Введение в PL/SQL

Материалы учебного курса

Владимир Викторович Пржиялковский
open-oracle.ru
prz@yandex.ru

Март 2019

Москва

... Не нужно более слов — дела нужны; ибо, с соизволения богов, я убежден, что вскоре исполнятся мои обещания.

Ганнибал у Полибия во "Всеобщей истории"

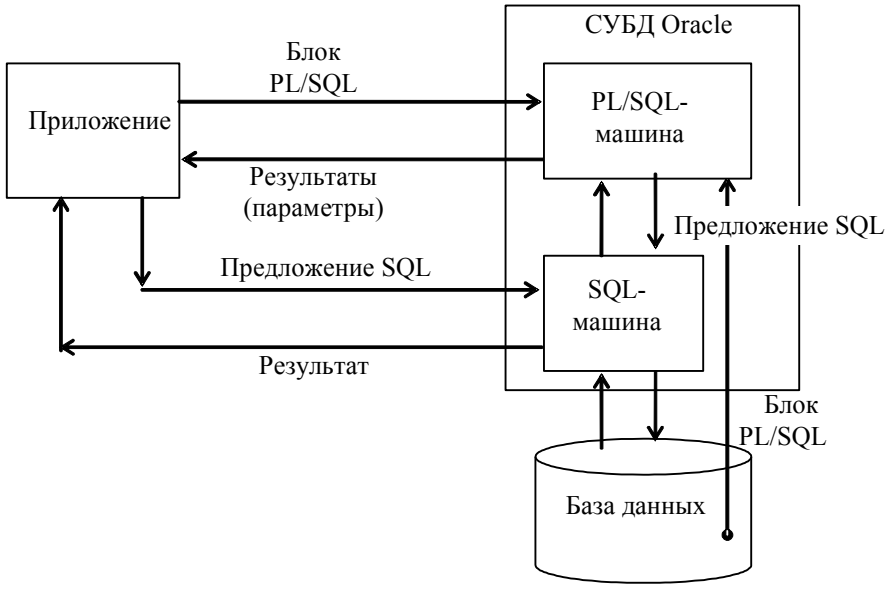
По делом своим приимете.

Лука 25:41 и др. Цит. по: Домострой, I-я редакция

Оглавление

1. Основные понятия

1.1. Место PL/SQL в архитектуре Oracle



Предоставляемый PL/SQL-машине код представлен в промежуточном виде и фактически интерпретируется. С версии СУБД 9 появиласть возможность транслировать программы на PL/SQL в машинный код.

1.2. Общая структура программы на PL/SQL

Общее устройство блока в PL/SQL:

```
[ Заголовок блока ]
[ IS | AS | DECLARE
    Раздел объявлений ]
  BEGIN
    Раздел выполнения
[ EXCEPTION
    Раздел обработки
    исключительных ситуаций ]
  END;
```

Все разделы, кроме раздела выполнения, имеют право отсутствовать. Например, может отсутствовать заголовок, и в этом случае блок называется неименованым или же анонимным.

Раздел выполнения может наряду с программным кодом содержать произвольное число других (вложенных) блоков PL/SQL.

Пример предъявления серверу блока PL/SQL (анонимного) посредством SQL*Plus:

```
SET SERVEROUTPUT ON

DECLARE
   v_msg VARCHAR2 ( 20 );

BEGIN
   /*
      комментарий может быть многострочным
   */                    -- хвостовой комментарий
   NULL;                                -- (1) пустой оператор: ничего не делать
   v_msg := 'Ave, Caesor !';            -- (2) оператор присвоения переменной значения
   DBMS_OUTPUT.PUT_LINE ( v_msg );-- (3) вызов процедуры
   BEGIN                                -- (4) вложенный блок: тоже оператор
      DBMS_OUTPUT.PUT_LINE ( 'Ave, imperator !' ); -- (4.1) вызов процедуры
   END;                                 -- (4) конец вложенного блока
END возможно_слово_пояснения;
/
```

Замечание. *слово_пояснения*, которое программист может по желанию добавить после замыкающего блок END, подчиняется правилам составления имен объектов хранения в БД. До версии Oracle12 на имена выделялось наибольшее 30 байтов. Если основная кодировка БД — Unicode, то каждая русская буква в имени будет требовать по 2 байта. Указанное в тексте выше слово (а именно, *возможно_слово_пояснения*) в таком случае, если его оставить, приведет к ошибке на этапе трансляции.

Замечание. За редкими исключениями код, принимаемый для исполнения программой SQL*Plus, правильно отрабатывает в среде SQL Developer.

2. Основные типы и структуры данных

Набор типов переменных в PL/SQL отличен от набора типов в Oracle SQL. С версии 9 это чистое расширение, так как с этой версии СУБД PL/SQL автоматически поддерживает все типы Oracle SQL за счет использования общего одного модуля обработки SQL-запросов (до этого PL/SQL использовал собственную SQL-машину).

Все переменные в PL/SQL можно отнести к одному из следующих видов:

Вид данных	***Описание***
Простой скалярный	Переменные, представляющие собой ровно одно значение (числовое, дату и т.д.)
Составной	Переменные, представляющие именованную группу значений (запись) или скаляр с объявленной структурой (объект, массив)
Ссылка	Ссылка на объект или курсор
LOB	Указание на массив большого размера

2.1. Простые скалярные типы

2.1.1. Числовые типы

Делятся (как и в SQL) на типы:

- как в SQL — с двоично-десятичным хранением (NUMBER) и с двоичным (BINARY_FLOAT и BINARY_DOUBLE[10-], с их подтипами специально для PL/SQL[11-]);
- собственные в PL/SQL — с двоичным BINARY_INTEGER и PLS_INTEGER, оба с подтипами.

[10-] С версии 10.
[11-] С версии 11.

Типы BINARY_INTEGER и PLS_INTEGER допускают значения из диапазона -2147483648 .. 2147483647. Тип PLS_INTEGER использует аппаратный формат хранения и частично программную арифметику (сравнения с NULL и переполнения), что до версии 10 делало его более производительным при вычислениях по сравнению с BINARY_INTEGER, использовавшим программный формат. Помимо внутреннего формата типы различались деталями поведения при переполнении.

Переполнение PLS_INTEGER *вызывает* исключительную ситуацию. До версии 9 включительно переполнение BINARY_INTEGER *не вызывает* исключительную ситуацию, если результат присваивается типу NUMBER.

В версии 10 тип BINARY_INTEGER сведен к типу PLS_INTEGER, и фактическая разница между ними стерлась, но формально несущественные технические отличия остались.

Определения подтипов BINARY_INTEGER:

```
SUBTYPE NATURAL     IS BINARY_INTEGER RANGE 0 .. 2147483647;
SUBTYPE NATURALN    IS NATURAL NOT NULL;
SUBTYPE POSITIVE    IS BINARY_INTEGER RANGE 1 .. 2147483647;
SUBTYPE POSITIVEN   IS POSITIVE NOT NULL;
SUBTYPE SIGNTYPE    IS BINARY_INTEGER RANGE -1 .. 1;
```

Следующие подтипы были введены в версии 11:

```
SUBTYPE SIMPLE_INTEGER    IS BINARY_INTEGER    NOT NULL;
SUBTYPE SIMPLE_FLOAT      IS BINARY_FLOAT      NOT NULL;
SUBTYPE SIMPLE_DOUBLE     IS BINARY_DOUBLE     NOT NULL;
```

Во время исполнения программы эти подтипы более производительны, чем их супертипы (*если* значение параметра СУБД PLSQL_CODE_TYPE = NATIVE), поскольку наличие у переменных этих подтипов значений (то есть, недопущение для них NULL) проверяется загодя, во время трансляции программы, а не по ходу исполнения. В то же время, при выполнении арифметических операций машина PL/SQL не будет проверять переполнение; это как раз и обуславливает высокую скорость обработки, но при больших значениях может приводить к содержательно неожиданным результатам (например, добавление единицы может сделать положительное целое отрицательным там, где для обычных типов произойдет переполнение).

2.1.2. Типы для строк

Типы переменных для строк текста и байтов:

Тип	***Описание***
CHAR	Строки текста фиксированной длины до 32767 байт (в *SQL* предел 2000 байт) — в основной кодировке БД
VARCHAR2	Строки текста переменной длины до 32767 байт (в *SQL* в версиях Oracle 7- предел 2000 байт, в версиях 8-11 предел 4000 байт, в версиях 12+ предел 32767 байт) — в основной кодировке БД
NCHAR NVARCHAR2	Аналогично CHAR и VARCHAR2, но для строк текста в дополнительной кодировке БД: Unicode.
RAW	Строки байтов переменной длины до 32767 байт (в Oracle *SQL* предел 2000 байт).
LONG	Строки текста переменной длины до 32767 байт (в Oracle *SQL* до 2 Гб - 1). Тип сохранен для обратной совместимости; находит применение в некоторых справочных таблицах
LONG RAW	Строки байтов переменной длины до 32767 байт. Тип сохранен для обратной совместимости версий Oracle

Следующие типы формально можно причислить к строковым, но используются они для представления физических адресов:

ROWID	Двоичный массив фиксированной длины для хранения физического адреса данных Oracle в шестнадцатеричном формате BBBBBBBB.RRRR.FFFF (Oracle 7-) и в формате OOOOOOFFFBBBBBBRRR (Oracle 8+)
UROWID[8.1-)	«Универсальный» формат для ROWID: шестнадцатеричная строка переменной длины (до 4000 байт) с логическим значением ROWID. Используется для хранения адресов строк в индексно организованных (index organized) таблицах или в таблицах DB2 (через шлюз)

[8.1-) С версии 8.1.

2.1.3. Типы для моментов и интервалов времени

То же, что в Oracle SQL плюс несколько специальных подтипов.

Следующие подтипы в PL/SQL используются для передачи параметров с максимально возможной точностью («без ограничения точности»):

Подтип ...	***... соответствует типу***
TIMESTAMP_UNCONSTRAINED	TIMESTAMP (9)
TIMESTAMP_TZ_UNCONSTRAINED	TIMESTAMP (9) WITH TIME ZONE
YMINTERVAL_UNCONSTRAINED	INTERVAL YEAR (9) TO MONTH
DSINTERVAL_UNCONSTRAINED	INTERVAL DAY (9) TO SECOND (9)
TIME_UNCONSTRAINED	TIME (9)(*)
TIME_TZ_UNCONSTRAINED	TIME (9) WITH TIME ZONE(*)

(*) Как самостоятельный тип в Oracle не реализован; фактически недореализован, но подтип возможен для описания переменных и параметров в PL/SQL.

2.1.4. Булев тип

«Булевым» в PL/SQL назван логический тип фактически не для булевой, а для трехзначной логики с допустимыми значениями TRUE, FALSE и NULL. Это противоречит точке зрения на NULL как на отсутствующее значение, тогда как ничто не мешает программисту наделять NULL подобным смыслом (наряду со смыслом «значение неизвестно»). Правила сравнения и операции — те же, что в Oracle SQL для условных выражений.

2.2. Типы LOB («большие неструктурированные объекты»)

В отличие от скалярных типов, LOB-типы позволяют хранить не сами данные, а «локаторы» (указатели) на данные, размещенные вне, либо внутри БД.

Тип	***Описание***
BFILE	Указатель на файл с данными в операционной системе. Средствами Oracle данные можно только читать.
BLOB	Указатель на большой неструктурированный массив в БД
CLOB	Указатель на большой символьный массив в БД
NCLOB	Указатель на большой символьный массив в многобайтовой кодировке

2.3. Объявление переменных и постоянных

Все переменные в PL/SQL должны быть описаны:

имя_переменной [CONSTANT] *тип_данных* [NOT NULL] [{:= | DEFAULT} *выражение*] ;

Примеры:

```
DECLARE
   v_surname      VARCHAR2 ( 40 );
   v_всё_включено BOOLEAN NOT NULL := FALSE;
BEGIN NULL; END;
/
```

Имена переменных (равно как и прочих объектов программы на PL/SQL) строятся по правилам Oracle SQL, но с небольшим отличием списка зарезервированных слов, запрещенных для употребления в этом качестве.

Указание CONSTANT задает неизменяемую «переменную», то есть постоянную, константу. (Неизменность значения отслеживается синтаксически запретом присвоения значения).

В PL/SQL, вдобавок к SQL, имеется несколько встроенных постоянных для типов BINARY_FLOAT и BINARY_DOUBLE. Их формальные определения («значения» автоматически встроены в код программы):

```
BINARY_FLOAT_MAX_NORMAL     CONSTANT BINARY_FLOAT;
BINARY_FLOAT_MIN_NORMAL     CONSTANT BINARY_FLOAT;
BINARY_FLOAT_MAX_SUBNORMAL  CONSTANT BINARY_FLOAT;
BINARY_FLOAT_MIN_SUBNORMAL  CONSTANT BINARY_FLOAT;
BINARY_DOUBLE_MAX_NORMAL    CONSTANT BINARY_DOUBLE;
BINARY_DOUBLE_MIN_NORMAL    CONSTANT BINARY_DOUBLE;
BINARY_DOUBLE_MAX_SUBNORMAL CONSTANT BINARY_DOUBLE;
BINARY_DOUBLE_MIN_SUBNORMAL CONSTANT BINARY_DOUBLE;
```

При отсутствии указания DEFAULT (:=) начальное значение переменной всегда NULL (пример следует ниже).

В выражении DEFAULT (:=) можно ссылаться на выше описанные переменные, если им были присвоены собственные начальные значения:

```
DECLARE
   x NUMBER := 1;
   y NUMBER := SIN ( x );
   z VARCHAR2 ( 10 );
BEGIN
   DBMS_OUTPUT.PUT_LINE ( x );
   DBMS_OUTPUT.PUT_LINE ( y );
   DBMS_OUTPUT.PUT_LINE ( 'Без DEFAULT значения нет:' || z || '<NULL');
END;
/
```

2.4. Составные типы

Составные типы в PL/SQL представлены записями, объектами и массивами.

2.4.1. Записи

Записи есть пример данных составного типа. В программе на PL/SQL они создаются с разными целями:

- для воспроизводения структуры таблицы в БД
- для воспроизводения структуры курсора в программе
- произвольно на усмотрение пользователя

Помимо этого к числу составных типов (допускающих структуру у значения) Oracle относит типы объектов и коллекций.

2.4.1.1. Объявление записей в программе

Записи в PL/SQL могут объявляться в разделе объявлений блока или в разделе глобальных описаний пакета. Записи, повторяющие структуру таблицы или курсора, объявляются с помощью атрибута %ROWTYPE. Записи, задаваемые пользователем, объявляются через предложение TYPE:

```
DECLARE
   TYPE name_rectype IS RECORD (
     first_name VARCHAR2 ( 30 )
   , last_name  VARCHAR2 ( 30 )
   );

   TYPE employee_rectype IS RECORD (
     emp_id    NUMBER ( 10 ) NOT NULL DEFAULT 1000
   , dept_no   dept.deptno%TYPE
   , title     VARCHAR2 ( 20 )
   , name      name_rectype
   , hire_date DATE := SYSDATE
   , fresh_out BOOLEAN
   );

   r_new_emp_rec employee_rectype;
BEGIN
   r_new_emp_rec.fresh_out := TRUE;
   r_new_emp_rec.name.last_name := 'Scott';
END;
/
```

Из примера видно, что записи могут быть вложенными.

2.4.1.2. Присвоения

Индивидуальные поля указываются через точку (например, new_emp_rec.name.first_name) и могут выставляться, читаться и сравниваться самостоятельно.

Присвоение значений всем полям записи зараз может выполняться:

- оператором присвоения: '**:=** ***имя_однотипной_записи***'
- предложением SELECT … **INTO** ***имя_записи*** FROM …
- предложением FETCH *имя_курсора* **INTO** ***имя_записи***

(примеры последуют ниже).

Записи как целое не могут сравниваться логическими операциями (=, <> и пр.).

2.4.2. Объекты

Для обычного типа объектов описание возможно только на уровне БД, но не в программе. Однако заводить в программе переменные типа объект возможно:

```
DECLARE
  addr address_type;
BEGIN
  NULL;
END;
/
```

Работа с объектными переменными в программе на PL/SQL имеет свои возможности и особенности, здесь и далее не рассматриваемые.

Коллекции (массивы) Oracle также причисляет к объектным типам, однако заводить тип коллекции можно уже не только в БД, но также и в программе. Примеры определения типов коллекции в программе и работы с ними будут приведены позже.

2.5. Ссылка на типы уже имеющихся данных и собственные подтипы

Два решения позволяют повышать в PL/SQL надежность описаний групп переменных, однотипных или почти однотипных по их прикладному назначению: разрешение ссылаться на тип ранее описаных переменных (или данных в БД) и собственные подтипы.

2.5.1. Ссылка на типы уже имеющихся данных

Атрибут %TYPE в описании скалярной переменной позволяет указать для нее тип, имеющийся у другой переменной или у столбца в таблице. Атрибут %ROWTYPE в описании записи позволяет указать для нее тип, имеющийся у курсора или строки таблицы.

Примеры:

```
DECLARE
  v_total_sales   NUMBER ( 20, 2 );
  v_monthly_sales v_total_sales%TYPE;     -- привязка к типу другой переменной
  v_quarter_sales v_total_sales%TYPE NOT NULL DEFAULT 1000;
                                          -- привязка с уточнением

  v_ename      emp.ename%TYPE;            -- привязка к типу столбца таблицы из БД
```

```
  vrec_emp     emp%ROWTYPE;            -- привязка к типу таблицы из БД

  CURSOR cur_emp IS SELECT * FROM emp;
  vrec_emp1    cur_emp%ROWTYPE;        -- привязка к типу курсора
BEGIN NULL; END;
/
```

2.5.2. Пользовательские подтипы

Для переменных в блоке или пакете PL/SQL можно определять собственные подтипы. Некоторые примеры:

```
DECLARE
SUBTYPE   dollar_amount_type IS NUMBER ( 20, 2 );
credit    dollar_amount_type;
debit     dollar_amount_type NOT NULL DEFAULT 0;

SUBTYPE   dollar_amount_typenn IS dollar_amount_type NOT NULL;
mycredit  dollar_amount_typenn DEFAULT 1000;
mydebit   dollar_amount_typenn DEFAULT 0;

cache    NUMBER ( 10, 2 );
SUBTYPE  my_cache   IS cache%TYPE NOT NULL;
SUBTYPE  your_cache IS cache%TYPE;

SUBTYPE  name_type IS emp.ename%TYPE;
myname   name_type;
yourname name_type;

SUBTYPE  emp_type IS emp%ROWTYPE;

SUBTYPE     bytes_type IS BINARY_INTEGER RANGE 1024 .. 2147483647;
all_bytes   bytes_type;
some_bytes  bytes_type RANGE 4096 .. 8196;

BEGIN NULL; END;
/
```

3. Выражения

Строятся в целом по тем же правилам, что в Oracle SQL, но есть и отличия.

Выражения в PL/SQL допускаются над данными всех типов за небольшим числом исключений (записи, курсоры и ссылки на курсор, частично коллекции и объекты). С версии 9 они подчиняются тем же правилам построения, что и в диалекте SQL в Oracle (до этого правила в PL/SQL были более ограничительны). Кроме этого в выражениях на PL/SQL могут участвовать скалярные типы, отсутствующие в Oracle SQL, записи или коллекции вида «ассоциативный массив». По естественным причинам при построении выражений в PL/SQL не могут участвовать агрегирующие и аналитические функции, вполне допустимые в выражениях Oracle SQL.

Примеры выражений (выделены жирным шрифтом):

```
DECLARE
   i NUMBER;
   n CONSTANT VARCHAR2 ( 10 ) := 'Scott';
   c BOOLEAN;

 BEGIN
   i := SIN ( 3 ) / COS ( 3 );
   DBMS_OUTPUT.PUT_LINE ( 'Tangens of 3 radians: ' || TO_CHAR ( i ) );
   -- неявное преобразование допускается, но не рекомендуется:
   DBMS_OUTPUT.PUT_LINE ( 'The same tangens: ' || i );

   c := USER LIKE 'S%';
   IF c THEN DBMS_OUTPUT.PUT_LINE ( USER || ' begins with S' ); END IF;
   DBMS_OUTPUT.PUT_LINE
          ( 'User ' ||
            CASE INITCAP ( USER )          -- вариант CASE для версии >= 9
               WHEN n                THEN 'is SCOTT'
               WHEN 'Scott'          THEN 'is SCOTT too'
               WHEN INITCAP ( USER ) THEN 'is SCOTT again'
               ELSE                       'SCOTT is not here'
            END
          );

  -- для версии >= 9
  IF TIMESTAMP '2003-04-14 15:16:17' > SYSTIMESTAMP THEN
           DBMS_OUTPUT.PUT_LINE ( 'OK' );
  END IF;
END;
/
```

Обратите внимание на использование встроенных функций, системных переменных и на неявное преобразование типов — все в соответствии с правилами Oracle SQL.

Логические выражения вычисляются экономно — до первого обнаружения гарантированно ожидаемого результата (но с незначительными отличиями от техники оценки выражений в Oracle SQL).

3.1. Ограничения для выражений в PL/SQL

При построении выражений в PL/SQL можно столкнуться с некоторыми неочевидными ограничениями по отношению к SQL. Часть из них связана с тем, что стандартные функции в SQL и PL/SQL имеют по факту два разных воплощения одного и того же кода. При этом некоторые функции для PL/SQL почему-то остались незапрограммированными.

Так например, в выражениях PL/SQL нельзя использовать функции DECODE и (в отличие от NVL) NVL2. Следующий код приводится для SQL*Plus:

```
SQL> VARIABLE n number
SQL> BEGIN :n := NVL ( 1, 2 ); END;
  2  /

PL/SQL procedure successfully completed.

SQL> BEGIN :n := NVL2 ( 1, 2, 3 ); END;
  2  /
BEGIN :n := NVL2 ( 1, 2, 3 ); END;
            *
ERROR at line 1:
ORA-06550: line 1, column 13:
PLS-00201: identifier 'NVL2' must be declared
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored


SQL> BEGIN SELECT NVL2 ( 1, 2, 3 ) INTO :n FROM dual; END;
  2  /

PL/SQL procedure successfully completed.
```

Здесь код функции NVL2 в машине SQL в СУБД запрограммирован, а в машине PL/SQL оказался нет.

В связи с именно этим примером можно вспомнить, что оператор CASE, не запрещенный для использования в выражениях PL/SQL (см. пример выше), способен заменить собой NVL, NVL2, COALESCE и DECODE. Однако для непосредственного употребления в выражениях PL/SQL могут оказаться закрыты также некоторые другие, как правило более специальные функции, например имеющиеся для типа XMLTYPE (EXISTSNODE, UPDATEXML, XMLQUERY и др.). При необходимости к ним придется обратиться из встроенного SQL.

В отличие от Oracle SQL, в числовых выражениях PL/SQL использовать функции CURRVAL и NEXTVAL для генераторов чисел можно только начиная с версии 11.

3.2. Расширения возможностей выражений в PL/SQL

Одновременно Oracle иногда разрешает в выражениях PL/SQL делать больше, чем допускает в SQL. Например для программной переменной типов LOB PL/SQL разрешает выполнять проверку на равенство пустому локатору (возвращаемого функциями EMPTY_CLOB и EMPTY_BLOB) , а SQL нет:

```
DECLARE clobval CLOB;
BEGIN
  IF clobval = EMPTY_CLOB ( ) THEN NULL; END IF;
  -- OK
  UPDATE dept SET loc = loc WHERE clobval = EMPTY_CLOB ( );
  -- Ошибка !
END;
/
```

Здесь первое применение условного выражения разбирается машиной PL/SQL, а второе – машиной SQL. В то же время EMPTY_CLOB и EMPTY_BLOB можно указать в SQL, но только в качестве объекта присвоения, по типу: `UPDATE ... SET xxx = EMPTY_CLOB ( ) ...`, или, аналогично, в INSERT. (При этом, если `SET xxx = NULL` уберет из строки таблицы локатор и удалит из БД значение LOB, на которое локатор ссылается, то `SET xxx = EMPTY_CLOB ( )` только уберет локатор, но само значение LOB в БД не тронет.)

Аналогично, сравнение двух значений типов LOB не поддерживается в выражениях в SQL, но обеспечено в PL/SQL.

3.3. Отличия в вычислении выражений в SQL и PL/SQL

Вычисление выражений машинами SQL и PL/SQL в целом ведется одинаковыми фрагментами кода и происходит одинаково. Однако в отдельных местах имеются различия.

Логические выражения нередко формулируются серией условий, соединенных операциями OR и AND. Oracle вычисляет их «экономно», то есть до выяснения первого значения TRUE для цепочек из OR и первого значения FALSE для AND. Этим часто пользуются программисты, когда стараются оптимизировать вычисления. Однако машина SQL цепочки из OR оценивает *слева направо*, а цепочки из AND *справа налево*, в то время как машина PL/SQL — *всегда* только *слева направо*.

В следующих примерах скобки в выражениях проставлены ради наглядности. В SQL:

```
SELECT 'что-нибудь' FROM dual
WHERE
  ( 1 = 1 / 0 ) AND ( 1 = SYSDATE + 0 - SYSDATE )
-- ошибки нет!
;

SELECT 'что-нибудь' FROM dual
WHERE
  ( 1 = SYSDATE + 0 - SYSDATE ) AND ( 1 = 1 / 0 )
-- ошибка ORA-01476: делитель равен нулю
;
```

В PL/SQL:

```
BEGIN
IF  ( 1 = 1 / 0 ) AND ( 1 = SYSDATE + 0 - SYSDATE ) THEN NULL;
END IF;
END;
-- ошибка ORA-01476: делитель равен нулю
/

BEGIN
IF  ( 1 = SYSDATE + 0 - SYSDATE ) AND ( 1 = 1 / 0 ) THEN NULL;
END IF;
END;
-- ошибки нет!
/
```

Упражнение. Проверить аналогичным образом порядок вычисления условных выражений, составленных из OR.

Еще пример: машина PL/SQL обрабатывает операцию возведения в степень (**), а машина SQL нет.

```
EXECUTE DBMS_OUTPUT.PUT_LINE ( 2**5 ); DBMS_OUTPUT.PUT_LINE ( 3**3 + 3 )

SELECT 2**5 FROM dual
-- ошибка!
/
```

Например, вычисление площади круга:

```
DECLARE
  r  NUMBER;
  PI CONSTANT NUMBER := 3.14;
  s  NUMBER;
BEGIN
```

```
  r := 2;
  s := PI * r ** 2;
  DBMS_OUTPUT.PUT_LINE ( s );
END;
/
```

4. Основные управляющие структуры

4.1. Ветвление программы

4.1.1. Предложение IF-THEN

IF *условное_выражение*
THEN
    программный код в случае TRUE
END IF

4.1.2. Предложение IF-THEN-ELSE

IF *условное_выражение*
THEN
    программный код в случае TRUE
ELSE
    программный код в случае FALSE/NULL
END IF

Некоторые советуют избегать предложений IF-THEN, отдавая для ясности кода предпочтение равносильной записи с IF-THEN-ELSE:

IF *условное_выражение*
THEN
    программный код в случае TRUE
ELSE
    NULL;
END IF

4.1.3. Предложение IF-THEN-ELSIF

IF *условное_выражение*$_1$ THEN *программный код*$_1$
ELSIF *условное_выражение*$_2$ THEN *программный код*$_2$
[ELSIF *условное_выражение*$_i$ THEN *программный код*$_i$] ...
[ELSE *программный код в случае FALSE/NULL*]
END IF

4.1.4. Предложения CASE

Существует, начиная с версии 9.

По аналогии с CASE-выражениями имеет две разновидности:

CASE
WHEN *условное_выражение*$_1$ THEN *программный код*$_1$
[WHEN *условное_выражение*$_i$ THEN *программный код*$_i$] …
[ELSE *программный код*]
END CASE

Этот вариант предложения CASE полностью равносилен предложению IF … ELSEIF …, однако по мнению специалистов более предпочтителен по методическим соображениям.

Для последующих примеров выдадим в SQL*Plus:

```
SQL> VARIABLE n NUMBER
```

Первый пример с CASE:

```
/* (A) */
BEGIN
 CASE
    WHEN :n = 0 THEN DBMS_OUTPUT.PUT_LINE ( '0' );
    WHEN :n > 0 THEN DBMS_OUTPUT.PUT_LINE ( 'Больше 0' );
    WHEN :n < 0 THEN DBMS_OUTPUT.PUT_LINE ( 'Меньше 0' );
    ELSE             DBMS_OUTPUT.PUT_LINE ( 'NULL' );
 END CASE;
END;
/
```

Вторая разновидность:

CASE *выражение*$_0$
WHEN *выражение_для_сравнения*$_1$ THEN *программный код*$_1$
[WHEN *выражение_для_сравнения*$_i$ THEN *программный код*$_i$] …
[ELSE *программный код*]
END CASE

Пример:

```
/* (Б) */
BEGIN
 CASE :n
    WHEN             0 THEN DBMS_OUTPUT.PUT_LINE ( '0' );
    WHEN   ABS ( :n ) THEN DBMS_OUTPUT.PUT_LINE ( 'Больше 0' );
    WHEN - ABS ( :n ) THEN DBMS_OUTPUT.PUT_LINE ( 'Меньше 0' );
    ELSE                   DBMS_OUTPUT.PUT_LINE ( 'NULL' );
 END CASE;
END;
/
```

Пример (Б) можно привести к примеру (А) следующим образом:

```
/* (B) */
BEGIN
 CASE TRUE
    WHEN :n = 0 THEN DBMS_OUTPUT.PUT_LINE ( '0' );
    WHEN :n > 0 THEN DBMS_OUTPUT.PUT_LINE ( 'Больше 0' );
    WHEN :n < 0 THEN DBMS_OUTPUT.PUT_LINE ( 'Меньше 0' );
    ELSE             DBMS_OUTPUT.PUT_LINE ( 'NULL' );
 END CASE;
END;
/
```

Конкретно последние примеры любопытно сравнить с другими, где вместо *предложений* CASE используются *выражения* CASE. Хотя бы так:

```
/* (Г) */
BEGIN
DBMS_OUTPUT.PUT_LINE (
    CASE :n
        WHEN             0 THEN '0'
        WHEN   ABS ( :n ) THEN 'Больше 0'
        WHEN - ABS ( :n ) THEN 'Меньше 0'
        ELSE                   'NULL'
    END
);
END;
/
```

Если конструкция ELSE отсутствует, *в отличие от выражений CASE* здесь порождается исключительная ситуация. Следующие примеры *вызовут* ошибку:

```
BEGIN CASE WHEN FALSE THEN NULL; END CASE; END;
/

BEGIN CASE :n WHEN 1 THEN NULL; END CASE; END;
/
```

Следующие примеры *не* приведут к ошибке:

```
BEGIN CASE WHEN TRUE THEN :n := 1; END CASE; END;
/

BEGIN CASE :n WHEN 1 THEN :n := 2; END CASE; END;
/

BEGIN CASE WHEN FALSE THEN :n := 1; ELSE :n := 2; END CASE; END;
/
```

Упражнение. Выше были приведены четыре способа запрограммировать одно и то же: (А), (Б), (В) и (Г). Сравнить их друг с другом и привести для каждого доводы за и против. (Доводы могут быть и объективными, и субъективными.) Какому и при каких обстоятельствах отдать предпочтение? Какие еще можно предложить формулировки?

4.1.5. Безусловная передача управления

Предложение:

GOTO *имя_метки*

Формат указания метки:

<<имя_метки>> предложение;

Предложение может быть любым, в частности NULL.

Ограничения области действия GOTO:

- нельзя передавать управление *внутрь* предложения IF, LOOP и вложенного блока
- нельзя передавать управление из одного раздела предложения IF в другой
- нельзя передавать управление извне/внутрь подпрограммы
- нельзя передавать управление из раздела обработки исключительных ситуаций в основной раздел блока PL/SQL
- нельзя передавать управление из основного раздела блока PL/SQL в раздел обработки исключительных ситуаций (это можно делать только с помощью RAISE)

Обратите внимание, что передавать управление *изнутри* предложения IF, LOOP и вложенного блока *вовне* возможно, например:

```
SQL> BEGIN IF TRUE THEN GOTO a; END IF; <<a>> NULL; END;
  2  /

PL/SQL procedure successfully completed.
```

Однако следующий код вызовет *ошибку* трансляции:

```
BEGIN IF TRUE THEN <<a>> NULL; END IF; GOTO a; END;
/
```

Или, чтобы не было сомнений в распознавании компилятором зацикленности кода:

```
BEGIN GOTO a; IF TRUE THEN <<a>> NULL; END IF; END;
/
```

Методология использования предложения GOTO в программировании имеет давнюю литературу и требует хотя бы общего знакомства с ней программиста. Считается, что пользоваться безусловными переходами в программе надо по меньшей мере с осторожностью, а некоторые полагают необходимым вообще исключить GOTO из текстов программ. Краткое знакомство с вопросом дает *Википедия*.

4.2. Циклы

4.2.1. Простой цикл

LOOP
    программный код
END LOOP [*слово-комментарий*]

Выход из цикла может осуществляться с помощью GOTO, однако правильнее это делать с помощью специальных операторов EXIT или EXIT WHEN:

```
BEGIN
   LOOP DBMS_OUTPUT.PUT_LINE ( 'Enough !' ); EXIT; END LOOP;
END;
/
```

С версии 11 в теле цикла действуют к тому же операторы CONTINUE и CONTINUE WHEN (переход к следующей итерации цикла — т. е. не связанные с выходом из цикла).

4.2.2. Цикл WHILE

WHILE *условное_выражение_равно_TRUE*
LOOP
    программный код
END LOOP [*слово-комментарий*]

4.2.3. Счетный цикл (FOR)

FOR *переменная_цикла* IN [REVERSE] *нижнее_значение .. верхнее_значение*
LOOP
    программный код
END LOOP [*слово-комментарий*]

переменная_цикла заводится в PL/SQL автоматически, с типом PLS_INTEGER; объявлять ее самостоятельно не требуется. *нижнее_значение* и *верхнее_значение* могут быть выражениями, оценка которых выполняется перед первым вхождением в цикл.

Пример, поясняющий область действия переменной цикла и возможность явного ухода из цикла FOR (обратите внимание, что значение переменной цикла в последнем случае не сохраняется, и если в этом возникнет необходимость, перед выходом потребуется сделать присвоение специально заведенной внешней переменной):

```
DECLARE i BINARY_INTEGER := 10;
BEGIN
   FOR i IN 1 .. 5 LOOP DBMS_OUTPUT.PUT_LINE ( i ); END LOOP;
   DBMS_OUTPUT.PUT_LINE ( i );                              -- 10?5?
   FOR i IN 1 .. 5 LOOP DBMS_OUTPUT.PUT_LINE ( i ); EXIT; END LOOP;
```

```
   DBMS_OUTPUT.PUT_LINE ( i );                                  -- 10 ? 5 ? 1 ?
END;
/
```

4.2.4. Цикл по курсору (FOR)

FOR *запись_цикла* IN [*имя_курсора* | *явное_предложение_SELECT*]
LOOP
    программный код
END LOOP [*слово-комментарий*]

Объявление *записи_цикла* в PL/SQL выполняется автоматически, с типом *имя_курсора*%TYPE; самостоятельного объявления этой записи не требуется.

Кроме того, в цикле такого рода автоматически:

- открывается курсор
- извлекаются данные при прохождении цикла
- курсор закрывается по выходу из цикла

Примеры см. ниже.

4.3. Метки в циклах и в блоках

Для повышения надежности кода циклы можно размечать метками, например:

```
BEGIN
<<year_loop>>
  FOR year IN 1993 .. 2003 LOOP
  <<month_loop>>
    FOR month IN 1 .. 12 LOOP
       IF year_loop.year = 2000 AND month_loop.month = 1
                /*-- вместо обычного:  ... year = 2000 AND month = 1 */
                /*-- или даже так:  ... year_loop.month_loop.month = 1 */
                                              -- уточнили имена ради ясности
       THEN EXIT year_loop;                   -- выход из year_loop
       END IF;
       CONTINUE year_loop WHEN year = 1998;   -- а можно и: year_loop.year
    END LOOP month_loop;                      -- указали month_loop для ясности
    DBMS_OUTPUT.PUT_LINE ( 'Year ' || year || ' gone ...' );
  END LOOP year_loop;                         -- указали year_loop для ясности
  DBMS_OUTPUT.PUT_LINE ( 'Year 2000 have come !' );
END;
/
```

Этот же пример показывает, как имена меток можно использовать для доступа к переменным охватывающих циклов. Аналогично, и преследуя те же цели, размечать метками можно вложенные блоки:

```
<<outer>>
DECLARE i NUMBER := 1;
BEGIN
  <<inner>>
    DECLARE i NUMBER := 2;
    BEGIN
       DBMS_OUTPUT.PUT_LINE ( i );            -- outer или inner ?  1 или 2 ?
       DBMS_OUTPUT.PUT_LINE ( outer.i );
       DBMS_OUTPUT.PUT_LINE ( inner.i );
       DBMS_OUTPUT.PUT_LINE ( outer.inner.i ); -- ... то же самое
```

```
    END;
END;
/
```

Замечание. SQL Developer, в отличие от SQL*Plus, может увидеть в помечивании внешнего неименованного блока, как выше, ошибку Чтобы она не возникала, достаточно приведенный выше код обрамить «скобками» BEGIN … END;.

К сожалению, иногда размечивание циклов и блоков сопособно приводить к недоразумению из-за совпадения имен. Сравните примеры:

```
SQL> BEGIN DBMS_OUTPUT.PUT_LINE ( user ); END;
  2  /
SCOTT

PL/SQL procedure successfully completed.

SQL> DECLARE user VARCHAR2 ( 10 ) := 'tiger';
  2  BEGIN DBMS_OUTPUT.PUT_LINE ( user ); END;
  3  /
tiger

PL/SQL procedure successfully completed.

SQL> <<standard>> DECLARE user VARCHAR2 ( 10 ) := 'tiger';
  2  BEGIN DBMS_OUTPUT.PUT_LINE ( standard.user ); END;
  3  /
tiger

PL/SQL procedure successfully completed.

SQL> <<nonstandard>> DECLARE user VARCHAR2 ( 10 ) := 'tiger';
  2  BEGIN DBMS_OUTPUT.PUT_LINE ( standard.user ); END;
  3  /
SCOTT

PL/SQL procedure successfully completed.

SQL> <<standard>> DECLARE user VARCHAR2 ( 10 ) := 'tiger';
  2  BEGIN DBMS_OUTPUT.PUT_LINE ( sys.standard.user ); END;
  3  /
SCOTT

PL/SQL procedure successfully completed.
```

Проблема может (крайне редко) возникать, когда в качестве метки выбрано слово STANDARD. Название метки NONSTANDARD в примере выше имеет смысл «любое имя, отличное от STANDARD».

5. Подпрограммы

5.1. Локальные подпрограммы

Внутри блока PL/SQL могут объявляться собственные подпрограммы, невидимые снаружи блока, но в остальном обладающие практически всеми свойствами обычных (хранимых) подпрограмм.

Пример определения локальной функции:

```
DECLARE
  FUNCTION деньнедели ( d DATE ) RETURN VARCHAR2 IS
  BEGIN
    RETURN TO_CHAR ( d, 'Day', 'NLS_DATE_LANGUAGE=RUSSIAN' );
  END;
BEGIN
  DBMS_OUTPUT.PUT_LINE ( деньнедели ( SYSDATE ) );
  DBMS_OUTPUT.PUT_LINE ( деньнедели ( SYSDATE + 1 ) );
  DBMS_OUTPUT.PUT_LINE ( деньнедели ( SYSDATE + 2 ) );
END;
/
```

Объявление локальных подпрограмм должно выполняться в конце раздела объявлений. Допускается использование ранее определенных локальных переменных (того же уровня или выше):

```
DECLARE
  выдача VARCHAR2 ( 32767 );

  PROCEDURE добавить_в_выдачу ( строка VARCHAR2 ) IS
    BEGIN выдача := выдача || строка || CHR ( 10 );
    END;
BEGIN
  добавить_в_выдачу ( 'Синус 1' );
  добавить_в_выдачу ( 'будет:' );
  добавить_в_выдачу ( TO_CHAR ( SIN ( 1 ) ) );
  DBMS_OUTPUT.PUT_LINE ( выдача );
END;
/
```

В общем такое обращение ко внешним для подпрограммы переменным не приветствуется, однако в отдельных случаях оно бывает оправдано, подобно как в примере выше.

Упражнение. Переписать вычисление площади круга (а) с использованием функции, (б) с использованием процедуры.

Исходный код	***С использованием локальной функции***	***С использованием локальной процедуры***

DECLARE
  r NUMBER;
  PI CONSTANT NUMBER := 3.14;
  s NUMBER;
BEGIN
  r := 2;
  s := PI * r ** 2;
  DBMS_OUTPUT.PUT_LINE (s);
END;
	DECLARE
	 FUNCTION circle (r NUMBER)
	 ...
	BEGIN
	 ...
	 DBMS_OUTPUT.PUT_LINE ...
	END;
		DECLARE
		 c NUMBER;
		 PROCEDURE summa (r NUMBER)
		 ... c := ...
		BEGIN
		 circle (10);
		 DBMS_OUTPUT.PUT_LINE (c);
		END;

Подпрограммы в PL/SQL не запрещают рекурсивное обращение к себе, и при исполнении глубина рекурсии не регламентируется. Это накладывает на разработчика ПО ответственность слежения за недопущением «бесконечной» рекурсии. Не следует забывать, что рекурсивный алгоритм всегда может быть заменен на итерационный, и в этом случае отслеживать зацикленость становится легче.

Пример рекурсивной функции, вычисляющей сложные проценты:

```
DECLARE

FUNCTION complexpercent ( summa NUMBER, months NATURAL, percent NUMBER )
RETURN NUMBER
AS
  nextsum NUMBER;
BEGIN
  IF months > 0 THEN
     nextsum := complexpercent ( summa * ( 1 + percent / 100 / 12 )
                               , months - 1
                               , percent );
     RETURN TRUNC ( nextsum, 2 );
  ELSE
     RETURN summa;
  END IF;
END;

BEGIN
  DBMS_OUTPUT.PUT_LINE ( complexpercent ( 100, 12, 7 ) );
END;
/
```

Другой пример рекурсивной функции, вычисляющей дату (не в формате DATE !) через указанное количество суток:

```
DECLARE
TYPE yearmonthday IS RECORD ( year  BINARY_INTEGER
                            , month BINARY_INTEGER
                            , day   BINARY_INTEGER );
mydate yearmonthday;

FUNCTION enddate ( startdate yearmonthday, days BINARY_INTEGER )
RETURN yearmonthday
IS
retdate  yearmonthday;
nextdays BINARY_INTEGER;
BEGIN
  retdate.year  := startdate.year;
  retdate.month := startdate.month;
  retdate.day   := startdate.day + days;
  nextdays := retdate.day -
              CASE
              WHEN retdate.month IN ( 1, 3, 5, 7, 8, 10, 12 ) THEN 31
              WHEN retdate.month IN ( 4, 6, 9, 11 )           THEN 30
              WHEN ( retdate.year MOD 4 ) = 0                 THEN 29
              ELSE 28
              END;
  IF nextdays > 0 THEN
     retdate.day   := 0;
     retdate.month := retdate.month + 1;
     IF retdate.month > 12 THEN
        retdate.month := 1;
        retdate.year  := retdate.year + 1;
     END IF;
     RETURN enddate ( retdate, nextdays );
  ELSE RETURN retdate;
  END IF;
END;
BEGIN
  mydate.year  := 2009;
  mydate.month := 6;
  mydate.day   := 24;
  mydate := enddate ( mydate, 38 );
  DBMS_OUTPUT.PUT_LINE ( mydate.year );
  DBMS_OUTPUT.PUT_LINE ( mydate.month );
  DBMS_OUTPUT.PUT_LINE ( mydate.day );
```

```
END;
/
```

Употребление в программе рекурсии крайне желательно сопровождать знанием вопроса, так как оно не является однозначным. Общее знакомство с темой дает *Википедия*.

Использование локальных подпрограмм позволяет компактнее и надежнее организовать код. Однако обращение к подпрограмме требует расходов, и оптимизатор кода PL/SQL способен автоматически заменять такие обращения на «живой» текст подпрограммы. С версии 11 прагма INLINE позволяет санкционировать или запретить такую замену явочным порядком:

```
  PRAGMA INLINE ( добавить_в_выдачу, 'YES' );
  добавить_в_выдачу ( 9 );          -- будет подставлен код
  DBMS_OUTPUT.PUT_LINE ( выдача );
  добавить_в_выдачу ( 10 );         -- будет подставлен вызов процедуры
  ...
```

5.2. Повторение в подпрограммах ранее задействованных имен

Локальные подпрограммы могут повторять имена подпрограмм, внешних по отношению к своему блоку (то есть ранее уже задействованные имена), в том числе так называемых «системных» программ. Областью действия такого «переопределения» является текущий блок и все в него вложенные. Пример:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ( sin ( 1234 ) );
  DECLARE
    FUNCTION sin ( x NUMBER ) RETURN NUMBER IS BEGIN RETURN 123; END;
  BEGIN
    DBMS_OUTPUT.PUT_LINE ( sin ( 1234 ) );
    BEGIN
      DBMS_OUTPUT.PUT_LINE ( sin ( 1234 ) );          -- «местный» синус ...
      DBMS_OUTPUT.PUT_LINE ( standard.sin ( 1234 ) );-- «системный» синус
    END;
  END;
  DBMS_OUTPUT.PUT_LINE ( sin ( 1234 ) );
END;
/
```

Пример показывает, что к подпрограммам, «переопределенным» локально, можно-таки обращаться. Если «переопределяется» подпрограмма из охватывающего блока, обратиться к ней можно, снабдив охватывающий блок меткою, которой и уточнить обращение к подпрограмме (см. выше).

5.3. Предваряющие (forward) объявления

Подпрограммы в PL/SQL допускают взаимный вызов (взаимную рекурсию). Технически это порождает проблему описания, и для преодоления этой проблемы в PL/SQL используется схема предваряющего объявления, например:

```
DECLARE
   PROCEDURE a (f NUMBER);          -- порядок может
   PROCEDURE b (f NUMBER);          --                быть любым

   PROCEDURE a (f NUMBER) IS содержание;          -- порядок может
   PROCEDURE b (f NUMBER) IS содержание;          --                быть любым
BEGIN a ( 0.5 ); END;
/
```

Взаимный вызов может быть и опосредованым, через цепочку обращений подпрограмм друг к другу:

```
DECLARE
FUNCTION деньнедели ( d DATE ) RETURN VARCHAR2 IS
BEGIN RETURN TO_CHAR ( d, 'Day', 'NLS_DATE_LANGUAGE=RUSSIAN' ) || ': ';
END;

PROCEDURE копияуровня0  ( время DATE, срок POSITIVE );
PROCEDURE копия1уровня1 ( время DATE, срок POSITIVE );
PROCEDURE копия2уровня1 ( время DATE, срок POSITIVE );
PROCEDURE копия3уровня1 ( время DATE, срок POSITIVE );

PROCEDURE копияуровня0 ( время DATE, срок POSITIVE ) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ( деньнедели ( время ) || 'Копия на уровне 0' );
  IF срок > 1 THEN копия1уровня1 ( время + 1, срок - 1 ); END IF;
END;
PROCEDURE копия1уровня1 ( время DATE, срок POSITIVE ) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ( деньнедели ( время ) || 'Копия 1 на уровне 1' );
  IF срок > 1 THEN копия2уровня1 ( время + 1, срок - 1 ); END IF;
END;
PROCEDURE копия2уровня1 ( время DATE, срок POSITIVE ) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ( деньнедели ( время ) || 'Копия 2 на уровне 1' );
  IF срок > 1 THEN копия3уровня1 ( время + 1, срок - 1 ); END IF;
END;
PROCEDURE копия3уровня1 ( время DATE, срок POSITIVE ) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ( деньнедели ( время ) || 'Копия 3 на уровне 1' );
  IF срок > 1 THEN копияуровня0 ( время + 1, срок - 1 ); END IF;
END;

BEGIN
   DBMS_OUTPUT.PUT_LINE ( 'График снятия инкрементальных копий БД:' );
   копияуровня0 ( SYSDATE, 10 );
END;
/
```

Возможный ответ:

```
График снятия инкрементальных копий БД:
Среда      : Копия на уровне 0
Четверг    : Копия 1 на уровне 1
Пятница    : Копия 2 на уровне 1
Суббота    : Копия 3 на уровне 1
Воскресенье: Копия на уровне 0
Понедельник: Копия 1 на уровне 1
Вторник    : Копия 2 на уровне 1
Среда      : Копия 3 на уровне 1
Четверг    : Копия на уровне 0
Пятница    : Копия 1 на уровне 1
```

При исполнении кода число итераций взаимных вызовов не регламентируется. Это накладывает на разработчика ответственность слежения за недопущением зацикливания.

5.4. Повторение имен подпрограмм одного уровня (overloading)

В PL/SQL имена подпрограмм, определенных на одном уровне, в одном разделе объявлений, могут повторяться. Однако для этого заголовки подпрограмм (помимо имени) должны различаться, чтобы компилятор имел возможность при транслировании программы подставить обращение к нужной процедуре

или функции. Поскольку имена таких подпрограмм совпадают, различия в их заголовках должны касаться количества и типов параметров.

Пример:

```
DECLARE
  FUNCTION a ( n NUMBER ) RETURN NUMBER IS BEGIN RETURN 2; END;
  FUNCTION a ( n BINARY_INTEGER ) RETURN NUMBER IS BEGIN RETURN 3; END;
  FUNCTION a ( s VARCHAR2 ) RETURN VARCHAR2 IS BEGIN RETURN 'b'; END;
  FUNCTION a ( d DATE ) RETURN VARCHAR2 IS
    BEGIN RETURN TO_CHAR ( d, 'Day', 'NLS_DATE_LANGUAGE=RUSSIAN' ); END;
  PROCEDURE a IS BEGIN DBMS_OUTPUT.PUT_LINE ( a ( 1 ) ); END;
BEGIN
  DBMS_OUTPUT.PUT_LINE ( a ( 'a' ) );     -- 'b'
  DBMS_OUTPUT.PUT_LINE ( a ( 1 ) );       -- 3
  DBMS_OUTPUT.PUT_LINE ( a ( 1.0 ) );     -- 2
  DBMS_OUTPUT.PUT_LINE ( a ( SYSDATE ) ); -- День недели
  a;                                      -- 3
END;
/
```

Здесь решение о том, к какой из подпрограмм A обратиться, компилятор примет, выяснив тип параметра.

Более жизненный пример:

```
DECLARE
  выдача VARCHAR2 ( 32767 );

  PROCEDURE добавить_в_выдачу ( строка VARCHAR2 ) IS
    BEGIN выдача := выдача || строка || CHR ( 10 );
    END;
  PROCEDURE добавить_в_выдачу ( число NUMBER ) IS
    BEGIN добавить_в_выдачу ( TO_CHAR ( число ) ); -- строчный вариант
    END;
BEGIN
  добавить_в_выдачу ( 'Синус 1' );
  добавить_в_выдачу ( 'будет:' );
  добавить_в_выдачу ( 1 );
  DBMS_OUTPUT.PUT_LINE ( выдача );
END;
/
```

Достаточными отличительными признаками для подмены *не* являются:

- отличие функций только типом результата (RETURN)
- отличие типов параметров в пределах однородного семейства (CHAR и VARCHAR2, INTEGER и NUMBER и т. д.)
- отличие параметров только своими режимами (IN, OUT, IN OUT; см. далее)

6. Взаимодействие с базой данных: статический SQL

Взаимодействие программы на PL/SQL с БД может осуществляться тремя разными методами:

1) с помощью статических явно указанных операторов SQL
2) с помощью курсоров
3) с помощью динамически порождаемых операторов SQL

В этой главе рассмотрен первый метод. Важное его расширение будет рассмотрено после темы «Коллекции».

6.1. Предложения SQL в качестве операторов PL/SQL

Элементами PL/SQL являются операторы SQL по работе с данными существующих таблиц:

- категорий DML (INSERT, UPDATE, DELETE и MERGE) и QL (SELECT), подпадающие под определение «встроенного SQL» в стандарте ANSI, однако в рамках диалекта SQL Oracle;
- управления транзакциями и блокировками.

С точки зрения PL/SQL они являются статическими конструкциями, так как их формулировки однозначно зафиксированы в тексте программы (в отличие от «динамического SQL», о котором ниже).

Вид DML-операторов для использования в программах PL/SQL:

SELECT *список_столбцов* **INTO** ***список_переменных_или_запись***
FROM *отбор_строк_как_обычно* [FOR UPDATE ...];

INSERT INTO *табличное_выражение* [(*перечень_столбцов*)] {VALUES (*список_значений*) | *оператор_SELECT*}
[**RETURNING** ***список_выражений*** **INTO** ***список_переменных_или_запись***];

UPDATE *табличное_выражение* SET *присвоение* [WHERE *условие*]
[**RETURNING** ***список_выражений*** **INTO** ***список_переменных_или_запись***];

DELETE FROM *табличное_выражение* [WHERE *условие*]
[**RETURNING** ***список_выражений*** **INTO** ***список_переменных_или_запись***];

MERGE INTO *остальные_конструкции_предложения_MERGE_как_обычно*;

(Оператор MERGE — начиная с версии 9).

Следуя стандарту ANSI SQL, *имя_переменной* выше может быть именем скалярной переменной или записи, а *список_переменных* — список скаляров. Начиная с версии 9 СУБД, это могут быть и массивы, однако в этом случае потребуется специальная синтаксическая конструкция, о которой речь позже.

Пример использования:

```
DECLARE
  employee emp.ename%TYPE;
BEGIN
  SELECT ename INTO employee FROM emp WHERE empno = 7369;
  DBMS_OUTPUT.PUT_LINE ( 'Employee ' || employee || ' selected' );

  INSERT INTO emp ( empno, ename ) VALUES ( 1111, 'BUSH' )
  RETURNING ename INTO employee;
  DBMS_OUTPUT.PUT_LINE ( 'Employee ' || employee || ' inserted' );

  UPDATE emp SET ename = 'LADEN' WHERE empno = 1111
```

```
  RETURNING ename INTO employee;
  DBMS_OUTPUT.PUT_LINE ( 'Employee ' || employee || ' replaced him' );

  DELETE FROM emp WHERE empno = 1111
  RETURNING ename INTO employee;
  DBMS_OUTPUT.PUT_LINE ( 'Employee ' || employee || ' deleted' );
END;
/
```

Еще пример:

```
CREATE SEQUENCE emp_empno_seq MINVALUE 1 MAXVALUE 9999
;
CREATE SEQUENCE dept_deptno_seq MINVALUE 1 MAXVALUE 99 START WITH 41
;

DECLARE
 out VARCHAR2 ( 100 );
 dno dept.deptno%TYPE;
BEGIN
   INSERT INTO dept ( deptno, dname, loc )
            VALUES ( dept_deptno_seq.NEXTVAL, 'WHOUSE', 'WASHINGTON' )
   RETURNING deptno INTO dno
   ;
   INSERT INTO emp ( empno, ename, deptno )
           VALUES ( emp_empno_seq.NEXTVAL, 'BUSH', dno )
   RETURNING 'Inserted number was ' || TO_CHAR ( empno ) INTO out
   ;
   DBMS_OUTPUT.PUT_LINE ( out );

   DELETE FROM emp WHERE ename = 'BUSH'
   RETURNING 'Employee '
          || ename
          || ' is deleted for his job is '
          || NVL ( job, 'NULL' )
   INTO out
   ;
   DELETE FROM dept WHERE deptno = dno
   ;
   DBMS_OUTPUT.PUT_LINE ( out );
END;
/
```

В случае INSERT и UPDATE фраза RETURNING позволяет эффективно получить в программу занесенное в БД значение для последующей обработки или же проверки присвоенного, однако не стоит забывать о ловушках сравнения с NULL, воспроизведенных в PL/SQL из SQL, и о неявных преобразованиях, способных сопровождать занесение данных в БД:

```
DECLARE
  newcomm  emp.comm%TYPE := NULL;
  realcomm emp.comm%TYPE;
  newcommm NUMBER;
BEGIN
  --  пример 1: сравнение с NULL
  UPDATE emp SET comm = newcomm WHERE ename = 'SMITH'
  RETURNING comm INTO realcomm
  ;
  CASE
    WHEN newcomm = realcomm
      THEN DBMS_OUTPUT.PUT_LINE
           ( 'Синтаксически правильно, но выдачи нет.  Как же так ?' );
    WHEN newcomm IS NULL AND realcomm IS NULL
      THEN DBMS_OUTPUT.PUT_LINE
           ( 'Содержательно правильное сравненение с NULL.' );
```

```
    ELSE NULL;
  END CASE;

  --  пример 2: самопроизвольное округление чисел при помещении в БД
  newcommm := 100.005;
  UPDATE emp SET comm = newcommm WHERE ename = 'SMITH'
  RETURNING comm INTO realcomm
  ;
  CASE
    WHEN newcommm = realcomm
      THEN DBMS_OUTPUT.PUT_LINE
           ( 'Правильно?... Но выдачи нет! Как же так?' );
    WHEN ROUND ( newcommm, 2 ) = realcomm
      THEN DBMS_OUTPUT.PUT_LINE
           ( 'Комиссионные хранятся с точностью до сотых.' );
    ELSE NULL;
  END CASE;
END;
/
```

С версии 10 разрешено указывать во фразе RETURNING функцию агрегирования, которая соблюдает скалярность результата, но уже не требует однострочности действия команд INSERT/UPDATE/DELETE … RETURNING …, например:

```
DECLARE s NUMBER;
BEGIN
  UPDATE emp
  SET    sal = sal * 1.5
  WHERE  comm IS NOT NULL
  RETURNING sum ( sal ) INTO s
  ;
  DBMS_OUTPUT.PUT_LINE ( s );
END;
/

ROLLBACK;
```

Первоначально такая возможность реализована с ошибкой: RETURNING COUNT (*) всегда возвращает 0, а не действительное количество строк. При том же COUNT (*выражение*) вычисляется как положено. Ошибка исправлена в версии 11.

6.2. Использование записей вместо списка скаляров

В версии 9.2 появилась возможность добавляемые или изменяемые значения указывать не списком скалярных переменных, а записью:

```
DECLARE
  employee emp%ROWTYPE;
BEGIN
  SELECT * INTO employee FROM emp WHERE empno = 7369;

  employee.empno := 1111;

  INSERT INTO emp VALUES employee;
-- или, например, так:
--INSERT INTO emp VALUES employee RETURNING ename INTO employee.ename;

  employee.empno := 2222;

  UPDATE emp SET ROW = employee WHERE empno = 1111;

  DELETE FROM emp WHERE empno = 2222;
```

```
END;
/
```

(Аналогичное использование записей во фразе INTO предложения SELECT и во фразе RETURNING допускалось и прежде).

Такое употребление накладывает ограничения на тип используемой записи: она может состоять только из скаляров (и в частности, не может быть вложенной).

6.3. Регулирование изменений в БД

В программах PL/SQL можно непосредственно (статически) использовать операторы (COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION), а также оператор LOCK TABLE.

6.3.1. Управление транзакциями

Ниже перечисляются операторы PL/SQL, позволяющие управлять выполнением транзакций из программы. Некоторые дополнительные возможности управления транзакциями имеются во встроенном пакете DBMS_TRANSACTION, а некоторые возможности по управлению взаимодействием транзакций доступны путем использования встроенного пакета DBMS_LOCK.

COMMIT [WORK] [COMMENT *текст*];

Слово WORK используется только для читаемости кода, а COMMENT *текст* может использоваться при разборе незавершившихся распределенных транзакций.

SAVEPOINT имя_точки_сохранения;

Создает точку сохранения в пределах транзакции. Если оператором SAVEPOINT повторить с тем же именем позже, указанная точка сохранения «переместится» вперед.

ROLLBACK [WORK] [TO SAVEPOINT] *имя_точки_сохранения*

Позволяет откатить действия транзакции назад до сделанной заранее точки сохранения.

SET TRANSACTION [READ WRITE | READ ONLY | ISOLATION LEVEL SERIALIZABLE];

Задание типа транзакции.

SET TRANSACTION USE ROLLBACK SEGMENT *имя_сегмента*;

Приписывает только начавшейся транзакции конкретный сегмент отката, что часто используется в интенсивно модифицирующих транзакциях (действует при использовании сегментов отката вместо сегментов отмены, ставших умолчательными с версии 9).

6.3.2. Блокировки таблицы

Блокировки таблицы выполняются оператором LOCK TABLE:

LOCK TABLE список_таблиц IN *вид_блокировки* MODE [NOWAIT];

Явно блокирует перечисленные в списке через запятую таблицы (полностью, а не на уровне строки) в указанном режиме. Если указано NOWAIT, то СУБД не переведет остальные запросы к таблице в состояние ожидания, пока блокировка не будет снята (стандартная ситуация), а выдаст сообщение об ошибке ORA-00054.

6.3.3. Ожидание освобождения строк таблицы в программе

Когда транзакции, вносящие изменения в таблицу в приложении, сравнительно коротки, программа, перед внесением собственных изменений, имеет возможность подождать, пока все конкурирующие транзакции, разместившие свои замки на таблице, и тем мешающие работать, не завершатся. Делается это с помощью системной процедуры, появившейся в версии Oracle 11.2. Пример:

```
DECLARE
timeout BINARY_INTEGER := NULL;
scn     NUMBER         := NULL;

BEGIN
IF
   NOT DBMS_UTILITY.WAIT_ON_PENDING_DML (
         tables  => 'emp, dept'
       , timeout => timeout
       , scn     => scn
       )
THEN
   RAISE_APPLICATION_ERROR ( -20100, 'Waiting for DML timeout.' );
END IF;

/* ... изменяем данные ... */
END;
/
```

Подставляя в код, подобный приведенному выше, значения для TIMEOUT (в секундах) и для SCN, можно подстроиться под конкретные потребности.

6.3.4. Автономные транзакции

С версии 8.1 отдельные блоки программы можно выполнять рамках того же сеанса, но в отдельной, в автономной транзакции, на которую, таким образом, не будут распространяться откаты или фиксации, осуществляемые в основной транзакции:

```
PROCEDURE main IS
BEGIN
        COMMIT;
        UPDATE …                -- начало основной транзакции
        DELETE …
        some_procedure;         -- начало автономной транзакции
        SELECT …
        INSERT …
        ROLLBACK;               -- конец основной транзакции
END;

PROCEDURE some_procedure IS
        PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN                           -- основная транзакция приостановлена
        SELECT …
        INSERT …                -- начало автономной транзакции
        UPDATE …
        DELETE …
```

```
    COMMIT;          -- конец автономной транзакции; фиксируются только
                     -- изменения, сделанные в рамках этой процедуры;
END;                 -- основная транзакция возобновляется
```

Правила видимости данных, измененных в БД основной и автономной транзакциями, соответствуют обычным правилам видимости для двух самостоятельных транзакций. То же касается регулирования изменений одних и тех же данных. Если основная транзакция заблокировала какую-нибудь таблицу, может возникнуть тупиковая ситуация. Избежать ее можно, используя указание NOWAIT или WAIT *n* в предложениях LOCK TABLE и SELECT … FOR UPDATE. Блокировка может возникнуть в том числе вследствие существования межтабличной связи «внешний ключ»; см. пример ниже.

Упражнение. Что получим в результате следующих действий ? Проверить в SQL*Plus.

Что видно из автономной транзакции:

```
SET SERVEROUTPUT ON

-- изменяем зарплату:
UPDATE emp SET sal = sal * 10 WHERE ename = 'SMITH';

-- автономная транзакция видит старое значение:
DECLARE
 PRAGMA AUTONOMOUS_TRANSACTION;
 salary emp.sal%TYPE;
BEGIN
 SELECT sal INTO salary FROM emp where ename = 'SMITH';
 DBMS_OUTPUT.PUT_LINE ( salary );
END;
/

-- а основная транзакция продолжает видеть новое значение:
SELECT sal FROM emp where ename = 'SMITH';

ROLLBACK;
```

Построение транзакций:

```
DECLARE
 PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
 INSERT INTO emp ( empno, deptno ) VALUES ( 2222, 40 );
END;
/
-- почему возникла ошибка ?

DECLARE
 PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
 INSERT INTO emp ( empno, deptno ) VALUES ( 2222, 40 );
 ROLLBACK;
END;
/
-- OK.

INSERT INTO dept ( deptno ) VALUES ( 50 );

DECLARE
 PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
 INSERT INTO emp ( empno, deptno ) VALUES ( 2222, 50 );
 ROLLBACK;
END;
```

```
/
-- почему возникла ошибка ?

ROLLBACK;
```

Закомментируйте INSERT INTO dept …. *-- почему возникла новая ошибка ?*

```
INSERT INTO dept ( deptno ) VALUES ( 50 );

DECLARE
 PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
 INSERT INTO emp ( empno, deptno ) VALUES ( 2222, 40 );
 ROLLBACK;
END;
/
-- OK.

ROLLBACK;
```

Что видно из основной транзакции:

```
-- исходная зарплата:
SELECT sal FROM emp WHERE ename = 'SMITH';

-- изменяем зарплату в автономной транзакции:
DECLARE
 PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
 UPDATE emp SET sal = sal * 10 WHERE ename = 'SMITH';
 COMMIT;
END;
/

-- увидим это изменение из основной ?
SELECT sal FROM emp WHERE ename = 'SMITH';

-- такая же проверка, только в основной транзакции READ ONLY:
SET TRANSACTION READ ONLY;

DECLARE
 PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
 UPDATE emp SET sal = sal / 10 WHERE ename = 'SMITH';
 COMMIT;
END;
/

-- а теперь увидим изменение ?
SELECT sal FROM emp WHERE ename = 'SMITH';

ROLLBACK;
-- вышли из транзакции READ ONLY

SELECT sal FROM emp WHERE ename = 'SMITH';
```

Объяснить разницу в результатах до ROLLBACK и после.

Тот же эффект даст использование транзакции ISOLATION LEVEL SERIALIZABLE (но, в отличие от READ ONLY, не будет препятствовать внесению изменений в БД из основной).

Примеры употребления автономных транзакций:

- программирование занесения сведений в журнальную таблицу, например о попытках обратиться к контролируемому ресурсу;
- в теле триггерных процедур:
  - при необходимости завершить транзакцию через COMMIT или ROLLBACK;
  - для возможности выполнить чтение данных таблицы в теле строчной триггерной процедуры, связанной с этой таблицей;
- обращение к подпрограммам без последствий на работу основной программы;
- изменение в функции, вызываемой из SELECT, таблицы-источника данных для этого оператора SELECT.

6.4. Статическое указание запроса при работе с БД через курсор

Существующий в PL/SQL особый вид цикла — цикл по курсору — допускает также статическое указание предложения SELECT, не во встроенном варианте, а в предложении цикла. Оно может выглядеть следующим образом:

```
FOR department IN ( SELECT * FROM dept ) LOOP
   DBMS_OUTPUT.PUT_LINE ( department.dname );
END LOOP;
```

Кроме того, статическое указание текста запроса допускает оператор открытия курсора по ссылке, например:

```
OPEN department FOR SELECT * FROM dept;
...
```

Замечательно, что коль скоро речь идет уже о работе с БД через курсор, предложения SELECT в этих случаях не связаны никакими ограничениями множественности результата. Однако работа с такого рода конструкциями имеет много общего с обработкой «обычных» курсоров (определяемых в программе как таковых самостоятельно), и потому рассматривается далее в разделе о курсорах.

7. Использование курсоров

Курсоры также могут быть причислены к статической технике формулирования запросов SQL в программе. Основная задача курсорного механизма в PL/SQL — позволить программе обрабатывать многострочные операторы SELECT. Курсор — именованный указатель на область, выделяемую СУБД для обработки (каждого) запроса SQL. Иногда под курсором более узко понимают указатель на «очередную» строку для извлечения с сервера в программу. В программе можно заводить переменную «типа курсор».

7.1. Явные курсоры

Курсоры для SQL-предложений, явно объявляемые в разделе описаний блока PL/SQL.

7.1.1. Объявление явных курсоров

Три возможных вида объявления явных курсоров.

Курсоры без параметров:

CURSOR employee_cur IS SELECT ename FROM emp;

Определение курсора имеет право ссылаться на локальную переменную, например:

department NUMBER (2);
CURSOR employee_cur IS SELECT ename FROM emp WHERE deptno = **department**;

В общем, однако, лучше использовать в таких случаях курсоры с параметрами:

CURSOR employee_cur **(department NUMBER)** IS
SELECT ename FROM emp WHERE deptno = **department**;

В этом примере можно было бы написать (department **NUMBER DEFAULT 10**), и тогда бы параметр получил умолчательное значение. Тогда фактическое значение при открытии курсора будет разрешено не указывать.

Заголовок курсора в интерфейсной части пакета (позволяет спрятать реализацию курсора в тело пакета, не предъявляя его в описание):

CURSOR employee_cur **RETURN emp%ROWTYPE**;

(В последнем случае в теле пакета может быть указано что-то вроде:

CURSOR employee_cur RETURN emp%ROWTYPE IS SELECT * FROM emp;)

7.1.2. Открытие явных курсоров

OPEN *имя_курсора* [(*список выражений для параметров*)];

Если параметры имеются, но все допускают умолчание, можно написать (с равным успехом) OPEN *имя_курсора* () или же просто OPEN *имя_курсора*.

По команде OPEN будет вычислен план выполнения запроса; к участкам памяти в курсоре будут привязаны связные переменные и параметры курсора; будет вычислен формат результата; указатель результата встанет на первую строку выдачи.

В цикле FOR операция OPEN выполняется неявно.

7.1.3. Извлечение результата через явный курсор

FETCH *имя_курсора* INTO *запись_или_список_переменных*;

Извлекается ровно одна строка результата.

7.1.4. Закрытие явного курсора

CLOSE *имя_курсора*;

Закрытие курсора после исчезновения в нем необходимости освобождает память для других запросов. Курсор, открытый в анонимном блоке или в блоке процедуры или функции закрывается автоматически по окончанию работы блока. Глобальные курсоры пакета автоматически закрываются только по окончанию сеанса связи с БД.

7.1.5. Отсутствие запрета изменений таблиц при открытом курсоре

Открытый курсор не запрещает изменять данные таблиц, выбираемых курсором, и даже удалять их строки, так как на курсор распространяется правило целостности операции SQL:

```
DECLARE
 CURSOR curs_a IS SELECT deptno FROM dept ORDER BY deptno DESC;
 CURSOR curs_b IS SELECT deptno FROM dept ORDER BY deptno DESC;
 d# dept.deptno%TYPE;

BEGIN
 OPEN curs_a;
 DELETE FROM dept WHERE deptno = 40;
 FETCH curs_a INTO d#; DBMS_OUTPUT.PUT_LINE ( 'cursor A:' || d# ); -- 40

 OPEN curs_b;

 FETCH curs_a INTO d#; DBMS_OUTPUT.PUT_LINE ( 'cursor A:' || d# ); -- 30
 FETCH curs_b INTO d#; DBMS_OUTPUT.PUT_LINE ( 'cursor B:' || d# ); -- 30
 FETCH curs_a INTO d#; DBMS_OUTPUT.PUT_LINE ( 'cursor A:' || d# ); -- 20
 FETCH curs_b INTO d#; DBMS_OUTPUT.PUT_LINE ( 'cursor B:' || d# ); -- 20
 FETCH curs_a INTO d#; DBMS_OUTPUT.PUT_LINE ( 'cursor A:' || d# ); -- 10
 FETCH curs_b INTO d#; DBMS_OUTPUT.PUT_LINE ( 'cursor B:' || d# ); -- 10

 CLOSE curs_a;
 CLOSE curs_b;
END;
/

SELECT * FROM dept;

ROLLBACK;
```

(Обратите внимание, что целостность данных, выдаваемых курсором, не пострадала).

Однако для изменения *текущей* строки, выбираемой с помощью курсора, есть более эффективное средство (см. фразу WHERE CURRENT OF в блокирующем курсоре ниже).

7.1.6. Атрибуты для явных курсоров

У каждого явно открываемого курсора имеются для использования в программе атрибуты:

Атрибут	*Значение*
%ISOPEN	TRUE, если курсор открыт и FALSE — если нет
%FOUND	Исключительная ситуация INVALID_CURSOR, если курсор не открыт оператором OPEN или закрыт оператором CLOSE; NULL — перед первым выполнением FETCH TRUE — после успешного выполнения FETCH FALSE — если FETCH не сумел выдать строку
%NOTFOUND	Исключительная ситуация INVALID_CURSOR, если курсор не открыт оператором OPEN или закрыт оператором CLOSE; NULL — перед первым выполнением FETCH FALSE — после успешного выполнения FETCH TRUE — если FETCH не сумел выдать строку
%ROWCOUNT	Исключительная ситуация INVALID_CURSOR, если курсор не открыт оператором OPEN или закрыт оператором CLOSE; общее количество строк, извлеченных по результату последней операции FETCH

Синтаксис использования атрибута: *имя_курсора%атрибут.*

7.2. Несколько примеров использования циклов и курсоров

Ниже приводится несколько типичных примеров разных использования курсоров и циклов разными способами.

Пример «низкоуровневого» использования:

```
DECLARE
   CURSOR dept_cursor IS SELECT dname FROM dept;
   dname_variable dept.dname%TYPE;
BEGIN
   OPEN dept_cursor;
   LOOP
      FETCH dept_cursor INTO dname_variable;
      EXIT WHEN dept_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE ( dname_variable );
   END LOOP;
   CLOSE dept_cursor;
END;
/
```

Аналогичный курсор типа «строка»:

```
DECLARE
   CURSOR dept_cursor IS SELECT * FROM dept;
   dept_record dept_cursor%ROWTYPE;
BEGIN
   OPEN dept_cursor;
   LOOP
      FETCH dept_cursor INTO dept_record;
      EXIT WHEN dept_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE ( dept_record.dname );
   END LOOP;
   CLOSE dept_cursor;
END;
/
```

Пример перебора результата запроса SQL более высокого уровня:

```
BEGIN
   FOR department IN ( SELECT * FROM dept ) LOOP
      DBMS_OUTPUT.PUT_LINE ( department.dname );
   END LOOP;
END;
/
```

То же самое с помощью явного курсора. Обратите внимание, что в этом случае появляется возможность обратиться к атрибуту курсора:

```
DECLARE
   CURSOR departments IS SELECT * FROM dept;
BEGIN
   FOR department IN departments LOOP
      DBMS_OUTPUT.PUT_LINE ( departments%ROWCOUNT || department.dname );
   END LOOP;
END;
/
```

Пример перебора для курсора с параметром:

```
DECLARE
   CURSOR coworkers ( dnumber NUMBER ) IS
      SELECT ename FROM emp WHERE deptno = dnumber;
BEGIN
   FOR employee IN coworkers ( 20 ) LOOP
      DBMS_OUTPUT.PUT_LINE ( employee.ename );
   END LOOP;
END;
/
```

То же самое, но параметр с умолчанием:

```
DECLARE
   CURSOR coworkers ( dnumber NUMBER := 10 ) IS
      SELECT ename FROM emp WHERE deptno = dnumber;
BEGIN
   FOR employee IN coworkers ( 20 ) LOOP
      DBMS_OUTPUT.PUT_LINE ( employee.ename );
   END LOOP;

   FOR employee IN coworkers (  ) LOOP        -- скобки можно и опустить
      DBMS_OUTPUT.PUT_LINE ( employee.ename );
   END LOOP;
END;
/
```

Пример перебора по ссылке на курсор см. выше («Встроенный динамический SQL»).

7.3. Курсоры с блокировкой строк таблицы

7.3.1. Предложение SELECT … FOR UPDATE

Используется для упреждающей блокировки набора строк, подпадающего под действие SELECT, при выполнении OPEN для курсора. Блокировка будет снята ближайшим оператором COMMIT или ROLLBACK.

Синтаксис:

SELECT … FROM … FOR UPDATE [OF список_имен_столбцов] [NOWAIT];

Если указан список_имен_столбцов, то блокировка будет касаться только строк, на которые этот список распространяется. Пример:

```
DECLARE
CURSOR clerks_in_new_york
IS
   SELECT empno, sal, comm
   FROM   emp INNER JOIN dept
          USING ( deptno )
   WHERE  job = 'CLERK'
     AND  loc = 'NEW YORK'
   FOR UPDATE OF emp.sal
;
BEGIN NULL; END;
/
```

Здесь при открытии (OPEN) курсора будут заблокированы только строки таблицы EMP, так как только ее столбцы (в единственном числе в данном случае) приведены в конструкции FOR UPDATE.

Указание FOR UPDATE возможно и в цикле по курсору, объявленному по месту обращения:

```
BEGIN
   FOR department IN ( SELECT * FROM dept FOR UPDATE ) LOOP
      DBMS_OUTPUT.PUT_LINE ( department.dname );
   END LOOP;
END;
/
```

7.3.2. Предосторожности употребления курсоров с блокировкой

Протяженная обработка курсоров с блокировкой в программе способна порождать нежелательные эффекты, отсутствующие при выдаче обычного предложения SELECT с блокировкой. Так, выдача команды завершения транзакции *до* закрытия такого курсора сделает невозможным его дальнейший просмотр. Например, выдав:

```
BEGIN
   FOR department IN ( SELECT * FROM dept FOR UPDATE ) LOOP
      DBMS_OUTPUT.PUT_LINE ( department.dname );
      /*
          изменяем данные в таблицах: DEPT или любой другой
      */
      COMMIT;
   END LOOP;
END;
/
```

получим на второй же записи об отделе сообщение об ошибке:

```
ACCOUNTING
BEGIN
*
ERROR at line 1:
ORA-01002: fetch out of sequence
ORA-06512: at line 2
```

При необходимости изменять данные в БД по мере чтения строк курсора следует выполнять эти изменения в автономной транзакции и использовать с этой целью подпрограмму, оттраслированную с прагмой AUTONOMOUS TANSACTION. Для изменения строк таблицы, выбираемой курсором (в примере выше — таблицы DEPT), применяется отдельная техника, о которой ниже.

7.3.3. Изменение строк, выбираемых курсором

Если объявлен курсор с конструкцией FOR UPDATE, к текущей выбранной строке разрешено применять операторы UPDATE и DELETE, если снабдить их особым указанием во фразе WHERE вместо условного выражения:

[UPDATE | DELETE] … WHERE **CURRENT OF *имя_курсора***;

Локализация «текущей» строки в этом случае будет осуществляться по ROWID, автоматически конструируемому фразой FROM предложения SELECT. Такие техника работы и оформление предусмотрены стандартом SQL.

Упражнение. Выполнить следующий код и объяснить его действие:

```
SAVEPOINT start_perestroika;

DECLARE
CURSOR emp_cur IS
   SELECT job, deptno
   FROM   emp
   WHERE  job = 'CLERK' OR deptno = 10
   FOR UPDATE NOWAIT;

BEGIN
FOR emp_rec IN emp_cur LOOP
   CASE
   WHEN emp_rec.job = 'CLERK' THEN
          DELETE FROM emp WHERE CURRENT OF emp_cur;
   WHEN emp_rec.deptno = 10 THEN
          UPDATE emp SET sal = sal * 10 WHERE CURRENT OF emp_cur;
   ELSE NULL;
   END CASE;
END LOOP;
END;
/

SELECT * FROM emp ORDER BY deptno, job  -- результат;

ROLLBACK TO SAVEPOINT start_perestroika;

SELECT * FROM emp ORDER BY deptno, job  -- контроль;
```

Несмотря на то, что SELECT … FOR UPDATE в состоянии заблокировать строки одновременно разных таблиц из запроса, внесение изменений с помощью указания CURRENT OF не дает полной свободы программисту и ограничено правилами выполнения UPDATE и DELETE к представлению (view), как если бы оно существовало на основе этого предложения SELECT. Так, если SELECT — это соединение, то изменить возможно будет только строки одной из соединяемых таблиц. Если применяется UPDATE, SELECT в курсоре должен извлекаться непреобразованный первичный ключ; если DELETE — то помимо этого удалению смогут подлежать только строки из первой по порядку таблицы во фразе FROM, и так далее. Если эти правила не соблюдать, то операторы UPDATE и DELETE для текущей строки (CURRENT OF …) могут молча игнорироваться.

Указанные ограничения несколько снижают полезные качества локализации изменяемых строк с помощью WHERE CURRENT OF.

7.4. Неявные курсоры

Создаются внутри СУБД автоматически при указании в тексте операторов INSERT, UPDATE, DELETE и SELECT INTO. Для этих операторов в программе не нужны OPEN, FETCH и CLOSE, так как СУБД выполняет эти действия самостоятельно.

В случае SELECT INTO возможны исключительные ситуации, при возникновении которых управление передается в раздел обработки исключительных ситуаций (приводятся их предопределенные имена):

NO_DATA_FOUND — если не извлечено ни единой строки
TOO_MANY_ROWS — если извлечено более одной строки

С неявными курсорами связаны следующие атрибуты:

Атрибут	***Значение***
%ISOPEN	Всегда FALSE, так как курсор открывается и далее обрабатывается неявно
%FOUND	NULL — перед выполнением SQL-оператора TRUE — если оператор обработал хотя бы одну строку FALSE — если оператор не обработал ни одной строки
%NOTFOUND	NULL — перед выполнением SQL-оператора TRUE — если оператор не обработал ни одной строки FALSE — если оператор обработал хотя бы одну строку
%ROWCOUNT	Общее количество обработанных строк
%BULK_ROWCOUNT	Ассоциативный массив с количествами строк, обработанных на каждом шаге цепочки исполнения однотипных операций (начиная с версии 8.1)

Синтаксис использования атрибута: SQL%*атрибут*. Качество программы выиграет, если к атрибутам неявного курсора обращение проставить сразу следом за командой SQL.

Пример использования в программе (использован параметр для указания значения в диалоге в SQL*Plus):

```
BEGIN
   UPDATE emp SET sal = sal WHERE deptno = &department_no;
   IF SQL%NOTFOUND THEN
      DBMS_OUTPUT.PUT_LINE ( 'В этом отделе нет сотрудников' );
   ELSE
      DBMS_OUTPUT.PUT_LINE ( SQL%ROWCOUNT || ' записей обновлено' );
   END IF;
END;
/
```

Неявный курсор, указанный по месту применения в цикле по курсору (FOR record **IN (SELECT … FROM …)** LOOP … END LOOP;) атрибутов не допускает.

Если неявный курсор не содержит функций обобщения (COUNT и прочих), он может быть блокирующим. Пример из SQL*Plus:

```
CONNECT scott/tiger
HOST echo VARIABLE sal NUMBER > wait5seconds.sql
BEGIN
 SELECT sal INTO :sal FROM emp WHERE ename = 'SMITH' FOR UPDATE WAIT 5;
END;
.
SAVE wait5seconds APPEND
@wait5seconds
HOST echo EXIT >> wait5seconds.sql
HOST sqlplus scott/tiger @wait5seconds
```

Выход из второго сеанса произойдет по прошествии пяти секунд.

8. Встроенный динамический SQL

В Oracle имеется два средства для динамического порождения программой предложений SQL:

- «встроенный динамический SQL» (native dynamic SQL, начиная с версии Oracle 8.1) и
- встроенный пакет DBMS_SQL (имелся и раньше).

Пакет DBMS_SQL обладает в целом несколько большей функциональностью, но существенно сложнее в употреблении. Фактически он предоставляет возможность в программе на PL/SQL пользоваться средствами OCI (Oracle call interface), иначе доступными в программах на C, хотя и с небольшими ограничениями.

8.1. Операторы встроенного динамического SQL

Встроенный динамический SQL воплощен в двух операторах:

- EXECUTE IMMEDIATE
- OPEN … FOR

8.1.1. EXECUTE IMMEDIATE

В отличие от статического SQL, EXECUTE IMMEDIATE позволяет выполнять из программы не только DML-, но и DCL- и DDL-операторы. Общий вид:

EXECUTE IMMEDIATE *строка_SQL*
[INTO {*список_переменных* | *запись* | *объектная_переменная*}]
[USING
[IN | OUT | IN OUT] *связная_переменная* [, [IN | OUT | IN OUT] *связная_переменная*]...
]
[{RETURNING | RETURN} INTO *список_связных_переменных*];

строка_SQL может быть любым SQL-предложением за исключением SELECT с множественным результатом. Примеры:

EXECUTE IMMEDIATE '**TRUNCATE** TABLE emp';
EXECUTE IMMEDIATE 'GRANT SELECT ON ' || **table_name** || ' TO ' || **grantee_list**;

Сравните последнее предложение с:

EXECUTE IMMEDIATE 'GRANT SELECT ON **:table** TO **:grantees**'
USING IN **table_name**, IN **grantee_list**;

Результат тот же, но для СУБД формулировка часто более предпочтительна, например по причине большей безопасности кода.

Заметьте, что переменные привязки в тексте для посылки именуются произвольно и независимо от имен сопоставляемых им переменных программы.

Привязка значений к переменным программы с использованием USING, RETURNING и INTO оформлена избыточно. Например, одинаковый результат получим от двух разных формулировок, с USING OUT и RETURNING INTO (текст для SQL*Plus):

```
VARIABLE c NUMBER
EXECUTE EXECUTE IMMEDIATE -
        'UPDATE emp SET sal = sal RETURNING COUNT ( * ) INTO :d' -
        USING OUT :c
```

```
PRINT c
EXECUTE EXECUTE IMMEDIATE -
        'UPDATE emp SET sal = sal RETURNING COUNT ( * ) INTO :d' -
        RETURNING INTO :c
PRINT c
```

Однако в целом конструкция USING более универсальна, так как допускает привязку к переменным программы как для передачи данных в запрос (USING IN), так и обратно (USING OUT).

Для SELECT следует пользоваться конструкцией INTO (текст для SQL*Plus):

```
EXECUTE EXECUTE IMMEDIATE 'SELECT COUNT ( * ) FROM emp' INTO :c
PRINT c
```

Пример для отправки на исполнение неименованного блока PL/SQL (текст для SQL*Plus):

```
EXECUTE EXECUTE IMMEDIATE 'BEGIN DBMS_OUTPUT.PUT_LINE ( 123 ); END;'
```

8.1.2. OPEN … FOR

Открытие курсора и возврат в программу ссылки на него. Динамическим здесь может быть только оператор запроса SELECT, но в отличие от случая с EXECUTE IMMEDIATE — с множественным результатом:

OPEN *ссылка_на_курсор* **FOR** *предложение_SELECT* [USING *список_связных_переменных*];

(А далее использоваться предложения

FETCH *ссылка_на_курсор* INTO {*список_переменных* | *запись* | *объектная_переменная*};

CLOSE *ссылка_на_курсор*;)

Пример:

```
DECLARE
  cv     SYS_REFCURSOR;
  vename VARCHAR2 ( 14 );
  vsal   NUMBER := 1000;
BEGIN
  OPEN cv FOR
          'SELECT ename, sal FROM emp WHERE sal > ' || TO_CHAR ( vsal );
  LOOP
    FETCH cv INTO vename, vsal;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE ( RPAD ( vename, 15 ) || vsal );
      -- дальше обрабатываем строку, как требуется ...
      --
   END LOOP;
   CLOSE cv;
END;
/
```

Если после слова FOR следует текстовое выражение, оператор OPEN … FOR собственно и следует считать динамическим. Однако этот оператор может существовать и в статическом варианте. В последнем примере он мог бы выглядеть так:

```
OPEN cv FOR
        SELECT ename, sal FROM emp WHERE sal > TO_CHAR ( vsal );
```

Обратите внимание на отсутствие здесь внешних кавычек. В этом случае текст запроса транслируется во время трансляции программы, а не выполнения оператора PL/SQL, просто это будет запрос с параметром.

8.2. Сравнительный пример двух способов работы с динамическим SQL

Для сравнения степени сложности работы с динамическим SQL средствами пакета DBMS_SQL с одной стороны, и встроенного динамического SQL с другой, можно привести два простых примера определения одной и той же по результату работы функции.

Пример с использованием пакета DBMS_SQL:

```
CREATE OR REPLACE PROCEDURE exec0 ( string IN VARCHAR2 )
AS
  cursor_name INTEGER;
  ret         INTEGER;
BEGIN
cursor_name := DBMS_SQL.OPEN_CURSOR;

-- DDL statements are run by the parse call,
-- which performs the implied commit.

DBMS_SQL.PARSE ( cursor_name, string, DBMS_SQL.NATIVE );
ret := DBMS_SQL.EXECUTE ( cursor_name );
DBMS_SQL.CLOSE_CURSOR ( cursor_name );
END;
/
```

Пример с использованием встроенного динамического SQL:

```
CREATE OR REPLACE PROCEDURE exec1 ( string IN VARCHAR2 )
AS
BEGIN
 EXECUTE IMMEDIATE string;
END;
/
```

Все же, у пакета DBMS_SQL есть свои преимущества перед встроенным динамическим SQL. Например, этот пакет:

- в версиях 10- обеспечивает работу с текстами запросов размером более 32 Кб (в версиях 11+ встроенный динамический SQL принимает выражения из строки типа CLOB)
- транслирует запросы в клиентской программе и может многократно повторять запрос без перетрансляции, что эффективнее полных повторений всего цикла обработки запросов, посылаемых на сервер встроенным динамическим SQL

8.3. Работа с динамическим SELECT посредством ссылки на курсор

8.3.1. Общие сведения

Ссылка на курсор — это именованная структура данных, ссылающаяся на курсор, ссылающийся, в свою очередь, на массив выбираемых курсором данных. Ее можно использовать, начиная с версии Oracle 7.2 для извлечения множественного результата в ответ на динамически подготовленное предложение SELECT, для маскировки незначительных изменений в запросе SQL, или для перераспределения структур обработки запроса в памяти СУБД в целях повышения производительности. Синтаксис объявления типа следующий:

TYPE *имя_типа_ссылки_на_курсор* IS REF CURSOR [RETURN *тип_записи*];

Если конструкция RETURN присутствует, ссылка на курсор называется строгой; если нет — нестрогой. Нестрогая может ссылаться на любой запрос, а строгая — только тот, что возвращает результат указанного типа.

Пример описания обоих типов ссылки на курсор:

```
DECLARE
TYPE    any_curstype      IS REF CURSOR;
```

```
TYPE    company_curstype IS REF CURSOR RETURN dept%ROWTYPE;
generic_cursvar any_curstype;
company_cursvar company_curstype;

BEGIN NULL; END;
/
```

Открытие курсора с помощью переменной-ссылки на курсор:

OPEN *ссылка_на_курсор* FOR *текст_предложения_SELECT*;

Команды FETCH и CLOSE используются как обычно.

Для удобства программирования поддерживается встроенный тип SYS_REFCURSOR для нестрогой ссылки на курсор:

```
DECLARE
   generic_cursor SYS_REFCURSOR;
   someemployee   emp.ename%TYPE;
BEGIN
   OPEN  generic_cursor FOR 'SELECT ename FROM emp';
   FETCH generic_cursor INTO someemployee;
   DBMS_OUTPUT.PUT_LINE ( 'Some employee: ' || someemployee );
   CLOSE generic_cursor;
END;
/
```

Предложения TYPE здесь не понадобилось. Более полные примеры использования имеются выше («Встроенный динамический SQL») и ниже по тексту.

Ограничения на использование ссылки на курсор:

- ссылки на курсор не могут объявляться как переменные пакета PL/SQL, и их нельзя передавать через переменные пакета
- ссылку на курсор нельзя связывать с блокирующим предложением SELECT … FOR UPDATE (в версии 10 уже можно)
- ссылкам на курсор нельзя присваивать значение NULL (в версии 10 уже можно) и их нельзя сравнивать друг с другом (но их можно присваивать друг другу)
- ссылки на курсор нельзя хранить в столбцах таблиц и в элементах коллекции
- ссылки на курсор нельзя передавать от сервера к серверу с помощью RPC
- ссылки на курсор нельзя использовать с пакетом DBMS_SQL

8.3.2. Пример употребления для структуризации программы

Использование курсора PL/SQL с курсорным выражением CURSOR (…) вкупе с обработкой результатов курсорного выражения при помощи ссылки на курсор позволяет отчетливее указать процедурный характер обработки вложенных запросов в программе. Рассмотрим пример (порождение данных в формате JSON; особого внимания отступам не уделяется):

```
SET SERVEROUTPUT ON

DECLARE
outbuffer VARCHAR2 ( 32767 );

CURSOR main_cursor
IS
   SELECT
     deptno
   , dname
   , CURSOR ( SELECT ename FROM emp WHERE emp.deptno = dept.deptno )
```

```
   FROM dept
;
deptnumber   dept.deptno%TYPE;
deptname     dept.dname%TYPE;
empname      emp.ename%TYPE;
inner_cursor SYS_REFCURSOR;

PROCEDURE put   ( n NUMBER ) AS
                BEGIN outbuffer := outbuffer || TO_CHAR ( n ); END;
PROCEDURE put   ( v VARCHAR2 ) AS
                BEGIN outbuffer := outbuffer || v; END;
PROCEDURE putln ( n NUMBER ) AS
                BEGIN put ( n ); put ( CHR ( 10 ) ); END;
PROCEDURE putln ( v VARCHAR2 ) AS
                BEGIN put ( v ); put ( CHR ( 10 ) ); END;

BEGIN
  OPEN main_cursor;
  putln ( '{ "departments": [' );

  LOOP
     FETCH main_cursor INTO deptnumber, deptname, inner_cursor;
     EXIT WHEN main_cursor%NOTFOUND;
     putln ( CASE WHEN main_cursor%ROWCOUNT > 1 THEN ', ' END
           || '"department": {'
           );
     put   ( '"number": ' ); put ( deptnumber ); putln ( ',' );
     putln ( '"name": "' || deptname || '",' );
     put   ( '"employees": [' );

     LOOP
        FETCH inner_cursor INTO empname;
        EXIT WHEN inner_cursor%NOTFOUND;
        put ( CASE WHEN inner_cursor%ROWCOUNT > 1 THEN ', ' END
            || '"' || empname || '"'
            );
     END LOOP inner_cursor;

     CLOSE inner_cursor;
     putln ( ']}' );
  END LOOP main_cursor;

  CLOSE main_cursor;
  putln ( ']}' );

  DBMS_OUTPUT.PUT_LINE ( outbuffer );
END;
/
```

Здесь уже в разделе описания, по внешнему виду курсора, усматривается способ его предполагаемой процедурной обработки в разделе выполнения. Если бы мы обошлись двумя отдельными курсорами (один для отделов и второй, параметризованный номером отдела, для сотрудников), но тогда будущая связь их друг с другом в тексте не казалась бы очевидной.

Обратите внимание, что открытие курсора по курсорному выражению CURSOR (…) происходит автоматически; внутренний курсор приходит от курсорного выражения уже открытым.

8.4. Сравнение статических и динамических курсоров в программировании

Курсорный механизм общения программы PL/SQL с БД имеет неоспоримые преимущества перед техникой использования статических запросов к данным. Тем не менее свои достоинства имеют и последние. В частности:

- статические запросы («неявные курсоры») надежнее в программе, так как позволяют выявить ошибки синтаксиса и обращения к объектам доступа еще на этапе трансляции программы, а не ее выполнения;
- статические запросы быстрее обрабатываются в программе в силу того, что часть их обработки опять-таки происходит еще на этапе трансляции программы;
- статические запросы не подвержены инъекциям кода, и тем самым безопаснее.

Программисту можно посоветовать прибегать к динамическому использованию SQL только в случаях, когда статическими средствами задача не решается.

9. Обработка исключительных ситуаций

При возникновении во время выполнения основного кода исключительной ситуации («исключения») управление передается специальному разделу обработки блока PL/SQL (см. выше).

9.1. Объявление исключительных ситуаций

Исключительные ситуации («исключения») в PL/SQL имеют номер и текст сообщения. Некоторые могут иметь вдобавок еще и имя, если об этом специально позаботиться. Наличие а программе имени у конкретной исключительной ситуации позволяет сослаться на нее в разделе обработки последствий возникновения таких ситуаций. Синтаксис объявления имени:

```
DECLARE
        имя_исключительной_ситуации EXCEPTION;
        …
```

Группа исключительных ситуаций поименована во встроенном пакете STANDARD, так что этими («предопределенными») именами можно пользоваться в любой программе (список ниже соответствует версиям Oracle 8.1+):

Номер ошибки	***Имя исключительной ситуации***
ORA-06511	CURSOR_ALREADY_OPEN
ORA-00001	DUP_VAL_ON_INDEX
ORA-00051	TIMEOUT_ON_RESOURCE
ORA-01001	INVALID_CURSOR
ORA-01012	NOT_LOGGED_ON
ORA-01017	LOGIN_DENIED
ORA-01403(*)	NO_DATA_FOUND
ORA-01476	ZERO_DIVIDE
ORA-01722	INVALID_NUMBER
ORA-01422	TOO_MANY_ROWS
ORA-06500	STORAGE_ERROR
ORA-06501	PROGRAM_ERROR
ORA-06502	VALUE_ERROR
ORA-06530	ACCESS_INTO_NULL
ORA-06531	COLLECTION_IS_NULL
ORA-06532	SUBSCRIPT_OUTSIDE_LIMIT
ORA-06533	SUBSCRIPT_BEYOND_COUNT
ORA-06504(10-)	ROWTYPE_MISMATCH
ORA-01410(10-)	SYS_INVALID_ROWID
ORA-30625(10-)	SELF_IS_NULL
ORA-06592(10-)	CASE_NOT_FOUND
ORA-01725(10-)	USERENV_COMMITSCN_ERROR
ORA-06548(10-)	NO_DATA_NEEDED

(*) Эта ошибка имеет сообщение no data found. Такое же сообщение выдает ошибка СУБД ORA-00100, однако смысл ее несколько иной (см. документацию).
(10-) Начиная с версии 10.

В пределах блока PL/SQL все имена исключительных ситуаций должны быть разными, но во вложенном блоке имя можно переопределить (конфликт имен разрешается в пользу «более близкого» определения).

С объявленной исключительной ситуацией почти всегда требуется связать номер ошибки Oracle:

```
DECLARE
        имя_исключительной_ситуации EXCEPTION;
        PRAGMA EXCEPTION_INIT (имя_исключительной_ситуации, номер_ошибки);
```

…

Формально этого можно не делать, но тогда исключительную ситуацию мы обязаны обработать в этом же блоке («локальная» исключительная ситуация, см. ниже), а иначе Oracle передаст ее в охватывающий блок под номером -06510 и с сообщением 'PL/SQL: unhandled user-defined exception'. Заданные с помощью PRAGMA EXCEPTION_INIT номера могут браться (а) из множества номеров сообщений СУБД (префикс ORA- в книге Error Messages документации по Oracle), и (б) из специально выделенного для программистов на PL/SQL диапазона от -20000 до -20999. Фирма Oracle, тем не менее, занимает несколько начальных номеров из «пользовательского» диапазона -20000 .. -20999 в ряде своих встроенных пакетов, как-то: DBMS_STATS, DBMS_DDL и других. Обращение к подпрограммам из этих пакетов в готовом приложении маловероятно, но при отладке возможно, и воизбежание недоразумений номера -20000 .. -20005 программисту лучше не занимать.

Исключительные ситуации с номерами из пользовательского диапазона должны активироваться в программе явно.

9.2. Примеры обработки

Именованные исключительные ситуации обрабатываются в разделе EXCEPTION блока PL/SQL с помощью конструкции WHEN. Пример:

```
…
EXCEPTION
        WHEN имя_исключительной_ситуации THEN программный код
…
END;
```

Пример обработки нескольких ситуаций:

```
…
EXCEPTION
        WHEN имя_ситуации1                          THEN  программный код1
        WHEN имя_ситуации2                          THEN  программный код2
        WHEN имя_ситуации3 OR имя_ситуации4 THEN  программный код3-4
         …
        WHEN OTHERS THEN
                программный код
END;
```

Конструкция WHEN OTHERS может возникать только в завершение списка и позволяет обрабатывать «все прочие» ситуации (не учтенные напрямую в разделе EXCEPTION выше).

9.3. Порождение исключительных ситуаций

Осуществляется «естественным путем» (СУБД, в процессе выполнения программы) или искуственно (программистом), одним из трех способов:

- естественным путем, со стороны СУБД
- искусственно, предложением RAISE (системные исключительные ситуации) в программе
- искусственно, обращением к системной процедуре RAISE_APPLICATION_ERROR (исключительные ситуации пользователя) в программе

Синтаксис предложения RAISE:

RAISE [*имя_исключительной_ситуации*];

(*имя_исключительной_ситуации* можно не указывать при выполнении RAISE из раздела обработки исключительных ситуаций блока PL/SQL).

Системная процедура RAISE_APPLICATION_ERROR используется для активации сигнала об ошибке приложения и имеет следующий заголовок:

```
RAISE_APPLICATION_ERROR (
  номер                          IN BINARY_INTEGER
, сообщение                      IN VARCHAR2
, сохранять_ли_стек_ошибок       IN BOOLEAN DEFAULT FALSE
);
```

Примеры:

```
DECLARE i NUMBER; BEGIN i := 1 / 0; END;
/

BEGIN RAISE ZERO_DIVIDE; END;
/

BEGIN RAISE NO_DATA_FOUND; END;
/

DECLARE x EXCEPTION; PRAGMA EXCEPTION_INIT ( x, -1498 );
BEGIN RAISE x; END;
/

BEGIN
 RAISE_APPLICATION_ERROR ( -20100, 'This is my application error' );
END;
/
```

Упражнение. Попробовать в двух последних примерах поменять местами ошибки -1498 и -20100.

```
BEGIN RAISE ZERO_DIVIDE; EXCEPTION WHEN ZERO_DIVIDE THEN NULL; END;
/
```

9.4. Зона действия и распространение

Обработчик исключительных ситуаций обрабатывает только ситуации, возникшие в исполняемом разделе блока PL/SQL.

Необработанные в разделе EXCEPTION (или активированные там повторно) исключительные ситуации передаются в охватывающий блок, снова в раздел EXCEPTION. Если ни в одном из охватывающих блоков ситуация не обрабатывается, то:

(1) программа прекращает работу, и
(2) автоматически выдается ROLLBACK.

Обработанная же исключительная ситуация далее по цепочке «наверх» не передается. По завершению обработки управление передается в основной раздел выполнения охватывающего блока, на оператор, следующий за обращением к блоку, где возникло исключение.

В то же время в коде обработки в разделе EXCEPTION позволительно явно породить исключительную ситуацию повторно оператором RAISE или процедурой RAISE_APPLICATION_ERROR. Такой пример приводится ниже, где в разделе EXCEPTION приведено обращение к RAISE без уточнений. Смысл состоит в

том, чтобы делегировать обработку той же ошибки –2292 охватывающим блокам, однако сопроводив это событие дополнительными действиями (здесь просто выдача на экран):

```
DECLARE
есть_сотрудники EXCEPTION;
PRAGMA EXCEPTION_INIT ( есть_сотрудники, -2292 );

PROCEDURE удалить ( отдел IN NUMBER ) IS
BEGIN
  DELETE FROM dept WHERE deptno = отдел;
EXCEPTION WHEN есть_сотрудники THEN
  DBMS_OUTPUT.PUT_LINE ( 'В отделе ' || отдел || ' есть сотрудники.' );
  RAISE;
END;

BEGIN
  COMMIT;
  удалить ( 40 );                                                    -- OK
  BEGIN удалить ( 30 ); EXCEPTION WHEN есть_сотрудники THEN NULL; END;
                                                                     -- OK
  удалить ( 20 );  -- аварийный останов программы и автоматический ROLLBACK
  удалить ( 10 );  -- оператор уже выполнен не будет
END;
/
```

Упражнение. Выполнить пример выше. Проверить список отделов. Убрать из текста удаление отделов 20 и 10 и выполнить пример снова. Посмотреть список отделов. Выполнить ROLLBACK. Посмотреть список отделов и объяснить результат.

9.5. Локальные исключительные ситуации

Если исключительная ситуация искусственно порождается и обрабатывается в пределах блока, где она определена, связывать ее с каким-либо номером ошибки нужды нет. Пример (для SQL*Plus):

```
DECLARE
 a EXCEPTION;
 b EXCEPTION;
BEGIN
 IF &true_or_false THEN RAISE a; ELSE RAISE b; END IF;
 EXCEPTION
  WHEN a THEN DBMS_OUTPUT.PUT_LINE ( 'a' );
  WHEN b THEN DBMS_OUTPUT.PUT_LINE ( 'b' );
END;
.
```

Проверка:

```
SQL> /
Enter value for true_or_false: true
a

PL/SQL procedure successfully completed.

SQL> /
Enter value for true_or_false: false
b

PL/SQL procedure successfully completed.
```

Это позволяет использовать исключительные ситуации для удобной организации кода в блоке, а не для передачи сигнала наружу, например:

```
DECLARE
   CURSOR nulls IS   SELECT comm FROM emp;
   comm              emp.comm%TYPE;
   nullscount        NATURAL := 0;
   two_or_more_nulls EXCEPTION;
   no_rows_at_all    EXCEPTION;

BEGIN
OPEN nulls;

FETCH nulls INTO comm;
IF nulls%NOTFOUND THEN RAISE no_rows_at_all; END IF;

LOOP
   FETCH nulls INTO comm;
   IF nulls%NOTFOUND THEN EXIT; END IF;
   IF comm IS NULL THEN
      nullscount := nullscount + 1;
      IF nullscount = 2 THEN RAISE two_or_more_nulls; END IF;
   END IF;
END LOOP;

CLOSE nulls;
DBMS_OUTPUT.PUT_LINE ( nullscount );

EXCEPTION
   WHEN no_rows_at_all THEN
     CLOSE nulls;
     DBMS_OUTPUT.PUT_LINE ( 'Столбец пуст' );

   WHEN two_or_more_nulls THEN
     CLOSE nulls;
     DBMS_OUTPUT.PUT_LINE ( 'Значение отсутствует два раза или более' );
END;
/
```

Разумеется, описанные выше действия можно заменить на равносильные другие, не прибегая к локальным исключительным ситуациям, и речь стоит только о разумном обустройстве текста программы.

9.6. Прочие техники обработки особых ситуаций в программе

Обработка особых случаев техникой «исключительных ситуаций» представляет собой общий и действенный механизм программирования. Однако при работе с данными БД PL/SQL дает и другие средства, более частного характера, но в отдельных случаях вполне приемлемые. Два из них упоминаются ниже.

9.6.1. Использование функций SQLCODE и SQLERRM

Встроенные функции SQLCODE и SQLERRM дают код завершения и разъясняющее сообщение СУБД для последнего выполнявшегося в программе оператора SQL, возможно вызвавшего исключительную ситуацию. Далее идет пример, поясняющий смысл, характеристики и употребление этих функций. Пример состоит из создания тренировочной таблицы ERR_TEST и программного кода по работе с ней:

```
CREATE TABLE err_test (
  name     VARCHAR2 ( 5 )
, quantity NUMBER
,
  CONSTRAINT big_first_letter
    CHECK ( SUBSTR ( name, 1, 1 ) BETWEEN 'A' AND 'Z' )
, CONSTRAINT no_small_numbers
    CHECK ( quantity > 1024 )
```

```
);

BEGIN
   INSERT INTO err_test VALUES ( 'Athen', 2000 );
   DBMS_OUTPUT.PUT_LINE ( SQLERRM );

   BEGIN INSERT INTO err_test VALUES ( 'Athen', 200 );
   EXCEPTION WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE ( SQLERRM );
   END;

   BEGIN INSERT INTO err_test VALUES ( 'athen', 2000 );
   EXCEPTION WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE ( SQLERRM );
   END;

   BEGIN INSERT INTO err_test VALUES ( 'Athena', 2000 );
   EXCEPTION WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE ( SQLERRM ); RAISE;
   END;

EXCEPTION WHEN OTHERS THEN
   CASE
   WHEN SQLCODE = -2290 AND SQLERRM LIKE '%NO_SMALL_NUMBERS%'
     THEN DBMS_OUTPUT.PUT_LINE ( 'Quantity is too small' );

   WHEN SQLCODE = -2290 AND SQLERRM LIKE '%BIG_FIRST_LETTER%'
     THEN DBMS_OUTPUT.PUT_LINE ( 'First letter should be capital ' );

   ELSE DBMS_OUTPUT.PUT_LINE (
     'Exception not hooked, ' || ' SQLCODE = ' || TO_CHAR ( SQLCODE ) );
   END CASE;
END;
/
```

Упражнение. Прогнать пример. Добавить оператор RAISE в раздел обработки исключительной ситуации одной из других операций вставки строки. Повторить выполнение. Снять скобки BEGIN ... END с одной из операций вставки. Повторить выполнение. Переписать пример иначе, воспользовавшись именами ситуаций и прибегнув к PRAGMA EXCEPTION_INIT. Повторить выполнение.

Функция SQLERRM может иметь аргумент в виде номера ошибки; в этом случае ее можно использовать для отладки и для работы с недокументированными возможностями Oracle. Пример, значимый для версии 8.1:

```
EXECUTE DBMS_OUTPUT.PUT_LINE ( SQLERRM ( -10406 ) )
```

Выдать стек сообщений исключительных ситуаций можно процедурой FORMAT_ERROR_STACK системного пакета DBMS_UTILITY.

9.6.2. Техника ухода от исключительных ситуаций

Обработка машиной PL/SQL исключительных ситуаций сравнительно затратна, как раз в силу «исключительности» последних. Когда особенно важна скорость, следует попытаться поискать иные пути программирования, и иногда это удается.

Статическое употребление в программе оператора SELECT, как известно, в случае нуль/многострочного результата породит исключительную ситуацию, обрабатывать которую придется в соответствующем такому событию разделе блока. В то же время использование курсора позволяет не доводить дело до исключительной ситуации. Схема программирования поясняется следующим кодом, назначение которого состоит в проверке количества строк в ответе на запрос, если требуется узнать, будет ли это одна строка, ни одной, или же более одной:

```
DECLARE
CURSOR emp_cur IS
         SELECT empno, sal, job, deptno
```

```
         FROM   emp
         WHERE  job = 'CLERK' OR deptno = 10
         ;
emp_rec  emp_cur%ROWTYPE;
out      VARCHAR2 ( 50 ) := 'Сотрудников нет';

BEGIN
OPEN emp_cur;
FETCH emp_cur INTO emp_rec;

IF emp_cur%FOUND THEN
   BEGIN
   out := 'Сотрудник ' || emp_rec.empno;
   FETCH emp_cur INTO emp_rec;
   IF emp_cur%FOUND THEN out := 'Сотрудник не один'; END IF;
   END;
END IF;

CLOSE emp_cur;
DBMS_OUTPUT.PUT_LINE ( out );
END;
/
```

Такое решение особенно удачно, когда запрос требуется задавать многократно.

Упражнение. Переписать последний пример с привлечением локальных исключительных ситуаций. Сравнить старый и новый тексты (а) с точки зрения удобства понимания написанного; (б) с точки зрения эффективности выполнения. Предложить третий вариант оформления, с использованием меток. Повторить сравнение.

«Уходить» от исключений можно и вне связи с командами SQL, например, в связи с ожидаемыми нарушениями типов:

```
DECLARE a BINARY_INTEGER := -1; b POSITIVE; BEGIN b := a; END;
                           -- ORA-06502: PL/SQL:  ошибка числа или значения
/
```

Один из следующих двух фрагментов запрограммирован с исключением, а другой без:

```
DECLARE a NUMBER NOT NULL := 1;
        b NUMBER;
BEGIN
 a := b;
EXCEPTION
 WHEN OTHERS THEN a := -1; DBMS_OUTPUT.PUT_LINE ( a );
END;
/

DECLARE a NUMBER NOT NULL := 1;
        b NUMBER;
BEGIN
 a := CASE WHEN b IS NULL THEN -1 END;
 DBMS_OUTPUT.PUT_LINE ( a );
END;
/
```

10. Хранимые процедуры и функции

Именованными единицами программного кода в PL/SQL, допускающими хранение в базе, могут быть:

- процедуры
- функции
- триггерные процедуры («триггеры»)
- пакеты
- типы объектов

Первые две категории под общим названием «подпрограммы» рассматриваются в этом разделе.

10.1. Общий синтаксис

Общий синтаксис создания процедуры в PL/SQL:

```
CREATE [OR REPLACE] PROCEDURE имя_процедуры
        [(список_параметров)]
        [AUTHID {CURRENT_USER | DEFINER}]
        [ACCESSIBLE BY (…)]
{IS | AS}
        [раздел объявлений]
BEGIN
        раздел кода
[EXCEPTION
        раздел обработки исключительных ситуаций]
END [имя_процедуры];
```

Пример вызова процедуры в блоке PL/SQL:

```
apply_discounts ( new_company_id, 0.15 );      -- скидка 15%
```

Пример самостоятельного вызова процедуры в SQL*Plus:

```
SQL> EXECUTE apply_discounts ( new_company_id, 0.15 )
```

Общий синтаксис функции в PL/SQL:

```
CREATE [OR REPLACE] FUNCTION имя_функции
        [(список_параметров)]
        RETURN тип_результата
        [AUTHID {CURRENT_USER | DEFINER}]
        [RESULT_CACHE [RELIES_ON ( имя_таблицы )]][11.1]
        [DETERMINISTIC]
        [PARALLEL_ENABLE]
        [PIPELINED …]
        [AGGREGATE …]
        [ACCESSIBLE BY (…)]
{IS | AS}
        [раздел объявлений]
BEGIN
        раздел кода, включающий предложение RETURN
[EXCEPTION
        раздел обработки исключительных ситуаций]
END [имя_функции];
```

[11.1] в версии 11.

(В заголовке функции могут указываться кроме указанных другие, более специальные конструкции, о которых см. в документации).

Функции на PL/SQL могут использоваться как *элемент выражений* в SQL и PL/SQL. Примеры:

- присваивание:
  ```
  sales99 := tot_sales(1999, 'C');
  ```
- выставление значения по умолчанию:
  ```
  DECLARE
      Sales99 NUMBER DEFAULT tot_sales(1999, 'C');
  BEGIN
  …
  ```
- в выражении:
  ```
  IF tot_sales(1999, 'C') > 10000
      THEN
      …
  ```
- в предложении SQL:
  ```
  SELECT first_name, surname FROM sellers
  WHERE tot_sales(1999, 'C') > 10000;
  ```
- в качестве аргумента в списке параметров:
  ```
  apply_discount(company_id, max_discount(SYSDATE));
  ```

Указания компилятору в предложениях создания подпрограмм и функций см. ниже.

Удаление подпрограмм выполняется командой DROP PROCEDURE/FUNCTION *имя*.

10.2. Параметры

список_параметров выше — перечисление параметров через запятую. Синтаксис объявления параметра в списке:

имя_параметра [*режим_использования*] [NOCOPY] *тип_ параметра* [{:= | DEFAULT} *выражение*]

10.2.1. Тип параметра

Типом параметра для процедур и функций могут быть любые основные типы данных в PL/SQL. Оговорка касается точности: в определении формального параметра и типа результата в функции ее указывать запрещено.

Следующие определения формальных параметров *недопустимы*:

```
name        VARCHAR2 ( 20 )
id          NUMBER ( 5, 2 )
moment      TIMESTAMP ( 4 )
timeint     INTERVAL YEAR ( 2 ) TO MONTH
```

Следующие определения формальных параметров *возможны*:

```
id          PLS_INTEGER
en          emp.ename%TYPE
emp         emp%ROWTYPE
name        VARCHAR2
id          NUMBER
tm          TIMESTAMP
tint        INTERVAL YEAR TO MONTH
```

timeint YMINTERVAL_UNCONSTRAINED

Фактическая длина для обработки в подпрограмме, если это необходимо, определиться при ее вызове.

Обратите внимание на последствия. Например, несмотря на возможность в PL/SQL определять целочисленные переменные, а в БД Oracle целочисленные столбцы, целочисленных функций Oracle не допускает. Отсюда, в частности, вытекает необходимость наличия в Oracle неявного преобразования типов, порицаемого специалистами.

10.2.2. Режим использования параметра

Режим использования (mode) параметра определяет, какую операцию реально можно производить с параметром в тексте подпрограммы:

Режим	***Операция***	***Передача по умолчанию***	***Пояснение***
IN	только чтение	по ссылке	В тексте подпрограммы к значению параметра можно только обращаться, но не изменять его.
OUT	только запись	по значению	До Oracle 8 в тексте подпрограммы значение параметра разрешалось только изменять; после — как изменять, так и использовать для присвоения переменным.
IN OUT	и чтение, и запись	по значению	Значение параметра разрешено как изменять, так и использовать для присвоения переменным.

По умолчанию для параметра используется режим IN.

При возникновении в подпрограмме исключительной ситуации сделанные для параметров OUT и IN OUT присвоения отменяются.

Можно предложить (с версии 8.1) компилятору изменить передачу параметров IN OUT, используемую по умолчанию, указав слово NOCOPY. В этом случае параметр будет передан по ссылке, однако, ввиду того, что NOCOPY — лишь подсказка, то не всегда, а когда компилятор сочтет это возможным (например, он не станет этого делать, если при подстановке параметра используется неявное преобразование типов). Замена передачи параметра значением на передачу по ссылке позволяет сэкономить расходы памяти и время на передачу значения, что дает заметную выгоду, когда значение — структура большого размера, например коллекция. Однако если во время работы подпрограммы произойдет откат (ROLLBACK) или исключение (exception), то начатые «снаружи» изменения значения отменены не будут, и для основной программе окажутся испорченными.

10.2.3. Значения по умолчанию

Могут использоваться для входных параметров (IN):

```
CREATE OR REPLACE PROCEDURE hire_employee
( emp_id          IN      VARCHAR2
, hire_date       IN      DATE := SYSDATE
, company_id      IN      NUMBER := 1
)
IS
        …
```

Если все параметры имеют умолчательные значения, то при обращении к подпрограмме их можно опускать, и тогда даже не указать скобок вовсе. Пример трех равносильных обращений:

```
DECLARE PROCEDURE p ( n NUMBER := 1 ) IS BEGIN NULL; END;
BEGIN
```

```
   p ( 1 );
   p ( );
   p;
END;
/
```

Передача отсутствующего значения (NULL) параметру с умолчанием не равносильна отсутствию передачи значения:

```
VARIABLE one NUMBER
VARIABLE two NUMBER

DECLARE
FUNCTION f ( n NUMBER := 1 ) RETURN NUMBER IS BEGIN RETURN n; END;
BEGIN
   :one := f ( NULL );
   :two := f ( );
END;
/
```

Проверка:

```
SQL> PRINT one two

       ONE
----------

       TWO
----------
         1
```

10.2.4. Способы указать фактические значения параметрам

Подстановка фактических параметров на место формальных при обращении к подпрограмме может осуществляться в PL/SQL двояко: позиционно и именованным перечислением. Пример:

```
BEGIN
  -- позиционная подстановка параметров
  empid_to_name ( 10, surname, first_name );

  -- именованная передача параметров
  empid_to_name ( in_id => 10, out_first_name => first_name );

  -- в точности то же самое
  empid_to_name ( out_first_name => first_name,  in_id => 10 );
END;
```

Допускается и комбинированный вариант, например:

```
EXEC dbms_stats.gather_table_stats ( 'scott', 'emp', cascade => TRUE )
```

При этом описание последней процедуры сбора статистик для оптимизатора запросов относительно выбранной таблицы выглядит (документация версии 11) примерно так:

```
DBMS_STATS.GATHER_TABLE_STATS (
  ownname         VARCHAR2,
  tabname         VARCHAR2,
  partname        VARCHAR2 DEFAULT NULL,
  ...
  cascade         BOOLEAN  DEFAULT to_cascade_type(get_param('CASCADE')),
```

```
  ...
  force          BOOLEAN DEFAULT FALSE);
```

Если хранимая функция вызывается в SQL-выражении, то фактические параметры для нее могут сообщаться только позиционно.

10.2.5. Обращение к параметрам и к локальным переменным в теле подпрограммы

Обращения к параметрам подпрограммы и к локальным переменным допускают уточнение именем подпрограммы, что можно использовать для повышения ясности кода и устранения конфликтов имен. Это правило равным образом распространяется и на локальные подпрограммы:

```
DECLARE
FUNCTION prettylook ( ename IN emp.ename%TYPE )
RETURN emp.ename%TYPE
IS
 job emp.job%TYPE;
BEGIN
 SELECT job                          -- можно emp.job
 INTO   prettylook.job               -- можно просто job
 FROM   emp
 WHERE  ename = prettylook.ename     -- нужно именно prettylook.ename
 ;
 RETURN INITCAP ( prettylook.job );  -- можно просто  job
END;

BEGIN
 DBMS_OUTPUT.PUT_LINE ( prettylook ( 'SMITH' ) );
END;
/
```

10.3. Рекурсия, взаимные вызовы и повторения имен

Для самостоятельно транслируемых хранимых подпрограмм:

- рекурсивный вызов *разрешен*, глубина рекурсии не регламентирована и зацикливание при исполнении кода СУБД не контролирует;
- взаимный вызов *разрешен* и зацикливание при исполнении кода СУБД не контролирует;
- повторение имен *запрещено*.

10.4. Указания компилятору при создании подпрограмм

Следующие указания могут использоваться в предложениях CREATE PROCEDURE и CREATE FUNCTION:

OR REPLACE

Если подпрограмма уже существует, она будет перетранслирована с сохранением имеющихся свойств

AUTHID

Указывает на то, будет ли выполнение подпрограммы осуществляться с правами пользователя, подпрограмму создавшего (DEFINER), или пользователя, к программе обратившегося (CURRENT_USER). До версии Oracle 8.1 последним качеством обладали лишь подпрограммы из системных пакетов DBMS_SQL и DBMS_UTILITY. Пример использования указания см. ниже.

ACCESSIBLE BY (…)

Позволяет указать перечень хранимых процедур, функций, пакетов, типов, в теле которых *исключительно* позволительно обращаться к подпрограмме. В частности, к такой подпрограмме невозможно будет обратиться из неименованного блока PL/SQL. Используется для упорядочения использования подпрограммы в коде приложения. Действует с версии 12.

Следующие указания могут использоваться только в предложениях CREATE FUNCTION:

RESULT_CACHE [RELIES_ON (*имя_таблицы*)]

Указывает на накопление в области SGA СУБД значений-результатов функции, из-за чего последующие к ней обращения могут выливаться всего лишь во взятие готовой, ранее вычислявшейся величины (с версии 11.2 RELIES_ON сообщать не обязательно, так как зависимости определяются автоматически).

DETERMINISTIC

Сообщает компилятору, что значение *функции* определяется исключительно значениями ее параметров (то есть в теле отсутствуют обращения к глобальным переменным пакета или к данным из базы данных). В этом случае PL/SQL-машина может оптимизировать вычисление функции при повторяющихся вызовах, подменяя собственно вычисление обращениями к ранее вычисленным значениям, хранимым в пределах сеанса. Указание DETERMINISTIC имеет обязательный (а не оптимизационный) характер, когда функцию требуется использовать в выражениях для индексов с преобразованием ключа (function-based index) или же в выражениях в запросе SQL, участвующем в формулировке материализованного представления данных со свойствами REFRESH FAST или ENABLE QUERY REWRITE.

PARALLEL_ENABLE

Подсказка оптимизатору о том, что вычисление *функции*, если она вызвана в теле запроса SQL, может быть распараллелено в случае применения СУБД параллельной обработки этого запроса; то есть в ней нет обращений к переменным сеанса (например, глобальным переменным пакета).

PIPELINED

Может использоваться для описания *табличной функции*, то есть функции, принимающей на входе набор строк и выдающей в качестве результата тоже набор. Если такая функция транслирована с указанием PIPELINED, результат начинает выдаваться построчно по мере формирования в теле функции, а не после полного подсчета всего набора строк. При этом в теле функции строки возвращаются по мере готовности индивидуально оператором PIPE ROW *имя_записи*, а не RETURN *результат*, как обычно. Действует с версии 9. Примеры см. ниже.

AGGREGATE USING

Используется для написания собственных обобщающих (агрегатных) функций (вдобавок к стандартным MIN, MAX и прочим). Действует с версии 9.

10.5. Хранимые подпрограммы и привилегии доступа к объектам в БД

10.5.1. Две логики реализации привилегий доступа к объектам в БД

Задаются указанием AUTHID **DEFINER**, либо AUTHID **CURRENT_USER**. Содержательный пример использования логики «права доступа соответствуют правам обратившегося к подпрограмме»:

```
CREATE OR REPLACE FUNCTION countme ( tabula IN VARCHAR2 )
RETURN NUMBER
AUTHID CURRENT_USER
AS
  cnt NUMBER;
BEGIN
```

```
  EXECUTE IMMEDIATE 'SELECT COUNT ( * ) FROM ' || tabula INTO cnt;
  RETURN cnt;
END;
/
```

Упражнение. Создать функцию COUNTME, как описано выше, и проверить в работе при обращении из «своей» и «чужой» схемы. Заменить AUTHID CURRENT_USER на AUTHID DEFINER и повторить опыт.

Обратите внимание, что «права создателя» дают возможность разрешать пользователям работать с таблицами БД, не предоставляя непосредственных привилегий доступа к ним. Таким образом, они дают вторую, после механизма views, возможность регламентировать доступ к данным БД без предоставления прямого доступа к таблицам.

Следующий пример показывает, как администратор может подготовить функцию, возвращающую пользователю полное имя трассировочного файла процесса СУБД, связанного с сеансом его текущей работы:

```
CONNECT / AS SYSDBA

CREATE OR REPLACE FUNCTION getmy_tracefile RETURN VARCHAR2
AUTHID DEFINER
AS
  filename VARCHAR2 ( 100 );
BEGIN
  SELECT tracefile
  INTO   filename
  FROM   v$process p INNER JOIN v$session s
         ON    ( p.addr = s.paddr
            AND s.sid  = SYS_CONTEXT ( 'USERENV', 'SID' )
         );
  RETURN filename;
END;
/
GRANT EXECUTE ON getmy_tracefile TO scott
;

CONNECT scott/tiger

SELECT sys.getmy_tracefile FROM dual
;
```

10.5.2. Особенности передачи привилегий через роли

Если подпрограмма транслирована в режиме AUTHID DEFINER и именована (не является анонимным блоком), то роли, присвоенные ее создателю («DEFINER»), не имеют действия при ее запуске. Все необходимые привилегии на использование объектов БД и на действия должны быть в этом случае переданы командой GRANT напрямую (не через роли).

Это вызвано особенностями компиляции программ на PL/SQL. Привилегии проверяются при транслировании программы. Знание ролей для компилятора, в отличие от знания явно присвоенных привилегий, ненадежно, так как роли могут включаться и отключаться в пределах отдельного сеанса (SQL предложение SET ROLE).

Следующий пример демонстрирует такое поведение (и рассчитан на версии, начиная с 9; в версии 8 и предшествовавших команды GRANT ниже следует выдавать от имени владельцев объектов):

```
CONNECT / AS SYSDBA

CREATE USER a IDENTIFIED BY a;

CREATE USER b IDENTIFIED BY b
   DEFAULT TABLESPACE users
```

```
   QUOTA UNLIMITED ON users
;

CREATE TABLE b.t AS SELECT * FROM scott.emp;

CREATE ROLE r;

GRANT SELECT ON b.t TO r;

GRANT r TO a;

CREATE OR REPLACE FUNCTION a.f RETURN NUMBER
AUTHID DEFINER
AS
retcount NUMBER;
BEGIN
   EXECUTE IMMEDIATE 'SELECT count ( * ) FROM b.t' INTO retcount;
   RETURN retcount;
END;
/

GRANT CREATE SESSION TO a;
```

Проверка:

```
SQL> CONNECT a/a
Connected.
SQL> SELECT * FROM session_roles;

ROLE
------------------------------
R

SQL> SELECT f FROM DUAL;
SELECT f FROM DUAL
       *
ERROR at line 1:
ORA-00942: table or view does not exist
ORA-06512: at "A.F", line 1
```

Выдаем привилегию напрямую и проверяем снова:

```
SQL> CONNECT / AS SYSDBA
Connected.
SQL> GRANT SELECT ON b.t TO a;

Grant succeeded.

SQL> CONNECT a/a
Connected.
SQL> SELECT f FROM DUAL;

  COUNTEMP
----------
        14
```

Использование статического SQL вместо динамического более надежно, так как не допустит трансляции функции A.F при отсутствии прямой привилегии на таблицу A.T с самого начала. *Упражнение*: проверьте это, заменив в A.F динамически выдаваемую операцию SELECT на статическую.

11. Триггерные процедуры

Триггерные процедуры — это процедуры, начинающие свою работу автоматически при возникновении события того или иного класса (см. ниже). Жаргонное название — «триггеры».

11.1. Создание триггерной процедуры

Синтаксис создания триггерной процедуры:

```
CREATE [OR REPLACE] TRIGGER имя_триггерной процедуры
        {BEFORE | AFTER | INSTEAD OF} класс_события
        ON
        {NESTED TABLE столбец_вложенной_таблицы OF представление
         | основная_таблица_или_представление
         | DATABASE
         | [схема.] SCHEMA
        }
        [именующая_фраза]
        [FOR EACH ROW]
        [FOLLOWS[11-) | PRECEDES[11.2-) имя_триггерной процедуры [,имя_триггерной процедуры ...]]
        [FORWARD | REVERSE CROSSEDITION][11.2-)
        [ENABLE | DISABLE][11-)
        [WHEN (условие_срабатывания)]
{
[DECLARE
        [PRAGMA AUTONOMOUS_TRANSACTION;]
        раздел объявлений]
BEGIN
        раздел кода
[EXCEPTION
        раздел обработки исключительных ситуаций]
END [имя_триггерной процедуры]
|
CALL имя_процедуры [(список_параметров)]
};
```

[11-) начиная с версии 11.
[11.2-) начиная с версии 11.2.

Классы событий для триггерных процедур перечислены ниже:

Группа событий	***Категория***	***Описание***
INSERT	DML	Срабатывает при постановке строки в таблицу. Не срабатывает при прямой загрузке (direct load).
UPDATE	DML	Срабатывает при модификации строки в таблице. При наличии уточнения OF (UPDATE OF *список_столбцов*) может срабатывать только при модификации определенных столбцов
DELETE	DML	Срабатывает при удалении строк из таблицы. Не срабатывает при TRUNCATE TABLE
CREATE[8.1-)	DDL	Срабатывает при выполнении операции CREATE создания объектов, попадающих в перечень в ALL_OBJECTS. Может создаваться на уровне отдельной схемы или всей БД
ALTER[8.1-)	DDL	Срабатывает при выполнении операции ALTER

		изменения свойств объектов, попадающих в перечень в ALL_OBJECTS. Может создаваться на уровне отдельной схемы или всей БД
DROP[8.1-)	DDL	Срабатывает при выполнении операции DROP удаления объектов, попадающих в перечень в ALL_OBJECTS. Может создаваться на уровне отдельной схемы или всей БД
ANALYZE[9-) ASSOCIATE STATISTICS[9-) AUDIT[9-) COMMENT[9-) DISASSOCIATE STATISTICS[9-) GRANT[9-) NOAUDIT[9-) RENAME[9-) REVOKE[9-) TRUNCATE[9-)	DDL	Прочие группы событий DDL
DDL[9-)	DDL	Обобщенное название для всех DDL-событий
SERVERERROR[8.1-)	База данных	Срабатывает при фиксации системой серверной ошибки. Может быть только типа AFTER
LOGON[8.1-)	База данных	Срабатывает при возникновении нового сеанса работы пользователя (программы). Может быть только типа AFTER
LOGOFF[8.1-)	База данных	Срабатывает при окончании работы сеанса. Может быть только типа BEFORE. Не срабатывает при разрыве сеанса.
STARTUP[8.1-)	База данных	Срабатывает при открытии БД. Может быть только типа AFTER
SHUTDOWN[8.1-)	База данных	Срабатывает при закрытии БД. Может быть только типа BEFORE. Не срабатывает при SHUTDOWN ABORT.
SUSPEND[9-)	База данных	Срабатывает при приостановке транзакции вследствие нехватки доступной дисковой памяти во время добавлении данных (исчерпана дисковая память, достигнут максимум числа экстентов в сегменте, исчерпана квота на табличное пространство). Если причину устранить, операцию DML можно продолжить.
DB_ROLE_CHANGE[10-)	База данных	Срабатывает при переключении ролями основной и резервной баз данных при использовании техники горячего резерва (standby)

[8.1-) Существуют, начиная с версии 8.1.
[9-) Существуют, начиная с версии 9.
[10-) Существуют, начиная с версии 10.

Определения AFTER и BEFORE используются для указания времени срабатывания триггерной процедуры: непосредственно до события, или после.

Определения FORWARD или BEFORE CROSSEDITION используются в целях администрирования приложения либо БД, для указания действий, необходимых к автоматическому осуществлению при переходе к новой редакции приложения.

Определения ENABLE или DISABLE указывают на исходное состояние триггерной процедуры: активное или деактивированное.

Указание FOR EACH ROW используется для DML-триггерные процедур и означает, что такая процедура «строчная», то есть будет исполняться для каждой изменяемой операцией строки таблицы.

именующая_фраза (REFERENCING) задает собственное имя для указания в теле процедуры записей с новыми и старыми значениями в полях для строчных DML-триггерных процедур. Без этого для ссылки на старые значения полей используется обозначение :OLD, а для ссылки на новые — обозначение :NEW. На столбцы типа LONG и LONG RAW такое именование не распространяется. Присвоение значений возможно только постолбцово (не на всей записи сразу). Для INSERT старые значения, а для DELETE новые — всегда NULL.

WHEN указывает на дополнительное условие, необходимое для фактического срабатывания DML-триггерной процедуры.

Пример создания триггерной процедуры:

```
CREATE OR REPLACE TRIGGER set_hiredate
   BEFORE INSERT ON emp
   REFERENCING NEW AS inserted_employee
   FOR EACH ROW
BEGIN :inserted_employee.hiredate := SYSDATE; END set_hiredate;
/
```

Примечание. Работа этой процедуры *не* эквивалентна следующему изменению описания таблицы EMP:

```
ALTER TABLE emp MODIFY ( hiredate DATE DEFAULT SYSDATE );
```

Упражнение. Проверить это.

Такое поведение триггерной процедуры объясняется тем, что присвоение значение полю записи :NEW Oracle рассматривает как *окончательное* в БД.

Пример того, как можно переделать предыдущую процедуру «почти» для имитации DEFAULT SYSDATE в определении столбца HIREDATE:

```
CREATE OR REPLACE TRIGGER set_hiredate
   BEFORE INSERT ON emp
   REFERENCING NEW AS inserted_employee
   FOR EACH ROW
   WHEN ( inserted_employee.hiredate IS NULL )
BEGIN :inserted_employee.hiredate := SYSDATE; END set_hiredate;
/
```

(Однако в отличие от DEFAULT SYSDATE для HIREDATE эта триггерная процедура не позволит оставить это поле в только что добавленой строке без значения).

Фраза CALL может заменять тело триггерной процедуры в виде блока PL/SQL вызовом обычной процедуры на PL/SQL, Java или C, например:

```
CREATE TRIGGER scott.salary_check
  BEFORE INSERT OR UPDATE OF sal, job ON scott.emp
  FOR EACH ROW
  WHEN ( NEW.job <> 'PRESIDENT' )
CALL check_sal ( :NEW.job, :NEW.sal, :NEW.name );
```

11.2. Отключение триггерных процедур

Имеющуюся триггерную процедуру можно «отключить» от события (чтобы она не запускалась автоматически) и снова «подключить» (чтобы запускалась):

ALTER TRIGGER *имя_триггерной_процедуры* {ENABLE | DISABLE};

Обобщенный вариант для DML-триггерных процедур, связанных с таблицей:

ALTER TABLE *имя_таблицы* {ENABLE | DISABLE} ALL TRIGGERS;

Замечание. Обе эти команды — категории DDL, а значит изменение состояния триггерной процедуры для таблицы приведет к неявному выполнению COMMIT (!), так как иначе был бы возможен конфликт с ближайшей операцией ROLLBACK.

Начальное состояние триггерных процедур до версии 11 было «включено», а с версии 11 устанавливается по выбору в момент создания (это оказалось вынужденной мерой ввиду того, что с версии 11 триггерные процедуры стали допускать взаимные ссылки).

Список имеющихся в схеме триггерных процедур с их свойствами находится в таблице USER_TRIGGERS.

11.3. Особенности триггерных процедур для событий категории DML

Триггерные процедуры для изменяющих операторов DML могут быть «уровня строки» (строковыми) и «уровня оператора» (вызвавшего срабатывание; иногда еще говорят «уровня таблицы»). Также, они могут запускаться непосредственно до (BEFORE), иди после (AFTER) выполнения оператора. С действиями над одной и той же таблицей разрешено связать сразу несколько триггерных процедур, всего, таким образом, четырех разных категорий.

Строковые триггерные процедуры имеют право обращаться к старым и новым значениями полей добавляемой, изменяемой или удаляемой строки. В случае изменения строковые триггерные процедуры типа имеют право и *назначать* новые значения полям строки, но при этом процедуры категории BEFORE, в отличие от AFTER, требуют дополнительного логического чтения блока с данными из БД, что сказывается на скорости их выполнения.

Триггерные процедуры не могут создаваться для таблиц пользователя SYS.

Некоторые другие особенности перечислены ниже.

11.3.1. Объединенные триггерные процедуры

Одну и ту же триггерную процедуру можно связать одновременно с любой комбинацией из трех основных событий DML. Это удобно, когда по замыслу разработчика тела процедур для INSERT, UPDATE или DELETE совпадают полностью или почти.

Например, пусть требуется обеспечить поступление в таблицу EMP имен сотрудников, записанных в «стандартном» формате (здесь — даваемом функцией INITCAP). Эту задачу выполнит триггерная процедура на INSERT и на UPDATE emp:

```
CREATE OR REPLACE TRIGGER emp_ename_right_in
BEFORE INSERT OR UPDATE OF ename ON emp
FOR EACH ROW
BEGIN
  :NEW.ename := INITCAP ( :NEW.ename );
end;
/
```

Часто такое построение присуще триггерным процедурам, играющим регистрирующую событие роль. Но в теле такой объединенной процедуры нередко бывает необходимо решать и обратную задачу: определения класса конкретного события, приведшего к срабатыванию. Этой цели служат специальные логические функции (предикаты) INSERTING, UPDATING и DELETING. Пример:

```
CREATE OR REPLACE TRIGGER emp_manipulation_logging
```

```
AFTER DELETE OR UPDATE OR INSERT ON emp
FOR EACH ROW
DECLARE
 dmltype CHAR ( 1 );
BEGIN
 dmltype :=  CASE
             WHEN DELETING  THEN 'D'
             WHEN UPDATING  THEN 'U'
             WHEN INSERTING THEN 'I'
             END;
 /*** регистрация события, например что-нибудь вроде:
       INSERT INTO emp_log ( emp_no,     who,  tstamp,  operation )
              VALUES       ( :new.empno, USER, SYSDATE, dmltype )
       ;
 ***/
END;
/
```

Функция UPDATING допускает уточнение именем столбца, например:

IF **UPDATING (comm)** THEN …

В этом исполнении она полезна и в одиночной триггерной процедуре для UPDATE, а не только в объединенной.

11.3.2. Управление транзакциями в теле триггерной процедуры

В теле триггерной процедуры не запрещено выдавать операторы DML, но запрещено выдавать COMMIT и ROLLBACK: непосредственно или через подпрограммы, — если только в заголовке не указана прагма автономной транзакции. При указании PRAGMA AUTONOMOUS_TRANSACTION команды управления транзакциями могут употребляться:

```
CREATE OR REPLACE TRIGGER before_department_update
        BEFORE UPDATE ON dept
        FOR EACH ROW
DECLARE
        PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
        …
        COMMIT;
        …
        ROLLBACK;
END;
/
```

Однако возможна и другая техника: объявить триггерную процедуру как обычно, но вставить в тело обращения к процедурам, транслированным для автономных транзакций.

Обращение к автономным транзакциям в триггерных процедурах позволяет программировать безоткатные операции в БД (например, запись в журнал), делать запросы и вносить правки в таблицу, к которой привязана эта процедура и решать другие задачи.

11.3.3. Последовательность срабатывания триггерных процедур

Когда триггерных процедур для операции DML над таблицей имеется несколько разных категорий, их порядок срабатывания при выполнении операции таков:

1) Триггерные процедуры категории BEFORE уровня *оператора*

2) ***Для каждой строки***, затронутой изменением:
   Строковые триггерные процедуры категории BEFORE
   [Операция DML со строкой]
   Строковые триггерные процедуры категории AFTER
3) Триггерные процедуры категории AFTER уровня *оператора*

При наличие на *одном* уровне нескольких триггерных процедур одной категории порядок срабатывания лучше полагать «произвольным» (на самом деле он вполне определен правилом, по которому первой срабатывает процедура с более поздним временем трансляции, однако в жизни такой порядок крайне ненадежен).

С версии 11 открылась возможность порядок срабатывания на одном уровне задать явно фразой FOLLOWS в команде CREATE TRIGGER.

Упражнение. Проделать пример:

```
CREATE OR REPLACE TRIGGER before_update_emp_1 BEFORE UPDATE ON emp
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'Fired before_update_emp_1' );
END;
/
CREATE OR REPLACE TRIGGER before_update_emp_2 BEFORE UPDATE ON emp
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'Fired before_update_emp_2' );
END;
/
CREATE OR REPLACE TRIGGER before_update_row_emp BEFORE UPDATE ON emp
FOR EACH ROW
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'Fired before_update_row_emp for ' || :old.ename );
END;
/
CREATE OR REPLACE TRIGGER after_update_row_emp_1 AFTER UPDATE ON emp
FOR EACH ROW
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'Fired after_update_row_emp_1 for ' || :old.ename ) ;
END;
/
CREATE OR REPLACE TRIGGER after_update_row_emp_2 AFTER UPDATE ON emp
FOR EACH ROW
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'Fired after_update_row_emp_2 for ' || :old.ename );
END;
/
CREATE OR REPLACE TRIGGER after_update_emp AFTER UPDATE ON emp
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'Fired after_update_emp' );
END;
/

UPDATE emp SET sal = sal WHERE ROWNUM <= 3;
```

Упражнение. В версиях с 11:

1) выполнить:
```
CREATE OR REPLACE TRIGGER before_update_emp_1 BEFORE UPDATE ON emp
FOLLOWS before_update_emp_2
BEGIN DBMS_OUTPUT.PUT_LINE ( 'Fired before_update_emp_1' ); END;
/
```
   выполнить UPDATE как выше и сравнить результат.

2) выполнить:
```
CREATE OR REPLACE TRIGGER before_update_emp_1 BEFORE UPDATE ON emp
BEGIN DBMS_OUTPUT.PUT_LINE ( 'Fired before_update_emp_1' ); END;
```

```
/
CREATE OR REPLACE TRIGGER before_update_emp_2 BEFORE UPDATE ON emp
FOLLOWS before_update_emp_1
BEGIN DBMS_OUTPUT.PUT_LINE ( 'Fired before_update_emp_2' ); END;
/
```

выполнить UPDATE как выше и сравнить результат.

11.3.4. Составные триггерные процедуры

С версии 11 возможны «составные триггерные процедуры» (compound trigger), имеющие общую часть объявлений и до четырех вариантов кодовой части соответственно каждой из четырех категорий. Схема устройства в общем случае:

```
CREATE TRIGGER compound_trigger
FOR UPDATE OF sal ON emp
COMPOUND TRIGGER
  -- пример общего раздела объявлений
  threshold CONSTANT SIMPLE_INTEGER := 200;

-- процедурная часть
BEFORE STATEMENT IS
BEGIN
    ...
END BEFORE STATEMENT;

BEFORE EACH ROW IS
BEGIN
    ...
END BEFORE EACH ROW;

AFTER EACH ROW IS
BEGIN
    ...
END AFTER EACH ROW;

AFTER STATEMENT IS
BEGIN
    ...
END AFTER STATEMENT;

END compound_trigger;
/
```

Ниже приводится простой пример составной триггерной процедуры, когда присутствует всего одна (из четырех возможных) частей:

```
CREATE OR REPLACE TRIGGER too_much_sal_at_once
FOR UPDATE OR INSERT OF sal ON temp
COMPOUND TRIGGER

maxsal CONSTANT NUMBER := 9000;
sumsal          NUMBER := 0;

AFTER EACH ROW IS
BEGIN
sumsal := sumsal + :new.sal;
IF sumsal > maxsal THEN
   RAISE_APPLICATION_ERROR ( -20100, 'Too much total salary at once' );
END IF;
END AFTER EACH ROW;
```

```
END too_much_sal_at_once;
/
```

Организовать подобное ограничение на изменение данных одним оператором DML можно было и до версии Oracle 11, но это требовало использования дополняющего триггерную процедуру пакета, и тем самым было менее надежно и удобно. Анализ изменяемых строк «в совокупности» может оказаться в жизни более сложным.

В других случаях составные триггерные процедуры позволяют обойти проблему «мутирующей таблицы», возникающую при попытке обратиться в построчной триггерной процедуре к данным таблицы, ответственной за срабатывание.

11.3.5. Триггерные процедуры INSTEAD OF для представлений данных (виртуальных таблиц)

Обычные изменяющие операторы DML применяться не только к основным таблицам БД, но и к виртуальным («представлениям» данных, view). Это может относиться и к представлениям, построенным на основе «соединения» (join). Однако это распространяется не на все представления, а только на «обновляемые». Перечень обновляемых столбцов для каждого обновляемого представления можно посмотреть в USER_UPDATABLE_COLUMNS.

Общие неудобства прямого применения операторов DML к представлением данных восполняются наличием триггерных процедур типа INSTEAD OF, придуманных специально ради права обновлять абсолютно любое представление данных вообще. При этом, однако ответственность за программирование, а также формальную и содержательную корректность самого действия СУБД перекладывает на плечи разработчика.

Пример:

```
CREATE OR REPLACE VIEW empl
AS
  SELECT empno, ename, dname
  FROM   emp INNER JOIN dept USING ( deptno )
;

CREATE OR REPLACE TRIGGER empl_insert
INSTEAD OF INSERT ON empl
DECLARE
  dept# dept.deptno%TYPE;
BEGIN
  SELECT deptno INTO dept# FROM dept WHERE dname = :new.dname;
  INSERT INTO emp ( empno, ename, deptno )
           VALUES ( :new.empno, :new.ename, dept# )
  ;
EXCEPTION
  WHEN no_data_found THEN
    RAISE_APPLICATION_ERROR (
      -20106
    , 'Department ''' || :new.dname || ''' not exists'
    );
END emp_info_insert;
/
```

Замечание. Использованное выше обращение к RAISE_APPLICATION_ERROR правильно по существу, но неправильно методически. Для достижения более надежного кода *прямые* обращения к RAISE_APPLICATION_ERROR целесообразно заменить в программе на *опосредованные* вызовы к таким же обращениям, но собранным в отдельном одном месте (— в пакете). Здесь (и ниже) этого не сделано только ради простоты примера.

Сообразуясь с потребностями приложения, можно дать другое определение триггерной процедуры, автоматически создающее новый отдел при необходимости по следующей схеме:

```
ALTER TRIGGER empl_insert DISABLE
;
CREATE OR REPLACE TRIGGER empl_insert_autodept
INSTEAD OF INSERT ON empl
DECLARE
  dept# dept.deptno%TYPE;
BEGIN
  BEGIN
    SELECT deptno INTO dept# FROM dept WHERE dname = :new.dname;
  EXCEPTION
    WHEN no_data_found THEN
      INSERT INTO dept ( deptno, dname )
             VALUES ( dept_deptno_seq.NEXTVAL, :new.dname )
      RETURNING deptno INDO dept#;
  END;

  INSERT INTO emp ( empno, ename, deptno )
           VALUES ( :new.empno, :new.ename, dept# )
  ;
END empl_insert_autodept;
/
```

И далее процедуры EMPL_INSERT и EMPL_INSERT_AUTODEPT по мере надобности включать и выключать на альтернативной основе.

Триггерные процедуры категории INSTEAD OF позволяют иногда решать технические задачи, связанные с обновлением таблиц, а не представлений. Пусть, например, имеется программа, порождающая команды INSERT на основе показаний какого-то датчика, и другая программа, принимающая эти команды INSERT, и выполняющая их. Допустим, что первая программа порождает много ненужных команд, например, с нулевым показанием датчика, которые нам хотелось бы игнорировать. Обе программы, однако, написаны сторонним исполнителем и поправить их тексты не представляется возможным. В этом случае проблему можно попытаться решить на уровне исключительно БД.

Положим, мы хотим игнорировать добавление сотрудников, для которых не указаны имена. Следующие действия позволят это сделать без вмешательства в код приложений:

```
ALTER TABLE emp RENAME TO emp_table
;
CREATE VIEW emp
AS
   SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno
   FROM   emp_table
;
CREATE OR REPLACE TRIGGER ignore_nonames
INSTEAD OF INSERT ON emp
DECLARE
   rec emp_table%rowtype;
BEGIN
   IF :new.ename IS NOT NULL THEN
        INSERT INTO emp_table VALUES
                       ( empno, ename, job, mgr, hiredate, sal, comm, deptno )
        ;
   END IF;
END;
/
```

11.4. Триггерные процедуры для событий категории DDL

Могут определяться как для отдельной схемы, так и БД целиком.

Пример процедуры, запрещающей удаление таблиц типа SCOTT.EMP% (то есть, начинающихся с букв EMP):

```
CREATE OR REPLACE TRIGGER no_drop_trigger
BEFORE DROP ON scott.SCHEMA
DECLARE
```

```
v_msg VARCHAR2 ( 1000 ) :=
     'No drop allowed on '
  || DICTIONARY_OBJ_OWNER || '.'
  || DICTIONARY_OBJ_NAME || ' from '
  || LOGIN_USER
;
BEGIN
IF    DICTIONARY_OBJ_OWNER = 'SCOTT'
  AND DICTIONARY_OBJ_TYPE  = 'TABLE'
  AND DICTIONARY_OBJ_NAME  LIKE 'EMP%'
THEN
  RAISE_APPLICATION_ERROR ( -20905, v_msg );
END IF;
END;
/
```

Упражнение. Завести приведенную выше триггерную процедуру в схеме SYS и ее же в схеме SCOTT. Скопировать EMP в EMPX. Проверить работу процедуры путем удаления EMPX из схем SCOTT и SYS. Удалить процедуры и создать заново, заменив:

BEFORE DROP ON scott.SCHEMA
на
BEFORE DROP ON DATABASE.

Проверить работу триггерных процедур вторично.

11.5. Триггерные процедуры для событий уровня схемы и БД

Создаются с естественными ограничениями на уточнения BEFORE и AFTER.

Пример триггерной процедуры, которая сразу после подключения программы к схеме YARD переключает контекст именования на схему HR:

```
CREATE OR REPLACE TRIGGER set_hr_schema
AFTER LOGON ON yard.SCHEMA
BEGIN
  EXECUTE IMMEDIATE 'ALTER SESSION SET CURRENT_SCHEMA = hr';
  EXECUTE IMMEDIATE 'ALTER SESSION SET RECYCLEBIN = OFF'; -- для версии > 9
END;
/
```

(Теперь YARD сможет сразу после входа в систему обращаться к объектам схемы HR по короткому имени, без префикса HR, но зато для обращения к своим собственным объектам вынужден будет использовать префикс YARD).

Пример триггерной процедуры, срабатывающей при любом соединении с БД:

```
CREATE OR REPLACE TRIGGER logon_trigger
AFTER LOGON ON DATABASE
DECLARE
   client_identifier VARCHAR2 ( 64 );
BEGIN
   SELECT SYS_CONTEXT ( 'USERENV', 'OS_USER' )
          || '@' ||
          SYS_CONTEXT ( 'USERENV', 'IP_ADDRESS' )
   INTO client_identifier
   FROM dual
   ;
   DBMS_SESSION.SET_IDENTIFIER ( client_identifier );
END;
/
```

Здесь средствами «контекста сеанса» выставляется «идентификатор клиента», к которому прикладная программа всегда может обратиться впоследствии с помощью функции SYS_CONTEXT, например запросом:

```
SELECT SYS_CONTEXT ( 'USERENV', 'CLIENT_IDENTIFIER' ) FROM dual;
```

Пример триггерной процедуры, срабатывающей всякий раз после старта СУБД и запускающей службу с основным именем DBSERV (для примера), или же останавливающей ее, при обмене основной и резервной баз данных ролями:

```
CREATE OR REPLACE TRIGGER manage_service
AFTER STARTUP ON DATABASE
DECLARE
  role VARCHAR ( 30 );
BEGIN
  SELECT database_role INTO role FROM v$database;
  IF role = 'PRIMARY' THEN
    DBMS_SERVICE.START_SERVICE ( 'DBSERV' );
  ELSE
    DBMS_SERVICE.STOP_SERVICE ( 'DBSERV' );
  END IF;
END;
/
```

12. Пакеты в PL/SQL

Пакет — способ группировки программных объектов PL/SQL. Некоторые мотивировки для использования пакетов:

- возможность сокрытия информации
- объектный стиль программирования
- проектирование от общего к частному
- возможность независимой трансляции взаимозависимых программных единиц (пакетов)
- возможность сохранения переменных от транзакции к транзакции
- более высокая производительность по сравнению с отдельными процедурами.

Элементы, из которых состоит пакет:

- процедуры
- функции
- переменные и постоянные
- курсоры
- имена исключительных ситуаций
- предложения TYPE описания типов.

12.1. Общая структура пакета

Две основные компоненты пакета:

- внешнее описание, PACKAGE
- внутреннее описание (тело, реализация), PACKAGE BODY

Внешнее описание пакета перечисляет все объекты, которые могут использоваться внешними приложениями, вместе с необходимой для работы информацией. Синтаксис описания:

```
CREATE [OR REPLACE] PACKAGE имя_пакета
{IS | AS}
[AUTHID {CURRENT_USER | DEFINER}]
[ACCESSIBLE BY (…)]
[PRAGMA SERIALLY_REUSABLE;]
        [определения общедоступных типов, исключительных ситуаций, переменных;
        объявления курсоров, процедур, функций]
END [слово-комментарий];
```

Если в пакете имеются процедуры, либо функции, либо закрытый (private) код, должно присутствовать тело пакета (иначе его может и не быть). В теле, помимо реализации описанных в спецификации процедур, функций и курсоров, описываются, если имеются, закрытые для внешнего использования объекты и, возможно, раздел инициализации пакета. Синтаксис задания тела:

```
CREATE [OR REPLACE] PACKAGE BODY имя_пакета
{IS | AS}
[PRAGMA SERIALLY_REUSABLE;]
        [определения закрытых типов, исключительных ситуаций, переменных;
        определения закрытых курсоров, процедур, функций
        определения общедоступных курсоров, процедур, функций]
[BEGIN
        программный_код
[EXCEPTION
        обработчики_исключительных_ситуаций]]
END [слово-комментарий];
```

Если имеется *программный_код*, то он образует раздел инициализации пакета и исполняется только однажды при первом обращении к какому-нибудь элементу пакета.

Для работы сначала должна быть транслирована спецификация пакета, а затем тело. Полномочие EXECUTE, выдаваемое на пакет пользователю или PUBLIC, дает доступ только к элементам, упомянутым в PACKAGE.

Полный пример создания пакета:

```
CREATE OR REPLACE PACKAGE time_package
AS
   FUNCTION gettimestamp RETURN DATE;
   PROCEDURE settimestamp ( d IN DATE );
END time_package;
/

CREATE OR REPLACE PACKAGE BODY time_package
AS
   savedtimestamp DATE := SYSDATE;    -- скрытая переменная пакета

   FUNCTION gettimestamp RETURN DATE IS
   BEGIN
      RETURN savedtimestamp;
   END gettimestamp;

   PROCEDURE settimestamp ( d IN DATE ) IS
   BEGIN
      savedtimestamp := d;
   END settimestamp;
END time_package;
/
```

Перенесение описания переменной STARTTIMESTAMP из тела пакета в интерфейсную часть сделает ее доступной для внешних программ непосредственно. Эквивалентный функционально пакет мог бы выглядеть в данном случае значительно проще:

```
CREATE OR REPLACE PACKAGE time_package1
AS
   savedtimestamp DATE := SYSDATE;    -- общедоступная переменная пакета
END time_package1;
/
```

Однако, в общем подобная практика предоставления прямого доступа к переменным пакета методически не оправдана в силу того, что может повлечь необходимость правки прикладных программ при изменениях в постановке задачи. Исключения из общей практики следует употреблять конкретно и обосновано.

Например, пакет TIME_PACKAGE1, в отличие от TIME_PACKAGE, беспомощен в соблюдении для переменной SAVEDTIMESTAMP значений только из области рабочего времени (скажем, с 8 утра до 4 пополудни), а не произвольного времени суток вообще. В TIME_PACKAGE изменение коснулось бы только процедуры SETTIMESTAMP:

```
PROCEDURE settimestamp ( d IN DATE ) IS
dbeg      DATE;
dend      DATE;
shortmask VARCHAR2 ( 30 ) := 'yyyy-mm-dd';
BEGIN
dbeg := TO_DATE ( TO_CHAR ( d, shortmask ) || ' 08:00:00'
                , shortmask || ' hh24:mi:ss'
                );
dend := TO_DATE ( TO_CHAR ( d, shortmask ) || ' 16:00:00'
                , shortmask || ' hh24:mi:ss'
```

```
                );

savedtimestamp := LEAST ( GREATEST ( d, dbeg ), dend );
END settimestamp;
```

В БД подобное обеспечивается техникой ограничений целостности данных таблиц.

12.2. Обращение к элементам пакета

К элементам из спецификации пакета можно обращаться из внешних процедур так:

имя_пакета.*имя_элемента*

Например:

DBMS_OUTPUT.PUT_LINE ('This is a parameter data');
time_package.settimestamp (SYSDATE + 1);

12.3. Данные пакета

Данные пакета — это данные, объявленные в PACKAGE или в PACKAGE BODY вне тел подпрограмм. Область их доступности ограничивается сеансом, то есть они являются глобальными для сеанса связи с БД. При этом для данных пакета полезно учитывать следующее:

- на них не распространяется действие COMMIT и ROLLBACK
- курсор, открытый где-нибудь в пакете остается в состоянии OPEN, пока не будет закрыт явно или пока не завершится сеанс
- хорошей практикой считается прятать описания структур данных в тело пакета, предоставляя для пользователей пакета процедуры типа Set и Get для обращения с данными

12.4. Рекурсия, взаимные вызовы и повторения имен

Для подпрограмм в составе пакета:

- рекурсивный вызов *разрешен*, глубина рекурсии не регламентирована и зацикленность при исполнении кода СУБД не контролирует;
- взаимный вызов *разрешен* и зацикленность при исполнении кода СУБД не контролирует;
- повторение имен *разрешено*.

Пример повторения имен в определении процедур из системного пакета DBMS_OUTPUT (до версии 10):

```
PACKAGE DBMS_OUTPUT
IS
        PROCEDURE PUT_LINE ( a VARCHAR2 );
        PROCEDURE PUT_LINE ( a NUMBER );
        PROCEDURE PUT_LINE ( a DATE );          -- была в ранних  версиях
END;
```

Здесь решение о том, к какой из трех процедур PUT_LINE поставить обращение, компилятор принимает на основе анализа типа параметра.

С версии 10 в пакет DBMS_OUTPUT внесена корректировка: оставлена всего одна процедура PUT_LINE, для аргумента-строки, а в основной программе при необходимости выполняется автоматическое приведение нестрокового значения к VARCHAR2 с помощью функции TO_CHAR (то есть выполняется неявное преобразование типа). Однако само название TO_CHAR по-прежнему используется для определения нескольких «системных» функций, различающихся типами и количеством аргументов.

12.5. Код начального исполнения в пакете

При первом обращении к какому-либо элементу пакета он весь (его интерпретируемый «m-код») целиком помещается в SGA экземпляра СУБД. С этого момента весь код пакета становится немедленно (без дополнительного обращения в БД) доступным для всех пользователей, обладающих привилегией EXECUTE на пакет.

Если в пакете объявлены данные, они размещаются в области UGA (сеанса), которая физически может располагаться в SGA или в PGA, в зависимости от способа соединения клиента с сервером. После этого, при наличии в определении тела пакета кода начального исполнения, автоматически запускается этот код. Более одного раза за сеанс он запускаться не будет.

Пример кода начального исполнения:

```
CREATE OR REPLACE PACKAGE BODY time_package
AS
   savedtimestamp DATE;

   FUNCTION gettimestamp RETURN DATE IS
   BEGIN
      RETURN savedtimestamp;
   END gettimestamp;

   PROCEDURE settimestamp ( d IN DATE ) IS
   dbeg      DATE;
   shortmask VARCHAR2 ( 30 ) := 'yyyy-mm-dd';
   BEGIN
      dbeg := TO_DATE ( TO_CHAR ( d, shortmask ) || ' 08:00:00'
                      , shortmask || ' hh24:mi:ss'
                      );
      savedtimestamp := GREATEST ( d, dbeg );
   END settimestamp;

BEGIN
   settimestamp ( SYSDATE );
END time_package;
/
```

В других случаях код начального исполнения мог бы считывать данные из БД, порта USB с целью получения ключа шифрования, и так далее.

12.6. Прагма SERIALLY_REUSABLE

Если переменные пакета не используются для передачи подпрограммами значений друг другу, а употребляются только в процессе выполнения подпрограмм, полезно в описании пакета указать PRAGMA SERIALLY_REUSABLE. В этом случае значения данных пакета будут сохраняться только на время выполнения подпрограммы, и при каждом обращении к подпрограмме инициализироваться заново. В частности, после каждого завершения подпрограммы на PL/SQL будут закрываться курсоры и освобождаться занимаемая ими память.

Технически при использовании прагмы SERIALLY_REUSABLE рабочие области данных пакета будут размещаться не в UGA, как обычно, а в SGA СУБД, и будет повторно использоваться при наличии разных сеансов, одновременно работающих с элементами пакета. В противном случае совокупная память, требуемая для работы пакета, будет расти пропорционально числу таких сеансов.

Пример использования SERIALLY_REUSABLE. Создадим в SQL*Plus файл *testpack.sql* с описанием проверочного пакета:

```
CREATE OR REPLACE PACKAGE testpack IS
&1.
PROCEDURE putval ( s VARCHAR2 );
FUNCTION getval RETURN VARCHAR2;
END;
.
SAVE testpack REPLACE
CREATE OR REPLACE PACKAGE BODY testpack IS
&1.
package_value VARCHAR2 ( 100 ) DEFAULT 'initial value';

PROCEDURE putval ( s VARCHAR2 ) IS
BEGIN package_value := s;
END;

FUNCTION getval RETURN VARCHAR2 IS
BEGIN RETURN package_value;
END;
END;
.
SAVE testpack APPEND
SET VERIFY OFF
SET SERVEROUTPUT ON
```

Проверка:

```
SQL> SET FEEDBACK OFF
SQL> @testpack ' '                          -- пакет без прагмы сериализации
SQL> EXECUTE testpack.putval ( 'some value' ); -
> DBMS_OUTPUT.PUT_LINE ( testpack.getval )
some value
SQL> EXECUTE testpack.putval ( 'some value' )
SQL> EXECUTE DBMS_OUTPUT.PUT_LINE ( testpack.getval );
some value
SQL> @testpack 'PRAGMA SERIALLY_REUSABLE;' -- добавляем прагму
SQL> EXECUTE testpack.putval ( 'some value' ); -
> DBMS_OUTPUT.PUT_LINE ( testpack.getval )
some value
SQL> EXECUTE testpack.putval ( 'some value' )
SQL> EXECUTE DBMS_OUTPUT.PUT_LINE ( testpack.getval );
initial value
SQL> SET FEEDBACK ON
```

Неименованный блок в данном случае играет роль подпрограммы. Очевидно, наличие прагмы SERIALLY_REUSABLE в случае последовательных обращений к подпрограммам пакета несущественно. Однако если обращение происходит из другой подпрограммы, то при отсутствии прагмы значения, установленные прежними подпрограммами, доступны; при наличии же прагмы эти значения не видны (были утеряны при выходе из подпрограммы).

13. Редакции именованных программных единиц

В версии 11.2 в Oracle введена техника работы с редакциями групп объектов. Из программных единиц эта техника распространяется на следующие их виды:

- PROCEDURE
- FUNCTION
- TRIGGER
- PACKAGE/PACKAGE BODY
- TYPE/TYPE BODY.

(Кроме них редакции допускаются для объектов БД вида VIEW, SYNONYM и LIBRARY).

Основное предназначение механизма редакций для хранимых объектов заключается в отладке изменений в схеме БД («новая редакция» объектов БД) без остановки работы основного приложения («старая редакция»).

Ниже приводятся примеры составления редакций для хранимых подпрограмм и триггерных процедур.

13.1. Подготовка схемы для редакций объектов

Ниже приводятся команды заведения схемы для объектов разных редакций и выполнения необходимых сопутствующих действий.

```
CONNECT / AS SYSDBA
CREATE USER yard IDENTIFIED BY pass;

GRANT CONNECT, RESOURCE, CREATE VIEW TO yard;

CREATE TABLE yard.emp AS SELECT * FROM scott.emp;

GRANT CREATE ANY EDITION, DROP ANY EDITION TO yard;

ALTER USER yard ENABLE EDITIONS;

CONNECT yard/pass
CREATE EDITION app_release_1;

GRANT USE ON EDITION app_release_1 TO scott;
```

В схеме YARD появилась таблица EMP с тою же структурой, что одноименная в схеме SCOTT и с теми же данными (но без ограничений целостности).

Заведем две редакции представления данных из YARD.EMP:

```
ALTER SESSION SET EDITION = ora$base;
CREATE OR REPLACE EDITIONING VIEW emp_eview
  AS
SELECT empno, ename, deptno FROM emp
;
ALTER SESSION SET EDITION = app_release_1;
CREATE OR REPLACE EDITIONING VIEW emp_eview
  AS
SELECT empno, ename FROM emp
;
```

13.2. Создание редакций процедур

Заведение разных редакций одной и той же процедуры в схеме со свойством EDITIONS_ENABLED = TRUE выглядит достаточно прозрачно. Так, для добавления данных о сотрудниках можно завести две редакции процедуры INSERT_EMPLOYEE следующим образом:

```
ALTER SESSION SET EDITION = ora$base;
CREATE OR REPLACE PROCEDURE insert_employee (
  eno NUMBER
, ename VARCHAR2
, dno NUMBER
)
AS
BEGIN
INSERT INTO yard.emp_eview
   ( empno, ename, deptno ) VALUES ( eno, ename, dno )
;
END;
/
GRANT EXECUTE ON insert_employee TO scott;

ALTER SESSION SET EDITION = app_release_1;
CREATE OR REPLACE PROCEDURE insert_employee (
  eno NUMBER
, ename VARCHAR2
)
AS
BEGIN
INSERT INTO yard.emp_eview
   ( empno, ename ) VALUES ( eno, ename )
;
END;
/
GRANT EXECUTE ON insert_employee TO scott;
```

Проверка:

```
SQL> CONNECT scott/tiger
Connected.
SQL> ALTER SESSION SET EDITION = ora$base;

Session altered.

SQL> EXECUTE yard.insert_employee ( 1111, 'OBAMA', 10 )

PL/SQL procedure successfully completed.

SQL> ROLLBACK;

Rollback complete.

SQL> ALTER SESSION SET EDITION = app_release_1;

Session altered.

SQL> EXECUTE yard.insert_employee ( 1111, 'OBAMA' )

PL/SQL procedure successfully completed.

SQL> ROLLBACK;

Rollback complete.
```

Откат транзакций сделан (а) чтобы сохранить прежние данные, и (б) в первом случае — чтобы закрыть транзакцию перед переключением на новую редакцию.

13.3. Создание редакций триггерных процедур

Теперь для добавления в БД данных о сотрудниках создадим две редакции триггерных процедур. Это делается аналогично обычным процедурам. Прикладной смысл триггерных процедур в данном случае состоит в нормализации имен сотрудников перед помещением в базу.

```
CONNECT yard/pass
ALTER SESSION SET EDITION = ora$base;

CREATE OR REPLACE TRIGGER empl_insert
BEFORE INSERT ON emp_eview
FOR EACH ROW
BEGIN
   :new.ename := INITCAP ( :new.ename );
END;
/
GRANT INSERT, SELECT ON emp_eview TO scott;
```

Обратите внимание, что для редактируемых представлений (EDITIONING VIEW) в триггерных процедурах *не действует* привязка к событию INSTEAD OF, как для обычных представлений, а вместо этого BEFORE и AFTER, как для основных таблиц. Это одно из проявлений особости редактируемых представлений от обычных.

Вторая редакция:

```
ALTER SESSION SET EDITION = app_release_1;

CREATE OR REPLACE TRIGGER empl_insert
BEFORE INSERT ON emp_eview
FOR EACH ROW
BEGIN
   :new.ename := LOWER ( :new.ename );
END;
/
GRANT INSERT, SELECT ON emp_eview TO scott;
```

Проверку можно выполнить следующей последовательностью команд в SQL*Plus:

```
CONNECT scott/tiger
ALTER SESSION SET EDITION = ora$base;
INSERT INTO yard.emp_eview VALUES ( 1111, 'OBAMA', 10 );
SELECT * FROM yard.emp_eview WHERE empno = 1111;
ROLLBACK;

ALTER SESSION SET EDITION = app_release_1;
INSERT INTO yard.emp_eview VALUES ( 1111, 'OBAMA' );
SELECT * FROM yard.emp_eview WHERE empno = 1111;
ROLLBACK;
```

13.4. Перекрестные триггерные процедуры для разных редакций

Когда отлаживается работа приложения с новой редакцией объектов БД, какое-то время обе редакции объектов (старая и новая) сосуществуют. Сложность в том, что работа с новой редакцией не должна портить данные, с которыми продолжает иметь дело старый вариант приложения. Если планируемые изменения в схеме однозначно взаимообратимы с исходным состоянием, помочь в этом способны перекрестные

триггерные процедуры для разных редакций (межредакционные триггерные процедуры; crossedition triggers, CET).

Рассмотрим пример подготовки к изменению структуры таблицы EMP в схеме YARD. Предположим, требуется хранить в БД самостоятельно сведения о должностях, как например максимальную зарплату и тому подобное. Ради этого придется завести отдельную новую таблицу с данными о должностях, а из таблицы EMP изъять столбец с названием должности сотрудника, и добавить ему на замену ссылку на сведения о должностях. Пока новая редакция приложения не будет объявлена основной, старый столбец придется какое-то время сохранять.

Подобное разбиение одной таблицы сотрудников на две — сотрудников и, отдельно, должностей — очевидно обратимо, так что на время отладки будет удобно воспользоваться межредакционными триггерными процедурами. Они будут отвечать при работе со старой редакцией за параллельное внесение изменений в новые структуры, а при работе с новой редакцией — в старые, обеспечивая в БД возможность предоставления и старого, и нового «взгляда» на данные.

13.4.1. Подготовка таблиц

Создадим таблицу должностей и добавим в таблицу сотрудников ссылку при том, что столбец с названием должности оставим до будущего перехода на новую редакцию приложения. Выполним в SQL*Plus:

```
CONNECT yard/pass
CREATE TABLE job (
  jobid  NUMBER   ( 2 ) PRIMARY KEY
, jname  VARCHAR2 ( 9 )
, maxsal NUMBER   ( 7, 2 )
);
INSERT INTO job ( jobid, jname, maxsal ) VALUES ( 1,   'ANALYST', 3500 );
INSERT INTO job ( jobid, jname, maxsal ) VALUES ( 2,     'CLERK', 2000 );
INSERT INTO job ( jobid, jname, maxsal ) VALUES ( 3,   'MANAGER', 3000 );
INSERT INTO job ( jobid, jname, maxsal ) VALUES ( 4, 'PRESIDENT', 6000 );
INSERT INTO job ( jobid, jname, maxsal ) VALUES ( 5,  'SALESMAN', 2000 );

ALTER TABLE emp ADD ( jobno NUMBER ( 2 ) REFERENCES job ( jobid ) );

UPDATE emp SET jobno = ( SELECT jobid FROM job WHERE jname = emp.job );
```

Переименуем таблицу сотрудников, и отдадим ее старое имя двум редакциям представления:

```
ALTER TABLE emp RENAME TO emp_tab;

ALTER SESSION SET EDITION = ora$base;
CREATE OR REPLACE EDITIONING VIEW emp
AS
SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno FROM emp_tab
;
GRANT INSERT, UPDATE, DELETE, SELECT ON emp TO scott;

ALTER SESSION SET EDITION = app_release_1;
CREATE OR REPLACE EDITIONING VIEW emp
AS
SELECT empno, ename, jobno, mgr, hiredate, sal, comm, deptno FROM emp_tab
;
GRANT INSERT, UPDATE, DELETE, SELECT ON emp TO scott;
```

13.4.2. Создание перекрестных межредакционных триггерных процедур

Одна из создаваемых ниже триггерных процедур отвечает за правку данных, необходимую для работы старой редакции приложений, во время работы нового, а вторая наоборот. Существенно, что транслироваться обе перекрестные процедуры должны в новой редакции:

```
ALTER SESSION SET EDITION = app_release_1;

CREATE OR REPLACE TRIGGER cross_forward_job
BEFORE INSERT OR UPDATE ON emp_tab
FOR EACH ROW
FORWARD CROSSEDITION
BEGIN
  SELECT jobid INTO :new.jobno FROM job WHERE :new.job = jname;
END;
/

CREATE OR REPLACE TRIGGER cross_reversed_job
BEFORE INSERT OR UPDATE ON emp_tab
FOR EACH ROW
REVERSE CROSSEDITION
BEGIN
  SELECT jname INTO :new.job FROM job j WHERE :new.jobno = jobid;
END;
/
```

Проверку способен организовать следующий код:

```
CONNECT scott/tiger
ALTER SESSION SET EDITION = ora$base;
INSERT INTO yard.emp ( empno, ename, job ) VALUES ( 1111, 'OBAMA', 'CLERK' );
COMMIT;
ALTER SESSION SET EDITION = app_release_1;
INSERT INTO yard.emp ( empno, ename, jobno ) VALUES ( 2222, 'LADEN', 2 );
COMMIT;
```

Как и раньше, если транзакция успела изменить какие-нибудь данные в БД, то для настройки на новую редакцию ее потребуется сначала закрыть. В результате получим:

```
SQL> SELECT * FROM yard.emp WHERE empno IN ( 1111, 2222 );

  EMPNO ENAME    JOBNO      MGR HIREDATE      SAL    COMM   DEPTNO
------- ------- ------ -------- --------- ------- ------- --------
   1111 OBAMA        2
   2222 LADEN        2

SQL> ALTER SESSION SET EDITION = ora$base;

Session altered.

SQL> SELECT * FROM yard.emp WHERE empno IN ( 1111, 2222 );

     EMPNO ENAME   JOB           MGR HIREDATE      SAL    COMM   DEPTNO
---------- ------- ------- -------- --------- ------- ------- --------
      1111 OBAMA   CLERK
      2222 LADEN   CLERK
```

При работе со старой редакцией воспроизводится поведение старой таблицы EMP, а при работе с новой — с тою же таблицей, но в новом варианте.

13.4.3. Дополнительные замечания по технологии

Приведенные примеры перекрестных триггерных процедур были намерено упрощены. В жизни в них следовало бы предусмотреть реакцию на указание в качестве нового значения отсутствующей должности. Предположим, что в старом приложении подобная обработка не программировалась, то есть в БД добавлялась ровно то название должности, которое было указано в INSERT/UPDATE. Тогда для сохранения поведения старой реакции приложения следовало бы на возникающую в SELECT … INTO ... FROM job ошибку NO_DATA_FOUND среагировать добавлением новой записи в таблицу JOB. Придется решить технический вопрос о поставке значений в JOBID; это может потребовать употребления генератора последовательности чисел (sequence) и прочих усложнений.

Далее. Иногда в теле перекрестной триггерной процедуры полезно воспользоваться информацией о редакции, в рамках которой она выполняется, и о редакции, которой принадлежит объект, послуживщий причиною срабатывания процедуры. Для перекрестных триггерных процедур это не одно и то же.

Определить обе редакции (сеанса, где сработала триггерная процедура, и той, для которой исполняется ее код) можно из двух нарочно созданных параметрах сеанса USERENV. Вот примеры обращений, которые можно вставить с указанной целью в код процедуры:

```
SYS_CONTEXT ( 'USERENV', 'SESSION_EDITION_NAME' )
SYS_CONTEXT ( 'USERENV', 'CURRENT_EDITION_NAME' )
```

Когда отладка работы с новой редакцией приложения завершена и решение перейти на нее созрело, перекрестные триггерные процедуры и редакционные представления данных следует удалить, а освободившееся имя EMP вернуть основной таблице примерно следующим образом:

```
CONNECT yard/pass
ALTER SESSION SET EDITION = ora$base;
DROP VIEW emp;

ALTER SESSION SET EDITION = app_release_1;
DROP VIEW emp;
DROP TRIGGER cross_reversed_job;
DROP TRIGGER cross_forward_job;

ALTER TABLE emp_tab RENAME TO emp;
ALTER TABLE emp DROP COLUMN job;

CONNECT / AS SYSDBA
ALTER DATABASE DEFAULT EDITION = app_release_1;
```

14. Вызов функций PL/SQL в предложениях SQL

Хранимые в БД функции можно использовать в SQL-выражениях, подобно тому, как это делается с DECODE, NVL, SIN и прочими. Синтаксис обращения к хранимой функции внутри SQL:

[*имя_схемы*.][*имя_пакета*.]*имя_функции*[*@имя_связи_БД*][(*список_параметров*)]

14.1. Требования и ограничения на применение функций пользователей в SQL

Для использования в SQL хранимых функций необходимо выполнение ряда требований и существует ряд ограничений:

- Все формальные параметры обязаны быть с режимом IN, но не IN OUT и не OUT;
- Все «обращенные наружу» типы данных (параметров и возвращаемого значения) должны быть типами БД (то есть не иметь тип BOOLEAN, записи, BINARY_INTEGER и пр.);
- Не содержать команды DML, COMMIT/ROLLBACK, SET ROLE, ALTER SYSTEM, DDL прямо или косвенно (через обращения к другим подпрограммам) — при условии что функция не транслирована как AUTONOMOUS TRANSACTION;
- Во множественном (не однострочном) INSERT и в UPDATE и DELETE функция не имеет право, к тому же, запрашивать таблицу, изменяемую операцией;
- В версиях 10- передача параметров допускается только их позиционным перечислением;
- В версиях Oracle 8- хранимые функции, принадлежащие пакетам, должны иметь в своих заголовках предложение PRAGMA RESTRICT_REFERENCES.

14.2. Обращение к функциям в SQL и погоня за эффективностью; версии 12+

Если к функции на PL/SQL программист хочет обратиться в выражении SQL, а сам запрос SQL встроен в код PL/SQL, то формально ничего особенного для этого не требуется. Но с точки зрения эффективности выполнения запроса с такой функцией в отдельных случаях (при большой частоте обращений) могут обнаружиться неприемлимые накладные расходы, тормозящие обработку. Они вызваны тем, что пользовательские функции вычисляются машиной PL/SQL в своем контексте, а запрос, машиной SQL, в своем; частые переключения между контекстами при вычислении запроса SQL будут приводить к общему замедлению. Решения для борьбы с замедлением появились в версии 12.

Когда речь идет о функции, определенной в программе, выходом может стать фраза WITH в запросе:

```
WITH
  FUNCTION with_function ( p_id IN NUMBER ) RETURN NUMBER IS
  BEGIN
    RETURN p_id;
  END;
SELECT AVG ( with_function ( object_id ) )
FROM   all_objects
/
```

Замечание. Из-за наличия фразы WITH, символ ';' в SQL*Plus не будет восприниматься как знак окончания набора текста запроса.

Во фразу WITH можно включать и процедуры, например:

```
WITH
  PROCEDURE with_procedure ( p_id IN NUMBER ) IS
  BEGIN
    IF p_id BETWEEN 10000 AND 10020
      THEN DBMS_OUTPUT.PUT_LINE ( p_id );
      ELSE NULL;
```

```
    END IF;
  END;

  FUNCTION with_function ( p_id IN NUMBER ) RETURN NUMBER IS
  BEGIN
    with_procedure ( p_id );
    RETURN p_id;
  END;
SELECT AVG ( with_function ( object_id ) )
FROM   all_objects
/
```

Оговорки:

1) Так построенные запросы нельзя статически вставлять в код на PL/SQL — только динамически.
2) Если подобный запрос используется в качестве подзапроса, в главный запрос требуется вставить подсказку WITH_PLSQL (`/*+ WITH_PLSQL */`).

Когда речь идет о хранимой функции, ускорению обращения к ней в запросе служит особая прагма:

```
CREATE OR REPLACE
FUNCTION normal_function ( p_id IN NUMBER ) RETURN NUMBER
IS
  PRAGMA UDF;
BEGIN
  RETURN p_id;
END;
/
```

14.3. Обращение в SQL к функциям из состава пакетов

Для версий Oracle 8.0 и предшествующих хранимые функции, принадлежащие пакетам, должны иметь в своем описании явное указание качества изменений (purity level), определяемого для функции. Начиная с версии 8.1, если такое указание отсутствует, оно будет проверяться PL/SQL-машиной автоматически во время выполнения программы или же их могут восполнять другие указания типа DETERMINISTIC или PARALLEL_ENABLE.

Синтаксис указания качества изменений в пакетной функции:

PRAGMA RESTRICT_REFERENCES (*имя_программы* | DEFAULT, *качество_изменений*);

Ключевое слово DEFAULT распространяет указываемый уровень качества изменений на все программы в пакете или все методы объектных типов.

Каждый из пяти имеющихся уровней качества изменений характеризует определенную степень свободы подпрограммы от побочных эффектов:

Purity level	***Описание***	***Степень ограничения в программном коде***
WNDS	Write No Database State	Отсутствуют операции INSERT, UPDATE, DELETE
RNDS	Read No Database State	Отсутствуют операции SELECT
WNPS	Write No Package State	Не делаются изменения переменных пакета
RNPS	Read No Package State	Отсутствуют обращения к переменным пакета
TRUST		Сообщение компилятору «поверить» продекларированному уровню качества изменений «на слово» и не отрабатывать специальных действий, ограждающих от побочных эффектов

(Уровень TRUST имеется только начиная с версии 8.1).

Для разных мест использования в SQL пакетной функции требуется иметь различный минимальный уровень качества производимых изменений:

- для использования в SQL-предложениях вообще, все хранимые функции должны быть транслированы с указанием WNDS

- все функции, используемые не в SELECT, VALUES или SET, должны быть транслированы с указанием WNPS
- чтобы иметь возможность быть вызванной с удаленного сервера, функция должна быть транслирована с указанием RNPS
- чтобы иметь возможность быть обработанной параллельно, функция должна быть транслирована с указанием одновременно четырех главных уровней (WNDS, RNDS, WNPS, RNPS) или PARALLEL_ENABLE (в Oracle 8.1)
- функция не должна обращаться к подпрограммам, имеющим более низкий уровень качества изменений
- в Oracle 7 в случае, если пакет содержит раздел инициализации, он также должен быть помечен определенным уровнем качества изменений
- подменяемые (overloaded) функции не обязаны иметь одинаковый уровень качества изменений, но должны иметь его (возможно, каждая — свой) все из них

Не все системные пакеты в Oracle имеют продекларированный уровень WNPS или RNPS (например, пакеты DBMS_OUTPUT, DBMS_PIPE, DBMS_SQL), и поэтому использование подпрограмм из них в SQL имеет свои ограничения.

14.4. Разрешение конфликта имен столбцов и функций

Если функция без параметров и ее имя совпадает с именем столбца, конфликт имен в SQL-предложении будет разрешен в пользу столбца. Для того, чтобы поменять решение Oracle в пользу функции, достаточно уточнить ее имя именем схемы, или же снабдить скобками для параметров, пусть и отсутствующих. Поясняющий пример:

```
CREATE FUNCTION ename RETURN VARCHAR2 AS
BEGIN RETURN 'I am a function'; END;
/
-- прочитается как имя столбца:
SELECT ename FROM emp;
-- прочитается как имя функции:
SELECT scott.ename FROM emp;
-- тоже прочитается как имя функции:
SELECT ename ( ) FROM emp;
```

К сожалению, в SQL допускается конфликт между именами отдельно транслированной функции и пакетированной. Он разрешается в пользу пакетированной:

```
CREATE OR REPLACE PACKAGE scott
IS FUNCTION ename RETURN VARCHAR2;
END;
/
CREATE OR REPLACE PACKAGE BODY scott IS
   FUNCTION ename RETURN VARCHAR2 IS
   BEGIN RETURN 'I am a packaged function'; END ename;
END;
/
-- теперь прочитается как имя пакетированной функции:
SELECT scott.ename FROM DUAL;
```

В отдельных случаях замаскировать самостоятельную функцию ENAME можно и не создавая пакета:

```
DROP PACKAGE scott;
CREATE TABLE scott AS SELECT * FROM emp;
-- функция ENAME снова не видна:
SELECT scott.ename FROM scott;
```

Если в приложении отказаться от использования самостоятельных функций в выражениях SQL, а иметь дело лишь с функциями из пакетов, часть проблем конфликта имен уходит. Помогает также указание пустых скобок после имени функции.

14.5. Табличные функции в SQL

Табличными в PL/SQL называются функции, принимающие в качестве аргумента коллекции и/или возвращающие коллекции. Их определение и способ обращения к ним из SQL-предложения имеют свои особенности. Более предметно будут рассмотрены после темы «Коллекции».

15. Составные типы данных: коллекции

Коллекции в PL/SQL используются для работы с группами значений и устроены заметно сложнее данных простых типов.

В PL/SQL имеется три вида коллекций, два из которых унаследованы из Oracle SQL. Все они представляют из себя одномерные массивы (векторы) однотипных элементов с нижеследующими особенностями.

Ассоциативный массив (associative array)

Доступен только в программах PL/SQL и не может храниться в БД. Ассоциативный массив может быть разреженным, то есть индексы элементов в нем не обязаны идти подряд. Числовые индексы элементов могут браться из диапазона от -2147483648 до 2147483647. До версии 9.2 имелся только один частный случай ассоциативного массива, называвшийся *индексированной таблицей (index-by tables;* до версии 8 он же — *таблица PL/SQL)*.

Вложенные таблицы (nested tables)

Со вложенными таблицами можно не только работать в PL/SQL, но их можно и хранить в БД в виде столбцов или таблиц БД. При своем создании они плотно заполнены (последовательные значения индекса), но в процессе использования «внутренние» элементы можно удалять. Индексы элементов могут браться из диапазона от 1 до 2147483647.

Массивы типа VARRAY

Массив, доступный для работы как в PL/SQL, так и в БД. Заполнение всегда плотное. В отличие от вложенных таблиц нумерация элементов при запоминании, а затем извлечении из БД сохраняется. Максимальный индекс может браться из диапазона от 1 до 2147483647.

15.1. Синтаксис объявления типов для коллекций

Коллекции в Oracle объявляются через механизм типов. Сначала предложением TYPE в разделе объявлений блока PL/SQL или командой CREATE TYPE в БД создается тип, после чего на этот тип можно ссылаться при создании экземпляра коллекции.

Синтаксис объявления типа для ассоциативного массива:

TYPE *имя_типа* IS
TABLE OF *тип_элемента* [NOT NULL]
INDEX BY
{
 BINARY_INTEGER | подтипы BINARY_INTEGER[9-] | PLS_INTEGER[9-]
| VARCHAR2 (*максимум_числа_элементов*)[9-] | CHAR (*число_элементов*)[9-]
};
[9-] Начиная с версии 9.

Синтаксис объявления типа для вложенной таблицы:

[CREATE [OR REPLACE]] TYPE *имя_типа* IS TABLE OF *тип_элемента* [NOT NULL];

Синтаксис объявления типа для массива VARRAY:

[CREATE [OR REPLACE]] TYPE *имя_типа* IS
{VARRAY | VARYING ARRAY} (*максимум_числа_элементов*)
OF *тип_элемента* [NOT NULL];

Слово CREATE в двух последних случаях соответствует аналогичному для программы предложению SQL DDL, создающему тип как хранимый в БД объект. Фраза OR REPLACE, при предварительном наличии объекта, пересоздаст его заново с сохранением имевшихся свойств и полномочий доступа.

Для версии 8 *тип_элемента* может быть скалярным, объектным или ссылкой (REF) на объект. Если *тип_элемента* объектный, в число его атрибутов не должны входить, в свою очередь, коллекции. Явно запрещено создание коллекций с элементами типов BOOLEAN, NCHAR, NCLOB, NVARCHAR2, REF CURSOR, NESTED TABLE, VARRAY.

В версии 9.2 большинство ограничений на *тип_элемента* снято. Например, возможно создание коллекции из коллекций:

```
DECLARE
TYPE typ_some_record IS RECORD ( x NUMBER, y VARCHAR2 ( 10 ) );
TYPE typ_collection  IS TABLE OF typ_some_record;

v_acollection typ_collection;

TYPE typ_coll_of_collections
IS
TABLE OF typ_collection INDEX BY PLS_INTEGER
;

v_anothercollection typ_coll_of_collections;
BEGIN NULL; END;
/
```

В PL/SQL или в БД в представлении данных (но не в основной таблице) стало позволительно употреблять элементы типов LOB, например:

```
DECLARE TYPE typ_coll IS VARRAY ( 10 ) OF CLOB; v_collx typ_coll;
BEGIN NULL; END;
/
```

Однако коллекции из ссылок на курсор по-прежнему запрещены.

С версии 9 для технических целей в PL/SQL имеется к употреблению «предопределенный» тип вложенной таблицы (пакет DBMS_STANDARD) с описанием:

```
TYPE ora_name_list_t IS TABLE OF VARCHAR2  ( 64 );
```

Той же цели служат несколько определений типов ассоциативных массивов, наличные в пакете DBMS_UTILITY:

```
TYPE index_table_type      IS TABLE OF BINARY_INTEGER INDEX BY BINARY_INTEGER;
TYPE lname_array           IS TABLE OF VARCHAR2 ( 4000 ) INDEX BY BINARY_INTEGER;
TYPE name_array IS TABLE OF VARCHAR2 ( 30 )    INDEX BY BINARY_INTEGER;
TYPE number_array          IS TABLE OF NUMBER             INDEX BY BINARY_INTEGER;
```

Ими можно пользоваться в обычных программах, например:

```
DECLARE
 employees      ora_name_list_t;                  -- вложж. таблица
 commissioners DBMS_UTILITY.index_table_type;   -- ассоц. массив
 salaries       DBMS_UTILITY.number_array;       -- ассоц. массив
BEGIN NULL; END;
/
```

С версии 10 разрешено изменять точность элементов коллекции, состоящей из скаляров (в общем случае), и изменять предельный размер массива VARRAY.

Наличие NOT NULL запрещает существование в коллекции элементов со значением NULL. Несмотря на это коллекция как целое может сама иметь значение NULL (например, когда ей не сделано начального присвоения значения).

15.2. Работа с ассоциативными массивами

Ассоциативные массивы готовы к использованию после их объявления. Обращение к элементам массива производится напрямую. Пример для версий 7+:

```
DECLARE
  TYPE      typ_dept is TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  dept_list typ_dept;
BEGIN
dept_list (   1 ) := 1.5;
dept_list ( -15 ) := 20.1;
dept_list ( -15 ) := 3.14;      -- переопределили ...
DBMS_OUTPUT.PUT_LINE ( 'Dept indexed by -15 is ' || dept_list ( -15 ) );
DBMS_OUTPUT.PUT_LINE ( 'Dept indexed by 1 is '   || dept_list (   1 ) );
DBMS_OUTPUT.PUT_LINE ( 'Dept indexed by 200 is ' || dept_list ( 200 ) );
                                                        -- Ошибка ?..
END;
/
```

Следующие примеры для ассоциативных массивов работают только начиная с версии 9.2.

Коллекции в программе могут играть роль своего рода «программного индекса». Пример:

```
DECLARE
TYPE     typ_sal IS TABLE OF NUMBER INDEX BY VARCHAR2 ( 14 );
v_salary typ_sal;
BEGIN
v_salary ( 'QUEEN' ) := 5000;
DBMS_OUTPUT.PUT_LINE ( v_salary ( 'QUEEN' ) );
END;
/
```

Более полный пример:

```
DECLARE
TYPE typ_emp IS TABLE OF emp%ROWTYPE INDEX BY emp.ename%TYPE;
emps typ_emp;

BEGIN
FOR e IN ( SELECT * FROM emp ) LOOP emps ( e.ename ) := e; END LOOP;

DBMS_OUTPUT.PUT_LINE ( 'Salary of ALLEN is ' || emps ( 'ALLEN' ).sal );
emps ( 'ALLEN' ).sal := 16000;
DBMS_OUTPUT.PUT_LINE ( 'Salary of ALLEN is ' || emps ( 'ALLEN' ).sal );
emps ( 'KNIGHT' ).sal := 2600;                          -- OK
DBMS_OUTPUT.PUT_LINE ( 'Salary of QUEEN is ' || emps ( 'QUEEN' ).sal );
                                                        -- Ошибка !
EXCEPTION
WHEN no_data_found
   THEN
      DBMS_OUTPUT.PUT_LINE ( 'QUEEN access-> ' || SQLERRM );
END;
/
```

Другие популярные исключения при работе со *всеми* видами коллекций:

```
COLLECTION_IS_NULL
NO_DATA_FOUND
SUBSCRIPT_BEYOND_COUNT
SUBSCRIPT_OUTSIDE_LIMIT
VALUE_ERROR
```

15.3. Создание вложенной таблицы и массива VARRAY в программе

В отличие от ассоциативного массива, для переменной типа вложенной таблицы или массива VARRAY простого объявления в программе недостаточно: у такой переменной не будет значения (будет IS NULL), и попытка обратиться к ней приведет к исключительной ситуации:

```
DECLARE
  nested_tab ora_name_list_t;
BEGIN
  IF nested_tab IS NULL THEN dbms_output.put_line ( 'IS NULL' ); END IF;
  nested_tab ( 1 ) := 'a';            -- Ошибка ORA-06531 !
END;
/
```

Для возможности работы с такими переменными им нужно дать первоначальное значение («проинициализировать коллекцию», «завести» ее). Получить первоначальное значение вложенная таблица и массив VARRAY могут:

- (явно) при помощи специального конструктора,
- (неявно) путем подчитывания данных из базы,
- (неявно) как результат присвоения значения однотипной коллекции с имеющимся значением.

Конструктор, в объектной технологии, — это системная функция, имя которой автоматически совпадает с именем типа коллекции. Конструктору в качестве аргументов списком передаются элементы, из которых он составляет коллекцию.

Пример создания вложенных таблиц моделей цветов и придания им начальных значений с помощью конструктора:

```
DECLARE
TYPE          colourset_typ IS TABLE OF VARCHAR2 ( 30 );
basic_colours colourset_typ := colourset_typ ( 'RED', 'GREEN', 'BLUE' );
my_colours    colourset_typ;

BEGIN
 my_colours := colourset_typ ( 'CYAN', 'YELLOW', 'MAGENTA', 'BLACK' );
 my_colours ( 1 ) := basic_colours ( 3 );
 -- и так далее ...
END;
/
```

Пример неявного (без привлечения конструктора) получения вложенной таблицей начального значения в результате чтения БД:

```
CREATE TYPE colourset_typ IS TABLE OF VARCHAR2 ( 32 );
/

CREATE TABLE colour_models (
  model_type VARCHAR2 ( 12 )
, colours    colourset_typ
)
NESTED TABLE colours STORE AS colour_model_colours_tab;

INSERT INTO colour_models
VALUES  (
```

```
  'RGB'
, colourset_typ ( 'RED', 'GREEN', 'BLUE' )
);

INSERT INTO colour_models
VALUES (
  'CMY'
, colourset_typ ( 'CYAN', 'MAGENTA', 'YELLOW' )
);

INSERT INTO colour_models
VALUES (
  'CYMK'
, colourset_typ ( 'CYAN', 'YELLOW', 'MAGENTA', 'BLACK' )
);

INSERT INTO colour_models
VALUES (
  'HSV'
, colourset_typ ( 'HUE', 'SATURATION', 'VALUE' )
);

COMMIT;

-- Теперь создана вспомогательная для colour_models таблица
-- в том же табличном пространстве.
-- Вложенная таблица может получить первоначальное значение в программе на PL/SQL:

DECLARE
basic_colours colourset_typ;

BEGIN
   SELECT colours INTO basic_colours
   FROM   colour_models
   WHERE  model_type = 'RGB';

   DBMS_OUTPUT.PUT_LINE ( basic_colours ( 3 ) );
   -- и так далее ...
END;
/
```

Пример неявного (без привлечения конструктора) получения вложенной таблицей начального значения в результате присваивания:

```
DECLARE
basic_colours colourset_typ := colourset_typ ( 'RED', 'GREEN', 'BLUE' );
my_colours    colourset_typ;

BEGIN
   my_colours       := basic_colours;
   my_colours ( 2 ) := 'MUSTARD';
   -- и так далее ...
END;
/
```

Переменную типа коллекции указанных двух видов можно вновь лишить значения:

```
DECLARE
  nested_tab ora_name_list_t := ora_name_list_t ( 'Christ', 'Buddha' );
BEGIN
  nested_tab       := NULL;
  nested_tab ( 1 ) := ( 'Bush' );           -- Ошибка !
EXCEPTION
  WHEN collection_is_null THEN
```

```
      dbms_output.put_line ( 'Oracle said -> ' || SQLERRM );
      DBMS_SESSION.FREE_UNUSED_USER_MEMORY; -- Желательно освободить память
END;
/
```

15.4. Добавление и убирание элементов в коллекциях

В ассоциативный массив элементы добавляются простым указанием в присваивании нового индекса.

Для того, чтобы добавить в программе элемент во вложенную таблицу или в массив VARRAY, нужно сначала нарастить коллекцию с помощью процедуры EXTEND, а затем присвоить новому элементу значение.

Удалить любой элемент вложенной таблицы можно с помощью процедуры DELETE. Удалить серию элементов из «хвоста» коллекции можно с помощью процедуры TRIM. Для одной и той же коллекции не рекомендуется использовать одновременно и DELETE, и TRIM.

Изменение состава вложенной таблицы и массива VARRAY в SQL не требует использования специальных методов.

15.5. Преобразования коллекций в SQL

Для преобразования коллекций из одного вида в другой, и в таблицы БД и обратно имеются специальные псевдофункций: MULTISET и TABLE, а также используются конструкторы, функция CAST и агрегирующая функция COLLECT.

TABLE

Отображает коллекцию в таблицу БД, в противоположность MULTISET (см. ниже):

```
SELECT * FROM TABLE ( colourset_typ ( 'WHITE', 'BLACK' ) );
```

Другой пример:

```
SELECT *
FROM TABLE ( SELECT colours
             FROM   colour_models
             WHERE  model_type = 'RGB'
           )
;
```

Еще пример:

```
SELECT *
FROM   colour_models c
WHERE  'RED' IN ( SELECT * FROM TABLE ( c.colours ) )
;
```

CAST

Приводит коллекцию к нужному типу, например:

```
CREATE TYPE colours_tab_vat IS VARRAY ( 100 ) OF VARCHAR2 ( 10 );
/

SELECT colours FROM colour_models;

SELECT CAST ( colours AS colours_tab_vat ) FROM colour_models;
```

Функция CAST носит более общий характер, и способна приводить значения не только к коллекциям, однако применительно к простым встроенным типам ее используют редко ввиду наличия в Oracle множества более продвинутых функций преобразования.

MULTISET и *COLLECT*

MULTISET отображает таблицу в коллекцию. Используя MULTISET и CAST, можно извлекать из БД строки таблицы и превращать их в коллекцию требуемого типа:

```
SELECT
     CAST ( MULTISET ( SELECT 'BLACK' FROM dual ) AS colourset_typ )
FROM dual
;
```

Еще пример:

```
SELECT
     CAST ( MULTISET ( SELECT loc FROM dept ) AS colourset_typ )
FROM dual
;
```

Когда речь идет о вложенных таблицах, то преобразование столбца в коллекцию можно выполнить проще и более естественно функцией COLLECT:

```
SELECT CAST ( COLLECT ( loc ) AS colourset_typ ) FROM dept;
```

COLLECT — это функция агрегирования, и разрешает использовать себя в агрегирующих (и аналитических) запросах:

```
COLUMN names FORMAT A70
SELECT   deptno
       , CAST ( COLLECT ( ename ) AS colourset_typ ) names
FROM     emp
GROUP BY deptno
;
```

Если тип коллекции определен в БД (а не в программе), то функция TABLE способна преобразовать в список строк также и коллекцию этого типа, созданную в программе:

```
SQL> VARIABLE colors VARCHAR2 ( 10 )
SQL> DECLARE colors colourset_typ := colourset_typ ( 'RED', 'BLUE', 'RED' );
  2  BEGIN
  3  SELECT COUNT ( DISTINCT COLUMN_VALUE ) INTO :colors FROM TABLE ( colors );
  4  END;
  5  /

PL/SQL procedure successfully completed.

SQL> PRINT colors

COLORS
--------------------------------
2
```

Пример показывает, как это позволяет употреблять для коллекций в программе агрегатные функции, доступные только в SQL.

15.6. Выражения со вложенными таблицами: множественные операции

До версии 10 с коллекциями можно было осуществлять только одну множественную операцию: присваивать в программе значение другой однотипной. С версии 10 круг множественных действий *для вложенных таблиц* расширился операциями из ANSI SQL. При этом:

а) множественные операции распространяются не на все типы вложенных таблиц;
б) к выражениям в PL/SQL множественные операции применимы не всегда или применимы с ограничениями на тип элементов вложенных таблиц.

В программе можно выполнять сравнение однотипных вложенных таблиц из скалярных элементов:

IF collection1 = collection2 THEN … ;

В SQL подобное сравнение можно выполнять в условных выражениях в конструкциях CASE и WHERE также для таблиц из объектов, но если в типе объекта сформулированы методы сравнения MAP или ORDER.

Можно применять множественные операции MULTISET {UNION | INTERSECT | EXCEPT}, SET и CARDINALITY, например:

collection1 := collection2 MULTISET UNION collection3;

collection1 := SET (collection2)

Семантика этих операций аналогична принятой в SQL, при том, что обозначения разные:

Операция с коллекцией ...	***... соответствует операции в SQL***
MULTISET UNION MULTISET INTERSECT MULTISET EXCEPT	UNION ALL INTERSECT ALL* MINUS ALL*
MULTISET UNION DISTINCT MULTISET INTERSECT DISTINCT MULTISET EXCEPT DISTINCT	UNION INTERSECT MINUS
SET	SELECT DISTINCT
CARDINALITY	SELECT COUNT (*)

* Как если бы такая операция существовала по аналогии с UNION ALL

Пример:

```
DECLARE
  nt1 ora_name_list_t := ora_name_list_t ( 'A', 'BC' );
  nt2 ora_name_list_t := ora_name_list_t ( 'BC', 'A' );
  nt3 ora_name_list_t := ora_name_list_t ( 'BCD', 'A' );
BEGIN
  DBMS_OUTPUT.PUT_LINE ( CARDINALITY ( nt1 ) );
  DBMS_OUTPUT.PUT_LINE ( CARDINALITY ( nt1 MULTISET UNION nt3 ) );
  DBMS_OUTPUT.PUT_LINE
                  ( CARDINALITY ( nt1 MULTISET UNION DISTINCT nt3 ) );
  DBMS_OUTPUT.PUT_LINE
                  ( CARDINALITY ( SET ( nt1 MULTISET UNION nt3 ) ) );
END;
/

SELECT
     colours MULTISET UNION colourset_typ ( 'RED', 'BLACK' )
FROM colour_models
;

SELECT
     CARDINALITY ( colours MULTISET UNION colourset_typ ( 'RED', 'BLACK' ) )
FROM colour_models
```

```
;

SELECT
     SET ( colours MULTISET UNION colourset_typ ( 'RED', 'BLACK' ) )
FROM colour_models
;

SELECT
     colours MULTISET UNION DISTINCT colourset_typ ( 'RED', 'BLACK' )
FROM colour_models
;
```

Несколько прочих операций не находят прямых аналогов в SQL:

Операция с коллекцией ...	***... содержание***
= IS [NOT] EMPTY IS [NOT] A SET	сравнение множеств проверка на пустоту проверка на отсутствие повторений элементов
POWERMULTISET	булеан (степень множества) за вычетом пустого множества
POWERMULTISET_BY_CARDINALITY	подмножества булеана с указаной кардинальностью
COLLECT	агрегирование коллекций по столбцу таблицы

Последние три операции допускаются только в выражениях в SQL.

Примеры:

```
SELECT *
FROM
 TABLE ( POWERMULTISET ( colourset_typ ( 'a', 'b' ) ) )
;

SELECT *
FROM
 TABLE ( POWERMULTISET_BY_CARDINALITY ( colourset_typ ( 'a', 'b' ), 1 ) )
;

SELECT
 COLLECT ( model_type )
FROM colour_models
;

SELECT
 CAST ( COLLECT ( model_type ) AS colourset_typ )
FROM colour_models
;

SELECT
 COLLECT ( colours )
FROM colour_models
;
```

Обратите внимание на определенную непоследовательность: прямое обращение к результату возможно для COLLECT (последний пример), но невозможно для POWERMULTISET — только через посредство CARDINALITY, TABLE или CAST. Следующий пример даст ошибку:

```
SELECT POWERMULTISET ( colours ) FROM colour_models;
```

Следующие примеры безошибочны:

```
SELECT CARDINALITY ( POWERMULTISET ( colours ) )
FROM   colour_models
```

```
;

CREATE TYPE colour_models_t AS TABLE OF colourset_typ;
/
SELECT CAST ( POWERMULTISET ( colours ) AS colour_models_t )
FROM   colour_models
;
```

Упоминавшаяся функция COLLECT в простом варианте применима к данным всех типов. Однако в версии 11.2 она получила развитие и стала допускать построение коллекции из неповторяющихся элементов (DISTINCT/UNIQUE) и упорядоченных (ORDER BY). А для этих случаев подойдут только типы элементов, допускающие сравнение значений. Сравните результаты запросов:

```
COLUMN jobs FORMAT A70 WORD
SELECT    deptno, CAST ( COLLECT ( job ) AS colourset_typ ) jobs
FROM      emp
GROUP  BY deptno
;
SELECT    deptno, CAST ( COLLECT ( DISTINCT job ) AS colourset_typ ) jobs
FROM      emp
GROUP  BY deptno
;
SELECT    deptno, CAST ( COLLECT ( job ORDER BY sal ) AS colourset_typ ) jobs
FROM      emp
GROUP  BY deptno
;
```

Примеры использования условных выражений:

```
DECLARE
  nt1 ora_name_list_t := ora_name_list_t ( 'A', 'BC' );
  nt2 ora_name_list_t := ora_name_list_t ( 'BC', 'A' );
  nt3 ora_name_list_t := ora_name_list_t ( 'BCD', 'A' );
BEGIN
  IF nt1 =  nt2 THEN DBMS_OUTPUT.PUT_LINE ( 'nt1 = nt2'  ); END IF;
  IF nt2 <> nt3 THEN DBMS_OUTPUT.PUT_LINE ( 'nt2 <> nt3' ); END IF;
  IF nt2 IS NOT EMPTY THEN DBMS_OUTPUT.PUT_LINE ( 'nt2 не пусто' ); END IF;
  IF nt1 MULTISET UNION nt2 IS NOT A SET
     THEN DBMS_OUTPUT.PUT_LINE ( 'в nt1 и nt2 есть общие элементы' ); END IF;
END;
/

SELECT model_type
FROM   colour_models
WHERE  colours MULTISET UNION colourset_typ ( 'RED' ) IS A SET
;

SELECT
 CASE colours
 WHEN colourset_typ ( 'RED', 'GREEN', 'BLUE') THEN 'RGB'
 WHEN colourset_typ ( 'CYAN', 'YELLOW', 'MAGENTA', 'BLACK' ) THEN 'CYMK'
 ELSE 'Другая модель'
 END CASE
FROM colour_models;
```

15.7. Методы для работы с коллекциями в программе

Для работы с коллекциями всех видов *в программе* имеется группа методов, обращение к которым записывается так:

имя_коллекции.имя_метода [([*параметры*])]

Методы для коллекций перечисляются ниже:

Метод коллекции	*Описание*
COUNT [()][1] Функция	Возвращает текущее число элементов в коллекции
DELETE [([i [,j]])] Процедура	Удаляет из вложенной таблицы или из ассоциативного массива элемент i или с i-го по j-й. При отсутствии параметров удаляет все элементы (единственно возможный вариант для VARRAY).
EXISTS (i) Функция	Выдает TRUE или FALSE в зависимости от существования i-го элемента. Для вложенной таблицы или массива VARRAY без значения выдает FALSE.
EXTEND [([n [,i]])] Процедура	Добавляет в коллекцию n элементов, задавая им значение элемента i. По умолчанию n = 1.
FIRST [()][1] Функция	Возвращает наименьший по значению индекс в коллекции. Для существующей коллекции, но пустой, значения не возвращает (NULL).
LAST [()][1] Функция	Возвращает наибольший по значению индекс в коллекции. Для существующей коллекции, но пустой, значения не возвращает (NULL).
LIMIT [()][1] Функция	Для конкретного массива VARRAY возвращает максимально допустимое число элементов. Для вложенных таблиц и ассоциативных массивов значения не возвращает (NULL).
PRIOR (i) Функция	Возвращает значение индекса, непосредственно предшествующее индексу i. Если i <= FIRST ( ), значения не возвращает (NULL).
NEXT (i) Функция	Возвращает значение индекса, непосредственно следующее за индексом i. Если i >= COUNT ( ), значения не возвращает (NULL).
TRIM (i) Процедура	Удаляет i последних элементов из коллекции. По умолчанию i = 1. Если параметр указан без значения (NULL), процедура ничего не делает. Неприменима к ассоциативным массивам.

[1] Список параметров всегда пустой, но PL/SQL равно допускает как указание, так и пропуск пустых круглых скобок. Тем не менее указание (хотя и пустых) круглых скобок соответствует правилам объектного подхода, а пропуск — нет.

Все функции, кроме EXIST, возвращают BINARY_INTEGER. Все параметры методов также типа BINARY_INTEGER. Оговорка: для ассоциативного массива, «индексированного» (хешированного) строкой, FIRST, LAST, PRIOR и NEXT возвращают строки, и в параметрах принимают строки.

Метод EXIST, примененный к коллекции в состоянии NULL, не вызовет исключительной ситуации COLLECTION_IS_NULL.

15.8. Примеры использования коллекций в программе

15.8.1. Пример обращения со встроенными коллекциями

Подобно традиционным объектным типам, с версии 9 коллекции находят себе употребление в разного рода встроенных в БД структурах. В случае вложенных таблиц и массивов VARRAY им, как правило, при этом начальных присвоений программисту делать не надо ввиду того, что значения им передаются через механизм параметров режима OUT. При этом используются подобные коллекционные значения как правило исключительно для чтения данных из них, но не изменения.

В частности, задачам статистической обработки служит встроенный пакет DBMS_STAT_FUNCS. В его составе определен тип записи SUMMARYTYPE для передачи результатов вычислений процедурами пакета. Некоторые поля записей этого типа скалярные, а некоторые задаются коллекционными типами, определенными в составе пакета:

```
TYPE n_arr     IS VARRAY ( 5 ) OF NUMBER;
TYPE num_table IS TABLE OF NUMBER;
```

Эти типы используются в определении типа записи в составе пакета:

```
TYPE summarytype IS RECORD (
  count            NUMBER
```

```
, min              NUMBER
, max              NUMBER
  ...                                -- другие поля записи здесь опущены из экономии
, cmode            num_table
  ...                                -- другие поля записи здесь опущены из экономии
, extreme_values   num_table
, top_5_values     n_arr
, bottom_5_values  n_arr
);
```

Вот как можно показать использование этих типов в программе сиатистического анализа зарплат сотрудников, выполненного при помощи процедуры SUMMARY из состава пакета:

```
DECLARE
  sigma NUMBER := 3;
  s     DBMS_STAT_FUNCS.SUMMARYTYPE;
  item  NUMBER;
  cnt   NUMBER;

BEGIN
DBMS_STAT_FUNCS.SUMMARY ( 'SCOTT', 'EMP', 'SAL', sigma, s );

DBMS_OUTPUT.PUT_LINE ( 'COUNT = ' || s.count );
DBMS_OUTPUT.PUT_LINE ( 'MIN = '   || s.min );
DBMS_OUTPUT.PUT_LINE ( 'MAX = '   || s.max );
-- ... прочие скаляры из экономии опускаются

FOR item IN s.cmode.FIRST .. s.cmode.LAST
LOOP
  DBMS_OUTPUT.PUT_LINE (
                  'MODE [' || item || '] = ' || s.cmode ( item ) );
END LOOP;

FOR item IN s.top_5_values.FIRST .. s.top_5_values.LAST
LOOP
  DBMS_OUTPUT.PUT_LINE (
         'TOP ' || item || ' VALUE = ' || s.top_5_values ( item ) );
END LOOP;

cnt := s.bottom_5_values.LAST;

FOR item IN s.bottom_5_values.FIRST .. s.bottom_5_values.LAST
LOOP
  DBMS_OUTPUT.PUT_LINE (
     'BOTTOM ' || cnt || ' VALUE = ' || s.bottom_5_values ( item ) );
cnt := cnt - 1;
END LOOP;
END;
/
```

Еще один пример обращения к коллекциям — атрибутам курсора BULK_EXCEPTIONS и BULK_ROWCOUNT последует ниже.

15.8.2. Пример работы с собственной коллекцией

Пример показывает внесение программой изменений в коллекцию и разъясняет некоторые тонкости ее устройства:

```
DECLARE
my_list colourset_typ := colourset_typ ( 'RED', 'GREEN', NULL, NULL );
out     VARCHAR2 ( 20 );
```

```
BEGIN
   DBMS_OUTPUT.PUT_LINE
        ( 'my_list has ' || my_list.COUNT ( ) || ' elements' );
   my_list.DELETE ( 2, 3 );
   DBMS_OUTPUT.PUT_LINE
        ( 'my_list has ' || my_list.COUNT ( ) || ' elements' );
   FOR element# IN my_list.FIRST .. my_list.LAST
   LOOP
      IF my_list.EXISTS ( element# )
      THEN
         out := NVL ( my_list ( element# ), '[NULL]' );
      ELSE
         out := 'deleted';
      END IF;
   DBMS_OUTPUT.PUT_LINE (
              'Element ' || element# || ' is ' || out || '. '
           || ' Prior= ' || NVL ( my_list.PRIOR ( element# ), -1 )
           || ' Next= '  || NVL ( my_list.NEXT  ( element# ), -1 )
   );
   END LOOP;
END;
/
```

15.8.3. Пример употребления множественных операций

Работает с версии 10:

```
CREATE OR REPLACE TYPE ntt IS TABLE OF NUMBER ( 4 )
/

DECLARE
 nt1 ntt;
 nt2 ntt;
 nt3 ntt;

BEGIN
 WITH
 emps AS ( SELECT empno FROM emp WHERE comm IS NOT NULL )
 SELECT CAST ( MULTISET ( SELECT * FROM emps ) AS ntt )
 INTO   nt1
 FROM   dual;

 WITH
 emps AS ( SELECT empno FROM emp WHERE job = 'SALESMAN' )
 SELECT CAST ( MULTISET ( SELECT * FROM emps ) AS ntt )
 INTO   nt2
 FROM   dual;

 IF nt1 = nt2
 THEN dbms_output.put_line ( 'All salesmen have all commissions' );
 END IF;

 WITH
 emps AS ( SELECT empno FROM emp WHERE job = 'ANALYST' )
 SELECT CAST ( MULTISET ( SELECT * FROM emps ) AS ntt )
 INTO   nt3
 FROM   dual;

 IF nt1 IN ( nt2, nt3 )
 THEN dbms_output.put_line
      ( 'Either all salesmen or all analysts have all commissions' );
 END IF;
END;
```

```
/
```

Другой пример подчитывания данных в коллекцию выглядит более громоздко, однако не требует создания типа в БД (действует, начиная с версии 8):

```
DECLARE
TYPE nt_type IS TABLE OF emp.empno%TYPE;
nt   nt_type;

BEGIN
 nt := nt_type ( );
 FOR e IN ( SELECT empno FROM emp WHERE job = 'SALESMAN' )
 LOOP
   nt.EXTEND ( );
   nt ( nt.LAST ) := e.empno;
 END LOOP;

 -- ... ... ...
END;
/
```

15.9. Привилегии

На использование типов коллекций, созданных в другой схеме БД, требуется иметь на них привилегию EXECUTE.

15.10. Серийное выполнение запросов и связывание значений с массивами

Возможность связывать элементы SQL-запроса не со скалярными переменными, а с массивами в форме коллекций, появилась в версии 8.1. Использование связывания массивами (= коллекциями) позволяет существенно ускорить обмен данными между программой и БД. Этому же способствует возможность выполнить одной командой серию однотипных операций по изменению данных.

Связывание массивами и серийное выполнение однотипных операций основывается на двух главных конструкциях: BULK COLLECT INTO и FORALL. Для них также создано несколько специальных атрибутов неявного курсора.

15.10.1. Серийное выполнение однотипных операций: конструкция FORALL

Синтаксис FORALL:

FORALL *переменная_цикла* IN *нижняя_граница .. верхняя_граница*
предложение_SQL;

переменная_цикла может использоваться для ссылки на элемент коллекции, упомянутой в *предложении_SQL*. В случае FORALL СУБД будет рассматривать серию операций как один запрос, а не последовательность самостоятельных *предложений_SQL*.

Пример:

FORALL i IN deptlist.FIRST .. deptlist.LAST
DELETE FROM emp WHERE deptno = deptlist (i);

По своему действию эта запись равносильна следующей:

FOR i IN deptlist.FIRST .. deptlist.LAST
LOOP

```
        DELETE FROM emp WHERE deptno = deptlist ( i );
END LOOP;
```

Однако первая запись короче второй и эффективнее выполняется.

Компилятор PL/SQL, имея дело с циклом FOR ... LOOP, способен по своей инициативе строить код так, чтобы обращения к БД в запросе посылались из программы не по одному, а группами по 100. (Этого не произойдет, например, если программа компилируется для отладки.) В таких случаях выигрыш FORALL в скорости перед FOR ... LOOP может отсутствовать, если только полное число итераций не исчисляется сотнями.

Прерывание выполнения может случиться по причине (а) отсутствия очередного элемента в коллекции и (б) ошибки при исполнении очередного оператора DML. В обоих случаях последствия прерывания будут те же, что при употреблении обычного цикла LOOP; тем самым предложение FORALL не обладает целостностью исполнения простых операторов DML.

Упражнение. Проверить последствия прерывания предложения FORALL по причине отсутствия индекса в коллекции и из-за ошибки в очередной операции DML. Как обрыв выполнения отражается на данных в БД ? Что можно предложить воизбежание рассогласования данных ?

Общую схему обработки прерывания цикличного выполнения можно построить так:

```
BEGIN
SAVEPOINT start_changes;

FORALL i IN deptlist.FIRST .. deptlist.LAST
        DELETE FROM emp WHERE deptno = deptlist ( i );

EXCEPTION
WHEN OTHERS -- а лучше избирательно указать конкретную ошибку
        THEN ROLLBACK TO SAVEPOINT start_changes;
END;
```

Чтобы исполнение операторов не прервалось досрочно по причине отсутствия элемента, в версиях 8 и 9 коллекция в FORALL должна быть плотной. С версии 10 плотности индексов не требуется, если употребить особый синтаксис указания пробежки по элементам коллекции (приводится далее).

С версии 9.2 можно потребовать не прерывать цикл выполнения при возникновении ошибки DML. Пример синтаксического оформления так же далее.

15.10.1.1. Работа с неплотными коллекциями и с выборочными элементами

С версии 10 предложение FORALL может без прерываний работать с неплотными коллекциями.

Пример выполнения серийной операции только с имеющимися в коллекции элементами:

```
FORALL i IN INDICES OF deptlist
        DELETE FROM emp WHERE deptno = deptlist ( i );
```

С другой стороны, если требуется выполнить серийную операцию только для отдельных элементов коллекции, а не для всех, можно занести нужные значения индексов в отдельную коллекцию, например, SELECTED_DEPTS, и выдать примерно следующее:

```
FORALL i IN VALUES OF selected_depts
        DELETE FROM emp WHERE deptno = deptlist ( i );
```

15.10.1.2. Обработка ошибок запросов и атрибут %BULK_EXCEPTIONS

С версии 9.2 серию операций, выполняемых с помощью FORALL, можно не прерывать из-за ошибок в отдельных операторах DML, а вместо этого копить номера ошибок в специальном атрибуте %BULK_EXCEPTIONS и получить специальную ошибку только по общему завершению цикла исполнения. Достигается это использованием фразы SAVE EXCEPTIONS:

FORALL i IN deptlist.FIRST .. deptlist.LAST **SAVE EXCEPTIONS**
DELETE FROM emp WHERE deptno = deptlist (i);

Атрибут %BULK_EXCEPTIONS имеет тип коллекции из записей с полями ERROR_INDEX и ERROR_CODE.

Пример использования:

```
SAVEPOINT all_departments_exist;

DECLARE
TYPE dlist_t IS TABLE OF dept.deptno%TYPE;
deptlist dlist_t := dlist_t ( 30, 40, 20 );
dml_errors EXCEPTION;
    PRAGMA EXCEPTION_INIT ( dml_errors, -24381 );

BEGIN

   FORALL i IN deptlist.FIRST .. deptlist.LAST SAVE EXCEPTIONS
      DELETE FROM dept WHERE deptno = deptlist ( i );

   DBMS_OUTPUT.PUT_LINE ( 'You will not see this message' );

EXCEPTION
   WHEN dml_errors THEN
      FOR i IN 1 .. SQL%BULK_EXCEPTIONS.COUNT LOOP
         DBMS_OUTPUT.PUT_LINE (
            'Iteration: '
         || SQL%BULK_EXCEPTIONS ( i ).ERROR_INDEX
         || '; Oracle error: '
         || SQL%BULK_EXCEPTIONS ( i ).ERROR_CODE
         );
      END LOOP;
END;
/

SELECT deptno "Departments that left:" FROM dept;

ROLLBACK TO SAVEPOINT all_departments_exist;
```

15.10.1.3. Атрибут %BULK_ROWCOUNT

При серийном выполнении операций можно пользоваться атрибутом неявного курсора %BULK_ROWCOUNT. Он представляет собой коллекцию, индексированную целыми, элементы которой хранят количества строк, затронутых отдельными операциями. *N*-й элемент SQL%BULK_ROWCOUNT содержит число строк, затронутых при *n*-ом исполнении SQL-запроса:

```
…
FORALL d# IN deptlist.FIRST .. deptlist.LAST
  DELETE FROM emp WHERE deptno = deptlist (d# );

FOR d# IN deptlist.FIRST .. deptlist.LAST
LOOP
  DBMS_OUTPUT.PUT_LINE ( ‘dept ‘ || deptlist ( d# ) || ’ lost ’ || SQL%BULK_ROWCOUNT ( d# ) );
```

END LOOP;

DBMS_OUTPUT.PUT_LINE (‘overall deleted ’ || **SQL%ROWCOUNT**);
…

Атрибут SQL%ROWCOUNT содержит сумму элементов массива SQL%BULK_ROWCOUNT. Атрибуты %FOUND и %NOTFOUND отражают результат последнего выполненного оператора SQL.

Атрибут SQL%BULK_ROWCOUNT нельзя передавать в качестве параметра или присваивать целиком другой коллекции.

15.10.2. Привязка значений в запросе к массиву: конструкция BULK COLLECT INTO

Несколько операторов PL/SQL имеют в своем составе конструкцию INTO для размещения результата в переменных программы. Заменой слова INTO на BULK COLLECT INTO программист получает возможность размещать результат не в скалярные переменные программы, а в коллекции. Тогда жесткое требование однострочности операции с данными БД теряет силу:

SELECT … INTO … → SELECT … **BULK COLLECT** INTO …
FETCH … INTO … → FETCH … **BULK COLLECT** INTO …
RETURNING … INTO … → RETURNING … **BULK COLLECT** INTO …

Синтаксис BULK COLLECT INTO:

BULK COLLECT INTO {*список_коллекций_из_скаляров* | *коллекция_из_записей*};

список_коллекций_из_скаляров — перечисление через запятую имен коллекций, по каждой на столбец, представленный в конструкции SELECT. *коллекция_из_записей* может использоваться начиная с версии 9.2.

Примеры употребления:

```
DECLARE
  TYPE emp_name__type IS TABLE OF emp.ename%TYPE;
  TYPE emp_sal__type  IS TABLE OF emp.sal%TYPE;
  ename_list emp_name__type;
  esal_list  emp_sal__type;
BEGIN
  SELECT ename, sal
  BULK COLLECT INTO ename_list, esal_list
  FROM   emp
  WHERE  deptno = 30;
  -- обрабатываем списки
END;
/
```

(Сравните с SELECT … INTO … для обычной скалярной программной переменной).

Более простой в офомлении вариант примера:

```
DECLARE
  TYPE emp__record_type IS RECORD
     ( ename emp.ename%TYPE, sal emp.sal%TYPE );
  TYPE emp_list__type IS TABLE OF emp__record_type;
  emp_list emp_list__type;
BEGIN
  SELECT ename, sal
  BULK COLLECT INTO emp_list
  FROM   emp
  WHERE  deptno = 30;
  -- обрабатываем список
```

```
END;
/
```

В следующем примере будут удалены сотрудники, перечисленные во входном списке, а выражение RETURNING вернет список удаленных:

```
DECLARE
   TYPE dlist__type   IS TABLE OF dept.deptno%TYPE;
   TYPE enolist__type IS TABLE OF emp.empno%TYPE;
   deptlist dlist__type := dlist__type ( 10, 30 );

   FUNCTION fire_in_depts ( deptlist dlist__type )
   RETURN enolist__type
   IS
      enolist enolist__type;
   BEGIN
      FORALL dept# IN deptlist.FIRST .. deptlist.LAST
         DELETE FROM emp WHERE deptno = deptlist ( dept# )
      RETURNING empno BULK COLLECT INTO enolist;
   RETURN enolist;
   END;

BEGIN
SAVEPOINT altogether;
DBMS_OUTPUT.PUT_LINE
     ( fire_in_depts ( deptlist ).COUNT || ' deleted' );
ROLLBACK TO SAVEPOINT altogether;
END;
/
```

15.10.2.1. Ограничение количества одновременно подчитываемых строк

Привязка к массиву в программе возможна и при работе с явным курсором, что дает возможность одной операцией FETCH извлечь сразу группу строк результата. Следующий пример показывает, как с помощью ключевого слова LIMIT можно ограничить порции подчитываемых в программу данных в случае неизвестного заранее размера результата (и предупредить переполнение памяти):

```
…
OPEN curs;
LOOP
        FETCH curs BULK COLLECT INTO list1, list2 LIMIT 1000;
        EXIT WHEN curs%NOTFOUND;
        FOR i IN 1 .. list1.COUNT
                LOOP
                -- обрабатываем list1(i) и list2(i)
                END LOOP;
END LOOP;
CLOSE curs;
…
```

15.10.2.2. BULK COLLECT INTO во встроенном динамическим SQL

С версии 9.2 конструкцию BULK COLLECT INTO можно использовать и во встроенном динамическом SQL:

```
DECLARE
  TYPE num_tab IS TABLE OF emp.ename%TYPE;
  saltab num_tab;
BEGIN
  EXECUTE IMMEDIATE
     'UPDATE emp SET sal = sal + 10 WHERE deptno = 10 '
```

```
  || 'RETURNING ename INTO :1'
  RETURNING BULK COLLECT INTO saltab
  ;

  FOR i IN saltab.FIRST .. saltab.LAST LOOP
    DBMS_OUTPUT.PUT_LINE ( saltab ( i ) );
  END LOOP;
END;
/

ROLLBACK;
```

15.10.3. Полный пример для схемы SCOTT

Пример связывания массивами и серийного выполнения для демонстрационной схемы БД SCOTT:

```
SAVEPOINT old_salaries;

DECLARE
TYPE up_ttype     IS TABLE OF NUMBER         INDEX BY BINARY_INTEGER;
TYPE newsal_ttype IS TABLE OF emp.sal%TYPE   INDEX BY BINARY_INTEGER;
TYPE names_ttype  IS TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;

toupgrade     up_ttype;
newsal        newsal_ttype;
upgraded_name names_ttype;

BEGIN
   toupgrade ( 1 ) := 10;
   toupgrade ( 2 ) := 20;

   FORALL i IN toupgrade.FIRST .. toupgrade.LAST
      UPDATE emp
      SET    sal = sal * 10
      WHERE  deptno = toupgrade ( i )
   RETURNING ename, sal BULK COLLECT INTO upgraded_name, newsal;

   FOR i IN upgraded_name.FIRST .. upgraded_name.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (
        RPAD ( upgraded_name ( i ), 10 ) || newsal ( i )
      );
   END LOOP;
EXCEPTION
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE ( q'[Can't upgrade -> ]' || SQLERRM );
END;
/

/

ROLLBACK TO SAVEPOINT old_salaries;
```

Упражнение. Конструкция FORALL не терпит пропусков элементов в коллекции (см. замечание выше). Предложите для примера выше более надежный вариант коллекции для списка отделов. Добавьте в код выше выдачу числа сотрудников с увеличенной зарплатой в каждом отделе и общего числа таких сотрудников.

15.11. Использование коллекций в табличных функциях (потоковой реализации)

Табличными в PL/SQL называются функции, принимающие в качестве аргумента коллекции и/или возвращающие коллекции. Их определение и способ обращения к ним из SQL-предложения имеют свои особенности. Так, для таких функций можно использовать потоковую выдачу результата по мере его выработки телом функции. Это позволяет

- сократить требования к памяти, особенно при больших таблицах и
- ускорить получение результата, особенно при наличии параллельной платформы.

15.11.1. Простой пример

Пример:

```
CREATE TYPE ename_type IS TABLE OF VARCHAR2 ( 256 )
/

CREATE OR REPLACE FUNCTION emps_in_dept ( deptname IN VARCHAR2 )
RETURN ename_type
PIPELINED
IS
BEGIN
  FOR employee IN
    ( SELECT emp.ename
      FROM   emp INNER JOIN dept
             USING ( deptno )
      WHERE  dept.dname = emps_in_dept.deptname
    )
  LOOP PIPE ROW ( employee.ename );
  END LOOP;
  RETURN;
END;
/
```

Обращение к такой функции из SQL:

```
SELECT emps_in_dept ( 'SALES' ) FROM dual;

SELECT * FROM TABLE ( emps_in_dept ( 'SALES' ) );
```

(Второй SELECT более реалистичен).

Упражнение. Выдайте `SELECT * FROM TABLE ( emps_in_dept ( 'sales' ) )`.

Упражнение. Создайте табличную функцию PIVOT, выдающую указанное ее параметром число строк с последовательными целыми значениями («опорная таблица», полезная при составлении отчетов).

Дополнительную эффективность табличной функции может придать дополнительное указание в заголовке PARALLEL_ENABLE, позволяющее ускорить ее вычисление при наличии параллельности на платформе.

15.11.2. Использование для преобразования данных

Более сложный пример поточного преобразования данных, характерный, в частности, для приложений типа Data Warehouse. Он потребует создания специального служебного пакета, поскольку для удобства использования мы хотим употребить ссылку на курсор (строгую !), а также воспользоваться вложенной таблицей из записей; в SQL такие типы отсутствуют:

```
CREATE OR REPLACE PACKAGE definitions IS
   TYPE emp_payment_type IS RECORD (
       ename   emp.ename%TYPE
     , payment VARCHAR2 ( 10 )
     , amount  NUMBER
     );
   TYPE payments_type  IS TABLE OF emp_payment_type;
   TYPE refcursor_type IS REF CURSOR RETURN emp%ROWTYPE;
END definitions;
/

CREATE OR REPLACE FUNCTION payments (
  emps IN definitions.refcursor_type
)
RETURN definitions.payments_type
PIPELINED
IS
   in_rec   emp%ROWTYPE;
   out_rec  definitions.emp_payment_type;
BEGIN
   LOOP
      FETCH emps INTO in_rec; EXIT WHEN emps%NOTFOUND;
      out_rec := NULL;

      IF in_rec.sal IS NOT NULL
      THEN
         out_rec.ename   := in_rec.ename;
         out_rec.payment := 'SALARY';
         out_rec.amount  := in_rec.sal;
         PIPE ROW ( out_rec );
      END IF;

      IF in_rec.comm IS NOT NULL
      THEN
         out_rec.ename   := in_rec.ename;
         out_rec.payment := 'COMMISSION';
         out_rec.amount  := in_rec.comm;
         PIPE ROW ( out_rec );
      END IF;
   END LOOP;
   RETURN;
END;
/
```

Пример обращения к функции PAYMENTS:

```
SELECT * FROM TABLE ( payments ( CURSOR ( SELECT * FROM emp ) ) );
```

Замечание по оформлению. Коль скоро уж для создания подобной преобразующей функции нам пришлось пойти на заведение пакета воимя определений типов, было бы естественно в жизни включить в состав этого пакета также и определение самой функции. Выше этого не было сделано для демонстрации общности и для некоторого приуменьшения загроможденности кода.

Ту же задачу преобразования формата Oracle позволяет решить технически чуть иначе, прибегнув к типу объекта и к типу ссылки на курсор SYS_REFCURSOR (последнее можно было сделать и в примере выше). Вот один из возможных вариантов:

```
DROP FUNCTION payments;
DROP TYPE payments_type;
DROP TYPE payment_type;

CREATE TYPE payment_type AS OBJECT
(
```

```
  ename       VARCHAR2 ( 9 )
, payment     VARCHAR2 ( 10 )
, amount      NUMBER
)
/

CREATE TYPE payments_type IS TABLE OF payment_type
/

CREATE OR REPLACE FUNCTION payments ( emps IN sys_refcursor )
RETURN payments_type
PIPELINED
IS
   in_rec  emp%ROWTYPE;
   out_rec payment_type;
BEGIN
   LOOP
      FETCH emps INTO in_rec; EXIT WHEN emps%NOTFOUND;
      out_rec := payment_type ( NULL, NULL, NULL );

      IF in_rec.sal IS NOT NULL
      THEN
         out_rec.ename   := in_rec.ename;
         out_rec.payment := 'SALARY';
         out_rec.amount  := in_rec.sal;
         PIPE ROW ( out_rec );
      END IF;

      IF in_rec.comm IS NOT NULL
      THEN
         out_rec.ename   := in_rec.ename;
         out_rec.payment := 'COMMISSION';
         out_rec.amount  := in_rec.comm;
         PIPE ROW ( out_rec );
      END IF;
   END LOOP;
   RETURN;
END;
/
```

Снова проверка:

```
SQL> SELECT * FROM TABLE ( payments ( CURSOR ( SELECT * FROM emp ) ) );

ENAME      PAYMENT         AMOUNT
---------- ---------- ----------
SMITH      SALARY             800
ALLEN      SALARY            1600
ALLEN      COMMISION          300
WARD       SALARY            1250
WARD       COMMISION          500
JONES      SALARY            2975
MARTIN     SALARY            1250
MARTIN     COMMISION         1400
BLAKE      SALARY            2850
CLARK      SALARY            2450
SCOTT      SALARY            3000
KING       SALARY            5000
TURNER     SALARY            1500
TURNER     COMMISION            0
ADAMS      SALARY            1100
JAMES      SALARY             950
FORD       SALARY            3000
MILLER     SALARY            1300

18 rows selected.
```

Обратите внимание, что использование типа объекта не дает возможности сослаться на типы столбцов, а отказ от вспомогательного пакета не позволяет использовать строгую ссылку на курсор.

Упражнение. Переписать этот пример, перенеся определение ссылки на курсор в пакет и сделав ссылку строгой. В пакет же можно перенести и описание типа для вложенной таблицы.

16. Настройка кода PL/SQL

Oracle дает ряд советов по организации кода на PL/SQL для повышения эффективности его исполнения или же краткости и ясности.

16.1. Использование типов данных

Преобразования типов

В выражениях PL/SQL, как и в SQL, действует правило неявного преобразования типов данных, и даже более усугубленное. В целом считается, что неявного преобразования типов нужно избегать ввиду следующего:

- их наличие способно приводить к неинтуитивным результатам;
- такие преобразования программист не контролирует;
- они замедляют вычисления.

Их следует заменять на явные преобразования (TO_NUMBER, TO_CHAR, CAST) или вовсе избегать. Пример последнего:

```
DECLARE n NUMBER;
BEGIN
  n := n + 15;    -- 15 принимается как значение PLS_INTEGER, и здесь преобразуется
  n := n + 15.0; -- 15 принимается как значение NUMBER, и здесь не преобразуется
END;
/
```

Аналогичные неявные преобразования типов характерны при передаче значений параметрам.

Объявления с NOT NULL

Переменные, объявленные как NOT NULL, требуют в выражениях дополнительного времени на обработку. Там, где это существенно, можно попробовать не прибегать к NOT NULL , а обойтись явными проверками:

```
PROCEDURE calc IS
  m NUMBER NOT NULL := 0;
  a NUMBER;
  b NUMBER;
BEGIN
  m := m + 1; -- код с обработкой NULL
  m := a + b; -- исключение ORA-06502
  ...
END;
```

```
PROCEDURE calc IS
  m NUMBER;
  a NUMBER;
  b NUMBER;
BEGIN
  m := m + 1; -- без обработки NULL
  m := a + b;
  IF m IS NULL THEN
     -- обработка
  END IF;
  ...
END;
```

Ограничения NOT NULL, в частности, встроены (по определению) в типы NATURALN, POSITIVEN, SIMPLE_INTEGER, SIMPLE_FLOAT и SIMPLE_DOUBLE.

Числовые данные

Типы PLS_INTEGER и BINARY_INTEGER экономят память и время выполнения операций по сравнению с NUMBER. То же BINARY_FLOAT и BINARY_DOUBLE, но в конкретных случаях по памяти они могут оказаться более затратными.

Типы SIMPLE_INTEGER, SIMPLE_FLOAT и SIMPLE_DOUBLE дают повышенную скорость обработки (в случае SIMPLE_INTEGER, как утверждается, до 2—10 крат) *при* компиляции кода в режиме NATIVE.

Правила модуляризации кода

Для общей эффективности, читаемости и надежности программирования рекомендуется:

- не помещать в блоке между BEGIN и END более 60 строк («не более странички текста»);
- достигать этого путем использования подпрограмм, если надо, локальных;
- заключать регулярно используемые выражения и «бизнес-правила» в тела функций.

SQL против кода PL/SQL

В целом программирование действий на SQL эффективнее процедурного. Однако в некоторых случаях программирование может дать преимущество, например, в выполнении связанных операций UPDATE:

```
DECLARE CURSOR get_bonus IS SELECT ename, sal FROM bonus;
BEGIN
  FOR emp IN get_bonus LOOP
    UPDATE emp SET sal =  emp.sal WHERE ename = emp.ename;
  END LOOP;
END;
/
```

Замечание. Конкретно приведенный (фирмой Oracle) пример не совсем корректен. Так, выполняемое им изменение данных можно осуществить командой MERGE быстрее его.

Также, в общем рекомендуется не вставлять команды SQL в код открыто, а упрятывать их в подпрограммы, через которые и обращаться к ним.

Коллекции в обращении к данным БД

Рекомендуется использовать аппарат коллекций при обращении к данным в БД (FORALL ... и BULK COLLECT INTO)

Порядок в условных выражениях

Условные выражения вычисляются экономно. Как только выяснится, каким гарантировано окажется результат, дальнейшие вычисления прекращаются.

Цепочки условий, составленные из OR, и цепочки из AND вычисляются слева направо. Это можно использовать для общего ускорения проверок, помещая в левую часть цепочек

(а) наиболее легко вычисляемые подвыражения,
(б) подвыражения с высокой вероятностью нужного результата.

Например, если *expr1* легче вычисляется, чем *expr2*, и чаще дает TRUE, то выражение *expr1* OR *expr2* чаще будет вычисляться быстрее, чем *expr2* OR *expr1*.

Передача параметров

При передаче параметров подпрограмме Oracle рекомендует:

- использовать записи,
- использовать коллекции.

То и другое позволяет избегать длинных списков выражений при вызове подпрограмм и сделать код яснее, а вызов быстрее.

В случае, когда формальный параметр описан как IN OUT, и имеет тип коллекции, предотвратить излишние затраты памяти и времени при возврате значений, способна подсказка NOCOPY.

Подстановка кода вместо вызова

Указание INLINE при вызове *локальной* подпрограммы приведет к более быстрому исполнению кода. Поскольку это достигается за счет увеличения общего размера кода, применять эту технику следует только для небольших по размеру локальных подпрограмм, и, вдобавок, часто исполняемых. Например, это подходит для вызовов локальных подпрограмм из тела цикла.

Подстановку кода программист указывает компилятору индивидуально перед конкретном обращении к подпрограмме, прагмой INLINE, когда действует параметр компиляции PLSQL_OPTIMIZE_LEVEL = 2, или автоматически для всех вызовов, когда PLSQL_OPTIMIZE_LEVEL = 3 (и в этом случае прагмой INLINE придется, наоборот, отменять замену кода на вызов, если это потребуется).

17. Компиляция программ на PL/SQL

17.1. Параметры СУБД управления компиляцией

Ряд параметров СУБД, допускающих динамическое изменение как на уровне СУБД, так и сеанса, непосредственно управляют получением кода результата компиляции программных единиц на PL/SQL:

PLSCOPE_SETTINGS[11)]
PLSQL_CCFLAGS[11)]
PLSQL_CODE_TYPE[10)]
PLSQL_OPTIMIZE_LEVEL[10)]
PLSQL_WARNINGS[10)]
PLSQL_DEBUG[10]
PLSQL_V2_COMPATIBILITY[10]
PLSQL_NATIVE_…[9] (PLSQL_NATIVE_LIBRARY_DIR, PLSQL_NATIVE_LINKER и так далее)
NLS_LENGTH_SEMANTICS
[9] До версии 9 включительно.
[10] В версии 10.
[10)] С версии 10.
[11)] С версии 11.
[10] До версии 11.

Они позволяют получить более оптимальный код за более длительное время компиляции, включить в код больше информации для отладки за счет его разбухания и так далее. Описания параметров приведены в документации по Oracle.

Примеры:

```
ALTER SESSION SET PLSQL_DEBUG = TRUE;
-- устарело; тот же эффект достигается так:
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 1;
```

Выполнен перевод машины PL/SQL в СУБД на автоматическое включение в компилируемый код данных для последующей отладки. (Выполнение самой отладки обеспечивается наличием у пользователя привилегий DEBUG ANY PROCEDURE и DEBUG CONNECT SESSION.)

```
ALTER SESSION SET PLSQL_WARNINGS = 'ENABLE:ALL';
ALTER SESSION SET PLSQL_WARNINGS
                = 'ENABLE:SEVERE', 'DISABLE:INFORMATIONAL';
ALTER SESSION SET PLSQL_WARNINGS
                = 'ENABLE: 5001', 'DISABLE: ( 6000, 6001 )';
```

Если подобные установки значений параметров компиляции становятся правилом, их можно выставить (администратору) на уровне СУБД командой ALTER SYSTEM … .

С версии 10 параметры компиляции существующих программных единиц схемы удобно определять по специальной таблице USER_PLSQL_OBJECT_SETTINGS.

```
COLUMN name                 FORMAT A30
COLUMN type                 FORMAT A15
COLUMN plsql_optimize_level FORMAT 999
COLUMN plsql_code_type      FORMAT A12
SELECT
  name
, type
, plsql_optimize_level
, plsql_code_type
FROM
```

```
  user_plsql_object_settings
;
```

17.2. Условная компиляция

Условная компиляция была введена в Oracle10. Ее предназначение в том, чтобы дать технологию построения разных вариантов одной и той же программы, отталкиваясь от единого ее текста. Варианты могут задаваться версией СУБД (программа, откомпилированная СУБД разных версий будет давать несколько иной конечный код) или внешними установками.

Достигается условная компиляция включением в исходный текст программы особенных конструкций, которые обрабатываются особой фазой предварительной компиляции. Предварительная компиляция — это текстовая обработка, результатом которой является *текст* программы, на уже чистом языке PL/SQL, и обработка которого (компиляция) производится далее обычным путем.

17.2.1. Управляющие конструкции условной компиляции

Управляющие конструкции условной компиляции — две т. н. «директивы»: условная и выдачи ошибки компиляции. Общий вид условной директивы:

$IF *статичное_условное_выражение* $THEN
  фрагмент текста программы
[$ELSEIF *статичное_условное_выражение* $THEN
  фрагмент текста программы
] …
[$ELSE
  фрагмент текста программы
]
$END

Общий вид директивы прекращения компиляции и выдачи сообщения об ошибке:

$ERROR *статичная_строка_текста* $END

Примеры:

```
BEGIN $IF NOT FALSE $THEN DBMS_OUTPUT.PUT_LINE ( 'Вставка кода' ); $END END;
/
BEGIN $IF FALSE $THEN DBMS_OUTPUT.PUT_LINE ( 'Вставки нет' ); $END NULL; END;
/
BEGIN $ERROR 'Прекращаем компиляцию' $END NULL; END;
/
BEGIN $IF TRUE $THEN $ERROR 'Прекращаем компиляцию' $END $END NULL; END;
/
```

Использование в качестве статичных условий явно указанных TRUE и FALSE — скорее исключение, чем правило. Способ формулировать более полезные для практики выражения приведен в следующем разделе.

Текст *хранимой* программной единицы (а именно из таковых разработчик строит приложение) после обработки препроцессором программист может получить с помощью специального пакета DBMS_PREPROCESSOR. Пример:

```
EXECUTE DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE -
                     ( 'FUNCTION', 'SCOTT', 'COUNTME' )
```

Процедура GET_POST_PROCESSED_SOURCE позволит извлечь обработанный предкомпилятором текст не на экран, а в программу.

17.2.2. Выражения в директивах условной компиляции

Выражения, допускаемые в директивах условной компиляции, должны быть *статичны*; они строятся по сильно упрощенным, чем в PL/SQL, правилам. Им не позволено использовать все богатство типов переменных PL/SQL и функций, а употребляемые величины обязаны быть только простейшими постоянными (константами). Точные правила их построения приведены в документации по Oracle.

В жизни большую пользу для составления таких выражений оказывает системный пакет DBMS_DB_VERSION, где определен ряд постоянных, позволяющих ссылаться на номер текущей версии Oracle. Например:

```
DECLARE
var $IF DBMS_DB_VERSION.VER_LE_10_2 $THEN BINARY_INTEGER
                                    $ELSE SIMPLE_INTEGER := 1
    $END
;
BEGIN var := 1; END;
/
```

Этот текст даст разный результат при компиляции в Oracle разных версий:

В версии 10	*В версии 11*
DECLARE var **BINARY_INTEGER** ; BEGIN var := 1; END;	DECLARE var **SIMPLE_INTEGER := 1** ; BEGIN var := 1; END;

Аналогично программист может завести собственный пакет с определением служебных постоянных, чтобы ссылаться на них в выражениях для компиляции. Изменив значения служебных постоянных, и перетранслировав описание пакета, программист добьется получения разных вариаций кода из одного и того же основного исходного текста.

Параметризовать статичные выражения, помимо постоянных пакета, можно с помощью так называемых «справочных директив». Их разрешено устанавливать как для всей СУБД, так и для отдельных сеансов. Пример:

```
ALTER SESSION SET PLSQL_CCFLAGS = 'all_versions: TRUE, initial_val: 10';

DECLARE
var $IF $$all_versions $THEN BINARY_INTEGER
                       $ELSE SIMPLE_INTEGER := $$initial_val
    $END
;
BEGIN var := $$initial_val; END;
/
```

Имеется несколько встроенных «справочных директив»:

«Справочная директива»	*Смысл*
PLSQL_LINE	Номер строки исходного кода на PL/SQL
PLSQL_UNIT	Название программной единицы PL/SQL
Параметр СУБД управления компиляцией	Установленное в момент компиляции значение
PLSQL_UNIT_OWNER[12]	Имя владельца программной единицы или NULL для неименованного блока
PLSQL_UNIT_TYPE[12]	Тип программной единицы

[12] С версии 12.

Примеры обращения:

```
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'Значение PLSQL_LINE     : ' || $$PLSQL_LINE );
DBMS_OUTPUT.PUT_LINE ( 'Значение PLSQL_UNIT     : ' || $$PLSQL_UNIT );
DBMS_OUTPUT.PUT_LINE ( 'Значение PLSQL_CCFLAGS  : ' || $$PLSQL_CCFLAGS );
DBMS_OUTPUT.PUT_LINE ( 'Значение PLSQL_CODE_TYPE: ' || $$PLSQL_CODE_TYPE );
END;
/

BEGIN
DBMS_OUTPUT.PUT_LINE ( 'PLSQL_UNIT_OWNER : ' || $$PLSQL_UNIT_OWNER );
DBMS_OUTPUT.PUT_LINE ( ' PLSQL_UNIT_TYPE : ' || $$PLSQL_UNIT_TYPE );
END;
/
```

17.2.3. Перекомпиляция

Компилировать *хранимую* подпрограмму с требуемым значением параметра условной компиляции можно, не прибегая к установке параметра сеанса `ALTER SESSION SET PLSQL_CCFLAGS =...`. Это способна сделать команда перекомпиляции `ALTER PROCEDURE/FUNCTION/PACKAGE BODY COMPILE ...`.

```
ALTER FUNCTION countme COMPILE;
```

Когда программная единица компилируется, это происходит с учетом текущих для сеанса установок параметров. При желании перекомпилировать программную единицу с установками параметров, соответствующих ее последней прошлой компиляции, в команду ALTER *тип_программной_единицы* следует добавить специальное указание, например:

```
ALTER FUNCTION countme COMPILE REUSE SETTINGS;
```

При этом не возбраняется какие-то параметры при повторной компиляции установить иначе. Например, если бы в коде функции COUNTME присутствовали формулировки компиляции, зависимой от условного параметра debuging, перекомпилировать ее для отладки можно было бы так:

```
ALTER FUNCTION countme COMPILE PLSQL_CCFLAGS = 'debuging:TRUE'
REUSE SETTINGS
;
```

17.3. Таблицы словаря-справочника

В отладке подпрограмм на PL/SQL полезны следующие таблицы словаря-справочника:

Таблица	*Описание*
USER_OBJECTS	Описание всех объектов пользователя, включая FUNCTION, PROCEDURE, TRIGGER, PACKAGE и PACKAGE BODY, TYPE
USER_PROCEDURES[9-]	Перечень подпрограмм пользователя и их общих характеристик
USER_ARGUMENTS[9-]	Перечень аргуменов подпрограмм пользователя и их общих характеристик
USER_PLSQL_OBJECT_SETTINGS[10-]	Параметры компиляции имеющихся программных единиц
USER_DEPENDENCIES	Взаимные зависимости объектов (в том числе программных элементов)
USER_OBJECT_SIZE	Описание размеров объектов пользователя, в том числе PL/SQL

USER_SOURCE	Исходный текст хранимых объектов
USER_ERRORS	Список ошибок для хранимых объектов пользователя
USER_TRIGGERS	Триггерные процедуры пользователя

[9-) С версии 9.
[10-) С версии 10.

В SQL*Plus ошибки последней трансляции хранимого объекта можно посмотреть командой SHOW ERRORS (в своей работе она обращается к USER_ERRORS).

Для редакций объектов БД начиная с версии Oracle 11.2 имеются свои аналогичные и другие связанные с ними справочные таблицы:

USER_OBJECTS_AE
USER_SOURCE_AE
USER_ERRORS_AE

а также:

ALL_EDITIONS
ALL_EDITION_COMMENTS
USER_VIEWS_AE
USER_EDITIONING_VIEWS
USER_EDITIONING_VIEWS_AE
USER_EDITIONING_VIEW_COLS
USER_EDITIONING_VIEW_COLS_AE

Для некоторых таблиц созданы дополнительные сценарии, повышающие удобство пользования. Так, для просмотра дерева зависимостей объекта (не обязательно только программной единицы), помимо таблицы USER_DEPENDENCIES, можно воспользоваться средствами из *utldtree.sql* из *%ORACLE_HOME %\rdbms\admin*. Прогон этого сценария создает в схеме пользователя несколько служебных объектов, включая процедуру DEPTREE_FILL, способную показать все объекты схемы, зависимые от указаного:

```
@?/rdbms/admin/utldtree
COLUMN dependencies FORMAT A80

EXECUTE deptree_fill ( 'procedure', 'scott', 'deptree_fill' )
SELECT * FROM idepttree;

EXECUTE deptree_fill ( 'sequence', 'scott', 'deptree_seq' )
SELECT * FROM idepttree;

-- ... и так далее.
```

17.4. Зависимости подпрограмм и перекомпиляция

Время последнего изменения описания объекта можно посмотреть в столбце LAST_DDL_TIME таблицы USER_OBJECTS. Если вызывающая подпрограмма обращается к объекту, который поменял определение (например, к другой подпрограмме), то СУБД, при ближайшем обращении к ней, сделает попытку ее перекомпилировать самостоятельно (и только после этого выполнить). Ради надежности и скорости работы программы при ближайшем вызове, бывает разумно перекомпилировать ее вручную, например:

```
ALTER FUNCTION countme COMPILE;
```

Необходимость предупредительной перекомпиляции проще бывает определить по значению столбца STATUS в той же таблице USER_OBJECTS. Чаще всего, после изменения объектов, от которых зависит подпрограмма, значение этого столбца для подпрограммы принимает значение INVALID.

Перекомпиляции администратором всех объектов БД служит сценарий *utlrp.sql* в *%ORACLE_HOME %\rdbms\admin*. Oracle советует запускать его после повышения либо понижения версии ПО, в том числе

после наложения на ПО заплатки, в результате чего многие объекты могут поменять свое состояние на INVALID.

Есть ограничение: когда вызываемая подпрограмма расположена в удаленной БД, СУБД не в состоянии самостоятельно зафиксировать ее изменение (буде оно произойдет), и необходимость предупредительной перекомпиляции основной подпрограммы разработчик вынужден будет определять, разве что сравнением времен последней компиляции подпрограмм.

17.5. Методы компиляции

С версии 11 параметрами компилятора, определяющими размер и качество результирующего кода, являются:

Параметр	*Значения*
PLSQL_CODE_TYPE	NATIVE, INTERPRETED
PLSQL_OPTIMIZE_LEVEL	0, 1, 2, 3

17.5.1. Тип кода-результата

Параметр PLSQL_CODE_TYPE указывает компилятору выдавать результат либо в виде интерпретируемого машиной PL/SQL байт-кода, либо в виде кода компьютера (но хранимого в БД).

Особенности PLSQL_CODE_TYPE = INTERPRETED

Быстрая компиляция.
Возможность отладки программ средствами SQL Developer и пр.

Особенности PLSQL_CODE_TYPE = NATIVE

Замедленная компиляция.
Более эффективный в исполнении код.
Касается только процедурных участков PL/SQL. Встроенные в программу запросы к БД исполняются, как обычно (то есть, если подпрограмма состоит из одних запросов к БД, или имеющиеся в ней запросы «тяжелые», исполняться быстрее она не станет).
При большом количестве одновременно используемых программных единиц (условно считается, свыше 15000), скомпилированных в этом режиме, требуется много памяти в SGA (shared pool).

Если решено взять за правило компилировать исходный код в режиме NATIVE, проще установить его командой ALTER SYSTEM. Для того, чтобы перекомпилировать все программные единицы БД в этом режиме, Oracle дает готовый сценарий *dbmsupgnv.sql* в *%ORACLE_HOME%\rdbms\admin*.

17.5.2. Степень оптимизации

Параметр PLSQL_OPTIMIZE_LEVEL заставляет компилятор применять более, или же менее сложные алгоритмы оптимизации исходного кода. Особенности (по мере возрастания уровня оптимизации):

PLSQL_OPTIMIZE_LEVEL = 0

Компиляция осуществляется с точным (как правило) соблюдением последовательности вычислений в исходном коде (правила Oracle9-).
Типы PLS_INTEGER и BINARY_INTEGER различаются в соответствии с правилами Oracle9-.

PLSQL_OPTIMIZE_LEVEL = 1

Последовательность вычислений в исходном коде в целом сохраняется.
Устраняется, где можно определить, код ненужных вычислений и исключений.

PLSQL_OPTIMIZE_LEVEL = 2

Последовательность вычислений в исходном коде может быть перестроена с соблюдением общего результата.

Возможна подстановка кода локальной подпрограммы вместо ее вызова там, где это потребовать прагмой PRAGMA INLINE (..., 'YES').

PLSQL_OPTIMIZE_LEVEL = 3

Имеет силу с версии Oracle 11.

Последовательность вычислений в исходном коде может быть перестроена с сохранением результата работы.

Осуществляется подстановка кода локальных подпрограмм вместо их вызова в исходном коде, если того не запрещать прагмой PRAGMA INLINE (..., 'NO').

Проверка:

```
CREATE OR REPLACE PROCEDURE counter IS
  x  NUMBER;
  t1 NUMBER;
  t2 NUMBER;
  t3 NUMBER;
BEGIN
  t1 := DBMS_UTILITY.GET_CPU_TIME ( );
  FOR a IN 1 .. 10000000 LOOP
    x := 1 + 2 - 3 * 4 / 5;
  END LOOP;
  t2 := DBMS_UTILITY.GET_CPU_TIME ( );
  t3 := ( t2 - t1 ) / 100;
  DBMS_OUTPUT.PUT_LINE ( 'CPU = ' || t3 || ' seconds' );
END;
/
```

Далее:

```
ALTER PROCEDURE counter COMPILE PLSQL_OPTIMIZE_LEVEL = 0;
EXECUTE counter
ALTER PROCEDURE counter COMPILE PLSQL_OPTIMIZE_LEVEL = 1;
EXECUTE counter
ALTER PROCEDURE counter COMPILE PLSQL_OPTIMIZE_LEVEL = 2;
EXECUTE counter
ALTER PROCEDURE counter COMPILE PLSQL_OPTIMIZE_LEVEL = 3;
EXECUTE counter
```

18. Отладка программ на PL/SQL

18.1. Системные пакеты в помощь отладке

Осуществлять отладку программ на PL/SQL помогают системные пакеты DBMS_OUTPUT (отладочная выдача на экран), DBMS_PROFILER, DBMS_TRACE, DBMS_UTILITY (частично) а также DBMS_DEBUG.

18.1.1. Пакет DBMS_PROFILER

Две функции пакета (существующие также в варианте процедур), имена которых характеризуют способ их употребления:

Функция	***Описание***
START_PROFILER	Стартует создание профиля текущего сеанса. Профиль будет сохранен в специальных таблицах. При старте профиль можно именовать.
STOP_PROFILER	Завершает профилирование сеанса

Профиль исполняемых в промежутке между их вызовами подпрограмм автоматически фиксируется в особых служебных таблицах. Просмотр профилей — путем запрашивания данных этих таблиц.

18.1.2. Пакет DBMS_HPROF

«Иерархический profiler», с версии 11, позволяет замерять профиль исполнения кода на PL/SQL примерно так же, как DBMS_PRIFILER, но с дополнительными удобствами и возможностями.

18.1.3. Пакет DBMS_TRACE

Позволяет включать и выключать трассировку выполнения (вызовы, исключительные ситуации) отдельных локальных для сервера подпрограмм (т.е. не удаленных) или же всех подпрограмм определенного сеанса. По умолчанию файл трассировки образуется в каталоге, указанном параметром СУБД USER_DUMP_DEST (до версии 11) или DIAGNOSTIC_DEST (с версии 11).

Старт трассировки:

```
EXECUTE DBMS_TRACE.SET_PLSQL_TRACE ( уровень_трассировки )
```

где *уровень_трассировки* может быть построен следующими масками:

```
DBMS_TRACE.TRACE_ALL_CALLS          CONSTANT INTEGER := 1;
DBMS_TRACE.TRACE_ENABLED_CALLS      CONSTANT INTEGER := 2;
DBMS_TRACE.TRACE_ALL_EXCEPTIONS     CONSTANT INTEGER := 4;
DBMS_TRACE.TRACE_ENABLED_EXCEPTIONS CONSTANT INTEGER := 8;
... (TRACE_ALL/ENABLED_SQL, TRACE_ALL/ENABLED_LINES)
```

Маски в употреблении могут складываться, по типу DBMS_TRACE.SET_PLSQL_TRACE
(DBMS_TRACE.TRACE_ALL_CALLS + DBMS_TRACE.TRACE_ENABLED_SQL).

Останов трассировки:

```
EXECUTE DBMS_TRACE.CLEAR_PLSQL_TRACE
```

При выборочной трассировке она будет распространяться на все создаваемые и заменяемые (CREATE, REPLACE) программы — в том случае, если было выдано:

```
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 1;
```

(Режим DEBUG.)

Чтобы при выборочной трассировке программы можно было включать в состав трассируемых индивидуально, нужно выдать:

```
ALTER {PROCEDURE | FUNCTION | PACKAGE BODY} имя_программы
   COMPILE PLSQL_OPTIMIZE_LEVEL = 1;
```

а затем:

```
EXECUTE DBMS_TRACE.SET_PLSQL_TRACE ( DBMS_TRACE.TRACE_ENABLED_CALLS )
```

(или маску DBMS_TRACE.TRACE_ENABLED_EXCEPTIONS, если желательно трассировать возникающие исключительные ситуации, или нужную суперпозицию масок).

В ORACLE_HOME имеется скрипт *tracetab.sql* для создания таблиц, куда будет заноситься трассировочная информация, и откуда ее можно будет получать запросом SELECT:

```
@?/rdbms/admin/tracetab
GRANT SELECT ON plsql_trace_runs TO scott;
GRANT SELECT ON plsql_trace_events TO scott;
```

Пакет DBMS_TRACE более прост в работе, чем DBMS_PROFILER, но отслеживает только передачу управления.

18.1.4. Функции пакета DBMS_UTILITY и пакет UTL_CALL_STACK

18.1.4.1. DBMS_UTILITY

Некоторые функции из пакета DBMS_UTILITY могут быть использованы в хранимых процедурах, например, в разделе обработки исключительных ситуаций, в целях отладки.

Функция FORMAT_ERROR_STACK возвращает содержимое магазина (стека) ошибок (объемом не более 2000 символов).

Функция FORMAT_CALL_STACK возвращает содержимое магазина вызовов подпрограмм (объемом не более 2000 символов).

Функция FORMAT_ERROR_BACKTRACE позволяет узнать номер строки, где случилась ошибка (с версии 10).

Подготовим сценарий для примера в SQL*Plus:

```
BEGIN
   RAISE NO_DATA_FOUND;
EXCEPTION WHEN OTHERS THEN
   DBMS_OUTPUT.PUT_LINE ( &1 );
END;
.
SAVE debug_option REPLACE
```

Выдадим в SQL*Plus:

```
SET verify off
@debug_option 1234
@debug_option DBMS_UTILITY.FORMAT_CALL_STACK
@debug_option DBMS_UTILITY.FORMAT_ERROR_STACK
```

```
@debug_option DBMS_UTILITY.FORMAT_ERROR_BACKTRACE
```

Упражнение. Объяснить результаты.

В версии 12 функции FORMAT_CALL_STACK развивает пакет UTL_CALL_STACK, приспособленный для программирования анализа стека ошибок.

18.1.4.2. UTL_CALL_STACK (12+)

Функции пакета UTL_CALL_STACK (с версии 12) дают возможность исследовать стеки вызовов, ошибок и строк обращений с большей подробностью и большей степенью удобства для программиста, выполняющего отладку.

18.1.5. Пакет DBMS_DEBUG

Позволяет производить на сервере отладку указанного сеанса связи с Oracle путем запуска параллельно специального вспомогательного сеанса. Позволяет расставлять в программах отлаживаемого сеанса (в процедурах, функциях, пакетах, телах пакетов, триггерных процедурах, анонимных блоках, объектных типах и в телах объектных типов) контрольные точки, осуществлять пошаговое выполнение (до возникновения интересующего события), а также читать и изменять переменные программ.

Наиболее распространенное употребление пакета — в графических отладчиках, преимущественно третьих фирм.

18.2. Пример построения профиля работы приложения

18.2.1. Средствами пакета DBMS_PROFILER

Употребление пакета обеспечивается следующими файлами:

Файл	***Описание***
dbmspbp.sql[rdbms]	Создает внешнее определение пакета DBMS_PROFILER
prvtpbp.sql[rdbms]	Создает тело пакета DBMS_PROFILER (текст файла — объектный код вместо исходного)
profload.sql[rdbms]	Запускает файлы *dbmspbp.sql* и *prvtpbp.sql* и делает необходимые проверки. Должен выполняться от имени SYS.
proftab.sql[rdbms]	Сценарий создания рабочих таблиц для сбора данные профилей исполнения подпрограмм на PL/SQL: - PLSQL_PROFILER_RUNS - PLSQL_PROFILER_UNITS - PLSQL_PROFILER_DATA и для создания генератора чисел PLSQL_PROFILER_RUNNUMBER
profrep.sql[pls]	Создает набор представлений данных и пакет PROF_REPORT_UTILITIES, который можно использовать для более удобного извлечения информации из рабочих таблиц.
profsum.sql[pls]	Набор специальных запросов к данным профиля с использованием попрограмм из PROF_REPORT_UTILITIES.
profdemo.sql[pls]	Демонстрационный пример употребления.

[rdbms] Файл находится в *%ORACLE_HOME%\rdbms\admin*.

[pls] Файл находится в *%ORACLE_HOME%\plsql\demo*.

Ниже описаны действия в SQL*Plus, выполняемые на сервере (поскольку там находится ПО Oracle с используемыми в примере сценариями). Собственно использование пакета, естественно, возможно и на клиенте.

18.2.1.1. Подготовка к работе с пакетом

Установка пакета DBMS_PROFILER от имени SYS с выполнением необходимых проверок:

```
CONNECT / AS SYSDBA

@?/rdbms/admin/profload
```

Создание таблиц для хранения служебных данных о прогонах программ:

```
CONNECT scott/tiger

@?/rdbms/admin/proftab
```

18.2.1.2. Пример употребления

Создадим пару простых процедур:

```
CREATE OR REPLACE PROCEDURE first
AS
n NUMBER := 1;

BEGIN
FOR i IN 1 .. 1000 LOOP
   n := n + 1;
END LOOP;
END;
/

CREATE OR REPLACE PROCEDURE second
AS

BEGIN
DBMS_OUTPUT.PUT_LINE ( SIN ( 1 ) );
first;
END;
/
```

Обратите внимание:

- процедура SECOND обращается к FIRST
- в текстах имеются пустые строки
- в процедуре SECOND есть обращение к «системной» функции SIN и к «встроенному» пакету DBMS_OUTPUT.

Создание профиля работы (состоящей из поочередного выполнения двух процедур):

```
EXECUTE dbms_profiler.start_profiler ( 'Run@ ' || SYSTIMESTAMP )
EXECUTE first
EXECUTE second
EXECUTE dbms_profiler.stop_profiler
```

Профиль работы получил собственный номер. Его можно выявить запросом:

```
COLUMN run_comment FORMAT A60  WORD
COLUMN runid       FORMAT 9999
SELECT
  runid
, run_comment
, run_date
FROM
  plsql_profiler_runs
;
```

Этот текст удобно разместить в файле, например *seeprofiles.sql* в текущем каталоге.

Полученый так номер используется в запросе собственно профиля, например в таком:

```
SET VERIFY OFF
COLUMN owner   FORMAT A10
COLUMN name    FORMAT A10
COLUMN text    FORMAT A45   WORD
COLUMN line    FORMAT 999
COLUMN occured FORMAT 99999
SELECT
  u.unit_owner                    AS owner
, u.unit_name                     AS name
, s.line
, total_occur occured
, TRUNC ( d.total_time / 1000000 ) AS "TIME(ms)"
, s.text
FROM
  all_source           s
, plsql_profiler_data  d
, plsql_profiler_units u
WHERE
    u.runid       = &1
AND u.runid       = d.runid
AND u.unit_number = d.unit_number
AND s.name        = u.unit_name
AND s.type        = u.unit_type
AND s.line        = d.line#
ORDER BY unit_owner, name, line
;
SET VERIFY ON
```

В этом примере SQL*Plus запросит номер в диалоге.

Приведенный текст удобно разместить в файле, например *seeprofile.sql* в текущем каталоге.

Пример употребления запросов о профиле может выглядеть так:

```
SQL> @seeprofiles

RUNID RUN_COMMENT                                          RUN_DATE
----- ---------------------------------------------------- ---------
   10 Run@ 17-JAN-07 03.56.55.613000000 PM +03:00          17-JAN-07

1 rows selected.

SQL> @seeprofile 10

OWNER  NAME   LINE OCCURED TIME(ms) TEXT
------ ------ ---- ------- -------- ---------------------------------
SCOTT  FIRST     1       0        3 PROCEDURE first
SCOTT  FIRST     3       2        0 n NUMBER := 1;
SCOTT  FIRST     6    2002      105 FOR i IN 1 .. 1000 LOOP
SCOTT  FIRST     7    2000      194 n := n + 1;
```

```
SCOTT  FIRST     9       2       17 END;
SCOTT  SECOND    1       0        3 PROCEDURE second
SCOTT  SECOND    5       1     6108 DBMS_OUTPUT.PUT_LINE ( SIN ( 1 ) );
SCOTT  SECOND    6       2        3 first;
SCOTT  SECOND    7       1        0 END;

9 rows selected.
```

18.2.1.3. Другие возможности

Если собирается профиль по большому заданию, количество строк в запросе выше может оказаться велико. В таких случаях можно построить другой запрос, отобрав только сведения о строках подпрограмм, исполнявшихся чаще остальных, либо же дольше всех остальных исполнявшихся.

Столбцы таблиц с данными профилей (PLSQL_PROFILER_RUNS, PLSQL_PROFILER_UNITS, PLSQL_PROFILER_DATA) содержат и другую полезную информацию, например:

PLSQL_PROFILER_RUNS.RUN_TOTAL_TIME PLSQL_PROFILER_RUNS.RUN_COMMENT	Общее время работы задания Комментарий пользователя
PLSQL_PROFILER_UNITS.TOTAL_TIME PLSQL_PROFILER_UNITS.UNIT_TIMESTAMP	Общее время работы подпрограммы Момент трансляции подпрограммы (для учета смены версий)
PLSQL_PROFILER_DATA.MIN_TIME PLSQL_PROFILER_DATA.MAX_TIME	Минимальное и максимальное время исполнения конкретной строки

Эти столбцы также можно использовать в запросах для получения более общих или более подробных сведений.

Запуск профилирования действий пользователя можно сделать автоматическим, если включить обращение к DBMS_PROFILER.START_PROFILER в тело триггерной процедуры AFTER LOGON.

18.2.1.4. Поддержка работоспособности

Ввиду того, что таблицы с профилями контролируемых программ будут регулярно пополняться необходимо выработать регламент их чистки.

Чистка, ввиду имеющихся между таблицами связей, выполняется определенным порядком:

```
DELETE FROM plsql_profiler_data;
DELETE FROM plsql_profiler_units;
DELETE FROM plsql_profiler_runs;
```

Таблицы создаются в умолчательном табличном пространстве пользователя. При желании их можно перенести в иное место.

Наконец, для таблиц с данными профилей и для генератора чисел можно создать особую схему, одну на всю БД, предоставив пользователям свободный к ней доступ при помощи публичных синонимов. Доступ к только «собственным» строкам в общих таблицах PLSQL_PROFILER_* при желании можно ограничить средствами «виртуальных частных БД» (средством избирательного доступа к фрагментам таблиц).

18.2.2. Средствами пакета DBMS_HPROF

Выдаем разрешения и ориентируемся на директорию ОС:

```
CONNECT / as sysdba
CREATE OR REPLACE DIRECTORY extfiles_dir AS 'c:\temp'
```

```
;
GRANT READ, WRITE ON DIRECTORY extfiles_dir TO scott
;
GRANT EXECUTE ON dbms_hprof TO scott
;
```

Готовим инфраструктуру для использования пакета:

```
CONNECT scott/tiger
@?/rdbms/admin/dbmshptab
```

Проверка:

```
SELECT table_name FROM user_tables
;
SELECT sequence_name FROM user_sequences
;
```

Воспользуемся моделирующими подпрограммами из предыдущего раздела. Прогон выполняем аналогично:

```
EXECUTE dbms_hprof.start_profiling -
        ( location => 'EXTFILES_DIR', filename => 'profiler.txt' )
EXECUTE first
EXECUTE second
EXECUTE dbms_hprof.stop_profiling
```

Анадиз данных прогона и размещение результатов в таблицах профилировщика:

```
DECLARE
  l_runid  NUMBER;
BEGIN
  l_runid := DBMS_HPROF.analyze (
               location    => 'EXTFILES_DIR'
             , filename    => 'profiler.txt'
             , run_comment => 'Test run.'
             );
  DBMS_OUTPUT.PUT_LINE ( 'l_runid=' || l_runid );
END;
/
```

Положим, полученный номер прогона 2. Более полные сведения даст таблица DBMSHP_RUNS.

Информация о проработавших программных единицах:

```
COLUMN owner    FORMAT A20
COLUMN module   FORMAT A20
COLUMN type     FORMAT A20
COLUMN function FORMAT A25
SELECT symbolid,
       owner,
       module,
       type,
       function
FROM   dbmshp_function_info
WHERE  runid = 2
ORDER BY symbolid
;
```

Запрос с выяснением порядка вызовов выглядит сложно:

```
COLUMN name FORMAT A100

SELECT RPAD(' ', ( level - 1 ) * 2, ' ') || a.name AS name,
      a.subtree_elapsed_time,
```

```
       a.function_elapsed_time,
       a.calls
FROM
( SELECT fi.symbolid,
         pci.parentsymid,
         RTRIM ( fi.owner || '.' || fi.module || '.' || NULLIF(fi.function,fi.module), '.' )
            AS name,
         NVL ( pci.subtree_elapsed_time, fi.subtree_elapsed_time )
            AS subtree_elapsed_time,
         NVL ( pci.function_elapsed_time, fi.function_elapsed_time )
            AS function_elapsed_time,
         NVL ( pci.calls, fi.calls )
            AS calls
  FROM   dbmshp_function_info fi LEFT JOIN dbmshp_parent_child_info pci
         ON fi.runid = pci.runid AND fi.symbolid = pci.childsymid
  WHERE  fi.runid = 2
  AND    fi.module <> 'DBMS_HPROF'
) a
CONNECT BY a.parentsymid = PRIOR a.symbolid
START WITH a.parentsymid IS NULL
;
```

Выдачу результатов профилирования в виде серии страниц HTML можно получить через специальную программу *plshprof* (в Windows типа *.exe*):

```
>plshprof -output \temp\plshprof_output \temp\profiler.txt
```

Начальной страницей для просмотра здесь будет *\temp\plshprof_output.html*, а остальные просматриваются через ссылки. (Если работа идет в Windows, может оказаться надежнее запускать программу *plshprof* в той директории, где планируется иметь файлы отчетов, и не указывать, благодаря этому, путь к имени файла начальной страницы; то есть, в нашем случае указать просто: `plshprof -output plshprof_output profiler.txt`).

18.3. Готовые системы программирования и отладки для PL/SQL

Разными фирмами создано достаточно много систем программирования на PL/SQL с графическим интерфейсом, разной степени распространенности. Многие из них носят более общий характер и совмещают в себе функции навигатора по БД, среды разработки ПО и даже администрирования БД и СУБД. Ниже перечисляются некоторые из них.

Название	***Фирма***	***Местонахождение в Internet и краткое описание***
SQL Developer	Oracle Corp.	*http://www.oracle.com*. Среда разработки и отладки для SQL и PL/SQL. Распространяется бесплатно; существует со времени версии Oracle10 (но развивается независимо).
Toad for Oracle, Toad Development Suite for Oracle	Quest Software, с 2012 г. Dell Software	*http://software.dell.com/products/toad-for-oracle/*, *http://software.dell.com/products/toad-development-suite-for-oracle/*. Среда разработки для SQL и PL/SQL с длинной историей и с широкой популярностью.
SQL Navigator	Quest Software, с 2012 г. Dell Software	*http://software.dell.com/products/sql-navigator/*. Среда для составления и отладки запросов SQL и программ на PL/SQL; имеет пошаговую отладку и пр.
PL/SQL Developer	Allround Automations	*http://www.allroundautomations.nl/plsqldev.html*. Редактор, отладчик SQL и программ на PL/SQL; навигатор по БД и т. д.
TOra	Open source	*https://github.com/tora-tool/tora*. Среда разработки для SQL Oracle, MySQL and PostgreSQL.
Keep Tool Toolkit	Keep Tool	*http://www.keeptool.com/en/products.html*. Состоит из трех компонент: Hora (среда администрирования и разработки), ER-Diagrammer (инструмент для разработчика БД) и PL/SQL-Debugger (отладчик).
Manage IT SQL-Station	Computer Assosiates	*http://www.ca.com/products/manageit_sqlsta.htm*. Среда для составления и отладки программ на

		PL/SQL. Включает, к тому же, навигатор и некоторые средства слежения за работой БД.
JDeveloper	Oracle Corp.	*www.oracle.com/technetwork*. Последние выпуски JDeveloper имеют неплохие возможности отладки программ на PL/SQL, включая пошаговую отладку и использование контрольных точек, хотя JDeveloper — намного более общее средство разработки приложений.
PLEdit	Benthic Software	*http://www.benthicsoftware.com*. Графическая среда для редактирования и компиляции хранимого кода на PL/SQL.

19. Системные пакеты PL/SQL

Набор пакетов, созданных от имени SYS или SYSTEM для большого числа разных целей в областях разработки, поддержки определенных особенностей СУБД и администрирования.

Большинство сценариев для создания системных пакетов в словаре-справочнике БД содержится в каталоге *rdbms\admin* и вызывается для исполнения в процессе создании БД. (Если при каком-нибудь режиме установки Oracle нужный пакет не оказался создан в базе, его можно создать, запустив соответствующий сценарий).

19.1. Пакеты STANDARD и DBMS_STANDARD

Пакет STANDARD содержит встроенные функции для использования в SQL-запросах:

- символьные
- числовые
- функции с данными
- преобразования
- для работы с LOB-элементами
- прочие

Пакет DBMS_STANDARD содержит «расширение пакета STANDARD на уровне ядра». Процедуры и функции:

- для работы с транзакциями (вариант версии Oracle 10):

procedure **commit**
procedure **commit_cm**
procedure **rollback_nr**
procedure **rollback_sv**
procedure **savepoint**
procedure **set_transaction_use**

- для использования в триггерных процедурах и подпрограммах (вариант версии Oracle 10):

procedure **raise_application_error** (num binary_integer, msg varchar2, keeperrorstack boolean default FALSE);
function **inserting** return boolean;
function **deleting** return boolean;
function **updating** return boolean;
function **updating** (colnam varchar2) return boolean;
function **sysevent** return varchar2
function **dictionary_obj_type** return varchar2
function **dictionary_obj_owner** return varchar2
function **dictionary_obj_name** return varchar2
function **database_name return** varchar2
function **instance_num** return binary_integer
function **login_user** return varchar2
function **is_servererror** (errno binary_integer) return boolean
function **server_error** (position binary_integer) return binary_integer
function **des_encrypted_password** (user varchar2 default null) return varchar2
function **is_alter_column** (column_name varchar2) return boolean
function **is_drop_column** (column_name varchar2) return boolean
function **grantee** (user_list out ora_name_list_t) return binary_integer
function r**evokee** (user_list out ora_name_list_t) return binary_integer
function **privilege_list** (priv_list out ora_name_list_t) return binary_integer
function **with_grant_option** return boolean
function **dictionary_obj_owner_list** (owner_list out ora_name_list_t) return binary_integer

function **dictionary_obj_name_list** (object_list out ora_name_list_t) return binary_integer
function **is_creating_nested_table** return boolean
function **client_ip_address** return varchar2
function **sql_txt** (sql_text out ora_name_list_t) return binary_integer
function **server_error_msg** (position binary_integer) return varchar2
function **server_error_depth** return binary_integer
function **server_error_num_params** (position binary_integer) return binary_integer
function **server_error_param** (position binary_integer, param binary_integer) return varchar2
function **partition_pos** return binary_integer

Подпрограммы пакетов STANDARD и DBMS_STANDARD не требуют указания имени пакета при вызове, однако подпрограммы из DBMS_STANDARD могут требовать расширения именем схемы SYS, например **SYS**.CLIENT_IP_ADDRESS.

19.2. Прочие системные пакеты

Далее приводится выборочный перечень примеров некоторых системных пакетов и пояснений к ним. Более исчерпывающую информацию по пакетам можно найти в документации по Oracle и в текстах сценариев их заведения в БД.

19.2.1. Запись данных из программы в файл и обратно

Для этой цели можно воспользоваться системным пакетом UTL_FILE. Его интерфейсная (внешняя описательная) часть включает описания:

- процедур и функций (открытие, закрытие, чтение, запись и проч.)
- типа FILE_TYPE для файловой переменной (file handle).
- исключительных ситуаций
  INVALID_PATH
  INVALID_MODE (неправильно указан режим открытия в FOPEN)
  INVALID_FILEHANDLE
  INVALID_OPERATION
  READ_ERROR
  WRITE_ERROR
  INTERNAL_ERROR (внутренняя ошибка PL/SQL)

Средствами пакета можно работать только с файлами, находящимися на сервере, и расположенными в каталоге, указанном:

(а) параметром СУБД UTL_FILE_DIR (если этот параметр не выставлен, пакет не сможет ничего сделать). Значение UTL_FILE_DIR = * в *INIT.ORA* позволит работать из PL/SQL с файлами из любого каталога на сервере.
(б) объектом вида DIRECTORY, на который у пользователя есть полномочия.

Второй вариант действует с версии 9 СУБД, считается современным и был введен для повышения безопасности при работе программы с файлами. Первый вариант из существующей практики должен быть исключен.

Пример выдачи в файл перечня сотрудников с зарплатой:

```
DECLARE
   myfile UTL_FILE.FILE_TYPE;

BEGIN
   myfile := UTL_FILE.FOPEN ( 'EXTFILES_DIR', 'utlfile.lst', 'w' );

   FOR dname_rec IN ( SELECT ename, sal FROM emp )
```

```
   LOOP
      UTL_FILE.PUT ( myfile, RPAD ( dname_rec.ename, 15 ) );
      UTL_FILE.PUT ( myfile, LPAD ( dname_rec.sal,    6 ) );
      UTL_FILE.NEW_LINE ( myfile );
   END LOOP;

   UTL_FILE.FCLOSE ( myfile );

EXCEPTION
   WHEN UTL_FILE.INVALID_PATH THEN
      DBMS_OUTPUT.PUT_LINE ( 'Invalid file name or path' );
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE ( 'Unrecognized error' );
END;
/
```

Здесь предполагается, что объект EXTFILES_DIR категории DIRECTORY предварительно создан в БД. Как это можно сделать, приводится ниже.

Другие применения пакета, помимо записи в файл, включают возможность программно:

(а) создавать файлы
(б) проверять наличие файла
(в) читать из файла в базу. Однако, если такое чтение не обязательно выполнять в рамках программы, то эффективнее его можно выполнять штатным загрузчиком SQL*Loader.

Ограничения пакета:

- Невозможно записывать или читать строки длиною более 32767 байтов (версия 8.1.5)
- Пакетом невозможно удалить файл — его можно только обнулить[-9.0]
- Невозможно переименовать файл — его можно только скопировать в другой[-9.0]
- Обеспечивается только последовательный доступ к файлу
- Нельзя работать со ссылками на файл — только с реальными файлами[-9.0]

[-9.0] Ограничение снято в версии 9.2.

В версии 9.2 введен более предпочтительный избирательный контроль доступа к файлам на сервере, посредством объекта вида DIRECTORY. Вот как можно организовать работу с файлами без установки UTL_FILE_DIR:

```
CONNECT / as sysdba
CREATE OR REPLACE DIRECTORY extfiles_dir AS 'c:\crs';
GRANT READ ON DIRECTORY extfiles_dir TO scott;

CONNECT scott/tiger
SET SERVEROUTPUT ON
DECLARE fexists BOOLEAN;   flength NUMBER;   fblocksize NUMBER;
BEGIN
   UTL_FILE.FGETATTR (
     'EXTFILES_DIR'
   , 'plsql.doc'
   , fexists
   , flength
   , fblocksize
   );
   IF fexists THEN
      DBMS_OUTPUT.PUT_LINE ( 'File length is: ' || flength );
   ELSE
      DBMS_OUTPUT.PUT_LINE ( 'File not found.' );
   END IF;
END;
/
```

19.2.2. Шифрование данных

Несмотря на механизм разграничения доступа к данным в Oracle, пользователь SYS обладает достаточными полномочиями, чтобы суметь обратиться к любым данным любого пользователя в БД. Для того, чтобы обезопаситься от этого, а также случайного доступа со стороны других пользователей, особо важные данные можно перед помещением в базу шифровать.

19.2.2.1. Пакет DBMS_OBFUSCATION_TOOLKIT

Начиная с версии 8.1.6 с Oracle поставляется пакет для шифрования и расшифровки методом DES под названием DBMS_OBFUSCATION_TOOLKIT. Процедурой DESENCRYPT этого пакета можно с помощью ключа зашифровать текстовую строку, а процедурой DESDECRYPT расшифровать.

Пример в SQL*Plus:

```
DECLARE
   x       VARCHAR2 (255) :=
   'Morgen, morgen, nur nicht heute, sagen alles faulen Leute';
   my_data VARCHAR2 ( 255 );
BEGIN
   DBMS_OUTPUT.PUT_LINE ( x );

   my_data := RPAD ( x, (TRUNC (LENGTH (x) / 8) + 1 ) * 8, CHR ( 0 ) );
   DBMS_OBFUSCATION_TOOLKIT.DESENCRYPT (
             input_string     => my_data
           , key_string       => 'MagicKey'
           , encrypted_string => x
           );
   DBMS_OUTPUT.PUT_LINE ( x );

   DBMS_OBFUSCATION_TOOLKIT.DESDECRYPT (
             input_string     => x
           , key_string       => 'MagicKey'
           , decrypted_string => my_data
           );
   x := RTRIM ( my_data, CHR( 0 ) );
   DBMS_OUTPUT.PUT_LINE ( x );
END;
/
```

Помимо этого пакет позволяет работать с шифрованием 3DES и сверткой MD5, а также порождать ключи для шифрования алгоритмами DES и 3DES.

Сценарий заведения пакета в БД находится в *rdbms\admin\catobtk.sql*.

19.2.2.2. Пакет DBMS_CRYPTO

В версии 10 в состав системных пакетов включен (в перспективе — на замену DBMS_OBFUSCATION_TOOLKIT) более функциональный пакет DBMS_CRYPTO, позволяющий шифровать данные других типов (не RAW и VARCHAR2, а RAW, CLOB и BLOB) и другими алгоритмами (не только DES, 3DES, но еще и AES, и RC4, и 3DES_2KEY плюс алгоритмы хеширования плюс параметризация этих алгоритмов). Он построен по иной технике, когда для шифрования используются всего две общие функции ENCRYPT и DECRYPT, а ссылка на алгоритм шифрования передается параметром. Вот как с помощью DBMS_CRYPTO может выглядеть DES-шифрование той же строки тем же ключом, что и выше:

```
DECLARE
input_string VARCHAR2 ( 255 ) :=
          'Morgen, morgen, nur nicht heute, sagen alles faulen Leute';
```

```
raw_input    RAW ( 4000 );

key_string       VARCHAR2 ( 8 )  := 'MagicKey';
raw_key          RAW ( 16 );

encrypted_raw    RAW ( 4000 );
encrypted_string VARCHAR2 ( 4000 );

decrypted_raw    RAW ( 4000 );
decrypted_string VARCHAR2 ( 4000 );

BEGIN
DBMS_OUTPUT.PUT_LINE ( input_string );

raw_input := UTL_I18N.STRING_TO_RAW ( input_string, 'AL32UTF8' );

raw_key := UTL_RAW.CAST_TO_RAW ( CONVERT ( key_string, 'AL32UTF8' ) );

encrypted_raw := DBMS_CRYPTO.ENCRYPT (
                   TYP => DBMS_CRYPTO.DES_CBC_PKCS5
                 , SRC => raw_input
                 , KEY => raw_key
                 );

decrypted_raw := DBMS_CRYPTO.DECRYPT (
                   TYP => DBMS_CRYPTO.DES_CBC_PKCS5
                 , SRC => encrypted_raw
                 , KEY => raw_key
                 );

decrypted_string := UTL_I18N.RAW_TO_CHAR ( decrypted_raw, 'AL32UTF8' );

DBMS_OUTPUT.PUT_LINE ( decrypted_string );
END;
/
```

(Чтобы пример проработал в сеансе *любого* пользователя, возможно потребуется выполнить от имени SYS:

```
GRANT EXECUTE ON DBMS_CRYPTO TO PUBLIC;
```
)

Здесь технология не требует искусственного удлинения строки до кратности 8 символам, но зато необходимо предъявлять параметры в формате RAW и в кодировке AL32UTF8.

Константа DBMS_CRYPTO.DES_CBC_PKCS5 выше есть сумма трех констант ENCRYPT_DES (алгоритм шифрования), CHAIN_CBC (размер блоков, на которые в процессе шифрования будет разбиваться исходная строка) и PAD_PKCS5 (схема дополнения строки до требуемой кратности). Чтобы *в точности* воспроизвести пример с DBMS_OBFUSCATION_TOOLKIT, нужно будет дополнять шифруемую строку нулями, и тогда вместо

```
TYP => DBMS_CRYPTO.DES_CBC_PKCS5
```

указать

```
TYP => DBMS_CRYPTO.ENCRYPT_DES
     + DBMS_CRYPTO.CHAIN_CBC
     + DBMS_CRYPTO.PAD_ZERO
```

Упражнение. Проверить это, зашифровав строку процедурой DESENCRYPT пакета DBMS_OBFUSCATION_TOOLKIT, а расшифровав функцией DECRYPT пакета DBMS_CRYPTO.

Замечание. Современная технология не рекомендует на практике дополнять строку нулями, а пользоваться схемой PKCS5.

Пример указания алгоритма шифрования AES ключом в 128 разрядов:

```
TYP => DBMS_CRYPTO.ENCRYPT_AES128
     + DBMS_CRYPTO.CHAIN_CBC
     + DBMS_CRYPTO.PAD_PKCS5
```

Порождение ключей для шифрования делается функцией DBMS_CRYPTO.RANDOMBYTES по стандартизованном алгоритму RSA X9.31.

Сценарий заведения пакета в БД находится в *rdbms\admin\catoctk.sql*.

19.2.3. Плановый запуск заданий в Oracle

Обеспечивается встроенными в СУБД планировщиками двух видов: «старого» (поддерживается для обратной совместимости) и «нового» (с версии 10 СУБД). Программная работа с ними обеспечивается встроенными пакетами DBMS_JOB и DBMS_SCHEDULER соответственно.

19.2.3.1. Пакет DBMS_JOB

Вплоть до версии 9.2 основой для планировщика заданий служит пакет DBMS_JOB. Сценарий создания пакета содержится в файле *rdbms\admin\dbmsjob.sql*, а сценарий создания справочных таблиц для работы с пакетом — в файле *catjobj.sql*.

Пример постановки задания на исполнение:

```
DECLARE
   jobno NUMBER;
BEGIN
   DBMS_JOB.SUBMIT (
     what => 'BEGIN STATSPACK.SNAP; END;'
   , next_date => SYSDATE
   , interval => 'SYSDATE + 1 / 24'
   , job => jobno
   );
   COMMIT;
   DBMS_OUTPUT.PUT_LINE ( 'New job system ID is: ' || jobno );
END;
/
```

Этим сценарием на ежедневное исполнение будет поставлена процедура сбора статистики работы системы с помощью программы STATSPACK.SNAP. Программа будет запускаться специальными процессами СУБД от имени пользователя Oracle, запустившего этот сценарий (создавшего задание). Задание получит системный номер, выданный выходной переменной job процедуры SUBMIT (этот номер берется из системной последовательности SYS.JOBSEQ).

При невозможности выполнить задание Oracle пытается его запускать повторно с некоторым интервалом времени и после 16 безуспешных попыток помечает задание как «неисправное» (broken). Следить за работой заданий Oracle можно с помощью таблиц DBA_JOBS, USER_JOBS и DBA_JOBS_RUNNING.

Работу пакета DBMS_JOB регулируют следующие параметры СУБД:

- JOB_QUEUE_PROCESSES — задает число фоновых процессов, обрабатывающих очередь запуска заданий. Для того, чтобы пакет работал фактически, этот параметр обязан быть больше 0 (в версиях до 9 — по умолчанию 0)
- JOB_QUEUE_INTERVAL — интервал в секундах, через который пакет DBMS_JOB проверяет наличие в очереди заданий[-9.0]
- JOB_QUEUE_KEEP_CONNECTIONS — устанавливает время сохранения активности фоновых процессов после обработки очереди[-8.1].

[-9.0] начиная с версии 9.0 сделан скрытым

[-8.1] начиная с версии 8.1 сделан скрытым, в версии 10 упразднен

Пакет позволяет управлять только собственными заданиями, например он не позволит снять чужое задание даже пользователю SYS. Для управления заданиями прочих пользователей (прерывания, перезапуска и т.д.) SYS может использовать недокументированный пакет DBMS_IJOB.

19.2.3.2. Пакет DBMS_SCHEDULER

В версии 10 в СУБД появился значительно более развитый планировщик заданий, программная работа с которым обеспечивается средствами пакета DBMS_SCHEDULER. Планировщик версии 10 позволяет запускать на исполнение не только внутренние процедуры, но и программы в ОС.
Он использует следующие понятия:

- *Schedule* (расписание)
- *Program* (программа)
- *Job* (плановое задание = расписание + программа)
- *Window* (интервал для включения разных ресурсных планов)
- Некоторые другие.

«Программа» может быть хранимой процедурой PL/SQL или Java, внешней процедурой на C, блоком PL/SQL или же командой ОС. Запускать задания можно по принципу пакета DBMS_JOB, но с бо́льшим разнообразием формулировок, например:

```
BEGIN
DBMS_SCHEDULER.CREATE_JOB (
  job_name   => 'MY_JOB'
, job_type   => 'EXECUTABLE'
, job_action => 'cmd.exe /C dir > \out.txt'
, enabled    => TRUE
);
END;
/
```

Указанная команда ОС будет выполнена однократно, после чего задание автоматически исчезнет. Добавление следующих параметров приведет к тому, что команда будет выполняться в текущее (на момент запуска) время суток каждое воскресение:

```
, start_date      => SYSTIMESTAMP
, repeat_interval => 'FREQ=WEEKLY; BYDAY=SUN'
```

В то же время можно заранее создать «программу» и «расписание» процедурами CREATE_PROGRAM и CREATE_SCHEDULE, после чего выдать:

```
BEGIN
DBMS_SCHEDULER.CREATE_JOB (
  job_name      => 'MY_JOB'
, program_name  => 'MY_PROGRAM'
, schedule_name => 'MY_SCHEDULE'
);
END;
```

/

Справочная информация — в таблицах %SHEDULER%. Для работы планировщика в ОС должна быть запущена программа *extjob*. На Windows она запускается службой *OracleJobScheduler<имя_СУБД>*. Возможность формулировать плановые задания и осуществлять сопутствующие действия регулируется серией системных привилегий (например, CREATE JOB) и предопределенной ролью SCHEDULER_ADMIN.

Версия 11 дополнила пакет возможностями:

- запуска «легковесных» заданий, делающая реальным их создание и удаление сотнями за секунду;
- запуска заданий на удаленных машинах посредством использования специального агента;
- запуска заданий только на основной БД физического горячего резерва или на страхующей логического.

19.2.4. Управление динамическим размещением объектов в библиотечном буфере

Раз попавшие в библиотечный буфер объекты можно закреплять в ОЗУ для того, чтобы они не участвовали в схеме освобождения памяти под новые объекты. Для этого имеется системный пакет DBMS_SHARED_POOL. Процедуры KEEP и UNKEEP этого пакета закрепляют или открепляют от памяти объекты следующих типов в соответствии с указанием поля FLAG:

P или p	Пакет, процедура или функция
Q или q	Генератор чисел
R или r	Триггерная процедура
T или t	Тип (версия 8.1 и выше)
Любой другой символ (логично указывать C или c)	Курсор

Перечень закрепленных в памяти объектов можно получить в таблице V$DB_OBJECT_CACHE.

Процедура SIZES позволяет выдать список объектов, размер которых не превышает аргумент этой функции, задаваемый в килобайтах, например:

```
SQL> SET SERVEROUTPUT ON SIZE 100000
SQL> EXECUTE SYS.DBMS_SHARED_POOL.SIZES ( 70 )
```

Процедура ABORTED_REQUEST_TRESHOLD позволяет задать порог в байтах на размер размещаемого в памяти нового объекта. Если размер нового размещаемого объекта больше порогового значения *и* в shared pool нет места для размещения, Oracle не будет пытаться освободить для такого объекта место, удаляя не используемые в данный момент объекты.

Размер подпрограмм — кандидатов на закрепление в памяти можно посмотреть непосредственно из таблицы DBA_OBJECT_SIZE или из DBA_KEEPSIZES (создается при заведении DBMS_SHARED_POOL).

Сценарий создания пакета хранится в файлах *dbmspool.sql* и *prvtpool.plb* в *rdbms/admin*.

Фирма Oracle не гарантирует, что пакет DBMS_SHARED_POOL в будущих версиях останется, но пока он не отменен.

19.2.5. Манипулирование большими неструктурированными объектами

Главный системный пакет в этой области — DBMS_LOB (сценарий *dbmslob.sql*). С его помощью можно выполнять, в частности, следующие операции над разными типами объектов (полный список в документации по DBMS_LOB):

Операция	BFILE	BLOB	CLOB	NCLOB
APPEND		X	X	X

COMPARE	X	X	X	X
COPY		X	X	X
ERASE		X	X	X
FILECLOSE	X			
FILECLOSEALL	X			
FILEEXISTS	X			
FILEGETNAME	X			
FILEISOPEN	X			
FILEOPEN	X			
GETLENGTH	X	X	X	X
INSTR	X	X	X	X
LOADFROMFILE	X	X	X	X
READ	X	X	X	X
SUBSTR	X	X	X	X
TRIM		X	X	X
WRITE		X	X	X

(Длина объекта LOB в версии 8.1.7 может достигать 4 Гб -1, а с версии 10 до терабайт.)

Пояснение.

Объект LOB хранится в базе самостоятельно, а в полях строк таблиц (и в переменной PL/SQL) хранится указатель на него. Если в строке в поле типа LOB, или же в переменной программы, указатель отсутствует, то в выражениях это обозначается как NULL:

```
SQL> DECLARE c CLOB;
  2  BEGIN IF c IS NULL THEN DBMS_OUTPUT.PUT_LINE ('[NULL]'); END IF; END;
  3  /
[NULL]
```

Если в поле типа LOB указатель направляет на LOB-объект нулевой длины, считается, что значение «пусто» (empty). На один LOB-объект может быть заведено несколько указателей. Список LOB-объектов можно посмотреть в таблице DBA_LOBS.

Пример использования пакета:

```
DECLARE
document CLOB;
initext  VARCHAR2 ( 4000 ) := 'Приказ № 1';
BEGIN
   DBMS_LOB.CREATETEMPORARY ( document, TRUE, DBMS_LOB.SESSION );
   DBMS_LOB.WRITE ( document, length ( initext ), 1, initext );
   -- Продолжаем работу с документом ...
   --
   DBMS_LOB.FREETEMPORARY ( document );
END;
/
```

Пояснение примера.

Здесь в программе использовалась «временная LOB-переменная», то есть переменная, временно хранимая в базе. Срок хранения указан третьим параметром, DBMS_LOB.SESSION — это сеанс. Процедура DBMS_LOB.FREETEMPORARY способна удалить такую переменную раньше.

На временную LOB-переменную не распространяются обычные транзакционные правила существования.

Процедура DBMS_LOB.CREATETEMPORARY по правилам своей работы создает пустой LOB-объект, а инициализация и занесение значения в него выполняются в данном случае процедурой DBMS_LOB.WRITE.

С версии 9.2 некоторые операции с объектами LOB можно совершать, не прибегая к DBMS_LOB, а используя традиционный синтаксис для обычных строк (оператор LIKE, функция SUBSTR и так далее). Так, пример выше может быть переписан проще:

```
DECLARE
document CLOB;
initext  VARCHAR2 ( 4000 ) := 'Приказ № 1';

BEGIN
   document := initext;
   -- Продолжаем работу с документом ...
   --
   DBMS_LOB.FREETEMPORARY ( document ); -- если выходим из блока -- не обязательно
END;
/
```

... или даже еще проще:

```
SET SERVEROUTPUT ON
DECLARE
document CLOB := 'Приказ № 1';

BEGIN
   IF document LIKE '%№ 1%' THEN DBMS_OUTPUT.PUT_LINE ( 'One' ); END IF;
   -- Продолжаем работу с документом ...
   --
   DBMS_LOB.FREETEMPORARY ( document ); -- если выходим из блока -- не обязательно
END;
/
```

В последних двух примерах обращаться к процедуре DBMS_LOB.FREETEMPORARY не обязательно (по завершении сеанса память, выделенная здесь неявно для объекта LOB, освободится автоматически), но рекомендуется: по той же причине, по которой советуют закрывать открытые в программе курсоры.

Еще пример:

```
CREATE DIRECTORY extfiles_dir AS 'c:\crs';

CREATE TABLE documents ( descr VARCHAR2 ( 60 ), doc BLOB );

DECLARE
  bfile_locator BFILE := BFILENAME ( 'EXTFILES_DIR', 'sql.pdf' );
  blob_locator  BLOB;

BEGIN
  INSERT INTO documents
  VALUES ( 'Programming Oracle with SQL' , EMPTY_BLOB ( ) )
  RETURNING doc INTO blob_locator
  ;
  DBMS_LOB.FILEOPEN ( bfile_locator, DBMS_LOB.FILE_READONLY );
  DBMS_LOB.LOADFROMFILE (
    blob_locator
  , bfile_locator
  , DBMS_LOB.GETLENGTH ( bfile_locator )
  );
  COMMIT;
  DBMS_LOB.FILECLOSE ( bfile_locator );
EXCEPTION
  WHEN OTHERS
    THEN DBMS_OUTPUT.PUT_LINE ( 'Exception: ' || SQLERRM );
END;
/
```

Пояснение примера.

Системная функция BFILENAME позволяет инициализировать BFILE-переменную ссылкой на файл. Системная функция EMPTY_BLOB позволяет инициализировать поле таблицы пустым BLOB-объектом (нулевой длины).

Замечание. Если задача состоит только в загрузке строк, содержащих поля типов LOB, ее можно решить также с помощью программы SQL*Loader, и при большом количестве строк часто более эффективно.

19.2.6. Обращение к прошлым значениям данных в таблице

В версии 9 появился пакет DBMS_FLASHBACK, дающий возможность пользователю при использовании табличного пространства типа UNDO (автоматическое управление откатами, «сегментами отмены») обращаться к «старым» данным, последовавшие изменения которых были зафиксированы командой COMMIT. Используется свойство пространства UNDO хранить данные, когда-то заведенные СУБД для возможности отката, в продолжение некоторого периода времени по завершении транзакции.

Для пользования пакетом пользователю SCOTT нужно обеспечить следующее.

- Значения параметров СУБД, например:

  UNDO_MANAGEMENT = AUTO
  UNDO_RETENTION = *нужное_число_секунд* (например, 1800)
  UNDO_TABLESPACE = *имя_табличного_пространства_типа_UNDO*

- Права на обращение к пакету:

```
GRANT EXECUTE ON DBMS_FLASHBACK TO scott;
```

Пример последовательности команд в SQL*Plus для иллюстрации возможности чтения (удаленных) данных по состоянию на 5 минут назад:

```
SELECT COUNT ( * ) FROM emp;
DELETE FROM emp;
COMMIT;
SELECT COUNT ( * ) FROM emp;

CALL DBMS_FLASHBACK.ENABLE_AT_TIME ( SYSDATE - 5 / 1440 );
SELECT COUNT ( * ) FROM emp;
CALL DBMS_FLASHBACK.DISABLE;
```

Замечание. Между вызовами DBMS_FLASHBACK.ENABLE и DBMS_FLASHBACK.DISABLE не допускаются модифицирующие операции.

Пример последовательности команд в SQL*Plus, иллюстрирующей возможность восстановления удаленных ранее строк по состоянию, заданному заблаговременно полученным от СУБД системным номером SCN внесения изменений в БД:

```
VAR scn_saved NUMBER
EXECUTE :scn_saved := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER
PRINT scn_saved

SELECT COUNT ( * ) FROM emp;
DELETE FROM emp;
COMMIT;

DECLARE
   CURSOR back_ emps IS SELECT * FROM emp;
   emp_rec emp%ROWTYPE;
BEGIN
```

```
   DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER ( :scn_saved );
   OPEN back_emps;
   DBMS_FLASHBACK.DISABLE;

   LOOP
      FETCH back_emps INTO emp_rec;
      EXIT WHEN back_emps%NOTFOUND;
      INSERT INTO emp VALUES emp_rec;
   END LOOP;
   CLOSE back_emps;
   COMMIT;
END;
/
```

До версии 9.2 вместо простого INSERT INTO emp VALUES **emp_rec** во фразе VALUES приходится расписывать явно выражения для всех полей добавляемой в EMP строки.

В версии 9.2 актуальность приведенного конкретного примера ослабла ввиду появившейся существенно более простой альтернативы:

```
INSERT INTO emp VALUES SELECT * FROM emp AS OF SCN ( :scn_saved );
```

Однако подобная альтернатива не обесценивает пакет вообще. Так, процедурное восстановление данных, как выше, потенциально более функционально, нежели с помощью SQL.

Другой пример применения. Использование курсорного выражения в запросе фактически скрывает в себе процедурную логику обработки, а потому внешний оператор SELECT теряет свойство целостности по чтению. Чтобы его обеспечить, достаточно выдать следующее (в SQL*Plus):

```
VARIABLE scn NUMBER

EXECUTE :scn := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER

SELECT
     dname
   , CURSOR (
        SELECT ename
        FROM   emp AS OF SCN :scn
        WHERE  emp.deptno = dept.deptno
     )
FROM dept;
```

Наконец, с версии 11 пакет DBMS_FLASHBACK содержит процедуру для отмены изменений в БД, произведенных несколькими смежными или зависимыми последними транзакциями.

19.2.7. Получение метаданных и их использование

С версии 9 в Oracle имеется пакет DBMS_METADATA, позволяющий извлекать из БД описания хранимых объектов в текстовой форме. Простые примеры употребления в SQL*Plus:

```
SET LONG 10000
SELECT DBMS_METADATA.GET_DDL ( 'TABLE', 'EMP', 'SCOTT' ) FROM dual;
SELECT DBMS_METADATA.GET_XML ( 'TABLE', 'EMP', 'SCOTT' ) FROM dual;
```

Первый запрос выдаст полное описание таблицы EMP в виде команды SQL CREATE TABLE, а второй — в виде документа XML. Второе описание очень подробное, и его сокращенный вариант можно получить с версии 11:

```
SELECT DBMS_METADATA.GET_SXML ( 'TABLE', 'EMP', 'SCOTT' ) FROM dual;
```

(Здесь SXML означает сокращение от simple XML, а не название известного в технологиях XML формата.)

С версии 11.2 пакет дополнен другим, DBMS_METADATA_DIFF, позволяющим выполнять сравнительные операции над описаниями объектов.

Следующий пример выдаст серию команд ALTER TABLE, выдача которых приведет описание таблицы EMP к описанию DEPT:

```
SELECT
   DBMS_METADATA_DIFF.COMPARE_ALTER ( 'TABLE', 'EMP', 'DEPT' )
FROM
   dual
;
```

Следующий пример выдаст разницу в описаниях таблиц EMP и DEPT в формате SXML (смотри выше):

```
SELECT
   DBMS_METADATA_DIFF.COMPARE_SXML ( 'TABLE', 'EMP', 'DEPT' )
FROM
   dual
;
```

Подобным образом допускается работа не только с описаниями таблиц, но и объектов многих прочих видов.

19.2.8. Рассылка сообщений из программы на PL/SQL

Достигается средствами пакетов UTL_SMTP («старый», с общением с почтовым сервером на низком уровне) и UTL_MAIL (более поздний и более высокого уровня).

Пакет UTL_SMTP позволяет посылать из программы на PL/SQL сообщения электронной почты по протоколу SMTP. Он делает это через другой пакет, UTL_TCP, который в версии 8.1 был написан на Java, а позже переписан на C. Поэтому чтобы он работал на версиях до 9, в состав СУБД и в БД должна быть включена поддержка хранимых Java-процедур. Кроме этого, должен быть доступен сервер SMTP.

Пример использования пакета:

```
VARIABLE s NUMBER
VARIABLE m VARCHAR2 ( 256 )

DECLARE
FUNCTION send_mail (
   sender    IN VARCHAR2 DEFAULT 'Larry.Ellison@hotmail.com'
,  recipient IN VARCHAR2 DEFAULT 'Bill.Gates@mail.ru'
,  subject   IN VARCHAR2
,  body      IN VARCHAR2
)
RETURN INTEGER
IS
   sendstatus NUMBER := 0;
   mailhost   VARCHAR2 ( 30 ) := 'mail.net-burg.com';
   mail_conn  UTL_SMTP.CONNECTION;
   crlf       UTL_TCP.CRLF;
BEGIN
   mail_conn := UTL_SMTP.OPEN_CONNECTION  ( mailhost, 25 );
   UTL_SMTP.HELO ( mail_conn, mailhost );
   UTL_SMTP.MAIL ( mail_conn, sender );
   UTL_SMTP.RCPT ( mail_conn, recipient );
   UTL_SMTP.OPEN_DATA ( mail_conn );
   UTL_SMTP.WRITE_DATA
      ( mail_conn, 'From: "Sender" <' || sender || '>' || crlf );
   UTL_SMTP.WRITE_DATA
```

```
      ( mail_conn, 'To: "Recipient" <' || recipient || '>' || crlf );
   UTL_SMTP.WRITE_DATA ( mail_conn, 'Subject: ' || subject || crlf);
   UTL_SMTP.WRITE_DATA ( mail_conn, crlf || body );
   UTL_SMTP.CLOSE_DATA ( mail_conn );
   UTL_SMTP.QUIT ( mail_conn );
   RETURN sendstatus;
EXCEPTION
   WHEN OTHERS THEN
   :m := SQLERRM;              -- Отладка (для SQL*Plus)
   sendstatus := SQLCODE;      -- Код ошибки
   RETURN sendstatus;
END send_mail;

BEGIN
:s := send_mail ( subject => 'Hello', body => 'Sincerely yours' );
END;
/

PRINT m s
```

С версии 10.1 для рассылки может использоваться более удобный для конечного пользователя пакет UTL_MAIL. Этот пакет не устанавливается автоматически при создании БД, у устанавливать его следует вручную из *rdbms\admin\utlmail.sql* и *rdbms\admin\prvtmail.sql*. Помимо этого он использует в работе специальную переменную СУБД SMTP_OUT_SERVER.

20. Примеры употребления ссылки на курсор для разделения обработки запроса в программе

Ссылка на курсор позволяет открыть курсор процедурой на сервере и передать с помощью ссылки в клиентской программе возможность работать с курсором. Ниже эта техника иллюстрируется в простом и в более реальным вариантах.

20.1. Простой пример разделения открытия курсора и обработки

Выдадим в SQL*Plus:

```
VARIABLE refcur REFCURSOR

DECLARE
  TYPE rct IS REF CURSOR;
  somename VARCHAR2(20);

  PROCEDURE getcur1 (rc OUT rct) IS BEGIN OPEN rc FOR 'SELECT ename FROM emp'; END;
  PROCEDURE getcur2 (rc OUT rct) IS BEGIN OPEN rc FOR 'SELECT dname FROM dept'; END;
  PROCEDURE getcur3 (rc OUT rct, num IN NUMBER) IS
    BEGIN OPEN rc FOR 'SELECT dname FROM dept WHERE deptno = ' || TO_CHAR ( num ); END;
  PROCEDURE fetchandclose (rc IN rct) IS
    BEGIN
    LOOP
    FETCH rc INTO somename;
    EXIT WHEN rc%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (somename );
    END LOOP;
    CLOSE rc;
    END;

BEGIN
getcur1 ( :refcur ); fetchandclose ( :refcur );

getcur2 ( :refcur ); fetchandclose ( :refcur );

getcur3 ( :refcur, 30 ); fetchandclose ( :refcur );
END;
/
```

20.2. Более сложный пример разделения работы

В реальной программе описание типа (RCT) из примера выше чаще всего будет вынесено в пакет, а переменная привязки (REFCUR) и программа обработки курсора (FETCHANDCLOSE) будут вынесены в клиентскую программу, написанную, например, на Java, C или на Object Pascal, в то время как программы вычисления курсора (GETCUR1, GETCUR2, GETCUR3) останутся на сервере. Вот пример, как это можно организовать.

Создадим обобщенный пакет:

```
CREATE OR REPLACE PACKAGE generic_ref_cursor
AS
  TYPE refcur IS REF CURSOR;

  PROCEDURE get_ref_cursor ( sqlselect IN VARCHAR2, rc OUT refcur );
END;
/

CREATE OR REPLACE PACKAGE BODY generic_ref_cursor
AS
  PROCEDURE get_ref_cursor ( sqlselect IN VARCHAR2, rc OUT refcur )
  IS
  BEGIN OPEN rc FOR sqlselect;
  END;
END;
/
```

Возможный вариант использования пакета в PL/SQL и SQL*Plus:

```
SET SERVEROUTPUT ON
VARIABLE refcur REFCURSOR

DECLARE
  PROCEDURE fetchandclose (rc IN generic_ref_cursor.refcur)
  IS
  somename VARCHAR2(20);

  BEGIN
  LOOP FETCH rc INTO somename; EXIT WHEN rc%NOTFOUND;
  DBMS_OUTPUT.PUT_LINE ( somename );
  END LOOP;
  CLOSE rc;
  END;

BEGIN
generic_ref_cursor.get_ref_cursor ( 'SELECT ename FROM emp', :refcur );
fetchandclose ( :refcur );

generic_ref_cursor.get_ref_cursor ( 'SELECT dname FROM dept', :refcur );
fetchandclose ( :refcur );

generic_ref_cursor.get_ref_cursor
                              ( 'SELECT ename, sal FROM emp', :refcur );
                              -- ... а результат выдадим в SQL*Plus.
END;
/

PRINT refcur

BEGIN
generic_ref_cursor.get_ref_cursor ( 'SELECT * FROM emp', :refcur );
END;
/

PRINT refcur
```

Возможный вариант использования этой же техники в клиентской программе на Java:

```
import java.sql.CallableStatement;
import java.sql.ResultSet;
import oracle.jdbc.OracleCallableStatement;
import oracle.jdbc.OracleTypes;
...
...
```

```
  CallableStatement cst;
  OracleCallableStatement ocst;
  ResultSet rs;

  cst = cn.prepareCall
        ( "BEGIN generic_ref_cursor.get_ref_cursor ( ?, ? ); END;" );
  cst.setString ( 1, "SELECT sal FROM emp" );
  cst.registerOutParameter ( 2, OracleTypes.CURSOR );

  cst.execute ();

  ocst = ( OracleCallableStatement ) cst;

  rs = ocst.getCursor ( 2 );

  while ( rs.next () ) { System.out.println ( rs.getInt ( 1 ) ); }

...
```

21. Атрибуты триггерных процедур уровня схемы БД и событий в СУБД

Общий синтаксис описания триггерных процедур для схемы таков:

CREATE [OR REPLACE] TRIGGER *имя_триггерной_процедуры*
{ BEFORE | AFTER }
{ SERVERERROR | LOGON | LOGOFF | CREATE | DROP | ALTER }
ON ***имя_схемы*.SCHEMA**
BEGIN
    Текст_на_PL/SQL
END;

С каждым событием из таблицы выше связано несколько атрибутов. Фактически эти атрибуты — системные функции, возвращающие при обращении к ним из тела процедуры некоторый результат. Ниже эти атрибуты перечисляются, причем первые шесть из них относятся к уровню базы данных (но могут использоваться в триггерных процедурах уровня схемы).

Имя	***Тип***	***Описание***
SYSEVENT	VARCHAR2(30)	Имя события, активизировавшего триггерную процедуру
LOGIN_USER	VARCHAR2(30)	Имя пользователя, инициировавшего сеанс работы с Oracle
INSTANCE_NUM	NUMBER	Имя экземпляра СУБД
DATABASE_NAME	VARCHAR2(50)	Имя БД
SERVER_ERROR	NUMBER	Функция, возвращающая номер ошибки на указанном месте магазина ошибок. 1 соответствует верхушке магазина. Пример: SERVER_ERROR(2) выдаст номер ошибки на втором от верха месте в магазине.
IS_SERVERERROR	BOOLEAN	Функция, возвращающая TRUE при наличии указанной ошибке в текущем магазине ошибок; FALSE в противном случае.
DICTIONARY_OBJ_OWNER	VARCHAR2(30)	Владелец объекта из словаря-справочника, действие с которым привело к активизации триггерной процедуры.
DICTIONARY_OBJ_NAME	VARCHAR2(30)	Имя объекта из словаря-справочника, действие с которым привело к активизации триггерной процедуры
DICTIONARY_OBJ_TYPE	VARCHAR2(30)	Тип объекта из словаря-справочника, действие с которым привело к активизации триггерной процедуры
DES_ENCRYPTED_PASSWORD	VARCHAR2(30)	Зашифрованный (DES) пароль создаваемого или изменяемого пользователя.

Вот какие правила и атрибуты свойственны каждому событию:

Событие	***Правило***	***Атрибуты***
LOGON	Условие можно указать, воспользовавшись USERID( ) или USERNAME( )	SYSEVENT LOGIN_USER INSTANCE_NUM

		DATABASE_NAME
LOGOFF	Условие можно указать, воспользовавшись USERID( ) или USERNAME( )	SYSEVENT LOGIN_USER INSTANCE_NUM DATABASE_NAME
BEFORE CREATE AFTER CREATE	В пределах триггерной процедуры удалять создаваемый объект нельзя. Процедура выполняется в рамках текущей транзакции.	SYSEVENT LOGIN_USER INSTANCE_NUM DATABASE_NAME DICTIONARY_OBJ_TYPE DICTIONARY_OBJ_NAME DICTIONARY_OBJ_OWNER
BEFORE ALTER AFTER ALTER	В рамках процедуры удалять изменяемый объект нельзя. Процедура выполняется в рамках текущей транзакции.	SYSEVENT LOGIN_USER INSTANCE_NUM DATABASE_NAME DICTIONARY_OBJ_TYPE DICTIONARY_OBJ_NAME DICTIONARY_OBJ_OWNER
BEFORE DROP AFTER DROP	В рамках процедуры удалять изменяемый объект нельзя. Процедура выполняется в рамках текущей транзакции.	SYSEVENT LOGIN_USER INSTANCE_NUM DATABASE_NAME DICTIONARY_OBJ_TYPE DICTIONARY_OBJ_NAME DICTIONARY_OBJ_OWNER

Общий синтаксис описания триггерной процедуры для БД таков:

CREATE [OR REPLACE] TRIGGER *имя_триггера*
{ BEFORE | AFTER }
{ SERVERERROR | LOGON | LOGOFF | STARTUP | SHUTDOWN }
ON **DATABASE**
BEGIN
Текст_на_PL/SQL
END;

С каждым событием из таблицы выше связано несколько атрибутов. Фактически эти атрибуты — системные функции, возвращающие при обращении к ним из тела процедуры некоторый результат. Ниже эти атрибуты перечисляются, причем первые шесть из них нам уже знакомы по триггерным процедурам событий уровня схемы.

Имя	*Тип*	*Описание*
SYSEVENT	VARCHAR2(30)	Имя системного события, приведшего к запуску процедуры
LOGIN_USER	VARCHAR2(30)	Имя пользователя, вышедшего на сеанс работы с Oracle
INSTANCE_NUM	NUMBER	Имя экземпляра СУБД
DATABASE_NAME	VARCHAR2(50)	Имя БД
SERVER_ERROR	NUMBER	Функция, возвращающая номер ошибки на указанном месте магазина ошибок. 1 соответствует верхушке магазина. Пример: SERVER_ERROR(2) выдаст номер ошибки на втором от верха месте в магазине.
IS_SERVERERROR	BOOLEAN	Функция, возвращающая TRUE при наличии указанной ошибке в текущем

		магазине ошибок; FALSE в противном случае.

Вот какие правила и атрибуты свойственны каждому событию:

Событие	*Правило*	*Атрибуты*
SERVERERROR	По умолчанию процедура будет запускаться при всех событиях. Однако специальным указанием можно сообщить, чтобы процедура запускалась только при интересующих нас событиях.	SYSEVENT LOGIN_USER INSTANCE_NUM DATABASE_NAME SERVER_ERROR IS_SERVERERROR
LOGON	Условие можно указать, воспользовавшись USERID( ) или USERNAME( )	SYSEVENT LOGIN_USER INSTANCE_NUM DATABASE_NAME
LOGOFF	Условие можно указать, воспользовавшись USERID( ) или USERNAME( )	SYSEVENT LOGIN_USER INSTANCE_NUM DATABASE_NAME
STARTUP	В теле процедуры не допускается использование операций DML (в том числе SELECT). Можно, однако, запускать программы (например, listener), закреплять в SGA пакеты и т. д.	SYSEVENT LOGIN_USER INSTANCE_NUM DATABASE_NAME
SHUTDOWN	В теле процедуры не допускается использование операций DML (в том числе SELECT). Можно, однако, останавливать программы (например, listener), или запускать (например, сбора статистики работы СУБД и занесения ее в журнал)	SYSEVENT LOGIN_USER INSTANCE_NUM DATABASE_NAME

При написании тела процедуры нужно учитывать следующие обстоятельства:

- При запуске триггерной процедуры для событий в СУБД Oracle открывает автономную транзакцию, осуществляет сам запуск и фиксирует (commit) выполнение всех DML-операций безотносительно к логике транзакций пользователя.
- В определении процедур для событий LOGON, STARTUP и SERVERERROR можно указывать только слово AFTER. Если указать BEFORE, при трансляции будет выдана ошибка.
- Аналогично, в определении триггерных процедур LOGOFF и SHUTDOWN можно указывать только BEFORE.
- Обращения из процедур DBMS_OUTPUT не дадут на экране никакой выдачи в пределах текущего сеанса. Для того, чтобы как-то записать информацию, нужно будет воспользоваться записью в таблицы, в файл ОС или пакетом DBMS_PIPE.
- Триггерная процедура для SERVERERROR не срабатывает на следующие пять событий: ORA-01403, ORA-01422, ORA 04030, ORA-01034 и ORA-01007.

Страница для заметок