

第29章 使用存储子过程、包与提供的包

本章要点：

- 定义存储子过程与包
- 建立与使用存储程序
- 使用SHOW ERRORS调试
- 检查存储程序或包的状态
- 建立与使用包
- 关于Oracle 8i数据库提供的包
- 描述提供的包
- 开始学习Oracle 8i提供的包
- 使用Oracle 8i提供的包

29.1 定义存储子过程与包

当你着手创建存储子程序和包的时候，真正的 Oracle8i应用开发开始了，存储子程序和包是经过编译并存储在数据库中的永久程序代码模块。它们是你设计的可共享、可重入并且可重用的软件对象。可以使用支持远程过程调用的语言从其他的 PL/SQL模块、SQL语句以及客户端应用中调用它们。

当你编译了一个存储子程序或包时，程序的源代码、编译代码、编译状态以及全部编译错误存储到数据字典中。各种不同的数据字典视图可以帮助你查看这些项。使用 USER_视图，能够得到关于所编译模块的信息；使用 ALL_视图，能够得到其他人编译并授权你访问的相关模块限制信息；假如你拥有数据库系统管理员权限，使用 DBA_视图，可以得到任何人编译的全部信息。这些视图在表 29-1中列出（为了简便，作为 DBA_视图列出）。

表29-1 存储子程序和包的数据字典视图

视图名称	说明
DBA_SOURCE	所有编译模块的文本源代码
DBA_ERRORS	全部模块编译错误的文本列表清单
DBA_OBJECT_SIZE	编译模块的状态，例如有效性、源代码以及对象大小
DBA_OBJECTS	编译模块种类（存储过程、函数、包、包程序体）
DBA_DEPENDENCIES	对象相关性列表，例如在包中引用的表

不存在暴露对象代码的视图，因为根本不需要看到它。你要知道的一切就是它是不是在这里以及它是否是合法的。当一个相关对象被修改或删除时，例如向一个表添加一列或删除一个表时，一个已编译模块变为非法。如果一个存储模块变为非法，必须由服务器自动地重新编译它或由它的拥有者手工编译它。

29.2 建立与使用存储程序

创建存储子程序的语法与在匿名 PL/SQL 块中定义子程序的语法非常类似。除含一些额外增加的特性外，存储子程序拥有子程序的全部相同特性，这些特性你已经在前面学习编写过。让我们使用你先前看到的 `bool_to_char` 函数编写一个存储函数（参见清单 29-1）。

清单29-1 BOOL2CHR.SQL——编写重用代码存储子程序

```
CREATE OR REPLACE FUNCTION bool_to_char(Pbool IN BOOLEAN)
RETURN VARCHAR2 IS
    str VARCHAR2(5); -- capture string to return
BEGIN
    IF (Pbool) THEN -- test Boolean value for TRUE
        str := 'TRUE';
    ELSIF (NOT Pbool) THEN -- FALSE
        str := 'FALSE';
    ELSE -- must be NULL
        str := 'NULL';
    END IF; -- test Boolean value
    RETURN (str);
END bool_to_char;
/
```

服务器回答：

Function created.

这就是你所得到的。服务器并不执行程序，它仅编译程序，以便日后当你从其他的 PL/SQL 块中调用它时，你可以执行它。

提示 `CREATE OR REPLACE` 语法创建一个新函数或替换一个已有函数。这意味着不需要给存储代码增加重编译或改变的源代码，而是使用新版本完全代替它们。

警告 当使用 `CREATE OR REPLACE` 时，需要良好的源代码管理手段。当替换子程序时，老代码将永远从数据字典中消失。它还意味着在模式中只能含有一个拥有这个名字的对象。

现在，使用一个未命名 PL/SQL 块运行这个新创建的存储函数，如程序清单 29-2 中所示。

清单29-2 TESTBOOL.SQL——测试执行存储子程序

```
SET SERVEROUTPUT ON
BEGIN
    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line(bool_to_char(TRUE));
    DBMS_OUTPUT.put_line(bool_to_char(FALSE));
    DBMS_OUTPUT.put_line(bool_to_char(NULL));
END;
/
```

这个例子差不多测试了存储函数可能返回值的全部可能性。它被称为一个单元测试。对于每项输入，验证它的输出结果。输入值应该测试所有的临界状态（输入定义限制内或接近输入限制的值，包括限制之间的一些随机值）。应该给你的存储子程序准备单元测试文件，就像这个文件，那么你就可以检验并验证你的代码是否工作正确。将单元测试程序与存储子程序一起保留，以便当你修改子程序的时候，可以再次测试它。

在运行程序清单 29-2 后，服务器送回下列输出结果信息：

```
TRUE
FALSE
NULL
```

PL/SQL procedure successfully completed.

如果尝试传入一个不是布尔值的其他值，将会发生什么？试试看：

```
BEGIN
  DBMS_OUTPUT.put_line(bool_to_char(0));
END;
/
```

服务器响应下列信息：

```
ERROR at line 1:
ORA-06550: line 2, column 24:
PLS-00306: wrong number or types of arguments in call to 'BOOL_TO_CHAR'
ORA-06550: line 2, column 3:
PL/SQL: Statement ignored
```

错误消息表明这是一个编译错误。PL/SQL 引擎的强类型检查捕获这个错误并返回详尽的错误消息。

看一下另外一个略微长一些的存储过程例子，如程序清单 29-3 所示：

清单 29-3 SHOWINDX.SQL——显示表的索引信息的存储过程

```
CREATE OR REPLACE PROCEDURE show_index(Ptable IN all_indexes.table_name%TYPE
DEFAULT NULL) IS
  -- local cursors
  CURSOR show_index_cur(Ctable all_indexes.table_name%TYPE) IS
    SELECT
      table_owner, table_name, tablespace_name, index_name, uniqueness, status
    FROM all_indexes
    WHERE
      (Ctable IS NULL OR table_name = Ctable) -- one table or all
    ORDER BY
      table_owner, table_name, index_name;
  -- local constants
  TB CONSTANT VARCHAR2(1) := CHR(9); -- tab character
  -- local record variables
  show_index_rec show_index_cur%ROWTYPE; -- based on cursor
  old_index_info show_index_cur%ROWTYPE; -- used to detect control break
  -- local variables
  status NUMERIC;
  local_table all_indexes.table_name%TYPE;
BEGIN
  status := 0;
  local_table := UPPER(Ptable); -- make upper case
  old_index_info.table_owner := 'GARBAGE_OWNER'; -- initialize
  old_index_info.table_name := 'GARBAGE_TABLE';
  IF (local_table IS NULL) THEN -- one table or all?
    DBMS_OUTPUT.put_line('User ' || USER || ': Index Information for All
Tables');
  ELSE
    DBMS_OUTPUT.put_line('User ' || USER || ': Index Information for Table ' ||
local_table);
  END IF; -- one table or all?
  OPEN show_index_cur(local_table);
  LOOP -- get index information
    FETCH show_index_cur INTO show_index_rec;
    EXIT WHEN show_index_rec%NOTFOUND;
    IF (old_index_info.table_owner != show_index_rec.table_owner OR
      old_index_info.table_name != show_index_rec.table_name) THEN -- control
```

```

break
    DBMS_OUTPUT.put_line(TB); -- double spacing between tables
END IF;
DBMS_OUTPUT.put_line('Table Owner: ' || show_index_rec.table_owner || TB ||
    'Table: ' || show_index_rec.table_name);
DBMS_OUTPUT.put_line('Index: ' || show_index_rec.index_name || TB || ' in '
||
    show_index_rec.tablespace_name || TB ||
    show_index_rec.uniqueness || TB ||
show_index_rec.status);
old_index_info := show_index_rec; -- copy new values to old
END LOOP; -- get index information
CLOSE show_index_cur;
EXCEPTION
WHEN OTHERS THEN
BEGIN
    status := SQLCODE;
    DBMS_OUTPUT.put_line('show_index: ' || SQLERRM(status)); -- display error
message
    IF (show_index_cur%ISOPEN) THEN -- close any open cursors
        CLOSE show_index_cur;
    END IF;
EXCEPTION
WHEN OTHERS THEN
    NULL; -- don't care
END;
END show_index;
/

```

这一次，服务器返回下面信息：

Procedure created.

要执行这个过程，可以从一个匿名 PL/SQL 块中调用它，或者简单地使用 EXECUTE 命令单行调用它：

```
EXECUTE show_index('DEPT');
```

提示 EXECUTE 命令只能用于运行单行语句。如果语句包含两个或多个语句行，必须使用匿名 PL/SQL 块（使用 BEGIN...END）。如果将两条或多条语句塞入一行，仍旧可以使用 EXECUTE 命令。实际上，EXECUTE 命令将一行扩展为一个 PL/SQL 块，它仅仅是简写形式。

下面是作者的测试系统生成的输出结果信息：

```

User SYSTEM: Index Information for Table DEPT

Table Owner: SYSTEM      Table: DEPT
Index: DEPT_PRIMARY_KEY  in USER_DATA  UNIQUE  VALID

PL/SQL procedure successfully completed.

```

第一次运行这个存储过程时，当等待服务器将过程加载到内存中的时候，你或许注意到了一个略微的停顿。在继续进行的调用中，因为过程已经被加载到数据库的高速缓存，运行它的速度明显加快了。

这个特定的例子展示了一些良好的特性，它们是在你自己的存储过程中所需要的。你应该与局部变量声明的结构保持一致，并且以相同的顺序放置它们。还有，注意例外处理程序中的块，一定要关闭游标，以免一个错误把它设为打开状态。如果没有这样做而且遇到一个例外，下一次当运行这个存储过程的时候，将由于 CURSOR_ALREADY_OPEN 错误立即产生

故障。因为游标在会话内保持打开状态直到它被关闭或会话结束。为了清除打开的游标，你的用户将不得不重新连接数据库。

如果用户需要显示所有的表和它们的索引，只需要简单地删除单独输入参数（和括号），就像下面所示：

```
EXECUTE show_index ;
```

必须将自己放到用户的角度来设想用户希望采用哪种方式使用这个工具。更好的方法是询问用户。

29.2.1 从SQL调用存储过程

假设不满意Oracle8i提供的内建函数TO_NUMBER（）。你的抱怨可能是一个字符到数字的转换失败了，原因是字符串不能表示一个合法的数字，SQL失败并突然终止。你所希望的是，至少错误能够被准确地处理，那么就可以继续处理剩余的数据集了。尝试使用存储函数解决这个问题，如程序清单29-4中所示：

清单29-4 CHAR2NUM.SQL——字符到数字转换

```
CREATE OR REPLACE FUNCTION char_to_number(Pstr IN VARCHAR2, Pformat IN VARCHAR2
DEFAULT NULL)
RETURN NUMBER IS
BEGIN
    IF Pformat IS NULL THEN -- optional format not supplied
        RETURN (TO_NUMBER(Pstr));
    ELSE
        RETURN (TO_NUMBER(Pstr, Pformat)); -- format supplied
    END IF; -- test for optional format
EXCEPTION
WHEN OTHERS THEN -- unknown value
    RETURN (NULL);
END char_to_number;
/
```

可以采用两种方式运行这个存储函数：

从一个PL/SQL块。

从一条SQL语句。

首先，尝试从PL/SQL块中运行它，如程序清单29-5所示。

清单29-5 TESTC2N1.SQL——从PL/SQL块测试char_to_number（）函数

```
DECLARE
    v VARCHAR2(1) := 0;
    w VARCHAR2(10) := '999.999'; -- try a floating point number
    x VARCHAR2(11) := '+4294967295'; -- try a big positive number
    y CHAR(11) := '-4294967296'; -- try a big negative number
    z VARCHAR2(10) := 'garbage'; -- this is NOT a number!
BEGIN
    -- stored function returns NULL on error, so convert NULL to error message
    DBMS_OUTPUT.put_line(v || ' is ' || NVL(TO_CHAR(char_to_number(v)), 'NOT A
NUMBER!'));
    DBMS_OUTPUT.put_line(w || ' is ' || NVL(TO_CHAR(char_to_number(w)), 'NOT A
NUMBER!'));
    DBMS_OUTPUT.put_line(x || ' is ' || NVL(TO_CHAR(char_to_number(x)), 'NOT A
NUMBER!'));
    DBMS_OUTPUT.put_line(y || ' is ' || NVL(TO_CHAR(char_to_number(y)), 'NOT A
```

```
NUMBER!'));
  DBMS_OUTPUT.put_line(z || ' is ' || NVL(TO_CHAR(char_to_number(z)), 'NOT A
NUMBER!'));
END;
/
```

服务器返回下列信息：

```
0 is 0
999.999 is 999.999
+4294967295 is 4294967295
-4294967296 is -4294967296
garbage is NOT A NUMBER!

PL/SQL procedure successfully completed.
```

现在让我们在一条SQL语句中测试它，如程序清单 29-6所示。

清单29-6 TESTC2N2.SQL——从SQL运行char_to_number () 函数

```
SELECT '0' str,
       NVL(TO_CHAR(char_to_number('0')), ' IS NOT A NUMBER!') num FROM DUAL;
SELECT '999.999' str,
       NVL(TO_CHAR(char_to_number('999.999')), ' IS NOT A NUMBER!') num FROM DUAL;
SELECT '+4294967295' str,
       NVL(TO_CHAR(char_to_number('+4294967295')), ' IS NOT A NUMBER!') num FROM DUAL;
SELECT '-4294967296' str,
       NVL(TO_CHAR(char_to_number('-4294967296')), ' IS NOT A NUMBER!') num FROM DUAL;
SELECT 'garbage' str,
       NVL(TO_CHAR(char_to_number('garbage')), ' IS NOT A NUMBER!') num FROM DUAL;
```

得到与先前相同的结果。当你刚刚完成从一个字符串到数字的转换，就将它转换回字符串，看起来有些愚蠢，但是这样验证了存储函数的操作，尤其在上一个查询中。

29.2.2 从PL/SQL调用存储子过程

已经见到过如何从一个 PL/SQL块中激活一个存储子程序。它们有什么更高的使用价值呢？已经尽力构造一个有 14个表连结而成的查询了吗？我们可以使用其他手段取而代之，为每个主表创建一组单行查找程序，或许在连结中用到过主表的键值。然后，在一个 PL/SQL块中，只要写入用于驱动一个游标循环的最少量的表。在循环内部，执行单行查找以找出附带的数据值。使用这项技术，可以看到速度提高了 100%或更多。先前需要执行 20分钟的查询，使用这种技术可以在 2分钟内或更短的时间内完成。如果使用了某种类型的报表编写器来格式化输出结果，可以将数据写入临时表，然后临时表可以使用报表工具快速扫描。显然，这样做在某种程度上减慢了程序执行速度，但是它还是比使用大量、复杂的查询让 Oracle8i服务器陷入停顿要快得多。另外，它还易于验证输出结果的正确性。如果严格地测试单行查找程序，必须验证的唯一事实是足够小的查询。

现在考虑一下有多少次不得不编写一个复杂查询的外部连结代码。在 PL/SQL块中，测试单行查找没有发现数据并且跳转到循环的结束或继续循环是非常容易的事情。例如，一个申请处理程序也许含有一个循环，它看起来有点像这样：

```
LOOP -- process all claims for the period selected
  FETCH claims_cur INTO claims_rec;
  EXIT WHEN claims_cur%NOTFOUND;
```

```
-- get related table info
get_claimant_info(claims_rec.claimant_ssn, status);
get_provider_info(claims_rec.provider_id, status);
get_approval_info(claims_rec.approvedby, status);
IF (status != 0) THEN -- no approval on file!
    GOTO SKIPIT; -- skip processing claim
END IF;
... -- more single row lookups, claims processing
<<SKIPIT>> -- continue with next claim
END LOOP; -- process all claims for the period selected
```

本例中，单行查找的SQLCODE结果值返回变量status（状态）。为了让这个程序真正有用，需要将有关为什么申请没有被处理（没有被批准）的信息储存到某种类型的应用错误表中。更好的方法是，在查找程序的内部，可以将查找键值和错误代码存储到你设计的一个错误表中，那样，你可以在日后判定引起一个申请未被处理的子程序和键值。调用程序可以存储额外的数据（例如环境信息），可以更加容易地排除故障。

存储过程的另外一个用途就是实现商务规则。可以从触发子、客户程序或其他的 PL/SQL 程序中调用这些规则，如下例所示：

```
CREATE OR REPLACE TRIGGER check_approval_status
BEFORE INSERT ON claim_disbursal
DECLARE
    status NUMERIC;
BEGIN -- check for approval
    get_approval_info(:new.approvedby, status);
    IF (status != 0) THEN -- no approval on file!
        RAISE_APPLICATION_ERROR(-20100, 'No approval on file!');
    END IF;
END; -- check for approval
END check_approval_status;
```

这里，应用定义例外引起一个 INSERT 操作被回滚。一个存储过程定义商务规则，可以在许多地方使用该商务规则。如果需要修改规则，可以在存储过程的程序体中更改它。与它相关的 PL/SQL 代码仅仅需要被重新编译即可。

29.3 使用SHOW ERRORS调试

到目前为止，存储过程都是没有任何问题地编译通过。不幸的是，可以预料到会犯错误从而导致编译错误。幸好，调试错误的信息唾手可得。但是，Oracle8i 提供的用于观察它们的功能有一些使用限制，原因是它相对比较简单，但是我们有更好的解决方案。

当未命名 PL/SQL 块不能成功编译时，服务器将错误信息直接转储回 SQL*Plus。对于存储子过程，这个过程略有区别。在编译失败后，立即输入下列命令：

```
SHOW ERRORS
```

每个错误以及错误出现的行号都详尽地显示出来。

提示 SHOW ERRORS 命令仅显示最后一个存储子程序或包提交的编译错误。如果在一行中提交了两个子程序，SHOW ERRORS 命令只显示第二个子程序的错误信息（如果有的话）。但是，在 USER_ERRORS 中仍然可以同时获得两个子程序的错误信息。

请考虑程序清单 29-7 中的源代码，在过程之中隐藏了一个缺陷：

清单29-7 SHOWERR1.SQL——具有缺陷的错误格式化过程

```

CREATE OR REPLACE PROCEDURE showerr1(
  Pname  IN user_errors.name%TYPE,
  Ptype  IN user_errors.type%TYPE) IS
  CURSOR get_errors(Cname IN user_errors.name%TYPE,
                    CType  IN user_errors.type%TYPE) IS
    SELECT * FROM USER_ERRORS
    WHERE name = Cname AND type = CType
    ORDER BY SEQUENCE;
  get_errors_rec get_errors%TYPE;
  status NUMERIC := 0;
  Lname  user_errors.name%TYPE;
  Ltype  user_errors.type%TYPE;
BEGIN
  Lname := UPPER(Pname);
  Ltype := UPPER(Ptype);
  DBMS_OUTPUT.put_line('Compilation errors for ' || Lname);
  OPEN get_errors(Lname, Ltype);
  LOOP -- display all errors for this object
    FETCH get_errors INTO get_errors_rec;
    EXIT WHEN get_errors%NOTFOUND;
    DBMS_OUTPUT.put_line('At Line/Col: ' || get_errors_rec.line ||
                        ' ' || get_errors_rec.position);
    DBMS_OUTPUT.put_line(get_errors_rec.text);
  END LOOP; -- display all errors
  CLOSE get_errors;
  DBMS_OUTPUT.put_line('Errors Found: ' || TO_CHAR(status));
EXCEPTION
  WHEN OTHERS THEN
    BEGIN
      status := SQLCODE;
      IF (get_errors%ISOPEN) THEN -- cursor still open
        CLOSE get_errors;
      END IF;
    END;
END showerr1;
/

```

服务器返回下列错误消息：

Warning: Procedure created with compilation errors.

要得到更多信息，需要输入下列命令：

SHOW ERRORS

SHOW ERRORS命令的输出结果如下所示：

Errors for PROCEDURE SHOWERR1:

LINE/COL ERROR

```

-----
9/18      PLS-00206: %TYPE must be applied to a variable or column, not
          'GET_ERRORS'

9/18      PL/SQL: Item ignored
19/5      PL/SQL: SQL Statement ignored
19/27     PLS-00320: the declaration of the type of this expression is
          incomplete or malformed

21/5      PL/SQL: Statement ignored
21/45     PLS-00320: the declaration of the type of this expression is
          incomplete or malformed

23/5      PL/SQL: Statement ignored

```

```

23/26      PLS-00320: the declaration of the type of this expression is
           incomplete or malformed

26/3       PL/SQL: Statement ignored

26/52      PLS-00201: identifier 'ERRS' must be declared

```

注意 PL/SQL引擎清除存储源代码的空白行。如果为了程序的可读性，在程序中留有两倍行距，会发现存储在USER_SOURCE中的源代码变得很快与源代码不一样了，这使在最初源代码文件上进行的调试更为困难。

某些错误消息相当清楚，另外一些却非常含糊。还提供了错误出现的行号和列号。不幸的是，出现问题的源代码行没有显示出来。可以使用下面的 SQL语句显示源代码：

```

SELECT line, text
FROM user_source
WHERE name='SHOWERR1' AND line IN (9, 19, 21, 23, 26)
ORDER BY line;

```

源代码显示如下：

```

LINE TEXT
-----
 9  get_errors_rec get_errors%TYPE;
19  FETCH get_errors INTO get_errors_rec;
21  DBMS_OUTPUT.put_line('At Line/Col: ' || get_errors_rec.line ||
23  DBMS_OUTPUT.put_line(get_errors_rec.text);
26  DBMS_OUTPUT.put_line('Errors Found: ' || TO_CHAR(errs));

```

配合上面的错误消息，将看到下列信息：

在第9行上，必须基于get_errors%ROWTYPE定义记录变量，而不是%TYPE。

第19行错误的原因是get_errors_rec没有正确地定义。

第21行错误的原因是get_errors_rec没有正确地定义。

第23行错误的原因是get_errors_rec没有正确地定义。

在第26行上，你忘记了声明变量errs。

可以看到一个错误是怎样扩展到整个代码的。在声明中修改错误后，相关的错误全部消失了。

还有，要注意在这次编译中没有暴露的一些语法错误。例如，列命令 type应该大写并且应该用双引号括起来，因为 TYPE实际上是一个保留字。在上面的五个错误清除后，这个错误出现了（试一试）。

修改并提高的存储过程在程序清单 29-8中可以找到。

清单29-8 SHOWERR.SQL——最终版本，PL/SQL错误格式化过程

```

CREATE OR REPLACE PROCEDURE showerr(
  Pname IN user_errors.name%TYPE,
  Ptype IN user_errors."TYPE"%TYPE DEFAULT NULL) IS
  CURSOR get_errors(Cname IN user_errors.name%TYPE,
                    Ctype IN user_errors."TYPE"%TYPE) IS
  SELECT
    E.sequence, E.line, E.position, E.text err_text, S.text src_text
  FROM USER_ERRORS E, USER_SOURCE S
  WHERE
    E.name = Cname AND E."TYPE" = Ctype AND
    S.name = E.name AND S."TYPE" = E."TYPE" AND S.line = E.line
  ORDER BY E.sequence;

```

```

get_errors_rec get_errors%ROWTYPE;
status NUMERIC := 0;
Lname user_errors.name%TYPE;
Ltype user_errors.type%TYPE;
errs NUMERIC := 0; -- number of errors
cols NUMERIC; -- extra column padding needed for error position
wspc VARCHAR2(2) := CHR(20) || CHR(10); -- trailing whitespace
dspc VARCHAR2(1) := CHR(9); -- double spacing
BEGIN
  Lname := UPPER(Pname);
  IF (Ptype IS NOT NULL) THEN -- user supplied type
    Ltype := UPPER(Ptype);
  ELSE -- look for 1 object and get the type
    BEGIN
      SELECT object_type INTO Ltype
      FROM user_objects
      WHERE object_name = Lname;
    EXCEPTION
      WHEN OTHERS THEN
        RAISE_APPLICATION_ERROR(-20100, Lname ||
          ' type ambiguously defined');
    END;
  END IF; -- user supplied type
  DBMS_OUTPUT.put_line('Compilation errors for ' ||
    Ltype || ' ' || Lname);
  OPEN get_errors(Lname, Ltype);
  LOOP -- display all errors for this object
    FETCH get_errors INTO get_errors_rec;
    EXIT WHEN get_errors%NOTFOUND;
    IF (SUBSTR(get_errors_rec.err_text, 1, 4) = 'PL/S') THEN
      GOTO SKIPIT; -- ignore the 'PL/SQL: Statement ignored' messages
    END IF;
    cols := LENGTH(TO_CHAR(get_errors_rec.line)) + 3;
    DBMS_OUTPUT.put_line([' ' || TO_CHAR(get_errors_rec.line) ||
      ' ' || RTRIM(get_errors_rec.src_text, wspc)];
    DBMS_OUTPUT.put_line(LPAD('^', get_errors_rec.position + cols, '-'));
    DBMS_OUTPUT.put_line([' ' || TO_CHAR(get_errors_rec.line) ||
      ' ' || get_errors_rec.err_text]);
    DBMS_OUTPUT.put_line(dspc); -- double space
    errs := errs + 1;
    <<SKIPIT>>
    NULL;
  END LOOP; -- display all errors
  CLOSE get_errors;
  DBMS_OUTPUT.put_line('Errors Found: ' || TO_CHAR(errs));
EXCEPTION
  WHEN OTHERS THEN
    BEGIN
      status := SQLCODE;
      IF (get_errors%ISOPEN) THEN -- cursor still open
        CLOSE get_errors;
      END IF;
      DBMS_OUTPUT.put_line('showerrs: ' || SQLERRM(status));
    EXCEPTION
      WHEN OTHERS THEN
        NULL; -- don't care
    END;
END showerr;
/

```

通过执行新的过程，得到的结果更为清晰、更为详细并且更易于阅读：

```

execute showerr('showerr1');
Compilation errors for PROCEDURE SHOWERR1
-----

```

```
[9]  get_errors_rec get_errors%TYPE;
-----^
[9] PLS-00206: %TYPE must be applied to a variable or column,
not 'GET_ERRORS'

[19]  FETCH get_errors INTO get_errors_rec;
-----^
[19] PLS-00320: the declaration of the type of this expression is
incomplete or malformed

[21]  DBMS_OUTPUT.put_line('At Line/Col: ' || get_errors_rec.line ||
-----^
[21] PLS-00320: the declaration of the type of this expression is
incomplete or malformed

[23]  DBMS_OUTPUT.put_line(get_errors_rec.text);
-----^
[23] PLS-00320: the declaration of the type of this expression is
incomplete or malformed

[26]  DBMS_OUTPUT.put_line('Errors Found: ' || TO_CHAR(errs));
-----^
[26] PLS-00201: identifier 'ERRS' must be declared

Errors Found: 5

PL/SQL procedure successfully completed.
```

你会更喜欢这样。或许你想控制出现问题行前后的源代码打印行数，以便能够得到相关的上下文。我把这个问题作为一个练习留给你来解决。

29.4 检查存储程序或包的状态

执行下面的SQL语句查看你的全部存储子程序的状态：

```
COLUMN object_name FORMAT A30
COLUMN timestamp FORMAT A20
SELECT object_name, object_type, timestamp, status
FROM user_objects
WHERE
    object_type IN ('FUNCTION', 'PROCEDURE', 'PACKAGE', 'PACKAGE BODY')
ORDER BY object_name, object_type;
```

这条命令的输出结果，在我的系统上如下所示：

OBJECT_NAME	OBJECT_TYPE	TIMESTAMP	STATUS
CHAR_TO_BOOL	FUNCTION	1997-11-01:16:06:43	VALID
CHAR_TO_NUMBER	FUNCTION	1997-11-01:13:34:26	VALID
SHOWERR	PROCEDURE	1997-11-05:06:01:31	VALID
SHOWERRS	PROCEDURE	1997-11-05:05:47:10	VALID
SHOWERR1	PROCEDURE	1997-11-04:19:51:31	INVALID
SHOW_INDEX	PROCEDURE	1997-11-01:13:17:37	VALID

6 rows selected.

注意过程SHOWERR1存在，即使它不能成功地编译。它确实在那里，源代码和它的所有内容。但是如果试图运行它的话，会得到一个错误。

提示 表结构的更改有时候会使存储过程和对象无效。可以使用ALTER...COMPILE命令重新编译指定的数据库对象，如下面的例子所示：

```
ALTER PROCEDURE SHOWERRS COMPILE;
```

另外一个要查看的有趣事情是代码统计。要查看这些信息，运行下面的语句：

```
SELECT name, "TYPE", source_size, parsed_size, code_size, error_size
FROM user_object_size
WHERE
  "TYPE" IN ('FUNCTION', 'PROCEDURE', 'PACKAGE', 'PACKAGE BODY')
ORDER BY name, "TYPE";
```

在我的系统上，这条语句的输出结果看起来如下所示：

NAME	TYPE	SOURCE_SIZE	PARSED_SIZE	CODE_SIZE	ERROR_SIZE
CHAR_TO_BOOL	FUNCTION	391	841	546	0
CHAR_TO_NUMBER	FUNCTION	375	575	384	0
SHOWERR	PROCEDURE	2587	5725	3598	0
SHOWERRS	PROCEDURE	2227	5035	3073	0
SHOWERR1	PROCEDURE	1149	0	0	495
SHOW_INDEX	PROCEDURE	2408	4221	2629	0

6 rows selected.

注意，因为没有能够成功编译，SHOWERR1没有语法分析或代码大小。有趣的是，代码和语法分析的大小大于源代码。在源代码大小与语法分析及代码大小之间的确没有太多的关联。代码分析的大小取决于游标、隐式 SQL、子程序等对象的数目——你可能会获得更多存储的事物。代码大小使你具有了每个对象在 SGA 中占用多少内存的概念。

提示 使用 DESCRIBE 命令，能够得到特定存储子程序参数列表的方便快捷的视图。

```
DESC SHOWERR
PROCEDURE SHOWERR
Argument Name          Type                      In/Out Default?
-----
PNAME                  VARCHAR2(30)             IN
PTYPE                  VARCHAR2(12)             IN      DEFAULT
```

29.5 建立与使用包

包是程序对象的集合，可以提供数据、游标和其他语言结构的连续性。当引用包里的任意对象时，整个包便被加载到内存中。虽然这看起来像是一个大的磁盘需求，但要记住当包被加载后，代码对所有的用户可用。只要有任意一个用户引用这个包，它就保留在 SGA 区中，当前在 SGA 区里面的全部程序代码都是可用的。另外一种做法是像加载存储子程序那样分别加载每段程序代码。因为被加载后，代码是可以重入的，所以对于全体用户都可用。代码重入弥补了第一次磁盘命中所造成的系统开销。从服务器视图的观点来说，这样非常有效，既节省了内存的使用又减少了磁盘存取。

包还允许子程序重载，这个特性对于纯粹的存储子程序不可用。使用重载，同一个子程序名称可以在不同组合及数目的参数类型中被重复使用。如果观察 STANDARD.SQL，可以看到 TO_CHAR、TO_DATE 和 TO_NUMBER 的几个声明，每种声明都带有不同数据类型的参数。

29.5.1 包声明与包体

PL/SQL 包由两个部分组成：

包声明（也被称为头）。

包体。

包体包含包的声明部分中提到的全部子程序的实际代码。

包声明含有希望展示给其他人的全部语言结构，它们可以是变量、用户定义数据类型、游标以及子程序，放置在声明中的任何对象都是可全局访问的，这里没有可执行代码，只有子程序的声明。程序清单 29-9 展示了一个包头的例子，其中使用了几个你已经见过的函数。

清单 29-9 LIBHDR.SQL——一个提议的库包的声明

```
CREATE OR REPLACE PACKAGE lib IS
-- public global user defined types
-- public global constants
MAXINT CONSTANT BINARY_INTEGER := +2147483647; -- +(2^31 - 1)
MININT CONSTANT BINARY_INTEGER := -2147483647; -- -(2^31 - 1)
MAXDATE CONSTANT DATE := TO_DATE('31-DEC-4712 AD', 'DD-MON-YYYY AD');
MINDATE CONSTANT DATE := TO_DATE('01-JAN-4712 BC', 'DD-MON-YYYY BC');
-- public global cursors
-- public global variables
-- public global subprograms
-- convert a BOOLEAN to a STRING ('TRUE', 'FALSE', 'NULL')
FUNCTION bool_to_char(Pbool IN BOOLEAN)
RETURN VARCHAR2;
-- convert a STRING to BOOLEAN (TRUE, FALSE, NULL)
FUNCTION char_to_bool(Pstr IN VARCHAR2) RETURN BOOLEAN;
-- safe STRING to NUMBER conversion; returns NUMBER or NULL
FUNCTION char_to_number(Pstr IN VARCHAR2, Pformat IN VARCHAR2 DEFAULT NULL)
RETURN NUMBER;
-- debug flag interface routines
PROCEDURE debug_on;      -- turns debug on
PROCEDURE debug_off;     -- turns debug off
PROCEDURE debug_toggle;  -- toggles debug on/off
FUNCTION debug_status    -- is debug on/off?
RETURN BOOLEAN;
END lib;
/
```

注意在其他的存储子程序对象中见到的标准的 CREATE OR REPLACE 语法。现在，不必像创建单个子程序那样显式地创建子程序，而使用以下方法代替，当子程序封装在块的内部时，就像原先那样声明它们，除了在包头中。只要定义一个声明，包括子程序的名称、参数（如果有的话）以及返回类型（如果有的话）。还有，注意子程序是如何遵循全部其他声明的。

那些熟悉 C 编程语言的人将认出这种函数原形。既然声明存在，不必开发这些程序的实际代码，只需要修改它们——不违背定义。只要声明可用，开发者可以立即着手应用的开发，即使程序体还没有编写。由于子程序声明和实现是分开的，如果仅仅代码升级了，使用这些例程的程序就不需要重新编译。只有当声明部分改变的时候，相关模块才需要重新编译。

注意 包编程的标准惯例是将包头与包体分别存放在不同的文件中。使用这种方式，如果仅仅是代码发生变化（最可能的情况是当包还比较新但仍然处于不断变化的状态时），仅需要重新编译包体。

使用所希望的任意标准化命名约定。有些人喜欢给包头和包体使用不同的扩展名（*.hdr 和 *.pkg），还有些人喜欢总是给包头和包体使用 *.sql 扩展名，但还是要指出包的内容（*.hdr.sql、*.pkg.sql）。

包体含有声明中显示的子程序的实际代码，以及私有定义子程序、变量、用户定义数据

类型以及游标。这些对象隐藏在最下面。包体还可以包含初始化代码，它们只在包被首次引用时执行一次，虽然这样做看起来限制了包的使用，但你可以使用它来激活输出、设置各种不同的应用标志（类似于调试开/关）、以及收集全局用户相关数据。虽然包体与包头是有所区别的对象，但在包头被成功地编译之后，包体才能够进行成功地编译。

这个对象在声明中被称为包体，程序被声明得好像存在于一个匿名块中。还要注意包体的名字必须与包头名字相同。头中声明的每个程序在包体中有相应的实现过程。

另外需要注意的事情是在它们的声明中初始化变量。当包第一次被调用的时候，这样的初始化只进行一次。这是因为这些变量只被分配并初始化一次，然后在整个会话间保持不变。如果我在包的声明中重新设置调试标志而节省了几毫秒，在包的初始化中，就需要更明显的并且更多的维护来做这件事情。

29.5.2 比较公有声明与私有声明

在包声明中声明的任何对象在使用范围上都是全局的。也就是说，在当前的会话内，任何PL/SQL程序可以调用并使用声明中定义的变量、游标和子程序。相反，只在包体中声明的变量、游标以及子程序，虽然对于包体自身来说是全局的，但对于外部世界是隐藏的。在两种情况下，变量和游标都是不变的，它们保存当前值，直到从内存中消失或会话终止。

我对使用全局变量的建议是尽量少用全局变量，或根本不用。还有，如果将他们隐藏在包体中，在控制程序访问它们时，将得到全部变量保持不变的好处。通过创建全局可用的子程序来管理它们，可以处理这个访问。

调试接口程序表明这样一个标准惯例，不要将一个变量本身暴露给外部世界，而是提供操作这个变量的子程序。例如，你不希望开发者直接修改和检查变量 `debug_flag` 的值，如果这个调试标志是公共和全局的，任何程序就能够使用带有潜在危害或者至少不希望的结果更改它。如果将来需要添加一些东西或更改调试功能实现的手段，对开发者隐藏具体的实现细节。这是面向对象编程的常用方法。

除了常量和各种表的记录变量，我很少使用全局变量。我还喜欢将声明以一个标准的顺序放置，如清单 29-10 中所示。这样一来，我总是能够精确地知道到哪里去查找需要的东西。

清单 29-10 LIBPKG.SQL——前面声明中的包体

```
CREATE OR REPLACE PACKAGE BODY lib IS
  -- private global user defined types
  -- private global constants
  -- private global cursors
  -- private global variables
  debug_flag BOOLEAN;
  user_name VARCHAR(30); -- application user's login ID
  appl_name VARCHAR2(30); -- application currently running
  modl_name VARCHAR2(30); -- module (within app) currently running
  context_pt VARCHAR2(30); -- check-in point within module
  run_date DATE; -- when an application was launched
  -- private global subprograms
  -- implementation of exposed subprograms
  -- convert a BOOLEAN to a STRING ('TRUE', 'FALSE', 'NULL')
  FUNCTION bool_to_char(Pbool IN BOOLEAN)
  RETURN VARCHAR2 IS
    str VARCHAR2(5); -- capture string to return
  BEGIN
    IF (Pbool) THEN -- test Boolean value for TRUE
```

```

    str := 'TRUE';
    ELSIF (NOT Pbool) THEN -- FALSE
        str := 'FALSE';
    ELSE -- must be NULL
        str := 'NULL';
    END IF; -- test Boolean value
    RETURN (str);
END bool_to_char;
-- convert a STRING to BOOLEAN (TRUE, FALSE, NULL)
FUNCTION char_to_bool(Pstr IN VARCHAR2) RETURN BOOLEAN IS
    Lstr VARCHAR2(32767); -- max string length
    Lbool BOOLEAN := NULL; -- local Boolean value (default)
BEGIN
    Lstr := UPPER(LTRIM(RTRIM(Pstr))); -- remove leading/trailing spaces,
uppercase
    IF (Lstr = 'TRUE') THEN
        Lbool := TRUE;
    ELSIF (Lstr = 'FALSE') THEN
        Lbool := FALSE;
    END IF;
    RETURN(Lbool);
END char_to_bool;
-- safe STRING to NUMBER conversion; returns NUMBER or NULL
FUNCTION char_to_number(Pstr IN VARCHAR2, Pformat IN VARCHAR2 DEFAULT NULL)
RETURN NUMBER IS
BEGIN
    IF Pformat IS NULL THEN -- optional format not supplied
        RETURN (TO_NUMBER(Pstr));
    ELSE
        RETURN (TO_NUMBER(Pstr, Pformat)); -- format supplied
    END IF; -- test for optional format
EXCEPTION
WHEN OTHERS THEN -- unknown value
    RETURN (NULL);
END char_to_number;
-- debug flag interface routines
PROCEDURE debug_on IS -- turns debug on
BEGIN
    debug_flag := TRUE;
END debug_on;
PROCEDURE debug_off IS -- turns debug off
BEGIN
    debug_flag := FALSE;
END debug_off;
PROCEDURE debug_toggle IS -- toggles debug on/off
BEGIN
    debug_flag := XOR(debug_flag, TRUE);
END debug_toggle;
FUNCTION debug_status -- is debug on/off?
RETURN BOOLEAN IS
BEGIN
    RETURN(debug_flag);
END debug_status;
BEGIN -- optional package initialization
    DBMS_OUTPUT.enable(1000000);
    debug_flag := FALSE; -- reset
END lib;
/

```

29.5.3 引用包元素

使用点符号，包外部的PL/SQL代码可以引用包的内容，如清单 29-11中所示。记住，点符

号被用来消除一个对象使用范围上所有含义模糊的地方。为了引用一个包装的对象，还必须提供包的名称，采用以下格式‘包名字.包对象’。如果引用对象的代码在同一个范围内，例如在同一个包内，则不需要使用限定符。

清单29-11 TESTLIB.SQL——测试提议的库包的执行情况

```
BEGIN
  DBMS_OUTPUT.put_line('DEBUG status is initially: ' ||
                        LIB.bool_to_char(LIB.debug_status));
  LIB.debug_on;
  DBMS_OUTPUT.put_line('DEBUG status is now: ' ||
                        LIB.bool_to_char(LIB.debug_status));
  LIB.debug_off;
  DBMS_OUTPUT.put_line('DEBUG status is now: ' ||
                        LIB.bool_to_char(LIB.debug_status));
  LIB.debug_toggle;
  DBMS_OUTPUT.put_line('DEBUG status is finally: ' ||
                        LIB.bool_to_char(LIB.debug_status));
END;
/
```

要使其成为一个真正的测试，首先彻底退出你的 SQL*Plus会话，然后重新进入，这样可以确保会话的确是新的。然后，输入命令 SET SERVER OUTPUT ON，并运行程序清单 29.11 中所指的未命名块。注意该块没有打开输出，这点小事在包的初始化代码中执行。调试标志正常情况下默认值为 NULL，也被初始复位了。服务器返回下列信息：

```
DEBUG status is initially: FALSE
DEBUG status is now: TRUE
DEBUG status is now: FALSE
DEBUG status is finally: TRUE

PL/SQL procedure successfully completed.
```

现在再次运行它。

```
DEBUG status is initially: TRUE
DEBUG status is now: TRUE
DEBUG status is now: FALSE
DEBUG status is finally: TRUE

PL/SQL procedure successfully completed.
```

注意，现在调试标志的初始值为 TRUE，这个值从一个程序传递到下一个程序。如果重新连接，一定会拥有一个新的会话，并且再次执行包的初始化。记住，代码静态地存在内存中，但是每位用户得到一个全新的局部私有变量集。运行第二个会话并验证它的行为。

29.6 Oracle 8i数据库提供的包

除了给予 Oracle 8i服务器基本功能的 STANDARD包，例如基本子类声明和数据类型转换程序外，还有一组专供数据库系统管理员和开发者使用的包。这些包根据它们的名字有所区别，它们使用 DBMS_或 UTL_ 开头，表示它们与数据库进行交互或提供常规目的的应用。首先，快速浏览这些包并在稍后更为严密地检查它们。

29.6.1 在服务器内交互

大多数 Oracle 8i提供的包在服务器环境内部处理数据：数据字典项、用户数据库对象或在

系统全局区（System Global Area）中发现的单独的对象，像共享内存缓冲池等。使用它们，可以管理快照、重新编译程序单元、当数据库项目改变时产生异步警报、运行作业等等。

大部分这样的数据库包与内建在数据库应用中的功能交互，或者通过调用可加载外部模块例如DLL来扩展数据库环境。既不希望也不需要理解或使用这些模块的入口点。我宁愿明确地告诉你不要尝试直接使用它们，一定要通过 PL/SQL 编程接口提供的手段来使用它们。

29.6.2 在服务器以外交互

一些包对调用环境给出反馈信息，或者通过其他手段给外部处理进程提供一个接口。例如 Oracle 8i 的管道特性，是专门用于会话之间通信的一种方法，它被严格地限定为只在内存中——没有数据库对象用作中间存储介质。另外，当在 SQL*Plus 中运行 PL/SQL 程序时，DBMS_OUTPUT 包使打印语句能够显示。

29.6.3 从服务器获取更多的信息

有几个包装例程可以令你获得你的数据库的附加调整信息，例如共享缓冲池的使用、段空间信息、运行追踪、获得常规数据库信息等等。在习惯使用它们后，它们都会变成工具箱中的标准部件。

29.7 描述提供的包

表29-2是一些重要的 Oracle 8i 提供的包的一个方便、快捷参考，它还是关于提供的包的内容的一个简明描述说明。

表29-2 提供的包小结

包 名 称	包头文件	说 明
DBMS_ALERT	dbmsalrt.sql	异步处理数据库事件
DBMS_APPLICATION_INFO	dbmsutil.sql	注册当前运行的应用的名称（用于性能监控）
DBMS_AQADM	dbmsaqad.sql	与高级队列选项一起使用
DBMS_DDL	dbmsutil.sql	重新编译存储过程程序和包，分析数据库对象
DBMS_DEBUG	dbmspb.sql	PL/SQL 调试器接口
DBMS_DEFER	dbmsdefr.sql	远程过程调用应用的用户接口
DBMS_DESCRIBE	dbmsdesc.sql	说明存储过程的参数
DBMS_JOB	dbmsjob.sql	按指定的时间或间隔执行用户定义的作业
DBMS_LOCK	dbmslock.sql	管理数据库块
DBMS_OUTPUT	dbmsotpt.sql	将文本行写入内存，供以后提取和显示
DBMS_PIPE	dbmspipe.sql	通过内存“管道”在会话之间发送并接收数据
DBMS_PROFILER	dbmspbp.sql	用于配置 PL/SQL 脚本以鉴别瓶颈问题
DBMS_REFRESH	dbmssnap.sql	管理能够被同步刷新的快照组
DBMS_SESSION	dbmsutil.sql	程序地执行 Alter Session（改变会话）语句
DBMS_SHARED_POOL	dbmspool.sql	查看并管理共享池内容
DBMS_SNAPSHOT	dbmssnap.sql	刷新、管理快照，并清除快照日志
DBMS_SPACE	dbmsutil.sql	获取段空间信息
DBMS_SQL	dbmssql.sql	执行动态 SQL 和 PL/SQL
DBMS_SYSTEM	dbmsutil.sql	开/关给定会话的 SQL 追踪
DBMS_TRANSACTION	dbmsutil.sql	管理 SQL 事务

(续)

包 名 称	包头文件	说 明
DBMS_UTILITY	dbmsutil.sql	多种实用工具：对于一个给定的模式，重新编译存储子程序和包、分析数据库对象、格式化错误信息并调用堆栈用于显示、显示实例是否以并行服务器模式运行、以10毫秒间隔获取当前时间、决定数据库对象的全名、将一个PL/SQL表转换为一个使用逗号分割的字符串或卷、获取数据库版本/操作系统字符串
UTL_RAW	utlraw.sql	RAW数据类型的字符串函数
UTL_FILE	utlfile.sql	读/写基于ASCII字符的操作系统文件
UTL_HTTP	utlhttp.sql	从给定的URL得到HTML格式的主页
DBMS_LOB	dbmslob.sql	管理巨型对象

29.8 开始学习Oracle 8i提供的包

除了与Oracle 8i服务器一起安装的缺省包之外，还有许多可以与选择软件一起安装到服务器的其他包。高级队列选项，即Oracle 8i的一个新建包，就是一个这样的例子。

在能够使用包含在Oracle 8i提供的包中的程序前，应该首先检查它们已被安装，并且是合法的。数据库系统管理员应该运行下面的查询：

```
SELECT object_name, object_type, status
FROM dba_objects
WHERE owner='SYS' AND object_type LIKE 'PACKAGE%'
ORDER BY object_name, object_type;
```

上面的查询会给出一个类似下面信息的列表：

OBJECT_NAME	OBJECT_TYPE	STATUS
DBMS_ALERT	PACKAGE	VALID
DBMS_ALERT	PACKAGE BODY	VALID
DBMS_APPLICATION_INFO	PACKAGE	VALID
DBMS_APPLICATION_INFO	PACKAGE BODY	VALID
DBMS_DDL	PACKAGE	VALID
DBMS_DDL	PACKAGE BODY	VALID
DBMS_DESCRIBE	PACKAGE	VALID
DBMS_DESCRIBE	PACKAGE BODY	VALID
DBMS_JOB	PACKAGE	VALID
DBMS_JOB	PACKAGE BODY	VALID
DBMS_LOCK	PACKAGE	VALID
DBMS_LOCK	PACKAGE BODY	VALID
DBMS_OUTPUT	PACKAGE	VALID
DBMS_OUTPUT	PACKAGE BODY	VALID
DBMS_PIPE	PACKAGE	VALID
DBMS_PIPE	PACKAGE BODY	VALID
DBMS_REFRESH	PACKAGE	VALID
DBMS_REFRESH	PACKAGE BODY	VALID
DBMS_SESSION	PACKAGE	VALID
DBMS_SESSION	PACKAGE BODY	VALID
DBMS_SHARED_POOL	PACKAGE	VALID
DBMS_SHARED_POOL	PACKAGE BODY	VALID
DBMS_SNAPSHOT	PACKAGE	VALID
DBMS_SNAPSHOT	PACKAGE BODY	VALID
DBMS_SPACE	PACKAGE	VALID
DBMS_SPACE	PACKAGE BODY	VALID
DBMS_SQL	PACKAGE	VALID
DBMS_SQL	PACKAGE BODY	VALID

DBMS_SYSTEM	PACKAGE	VALID
DBMS_SYSTEM	PACKAGE BODY	VALID
DBMS_TRANSACTION	PACKAGE	VALID
DBMS_TRANSACTION	PACKAGE BODY	VALID
DBMS_UTILITY	PACKAGE	VALID
DBMS_UTILITY	PACKAGE BODY	VALID

还有一些其他包没有显示在上面，但是这里不必关心它们。这些包中的多数不是为用户应用调用而准备的，仅仅在内部使用而已。

29.8.1 定位DBMS包

Oracle 8i提供的包位于 %Oracle_HOME%\RDBMS\ADMIN下，其中Oracle_HOME是指向Oracle主目录的路径（可以检查 UNIX系统中名字相同的系统变量或检查 Windows NT环境下的注册项）。可以检查表29-2中列出的每个包的包头文件，查看有哪些程序和全局变量可用。

或许还要注意具有 prvt*.sql和prvt*.plb形式的文件（例如，prvtpipe.sql和prvtpipe.plb）的存在，前面一种是提供的包的包体，采用 ASCII格式；后面一种是包体的二进制编译版本。PLB文件对于PL/SQL引擎是可识别的，当提交给 Oracle8i服务器时，产生一个合法并且可执行的包体。它们代表包体的“发布”形式。

警告 不要试图修改任何Oracle 8i提供的包的包体并重新编译，这样会破坏一些东西。

29.8.2 确定包已正确安装

需要验证两件事：

包存在并且合法（就像前面的查询显示的一样）。

用户对它们拥有EXECUTE权限或者对用户打算使用的包拥有EXECUTE权限。

使用下面的查询，一个用户可以查看他们是否对 Oracle8i提供的包拥有执行权限：

```
SELECT table_name, grantee
FROM all_tab_privs
WHERE grantor='SYS' and privilege='EXECUTE'
ORDER BY table_name;
```

一个典型的回答应该如下所示：

TABLE_NAME	GRANTEE
DBMS_APPLICATION_INFO	PUBLIC
DBMS_DDL	PUBLIC
DBMS_DESCRIBE	PUBLIC
DBMS_JOB	PUBLIC
DBMS_OUTPUT	PUBLIC
DBMS_PIPE	PUBLIC
DBMS_SESSION	PUBLIC
DBMS_SNAPSHOT	PUBLIC
DBMS_SPACE	PUBLIC
DBMS_SQL	PUBLIC
DBMS_STANDARD	PUBLIC
DBMS_TRANSACTION	PUBLIC
DBMS_UTILITY	PUBLIC
DBMS_REFRESH	PUBLIC
UTL_FILE	PUBLIC

所有的Oracle 8i提供的包应该被授予 PUBLIC上的EXECUTE特权。我使用数据字典视图 all_flavor验证这点。如果打算使用的任何包丢失，数据库系统管理员可以运行适当的脚本重

新生成它们。如果包存在但却是非法的，数据库系统管理员可以使用下面的命令重新编译它：

```
ALTER PACKAGE SYS.<name> COMPILE PACKAGE;
```

其中name是非法包的名字。子句 COMPILE PACKAGE告诉Oracle8i重新编译包声明和包体。如果仅仅包体需要重新编译，运行下面的命令：

```
ALTER PACKAGE SYS.<name> COMPILE BODY;
```

提示 一些提供的包可能与其他包有相关性。如果你必须重新编译一个包头，会让所有相关的包非法。再次检查在重新编译后其他的包是否合法。如果只有包体被重新编译，相关的包不会变为非法。这种方式也有可能使任何用户写入的存储子程序和包受到影响。

29.9 使用Oracle 8i提供的包

下面将使用一些难易适度并且非常有用的例子表明每个提供的包中的一些有趣的内容。

29.9.1 使用DBMS_APPLICATION_INFO监控

这个包让开发者能够在视图 V\$sqlarea和V\$session中添加追踪信息，可以追踪的数据种类如下所示：

存储在V\$sqlarea.module和V\$session.module中的模块名称（例如当前运行着的应用的名称），最长可以是48字节的VARCHAR2字符串。

存储在V\$sqlarea.action和V\$session.action中的当前动作（例如，“更新客户信息”、“验证信用卡限额”），最长可以达到32个字节的VARCHAR2字符串。

存储在V\$session.client_info中的任意客户定义信息，最长可以达到 64个字节的 VARCHAR2字符串。

Oracle 8i不对这些信息做任何处理，这些信息提供给数据库系统管理员使用，这样她就可以针对每个应用和部件操作执行统计。超过所能支持长度的字符串将被截断。程序清单 29-12显示了一个简单的例子。

清单29-12 APPINFO.SQL——设置并读取应用信息

```
DECLARE
  module VARCHAR2(48); -- application info
  action VARCHAR2(32);
  client VARCHAR2(64);
  ldate VARCHAR2(30); -- to capture system date
BEGIN
  DBMS_OUTPUT.enable;
  module := 'CLAIM TRACKING';
  action := 'VERIFY ELIGIBILITY';
  DBMS_APPLICATION_INFO.set_module(module, action);
  DBMS_APPLICATION_INFO.set_client_info(USER);
  DBMS_APPLICATION_INFO.read_module(module, action);
  DBMS_APPLICATION_INFO.read_client_info(client);
  DBMS_OUTPUT.put_line(client || ' is running ' || module || ': ' || action);
  SELECT TO_CHAR(SYSDATE, 'YYYY-MON-DD HH:MI:SS') INTO ldate
  FROM DUAL;
END;
```

得到的响应信息如下所示：

```
SCOTT is running CLAIM TRACKING: VERIFY ELIGIBILITY
```

在会话启动期间，数据库系统管理员可以使用下面的程序检查谁在做什么：

```
COLUMN module FORMAT A20
COLUMN action FORMAT A20
COLUMN client_info FORMAT A20
SELECT client_info, module, action
FROM V$session
WHERE client_info IS NOT NULL;
```

这次得到下列信息：

CLIENT_INFO	MODULE	ACTION
SCOTT	CLAIM TRACKING	VERIFY ELIGIBILITY

数据库系统管理员可以使用下面的语句查看每个人运行这个模块和动作的效果：

```
SELECT
  sql_text, SUM(sharable_mem) smem, SUM(persistent_mem) pmem,
  SUM(runtime_mem) rmem, SUM(sorts) sorts, SUM(loads) loads,
  SUM(disk_reads) rdisk, SUM(buffer_gets) bget,
  SUM(rows_processed) prows
FROM V$sqlarea
WHERE module = 'CLAIM TRACKING' AND action = 'VERIFY ELIGIBILITY'
GROUP BY sql_text;
```

结果非常有趣：

SQL_TEXT	SMEM	PMEM	RMEM	SORTS	LOADS	RDISK	BGET	PROWS
SELECT TO_CHAR(SYSDATE, 'YYYY-MON-DD HH:MI:SS') FROM DUAL	4267	508	792	0	1	0	4	2
SELECT USER FROM SYS.DUAL	3596	508	688	0	1	0	4	1
begin dbms_output.get_lines(:lines, :numlines); end;	4317	592	420	0	1	0	0	1

我明确地做出的唯一查询是第一个查询，但是很明显 SQL*Plus在这之后做了一些工作。

因为一个模块执行不同的动作，开发者应该对 set_action () 发出正确的调用以反映这点。采用这种方式，可以收集到一些有趣的统计信息，这些信息反映了一个应用是如何通过查询、动作、模块、用户或用户组恶劣地冲击服务器的。

注意 甚至当用户断开连接之后，项仍然存在于 V\$sqlarea中直到相关的SQL过时，从 SGA中消失为止。

29.9.2 使用DBMS_DDL重新编译包

可以对这个包做两件事：

使用alter_compile () 重新编译存储过程程序和包。

使用analyze_object () 分析一个表、索引或簇。

一个alter_compile () 的简单使用会发现哪些存储程序对象是合法的，然后重新编译它们。程序清单 29-13表明这种情形。

清单29-13 RECOMPIL.SQL——只重新编译非法的程序对象

```
-- recompile invalid stored program objects
-- CAVEAT: does not take package dependencies
--      into account!
DECLARE
  CURSOR invalid_prog_obj IS
    SELECT object_name, object_type
      FROM user_objects
     WHERE status = 'INVALID';
  rec invalid_prog_obj%ROWTYPE;
  status NUMERIC;
BEGIN
  DBMS_OUTPUT.enable;
  OPEN invalid_prog_obj;
  LOOP -- recompile each stored program object
    FETCH invalid_prog_obj INTO rec;
    EXIT WHEN invalid_prog_obj%NOTFOUND;
    DBMS_OUTPUT.put('Recompile ' || rec.object_type ||
                   ' ' || rec.object_name);

    DBMS_DDL.alter_compile(rec.object_type, NULL, rec.object_name);
    DBMS_OUTPUT.put_line(' SUCCESSFUL'); -- recompile succeeded
  END LOOP; -- invalid program objects
  CLOSE invalid_prog_obj;
EXCEPTION
  WHEN OTHERS THEN
    BEGIN
      status := SQLCODE;
      DBMS_OUTPUT.put_line(' FAILED with ' || SQLERRM(status));
      IF (invalid_prog_obj%ISOPEN) THEN
        CLOSE invalid_prog_obj;
      END IF;
    EXCEPTION WHEN OTHERS THEN
      NULL; -- do nothing
    END;
END;
/
```

程序会返回如下所示的信息：

```
Recompile FUNCTION TABLE_EXISTS SUCCESSFUL
1 Program Objects Recompiled
PL/SQL procedure successfully completed.
```

警告 如果一个程序对象不能够成功地重新编译，alter_compile将不会通知你！在运行alter_compile后，应该检查包的状态，确认它被成功地编译。

如果提供了一个并不存在的程序对象名，或者输错了程序对象名称，会得到下列信息：

```
ORA-20000: Unable to compile PACKAGE "BLICK", insufficient privileges or does not exist
```

可以程序地分析（生成统计数据）表、索引和簇。例如可以分析模式中全部或选定的对象，如程序清单29-14所示。

清单29-14 runstats.sql——程序地运行统计

```
-- analyze all tables, indexes and clusters in your own schema
-- computes exact statistics
SET ECHO OFF
ACCEPT method PROMPT 'ANALYZE Method ([COMPUTE]|ESTIMATE|DELETE): '
ACCEPT estrow PROMPT ' IF ANALYZE Method is ESTIMATE, #Rows (0-n) [100]: '
```

```

ACCEPT estpct PROMPT ' IF ANALYZE Method is ESTIMATE, %Rows (0-99) [20]: '
DECLARE
  -- application-defined exceptions
  bad_method EXCEPTION; -- user entered an invalid method
  bad_estrow EXCEPTION; -- user entered an invalid est. #rows
  bad_estpct EXCEPTION; -- user entered an invalid est. %rows
  -- cursors
  CURSOR analyze_obj IS
    SELECT object_name, object_type
    FROM user_objects
    WHERE object_type = ANY ('TABLE', 'INDEX', 'CLUSTER');
  -- constants
  METHOD CONSTANT VARCHAR2(30) := NVL(UPPER('&method'), 'COMPUTE');
  -- variables
  estrow NUMBER := NVL('&estrow', '100'); -- user input est. #rows
  estpct NUMBER := NVL('&estpct', '20'); -- user input est. pct
  rec analyze_obj%ROWTYPE;
  status NUMERIC := 0;
  cnt NUMERIC := 0;
BEGIN
  DBMS_OUTPUT.enable;
  -- validate user input
  IF (METHOD NOT IN ('COMPUTE', 'ESTIMATE', 'DELETE')) THEN
    RAISE bad_method;
  ELSIF (METHOD IN ('COMPUTE', 'DELETE')) THEN -- ignore est. #/row
    estrow := NULL;
    estpct := NULL;
  ELSE -- picked ESTIMATE; must provide either est. #rows or %rows
    IF (estrow < 1 AND estpct = 0) THEN
      RAISE bad_estrow;
    ELSIF (estpct NOT BETWEEN 1 AND 99) THEN
      RAISE bad_estpct;
    END IF;
  END IF; -- validate input
  OPEN analyze_obj;
  LOOP -- analyze schema objects
    FETCH analyze_obj INTO rec;
    EXIT WHEN analyze_obj%NOTFOUND;
    -- COMPUTE STATISTICS for this schema only
    DBMS_OUTPUT.put('Analyze ' || METHOD || ' ' ||
      rec.object_type || ' ' || rec.object_name);
    DBMS_DDL.analyze_object(rec.object_type, NULL, rec.object_name, 'COMPUTE');
    DBMS_OUTPUT.put_line(' SUCCESSFUL');
    cnt := cnt + 1;
  END LOOP; -- analyze schema objects
  CLOSE analyze_obj;
  DBMS_OUTPUT.put_line(TO_CHAR(cnt) || ' objects analyzed');
EXCEPTION
  WHEN bad_method THEN
    DBMS_OUTPUT.put_line('Invalid Method! Must be COMPUTE, ESTIMATE or DELETE
  only');
  WHEN bad_estrow THEN
    DBMS_OUTPUT.put_line('Invalid Est. #Rows! Must be >= 1');
  WHEN bad_estpct THEN
    DBMS_OUTPUT.put_line('Invalid Est. %Rows! Must be between 1 and 99');
  WHEN OTHERS THEN
    BEGIN
      status := SQLCODE;
      DBMS_OUTPUT.put_line(' FAILED with ' || SQLERRM(status));
      IF (analyze_obj%ISOPEN) THEN
        CLOSE analyze_obj;
      END IF;
    EXCEPTION WHEN OTHERS THEN
      NULL;

```

```
END;
END;
/
```

注意必须做全部输入验证。部分原因是因为 analyze_object 函数自己对此做得很少。例如，如果提供了一条使用 COMPUTE 方法估算的纪录行，会得到下列信息：

```
Analyze TABLE <table> FAILED with ORA-01490: invalid ANALYZE command.
```

当选择全部默认值计算统计信息时，运行结果如下所示：

```
ANALYZE Method ([COMPUTE]|ESTIMATE|DELETE):
  IF ANALYZE Method is ESTIMATE; #Rows (1-n) [1]:
  IF ANALYZE Method is ESTIMATE, %Rows (1-99) [20]:
old 12:  METHOD CONSTANT VARCHAR2(30) := NVL(UPPER('&method'), 'COMPUTE');
new 12:  METHOD CONSTANT VARCHAR2(30) := NVL(UPPER(''), 'COMPUTE');
old 14:  estrow NUMBER := NVL('&estrow', '1');          -- user input est.
#rows
new 14:  estrow NUMBER := NVL('1', '1');              -- user input est. #rows
old 15:  estpct NUMBER := NVL(TRUNC('&estpct'), '20'); -- user input est. pct
new 15:  estpct NUMBER := NVL(TRUNC(''), '20');       -- user input est. pct
Analyze COMPUTE TABLE BONUS SUCCESSFUL
Analyze COMPUTE TABLE DEPT SUCCESSFUL
Analyze COMPUTE TABLE EMP SUCCESSFUL
Analyze COMPUTE INDEX PK_DEPT SUCCESSFUL
Analyze COMPUTE INDEX PK_EMP SUCCESSFUL
Analyze COMPUTE TABLE SALGRADE SUCCESSFUL
6 objects analyzed
PL/SQL procedure successfully completed.
```

如果输入了一个非法的方法，服务器返回下列信息：

```
Invalid Method! Must be COMPUTE, ESTIMATE or DELETE only
```

这正是所希望的，反复使用它你会看到你是如何喜欢它。

ANALYZE 命令的这种实现方式没有做 VALID STRUCTURE 和 LIST CHAINED ROWS。因此，它是一种不完全的实现过程，但是仍然相当有用。

29.9.3 使用 DBMS_OUTPUT 格式化输出

如果跟着教材学习，你已经非常熟悉包中的 put_line 过程，这里是关于这个包实现的一些细节。

每行使用一个换行符结束。

每行最多允许 255 个字节，包括换行符。

每行存储在私有的 PL/SQL 表中。

除非首先调用 DBMS_OUTPUT.ENABLE，否则不存储任何东西。

缓存大小范围规定为 2 000 至 1 000 000 之间。

使用过程 enable 打开输出特性。如果没有首先打开输出，put_line 调用将被忽略。当激活输出时，还要指定一个缓存大小，如下所示：

```
DBMS_OUTPUT.enable(1000000); -- 1 million bytes is the max
```

反之，可以使用 disable 关闭输出，如下所示：

```
DBMS_OUTPUT.disable; -- turn off output
```

可以使用 put_line 存储一行文本。该函数被重载，可以接受一个 DATE、NUMBER 或 VARCHAR2 数值。通过使用 put 过程（它也以同样方式被重载），还可以存储一行没有结束的

文本。如果创建的文本行需要一些逻辑，则这个过程非常有用，如下面的例子所示：

```
-- excerpt taken from Package rev_eng
IF (Ltable IS NULL) THEN -- parameter comments
  DBMS_OUTPUT.put('-- ALL TABLES');
ELSE
  DBMS_OUTPUT.put('-- TABLE ' || Ltable);
END IF;
DBMS_OUTPUT.put_line(' FOR OWNER ' || Lowner ||
                      ', TABLESPACE ' || Ltspc);
```

通过使用new_line过程，可以结束使用put过程提交的文本。该过程使用一个标志表示当前文本行结束，并将文本行的长度与文本行一起存储（这对于用户来说是完全透明的）。注意，如果试图不首先使用put过程就发送新文本行，以两倍或三倍行距打印输出行，该过程将不工作：

```
BEGIN
  DBMS_OUTPUT.put('Hey, ');
  DBMS_OUTPUT.put('Dan!');
  DBMS_OUTPUT.new_line;
  DBMS_OUTPUT.new_line;
  DBMS_OUTPUT.put_line('Time to go to the library!');
end;
/
```

这次给出下列信息：

```
Hey, Dan!
Time to go to the library!
PL/SQL procedure successfully completed.
```

请注意没有打印两倍行距。

如果试图在使用new_line过程结束一行前，输出一个超过255个字节可接受长度的字符串，会得到一个例外：

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

为什么Oracle公司的人将字符串的长度限制为255？回顾前面的例子，在那里一个VARCHAR2(32767)类型的PL/SQL表很快地耗光了系统的可用内存（第28章“PL/SQL基本原理”，程序清单28-10）。换句话说，他们选择了一个他们认为合理的限制。

使用get_line返回字符串，这正是SQL*Plus返回使用put_line写入的字符串所做的事情。它以你看不见的方式调用get_line过程，直到再没有任何行可用为止。如果正在编写一个3GL程序接受使用put_line写入的文本行，确实只需要将注意力放到get_line（还有它的多行版本get_lines）过程的使用上。如果愿意，可以在一个PL/SQL程序中使用它，或许将它们插入一个表中，例如缓存文本行然后以FIFO方式访问它们。

缓存大小有什么用？毕竟，一个PL/SQL表可以含有超过四十亿的元素！设想你编写了一个C程序通过get_line过程接收缓存的字符串，分配一块4.29千兆×255字节的内存的确是一个浪费（而且也不是一个好主意）。你将需要上万亿字节的内存！虽然内存今天便宜了许多，但还没有便宜到那种地步。从而，你要求用户规定缓存的大小，比方说，在2K与1MB之间，并且那正是你所分配的。包追踪高速缓存空间的使用，因为你的C程序预期的正是用户指定的数据。然后将返回到该内存区的文本行压缩，浪费很少的空间或基本不浪费空间。最后，所有的文本行都可以显示给用户。这或许看起来不像是一个特别完善的模式，但它简单而且实用。

如果扩展缓存空间为所指定的大小，将得到下面的例外：

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

29.10 小结

本章学习了关于如何编写存储过程和包的丰富信息，还学习了如何使用与数据库服务器一起提供的包。学习了如何使用 SHOW ERRORS 命令调试存储程序，甚至能够编写一个脚本提高那些命令的输出。最后，学习了如何检查存储过程的合法性，以及如果存储过程变为非法时如何重新编译它们。现在应该能够理解本章中的例子并能够使用你自己的用户化要求来提高它们，或创建你自己设计的新过程。