件名。如果 open_mode 为‘w’，则将现存文件覆盖。
open_mode	VARCHAR2	打开方式。其合法值有：‘r’ 读文本，‘w’ 写文本，‘a’ 追加文本。该参数对大小写不敏感。如果指定了‘a’ 方式但该文件并不存在，则就按‘w’ 方式创建新文件
return value	UTL_FILE.FILE_TYPE	后续函数使用的文件句柄

表7-5 UTL_FILE引发的异常

异 常	引 发 条 件	引 发 函 数
INVALID_PATH	非法或不能访问的目录或文件名	FOPEN
INVALID_MODE	非法打开模式	FOPEN
INVALID_FILEHANDLE	文件句柄指示的文件没有打开	FCLOSE, GET_LINE, PUT, PUT_LINE, NEW LINE, PUTF, FFLUSH
INVALID_OPERATION	文件不能按要求打开，该异常与操作系统的权限有关。该异常也可能在企图对以读方式打开的进行写操作时，或企图对以写方式打开的文件进行读操作时引发	GET_LINE, PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH
INVALID_MAXLINESIZE	所指定的最大行数太大或太小。该异常只在 Oracle8.0.5 及更高的版本下引发	FOPEN
READ_ERROR	在读操作期间出现的操作系统错误	GET_LINE
WEITE_ERROR	在写操作期间出现的操作系统错误	PUT, PUT_LINE, PUTF, NEW_LINE, FFLUSH, FCLOSE, FCLOSE_ALL
INTERNAL_ERROR	未说明的内部错误	所有函数
NO_DATA_FOUND	读操作中遇到了文件结束符	GET_LINE
VALUE_ERROR	输入文件大于 GET_LINE 中指定的缓冲区	GET_LINE

FOPEN 可以引发下列异常之一：

- UTL_FILE.INVALID_PATH

- UTL_FILE.INVALID_MODE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.INTERNAL_ERROR

2. 使用参数max_linesize的FOPEN

在Oracle8.0.5及更高版本下，UTL_FILE提供了FOPEN的另一个重载版本：

```
FUNCTION FOPEN( location IN VARCHAR2,
                filename IN VARCHAR2,
                open_mode IN VARCHAR2,
                max_linesize IN BINARY_INTEGER)

    RETURN FILE_TYPE;
```

其中，location,filename,open_mode参数的意义与FOPEN的第一版相同。参数max_linesize用来说明文件的最大行数，其范围从1-32767.如果没有使用该参数，则最大行数为1024。如果该参数的值小于1或大于32767，则引发UTL_FILE.INVALID_MAXLINESIZE异常。

3. FCLOSE

当文件的读写操作结束后，要使用FCLOSE将该文件关闭。关闭文件将释放UTL_FILE用来操作该文件所占用的资源。FCLOSE的语法如下：

```
PROCEDURE FCLOSE (file_handle IN OUT FILE_TYPE);
```

FCLOSE的唯一参数是文件句柄file_handle。在关闭该文件之前，任何等待写入文件的未决变更都将实现。如果在写入过程中出现了错误，则将引发UTL_FILE.WRITE_ERROR异常。如果文件句柄没有指向合法打开的文件，将引发UTL_FILE.INVALID_FILEHANDLE异常。

4. IS_OPEN

该逻辑值函数在指定的文件处于打开的状态下返回TRUE，反之返回FALSE。IS_OPEN的定义如下：

```
FUNCTION IS_OPEN(file_handle IN FILE_TYPE)
    RETURN BOOLEAN;
```

即使IS_OPEN返回TRUE，在文件处于打开状态下也存在着操作系统出错的可能。

5. FCLOSE_ALL

该过程将关闭所有打开的文件。该功能可用来清理错误句柄。该过程的定义如下：

```
PROCEDURE FCLOSE_ALL;
```

该过程没有参数。在文件关闭前，文件的所有未决状态都将被写入文件。由于可能执行写操作，所以如果在写操作中出现错误时，该过程可能会引发UTL_FILE.WRITE_ERROR异常。

警告 FCLOSE_ALL将关闭文件并释放UTL_FILE占用的资源。然而，该过程并不对文件做关闭标志。这样一来，在执行FCLOSE_ALL后，IS_OPEN仍将返回TRUE。除此之外，执行FCLOSE_ALL之后的任何读或写操作都将失败。

7.2.4 文件输出

有五个过程可用来把数据输出到文件中。它们分别是：PUT、PUT_LINE、NEW_LINE、

PUTF和FFLUSH。其中PUT、PUT_LINE、NEW_LINE的作用非常类似于我们在第3章中讨论过的包DBMS_OUTPUT中的对应参数。输出记录的最大容量是1023个字节（除非使用FOPEN说明新的容量值）。该记录中包括了一个表示新行的字符。

1. 过程PUT

PUT将把指定的字符串输出到指定文件中。该文件在执行PUT操作前应处于打开状态。PUT的定义如下：

```
PROCEDURE PUT ( file_handle IN FILE_TYPE,
                buffer IN VARHCAR2 );
```

PUT将不在该文件中追加新行字符，如果需要的话，则必须使用PUT_LINE或NEW_LINE在行中加入行结束符。如果在写入操作时出现了操作系统错误，将引发UTL_FILE.WRITE_ERROR异常。PUT参数的说明如下：

参 数	类 型	说 明
file_handle	UTL_FILE.FILE_TYPE	由FOPEN返回的文件句柄。如果该句柄是非法的，则将引发UTL_FILE.INVALID_FILEHANDLE异常
buffer	VARCHAR2	等待写入文件中的文本字符串。如果没有使用‘w’或‘a’方式来打开文件，就引发UTL_FILE.INVALID_OPERATION

2. 过程NEW_LINE

NEW_LINE将把一个或多个行结束符写入到指定的文件中。该过程的语法定义如下：

```
PROCEDURE NEW_LINE(file_handle IN FILE_TYPE,
                   lines IN NATURAL:=1);
```

行结束符与操作系统有关，不同的操作系统可能使用不同的行结束符。如果在写入操作时出现了操作系统错误，则引发UTL_FILE.WRITE_ERROR异常。参数NEW_LINE的说明如下：

参 数	类 型	说 明
file_handle	UTL_FILE.FILE_TYPE	由FOPEN返回的文件句柄。如果该句柄不合法，将引发UTL_FILE.INVALID_FILEHANDLE异常
lines	NATURAL	输出的行结束符个数。其默认值是1，即输出一个新行。如果文件没有用‘w’或‘a’方式打开，系统将引发UTL_FILE.INVALID_OPERATION异常

3. PUT_LINE

PUT_LINE把指定的字符串输出到指定的文件中，被写入的文件必须以写操作方式打开，在字符串写入该文件后，系统在行尾加上系统定义的行结束符。PUT_LINE的定义如下：

```
PROCEDURE PUT_LINE(file_handle IN FILE_TYPE,
                   buffer IN VARCHAR2);
```

PUT_LINE的参数的含义如下表所示。调用PUT_LINE等价于调用PUT后再调用NEW_LINE来写入行结束符。如果在写入期间发生了操作系统错误，则引发UTL_FILE.WRITE_ERROR异常。

参 数	类 型	说 明
file_handle	UTL_FILE.FILE_TYPE	由FOPEN返回的文件句柄。如果该句柄不合法，将引发 UTL_FILE.INVALID_FILEHANDLE异常
buffer	VARCHAR2	写入到文件中的文本字符串。如果写入文件没有以‘w’或‘a’模式打开，则引发 UTL_FILE.INVALID_OPERATION异常

4. 过程PUTF

PUTF的功能与PUT类似，但该过程允许对输出字符串进行格式。PUTF实现了C语言printf()的部分功能，其语法也类似于C语言的printf()语句。PUTF的语法如下：

```
PROCEDURE PUTF( file_handle IN FILE_TYPE,
                 format IN VARCHAR2,
                 arg1 IN VARCHAR2 DEFAULT NULL,
                 arg2 IN VARCHAR2 DEFAULT NULL,
                 arg3 IN VARCHAR2 DEFAULT NULL,
                 arg4 IN VARCHAR2 DEFAULT NULL,
                 arg5 IN VARCHAR2 DEFAULT NULL);
```

请注意，参数arg1-arg5具有默认值，因此这几个参数可以省略。格式字符串format除了带有正常的文本外，还有两个特殊字符‘%s’和‘\n’。在格式字符串中出现‘%s’的地方都将用一个上述可选参数替代。格式字符串中出现‘\n’的地方都将用一个新的行结束符替代。上述参数的详细用法在下面案例后面的表中给予说明。对于PUT和PUT_LINE，如果在写入过程中出现了操作系统错误，则引发 UTL_FILE.WRITE_ERROR异常。

例如，如果我们运行下面的块：

```
DECLARE
  v_OutputFile UTL_FILE.FILE_TYPE;
  v_Name VARCHAR2(10) := 'Scott';
BEGIN
  v_OutputFile := UTL_FILE.FOPEN(...);
  UTL_FILE.PUTF(v_OutputFile,
    'Hi there!\nMy name is %s, and I am a %s major.\n',
    v_Name, 'Computer Science');
  FCLOSE(v_OutputFile);
END;
```

则输出文件将带有下面内容：

```
Hi There!
My name is Scott, and I am a Computer Science major.
```

参 数	类 型	说 明
file_handle	UTL_FILE.FILE_TYPE	FOPEN返回的文件句柄。如果该句柄不合法，则引发 UTL_FILE.INVALID_FILEHANDLE异常
format	VARCHAR2	格式字符串包括常规文本和两个特殊的格式字符‘%s’和‘\n’。如果文件没有以‘w’或‘a’的模式打开，则引发 UTL_FILE.INVALID_OPERATION异常
arg1-arg5	VARCHAR2	五个可选的参数。每个参数将被对应‘%s’的格式字符代替。如果‘%s’字符多于参数的话，就使用空字符串来代替格式字符

下载

5. 过程FFLUSH

使用PUT , PUT_LINE,PUTF , 或NEW_LINE输出的数据通常是存储在缓冲区中的。当该缓冲区装满后，该缓冲区中的字符将被送往输出文件。FFLUSH强行把缓冲区中的字符立即写入指定文件。需要提醒的是，FFLUSH只是把缓冲区中以 NEW_LINE字符结尾的行写入文件中。任何最后执行PUT操作放入缓冲区的字符都将在缓冲区中保留。该过程的语法是：

```
PROCEDURE FFLUSH (file_handle IN FILE_TYPE);
```

FFLUSH可以引发下列异常：

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.WRITE_ERROR

7.2.5 文件输入

GET_LINE是用来从文件中执行读入操作的，该过程每次从指定的文件中读入一行文本并到将该文本串送入缓冲区，新行字符不包括在返回字符串中。下面是该过程的定义：

```
PROCEDURE GET_LINE(file_handle IN FILE_TYPE,
                   buffer OUT VARCHAR2);
```

当从指定文件中读入最后一行时，异常 NO_DATA_FOUND将会引发。如果该行不适宜进入作为实参提供的缓冲区，就将引发异常 VALUE_ERROR。读入空行将返回一个空字符串 NULL。如果在读入期间，操作系统系统出现了错误，则引发 UTL_FILE.READ_ERROR异常。输入行的最大长度是1022字节。（除非使用FOPEN的参数max_linesize指定最大行数。）下面是这些参数的说明：

参 数	类 型	说 明
file_handle	UTL_FILE.FILE_TYPE	FOPEN返回的文件句柄。如果该句柄不合法，则将引发 UTL_FILE.INVALID_FILEHANDLE异常
buffer	VARCHAR2	将一行写入的缓冲区。如果文件没有以读‘r’的模式打开，则引发 UTL_FILE.INVALID_OPERATION异常

7.2.6 文件操作案例

本节将分析三个使用 UTL_FILE的程序案例。第一个案例是我们在第3章介绍的包DEBUG的新版本。第二个案例从一个文件中将学生信息读入并装入到表中。第三个程序是打印成绩单。

1. 包DEBUG

UTL_FILE的第一种用法是实现调试程序包。由于 DBMS_OUTPUT只能在块运行结束后才能打印执行结果（我们在第3章讨论过该问题），而UTL_FILE可以提供更快捷的输出。下面是用 UTL_FILE实现的DEBUG包：

节选自在线代码Debug.sql

```
CREATE OR REPLACE PACKAGE Debug AS
/* Global variables to hold the name of the debugging file and
```

```
directory. */
v_DebugDir VARCHAR2(50) := '/tmp';
v_DebugFile VARCHAR2(20) := 'debug.out';
/* Call Debug to output a line consisting of:
   p_Description: p_Value
   to the debugging file. */
PROCEDURE Debug(p_Description IN VARCHAR2,
                p_Value IN VARCHAR2);

/* Closes the debugging file first, then calls FileOpen to
set the packaged variables and open the file with the new
parameters. */
PROCEDURE Reset(p_NewFile IN VARCHAR2 := v_DebugFile,
                p_NewDir IN VARCHAR2 := v_DebugDir);

/* Sets the packaged variables to p_NewFile and p_NewDir, and
   opens the debugging file. */
PROCEDURE FileOpen(p_NewFile IN VARCHAR2 := v_DebugFile,
                   p_NewDir IN VARCHAR2 := v_DebugDir);

/* Closes the debugging file. */
PROCEDURE FileClose;
END Debug;

CREATE OR REPLACE PACKAGE BODY Debug AS
  v_DebugHandle UTL_FILE.FILE_TYPE;

  PROCEDURE Debug(p_Description IN VARCHAR2,
                  p_Value IN VARCHAR2) IS
  BEGIN
    IF NOT UTL_FILE.IS_OPEN(v_DebugHandle) THEN
      FileOpen;
    END IF;
    /* Output the info, and flush the file. */
    UTL_FILE.PUTF(v_DebugHandle, '%s: %s\n',
                  p_Description, p_Value);
    UTL_FILE.FFLUSH(v_DebugHandle);
  EXCEPTION
    WHEN UTL_FILE.INVALID_OPERATION THEN
      RAISE_APPLICATION_ERROR(-20102,
                             'Debug: Invalid Operation');
    WHEN UTL_FILE.INVALID_FILEHANDLE THEN
      RAISE_APPLICATION_ERROR(-20103,
                             'Debug: Invalid File Handle');
    WHEN UTL_FILE.WRITE_ERROR THEN
      RAISE_APPLICATION_ERROR(-20104,
                             'Debug: Write Error');
    WHEN UTL_FILE.INTERNAL_ERROR THEN
```

```
RAISE_APPLICATION_ERROR(-20104,
                           'Debug: Internal Error');

END Debug;

PROCEDURE Reset(p_NewFile IN VARCHAR2 := v_DebugFile,
                p_NewDir IN VARCHAR2 := v_DebugDir) IS
BEGIN

    /* Make sure the file is closed first. */
    IF UTL_FILE.IS_OPEN(v_DebugHandle) THEN
        FileClose;
    END IF;

    FileOpen(p_NewFile,p_NewDir);

END Reset;

PROCEDURE FileOpen(p_NewFile IN VARCHAR2 := v_DebugFile,
                   p_NewDir IN VARCHAR2 := v_DebugDir) IS
BEGIN

    /* Open the file for writing. */
    v_DebugHandle := UTL_FILE.FOPEN(p_NewDir, p_NewFile, 'w');
    /* Set the packaged variables to the values just passed in. */
    v_DebugFile := p_NewFile;
    v_DebugDir := p_NewDir;
EXCEPTION
    WHEN UTL_FILE.INVALID_PATH THEN
        RAISE_APPLICATION_ERROR(-20100, 'Open: Invalid Path');
    WHEN UTL_FILE.INVALID_MODE THEN
        RAISE_APPLICATION_ERROR(-20101, 'Open: Invalid Mode');
    WHEN UTL_FILE.INVALID_OPERATION THEN
        RAISE_APPLICATION_ERROR(-20101, 'Open: Invalid Operation');
    WHEN UTL_FILE.INTERNAL_ERROR THEN
        RAISE_APPLICATION_ERROR(-20101, 'Open: Internal Error');
END FileOpen;

PROCEDURE FileClose IS
BEGIN
    UTL_FILE.FCLOSE(v_DebugHandle);
EXCEPTION
    WHEN UTL_FILE.INVALID_FILEHANDLE THEN
        RAISE_APPLICATION_ERROR(-20300,
                               'Close: Invalid File Handle');
    WHEN UTL_FILE.WRITE_ERROR THEN
        RAISE_APPLICATION_ERROR(-20301,
                               'Close: Write Error');
    WHEN UTL_FILE.INTERNAL_ERROR THEN
        RAISE_APPLICATION_ERROR(-20302,
```

```
'Close: Internal Error');

END FileClose;
END Debug;
```

使用该DEBUG包的方法是很直观的。该包的变量 v_DebugDir和v_DebugFile指定了输出文件的位置和名称。如果我们先调用 Debug.Debug，则该文件打开时将带有上述的值。为了修改这些值，我们可以调用 Debug.FileOpen来复位该包的变量并重新打开该文件。或者调用 Debug.Reset首先将该文件关闭，然后再执行 FileOpen的操作。Debug.FileClose将在操作结束时关闭该文件。例如，如果我们执行下面的块：

节选自在线代码CallDebug.sql

```
BEGIN
    Debug.Debug('Scott', 'First call');
    Debug.Reset('debug2.out', '/tmp');
    Debug.Debug('Scott', 'Second call');
    Debug.Debug('Scott', 'Third call');
    Debug.FileClose;
END;
```

接着文件/tmp/debug.out中将包括下面的内容：

Scott: First call

文件/tmp/debug2.out将包括下面的内容：

Scott:Second call

Scott:Third call

提示 请注意程序中各种例程的异常处理程序。这些异常处理程序识别引发的错误类型以及引发该错误的过程。这种方法是值得我们在使用 UTL_FILE时借鉴。

2. 加载学生数据程序

过程LoadStudents将根据其接收的文件内容来对表进行插入操作。该文件是用逗号分界的，也就是说，该文件的每一行都是一个记录，该行中的逗号用来分割字段。这是文本文件的常用格式。下面是该过程的代码：

节选自在线代码LoadStudents.sql

```
CREATE OR REPLACE PROCEDURE LoadStudents (
    /* Loads the students table by reading a comma-delimited file.
       The file should have lines that look like

       first_name,last_name,major

       The student ID is generated from student_sequence.
       The total number of rows inserted is returned by
       p_TotalInserted. */
    p_FileDir IN VARCHAR2,
    p_FileName IN VARCHAR2,
    p_TotalInserted IN OUT NUMBER) AS
```

```
v_FileHandle UTL_FILE.FILE_TYPE;
v_NewLine VARCHAR2(100); -- Input line
v_FirstName students.first_name%TYPE;
v_LastName students.last_name%TYPE;
v_Major students.major%TYPE;
/* Positions of commas within input line. */
v_FirstComma NUMBER;
v_SecondComma NUMBER;

BEGIN
    -- Open the specified file for reading.
    v_FileHandle := UTL_FILE.FOPEN(p_FileDir, p_FileName, 'r');

    -- Initialize the output number of students.
    p_TotalInserted := 0;

    -- Loop over the file, reading in each line. GET_LINE will
    -- raise NO_DATA_FOUND when it is done, so we use that as the
    -- exit condition for the loop.
LOOP
    BEGIN
        UTL_FILE.GET_LINE(v_FileHandle, v_NewLine);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            EXIT;
    END;

    -- Each field in the input record is delimited by commas. We
    -- need to find the locations of the two commas in the line
    -- and use these locations to get the fields from v_NewLine.
    -- Use INSTR to find the locations of the commas.
    v_FirstComma := INSTR(v_NewLine, ',', 1, 1);
    v_SecondComma := INSTR(v_NewLine, ',', 1, 2);

    -- Now we can use SUBSTR to extract the fields.
    v_FirstName := SUBSTR(v_NewLine, 1, v_FirstComma - 1);
    v_LastName := SUBSTR(v_NewLine, v_FirstComma + 1,
                         v_SecondComma - v_FirstComma - 1);
    v_Major := SUBSTR(v_NewLine, v_SecondComma + 1);

    -- Insert the new record into students.
    INSERT INTO students (ID, first_name, last_name, major)
        VALUES (student_sequence.nextval, v_FirstName,
                v_LastName, v_Major);
    p_TotalInserted := p_TotalInserted + 1;
```

过程LoadStudents使用的一个输入文件的内容如下：

节选自在线代码students.sql

```
    v_LastName, v_Major);
    p_TotalInserted := p_TotalInserted + 1;
```

```
END LOOP;
-- Close the file.
UTL_FILE.FCLOSE(v_FileHandle);

COMMIT;
EXCEPTION
  -- Handle the UTL_FILE exceptions meaningfully, and make sure
  -- that the file is properly closed.
WHEN UTL_FILE.INVALID_OPERATION THEN
  UTL_FILE.FCLOSE(v_FileHandle);
  RAISE_APPLICATION_ERROR(-20051,
    'LoadStudents: Invalid Operation');
WHEN UTL_FILE.INVALID_FILEHANDLE THEN
  UTL_FILE.FCLOSE(v_FileHandle);
  RAISE_APPLICATION_ERROR(-20052,
    'LoadStudents: Invalid File Handle');
WHEN UTL_FILE.READ_ERROR THEN
  UTL_FILE.FCLOSE(v_FileHandle);
  RAISE_APPLICATION_ERROR(-20053,
    'LoadStudents: Read Error');
WHEN UTL_FILE.INVALID_PATH THEN
  UTL_FILE.FCLOSE(v_FileHandle);
  RAISE_APPLICATION_ERROR(-20054,
    'LoadStudents: Invalid Path');
WHEN UTL_FILE.INVALID_MODE THEN
  UTL_FILE.FCLOSE(v_FileHandle);
  RAISE_APPLICATION_ERROR(-20055,
    'LoadStudents: Invalid Mode');
WHEN UTL_FILE.INTERNAL_ERROR THEN
  UTL_FILE.FCLOSE(v_FileHandle);
  RAISE_APPLICATION_ERROR(-20056,
    'LoadStudents: Internal Error');
WHEN VALUE_ERROR THEN
  UTL_FILE.FCLOSE(v_FileHandle);
  RAISE_APPLICATION_ERROR(-20057,
    'LoadStudents: Value Error');
WHEN OTHERS THEN
  UTL_FILE.FCLOSE(v_FileHandle);
  RAISE;
END LoadStudents;
```

过程LoadStudents使用的一个输入文件的内容如下

```
Scott,Smith,Computer Science
Margaret,Mason,History
Joanne,Junebug,Computer Science
Manish,Murgratroid,Economics
Patrick,Poll,History
Timothy,Taller,History
```

```
Barbara,Blues,Economics  
David,Dinsmore,Music  
Ester,Elegant,Nutrition  
Rose,Riznit,Music  
Rita,Razmataz,Nutrition  
Shay,Shariatpanahy,Computer Science
```

注意 过程LoadStudents使用student_sequence序列来确认学生的ID。如果该序列在表students被载入后进行了重建的话，该过程将可能返回一个已经在表 students中的值。在这种情况下，表students的主约束键将会非法。

3. 打印成绩单程序

该案例介绍了如何使用UTL_FILE来打印学生的成绩单。下面是过程CalculateGPA的代码：

```
节选自在线代码CalculateGPA.sql  
CREATE OR REPLACE PROCEDURE CalculateGPA (  
/* Returns the grade point average for the student identified  
by p_StudentID in p_GPA. */  
p_StudentID IN students.ID%TYPE,  
p_GPA OUT NUMBER) AS  
  
CURSOR c_ClassDetails IS  
SELECT classes.num_credits, rs.grade  
FROM classes, registered_students rs  
WHERE classes.department = rs.department  
AND classes.course = rs.course  
AND rs.student_id = p_StudentID;  
v_NumericGrade NUMBER;  
v_TotalCredits NUMBER := 0;  
v_TotalGrade NUMBER := 0;  
  
BEGIN  
FOR v_ClassRecord in c_ClassDetails LOOP  
-- Determine the numeric value for the grade.  
SELECT DECODE(v_ClassRecord.grade, 'A', 4,  
               'B', 3,  
               'C', 2,  
               'D', 1,  
               'E', 0)  
INTO v_NumericGrade  
FROM dual;  
  
v_TotalCredits := v_TotalCredits + v_ClassRecord.num_credits;  
v_TotalGrade := v_TotalGrade +  
               (v_ClassRecord.num_credits * v_NumericGrade);  
END LOOP;  
  
p_GPA := v_TotalGrade / v_TotalCredits;
```

```
END CalculateGPA;
```

PrintTranscript 用以下代码创建

节选自在线代码printTranscript.sql

```
CREATE OR REPLACE PROCEDURE PrintTranscript (
    /* Outputs a transcript to the indicated file for the indicated
       student. The transcript will consist of the classes for which
       the student is currently registered and the grade received
       for each class. At the end of the transcript, the student's
       GPA is output. */
    p_StudentID IN students.ID%TYPE,
    p_FileDir IN VARCHAR2,
    p_FileName IN VARCHAR2) AS

v_StudentGPA NUMBER;
v_StudentRecord students%ROWTYPE;
v_FileHandle UTL_FILE.FILE_TYPE;
v_NumCredits NUMBER;

CURSOR c_CurrentClasses IS
    SELECT *
        FROM registered_students
        WHERE student_id = p_StudentID;

BEGIN
-- Open the output file in append mode.
v_FileHandle := UTL_FILE.FOPEN(p_FileDir, p_FileName, 'a');

SELECT *
    INTO v_StudentRecord
    FROM students
    WHERE ID = p_StudentID;

-- Output header information. This consists of the current
-- date and time, and information about this student.

UTL_FILE.PUTF(v_FileHandle, 'Student ID: %s\n',
    v_StudentRecord.ID);
UTL_FILE.PUTF(v_FileHandle, 'Student Name: %s %s\n',
    v_StudentRecord.first_name, v_StudentRecord.last_name);
UTL_FILE.PUTF(v_FileHandle, 'Major: %s\n',
    v_StudentRecord.major);
UTL_FILE.PUTF(v_FileHandle, 'Transcript Printed on: %s\n\n\n',
    TO_CHAR(SYSDATE, 'Mon DD,YYYY HH24:MI:SS'));

UTL_FILE.PUT_LINE(v_FileHandle, 'Class Credits Grade');
UTL_FILE.PUT_LINE(v_FileHandle, '----- ----- -----');
FOR v_ClassesRecord in c_CurrentClasses LOOP
```

```
-- Determine the number of credits for this class.  
SELECT num_credits  
    INTO v_NumCredits  
   FROM classes  
 WHERE course = v_ClassesRecord.course  
AND department = v_ClassesRecord.department;  
  
-- Output the info for this class.  
UTL_FILE.PUTF(v_FileHandle, '%s %s %s\n',  
    RPAD(v_ClassesRecord.department || ' ' ||  
          v_ClassesRecord.course, 7),  
    LPAD(v_NumCredits, 7),  
    LPAD(v_ClassesRecord.grade, 5));  
END LOOP;  
  
-- Determine the GPA.  
CalculateGPA(p_StudentID, v_StudentGPA);  
  
-- Output the GPA.  
UTL_FILE.PUTF(v_FileHandle, '\n\nCurrent GPA: %s\n',  
    TO_CHAR(v_StudentGPA, '9.99'));  
  
-- Close the file.  
UTL_FILE.FCLOSE(v_FileHandle);  
EXCEPTION  
-- Handle the UTL_FILE exceptions meaningfully, and make sure  
-- that the file is properly closed.  
WHEN UTL_FILE.INVALID_OPERATION THEN  
    UTL_FILE.FCLOSE(v_FileHandle);  
    RAISE_APPLICATION_ERROR(-20061,  
        'PrintTranscript: Invalid Operation');  
WHEN UTL_FILE.INVALID_FILEHANDLE THEN  
    UTL_FILE.FCLOSE(v_FileHandle);  
    RAISE_APPLICATION_ERROR(-20062,  
        'PrintTranscript: Invalid File Handle');  
WHEN UTL_FILE.WRITE_ERROR THEN  
    UTL_FILE.FCLOSE(v_FileHandle);  
    RAISE_APPLICATION_ERROR(-20063,  
        'PrintTranscript: Write Error');  
WHEN UTL_FILE.INVALID_MODE THEN  
    UTL_FILE.FCLOSE(v_FileHandle);  
    RAISE_APPLICATION_ERROR(-20064,  
        'PrintTranscript: Invalid Mode');  
WHEN UTL_FILE.INTERNAL_ERROR THEN  
    UTL_FILE.FCLOSE(v_FileHandle);  
    RAISE_APPLICATION_ERROR(-20065,  
        'PrintTranscript: Internal Error');  
END PrintTranscript;
```

假设我们给定表registered_students (由过程relTables.sql创建的) 的内容如下：

```
SQL> select * from registered_students;
STUDENT_ID DEP COURSE G
-----
10000 CS 102 A
10002 CS 102 B
10003 CS 102 C
10000 HIS 101 A
10001 HIS 101 B
10002 HIS 101 B
10003 HIS 101 A
10004 HIS 101 C
10005 HIS 101 C
10006 HIS 101 E
10007 HIS 101 B
10008 HIS 101 A
10009 HIS 101 D
10010 HIS 101 A
10008 NUT 307 A
10010 NUT 307 A
10009 MUS 410 B
10006 MUS 410 E
10011 MUS 410 B
10000 MUS 410 B
20 rows selected.
```

如果我们调用过程PrintTranscript来打印学生ID号为10000和10009的成绩单，我们将得到下面两个输出文件：

```
Student ID: 10000
Student Name: Scott Smith
Major: Computer Science
Transcript Printed on: Apr 26,1999 22:24:07
Class Credits Grade
-----
CS 102      4      A
HIS 101      4      A
MUS 410      3      B
```

Current GPA: 3.73

```
Student ID: 10009
Student Name: Rose Riznit
Major: Music
Transcript Printed on: Apr 26,1999 22:24:31
```

Class Credits Grade

HIS 101	4	D
MUS 410	3	B

Current GPA: 1.86

7.3 小结

我们在本章分析了两个实用程序包：一个是 DBMS_JOB，另一个是 UTL_FILE。数据库作业允许过程由数据库在预先定义的时间自动启动运行。包 UTL_FILE 在确保服务器安全的前提下为 PL/SQL 语言增加了文件输入输出的功能。这些实用程序为 PL/SQL 语言提供了新的功能。