Основы языка PL/SQL: структура блока PL/SQL и область видимости. Процедуры и функции. Разветвления и циклы.

Каждый язык, будь то естественный или компьютерный, имеет определенный синтаксис, лексикон и набор символов. Чтобы общаться на этом языке, необходимо изучить правила его использования. Многие с опаской приступают к изучению новых компьютерных языков, но обычно они очень просты, и PL/SQL не является исключением. Трудности общения на компьютерных языках связаны не с самим языком, а с компилятором или компьютером, с которым мы «общаемся». Компиляторы не обладают творческим, гибким мышлением, а их лексикон крайне ограничен. Разве что соображают они очень, очень быстро... но только в рамках заданных правил.

Если приказать PL/SQL «подкинь-ка мне еще с полдюжины записей», едва ли вы получите требуемое. С точки зрения синтаксиса, для использования PL/SQL нужно расставлять все точки над «i». Поэтому в данной статье блога изложены основные правила языка, которые помогут вам общаться с компилятором PL/SQL.

Структура блока PL/SQL

В PL/SQL, как и в большинстве других процедурных языков, наименьшей единицей группировки кода является **блок**. Он представляет собой фрагмент программного кода, определяющий границы выполнения и области видимости для объявлений переменных и обработки исключений. PL/SQL позволяет создавать как *именованные*, так и *анонимные блоки* (то есть блоки, не имеющие имени), которые представляют собой **пакеты**, **процедуры**, **функции**, **триггеры** или **объектные типы**.

Блок PL/SQL может содержать до четырех разделов, однако только один из них является обязательным.

- **Заголовок**. Используется только в именованных блоках, определяет способ вызова именованного блока или программы. Не обязателен.
- **Раздел объявлений**. Содержит описания переменных, курсоров и вложенных блоков, на которые имеются ссылки в исполняемом разделе и разделе исключений. Не обязателен.
- **Исполняемый раздел**. Команды, выполняемые ядром PL/SQL во время работы приложения. Обязателен.
- **Раздел исключений**. Обрабатывает исключения (предупреждения и ошибки). Не обязателен.

Структура блока PL/SQL для процедуры показана на рис. 1.

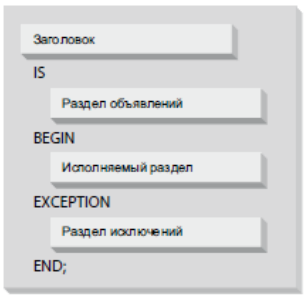


Рис. 1. Структура блока PL/SQL

На рис. 2 показана процедура, содержащая все четыре раздела. Этот конкретный блок начинается с ключевого слова *PROCEDURE* и, как и все блоки, завершается ключевым словом *END*.

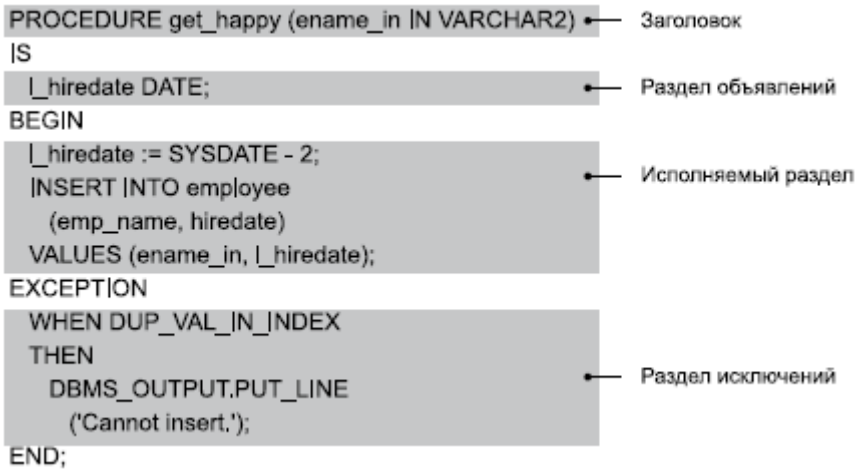


Рис. 2. Процедура, содержащая все четыре раздела

Анонимные блоки PL/SQL

Когда кто-то хочет остаться неизвестным, он не называет своего имени. То же можно сказать и об анонимном блоке PL/SQL, показанном на рис. 3: в нем вообще нет раздела заголовка, блок начинается ключевым словом DECLARE (или BEGIN). Анонимный блок не может быть вызван из другого блока, поскольку он не имеет идентификатора, по которому к нему можно было бы обратиться. Таким образом, анонимный блок представляет собой контейнер для хранения команд PL/SQL — обычно с вызовами процедур и функций. Поскольку анонимные блоки могут содержать собственные разделы объявлений и исключений, разработчики часто используют вложение анонимных блоков для ограничения области видимости идентификаторов и организации обработки исключений в более крупных программах.

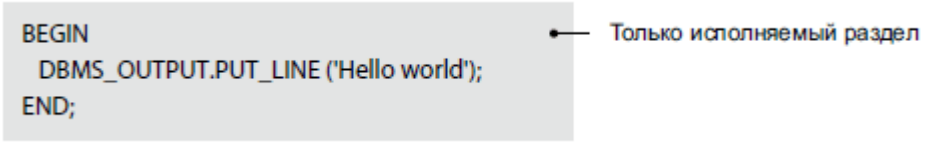


Рис. 3. Анонимный блок без разделов объявлений и исключений

Общий синтаксис анонимного блока PL/SQL:

[DECLARE ... объявления ...]

BEGIN ... одна или несколько исполняемых команд ...

[EXCEPTION

... команды обработки исключений ...]

END;

Квадратными скобками обозначаются необязательные составляющие синтаксиса. Анонимный блок обязательно содержит ключевые слова *BEGIN* и *END*, и между ними должна быть как минимум одна исполняемая команда. Несколько примеров:

- Простейший анонимный блок:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(SYSDATE);
END;
```

- Анонимный блок с добавлением раздела объявлений:

```
DECLARE
  l_right_now VARCHAR2(9);
BEGIN
  l_right_now := SYSDATE;
  DBMS_OUTPUT.PUT_LINE (l_right_now);
END;
```

- Тот же блок, но с разделом исключений:

```
DECLARE
  l_right_now VARCHAR2(9);
BEGIN
  l_right_now := SYSDATE;
  DBMS_OUTPUT.PUT_LINE (l_right_now);
EXCEPTION
  WHEN VALUE_ERROR
  THEN
  DBMS_OUTPUT.PUT_LINE('l_right_now не хватает места '
      || ' для стандартного формата даты');
END;
```

Анонимный блок выполняет серию команд, а затем завершает свою работу, то есть по сути является аналогом процедуры. Фактически каждый анонимный блок является анонимной процедурой. Они используются в различных ситуациях, в которых код PL/SQL выполняется либо непосредственно, либо как часть другой программы. Типичные примеры:

- **Триггеры баз данных**. Триггеры выполняют анонимные блоки при наступлении определенных событий.

- **Специализированные команды или сценарии**. В SQL*Plus и других аналогичных средах анонимные блоки активизируются из кода, введенного вручную, или из сценариев, называемых хранимыми программами. Кроме того, команда SQL*Plus *EXECUTE* преобразует свой аргумент в анонимный блок, заключая его между ключевыми словами *BEGIN* и *END*.
- **Откомпилированная программа 3GL**. В Pro*C и OCI анонимные блоки используются для внедрения вызовов хранимых программ во внешний код.

Во всех случаях контекст — и возможно, механизм присваивания имени — предоставляется внешним объектом (будь то триггер, программа командной строки или откомпилированная программа).

Именованные блоки PL/SQL

Хотя анонимные блоки PL/SQL применяются во многих приложениях Oracle, вероятно, большая часть написанного вами кода будет оформлена в виде именованных блоков. Ранее вы уже видели несколько примеров хранимых процедур (см. рис. 1) и знаете, что их главной особенностью является наличие заголовка. Заголовок процедуры выглядит так:

PROCEDURE [схема.]имя [(параметр [, параметр ...])]

[AUTHID {DEFINER | CURRENT_USER}]

Заголовок функции в целом очень похож на него, но дополнительно содержит ключевое слово *RETURN*:

FUNCTION [схема.]имя [(параметр [, параметр ...])]

RETURN возвращаемый_тип

[AUTHID {DEFINER | CURRENT_USER}]

[DETERMINISTIC]

[PARALLEL ENABLE ...]

[PIPELINED [USING...] | AGGREGATE USING...]

Поскольку Oracle позволяет вызывать некоторые функции из SQL-команд, заголовок функции содержит больше необязательных компонентов, чем заголовок процедуры (в зависимости от функциональности и производительности исполнительной среды SQL).

Запуск хранимой процедуры в SqlDeveloper

Ниже приводится пример по созданию и запуску некоторой хранимой процедуры, выводящей в консоль традиционную строку:

```
Create or replace procedure HelloWorld is
    l_str_my varchar2(20);
begin
```

```
        l_str_my := ‘Hello world’;
        dbms_output.put_line(l_str_my);
end;


set serveroutput on;
execute hr.HelloWordl;
```

Вложенные блоки PL/SQL

PL/SQL, как и языки Ada и Pascal, относится к категории языков с блочной структурой, то есть блоки PL/SQL могут вкладываться в другие блоки. С другой стороны, язык С тоже поддерживает блоки, но стандартный С не является строго блочно-структурированным языком, потому что вложение подпрограмм в нем не допускается.

В следующем примере PL/SQL показана процедура, содержащая анонимный вложенный блок:

```
PROCEDURE calc_totals
IS
  year_total NUMBER;
BEGIN
  year_total := 0;
  /* Начало вложенного блока */
  DECLARE
    month_total NUMBER;
  BEGIN
    month_total := year_total / 12;
  END set_month_total;
  /* Конец вложенного блока */
END;
```

Ограничители */* и */* обозначают начало и конец комментариев. Анонимные блоки также могут вкладываться более чем на один уровень (рис. 4).

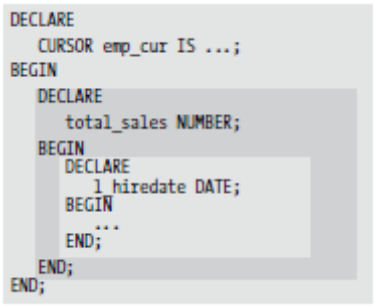


Рис. 4. Вложенные анонимные блоки

Главное преимущество вложенных блоков заключается в том, что они позволяют ограничивать области видимости и действия синтаксических элементов кода.

Область действия в PL/SQL

В любом языке программирования областью действия (scope) называется механизм определения «сущности», обозначаемой некоторым идентификатором. Если программа содержит более одного экземпляра идентификатора, то используемый экземпляр определяется языковыми правилами области действия. Управление областью видимости идентификаторов не только помогает контролировать поведение программы, но и уменьшает вероятность того, что программист по ошибке изменит значение не той переменной.

В PL/SQL переменные, исключения, модули и некоторые другие структуры являются локальными для блока, в котором они объявлены. Когда выполнение блока будет завершено, все эти структуры становятся недоступными. Например, в приведенной выше процедуре *calc_totals* можно обращаться к элементам внешнего блока (например, к переменной *year_total*), тогда как элементы, объявленные во внутреннем блоке, для внешнего блока недоступны.

У каждой переменной PL/SQL имеется некоторая область действия — участок программы (блок, подпрограмма или пакет), в котором можно ссылаться на эту переменную. Рассмотрим следующее определение пакета:

```
PACKAGE scope_demo
IS
  g_global NUMBER;
  PROCEDURE set_global (number_in IN NUMBER);
END scope_demo;

PACKAGE BODY scope_demo
IS
  PROCEDURE set_global (number_in IN NUMBER)
```

```
  IS
    l_salary NUMBER := 10000;
    l_count PLS_INTEGER;
  BEGIN

    <<local_block>>
    DECLARE
      l_inner NUMBER;
    BEGIN
      SELECT COUNT (*)
        INTO l_count
        FROM employees
      WHERE department_id = l_inner AND salary > l_salary;
    END local_block;

    g_global := number_in;
  END set_global;
END scope_demo;
```

Переменная *scope_demo.g_global* может использоваться в любом блоке любой схемы, обладающем привилегией *EXECUTE* для *scope_demo*.

Переменная *l_salary* может использоваться только в процедуре *set_global*.

Переменная *l_inner* может использоваться только в локальном или вложенном блоке; обратите внимание на использование метки *local_block* для присваивания имени вложенному блоку.

Ссылки на переменные и столбцы в предыдущем примере не уточнялись именами области действия. Далее приводится другая версия того же пакета, но на этот раз с уточнением ссылок (выделены полужирным шрифтом):

```
PACKAGE BODY scope_demo
IS
  PROCEDURE set_global (number_in IN NUMBER)
  IS
    l_salary NUMBER := 10000;
    l_count PLS_INTEGER;
```

```
  BEGIN


    <<local_block>>
    DECLARE
      l_inner PLS_INTEGER;
    BEGIN
      SELECT COUNT (*)
        INTO set_global.l_count
        FROM employees e
       WHERE e.department_id = local_block.l_inner
         AND e.salary > set_global.l_salary;
    END local_block;
    scope_demo.g_global := set_global.number_in;
  END set_global;
END scope_demo;
```

В новой версии каждая ссылка на столбец и переменную уточняется псевдонимом таблицы, именем пакета, именем процедуры или меткой вложенного блока.

Итак, теперь вы знаете об этой возможности — но зачем тратить время на уточнение имен? Для этого есть несколько очень веских причин:

- Удобство чтения кода.
- Предотвращение ошибок, возникающих при совпадении имен переменных с именами столбцов.
- Возможность использования детализированных зависимостей появилась в Oracle11g.

Давайте поближе рассмотрим первые две из этих причин.

Удобство чтения

Практически любая команда SQL, встроенная в программу PL/SQL, содержит ссылки на столбцы и переменные. В небольших, простых командах SQL различать эти ссылки относительно просто. Однако во многих приложениях используются очень длинные, исключительно сложные команды SQL с десятками и даже сотнями ссылок на столбцы и переменные.

Без уточнения ссылок вам будет намного сложнее различать переменные и столбцы. С уточнениями сразу видно, к чему относится та или иная ссылка.

«Один момент… Мы используем четко определенные схемы назначения имен, при помощи которых мы различаем строки и столбцы. Имена всех локальных переменных начинаются с „l_“, поэтому мы сразу видим, что идентификатор представляет локальную переменную».

Да, все правильно; все мы должны иметь (и соблюдать) правила назначения имен, чтобы имена идентификаторов содержали дополнительную информацию о них (что это — параметр, переменная? К какому типу данных она относится?).

Безусловно, правила назначения имен полезны, но они еще не гарантируют, что компилятор PL/SQL всегда будет интерпретировать ваши идентификаторы именно так, как вы задумали.

Предотвращение ошибок

Если не уточнять ссылки на переменные PL/SQL во встроенных командах SQL, код, который правильно работает сегодня, может внезапно утратить работоспособность в будущем. И разработчику будет очень трудно понять, что же пошло не так.

Вернемся к встроенной команде SQL без уточнения ссылок:

```
SELECT COUNT (*)
  INTO l_count
  FROM employees
WHERE department_id = l_inner AND salary > l_salary;
```

Сегодня идентификатор *l_salary* однозначно представляет переменную *l_salary*, объявленную в процедуре *set_global*. Я тестирую свою программу — она работает! Программа поставляется клиентам, все довольны.

А через два года пользователи просят своего администратора базы данных добавить в таблицу *employees* столбец, которому по случайности присваивается имя «*l_salary*». Видите проблему?

Во встроенной команде SQL база данных Oracle всегда начинает поиск соответствия для неуточненных идентификаторов со столбцов таблиц. Если найти столбец с указанным именем не удалось, Oracle переходит к поиску среди переменных PL/SQL в области действия. После добавления в таблицу *employee* столбца *l_salary* моей неуточненной ссылке *l_salary* в команде *SELECT* ставится в соответствие не переменная PL/SQL, а столбец таблицы. Результат?

Пакет *scope_demo* по-прежнему компилируется без ошибок, но секция *WHERE* запроса ведет себя не так, как ожидалось. База данных не использует значение переменной *l_salary*, а сравнивает значение столбца *salary* в строке таблицы employees со значением столбца *l_salary* той же строки. Отыскать подобную ошибку бывает очень непросто!

Не полагайтесь только на правила назначения имен для предотвращения «коллизий» между идентификаторами; уточняйте ссылки на все имена столбцов и переменных во

встроенных командах SQL. Это существенно снизит риск непредсказуемого поведения программ в будущем при возможных модификациях таблиц.

Видимость переменных PL/SQL

Важным свойством переменной, связанным с областью ее действия, является видимость. Данное свойство определяет, можно ли обращаться к переменной только по ее имени, или же к имени необходимо добавлять префикс.

«Видимые» идентификаторы

Начнем с тривиального случая:

```
DECLARE
  first_day DATE;
  last_day DATE;
BEGIN
  first_day := SYSDATE;
  last_day := ADD_MONTHS (first_day, 6);
END;
```

Обе переменные *first_day* и *last_day* объявляются в том же блоке, где они используются, поэтому при обращении к ним указаны только имена без уточняющих префиксов. Такие идентификаторы называются видимыми. В общем случае видимым идентификатором может быть:

- идентификатор, объявленный в текущем блоке;
- идентификатор, объявленный в блоке, который включает текущий блок;
- отдельный объект базы данных (таблица, представление и т. д.) или объект PL/SQL (процедура, функция), владельцем которого вы являетесь;
- отдельный объект базы данных или объект PL/SQL, на который у вас имеются соответствующие привилегии и который определяется видимым синонимом;
- индексная переменная цикла (видима и доступна только внутри цикла). PL/SQL также позволяет обращаться к существующим объектам, которые не находятся в пределах непосредственной видимости блока. О том, как это делается, рассказано в следующем разделе.

Уточненные идентификаторы

Типичным примером идентификаторов, невидимых в области кода, где они используются, являются идентификаторы, объявленные в спецификации пакета (имена переменных, типы данных, имена процедур и функций). Чтобы обратиться к такому объекту,

необходимо указать перед его именем префикс и точку (аналогичным образом имя столбца уточняется именем таблицы, в которой он содержится). Например:

- *price_util.compute_means* — программа с именем compute_means из пакета *price_util*.
- *math.pi* — константа с именем *pi*, объявленная и инициализированная в пакете *math*.

Дополнительное уточнение может определять владельца объекта. Например, выражение

scott.price_util.compute_means

обозначает процедуру *compute_means* пакета *price_util*, принадлежащего пользователю Oracle с учетной записью *scott*.

Уточнение идентификаторов именами модулей

PL/SQL предоставляет несколько способов уточнения идентификаторов для логического разрешения ссылок. Так, использование пакетов позволяет создавать переменные с глобальной областью действия. Допустим, имеется пакет *company_pkg* и в спецификации пакета объявлена переменная с именем *last_company_id*:

```
PACKAGE company_pkg
IS
  last_company_id NUMBER;
  ...
END company_pkg;
```

На переменную можно ссылаться за пределами пакета — необходимо лишь указать перед ее именем имя пакета:

```
IF new_company_id = company_pkg.last_company_id THEN
```

По умолчанию значение, присвоенное переменной пакетного уровня, продолжает действовать на протяжении текущего сеанса базы данных; оно не выходит из области действия вплоть до разрыва подключения.

Идентификатор также можно уточнить именем модуля, в котором он определен:

```
PROCEDURE calc_totals
IS
  salary NUMBER;
BEGIN
  ...
```

```
  DECLARE
    salary NUMBER;
  BEGIN
    salary := calc_totals.salary;
  END;
  ...
END;
```

В первом объявлении создается переменная *salary*, областью действия которой является вся процедура. Однако затем во вложенном блоке объявляется другой идентификатор с тем же именем. Поэтому ссылка на переменную *salary* во внутреннем блоке всегда сначала разрешается по объявлению в этом блоке, где переменная видима безо всяких уточнений. Чтобы во внутреннем блоке обратиться к переменной *salary*, объявленной на уровне процедуры, необходимо уточнить ее имя именем процедуры (*cal_totals*.*salary*).

Этот метод уточнения идентификаторов работает и в других контекстах. Что произойдет при выполнении следующей процедуры (*order_id* — первичный ключ таблицы *orders*):

```
PROCEDURE remove_order (order_id IN NUMBER)
IS
BEGIN
  DELETE orders WHERE order_id = order_id; -- Катастрофа!
END;
```

Этот фрагмент удалит из таблицы *orders* все записи независимо от переданного значения *order_id*. Дело в том, что механизм разрешения имен SQL сначала проверяет имена столбцов и только потом переходит к идентификаторам PL/SQL. Условие *WHERE* (*order_id = order_id*) всегда истинно, поэтому все данные пропадают.

Возможное решение проблемы выглядит так:

```
PROCEDURE remove_order (order_id IN NUMBER)
IS
BEGIN
  DELETE orders WHERE order_id = remove_order.order_id;
END;
```

В этом случае при разборе имя переменной будет интерпретировано правильно. (Решение работает даже при наличии в пакете функции с именем *remove_order*.*order_id*.)

В PL/SQL установлен целый ряд правил разрешения конфликтов имен, а этой проблеме уделяется серьезное внимание. И хотя знать эти правила полезно, лучше использовать уникальные идентификаторы, чтобы избежать подобных конфликтов. Старайтесь писать надежный код! Если же вы не хотите уточнять каждую переменную, чтобы обеспечить ее уникальность, вам придется тщательно проработать схему назначения имен для предотвращения подобных конфликтов.

Вложенные программы

Завершая тему вложения, области действия и видимости, стоит упомянуть о такой полезной возможности PL/SQL, как вложенные программы (nested programs). Вложенная программа представляет собой процедуру или функцию, которая полностью размещается в разделе объявлений внешнего блока. Вложенная программа может обращаться ко всем переменным и параметрам, объявленным ранее во внешнем блоке, как показывает следующий пример:

```
PROCEDURE calc_totals (fudge_factor_in IN NUMBER)
IS
  subtotal NUMBER := 0;
  /* Начало вложенного блока (в данном случае процедуры).
  | Обратите внимание: процедура полностью размещается
  | в разделе объявлений calc_totals.
  */
  PROCEDURE compute_running_total (increment_in IN PLS_INTEGER)
  IS
  BEGIN
    /* Переменная subtotal (см. выше) видима и находится в области действия */
    subtotal := subtotal + increment_in * fudge_factor_in;
  END;
  /* Конец вложенного блока */
BEGIN
  FOR month_idx IN 1..12
  LOOP
    compute_running_total (month_idx);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Годовой итог: ' || subtotal);
```

END;

Вложенные программы упрощают чтение и сопровождение кода, а также позволяют повторно использовать логику, задействованную в нескольких местах блока.

Ветвления.

Есть два типа управляющих команд ветвления в PL/SQL: условные команды и команды перехода. Команды первого типа, присутствующие почти во всех программах, управляют последовательностью выполнения программного кода в зависимости от заданных условий. В языке PL/SQL к этой категории относятся команды IF-THEN-ELSE и CASE. Также существуют так называемые *CASE-выражения*, которые иногда позволяют обойтись без команд IF и CASE. Значительно реже используются команды второго типа: GOTO (безусловный переход) и NULL (не выполняет никаких действий).

Команды IF

Команда IF реализует логику условного выполнения команд программы. С ее помощью можно реализовать конструкции следующего вида:

- Если оклад находится в пределах от $10 000 до $20 000, начислить премию в размере $1500.
- Если коллекция содержит более 100 элементов, удалить лишнее.

Команда IF существует в трех формах, представленных в следующей таблице.

Разновидность IF	**Характеристики**
IF THEN END IF;	Простейшая форма команды IF. Условие между IF и THEN определяет, должна ли выполняться группа команд, находящаяся между THEN и END IF. Если результат проверки условия равен FALSE или NULL, то код не выполняется
IF THEN ELSE END IF;	Реализация логики «или-или». В зависимости от условия между ключевыми словами IF и THEN выполняется либо код, находящийся между THEN и ELSE, либо код между ELSE и END IF. В любом случае выполняется только одна из двух групп исполняемых команд
IF THEN ELSIF ELSE END IF;	Последняя, и самая сложная, форма IF выбирает действие из набора взаимоисключающих условий и выполняет соответствующую группу исполняемых команд. Если вы пишете подобную конструкцию IF в версии Oracle9i Release 1 и выше, подумайте, не заменить ли ее командой выбора CASE

Комбинация IF-THEN

Общий синтаксис конструкции IF-THEN выглядит так:

```
IF условие
THEN
  ... последовательность исполняемых команд ...
END IF;
```

Здесь условие — это логическая переменная, константа или логическое выражение с результатом TRUE, FALSE или NULL. Исполняемые команды между ключевыми словами THEN и END IF выполняются, если результат проверки условия равен TRUE, и не выполняются — если он равен FALSE или NULL.

Трехзначная логика

Логические выражения могут возвращать три возможных результата. Когда все значения в логическом выражении известны, результат равен TRUE или FALSE. Например, истинность или ложность выражений вида

```
(2 < 3) AND (5 < 10)
```

сомнений не вызывает. Однако иногда оказывается, что некоторые значения в выражении неизвестны. Это может быть связано с тем, что соответствующие столбцы базы данных содержат NULL или остались пустыми. Каким должен быть результат выражений с NULL, например:

```
2 < NULL
```

Так как отсутствующее значение неизвестно, на этот вопрос можно дать только один ответ: «Неизвестно». В этом и заключается суть так называемой трехзначной логики — возможными результатами могут быть не только TRUE и FALSE, но и NULL.

Следующая условная команда IF сравнивает два числовых значения. Учтите, что если одно из них равно NULL, то и результат всего выражения равен NULL (если переменная salary равна NULL, то give_bonus не выполняется):

```
IF salary > 40000
THEN
  give_bonus (employee_id,500);
END IF;
```

У правила, согласно которому NULL в логическом выражении дает результат NULL, имеются исключения. Некоторые операторы и функции специально реализованы так, чтобы при работе с NULL они давали результаты TRUE и FALSE (но не NULL). Например, для проверки значения NULL можно воспользоваться конструкцией IS NULL:

```
IF salary > 40000 OR salary IS NULL
THEN
  give_bonus (employee_id,500);
END IF;
```

В этом примере условие salary IS NULL дает результат TRUE, если salary не содержит значения, и результат FALSE во всех остальных случаях.

Для обнаружения возможных значений NULL и их обработки удобно применять такие операторы, как IS NULL и IS NOT NULL, или функции COALESCE и NVL2. Для каждой переменной в каждом написанном вами логическом выражении подумайте, что произойдет, если эта переменная содержит NULL.

Ключевые слова IF, THEN и END IF не обязательно размещать в отдельных строках. В командах IF разрывы строк не важны, поэтому приведенный выше пример можно было бы записать так:

IF salary > 40000 THEN give_bonus (employee_id,500); END IF;

Размещение всей команды в одной строке отлично подходит для простых конструкций IF — таких, как в приведенном примере. Но любая хоть сколько-нибудь сложная команда гораздо лучше читается, когда каждое ключевое слово размещается в отдельной строке. Например, если записать следующий фрагмент в одну строку, в нем будет довольно трудно разобраться. В нем нелегко разобраться даже тогда, когда он записан в три строки:

IF salary > 40000 THEN INSERT INTO employee_bonus (eb_employee_id, eb_bonus_amt)

VALUES (employee_id, 500); UPDATE emp_employee SET emp_bonus_given=1 WHERE

emp_employee_id=employee_id; END IF;

И та же команда вполне нормально читается при разбиении на строки:

```
IF salary > 40000
THEN
  INSERT INTO employee_bonus
    (eb_employee_id, eb_bonus_amt)
  VALUES (employee_id, 500);


  UPDATE emp_employee
    SET emp_bonus_given=1
  WHERE emp_employee_id=employee_id;
END IF;
```

Вопрос удобочитаемости становится еще более важным при использовании ключевых слов ELSE и ELSIF, а также вложенных команд IF. Поэтому, чтобы сделать логику команд IF максимально наглядной, мы рекомендуем применять все возможности отступов и форматирования. И те программисты, которым придется сопровождать ваши программы, будут вам очень признательны.

Конструкция IF-THEN-ELSE

Конструкция IF-THEN-ELSE применяется при выборе одного из двух взаимоисключающих действий. Формат этой версии команды IF:

```
IF условие
THEN
  ... последовательность команд для результата TRUE ...
ELSE
  ... последовательность команд для результата FALSE/NULL ...
END IF;
```

Здесь условие — это логическая переменная, константа или логическое выражение. Если его значение равно TRUE, то выполняются команды, расположенные между ключевыми словами THEN и ELSE, а если FALSE или NULL — команды между ключевыми словами ELSE и END IF.

Важно помнить, что в конструкции IF-THEN-ELSE всегда выполняется одна из двух возможных последовательностей команд. После выполнения соответствующей последовательности управление передается команде, которая расположена сразу после ключевых слов END IF.

Следующая конструкция IF-THEN-ELSE расширяет пример IF-THEN, приведенный в предыдущем разделе:

```
IF salary <= 40000
THEN
  give_bonus (employee_id, 0);
ELSE
  give_bonus (employee_id, 500);
END IF;
```

в этом примере сотрудники с окладом более 40 000 получат премию в 500, а остальным премия не назначается. Или все же назначается? Что произойдет, если у сотрудника по какой-либо причине оклад окажется равным NULL? В этом случае будут выполнены команды, следующие за ключевым словом ELSE, и работник получит премию, положенную только высокооплачиваемому составу. Поскольку мы не можем быть уверены в том, что оклад ни при каких условиях не окажется равным NULL, нужно защититься от подобных проблем при помощи функции NVL:

```
IF NVL(salary,0) <= 40000
THEN
  give_bonus (employee_id, 0);
ELSE
  give_bonus (employee_id, 500);
```

```
END IF;
```

Функция NVL возвращает нуль, если переменная salary равна NULL. Это гарантирует, что работникам с окладом NULL будет начислена нулевая премия (не позавидуешь!).

Логические флаги

Логические переменные удобно использовать в качестве флагов, чтобы одно и то же логическое выражение не приходилось вычислять по нескольку раз. Помните, что результат такого выражения можно присвоить логической переменной. Например, вместо

```
IF :customer.order_total > max_allowable_order
THEN
  order_exceeds_balance := TRUE;
ELSE
  order_exceeds_balance := FALSE;
END IF;
```

можно воспользоваться следующим, гораздо более простым выражением (при условии, что ни одна из переменных не равна NULL):

```
order_exceeds_balance:= :customer.order_total > max_allowable_order;
```

Теперь если где-либо в программном коде потребуется проверить, не превышает ли сумма заказа (order_total) максимально допустимое значение (max_allowable_order), достаточно простой и понятной конструкции IF:

```
IF order_exceeds_balance
THEN
  ...
```

Если вам еще не приходилось работать с логическими переменными, возможно, на освоение этих приемов уйдет некоторое время. Но затраты окупятся сполна, поскольку в результате вы получите более простой и понятный код.

Конструкция IF-THEN-ELSIF

Данная форма команды IF удобна для реализации логики с несколькими альтернативными действиями в одной команде IF. Как правило, ELSIF используется с взаимоисключающими альтернативами (то есть при выполнении команды IF истинным может быть только одно из условий). Обобщенный синтаксис этой формы IF выглядит так:

```
IF условие-1
THEN
  команды-1
ELSIF condition-N
THEN
```

```
  команды-N
[ELSE
  команды_else]
END IF;
```

Некоторые программисты пытаются записывать ELSIF в виде ELSEIF или ELSE IF. Это очень распространенная синтаксическая ошибка.

Формально конструкция IF-THEN-ELSIF представляет собой один из способов реализации функций команды CASE в PL/SQL. Конечно, если вы используете Oracle9i, лучше воспользоваться командой CASE.

В каждой секции ELSIF (кроме секции ELSE) за условием должно следовать ключевое слово THEN. Секция ELSE в IF-ELSIF означает «если не выполняется ни одно из условий», то есть когда ни одно из условий не равно TRUE, выполняются команды, следующие за ELSE. Следует помнить, что секция ELSE не является обязательной — конструкция IFELSIF может состоять только из секций IF и ELSIF. Если ни одно из условий не равно TRUE, то никакие команды блока IF не выполняются.

Далее приводится полная реализация логики назначения премий, описанной в начале статьи, на базе конструкции IF-THEN-ELSEIF:

```
IF salary BETWEEN 10000 AND 20000
THEN
  give_bonus(employee_id, 1500);
ELSIF salary BETWEEN 20000 AND 40000
THEN
  give_bonus(employee_id, 1000);
ELSIF salary > 40000
THEN
  give_bonus(employee_id, 500);
ELSE
  give_bonus(employee_id, 0);
END IF;
```

Ловушки синтаксиса IF

Запомните несколько правил, касающихся применения команды IF:

- Каждая команда IF должна иметь парную конструкцию END IF. Все три разновидности данной команды обязательно должны явно закрываться ключевым словом END IF.
- Не забывайте разделять пробелами ключевые слова END и IF. Если вместо END IF ввести ENDIF, компилятор выдаст малопонятное сообщение об ошибке:

ORA-06550: line 14, column 4:

PLS-00103: Encountered the symbol ";" when expecting one of the following:

- Ключевое слово ELSIF должно содержать только одну букву «E». Если вместо ключевого слова ELSIF указать ELSEIF, компилятор не воспримет последнее как часть команды IF. Он интерпретирует его как имя переменной или процедуры.
- Точка с запятой ставится только после ключевых слов END IF. После ключевых слов THEN, ELSE и ELSIF точка с запятой не ставится. Они не являются отдельными исполняемыми командами и, в отличие от END IF, не могут завершать команду PL/SQL. Если вы все же поставите точку с запятой после этих ключевых слов, компилятор выдаст сообщение об ошибке.

Условия IF-ELSIF всегда обрабатываются от первого к последнему. Если оба условия равны TRUE, то выполняются команды первого условия. В контексте текущего примера для оклада $20000 будет начислена премия $1500, хотя оклад $20 000 также удовлетворяет условию премии $1000 (проверка BETWEEN включает границы). Если какое-либо условие истинно, остальные условия вообще не проверяются.

Команда CASE позволяет решить задачу начисления премии более элегантно, чем решение IF-THEN-ELSIF.

И хотя в команде IF-THEN-ELSIF разрешены перекрывающиеся условия, лучше избегать их там, где это возможно. В моем примере исходная спецификация немного неоднозначна в отношении граничных значений (таких, как 20 000). Если предположить, что работникам с низшими окладами должны начисляться более высокие премии (что на мой взгляд вполне разумно), я бы избавился от а BETWEEN и воспользовался логикой «меньше/больше» (см. далее). Также обратите внимание на отсутствие секции ELSE — я опустил ее просто для того, чтобы показать, что она не является обязательной:

```
IF salary >= 10000 AND salary <= 20000

THEN

  give_bonus(employee_id, 1500);

ELSIF salary > 20000 AND salary <= 40000

THEN

  give_bonus(employee_id, 1000);

ELSIF salary > 40000

THEN

  give_bonus(employee_id, 400);

END IF;
```

Принимая меры к предотвращению перекрывающихся условий в IF-THEN-ELSIF, я устраняю возможный (и даже вероятный) источник ошибок для программистов, которые будут работать с кодом после меня. Я также устраняю возможность введения случайных ошибок в результате переупорядочения секций ELSIF. Однако следует заметить, что если

значение salary равно NULL, никакой код выполнен не будет, потому что секции ELSE отсутствует.

Язык не требует, чтобы условия ELSIF были взаимоисключающими. Всегда учитывайте вероятность того, что значение может подходить по двум и более условиям, поэтому порядок условий ELSIF может быть важен.

Вложенные команды IF

Команды IF можно вкладывать друг в друга. В следующем примере представлены команды IF с несколькими уровнями вложенности:

```
IF условие1
THEN
  IF условие2
  THEN
    команды2
  ELSE
    IF условие3
    THEN
      команды3
    ELSIF условие4
    THEN
      команды4
    END IF;
  END IF;
END IF;
```

Сложную логику часто невозможно реализовать без вложенных команд IF, но их использование требует крайней осторожности. Вложенные команды IF, как и вложенные циклы, затрудняют чтение программы и ее отладку. И если вы собираетесь применить команды IF более чем с тремя уровнями вложения, подумайте, нельзя ли пересмотреть логику программы и реализовать требования более простым способом. Если такового не найдется, подумайте о создании одного или нескольких локальных модулей, скрывающих внутренние команды IF. **Главное преимущество** вложенных структур IF заключается в том, что они **позволяют отложить проверку внутренних условий**. Условие внутренней команды IF проверяется только в том случае, если значение выражения во внешнем условии равно TRUE. Таким образом, очевидной причиной вложения команд IF может быть проверка внутреннего условия только при истинности другого. Например, код начисления премий можно было бы записать так:

```
IF award_bonus(employee_id) THEN
```

```
  IF print_check (employee_id) THEN
    DBMS_OUTPUT.PUT_LINE('Check issued for ' || employee_id);
  END IF;
END IF;
```

Такая реализация вполне разумна, потому что для каждой начисленной премии должно выводиться сообщение, но если премия не начислялась, сообщение с нулевой суммой выводиться не должно.

Ускоренное вычисление

В PL/SQL используется ускоренное вычисление условий; иначе говоря, вычислять все выражения в условиях IF не обязательно. Например, при вычислении выражения в следующей конструкции IF PL/SQL прекращает обработку и немедленно выполняет ветвь ELSE, если первое условие равно FALSE или NULL:

```
IF условие1 AND условие2
THEN
  ...
ELSE
  ...
END IF;
```

PL/SQL прерывает вычисление, если условие_1 равно FALSE или NULL, потому что ветвь THEN выполняется только в случае истинности всего выражения, а для этого оба подвыражения должны быть равны TRUE. Как только обнаруживается, что хотя бы одно подвыражение отлично от TRUE, дальнейшие проверки излишни — ветвь THEN все равно выбрана не будет.

Изучая поведение ускоренного вычисления в PL/SQL, я обнаружил нечто интересное: его поведение зависит от контекста выражения. Возьмем следующую команду:

```
my_boolean := condition1 AND condition2
```

В отличие от случае с командой IF, если условие1 равно NULL, ускоренное вычисление применяться не будет. Почему? Потому что результат может быть равен NULL или FALSE в зависимости от условия2. Для команды IF оба значения NULL и FALSE ведут к ветви ELSE, поэтому ускоренное вычисление возможно. Но для присваивания должно быть известно конечное значение, и ускоренное вычисление в этом случае может (и будет) происходить только в том случае, если условие1 равно FALSE.

Аналогичным образом работает ускоренное вычисление в операциях OR: если первый операнд OR в конструкции IF равен TRUE, PL/SQL немедленно выполняет ветвь THEN:

```
IF условие1 OR условие2
```

```
THEN
  ...
ELSE
  ...
END IF;
```

Ускоренное вычисление может быть полезно в том случае, если одно из условий требует особенно серьезных затрат ресурсов процессора или памяти. Такие условия следует размещать в конце составного выражения:

```
IF простое_условие AND сложное_условие
THEN
  ...
END IF;
```

Сначала проверяется простое_условие, и если его результата достаточно для определения конечного результата операции AND (то есть если результат равен FALSE), более затратное условие не проверяется, а пропущенная проверка улучшает быстродействие приложения.

Но если работа вашей программы зависит от вычисления второго условия — например, из-за побочных эффектов от вызова хранимой функции, вызываемой в условии, — значит, вам придется пересмотреть структуру кода. Я считаю, что такая зависимость от побочных эффектов нежелательна.

Ускоренное вычисление легко имитируется при помощи вложения IF:

```
IF низкозатратное_условие
THEN
  IF высокозатратное_условие
  THEN
    ...
  END IF;
END IF;
```

Сложное_условие проверяется только в том случае, если простое_условие окажется истинным. Происходит то же, что при ускоренном вычислении, но зато при чтении программы можно с первого взгляда сказать, что же в ней происходит. Кроме того, сразу понятно, что в соответствии с намерениями программиста простое_условие должно проверяться первым.

Циклы

В этом блоге я расскажу Вам об **управляющих структуры PL/SQL**, называемых **циклами** и предназначенных для **многократного выполнения программного кода**. Также мы рассмотрим команду CONTINUE, появившуюся в Oracle 11g. PL/SQL поддерживает циклы трех видов: простые LOOP (бесконечные), FOR и WHILE. Каждая разновидность циклов предназначена для определенных целей, имеет свои нюансы и правила использования. Из представленных ниже таблиц вы узнаете, как завершается цикл, когда проверяется условие его завершения и в каких случаях применяются циклы того или иного вида.

Свойство	**Описание**
Условие завершения цикла	Код выполняется многократно. Как остановить выполнение тела цикла?
Когда проверяется условие завершения цикла	Когда выполняется проверка условия завершения — в начале или в конце цикла? К каким последствиям это приводит?
В каких случаях используется данный цикл	Какие специальные факторы необходимо учитывать, если цикл подходит для вашей ситуации?

Основы циклов языка PL/SQL

Зачем нужны три разновидности циклов? Чтобы вы могли выбрать оптимальный способ решения каждой конкретной задачи. В большинстве случаев задачу можно решить с помощью любой из трех циклических конструкций, но при неудачном выборе конструкции вам придется написать множество лишних строк программного кода, а это затруднит понимание и сопровождение написанных модулей.

Примеры разных циклов

Чтобы дать начальное представление о разных циклах и о том, как они работают, рассмотрим три процедуры. В каждом случае для каждого года в диапазоне от начального до конечного значения вызывается процедура display_total_sales.

Простой цикл начинается с ключевого слова LOOP и завершается командой END LOOP. Выполнение цикла прерывается при выполнении команды EXIT, EXIT WHEN или RETURN в теле цикла (или при возникновении исключения):

```
PROCEDURE display_multiple_years (
  start_year_in IN PLS_INTEGER
  ,end_year_in IN PLS_INTEGER
  )
IS
  l_current_year PLS_INTEGER := start_year_in;
BEGIN
```

```
  LOOP
    EXIT WHEN l_current_year > end_year_in;
    display_total_sales (l_current_year);
    l_current_year := l_current_year + 1;
  END LOOP;
END display_multiple_years;
```

Цикл FOR существует в двух формах: числовой и курсорной. В числовых циклах FOR программист задает начальное и конечное целочисленные значения, а PL/SQL перебирает все промежуточные значения, после чего завершает цикл:

```
PROCEDURE display_multiple_years (
  start_year_in IN PLS_INTEGER
  ,end_year_in IN PLS_INTEGER
  )
IS
BEGIN
  FOR l_current_year IN start_year_in .. end_year_in
  LOOP
    display_total_sales (l_current_year);
  END LOOP;
END display_multiple_years;
```

Курсорная форма цикла FOR имеет аналогичную базовую структуру, но вместо границ числового диапазона в ней задается курсор или конструкция SELECT:

```
PROCEDURE display_multiple_years (
  start_year_in IN PLS_INTEGER
  ,end_year_in IN PLS_INTEGER
)
IS
BEGIN
  FOR l_current_year IN (
    SELECT * FROM sales_data
    WHERE year BETWEEN start_year_in AND end_year_in)
  LOOP
    -- Процедуре передается запись, неявно объявленная
```

```
    -- с типом sales_data%ROWTYPE...
    display_total_sales (l_current_year);
  END LOOP;
END display_multiple_years;
```

Цикл WHILE имеет много общего с простым циклом. Принципиальное отличие заключается в том, что условие завершения проверяется перед выполнением очередной итерации. Возможны ситуации, в которых тело цикла не будет выполнено ни одного раза:

```
PROCEDURE display_multiple_years (
  start_year_in IN PLS_INTEGER
  ,end_year_in IN PLS_INTEGER
)
IS
  l_current_year PLS_INTEGER := start_year_in;
BEGIN
  WHILE (l_current_year <= end_year_in)
  LOOP
    display_total_sales (l_current_year);
    l_current_year := l_current_year + 1;
  END LOOP;
END display_multiple_years;
```

В приведенных примерах самым компактным получился цикл FOR. Однако его можно использовать только потому, что нам заранее известно, сколько раз будет выполняться тело цикла. Во многих других случаях количество повторений может быть заранее неизвестно, поэтому для них цикл FOR не подходит.

Структура циклов PL/SQL

Несмотря на различия между разными формами циклических конструкций, каждый цикл состоит из двух частей: ограничителей и тела цикла.

Ограничители — ключевые слова, определяющие начало цикла, условие завершения, и команда END LOOP, завершающая цикл**. Тело цикла** — последовательность исполняемых команд внутри границ цикла, выполняемых на каждой итерации.

На рис. 1 изображена структура цикла WHILE.

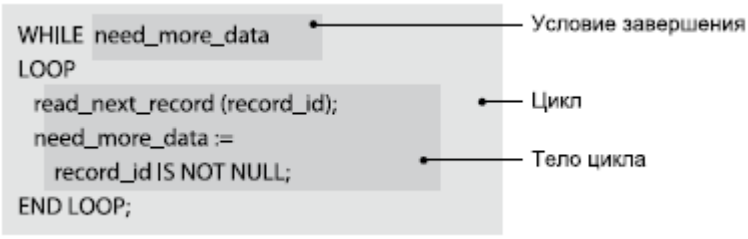


Рис. 1. Цикл WHILE и его тело

В общем случае цикл можно рассматривать как процедуру или функцию. Его тело — своего рода «черный ящик», а условие завершения — это интерфейс «черного ящика». Код, находящийся вне цикла, ничего не должен знать о происходящем внутри него. Помните об этом при **рассмотрении различных форм циклов** в этой статье.

Простой цикл PL/SQL

Структура простого цикла является самой элементарной среди всех циклических конструкций. Такой цикл состоит из ключевого слова LOOP, исполняемого кода (тела цикла) и ключевых слов END LOOP:

```
LOOP
  исполняемые_команды
END LOOP;
```

Цикл начинается командой LOOP, а заканчивается командой END LOOP. Тело цикла должно содержать как минимум одну исполняемую команду. Свойства простого цикла описаны в следующей таблице.

Свойство	**Описание**
Условие завершения цикла	Если в теле цикла выполняется команда EXIT. В противном случае цикл выполняется бесконечно
Когда проверяется условие завершения цикла	В теле цикла и только при выполнении команды EXIT или EXIT WHEN. Таким образом, тело цикла (или его часть) всегда выполняется как минимум один раз
В каких случаях используется данный цикл	(1) Если не известно, сколько раз будет выполняться тело цикла; (2) тело цикла должно быть выполнено хотя бы один раз

Простой цикл удобно использовать, когда нужно гарантировать хотя бы однократное выполнение тела цикла (или хотя бы его части). Так как цикл не имеет условия, которое бы определяло, должен ли он выполняться или нет, тело цикла всегда будет выполнено хотя бы один раз.

Простой цикл завершается только в том случае, если в его теле выполняется команда EXIT (или ее «близкий родственник» — EXIT WHEN) или же в нем инициируется исключение (оставшееся необработанным).

Завершение простого цикла: EXIT и EXIT WHEN

Если вы не хотите, чтобы программа «зациклилась», в теле цикла следует разместить команду EXIT или EXIT WHEN:

EXIT;

EXIT WHEN условие;

Здесь условие — это логическое выражение.

В следующем примере команда EXIT прерывает выполнение цикла и передает управление команде, следующей за командой END LOOP. Функция account_balance возвращает остаток денег на счету с идентификатором account_id. Если на счету осталось менее 1000 долларов, выполняется команда EXIT и цикл завершается. В противном случае программа снимает с банковского счета клиента сумму, необходимую для оплаты заказов.

```
LOOP
  balance_remaining := account_balance (account_id);
  IF balance_remaining < 1000
  THEN
    EXIT;
  ELSE
    apply_balance (account_id, balance_remaining);
  END IF;
END LOOP;
```

Команда EXIT может использоваться только в цикле LOOP. В PL/SQL для выхода из цикла также предусмотрена команда EXIT WHEN, предназначенная для завершения цикла с проверкой дополнительного условия. В сущности, EXIT WHEN сочетает в себе функции IF-THEN и EXIT. Приведенный пример можно переписать с использованием EXIT WHEN:

```
LOOP
  /* Вычисление баланса */
  balance_remaining := account_balance (account_id);
  /* Условие встраивается в команду EXIT */
EXIT WHEN balance_remaining < 1000;
  /* Если цикл все еще выполняется, с баланса списываются средства */
  apply_balance (account_id, balance_remaining);
END LOOP;
```

Как видите, во второй форме для проверки условия завершения команда IF не нужна. Логика проверки условия встраивается в команду EXIT WHEN. Так в каких же случаях следует использовать команду EXIT WHEN, а в каких — просто EXIT?

- Команда EXIT WHEN подойдет, когда условие завершения цикла определяется одним выражением. Предыдущий пример наглядно демонстрирует этот сценарий.
- При нескольких условиях завершения цикла или если при выходе должно быть определено возвращаемое значение, предпочтительнее IF или CASE с EXIT.

В следующем примере удобнее использовать команду EXIT. Фрагмент кода взят из функции, сравнивающей содержимое двух файлов:

```
  ...
  IF (end_of_file1 AND end_of_file2)
  THEN
    retval := TRUE;
    EXIT;
  ELSIF (checkline != againstline)
  THEN
    retval := FALSE;
    EXIT;
  ELSIF (end_of_file1 OR end_of_file2)
  THEN
    retval := FALSE;
    EXIT;
  END IF;
END LOOP;
```

Моделирование цикла REPEAT UNTIL

В PL/SQL отсутствует традиционный цикл REPEAT UNTIL, в котором условие проверяется после выполнения тела цикла (что гарантирует выполнение тела как минимум один раз). Однако этот цикл легко моделируется простым циклом следующего вида:

```
LOOP
  ... тело цикла ...
  EXIT WHEN логическое_условие;
END LOOP;
```

Здесь логическое_условие — логическая переменная или выражение, результатом проверки которого является значение TRUE или FALSE (или NULL).

Бесконечный цикл

Некоторые программы (например, средства наблюдения за состоянием системы) рассчитаны на непрерывное выполнение с накоплением необходимой информации. В таких случаях можно намеренно использовать бесконечный цикл:

```
LOOP
  сбор_данных;
END LOOP;
```

Но каждый программист, имевший дело с зацикливанием, подтвердит: бесконечный цикл обычно поглощает значительную часть ресурсов процессора. Проблема решается приостановкой выполнения между итерациями (и, разумеется, максимально возможной эффективностью сбора данных):

```
LOOP
  сбор_данных;
  DBMS_LOCK.sleep(10); -- ничего не делать в течение 10 секунд
END LOOP;
```

Во время приостановки программа практически не расходует ресурсы процессора.

Цикл WHILE

Условный цикл WHILE выполняется до тех пор, пока определенное в цикле условие остается равным TRUE. А поскольку возможность выполнения цикла зависит от условия и не ограничивается фиксированным количеством повторений, он используется именно в тех случаях, когда количество повторений цикла заранее не известно.

Прерывание бесконечного цикла

На практике возможна ситуация, в которой бесконечный цикл потребуется завершить. Если цикл выполняется в анонимном блоке в SQL*Plus, скорее всего, проблему можно решить вводом терминальной комбинации завершения (обычно Ctrl+C). Но реальные программы чаще выполняются в виде сохраненных процедур, и даже уничтожение процесса, запустившего программу (например, SQL*Plus), не приведет к остановке фоновой задачи. Как насчет команды ALTER SYSTEM KILL SESSION? Хорошая идея, но в некоторых версиях Oracle эта команда не уничтожает зациклившиеся сеансы (почему — никто не знает). Как же «прикончить» выполняемую программу?

Возможно, вам придется прибегнуть к таким средствам операционной системы, как команда kill в Unix/Linux и orakill.exe в Microsoft Windows. Для выполнения этих команд необходимо знать идентификатор процесса «теневой задачи» Oracle; впрочем, нужную информацию легко получить при наличии привилегий чтения для представлений V$SESSION и V$PROCESS. Но даже если неэлегантное решение вас не

пугает, приходится учитывать другой фактор: в режиме сервера это, вероятно, приведет к уничтожению других сеансов. Лучшее решение, которое я могу предложить, — вставить в цикл своего рода «интерпретатор команд», использующий встроенный в базу данных механизм межпроцессных коммуникаций — «каналов» (pipes):

```
DECLARE
  pipename CONSTANT VARCHAR2(12) := 'signaler';
  result INTEGER;
  pipebuf VARCHAR2(64);
BEGIN
  /* Создание закрытого канала с известным именем */
  result := DBMS_PIPE.create_pipe(pipename);
  LOOP
    data_gathering_procedure;
    DBMS_LOCK.sleep(10);
    /* Проверка сообщений в канале */
    IF DBMS_PIPE.receive_message(pipename, 0) = 0
    THEN
      /* Интерпретация сообщения с соответствующими действиями */
      DBMS_PIPE.unpack_message(pipebuf);
      EXIT WHEN pipebuf = 'stop';
    END IF;
  END LOOP;
END;
```

Использование DBMS_PIPE не оказывает заметного влияния на общую загрузку процессора.

Простая вспомогательная программа может уничтожить зациклившуюся программу, отправив по каналу сообщение «stop»:

```
DECLARE
  pipename VARCHAR2 (12) := 'signaler';
  result INTEGER := DBMS_PIPE.create_pipe (pipename);
BEGIN
  DBMS_PIPE.pack_message ('stop');
  result := DBMS_PIPE.send_message (pipename);
```

END;

По каналу также можно отправлять другие команды — например, команду увеличения или уменьшения интервала ожидания. Кстати говоря, в приведенном примере используется закрытый канал, так что сообщение STOP должно отправляться с той же учетной записи пользователя, которая выполняет бесконечный цикл. Также следует заметить, что пространство имен базы данных для закрытых каналов глобально по отношению ко всем сеансам текущего пользователя. Следовательно, если вы захотите, чтобы в бесконечном цикле выполнялось сразу несколько программ, необходимо реализовать дополнительную логику для (1) создания имен каналов, уникальных для каждого сеанса, и (2) определения имен каналов для отправки команды STOP.

Общий синтаксис цикла WHILE:

WHILE условие

LOOP

  исполняемые_команды

END LOOP;

Здесь условие — логическая переменная или выражение, результатом проверки которого является логическое значение TRUE, FALSE или NULL. Условие проверяется при каждой итерации цикла. Если результат оказывается равным TRUE, тело цикла выполняется.

Если же результат равен FALSE или NULL, то цикл завершается, а управление передается исполняемой команде, следующей за командой END LOOP. Основные свойства цикла WHILE приведены в таблице.

Свойство	**Описание**
Условие завершения цикла	Если значением логического выражения цикла является FALSE или NULL
Когда проверяется условие завершения цикла	Перед первым и каждым последующим выполнением тела цикла. Таким образом, не гарантируется даже однократное выполнение тела цикла WHILE
В каких случаях используется данный цикл	(1) Если не известно, сколько раз будет выполняться тело цикла; (2) возможность выполнения цикла должна определяться условием; (3) тело цикла может не выполняться ни одного раза

Условие WHILE проверяется в начале цикла и в начале каждой его итерации, перед выполнением тела цикла. Такого рода проверка имеет два важных последствия:

- Вся информация, необходимая для вычисления условия, должна задаваться перед первым выполнением цикла.
- Может оказаться, что цикл WHILE не будет выполнен ни одного раза.

Следующий пример цикла WHILE взят из файла datemgr.pkg. Здесь используется условие, представленное сложным логическим выражением. Прерывание цикла WHILE вызвано одной из двух причин: либо завершением списка масок даты, которые применяются для выполнения преобразования, либо успешным завершением преобразования (и теперь переменная date_converted содержит значение TRUE):

```
WHILE mask_index <= mask_count AND NOT date_converted
LOOP
  BEGIN
    /* Попытка преобразования строки по маске в записи таблицы */
    retval := TO_DATE (value_in, fmts (mask_index));
    date_converted := TRUE;
  EXCEPTION
    WHEN OTHERS
    THEN
      mask_index:= mask_index+ 1;
  END;
END LOOP;
```

Цикл FOR со счетчиком

В PL/SQL существует два вида цикла FOR: с числовым счетчиком и с курсором. Цикл со счетчиком — это традиционный, хорошо знакомый всем программистам цикл FOR, поддерживаемый в большинстве языков программирования. Количество итераций этого цикла известно еще до его начала; оно задается в диапазоне между ключевыми словами FOR и LOOP.

Диапазон неявно объявляет управляющую переменную цикла (если она не была явно объявлена ранее), определяет начальное и конечное значения диапазона, а также задает направление изменения счетчика (по возрастанию или по убыванию).

Общий синтаксис цикла FOR:

```
FOR счетчик IN [REVERSE] начальное_значение .. конечное_значение
LOOP
  исполняемые_команды
END LOOP;
```

Между ключевыми словами LOOP и END LOOP должна стоять хотя бы одна исполняемая команда. Свойства цикла FOR с числовым счетчиком приведены в следующей таблице.

Свойство	Описание
Условие завершения цикла	Числовой цикл FOR безусловно завершается при выполнении количества итераций, определенного диапазоном значений счетчика. (Цикл может завершаться и командой EXIT, но делать этого не рекомендуется)
Когда проверяется условие завершения цикла	После каждого выполнения тела цикла компилятор PL/SQL проверяет значение счетчика. Если оно выходит за пределы заданного диапазона, выполнение цикла прекращается. Если начальное значение больше конечного, то тело цикла не выполняется ни разу
В каких случаях используется данный цикл	Если тело цикла должно быть выполнено определенное количество раз, а выполнение не должно прерываться преждевременно

Правила для циклов FOR с числовым счетчиком

При использовании цикла FOR с числовым счетчиком необходимо следовать некоторым правилам:

- **Не объявляйте счетчик цикла**. PL/SQL автоматически неявно объявляет локальную переменную с типом данных INTEGER. Область действия этой переменной совпадает с границей цикла; обращаться к счетчику за пределами цикла нельзя.
- **Выражения, используемые при определении диапазона** (начального и конечного значений), вычисляются один раз. Они не пересчитываются в ходе выполнения цикла. Если изменить внутри цикла переменные, используемые для определения диапазона значений счетчика, его границы останутся прежними.
- **Никогда не меняйте значения счетчика и границ диапазона внутри цикла**. Это в высшей степени порочная практика. Компилятор PL/SQL либо выдаст сообщение об ошибке, либо проигнорирует изменения — в любом случае возникнут проблемы.
- **Чтобы значения счетчика уменьшались в направлении от конечного к начальному, используйте ключевое слово REVERS**E. При этом первое значение в определении диапазона (начальное_значение) должно быть меньше второго (конечное_значение). Не меняйте порядок следования значений — просто поставьте ключевое слово REVERSE.

Примеры циклов FOR с числовым счетчиком

Следующие примеры демонстрируют некоторые варианты синтаксиса циклов FOR с числовым счетчиком.

- Цикл выполняется 10 раз; счетчик увеличивается от 1 до 10:

```
FOR loop_counter IN 1 .. 10
LOOP
  ... исполняемые_команды ...
END LOOP;
```

- Цикл выполняется 10 раз; счетчик уменьшается от 10 до 1:

```
FOR loop_counter IN REVERSE 1 .. 10
LOOP
  ... исполняемые_команды ...
END LOOP;
```

- Цикл не выполняется ни разу. В заголовке цикла указано ключевое слово REVERSE, поэтому счетчик цикла loop_counter изменяется от большего значения к меньшему. Однако начальное и конечное значения заданы в неверном порядке:

```
FOR loop_counter IN REVERSE 10 .. 1
LOOP
  /* Тело цикла не выполнится ни разу! */
  ...
END LOOP;
```

Даже если задать обратное направление с помощью ключевого слова REVERSE, меньшее значение счетчика все равно должно быть задано перед большим. Если первое число больше второго, тело цикла не будет выполнено. Если же граничные значения одинаковы, то тело цикла будет выполнено один раз.

- Цикл выполняется для диапазона, определяемого значениями переменной и выражения:

```
FOR calc_index IN start_period_number ..
        LEAST (end_period_number, current_period)
LOOP
  ... исполняемые команды ...
END LOOP;
```

В этом примере количество итераций цикла определяется во время выполнения программы. Начальное и конечное значения вычисляются один раз, перед началом цикла, и затем используются в течение всего времени его выполнения.

Нетривиальные приращения

В PL/SQL не предусмотрен синтаксис задания шага приращения счетчика. Во всех разновидностях цикла FOR с числовым счетчиком значение счетчика на каждой итерации всегда увеличивается или уменьшается на единицу.

Если приращение должно быть отлично от единицы, придется писать специальный код. Например, что нужно сделать, чтобы тело цикла выполнялось только для четных чисел из диапазона от 1 до 100? Во-первых, можно использовать числовую функцию MOD, как в следующем примере:

```
FOR loop_index IN 1 .. 100
LOOP
  IF MOD (loop_index, 2) = 0
  THEN
    /* Число четное, поэтому вычисления выполняются */
    calc_values (loop_index);
  END IF;
END LOOP;
```

Также возможен и другой способ — умножить значение счетчика на два и использовать вдвое меньший диапазон:

```
FOR even_number IN 1 .. 50
LOOP
  calc_values (even_number*2);
END LOOP;
```

В обоих случаях процедура calc_values выполняется только для четных чисел. В первом примере цикл FOR повторяется 100 раз, во втором — только 50.

Но какое бы решение вы ни выбрали, обязательно подробно его опишите. Комментарии помогут другим программистам при сопровождении вашей программы.

Цикл FOR с курсором

Курсорная форма цикла FOR связывается с явно заданным курсором (а по сути, определяется им) или инструкцией SELECT, заданной непосредственно в границах цикла. Используйте эту форму только в том случае, если вам нужно извлечь и обработать все записи курсора (впрочем, при работе с курсорами это приходится делать довольно часто).

Цикл FOR с курсором — одна из замечательных возможностей PL/SQL, обеспечивающая тесную и эффективную интеграцию процедурных конструкций с мощью языка доступа к базам данных SQL. Его применение заметно сокращает объем кода, необходимого для выборки данных из курсора, а также уменьшает вероятность возникновения ошибок при циклической обработке данных — ведь именно циклы являются одним из основных источников ошибок в программах.

Базовый синтаксис цикла FOR с курсором:

FOR запись IN { имя_курсора | (команда_SELECT) }

LOOP

  исполняемые команды

END LOOP;

Здесь запись — неявно объявленная запись с атрибутом %ROWTYPE для курсора имя_курсора.

Не объявляйте явно запись с таким же именем, как у индексной записи цикла. В этом нет необходимости, поскольку запись объявляется автоматически, к тому же это может привести к логическим ошибкам. О том, как получить доступ к информации о записях, обработанных в цикле FOR после его выполнения, рассказывается далее в этом блоге.

В цикле FOR можно также задать не курсор, а непосредственно SQL-инструкцию SELECT, как показано в следующем примере:

FOR book_rec IN (SELECT * FROM books)

LOOP

  show_usage (book_rec);

END LOOP;

Мы не рекомендуем использовать эту форму, поскольку встраивание инструкций SELECT в «неожиданные» места кода затрудняет его сопровождение и отладку.

Свойства цикла FOR с использованием курсора приведены в следующей таблице.

Свойство	**Описание**
Условие завершения цикла	Выборка всех записей курсора. Цикл можно завершить и командой EXIT, но поступать так не рекомендуется
Когда проверяется условие завершения цикла	После каждого выполнения тела цикла компилятор PL/SQL осуществляет выборку очередной записи. Если значение атрибута курсора %NOTFOUND% оказывается равным TRUE, цикл завершается. Если курсор не возвратит ни одной строки, тело цикла никогда не будет выполнено
В каких случаях используется данный цикл	При необходимости выбрать и обработать каждую запись курсора

Примеры цикла FOR с курсором

Допустим, нам необходимо обновить счета владельцев всех животных, живущих в специальном отеле. Следующий пример включает анонимный блок, в котором для выбора номера комнаты и идентификатора животного используется курсор occupancy_cur. Процедура update_bill вносит все изменения в счет:

```
1 DECLARE
2    CURSOR occupancy_cur IS
3    SELECT pet_id, room_number
4     FROM occupancy WHERE occupied_dt = TRUNC (SYSDATE);
5    occupancy_rec occupancy_cur%ROWTYPE;
6 BEGIN
7    OPEN occupancy_cur;
8    LOOP
9     FETCH occupancy_cur INTO occupancy_rec;
10     EXIT WHEN occupancy_cur%NOTFOUND;
11     update_bill
12       (occupancy_rec.pet_id, occupancy_rec.room_number);
13   END LOOP;
14   CLOSE occupancy_cur;
15 END;
```

Этот код последовательно и явно выполняет все необходимые действия: мы определяем курсор (строка 2), явно объявляем запись для этого курсора (строка 5), открываем курсор (строка 7), начинаем бесконечный цикл (строка 8), производим выборку записи из курсора (строка 9), проверяем условие выхода из цикла (конец данных) по атрибуту %NOTFOUND курсора (строка 10) и, наконец, выполняем обновление (строка 11). После этого программист должен закрыть курсор (строка 14). Вот что получится, если переписать тот же код с использованием цикла FOR с курсором:

```
DECLARE
  CURSOR occupancy_cur IS
    SELECT pet_id, room_number
     FROM occupancy WHERE occupied_dt = TRUNC (SYSDATE);
BEGIN
  FOR occupancy_rec IN occupancy_cur
  LOOP
    update_bill (occupancy_rec.pet_id, occupancy_rec.room_number);
  END LOOP;
END;
```

Как все просто и понятно! Исчезло объявление записи. Исчезли команды OPEN, FETCH и CLOSE. Больше не нужно проверять атрибут %NOTFOUND.

Нет никаких сложностей с организацией выборки данных. По сути, вы говорите PL/SQL: «Мне нужна каждая строка таблицы, и я хочу, чтобы она была помещена в запись, соответствующую курсору». И PL/SQL делает то, что вы хотите, как это должен делать любой современный язык программирования.

Курсору в цикле FOR, как и любому другому курсору, можно передавать параметры. Если какой-либо из столбцов списка SELECT определяется выражением, обязательно определите для него псевдоним. Для обращения к конкретному значению в записи курсора в пределах цикла необходимо использовать «точечный» синтаксис (имя_записи.имя_столбца — например, occupancy_rec.room_number), так что без псевдонима к столбцу-выражению обратиться не удастся.

Метки циклов

Циклу можно присвоить имя при помощи метки. Метка цикла в PL/SQL имеет стандартный формат:

<<*имя_метки*>>

Метка располагается непосредственно перед командой LOOP:

```
<<all_emps>>
FOR emp_rec IN emp_cur
LOOP
  ...
END LOOP;
```

Эту же метку можно указать и после ключевых слов END LOOP, как в следующем примере:

```
<<year_loop>>
WHILE year_number <= 1995
LOOP
  <<month_loop>>
  FOR month_number IN 1 .. 12
  LOOP
    ...
  END LOOP month_loop;
  year_number := year_number + 1;
END LOOP year_loop;
```

Метки циклов могут пригодиться в нескольких типичных ситуациях:

- Если вы написали очень длинный цикл с множеством вложенных циклов (допустим, начинающийся в строке 50, завершается в строке 725 и содержащий 16 вложенных циклов), используйте метку цикла для того, чтобы явно связать его конец с началом. Визуальная пометка поможет при отладке и сопровождении программы; без нее будет трудно уследить, какая команда LOOP соответствует каждой из команд END LOOP.
- Метку цикла можно использовать для уточнения имени управляющей переменной цикла (записи или счетчика), что также упрощает чтение программы:

```
<<year_loop>>
FOR year_number IN 1800..1995
LOOP
  <<month_loop>>
  FOR month_number IN 1 .. 12
  LOOP
    IF year_loop.year_number = 1900 THEN ... END IF;
  END LOOP month_loop;
END LOOP year_loop;
```

- При использовании вложенных циклов метки упрощают чтение кода и повышают эффективность их выполнения. При желании выполнение именованного внешнего цикла можно прервать при помощи команды EXIT с заданной в нем меткой цикла:

```
EXIT метка_цикла;
EXIT метка_цикла WHEN условие;
```

Но обычно применять метки циклов подобным образом не рекомендуется, так как они ухудшают структуру логики программы (по аналогии с GOTO) и усложняют отладку. Если вам потребуется использовать подобный код, лучше изменить структуру цикла, а возможно, заменить его простым циклом или WHILE.

Команда CONTINUE

В Oracle11g появилась новая возможность для работы с циклами: команда CONTINUE. Он используется для выхода из текущей итерации цикла и немедленного перехода к следующей итерации. Как и EXIT, эта команда существует в двух формах: безусловной (CONTINUE) и условной (CONTINUE WHEN).

Простой пример использования CONTINUE WHEN для пропуска итераций с четными значениями счетчика:

```
BEGIN
  FOR l_index IN 1 .. 10
```

```
  LOOP
    CONTINUE WHEN MOD (l_index, 2) = 0;
    DBMS_OUTPUT.PUT_LINE ('Счетчик = ' || TO_CHAR (l_index));
  END LOOP;
END;
/
```

Результат:

Счетчик = 1

Счетчик = 3

Счетчик = 5

Счетчик = 7

Счетчик = 9

Конечно, того же эффекта можно добиться при помощи команды IF, но команда CONTINUE предоставляет более элегантный и понятный способ представления реализуемой логики.

Команда CONTINUE чаще всего применяется для модификации существующего кода с внесением целенаправленных изменений и немедленным выходом из цикла для предотвращения побочных эффектов.

Так ли плоха команда CONTINUE?

Когда я впервые узнал о команде CONTINUE, на первый взгляд мне показалось, что она представляет очередную форму неструктурированной передачи управления по аналогии с GOTO, поэтому ее следует по возможности избегать (я прекрасно обходился без нее годами!). Чарльз Уэзерелл, один из руководителей группы разработки PL/SQL, развеял мои заблуждения:

Уже давно (еще в эпоху знаменитого манифеста Дейкстры «о вреде goto») конструкции exit и continue были проанализированы и отнесены к структурным средствам передачи управления. Более того, команда exit была признана в одной из авторитетных работ Кнута как способ корректного прерывания вычислений.

Бем и Якопини доказали, что любая программа, использующая произвольные синхронные управляющие элементы (например, циклы или goto), может быть переписана с использованием циклов while, команд if и логических переменных в полностью структурной форме. Более того, преобразование между «плохой» неструктурированной версией и «хорошей» структурированной версией в программе может быть автоматизировано. К сожалению, новая «хорошая» программа может на порядок увеличиваться в размерах из-за необходимости введения многочисленных логических переменных и копирования кода во множественные ветви if. На практике в реальных программах такое увеличение размера встречается редко, но для моделирования эффекта continue и exit часто применяется копирование кода. Оно создает проблемы с

сопровождением, потому что если в будущем программу потребуется модифицировать, программист должен помнить, что изменить нужно все копии вставленного кода.

Команда continue полезна тем, что она делает код более компактным и понятным, а также сокращает необходимость в логических переменных, смысл которых трудно понять с первого взгляда. Чаще всего она используется в циклах, в которых точная обработка каждого элемента зависит от подробных структурных тестов. Заготовка цикла может выглядеть так, как показано ниже; обратите внимание на команду exit, которая проверяет, не пора ли завершить обработку. Также стоит заметить, что последняя команда continue (после условия5) не является строго необходимой. С другой стороны, включение continue после каждого действия упрощает добавление новых действий в произвольном порядке без нарушения работоспособности других действий.

```
LOOP
  EXIT WHEN условие_выхода;
  CONTINUE WHEN условие1;
  CONTINUE WHEN условие2;
  подготовительная_фаза;
  IF условие4 THEN
    выполнено_действие4;
    CONTINUE;
  END IF;
  IF условие5 THEN
    выполнено_действие5;
    CONTINUE; -- Не является строго необходимой.
  END IF;
END LOOP;
```

Без команды continue мне пришлось бы реализовать тело цикла следующим образом:

```
LOOP
  EXIT WHEN exit_condition_met;
  IF condition1
    THEN
      NULL;
    ELSIF condition2
    THEN
      NULL;
    ELSE
```

```
      setup_steps_here;
    IF condition4 THEN
      action4_executed;
    ELSIF condition5 THEN
    action5_executed;
    END IF;
  END IF;
END LOOP;
```

Даже в этом простом примере команда continue позволяет обойтись без нескольких секций elsif, сокращает уровень вложенности и наглядно показывает, какие логические проверки (и сопутствующая обработка) должны выполняться на том же уровне. В частности, continue существенно сокращает глубину вложенности. Умение правильно использовать команду continue безусловно помогает программистам PL/SQL писать более качественный код.

Также команда CONTINUE пригодится для завершения внутренних циклов и немедленного продолжения следующей итерации внешнего цикла. Для этого циклам присваиваются имена при помощи меток. Пример:

```
BEGIN
  <<outer>>
  FOR outer_index IN 1 .. 5
  LOOP
    DBMS_OUTPUT.PUT_LINE (
      'Внешний счетчик = ' || TO_CHAR (outer_index));


    <<inner>>
    FOR inner_index IN 1 .. 5
    LOOP
      DBMS_OUTPUT.PUT_LINE (
        ' Внутренний счетчик = ' || TO_CHAR (inner_index));
      CONTINUE outer;
    END LOOP inner;
  END LOOP outer;
END;
/
```

Результат:

Внешний счетчик = 1

Внутренний счетчик = 1

Внешний счетчик = 2

Внутренний счетчик = 1

Внешний счетчик = 3

Внутренний счетчик = 1

Внешний счетчик = 4

Внутренний счетчик = 1

Внешний счетчик = 5

Внутренний счетчик = 1

Полезные советы по работе с циклами в PL/SQL

Циклы — очень мощные и полезные конструкции, но при их использовании необходима осторожность. Именно циклы часто создают проблемы с быстродействием программ, и любая ошибка, возникшая в цикле, повторяется ввиду многократности его выполнения. Логика, определяющая условие остановки цикла, бывает очень сложной. В этом разделе приводятся несколько советов по поводу того, как сделать циклы более четкими и понятными, а также упростить их сопровождение.

Используйте понятные имена для счетчиков циклов

Не заставляйте программиста, которому поручено сопровождать программу, с помощью сложной дедукции определять смысл начального и конечного значения счетчика цикла FOR. Применяйте понятные и информативные имена переменных и циклов, и тогда другим программистам (да и вам самим некоторое время спустя) легко будет разобраться в таком коде.

Как можно понять следующий ход, не говоря уже о его сопровождении?

```
FOR i IN start_id .. end_id
LOOP
  FOR j IN 1 .. 7
  LOOP
    FOR k IN 1 .. 24
    LOOP
      build_schedule (i, j, k);
    END LOOP;
```

```
  END LOOP;
END LOOP;
```

Трудно представить, зачем использовать однобуквенные имена переменных, словно сошедшие со страниц учебника начального курса алгебры, но это происходит сплошь и рядом. Вредные привычки, приобретенные на заре эпохи программирования, искоренить невероятно сложно. А исправить этот код очень просто — достаточно присвоить переменным более информативные имена:

```
FOR focus_account IN start_id .. end_id
LOOP
  FOR day_in_week IN 1 .. 7
  LOOP
    FOR month_in_biyear IN 1 .. 24
    LOOP
      build_schedule (focus_account, day_in_week, month_in_biyear);
    END LOOP;
  END LOOP;
END LOOP;
```

С содержательными именами переменных сразу видно, что внутренний цикл просто перебирает месяцы двухлетнего периода (12 × 2 = 24).

Корректно выходите из цикла

Один из фундаментальных принципов структурного программирования звучит так: «один вход, один выход»; иначе говоря, программа должна иметь одну точку входа и одну точку выхода. Первая часть в PL/SQL реализуется автоматически. Какой бы цикл вы ни выбрали, у него всегда только одна точка входа — первая исполняемая команда, следующая за ключевым словом LOOP. Но вполне реально написать цикл с несколькими точками выхода. Однако так поступать не рекомендуется, поскольку цикл с несколькими путями выхода трудно отлаживать и сопровождать.

При завершении цикла следует придерживаться следующих правил:

- Не используйте в циклах FOR и WHILE команды EXIT и EXIT WHEN. Цикл FOR должен завершаться только тогда, когда исчерпаны все значения диапазона (целые числа или записи). Команда EXIT в цикле FOR прерывает этот процесс, а следовательно, идет вразрез с самим назначением цикла FOR. Точно так же условие окончания цикла WHILE задается в самой команде WHILE и нигде более задавать или дополнять его не следует.
- Не используйте в циклах команды RETURN и GOTO, поскольку это вызывает преждевременное и неструктурированное завершение цикла. Применение указанных команд может выглядеть заманчиво, так как они сокращают объем кода.

Однако спустя некоторое время вы потратите больше времени, пытаясь понять, изменить и отладить такой код.

Рассмотрим суть этих правил на примере цикла FOR с курсором. Как вы уже видели, данный цикл облегчает перебор возвращаемых курсором записей, но не подходит для случаев, когда выход из цикла определяется некоторым условием, основанным на данных текущей записи. Предположим, что в цикле записи курсора просматриваются до тех пор, пока сумма значений определенного столбца не превысит максимальное значение, как показано в следующем примере. Хотя это можно сделать с помощью цикла FOR и курсора, выполнив внутри этого цикла команду EXIT, поступать так не следует.

```
1 DECLARE
2    CURSOR occupancy_cur IS
3       SELECT pet_id, room_number
4          FROM occupancy WHERE occupied_dt = TRUNC (SYSDATE);
5    pet_count INTEGER := 0;
6 BEGIN
7    FOR occupancy_rec IN occupancy_cur
8    LOOP
9       update_bill
10         (occupancy_rec.pet_id, occupancy_rec.room_number);
11      pet_count := pet_count + 1;
12      EXIT WHEN pet_count >= pets_global.max_pets;
13   END LOOP;
14 END;
```

В заголовке цикла FOR явно указано, что его тело должно быть выполнено n раз (где n — количество итераций в цикле со счетчиком или количество записей в цикле с курсором). Команда EXIT в цикле FOR (строка 12) изменяет логику его выполнения, и в результате получается код, который трудно понять и отладить.

Поэтому если нужно прервать цикл на основании информации текущей записи, лучше воспользоваться циклом WHILE или простым циклом, чтобы структура кода лучше отражала ваши намерения.

Получение информации о выполнении цикла FOR

Циклы FOR — удобные, четко формализованные структуры, которые выполняют в программе большую «административную» работу (особенно циклы с курсором). Однако у них есть и существенный недостаток: позволяя Oracle выполнять работу за нас, мы ограничиваем собственные возможности доступа к конечным результатам цикла после его завершения.

Предположим, нам нужно узнать, сколько записей обработано в цикле FOR с курсором, и затем использовать это значение в программе. Было бы очень удобно написать примерно такой код:

```
BEGIN
  FOR book_rec IN books_cur (author_in => 'FEUERSTEIN,STEVEN')
  LOOP
    ... обработка данных ...
  END LOOP;
  IF books_cur%ROWCOUNT > 10 THEN ...
```

Но попытавшись это сделать, мы получим сообщение об ошибке, поскольку курсор неявно открывается и закрывается Oracle. Как же получить нужную информацию из уже завершившегося цикла? Для этого следует объявить переменную в том блоке, в который входит цикл FOR, и присвоить ей значение в теле цикла — в таком случае переменная останется доступной и после завершения цикла. Вот как это делается:

```
DECLARE
  book_count PLS_INTEGER := 0;
BEGIN
  FOR book_rec IN books_cur (author_in => 'FEUERSTEIN,STEVEN')
  LOOP
    ... обработка данных ...
    book_count := books_cur%ROWCOUNT;
  END LOOP;
  IF book_count > 10 THEN ...
```

Команда SQL как цикл

На самом деле команда SQL (например, SELECT) тоже может рассматриваться как цикл, потому что она определяет действие, выполняемое компилятором SQL с набором данных. В некоторых случаях при реализации определенной задачи можно даже выбрать между использованием цикла PL/SQL и команды SQL. Давайте рассмотрим пример, а затем сделаем некоторые выводы о том, какое решение лучше.

Предположим, наша программа должна перенести информацию о выбывших из отеля животных из таблицы occupancy в таблицу occupancy_history. Опытный программист PL/SQL сходу выбирает цикл FOR с курсором. В теле цикла каждая выбранная из курсора запись сначала добавляется в таблицу occupancy_history, а затем удаляется из таблицы occupancy:

```
DECLARE
```

```
  CURSOR checked_out_cur IS
    SELECT pet_id, name, checkout_date
   v FROM occupancy WHERE checkout_date IS NOT NULL;
BEGIN
  FOR checked_out_rec IN checked_out_cur
  LOOP
    INSERT INTO occupancy_history (pet_id, name, checkout_date)
      VALUES (checked_out_rec.pet_id, checked_out_rec.name,
            checked_out_rec.checkout_date);
    DELETE FROM occupancy WHERE pet_id = checked_out_rec.pet_id;
  END LOOP;
END;
```

Программа работает, но является ли данное решение единственным? Конечно же, нет. Ту же логику можно реализовать с помощью команд SQL INSERT-SELECT FROM с последующей командой DELETE:

```
BEGIN
  INSERT INTO occupancy_history (pet_id, NAME, checkout_date)
  SELECT pet_id, NAME, checkout_date
    FROM occupancy WHERE checkout_date IS NOT NULL;
  DELETE FROM occupancy WHERE checkout_date IS NOT NULL;
END;
```

Каковы преимущества такого подхода? Код стал короче и выполняется более эффективно благодаря уменьшению количества переключений контекста (переходов от исполняемого ядра PL/SQL к исполняемому ядру SQL и обратно). Теперь обрабатываются только одна команда INSERT и одна команда DELETE.

Однако у «чистого» SQL-подхода имеются свои недостатки. Команда SQL обычно действует по принципу «все или ничего». Иначе говоря, если при обработке хотя бы одной записи occupancy_history происходит ошибка, то отменяются все инструкции INSERT и DELETE и ни одна запись не будет вставлена или удалена. Кроме того, приходится дважды записывать условие WHERE. В данном примере это не очень важно, но в более сложных запросах данное обстоятельство может иметь решающее значение. А первоначальный цикл FOR позволяет избежать дублирования потенциально сложной логики в нескольких местах.

Кроме того, PL/SQL превосходит SQL в отношении гибкости. Допустим, нам хотелось бы переносить за одну операцию максимально возможное количество записей, а для тех записей, при перемещении которых произошли ошибки, просто записывать сообщения в

журнал. В этом случае стоит воспользоваться циклом FOR с курсором, дополненным разделом исключений:

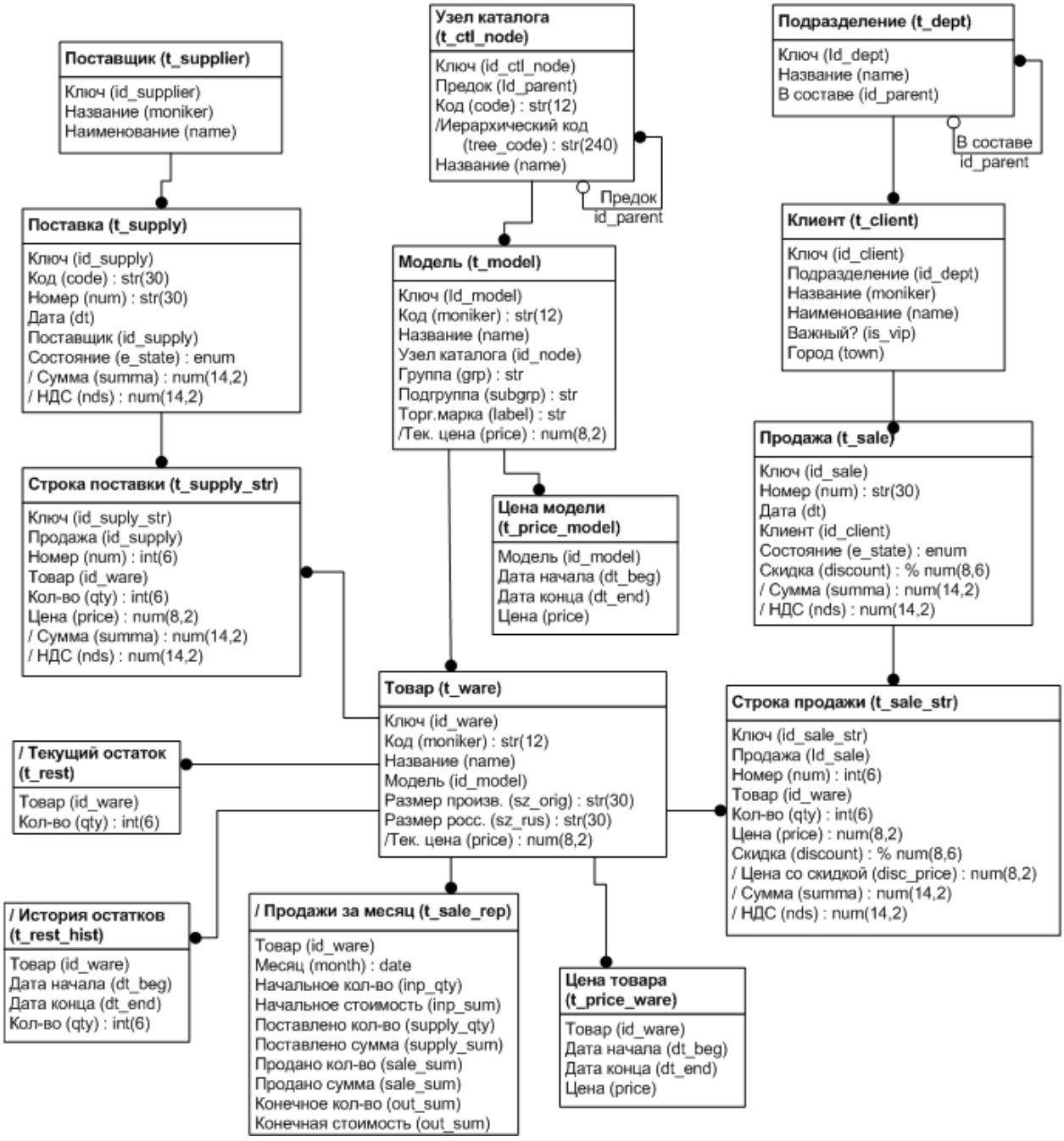
```
BEGIN
  FOR checked_out_rec IN checked_out_cur
  LOOP
    BEGIN
      INSERT INTO occupancy_history ...
      DELETE FROM occupancy ...
    EXCEPTION
      WHEN OTHERS THEN
        log_checkout_error (checked_out_rec);
    END;
  END LOOP;
END;
;
```

PL/SQL позволяет обрабатывать записи по одной и для каждой из них выполнять необходимые действия (которые могут базироваться на сложной процедурной логике, зависящей от содержимого конкретной записи). В таких случаях удобнее использовать комбинацию PL/SQL и SQL. Но если ваша задача позволяет ограничиться одним SQL, лучше им и воспользоваться — код получится и короче, и эффективнее.

Продолжить выполнение после ошибок в командах SQL можно двумя способами: (1) использовать конструкцию LOG ERRORS со вставкой, обновлением и удалением в Oracle10g Release 2 и выше; и (2) использовать конструкцию SAVE EXCEPTIONS в командах FOR ALL.

Задания по теме для аудиторной и самостоятельной работы

База данных эмулирует географически распределенную торговую компанию, торгующую с центрального склада. Каждое низовое подразделение отвечает за определенный регион (город), где набирает клиентов и организует продажи. Ассортимент товаров, цены и закупки осуществляются централизованно. Часть схемы, относящаяся к закупкам (и остаткам), пока не наполнена данными.



Особенности отдельных сущностей.

В интервалах цен Дата конца — всегда не входит.

Подразделения образуют иерархию. Для отчетов (запросов) будет использоваться выборка по иерархии. Однако название однозначно идентифицирует подразделение и его достаточно при показе информации.

Для именования клиентов и поставщиков применяется два атрибута: Название (moniker), которое должно быть уникальным и Наименование (name), которое соответствует официальным документам и используется в печатных формах.

Каталог товаров также иерархический. В отличие от подразделений, на каждом уровне ветки именуются коротким кодом, а для отражения в плоских отчетах и интерфейсах применяется иерархический код.

На строку продажи действуют две скидки: указанная в продаже (t_sale.discount) и дополнительная, назначенная индивидуально для строки (t_sale_str.discount).

Стоимость товара в каждой строке, естественно, получается умножением цены на количество, а для продаж — дополнительно с применением скидки. Теоретически его можно не хранить, либо можно не хранить цену, однако и то и другое часто нужно в разных отчетах, и поэтому оба атрибута обычно хранимые. Кроме того, необходима воспроизводимость при печати документов, не смотря на возможные проблемы округления, что тоже является аргументом за хранение всех атрибутов, а также НДС.

Общая сумма продажи и НДС продажи, естественно, равна сумме по всем позициям. Однако из тех же соображений удобства отчетов она обычно является хранимой.

Поставки и Продажи имеют два состояния: Черновик (new) и Исполнена (done). При исполнении документа изменяются остатки на складе. Пока документ — черновик, его можно свободно изменять. А исполненные документы править нельзя. Естественно, в реальных системах состояний больше, но для модельных целей этого достаточно.

Текущий остаток (t_rest) и История остатков (t_rest_hist) содержат текущий остаток на складе. Таблицы заполнены с учетом всех накладных и должны изменяться при их исполнении. В задания входит разработка соответствующих процедур, а заполненность таблиц используется в заданиях на запросы.

Продажи месяца (t_sale_rep) — таблица месячных отчетов по продажам. Естественно, в реальных системах отчетность устроена сложнее, однако отдельные таблицы для месячной или другой регулярной отчетности — достаточно типичная вещь. Отчет содержит данные за каждый месяц по каждому товару. Отчетные данные представлены как в штуках, так и в денежном выражении, и при построении итоговых отчетов, как правило, используются именно данные в деньгах. Товарный запас на складе оценивается в ценах поставки. Оценка на конец месяца получается следующим образом: к стоимости на начало месяца добавляют сумму закупок за месяц, а затем вычитают стоимость доли товара, которая была продана за месяц: out_sum = (inp_sum+supply_sum) * (1-sale_qty/(inp_qty+supply_qty))

Запросы:

1. Найти товары, цены на которые отличаются от цены на модель.
2. Найти продаже, цены без скидки в которых отличаются от цены на товар.
3. Вывод каталога с отступами в соответствии с уровнями каталога. Для каждого узла — число нижележащих узлов.
4. Вывод таблицы подразделений с отступами в соответствии с уровнями каталога
5. Вывод каталога с отступами, для листа с товарами — число моделей и товаров в ним.
6. Вывод каталога с отступами, для каждого узла — число моделей и товаров в нем и нижележащих узлах
7. Вычислить число продаж, их общую сумму, средневзвешенную, максимальную и минимальную скидки по каждому клиенту, у которых были продажи со скидкой более, чем 25%. Все — без учета скидок на строки
8. Вычислить число строк продаж, количество, сумму средневзвешенную, максимальную и минимальную скидку по торговой марке (t_model.label).
9. Выдать отчет о продажах по подразделениям, для каждого подразделения в дереве в отчет включать подчиненные. В отчете — число продаж, число строк, количество, общая сумма. Указание. Для начала сделать запрос, который бы для каждого подразделения выводил все подчиненные.
10. Выявить наиболее характерные значения скидок. Для этого построить распределение по предоставляемым скидкам, для каждой определить количество позиций, товаров и моделей, количество и сумму продаж.

11. Найти модели, на которые предоставлены нехарактерно большие скидки, с учетом предыдущего анализа. Сделать предположение о составе этих моделей

12. Вывод таблицы подразделений, для каждого узла — сумма продаж за заданный месяц. Написать запрос по таблице с отчетом и по исходным документам. Проверить соответствие.

13. Стандартная оборотка по товару в штуках за период: остаток на начало, приход, уход, остаток на конец. Различные группировки. Остатки выбираются из таблицы остатков, а приход и уход — из документов. С помощью отчета проверить таблицу остатков.

14. Различные отчеты по прибылям от продажи — в разрезе моделей, товарных групп, торговых марок, клиентов и подразделений. Сортировка по максимальной абсолютной и относительной прибыли.