

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，你将会在自己或其它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的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，其原因是我们只需在程序中插入 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

```
/* 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
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和内部行计数器（该行计数器过程也被包的初始化代码调用）。行计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.sqp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着空白字符，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；

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_Department 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行，因此给变量 p_NumberStudents 赋值为 0 的原因。

```
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
```

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。

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：

```
SQL> SELECT DISTINCT DUMP(department)
```

```
2 FROM registered_students
```

```
3 WHERE department = 'CS';
```

```
DUMP(DEPARTMENT)
```

```
-----
```

```
Typ=96 Len=3: 67,83,32
```

在做了上述修改后，程序 AverageGrade 就可以正确运行了：

```
CREATE OR REPLACE FUNCTION AverageGrade (
```

```
    p_Department IN CHAR,
```

```
    p_Course IN NUMBER) RETURN VARCHAR2 AS
```

```
...
```

```
BEGIN
```

```
...
```

```
END AverageGrade;
```

完成这些后，我们得到 AverageGrade 的正确结果：

```
L 3-13 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 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_OUINPUT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，就必须声名一个类型为DBMS_OUINPUT.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

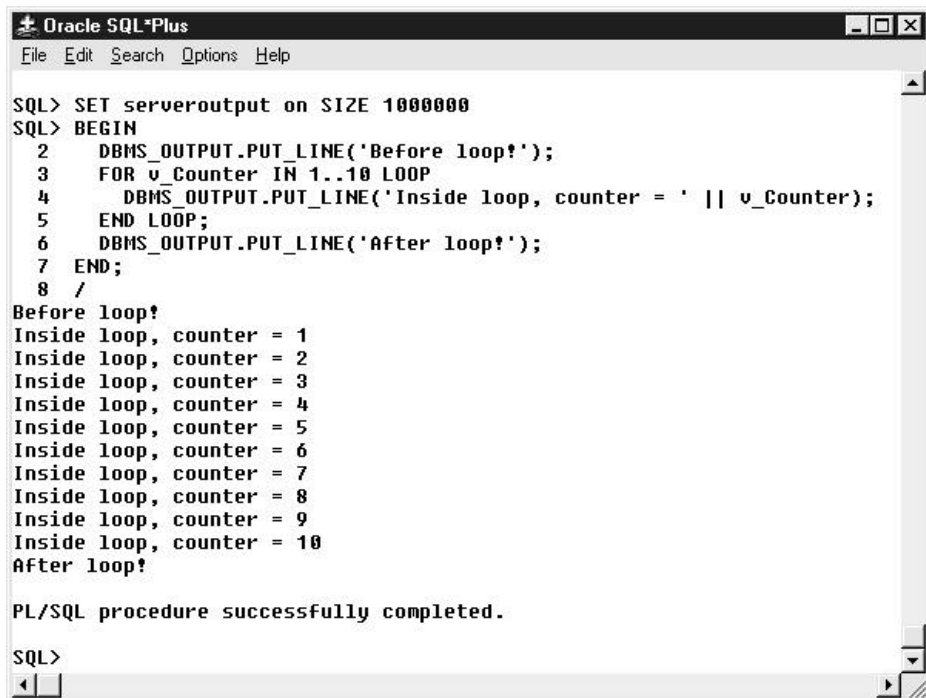
包DBMS_OUTPUT自身并不具有任何打印机制,该包只是简单地实现了先入先出（FIFO）的数据结构。但这样如何实现打印呢？SQL *Plus和服务器管理器都提供了叫做 SERVEROUTPUT的选择项。另外，许多第三方提供的开发工具（包括图形开发和调试工具）提供了显示DBMS_OUTPUT数据的功能。如果指定 SERVEROUTPUT选项，SQL *Plus将在PL/SQL块结束时自动调用DBMS_OUTPUT.GET_LINES并把结果打印到屏幕。图3-1是打印窗口。

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，它不适用于用来生成报表。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_OUTPUT和SQL*Plus。



```
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 /
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!

PL/SQL procedure successfully completed.

SQL>
```

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.
```

or

```
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还非常好的。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/SQL 2.1 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
  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;
```

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。

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;
```

表3-1 PL/SQL调试器功能比较

功能	Rapid SQL	XPEDITER/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

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单 Debug | Start Debugging, 或单击如图 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。

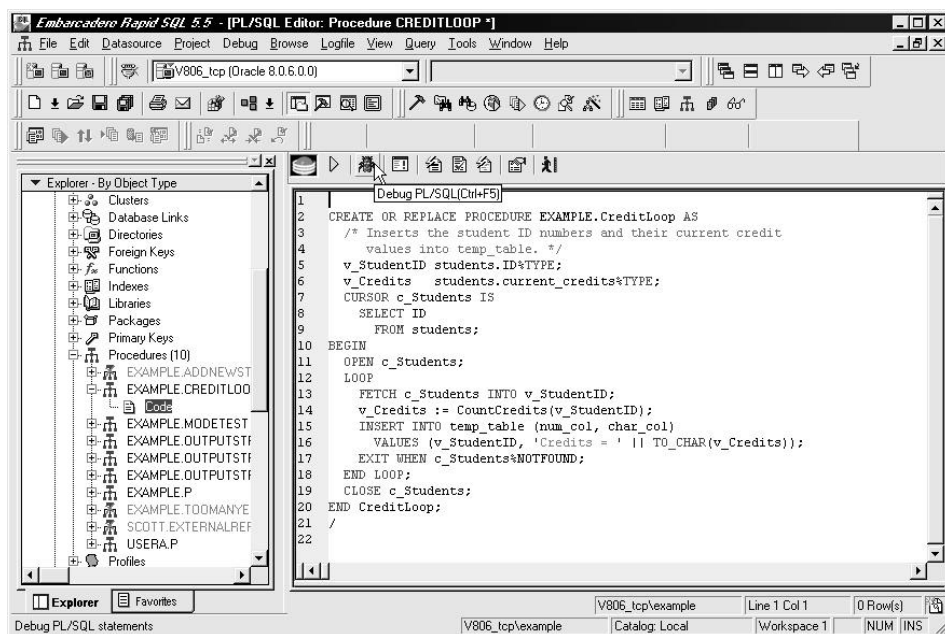


图3-2 准备开始调试过程 CreditLoop的窗口

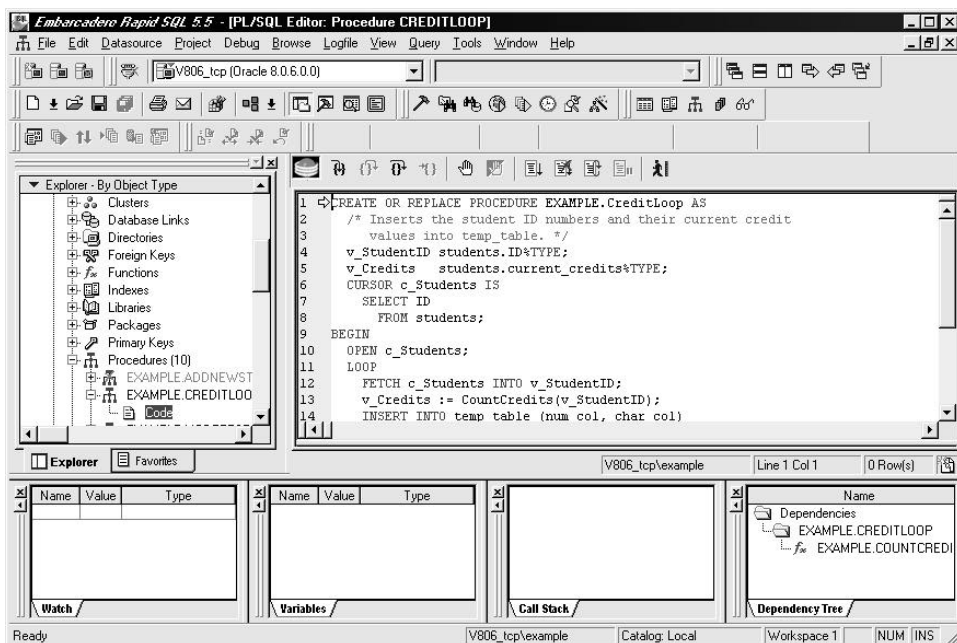


图3-3 初始化的CreditLoop调试会话窗口

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。

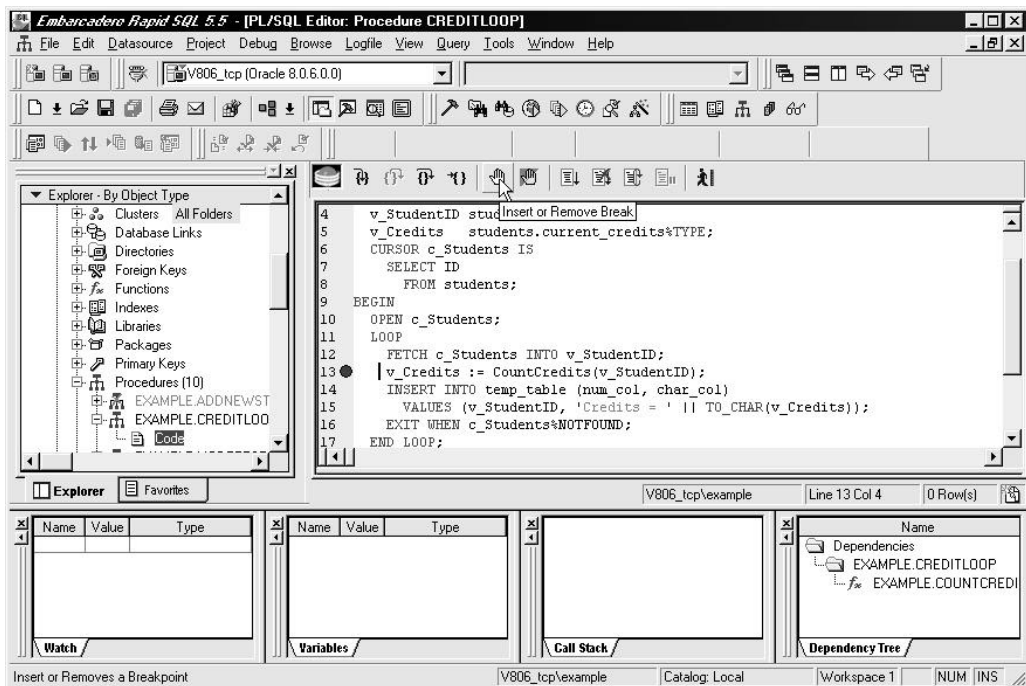


图3-4 在窗口中设置断点

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
```

```
-----
```

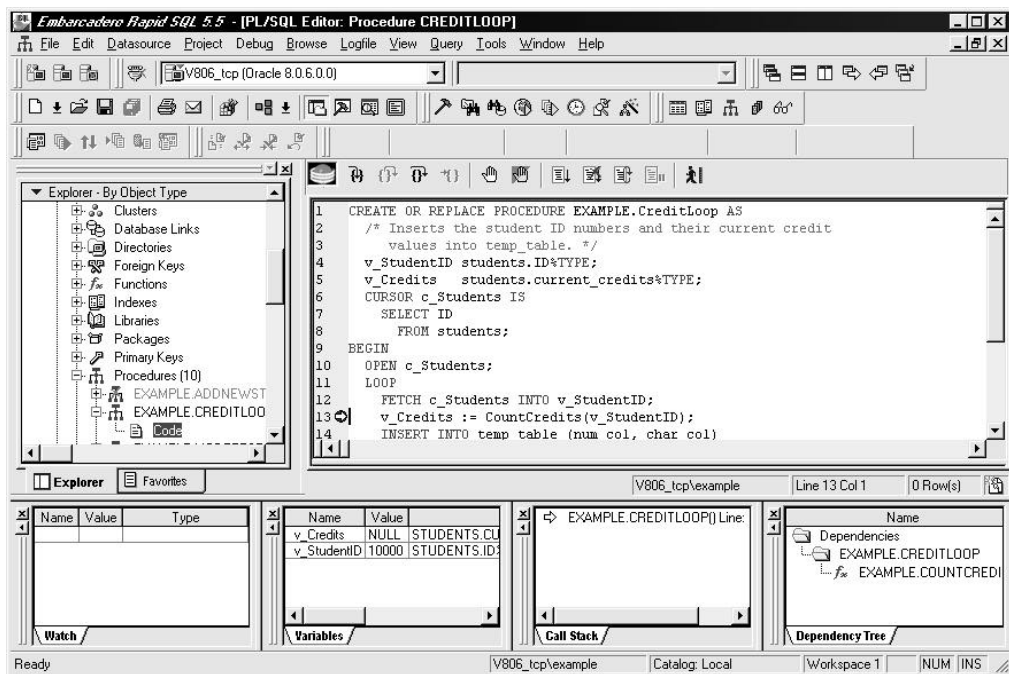



图3-5 停止在断点的运行窗口

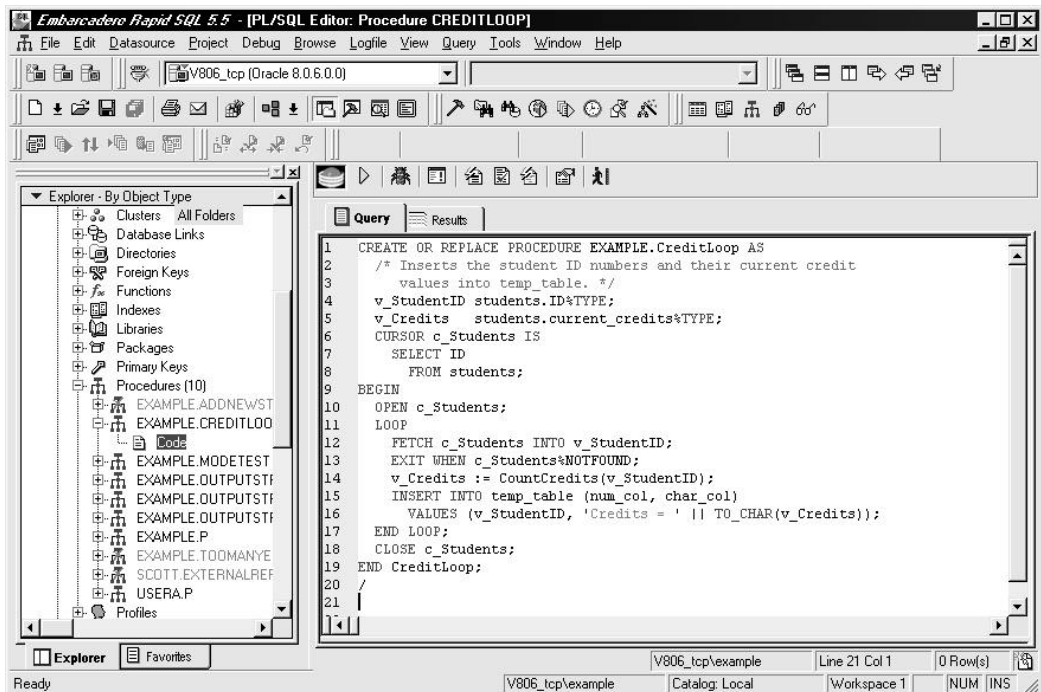


图3-6 CreditLoop的正确版本

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。这样一来就使我们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。

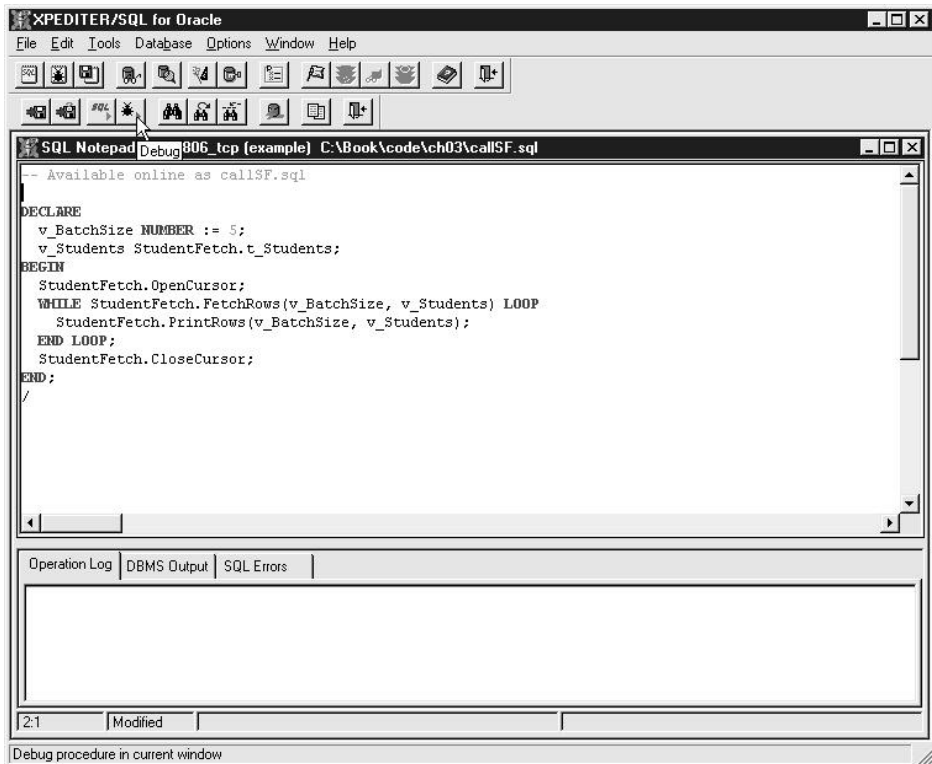


图3-7 准备调试调用块的窗口

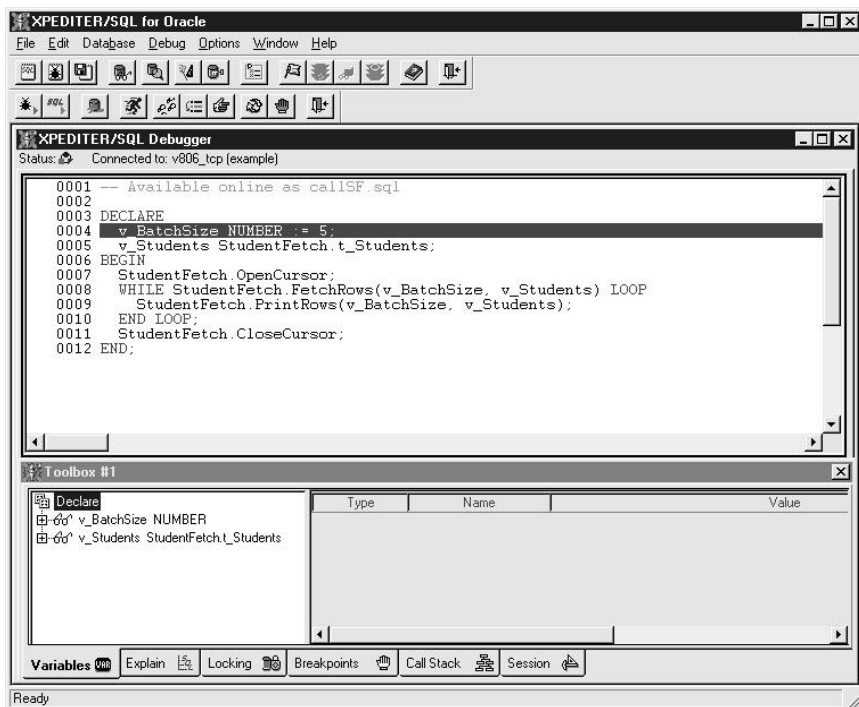


图3-8 调试窗口显示的调用块

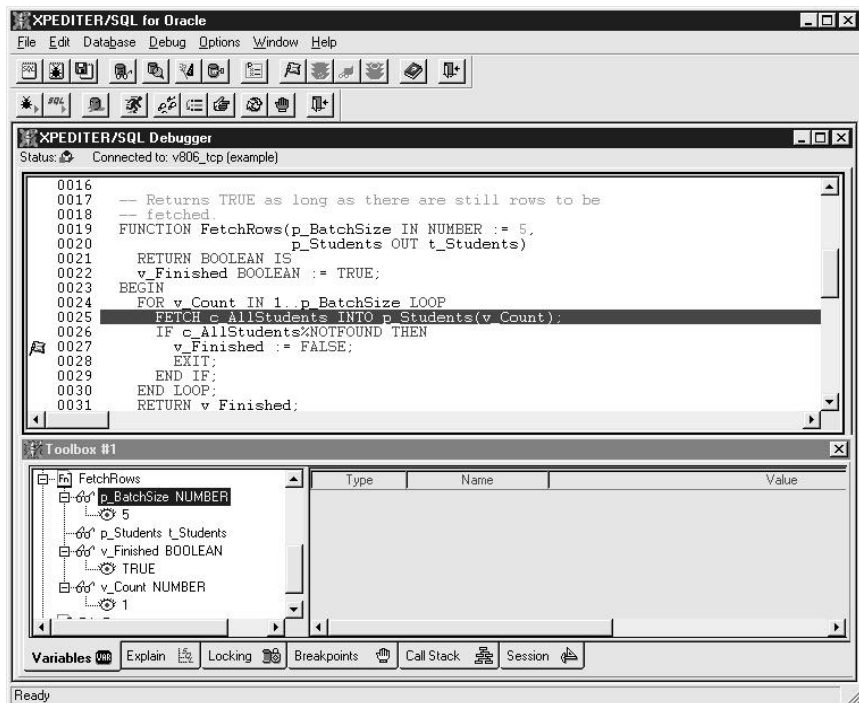


图3-9 设置断点

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 XPEDITER/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。

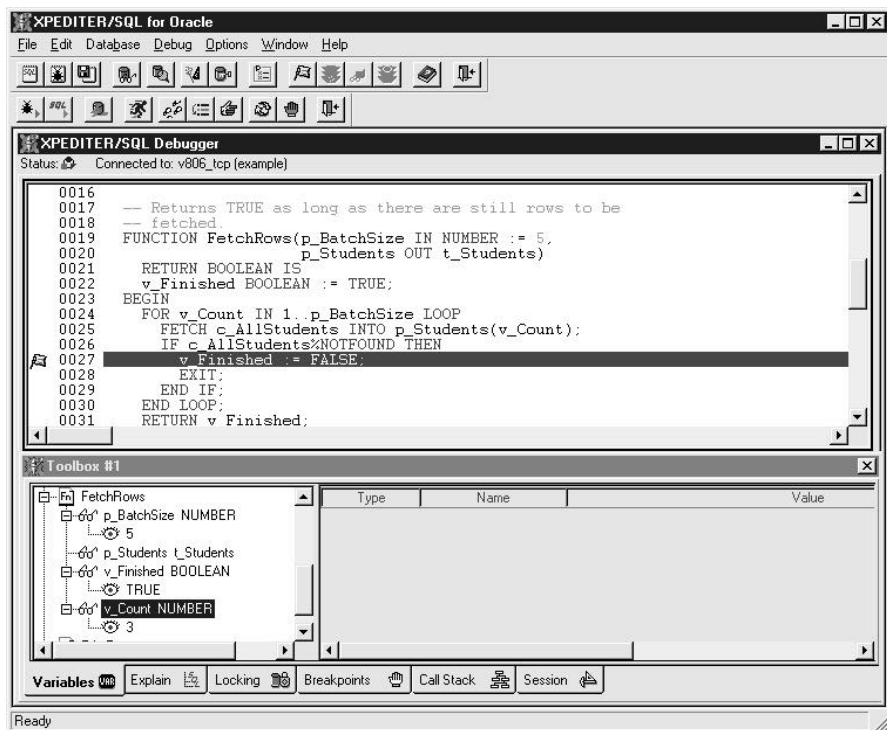


图3-10 停在断点的调试窗口

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 FetchRows 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 fetch 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。

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));

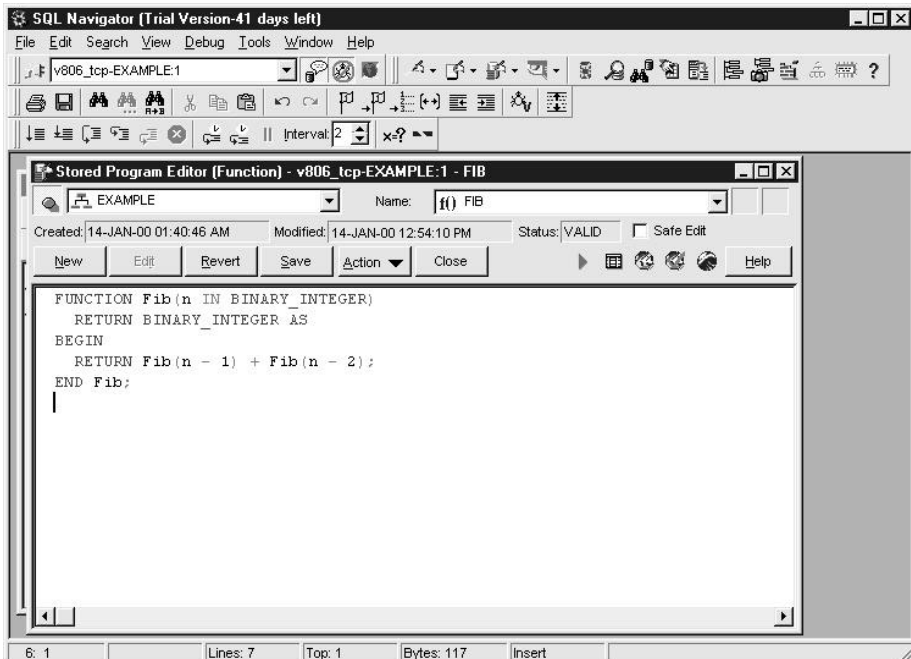


图3-11 准备直接调试函数Fib

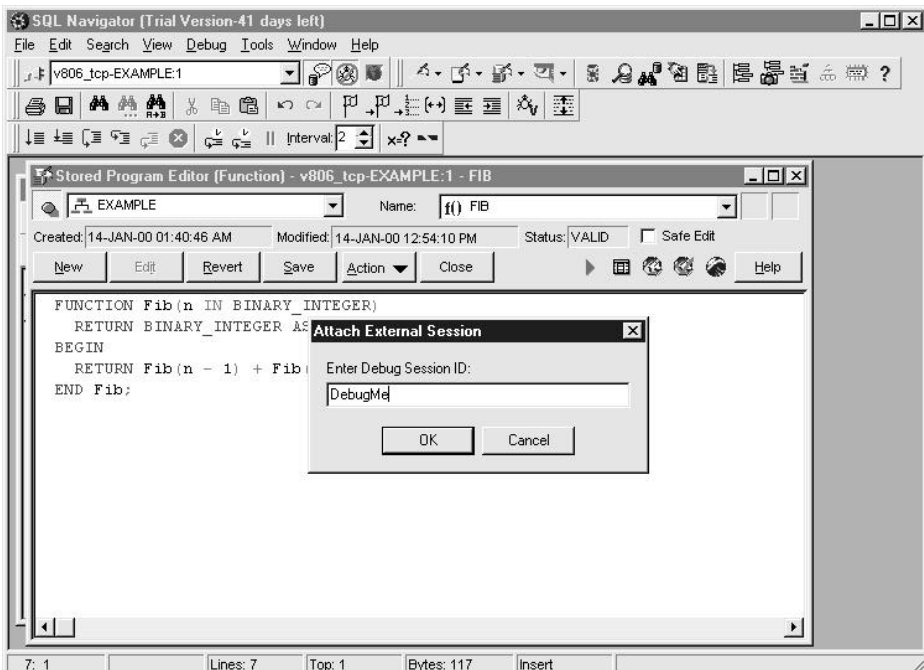


图3-12 连接会话会话框

现在,我们通过选择菜单项 **Debug | Attach External Session**来连接SQL Navigator的对话。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。

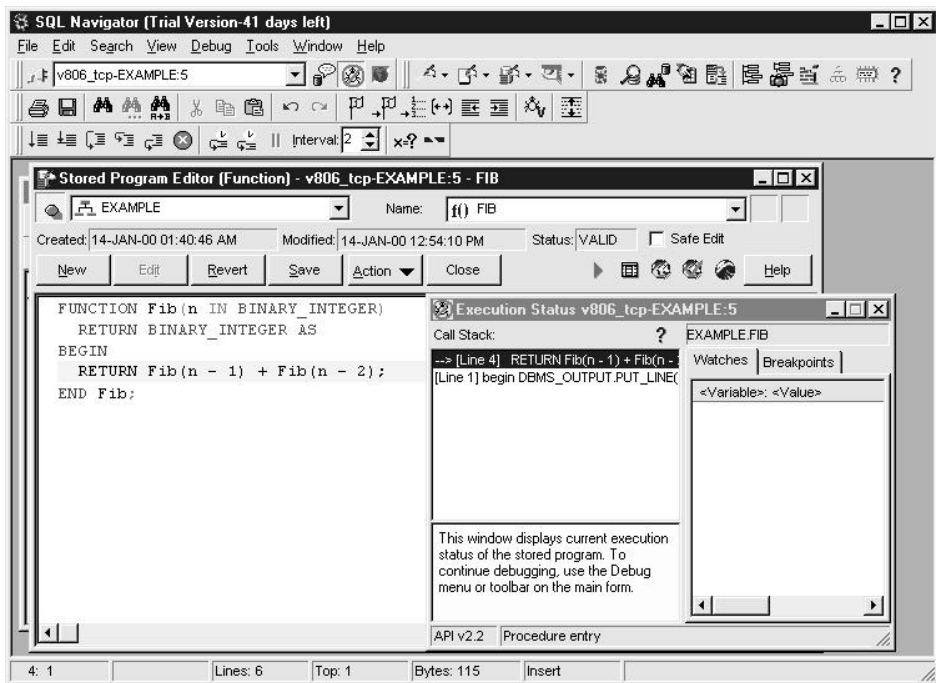


图3-13 调试函数Fib的窗口

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,我们可以发现出现的问题的原因。

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 Fib;

```

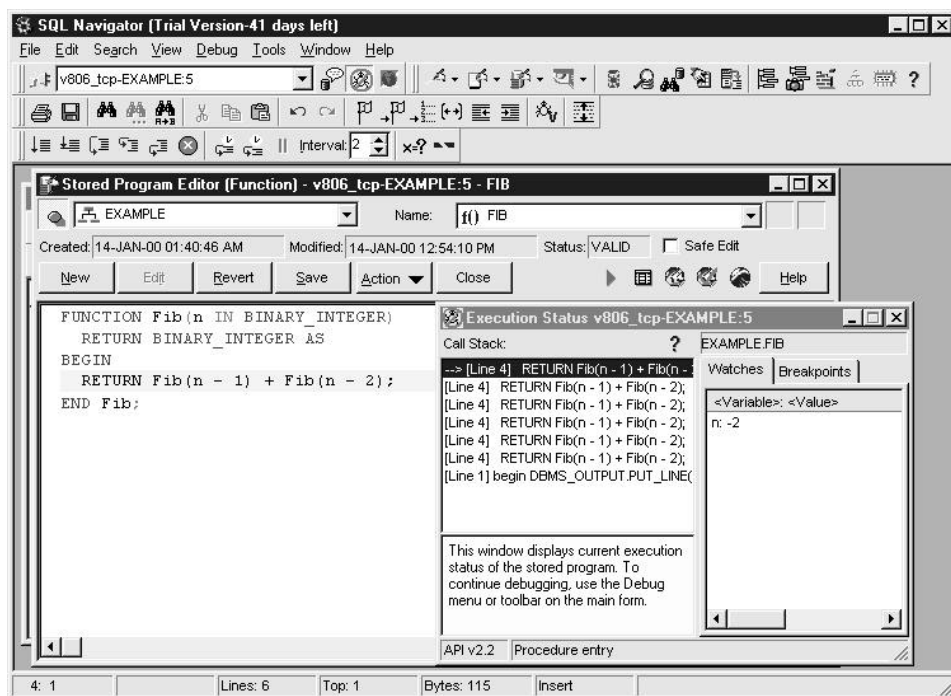


图3-14 变量的负值

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。

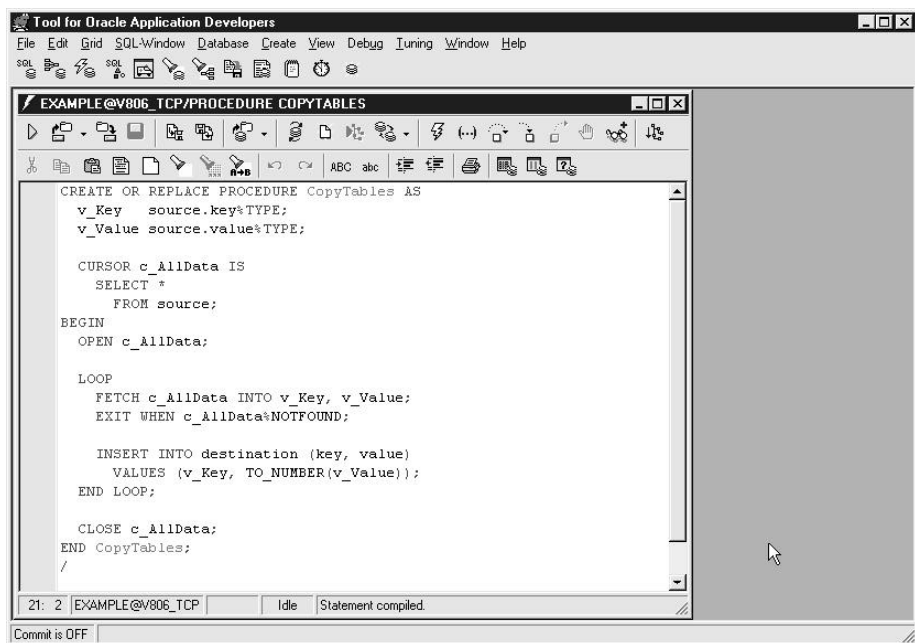


图3-15 显示Edit/Compile窗口中的CopyTables

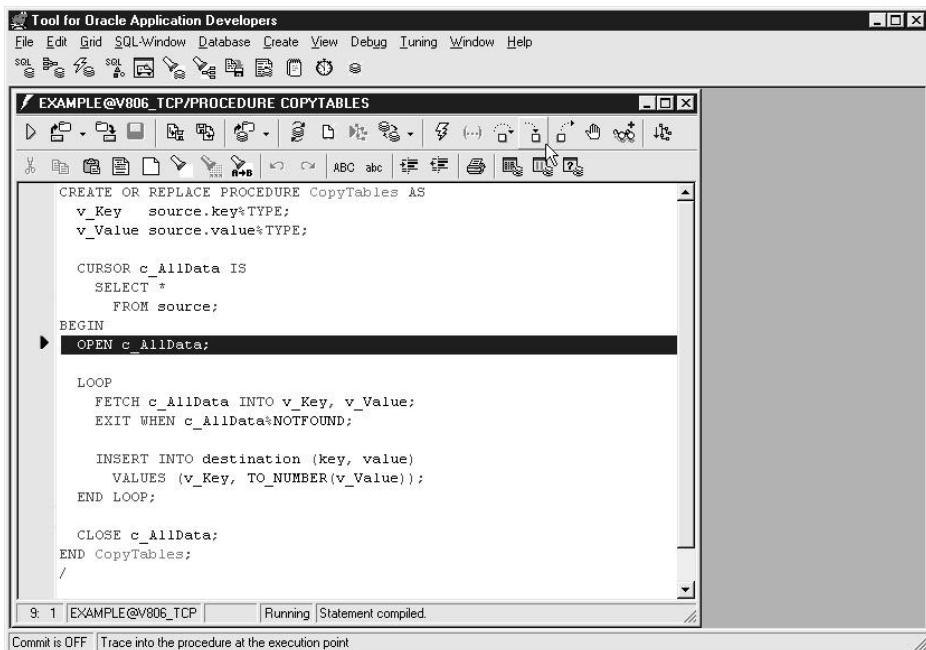


图3-16 停止在过程第一行的窗口

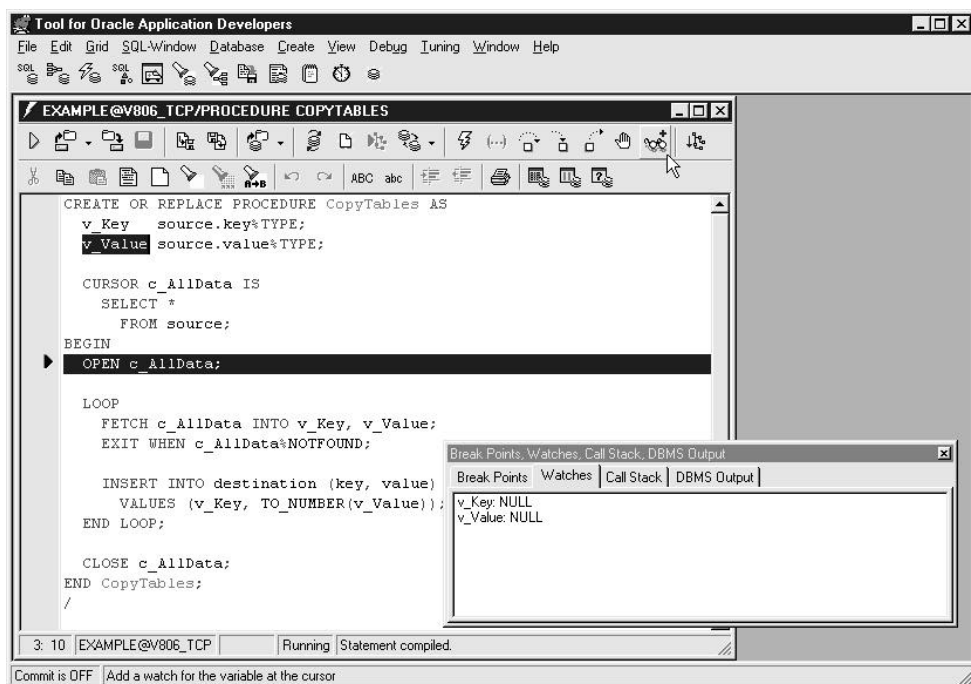


图3-17 设置观察点

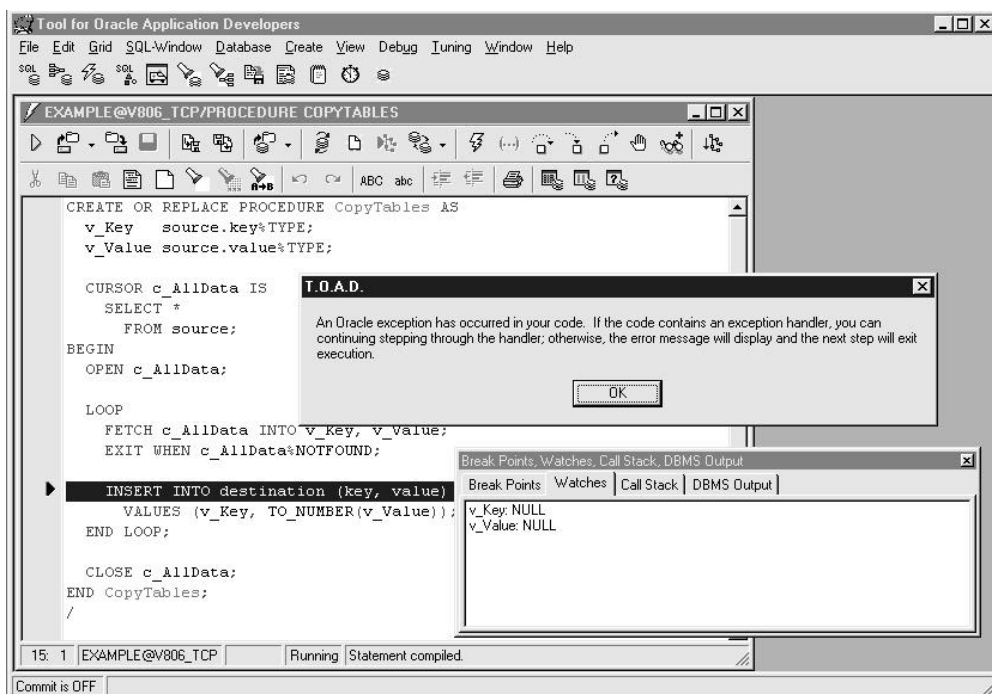


图3-18 停止在错误处的调试窗口

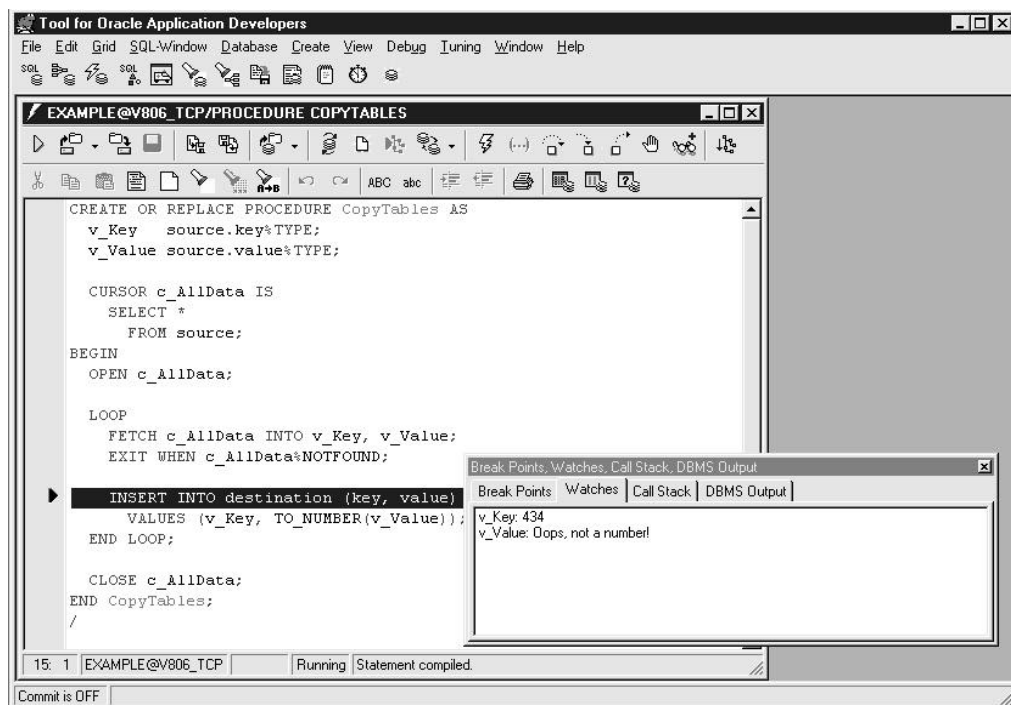


图3-19 有错误的数据

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。

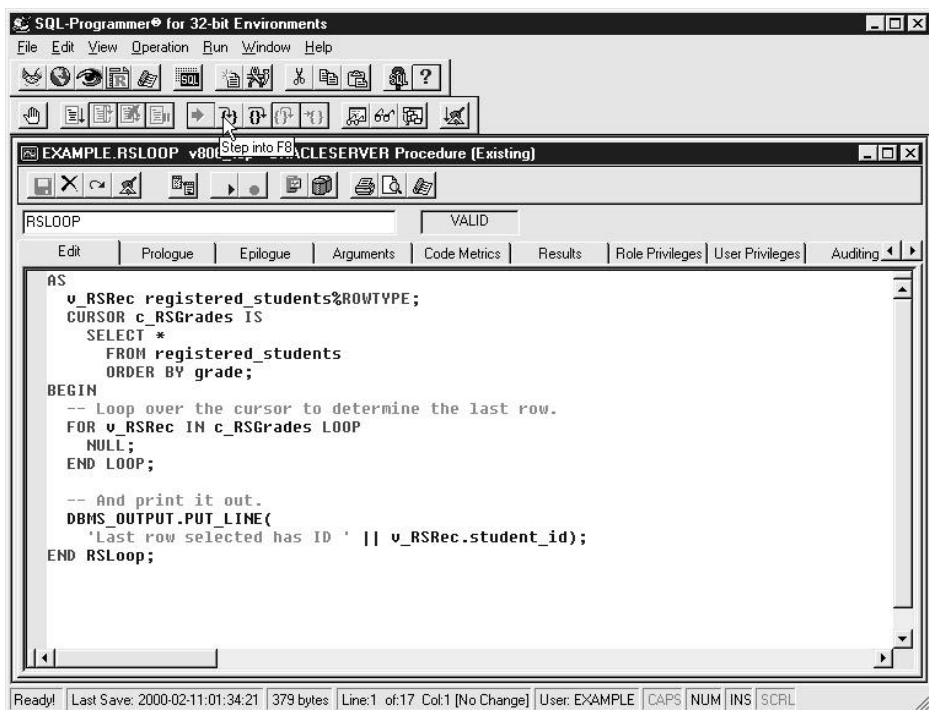


图3-20 准备调试过程RSLoop的窗口

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。

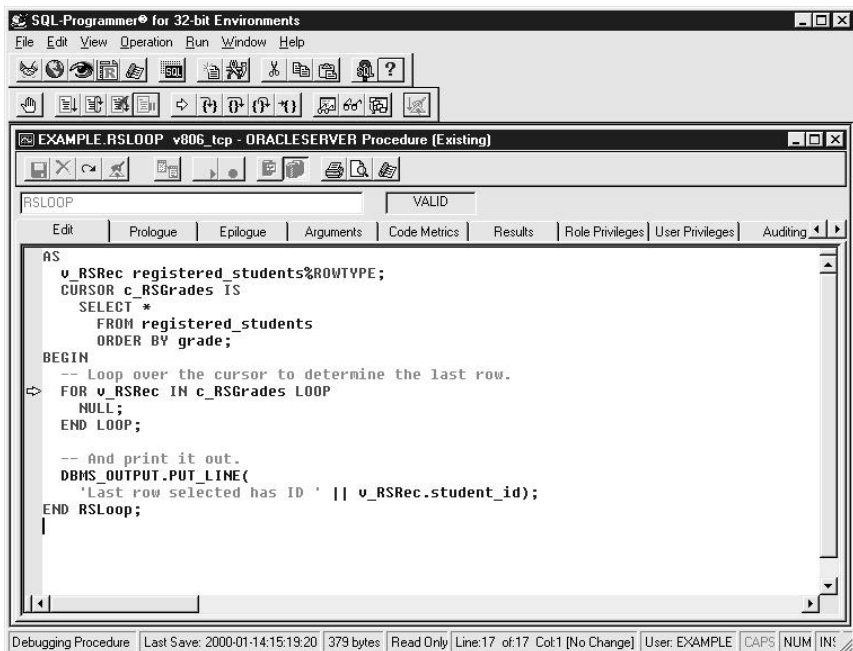


图3-21 停在过程代码第一行的调试窗口

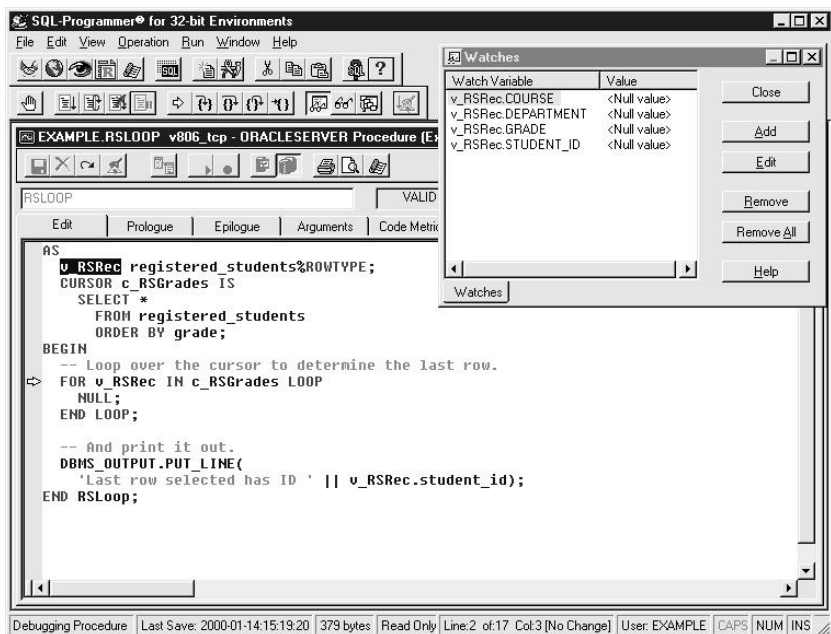


图3-22 观察变量v_RSRec的调试窗口

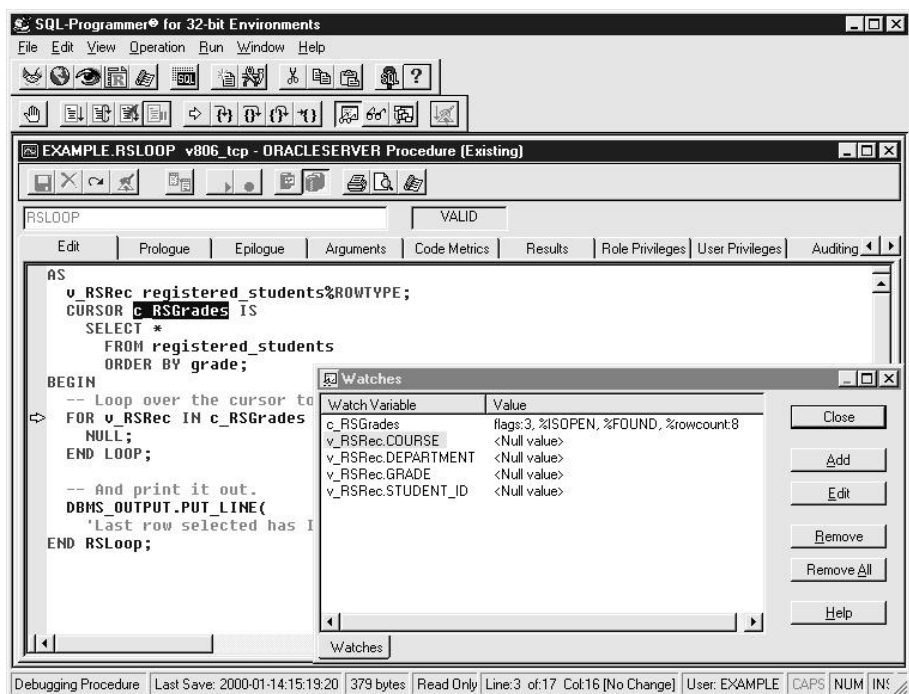


图3-23 显示v_RSRec仍然为空的观察窗口

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;
  -- 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 Oracle 7.3.4 版本中首次提供的；Oracle 数据库的 Oracle 8i 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 Oracle 8i 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;
    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 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新启动后，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一定的影响，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 errorsstack
```

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 Oracle 7 版本的 7.3.4 以及 Oracle 8 版的 8.0.5 和更高版本中使用。低于 Oracle 8 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“允许调用和异常”一节）。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，这就限制了堆栈的量长度。
- 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
```

Raiselt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 Raiselt。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; bf1: 20;
op: GBVAR; pos: 2; buf: 10bd7d8; len: 7; ind: 0; bf1: 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，也不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，而不能是编译错误。

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：

- _PLSQ_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_SSCALAR: 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_SSCALAR: FP+16, d=FP+80, n=FP+104
TC_SSCALAR: FP+24, d=FP+108, n=FP+132
TC_SSCALAR: FP+32, d=FP+136, n=FP+160
TC_SSCALAR: FP+40, d=FP+164, n=FP+188
TC_SSCALAR: 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 SOLT_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Raiselt入口。每次调用都显示源代码。

5) 单步进入 Raiselt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）
GBRVAR，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 oacfl2=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 oacfl2=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 oacfl2=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 oacfl2=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 oacfl2=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，`dt`、`mxl` 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
<code>bind</code>	界标位置（从 0 开始）
<code>dt</code>	数据类型，该类型对应应在头文件 <code>ocidfn.h</code> 中发现的 OCI 数据类型
<code>mxl</code>	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
<code>mal</code>	数据连接的数组长度
<code>scl</code>	比例
<code>pre</code>	精度
<code>oacflg,oacfl2</code>	内部标志
<code>size</code>	成组连接缓冲区
<code>offset</code>	成组连接缓冲区内部偏移量的总长度
<code>bfp</code>	连接缓冲区的地址
<code>bin</code>	连接缓冲区的长度
<code>avl</code>	实际变量长度
<code>flg</code>	内部标志
<code>value</code>	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 第一版 (8.1.5) 的 DBMS_TRACE

Oracle8i 第一版 (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 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8.1.6DBMS_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 BINARY_INTEGER:=8192);
```

由参数 limit 指定的记录将被保留下来 (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形缓冲区的容量)。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_VERSION_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 tracetable.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 `plssql_trace_runs` 中，该表结构如下所示：

列	数据类型	说明
<code>runid</code>	NUMBER	唯一的操作ID
<code>run_data</code>	DATE	操作开始的时间
<code>run_owner</code>	VARCHAR2(31)	已开始运行的数量
<code>run_comment</code>	VARCHAR2(2047)	用户提供的注释
<code>run_comment1</code>	VARCHAR2(2047)	附加的注释。注意， <code>COMMENT_PLSQL_TRACE</code> 将只修改 <code>run_comment</code> ，因此，该列必须由手工修改
<code>run_end</code>	DATE	操作结束的时间
<code>run_flags</code>	VARCHAR2(2047)	操作使用的标志
<code>related_run</code>	NUMBER	用于客户或服务端相关操作
<code>run_system_info</code>	VARCHAR2(2047)	没有启用
<code>spare1</code>	VARCHAR2(256)	没有启用

表 `plssql_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
<code>runid</code>	NUMBER	运行ID
<code>event_seq</code>	NUMBER	事件ID
<code>event_time</code>	DATE	该事件的时间
<code>related_event</code>	NUMBER	相关事件的ID
<code>event_kind</code>	VARCHAR2(31)	事件类型
<code>event_unit_dblink</code>	VARCHAR2(31)	当前库单元的数据库连接
<code>event_unit_owner</code>	VARCHAR2(31)	当前库单元的拥有者
<code>event_unit</code>	VARCHAR2(31)	当前库单元的名称
<code>event_unit_kind</code>	VARCHAR2(31)	当前库单元的类型
<code>event_line</code>	NUMBER	当前行
<code>event_proc_name</code>	VARCHAR2(31)	当前存在的过程名
<code>stack_depth</code>	NUMBER	当前栈深度
<code>proc_name</code>	VARCHAR2(31)	被调用过程的名称
<code>proc_dblink</code>	VARCHAR2(31)	被调用过程的数据库连接
<code>proc_owner</code>	VARCHAR2(31)	被调用过程的拥有者
<code>proc_unit</code>	VARCHAR2(31)	调用的库单元
<code>proc_unit_kind</code>	VARCHAR2(31)	调用过程的类型
<code>proc_line</code>	NUMBER	调用过程的行
<code>proc_params</code>	VARCHAR2 (2047)	过程参数
<code>icd_index</code>	NUMBER	调用PL/SQL内部程序的ICD索引
<code>user_excp</code>	NUMBER	用户定义的异常号
<code>excp</code>	NUMBER	预定义异常号
<code>event_comment</code>	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, 如快速 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 BINARY_INTEGER 的值来说明成功或失败, 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_commentIN VARCHAR2 := SYSDATE,
                        run_comment1IN VARCHAR2 := "",
                        run_number OUT BINARY_INTEGER)

RETURN BINARY_INTEGER;
PROCEDURE START_PROFILER( run_commentIN VARCHAR2 := SYSDATE,
                        run_comment1IN VARCHAR2 := "",
                        run_number OUT BINARY_INTEGER);
FUNCTION START_PROFILER( run_commentIN VARCHAR2 := SYSDATE,
                        run_comment1IN VARCHAR2 := "")

RETURN BINARY_INTEGER;
PROCEDURE START_PROFILER( run_commentIN VARCHAR2 := SYSDATE,
                        run_comment1IN 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 数据存储三个数据库表中，这些表可以用文件 profstab.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。

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	该行运行所需的最小时间
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_OUTOUT 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。