И.С. Задворьев

ЯЗЫК PL/SQL

Учебно-методическое пособие

УК № 3800

Москва 2017

И.С. Задворьев

Язык PL/SQL

учебно-методическое пособие

УК № 3800

Москва 2017

Задворьев И.С. Язык PL/SQL. Учебно-методическое пособие. — М., 2017. – 188 с.

Автор:
Задворьев И.С., кандидат технических наук.

В учебно-методическом пособии рассматриваются основы языка программирования PL/SQL, реализованного в системе управления базами данных Oracle Database Server. Приводятся сведения о поддерживаемых типах данных, структуре программ PL/SQL и выполнении SQL-предложений в них. Отдельно рассмотрено создание хранимых в базах данных Oracle программ PL/SQL — процедур, функций, пакетов и триггеров.

konaz@mail.ru

Введение в PL/SQL

Назначение PL/SQL

PL/SQL — «Procedural Language extensions to the Structured Query Language», что переводится как «Процедурные языковые расширения для языка SQL».

Практически в каждой СУБД корпоративного уровня есть язык программирования, предназначенный для расширения возможностей SQL:

- PL/SQL — в Oracle Database Server;
- Transact-SQL — в Microsoft SQL Server;
- SQL PL — в IBM DB2;
- PL/pgSQL — в PostgreSQL.

На этих языках создаются программы, которые хранятся непосредственно в базах данных и выполняются СУБД, поэтому их называют языками хранимых процедур (stored procedure languages). Языки хранимых процедур имеют схожие синтаксис и семантику, поэтому после освоения языка PL/SQL впоследствии можно будет довольно легко перейти, например, на Transact-SQL или PL/pgSQL.

Функция на языке Oracle PL/SQL	Функция на языке PL/pgSQL PostgreSQL

`CREATE FUNCTION F1 RETURN INT AS`
`BEGIN`
`  FOR r IN (SELECT * FROM tab1) LOOP`
`  UPDATE tab2 SET at3=r.at2;`
`  END LOOP;`
`  RETURN 1;`
`END;`
	`CREATE FUNCTION F1() RETURNS int AS '`
	`DECLARE`
	`  r RECORD;`
	`BEGIN`
	`  FOR r IN SELECT * FROM tab1 LOOP`
	`  UPDATE tab2 SET at3=r.at2;`
	`  END LOOP;`
	`  RETURN 1;`
	`END;`
	`' LANGUAGE plpgsql;`

Задачи, решаемые PL/SQL

PL/SQL, в отличие от Java, Python или C++, не используется для разработки математических приложений, игр и т. п. Это специфичный язык программирования третьего поколения, предназначенный для работы с базами данных Oracle прямо в ядре сервера Oracle.

Фактически программы на PL/SQL — это обертки вокруг предложений SQL.

Язык PL/SQL используется для решения следующих задач:

- реализация серверной бизнес-логики в виде хранимых программ;
- автоматизация задач администрирования баз данных Oracle;
- разработка web-приложений;
- разработка клиентских приложений в среде Oracle Developer.

Мы не будем останавливаться на автоматизации задач администрирования баз данных и разработке клиентских приложений, а сосредоточимся на главном направлении использования PL/SQL — реализации бизнес-логики на стороне сервера в виде хранимых программ.

Сценарий использования программ PL/SQL

Пусть в корпоративной сети на Linux-сервере находится база данных Oracle со сведениями о клиентах организации. Подключимся к серверу Oracle с ноутбука по сети с помощью утилиты SQL*Plus. Запуск на выполнение из SQL*Plus процедуры PL/SQL `calc_clients_debt` для расчета клиентской задолженности может выглядеть примерно так:

```
SQL> BEGIN
  2    calc_clients_debt(p_account_from=>100001,p_account_to=>200000);
  3  END;
  4  /

PL/SQL procedure successfully completed.
```

Только четыре строки для запуска процедуры `calc_clients_debt` будут переданы с ноутбука на Linux-сервер, где сервер баз данных Oracle, получив эти строки, выполнит процедуру PL/SQL. На ноутбук обратно вернутся только сведения об успешности завершения работы процедуры — одна строка. Требуемые для расчетов гигабайты финансовых данных для заданного диапазона в 100 000 лицевых счетов на ноутбук по сети передаваться не будут — выборка всех данных клиентов с помощью выполнения SQL из PL/SQL и все расчеты по ним в PL/SQL будут осуществляться ядром СУБД Oracle на мощном Linux-сервере. На этом же сервере, в этой же базе данных Oracle процедурой `calc_clients_debt` будут сохранены и результаты вычислений.

Так расчет задолженности мог выглядеть, если бы его запускал технический специалист, знающий устройство базы данных и предпочитающий работать с ней в SQL*Plus. Понятно, что сотрудники бухгалтерии или клиентского отдела не работают с базой данных в SQL*Plus. Для них должна быть разработана и установлена клиентская программа на C#, Java или другом языке программирования с экранными формами и отчетами. В этой программе на экранной форме пользователь задает диапазон обрабатываемых лицевых счетов и нажимает кнопку «Рассчитать задолженность».

Клиентская программа через соответствующие программные интерфейсы, которые есть в большинстве современных языков программирования, запускает в Oracle на выполнение хранимую процедуру `calc_clients_debt` и начинает показывать пользователю наполняющиеся песочные часы или бегающую полоску (progress bar). Сама программа при этом не осуществляет обработку данных, которая в это время идет на удаленном Linux-сервере. Как только хранимая процедура успешно завершится и сервер Oracle сообщит об этом клиентской программе, та выдаст пользователю сообщение «Задолженность успешно рассчитана».

Это типичный сценарий использования PL/SQL: реализация бизнес-логики (в данном примере — расчета клиентской задолженности) в виде хранимой в базе данных процедуры на PL/SQL с ее запуском из клиентской программы, подключившейся к серверу Oracle по сети. Обычно программы на PL/SQL работают «под капотом» и их не видно снаружи.

Достоинства и недостатки хранимых программ

При реализации бизнес-логики вполне можно обойтись и без использования хранимых программ. Так, задачу расчета клиентской задолженности можно решить двумя способами:

- разработать одно или несколько (frontend, backend) приложений на Java, JavaScript, C++, Python и т. п., реализующих только пользовательский интерфейс, а бизнес-логику собственно расчета задолженности реализовать в виде хранимой программы, которую вызывают приложения при запуске процесса расчета;
- разработать одно или несколько (frontend, backend) приложений на Java, JavaScript, C++, Python и т. п., реализующих и пользовательский интерфейс, и бизнес-логику расчета задолженности.

Для второго способа база данных используется только для хранения данных. Все необходимые данные по каждому клиенту извлекаются приложением из базы, обсчитываются приложением и полученные сведения о задолженности сохраняются обратно в базу. Обсчитывающее данные приложение часто размещают на том же сервере, где находится база данных — чтобы сеть не стала узким местом системы.

Выбор используемого способа решения задачи является обязанностью архитектора системы, при этом следует учитывать много факторов, формируемых в каждом конкретном случае на основе известных достоинств и недостатков использования хранимых программ.

Достоинства хранимых программ:

- переносимость хранимых программ вместе с базой данных;
- повышенная производительность обработки за счет отсутствия передачи данных вне сервера баз данных;
- тесная интеграция с подсистемой выполнения SQL (предложения SQL в хранимых программах выполняются без использования дополнительных интерфейсов и драйверов);
- управление доступом к данным на основе хранимых программ (доступ предоставляется не к таблицам базы данных на чтение и запись данных в них, а на выполнение хранимых программ — тем самым выполняется изоляция данных от прикладных программ);
- реализация динамических ограничений целостности и концепции активных баз данных с помощью механизма триггеров.

Недостатки хранимых программ:

- «размазывание» логики работы системы по нескольким программах, написанных на разных языках;
- необходимость наряду с программистами на Java, Python, C++ иметь в команде программиста баз данных;
- скудность выразительных возможностей языков хранимых процедур на фоне современных языков Java, Python, C++;
- непереносимость хранимых программ между различными СУБД;
- возможные проблемы с масштабированием.

Наиболее существенным недостатком хранимых программ является их привязка к конкретной СУБД. Например, при переходе с Oracle на PostgreSQL в рамках актуальной темы импортозамещения, все хранимые программы придется переписывать с PL/SQL на PL/pgSQL, а это приведет к существенным затратам на реинжиниринг кода PL/SQL, объем которого может составлять сотни тысяч строк[1].

Что же касается проблем масштабирования, то обработка данных непосредственно в базе данных средствами самой СУБД является достоинством хранимых программ до тех пор, пока обеспечивается требуемый уровень производительности. В противном случае это обстоятельство помешает масштабированию, так как установка дополнительных серверов потребует большого объема работ. Придется на каждом новом сервере устанавливать СУБД, создавать свою базу данных с хранимыми программами и решать задачу распределения данных по нескольким базам. Из достоинства хранимых программ интеграция хранения и обработки данных, таким образом, может стать недостатком. С отдельными приложениями, реализующими серверную бизнес-логику без хранимых программ, проблем масштабирования обычно нет — добавить новые сервера только для обсчитывающих приложений обычно довольно легко.

Переносимость программ PL/SQL

Переносимость программ PL/SQL вместе с базой данных обеспечивается архитектурой языка PL/SQL, похожей на архитектуру языка Java.

При программировании на языках C/C++, Pascal в результате работы компилятора получается исполняемый (executable) файл. Этот файл содержит машинные команды конкретной аппаратной платформы и предназначен для работы в конкретной операционной системе. Поэтому если исполняемый файл для Windows скопировать на компьютер с операционной системой Linux, то он там не запустится. Если исполняемый файл для одной аппаратной платформы

---

[1] На PL/SQL возможна реализация больших проектов. Например, при разработке системы управления предприятием Oracle E-Business Suite было написано более 60 миллионов строк кода на PL/SQL!

перенести на другую платформу (компьютер с другими машинными командами), то он там тоже не запустится. В итоге, если требуется обеспечить кроссплатформенность, то для одной и той же программы на C++, приходится иметь версию для Windows и версию для Linux, версию для x86 (32-х разрядную) и версию для x64 (64-х разрядную) и так далее.

Иначе дело обстоит с программированием на Java. В результате работы компилятора Java получается не исполняемый файл с машинными командами, а файл с байт-кодом (bytecode)[1] — машинно-независимым кодом низкого уровня, исполняемым интерпретатором байт-кода. Этот интерпретатор байт-кода называется виртуальной машиной Java (Java Virtual Machine, JVM). При запуске программы Java файл с ее байт-кодом подается на вход виртуальной машине Java, которая преобразует инструкции байт-кода в машинные коды конкретной платформы. Таким образом, для того, чтобы запустить программу на Java в той или иной среде, достаточно иметь для этой среды JVM[2]. Например, в ядре сервера Oracle есть виртуальная машина Aurora JVM, предназначенная для выполнения хранимых в базах данных Oracle программ Java.

В результате работы компилятора PL/SQL тоже получается не исполняемый файл, а байт-код, который называется p-code. Этот байт-код при запуске программ PL/SQL интерпретируется виртуальной машиной PL/SQL (PL/SQL Vitual Machine, PVM), находящейся в ядре СУБД Oracle. Виртуальная машина PL/SQL есть во всех версиях СУБД Oracle для любой операционной системы и для любой аппаратной платформы, поэтому программы PL/SQL остаются работоспособными при переносах баз данных Oracle с одних вычислительных систем на другие[3].

---

[1] Байт-код называется так, потому что длина каждого кода операции традиционно составляет один байт.

[2] Девиз языка Java так и звучит — Write Once, Run Anywhere.

[3] Может понадобиться перекомпиляция программ PL/SQL при переносе на другую платформу, но изменять исходный код не придется.

Оценка языка PL/SQL

Как язык программирования PL/SQL имеет следующие достоинства:

- статическая типизация[1];
- наличие средств обработки ошибок и пользовательских исключений;
- наличие лаконичных и удобных языковых конструкций для выполнения предложений языка SQL.

Считается, что эффективный высокопроизводительный код для работы с базой данных Oracle проще написать на PL/SQL, чем на любом другом процедурном языке программирования. В частности, в PL/SQL имеются специальные средства массовой обработки данных (bulk processing), позволяющие повысить производительность на порядок и более.

Приведем достаточно большую цитату из книги «Oracle для профессионалов», написанной Томом Кайтом, вице-президентом Oracle Corporation [18, стр. 48]:

«При разработке программного обеспечения базы данных я придерживаюсь достаточно простой философии, которая оставалась неизменной на протяжении многих лет:

- Все, что только возможно, должно делаться в **одном** предложении SQL. Верите или нет, но это возможно почти всегда. С течением времени это утверждение становится еще более справедливым. SQL — исключительно мощный язык.
- **Если что-то нельзя сделать в одном предложении SQL, то это необходимо реализовать на языке PL/SQL с помощью как можно более краткого кода. Следуйте принципу «больше кода = больше ошибок, меньше кода = меньше ошибок».**
- Если задачу нельзя решить средствами PL/SQL, попробуйте воспользоваться хранимой процедурой Java. Однако после выхода Oracle 9i и последующих версий потребность в этом возникает очень редко. PL/SQL является полноценным и популярным

[1] Статическая типизация рассматривается в параграфе «Типы данных PL/SQL».

языком третьего поколения (third-generation programming language — 3GL).

- Если задачу не удается решить на языке Java, попробуйте написать внешнюю процедуру C. Именно такой подход применяют наиболее часто, когда нужно обеспечить высокую скорость работы приложения, либо использовать API-интерфейс от независимых разработчиков, реализованный на языке C.
- Если вы не можете решить задачу с помощью внешней процедуры C, всерьез задумайтесь над тем, есть ли в ней необходимость.

... Мы будем использовать PL/SQL и его объектные типы для решения задач, которые в SQL решить невозможно или неэффективно. Язык PL/SQL существует уже очень долгое время — на его отработку ушло более 27 лет (к 2015 году); в действительности, возвращаясь к версии Oracle 10g, был переписан сам компилятор PL/SQL, чтобы стать в первую очередь оптимизирующим компилятором.

Никакой другой язык не связан настолько тесно с SQL и не является до такой степени оптимизированным для взаимодействия с SQL. Работа с SQL в среде PL/SQL происходит совершенно естественным образом, в то время как в любом другом языке, от Visual Basic до Java, применение SQL может оказаться довольно-таки обременительным».

Опция New Procedural Option — PL/SQL появилась в версии Oracle 6.0 в 1988 году. С тех пор на PL/SQL написаны миллионы строк кода серверной бизнес-логики и разработаны тысячи клиентских форм и отчетов в среде Oracle Developer. Многие годы Oracle Corporation демонстрировала свою приверженность PL/SQL и с выходом каждой новой версии Oracle Database Server в PL/SQL вводятся новые усовершенствования. Язык PL/SQL является неотъемлемой частью технологий Oracle и в планах корпорации декларируется его развитие и поддержка в будущем.

Первая программа на PL/SQL

По давней традиции, восходящей к языку C, учебники по языкам программирования начинаются с программы «Hello, World!» и описания типов данных изучаемого языка. Не будем нарушать эту традицию, но внесем в нее одно изменение. Так как язык PL/SQL предназначен для работы с базами данных Oracle, то строку «Hello,

World!» не будем задавать статически в исходном коде, а возьмем из таблицы базы данных.

```
CREATE TABLE hello_world_table(message VARCHAR2(30));
INSERT INTO hello_world_table VALUES('Hello, World!');
```

Выполним в SQL*Plus следующий код:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2    l_message VARCHAR2(30);
  3  BEGIN
  4    SELECT message INTO l_message FROM hello_world_table;
  5    DBMS_OUTPUT.PUT_LINE(l_message);
  6  END;
  7  /
Hello, World!

PL/SQL procedure successfully completed.
```

В представленной выше программе PL/SQL с помощью команды `SELECT INTO` значение столбца `message` строки таблицы `hello_world_table` считывается из базы данных и присваивается локальной переменной `l_message`, значение которой затем выводится в окне SQL*Plus. Переменная `l_message` предварительно объявлена в разделе объявлений после ключевого слова `DECLARE`.

Экранный вывод в PL/SQL осуществляет процедура `PUT_LINE` встроенного пакета `DBMS_OUTPUT`, который есть во всех базах данных Oracle. Можно считать, что процедура `DBMS_OUTPUT.PUT_LINE` в языке PL/SQL — аналог процедуры `printf` в языке C.

Напомним о двух вещах, важных при работе с утилитой SQL*Plus:

- для запуска в SQL*Plus программы PL/SQL на выполнение необходимо на новой строке напечатать символ / и нажать клавишу `Enter` на клавиатуре;
- в SQL*Plus экранный вывод программ на PL/SQL включается командой `SET SERVEROUTPUT ON` (по умолчанию экранный вывод выключен).

Если не выполнить команду `SET SERVEROUTPUT ON`, то в консоли SQL*Plus ничего печататься не будет. В популярном GUI-клиенте Quest SQL Navigator экранный вывод PL/SQL тоже по умолчанию выключен и включается специальной кнопкой «Turn the server

output», которая после нажатия должна остаться во «вдавленном» положении.

Типы данных PL/SQL

Напомним, что типом данных (data type) называется именованное множество значений данных заданной структуры, удовлетворяющих специфицированным ограничениям целостности и допускающих выполнение над ними определенного, ассоциированного с этим множеством набора операций. Например, числа и даты можно складывать и вычитать[1], а строки и логические значения нельзя.

Язык PL/SQL относится к языкам со статической типизацией. Статической типизацией называется проверка типов данных во время компиляции программ. К числу языков программирования со статической типизацией относятся Pascal, Java, C/C++/C#. Языки с динамической типизацией (JavaScript, Python, Ruby) выполняют большинство проверок типов во время выполнения программ. Статическая типизация позволяет выявлять ошибки во время компиляции, что повышает надежность программ.

Виды типов данных

Так как язык PL/SQL является процедурным расширением языка SQL, то в PL/SQL есть все типы данных, которые имеются в диалекте Oracle SQL с некоторыми несущественными различиями. В дополнение к ним в PL/SQL есть и такие типы данных, которых нет в Oracle SQL.

В PL/SQL имеются скалярные и составные типы данных:

- данные скалярных типов состоят из одного неделимого (атомарного) значения (логические значения, числа, даты, строки);
- данные составных типов состоят из нескольких значений (записи и коллекции).

---

[1] Напомним, что в Oracle для значений типа `DATE` реализована следующая семантика операции вычитания: date1-date2 = число дней между этими датами.

Скалярные типы данных PL/SQL

Объявления типов данных PL/SQL находятся в пакете STANDARD, находящемся в схеме пользователя SYS:

```
package STANDARD AUTHID CURRENT_USER is

  type BOOLEAN is (FALSE, TRUE);

  type DATE is DATE_BASE;

  type NUMBER is NUMBER_BASE;
  subtype FLOAT is NUMBER;            -- NUMBER(126)
  subtype REAL is FLOAT;              -- FLOAT(63)
  subtype INTEGER is NUMBER(38,0);
  subtype INT is INTEGER;
  subtype DEC is DECIMAL;
  ...
  subtype BINARY_INTEGER is INTEGER range '-2147483647'..2147483647;
  subtype NATURAL is BINARY_INTEGER range 0..2147483647;
  ...
  type VARCHAR2 is NEW CHAR_BASE;
  subtype VARCHAR is VARCHAR2;
  subtype STRING is VARCHAR2;
  ...
```

Видно, что объявленные в пакете STANDARD типы данных PL/SQL либо соответствуют типам данных Oracle SQL (_BASE-типы), либо вводятся как их подтипы (subtype).

Отметим наличие типа данных BOOLEAN, которого нет в Oracle SQL[1]. Значения типа BOOLEAN можно, например, использовать в коде следующего вида:

```
l_amount_negative_flag BOOLEAN := amount<0;
IF l_amount_negative_flag THEN … END IF;
```

Существенным отличием типов данных PL/SQL и Oracle SQL является бо́льшая максимальная длина значений типов CHAR и VARCHAR2,

[1] В таблицах базы данных Oracle для представления логических значений обычно создают столбец целочисленного типа и хранят в нем 0 (FALSE) или 1 (TRUE).

предназначенных для представления строк фиксированной и переменной длины[1]:

- для VARCHAR2 в PL/SQL максимальная длина значений находится в диапазоне от 1 до 32 767 байт (в Oracle SQL до версии Oracle 12c максимальная длина VARCHAR2 была до 4000 байт, в Oracle 12c она была увеличена также до 32 767 байт);
- для CHAR в PL/SQL максимальная длина значений находится в диапазоне от 1 до 32 767 байт (в Oracle SQL до версии Oracle 12c максимальная длина CHAR была до 2000 байт, в Oracle 12c она была увеличена также до 32 767 байт).

Записи PL/SQL

Записи PL/SQL относятся к составным типам данных и определяются как наборы атрибутов, связанных определенными отношениями. Атрибуты записи могут быть как скалярных типов данных, так и других составных типов — другими записями и коллекциями.

Запись PL/SQL объявляется как пользовательский тип данных с помощью ключевого слова RECORD, в целом работа с записями PL/SQL похожа на работу с записями в языке Pascal или структурами в языке C:

```
DECLARE
  TYPE t_person IS RECORD
  (name    VARCHAR2 (100),
   secname VARCHAR2 (100),
   surname VARCHAR2 (100),
   born    DATE);
  l_person t_person;
BEGIN
  l_person.surname := 'Ильин';
  l_person.name    := 'Виктор';
  l_person.secname := 'Семенович';
  l_person.born    := TO_DATE('07.08.1969','dd.mm.yyyy');
  print(l_person);
END;
```

[1] Младшие версии Oracle (до 12c) еще будут эксплуатироваться десятилетиями.

Назначение записей PL/SQL:

- считывание в записи PL/SQL строк результирующих выборок SQL-запросов (при объявлении записей PL/SQL на основе таблиц и курсоров с помощью атрибута `%ROWTYPE`);
- объединение в одну структуру нескольких параметров процедур и функций (вместо большого числа параметров скалярных типов удобнее передавать в процедуры и функции один параметр составного типа).

Компактность и расширяемость исходного кода — основное преимущество от использования записей PL/SQL. Сравните два варианта вызова процедуры печати сведений о человеке — с одним параметром-записью PL/SQL и с несколькими параметрами скалярных типов данных:

`print(l_person)` и `print(l_name, l_secname, l_surname, l_born)`

Первый вариант вызова выглядит более компактным. Кроме того, если появится необходимость обрабатывать новые сведения о человеке, например, ИНН и СНИЛС, то для второго варианта во все вызовы процедуры `print` по всему коду понадобится дописать по два новых параметра. Если же передавать описание человека в виде записи PL/SQL, то потребуется только добавить новые атрибуты в объявление типа `t_person`. Вносить изменения в заголовок функции `print` и в ее вызовы по исходному коду не потребуется. Тем самым с помощью использования записей PL/SQL обеспечивается расширяемость исходного кода.

Приведем основные правила работы с записями PL/SQL:

- в определении атрибутов записей могут быть указаны ограничения `NOT NULL` и заданы значения атрибутов по умолчанию;
- присвоение записи `NULL` присваивает `NULL` всем ее атрибутам;
- чтобы сравнить две записи на равенство или неравенство нужно последовательно попарно сравнить значения всех атрибутов.

Так как записи PL/SQL похожи на строки таблиц, то особенно выпукло преимущества их использования для обеспечения компактности и расширяемости кода проявляются при выполнении предложений SQL в PL/SQL. Одна строка таблицы — одна запись PL/SQL. Строки таблиц «живут» в базе данных, записи PL/SQL «живут» в программах. Строку таблицы можно одной командой считать в за-

пись PL/SQL, запись PL/SQL можно одной командой вставить как строку таблицы, то есть обеспечивается движение данных в обоих направлениях. В PL/SQL также есть специальные языковые конструкции, которые позволяют перемещать между базой данных и программой PL/SQL не отдельные записи PL/SQL и строки таблиц, а их множества. И все это делается очень компактно — одной-двумя строками кода PL/SQL.

Объявление переменных с привязкой

Так как язык PL/SQL предназначен для обработки данных, которые находятся в таблицах базы данных Oracle, то в нем предусмотрена возможность объявления переменных с привязкой к схемам этих таблиц. Например, если какая-то переменная используется для считывания в нее значений столбца `surname` таблицы `person`, то логично было бы указать при объявлении этой переменной тип данных, совпадающий с типом данных столбца.

Существует два вида привязки переменных:

- скалярная привязка (с помощью атрибута `%TYPE` переменная объявляется с типом данных указанного столбца таблицы);
- привязка к записи (с помощью атрибута `%ROWTYPE` объявляется переменная-запись PL/SQL с атрибутами по числу столбцов указанной таблицы или курсора).

Рассмотрим пример. Пусть в базе данных имеется таблица `tab1` со столбцами `at1` типа `DATE` и `at2` типа `VARCHAR2(20)`. Тогда в коде PL/SQL можно объявить переменные следующим образом:

```
l_tab1 tab1%ROWTYPE;
l_at1  tab1.at1%TYPE;
```

Переменная `l_tab1` будет являться записью PL/SQL с двумя атрибутами `at1`, `at2`, типы данных которых будут такими же, как типы данных столбцов `at1`, `at2` таблицы `tab1`, то есть `DATE` и `VARCHAR2(20)` соответственно. Переменная `l_at1` будет иметь тип данных, такой же, как у столбца `at1`, то есть `date`.

Преимущества объявления переменных с привязкой:

- автоматически выполняется синхронизация со схемами таблиц;
- компактный расширяемый код для считывания строк результирующих выборок SQL-запросов без перечисления столбцов.

Автоматическая синхронизация объявлений переменных в программах PL/SQL и схем таблиц базы данных делает программы PL/SQL устойчивыми к возможным в будущем изменениям, таким как добавление, удаление или переименование столбцов таблиц, изменениям их типов данных. На практике такие изменения схем таблиц происходят довольно часто.

Приведем конкретный пример. В базе данных CRM-системы была таблица clients, в которой имелся столбец inn. На момент разработки системы клиентами могли быть только юридические лица, у которых длина ИНН составляет 10 символов. Со временем компания стала обслуживать и физических лиц, у которых длина ИНН 12 символов. Администратор базы данных изменил тип данных столбца inn таблицы clients с VARCHAR2(10) на VARCHAR2(12) и в таблице стал появляться строки с длинными ИНН. Так как в коде PL/SQL все переменные для работы с ИНН были объявлены как VARCHAR2(10), то при считывании из базы данных ИНН физических лиц в программах PL/SQL стали происходить ошибки. Если бы переменные для ИНН в свое время были объявлены с привязкой к столбцу inn с помощью атрибута %TYPE, то они автоматически «расширились» бы сами и ошибок на стадии выполнения не происходило бы.

Без перечисления столбцов результирующих выборок SQL-запросов пишется очень компактный код вида

```
l_person person%ROWTYPE;
SELECT * INTO l_person FROM person WHERE id=13243297;
print(l_person);
```

SQL-запрос выбирает все столбцы таблицы person, и у объявленной с помощью %ROWTYPE переменной l_person будет ровно столько же атрибутов, сколько столбцов у таблицы person, с такими же именами и типами данных, в том же порядке следования. Значения всех столбцов считываемой строки таблицы присвоятся соответствующим атрибутам записи PL/SQL. Если в таблице person в будущем появится новый столбец, он автоматически «подхватится» и SQL-запросом (SELECT *) и объявлением переменной l_person в программе PL/SQL. Никаких изменений в код вносить не потребуется, автоматическая перекомпиляция программы произойдет при первом обращении к ней.

Объявлять переменные как записи PL/SQL с помощью атрибута %ROWTYPE можно не только на основе какой-то одной таблицы, но и на

основе столбцов результирующих выборок произвольных SQL-запросов. Для этого записи PL/SQL объявляются на основе явных курсоров, рассматриваемых далее.

Структура программы PL/SQL

Структура блока

В PL/SQL, как и в большинстве процедурных языков программирования, наименьшей единицей группировки исходного кода является блок. Он представляет собой фрагмент кода, определяющий границы выполнения кода и области видимости для объявлений. Блоки могут вкладываться друг в друга.

Разделы блока PL/SQL

Блок PL/SQL состоит из четырех разделов:

- раздел заголовка;
- раздел объявлений;
- исполняемый раздел;
- раздел обработки исключений.

Разделов заголовка, объявлений и обработки исключений в блоке может не быть, обязательным является только исполняемый раздел.

Синтаксически блок PL/SQL выглядит следующим образом:

```
раздел заголовка
DECLARE
  раздел объявлений
BEGIN
  исполняемый раздел
EXCEPTION
  раздел обработки исключений
END;
```

В разделе заголовка указываются:

- тип блока (процедура, функция);
- имя блока (имя процедуры, функции);
- имена параметров, их типы данных и режимы передачи значений.

В разделе объявлений объявляются пользовательские типы данных, переменные и константы, которые потом используются в исполняемом разделе и разделе обработки исключений. В исполняемом разделе реализуется собственно логика программы. В вырожденном случае там может присутствовать только одна «пустая» команда NULL. Обработка исключений рассматривается далее в отдельном параграфе.

Ключевые слова BEGIN и END в языке PL/SQL являются операторными скобками, подобными символам { и } в других языках программирования и отмечают начало исполняемого раздела и конец блока. Каждая команда в PL/SQL должна завершаться точкой с запятой (символом ;).

Виды блоков PL/SQL

В PL/SQL есть два вида блоков:

- именованные блоки (с разделом заголовка);
- анонимные блоки (без раздела заголовка).

Именованные блоки в свою очередь тоже бывают двух видов:

- именованные блоки хранимых в базе данных программ (процедур, функций, пакетов и триггеров);
- именованные блоки в разделах объявлений других блоков (анонимных или именованных).

Хранимые программы (stored programs) являются объектами базы данных Oracle и создаются DDL-командой CREATE, после которой записывается именованный блок PL/SQL. Имя блока, указанное в разделе заголовка, будет являться именем объекта базы данных.

Анонимные блоки (anonymous blocks) раздела заголовка не имеют. Если блок не имеет раздела заголовка, то он не имеет и имени, которое в этом разделе указывается, поэтому такие блоки и называются анонимными.

Анонимные блоки либо вкладываются в другие блоки, либо хранятся в виде текстовых файлов-сценариев. В последнем случае анонимные блоки, как правило, используются для вызова хранимых программ или для автоматизации задач администрирования баз данных.

Анонимный блок-сценарий file1.sql	Вложенные анонимные блоки в именованном блоке хранимой программы

DECLARE
  i INTEGER;
  -- именованный блок процедуры proc1
  -- в разделе анонимного блока
  PROCEDURE proc1 IS
  BEGIN
  NULL;
  END;
BEGIN
  -- вызов процедуры proc1
  proc1;
END;
	-- именованный блок процедуры proc2
	CREATE PROCEDURE proc2 AS
	BEGIN
	  -- родительский анонимный блок,
	  -- вложенный в именованный proc2:
	  DECLARE
	  BEGIN
	  -- еще один анонимный блок
	  -- вложенный в родительский:
	  DECLARE
	  BEGIN
	  NULL;
	  END; -- конец вложенного блока
	  END; -- конец родительского блока
	END;-- конец именованного блока proc2

Комментарии

В любом месте исходного кода на PL/SQL могут быть комментарии, однострочные и многострочные.

Однострочные комментарии начинаются с двух дефисов (символы `--`). Весь текст после двух дефисов и до конца строки рассматривается как комментарий и игнорируется компилятором. Если два дефиса стоят в начале строки, то комментарием является вся строка.

Многострочный комментарий размещается между начальным (`/*`) и конечным (`*/`) ограничителями. Вложение многострочных комментариев друг в друга не допускается.

Тема комментирования исходного кода заслуживает отдельного рассмотрения. Автор книги является сторонником следующего принципа: «Комментируйте неочевидные участки кода. Не комментируйте очевидные».

О том, как следует комментировать код, есть несколько хороших статей, также эта тема подробно рассмотрена в книгах, посвященных выработке хорошего стиля программирования. Особое внимание следует уделить тому, чтобы тексты комментариев соответствовали актуальной версии кода. Довольно часто после внесения изменений в коде забывают их отразить в комментариях.

Приведем пример анонимного блока, в котором объявлены одна константа и две переменные, а в исполняемом разделе выполняются действия по вычислению натуральных логарифмов чисел 2 и 3.

```
/* Вычисление
     двух  логарифмов */
SQL> DECLARE
  2    header1 CONSTANT VARCHAR2(20) := 'Логарифм двух равен ';
  3    header2 CONSTANT VARCHAR2(20) := 'Логарифм трех равен ';
  4    arg INTEGER := 2;
  5  -- исполняемый раздел
  6  BEGIN
  7    DBMS_OUTPUT.PUT_LINE(header1||LN(arg));
  8    arg := arg+1;
  9    DBMS_OUTPUT.PUT_LINE(header2||LN(arg));
 10  END;
 11  /
Логарифм двух равен .693147180559945309417232121458176568078 2
Логарифм трех равен 1.098612288668109691395245236922525704 65
PL/SQL procedure successfully completed.
```

В разделе объявлений можно объявлять как переменные, так и константы (с помощью ключевого слова CONSTANT). Константа отличается от переменной тем, что ее значение нельзя изменять после объявления. Если указать константу в левой части оператора присваивания и т. п., то это будет определено как ошибка еще на этапе компиляции. Переменным присваивать значения можно в любом разделе, в том числе прямо при ее объявлении в разделе объявлений. По умолчанию переменная инициализируется «пустым» значением NULL.

Имена констант, переменных, пользовательских типов данных в грамматике PL/SQL называются идентификаторами. К идентификаторам предъявляются следующие требования:

- идентификатор должен состоять только из букв, цифр и символов _$#
- идентификатор должен начинаться с буквы;
- длина идентификатора должна быть до 30 символов;

- идентификатор не должен быть зарезервированным словом[1].

Примеры недопустимых идентификаторов:

- `2_name` (начинается не с буквы, правильно — `l_second_name`);
- `l_exchange_rate_on_current_date` (длина свыше 30 символов).

Рекомендуется блокам PL/SQL, пользовательским типам данных, переменным и константам давать имена, соответствующие некоторому соглашению об именовании.

В языке PL/SQL переменные, константы и пользовательские типы данных являются локальными для блока, в котором они объявлены. Когда выполнение блока будет завершено, все эти объекты внутри программы становятся недоступными. Можно сказать, что у каждого объявленного в программе PL/SQL элемента имеется некоторая область видимости — участок программы, в котором можно ссылаться на этот элемент (блок, в котором элемент объявлен, и все вложенные в него блоки).

Структуры управления вычислениями

Известно, что для реализации алгоритмов на процедурном языке программирования требуется наличие в нем трех следующих структур управления вычислениями, причем должна быть возможность вкладывать все структуры друг в друга произвольным образом:

- последовательность команд (выполнение команд согласно их упорядоченности);
- выбор (проверка условия и выполнение той или иной последовательности команд в зависимости от истинности или ложности условия);
- повторение (выполнение последовательности команд до тех пор, пока условие повторения принимает истинное значение).

---

[1] Список зарезервированных слов можно посмотреть в словаре-справочнике данных Oracle, выполнив запрос `SELECT * FROM v$reserved_words`.

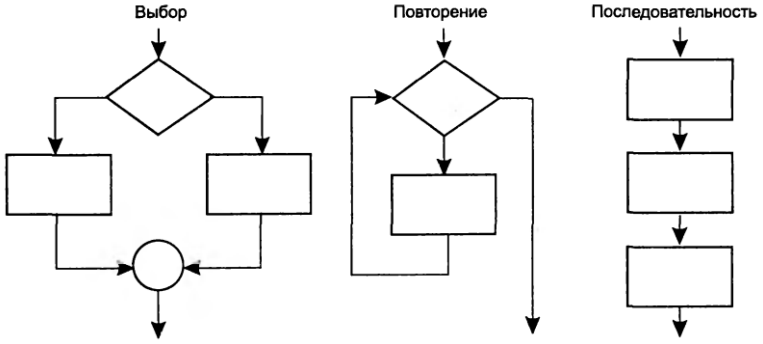

Рис. 1. Структуры управления вычислениями.

Команды, указанные в коде PL/SQL, выполняются последовательно. Такая схема называется потоком команд, то есть первая требуемая структура управления вычислениями (последовательность) в PL/SQL имеется. Рассмотрим языковые конструкции PL/SQL для выбора и повторения (условные команды, команды перехода и циклы).

Условные команды и команды перехода

В PL/SQL к условным командам относятся команды `IF` и `CASE`, переходы реализуются командами `GOTO` и `NULL`.

Условная команда IF

Условная команда `IF` позволяет проверить заданное логическое условие и, в зависимости от результатов проверки (`TRUE`, `FALSE` или `UNKNOWN`), выполнить различные ветви кода. В PL/SQL имеется три формы команды `IF`:

- `IF THEN END IF`
- `IF THEN ELSE END IF`
- `IF THEN ELSIF ELSE END IF`

Границы действия команды `IF` определяются закрывающими ключевыми словами `END IF`. Альтернативная последовательность команд следует после ключевого слова `ELSE`, для расширения структуры

ветвления дополнительно можно задать дополнительное условие после ключевого слова ELSIF.

Команда IF имеет следующий синтаксис:

```
IF условие 1 THEN
  команды 1;  -- ветвь 1
ELSIF условие 2 THEN
  команды 2;  -- ветвь 2
ELSE
  команды n;  -- альтернативная последовательность команд (ELSE-ветвь)
END IF;
```

Команды первой ветви кода выполняются только тогда, когда проверка условия определяет его истинность (TRUE). Если же проверка условия определяет FALSE или UNKNOWN (например, при сравнении с NULL), то выполняются команды ELSE-ветви.

Приведем пример программы, которая выводит сообщение о классе излучения в зависимости от значения вводимого параметра длины волны (длина волны предполагается заданной в микронах).

```
SQL> DECLARE
  2   lamda NUMBER;      -- Длина волны
  3   text1 VARCHAR2(30) := 'Инфракрасное излучение';
  4   text2 VARCHAR2(30) := 'Видимый свет';
  5   text3 VARCHAR2(30) := 'Ультрафиолет';
  6 -- исполняемый раздел
  7 BEGIN
  8   lamda := &Input_Data;
  9   DBMS_OUTPUT.PUT_LINE('');
 10   IF (lamda > 0.65)
 11     THEN DBMS_OUTPUT.PUT_LINE(text1);
 12   ELSIF (lamda < 0.41)
 13     THEN DBMS_OUTPUT.PUT_LINE(text3);
 14   ELSE
 15     DBMS_OUTPUT.PUT_LINE(text2);
 16   END IF;
 17 END;
/
Enter value for input_data: 0.33
old   8: lamda := &Input_Data;
new   8: lamda := 0.33;
Ультрафиолет
PL/SQL procedure successfully completed.
```

При сложной логике ветвления и проверке многочисленных условий рекомендуется вместо вложенных команд IF использовать условную

команду CASE, так как с ней получается более понятный и компактный код.

Условная команда CASE

Команда CASE имеет две разновидности:

- простая команда CASE, которая связывает одну или несколько последовательностей команд PL/SQL с некоторыми значениями (выполняемая последовательность команд выбирается при совпадении результата вычисления заданного выражения со значением, связанным с этой последовательностью команд);
- поисковая команда CASE, которая выбирает для выполнения последовательность команд в зависимости от результатов проверки списка логических условий (выполняется последовательность команд, связанная с первым логическим условием в списке, результат проверки которого оказался равным TRUE).

Несмотря на громоздкое описание, работать с командой CASE обеих разновидностей просто и удобно.

Простая команда CASE имеет следующий синтаксис[1]:

```
CASE выражение
  WHEN результат 1 THEN
    последовательность команд 1;
  WHEN результат 2 THEN
    последовательность команд 2;
…
  ELSE
    альтернативная последовательность команд;
END CASE;
```

Простая команда CASE обычно используется для избавления от многочисленных команд IF и конструкций ELSE в них путем формирования хорошо структурированных ветвей кода в зависимости от списка значений, которые может принимать некоторая управляющая

[1] Если в команде CASE отсутствует конструкция ELSE, и на этапе выполнения к вычисленному значению выражения не подошел ни один результат из конструкций WHEN, то произойдет ошибка этапа выполнения.

переменная. Приведем пример поиска слова на русском языке по английскому аналогу:

```
SQL> DECLARE
  2    english_termin VARCHAR2(20);
  3    text1          VARCHAR2(30) := 'Инфракрасное излучение';
  4    text2          VARCHAR2(30) := 'Видимый свет';
  5    text3          VARCHAR2(30) := 'Ультрафиолет';
  6    text4          VARCHAR2(30) := 'Неизвестный термин';
  7  BEGIN
  8    english_termin := &Input_Data;
  9    CASE english_termin
 10      WHEN 'Infrared radiation' THEN DBMS_OUTPUT.PUT_LINE(text1);
 11      WHEN 'Visible light' THEN DBMS_OUTPUT.PUT_LINE(text2);
 12      WHEN 'Ultraviolet' THEN DBMS_OUTPUT.PUT_LINE(text3);
 13      ELSE DBMS_OUTPUT.PUT_LINE(text4);
 14    END CASE;
 15  END;
 16  /

Enter value for input_data: 'Ultraviolet'
old   8:   english_termin := &Input_Data;
new   8:   english_termin := 'Ultraviolet';
Ультрафиолет

PL/SQL procedure successfully completed.
```

Поисковая команда CASE **имеет следующий синтаксис:**

```
CASE
  WHEN верно логическое условие 1 THEN
    последовательность команд 1;
  WHEN верно логическое условие 2 THEN
    последовательность команд 2;
…
  ELSE
    альтернативная последовательность команд;
END CASE;
```

Перепишем пример определения источника излучения с использованием поисковой команды CASE **вместо команды** IF **(сравните с предыдущей версией):**

```
SQL> DECLARE
  2    lamda NUMBER;
  3    text1 VARCHAR2(30) := 'Инфракрасное излучение';
  4    text2 VARCHAR2(30) := 'Видимый свет';
  5    text3 VARCHAR2(30) := 'Ультрафиолет';
  6  BEGIN
  7    lamda := &Input_Data;
```

```
  8    CASE
  9      WHEN (lamda > 0.65)
 10        THEN DBMS_OUTPUT.PUT_LINE(text1);
 11      WHEN (Lamda < 0.41)
 12        THEN DBMS_OUTPUT.PUT_LINE(text3);
 13      ELSE
 14        DBMS_OUTPUT.PUT_LINE(text2);
 15    END CASE;
 16  END;
 17  /
Enter value for input_data: 0.50
old   7:   lamda := &Input_Data;
new   7:   lamda := 0.50;
Видимый свет

PL/SQL procedure successfully completed.
```

Помимо условной команды `CASE`**, в PL/SQL есть выражение** `CASE`**, которое последовательно вычисляет логические выражения из заданного списка, пока одно из них не станет истинным, а затем возвращает результат связанного с ним выражения. В этом заключается отличие выражения** `CASE` **от команды** `CASE` **— команда** `CASE` **управляет потоком других команд, а выражение** `CASE` **вычисляется и его значение может быть присвоено переменным, использовано в других выражениях и т. д.**

```
DECLARE
  lamda  NUMBER := 0.50;
  text1  VARCHAR2(30) := 'Инфракрасное излучение';
  text2  VARCHAR2(30) := 'Видимый свет';
  text3  VARCHAR2(30) := 'Ультрафиолет';
  answer VARCHAR2(30);
BEGIN
  answer :=  CASE WHEN (lamda > 0.65) THEN text1
                  WHEN (Lamda < 0.41) THEN text3
                  ELSE text2
             END;
  DBMS_OUTPUT.PUT_LINE(answer);
END;
```

Команда перехода GOTO

Команда перехода `GOTO` **позволяет осуществить переход по метке, присутствующей в коде PL/SQL. С помощью уникального идентификатора, заключенного в двойные угловые скобки (метки), можно пометить любую часть исполняемого блока PL/SQL для перехода в это место.**

```
SQL> DECLARE
  2    s NUMBER := 1;
  3  BEGIN
  4    IF s = 1 THEN
  5      GOTO mybranch;  -- переход по метке mybranch
  6    ELSE
  7      s := 1;
  8    END IF;
  9    <<mybranch>>      -- установка метки mybranch
 10    NULL;
 11  END;
 12  /
PL/SQL procedure successfully completed.
```

Команда GOTO в языках программирования является объектом критики, поскольку чрезмерное ее применение приводит к созданию нечитаемого «спагетти-кода». Впервые эта точка зрения была отражена в статье Эдсгера Дейкстры «Доводы против команды GOTO», в которой утверждалось, что квалификация программистов обратно зависит от частоты использования команды GOTO в их программах. Многие преподаватели не принимают написанные студентами программы с командами GOTO по той причине, что наличие GOTO свидетельствует о неумении правильно структурировать исходный код.

Команда NULL

Команда NULL («пустая» команда) обычно используется как «заглушка» в месте, где надо написать какую-нибудь команду, потому что ничего не написать там нельзя по требованиям синтаксиса PL/SQL. Потом, по мере появления определенности, «заглушка» заменяется на функциональный код:

```
CASE sex
  WHEN 'M' THEN
    sex_decoded := 'male';
  WHEN 'F' THEN
    sex_decoded := 'female';
  ELSE
    NULL; -- toDo: write code for exception sex not in list {F,M} ;))
END CASE;
```

Также команда NULL используется при обработке исключений, когда обработка какого-нибудь исключения заключается в отсутствии каких-либо действий (ничегонеделании). Такая практика «замалчивания» исключений обычно не приветствуется, так как она приводит к

сложно выявляемым проблемам и неожиданным результатам работы программ.

Циклы

В языке PL/SQL имеется три вида циклов:

- простой цикл, который начинается с ключевого слова LOOP и завершается командой END LOOP;
- цикл WHILE с предусловием, который позволяет выполнить одну и ту же последовательность команд, пока проверяемое условие истинно;
- цикл FOR со счетчиком.

Простой цикл

Простой цикл рассмотрим на примере определения числа, факториал которого является наименьшим числом, впервые превышающим заданную константу (1 000 000 000).

```
SQL> DECLARE
  2    arg NUMBER;                 -- Переменная для вычисления факториала
  3    i NUMBER;                   -- Переменная-счетчик
  4    limit NUMBER := 1000000000; -- Граница
  5    text1 VARCHAR2(80) := 'n! числа, впервые превышающий 1000000000';
  6
  7  BEGIN
  8    i := 0;
  9    arg := 1;
 10    LOOP
 11      EXIT WHEN arg > limit;
 12      arg := arg*(i+1);
 13      i := i + 1;
 14    END LOOP;
 15    DBMS_OUTPUT.PUT_LINE(text1);
 16    DBMS_OUTPUT.PUT_LINE(TO_CHAR(arg));
 17    DBMS_OUTPUT.PUT_LINE('Искомое число = '||TO_CHAR(i));
 18  END;
/

n! числа, впервые превышающий 1000000000
6227020800
Искомое число = 13
PL/SQL procedure successfully completed.
```

Из любого цикла в PL/SQL можно выйти командой EXIT с указанием логического условия выхода. В основном команда EXIT используется

в простых циклах, потому что в циклах FOR и WHILE и так явно указываются условия окончания цикла, а иметь в коде больше одного условия окончания для цикла является плохим стилем программирования.

Если происходит зацикливание (выполнение бесконечного цикла без выхода из него), то программа PL/SQL «уходит в себя» («повисает»). Для прекращения выполнения такой программы в SQL*Plus следует нажать на клавиатуре комбинацию клавиш Ctrl+C:

```
SQL> BEGIN LOOP NULL; END LOOP; END;
  2  /
^C
```

Цикл WHILE

Цикл WHILE с предусловием позволяет выполнить одну и ту же последовательность команд PL/SQL пока истинно проверяемое предусловие.

С помощью цикла WHILE найдем число, факториал которого является наименьшим числом, впервые превышающим 1 000 000 000 000:

```
SQL> DECLARE
  2    arg NUMBER;                    -- Переменная для вычисления факториала
  3    i NUMBER;                      -- Переменная-счетчик
  4    limit NUMBER := 1000000000000;-- Граница
  5    text1 VARCHAR2(80):='n! числа, впервые превышающий 1000000000000;
  6
  7  BEGIN
  8    i := 0;
  9    arg := i;
 10    WHILE arg < 1000000000000 LOOP
 11      arg := arg*(i+1);
 12      i := i + 1;
 13    END LOOP;
 14    DBMS_OUTPUT.PUT_LINE(text1);
 15    DBMS_OUTPUT.PUT_LINE(TO_CHAR(arg));
 16    DBMS_OUTPUT.PUT_LINE('Искомое число = '||TO_CHAR(i));
 17  END;
/
n! числа,впервые превышающий 1000000000000
1307674368000
Искомое число = 15
PL/SQL procedure successfully completed.
```

Отметим, что если условие цикла WHILE изначально ложно (FALSE), то цикл не выполнится ни разу.

Цикл FOR

Цикл FOR («цикл со счетчиком»), используется в том случае, когда известно, сколько раз нужно выполнить итерацию цикла. Приведем пример вычисления факториала заданного числа.

```
SQL> DECLARE
  2    arg NUMBER         := 1;
  3    n   NUMBER         := 20;
  4    text1 VARCHAR2(30) := 'Факториал числа '||n||' = ';
  5  BEGIN
  6    FOR i IN 1..n LOOP
  7      arg := arg*i;
  8    END LOOP;
  9    DBMS_OUTPUT.PUT_LINE(text1||TO_CHAR(arg));
 10  END;
/
Факториал числа 20 = 2432902008176640000
PL/SQL procedure successfully completed.
```

Обратите внимание, что счетчик — управляющую переменную цикла (в данном случае i) объявлять в разделе объявлений не нужно, она объявляется автоматически с областью видимости между ключевыми словами LOOP и END LOOP.

При рассмотрении циклов FOR обычно возникают два вопроса:

- есть ли возможность сделать так, чтобы значения счетчика цикла не возрастали, а уменьшались?
- есть ли возможность нетривиальных, то есть не на единицу, приращений шага счетчика цикла?

Цикл с ключевым словом REVERSE

```
SQL> BEGIN
  2    FOR i IN REVERSE 1..5 LOOP
  3      DBMS_OUTPUT.PUT_LINE(i);
  4    END LOOP;
  5  END;
  6  /
5
4
3
2
1

PL/SQL procedure successfully completed.
```

Цикл со счетчиком, кратным 10

```
SQL> BEGIN
  2    FOR i IN 1..20 LOOP
  3      IF MOD(i,10)=0 THEN
  4        -- тело цикла
  5        DBMS_OUTPUT.PUT_LINE(i);
  6      END IF;
  7    END LOOP;
  8  END;
  9  /
10
20

PL/SQL procedure successfully completed.
```

На оба вопроса ответы положительные — такие возможности в языке PL/SQL имеются. Для обратного цикла в конструкции FOR LOOP следует указать ключевое слово REVERSE. Для нетривиальных приращений нужно внутри цикла с помощью команды IF просто пропускать шаги с ненужными значениями счетчика. Счетчик цикла всегда изменяется на единицу, просто на некоторых шагах цикла ничего делать не нужно.

Команда CONTINUE

Команда CONTINUE появилась в версии Oracle 11g. При истинности условия, записанного в конструкции WHEN команды CONTINUE, выполнение текущей итерации цикла прекращается и происходит переход на начало следующей итерации.

Благодаря команде CONTINUE можно, например, вынести проверки в начало цикла. Перепишем приведенный выше пример с нетривиальным приращением счетчика цикла FOR без помещения кода обработки в условную команду IF:

```
BEGIN
  FOR i IN 1..20 LOOP
    CONTINUE WHEN MOD(i,10)!=0;
    DBMS_OUTPUT.PUT_LINE(i);
  END LOOP;
END;
```

Работа с коллекциями

Коллекции относятся к составным типам данных PL/SQL. Так как основная операция с коллекцией — это перебор и обработка всех ее элементов, то естественно познакомиться с особенностями работы с коллекциями после того, как рассмотрены циклы.

Виды и свойства коллекций

Коллекция называется ограниченной, если заранее определены границы возможных значений индексов ее элементов. В противном случае коллекция называется неограниченной.

В PL/SQL есть три вида коллекций:

- ассоциативные массивы (associative array) — неограниченные коллекции, объявляемые только в программах PL/SQL (поэтому

в литературе иногда эти коллекции называются таблицами PL/SQL);

- вложенные таблицы (nested tables) — неограниченные коллекции, типы данных на основе которых могут создаваться как объекты баз данных и объявляться в программах PL/SQL;
- массивы переменной длины (variable-size array, VARRAY) — ограниченные коллекции, типы данных на основе которых могут создаваться как объекты баз данных с помощью DDL-команд языка SQL и объявляться в программах PL/SQL.

Коллекция называется плотной, если все ее элементы, от первого до последнего, определены и им присвоены некоторые значения, включая NULL. Если же у коллекции в диапазоне индексов есть пропуски (какие-то элементы коллекции отсутствуют), то коллекция называется разреженной[1]. Массивы VARRAY всегда являются плотными. Вложенные таблицы первоначально всегда плотные, но по мере удаления некоторых элементов становятся разреженными. Таблицы PL/SQL могут быть разреженными.

Вне зависимости от вида коллекции, все ее элементы будут одного типа данных, то есть коллекции PL/SQL являются однородными.

Доступ к элементам коллекций PL/SQL всех трех видов осуществляется по их целочисленным индексам. Таблицы PL/SQL кроме чисел также могут индексироваться символьными строками.

Работа с таблицей PL/SQL с помощью встроенных методов

Для иллюстрации техники работы с таблицей PL/SQL покажем, как ее объявить, затем создадим два элемента таблицы и переберем их в цикле:

```
SQL> DECLARE
  2    TYPE t_job       IS RECORD (position VARCHAR2(100),salary INTEGER);
  3    TYPE t_job_table IS TABLE OF t_job INDEX BY PLS_INTEGER;
  4    TYPE t_person    IS RECORD (surname VARCHAR2(30),jobs t_job_table);
  5    l_person         t_person;
  6    l_job            t_job;
  7    l_job_table      t_job_table;
  8    l_row_index      PLS_INTEGER;
```

[1] Не «разрЯженной», а именно «разрЕженной» — от «проредить», «редко».

```
 9  BEGIN
10    l_person.surname := 'Ильинский К.В.';
11
12    l_job := NULL;
13    l_job.position   := 'инженер';
14    l_job.salary     := 50000;
15    l_job_table(9)   := l_job;
16
17    l_job := NULL;
18    l_job.position   := 'старший инженер';
19    l_job.salary     := 60000;
20    l_job_table(267) := l_job;
21
22    l_person.jobs    := l_job_table;
23
24    DBMS_OUTPUT.PUT_LINE('Сотрудник: '||l_person.surname);
25    l_row_index      := l_person.jobs.first();
26    WHILE l_row_index IS NOT NULL LOOP
27      DBMS_OUTPUT.PUT_LINE(l_person.jobs(l_row_index).position
28                    ||' ('||l_person.jobs(l_row_index).salary||' руб.)');
29      l_row_index    := l_person.jobs.next(l_row_index);
30    END LOOP;
31
32  END;
33  /
Сотрудник: Ильинский К.В.
инженер (50000 руб.)
старший инженер (60000 руб.)
PL/SQL procedure successfully completed.
```

Таблицы PL/SQL являются разреженными. Чтобы подчеркнуть это, в примере выше специально использованы случайно выбранные индексы 9 и 267, а не 1 и 2. Для перебора таблицы PL/SQL использованы ее встроенные методы FIRST и NEXT.

Коллекции PL/SQL имеют восемь встроенных методов.

Таблица 1. Встроенные методы коллекций PL/SQL.

Метод коллекции	Описание метода
COUNT (функция)	возвращает текущее число элементов в коллекции
DELETE (процедура)	удаляет из коллекции один или несколько элементов
EXISTS (функция)	определяет, существует ли в коллекции заданный элемент
EXTEND (процедура)	увеличивает количество элементов во вложенной таблице или массиве переменной длины

FIRST, LAST (функции)	возвращают индексы первого (FIRST) и последнего (LAST) элемента в коллекции
LIMIT (функция)	возвращает максимальное количество элементов в массиве переменной длины
PRIOR, NEXT (функции)	возвращают индексы элементов, предшествующих заданному (PRIOR) и следующему за ним (NEXT).
TRIM (процедура)	удаляет элементы, начиная с конца коллекции

Рекомендуется перебор элементов коллекций осуществлять с помощью методов FIRST и NEXT, а не с помощью циклов со счетчиком FOR, исходя из ожидаемой плотности коллекции. Цикл FOR перебирает весь заданный диапазон индексов подряд, что может привести к ошибке — обращению к отсутствующему элементу. Метод NEXT перемещается по индексам только «живых» элементов и ошибок из-за пропусков в нумерации не будет.

Индексы-строки таблиц PL/SQL

В версии Oracle 9i появилась возможность использовать для индексирования таблиц PL/SQL символьные строки. Это очень удобно, например, для работы со справочниками, в которых и коды и термины являются строками.

Рассмотрим пример.

```
SQL> DECLARE
  2    TYPE t_tab IS TABLE OF VARCHAR2(100) INDEX BY VARCHAR2(2);
  3    l_tab t_tab;
  4    l_code varchar2(3) := 'MD';
  5  BEGIN
  6    -- заполняем таблицу PL/SQL
  7    l_tab('UA') := 'Украина';
  8    l_tab('MD') := 'Молдавия';
  9    -- работаем с таблицей PL/SQL
 10    l_code := 'MD'
 11    DBMS_OUTPUT.PUT_LINE('1) Термин для '||l_code||' - '||l_tab(l_code));
 12    l_code := 'UA'
 13    IF l_tab.EXISTS('UA') THEN
 14      DBMS_OUTPUT.PUT_LINE('2) Код '||l_code||' есть в справочнике');
 15    END IF;
 16  END;
 17  /
1) Термин для MD - Молдавия
2) Код UA есть в справочнике
PL/SQL procedure successfully completed.
```

Массивы переменной длины и вложенные таблицы

Типы данных на основе вложенных таблиц и массивов переменной длины в основном создаются как объекты баз данных и используются в объектно-реляционных расширениях Oracle. Соответственно, для работы со считываемыми из баз данных массивами и вложенными таблицами в программах PL/SQL следует использовать переменные таких же типов данных.

Рассмотрим объектные расширения Oracle и работу с ними в PL/SQL на следующем примере.

Пусть есть таблица `students` со сведениями о студентах, у которой первые три столбца имеют скалярные типы данных, а столбцы `course_works` (курсовые работы) и `elective_courses` (факультативы) объявлены как массив переменной длины и вложенная таблица.

ID	sirname	name	course_works (VARRAY(6))	elective_courses (nested table)
18	Ильин	Виктор	{4; 4; NULL; 5; 5}	Оптимизация баз данных
				Теория надежности
				...
19	Варин	Павел	{4; 5; 5; 5,NULL,NULL}	Нечеткие модели
				...
20	Лунь	Михаил	{5; 5; 5; 5; 5; 5}	Нечеткие модели
				...
...	...	...	...	...

Считаем, что студенты учатся максимум 6 лет (могут меньше) и на каждом курсе может быть только одна курсовая работа (на каких-то курсах курсовых работ может не быть). Из сказанного следует, что

- больше 6 курсовых работ точно быть не может;
- если оценки за курсовые работы выписать в виде упорядоченного множества (списка), то порядковый номер оценки будет соответствовать курсу обучения (для курсов, на которых не было курсовых работ, следует на эти места поместить значения `NULL`).

Массивы переменной длины как раз и предназначены для представления упорядоченных множеств (списков) с заданным ограничением на максимальное число элементов. На физическом уровне в базах данных Oracle такие массивы хранятся в строках таблицы, рядом со значениями скалярных типов.

Что же касается факультативов, то заранее известной верхней оценки их числа для одного студента нет и обеспечить упорядочение их названий по какому-то правилу не требуется. В этих условиях для хранения данных о факультативах целесообразно использовать вложенные таблицы — в ячейку студента Ильина вкладывается одностолбцовая таблица со списком прослушанных им факультативов, в ячейку студента Варина вкладывается другая таблица факультативов и так далее.

```
SQL> CREATE TYPE t_course_works AS VARRAY(6) OF INTEGER;
  2  /
Type created.

SQL> CREATE TYPE t_elective_courses AS TABLE OF VARCHAR2(100);
  2  /
Type created.

SQL> CREATE TABLE students(id INTEGER,
  2                        surname          VARCHAR(100),
  3                        name             VARCHAR(100),
  4                        course_works     t_course_works,
  5                        elective_courses t_elective_courses)
  6  NESTED TABLE elective_courses STORE AS elective_courses_tab;
Table created.

SQL> INSERT INTO students VALUES(18,'Ильин','Виктор',
  2                          t_course_works(4,4,NULL,5,5),
  3                          t_elective_courses('Оптимизация баз данных',
  4                                             'Теория надежности'));
1 row created.

SQL> SET FEEDBACK ON
SQL> SELECT * FROM students;

ID    SURNAME   NAME       COURSE_WORKS
---   --------  ---------  ------------------------------
18    Ильин     Виктор     T_COURSE_WORKS(4, 4, NULL, 5, 5)

ELECTIVE_COURSES
------------------------------
T_ELECTIVE_COURSES('Оптимизация баз данных', 'Теория надежности')
1 row selected.
```

На физическом уровне в базе данных для столбца `elective_courses` **будет неявно создана вспомогательная таблица (мы дали ей имя** `elective_courses_tab`**), в которой будут храниться все строки всех вложенных таблиц столбца** `elective_courses`**. Эти строки будут ссылаться на строки основной таблицы** `students`**, то есть фактически с**

помощью основной и вспомогательной таблиц и механизма ключей будет классическим способом моделироваться отношение «один ко многим» между студентами и факультативами. Рассмотрим теперь, как с массивами VARRAY и вложенными таблицами работают в коде PL/SQL. Напишем программу, которая выводит сведения о студенте, его оценки за курсовые работы на младших[1] и старших курсах отдельно, а также о список прослушанных студентом факультативов.

```
SQL> DECLARE
  2    l_surname          students.surname%TYPE;
  3    l_course_works     t_course_works;
  4    l_elective_courses t_elective_courses;
  5    l_row_index        PLS_INTEGER;
  6    l_student_id       students.id%TYPE := 18;
  7  BEGIN
  8
  9    SELECT surname,course_works,elective_courses
 10    INTO l_surname,l_course_works,l_elective_courses
 11    FROM students WHERE id=l_student_id;
 12
 13    DBMS_OUTPUT.PUT_LINE('Студент: '||l_surname);
 14
 15    IF l_course_works.EXISTS(1) or l_course_works.EXISTS(2) THEN
 16      DBMS_OUTPUT.PUT_LINE('Курсовые на младших курсах:');
 17    ELSE
 18      DBMS_OUTPUT.PUT_LINE('Курсовые на младших курсах отсутствуют')
 19    END IF;
 20
 21    FOR i in 1..2 LOOP
 22      IF l_course_works.EXISTS(i) THEN
 23        DBMS_OUTPUT.PUT_LINE('   Курсовая на '||i||' курсе: ' ||
 24                             ' оценка '||l_course_works(i));
 25      END IF;
 26    END LOOP;
 27
 28    DBMS_OUTPUT.PUT_LINE('Курсовые на старших курсах:');
 29
 30    l_row_index      := l_course_works.NEXT(2);
 31    WHILE l_row_index IS NOT NULL LOOP
 32      DBMS_OUTPUT.PUT_LINE('  Курсовая на '||l_row_index
 33                       ||' курсе: оценка ' ||l_course_works(l_row_index));
 34      l_row_index    := l_course_works.NEXT(l_row_index);
 35    END LOOP;
 36
```

[1] Младшими являются 1 и 2 курсы.

```
 37    DBMS_OUTPUT.PUT_LINE('Факультативы (всего '
 38                         ||l_elective_courses.COUNT()||'):');
 39
 40    l_row_index      := l_elective_courses.FIRST();
 41    WHILE l_row_index IS NOT NULL LOOP
 42      DBMS_OUTPUT.PUT_LINE('    ' ||l_elective_courses(l_row_index));
 43      l_row_index    := l_elective_courses.NEXT(l_row_index);
 44    END LOOP;
 45
 46  END;
 47  /
Студент: Ильин
Курсовые на младших курсах:
   Курсовая на 1 курсе:  оценка 4
   Курсовая на 2 курсе:  оценка 4
Курсовые на старших курсах:
   Курсовая на 3 курсе:
   Курсовая на 4 курсе: оценка 5
   Курсовая на 5 курсе: оценка 5
Факультативы (всего 2):
   Оптимизация баз данных
   Теория надежности
PL/SQL procedure successfully completed.
```

Чаще всего в программах PL/SQL используются таблицы PL/SQL, поскольку считается, что с ними проще всего работать. Если же у программиста есть свобода выбора видов используемых коллекций, то для каждого конкретного случая следует учитывать несколько факторов, рассмотренных в литературе по PL/SQL.

Обработка исключений

Распространено мнение, что только половина профессионально написанного исходного кода реализует собственно функциональность программы. Остальной код — это ведение журнала программы, сохранение отладочной информации и обработка всевозможных ошибок.

Понятие исключения

Исключением (exception) в PL/SQL называется ситуация, которая не должна возникать при нормальном выполнении программы PL/SQL.

Существует два типа исключений PL/SQL:

- системные исключения (run-time system exceptions), которые автоматически инициируются виртуальной машиной PL/SQL при возникновении программных ошибок этапа выполнения;
- пользовательские исключения (user-defined exceptions), объявляемые программистом в коде PL/SQL и используемые при реализации бизнес-логики.

Программной ошибкой этапа выполнения (run-time program error) называется ситуация, когда наблюдается неожиданное поведение программы, затрудняющее или делающее невозможным достижение целей пользователя. Примерами программных ошибок могут служить попытки деления на ноль, ошибки преобразования символов в числа, ошибки выполнения предложений SQL.

Пользовательские исключения инициируются в программах PL/SQL в том случае, когда на прикладном уровне возникли отклонения от стандартного процесса обработки данных. Например, при обработке поступивших данных встретился чек с отрицательной суммой покупки или не в рублях. С точки зрения правил бизнес-логики это такая же ошибка, как и деление на ноль с точки зрения правил арифметики. Для попытки деления на ноль в ходе выполнения программы системное исключение будет автоматически инициировано виртуальной машиной PL/SQL, потому что она «знает» правила арифметики. Для поступающих ошибочных платежей инициировать пользовательское исключение должен в своем коде программист PL/SQL, потому что он знает правила бизнес-логики вида «Платежи принимаются только в рублях, на положительные суммы с точностью до копеек», «Платежи принимаются только для открытой смены контрольно-кассовой машины (ККМ)» и так далее.

Таким образом, основное различие системных и пользовательских исключений заключается в том, что они инициируются по-разному. Системное исключение автоматически инициируется виртуальной машиной, происходит неожиданно и обычно его появление говорит о том, что скоро придется решать проблемы самого разного вида. С пользовательскими исключениями все гораздо спокойнее — сами исключения, их инициирование специальными командами PL/SQL в коде и штатная обработка для выправления положения заранее предусматриваются программистом при проектировании.

Правила работы с исключениями:

- пользовательские исключения объявляются в разделах объявлений блоков PL/SQL и имеют имена;
- системные исключения имен не имеют, они характеризуются номером ошибки;
- имеется возможность объявить пользовательское исключение и с помощью директивы компилятору связать его с некоторым номером ошибки.

Несколько исключений для часто возникающих в программах PL/SQL ошибок объявлено во встроенном пакете STANDARD с привязкой к соответствующим номерам ошибок. Эти исключения называются предопределенными исключениями PL/SQL (predefined exception) и их можно использовать в любых программах PL/SQL без дополнительных объявлений.

Таблица 2. Предопределенные исключения PL/SQL.

Исключение	Описание исключения (номер ошибки)
INVALID_CURSOR	ссылка на несуществующий курсор (ORA-01001)
NO_DATA_FOUND	не найдены данные командой SELECT INTO (ORA-01403)
DUP_VAL_ON_INDEX	попытка вставить в столбец с ограничением на уникальность значение-дубликат (ORA-00001)
TOO_MANY_ROWS	команда SELECT INTO возвращает более одной строки (ORA-01422)
VALUE_ERROR	арифметическая ошибка, ошибка преобразования или усечения чисел и строк (ORA-06502)
INVALID_NUMBER	ошибка преобразования строки в число (ORA-01722)
PROGRAM_ERROR	внутренняя ошибка PL/SQL (ORA-06501)
ZERO_DIVIDE	попытка деления на ноль (ORA-01476)

Схема обработки исключений в Java

В языке программирования Java при описании работы с исключениями используется бейсбольная терминология. При возникновении исключения бросается (throws) объект-исключение. Этот объект как бейсбольный мяч пролетает через исходный код, появившись сначала в том методе, где произошло исключение. В одном или нескольких местах кода объект-исключение пытаются (try) поймать (catch) и

обработать. Причем исключение можно обработать в одном месте кода полностью, а можно обработать исключение частично, выбросить его из обработчика снова, поймать в другом месте и обрабатывать дальше.

Приведем пример кода на Java с попыткой поймать два исключения — связанные с ошибками арифметических вычислений и нарушением правил работы с массивами (выход индекса массива за границы диапазона):

```
try{
...
}
catch(ArithmeticException ae){
     System.out.println("From Arithm.Exc. catch: "+ae);
   }
   catch(ArraylndexOutOfBoundsException arre){
     System.out.println("From Array.Exc.catch: "+arre);
   }
}
```

Схема обработки исключений в PL/SQL

Работа с исключениями в PL/SQL очень похожа на то, как это делается в Java.

Для обработки исключений предназначен последний раздел блока PL/SQL — раздел обработки исключений. Этот последний раздел блока после ключевого слова EXCEPTION похож на то, что в Java указывается после ключевого слова catch. Перед обсуждением правил обработки исключений приведем небольшой пример с комментариями в коде.

```
DECLARE
  a INTEGER;
BEGIN

  a := 1;
  a := 2/0; -- бросается предопределенное исключение ZERO_DIVIDE
  a := 3;   -- над этой командой пролетает, команда не выполняется
  a := 4;   -- над этой командой тоже пролетает, команда не выполняется

-- управление передается в раздел обработки исключений,
-- начинаем «примерку» обработчиков
EXCEPTION

  -- не подходит по имени ловимого исключения к прилетевшему ZERO_DIVIDE
  WHEN PROGRAM_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('Программная ошибка');
```

```
  -- оба имени ловимых исключений не подходят к прилетевшему ZERO_DIVIDE
  WHEN INVALID_NUMBER OR VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('Ошибка работы с числами и строками');

  -- подходит по имени к ZERO_DIVIDE (поймали), заходим внутрь обработчика
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Ошибка деления на ноль');

  -- OTHERS ловит все, что не поймали другие до него,
  -- но сюда в этом случае «примерка» не дошла, раньше поймали
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('При выполнении произошла ошибка '||SQLERRM);

END;
```

При инициировании исключения в блоке PL/SQL выполнение потока команд блока прекращается, и управление передается в раздел обработки исключений этого блока, если такой раздел есть, или в родительский блок, если раздела обработки исключений у блока нет.

Сразу после инициирования исключение получает статус «не обработано», и можно сказать, что с этим статусом исключение бросается и летит над кодом программы. Летит оно именно над кодом, пропуская все команды исполняемых разделов, задерживаясь только в разделах обработки исключений вложенных блоков. Полет исключения прекращается в том блоке, в разделе обработки исключений которого исключение смогли поймать и обработать. Сразу после этого управление будет передано блоку, родительскому по отношению к тому блоку, где эта обработка произошла.

Действия в разделе обработки исключений

В разделе обработки исключений прилетевшее исключение пытаются обработать имеющимися в этом разделе обработчиками исключений, которых в блоке может быть несколько. После передачи управления в раздел обработки исключений осуществляется два действия:

- определение, какой обработчик в разделе ловит прилетевшее исключение («примерка» обработчиков);
- обработка исключения подходящим обработчиком.

«Примерка» обработчиков осуществляется по именам исключений — перед каждым обработчиком указывается список имен исключений, которые он ловит.

Если исключение не имеет имени или его имя не соответствует ни одному из имен исключений, указанных в разделе обработки исключений, то оно обрабатывается OTHERS-обработчиком, если он имеется. OTHERS-обработчик в разделе обработки исключений указывается последним и на него возлагается задача поймать все то, что не поймали другие обработчики перед ним — и системные исключения и пользовательские исключения с любыми именами.

После прилета исключения в раздел обработки возможны два случая:

- если никакой обработчик исключению не подошел, то исключение со статусом «не обработано» бросается в дальнейший полет уже в родительском блоке (блоке, предыдущим по вложенности) с того места кода, где заканчивается вложенный блок;
- если в результате «примерки» нашелся подходящий исключению обработчик, то управление передается ему.

Работа обработчика свою очередь может завершиться тремя исходами:

- команды обработчика успешно выполнились, исключение получает статус «обработано» и управление передается родительскому блоку в то место кода, где заканчивается вложенный блок;
- в процессе работы обработчика принято решение, что обрабатывать исключение надо не в этом обработчике, тогда исключение здесь же в обработчике инициируется повторно вызовом команды RAISE без параметров;
- в ходе выполнения команд обработчика инициировано новое исключение (такое бывает, например, если в обработчике ошибки регистрируются в специальной таблице, а для нее кончилось место), прилетевшее исходное исключение тогда получает статус «обработано».

Два последних исхода работы обработчика предполагают, что из блока даже с подходящим обработчиком исключение бросится дальше — либо то же самое (после вызова команды RAISE), либо уже другое. Могло прилететь пользовательское исключение, обработаться со своей ошибкой, поэтому из обработчика бросится и в родительском блоке полетит дальше уже системное исключение, как в примере с ошибкой добавления строки в специальную таблицу журнала ошибок.

Примеры обработки исключений

Рассмотрим примеры полетов исключений в программе из трех вложенных блоков:

```
BEGIN                          -- начало блока1
  команда1_блока1;
  команда2_блока1;
  команда3_блока1;
  BEGIN                        -- начало блока2
    команда1_блока2;
    команда2_блока2 l_int := 1/TO_NUMBER(l_var)  (l_var='1' или '0' или 'a')
    команда3_блока2;
  EXCEPTION
    WHEN ZERO_DIVIDE THEN
       команда1_zero_блока2;
       команда2_zero_блока2;
  END;                         -- конец блока2
  команда4_блока1;
  команда5_блока1;
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    команда1_zero_блока1;
    команда2_zero_блока1;
  WHEN OTHERS THEN
    команда1_others_блока1;
    команда2_others_блока1;
END;                           -- конец блока 1
```

Пусть команда2_блока2 **имеет вид** l_int:=1/TO_NUMBER(l_var); **где** l_int — **целочисленная переменная,** l_var — **символьная.**

l_var='1' (без исключений)	l_var='0' (zero divide)	l_var='a' (conversion error)
команда1_блока1; команда2_блока1; команда3_блока1; команда1_блока2; команда2_блока2; команда3_блока2; команда4_блока1; команда5_блока1;	команда1_блока1; команда2_блока1; команда3_блока1; команда1_блока2; команда2_блока2(error); - в блоке 2 - ловится ZERO_DIVIDE: команда1_zero_блока2; команда2_zero_блока2; -продолжение блока 1: команда4_блока1; команда5_блока1;	команда1_блока1; команда2_блока1; команда3_блока1; команда1_блока2; команда2_блока2(error); -в блоке 2 ошибка -преобразования -не ловится, т.к. -там только ZERO_DIVIDE - в блоке 1 - ZERO_DIVIDE - второй раз не ловит, - а ловит OTHERS - (он же все ловит): команда1_others_блока1; команда2_others_блока1;

Рассмотрим три случая в зависимости от значения, которое принимает переменная l_var ('1', или '0', или 'a').

Когда l_var=1 (первый столбец таблицы) исключения не инициируются выполняются все команды из разделов выполнения в той последовательности, как они записаны в коде.

В случае ошибки деления на ноль (второй столбец таблицы, l_var='0') в команде2_блока2 выполнение блока 2 прекращается, все остальные команды в блоке 2 после нее не выполняются, управление передается в раздел EXCEPTION блока 2, где пытаются поймать исключение деления на ноль (ZERO_DIVIDE). Подходящий обработчик в разделе обработки исключений блока 2 есть, поэтому исключение ловится в блоке 2, в котором успешно выполняются команды обработчика. После успешной обработки продолжается выполнение команд блока 1, родительского для блока 2, в котором произошла обработка исключения.

В случае ошибки преобразования символа к числу (третий столбец таблицы, l_var='a') исключение ошибки преобразования не ловится в разделе EXCEPTION блока 2 и PL/SQL передает управление в родительский блок 1, сразу после END блока 2 и исключение пытаются поймать в разделе EXCEPTION блока 1. В разделе обработки исключений блока 1 есть два обработчика (ZERO_DIVIDE и OTHERS). «Примерка» обработчиков к прилетевшему исключению начинается в той последовательности, как они записаны в коде (сверху вниз). ZERO_DIVIDE для этого исключения не подходит при «примерке» уже второй раз, а OTHERS-обработчик ловит все исключения, поэтому управление передается ему и выполняются две его команды. После успешного выполнения команд обработчика исключение получает статус «обработано».

Передача исключений в вызывающую среду

При разработке клиентских приложений при любом обращении к базе данных нужно предусмотреть обработку ошибок, которые могут произойти как при вызове хранимых программ PL/SQL, так и при выполнении предложений SQL. В коде клиентских программ для этого следует использовать конструкции try/catch, имеющиеся в Java и C, или try/except — в Python.

Если в ходе работы программы PL/SQL произошло так никем и не обработанное исключение, то оно вылетит «наружу», то есть будет

передано вызывавшей среде. Например, в SQL*Plus или в прикладное клиентское приложение. В SQL*Plus это выглядит вот так:

Без вылета исключения «наружу»	С вылетом исключения «наружу»

SQL> DECLARE
  2   i int;
  3 BEGIN
  4   i := 1/0;
  5 EXCEPTION
  6   WHEN OTHERS THEN
  7   DBMS_OUTPUT.PUT_LINE('/0');
  8 END;
  9 /
/0
 	
PL/SQL procedure successfully
completed.
	SQL> DECLARE
	  2 i int;
	  3 BEGIN
	  4 i := 1/0;
	  5 END;
	  6 /
	DECLARE
	*
	ERROR at line 1:
	ORA-01476: divisor is equal to zero
	ORA-06512: at line 4

Если же обработка исключений не предусмотрена ни в хранимых программах PL/SQL, ни в клиентском приложении, то исключение PL/SQL пролетит через все вложенные блоки PL/SQL, а потом через весь вызывающий код клиентского приложения. В итоге клиентское приложение покажет пользователю MessageBox с красным кругом или желтым восклицательным знаком, под которым будет написано что-то вроде

```
En error was encountered performing the requested operation
ORA-00604: error occurred at recursive SQL level 1
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 7
View program sources of error stack?
```

Как-то раз автор книги был свидетелем того, как характерная «орakловая» ошибка вида ORA-123456 выскочила в сообщении приложения банковской информационной системы, когда операционистка оформляла вклад[1]. Девушка сначала некоторое время пыталась понять, что же ей программа хочет сказать на английском языке, потом подозвала более опытного коллегу, который со слова-

[1] Монитор у операционистки стоял под таким углом, что экран был виден клиентам. Это неправильно.

ми «Это нормально, бывает» закрыл сообщение, и оформление продолжилось.

Это не нормально. Все возможные исключения в программах PL/SQL должны обрабатываться, причем продуманным унифицированным способом. В книге Стивена Фейерштейна «PL/SQL для профессионалов» теме обработки исключений посвящена отдельная глава объемом 34 страницы, что больше, чем написано в этой книге про условные команды и циклы вместе взятые.

Диагностические функции

В PL/SQL имеется несколько диагностических функций для получения информации об исключениях:

- SQLCODE — возвращает код ошибки последнего исключения, инициированного в блоке PL/SQL;
- SQLERRM — возвращает сообщение об ошибке последнего исключения, инициированного в блоке PL/SQL;
- DBMS_UTILITY.FORMAT_ERROR_BACKTRACE — возвращает отформатированную строку с содержимым стека программ и номеров строк кода.

Максимальная длина строки, возвращаемой функцией SQLERRM, составляет 512 байт. Из-за этого ограничения рекомендуется использовать вместо SQLERRM функцию встроенного пакета DBMS_UTILITY.FORMAT_ERROR_STACK, которая выводит строку с отформатированным стеком сообщений. Приведем несколько примеров использования диагностических функций:

```
SQL> CREATE OR REPLACE PROCEDURE error_proc IS
  2    i INTEGER;
  3  BEGIN
  4    i := 1/0;
  5    i := 15;
  6  END;
  7  /
Procedure created.

SQL> CREATE OR REPLACE PROCEDURE parent_proc IS
  2  BEGIN
  3    error_proc;
  4  END;
  5  /
Procedure created.
```

```
SQL> BEGIN
  2    parent_proc;
  3  EXCEPTION
  4    WHEN OTHERS THEN
  5      DBMS_OUTPUT.PUT_LINE('SQLCODE: '||SQLCODE);
  6      DBMS_OUTPUT.PUT_LINE('SQLERRM: '||SQLERRM);
  7      DBMS_OUTPUT.PUT_LINE('DBMS_UTILITY.FORMAT_ERROR_BACKTRACE:');
  8      DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.FORMAT_ERROR_BACKTRACE);
  9  END;
 10  /
SQLCODE: -1476
SQLERRM: ORA-01476: divisor is equal to zero
DBMS_UTILITY.FORMAT_ERROR_BACKTRACE:
ORA-06512: at "U1.ERROR_PROC", line 4
ORA-06512: at "U2.PARENT_PROC", line 3
ORA-06512: at line 2

PL/SQL procedure successfully completed.
```

По строке, которую вернула FORMAT_ERROR_BACKTRACE, видно, как пролетало системное исключение по строкам кода: сначала она возникла в процедуре error_proc на четвертой строке, управление из error_proc сразу вернулось в родительский для error_proc блок — процедуру parent_proc (на третью строку, где вызывалась error_proc). Далее выводятся сведения о второй строке анонимного блока, в котором вызывалась parent_proc. При этом в стеке появились три ошибки ORA-06512[1].

Также видно, что функция DBMS_UTILITY.FORMAT_ERROR_BACKTRACE не выдает сообщение о самой исходной ошибке, поэтому совместно с этой функцией следует использовать функции SQLERRM или DBMS_UTILITY.FORMAT_ERROR_STACK.

Пользовательские исключения

Пользовательские исключения объявляются следующим образом:

```
имя исключения EXCEPTION;
```

[1] ORA-06512 is caused by the stack being unwound by unhandled exceptions in your PL/SQL code.

Пользовательские исключения инициируются командой RAISE**, у которой есть две формы:**

- RAISE имя исключения **(исключение должно быть предопределенным в пакете** STANDARD **или объявленным в области видимости);**
- RAISE **(может быть вызвана только внутри обработчиков исключений, когда в обработчике нужно повторно инициировать то же самое исключение).**

Приведем пример работы с пользовательскими исключениями:

```
DECLARE
  l_amount INTEGER           := -100;
  l_crncy  VARCHAR2(3)       := 'RUR';
  ex_negative_payment_amount EXCEPTION;
  ex_non_rur_payment         EXCEPTION;
BEGIN

  IF l_amount < 0 THEN
    RAISE ex_negative_payment_amount;
  END IF;

  IF l_crncy <> 'RUR' THEN
    RAISE ex_non_rur_payment;
  END IF;

  ... все проверки пройдены, обрабатываем платеж

EXCEPTION
  WHEN ex_negative_payment_amount THEN
    DBMS_OUTPUT.PUT_LINE('Ошибка: сумма отрицательная: '||l_amount);
  WHEN ex_non_rur_payment THEN
    DBMS_OUTPUT.PUT_LINE('Ошибка: платеж не в рублях');
END;
```

Видно, что код имеет линейный вид: проверки записаны одна за одной и если платеж не проходит проверку, то управление сразу переходит в раздел обработки исключений. Без исключений код выглядит нелинейно — как несколько ветвей команды IF**:**

```
DECLARE
  l_amount INTEGER           := -100;
  l_crncy  VARCHAR2(3)       := 'RUR';
  ex_negative_payment_amount EXCEPTION;
  ex_non_rur_payment         EXCEPTION;
BEGIN

  IF l_amount < 0 THEN
    DBMS_OUTPUT.PUT_LINE('Ошибка: сумма отрицательная: '||l_amount);
```

```
  ELSE
    -- второй уровень вложенности IF
    IF l_crncy <> 'RUR' THEN
      DBMS_OUTPUT.PUT_LINE('Ошибка: платеж не в рублях');
    ELSE
      -- потом будет третий уровень вложенности IF
      ... наконец все проверки пройдены, обрабатываем платеж
    END IF;
  END IF;
END;
```

Такой код труднее сопровождать и поддерживать, особенно если логика обработки распределена по многим вложенным вызовам процедур и функций. В этом случае пришлось бы использовать переменные-флаги, передавать и анализировать при каждом вызове коды завершения и т. д.

Посмотрим на поведение диагностических функций при инициировании пользовательских исключений:

```
SQL> DECLARE
  2    exception1 EXCEPTION;
  3  BEGIN
  4    RAISE exception1;
  5  EXCEPTION
  6    WHEN OTHERS THEN
  7      DBMS_OUTPUT.PUT_LINE('SQLCODE print: '||SQLCODE);
  8      DBMS_OUTPUT.PUT_LINE('SQLERRM print: '||SQLERRM);
  9  END;
 10  /
SQLCODE print: 1
SQLERRM print: User-Defined Exception
PL/SQL procedure successfully completed.
```

Видно, что пользовательские исключения имеют код ошибки 1, а сообщение об ошибке малоинформативно. Поэтому при работе над кодом нужно следить, чтобы пользовательские исключения обрабатывались соответствующими им обработчиками, а не OTHERS-обработчиком, внутри которого нельзя узнать, какое именно пользовательское исключение прилетело. И уж тем более пользовательские исключения PL/SQL не должны вылетать «наружу» в вызывающую среду.

Процедура RAISE_APPLICATION_ERROR

Если пользовательское исключение все-таки вылетит «наружу» в вызывающую среду, то независимо от того, какое исключение выле-

тело, «снаружи» выглядеть это будет одинаково — как ошибка ORA-06510.

Исключение exception1	Исключение exception2

SQL> DECLARE
  2 exception1 EXCEPTION;
  3 BEGIN
  4 RAISE exception1;
  5 END;
  6 /
DECLARE
*
ERROR at line 1:
ORA-06510: PL/SQL:
unhandled user-defined exception
ORA-06512: at line 4
	SQL> DECLARE
	  2 exception2 EXCEPTION;
	  3 BEGIN
	  4 RAISE exception2;
	  5 END;
	  6 /
	DECLARE
	*
	ERROR at line 1:
	ORA-06510: PL/SQL:
	unhandled user-defined exception
	ORA-06512: at line 4

С системными исключениями дело обстоит иначе — разные системные исключения вылетают «наружу» с разными кодами и сообщениями, что позволяет их обрабатывать в вызывающей среде, так как по коду понятно, какая ошибка произошла.

Как отмечалось выше, системные исключения автоматически инициируются виртуальной машиной PL/SQL при возникновении программных ошибок этапа выполнения. В то же время есть возможность инициировать системные исключения и вручную. Для этого используется процедура RAISE_APPLICATION_ERROR, которая инициирует системные исключения с задаваемыми программистом сообщениями и номерами ошибок из диапазона [-20999,-20000].

```
SQL> DECLARE
  2    l_error_number  INTEGER := -20187;
  3    l_error_message VARCHAR2(100) := 'Отрицательная сумма платежа';
  4    l_amount        INTEGER := -10;
  5  BEGIN
  6    IF l_amount < 0 THEN
  7      RAISE_APPLICATION_ERROR(l_error_number,l_error_message);
  8    END IF;
  9  END;
 10  /
DECLARE
*
ERROR at line 1:
ORA-20187: Отрицательная сумма платежа
ORA-06512: at line 7
```

Привязка пользовательских исключений к ошибкам

В рассмотренных примерах работы с исключениями часто использовались предопределенные исключения, например, исключение ZERO_DIVIDE, которое соответствует ошибке ORA-01476 Divisor is equal to zero. В PL/SQL есть возможность практически к любой ошибке сервера привязать пользовательское исключение и ловить ошибки по именам таких исключений, а не в OTHERS-обработчике.

Для привязки ошибки к пользовательскому исключению достаточно узнать номер ошибки и записать соответствующую директиву компилятору:

```
PRAGMA EXCEPTION_INIT (имя пользовательского исключения, номер ошибки)
```

Обработаем ошибку преобразования символьного значения в дату по заданной маске с использованием привязанного исключения:

```
-- сначала специально получаем ошибку и узнаем ее номер,
-- это вспомогательный шаг
SQL>  DECLARE
  2     v1 DATE;
  3  BEGIN
  4    v1:=TO_DATE('abc','dd.mm.yyyy'); -- ожидается не abc, а дата по маске
  5  END;
  6  /
 DECLARE
*
ERROR at line 1:
ORA-01858: a non-numeric character was found where a numeric was expected
ORA-06512: at line 4

-- объявляем исключение и привязываем его к ORA-01858
SQL>  DECLARE
  2     v1 DATE;
  3     to_date_convert_error EXCEPTION;
  4     PRAGMA EXCEPTION_INIT (to_date_convert_error, -01858);
  5  BEGIN
  6    v1:=TO_DATE('abc','dd.mm.yyyy');
  7  EXCEPTION
  8    WHEN to_date_convert_error THEN
  9      DBMS_OUTPUT.PUT_LINE('Ошибка преобразования строки в дату');
 10  END;
 11  /
Ошибка преобразования строки в дату
PL/SQL procedure successfully completed.
```

Если бы возможности привязки к ошибкам пользовательских исключений не было, то пришлось бы все ошибки обрабатывать в OTHERS-обработчике примерно так:

```
BEGIN

    v1:=TO_DATE('abc','dd.mm.yyyy');

EXCEPTION

  WHEN OTHERS THEN

    IF SQLCODE = -01858 THEN
      DBMS_OUTPUT.PUT_LINE('Ошибка преобразования строки в дату');
    END IF;

    IF SQLCODE = -... THEN
      ...
    END IF;

    ... и так для каждой ожидаемой ошибки
    ... (в конце ELSE-ветвь - для неожидаемых ошибок)

END;
```

В результате применения такого подхода OTHERS-обработчик превратился бы в громоздкий фрагмент кода значительного объема с большим числом команд IF и/или CASE. Привязка пользовательских исключений к ошибкам позволяет избежать этого, распределив обработку ошибок по отдельным обработчикам.

Хорошим стилем является объявление в одном месте кода всех пользовательских исключений[1] с привязкой к ошибкам. Далее эти ошибки можно ловить по именам исключений и иметь отдельные обработчики с обозримым объемом кода для каждой ошибки. При необходимости можно разделить ошибки на критичные и некритичные. Некритичные ошибки следует ловить и обрабатывать с продолжением вычислений (например, в циклах обычно обрабатывается ошибка на сбойной итерации и происходит переход на следующую итерацию цикла). Критичные ошибки «катапультировать» через все

[1] Обычно для этого используются спецификации пакетов, рассматриваемые далее.

вложенные блоки, вызывая во всех обработчиках исключений команду RAISE до тех пор, пока критичная ошибка не долетит до раздела обработки критичных ошибок.

Использование обработчика OTHERS

Как было сказано ранее, обработчик OTHERS указывается последним в разделе обработки исключений. Он ловит все исключения, которые не поймали обработчики, перечисленные раньше него. Чаще всего обработчик OTHERS используется для того, чтобы обрабатывать системные исключения, инициируемые из-за возникновения ошибок.

При неправильном использовании наличие обработчика OTHERS может стать причиной «потери» для клиентских приложений ошибок в программах PL/SQL. Все вызовы программы PL/SQL с обработчиком WHEN OTHERS в разделе обработки исключений самого внешнего блока будут завершаться успешно, никакие ошибки «наружу» вылетать не будут. При этом в программе на P/SQL могут происходить и фатальные ошибки, но о них никто сразу не узнает, они будут «подавлены» в разделе обработки исключений.

Особенно плохой практикой является использование обработчиков вида

```
BEGIN
  …
EXCEPTION
  WHEN OTHERS THEN NULL;
END;
```

В этом случае исключение даже никак не регистрируется и просто теряется[1]. Автор книги в унаследованной системе как-то раз столкнулся с тем, что при загрузке с помощью SQL*Loader некоторые строки то загружались, то не загружались. После нескольких часов жизни, напрасно потерянных на проверку различных гипотез, случайно был обнаружен триггер, который срабатывал на вставку строк предложением INSERT. В этом триггере осуществлялось преобразование строки в дату, которое то было успешным, то завершалось

[1] Начиная с версии Oracle 11g для подобных блоков кода на этапе компиляции даже выдается соответствующее предупреждение (warning).

ошибкой. Ошибки эти обрабатывались так, как показано выше, то есть в новых записях то проставлялись даты, то нет. В результате в логах SQL*Loader появлялись сообщения об ошибках нарушения ограничения целостности для столбца с датами, а причина этого была неясна.

В том памятном случае разработчик триггера сделал три ошибки сразу:

- понадеялся на неявное преобразование строки в дату без указания маски (плохая практика) и в этом была причина ошибки, которая то была, то нет;
- с помощью изменения псевдозаписей :NEW тихо подменял значения столбцов добавляемых в таблицу строк в BEFORE-триггере (очень плохая практика);
- подавлял происходящие ошибки в обработчике OTHERS (очень-очень плохая практика).

SQL в программах PL/SQL

Выполнение SQL в программах PL/SQL

Выполнение предложений SQL в программах PL/SQL происходит совершенно естественным образом. И языковые конструкции PL/SQL, и компилятор, и виртуальная машина изначально разрабатывались и непрерывно улучшались специально для этого.

Вообще говоря, сервер Oracle не делает для виртуальной машины PL/SQL никаких преференций в части выполнения предложений SQL за то, что она работает в его ядре. В ходе выполнения предложений SQL из байт-кода PL/SQL происходят точно такие же действия, как и в ходе выполнения предложений SQL из любых других программ, подключающихся к серверу Oracle.

При подключении к серверу Oracle клиентской программы создается выделенный серверный процесс[1], который обрабатывает поступающие предложения SQL и вызовы программ PL/SQL. Эта обработка

[1] Для простоты считаем, что подключение к экземпляру Oracle производится в режиме выделенного сервера.

происходит на нескольких уровнях ядра Oracle, при этом собственно выполнение и SQL и PL/SQL осуществляется на одном и том же уровне — уровне выполнения (Execution Layer (KX)[1]). Когда выполняется какое-то предложение SQL, то действия процесса осуществляются в контексте SQL, а когда происходит вызов программы PL/SQL, то действия процесса осуществляются в контексте PL/SQL. Когда же в ходе работы программы PL/SQL потребуется выполнить какое-нибудь предложение SQL из ее байт-кода, то произойдет переключение контекста PL/SQL-SQL и это предложение SQL будет выполнено серверным процессом на уровне KX точно так же, как будто бы оно поступило не из программы PL/SQL, а из любой другой программы. После обработки предложения SQL произойдет обратное переключение контекста SQL-PL/SQL.

Достоинства использования PL/SQL для выполнения предложений SQL заключаются в следующем

- в PL/SQL есть удобные и лаконичные языковые конструкции обработки результирующих выборок SQL-запросов;
- компилятор PL/SQL по исходному коду программы PL/SQL формирует предложения SQL со связываемыми переменными, использование которых позволяет избежать многих проблем с сервером Oracle;
- PL/SQL автоматически оптимально управляет курсорами — важнейшими внутренними механизмами Oracle для выполнения предложений SQL;
- в PL/SQL есть средства дополнительной оптимизации для массовой обработки (bulk collect) данных.

Можно сказать, что PL/SQL — это такой своеобразный движок (engine) для отправки предложений SQL на выполнение и работы с возвращаемыми ими результатами. Движок этот работает в ядре сервера Oracle и написан сотрудниками самой компании Oracle, по-

---

[1] The execution layer (KX). This layer handles the binding and execution of SQL statements and PL/SQL program units. It is also responsible for the execution of recursive calls for trigger execution, and for the execution of SQL statements within PL/SQL program units.

этому он является очень эффективным средством реализации бизнес-логики с использованием языка SQL.

Выборка данных с использованием курсоров

Выборка данных является важнейшей операцией при реализации серверной бизнес-логики. Поэтому разработчики языка PL/SQL продумали и реализовали языковые конструкции, позволяющие просто и эффективно выполнять предложения `SELECT` языка SQL и осуществлять обработку их результирующих выборок. В других языках программирования с этим все намного сложнее.

Приведем цитату из интервью с Майклом Стоунбрейкером.

«...Сейчас мы общаемся с базами данных, используя ODBC и JDBC, встроенные в языки программирования. Это наихудшие интерфейсы на нашей планете. Я имею в виду, что они настолько ужасны, что их не пожелаешь даже злейшему врагу.

Взгляните на такой язык, как Ruby on Rails (www.rubyonrails.org). Этот язык расширен встроенными средствами доступа к базам данных. Не нужно обращаться к SQL; достаточно сказать: «`for E in employee do`», и для доступа к базе данных используются языковые конструкции и переменные. Это существенно облегчает работу программиста...».

Каркас приложений (framework) «Ruby на рельсах» для модного языка Ruby появился в 2004 году, а еще за 15 лет до этого в языке PL/SQL уже имелся курсорный цикл `FOR` и достаточно было написать «`FOR E in (SELECT * FROM employee) LOOP`». Простой и элегантный код.

Понятие курсора

Напомним, что к DML-предложениям языка SQL относятся предложения `INSERT`, `UPDATE`, `DELETE` и предложение `SELECT`, которое дальше будет также называться SQL-запросом. Курсором (cursor) в Oracle называется именованный указатель на приватную рабочую область в памяти, используемую в ходе обработки DML-предложений. Выполняя действия с курсором, можно получить доступ к результирующей выборке связанного в текущий момент времени с этим курсором SQL-запроса и к другим сведениям о ходе обработки SQL, например, получить число обработанных строк для предложений `INSERT`, `UPDATE`, `DELETE`.

В некоторых книгах проводится аналогия между курсором в окне текстового редактора и курсором в базе данных. В текстовом редакторе клавишами ↑↓ можно двигаться по просматриваемому тексту вверх и вниз, точно так же с помощью курсора базы данных можно пролистывать результирующую выборку для курсора с SQL-запросом[1].

В PL/SQL есть явные и неявные курсоры (explicit and implicit cursors):

- явные курсоры объявляются с указанием текстов SQL-запросов в разделах объявлений блоков PL/SQL;
- неявные курсоры используются при выполнении команд SELECT INTO и команд INSERT, UPDATE и DELETE.

Неявный курсор не объявляется в разделах объявлений, не имеет имени и называется неявным потому, что виртуальная машина PL/SQL автоматически неявно (то есть без участия программиста) выполняет необходимые действия с ним.

Явный курсор имеет имя, указываемое при объявлении курсора, и все действия с таким курсором должны быть явно указаны в исходном коде.

Код программы на языке PL/SQL состоит из команд PL/SQL. Отметим, что рассматриваемые далее INSERT, DELETE, UPDATE и SELECT INTO — это именно команды PL/SQL, а не предложения SQL, хотя и очень на них похожие. Для текста таких команд PL/SQL компилятором осуществляется препроцессинг, то есть обработка исходного кода для передачи на следующий шаг компиляции. Эта обработка заключается в подготовке предложений SQL для последующего их размещения в байт-коде программ PL/SQL, причем текст SQL будет отличаться от того текста, который был в соответствующих командах PL/SQL. Например, все переменные PL/SQL будут заменены на связываемые переменные SQL, а текст сформированных предложений SQL приведен к верхнему регистру.

---

[1] В Oracle результирующую выборку с помощью курсора можно пролистывать только в одну сторону — вниз.

Также в подготовленном компилятором PL/SQL байт-коде будут предусмотрены низкоуровневые вызовы сервера Oracle для выполнения этих сформированных предложений SQL: открытие курсоров, привязка значений переменных, выполнение, считывание строк результирующих выборок и закрытие курсоров.

Для неявных курсоров компилятор эти вызовы разместит в байт-коде автоматически, для явных курсоров — по командам PL/SQL, явно заданным программистом в исходном коде. Ответственность за правильное расположение этих команд лежит на программисте. Нарушение последовательности действий с явным курсором приводит к ошибкам этапа выполнения. Если, например, попытаться считать запись из неоткрытого курсора, то будет инициировано системное исключение.

Неявные курсоры для выборки данных

Неявный курсор для выборки данных используется для команды PL/SQL `SELECT INTO`, обладающей следующими свойствами:

- результирующая выборка SQL-запроса должна содержать ровно одну строку (не ноль строк, не две, не три строки, а ровно одну);
- конструкция `INTO` представляет собой механизм передачи значений столбцов строки выборки в переменные программы PL/SQL[1].

Рассмотрим пример.

Пусть в базе данных существует таблица `tab1`, созданная и заполненная следующим образом:

```
CREATE TABLE tab1 (at1 NUMBER, at2 VARCHAR2(1));
INSERT INTO tab1 VALUES (1, 'A');
INSERT INTO tab1 VALUES (2, 'B');
INSERT INTO tab1 VALUES (3, 'C');
```

[1] В параграфе «Массовая обработка» рассматривается конструкция `BULK COLLECT`, которая позволяет считывать из неявного курсора не ровно одну, а несколько строк результирующей выборки в коллекцию.

Приведем примеры различных ситуаций, возникающих при выборке данных с использованием неявного курсора.

```
SQL> DECLARE
 2   l_at1 NUMBER;
 3   l_at2 VARCHAR2(1);
 4 BEGIN
 5   SELECT at1,at2 INTO l_at1,l_at2
 6   FROM tab1 WHERE at1=1;
 7   DBMS_OUTPUT.PUT_LINE(TO_CHAR(l_at1)||''||l_at2);
 8 END;
 9 /
1  A

SQL> DECLARE
 2   l_at1 NUMBER;
 3   l_at2 VARCHAR2(1);
 4 BEGIN
 5   SELECT at1,at2 INTO l_at1,l_at2
 6   FROM tab1 WHERE at1=4;
 7   DBMS_OUTPUT.PUT_LINE(TO_CHAR(l_at1)||''||l_at2);
 8 END;
 9 /
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 5

SQL> DECLARE
 2  l_at1 NUMBER;
 3  l_at2 VARCHAR2(1);
 4 BEGIN
 5   SELECT at1,at2 INTO l_at1,l_at2 FROM tab1
 6   WHERE at1 IN (1,2);
 7   DBMS_OUTPUT.PUT_LINE(TO_CHAR(l_at1)||''||l_at2);
 8 END;
 9  /
DECLARE
*
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 5
```

Если SQL-запрос команды `SELECT INTO` **возвращает больше одной строки, то инициируется предопределенное исключение** `TOO_MANY_ROWS`**. Если возвращается пустая выборка, то инициируется другое предопределенное исключение —** `NO_DATA_FOUND`**. По этой при-**

чине команду SELECT INTO рекомендуется помещать в отдельный блок с обработкой этих исключений:

```
BEGIN
...
  BEGIN
    SELECT INTO ...
  EXCEPTION
    WHEN TOO_MANY_ROWS THEN ...
    WHEN NO_DATA_FOUND THEN ...
  END;
...
END;
```

Команда SELECT INTO обычно используется тогда, когда есть уверенность, что ее SQL-запрос вернет ровно одну строку, например, для запроса строк таблицы с условием на значение ее первичного ключа.

Явные курсоры

Объявив SQL-запрос с помощью явного курсора, программист получает полный контроль над этапами его выполнения. Можно определить, когда открыть курсор (OPEN), когда считывать из него строки (FETCH) и когда закрыть курсор (CLOSE).

Объявим курсор cur1:

```
CURSOR cur1 IS SELECT at1,at2 FROM tab1;
```

Первым шагом работы с курсором является его открытие:

```
OPEN cur1;
```

Считывание строк результирующей выборки из курсора выполняется командой FETCH в набор переменных PL/SQL подходящих типов (число переменных должно совпадать с числом столбцов выборки):

```
FETCH cur1 INTO l_at1, l_at2;
```

Полностью код для получения трех строк из tab1 выглядит так:

```
SQL> DECLARE
  2    CURSOR cur1 IS SELECT * FROM tab1;
  3    rec tab1%ROWTYPE;
  4  BEGIN
  5    OPEN cur1;
  6    FOR i IN 1..3 LOOP
```

```
  7      FETCH cur1 INTO rec;
  8      DBMS_OUTPUT.PUT_LINE(TO_CHAR(rec.at1)||'  '||rec.at2);
  9    END LOOP;
 10  END;
 11  /
1  A
2  B
3  C
PL/SQL procedure successfully completed.
```

После того, как курсор стал ненужным, его следует закрыть:

```
CLOSE cur1;
```

Если забыть закрыть явный курсор, как в приведенном выше примере, то можно считать, что запрограммирована утечка памяти в сервере Oracle. Вообще говоря, виртуальная машина PL/SQL автоматически сама закрывает и уничтожает открытые курсоры, как только они оказываются вне области видимости для выполняющегося в настоящий момент блока. Однако делается это не сразу, какое-то время такой курсор существует и остается открытым. У экземпляра Oracle есть ограничение на число одновременно открытых курсоров, которое задается параметром экземпляра `open_cursors` (по умолчанию параметр выставлен в 300). Если превысить значение этого параметра, то выполнение любого предложения SQL будет завершаться ошибкой. При параллельной работе большого числа сессий это весьма вероятно, поэтому чтобы не сталкиваться с ошибками такого вида, настоятельно рекомендуется аккуратно закрывать курсоры.

Программа, представленная выше, неудачна еще и тем, что цикл `FOR` со счетчиком предусматривает считывание конкретного числа строк, которых, вообще говоря, может и не быть в результирующей выборке. Строк в выборке может быть больше, чем указано (в данном случае у цикла `FOR` счетчик изменяется до 3) и тогда какие-то строки останутся несчитанными. Также возможна ситуация, когда число строк в выборке меньше значения счетчика — тогда произойдет повторное считывание последней строки.

Отметим, что повторное считывание из курсора последней строки выборки не приводит к ошибкам. Если в выборке, например, n строк, а команда `FETCH` выполнена k раз (k>n), то повторные считывания последней (n-й) строки не приведут к инициированию системных исключений, просто последняя строка выборки будет считана и выведена на экран несколько (k-n+1) раз. Для организации перебора

строк результирующей выборки предназначены атрибуты явных курсоров, которые рассматриваются далее.

Объявление записей PL/SQL на основе курсоров

Переменная rec, в которую считывались строки результирующей выборки, была объявлена с помощью атрибута %ROWTYPE как запись PL/SQL на основе таблицы tab1. В данном случае это оправдано, потому что в SQL-запросе осуществляется выборка всех столбцов одной таблицы tab1 (SELECT * FROM tab1). Число атрибутов записи PL/SQL будет соответствовать числу столбцов строк выборки и считывание строк пройдет без ошибок.

Однако столбцы результирующей выборки могут быть не из одной, а из нескольких таблиц или вовсе могут являться выражениями:

```
CURSOR cur_short_person IS
SELECT born,
       surname||' '||SUBSTR(name,1,1)||'.'||SUBSTR(secname,1,1)||'.' AS fio,
       passport.seria||' '||passport.num AS passport_data
FROM person, passport
WHERE person.id=13243297
      AND person.id=passport.r$person
```

В столбце fio результирующей выборки для каждой строки таблицы person будет результат выражения — фамилия и инициалы (например, Кислов Виктор Михайлович — Кислов В.М.). В столбце passport_data будут паспортные данные из таблицы passport и тоже выражением — серия и номер паспорта, «склеенные» через пробел.

Самый правильный способ определить то, во что будем «принимать» результирующую выборку SQL-запроса курсора — это объявить с помощью атрибута %ROWTYPE переменную-запись PL/SQL, основанную не на схеме одной таблицы, а прямо на курсоре. В этом случае список атрибутов записи PL/SQL по числу столбцов результирующей выборки будет сформирован автоматически.

```
l_short_person cur_short_person%ROWTYPE;
OPEN cur_short_person;
FETCH cur_short_person INTO l_short_person;

DBMS_OUTPUT.PUT_LINE('ФИО:           '||l_short_person.fio);
DBMS_OUTPUT.PUT_LINE('Дата рождения: '||TO_CHAR(l_short_person.born));
DBMS_OUTPUT.PUT_LINE('Паспорт:       '||l_short_person.passport_data);
```

В приведенном коде считывание строки результирующей выборки в запись PL/SQL осуществляется одной короткой командой FETCH без указания столбцов. При появлении новых столбцов во фразе SELECT запроса курсора новые атрибуты также автоматически появятся в записях PL/SQL, объявленных на основе курсора. Таким образом, объявление записей PL/SQL на основе курсоров позволяет писать компактный, поддерживаемый и расширяемый код.

Атрибуты явного курсора

Для управления считыванием строк из явных курсоров используются их атрибуты. В частности, они позволяют выполнить считывание для последующей обработки в программе всех строк результирующей выборки.

Таблица 3. Атрибуты явного курсора.

Атрибут курсора	Описание атрибута
%FOUND	TRUE, если из курсора считана очередная строка
%NOTFOUND	FALSE, если из курсора считана очередная строка
%ROWCOUNT	количество считанных до настоящего момента строк
%ISOPEN	TRUE, если курсор открыт

Основная нагрузка при считывании всех строк результирующей выборки ложится на атрибуты курсора %NOTFOUND и %FOUND, которые всегда находятся в связке — либо принимают противоположные логические значения TRUE и FALSE, либо оба UNKNOWN.

Атрибут %FOUND равен TRUE и атрибут %FOUND равен FALSE в то время, пока команда FETCH считывает из курсора все новые и новые (очередные) строки. После того, как последняя строка результирующей выборки будет считана дважды (второй раз, выходит, уже не как очередная), атрибут курсора %FOUND станет FALSE, а %NOTFOUND станет TRUE. На этом поведении атрибутов курсора обычно и формируется условие выхода из циклов, предназначенных для считывания из курсора всех строк результирующей выборки.

Еще одним важным фактом является то, что после открытия курсора, но до выполнения первой команды FETCH, атрибуты %FOUND и %NOTFOUND имеют неопределенное логическое значение (UNKNOWN). Если это не учитывать, то можно совершить одну из распространенных

ошибок — в цикле WHILE с условием на истинность атрибута %FOUND цикл не будет выполнен ни разу, несмотря на то, что в результирующей выборке есть строки. Выполнить команду FETCH первый раз надо еще до входа в цикл, тем самым проинициализировав атрибуты курсора.

Приведем пример использования атрибутов курсора в цикле WHILE для считывания всех строк результирующей выборки.

```
SQL> DECLARE
  2    CURSOR cur1 IS SELECT * FROM tab1;
  3    rec cur1%ROWTYPE;
  4  BEGIN
  5    OPEN cur1;
  6    FETCH cur1 INTO rec;
  7    WHILE cur1%FOUND LOOP
  8      DBMS_OUTPUT.PUT_LINE(TO_CHAR(rec.at1)||'  '||rec.at2);
  9      FETCH cur1 INTO rec;
 10    END LOOP;
 11    CLOSE cur1;
 12  END;
 13  /
1  A
2  B
3  C

PL/SQL procedure successfully completed.
```

Еще один пример показывает использование атрибутов курсора для считывания всех строк выборки в простом цикле LOOP END LOOP с условием выхода EXIT WHEN.

```
SQL> DECLARE
  2    TYPE tab1_rec_type IS RECORD
  3    (arg1 tab1.at1%TYPE,
  4     arg2 tab1.at2%TYPE);
  5    tab1_rec tab1_rec_type;
  6    CURSOR cur1 IS SELECT * FROM tab1;
  7  BEGIN
  8    OPEN cur1;
  9    LOOP
 10      EXIT WHEN (cur1%NOTFOUND);
 11      FETCH cur1 INTO tab1_rec;
 12      DBMS_OUTPUT.PUT_LINE(cur1%ROWCOUNT||' '||tab1_rec.arg2);
 13    END LOOP;
 14    CLOSE cur1;
 15  END;
/
```

```
1 A
2 B
3 C
3 C
PL/SQL procedure successfully completed.
```

Обратите внимание на повторный вывод последней строки (3 C). Это еще одна распространенная ошибка. В ходе проведения занятий со студентами автор десятки раз видел считывание и обработку последней строки выборки дважды. Системное исключение при повторном считывании последней строки выборки, напомним, не инициируется, поэтому такие ошибки в коде трудно обнаруживаются.

Рекомендуется после написания кода, реализующего считывание и обработку всех строк выборки, проверить его с помощью небольших тестов на отсутствие двух распространенных ошибок:

- цикл считывания не выполняется ни разу;
- последняя строка выборки в цикле обрабатывается дважды.

В приведенном выше примере показано, что значение %ROWCOUNT увеличивается на единицу с каждой считанной строкой, а не отражает общее число отобранных SQL-запросом строк. Видно и что повторное считывание последней строки выборки не влияет на значение атрибута %ROWCOUNT: оно остается равным значению, присвоенному при первом считывании последней строки. В примере значение атрибута %ROWCOUNT как стало равным трем при первом считывании последней строки, так и осталось без изменений после еще одного считывания.

Курсорный цикл FOR

Курсорный цикл FOR позволяет в цикле обработать все строки результирующей выборки SQL-запроса.

```
SQL> DECLARE
  2    CURSOR cur1 IS SELECT at1,at2 FROM tab1;
  3    v1 VARCHAR2(4000);
  4  BEGIN
  5    FOR rec IN cur1 LOOP
  6      v1:=LTRIM(v1||' '||rec.at2);
  7    END LOOP;
  8    DBMS_OUTPUT.PUT_LINE(v1);
  9  END;
 10  /
A B C
PL/SQL procedure successfully completed.
```

Обратите внимание, переменная rec, в которую в цикле считываются данные, не требует объявления. Она будет являться записью PL/SQL, такой же, как записи PL/SQL, объявленные с помощью атрибута %ROWTYPE на основе курсора.

Все очень просто. Не нужно явно открывать и закрывать курсор. Вместо команды FETCH просто следует обратиться к текущему значению записи PL/SQL, которая здесь является своеобразной управляющей переменной цикла. Для выхода из цикла больше не нужно проверять атрибуты курсора %NOTFOUND и %FOUND. Если SQL-запрос не отберет ни одной строки, тело цикла просто не выполнится ни разу, если же результирующая выборка непустая, то после перебора всех строк цикл завершится автоматически.

По сути, программист тремя строчками кода говорит компилятору PL/SQL «Мне нужна каждая строка результирующей выборки, и я хочу, чтобы она была помещена в запись PL/SQL, соответствующую курсору». Компилятор PL/SQL формирует соответствующий байт-код со всеми низкоуровневыми вызовами сервера.

Простейший вариант курсорного цикла FOR имеет SQL-запрос, встроенный прямо в описание цикла:

```
SQL> DECLARE
  2    v1 VARCHAR2(4000);
  3  BEGIN
  4    FOR rec IN (SELECT at1,at2 FROM tab1) LOOP
  5      v1:= v1||' '||rec.at2;
  6    END LOOP;
  7    DBMS_OUTPUT.PUT_LINE(LTRIM(v1));
  8  END;
  9  /
A B C

PL/SQL procedure successfully completed.
```

Как и для неявных курсоров, для курсорного цикла FOR компилятор сам разместит в байт-коде низкоуровневые вызовы открытия курсора, считывания из него строк и закрытия.

Параметры курсора

Объявление курсора может содержать параметрический запрос, значения параметров которого передаются при открытии курсора. Рассмотрим соответствующий пример.

```
SQL> SELECT * FROM tab1;
       AT1 A
---------- -
         1 A
         2 B
         3 C

SQL> DECLARE
  2    CURSOR cur2 (i INTEGER) IS SELECT * FROM tab1 WHERE at1>=i;
  3    cur2_rec cur2%ROWTYPE;
  4  BEGIN
  5    OPEN cur2(2); -- курсор открыт с параметром i, равным 2
  6    FETCH cur2 INTO cur2_rec;
  7    WHILE cur2%FOUND LOOP
  8      DBMS_OUTPUT.PUT_LINE(cur2_rec.at1);
  9      FETCH cur2 INTO cur2_rec;
 10    END LOOP;
 11    CLOSE cur2;
 12  END;
 13  /
2
3

PL/SQL procedure successfully completed.
```

Помимо явных курсоров параметризировать можно и команды SELECT INTO и курсорные циклы FOR. Для этого в коде SQL в качестве параметров надо использовать ранее объявленные переменные PL/SQL скалярных типов данных. При препроцессинге эти переменные будут автоматически заменены компилятором PL/SQL на связываемые переменные SQL.

Добавление, изменение и удаление данных

В исходном коде программы PL/SQL можно указывать команды добавления, изменения и удаления данных, для которых компилятор формирует в байт-коде вызовы соответствующих предложений SQL — INSERT, UPDATE и DELETE.

```
SQL> CREATE TABLE tab1 (at1 integer);
Table created.
```

```
SQL> DECLARE
  2    l_at1 tab1.at1%TYPE;
  3  BEGIN
  4    l_at1 := 1;
  5    INSERT INTO tab1 VALUES (l_at1);
  6
  7    l_at1 := 2;
  8    INSERT INTO tab1 VALUES (l_at1);
  9
 10    l_at1 := 3;
 11    INSERT INTO tab1 VALUES (l_at1);

 12    DELETE FROM tab1 WHERE at1=1;
 13
 14    UPDATE tab1 SET at1=at1+1 WHERE at1=l_at1;
 15
 16  END;
 17  /
PL/SQL procedure successfully completed.

SQL> SELECT * FROM tab1;
       AT1
----------
         2
         4
```

Используемые для выполнения команд INSERT, UPDATE, DELETE неявные курсоры тоже имеют атрибуты. Чтобы получить их значения, следует указывать имя курсора SQL%.

Таблица 4. Атрибуты неявного курсора.

SQL%FOUND	возвращает TRUE, если хотя бы одна строка была обработана DML-предложением SQL
SQL%NOTFOUND	возвращает TRUE, если ни одной строки не было обработано
SQL%ROWCOUNT	возвращает количество обработанных строк
SQL%ISOPEN	для неявных курсоров всегда возвращает FALSE, поскольку Oracle закрывает и открывает их автоматически

Эти атрибуты относятся к последнему использовавшемуся в программе неявному курсору, независимо от того, в каком блоке этот курсор использовался. До выполнения в программе первой команды

PL/SQL с использованием неявного курсора атрибуты курсора с именем SQL% остаются неинициализированными (имеют значения UNKNOWN и NULL).

Наиболее часто используются атрибуты SQL%FOUND и SQL%ROWCOUNT, которые позволяют получить информацию о результатах обработки данных — сколько строк было обработано (добавлено, изменено или удалено) и были ли они вообще.

Приведем пример использования атрибутов неявных курсоров.

```
SQL> DECLARE
  2    l_at1 tab1.at1%TYPE;
  3  BEGIN
  4
  5    l_at1 := 1;
  6
  7    INSERT INTO tab1 VALUES (l_at1);
  8    DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT);
  9
 10    INSERT INTO tab1 SELECT * FROM tab1;
 11    DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT);
 12
 13    INSERT INTO tab1 SELECT * FROM tab1;
 14    DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT);
 15
 16    UPDATE tab1 SET at1=2 WHERE at1=1;
 17    IF SQL%FOUND THEN
 18      DBMS_OUTPUT.PUT_LINE('Строки изменялись');
 19    ELSE
 20      DBMS_OUTPUT.PUT_LINE('Строки не изменялись');
 21    END IF;
 22
 23  END;
 24  /
1
1
2
Строки изменялись
PL/SQL procedure successfully completed.
```

Возможность с помощью атрибутов неявных курсоров определять, были ли изменены данные после выполнения предложений SQL, используется, например, при реализации в программах PL/SQL оптимистической стратегии многопользовательского доступа.

Рассмотрим еще две возможности языка PL/SQL: конструкцию RETURNING и использование записей PL/SQL в DML-командах. Эти

возможности наглядно иллюстрируют удобство использования языка PL/SQL при работе с базами данных Oracle.

Конструкция RETURNING

Конструкция RETURNING позволяет получить новые значения данных в таблицах после их добавления или изменения. Например, после увеличения оклада сотрудника на 10% в дальнейших вычислениях в коде может понадобиться новое значение оклада. Конечно, можно сразу после изменения выполнить выборку данных по этому сотруднику, но это будет еще одна операция, на которую потребуются дополнительные расходы ресурсов. Конструкция RETURNING позволяет их избежать.

Конструкцию RETURNING часто используют для получения значения первичного ключа после добавления новой строки в таблицу с использованием последовательности.

```
SQL> CREATE SEQUENCE sq1 START WITH 1 INCREMENT BY 2;
Sequence created.

SQL> DECLARE
  2    l_at1 tab1.at1%TYPE;
  3  BEGIN
  4    INSERT INTO tab1 VALUES(sq1.NEXTVAL) RETURNING at1 INTO l_at1;
  5    DBMS_OUTPUT.PUT_LINE(l_at1);
  6    INSERT INTO tab1 VALUES(sq1.NEXTVAL) RETURNING at1 INTO l_at1;
  7    DBMS_OUTPUT.PUT_LINE(l_at1);
  8  END;
  9  /
1
3
PL/SQL procedure successfully completed.
```

Использование записей PL/SQL в DML-командах

В DML-командах языка PL/SQL можно использовать и параметры-записи PL/SQL:

- для указания того, что в команде UPDATE следует изменить целиком строку таблицы, используется ключевое слово ROW;
- в команде INSERT после ключевого слова VALUES вместо списка переменных скалярных типов со значениями всех столбцов добавляемой строки указывается одна переменная-запись PL/SQL, которая целиком «укладывается» в таблицу в виде новой строки.

```
SQL> CREATE TABLE tab1 (at1 INTEGER, at2 VARCHAR2(1));

Table created.

SQL> DECLARE
  2    l_tab1 tab1%ROWTYPE;
  3  BEGIN
  4    l_tab1.at1 := 1;
  5    l_tab1.at2 := 'a';
  6    INSERT INTO tab1 VALUES l_tab1;
  7    l_tab1.at1 := 2;
  8    l_tab1.at2 := 'b';
  9    INSERT INTO tab1 VALUES l_tab1;
 10    l_tab1.at2 := 'c';
 11    UPDATE tab1 SET ROW = l_tab1 WHERE at1=2;
 12  END;
 13  /

PL/SQL procedure successfully completed.

SQL> SELECT * FROM tab1;
       AT1 AT2
---------- ----
         1 a
         2 c
```

Рекомендуется используемые в DML-командах записи PL/SQL объявлять на основе схем таблиц с помощью атрибута %ROWTYPE. Если впоследствии схемы этих таблиц изменятся, то код PL/SQL останется работоспособным. Таким образом, использование в DML-командах одной записи PL/SQL вместо нескольких переменных скалярных типов приводит к тому, что код становится более компактным и повышается его надежность.

Формирование предложений SQL со связываемыми переменными

Ранее отмечалось, что достоинством языка PL/SQL является формирование компилятором предложений SQL со связываемыми переменными.

Напомним, что связываемой переменной (bind variable) называется метка (placeholder) для переменной в тексте предложения SQL. Перед выполнением предложения SQL происходит связывание переменных (binding variables) — для них задаются фактические значения.

Мы сейчас не будем вдаваться в подробности, скажем только, что использование в предложениях SQL связываемых переменных вместо жестко кодируемых литералов (hard-coded literals) является обязательным условием достижения высокой производительности сервера Oracle.

Выполним программу PL/SQL с тремя командами добавления строк в таблицу `test_tab` и посмотрим, какие SQL-предложения `INSERT` были сформированы компилятором PL/SQL для байт-кода и потом выполнены в базе данных виртуальной машиной PL/SQL на самом деле[1]:

```
CREATE TABLE test_tab (a INTEGER)

SQL> DECLARE
  2    l_value INTEGER;
  3  BEGIN
  4    INSERT INTO test_tab VALUES(123);
  5    l_value := 124;
  6    INSERT INTO test_tab VALUES(l_value);
  7    l_value := 125;
  8    INSERT INTO test_tab VALUES(l_value);
  9  END;
 10    /

PL/SQL procedure successfully completed.

SQL> SELECT SUBSTR(sql_text,1,70) AS SQL_TEXT FROM V$SQL
  2  WHERE LOWER(sql_text) LIKE LOWER('%test_tab%');
```

[1] В представлении словаря-справочника данных `V$SQL` отображается статистика выполненных в базе данных предложений SQL.

```
SQL_TEXT
------------------------------------------------------------------
INSERT INTO TEST_TAB VALUES(123)
INSERT INTO TEST_TAB VALUES(:B1 )
```

Видно, что для двух команд INSERT с переменной l_value компилятором PL/SQL сформировано одно предложение SQL со связываемой переменной :B1, потом оно было дважды выполнено с разными привязанными значениями :B1 (124 и 125). Для жестко закодированного (hard coded) литерала 123 замена на связываемую переменную компилятором PL/SQL не производилась[1].

Управление транзакциями в PL/SQL

Транзакции в базах данных Oracle

Транзакцией в базе данных Oracle называется атомарная (неделимая) логическая единица (unit) работы с базой данных, состоящая из одного или нескольких предложений языка SQL. Все транзакции в базах данных Oracle обладают четырьмя основными свойствами транзакций в базах данных:

- атомарность (atomcity) — свойство транзакции, заключающееся в том, что она не может быть выполнена частично: транзакция либо успешно завершается с сохранением всех изменений в данных, либо все без исключения внесенные ею изменения данных полностью отменяются и измененные данные возвращаются к тому состоянию, в котором они находились перед началом транзакции;
- согласованность (consistency) — свойство транзакции, заключающееся в том, что транзакция переводит базу данных из одного согласованного состояния в другое согласованное состояние;
- изолированность (isolation) — свойство транзакции, заключающееся в ограничении влияния на ее работу других транзакций;

[1] Вообще говоря, сервер Oracle умеет и литералы-параметры заменять на связываемые переменные уже в ходе обработки предложений SQL. Эта замена включается параметром экземпляра CURSOR_SHARING=TRUE.

- сохраняемость (durability) — свойство транзакции, обеспечивающее сохранность сделанных ею изменений данных после фиксации транзакции, независимо от программных сбоев и отказов оборудования.

Транзакции, обладающие всеми этими свойствами, называются ACID-транзакциями (по первым буквам английских названий свойств). Атомарность (неделимость) — главное свойство транзакции, наличие которого требуется ее определением. Все без исключения изменения транзакции или должны сохраняться в базе данных или же должны полностью отменяться.

Транзакция в Oracle начинается с первого модифицирующего данные предложения SQL (`INSERT`, `UPDATE`, `DELETE`), выполненного после окончания предыдущей транзакции, то есть предложения `SELECT` транзакцию не начинают[1]. После своего начала транзакция может находиться в одном из трех состояний:

- активная транзакция (active transaction) — транзакция, которая начата, но и не зафиксирована и для нее не выполнена отмена;
- зафиксированная транзакция (committed transaction) — транзакция, для которой все выполненные ею изменения в данных являются постоянными (permanent);
- отмененная транзакция[2] (rolled back transaction) — транзакция, для которой все выполненные ею изменения в данных отменены (все измененные данные возвращены в исходное состояние, в котором они находились перед началом транзакции).

Активная транзакция в каждый момент времени характеризуется степенью своей активности:

---

[1] В Oracle нет SQL-команды вроде `START TRANSACTION`, имеющейся в PostgreSQL. Транзакции в Oracle начинаются неявно, без выполнения специальных команд.

[2] Вообще говоря, вместо «отмена транзакции» более правильно говорить «откат транзакции», так как соответствующее действие именуется rollback transaction. Однако выражение «откаченная транзакция» звучит хуже «отмененной» и не прижилось. Далее всюду в тексте книги будем отменять транзакции, а не их откатывать. Выражению «отмена до точки сохранения» будет соответствовать rollback to savepoint.

- выполняющаяся транзакция (running transaction) — транзакция, в рамках которой в этот момент времени выполняется одно из ее предложений SQL;
- простаивающая транзакция (idle transaction).

Простаивающую активную транзакцию, все предложения SQL которой выполнены, но фиксации или отмены пока еще не было, называют выполненной. Фиксацию транзакции осуществляет SQL-команда `COMMIT`, отмену — SQL-команда `ROLLBACK`. Зафиксированная или отмененная транзакция называется завершенной (completed). После того, как транзакция завершилась, в этой пользовательской сессии следующее модифицирующее данные предложение SQL начинает следующую транзакцию.

В Oracle минимальной единицей данных, которая может быть изменена транзакцией, является строка. Получается, что в каждый момент времени любая строка любой таблицы базы данных может находиться в одном из двух состояний:

- отсутствует активная транзакция, изменившая строку (то есть когда-то ранее строка была добавлена в базу данных какой-то давнишней активной транзакцией, потом строка, возможно, изменялась предложениями `UPDATE` других активных транзакций, но все эти транзакции в свое время были зафиксированы или отменены);
- строка изменена активной транзакцией (неважно, выполняющейся или выполненной — главное, что строка изменена, и эти изменения не зафиксированы и не отменены).

В литературе на русском языке часто встречаются фразы вроде «Запрос видит только зафиксированные данные на SCN 10023», представляющие собой кальковый перевод фраз вида «The query only sees committed data with respect to SCN 10023». Дело в том, что более правильные с точки зрения терминологии выражения выглядят довольно громоздко. Мы тоже будем дальше использовать эти ставшие общепринятыми понятия и выражения, просто приведем их определения:

- зафиксированными данными называются данные, измененные зафиксированной транзакцией (изменения данных стали постоянными);
- предложение SQL «видит» строки, если они используются (не игнорируются) им в ходе своего выполнения — строки участ-

вуют в определении истинности критерия отбора из фразы WHERE и могут попадать в результирующую выборку.

Обращение к данным сервер Oracle осуществляет в одном из двух режимов[1]:

- обращение в текущем режиме (current mode, db block gets), при котором происходит обращение к строкам в их текущем состоянии, то есть в состоянии, в котором они находятся прямо сейчас (right now);
- обращение в режиме согласованного чтения (consistent read mode, consistent gets), при котором происходит обращение к строкам, находящимся по состоянию на некоторый момент времени, то есть к предыдущим версиям строк.

В Oracle любой SQL-запрос SELECT видит только зафиксированные данные по состоянию на начало своего выполнения. Если другие транзакции уже после начала выполнения SQL-запроса сделают изменения в данных, и даже если эти изменения будут зафиксированы, то все равно SQL-запрос увидит версию данных по состоянию на момент начала его выполнения. Это и называется «согласованным чтением» (consistency read) на уровне отдельных предложений SQL. Часто версии данных по состоянию на некоторый момент времени в прошлом называют старыми данными.

Все сказанное верно и для критерия отбора из конструкции WHERE предложений DELETE, UPDATE и INSERT SELECT. Для удаления и изменения отбираются строки по состоянию на момент начала выполнения этих предложений.

«Под капотом» сервера Oracle для обеспечения корректного выполнения транзакций, согласованного чтения и вот этого всего имеются следующие специальные механизмы:

- сегменты отката, обеспечивающие версионность данных;
- система блокировок.

---

[1] Доступ в Oracle на самом деле осуществляется не к отдельным строкам, а к целым блокам, в которых на физическом уровне размещаются строки. Для описания языка программирования PL/SQL это обстоятельство не существенно.

Блокировки

Наличие системы блокировок позволяет обеспечить корректное изменение одних и тех же данных.

Блокировкой (lock) называется средство организации доступа к совместно используемому ресурсу. Наложить на ресурс блокировку — это ограничить возможности других по работе с этим ресурсом. В Oracle есть несколько видов блокировок, мы рассмотрим, как работают самые распространенные блокировки строк таблиц транзакциями (блокировки TX).

Общие правила работы блокировок TX в Oracle выглядят так:

- блокировки накладывают и снимают транзакции;
- наложенную транзакцией блокировку может снять только она сама в ходе завершения транзакции командами `COMMIT` или `ROLLBACK`;
- блокировки накладываются на строки таблиц (одна блокировка может накладываться на несколько строк сразу);
- чтобы изменить или удалить строку, ее сначала надо заблокировать;
- заблокированные одной транзакцией строки другая транзакция не может ни заблокировать, ни тем более изменить или удалить;
- существует специальная форма SQL-запроса `SELECT FOR UPDATE`, которая накладывает блокировки на отбираемые по критерию отбора строки;
- блокировки являются атрибутами самих строк — чтобы определить, есть ли на строке блокировка, к ней необходимо обратиться.

Для новых строк, добавленных в таблицу предложением `INSERT`, блокировка TX накладывается транзакцией, выполнившей это предложение.

При изменении строк предложением `UPDATE` к ним обращаются дважды:

- поиск строк-кандидатов для изменений по условию в конструкции `WHERE`, который осуществляется в режиме согласованного чтения;

- собственно изменение найденных строк (строки находятся в текущем состоянии — они или зафиксированы когда-то выполнившимися транзакциями и сейчас с ними никто не работает, или заблокированы активными транзакциями).

Из сказанного следует, что если какая-то подходящая по условию WHERE строка уже после начала выполнения нашего UPDATE будет кем-то добавлена в таблицу и зафиксирована, то так как поиск строк по WHERE осуществляется в режиме согласованного чтения, эта новая строка будет проигнорирована и менять ее наш UPDATE не будет. Для предложения DELETE аналогичное поведение — добавленные после начала его выполнения строки как кандидаты на удаление не находятся (тоже действует режим согласованного чтения на начало выполнения DELETE).

Когда наш процесс приступает к собственно изменению или удалению строки, найденной по критерию отбора в конструкции WHERE, сначала он попытается наложить на эту строку блокировку ТХ. При этом возможны два исхода в зависимости от того, есть ли на этой строке блокировка, установленная другой (чужой) активной транзакцией (наличие или отсутствие такой блокировки определяется в ходе обращения к строке):

- если на строке нет чужой блокировки, то наша транзакция накладывает на нее свою блокировку;
- если выясняется, что на строке есть чужая блокировка, то наш процесс переходит в режим ожидания с подпиской на сообщение о снятии чужой блокировки (дальше по таблице наша транзакция не идет, «повисает» на заблокированной чужой транзакцией строке).

В Oracle вся схема обработки изменений построена исходя из оптимистичного предположения, что в большинстве случаев чужих блокировок на строках не будет, то есть сам процесс изменения, как правило, происходит по схеме «нашел строку — заблокировал ее — изменил — стал искать следующую строку».

После успешного наложения блокировки на строку, наша транзакция изменяет значения ее столбцов в соответствии с тем, что написано в конструкции SET предложения UPDATE. После изменения строки наша блокировка на ней остается. Процесс пойдет по таблице дальше, отбирая строки по критерию из WHERE в режиме согласованного чтения, блокируя и меняя их в текущем режиме. В конце кон-

цов, таблица кончится, и процесс оставит за собой ее измененные строки с блокировкой на каждой. Так будет для каждого входящего в нашу транзакцию предложения SQL INSERT, UPDATE, DELETE. Никакие другие транзакции изменять заблокированные нами строки не смогут — при своих попытках заблокировать строки пере изменением все они будут переходить в режим ожидания с подпиской на оповещения о снятии наших блокировок. Эти оповещения придут к ним, как только наша транзакция будет зафиксирована или отменена и блокировки снимутся.

В свою очередь, если сами будем ждать возможности установить блокировку из-за чужой транзакции, то рано или поздно чужая активная транзакция будет завершена, в ходе фиксации или отмены все ее блокировки будут сняты. Наш процесс получит об этом сообщение и «отвиснет» для того, чтобы повторно попытаться заблокировать и после определенных проверок изменить строку, на которой он «висел» в ожидании снятия со строки блокировки.

Уровни изоляции транзакции

Каждая транзакция в Oracle выполняется на заданном уровне изоляции, который не меняется на всем протяжении жизни транзакции. Уровень изоляции (isolation level) определяется как степень исключения взаимного влияния транзакций друг на друга.

Всего в Oracle поддерживается три уровня изоляции транзакций:

- чтение зафиксированных данных (read committed) — любое выполняемое в транзакции предложение SQL видит только зафиксированные данные;
- только чтение (read only) — в транзакции допускаются только SQL-запросы SELECT, которые видят только зафиксированные данные по состоянию на начало транзакции (все предложения INSERT, UPDATE и DELETE на этом уровне изоляции завершаются с ошибкой);
- упорядоченность выполнения (serializable) — транзакция полностью изолируются от других транзакций, она выполняется так, как будто параллельных транзакций не существует (если имеется возможность такого выполнения).

Уровень изоляции read only используется, как правило, в отчетных системах для того, чтобы результаты нескольких подряд выполняющихся SQL-запросов соответствовали одному и тому же состоянию

базы данных, которое было на начало транзакции. Этот сценарий для программ PL/SQL используется не часто. Использование в программах PL/SQL уровня изоляции serializable на практике встречается тоже очень редко. По этим причинам мы не будем рассматривать уровни изоляции read only и serializable.

Уровень чтения зафиксированных данных (read committed) используется в Oracle по умолчанию. То есть, если не начать транзакцию командой `SET TRANSACTION` с указанием одного из двух других уровней изоляции, то будет использоваться именно read commited. Этот уровень изоляции транзакций вполне соответствует представлениям на бытовом уровне о параллельной работе нескольких людей с одними и теми же данными:

- активно вносишь изменения (находишься в процессе), но пока не зафиксировал и не отменил их — пусть пока другие люди видят старые (неизмененные) версии данные, мало ли что;
- внес изменения и зафиксировал их — пусть их увидят другие люди;
- внес изменения и потом их отменил — другим людям такие изменения видеть абсолютно незачем.
- естественно, что сам свои текущие изменения видишь всегда и без их подтверждения.

По последнему пункту почему-то иногда у обучаемых возникает недопонимание. При вопросе «В столбце тройка, меняю на четверку, транзакцию не фиксирую, делаю `SELECT` — что увижу?» бывает так, что мнения учебной группы разделяются. Находятся и те, кто считают, что будет тройка. Вообще говоря, на бытовом уровне было бы странно что-то поменять и потом самому этого при следующем обращении не увидеть.

Отдельный SQL-запрос в ходе своего выполнения не испытывает воздействия со стороны других транзакций (не видит внесенные ими изменения), а вот состоящая из таких операторов вся транзакция с уровнем изоляции read commited согласно определению уровня изоляции такому воздействию подвержена. Если в ходе транзакции один SQL-запрос выполнялся и проверял свой критерий отбора на одном «срезе» базы данных из зафиксированных данных, то через некоторое время в этой же транзакции другой SQL-запрос может увидеть отличающийся «срез» с изменениями, сделанными за прошедшее время другими зафиксированными транзакциями. Логично

— были сделаны подтвержденные изменения, их всем следует увидеть. Результирующая выборка будет при втором выполнении другой. Для иллюстрации всего вышесказанного рассмотрим поведение параллельно выполняющихся транзакций в режиме изоляции read commited, изменяя и читая строки в таблице `test`:

SQL*Plus первой сессии	t	SQL*Plus второй сессии
Начальное заполнение: SQL> SELECT * FROM test; AT1 A ---------- - 1 a		**Начальное заполнение:** SQL> SELECT * FROM test; AT1 A ---------- - 1 a
Пример 1: -- начало транзакции SQL> UPDATE test set at1=2; 1 row updated. -- свои изменения видны -- (в т.ч.) незафиксированные SQL> SELECT * FROM test; AT1 A ---------- - **2** a -- транзакция фиксируется SQL> COMMIT; Commit complete. -- тем более видны изменения SQL> SELECT * FROM test; AT1 A ---------- - **2** a	t0 t1 t2 t3	 -- изменения чужой активной -- транзакции не видны SQL> SELECT * FROM test; AT1 A ---------- - **1** a -- изменения зафиксированной -- транзакции стали видны SQL> SELECT * FROM test; AT1 A ---------- - **2** a
Пример 2: -- начало транзакции SQL> UPDATE test set at1=3; 1 row updated. -- свои изменения видны -- (в т.ч.) незафиксированные SQL> SELECT * FROM test; AT1 A ---------- - **3** a -- транзакция отменяется SQL> ROLLBACK; Rollback complete. -- отмененные изменения не видны SQL> SELECT * FROM test; AT1 A ---------- - **2** a	t0 t1 t2 t3	 -- изменения чужой активной -- транзакции не видны SQL> SELECT * FROM test; AT1 A ---------- - **2** a -- о том, что были какие-то -- отмененные изменения, -- никто и не узнает никогда SQL> SELECT * FROM test; AT1 A ---------- - **2** a

Пример 3:		
-- начало транзакции в сессии 1		-- начало транзакции в сессии 2
SQL> UPDATE test set at2=4;	t0	-- серверный процесс переходит
1 row updated.		-- в режим ожидания снятия блокировки
		-- SQL*Plus «повисает»
	t1	SQL> UPDATE test set at2=5;
-- транзакция фиксируется		
-- блокировка со строки снимается		-- получаем сообщение, «отвисаем»
SQL> COMMIT;	t2	-- блокируем строку и меняем ее
Commit complete.	t3	1 row updated.
-- видна зафиксированная строка		-- видны свои изменения
SQL> SELECT * FROM test;	t4	SQL> SELECT * FROM test;
AT1 A		AT1 A
--------- -		--------- -
4 a		**5** a

Транзакции и средства работы с ними являются довольно сложно понимаемым учебным материалом. В том числе, это вызвано и тем, что в книгах по теории баз данных по этой теме написано одно, в разных имеющихся на рынке СУБД (Oracle, Microsoft SQL Server, IBM DB2) реализовано другое, причем в книгах про эти СУБД описано далеко не все. Для понимания того, как же все это устроено и работает в Oracle, автор настоятельно рекомендует прочитать подряд идущие главы «Блокировка и защелкивание данных», «Параллелизм и многоверсионность», «Транзакции», «Повтор и отмена» книги Томаса Кайта «Oracle для профессионалов. Архитектура, методики программирования и основные особенностей версий 9i, 10g, 11g и 12c» — всего около двухсот страниц мелким шрифтом. Текст Кайта предельно понятен и реально просветляет.

Команды управления транзакциями

В языке PL/SQL имеются следующие команды для управления транзакциями, соответствующие аналогичным предложениям SQL (напомним, что по этим командам компилятор PL/SQL размещает в байт-коде предложения SQL):

- SET TRANSACTION — устанавливает уровень изоляции транзакции;
- COMMIT — фиксирует транзакцию (сохраняет все внесенные транзакцией изменения данных и снимает все наложенные транзакцией блокировки);
- ROLLBACK — отменяет транзакцию (отменяет все внесенные транзакцией изменения данных и снимает все наложенные транзакцией блокировки);

- SAVEPOINT — устанавливает точку сохранения (точкой сохранения называется именованный номер изменения в транзакции, до которого может быть выполнена отмена изменений)[1];
- ROLLBACK TO SAVEPOINT — отменяет все изменения, внесенные транзакцией после установки указанной точки сохранения и снимает блокировки (сама транзакция при этом остается активной, то есть является транзакцией, которая начата, но и не зафиксирована и для нее не выполнена отмена);
- LOCK TABLE — блокирует указанную таблицу в заданном режиме.

Приведем пример использования команд для управлениями транзакциями в PL/SQL.

```
CREATE TABLE tab3 (at1 INTEGER);
INSERT INTO tab3 VALUES(7);
```

Сначала запускаем анонимный блок в SQL*Plus первой сессии, она «засыпает» на десять секунд («засыпание» обеспечивает процедура SLEEP встроенного пакета DBMS_LOCK). Пока это время не прошло, переключаемся в SQL*Plus второй сессии и запускаем другой анонимный блок, который отрабатывает мгновенно. Через десять секунд «проснется» первая сессия.

SQL*Plus первой сессии	SQL*Plus второй сессии

SQL> DECLARE
  2 l_at1 tab3.at1%TYPE;
  3 BEGIN
  4 SET TRANSACTION READ ONLY;
  5 DBMS_LOCK.sleep(10);
  6 SELECT at1 INTO l_at1 FROM tab3;
  7 DBMS_OUTPUT.PUT_LINE(l_at1);
  8 COMMIT;
  9 END;
 10 /
7
 	
PL/SQL procedure successfully completed
	SQL> DECLARE
	  2 l_at1 tab3.at1%TYPE;
	  3 BEGIN
	  4 UPDATE tab3 SET at1=8;
	  5 COMMIT;
	  6 SELECT at1 INTO l_at1 FROM tab3;
	  7 DBMS_OUTPUT.PUT_LINE(l_at1);
	  8 END;
	  9 /
	8
	
	PL/SQL procedure successfully completed

[1] Работа с точками сохранения подробно рассматривается далее.

Видно, что выставленный для первой транзакции уровень изоляции READ ONLY обеспечил чтение старой версии данных (прочитана семерка), несмотря на то, что данные до их чтения изменены (на восьмерку) зафиксированной транзакцией второй сессии.

Следует помнить, что транзакция состоит из предложений SQL. Вызовы программ на PL/SQL внутри транзакции следует рассматривать как промежуточные. Можно в SQL*Plus начать транзакцию выполнением SQL-предложения UPDATE, после него вызвать программу на PL/SQL, которая выполнит 5 предложений SQL из своего байт-кода, потом подождать полчаса, потом выполнить еще пару SQL-предложений INSERT, потом опять вызвать программу PL/SQL. Все это время транзакция будет являться активной и с точки зрения сервера выполнит 1+5+2+5 предложений SQL. То, что часть из них была выполнена из программ PL/SQL, значения не имеет. Зафиксировать или отменить транзакцию также можно как в программе на PL/SQL, так и в «чистом» SQL.

Точки сохранения для предложений SQL

Выполнение предложений SQL сопровождается установкой ядром Oracle неявных точек сохранения (savepoints) перед каждым предложением по следующей трехэтапной схеме:

```
неявно SET SAVEPOINT implicit_savepoint;
выполняется предложение SQL, например, UPDATE;
IF SQLerror THEN отмена до implicit_savepoint;
```

При отмене до неявно установленной точки сохранения отменяются изменения, которые сделаны в ходе обработки предложения SQL до возникновения ошибки. Например, пусть предложение UPDATE изменило две строки, а на третьей произошла ошибка — нарушено ограничение целостности.

```
CREATE TABLE tab7 (at1 INTEGER CHECK (at1<=4));
INSERT INTO tab7 VALUES (2);
INSERT INTO tab7 VALUES (3);
INSERT INTO tab7 VALUES (4);

-- установка неявной точки сохранения
SQL> UPDATE tab7 SET at1=at1+1;
UPDATE tab7 SET at1=at1+1
*
ERROR at line 1:
ORA-02290: check constraint (U1.SYS_C0012475) violated
```

```
       AT1             пояснение
----------
         2             +1 = 3 (3<=4, OK)
         3             +1 = 4 (4<=4, OK)
         4             +1 = 5 (5>4,  Error, отмена до точки сохранения)
```

Ошибки предложений SQL в транзакции

В языке SQL в рамках одной транзакции изменения, внесенные одними предложениями SQL, не отменяются из-за ошибок других предложений SQL. Если из SQL*Plus в рамках одной транзакции выполнить пять предложений INSERT, из которых два завершатся ошибкой, то в таблице все равно будет три новые строки. Успешное добавление этих трех строк не отменится, они не пропадут, а останутся «изменениями, внесенными активной транзакцией».

```
SQL> CREATE TABLE transaction_test (a INTEGER);
Table created.

SQL> INSERT INTO transaction_test VALUES(1);
1 row created.

SQL> INSERT INTO transaction_test VALUES(2);
1 row created.

SQL> INSERT INTO transaction_test VALUES(3/0);
INSERT INTO transaction_test VALUES(3/0)
                                    *
ERROR at line 1:
ORA-01476: divisor is equal to zero

SQL> INSERT INTO transaction_test VALUES(4);
1 row created.
SQL> INSERT INTO transaction_test VALUES(5/0);
INSERT INTO transaction_test VALUES(5/0)
ERROR at line 1:
ORA-01476: divisor is equal to zero

SQL> SELECT * FROM transaction_test;
         A
----------
         1
         2
         4
```

В SQL*Plus возникновение ошибок не влияет на статус транзакции, она остается активной и ее можно будет зафиксировать или отменить. Это же верно и для Quest SQL Navigator. Для других про-

грамм, выполняющих транзакции в Oracle, действия при возникновении ошибок могут отличаться. Например, для прикладного программного обеспечения (каких-нибудь бухгалтерских программ с GUI на C#, скриптов обсчета данных в базе на Python) программисты часто согласно требованиям бизнес-логики предусматривают отмену транзакции при возникновении первой же ошибки выполнения предложения SQL в ходе транзакции.

Точки сохранения для вызовов PL/SQL

Для вызовов программ PL/SQL внутри транзакции неявная точка сохранения также устанавливается перед передаваемым на выполнение блоком PL/SQL и в случае завершения выполнения блока с ошибкой, все внесенные им изменения данных будут автоматически отменены. То есть в этой части вызываемые блоки PL/SQL обрабатываются аналогично предложениям SQL.

Рассмотрим две ситуации

```
CREATE TABLE tab4 (a INTEGER);
```

Ситуация 1: необработанное исключение

```
-- все в рамках одной транзакции

-- первый блок (успешно)
SQL> BEGIN
  2   INSERT INTO tab4 VALUES(1);
  3   INSERT INTO tab4 VALUES(2);
  4  END;
  5  /

PL/SQL procedure successfully completed.

-- второй блок (с ошибкой)
SQL> BEGIN
  2   INSERT INTO tab4 VALUES(3);
  3   INSERT INTO tab4 VALUES('abc');
  4  END;
  5  /
BEGIN
*
ERROR at line 1:
ORA-01722: invalid number
ORA-06512: at line 3

SQL>  SELECT * FROM tab4;
         A
----------
         1
         2
```

Ситуация 2: обработанное исключение

```
-- все в рамках одной транзакции

-- первый блок (успешно)
SQL> BEGIN
  2   INSERT INTO tab4 VALUES(1);
  3   INSERT INTO tab4 VALUES(2);
  4  END;
  5  /

PL/SQL procedure successfully completed.

-- второй блок (успешно)
SQL> BEGIN
  2   INSERT INTO tab4 VALUES(3);
  3   INSERT INTO tab4 VALUES('abc');
  4  EXCEPTION
  5    WHEN OTHERS THEN NULL;
  6  END;
  7  /
PL/SQL procedure successfully completed.


SQL> SELECT * FROM tab4;
         A
----------
         1
         2
         3
```

В первой ситуации (есть блок с необработанным исключением):

- для второго блока PL/SQL при ошибке выполнения команды `INSERT` со значением `abc` произошла отмена до неявной точки сохранения перед `INSERT` и изменения, внесенные этим предложением, были отменены;
- так как во втором блоке PL/SQL нет раздела обработки исключений, то с ошибкой завершился весь блок и произошла отмена к неявно установленной точке сохранения перед ним (в результате была отменено успешное добавление тройки на второй строке кода этого блока);
- первый блок PL/SQL выполнился без ошибок; последовавшие потом ошибки второго блока, как и должно быть, никак на внесенные его командами изменения данных не повлияли и обе добавленные в ходе обработки первого блока строки есть в таблице (единица и двойка).

Во второй ситуации (исключение обработано):

- точно так же для второго блока PL/SQL при ошибке выполнения предложения `INSERT` со значением `abc` произошла отмена к неявной точке сохранения перед `INSERT` и изменения, внесенные этим предложением, были отменены;
- так как во втором блоке есть раздел обработки исключений с `OTHERS`-обработчиком, то вызов второго блока завершился успешно, без передачи ошибки вызывающей среде, поэтому добавление тройки не отменялось;
- после выполнения обоих блоков в таблице `tab4` будет три строки — две от первого блока и одна от второго.

Использование именованных точек сохранения

До неявно устанавливаемых внутри активных транзакций точкам сохранения отмена автоматически осуществляется ядром сервера Oracle после неуспешных программных вызовов. До явно устанавливаемых программистом именованным точкам сохранения отмена осуществляется согласно логике обработки ошибок и нестандартных ситуаций. Поэтому отмена до таких точек сохранения тоже должна явно инициироваться программистом. Можно сказать, что точки

сохранения своими именами «размечают» активную транзакцию на участки, изменения на которых есть возможность отменить.

Приведем правила работы с именованными точками сохранения:

- область видимости точки сохранения не ограничивается блоком PL/SQL, в котором она установлена (точки сохранения «живут» на уровне всей транзакции, в частности, вообще можно установить точку сохранения в PL/SQL, а выполнить отмену до нее в SQL и наоборот);
- если в ходе транзакции имя точки сохранения используется повторно, эта точка сохранения просто передвигается вперед по транзакции;
- после отмены до точки сохранения отменяются изменения данных, сделанные после установки этой точки, также снимаются наложенные после ее установки блокировки (блокировки и изменения данных, сделанные транзакцией до точки сохранения, остаются);
- транзакция после отмены до точки сохранения остается активной, ее можно возобновить (начать выполнять новые предложения SQL) и потом зафиксировать или отменить;
- при отмены до точки сохранения все установленные после нее другие точки сохранения становятся недоступными, но сама точка сохранения, к которой была отмена, остается (это означает, что с нее можно возобновить транзакцию и при необходимости снова выполнить отмену до нее же — получается определенная «точка опоры» внутри транзакции).

В качестве примера работы с точками сохранения рассмотрим программу, реализующую следующую бизнес-логику.

Имеется таблица с тремя необработанными заданиями и две пустые таблицы для сохранения результатов обработки заданий:

```
CREATE TABLE jobs (id INTEGER,state VARCHAR2(10),a INTEGER,b INTEGER);
INSERT INTO jobs VALUES (1,'created',8,4);
INSERT INTO jobs VALUES (2,'created',4,0);
INSERT INTO jobs VALUES (3,'created',15,5);

CREATE TABLE jobs_mult_res (id INTEGER, result NUMBER);
CREATE TABLE jobs_div_res (id INTEGER, result NUMBER);
```

Обработка задания заключается в умножении и делении двух чисел из столбцов a и b таблицы jobs с сохранением результатов в двух таблицах. После успешного выполнения и умножения и деления задание должно менять состояние на processed. Видно, что попытка деления для задания из второй строки jobs должна завершиться ошибкой деления на ноль (4/0). Ошибочные задания должны получать статус error. Для ошибочных заданий не должно быть строк ни в одной из двух таблиц результатов.

Напрашивается следующая реализация: в начале каждой итерации цикла по заданиям будем устанавливать точку сохранения, до которой при обработке исключений и будем осуществлять отмену. Устанавливаемая точка сохранения будет перемещаться вперед по транзакции, позволяя отменить только изменения, внесенные в ходе обработки последнего (ошибочного) задания.

```
SQL> BEGIN
  2    FOR job_rec IN (SELECT * FROM jobs WHERE state='created') LOOP
  3      BEGIN
  4        SAVEPOINT sp_job;
  5        INSERT INTO jobs_mult_res VALUES(job_rec.id,job_rec.a*job_rec.b);
  6        DBMS_OUTPUT.PUT_LINE('Insert mult for job '||job_rec.id||' OK');
  7        INSERT INTO jobs_div_res VALUES(job_rec.id,job_rec.a/job_rec.b);
  8        DBMS_OUTPUT.PUT_LINE('Insert  div for job '||job_rec.id||' OK');
  9        UPDATE jobs SET state='processed' WHERE id=job_rec.id;
 10      EXCEPTION
 11        WHEN OTHERS THEN
 12          ROLLBACK TO sp_job;
 13          UPDATE jobs SET state='error' WHERE id=job_rec.id;
 14      END;
 15    END LOOP;
 16  END;
 17  /
Insert mult for job 1 .. OK
Insert  div for job 1 .. OK
Insert mult for job 2 .. OK
Insert mult for job 3 .. OK
Insert  div for job 3 .. OK
PL/SQL procedure successfully completed.

SQL> SELECT  j.*,jmr.result mult,jdr.result div FROM jobs j
  2  LEFT OUTER JOIN jobs_mult_res jmr ON j.id=jmr.id
  3  LEFT OUTER JOIN jobs_div_res jdr ON j.id=jdr.id
  4  ORDER BY j.id;
        ID STATE               A          B       MULT        DIV
---------- ---------- ---------- ---------- ---------- ----------
         1 processed           8          4         32          2
         2 error               4          0
         3 processed          15          5         75          3
```

Видно, что для второго задания было успешно выполнено умножение, однако при делении произошла ошибка и инициировано системное исключение. В OTHERS-обработчике была выполнена отмена до точки сохранения, выставленной в начале итерации цикла, поэтому успешное добавление строки с результатом умножения в таблицу jobs_mult_res было отменено. Об этом говорят пустые значения в обоих столбцах mult и div выборки, которые получаются в результате внешнего соединения.

Автономные транзакции

В PL/SQL существует еще одна интересная возможность при работе с транзакциями — блок PL/SQL можно объявить автономной транзакцией, независимой от основной транзакции. На время выполнения блока, объявленного автономной транзакцией, вызвавшая его основная транзакция приостанавливается и возобновляется после фиксации или отмены автономной транзакции. Основная транзакция и все ее автономные транзакции фиксируются и отменяются независимо. Из этого свойства автономных транзакций следуют типичные случаи их применения:

- ведение журналов изменений данных;
- изменение данных в функциях PL/SQL, вызываемых в SQL.

Для объявления блока PL/SQL автономной транзакцией достаточно указать директиву компилятору PRAGMA AUTONOMOUS_TRANSACTION.

Ведение журнала изменений данных

Довольно часто в программах PL/SQL встречается реализация журналов изменений данных, чтобы в случае разбирательств можно было узнать, кто выполнил недозволенную операцию, например, закрыл лицевой счет VIP-клиента, когда все сотрудники говорят «это не я». Такой журнал обычно представляет собой таблицу базы данных, в которую программы PL/SQL после каждой операции с данными пишут сведения о том, кто ее выполнил. Проблема заключается в том, что и изменения самих данных и добавление строк в таблицу журнала обычно выполняются в рамках одной транзакции. При отмене этой транзакции вместе с отменой изменений данных будет отменено и добавление строк в таблицу журнала изменений, что может противоречить бизнес-правилам ведения журнала. Эти правила часто предусматривают, что в журнале должны фиксироваться все действия, включая отмененные.

Обычно добавление строк в таблицу журнал производят путем вызова процедуры, которой передают описание совершаемых действий. Если блок этой процедуры объявить как автономную транзакцию, то сведения в журнал изменений данных попадут независимо от фиксации или отмены основной транзакции.

```
CREATE TABLE accounts (id INTEGER PRIMARY KEY, status varchar2(10));
INSERT INTO accounts VALUES(133,'active');

CREATE TABLE change_log
(change_date DATE,          message     VARCHAR2(4000),
 username    VARCHAR2(100), ip_address  VARCHAR2(100));

SQL> CREATE OR REPLACE PROCEDURE log(p_message IN VARCHAR2) IS
  2    PRAGMA AUTONOMOUS_TRANSACTION;
  3  BEGIN
  4   INSERT INTO change_log
  5   VALUES(sysdate,p_message,user,sys_context('USERENV','IP_ADDRESS'));
  6   COMMIT;
  7  END;
  8  /
Procedure created.

SQL> SELECT * FROM change_log;
no rows selected

SQL> DECLARE
  2    l_account_id INTEGER := 133;
  3  BEGIN
  4    log('UPDATE accounts,id='||l_account_id||',set status=closed');
  5    UPDATE accounts SET status='closed' WHERE id=l_account_id;
  6    ROLLBACK;
  7  END;
  8  /
PL/SQL procedure successfully completed.
SQL> SELECT * FROM change_log;
CHANGE_DATE MESSAGE                                  USERNAME IP_ADDRESS
----------- ---------------------------------------- -------- -----------
16.01.2015  UPDATE accounts,id=133,set status=closed U1       192.168.0.8
```

Так как процедура `log` с помощью директивы компилятору `PRAGMA AUTONOMOUS_TRANSACTION` объявлена автономной транзакцией, то, несмотря на вызов команды `ROLLBACK` для отмены основной транзакции, в журнале запись о попытке закрытия лицевого счета не пропала.

Следует отметить, что автономные транзакции изначально использовались только внутренними механизмами сервера и были недоступны программистам. Механизмами сервера автономные транзакции

используются до сих пор — в основном для регистрации сообщений об ошибках, сохранения статистических данных о нагрузке и т. п. Настоятельно рекомендуется в своих проектах использовать их только для решения подобных же служебных задач с низкой степенью критичности возможных допущенных при программировании ошибок. Для бизнес-логики следует продумывать схему ее реализации обычными (не автономными) транзакциями.

Изменение данных в функциях PL/SQL, вызываемых в SQL

В коде функций PL/SQL, вызываемых в предложениях SQL, нельзя использовать DML-команды INSERT, UPDATE, DELETE. Рассмотрим следующий пример:

```
CREATE TABLE tab5 (at5 INTEGER)
CREATE TABLE tab6 (at6 INTEGER)
INSERT INTO tab5 VALUES(5)

SQL> CREATE OR REPLACE FUNCTION function1 RETURN INTEGER AS
  2  BEGIN
  3    INSERT INTO tab6 VALUES(123);
  4  END;
  5  /
Function created.

SQL> SELECT at5,function1 FROM tab5;
SELECT at5,function1 FROM tab5
           *
ERROR at line 1:
ORA-14551: cannot perform a DML operation inside a query
ORA-06512: at "U1.FUNCTION1", line 3
```

Если функцию объявить автономной транзакцией, то это ограничение снимается.

```
SQL> CREATE OR REPLACE FUNCTION function1 RETURN INTEGER AS
  2    PRAGMA AUTONOMOUS_TRANSACTION;
  3  BEGIN
  4    INSERT INTO tab6 VALUES(123);
  5    COMMIT;
  6    RETURN 7;
  7  END;
  8  /
Function created.

SQL> SELECT at5,function1 FROM tab5;
       AT5  FUNCTION1
---------- ----------
         5          7
```

```
SQL> SELECT * FROM tab6;
       AT1
----------
       123
```

В частности, с помощью автономных транзакций можно реализовать аудит обращений к некоторой таблице с особо конфиденциальной информацией (предполагаем, что у таблицы есть целочисленный первичный ключ):

- создаем функцию с целочисленным параметром, оформляя ее как автономную транзакцию, в теле функции помещаем команду вставки в таблицу журнала строки со значением переданного параметра и стандартными данными аудита — именем пользователя, IP-адресом, временем обращения; отметим, что не играет роли, что конкретно функция возвращает в качестве результата;
- создаем представление (VIEW), в котором в списке столбцов указываем нашу функцию, передавая ей в качестве параметра название столбца-первичного ключа таблицы;
- у всех пользователей отзываем привилегии выполнения предложений SELECT к этой таблице и предоставляем вместо них привилегии на SELECT к нашему представлению.

Теперь доступ может осуществляться не напрямую к таблице, а только через представление. Представления в Oracle устроены так, что для каждой строки, считанной из представления, будет вызываться наша функция и в журнале будет сохраняться значение столбца-первичного ключа строки таблицы с конфиденциальной информацией и сведения о том, кто, откуда и когда к ней обращался[1].

Курсоры FOR UPDATE

Обычная схема обработки данных с помощью явного курсора выглядит следующим образом: с помощью команды FETCH в цикле считываются отобранные SQL-запросом курсора строки и внутри цикла на

[1] Для аудита запросов в Oracle предназначен FGA (Fine Grained Audit). Однако FGA регистрирует в журнале аудита предложения SQL и их параметры, а не сами считываемые данные.

каждой строке выполняются некоторые вычисления, результаты которых затем сохраняются в базе данных. При этом считываемые из курсора строки не блокируются и могут быть изменены в это время другими транзакциями, что может привести к различным проблемам.

Рассмотрим следующий пример.

Пусть в базе данных есть таблица балансов клиентов:

```
CREATE TABLE balances (client_id INTEGER, balance   NUMBER);
```

Пусть в новогоднюю ночь в качестве подарка от компании для активных клиентов следует увеличить их баланс на пять процентов, а для всех остальных клиентов на один процент. Активность клиента определяется количеством услуг, которые ему были оказаны в уходящем году, поэтому определение типа клиента «активный / не активный» занимает в среднем одну минуту для одного клиента (ведь надо рассмотреть услуги, оказанные за целый год). Реализовать такую логику можно, например, следующим анонимным блоком.

```
DECLARE
  CURSOR c_balances IS SELECT * FROM balances;
  l_client_type    VARCHAR2(100);
  l_new_year_coeff NUMBER;
BEGIN

  FOR rec_balance IN c_balances LOOP

    -- определяем тип клиента (активный / не активный)
    l_client_type := getClientType(rec_balance); -- вызов длится 1 минуту

    CASE l_client_type
      WHEN 'active' THEN l_new_year_coeff := 1.05;
      WHEN 'non-actve' THEN l_new_year_coeff := 1.01;
    END CASE;

    UPDATE balances SET balance = rec_balance.balance*l_new_year_coeff
    WHERE balances.client_id = rec_balance.client_id;

  END LOOP;

  COMMIT;

END;
```

Если в базе данных 100 клиентов, то анонимный блок будет выполняться 100 минут. Пусть на 30-й минуте с момента начала выполнения блока клиент одной из еще не считанной из курсора строк (для определенности, пусть 70-й по счету) вносит платеж и его баланс увеличивается на сумму платежа. Транзакция увеличения баланса этого клиента фиксируется (изменить 70-ю строку этой транзакции можно, строка не блокирована, так как ней пока еще не обращались в ходе «подарочного» расчета).

Когда на 70-й минуте курсор c_balances дочитает до этой строки, то обнаружится, что строка с момента начала выполнения SQL-запроса курсора была изменена зафиксированной транзакцией и для обеспечения согласованности чтения восстановит ее предыдущую версию (с балансом без нового платежа). Соответственно, после выполнения команды UPDATE для этого клиента будет проставлена увеличенная на один или пять процентов сумма старого (без учета нового платежа) баланса.

В теории баз данных это называется явлением пропавшего обновления (lost update phenomena) — пропало отражение на балансе нового платежа, оно было перезаписано ранее считанными данными. В данном случае более правильная формулировка — перезаписано данными, восстановленными по состоянию, которое было ранее. Ведь собственно считывалась-то 70-я строка из курсора уже после поступления нового платежа, просто в ходе ее считывания для обеспечения согласованности произошло восстановление старой версии строки.

Чтобы не сталкиваться с проблемой пропавшего обновления, необходимо на все время «подарочной» транзакции заблокировать данные балансов от изменений. Для этого в курсоре c_balances следует записать не просто SQL-запрос, а SQL-запрос с блокировкой отбираемых строк:

```
CURSOR c_balances IS SELECT * FROM balances FOR UPDATE;
```

Посмотрим, что изменится в этом случае. Для запросов с опцией FOR UPDATE блокировка на отбираемые строки накладывается в ходе открытия курсора командой OPEN, еще до считывания первой строки командой FETCH. Поэтому если курсор с FOR UPDATE успешно открылся, то это значит, что все отбираемые SQL-запросом курсора строки уже заблокированы. Если какие-то строки с балансами заблокированы другими активными транзакциями, то наш процесс во время открытия курсора с SELECT FOR UPDATE будет ждать снятия этих блокировок.

После успешного открытия курсора c_balances с FOR UPDATE строки всех балансов будут заблокированы нашей транзакцией и до снятия этой блокировки только она может их изменять. Все другие транзакции, которые будут вносить платежи и изменять (увеличивать) балансы, сами будут переходить в режим ожидания снятия блокировки с балансов, установленной нашей «подарочной» транзакцией.

Как только на всех балансах будут отражены подарочные суммы и цикл считывания и обработки строк курсора c_balances завершится, в конце анонимного блока есть команда COMMIT, фиксирующая транзакцию. После этого все транзакции новых платежей, поступавших во время выполнения «подарочной» транзакции и ожидавшие ее завершения, выйдут из режима ожидания и увеличат уже увеличенные «по-новогоднему» балансы на суммы своих платежей. Проблема пропавшего обновления не возникнет[1].

Конструкция WHERE CURRENT OF в DML-командах

Конструкция WHERE CURRENT OF предназначена для удаления или изменения той строки таблицы, которая является текущей в курсоре. Преимущество использования этой конструкции заключается в однократном задании критерия отбора данных в SQL.

Перепишем рассмотренный выше пример с расчетом новогоднего подарочного увеличения баланса с использованием конструкции WHERE CURRENT OF и немного изменим его, сделав расчет не для всех клиентов, а для баланса одного конкретного клиента по его идентификатору.

Первый вариант программы (без использования WHERE CURRENT OF):

```
DECLARE
  l_client_id      INTEGER := 122329;
  l_balance        balances%ROWTYPE;
  l_new_year_coeff NUMBER;
```

[1] Пример носит учебный характер для иллюстрации решения проблемы пропавшего обновления. Конечно, более правильно для каждого клиента определить статус предварительно и сохранить эти сведения во временной таблице. Операция «подарочного» увеличения баланса в этом случае делается одним быстро выполняющимся SQL-предложением UPDATE вообще без PL/SQL.

```
BEGIN
  SELECT * INTO l_balance FROM balances
  WHERE balances.client_id=l_client_id FOR UPDATE;
  CASE getClientType(l_balance);
    WHEN 'active' THEN l_new_year_coeff := 1.05;
    WHEN 'non-actve' THEN l_new_year_coeff := 1.01;
  END CASE;
  UPDATE balances SET balance = l_balance.balance*l_new_year_coeff
  WHERE balances.client_id=l_client_id;
END;
```

Второй вариант программы (с использованием `WHERE CURRENT OF`**):**

```
DECLARE
  CURSOR c_balances IS SELECT * FROM balances
                       WHERE balances.client_id=122329 FOR UPDATE;
  l_client_type    VARCHAR2(100);
  l_new_year_coeff NUMBER;
BEGIN
  FOR rec_balance IN c_balances LOOP
    CASE getClientType(rec_balance)
      WHEN 'active' THEN l_new_year_coeff := 1.05;
      WHEN 'non-actve' THEN l_new_year_coeff := 1.01;
    END CASE;
      UPDATE balances SET balance = rec_balance.balance *l_new_year_coeff
      WHERE CURRENT OF c_balances;
    END LOOP;
END;
```

В первом варианте видно, что один и тот же критерий отбора данных (`WHERE balances.client_id=l_client_id`**) используется дважды — и в** `SELECT` **и в** `UPDATE`**. Ничего особо страшного в этом нет, но нарушается принцип DRY[1]: одна и та же логика программируется в двух местах, и при внесении каких-либо изменений придется следить за синхронизацией этих участков кода.**

При использовании конструкции `WHERE CURRENT OF` **во втором варианте программы PL/SQL компилятор неявно добавил столбец** `ROWID` **в список столбцов, возвращаемых запросом курсора. Напомним, что** `ROWID` **(row identifier) является физическим указателем на место размеще-**

[1] Don't repeat yourself, DRY («не повторяйся») — это принцип разработки программного обеспечения, нацеленный на снижение повторения в коде программ информации различного рода.

ния строки таблицы в файле данных и переход по ROWID — самый быстрый способ обращения к строке.

Для команды PL/SQL UPDATE с конструкцией WHERE CURRENT OF компилятором PL/SQL условие во фразе WHERE соответствующего SQL-предложения UPDATE будет сформировано как ROWID=:B1. На каждой итерации цикла перед выполнением UPDATE с переменной :B1 связывается значение ROWID строки таблицы balances, считанной из курсора на этой итерации.

Получается даже два положительных эффекта от использования конструкции WHERE CURRENT OF — и критерий отбора данных указывается один раз, и UPDATE строки таблицы по ее ROWID самый быстрый.

Оптимизация выполнения SQL из PL/SQL

Время работы программ PL/SQL, как правило, определяется суммарным временем выполнения предложений SQL и обработки их результатов. В языке PL/SQL имеются средства, позволяющие оптимизировать выполнение предложений SQL. Применение этих средств по некоторым оценкам позволяет повысить общую производительность программ PL/SQL на порядок[1].

Массовая обработка

Во время интерпретации байт-кода программ PL/SQL виртуальная машина PL/SQL имеющиеся в байт-коде предложения SQL передает ядру Oracle, которое выполняет их и возвращает результаты обработки обратно виртуальной машине PL/SQL. Передача управления между PL/SQL и SQL называется переключением контекста. Число переключений контекста определяется количеством выполненных команд INSERT, UPDATE, DELETE и количеством считанных строк результирующих выборок курсоров, причем на каждую считанную из курсора строку будет два переключения контекста — из PL/SQL в SQL и обратно.

---

[1] Речь идет не об оптимизации выполнения самих предложений SQL путем настройки их планов выполнения. Рассматривается оптимальная организация отправки из программ PL/SQL предложений SQL на выполнение и обработки их результатов.

Рассмотрим следующий пример. Пусть на обработку поступает «пачка» платежей. Требуется для каждого платежа увеличить баланс соответствующего лицевого счета на сумму платежа.

```
CREATE TABLE balances (account INTEGER, balance NUMBER);
INSERT INTO balances VALUES(101,500);
INSERT INTO balances VALUES(102,800);
INSERT INTO balances VALUES(103,532);
```

Первый вариант решения задачи — с последовательным выполнением команд UPDATE в цикле по всем платежам в «пачке»:

```
DECLARE
  TYPE t_payment IS RECORD
    (account INTEGER,
     amount  NUMBER,
     in_date DATE);
  TYPE t_payment_pack IS TABLE OF t_payment;
  l_payment_pack t_payment_pack := t_payment_pack();
BEGIN

  -- в пачке два платежа
  l_payment_pack.EXTEND(2);

  -- формируем первый платеж (50 рублей на лицевой счет 101)
  l_payment_pack(1).account := 101;
  l_payment_pack(1).amount  := 50;
  l_payment_pack(1).in_date := TO_DATE('02.03.2015','dd.mm.yyyy');

  -- формируем второй платеж (400 рублей на лицевой счет 102)
  l_payment_pack(2).account := 102;
  l_payment_pack(2).amount  := 400;
  l_payment_pack(2).in_date := TO_DATE('23.05.2015','dd.mm.yyyy');

  -- в цикле обновляем балансы
  FOR i IN 1..l_payment_pack.count LOOP
    UPDATE balances SET balance=balance+l_payment_pack(i).amount
    WHERE balances.account=l_payment_pack(i).account;
  END LOOP;

END;
```

В цикле будет выполнено две DML-команды UPDATE и произойдет четыре переключения контекста SQL-PL/SQL. Если бы в пачке платежей было 10 000 платежей, то переключений контекста было бы 20 000.

Каждое переключение контекста приводит к дополнительным затратам ресурсов, поэтому их число следует минимизировать. Идеаль-

ным решением является внесение всех изменений данных одним единственным предложением SQL. Во многих случаях этого можно добиться, однако все же бывает так, что или без выполнения команд INSERT, UPDATE, DELETE в цикле никак не обойтись, или предстоит считывание большого числа строк из курсора выполнением команды FETCH для каждой строки. Для таких случаев в языке PL/SQL есть средства массовой обработки данных (bulk processing), использование которых минимизирует число переключений контекста и повышает общую производительность программ PL/SQL:

- команда FORALL для выполнения наборов команд INSERT, UPDATE, DELETE;
- конструкция BULK COLLECT для считывания из курсора всех строк результирующей выборки одной командой.

Команда FORALL

Команда FORALL позволяет вместо циклического выполнения предложений SQL для команд INSERT, UPDATE, DELETE с постоянным переключением контекста PL/SQL-SQL собрать одинаковые предложения SQL в один набор и выполнить их все вместе в ходе одного обращения к ядру Oracle.

Команда FORALL имеет следующий синтаксис:

```
FORALL индекс IN [ нижняя граница ... верхняя граница |
INDICES OF коллекция | VALUES OF коллекция][ SAVE EXCEPTIONS ]
DML-команда (INSERT | UPDATE | DELETE)
```

Необязательная конструкция SAVE EXCEPTIONS указывает на необходимость обработки всех предложений SQL из набора с сохранением всех возникающих исключений. Так как для одной команды FORALL выполняется несколько предложений SQL, то возникает вопрос о том, что будет, если при выполнении одного из них произойдет ошибка. Общие правила здесь следующие:

- изменения, сделанные предложением SQL, завершившимся с ошибкой, отменяются;
- изменения, сделанные предшествующими успешно выполненными предложениями SQL из набора этой команды FORALL, не отменяются;
- если отсутствует конструкция SAVE EXCEPTIONS, то выполнение FORALL останавливается.

Приведем второй вариант решения задачи обновления балансов для нескольких поступивших платежей.

```
DECLARE
  TYPE t_payment IS RECORD
    (account INTEGER,
     amount  NUMBER,
     in_date DATE);
  TYPE t_payment_pack IS TABLE OF t_payment;
  l_payment_pack t_payment_pack := t_payment_pack();
BEGIN

  l_payment_pack.EXTEND(2);

  l_payment_pack(1).account := 101;
  l_payment_pack(1).amount  := 50;
  l_payment_pack(1).in_date := TO_DATE('02.03.2015','dd.mm.yyyy');

  l_payment_pack(2).account := 102;
  l_payment_pack(2).amount  := 400;
  l_payment_pack(2).in_date := TO_DATE('23.05.2015','dd.mm.yyyy');

  FORALL indx IN 1..l_payment_pack.COUNT
    UPDATE balances SET balance=balance+l_payment_pack(indx).amount
    WHERE balances.account=l_payment_pack(indx).account;
  END;
```

Два предложения UPDATE выполнились в составе одного набора. Вместо четырех переключений контекста PL/SQL-SQL их произошло два. Если бы в пачке платежей было 10 000 платежей, то число переключений контекста по-прежнему осталось бы равным двум, а не 20 000.

Конструкция BULK COLLECT

Использование конструкции BULK COLLECT позволяет считать из курсора сразу все строки результирующей выборки SQL-запроса. Курсор при этом может быть как явным, так и неявным — для команды SELECT INTO. «Приемником» для строк, считанных с использованием конструкции BULK COLLECT, должна быть коллекция. При массовом считывании также не происходит переключений контекстов и выборка данных осуществляется оптимальным образом.

Перепишем приведенные ранее блоки PL/SQL для считывания всех строк из явного курсора. Для наглядности приведем обе реализации (с циклом и без него).

Считывание в цикле по одной строке	Использование BULK COLLECT

DECLARE
  CURSOR c1 IS SELECT * FROM tab1;
  rec c1%ROWTYPE;
BEGIN
  OPEN c1;
  FETCH c1 INTO rec;
  WHILE c1%FOUND LOOP
  FETCH c1 INTO rec;
  END LOOP;
  CLOSE c1;
END;
	DECLARE
	 CURSOR c1 IS SELECT * FROM tab1;
	 TYPE t_tab IS TABLE OF c1%ROWTYPE;
	 l_tab t_tab;
	BEGIN
	  OPEN c1;
	  FETCH c1 BULK COLLECT INTO l_tab;
	  CLOSE c1;
	END;

Обратите внимание, в коде объявлена коллекция на основе курсора, в эту коллекцию и осуществляется считывание. В результате получается очень компактный код, в котором, например, в коде считывания строк результирующей выборки нигде не указаны столбцы выборки, а сам код считывания занимает три строчки, при этом не используются команды циклов.

Хранимые программы

Виды хранимых программ

В PL/SQL имеются следующие виды хранимых программ:

- процедура (procedure) — программа, которая выполняет одно или несколько действий и вызывается как исполняемая команда PL/SQL;
- функция (function) — программа, которая возвращает одно значение и используется как выражение PL/SQL;
- пакет (package) — набор процедур, функций, переменных, констант и типов данных, объединенных общим функциональным назначением;
- триггер (trigger) — программа, которая автоматически запускается при наступлении событий, указанных при создании триггера.

Создание, изменение и удаление хранимых программ

Хранимые программы являются объектами баз данных Oracle. Как и другие объекты баз данных, хранимые программы создаются DDL-

командами CREATE, изменяются DDL-командами ALTER и удаляются DDL-командами DROP.

Чтобы создать хранимую процедуру в своей схеме, пользователю необходимо иметь системную привилегию CREATE PROCEDURE или роль с этой привилегией, например, роль RESOURCE. Привилегии CREATE FUNCTION в Oracle SQL нет, привилегия CREATE PROCEDURE позволяет создавать и процедуры, и функции, и пакеты.

Для создания этих хранимых программ в схемах других пользователей требуется наличие системной привилегии CREATE ANY PROCEDURE, предоставленной явно или через роль. Для создания триггеров требуются отдельные привилегии CREATE TRIGGER и CREATE ANY TRIGGER.

DDL-команды CREATE для создания хранимых программ PL/SQL имеют необязательные ключевые слова CREATE [OR REPLACE], указывающую на замену существующей программы новой программой с тем же именем. Если слова OR REPLACE не указаны в команде CREATE, а хранимая программа с таким именем в базе данных уже есть, то создание программы завершится с ошибкой.

```
SQL> CREATE PROCEDURE proc1 AS
  2  BEGIN
  3    NULL;
  4  END;
  5  /
Procedure created.

SQL> CREATE PROCEDURE proc1 AS
  2  BEGIN
  3    NULL;
  4  END;
  5  /
CREATE PROCEDURE proc1 AS
                 *
ERROR at line 1:
ORA-00955: name is already used by an existing object

SQL> CREATE OR REPLACE PROCEDURE proc1 AS
  2  BEGIN
  3    NULL;
  4  END;
  5  /
Procedure created.
```

Можно было бы сначала удалить существующую программу, а потом создать новую с тем же именем, но рекомендуется так не делать по следующей причине.

Для хранимых программ PL/SQL пользователям и ролям базы данных предоставляются объектные привилегии на их выполнение. Если удалить хранимую программу, то эти привилегии пропадут (правильнее сказать — автоматически отзовутся в связи с удалением объекта доступа). После того, как хранимая программа с таким же именем заново будет создана, привилегии эти сами по себе не восстановятся, владельцу программы придется предоставлять их другим пользователям снова. При пересоздании хранимой программы DDL-командой CREATE OR REPLACE с привилегиями на ее выполнение ничего не происходит.

Находящиеся в базе данных хранимые программы можно перекомпилировать с помощью DDL-команды ALTER:

```
SQL> ALTER PROCEDURE proc1 COMPILE;
Procedure altered.
```

Как и другие объекты базы данных, хранимые программы могут быть удалены. Пользователю не требуются дополнительные привилегии для удаления программ в своей схеме, для удаления программ в схеме другого пользователя необходимо наличие привилегии DROP ANY PROCEDURE.

```
SQL> DROP PROCEDURE proc1;
Procedure dropped.
```

Процедуры и функции

Функция отличается от процедуры тем, что функция возвращает значение указанного при создании функции типа данных, а процедура ничего не возвращает. Вызов функции всегда включается в некоторое выражение, то есть возвращаемый функцией результат обязательно нужно куда-то деть — присвоить его значение некоторой переменной или передать в качестве параметра другой функции или процедуре. Функции на PL/SQL можно использовать в предложениях SQL наряду со встроенными функциями языка SQL.

Обычно процедуры и функции создаются для решения определенных небольших задач. При продуманной структуре исходного кода каждая процедура или функция со всеми разделами и вложенными бло-

ками должна умещаться на одном экране (максимум 30-40 строк). Если код процедуры или функции разрастается, то имеет смысл продумать его декомпозицию, использовать пакеты или перегружаемые программы.

Процедуры

Команда создания процедуры имеет следующий синтаксис:

```
CREATE [OR REPLACE]
-- раздел заголовка блока PL/SQL
PROCEDURE
[имя схемы.]имя процедуры
[(имя параметра [{IN | OUT | IN OUT}] тип данных
[,имя параметра [{IN | OUT | IN OUT}] тип данных ...])]
{IS | AS}
остальные разделы блока PL/SQL (объявлений,исполняемый,обработки исключений)
```

В процедурах не используется ключевое слово DECLARE — объявление пользовательских типов данных, переменных, курсоров начинается сразу после ключевого слова AS. Областью видимости объявленных здесь элементов будет являться вся процедура. В разделе объявлений процедуры можно реализовать и другую процедуру или функцию, которые будут видны только внутри родительской процедуры:

```
CREATE OR REPLACE PROCEDURE proc2 AS
  FUNCTION nested_proc RETURN INTEGER IS
  BEGIN
    NULL;
  END;
BEGIN
  nested_proc();
END;
```

Пусть таблица tab1 создана следующей DDL-командой:

```
CREATE TABLE tab1 (at1 NUMBER, at2 DATE);
```

Создадим процедуру insRec, которая заносит в таблицу 1/2 переданного значения числового параметра и текущую дату.

```
SQL> CREATE OR REPLACE PROCEDURE insRec(p_arg1 IN NUMBER) AS
  2    coeff CONSTANT NUMBER := 0.5;
  3  BEGIN
  4    INSERT INTO tab1 VALUES(coeff*p_arg1,SYSDATE);
  5  END;
/
Procedure created.
```

После создания процедуру можно вызвать из любого блока PL/SQL, указав ее имя и параметры.

```
SQL> DECLARE
  2    l_arg1 NUMBER := 240;
  3  BEGIN
  4    insRec(l_arg1);
  5  END;
  6  /
PL/SQL procedure successfully completed.

SQL> SELECT * FROM Tab1;
AT1          AT2
-----------  ---------------
120          04.05.2015
```

В SQL*Plus для вызова процедур есть команда EXECUTE.

```
SQL> EXECUTE insRec(100);
PL/SQL procedure successfully completed.

SQL> SELECT * FROM Tab1;
AT1          AT2
-----------  ---------------
120          04.05.2015
 50          04.05.2015
```

В процедурах можно использовать команду RETURN. Как только в потоке команд в процедуре встретится команда RETURN, выполнение процедуры прекращается и управление передается вызвавшему процедуру блоку.

Функции

Команда создания функции имеет следующий синтаксис:

```
CREATE [OR REPLACE] FUNCTION
-- раздел заголовка блока PL/SQL
[имя схемы.]имя функции
[(имя параметра [{IN | OUT | INOUT}] тип данных
[,имя параметра [{IN | OUT | INOUT}] тип данных ...])] RETURN тип данных AS
остальные разделы блока PL/SQL (объявлений,исполняемый,обработки исключений)
```

Пусть таблица tab1 создана и заполнена следующим образом:

```
CREATE TABLE tab1 (at1 NUMBER, at2 DATE);
INSERT INTO tab1 VALUES(5, SYSDATE);
INSERT INTO tab1 VALUES(6, SYSDATE);
INSERT INTO tab1 VALUES(7, SYSDATE+1);
```

Создадим функцию, которая вычисляет сумму значений столбцов таблицы, таких, что дата попадает в заданный интервал.

```
SQL> CREATE OR REPLACE FUNCTION sumRecInt(arg1 IN DATE,
  2                                       arg2 IN DATE) RETURN NUMBER AS
  3    sum_var NUMBER := 0;
  4  BEGIN
  5    SELECT SUM(at1) INTO sum_var FROM tab1
  6      WHERE at2 BETWEEN arg1 AND arg2;
  7    RETURN sum_var;
  8  END;
  9  /
Function created.

SQL> BEGIN
  2    DBMS_OUTPUT.PUT_LINE(sumRecInt(SYSDATE-1/2, SYSDATE+1/2));
  3  END;
  4  /
11
PL/SQL procedure successfully completed.
```

Ход вычислений функции обязательно должен завершаться вызовом в ее теле команды `RETURN возвращаемое значение`**. Если этого не произойдет, то возникнет ошибка этапа выполнения:**

```
SQL> CREATE FUNCTION func2 RETURN INTEGER AS
  2  BEGIN
  3    NULL;
  4  END;
  5  /

Function created.

SQL> BEGIN
  2    DBMS_OUTPUT.PUT_LINE(func2);
  3  END;
  4  /
BEGIN
*
ERROR at line 1:
ORA-06503: PL/SQL: Function returned without value
ORA-06512: at "U1.FUNC2", line 3
ORA-06512: at line 2
```

Иногда на лекциях студентами задается вопрос, поддерживаются ли в PL/SQL рекурсивные функции, то есть функции, вызывающие сами себя. Поддерживаются, приведем пример наиболее понятной на все времена рекурсивной функции:

```
SQL> CREATE OR REPLACE FUNCTION factorial(n IN INTEGER) RETURN INTEGER IS
  2  BEGIN
  3    IF n=0 THEN
  4      RETURN 1;
  5    ELSE
  6      RETURN n*factorial(n-1);
  7    END IF;
  8  END;
  9  /
Function created.

SQL> DECLARE
  2    l_number INTEGER := 3;
  3  BEGIN
  4    DBMS_OUTPUT.PUT_LINE(factorial(l_number));
  5    DBMS_OUTPUT.PUT_LINE(factorial(COS(0)));
  6  END;
  7  /
6
1
PL/SQL procedure successfully completed.
```

Параметры процедур и функций

Процедуры и функции могут иметь параметры, для которых указываются имена, типы данных и режимы передачи значений.

Важно понимать различия между формальными и фактическими параметрами. Формальные параметры указываются в списке параметров заголовка программы при ее объявлении, тогда как фактические параметры — это значения и выражения, которые помещаются в список параметров при ее вызове. Иными словами, значения фактических параметров передаются при вызове внутрь процедур и функций, где становятся значениями формальных параметров. Фактическим параметром при первом вызове функции `factorial` являлась переменная `l_number`, объявленная в вызывающем блоке. Эта переменная имела значение 3, которое и было использовано внутри функции (3!=6). При втором вызове функции `factorial` фактическим параметром являлось выражение `COS(0)`. Как известно, 1!=1;

Соответствие формальных и фактических параметров

Соответствие между формальными и фактическими параметрами можно устанавливать двумя способами:

- связывание по позиции (неявное связывание);
- связывание по имени.

При неявном связывании фактические параметры указываются в круглых скобках после имени программы в той же последовательности, в которой были перечислены формальные параметры при создании программы.

Связывание формальных и фактических параметров по имени осуществляется с помощью конструкций вида

```
имя формального параметра => имя фактического параметра
```

С точки зрения выполнения программы нет разницы между используемыми ней способами установления соответствия между параметрами, которые обычно определяются принятым стилем программирования, корпоративными стандартами кодирования и рядом других факторов. В программах с небольшим числом формальных параметров оправдано использование соответствия параметров по позиции. В программах с большим числом формальных параметров связывание параметров по имени более информативно, более ясно показывает связь между формальными и фактическими параметрами.

```
SQL> CREATE PROCEDURE print(phrase IN VARCHAR2,punctuation_mark IN CHAR) IS
  2  BEGIN
  3    DBMS_OUTPUT.PUT_LINE(phrase||' '||punctuation_mark);
  4  END;
  5  /

Procedure created.

SQL> BEGIN
  2    print('Hello,world','!');
  3  END;
  4  /
Hello,world !

PL/SQL procedure successfully completed.

SQL> BEGIN
  2    print(punctuation_mark=>'!',phrase=>'Hello,world');
  3  END;
  4  /
Hello,world !

PL/SQL procedure successfully completed.
```

Если у программы в будущем появятся новые формальные параметры, то код, в котором она вызывается со связыванием параметров по имени, останется работоспособным. Если для новых параметров ука-

заны значения по умолчанию, то они будут использованы как фактические параметры, если значения по умолчанию отсутствуют, то новые параметры получат значения NULL. А вот все вызовы со связыванием по позиции при появлении у вызываемой программы новых формальных параметров потребуется изменить так, чтобы фактических параметров снова стало столько же, сколько формальных.

Отметим, что при хорошем стиле программирования не принято объявлять процедуры и функции с большим (больше 10) числом параметров скалярных типов данных. В этом случае надо использовать небольшое число параметров составных типов данных (записи PL/SQL или коллекции).

Режимы передачи значений параметров

В PL/SQL есть три режима передачи значений параметров.

Таблица 5. Режимы передачи значений параметров в PL/SQL.

Режим	Предназначение	Использование
IN	только для чтения	переданное значение параметра может читаться, но не может быть изменено внутри процедуры или функции
OUT	для записи	для записи как в неинициализированную переменную (значению параметра внутри процедуры или функции сразу присваивается значение NULL, в дальнейшем оно может изменяться)
IN OUT	для чтения и записи	передается значение, которое можно читать и изменять внутри процедуры или функции

В большинстве случаев параметры передаются в процедуры и функции в режиме IN (именно этот режим используется по умолчанию). Режимы передачи параметров OUT и IN OUT в свою очередь позволяют, например, реализовать возвращение нескольких значений для функции.

Часто функции возвращают код завершения своей работы, который указывается как параметр команды RETURN в теле функции (например, ноль — успешное завершение, ненулевое значение — номер ошибки). По смыслу функции получить от нее помимо результата еще что-то, например, диагностическое сообщение с подробностями

к коду завершения, невозможно. В команде RETURN может быть указан только один параметр. Выходом является использование формального параметра с режимом передачи OUT. В теле функции следует предусмотреть формирование и запись в этот параметр текстов сообщений, и после каждого вызова функции эти сообщения будут доступны в вызывающем коде в переменных-фактических параметрах.

Основное отличие режима передачи OUT от режима IN OUT заключается в том, что OUT-параметр становится неинициализированным при передаче внутрь процедуры или функции, то есть то значение, которое имела во внешнем блоке переменная-фактический параметр, теряется (становится равным NULL). Это верно во всех случаях, за исключением ситуации, когда внутри процедуры или функции инициируется необработанное в ней исключение. Тогда во внешнем блоке у переменной-фактического параметра для формального OUT-параметра сохранится то ее значение, которое было до передачи. У фактических параметров для формальных IN OUT-параметров значение в NULL не сбрасывается. Если значение фактического параметра внутри программы не меняли, то и после завершения вызова программы оно будет таким же, каким оно было до передачи в программу. Приведем примеры передачи значений параметров в различных режимах.

```
SQL> CREATE OR REPLACE PROCEDURE test(p1 IN INTEGER,
  2                                   p2 OUT INTEGER,
  3                                   p3 IN OUT INTEGER) IS
  4  BEGIN
  5    p2 := 11;
  6    p3 := 12;
  7  END;
  8  /

Procedure created.

SQL> DECLARE
  2    l_arg1 INTEGER := 5;
  3    l_arg2 INTEGER := 6;
  4    l_arg3 INTEGER := 7;
  5  BEGIN
  6    DBMS_OUTPUT.PUT_LINE('before l_arg2='||l_arg2);
  7    DBMS_OUTPUT.PUT_LINE('before l_arg3='||l_arg3);
  8    test(p1 => l_arg1,p2 => l_arg2,p3 => l_arg3);
  9    DBMS_OUTPUT.PUT_LINE('after  l_arg2='||l_arg2);
 10    DBMS_OUTPUT.PUT_LINE('after  l_arg3='||l_arg3);
 11  END;
 12  /
```

```
before l_arg2=6
before l_arg3=6
after  l_arg2=11
after  l_arg3=12
PL/SQL procedure successfully completed.
```

Видно, что значение переменной `l_arg2`**, которое было до вызова процедуры** `test` **равным 6, внутри процедуры было изменено на 11. Значение переменной** `l_arg3` **после вызова процедуры стало равным 12.**

Изменим код процедуры `test`**, заменив ее исполняемый блок пустой командой** `NULL` **(то есть с параметрами в коде процедуры никаких действий осуществляться не будет) и вызовем ее еще раз с такими же значениями фактических параметров:**

```
SQL> CREATE OR REPLACE PROCEDURE test(p1 IN INTEGER,
  2                                   p2 OUT INTEGER,
  3                                   p3 IN OUT INTEGER) IS
  4  BEGIN
  5    NULL;
  6  END;
  7  /
Procedure created.

SQL> DECLARE
  2    l_arg1 INTEGER := 5;
  3    l_arg2 INTEGER := 6;
  4    l_arg3 INTEGER := 7;
  5  BEGIN
  6    DBMS_OUTPUT.PUT_LINE('before l_arg2='||l_arg2);
  7    DBMS_OUTPUT.PUT_LINE('before l_arg3='||l_arg3);
  8    test(p1 => l_arg1,p2 => l_arg2,p3 => l_arg3);
  9    DBMS_OUTPUT.PUT_LINE('after l_arg2='||l_arg2);
 10    DBMS_OUTPUT.PUT_LINE('after l_arg3='||l_arg3);
 11  END;
 12  /
before l_arg2=6
before l_arg3=7
after l_arg2=
after l_arg3=7
PL/SQL procedure successfully completed.
```

Как и ожидалось, значение переменной `l_arg2`**, переданной в процедуру** `test` **как** `OUT`**-параметр, стало** `NULL`**. Значение переменной** `l_arg3` **не изменилось.**

Способы передачи значений параметров

Виртуальная машина PL/SQL во время выполнения программ PL/SQL применяет два способа передачи значений параметров:

- по ссылке — с соответствующим формальным параметром связывается указатель, а не фактическое значение (после этого и формальный и фактический параметры ссылаются на ячейку памяти, содержащую значение параметра);
- по значению — значение фактического параметра копируется в соответствующий формальный параметр (если впоследствии программа завершается без необработанных исключений, то значение формального параметра присваивается обратно фактическому).

Понятно, что для режима передачи значений параметров `IN` используется передача параметров по ссылке (ведь `IN`-параметры не изменяются внутри процедур и функций, поэтому значение достаточно только читать по ссылке). Для режимов `OUT` и `IN OUT` обычно используется передача по значению.

Ошибки компиляции программ PL/SQL

На практике в большинстве случаев первая попытка откомпилировать программу на языке PL/SQL приводит к получению сообщения о наличии ошибок в ее коде. Чтобы увидеть выявленные компилятором ошибки, можно воспользоваться командой утилиты SQL*Plus `SHOW ERRORS`. Если команда `SHOW ERRORS` используется без параметров, то возвращаются ошибки последней компилированной программы.

Создадим процедуру PL/SQL с синтаксической ошибкой (пропущен символ `;` после команды `NULL`):

```
SQL> CREATE PROCEDURE proc1 AS
  2  BEGIN
  3    NULL
  4  END;
  5  /
Warning: Procedure created with compilation errors.

SQL> SHOW ERRORS
Errors for PROCEDURE PROC1:

LINE/COL ERROR
-------- -----------------------------------------------------------------
4/1      PLS-00103: Encountered the symbol "END" when expecting one of the
         following: ; The symbol ";" was substituted for "END" to continue.
```

Попробуем создать процедуру с другой ошибкой:

```
SQL> CREATE OR REPLACE PROCEDURE test(p1 IN INTEGER,
  2                                   p2 OUT INTEGER,
  3                                   p3 IN OUT INTEGER) IS
  4    l_p INTEGER := 10;
  5  BEGIN
  6    p1 := l_p;
  7  END;
  8  /
Warning: Procedure created with compilation errors.
SQL> SHOW ERRORS
Errors for PROCEDURE TEST:

LINE/COL ERROR
-------- -----------------------------------------------------------------
6/3      PL/SQL: Statement ignored
6/3      PLS-00363: expression 'P1' cannot be used as an assignment target
```

В коде процедуры `test` имеется семантическая (смысловая) ошибка — попытка изменить значение параметра с режимом передачи `IN`. Компилятор PL/SQL при анализе кода проверяет отсутствие таких параметров в левой части команд присваивания, в конструкциях `SELECT INTO` и в других местах кода, где значения таких параметров может быть изменено.

В обоих случаях процедуры `proc1` и `test` как новые объекты базы данных создавались, но с ошибками (Procedure created with compilation errors). Такие объекты базы данных получают статус `INVALID` и непригодны для использования.

Попытка вызвать процедуру `test` приведет к ошибке:

```
SQL> DECLARE
  2    l_arg1 INTEGER :=5;
  3    l_arg2 INTEGER :=6;
  4    l_arg3 INTEGER :=7;
  5  BEGIN
  6    test(p1 => l_arg1,p2 => l_arg2,p3 => l_arg3);
  7  END;
  8  /
  test(p1 => l_arg1,p2 => l_arg2,p3 => l_arg3);
  *
ERROR at line 6:
ORA-06550: line 6, column 3:
PLS-00905: object USER1.TEST is invalid
ORA-06550: line 6, column 3:
PL/SQL: Statement ignored
```

Хранимая программа PL/SQL может получить статус INVALID как из-за наличия в ее коде синтаксических и семантических ошибок, так и по другим причинам, например, если какие-то объекты базы данных, к которым есть обращения в коде программы, стали недоступными (были удалены, были отозваны привилегии доступа к ним и т. п.).

Отладка программ на PL/SQL

Исправлять ошибки, выявленные компилятором PL/SQL в ходе анализа кода, обычно довольно просто. Для исправления выявленных пользователями ошибок этапа выполнения, следует использовать отладчик PL/SQL. Для удобства отладки можно порекомендовать использовать специализированные средства, например, интегрированную среду разработки Quest SQL Navigator, в которой есть и breakpoints, и watches, и step into, и step over — в общем, все средства, достаточные для эффективной отладки программ на процедурном языке программирования.

Для использования отладчика отлаживаемую программу PL/SQL необходимо перекомпилировать с опцией добавления отладочной информации.

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

Редактировать код хранимых программ по опыту автора также рекомендуется в специализированном Stored Program Editor, который есть в Quest SQL Navigator, TOAD, PL/SQL Developer и Oracle SQL Developer:

- после открытия в редакторе исходного текста хранимой программы с ошибками курсор в тексте сразу позиционируется на место ошибки с отображением сообщения об ошибке;
- в Stored Program Editor редактируется актуальная версия кода, которая находится в словаре-справочнике данных базы данных Oracle;
- есть стандартные для современных IDE подсветка синтаксиса и автодополнение кода, что очень удобно;
- нажатием клавиш Ctrl+S или соответствующей кнопки интерфейса можно быстро отправить код программы на компиляцию.

Пакеты

Объединенные общим функциональным назначением процедуры и функции принято оформлять в виде пакета PL/SQL. Можно считать, что пакет — это аналог библиотеки программ. Прием оформления родственных программ в библиотеки хорошо известен из практики разработки программного обеспечения. В информационной системе с развитой серверной бизнес-логикой могут быть тысячи процедур и функций на языке PL/SQL. Чтобы они не лежали в базе данных тысячами объектов, правильно объединить их по функциональному признаку в пакеты, дав им названия, соответствующие области применения. Например, в базе данных могут быть такие пакеты:

- `pk_clients` (пакет для работы с клиентскими данными);
- `pk_stocks` (пакет для работы со складами);
- `pk_orders` (пакет для обработки заказов);
- ...

Для каждого пакета следует назначить ответственного за него программиста, который будет сопровождать пакет и развивать его функциональность. Часть кода, реализующего общесистемную логику, например, унифицированную обработку ошибок и ведение журналов изменения данных, можно выделить в отдельную группу пакетов ядра прикладной системы (kernel packages), назначив ответственными за них самых опытных программистов.

Спецификация и тело пакета

Пакет PL/SQL состоит из двух объектов базы данных: спецификации пакета (`PACKAGE`) и тела пакета (`PACKAGE BODY`). Команда создания спецификации пакета имеет следующий синтаксис:

```
CREATE [OR REPLACE] PACKAGE [имя_схемы.]имя_пакета {IS | AS}
  спецификация пакета
END;
```

Команда создания тела пакета имеет следующий синтаксис:

```
CREATE [OR REPLACE] PACKAGE BODY [имя_схемы.]имя_пакета {IS | AS}
  [спецификация локальных элементов пакета]
  блоки PL/SQL реализации процедур и функций, объявленных в спецификации
  блоки PL/SQL локальных процедур и функций
[BEGIN секция инициализации пакета]
END;
```

В спецификации пакета находится описание следующих программных элементов, доступных из других программ PL/SQL (то есть элементов, видимых извне):

- пользовательские типы данных;
- пользовательские исключения;
- процедуры и функции;
- переменные;
- константы;
- курсоры.

Эти программные элементы называются глобальными пакетными переменными, глобальными пакетными курсорами и т. п.

Для процедур и функций в спецификации пакета присутствуют только заголовки — названия процедур и функций и описания их параметров. В спецификации пакета нет блоков PL/SQL, реализующих логику процедур и функций, вся она находится в теле пакета. Можно считать, что спецификация пакета является интерфейсной частью — аналогом заголовочных файлов (header files), имеющихся, например, в языке программирования C++.

В теле пакета могут быть объявлены все те же виды программных элементов, что и в спецификации пакета с той лишь разницей, что они не будут доступны из других программ PL/SQL (не видны извне тела пакета). Эти элементы называются локальными пакетными переменными, локальными пакетными процедурами и т. п.

Локальные программные элементы предназначены исключительно для использования только процедурами и функциями самого пакета. Тем самым в PL/SQL реализовано сокрытие, то есть принцип проектирования программного обеспечения, заключающийся в разграничении доступа различных программ к внутренним компонентам друг

друга[1]. Подчеркнем, разграничивается доступ именно к внутренним компонентам.

Покажем области видимости объявленных в пакетах переменных и программ:

```
CREATE OR REPLACE PACKAGE pkg1 AS

  -- g_var1 - глобальная пакетная переменная
  -- видна и в теле пакета и снаружи (причем может изменяться снаружи)
  g_var1 INTEGER;

  -- глобальная пакетная процедура, видна и в теле пакета и снаружи
  PROCEDURE proc1;

END;

CREATE OR REPLACE PACKAGE BODY pkg1 AS

  -- локальная переменная, видна внутри тела пакета, снаружи не видна
  l_var2 DATE;

  -- локальная функция, видна внутри тела пакета, снаружи не видна
  FUNCTION function1 RETURN VARCHAR2 IS
  BEGIN
    RETURN TO_CHAR(SYSDATE,'DD.MM.YYYY HH24:MI:SS');
  END;

  -- реализация в теле логики процедуры proc1, объявленной в спецификации
  PROCEDURE proc1 IS
    -- l_p_var3 - локальная переменная процедуры proc1
    -- видна только внутри процедуры proc1
    l_p_var3 VARCHAR2(2000);
  BEGIN
    l_p_var3 := function1||' '||to_char(l_var2)||to_char(g_var1);
  END;

END;
```

[1] Понятие сокрытия тесно связано с понятием инкапсуляции — упаковкой данных или программных модулей в единый компонент. Инкапсуляция используется для управления сокрытием.

Достоинства использования пакетов

Сформулируем достоинства использования пакетов PL/SQL при разработке серверной бизнес-логики:

- упрощение сопровождения и расширения программ PL/SQL, так как пакеты обеспечивают инкапсуляцию кода и позволяют группировать логически связанные процедуры и функции;
- разграничение доступа различных пакетов к внутренним компонентам друг друга;
- сохранение данных сессии пользователя в глобальных пакетных переменных, в том числе повышение производительности приложений за счет кэширования постоянно использующихся в программах PL/SQL данных, например, справочников (данные кэшируются в коллекциях-глобальных переменных пакетов);
- исключение жестко кодируемых литералов (hard-coded literals)[1].

Хороший стиль программирования на PL/SQL предусматривает даже для небольших проектов наличие спецификаций пакетов, в которых объявлены

- все пользовательские типы данных;
- все константы и переменные, которые инициализируются жестко кодируемыми литералами, в том числе магическими числами[2];
- все SQL-запросы в виде объявлений явных курсоров;
- все пользовательские исключения.

Настоятельно рекомендуется все объявления программных элементов такого рода всех программ PL/SQL, реализующих серверную бизнес-логику системы, собрать в одной или нескольких специфика-

---

[1] Литерал — это элемент программы, который непосредственно представляет значение (число, строка, дата, логическое значение).

[2] Магическое число — целочисленная константа. Такое число само по себе не несет никакого смысла и может вызвать недоумение, встретившись в коде программы без соответствующего контекста или комментария, при этом попытка изменить его на другое, может привести к абсолютно непредсказуемым последствиям. По этой причине подобные числа были иронично названы магическими.

циях пакетов, а не «размазывать» объявления типов, исключений, констант и т. п. по всему коду или переписывать одну и ту же команду SELECT INTO в нескольких местах. Иногда даже создают отдельные спецификации пакетов только для объявлений типов, переменных, исключений и курсоров без объявлений процедур и функций. Для таких спецификаций изначально не планируется создавать тела пакетов.

Ни в коем случае не следует расставлять по всему исходному коду PL/SQL жестко кодируемые литералы. Например, если в коде в сорока местах для вычисления сумм «чистыми» использовать выражения вида (...)*0.87, то когда ставка подоходного налога перестанет быть равной 13%, надо будет найти все сорок мест и заменить 0.87 на новое значение. А самое интересное начнется, если почти везде по коду поменять значение литерала на новое, а где-то забыть и оставить старое. Чтобы не заниматься всем этим, правильно один раз объявить в спецификации пакета константу

```
g_c_tax_percent INTEGER := 13;
```

и во всем коде в дальнейшем использовать только ее. Если потребуется внести изменения, то новое число нужно будет указать только в одном месте — в значении константы.

Отношения между спецификацией и телом пакета

Отношения между спецификацией и телом пакета описываются следующим образом:

- сначала создается спецификация пакета, затем его тело;
- тело пакета не может существовать без спецификации и даже не создастся DDL-командой CREATE PACKAGE BODY с выдачей сообщения об ошибке;
- спецификация пакета без тела существовать может, на объявленные в ней глобальные пакетные процедуры и функции можно ссылаться из других программ PL/SQL (ошибка обращения к такому бестелесному пакету возникнет только на этапе выполнения);
- при перекомпиляции спецификации пакета автоматически перекомпилируется его тело, при перекомпиляции тела пакета его спецификация не перекомпилируется;

- при удалении спецификации пакета автоматически удаляется его тело, при удалении тела пакета с его спецификацией ничего не происходит.

Эти отношения между спецификацией и телом пакета имеют следующие положительные аспекты, которые особенно полезны при большом объеме кода на PL/SQL в крупных проектах:

- возможность создания на этапе прототипирования «заглушек» — пакетов без тел, реализация объявленных в их спецификациях процедур и функций будет осуществлена позже;
- при перекомпиляции тела пакета с зависящими от его спецификации другими программами на PL/SQL ничего не происходит.

Перегружаемые программы

Часто использующейся возможностью пакетов является поддержка перегружаемых (overloaded) процедур и функций, для которых в пакете существует несколько версий программ с одинаковыми именами, но с различными по составу или типам данных параметрами:

```
-- статус клиента на текущую дату
FUNCTION getClientStatus(p_client_id IN INTEGER) RETURN VARCHAR2

-- статус клиента на дату-параметр
FUNCTION getClientStatus(p_client_id IN INTEGER,
                         p_status_date IN DATE) RETURN VARCHAR2
```

При вызове перегруженной процедуры или функции по числу и типу передаваемых фактических параметров автоматически определяется требуемая версия процедуры или функции, которая и вызывается. Для приведенных вариантов объявления функции при указании одного фактического параметра вызовется первая версия функции `getClientStatus`, с двумя фактическими параметрами — вторая. Довольно часто перегружаемые программы вызывают друг друга, например, первую версию функции `getClientStatus` было бы правильно реализовать с помощью ее же второй версии:

```
FUNCTION getClientStatus(p_client_id IN INTEGER) IS
BEGIN
  RETURN getClientStatus(p_client_id => p_client_id,
                         p_status_date => SYSDATE);
END;
```

Поддержка перегружаемых программ позволяет не придумывать различные имена для функций с одинаковой по смыслу функциональностью, но отличающимися типами параметров. Например, вместо того, чтобы помнить названия процедур printPerson, printPassport, … и постоянно добавлять к ним новые подобные процедуры, гораздо проще знать, что все составные типы данных печатаются одной перегруженной процедурой print, имеющей несколько версий.

```
print(p_person t_person), print(p_passport t_passport), …
```

Создание пакетов

Спецификацию пакета принято начинать с объявления пользовательских типов данных (подтипов имеющихся в языке скалярных типов, записей PL/SQL, коллекций), потом обычно объявляются курсоры, затем константы и переменные (в том числе и пользовательских типов, объявленных в спецификации выше). После всех этих объявлений помещают заголовки глобальных пакетных процедур и функций.

Создадим спецификации двух пакетов. В спецификации package1 сделаем объявления, которые потом будем использовать в теле пакета package2.

```
CREATE TABLE tab1(at1 NUMBER, at2 DATE);

SQL> CREATE OR REPLACE PACKAGE package1 AS
  2
  3    at1_treshold CONSTANT NUMBER := 10;
  4
  5    at1_treshold_excess EXCEPTION;
  6
  7    CURSOR c_tab1_count IS SELECT COUNT(*) c FROM tab1;
  8
  9  END;
 10  /
Package created.

SQL> CREATE OR REPLACE PACKAGE package2 AS
  2
  3    PROCEDURE tab1_insert(p_at1 IN tab1.at1%TYPE,p_at2 IN tab1.at2%TYPE);
  4
  5    FUNCTION tab1_count RETURN INTEGER;
  6
  7  END;
  8  /
Package created.
```

Реализуем в теле пакета package2 **процедуру** tab1_insert **добавления строк в таблицу** tab1 **и функцию** tab1_count**, возвращающую число строк в таблице.**

```
SQL> CREATE OR REPLACE PACKAGE BODY package2 AS
  2
  3    PROCEDURE check_at1(p_at1 IN tab1.at1%TYPE) IS
  4    BEGIN
  5      IF p_at1 > package1.at1_treshold THEN
  6        RAISE package1.at1_treshold_excess;
  7      END IF;
  8    END;
  9
 10    PROCEDURE tab1_insert(p_at1 IN tab1.at1%TYPE,
 11                          p_at2 IN tab1.at2%TYPE) IS
 12    BEGIN
 13      check_at1(p_at1);
 14      INSERT INTO tab1 VALUES(p_at1,p_at2);
 15    END;
 16
 17    FUNCTION tab1_count RETURN INTEGER IS
 18      result INTEGER;
 19    BEGIN
 20      OPEN package1.c_tab1_count;
 21      FETCH package1.c_tab1_count INTO result;
 22      CLOSE package1.c_tab1_count;
 23      RETURN result;
 24    END;
 25
 26  END;
 27  /
Package body created.
```

В теле пакета package2 **используются константа, пользовательское исключение и явный курсор, объявленные в спецификации пакета** package1**.**

Для обращения к глобальным пакетным программным элементам нужно указывать перед их именами имя пакета.

```
SQL> BEGIN
  2    DBMS_OUTPUT.PUT_LINE(package1.at1_treshold);
  3  END;
  4  /
10
PL/SQL procedure successfully completed.
```

Локальные процедуры, функции, переменные и другие программные конструкции не видны вне тела пакета (снаружи).

Приведем соответствующий пример:

```
SQL> BEGIN
  2    DBMS_OUTPUT.PUT_LINE(package2.check_at1(15));
  3  END;
  4  /
  DBMS_OUTPUT.PUT_LINE(package2.check_at1(15));
                               *
ERROR at line 2:
ORA-06550: line 2, column 33:
PLS-00302: component 'CHECK_AT1' must be declared
ORA-06550: line 2, column 3:
PL/SQL: Statement ignored
```

Вызовем функцию созданного пакета:

```
SQL> BEGIN
  2    package2.tab1_insert(1,'A');
  3    DBMS_OUTPUT.PUT_LINE('Rows in tab1 - '||package2.tab1_count());
  4  END;
  5  /
Rows in tab1 - 1
PL/SQL procedure successfully completed.

SQL> BEGIN
  2    package2.tab1_insert(20,'A');
  3  END;
  4  /
BEGIN
*
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at "U1.PACKAGE2", line 6
ORA-06512: at "U1.PACKAGE2", line 13
ORA-06512: at line 2
```

При втором вызове было инициировано пользовательское исключение `at1_treshold_excess`**, объявленное в спецификации пакета** `package1`**.**

Состояние пакетов

Состоянием пакета (package state) называется совокупность текущих значений его глобальных и локальных переменных и констант, а также текущих состояний курсоров (курсор открыт или закрыт, а

также значения атрибутов курсора, отражающие в том числе сколько строк было считано из открытого курсора к настоящему времени).

Состояния пакетов сохраняется в течение всей жизни сессии пользователя. Очень важно то, что состояния пакетов у каждой сессии пользователя свое. Нельзя «подсмотреть» или «подправить» значения пакетных переменных или считать строку курсора из состояния пакета другой сессии. Во многих приложениях на этом построены специфические системы управления доступом.

Пакетные переменные используются для решения следующих задач:

- кэширование постоянно использующихся в программах PL/SQL данных, например, справочников;
- обмен данным между программами PL/SQL в сессии.

В определенном смысле состояние пакетов — это аналог реестра Windows, в который программы для Windows записывают свои параметры в виде пар «параметр-значение». Эти значения параметров из реестра Windows обычно используются во время работы программы, сохраняются при завершении ее работы и считываются программой при следующем ее запуске. В этом сравнении отличие пакетов Oracle от реестра Windows в том, что состояние пакетов для завершившихся сессий пользователя теряется, то есть их можно рассматривать как «временный реестр» (реестр на время сессии)[1]. Состояние пакета используется им же при повторном вызове его процедур и функций во время одной сессии или используется другими программами PL/SQL.

Жизненный цикл состояния пакета содержит следующие этапы:

- инициализация состояния пакета;
- работа с программными элементами пакета пользователя (обращение к переменным, а также изменение их значений);
- сброс состояния пакета.

---

[1] На PL/SQL можно запрограммировать сохранение в таблицах базы данных состояния пакетов при завершении сессии пользователя и восстановление состояния пакетов по этим сохраненным данным для новых сессий.

Инициализация состояния пакета

Состояние пакета инициализируется тогда, когда в ходе сессии пользователя впервые происходит обращение к программному элементу, объявленному в спецификации этого пакета (вызывается объявленная в спецификации процедура или функция, считывается или изменяется значение глобальной пакетной переменной и т. п.).

При инициализации пакета выполняются следующие действия:

- создание в памяти экземпляров переменных и констант;
- присваивание переменным и константам значений, указанных в их объявлениях;
- выполнение кода секции инициализации пакета, представляющей собой анонимный блок в самом конце тела пакета.

Обычно в секции инициализации как раз и происходит размещение в пакетных переменных часто используемых в коде PL/SQL данных, например, справочников.

Справочники хранятся в таблицах базы данных и для часто выполняющихся операций кодирования и раскодирования (получения по термину кода и наоборот) каждый раз приходится выполнять SQL-запросы к этим таблицам. Это приводит к заметным затратам системных ресурсов, в том числе из-за частых переключений контекста: ... – SQL – PL/SQL – SQL – PL/SQL – ....

Для снижения этого эффекта справочники можно один раз прочитать из таблиц базы данных и закэшировать в виде пакетных переменных-коллекций, а далее операции кодирования и раскодирования проводить по этим коллекциям и в контекст SQL не переключаться.

Именно в этом и заключается одно из отмеченных выше достоинств использования пакетов — повышение производительности приложений за счет кэширования постоянно использующихся в приложении данных.

Приведем пример кэширования справочника кодов валют.

```
CREATE TABLE crcy_dictionary (crcy_code INTEGER,crcy_name VARCHAR2(100));

INSERT INTO crcy_dictionary VALUES(840,'Доллар США');
INSERT INTO crcy_dictionary VALUES(643,'Российский рубль');
INSERT INTO crcy_dictionary VALUES(978,'Евро');
INSERT INTO crcy_dictionary VALUES(986,'Бразильский реал');
```

```
CREATE OR REPLACE PACKAGE pack_dict_decode AS
  FUNCTION get_crcy_name(p_crcy_code in INTEGER)
                        RETURN crcy_dictionary.crcy_name%TYPE;
END;

CREATE OR REPLACE PACKAGE BODY pack_dict_decode AS

  TYPE t_crcy_dict IS
        TABLE OF crcy_dictionary.crcy_name%TYPE INDEX BY PLS_INTEGER;
  CURSOR c_crcy_dictionary IS SELECT * FROM crcy_dictionary;

  g_crcy_dict t_crcy_dict;

  FUNCTION get_crcy_name(p_crcy_code in INTEGER)
                        RETURN crcy_dictionary.crcy_name%TYPE IS
    ret crcy_dictionary.crcy_name%TYPE;
  BEGIN
    IF g_crcy_dict.EXISTS(p_crcy_code) THEN
      ret := g_crcy_dict(p_crcy_code);
    ELSE
      ret := 'Не определен';
    END IF;
    RETURN ret;
  END;

  PROCEDURE crcy_dict_ini IS
    TYPE t_crcy_dict_row_tab IS TABLE OF crcy_dictionary%ROWTYPE;
    l_crcy_dict_row_tab t_crcy_dict_row_tab;
  BEGIN
    OPEN c_crcy_dictionary;
    FETCH c_crcy_dictionary BULK COLLECT INTO l_crcy_dict_row_tab;
    CLOSE c_crcy_dictionary;
    FOR i IN 1..l_crcy_dict_row_tab.COUNT LOOP
      g_crcy_dict(l_crcy_dict_row_tab(i).crcy_code) :=
                                  l_crcy_dict_row_tab(i).crcy_name;
    END LOOP;
  END;

-- секция инициализации пакета
BEGIN
   crcy_dict_ini();
END;

SQL> BEGIN
  2    DBMS_OUTPUT.PUT_LINE('643:'||pack_dict_decode.get_crcy_name(643));
  3    DBMS_OUTPUT.PUT_LINE('771:'||pack_dict_decode.get_crcy_name(771));
  4  END;
  5  /
643: Российский рубль
771: Не определен

PL/SQL procedure successfully completed.
```

Сброс состояния пакетов

Состояние пакета штатно сбрасывается при завершении сессии пользователя, а также при перекомпиляции пакета или изменений объектов базы данных, от которых он зависит (в ходе таких изменений пакет получает статус invalid).

В то же время может возникнуть необходимость специально сбросить состояния пакетов для сессии без ее завершения, например, для повторного помещения в пакетные переменные изменившегося содержимого кэшируемых справочников (перечитывание справочников) или просто для освобождения памяти.

Для сброса состояния пакетов предназначена процедура RESET_PACKAGE встроенного пакета DBMS_SESSION.

Пусть пакет package1 имеет пакетную переменную v1.

```
SQL> BEGIN
  2    package1.v1:='A';
  3    DBMS_OUTPUT.PUT_LINE(package1.v1);
  4  END;
  5  /
A
PL/SQL procedure successfully completed.

SQL> BEGIN
  2    DBMS_SESSION.RESET_PACKAGE;
  3  END;
  4  /
PL/SQL procedure successfully completed.

SQL> BEGIN
  2    DBMS_OUTPUT.PUT_LINE(NVL(package1.v1,'v1 имеет значение NULL'));
  3  END;
  4  /
v1 имеет значение NULL
PL/SQL procedure successfully completed.
```

После сброса состояния пакетов для сессии пакеты повторно инициализируются при первых обращениям к их программным элементам.

Удаление и изменение пакетов

Для удаления спецификации пакета и его тела используются следующие DDL-команды:

```
DROP PACKAGE [имя_схемы.]имя_пакета
DROP PACKAGE BODY [имя_схемы.]имя_пакета
```

Удалим спецификацию нашего пакета `package1`:

```
SQL> DROP PACKAGE package1;
Package dropped.
```

Напомним, что при удалении спецификации пакета автоматически удаляется его тело. Кроме создания и удаления, спецификации и тела пакетов можно изменять DDL-командами

```
ALTER PACKAGE имя_пакета COMPILE [DEBUG]
```

```
ALTER PACKAGE BODY имя_пакета COMPILE [DEBUG]
```

При выполнении DDL-команд `ALTER` происходит перекомпиляция байт-кодов по хранящемся в базе данных исходным текстам.

Триггеры

Триггер базы данных — это хранимая в базе данных программа, которая автоматически запускается при наступлении событий, указанных при создании триггера. В книгах на английском языке часто встречается выражение «triggers fires», то есть триггеры «зажигаются».

Следует отметить, что триггеры занимают особое место среди видов хранимых программ на PL/SQL. Ранее отмечалось, что реализация серверной бизнес-логики возможна без использования PL/SQL — в виде программ на языках высокого уровня Java, C++, работающих на серверах приложений или прямо на серверах баз данных. В то же время сделать так, чтобы при наступлении событий с данными гарантированно происходили одни и те же сопровождающие действия, можно только с помощью триггеров.

Дело в том, что с базой данных могут работать несколько приложений, в которых сопровождающие действия с данными, вообще говоря, могут быть реализованы по-разному. Кроме того, изменения в данных могут вноситься и выполнением предложений SQL в SQL*Plus или Quest SQL Navigator, при этом нет никакой гарантии, что необходимые сопровождающие действия будут выполнены правильно и будут выполнены вообще. Триггеры же «вешаются» на операции с данными и работают как часы, вне зависимости от того, кто и из какого приложения эти операции выполнил.

Назначение триггеров

Триггеры используются для решения следующих задач:

- реализация серверной бизнес-логики в рамках концепции активных баз данных;
- реализация динамических ограничений целостности;
- ведение журналов аудита действий с данными;
- автоматизация администрирования баз данных.

Триггеры — важнейший механизм для так называемых активных баз данных, которые являются не пассивными системами хранения, а активно реагируют на изменения в данных путем генерации различных событий и их обработки.

В литературе приводятся самые разные примеры таких событий и реагирования на них. Это может быть простая серверная бизнес-логика, когда после добавления данных о платеже триггер на это событие увеличивает на внесенную сумму баланс соответствующего лицевого счета. Это может быть и реализация весьма изощренной бизнес-логики вида «Если клиент два раза подряд не дозвонился в наш call-центр, то в качестве компенсации для повышения лояльности следует на его лицевой счет начислить 100 бонусных баллов и отправить ему сообщение об этом в мессенджере WhatsApp». В этом случае реализация бизнес-логики осуществляется в триггере на добавление строк в таблицу логов звонков в call-центр.

Помимо реализации бизнес-логики в рамках концепции активных баз данных, триггеры используются для выполнения всевозможных проверок допустимости действий над данными в таблицах (сюда относится и реализация динамических ограничений целостности),

проверок правомерности создания объектов баз данных, предоставления привилегий и т. п.

Отношение к триггерам в среде специалистов по обработке данных двоякое и к тому же меняется со временем. Есть те, кто на дух не переносит триггеры, и не готов даже обсуждать возможность их использования в своих проектах. Как правило, это разработчики приложений, требующие, чтобы за все аспекты работы с данными отвечали клиентские приложения, в том числе и за активную реакцию на события, которые происходят с данными. Некоторые авторитетные специалисты в области технологий Oracle на основе опыта своей работы с течением времени пришли к выводу, что триггеры, как правило, используются неправильно и являются одним из основных источников ошибок. По их мнению, триггеры вызывают побочные эффекты, о триггерах забывают, что приводит к ненужным неожиданностям, триггеры реализуют ограничения целостности, и в то же время часто не включаются в состав объектов, для которых формируются DDL-команды при извлечении схемы базы данных и т. д.

В марте 2007 года Томас Кайт написал в своем блоге.

...Things I don't like:

- generic implementations that are not necessary;
- triggers;
- `WHEN OTHERS` (not followed by `RAISE`!!);
- triggers;
- not using bind variables;
- triggers.

Мы просто оставим это здесь без перевода и комментариев:

I hate triggers, i hate autonomous transactions, i hate `WHEN OTHERS`. If we removed those three things from PL/SQL — we would solve 90% of all application bugs, i think... No kidding...

Moral to this story however is:

- avoid triggers unless you absolutely need them (and you hardly ever do);

- do nothing, that doesn't rollback in them — ever — unless you can live with the side effects (triggers can always fire more than once!);
- autonomous transactions in triggers are pure evil.

При этом Т. Кайт пишет, что в начале своей Oracle-карьеры считал триггеры хорошим инструментом и активно их использовал. И не он один поступал таким образом. Как следствие, триггеры есть в многих системах, которые еще десятилетия будут эксплуатироваться. Поэтому любой специалист по технологиям Oracle должен уметь разбираться в этой теме.

Виды событий для срабатывания триггеров

Долгое время имевшиеся в базах данных Oracle триггеры срабатывали только на добавление, удаление или изменение данных в таблицах. Постепенно перечень видов событий, на которые можно «навесить» триггер, расширялся и в версии Oracle 12c имеется три вида таких событий:

- выполнение DML-предложений SQL добавления, изменения и удаления данных таблиц в таблицах — `INSERT, UPDATE, DELETE`;
- выполнение DDL-команд (команд создания, изменения и удаления объектов базы данных — `CREATE, ALTER, DROP` и некоторых других);
- события уровня базы данных (запуск и остановка базы данных, возникновение системных ошибок и т. п.).

Для реализации серверной бизнес-логики и динамических ограничений целостности обычно используются триггеры, срабатывающие на выполнение предложений `INSERT, UPDATE, DELETE`. Триггеры для остальных двух видов событий, как правило, используются администраторами баз данных для решения задач администрирования.

Триггеры на выполнение DML-предложений

Каждый триггер на выполнение предложений INSERT, UPDATE, DELETE «навешивается» на одну конкретную таблицу и имеет три основные настройки:

- набор предложений SQL INSERT, UPDATE, DELETE, при выполнении которых будет срабатывать триггер[1];
- тип срабатывания — до (BEFORE) или после (AFTER) внесения изменений в данные в ходе выполнения предложения SQL, вызвавшего срабатывание триггера;
- сколько раз триггер будет срабатывать — один раз или по числу обработанных предложением SQL строк.

Рассмотрим эти настройки подробнее.

Для одного триггера можно указать любую непустую комбинацию из трех предложений INSERT, DELETE, UPDATE (всего получается $2^3-1=7$ комбинаций). Если эта комбинация включает предложение UPDATE, то могут быть указаны конкретные столбцы таблицы, значения которых должны изменяться предложениями UPDATE, чтобы вызвать срабатывание триггера.

По количеству срабатываний триггеры делятся на два вида[2]:

- триггеры уровня предложения (statement-level triggers) — срабатывают один раз при выполнении вызвавшего срабатывание предложения SQL;
- триггеры уровня строки (row-level triggers) — срабатывают на каждой строке, обрабатываемой вызвавшим срабатывание триггера предложением SQL.

Триггер уровня предложения при выполнении в базе данных предложения SQL, на которое он настроен, срабатывает всегда и срабатывает ровно один раз. А вот триггер уровня строки может не сработать ни разу, если предложение SQL не обработало ни одной строки.

---

[1] Триггеров на выполнение предложений SELECT в Oracle нет, и никогда не было.

[2] Триггеры с типом срабатывания «вместо» (instead of DML triggers) и составные триггеры (compound triggers) в книге не рассматриваются.

Если же предложение SQL обработало три строки, то триггер уровня строки сработает три раза, обработка десяти строк вызовет десять срабатываний такого триггера и так далее.

Условие срабатывания триггера уровня строки может быть уточнено дополнительным логическим условием в конструкции WHEN команды CREATE TRIGGER.

Команда создания триггера на выполнение DML-предложений имеет следующий синтаксис:

```
CREATE [OR REPLACE] TRIGGER имя_триггера
{BEFORE | AFTER} -- тип срабатывания
{ INSERT | DELETE | UPDATE | UPDATE OF список столбцов } ON имя таблицы
[FOR EACH ROW]   -- триггер уровня строки
[WHEN (...)]     -- дополнительное логическое условие срабатывания
остальные разделы блока PL/SQL (объявлений,исполняемый,обработки исключений)
END;
```

В коде триггеров можно использовать специфичные средства:

- операционные директивы INSERTING, UPDATING, DELETING;
- псевдозаписи :NEW и :OLD (только для триггеров уровня строки).

Операционные директивы

Операционные директивы INSERTING, UPDATING, DELETING предназначены для идентификации предложения SQL, вызвавшего срабатывание триггера. Так как при создании триггера может указываться любая непустая комбинация из трех предложений INSERT, UPDATE, DELETE, то с помощью операционных директив INSERTING, UPDATING, DELETING внутри блока PL/SQL можно реализовать отдельные ветви потока команд для каждого из этих предложений.

Пусть, например, триггер срабатывает на INSERT и на DELETE, тогда исполняемый раздел блока триггера может быть построен следующим образом:

```
CASE
  WHEN INSERTING THEN
    логика обработки при срабатывании на INSERT
  WHEN DELETING THEN
    логика обработки при срабатывании на DELETE
END CASE;
```

Псевдозаписи :NEW и :OLD

Интуитивно ясно, что внутри триггеров уровня строки должна быть возможность обращаться к значениям столбцов строк, на которых срабатывают триггеры этого вида.

При каждом запуске триггера уровня строки виртуальная машина PL/SQL создает и заполняет две структуры данных — псевдозаписи :NEW и :OLD. Их структура идентична структуре записи PL/SQL, объявленной с помощью атрибута %ROWTYPE, то есть псевдозапись имеет все атрибуты с такими же именами и типами данных, какие есть столбцы у таблицы, на которую «навешен» триггер[1]. В атрибутах псевдозаписи :OLD находятся исходные значения столбцов строки, на которой сработал триггер, а в атрибутах псевдозаписи :NEW — новые значения столбцов.

Перечислим понятные ограничения, касающиеся этих псевдозаписей:

- у триггеров для INSERT нет данных в атрибутах :OLD;
- у триггеров для DELETE нет данных в атрибутах :NEW и изменять их нельзя;
- значения атрибутов :OLD изменять нельзя;
- значения атрибутов :NEW можно изменять в BEFORE-триггерах.

Полностью сведения о значениях атрибутов псевдозаписей :NEW и :OLD приведены в следующей таблице.

Таблица 6. Псевдозаписи :NEW и :OLD.

SQL	:OLD	:NEW
INSERT	NULL	значения столбцов после добавления
UPDATE	значения до изменения	значения столбцов после изменения
DELETE	значения перед удалением	NULL

[1] Дополнительно псевдозаписи :NEW и :OLD имеют еще атрибут с именем ROWID, в котором проставляется ROWID текущей строки таблицы.

Если для таблицы имеется несколько BEFORE-триггеров, то в ходе срабатываний друг за другом они могут несколько раз изменять значения псевдозаписи :NEW и каждый срабатывающий триггер будет видеть текущее ее состояние — после последнего изменения.

То обстоятельство, что в триггерах уровня строки со срабатыванием на INSERT и UPDATE значения атрибутов псевдозаписи :NEW можно изменять, позволяет подменять в триггере новые значения столбцов обрабатываемых этими предложениями SQL строк. Иными словами, если какое-нибудь предложение UPDATE делало в таблице из семерок восьмерки, то в конечном итоге в базе могут оказаться девятки, подмена на которые была выполнена в BEFORE-триггере. Как отмечалось ранее, наличие таких неожиданных эффектов при выполнении предложений SQL — это одна из причин считать использование триггеров плохой практикой.

Пример использования триггера

Пусть таблица tab1 создана и заполнена следующим образом:

```
CREATE TABLE tab1 (at1 NUMBER);
INSERT INTO tab1 VALUES(1);
INSERT INTO tab1 VALUES(3);
INSERT INTO tab1 VALUES(5);
```

Создадим триггер, который выдает ошибку, если значение столбца добавляемой строки слишком уклоняется от среднего значения для текущего состояния таблицы. В роли меры слишком большого уклонения выберем широко применяемое в инженерной практике правило «трех сигм»:

```
SQL> CREATE OR REPLACE TRIGGER trig_tb1
  2  BEFORE INSERT ON tab1 FOR EACH ROW
  3  DECLARE
  4    stat_avg NUMBER;
  5    stat_std NUMBER;
  6    stat_n   NUMBER;
  7  BEGIN
  8    SELECT COUNT(at1),SUM(at1),STDDEV(at1)
  9    INTO stat_n,stat_avg,stat_std FROM tab1;
 10    IF (ABS(stat_avg-stat_n*(:NEW.at1))/(SQRT(stat_n)*stat_std)>3) THEN
 11      RAISE_APPLICATION_ERROR(-20002, 'Слишком большое уклонение');
 12    END IF;
 13  END;
 14  /
Trigger created.
```

```
SQL> INSERT INTO tab1 VALUES(4);
1 row created.

SQL> INSERT INTO tab1 VALUES(7);
INSERT INTO tab1 VALUES(7)
            *
ERROR at line 1:
ORA-20002: Слишком большое уклонение
ORA-06512: at "U1.TRIG_TB1", line 9
ORA-04088: error during execution of trigger 'U1.TRIG_TB1'

SQL> SELECT * FROM tab1;
   AT1
------
     1
     3
     5
     4
```

При добавлении значения 4, достаточно близкого к среднему, исключение в триггере не инициируется. При добавлении значения 7, определяется большое уклонение от среднего, инициируется исключение и новая строка в таблицу не добавляется.

Если код триггера содержит ошибки, то он все равно будет создан, но выполнение предложений SQL, на которые он должен срабатывать, будет завершаться ошибкой. Такие триггеры следует или удалить, или исправить, или временно отключить командой ALTER TRIGGER … DISABLE.

Использование триггеров с различными настройками

Возможные значения трех настроек дают 12 вариантов событий для срабатывания триггеров для выполнения DML-предложений:

12=2 (BEFORE/AFTER) * 2 (уровня строки / предложения) * 3 (INS/UPD/DEL)[1]

[1] Выше было сказано, что триггер настраивается на любой из семи непустых наборов из трех предложений INSERT, UPDATE, DELETE, то есть третий множитель на самом деле равен не трем, а семи, и общее число вариантов событий равно 2*2*7=28. Однако на практике обычно для каждого предложения создается отдельный триггер, в литературе и документации по PL/SQL тоже общепринято рассматривать 12 вариантов.

Триггеры уровня предложения SQL часто используются для реализации правил, определяющих возможность выполнения предложения SQL. Например, пусть в некоторой организации нельзя оформлять пропуска посетителям в нерабочее время. Это требования может быть реализовано BEFORE-триггером для предложения INSERT, «навешенным» на таблицу пропусков. Внутри этого триггера надо проверять, что текущее время находится в заданном интервале рабочих часов 09:00-18:00, а текущий день не является выходным. Если эта проверка не выполняется, то в триггере инициируется исключение. Если в BEFORE-триггере инициируется исключение, то до добавления записей посредством предложения INSERT в таблицу пропусков дело не дойдет, что и требуется.

Триггеры уровня строки обычно используются для реализации собственно бизнес-логики. Можно считать, что каждая добавленная, удаленная или измененная строка в таблице — это отдельное событие, которое требует своей обработки. Например, если предложение DELETE удаляет из таблицы платежей несколько ошибочно добавленных в нее строк, то требуется по каждому удаленному платежу изменить (уменьшить) баланс лицевого счета, на который в свое время поступил этот платеж. Понятно, что код, осуществляющий это действие, должен выполняться для каждой удаленной строки, то есть в триггере уровня строки.

Рассмотрим, какие типы триггеров целесообразно использовать для решения двух типовых задач с учетом имеющихся ограничений на работу с псевдозаписями :NEW и :OLD:

- для модификации (подмены) значений строк с помощью :NEW следует использовать BEFORE-триггер уровня строки (потому что изменения в атрибутах :NEW возможны только в BEFORE-триггерах);
- для проверки (validation) новых значений столбцов обрабатываемой строки следует использовать AFTER-триггер уровня строки.

Вообще говоря, проверки новых данных можно делать и в BEFORE-триггере (:NEW в таких триггерах доступна и на чтение и на запись), однако так делать можно только в том случае, когда BEFORE-триггер один и в нем осуществляется и подмена значений столбцов и их

проверка. Порядок срабатывания триггеров одного типа в Oracle до недавнего времени был не определен[1]. Поэтому если триггеров уровня строки на одно событие несколько, то триггер, подменяющий значения столбцов, может сработать и после триггера с проверкой, выставив некорректное с точкой зрения проверки значения (проверка окажется преждевременной). Такой ситуации не возникнет для AFTER-триггеров, которые «видят» псевдозапись :NEW, которая теперь точно никак уже не изменится (изменения строки уже внесены в блоки данных таблицы предложением SQL и поменять строку там еще раз ни в одном AFTER-триггере невозможно). Именно окончательную версию :NEW следует проверить на корректность в AFTER-триггере.

Таким образом, общее правило для триггеров уровня строки такое: «подменяем значения столбцов обрабатываемых строк на новые в BEFORE-триггерах, проверяем новые значения в AFTER-триггерах».

Если триггер реализует реакцию на совершение какого-либо события, то выполнять его правильно после предложения SQL, относящегося к этому событию. Например, если требуется обновлять баланс по итогам добавления нового платежа, то следует делать это AFTER-триггером уже после успешного добавления строки в таблицу платежей, так как баланс логично обновлять только после того, как успешно прошел платеж.

Триггеры в транзакциях

Выполняемые в коде триггера предложения SQL являются частью транзакции, в которую входит предложение SQL, вызвавшее срабатывание триггера. Все предложения SQL в коде триггера выполняются на том же «срезе» базы данных, что и вызвавшее срабатывание триггера предложение SQL[2]. Это распространяется на изменения, внесенные другими транзакциями, их в теле триггера не видно. Если же в ходе выполнения одного предложения SQL происходит не-

---

[1] В версии Oracle 11g появилась конструкция FOLLOWS, с помощью которой можно для триггера указать, после запуска какого однотипного триггера он должен запускаться (with the FOLLOWS clause you can specify after which other trigger of the same type the trigger should fire). То есть с помощью этой конструкции можно задать очередь срабатываний триггеров.

[2] То есть выполняются на тот же SCN (system change number), который был на начало выполнения предложения SQL, на которое сработал триггер.

сколько срабатываний триггеров, то предложения SQL каждого сработавшего триггера видят изменения, сделанные на предыдущих срабатываниях. Все как всегда — чужие изменения на уровне отдельного предложения SQL не видны и в транзакции всегда видны свои изменения.

Отметим следующие важные обстоятельства:

- если в триггере будет инициировано необработанное исключение, то вызвавшее срабатывание триггера предложение SQL завершится с ошибкой и будет выполнена отмена и всех изменений, сделанных предложением SQL, и всех изменений, сделанных всеми триггерами на него (в ходе отмены до неявно установленной точки сохранения перед предложением);
- в триггере нельзя выполнять команды фиксации и отмены транзакций `COMMIT` и `ROLLBACK` (написать в теле триггера команды `COMMIT` или `ROLLBACK` можно — триггер будет успешно создан, но ошибка возникнет на этапе выполнения).

В примере с запретом выдачи пропусков в нерабочее время следует использовать `BEFORE`-триггер уровня предложения. Отмена изменений происходить не будет, так как не будет самих изменений данных — исключение в триггере будет инициировано еще до выполнения `INSERT` в таблицу пропусков. Если же в примере с обновлением триггером баланса после поступления платежа произойдет необработанная ошибка в триггере, то сам платеж, на добавление которого сработал триггер, тоже будет отменен — будет отменено добавление строки в таблицу платежей (новая строка платежа «исчезнет»).

Транзакция после ошибки в триггере остается в активном статусе, то есть сама по себе не отменяется и не фиксируется. Просто завершилось с ошибкой одно из входивших в нее предложений SQL, всю транзакцию потом можно будет зафиксировать или отменить. Понятно, что если фиксируется или отменяется транзакция, то это относится и ко всем изменениям, сделанным триггерами, срабатывавшими на предложениях SQL транзакции.

При наличии `BEFORE`-триггера к строке происходит три обращения (на примере `UPDATE`):

- в режиме согласованного чтения строка отбирается предложением `UPDATE` для изменения (первое обращение);

- выполняется блокирование строки командой SELECT FOR UPDATE (второе обращение);
- срабатывает BEFORE-триггер и значения столбцов заблокированной строки передаются в его псевдозапись :OLD;
- в ходе изменения строки предложением UPDATE происходит третье обращение к строке в текущем состоянии.

Наличие AFTER-триггеров не приводит к дополнительным обращениям к строке. Блокирования строки не происходит, после отбора строки в режиме согласованного чтения и ее изменения в текущем состоянии срабатывает AFTER-триггер, которому передаются данные для заполнения псевдозаписей :NEW и :OLD. Как отмечалось выше, изменить значения столбцов строки он уже не сможет.

Последовательность срабатывания триггеров

Пусть, например, на некоторую таблицу «навешено» все 2*2=4 триггера со срабатыванием на предложение UPDATE:

- BEFORE-триггер уровня предложения SQL tr1;
- BEFORE-триггер уровня строки tr2;
- AFTER-триггер уровня строки tr3;
- AFTER-триггер уровня предложения SQL tr4.

Последовательность событий при выполнении предложения UPDATE, которое изменит, скажем, две строки в таблице, будет следующая:

- один раз сработает триггер tr1;
- на первой изменяемой строке сработает триггер tr2;
- выполнится изменение первой строки предложением UPDATE;
- на первой измененной строке сработает триггер tr3;
- на второй изменяемой строке сработает триггер tr2;
- выполнится изменение второй строки предложением UPDATE;
- на второй измененной строке сработает триггер tr3;
- один раз сработает триггер tr4.

Проверим возможность изменять значения атрибуты псевдозаписи :NEW, заодно и проиллюстрируем приведенную выше последовательность срабатывания триггеров:

```
CREATE TABLE tab5 (at1 INTEGER); INSERT INTO tab5 VALUES(5);

CREATE OR REPLACE TRIGGER before_statement BEFORE UPDATE ON tab5
BEGIN
  dbms_lock.sleep(2);
  DBMS_OUTPUT.PUT_LINE('Fire before statement-level trigger at '
             ||TO_CHAR(SYSDATE, 'DD.MM.YYYY HH24:MI:SS'));
END;

CREATE OR REPLACE TRIGGER before_row BEFORE UPDATE ON tab5 FOR EACH ROW
BEGIN
  dbms_lock.sleep(2);
  DBMS_OUTPUT.PUT_LINE('Fire before row-level trigger at '
             ||TO_CHAR(SYSDATE, 'DD.MM.YYYY HH24:MI:SS'));
  DBMS_OUTPUT.PUT_LINE(':OLD.at1='||:OLD.at1);
  DBMS_OUTPUT.PUT_LINE(':NEW.at1='||:NEW.at1);
  :NEW.at1 := 6;
  DBMS_OUTPUT.PUT_LINE('Set :NEW.at1='||:NEW.at1);
  DBMS_OUTPUT.PUT_LINE('Finish before row-level trigger');
END;

CREATE OR REPLACE TRIGGER after_statement AFTER UPDATE ON tab5
BEGIN
  dbms_lock.sleep(2);
  DBMS_OUTPUT.PUT_LINE('Fire after statement-level trigger at '
              ||TO_CHAR(SYSDATE, 'DD.MM.YYYY HH24:MI:SS'));
END;

CREATE OR REPLACE TRIGGER after_row AFTER UPDATE ON tab5 FOR EACH ROW
BEGIN
  dbms_lock.sleep(2);
  DBMS_OUTPUT.PUT_LINE('Fire after row-level trigger at '
              ||TO_CHAR(SYSDATE, 'DD.MM.YYYY HH24:MI:SS'));
  DBMS_OUTPUT.PUT_LINE(':OLD.at1='||:OLD.at1);
  DBMS_OUTPUT.PUT_LINE(':NEW.at1='||:NEW.at1);
  DBMS_OUTPUT.PUT_LINE('Finish after row-level trigger');
END;

SQL> UPDATE tab5 SET at1=10;

Fire before statement-level trigger at 18.01.2015 12:00:05

Fire before row-level trigger at 18.01.2015 12:00:07
:OLD.at1=5
:NEW.at1=10
Set :NEW.at1=6
Finish before row-level trigger
```

```
Fire after row-level trigger at 18.01.2015 12:00:09
:OLD.at1=5
:NEW.at1=6
Finish after row-level trigger

Fire after statement-level trigger at 18.01.2015 12:00:11

1 row updated.

SQL> select * from tab5;
       AT1
 ----------
         6
```

Меняли предложением UPDATE пятерку на десятку, в итоге в базе шестерка. Налицо неожиданный побочный эффект, по этой причине триггеры и не рекомендуют использовать.

У сервера Oracle для обеспечения согласованности изменений данных при необходимости осуществляется автоматический перезапуск предложений UPDATE и DELETE[1]. Перед перезапуском выполняется отмена до неявно установленной точки сохранения, в ходе которой в том числе отменяются изменения, сделанные сработавшими до перезапуска триггерами уровня строки. Затем в ходе повторной обработки строк эти триггеры срабатывают снова. Может случиться так, что эти строки окажутся другими, не теми, которые пытались обработать в первый раз. Чаще же происходят ситуации, когда триггеры срабатывают на одних и тех же строках и при первой (отмененной) обработке строк, и в ходе перезапуска. Таким образом, на одной строке один и тот же триггер уровня строки может сработать дважды.

Если в триггере не используются автономные транзакции, то ничего страшного в его повторных срабатываниях нет — сделанные во время первых срабатываний изменения будут отменены при автоматической отмене к неявной точке сохранения перед перезапуском. Зафиксированы будут только изменения, сделанные во время вторых срабатываний триггеров.

---

[1] Перезапуск выполнения предложений SQL, выполняющих изменение или удаление данных, рассматривается в параграфе «Управление транзакциями».

Использовать в триггерах автономные транзакции рекомендуется только для регистрации ошибок в журналах. Реализация бизнес-логики с помощью автономных транзакций, как правило, содержит ошибки обеспечения корректного многопользовательского доступа, которые обязательно произойдут рано или поздно.

Дополнительное условие срабатывания триггера

Срабатывание триггеров может существенно замедлить выполнение предложений SQL, особенно когда обрабатывается много строк и на каждой из них срабатывает триггер уровня строки. Этот триггер в соответствии с бизнес-логикой может для каких-то ситуаций не выполнять никаких операций с данными, но все равно его срабатывание на каждой строке будет ухудшать производительность.

В заголовке триггера после необязательного ключевого слова WHEN можно задать дополнительное логическое условие, сужающее область событий, при наступлении которых триггер запускается. Это очень ценная возможность, которая позволяет сделать так, чтобы какие-то строки обрабатывались без срабатывания триггера.

Рассмотрим соответствующий пример. Пусть имеется таблица платежей

```
CREATE TABLE payments (pay_date DATE,account INTEGER,
                       amount INTEGER, source VARCHAR2(20));
```

Платежи поступают сотнями тысяч и бывают двух типов — через кассы и через сайт. Для платежей, поступивших через сайт, требует проводить дополнительную обработку, для платежей через кассу этого не требуется.

```
CREATE OR REPLACE TRIGGER tr$payments$b$i
BEFORE INSERT ON payments FOR EACH ROW WHEN (NEW.source = 'online')
BEGIN
  dbms_output.put_line('Триггер сработал');
  -- process_onine_payment(:NEW.account,:NEW.amount);
END;

SQL> INSERT INTO payments VALUES(SYSDATE,3452,1000,'online');
Триггер сработал
1 row(s) inserted

SQL> INSERT INTO payments VALUES(SYSDATE,7854,500,'cashbox');
1 row(s) inserted
```

Видно, что во втором случае срабатывания триггера не было. Для этой же цели минимизации ненужного использования ресурсов сервера триггерами предназначена и возможность их временного отключения DDL-командой ALTER:

```
SQL> ALTER TRIGGER trig_tb1 DISABLE;
Trigger altered.

SQL> ALTER TRIGGER trig_tb1 ENABLE;
Trigger altered.
```

Для триггеров, срабатывающих при выполнении предложений UPDATE, также можно указать в конструкции UPDATE OF список столбцов, которые должны изменяться для того, чтобы триггер сработал. Все условия в заголовке и в конструкции WHEN проверяются без запуска триггера во время выполнения предложения SQL. По этой причине в конструкции WHEN можно использовать только встроенные функции SQL.

Чтобы подчеркнуть важность рассмотренного вопроса минимизации числа ненужного срабатываний триггеров, отметим, что по некоторым оценкам замедление выполнения DML-предложений из-за наличия одного триггера может составить до 30%.

Мутирующие таблицы

Мутирующая таблица (mutating table) — это таблица, строки которой в данный момент изменяются предложением SQL. Таблицы, строки в которых которые изменяются в результате ссылочных действий (ON DELETE CASCADE, ON DELETE SET NULL), также являются изменяющимися.

Предложения SQL в теле триггера уровня строки не могут обращаться к строкам любой таблицы, изменяющейся предложением SQL, на которое сработал триггер. При попытке такого обращения при выполнении триггера будет выдано сообщение об ошибке.

Приведем пример мутирующей таблицы:

```
CREATE TABLE tab1 (at1 INTEGER,at2 INTEGER);
INSERT INTO tab1 VALUES(1,1);
INSERT INTO tab1 VALUES(2,1);

SQL> CREATE OR REPLACE TRIGGER tr1
  2  BEFORE DELETE ON tab1 FOR EACH ROW
  3   BEGIN
```

```
  4     IF :OLD.at1=:OLD.at2 THEN
  5       UPDATE tab1 SET at2=NULL
  6       WHERE at2=:OLD.at1;
  7     END IF;
  8   END;
  9  /
Trigger created.

SQL> DELETE FROM tab1 WHERE at1=at2;
DELETE FROM tab1 WHERE at1=at2
            *
ERROR at line 1:
ORA-04091: table U1.TAB1 is mutating, trigger/function may not see it
ORA-06512: at "U1.TR1", line 2
ORA-04088: error during execution of trigger 'U1.TR1'
```

Причина запрета обращения к мутирующим таблицам из триггеров уровня строки заключается в том, что для предложений SQL не определен порядок обработки строк. Рассмотрим гипотетический пример того, как могла бы происходить обработка строк, если бы такого запрета не было.

Пусть таблица `tab1` **имеет один столбец** `at1` **и пять строк:**

```
CREATE TABLE tab1 (at1 INTEGER)

SQL> SELECT * FROM tab1;
       AT1
----------
         1
         2
         3  -- смотрим срабатывание триггера на этой строке
         0
         4
```

Выполняем предложение

```
UPDATE tab2 SET at1=at1+1;
```

Пусть на каждой обрабатываемой строке срабатывает `AFTER`**-триггер уровня строки, в коде которого выполняется запрос**

```
SELECT COUNT(*) FROM tab2 WHERE at1<3
```

Смотрим результаты этого SQL-запроса для строки с тройкой при двух разных вариантах порядка обработки строк (`o,old` **— старое значение,** `n,new` **— новое значение):**

Первый вариант порядка обработки строк	Второй вариант порядка обработки строк
`|`[1] o1-> n2 o3-> n4 COUNT:return **3**:row(n2,o2,o0) o2-> o2 (пока не менялось) o0-> o0 (пока не менялось) o4-> o4 (пока не менялось)	`|` o1-> n2 o4-> n5 o0-> n1 o2-> n3 o3-> n4 COUNT:return **2**:row(n2,n1)

Для первого варианта порядка обработки строк (1,3,2,0,4) SQL-запрос в теле триггера возвращает число 3 для COUNT(*), для второго варианта (1,4,0,2,3) — число 2 (3<>2). То есть один и тот же запрос при одинаковом исходном содержимом таблицы в ходе срабатывания триггера на одной и той же строке может вернуть различные результаты.

Неоднозначности в результатах выполнения предложений SQL быть не должно, поэтому выполнение предложений SQL к мутирующим таблицам в триггерах уровня строки не допускается.

Запрет доступа к мутирующим таблицам относится только к триггерам уровня строки. Триггеры уровня предложения SQL могут и считывать, и записывать данные мутирующей таблицы. Это понятно — перед триггером уровня предложения «лежит» множество всех строк, обрабатываемых предложением SQL. Для AFTER-триггера они все уже обработаны, для BEFORE-триггера они все еще не обработаны. В таких условиях действия с данными в мутирующей таблице в триггере при любом исходном порядке строк в таблице будут завершаться с одинаковыми результатами.

Исключение из запрета доступа к мутирующим таблицам

У запрета доступа к мутирующим таблицам из триггеров уровня строки есть исключение.

Рассуждения строятся следующим образом. Запрет введен для недопущения неоднозначности результатов обращений из триггера к мутирующей таблице из-за отсутствия порядка обработки строк. По-

[1] Под чертой | значения, которые увидит запрос COUNT(*) в триггере.

нятно, что этой неоднозначности не будет, если предложение SQL обрабатывает ровно одну строку — в этом вырожденном случае обработка строк, очевидно, упорядочена. Таким предложением SQL является предложение INSERT.

Для предложений UPDATE и DELETE понять, сколько строк они обработают, находясь на первой из обработанных им строк, нельзя. Неясно, будет ли после этой строки потом обработана еще вторая, третья и последующие строки. В то же время сам синтаксис предложения INSERT предусматривает, что оно добавляет в таблицу ровно одну строку, поэтому в BEFORE-триггерах уровня строки для таких предложений INSERT можно обращаться к мутирующей таблице:

```
SQL> CREATE TABLE tab3 (at1 INTEGER);
Table created.

SQL> CREATE OR REPLACE TRIGGER tr$tab3$i
  2    BEFORE INSERT ON tab3 FOR EACH ROW
  3  DECLARE
  4    l_count INTEGER;
  5  BEGIN
  6    SELECT count(*) INTO l_count FROM tab3;
  7  END;
  8  /
Trigger created.

SQL> INSERT INTO tab3 VALUES (1);
1 row created.
```

В то же время, если выполнить специальную форму предложения INSERT — INSERT SELECT, которая потенциально может добавить в таблицу не одну, а несколько строк, произойдет ошибка.

```
SQL> INSERT INTO tab3 SELECT * FROM tab3;
INSERT INTO tab3 SELECT * FROM tab3
            *
ERROR at line 1:
ORA-04091: table U1.TAB3 is mutating, trigger/function may not see it
ORA-06512: at "U1.TR$TAB3$I", line 4
ORA-04088: error during execution of trigger 'U1.TR$TAB3$I'
```

Для срабатывания триггера на эту форму предложения INSERT запрет обращения к мутирующей таблице распространяется. Таким образом, исключение для однострочных предложений INSERT как нельзя лучше подтверждает общее правило.

Решения проблемы мутирующей таблицы

Для решения проблемы мутирующей таблицы применяются три основных способа:

- использование в триггерах автономных транзакций;
- использование составных триггеров (compound triggers);
- перенос логики триггеров уровня строки в триггеры уровня предложения SQL.

Существует известная техника решения проблемы мутирующей таблицы с условным наименованием «один пакет и три триггера»:

- создать BEFORE-триггер уровня предложения, который обнуляет «индекс» таблицы PL/SQL, объявленной как глобальная переменная в спецификации пакета;
- создать BEFORE-триггер уровня строки, который для каждой обработанной предложением SQL строки запоминает требуемые значения в записи таблицы PL/SQL;
- создать AFTER-триггер уровня предложения, выполняющий требуемые изменения по значениям, запомненным в таблице PL/SQL.

Автор считает возможным предостеречь читателя от применения подобных способов, особенно автономных транзакций (autonomous transactions in triggers are pure evil). Они работоспособны только в условиях однопользовательской обработки. Возникновение проблемы мутирующей таблицы, если ее не удалось решить изменением логики в коде самого триггера, следует рассматривать как повод для решения вовсе отказаться от триггера в этом случае и переработать логику обработки данных без него.

Реализация динамических ограничений целостности

Помимо бизнес-логики, триггеры используются для реализации динамических ограничений целостности.

Динамическим ограничение целостности (dynamic integrity constraint) называется динамически проверяемое ограничение, определяющее возможность перехода моделируемой предметной области из одного состояния в другое состояние. Это такие ограничения, которые невозможно реализовать в виде статических ограничений целостности для таблиц (первичных и внешних ключей, ограничений на уникальность и ограничений целостности, задаваемых предика-

том CHECK). Динамические ограничения целостности являются более сложными — не декларируемыми, а программируемыми. Рассмотрим пример такого ограничения.

Пусть в базе данных хранятся сведения о договорах клиентов и их лицевых счетах. Отношение между договорами и счетами — «один ко многим», то есть для одного договора есть несколько лицевых счетов.

```
CREATE TABLE contracts
(id     INTEGER PRIMARY KEY,
 num    VARCHAR2(10),
 status VARCHAR2(10));

CREATE TABLE accounts
(id             INTEGER,
 num            VARCHAR2(10),
 r$contract$id  INTEGER REFERENCES contracts,
 status         VARCHAR2(10));

INSERT INTO contracts VALUES(12,'562/323-21','operating');
INSERT INTO accounts VALUES(45,'321/21-1',12,'operating');
INSERT INTO accounts VALUES(46,'321/21-2',12,'closed');
```

Пусть имеется динамическое ограничение целостности — запрет закрытия контракта клиента до тех пор, пока не закрыты все его лицевые счета. Такое ограничение целостности можно реализовать BEFORE-триггером уровня строки.

```
SQL> CREATE OR REPLACE TRIGGER tr$contracts$u
  2    BEFORE UPDATE ON contracts FOR EACH ROW WHEN (NEW.status = 'closed')
  3  DECLARE
  4    l_account_count INTEGER;
  5  BEGIN
  6
  7    SELECT count(*) INTO l_account_count
  8    FROM accounts WHERE accounts.r$contract$id = :NEW.id
  9                            AND accounts.status <> 'closed';
 10
 11    IF l_account_count > 0 THEN
 12       RAISE_APPLICATION_ERROR(-20001,
 13            'У контракта '||:NEW.id||' имеются незакрытые лицевые счета');
 14    END IF;
 15
 16  END;
 17  /

Trigger created.
```

```
SQL> UPDATE contracts SET status='closed' WHERE contracts.id=12;
UPDATE contracts SET status='closed' WHERE contracts.id=12
       *
ERROR at line 1:
ORA-20001: У контракта 12 имеются незакрытые лицевые счета
ORA-06512: at "U1.TR$CONTRACTS$U", line 10
ORA-04088: error during execution of trigger 'U1.TR$CONTRACTS$U'

-- закрываем лицевые счет 12-го контракта
SQL> UPDATE accounts SET status='closed' WHERE r$contract$id=12;
2 rows updated.

-- теперь закрыть контракт можно
SQL> UPDATE contracts SET status='closed' WHERE contracts.id=12;
1 row updated.
```

Еще раз отметим, что использование триггеров, в том числе и для реализации динамических ограничений целостности, следует рассматривать только в случае крайней необходимости. Так, если ограничение целостности нельзя реализовать статически для существующей схемы базы данных, но такая возможность появится после внесения в схему изменений, то следует об этом подумать. Особенно на этапе проектирования, пока от схемы базы данных еще не стали зависеть приложения и другие компоненты системы.

Триггеры на создание, изменение и удаление объектов базы данных

Это относительно новый вид триггеров, срабатывающих при выполнении DDL-команд. Ранее рассматривались триггеры на события с данными в таблицах, эти же триггеры запускаются при событиях с самими таблицами, а также представлениями, последовательностями и другими объектами баз данных.

Команда создания триггера на создание, изменение и удаление объектов базы данных имеет следующий синтаксис:

```
CREATE [OR REPLACE] TRIGGER имя триггера
{BEFORE | AFTER} -- тип срабатывания
{событие с объектом базы данных } ON {база данных | схема}
[WHEN (...)]     -- дополнительное логическое условие
остальные разделы блока PL/SQL (объявлений,исполняемый,обработки исключений)
```

Под событиями с объектами базы данных понимается выполнение команд из фиксированного перечня: CREATE, ALTER, DROP, GRANT, REVOKE, TRUNCATE TABLE **и некоторые другие.**

Для получения в триггерах информации об объектах баз данных и о типах происходящих с ними событий, предназначены атрибутные функции.

В версии Oracle 12c имеется 20 атрибутных функций, приведем описание некоторых из них.

Таблица 7. Атрибутные функции.

Атрибутная функция	Описание функции
ORA_CLIENT_IP_ADDRESS	IP-адрес клиента
ORA_DICT_OBJ_NAME	имя объекта базы данных, связанного с DDL-командой, которая вызвала срабатывание триггера
ORA_DICT_OBJ_OWNER	владелец объекта, связанного с DDL-командой, которая вызвала срабатывание триггера
ORA_DICT_OBJ_TYPE	тип объекта, связанного с DDL-командой, которая вызвала срабатывание триггера
ORA_SYSEVENT	тип события, вызвавшего срабатывание триггера (например, CREATE, DROP или ALTER)

Триггеры рассматриваемого типа позволяют эффективно дополнить программной логикой имеющиеся в Oracle средства управления доступом на основе классических дискреционной, ролевой и мандатной моделей. Для иллюстрации этого с помощью триггеров внесем две настройки:

- запретим удаление любых таблиц (триггер tr$drop_table$disable);
- разрешим назначение привилегий только при подключениях к серверу баз данных сети с конкретного IP-адреса в локальной сети (триггер tr$check_grantee_ip).

Такого вида триггеры могут создаваться администраторами баз данных (администраторами безопасности) для повышения степени контроля за системой:

- «чтобы никто ни одной таблички не смог удалить ни при каких обстоятельствах — только я, причем несколько раз подумав и предварительно отключив триггер»;
- «назначать привилегии можно было только с моей рабочей станции».

```
SQL> CREATE OR REPLACE TRIGGER tr$drop_table$disable
  2  BEFORE DROP ON DATABASE
  3  BEGIN
  4    IF ORA_SYSEVENT = 'DROP'
  5      AND ORA_DICT_OBJ_TYPE = 'TABLE' THEN
  6      RAISE_APPLICATION_ERROR (
  7            -20000,
  8            'ERROR : Tables cannot be dropped in my database!');
  9    END IF;
 10  END;
 11  /
Trigger created.

SQL> DROP TABLE tab1;
DROP TABLE tab1
*
ERROR at line 1:
ORA-00604: error occurred at recursive SQL level 1
ORA-20000: ERROR : Tables cannot be dropped in my database!
ORA-06512: at line 4

SQL> CREATE OR REPLACE TRIGGER tr$check_grantee_ip
  2  BEFORE GRANT ON DATABASE
  3  DECLARE
  4    c_valid_ip CONSTANT VARCHAR2(20) := '192.168.0.8';
  5    l_current_ip        VARCHAR2(20);
  6  BEGIN
  7    l_current_client_ip := sys_context('USERENV','IP_ADDRESS');
  8    IF ORA_SYSEVENT = 'GRANT'
  9      AND l_current_client_ip <> c_valid_ip THEN
 10      RAISE_APPLICATION_ERROR (
 11            -20000,
 12           'ERROR: Grants from '||l_current_ip||' not allowed');
 13    END IF;
 14  END;
 15  /
Trigger created.

SQL> GRANT SELECT ON tab1 TO u1;
GRANT SELECT ON tab1 TO u1
              *
ERROR at line 1:
ORA-00604: error occurred at recursive SQL level 1
ORA-20000: ERROR: Grants from 127.0.0.1 not allowed
ORA-06512: at line 8
```

Триггеры на события базы данных

Триггеры на события базы данных запускаются при возникновении событий уровня базы данных. В Oracle 12c восемь таких типов событий, перечислим некоторые из них:

- STARTUP — открытие базы данных;
- SHUTDOWN — нормальное закрытие базы данных;
- SERVERERROR — возникновение ошибки;
- LOGON — создание сеанса;
- LOGOFF — нормальное завершение сеанса.

Ясно, что триггеры на эти события в основном используются для решения задач администрирования и обеспечения безопасности. Например, в триггерах на LOGON могут осуществляться дополнительные проверки правомерности создания сеанса (проверяться могут время создания сессии, IP-адрес клиента, название клиентского приложения), или устанавливаться переменные окружения сессии пользователя. В триггере на завершение сессии может собираться статистика о выполненных в ходе этой сессии операциях.

Триггеры на события базы данных обычно создаются самими администраторами баз данных или самыми опытными разработчиками прикладных систем.

Защита исходного кода

При создании хранимых программ PL/SQL в словаре-справочнике базы данных Oracle сохраняются и байт-код программ и их исходный код, который доступен для построчного просмотра в представлениях словаря-справочника данных DBA_SOURCE, ALL_SOURCE и USER_SOURCE.

```
SQL> CREATE OR REPLACE PROCEDURE test AS
  2    i INTEGER := 1;
  3  BEGIN
  4    SELECT i+1 INTO i FROM dual;
  5  END;
  6  /
Procedure created.

SQL> SELECT TEXT FROM USER_SOURCE
  2  WHERE name='TEST' AND type='PROCEDURE'
  3  ORDER BY line
  4  /
```

```
TEXT
------------------------------------------------------------------------
PROCEDURE test AS
  i INTEGER := 1;
BEGIN
  SELECT i+1 INTO i FROM dual;
END;
```

Именно из этих представлений словаря-справочника данных формируют исходный код хранимых программ GUI-клиенты (Quest SQL Navigator, TOAD, PL/SQL Developer) при открытии программ в Stored Program Editor и выполнении операций Extract DDL. Администратор базы данных в представлении DBA_SOURCE может посмотреть исходный код любой хранимой программы и фактически получается так, что все программное обеспечение, написанное на PL/SQL, является open-source software.

Разобраться в коде PL/SQL довольно легко и у кого-то может возникнуть желание внести изменения в логику чужой программы или в ее настройки. Довольно часто настройки программ PL/SQL «зашивают» в коде как локальные константы тел пакетов, и может возникнуть желание несанкционированно их поменять. Помимо этого на исходный код программ PL/SQL могут иметься права как на интеллектуальную собственность.

По этим причинам возникает необходимость защитить исходный код программ PL/SQL от чтения и изменения. Сделать так, чтобы исходный код не распространялся вместе с программами нельзя, но можно его специальным образом обработать — привести к нечитаемому виду. Мы будем использовать термин «нечитаемый код» («скрытый код»), а процесс приведения кода к нечитаемому виду называть сокрытием кода.

Для приведения кода PL/SQL к нечитаемому виду есть три средства:

- утилита wrap;
- встроенный пакет DBMS_DDL;
- встроенный пакет DBMS_WRAP (с Oracle 10g Release 2).

Общая схема сокрытия исходного кода PL/SQL следующая:

- программист разрабатывает программу PL/SQL как обычно;
- выгружает исходный код программы в текстовый файл (операция Extract DDL средства разработки на PL/SQL);

- обрабатывает исходный код в файле утилитой wrap;
- полученный DDL-скрипт с нечитаемым исходным кодом программы PL/SQL доставляется администратору, который «прогоняет» его в базе данных.

Утилита wrap работает из командной строки операционной системы. Для Windows ее исполняемый файл wrap.exe нужно запустить из подкаталога BIN домашнего каталога инсталляции Oracle. Формат вызова утилиты:

```
wrap iname=исходный_файл [oname=обработанный файл]
```

Обработаем исходный код нашей процедуры test.

```
C:\Temp>wrap.exe iname=test.sql oname=test_wrapped.sql

PL/SQL Wrapper: Release 11.2.0.2.0-Production on Wed Jan 13 17:51:44
2015

Copyright (c) 1993, 2009, Oracle.  All rights reserved.

Processing test.sql to test_wrapped.sql
```

Можно сразу посмотреть в текстовом редакторе получившийся файл test_wrapped.sql, но лучше мы сначала пересоздадим нашу процедуру, а потом посмотрим ее код в USER_SOURCES.

```
C:\Temp\sqlplus.exe u1/u1password @test_wrapped.sql

SQL*Plus: Release 11.2.0.2.0 Production on Wed Jan 13 17:57:02 2015
Copyright (c) 1982, 2010, Oracle.  All rights reserved.

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, Oracle Label Security, OLAP, Data Mining,
Oracle Database Vault and Real Application Testing options

Procedure created.

SQL> SELECT text FROM USER_SOURCE
  2  WHERE name='TEST' AND type='PROCEDURE'
  3  ORDER BY line
  4  /

TEXT
--------------------------------------------------------------------------------
PROCEDURE test wrapped
a000000
ab
abcd
```

```
...
abcd
7
51 8d
suQy2odKI6E8Sra9zNpbNb8IqrAwg5SXf8upynREacFBBrvc2NNbRvLsynXhgKU2wLXjHtmY
DTDLTIiGQltxbGfKvlKRlWFqVHtX8kGHrSOgyD3kaOe/J74vRQynV1uZUJv7WQW2hw==
```

Нечитаемый исходный код еще называют «wrap'ленный исходник». Отметим его следующие свойства:

- исходный текст процедуры `test` стал совершенно нечитаемым;
- после названия хранимой процедуры `test` есть слово `wrapped`, говорящее о проведенном сокрытии кода.

При сокрытии исходного кода PL/SQL следует учитывать следующие обстоятельства:

- сокрытие кода никак не влияет на функциональность программы PL/SQL, просто становится нечитаемым ее исходный код;
- общепринято оставлять читаемым спецификации пакетов (то есть интерфейсную часть) и «закрывать» их тела (в частности, так сделано практически для всех встроенных пакетов);
- нельзя делать нечитаемым исходный код триггеров (для этого следует вынести логику из триггеров в хранимые процедуры и функции с нечитаемым кодом и в триггерах только вызывать их);
- программы с нечитаемым кодом не могут быть созданы в базе данных версии ниже той, чем версия утилиты, которой они были обработаны;
- на каждую версию `wrap` рано или поздно найдется `unwrap` или `rewrap`.

Известно несколько программ, выполняющих обратное преобразование нечитаемого кода, для которых надо только аккуратно выгрузить нечитаемый код из словаря-справочника данных (обычная операция Extract DDL здесь непригодна). Имеются и публикации на эту тему.

Встроенные пакеты

В базах данных Oracle имеется большое количество встроенных пакетов PL/SQL, предназначенных для облегчения и унификации про-

цесса разработки на PL/SQL. Эти пакеты следует рассматривать как стандартные библиотеки, имеющиеся в других языках программирования, например, в C++ и Python. Встроенные пакеты PL/SQL также иногда называют поставляемыми пакетами (Oracle Supplied PL/SQL Packages).

Встроенные пакеты находятся в схеме пользователя SYS и имеют имена с префиксами DBMS и UTL. Число встроенных пакетов растет с выходом каждой новой версии сервера Oracle, вместе с Oracle 11g поставляется свыше 230 встроенных пакетов[1].

Некоторые встроенные пакеты, например, DBMS_SQL и DBMS_LOB используются при разработке программ PL/SQL очень часто, другие встроенные пакеты используются относительно редко. В литературе по PL/SQL обычно рассматриваются встроенные пакеты, предназначенные для организации ввода-вывода из программ PL/SQL и некоторые встроенные пакеты, предназначенные для автоматизации администрирования баз данных.

Далее приведены примеры использования следующих четырех встроенных пакетов:

- DBMS_JOB (управление заданиями);
- UTL_FILE (файловый ввод-вывод);
- DBMS_LOB (работа с большими объектами);
- DBMS_SQL (динамический SQL).

Управление заданиями

В большинстве вычислительных систем существует механизм, который позволяет автоматически запускать задания. Например, в операционной системе Microsoft Windows есть «Планировщик задач» (Task scheduler), который предоставляет возможность запланировать запуск программ в определенные моменты времени или через задан-

[1] Если считать и пакеты, размещенные в других стандартно поставляемых схемах (XDB, OUTLN, CTXSYS, DBSNMP и т. п.), то всего в Oracle 12c насчитывается более 1 000 встроенных пакетов. Если к этому добавить встроенные объектные типы (многие API реализованы посредством и пакетов и типов), то число превысит 3 000.

ные временные интервалы. В UNIX-подобных операционных системах (Linux, Oracle Solaris) есть cron — классический демон-планировщик задач, использующийся для запуска заданий в определенное время.

Для сервера Oracle задания запускаются специальными фоновыми процессами (Job queue processes, Jnnn), а для управления заданиями предназначен встроенный пакет DBMS_JOB. Число фоновых процессов Jnnn настраивается параметром экземпляра job_queue_processes. Если задания не запускаются в назначенное им время, то следует проверить значение этого параметра, выставить при необходимости для него ненулевое значение и перезапустить экземпляр Oracle.

Автоматический запуск заданий в Oracle может использоваться для решения задач администрирования баз данных, например, автоматического сбора статистики или анализа журналов аудита. Также в заданные моменты времени могут автоматически запускаться и прикладные задачи, такие, например, как загрузка данных, поступивших из внешних источников, формирование месячных счетов к оплате или расчет клиентской задолженности. Ресурсоемкие операции, такие как подготовка аналитических отчетов, следует запускать в то время, когда серверы баз данных загружены меньше — ночью или в выходные дни. Понятно, что такие задания также должны запускаться автоматически, без участия человека.

При использовании пакета DBMS_JOB можно поставить в очередь заданий новое задание, которое будет автоматически выполнено в указанные моменты времени. Заданием в Oracle является анонимный блок PL/SQL, в котором, как правило, вызываются хранимые программы.

Распространено мнение, что среди всех встроенных пакетов DBMS_JOB — наиболее часто используемый администраторами баз данных Oracle.

Таблица 8. Программы пакета DBMS_JOB.

Программа	Описание программы
SUBMIT (процедура)	отправляет новое задание в очередь заданий
CHANGE (процедура)	изменяет параметры задания
WHAT (процедура)	изменяет описание задания
NEXT_DATE (процедура)	изменяет следующее время выполнения задания

REMOVE (процедура)	удаляет задание из очереди
RUN (процедура)	указывает немедленно выполнить задание
INTERVAL (процедура)	изменяет интервал между запусками задания

Будем рассматривать в качестве задания анонимный блок PL/SQL, запускающий процедуру p1.

```
CREATE TABLE job_test (insert_date DATE);

CREATE PROCEDURE p1 AS
BEGIN
  INSERT INTO job_test VALUES(SYSDATE);
  COMMIT;
END;
```

Отправим задание в очередь со следующими параметрами:

```
SQL> DECLARE
  2    l_job_num INTEGER;
  3  BEGIN
  4    DBMS_JOB.submit(job =>l_job_num,
  5                    what=>'BEGIN p1; END;',
  6                    next_date=>sysdate,
  7                    interval=>'sysdate+10/24/60/60');
  8    COMMIT;
  9    DBMS_OUTPUT.PUT_LINE(l_job_num);
 10  END;
 11  /
23
PL/SQL procedure successfully completed.
```

В очередь будет помещено новое задание BEGIN p1; END; с немедленным выполнением после помещения в очередь и последующим выполнением каждые десять секунд, что задается выражением sysdate+10/24/60/60. На экран выводится уникальный номер задания, назначенный ему при постановке в очередь (23). Важно отметить, что после вызова процедуры DBMS_JOB.submit необходимо явно зафиксировать транзакцию, в противном случае задание в очередь поставлено не будет.

Посмотрим на результаты работы процедуры p1 и убедимся, что задание запускается каждые десять секунд:

```
SQL> SELECT TO_CHAR(insert_date,'DD.MM.YYYY HH24:MI:SS') AS insert_date
  2  FROM job_test ORDER BY insert_date;
```

```
INSERT_DATE
-------------------
25.01.2015 11:26:18
25.01.2015 11:26:28
25.01.2015 11:26:38
```

Данные о заданиях пользователя, которые в данный момент находятся в очереди, можно просмотреть в представлении словаря-справочника данных USER_JOBS:

```
SQL> SELECT job,log_user,last_sec,next_sec,broken,interval,what
  2  FROM USER_JOBS;
JOB LAST_SEC NEXT_SEC BROKEN INTERVAL                WHAT
--- -------- -------- ------ -------------------    --------------
23  11:26:51 11:27:01 N      sysdate+10/24/60/60    BEGIN p1; END;
```

В первом столбце отображаются номера заданий, назначаемые им при постановке в очередь. В следующих столбцах отображаются время последнего и следующего выполнения задания, интервал выполнения и анонимный блок PL/SQL для выполнения задания. Меткой BROKEN (заблокировано) помечаются те задания, при шестнадцати попытках выполнения которых произошли ошибки. Заблокированные задания перестают выполняться, поэтому администратору баз данных следует контролировать содержание представления DBA_JOBS, в котором отображаются сведения обо всех заданиях. Если в задании временно отпала необходимость, то его можно пометить как заблокированное специально, используя для этого процедуру DBMS_JOB.broken.

Удаляется из очереди задание следующим образом:

```
SQL> BEGIN
  2    DBMS_JOB.remove(23);
  3    COMMIT;
  4  END;
  5  /
PL/SQL procedure successfully completed.
```

Файловый ввод-вывод

Встроенный пакет UTL_FILE предоставляет программам PL/SQL возможность работать с файлами на сервере. С помощью этого пакета можно решать следующие задачи:

- формирование на сервере небольших текстовых отчетов;
- загрузка в базу данных строк небольших текстовых файлов;

- чтение log-файлов сервера для мониторинга ошибок;
- формирование log-файлов программ PL/SQL.

Для загрузки/выгрузки данных значительных (мегабайты, гигабайты) объемов следует использовать не пакет UTL_FILE, а использовать клиентские приложения — SQL*Loader для загрузки данных и SQL*Plus для их выгрузки. Если запускать эти утилиты на сервере баз данных, то их использование — самый быстрый способ загрузить и выгрузить данные из базы в файлы. В то же время при разработке серверной бизнес-логики встречаются ситуации, когда читать и писать в файлы хотелось бы именно в хранимых программах на PL/SQL без усложнений, вызванных обращениями к другим средствам. Чтобы для каждого поступившего платежа выгружать на сервере XML-файл размером меньше килобайта пакет UTL_FILE вполне подходит.

Настройка ограничений доступа к каталогам

Считается, что UTL_FILE потенциально может представлять большую угрозу безопасности баз данных Oracle. Этот пакет дает возможность доступа к файлам от имени такого пользователя операционной системы, который в ней имеет доступ ко всем файлам данных, журнала базы данных, управляющим файлам, файлам паролей и параметров. Получив неправомерный доступ к пакету UTL_FILE, при определенных навыках можно довольно быстро грамотно вывести базу данных из строя или реализовать угрозу нарушения конфиденциальности информации (например, прочитать файлы данных).

По этой причине перед использованием пакета UTL_FILE требуется настроить ограничения доступа к каталогам, с файлами из которых сможет осуществлять операции пакет. Настроить ограничения доступа к каталогам можно сделать двумя способами:

- использовать параметр базы данных utl_file_dir;
- создать специальные объекты баз данных — директории (DIRECTORY) и предоставить пользователям привилегии доступа к ним.

В параметре utl_file_dir задается список каталогов, с файлами из которых может осуществлять операции UTL_FILE. Для просмотра текущего значения параметра utl_file_dir можно использовать представление словаря данных V$PARAMETER или команду SHOW PARAMETER утилиты SQL*Plus:

```
SQL> SELECT value dir FROM V$PARAMETER
  2   WHERE name='utl_file_dir';

DIR
----------------------------------------
C:\Dir1

SQL> SHOW PARAMETER utl_file_dir

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------
utl_file_dir                         string      C:\Dir1
```

Значением параметра utl_file_dir может быть и символ *, что означает, что с помощью пакета UTL_FILE можно получить доступ к файлам в любых каталогах, к которым есть доступ у пользователя oracle операционной системы сервера. Выставление этого параметра таким способом обычно практикуется в тестовых инсталляциях и должно быть совершенно исключено в production по требованиям безопасности.

Директория (directory) — объект баз данных Oracle, являющийся псевдонимом каталога в файловой системе сервера. Директории создаются DDL-командами следующего вида:

```
SQL> CREATE OR REPLACE DIRECTORY dir1 AS 'C:\Dir1';
Directory created.
```

По сравнению с использованием параметра utl_file_dir, директории дают более гибкие возможности по управлению доступом к файлам. Так, после того как директория создана, администратор базы данных может предоставить конкретным пользователям привилегии только на чтение из нее:

```
SQL> GRANT READ ON DIRECTORY dir1 TO user1;
Grant succeeded.
```

Параметр utl_file_dir задает список каталогов сразу для всех пользователей, которые имеют привилегии на выполнение программ пакета UTL_FILE, причем настроить конкретные действия (чтение или запись) с файлами из этих каталогов с его помощью нельзя.

Использование UTL_FILE

В пакете UTL_FILE используется следующая последовательность действий с файлами — сначала файл открывается в заданном режиме,

затем производятся действия с его содержимым, по окончании которых файл закрывается. После открытия файла во все процедуры и функции пакета UTL_FILE передается его идентификатор, который представляет собой переменную-запись PL/SQL объявленного в пакете UTL_FILE типа.

Таблица 9. Программы пакета UTL_FILE.

Программа	Описание программы
FOPEN (функция)	открывает файл для чтения/записи
IS_OPEN (функция)	проверяет, открыт ли файл
FCLOSE (процедура)	закрывает открытый файл
FCLOSE_A (процедура)	закрывает все открытые файлы (a — all)
GET_LINE (процедура)	считывает строку из файла
PUT (процедура)	записывает в файл строку без символа конца строки
PUT_LINE (процедура)	записывает в файл строку с символом конца строки
PUTF (процедура)	записывает в файл форматированный текст
FFLUSH (процедура)	вызывает физическую запись буферизированных данных

Файл может быть открыт процедурой UTL_FILE.FOPEN в одном из трех режимов:

- для чтения (read) — содержимое файла не будет изменяться;
- для записи (write) — содержимое файла будет перезаписано;
- для добавления данных (append) — запись будет осуществляться в конец файла без перезаписи имевшегося содержимого.

В качестве примера работы с пакетом UTL_FILE приведем код процедуры table_copy, которая построчно сохраняет выборку SQL-запроса к таблице tab1 в файл. У table_copy имеется параметр p_mode, определяющий режим открытия файла.

```
SQL> CREATE OR REPLACE PROCEDURE table_copy(p_mode IN VARCHAR2) IS
  2    fid UTL_FILE.FILE_TYPE;
  3  BEGIN
  4    fid := UTL_FILE.FOPEN (location=> 'C:\Dir1',
                              filename => 'f-name.txt',
                              open_mode=> p_mode);
```

```
  5    FOR rec IN (SELECT at1,at2 FROM tab1) LOOP
  6      UTL_FILE.PUT_LINE (fid, rec.at1||' '||rec.at2);
  7    END LOOP;
  8    UTL_FILE.FCLOSE (fid);
  9  EXCEPTION
 10     WHEN UTL_FILE.INVALID_PATH
 11      THEN DBMS_OUTPUT.PUT_LINE('Неверный каталог');
 12     WHEN UTL_FILE.INVALID_MODE
 13      THEN DBMS_OUTPUT.PUT_LINE('Неверный режим работы с файлом');
 14     WHEN UTL_FILE.INVALID_FILEHANDLE
 15      THEN DBMS_OUTPUT.PUT_LINE('Ошибочный дескриптор файла');
 16     WHEN UTL_FILE.READ_ERROR
 17      THEN DBMS_OUTPUT.PUT_LINE('Ошибка при чтении файла');
 18     WHEN UTL_FILE.WRITE_ERROR
 19      THEN DBMS_OUTPUT.PUT_LINE('Ошибка при записи в файл');
 20     WHEN UTL_FILE.INTERNAL_ERROR
 21      THEN DBMS_OUTPUT.PUT_LINE('Произошла внутренняя ошибка');
 22     WHEN OTHERS
 25      THEN DBMS_OUTPUT.PUT_LINE(SQLERRM);
 26  END;
 27  /
Procedure created.

SQL> BEGIN
  2    table_copy('A');
  3  END;
  4  /
PL/SQL procedure successfully completed.

SQL> SELECT * FROM tab1;
       AT1 A
---------- -
         1 A
         2 B
         3 C

-- команда HOST утилиты SQL*Plus позволяет выполнять команды ОС
-- выполняем прямо из SQL*Plus команду type ОС Windows
SQL> HOST type C:\Dir1\f-name.txt
1 A
2 B
3 C
```

В ходе выполнения процедуры `table_copy` **с параметром** `'A'` **(append) в конец файла** `fname.txt`**, находящийся в каталоге** `C:\Dir1`**, будут записаны все строки таблицы** `tab1`**. Обратите внимание — если вызвать процедуру** `table_copy` **с параметром** `'W'` **(write), то существующее содержимое файла будет перезаписано на содержимое таблицы.**

В старших версиях сервера Oracle с помощью пакета UTL_FILE можно копировать и удалять файлы. Также для работы с файлами на сервере можно использовать хранимые программы на Java.

Работа с большими объектами

В Oracle имеются специальные типы данных для хранения больших объектов (Large Objects, LOB), размеры которых могут измеряться в террабайтах:

- BLOB — тип для представления бинарных данных (значение содержит локатор на большой бинарный объект, хранящийся в базе данных);
- CLOB — тип для представления символьных данных (значение содержит локатор на большой символьный объект, хранящийся в базе данных);
- BFILE — тип данных для описания файлов (значение содержит указатель на файл, который находится вне базы данных Oracle).

Локатором называется хранящийся в базе данных указатель на данные большого объекта. Значение типа BLOB или CLOB может характеризоваться одним из трех состояний:

- содержит NULL (не содержит локатор);
- содержит локатор, указывающий данные большого объекта;
- содержит локатор, не указывающий ни на какие данные.

Про последнее состояние говорят, что это «пустой» (empty) LOB-объект. «Пустые» LOB-объекты инициализируются встроенными функциями EMPTY_BLOB() и EMPTY_CLOB(). Для определения текущего состояния значения из трех возможных используется следующая логика:

```
IF some_clob IS NULL THEN
  -- нет ни данных, ни локатора
ELSIF DBMS_LOB.GETLENGTH(some_CLOB)=0 THEN
  -- пустой (empty) LOB-объект
ELSE
  -- данные в LOB-объекте есть
END IF
```

Значения типа BFILE используются только для чтения из файлов. Удаление в строке таблицы значения типа BFILE или его копирование никак не влияют на сам файл в каталоге операционной системы,

с ним ничего не происходит. Все эти операции выполняются только над указателями на файлы.

Для работы с данными типа LOB нужно сначала извлечь локатор, а затем с помощью процедур и функций встроенного пакета DBMS_LOB прочитать или записать собственно данные.

Таблица 10. Программы пакета DBMS_LOB.

Программа	Описание программы
APPEND (процедура)	записывает данные в конец LOB-объекта
WRITE (процедура)	записывает данные в LOB-объект по смещению
COMPARE (функция)	сравнивает два LOB-объекта одного типа
GETLENGTH (функция)	возвращает длину LOB-объекта
INSTR (функция)	возвращает позицию вхождения строки в объект
READ (процедура)	считывает часть LOB-объекта
SUBSTR (функция)	возвращает часть LOB-объекта по смещению
FILECLOSE (процедура)	закрывает файл по указателю-значению BFILE
FILEEXISTS (функция)	проверяет наличие файла по указателю
FILEOPEN (процедура)	открывает файл для значения BFILE
COPY (процедура)	копирует LOB-объекты
ERASE (процедура)	удаляет LOB-объект полностью или частично

Работа с файлами с помощью пакета DBMS_LOB

В качестве примера использования пакета DBMS_LOB приведем процедуру f_compare, которая сравнивает файлы в каталоге dir1. Имена файлов передаются как параметры:

```
SQL> CREATE DIRECTORY dir1 AS 'C:\WORK';
Directory created.

SQL> CREATE OR REPLACE PROCEDURE f_compare
  2  (fname1 IN VARCHAR2, fname2 IN VARCHAR2) IS
  3    file_1  BFILE;
  4    file_2  BFILE;
  5    result  INTEGER;
  6  BEGIN
  7    file_1 := BFILENAME('DIR1',fname1);
  8    file_2 := BFILENAME('DIR1',fname2);
  9    DBMS_LOB.FILEOPEN(file_1);
 10    DBMS_LOB.FILEOPEN(file_2);
```

```
 11    result := DBMS_LOB.COMPARE(file_1,file_2,
 12                               DBMS_LOB.LOBMAXSIZE,1,1);
 13    IF (result != 0) THEN
 14      DBMS_OUTPUT.PUT_LINE('Файлы различные');
 15    ELSE
 16      DBMS_OUTPUT.PUT_LINE('Файлы одинаковые');
 17    END IF;
 18    DBMS_LOB.FILECLOSE(file_1);
 19    DBMS_LOB.FILECLOSE(file_2);
 20  END;
 21  /
Procedure created.

SQL> BEGIN
  2    f_compare('fname.txt','fname.txt');
  3  END;
  4  /
Файлы одинаковые

SQL> BEGIN
  2    f_compare('fname.txt','fname2.txt');
  3  END;
  4  /
Файлы различные

SQL> BEGIN
  2    f_compare('fname.txt','fname3.txt');
  3  END;
  4  /
BEGIN
*
ERROR at line 1:
ORA-22288: file or LOB operation FILEOPEN failed
The system cannot find the path specified
ORA-06512: at "SYS.DBMS_LOB", line 475
ORA-06512: at "SYSTEM.F_COMPARE", line 9
ORA-06512: at line 2
```

При последнем вызове процедуры `f_compare` **не удалось открыть указанный файл. Обратите внимание, ошибка произошла при попытке открыть файл, установка указателя** `BFILE` **произошла нормально.**

Для загрузки файлов в базу данных как LOB-объектов предназначена пакетная процедура `DBMS_LOB.LOADFROMFILE`, **которой в качестве параметров передается переменная типа** `BFILE`**, связанная с загружаемым файлом, количество байт, считываемое из файла, и указатель на объект-приемник.**

```
SQL> CREATE TABLE tab1 (at1 NUMBER, at2 BLOB, at3 BFILE);
Table created.
```

```
SQL> INSERT INTO tab1 VALUES (2,EMPTY_BLOB(),NULL);
1 row created.

SQL> DECLARE
  2    l_BLOB BLOB;
  3    file_1 BFILE;
  4  BEGIN
  5    SELECT at2 INTO l_BLOB FROM tab1
  6    WHERE at1=2 FOR UPDATE;
  7    file_1 := BFILENAME('DIR1','fname.txt');
  8    DBMS_LOB.FILEOPEN(file_1);
  10   DBMS_LOB.LOADFROMFILE(l_BLOB,file_1,
  11                         DBMS_LOB.GETLENGTH(file_1));
  12   COMMIT;
  13  END;
  14  /
PL/SQL procedure successfully completed.
```

В данном случае сначала строка таблицы с пустым LOB-объектом блокируется с помощью команды SELECT FOR UPDATE**, а затем пакетная процедура** DBMS_LOB.LOADFROMFILE **осуществляет в него загрузку из файла.**

Семантика SQL для LOB-объектов

Начиная с версии Oracle 9i, реализована поддержка семантики SQL для LOB-объектов. Это означает, что с BLOB **и** CLOB **могут работать обычные встроенные функции как со значениями типов** VARCHAR2 **и** CHAR **(используются перегруженные версии встроенных функций):**

```
SQL> CREATE TABLE clob_table (at1 CLOB);
Table created.

SQL> INSERT INTO clob_table VALUES ('I say :');
1 row created.

SQL> UPDATE clob_table SET at1 = 'Hello, world'||rpad(at1, 1000000, '!');
1 row updated.

SQL> SELECT LENGTH (at1) AS len, TO_CHAR (SUBSTR (at1, 1, 12)) AS words
  2  FROM clob_table;
       LEN WORDS
---------- ------------
   1000012 Hello, world
```

Столбец at1 **типа** CLOB **при выполнении предложений** UPDATE **и** SELECT **передавался как параметр встроенным функциям определения дли-**

ны строки LENGTH, выделения подстроки в строке SUBSTR и дополнения строки до заданной длинны RPAD.

Динамический SQL

Предложения SQL, которые не изменяются с момента компиляции программы PL/SQL, называются статическими. Статические предложения SQL формируются компилятором PL/SQL по объявлениям явных курсоров, по командам SELECT INTO и остальным DML-командам языка PL/SQL. После формирования они сохраняются в байт-коде хранимых программ PL/SQL и больше не изменяются.

Термином «динамический SQL» (dynamic SQL) называются предложения SQL, которые динамически формируются как символьные строки непосредственно во время выполнения программ PL/SQL. Эти предложения SQL в байт-коде отсутствуют, поэтому для из выполнения используются специальные механизмы PL/SQL, рассматриваемые далее.

Динамический SQL в PL/SQL в основном применяется для решения следующих задач:

- выполнение DDL-команд (CREATE, ALTER, DROP);
- поддержка нерегламентированных SQL-запросов (SQL ad hoc queries).

«Создать табличку» в программе PL/SQL нельзя:

```
SQL> BEGIN
  2    CREATE TABLE tab1(at1 INTEGER);
  3  END;
  4  /
  CREATE TABLE tab1(at1 INTEGER);
  *
ERROR at line 2:
ORA-06550: line 2, column 3:
PLS-00103: Encountered the symbol "CREATE" when expecting
one of the following: (begin case declare exit for goto if loop pragma …
```

По префиксу PLS видно, что ошибку выдал компилятор PL/SQL, а из текста сообщения следует, что она произошла на этапе синтаксического анализа кода программы. Даже в грамматике языка PL/SQL не предусмотрено наличие в коде PL/SQL команд, похожих на DDL-команды CREATE.

«…, но если очень хочется, то можно» — для это следует использовать динамический SQL, передавая DDL-команду как символьную строку:

```
SQL> BEGIN
  2    EXECUTE IMMEDIATE 'CREATE TABLE tab1(at1 INTEGER)';
  3  END;
  4  /
PL/SQL procedure successfully completed.

SQL> SELECT * FROM tab1;
no rows selected
```

Выборка из tab1 проходит без ошибок, значит, таблица существует.

Нерегламентированные запросы SQL

Нерегламентированным является SQL-запрос, у которого до этапа выполнения могут быть не определены следующие три составляющие:

- текст SQL-запроса, включая список таблиц во фразе FROM, критерий отбора данных во фразе WHERE, фразы группировки и сортировки;
- перечень возвращаемых столбцов;
- список параметров.

Примером выполнения нерегламентированных SQL-запросов может быть подбор моделей телефонов по параметрам, похожий на соответствующий сервис на «Яндекс.Маркете».

Пусть таблица моделей телефонов имеет следующий вид[1]:

```
CREATE TABLE phone_models (model    VARCHAR2(100),
                           LTE      INTEGER,
                           dual_sim INTEGER,
                           price    INTEGER,
                           color    VARCHAR2(100));
```

[1] Для примера в таблице хватит названия модели и четырех характеристик. А так вообще по состоянию на 01.02.2016 г. на «Яндекс.Маркете» модели мобильных телефонов можно было подбирать по 153 характеристикам.

```
INSERT INTO phone_models VALUES('Xiaomi Redmi Note 2',1,1,12500,'black');
INSERT INTO phone_models VALUES('Meizu M2 mini',1,1,11400,'white');
...
```

Один человек может подбирать себе телефон по двум параметрам «вид — смартфон, цена — в пределах 10 000 – 15 000 рублей», другой человек может подбирать модель не по двум, а по трем параметрам «LTE — да, две SIM-карты – да, цвет — черный». Приложению потребуется сформировать и выполнить в базе данных два разных SQL-запроса. Для первого поиска это будет SQL-запрос с тремя связываемыми переменными:

```
SELECT * FROM phone_models
WHERE type=:p_1
      AND price BETWEEN :p2 AND :p3
```

со значениями переменных `:p1='smartphone', :p2=10000, :p3=15000.`

Для второго поиска это будет SQL-запрос тоже с тремя связываемыми переменными, но для других ограничений:

```
SELECT * FROM phone_models
WHERE LTE=:p_1
      AND dual_sim=:p2 AND color=:p3
```

со значениями переменных `:p1=1, :p2=1, :p3='black'`

Механизмы выполнения динамического SQL в PL/SQL

Для выполнения динамического SQL в PL/SQL есть два механизма:

- встроенный динамический SQL (Native Dynamic SQL, NDS);
- встроенный пакет `DBMS_SQL`.

Динамический SQL в Oracle принято делить на четыре категории.

Таблица 11. Категории динамического SQL в Oracle.

Категория	Описание категории
Категория 1	DDL-команды и предложения `UPDATE`, `INSERT` и `DELETE` без параметров
Категория 2	DDL-команды и предложения `UPDATE`, `INSERT` и `DELETE` с фиксированным количеством параметров
Категория 3	предложения `SELECT` с фиксированным количеством столбцов и параметров

Категория 4	DML-предложения, в которых количество выбранных столбцов (для запросов) или количество параметров (для всех предложений) неизвестно до стадии выполнения

С помощью встроенного пакета DBMS_SQL можно выполнить динамический SQL всех четырех категорий, с помощью NDS — первых трех категорий, на которые приходится, по некоторым оценкам, до 90% всего динамического SQL.

Встроенный динамический SQL

Главным достоинством NDS является его простота. Для выполнения динамического SQL в пакете DBMS_SQL в общем случае требуется 8 этапов, при этом код PL/SQL выглядит довольно громоздко и далее будет возможность в этом убедиться. С NDS обходятся вызовом одной команды EXECUTE IMMEDIATE («выполнить немедленно»), которая имеет следующий синтаксис:

```
EXECUTE IMMEDIATE предложение SQL
[ [ BULK COLLECT] INTO {переменная[, переменная]... | запись PL/SQL}]
[USING аргумент[,аргумент]...];
```

Сразу после ключевых слов EXECUTE IMMEDIATE в одинарных кавычках указывается текст предложения SQL, также в этом месте можно указать символьную переменную с текстом предложения SQL, причем эта переменная может иметь тип данных как VARCHAR2, так и CLOB.

Конструкция INTO со списком переменных предназначена для получения значений столбцов результирующей выборки и используется в том случае, если выполняется предложение SELECT. Число переменных и число столбцов должно совпадать. Переменные в конструкции INTO должны быть скалярных типов данных, соответствующих типам столбцов, или одной записью PL/SQL.

Конструкция USING со списком переменных и констант используется для передачи значений, которые должны быть связаны с имеющимися в тексте предложения SQL связываемыми переменными. Связывание значений в NDS осуществляется по позициям связываемых переменных. Количество передаваемых значений, естественно, должно совпадать с количеством связываемых переменных.

Для SQL-запросов команда EXECUTE IMMEDIATE фактически является аналогом команды SELECT INTO с таким же ограничением на результирующую выборку: запросом должна отбираться ровно одна строка, в противном случае инициируются предопределенные исключения.

Рассмотрим пример использования команды EXECUTE IMMEDIATE:

```
CREATE TABLE tab1 (at1 INT, at2 VARCHAR2(1));
INSERT INTO tab1 VALUES(1,'A');
INSERT INTO tab1 VALUES(2,'B');

SQL> DECLARE
  2    l_tab1     tab1%ROWTYPE;
  3    l_sql_text VARCHAR2(100) := 'SELECT * FROM tab1 WHERE at1>=:p_at1';
  4  BEGIN
  5
  6    EXECUTE IMMEDIATE l_sql_text INTO l_tab1_rec USING 2;
  7    DBMS_OUTPUT.PUT_LINE('Отобранная строка: '||l_tab1.at1||l_tab1.at2);
  8
  9    BEGIN
 10      EXECUTE IMMEDIATE l_sql_text INTO l_tab1 USING 1;
 11    EXCEPTION
 12      WHEN OTHERS THEN
 13        DBMS_OUTPUT.PUT_LINE(SQLERRM);
 14    END;
 15
 16    BEGIN
 17      EXECUTE IMMEDIATE l_sql_text INTO l_tab1 USING 3;
 18    EXCEPTION
 19      WHEN OTHERS THEN
 20        DBMS_OUTPUT.PUT_LINE(SQLERRM);
 21    END;
 22
 23  END;
 24  /
Отобранная строка: 2B
ORA-01422: exact fetch returns more than requested number of rows
ORA-01403: no data found
PL/SQL procedure successfully completed.
```

Рассмотрим еще два примера использования NDS.

Пусть база данных спроектирована таким образом, что у каждой таблицы столбец первичного ключа называется id и имеет тип INTEGER.

Напишем процедуру, которая удаляет строку в любой таблице по значению ее первичного ключа. Параметрами процедуры будут имя таблицы и значение столбца id удаляемой строки.

```
CREATE TABLE tab1(id INTEGER PRIMARY KEY,at1 CHAR(1));
INSERT INTO tab1 VALUES(1,'a');
INSERT INTO tab1 VALUES(2,'b');

CREATE TABLE tab2(id INTEGER PRIMARY KEY,at1 CHAR(1));
INSERT INTO tab2 VALUES(20,'x');
INSERT INTO tab2 VALUES(30,'y');

SQL> CREATE OR REPLACE PROCEDURE delete_by_id (p_table_name IN VARCHAR2,
  2                                            p_id IN INTEGER) IS
  3  BEGIN
  4    EXECUTE IMMEDIATE 'DELETE FROM '||p_table_name||' WHERE id=:p_id'
  5                      USING p_id;
  6    DBMS_OUTPUT.PUT_LINE('In table '||p_table_name||' '
  7                            ||SQL%ROWCOUNT||' rows deleted');
  8  END;
  9  /

Procedure created.

SQL> set serveroutput on
SQL> EXECUTE delete_by_id('tab1',1);
In table tab1 1 rows deleted
PL/SQL procedure successfully completed.

SQL> EXECUTE delete_by_id('tab1',-1);
In table tab1 0 rows deleted
PL/SQL procedure successfully completed.

SQL> EXECUTE delete_by_id('tab2',20);
In table tab2 1 rows deleted
PL/SQL procedure successfully completed.
```

Для NDS в PL/SQL поддерживаются средства массовой обработки данных (bulk processing). Конструкция `BULK COLLECT` **указывается в том случае, когда известно, что SQL-запрос может иметь в результирующей выборке не одну, а несколько строк. Тогда переменной, в которую помещается результирующая выборка, должна быть коллекция, то есть и здесь прослеживается аналогия с командой** `SELECT INTO` **для статических предложений SQL. Также команда** `EXECUTE IMMEDIATE` **может использоваться совместно с рассматриваемой ранее командой** `FORALL`**.**

Вернем содержимое таблиц `tab1`**,** `tab2` **в исходное состояние и создадим теперь процедуру** `print_id_list` **со считыванием в коллекцию всех строк результирующей выборки с помощью конструкции** `BULK COLLECT`**.**

```
SQL> CREATE OR REPLACE PROCEDURE print_id_list(p_table_name IN VARCHAR2,
  2                              p_id IN INTEGER) IS
  3    TYPE t_table IS TABLE OF INTEGER;
  4    l_table t_table;
  5  BEGIN
  6    EXECUTE IMMEDIATE 'SELECT id FROM '||p_table_name||' WHERE id>:p_id'
  7            BULK COLLECT INTO l_table
  8            USING p_id;
  9    FOR i IN 1..l_table.COUNT LOOP
 10      DBMS_OUTPUT.PUT_LINE(l_table(i));
 11    END LOOP;
 12  END;
 13  /
Procedure created.

SQL> EXECUTE print_id_list('tab1',0);
1
2
PL/SQL procedure successfully completed.

SQL> EXECUTE print_id_list('tab2',20);
30
PL/SQL procedure successfully completed.
```

Как видно, использование NDS позволяет писать очень компактный код.

Пакет DBMS_SQL

Использование встроенного пакета DBMS_SQL для выполнения динамического SQL предусматривает в общем случае последовательность из 8 этапов.

Таблица 12. Этапы выполнения динамического SQL с помощью DBMS_SQL.

Программа	Описание этапа
OPEN_CURSOR	открывается курсор DBMS_SQL
PARSE	производится синтаксический разбор предложения SQL в курсоре (DDL-команды сразу и выполняются на этом этапе)
BIND_VARIABLE	со всеми связываемыми переменными предложения SQL в курсоре связываются значения
DEFINE_COLUMN	для SQL-запросов указывается, значения каких столбцов выборки в какие переменные PL/SQL будут считываться

EXECUTE	для открытого курсора выполняется предложение SQL
FETCH_ROWS	для SQL-запросов считывается строка выборки (обычно считывание осуществляется в цикле по всей выборке)
COLUMN_VALUE	переменным PL/SQL присваиваются значения столбцов текущей считанной строки из курсора
CLOSE_CURSOR	закрывается курсор DBMS_SQL

Перепишем процедуру print_id_list с использованием вместо NDS встроенного пакета DBMS_SQL.

```
SQL> CREATE PROCEDURE  print_id_list_dbms_sql(p_table_name IN VARCHAR2,
  2                                           p_id IN INTEGER) IS
  3    c_cursor INTEGER;
  4    ignore   INTEGER;
  5    l_id     INTEGER;
  6  BEGIN
  7    c_cursor := DBMS_SQL.open_cursor;
  8    DBMS_SQL.parse(c_cursor,
  9                   'SELECT ID FROM '||p_table_name||' WHERE id>:p_id',
 10                  DBMS_SQL.NATIVE);
 11    DBMS_SQL.define_column(c_cursor, 1, l_id);
 12    DBMS_SQL.bind_variable(c_cursor, 'p_id', p_id);
 13    ignore := DBMS_SQL.execute(c_cursor);
 14    LOOP
 15      IF DBMS_SQL.fetch_rows(c_cursor)>0 THEN
 16        DBMS_SQL.column_value(c_cursor, 1, l_id);
 17        DBMS_OUTPUT.PUT_LINE(l_id);
 18      ELSE
 19        EXIT;
 20      END IF;
 21    END LOOP;
 22    DBMS_SQL.close_cursor(c_cursor);
 23  END;
 24  /

Procedure created.

SQL> EXECUTE print_id_list_dbms_sql('tab1',0);
1
2

PL/SQL procedure successfully completed.

SQL> EXECUTE print_id_list_dbms_sql('tab2',20);
30

PL/SQL procedure successfully completed.
```

Код новой версии процедуры print_id_list выглядит более громоздким. Этим и объясняется то, что пакет DBMS_SQL, как правило, используют только тогда, когда использовать NDS нельзя. В остальных случаях обходятся одной строчкой кода с командой EXECUTE IMMEDIATE.

Выполнение динамического SQL четвертой категории

Читатель, вероятно, уже заметил в синтаксисе команды EXECUTE IMMEDIATE ограничение, мешающее использовать встроенный динамический SQL во всех случаях — в EXECUTE IMMEDIATE после конструкций INTO и USING необходимо указывать жестко заданные перечни переменных и констант PL/SQL. Они фиксируются на этапе написания программы и изменяться не могут. Поэтому NDS не подходит для выполнения четвертой категории динамического SQL, когда до стадии выполнения неизвестно количество столбцов результирующей выборки или количество параметров.

Пакет DBMS_SQL позволяет выполнять динамический SQL четвертой категории, так как на стадии выполнения его процедуры и функции можно вызывать любое количество раз. То есть надо просто вызывать процедуру DBMS_SQL.DEFINE_COLUMN и функцию DBMS_SQL.COLUMN_VALUE по числу возвращаемых SQL-запросом столбцов, а процедуру DBMS_SQL.BIND_VARIABLE — по числу имен связываемых переменных.

Задание для самостоятельной разработки

Пусть список значений параметров, которые указывают пользователи на web-странице при подборе моделей телефонов, формируется frontend-приложением в виде символьной строки из пар «параметр=значение», разделенных символом «;». Для цены передается значение вида from/to с указанными пользователем границами диапазона. Название параметра соответствует названию столбца таблицы.

Примеры поисковых запросов:

```
1) LTE=1;dual_sim=1
2) price=10000/12000;color=black;LTE=1
```

Необходимо написать процедуру PL/SQL, которая печатает на экране список моделей телефонов, удовлетворяющих заданным

условиям. Понятно, что это будет процедура, динамически формирующая и выполняющая нерегламентированные запросы SQL.

Задача облегчается тем, что список столбцов динамического SQL-запроса фиксирован (считаем, что всегда необходимо выводить два столбца — наименование модели и цену).

Как писать процедуру поиска моделей телефонов, в целом понятно. Строку параметров необходимо разобрать на пары «параметр=значение» по разделителям, потом в цикле дополнить шаблон SQL-запроса ограничениями со связываемыми переменными через `AND` по числу выделенных из строки пар, открыть курсор `DBMS_SQL`, разобрать в нем сформированное предложение SQL, связать значения переменных и так далее.

Рекомендуется использовать объявленные с помощью атрибута `%ROWTYPE` записи PL/SQL и считывание из курсора всех строк с помощью конструкции `BULK COLLECT`.

После проверки работоспособности процедуры можно попробовать добавить в таблицу `phone_models` новый столбец, например, `manufacturer`. Если код PL/SQL написан правильно и в хорошем стиле, то для того, чтобы процедура смогла обрабатывать запросы с новым параметром вида

```
3) LTE=1;dual_sim=1,manufacturer=Samsung
```

изменять в коде ничего не придется.

Рекомендации по дальнейшему изучению

Нет сомнений, что программист обычно ставит перед собой цель написать хорошую программу. Для понимания, достигнута ли эта цель, необходимо определиться с тем, какая программа является хорошей.

Автор книги является сторонником следующего утверждения: «Хорошая программа — это программа, которая работает, соответствует предъявляемым к ней требованиям и которую легко сопровождать и развивать».

Часто сопровождение программы почти полностью состоит из выявления и исправления ошибок в ней. Этот процесс является относительно легким в том случае, если выполняется три условия:

- в программе есть развитые механизмы протоколирования (логирования) своих действий;
- в программе используется ясная и продуманная схема обработки исключений;
- программа имеет код, который хорошо структурирован, понятен, легко изменяется и расширяется.

Заниматься сопровождением и развитием программы, не ведущей лог (log) высокой степени детализации — все равно, что собирать пазл картинкой вниз. Это трудоемко и не всегда приводит к успеху.

Для современных языков программирования C++, Java, C# есть готовые библиотеки ведения логов — log4cpp, log4J, log4net. Для PL/SQL тоже есть библиотека логирования LOG4PLSQL[1], которая имеет открытый исходный код и построена по образцу log4J — распространенного механизма логирования для программ на Java. Для обработки исключений в PL/SQL также есть библиотеки с открытым исходным кодом, например, Quest Error Manager. Настоятельно рекомендуется освоить и использовать эти механизмы при программировании на PL/SQL.

---

[1] LOG4PLSQL is PL/SQL generic tools for log, trace and debug some message and variable in all PL/SQL code (log4plsql.sourceforge.net).

Для PL/SQL также существует несколько систем управления версиями исходного кода, учитывающих специфику языка. Существуют кодогенераторы и системы автоматизированного модульного тестирования программ на PL/SQL. Все это можно найти в интернете и использовать в своих проектах.

Что же касается приобретения умения написания кода PL/SQL, который хорошо структурирован, понятен, легко изменяется и расширяется, то рекомендуется прочитать книгу Стивена Фейерштейна «Oracle PL/SQL для профессионалов». В 2015 году на русском языке вышло уже шестое издание этой книги. С книгами по PL/SQL вообще довольно интересная ситуация — по языку SQL, по основам технологий Oracle, по администрированию баз данных десятками авторов написаны сотни книг, а вот по языку PL/SQL содержательные книги можно пересчитать по пальцам двух рук, и почти все они написаны Стивеном Фейрштейном.

После получения некоторой практики программирования на PL/SQL, для проверки знаний рекомендуется пройти онлайн-тестирование на сайте PL/SQL Challenge, основанном на концепции «активного изучения»[1]. Тем, кто уже сделал PL/SQL своим основным рабочим инструментом и по восемь часов в день программирует на этом языке, рекомендуется прочитать книгу Коннора МакДональда «Oracle PL/SQL для профессионалов. Практические решения». Это книга экспертного уровня по PL/SQL.

В определенный момент в ходе работы с PL/SQL станет понятно, каких возможностей этого языка не хватает для реализации серверной бизнес-логики. В первую очередь это относится к решению тех задач, которые предусматривают «выход» за пределы базы данных: подготовку и рассылку файлов различных форматов, обращение к web-ресурсам, оповещение о наступлении событий с данными посредством SMS-сообщений и сообщений мессенджеров и т. п.

В соответствии с правилами Томаса Кайта, задачи, не решаемые средствами SQL и PL/SQL, следует решать путем разработки хранимых программ на Java и внешних библиотек на C++. Мы рекомендуем читателю попробовать самому создать в базе данных Oracle

[1] https://plsqlchallenge.oracle.com

программы на Java и написать для них обертки (wrappers) на PL/SQL. Этой теме посвящены отдельные главы в книгах Фейерштейна, есть и статьи в интернете.

Вопросы для самопроверки

1) Перечислите достоинства и недостатки программ на PL/SQL.
2) В чем заключаются преимущества объявлений с привязкой?
3) Чем вложенные таблицы отличаются от массивов?
4) Почему OTHERS-обработчик указывается последним в списке обработчиков в разделе обработки исключений?
5) Почему для обработки пользовательских исключений неэффективно использование OTHERS-обработчика?
6) Для чего предназначена процедура RAISE_APPLICATION_ERROR?
7) Для чего используется привязка ошибок сервера к пользовательским исключениям?
8) Почему неявные курсоры так названы?
9) Какие требования предъявляются к SQL-запросу неявного курсора?
10) В какой момент инициализируются атрибуты явного курсора?
11) В какой момент атрибут %FOUND курсора примет значение FALSE?
12) За счет чего повышается производительность при использовании команды FORALL?
13) Когда сервер Oracle устанавливает неявные точки сохранения?
14) Для чего используются автономные транзакции?
15) Чем режим передачи значений параметров OUT отличается от IN OUT?
16) Для какого режима передачи значений параметров всегда используется передача по ссылке?
17) Какие переменные называются локальными переменными пакета?
18) В каком месте кода следует объявлять магические числа?
19) Для решения каких задач используются триггеры?
20) Для таблицы имеется триггер на выполнение предложения UPDATE. В каких случаях он не сработает ни разу?
21) С какой целью производится временное отключение триггеров?
22) В чем заключается причина ошибок доступа к мутирующей таблице?
23) Для чего предназначены встроенные пакеты?
24) Чем статический SQL отличается от динамического?
25) Какие виды предложений SQL не могут быть выполнены с помощью NDS (Native Dynamic SQL) и почему?

Рекомендуемая литература

1. Андон Ф., Резниченко В. Язык запросов SQL. — СПб.: Питер, 2006 г. — 416 с.
2. Гупта С. Oracle PL/SQL: руководство для разработчиков. — М.: Лори, 2014 г. — 464 с.
3. Зудилова Т. В., Иванов С. Е., Хоружников С. Э. SQL и PL/SQL для разработчиков СУБД Oracle. — СПб:НИУ ИТМО, 2012 г. — 74 с.
4. Кайт Т., Кун Д. Oracle для профессионалов. Архитектура и методики программирования. 3-е изд. — М.: Вильямс, 2015 г. — 960 с.
5. Карвин Б. Программирование баз данных SQL. Типичные ошибки и их устранение. — М.: Рид Груп, 2012 г. — 336 с.
6. Коннор МакДональд. Oracle PL/SQL для профессионалов. Практические решения. — СПб: ДиаСофт, 2005 г., — 560 с.
7. Лаврентьев В.С. Освоение SQL и PL/SQL Oracle. Лабораторные работы. — М.: Национальный исследовательский ядерный университет «МИФИ», 2009 г. — 105 с.
8. Мак-Локлин М. Oracle Database 11g. Программирование на языке PL/SQL. — М.: Лори, 2014 г. — 879 с.
9. Пржиялковский В.В. Введение в Oracle SQL. — М.: Бином, 2012 г. — 320 с.
10. Смирнов С.Н., Задворьев И.С. Работаем с Oracle: Учебное пособие / 2-е изд., испр. и доп. — М.: Гелиос АРВ, 2002 г.— 496 с.
11. Смирнов С.Н., Киселев А.В. Практикум по работе с базами данных. — М.: Гелиос АРВ, 2012 г. — 160 с.
12. Урман С., Хардман Р., МакЛафлин М. Oracle Database 10g. Программирование на языке PL/SQL. — М.: Лори, 2010 г. — 816 с.
13. Фейерштейн С., Нанда А. Oracle PL/SQL для администраторов баз данных. — СПб.: Символ-Плюс, 2008 г. — 496 с.
14. Фейерштейн С., Прибыл Б. Oracle PL/SQL для профессионалов. 6-е изд. — СПб.: Питер, 2015 г. — 1024 с.

Оглавление

Всего пронумеровано 188 стр.

Подписано к печати 05.10.2016 г.
Авт. л. 7,83. Усл. печ. л. 11,75.
Заказ 2052/16.
Себестоимость экземпляра 140 руб. 29 коп.