*С.Урман*

**ORACLE8. ЯЗЫК ПРОГРАММИРОВАНИЯ PL/SQL**

Содержание

# Введение

Oracle является чрезвычайно мощной и гибкой системой реляционных баз данных. Однако эти качества привносят и свои сложности. Чтобы на основе Oracle разрабатывать полезные приложения, необходимо понимать, каким образом Oracle работает с хранящимися в системе данными. PL/SQL — это важное средство, разработанное для манипулирования данными как "внутри" Oracle, так и "снаружи", в приложениях пользователей. PL/SQL можно применять в различных средах самого разнообразного назначения.

Данная книга поможет понять, что такое PL/SQL, и оценить возможности этого уникального языка программирования. После ее прочтения вы сможете легко и эффективно использовать PL/SQL в своих приложениях. Когда вы освоите основы PL/SQL, книга станет надежным справочным руководством в повседневной работе с этим языком.

## Отличия от первого издания

Это второе издание книги Oracle PL/SQL Programming. Так в чем же его отличие от первого издания, и зачем нужно его читать, если вы уже знакомы с предыдущим? Во-первых, в это издание включены сведения, относящиеся к Oracle8, например объектные типы и внешние процедуры.

Во-вторых, эта книга полезна и тогда, когда не используется Oracle8, так как в нее внесен ряд других важных изменений по сравнению с первой редакцией. Материал, изложенный в первом издании, здесь реорганизован таким образом, что различные конструкции PL/SQL представлены более наглядно. Кроме того, в новую редакцию включены примеры рубрики "PL/SQL в работе", а также прилагаемый компакт-диск.

### PL/SQL в работе

Многие читатели, возможно, собираются использовать PL/SQL для разработки реальных приложений, что естественно, так как данный язык создан именно для этого. Чтобы облегчить решение подобной задачи, автор включил в книгу множество примеров, объединенных под общей рубрикой "PL/SQL в работе". Их можно применять в собственных приложениях в том виде, в каком они представлены, либо использовать в качестве отправной точки при разработке приложений. Идеи, лежащие в основе этих примеров, были предложены в основном вами — читателями первого издания, — поэтому они будут вполне применимы во время реального программирования на PL/SQL.

### Прилагаемый компакт-диск

Исходный текст программ, использовавшихся в первом издании, можно было перегрузить с web-узла Oracle Press **www.osborne.com/oracle.** Данные программы по-прежнему доступны на этом узле и, кроме того, они находятся на компакт-диске, прилагаемом к книге. Помимо всех примеров, на этом компакт-диске содержатся также демонстрационные версии двух программных продуктов, использующих PL/SQL: Oracle WebServer корпорации Oracle и SQL-Station компании Platinum, которые также обсуждаются в тексте. Воспользуйтесь этими средствами: они могут быть достаточно полезны. Более подробную информацию об SQL-Station можно найти на web-узле Platinum **www.platinum.com,** а информацию об Oracle WebServer — на web-узле Oracle **www.oracle.com.**

## Назначение данной книги

Эту книгу можно считать и руководством пользователя, и справочным пособием по PL/SQL. Она полезна как опытным программистам, которых интересуют лишь синтаксис PL/SQL и его новые средства, так и программистам-новичкам, которые не знакомы с другими языками программирования третьего поколения. Рекомендуем читателю перед началом работы с данной книгой ознакомиться с общими принципами Oracle (соединение с базой данных и ее использование, базовый SQL и т.д.).

## Работа с данной книгой

Книга поделена на 22 главы и 4 приложения. Глава 1 — это введение, а в главах со 2 по 12 описываются синтаксис и семантика языка PL/SQL. Кроме того, в главе 12 говорится о функционировании и настройке программ. В главах 13 и 14 обсуждаются среда выполнения программ и методы их отладки, а в главах с 15 по 21 — новые возможности этого языка, в том числе и встроенные программные модули. Приложения служат справочниками по различному изложенному в книге материалу.

### Глава 1. Введение в PL/SQL

Первая глава дает общее понятие о PL/SQL и описывает некоторые важнейшие средства этого языка. Кроме того, здесь говорится о версиях PL/SQL и о том, каким версиям системы баз данных они соответствуют. Глава завершается описанием схемы базы данных, используемой в качестве примера на протяжении всей книги.

### Глава 2. Основы PL/SQL

В этой главе описывается синтаксис PL/SQL. Здесь обсуждаются: структура программ, переменных и типов, выражений и операторов PL/SQL, а также управляющие структуры (циклы и условные операторы). Глава завершается рекомендациями по стилю программирования на PL/SQL и советами, как составлять более удобочитаемые и простые в использовании тексты программ.

### Глава 3. Записи и таблицы

Записи и таблицы — это типы данных, которые определяются пользователями и применяются в PL/SQL версии 2 и Oracle7. С помощью записей можно управлять несколькими связанными переменными как одной единицей, а таблицы позволяют организовать доступ к информации как к массивам данных. В главе 3 поясняется, как использовать эти типы данных.

### Глава 4. SQL в PL/SQL

В данной главе рассмотрены SQL-операторы, применяемые в PL/SQL, — команды манипулирования данными Oracle. Здесь описаны также привилегии и управление транзакциями.

### Глава 5. Встроенные SQL-функции

Здесь дается представление о встроенных SQL-функциях, доступных в PL/SQL.

### Глава 6. Курсоры

В этой главе подробно объясняется, что такое курсоры и как их использовать для работы с большими объемами данных. Здесь обсуждается синтаксис, применяемый при объявлении и использовании курсоров, дано описание атрибутов курсоров и приведены примеры применения курсорных переменных.

### Глава 7. Подпрограммы: процедуры и функции

В главах с 7 по 9 рассматриваются четыре различных вида именованных блоков PL/SQL. Обсуждение начинается с главы 7, где говорится о процедурах и функциях, о синтаксисе и назначении каждой из них, а также о различиях между ними. Кроме того, в этой главе подробно изложены способы использования функций в SQL-операторах, взаимодействие ролей и процедур, а также связь между хранимыми подпрограммами и словарем данных.

### Глава 8. Модули

Модуль позволяет группировать связанные подпрограммы в виде одной программной единицы. Многие новые средства PL/SQL реализованы как модули, способствующие разработке более совершенных приложений через абстрактное представление данных.

### Глава 9. Триггеры

Триггеры — это последний тип именованных блоков PL/SQL. Они активизируются автоматически при модификации данных Oracle и поэтому позволяют реализовывать более сложные бизнес-процессы, которые не могут быть заданы при помощи ограничений ссылочной целостности.

### Глава 10. Обработка ошибок

Обработка ошибок является важнейшим элементом любого грамотно спроектированного приложения. В этой главе даются пояснения, как в PL/SQL использовать исключительные ситуации, чтобы гарантировать надежность программы и обеспечить возможность обработки ошибок в процессе ее выполнения. Кроме того, здесь приводятся рекомендации по эффективному использованию исключительных ситуаций.

## Глава 11. Объекты

PL/SQL 8.0 ... и ВЫШЕ

Объектные типы, введенные в Oracle8, позволяют применять различные методы программирования. В этой главе анализируется работа модели объектно-реляционной базы данных Oracle8 и создание объектных типов и методов.

## Глава 12. Сборные конструкции

PL/SQL 8.0 ... и ВЫШЕ

В этой главе обсуждаются сборные конструкции, в число которых входят вложенные таблицы и изменяемые массивы. Эти новые типы данных, применяемые в Oracle8, расширяют функциональные возможности таблиц PL/SQL, о которых говорится в главе 2. В данной главе рассказывается о том, как использовать эти типы, в том числе и о методах сборных конструкций.

## Глава 13. Среды выполнения программ PL/SQL

PL/SQL может работать в самых различных средах. В этой главе сравнивается выполнение программ PL/SQL на станциях клиентов и на сервере, а также приводится детальная информация об использовании PL/SQL: в SQL*Plus, предкомпиляторах Oracle, OCI, наборе инструментальных средств Developer 2000 и в средствах, предлагаемых третьими фирмами, например SQL-Station.

## Глава 14. Тестирование и отладка

В данной главе описаны различные методы отладки приложений PL/SQL, в том числе отладчик SQL-Station Debugger. Эти методы проиллюстрированы на примере разрешения трех часто встречающихся в PL/SQL ситуаций. Они применимы при разработке различных приложений пользователей. Глава завершается обсуждением процесса эффективной разработки программ PL/SQL.

## Глава 15. Динамический PL/SQL

Динамический PL/SQL – это весьма эффективный метод программирования, позволяющий создавать очень гибкие программы. В этой главе представлен модуль DBMS_SQL, с помощью которого реализуется динамический PL/SQL. Этот модуль может быть использован также для преодоления ограничения, которое разрешает применение в PL/SQL только операторов DML.

## Глава 16. Взаимодействие между соединениями

В этой главе обсуждаются два встроенных модуля, применяемые для непосредственного взаимодействия между сеансами работы с базой данных, – программные каналы базы данных (DBMS_PIPES) и оповещения базы данных (DBMS_ALERT). Здесь приведен ряд примеров, а также дается сравнение этих двух модулей.

## Глава 17. Улучшенная организация очередей Oracle

PL/SQL 8.0 ... и ВЫШЕ

Средство Oracle/AQ (Advanced Queuing – улучшенная организация очередей) реализует надежную систему организации очередей, подобную той, которая применяется в мониторах обработки транзакций. В этой главе говорится об использовании Oracle/AQ и приведен исчерпывающий пример.

## Глава 18. Задания для баз данных и файловый ввод/вывод

С помощью модуля DBMS_JOB можно планировать задания PL/SQL (в форме хранимых процедур) так, чтобы они выполнялись автоматически в определенное время. Модуль UTL_FILE позволяет при помощи PL/SQL считывать и записывать файлы средствами операционной системы . Оба модуля обсуждаются достаточно подробно, с анализом примеров.

## Глава 19. Программа Oracle WebServer

В этой главе объясняется, как PL/SQL встраивается в среду web-сервера Oracle и как можно генерировать выходные данные HTML из хранимых процедур PL/SQL. Такое использование PL/SQL позволяет создавать динамические web-страницы из информации, хранящейся в базе данных.

## Глава 20. Внешние процедуры

PL/SQL 8.0 ... и ВЫШЕ

В PL/SQL версии 8 (с Oracle8) можно непосредственно вызывать процедуры и функции, написанные на языке C. Это ценное свойство, расширяющее возможности PL/SQL и позволяющее использовать все функциональные средства языка программирования C, является предметом данной главы, как и установление соответствия типов данных PL/SQL и C.

## Глава 21. Большие объекты

PL/SQL 8.0 ... и ВЫШЕ

Большие объекты (LOB – Large Objects) могут содержать до 4-х Гбайт текстовых или двоичных данных. В Oracle8 большими объектами можно управлять с помощью модуля DBMS_LOB, описанного в этой главе. DBMS_LOB позволяет произвольным образом считывать и записывать информацию больших объектов, что намного удобнее, чем при использовании типов данных LONG и LONG RAW, применяемых в Oracle7. Кроме того, при желании можно хранить двоичные данные вне базы данных.

## Глава 22. Производительность и настройка

Правильно написанная программа PL/SQL должна не только выдавать нужный результат, но и делать это максимально эффективно. В этой главе обсуждаются методы повышения производительности и настройки программ, в том числе использование разделяемого пула, настройка SQL-операторов и работа с интерфейсом массивов Oracle.

## Приложение A. Зарезервированные слова PL/SQL

В приложении A приведен список слов, зарезервированных для PL/SQL и для собственно базы данных.

## Приложение B. Руководство по работе со встроенными модулями DBMS

В приложении B описаны все встроенные модули, применяемые в PL/SQL. С их помощью реализуются такие средства, как файловый ввод/вывод, планирование заданий, взаимодействие соединений, динамическое программирование и управление разделяемым пулом.

## Приложение C. Глоссарий средств PL/SQL

Приложение C представляет собой справочный список средств PL/SQL, для удобства составленный в алфавитном порядке. Здесь дается краткое описание каждого пункта, а также приводится ссылка на главу, в которой подробно описано конкретное средство.

## Приложение D. Словарь данных

В приложении D описаны представления словаря данных, полезные для программистов, работающих на PL/SQL.

# Глава 1

## Введение в PL/SQL

PL/SQL – это мощный язык программирования, используемый для обращения к базам данных Oracle из различных сред. PL/SQL интегрирован с сервером базы данных, поэтому его программы могут быть выполнены быстро и эффективно. Этот язык применяется также в некоторых клиентских инструментальных средствах Oracle. В данной главе рассмотрены причины использования PL/SQL и его основные свойства, а также различные версии этого языка и систем баз данных.

# Почему PL/SQL?

Oracle – это реляционная база данных. Язык программирования, применяемый для обращения к реляционным базам данных, называется *языком структурированных запросов (SQL – Structured Query Language;* часто произносится как "сикуэл"). SQL – это гибкий и эффективный язык, все средства которого применяются для манипулирования реляционными данными и для их исследования. Например, ниже показано, как с помощью SQL-оператора можно удалить из базы данных сведения обо всех студентах (students), профилирующей дисциплиной (major) которых является питание (nutrition):

```
DELETE FROM students
WHERE major = 'Nutrition';
```

(*Таблицы, используемые в этой книге, в том числе и таблица* **students**, *описаны в конце данной главы.*)

SQL является *языком четвертого поколения (4GL – fourth-generation language).* Это означает, что данный язык описывает то, что нужно выполнить, но не как это должно быть сделано. Например, в приведенном выше операторе DELETE (удалить) неизвестно, как база данных в действительности определит студентов, специализирующихся по питанию. Чтобы узнать, какие записи необходимо удалить, сервер, скорее всего, просмотрит сведения обо всех студентах, однако реальную картину этой операции пользователи не увидят.

*Языки третьего поколения,* например C или COBOL, по природе своей более процедурны. Программа, составленная на языке третьего поколения (3GL – third-generation language), реализуется путем выполнения пошагового алгоритма. Например, можно выполнить операцию DELETE таким образом:

```
LOOP (циклически просмотреть) запись о каждом студенте
  IF (если) для этой записи major = 'Nutrition' THEN (тогда)
    DELETE (удалить) эту запись;
  END IF (конец если);
END LOOP (конец цикла);
```

У каждого языка программирования есть свои достоинства и недостатки. Языки четвертого поколения, подобные SQL, как правило, достаточно просты (по сравнению с языками третьего поколения) и содержат меньшее число команд. Кроме того, они изолируют пользователя от базовых структур данных и алгоритмов. Однако в некоторых случаях процедурные конструкции языков 3GL полезны для более точного описания программ. Именно для этого применяется PL/SQL, который объединяет мощь и гибкость SQL (языка 4GL) и процедурные конструкции языка 3GL.

PL/SQL означает Procedural Language/SQL (процедурный язык/SQL). Как видно из его названия, PL/SQL расширяет возможности SQL, добавляя в него конструкции процедурных языков, такие как:

- Переменные и типы (как предварительно определенные, так и определяемые пользователями)
- Управляющие структуры, такие как операторы и циклы IF-THEN-ELSE
- Процедуры и функции
- Объектные типы и методы (PL/SQL версии 8 и выше)

Процедурные конструкции полностью объединяются с Oracle SQL, что дает в результате структурированный и эффективный язык программирования. Предположим, что требуется изменить профилирующую дисциплину некоего студента. Если этот студент не существует, нужно создать новую запись. Это можно сделать при помощи следующей программы PL/SQL:

```
-- Этот пример содержится в файле 3gl_4gl.sql.
DECLARE
  /* Объявим переменные, которые будут использоваться в SQL-операторах. */
  v_NewMajor VARCHAR2(10) := 'History';
  v_FirstName VARCHAR2(10) := 'Scott';
  v_LastName VARCHAR2(10) := 'Urman';
BEGIN
```

```
    /* Обновим таблицу students. */
    UPDATE students
        SET major = v_NewMajor
        WHERE first_name = v_FirstName
        AND last_name = v_LastName;
    /* Проверим, найдена ли запись. Если запись отсутствует,
        нужно ее ввести. */
    IF SQL%NOTFOUND THEN
        INSERT INTO students (ID, first_name, last_name, major)
          VALUES (student_sequence.NEXTVAL, v_FirstName, v_LastName,
          v_NewMajor);
    END IF;
END;
```

В этом примере содержатся два разных SQL-оператора (UPDATE (обновить) и INSERT (ввести)), а также ряд операторов объявления переменных и условный оператор IF (если).

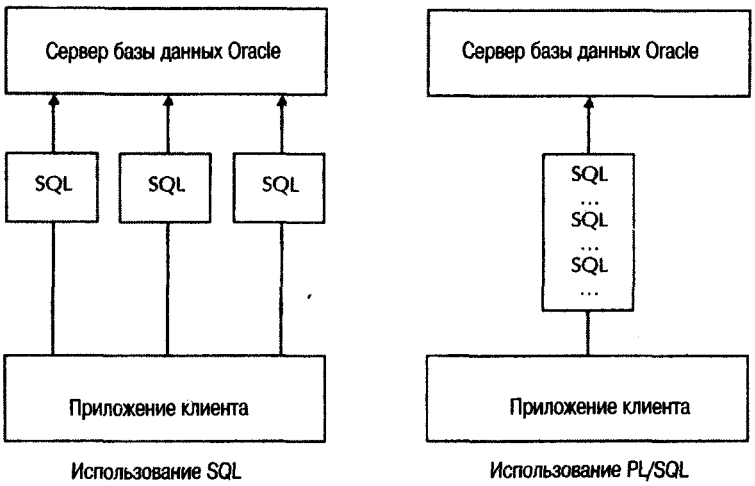
**▼ ВНИМАНИЕ** Чтобы выполнить приведенный пример, необходимо вначале создать объекты базы данных, на которые производятся ссылки (таблицу **students** и последовательность **student_sequence**). Это может быть сделано с помощью сценария **tables.sql**, являющегося частью оперативной документации. Более подробно о создании таких объектов и о поставляемых текстах программ рассказано в разделе "Содержимое компакт-диска" ниже в этой главе.

PL/SQL уникален тем, что соединяет гибкость SQL с мощью и способностью к конфигурированию языка 3GL. В нем имеются как необходимые процедурные конструкции, так и возможность обращения к базе данных. Таким образом, этот язык программирования является надежным, эффективным языком, хорошо подходящим для разработки сложных приложений.

## Модель клиент/сервер

Многие приложения для работы с базами данных создаются с использованием модели клиент/сервер. Сама программа размещается на компьютере клиента и посылает запросы на получение информации серверу базы данных. Запросы инициируются при помощи SQL, что, как правило, приводит к наличию в сети большого числа посылок — по одной на каждый SQL-оператор. Эта ситуация иллюстрируется в левой половине рис. 1.1. Теперь обратимся к его правой половине. Несколько SQL-операторов могут быть объединены в единый блок PL/SQL и посланы серверу как единое целое. В результате сетевой трафик снижается, а приложение функционирует намного быстрее.

**Рис. 1.1.**
*PL/SQL в среде клиент/сервер*



Даже если клиент и сервер функционируют на одном и том же компьютере, производительность системы повышается. При этом сеть отсутствует, но пакетирование SQL-операторов тем не менее ведет к упрощению программы, которой теперь приходится реже обращаться к базе данных.

## Стандарты

В Oracle язык SQL соответствует стандарту ANSI (American National Standards Institute – Американский национальный институт стандартов), определенному документом ANSI X3.135-1992 "Database Language SQL". Этот стандарт, известный как SQL92 (или SQL2), определяет только язык SQL и не описывает его 3GL-расширений, которые предлагаются в PL/SQL. Стандарт SQL92 имеет три уровня согласования: Entry (начальный), Intermediate (промежуточный) и Full (полный). Oracle7 варианта 7.2 (и все другие версии более высокого уровня, в том числе и Oracle8) соответствует стандартам Entry SQL92, что одобрено NIST (National Institute for Standards and Technology – Национальный институт стандартов и технологий). В настоящее время Oracle работает с ANSI, чтобы обеспечить соответствие будущих версий Oracle и PL/SQL полному стандарту.

# Средства PL/SQL

Различные средства и возможности PL/SQL лучше проиллюстрировать на примерах. Ниже описаны некоторые из самых важных средств этого языка. Полный их анализ будет проводиться на протяжении всей книги.

## Блочная структура

Базовой единицей PL/SQL является блок (block). Все программы PL/SQL состоят из блоков, которые могут быть вложены один в другой. Как правило, в программе каждый блок выполняет определенную логическую единицу работы, что помогает отделить различные задачи. Блок имеет следующую структуру:

```
DECLARE
   /* Раздел объявлений – переменные, типы, курсоры и логические
      подпрограммы PL/SQL. */
BEGIN
   /* Выполняемый раздел – процедурные и SQL-операторы. Это
      основной раздел блока и единственный, являющийся обязательным. */
EXCEPTION
   /* Раздел исключительных ситуаций – операторы обработки ошибок. */
END;
```

Обязателен только выполняемый раздел; разделы объявлений и исключительных ситуаций необязательны. Кроме того, выполняемый раздел должен содержать, по меньшей мере, один выполняемый оператор. Разные разделы блока в программе PL/SQL предназначены для выполнения различных функций.

Язык программирования PL/SQL разработан на основе языка третьего поколения Ada. Многие конструкции, применяемые в Ada, можно найти также и в PL/SQL. Одним из общих свойств этих языков является их блочная структура. Другие свойства Ada, которые являются частью и PL/SQL, – это обработка исключительных ситуаций, синтаксис объявления процедур и функций, а также модули. На протяжении всей этой книги будут приводиться примеры, иллюстрирующие схожесть языков программирования Ada и PL/SQL.

## Обработка ошибок

Раздел исключительных ситуаций используется для реагирования на ошибки, встречающиеся во время выполнения программы. При отделении программного текста, предназначенного для обработки ошибок, от основного тела программы структура всей программы становится гораздо понятнее. Для примера рассмотрим блок PL/SQL, в котором имеется раздел исключительных ситуаций, регистрирующий ошибку, текущее время и пользователя, обнаружившего эту ошибку:

```
-- Этот пример содержится в файле error.sql.
DECLARE
  v_ErrorCode  NUMBER; – код ошибки
  v_ErrorMsg  VARCHAR2(200); – сообщение об ошибке
```

```
    v_CurrentUser  VARCHAR2(8);  -  текущий пользователь базы данных
    v_Information  VARCHAR2(100);  -  информация об ошибке
BEGIN
    /* Здесь обрабатываются некоторые данные. */
EXCEPTION
    WHEN OTHERS THEN
        -- С помощью встроенных функций присвоим некоторые значения
        -- переменным регистрации.
        v_ErrorCode := SQLCODE;
        v_ErrorMsg := SQLERRM;
        v_CurrentUser := USER;
        v_Information := 'Error encountered on ' ||
          TO_CHAR(SYSDATE) || ' by database user ' || v_CurrentUser;
        -- Внесем сообщение об ошибке в таблицу регистрации log_table.
        INSERT INTO log_table (code, message, info)
          VALUES (v_ErrorCode, v_ErrorMsg, v_Information);
END;
```

**▼ ВНИМАНИЕ** Приведенный пример, как и многие другие, используемые в этой книге, можно найти в оперативной документации. Более подробно об этом рассказано в разделе "Содержимое компакт-диска" в конце этой главы.

# Переменные и типы

Информация, определяемая PL/SQL, передается базе данных и обратно при помощи переменных. Переменная (variable) — это область памяти, которая может быть считана программой или присвоена чему-либо. В предыдущем примере **v_CurrentUser, v_ErrorCode** и **v_Information** — это переменные. Переменные описываются в разделе объявлений блока.

Каждой переменной назначается конкретный тип (type). Он определяет вид информации, которая может храниться в данной переменной. Переменные PL/SQL могут иметь тот же тип, что и столбцы таблиц базы данных:

```
DECLARE
    v_StudentName  VARCHAR2(20);
    v_CurrentDate  DATE;
    v_NumberCredits  NUMBER(3);
```

или быть другого, дополнительного типа:

```
DECLARE
    v_LoopCounter  BINARY_INTEGER;
    v_CurrentlyRegistered  BOOLEAN;
```

В PL/SQL также поддерживаются типы, определяемые пользователями: таблицы и записи. Такие типы дают возможность описывать собственную структуру данных, с которыми работает программа:

```
DECLARE
    TYPE  t_StudentRecord  IS  RECORD  (
        FirstName  VARCHAR2(10),
        LastName  VARCHAR2(10),
        CurrentCredits  NUMBER(3)
    );
    v_Student  t_StudentRecord;
```

**PL/SQL 8.0 ... и ВЫШЕ** *Кроме того, в Oracle8 (с PL/SQL 8) поддерживаются объектные типы, которые имеют атрибуты и методы и могут храниться в таблицах базы данных.* В следующем примере создается объектный тип:

```
-- Этот пример содержится в файле tables8.sql.
CREATE OR REPLACE TYPE StudentObj AS OBJECT (
  ID                NUMBER(5),
  First_name        VARCHAR2(20),
  Last_name         VARCHAR2(20),
  Major             VARCHAR2(30),
  Current_credits   NUMBER(3),

  -- Возвращаются имена и фамилии, разделенные пробелами.
  MEMBER FUNCTION FormattedName
     RETURN VARCHAR2,
  PRAGMA RESTRICT_REFERENCES(FormattedName, RNDS, WNDS, RNPS, WNPS),

  -- Профилирующая дисциплина заменяется в p_NewMajor указанным
  -- значением.
  MEMBER PROCEDURE ChangeMajor(p_NewMajor IN VARCHAR2),
  PRAGMA RESTRICT_REFERENCES(ChangeMajor, RNDS, WNDS, RNPS, WNPS),

  -- Текущие зачеты (current_credits) изменяются при добавлении числа
  -- зачетов к текущему значению в p_CompletedClass.
  MEMBER PROCEDURE UpdateCredits(p_CompletedClass IN ClassObj),
  PRAGMA RESTRICT_REFERENCES(UpdateCredits, RNDS, WNDS, RNPS, WNPS)
);
```

## Циклические конструкции

В PL/SQL поддерживаются различные виды циклов. *Цикл (loop)* позволяет неоднократно выполнять одну и ту же последовательность операторов. Например, ниже приведен блок, в котором для ввода цифр от 1 до 50 в таблицу temp_table используется *простой цикл*:

```
-- Этот пример содержится в файле simple.sql.
DECLARE
  v_LoopCounter BINARY_INTEGER := 1;
BEGIN
  LOOP
     INSERT INTO temp_table (num_col)
       VALUES (v_LoopCounter);
     v_LoopCounter := v_LoopCounter + 1;
     EXIT WHEN v_LoopCounter > 50;
  END LOOP;
END;
```

Другим типом цикла является *числовой цикл FOR*. Эта циклическая конструкция еще более упрощает синтаксис программы. Можно выполнить операции, описанные в предыдущем примере, следующим образом:

```
-- Этот пример содержится в файле numeric.sql.
BEGIN
  FOR v_LoopCounter IN 1..50 LOOP
     INSERT INTO temp_table (num_col)
       VALUES (v_LoopCounter);
  END LOOP;
END;
```

## Курсоры

*Курсор (cursor)* используется для обработки нескольких строк, считываемых в базе данных с помощью оператора SELECT (выбрать). Посредством курсора программа может по очереди обрабатывать строки из возвращаемого набора. Например, ниже показано, как можно считать в базе данных имена и фамилии всех студентов:

```
-- Этот пример содержится в файле cursor.sql.
DECLARE
   v_FirstName  VARCHAR2(20);
   v_LastName  VARCHAR2(20);
   -- Объявление курсора. Задаем SQL-оператор, возвращающий нужные строки.
   CURSOR  c_Students  IS
       SELECT  first_name,  last_name
          FROM  students;
BEGIN
   -- Начинаем обработку курсора.
   OPEN  c_Students;
   LOOP
       -- Считываем одну строку.
       FETCH  c_Students  INTO  v_FirstName,  v_LastName;
       -- Выходим из цикла после того, как считаны все строки.
       EXIT  WHEN  c_Students%NOTFOUND;
       /* Обрабатываем данные. */
   END  LOOP;
   -- Заканчиваем обработку.
   CLOSE  c_Students;
END;
```

# Соглашения, принятые в этой книге

Здесь обсуждаются некоторые соглашения, принятые в оставшейся части книги. В их число входят значки, демонстрирующие различия между версиями PL/SQL, ссылки на документацию Oracle и местонахождение примеров, являющихся частью оперативной документации.

## PL/SQL и версии Oracle

PL/SQL содержится внутри Oracle-сервера. Первая версия PL/SQL, называемая 1.0, была создана вместе с Oracle версии 6. Oracle7 содержит PL/SQL 2.0. В Oracle8 номер версии PL/SQL был изменен на 8. Каждая последующая версия базы данных содержит PL/SQL соответствующей версии, как показано в таблице 1.1. В эту таблицу также включено описание основных свойств каждой версии. В данной книге рассмотрены версии PL/SQL с 2.0 до 8.0. Средства, применяемые только в конкретных версиях, выделены значками:

**PL/SQL 2.1 ... и ВЫШЕ**

В этом параграфе обсуждается средство, доступное в PL/SQL 2.1 и выше, — модуль DBMS_SQL.

**PL/SQL 2.2 ... и ВЫШЕ**

В этом параграфе обсуждается средство, доступное в PL/SQL 2.2 и выше, — курсорные переменные.

**PL/SQL 2.3 ... и ВЫШЕ**

В этом параграфе обсуждается средство, доступное в PL/SQL 2.3 и выше, — модуль UTL_FILE.

**PL/SQL 8.0 ... и ВЫШЕ**

В этом параграфе обсуждается средство, доступное в PL/SQL 8.0 и выше, — объектные типы.

## ТАБЛИЦА 1.1. Версии Oracle и PL/SQL

| Версия Oracle | Версия PL/SQL | Добавленные или измененные средства |
|---|---|---|
| 6 | 1.0 | Первоначальная версия |
| 7.0 | 2.0 | Тип данных CHAR стал иметь фиксированную длину<br>Подпрограммы (процедуры, функции, модули и триггеры)<br>Составные типы данных, определяемые пользователями: таблицы и записи<br>Взаимодействие между соединениями с помощью модулей DBMS_PIPE и DBMS_ALERT<br>Передача выходных данных в SQL*Plus или Server Manager с помощью модуля DBMS_OUTPUT |
| 7.1 | 2.1 | Подтипы, определяемые пользователями<br>Возможность применения в SQL-операторах функций, определяемых пользователями<br>Динамический PL/SQL с помощью модуля DBMS_SQL |
| 7.2 | 2.2 | Курсорные переменные<br>Ограниченные подтипы, определяемые пользователями<br>Возможность планирования обработки пакетов PL/SQL при помощи модуля DBMS_JOB |
| 7.3 | 2.3 | Усовершенствованные курсорные переменные (разрешен выбор данных на сервере)<br>Файловый ввод/вывод при помощи модуля UTL_FILE<br>Атрибуты таблиц PL/SQL и таблицы записей<br>Триггеры, хранимые в скомпилированном виде |
| 8.0 | 8.0 | Объектные типы и методы<br>Типы сборных конструкций: вложенные таблицы и изменяемые массивы<br>Средство Advanced Queuing (усовершенствованная организация очередей)<br>Внешние процедуры<br>Усовершенствованные LOB (большие объекты) |

Важно знать версию применяемого PL/SQL, чтобы полностью использовать преимущества программных средств. При соединении с базой данных в начальных строках будет содержаться ее версия. Например, приведенные ниже две группы начальных строк вполне корректны:

```
Connected to:
Personal Oracle7 Release 7.3.2.1.1 - Production Release
With the distributed and replication options
PL/SQL Release 2.3.2.0.0 - Production
```

и

```
Connected to:
Oracle8 Server Realease 8.0.3.0.0 - Production
With the distributed, heterogeneous, replication, objects,
parallel query and Spatial Data options
```

```
PL/SQL Release 8.0.3.0.0 - Production
```

*Обратите внимание, что версия PL/SQL соответствует версии базы данных.*

Большинство примеров, приведенных в этой книге, было выполнено в Personal Oracle 7.3.2.1.1, работающей под Microsoft Windows 95. Примеры для PL/SQL 8.0 были выполнены с базой данных Oracle8 в системе Unix. Все рисунки с изображением различных экранов получены в Windows с системой Personal Oracle, работающей в качестве сервера базы данных.

## Документация Oracle

Во многих разделах этой книги автор ссылается на документацию Oracle, в которой приводится более подробная информация. Названия различных руководств меняются с версиями, поэтому, как правило, используется обобщенный вариант названия. Например, Oracle Server Reference — это как Oracle7 Server Reference, так и Oracle8 Server Reference, в зависимости от используемой версии Oracle.

### Содержимое компакт-диска

*На компакт-диске, прилагаемом к этой книге, содержится информация трех видов:*

1. Демонстрационная версия программного средства SQL-Station компании Platinum, работающая в среде Windows 95 или NT. SQL-Station — это среда разработки программ PL/SQL, в состав которой входит *отладчик программ PL/SQL (она описана в главах 13, 14 и 22).*
2. Демонстрационная версия программы Oracle WebServer. На компакт-диске содержатся ее версии как для Solaris, так и для NT. В WebServer широко применяется PL/SQL (эта программа описана в главе 19).
3. Текст примеров, используемых в этой книге, можно найти на web-странице Osborne's Oracle Press **www.osborne.com/oracle.**

Более подробно об информации, содержащейся на компакт-диске, рассказано в файле README, который находится в корневом каталоге этого диска.

### Местонахождение примеров

Имя файла, в котором находится пример на компакт-диске, указано в комментарии в первой строке примера. Все примеры находятся на компакт-диске (и на web-странице) в каталоге под названием code, причем подкаталоги этого каталога соответствуют номерам глав. Продемонстрируем это на примере цикла, который был рассмотрен выше:

```
-- Этот пример содержится в файле simple.sql.
DECLARE
v_LoopCounter BINARY_INTEGER := 1;
BEGIN
LOOP
INSERT INTO temp_table (num_col)
VALUES (v_LoopCounter);
v_LoopCounter := v_LoopCounter + 1;
EXIT WHEN v_LoopCounter > 50;
END LOOP;
END;
```

Этот пример можно найти в файле code/ch01/simple.sql. В файле code/readme.txt описаны все примеры.

# Примеры таблиц

Во всех примерах, приведенных в этой книге, используется общий набор таблиц базы данных, с помощью которых реализуется система регистрации студентов некоторого колледжа. Существует три основных таблицы: **students (студенты), classes (учебные_группы) и rooms (аудитории).** В них содержатся главные элементы, необходимые для системы. В дополнение к основным таблицам в таблице **registered_students (зарегистрированные_студенты)** имеется информация о студентах, которые записались в какие-либо учебные группы. Далее подробно описана структура этих таблиц и указаны SQL-операторы, необходимые для их создания.

▼ ВНИМАНИЕ Все эти таблицы можно создать при помощи SQL-сценария **tables.sql** (имеется на компакт-диске). Таблицы для Oracle8 находятся в файле **tables8.sql.** Оба файла расположены в подкаталоге code/ch01. В этом разделе показаны таблицы только для Oracle7 (варианты таблиц для Oracle8 см. в **tables8.sql**).

## student_sequence

Последовательность **student_sequence** используется для генерирования уникальных значений первичного ключа таблицы students.

```
CREATE SEQUENCE student_sequence
START WITH 10000
INCREMENT BY 1;
```

## students

В таблице **students** содержится информация о студентах, посещающих колледж (здесь: id — идентификатор, first_name — имя, last_name — фамилия, major — профилирующая_дисциплина, current_credits — текущее_число_зачетов. — *Прим. пер.*).

```
CREATE TABLE students (
  Id              NUMBER(5) PRIMARY KEY,
  First_name      VARCHAR2(20),
  Last_name       VARCHAR2(20),
  Major           VARCHAR2(30),
  Current_credits NUMBER(3)
  );

INSERT INTO students (id, first_name, last_name, major,
                      current_credits)
  VALUES (student_sequence.NEXTVAL, 'Scott', 'Smith',
          'Computer Science', 0);

INSERT INTO students (id, first_name, last_name, major,
                      current_credits)
  VALUES (student_sequence.NEXTVAL, 'Margaret', 'Mason',
          'History', 0);

INSERT INTO students (id, first_name, last_name, major,
                      current_credits)
  VALUES (student_sequence.NEXTVAL, 'Joanne', 'Junebug',
          'Computer Science', 0);

INSERT INTO students (id, first_name, last_name, major,
                      Current_credits)

  VALUES (student_sequence.NEXTVAL, 'Manish', 'Murgratroid',
          'Economics', 0);

INSERT INTO students(id, first_name, last_name, major,
                     Current_credits)
  VALUES(student_sequence.NEXTVAL, 'Patrick', 'Poll',
        'History', 0);

INSERT INTO students(id, first_name, last_name, major,
                     current_credits)
  VALUES (student_sequence.NEXTVAL, 'Timothy', 'Taller',
```

```
            'History', 0);

INSERT INTO students(id, first_name, last_name, major,
                     current_credits)
  VALUES (student_sequence.NEXTVAL, 'Barbara', 'Blues',
          'Economics', 0);

INSERT INTO students(id, first_name, last_name, major,
                     current_credits)
  VALUES (student_sequence.NEXTVAL, 'David', 'Dinsmore',
          'Music', 0);

INSERT INTO students(id, first_name, last_name, major,
                     current_credits)
  VALUES (student_sequence.NEXTVAL, 'Ester', 'Elegant',
          'Nutrition', 0);

INSERT INTO students(id, first_name, last_name, major,
                     current_credits)
  VALUES (student_sequence.NEXTVAL, 'Rose', 'Riznit',
          'Music', 0);

INSERT INTO STUDENTS(id, first_name, last_name, major,
                     current_credits)
  VALUES (student_sequence.NEXTVAL, 'Rita', 'Razmataz',
          'Nutrition', 0);
```

## major_stats

В таблице **major_stats** хранятся статистические сведения о различных профилирующих дисциплинах (здесь: total_credits — общее_число_зачетов, total_students — общее_число_студентов. — *Прим. пер.*).

```
CREATE TABLE major_stats (
major VARCHAR2(30),
total_credits NUMBER,
total_students NUMBER);
```

## rooms

В таблице **rooms** хранится информация о существующих аудиториях (здесь: room_id — идентификатор_аудитории, building — здание, room_number — номер_аудитории, number_seats — число_мест, description — описание. — *Прим. пер.*).

```
CREATE TABLE rooms (
  Room_id        NUMBER(5) PRIMARY KEY,
  Building       VARCHAR2(15),
  Room_number    NUMBER(4),
  Number_seats   NUMBER(4),
  Description    VARCHAR2(50)
  );

INSERT INTO rooms
  (room_id, building, room_number, number_seats, description)
  VALUES (99999, 'Building 7', 310, 1000, 'Large Lecture Hall');

INSERT INTO rooms
(room_id, building, room_number, number_seats, description)
```

```
VALUES (99998, 'Building 6', 101, 500, 'Small Lecture Hall');

INSERT INTO rooms
(room_id, building, room_number, number_seats, description)
VALUES (99997, 'Building 6', 150, 50, 'Discussion Room A');

INSERT INTO rooms
(room_id, building, room_number, number_seats, description)
VALUES (99996, 'Building 6', 160, 50, 'Discussion Room B');

INSERT INTO rooms
( room_id, building, room_number, number_seats, description)
  VALUES (99995, 'Building 6', 170, 50, 'Discussion Room C');

INSERT INTO rooms
  (room_id, building, room_number, number_seats, description)
  VALUES (99994, 'Music Building', 100, 10, 'Music Practice Room');

INSERT INTO rooms
  (room_id, building, room_number, number_seats, description)
  VALUES (99993, 'Music Building', 200, 1000, 'Concert Room');

INSERT INTO rooms
  (room_id, building, room_number, number_seats, description)
  VALUES (99992, 'Building 7', 300, 75, 'Discussion Room D');

INSERT INTO rooms
  (room_id, building, room_number, number_seats, description)
  VALUES (99991, 'Building 7', 310, 50, 'Discussion Room E');
```

## classes

Таблица **classes** описывает курсы лекций, которые могут прослушать студенты (здесь: department — факультет, course — курс_лекций, max_students — максимальное_число_студентов, current_students — текущее_число_студентов, num_credits — число_зачетов. — *Прим. пер.*).

```
CREATE TABLE classes (
  Department          CHAR(3),
  Course              NUMBER(3),
  Description         VARCHAR2(2000),
  Max_students        NUMBER(3),
  Current_students    NUMBER(3),
  Num_credits         NUMBER(1),
  Room_id             NUMBER(5),
  CONSTRAINT classes_department_course
      PRIMARY KEY (department, course),
  CONSTRAINT classes_room_id
      FOREIGN KEY (room_id) REFERENCES rooms (room_id)
  );

INSERT INTO classes
  (department, course, description, max_students,
   current_students, num_credits, room_id)
  VALUES ('HIS', 101, 'History 101', 30, 0, 4, 99999);

INSERT INTO classes
  (department, course, description, max_students,
```

```
    current_students, num_credits, room_id)
  VALUES ('HIS', 301, 'History 301', 30, 0, 4, 99995);

INSERT INTO classes
  (department, course, description, max_students,
   current_students, num_credits, room_id)
  VALUES ('CS', 101, 'Computer Science 101', 50, 0, 4, 99998);

INSERT INTO classes
  (department, course, description, max_students,
   current_students, num_credits, room_id)
  VALUES ('ECN', 203, 'Economics 203', 15, 0, 3, 99997);

INSERT INTO classes
  (department, course, description, max_students,
   current_students, num_credits, room_id)
  VALUES ('CS', 102, 'Computer Science 102', 35, 0, 4, 99996);

INSERT INTO classes
  (department, course, description, max_students,
   current_students, num_credits, room_id)
  VALUES ('MUS', 410, 'Music 410', 5, 0, 3, 99994);

INSERT INTO classes
  (department, course, description, max_students,
   current_students, num_credits, room_id)
  VALUES ('ECN', 101, 'Economics 101', 50, 0, 4, 99992);

INSERT INTO classes
  (department, course, description, max_students,
   current_students, num_credits, room_id)
  VALUES ('NUT', 307, 'Nutrition 307', 20, 0, 4, 99991);
```

## registered_students

В таблице **registered_students** содержится информация о курсах лекций, которые студенты проходят в настоящее время (здесь: student_id — идентификатор_студента, grade — оценка. — *Прим. пер.*).

```
CREATE TABLE registered_students (
  Student_id  NUMBER(5) NOT NULL,
  Department  CHAR(3) NOT NULL,
  Course      NUMBER(3) NOT NULL,
  Grade       CHAR(1),
  CONSTRAINT rs_grade
      CHECK (grade IN ('A', 'B', 'C', 'D', 'E')),
  CONSTRAINT rs_student_id
      FOREIGN KEY (student_id) REFERENCES students (id),
  CONSTRAINT rs_department_course
      FOREIGN KEY (department, course)
      REFERENCES classes (department, course)
  );

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10000, 'CS', 102, 'A');
```

```
INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10002, 'CS', 102, 'B');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10003, 'CS', 102, 'C');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10000, 'HIS', 101, 'A');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10001, 'HIS', 101, 'B');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10002, 'HIS', 101, 'B');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10003, 'HIS', 101, 'A');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10004, 'HIS', 101, 'C');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10005, 'HIS', 101, 'C');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10006, 'HIS', 101, 'E');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10007, 'HIS', 101, 'B');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10008, 'HIS', 101, 'A');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10009, 'HIS', 101, 'D');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10010, 'HIS', 101, 'A');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10008, 'NUT', 307, 'A');
```

```
INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10010, 'NUT', 307, 'A');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10009, 'MUS', 410, 'B');

INSERT INTO registered_students
  (student_id, department, course, grade)
  VALUES (10006, 'MUS', 410, 'E');
```

## RS_audit

Таблица **RS_audit** используется для записи изменений, вносимых в таблицу registered_students (здесь: change_type — вид_изменения, changed_by — кем_изменено, timestamp — временная_метка, old_student_id — старый_идентификатор_студента, old_department — старый_факультет, old_course — старый_курс_лекций, old_grade — старая_оценка, new_student_id — новый_идентификатор_студента, new_department — новый_факультет, new_course — новый_курс_лекций, new_grade — новая_оценка. — *Прим. пер.*)

```
CREATE TABLE RS_audit (
  Change_type       CHAR(1) NOT NULL,
  Changed_by        VARCHAR2(8) NOT NULL,
  Timestamp         DATE NOT NULL,
  old_student_id    NUMBER(5),
  old_department    CHAR(3),
  Old_course        NUMBER(3),
  Old_grade         CHAR(1),
  new_student_id    NUMBER(5),
  new_department    CHAR(3),
  New_course        NUMBER(3),
  New_grade         CHAR(1)
  );
```

## log_table

Таблица **log_table** используется для записи ошибок Oracle (здесь: code — код, message — сообщение, info — информация. — *Прим. пер.*).

```
CREATE TABLE log_table (
  code NUMBER,
  message VARCHAR2(200),
  info VARCHAR2(100)
  );
```

## temp_table

Таблица **temp_table** используется для хранения временных данных, которые не относятся к другой информации (здесь: num_col — числовой_столбец, char_col — символьный_столбец. — *Прим. пер.*).

```
CREATE TABLE temp_table (
  num_col NUMBER,
  char_col VARCHAR2(60)
  );
```

## debug_table

Таблица **debug_table** используется модулем Debug (отладка), описанным в главе 14, для хранения отладочной информации PL/SQL (здесь: linecount — счетчик_строк, debug_str — отладочное_сообщение. — *Прим. пер.*).

```
CREATE TABLE debug_table (
   linecount  NUMBER,
   debug_str  VARCHAR2(100)
   );
```

# Итоги

В этой главе был дан общий обзор PL/SQL, в том числе назначения этого языка программирования и его основных свойств. Кроме того, было рассказано о версиях PL/SQL и базы данных, а также о том, как они соотносятся между собой. В заключение было дано описание прилагаемого компакт-диска и таблиц, используемых в этой книге в качестве примеров. В следующей главе мы начнем подробное обсуждение PL/SQL, его синтаксиса и программных конструкций.

# Глава 2

## Основы PL/SQL

Перед тем как перейти к обсуждению новых средств PL/SQL, необходимо проанализировать синтаксис, лежащий в основе этого языка. Синтаксические правила определяют способы создания конструкций любого языка программирования, в том числе и PL/SQL. В этой главе рассматриваются составные части блока PL/SQL, поясняются способы объявления переменных и их типы данных, приводятся базовые процедурные конструкции, а также дается краткий обзор курсоров и подпрограмм. Здесь также говорится о стиле программирования в PL/SQL и демонстрируются эффективные методы программирования, способствующие созданию элегантных и понятных текстов программ.

Все операторы PL/SQL являются либо процедурными, либо SQL-операторами (Следует различать два понятия: оператор (operator) – действие, которое может быть выполнено над одним или несколькими операндами для получения результата, и процедурный либо SQL-оператор (statement) – команда (группа команд), используемая для создания программ на языке PL/SQL. – *Прим. пер.*). В состав процедурных операторов входят объявления переменных, вызовы процедур и циклические конструкции. SQL-операторы используются для организации доступа к базам данных. Предметами рассмотрения этой главы и главы 3 являются процедурные операторы PL/SQL, а в главах 4, 5 и 6 анализируются SQL-операторы.

# Блок PL/SQL

Базовой единицей любой программы, написанной на PL/SQL, является блок. Из блоков состоят все программы PL/SQL, причем блоки могут следовать один за другим либо быть вложенными один в другой. Допустимы следующие виды блоков:

- *Анонимные блоки (anonymous blocks)* создаются, как правило, динамически и выполняются только один раз.
- *Именованные блоки (named blocks)* – это анонимные блоки с метками, дающими блокам имена. Они также создаются, как правило, динамически и выполняются только один раз.
- *Подпрограммы (subprograms)* – это процедуры, модули и функции, хранимые в базе данных. Эти блоки, как правило, не изменяются после своего создания и выполняются многократно явным образом посредством вызова процедуры, модуля или функции.
- *Триггеры (triggers)* – это именованные блоки, которые также хранятся в базе данных. Они тоже, как правило, не изменяются после своего создания и выполняются многократно неявным образом при наступлении соответствующих событий. Событием, вызывающим активизацию триггера, является оператор языка манипулирования данными (DML – data manipulation language), выполняемый над некоторой таблицей базы данных. Операторы DML – это INSERT (ввести), UPDATE (обновить) и DELETE (удалить).

Например, ниже приведен анонимный блок PL/SQL, с помощью которого в таблицу **temp_table** две строки вводятся, затем выбираются и отображаются на экране:

```
-- Этот пример содержится в файле anon.sql.
DECLARE
  /* Объявляем переменные, используемые в этом блоке. */
  V_Num1      NUMBER := 1;
  V_Num2      NUMBER := 2;
  V_String1   VARCHAR2(50) := 'Hello World!';
  V_String2   VARCHAR2(50) := '-- This message brought to youv by PL/SQL!';
  v_OutputStr VARCHAR2(50);
BEGIN
  /* Вначале с помощью значений переменных введем две строки в
     таблицу temp_table. */
  INSERT INTO temp_table (num_col, char_col)
     VALUES (v_Num1, v_String1);
  INSERT INTO temp_table (num_col, char_col)
     VALUES (v_Num2, v_String2);

  /* Теперь запросим в temp_table две только что введенные строки и
     при помощи модуля DBMS_OUTPUT выведем их на экран. */
  SELECT char_col
```

```
    INTO v_OutputStr
  FROM temp_table
  WHERE num_col = v_Num1;
  DBMS_OUTPUT.PUT_LINE(v_OutputStr);

  SELECT char_col
    INTO v_OutputStr
  FROM temp_table
  WHERE num_col = v_Num2;
  DBMS_OUTPUT.PUT_LINE(v_OutputStr);
END;
```

**▼ ВНИМАНИЕ** Для выполнения приведенного примера, как и большинства других примеров этой книги, необходимо создать несколько таблиц — в том числе таблицу **temp_table** — при помощи сценария tables.sql, который имеется на прилагаемом компакт-диске (на нем находится большинство примеров.) Имена нужных файлов указываются в начале каждого примера, а также могут быть определены с помощью файла README на компакт-диске. Более подробно о компакт-диске и его содержимом, в том числе и о таблицах, используемых в примерах, рассказано в главе 1. Модуль DBMS_OUTPUT описан в главе 14.

Чтобы дать блоку имя, нужно указать перед ключевым словом DECLARE метку, как показано в следующем примере. При желании можно разместить метку и после ключевого слова END. Метки будут рассмотрены более подробно ниже в этой главе.

```
-- Этот пример содержится в файле labeled.sql.
<<l_InsertIntoTemp>>
DECLARE
  /* Объявляем переменные, используемые в этом блоке. */
  V_Num1         NUMBER := 3;
  V_Num2         NUMBER := 4;
  V_String1      VARCHAR2(50) := 'Hello World!';
  V_String2      VARCHAR2(50) := '-- This message brought to you by
                                  PL/SQL!';
  v_OutputStr VARCHAR2(50);
BEGIN
  /* Вначале с помощью значений переменных введем две строки в
     таблицу temp_table. */
  INSERT INTO temp_table (num_col, char_col)
     VALUES (v_Num1, v_String1);
  INSERT INTO temp_table (num_col, char_col)
     VALUES (v_Num2, v_String2);

  /* Теперь запросим в temp_table две только что введенные строки и
     с помощью модуля DBMS_OUTPUT выведем их на экран. */
  SELECT char_col
     INTO v_OutputStr
  FROM temp_table
  WHERE num_col = v_Num1;
  DBMS_OUTPUT.PUT_LINE(v_OutputStr);

  SELECT char_col
     INTO v_OutputStr
  FROM temp_table
  WHERE num_col = v_Num2;
  DBMS_OUTPUT.PUT_LINE(v_OutputStr);
END l_InsertIntoTemp;
```

Можно превратить этот блок в хранимую процедуру, если заменить ключевое слово DECLARE на ключевые слова CREATE OR REPLACE PROCEDURE. (Более подробно процедуры обсуждаются в главе 7.) Обратите внимание, что имя процедуры указано после ключевого слова END.

```
-- Этот пример содержится в файле proc.sql.
CREATE OR REPLACE PROCEDURE InsertIntoTemp
  /* Объявляем переменные, используемые в этом блоке. */
  V_Num1      NUMBER := 5;
  V_Num2      NUMBER := 6;
  V_String1   VARCHAR2(50) := 'Hello World!';
  V_String2   VARCHAR2(50) := '-- This message brought to you by PL/SQL!';
  V_OutputStr   VARCHAR2(50);
BEGIN
  /* Вначале с помощью значений переменных введем две строки в
     таблицу temp_table. */
  INSERT INTO temp_table (num_col, char_col)
     VALUES (v_Num1, v_String1);
  INSERT INTO temp_table (num_col, char_col)
     VALUES (v_Num2, v_String2);

  /* Теперь запросим в temp_table две только что введенные строки и
     с помощью модуля DBMS_OUTPUT выведем их на экран. */
  SELECT char_col
     INTO v_OutputStr
  FROM temp_table
  WHERE num_col = v_Num1;
  DBMS_OUTPUT.PUT_LINE(v_OutputStr);

  SELECT char_col
     INTO v_OutputStr
  FROM temp_table
  WHERE num_col = v_Num2;
  DBMS_OUTPUT.PUT_LINE(v_OutputStr);
END InsertIntoTemp;
```

В заключение создадим для таблицы **temp_table** триггер, чтобы гарантировать ввод в столбец **num_col** только положительных значений. (Более подробно триггеры обсуждаются в главе 9.) Этот триггер будет вызываться всякий раз, когда в таблицу **temp_table** вводится новая строка или обновляется существующая.

```
-- Этот пример содержится в файле trigger.sql.
CREATE OR REPLACE TRIGGER OnlyPositive
  BEFORE INSERT OR UPDATE OF num_col
  ON temp_table
  FOR EACH ROW
BEGIN
  IF :new.num_col < 0 THEN
     RAISE_APPLICATION_ERROR(-20100, 'Please insert a positive value');
  END IF;
END OnlyPositive;
```

## Структура блока

Каждый блок состоит из трех различных разделов: раздела объявлений, выполняемого раздела и раздела исключительных ситуаций. Обязателен только выполняемый раздел; два других необязательны. Например, ниже приведен анонимный блок, состоящий из трех разделов:

```
-- Этот пример содержится в файле allthree.sql.
DECLARE
  /* Начало раздела объявлений. */
  v_StudentID NUMBER(5) := 10000;   -- Числовой переменной
                                    -- присваивается значение 10000
  v_FirstName VARCHAR2(20);         -- Максимальная длина
                                    -- последовательности символов
                                    -- для переменной равна 20
BEGIN
  /* Начало выполняемого раздела. */
  -- Выбираем имя студента, идентификатор которого равен 10000.
  SELECT first_name
      INTO v_FirstName
      FROM students
      WHERE id = v_StudentID;
EXCEPTION
  /* Начало раздела исключительных ситуаций. */
  WHEN NO_DATA_FOUND THEN
      -- Обрабатываем ошибку.
      INSERT INTO log_table (info)
        VALUES ('Student 10,000 does not exist!');
END;
```

**▼ ВНИМАНИЕ** В этом примере производятся ссылки на две дополнительные таблицы, которые создаются при помощи tables.sql, — **students** и **log_table.** Напомним, что полное описание схемы, используемой в качестве примера, находится в главе 1.

В разделе *объявлений* размещены все переменные, курсоры и типы, используемые данным блоком. В этом разделе могут быть объявлены также локальные процедуры и функции. Такие подпрограммы будут доступны только в пределах данного блока. Компоненты раздела объявлений более детально рассмотрены ниже в этой главе и в главе 3.

В *выполняемом* разделе указывается, какие операции выполняются в данном блоке. В этом разделе могут находиться как процедурные, так и SQL-операторы. Содержимое выполняемого раздела анализируется в главах 4, 5 и 6.

Ошибки обрабатываются в разделе *исключительных ситуаций*. Содержащийся в нем программный код не будет выполнен, если не произойдет ни одной ошибки. Исключительные ситуации, а также способы их использования для распознавания и обработки ошибок рассмотрены в главе 10.

Разделы блока ограничены ключевыми словами DECLARE (объявить), BEGIN (начало), EXCEPTION (исключительная ситуация) и END (конец). Кроме того, в конце блока необходимо ставить точку с запятой — это синтаксическое правило обязательно для блока. Таким образом, структура анонимного блока выглядит так:

```
DECLARE
  /* Раздел объявлений */
BEGIN
  /* Выполняемый раздел */
EXCEPTION
  /* Раздел исключительных ситуаций */
END;
```

**▼ ВНИМАНИЕ** При создании процедуры ключевое слово DECLARE необязательно. Более того, его использование будет ошибкой. Однако DECLARE требуется при создании триггера. Более подробно об этом рассказано в главах 7-9.

Если раздел объявлений отсутствует, выполнение блока начинается с ключевого слова BEGIN. Если отсутствует раздел исключительных ситуаций, ключевое слово EXCEPTION пропускается и блок заканчивается ключевым словом END с точкой с запятой. Таким образом, структура блока, состоящего только из выполняемого раздела, будет выглядеть так:

```
BEGIN
   /* Выполняемый раздел */
END;
```

а блок с разделом объявлений и выполняемым разделом, но без раздела исключительных ситуаций будет выглядеть следующим образом:

```
DECLARE
   /* Раздел объявлений */
BEGIN
   /* Выполняемый раздел */
END;
```

**▼ ВНИМАНИЕ** В приведенных блоках конструкции, заключенные в /* и */, являются комментариями. О комментариях более подробно рассказано в следующем разделе.

# Лексические единицы

Любая программа PL/SQL состоит из лексических единиц — строительных блоков языка. По сути дела, лексическая единица — это последовательность символов, являющихся частью набора, разрешенного в PL/SQL. В состав этого набора символов входят:

- Буквы верхнего и нижнего регистров: A-Z и a-z
- Цифры: 0-9
- Разделители: символы табуляции, пробелы и символы возврата каретки
- Математические символы: + – * / < > =
- Символы пунктуации: () {} [] ? ! ~ ; : . ' " @ # % $ ^ & _ |

В программе PL/SQL может быть использован любой символ, но только из этого набора. Подобно SQL, PL/SQL не учитывает регистр символов. Иначе говоря, буквы верхнего и нижнего регистров эквивалентны, за исключением строк символов, заключенных в кавычки.

Стандартный набор символов PL/SQL является частью набора символов ASCII. ASCII — это набор однобайтовых символов, т. е. каждый его символ может быть представлен в виде одного байта данных. При этом общее число символов ограничено 256. В Oracle поддерживаются и наборы многобайтовых символов, которые могут состоять более чем из 256 символов. Это необходимо для поддержки языков, не использующих английский алфавит. Подробный анализ многобайтовых символов не является предметом данной книги — если нужна более детальная информация, обратитесь к документации по Oracle.

Лексические единицы подразделяются на идентификаторы, ограничители, литералы и комментарии.

## Идентификаторы

*Идентификаторы* используются для именования объектов PL/SQL, таких как переменные, курсоры, типы и подпрограммы. Идентификатор начинается с буквы, за которой может следовать любая последовательность символов, состоящая из букв, цифр, знаков доллара, знаков подчеркивания и знаков фунта. Другие символы запрещены. Максимальная длина идентификатора — 30 символов, причем все они являются значащими. Ниже приведен ряд разрешенных идентификаторов:

```
x
v_StudentID
TempVar
v1
v2_
social_security_#
```

А эти идентификаторы запрещены:

```
X+y          -- Запрещенный символ +
_Temp_       -- Идентификатор должен начинаться с буквы,
             -- а не со знака подчеркивания
```

```
First Name                          -- Запрещенный пробел
This_is_a_really_long_identifier    -- Более 30 символов
1_variable                          -- Идентификатор не может
                                    -- начинаться с цифры
```

PL/SQL не учитывает регистр символов, поэтому для PL/SQL следующие идентификаторы эквивалентны:

```
Room_Description
room_description
ROOM_DESCRIPTION
rOOm_DEscriPTIOn
```

Хорошим стилем программирования считается создание достаточно наглядных идентификаторов и корректной схемы их именования. Более подробно об этом рассказано в разделе "Стиль программирования на PL/SQL" в конце этой главы.

## Зарезервированные слова

Многие идентификаторы, называемые *зарезервированными* (или ключевыми) словами, имеют в PL/SQL особое значение. Использовать эти слова для именования собственных идентификаторов нельзя. Например, ключевые слова BEGIN и END применяются для ограничения блоков PL/SQL, поэтому их нельзя использовать в качестве имен для переменных. Ниже приведен раздел объявлений, описание которого неверно, так как begin является зарезервированным словом. При обработке этого раздела будет выдано сообщение об ошибке компиляции:

```
DECLARE
  begin NUMBER;
```

Эти слова нельзя использовать как собственно идентификаторы. В состав же других идентификаторов они могут быть включены. Например, следующий раздел объявлений верен, хотя date и является зарезервированным словом:

```
DECLARE
  v_BeginDate DATE;
```

В этой книге зарезервированные слова выделены заглавными буквами, чтобы сделать программы более удобочитаемыми. Подробнее об этом говорится в разделе "Стиль программирования на PL/SQL" в конце данной главы. Полный список зарезервированных слов содержится в приложении А.

## Идентификаторы в кавычках

Если нужно сделать идентификатор чувствительным к регистру символов, включить в его состав такие символы, как пробелы, или воспользоваться зарезервированными словами, можно заключить такой идентификатор в двойные кавычки. Для примера ниже приведены вполне корректные и отличающиеся друг от друга идентификаторы:

```
"A number"
"Linda's variable"
"x/y"
"X/Y"
```

Максимальная длина идентификаторов в кавычках также равна 30 символам (без учета двойных кавычек). В состав идентификатора в кавычках может входить любой печатный символ, за исключением двойных кавычек.

Идентификаторы в кавычках полезны в случае необходимости использовать зарезервированное слово PL/SQL в SQL-операторе. В PL/SQL зарезервировано больше слов, чем в SQL (см. приложение А). Например, обратиться с запросом к таблице, содержащей столбец exception (зарезервированное слово), можно так:

```
DECLARE
  v_Exception VARCHAR2 (10);
BEGIN
  SELECT "EXCEPTION"
    INTO v_Exception
```

```
        FROM exception_table;
END;
```

Обратите внимание, что слово **"EXCEPTION"** написано прописными буквами. Все идентификаторы хранятся в словаре данных в виде символов верхнего регистра, если только идентификатор не создан явным образом (оператором CREATE (создать) таблицу) как идентификатор в кавычках с помощью символов нижнего регистра.

Использование зарезервированных слов в качестве идентификаторов хотя и не запрещено, но считается плохим стилем программирования и затрудняет понимание программы. Оно может потребоваться в единственном случае, когда в таблице базы данных зарезервированное слово PL/SQL используется как имя некоторого столбца. Поскольку в PL/SQL больше зарезервированных слов, чем в SQL, в таблице может содержаться столбец, имя которого является зарезервированным словом PL/SQL, но не является таковым в SQL. Подобную ситуацию иллюстрирует таблица **exception_table** из предыдущего примера.

▼ СОВЕТУЕМ

Хотя таблица **exception_table** вполне может использоваться в PL/SQL, рекомендуется переименовать некорректный столбец. Если описание таблицы изменить нельзя, в качестве альтернативы можно создать представление, в котором данный столбец будет иметь другое имя. Затем это представление можно использовать в PL/SQL. Для примера предположим, что таблица **exception_table** создана следующим образом:

```
-- Этот пример является частью файла tables.sql.
CREATE TABLE exception_table (
   Exception       VARCHAR2(20),
   Date_occured    DATE);
```

С учетом этого описания можно создать представление:

```
CREATE VIEW exception_view AS
   SELECT exception exception_description,
          date_occured
     FROM exception_table;
```

Теперь вместо таблицы **exception_table** можно использовать представление **exception_view,** и на рассматриваемый столбец можно ссылаться как на **exception_description,** что не зарезервировано.

## Ограничители

*Ограничители* – это символы (один символ или их последовательность), которые имеют специальное значение в PL/SQL. Они применяются для отделения идентификаторов друг от друга. Список ограничителей PL/SQL, приведен в таблице 2.1.

### ТАБЛИЦА 2.1. Ограничители PL/SQL

| Символ | Описание | Символ | Описание |
|---|---|---|---|
| + | Знак операции сложения | - | Знак операции вычитания |
| * | Знак операции умножения | / | Знак операции деления |
| = | Знак операции "равно" | < | Знак операции "меньше" |
| > | Знак операции "больше" | ( | Ограничитель начала выражения |
| ) | Ограничитель конца выражения | ; | Признак конца оператора |
| % | Показатель атрибута | ' | Разделитель пунктов |
| . | Выбор компонента | @ | Знак операции "не равно" (эквивалентен <>) |
| , | Ограничитель последовательности символов | " | Ограничитель строки в кавычках |
| : | Присваивание значения переменной | ** | Знак операции возведения в степень |
| <> | Знак операции "не равно" | != | Знак операции "не равно" (эквивалентен <>) |

**ТАБЛИЦА 2.1. Ограничители PL/SQL (продолжение)**

| Символ | Описание | Символ | Описание |
|---|---|---|---|
| ~= | Знак операции "не равно" (эквивалентен !=) | ^= | Знак операции "не равно" (эквивалентен ~=) |
| <= | Знак операции "меньше или равно" | >= | Знак операции "больше или равно" |
| := | Знак операции присваивания | => | Знак операции ассоциативности |
| .. | Знак операции диапазона | \|\| | Знак операции конкатенации строк |
| << | Ограничитель начала метки | >> | Ограничитель конца метки |
| -- | Показатель однострочного комментария | /* | Показатель начала многострочного комментария |
| */ | Показатель конца многострочного комментария | <space> | Пробел |
| <tab> | Символ табуляции | <cr> | Возврат каретки |

# Литералы

*Литерал* — это символьное, числовое или логическое значение, которое не является идентификатором. Например, как -23.456, так и NULL — литералы. Логические, символьные и числовые типы рассмотрены в разделе "Типы PL/SQL" ниже в этой главе.

## Символьные литералы

Символьные литералы (называемые также строковыми) состоят из одного или нескольких символов, ограниченных одиночными кавычками. Символьные литералы могут быть присвоены переменным, имеющим типы CHAR и VARCHAR2, без преобразования. Например, ниже приведены корректные символьные литералы:

```
'12345'
'Four score and seven years ago...'
'100%'
'"'
```

Считается, что все строковые литералы имеют тип данных CHAR. Частью литерала может быть любой печатный символ из набора символов PL/SQL, в том числе и еще одна одиночная кавычка. Чтобы включить одиночную кавычку в строковый литерал, нужно расположить две одиночные кавычки рядом друг с другом. Например, поместить строку "Mike's string" в литерал следует так:

```
'Mike''s string'
```

Следовательно, в PL/SQL строка, состоящая лишь из одной одиночной кавычки, будет выглядеть следующим образом:

```
''''
```

Первая одиночная кавычка ограничивает начало строки, следующие две идентифицируют в строке символ одиночной кавычки, а последняя — указывает на конец строки. Обратите внимание, что это не эквивалентно литералу

```
''
```

который обозначает строку с нулевой длиной. В PL/SQL такая строка считается тождественной NULL-значению.

## Числовые литералы

Числовой литерал может представлять как целое, так и действительное значение. Числовые литералы могут быть присвоены без преобразования переменным, имеющим тип NUMBER. Это единственный вид литералов, которые разрешено использовать в арифметических выражениях. Целые литералы состоят из цифр, перед которыми может присутствовать знак литерала (+ или -). В таких литералах запрещается указывать десятичную точку. Ниже приведены правильные целые литералы:

```
123
-7
+12
0
```

Действительные литералы состоят из цифр (с указанием десятичной точки), перед которыми может стоять знак литерала. Ниже приведены правильные действительные литералы:

```
-17.1
23.0
3.
```

Хотя значения **23.0** и **3.** фактически не содержат дробной части, они тем не менее считаются в PL/SQL действительными литералами. При желании действительные литералы могут быть записаны при помощи экспоненциального представления. Приведенные ниже действительные литералы также вполне корректны:

```
1.345E7
9.87E-3
-7.12e+12
```

После **E** или **e** можно указывать только целый литерал. **E** означает "экспонента", т. е. "умножить на 10 в степени ...". Таким образом, приведенные выше три значения можно представить как:

```
1.345E7  = 1.345 умножить на 10 в степени 7
         = 1.345 x 10,000,000 = 13,450,000
9.87E-3  = 9.87 умножить на 10 в степени -3
         = 9.87 x .001 = 0.00987
-7.12e+12 = -7.12 умножить на 10 в степени 12
          = -7.12 x 1,000,000,000,000
          = -7,120,000,000,000
```

(Здесь запятые указаны для наглядности — в числовых литералах использование запятых запрещено).

## Логические литералы

Существует только три логических литерала: TRUE (истина), FALSE (ложь) и NULL. Эти значения могут быть присвоены лишь логическим (булевым) переменным. Логические литералы обозначают истинность или ошибочность некоторых условий и используются в операторах IF и LOOP.

# Комментарии

*Комментарии* повышают удобочитаемость программ и делают их более понятными. Компилятор PL/SQL игнорирует комментарии. Существуют комментарии двух видов: однострочные и многострочные; последние часто называют комментариями С-типа.

## Однострочные комментарии

Однострочный комментарий начинается с двух символов тире и продолжается до конца строки (ограниченной символом возврата каретки). Рассмотрим некоторый блок PL/SQL:

```
-- Этот пример является частью файла comments.sql.
DECLARE
  V_Department   CHAR(3);
  V_Course       NUMBER;
BEGIN  PL/SQL
  INSERT INTO classes (department, course)
      VALUES (v_Department, v_Course);
END;
```

Теперь для удобства добавим к этому блоку однострочные комментарии:

```
-- Этот пример является частью файла comments.sql.
DECLARE
```

```
  v_Department CHAR(3);     -- Переменная для хранения трехсимвольного
                            -- кода факультета.
  v_Course NUMBER;          -- Переменная для хранения номера курса.
BEGIN
  -- Введем в таблицу classes базы данных курс, указанный в
  -- v_Department и v_Course.
  INSERT INTO classes (department, course)
     VALUES (v_Department, v_Course);
END;
```

▼ ВНИМАНИЕ Если комментарий распространяется более чем на одну строку, то двойное тире (--) необходимо указывать в начале каждой строки.

## Многострочные комментарии

Многострочные комментарии начинаются с ограничителя /* и заканчиваются ограничителем */, как это делается в языке программирования C. Например:

```
-- Этот пример является частью файла comments.sql.
DECLARE
  v_Department CHAR(3);     /* Переменная для хранения трехсимвольного
                               кода факультета. */
  v_Course NUMBER;          /* Переменная для хранения номера курса. */
BEGIN
  /* Введем в таблицу classes базы данных курс, указанный в
     v_Department и v_Course. */
  INSERT INTO classes (department, course)
     VALUES (v_Department, v_Course);
END;
```

Многострочные комментарии могут распространяться на сколь угодно большое число строк, однако они не могут быть вложенными, т.е. до начала одного комментария другой должен быть закончен. Ниже приведен пример блока, который неверен, так как в нем имеются вложенные комментарии:

```
-- Этот пример является частью файла comments.sql.
BEGIN
  /* Это комментарий. Если начать другой комментарий следующим
     образом: /* вот так */, он будет неверен. */
  NULL;
END;
```

# Объявление переменных

В блоках PL/SQL взаимодействие с базой данных осуществляется посредством переменных. *Переменные* (variables) — это области памяти, в которых могут храниться некоторые значения данных. По мере выполнения программы содержимое переменных модифицируется. Переменной может быть присвоена определенная информация, сохраняемая в базе данных, либо содержимое переменной может быть внесено в базу данных. Переменные описываются в разделе объявлений блока. Каждая переменная имеет конкретный тип, определяющий тип хранящейся в ней информации. Сведения о типах приведены в сжатом виде.

## Синтаксис объявления

Переменные описываются в разделе объявлений блока. Общий синтаксис объявления переменных таков:

*имя_переменной тип* [CONSTANT] [NOT NULL] [:= *значение*];

где *имя_переменной* — это имя переменной, *тип* — это тип, а *значение* — начальное значение переменной. Ниже приведены примеры корректного объявления переменных:

```
DECLARE
   V_Description     VARCHAR2(50);
   V_NumberSeats     NUMBER := 45;
   V_Counter         BINARY_INTEGER := 0;
```

В качестве имени переменной может быть использован любой разрешенный идентификатор PL/SQL. Типы **VARCHAR2, NUMBER** и **BINARY_INTEGER** – это корректные типы PL/SQL. В этом примере для переменных **v_NumberSeats** и **v_Counter** заданы начальные значения соответственно 45 и 0. Если для переменной, например для **v_Description,** не задано начальное значение (она не инициализирована), по умолчанию ей присваивается NULL-значение. Если в объявлении указано NOT NULL, переменная должна быть инициализирована. Более того, запрещается присваивать NULL-значение переменной, которая ограничена как NOT NULL в выполняемом разделе или в разделе исключительных ситуаций блока. Нижеприведенное объявление неверно, так как переменная **v_TempVar** ограничена как NOT NULL, но не инициализирована:

```
DECLARE
   v_TempVar NUMBER NOT NULL;
```

Это можно исправить, если присвоить **v_TempVar** значение по умолчанию, например:

```
DECLARE
   v_TempVar NUMBER NOT NULL := 0;
```

Если в объявлении переменной указано CONSTANT, то она должна быть инициализирована и ее начальное значение не может быть изменено. Переменная-константа в оставшейся части блока рассматривается в качестве переменной "только для чтения". Константы часто используются для хранения тех значений, которые известны к моменту создания блока, например:

```
DECLARE
   c_MinimumStudentID CONSTANT NUMBER(5) := 10000;
```

При желании вместо := можно воспользоваться ключевым словом DEFAULT (по умолчанию). Например:

```
DECLARE
   V_NumberSeats     NUMBER DEFAULT 45;
   V_Counter         BINARY_INTEGER DEFAULT 0;
   V_FirstName       VARCHAR2 (20) DEFAULT 'Scott';
```

В разделе объявлений в одной строке может быть описана только одна переменная. Нижеприведенный раздел неверен, так как в одной и той же строке объявляются две переменные:

```
DECLARE
   v_FirstName, v_LastName VARCHAR2 (20);
```

Корректный вариант этого блока должен выглядеть так:

```
DECLARE
   v_FirstName VARCHAR2 (20);
   v_LastName VARCHAR2 (20);
```

## Инициализация переменных

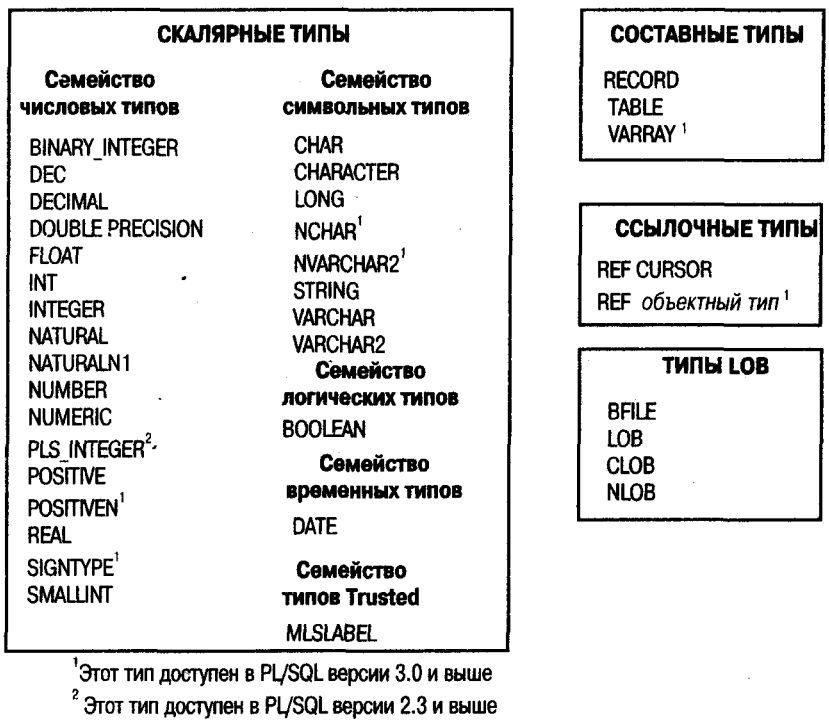
Во многих языках программирования не указано, что содержится в неинициализированных переменных. В итоге во время выполнения программы в таких переменных могут оказаться случайные либо неизвестные значения. Это считается некорректным стилем программирования; вообще говоря, *рекомендуется инициализировать переменные, если их значения могут быть определены.*

В PL/SQL информация, содержащаяся в неинициализированной переменной, определяется, т.е. такой переменной присваивается NULL-значение. NULL означает "пропущенное или неизвестное значение", поэтому вполне логично, что NULL-значение по умолчанию присваивается любой неинициализированной переменной. Это свойство присуще PL/SQL, а во многих других языках программирования (в том числе в C и Ada) значения для неинициализированных переменных не определяются.

# Типы PL/SQL

В PL/SQL версий 1 и 2 имелось три категории типов: скалярные, составные и ссылочные. В PL/SQL 8.0 введена дополнительная категория — типы LOB. Внутри скалярных типов не содержится никаких компонентов, в то время как в составных типах они присутствуют. Ссылочный тип является указателем на другой тип. На рис. 2.1 приведен список всех типов PL/SQL, которые описаны в последующих разделах.

**Рис. 2.1.**
*Типы PL/SQL*




[1] Этот тип доступен в PL/SQL версии 3.0 и выше
[2] Этот тип доступен в PL/SQL версии 2.3 и выше

Типы PL/SQL определены в модуле под названием STANDARD. Обратиться к содержимому этого модуля можно из любого блока PL/SQL. Помимо типов, в модуле STANDARD определены встроенные SQL-функции и функции преобразования.

## Скалярные типы

Допустимые скалярные типы включают типы, аналогичные тем, которые применяются для определения столбцов таблиц базы данных, и ряд дополнительных типов. Скалярные типы можно разделить на семь семейств: числовые типы, символьные типы, типы без обработки, типы даты, типы ROWID, логические типы и типы Trusted. Все они описаны ниже.

### Семейство числовых типов

При помощи типов этого семейства хранятся целые и действительные значения. Существует три базовых типа: NUMBER, PLS_INTEGER и BINARY_INTEGER. В переменных, имеющих тип NUMBER, можно сохранять как целые, так и действительные величины, а в переменных типа BINARY_INTEGER или PLS_INTEGER — только целые числа.

**NUMBER** С помощью данного типа можно хранить числа, как целые, так и с плавающей точкой. Он абсолютно аналогичен типу NUMBER, применяемому в базах данных. Синтаксис объявления некоторого числа таков:

```
NUMBER (P,S);
```

где $P$ — точность (precision), а $S$ — масштаб (scale). Точность — это количество цифр в значении, а масштаб — количество цифр справа от десятичной точки. Использование как точности, так и масштаба необязательно, однако при указании масштаба необходимо указывать и точность. Различные комбинации точности и масштаба продемонстрированы в таблице 2.2.

**ТАБЛИЦА 2.2. Значения точности и масштаба**

| Объявление | Присвоенное значение | Хранимое значение |
|---|---|---|
| NUMBER; | 1234.5678 | 1234.5678 |
| NUMBER(3); | 123 | 123 |
| NUMBER(3); | 1234 | Ошибка — превышение точности |
| NUMBER(4,3); | 123.4567 | Ошибка — превышение точности |
| NUMBER(4,3); | 1.234567 | 1.235[1] |
| NUMBER(7,2); | 12345.67 | 12345.67 |
| NUMBER(3,-3); | 1234 | 1000[2] |
| NUMBER(3,-1); | 1234 | 1230[2] |

[1]Если масштаб присваиваемого значения превышает указанный, то сохраняемое значение округляется до числа цифр, указанного в масштабе.

[2]Если масштаб отрицателен, то сохраняемое значение округляется до числа цифр, указанного в масштабе.

Максимальной точностью является 38 цифр, а масштаб может быть числом в диапазоне от -84 до 127.

*Подтип* (subtype) — это альтернативное имя для некоторого типа, с помощью которого можно ограничить множество значений, допустимых для переменных, имеющих данный подтип. (Подтипы подробно рассмотрены в разделе "Подтипы, определяемые пользователями" ниже в этой главе.) Типу NUMBER эквивалентен целый ряд подтипов, которые, в сущности, являются тем же самым, что и NUMBER, так как ни один из них не ограничен. Альтернативные имена можно использовать для повышения удобочитаемости программ либо для обеспечения совместимости с типами данных, применяемыми в других базах данных. Типу NUMBER эквивалентны:

- DEC
- DECIMAL
- DOUBLE PRECISION
- INTEGER
- INT
- NUMERIC
- REAL
- SMALLINT

**BINARY_INTEGER** Данные, имеющие тип NUMBER, хранятся в десятичном формате, который специально создан для точного и эффективного хранения информации. Вследствие этого арифметические операции нельзя выполнять непосредственно над значениями типа NUMBER. Чтобы числовые величины могли быть обработаны, значения типа NUMBER должны быть преобразованы к двоичному типу. Если имеется арифметическое выражение, в котором используются значения типа NUMBER, то в PL/SQL это преобразование, а также (при необходимости) преобразование результата обратно к типу NUMBER выполняются автоматически.

Однако, если некоторое значение не будет храниться в базе данных, а будет применяться лишь в вычислениях, можно воспользоваться типом данных BINARY_INTEGER. Этот тип применяется для хранения знаковых целых значений, которые могут лежать в диапазоне от -2147483647 до +2147483647. Данные хранятся в формате точного двоичного дополнения, а это означает, что для их использования в различных вычислениях не требуется выполнять их преобразование. Тип BINARY_INTEGER часто имеют счетчики циклов.

Как и для NUMBER, для типа BINARY_INTEGER определен ряд подтипов. Однако в отличие от NUMBER подтипы BINARY_INTEGER *ограничены*, т.е. с их помощью можно хранить значения, на которые наложены некоторые ограничения. Подтипы BINARY_INTEGER приведены в таблице 2.3.

**ТАБЛИЦА 2.3. Подтипы BINARY_INTEGER**

| Подтип | Ограничение |
|---|---|
| NATURAL | 0..2147483647 |
| NATURALN | 0..2147483647 NOT NULL |
| POSITIVE | 1..2147483647 |
| POSITIVEN | 1..2147473647 NOT NULL |
| SIGNTYPE | -1, 0, 1 |

PL/SQL 2.3 ... и ВЫШЕ

**PLS_INTEGER** Тип PLS_INTEGER имеет тот же диапазон значений, что и BINARY_INTEGER – от -2147483647 до +2147483647 – и реализован также при помощи формата точного двоичного дополнения. Однако, если при выполнении некоторой операции переполняется значение типа PLS_INTEGER, возникает ошибка. В случае же переполнения значения, имеющего тип BINARY_INTEGER, результат может быть присвоен переменной типа NUMBER (который имеет больший диапазон) без всякой ошибки.

## Семейство символьных типов

Переменные этих типов используются для хранения строковых, или символьных, данных. В это семейство входят типы VARCHAR2, CHAR и LONG, а также NCHAR и NVARCHAR2 (два последних типа доступны только в PL/SQL версии 8.0 и выше).

**VARCHAR2** Этот тип аналогичен типу VARCHAR2, применяемому в базах данных. При помощи переменных типа VARCHAR2 можно хранить строки символов переменной длины. Синтаксис объявления переменной, имеющей тип VARCHAR2, таков:

```
VARCHAR2(L);
```

где *L* – максимальная длина (length) переменной. Указание длины обязательно – значения по умолчанию не существует. Максимальная длина переменной типа VARCHAR2 составляет 32 767 байтов. Обратите внимание, что в поле столбца базы данных, имеющем тип VARCHAR2, можно хранить только 2000 байтов. Если длина переменной типа VARCHAR2 PL/SQL превышает 2000 байтов, ее можно ввести лишь в столбец базы данных, имеющий тип LONG, максимальный размер которого составляет 2 Гбайта. Аналогично, данные LONG нельзя выбрать в переменную VARCHAR2, если их размер превышает 32 767 байтов.

▼ ВНИМАНИЕ

В Oracle8 в поле столбца базы данных, имеющем тип VARCHAR2, можно сохранять 4000 байтов. Таким образом, переменная PL/SQL VARCHAR2 может быть введена в столбец Oracle8 VARCHAR2, только если ее длина не превышает 4000 байтов.

Для значений VARCHAR2 длина указывается не в символах, а в байтах. Реальная информация хранится в базе данных с помощью принятого набора символов, например ASCII или EBCDIC Code Page 500. Если в некотором наборе символов базы данных содержатся многобайтовые символы, максимальное число символов, которое может храниться в переменной типа VARCHAR2, возможно, будет меньше указанной длины. Дело в том, что для представления одного символа может использоваться более одного байта.

Подтип VARCHAR эквивалентен типу VARCHAR2.

**CHAR** Переменные этого типа представляют собой строки символов фиксированной длины. Синтаксис объявления переменной CHAR таков:

```
CHAR(L);
```

где *L* – максимальная длина в байтах. Однако, в отличие от типа VARCHAR2, в этом случае указание длины необязательно. Если она не указана, принимается значение по умолчанию 1, причем круглые скобки не нужны. Переменные типа CHAR имеют фиксированную длину, поэтому при необходимости они заполняются до максимальной длины пробелами. В результате переменные типа CHAR не всегда будут совпадать при выполнении операций сравнения символов. Более подробно о сравнении символов рассказано в разделе "Логические выражения" ниже в этой главе.

Максимальная длина переменной типа CHAR равна 32 767 байтам. Максимальная же длина поля столбца базы данных, имеющего тип CHAR, составляет 255 байтов. Таким образом, если в переменной CHAR содержится более 255 байтов, ее можно ввести только в столбец типа VARCHAR2 или LONG.

Аналогично, данные типа LONG можно выбрать в переменную типа CHAR только в случае, если их размер составляет не более 32 767 байтов.

▼ ВНИМАНИЕ **В Oracle8 в поле столбца базы данных, имеющем тип CHAR, можно сохранять до 2000 байтов.**

Как и для VARCHAR2, длина переменной типа CHAR указывается не в символах, а в байтах. Если в некотором наборе символов базы данных содержатся многобайтовые символы, то максимальное число символов, которое может храниться в переменной типа VARCHAR2, возможно, будет меньше указанной длины.

Подтипом CHAR, имеющим те же ограничения, является CHARACTER. Семантика переменных VARCHAR2 и переменных CHAR существенно различается (более подробно об этом рассказано в разделе "Логические выражения" ниже в этой главе).

**LONG** В отличие от типа LONG, применяемого в базах данных и позволяющего хранить до 2-х Гбайтов информации, при помощи типа PL/SQL LONG можно сохранять последовательности символов переменной длины, максимальный размер которых равен 32 760 байтам. Переменные LONG очень похожи на переменные VARCHAR2. Как и в случае с переменными VARCHAR2, если в поле столбца LONG базы данных содержится более 32 760 байтов информации, выбрать эту информацию в переменную PL/SQL LONG нельзя. Однако максимальная длина переменной PL/SQL LONG меньше, чем поле LONG базы данных, поэтому переменная PL/SQL LONG может быть введена в столбец LONG базы данных без всяких ограничений.

PL/SQL 8.0 ... и ВЫШЕ **NCHAR и NVARCHAR2** В Oracle8 предусмотрены два дополнительных типа, которые используются и в PL/SQL 8.0. Это символьные типы NLS (NLS — National Language Support -- поддержка национальных языков. — *Прим. пер.*): NCHAR и NVARCHAR2. Они применяются для хранения строк символов при помощи набора символов, отличного от того, который используется в языке программирования PL/SQL. Такой набор называется *национальным набором символов* (national character set).

Переменные типов NCHAR и NVARCHAR2 описываются точно так же, как и переменные типов CHAR и VARCHAR2. Однако длина может меняться в зависимости от применяемого национального набора символов. Если в таком наборе размер символов фиксирован, длина указывается в символах. Если же их размер *непостоянен, длина указывается в байтах.*

Более подробно о типах NCHAR, NVARCHAR2 и о NLS вообще рассказано в справочном руководстве Oracle8 Server SQL Reference.

## Семейство типов без обработки

Типы этого семейства используются для хранения двоичных данных. При необходимости Oracle автоматически преобразует символьные переменные, для которых применяются разные наборы символов. Это может происходить, когда информация посредством связи баз данных передается из одной базы в другую, причем каждая из них использует свой набор символов. Для переменных, имеющих тип без обработки (raw type), это не выполняется.

**RAW** Переменные типа RAW похожи на переменные типа CHAR, однако они не преобразуются из одного набора символов в другой. Синтаксис описания переменной RAW таков:

RAW(*L*);

где L — длина переменной в байтах. Тип RAW используется для хранения двоичных данных фиксированной длины. В отличие от символьных данных, данные типа RAW не преобразуются из одного набора символов в другой при их передаче из одной базы данных в другую. Максимальная длина переменной RAW равна 32 767 байтам. Максимальная же длина поля RAW базы данных составляет 255 байтов, поэтому, если размер данных превышает 255 байтов, они не могут быть введены в столбец RAW базы данных. Однако они могут быть введены в столбец базы данных, имеющий тип LONG RAW, максимальная длина которого составляет 2 Гбайта. Аналогично, если длина данных в поле LONG RAW превышает 32 767 байтов, выбрать их в переменную PL/SQL RAW нельзя.

**LONG RAW** Данные типа LONG RAW похожи на данные типа LONG, за исключением того, что в PL/SQL не происходит их преобразования из одного набора символов в другой. Максимальная длина переменной LONG RAW равна 32 760 байтам. Максимальная длина поля LONG RAW базы данных составляет 2 Гбайта, поэтому, если фактический размер данных превышает 32 760 байтов, выбрать их в переменную PL/SQL LONG RAW нельзя. Но максимальная длина переменной PL/SQL LONG RAW вполне подходит для ввода такой переменной в поле LONG RAW базы данных, поэтому ограничений на ввод переменных PL/SQL LONG RAW в поля LONG RAW базы данных не существует.

## Семейство типов даты

В этом семействе имеется только один тип – DATE, который абсолютно аналогичен типу DATE, применяемому в базах данных. Тип DATE используется для хранения информации как о датах, так и о времени, в том числе о веках, годах, месяцах, днях, часах, минутах и секундах. Размер переменной DATE составляет 7 байтов, по одному байту на каждый компонент (от века до секунды).

Значения переменным DATE обычно присваиваются посредством встроенной функции TO_DATE. Это позволяет легко преобразовывать символьные переменные в переменные DATE. Подобно этому при помощи функции TO_CHAR можно преобразовывать переменные DATE в символьные переменные. Встроенные функции преобразования описаны в разделе "Преобразование типов данных" ниже в этой главе, а также в главе 3.

## Семейство типов ROWID

В этом семействе только один тип – ROWID, который абсолютно аналогичен типу, используемому для работы с псевдостолбцами ROWID базы данных. Он дает возможность сохранять идентификаторы строк (rowids), которые можно рассматривать в качестве ключей, однозначно определяющих каждую строку базы данных. Идентификаторы строк хранятся внутри базы данных в виде двоичных значений фиксированной длины, размер которых зависит от применяемой операционной системы. Для работы с идентификаторами строк их можно преобразовать в последовательности символов при помощи встроенной функции ROWIDTOCHAR. Результатом работы этой функции является 18-символьная последовательность, имеющая формат:

```
BBBBBBBB.RRRR.FFFF
```

где BBBBBBBB определяет блок в файле базы данных, RRRR – строку в блоке, а FFFF – номер файла. Каждый элемент идентификатора строки представлен в виде шестнадцатиричного числа. Например, идентификатор строки

```
0000001E.00FF.0001
```

указывает на 30-й блок, 255-ую строку в этом блоке, который расположен в файле 1. Идентификаторы строк обычно не создаются программами PL/SQL; они выбираются в псевдостолбце ROWID таблицы. Затем выбранное значение может быть использовано в задаваемом условии оператора UPDATE или DELETE.

**PL/SQL 8.0 ... и ВЫШЕ**

В Oracle8 возможности типа ROWID расширены, в частности включена поддержка разделенных таблиц и индексов посредством использования *номеров объектов данных* (data object numbers), которые идентифицируют сегменты базы данных. Этот расширенный формат ROWID применяется для работы с модулем DBMS_ROWID, описанным в приложении В.

## Семейство логических типов

Единственным типом данных этого семейства является тип BOOLEAN. Логические переменные используются в управляющих структурах PL/SQL, таких как операторы IF-THEN-ELSE и LOOP. Значение типа BOOLEAN может быть только TRUE, FALSE и NULL. Поэтому приведенный ниже раздел объявлений неверен, так как 0 не является допустимым значением для типа BOOLEAN:

```
DECLARE
   v_ContinueFlag  BOOLEAN  :=  0;
```

## Семейство типов Trusted

Единственным типом данных этого семейства является тип MLSLABEL, используемый в Trusted Oracle для хранения двоичных меток переменной длины. В стандартной системе Oracle в переменных и столбцах, имеющих тип MLSLABEL, могут содержаться только NULL-значения. Внутренняя длина переменных типа MLSLABEL составляет от 2 до 5 байтов, однако их можно автоматически преобразовывать в символьные переменные, а также выполнять обратное преобразование. Максимальная длина символьного представления переменной MLSLABEL составляет 255 байтов.

# Составные типы

В PL/SQL применяются три составных типа: записи, таблицы и изменяемые массивы. *Составной тип* содержит некоторые компоненты. В переменной, имеющей составной тип, находится одна или несколько скалярных переменных. Более детально составные типы обсуждаются в следующей главе.

## Ссылочные типы

PL/SQL 2.2 ... и ВЫШЕ

После того как переменная PL/SQL объявлена и ей назначен некоторый скалярный или составной тип, для ее хранения выделяется определенная область памяти. Переменная дает выделенной области имя и впоследствии используется в программе для ссылки на нее. Однако нельзя отменить выделение памяти и одновременно сохранить возможность работы с переменной – память не освобождается до тех пор, пока переменная находится в области своего действия (об области действия говорится в разделе "Области действия и области видимости переменных" ниже в этой главе). Для ссылочных типов такого ограничения нет. Ссылочный тип PL/SQL – это то же самое, что и указатель в C. Переменная, объявленная со ссылочным типом, во время выполнения программы может указывать на различные области памяти.

В PL/SQL 2.2 применяется лишь один ссылочный тип – REF CURSOR, называемый также курсорной переменной. Он подробно рассмотрен в главе 4. В PL/SQL 8.0 представлен объектный тип REF, который может указывать на некоторый объект. Объекты REF описаны в главе 11.

## Типы LOB

PL/SQL 8.0 ... и ВЫШЕ

Типы LOB используются для хранения больших объектов. Большой объект (large object) может быть либо двоичным, либо символьным значением размером до 4-х Гбайтов. В больших объектах могут содержаться неструктурированные данные, доступ к которым осуществляется эффективнее, чем к данным типа LONG или LONG RAW, с меньшим числом ограничений. Типы LOB управляются с помощью модуля DBMS_LOB (более детально они обсуждаются в главе 21).

## Использование %TYPE

Во многих случаях для работы с информацией, хранимой в таблицах базы данных, используются переменные PL/SQL. При этом переменной, работающей с некоторым столбцом, следует присваивать тип, соответствующий типу столбца. Например, столбец **first_name** таблицы **students** имеет тип VARCHAR2(20). С учетом этого можно объявить переменную следующим образом:

```
DECLARE
   V_FirstName    VARCHAR2(20);
```

Теперь посмотрим, что произойдет, если описание столбца **first_name** изменится. Предположим, что таблица модифицирована и тип столбца **first_name** стал VARCHAR2(25). Текст программы PL/SQL, в которой используется этот столбец, должен быть изменен:

```
DECLARE
   V_FirstName    VARCHAR2(25);
```

Если программа PL/SQL достаточно велика, этот процесс может занять длительное время и, кроме того, существует большая вероятность ошибки. В этом случае вместо жесткого кодирования типа переменной можно воспользоваться атрибутом %TYPE. Он добавляется к ссылке на столбец таблицы или к другой переменной и возвращает тип переменной. Например:

```
DECLARE
   v_FirstName  students.first_name%TYPE;
```

При использовании %TYPE переменная **v_FirstName** будет иметь тот тип, который присвоен столбцу **first_name** таблицы **students.** Тип определяется всякий раз, когда данный блок выполняется для анонимных и именованных блоков и когда компилируются хранимые объекты (процедуры, функции и т.д.). %TYPE может быть также использован с переменными PL/SQL, объявленными ранее. Нижеприведенный пример иллюстрирует различные варианты применения атрибута %TYPE.

```
DECLARE
   V_RoomID       classes.room_id%TYPE;    -- возвращает NUMBER (5)
   V_RoomID2      v_RoomID%TYPE;           -- возвращает NUMBER (5)
   V_TempVar      NUMBER(7,3) NOT NULL := 12.3;
   V_AnotherVar   v_TempVar%TYPE;          -- возвращает NUMBER (7,3)
```

Если атрибут %TYPE применяется для переменной (столбца), которая (который) содержит ограничение NOT NULL (например, **classes.room_id** или **v_TempVar**), то возвращаемый тип не имеет этого

ограничения. Приведенный выше блок вполне корректен, хотя переменные **v_RoomID, v_RoomID2** и **v_AnotherVar** не инициализированы. Дело в том, что эти переменные могут содержать NULL-значения.

Использование атрибута %TYPE считается хорошим стилем программирования, так как при этом программы PL/SQL становятся гибче и проще адаптируются к изменению различных параметров, задаваемых в базе данных.

## Подтипы, определяемые пользователями

PL/SQL 2.1 ... и ВЫШЕ

Подтип (subtype) – это тип PL/SQL, в основе которого лежит существующий тип. С помощью подтипа можно дать типу альтернативное имя, которое более точно описывает назначение этого типа. Ряд подтипов PL/SQL (например, DECIMAL и INTEGER – подтипы NUMBER) определен в модуле STANDARD. В PL/SQL версии 2.1 и выше пользователи могут сами определять подтипы, добавляя их к заранее определенным подтипам. При этом применяется следующий синтаксис:

SUBTYPE *новый_тип* IS *исходный_тип*;

где *новый_тип* – имя нового подтипа, а *исходный_тип* указывает базовый тип. Базовый тип может быть либо заранее определенным типом, либо подтипом, либо ссылкой %TYPE. Например:

```
DECLARE
   SUBTYPE t_LoopCounter IS NUMBER;  -- определяем новый подтип
   v_LoopCounter t_LoopCounter;      -- объявляем переменную с этим подтипом
   SUBTYPE t_NameType IS students.first_name%TYPE;
```

Подтип нельзя ограничивать непосредственно в определении SUBTYPE. Приведенный ниже блок неверен:

```
DECLARE
   SUBTYPE T_LoopCounter IS NUMBER (4); -- неверное ограничение
```

Однако это правило можно обойти: объявить фиктивную переменную нужного типа (с ограничением) и в определении SUBTYPE воспользоваться атрибутом %TYPE:

```
DECLARE
   v_DummyVar NUMBER (4);  -- фиктивная переменная, которая не
                           -- будет использоваться
   SUBTYPE t_LoopCounter is v_DummyVar%TYPE; -- возвращает NUMBER(4)
   v_Counter t_LoopCounter;
```

При объявлении переменной с неограниченным подтипом ее тип может быть ограничен:

```
DECLARE
   SUBTYPE t_Numeric IS NUMBER;    -- определяем неограниченный подтип,
   SUBTYPE t_Numeric (5);          -- но ограниченную переменную
```

Подтип принадлежит к тому же семейству типов, что и базовый тип.

## Преобразование типов данных

В PL/SQL можно выполнять преобразование скалярных типов данных, принадлежащих разным семействам типов. В пределах одного семейства типы данных можно преобразовывать произвольно, за исключением ограничений, налагаемых на переменные. Например, переменную типа CHAR(10) нельзя преобразовать в переменную типа VARCHAR2(1), так как не будет хватать места для хранения значения этой переменной. Ограничения точности и масштаба могут также не позволить преобразование, например, переменной NUMBER(3,2) в переменную NUMBER(3) и наоборот. В случае нарушения ограничения компилятор PL/SQL не выдаст сообщения об ошибке, однако ошибки могут появиться во время выполнения программы – все зависит от значений преобразуемых переменных.

Вообще говоря, выполнять преобразование составных типов данных между собой нельзя, потому что они слишком разнородны. Однако при необходимости для преобразования таких типов можно создать специальную функцию, взяв за основу назначение типов данных в программе.

Независимо от типа существует два способа преобразования типов данных: явный и неявный.

## Явное преобразование типов данных

Встроенные функции преобразования, применяемые в SQL, используются также и в PL/SQL. Краткое описание этих функций приведено в таблице 2.4. При необходимости их можно применять для явного преобразования типов данных, относящихся к различным семействам типов. Более подробную информацию о функциях преобразования и примеры их использования можно найти в главе 5.

### ТАБЛИЦА 2.4. Функции преобразования типов данных PL/SQL и SQL

| Функция | Описание | Семейства типов, доступные для преобразования |
|---|---|---|
| TO_CHAR | Преобразует аргумент к типу VARCHAR2 в зависимости от указанного формата (указание формата необязательно) | Числовые типы, типы даты |
| TO_DATE | Преобразует аргумент к типу DATE в зависимости от указанного формата (указание формата необязательно) | Символьные типы |
| TO_NUMBER | Преобразует аргумент к типу NUMBER в зависимости от указанного формата (указание формата необязательно) | Символьные типы |
| RAWTOHEX | Преобразует значение RAW в шестнадцатиричное представление двоичной величины | Типы без обработки Raw |
| HEXTORAW | Преобразует шестнадцатиричное представление в эквивалентную двоичную величину | Символьные типы (значения должны быть представлены в шестнадцатиричном виде) |
| CHARTOROWID | Преобразует символьное представление значения ROWID во внутренний двоичный формат | Символьные типы (значения должны быть представлены в 18-символьном формате идентификаторов строк) |
| ROWIDTOCHAR | Преобразует внутреннюю двоичную переменную ROWID в 18-символьный внешний формат | Типы ROWID |

## Неявное преобразование типов данных

В PL/SQL осуществляется автоматическое преобразование типов данных разных семейств, когда это возможно. Например, при помощи блока, приведенного ниже, выбирается текущее число зачетов для студента 10002:

```
DECLARE
  v_CurrentCredits VARCHAR2(5);
BEGIN
  SELECT current_credits
      INTO v_CurrentCredits
      FROM students
      WHERE id = 10002;
END;
```

В этой базе данных поле **current_credits** имеет тип NUMBER(3), однако **v_CurrentCredits** – это переменная типа VARCHAR2(5). PL/SQL автоматически преобразует числовые данные в строку символов, а затем присваивает ее символьной переменной. В PL/SQL может осуществляться преобразование:

- строк символов и чисел
- строк символов и дат

Хотя в PL/SQL производится неявное преобразование типов данных, при программировании рекомендуется использовать явные функции преобразования. В следующем примере та же операция выполнена при помощи функции TO_CHAR:

```
DECLARE
  v_CurrentCredits VARCHAR2(5);
BEGIN
```

```
    SELECT TO_CHAR (current_credits)
        INTO v_CurrentCredits
        FROM students
        WHERE id = 10002;
END;
```

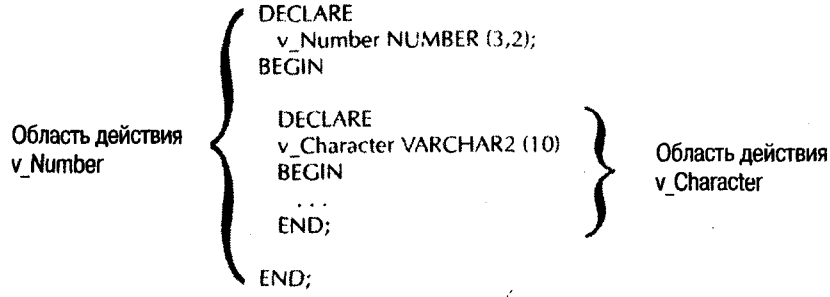
Преимущество такого способа заключается в том, что при желании строка символов явного формата может быть использована также в функции TO_CHAR. Это облегчает понимание программы и подчеркивает преобразование типов.

Автоматическое преобразование типов может осуществляться также при обработке в PL/SQL различных выражений (которые всесторонне обсуждены в разделе "Выражения и операции" ниже в этой главе). Для выражений применима та же рекомендация: следует использовать явные функции преобразования типов.

## Области действия и области видимости переменных

*Область действия* (scope) переменной – это фрагмент программы, в котором возможно обращение к этой переменной. Для переменной PL/SQL – это фрагмент с момента ее объявления и до конца блока. Когда переменная выходит из своей области действия, PL/SQL освобождает память, используемую для хранения данной переменной, так как в этом случае ссылки на нее становятся невозможны. Все это проиллюстрировано на рис. 2.2. Областью действия переменной **v_Character** является только внутренний блок; после ключевого слова END внутреннего блока ее область действия заканчивается. Область действия переменной **v_Number** распространяется до ключевого слова END внешнего блока. Внутренний блок является областью действия обеих переменных.

**Рис. 2.2.**
*Область действия переменных*




*Область видимости* (visibility) переменной – это фрагмент программы, в котором возможно обращение к этой переменной без использования ее квалифицированного имени. Область видимости всегда лежит в пределах области действия; если переменная находится вне области своего действия, она невидима. Рассмотрим рис.2.3. В точке 1 обе переменные – **v_AvailableFlag** и **v_SSN** – находятся в границах своих областей действия и видимы. В точке 2 обе эти переменные находятся в пределах своих областей действия, но теперь видима только переменная **v_AvailableFlag**. При повторном объявлении переменной **v_SSN** с типом CHAR(11) объявление NUMBER(9) было скрыто. В этой точке все четыре переменные – в границах своих областей действия, но видимы только три: **v_AvailableFlag, v_StartDate** и CHAR(11) **v_SSN**. В точке 3 переменные **v_StartDate** и CHAR(11) **v_SSN** находятся вне пределов своих областей действия и поэтому больше невидимы. Здесь находятся в границах областей действия и видимы те же переменные, что и в точке 1, – **v_AvailableFlag** и NUMBER(9) **v_SSN**.

Если переменная находится в своей области действия, но невидима, – то как обратиться к ней в программе? Рассмотрим рис. 2.4. Это тот же самый блок, который был проанализирован на рис. 2.3, но для внешнего блока создана метка **<<l_Outer>>** (метки рассмотрены более подробно в разделе "Управляющие структуры PL/SQL" ниже в этой главе). В точке 2 переменная NUMBER(9) **v_SSN** невидима. Однако обратиться к ней можно с помощью метки в квалифицированном имени:

```
l_Outer.v_SSN
```

**Рис. 2.3.**
*Области действия и области видимости*

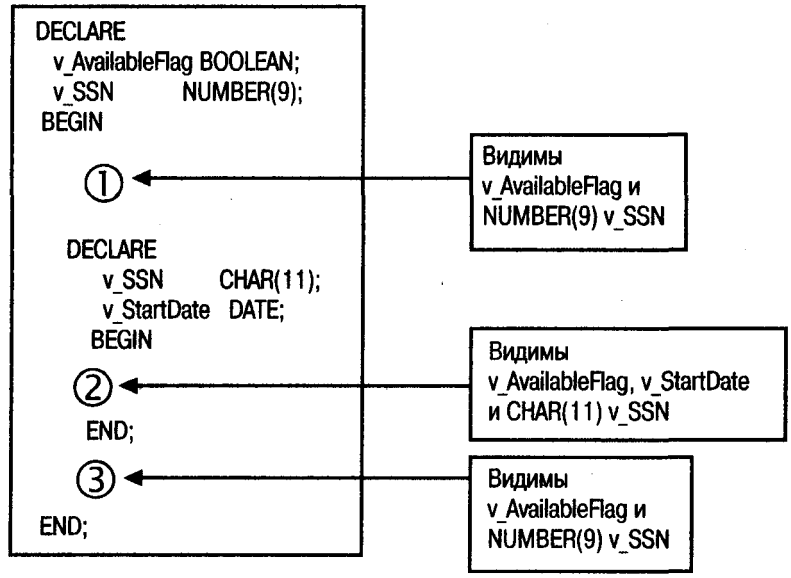
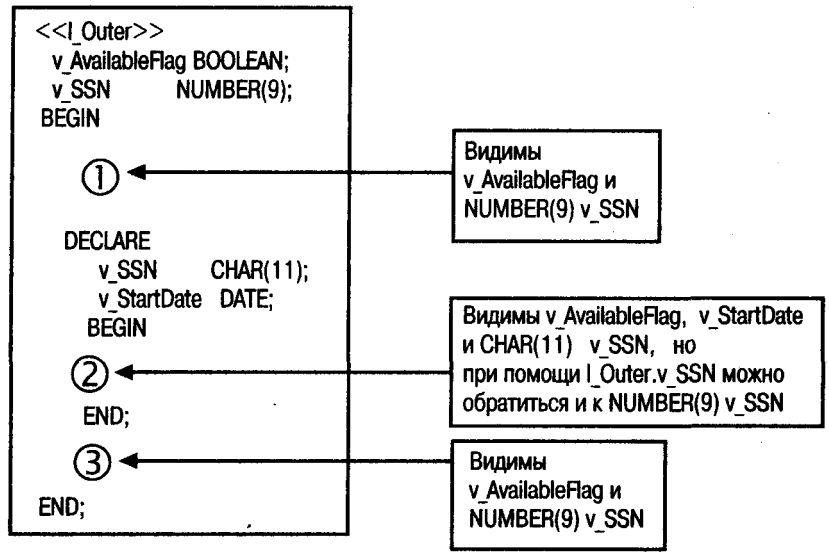



**Рис. 2.4.**
*Использование метки для ссылки на переменную*




# Выражения и операции

Посредством выражений и операций осуществляется связывание переменных PL/SQL. С помощью операций определяются способы присваивания переменным конкретных значений и способы работы с этими значениями. *Выражение* (expression) — это некоторая последовательность переменных и литералов, разделенных знаками операций, или *операторами* (operators). Значение выражения определяется значениями переменных и литералов, составляющих это выражение, а также описанием используемых операций.

## Присваивание

Основной операцией является операция присваивания (assignment). Ее синтаксис таков:

*переменная* := *выражение*;

где *переменная* – это переменная PL/SQL, а *выражение* – это выражение PL/SQL. Данная операция разрешена в выполняемом разделе и разделе исключительных ситуаций блока. Ниже приведен ряд примеров использования операции присваивания.

```
DECLARE
  v_String1  VARCHAR2(10);
  v_String2  VARCHAR2(15);
  v_Numeric  NUMBER;
BEGIN
  v_String1  :=  'Hello';
  v_String2  :=  v_String1;
  v_Numeric  :=  -12.4;
END;
```

Величина, находящаяся в левой части операции присваивания, называется именующим *значением* (lvalue), а величина, находящаяся в правой части, – значением *выражения* (rvalue). Именующее значение должно ссылаться на некоторую реальную область памяти, так как в нее будет записываться значение выражения. В примере, приведенном выше, все именующие значения являются переменными. PL/SQL выделяет для хранения переменных некоторую область памяти, и значения **'Hello'** и **-12.4** могут быть размещены в этой области. Значение выражения может представлять собой содержимое области памяти, на которую ссылается переменная или литерал. В нашем примере проиллюстрированы оба случая: **'Hello'** – это литерал, а **v_String1** – переменная. Значение выражения будет считываться, а именующее значение – записываться. Кроме того, все именующие значения являются и значениями выражений.

В отличие от других языков программирования, например C, в любом операторе (statement) PL/SQL может быть только одна операция присваивания. Таким образом, следующая операция присваивания неверна:

```
DECLARE
  v_Val1  NUMBER;
  v_Val2  NUMBER;
  v_Val3  NUMBER;
BEGIN
  v_Val1  :=  v_Val2  :=  v_Val3  :=  0;
END;
```

# Выражения

Выражения PL/SQL используются как значения выражений (rvalues), поэтому само по себе как оператор выражение бессмысленно: оно должно быть элементом другого оператора. Например, выражение может находиться в правой части операции присваивания или быть элементом SQL-оператора. Тип выражения определяют операции, составляющие это выражение, а также тип их операндов.

*Операнд* – это аргумент операции. В операциях PL/SQL используется либо один аргумент (унарная, или одноместная, операция), либо два аргумента (бинарная, или двухместная, операция). Например, операция отрицания (-) является унарной, а операция умножения (*) – бинарной.

Приоритет операций выражения определяет порядок их выполнения. Рассмотрим такое числовое выражение:

```
3 + 5 * 7
```

Приоритет операции умножения выше приоритета операции сложения, поэтому результатом этого выражения будет 38 (3+35), а не 56 (8*7). Чтобы изменить в выражении порядок приоритетов, заданный по умолчанию, следует использовать круглые скобки. Например, результатом приведенного ниже выражения будет 56:

```
(3 + 5) * 7
```

## Символьные выражения

Существует лишь одна символьная операция – это операция конкатенации, или сцепления (||). При помощи данной операции соединяются две строки символов (или аргументы, которые могут быть неявно преобразованы в строки символов). Например, результатом выражения

```
'Hello' || 'World' || '!'
```

будет

```
'Hello World !'
```

Если все операнды в выражении конкатенации имеют тип CHAR, то и оно само имеет тип CHAR. Если же хотя бы один операнд имеет тип VARCHAR2, выражение имеет тип VARCHAR2. Считается, что строковые литералы имеют тип CHAR, поэтому и результатом предыдущего примера является значение типа CHAR. Однако в блоке, рассмотренном ниже, переменной **v_Result** присваивается выражение, результат которого -- значение типа VARCHAR2:

```
DECLARE
  v_TempVar  VARCHAR2(10)  :=  'PL';
  v_Result  VARCHAR2(20);
BEGIN
  v_Result  :=  v_TempVar  ||  '/SQL';
END;
```

## Логические выражения

Во всех управляющих структурах PL/SQL (за исключением GOTO) используются логические выражения, называемые также условиями. *Логическое*, или *булево*, выражение – это любое выражение, которое дает в результате логическое значение (TRUE (истина), FALSE (ложь) или NULL). Ниже приведен ряд логических выражений:

```
X  >  Y
NULL
(4  >  5)  OR  (-1  !=  Z)
```

В трех операциях – AND (и), OR (или) и NOT (не) – логические значения используются в качестве аргументов и возвращаются в качестве результатов. Возможные ситуации описаны в таблицах истинности, изображенных на рис. 2.5. С помощью этих операций реализуется стандартная трехзначная логика. AND возвращает TRUE только в том случае, если истинны оба операнда, OR возвращает FALSE только тогда, когда оба операнды ложны.

**Рис. 2.5.**
*Таблицы истинности*

| **NOT** | **TRUE** | **FALSE** | **NULL** |
|---|---|---|---|
| | FALSE | TRUE | NULL |

| **AND** | **TRUE** | **FALSE** | **NULL** |
|---|---|---|---|
| TRUE | TRUE | FALSE | NULL |
| FALSE | FALSE | FALSE | FALSE |
| NULL | NULL | FALSE | NULL |

| **OR** | **TRUE** | **FALSE** | **NULL** |
|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE |
| FALSE | TRUE | FALSE | NULL |
| NULL | TRUE | NULL | NULL |

NULL-значения усложняют логические выражения (напомним, что NULL – это пропущенное или неизвестное значение). Результатом выражения

```
TRUE  AND  NULL
```

является NULL, так как неизвестно, истинен ли второй операнд или нет. Более подробно об этом рассказано в разделе "NULL-условия" ниже в этой главе.

В операциях *сравнения*, или *отношения*, в качестве операндов используются числа, символы или данные, а возвращаются логические значения. Ниже приведена таблица, где описаны эти операции.

| Операция | Описание |
|---|---|
| = | Равно (равенство) |
| != | Не равно (неравенство) |
| < | Меньше |
| > | Больше |
| <= | Меньше или равно |
| >= | Больше или равно |

В операции IS NULL значение TRUE возвращается только тогда, когда операндом является NULL. NULL-значения не могут быть проверены на истинность при помощи операций отношения, так как любое выражение отношения с NULL в качестве операнда возвращает NULL.

Операция LIKE (подобие) применяется для сопоставления с некоторым образцом строк символов, подобно тому как это делается в регулярных выражениях системы Unix. Знак подчеркивания (_) соответствует только одному символу, а знак процента (%) — нулю и более символов. Приведенные ниже выражения возвращают истинные значения (TRUE):

```
'Scott' LIKE 'Sc%t'
'Scott' LIKE 'Sc_tt'
'Scott' LIKE '%'
```

В операции BETWEEN (между) операции <= и >= объединяются в одном выражении. Например, приведенное ниже выражение возвращает ложное значение (FALSE):

```
100 BETWEEN 110 AND 120
```

а это выражение возвращает TRUE:

```
100 BETWEEN 90 AND 110
```

Результатом операции IN (в) будет истинное значение, когда первый операнд содержится в наборе, определяемом вторым операндом. Например, результат этого выражения — FALSE:

```
'Scott' IN ('Mike', 'Pamela', 'Fred')
```

Если в наборе содержатся NULL-значения, они игнорируются, так как при сравнении некоторого значения с NULL-значением всегда будет возвращаться NULL.

# Управляющие структуры PL/SQL

В PL/SQL, как и в других языках программирования третьего поколения, имеются различные структуры, позволяющие управлять работой блока. Этими структурами являются условные операторы и циклы. Именно эти структуры совместно с переменными обеспечивают мощь и гибкость PL/SQL.

## IF-THEN-ELSE

Синтаксис оператора IF-THEN-ELSE (если-то-иначе):

```
IF логическое_выражение1 THEN
   последовательность_операторов1;
[ELSIF логическое_выражение2 THEN
   последовательность_операторов2;]
...
[ELSE
   последовательность_операторов3;]
END IF;
```

где *логическое_выражение* — любое выражение, результатом которого является логическое значение (см. предыдущий раздел "Логические выражения"). Условия ELSIF и ELSE необязательны, причем условий ELSIF может быть сколь угодно много. Например, ниже представлен блок, демонстрирующий использование оператора IF-THEN-ELSE с одним условием ELSIF и одним условием ELSE:

```
-- Этот пример содержится в файле if1.sql.
DECLARE
  v_NumberSeats rooms.number_seats%TYPE;
  v_Comment VARCHAR2(35);
BEGIN
  /* Выбираем число мест в аудитории, идентификатор которой 99999.
     Сохраняем результат в v_NumberSeats. */
  SELECT number_seats
      INTO v_NumberSeats
      FROM rooms
      WHERE room_id = 99999;
  IF v_NumberSeats < 50 THEN
      v_Comment := 'Fairly small';
  ELSIF v_NumberSeats < 100 THEN
      v_Comment := 'A little bigger';
  ELSE
      v_Comment := 'Lots of room';
  END IF;
END;
```

Функционирование приведенного блока происходит в точности так, как это указано с помощью ключевых слов. Если первое условие истинно, то выполняется первая последовательность операторов. В нашем случае первым условием является

```
v_NumberSeats < 50
```

а первая последовательность операторов

```
v_Comment := 'Fairly small';
```

Если число мест не менее 50, оценивается второе условие:

```
v_NumberSeats < 100
```

Если оно истинно, выполняется вторая последовательность операторов:

```
v_Comment := 'A little bigger';
```

Наконец, если число мест не менее 100, выполняется завершающая последовательность операторов:

```
v_Comment := 'Lots of room';
```

Каждая последовательность операторов выполняется только в случае истинности соответствующего условия.

В этом примере в каждой последовательности операторов имеется только один процедурный оператор. Однако вообще можно включать в такие последовательности сколь угодно большое число операторов (процедурных либо SQL-операторов). Следующий блок иллюстрирует эту возможность.

```
-- Этот пример содержится в файле if2.sql.
DECLARE
  v_NumberSeats rooms.number_seats%TYPE;
  v_Comment VARCHAR2(35);
BEGIN
  /* Выбираем число мест в аудитории, идентификатор которой 99999.
     Сохраняем результат в v_NumberSeats. */
  SELECT number_seats
      INTO v_NumberSeats
      FROM rooms
      WHERE room_id = 99999;
  IF v_NumberSeats < 50 THEN
      v_Comment := 'Fairly small';
```

```
        INSERT INTO temp_table (char_col)
          VALUES ('Nice and cozy');
    ELSIF v_NumberSeats < 100 THEN
        v_Comment := 'A little bigger';
        INSERT INTO temp_table (char_col)
          VALUES ('Some breathing room');
    ELSE
        v_Comment := 'Lots of room';
    END IF;
END;
```

▼ ВНИМАНИЕ Обратите внимание на правописание ELSIF — в этом слове отсутствует E и нет пробела. Такой синтаксис заимствован из языка программирования Ada.

## NULL-условия

*Последовательность операторов в операторе IF-THEN-ELSE выполняется только в случае истинности* соответствующего условия. Если результатом условия является FALSE или NULL, последовательность операторов не выполняется. В качестве примера рассмотрим два блока:

```
/* Блок 1 */
DECLARE
  V_Number1   NUMBER;
  V_Number2   NUMBER;
  V_Result    VARCHAR2(7);
BEGIN
  ...
  IF v_Number1 < v_Number2 THEN
      v_Result := 'Yes';
  ELSE
      v_Result := 'No';
  END IF;
END;

/* Блок 2 */
DECLARE
  V_Number1   NUMBER;
  V_Number2   NUMBER;
  V_Result    VARCHAR2(7);
BEGIN
  ...
  IF v_Number1 >= v_Number2 THEN
      v_Result := 'No';
  ELSE
      v_Result := 'Yes';
  END IF;
END;
```

Одинаково ли будут вести себя эти два блока? Предположим, что **v_Number1** = 3, а **v_Number2** = 7. В результате условие блока 1 (3 < 7) будет истинно и переменная **v_Result** будет установлена в 'Yes'. Условие блока 2 будет ложно, а переменная **v_Result** также будет установлена в 'Yes'. Для любых значений **v_Number1** и **v_Number2**, не являющихся NULL-значениями, рассмотренные блоки функционируют одинаково.

Теперь предположим, что **v_Number1** = 3, но **v_Number2** является NULL-значением. Что произойдет в этом случае? Условие блока 1 (3 < NULL) дает в результате NULL, поэтому будет выполнено другое условие — ELSE — и переменной **v_Result** будет присвоено значение 'No'. Условие блока 2 также дает в результате NULL, поэтому будет выполнено условие ELSE и переменной **v_Result** будет присвое-

но значение 'Yes'. Если одна из переменных **v_Number1** и **v_Number2** содержит NULL-значение, то блоки функционируют по-разному.

Можно сделать так, чтобы блоки функционировали одинаково, если ввести в них проверку на наличие NULL-значений:

```
/* Блок 1 */
DECLARE
   V_Number1   NUMBER;
   V_Number2   NUMBER;
   V_Result    VARCHAR2(7);
BEGIN
   ...
   IF v_Number1 IS NULL OR
         v_Number2 IS NULL THEN
      v_Result := 'Unknown'
   ELSIF v_Number1 < v_Number2 THEN
      v_Result := 'Yes';
   ELSE
      v_Result := 'No';
   END IF;
END;

/* Блок 2 */
DECLARE
   V_Number1   NUMBER;
   V_Number2   NUMBER;
   V_Result    VARCHAR2(7);
BEGIN
   ...
   IF v_Number1 IS NULL OR
         v_Number2 IS NULL THEN
      v_Result := 'Unknown'
   ELSIF v_Number1 >= v_Number2 THEN
      v_Result := 'No';
   ELSE
      v_Result := 'Yes';
   END IF;
END;
```

Условие **IS NULL** будет истинно только тогда, когда проверяемая переменная содержит NULL-значение. В противном случае условие будет ложно. После введения в рассмотренные блоки проверки на наличие NULL-значений переменная **v_Result** будет принимать значение 'Unknown' (неизвестное), если одна из других переменных содержит NULL-значение. Сравнение переменных **v_Number1** и **v_Number2** будет выполняться только в том случае, если абсолютно точно известно, что обе переменные не содержат NULL-значений; при этом блоки (после проверки) будут функционировать одинаково.

## Циклы

В PL/SQL имеется возможность повторения операторов посредством *циклов* (loops). Циклы подразделяются на четыре категории. Простые циклы, циклы WHILE и циклы FOR рассматриваются в последующих разделах. Курсорные циклы FOR обсуждаются в главе 6.

### Простые циклы

Синтаксис простых циклов (основных циклов языка) таков:

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

*Последовательность_операторов* будет выполняться бесконечно долго, так как в этом цикле отсутствует условие его завершения. Такое *условие* можно предусмотреть, если добавить оператор EXIT (выход), имеющий следующий синтаксис (WHEN – когда):

EXIT [WHEN *условие*]

Ниже рассмотрен блок, с помощью которого в таблицу **temp_table** вводится 50 строк.

```
-- Этот пример содержится в файле simple.sql.
DECLARE
  v_Counter BINARY_INTEGER := 1;
BEGIN
  LOOP
      -- Введем в таблицу temp_table строку с текущим значением
      -- счетчика цикла.
      INSERT INTO temp_table
        VALUES (v_Counter, 'Loop index');
      v_Counter := v_Counter + 1;
      -- Условие выхода – когда счетчик цикла > 50, цикл будет завершен.
      IF v_Counter > 50 THEN
        EXIT;
      END IF;
  END LOOP;
END;
```

Оператор

EXIT WHEN *условие*

эквивалентен оператору:

IF *условие* THEN
  EXIT;
END IF;

Поэтому можно изменить рассмотренный блок, но его функционирование останется прежним:

```
-- Этот пример содержится в файле exitwhen.sql.
DECLARE
  v_Counter BINARY_INTEGER := 1;
BEGIN
  LOOP
      -- Введем в таблицу temp_table строку с текущим значением
      -- счетчика цикла.
      INSERT INTO temp_table
        VALUES (v_Counter, 'Loop index');
      v_Counter := v_Counter + 1;
      -- Условие выхода – когда счетчик цикла > 50, цикл будет завершен.
      EXIT WHEN v_Counter > 50
  END LOOP;
END;
```

## Циклы WHILE

Синтаксис цикла WHILE (цикла с условием продолжения) таков:

WHILE *условие* LOOP
  *последовательность_операторов*;
END LOOP;

Проверка *условия* происходит перед каждой итерацией (шагом) цикла. Если условие истинно, выполняется *последовательность_операторов*. Если же проверка *условия* дает ложное или NULL-значение, цикл завершается и управление программой передается оператору, следующему за оператором END LOOP. Теперь перепишем рассматриваемый блок при помощи цикла WHILE:

```
-- Этот пример содержится в файле while1.sql.
DECLARE
  v_Counter BINARY_INTEGER := 1;
BEGIN
  -- Проверяем счетчик цикла перед каждой итерацией цикла,
  -- чтобы гарантировать, что значение счетчика меньше 50.
  WHILE v_Counter <= 50 LOOP
      INSERT INTO temp_table
        VALUES (v_Counter, 'Loop index');
      v_Counter := v_Counter + 1;
  END LOOP;
END;
```

Чтобы прервать цикл и выйти из него, можно внутри цикла WHILE использовать операторы EXIT и EXIT WHEN.

Учтите, что, если при первой проверке условие цикла не истинно, цикл не выполняется вообще. Если в нашем примере убрать инициализацию переменной **v_Counter**, то результатом проверки условия **v_Counter <= 50** будет NULL-значение и в таблицу temp_table не будет введено ни одной строки:

```
-- Этот пример содержится в файле while2.sql.
DECLARE
  v_Counter BINARY_INTEGER ;
BEGIN
  -- Результатом проверки условия будет NULL-значение, так как по
  -- умолчанию значение счетчика v_Counter является NULL-значением.
  WHILE v_Counter <= 50 LOOP
      INSERT INTO temp_table
        VALUES (v_Counter, 'Loop index');
      v_Counter := v_Counter + 1;
  END LOOP;
END;
```

## Числовые циклы FOR

Число итераций в простых циклах и циклах WHILE заранее не известно — оно зависит от условий, заданных в циклах. В числовых же циклах FOR число итераций заранее определено. Синтаксис цикла FOR таков:

FOR *счетчик_цикла* IN [REVERSE] *нижняя_граница .. верхняя_граница* LOOP
  *последовательность_операторов*
END LOOP;

где *счетчик_цикла* — неявно создаваемая индексная переменная, *нижняя_граница* и *верхняя_граница* указывают число итераций, а *последовательность_операторов* является содержимым цикла.

Границы цикла указываются один раз и определяют общее число итераций, проходимых *счетчиком_цикла* от *нижней_границы до верхней_границы*. При этом счетчик каждый раз увеличивается на 1 до тех пор, пока цикл не завершится. Представим наш пример цикла с помощью цикла FOR:

```
-- Этот пример содержится в файле forloop.sql.
BEGIN
  FOR v_Counter IN 1..50 LOOP
      INSERT INTO temp_table
        VALUES (v_Counter, 'Loop Index');
  END LOOP;
END;
```

**Правила, определяющие область действия счетчика** Счетчик (индекс) цикла FOR неявно объявляется с типом BINARY_INTEGER. Объявлять его перед циклом необязательно. Если он все же объявлен, цикл скрывает это внешнее объявление так же, как объявление переменной во внутреннем блоке скрывает ее объявление во внешнем блоке:

```
-- Этот пример содержится в файле forscope.sql.
DECLARE
  v_Counter NUMBER := 7;
BEGIN
  -- Введем значение 7 в таблицу temp_table.
  INSERT INTO temp_table (num_col)
      VALUES (v_Counter);
  -- Этот цикл переобъявляет v_Counter как BINARY_INTEGER, что
  -- скрывает объявление v_Counter как NUMBER.
  FOR v_Counter IN 20..30 LOOP
      -- Внутри цикла v_Counter изменяется от 20 до 30.
      INSERT INTO temp_table (num_col)
        VALUES (v_Counter);
  END LOOP;
  -- Еще раз введем значение 7 в таблицу temp_table.
  INSERT INTO temp_table (num_col)
      VALUES (v_Counter);
END;
```

**Использование REVERSE** Если в цикле FOR указывается ключевое слово REVERSE (обратный порядок), индекс цикла будет изменяться от верхней границы до нижней. Обратите внимание, что в этом случае синтаксис остался прежним — нижняя граница по-прежнему указывается первой:

```
BEGIN
  FOR v_Counter IN REVERSE 10..50 LOOP
      -- v_Counter начнется с 50 и каждый раз будет уменьшаться на 1.
      NULL;
  END LOOP;
END;
```

**Диапазоны циклов** Верхняя и нижняя границы необязательно должны быть числовыми литералами. Они могут быть любыми выражениями, для которых возможно преобразование в числовые значения. Приведем пример:

```
DECLARE
  v_LowValue NUMBER := 10;
  v_HighValue NUMBER := 40;
BEGIN
  FOR v_Counter IN REVERSE v_LowValue .. v_HighValue LOOP
      INSERT INTO temp_table
        VALUES (v_Counter, 'Dynamically specified loop ranges');
  END LOOP;
END;
```

# Операторы GOTO и метки

В языке программирования PL/SQL применяются операторы GOTO (перейти к). Синтаксис оператора GOTO таков:

GOTO *метка*

где *метка* — это метка, определяемая в блоке PL/SQL. Метки заключаются в двойные угловые скобки. При выполнении оператора GOTO управление программой сразу же передается оператору, на который указывает метка. Представим рассматриваемый пример цикла следующим образом:

```
-- Этот пример содержится в файле goto.sql.
DECLARE
  v_Counter BINARY_INTEGER := 1;
```

```
BEGIN
   LOOP
      INSERT INTO temp_table
        VALUES (v_Counter, 'Loop count');
      v_Counter := v_Counter + 1;
      IF v_Counter > 50 THEN
        GOTO l_EndOfLoop;
      END IF;
   END LOOP;

   <<l_EndOfLoop>>
   INSERT INTO temp_table (char_col)
      VALUES ('Done!');
END;
```

## Ограничения при использовании GOTO

В PL/SQL на использование операторов GOTO налагаются определенные ограничения. Нельзя передавать управление программой во внутренний блок, цикл или оператор IF. Это иллюстрирует приведенный ниже пример:

```
BEGIN
   GOTO l_InnerBlock;     -- неверно, так как нельзя передавать
                          -- управление программой во внутренний блок
   BEGIN
      ...
      <<l_InnerBlock>>
      ...
   END;

   GOTO l_InsideIf;     -- неверно, так как нельзя передавать
                        -- управление программой в оператор IF
   IF x > 3 THEN
      ...
      <<l_InsideIf>>
      INSERT INTO ...
   END IF;
END;
```

Если бы это было разрешено, операторы внутри оператора IF могли бы быть выполнены, даже если бы условие IF не было истинным. В примере, приведенном выше, оператор INSERT выполнился бы даже тогда, когда x = 2.

Кроме того, запрещается передавать управление программой из одной последовательности операторов условного оператора IF в другую:

```
BEGIN
   IF x > 3 THEN
      ...
      GOTO l_NextCondition;
   ELSE
      <<l_NextCondition>>
      ...
   END IF;
END;
```

Наконец, нельзя передавать управление из обработчика исключительной ситуации обратно в текущий блок. Исключительные ситуации рассмотрены в главе 10.

```
DECLARE
  v_Room rooms%ROWTYPE;
BEGIN
  -- Выберем одну строку в таблице rooms.
  SELECT *
      INTO v_Room
      FROM rooms
      WHERE rowid = 1;
  <<l_Insert>>
  INSERT INTO temp_table (char_col)
      VALUES ('Found a row!');
EXCEPTION
  WHEN NO_DATA_FOUND THEN
      GOTO l_Insert;     -- неверно, так как передавать управление в
                         -- текущий блок нельзя
END;
```

## Помеченные циклы

Помечены могут быть и сами циклы. При этом метка может быть указана в операторе EXIT, чтобы определить цикл, который нужно прервать. Например:

```
BEGIN
  <<l_Outer>>
  FOR v_OuterIndex IN 1..50 LOOP
      ...
      <<l_Inner>>
  FOR v_InnerIndex IN 2..10 LOOP
      ...
      IF v_OuterIndex > 40 THEN
        EXIT l_Outer; -- выход из обоих циклов
      END IF;
  END LOOP l_Inner;
END LOOP l_Outer;
```

Если цикл помечен, имя метки можно указывать после оператора END LOOP, как показано в этом примере.

## Рекомендации по использованию GOTO

При использовании операторов GOTO следует соблюдать осторожность. Лишние операторы GOTO могут привести к нарушению структуры программы: управление программой может перемещаться с места на место без видимой причины, что затруднит ее понимание и сопровождение.

Все случаи использования операторов GOTO могут быть записаны с помощью других управляющих структур PL/SQL, например циклов или условных выражений. Для выхода из цикла, имеющего множество вложений, можно вместо оператора перехода к концу блока воспользоваться исключительной ситуацией.

## NULL как оператор

В некоторых случаях необходимо явно указывать, что никаких действий выполнять не нужно. Это можно сделать при помощи оператора NULL, который не приводит ни к каким действиям; он служит лишь в качестве заполнителя. Например:

```
-- Этот пример содержится в файле null.sql.
DECLARE
  v_TempVar NUMBER := 7;
BEGIN
  IF v_TempVar < 5 THEN
      INSERT INTO temp_table (char_col)
```

```
        VALUES ('Too small');
    ELSIF v_TempVar < 10 THEN
        INSERT INTO temp_table (char_col)
          VALUES ('Just right');
    ELSE
        NULL; -- никаких действий не выполняется
    END IF;
END;
```

## Прагмы

*Прагмы* (pragmas) — это директивы компилятора, подобные директивам **#pragma** или **#define** языка C. Они служат в качестве инструкций компилятору PL/SQL, который выполняет прагму во время компиляции блока. Например, прагма RESTRICT_REFERENCES (ограничить ссылки) устанавливает ограничения на способ использования SQL-операторов в некоторой функции. Компилятор, помимо компиляции функции как нормальной, должен проверить, какие установлены ограничения. Прагма RESTRICT_REFERENCES рассмотрена в главе 8. В PL/SQL имеется ряд прагм, которые будут встречаться на протяжении всей этой книги.

Прагмы — это еще одно средство, которое является общим для языков PL/SQL и Ada.

# Стиль программирования на PL/SQL

Для стиля программирования не существует каких-либо незыблемых правил. Стиль программирования определяют такие моменты, как имена переменных, использование заглавных букв и разделителей, а также комментариев. Все эти элементы не оказывают воздействия на функционирование программ — программы, написанные в разном стиле, могут выполнять одну и ту же задачу. Однако грамотную и в хорошем стиле составленную программу намного легче понимать и сопровождать, чем ту, которая написана небрежно.

Хороший стиль означает, что на понимание смысла программы будет затрачено достаточно мало времени. Кроме того, программа должна быть понятной как во время ее написания, так и месяц спустя.

В качестве примера рассмотрим два блока и выясним, какой из них легче для понимания.

```
declare
x number;
y number;
begin if x < 10 then y := 7; else y := 3; end if; end;

DECLARE
  v_Test NUMBER;      -- Переменная, которая будет анализироваться
  v_Result NUMBER;    -- Переменная для хранения результата
BEGIN
  -- Проанализируем v_Test и присвоим v_Result значение 7, если
  -- v_Test < 10.
  IF v_Test < 10 THEN
      v_Result := 7;
  ELSE
      v_Result := 3;
  END IF;
END;
```

Оба блока выполняют одну и ту же задачу, однако ход программы во втором блоке намного легче для понимания.

В этом разделе рассматривается несколько моментов, касающихся стиля программирования. Для написания хороших программ рекомендуется придерживаться предлагаемых советов. Все приведенные в книге примеры составлены с учетом этих рекомендаций и служат иллюстрациями стиля программирования на языке PL/SQL.

## Комментарии

Комментарии – это основной способ информирования читателя о том, каково назначение программы и как она работает. Автор рекомендует располагать комментарии:

- В начале каждого блока и/или процедуры. В комментариях можно объяснить, для чего предназначен блок (процедура). Это особенно важно для процедур, так как в комментариях можно указать, какие параметры будут считаны процедурой (входные параметры) и какие переменные или параметры будут записаны (выходные параметры). Кроме того, рекомендуется указывать таблицы базы данных, к которым обращается программа.
- При каждом объявлении переменной. Сообщайте, для чего она будет использоваться. Часто достаточно однострочного комментария, например:

```
V_SSN CHAR(11); -- Social Security Number
```

(номер службы социального обеспечения США. – *Прим. пер.*)

- Перед каждым важным разделом блока. Не нужно сопровождать комментариями каждый оператор, однако весьма полезен комментарий, поясняющий назначение группы операторов. Используемый алгоритм может не во всем совпадать с текстом программы, поэтому рекомендуется описывать не применяемый метод, а назначение данной группы операторов.

Вполне возможно, что комментариев в программе будет слишком много. При принятии решения о необходимости комментария спрашивайте себя: "Что хочет знать программист, видящий это в первый раз?" Учтите, что таким программистом можете стать вы сами через месяц-другой после написания программы!

Комментарий должен иметь определенный смысл и не повторять того, что написано непосредственно в программе. К примеру, ниже приведен комментарий, который не сообщает ничего нового по сравнению с тем, что написано в программе PL/SQL, и поэтому практически бесполезен:

```
DECLARE
   v_Temp NUMBER := 0; - Присвоить v_Temp значение 0
```

Однако следующий комментарий полезен, так как в нем сообщается о назначении переменной **v_Temp**:

```
DECLARE
   v_Temp NUMBER := 0; - Временная переменная, используемая в основном цикле
```

## Имена переменных

Именам переменных следует придавать определенный смысл. Объявление

```
x number;
```

ничего не говорит нам о том, в каких целях будет использоваться переменная x. Однако объявление

```
v_StudentID NUMBER(5);
```

поясняет, что данная переменная, скорее всего, будет использоваться для описания идентификационного номера студента, хотя никакого комментария при этом не приведено. Не забывайте, что максимальная длина идентификатора PL/SQL равна 30 символам и каждый из них является значащим. Для создания достаточно информативного имени обычно тридцати символов бывает вполне достаточно.

В имени переменной может быть сказано и о ее использовании. Автор предлагает такой способ именования переменных: буква, отделенная от остальной части знаком подчеркивания. Например:

```
V_VariableName    программная переменная
E_ExceptionName   исключительная ситуация, определяемая пользователем
T_TypeName        тип, определяемый пользователем
P_ParameterName   параметр процедуры или функции
C_ConstantValue   переменная, ограниченная как CONSTANT (константа)
```

## Выделение заглавными буквами

PL/SQL не учитывает регистра. Однако автор считает, что надлежащее использование прописных и строчных букв существенно повышает удобочитаемость программ. Рекомендуется придерживаться следующих правил:

- Зарезервированные слова пишутся прописными буквами (например, BEGIN, DECLARE, ELSIF).

- Встроенные функции пишутся прописными буквами (SUBSTR, COUNT, TO_CHAR).
- Предварительно определенные типы пишутся прописными буквами (NUMBER(7,2), BOOLEAN, DATE).
- Ключевые слова SQL пишутся прописными буквами (SELECT, INTO, UPDATE, WHERE).
- Объекты базы данных пишутся строчными буквами (log_table, classes, students).
- Имена переменных пишутся с использованием символов обоих регистров; каждое слово в имени пишется с заглавной буквы (v_HireDate, e_TooManyStudents, t_StudentRecordType).

## Структурирование текста

Одним из наиболее простых способов, применяемых для структурирования текста, является использование разделителей (символов возврата каретки, пробела и табуляции). Это положительно влияет на повышение удобочитаемости программ. Сравним две одинаковые вложенные конструкции типа IF-THEN-ELSE:

```
IF x < y THEN IF z IS NULL THEN x := 3; ELSE x := 2; END IF; ELSE
x := 4; END IF;
```

```
IF x < y THEN
  IF z IS NULL THEN
     x := 3;
  ELSE
     x := 2;
  END IF;
ELSE
  x := 4;
END IF;
```

Автор использует следующие правила: начинать каждую строку в блоке с двух пробелов; выделять отступом содержимое блока по отношению к ключевым словам DECLARE..END; выделять отступом циклы и условные операторы. Кроме того, автор выделяет также и SQL-операторы, которые располагаются на нескольких строках, например:

```
SELECT id, first_name, last_name
  INTO v_StudentID, V_FirstName, v_LastName
  FROM STUDENTS
  WHERE id = 10002;
```

## Общий стиль

По мере написания программ на PL/SQL вам, скорее всего, придется вырабатывать собственный стиль программирования. Приведенные рекомендации совершенно необязательны, однако автор подчеркивает, что для него они оказались достаточно полезными и поэтому он использует их в примерах. Покажите текст своей программы другому программисту и спросите его о ее назначении. Если он сможет рассказать, как работает программа, значит, она составлена вполне грамотно и стиль ее достаточно хорош.

Кроме того, во многих организациях, занимающихся разработкой программ, приняты собственные правила по их стилю и документированию, причем эти правила могут применяться к языкам программирования, отличным от PL/SQL. Однако их вполне реально применить и в PL/SQL; если ранее был выработан какой-либо стиль программирования на языке C, то, скорее всего, этот стиль можно приспособить и для PL/SQL.

# Итоги

В этой главе говорилось об основных элементах PL/SQL: о структуре блока PL/SQL, о переменных и типах данных (скалярных, составных и ссылочных), о выражениях и операциях, о правилах преобразования типов данных и об основных управляющих структурах. Кроме того, разговор шел о стиле программирования на PL/SQL, который помогает писать более понятные и удобочитаемые программы. Теперь, когда выяснены общие концепции языка, можно перейти к анализу составных типов (глава 3) и к добавлению SQL-операторов к процедурным конструкциям, рассмотренным ниже (глава 4).

# Глава 3

## *Записи и таблицы*

В предшествующей главе было сказано о скалярных типах PL/SQL. Эти типы определены предварительно и являются основой всей системы типов языка. В этой главе обсуждаются составные типы, получившие применение в PL/SQL версии 2: записи и таблицы. Объектные типы рассматриваются в главе 11, а составные типы, применяемые в PL/SQL версии 8.0, – в главе 12.

# Записи PL/SQL

Скалярные типы (NUMBER, VARCHAR2, DATE и т.д.) предварительно определены в модуле STANDARD. Поэтому для использования одного из таких типов в программе требуется лишь объявить переменную этого типа. Составные же типы определяются пользователями. Чтобы применить составной тип, нужно вначале его описать и лишь затем объявить переменную этого типа. Посмотрим, как это можно сделать.

Записи (records) PL/SQL аналогичны структурам языка C. С помощью записи можно работать с несколькими отдельными, но связанными переменными как с одной программной единицей. Рассмотрим следующий раздел объявлений:

```
DECLARE
    v_StudentID  NUMBER(5);
    v_FirstName  VARCHAR2(20);
    v_LastName  VARCHAR2(20);
```

Все эти три переменные логически связаны, так как они имеют отношение к общим полям таблицы **students**. Когда для этих переменных объявляется тип записи, существующая между ними взаимосвязь становится очевидной, и переменными можно управлять как одной единицей. Рассмотрим пример:

```
DECLARE
    /* Объявим тип записи для хранения общей информации о студентах. */
    TYPE t_StudentRecord IS RECORD (
        StudentID  NUMBER(5),
        FirstName  VARCHAR2(20),
        LastName  VARCHAR2(20));

    /* Объявим переменную с этим типом. */
    v_StudentInfo t_StudentRecord;
```

Общий синтаксис описания типа записи таков:

```
TYPE тип_записи IS RECORD (
    поле1 тип1 [NOT NULL] [:= выражение1],
    поле2 тип2 [NOT NULL] [:= выражение2],
    ...
    полеn типn [NOT NULL] [:= выраженияn]);
```

Здесь *тип_записи* – это имя нового типа, *поле1* ... *полеn* – имена полей записи, а *тип1* ... *типn* – типы соответствующих полей. В записи может быть сколь угодно много полей. Описание каждого поля в принципе аналогично описанию переменных вне записи. Сходство этих описаний состоит еще и в том, что для полей тоже можно указывать ограничения NOT NULL и исходные значения. *выражение1* ... *выражениеn* представляют исходные значения для каждого поля. Подобно описанию переменной вне записи, исходное значение и ограничение NOT NULL для поля необязательны. Ниже приведен раздел объявлений, в котором сначала определяется тип записи **t_SampleRecord**, а затем объявляются две переменные с этим типом.

```
DECLARE
    TYPE t_SampleRecord IS RECORD (
        Count           NUMBER(4),
        Name            VARCHAR2(10)  :=  'Scott',
        EffectiveDate   DATE,
        Description     VARCHAR2(45)  NOT  NULL  :=  'Unknown');
        v_Sample1  t_SampleRecord;
        v_Sample2  t_SampleRecord;
```

Если поле ограничено как NOT NULL, то для него должно быть указано исходное значение. Любое поле без исходного значения инициализируется как NULL. Для указания значения по умолчанию можно использовать как ключевое слово DEFAULT, так и знак :=.

Для ссылки на поле некоторой записи используется уточняющая запись через точку. Ее синтаксис таков:

*имя_записи.имя_поля*

Ниже показано, как можно обратиться к некоторым полям в **v_Sample1** и **v_Sample2**:

```
BEGIN
   /* SYSDATE - это встроенная функция, которая возвращает текущие
      дату и время. */
   v_Sample1.EffectiveDate := SYSDATE;
   v_Sample2.Description := 'Pesto Pizza';
END;
```

Такая ссылка является именующим выражением (lvalue), поэтому ее можно указывать с обеих сторон операции присваивания.

## Присваивание записей

Чтобы присвоить одной записи значение другой, они должны быть одного типа. Далее приведен пример правильной операции присваивания, в которой используются записи v_Sample1 и v_Sample2, рассмотренные выше.

```
v_Sample1 := v_Sample2
```

При выполнении подобных операций присваивания используется так называемая *семантика копирования* – значениям полей **v_Sample1** присваиваются значения соответствующих полей **v_Sample2**. Нельзя присваивать значения записей, имеющих разные типы, даже когда описания их полей совпадают. Ниже приведен пример, который неверен, и в результате его выполнения возвращается сообщение об ошибке "PLS-382: expression is of wrong type" ("тип выражения неверен").

```
-- Этот пример содержится в файле assign.sql.
DECLARE
  TYPE t_Rec1Type IS RECORD (
      Field1 NUMBER,
      Field2 VARCHAR2(5));
  TYPE t_Rec2Type IS RECORD (
      Field1 NUMBER,
      Field2 VARCHAR2(5));
  v_Rec1 t_Rec1Type;
  v_Rec2 t_Rec2Type;
BEGIN
  /* Хотя записи v_Rec1 и v_Rec2 имеют одинаковые имена и типы полей,
     типы собственно записей различны. Такая операция присваивания
     неверна, и ее выполнение приводит к установлению флага ошибки
     PLS-382. */
  v_Rec1 := v_Rec2;

  /* Однако типы полей совпадают, поэтому следующие операции
     присваивания правильны. */
  v_Rec1.Field1 := v_Rec2.Field1;
  v_Rec2.Field2 := v_Rec2.Field2;
END;
```

Присвоить записи некоторое значение можно также с помощью оператора SELECT. При этом данные будут выбираться в базе данных и сохраняться в записи. Поля записи должны соответствовать полям, указываемым в списке выбора запроса. (Оператор SELECT более подробно описан в главе 4.) Проиллюстрируем это на примере:

```
-- Этот пример содержится в файле select.sql.
DECLARE
  -- Объявим запись так, чтобы ее поля соответствовали некоторым
  -- полям таблицы students. Обратите внимание на использование
  -- для полей атрибута %TYPE.
  TYPE t_StudentRecord IS RECORD (
      FirstName  students.first_name%TYPE,
      LastName   students.last_name%TYPE,
      Major      students.major%TYPE);

  -- Объявим переменную, предназначенную для получения данных.
  v_Student t_StudentRecord;
BEGIN
  -- Выберем информацию о студенте, чей идентификатор равен 10000.
  -- Обратите внимание, что запрос возвращает поля,
  -- соответствующие полям v_Student.
  SELECT first_name, last_name, major
      INTO v_Student
      FROM students
      WHERE ID = 10000;
END;
```

## Использование %ROWTYPE

В PL/SQL широко используется объявление записей с теми же типами, которые имеют строки базы данных. Для облегчения этого процесса в PL/SQL предлагается операция %ROWTYPE. Как и %TYPE, %ROWTYPE будет возвращать тип на основании описания таблицы. Например, описание

```
DECLARE
  v_RoomRecord  rooms%ROWTYPE
```

определяет запись, поля которой соответствуют полям столбцов таблицы **rooms.** Конкретно запись **v_RoomRecord** будет выглядеть следующим образом:

```
(Room_id       NUMBER(5),
Building       VARCHAR2(15),
Room_number    NUMBER(4),
Number_seats   NUMBER(4),
Description    VARCHAR2(50))
```

Как и в случае с %TYPE, ограничение NOT NULL для столбцов отсутствует. Однако указана длина полей столбцов VARCHAR2 и CHAR, а также точность и масштаб полей столбцов NUMBER.

Если описание таблицы изменяется, %ROWTYPE также изменяется. Как и %TYPE, %ROWTYPE выполняется всякий раз, когда анонимный блок передается PL/SQL и когда компилируется хранимый объект.

# Таблицы

Таблицы PL/SQL аналогичны массивам языка C. Синтаксически они и представляют собой массивы, однако реализованы по-другому. Чтобы объявить таблицу PL/SQL, сначала нужно определить ее тип, а затем объявить переменную данного типа. Это иллюстрирует следующий раздел объявлений:

```
DECLARE
  /* Определим тип таблицы. Переменные этого типа могут содержать
     последовательности символов, причем максимальная длина каждой
     последовательности равна 10 символам. */
  TYPE t_CharacterTable IS TABLE OF VARCHAR2(10)
```

```
    INDEX BY BINARY_INTEGER;

/* Объявим переменную этого типа и тем самым выделим фактическое
   место хранения информации. */
v_Characters t_CharacterTable
```

Общий синтаксис описания таблицы таков:

TYPE *тип_таблицы* IS TABLE OF *тип* INDEX BY BINARY_INTEGER;

где *тип_таблицы* – имя нового типа, а *тип* – ранее определенный скалярный тип или ссылка на скалярный тип посредством %TYPE. В предыдущем примере тип_таблицы – это **t_CharacterTable**, а тип – **VARCHAR2(10)**. Раздел объявлений, приведенный ниже, иллюстрирует описание ряда различных типов таблиц и переменных PL/SQL.

```
DECLARE
  TYPE t_NameTable IS TABLE OF students.first_name%TYPE
      INDEX BY BINARY_INTEGER;
  TYPE t_DateTable IS TABLE OF DATE
      INDEX BY BINARY_INTEGER;
  v_Names t_NameTable;
  v_Dates t_DateTable;
```

▼ ВНИМАНИЕ В PL/SQL версии 2 конструкция **INDEX BY BINARY_INTEGER** обязательна при описании таблицы. Для таблиц, используемых в версии 8, эта конструкция необязательна. Такие таблицы обсуждаются в главе 12.

После того как объявлены тип и переменная, можно ссылаться на отдельные элементы таблицы PL/SQL следующим образом:

*имя_таблицы(индекс)*

где *имя_таблицы* – это имя таблицы, а *индекс* – либо переменная, имеющая тип BINARY_INTEGER, либо переменная или выражение, которые могут быть преобразованы к типу BINARY_INTEGER. Продолжим наш блок PL/SQL:

```
BEGIN
  v_Names(1) := 'Scott';
  v_Dates(-4) := SYSDATE - 1; /* SYSDATE -1 дает в результате
                                 дату и время, бывшие 24 часа назад. */
END;
```

Обратите внимание, что ссылка на таблицу, как и ссылка на запись или переменную, является именующим выражением (lvalue), так как она указывает на область памяти, выделенной PL/SQL.

## Таблицы и массивы

Синтаксически таблицы PL/SQL представляют собой массивы. Однако фактическая реализация таблицы отличается от реализации массива. Таблица PL/SQL схожа с таблицей базы данных и содержит два столбца – KEY (ключ) и VALUE (значение). Тип ключа – BINARY_INTEGER, а тип значения – это тип, указанный в описании (тип в синтаксисе описания таблицы, приведенном выше).

С учетом описания **t_NameTable** и **v_Names** в разделе объявлений, приведенном выше, предположим, что выполняется следующая последовательность операторов:

```
v_Names(0) := 'Harold';
v_Names(-7) := 'Susan';
v_Names(3) := 'Steve';
```

Структура данных будет выглядеть так:

| Ключ | Значение |
|---|---|
| 0 | Harold |
| -7 | Susan |
| 3 | Steve |

При работе с рассмотренными таблицами PL/SQL следует обратить внимание на следующее:

- Число строк таблицы ничем не ограничено. Единственное ограничение – это значения, которые могут быть представлены типом BINARY_INTEGER.
- Порядок элементов таблицы PL/SQL необязательно должен быть строго определен. Эти элементы хранятся в памяти не подряд, как массивы, и поэтому могут вводиться с произвольными ключами.
- Ключи, используемые в таблице PL/SQL, необязательно должны быть последовательными. В качестве индекса таблицы может быть использовано любое значение или выражение, имеющее тип BINARY_INTEGER.

*Присваивание значения i-му элементу таблицы PL/SQL создает этот элемент. Это очень похоже на операцию ввода INSERT, выполняемую над таблицей базы данных. Если ссылка на i-й элемент производится до его создания, PL/SQL возвращает сообщение об ошибке:*

```
ORA-1403: no data found
```

(данные не найдены. – *Прим. пер.*)

Память таблице выделяется по мере ввода данных и возрастает с увеличением числа строк таблицы. Память не зависит от того, какие значения используются для ключа.

**PL/SQL 2.3 ... и ВЫШЕ**

В версиях PL/SQL, предшествующих 2.3, в таблицах могут храниться данные, имеющие только скалярные типы. Однако в версии 2.3 эти ограничения сняты и разрешены таблицы, в состав которых входят записи. Ниже приведен блок, который можно использовать только в PL/SQL версии 2.3 и выше:

```
-- Этот пример содержится в файле tabrec.sql.
DECLARE
  TYPE t_StudentTable IS TABLE OF students%ROWTYPE
      INDEX BY BINARY_INTEGER;
  /* Каждый элемент v_Students является записью. */
  v_Students t_StudentTable;
BEGIN
  /* Выбираем запись с идентификатором 10001 и сохраняем ее в
     v_Students(10001). */
  SELECT *
      INTO v_Students(10001)
      FROM students
      WHERE id = 10001;
END;
```

Каждый элемент этой таблицы является записью, поэтому можно обращаться к полям данной записи следующим образом:

*таблица(индекс).поле*

К примеру, можно продолжить блок, рассмотренный выше:

```
v_Students(10001).first_name := 'Larry';
DBMS_OUTPUT.PUT_LINE(v_Students(10001).first_name);
```

Здесь **v_Students(10001)** является записью типа **students**%ROWTYPE, а **first_name** – поле этой записи. Ссылки на запись и на поле разделяет точка.

**▼ ВНИМАНИЕ** В предыдущем примере используется процедура DBMS_OUTPUT.PUT_LINE. Она может применяться для вывода на экран результатов сеанса работы в SQL*Plus. Более подробно о DBMS_OUTPUT рассказано в главе 14.

Таблицы и записи существенно расширяют функциональные возможности таблиц PL/SQL, так как при этом для хранения информации обо всех полях таблицы базы данных требуется описать только одну таблицу. В версиях PL/SQL, предшествующих версии 2.3, для каждого поля базы данных необходимо описывать отдельную таблицу.

## Атрибуты таблиц

PL/SQL 2.3 ... и ВЫШЕ

Помимо того что в PL/SQL версии 2.3 разрешается использовать таблицы записей, эта версия расширяет функциональные возможности таблиц PL/SQL тем, что для них определяются атрибуты. Синтаксис использования атрибута таков:

*таблица.атрибут*

где *таблица* — это ссылка на таблицу PL/SQL, а *атрибут* — нужный атрибут. Атрибуты таблиц PL/SQL описаны в таблице 3.1 и в последующих разделах.

**ТАБЛИЦА 3.1. Атрибуты таблиц PL/SQL**

| Атрибут | Возвращаемый тип | Описание |
|---|---|---|
| COUNT | NUMBER | Возвращает число строк таблицы |
| DELETE | — | Удаляет строки таблицы |
| EXISTS | BOOLEAN | Возвращает TRUE, если указанный элемент находится в таблице |
| FIRST | BINARY_INTEGER | Возвращает индекс первой строки таблицы |
| LAST | BINARY_INTEGER | Возвращает индекс последней строки таблицы |
| NEXT | BINARY_INTEGER | Возвращает индекс строки таблицы, которая следует за указанной строкой |
| PRIOR | BINARY_INTEGER | Возвращает индекс строки таблицы, которая предшествует указанной строке |

### COUNT

Атрибут COUNT (число) возвращает текущее число строк таблицы PL/SQL. Рассмотрим следующий блок:

```
-- Этот пример содержится в файле count.sql.
DECLARE
  TYPE t_NumberTable IS TABLE OF NUMBER
      INDEX BY BINARY_INTEGER;
  v_Numbers t_NumberTable;
  v_Total NUMBER;
BEGIN
  -- Введем 50 строк в таблицу.
  FOR v_Counter IN 1..50 LOOP
      v_Numbers(v_Counter) := v_Counter;
  END LOOP;

  v_Total := v_Numbers.COUNT;
END;
```

**v_Numbers.COUNT** возвращает 50, и это значение присваивается переменной **v_Total**.

### DELETE

Атрибут DELETE (удалить) удаляет строки таблицы PL/SQL. Он используется следующим образом:

- *таблица*.DELETE — удаляются все строки таблицы
- *таблица*.DELETE(*i*) — удаляется строка таблицы с индексом *i*
- *таблица*.DELETE (*i*,*j*) — удаляются все строки таблицы с индексами от *i* до *j*

Проиллюстрируем вышесказанное на примере:

```
-- Этот пример содержится в файле delete.sql.
DECLARE
  TYPE t_ValueTable IS TABLE OF VARCHAR2(10)
      INDEX BY BINARY_INTEGER;
  v_Values t_ValueTable;
BEGIN
  -- Введем строки в таблицу.
  v_Values(1) := 'One';
  v_Values(3) := 'Three';
  v_Values(-2) := 'Minus Two';
  v_Values(0) := 'Zero';
  v_Values(100) := 'Hundred';

  DBMS_OUTPUT.PUT_LINE('Before DELETE, COUNT=' || v_Values.COUNT);
  v_Values.DELETE(100); -- удаляет 'Hundred'
  DBMS_OUTPUT.PUT_LINE('After first DELETE, COUNT=' ||
                       v_Values.COUNT);
  v_Values.DELETE(1,3); -- удаляет 'One' и 'Three'
  DBMS_OUTPUT.PUT_LINE('After second DELETE, COUNT=' ||
                       v_Values.COUNT);
  V_Values.DELETE;      -- удаляет все оставшиеся значения
  DBMS_OUTPUT.PUT_LINE('After last DELETE, COUNT=' ||
                       v_Values.COUNT);
END;
```

Обратите внимание, что атрибут DELETE сам по себе является оператором; он не вызывается в качестве части выражения, как другие атрибуты.

## EXISTS

При использовании атрибута EXISTS (существует) в виде таблица.EXISTS(*i*) истинное значение возвращается, когда индекс *i* существует в таблице, а ложное — когда не существует. Этот атрибут полезен для того, чтобы избежать возвращения ошибки ORA-1403, возникающей при ссылке на несуществующий элемент таблицы. Приведем пример:

```
-- Этот пример содержится в файле exists.sql.
DECLARE
  TYPE t_FirstNameTable IS TABLE OF students.first_name%TYPE
      INDEX BY BINARY_INTEGER;
  FirstNames t_FirstNameTable;
BEGIN
  -- Введем строки в таблицу.
  FirstNames(1) := 'Scott';
  FirstNames(3) := 'Joanne';

  -- Посмотрим, существуют ли строки.
  IF FirstNames.EXISTS(1) THEN
      INSERT INTO temp_table (char_col) VALUES
        ('Row 1 exists!');
  ELSE
      INSERT INTO temp_table (char_col) VALUES
        ('Row 1 doesn''t exist!');
  END IF;
  IF FirstNames.EXISTS(2) THEN
      INSERT INTO temp_table (char_col) VALUES
        ('Row 2 exists!');
  ELSE
```

```
        INSERT INTO temp_table (char_col) VALUES
          ('Row 2 doesn''t exist!');
      END IF;
    END;
```

После выполнения этого блока в таблицу **temp_table** будет введено "Row 1 exists!" ("Строка 1 существует!") и "Row 2 doesn't exist!" ("Строка 2 не существует!").

## FIRST и LAST

FIRST и LAST возвращают индексы соответственно первой (first) и последней (last) строк таблицы PL/SQL. Обратите внимание: они возвращают не значение, содержащееся в этих строках, а только индексы. Первая строка – это строка с минимальным индексом, а последняя – с максимальным. Приведем пример:

```
-- Этот пример содержится в файле frstlast.sql.
DECLARE
  TYPE t_LastNameTable IS TABLE OF students.last_name%TYPE
      INDEX BY BINARY_INTEGER;
  v_LastNames t_LastNameTable;
  v_Index BINARY_INTEGER;
BEGIN
  -- Введем строки в таблицу.
  v_LastNames(43) := 'Mason';
  v_LastNames(50) := 'Junebug';
  v_LastNames(47) := 'Taller';

  -- Присвоим значение 43 переменной v_Index.
  v_Index := v_LastNames.FIRST;

  -- Присвоим значение 50 переменной v_Index.
  v_Index := v_LastNames.LAST;
END;
```

## NEXT и PRIOR

В атрибутах NEXT (следующий) и PRIOR (предшествующий) используется один аргумент, как и в атрибуте DELETE. Они возвращают соответственно индекс следующего или предшествующего элемента таблицы. Эти атрибуты можно применять в цикле, который перебирает всю таблицу независимо от того, какие значения используются в индексах. Приведем пример:

```
-- Этот пример содержится в файле nxtprior.sql.
DECLARE
  TYPE t_MajorTable IS TABLE OF students.major%TYPE
      INDEX BY BINARY_INTEGER;
  v_Majors t_MajorTable;
  v_Index BINARY_INTEGER;
BEGIN
  -- Введем значения в таблицу.
  v_Majors(-7) := 'Computer Science';
  v_Majors(4) := 'History';
  v_Majors(5) := 'Economics';
  -- Циклически переберем все строки таблицы и введем их в таблицу
  -- temp_table.
  v_Index := v_Majors.FIRST;
  LOOP
      INSERT INTO temp_table (num_col, char_col)
        VALUES (v_Index, v_Majors(v_Index));
      EXIT WHEN v_Index = v_Majors.LAST;
```

```
        v_Index := v_Majors.NEXT(v_Index);
    END LOOP;
END;
```

## Рекомендации по использованию таблиц PL/SQL

Ниже приведены краткие рекомендации по использованию таблиц PL/SQL.

1. Используйте отдельную переменную для счетчика строк таблицы. В PL/SQL 2.3 это необязательно, так как всегда можно воспользоваться атрибутом COUNT, однако рекомендуется применять это метод. Размер таблиц не ограничен, поэтому программа должна контролировать число строк, вводимых в таблицу.
2. Начинайте со значения индекса, равного 1, и увеличивайте его на 1 с каждым новым элементом. Следующий элемент должен иметь индекс 2, далее — 3 и т.д. С помощью этого способа достаточно просто циклически проходить элементы таблицы, управляя при этом самим процессом. Кроме того, такой способ индексирования таблицы делает возможным связывание ее с массивом C, когда блок PL/SQL вызывается из программы Pro*C или OCI либо встраивается в такую программу. Об использовании PL/SQL в Pro*C и OCI рассказано в главе 13.
3. Не забывайте, что элемент таблицы не определен до тех пор, пока ему не присвоено некоторое значение. Если обратиться к элементу таблицы до того, как ему присвоено значение, возвращается сообщение об ошибке ORA-1403.
4. За исключением атрибута DELETE, в PL/SQL 2.3 не существует способа удалить все строки таблицы. Однако при первом создании таблицы строк в ней нет. Поэтому, если нужно удалить всю таблицу PL/SQL, можно присвоить ей пустую таблицу:

```
-- Этот пример содержится в файле nulltab.sql.
DECLARE
  TYPE t_NameTable IS TABLE OF students.first_name%TYPE
      INDEX BY BINARY_INTEGER;
  v_Names t_NameTable;
  v_EmptyTable t_NameTable;
BEGIN
  /* Присвоим несколько строк v_Names. */
  v_Names(1) := 'Scott';
  v_Names(2) := 'Lefty';
  v_Names(3) := 'Susan';
  /* Удалим все, что находится в v_Names */
  v_Names := v_EmptyTable;
END;
```

# Итоги

В этой главе был рассмотрен ряд определяемых пользователями составных типов, которые могут быть полезны программистам, работающим на PL/SQL. Оставшиеся составные типы рассматриваются в главах 11 и 12. Предварительно необходимо обсудить основы PL/SQL, начиная с взаимодействия PL/SQL с SQL. Именно этому посвящена следующая глава.

# Глава 4

## SQL в PL/SQL

Язык структурированных запросов (SQL – Structured Query Language) определяет способы работы с данными в Oracle. Процедурные конструкции, рассмотренные в главах 2 и 3, становятся намного полезнее, когда они объединяются с мощными средствами обработки информации, предлагаемыми SQL, так как это позволяет программам PL/SQL манипулировать данными в Oracle. В этой главе говорится о том, какие SQL-операторы разрешено использовать в PL/SQL и какие операторы управляют функционированием транзакций и тем самым обеспечивают согласованность данных. Встроенные SQL-функции обсуждаются в главе 5.

# SQL-операторы

Как показано ниже, SQL-операторы можно подразделить на шесть категорий. В таблице 4.1 представлены примеры операторов, относящихся к разным категориям. Все SQL-операторы подробно описаны в руководстве Server SQL Reference.

- С помощью ***операторов языка манипулирования данными*** (DML – data manipulation language) можно модифицировать и запрашивать информацию, содержащуюся в таблицах, однако нельзя изменять структуру таблиц и других объектов.
- С помощью ***операторов языка определения данных*** (DDL – data definition language) можно создавать, удалять и изменять структуру объектов схем. К операторам DDL относятся также команды, используемые для изменения полномочий на объекты схем.
- С помощью ***операторов управления транзакциями*** обеспечивается согласованность данных. Это достигается объединением SQL-операторов в логические транзакции, которые выполняются либо успешно, ибо неуспешно как единое целое.
- С помощью ***операторов управления сеансом работы*** можно изменять установки, задаваемые для отдельного соединения с базой данных, например для того, чтобы разрешить трассировку SQL.
- С помощью ***операторов управления системой*** можно изменять установки, задаваемые для всей базы данных, например для того, чтобы разрешить или запретить архивирование.
- ***Встроенные SQL-команды*** используются для программ предкомпилятора Oracle.

**ТАБЛИЦА 4.1. Категории SQL-операторов**

| Категория | Примеры SQL-операторов |
|---|---|
| Язык манипулирования данными (DML) | SELECT, INSERT, UPDATE, DELETE, SET TRANSACTION, EXPLAIN PLAN |
| Язык определения данных (DDL) | DROP, CREATE, ALTER, GRANT, REVOKE |
| Управление транзакциями | COMMIT, ROLLBACK, SAVEPOINT |
| Управление сеансом работы | ALTER SESSION, SET ROLE |
| Управление системой | ALTER SYSTEM |
| Встроенные SQL-команды | CONNECT, DECLARE CURSOR, ALLOCATE[1] |

[1]Встроенная SQL-команда ALLOCATE доступна в Oracle 7.2 и выше.

## Использование SQL в PL/SQL

Из всех SQL-операторов в программах PL/SQL можно использовать лишь операторы DML и операторы управления транзакциями. Иначе говоря, операторы DDL использовать нельзя. Оператор EXPLAIN PLAN (объяснить план) – хотя он и относится к категории DML – применять также не разрешается. Чтобы пояснить смысл этих ограничений, рассмотрим принципы создания программ PL/SQL.

Вообще говоря, в любом языке программирования привязка переменных может быть либо ранней, либо поздней. *Привязка* (binding) переменной – это процесс указания области памяти, соответствующей идентификатору программы. В PL/SQL в процесс привязки входит также проверка базы данных на наличие полномочий, позволяющих обращаться к объектам схем. В том языке, где используется *ранняя привязка* (early binding), этот процесс осуществляется на этапе компиляции программы, а в языке, где применяется поздняя привязка (late binding), она откладывается до этапа выполнения программы. Ран-

няя привязка означает, что компиляция программы будет занимать большее время (так как при этом нужно привязывать переменные), однако выполняться такая программа будет быстрее, потому что привязка будет уже завершена. Поздняя привязка сокращает время компиляции, но увеличивает время выполнения программы.

При разработке PL/SQL было принято решение об использовании ранней привязки, чтобы к моменту выполнения блока объекты базы данных были уже проверены и чтобы блок мог быть выполнен максимально быстро. Это вполне оправданно, так как блоки PL/SQL можно хранить в базе данных как процедуры, функции, модули и триггеры. Такие объекты хранятся в скомпилированном виде, т. е. при необходимости их можно загрузить из базы данных и выполнить (более подробно о хранимых объектах говорится в главах 7, 8 и 9). Именно поэтому операторы DDL использовать нельзя. Оператор DDL модифицирует объект базы данных, следовательно, полномочия на объект должны быть подтверждены вновь. Процесс подтверждения полномочий требует привязки идентификаторов, а это уже было сделано во время компиляции.

Для иллюстрации вышесказанного рассмотрим гипотетический блок PL/SQL:

```
BEGIN
   CREATE TABLE temp_table (
      num_value NUMBER,
      char_value CHAR(10));
   INSERT INTO temp_table (num_value, char_value)
      VALUES (10, 'Hello');
END;
```

Чтобы скомпилировать этот блок, нужно привязать идентификатор **temp_table.** Процесс привязки будет проверять, существует ли указанная таблица. Пока блок не выполнен, таблица не существует. В результате данный блок не может быть скомпилирован, ни тем более выполнен.

Операторы DML и операторы управления транзакциями — это единственные SQL-операторы, с помощью которых нельзя модифицировать объекты схем или полномочия на объекты схем, поэтому в PL/SQL можно использовать только эти SQL-операторы.

PL/SQL 2.1 ... и ВЫШЕ

**Использование DDL** Тем не мнее существует способ обойти это ограничение. В PL/SQL 2.1 и выше можно воспользоваться модулем DBMS_SQL. Он позволяет создать SQL-оператор динамически, во время выполнения программы, а затем провести его синтаксический анализ и обработку. Такой оператор до момента выполнения программы фактически еще не создан, поэтому от компилятора PL/SQL не требуется привязывать идентификаторы этого оператора, что позволяет компилировать блок. (Модуль DBMS_SQL детально описан в главе 15.) К примеру, можно было бы воспользоваться этим модулем для выполнения оператора CREATE TABLE, рассмотренного в предыдущем блоке. Однако скомпилировать оператор INSERT не удастся, так как таблица не будет существовать до выполнения блока. Чтобы выйти из этой ситуации, можно и для оператора INSERT воспользоваться модулем DBMS_SQL.

# DML в PL/SQL

Операторы DML — это SELECT (выбрать), INSERT (вставить), UPDATE (обновить) и DELETE (удалить). Каждый оператор действует в соответствии со своим названием: с помощью SELECT в таблице выбираются строки, указанные в условии WHERE; посредством INSERT к таблице базы данных добавляются строки; с помощью UPDATE строки, указанные в условии WHERE, модифицируются; с помощью DELETE эти строки удаляются. Помимо условия WHERE, в этих операторах могут быть и другие конструкции, которые описаны ниже.

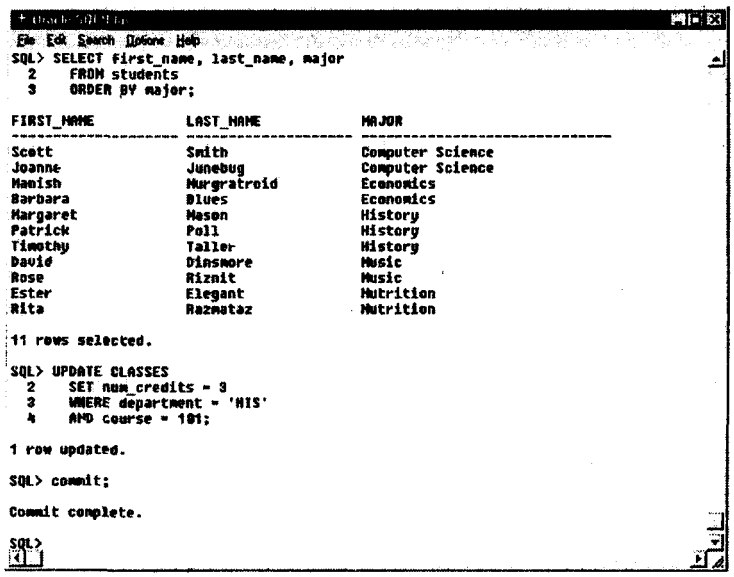
Когда SQL-операторы выполняются из SQL*Plus, результаты выводятся на экран (рис. 4.1). Для операторов UPDATE, INSERT или DELETE SQL*Plus возвращает число обработанных строк, а для оператора SELECT — строки, указанные в запросе.

Рассмотрим оператор UPDATE, изображенный на рис. 4.1:

```
UPDATE CLASSES
   SET num_credits = 3
   WHERE department = 'HIS'
   AND course = 101;
```

Все значения, используемые для изменения содержимого таблицы classes, жестко кодированы, т. е. известны во время написания этого оператора. В PL/SQL ограничения на использование переменных

**Рис. 4.1.**
*Результаты выполнения SQL-операторов в SQL*Plus*




отсутствуют. Переменные разрешены в любом месте SQL-оператора, где разрешены выражения. Переменные, используемые таким образом, называются переменными присваивания, или *переменными привязки* (bind variables). Например, в приведенном выше операторе UPDATE можно заменить жестко заданное значение, обозначающее число зачетов, на переменную присваивания:

```
-- Этот пример содержится в файле bindvar.sql.
DECLARE
  v_NumCredits classes.num_credits%TYPE;
BEGIN
  /* Присвоим значение переменной v_NumCredits. */
  v_NumCredits := 3;
  UPDATE CLASSES
      SET num_credits = v_NumCredits
      WHERE department = 'HIS'
      AND course = 101;
END;
```

В SQL-операторе можно заменять на переменные только выражения. Особенно важно указывать имена таблиц и столбцов. Дело в том, что при ранней привязке имена объектов Oracle должны быть известны во время компиляции. По определению, значение переменной до выполнения программы неизвестно. Чтобы обойти это ограничение, можно вновь воспользоваться модулем DBMS_SQL.

# SELECT

С помощью оператора SELECT данные выбираются в базе данных и записываются в переменные PL/SQL. Общий вид оператора SELECT приведен на рис. 4.2.

Каждый из компонентов этого оператора описан в таблице 4.2.

▼ ВНИМАНИЕ

В операторе SELECT можно использовать большее число различных конструкций, в частности ORDER BY (упорядочить по) и GROUP BY (сгруппировать по). Это подробно обсуждается в главе 6. За более детальной информацией обратитесь к справочному руководству Server SQL Reference.

**Рис. 4.2.**
*Синтаксис оператора SELECT*

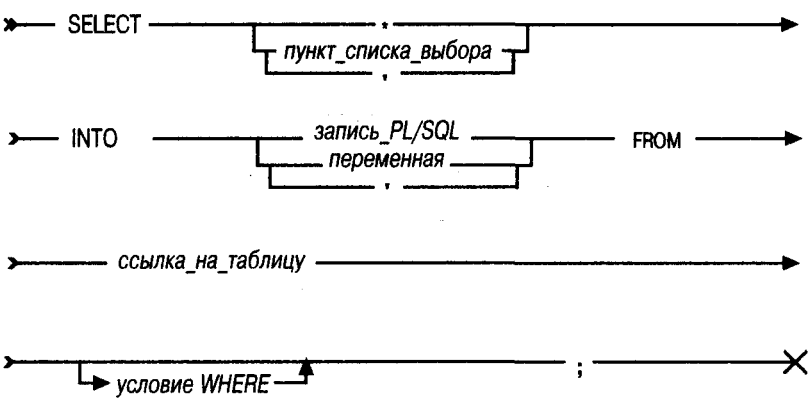



**ТАБЛИЦА 4.2**

| Элемент выбора | Описание |
|---|---|
| *пункт_списка_выбора* | Выбранный столбец (или выражение). Каждый *пункт списка выбора* отделяется запятой и при желании может быть указан как псевдоним. Полный набор пунктов списка выбора называется *списком выбора* (select list). Символ * является кратким условным обозначением всей строки. При его использовании выбирается каждое поле строки таблицы в том порядке, в котором они определены. |
| *переменная* | Переменная PL/SQL, в которую попадает выбранный столбец. Каждая переменная должна быть совместима с соответствующим пунктом списка выбора. Число пунктов списка выбора и выходных переменных должно совпадать. |
| *запись_PL/SQL* | Может использоваться вместо списка переменных. Запись должна содержать поля, которые соответствуют списку выбора, но обеспечивают более простое управление возвращаемыми данными. При помощи записей связанные поля группируются в одну синтаксическую единицу, поэтому к полям записи можно обращаться как к единому целому либо по отдельности. (Более подробно о записях рассказано ниже в этой главе.) Если в качестве списка выбора указан символ *, то такая запись может быть определена как *ссылка_на_таблицу*%ROWTYPE. |
| *ссылка_на_таблицу* | Определяет таблицу, из которой выбираются данные. Может быть синонимом либо таблицей, принадлежащей той удаленной базе данных, на которую указывает связь баз данных. Более подробно о ссылках на таблицы рассказано ниже. |
| *условие WHERE* | Критерий запроса. С помощью этого условия задается строка, которая будет возвращена запросом. Условие WHERE состоит из логических условий, соединенных знаками логических операций, и также описано более детально ниже в этой главе. |

При помощи оператора SELECT, вид которого описан выше, можно выбрать не более одной строки. Заданный критерий выбора будет сопоставляться с каждой строкой таблицы, и, если условию будет удовлетворять несколько строк, PL/SQL вернет сообщение об ошибке:

```
ORA-1427: Single-row query returns more than one row
```

(Однострочный запрос возвращает более одной строки. — *Прим. пер.*)

В этом случае следует воспользоваться курсором и выбрать каждую строку по отдельности. О курсорах рассказывается в главе 6.

Ниже приведены два различных оператора SELECT.

```
-- Этот пример содержится в файле select.sql.
DECLARE
  V_StudentRecord  students%ROWTYPE;
  V_Department     classes.department%TYPE;
  V_Course         classes.course%TYPE;
```

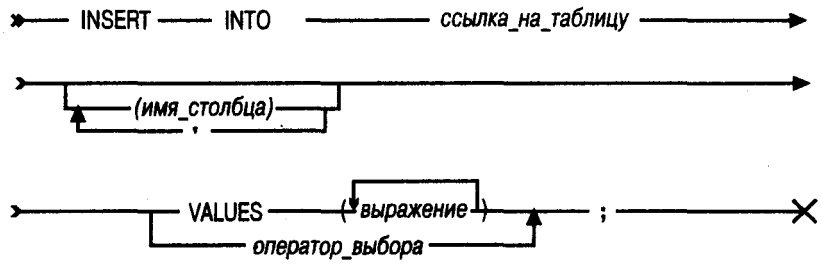
```
BEGIN
   -- Выберем одну запись в таблице students и сохраним ее в
   -- переменной v_StudentRecord. Обратите внимание, что условию WHERE
   -- удовлетворяет только одна строка таблицы. Кроме того, заметьте,
   -- что запрос возвращает все поля таблицы students (так как выбран
   -- символ *). Поэтому запись, в которую считывается информация,
   -- определена как students%ROWTYPE.
SELECT *
   INTO v_StudentRecord
   FROM students
   WHERE id = 10000;

   -- Выберем два поля в таблице classes и сохраним их в переменных
   -- v_Department и v_Course. Условию WHERE вновь удовлетворяет только
   -- одна строка таблицы.
   SELECT department, course
       INTO v_Department, v_Course
       FROM classes
       WHERE room_id = 99997;
END;
```

## INSERT

Синтаксис оператора INSERT приведен на рис. 4.3. Обратите внимание, что в операторе отсутствует условие WHERE (хотя его можно задать в подзапросе).

Здесь *ссылка_на_таблицу* указывает на таблицу Oracle, имя_столбца — на столбец этой таблицы, а выражение является выражением SQL или PL/SQL, как это было определено в предыдущей главе. (Ссылки на таблицы более подробно обсуждаются ниже.) Если в операторе INSERT имеется *оператор_выбора*, то список выбора должен соответствовать столбцам, в которые вносится информация.

**Рис. 4.3.**
*Синтаксис оператора INSERT*




Ниже приведен пример, иллюстрирующий правильное использование операторов INSERT:

```
-- Этот пример содержится в файле insert.sql.
DECLARE
  v_StudentID students.id%TYPE;
BEGIN
  -- Считаем идентификационный номер нового студента.
  SELECT student_sequence.NEXTVAL
      INTO v_StudentID
      FROM dual;

  -- Добавим строку в таблицу students.
  INSERT INTO students (id, first_name, last_name)
```

```
        VALUES (v_StudentID, 'Timothy', 'Taller');

    -- Добавим вторую строку, но при этом воспользуемся
    -- последовательным номером непосредственно в операторе INSERT.
    INSERT INTO students (id, first_name, last_name)
        VALUES (student_sequence.NEXTVAL, 'Patrick', 'Poll');
END;
```

Ниже приведен пример, который неверен, так как список выбора в подзапросе не соответствует столбцам, в которые вводятся данные. В результате выполнения этого оператора возвращается ошибка: ORA-913: Too many values (Слишком много значений. — *Прим. пер.*)

```
INSERT INTO rooms
  SELECT * FROM classes;
```

Однако следующий пример правилен: в нем при помощи ввода второй копии каждой из строк удваивается размер таблицы classes:

```
INSERT INTO classes
  SELECT * FROM classes
```

**PL/SQL 8.0 ... и ВЫШЕ**

В версии Oracle8, допускающей применение объектов, в операторе INSERT можно использовать дополнительную конструкцию REF INTO. При использовании этой конструкции с объектными таблицами она будет возвращать ссылку на объект, в который вводится информация. Более подробно об этом говорится в главе 11.

## UPDATE

Синтаксис оператора UPDATE приведен на рис. 4.4.

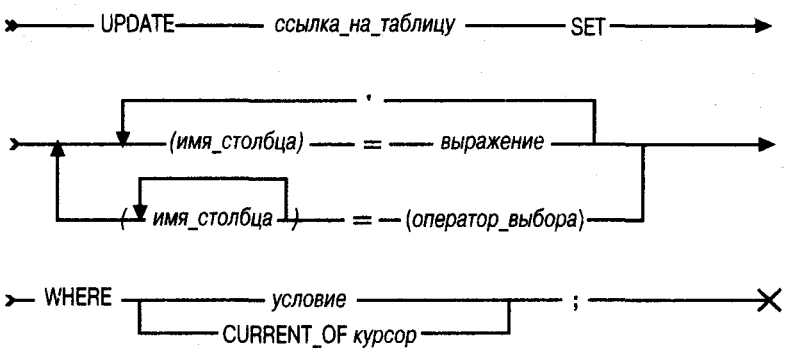



**Рис. 4.4.** *Синтаксис оператора UPDATE*

Здесь *ссылка_на_таблицу* указывает на обновляемую таблицу, *имя_столбца* — на столбец, значение которого будет изменено, а выражение — это выражение SQL, как определено в главе 2. Если в операторе содержится *оператор_выбора*, то список выбора должен соответствовать столбцам команды SET (установить).

Ниже приведен пример использования оператора UPDATE.

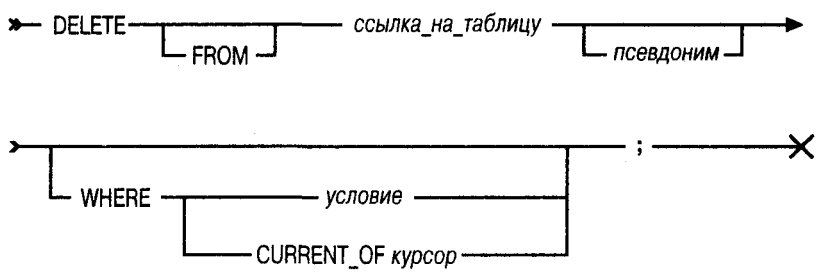
```
-- Этот пример содержится в файле update.sql.
DECLARE
  V_Major           students.major%TYPE;
  V_CreditIncrease  NUMBER := 3;
BEGIN
  -- При помощи оператора UPDATE к значениям полей
  -- current_credits всех студентов, специализирующихся
  -- по истории, будет добавлено значение 3.
  v_Major := 'History';
  UPDATE students
      SET current_credits = current_credits + v_CreditIncrease
```

```
        WHERE major = V_Major;
END;
```

## DELETE

При помощи оператора DELETE из таблицы базы данных удаляются строки. Условие WHERE оператора указывает на те строки, которые должны быть удалены. Синтаксис оператора DELETE приведен на рисунке 4.5.

**Рис. 4.5.**
*Синтаксис оператора DELETE*




Здесь *ссылка_на_таблицу* указывает на таблицу Oracle, а *условие* определяет группу строк, которые будут удалены. Специальный синтаксис CURRENT OF *курсор* используется вместе с курсором (см. главу 6). О ссылках на таблицы и об условии WHERE рассказывается далее.

Ниже приведен блок, иллюстрирующий применение различных операторов DELETE.

```
-- Этот пример содержится в файле delete.sql.
DECLARE
  v_StudentCutoff NUMBER;
BEGIN
  v_StudentCutoff := 10;
  -- Удалим те учебные группы, в которых число зарегистрированных
  -- студентов слишком мало.
  DELETE FROM classes
      WHERE current_students < v_StudentCutoff;

  -- Удалим всех студентов-экономистов, которые еще не сдали ни
  -- одного зачета.
  DELETE FROM students
      WHERE current_credits = 0
      AND major = 'Economics';
END;
```

## Условие WHERE

В состав операторов SELECT, UPDATE и DELETE в качестве важнейшей части входит условие WHERE. Оно определяет *активный набор* (active set) — множество строк, которое возвращается запросом SELECT или на которое воздействуют операторы UPDATE или DELETE.

Условие WHERE состоит из конструкций, соединенных логическими операциями AND (и), OR (или) и NOT (не). Эти конструкции обычно являются отношениями, например в следующем операторе DELETE:

```
DECLARE
  v_Department CHAR(3);
```

```
BEGIN
   v_Department := 'CS';
   -- Удалим все компьютерные группы.
   DELETE FROM classes
       WHERE department = v_Department;
END;
```

С помощью приведенного блока будут удалены все строки таблицы **classes**, для которых заданное условие истинно (т.е. значение поля **department** равно 'CS'). При вычислении отношений следует обращать внимание на несколько моментов, в том числе на важность имен переменных и на сравнение символов.

## Имена переменных

Представим себе, что имя переменной, рассмотренной в предыдущем блоке, изменено с **v_Department** на **department**:

```
DECLARE
   Department CHAR(3);
BEGIN
   Department := 'CS';
   -- Удалим все компьютерные группы.
   DELETE FROM classes
       WHERE department = Department;
END;
```

Это простое изменение оказывает значительное влияние на результаты выполнения оператора: с помощью модифицированного блока из таблицы **classes** будут удалены все строки, а не только те, для которых значение факультета равно 'CS'! Все дело в способе осуществления синтаксического анализа идентификаторов SQL-оператора. Когда PL/SQL встречает условие вида

*выражение1 = выражение2*

эти выражения вначале проверяются на соответствие столбцам таблицы, над которой выполняется операция. Затем производится проверка наличия в блоке PL/SQL переменных. PL/SQL нечувствителен к регистру символов, поэтому в нашем блоке как **department**, так и **Department** ассоциируются со столбцом таблицы **classes**, а не с переменной. Указанное условие истинно для каждой строки таблицы, поэтому все строки будут удалены.

Если для блока создана метка, то для переменной и столбца можно использовать одинаковые имена. Для этого надо указать метку при ссылке на переменную. Ниже приведен блок, выполнение которого дает нужный эффект, т.е. удаляются только те строки, в которых **department** = 'CS'.

```
<<l_DeleteBlock>>
DECLARE
   Department CHAR(3);
BEGIN
   Department := 'CS';
   -- Удалим все компьютерные группы.
   DELETE FROM classes
       WHERE department = l_DeleteBlock.Department;
END;
```

Хотя применять такой способ разрешено, использование одного и того же имени для переменной PL/SQL и для столбца таблицы считается плохим стилем программирования (стиль обсуждался в конце главы 2).

## Сравнение символов

Когда в Oracle сравниваются два символьных значения, как в предыдущем примере, можно воспользоваться двумя различными методами: сравнением с дополнением пробелами и сравнением без него. Эти два метода различаются способами сравнения строк символов с разной длиной. Предположим, что сравниваются две строки символов – **строка1** и **строка2.** При сравнении с дополнением пробелами используется следующий алгоритм:

1. Если длина строк различна, то сначала более короткое значение дополняется пробелами, чтобы длина строк сравнялась.

2. Строки сравниваются посимвольно, начиная слева. Предположим, что сравниваются **символ1** – символ **строки1**, и **символ2** – символ **строки2**.
3. Если ASCII(**символ1**) < ASCII(**символ2**), то **строка1 < строка2.** Если ASCII(**символ1**) > ASCII(**символ2**), то **строка1 > строка2**. Если ASCII(**символ1**) = ASCII(**символ2**), то берутся следующие символы **строки1** и **строки2**.
4. Если достигнут конец каждой из строк, то строки равны.

При использовании данного метода следующие условия будут возвращены как истинные:

```
'abc' = 'abc'
'abc ' = 'abc' -- Обратите внимание на конечные пробелы в первой строке.
'ab' < 'abc'
'abcd' > 'abcc'
```

Алгоритм сравнения строк символов без дополнения пробелами выглядит несколько по-другому:

1. *Строки сравниваются посимвольно, начиная слева. Предположим, что сравниваются* **символ1** – символ **строки1**, и **символ2** – символ **строки2**.
2. Если ASCII(**символ1**) < ASCII(**символ2**), то **строка1 < строка2.** Если ASCII(**символ1**) > ASCII(**символ2**), то **строка1 > строка2**. Если ASCII(**символ1**) = ASCII(**символ2**), то берутся следующие символы **строки1** и **строки2**.
3. Если конец **строки1** достигается раньше конца **строки2**, то **строка1 < строка2.** Если конец **строки2** достигается раньше конца **строки1**, то **строка1 > строка2**.
4. Если достигнут конец каждой из строк, то строки равны.

При использовании данного метода сравнения символов будут истинны следующие условия:

```
'abc' = 'abc'
'ab' < 'abc'
'abcd' > 'abcc'
```

Однако сравнение символов, приведенное ниже, даст ложный результат, так как длина строк различна. Именно в этом заключается основное различие двух методов сравнения символов.

```
'abc ' = 'abc' -- Обратите внимание на конечные пробелы в первой строке.
```

Так какой же метод когда использовать? В PL/SQL метод сравнения с дополнением пробелами используется только в том случае, когда оба сравниваемых значения имеют *фиксированную* длину. Если же одно из значений имеет переменную длину, то применяется сравнение без дополнения пробелами. Данные типа CHAR имеют фиксированную длину, а данные типа VARCHAR2 – переменную. Считается, что символьные литералы (заключенные в одиночные кавычки) всегда имеют фиксированную длину.

Если оператор не оказывает воздействия на нужные строки, проверьте типы данных, используемые в условии WHERE. Ниже приведен блок, в результате выполнения которого ни одна строка удалена не будет, так как переменная **v_Department** имеет тип VARCHAR2, а не CHAR.

```
DECLARE
  v_Department VARCHAR2(3);
BEGIN
  v_Department := 'CS';
  -- Удалим все компьютерные группы.
  DELETE FROM classes
      WHERE department = v_Department;
END;
```

Столбец **department** таблицы **classes** имеет тип CHAR. Поэтому для всех компьютерных групп значение **department** будет равно 'CS ' (обратите внимание на конечный пробел). Значение переменной **v_Department** равно 'CS' (конечный пробел отсутствует), и она имеет тип данных переменной длины, поэтому оператор DELETE не оказывает никакого влияния на строки.

Для получения нужного эффекта от использования условия WHERE следите за тем, чтобы переменные в блоке PL/SQL обязательно имели те же типы, что и столбцы базы данных, с которыми сравниваются переменные. Это можно гарантировать, если использовать атрибут %TYPE.

## Ссылки на таблицы

Во всех операциях DML производятся ссылки на таблицы. Общий вид ссылки на таблицу можно представить следующим образом:

*[схема.]таблица[@связь_баз_данных]*

где *схема* идентифицирует владельца таблицы, а *связь_баз_данных* — таблицу удаленной базы данных.

Чтобы установить соединение с базой данных, необходимо указать имя пользователя и пароль для конкретной схемы. SQL-операторы, выполняемые впоследствии во время сеанса работы, будут обращаться к этой схеме по умолчанию. Если ссылка на таблицу не уточнена, например:

```
UPDATE students
  SET major = 'Music'
  WHERE id = 10005;
```

то имя таблицы (в этом примере **students**) должно указывать на таблицу, которая принадлежит схеме, заданной по умолчанию. Если это не так, возвращается сообщение об ошибке:

```
ORA-942: table or view does not exist
```

(таблицы или представления не существует. — *Прим. пер.*)

или

```
PLS-201: identifier must be declared
```

(необходимо объявить идентификатор. — *Прим. пер.*)

Схема по умолчанию — это схема, с которой пользователь соединяется до начала выполнения каких-либо команд SQL или PL/SQL. Если таблица находится в другой схеме, на нее можно сослаться при помощи имени схемы, например:

```
UPDATE example.students
  SET major = 'Music'
  WHERE id = 10005;
```

Данный оператор UPDATE будет выполнен в том случае, если соединение установлено со схемой example или с другой схемой, которой предоставлена привилегия UPDATE на таблицу students.

## Связи баз данных

Если в системе установлено программное средство SQL*Net, то можно применять связи баз данных. *Связь баз данных* (database link) — это ссылка на удаленную базу данных, которая может быть размещена в системе, не имеющей ничего общего с локальной базой данных. Связь баз данных создается с помощью следующего оператора DDL:

CREATE DATABASE LINK *имя_связи*
  CONNECT TO *имя_пользователя* IDENTIFIED BY *пароль*
  USING *sqlnet_строка;*

Синтаксис для имени связи баз данных — *имя_связи* — подчиняется обычным правилам, установленным для идентификаторов баз данных. Элементы *имя_пользователя* и пароль идентифицируют схему удаленной базы данных, а *sqlnet_строка* является корректной последовательностью символов, применяющейся для соединения с удаленной базой данных. Предположим, что нужные схемы созданы, а средство SQL*Net версии 2 инсталлировано; теперь создадим связь баз данных:

```
CREATE DATABASE LINK example_backup
  CONNECT TO example IDENTIFIED BY example
  USING 'backup_database';
```

Более подробную информацию о том, как инсталлировать и конфигурировать SQL*Net, можно найти в руководстве SQL*Net User's Guide and Reference. С учетом созданной связи обновим удаленную таблицу students:

```
UPDATE students@example_backup
  SET major = 'Music'
  WHERE id = 10005;
```

Когда связь используется как часть некоторой транзакции, эту транзакцию называют *распределенной* (distributed transaction), так как с ее помощью модифицируется несколько баз данных. За более подроб-

ными сведениями о распределенных транзакциях, а также об их назначении и о работе с ними обращайтесь к справочному руководству Server SQL Reference.

## Синонимы

Структура ссылок на таблицы может быть достаточно сложна, особенно когда в ссылках указываются схемы и/или связи баз данных. Для упрощения работы в Oracle разрешено создавать синонимы для сложных ссылок. *Синоним* (synonym), по существу, дает ссылке на таблицу другое имя, подобно тому как псевдоним (alias) дает новое имя пункту списка выбора. Синоним является объектом словаря данных и создается при помощи оператора DDL CREATE SYNONYM:

CREATE SYNONYM *имя_синонима* FOR *ссылка*;

Здесь *имя_синонима* — это имя синонима, а *ссылка* — ссылка на объект схемы. Объект схемы может быть таблицей, как в следующем примере, либо процедурой, последовательностью или другим объектом базы данных.

```
CREATE SYNONYM backup_students
  FOR students@example_backup;
```

С учетом этого синонима перепишем наш распределенный оператор UPDATE:

```
UPDATE backup_students
  SET major = 'Music'
  WHERE id = 10005;
```

▼ ВНИМАНИЕ Создание синонима не дает каких-либо привилегий объекту, на который происходит ссылка; синоним — это всего лишь альтернативное имя объекта. Если к объекту необходимо обратиться из другой схемы, это право можно предоставить либо явным образом, либо через роль (при помощи оператора GRANT).

# Псевдостолбцы

*Псевдостолбцы* (pseudocolumns) — это дополнительные функции, которые можно вызвать только из SQL-операторов. Синтаксически они аналогичны столбцам таблиц, однако реально псевдостолбцы совсем не похожи на них. Их скорее можно назвать частью процесса выполнения SQL-операторов.

## CURRVAL и NEXTVAL

Эти два псевдостолбца — CURRVAL (текущее значение) и NEXTVAL (следующее значение) — применяются в последовательностях. *Последовательность* (sequence) — это объект Oracle, который используется для генерирования уникальных чисел (не путайте с последовательностью, или строкой (string), символов. — *Прим. пер.*). Последовательность создается с помощью команды DDL CREATE SEQUENCE. После того как последовательность создана, к ней можно обратиться следующим образом:

*последовательность*.CURRVAL

или

*последовательность*.NEXTVAL

где *последовательность* — это имя последовательности. CURRVAL возвращает текущее значение последовательности, а NEXTVAL увеличивает ее и возвращает новое значение. Как CURRVAL, так и NEXTVAL возвращают значения, имеющие тип NUMBER.

Значения последовательностей могут быть использованы в списках выбора запросов, в списках значений (VALUES) операторов INSERT и в командах SET операторов UPDATE. Однако их нельзя указывать в условиях WHERE и в процедурных операторах PL/SQL. Ниже приведены примеры правильного использования CURRVAL и NEXTVAL.

```
CREATE SEQUENCE student_sequence
  START WITH 10000;

-- В качестве значения идентификатора в этом операторе будет
-- использоваться число 10000.
INSERT INTO students (id, first_name, last_name)
  VALUES (student_sequence.NEXTVAL, 'Scott', 'Smith');
```

```
-- В качестве значения идентификатора в этом операторе будет
--- использоваться число 10001.
INSERT INTO students (id, first_name, last_name)
  VALUES (student_sequence.NEXTVAL, 'Margaret', 'Mason');

SELECT student_sequence.NEXTVAL "Value"
  FROM dual; -- Сначала увеличивается значение последовательности.
Value
--------
10002

SELECT student_sequence.CURRVAL "Value"
FROM dual; -- Возвращается текущее значение.
Value
--------
10002
```

## LEVEL

LEVEL (уровень) используется в операторах SELECT для просмотра таблицы с учетом иерархического дерева, с помощью конструкций START WITH (начать с) и CONNECT BY (соединиться посредством). Псевдостолбец LEVEL возвращает текущий уровень дерева в виде значения типа NUMBER. Более подробно об этом говорится в руководстве Server SQL Reference.

## ROWID

Псевдостолбец ROWID (идентификатор строки) используется в списках выбора запросов и возвращает идентификатор конкретной строки. Внешним форматом ROWID является 18-символьная последовательность (см. главу 2). Псевдостолбец ROWID возвращает значение, имеющее тип ROWID. К примеру, следующий запрос возвращает идентификаторы всех строк таблицы **rooms**:

```
SELECT ROWID
  FROM rooms;

ROWID
------------------
00000045.0000.0002
00000045.0001.0002
00000045.0002.0002
00000045.0003.0002
00000045.0004.0002
```

▼ ВНИМАНИЕ Форматы ROWID, используемые в Oracle7 и Oracle8, различны. Однако внешний формат в обеих версиях одинаков и представляет собой 18-символьную последовательность (более подробно см. в главе 2).

## ROWNUM

Псевдостолбец ROWNUM возвращает номер текущей строки. Он полезен для ограничения общего числа строк и используется в первую очередь в условиях WHERE запросов и командах SET операторов UPDATE. Значение, возвращаемое ROWNUM, имеет тип NUMBER. Например, следующий запрос возвращает только две первые строки таблицы **students**:

```
SELECT *
  FROM students
  WHERE ROWNUM < 3;
```

ROWNUM первой строки равен 1, ROWNUM второй строки — 2 и т.д.

▼ ВНИМАНИЕ Значение ROWNUM присваивается строке до выполнения сортировки строк (посредством ORDER BY). В результате нельзя пользоваться ROWNUM для считывания *n* старших строк, расположенных в заданном порядке. Рассмотрим оператор:

```
SELECT first_name, last_name
    FROM students
    WHERE ROWNUM < 3
    ORDER BY first_name;
```

Хотя в результате выполнения этого оператора будут возвращены две строки таблицы **students**, упорядоченной по столбцу **first_name**, вовсе не обязательно, что это будут две первые строки в общем порядке сортировки. Чтобы гарантировать выбор нужных строк, для этого запроса следует создать курсор и считать только две первые строки. О курсорах и об их применении рассказывается в главе 6.

# GRANT, REVOKE и привилегии

Такие операторы DDL, как GRANT (предоставить) и REVOKE (отменить), нельзя использовать непосредственно в PL/SQL, – они служат для определения возможности выполнения других SQL-операторов. Чтобы выполнить над таблицей Oracle некоторую операцию, например INSERT или DELETE, необходимо иметь соответствующие полномочия, которые предоставляются и отменяются с помощью SQL-команд GRANT и REVOKE.

## Объектные и системные привилегии

Существуют привилегии двух различных видов: объектные и системные. *Объектная привилегия* (object privilege) разрешает выполнение определенной операции над конкретным объектом (например, над таблицей). *Системная привилегия* (system privilege) разрешает выполнение операций над целым классом объектов.

В таблице 4.3 представлены объектные привилегии. Объектные привилегии DDL – ALTER (изменить), INDEX (индексировать), REFERENCES (ссылки) – нельзя применить непосредственно в PL/SQL (за исключением модуля DBMS_SQL), так как они разрешают выполнение операций DDL над определенным объектом.

**ТАБЛИЦА 4.3. Объектные привилегии SQL**

| Объектная привилегия | Описание | Типы объектов схемы |
|---|---|---|
| ALTER | Обладателю этой привилегии разрешено выполнить над объектом ALTER (например, ALTER TABLE) | Таблицы, последовательности |
| DELETE | Обладателю этой привилегии разрешено выполнить над объектом оператор DELETE | Таблицы, представления |
| EXECUTE | Обладатель этой привилегии может выполнить хранимый объект PL/SQL (хранимые объекты рассмотрены в главах 7-9) | Процедуры, функции, модули |
| INDEX | Обладатель этой привилегии может создать индекс для таблицы с помощью команды CREATE INDEX | Таблицы |
| INSERT | Обладателю этой привилегии разрешено выполнить над объектом оператор INSERT | Таблицы, представления |
| REFERENCES | Обладатель этой привилегии может наложить ограничение на таблицу | Таблицы |
| SELECT | Обладателю этой привилегии разрешено выполнить над объектом оператор SELECT | Таблицы, представления, последовательности, моментальные снимки |
| UPDATE | Обладателю этой привилегии разрешено выполнить над объектом оператор UPDATE | Таблицы, представления |

Существует множество системных привилегий, соответствующих практически всем возможным операциям DDL. Например, системная привилегия CREATE TABLE позволяет ее обладателю создавать таблицы. Системная привилегия CREATE ANY TABLE дает возможность создавать таблицы в других схемах. Все существующие системные привилегии представлены в руководстве Server SQL Reference.

# GRANT и REVOKE

Оператор GRANT используется для открытия другой схеме доступа к привилегии, а оператор REVOKE – для запрещения доступа, разрешенного оператором GRANT. Оба оператора могут использоваться как для объектных, так и для системных привилегий.

## GRANT

*Для объектных привилегий синтаксис оператора* GRANT таков:

GRANT *привилегия* ON *объект* TO *обладатель_привилегии [WITH GRANT OPTION];*

где *привилегия* – это нужная привилегия, *объект* – это объект, к которому разрешается доступ, а *обладатель_привилегии* – пользователь, получающий привилегию. Для примера предположим, что **userA** – это корректная схема базы данных. Тогда оператор GRANT может выглядеть следующим образом:

```
GRANT SELECT ON classes TO userA;
```

Если указывается параметр WITH GRANT OPTION (с возможностью предоставления привилегий), то пользователь **userA** будет иметь возможность и сам предоставлять привилегии другим пользователям. В операторе GRANT можно указывать несколько привилегий, например:

```
GRANT UPDATE, DELETE ON students TO userA;
```

Для системных привилегий синтаксис оператора GRANT таков:

GRANT *привилегия* TO *обладатель_привилегии* [WITH ADMIN OPTION];

где *привилегия* – это предоставляемая системная привилегия, а *обладатель_привилегии* – пользователь, получающий привилегию. Если в оператор включается параметр WITH ADMIN OPTION (с возможностью администрирования), то обладатель_привилегии будет иметь возможность предоставлять привилегии другим пользователям. Например:

```
GRANT CREATE TABLE, ALTER ANY PROCEDURE TO userA;
```

Как и в случае с объектными привилегиями, в одном и том же операторе GRANT можно указывать несколько привилегий.

Оператор GRANT является оператором DDL, поэтому результаты его выполнения сказываются немедленно, и, после того как он выполнен, осуществляется неявное завершение (COMMIT) транзакции.

## REVOKE

*Для объектных привилегий синтаксис оператора* REVOKE таков:

REVOKE *привилегия* ON *объект* FROM *обладатель_привилегии* [CASCADE CONSTRAINTS];

где *привилегия* – это отменяемая привилегия, *объект* – это объект, на который предоставлена привилегия, а *обладатель_привилегии* – пользователь, получивший эту привилегию. Например, следующая команда REVOKE верна:

```
REVOKE SELECT ON classes FROM userA;
```

Если в оператор включена конструкция CASCADE CONSTRAINTS (каскадные ограничения) и отменяется привилегия REFERENCES, удаляются также все ограничения ссылочной целостности, созданные обладателем этой привилегии. В одном операторе можно отменять несколько привилегий:

```
REVOKE UPDATE, DELETE, INSERT ON students FROM userA;
```

Для системных привилегий синтаксис оператора REVOKE таков:

REVOKE *привилегия* FROM *обладатель_привилегии*;

где *привилегия* – это отменяемая системная привилегия, а *обладатель_привилегии* – пользователь, который более не будет ее иметь. Например:

```
REVOKE ALTER TABLE, EXECUTE ANY PROCEDURE FROM userA;
```

## Роли

В больших системах Oracle, в которых работает множество пользователей, управление привилегиями – достаточно сложная задача. Для ее упрощения можно использовать функциональное средство Oracle,

называемое ролями. Роль (role) является, по существу, совокупностью привилегий, как объектных, так и системных. Рассмотрим следующую группу операторов:

```
CREATE ROLE table_query;
GRANT SELECT ON students TO table_query;
GRANT SELECT ON classes TO table_query;
GRANT SELECT ON rooms TO table_query;
```

Роль **table_query** предоставлена привилегии SELECT на три различные таблицы. Теперь можно предоставить эту роль другим пользователям:

```
GRANT table_query TO userA;
GRANT table_query TO userB;
```

Теперь привилегию SELECT на три таблицы имеют пользователи **userA** и **userB**. Это упрощает администрирование: ведь без использования ролей пришлось бы шесть раз предоставлять привилегии пользователям.

В Oracle имеется предопределенная роль, называемая PUBLIC (общая), которая автоматически предоставляется каждому пользователю. Поэтому выполнение оператора вида

```
GRANT привилегия TO PUBLIC;
```

одновременно предоставляет указанную привилегию всем пользователям Oracle.

В Oracle предварительно определен ряд других ролей, в который включены часто применяемые системные привилегии. Список этих ролей приведен в таблице 4.4. Предварительно определенный пользователь Oracle с именем SYSTEM автоматически получает все эти роли.

### ТАБЛИЦА 4.4. Предварительно определенные системные роли

| Имя роли | Предоставленные привилегии |
|---|---|
| CONNECT | ALTER SESSION, CREATE CLUSTER, CREATE DATABASE LINK, CREATE SEQUENCE, CREATE SESSION, CREATE SYNONYM, CREATE TABLE, CREATE VIEW |
| RESOURCE | CREATE CLUSTER, CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE |
| DBA | Все системные привилегии (с ADMIN OPTION; поэтому эти привилегии могут быть предоставлены вновь), а также EXP_FULL_DATABASE и IMP_FULL_DATABASE |
| EXP_FULL_DATABASE | SELECT ANY TABLE, BACKUP ANY TABLE, а также INSERT, UPDATE, DELETE для системных таблиц **sys.incexp, sys.incvid** и **sys.incfil** |
| IMP_FULL_DATABASE | BECOME USER |

Обычно обе роли — CONNECT (соединение) и RESOURCE (ресурс) — предоставляются тем пользователям базы данных, которые будут создавать объекты схем, а тем пользователям, которые будут обращаться к объектам с запросами, — лишь роль CONNECT. Этим пользователям потребуются дополнительные привилегии на объекты, к которым они будут обращаться.

# Управление транзакциями

*Транзакция* (transaction) — это группа SQL-операторов, которые выполняются (успешно или неуспешно) как единое целое. Транзакции являются стандартным элементом реляционных баз данных и обеспечивают согласованность информации. Классическим примером транзакции является банковская операция. Рассмотрим два SQL-оператора, с помощью которых осуществляется перевод некоторой суммы, определяемой числом **transaction_amount**, с одного банковского счета **from_acct** на другой — **to_acct**.

```
UPDATE accounts
  SET balance = balance - transaction_amount
  WHERE account_no = from_acct;
UPDATE accounts
```

```
SET balance = balance + transaction_amount
WHERE account_no = to_acct;
```

Предположим, что первый оператор UPDATE выполнен успешно, однако второй не был выполнен в результате ошибки (например, из-за выхода из строя базы данных или сети). Тогда данные окажутся несогласованными — деньги сняты со счета **from_acct**, но не переведены на счет **to_acct**. Не стоит говорить, что такая ситуация весьма неприятна (особенно когда владельцем счета **from_acct** являетесь вы сами). Но ее можно предотвратить, если объединить два оператора в одну транзакцию. При этом оба оператора будут либо выполнены, либо не выполнены, но это обеспечит согласованность данных.

Транзакция начинается с первого SQL-оператора, выданного после окончания предшествующей транзакции или после соединения с базой данных. Завершается транзакция оператором COMMIT или ROLLBACK.

## COMMIT и ROLLBACK

Когда над базой данных выполняется оператор COMMIT (завершить), транзакция заканчивается и:

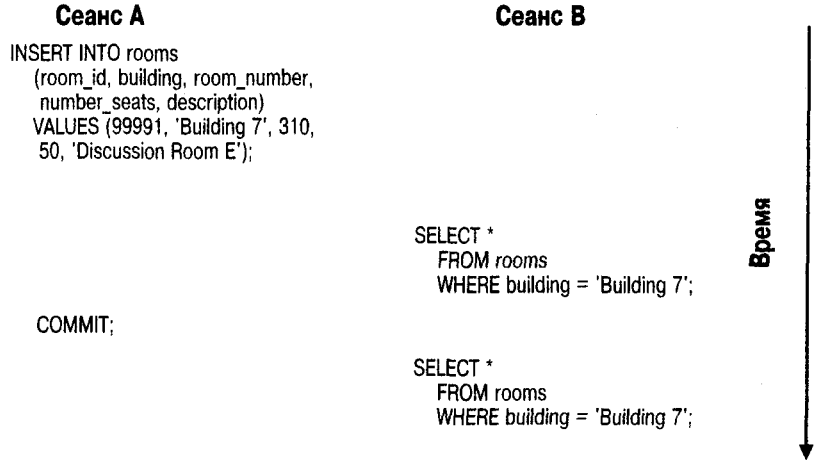
- Вся работа, совершенная этой транзакцией, сохраняется.
- Другие сеансы могут видеть, какие изменения были внесены этой транзакцией.
- Все блокировки, установленные этой транзакцией, снимаются.

Синтаксис оператора COMMIT таков:

```
COMMIT [WORK];
```

Необязательное ключевое слово WORK (работа) применяется для повышения удобочитаемости программ. До тех пор пока транзакция не завершена, изменения, внесенные этой транзакцией, может видеть только тот сеанс, который ее выполняет (рис. 4.6). Вначале сеанс А выдает оператор INSERT. Сеанс В обращается с запросом к таблице **rooms**, но не видит информацию, внесенную сеансом А, так как оператор INSERT не завершен. Затем сеанс А завершает операцию, и во втором операторе SELECT сеанс В увидит вновь введенные строки.

**Рис. 4.6.**
*Два сеанса*




Когда над базой данных выполняется оператор ROLLBACK (откатить), транзакция заканчивается и:

- Вся работа, совершенная этой транзакцией, отменяется, как если бы транзакция и не выполнялась.
- Все блокировки, установленные этой транзакцией, снимаются.

Синтаксис оператора ROLLBACK таков:

```
ROLLBACK [WORK]
```

Как и в операторе COMMIT, ключевое слово WORK необязательно и применяется для удобства чтения программ. Явный оператор ROLLBACK часто используется в том случае, когда программа обнаруживает ошибку и ее дальнейшее выполнение прекращается. Если сеанс отключается от базы данных без

окончания текущей транзакции при помощи оператора COMMIT или ROLLBACK, транзакция автоматически откатывается базой данных.

▼ ВНИМАНИЕ

SQL*Plus автоматически выполняет оператор COMMIT, когда пользователь выходит из программы. Кроме того, средство **автозавершения** (autocommit) выполняет COMMIT после каждого SQL-оператора. Это не оказывает влияния на SQL-операторы внутри блока PL/SQL, так как SQL*Plus не управляет ходом программы до окончания блока.

## Точки сохранения

Как было показано выше, оператор ROLLBACK отменяет всю транзакцию. Однако с помощью команды SAVEPOINT (точка сохранения) можно отменить лишь часть транзакции. Синтаксис оператора SAVEPOINT таков:

SAVEPOINT *имя*;

где *имя* — это имя точки сохранения. Синтаксис имен точек сохранения подчиняется обычным правилам, установленным для идентификаторов SQL (см. главу 2). Обратите внимание, что точки сохранения не описываются в разделе объявлений, так как они являются глобальными для всей транзакции, а она может продолжаться и после окончания блока. После того как точка сохранения создана, можно осуществлять откат программы к этой точке следующим образом:

ROLLBACK [WORK] TO SAVEPOINT *имя*;

Когда формируется оператор ROLLBACK TO SAVEPOINT, происходит следующее:

- Вся работа, выполненная после этой точки сохранения, отменяется. Однако при этом точка сохранения остается активной. При желании откат можно повторить.
- Все блокировки, установленные SQL-операторами после точки сохранения, снимаются, и освобождаются ресурсы, запрошенные SQL-операторами после этой точки.
- Транзакция не заканчивается, так как SQL-операторы находятся в состоянии ожидания.

Рассмотрим фрагмент блока PL/SQL:

```
BEGIN
   INSERT INTO temp_table (char_col) VALUES ('Insert One');
   SAVEPOINT A;
   INSERT INTO temp_table (char_col) VALUES ('Insert Two');
   SAVEPOINT B;
   INSERT INTO temp_table (char_col) VALUES ('Insert Three');
   SAVEPOINT C;
   /* Опущенные операторы. */
   COMMIT;
END;
```

Если вместо опущенных операторов вставить

```
ROLLBACK TO B;
```

третий оператор INSERT и точка сохранения C будут отменены. Однако первые два оператора INSERT будут обработаны. Если же вместо опущенных операторов вставить

```
ROLLBACK TO A;
```

будут отменены второй и третий операторы INSERT и останется только первый.

Команда SAVEPOINT часто используется перед сложным фрагментом транзакции. Если эта часть транзакции выполняется неуспешно, ее можно откатить, что позволит продолжить выполнение предшествующей части.

## Транзакции и блоки

Важно различать транзакции и блоки PL/SQL. Если начинается блок, то это не значит, что начинается транзакция. Аналогично, начало транзакции необязательно должно совпадать с началом блока. Для примера предположим, что выполняется несколько операторов из SQL*Plus:

```
-- Этот пример содержится в файле 1trans.sql.
INSERT INTO classes
```

```
        (department, course, description, max_students,
          current_students, num_credits, room_id)
    VALUES ('CS', 101, 'Computer Science 101', 50, 10, 4, 99998);

BEGIN
  UPDATE rooms
      SET room_id = room_id - 1000;
  ROLLBACK WORK;
END;
```

Сначала выполняется оператор INSERT, а затем анонимный блок PL/SQL. В свою очередь, в блоке выполняются операторы UPDATE и ROLLBACK. Оператор ROLLBACK отменяет результаты работы не только оператора UPDATE, но и предшествующего оператора INSERT, который вместе с блоком является частью одного сеанса и, следовательно, одной транзакции.

Точно так же в одном блоке PL/SQL может содержаться несколько транзакций:

```
-- Этот пример содержится в файле 1block.sql.
DECLARE
  v_NumIterations NUMBER;
BEGIN
  -- Пройдем цикл от 1 до 500, внося значения счетчика в таблицу
  -- temp_table. Через каждые 50 строк будем завершать транзакцию.
  FOR v_LoopCounter IN 1..500 LOOP
      INSERT INTO temp_table (num_col) VALUES (v_LoopCounter);
      v_NumIterations := v_NumIterations + 1;
      IF v_NumIterations = 50 THEN
        COMMIT;
        v_NumIterations := 0;
      END IF;
  END LOOP;
END;
```

С помощью этого блока в таблицу temp_table будут вводиться числа от 1 до 500, причем транзакции будут завершаться через каждые 50 строк. Таким образом, за время выполнения первого блока будет завершено 10 транзакций.

# Итоги

В этой главе рассматривался язык SQL в целом, а также операторы DML и операторы управления транзакциями, используемые в PL/SQL. Кроме того, говорилось о привилегиях и ролях, а также о том, как транзакции помогают обеспечивать согласованность данных. В следующей главе обсуждаются встроенные функции SQL, а в главе 6 — курсоры, применяемые в многострочных запросах.

# Глава 5

## Встроенные SQL-функции

Базовые команды SQL, рассмотренные в главе 4, дополняются множеством встроенных функций. В этой главе обсуждаются различные виды функций и способы их применения.

# Введение

В SQL имеется ряд встроенных функций, которые могут быть вызваны из SQL-оператора. К примеру, ниже приведен оператор SELECT, в котором функция UPPER (верхний регистр) используется для получения имен студентов, представленных прописными буквами, а не в том виде, в котором они хранятся:

```
SELECT UPPER(first_name)
  FROM students;
```

Многие SQL-функции могут быть вызваны и из процедурных операторов PL/SQL. Например, в следующем блоке также используется функция UPPER, в операторе присваивания:

```
DECLARE
  v_FirstName  students.first_name%TYPE;
BEGIN
  v_FirstName  := UPPER('Charlie');
END;
```

SQL-функции можно разделить на три категории в соответствии с типом используемых аргументов. Например, для функции UPPER предполагается использование символьного аргумента. Если указать некорректный аргумент, то до вызова функции он будет автоматически преобразован PL/SQL в соответствии с правилами преобразования типов данных (которые были рассмотрены в главе 2). Кроме того, по числу обрабатываемых строк SQL-функции можно разделить на групповые и однострочные. *Групповые функции* обрабатывают несколько строк данных и возвращают один результат. Их разрешено применять только в списках выбора или в конструкциях HAVING (имея) запросов. В процедурных функциях PL/SQL их использование запрещено. Примером групповой функции является функция COUNT (число строк). *Однострочные функции*, например UPPER, обрабатывают одно значение, а возвращают другое. Их можно использовать в любых SQL-операторах, где разрешено применять выражения, а также в процедурных операторах PL/SQL.

Встроенные функции подробно описаны ниже. В каждом разделе приводится синтаксис, назначение и область применения функции, а также рассматриваются примеры. Функции приведены в порядке английского алфавита. В некоторых функциях используются необязательные аргументы, которые в описании синтаксиса функций заключены в квадратные скобки ([]).

# Символьные функции, возвращающие символьные значения

Эти функции в качестве аргументов используют символы (кроме функции CHR) и возвращают символьные значения. Большинство из них возвращает значения, имеющие тип VARCHAR2; функции, возвращающие другие значения, помечены специально. Тип данных, возвращаемых символьными функциями, имеет те же ограничения, что и основной тип базы данных, т.е. значения типа VARCHAR2 ограничены 2000 символов (4000 в Oracle8), а значения типа CHAR – 255 символами (2000 в Oracle8). При использовании в процедурных операторах эти значения можно присваивать переменным PL/SQL, имеющим тип либо VARCHAR2, либо CHAR.

## CHR

### Синтаксис

CHR(*x*)

**Назначение** Возвращает символ, имеющий код, равный *x* в наборе символов базы данных. Функции CHR и ASCII являются обратными по отношению друг к другу. CHR возвращает символ для кода символа, а ASCII – код символа для символа.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SQL> SELECT CHR(37) a, CHR(100) b, CHR(101) c
  2 FROM dual;

A B C
- - -
% d e
```

## CONCAT

### Синтаксис

CONCAT(*строка_символов1, строка_символов2*)

**Назначение** Возвращает *строку символов 1*, конкатенированную (сцепленную) со *строкой символов 2*. Эта функция идентична операции ||.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT CONCAT('Alphabet ', 'Soup') "Dinner"
  FROM dual;
Dinner
----------------
Alphabet Soup
```

## INITCAP

### Синтаксис

INITCAP(*строка_символов*)

**Назначение** Возвращает *строку символов*, в которой каждое слово начинается с прописной буквы и продолжается строчными. Слова разделяются пробелами или не буквенно-цифровыми символами. Символы, не являющиеся буквами, функцией не изменяются.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT INITCAP('4 scoRE and 7 YEARS ago...') "Speech"
  FROM dual;
```

```
Speech
--------------------------------
4 Score and 7 Years Ago...
```

## LOWER

### Синтаксис

LOWER(*строка_символов*)

**Назначение** Возвращает *строку* со строчными символами. Символы, не являющиеся буквами, не изменяются. Если тип строки символов – CHAR, результат также имеет тип CHAR. Если тип строки символов – VARCHAR2, результат также имеет тип VARCHAR2.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT LOWER('4 scoRE and 7 YEARS ago...') "Speech"
  FROM dual;
Speech
----------------------------
4 score and 7 years ago...
```

## LPAD

### Синтаксис

LPAD(*строка_символов1*, *x* [,*строка_символов2*])

**Назначение** Возвращает *строку символов 1*, дополненную слева до размера *x* символами *строки 2*. Если размер *строки символов 2* меньше *x*, то при необходимости она дублируется. Если размер *строки символов 2* больше *x*, то берутся только первые *x* ее символов. Если *строка символов 2* не указана, по умолчанию ее заменяет знак пробела. Обратите внимание, что *x* указывается как размер строки символов, отображаемой на экране, а не как реальный размер. Если набор символов базы данных состоит из многобайтовых символов, то размер отображаемой строки может быть больше реального размера строки в байтах. Функция LPAD аналогична функции RPAD, но применяется для дополнения строк символами слева, а не справа.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT LPAD('Short String', 15) "First"
  FROM dual;
First
--------------
Short String

SELECT LPAD('Short String', 20, 'XY') "Second"
  FROM dual;
Second
----------------------
XYXYXYXYShort String

SELECT LPAD('Short String', 13, 'XY') "Third"
  FROM dual;
Third
---------------
XShort String
```

## LTRIM

### Синтаксис

LTRIM(*строка_символов1*, *строка_символов2*)

**Назначение** Возвращает *строку символов 1*, в которой удалены крайние левые символы, идентичные символам *строки 2*. Значением по умолчанию для *строки символов 2* является знак пробела. *Строка символов 1* просматривается с левого края, и при встрече первого символа, не совпадающего с символом *строки 2*, возвращается результат. Функция LTRIM аналогична функции RTRIM.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT LTRIM('  End of the string') "First"
```

```
  FROM dual;
First
--------------------
End of the string

SELECT LTRIM('xxxEnd of the string', 'x') "Second"
  FROM dual;
Second
--------------------
End of the string

SELECT LTRIM('xyxyxyEnd of the string', 'xy') "Third"
  FROM dual;
Third
--------------------
End of the string

SELECT LTRIM('xyxyxxxyEnd of the string', 'xy') "Fourth"
  FROM dual;
Fourth
--------------------
End of the string
```

# NLS_INITCAP

## Синтаксис

NLS_INITCAP(*строка_символов* [,*nls_параметр*])

**Назначение** Возвращает *строку символов*, в которой каждое слово начинается с прописной буквы и продолжается строчными. *nls_параметр* указывает последовательность сортировки, отличную от той, которая установлена по умолчанию для данного сеанса. Если параметр не указан, NLS_INITCAP ведет себя точно так же, как и INITCAP. nls_параметр должен иметь следующий вид:

'NLS_SORT = *последовательность_сортировки*'

где *последовательность_сортировки* обозначает последовательность сортировки символов для определенного языка. Более подробно о параметрах NLS и об их использовании рассказано в руководстве Server SQL Reference.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT NLS_INITCAP('ijsbeer', 'NLS_SORT = Xdutch') "Result"
  FROM dual;
Result
--------
IJsbeer
```

# NLS_LOWER

## Синтаксис

NLS_LOWER(*строка_символов* [,*nls_параметр*])

**Назначение** Возвращает *строку* символов, в которой все буквы строчные. Небуквенные символы не изменяются. *nls_параметр* имеет ту же форму и служит для того же, что и в NLS_INITCAP. Если *nls_параметр* не указан, функция NLS_LOWER ведет себя точно так же, как и функция LOWER.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT NLS_LOWER('CITA''DEL', 'NLS_SORT = Xgerman') "Result"
  FROM dual;
Result
-----------
citàdel
```

## NLS_UPPER

### Синтаксис

NLS_UPPER(*строка_символов* [,*nls_параметр*])

**Назначение** Возвращает *строку символов*, в которой все буквы прописные. Небуквенные символы не изменяются. *nls_параметр* имеет ту же форму и служит для того же, что и в NLS_INITCAP. Если *nls_параметр* не указан, функция NLS_UPPER ведет себя точно так же, как и функция UPPER.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT NLS_UPPER('große', 'NLS_SORT = Xgerman') "Result"
  FROM dual;
Result
--------
GROSS
```

## REPLACE

### Синтаксис

REPLACE(*строка_символов, строка_поиска* [,*строка_замены*])

**Назначение** Возвращает *строку символов*, в которой каждое вхождение строки поиска заменяется на *строку замены*. Если *строка замены* не указана, то все вхождения *строки поиска* удаляются из строки символов. Функциональные возможности REPLACE являются подмножеством функциональных возможностей функции TRANSLATE.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT REPLACE('This and That', 'Th', 'B') "First"
  FROM dual;
First
-------------
Bis and Bat
SELECT REPLACE('This and That', 'Th') "Second"
  FROM dual;
Second
------------
is and at
SELECT REPLACE('This and That', NULL) "Third"
  FROM dual;
Third
---------------
This and That
```

# RPAD

## Синтаксис

RPAD(*строка_символов1, x* [,*строка_символов2*])

**Назначение** Возвращает строку символов 1, дополненную справа до размера x символами строки 2. Если размер строки символов 2 меньше x, то при необходимости она дублируется. Если размер строки символов 2 больше x, то берутся только первые x ее символов. Если строка символов 2 не указана, то по умолчанию ее заменяет знак пробела. Обратите внимание, что x указывается как размер строки символов, отображаемой на экране, а не как реальный размер. Если набор символов базы данных состоит из многобайтовых символов, то размер отображаемой строки может быть больше реального размера строки в байтах. Функция RPAD аналогична функции LPAD, но применяется для дополнения строк символами справа, а не слева.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT RPAD('Nifty', 10, '!') "First"
  FROM dual;
First
------------
Nifty!!!!!

SELECT RPAD('Nifty', 10, 'AB') "Second"
  FROM dual;
Second
------------
NiftyABABA
```

# RTRIM

## Синтаксис

RTRIM(*строка_символов1,* [*строка_символов2*])

**Назначение** Возвращает *строку символов 1*, в которой удалены крайние правые символы, идентичные символам *строки 2*. Значением по умолчанию для *строки символов 2* является знак пробела. *Строка символов 1* просматривается с правого края, и при встрече первого символа, не совпадающего с символом *строки 2*, возвращается результат. Функция RLTRIM аналогична функции LTRIM.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT RTRIM('This is the stringxxxxx', 'x') "First"
  FROM dual;
First
--------------------
This is the string

SELECT RTRIM('This is also a stringxxXXxx', 'x') "Second"
  FROM dual;
Second
-----------------------------
This is also a stringxxXX

SELECT RTRIM('This is a string as well', 'well') "Third"
FROM dual;
Third
```

```
-----------------------
This is a string as
```

# SOUNDEX

## Синтаксис

SOUNDEX(*строка_символов*)

**Назначение** Возвращает фонетическое представление строки символов. Это удобно для сравнения слов, правописание которых различно, но произношение одинаково. Фонетическое представление слов описано в книге Дональда Е. Кнута "The Art of Computer Programming, Volume 3: Sorting and Searching" (русский перевод: Д.Е. Кнут. "Искусство программирования на ЭВМ. Т.3. Поиск и сортировка" — М., "Мир", 1978. — *Прим. пер.*). Алгоритм построения фонетического разбора слов следующий:

- Сохраняем первую букву и удаляем все вхождения a, e, h, i, o, u, w, а также y;
- Назначаем оставшимся буквам номера:
  1. b, f, p, v
  2. c, g, j, k, q, s, x, z
  3. d, t
  4. l
  5. m, n
  6. r
- Если несколько одинаковых номеров идут подряд, удаляем все вхождения, кроме первого.
- Оставляем первые 4 байта, дополненные нулями.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT first_name, SOUNDEX(first_name)
  FROM students;
FIRST_NAME  SOUN
----------  ------
Scott       S330
Margaret    M626
Joanne      J500
Manish      M520
Patrick     P362
Timothy     T530

SELECT first_name
  FROM students
  WHERE SOUNDEX(first_name) = SOUNDEX('skit');
FIRST_NAME
---------------
Scott
```

# SUBSTR

## Синтаксис

SUBSTR(*строка_символов, a* [,*b*])

**Назначение** Возвращает часть *строки символов*, начинающуюся с символа с номером *a* и имеющую длину b символов. Если *a* = 0, это равносильно тому, что *a* = 1 (начало строки). Если *b* положительно, возвращаемые символы считаются слева направо. Если *b* отрицательно, символы возвращаются, начиная с конца строки, и считаются справа налево. Если *b* отсутствует, то по умолчанию возвращаются все

символы до конца строки. Если *b* меньше 1, возвращается NULL-значение. Если в качестве *a* или *b* указано число с плавающей точкой, его дробная часть отбрасывается.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT SUBSTR('abc123def', 4, 4) "First"
  FROM dual;
First
------
123d

SELECT SUBSTR('abc123def', -4, 4) "Second"
  FROM dual;
Second
------
3def

SELECT SUBSTR('abc123def', 5) "Third"
  FROM dual;
Third
------
23def
```

## SUBSTRB

### Синтаксис

SUBSTRB(*строка_символов, a* [,*b*])

**Назначение** Ведет себя так же, как и SUBSTR, но *a* и *b* указываются не в символах, а в байтах. Для строк, состоящих из однобайтовых символов, например символов ASCII, функция SUBSTRB идентична функции SUBSTR.

**Область применения** Процедурные и SQL-операторы.

### Пример (для набора двухбайтовых символов)

```
SELECT SUBSTRB("abc123def", 2, 6) "Example"
  FROM dual;
Example
------
bc1
```

## TRANSLATE

### Синтаксис

TRANSLATE(*строка_символов, заменяемая_строка, вносимая_строка*)

**Назначение** Возвращает *строку символов,* в которой все вхождения каждого символа *заменяемой строки* замещаются соответствующим символом *вносимой строки.* Функция TRANSLATE является расширением функции REPLACE. Если *заменяемая строка* длиннее *вносимой,* все ее лишние символы удаляются, поскольку для них нет соответствующих символов во *вносимой строке. Вносимая строка* не может быть пустой. Oracle интерпретирует пустую строку как NULL-значение, а если любой аргумент функции TRANSLATE является NULL-значением, то результат также будет NULL-значением.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT TRANSLATE('abcdefghij', 'abcdef', '123456')
  FROM dual;
TRANSLATE(
----------
123456ghij

SELECT TRANSLATE('abcdefghij', 'abcdefghij', '123456')
  FROM dual;
TRANSL
--------
123456
```

## UPPER

### Синтаксис

UPPER(*строка_символов*)

**Назначение** Возвращает *строку символов*, в которой все символы прописные. Если тип строки символов – CHAR, результат также имеет тип CHAR. Если тип строки символов VARCHAR2, тип результата также VARCHAR2. Небуквенные символы не изменяются.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT UPPER('THE quick bROwn Fox jumped over THE LAZY dOg...')
"Result"
  FROM dual;
Result
--------------------------------------------------
THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG...
```

# Символьные функции, возвращающие числовые значения

Эти функции в качестве аргументов используют символы, а возвращают числовые значения. Аргументы могут иметь тип CHAR или VARCHAR2. Хотя фактически многие результирующие значения являются целыми числами, они имеют тип NUMBER без заданных точности и масштаба.

## ASCII

### Синтаксис

ASCII(*строка_символов*)

**Назначение** Возвращает десятичное представление первого байта *строки символов* согласно применяемому набору символов. Обратите внимание, что эта функция называется ASCII, хотя набор символов может не быть 7-битовым ASCII. Функции CHR и ASCII являются обратными по отношению друг к другу. CHR возвращает символ для кода символа, а ASCII – код символа для символа.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT ASCII(' ')
  FROM dual;
```

```
ASCII('')
-----------
         32

SELECT  ASCII('a')
  FROM  dual;
ASCII('A')
-----------
         97
```

# INSTR

## Синтаксис

INSTR(*строка_символов1, строка_символов2* [,*a* [,*b*]])

**Назначение** Возвращает местоположение *строки символов 2* в *строке символов 1*. *Строка символов 1* просматривается слева, начиная с позиции *a*. Если *a* отрицательно, то *строка символов 1* просматривается справа. Возвращается позиция, указывающая местоположение *b*-го вхождения. Значением по умолчанию как для *a*, так и для *b* является 1, что дает в результате позицию первого вхождения *строки символов 2* в *строке символов 1*. Если при заданных *a* и *b* строка символов 2 не найдена, возвращается 0. Позиция определяется относительно начала *строки символов 1* независимо от значений *a* и *b*.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT  INSTR('Scott''s spot',  'ot',  1,  2)  "First"
  FROM  dual;
First
---------
       11

SELECT  INSTR('Scott''s spot',  'ot',  -1,  2)  "Second"
  FROM  dual;
Second
---------
       3

SELECT  INSTR('Scott''s spot',  'ot',  5)  "Third"
  FROM  dual;
Third
---------
       11

SELECT  INSTR('Scott''s spot',  'ot',  12)  "Fourth"
  FROM  dual;
Fourth
---------
       0
```

# INSTRB

## Синтаксис

INSTRB(*строка_символов1, строка_символов2* [,*a* [,*b*]])

**Назначение** Ведет себя так же, как и INSTR, но a и возвращаемое значение указываются в байтах. Подобно SUBSTRB, для наборов, состоящих из однобайтовых символов, функция INSTRB идентична функции INSTR.

**Область применения** Процедурные и SQL-операторы.

### Пример (для набора двухбайтовых символов)

```
SELECT INSTRB('Scott''s spot', 'ot', 1, 2) "INSTRB"
  FROM dual;
INSTRB
---------
       21
```

## LENGTH

### Синтаксис

LENGTH(*строка_символов*)

**Назначение** Возвращает размер строки в символах. Значения типа CHAR дополняются пробелами, поэтому, если строка символов имеет тип CHAR, в размере указываются и конечные пробелы. Если строка символов является NULL-значением, то функция возвращает NULL.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT LENGTH('Mary had a little lamb') "Length"
  FROM dual;
Length
---------
       22
```

## LENGTHB

### Синтаксис

LENGTHB(*строка_символов*)

**Назначение** Ведет себя так же, как и LENGTH, но возвращаемое значение указывается не в символах, а в байтах. Подобно INSTRB, для наборов, состоящих из однобайтовых символов, функция LENGTHB идентична функции LENGTH.

**Область применения** Процедурные и SQL-операторы.

### Пример (для набора двухбайтовых символов)

```
SELECT LENGTHB('Mary had a little lamb') "Length"
  FROM dual;
Length
---------
       44
```

## NLSSORT

### Синтаксис

NLSSORT(*строка_символов* [,*nls_параметр*])

**Назначение** Возвращает строку байтов, используемую для упорядочения *строки символов*. Все символьные значения преобразуются в строки байтов, как это делается для согласования различных наборов символов базы данных. *nls_параметр* предназначен для того же, что и в функции NLS_INITCAP. Если *nls_параметр* опущен, используется последовательность сортировки, заданная по умолчанию для сеанса. Более подробно о последовательностях сортировки рассказано в разделе "National Language Support" руководства Server SQL Reference.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT NLSSORT('Scott') "NLS"
  FROM dual;
NLS
------------------------------------
53636F747400
```

# Числовые функции

Эти функции в качестве аргументов используют и возвращают значения типа NUMBER. Значения, возвращаемые иррациональными и тригонометрическими функциями, указываются с точностью до 36 десятичных цифр. Значения функций ACOS, ASIN, ATAN и ATAN2 (впервые появившихся в Oracle8) указываются с точностью до 30 десятичных цифр.

## ABS

### Синтаксис

ABS(*x*)

**Назначение** Возвращает абсолютное значение для *x*.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT ABS(-7), ABS(7)
  FROM dual;

  ABS(-7)     ABS(7)
---------   ---------
       7           7
```

## ACOS

### Синтаксис

ACOS(*x*)

**Назначение** Возвращает арккосинус *x*. Область определения *x* от -1 до 1, а область значений от 0 до ¶, выраженная в радианах.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT ACOS(-1), ACOS(0.5)
  FROM dual;

ACOS(-1)       ACOS(0.5)
----------  --------------
3.14159265    1.04719755
```

## ASIN

### Синтаксис

ASIN(*x*)

**Назначение** Возвращает арксинус *x*. Область определения *x* от -1 до 1, а область значений от -¶/2 до ¶/2, выраженная в радианах.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT ASIN(1), ASIN(0.5)
  FROM dual;

  ASIN(1)        ASIN(0.5)
----------     ------------
1.57079633       .523598776
```

## ATAN

### Синтаксис

ATAN(*x*)

**Назначение** Возвращает арктангенс *x*. Область значений от -¶/2 до ¶/2, выраженная в радианах.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT ATAN(0), ATAN(0.5)
  FROM dual;

   ATAN(0)      ATAN(0.5)
--------      -------------
       0        .463647609
```

## ATAN2

### Синтаксис

ATAN2(*x*, *y*)

**Назначение** Возвращает арктангенс *x* и *y*. Область значений от -¶/2 до ¶/2; она зависит от знаков *x* и *y* и выражена в радианах. ATAN2(*x*, *y*) — это то же самое, что и ATAN(*x*/*y*).

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT ATAN2(1, 2)
  FROM dual;

ATAN(1,2)
------------
.463647609
```

## CEIL

### Синтаксис

CEIL(*x*)

**Назначение** Возвращает наименьшее целое число, большее или равное *x*.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT CEIL(18.1), CEIL(-18.1)
  FROM dual;
CEIL(18.1)  CEIL(-18.1)
---------   -------------
        19            -18
```

## COS

### Синтаксис

COS(*x*)

**Назначение** Возвращает косинус угла в *x* радиан.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT COS(0), COS(90 * 3.14159265359/180)
  FROM dual;
     COS(0)  COS(90*3.14159265359/180)
------------ --------------------------
           1                         -1
```

## COSH

### Синтаксис

COSH(*x*)

**Назначение** Возвращает гиперболический косинус *x*.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT COSH(0), COSH(90 * 3.14159265359/180)
  FROM dual;
  COSH(0)   COSH(90*3.14159265359/180)
---------   ----------------------------
        1                     2.5091785
```

## EXP

### Синтаксис

EXP(*x*)

**Назначение** Возвращает *e* в степени *x*. (*e* = 2.71828183...)

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT EXP(1), EXP(2.7)
  FROM dual;
  EXP(1)      EXP(2.7)
---------     -----------
2.7182818      14.879732
```

# FLOOR

## Синтаксис

FLOOR(*x*)

**Назначение** Возвращает наибольшее целое число, меньшее или равное *x*.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT  FLOOR(-23.5),  FLOOR(23.5)
  FROM  dual;
FLOOR(-23.5)   FLOOR(23.5)
------------    -----------
         -24           23
```

# LN

## Синтаксис

LN(*x*)

**Назначение** Возвращает натуральный логарифм *x*. (*x* должно быть больше 0.)

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT  LN(100)
  FROM  dual;
   LN(100)
------------
 4.6051702
```

# LOG

## Синтаксис

LOG(*x*, *y*)

**Назначение** Возвращает логарифм *y* по основанию *x*. Основание должно быть положительным числом, отличным от 0 и 1, а *y* может быть любым положительным числом.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT  LOG(2,  32),  LOG(5,  25)
  FROM  dual;
LOG(2,32)  LOG(5,25)
--------    --------
       5           2
```

# MOD

## Синтаксис

MOD(*x*, *y*)

**Назначение** Возвращает остаток от деления *x* нацело на *y*. Если *y* равно 0, то возвращает *x*.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT MOD(23, 5), MOD(4, 1.3)
  FROM dual;
MOD(23,5)   MOD(4,1.3)
--------    ----------
       3          .1
```

**▼ ВНИМАНИЕ** Когда значение *x* отрицательно, функция MOD ведет себя не так, как классическая функция деления по модулю. Классическое деление по модулю можно определить следующим образом:

$$x - y * \text{FLOOR}(x/y)$$

## POWER

### Синтаксис

POWER(*x*, *y*)

**Назначение** Возвращает *x* в степени *y*. Основание *x* и порядок *y* не должны быть обязательно положительными целыми числами, но если *x* — отрицательное число, то *y* должен быть целым числом.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT POWER(4, 3), POWER(1.1, 2.6), POWER(25, -2), POWER(-2, 3)
  FROM dual;
POWER(4,3)  POWER(1.1,2.6)  POWER(25,-2)  POWER(-2,3)
----------  --------------  ------------  ------------
        64        1.281212         .0016            -8
```

## ROUND

### Синтаксис

ROUND(*x*, [,*y*])

**Назначение** Возвращает *x*, округленное до *y* разрядов справа от десятичной точки. Значением по умолчанию для *y* является 0; при этом *x* округляется до ближайшего целого числа. Если *y* — отрицательное число, то округляются цифры слева от десятичной точки. *y* должен быть целым числом.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT ROUND(1.56), ROUND(1.56, 1), ROUND(12.34, -2)
  FROM dual;
ROUND(1.56)  ROUND(1.56,1)  ROUND(12.34,-2)
-----------  -------------  ----------------
          2            1.6                 0
```

## SIGN

### Синтаксис

SIGN(*x*)

**Назначение** Если $x < 0$, возвращает -1. Если $x = 0$, возвращает 0. Если $x > 0$, возвращает 1.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT SIGN(-47.3), SIGN(0), SIGN(47.3)
  FROM dual;
SIGN(-47.3)  SIGN(0)  SIGN(47.3)
-----------  --------  -----------
         -1         0            1
```

## SIN

### Синтаксис

SIN(*x*)

**Назначение** Возвращает синус угла в *x* радиан.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT SIN(0), SIN(60 * 3.14159265359/180)
  FROM dual;
SIN(0)  SIN(60*3.14159265359/180)
------  -----------------------------
     0                       .8660254
```

## SINH

### Синтаксис

SINH(*x*)

**Назначение** Возвращает гиперболический синус *x*.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT SINH(0), SINH(60 * 3.14159265359/180)
  FROM dual;
SINH(0)  SINH(60*3.14159265359/180)
-------  ------------------------------
      0                       1.2493671
```

## SQRT

### Синтаксис

SQRT(*x*)

**Назначение** Возвращает квадратный корень *x*. *x* не может быть отрицательным.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT SQRT(64), SQRT(97.654)
  FROM dual;
SQRT(64)  SQRT(97.654)
--------  -------------
       8     9.8820038
```

## TAN

### Синтаксис

TAN(*x*)

**Назначение** Возвращает тангенс угла в *x* радиан.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT TAN(0), TAN(-60 * 3.14159265359/180)
  FROM dual;
TAN(0)  TAN(-60*3.14159265359/180)
------  -----------------------------
     0                     -1.732051
```

## TANH

### Синтаксис

TANH(*x*)

**Назначение** Возвращает гиперболический тангенс *x*.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT TANH(0), TANH(-60 * 3.14159265359/180)
  FROM dual;
TANH(0)  TANH(-60*3.14159265359/180)
-------  ------------------------------
      0                      -.7807144
```

## TRUNC

### Синтаксис

TRUNC(*x*, [,*y*])

**Назначение** Возвращает *x*, усеченное (не округленное) до *y* десятичных разрядов. Значением по умолчанию для *y* является 0; при этом *x* усекается до целого числа. Если *y* отрицательно, то усекаются цифры слева от десятичной точки.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT TRUNC(-123.456), TRUNC(-123.456, 1), TRUNC(-123.456, -1)
  FROM dual;
TRUNC(-123.456)  TRUNC(-123.456,1)  TRUNC(-123.456,-1)
---------------  -----------------  -------------------
           -123             -123.4                 -120
```

# Временные функции

Аргументы временных функций имеют тип DATE. За исключением функции MONTHS_BETWEEN, которая возвращает значение типа NUMBER, все функции возвращают значения типа DATE. В этом разделе рассмотрены также арифметические операции с датами.

# ADD_MONTHS

## Синтаксис

ADD_MONTHS(*d*, *x*)

**Назначение** Возвращает дату *d* плюс *x* месяцев. *x* может быть любым целым числом. Если в месяце, полученном в результате, число дней меньше, чем в месяце *d*, то возвращается последний день месяца-результата. Если число дней не меньше, то день месяца-результата и день месяца *d* совпадают. Временные компоненты даты *d* и результата одинаковы.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT   ADD_MONTHS('02-FEB-91', 1), ADD_MONTHS('19-JAN-87', 1),
         ADD_MONTHS('30-JAN-87', 13)
  FROM dual;
ADD_MONTH  ADD_MONTH  ADD_MONTH
---------  ---------- ----------
02-MAR-91  19-FEB-87  29-FEB-88
```

# LAST_DAY

## Синтаксис

LAST_DAY(*d*)

**Назначение** Возвращает дату последнего дня того месяца, в который входит *d*. Эту функцию можно применять для определения количества дней, оставшихся в текущем месяце.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT   LAST_DAY('12-APR-71') "Current"
         LAST_DAY('12-APR-71') - TO_DATE('12-APR-71') "Days Left"
  FROM dual;
Current     Days Left
--------- ------------
30-APR-71           18
```

# MONTHS_BETWEEN

## Синтаксис

MONTHS_BETWEEN(*дата1*, *дата2*)

**Назначение** Возвращает число месяцев между *датой 1* и *датой 2*. Если дни в *дате 1* и *дате 2* совпадают или если обе даты являются последними днями своих месяцев, то результат представляет собой целое число. В противном случае результат будет содержать дробную часть (по отношению к 31-дневному месяцу).

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT   MONTHS_BETWEEN('12-APR-71', '12-MAR-97') "First"
         MONTHS_BETWEEN('12-APR-71', '22-MAR-60') "Second"
  FROM dual;
First      Second
-----  ----------
 -311  132.67742
```

# NEW_TIME

## Синтаксис

NEW_TIME(*d, пояс1, пояс2*)

**Назначение** Возвращает дату и время часового *пояса 2* для того момента, когда датой и временем часового *пояса 1* является *d*. *Пояс 1 и пояс 2* – это строки символов, и их значение описано в таблице 5.1.

**ТАБЛИЦА 5.1.**

| Строка | Часовой пояс |
|---|---|
| AST | Atlantic Standard Time – атлантическое (нью-йоркское) поясное время |
| ADT | Atlantic Daylight Time – атлантическое (нью-йоркское) летнее поясное время |
| BST | Bering Standard Time – поясное время Берингова пролива |
| BDT | Bering Daylight Time – летнее поясное время Берингова пролива |
| CST | Central Standard Time – центральное поясное время |
| CDT | Central Daylight Time – центральное летнее поясное время |
| EST | Eastern Standard Time – восточное поясное время |
| EDT | Eastern Daylight Time – восточное летнее поясное время |
| GMT | Greenwich Mean Time – среднее время по Гринвичскому меридиану |
| HST | Alaska-Hawaii Standard Time – поясное время Аляски и Гавайских островов |
| HDT | Alaska-Hawaii Daylight Time – летнее поясное время Аляски и Гавайских островов |
| MST | Mountain Standard Time – поясное время Среднего Запада США |
| MDT | Mountain Daylight Time – летнее поясное время Среднего Запада США |
| NST | Newfounland Standard Time – поясное время Ньюфаундленда |
| PST | Pacific Standard Time – тихоокеанское поясное время |
| PDT | Pacific Daylight Time – тихоокеанское летнее поясное время |
| YST | Yukon Standard Time – поясное время Юкона |
| YDT | Yukon Daylight Time – летнее поясное время Юкона |

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT TO_CHAR(NEW_TIME(TO_DATE('12-APR-71 12:00:00'
                                'DD-MON-YY HH24:MI:SS'),
                        'PST', 'EST'),
               'DD-MON-YY HH24:MI:SS') "Pacific -> Eastern"
  FROM dual;
Pacific ->    Eastern
---------    ----------
12-APR-71    15:00:00
```

# NEXT_DAY

## Синтаксис

NEXT_DAY(*d, строка_символов*)

**Назначение** Возвращает дату первого дня, наступающего после даты *d* и обозначенного строкой символов. *Строка символов* указывает день недели на языке текущего сеанса. Временной компонент возвращаемого значения тот же, что и временной компонент *d*. Регистр символов строки значения не имеет.

**Область применения** Процедурные и SQL-операторы.

## Пример

В этом примере возвращается дата первого четверга, следующего за 12 апреля 1971 года.

```
SELECT NEXT_DAY('12-APR-71', 'thursday') "Result"
  FROM dual;
Result
----------
15-APR-71
```

# ROUND

## Синтаксис

ROUND(*d* [, *формат*])

**Назначение** Округляет дату *d* до единицы, указанной *форматом*. *Форматы*, применяемые в функциях ROUND и TRUNC, описаны в таблице 5.2 (в соответствии с таблицей 3.11 руководства SQL Language Reference). Если *формат* не указан, принимается формат по умолчанию 'DD', который округляет d до ближайшего дня.

### ТАБЛИЦА 5.2. Форматы дат функций ROUND и TRUNC

| Формат | Единица округления или усечения |
|---|---|
| CC, SCC | Век |
| SYYYY, YYYY, YEAR, SYEAR, YYY, YY, Y | Год (округляется до 1 июля) |
| IYYY, IYY, IY, I | Год ISO |
| Q | Квартал (округляется до шестнадцатого дня второго месяца квартала) |
| MONTH, MON, MM, RM | Месяц (округляется до шестнадцатого дня) |
| WW | Тот же день недели, что и первый день года |
| IW | Тот же день недели, что и первый день года ISO |
| W | Тот же день недели, что и первый день месяца |
| DDD, DD, J | День |
| Day, DY, D | Первый день недели |
| HH, HH12, HH24 | Час |
| MI | Минута |

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT ROUND(TO_DATE(12-APR-71'), 'MM') "Nearest Month"
  FROM dual;
Nearest Month
--------------
01-APR-71
```

# SYSDATE

## Синтаксис

SYSDATE

**Назначение** Возвращаются текущие дата и время с типом DATE. Аргументов нет. При использовании функции SYSDATE в распределенных SQL-операторах она возвращает дату и время локальной базы данных.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT TO_CHAR(SYSDATE, 'Month DD, YYYY HH24:MI:SS') "Now"
  FROM dual;
Now
---------------------------
April 13, 1997 23:31:17
```

## TRUNC

### Синтаксис

TRUNC(*d* [, *формат*])

**Назначение** Возвращает дату *d*, усеченную до единицы, указанной *форматом*. Применяются те же форматы, что и в функции ROUND (см. таблицу 5.2). Если формат опущен, то принимается формат по умолчанию 'DD', который усекает *d* до ближайшего дня.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT TRUNC( TO_DATE('12-APR-71 13:21:00',
              'DD-MON-YY HH24:MI:SS'),
              'Year') "First Day"
  FROM dual;
First Day
-----------
01-JAN-71
```

## Арифметические операции с датами

Использование арифметических операций по отношению к датам и числам описано в таблице 5.3. Обратите внимание, что при вычитании одной даты из другой результатом является число.

**ТАБЛИЦА 5.3. Семантика арифметических операций с данными**

| Операция | Тип возвращаемого значения | Результат |
|---|---|---|
| *d1 - d2* | NUMBER | Возвращается разница в днях между *d1* и *d2*. Это значение является действительным числом, где дробная часть означает неполный день. |
| *d1 + d2* | — | Запрещено — разрешается только вычитание одной даты из другой. |
| *d1 + n* | DATE | К *d1* добавляется *n* дней и возвращается результат, имеющий тип DATE. *n* может быть вещественным числом, содержащим неполный день. |
| *d1 - n* | DATE | Из *d1* вычитается *n* дней и возвращается результат, имеющий тип DATE. *n* может быть вещественным числом, содержащим неполный день. |

Ниже приведены примеры правильных арифметических выражений, в которых задействованы даты.

```
SELECT SYSDATE "Today", SYSDATE + 1 "Tomorrow"
  FROM dual;
Today      Tomorrow
---------  ----------
```

```
29-MAR-97  30-MAR-97
SELECT   TO_DATE('12-APR-71  12:00:00',  'DD-MON-YY  HH24:MI:SS')  -
         TO_DATE('15-MAR-71  15:00:00',
                 'DD-MON-YY  HH24:MI:SS')  "Difference"
  FROM  dual;
Difference
------------
    27.875
```

# Функции преобразования

Функции преобразования используются для преобразования типов данных PL/SQL, который выполняет большую часть преобразований автоматически, неявно вызывая определенную функцию. Однако пользователь не может управлять форматом данных во время неявного вызова функции, что иногда делает текст программ менее понятным. Поэтому считается хорошим стилем явно указывать функции преобразования, а не полагаться на неявное преобразование типов данных, производимое PL/SQL.

## CHARTOROWID

### Синтаксис

CHARTOROWID(*строка_символов*)

**Назначение** Преобразует значение типа CHAR или VARCHAR, содержащее идентификатор строки (ROWID) во внешнем формате, – во внутренний двоичный формат. Аргумент *строка_символов* должен быть 18-символьной строкой, содержащей идентификатор строки, который представлен во внешнем формате (см. главу 2). Внешние форматы в Oracle7 и Oracle8 различны. Функция CHARTOROWID является инверсией функции ROWIDTOCHAR.

**Область применения** Процедурные и SQL-операторы.

### Пример (используется формат идентификатора строки Oracle7)

```
SELECT description
  FROM classes
  WHERE rowid = CHARTOROWID('0000002D.0002.0002');
DESCRIPTION
-------------
Economics 203
```

## CONVERT

### Синтаксис

CONVERT(*строка_символов, целевой_набор* [, *исходный_набор*]

**Назначение** Преобразует строку, состоящую из символов набора, который указан как *исходный набор*, в строку, состоящую из символов набора, который указан как *целевой набор*. Если исходный набор не задан, по умолчанию принимается набор символов, используемый в базе данных. Наиболее распространенные наборы символов приведены в таблице 5.4.

Для полного преобразования строк целевой набор должен содержать представление всех символов, входящих в исходный набор. Если это не так, в целевом наборе используется набор символов замены, который является частью описания собственно набора символов.

**Область применения** Процедурные и SQL-операторы.

**ТАБЛИЦА 5.4.**

| Идентификатор набора символов | Описание |
|---|---|
| US7ASCII | U.S. (США) 7-битовый ASCII. Этот набор символов используется в большинстве операционных систем Unix и в базах данных Oracle, функционирующих в Unix. |
| WE8DEC | West European (западноевропейский) 8-битовый компании DEC |
| WE8HP | West European (западноевропейский) 8-битовый HP (компании Хьюлетт-Паккард) для LaserJet |
| F7DEC | French (французский) 7-битовый компании DEC |
| WE8EBCDIC500 | West European (западноевропейский) EBCDIC Code Page 500 компании IBM |
| WE8PC850 | PC Code Page-500 компании IBM. Этот набор символов используется в большинстве систем PC и в системах Oracle, функционирующих на PC. |
| WE8ISO8859P1 | West European (западноевропейский) 8-битовый ISO-8859-1 |

### Пример (из руководства Server SQL Reference)

```
SELECT CONVERT('Groß', 'WE8HP', 'WE8DEC') "Conversion"
  FROM dual;
Conversion
----------
Groß
```

## HEXTORAW

### Синтаксис

HEXTORAW(*строка_символов*)

**Назначение** Преобразует двоичное значение, представленное *строкой символов*, в значение типа RAW. *Строка символов* должна содержать шестнадцатеричные значения. Каждые два символа строки представляют один байт результирующего значения RAW. Функции HEXTORAW и RAWTOHEX являются обратными по отношению друг к другу.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
INSERT INTO raw_table (raw_column)
  VALUES (HEXTORAW('017D3F'));
```

## RAWTOHEX

### Синтаксис

RAWTOHEX(*значение*)

**Назначение** Преобразует *значение* типа RAW в строку символов, состоящую из шестнадцатеричных значений. Каждый байт значения RAW преобразуется в двухсимвольную строку. Функции RAWTOHEX и HEXTORAW являются обратными по отношению друг к другу.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT RAWTOHEX(raw_column)
  FROM raw_table;
RAW_COLUMN
------------
017D3F
```

## ROWIDTOCHAR

### Синтаксис

ROWIDTOCHAR(*идентификатор_строки*)

**Назначение** Преобразует *идентификатор строки* (значение типа ROWID) в его внешнее представление в виде 18-символьной строки, которое различно в Oracle7 и Oracle8. Функции ROWIDTOCHAR и CHARTOROWID являются обратными по отношению друг к другу.

**Область применения** Процедурные и SQL-операторы.

### Пример (используется формат идентификатора строки Oracle7)

```
SELECT ROWIDTOCHAR(rowid)
  FROM classes;
ROWIDTOCHAR(ROWID)
------------------
0000002D.0000.0002
0000002D.0002.0002
0000002D.0003.0002
```

## TO_CHAR (даты)

### Синтаксис

TO_CHAR (*d* [, *формат*[, *nls_параметр*]])

**Назначение** Преобразует дату *d* в строку символов типа VARCHAR2. Если *формат* указан, он используется для управления структурой результата. Строка формата состоит из *элементов формата*. Каждый элемент возвращает часть фрагмента даты, например месяц. Элементы формата дат описаны в таблице 5.5. Если формат не указан, используется формат дат по умолчанию для конкретного сеанса. Если указан *nls_параметр*, то он управляет выбором языка, используемого для обозначения месяца и дня в возвращаемой строке. Для *nls_параметра* применяется следующий формат:

'NLS_DATE_LANGUAGE = *язык*'

где *язык* обозначает используемый язык. Более подробно о функции TO_CHAR и об элементах формата рассказано в руководстве Server SQL Reference.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT TO_CHAR(SYSDATE, 'DD-MON-YY', HH24:MI:SS') "Right Now"
  FROM dual;
Right Now
---------

15-NOV-95 01:17:14
```

## ТАБЛИЦА 5.5. Элементы формата дат

| Элемент формата дат | Описание |
|---|---|
| Знаки пунктуации | Все знаки пунктуации дублируются в результирующей строке символов. |
| "Текст" | Текст, заключенный в двойные кавычки, также дублируется. |
| AD, A.D. | Показатель "нашей эры" (с точками или без точек). |
| AM, A.M. | Показатель времени до полудня (с точками или без точек). |
| BC, B.C. | Показатель "до нашей эры" (с точками или без точек). |
| CC, SCC | Век. SCC возвращает даты "до нашей эры" как отрицательные значения. |
| D | День недели (1-7). |
| DAY[1] | Название дня, дополненное пробелами до девяти символов. |
| DD | День месяца (1-31). |
| DDD | День года (1-366). |
| DY[1] | Сокращенное название дня. |
| IW | Неделя года (1-52, 1-53); в основе лежит стандарт ISO. |
| IYY, IY, I | Последние три, две или одна цифры года ISO. |
| IYYY | Четырехцифровое обозначение года, основанное на стандарте ISO. |
| HH, HH12 | Час дня (1-12). |
| HH24 | Час дня (0-23). |
| J | День по Юлианскому календарю. Число дней с 1 января 4712 года до н.э. Соответствующий результат будет целым значением. |
| MI | Минута (0-59). |
| MM | Месяц (1-12). JAN = 1, DEC = 12. |
| MONTH[1] | Название месяца, дополненное пробелами до девяти символов. |
| MON[1] | Сокращенное название месяца. |
| PM, P.M. | Показатель времени после полудня (с точками или без точек). |
| Q | Квартал года (1-4). С января по март — первый квартал. |
| RM | Месяц, обозначенный римскими цифрами (I-XII). JAN = I, DEC = XII. |
| RR | Последние две цифры года для других веков. |
| SS | Секунда (0-59). |
| SSSSS | Секунды после полуночи (0-86339). Модель формата 'J.SSSSS' всегда будет давать в результате числовое значение. |
| WW | Неделя года (1-53). Неделя 1 начинается с первого дня года и продолжается до седьмого дня. Таким образом, недели не всегда начинаются с воскресенья (как принято в США. — *Прим. пер.*) |
| W | Неделя месяца (1-5). Недели определяются так же, как и для элемента WW. |
| Y, YYY | Год с запятой в указанной позиции. |
| YEAR, SYEAR[1] | Год, записанный буквами. SYEAR возвращает даты до нашей эры как отрицательные значения. |

**ТАБЛИЦА 5.5. Элементы формата дат (продолжение)**

| Элемент формата дат | Описание |
|---|---|
| YYYY, SYYYY | Четырехцифровой год. SYYYY возвращает даты до нашей эры как отрицательные значения. |
| YYY, YY, Y | Последние (последняя) три, две или одна цифра (цифры) года |

[1]Эти элементы чувствительны к регистру символов. Например, 'MON' возвращает 'JAN', а 'Mon' — 'Jan'.

# TO_CHAR (метки)

## Синтаксис

TO_CHAR (*метка* [, *формат*])

**Назначение** Преобразует *метку* типа MLSLABEL к типу VARCHAR2. Если *формат* метки указан, он используется при преобразовании. Если формат не указан, применяется формат метки, заданный по умолчанию. Эта функция нужна только при работе с системой Trusted Oracle. Функция TO_LABEL и данная версия функции TO_CHAR являются обратными по отношению друг к другу.

**Область применения** Процедурные и SQL-операторы в базе данных Trusted Oracle.

## Пример

Обратитесь к руководству Trusted Oracle Server Administrator's Guide.

# TO_CHAR (числа)

## Синтаксис

TO_CHAR (*число* [, *формат* [, *nls_параметр*]])

**Назначение** Преобразует аргумент, являющийся значением типа NUMBER, в значение типа VARCHAR2. Если *формат* указан, то он управляет процессом преобразования. Применяемые форматы чисел приведены в таблице 5.6. Если формат не указан, то результирующая строка содержит столько символов, сколько необходимо для хранения значащих цифр числа. *nls_параметр* применяется для указания десятичных разделителей и разделителей групп, а также для указания символа денежной единицы. Он имеет следующий формат:

'NLS_NUMERIC_CHARS = "*dg*" NLS_CURRENCY = "*строка_символов*"

где *d* и *g* представляют соответственно десятичный разделитель и разделитель групп, а *строка_символов* — знак денежной единицы. Например, в США десятичным разделителем обычно является точка (.), разделителем групп — запятая (,), а символом денежной единицы — знак доллара ($). Подробно о средстве поддержки национальных языков (National Language Support) рассказано в руководстве Oracle Server SQL Reference.

**ТАБЛИЦА 5.6. Элементы формата чисел**

| Элемент формата | Пример строки данного формата | Результат |
|---|---|---|
| 9 | 99 | Каждая цифра 9 представляет значащую цифру результата. Число значащих цифр возвращаемого значения равно числу цифр 9; отрицательное значение предваряется знаком минуса. Все начальные нули заменяются пробелами. |
| 0 | 0999 | Возвращается число с начальными нулями, а не с пробелами. |
| 0 | 9990 | Возвращается число с конечными нулями, а не с пробелами. |
| $ | $999 | Возвращаемое значение предваряется знаком доллара независимо от используемого символа денежной единицы. Это можно применять совместно с начальными или конечными нулями. |

**ТАБЛИЦА 5.6. Элементы формата чисел (продолжение)**

| Элемент формата | Пример строки данного формата | Результат |
|---|---|---|
| B | B999 | Вместо нулевой целой части десятичного числа возвращаются пробелы. |
| MI | 999MI | Возвращается отрицательное число, у которого знак минуса указан не в начале, а в конце. В положительном значении на этом месте будет находиться пробел. |
| S | S9999 | Возвращаемое число предваряется знаком: + для положительных чисел, - для отрицательных чисел. |
| S | 9999S | Возвращаемое число заканчивается знаком: + для положительных чисел, - для отрицательных чисел. |
| PR | 99PR | Возвращается отрицательное число в угловых скобках "<" и ">". В положительном числе на этих местах будут находиться пробелы. |
| D | 99D9 | Возвращается число с десятичной точкой в указанной позиции. Число цифр 9 с обеих сторон указывает максимальное число цифр. |
| G | 9G999 | Возвращается число с разделителем групп в указанной позиции. G может появляться в строке формата неоднократно. |
| C | C99 | Возвращается число с символом денежной единицы ISO в указанной позиции. C может появляться в строке формата неоднократно. |
| L | L999 | Возвращается число с символом денежной единицы национального языка в указанной позиции. |
| , | 999,999 | Возвращается число с запятой в указанной позиции независимо от выбранного разделителя групп. |
| . | 99.99 | Возвращается число с десятичной точкой в указанной позиции независимо от выбранного десятичного разделителя. |
| V | 99V999 | Возвращается число, умноженное на $10^n$, где $n$ — это число цифр 9 после V. При необходимости значение округляется. |
| EEEE | 9.99EEEE | Возвращается число в экспоненциальном представлении. |
| RM | RM | Возвращается число при помощи римских цифр верхнего регистра. |

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT TO_CHAR(123456, '99G99G99') "Result"
  FROM dual;
Result
---------
12,34,56
SELECT TO_CHAR(123456, 'L99G99D99'
                'NLS_NUMERIC_CHARACTERS = '',.''
                 NLS_CURRENCY = ''Money'' ') "Result 2"
  FROM dual;
Result 2
--------------
Money12,34.56
```

# TO_DATE

## Синтаксис

TO_DATE (*строка_символов* [, *формат* [, *nls_параметр*]])

**Назначение** Преобразует *строку символов* типа CHAR или VARCHAR2 в дату (значение типа DATE). *Формат* – это формат дат (см. таблицу 5.5). Если он не указан, используется формат дат по умолчанию для данного сеанса. *nls_параметр* применяется точно так же, как и в функции TO_CHAR. Функции TO_DATE и TO_CHAR являются обратными по отношению друг к другу.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
DECLARE
  v_CurrentDate  DATE;
BEGIN
  v_CurrentDate  :=  TO_DATE('January  7,  1973',  'Month  DD,  YYYY');
END;
```

# TO_LABEL

## Синтаксис

TO_LABEL(*строка_символов* [, *формат*])

**Назначение** Преобразует *строку символов* к типу MLSLABEL. Строка символов может иметь тип CHAR или VARCHAR2. Если *формат* указан, то он используется при преобразовании. Если он не указан, используется формат преобразования, заданный по умолчанию. Эта функция нужна только при работе с системой Trusted Oracle. Функции TO_LABEL и TO_CHAR являются обратными по отношению друг к другу.

**Область применения** Процедурные и SQL-операторы в базе данных Trusted Oracle.

## Пример

Обратитесь к руководству Trusted Oracle Server Administrator's Guide.

# TO_MULTI_BYTE

## Синтаксис

TO_MULTI_BYTE(*строка_символов*)

**Назначение** Возвращает *строку символов*, в которой все однобайтовые символы заменены на эквивалентные многобайтовые символы. Эта функция применяется, когда набор символов базы данных состоит из однобайтовых и из многобайтовых символов. Если это не так, строка не изменяется. Функции TO_MULTI_BYTE и TO_SINGLE_BYTE являются обратными по отношению друг к другу.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT  TO_MULTI_BYTE('Hello')  "Multi"
  FROM  dual;
Multi
--------
Hello
```

# TO_NUMBER

## Синтаксис

TO_NUMBER (*строка_символов* [, *формат* [, *nls_параметр*]])

**Назначение** Преобразует *строку символов* типа CHAR или VARCHAR2 в число (значение типа NUMBER). Если *формат* указан, строка символов должна соответствовать формату числа. Назначение *nls_параметра* то же, что и в функции TO_CHAR. Функции TO_NUMBER и TO_CHAR являются обратными по отношению друг к другу.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
DECLARE
  v_Num  NUMBER;
BEGIN
  v_Num  :=  TO_NUMBER('$12345.67',  '$99999.99');
END;
```

## TO_SINGLE_BYTE

### Синтаксис

TO_SINGLE_BYTE(*строка_символов*)

**Назначение** Преобразует все многобайтовые символы строки в эквивалентные однобайтовые символы. Эта функция применяется, когда набор символов базы данных состоит как из однобайтовых, так и из многобайтовых символов. Если это не так, строка не изменяется. Функции TO_SINGLE_BYTE и TO_MULTI_BYTE *являются обратными по отношению друг к другу.*

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT  TO_SINGLE_BYTE('Greetings')  "Single"
  FROM  dual;
Single
----------
Greetings
```

# Групповые функции

Групповые функции обрабатывают по нескольку строк, но возвращают один результат. В этом их отличие от однострочных функций, которые возвращают результат для каждой строки базы данных. Например, групповая функция COUNT возвращает число выбираемых строк. Эти функции можно применять только в списках выбора запросов и в конструкции GROUP BY.

В большинстве этих функций допускается использование квалификаторов (уточнителей) аргументов: DISTINCT (*отличные от других*) и ALL (*все*). *Если указывается* DISTINCT, *рассматриваются только те* значения, возвращаемые запросом, которые отличны от других. Когда используется квалификатор ALL, функция рассматривает все значения, возвращаемые запросом. Если не указано другое условие, ALL принимается как параметр, заданный по умол1чанию.

## AVG

### Синтаксис

AVG([DISTINCT | ALL] *столбец*)

**Назначение** Возвращает среднее для значений *столбца*

**Область применения** Только списки выбора запросов и конструкции GROUP BY.

### Пример

```
SELECT  AVG(number_seats)
  FROM  rooms;
```

```
AVG(NUMBER_SEATS)
--------------------
                 330
```

# COUNT

## Синтаксис

COUNT([* | DISTINCT | ALL] *столбец*)

**Назначение** Возвращает число строк в запросе. Если указана звездочка *, возвращается общее число строк. Если указан пункт списка выбора, то подсчитываются не-NULL-значения.

**Область применения** Только списки выбора запросов и конструкции GROUP BY.

## Пример

```
SELECT  COUNT(*)
  FROM  students;
  COUNT  (*)
------------
         6
SELECT  COUNT(DISTINCT  major)  "Majors"
  FROM  students;
      Majors
-------------
           3
SQL>  SELECT  major,  COUNT(major)
  2    FROM  students
  3    GROUP  BY  major;

MAJOR                          COUNT(MAJOR)
-----------------------------  -------------
Computer  Science                          2
Economics                                  2
History                                    3
Music                                      2
Nutrition                                  2
```

# GLB

## Синтаксис

GLB([DISTINCT | ALL] *метка*)

**Назначение** Возвращает наибольшую нижнюю границу *метки*. Эта функция имеет смысл только в Trusted Oracle.

**Область применения** Только списки выбора запросов и конструкции GROUP BY в базе данных Trusted Oracle.

## Пример

Примеры и определение наибольшей нижней границы приведены в руководстве Trusted Oracle Server Administrator's Guide.

# LUB

## Синтаксис

LUB([DISTINCT | ALL] *метка*)

**Назначение** Возвращает наименьшую нижнюю границу *метки*. Эта функция имеет смысл только в Trusted Oracle.

**Область применения** Только списки выбора запросов и конструкции GROUP BY.

### Пример

Примеры и определение наименьшей нижней границы приведены в руководстве Trusted Oracle Server Administrator's Guide.

## MAX

### Синтаксис

MAX([DISTINCT | ALL] *столбец*)

**Назначение** Возвращает максимальное значение для пункта списка выбора. Заметьте, что квалификаторы DISTINCT и ALL не оказывают на функцию никакого влияния, поскольку в любом случае максимальное значение будет одним и тем же.

**Область применения** Только списки выбора запросов и конструкции GROUP BY.

### Пример

В этом примере возвращается размер самого длинного имени в таблице students.

```
SELECT  MAX(LENGTH(first_name))
   FROM  students;
MAX(LENGTH(FIRST_NAME))
-----------------------
                      8
```

## MIN

### Синтаксис

MIN([DISTINCT | ALL] *столбец*)

**Назначение** Возвращает минимальное значение для пункта списка выбора. Заметьте, что квалификаторы DISTINCT и ALL не оказывают на функцию никакого влияния, поскольку в любом случае минимальное значение будет одним и тем же.

**Область применения** Только списки выбора запросов и конструкции GROUP BY.

### Пример

```
SELECT  MIN(id)
   FROM  students;
   MIN(ID)
---------
   10000
```

## STDDEV

### Синтаксис

STDDEV([DISTINCT | ALL] *столбец*)

**Назначение** Возвращает стандартное среднеквадратичное отклонение для пункта списка выбора. Это значение определяется как квадратный корень из дисперсии.

**Область применения** Только списки выбора запросов и конструкции GROUP BY.

### Пример

```
SELECT  STDDEV(number_seats)
```

```
  FROM rooms;
STDDEV(NUMBER_SEATS)
--------------------
          422.19664
```

## SUM

### Синтаксис

SUM([DISTINCT | ALL] *столбец*)

**Назначение** Возвращает сумму значений для пункта списка выбора.

**Область применения** Только списки выбора запросов и конструкции GROUP BY.

### Пример

```
SELECT department dept, SUM(num_credits)
  FROM classes
  GROUP by department;
DEPT  SUM(NUM_CREDITS)
----  ----------------
CS                   4
ECN                  3
HIS                  4
```

## VARIANCE

### Синтаксис

VARIANCE([DISTINCT | ALL] *столбец*)

**Назначение** Возвращает статическую дисперсию для пункта списка выбора. Oracle вычисляет дисперсию по следующей формуле:

$$\frac{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left[\sum_{i=1}^{n} x_i\right]^2}{n-1}$$

В этой формуле $x_i$ является значением одной строки, а $n$ – общим числом элементов набора. Если $n = 1$, то дисперсия равна нулю.

**Область применения** Только списки выбора запросов и конструкции GROUP BY.

### Пример

```
SELECT VARIANCE(number_seats)
  FROM rooms;
VARIANCE(NUMBER_SEATS)
----------------------
                178250
```

# Другие функции

В этом разделе приведены оставшиеся функции, которые не входят ни в одну из рассмотренных категорий.

## BFILENAME

### Синтаксис

BFILENAME(*каталог, имя_файла*)

**Назначение** Возвращает локатор BFILE, соответствующий имени физического *файла* операционной системы. *Каталог* должен быть объектом DIRECTORY в словаре данных.

**Область применения** Процедурные и SQL-операторы.

### Пример

Более подробно об объектах LOB и о модуле DBMS_LOB, в том числе и об объектах BFILE, говорится в главе 21.

## DECODE

### Синтаксис

DECODE(*базовое_выражение, выражение_сравнения1, значение1,*
*выражение_сравнения2, значение2,*
...
*значение_по_умолчанию*)

**Назначение** Функция DECODE аналогична серии вложенных операторов IF-THEN-ELSE. *Базовое выражение* последовательно сравнивается с *выражением 1, выражением 2* и т.д. Если *базовое выражение* соответствует *i*-му пункту *сравнения*, возвращается *i-е значение*. Если *базовое выражение* не соответствует ни одному пункту, возвращается *значение по умолчанию*.

*Выражения сравнения* рассматриваются по очереди. Если найдено соответствие, оставшиеся пункты *сравнения* (если они есть) не рассматриваются. Если базовое выражение является NULL-значением, оно считается эквивалентным *выражению сравнения* типа NULL.

**Область применения** Только SQL-операторы.

### Пример

```
SELECT  DECODE('abc',  'a',  1,
                       'b',  2,
                       'abc',  3,
                       'd',  4,
                       -1)  "Decode  1"
  FROM  dual;
Decode   1
----------
         3

SELECT  DECODE(NULL,   'a',  1,
                       NULL,  2)  "Decode  2"
  FROM  dual;
Decode   2
-----------
         2
```

## DUMP

### Синтаксис

DUMP(*выражение* [, *формат_числа* [, *начальная_позиция*] [, *длина*]]])

**Назначение** Возвращает значение типа VARCHAR2, содержащее информацию о внутреннем представлении *выражения*. *Формат числа* указывает основание системы счисления для возвращаемых значений в соответствии с таблицей 5.7.

**ТАБЛИЦА 5.7.**

| Формат числа | Вид возвращаемого результата |
|---|---|
| 8 | Восьмеричное представление |
| 10 | Десятичное представление |
| 16 | Шестнадцатеричное представление |
| 17 | Одиночный символ |

Если *формат числа* не указан, результат возвращается в десятичном виде.

Если указаны *начальная позиция и длина*, возвращается столько байтов, сколько указано в длине, с той *позиции*, которая является начальной. По умолчанию возвращается полное представление выражения. Тип данных возвращается в виде числа, соответствующего внутреннему типу данных:

**ТАБЛИЦА 5.8.**

| Код | Тип данных |
|---|---|
| 1 | VARCHAR2 |
| 2 | NUMBER |
| 8 | LONG |
| 12 | DATE |
| 23 | RAW |
| 24 | LONG RAW |
| 69 | ROWID |
| 96 | CHAR |
| 106 | MLSLABEL |

**Область применения** Только SQL-операторы.

## Пример

```
SELECT first_name, DUMP(first_name) "Dump"
  FROM students;
FIRST_NAME          Dump
---------------     ------------------------------------------
Scott               Typ=1 Len=5: 83,99,111,116,116
Margaret            Typ=1 Len=8: 77,97,114,103,97,114,101,116
Joanne              Typ=1 Len=6: 74,111,97,110,110,101
Manish              Typ=1 Len=6: 77,97,110,105,115,104
Patrick             Typ=1 Len=7: 80,97,116,114,105,99,107
Timothy             Typ=1 Len=7: 84,105,109,111,116,104,121

SELECT first_name, DUMP(first_name, 17) "Dump"
  FROM students;
FIRST_NAME          Dump
----------------- ------------------------------------------
Scott               Typ=1 Len=5: S,c,o,t,t
Margaret            Typ=1 Len=8: M,a,r,g,a,r,e,t
Joanne              Typ=1 Len=6: J,o,a,n,n,e
Manish              Typ=1 Len=6: M,a,n,i,s,h
Patrick             Typ=1 Len=7: P,a,t,r,i,c,k
Timothy             Typ=1 Len=7: T,i,m,o,t,h,y

SELECT first_name, DUMP(first_name, 17, 2, 4) "Dump"
  FROM students;
```

```
FIRST_NAME          Dump
------------------  ------------------------------
Scott               Typ=1  Len=5:  c,o,t,t
Margaret            Typ=1  Len=8:  a,r,g,a
Joanne              Typ=1  Len=6:  o,a,n,n
Manish              Typ=1  Len=6:  a,n,i,s
Patrick             Typ=1  Len=7:  a,t,r,i
Timothy             Typ=1  Len=7:  i,m,o,t
```

# EMPTY_CLOB/EMPTY_BLOB

## Синтаксис

EMPTY_CLOB
EMPTY_BLOB

**Назначение** Возвращают пустые локаторы LOB. EMPTY_CLOB возвращает символьный локатор, а EMPTY_BLOB – двоичный локатор.

**Область применения** Процедурные и SQL-операторы.

## Пример

Более подробно об объектах LOB и о модуле DBMS_LOB рассказано в главе 21.

# GREATEST

## Синтаксис

GREATEST(*выражение1* [, *выражение2*]...)

**Назначение** Возвращает наибольшее *выражение*. Перед сравнением каждое выражение неявно преобразуется к типу *выражения 1*. Если *выражение 1* имеет символьный тип, то сравнение выполняется без дополнения пробелами, причем результат имеет тип VARCHAR2.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT GREATEST(10, '7', -1)
  FROM dual;
GREATEST(10,'7',-1)
--------------------
                10
```

# GREATEST_LB

## Синтаксис

GREATEST_LB(*метка1* [, *метка2*]...)

**Назначение** Возвращает наибольшую нижнюю границу для списка *меток*. Каждая метка должна иметь тип MLSLABEL, RAW MLSLABEL или представлять собой литерал в виде строки символов в кавычках. Эту функцию можно использовать только в Trusted Oracle.

Более подробная информация об этом, в том числе примеры и определение наибольшей нижней границы, приведены в руководстве Trusted Oracle Server Administrator's Guide.

# LEAST

## Синтаксис

LEAST(*выражение1* [, *выражение2*]...)

**Назначение** Возвращает наименьшее значение из списка выражений. Функция LEAST похожа на функцию GREATEST тем, что все выражения неявно преобразуются к типу данных первого выражения. Все операции сравнения символов выполняются методом сравнения без дополнения пробелами.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT LEAST('abcd', 'ABCD', 'a', 'xyz') "Least"
  FROM dual;
Least
-------
ABCD
```

## LEAST_UB

### Синтаксис

LEAST_UB(*метка1* [, *метка2*]...)

**Назначение** Подобно GREATEST_LB, функция LEAST_UB возвращает наименьшую нижнюю границу для списка *меток*. *Метки* должны иметь тип данных MLSLABEL или быть литералами в кавычках. Возвращаемое значение имеет тип RAW MLSLABEL. Более подробная информация об этом, в том числе примеры и определение наименьшей нижней границы, приведены в руководстве Trusted Oracle Server Administrator's Guide.

## NVL

### Синтаксис

NVL(*выражение1, выражение2*)

**Назначение** Возвращает *выражение 2*, если *выражение 1* является NULL-значением; в противном случае возвращает *выражение 1*. Если *выражение 1* не является строкой символов, то возвращаемое значение имеет тот же тип данных, что и *выражение 1*. Иначе возвращаемое Назначение имеет тип данных VARCHAR2. Эта функция позволяет обеспечить отсутствие NULL-значений в активном наборе запроса.

**Область применения** Процедурные и SQL-операторы.

### Пример

```
SELECT   NVL('non null value', 7) "First",
         NVL(NULL, 'null value') "Second"
  FROM dual;
First           Second
--------------- ----------------
non null value  null value
```

## UID

### Синтаксис

UID

**Назначение** Возвращает целое число, однозначно идентифицирующее текущего пользователя базы данных. Функция UID аргументов не имеет.

**Область применения** Процедурные и SQL-операторы.

### Пример

*В этом примере показан сеанс работы в SQL*Plus.*

```
SQL> connect scott/tiger
Connected.
```

```
SQL>  SELECT UID
  2     FROM dual;
      UID
----------
         8
SQL> connect system/manager
Connected.
SQL> SELECT UID
  2     FROM dual;
      UID
----------
         5
```

# USER

## Синтаксис

USER

**Назначение** Возвращает значение типа VARCHAR2, содержащее имя текущего пользователя Oracle. Функция USER аргументов не имеет.

**Область применения** Процедурные и SQL-операторы.

## Пример

В этом примере показан сеанс работы в SQL*Plus.

```
SQL> connect scott/tiger
Connected.
SQL> SELECT USER
  2     FROM dual;
USER
------------------------------
SCOTT
SQL> connect sys/change_on_install
Connected.
SQL> SELECT USER
  2     FROM dual;
USER
------------------------------
SYS
```

# USERENV

## Синтаксис

USERENV(*параметр*)

**Назначение** Возвращает значение типа VARCHAR2, содержащее сведения о текущем сеансе с учетом *параметра*. Поведение функции описано в таблице 5.9.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT USERENV('TERMINAL'), USERENV('LANGUAGE')
  FROM dual;
USERENV( USERENV('LANGUAGE')
-------  ----------------------------
Windows  AMERICAN_AMERICA.WE8ISO8859P1
```

**ТАБЛИЦА 5.9. Результаты, возвращаемые функцией USERENV при различных значениях параметра**

| Значение параметра | Поведение функции USERENV |
|---|---|
| 'OSDBA'[1] | Если в текущем сеансе разрешена роль OSDBA, то возвращается 'TRUE'; в противном случае возвращается 'FALSE'. Обратите внимание, что возвращаемое значение имеет тип VARCHAR2, а не BOOLEAN. |
| 'LABEL' | Применяется только в Trusted Oracle. Возвращает метку текущего сеанса. Более подробная информация приведена в руководстве Trusted Oracle Server Administrator's Guide. |
| 'LANGUAGE' | Возвращает язык и территорию, которые используются в этом сеансе, а также набор символов базы данных. Это параметр средства NLS. Возвращаемое значение имеет вид: *язык_территория.наборсимволов*. |
| 'TERMINAL' | Возвращает идентификатор терминала текущего соединения, зависящий от вида операционной системы. Для распределенных SQL-операторов возвращается идентификатор *локального соединения*. |
| 'SESSIONID' | Возвращает идентификатор соединения, способного выполнять аудит, если параметр инициализации AUDIT_TRAIL установлен в TRUE (истина). USERENV('SESSIONID') нельзя применять в распределенных SQL-операторах. |
| 'ENTRYID' | Возвращает идентификатор доступного входа, способного выполнять аудит, если параметр инициализации AUDIT_TRAIL установлен в TRUE (истина). USERENV('ENTRYID') нельзя применять в распределенных SQL-операторах. |
| 'LANG'[2] | Возвращает сокращенное название языка, принятое в ISO. Это укороченный вид USERENV('LANGUAGE'). |

[1]USERENV('OSDBA') применяется в PL/SQL версии 2.2 (Oracle 7.2) и выше.
[2]USERENV('LANG') применяется в PL/SQL версии 8.0 (Oracle 8.0) и выше.

# VSIZE

## Синтаксис

VSIZE(*значение*)

**Назначение** Возвращает количество байтов, содержащихся во внутреннем представлении *значения*. Эта информация выдается также функцией DUMP. Если *значение* – NULL-значение, результат также является NULL-значением.

**Область применения** Процедурные и SQL-операторы.

## Пример

```
SELECT last_name, VSIZE(last_name) "Size"
  FROM students;
LAST_NAME      Size
------------ ------
Smith             5
Mason             5
Junebug           7
Murgratroid      11
Poll              4
Taller            6
```

# PL/SQL в работе: печать чисел прописью

В этом примере для представления чисел прописью используются символьные функции и функции преобразования. В модуле DH_UTIL содержатся две общедоступные функции: SPELL и CHECK_PROTECT.

▼ ВНИМАНИЕ Модули и их использование обсуждаются в главе 8.

Функция SPELL возвращает строку символов, представляющую аргумент прописью. Применение этой функции демонстрируется с помощью сеанса работы в SQL*Plus, приведенного ниже (здесь также используется модуль DBMS_OUTPUT, описанный в главе 14).

```
SQL> set serveroutput on
SQL> BEGIN
  2      DBMS_OUTPUT.PUT_LINE('12340:  ' || DH_UTIL.SPELL(12340));
  3      DBMS_OUTPUT.PUT_LINE('987.123:  ' || DH_UTIL.SPELL(987.123));
  4      DBMS_OUTPUT.PUT_LINE('10000:  ' || DH_UTIL.SPELL(10000));
  5   END;
  6 /
12340:  Twelve Thousand Three Hundred Forty
987.123:  Nine Hundred Eighty-Seven and One Hundred
        Twenty-Three / Thousandths
10000:  Ten Thousand

PL/SQL procedure successfully completed.
```

Функция CHECK_PROTECT представляет прописью сумму в долларах, как это делается при печати чека. Например:

```
SQL> set serveroutput on
SQL> BEGIN
  2      DBMS_OUTPUT.PUT_LINE('12340:  ' ||
                              DH_UTIL.CHECK_PROTECT(12340));
  3      DBMS_OUTPUT.PUT_LINE('987.12:  ' ||
                              DH_UTIL.CHECK_PROTECT(987.123));
  4      DBMS_OUTPUT.PUT_LINE('10000:  ' ||
                              DH_UTIL.CHECK_PROTECT(10000));
  5   END;
  6 /
12340:  Twelve Thousand Three Hundred Forty Dollars and Zero Cents
987.12:  Nine Hundred Eighty-Seven Dollars and Twelve Cents
10000:  Ten Thousand Dollars and Zero Cents

PL/SQL procedure successfully completed.
```

Для создания модуля DH_UTIL в SQL*Plus можно выполнить следующий сценарий:

```
-- Этот пример содержится в файле spelcheck.sql.
REM ************************************************************
REM Дэвид Л. Хант (автор файла) предлагает этот и другие
REM файлы-сценарии только для обучения, чтобы проиллюстрировать
REM применение различных методов обработки данных. Ни автор, ни
REM корпорация Oracle не гарантируют, что этот сценарий будет
REM пригоден в каких-либо промышленных приложениях. Кроме того, нет
REM гарантии, что этот или любые другие аналогично
REM распространяемые сценарии лишены ошибок и что их можно
REM использовать в иных целях, кроме простой иллюстрации примеров.
REM
REM Если вы хотите высказать свои замечания и предложения по поводу
REM функций этих модулей, а также если вы встретились с
REM определенными трудностями, обращайтесь, пожалуйста, к автору по
REM электронной почте (адрес приведен ниже).
```

```
REM
REM [Сохраняйте, пожалуйста, вышеприведенный текст и встроенную
REM электронную документацию вместе с этим сценарием.]
REM ************************************************************
REM Об этом сценарии-файле:
REM
REM ИМЯ: SPELCHEK.SQL - текст, написанный на PL/SQL для создания
REM      модуля (DH_UTIL), который обеспечивает: 1) представление
REM      чисел в виде текстовых слов и 2) защиту банковских
REM      векселей и чеков при помощи представления в текстовом
REM      виде денежных сумм.
REM
REM АВТОР: Дэйв Хант, старший ответственный инструктор
REM        Oracle Eduction Services
REM        170 South Main Street, Suite 1150
REM        Salt Lake City, Utah, USA 84101
REM        dhunt@us.oracle.com
REM
REM ************************************************************
REM История создания:
REM
REM 24-APR-96:  исходный текст;
REM 03-MAR-96:  усовершенствован, чтобы обрабатывать 1) отрицательные
REM             числа и 2) нули;
REM 16-JAN-97:  усовершенствован, чтобы обрабатывать
REM             десятичные дроби;
REM 20-JAN-97:  усовершенствован, чтобы обрабатывать числа в диапазоне:
REM             от (-10 ** 100)+1 до (10 ** 100) -1 с точностью до 40
REM             цифр. Могут быть представлены прописью числа,
REM             в которых после десятичной точки до 40 цифр;
REM 25-JAN-97:  внесена дополнительная документация и комментарии.
REM ************************************************************
REM В этом модуле содержится две глобальных (GLOBAL) функции:
REM  1) DH_UTIL.SPELL: преобразует число в английские слова.
REM     [Примечание: в этой версии форма названия чисел
REM                  американская, а не английская:
REM
REM          число американская форма·        английская форма
REM ------------------------ ------------ --------------------
REM             1,000,000,000  Billion        Milliard
REM         1,000,000,000,000  Trillion       Billion
REM     1,000,000,000,000,000  Quadrillion    Thousand Billion
REM 1,000,000,000,000,000,000  Quintillion    Trillion
REM
REM     Чтобы воспользоваться английской формой, модифицируйте
REM     таблицу в конце тела этого модуля.]
REM
REM ************************************************************
REM     Использование функции 1: "DH_UTIL.SPELL(любое_число)"
REM     Пример SQL:
REM       SELECT last_name,
REM         salary, DH_UTIL.SPELL(salary) Worded
REM   FROM s_emp;"
```

```
REM
REM  LAST_NAME       SALARY  WORDED
REM  -------------   ------  ----------------------------------
REM  Velasquez       2500    Two Thousand Five Hundred
REM  Ngao            1450    One Thousand Four Hundred Fifty
REM  Nagayama        1400    One Thousand Four Hundred
REM
REM  ************************************************************
REM    Пример PL/SQL:
REM       BEGIN
REM          DBMS_OUTPUT.PUT_LINE
REM  (          dh_util.spell(-123456789.123456789));
REM       END;
REM       /
REM       Negative One Hundred Twenty-Three Million Four
REM       Hundred Fifty-Six Thousand Seven Hundred Eighty-Nine
REM       and One Hundred Twenty-Three Million Four
REM       Hundred Fifty-Six Thousand Seven Hundred
REM       Eighty-Nine / Billionths
REM
REM  ************************************************************
REM  ************************************************************
REM    2)  DH_UTIL.CHECK_PROTECT: преобразует число в пропись
REM          вида "доллары и центы".
REM       Использование функции 2: "DH_UTIL.CHECK_PROTECT(любое_число)"
REM  ************************************************************
REM    Пример SQL:
REM       select 'Pay to the order of: '||
REM          rpad(ltrim(first_name||' '||last_name||' '),22,'*')
REM          ||' '||lpad(rtrim(' '||ltrim(nvl(
REM          to_char(salary,'$99,999,990.00'),'Null Amount')
REM               )),16,'*')||chr(10)||
REM          rpad('** '||
REM          dh_util.check_protect(SALARY)||' ',56,'*') " "
REM       from s_emp
REM       where rownum <= 3;
REM
REM  Pay to the order of: Carmen Velasquez ***** ****** $2,500.00
REM  ** Two Thousand Five Hundred Dollars and Zero Cents *********
REM
REM  Pay to the order of: LaDoris Ngao ********* ****** $1,450.00
REM  ** One Thousand Four Hundred Fifty Dollars and Zero Cents ***
REM
REM  Pay to the order of: Midori Nagayama ****** ****** $1,400.00
REM  ** One Thousand Four Hundred Dollars and Zero Cents *********
REM
REM  ************************************************************
REM    Пример PL/SQL:
REM       begin
REM          dbms_output.put_line
REM             (dh_util.check_protect(123456789.56));
REM       end;
REM       /
```

```
REM    One Hundred Twenty-Three Million Four Hundred
REM    Fifty-Six Thousand Seven Hundred Eighty-Nine Dollars
REM  and Fifty-Six Cents
REM
REM  ************************************************************
REM  Описание модуля DH_UTIL
REM  ************************************************************
create or replace package dh_util is
   function spell (x in number) return varchar2;
   function check_protect (x in number) return varchar2;
   pragma restrict_references(spell,WNDS);
   pragma restrict_references(check_protect,WNDS);
end;
/

REM  ************************************************************
REM  Тело модуля DH_UTIL
REM  ************************************************************
create or replace package body dh_util is
   result varchar2(2000);
   working_integer number;
   working_decimal varchar2(100);
   working_dec_mag number;
   working_integer_spell varchar2(2000);
   working_decimal_spell varchar2(2000);
   working_fraction_spell varchar2(2000);
   Type   number_stencil is table of number
          index by binary_integer;
   Type   varchar2_stencil is table of varchar2(2000)
          index by binary_integer;
denom varchar2_stencil;
pad_factor number_stencil;
hold varchar2_stencil;
--  ************************************************************

--  Описание глобальной функции модуля: DH_UTIL.SPELL

--  ************************************************************
function spell (x in number) return varchar2 is
--  ************************************************************

--  Описание локальной функции: WORDING

--  ************************************************************
   function wording (x in number) return varchar2 is
   begin
       if x = 0 then
          return 'Zero';
       else
          return to_char(to_date(x,'j'),'Jsp'); -- число-в-слова
       end if;
   end wording;
--  ************************************************************
```

```
-- Описание локальной функции: INTEGER_TRANSLATION

-- ***************************************************************
   Function integer_translation (working_x in number)
            return varchar2 is
         x_char varchar2(128);
         Denoms_to_do number;
         Start_byte   number;
         Pointer      binary_integer;
         interim_spelling varchar2(2000);
      begin
         if working_x is null then
            return 'Null';
         elsif working_x = 0 then
            return 'Zero';
         end if;
      x_char := abs(working_x);
      pointer := 3-mod(length(x_char),3);
      x_char :=
   lpad(x_char,length(x_char)+pad_factor(pointer),'0');
      denoms_to_do := length(x_char)/3;
      result := null;
      for i in 1..denoms_to_do loop
         start_byte := ((i-1)*3)+1;
         interim_spelling := wording(substr(x_char,start_byte,3));
         pointer := (denoms_to_do+1)-i;
            if upper(interim_spelling) <> 'ZERO' then
               result := rtrim(ltrim(result||' '||interim_spelling||
                  ' '||denom(pointer)));
            end if;
            hold(i) := result;
         end loop;
         return result;
   end integer_translation;
-- ***************************************************************

-- Процедурный раздел глобальной функции SPELL

-- ***************************************************************
begin
   working_integer_spell := null;
   working_decimal_spell := null;
   working_fraction_spell := null;
   working_integer := trunc(x);
   if abs(x) > abs(working_integer) then
      working_decimal :=
         substr(rtrim(to_char(abs(x)-abs(working_integer),
         '.00000000000000000000000000000000000000000000'),
         '0'),3);
   else
      working_decimal := null;
      working_dec_mag := null;
   end if;
   working_integer_spell := integer_translation(working_integer);
```

```
   If  working_decimal  is  not  null  then
       working_dec_mag  :=  10  **  length(working_decimal);
       working_decimal_spell  :=
          '  and  '||integer_translation(working_decimal);
       working_fraction_spell  :=
          integer_translation(working_dec_mag)||'th';
       if  working_decimal  >  1  then
          working_fraction_spell  :=  working_fraction_spell||'s';
       end  if;
       If  upper(substr(working_fraction_spell,1,3))='ONE'  then
          working_fraction_spell  :=
substr(working_fraction_spell,5);
       end  if;
       working_fraction_spell  :=  '  /  '||working_fraction_spell;
   end  if;
   if  working_integer  =  0  and  working_decimal_spell  is  not  null  then
       result  :=  substr(working_decimal_spell,5)||
          working_fraction_spell;
   else
       result  :=  working_integer_spell||
          working_decimal_spell||working_fraction_spell;
   end  if;
   if  x  <  0  then
       result  :=  'Negative  '||result;
   end  if;
   result  :=  replace(result,'  ','  ');
   return  result;
end  spell;
--  ************************************************************

--  Конец глобальной функции:  SPELL

--  ************************************************************

--  ************************************************************

--  Описание глобальной функции:  CHECK_PROTECT

--  ************************************************************
function  check_protect  (x  in  number)  return  varchar2  is
       hold_dollar  number;
       hold_cents  number;
       function  check_for_single  (y  in  number,  currency  in  varchar2)
          return  varchar2  is
   begin
       if  y  =  1  then
          return  'One  '||currency;
       else
          return  spell(y)  ||'  '||currency||'s';
       end  if;
   end;
begin
   if  x  is  null  then
       return  'Non  Negotiable';
```

```
    end if;
    hold_dollar := trunc(x);
    hold_cents := (abs(x) - trunc(abs(x)))*100;
    return check_for_single(hold_dollar,'Dollar')||' and '||
    check_for_single(hold_cents,'Cent');
  end check_protect;
  -- ************************************************************

  -- Первоначальная однократная инициализация значений для модуля

  -- ************************************************************
  begin
    pad_factor(1) := 1;
    pad_factor(2) := 2;
    pad_factor(3) := 0;
    denom(1) := null;
    denom(2) := 'Thousand';
    denom(3) := 'Million';
    denom(4) := 'Billion';
    denom(5) := 'Trillion';
    denom(6) := 'Quadrillion';
    denom(7) := 'Quintillion';
    denom(8) := 'Sextillion';
    denom(9) := 'Septillion';
    denom(10) := 'Octillion';
    denom(11) := 'Nonillion';
    denom(12) := 'Decillion';
    denom(13) := 'Undecillion';
    denom(14) := 'Duodecillion';
    denom(15) := 'Tredecillion';
    denom(16) := 'Quattuordecillion';
    denom(17) := 'Quindecillion';
    denom(18) := 'Sexdecillion';
    denom(19) := 'Septendecillion';
    denom(20) := 'Octodecillion';
    denom(21) := 'Novemdecillion';
    denom(22) := 'Vigintillion';
    denom(23) := 'Unvigintillion';
    denom(24) := 'Duovigintillion';
    denom(25) := 'Trevigintillion';
    denom(26) := 'Quattuorvigintillion';
    denom(27) := 'Quinvigintillion';
    denom(28) := 'Sexvigintillion';
    denom(29) := 'Septenvigintillion';
    denom(30) := 'Octovigintillion';
    denom(31) := 'Novemvigintillion';
    denom(32) := 'Tregintillion';
    denom(33) := 'Untregintillion';
    denom(34) := 'Duotregintillion';
  end dh_util;
  -- ************************************************************
  -- Конец глобальной функции: SPELL
  -- ************************************************************
  /
```

# Итоги

В этой главе подробно рассматривались различные типы встроенных функций и было указано, когда следует применять каждый тип. В следующей главе обсуждаются курсоры, используемые для многострочных запросов.

# Глава 6

## Курсоры

В главах 4 и 5 уже говорилось о том, как использовать SQL-операторы в PL/SQL. Курсоры расширяют эти функциональные возможности и позволяют программам явным образом управлять процессом обработки SQL-операторов. В этой главе поясняется, как использовать курсоры для многострочных запросов и других SQL-операторов. Кроме того, рассматриваются курсорные переменные — одно из основных новых средств PL/SQL 2.2, усовершенствованное в версии 2.3.

# Определение курсора

Для обработки SQL-оператора Oracle выделяет область памяти, называемую *контекстной областью* (context area). Она содержит информацию, необходимую для завершения обработки, в том числе: число строк, обрабатываемых оператором, указатель на представление этого оператора после синтаксического анализа и *активный набор* (active set), т.е. набор строк, возвращаемых запросом.

*Курсор* (cursor) — это указатель на контекстную область, с помощью которого программа PL/SQL может управлять контекстной областью и ее состоянием во время обработки оператора. Ниже приведен блок PL/SQL, иллюстрирующий использование цикла выборки курсора. Здесь запрос возвращает несколько строк данных.

```
-- Этот пример содержится в файле curexamp.sql.
DECLARE
  /* Выходные переменные для хранения результатов запроса. */
  V_StudentID    students.id%TYPE;
  V_FirstName    students.first_name%TYPE;
  V_LastName     students.last_name%TYPE;

  /* Переменная привязки, используемая в запросе. */
  V_Major        students.major%TYPE := 'Computer Science';

  /* Создание курсора. */
  CURSOR c_Students IS
      SELECT id, first_name, last_name
        FROM students
        WHERE major = v_Major;
BEGIN
  /* Обозначим строки активного набора и подготовимся к дальнейшей
     обработке данных. */
  OPEN c_Students;
  LOOP
  /* Выберем каждую строку активного набора в переменные PL/SQL. */
     FETCH c_Students INTO v_StudentID, v_FirstName, v_LastName;
  /* Если строки, которые нужно выбрать, закончились, выйдем из
     цикла. */
     EXIT WHEN c_Students%NOTFOUND;
  END LOOP;

  /* Освободим ресурсы, используемые запросом. */
  CLOSE c_Students;
END;
```

В этом примере иллюстрируется использование *явного* (explicit) курсора. В нем имя курсора явно присвоено оператору SELECT при помощи оператора CURSOR..IS. Для всех других SQL-операторов применяются *неявные* (implicit) курсоры. Обработка явного курсора выполняется в четыре этапа, которые описаны в следующем разделе. Обработка неявного курсора осуществляется в PL/SQL автоматически.

# Обработка явных курсоров

Для обработки явного курсора в PL/SQL необходимо выполнить четыре шага:

1. Объявить курсор.

2. Открыть курсор для запроса.
3. Выбрать результаты в переменные PL/SQL.
4. Закрыть курсор.

Объявление курсора является единственным шагом, который выполняется в разделе объявлений блока; другие три шага являются частью выполняемого раздела или раздела исключительных ситуаций.

## Объявление курсора

При объявлении курсора ему назначается имя и ставится в соответствие некоторый оператор SELECT. Синтаксис объявления курсора таков:

CURSOR *имя_курсора* IS *оператор_SELECT*

где *имя_курсора* — это имя курсора, а *оператор_SELECT* — запрос, который будет обрабатываться. Что касается области действия и области видимости, курсоры отвечают стандартным правилам, установленным для идентификаторов PL/SQL (см. главу 2). Так как имя курсора является идентификатором PL/SQL, оно должно быть объявлено до того, как на него будет произведена ссылка. Можно использовать любые операторы SELECT, в том числе соединения (joins) и операторы, в состав которых входят конструкции UNION (объединение) и MINUS (минус).

▼ ВНИМАНИЕ

Конструкция INTO (в) не входит в состав оператора SELECT, а является частью оператора FETCH (выбрать, или считать).

При объявлении курсора можно ссылаться на переменные PL/SQL в условии WHERE, которые рассматриваются в качестве переменных присваивания, или привязки (см. главу 3). Для курсоров справедливы обычные правила по определению области действия, поэтому эти переменные должны быть видимы в точке объявления курсора. Например, ниже приведен правильный раздел объявлений:

```
DECLARE
   V_Department   classes.department%TYPE;
   V_Course       classes.course%TYPE;
   CURSOR c_Classes IS
       SELECT * FROM classes
          WHERE department = v_Department;
             AND course = v_Course;
```

Однако следующий раздел неверен, так как переменные **v_Department** и **v_Course** не объявлены до того, как на них производится ссылка:

```
DECLARE
   CURSOR c_Classes IS
       SELECT * FROM classes
          WHERE department = v_Department;
          AND course = v_Course;
   V_Department   classes.department%TYPE;
   V_Course       classes.course%TYPE;
```

Чтобы гарантировать, что все переменные, на которые происходят ссылки при объявлении курсора, объявлены до этих ссылок, рекомендуется создавать все курсоры в конце раздела объявлений. В данной книге соблюдается именно это соглашение. Единственным исключением является ситуация, когда в ссылке, например в атрибуте %ROWTYPE, указывается имя самого курсора. В этом случае курсор должен быть объявлен до того, как на него будет произведена ссылка.

## Открытие курсора

Синтаксис открытия курсора выглядит следующим образом:

OPEN *имя_курсора;*

где *имя_курсора* обозначает предварительно объявленный курсор. Когда курсор открывается, происходит следующее:

- Анализируются значения переменных привязки.
- На основе значений переменных привязки определяется активный набор.
- Указатель активного набора устанавливается на первую строку.

Переменные привязки анализируются только во время открытия курсора. Для примера рассмотрим следующий блок PL/SQL:

```
-- Этот пример содержится в файле binds.sql.
DECLARE
  V_RoomID       classes.room_id%TYPE;
  V_Building     rooms.building%TYPE;
  V_Department   classes.department%TYPE;
  V_Course       classes.course%TYPE;
  CURSOR c_Buildings IS
      SELECT building
        FROM rooms, classes
        WHERE rooms.room_id = classes.room_id
        and department = v_Department
        and course = v_Course;
BEGIN
  -- Перед открытием курсора присвоим переменным привязки
  -- определенные значения.
  v_Department := 'HIS';
  v_Course := 101;

  -- Откроем курсор.
  OPEN c_Buildings;

  -- Присвоим переменным привязки новые значения - это никак не
  -- влияет на программу, так как курсор уже открыт.
  v_Department := 'XXX';
  v_Course := -1;
END;
```

Когда открыт курсор **c_Building**, в переменных **v_Department** и **v_Course** содержится соответственно 'HIS' и 101. Эти значения используются в запросе. Хотя после открытия курсора значения переменных **v_Department** и **v_Course** изменились, это никак не влияет на активный набор запроса. Такой алгоритм называется согласованностью чтения (read-consistency). Он разработан для обеспечения целостности информации базы данных. Согласованность чтения более подробно обсуждается в разделе "Курсоры SELECT FOR UPDATE" ниже в этой главе. Чтобы просмотреть новые значения, нужно закрыть и повторно открыть курсор. Запрос будет видеть изменения, внесенные в базу данных до выполнения оператора OPEN. Если другой пользователь изменил данные, но еще не зафиксировал эти изменения, они видны не будут.

Активный набор, или набор строк, удовлетворяющих условию запроса, определяется во время открытия курсора. К примеру, в предыдущем запросе возвращается одна строка ('Building Seven'). Для таблицы (таблиц), на которую производится ссылка в конструкции FROM запроса, проверяется условие WHERE, и все строки, для которых условие истинно, добавляются к активному набору. Указатель на этот набор также создается во время открытия курсора и сообщает, какая строка должна быть считана курсором в следующий раз.

Можно открыть уже открытый курсор. Перед вторым открытием PL/SQL неявно выполняет оператор CLOSE. Кроме того, в любой момент может быть открыто несколько курсоров.

## Считывание строк из курсора

Частью оператора FETCH является список INTO. Оператор FETCH имеет две формы:

FETCH *имя_курсора* INTO *список_переменных;*

и

FETCH *имя_курсора* INTO *запись_PL/SQL;*

где *имя_курсора* обозначает предварительно объявленный и открытый курсор, *список_переменных* представляет собой список предварительно объявленных переменных PL/SQL, разделенных запятыми, а *запись_PL/SQL* – предварительно объявленная запись PL/SQL. В любом случае переменная (переменные) в конструкции INTO должна иметь тип, совместимый со списком выбора запроса. С учетом созданного курсора **c_Buildings** выполним следующий корректный оператор:

```
FETCH c_Buildings INTO v_Building;
```

В примере, приведенном ниже, иллюстрируются правильные и неправильные операторы FETCH.

```
-- Этот пример содержится в файле badfetch.sql.
DECLARE
  V_Department  classes.department%TYPE;
  V_Course      classes.course%TYPE;
  CURSOR c_AllClasses IS
      SELECT *
        FROM classes;
  v_ClassesRecord c_AllClasses%ROWTYPE;
BEGIN
  OPEN c_AllClasses;

  -- Это верный оператор FETCH, возвращающий первую строку в запись
  -- PL/SQL, которая соответствует списку выбора запроса.
  FETCH c_AllClasses INTO v_ClassesRecord;

  -- Этот оператор FETCH неверен, так как список выбора запроса
  -- возвращает все 7 столбцов таблицы classes, но считывание
  -- происходит только в 2 переменные.
  -- Это приведет к установлению флага ошибки "PLS-394: wrong number
  -- of values in the INTO list of a FETCH statement"
  ("неверное число значений в списке INTO оператора FETCH." – Прим.пер.).
  FETCH c_AllClasses INTO v_Department, v_Course;
END;
```

После каждого считывания указатель активного набора увеличивается и переходит к следующей строке. Таким образом, каждый оператор FETCH будет последовательно возвращать строки активного набора до тех пор, пока не будет возвращен весь набор.

Для определения момента считывания всего набора используется атрибут %NOTFOUND, описанный в разделе "Курсорные атрибуты". При выполнении последнего считывания выходным переменным не будут присваиваться новые значения, т.е. они будут содержать прежние значения.

## Закрытие курсора

Когда выбран весь активный набор, курсор следует закрыть. Это означает, что программа закончила работу с курсором и отведенные для него ресурсы могут быть освобождены. В состав этих ресурсов входит пространство для хранения активного набора, а также временное пространство, используемое для определения такого набора. Синтаксис закрытия курсора таков:

**CURSOR *имя_курсора;***

где *имя_курсора* обозначает ранее открытый курсор. После закрытия курсора считывать из него строки нельзя. Если попытаться это сделать, Oracle выдаст сообщение об ошибке:

```
ORA-1001: Invalid Cursor
```

(неверный курсор. – *Прим. пер.*)

*или*

```
ORA-1002: Fetch out of Sequence
```

(*непоследовательное считывание.* – *Прим. пер.*)

Попытка закрыть ранее закрытый курсор также приведет к возникновению ошибки ORA-1001.

## Курсорные атрибуты

В PL/SQL существуют четыре атрибута, которые могут быть применены к курсорам. Курсорные атрибуты добавляются к имени курсора в блоке PL/SQL, подобно атрибутам %TYPE и %ROWTYPE. Однако вместо того, чтобы возвращать тип, курсорные атрибуты возвращают значения, которые могут быть использованы в выражениях. Эти атрибуты – %FOUND, %NOTFOUND, %ISOPEN и %ROWCOUNT – описаны в последующих разделах, причем при описании каждого курсора в качестве примера использу-

**Рис. 6.1.**
*Фрагмент программы, в которой используется курсорный атрибут*

```
DECLARE
      -- Cursor declaration
           CURSOR c_Tempdata IS
         SELECT * from temp_table;
      -- Record to store the fetched data
      v_TempRecord c_TempData%ROWTYPE;
BEGIN
   -- location ①
   OPEN c_TempData;                          -- Open cursor
   -- location
   FETCH c_TempData INTO v_TempRecord  -- Fetch first row
-- location ③
   FETCH c_TempData INTO v_TempRecord  -- Fetch second row
-- location ④
   FETCH c_TempData INTO v_TempRecord  -- Third fetch
-- location ⑤
   CLOSE c_TempData
-- location ⑥
END
```

ется листинг программы, приведенный на рис. 6.1. В этом примере предполагается, что в таблице **temp_table** содержатся две строки со следующими данными:

| **num_col** | **char_col** |
|---|---|
| 10 | 'Hello' |
| 20 | 'There' |

**%FOUND** Это логический атрибут. Он возвращает TRUE, если при предшествующем считывании была выбрана строка, и FALSE – если строка выбрана не была. При проверке %FOUND, когда курсор не открыт, выдается ошибка ORA-1001 (неверный курсор). Для иллюстрации поведения атрибута %FOUND приведена таблица 6.1, в которой цифры соответствуют точкам, помеченным на рис. 6.1.

**ТАБЛИЦА 6.1.**

| Точка | Значение c_TempData%FOUND | Пояснение |
|---|---|---|
| 1 | Ошибка ORA-1001 | Курсор **c_TempData** еще не открыт, и активного набора для него не существует. |
| 2 | NULL | Хотя курсор **c_TempData** открыт, не было произведено ни одного считывания строк. Значение атрибута не может быть определено. |
| 3 | TRUE | С помощью предшествующего оператора FETCH выбрана первая строка таблицы **temp_table.** |
| 4 | TRUE | С помощью предшествующего оператора FETCH выбрана вторая строка таблицы **temp_table.** |
| 5 | FALSE | Предшествующий оператор FETCH не вернул никаких данных, так как все строки активного набора выбраны. |
| 6 | Ошибка ORA-1001 | Курсор **c_TempData** закрыт, и вся хранившаяся информация об активном наборе удалена. |

**%NOTFOUND** ведет себя противоположно %FOUND: если предшествующее считывание возвращает строку – значение %NOTFOUND ложно. Атрибут %NOTFOUND возвращает TRUE, *только если во* время предшествующего считывания строка выбрана не была. Этот атрибут часто используется в качестве условия выхода из цикла выборки. В таблице 6.2 описано поведение %NOTFOUND в примере, изображенном на рис. 6.1.

**ТАБЛИЦА 6.2.**

| Точка | Значение c_TempData%NOTFOUND | Пояснение |
|---|---|---|
| 1 | Ошибка ORA- 1001 | Курсор **c_TempData** еще не открыт, и активного набора для него не существует. |
| 2 | NULL | Хотя курсор **c_TempData** открыт, не было произведено ни одного считывания строк. Значение атрибута не может быть определено. |
| 3 | FALSE | С помощью предшествующего оператора FETCH выбрана первая строка таблицы **temp_table.** |
| 4 | FALSE | С помощью предшествующего оператора FETCH выбрана вторая строка таблицы **temp_table**. |
| 5 | TRUE | Предшествующий оператор FETCH не вернул никаких данных, так как все строки активного набора выбраны. |
| 6 | Ошибка ORA-1001 | Курсор **c_TempData** закрыт, и вся хранившаяся информация об активном наборе удалена. |

**%ISOPEN** Этот логический атрибут используется для определения, открыт или нет соответствующий курсор. Если открыт, то %ISOPEN возвращает TRUE, а если не открыт – FALSE. Это иллюстрируется в таблице 6.3.

**ТАБЛИЦА 6.3.**

| Точка | Значение c_TempData%ISOPEN | Пояснение |
|---|---|---|
| 1 | FALSE | Курсор **c_TempData** еще не открыт. |
| 2 | TRUE | Курсор **c_TempData** был открыт. |
| 3 | TRUE | Курсор **c_TempData** еще открыт. |
| 4 | TRUE | Курсор **c_TempData** еще открыт. |
| 5 | TRUE | Курсор **c_TempData** еще открыт. |
| 6 | FALSE | Курсор **c_TempData** закрыт |

**%ROWCOUNT** Этот числовой атрибут возвращает число строк, считанных курсором на данный момент. Если он указывается, когда соответствующий курсор еще не открыт, возвращается ошибка ORA-1001. Поведение атрибута %ROWCOUNT описано в таблице 6.4.

**ТАБЛИЦА 6.4.**

| Точка | Значение c_TempData%ROWCOUNT | Пояснение |
|---|---|---|
| 1 | Ошибка ORA-1001 | Курсор **c_TempData** еще не открыт, и активного набора для него не существует. |
| 2 | 0 | Курсор **c_TempData** открыт, но не было произведено ни одного считывания строк. |
| 3 | 1 | Считана первая строка таблицы **temp_table.** |
| 4 | 2 | Считана вторая строка таблицы **temp_table**. |
| 5 | 2 | К данному моменту считаны две строки таблицы **temp_table.** |
| 6 | Ошибка ORA-1001 | Курсор **c_TempData** закрыт, и вся хранившаяся информация об активном наборе удалена. |

**Сравнение курсорных атрибутов** Для сравнения в таблице 6.5 приведены значения всех четырех курсорных атрибутов при выполнении рассмотренного блока.

**ТАБЛИЦА 6.5. Значения, возвращаемые курсорными атрибутами**

| Точка | c_TempData %FOUND | c_TempData %NOTFOUND | c_TempData %ISOPEN | c_TempData %ROWCOUNT |
|---|---|---|---|---|
| 1 | ORA-1001 | ORA-1001 | FALSE | ORA-1001 |
| 2 | NULL | NULL | TRUE | 0 |
| 3 | TRUE | FALSE | TRUE | 1 |
| 4 | TRUE | FALSE | TRUE | 2 |
| 5 | FALSE | TRUE | TRUE | 2 |
| 6 | ORA-1001 | ORA-1001 | FALSE | ORA-1001 |

## Параметризованные курсоры

Существует еще один способ использования переменных привязки в курсоре. *Параметризованные* (parameterized) курсоры, подобно процедурам, принимают определенные аргументы (о процедурах более подробно см. в главе 7). Рассмотрим курсор **c_Classes**, который обсуждался выше:

```
DECLARE
   V_Department   classes.department%TYPE;
   V_Course       classes.course%TYPE;
   CURSOR c_Classes IS
      SELECT *
      FROM classes
        WHERE department = v_Department
        AND course = v_Course;
```

В **c_Classes** содержится две переменных — **v_Department** и **v_Course**. Можно превратить курсор **c_Classes** в эквивалентный параметризованный курсор:

```
DECLARE
   CURSOR c_Classes (p_Department classes.department%TYPE,
                     p_Course classes.course%TYPE) IS
   SELECT *
      FROM classes
      WHERE department = p_Department
      AND course = p_Course;
```

При работе с параметризованным курсором фактические значения указываются в операторе OPEN, например:

```
OPEN c_Classes('HIS', 101);
```

В этом случае 'HIS' передается как значение для p_Department, а 101 — для p_Course. Кроме того, параметры можно передавать при помощи позиционного или именного представления. Полная информация о передаче параметров приведена в главе 7.

# Обработка неявных курсоров

Как было показано в предыдущем разделе, явные курсоры используются для обработки тех операторов SELECT, которые возвращают более одной строки. Однако каждый оператор SELECT выполняется в пределах контекстной области и поэтому имеет курсор, указывающий на конкретную контекстную область. Такой курсор называется *SQL-курсором*. В отличие от явных курсоров SQL-курсор не открывается и не закрывается программой. PL/SQL неявно открывает SQL-курсор, обрабатывает SQL-оператор и впоследствии закрывает этот курсор.

Неявные курсоры используются для обработки операторов INSERT, UPDATE, DELETE, а также однострочных операторов SELECT..INTO. SQL-курсор открывается и закрывается PL/SQL, поэтому команды OPEN, FETCH и CLOSE не нужны. Однако для SQL-курсоров можно применять курсорные атрибуты. Например, ниже приведен блок, в котором оператор INSERT выполняется, хотя оператор UPDATE не соответствует ни одной из строк.

```
-- Этот пример содержится в файле nomatch1.sql.
BEGIN
  UPDATE rooms
      SET number_seats = 100
      WHERE room_id = 99980;
  -- Если предыдущий оператор UPDATE не выбирает ни одной
  -- строки, то введем новую строку в таблицу rooms.
  IF SQL%NOTFOUND THEN
      INSERT INTO rooms (room_id, number_seats)
        VALUES (99980, 100);
  END IF;
END;
```

Ту же задачу можно выполнить при помощи атрибута SQL%ROWCOUNT:

```
-- Этот пример содержится в файле nomatch2.sql.
BEGIN
  UPDATE rooms
      SET number_seats = 100
      WHERE room_id = 99980;
  -- Если предыдущий оператор UPDATE не выбирает ни одной
  -- строки, введем новую строку в таблицу rooms.
  IF SQL%ROWCOUNT = 0 THEN
      INSERT INTO rooms (room_id, number_seats)
        VALUES (99980, 100);
  END IF;
END;
```

Хотя атрибут SQL%NOTFOUND можно использовать в операторах SELECT..INTO, он мало полезен для этого. Дело в том, что, когда оператор SELECT..INTO не выбирает ни одной строки, Oracle выдает сообщение об ошибке:

```
ORA-1403: no data found
```

(данные не найдены. — *Прим. пер.*)

В результате управление блоком сразу же передается разделу обработки исключительных ситуаций и атрибут SQL%NOTFOUND не проверяется. Проиллюстрируем вышесказанное на примере:

```
-- Этот пример содержится в файле nodata.sql.
DECLARE
  -- Запись для хранения информации об аудиториях.
  v_RoomData rooms%ROWTYPE;
BEGIN
  -- Считаем информацию об аудитории с идентификатором -1.
  SELECT *
      INTO v_RoomData
      FROM rooms
      WHERE room_id = -1;

  -- Следующий оператор не может быть выполнен, так как управление
  -- программой сразу же передается обработчику исключительной ситуации.
  IF SQL%NOTFOUND THEN
      INSERT INTO temp_table (char_col)
```

```
        VALUES ('Not found!');
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        INSERT INTO temp_table (char_col)
            VALUES ('Not found, exception handler');
END;
```

Обработка исключительных ситуаций детально обсуждается в главе 10. Заметим, что SQL%NOTFOUND можно проверить внутри обработчика исключительной ситуации NO_DATA_FOUND, но здесь это условие всегда будет истинным.

Можно воспользоваться также атрибутом SQL%ISOPEN, однако он всегда будет возвращать FALSE, поскольку неявный курсор автоматически закрывается после того, как в нем обработан оператор.

# Циклы выборки

Чаще всего курсоры используются для считывания всех строк активного набора с помощью циклов выборки. *Цикл выборки* (fetch loop) — это обычный цикл, в котором строки активного набора обрабатываются по порядку, одна за другой. В следующих разделах рассматривается несколько различных типов циклов выборки курсоров и их применение.

## Простые циклы

В этих циклах для обработки курсора используется синтаксис простого цикла (LOOP..END LOOP). Количество выполняемых циклов задается с помощью атрибутов явного курсора. Рассмотрим пример:

```
-- Этот пример содержится в файле simple.sql..
DECLARE
  -- Объявим переменные для хранения информации о студентах,
  -- специализирующихся по истории.
  v_StudentID students.id%TYPE;
  v_FirstName students.first_name%TYPE;
  v_LastName students.last_name%TYPE;

  -- Курсор для считывания информации о студентах-историках.
  CURSOR c_HistoryStudents IS
      SELECT id, first_name, last_name
        FROM students
        WHERE major = 'History';
BEGIN
  -- Откроем курсор и инициализируем активный набор.
  OPEN c_HistoryStudents;
  LOOP
      -- Считаем информацию о следующем студенте.
      FETCH c_HistoryStudents INTO v_StudentID, v_FirstName,
                                   v_LastName;
  -- Выйдем из цикла, когда строк для выбора больше нет.
  EXIT WHEN c_HistoryStudents%NOTFOUND;

  -- Обработаем считанные строки. В нашем случае запишем всех
  -- студентов в группу History 301, введя информацию о них в
  -- таблицу registered_students. Кроме того, запишем имена и
  -- фамилии в таблицу temp_table.
  INSERT INTO registered_students (student_id, department,
                                   course)
      VALUES (v_StudentID, 'HIS', 301);
```

```
      INSERT INTO temp_table (num_col, char_col)
        VALUES (v_StudentID, v_FirstName || ' ' || v_LastName);

  END LOOP;

  -- Освободим ресурсы, используемые курсором.
  CLOSE c_HistoryStudents;

  -- Завершим работу.
  COMMIT;
END;
```

Обратите внимание, что оператор EXIT WHEN расположен сразу после оператора FETCH. После того как считана последняя строка, **c_HistoryStudents%NOTFOUND** становится истинным и происходит выход из цикла. Кроме того, оператор EXIT WHEN находится перед фрагментом обработки данных. Это делается для того, чтобы не обрабатывались повторяющиеся строки.

Рассмотрим блок, который очень похож на предыдущий, за исключением того, что оператор EXIT WHEN перенесен на конец цикла:

```
-- Этот пример содержится в файле exitwhen.sql.
DECLARE
  -- Объявим переменные для хранения информации о студентах,
  -- специализирующихся по истории.
  V_StudentID    students.id%TYPE;
  V_FirstName    students.first_name%TYPE;
  V_LastName     students.last_name%TYPE;

  -- Курсор для считывания информации о студентах-историках.
  CURSOR c_HistoryStudents IS
    SELECT id, first_name, last_name
      FROM students
      WHERE major = 'History';
  BEGIN
    -- Откроем курсор и инициализируем активный набор.
    OPEN c_HistoryStudents;
    LOOP
      -- Считаем информацию о следующем студенте.
      FETCH c_HistoryStudents INTO v_StudentID, v_FirstName,
                                   V_LastName;

    -- Обработаем считанные строки. В нашем случае запишем всех
    -- студентов в группу History 301, введя информацию о них в
    -- таблицу registered_students. Кроме того, запишем имена и
    -- фамилии в таблицу temp_table.
    INSERT INTO registered_students (student_id, department,
                                     course)
      VALUES (v_StudentID, 'HIS', 301);

    INSERT INTO temp_table (num_col, char_col)
      VALUES (v_StudentID, v_FirstName || ' ' || v_LastName);

    -- Выйдем из цикла, когда строк для выбора больше нет.
    EXIT WHEN c_HistoryStudents%NOTFOUND;

  END LOOP;

  -- Освободим ресурсы, используемые курсором.
```

```
  CLOSE c_HistoryStudents;

  -- Завершим работу.
  COMMIT;
END;
```

Теперь при последнем считывании переменные **v_StudentID, v_FirstName** и **v_LastName** не будут модифицированы, так как строк в активном наборе больше нет. Следовательно, в выходных переменных будут содержаться значения, соответствующие ранее считанным строкам. Из-за того что проверка выполняется после обработки данных, эти дублирующие значения будут введены в таблицы **registered_students** и **temp_table**, что приведет не к тому результату, который был нужен.

## Циклы WHILE

Циклы выборки курсоров можно построить также при помощи синтаксиса WHILE..LOOP. Рассмотрим пример:

```
-- Этот пример содержится в файле while.sql.
DECLARE
  -- Курсор для считывания информации о студентах-историках.
  CURSOR c_HistoryStudents IS
      SELECT id, first_name, last_name
        FROM students
        WHERE major = 'History';

-- Объявим запись для хранения считанной информации.
  v_StudentData c_HistoryStudents%ROWTYPE;
BEGIN
  -- Откроем курсор и инициализируем активный набор.
  OPEN c_HistoryStudents;

  -- Считаем первую строку, чтобы установить цикл WHILE.
  FETCH c_HistoryStudents INTO v_StudentData;

  -- Продолжим выполнение цикла, пока есть строки для считывания.
  WHILE c_HistoryStudents%FOUND LOOP
      -- Обработаем считанные строки. В нашем случае запишем всех
      -- студентов в группу History 301, введя информацию о них в
      -- таблицу registered_students. Кроме того, запишем имена и
      -- фамилии в таблицу temp_table.
      INSERT INTO registered_students (student_id, department,
                                       course)
        VALUES (v_StudentData.ID, 'HIS', 301);

      INSERT INTO temp_table (num_col, char_col)
        VALUES (v_StudentData.ID,
                v_StudentData.first_name || ' ' ||
                V_StudentData.last_name);

      -- Считаем следующую строку. Условие %FOUND будет проверяться
      -- перед тем, как цикл будет продолжен.
      FETCH c_HistoryStudents INTO v_StudentData;
  END LOOP;

  -- Освободим ресурсы, используемые курсором.
  CLOSE c_HistoryStudents;
```

```
    -- Завершим работу.
    COMMIT;
END;
```

Этот цикл выборки функционирует точно так же, как и цикл LOOP..END LOOP из примера, рассмотренного в предыдущем разделе. Обратите внимание, что оператор FETCH появляется дважды – один раз перед циклом и один раз после обработки данных. Условие **c_HistoryStudents%FOUND** следует проверять для каждой итерации цикла.

## Курсорные циклы FOR

В обоих циклах выборки, описанных выше, необходимо обрабатывать курсоры явным образом с помощью операторов OPEN, FETCH и CLOSE. В PL/SQL имеется упрощенный вид цикла – курсорный цикл FOR, – в котором управление обработкой курсора осуществляется неявно. В качестве примера рассмотрим цикл, эквивалентный двум рассмотренным выше.

```
-- Этот пример содержится в файле forloop.sql.
DECLARE
  -- Курсор для считывания информации о студентах-историках.
  CURSOR c_HistoryStudents IS
      SELECT id, first_name, last_name
        FROM students
        WHERE major = 'History';
BEGIN
  -- Начнем цикл. Здесь происходит неявное открытие курсора
  -- c_HistoryStudents.
  FOR v_StudentData IN c_HistoryStudents LOOP
      -- Здесь происходит неявное считывание.

      -- Обработаем считанные строки. В нашем случае запишем всех
      -- студентов в группу History 301, введя информацию о них в
      -- таблицу registered_students. Кроме того, запишем имена и
      -- фамилии в таблицу temp_table.
      INSERT INTO registered_students (student_id, department,
                                       course)
        VALUES (v_StudentData.ID, 'HIS', 301);
      INSERT INTO temp_table (num_col, char_col)
        VALUES (v_StudentData.ID,
                v_StudentData.first_name || ' ' ||
                v_StudentData.last_name);
      -- Перед продолжением цикла здесь происходит проверка
      -- c_HistoryStudents%NOTFOUND.
  END LOOP;
  -- Теперь после окончания цикла выполняется неявное закрытие
  -- c_HistoryStudents.

  -- Завершим работу.
  COMMIT;
END;
```

Необходимо отметить два важных аспекта, касающихся этого примера. Во-первых, запись **v_StudentData** не создается в разделе объявлений блока. Эта переменная неявно объявляется компилятором PL/SQL, подобно индексу цикла в числовых циклах FOR. Она имеет тип **c_HistoryStudents%ROWTYPE**, а ее область действия – лишь собственно цикл FOR. Неявное объявление индекса цикла и область действия этого объявления те же, что и в числовом цикле FOR (см. главу 2). Поэтому *нельзя присвоить какое-либо значение переменной цикла внутри курсорного цикла* FOR.

Во-вторых, курсор **c_HistoryStudents** открывается, считывается и закрывается неявным образом (эти места отмечены комментариями). Он открывается перед началом цикла. Перед каждой итерацией цик-

ла атрибут %FOUND проверяется для установления наличия строк в активном наборе. Когда активный набор полностью выбран, курсор закрывается с окончанием цикла.

Курсорные циклы FOR хороши тем, что, обеспечивая функциональные возможности циклов выборки курсоров, они делают процесс считывания данных доступнее и понятнее, упрощая при этом тексты программ.

## NO_DATA_FOUND и %NOTFOUND

Исключительная ситуация NO_DATA_FOUND (данные не найдены) устанавливается только для операторов SELECT..INTO, когда условию запроса не соответствует ни одна из строк. В этом случае атрибут %NOTFOUND принимает значение TRUE. Если нет соответствия строкам в условии оператора UPDATE или DELETE, SQL%NOTFOUND также принимает значение TRUE. Поэтому во всех циклах выборки, рассмотренных выше, для определения условия выхода из цикла используется не исключительная ситуация NO_DATA_FOUND, а атрибуты %NOTFOUND или %FOUND.

## Курсоры SELECT FOR UPDATE

Очень часто при обработке данных в цикле выборки модифицируются строки, которые были считаны курсором. В PL/SQL для этого предлагается удобное средство, состоящее из двух частей: конструкции FOR UPDATE (для обновления) в объявлении курсора и конструкции WHERE CURRENT OF (где текущая строка ...) в операторе UPDATE или DELETE.

### FOR UPDATE

Конструкция FOR UPDATE является частью оператора SELECT. Ее разрешается использовать в качестве последней конструкции оператора, после ORDER BY (при ее наличии):

SELECT ... FROM ... FOR UPDATE [OF *ссылка_на_столбец*] [NOWAIT]

где *ссылка_на_столбец* – это столбец таблицы, для которой выполняется запрос. Можно также воспользоваться списком столбцов. Например, ниже приведен раздел объявлений, в котором описано два курсора, являющихся правильными формами курсора SELECT..FOR UPDATE.

```
DECLARE
   -- В этом курсоре в конструкции FOR UPDATE указано два столбца.
   CURSOR c_AllStudentd IS
      SELECT *
         FROM students
         FOR UPDATE OF first_name, last_name;

   -- В этом курсоре столбцы не указаны.
   CURSOR c_LargeClasses IS
      SELECT department, course
         FROM classes
         WHERE max_students > 50
         FOR UPDATE;
```

Обычно при выполнении операции SELECT строки, к которым происходит обращение, не блокируются. Это дает возможность другим сеансам, соединенным с базой данных, изменять выбираемые данные. Тем не менее результирующий набор данных остается согласованным. Во время открытия курсора, когда определяется активный набор, Oracle извлекает моментальный снимок (snapshot) таблицы. Все изменения, зафиксированные к этому моменту, отражаются на активном наборе. Все изменения, внесенные после этого момента, не отражаются, даже когда они зафиксированы. Они будут отражены, если открыть курсор вновь, что приведет к повторному определению активного набора. Это и есть алгоритм согласованного чтения данных, упомянутый в начале главы. Однако при использовании конструкции FOR UPDATE до открытия курсора устанавливаются исключающие строчные блокировки строк активного набора. Эти блокировки предотвращают изменение строк активного набора другими сеансами до завершения транзакции.

Если другой сеанс уже блокировал строки активного набора, операция SELECT FOR UPDATE будет ждать снятия этих блокировок неограниченное время до тех пор, пока другой сеанс не снимет блокировку. Для разрешения этой ситуации можно воспользоваться параметром NOWAIT (без ожидания). Если строки блокированы другим сеансом, оператор OPEN сразу же возвратит сообщение об ошибке:

```
ORA-54: resource busy and acquire with NOWAIT specified
```

(ресурс занят и запрошен с указанием NOWAIT. – *Прим. пер.*)

В этом случае можно попытаться открыть курсор позже или изменить активный набор так, чтобы считывать неблокированные строки.

## WHERE CURRENT OF

Если курсор объявлен с конструкцией FOR UPDATE, в операторе UPDATE или SELECT может быть указана конструкция WHERE CURRENT OF. Синтаксис конструкции WHERE CURRENT OF таков:

WHERE CURRENT OF *курсор*

где *курсор* — это имя курсора, объявленного с конструкцией FOR UPDATE. Конструкция WHERE CURRENT OF определяет строку, только что считанную курсором. Например, ниже приведен блок, с помощью которого будут обновляться текущие зачеты всех студентов, зарегистрированных в HIS 101.

```
-- Этот пример содержится в файле forupdat.sql.
DECLARE
  -- Число зачетов, которое нужно добавить к общему числу зачетов
  -- каждого студента.
  v_NumCredits classes.num_credits%TYPE;

  -- Этот курсор будет выбирать только тех студентов, которые
  -- в настоящий момент зарегистрированы в HIS 101.
  CURSOR c_RegisteredStudents IS
      SELECT *
        FROM students
        WHERE id IN (SELECT student_id
          FROM registered_students
          WHERE department= 'HIS'
          AND course = 101)
      FOR UPDATE OF current_credits;

BEGIN
  -- Установим цикл выборки для этого курсора.
  FOR v_StudentInfo IN c_RegisteredStudents LOOP
  -- Определим число зачетов для HIS 101.
  SELECT num_credits
      INTO v_NumCredits
      FROM classes
      WHERE department = 'HIS'
      AND course = 101;

  -- Обновим строку, только что считанную из курсора.
  UPDATE students
      SET current_credits = current_credits + v_NumCredits
      WHERE CURRENT OF c_RegisteredStudents;
  END LOOP;

  -- Завершим работу.
  COMMIT;
END;
```

Заметьте, что оператор UPDATE обновляет только тот столбец, который указан в конструкции FOR UPDATE при объявлении курсора. Если не указан ни один столбец, можно обновлять любые столбцы.

Разрешается выполнять запрос с конструкцией FOR UPDATE, но при этом не ссылаться на строки, выбранные посредством WHERE CURRENT OF. В этом случае строки остаются блокированными и поэтому могут быть модифицированы только текущим сеансом (удерживающим блокировку). Операторы UPDATE и DELETE, изменяющие эти строки, не будут блокировать их, если выполняются сеансом, удерживающим блокировку.

## Размещение оператора COMMIT при считывании строк

Обратите внимание, что в предыдущем разделе оператор COMMIT выполняется после окончания цикла выборки. Дело в том, что этот оператор снимает все блокировки, удерживаемые сеансом. Конструкция FOR UPDATE устанавливает блокировки, а оператор COMMIT будет их снимать. Когда это происходит, действие курсора прекращается и любая последующая попытка считать строки приводит к ошибке Oracle:

```
ORA-1002: fetch out of sequence
```

(непоследовательное считывание. — *Прим. пер.*)

Рассмотрим пример, в результате выполнения которого возвращается эта ошибка.

```
-- Этот пример содержится в файле commit1.sql.
DECLARE
  -- Курсор для считывания информации обо всех студентах, а также
  -- для блокирования строк.
  CURSOR c_AllStudents IS
      SELECT *
        FROM students
        FOR UPDATE;

  -- Переменная для выбранных данных.
  v_StudentInfo c_AllStudents%ROWTYPE;
BEGIN
  -- Откроем курсор. При этом будут установлены блокировки.
  OPEN c_AllStudents;

  -- Считаем первую запись.
  FETCH c_AllStudents INTO v_StudentInfo;

  -- Выполним оператор COMMIT. При этом блокировки будут сняты
  -- и действие курсора прекращено.
  COMMIT WORK;

  -- Этот оператор FETCH приведет к ошибке ORA-1002.
  FETCH c_AllStudents INTO v_StudentInfo;
END;
```

Таким образом, если оператор COMMIT расположен внутри цикла выборки SELECT FOR UPDATE, считывание строк после COMMIT будет завершаться неуспешно. Поэтому не рекомендуется размещать COMMIT внутри цикла. Если курсор не определен как SELECT FOR UPDATE, никаких проблем не возникает.

Как поступить, если требуется обновить строку, только что считанную из курсора, и при этом применить оператор COMMIT внутри цикла? Конструкцией WHERE CURRENT OF воспользоваться нельзя, так как курсор не может быть определен с конструкцией FOR UPDATE. В этом случае можно в условии WHERE оператора UPDATE использовать первичный ключ таблицы:

```
-- Этот пример содержится в файле commit2.sql.
DECLARE
  -- Число зачетов, которое нужно добавить к общему числу зачетов
  -- каждого студента.
  v_NumCredits classes.num_credits%TYPE;

  -- Этот курсор будет выбирать только тех студентов, которые
  -- в настоящий момент зарегистрированы в HIS 101.
  CURSOR c_RegisteredStudents IS
      SELECT *
        FROM students
        WHERE id IN (SELECT student_id
```

```
                        FROM registered_students
                        WHERE department= 'HIS'
                        AND course = 101);

BEGIN
   -- Установим цикл выборки для этого курсора.
   FOR v_StudentInfo IN c_RegisteredStudents LOOP
   -- Определим число зачетов для HIS 101.
   SELECT num_credits
       INTO v_NumCredits
       FROM classes
       WHERE department = 'HIS'
       AND course = 101;

   -- Обновим строку, только что считанную из курсора.
   UPDATE students
       SET current_credits = current_credits + v_NumCredits
       WHERE id = v_StudentInfo.id;

   -- Можно завершить работу внутри цикла, так как курсор не
   -- объявлен как FOR UPDATE.
   COMMIT;
   END LOOP;
END;
```

В этом примере, по существу, имитируется конструкция WHERE CURRENT OF, однако при этом строки активного набора не блокируются. В результате этот блок может выполняться не так, как ожидалось, если другие сеансы будут параллельно обращаться к данным.

# Курсорные переменные

Во всех приведенных выше примерах явных курсоров рассматривались *статические курсоры* (static cursors), связанные с одним SQL-оператором, который был известен при компиляции блока. *Курсорная переменная* (cursor variable) во время выполнения программы может быть связана с различными операторами. Курсорные переменные аналогичны переменным PL/SQL, в которых могут содержаться различные значения. Статические же курсоры аналогичны константам PL/SQL, так как они могут быть связаны только с одним запросом этапа выполнения программы.

PL/SQL 2.2 ... и ВЫШЕ

В версиях PL/SQL, предшествовавших версии 2.2, курсорные переменные не применялись. Они впервые представлены в версии 2.2, а в версии 2.3 их функциональные возможности были расширены. В этом разделе описаны свойства курсорных переменных, доступные в обеих версиях. Во всех примерах используется версия 2.2 и, где указано специально, версия 2.3 и выше.

Чтобы воспользоваться курсорной переменной, ее необходимо предварительно объявить. После этого во время выполнения программы нужно выделить место для хранения этой переменной, так как курсорные переменные имеют тип REF. Затем ее можно открывать, считывать и закрывать так же, как и статический курсор.

## Объявление курсорной переменной

Курсорные переменные имеют ссылочный тип. В версиях PL/SQL 2.2 и 2.3 применяется только один ссылочный тип. Как было рассказано в главе 2, ссылочный тип — это то же самое, что и указатель в языке программирования C или Pascal. С помощью такого типа можно именовать области хранения данных во время выполнения программы. Чтобы воспользоваться ссылочным типом, необходимо сначала объявить переменную, а затем выделить область для ее хранения. Ссылочные типы PL/SQL объявляются следующим образом:

REF *тип*

где *тип* – это предварительно определенный тип. Ключевое слово REF означает, что новый тип будет указателем на ранее определенный тип. Таким образом, типом курсорной переменной является тип REF CURSOR. Это единственный разрешенный тип в версиях PL/SQL, предшествующих версии 2.3 (в PL/SQL 8.0 можно ссылаться на объекты, описанные в главе 11). Полный синтаксис описания типа курсорной переменной таков:

TYPE *имя_типа* IS REF CURSOR RETURN *возвращаемый_тип*;

где *имя_типа* – это имя нового ссылочного типа, а *возвращаемый_тип* – тип записи, указывающий типы списка выбора, которые в итоге будут возвращаться курсорной переменной.

Типом, возвращаемым курсорной переменной, должен быть тип записи. Запись может быть объявлена явно как запись, определяемая пользователем, или неявно, при помощи %ROWTYPE. После того как ссылочный тип описан, можно объявлять переменную. В разделе объявлений, приведенном ниже, показано, как можно описывать различные курсорные переменные.

```
DECLARE
  -- Описание при помощи %ROWTYPE.
  TYPE t_StudentsRef IS REF CURSOR
      RETURN students%ROWTYPE;

  -- Опишем новый тип записи,
  TYPE t_NameRecord IS RECORD (
      First_name    students.first_name%TYPE,
      Last_name     students.last_name%TYPE);

  -- переменную этого типа
  v_NameRecord t_NameRecord;

  -- и тип курсорной переменной, использующей этот тип записи.
  TYPE t_NamesRef IS REF CURSOR
      RETURN t_NameRecord;

  -- При помощи %TYPE для записи, описанной выше, можно объявить
  -- еще один тип.
  TYPE t_NamesRef2 IS REF CURSOR
      RETURN t_NameRecord%TYPE;

  -- Объявим курсорные переменные, использующие типы, созданные выше.
  v_StudentCV t_StudentsRef;
  v_NameCV t_NamesRef;
```

## Ограниченные и неограниченные курсорные переменные

Курсорные переменные, рассмотренные в предыдущем разделе, являются *ограниченными* (constrained): *они объявляются только для конкретного возвращаемого типа. Переменная должна открываться для* такого запроса, список выбора которого соответствует типу, возвращаемому курсором. В противном случае возникает предварительно определенная исключительная ситуация ROWTYPE_MISMATCH (*несоответствие типов строк*).

**PL/SQL 2.3 ... и ВЫШЕ**

Однако в PL/SQL 2.3 разрешается объявлять неограниченные (unconstrained) курсорные переменные, для которых оборот RETURN отсутствует. Когда такая переменная впоследствии открывается, она может быть открыта для любого запроса. Ниже приведен раздел объявлений, в котором описывается неограниченная курсорная переменная.

```
DECLARE
  -- Опишем неограниченный ссылочный тип
  TYPE t_FlexibleRef IS REF CURSOR;

  -- и переменную этого типа.
  v_CursorVar t_FlexibleRef;
```

# Выделение памяти курсорным переменным

Курсорная переменная имеет ссылочный тип, поэтому при ее объявлении для хранения переменной память не отводится. Перед началом работы с курсорной переменной необходимо указать на подходящую область памяти, которую можно создать двумя способами: выделив ее в программе OCI или предкомпилятора либо автоматически самим PL/SQL.

▼ ВНИМАНИЕ

**PL/SQL 2.2 не может автоматически выделять пространство для хранения ссылочных переменных. Поэтому для использования курсорных переменных в этой версии требуется либо OCI, либо предкомпилятор, так как только эти средства могут выделять необходимую память. PL/SQL 2.3 может выделять пространство для хранения курсорных переменных, и поэтому наличие предкомпилятора или OCI здесь не является необходимым условием работы.**

## Использование EXEC SQL ALLOCATE

Чтобы выделить память при использовании предкомпилятора Pro*C, необходимо объявить переменную с типом SQL_CURSOR в разделе DECLARE программы Pro*C, так как она является базовой переменной. После этого можно выделить для нее память при помощи команды EXEC SQL ALLOCATE. Например, в следующем фрагменте Pro*C курсорная переменная объявляется и для нее выделяется память:

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL_CURSOR v_CursorVar;
EXEC SQL END DECLARE SECTION;

EXEC SQL ALLOCATE :v_CursorVar;
```

Использование Pro*C и других предкомпиляторов более подробно рассматривается в главе 13. Базовые курсорные переменные являются неограниченными, так как с ними не связываются возвращаемые типы. Это единственный способ объявления неограниченных курсорных переменных в PL/SQL 2.2.

## Автоматическое выделение памяти

PL/SQL 2.3 ... и ВЫШЕ

**В PL/SQL 2.3 память для курсорных переменных выделяется автоматически по мере необходимости. Когда переменная выходит за пределы области своего действия и поэтому больше не ссылается на место своего хранения, выделение памяти отменяется.**

# Открытие курсорной переменной для запроса

Для связи курсорной переменной с определенным оператором SELECT используется расширение синтаксиса оператора OPEN, позволяющее указать требуемый запрос:

OPEN *курсорная_переменная* FOR *оператор_выбора;*

где *курсорная_переменная* — это ранее объявленная курсорная переменная, а *оператор_выбора* — требуемый запрос. Если курсорная переменная ограничена, список выбора должен соответствовать типу, возвращаемому курсором. Если переменная не ограничена, выдается сообщение об ошибке:

```
ORA-6504: PL/SQL: return types of result set variables or query
          do not match
```

(PL/SQL: не установлено соответствие между возвращаемыми типами переменных результирующего набора и запросом. — *Прим. пер.*)

Более детальная информация об ошибках PL/SQL и том, что делать в случае их возникновения, приведена в главе 10. В качестве примера рассмотрим следующую курсорную переменную:

```
DECLARE
      TYPE t_ClassesRef IS REF CURSOR RETURN classes%ROWTYPE;
v_ClassesCV t_ClassesRef;
```

Теперь откроем **v_ClassesCV**:

```
OPEN v_ClassesCV FOR
  SELECT * FROM classes;
```

Если же попытаться открыть **v_ClassesCV** так:

```
OPEN v_ClassesCV FOR
    SELECT department, course FROM classes;
```

то будет выдано сообщение об ошибке ORA-6504, так как список выбора запроса не соответствует типу, возвращаемому этой курсорной переменной.

Оператор OPEN..FOR в принципе аналогичен оператору OPEN – анализируются переменные привязки и определяется активный набор. После выполнения OPEN..FOR можно считывать информацию из курсорной переменной с помощью оператора FETCH. В версии 2.2 операцию считывания необходимо выполнять на станции клиента, а в версии 2.3 эта операция может быть выполнена на сервере.

## Закрытие курсорных переменных

Курсорные переменные закрываются точно так же, как и статические курсоры – при помощи оператора CLOSE. При этом освобождаются ресурсы, используемые запросом, однако память, отведенная для хранения самой курсорной переменной, освобождается не всегда, а только когда переменная выходит из области своего действия. Запрещается закрывать ранее закрытые курсоры и курсорные переменные. Курсорные переменные могут быть закрыты либо на станции клиента, либо на сервере.

## Пример курсорной переменной 1

Ниже приведена полная программа Pro*C, демонстрирующая использование курсорных переменных. В ней имеется встроенный блок PL/SQL, предназначенный для выбора данных либо из таблицы **classes**, либо из таблицы **rooms**, в зависимости от ввода пользователя. Для читателей, незнакомых с языком C, в текст программы внесено больше комментариев, чем обычно. Эта программа работает с PL/SQL 2.2 и выше.

```
-- Этот пример содержится в файле cursor1.pc.
/* Подключим файлы заголовков C и SQL . */
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;

/* Раздел объявлений SQL. Здесь должны быть объявлены все базовые
   переменные. */
EXEC SQL BEGIN DECLARE SECTION;
  /* Строка символов для хранения имени и пароля пользователя. */
  char *v_Username = "example/example";

  /* Курсорная переменная SQL. */
  SQL_CURSOR v_CursorVar;

  /* Целочисленная переменная, используемая для управления выбором таблицы. */
  int v_Table;

  /* Выходная переменная для таблицы rooms. */
  int v_RoomID;
  VARCHAR v_Description[2001];

  /* Выходная переменная для таблицы classes. */
  VARCHAR v_Department[4];
  int v_Course;
EXEC SQL END DECLARE SECTION;

/* Подпрограмма обработки ошибок – печать ошибки и выход. */
void handle_error() {
  printf("SQL Error occurred!\n");
  printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
```

```
}

int main() {
  /* Строка символов для хранения ввода пользователя. */
  char v_Choice[20];

  /* Установим обработку ошибок. Всякий раз, когда происходит ошибка
     SQL, будет вызываться подпрограмма handle_error(). */
  EXEC SQL WHENEVER SQLERROR DO handle_error();

  /* Соединимся с базой данных. */
  EXEC SQL CONNECT :v_Username;
  printf("Connected to Oracle.\n");

  /* Выделим память для курсорной переменной. */
  EXEC SQL ALLOCATE :v_CursorVar;

  /* Напечатаем сообщение, запрашивающее ввод пользователя, и
     запишем введенное значение в переменную v_Choice. */
  printf("Choose from (C)lasses or (R)ooms. Enter c or r: ");
  Gets(v_Choice);

  /* Определим нужную таблицу. */
     if (v_Choice[0] == 'c')
       v_Table = 1;
     else
       v_Table = 2;

  /* Откроем курсорную переменную с помощью встроенного блока PL/SQL. */
EXEC SQL EXECUTE
  BEGIN
     IF :v_Table = 1 THEN
       /* Откроем переменную для таблицы classes. */
       OPEN :v_CursorVar FOR
         SELECT department, course
           FROM classes;
     ELSE
       /* Откроем переменную для таблицы rooms. */
       OPEN :v_CursorVar FOR
         SELECT room_id, description
           FROM rooms;
     END IF;
  END;
END-EXEC;

/* Когда произведено считывание данных, выйдем из цикла. */
EXEC SQL WHENEVER NOT FOUND DO BREAK;

/* Начнем цикл выборки. */
for (;;) {
  if (v_Table == 1) {
     /* Считаем информацию о группе. */
     EXEC SQL FETCH :v_CursorVar
       INTO :v_Department, :v_Course;
```

```
        /* Отобразим полученную информацию на экране. v_Department
           имеет тип VARCHAR, поэтому воспользуемся полем .len для
           фактической длины и полем .arr для данных. */
        printf("%.*s%d\n", v_Department.len, v_Department.arr,
                           v_Course);
    }
  else {
    /* Считаем информацию об аудитории. */ /* Fetch room info. */
    EXEC SQL FETCH :v_CursorVar
        INTO :v_RoomID, v_Description;

    /* Отобразим полученную информацию на экране. v_Description
       имеет тип VARCHAR, поэтому воспользуемся полем .len для
       фактической длины и полем .arr для данных. */
    printf("%d %.*s\n", v_RoomID, v_Description.len,
                        v_Description.arr);
        }
    }

    /* Закроем курсор. */
    EXEC SQL CLOSE :v_CursorVar;

    /* Отключимся от базы данных. */
    EXEC SQL COMMIT WORK RELEASE;
}
```

В рассмотренной программе курсор открывается на сервере (посредством встроенного анонимного блока), а считывается и закрывается на станции клиента. Курсорная переменная объявлена как базовая, поэтому она не ограничена. Таким образом, одна и та же переменная была использована для выбора информации как из таблицы **classes**, так и из таблицы **rooms**.

## Пример курсорной переменной 2

PL/SQL 2.3 ... и ВЫШЕ

Следующий пример похож на рассмотренный в предыдущем разделе пример программы Pro*C, однако он написан целиком на PL/SQL. Это хранимая процедура, с помощью которой выбирается информация из таблицы **classes** или из таблицы **rooms**, в зависимости от ввода. Считывание строк происходит в процедуре, т.е. на сервере, поэтому данная программа будет работать только в PL/SQL 2.3. Более подробно о хранимых процедурах говорится в главе 7.

```
-- Этот пример содержится в файле cursor2.sql.
CREATE OR REPLACE PROCEDURE ShowCursorVariable
  /* Демонстрирует использование курсорной переменной на сервере.
     Если p_Table - это 'classes', то в таблицу temp_table вводится
     информация из таблицы classes. Если p_Table - 'rooms',
     то вводится информация из таблицы rooms. */
(p_Table IN VARCHAR2) AS

  /* Опишем тип курсорной переменной */
  TYPE t_ClassesRooms IS REF CURSOR;

  /* и собственно переменной. */
  v_CursorVar t_ClassesRooms;

  /* Переменные для хранения результатов. */
  v_Department classes.department%TYPE;
```

```
    v_Course  classes.course%TYPE;
    v_RoomID  rooms.room_id%TYPE;
    v_Description  rooms.description%TYPE;
  BEGIN
    -- На основании входного параметра откроем курсорную переменную.
    IF p_Table = 'classes' THEN
        OPEN v_CursorVar FOR
          SELECT department, course
            FROM classes;
    ELSIF p_table = 'rooms' THEN
        OPEN v_CursorVar FOR
          SELECT room_id, description
            FROM rooms;
    ELSE
        /* Введено неверное значение - возникает ошибка. */
        RAISE_APPLICATION_ERROR(-20000,
          'Input must be ''classes''' or ''rooms''');
    END IF;

    /* Цикл выборки. Обратите внимание, что оператор EXIT WHEN
       расположен после оператора FETCH - в PL/SQL 2.3 нельзя
       с курсорными переменными использовать курсорные атрибуты. */
    LOOP
        IF p_Table = 'classes' THEN
          FETCH v_CursorVar INTO
            v_Department, v_Course;
          EXIT WHEN v_CursorVar%NOTFOUND;

          INSERT INTO temp_table (num_col, char_col)
            VALUES (v_Course, v_Department);
        ELSE
          FETCH v_CursorVar INTO
            v_RoomID, v_Description;
          EXIT WHEN v_CursorVAR%NOTFOUND;

          INSERT INTO temp_table (num_col, char_col)
            VALUES (v_RoomID, SUBSTR(v_Description, 1, 60));
        END IF;
    END LOOP;

    /* Закроем курсор. */
    CLOSE v_CursorVar;
    COMMIT;
  END ShowCursorVariable;
```

## Ограничения на использование курсорных переменных

Курсорные переменные являются достаточно мощным средством, и их применение намного упрощает обработку информации, поскольку они позволяют возвращать в одной переменной данные самых различных типов. Однако на их использование налагается ряд ограничений (в версиях 2.3 и 8.0 некоторые из них были отменены). Рассмотрим эти ограничения:

- В PL/SQL 2.2 курсорные переменные нельзя объявлять в модуле. Дело в том, что память для хранения курсорной переменной необходимо выделять при помощи Pro*C или OCI. В PL/SQL 2.2 существует единственный способ передачи курсорной переменной блоку PL/SQL — через переменную привязки или параметр процедуры. Это ограничение устранено в PL/SQL 2.3.

- Удаленные подпрограммы не могут возвращать значение курсорной переменной. Это ограничение действует и в версии 2.3 при передаче курсорных переменных от одного сервера другому, однако между клиентами и серверами они могут передаваться без всяких ограничений.
- В PL/SQL 2.2 нельзя использовать курсорные атрибуты с курсорными переменными. Это ограничение устранено в версии 2.3. Однако атрибут %ROWTYPE не может быть применен для курсорной переменной в любой версии.
- В версии 2.2 все операции считывания строк из курсорной переменной необходимо выполнять на станциях клиентов с помощью программ, написанных на Pro*C или OCI. В версии 2.3 можно выбирать информацию из курсорных переменных на сервере.
- В обеих версиях нельзя хранить курсорные переменные в таблицах PL/SQL.
- Нельзя использовать курсорные переменные с динамическим SQL в Pro*C.
- Запрос, связанный с курсорной переменной в операторе OPEN..FOR, не может иметь тип FOR UPDATE.

# Итоги

В этой главе были рассмотрены операции, выполняемые при обработке курсоров, которые позволяют явным образом управлять обработкой SQL-операторов. Для явных курсоров это операции объявления, открытия, считывания и закрытия курсора. Курсорные атрибуты используются для определения текущего состояния курсора, т.е. для управления его функционированием. Кроме того, обсуждались различные виды циклов выборки и был дан анализ курсорных переменных и различий в их реализации в PL/SQL 2.2 и 2.3. В следующих трех главах продолжается рассмотрение основных средств PL/SQL: процедур, функций, модулей и триггеров.

# Глава 7

## *Подпрограммы: процедуры и функции*

Блоки PL/SQL, рассмотренные выше, были анонимными. Анонимные блоки компилируются каждый раз при их выполнении. Они не хранятся в базе данных, и их нельзя непосредственно вызывать из других блоков PL/SQL. В следующих трех главах описываются программные конструкции: процедуры, функции, модули и триггеры. Они являются именованными блоками, и поэтому для них вышеназванные ограничения отсутствуют. Такие конструкции можно хранить в базе данных и выполнять по мере необходимости. Процедуры и функции — предмет изучения этой главы.

# Создание процедур и функций

Процедуры и функции PL/SQL очень похожи на процедуры и функции других языков третьего поколения и обладают многими их свойствами. В совокупности процедуры и функции называются *подпрограммами* (subprograms). Для примера рассмотрим следующий фрагмент программы, создающей процедуру в базе данных:

```
-- Этот пример содержится в файле addstud.sql.
CREATE OR REPLACE PROCEDURE AddNewStudent (
  P_FirstName    students.first_name%TYPE,
  P_LastName     students.last_name%TYPE,
  P_Major        students.major%TYPE) AS
BEGIN
  -- Внесем новую строку в таблицу students. Воспользуемся
  -- последовательностью student_sequence, чтобы сгенерировать идентификатор
  -- для нового студента и 0 для current_credits.
  INSERT INTO students (ID, first_name, last_name,
                        major, current_credits)
     VALUES (student_sequence.nextval, p_FirstName, p_LastName,
             p_Major, 0);

  COMMIT;
END AddNewStudent;
```

После того как процедура создана, ее можно вызывать из другого блока PL/SQL, как, например, в следующем примере:

```
BEGIN
  AddNewStudent('David', 'Dinsmore', 'Music');
END;
```

Этим примером иллюстрируются важные аспекты:

- Процедура **AddNewStudent** создается при помощи оператора CREATE OR REPLACE PROCEDURE. Она сначала компилируется, а затем сохраняется в базе данных в скомпилированном виде. Скомпилированный код можно впоследствии выполнить из другого блока PL/SQL.
- *При вызове процедуры ей можно передавать параметры.* В *нашем примере во время выполнения* процедуре передаются имя, фамилия и профилирующий предмет студента. Внутри процедуры параметр **p_FirstName** будет иметь значение 'David', **p_LastName** — 'Dinsmore', а **p_Major** — 'Music', так как именно эти литералы были указаны при вызове процедуры.
- Вызовом процедуры является оператор PL/SQL; вызов процедуры не может быть частью выражения. При вызове процедуры управление программой передается первому исполняемому оператору этой процедуры. Когда процедура заканчивается, управление возвращается оператору, следующему за вызовом этой процедуры. В этом смысле процедуры PL/SQL функционируют точно так же, как и процедуры других языков третьего поколения.
- Процедура — это блок PL/SQL, в состав которого входят раздел объявлений, выполняемый раздел и раздел исключительных ситуаций. Как и в анонимных блоках, необходимым здесь является только выполняемый раздел. Например, в процедуре **AddNewStudent** содержится лишь выполняемый раздел.

## Создание процедуры

Синтаксис оператора CREATE OR REPLACE PROCEDURE таков:

```
CREATE [OR REPLACE] PROCEDURE имя_процедуры
[(аргумент [{IN | OUT |IN OUT}] тип,
...
аргумент[{IN | OUT |IN OUT}] тип)] {IS | AS}
тело_процедуры
```

где *имя_процедуры* – это имя создаваемой процедуры, *аргумент* – имя параметра процедуры, *тип* – это тип соответствующего параметра, а *тело_процедуры* – блок PL/SQL, в котором содержится текст процедуры.

Для изменения текста процедуры необходимо удалить и повторно создать ее. Во время разработки процедур эта операция выполняется достаточно часто, поэтому ключевые слова OR REPLACE (или заменить) позволяют выполнить такую операцию за один раз. Если процедура существует, она сначала удаляется безо всякого предупреждения, для чего используется команда DROP PROCEDURE, описанная в разделе "Удаление процедур и функций" в этой главе ниже. Если процедура до этого не существовала, ее нужно создать. Если же процедура существует, а ключевые слова OR REPLACE не указаны, оператор CREATE возвращает ошибку Oracle:

```
ORA-00955: name is already used by an existing object
```

(имя уже используется существующим объектом. – *Прим. пер.*)

Как и другие операторы CREATE, создание процедуры является операцией DDL, поэтому до и после создания процедуры неявно выполняются операторы COMMIT. При этом можно использовать эквивалентные друг другу ключевые слова IS и AS.

## Параметры и их виды

Обратимся к рассмотренной выше процедуре AddNewStudent и вызовем ее из следующего анонимного блока PL/SQL:

```
-- Этот пример содержится в файле callproc.sql.
DECLARE
  -- Переменные, описывающие нового студента
  v_NewFirstName students.first_name%TYPE := 'Margaret';
  v_NewLastName students.last_name%TYPE := 'Mason';
  v_NewMajor students.major%TYPE := 'History';
BEGIN
  -- Введем сведения о Margaret Mason в базу данных.
  AddNewStudent(v_NewFirstName, v_NewLastName, v_NewMajor);
END;
```

Переменные, объявленные в предыдущем блоке (**v_NewFirstName, v_NewLastName, v_NewMajor**), передаются процедуре **AddNewStudent** в качестве аргументов. В этом контексте они являются *фактическими параметрами* (actual parameters), а параметры, указанные при объявлении процедуры (**p_FirstName, p_LastName, p_Major**), – *формальными параметрами* (formal parameters). Фактические параметры содержат значения, передаваемые процедуре при ее вызове, и результаты, возвращаемые процедурой. Именно значения фактических параметров используются в процедуре. Формальные параметры выступают в роли вместилища фактических параметров. При вызове процедуры формальным параметрам присваиваются значения фактических параметров. Внутри процедуры все действия выполняются над формальными параметрами. По окончании процедуры фактическим параметрам присваиваются значения формальных параметров. Все операции присваивания соответствуют стандартным правилам, установленным в PL/SQL для таких операций, в том числе правилам преобразования типов.

Формальные параметры бывают трех видов (modes): IN (в), OUT (из) и IN OUT. Если же вид не указан, то по умолчанию устанавливается IN. Различия между этими видами описаны в таблице 7.1 и проиллюстрированы следующим примером:

```
-- Этот пример содержится в файле modetest.sql.
CREATE OR REPLACE PROCEDURE ModeTest (
  p_InParameter IN NUMBER,
  p_OutParameter OUT NUMBER,
  p_InOutParameter IN OUT NUMBER) IS
```

```
    v_LocalVariable NUMBER;
BEGIN
    /* Присвоим p_InParameter переменной v_LocalVariable. Это
       разрешается делать, так как производится чтение параметра IN и
       информация в него не записывается. */
    v_LocalVariable := p_InParameter; -- Правильно.

    /* Присвоим 7 параметру p_InParameter. Этого делать НЕЛЬЗЯ, так
       как производится запись в параметр IN. */
    p_InParameter := 7; -- Неправильно.

    /* Присвоим 7 параметру p_OutParameter. Это разрешается делать,
       так как производится запись в параметр OUT, а не чтение из него. */
    p_OutParameter := 7; -- Правильно.

    /* Присвоим p_OutParameter переменной v_LocalVariable. Этого
       делать НЕЛЬЗЯ, так как производится чтение параметра OUT. */
    v_LocalVariable := p_outParameter; -- Неправильно.

    /* Присвоим p_InOutParameter переменной v_LocalVariable. Это
       разрешается делать, так как производится чтение параметра
       IN OUT. */
    v_LocalVariable := p_InOutParameter; -- Правильно.

    /* Присвоим 7 параметру p_InOutParameter. Это разрешается делать,
       так как производится запись в параметр IN OUT. */
    p_InOutParameter := 7; -- Правильно.
END ModeTest;
```

▼ ВНИМАНИЕ

В примере ***ModeTest*** показаны правильные и неправильные операции присваивания PL/SQL. Значения переменным можно присвоить и в конструкции INTO операторов SELECT..INTO и FETCH..INTO, причем на эти переменные налагаются те же самые ограничения.

**ТАБЛИЦА 7.1. Виды параметров**

| Вид | Описание |
|---|---|
| IN | Значение фактического параметра передается в процедуру при ее вызове. Внутри процедуры формальный параметр рассматривается в качестве параметра *только для чтения* — он не может быть изменен. Когда процедура завершается и управление программой возвращается в вызывающую среду, фактический параметр не изменяется. |
| OUT | Любое значение, имеющее фактический параметр при вызове процедуры, игнорируется. Внутри процедуры формальный параметр рассматривается как параметр *только для записи* — ему можно только присвоить значение, но считать из него значение нельзя. Когда процедура завершается и управление программой возвращается в вызывающую среду, содержимое формального параметра присваивается фактическому параметру. |
| IN OUT | Этот вид представляет собой комбинацию видов IN и OUT. Значение фактического параметра передается в процедуру при ее вызове. Внутри процедуры формальный параметр может быть считан, и в него может быть записано значение. Когда процедура завершается и управление программой возвращается в вызывающую среду, содержимое формального параметра присваивается фактическому параметру. |

При создании процедуры PL/SQL проверяет правильность операций присваивания. Например, при попытке скомпилировать **ModeTest** будет выдано сообщение об ошибке, так как в этой процедуре содержатся неверные операции присваивания:

```
PLS-363: expression 'P_INPARAMETER' cannot be used as an assignmenttarget
```
(выражение 'P_INPARAMETER' нельзя использовать в качестве объекта назначения операции присваивания. — *Прим. пер.*)
```
PLS-365: 'P_OUTPARAMETER' is an OUT parameter and cannot be read
```
('P_OUTPARAMETER' является параметром OUT и не может быть считан. — *Прим. пер.*)

Параметр IN является значением выражения (rvalue) внутри процедуры; его можно указывать только в правой части операции присваивания. Параметр OUT — это именующее значение (lvalue); его можно указывать только в левой части операции присваивания и никогда в правой, даже когда он присваивается чему-либо в процедуре (более подробно о значениях выражений и именующих значениях см. в главе 2). Параметр IN OUT — это и значение выражения, и именующее значение, поэтому он может быть указан в любой части операции присваивания.

Если параметр имеет вид OUT или IN OUT, то процедура будет записывать значения в фактический параметр. При завершении процедуры содержимое формальных параметров записывается в соответствующие фактические параметры, которые поэтому должны быть именующими значениями. Литералы являются значениями выражений, и память для их постоянного хранения не выделяется; поэтому литералы не могут быть использованы для параметров OUT или IN OUT. Например, ниже приведен правильный вызов процедуры ModeTest, так как фактическими параметрами для **p_OutParameter** и **p_InOutParameter** являются переменные:

```
DECLARE
  v_Variable1 NUMBER;
  v_Variable2 NUMBER;
BEGIN
  ModeTest(12, v_Variable1, v_Variable2);
END;
```

Однако если заменить **v_Variable2** литералом, то вызов процедуры будет неверен:

```
DECLARE
  v_Variable1 NUMBER;
BEGIN
  ModeTest(12, v_Variable1, 11);
END;
```

При этом выдаются следующие сообщения об ошибках:

```
ERROR at line 4:
ORA-06550: line 4, column 29:
PLS-363: expression '11' cannot be used as an assignment target
ORA-06550: line 4, column 3:
PL/SQL: Statement ignored
```
(**ОШИБКА** в строке 4:
ORA-06550: строка 4, столбец 29:
PLS-00363: выражение '11' нельзя использовать в качестве назначения операции присваивания
ORA-06550: строка 4, столбец 3:
PL/SQL: Оператор игнорируется. — *Прим. пер.*)

Если процедуре не передаются параметры, то круглые скобки не нужны ни при ее объявлении, ни при ее вызове. Это справедливо также и для функций, описанных ниже в данной главе.

## Тело процедуры

Тело (body) процедуры — это блок PL/SQL, содержащий раздел объявлений, выполняемый раздел и раздел исключительных ситуаций. Раздел объявлений располагается между ключевыми словами IS или AS и ключевым словом BEGIN. Выполняемый раздел (единственный, который обязателен) — между ключевыми словами BEGIN и EXCEPTION. Раздел исключительных ситуаций находится между ключевыми словами EXCEPTION и END.

▼ ВНИМАНИЕ

В описании процедуры или функции ключевое слово DECLARE отсутствует. Вместо него используется ключевое слово IS или AS. В этом вновь проявляется аналогия с синтаксисом языка программирования Ada.

Таким образом, структура процедуры такова:

```
CREATE OR REPLACE PROCEDURE имя_процедуры AS
  /* Раздел объявлений. */
BEGIN
  /* Выполняемый раздел. */
EXCEPTION
  /* Раздел исключительных ситуаций. */
END [имя_процедуры];
```

При описании процедуры ее имя можно по желанию указывать в конце последнего оператора END. Если после END нет никакого идентификатора, этот оператор должен соответствовать нужному имени процедуры. Указание имени процедуры в конце является хорошим стилем, так как выделяет оператор END, соответствующий оператору CREATE.

## Ограничения на формальные параметры

При вызове процедуры ей передаются значения фактических параметров, и внутри процедуры к этим значениям обращаются с помощью формальных параметров. При этом передаются не только значения, но и ограничения, наложенные на переменные. Описывая процедуры, запрещается ограничивать длину параметров типа CHAR и VARCHAR2, а также точность и/или масштаб параметров типа NUMBER. Например, следующее описание процедуры неверно и приводит к выдаче ошибки компиляции:

```
-- Этот пример является частью файла plength.sql.
CREATE OR REPLACE PROCEDURE ParameterLength (
  p_Parameter1 IN OUT VARCHAR2(10),
  p_Parameter2 IN OUT NUMBER(3,2)) AS
BEGIN
  p_Parameter1 := 'abcdefghijklm';
  p_Parameter2 := 12.3;
END ParameterLength;
```

Правильное описание должно выглядеть примерно так:

```
-- Этот пример является частью файла plength.sql.
CREATE OR REPLACE PROCEDURE ParameterLength (
  p_Parameter1 IN OUT VARCHAR2,
  p_Parameter2 IN OUT NUMBER) AS
BEGIN
  p_Parameter1 := 'abcdefghijklmno';
  p_Parameter2 := 12.3;
END ParameterLength;
```

Итак, ограничения, наложенные на **p_Parameter1** и **p_Parameter2**, возникли из фактических параметров. Если вызвать **ParameterLength** следующим образом:

```
-- Этот пример является частью файла plength.sql.
DECLARE
  v_Variable1 VARCHAR2(40);
  v_Variable2 NUMBER(3,4);
BEGIN
  ParameterLength(v_Variable1, v_Variable2);
END;
```

то параметр **p_Parameter1** будет иметь максимальную длину 40 (задано фактическим параметром **v_Variable1**), а для параметра **p_Parameter2** будут установлены точность 3 и масштаб 4 (заданы фактическим параметром **v_Variable2**). Очень важно понимать это. Рассмотрим блок, в котором также вызывается **ParameterLength**:

```
-- Этот пример является частью файла plength.sql.
DECLARE
  v_Variable1 VARCHAR2(10);
```

```
  v_Variable2  NUMBER(3,4);
BEGIN
  ParameterLength(v_Variable1,  v_Variable2);
END;
```

Единственным отличием этого блока от предыдущего является то, что длина **v_Variable1**, а следовательно, и **p_Parameter1** равна 10, а не 40. В процедуре **ParameterLength** параметру **p_Parameter1** (а следовательно, и **v_Variable1**) присваивается строка, состоящая из 15 символов, поэтому пространства для нее недостаточно. В результате при вызове данной процедуры Oracle выдает следующую ошибку:

```
ORA-6502: numeric or value error
```

(ошибка числа или значения. — *Прим. пер.*)

Причина ошибки заключается не в процедуре, а в тексте программы, вызывающей эту процедуру.

▼ СОВЕТУЕМ **Во избежание ошибок, подобных ORA-6502, при создании процедур документируйте все ограничения, налагаемые на фактические параметры, — вносите в сохраняемые процедуры комментарии, а также, кроме описания каждого параметра, записывайте функции, выполняемые самой процедурой.**

**%TYPE и параметры процедур** Единственным способом наложения ограничения на формальные параметры является использование атрибута %TYPE. Если формальный параметр объявлен при помощи %TYPE, а базовый тип ограничен, это ограничение распространяется не на фактический параметр, а на формальный. Если **ParameterLength** объявить следующим образом:

```
-- Этот пример является частью файла plength.sql.
CREATE OR REPLACE PROCEDURE ParameterLength (
  p_Parameter1 IN OUT VARCHAR2,
  p_Parameter2 IN OUT students.current_credits%TYPE) AS
BEGIN
  p_Parameter2 := 12345;
END ParameterLength;
```

то **p_Parameter2** будет ограничен точностью 3, поскольку эта точность задана для столбца **current_credits.** Даже если вызвать **ParameterLength** с фактическим параметром достаточной точности, то используется точность формального параметра. Ниже приведен пример, в результате выполнения которого возвращается ошибка ORA-6502:

```
-- Этот пример является частью файла plength.sql.
DECLARE
  v_Variable1 VARCHAR2(1);
  v_Variable2 NUMBER; — Объявим v_Variable2 без ограничений.
BEGIN
  -- Хотя в фактическом параметре достаточно места для 12345,
  -- учитывается ограничение, наложенное на формальный параметр,
  -- и при вызове этой процедуры выдается ошибка ORA-6502.
  ParameterLength(v_Variable1, v_Variable2);
END;
```

## Позиционное и именное представления

Во всех предыдущих примерах этой главы фактические аргументы связаны с формальными аргументами по позициям. Взяв за основу процедуру

```
-- Этот пример является частью файла callme.sql.
CREATE OR REPLACE PROCEDURE CallMe (
  p_ParameterA VARCHAR2,
  p_ParameterB NUMBER,
  p_ParameterC BOOLEAN,
  p_ParameterD DATE) AS
BEGIN
  NULL;
```

```
END CallMe;
```

и вызывающий блок

```
-- Этот пример является частью файла callme.sql.
DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,6);
  v_Variable3 BOOLEAN;
  v_Variable4 DATE;
BEGIN
  CallMe(v_Variable1, v_Variable2, v_Variable3, v_Variable4);
END;
```

можно видеть, что фактические параметры связаны с формальными по позициям: **v_Variable1** связана с **p_ParameterA**, **v_Variable2** – с **p_ParameterB** и т.д. Эта связь называется *позиционным представлением* (positional notation) и применяется наиболее часто. Именно такое представление используется в других языках третьего поколения, например в C.

В качестве альтернативы можно вызвать процедуру при помощи *именного представления* (named notation):

```
-- Этот пример является частью файла callme.sql.
DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,6);
  v_Variable3 BOOLEAN;
  v_Variable4 DATE;
BEGIN
  CallMe(p_ParameterA => v_Variable1,
  p_ParameterB => v_Variable2,
  p_ParameterC => v_Variable3,
  p_ParameterD => v_Variable4);
END;
```

При именном представлении для каждого аргумента указываются как формальный, так и фактический параметры. Это дает возможность при желании установить собственный порядок аргументов. Например, в следующем блоке вызывается CallMe с теми же аргументами:

```
-- Этот пример является частью файла callme.sql.
DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,6);
  v_Variable3 BOOLEAN;
  v_Variable4 DATE;
BEGIN
  CallMe(p_ParameterB => v_Variable2,
        p_ParameterC => v_Variable3,
        p_ParameterD => v_Variable4,
        p_ParameterA => v_Variable1);
END;
```

Кроме того, при необходимости позиционное и именное представления можно комбинировать в одном и том же вызове. Несколько первых аргументов нужно указывать по позициям, а оставшиеся – по именам. Проиллюстрируем такой способ:

```
-- Этот пример является частью файла callme.sql.
DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,6);
  v_Variable3 BOOLEAN;
  v_Variable4 DATE;
```

```
BEGIN
  -- Первые 2 параметра указываются по позициям, а другие 2 - по именам.
  CallMe(v_Variable1, v_Variable2,
         p_ParameterC => v_Variable3,
         p_ParameterD => v_Variable4);
END;
```

Именное представление – это еще одно свойство PL/SQL, заимствованное из языка Ada. Ни позиционное представление, ни именное не является более или менее эффективным, поэтому использование их зависит от стиля программирования. Некоторые стилистические особенности этих представлений, а также различие между ними проиллюстрированы в таблице 7.2.

**ТАБЛИЦА 7.2. Позиционное и именное представления**

| Позиционное представление | Именное представление |
|---|---|
| Для фактических параметров задаются более понятные имена, демонстрирующие назначение каждого из них | Четче показывает связь между фактическими и формальными параметрами |
| Имена, используемые для формальных и фактических параметров, не зависят друг от друга; одно может быть изменено без модификации другого | Сложнее изменять все вызовы процедуры при модификации формальных параметров |
| Сложнее изменять все вызовы процедуры при модификации порядка формальных параметров | Порядок указания формальных параметров не зависит от порядка указания фактических; один может быть изменен без модификации другого |
| Более краткое, чем именное представление | Программы сложнее, так как при вызове процедуры указываются и формальные, и фактические параметры |
| Формальные параметры, значения которых заданы по умолчанию, нужно указывать в конце списка параметров | Разрешается использовать для формальных параметров значения по умолчанию[1], независимо от того, какой параметр имеет значение по умолчанию |

[1]Параметры по умолчанию обсуждаются в следующем разделе

▼ СОВЕТУЕМ Чем больше параметров в процедуре, тем сложнее ее вызывать и тем труднее убеждаться в наличии всех требуемых параметров. Если необходимо передать в процедуру или получить из нее достаточно большое число параметров, то рекомендуется определить тип записи, полями которой будут эти параметры. Затем можно использовать единственный параметр, имеющий тип записи. В PL/SQL не установлено явного ограничения на число параметров.

## Значения параметров по умолчанию

Как и переменные, формальные параметры процедуры или функции могут иметь значения по умолчанию. В таком случае параметр можно не передавать из вызывающей среды. Если же параметр передается, вместо значения по умолчанию берется фактический параметр. Значение по умолчанию для параметра указывается следующим образом:

*имя_параметра* [*вид*] *тип_параметра* {:= | DEFAULT} *исходное_значение*

где *имя_параметра* – это имя формального параметра, *вид* – вид параметра ((IN, OUT или IN OUT), *тип_параметра* – тип параметра (либо предварительно определенный, либо определяемый пользователем), а *исходное_значение* – значение, присваиваемое формальному параметру по умолчанию. Можно применять или символы :=, или ключевое слово DEFAULT. Для примера перепишем процедуру **AddNewStudent**, присвоив по умолчанию всем новым студентам профилирующую дисциплину "Экономика" (если это не отменяется явным аргументом):

```
-- Этот пример содержится в файле default.sql.
CREATE OR REPLACE PROCEDURE AddNewStudent (
  p_FirstName students.first_name%TYPE,
  p_LastName students.last_name%TYPE,
  p_Major students.major%TYPE DEFAULT 'Economics') AS
BEGIN
```

```
    -- Внесем новую строку в таблицу students. Воспользуемся
    -- последовательностью student_sequence, чтобы
    -- сгенерировать идентификатор для нового студента и 0 для
    -- current_credits.
  INSERT INTO students VALUES (student_sequence.nextval,
       p_FirstName, p_LastName, p_Major, 0);

    COMMIT;
  END AddNewStudent;
```

Если формальный параметр **p_Major** не связан в вызове процедуры с фактическим параметром, используется значение по умолчанию. Связать параметры можно при помощи позиционного представления:

```
BEGIN
  AddNewStudent('Barbara', 'Blues');
END;
```

или именного представления:

```
BEGIN
  AddNewStudent(p_FirstName => 'Barbara',
                p_LastName => 'Blues');
END;
```

Если применяется позиционное представление, то все те параметры со значениями по умолчанию, для которых не установлено соответствие с фактическими параметрами, должны находиться в конце списка параметров. Рассмотрим следующий пример:

```
CREATE OR REPLACE PROCEDURE DefaultTest (
  p_ParameterA NUMBER DEFAULT 10,
  p_ParameterB VARCHAR2 DEFAULT 'abcdef',
  p_ParameterC DATE DEFAULT sysdate) AS
BEGIN
  ...
END DefaultTest;
```

Все три параметра для **DefaultTest** используют аргументы, заданные по умолчанию. Если нужно задать значение по умолчанию только для параметра **p_ParameterB**, а для параметров **p_ParameterA** и **p_ParameterC** указать некоторые значения, необходимо использовать именное представление:

```
BEGIN
  DefaultTest(p_ParameterA => 7, p_ParameterC => '30-DEC-95');
END;
```

Если требуется указать значение по умолчанию для параметра **p_ParameterB** и использовать при этом позиционное представление, то для параметра **p_ParameterC** также нужно указывать значение по умолчанию. В позиционном представлении все параметры по умолчанию, для которых не установлено соответствие с фактическими параметрами, должны находиться в конце списка параметров:

```
BEGIN
  /* Значения по умолчанию задаются как для параметра p_ParameterB,
     так и для параметра p_ParameterC. */
  DefaultTest(7);
END;
```

▼ СОВЕТУЕМ Если возможно, значения по умолчанию указывайте последними в списке аргументов. В этом случае используются как позиционное, так и именное представления.

## Создание функций

Функции очень похожи на процедуры. Как те, так и другие принимают аргументы, которые могут иметь любой вид. Функции и процедуры – это различные формы блоков PL/SQL, в состав каждого из них мо-

гут входить раздел объявлений, выполняемый раздел и раздел исключительных ситуаций. Как функции, так и процедуры можно хранить в базе данных или описывать в блоке (процедуры и функции, не хранимые в базе данных, рассматриваются ниже, в разделе "Размещение подпрограмм"). Однако вызов процедуры сам по себе является оператором PL/SQL, в то время как вызов функции — это часть некоторого выражения.

Вызов функции является значением выражения (rvalue). Ниже приведен пример вызова функции, возвращающего TRUE, если указанная учебная группа заполнена более чем на 90 процентов, и FALSE — в противном случае.

```
-- Этот пример содержится в файле almostfl.sql.
CREATE OR REPLACE FUNCTION AlmostFull (
  p_Department classes.department%TYPE,
  p_Course classes.course%TYPE)
  RETURN BOOLEAN IS

  V_CurrentStudents   NUMBER;
  V_MaxStudents       NUMBER;
  V_ReturnValue       BOOLEAN;
  V_FullPercent       CONSTANT NUMBER := 90;
BEGIN
  -- Узнаем текущее и максимальное число студентов в указанной группе.
  SELECT current_students, max_students
      INTO v_CurrentStudents, v_MaxStudents
      FROM classes
      WHERE department = p_Department
      AND course = p_Course;

  -- Если процент заполнения группы более заданного в v_FullPercent,
  -- возвращается TRUE. В противном случае возвращается FALSE.
  IF (v_CurrentStudents / v_MaxStudents * 100) > v_FullPercent
  THEN
      v_ReturnValue := TRUE;
  ELSE
      v_ReturnValue := FALSE;
  END IF;
      RETURN v_ReturnValue;
END AlmostFull;
```

Функция **AlmostFull** возвращает логическое значение. Ниже показан блок PL/SQL, в котором вызывается эта функция. Внимание — вызов функции не является оператором, он представляет собой фрагмент условного оператора IF, расположенного внутри цикла.

```
-- Этот пример содержится в файле callfunc.sql.
DECLARE
  CURSOR c_Classes IS
      SELECT department, course
        FROM classes;
BEGIN
  FOR v_ClassRecord IN c_Classes LOOP
  -- Запишем информацию обо всех группах, в которых
  -- осталось слишком мало места, в таблицу temp_table.
  IF AlmostFull(v_ClassRecord.department, v_ClassRecord.course)
  THEN
      INSERT INTO temp_table (char_col) VALUES
        (v_ClassRecord.department || ' ' || v_ClassRecord.course
         || ' is almost full!');
      END IF;
```

```
    END LOOP;
END;
```

## Описание функций

Синтаксис для создания хранимой функции очень похож на синтаксис для создания процедуры:

```
CREATE [OR REPLACE] FUNCTION имя_функции
  [(аргумент [{IN | OUT |IN OUT}] тип,
     ...
     аргумент[{IN | OUT |IN OUT}] тип)]
RETURN возвращаемый_тип {IS | AS}
тело_функции
```

где *имя_функции* — это имя функции; *аргумент и тип* аналогичны аргументу и типу, указываемым при создании процедуры; *возвращаемый_тип* — это тип значения, возвращаемого функцией; *тело_функции* — блок PL/SQL, содержащий программный текст данной функции.

Как и для процедур, список аргументов необязателен. В этом случае ни при описании функции, ни при ее вызове круглые скобки указывать не нужно. Однако тип, возвращаемый функцией, необходим, так как вызов функции является частью некоторого выражения. Тип функции используется для определения типа выражения, содержащего вызов этой функции.

## Оператор RETURN

Внутри тела функции оператор RETURN применяется для возврата управления программой и результата выполнения функции в вызывающую среду. Общий синтаксис оператора RETURN выглядит следующим образом:

```
RETURN выражение;
```

где *выражение* — это возвращаемое значение. Значение выражения преобразуется к типу, указанному в команде RETURN при описании функции, если это значение уже не имеет данный тип. При выполнении оператора RETURN управление программой сразу же возвращается в вызывающую среду.

В функции может быть несколько операторов RETURN, хотя выполняться будет только один из них. Завершение функции без оператора RETURN является ошибкой. Ниже приведен пример, иллюстрирующий использование в одной функции нескольких операторов RETURN. При наличии пяти различных операторов RETURN выполняется лишь один из них. Какой оператор выполняется, зависит от заполненности учебной группы, указанной при помощи **p_Department** и **p_Course.**

```
-- Этот пример содержится в файле clasinfo.sql.
CREATE OR REPLACE FUNCTION ClassInfo (
  /* Возвращает 'Full', если группа заполнена до предела,
      'Some Room', если группа заполнена на 80%,
      'More Room', если группа заполнена на 60%,
      'Lots of Room', если группа заполнена менее чем на 60%, и
      'Empty', если ни одного студента не зарегистрировано. */
  p_Department classes.department%TYPE,
  p_Course classes.course%TYPE)
  RETURN VARCHAR2 IS

  V_CurrentStudents   NUMBER;
  V_MaxStudents       NUMBER;
  V_PercentFull       NUMBER;
BEGIN
  -- Узнаем текущее и максимальное число студентов в указанной группе.
  SELECT current_students, max_students
      INTO v_CurrentStudents, v_MaxStudents
      FROM classes
      WHERE department = p_Department
      AND course = p_Course;

  -- Рассчитаем текущее соотношение.
```

```
    v_PercentFull := v_CurrentStudents / v_MaxStudents * 100;

    IF v_PercentFull = 100 THEN
        RETURN 'Full';
    ELSIF v_PercentFull > 80 THEN
        RETURN 'Some Room';
    ELSIF v_PercentFull > 60 THEN
        RETURN 'More Room';
    ELSIF v_PercentFull > 0 THEN
        RETURN 'Lots of Room';
    ELSE
        RETURN 'Empty';
    END IF;
END ClassInfo;
```

При использовании оператора RETURN в функции с ним должно быть связано выражение. Однако RETURN можно использовать и в процедуре. В этом случае аргументы, приводящие к немедленной передаче управления программой обратно в вызывающую среду, не указываются. Текущие значения формальных параметров, описанных как OUT или IN OUT, присваиваются фактическим параметрам, и выполнение программы продолжается с оператора, следующего за вызовом процедуры.

## Свойства функций

Многие из свойств функций аналогичны свойствам процедур:

- Функции могут возвращать более одного значения при помощи параметра вида OUT
- Программный код функции состоит из раздела объявлений, выполняемого раздела и раздела исключительных ситуаций
- Функции могут использовать значения по умолчанию
- Функции можно вызывать, используя позиционное или именное представление

Когда применять функцию, а когда процедуру зависит от того, сколько значений должна возвращать данная подпрограмма и как будут использоваться эти значения. Обычно принято следующее правило: если возвращается более одного значения, нужно использовать процедуру, а если ровно одно, — то функцию. Хотя, в принципе, и функции не запрещено иметь параметры OUT (то есть возвращать несколько значений), однако это не считается хорошим стилем программирования.

# Исключительные ситуации, устанавливаемые в подпрограммах

При возникновении ошибки в подпрограмме устанавливается исключительная ситуация, которая может быть предварительно описана или определена пользователем. Если в процедуре отсутствует обработчик данной исключительной ситуации, управление программой сразу же передается из процедуры в вызывающую среду по правилам, установленным для передачи исключительных ситуаций (исключительные ситуации и правила их передачи подробно обсуждаются в главе 10). Однако в этом случае значения формальных параметров, имеющих вид OUT или IN OUT, не возвращаются фактическим параметрам. Фактические параметры будут иметь те же значения, которые они имели бы, если бы процедура не вызывалась. Предположим, что создается следующая процедура:

```
-- Этот пример является частью файла error.sql.
CREATE OR REPLACE PROCEDURE RaiseError (
  /* Иллюстрирует поведение необрабатываемых исключительных
     ситуаций и переменных OUT. Если значение p_Raise истинно,
     устанавливается флаг необрабатываемой ошибки. Если значение
     p_Raise ложно, процедура успешно завершается. */
  p_Raise IN BOOLEAN := TRUE,
  p_ParameterA OUT NUMBER) AS
BEGIN
  p_ParameterA := 7;
```

```
    IF p_Raise THEN
        /* Хотя параметру p_ParameterA присвоено значение 7, эта
           необрабатываемая исключительная ситуация вызывает немедленный
           возврат управления программой; при этом значение 7 не передается
           фактическому параметру, связанному с параметром p_ParameterA. */
        RAISE DUP_VAL_ON_INDEX;
    ELSE
        /* Возврат без ошибок. Фактическому параметру возвращается
            значение 7. */
        RETURN;
    END IF;
END RaiseError;
```

Если вызвать **RaiseError** в следующем блоке

```
-- Этот пример является частью файла error.sql.
DECLARE
  v_TempVar NUMBER := 1;
BEGIN
  INSERT INTO temp_table (num_col, char_col)
      VALUES (v_TempVar, 'Initial value');
  RaiseError(FALSE, v_TempVar);

  INSERT INTO temp_table (num_col, char_col)
      VALUES (v_TempVar, 'Value after successful call');

  v_TempVar := 2;
  INSERT INTO temp_table (num_col, char_col)
      VALUES (v_TempVar, 'Value before 2nd call');
  RaiseError(TRUE, v_TempVar);
EXCEPTION
  WHEN OTHERS THEN
      INSERT INTO temp_table (num_col, char_col)
        VALUES (v_TempVar, 'Value after unsuccessful call');
END;
```

и выбрать информацию в таблице **temp_table**, получим такие результаты:

```
SQL> SELECT * FROM temp_table;

   NUM_COL  CHAR_COL
---------- ---------------------------------
         1  Initial value
         7  Value after successful call
         2  Value before 2nd call
         2  Value after unsuccessful call
```

Перед первым вызовом **RaiseError** в переменной **v_TempVar** содержалось значение 1. Первый вызов был успешен, и **v_TempVar** присвоено значение 7. Затем, перед вторым вызовом **RaiseError**, значение **v_TempVar** изменилось на 2. Второй вызов завершился неуспешно, и переменная **v_TempVar** осталась равной 2 (а не изменилась снова на 7).

## Удаление процедур и функций

Процедуры и функции, как и таблицы, могут быть удалены. При выполнении этой операции процедура или функция удаляется из словаря данных. Синтаксис удаления процедуры выглядит следующим образом:

DROP PROCEDURE *имя_процедуры*;

а синтаксис удаления функции

DROP FUNCTION *имя_функции*;

где *имя_процедуры* — имя существующей процедуры, а *имя_функции* — имя существующей функции. Например, при помощи следующего оператора удаляется процедура **AddNewStudent**:

```
DROP PROCEDURE AddNewStudent;
```

Если удаляемый объект является функцией, то нужно использовать оператор DROP FUNCTION, а если процедурой, — DROP PROCEDURE. DROP — это команда DDL, поэтому перед оператором DROP и после него неявно выполняется оператор COMMIT.

# Размещение подпрограмм

Как показано в рассмотренных примерах этой главы, подпрограммы можно хранить в словаре данных. Подпрограмма сначала создается при помощи команды CREATE OR REPLACE, а затем вызывается из другого блока PL/SQL. Кроме того, подпрограмму можно описать в разделе объявлений блока; в этом случае она называется *локальной подпрограммой* (local subprogram).

## Хранимые подпрограммы и словарь данных

Когда подпрограмма создается при помощи команды CREATE OR REPLACE, она сохраняется в базе данных в скомпилированной форме, которая называется *p-кодом* (p-code). В p-коде содержатся все обработанные ссылки подпрограммы, а исходный текст преобразован в вид, удобный для чтения системой поддержки PL/SQL. При вызове подпрограммы p-код считывается с диска и выполняется. P-код аналогичен объектному коду, генерируемому компиляторами других языков третьего поколения. В p-коде содержатся обработанные ссылки на объекты (это свойство ранней привязки, рассмотренной в главе 4), поэтому выполнение p-кода является сравнительно недорогой (нересурсоемкой) операцией.

Информацию о подпрограмме можно получить при помощи различных представлений словаря данных. В представлении **user_objects** содержатся сведения обо всех объектах. Здесь можно узнать о том, когда объект был создан и последний раз модифицирован, каков тип объекта (таблица, последовательность, функция и т.д.), а также о достоверности объекта. В состав представления **user_source** входит *исходный программный текст объекта*, а в состав представления **user_errors** — *информация об ошибках* компиляции.

Рассмотрим процедуру:

```
CREATE OR REPLACE PROCEDURE Simple AS
  v_Counter NUMBER;
BEGIN
  v_Counter := 7;
END Simple;
```

После создания процедуры в представлении user_objects можно узнать о том, что она достоверна (VALID), а в представлении user_source — ее исходный текст. В user_errors нет сообщений об ошибках, поскольку наша процедура была успешно скомпилирована (см. рис. 7.1).

Если же изменить текст процедуры Simple таким образом, что возникает ошибка при ее компиляции (пропущена точка с запятой), например:

```
CREATE OR REPLACE PROCEDURE Simple AS
  v_Counter NUMBER;
BEGIN
  v_Counter := 7
END Simple;
```

и проанализировать те же три представления словаря данных (см. рис. 7.2), заметна разница. В **user_source** по-прежнему показан исходный текст процедуры, однако в **user_objects** ее состояние отображено уже как недостоверное (INVALID), а в **user_errors** можно видеть, что произошла ошибка компиляции PLS-103.

▼ СОВЕТУЕМ В SQL*Plus можно обращаться к ***user_errors*** и форматировать результаты в удобный для чтения вид при помощи команды SHOW ERRORS (показать ошибки). Использовать SHOW ERRORS следует при получении сообщения "Warning: Procedure created with compilation errors" ("Внимание! Процедура создана с ошибками компиляции". — *Прим. пер.*). Более подробно об этом рассказано в главе 13.

**Рис. 7.1.**
*Представления словаря данных после успешной компиляции*

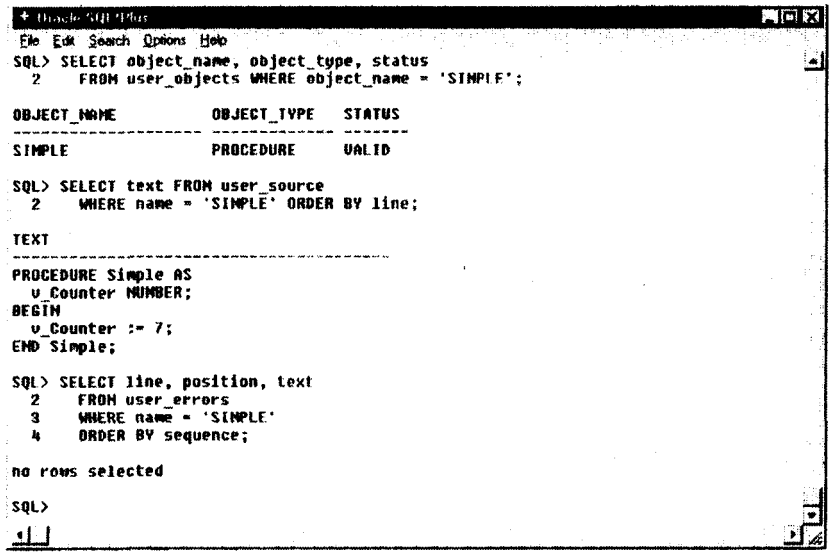
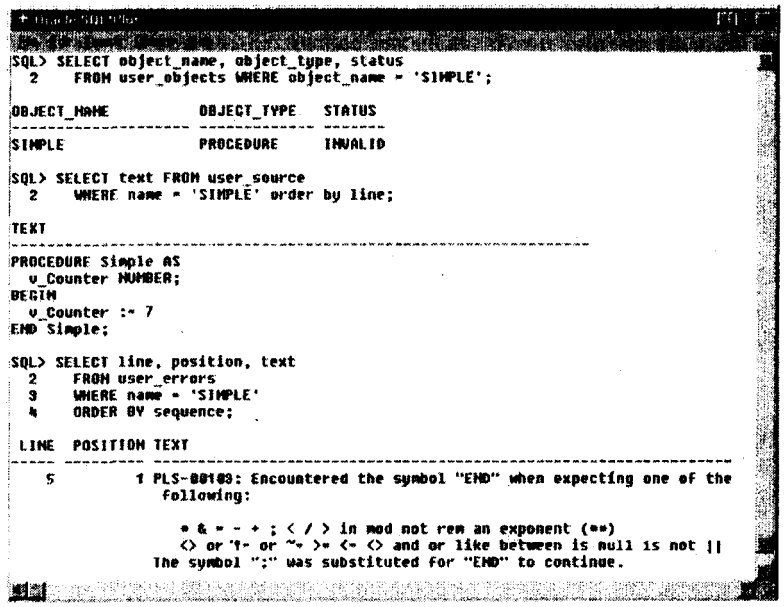



**Рис. 7.2.**
*Представления словаря данных после неуспешной компиляции*




Недостоверные хранимые подпрограммы, тем не менее, сохраняются в базе данных. Однако до устранения ошибок успешно вызывать их нельзя. При вызове недостоверной процедуры возникает следующая ошибка:

```
PLS-905: object is invalid
```
(объект недостоверен. — *Прим. пер.*)

Словарь данных более детально рассмотрен в приложении D.

## Локальные подпрограммы

Локальная подпрограмма, описываемая в разделе объявлений блока PL/SQL, приведена в следующем примере:

```
-- Этот пример содержится в файле local.sql.
DECLARE
```

```
    CURSOR c_AllStudents IS
        SELECT first_name, last_name
          FROM students;

    v_FormattedName VARCHAR2(50);

    /* Функция, возвращающая конкатенированные (связанные) имя и
       фамилию, разделенные пробелом. */
    FUNCTION FormatName(p_FirstName IN VARCHAR2,
                        p_LastName IN VARCHAR2)
        RETURN VARCHAR2 IS
    BEGIN
        RETURN p_FirstName || ' ' || p_LastName;
    END FormatName;

  -- Начало основного блока.
  BEGIN
    FOR v_StudentRecord IN c_AllStudents LOOP
        v_FormattedName :=
          FormatName(v_StudentRecord.first_name,
                     v_StudentRecord.last_name);
        INSERT INTO temp_table (char_col)
          VALUES (v_FormattedName);
    END LOOP;

    COMMIT;
  END;
```

Функция **FormatName** описана в разделе объявлений анонимного блока. Имя функции является идентификатором PL/SQL и поэтому подчиняется тем же самым правилам ограничения области действия и области видимости, что и другие идентификаторы PL/SQL. Говоря точнее, она видима только в том блоке, в котором описана, а область ее действия – от точки ее описания до конца блока. Из другого блока вызвать **FormatName** нельзя, поскольку она там невидима. Правила по определению области действия и области видимости см. в главе 2.

Любую локальную подпрограмму необходимо описывать в конце раздела объявлений. Если **FormatName** перенести выше описания курсора **c_AllStudents**, выдается сообщение об ошибке:

```
-- Этот пример содержится в файле local2.sql.
DECLARE
    /* Сначала объявим FormatName. Это приведет к ошибке компиляции,
       так как все другие объявления должны быть сделаны до локальных
       подпрограмм. */
    FUNCTION FormatName(p_FirstName IN VARCHAR2,
                        p_LastName IN VARCHAR2)
        RETURN VARCHAR2 IS
    BEGIN
        RETURN p_FirstName || ' ' || p_LastName;
    END FormatName;

    CURSOR c_AllStudents IS
        SELECT first_name, last_name
          FROM students;

    v_FormattedName VARCHAR2(50);
  -- Начало основного блока.
  BEGIN
    NULL;
```

```
END;
```

## Предварительное объявление

Имена локальных подпрограмм PL/SQL являются идентификаторами, поэтому они должны быть описаны до их использования. Обычно это не создает проблем, но когда подпрограммы ссылаются друг на друга, возникают трудности. Например:

```
-- Этот пример содержится в файле mutual.sql.
DECLARE
  v_TempVal  BINARY_INTEGER := 5;

  -- Локальная процедура A. Обратите внимание на то, что в тексте
     процедуры A вызывается процедура B.
  PROCEDURE A(p_Counter IN OUT BINARY_INTEGER) IS
  BEGIN
      IF p_Counter > 0 THEN
        B(p_Counter);
        p_Counter := p_Counter - 1;
      END IF;
  END A;

  -- Локальная процедура B. Обратите внимание на то, что в тексте
     процедуры B вызывается процедура A.
  PROCEDURE B(p_Counter IN OUT BINARY_INTEGER) IS
  BEGIN
      p_Counter := p_Counter - 1;
      A(p_Counter);
  END B;
BEGIN
  B(v_TempVal);
END;
```

Программу, приведенную в этом примере, скомпилировать невозможно. Процедура **A** вызывает процедуру **B**, поэтому **B** должна быть объявлена раньше **A**, чтобы можно было ссылаться на **B**. В свою очередь, процедура **B** вызывает процедуру **A**, поэтому **A** должна быть объявлена раньше **B**, чтобы можно было ссылаться на **A**. Одновременно эти условия невыполнимы. Для исправления сложившейся ситуации следует воспользоваться *предварительным объявлением* (forward declaration), являющимся просто именем процедуры с формальными параметрами. Это позволяет создавать процедуры, ссылающиеся друг на друга. Данный метод проиллюстрирован следующим примером:

```
-- Этот пример содержится в файле forward.sql.
DECLARE
  v_TempVal  BINARY_INTEGER := 5;

  -- Предварительное объявление процедуры B.
  PROCEDURE B(p_Counter IN OUT BINARY_INTEGER);

  PROCEDURE A(p_Counter IN OUT BINARY_INTEGER) IS
  BEGIN
      IF p_Counter > 0 THEN
        B(p_Counter);
        p_Counter := p_Counter - 1;
      END IF;
  END A;

  PROCEDURE B(p_Counter IN OUT BINARY_INTEGER) IS
  BEGIN
```

```
        p_Counter := p_Counter - 1;
        A(p_Counter);
    END B;
BEGIN
    B(v_TempVal);
END;
```

## Хранимые и локальные подпрограммы

Хранимые подпрограммы во многом отличаются от локальных. Автор предпочитает работать с хранимыми подпрограммами. При создании подпрограммы велика вероятность того, что придется вызывать ее из нескольких блоков, для чего подпрограмма должна храниться в базе данных. Размеры и сложность также оказывают влияние на выбор типа подпрограмм. Единственный вид процедур и функций, которые рекомендуется объявлять локально в блоке, — это короткие подпрограммы, вызывающиеся только из одного конкретного фрагмента программы (содержащего их блока). Различия между хранимыми и локальными подпрограммами обобщены в таблице 7.3.

### ТАБЛИЦА 7.3. Хранимые и локальные подпрограммы

| Хранимые подпрограммы | Локальные подпрограммы |
|---|---|
| Хранятся в базе данных в скомпилированном p-коде; при вызове процедуру не нужно компилировать. | Компилируются как фрагменты содержащих их блоков. Если блок выполняется несколько раз, подпрограмма должна компилироваться при каждом выполнении. |
| Могут вызываться из любого блока, запущенного на выполнение пользователем, который имеет привилегии EXECUTE для подпрограммы. | Могут вызываться только из содержащих их блоков. |
| Код подпрограммы хранится отдельно от вызывающего блока, поэтому вызывающий блок короче и легче для понимания. Кроме того, при желании с подпрограммой и вызывающим блоком можно работать по отдельности. | Подпрограмма и вызывающий блок находятся в одном месте, что может привести к путанице. Если изменение вносится в вызывающий блок, то подпрограмму необходимо компилировать по-новому. |
| Скомпилированный p-код можно закрепить в разделяемом пуле при помощи модульной процедуры DBMS_SHARED_POOL.KEEP[1]. Это повышает производительность системы. | Непосредственно локальные подпрограммы нельзя закреплять в разделяемом пуле. |

[1]Разделяемый пул и модуль DBMS_SHARED_POOL рассматриваются в главе 22.

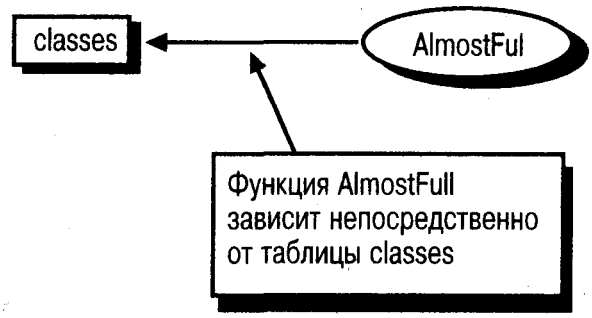
# Зависимости в подпрограммах

При компиляции процедуры или функции все объекты Oracle, на которые она ссылается, записываются в словарь данных. Говорят, что процедура *зависит* (depend) от этих объектов. Выше было показано, что подпрограмма, при компиляции которой возникли ошибки, помечается в словаре данных как недостоверная. Хранимая подпрограмма может стать недостоверной и в том случае, когда над одним из объектов, от которых она зависит, выполняется некоторая операция DDL. Проиллюстрируем это на примере. Функция **AlmostFull** (описанная выше в этой главе) обращается с запросом к таблице **classes.** Зависимости **AlmostFull** показаны на рис. 7.3. **AlmostFull** зависит только от одного объекта — **classes** (показано стрелкой).

Создадим процедуру, вызывающую **AlmostFull** и вносящую результаты в таблицу **temp_table**. Назовем эту процедуру **RecordFullClasses**:

```
-- Этот пример содержится в файле rfclass.sql.
CREATE OR REPLACE PROCEDURE RecordFullClasses AS
  CURSOR c_Classes IS
      SELECT department, course
        FROM classes;
BEGIN
```

**Рис. 7.3.**
*Зависимости функции AlmostFull*




```
    FOR v_ClassRecord IN c_Classes LOOP
        -- Запишем информацию обо всех группах, в которых
        -- осталось слишком мало места, в таблицу temp_table.
        IF AlmostFull(v_ClassRecord.department, v_ClassRecord.course)
        THEN
           INSERT INTO temp_table (char_col) VALUES
              (v_ClassRecord.department || ' ' || v_ClassRecord.course
              || ' is almost full!');
        END IF;
    END LOOP;
END RecordFullClasses;
```

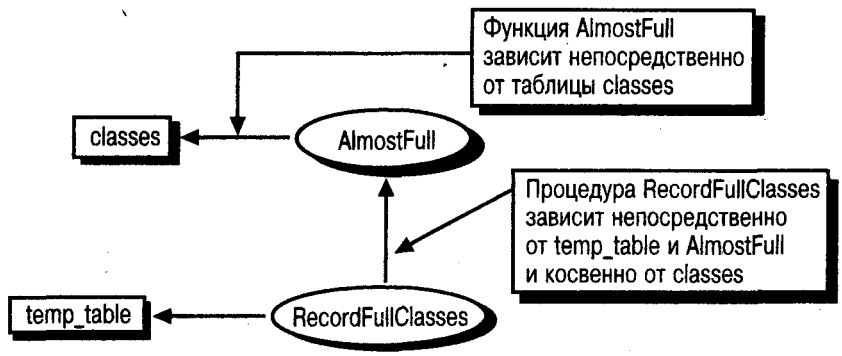
Существующие зависимости показаны стрелками на рис. 7.4. **RecordFullClasses** зависит как от **AlmostFull**, так и от **temp_table.** Такие зависимости называются *непосредственными* (direct), поскольку **RecordFullClasses** ссылается непосредственно на **AlmostFull** и **temp_table. AlmostFull** зависит от **classes**, поэтому **RecordFullClasses** имеет *косвенную* (indirect) зависимость от **classes.**

Если над **classes** выполняется операция DDL, то все объекты, зависящие от **classes** (непосредственно или косвенно), становятся недостоверными. Изменим таблицу classes, добавив к ней дополнительный столбец:

```
ALTER TABLE classes ADD (
student_rating NUMBER(2)); -- Уровень сложности, от 1 до 10.
```

В этом случае как AlmostFull, так и RecordFullClasses станут недостоверны, так как эти объекты зависят от classes. Вышесказанное проиллюстрировано на примере сеанса работы в SQL*Plus, приведенного на рис. 7.5.

**Рис. 7.4.**
*Зависимости процедуры RecordFullClasses*



## Определение зависимостей

Для определения момента анализа объектов производится сравнение временных меток, соответствующих последним модификациям этих объектов. *Временная метка* (timestamp) содержится в поле LAST_DDL_TIME представления **user_objects**. Этот метод хорош при сравнении двух объектов, принадлежащих одной и той же базе данных, однако возникают трудности, если объекты содержатся в разных системах поддержки, работающих с PL/SQL.

Предположим, имеется две процедуры, **P1** и **P2. P1** вызывает **P2** при помощи связи баз данных (рис. 7.6.). Эти две процедуры размещены в разных системах PL/SQL, в различных базах данных. Кро-

**Рис. 7.5.**
*В результате выполнения операции DDL объекты становятся недостоверными*

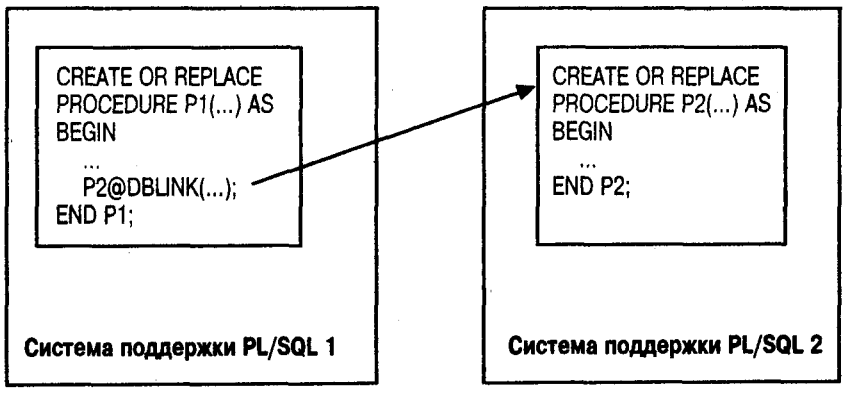
```
SQL> SELECT object_name, object_type, status
  2    FROM user_objects
  3    WHERE object_name IN ('ALMOSTFULL', 'RECORDFULLCLASSES');

OBJECT_NAME          OBJECT_TYPE   STATUS
-------------------- ------------- -------
RECORDFULLCLASSES    PROCEDURE     VALID
ALMOSTFULL           FUNCTION      VALID

SQL> ALTER TABLE classes ADD (
  2    student_rating NUMBER(2)  -- Difficulty rating from 1 to 10
  3  );

Table altered.

SQL> SELECT object_name, object_type, status
  2    FROM user_objects
  3    WHERE object_name IN ('ALMOSTFULL', 'RECORDFULLCLASSES');

OBJECT_NAME          OBJECT_TYPE   STATUS
-------------------- ------------- -------
RECORDFULLCLASSES    PROCEDURE     INVALID
ALMOSTFULL           FUNCTION      INVALID

SQL>
```

ме того, **P1** может быть размещена и на станции клиента, работающего с PL/SQL, в то время как **P2** находится на сервере (более подробно о клиентском и серверном PL/SQL рассказано в главе 13). В какой момент **P1** станет недостоверной, если выполнить операцию DDL над **P2**? Для определения этого момента существуют: метод временных меток и метод подписей.

## Модель временных меток

Модель временных меток работает так, будто **P1** и **P2** находятся в одной и той же системе PL/SQL. С изменением **P2** процедура **P1** становится недостоверной при следующем ее вызове. Однако в этом случае могут выполняться ненужные операции компиляции. Если изменяется только тело **P2**, а не ее описание, то перекомпилировать **P1** фактически не требуется, так как вызов

**Рис. 7.6.**
*Вызов удаленной процедуры*



```
P2(...) @DBLINK
```

в **P1** изменять не надо. Однако при использовании модели временных меток **P1** будет перекомпилироваться.

Проблема становится несколько серьезнее, если **P1** размещена в клиентской системе PL/SQL, например в Oracle Forms. В этом случае существует вероятность, что P1 не будет перекомпилирована, поскольку ее исходный код может отсутствовать в варианте Oracle Forms, применяемом во время выполнения программы.

## Модель подписей

**PL/SQL 2.3 ... и ВЫШЕ**

В PL/SQL 2.3 предлагается другой способ определения момента перекомпиляции удаленных зависимых объектов, называемый моделью подписей. При создании процедуры вместе с ней сохраняется *подпись*, или *сигнатура* (signature). Подпись кодирует описание процедуры, в том числе типы и порядок параметров. В этой модели подпись **P2** будет изменяться только, если изменяется описание данной процедуры. При компиляции **P1** используется подпись (а не временная метка) **P2**, поэтому при изменении подписи **P2** необходимо лишь перекомпилировать **P1**.

Для использования модели подписей необходимо установить параметр REMOTE_DEPENDENCIES_MODE в SIGNATURE, который содержится в инициализационном файле базы данных, называемом по умолчанию INIT.ORA. Кроме того, установить данный параметр в диалоговом режиме можно тремя способами:

1. Добавить в файл INIT.ORA строчку REMOTE_DEPENDENCIES_MODE=SIGNATURE. При следующем старте базы данных режим для всех сеансов будет установлен в SIGNATURE.
2. Выполнить команду:

   ```
   ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE = SIGNATURE;
   ```

   Это будет воздействовать на всю базу данных (на все сеансы), начиная с момента выполнения оператора. Для выдачи такой команды необходимо иметь системную привилегию ALTER SYSTEM.
3. Выполнить команду:

   ```
   ALTER SESSION SET REMOTE_DEPENDENCIES_MODE = SIGNATURE;
   ```

Это будет воздействовать только на определенный сеанс. Для объектов, создаваемых после этого момента в работе текущего сеанса, будет использоваться метод подписей.

Во всех рассмотренных вариантах можно вернуться к режиму, применяемому в версии 2.2, для чего вместо SIGNATURE указать TIMESTAMP. Режим TIMESTAMP задан по умолчанию, поэтому если REMOTE_DEPENDENCIES_MODE не изменять, система будет функционировать по-старому.

При использовании метода подписей следует учитывать ряд моментов:

- Если изменяются значения, заданные по умолчанию для формальных параметров, подписи не модифицируются. Предположим, что один из параметров процедуры **P2** имеет некоторое значение по умолчанию, а в **P1** это значение по умолчанию используется. Если изменить это значение в описании **P2**, по умолчанию **P1** не будет перекомпилироваться. Старое значение параметра, заданное по умолчанию, будет использоваться до тех пор, пока **P1** не будет перекомпилирована вручную. Это применимо только для параметров, имеющих вид IN.
- Если **P1** вызывает модульную процедуру **P2** и в удаленный модуль добавляется новый, переопределенный вариант **P2,** подпись не изменяется. **P1** будет использовать старый (не переопределенный) вариант до тех пор, пока **P1** не перекомпилировать вручную (модули и переопределение обсуждаются в следующей главе).
- Для того чтобы вручную перекомпилировать процедуру, используйте команду:

  ALTER PROCEDURE *имя_процедуры* COMPILE;

  где *имя_процедуры* — это имя компилируемой процедуры, а чтобы перекомпилировать функцию, — команду:

  ALTER FUNCTION *имя_функции* COMPILE;

За более подробной информацией о модели подписей обратитесь к руководству Oracle Server Application Developer's Guide версии 7.3 или более поздней.

# Привилегии и хранимые подпрограммы

Хранимые подпрограммы и модули являются объектами словаря данных, и поэтому их владельцы — это конкретные пользователи, или схемы, базы данных. Другие пользователи могут обращаться к таким объектам в случаях предоставления им необходимых привилегий. Привилегии и роли начинают действовать при создании хранимого объекта.

## Привилегия EXECUTE

Чтобы разрешить доступ к таблице, применяются объектные привилегии SELECT, INSERT, UPDATE и DELETE. Эти привилегии предоставляются пользователю или роли базы данных при помощи оператора GRANT (см. главу 4). Для хранимых подпрограмм и модулей необходима привилегия EXECUTE. Обратимся к процедуре **RecordFullClasses**, рассмотренной выше:

```
-- Этот пример содержится в файле rfclass.sql.
CREATE OR REPLACE PROCEDURE RecordFullClasses AS
  CURSOR c_Classes IS
      SELECT department, course
        FROM classes;
BEGIN
  FOR v_ClassRecord IN c_Classes LOOP
      -- Запишем информацию обо всех группах, в которых
      -- осталось слишком мало места, в таблицу temp_table.
      IF AlmostFull(v_ClassRecord.department, v_ClassRecord.course) THEN
        INSERT INTO temp_table (char_col) VALUES
          (v_ClassRecord.department || ' ' || v_ClassRecord.course
           || ' is almost full!');
      END IF;
  END LOOP;
END RecordFullClasses;
```

Предположим, что владельцем объектов, от которых зависит **RecordFullClasses** (функция **AlmostFull** и таблицы **classes** и **temp_table**), является пользователь **UserA**. Владельцем **RecordFullClasses** также является **UserA**. Если предоставить привилегию EXECUTE на **RecordFullClasses** другому пользователю базы данных, скажем, **UserB**:

```
GRANT EXECUTE ON RecordFullClasses TO UserB;
```

то **UserB** сможет выполнить **RecordFullClasses** при помощи следующего блока:

```
BEGIN
  UserA.RecordFullClasses;
END;
```

В этой ситуации владельцем всех объектов базы данных является пользователь **UserA**, что проиллюстрировано на рис. 7.7. Пунктирная линия на нем означает предоставление полномочия (оператор GRANT) пользователем **UserA** пользователю **UserB**, а непрерывная линия — зависимости, существующие между объектами.

Предположим, что **UserB** владеет еще одной таблицей, также называемой **temp_table** (рис. 7.8). Если **UserB** вызывает **UserA.RecordFullClasses**, таблица пользователя **UserA** будет модифицирована. Правило гласит:

- *Подпрограмма выполняется на основании набора привилегий ее владельца.*

Хотя **UserB** вызывает **RecordFullClasses**, этой процедурой владеет **UserA**. Поэтому идентификатор **temp_table** будет поставлен в соответствие таблице, принадлежащей пользователю **UserA**, а *не* **UserB**.

## Хранимые подпрограммы и роли

Теперь немного изменим ситуацию, отраженную на рис. 7.8. Предположим, что **UserA** не владеет ни **temp_table**, ни **RecordFullClasses**, а оба эти объекты принадлежат пользователю **UserB**. Кроме того, изменим **RecordFullClasses** так, чтобы эта процедура явно ссылалась на объекты пользователя **UserA**. Иллюстрация вышесказанного — на примере и на рис. 7.9.

**Рис. 7.7.**
*Пользователь UserA является владельцем всех объектов базы данных*

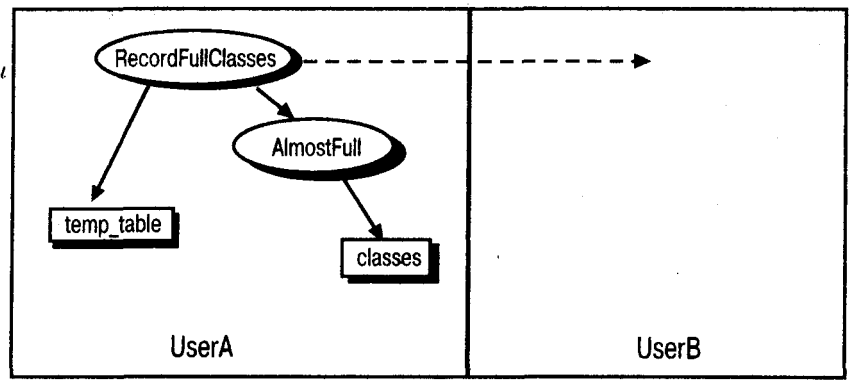
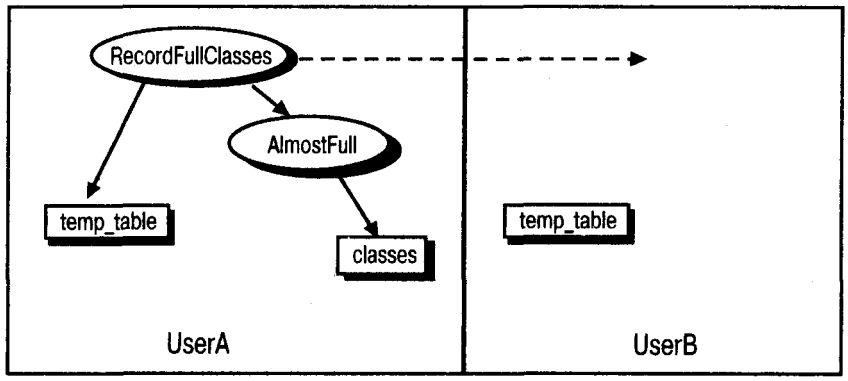


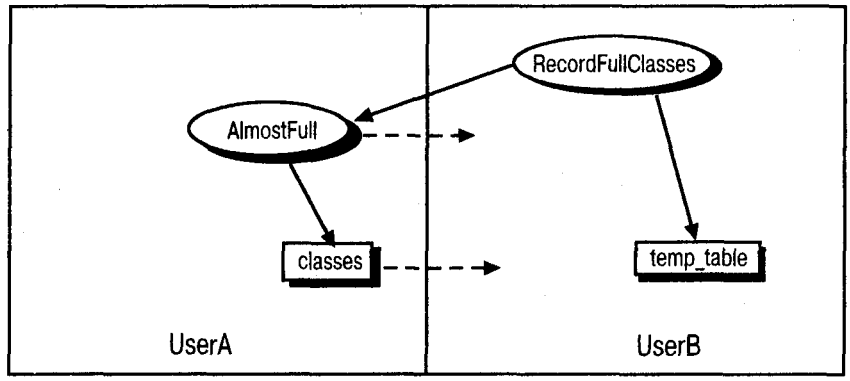

**Рис. 7.8.**
*Пользователи UserA и UserB являются владельцами таблиц temp_table*




```
CREATE OR REPLACE PROCEDURE RecordFullClasses AS
  CURSOR c_Classes IS
      SELECT department, course
        FROM UserA.classes;
BEGIN
  FOR v_ClassRecord IN c_Classes LOOP
      -- Запишем информацию обо всех группах, в которых
      -- осталось слишком мало места, в таблицу temp_table.
      IF UserA.AlmostFull(v_ClassRecord.department,
                          v_ClassRecord.course) THEN
        INSERT INTO temp_table (char_col) VALUES
          (v_ClassRecord.department || ' ' || v_ClassRecord.course
           || ' is almost full!');
      END IF;
  END LOOP;
END RecordFullClasses;
```

Для того чтобы процедура **RecordFullClasses** была корректно скомпилирована, пользователь **UserA** должен предоставить пользователю **UserB** привилегию SELECT на таблицу classes и привилегию EXECUTE на функцию **AlmostFull**. Это обозначают пунктирные линии на рис. 7.9. Более того, предоставить эти привилегии необходимо явно, а не посредством роли. Ниже приведены операторы GRANT, выполняемые пользователем UserA, которые обеспечивают успешную компиляцию **UserB.RecordFull Classes.**

**Рис. 7.9.**
*Пользователь UserB является владельцем процедуры RecordFullClasses*




```
GRANT SELECT ON classes TO UserB;
GRANT EXECUTE ON AlmostFull TO UserB;
```

Если же воспользоваться промежуточной ролью:

```
CREATE ROLE UserA_Role;
GRANT SELECT ON classes TO UserA_Role;
GRANT EXECUTE ON AlmostFull TO UserA_Role;
GRANT UserA_Role TO UserB;
```

то привилегии предоставлены не будут. Использование роли показано на рис. 7.10.

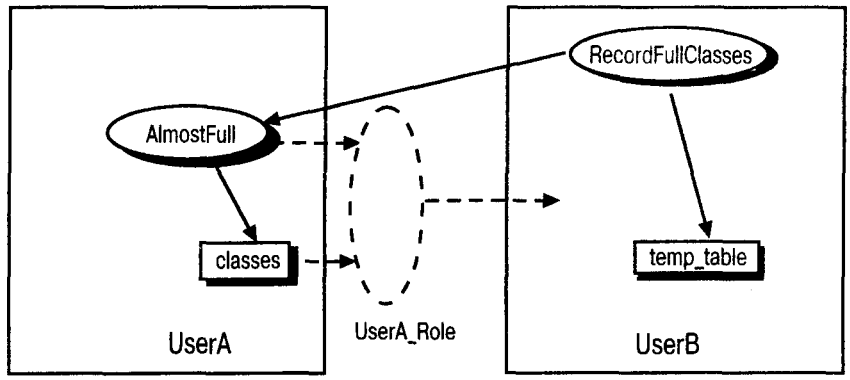
Итак, можно уточнить правило, сформулированное в предыдущем разделе:

- Подпрограмма выполняется на основании привилегий, предоставленных ее владельцу явно, а не посредством роли.

Если предоставить привилегии посредством роли, то при попытке компиляции **RecordFullClasses** будет выдано сообщение об ошибке PLS-201:

```
PLS-201: identifier 'CLASSES' must be declared
```

**Рис. 7.10.**
*Предоставление привилегий посредством роли*




```
PLS-201: identifier 'ALMOSTFULL' must be declared
```

(идентификатор '...' должен быть объявлен. — *Прим. пер.*)

Это правило распространяется также на триггеры и модули, хранящиеся в базе данных. Таким образом, в хранимых процедурах, функциях, модулях и триггерах *доступны только объекты, принадлежащие владельцам этих подпрограмм или пользователям, которым явно предоставлены привилегии на применение этих подпрограмм.*

Почему же установлено такое ограничение? Для ответа на этот вопрос необходимо вновь обратиться к процессу привязки. Напомним, что в PL/SQL используется ранняя привязка, – ссылки определяются во время компиляции, а не выполнения подпрограммы. Операторы GRANT и REVOKE являются операторами DDL. Они начинают действовать немедленно, и новые привилегии записываются в словарь данных. Во всех соединениях, установленных с базой данных, будет виден новый набор привилегий. Однако это не всегда справедливо для ролей. Ее можно предоставить пользователю, а этот пользователь затем вправе запретить роль при помощи команды SET ROLE. Отличие заключается в том, что команда SET ROLE действует только на один сеанс базы данных, а операторы GRANT и REVOKE применяются для всех сеансов. Роль может быть запрещена в одном сеансе, но разрешена в других.

Для использования привилегий, предоставленных посредством роли в хранимых подпрограммах и триггерах, эти привилегии нужно проверять всякий раз при запуске процедуры. Проверка привилегий является этапом процесса привязки. Но ранняя привязка означает то, что привилегии проверяются во время компиляции, а не выполнения. *Чтобы выполнить раннюю привязку, необходимо запретить все роли внутри хранимых процедур и триггеров.*

# Итоги

Подпрограммы необходимы для разработки программ на PL/SQL. В этой главе были обсуждены различия между процедурами и функциями, в том числе способы их создания и вызова. Кроме того, рассказано о том, как управлять зависимостями, установленными между объектами PL/SQL, и какое влияние оказывают эти зависимости на процесс разработки подпрограмм. Глава завершается обсуждением ролей и процедур. Далее будет показано, как объединять процедуры и функции PL/SQL вместе с таким внутренними объектами, как переменные, типы и курсоры, в программные модули.

# Глава 8

## *Модули*

Модули, наряду с процедурами и функциями, являются третьим видом именованных блоков PL/SQL. Это очень полезное средство, расширяющее возможности языка программирования PL/SQL. В этой главе вначале будет рассмотрен синтаксис создания модулей, а затем обсуждены некоторые преимущества их использования.

# Модули

*Модуль* (package) – еще одно средство, пришедшее в PL/SQL из языка программирования Ada. Модуль – это конструкция PL/SQL, позволяющая хранить связанные объекты в одном месте. Модуль состоит из двух различных частей: описания и тела, каждая из которых хранится по отдельности в словаре данных. В отличие от процедур и функций, которые содержатся локально в блоке или хранятся в базе данных, модули могут быть только хранимыми и никогда локальными. Модули позволяют объединять связанные объекты, а также используют менее ограничений, определяемых зависимостями. Кроме того, они имеют ряд свойств, повышающих производительность системы (см. главу 22).

В сущности, модуль представляет собой именованный раздел объявлений. Все входящее в состав раздела объявлений блока, может входить и в модуль: процедуры, функции, курсоры, типы и переменные. Размещение их в модуле позволяет ссылаться на них из других блоков PL/SQL, поэтому в модулях можно описывать глобальные переменные для PL/SQL.

## Описание модуля

В описании модуля (package specification),называемом также *заголовком модуля* (package header), содержится информация о составе модуля, однако в описание не входит текст процедур. Рассмотрим пример:

```
-- Этот пример является частью файла clpack.sql.
CREATE OR REPLACE PACKAGE ClassPackage AS
  -- Добавляет нового студента в указанную группу.
  PROCEDURE AddStudent( p_StudentID  IN students.id%TYPE,
                        P_Department IN classes.department%TYPE,
                        P_Course     IN classes.course%TYPE);

  -- Удаляет указанного студента из указанной группы.
  PROCEDURE RemoveStudent(p_StudentID  IN students.id%TYPE,
                          P_Department IN classes.department%TYPE,
                          P_Course     IN classes.course%TYPE);

  -- Исключительная ситуация, устанавливаемая процедурой RemoveStudent.
  e_StudentNotRegistered EXCEPTION;

  -- Табличный тип, используемый для хранения информации о студентах.
  TYPE t_StudentIDTable IS TABLE OF students.id%TYPE
      INDEX BY BINARY_INTEGER;

  -- Возвращает таблицу PL/SQL со сведениями о студентах,
  -- включенных в указанную группу в настоящий момент.
  PROCEDURE ClassList(p_Department IN classes.department%TYPE,
                      p_Course IN classes.course%TYPE,
                      p_IDs OUT t_StudentIDTable,
                      p_NumStudents IN OUT BINARY_INTEGER);
END ClassPackage;
```

В модуле **ClassPackage** содержатся описания трех процедур, типа и исключительной ситуации (исключительные ситуации используются для обработки ошибок PL/SQL и подробнее обсуждаются в главе 10). Общий синтаксис создания заголовка модуля выглядит следующим образом:

```
CREATE [OR REPLACE] PACKAGE имя_модуля {IS | AS}
описание_процедуры |
описание_функции |
объявление_переменной |
```

определение_типа |
объявление_исключительной_ситуации |
объявление_курсора
END [*имя_модуля*];

где *имя_модуля* — это имя модуля. *Элементы* модуля (описания процедур и функций, переменные и т.д.) аналогичны указанным в разделе объявлений анонимного блока. Для заголовка модуля верны те же синтаксические правила, установленные для раздела объявлений, за исключением объявлений процедуры и функции. Перечислим эти правила:

- Элементы модуля могут указываться в любом порядке. Однако, как и в разделе объявлений, объект должен быть объявлен до того, как на него будут произведены ссылки. Например, если частью условия WHERE курсора является некоторая переменная, то она должна быть объявлена до объявления курсора.
- Присутствие элементов всех видов совсем не обязательно. Например, модуль может состоять только из описаний процедур и функций, без объявления исключительных ситуаций или типов.
- Объявления всех процедур и функций должны быть предварительными. В этом отличие модуля от раздела объявлений блока, где могут находиться как предварительные объявления, так и реальный текст процедур и функций. Программный текст процедур и функций модуля содержится в теле этого модуля.

## Тело модуля

*Тело модуля* (package body) — это объект словаря данных, хранящийся отдельно от заголовка модуля. Тело модуля нельзя успешно скомпилировать без успешной компиляции заголовка. В теле содержится текст подпрограмм, предварительно объявленных в заголовке модуля. Тело модуля **ClassPackage** показано в следующем примере:

```
-- Этот пример является частью файла clpack.sql.
CREATE OR REPLACE PACKAGE BODY ClassPackage AS
  -- Добавляет нового студента в указанную группу.
  PROCEDURE AddStudent(p_StudentID IN students.id%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course IN classes.course%TYPE) IS
  BEGIN
      INSERT INTO registered_students (student_id, department, course)
        VALUES (p_StudentID, p_Department, p_Course);
      COMMIT;
  END AddStudent;

  -- Удаляет указанного студента из указанной группы.
  PROCEDURE RemoveStudent(p_StudentID IN students.id%TYPE,
                          p_Department IN classes.department%TYPE,
                          p_Course IN classes.course%TYPE) IS
  BEGIN
      DELETE FROM registered_students
        WHERE student_id = p_StudentID
        AND department = p_Department
        AND course = p_Course;
      -- Проверим, успешно ли была выполнена операция DELETE. Если
      -- указанные строки не найдены, устанавливается флаг ошибки.
      IF SQL%NOTFOUND THEN
        RAISE e_StudentNotRegistered;
      END IF;

      COMMIT;
  END RemoveStudent;
```

```
    -- Возвращает таблицу PL/SQL со сведениями о студентах,
    -- включенных в указанную группу в настоящий момент.
    PROCEDURE ClassList(p_Department   IN classes.department%TYPE,
                        P_Course       IN classes.course%TYPE,
                        P_IDs          OUT t_StudentIDTable,
                        P_NumStudents  IN OUT BINARY_INTEGER) IS

        v_StudentID registered_students.student_id%TYPE;

        -- Локальный курсор для выбора зарегистрированных студентов.
        CURSOR c_RegisteredStudents IS
          SELECT student_id
            FROM registered_students
            WHERE department = p_Department
            AND course = p_Course;

    BEGIN
        /* p_NumStudents будет индексом таблицы. Этот индекс будет
           начинаться с 0 и увеличиваться с каждым повторением цикла выборки.
           В конце цикла в индексе будет содержаться число считанных
           строк, то есть число строк, возвращенных в p_IDs. */
        p_NumStudents := 0;

        OPEN c_RegisteredStudents;
        LOOP
          FETCH c_RegisteredStudents INTO v_StudentID;
          EXIT WHEN c_RegisteredStudents%NOTFOUND;

          p_NumStudents := p_NumStudents + 1;
          p_IDs(p_NumStudents) := v_StudentID;
        END LOOP;
    END ClassList;
END ClassPackage;
```

Текст подпрограмм, предварительно объявленных в заголовке модуля, содержится в теле этого модуля. На объекты в заголовке, не объявленные предварительно (например **e_StudentNotRegistered**), можно ссылаться в теле модуля без повторного объявления.

Тело модуля не является обязательной его частью. Если в заголовке не указаны какие-либо процедуры или функции (а только переменные, курсоры, типы и т.д.), тело можно не создавать. Такой способ целесообразен при объявлении глобальных переменных, поскольку все объекты модуля видимы вне его пределов (область действия и область видимости элементов модуля обсуждаются в следующем разделе).

Любое предварительное объявление в заголовке модуля должно быть раскрыто в его теле. Описание процедуры или функции должно быть таким же и включать в свой состав имя подпрограммы, имена ее параметров и вид каждого параметра. Например, приведенный ниже заголовок не соответствует телу модуля, так как в теле используется список параметров функции **FunctionA**, отличный от списка параметров, указанного в заголовке.

```
CREATE OR REPLACE PACKAGE PackageA AS
  FUNCTION FunctionA(p_Parameter1 IN NUMBER,
                     p_Parameter2 IN DATE)
    RETURN VARCHAR2;
END PackageA;

CREATE OR REPLACE PACKAGE BODY PackageA AS
  FUNCTION FunctionA(p_Parameter1 IN CHAR)
    RETURN VARCHAR2;
END PackageA;
```

Если попытаться создать **PackageA** так, как показано выше, то выдаются сообщения об ошибках:

```
PLS-00328: A subprogram body must be defined for the forward
declaration of FUNCTIONA.
```

(тело подпрограммы должно быть описано так, как это сделано в предварительном объявлении FUNCTIONA. — *Прим. пер.*)

```
PLS-00323: subprogram or cursor 'FUNCTIONA' is declared in a
package specification and must be defined in the package body.
```

(подпрограмма (курсор) объявлена в описании модуля и должна быть описана в теле модуля. — *Прим. пер.*)

## Модули и области действия

Любой объект, объявленный в заголовке модуля, находится в области действия и видим вне границ этого модуля. Для обращения к объекту нужно указать имя модуля при ссылке на этот объект. Например, можно вызвать процедуру **ClassPackage.RemoveStudent** из следующего блока PL/SQL:

```
BEGIN
ClassPackage.RemoveStudent(10006, 'HIS', 101);
END;
```

При этом вызов процедуры аналогичен вызову процедуры, не включенной в модуль. Единственное отличие такого вызова — присутствие перед именем процедуры имени модуля. Для модульных процедур могут задаваться параметры по умолчанию, и вызывать такие процедуры можно при помощи как позиционного, так и именного представления, то есть точно так же, как и обычные хранимые процедуры.

Кроме того, в модуле можно применять типы данных, определяемые пользователями. Например, для вызова **ClassList** необходимо объявить переменную типа **ClassPackage.t_StudentIDTable**:

```
-- Этот пример содержится в файле cllist.sql.
DECLARE
  v_HistoryStudents ClassPackage.t_StudentIDTable;
  v_NumStudents BINARY_INTEGER := 20;
BEGIN
  -- Заполним таблицу PL/SQL первыми 20 студентами группы History 101.
  ClassPackage.ClassList('HIS', 101, v_HistoryStudents,
                         V_NumStudents);

  -- Внесем этих студентов в таблицу temp_table.
  FOR v_LoopCounter IN 1..v_NumStudents LOOP
      INSERT INTO temp_table (num_col, char_col)
        VALUES (v_HistoryStudents(v_LoopCounter),
                'In History 101');
  END LOOP;
END;
```

Внутри тела модуля можно ссылаться без указания имени модуля на объекты, представленные в его заголовке. Например, процедура **RemoveStudent** может ссылаться на исключительную ситуацию просто как на **e_StudentNotRegistered**, а не как на **ClassPackage.e_StudentNotRegistered**. Однако можно воспользоваться как полным, так и уточненным именем.

## Переопределение модульных подпрограмм

Процедуры и функции внутри модуля могут быть *переопределены* (overloaded). Это означает, что может существовать несколько процедур или функций с одним и тем же именем, но разными параметрами. Это очень удобно, так как позволяет выполнять одну и ту же операцию над объектами различных типов. Предположим, нужно внести некоторого студента в состав одной из групп, указав либо идентификатор этого студента, либо его имя и фамилию. Это можно сделать, изменив **ClassPackage** следующим образом:

```
-- Этот пример содержится в файле overload.sql.
CREATE OR REPLACE PACKAGE ClassPackage AS
  -- Добавляет нового студента в указанную группу.
  PROCEDURE AddStudent( p_StudentID  IN students.id%TYPE,
                        P_Department IN classes.department%TYPE,
                        P_Course     IN classes.course%TYPE);

  -- Также добавляет нового студента, но указанием не
  -- идентификатора этого студента, а его имени и фамилии.
  PROCEDURE AddStudent(p_FirstName   IN students.first_name%TYPE,
                       P_LastName    IN students.last_name%TYPE,
                       P_Department  IN classes.department%TYPE,
                       P_Course      IN classes.course%TYPE);
  ...
END ClassPackage;

CREATE OR REPLACE PACKAGE BODY ClassPackage AS
  -- Добавляет нового студента в указанную группу.
  PROCEDURE AddStudent(p_StudentID   IN students.id%TYPE,
                       P_Department  IN classes.department%TYPE,
                       P_Course      IN classes.course%TYPE) IS
  BEGIN
      INSERT INTO registered_students (student_id, department, course)
        VALUES (p_StudentID, p_Department, p_Course);
      COMMIT;
  END AddStudent;

  -- Добавляет нового студента не по идентификатору, а по имени.
  PROCEDURE AddStudent(p_FirstName   IN students.first_name%TYPE,
                       P_LastName    IN students.last_name%TYPE,
                       P_Department  IN classes.department%TYPE,
                       P_Course      IN classes.course%TYPE) IS
      v_StudentID students.id%TYPE;
  BEGIN
      /* Сначала узнаем нужный идентификатор в таблице students. */
      SELECT ID
        INTO v_StudentID
        FROM students
        WHERE first_name = p_FirstName
        AND last_name = p_LastName;
      -- Теперь добавим студента по его идентификатору.
      INSERT INTO registered_students (student_id, department, course)
        VALUES (p_StudentID, p_Department, p_Course);
      COMMIT;
  END AddStudent;
  ...
END ClassPackage;
```

Теперь можно внести студента в группу Music 410:

```
BEGIN
  ClassPackage.AddStudent(10000, 'MUS', 410);
END;
```

или:

```
BEGIN
  ClassPackage.AddStudent('Barbara', 'Blues', 'MUS', 410);
END;
```

Переопределение полезно тогда, когда одна и та же операция может быть выполнена над аргументами разных типов. Однако на переопределение налагается ряд ограничений:

1. Нельзя переопределять две подпрограммы, если их параметры отличаются только именами или видами. Например:

```
PROCEDURE OverloadMe(p_TheParameter IN NUMBER);
PROCEDURE OverloadMe(p_TheParameter OUT NUMBER);
```

2. Нельзя переопределять две функции, отличающиеся лишь типами возвращаемых ими данных. Например:

```
FUNCTION OverloadMeToo RETURN DATE;
FUNCTION OverloadMeToo RETURN NUMBER;
```

3. Наконец, типы параметров переопределяемых функций должны принадлежать различным семействам типов (о семействах типов рассказано в главе 2). Например, типы CHAR и VARCHAR2 входят в одно и то же семейство, поэтому нельзя переопределить следующие процедуры:

```
PROCEDURE OverloadChar(p_TheParameter IN CHAR);
PROCEDURE OverloadChar(p_TheParameter IN VARCHAR2);
```

▼ ВНИМАНИЕ

На самом деле компилятор PL/SQL разрешает создавать модули, в которые входят подпрограммы, нарушающие приведенные выше ограничения. Однако во время выполнения программы указанные ссылки не смогут быть реализованы, и всегда будет выдаваться сообщение об ошибке *PLS-307: too many declarations of 'подпрограмма' match this call* (этому вызову соответствует слишком много объявлений 'подпрограмма'. — *Прим. пер.*).

## Инициализация модуля

При вызове первый раз модуль *конкретизируется* (instantiated). Это значит, что модуль считывается с диска в память, а затем запускается p-код. В этот момент для всех переменных, описанных в модуле, выделяется память. У каждого сеанса будет собственная копия модульных переменных; это гарантирует, что два сеанса, выполняющие подпрограммы одного и того же модуля, будут использовать различные области памяти.

Во многих случаях код инициализации нужно запускать на выполнение при первой конкретизации модуля. Это можно сделать, если к телу модуля добавить раздел инициализации, разместив его после всех объектов:

```
CREATE OR REPLACE PACKAGE BODY имя_модуля {IS | AS}
  ...
BEGIN
  код_инициализации;
END [имя_модуля];
```

где *имя_модуля* — это имя модуля, а *код_инициализации* — запускаемый код. Ниже приведен пример модуля, с помощью которого реализуется функция случайных чисел.

```
-- Этот пример содержится в файле random.sql.
CREATE OR REPLACE PACKAGE Random AS
/* Генератор случайных чисел. Используется тот же алгоритм, что и в
   функции rand() в языке C. */

  -- Используется для изменения исходного значения (seed). Начиная
  -- с одного и того же исходного значения, будут генерироваться
  -- одинаковые последовательности случайных чисел.
  PROCEDURE ChangeSeed(p_NewSeed IN NUMBER);
```

```
    -- Возвращает случайное целое число в диапазоне от 1 до 32767.
    FUNCTION Rand RETURN NUMBER;

    -- Аналогична функции Rand, но с процедурным интерфейсом.
    PROCEDURE GetRand(p_RandomNumber OUT NUMBER);

    -- Возвращает случайное целое число в диапазоне от 1 до p_MaxVal.
    FUNCTION RandMax(p_MaxVal IN NUMBER) RETURN NUMBER;

    -- Аналогична функции RandMax, но с процедурным интерфейсом.
    PROCEDURE GetRandMax(p_RandomNumber OUT NUMBER,
                         p_MaxVal IN NUMBER);

END Random;

CREATE OR REPLACE PACKAGE BODY Random AS

    /* Используется для вычисления следующего числа. */
    V_Multiplier  CONSTANT NUMBER := 22695477;
    V_Increment   CONSTANT NUMBER := 1;

    /* Исходное значение, используемое для генерирования
       последовательности случайных чисел. */
    V_Seed   number := 1;

    PROCEDURE ChangeSeed(p_NewSeed IN NUMBER) IS
    BEGIN
        v_Seed := p_NewSeed;
    END ChangeSeed;

    FUNCTION Rand RETURN NUMBER IS
    BEGIN
        v_Seed := MOD(v_Multiplier * v_Seed + v_Increment,
                      (2 ** 32));
        RETURN BITAND(v_Seed/(2 ** 16), 32767);
    END Rand;

    PROCEDURE GetRand(p_RandomNumber OUT NUMBER) IS
    BEGIN
        -- Просто вызовем Rand и возвратим значение.
        p_RandomNumber := Rand;
    END GetRand;

    FUNCTION RandMax(p_MaxVal IN NUMBER) RETURN NUMBER IS
    BEGIN
        RETURN MOD(Rand, p_MaxVal) + 1;
    END RandMax;

    PROCEDURE GetRandMax(p_RandomNumber OUT NUMBER,
                         p_MaxVal IN NUMBER) IS
    BEGIN
        -- Просто вызовем RandMax и возвратим значение.
        p_RandomNumber := RandMax(p_MaxVal);
    END GetRandMax;
```

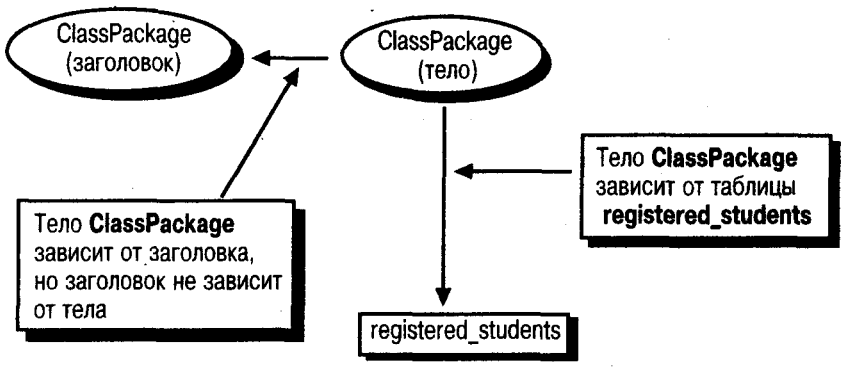
```
BEGIN
   /* Инициализация модуля. Инициализируем исходное значение
      текущим временем в секундах. */
   ChangeSeed(TO_NUMBER(TO_CHAR(SYSDATE, 'SSSSS')));
END Random;
```

Для получения случайного числа можно просто вызвать **Random.Rand.** Последовательность случайных чисел зависит от исходного значения — для одного и того же исходного значения генерируются одинаковые последовательности. Поэтому для получения более случайных значений необходимо инициализировать исходное значение различными значениями всякий раз при подтверждении модуля. Для этого из раздела инициализации модуля вызывается процедура **ChangeSeed.** В модуле **Debug** (см. главу 14) также имеется раздел инициализации.

## Модули и зависимости

Картина зависимостей, существующих в модуле **ClassPackage**, показана на рис. 8.1. Тело модуля зависит от таблицы **registered_students** и от заголовка модуля, не зависящего ни от чего. В этом заключается преимущество модулей. Можно изменить тело модуля, не изменяя его заголовок. При этом другие объекты, зависящие от заголовка, не нужно перекомпилировать, так как они никогда не станут недостоверными. При изменении заголовка тело модуля автоматически становится недостоверным, поскольку оно зависит от заголовка.

**Рис. 8.1.**
*Зависимости в модуле ClassPackage*




Иллюстрацией вышесказанного служит пример сеанса работы в SQL*Plus:

```
-- Этот пример содержится в файле depend.sql.
SQL> -- Сначала создадим простую таблицу.
SQL> CREATE TABLE simple_table (f1 NUMBER);

Table created.

SQL>
SQL> -- Теперь создадим модульную процедуру, ссылающуюся на таблицу.
SQL> CREATE OR REPLACE PACKAGE Dependee AS
  2    PROCEDURE Example(p_Val IN NUMBER);
  3  END Dependee;
  4  /

Package created.

SQL>
SQL> CREATE OR REPLACE PACKAGE BODY Dependee AS
  2    PROCEDURE Example(p_Val IN NUMBER) IS
  3    BEGIN
```

```
  4        INSERT INTO simple_table VALUES (p_Val);
  5     END Example;
  6  END Dependee;
  7  /

Package body created.

SQL>
SQL>  -- Теперь создадим процедуру, ссылающуюся на модуль Dependee.
SQL>  CREATE OR REPLACE PROCEDURE Depender(p_Val IN NUMBER) AS
  2   BEGIN
  3      Dependee.Example(p_Val + 1);
  4   END Depender;
  5   /

Procedure created.

SQL>
SQL>  -- Обратимся к user_objects и посмотрим, все ли объекты
SQL>  -- достоверны.
SQL>  SELECT object_name, object_type, status
  2     FROM user_objects
  3     WHERE object_name IN ('DEPENDER', 'DEPENDEE', 'SIMPLE_TABLE');

OBJECT_NAME                    OBJECT_TYPE     STATUS
-------------------------   ------------     --------
SIMPLE_TABLE                   TABLE           VALID
DEPENDEE                       PACKAGE         VALID
DEPENDEE                       PACKAGE BODY    VALID
DEPENDER                       PROCEDURE       VALID

SQL>
SQL>  -- Изменим лишь тело модуля. Заметьте, что заголовок не
SQL>  -- изменяется.
SQL>  CREATE OR REPLACE PACKAGE BODY Dependee AS
  2     PROCEDURE Example(p_Val IN NUMBER) IS
  3     BEGIN
  4        INSERT INTO simple_table VALUES (p_Val - 1);
  5     END Example;
  6  END Dependee;
  7  /

Package body created.

SQL>
SQL>  -- Из user_objects видно, что процедура Depender по-прежнему
SQL>  -- достоверна.
SQL>  SELECT object_name, object_type, status
  2     FROM user_objects
  3     WHERE object_name IN ('DEPENDER', 'DEPENDEE', 'SIMPLE_TABLE');

OBJECT_NAME                    OBJECT_TYPE     STATUS
-------------------------   ------------ --------
SIMPLE_TABLE                   TABLE           VALID
DEPENDEE                       PACKAGE         VALID
```

```
DEPENDEE                        PACKAGE BODY    VALID
DEPENDER                        PROCEDURE       VALID

SQL>
SQL>  -- Хотя таблица удаляется, недостоверным становится только
SQL>  -- тело модуля.
SQL>  DROP TABLE simple_table;

Table dropped.

SQL>  SELECT object_name, object_type, status
  2     FROM user_objects
  3     WHERE object_name IN ('DEPENDER', 'DEPENDEE', 'SIMPLE_TABLE');
OBJECT_NAME                     OBJECT_TYPE     STATUS
------------------------------  -------------   ---------
DEPENDEE                        PACKAGE         VALID
DEPENDEE                        PACKAGE BODY    INVALID
DEPENDER                        PROCEDURE       VALID
```

PL/SQL 8.0 ... и ВЫШЕ

***Примечание.*** В Oracle8 можно использовать представления словаря данных **user_dependencies, all_dependencies** и **dba_dependencies**. В представлениях отображаются все взаимосвязи между объектами схем. Более подробно об этих представлениях см. в главе 11 и приложении D.

# Использование хранимых функций в SQL-операторах

PL/SQL 2.1 ... и ВЫШЕ

В главе 4 были рассмотрены различия между процедурными и SQL-операторами. Вызовы подпрограмм по сути своей процедурны, поэтому их нельзя выполнять в SQL-операторах. Однако в PL/SQL 2.1 для хранимых функций эти ограничения отменены. Если обычная или модульная функция отвечает определенным требованиям, ее можно вызывать во время выполнения SQL-оператора. Эта возможность доступна в PL/SQL версии 2.1 (Oracle7 7.1) и выше.

Функция, созданная пользователем, вызывается точно так же, как и встроенные функции, рассмотренные в главе 4 (например TO_CHAR, UPPER или ADD_MONTHS). В зависимости от области применения функции она должна отвечать конкретным требованиям. Эти требования определяются в терминах так называемых уровней строгости.

## Уровни строгости

Существует четыре различных уровня строгости (см. таблицу 8.1). *Уровень строгости* (purity level) определяет структуры данных, которые может считывать или модифицировать функция.

В зависимости от уровня строгости функция отвечает следующим требованиям:

- Любая функция, вызываемая из SQL-оператора, не может модифицировать таблицы базы данных (WNDS).
- Для того чтобы функция могла быть выполнена удаленно (через связь баз данных) или параллельно, она не должна читать или записывать значения модульных переменных (RNPS или WNPS).
- Функции, вызываемые из команд SELECT, VALUES или SET, могут записывать модульные переменные. Во всех других командах функции должны иметь уровень строгости WNPS.
- *Функция строга настолько, насколько строги вызываемые ею подпрограммы.* Если функция вызывает хранимую процедуру, которая выполняет, к примеру, обновление информации (оператор UPDATE), то эта функция не имеет уровня строгости WNDS и, следовательно, не может быть использована в SQL-операторе.

- Независимо от уровня строгости, хранимые функции PL/SQL нельзя использовать в ограничении CHECK команды CREATE TABLE или ALTER TABLE, а также использовать для указания значения по умолчанию для столбца, так как в этих ситуациях требуется, чтобы описания не изменялись.

**ТАБЛИЦА 8.1. Уровни строгости функций**

| Уровень строгости | Значение | Описание |
|---|---|---|
| WNDS | Write no database state (не записывать состояние базы данных) | Функция не модифицирует таблицы базы данных (при помощи операторов DML) |
| RNDS | Read no database state (не читать состояние базы данных) | Функция не читает таблицы базы данных (при помощи оператора SELECT) |
| WNPS | Write no package state (не записывать состояние модуля) | Функция не модифицирует модульные переменные (модульные переменные не используются в левой части операции присваивания и в операторе FETCH) |
| RNPS | Read no package state (не читать состояние модуля) | Функция не анализирует модульные переменные (модульные переменные не используются в правой части операции присваивания и в процедурных или SQL-выражениях) |

Кроме приведенных ограничений, функция, созданная пользователем, должна отвечать также дополнительным требованиям, чтобы ее можно было вызывать из SQL-операторов. Все встроенные функции тоже отвечают этим требованиям.

- Функция должна храниться в базе данных или отдельно, или быть частью модуля. Она не должна быть локальной по отношению к другому блоку.
- Функция может принимать только параметры IN, но не IN OUT или OUT.
- Для формальных параметров должны использоваться только те типы, которые применяются в базе данных, но не типы PL/SQL, такие как BOOLEAN или RECORD. Типы базы данных – это NUMBER, CHAR, VARCHAR2, ROWID, LONG, LONG RAW и DATE.
- *Тип, возвращаемый функцией, также должен быть типом базы данных.*

В качестве примера рассмотрим функцию **FullName**, входным параметром которой является идентификатор студента и которая возвращает конкатенированные имя и фамилию.

```
-- Этот пример содержится в файле fullname.sql.
CREATE OR REPLACE FUNCTION FullName (
  p_StudentID students.ID%TYPE)
  RETURN VARCHAR2 IS

  v_Result VARCHAR2(100);
BEGIN
  SELECT first_name || ' ' || last_name
      INTO v_Result
      FROM students
      WHERE ID = p_StudentID;

  RETURN v_Result;
END FullName;
```

Функция FullName удовлетворяет всем ограничениям, поэтому ее можно вызвать из SQL-оператора:

```
SQL> SELECT ID, FullName(ID) "Full Name"
  2      FROM students;
```

```
     ID   Full Name
  -----   ---------------------------------
  10000   Scott Smith
  10001   Margaret Mason
  10002   Joanne Junebug
  10003   Manish Murgratroid
  10004   Patrick Poll
  10005   Timothy Taller
  10006   Barbara Blues
  10007   David Dinsmore
  10008   Ester Elegant
  10009   Rose Riznit
  10010   Rita Razmataz

11 rows selected.
SQL> INSERT INTO temp_table (char_col)
  2     VALUES (FullName(10010));

1 row created.
```

## Прагма RESTRICT_REFERENCES

В PL/SQL можно устанавливать уровни строгости для обычных, автономных функций. При вызове функции из SQL-оператора уровень строгости проверяется. Если он не удовлетворяет предъявляемым требованиям, возвращается сообщение об ошибке. Однако для модульных функций необходима прагма RESTRICT_REFERENCES (ограничить ссылки). Эта прагма устанавливает уровень строгости для конкретной функции:

PRAGMA RESTRICT_REFERENCES(*имя_функции*,
WNDS[, WNPS] [, RNDS] [, RNPS]);

где *имя_функции* – имя модульной функции. Поскольку уровень WNDS обязателен для всех функций, вызываемых в SQL-операторах, он обязательно указывается в прагме. Другие уровни строгости можно указывать в любом порядке. Прагма помещается в заголовок модуля вместе с описанием функции. К примеру, в модуле **StudentOps** прагма RESTRICT_REFERENCES используется дважды:

```
-- Этот пример содержится в файле studops.sql.
CREATE OR REPLACE PACKAGE StudentOps AS
  FUNCTION FullName(p_StudentID IN students.ID%TYPE)
     RETURN VARCHAR2;
  PRAGMA RESTRICT_REFERENCES(FullName, WNDS, WNPS, RNPS);
  /* Возвращает число студентов, профилирующей дисциплиной которых
     является история. */
  FUNCTION NumHistoryMajors
     RETURN NUMBER;
  PRAGMA RESTRICT_REFERENCES(NumHistoryMajors, WNDS, WNPS, RNPS);
END StudentOps;

CREATE OR REPLACE PACKAGE BODY StudentOps AS

  -- Модульная переменная для хранения числа студентов-историков.
  v_NumHist NUMBER;

  FUNCTION FullName(p_StudentID IN students.ID%TYPE)
     RETURN VARCHAR2 IS
     v_Result VARCHAR2(100);
  BEGIN
     SELECT first_name || ' ' || last_name
        INTO v_Result
```

```
        FROM students
        WHERE ID = p_StudentID;
    RETURN v_Result;
END FullName;

FUNCTION NumHistoryMajors RETURN NUMBER IS
    v_Result NUMBER;
BEGIN
    IF v_NumHist IS NULL THEN
      /* Определим ответ */
      SELECT COUNT(*)
        INTO v_Result
        FROM students
        WHERE major = 'History';
      /* и сохраним его для будущего использования. */
      v_NumHist := v_Result;
    ELSE
      v_Result := v_NumHist;
    END IF;

    RETURN v_Result;
  END NumHistoryMajors;
END StudentOps;
```

▼ ВНИМАНИЕ Уровни строгости WNPS и RNPS применяются к переменным других модулей. Функция **NumHistoryMajors** модифицирует модульную переменную **v_NumHist**, но **v_NumHist** находится в том же модуле **StudentOps**, поэтому можно устанавливать для **NumHistoryMajors** уровни строгости WNPS и RNPS.

**Обоснование использования RESTRICT_REFERENCES.** Почему прагма обязательна для модульной функции, но не нужна для автономной? Ответить на этот вопрос можно, проанализировав взаимосвязи, между заголовком и телом модуля. Напомним, что блоки PL/SQL, вызывающие модульную функцию, зависят только от заголовка модуля, но не от его тела. Более того, при создании вызывающего блока тело модуля может вообще не существовать. Поэтому компилятору PL/SQL необходимо указание, помогающее определить уровни строгости модульной функции, чтобы проверить корректность использования функции в вызывающем блоке. Всякий раз при последующей модификации (или при первом создании) тела модуля код функции проверяется на соответствие заданной прагме.

**Раздел инициализации** В тексте раздела инициализации модуля также можно указывать уровни строгости. При вызове любой из функций модуля первый раз выполняется раздел инициализации. Таким образом, модульная функция строга настолько, насколько строг раздел инициализации. Уровни строгости для модуля устанавливаются при помощи прагмы RESTRICT_REFERENCES, но вместо имени функции указывается имя модуля:

```
CREATE OR REPLACE PACKAGE StudentOps AS
  PRAGMA RESTRICT_REFERENCES (StudentOps, WNDS, WNPS, RNPS);
  ...
END StudentOps;
```

**Переопределенные функции** Прагма RESTRICT_REFERENCES может располагаться в любом месте описания модуля после объявления функции, но относиться она может только к одной функции. Поэтому в случае с переопределенными функциями прагма относится к функции, чье описание предшествует этой прагме и расположено к ней ближе всего. В следующем примере прагма относится ко второму объявлению функции F:

```
CREATE OR REPLACE PACKAGE TestPackage AS
  FUNCTION F(p_ParameterOne IN NUMBER) RETURN VARCHAR2;
  FUNCTION F RETURN DATE;
```

```
    PRAGMA RESTRICT_REFERENCES (F, WNDS, RNDS);
END TestPackage;
```

**Встроенные модули** Модули, поставляемые в составе PL/SQL, не являются строгими. В состав этих модулей входят DBMS_OUTPUT, DBMS_PIPE, DBMS_ALERT, DBMS_SQL и UTL_FILE. В результате любая функция, которая использует эти модули, также не является строгой и поэтому не может быть использована в SQL-операторах. Эти модули подробно обсуждаются в последующих главах.

## Параметры по умолчанию

При вызове функции из процедурного оператора можно использовать для формальных параметров значения по умолчанию. Однако при вызове функции из SQL-оператора все параметры должны быть указаны. Кроме того, требуется применять позиционное представление и запрещается применять именное представление. Ниже приведен неверный вызов **FullName**:

```
SELECT FullName(p_StudentID => 10000) FROM dual;
```

# PL/SQL в работе: экспортер схем PL/SQL

PL/SQL 2.3 ... и ВЫШЕ

Исходный программный текст объектов PL/SQL, хранимых в базе данных, находится в словаре данных. Можно считать этот текст в представлениях словаря данных, а затем воспользоваться модулем UTL_FILE и записать текст вместе с нужными операторами CREATE OR REPLACE в файл. В результате получится файл, содержащий все операторы, необходимые для воссоздания объектов PL/SQL. Затем этот файл может быть выполнен из SQL*Plus.

В модуле **Export** используются как модуль UTL_FILE (см. главу 18), так и курсорные переменные (см. главу 6), поэтому для его функционирования требуется PL/SQL версии 2.3 и выше.

- Для создания выходного файла в **Export** используется модуль UTL_FILE. Поэтому владелец модуля **Export** должен иметь полномочие EXECUTE на модуль UTL_FILE. Кроме того, необходимо параметр UTL_FILE_DIR в файле init.ora установить надлежащим образом. Более подробно о параметре UTL_FILE_DIR и о его использовании рассказано в главе 18.
- Для определения исходного текста в **Export** используются представления словаря данных all_objects, all_source и all_triggers. Эти представления отображают информацию об объектах, на которые имеет привилегии EXECUTE владелец **Export**. Таким образом, если владелец Export не может выполнить объект PL/SQL, то он не сможет и экспортировать этот объект.
- Если для шифрования исходного текста объекта была использована оболочка PL/SQL, то в словаре данных будет доступен только зашифрованный текст. Именно такой текст модуль **Export** запишет в файл.

В **Export** содержатся две процедуры: **OneObj** и **AllObj. OneObj** применяется для экспортирования текста только одного объекта, а **AllObj** – для экспортирования всех объектов, принадлежащих конкретному владельцу конкретного типа (или всех типов). Ниже даны текст модуля **Export**, а также примеры его последующего использования:

```
-- Этот пример содержится в файле export.sql.
CREATE OR REPLACE PACKAGE Export AS
  /* Экспортирует один объект. Параметры используются следующим
   * образом:
   * P_Schema:      Указывает владельца экспортируемого объекта.
   * P_ObjType:     Указывает тип объекта. Возможные значения:
   *                'PACKAGE', 'PACKAGE BODY', 'PROCEDURE',
   *                'FUNCTION' и 'TRIGGER'. Если NULL, то тип
   *                объекта определяется выбором из all_objects.
   * P_BothTypes:   Если истинен, когда p_ObjType = 'PACKAGE', то
   *                экспортироваться будут как модуль, так и его тело.
   * P_FileDir:     Каталог, где создается выходной файл.
   * P_FileName:    Имя выходного файла.
   * P_Mode:        Вид (либо 'A' – добавление, либо 'W' – запись)
```

```
    *                выходного файла.
    */
   PROCEDURE OneObj(p_Schema IN VARCHAR2,
                    p_ObjName IN VARCHAR2,
                    p_ObjType IN VARCHAR2 DEFAULT NULL,
                    p_BothTypes IN BOOLEAN DEFAULT TRUE,
                    p_FileDir IN VARCHAR2,
                    p_FileName IN VARCHAR2,
                    p_Mode IN VARCHAR2);
   /* Экспортирует все объекты данного типа. Параметры используются
    * следующим образом:
    * P_Schema:      Указывает владельца экспортируемого объекта.
    * P_ObjType:     Указывает тип объекта. Возможные значения:
    *                'PACKAGE', 'PACKAGE BODY', 'PROCEDURE',
    *                'FUNCTION' и 'TRIGGER'. Если тип указан, то
    *                будут экспортироваться все объекты данного типа,
    *                которыми владеет p_Schema. Если NULL, то
    *                экспортироваться будут все объекты, которыми
    *                владеет p_Schema.
    * P_FileDir:     Каталог, где создается выходной файл.
    * P_FileName:    Имя выходного файла.
    * P_Mode:        Вид (либо 'A' - добавление, либо 'W' - запись)
    *                выходного файла.
    */
   PROCEDURE AllObjs(p_Schema IN VARCHAR2,
                     p_ObjType IN VARCHAR2 DEFAULT NULL,
                     p_FileDir IN VARCHAR2,
                     p_FileName IN VARCHAR2,
                     p_Mode IN VARCHAR2);
END Export;

CREATE OR REPLACE PACKAGE BODY Export AS
   /* Это основная рабочая процедура модуля. OutputObj будет
    * выдавать один объект, указанный параметрами p_Schema,
    * p_ObjName и p_ObjType, в файл, указанный параметром
    * p_FileHandle. Файл предварительно должен быть открыт на
    * запись (в режиме 'W' или 'A').
   */
   PROCEDURE OutputObj(p_FileHandle IN OUT UTL_FILE.FILE_TYPE,
                       p_Schema IN VARCHAR2,
                       p_ObjName IN VARCHAR2,
                       p_ObjType IN VARCHAR2) IS

    /* Эти переменные используются для считывания текста триггера.
     * Поскольку в all_triggers тело триггера хранится в виде
     * данных LONG, для выбора порций данных (chunks) типа LONG
     * нужно использовать модуль DBMS_SQL. */
    v_SQLStmt VARCHAR2(200) :=
        'SELECT description, trigger_body
           FROM all_triggers
           WHERE owner = :v_owner
           AND trigger_name = :v_name';
    V_Cursor    INTEGER;
    V_NumRows   INTEGER;
```

```
    V_Dummy          INTEGER;
    V_Description    all_triggers.description%TYPE;
    V_BodyChunk      VARCHAR2(100);
    V_ChunkSize      NUMBER := 100;
    V_CurPos         NUMBER := 0;
    V_ReturnedLength NUMBER := 0;

  /* Эти переменные используются для считывания исходного текста
   * объектов других видов. Здесь DBMS_SQL не нужен, так как в
   * all_source каждая строка (line) исходного текста хранится
   * отдельно от других.
   */
  v_TextLine all_source.text%TYPE;
   CURSOR c_ObjCur IS
       SELECT text
          FROM all_source
          WHERE owner = p_Schema
          AND name = p_ObjName
          AND type = p_ObjType
          ORDER BY line;
BEGIN
  -- Сначала запишем 'CREATE OR REPLACE ' в файл.
  UTL_FILE.PUT(p_FileHandle, 'CREATE OR REPLACE ');

  IF (p_ObjType = 'TRIGGER') THEN
      BEGIN
         -- Запишем в файл тип объекта (в нашем случае TRIGGER).
         UTL_FILE.PUT(p_FileHandle, 'TRIGGER ');

         -- Откроем курсор и проведем синтаксический анализ оператора.
         v_Cursor := DBMS_SQL.OPEN_CURSOR;
         DBMS_SQL.PARSE(v_Cursor, v_SQLStmt, DBMS_SQL.V7);

         -- Выполним привязку входных переменных к местам их хранения.
         DBMS_SQL.BIND_VARIABLE(v_Cursor, ':v_owner', p_Schema);
         DBMS_SQL.BIND_VARIABLE(v_Cursor, ':v_name', p_Objname);

         -- Опишем выходные переменные. Обратите внимание на
         -- использование DEFINE_COLUMN_LONG для текста тела триггера.
         DBMS_SQL.DEFINE_COLUMN(v_Cursor, 1, v_Description, 2000);
         DBMS_SQL.DEFINE_COLUMN_LONG(v_Cursor, 2);

         -- Выполним оператор и считаем строку (row). Производить
         -- считывание в цикл не нужно, так как на триггер
         -- приходится только одна строка.
         v_Dummy := DBMS_SQL.EXECUTE(v_Cursor);
         v_NumRows := DBMS_SQL.FETCH_ROWS(v_Cursor);

         -- Считаем значение, соответствующее описанию триггера, и
         -- поместим его в файл. Заметьте, что используется не
         -- UTL_FILE.PUT_LINE, а UTL_FILE.PUT, так как
         -- в таблице содержится конечный символ новой строки (newline).
         DBMS_SQL.COLUMN_VALUE(v_Cursor, 1, v_Description);

         UTL_FILE.PUT(p_FileHandle, v_Description);
```

```
      -- Будем выполнять цикл до тех пор, пока не считаем все тело
      -- триггера. Будем считывать символы v_ChunkSize при каждом
      -- повторении цикла.
      LOOP
        DBMS_SQL.COLUMN_VALUE_LONG(v_Cursor, 2, v_ChunkSize,
                                   v_CurPos, v_BodyChunk,
                                   V_ReturnedLength);

        IF v_ReturnedLength < v_ChunkSize THEN
          -- Считана последняя порция данных. В силу определенных
          -- причин в таблице после символа новой строки хранится
          -- дополнительный символ NULL. Поэтому сначала нужно
          -- удалить этот дополнительный символ.
          v_BodyChunk := SUBSTR(v_BodyChunk, 1, LENGTH(v_BodyChunk) - 1);

          -- Запишем усеченную порцию данных и выйдем из цикла.
          UTL_FILE.PUT(p_FileHandle, v_BodyChunk);
          EXIT;
        ELSE
          -- Считана порция данных из середины данных LONG.
          -- Запишем ее в файл и обновим текущую позицию для
          -- следующего повторения цикла.
          UTL_FILE.PUT(p_FileHandle, v_BodyChunk);
          v_CurPos := v_CurPos + v_ReturnedLength;
        END IF;
      END LOOP;

      -- Закроем курсор, так как его обработка закончена.
      DBMS_SQL.CLOSE_CURSOR(v_Cursor);
    EXCEPTION
      WHEN OTHERS THEN
        -- В случае ошибки сначала закроем курсор, а затем
        -- повторно установим исключительную ситуацию для
        -- обработки ее в вызывающей среде.
        DBMS_SQL.CLOSE_CURSOR(v_Cursor);
        RAISE;
      END;
    ELSE
      -- Если достигнуто это место программы, то триггер не
      -- записывается. Поэтому можно просто циклически выбрать
      -- каждую строку исходного текста из all_source. Обратите
      -- внимание на то, что в первой строке будет содержаться тип и
      -- имя объекта, а также 'IS' или 'AS'.
    OPEN c_ObjCur;
    LOOP
      FETCH c_ObjCur INTO v_TextLine;
      EXIT WHEN c_ObjCur%NOTFOUND;

      -- В каждой строке уже имеется конечный символ новой строки,
      -- поэтому можно использовать UTL_FILE.PUT, а не UTL_FILE.PUT_LINE.
      UTL_FILE.PUT(p_FileHandle, v_TextLine);
    END LOOP;
    CLOSE c_ObjCur;
  END IF;
```

```
    -- Запишем конечный символ '/'.
    UTL_FILE.PUT_LINE(p_FileHandle, '/');
END OutputObj;

PROCEDURE OneObj(p_Schema IN VARCHAR2,
                 p_ObjName IN VARCHAR2,
                 p_ObjType IN VARCHAR2 DEFAULT NULL,
                 p_BothTypes IN BOOLEAN DEFAULT TRUE,
                 p_FileDir IN VARCHAR2,
                 p_FileName IN VARCHAR2,
                 p_Mode IN VARCHAR2) IS
    v_FileHandle UTL_FILE.FILE_TYPE;
    v_ObjType all_objects.object_type%TYPE;
BEGIN
    -- Проверим правильность входных параметров.
    IF p_BothTypes AND (p_ObjType != 'PACKAGE') THEN
      RAISE_APPLICATION_ERROR(-20000,
        'Export.OneObj: BothTypes set but type != PACKAGE');
    ELSIF p_ObjType IS NOT NULL AND p_ObjType NOT IN
      ('PACKAGE', 'PACKAGE BODY', 'PROCEDURE', 'FUNCTION',
      'TRIGGER') THEN RAISE_APPLICATION_ERROR(-20001,
      'Export.OneObj: Illegal value ' || p_ObjType ||
      ' for object type');
    ELSIF p_FileDir IS NULL OR p_FileName IS NULL or p_Mode IS
NULL THEN
      RAISE_APPLICATION_ERROR(-20002,
        'Export.OneObj: Directory, Filename and Mode must be
        non-NULL');
    ELSIF p_Mode NOT IN ('A', 'a', 'W', 'w') THEN
      RAISE_APPLICATION_ERROR(-20003,
        'Export.OneObj: Mode ' || p_Mode || ' not ''A'' or ''W''');
    END IF;

    -- Определим правильный тип объекта и подтвердим наличие
    -- этого объекта.
    BEGIN
      IF p_ObjType IS NULL THEN
        -- Тип объекта не указан - проверим его наличие без
        -- указания типа.
        SELECT object_type
          INTO v_ObjType
          FROM all_objects
          WHERE owner = UPPER(p_Schema)
          AND object_name = UPPER(p_Objname)
          AND object_type IN ('PROCEDURE', 'FUNCTION', 'PACKAGE',
                              'PACKAGE BODY', 'TRIGGER');
      ELSIF p_BothTypes THEN
        -- Указан BothTypes - сначала проверим наличие заголовка
        -- модуля.
      SELECT object_type
        INTO v_ObjType
        FROM all_objects
        WHERE owner = UPPER(p_Schema)
```

```
            AND object_name = UPPER(p_Objname)
            AND object_type = 'PACKAGE';

        -- Теперь проверим наличие тела модуля.

        BEGIN
            SELECT object_type
              INTO v_ObjType
              FROM all_objects
              WHERE owner = UPPER(p_Schema)
              AND object_name = UPPER(p_Objname)
              AND object_type = 'PACKAGE BODY';
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
              RAISE_APPLICATION_ERROR(-20006,
                'Export.ObjObj: BothTypes set but package body ' ||
                p_Schema || '.' || p_Objname || ' not found');
        END;
    ELSE
        -- Указан тип объекта, но Bothtypes не указан - проверим
        -- существование объекта при помощи его типа.
        SELECT object_type
          INTO v_ObjType
          FROM all_objects
          WHERE owner = UPPER(p_Schema)
          AND object_name = UPPER(p_Objname)
          AND object_type = p_ObjType;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR(-20004,
        'Export.OneObj: Object ' || p_Schema || '.' || p_Objname ||
        ' not found');
    WHEN TOO_MANY_ROWS THEN
      RAISE_APPLICATION_ERROR(-20005,
        'Export.OneObj: More than one match for ' || p_Schema || '.' ||
        p_Objname);
END;

-- Если достигнуто это место программы, то объект существует,
-- поэтому можно открывать файл и записывать в него информацию.
-- Если установлен режим 'A', то сначала запишем 2 пустых строки.
v_FileHandle := UTL_FILE.FOPEN(p_FileDir, p_FileName, p_Mode);
IF p_Mode IN ('A', 'a') THEN
    UTL_FILE.NEW_LINE(v_FileHandle, 2);
END IF;

-- Запишем информацию об объекте.
IF p_ObjType = 'PACKAGE' AND p_BothTypes THEN
    OutputObj(v_FileHandle, p_Schema, p_ObjName, 'PACKAGE');
    UTL_FILE.NEW_LINE(v_FileHandle, 2);
    OutputObj(v_FileHandle, p_Schema, p_ObjName, 'PACKAGE BODY');
ELSE
    OutputObj(v_FileHandle, p_Schema, p_ObjName, v_ObjType);
```

```
    END IF;

    -- Закроем выходной файл.
    UTL_FILE.FCLOSE(v_FileHandle);
  EXCEPTION
    -- Обработаем исключительные ситуации UTL_FILE и убедимся в том,
    -- что файл закрыт надлежащим образом.
    WHEN UTL_FILE.INVALID_PATH THEN
        UTL_FILE.FCLOSE(v_FileHandle);
        RAISE_APPLICATION_ERROR(-20010,
                                'Export.OneObj: Invalid Path');
    WHEN UTL_FILE.INVALID_OPERATION THEN
        UTL_FILE.FCLOSE(v_FileHandle);
        RAISE_APPLICATION_ERROR(-20010,
                                'Export.OneObj: Invalid Operation');
    WHEN UTL_FILE.INVALID_FILEHANDLE THEN
        UTL_FILE.FCLOSE(v_FileHandle);
        RAISE_APPLICATION_ERROR(-20010,
                                'Export.OneObj: Invalid File Handle');
    WHEN UTL_FILE.WRITE_ERROR THEN
        UTL_FILE.FCLOSE(v_FileHandle);
        RAISE_APPLICATION_ERROR(-20010,
                                'Export.OneObj: Write Error');
    WHEN UTL_FILE.INTERNAL_ERROR THEN
        UTL_FILE.FCLOSE(v_FileHandle);
        RAISE_APPLICATION_ERROR(-20010,
                                'Export.OneObj: Internal Error');
    WHEN OTHERS THEN
        UTL_FILE.FCLOSE(v_FileHandle);
        RAISE;
  END OneObj;

  PROCEDURE AllObjs(p_Schema IN VARCHAR2,
                    p_ObjType IN VARCHAR2 DEFAULT NULL,
                    p_FileDir IN VARCHAR2,
                    p_FileName IN VARCHAR2,
                    p_Mode IN VARCHAR2) IS
    v_FileHandle UTL_FILE.FILE_TYPE;
    v_ObjName all_objects.object_name%TYPE;
    v_ObjType all_objects.object_type%TYPE;
    v_ObjectFound BOOLEAN := FALSE;

    -- Поскольку запрос может быть двух видов, воспользуемся для
    -- него курсорной переменной.
    TYPE t_AllObjs IS REF CURSOR;
    c_AllObjsCur t_AllObjs;

  BEGIN
    -- Проверим правильность входных параметров.
    IF p_ObjType IS NOT NULL AND p_ObjType NOT IN
          ('PACKAGE', 'PACKAGE BODY', 'PROCEDURE', 'FUNCTION',
          'TRIGGER') THEN
        RAISE_APPLICATION_ERROR(-20001,
                                'Export.AllObjs: Illegal value ' || p_ObjType ||
```

```
                                    ' for object type');
ELSIF p_FileDir IS NULL OR p_FileName IS NULL or p_Mode IS NULL THEN
    RAISE_APPLICATION_ERROR(-20002,
        'Export.AllObjs: Directory, Filename and Mode must be non-NULL');
ELSIF p_Mode NOT IN ('A', 'a', 'W', 'w') THEN

    RAISE_APPLICATION_ERROR(-20003,
        'Export.AllObjs: Mode ' || p_Mode || ' not ''A'' or ''W''');
END IF;

-- Если тип объекта не указан, откроем курсор и запросим все
-- объекты, принадлежащие схеме p_Schema. Если тип указан, то откроем
-- курсор и запросим объекты только этого типа.
IF p_ObjType IS NULL THEN
    OPEN c_AllObjsCur FOR
      SELECT object_name, object_type
        FROM all_objects
        WHERE owner = UPPER(p_Schema)
        AND object_type in ('PACKAGE', 'PACKAGE BODY',
                            'PROCEDURE', 'FUNCTION', 'TRIGGER');
ELSE
    OPEN c_AllObjsCur FOR
      SELECT object_name, object_type
        FROM all_objects
        WHERE owner = UPPER(p_Schema)
        AND object_type = p_ObjType;
END IF;

-- Последовательно выберем все объекты, удовлетворяющие критериям
-- выбора, и каждый из них запишем, разделив 2 пустыми строками.
LOOP
    FETCH c_AllObjsCur INTO v_ObjName, v_ObjType;
    EXIT WHEN c_AllObjsCur%NOTFOUND;

    IF NOT v_ObjectFound THEN
      -- Найден, по меньшей мере, один объект, соответствующий
      -- входным параметрам. Откроем файл и, если задан режим 'A',
      -- сначала запишем 2 пустые строки.
      v_ObjectFound := TRUE;
      v_FileHandle := UTL_FILE.FOPEN(p_FileDir, p_FileName, p_Mode);
      IF p_Mode IN ('A', 'a') THEN
        UTL_FILE.NEW_LINE(v_FileHandle, 2);
    END IF;
END IF;
    OutputObj(v_FileHandle, p_Schema, v_ObjName, v_ObjType);
    UTL_FILE.NEW_LINE(v_FileHandle, 2);
END LOOP;

-- Проверим, найден ли объект, а затем закроем курсор и файл.
CLOSE c_AllObjsCur;
IF NOT v_ObjectFound THEN
    RAISE_APPLICATION_ERROR(-20004,
                            'Export.AllObjs: No objects found');
END IF;
```

```
    UTL_FILE.FCLOSE(v_FileHandle);
  EXCEPTION
    -- Обработаем исключительные ситуации UTL_FILE и убедимся в том,
    -- что файл закрыт надлежащим образом.
    WHEN UTL_FILE.INVALID_PATH THEN
        UTL_FILE.FCLOSE(v_FileHandle);
        RAISE_APPLICATION_ERROR(-20010,
                                'Export.AllObjs: Invalid Path');
    WHEN UTL_FILE.INVALID_OPERATION THEN
        UTL_FILE.FCLOSE(v_FileHandle);
        RAISE_APPLICATION_ERROR(-20010,
                                'Export.AllObjs: Invalid Operation');
    WHEN UTL_FILE.INVALID_FILEHANDLE THEN
        UTL_FILE.FCLOSE(v_FileHandle);
        RAISE_APPLICATION_ERROR(-20010,
                                'Export.AllObjs: Invalid File Handle');
    WHEN UTL_FILE.WRITE_ERROR THEN
        UTL_FILE.FCLOSE(v_FileHandle);
        RAISE_APPLICATION_ERROR(-20010,
                                'Export.AllObjs: Write Error');
    WHEN OTHERS THEN
        UTL_FILE.FCLOSE(v_FileHandle);
        RAISE;
    END AllObjs;
  END Export;
```

Для примера можно воспользоваться процедурой **Export.OneObj**, чтобы экспортировать процедуру **AddNewStudent** (созданную в главе 7), выполнив следующий блок:

```
BEGIN
  Export.OneObj('EXAMPLE', 'ADDNEWSTUDENT', 'PROCEDURE', FALSE,
                'c:\temp', 'addstud.sql', 'W');
END;
```

При этом создается файл c:\temp\addstud.sql, в котором содержится следующее:

```
CREATE OR REPLACE PROCEDURE AddNewStudent (
  P_FirstName    students.first_name%TYPE,
  P_LastName     students.last_name%TYPE,
  P_Major        students.major%TYPE) AS
BEGIN
  -- Внесем новую строку в таблицу students. Воспользуемся
  -- последовательностью student_sequence для того, чтобы
  -- сгенерировать идентификатор для нового студента и 0 для
  -- current_credits.
  INSERT INTO students (ID, first_name, last_name,
                        major, current_credits)
      VALUES (student_sequence.nextval, p_FirstName, p_LastName,
              p_Major, 0);
  COMMIT;
END AddNewStudent;
/
```

Впоследствии этот файл может быть выполнен непосредственно из SQL*Plus для повторного создания указанной процедуры.

# Итоги

В этой главе подробно проанализированы программные модули, их синтаксис и назначение. Рассказано о том, какие преимущества имеют модули в отношении зависимостей, существующих между объектами. В следующей главе будет обсуждаться последний тип именованных блоков PL/SQL – триггеры.

# Глава 9

## Триггеры

Четвертый тип именованных блоков PL/SQL – триггеры. Триггеры во многом похожи на подпрограммы, но некоторые их характеристики существенно различаются. В этой главе показано, как создавать триггеры, и приведены примеры, иллюстрирующие их применение.

# Создание триггеров

Триггеры так же, как процедуры и функции, являются именованными блоками PL/SQL с разделом объявлений, выполняемым разделом и разделом исключительных ситуаций. Подобно модулям, триггеры необходимо хранить в базе данных, а не локально в блоке. Процедура вызывается явным образом из другого блока, причем при вызове процедуры ей могут передаваться различные аргументы. Триггер же выполняется неявно, всякий раз, когда происходит событие, запускающее этот триггер, причем использование аргументов не допускается. Акт выполнения триггера называется его *активизацией* (firing). Запускается триггер операцией DML (INSERT, UPDATE или DELETE), выполняемой над базой данных.

Триггеры можно использовать для:

- Реализации сложных ограничений целостности данных, которые невозможно осуществить через описательные ограничения, устанавливаемые при создании таблицы
- Слежения за информацией, хранимой в таблице, путем записи вносимых изменений и пользователей, вносящих эти изменения
- Автоматического оповещения других программ о том, что делать в случае изменения информации, содержащейся в таблице

Предположим, что требуется отслеживать статистические показатели, касающиеся различных профилирующих дисциплин студентов, в том числе количество зарегистрированных студентов и общее число полученных зачетов. Результаты будут храниться в таблице **major_stats**:

```
-- Этот пример является частью файла tables.sql.
CREATE TABLE major_stats (
  Major            VARCHAR2(30),
  Total_credits    NUMBER,
  Total_students   NUMBER);
```

Для актуальности информации в таблице **major_stats** создадим триггер для таблицы **students**, который будет обновлять **major_stats** всякий раз при изменении **students.** Назовем этот триггер **UpdateMajorStats.** Он будет срабатывать после выполнения любой операции DML над **students.** Тело триггера обращается к таблице **students** с запросом и обновляет статистические показатели таблицы **major_stats** свежей информацией:

```
-- Этот пример содержится в файле updatems.sql.
CREATE OR REPLACE TRIGGER UpdateMajorStats
  /* Поддерживает актуальность таблицы major_stats, отслеживая все
     изменения, вносимыев таблицу students. */
  AFTER INSERT OR DELETE OR UPDATE ON students
DECLARE
  CURSOR c_Statistics IS
     SELECT major, COUNT(*) total_students,
            SUM(current_credits) total_credits
       FROM students
       GROUP BY major;
BEGIN
  /* Последовательно просмотрим информацию о каждой дисциплине и
     попытаемся обновить статистические показатели major_stats,
     соответствующие дисциплине. Если строка не существует,создадим ее. */
  FOR v_StatsRecord in c_Statistics LOOP
     UPDATE major_stats
       SET  total_credits = v_StatsRecord.total_credits,
            total_students = v_StatsRecord.total_students
       WHERE major = v_StatsRecord.major;
```

```
        /* Проверим наличие строки. */
        IF SQL%NOTFOUND THEN
          INSERT INTO major_stats (major, total_credits, total_students)
            VALUES (v_StatsRecord.major, v_StatsRecord.total_credits,
                    v_StatsRecord.total_students);
        END IF;
      END LOOP;
    END UpdateMajorStats;
```

Общий синтаксис создания триггера таков:

```
CREATE [OR REPLACE] TIGGER имя_триггера
{BEFORE | AFTER} активизирующее_событие ON ссылка_на_таблицу
[FOR EACH ROW [WHEN условие_срабатывания]]
тело_триггера;
```

где *имя_триггера* – это имя триггера, *активизирующее_событие* указывает момент активизации триггера (в случае с **UpdateMajorStats** после любой операции DML), *ссылка_на_таблицу* – таблица, для которой создан триггер, а *тело_триггера* – программный текст триггера. Если присутствует *условие_срабатывания* в условии WHEN (когда), то вначале оно вычисляется, а тело триггера выполняется только в том случае, если это условие истинно.

# Элементы триггера

Обязательными элементами триггера являются его имя, активизирующее событие и тело. Условие WHEN необязательно.

## Имена триггеров

Пространство имен триггеров отличается от пространств имен других подпрограмм. *Пространством имен* (namespace) называется набор идентификаторов, разрешенных для использования в качестве имен объекта. Для процедур, модулей и таблиц применяется одно и то же пространство имен. Это означает, что в пределах одной схемы базы данных все объекты, использующие одно и то же пространство имен, должны иметь уникальные имена. Например, процедуре и модулю запрещается давать одинаковые имена.

Для триггеров определено собственное пространство имен, то есть триггер может иметь то же имя, что и какая-либо таблица или процедура. Однако в пределах одной схемы конкретное имя может быть дано только одному триггеру. Имена триггеров – это идентификаторы базы данных и поэтому подчиняются тем же правилам, что и другие идентификаторы (см. главу 2). Например, для таблицы major_stats можно создать триггер **major_stats**, но процедуру с именем **major_stats** создать нельзя. Ниже приведен пример сеанса работы в SQL*Plus:

```
-- Этот пример содержится в файле samename.sql.
SQL>  CREATE OR REPLACE TRIGGER major_stats
  2     BEFORE INSERT ON major_stats
  3   BEGIN
  4     INSERT INTO temp_table (char_col)
  5       VALUES ('Trigger fired!');
  6   END major_stats;
  7   /

Trigger created.

SQL> CREATE OR REPLACE PROCEDURE major_stats AS
  2   BEGIN
  3     INSERT INTO temp_table (char_col)
  4       VALUES ('Procedure called!');
  5   END major_stats;
  6   /
CREATE OR REPLACE PROCEDURE major_stats AS
*
ERROR at line 1;
```

```
ORA-00955: name is already used by an existing object
```
(имя уже используется существующим объектом. — *Прим. пер.*)

▼ СОВЕТУЕМ Хотя применять для таблицы и триггера одинаковые имена и не запрещено, автор не рекомендует этого делать. Лучше дать каждому триггеру уникальное имя, указывающее на то, какие функции он выполняет, точно так же, как и таблице, для которой он создан.

## Типы триггеров

Тип триггера определяется тем, какое событие его активизирует: INSERT (ввод), UPDATE (обновление) или DELETE (удаление). Триггеры могут активизироваться до (BEFORE) или после (AFTER) операции, а также для строки или оператора. Возможные варианты триггеров приведены в таблице 9.1.

### ТАБЛИЦА 9.1. Типы триггеров

| Категория | Значение | Комментарии |
|---|---|---|
| Оператор | INSERT, UPDATE или DELETE | Определяет, какой оператор DML вызывает активизацию триггера. |
| Момент времени | BEFORE или AFTER | Определяет момент активизации триггера: до или после выполнения оператора. |
| Уровень | Строка или оператор | Если триггер является строковым, то он активизируется один раз для каждой из строк, на которые воздействует оператор, вызывающий срабатывание триггера. Если триггер является операторным, то он активизируется один раз до или после оператора. Строковые триггеры содержат условие FOR EACH ROW (для каждой строки) в описании триггера. |

Значения, заданные для оператора, момента времени и уровня, определяют тип триггера. Всего существует 12 возможных типов: 3 оператора ö 2 момента времени ö 2 уровня. Ниже приведены примеры правильных триггеров:

- До выполнения операции обновления на операторном уровне
- После выполнения операции ввода на уровне строк
- До выполнения операции удаления на уровне строк

PL/SQL 2.1 ... и ВЫШЕ Для таблицы может быть создано до 12 триггеров — по одному каждого типа. Однако начиная с PL/SQL 2.1 (Oracle7 версии 7.1) для таблицы может создаваться по нескольку триггеров каждого вида. Это дает возможность определять для одной таблицы сколь угодно много триггеров. Порядок активизации триггеров описан ниже в разделе "Порядок активизации триггеров".

Кроме того, триггер может активизироваться для нескольких типов операторов, вызывающих его срабатывание. Например, триггер **UpdateMajorStats** срабатывает на операторы INSERT, UPDATE и DELETE. Активизирующее событие указывает одну или несколько операций DML, которые вызывают срабатывание триггера.

PL/SQL 8.0 ... и ВЫШЕ

## Триггеры INSTEAD OF

В PL/SQL 8.0 предлагается еще один вид триггеров — INSTEAD OF (вместо), которые можно создавать только для представлений (объектных или реляционных); причем такие триггеры будут активизироваться вместо операторов DML, вызывающих их срабатывание. Триггеры INSTEAD OF должны быть строковыми триггерами. Такие триггеры необходимы, поскольку в основе представлений, для которых они создаются, могут лежать соединения (joins), а не все соединения являются обновляемыми. Триггер же позволяет выполнять обновление информации надлежащим образом. Для примера рассмотрим представление **room_summary**:

```
-- Этот пример является частью файла instead.sql.
CREATE VIEW room_summary AS
  SELECT building, sum(number_seats) total_seats
```

```
        FROM rooms
    GROUP BY building;
```

Удалить информацию непосредственно из этого представления нельзя:

```
SQL> DELETE FROM room_summary where building = 'Building 7';
DELETE FROM room_summary where building = 'Building 7';
            *
ERROR at line 1:
ORA-01732: data manipulation operation not legal on this view
```

(операция манипуляции данными для этого представления запрещена. — *Прим. пер.*)

Однако можно создать триггер INSTEAD OF и с его помощью выполнить требуемую операцию — удалить указанные строки таблицы **rooms**:

```
-- Этот пример является частью файла instead.sql.
CREATE TRIGGER room_summary_delete
  INSTEAD OF DELETE ON room_summary
  FOR EACH ROW
BEGIN
  -- Удалим все строки таблицы rooms, соответствующие
  -- строке представления room_summary.
  DELETE FROM rooms
      WHERE building = :old.building;
END room_summary_delete;
```

С помощью триггера room_summary_delete оператор DELETE выполняется успешно и делает именно то, что нужно. Более подробно о триггерах INSTEAD OF см. главу 11.

## Ограничения, налагаемые на триггеры

Тело триггера является блоком PL/SQL. Любой оператор, выполнение которого разрешено в блоке PL/SQL, можно выполнить и в теле триггера при условии соблюдения следующих ограничений:

- В триггере нельзя задавать ни один из операторов управления транзакциями: COMMIT, ROLLBACK или SAVEPOINT. Срабатывание триггера является частью процесса выполнения активизирующего оператора, то есть частью той транзакции, которая охватывает и активизирующий оператор. Когда этот оператор завершается или откатывается, все выполненное триггером также завершается или откатывается.
- В процедурах и функциях, вызывающихся в теле триггера, также нельзя задавать какие-либо из операторов управления транзакциями.
- В теле триггера нельзя объявлять переменные с типами LONG и LONG RAW. Кроме того, в псевдозаписях :new и :old (см. ниже) нельзя ссылаться на столбцы типов LONG и LONG RAW таблицы, для которой определен триггер.

Из тела триггера можно обращаться не ко всем таблицам. В зависимости от типа триггера и ограничений (constraints), налагаемых на таблицы, таблицы могут быть изменяющимися (mutating). Эта ситуация детально обсуждается ниже в разделе "Изменяющиеся таблицы".

# Триггеры и словарь данных

Как и в случае с хранимыми подпрограммами, в некоторых представлениях словаря данных содержится информация о триггерах и о состоянии каждого из них. При создании или удалении какого-либо триггера эти представления изменяются.

## Представления словаря данных

При создании триггера его исходный текст сохраняется в представлении словаря данных, называемом **user_triggers**. В этом представлении содержится тело триггера, условие WHEN, активизирующая таблица и вид триггера. Например, в следующем запросе возвращается информация о триггере **UpdateMajorStats**:

```
SQL> SELECT trigger_type, table_name, triggering_event
  2    FROM user_triggers
  3    WHERE trigger_name = 'UPDATEMAJORSTATS';
```

```
TRIGGER_TYPE        TABLE_NAME          TRIGGERING_EVENT
----------------    ------------------  ---------------------------------
AFTER STATEMENT     STUDENTS            INSERT OR UPDATE OR DELETE
```

Более подробно о представлениях словаря данных рассказано в приложении D.

## Удаление и запрещение триггеров

Триггеры, как и процедуры и модули, можно удалять. Синтаксис команды удаления триггера таков:

DROP TRIGGER *имя_триггера*;

где *имя_триггера* – имя удаляемого триггера. При этом триггер удаляется из словаря данных. В операторе создания триггера можно указывать ключевые слова OR REPLACE, как это делается для подпрограмм. В этом случае, если триггер существует, он сначала удаляется.

Однако в отличие от процедур и функций, можно, не удаляя триггер, запретить (disable) его использование. Когда триггер запрещен, он по-прежнему находится в словаре данных, но никогда не активизируется. Для запрещения триггера применяется оператор ALTER TRIGGER:

ALTER TRIGGER *имя_триггера* {DISABLE | ENABLE};

где *имя_триггера* – это имя триггера. При создании каждого триггера его использование разрешено (enable) по умолчанию. С помощью оператора ALTER TRIGGER можно запретить, а затем повторно разрешить любой триггер. Ниже приведен пример запрещения и повторного разрешения триггера **UpdateMajorStats**:

```
SQL> ALTER TRIGGER UpdateMajorStats DISABLE
Trigger altered.

SQL> ALTER TRIGGER UpdateMajorStats ENABLE
Trigger altered.
```

Кроме того, при помощи команды ALTER TABLE можно разрешить или запретить использование всех триггеров определенной таблицы, если добавить в эту команду конструкцию ENABLE ALL TRIGGERS (разрешить все триггеры) или DISABLE ALL TRIGGERS (запретить все триггеры). Например:

```
SQL> ALTER TABLE students
  2    ENABLE ALL TRIGGERS;
Table altered.

SQL> ALTER TABLE students
  2    DISABLE ALL TRIGGERS;
Table altered.
```

В столбце status представления user_triggers находится либо 'ENABLED', либо 'DISABLED', и это значение показывает текущее состояние триггера. Запрещение триггера не удаляет его из словаря данных, как это происходит при выполнении команды DROP.

## P-код триггера

Когда в словаре данных хранится модуль или подпрограмма, в дополнение к исходному программному тексту объекта хранится еще и его скомпилированный p-код. Однако для триггеров это не так. В словаре данных хранится исходный текст триггера, но не его p-код, поэтому триггер должен компилироваться всякий раз, когда он считывается из словаря. Это не оказывает влияния на способы создания и использования триггера, но может повлиять на его производительность. Более подробно о производительности и настройке системы рассказано в главе 22.

PL/SQL 2.3 ... и ВЫШЕ

В PL/SQL 2.3 (Oracle7 версии 7.3) триггеры, как процедуры, функции и модули, хранятся в скомпилированном виде. Это позволяет вызывать триггеры без перекомпиляции. Однако из-за того что триггеры являются такими же хранимыми объектами, как и модули и подпрограммы, вместе с триггерами хранится информация о зависимостях, установленных для них. В результате триггеры так же, как модули и подпрограммы, могут автоматически становиться недостоверными. Становясь недостоверным, триггер перекомпилируется при следующей его активизации.

# Порядок активизации триггера

Триггер активизируется при выполнении оператора DML. Алгоритм выполнения оператора DML таков:

1. Выполняется операторный триггер BEFORE (при его наличии).
2. Для каждой строки, на которую воздействует оператор:
   a. выполняется строковый триггер BEFORE (при его наличии);
   b. выполняется собственно оператор;
   c. выполняется строковый триггер AFTER (при его наличии);
3. Выполняется операторный триггер AFTER (при его наличии).

Для иллюстрации вышесказанного создадим все четыре вида триггеров UPDATE для таблицы **classes** – BEFORE и AFTER, операторный и строковый:

```
-- Этот пример содержится в файле order.sql.
CREATE SEQUENCE trigger_seq
  START WITH 1
  INCREMENT BY 1;

CREATE OR REPLACE TRIGGER classes_BStatement
  BEFORE UPDATE ON classes
BEGIN
  INSERT INTO temp_table (num_col, char_col)
      VALUES (trigger_seq.NEXTVAL, 'Before Statement trigger');
END classes_BStatement;

CREATE OR REPLACE TRIGGER classes_AStatement
  AFTER UPDATE ON classes
BEGIN
  INSERT INTO temp_table (num_col, char_col)
      VALUES (trigger_seq.NEXTVAL, 'After Statement trigger');
END classes_AStatement;

CREATE OR REPLACE TRIGGER classes_BRow
  BEFORE UPDATE ON classes
  FOR EACH ROW
BEGIN
  INSERT INTO temp_table (num_col, char_col)
      VALUES (trigger_seq.NEXTVAL, 'Before Row trigger');
END classes_BRow;

CREATE OR REPLACE TRIGGER classes_ARow
  AFTER UPDATE ON classes
  FOR EACH ROW
BEGIN
  INSERT INTO temp_table (num_col, char_col)
      VALUES (trigger_seq.NEXTVAL, 'After Row trigger');
END classes_ARow;
```

Теперь выполним оператор UPDATE:

```
UPDATE classes
  SET num_credits = 4
  WHERE department IN ('HIS', 'CS');
```

Этот оператор воздействует на четыре строки. Каждый из операторных триггеров BEFORE и AFTER выполняется один раз, а каждый из строковых триггеров BEFORE и AFTER – четыре раза. Обратившись после этого с запросом к таблице **temp_table**, получим следующие результаты:

```
-- Этот пример является частью файла order.sql.
```

```
SQL>  SELECT  *  FROM  temp_table
  2    ORDER  BY  num_col;
  NUM_COL   CHAR_COL
---------  --------------------------------
       1     Before  Statement  trigger
       2     Before  Row  trigger
       3     After  Row  trigger
       4     Before  Row  trigger
       5     After  Row  trigger
       6     Before  Row  trigger
       7     After  Row  trigger
       8     Before  Row  trigger
       9     After  Row  trigger
      10     After  Statement  trigger
```

При активизации каждого из триггеров будут видны изменения, сделанные предыдущими триггерами, а также внесенные оператором. Порядок, в котором активизируются триггеры одного и того же типа, не определен. Если этот порядок важен, следует объединить все операции в один триггер.

## Использование :old и :new в строковых триггерах

Строковый триггер срабатывает один раз для каждой строки, обрабатываемой активизирующим оператором. Внутри триггера можно обращаться к строке, обрабатываемой в данный момент. Для этого служат две *псевдозаписи* — :old и :new. Хотя синтаксически они рассматриваются как записи, фактически они записями не являются (более детально этот вопрос обсуждается ниже). Поэтому их называют псевдозаписями, и их назначение приведено в таблице 9.2. Тип обеих псевдозаписей определяется так:

*активизирующая_таблица*%ROWTYPE;

где *активизирующая_таблица* — это таблица, для которой создан триггер.

### ТАБЛИЦА 9.2. :old и :new

| Активизирующий оператор | :old | :new |
|---|---|---|
| INSERT | Не определена — во всех полях содержатся NULL-значения | Значения, которые будут введены после выполнения оператора |
| UPDATE | Исходные значения, содержащиеся в строке перед обновлением данных | Новые значения, которые будут введены после выполнения оператора |
| DELETE | Исходные значения, содержащиеся в строке перед ее удалением | Не определена — во всех полях содержатся NULL-значения |

▼ ВНИМАНИЕ Псевдозапись :old не определена для операторов INSERT, а псевдозапись :new — для операторов DELETE. При использовании :old в операторе INSERT или :new в операторе DELETE компилятор PL/SQL не будет генерировать ошибку, но значения полей обеих записей будут NULL-значениями. Двоеточие перед :new и :old обязательно, причем это единственная ситуация в PL/SQL, когда двоеточие используется для ограничения переменных привязки. Дело в том, что :new и :old фактически реализованы как переменные привязки (в качестве базовых переменных, используемых во встроенном PL/SQL). Двоеточие отличает их от обычных переменных PL/SQL, как и другие переменные привязки. Переменные привязки используются в различных средах выполнения программ PL/SQL (см. главу 13). Ссылка, подобная :new.поле, будет правильной только в том случае, если поле является полем активизирующей таблицы.

▼ ОСТОРОЖНО Хотя :new и :old синтаксически рассматриваются в качестве записей типа активизирующая_таблица%ROWTYPE, в действительности они записями не являются. Поэтому операции, вполне нормально выполняющиеся над записями, не могут быть выполнены над :new и :old. Например, эти псевдозаписи нельзя присваивать чему-либо в качестве записей целиком. Присваивать можно только отдельные поля этих псевдозаписей. Проиллюстрируем сказанное на примере:

```
-- Этот пример содержится в файле pseudo.sql.
CREATE OR REPLACE TRIGGER TempDelete
BEFORE DELETE ON temp_table
FOR EACH ROW
DECLARE
    v_TempRec temp_table%ROWTYPE;
BEGIN
    /* Этот оператор неверен, так как :old не является настоящей
       записью. */
    v_TempRec := :old;

    /* Однако можно выполнить то же самое, присвоив поля по
       отдельности. */
    v_TempRec.char_col := :old.char_col;
    v_TempRec.num_col := :old.num_col;
END TempDelete;
```

Кроме того, :old и :new нельзя передавать процедурам или функциям, принимающим аргументы типа *активизирующая_таблица%ROWTYPE*.

В триггере **GenerateStudentID**, приведенном ниже, используется :new. Это триггер BEFORE, срабатывающий на операторы INSERT и UPDATE, и его назначением является заполнение поля **ID** таблицы **students** значением, генерируемым при помощи последовательности **student_sequence**.

```
-- Этот пример содержится в файле studid.sql.
CREATE OR REPLACE TRIGGER GenerateStudentID
  BEFORE INSERT OR UPDATE ON students
  FOR EACH ROW
BEGIN
  /* Заполним поле ID таблицы students следующим значением из
     student_sequence. Поскольку ID - это столбец таблицы students,
     :new.ID является правильной ссылкой. */
  SELECT student_sequence.nextval
     INTO :new.ID
     FROM dual;
END GenerateStudentID;
```

**GenerateStudentID** фактически модифицирует значение **:new.ID**. Это одно из полезных свойств псевдозаписи :new – когда выполнение оператора завершается, используются значения, содержащиеся в :new. С помощью триггера **GenerateStudentID** можно выполнить оператор

```
INSERT INTO students (first_name, last_name)
  VALUES ('Lolita', 'Lazarus');
```

без ошибок. Хотя значение для столбца **ID**, являющегося первичным ключом, указано не было (что обязательно), это значение будет введено триггером. Если указать значение для **ID**, то оно будет проигнорировано, так как триггер его изменит. Если выполнить оператор

```
INSERT INTO students (ID, first_name, last_name)
  VALUES (-7, 'Lolita', 'Lazarus');
```

будет получен тот же самый результат. В любом случае для поля **ID** будет использовано значение **student_sequence.nextval**.

Именно поэтому нельзя изменить :new в строковом триггере AFTER, так как оператор будет обработан раньше. Вообще говоря, :new модифицируется только в строковых триггерах BEFORE, а :old никогда не модифицируется, а лишь считывается.

Псевдозаписи :new и :old разрешено использовать только в строковых триггерах. Если сослаться на одну из них в операторном триггере, то будет выдана ошибка компиляции. Поскольку операторный триггер выполняется лишь однажды (даже, если в операторе обрабатывается несколько строк), в этой ситуации псевдозаписи :old и :new не имеют никакого смысла. Действительно, на какую из строк будет ссылаться каждая из них?

## Условие WHEN

Условие WHEN можно использовать только для строковых триггеров. Если оно присутствует в триггере, то тело триггера будет выполняться только для строк, соответствующих условию, указанному в WHEN. Общий вид условия WHEN таков:

WHEN *условие*

где *условие* является логическим выражением, вычисляющимся для каждой строки. В условии можно ссылаться на псевдозаписи :new и :old, но двоеточие в данном случае не применяется. Двоеточие можно указывать только в теле триггера. Например, тело триггера **CheckCredits** выполняется в том случае, когда текущее число зачетов, полученных студентом, превышает 20:

```
CREATE OR REPLACE TRIGGER CheckCredits
  BEFORE INSERT OR UPDATE OF current_credits ON students
  FOR EACH ROW
  WHEN (new.current_credits > 20)
BEGIN
  /* Тело триггера. */
END;
```

Триггер CheckCredits можно было бы записать следующим образом:

```
CREATE OR REPLACE TRIGGER CheckCredits
  BEFORE INSERT OR UPDATE OF current_credits ON students
  FOR EACH ROW
BEGIN
  IF :new.current_credits > 20 THEN
     /* Тело триггера. */
  END IF;
END;
```

## Использование триггерных предикатов INSERTING, UPDATING и DELETING

Триггер **UpdateMajorStats,** *описанный выше, является триггером* INSERT, UPDATE и DELETE. Внутри триггера такого типа (который срабатывает на различные виды операторов DML) можно использовать три логические функции, определяющие тип выполняемой операции. Эти логические функции (предикаты) – INSERTING, UPDATING и DELETING. Их работа описана в таблице 9.3.

**ТАБЛИЦА 9.3.**

| Предикат | Принимаемое значение |
|---|---|
| INSERTING | TRUE, если активизирующий оператор INSERT; FALSE в противном случае |
| UPDATING | TRUE, если активизирующий оператор UPDATE; FALSE в противном случае |
| DELETING | TRUE, если активизирующий оператор DELETE; FALSE в противном случае |

В триггере **LogRSChanges** эти предикаты используются для записи всех изменений, вносимых в таблицу **registered_students.** Помимо изменения записывается пользователь, внесший это изменение. Записи сохраняются в таблице **RS_audit**, которая выглядит следующим образом:

```
-- Этот пример является частью файла tables.sql.
CREATE TABLE RS_audit (
  Old_student_id    NUMBER(5),
  Old_department    CHAR(3),
  Old_course        NUMBER(3),
  Old_grade         CHAR(1),
  New_student_id    NUMBER(5),
  New_department    CHAR(3),
  New_course        NUMBER(3),
  New_grade         CHAR(1)
  Changed_by        VARCHAR2(8),
  Timestamp         DATE
  );
```

Создадим триггер **LogRSChanges**:

```
-- Этот пример содержится в файле rschange.sql.
CREATE OR REPLACE TRIGGER LogRSChanges
  BEFORE INSERT OR DELETE OR UPDATE ON registered_students
  FOR EACH ROW
DECLARE
  v_ChangeType CHAR(1);
BEGIN
  /* Используем 'I' для INSERT, 'D' для DELETE и 'U' для UPDATE. */
  IF INSERTING THEN
      v_ChangeType := 'I';
  ELSIF UPDATING THEN
      v_ChangeType := 'U';
  ELSE
      v_ChangeType := 'D';
  END IF;

  /* Запишем все изменения, внесенные в таблицу registered_students,
     в таблицу RS_audit. Для генерирования временной метки
     воспользуемся функцией SYSDATE, а для получения идентификатора
     текущего пользователя - функцией USER. */
  INSERT INTO RS_audit
      (change_type, changed_by, timestamp,
      old_student_id, old_department, old_course, old_grade,
      new_student_id, new_department, new_course, new_grade)
  VALUES
      (v_ChangeType, USER, SYSDATE,
      :old.student_id, :old.department, :old.course, :old.grade,
      :new.student_id, :new.department, :new.course, :new.grade);
END LogRSChanges;
```

Обычно триггеры используются для аудита, или контроля информации, что демонстрирует триггер **LogRSChanges**. В базе данных Oracle имеется специальное средство, позволяющее осуществлять аудит данных, который с помощью триггеров можно сделать более гибким. Например, можно изменить **LogRSChanges** так, чтобы записывались только изменения, вносимые определенными пользователями. Можно также проверять наличие у пользователей полномочий на внесение изменений и устанавливать исключительную ситуацию (с помощью RAISE_APPLICATION_ERROR), если полномочия отсутствуют.

# Изменяющиеся таблицы

*Из тела триггера можно обращаться не ко всем таблицам и столбцам.* Чтобы определить, к каким таблицам возможен доступ, необходимо понимать, что такое изменяющиеся и ограничивающие таблицы. *Изменяющаяся таблица* (mutating table) — это таблица, которая в данный момент модифицируется оператором DML. Для триггера это та таблица, для которой он был создан. Таблицы, обновляющиеся в результате реализации ограничений ссылочной целостности DELETE CASCADE — каскадного удаления, также являются изменяющимися (более подробно об ограничениях ссылочной целостности рассказано в руководстве Oracle Server Reference). *Ограничивающая таблица* (constraining table) — это таблица, информация которой может быть считана при реализации ограничения ссылочной целостности. Проиллюстрируем вышесказанное на примере создания таблицы **registered_students**:

```
-- Этот пример является частью файла tables.sql.
CREATE TABLE registered_students (
  student_id NUMBER(5) NOT NULL,
  department CHAR(3) NOT NULL,
  course NUMBER(3) NOT NULL,
  grade CHAR(1),
  CONSTRAINT rs_grade
     CHECK (grade IN ('A', 'B', 'C', 'D', 'E')),
  CONSTRAINT rs_student_id
     FOREIGN KEY (student_id) REFERENCES students (id),
  CONSTRAINT rs_department_course
     FOREIGN KEY (department, course)
  REFERENCES classes (department, course)
);
```

Для таблицы **registered_students** установлено два ограничения ссылочной целостности. Поэтому таблицы **students** и **classes** являются ограничивающими по отношению к **registered_students**. Сама же таблица **registered_students** изменяется во время выполнения над ней оператора DML. Благодаря установленным ограничениям информация обеих таблиц, *classes и students*, также будет модифицирована и/или считана оператором DML.

SQL-операторы в теле триггера не могут:

- Считывать или модифицировать информацию любой таблицы, изменяющейся в результате выполнения активизирующего оператора. В число таких таблиц входит и сама активизирующая таблица.
- Считывать или модифицировать информацию столбца первичного ключа, уникальных столбцов и столбцов внешних ключей таблицы, являющейся ограничивающей по отношению к изменяющейся таблице. Другие столбцы можно модифицировать.

Эти правила верны для всех строковых триггеров. Для операторных триггеров они применимы только в тех случаях, когда те активизируются в результате выполнения операции каскадного удаления информации.

**▼ ВНИМАНИЕ** Если оператор INSERT воздействует только на одну строку, то для строковых триггеров BEFORE и AFTER, работающих с этой строкой, активизирующая таблица не является изменяющейся. Это единственная ситуация, когда строковый триггер может считывать и модифицировать информацию активизирующей таблицы. Для таких операторов, как

```
INSERT INTO таблица SELECT ...
```

активизирующая таблица всегда является изменяющейся, даже когда в подзапросе возвращается только одна строка.

В качестве примера рассмотрим триггер **CascadeRSInserts.** Он, хотя и модифицирует обе таблицы **students** и **classes,** тем не менее, верен, так как модифицируемые столбцы таблиц **students** и **classes** не являются ключевыми столбцами. Пример неверного триггера будет рассмотрен в следующем разделе.

```
-- Этот пример содержится в файле rsinsert.sql.
CREATE OR REPLACE TRIGGER CascadeRSInserts
  /* Поддерживает синхронизацию таблиц registered_students,
```

```
        students и classes. */
    BEFORE INSERT ON registered_students
    FOR EACH ROW
DECLARE
    v_Credits classes.num_credits%TYPE;
BEGIN
    -- Определим число зачетов для данной группы.
    SELECT num_credits
        INTO v_Credits
        FROM classes
        WHERE department = :new.department
        AND course = :new.course;

    -- Модифицируем текущее число зачетов для данного студента.
    UPDATE students
        SET current_credits = current_credits + v_Credits
          WHERE ID = :new.student_id;
    -- Добавим единицу к числу студентов в группе.
    UPDATE classes
        SET current_students = current_students + 1
          WHERE department = :new.department
          AND course = :new.course;
END CascadeRSInserts;
```

Для завершения общей картины следует создать триггеры, обновляющие таблицы **students** и **classes** при удалении или обновлении информации, содержащейся в таблице **registered_students**. Эти триггеры должны иметь примерно такой же вид, что и **CascadeRSInserts**.

## Пример изменяющейся таблицы

Предположим, что общее число студентов, занимающихся каждой из дисциплин, требуется ограничить пятью. Это можно сделать, если создать для таблицы **students** строковый триггер BEFORE INSERT или UPDATE:

```
-- Этот пример содержится в файле limmajor.sql.
CREATE OR REPLACE TRIGGER LimitMajors
    /* Ограничивает общее число студентов, занимающихся каждой из
       дисциплин, пятью студентами. Если этот предел превышается, то
       посредством raise_application_error устанавливается ошибка. */
    BEFORE INSERT OR UPDATE OF major ON students
    FOR EACH ROW
DECLARE
    v_MaxStudents CONSTANT NUMBER := 5;
    v_CurrentStudents NUMBER;
BEGIN
    -- Определим текущее число студентов, занимающихся данной
    -- дисциплиной.
    SELECT COUNT(*)
        INTO v_CurrentStudents
        FROM students
        WHERE major = :new.major;

    -- Если места для нового студента нет, устанавливается ошибка.
    IF v_CurrentStudents + 1 > v_MaxStudents THEN
        RAISE_APPLICATION_ERROR(-20000,
            'Too many students in major ' || :new.major);
```

```
    END IF;
END LimitMajors;
```

Процедура RAISE_APPLICATION_ERROR (установить ошибку приложения) является стандартной процедурой, которая более детально рассматривается в главе 10. Сразу кажется, что при выполнении этого триггера будет получен желаемый результат. Однако если обновить **students** и активизировать триггер, результат будет таким:

```
SQL> UPDATE students
  2    SET major = 'History'
  3    WHERE ID = 10003;
UPDATE students
  *
ERROR at line 1:
ORA-04091: table EXAMPLE.STUDENTS is mutating, trigger/function
           may not see it
ORA-06512: at line 7
ORA-04088: error during execution of trigger 'EXAMPLE.LIMITMAJORS'
```

(ОШИБКА в строке 1:
ORA-04091: таблица EXAMPLE.STUDENTS является изменяющейся,
триггер/функция ее, возможно, не видит
ORA-06512: в строке 7
ORA-04088: ошибка во время выполнения триггера
'EXAMPLE.LIMITMAJORS'. — *Прим. пер.*)

Ошибка ORA-4091 возникает в результате обращения триггера **LimitMajors** к собственной активизирующей таблице, которая изменяется. ORA-4091 устанавливается при активизации, а не при создании триггера.

## Как избежать ошибок, связанных с изменяющимися таблицами

Таблица **students** является изменяющейся только для строкового триггера. Значит, в строковом триггере обратиться к ней с запросом нельзя, но можно — в операторном триггере. Однако просто преобразовать триггер **LimitMajors** в операторный нельзя, поскольку в теле триггера необходимо использовать значение **:new.major**. Эта ситуация разрешима, если создать два триггера — строковый и операторный. В строковом триггере записывается значение **:new.major**, но таблица **students** не запрашивается. Этот запрос будет выполняться в операторном триггере, причем, со значением, записанным в строковом триггере.

Записать это значение лучше всего, использовав таблицы PL/SQL внутри программного модуля. При этом можно сократить число значений, обрабатываемых во время каждой операции обновления данных. Кроме того, для каждого соединения осуществляется своя собственная конкретизация модульных переменных, поэтому не надо беспокоиться о выполнении обновления информации одновременно различными соединениями. Такой метод реализуется с помощью модуля **student_data** и триггеров **RLimitMajors** и **SLimitmajors**:

```
-- Этот пример содержится в файле mutating.sql.
CREATE OR REPLACE PACKAGE StudentData AS
  TYPE t_Majors IS TABLE OF students.major%TYPE
      INDEX BY BINARY_INTEGER;
  TYPE t_IDs IS TABLE OF students.ID%TYPE
      INDEX BY BINARY_INTEGER;
  V_StudentMajors  t_Majors;
  V_StudentIDs     t_IDs;
  V_NumEntries     BINARY_INTEGER := 0;
END StudentData;

CREATE OR REPLACE TRIGGER RLimitMajors
```

```
    BEFORE INSERT OR UPDATE OF major ON students
    FOR EACH ROW
BEGIN
    /* Запишем новые данные в StudentData. В таблицу students
       изменения вносить не будем, чтобы избежать ошибки ORA-4091. */
    StudentData.v_NumEntries := StudentData.v_NumEntries + 1;
    StudentData.v_StudentMajors(StudentData.v_NumEntries) :=
       :new.major;
    StudentData.v_StudentIDs(StudentData.v_NumEntries) := :new.id;
END RLimitMajors;

CREATE OR REPLACE TRIGGER SLimitMajors
  AFTER INSERT OR UPDATE OF major ON students
DECLARE
    V_MaxStudents       CONSTANT NUMBER := 5;
    V_CurrentStudents   NUMBER;
    V_StudentID         students.ID%TYPE;
    V_Major             students.major%TYPE;
BEGIN
    /* Последовательно просмотрим, не внесен ли новый студент и не
       изменились ли характеристики каждого студента, и проверим,
       соблюдается ли установленное условие. */
    FOR v_LoopIndex IN 1..StudentData.v_NumEntries LOOP
       v_StudentID := StudentData.v_StudentIDs(v_LoopIndex);
       v_Major := StudentData.v_StudentMajors(v_LoopIndex);

       -- Определим текущее число студентов, занимающихся данной дисциплиной.
       SELECT COUNT(*)
          INTO v_CurrentStudents
          FROM students
          WHERE major = v_Major;

       -- Если места для нового студента нет, устанавливается ошибка.
       IF v_CurrentStudents > v_MaxStudents THEN
          RAISE_APPLICATION_ERROR(-20000,
             'Too many students for major ' || v_Major ||
             ' because of student ' || v_StudentID);
       END IF;
    END LOOP;

    -- Сбросим счетчик, чтобы при следующем выполнении триггера
    -- использовались новые данные.
    StudentData.v_NumEntries := 0;
END LimitMajors;
```

▼ ВНИМАНИЕ **Перед запуском на выполнение приведенного выше сценария обязательно удаляйте неверный триггер *LimitMajors*.**

Теперь можно проверить эту группу триггеров, обновляя таблицу **students** до тех пор, пока не окажется слишком много студентов, профилирующая дисциплина которых — история.

```
SQL> UPDATE students
  2    SET major = 'History'
  3    WHERE ID = 10003;
1 row updated.
```

```
SQL> UPDATE students
  2    SET major = 'History'
  3    WHERE ID = 10002;
1 row updated.

SQL> UPDATE students
  2   SET major = 'History'
  3   WHERE ID = 10009;
UPDATE students
  *
ERROR at line 1:
ORA-20000: Too many students for major History because of student
           10009
ORA-06512: at line 7
ORA-04088: error during execution of trigger
           'EXAMPLE.SLIMITMAJORS'
```

(ОШИБКА в строке 1:
ORA-20000: слишком много студентов-историков из-за студента 10009
ORA-06512: в строке 20
ORA-04088: ошибка во время выполнения триггера
'EXAMPLE.SLIMITMAJORS'. — *Прим. пер.*)

Итак, получен нужный результат. Такой метод может применяться при возникновении ошибки ORA-4091, когда строковый триггер считывает или модифицирует информацию изменяющейся таблицы. Вместо некорректного строкового триггера при обработке данных применяется правильный операторный триггер AFTER. Для хранения модифицированных строк используются таблицы PL/SQL.

При анализе этого примера необходимо обратить внимание на следующее:

- Таблицы PL/SQL содержатся в модуле, поэтому они будут видимы как строковому, так и операторному триггерам. Единственный способ гарантировать то, что используемые переменные являются глобальными, — поместить их в модуль.
- Здесь используется переменная-счетчик, **StudentData.v_NumEntries.** При создании модуля ей присваивается начальное значение 0, которое затем увеличивается строковым триггером. Операторный триггер обращается к этой переменной и после обработки сбрасывает ее в ноль, чтобы установить корректное значение для следующего оператора UPDATE, выполняемого одним из соединений.
- Проверка максимального числа студентов, изучающих определенную дисциплину, которая производится в **SLimitMajors**, немного изменена. Дело в том, что теперь это операторный триггер AFTER, и в **v_CurrentStudents** будет храниться число студентов, полученное не до, а после выполнения операции ввода или обновления данных. Именно поэтому проверка **v_CurrentStudents+1**, выполняющаяся в триггере **LimitMajors,** заменена проверкой **v_CurrentStudents.**
- Вместо таблицы PL/SQL можно было бы воспользоваться таблицей базы данных. Автор не рекомендует этого делать, поскольку соединения, одновременно выполняющие операторы UPDATE, могут мешать работе друг друга. Модульные таблицы PL/SQL уникальны среди работающих соединений, и дают возможность избежать возникновения таких проблем.

# PL/SQL в работе: реализация каскадного обновления данных

В Oracle7 имеется возможность выполнять *каскадное удаление* (delete cascade) информации — при удалении строки из родительской таблицы удаляются соответствующие строки дочерней таблицы, которая связана с родительской посредством внешнего ключа. Однако *каскадное обновление* (update cascade) по умолчанию не выполняется. Ниже приведен модуль, при помощи которого реализуется каскадное обновление. В этом модуле создаются модули и триггеры, необходимые для выполнения рассматриваемой операции, и при этом не нарушаются объявленные ограничения ссылочной целостности.

Данный пример создан группой Government Products корпорации Oracle и используется в соответствии с предоставленными полномочиями. Более подробную информацию об этой и о других полезных утилитах можно получить, просмотрев собственную страницу группы Government Products **http://govt.us.oracle.com.**

В этом модуле используются следующие средства:

- Таблицы с составными (многостолбцовыми) первичными ключами (первичный ключ (a,c,b)).
- Каскадное обновление многих дочерних таблиц из одной родительской.
- Самоссылочная целостность, подобная используемой в таблице SCOTT.EMP (mgr->empno).
- Прозрачное приложение — приложение не знает о том, что случилось.
- Версии системы базы данных 7.0 и выше.
- Настройка и оптимизация таким образом, чтобы избежать полного просмотра всех таблиц (с помощью утилиты, показывающей неиндексированные внешние ключи в схеме. Каскадное обновление неиндексированных внешних ключей может быть нежелательно).

При этом необходимо соблюдать следующие правила:

- Все внешние ключи должны указывать на ограничение первичного ключа родительской таблицы и не могут указывать на ограничение уникальности.
- Для родительской таблицы нельзя создавать никакие ограничения уникальности или уникальные индексы; можно создавать лишь ограничение первичного ключа.
- Операции обновления первичных ключей, не создающие новые первичные ключи, не поддерживаются. В качестве примера рассмотрим стандартную таблицу **dept**. Оператор

```
UPDATE dept SET deptno = deptno + 10;
```

выполняться не будет, в то время как оператор

```
UPDATE dept SET deptno = deptno + 1;
```

будет выполнен. В первом случае первичные ключи изменяются по следующей схеме: 10->20, 20->30 и т.д. Все дело в том, что при выполнении операции 10->20 "новый" первичный ключ не создается, так как отдел (department — dept), имеющий номер 20, уже существует. При выполнении второго оператора проблем не возникает, поскольку первичные ключи изменяются по такой схеме: 10->11, 20->21 и т.д. Заметьте, что таких проблем не возникает и при обновлении одной строки.

- Владелец родительской таблицы должен также быть владельцем дочерних таблиц.
- Владелец родительской таблицы должен выполнить предлагаемый модуль в своей схеме. Этот модуль необходимо инсталлировать для каждого пользователя, который хочет иметь поддержку каскадного обновления информации. После установки поддержки каскадного обновления модуль может быть удален.
- Владелец родительской таблицы должен иметь привилегии на создание процедур и триггеров. Поскольку в данном модуле для создания необходимых объектов используется DBMS_SQL, эти привилегии не могут наследоваться через роль.

В этой утилите используются модули DBMS_SQL и DBMS_OUTPUT. Модуль DBMS_SQL подробно описан в главе 15, а модуль DBMS_OUTPUT — в главе 14.

## Состав утилиты

Утилита каскадного обновления состоит из четырех SQL-сценариев, описанных в таблице 9.4 и в последующих разделах. Эти сценарии можно найти на прилагаемом компакт-диске.

**ТАБЛИЦА 9.4.**

| Имя сценария | Назначение |
|---|---|
| uc.sql | Создает модуль **update_cascade** |
| demobld.sql | Создает демонстрационный пример схемы |
| unindex.sql | Показывает индексы, которые необходимо создать для избежания полного просмотра таблиц |
| generate.sql | Создает вызовы модуля **update_cascade** для всех родительских таблиц данной схемы |

## uc.sql

С помощью сценария uc.sql в текущей схеме создается модуль **update_cascade** и его тело. В **update_cascade** содержится одна процедура, **on_table**, описание которой выглядит следующим образом:

```
PROCEDURE on_table (
table_name IN VARCHAR2,
p_preserve_rowid IN BOOLEAN DEFAULT TRUE,
p_use_dbms_output IN BOOLEAN DEFAULT FALSE);
```

Параметры процедуры on_table описаны в таблице 9.5.

**ТАБЛИЦА 9.5.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *table_name* | VARCHAR2 | Имя родительской таблицы, для которой создаются триггеры и модуль каскадного обновления |
| *p_preserve_rowid* | BOOLEAN | Если TRUE (установка по умолчанию), — идентификатор обновляемой родительской строки не изменяется. Если FALSE — идентификатор обновляемой строки изменяется, однако программный код выполняется примерно на треть быстрее, чем раньше |
| *p_use_dbms_output* | BOOLEAN | Если истинен (TRUE), то программный текст, создающий объекты, будет выводиться при помощи DBMS_OUTPUT. Если ложен (FALSE — значение по умолчанию), то объекты будут создаваться при помощи DBMS_SQL |

Например, создадим объекты, необходимые для реализации каскадного обновления таблицы **dept**, вызвав модуль **update_cascade**. Идентификаторы строк при этом будут сохраняться.

```
SQL> exec update_cascade.on_table('dept')
```

На примере приведенного ниже сеанса работы в SQL*Plus показано, что генерировать файл-сценарий для создания нужных объектов можно в том случае, когда системные привилегии CREATE TRIGGER или CREATE PROCEDURE предоставлены текущей схеме не непосредственно, а через роль.

```
SQL> SET feedback off
SQL> SPOOL tmp.sql
SQL> SET serveroutput on SIZE 1000000
SQL> exec update_cascade.on_table(p_table_name => 'dept',
                                  p_use_dbms_output => TRUE)
SQL> SPOOL off
```

## demobld.sql

При помощи сценария demobl.sql сначала создается пользователь **ucdemo** базы данных, а затем шесть демонстрационных таблиц в этой схеме. Подразумевается, что созданная схема будет выполняться пользователем system, являющимся администратором базы данных (DBA). В процессе работы выполняется оператор "DROP USER ucdemo CASCADE", поэтому следите за тем, чтобы перед запуском этого оператора реальный пользователь **ucdemo** не существовал. Перечислим создаваемые таблицы:

- **dept** (отделы) с первичным ключом
- **emp** (служащие) с первичным ключом, ограничением ссылочной целостности для **dept** и ограничением ссылочной целостности для emp
- **projects** (проекты) с первичным ключом и ограничением ссылочной целостности для **emp**
- **t1** с трехстолбцовым первичным ключом
- **t2** с трехстолбцовым первичным ключом и с трехстолбцовым внешним ключом для **t1**
- **t3** с трехстолбцовым первичным ключом и с трехстолбцовым внешним ключом для **t2**

После создания этих таблиц попытаемся выполнить следующий оператор UPDATE:

```
SQL> UPDATE dept SET deptno = deptno + 1;
UPDATE dept SET deptno = deptno + 1
       *
ERROR at line 1:
ORA-02292: integrity constraint (UCDEMO.SYS_C00815) violated -
           child record found
```

(ОШИБКА в строке 2:
ORA-02292: нарушено ограничение целостности (UCDEMO.SYS_C00815) –
найдена дочерняя запись. – *Прим. пер.*)

Причиной неудачного выполнения этого оператора является наличие ограничения целостности, связывающего таблицы **emp** и **dept**. Каскадное обновление будет выполнено надлежащим образом, если вызвать **update_cascade.on_table** и задать тот же самый оператор UPDATE.

```
SQL> exec update_cascade.on_table('dept');
PL/SQL procedure successfully completed.

SQL> SELECT * FROM dept;
  DEPTNO DNAME        LOC
-------- ------------ ----------------------
    10 ACCOUNTING   NEW YORK
    20 RESEARCH     DALLAS
    30 SALES        CHICAGO
    40 OPERATIONS   BOSTON

SQL> SELECT empno, deptno FROM emp WHERE deptno = 10;
  EMPNO  DEPTNO
------- ---------
  7839      10
  7782      10
  7934      10
SQL> UPDATE dept SET deptno = deptno + 1;
4 rows updated.

SQL> SELECT * FROM dept;
  DEPTNO DNAME        LOC
-------- ------------ ------------------------
      11 ACCOUNTING   NEW YORK
      21 RESEARCH     DALLAS
      31 SALES        CHICAGO
      41 OPERATIONS   BOSTON

SQL> SELECT empno, deptno FROM emp WHERE deptno IN (10, 11);
  EMPNO  DEPTNO
------- --------
  7839   11
  7782   11
  7934   11
```

## unindex.sql

Если для внешнего ключа не создан индекс, то каскадное обновление может выполняться неудовлетворительно. Для примера рассмотрим таблицу **emp**, создаваемую следующим образом:

```
CREATE TABLE emp (
  EMPNO NUMBER(4) primary key,
```

```
ENAME VARCHAR2 (10),
JOB VARCHAR2 (9),
MGR NUMBER(4) references emp,
HIREDATE DATE,
SAL NUMBER(7, 2),
COMM NUMBER(7,2),
DEPTNO NUMBER(2) references dept);
```

Обратите внимание на то, что для **mgr** и **deptno** индекс не создан. Теперь, если выполнить оператор

```
UPDATE emp SET empno = empno + 1;
```

то генерируемые триггеры будут в итоге выдавать оператор следующего вида:

```
UPDATE emp
  SET mgr = <новое значение>
  WHERE mgr <старое значение>
```

Предположим, что индекса для **mgr** нет. В этом случае, чтобы выполнить приведенный оператор, требуется полный просмотр таблицы. Более того, этот оператор будет выполняться для каждой строки таблицы emp, обновленной первым оператором.

Можно увидеть эту ситуацию при выполнении **unindex.sql** из схемы **ucdemo** (после удаления индекса для **mgr**):

```
SQL> @unindex
STAT  TABLE_NAME              COLUMNS          COLUMNS
----  ----------------------  ---------------  ---------------
****  EMP                     MGR
Ok    EMP                     DEPTNO           DEPTNO
Ok    PROJECTS                EMPNO            EMPNO, PROJ_NO
Ok    T2                      A                A, B
Ok    T3                      A, B             A, B, C
```

**** показывает, что для повышения производительности необходим индекс.

## generate.sql

Это простой сценарий, при выполнении которого создается список вызовов **update_cascade.on_table**, необходимых для всех родительских таблиц схемы. Например, для демонстрационной схемы будет получен следующий результат:

```
SQL> @generate
prompt Update Cascade on table: DEPT
execute update_cascade.on_table( 'DEPT' )

prompt Update Cascade on table: EMP
execute update_cascade.on_table( 'EMP' )

prompt Update Cascade on table: T1
execute update_cascade.on_table( 'T1' )

prompt Update Cascade on table: T2
execute update_cascade.on_table( 'T2' )
```

# Функционирование утилиты

Модуль **update_cascade** генерирует три триггера, осуществляющих каскадное обновление:

- Триггер BEFORE UPDATE используется для сброса некоторых переменных модуля
- Строковый триггер BEFORE UPDATE используется для фиксирования в таблицах PL/SQL значения первичных ключей до и после выполнения операции обновления; он также отменяет обновление первичного ключа

■ Триггер AFTER UPDATE выполняет следующие действия:

1. Размножает родительские записи с их новыми первичными ключами, выполняя оператор вида:

```
INSERT INTO родительская_таблица
SELECT новый_ключ, другие_столбцы
FROM родительская_таблица
WHERE текущий_ключ = ( SELECT старый_ключ FROM dual );
```

Например, при выполнении оператора

```
UPDATE dept SET deptno = deptno + 1;
```

в таблицу dept вносятся значения 11, 21, 31, 41. Значение 11 вносится в столбцы, в которых было 10, 21 — в столбцы, где было 20, и т.д.

2. Если **p_preserve_rowids** = TRUE, то первичные ключи порожденной и исходной строк меняются местами. Например, если задать:

```
UPDATE dept SET deptno = 11 WHERE deptno = 10;
```

то 10 изменяется на новое значение 11, а 11 изменяется на старое значение 10.

3. Устанавливает новые родительские записи для дочерних записей всех подчиненных таблиц, что аналогично выполнению оператора:

```
UPDATE дочерняя_таблица
SET внешний_ключ = ( SELECT новый_ключ FROM dual )
WHERE внешний_ключ = ( SELECT старый_ключ FROM dual );
```

4. Удаляет размноженные родительские записи или запись, содержащую старое значение первичного ключа.

После вызова "update_cascade.on_table('dept', TRUE, TRUE)" генерируется текст программы. Для пояснения функций, выполняемых каждой частью программы, в этот текст включены комментарии, которые в реальном тексте отсутствуют.

В генерируемом программном тексте, выделенном жирным шрифтом, идентификаторы строк сохраняются. Если сохранение идентификаторов строк было бы запрещено, то этот текст отсутствовал бы в создаваемом модуле.

Имя модуля всегда обозначается как u || ИМЯ_ТАБЛИЦЫ || p. Использование разных регистров символов предотвращает конфликты с объектами других пользователей.

```
create or replace package "uDEPTp"
as
  -- Счетчик строк rowCnt используется для сбора информации о числе
  -- строк, обрабатываемых данным оператором UPDATE. Счетчик
  -- сбрасывается в подпрограмме uDEPTp.reset триггером BEFORE
  -- UPDATE. Переменная inTrigger используется для предотвращения
  -- рекурсивной активизации триггеров, когда p_preserve_rowids =
  -- TRUE.
  rowCnt number default 0;
  inTrigger boolean default FALSE;

  -- Для каждого элемента первичного ключа создаются табличный тип,
  -- а затем массивы этого типа. В массивах хранятся значения до
  -- обновления и значения после обновления. Пустой массив
  -- используется для обнуления двух предыдущих массивов.

  type C1_type is table of "DEPT"."DEPTNO"%type
      index by binary_integer;

  empty_C1  C1_type;
  old_C1  C1_type;
  new_C1  C1_type;
```

```
    -- Подпрограмма reset активизируется триггером BEFORE UPDATE,
    -- который сбрасывает переменную rowCnt и очищает массивы,
    -- заполненные в результате предшествующих вызовов.
    procedure reset;

    -- Процедура do_cascade, будучи основной рабочей подпрограммой,
    -- выполняет реальное каскадное обновление данных после
    -- активизации ее из триггера AFTER UPDATE.
    procedure do_cascade;

    -- Процедура add_entry просто увеличивает счетчик строк и
    -- собирает все значения первичных ключей до и после обновления.
    -- Она также отменяет обновление первичных ключей, обращаясь к
    -- переменным :new и :old.
    procedure add_entry
        (
    p_old_C1 in "DEPT"."DEPTNO"%type,
    p_new_C1 in out "DEPT"."DEPTNO"%type
    );
end "uDEPTp";
```

Ниже приведено генерируемое тело модуля, реализующее рассмотренное описание:

```
create or replace package body "uDEPTp"
as
  procedure reset is
  begin
      -- Эта строка присутствует во всех подпрограммах, где
      -- p_preserve_rowids = TRUE. Она предотвращает рекурсивную
      -- активизацию триггеров.
      if ( inTrigger ) then return; end if;

      rowCnt := 0;
      old_C1 := empty_C1;
      new_C1 := empty_C1;
  end reset;

  procedure add_entry
  (
      p_old_C1 in "DEPT"."DEPTNO"%type,
      p_new_C1 in out "DEPT"."DEPTNO"%type
  ) is
  begin
      if ( inTrigger ) then return; end if;
      -- Значения до и после обновления сохраняются в таблицах PL/SQL;
      -- кроме того, отменяется обновление первичного ключа —
      -- значения новых столбцов изменяются на значения старых.
      If (
        p_old_C1 <> p_new_C1
        ) then
        rowCnt := rowCnt + 1;
        old_C1( rowCnt ) := p_old_C1;
        new_C1( rowCnt ) := p_new_C1;
        p_new_C1 := p_old_C1;
      end if;
```

```
end add_entry;

procedure do_cascade is
begin
    if ( inTrigger ) then return; end if;
    inTrigger := TRUE;
    -- Выполняется размножение и каскадное удаление каждой
    -- обновленной строки.
    for i in 1 .. rowCnt loop
      -- Выполняется вставка размноженных строк, где старым
      -- значениям соответствуют новые первичные ключи.
      insert into DEPT ("DEPTNO", "DNAME", "LOC")
        select new_C1(i), "DNAME", "LOC"
          from "DEPT" a
            where ( "DEPTNO" ) =
              ( select old_C1(i) from dual );

      -- Этот программный текст создается только в том случае,
      -- если p_preserve_rowids = TRUE. Старые и новые первичные
      -- ключи меняются местами, тем самым сохраняя идентификатор
      -- исходной родительской строки.
      update "DEPT" set
        ( "DEPTNO" ) =
        ( select
            decode ( "DEPTNO", old_C1(i), new_C1(i), old_C1(i) )
          from dual )
        Where   ( "DEPTNO" ) =
                ( select new_C1(i)
                    from dual )
          OR    ( "DEPTNO" ) =
                ( select old_C1(i)
                    from dual );

      -- Выполняется каскадное обновление всех дочерних таблиц.
      update "EMP" set
      ( "DEPTNO" ) =
      ( select new_C1(i)
          from dual )
      where ( "DEPTNO" ) =
            ( select old_C1(i)
                from dual );

      -- Удаляется старое значение первичного ключа.
      delete from "DEPT"
        where ( "DEPTNO" ) =
              ( select old_C1(i)
                  from dual );
    end loop;
    inTrigger := FALSE;
    reset;
exception
    when others then
      inTrigger := FALSE;
      reset;
```

```
            raise;
        end do_cascade;
    end "uDEPTp";
```

Наконец, созданы три триггера для родительской таблицы, которые и осуществляют каскадное обновление. Первый триггер сбрасывает переменные модуля, приведенного выше:

```
create or replace trigger "uc$DEPT_bu"
before update of
      "DEPTNO"
on "DEPT"
begin "uDEPTp".reset; end;
```

*Следующий, строковый, триггер вызывает* **add_entry** *для каждой изменяемой строки:*

```
create or replace trigger "uc$DEPT_bufer"
before update of
      "DEPTNO"
on "DEPT"
for each row
begin
  "uDEPTp".add_entry(
      :old."DEPTNO",
      :new."DEPTNO"
    );
end;
```

Последний триггер вызывает **do_cascade** и вносит необходимые изменения:

```
create or replace trigger "uc$DEPT_au"
after update of
      "DEPTNO"
on "DEPT"
begin "uDEPTp".do_cascade; end;
```

# Итоги

Итак, триггеры являются очень важным элементом PL/SQL и Oracle. Их можно использовать для реализации ограничений данных, более сложных, чем обычные ограничения ссылочной целостности. Триггерами завершается обсуждение именованных блоков PL/SQL, продолжавшееся на протяжении трех последних глав. В следующей главе мы перейдем к рассмотрению процесса обработки исключительных ситуаций в PL/SQL.

# Глава 10

## Обработка ошибок

В любой грамотной программе должна существовать возможность надлежащей обработки, а при необходимости – устранения ошибок. В PL/SQL обработка ошибок реализуется с помощью *исключительных ситуаций* (exceptions) и *обработчиков исключительных ситуаций* (exception handlers). Исключительные ситуации могут быть связаны с ошибками Oracle или с ошибками, определяемыми пользователями. В этой главе описан синтаксис исключительных ситуаций и их обработчиков, а также приводятся правила их передачи. В конце главы даны рекомендации по использованию исключительных ситуаций.

# Понятие исключительной ситуации

В основе PL/SQL лежит язык программирования Ada, одним из свойств которого, привнесенным в PL/SQL, является механизм исключительных ситуаций. Используя этот механизм, написанные на PL/SQL программы становятся гораздо надежнее, и во время их выполнения предоставляется возможность обработки как запланированных, так и незапланированных ошибок.

Классификация ошибок, возникающих в программах PL/SQL, приведена в таблице 10.1. Исключительные ситуации создаются только для ошибок процесса выполнения программ, но не для ошибок компиляции. Ошибки времени компиляции распознаются системой поддержки PL/SQL, и сообщения о них передаются пользователям. Такие ошибки программа обработать не может, поскольку на этом этапе она еще не выполняется. Ниже приведен блок, в котором возникает ошибка компиляции:

```
PLS-201: identifier 'SSTUDENTS' must be declared
```

(идентификатор 'SSTUDENTS' должен быть описан. – *Прим. пер.*)

так как в операторе выбора неверно записан идентификатор students:

```
DECLARE
  v_NumStudents NUMBER;
BEGIN
  SELECT COUNT(*)
      INTO v_NumStudents
      FROM sstudents;
END;
```

**ТАБЛИЦА 10.1. Типы ошибок PL/SQL**

| Вид ошибки | Источник сообщения | Обработка |
|---|---|---|
| Этапа компиляции | Компилятор PL/SQL | Интерактивно: компилятор сообщает об ошибках, а пользователь их исправляет |
| Этапа выполнения | Система поддержки PL/SQL | Программно: исключительные ситуации устанавливаются и распознаются обработчиками исключительных ситуаций |

Исключительные ситуации и их обработчики – это метод, при помощи которого программа реагирует на ошибки процесса выполнения и устраняет их. В число ошибок процесса выполнения входят ошибки SQL, например:

```
ORA-0001: unique constraint violated
```

(нарушение ограничения уникальности. – *Прим. пер.*)

и процедурные ошибки, например:

```
ORA-06502: PL/SQL: numeric or value error
```

(ошибка числа или значения. – *Прим. пер.*)

В случае возникновения ошибки устанавливается (raised) исключительная ситуация. При этом управление программой передается обработчику, который является отдельным фрагментом программы. Отделение обработки ошибок от остальной части программы упрощает понимание логической структуры программы и обеспечивает фиксацию всех ошибок.

Для обеспечения обработки всех ошибок в тех языках программирования, в которых не используется модель исключительных ситуаций (например, в языке C), программы должны содержать операторы, реализующие эту обработку. Например:

```
int x, y, z;
f(x); /* Вызов функции; x передается в качестве аргумента. */
if <ошибка>
  обработчик_ошибки (...);
y = 1 / z;
if <ошибка>
  обработчик_ошибки (...);
z = x + y;
if <ошибка>
  обработчик_ошибки (...);
```

Обратите внимание: проверка ошибок должна выполняться после каждого оператора программы. Если не внести в программу такую проверку, то ошибка не будет обработана надлежащим образом. Кроме того, операторы, реализующие обработку ошибок, засоряют программу и затрудняют понимание ее логической структуры. Сравним программу, приведенную в предыдущем примере, с аналогичной программой, написанной на PL/SQL:

```
DECLARE
  x NUMBER;
  y NUMBER;
  z NUMBER;
BEGIN
  f(x);
  y := 1 / z;
  z = x + y;
EXCEPTION
  WHEN OTHERS THEN
      /* Обработчик всех ошибок. */
  (...);
END;
```

Заметьте, что раздел обработки ошибок отделен от логической схемы программы: это устраняет трудности, возникающие в случае использования программы на языке C:

- Логика программы наглядна и тем самым легче для понимания
- Программа распознает и обработает любую ошибку, независимо от того, какой оператор является причиной этой ошибки

## Объявление исключительных ситуаций

Исключительные ситуации описываются в разделе объявлений блока, устанавливаются в выполняемом разделе, а обрабатываются в разделе исключительных ситуаций. Существует два вида исключительных ситуаций: *определяемые пользователями* и *стандартные, или предопределенные.*

### Исключительные ситуации, определяемые пользователями

Исключительная ситуация, определяемая пользователем, обозначает такую ошибку, которая описывается в программе, причем совсем не обязательно, чтобы эта ошибка была ошибкой Oracle, – она может быть, например, ошибкой данных. Стандартные же исключительные ситуации соответствуют типичным ошибкам SQL.

Исключительные ситуации, определяемые пользователями, описываются в разделе объявлений блока PL/SQL. Как и переменные, исключительные ситуации имеют собственный тип (EXCEPTION) и области действия. Например:

```
DECLARE
  e_TooManyStudents EXCEPTION;
```

**e_TooManyStudents** – это идентификатор, который виден во всем блоке. Заметьте, что область действия исключительной ситуации совпадает с областью действия переменной или курсора, описанных в том же разделе объявлений. О правилах, определяющих область действия и область видимости идентификаторов PL/SQL, рассказано в главе 2.

## Стандартные исключительные ситуации

В Oracle существует ряд исключительных ситуаций, которые соответствуют типичным ошибкам Oracle. Как и для стандартных типов (NUMBER, VARCHAR2 и т.д.), идентификаторы таких исключительных ситуаций описаны в модуле STANDARD (более подробно о стандартных типах и о модуле STANDARD рассказано в главе 2). Поэтому данные идентификаторы уже доступны программе, и их не надо описывать в разделе объявлений, в отличие от исключительных ситуаций, определяемых пользователями. Стандартные исключительные ситуации приведены в таблице 10.2.

### ТАБЛИЦА 10.2. Стандартные исключительные ситуации

| Ошибка Oracle | Соответствующая исключительная ситуация | Описание |
|---|---|---|
| ORA-0001 | DUP_VAL_ON_INDEX | Нарушено ограничение уникальности |
| ORA-0051 | TIMEOUT_ON_RESOURCE | Блокировка по времени при ожидании ресурса |
| ORA-0061 | TRANSACTION_BACKED_OUT[1] | Произведен откат транзакции при возникновении тупика, или взаимоблокировки транзакций |
| ORA-1001 | INVALID_CURSOR | Запрещенная операция над курсором |
| ORA-1012 | NOT_LOGGED_ON | Отсутствует соединение с Oracle |
| ORA-1017 | LOGIN_DENIED | Неверные имя/пароль пользователя |
| ORA-1403 | NO_DATA_FOUND | Данные не найдены |
| ORA-1422 | TOO_MANY_ROWS | Оператор SELECT..INTO возвращает более одной строки |
| ORA-1476 | ZERO_DIVIDE | Деление на ноль |
| ORA-1722 | INVALID_NUMBER | Неудачная попытка преобразования к типу NUMBER; например 1A является запрещенным значением |
| ORA-6500 | STORAGE_ERROR | Внутренняя ошибка PL/SQL; устанавливается, если PL/SQL недостаточно памяти |
| ORA-6501 | PROGRAM_ERROR | Внутренняя ошибка PL/SQL |
| ORA-6502 | VALUE_ERROR | Ошибка усечения, арифметическая ошибка или ошибка преобразования |
| ORA-6504 | ROWTYPE_MISMATCH[2] | Базовая курсорная переменная и курсорная переменная PL/SQL имеют несовместимые типы строк |
| ORA-6511 | CURSOR_ALREADY_OPEN | Попытка открыть курсор, который уже открыт |
| ORA-6530 | ACCESS_INTO_NULL[3] | Попытка присвоить значение атрибуту NULL-объекта |
| ORA-6531 | COLLECTION_IS_NULL[3] | Попытка использовать для таблицы или изменяемого массива PL/SQL типа NULL метод сборных конструкций, отличный от EXISTS |
| ORA-6532 | SUBSCRIPT_OUTSIDE_LIMIT[3] | Ссылка на индекс таблицы или изменяемого массива, лежащий вне объявленного диапазона (например -1) |
| ORA-6533 | SUBSCRIPT_BEYOND_COUNT[3] | Ссылка на индекс таблицы или изменяемого массива, больший, чем число элементов данной сборной конструкции |

[1]Только для PL/SQL 2.0 и 2.1.

[2]Для PL/SQL 2.2 и выше.

[3]Для PL/SQL 8.0 и выше.

Краткое описание некоторых стандартных исключительных ситуаций приведено ниже. Более подробно об этих ошибках рассказано в приложении С.

**INVALID_CURSOR** Эта ошибка возникает в случае выполнения над курсором запрещенной операции, например при попытке закрыть уже закрытый курсор. Попытка открыть уже открытый курсор вызывает установление исключительной ситуации CURSOR_ALREADY_OPEN.

**NO_DATA_FOUND** Эта исключительная ситуация устанавливается в двух случаях. Первый случай — когда оператор SELECT..INTO не возвращает ни одной строки. Если этот оператор возвращает более одной строки, устанавливается исключительная ситуация TOO_MANY_ROWS. Второй случай — попытка обращения к элементу таблицы PL/SQL, которому не присвоено значение. Ниже приведен анонимный блок, в котором устанавливается исключительная ситуация NO_DATA_FOUND:

```
DECLARE
  TYPE t_NumberTableType IS TABLE OF NUMBER
      INDEX BY BINARY_INTEGER;
  v_NumberTable t_NumberTableType;
  v_TempVar NUMBER;
BEGIN
  v_TempVar := v_NumberTable(1);
END;
```

**INVALID_NUMBER** Эта исключительная ситуация устанавливается в SQL-операторах при неудачной попытке преобразования строки символов в число. В процедурных операторах вместо нее устанавливается другая исключительная операция — VALUE_ERROR, например, в следующем операторе — INVALID_NUMBER, так как 'X' не является числом:

```
INSERT INTO students (id, first_name, last_name)
  VALUES ('X', 'SCOTT', 'Smith');
```

**STORAGE_ERROR или PROGRAM_ERROR** Это внутренние исключительные ситуации, которые обычно не устанавливаются. Причиной их возникновения является либо нехватка памяти (STORAGE_ERROR), либо внутренняя ошибка PL/SQL (PROGRAM_ERROR). Внутренние ошибки часто бывают вызваны сбоями в работе системы PL/SQL, и о них следует сообщать в службу технической поддержки корпорации Oracle (Oracle Technical Support).

**VALUE_ERROR** Эта исключительная ситуация устанавливается при появлении арифметической ошибки, ошибки преобразования, усечения или ограничения в процедурном операторе. Если же ошибка возникает в SQL-операторе, то устанавливается исключительная ситуация INVALID_NUMBER. Причиной возникновения ошибки VALUE_ERROR является либо операция присваивания, либо оператор SELECT..INTO. В обоих рассмотренных ниже примерах устанавливается VALUE_ERROR:

```
DECLARE
  v_TempVar VARCHAR2(3);
BEGIN
  v_TempVar := 'ABCD';
END;

DECLARE
  v_TempVar NUMBER(2);
BEGIN
  SELECT id
      INTO v_TempVar
      FROM students
      WHERE last_name = 'Smith';
END;
```

PL/SQL 2.2 ... и ВЫШЕ

**ROWTYPE_MISMATCH** Эта исключительная ситуация устанавливается при несоответствии типов базовой курсорной переменной и курсорной переменной PL/SQL. Например, ROWTYPE_MISMATCH устанавливается при несоответствии фактического и формального типов, которые возвращаются процедурой, использующей в качестве аргумента курсорную переменную. В главе 6 более подробно рассказывается о курсорных переменных, а также приводится пример, где устанавливается эта исключительная ситуация.

## Установление исключительных ситуаций

Когда происходит ошибка, связанная с некоторой исключительной ситуацией, устанавливается эта исключительная ситуация. Исключительные ситуации, определяемые пользователями, устанавливаются явно при помощи оператора RAISE, в то время как стандартные исключительные ситуации — неявно при возникновении соответствующих ошибок Oracle. Однако при желании и стандартные исключительные ситуации можно устанавливать с помощью оператора RAISE. Если усложнить пример, рассмотренный в разделе "Исключительные ситуации, определяемые пользователями", он примет следующий вид:

```
-- Этот пример содержится в файле handle.sql.
DECLARE
  e_TooManyStudents EXCEPTION;  -- Исключительная ситуация для
                                -- указания условия ошибки.
  v_CurrentStudents NUMBER(3);  -- Текущее число студентов,
                                -- зарегистрированных в HIS-101.
  v_MaxStudents NUMBER(3);      -- Максимальное число студентов в HIS-101.

BEGIN
  /* Определим текущее число зарегистрированных студентов и
     максимальное число студентов. */
  SELECT current_students, max_students
      INTO v_CurrentStudents, v_MaxStudents
      FROM classes
      WHERE department = 'HIS' AND course = 101;
  /* Сравним полученные значения. */
  IF v_CurrentStudents > v_MaxStudents THEN
      /* Зарегистрировано слишком много студентов -- установим
          исключительную ситуацию. */
      RAISE e_TooManyStudents;
  ENF IF;
END;
```

При установлении исключительной ситуации управление программой сразу же передается разделу исключительных ситуаций блока. Если такого раздела нет, исключительная ситуация передается блоку, в который входит данный блок (см. ниже раздел "Передача исключительных ситуаций"). После передачи управления обработчику вернуться в выполняемый раздел блока невозможно. Вышесказанное иллюстрирует рис. 10.1.

Стандартные ситуации устанавливаются автоматически при возникновении соответствующей ошибки Oracle. Например, в этом блоке PL/SQL устанавливается исключительная ситуация DUP_VAL_ON_INDEX:

```
BEGIN
  INSERT INTO students (id, first_name, last_name)
      VALUES (10001, 'John', 'Smith');
  INSERT INTO students (id, first_name, last_name)
      VALUES (10001, 'Susan', 'Ryan');
END;
```

Исключительная ситуация устанавливается благодаря тому, что столбец **id** таблицы students является первичным ключом, и поэтому для него задается ограничение уникальности. При попытке второго оператора INSERT ввести значение 10001 в этот столбец происходит ошибка:

```
ORA-0001: unique constraint violated
```

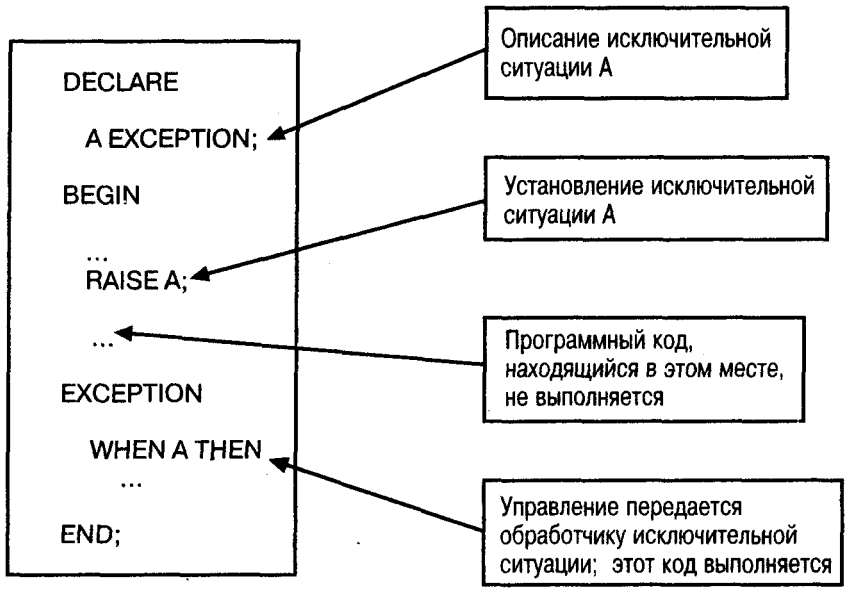
(нарушение ограничения уникальности. — *Прим. пер.*)

Эта ошибка соответствует исключительной ситуации DUP_VAL_ON_INDEX.

## Обработка исключительных ситуаций

Как показано на рис. 10.1, при установлении исключительной ситуации управление программой передается разделу исключительных ситуаций блока. В этот раздел входят *обработчики* (handlers) для всех

**Рис. 10.1.**
*Передача управления обработчику исключительной ситуации*




исключительных ситуаций. В обработчике содержится программный текст, выполняющийся при возникновении ошибки и последующем установлении данной исключительной ситуации. Ниже приведен синтаксис этого раздела:

**EXCEPTION**
**WHEN** *имя_исключительной_ситуации1* **THEN**
*последовательность_операторов1;*
**WHEN** *имя_исключительной_ситуации2* **THEN**
*последовательность_операторов2;*
**WHEN** OTHERS **THEN**
*последовательность_операторов3;*
**END;**

Каждый обработчик состоит из условия WHEN (когда) и операторов, выполняющихся при установлении исключительной ситуации. В условии WHEN указывается, для какой исключительной ситуации предназначен данный обработчик. Продолжим рассмотренный выше пример:

```
DECLARE
  e_TooManyStudents EXCEPTION;  -- Исключительная ситуация для
                                -- указания условия ошибки.
  v_CurrentStudents NUMBER(3);  -- Текущее число студентов,
                                -- зарегистрированных в HIS-101.
  v_MaxStudents NUMBER(3);      -- Максимальное число студентов в
                                -- HIS-101.
BEGIN
  /* Определим текущее число зарегистрированных студентов и
     максимальное число студентов. */
  SELECT current_students, max_students
      INTO v_CurrentStudents, v_MaxStudents
      FROM classes
      WHERE department = 'HIS' AND course = 101;
  /* Сравним полученные значения. */
  IF v_CurrentStudents > v_MaxStudents THEN
      /* Зарегистрировано слишком много студентов - установим
```

```
        исключительную ситуацию. */
      RAISE e_TooManyStudents;
    ENF IF;
  EXCEPTION;
    WHEN e_TooManyStudents THEN
      /* Обработчик, выполняющийся в том случае, когда в HIS-101
         зарегистрировано слишком много студентов. Введем сообщение,
         поясняющее сложившуюся ситуацию. */
      INSERT INTO log_table (info) VALUES ('History 101 has' ||
        v_CurrentStudents || 'students: max allowed is' ||
        v_MaxStudents);
  END;
```

Один обработчик может выполняться в нескольких исключительных ситуациях, для чего нужно просто перечислить их имена в условии WHEN, отделив одно от другого ключевым словом OR (или):

```
EXCEPTION
  WHEN NO_DATA_FOUND OR TOO_MANY_ROWS THEN
    INSERT INTO log_table (info) VALUES ('A select error occured.');
END;
```

## Обработчик OTHERS

Обработчик OTHERS (другие) выполняется для всех оставшихся исключительных ситуаций. Он всегда должен быть последним обработчиком в блоке. Обработчик OTHERS рекомендуется указывать на самом высоком уровне программы (в самом внешнем блоке) для обеспечения распознавания всех без исключения ошибок. Добавим в рассматриваемый пример обработчик OTHERS:

```
-- Этот пример содержится в файле others.sql.
DECLARE
  e_TooManyStudents EXCEPTION;  -- Исключительная ситуация для
                                -- указания условия ошибки.
  v_CurrentStudents NUMBER(3);  -- Текущее число студентов,
                                -- зарегистрированных в HIS-101.
  v_MaxStudents NUMBER(3);      -- Максимальное число студентов в
                                -- HIS-101.
BEGIN
  /* Определим текущее число зарегистрированных студентов и
     максимальное число студентов. */
  SELECT current_students, max_students
    INTO v_CurrentStudents, v_MaxStudents
    FROM classes
    WHERE department = 'HIS' AND course = 101;
  /* Сравним полученные значения. */
  IF v_CurrentStudents > v_MaxStudents THEN
    /* Зарегистрировано слишком много студентов — установим
       исключительную ситуацию. */
    RAISE e_TooManyStudents;
  ENF IF;
EXCEPTION;
  WHEN e_TooManyStudents THEN
    /* Обработчик, выполняющийся в том случае, когда в HIS-101
       зарегистрировано слишком много студентов. Введем сообщение,
       поясняющее сложившуюся ситуацию. */
    INSERT INTO log_table (info) VALUES ('History 101 has' ||
      v_CurrentStudents || 'students: max allowed is' ||
      v_MaxStudents);
```

```
    WHEN OTHERS THEN
        /* Обработчик, выполняющийся для всех других ошибок. */
        INSERT INTO log_table (info) VALUES ('Another error occurred');
END;
```

Обработчик OTHERS из этого примера просто регистрирует факт возникновения ошибки, но не определяет ее. Ошибку, приведшую к установлению исключительной ситуации, обрабатываемой OTHERS, можно узнать при помощи предопределенных функций SQLCODE и SQLERRM, описанных ниже.

**SQLCODE и SQLERRM** В обработчике OTHERS часто бывает полезно знать, какая ошибка Oracle установила исключительную ситуацию. В этом случае можно регистрировать не только факт возникновения ошибки, но и ее тип. Иногда нужно знать тип ошибки для выполнения конкретных действий. В PL/SQL такие сведения получают при помощи двух встроенных функций SQLCODE и SQLERRM. SQLCODE возвращает код текущей ошибки, а SQLERRM — текст сообщения о данной ошибке.

▼ ВНИМАНИЕ **Функция DBMS_UTILITY.FORMAT_ERROR_STACK также возвращает код текущей ошибки и может быть использована в дополнение к функции SQLERRM. Это иллюстрируется в примере раздела "PL/SQL в работе".**

Ниже приведен полный текст блока PL/SQL, который создавался на протяжении этой главы, с использованием обработчика исключительных ситуаций OTHERS.

```
-- Этот пример содержится в файле sqlerrm.sql.
DECLARE
  e_TooManyStudents EXCEPTION;  -- Исключительная ситуация для
                                -- указания условия ошибки.
  v_CurrentStudents NUMBER(3);  -- Текущее число студентов,
                                -- зарегистрированных в HIS-101.
  v_MaxStudents NUMBER(3);      -- Максимальное число студентов в
                                -- HIS-101.

  v_ErrorCode NUMBER;           -- Переменная для хранения кода
                                -- ошибки.
  v_ErrorText VARCHAR2(200);    -- Переменная для хранения текста
                                -- сообщения об ошибке.

BEGIN
  /* Определим текущее число зарегистрированных студентов и
     максимальное число студентов. */
  SELECT current_students, max_students
      INTO v_CurrentStudents, v_MaxStudents
      FROM classes
      WHERE department = 'HIS' AND course = 101;
  /* Сравним полученные значения. */
  IF v_CurrentStudents > v_MaxStudents THEN
      /* Зарегистрировано слишком много студентов - установим
          исключительную ситуацию. */
      RAISE e_TooManyStudents;
  ENF IF;
EXCEPTION
  WHEN e_TooManyStudents THEN
      /* Обработчик, выполняющийся в случае, когда в HIS-101
          зарегистрировано слишком много студентов. Введем сообщение,
          поясняющее сложившуюся ситуацию. */
      INSERT INTO log_table (info) VALUES ('History 101 has' ||
        v_CurrentStudents || 'students: max allowed is' ||
        v_MaxStudents);
```

```
    WHEN OTHERS THEN
        /* Обработчик, выполняющийся для всех других ошибок. */
        v_ErrorCode := SQLCODE;
        v_ErrorText := SUBSTR(SQLERRM, 1, 200);    -- Обратите внимание на
                                                   -- использование SUBSTR.
        INSERT INTO log_table (code, message, info) VALUES
          (v_ErrorCode, v_ErrorText, 'Oracle error occurred');
END;
```

Максимальная длина сообщения об ошибке Oracle составляет 512 символов. В рассматриваемом примере переменная **v_ErrorText** ограничена 200 символами (для соответствия полю **code** таблицы log_table). Если текст сообщения об ошибке превышает 200 символов, операция

```
v_ErrorText := SQLERRM;
```

сама вызывает установление стандартной исключительной ситуации VALUE_ERROR. Для избежания этого используется встроенная функция SUBSTR, обеспечивающая присваивание переменной **v_ErrorText** не более чем 200 символов. О SUBSTR и о других встроенных функциях PL/SQL рассказано в главе 5.

Заметьте, что значения функций SQLCODE и SQLERRM сначала присваиваются локальным переменным, и только потом эти переменные указываются в SQL-операторе. Данные функции являются процедурными, поэтому их нельзя использовать непосредственно в SQL-операторе.

Функцию SQLERRM можно вызывать с аргументом, представляющим собой некоторое число. В этом случае она возвращает текст сообщения об ошибке, код которой равен заданному числу. Аргумент должен всегда быть отрицателен. Если аргумент SQLERRM равен нулю, то возвращается сообщение

```
ORA-0000: normal, succesful completion
```

(нормальное, успешное завершение. — *Прим. пер.*)

Если SQLERRM вызывается с любым положительным значением, отличным от +100, то выдается сообщение

```
User-Defined Exception
```

(исключительная ситуация, определяемая пользователем. — *Прим. пер.*)

SQLERRM(100) возвращает

```
ORA-1403: no data found
```

(данные не найдены. — *Прим. пер.*)

При вызове в обработчике функции SQLCODE она возвращает отрицательное число, обозначающее ошибку Oracle. Единственным исключением является ошибка "ORA-1403: no data found", когда SQLCODE возвращает +100.

Если SQLERRM без аргументов вызывается из выполняемого раздела блока, она всегда возвращает сообщение

```
ORA-0000: normal, succesful completion.
```

(нормальное, успешное завершение. — *Прим. пер.*)

а SQLCODE возвращает 0. Все эти ситуации показаны в следующем примере:

```
-- Этот пример содержится в файле sqlerrmz2.sql.
DECLARE
  v_ErrorText log_table.message%TYPE;   -- Переменная для хранения
                                        -- текста сообщения об ошибке.
BEGIN
  /* SQLERRM(0). */
  v_ErrorText := SUBSTR(SQLERRM(0), 1, 200);
  INSERT INTO log_table (code, message, info)
      VALUES (0, v_ErrorText, 'SQLERRM(0)');

  /* SQLERRM(100). */
  v_ErrorText := SUBSTR(SQLERRM(100), 1, 200);
  INSERT INTO log_table (code, message, info)
```

```
        VALUES (100, v_ErrorText, 'SQLERRM(100)');

    /* SQLERRM(10). */
    v_ErrorText := SUBSTR(SQLERRM(10), 1, 200);
    INSERT INTO log_table (code, message, info)
        VALUES (10, v_ErrorText, 'SQLERRM(10)');

    /* SQLERRM без аргументов. */
    v_ErrorText := SUBSTR(SQLERRM, 1, 200);
    INSERT INTO log_table (code, message, info)
        VALUES (NULL, v_ErrorText, 'SQLERRM with no argument');

    /* SQLERRM(-1). */
    v_ErrorText := SUBSTR(SQLERRM(-1), 1, 200);
    INSERT INTO log_table (code, message, info)
        VALUES (-1, v_ErrorText, 'SQLERRM(-1)');

    /* SQLERRM(-54). */
    v_ErrorText := SUBSTR(SQLERRM(-54), 1, 200);
    INSERT INTO log_table (code, message, info)
        VALUES (-54, v_ErrorText, 'SQLERRM(-54)');

END;
```

Если программу из этого примера запустить на выполнение, то в таблице log_table будут содержаться значения, приведенные в таблице 10.3.

**ТАБЛИЦА 10.3. Содержание таблицы log_table**

| Код | Сообщение | Способ использования функции SQLERRM |
|---|---|---|
| 0 | ORA-0000: normal, succesful completion | SQLERRM(0) |
| +100 | ORA-01403: *no data found* | SQLERRM(100) |
| +10 | User-Defined Exception | SQLERRM(10) |
| NULL | ORA-0000: normal, succesful completion | SQLERRM без аргументов |
| -1 | ORA-00001: unique constraint (.) violated | SQLERRM(-1) |
| -54 | ORA-00054: resource busy and acquire with NOWAIT specified | SQLERRM(-54) |

# Прагма EXCEPTION_INIT

Существует возможность связывать именованные исключительные ситуации с конкретными ошибками Oracle, что позволяет обнаруживать эти ошибки непосредственно, а не через обработчик OTHERS. Для этого служит прагма EXCEPTION_INIT. (Более подробно о прагмах и об их применении рассказано в главе 2). Прагма EXCEPTION_INIT используется следующим образом:

PRAGMA EXCEPTION_INIT (*имя_исключительной_ситуации, номер_ошибки_Oracle*);

где *имя_исключительной_ситуации* — это имя исключительной ситуации, описанной до прагмы, а *номер_ошибки_Oracle* — код ошибки, которую нужно связать с этой именованной исключительной ситуацией. Данная прагма должна быть указана в разделе объявлений.

Ниже приведен пример, в котором исключительная ситуация **e_MissingNull** определяется пользователем и устанавливается, если во время выполнения программы происходит ошибка "ORA-1400: mandatory NOT NULL column missing or NULL during insert" (при вводе данных в столбец NOT NULL пропущено значение или указано NULL-значение. — *Прим. пер.*).

```
-- Этот пример содержится в файле pragma.sql.
DECLARE
```

```
    e_MissingNull EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_MissingNull, -1400);
  BEGIN
    INSERT INTO students (id) VALUES (NULL);
  EXCEPTION
    WHEN e_MissingNull then
    INSERT INTO log_table (info) VALUES ('ORA-1400 occurred');
  END;
```

В каждом вхождении прагмы EXCEPTION_INIT можно связывать с ошибкой Oracle только одну исключительную ситуацию, определяемую пользователем. В обработчике этой исключительной ситуации функции SQLCODE и SQLERRM будут возвращать код и сообщение, соответствующие произошедшей ошибке Oracle, а не сообщение, заданное пользователем.

## Использование функции RAISE_APPLICATION_ERROR

Для создания собственных сообщений об ошибках, более содержательных, чем именованные исключительные ситуации, пользователи могут применять функцию RAISE_APPLICATION_ERROR (установить ошибку приложения). Сообщения об ошибках, определяемых пользователями, передаются из блока в вызывающую среду так же, как и сообщения об ошибках Oracle. Синтаксис функции RAISE_APPLICATION_ERROR таков:

RAISE_APPLICATION_ERROR(*номер_ошибки, сообщение_об_ошибке,* [*сохранение_ошибки*]);

где *номер_ошибки* — это параметр, лежащий в диапазоне от -20000 до -20999, *сообщение_об_ошибке* — текст, соответствующий данной ошибке, а *сохранение_ошибки* — логическое значение. Длина параметра *сообщение_об_ошибке* должна быть не больше, чем 512 символов. Логический параметр сохранение_ошибки необязателен. Если он TRUE, то новая ошибка пополняет список ранее произошедших ошибок (при наличии этого списка), а если FALSE, то новая ошибка замещает текущий список ошибок. Для примера рассмотрим процедуру, которая перед регистрацией нового студента проверяет, имеется ли для него свободное место в учебной группе:

```
-- Этот пример содержится в файле register.sql.
CREATE OR REPLACE PROCEDURE Register (
  /* Регистрирует студента, указанного параметром p_StudentID, в
     группе, указанной параметрами p_Department и p_Course. Перед
     вызовом процедуры ClassPackage.AddStudent, которая фактически
     вносит студента в список группы, процедура Register
     проверяет, имеется ли свободное место в группе и существует
     ли эта группа вообще. */
  p_StudentID IN students.id%TYPE,
  p_Department IN classes.department%TYPE,
  p_Course IN classes.course%TYPE) AS

  v_CurrentStudents NUMBER;  -- Текущее число студентов в группе.
  v_MaxStudents NUMBER;      -- Максимальное число студентов в группе.

BEGIN
  /* Определим текущее число зарегистрированных студентов и
     максимальное число студентов в группе. */
  SELECT current_students, max_students
      INTO v_CurrentStudents, v_MaxStudents
      FROM classes
      WHERE course = p_Course
      AND department = p_Department;

  /* Убедимся, что для нового студента в группе имеется свободное
     место. */
  IF v_CurrentStudents + 1 > v_MaxStudents THEN
      RAISE_APPLICATION_ERROR(-20000, 'Can''t add more students to '
```

```
        || p_Department || ' ' || p_Course);
    END IF;

    /* Внесем студента в список группы. */
    ClassPackage.AddStudent(p_StudentID, p_Department, p_Course);

  EXCEPTION
    WHEN NO_DATA_FOUND THEN
        /* Процедуре не передается никакой информации о группе.
           Установим исключительную ситуацию для того, чтобы сообщить
           об этом вызывающей программе. */
        RAISE_APPLICATION_ERROR(-20001, p_Department || ' ' ||
          p_Course || ' doesn''t exist!');
  END Register;
```

В процедуре **Register** функция RAISE_APPLICATION_ERROR используется в двух случаях. Первое, процедура определяет текущее число студентов, зарегистрированных в группе, при помощи оператора SELECT..INTO. Если этот оператор возвращает NO_DATA_FOUND, то управление программой передается обработчику исключительной ситуации, и RAISE_APPLICATION_ERROR применяется для уведомления пользователя о том, что группа не существует. Если же группа существует, то процедура проверяет, имеется ли свободное место для нового студента. Если свободного места нет, то вновь применяется функция RAISE_APPLICATION_ERROR, сообщая пользователю об отсутствии свободного места. И наконец, если свободное место есть, то студент вносится в список группы при помощи модульной процедуры **ClassPackage.AddStudent**, описанной в главе 8.

На рис. 10.2 показаны результаты вызова процедуры **Register** из SQL*Plus. Обратите внимание на результаты выполнения оператора SELECT: можно видеть, что группа History 101 заполнена (**max_students = current_students**). Поэтому при вызове процедуры выдается сообщение о том, что свободного места нет:

```
ORA-20000: Can't add more students to HIS 101
```

(добавить нового студента в HIS 101 невозможно. — *Прим. пер.*)

Сравним результаты выполнения программы, приведенной на рисунке 10.2, с результатами, полученными на рисунке 10.3, где показано выполнение анонимного блока при помощи SQL*Plus. В этом блоке просто устанавливается исключительная ситуация NO_DATA_FOUND. Обратите внимание: при желании и стандартные исключительные ситуации можно устанавливать явным образом. Ошибка выводится на экран:

```
ORA-1403: no data found
```

(данные не найдены. — *Прим. пер.*)

**Рис. 10.2.**
*Результаты вызова процедуры Register*

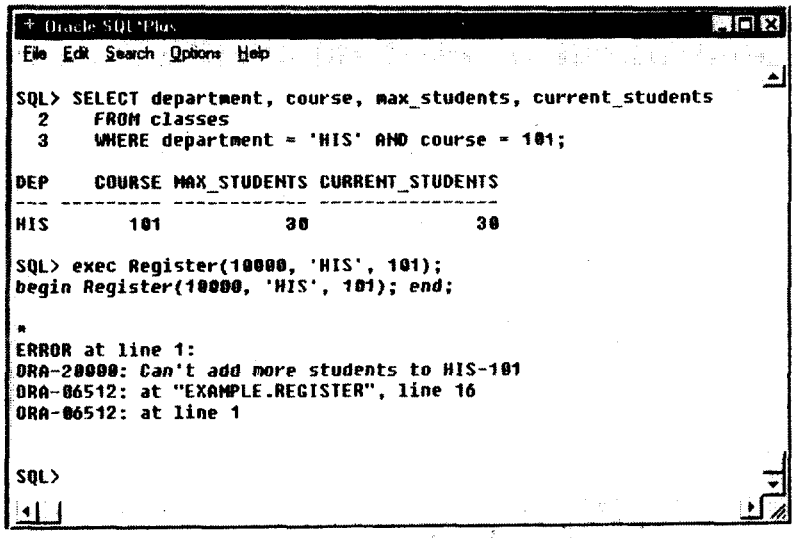
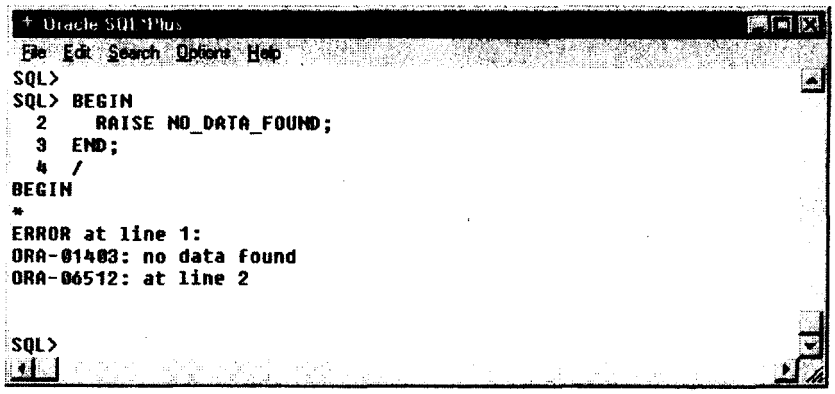


**Рис. 10.3.**

*Результаты выполнения анонимного блока, устанавливающего исключительную ситуацию NO_DATA_FOUND*




Формат полученных результатов одинаков – номер ошибки Oracle и соответствующий текст. В обоих случаях присутствует сообщение ORA-6512, указывающее строку программы, в которой произошла ошибка. Таким образом, можно применять функцию RAISE_APPLICATION_ERROR для сообщения пользователям об условиях возникновения ошибок так, как это делается в случае стандартных ошибок Oracle. Это очень удобно, поскольку для обработки ошибок, определяемых пользователями, не требуются специальные обработчики.

# Передача исключительных ситуаций

Исключительные ситуации можно устанавливать в разделе объявлений, выполняемом разделе или разделе исключительных ситуаций блока PL/SQL. В предыдущем примере было показано, что происходит в том случае, когда исключительная ситуация устанавливается в выполняемом разделе блока и при этом существует обработчик данной исключительной ситуации. Но что произойдет, когда обработчик отсутствует или когда исключительная ситуация устанавливается в другом разделе? Ответ на этот вопрос можно дать, рассмотрев функционирование процесса, называемого *передачей исключительных ситуаций* (exception propagation).

## Исключительные ситуации, устанавливаемые в выполняемом разделе

Когда исключительная ситуация устанавливается в выполняемом разделе блока PL/SQL, для определения обработчика, который должен быть вызван, используется следующий алгоритм:

1. Если в текущем блоке имеется обработчик данной исключительной ситуации, он выполняется, блок успешно завершается и управление программой передается вышестоящему блоку.
2. Если обработчик отсутствует, исключительная ситуация передается в вышестоящий блок и устанавливается там. После этого в вышестоящем блоке выполняется шаг 1.

Прежде чем перейти к подробному анализу приведенного алгоритма, необходимо дать определение *вышестоящему, или объемлющему блоку* (enclosing block). Блок может быть частью другого блока, то есть внешний блок может включать в свой состав внутренний. Рассмотрим пример:

```
DECLARE
  -- Начало внешнего блока.
...
BEGIN
  ...
DECLARE
  -- Начало внутреннего блока 1, который встроен во внешний блок.
```

```
    ...
    BEGIN
        ...
    END;
    ...
    BEGIN
        -- Начало внутреннего блока 2, который также встроен во
        -- внешний блок. Обратите внимание, что в этом блоке
        -- отсутствует раздел объявлений.
        ...
    END;
    ...
    -- Конец внешнего блока.
END;
```

В приведенном листинге внешний блок охватывает оба внутренних блока. Любая исключительная ситуация, не обработанная в блоках 1 и 2, передается во внешний блок.

При вызове процедуры также может создаваться объемлющий блок, что проиллюстрировано на следующем примере:

```
BEGIN
    -- Начало внешнего блока.
    -- Вызов процедуры. Внешний блок охватывает эту процедуру.
    F(...);
END;
```

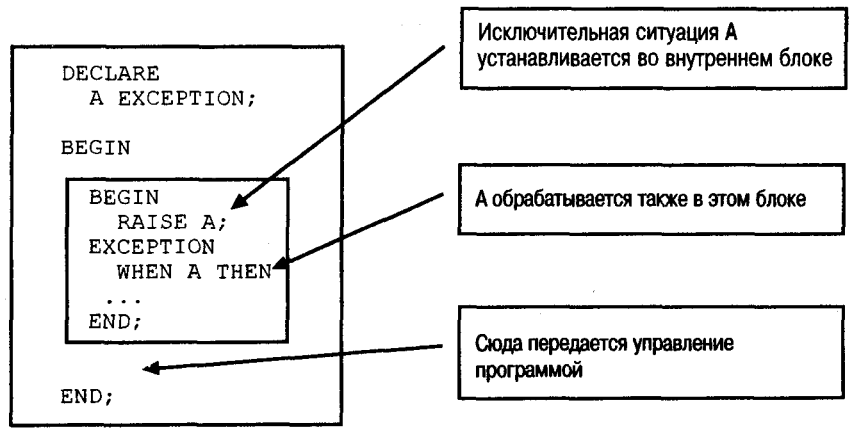
Если процедура **F** устанавливает исключительную ситуацию, которая не обрабатывается, то эта исключительная ситуация передается во внешний блок, так как он охватывает данную процедуру.

Различные варианты работы алгоритма передачи исключительных ситуаций продемонстрированы в примерах 1, 2, и 3, а также в последующих разделах.

## Пример 1

В этом примере показано применение правила 1. Исключительная ситуация A устанавливается и обрабатывается во внутреннем блоке. После этого управление программой передается внешнему блоку (рис.10.4).

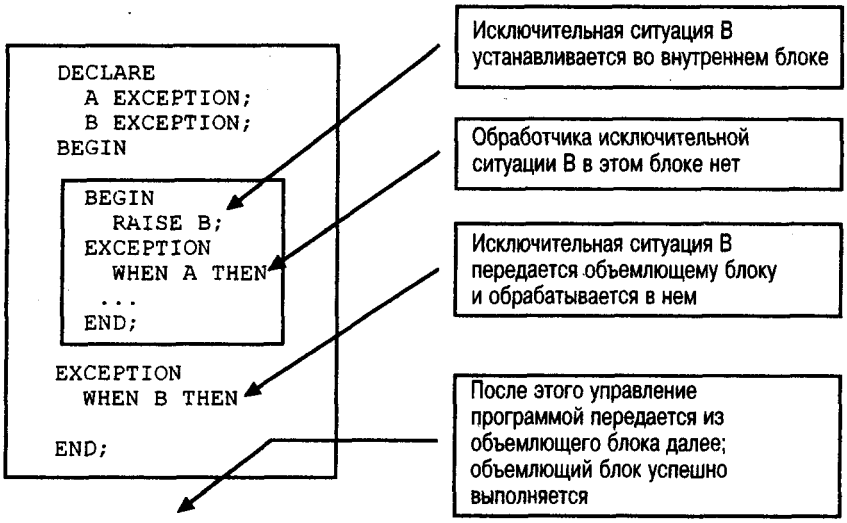
**Рис. 10.4.**



## Пример 2

В этом примере показано применение правила 2 для внутреннего блока. Исключительная ситуация передается объемлющему блоку; после этого применяется правило 1. Затем объемлющий блок успешно выполняется (рис. 10.5).

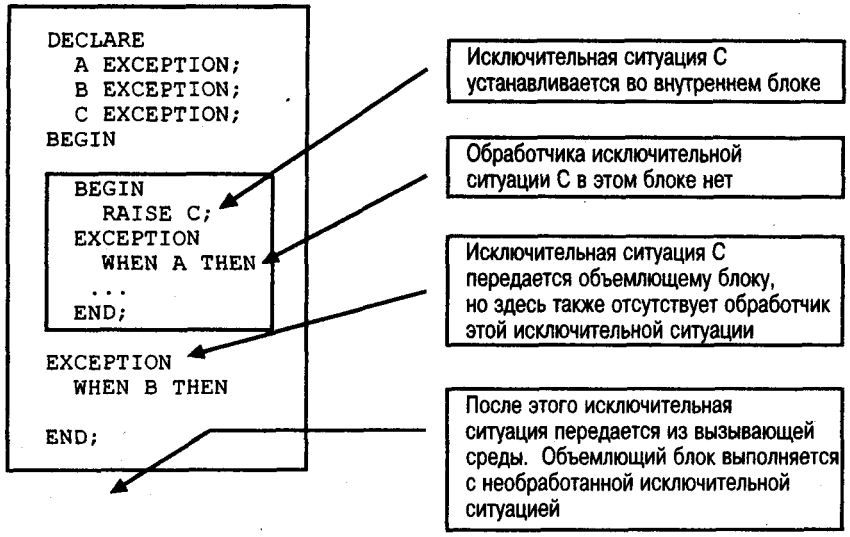
**Рис. 10.5.**




## Пример 3

В этом примере правило 2 вновь применяется для внутреннего блока. Исключительная ситуация передается объемлющему блоку, в котором отсутствует обработчик этой исключительной ситуации. Правило 2 применяется повторно, и объемлющий блок выполняется неуспешно, так как исключительная ситуация не обработана (рис. 10.6).

**Рис. 10.6.**



# Исключительные ситуации, устанавливаемые в разделе объявлений

Если в операции присваивания раздела объявлений устанавливается исключительная ситуация, она немедленно передается объемлющему блоку. В нем для дальнейшей передачи исключительной ситуации используются правила, сформулированные в предыдущем разделе. Даже если в текущем блоке имеется обработчик этой исключительной ситуации, он не выполняется. Вышесказанное проиллюстрировано на примерах 4 и 5.

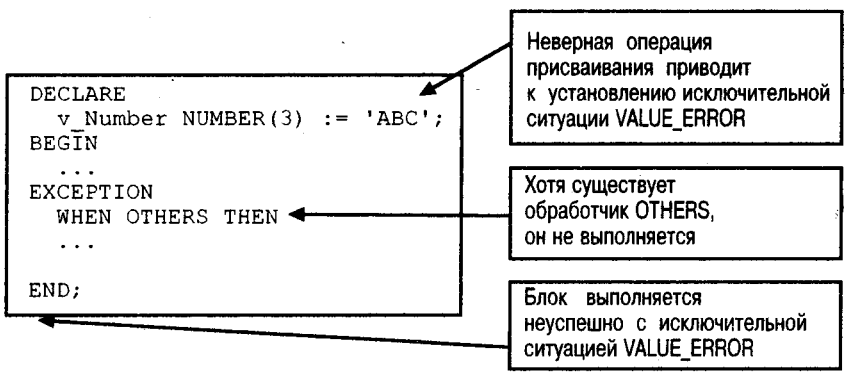
## Пример 4

В этом примере в описании

```
v_Number NUMBER(3) := 'ABC';
```

устанавливается исключительная ситуация VALUE_ERROR. Она немедленно передается объемлющему блоку. Хотя в рассматриваемом блоке имеется обработчик OTHERS, он, тем не менее, не выполняется (рис. 10.7). Если бы для данного блока существовал внешний блок, то исключительная ситуация была бы перехвачена в этом внешнем блоке (такая ситуация рассмотрена в примере 5).

**Рис. 10.7.**




## Пример 5

Как и в примере 4, в разделе объявлений внутреннего блока устанавливается исключительная ситуация VALUE_ERROR. При этом она сразу же передается внешнему блоку, в котором существует обработчик исключительных ситуаций OTHERS, поэтому исключительная ситуация обрабатывается, и внешний блок выполняется успешно (рис. 10.8).

# Исключительные ситуации, устанавливаемые в разделе исключительных ситуаций

Исключительные ситуации могут устанавливаться и в обработчиках исключительных ситуаций, либо явно, посредством оператора RAISE, либо неявно, при ошибке выполнения программы. В любом случае исключительная ситуация немедленно передается объемлющему блоку, так же, как и для раздела объявлений блока. Это делается потому, что в разделе исключительных ситуаций в каждый конкретный момент времени активной может быть лишь одна исключительная ситуация. Пока она обрабатывается, может быть установлена другая, однако установить несколько исключительных ситуаций одновременно нельзя. Вышесказанное иллюстрируется на примерах 6, 7 и 8.

## Пример 6

В этом примере исключительная ситуация A устанавливается, а затем обрабатывается. Однако в обработчике исключительной ситуации A устанавливается еще одна исключительная ситуация, B. Она немедленно передается внешнему блоку, минуя свой обработчик (рис. 10.9). Как и в примере 5, если бы этот блок был заключен во внешний блок, то внешний блок перехватил бы исключительную ситуацию B (см. пример 7).

**Рис. 10.8.**

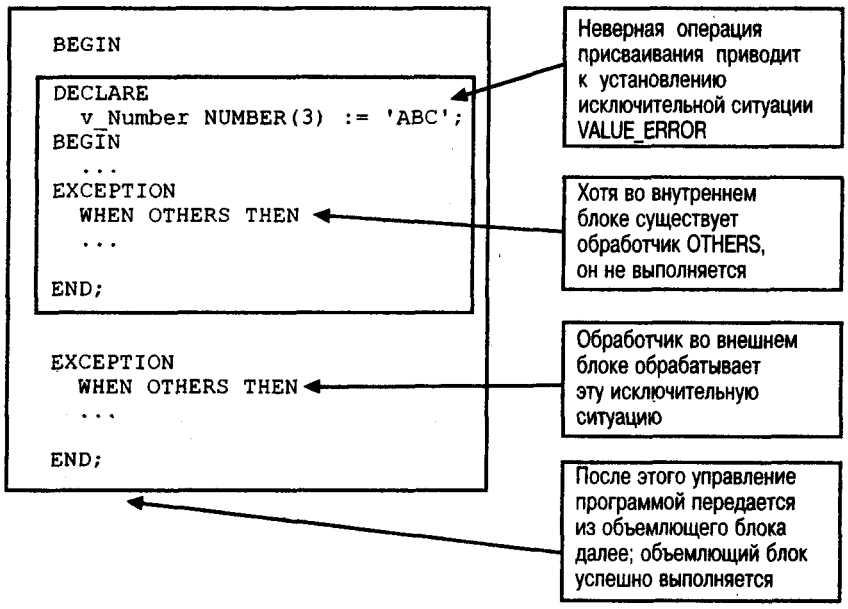
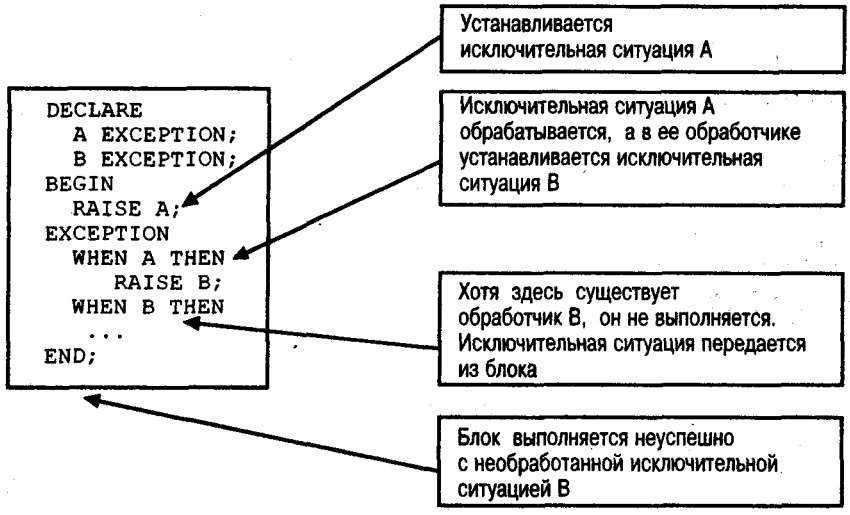


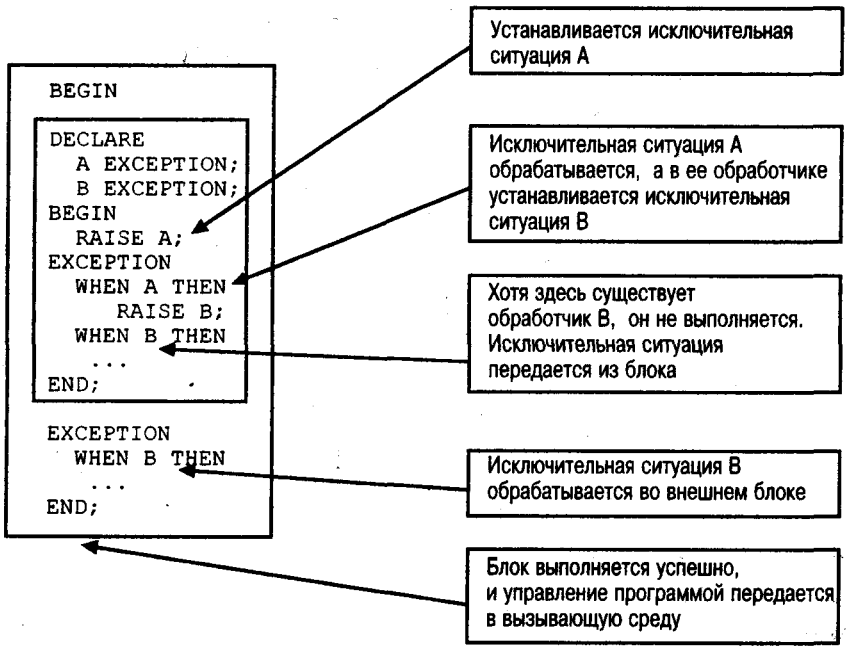

**Рис. 10.8.**




## Пример 7

Как и в примере 6, в обработчике исключительной ситуации A устанавливается исключительная ситуация B. Она немедленно передается охватывающему блоку, минуя внутренний обработчик B. Однако в этом примере существует внешний блок, который обрабатывает исключительную ситуацию B и успешно завершается (рис. 10.10).

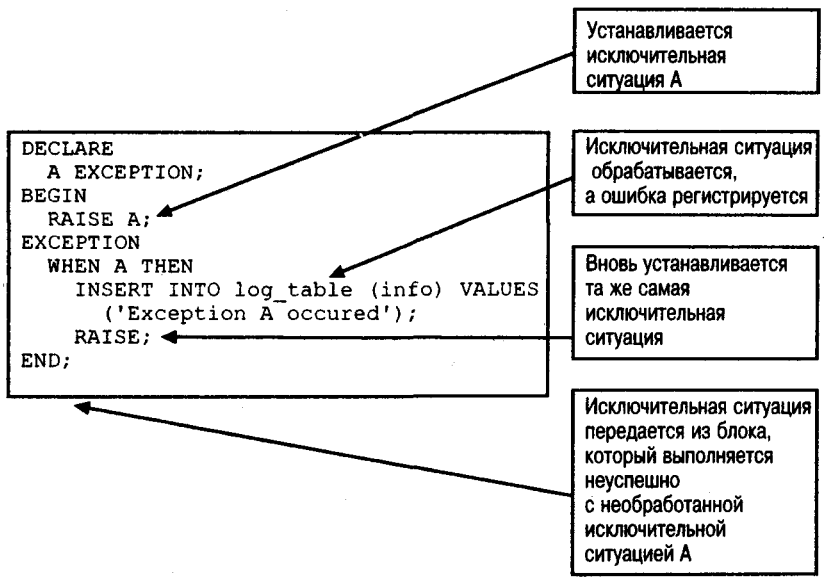
Рис. 10.10.




## Пример 8

Как показано в примерах 6 и 7, для установления исключительной ситуации в обработчике можно использовать оператор RAISE без аргументов, что приводит к передаче текущей исключительной ситуации в объемлющий блок (рис. 10.11.). Данный метод очень удобен для регистрации ошибки и/или для того, чтобы отменить изменения, вносимые ошибкой, и сообщить о ней объемлющему блоку. Этот метод будет рассмотрен в главе 18 при анализе программного модуля UTL_FILE.

Рис. 10.11.



# Рекомендации по использованию исключительных ситуаций

В этом разделе предлагаются рекомендации и советы по наилучшему использованию исключительных ситуаций в программах. Здесь рассказано об области действия исключительных ситуаций, о том, как не допускать необработанных исключительных ситуаций, как определять оператор, вызвавший установление конкретной исключительной ситуации. Эти рекомендации и советы помогут вам при работе со своими программами эффективно использовать исключительные ситуации и при этом избежать типичных ошибок.

## Область действия исключительной ситуации

Область действия исключительной ситуации аналогична области действия переменной. Если исключительная ситуация, определенная пользователем, будет передана из блока и окажется вне области своего действия, ссылаться на нее по имени будет невозможно. Проиллюстрируем это на примере:

```
DECLARE
  ...
BEGIN
  ...
  DECLARE
      e_UserDefinedException EXCEPTION;
  BEGIN
      RAISE e_UserDefinedException;
  END;
EXCEPTION
  /* e_UserDefinedException вне области своего действия -
     ее можно обработать только при помощи OTHERS. */
  WHEN OTHERS THEN
      /* Обработаем ошибку. */
END;
```

Если сообщение об ошибке, определяемой пользователем, нужно передать из блока, рекомендуется описывать исключительную ситуацию в модуле так, чтобы она была видима вне этого блока, или воспользоваться функцией RAISE_APPLICATION_ERROR. (Более подробно об этом см. в разделе "Использование функции RAISE_APPLICATION_ERROR"). Если создать модуль, называемый, скажем, **Globals**, и описать **e_UserDefinedException** в этом модуле, то данная исключительная ситуация будет видима и во внешнем блоке. Например:

```
CREATE OR REPLACE PACKAGE Globals
  /* В этом модуле выполняется глобальное объявление объектов.
     Объекты, объявленные здесь, будут видимы во всех других блоках
     и процедурах. Заметьте, что для этого модуля тело не создается. */

  /* Исключительная ситуация, определяемая пользователем. */
  e_UserDefinedException EXCEPTION;
END Globals;
```

С учетом созданного модуля Globals можно переписать рассмотренный выше фрагмент программы следующим образом:

```
DECLARE
  ...
BEGIN
  ...
  BEGIN
      /* Обратите внимание на то, что исключительную ситуацию
      e_UserDefinedException необходимо указывать с именем модуля. */
```

```
        RAISE Globals.e_UserDefinedException;
    END;
EXCEPTION
/* Поскольку e_UserDefinedException видима и здесь, можно
обработать ее явным образом. */
WHEN Globals.e_UserDefinedException THEN
/* Обработаем ошибку. */
END;
```

Помимо исключительных ситуаций, модулем Globals можно воспользоваться и для общих таблиц, переменных и типов PL/SQL. О модулях более детально рассказано в главе 8.

## Недопущение необрабатываемых исключительных ситуаций

Не следует допускать установления в своих программах исключительных ситуаций, которые не обрабатываются. Для этого можно воспользоваться обработчиком OTHERS, создав его на самом верхнем уровне программы. Например, если этот обработчик будет регистрировать факт возникновения каждой ошибки и время ее возникновения, то ни одна ошибка не будет оставлена без внимания:

```
DECLARE
  v_ErrorNumber NUMBER;          -- Переменная для хранения кода
                                 -- сообщения об ошибке.
  v_ErrorText VARCHAR2(200);     -- Переменная для хранения текста
                                 -- сообщения об ошибке.
BEGIN
  /* Обычная обработка информации. */
  ...
EXCEPTION
  WHEN OTHERS THEN
     /* Будем регистрировать все исключительные ситуации так, чтобы
        блок был успешно завершен. */
     v_ErrorNumber := SQLCODE;
     v_ErrorText := SUBSTR(SQLERRM, 1, 200);
     INSERT INTO log_table (code, message, info) VALUES
        (v_ErrorNumber, v_ErrorText, 'Oracle error occurred at' ||
         TO_CHAR(SYSDATE, 'DD-MON-YY HH24:MI:SS'));
END;
```

## Обнаружение ошибок

Иногда трудно определить, который из SQL-операторов стал причиной ошибки, так как раздел исключительных ситуаций анализируется для всего блока. Рассмотрим следующий пример:

```
BEGIN
  SELECT ...
  SELECT ...
  SELECT ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
     -- Какой из операторов выбора данных установил исключительную
     -- ситуацию?
END;
```

Для решения этой проблемы существует два способа. Первый — увеличение счетчика, указывающего на SQL-оператор:

```
DECLARE
  v_SelectCounter NUMBER := 1;    -- Переменная для хранения номера
                                  -- оператора выбора.
BEGIN
  SELECT ...
  v_SelectCounter := 2;
  SELECT ...
  v_SelectCounter := 3;
  SELECT ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
      INSERT INTO log_table (info) VALUES ('No data found in select'
        || v_SelectCounter);
END;
```

Второй способ — размещение каждого оператора в собственном внутреннем блоке:

```
BEGIN
  BEGIN
      SELECT ...
  EXCEPTION
      WHEN NO_DATA_FOUND THEN
        INSERT INTO log_table (info) VALUES ('No data found in select 1');
  END;
  BEGIN
      SELECT ...
  EXCEPTION
      WHEN NO_DATA_FOUND THEN
        INSERT INTO log_table (info) VALUES ('No data found in select 2');
  END;
  BEGIN
      SELECT ...
  EXCEPTION
      WHEN NO_DATA_FOUND THEN
        INSERT INTO log_table (info) VALUES ('No data found in select 3');
  END;
END;
```

# PL/SQL в работе: универсальный обработчик ошибок

Одна из проблем, возникающих при использовании исключительных ситуаций, заключается в том, что достаточно трудно определить, какой именно фрагмент программы выполняется в момент установления конкретной исключительной ситуации. В PL/SQL функция DBMS_UTILITY.FORMAT_CALL_STACK позволяет устранить эту трудность, возвращая значение типа VARCHAR2, являющееся текущим значением в списке или стеке вызовов (call stack). Рассмотрим пример, в котором фигурируют процедуры **A**, **B** и **C**:

```
-- Этот пример содержится в файле abc1.sql.
CREATE OR REPLACE PROCEDURE C AS
  v_CallStack VARCHAR2(2000);
BEGIN
  v_CallStack := DBMS_UTILITY.FORMAT_CALL_STACK;
  INSERT INTO temp_table (char_col) VALUES (v_CallStack);
  INSERT INTO temp_table (num_col)
```

```
      VALUES (-1);
END C;
CREATE OR REPLACE PROCEDURE B AS
BEGIN
  C;
END B;
CREATE OR REPLACE PROCEDURE A AS
BEGIN
  B;
END A;
```

Обратите внимание на то, что процедура **A** вызывает процедуру **B**, которая, в свою очередь, вызывает процедуру **C**. Если вызвать процедуру **A** и считать информацию таблицы temp_table, получится результат:

```
------ PL/SQL Call Stack -----
 Object      line   object
 Handle    number   name
16998f0          4   procedure EXAMPLE.C
1699ca0          3   procedure EXAMPLE.B
169f918          3   procedure EXAMPLE.A
1667ef0          1   anonymous block
```

Функция DBMS_UTILITY.FORMAT_ERROR_STACK, как и CALL_STACK, возвращает текущую последовательность ошибок. С помощью этих функций можно создать универсальный обработчик ошибок, регистрирующий как местонахождение ошибки, так и ее тип.

Ниже приведено описание таблиц, необходимых для создаваемого программного модуля.

```
-- Этот пример содержится в файле e_tables.sql.
CREATE TABLE errors (
  Module          VARCHAR2(50),
  Seq_number      NUMBER,
  Error_number    NUMBER,
  Error_mesg      VARCHAR2(100),
  Error_stack     VARCHAR2(2000),
  Call_stack      VARCHAR2(2000),
  Timestamp       DATE,
  PRIMARY KEY     (module, seq_number));

CREATE TABLE call_stacks (
  Module            VARCHAR2(50),
  Seq_number        NUMBER,
  Call_order        NUMBER,
  Object_handle     VARCHAR2(10),
  Line_num          NUMBER,
  Object_name       VARCHAR2(80),
  PRIMARY KEY (module, seq_number, call_order),
  FOREIGN KEY (module, seq_number) REFERENCES errors ON DELETE CASCADE);

DROP TABLE error_stacks;
CREATE TABLE error_stacks (
  Module          VARCHAR2(50),
  Seq_number      NUMBER,
  Error_order     NUMBER,
  Facility        CHAR(3),
  Error_number    NUMBER(5),
  Error_mesg      VARCHAR2(100),
```

```
    PRIMARY KEY (module, seq_number, error_order),
    FOREIGN KEY (module, seq_number) REFERENCES errors ON DELETE CASCADE);

CREATE SEQUENCE error_seq
    START WITH 1
    INCREMENT BY 1;
```

Используя приведенные таблицы, создадим модуль **ErrorPkg**:

```
-- Этот пример содержится в файле e_pkg.sql.
CREATE OR REPLACE PACKAGE ErrorPkg AS
    /* Универсальный программный модуль обработки ошибок, в котором
       используются функции DBMS_UTILITY.FORMAT_ERROR_STACK и
       DBMS_UTILITY.FORMAT_CALL_STACK. Этот модуль будет сохранять
       общую информацию об ошибках в таблице errors, а подробную
       информацию о списке вызовов и списке ошибок — соответственно в
       таблицах call_stacks и error_stacks. */

    -- Начало обработки ошибок. Процедуру HandleAll следует вызывать
    -- из каждого обработчика исключительной ситуации,
    -- регистрирующего ошибку. Переменная p_Top должна быть истинна
    -- только на самом высоком уровне вложенности. На других уровнях
    -- она должна быть ложна. Подробно об использовании процедуры
    -- см. в файле error_readme.txt.
    PROCEDURE HandleAll(p_Top BOOLEAN);

    -- Выводит на экран списки вызовов и ошибок (при помощи
    -- DBMS_OUTPUT) для заданных модуля и последовательного номера.
    PROCEDURE PrintStacks(p_Module IN errors.module%TYPE,
                          p_SeqNum IN errors.seq_number%TYPE);

    -- Расшифровывает списки вызовов и ошибок и сохраняет результаты
    -- в таблицах errors и call_stacks. Возвращает последовательный
    -- номер, под которым запоминается ошибка.
    -- Если параметр p_CommitFlag истинен (TRUE), то все операции
    -- ввода данных завершены.
    -- Для использования StoreStacks ошибка должна быть
    -- предварительно обработана. Поэтому необходимо всегда вызывать
    -- HandleAll с p_Top = TRUE.
    PROCEDURE StoreStacks(p_Module IN errors.module%TYPE,
                          p_SeqNum OUT errors.seq_number%TYPE,
                          p_CommitFlag BOOLEAN DEFAULT FALSE);
END ErrorPkg;

CREATE OR REPLACE PACKAGE BODY ErrorPkg AS

    v_NewLine CONSTANT CHAR(1) := CHR(10);

    v_Handled BOOLEAN := FALSE;
    v_ErrorStack VARCHAR2(2000);
    v_CallStack VARCHAR2(2000);

    PROCEDURE HandleAll(p_Top BOOLEAN) IS
    BEGIN
        IF p_Top THEN
```

```
      v_Handled := FALSE;
    ELSIF NOT v_Handled THEN
      v_Handled := TRUE;
      v_ErrorStack := DBMS_UTILITY.FORMAT_ERROR_STACK;
      v_CallStack := DBMS_UTILITY.FORMAT_CALL_STACK;
    END IF;
END HandleAll;

PROCEDURE PrintStacks(p_Module IN errors.module%TYPE,
                      p_SeqNum IN errors.seq_number%TYPE) IS
    v_TimeStamp errors.timestamp%TYPE;
    v_ErrorMsg errors.error_mesg%TYPE;

    CURSOR c_CallCur IS
      SELECT object_handle, line_num, object_name
        FROM call_stacks
        WHERE module = p_Module
        AND seq_number = p_SeqNum
        ORDER BY call_order;

    CURSOR c_ErrorCur IS
      SELECT facility, error_number, error_mesg
        FROM error_stacks
        WHERE module = p_Module
        AND seq_number = p_SeqNum
        ORDER BY error_order;
BEGIN
    SELECT timestamp, error_mesg
      INTO v_TimeStamp, v_ErrorMsg
      FROM errors
      WHERE module = p_Module
      AND seq_number = p_SeqNum;

    -- Выведем общую информацию об ошибке.
    DBMS_OUTPUT.PUT(TO_CHAR(v_TimeStamp, 'DD-MON-YY HH24:MI:SS'));
    DBMS_OUTPUT.PUT(' Module: ' || p_Module);
    DBMS_OUTPUT.PUT(' Error #' || p_SeqNum || ': ');
    DBMS_OUTPUT.PUT_LINE(v_ErrorMsg);

    -- Выведем список вызовов.
    DBMS_OUTPUT.PUT_LINE('Complete Call Stack:');
    DBMS_OUTPUT.PUT_LINE(' Object Handle Line Number Object Name');
    DBMS_OUTPUT.PUT_LINE(' ------------- ------------ ------------');
    FOR v_CallRec in c_CallCur LOOP
      DBMS_OUTPUT.PUT(RPAD(' ' || v_CallRec.object_handle, 15));
      DBMS_OUTPUT.PUT(RPAD(' ' || TO_CHAR(v_CallRec.line_num), 13));
      DBMS_OUTPUT.PUT_LINE(' ' || v_CallRec.object_name);
    END LOOP;
    -- Выведем список ошибок.
    DBMS_OUTPUT.PUT_LINE('Complete Error Stack:');
    FOR v_ErrorRec in c_ErrorCur LOOP
      DBMS_OUTPUT.PUT(' ' || v_ErrorRec.facility || '-');
      DBMS_OUTPUT.PUT(TO_CHAR(v_ErrorRec.error_number) || ': ');
      DBMS_OUTPUT.PUT_LINE(v_ErrorRec.error_mesg);
```

```
    END LOOP;

END PrintStacks;

PROCEDURE StoreStacks(p_Module IN errors.module%TYPE,
                      p_SeqNum OUT errors.seq_number%TYPE,
                      p_CommitFlag BOOLEAN DEFAULT FALSE) IS
    V_SeqNum        NUMBER;
    V_Index         NUMBER;
    V_Length        NUMBER;
    V_End           NUMBER;

    V_Call          VARCHAR2(100);
    V_CallOrder     NUMBER := 1;
    V_Handle        call_stacks.object_handle%TYPE;
    V_LineNum       call_stacks.line_num%TYPE;
    V_ObjectName    call_stacks.object_name%TYPE;

    V_Error         VARCHAR2(120);
    V_ErrorOrder    NUMBER := 1;
    V_Facility      error_stacks.facility%TYPE;
    V_ErrNum        error_stacks.error_number%TYPE;
    V_ErrMsg        error_stacks.error_mesg%TYPE;

    v_FirstErrNum errors.error_number%TYPE;
    v_FirstErrMsg errors.error_mesg%TYPE;
BEGIN
-- Сначала получим последовательный номер ошибки.
SELECT error_seq.nextval
    INTO v_SeqNum
    FROM dual;

p_SeqNum := v_SeqNum;

-- Внесем первую часть информации заголовка в таблицу errors.
INSERT INTO errors
    (module, seq_number, error_stack, call_stack, timestamp)
VALUES
    (p_Module, v_SeqNum, v_ErrorStack, v_CallStack, SYSDATE);

-- Расшифруем список ошибок и получим сведения о каждой из них.
-- Для этого просмотрим список ошибок. Начнем
-- с индекса в начале этого списка.
v_Index := 1;

-- Последовательно просмотрим список, отмечая каждый символ
-- новой строки. Этот символ заканчивает каждую ошибку в списке.
WHILE v_Index < LENGTH(v_ErrorStack) LOOP
    -- v_End - это позиция символа новой строки.
    v_End := INSTR(v_ErrorStack, v_NewLine, v_Index);

    -- Таким образом, сведения об ошибке расположены между
    -- текущим индексом и символом новой строки.
    v_Error := SUBSTR(v_ErrorStack, v_Index, v_End - v_Index);
```

```
      -- Перейдем к следующей итерации.
      v_Index := v_Index + LENGTH(v_Error) + 1;

      -- Информация об ошибке выглядит следующим образом:
      -- 'средство-номер: сообщение'. Необходимо выбрать каждый
      -- элемент для последующего ввода в таблицу.
      -- Средство - это первые 3 символа в информации об ошибке.

      v_Facility := SUBSTR(v_Error, 1, 3);
      -- Удалим средство и дефис (всегда 4 символа).

      v_Error := SUBSTR(v_Error, 5);
      -- Теперь получим номер ошибки.
      v_ErrNum := TO_NUMBER(SUBSTR(v_Error, 1,
      INSTR(v_Error, ':') - 1));

      -- Удалим номер ошибки, двоеточие и пробел (всегда 7 символов).
      v_Error := SUBSTR(v_Error, 8);

      -- Осталось сообщение об ошибке.
      v_ErrMsg := v_Error;

      -- Введем сведения об ошибках в таблицу и зафиксируем номер и
      -- текст сообщения первой ошибки.
      INSERT INTO error_stacks
         (module, seq_number, error_order, facility, error_number,
           error_mesg)
      VALUES
           (p_Module, p_SeqNum, v_ErrorOrder, v_Facility, v_ErrNum,
             V_ErrMsg);

      IF v_ErrorOrder = 1 THEN
        v_FirstErrNum := v_ErrNum;
        v_FirstErrMsg := v_Facility || '-' || TO_NUMBER(v_ErrNum) ||
                              ': ' || v_ErrMsg;
      END IF;

      v_ErrorOrder := v_ErrorOrder + 1;
END LOOP;

-- Пополним таблицу errors, введя в нее текст сообщения и код ошибки.
UPDATE errors
      SET error_number = v_FirstErrNum,
           error_mesg = v_FirstErrMsg
      WHERE module = p_Module
      AND seq_number = v_SeqNum;

-- Теперь необходимо расшифровать список вызовов и получить
-- сведения о каждом из них. Для этого просмотрим список вызовов.
-- Начнем с индекса, находящегося после
-- первого вызова в списке, то есть после первой пары из
-- слова 'name' и символа новой строки.
v_Index := INSTR(v_CallStack, 'name') + 5;
```

```
-- Последовательно просмотрим список, отмечая каждый символ
-- новой строки. Этот символ заканчивает каждый вызов в списке.
WHILE v_Index < LENGTH(v_CallStack) LOOP
    -- v_End - это позиция символа новой строки.
    v_End := INSTR(v_CallStack, v_NewLine, v_Index);

    -- Таким образом, сведения о вызове расположены между
    -- текущим индексом и символом новой строки.
    v_Call := SUBSTR(v_CallStack, v_Index, v_End - v_Index);

    -- Перейдем к следующей итерации.
    v_Index := v_Index + LENGTH(v_Call) + 1;

    -- В вызове находятся логический номер объекта, затем
    -- номер строки, а после этого номер объекта, разделенные
    -- пробелами. Необходимо выделить каждый из этих элементов для
    -- последующего ввода в таблицу.

    -- Сначала удалим из вызова пробел.
    v_Call := LTRIM(v_Call);

    -- Получим логический номер объекта.
    v_Handle := SUBSTR(v_Call, 1, INSTR(v_Call, ' '));

    -- Теперь удалим из вызова логический номер объекта, а затем пробел.
    v_Call := SUBSTR(v_Call, LENGTH(v_Handle) + 1);
    v_Call := LTRIM(v_Call);

    -- Теперь можно узнать номер строки.
    v_LineNum := TO_NUMBER(SUBSTR(v_Call, 1, INSTR(v_Call, ' ')));

    -- Удалим номер строки и пробел.
    v_Call := SUBSTR(v_Call, LENGTH(v_LineNum) + 1);
    v_Call := LTRIM(v_Call);

    -- Осталось имя объекта.
    v_ObjectName := v_Call;

    -- Внесем в таблицу все вызовы, исключая вызов ErrorPkg.
    IF v_CallOrder > 1 THEN
      INSERT INTO call_stacks
        (module, seq_number, call_order, object_handle, line_num,
          object_name)
      VALUES
        (p_Module, v_SeqNum, v_CallOrder, v_Handle, v_LineNum,
          v_ObjectName);
    END IF;

    v_Callorder := v_CallOrder + 1;

END LOOP;

IF p_CommitFlag THEN
    commit;
END IF;
```

```
  END StoreStacks;
END ErrorPkg;
```

Теперь создадим для таблицы **temp_table** триггер, устанавливающий исключительную ситуацию ZERO_DIVIDE, и модифицируем процедуры **A**, **B** и **C** следующим образом:

```
-- Этот пример содержится в файле abc2.sql.
CREATE OR REPLACE TRIGGER temp_insert
  BEFORE INSERT ON temp_table
BEGIN
  RAISE ZERO_DIVIDE;
END temp_insert;

CREATE OR REPLACE PROCEDURE C AS
BEGIN
  INSERT INTO temp_table (num_col) VALUES (7);
EXCEPTION
  WHEN OTHERS THEN
      ErrorPkg.HandleAll(FALSE);
      RAISE;
END C;

CREATE OR REPLACE PROCEDURE B AS
BEGIN
  C;
EXCEPTION
  WHEN OTHERS THEN
      ErrorPkg.HandleAll(FALSE);
      RAISE;
END B;

CREATE OR REPLACE PROCEDURE A AS
  v_ErrorSeq NUMBER;
BEGIN
  B;
EXCEPTION
  WHEN OTHERS THEN
      ErrorPkg.HandleAll(TRUE);
      ErrorPkg.StoreStacks('Error Test', v_ErrorSeq, TRUE);
      ErrorPkg.PrintStacks('Error Test', v_ErrorSeq);
END A;
```

Надо заметить, что, кроме самого высокого уровня вложенности (процедура **A**), все обработчики исключительных ситуаций выглядят следующим образом:

```
WHEN OTHERS THEN
  ErrorPkg.HandleAll(FALSE);
RAISE;
```

Это означает, что модуль **ErrorPkg** должен записать в таблицы (если нужно) списки вызовов и ошибок, а затем передать сведения об ошибке в вызывающую процедуру. На самом высоком уровне процедура **HandleAll** должна вызываться с параметром TRUE. Это сообщает модулю **ErrorPkg** о том, что достигнут самый верхний уровень вложенности и что дальше передавать сведения об ошибке не нужно. После этого процедура **StoreStacks** сохраняет списки вызовов и ошибок в таблицах **errors**, **error_stacks** и **call_stacks**, индексированных по имени модуля. Имя модуля (module) должно быть именем программного модуля (package) или другим ограничивающим идентификатором. Каждая ошибка сохраняется с указанием как имени модуля, так и номера ошибки, которые возвращаются процедурой **StoreStacks**. При вызове **PrintStacks** запрашивается таблица errors, и результаты выводятся на экран с помощью мо-

дуля DBMS_OUTPUT. Например, если вызвать из SQL*Plus процедуру A, то на экране будет отображена следующая информация:

```
SQL> SET SERVEROUTPUT ON SIZE 1000000 FORMAT TRUNCATED
SQL> EXEC A;
18-JAN-97 22:14:49 Module: Error Test Error #7: ORA-1476: divisor is
equal to zero
Complete Call Stack:
  Object Handle    Line Number    Object Name
---------------   ------------   ----------------------
  16998f0          6              procedure EXAMPLE.C
  1699ca0          3              procedure EXAMPLE.B
  169f918          4              procedure EXAMPLE.A
  16570e0          1              anonymous block
Complete Error Stack:
  ORA-1476: divisor is equal to zero
  ORA-6512: at "EXAMPLE.TEMP_INSERT", line 2
  ORA-4088: error during execution of trigger 'EXAMPLE.TEMP_INSERT'

PL/SQL procedure successfully completed.
```

# Итоги

В этой главе было рассказано о том, как обнаруживаются ошибки при выполнении программ PL/SQL и как программы реагируют на такие ошибки. Для этого в арсенале PL/SQL имеется специальное средство — исключительные ситуации и обработчики исключительных ситуаций. Было показано, как описываются исключительные ситуации и как устанавливается соответствие между ними и различными ошибками, как определяемыми пользователями, так и стандартными ошибками Oracle. Кроме того, были рассмотрены алгоритмы, посредством которых осуществляется передача исключительных ситуаций во всех разделах блоков PL/SQL. Завершается глава рекомендациями по использованию исключительных ситуаций.

# Глава 11

## Объекты

Объекты являются одним из основных новых средств Oracle8 и PL/SQL 8. В этой главе говорится о том, как создавать и применять объекты, в том числе о методах и конструкторах, а также о ссылках на объекты. Кроме того, обсуждаются вопросы хранения объектов в базе данных и манипулирования ими в операторах DML.

# Вступление

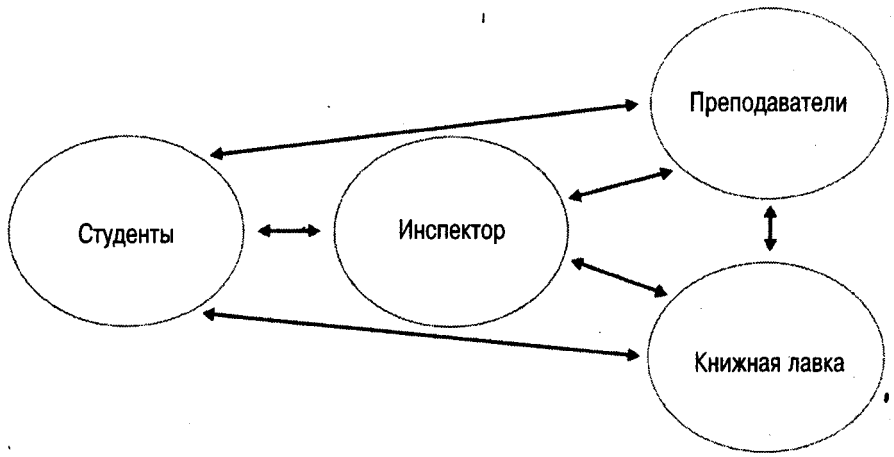
Перед тем как познакомиться с реализацией объектов в системе Oracle, необходимо обсудить основы объектно-ориентированной методологии разработки приложений. Подробное изложение этой методологии не является темой данной книги, однако ее основные принципы здесь будут раскрыты. Затем будет рассмотрена реализация объектных типов Oracle.

## Основы объектно-ориентированного программирования

Для чего создаются компьютерные приложения? Один из возможных ответов на этот вопрос – для моделирования ситуаций, возникающих в повседневной жизни. Программные средства разрабатываются для имитации как реальных объектов, так и взаимосвязей, существующих между ними. Используя построенные модели объектов и способов их взаимодействия, приложение может контролировать эволюцию объектов и автоматизировать процессы, происходящие в процессе этой эволюции. Например, система ввода заказов дает продавцу возможность сосредоточить все усилия на продаже товаров, предоставляя ему всю необходимую информацию и обеспечивая его взаимодействие с другими отделами организации, например с бухгалтерией или складом.

В качестве примера рассмотрим модель университета. Какие ее элементы являются главными, или сущностями? В университете обучаются студенты, которые обращаются к инспектору, чтобы записаться в учебные группы. Инспектор, в свою очередь, информирует преподавателей о том, какие именно студенты записались в какие группы. Преподаватели взаимосвязаны со студентами во время обучения и сдачи экзаменов. Работники университетской книжной лавки должны знать, какие книги преподаватели хотят использовать в работе, и обеспечить этими книгами студентов. Рассмотренная модель иллюстрируется на рис. 11.1. Здесь окружности обозначают сущности, действующие в модели университета (студенты, инспектор, преподаватели и книжная лавка), а стрелки – взаимодействие сущностей между собой, например приобретение книг студентами.

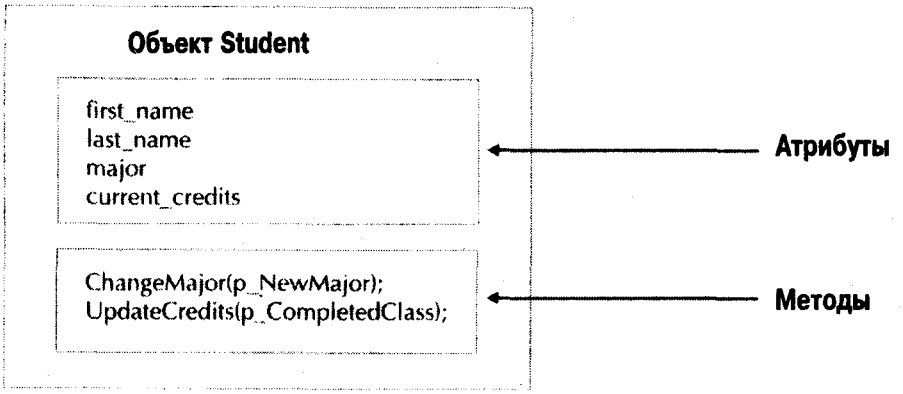
**Рис. 11.1.**
*Модель университета*




Это достаточно информативная модель, проанализировав которую можно получить различные сведения о реальном мире. К примеру, можно узнать, что инспектор является центральной фигурой в этой модели, поскольку все другие сущности должны с ним взаимодействовать.

С помощью объектно-ориентированного проектирования на основе этой модели создается компьютерное приложение. Каждая из сущностей в проектируемой системе обозначает конкретный объект. Объект представляет свойства, или атрибуты, реальной сущности и операции, выполняемые над этими

**Рис. 11.2.**
*Объект Student*




свойствами. Рассмотрим объект Student, представляющий студента (рис. 11.2). Студент имеет *атрибуты*, т.е. свойства (attributes): имя (first_name), фамилию (last_name), профилирующую дисциплину (major) и текущее число полученных зачетов (current_credits). Кроме того, на рисунке указаны операции, влияющие на эти атрибуты: **ChangeMajor** (изменяет профилирующую дисциплину) и **UpdateCredits** (обновляет сведения о зачетах студента в укомплектованной группе). Эти операции называются *методами* (methods).

Объекты взаимодействуют между собой, вызывая конкретные методы. Например, инспектор может вызвать метод **UpdateCredits** после окончания одним из студентов курса обучения.

## Абстракция

Как было показано на примере программных модулей (см. главу 8), атрибуты и методы объекта достаточно точно реализуют абстрактное представление данных и процедур. В идеале клиент, использующий данный объект, управляет атрибутами только через методы. При этом приложение клиента может просто вызывать методы, не зная о том, как они реально работают.

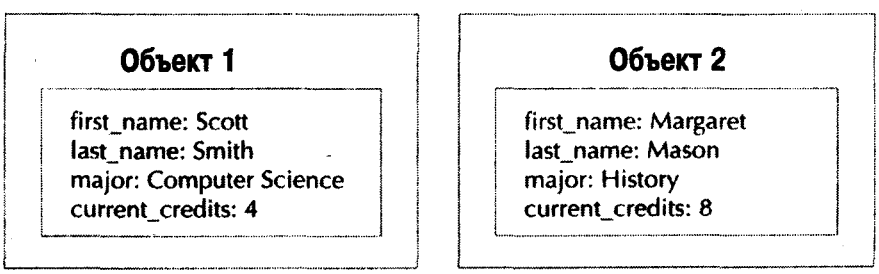
## Объекты и экземпляры объектов

Следует отметить различие, существующее между собственно объектным типом и экземпляром (instance) этого типа. В системе может быть только один объектный тип с конкретным именем, но много экземпляров этого типа. Экземпляр объекта похож на переменную: каждый экземпляр имеет свою собственную область памяти и, как следствие, собственную копию атрибутов объекта. К примеру, на рис. 11.3 показано два экземпляра объектного типа **StudentObj**. Они аналогичны двум различным записям PL/SQL, объявленным с одним и тем же типом.

## Объектно-реляционные базы данных

В настоящее время применяется множество объектно-ориентированных языков программирования, в том числе C++ и Java. Такие языки дают возможность описывать объекты и манипулировать ими, однако имеют существенный недостаток — они не обеспечивают надежного и корректного хранения и счи-

**Рис. 11.3.**
*Экземпляры типа StudentObj*



тывания объектов. Именно в этом случае призваны спасти положение объектно-реляционные базы данных, подобные Oracle8. Система Oracle8 создана для хранения объектных данных и для работы с ними. Управление объектными данными аналогично управлению реляционными данными и осуществляется с помощью языка SQL, выступающего в роли средства взаимодействия с базами данных. В объектно-реляционной базе данных язык SQL (и PL/SQL) используется для манипулирования как реляционными, так и объектными данными. Кроме того, Oracle8 обеспечивает: эффективное управление транзакциями, надежное резервное копирование и восстановление информации, высокопроизводительную обработку запросов, блокирование данных, параллельность работы пользователей, а также расширяемость самой системы. Объединение объектов с реляционной моделью дает отличные результаты — эффективность и надежность реляционной базы данных соединяется с гибкостью и средствами моделирования объектной структуры.

Далее в этой главе рассматриваются вопросы хранения объектов в Oracle8, а также способы считывания содержащейся в них информации. Затем говорится о создании объектов и методов, а также о построении информационных структур, применяемых для хранения объектов и методов в базе данных, — объектных таблиц. Кроме того, в данной главе обсуждаются объектные представления и возможность с их помощью представлять реляционные данные в виде объектов, что очень важно при переносе существующих моделей данных Oracle7 в новую систему.

# Объектные типы

PL/SQL 8.0 ... и ВЫШЕ

В Oracle8 объекты создаются с помощью *объектных типов* (object types), которые описывают атрибуты и методы конкретного вида объектов. Ниже показано, как создавать и использовать объектные типы. Однако для этого должно быть доступным *средство работы с объектами* (objects option). После соединения с базой данных Oracle8 должно появляться сообщение, похожее на это:

```
Oracle8 Enterprise Edition Release 8.0.3.0.0 - Production
With the Partitioning and Objects options
PL/SQL Release 8.0.3.0.0 - Production
```

Если в сообщении не указывается наличие средства работы с объектами, значит, оно не было установлено, и создавать и использовать объектные типы в этом случае невозможно. В такой ситуации необходимо с помощью программы-инсталлятора Oracle подключить к системе Oracle средство работы с объектами. За более подробной информацией обратитесь к руководству по инсталляции и руководству пользователя по системе. В данной главе подразумевается, что средство работы с объектами инсталлировано в системе.

## Создание объектных типов

Объектный тип похож на модуль и также имеет описание (спецификацию) и тело. В описании типа содержатся атрибуты и предварительные объявления методов, а его тело состоит из реального программного текста методов. Синтаксис создания спецификации типа приведен ниже, а синтаксис создания тела типа рассмотрен в разделе "Методы" далее в этой главе.

Описание объектного типа создается при помощи оператора CREATE TYPE ... AS OBJECT:

```
CREATE [OR REPLACE] TYPE [схема.] имя_типа AS OBJECT (
имя_атрибута тип_данных[, имя_атрибута тип_данных]...
| [{MAP | ORDER} MEMBER описание_функции]
| [MEMBER {описание_процедуры | описание_функции}
[, MEMBER {описание_процедуры | описание_функции}]...]
| [PRAGMA RESTRICT_REFERENCES (имя_метода, ограничения)
[, PRAGMA RESTRICT_REFERENCES (имя_метода, ограничения)]...]
);
```

где *имя_типа* — имя нового объектного типа, а *схема* — его владелец. Сначала через запятую перечисляются атрибуты типа:

*имя_атрибута тип_данных*

где *имя_атрибута* — это имя атрибута, а *тип_данных* — встроенный тип Oracle, или ранее определенный пользователем тип данных, или ссылка на объектный тип (см. раздел "Идентификаторы объектов и ссылки на объекты" ниже в этой главе). После атрибутов указываются методы вместе с прагмами

PRAGMA RESTRICT_REFERENCES для этих методов (см. главу 8). Синтаксис создания различных методов, в том числе методов MAP и ORDER, приведен в разделе "Методы".

Например, следующий оператор создает объектный тип для студентов:

```
-- Этот пример является частью файла tables8.sql.
CREATE OR REPLACE TYPE StudentObj AS OBJECT (
  ID              NUMBER(5),
  First_name      VARCHAR2(20),
  Last_name       VARCHAR2(20),
  Major           VARCHAR2(30),
  Current_credits NUMBER(3),
  );
```

**▼ ВНИМАНИЕ** При выполнении файла-сценария tables8.sql, находящегося на прилагаемом компакт-диске, создаются все типы и их методы, описанные в данной главе. Все примеры этого и последующих разделов являются частью указанного файла. В tables8.sql заново определяются многие таблицы, фигурирующие в рассматриваемых нами примерах (в том числе таблицы **students** и **classes**). Новые таблицы используются только в главах, посвященных Oracle8: 11, 12, 17, 20 и 21.

**▼ СОВЕТУЕМ** При создании спецификации или тела типа с помощью SQL*Plus необходимо указывать в строке, следующей за определением типа, косую черту (/), чтобы выполнился оператор CREATE TYPE ... AS OBJECT (как это делается при выполнении блока PL/SQL).

Следует сделать ряд замечаний относительно объектных типов:

1. Оператор CREATE TYPE является оператором DDL, поэтому его нельзя явно указывать в блоке PL/SQL. Для задания этого оператора можно воспользоваться модулем DBMS_SQL (описанным в главе 15).
2. *Для создания объектного типа необходимо иметь системную привилегию* CREATE TYPE (*являющуюся частью роли* RESOURCE).
3. Объектные типы создаются как объекты словаря данных. Следовательно, они создаются в текущей схеме, если только в операторе CREATE TYPE ... AS OBJECT не указана другая схема.
4. Атрибуты вновь создаваемого типа указываются подобно полям записи PL/SQL или столбцам таблицы в операторе CREATE TABLE.
5. В отличие от полей записи атрибуты объектного типа нельзя ограничивать как NOT NULL и инициализировать значениями по умолчанию.
6. На атрибуты объекта, как и на запись PL/SQL, можно ссылаться при помощи уточняющей записи через точку.

При использовании типов данных к атрибутам объектов предъявляется ряд требований. Атрибуты объектов могут иметь любой тип данных Oracle8, за исключением:

- Типов LONG и LONG RAW. Однако атрибуты могут иметь тип LOB (описанный в главе 21).
- Любых типов данных, используемых для поддержки национальных языков: NCHAR, NVARCHAR2 или NCLOB.
- Типа данных ROWID.
- Типов, доступных только в PL/SQL, но не в базе данных. В число таких типов входят BINARY_INTEGER, BOOLEAN, PLS_INTEGER, RECORD и REF CURSOR.
- Типов, определенных с %TYPE или %ROWTYPE.
- Типов, определенных в модуле PL/SQL.

Причиной таких требований является то, что в системе Oracle Release 8 объектный тип является объектом словаря данных. Поэтому разрешено использовать только те средства и типы, которые применяются непосредственно в базе данных, а типы PL/SQL и конструкции, подобные %TYPE, в базе данных не применяются. В последующих версиях Oracle8 эти ограничения, весьма вероятно, будут отменены, и можно будет объявлять объектные типы как локально в блоках PL/SQL, так и в словаре данных.

# Объявление и инициализация объектов

Как и другие переменные PL/SQL, объект описывается в разделе объявлений блока, и после его имени указывается соответствующий тип. Например:

```
DECLARE
  v_Student  StudentObj;
```

В этом блоке **v_Student** описывается как экземпляр объектного типа **StudentObj**. В соответствии с правилами PL/SQL, экземпляр объекта, объявленный таким образом, инициализируется NULL-значением. Однако, хотя объект в целом является NULL-объектом, вовсе не обязательно, чтобы его атрибуты также были NULL-значениями. Если после такого объявления объект становится NULL-объектом, ссылаться на его атрибуты запрещается. Вопросы, связанные с NULL-объектами, обсуждаются в разделе "NULL-объекты и NULL-атрибуты".

## Инициализация объектов

Так как же инициализировать объекты? Это можно сделать с помощью *конструктора* (constructor) – функции, возвращающей инициализированный объект и использующей в качестве аргументов значения атрибутов этого объекта. Для каждого объектного типа Oracle заранее определяет конструктор, имеющий то же имя, что и тип. Например, конструктор **StudentObj** можно описать так:

```
FUNCTION StudentObj(ID IN NUMBER,
                    first_name IN VARCHAR2,
                    last_name IN VARCHAR2,
                    major IN VARCHAR2,
                    current_credits IN NUMBER)
  RETURN StudentObj;
```

**▼ ВНИМАНИЕ** Конструктор не описывается явным образом, однако можно считать, что его описание приведено раньше. Имя конструктора совпадает с именем объектного типа.

Таким образом, можно создать инициализированный экземпляр типа StudentObj и обратиться к его атрибутам:

```
-- Этот пример содержится в файле objinit.sql.
DECLARE
  -- Создает экземпляр объекта с набором атрибутов.
  v_Student StudentObj :=
      StudentObj(10020, 'Chuck', 'Choltry', NULL, 0);
BEGIN
  -- Изменяет значение атрибута major на 'Music'. Обратите
  -- внимание на использование уточняющей записи через точку при
  -- ссылке на атрибут.
  v_Student.major := 'Music';
END;
```

## NULL-объекты и NULL-атрибуты

При объявлении объекта без конструктора создается NULL-объект. Важно различать понятия NULL-объекта и NULL-атрибута. Если объект является NULL-объектом, или **неделимым** (atomical) NULL, то на его атрибуты ссылаться нельзя. К примеру, в приведенном ниже блоке возникает ошибка "ORA-06530: Reference to uninitialized composite" (ссылка на неинициализированный набор. – *Прим. пер.*).

```
DECLARE
  v_Student  StudentObj;
BEGIN
  v_Student.ID := 10020;
END;
```

Для проверки, является ли объект NULL-объектом или нет, используется условие IS NULL. Например, процедура AssignName перед присваиванием аргументам некоторых значений проверяет, не имеют ли аргументы тип NULL:

```
-- Этот пример содержится в файле aname.sql.
CREATE OR REPLACE PROCEDURE AssignName(
  p_Student IN OUT StudentObj,
  p_FirstName IN VARCHAR2,
  p_LastName IN VARCHAR2) AS
BEGIN
  IF p_Student IS NULL THEN
      RAISE_APPLICATION_ERROR(-20000, 'Student is NULL');
  ELSE
      p_Student.first_name := p_FirstName;
      p_Student.last_name := p_LastName;
  END IF;
END AssignName;
```

## Предварительное объявление типов

Иногда полезно создать тип до того, как станет известно о его атрибутах и/или методах. Это можно сделать путем *предварительного объявления типа* (forward type declaration), которое аналогично предварительному объявлению процедуры или метода:

CREATE TYPE *имя_типа*;

Это облегчает работу с таблицами, ссылающимися друг на друга, а также позволяет другим типам ссылаться на создаваемый тип до того, как он будет полностью сформирован.

# Методы

В описании объектного типа методы объявляются после атрибутов, а реализуются в теле типа. Напомним синтаксис описания объектного типа, приведенный выше:

CREATE [OR REPLACE] TYPE [*схема.*] *имя_типа* AS OBJECT (
*имя_атрибута тип_данных*[, *имя_атрибута тип_данных*]...
| [{MAP | ORDER} MEMBER *описание_функции*]
| [MEMBER {*описание_процедуры* | *описание_функции*}
[, MEMBER {*описание_процедуры* | *описание_функции*}]...]
| [PRAGMA RESTRICT_REFERENCES (*имя_метода, ограничения*)
[, PRAGMA RESTRICT_REFERENCES (*имя_метода, ограничения*)]...]
);

При описании методов каждый из них отделяется запятой и выглядит как обычная хранимая подпрограмма PL/SQL. Единственное отличие описания метода от описания хранимой подпрограммы состоит в обязательном ключевом слове MEMBER (компонент). Расширим описание типа **StudentObj**:

```
-- Этот пример является частью файла tables8.sql.
CREATE OR REPLACE TYPE StudentObj AS OBJECT (
  ID              NUMBER(5),
  First_name      VARCHAR2(20),
  Last_name       VARCHAR2(20),
  Major           VARCHAR2(30),
  Current_credits NUMBER(3),

  -- Возвращает имя и фамилию, разделенные пробелом.
  MEMBER FUNCTION FormattedName
  RETURN VARCHAR2,
  PRAGMA RESTRICT_REFERENCES(FormattedName,
  RNDS, WNDS, RNPS, WNPS),

  -- Обновляет major значением, указанным в p_NewMajor.
```

```
    MEMBER PROCEDURE ChangeMajor(p_NewMajor IN VARCHAR2),
    PRAGMA RESTRICT_REFERENCES(ChangeMajor,
        RNDS, WNDS, RNPS, WNPS),

    -- Обновляет current_credits, добавляя число зачетов,
    -- содержащееся в p_CompletedClass, к текущему значению.
    MEMBER PROCEDURE UpdateCredits(p_CompletedClass IN ClassObj),
    PRAGMA RESTRICT_REFERENCES(UpdateCredits,
        RNDS, WNDS, RNPS, WNPS)
);
```

Следует сделать ряд замечаний относительно объявления методов в спецификации типа:

1. Перед предварительным объявлением каждого метода должно указываться ключевое слово MEMBER.
2. Вместо точки с запятой после каждого объявления (или прагмы) нужно ставить запятую. Это справедливо для всех элементов спецификации типа, в том числе и для атрибутов, исключая последний.
3. Методы должны объявляться после атрибутов.
4. Функции MAP и ORDER используются для задания порядка сортировки в данном объектном типе. Эти функции обсуждаются ниже.
5. Для разрешения вызова метода из SQL-оператора можно использовать прагму RESTRICT_REFERENCES. При этом для метода справедливы те же правила, что и при использовании ее для подпрограммы (см. главу 8).

Тело типа создается с помощью команды CREATE TYPE BODY, которая аналогична команде CREATE PACKAGE BODY, используемой при создании программного модуля. Синтаксис создания тела типа таков:

```
CREATE [OR REPLACE] TYPE [схема.] имя_типа BODY {IS | AS}
[{MAP | ORDER} MEMBER объявление_функции;]
| [MEMBER {объявление_процедуры | объявление_функции};
[MEMBER {объявление_процедуры | объявление_функции}]...]
END;
```

Создадим тело типа **StudentObj**:

```
-- Этот пример является частью файла tables8.sql.
CREATE OR REPLACE TYPE BODY StudentObj AS
  MEMBER FUNCTION FormattedName
      RETURN VARCHAR2 IS
  BEGIN
      RETURN SELF.first_name || ' ' || SELF.last_name;
  END FormattedName;

  MEMBER PROCEDURE ChangeMajor(p_NewMajor IN VARCHAR2) IS
  BEGIN
      major := p_NewMajor;
  END ChangeMajor;

  MEMBER PROCEDURE UpdateCredits(p_CompletedClass IN ClassObj) IS
  BEGIN
      current_credits := current_credits + p_CompletedClass.num_credits;
  END UpdateCredits;
END;
```

▼ ВНИМАНИЕ

В теле типа содержатся такие программные конструкции, как ключевое слово SELF и объектный тип **ClassObj** (см. ниже).

Объектные типы во многом похожи на модули, например:

- И те и другие абстрагируются от данных путем разделения спецификации и тела и создания из них различных объектов словаря данных. При таком разделении зависимости, существующие в теле объектного типа, разрываются точно так же, как в теле модуля.
- И те и другие создают объекты словаря данных.

Однако между объектными типами и модулями имеется ряд довольно существенных различий:

- В тело модуля можно включать дополнительные объявления, отсутствующие в его описании. Областью действия этих частных объявлений является только тело модуля. В теле же объектного типа могут находиться лишь компонентные подпрограммы.
- Объектные типы являются реальными типами PL/SQL – можно объявить переменную с конкретным объектным типом. Модуль же – это особый вид объектов схемы; он объединяет связанные объявления.
- В состав модуля может входить раздел инициализации; в состав же объекта он входить не может. Начальные значения для атрибутов устанавливаются с помощью функции-конструктора, описанной выше в разделе "Объявление и инициализация объектов".
- Тело объектного типа может содержать операторы только тех методов, которые указаны в спецификации типа. В отличие от тела модуля тело объектного типа не может содержать дополнительные атрибуты или частные методы. В будущих версиях эти ограничения, скорее всего, будут устранены.
- После ключевого слова END в конце объектного типа имя типа не указывается.

## Вызов метода

Хотя синтаксически методы похожи на модульные подпрограммы, вызов метода отличается от вызова подпрограммы. Хранимая подпрограмма – это отдельный объект, вызываемый непосредственно из блока PL/SQL. В случае же с объектными типами каждый экземпляр объекта находится в определенном, свойственном только ему состоянии. Методы объекта используются для модификации его состояния, поэтому каждый метод должен ссылаться на конкретный экземпляр объекта. Таким образом, чтобы вызвать метод для конкретного экземпляра, нужно использовать следующий синтаксис:

*имя_объекта.имя_метода*

где *имя_объекта* – имя объектной переменной, а *имя_метода* – имя метода. Ниже приведен блок, иллюстрирующий вызов методов для различных объектов студентов.

▼ ВНИМАНИЕ Если метод не содержит аргументов, его можно вызывать без круглых скобок, как и обычную процедуру PL/SQL, либо со скобками и пустым списком аргументов. Это продемонстрировано на примере того же блока.

```
-- Этот пример содержится в файле mcall.sql.
DECLARE
  v_Student1 StudentObj :=
    StudentObj(10020, 'Chuck', 'Choltry', NULL, 0);
  v_Student2 StudentObj :=
    StudentObj(10021, 'Denise', 'Davenport', NULL, 0);
BEGIN
  -- Изменим профилирующие дисциплины обоих студентов.
  v_Student1.ChangeMajor('Economics');
  v_Student2.ChangeMajor('Computer Science');

  -- Отобразим на экране имя первого студента. Обратите внимание:
  -- в этом вызове круглые скобки отсутствуют.
  DBMS_OUTPUT.PUT_LINE(v_Student1.FormattedName);

  -- Отобразим на экране имя второго студента. В этом вызове
  -- указан пустой список аргументов.
  DBMS_OUTPUT.PUT_LINE(v_Student2.FormattedName());
END;
```

Как и обычные процедуры, методы можно вызывать с использованием как именного, так и позиционного представления, а их параметры могут иметь значения по умолчанию. Можно переопределять тип и число аргументов метода. (Позиционное и именное представления, значения по умолчанию и переопределение обсуждаются в главах 7 и 8.)

## Ключевое слово SELF

Рассмотрим метод ChangeMajor:

```
MEMBER PROCEDURE ChangeMajor(p_NewMajor IN VARCHAR2) IS
  BEGIN
      major := p_NewMajor;
  END ChangeMajor;
```

Этот метод вызывается для модификации атрибута **major** типа **StudentObj**. Поэтому внутри метода идентификатор major связывается с экземпляром объекта. Для ясности в PL/SQL применяется ключевое слово SELF (сам), которое автоматически привязано к экземпляру объекта в методе. Перепишем метод **ChangeMajor** следующим образом:

```
MEMBER PROCEDURE ChangeMajor(p_NewMajor IN VARCHAR2) IS
  BEGIN
      SELF.major := p_NewMajor;
  END ChangeMajor;
```

В данном случае использование SELF не является обязательным, однако если нужно передать текущий экземпляр объекта другой процедуре или функции в качестве аргумента либо сослаться на этот экземпляр, то указывать SELF необходимо.

## Использование атрибута %TYPE с объектами

Атрибут %TYPE нужно использовать не непосредственно с атрибутами объектного типа, а с атрибутами экземпляра объектного типа. Это правило справедливо также и для записей. Проиллюстрируем вышесказанное на примере:

```
-- Этот пример содержится в файле prcnttyp.sql.
DECLARE
  -- Сначала объявим тип записи и переменные, имеющие тип записи и
  -- объектный тип.
  TYPE t_Rec IS RECORD (
      f1 NUMBER,
      f2 VARCHAR2(10));
  v_Student StudentObj;
  v_Rec t_Rec;

  -- Это объявление правильно, так как %TYPE указан с переменной.
  v_ID v_Student.ID%TYPE;
  -- Это объявление приводит к ошибке PLS-206, так как %TYPE
  -- указан с объектным типом.
  v_ID2 StudentObj.ID%TYPE;

  -- Это объявление правильно, так как %TYPE указан с переменной.
  v_F1 v_Rec.f1%TYPE;
  -- Это объявление приводит к ошибке PLS-206, так как %TYPE
  -- указан с типом записи.
  v_F2 t_Rec.f2%TYPE;
BEGIN
  NULL;
END;
```

В результате обоих неправильных объявлений возникает ошибка:

```
PLS-206: %TYPE must be applied to a variable, column, field or
Attribute
```

(%TYPE нужно применять с переменными, столбцами, полями и атрибутами. — *Прим. пер.*), потому что в каждом из этих объявлений отсутствует конструкция, которой можно присвоить некоторое значение.

## Исключительные ситуации и атрибуты объектных типов

Как было сказано в главе 7, параметрам вида OUT или IN OUT не присваиваются значения, если в хранимой подпрограмме не выполняется обработка исключительных ситуаций. Это справедливо и тогда, когда необрабатываемая исключительная ситуация устанавливается в методе. Кроме того, любая операция присваивания значений атрибутам, выполненная в методе, не завершается. Рассмотрим объект **ErrorObj**:

```
-- Этот пример является частью файла error.sql.
CREATE OR REPLACE TYPE ErrorObj AS OBJECT (
  attribute NUMBER,
  MEMBER PROCEDURE RaiseError(p_RaiseIt IN BOOLEAN,
                              p_OutParam IN OUT NUMBER),
  MEMBER PROCEDURE Print(p_Comment IN VARCHAR2 DEFAULT NULL)
);

CREATE OR REPLACE TYPE BODY ErrorObj AS
  MEMBER PROCEDURE RaiseError(p_RaiseIt IN BOOLEAN,
                              p_OutParam IN OUT NUMBER) IS
  BEGIN
      -- Присвоим значение параметра IN атрибуту и увеличим это
      -- значение на 1 для параметра OUT.
      SELF.attribute := p_OutParam;
      p_OutParam := p_OutParam + 1;
      IF p_RaiseIt THEN
        RAISE NO_DATA_FOUND;
      END IF;
  END RaiseError;

  MEMBER PROCEDURE Print (p_Comment IN VARCHAR2 DEFAULT NULL) IS
  BEGIN
      -- Выведем на экран комментарий вместе со значением атрибута.
      IF p_Comment IS NOT NULL THEN
        DBMS_OUTPUT.PUT(p_Comment || ', ');
      END IF;
      DBMS_OUTPUT.PUT_LINE('attribute = ' || attribute);
  END Print;
END;
```

Если после этого выполнить следующий блок:

```
-- Этот пример является частью файла error.sql.
DECLARE
  v_Test ErrorObj := ErrorObj(1);
  v_NumVal NUMBER := 10;
BEGIN
  -- Сначала выведем на экран атрибут и v_NumVal.
  v_Test.Print('After initialization, v_NumVal = ' || v_NumVal);
  -- Вызовем RaiseError со значением FALSE, чтобы параметру
  -- и атрибуту были присвоены некоторые значения.
  v_Test.RaiseError(FALSE, v_NumVal);
```

```
    v_Test.Print('After call with no exception, v_NumVal = ' ||
                 v_NumVal);
    -- Вызовем RaiseError со значением TRUE, чтобы параметру и
    -- атрибуту значения не присваивались.
    v_Test.RaiseError(TRUE, v_NumVal);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
        v_Test.Print('After call with exception, v_NumVal = ' ||
                     v_NumVal);
  END;
```

то будет получен такой результат:

```
After initialization, v_NumVal = 10, attribute = 1
After call with no exception, v_NumVal = 11, attribute = 10
After call with exception, v_NumVal = 11, attribute = 10
```

(После инициализации v_NumVal = 10, attribute = 1

После вызова без исключительной ситуации v_NumVal = 11, attribute = 10

После вызова с исключительной ситуацией v_NumVal = 11, attribute = 10. — *Прим. пер.*)

Обратите внимание, что как атрибуту, так и параметру OUT присваиваются некоторые значения, если процедура выполняется успешно, но, если устанавливается исключительная ситуация NO_DATA_FOUND, они сохраняют свои первоначальные значения.

# Изменение и удаление типов

Существующие объектные типы, как и другие объекты схемы, можно изменять. Для этого служит оператор ALTER TYPE, который можно использовать для компиляции описания или тела типа либо для внесения в тип новых методов. Удаляются типы с помощью оператора DROP TYPE.

## ALTER TYPE ... COMPILE

Эта форма команды ALTER TYPE имеет следующую структуру:

ALTER TYPE *имя_типа* COMPILE [SPECIFICATION | BODY];

где *имя_типа* — имя изменяемого типа. С помощью этой команды можно компилировать спецификацию или тело типа, применяя определение, хранящееся в словаре данных. Если не указывается ни SPECIFICATION, ни BODY, то перекомпилироваться будут как описание, так и тело типа. Например, следующая команда вызывает перекомпиляцию тела типа **StudentObj**:

```
ALTER TYPE StudentObj COMPILE BODY;
```

## ALTER TYPE ... REPLACE AS OBLECT

Другая форма команды ALTER TYPE используется для добавления к типу новых методов. Синтаксис ее таков:

ALTER TYPE *имя_типа* REPLACE AS OBLECT (
*спецификация_объектного_типа*);

где *имя_типа* — это имя объектного типа, а *спецификация_объектного_типа* — полное описание типа, определенное командой CREATE TYPE. Новое описание должно во всем, кроме дополнительных методов, совпадать с исходным. Должны быть указаны первоначальные атрибуты и типы. Если тело типа уже существует, оно становится недостоверным, поскольку в нем не описаны новые методы. Использование команды ALTER TYPE ... REPLACE AS OBLECT иллюстрируется на примере следующего сеанса работы в SQL*Plus:

```
-- Этот пример содержится в файле alter.sql.
SQL> /* Создадим простой объектный тип с двумя атрибутами и двумя
        методами. */
SQL> CREATE OR REPLACE TYPE DummyObj AS OBJECT (
  2    f1 NUMBER,
  3    f2 NUMBER,
  4    MEMBER PROCEDURE Method1 (x IN VARCHAR2),
```

```
  5    MEMBER FUNCTION Method2 RETURN DATE
  6 );
  7 /

Type created.

SQL> /* Создадим тело типа. */
SQL> CREATE OR REPLACE TYPE BODY DummyObj AS
  2    MEMBER PROCEDURE Method1 (x IN VARCHAR2) IS
  3    BEGIN
  4    NULL;
  5    END Method1;
  6    MEMBER FUNCTION Method2 RETURN DATE IS
  7    BEGIN
  8    RETURN SYSDATE;
  9    END Method2;
10 END;
11 /

Type body created.

SQL> SELECT object_name, object_type, status
  2    FROM user_objects
  3    WHERE object_name = 'DUMMYOBJ';

OBLECT_NAME      OBJECT_TYPE     STATUS
-------------    -------------  ---------
DUMMYOBJ         TYPE           VALID
DUMMYOBJ         TYPE BODY      VALID

SQL> /* Изменим тип, добавив к нему новый метод. При этом тело типа
        становится недостоверным. */
SQL> ALTER TYPE DummyObj REPLACE AS OBJECT (
  2    f1 NUMBER,
  3    f2 NUMBER,
  4    MEMBER PROCEDURE Method1 (x IN VARCHAR2),
  5    MEMBER FUNCTION Method2 RETURN DATE,
  6    MEMBER PROCEDURE Method3
  7 );

Type altered.

SQL> SELECT object_name, object_type, status
  2    FROM user_objects
  3    WHERE object_name = 'DUMMYOBJ';

OBLECT_NAME         OBJECT_TYPE        STATUS
----------------    ---------------    ----------
DUMMYOBJ            TYPE               VALID
DUMMYOBJ            TYPE BODY          INVALID
```

## DROP TYPE

Команда DROP TYPE используется для удаления объектного типа или тела объектного типа и имеет следующий синтаксис:

DROP TYPE [*схема*.]*имя_типа* [FORCE];

Если параметр FORCE (удалять принудительно) не указан, то объектный тип будет удален только при отсутствии в схеме другого объекта, зависящего от данного типа. Если параметр FORCE указан, то объектный тип удаляется, делая при этом недействительными все зависимые от него объекты.

Команда DROP TYPE BODY удаляет только тело объектного типа, не трогая спецификацию типа и все зависимые от него объекты:

DROP TYPE BODY [*схема.*]*имя_типа*;

## Зависимости объектов

Как и в случае с записями, в один объектный тип может быть встроен другой объектный тип. Рассмотрим следующий пример:

```
-- Этот пример содержится в файле odepend.sql.
CREATE OR REPLACE TYPE Obj1 AS OBJECT (
   F1  NUMBER,
   F2  VARCHAR2(10),
   F3  DATE
);

CREATE OR REPLACE TYPE Obj2 AS OBJECT (
   F1  DATE,
   F2  CHAR(1)
);

CREATE OR REPLACE TYPE Obj3 AS OBJECT (
   a Obj1,
   b Obj2
);
```

Обратите внимание: атрибуты типа **Obj3** имеют типы **Obj1** и **Obj2.** В итоге **Obj3** зависит от **Obj1** и **Obj2** точно так же, как процедура зависит от таблиц. Поэтому нельзя удалить или изменить **Obj1** или **Obj2**, не удалив сначала **Obj3**. Рассмотрим следующий пример сеанса работы в SQL*Plus:

```
SQL> DROP TYPE Obj1;
DROP TYPE Obj1
*
ERROR at line 1:
ORA-02303: cannot drop or replace a type with type or table
           Dependents

(ОШИБКА в строке 1:
ORA-02303: нельзя удалить или заменить тип, от которого зависит
тип или таблица. — Прим. пер.)

SQL> DROP TYPE Obj3;
Type dropped.

SQL> DROP TYPE Obj1;
Type dropped.
```

▼ ВНИМАНИЕ Если объект имеет атрибут, представляющий собой ссылку на другой объектный тип, то объект также зависит от этого объектного типа. Аналогично, если для некоторого объектного типа создана объектная таблица, то объектный тип зависит от этой таблицы. (Ссылки на объекты и объектные таблицы обсуждаются в следующем разделе.)

# Объекты в базе данных

Рассмотренная выше объектная методология является в принципе общей для всех языков программирования, использующих объектно-ориентированную разработку приложений. Объявление и создание объектов, инициализация методов — все эти операции схожи в любой объектной системе. Однако в Oracle8 имеется возможность хранить создаваемые объекты в базе данных, что повышает надежность объектов.

## Размещение объектов

Объекты можно располагать в различных частях приложения Oracle8: хранить в базе данных, локально *объявлять в блоках PL/SQL или сохранять в кэш-памяти на станции клиента. Местонахождение объекта* определяет его свойства, а также операции, которые разрешено над ним выполнять.

### Устойчивые и неустойчивые объекты

*Устойчивым* (persistent) объектом называется объект, хранящийся в базе данных, а *неустойчивым* (transient) — объект, являющийся локальным по отношению к некоторому блоку PL/SQL. Когда неустойчивый объект, например локальная переменная PL/SQL, покидает область действия, занимаемое им место освобождается; устойчивый же объект остается доступным до тех пор, пока не будет удален явным образом. Устойчивые объекты хранятся в таблицах базы данных, как и стандартные скалярные типы Oracle7 (NUMBER, VARCHAR2, DATE и т.д.). Существует два различных способа хранения объектов в таблице: в качестве объекта-строки или объекта-столбца.

**Объекты-Строки** Объект-строка (row object) занимает целую строку таблицы базы данных, причем в строке содержится только он — полей других столбцов в данной строке быть не может. Таблица, состоящая из таких строк, называется объектной (object table) и создается при помощи следующего оператора:

CREATE TABLE *имя_таблицы* OF *объектный_тип*;

где *имя_таблицы* — это имя создаваемой таблицы, а *объектный_тип* — тип объекта-строки. В качестве примера создадим объектную таблицу rooms:

```
-- Этот пример является частью файла tables8.sql.
-- Сначала создадим описание и тело объектного типа.
CREATE OR REPLACE TYPE RoomObj AS OBJECT (
   ID             NUMBER(5),
   Building       VARCHAR2(15),
   Room_number    NUMBER(4),
   Number_seats   NUMBER(4),
   Description    VARCHAR2(50),
   MEMBER PROCEDURE Print,
);

CREATE OR REPLACE TYPE BODY RoomObj AS
   MEMBER PROCEDURE Print IS
   BEGIN
       DBMS_OUTPUT.PUT('Room ID:' || ID || ' is located in ');
       DBMS_OUTPUT.PUT(building || ', room ' || room_number);
       DBMS_OUTPUT.PUT(', and has ' || number_seats || ' seats.');
       DBMS_OUTPUT.NEW_LINE;
   END Print;
END;

-- Теперь создадим объектную таблицу.
CREATE TABLE rooms OF RoomObj;
```

В каждой строке таблицы **rooms** находится экземпляр типа **RoomObj**. Следовательно, в эту таблицу можно вводить только объекты. Ниже приведен ряд примеров ввода объектов в таблицу rooms. Обратите внимание на использование функции-конструктора **RoomObj**.

```
-- Этот пример является частью файла tables8.sql.
INSERT INTO rooms VALUES
```

```
    (RoomObj(99999, 'Building 7', 310, 1000,
            'Large Lecture Hall'));
INSERT INTO rooms VALUES
    (RoomObj(99998, 'Building 6', 101, 500,
            'Small Lecture Hall'));
INSERT INTO rooms VALUES
    (RoomObj(99997, 'Building 6', 150, 50,
            'Discussion Room A'));
INSERT INTO rooms VALUES
    (RoomObj(99996, 'Building 6', 160, 50,
            'Discussion Room B'));
```

Объектные таблицы очень похожи на обычные реляционные таблицы. Действительно, все операции, выполняемые над реляционными таблицами, можно выполнять и над объектными. К примеру, можно ввести данные в таблицу **rooms** с помощью следующего оператора INSERT:

```
INSERT INTO rooms VALUES
(99999, 'Building 7', 310, 1000, 'Large Lecture Hall');
```

Это обеспечивает достаточно простой и удобный переход от Oracle7 к Oracle8. Во время такой модернизации различные реляционные таблицы можно создать заново как объектные, причем существующие приложения изменять вовсе не обязательно. Можно написать новые приложения, использующие объектные конструкторы и другие методы, определенные для объектов. (Дополнительные примеры операторов INSERT приведены ниже в этой главе.)

**Объекты-Столбцы** *Объект-столбец* (column object) занимает только один столбец таблицы. Для создания таблицы, содержащей объект-столбец, нужно просто указать объектный тип в качестве типа столбца в операторе CREATE TABLE. При создании одной таблицы можно одновременно указать и скалярный тип, и тип объекта-столбца. Для примера создадим таблицу **students** (с учетом описанного выше типа **StudentObj**):

```
-- Этот пример является частью файла tables8.sql.
CREATE OR REPLACE TYPE AddressObj AS OBJECT (
  Line1    VARCHAR2(40),
  Line2    VARCHAR2(40),
  City     VARCHAR2(30),
  State    CHAR(2),
  Zipcode  NUMBER(5)
);
CREATE TABLE students (
  student StudentObj,
  address AddressObj
);
```

Таблица **students** состоит из двух столбцов: **student** и **address**, поэтому можно ввести в нее данные из следующего примера. Обратите внимание, что используются обе функции-конструкторы – **StudentObj** и **AddressObj** – и что в третьей вводимой строке значением для **address** является NULL.

```
-- Этот пример является частью файла tables8.sql.
INSERT INTO students VALUES
  (StudentObj(student_sequence.NEXTVAL, 'Scott', 'Smith',
            'Computer Science', 0),
   AddressObj('100 Main St', NULL, 'East Brunswick', 'CA',
            91234));
INSERT INTO students VALUES
  (StudentObj(student_sequence.NEXTVAL, 'Margaret', 'Mason',
            'History', 0),
   AddressObj('350 Sorority Row', 'Apt# 2B', 'East Brunswick',
            'CA', 91234));
INSERT INTO students VALUES
```

```
    (StudentObj(student_sequence.NEXTVAL, 'Joanne', 'Junebug',
                'Computer Science', 0),
    NULL);
```

## Идентификаторы объектов и ссылки на объекты

*Идентификатор объекта* (OID – object identifier) – это уникальный указатель на устойчивый объект определенного типа. Как и идентификатор строки (ROWID), который однозначно определяет строку, идентификатор объекта однозначно определяет объект. Гарантируется, что идентификаторы объектов уникальны по всему пространству Oracle8: два объекта не могут иметь один и тот же идентификатор. Более того, существующий идентификатор никогда не будет создан вновь, даже если объект, который он идентифицирует, удаляется. Идентификатор объекта – это внутрисистемная структура; общее число идентификаторов составляет $2^{128}$ различных значений.

Идентификаторы объекта имеются только у объектов-строк и строк объектных представлений (см. ниже в этой главе). Ни объекты-столбцы, ни неустойчивые объекты (локальные по отношению к блокам PL/SQL) идентификаторов не имеют. Если объект имеет идентификатор, на этот объект можно ссылаться. Как уже отмечалось в главе 2, в PL/SQL существует два вида ссылочных типов. Ссылки на курсоры реализуются при помощи типов REF CURSOR. Переменная REF CURSOR – это совсем не то же самое, что курсорная переменная, а, скорее, указатель на курсор. Смысл ссылок на объекты примерно тот же: ссылка – это указатель на некоторый объект, а не собственно объект. Ссылка на объект описывается в разделе объявлений или указывается в описании таблицы и выглядит следующим образом:

*имя_переменной* REF *объектный_тип*;

где *имя_переменной* – это имя ссылки на объект, а *объектный_тип* – объектный тип. Например, в описании типа **ClassObj** содержится ссылка на объект **RoomObj**:

```
-- Этот пример является частью файла tables8.sql.
CREATE OR REPLACE TYPE ClassObj AS OBJECT (
  Department          CHAR(3),
  Course              NUMBER(3),
  Description         VARCHAR2(2000),
  Max_students        NUMBER(3),
  Current_students    NUMBER(3),
  Num_credits         NUMBER(1),
  Room                REF RoomObj
);
```

Ссылки на объекты можно указывать в блоках PL/SQL и в SQL-операторах, используя при этом операции VALUE и REF (см. следующий раздел).

# Объекты в операторах DML

Во многих отношениях объекты в операторах DML аналогичны скалярам. Например, можно считать объект из таблицы базы данных в переменную того же типа, что и объект, или обновить объектную таблицу, указав объект в условии WHERE. Операции DML, выполняемые над таблицами, в которых содержатся объекты-строки или объекты-столбцы, абсолютно идентичны реляционным операциям DML: операции обоих видов выполняются в границах транзакций и подчиняются одинаковым правилам по обеспечению согласованности чтения и отката транзакций. Однако некоторые операторы DML стоит рассмотреть более детально (что и сделано ниже).

## INSERT

В ряде примеров уже рассматривалось применение объектов в операторах INSERT. При вводе объекта можно воспользоваться объектным конструктором или объектной переменной PL/SQL, содержащей вводимый объект. В операторах INSERT объекты ведут себя практически так же, как и скалярные типы.

## UPDATE

В операторе UPDATE (в условии WHERE или в конструкции VALUES) можно указывать объекты в качестве переменных привязки. В следующем блоке посредством оператора UPDATE создается новая аудитория и пополняется таблица **classes**:

```
-- Этот пример содержится в файле update.sql.
DECLARE
  v_NewRoom  RoomObj  :=
```

```
RoomObj(99990, 'Building 7', 200, 50, 'Discussion Room F');
v_RoomRef REF RoomObj;
BEGIN
- Конструкция RETURNING этого оператора помещает ссылку на вновь
- введенную аудиторию в v_RoomRef.
INSERT INTO rooms r VALUES (v_NewRoom)
RETURNING REF(r) INTO v_RoomRef;
UPDATE classes
SET room = v_RoomRef
WHERE department = 'NUT' and course = 307;
END;
```

## DELETE

В операторе DELETE (в условии WHERE) можно точно так же ссылаться на объект или атрибуты объекта. Например, с помощью следующего оператора DELETE будут удалены все сведения о студентах-историках:

```
DELETE FROM students s
  WHERE s.student.major = 'History';
```

## Объекты-столбцы в операторах SELECT

Если объект хранится в таблице в виде объекта-столбца, содержащуюся в нем информацию можно считывать с помощью обычного оператора SELECT:

```
-- Этот пример является частью файла colsel.sql.
DECLARE
  v_Student StudentObj;
  v_Address AddressObj;

  CURSOR c_Students IS
      SELECT student, address
        FROM students;
BEGIN
  -- Выведем на экран идентификаторы всех студентов.
  OPEN c_Students;

  LOOP
      FETCH c_Students INTO v_Student, v_Address;
      EXIT WHEN c_Students%NOTFOUND;

      DBMS_OUTPUT.PUT_LINE('Student ID: ' || v_Student.ID);
  END LOOP;

  CLOSE c_Students;
END;
```

Кроме того, *можно сослаться на объект-столбец в условии* WHERE, указав *полное имя этого объекта*. Ниже приведен пример правильного использования оператора SELECT, так как здесь указано квалифицированное имя атрибута ID:

```
SELECT student
  FROM students
  WHERE students.student.id = 10009;
```

Однако в PL/SQL тот же самый оператор выполнить нельзя, поскольку для таблицы не указан псевдоним. Например, при выполнении следующего блока возникает ошибка PLS-327: "students is not in SQL scope here" (таблица students вне области действия SQL. — *Прим. пер.*)

```
-- Этот пример является частью файла colsel.sql.
DECLARE
  v_Student  StudentObj;
BEGIN
  -- Устанавливает исключительную ситуацию PLS-327.
  SELECT  student
      INTO  v_Student
      FROM  students
      WHERE  students.student.ID  =  10009;
END;
```

Выходом из этого положения является использование псевдонима:

```
-- Этот пример является частью файла colsel.sql.
DECLARE
  v_Student  StudentObj;
BEGIN
  -- Выполняется успешно, так как для таблицы указан псевдоним.
  SELECT  student
      INTO  v_Student
      FROM  students  s
      WHERE  s.student.ID  =  10009;
END;
```

▼ СОВЕТУЕМ При выборе данных из таблицы, содержащей объекты-столбцы или объекты-строки, всегда используйте псевдонимы. Это дает гарантию работоспособности запроса как в PL/SQL, так и в SQL.

## Объекты-строки в операторах SELECT

В запросе объект-строка ведет себя иначе. Поскольку объектная таблица описывается точно так же, как и реляционная, сам объект именовать нельзя. Следовательно, для выбора объекта или для ссылки на него нужно использовать операцию VALUE или REF.

**Операция VALUE** Операция VALUE (значение) возвращает объект, а не список атрибутов. В качестве аргумента используется *переменная корреляции* (correlation variable). В этом контексте переменная корреляции — это просто псевдоним таблицы. Использование VALUE иллюстрируется в следующем примере:

```
-- Этот пример содержится в файле valueop.sql.
DECLARE
  V_RoomID          rooms.id%TYPE;
  V_Building        rooms.building%TYPE;
  V_RoomNumber      rooms.room_number%TYPE;
  V_NumberSeats     rooms.number_seats%TYPE;
  V_Description     rooms.description%TYPE;
  V_RoomObj         RoomObj;
BEGIN
  -- Считаем данные, не указывая VALUE. Это полный аналог
  -- реляционного запроса.
  SELECT  *
      INTO  v_RoomID,  v_Building,  v_RoomNumber,  v_NumberSeats,
            v_Description
      FROM  rooms  r
      WHERE  ID  =  99993;

  -- Считаем данные, указав VALUE. В этом случае выбирается RoomObj.
  SELECT  VALUE(r)
```

```
        INTO v_RoomObj
        FROM rooms r
        WHERE ID = 99993;
END;
```

Результатом запроса, возвращающим VALUE, является набор объектов, а не набор атрибутов.

**Операция REF** Результатом выполнения операции REF является ссылка на запрашиваемый объект, а не сам объект. Как и для VALUE, аргумент REF – это переменная корреляции. Приведем пример:

```
-- Этот пример является частью файла refop.sql.
DECLARE
  V_RoomRef   REF RoomObj;
  V_Room      RoomObj;
BEGIN
  -- Выберем не саму таблицу room, а ссылку на нее.
  SELECT REF(r)
      INTO v_RoomRef
      FROM rooms r
      WHERE ID = 99993;
END;
```

**Операция DEREF** Операция DEREF возвращает исходный объект для заданной на него ссылки. Продолжим предыдущий пример:

```
-- Этот пример является частью файла refop.sql.
DECLARE
  v_RoomRef  REF RoomObj;
  V_Room     RoomObj;
BEGIN
  -- Выберем не саму таблицу room, а ссылку на нее.
  SELECT REF(r)
      INTO v_RoomRef
      FROM rooms r
      WHERE ID = 99993;

  -- При помощи операции DEREF (v_RoomRef) получим объект и
  -- обновим его. При этом возвращается локальный объект, не
  -- эквивалентный объекту, хранимому в таблице rooms.
  SELECT DEREF(v_RoomRef)
      INTO v_Room
      FROM dual;

  -- Обновим локальный объект.
  v_Room.room_number := 201;
END;
```

**Висячие ссылки** Если объект, на который указывает REF, удален, то ссылку называют *висячей* (dangling), поскольку теперь она указывает на несуществующий объект. Осуществлять обратную ссылку (DEREF) для висячей ссылки нельзя. Однако можно проверить, не является ли ссылка висячей. Для этого применяется предикат IS DANGLING, что иллюстрирует пример следующего оператора UPDATE:

```
BEGIN
  -- Установим для всех висячих ссылок NULL-значения.
  UPDATE classes
      SET room = NULL
      WHERE room IS DANGLING;
END;
```

▼ ВНИМАНИЕ Все рассмотренные операции (VALUE, REF, DEREF и IS DANGLING) можно использовать только в SQL-операторах. В процедурных операторах это запрещено.

## Конструкция RETURNING

В Oracle8 в операторах INSERT и UPDATE используется новая конструкция — RETURNING (возвращая). Она применяется для считывания информации из вновь введенной или обновленной строки; при этом формировать дополнительный запрос не требуется. Синтаксис конструкции RETURNING таков:

RETURNING *список_выбора* INTO *список_ввода*;

где *список_выбора* аналогичен списку выбора запроса, а *список_ввода* — это то же самое, что и оборот INTO запроса. Например, если в объектную таблицу вводится некоторый объект, то можно вернуть ссылку на вновь вводимый объект следующим образом:

```
-- Этот пример содержится в файле return.sql.
DECLARE
  v_ClassRef REF ClassObj;
BEGIN
  INSERT INTO CLASSES c VALUES
      (ClassObj('HIS', 101, 'History 101', 30, 0, 4, NULL))
      RETURNING REF(c) INTO v_ClassRef;
END;
```

# Методы MAP и ORDER

Для стандартных скалярных типов неявно задан способ упорядочения значений. Например, можно сравнить два числа и узнать, какое из них больше. Для объектов можно проверять только эквивалентность. Это делает невозможным использование объектных типов в таких конструкциях, как ORDER BY или DISTINCT, поскольку при их выполнении требуется упорядочить значения. Однако существует выход из этой ситуации — методы MAP и ORDER, используемые для упорядочения объектов.

## MAP

Метод MAP (метод карт) — это функция, возвращающая скалярный тип. При необходимости отсортировать объект базы данных можно вызвать функцию MAP и преобразовать объект к типу, который разрешается упорядочивать. Действие этого метода аналогично действию хэш-функции. Ниже приведен пример использования метода карт для объекта **RoomObj**:

```
-- Этот пример является частью файла tables8.sql.
CREATE OR REPLACE TYPE RoomObj AS OBJECT (
  ID NUMBER(5),
  ...
  -- Функция MAP используется для сортировки таблицы rooms.
  MAP MEMBER FUNCTION ReturnID RETURN NUMBER
);

CREATE OR REPLACE TYPE BODY RoomObj AS
  ...
  MAP MEMBER FUNCTION ReturnID RETURN NUMBER IS
  BEGIN
      RETURN SELF.ID;
  END ReturnID;
END RoomObj;
```

Метод MAP идентифицируется ключевым словом MAP, указываемым перед объявлением функции. Эта функция не может принимать какие-либо аргументы и возвращает только один из следующих скалярных типов: DATE, NUMBER, VARCHAR2, CHAR или REAL. Функция **ReturnID** возвращает идентификаторы объектов для аудиторий, определяющие порядок сортировки аудиторий. После создания этой функции можно выполнить, например, такой оператор SELECT:

```
SQL> SELECT VALUE(r)
  2    FROM rooms r
```

```
  3    ORDER BY id;
VALUE(R) (ID, BUILDING, ROOM_NUMBER, NUMBER_SEATS, DESCRIPTION)
--------------------------------------------------------------------
ROOMOBJ(99991, 'Building 7', 310, 50, 'Discussion Room E')
ROOMOBJ(99992, 'Building 7', 300, 75, 'Discussion Room D')
ROOMOBJ(99993, 'Music Building', 200, 1000, 'Concert Room')
ROOMOBJ(99994, 'Music Building', 100, 10, 'Music Practice Room')
ROOMOBJ(99995, 'Building 6', 170, 50, 'Discussion Room C')
ROOMOBJ(99996, 'Building 6', 160, 50, 'Discussion Room B')
ROOMOBJ(99997, 'Building 6', 150, 50, 'Discussion Room A')
ROOMOBJ(99998, 'Building 6', 101, 500, 'Small Lecture Hall')
ROOMOBJ(99999, 'Building 7', 310, 1000, 'Large Lecture Hall')
```

## ORDER

В качестве альтернативы методу карт можно использовать метод ORDER (метод упорядочения), который принимает один аргумент (объектного типа) и возвращает следующие значения:

- -1, если параметр больше SELF;
- 1, если параметр меньше SELF;
- 0, если параметр равен SELF.

*Метод ORDER используется аналогично методу MAP. Создадим метод ORDER для объекта* StudentObj; этот метод сортирует сведения о студентах по именам и фамилиям.

```
-- Этот пример является частью файла tables8.sql.
CREATE OR REPLACE TYPE StudentObj AS OBJECT (
  ID                NUMBER(5),
  First_name        VARCHAR2(20),
  Last_name         VARCHAR2(20),
  Major             VARCHAR2(30),
  Current_credits   NUMBER(3),

  ...
  -- Функция ORDER используется для сортировки сведений о студентах.
  ORDER MEMBER FUNCTION CompareStudent(p_Student IN StudentObj)
     RETURN NUMBER
);
CREATE OR REPLACE TYPE BODY StudentObj AS
  ...
  ORDER MEMBER FUNCTION CompareStudent(p_Student IN StudentObj)
     RETURN NUMBER IS
  BEGIN
     IF p_Student.last_name = SELF.last_name THEN
        IF p_Student.first_name < SELF.first_name THEN
           RETURN 1;
        ELSIF p_Student.first_name > SELF.first_name THEN
           RETURN -1;
        ELSE
           RETURN 0;
        END IF;
     ELSE
        IF p_Student.last_name < SELF.last_name THEN
           RETURN 1;
        ELSIF p_Student.last_name > SELF.last_name THEN
           RETURN -1;
        ELSE
```

```
            RETURN 0;
          END IF;
        END IF;
    END CompareStudent;
END;
```

После создания метода ORDER выполним следующий оператор:

```
SQL> SELECT student
  2    FROM students
  3    ORDER BY student DESC;

STUDENT(ID, FIRST_NAME, LAST_NAME, MAJOR, CURRENT_CREDITS)
----------------------------------------------------------------
STUDENTOBJ(10005, 'Timothy', 'Taller', 'History', 0)
STUDENTOBJ(10000, 'Scott', 'Smith', 'Computer Science', 0)
STUDENTOBJ(10009, 'Rose', 'Riznit', 'Music', 0)
STUDENTOBJ(10010, 'Rita', 'Razmataz', 'Nutrition', 0)
STUDENTOBJ(10004, 'Patrick', 'Poll', 'History', 0)
STUDENTOBJ(10003, 'Manish', 'Murgratroid', 'Economics', 0)
STUDENTOBJ(10001, 'Margaret', 'Mason', 'History', 0)
STUDENTOBJ(10002, 'Joanne', 'Junebug', 'Computer Science', 0)
STUDENTOBJ(10008, 'Ester', 'Elegant', 'Nutrition', 0)
STUDENTOBJ(10007, 'David', 'Dinsmore', 'Music', 0)
STUDENTOBJ(10006, 'Barbara', 'Blues', 'Economics', 0)
```

## Рекомендации по использованию методов MAP и ORDER

- Для конкретного объектного типа можно задавать либо метод MAP, либо метод ORDER; нельзя задавать оба этих метода одновременно.
- Метод MAP более эффективен, так как он преобразует все множество объектов к более простому типу (как и хэш-функция), который затем может быть применен для сортировки значений. При использовании объекта ORDER за один раз могут сравниваться только два объекта, и поэтому данный метод нужно вызывать повторно.
- Если не используются ни метод MAP, ни метод ORDER, объекты можно проверять только на эквивалентность и только в SQL-операторах. Методы MAP и ORDER дают возможность сортировать объекты, а также сравнивать их в процедурных операторах.

# Итоги

В начале этой главы рассматривались общие положения объектно-ориентированной методологии разработки приложений, а также способы ее реализации в Oracle с помощью объектных типов. Кроме того, обсуждался синтаксис описания объектных типов и методов и синтаксис создания таблиц базы данных, содержащих объекты-строки или объекты-столбцы. Завершилась глава рассказом о ссылках на объекты и о выполнении таких ссылок. В следующей главе речь пойдет о сборных конструкциях — объектах специального типа.

# Глава 12

## *Сборные конструкции*

**PL/SQL 8.0 ... и ВЫШЕ**

В PL/SQL имеются типы данных, позволяющие оперировать несколькими переменными как единым целым. Такие типы данных называются сборными конструкциями (collections). В PL/SQL версии 2 существует лишь один тип сборных конструкций — таблица PL/SQL (см. главу 2). В PL/SQL 8.0 к этому типу добавлено еще два — вложенные таблицы и изменяемые массивы. Каждый из указанных типов можно рассматривать в качестве объектного типа со своими атрибутами и методами. В данной главе обсуждаются свойства этих новых объектных типов.

# Вложенные таблицы

Вложенные таблицы (nested tables) очень похожи на таблицы PL/SQL (см. главу 2). В Oracle8 последние обычно называются индексными таблицами (index-by tables). Вложенные таблицы расширяют функциональные возможности индексных таблиц, предоставляя дополнительные методы сборных конструкций (называемые атрибутами индексных таблиц). Кроме того, вложенные таблицы можно хранить в таблице базы данных (именно поэтому они называются вложенными). Ими можно управлять непосредственно из SQL, и для них предусмотрены дополнительные исключительные ситуации.

В остальном использование вложенных таблиц ничем не отличается от использования таблиц PL/SQL. Вложенную таблицу можно рассматривать как таблицу базы данных, содержащую два столбца: столбец ключей и столбец значений (см. главу 3). Как и индексные, вложенные таблицы могут быть разреженными, и их ключи необязательно должны быть последовательными.

## Объявление вложенной таблицы

Синтаксис создания вложенной таблицы выглядит следующим образом:

TYPE *имя_таблицы* TABLE OF *тип_таблицы* [NOT NULL];

где *имя_таблицы* — это имя нового типа, а *тип_таблицы* — тип каждого элемента вложенной таблицы. *Тип_таблицы* может быть стандартным типом, объектным типом (определяемым пользователем) или выражением, в котором используется %TYPE.

**▼ ВНИМАНИЕ**

Единственное отличие индексной таблицы от вложенной заключается в присутствии в операторе создания таблицы конструкции INDEX BY BINARY_INTEGER. Если она не указана, создаваемый тип является типом вложенной таблицы и ему присущи свойства Oracle8, описанные в этой главе. Если же эта конструкция указана, то создаваемый тип — тип индексной таблицы со свойствами, присущими таблице PL/SQL Oracle7 (см. главу 3).

Ниже приведен раздел объявлений некоторого блока, демонстрирующий правильное объявление вложенных таблиц.

```
-- Этот пример является частью файла nested.sql.
DECLARE
  -- Создадим табличный тип на основе объектного типа.
  TYPE t_ClassesTab IS TABLE OF ClassObj;

  -- Тип, основанный на %ROWTYPE.
  TYPE t_StudentsTab IS TABLE OF students%ROWTYPE;

  -- Переменные, которые имеют типы, созданные выше.
  v_ClassList t_ClassesTab;
  v_StudentList t_StudentsTab;
```

**▼ ВНИМАНИЕ**

Все объектные типы и таблицы, указанные в этом примере (**ClassObj**, **students** и т.д.), описаны в файле tables8.sql.

### Инициализация таблиц

Если таблица объявляется так, как в предыдущем примере, то, подобно объектному типу, она автоматически инициализируется NULL-значениями. При попытке присвоить некоторые значения NULL-таблице возвращается сообщение об ошибке "ORA-6531: Reference to uninitialized collection" (ссылка на

неинициализированную сборную конструкцию. – *Прим. пер.*). Эта ошибка соответствует предопределенной исключительной ситуации COLLECTION_IS_NULL. Продолжим рассмотренный выше пример:

```
-- Этот пример является частью файла nested.sql.
BEGIN
  -- При выполнении этой операции присваивания будет установлена
  -- исключительная ситуация COLLECTION_IS_NULL, так как таблице
  -- v_ClassList автоматически присваиваются NULL-значения.
  v_ClassList(1) := ClassObj('HIS', 101, 'History 101', 30, 0, 4,
NULL);
END;
```

Так как же инициализировать вложенную таблицу? Это делается при помощи функции-конструктора. Как и конструктор объектного типа, конструктор вложенной таблицы имеет то же имя, что и сама таблица. Однако в качестве аргумента он принимает список элементов, каждый из которых должен иметь тип, совместимый с типом таблицы. Следующий пример иллюстрирует использование конструкторов вложенных таблиц:

```
-- Этот пример содержится в файле tconstr.sql.
DECLARE
  TYPE t_NumbersTab IS TABLE OF NUMBER;

  -- Создадим таблицу с одним элементом.
  v_Tab1 t_NumbersTab := t_NumbersTab(-1);

  -- Создадим таблицу с пятью элементами.
  v_Primes t_NumbersTab := t_NumbersTab(1, 2, 3, 5, 7);

  -- Создадим таблицу без элементов.
  v_Tab2 t_NumbersTab := t_NumbersTab();
BEGIN
  -- Присвоим значение v_Tab1(1). При этом значение, заданное для
  -- v_Tab(1) при ее инициализации (-1), будет заменено.
  v_Tab1(1) := 12345;
END;
```

**Пустые таблицы** Обратите внимание на объявление **v_Tab2** в предыдущем блоке:

```
-- Создадим таблицу без элементов.
  v_Tab2 t_NumbersTab := t_NumbersTab();
```

**v_Tab2** инициализируется вызовом конструктора без аргументов. При этом создается таблица без элементов, однако она не становится цельной NULL-таблицей. Проиллюстрируем вышесказанное на примере:

```
-- Этот пример содержится в файле nulltab.sql.
DECLARE
  TYPE t_WordsTab IS TABLE OF VARCHAR2(50);

  -- Создадим NULL-таблицу.
  v_Tab1 t_WordsTab;

  -- Создадим таблицу с одним элементом, значение которого NULL.
  v_Tab2 t_WordsTab := t_WordsTab();
BEGIN
  IF v_Tab1 IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('v_Tab1 is NULL');
  ELSE
      DBMS_OUTPUT.PUT_LINE('v_Tab1 is not NULL');
```

```
    END IF;

    IF v_Tab2 IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('v_Tab2 is NULL');
    ELSE
        DBMS_OUTPUT.PUT_LINE('v_Tab2 is not NULL');
    END IF;
END;
```

Если выполнить этот блок с командой set serveroutput on в SQL*Plus (о модуле DBMS_OUTPUT см. главу 14), получится следующий результат:

```
v_Tab1 is NULL
v_Tab2 is not NULL
```

**Ключи при инициализации** При инициализации таблицы с помощью конструктора ее элементы последовательно нумеруются в диапазоне от 1 до числа элементов, указанных в вызове конструктора. Во время последующей обработки хранимые в некоторых ключах значения могут быть удалены (с помощью метода DELETE, описанного ниже в этой главе). Когда вложенная таблица считывается в базе данных (см. раздел "Вложенные таблицы в базе данных" ниже), ключи перенумеровываются, если необходимо, чтобы они были последовательными, как при инициализации.

## Добавление элементов в существующую таблицу

Хотя для таблицы не устанавливается никаких ограничений, нельзя присвоить значение еще не существующему элементу, так как это приведет к увеличению размера таблицы. Если попытаться это сделать, PL/SQL выдаст сообщение об ошибке "ORA-6533: Subscript beyond count" (неправильный индекс. — *Прим. пер.*), которая эквивалентна предопределенной исключительной ситуации SUBSCRIPT_BEYOND_COUNT. Приведем пример:

```
-- Этот пример содержится в файле tassign.sql.
DECLARE
  TYPE t_NumbersTab IS TABLE OF NUMBER;
  v_Numbers t_NumbersTab := t_NumbersTab(1, 2, 3);
BEGIN
  -- Таблица v_Numbers инициализирована как состоящая из 3-х
  -- элементов. Поэтому следующие операции присваивания правильны.
  v_Numbers(1) := 7;
  v_Numbers(2) := -1;

  -- Однако эта операция присваивания приводит к ошибке ORA-6533.
  v_Numbers(4) := 4;
END;
```

▼ СОВЕТУЕМ Можно увеличить размер вложенной таблицы с помощью метода EXTEND, описанного ниже в этой главе.

# Вложенные таблицы в базе данных

Вложенные таблицы могут храниться в виде столбцов базы данных, т.е. вся вложенная таблица размещается в одной строке таблицы базы данных, а каждая строка таблицы базы данных может содержать собственную вложенную таблицу. Для сохранения вложенной таблицы в базе данных необходимо создать тип вложенной таблицы при помощи оператора CREATE TYPE, а не TYPE в блоке PL/SQL. При использовании оператора CREATE TYPE тип сохраняется в словаре данных и поэтому доступен для использования в качестве типа-столбца. Пример создания вложенной таблицы как столбца базы данных приведен ниже (здесь: title — название_книги, author — автор, catalog_number — номер_в_каталоге, BookList — СписокКниг, course_material — учебный_материал_курса, department — факультет, course — курс, required_reading — обязательное_чтение. — *Прим. пер.*).

```
-- Этот пример является частью файла tables8.sql.
CREATE TYPE BookObj AS OBJECT (
```

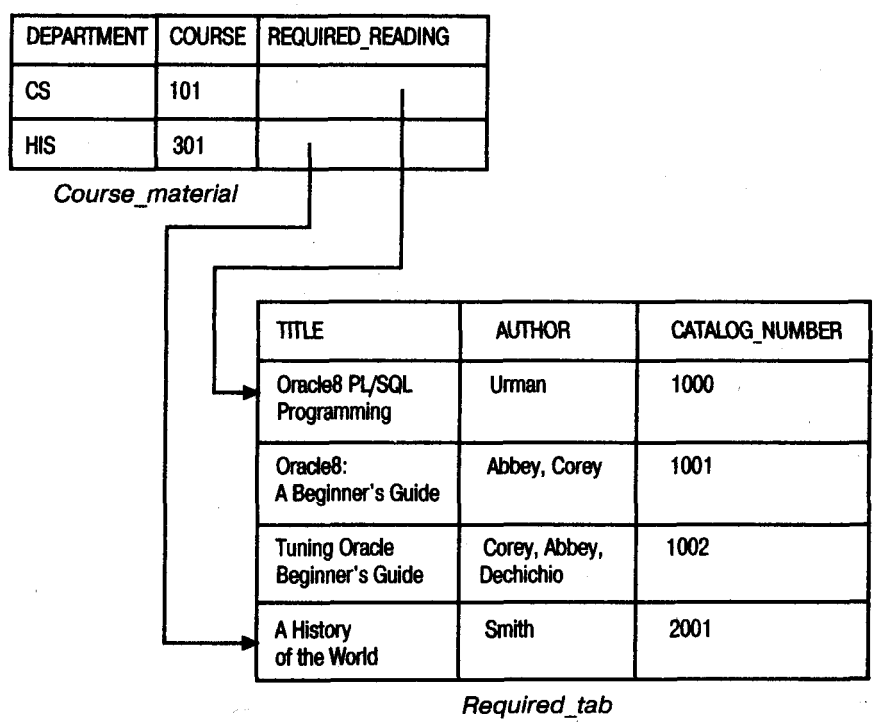
```
    Title            VARCHAR2(40),
    Author           VARCHAR2(40),
    Catalog_number   NUMBER(4)
);

CREATE TYPE BookList AS TABLE OF BookObj;

CREATE TABLE course_material (
    Department       CHAR(3),
    Course           NUMBER(3),
    required_reading BookList)
    NESTED TABLE required_reading STORE AS required_tab;
```

Следует сказать несколько слов по поводу приведенного листинга и создания вложенных таблиц в базе данных:

- Табличный тип создается с помощью оператора CREATE TYPE и поэтому может храниться в словаре данных.
- Табличный тип используется при описании таблицы так же, как и объект-столбец.
- Для каждой вложенной таблицы, размещенной в некоторой таблице базы данных, необходимо указывать конструкцию NESTED TABLE, которая определяет имя таблицы хранения.

*Таблица хранения* (store table) создается системой и используется для хранения фактических данных вложенной таблицы. Эти данные не встраиваются в оставшиеся столбцы таблицы, а располагаются отдельно. Реально в столбце **required_reading** будет храниться ссылка на таблицу **required_tab**, в которой и будет находиться список книг. Место хранения таблицы **course_material** показано на рис. 12.1. Для каждой строки таблицы **course_material** в столбце **required_reading** содержится ссылка на соответствующую строку таблицы **required_tab**.

**Рис. 12.1.** *Вложенные таблицы в базе данных*



▼ ВНИМАНИЕ Таблица хранения (в рассмотренном примере это **required_tab**) может находиться в другой схеме и иметь параметры хранения, отличные от основной таблицы. Таблицу хранения можно описать и разместить в **user_tables**, но обратиться к ней непосредственно нельзя. При попытке обратиться к ней с запросом или модифицировать ее будет возвращена ошибка Oracle "ORA-22812: cannot reference nested table column's storage table" (нельзя ссылаться на таблицу хранения столбца вложенной таблицы. — *Прим. пер.*). Работа с содержимым таблицы хранения осуществляется через SQL-операторы, выполняемые над основной таблицей.

## Работа с таблицей целиком

Над вложенной таблицей, хранящейся в таблице базе данных, можно выполнять различные операции, причем как над таблицей целиком, так и над отдельными ее строками. В любом случае можно использовать SQL-операторы. Вложенная таблица, как и обычная, в базе данных не упорядочена. Индексы для вложенных таблиц можно использовать только в PL/SQL.

**INSERT** Для ввода таблицы в строку базы данных используется оператор INSERT. Покажем это на примере. Обратите внимание, что в PL/SQL таблица сначала создается и инициализируется, и только потом вводится в базу данных.

```
-- Этот пример содержится в файле tinsert.sql.
DECLARE
  v_Books Booklist :=
      BookList(BookObj('A History of the World', 'Smith', 2001));
BEGIN
  -- Введем информацию вновь создаваемой вложенной
  -- таблицы, состоящей из 3-х элементов.
  INSERT INTO course_material VALUES (
  'CS', 101,
      BookList(BookObj('Oracle8 PL/SQL Programming',
              'Urman', 1000),
      BookObj('Oracle8: A Beginner''s Guide',
              'Abbey, Corey', 1001),
      BookObj('Tuning Oracle',
              'Corey, Abbey, Dechichio', 1002)));

  -- Введем информацию ранее инициализированной вложенной
  -- таблицы, состоящей из 1-го элемента.
  INSERT INTO course_material VALUES (
      'HIS', 301, v_Books);
END;
```

После выполнения приведенного примера в таблице **course_material** будут содержаться значения, показанные на рис. 12.1.

**UPDAT** Аналогично для модификации хранимых таблиц используется оператор UPDATE. В следующем примере обновляется столбец required_reading для группы History 301:

```
-- Этот пример содержится в файле tupdate.sql.
DECLARE
  v_Books Booklist :=
      BookList(BookObj('A History of the World', 'Smith', 2001),
               BookObj('Another World History', 'Jones', 2002));
BEGIN
  UPDATE course_material
      SET required_reading = v_Books
      WHERE department = 'HIS'
      AND course = 301;
END;
```

**DELETE** С помощью оператора DELETE можно удалить строку, содержащую вложенную таблицу, что показано в следующем примере:

```
-- Этот пример содержится в файле tdelete.sql.
BEGIN
  -- Удалим сведения о книгах, обязательных для чтения для всех
  -- учебных курсов истории.
  DELETE FROM course_material
      WHERE department = 'HIS';
END;
```

**SELECT** Когда вложенная таблица считывается в переменную PL/SQL, ей присваиваются ключи со значениями от 1 и до числа, равного числу элементов таблицы (это те же ключи, которые были заданы конструктором таблицы). Значение этого числа можно узнать методом COUNT, который обсуждается ниже в данной главе. Проиллюстрируем вышесказанное на примере:

```
-- Этот пример содержится в файле tselect.sql.
DECLARE
  v_Books course_material.required_reading%TYPE;
  v_Course course_material.course%TYPE;
  v_Department course_material.department%TYPE;

  CURSOR c_AllBooks IS
      SELECT required_reading, course, department
        FROM course_material;
BEGIN
  -- Последовательно просмотрим информацию обо всех учебных курсах и
  -- выведем на экран (с помощью DBMS_OUTPUT) названия нужных книг.
  OPEN c_AllBooks;

  LOOP
      -- Выберем в этой строке поля всех столбцов, в том числе и всю
      -- вложенную таблицу, хранимую в required_reading.
      FETCH c_AllBooks INTO v_Books, v_Course, v_Department;
      EXIT WHEN c_AllBooks%NOTFOUND;

      DBMS_OUTPUT.PUT('Required reading for ' || v_Department || ' ');
      DBMS_OUTPUT.PUT_LINE(v_Course || ':');

      -- Последовательно просмотрим всю выбранную таблицу, выводя на экран
      -- каждую строку.
      FOR v_Index IN 1..v_Books.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(' ' || v_Books(v_Index).title);
      END LOOP;
  END LOOP;

  CLOSE c_AllBooks;
END;
```

Если предположить, что выполнены все операторы INSERT и UPDATE (но не DELETE), рассмотренные в предыдущих примерах, то при выполнении этого блока в SQL*Plus получается следующий результат:

```
Required reading for CS 101:
  Oracle8 PL/SQL Programming
  Oracle8: A Beginner's Guide
  Tuning Oracle
```

```
Required reading for HIS 301:
  A History of the World
  Another World History
```

▼ ВНИМАНИЕ Обратите внимание, что ни в одном из рассмотренных примеров вложенные таблицы не были указаны в условии WHERE. Это ограничение налагается на использование вложенных таблиц. Поскольку методы MAP и ORDER не определены, вложенные таблицы нельзя проверять на эквивалентность, что необходимо для условия WHERE. Это означает также, что вложенные таблицы невозможно применять там, где требуется неявное сравнение значений, например в конструкциях ORDER BY, GROUP BY или DISTINCT.

### Работа с отдельными строками

В нескольких последних примерах хранимые вложенные таблицы сначала модифицировались в PL/SQL, а затем изменялись в базе данных. В Oracle8 существует операция THE, позволяющая манипулировать вложенной таблицей, хранимой в таблице базы данных, с помощью не PL/SQL, а DML. В качестве аргумента операции THE используется подзапрос, а возвращает вложенную таблицу, пригодную для обработки оператором DML. Подзапрос должен возвращать один из столбцов вложенной таблицы. Например, с помощью следующего оператора UPDATE к номерам книг, необходимых группе CS 101, добавляется 10:

```
UPDATE THE(SELECT required_reading
           FROM course_material
           WHERE department = 'CS' AND course = 101)
SET catalog_number = catalog_number + 10;
```

## Вложенные и индексные таблицы

Вложенные таблицы во многом похожи на индексные таблицы Oracle7, например:

- Структура табличных типов данных этих таблиц одинакова.
- Обращение к элементам таблиц обоих типов осуществляется с помощью индексов.
- В состав методов, применяемых для вложенных таблиц, входят все атрибуты индексных таблиц версии 2.3.
- Программный текст, написанный для индексных таблиц, работает и для вложенных таблиц.

Однако существует и ряд серьезных различий:

- С вложенными таблицами можно работать при помощи SQL и сохранять их в базе данных, в то время как с индексными таблицами этого делать нельзя.
- Диапазон возможных значений для индексов вложенных таблиц 1..2147483647, а для индексных таблиц – -2147483647..2147483647, т.е. для индексных таблиц можно использовать отрицательные индексы, а для вложенных – нельзя;
- Вложенные таблицы могут целиком становиться NULL-таблицами (это проверяется при помощи операции IS NULL).
- Для вложенных таблиц существует ряд дополнительных методов, например EXTEND и TRIM (см. раздел "Методы сборных конструкций" ниже).

# Изменяемые массивы

*Изменяемый массив, или массив с переменной длиной* (varray – varying array или variable length array) – это тип данных, практически идентичный массиву в языке программирования C или Pascal. Синтаксически обращение к массиву происходит почти так же, как к вложенной или индексной таблице. Однако реализован массив по-другому. Структура массива – это не разреженная структура данных без верхней границы, как у таблицы: элементы вводятся в массив начиная с индекса 1 и до максимального значения, заданного в описании типа изменяемого массива.

Место хранения изменяемого массива определяется так же, как в языках C или Pascal, в отличие от места хранения вложенной таблицы, которая больше похожа на таблицу базы данных.

## Объявление изменяемого массива

Тип изменяемого массива объявляется следующим образом:

TYPE *имя_типа* IS {VARRAY | VARYING ARRAY} (*максимальный_размер*)
OF *тип_элементов* [NOT NULL];

*имя_типа* – это имя нового типа изменяемого массива, *максимальный_размер* – целое число, определяющее максимальное количество элементов изменяемого массива, а *тип_элементов* – скалярный тип, тип записи или объектный тип PL/SQL. Кроме того, тип_элементов можно указать при помощи %TYPE, но он не может быть типом BOOLEAN, NCHAR, NCLOB, NVARCHAR2, REF CURSOR, TABLE или типом другого изменяемого массива. Приведем пример правильного использования типов изменяемых массивов:

```
DECLARE
  TYPE t_BookList IS VARRAY(25) OF BookObj;
  TYPE t_Numbers IS VARRAY(10) OF NUMBER(3) NOT NULL;
  TYPE t_Students IS VARRAY(100) OF students%ROWTYPE;
```

## Инициализация изменяемых массивов

Как и таблицы, изменяемые массивы инициализируются с помощью функций-конструкторов. Проиллюстрируем это на примере:

```
-- Этот пример содержится в файле vconstr.sql.
DECLARE
  -- Опишем тип изменяемого массива.
  TYPE t_Numbers IS VARRAY(20) OF NUMBER(3);

  -- Объявим изменяемый массив как NULL-массив.
  v_NullList t_Numbers;

  -- В этом массиве содержится 2 элемента.
  v_List1 t_Numbers := t_Numbers(1, 2);

  -- В этом массиве находится 1 элемент, содержащий NULL-значение.
  v_List2 t_Numbers := t_Numbers(NULL);
BEGIN
  IF v_NullList IS NULL THEN
     DBMS_OUTPUT.PUT_LINE('v_NullList is NULL');
  END IF;

  IF v_List2(1) IS NULL THEN
     DBMS_OUTPUT.PUT_LINE('v_List2(1) is NULL');
  END IF;
END;
```

При выполнении этого блока в SQL*Plus будут получены такие результаты:

```
v_NullList is NULL
v_List2(1) is NULL
```

## Работа с элементами изменяемых массивов

Как и для вложенной таблицы, начальный размер изменяемого массива определяется числом элементов, указываемых в конструкторе при объявлении массива. Как и в случае с вложенной таблицей, присваивание значений элементам, не попадающим в указанный диапазон, приводит к ошибке "ORA-6533: Subscript beyond count" (неправильный индекс. – *Прим. пер.*) Проиллюстрируем вышесказанное на примере:

```
-- Этот пример содержится в файле vassign.sql.
DECLARE
  TYPE t_Strings IS VARRAY(5) OF VARCHAR2(10);
```

```
  -- Объявим изменяемый массив, состоящий из 3-х элементов.
  -- Максимальный размер этого массива — 5 элементов.
  v_List t_Strings := t_Strings('Scott', 'David', 'Urman');
BEGIN
  -- Значение индекса в диапазоне от 1 до 3, поэтому данная операция
  -- присваивания верна.
  v_List(2) := 'DAVID';

  -- Значение вне диапазона; устанавливается ORA-6533.
  v_List(4) := '!!!';
END;
```

▼ СОВЕТУЕМ Подобно вложенным таблицам, размер изменяемого массива можно увеличивать при помощи метода EXTEND, описанного ниже в этой главе. Однако в отличие от вложенных таблиц изменяемый массив не может увеличиться сверх размера, указанного как максимальный при объявлении типа изменяемого массива.

## Изменяемые массивы в базе данных

Изменяемые массивы, как и вложенные таблицы, можно хранить в столбцах базы данных. Однако в отличие от вложенных таблиц работать можно только с изменяемым массивом целиком — модифицировать его отдельные элементы (с помощью операции THE) нельзя. Рассмотрим пример:

```
-- Этот пример является частью файла tables8.sql.
CREATE OR REPLACE TYPE BookList2 AS VARRAY(10) OF BookObj;

CREATE TABLE checked_out (
  student_id number(5),
  books BookList2
);
```

В таблице **checked_out** (отмеченные_книги) будет содержаться список книг, отмеченных в библиотеке для каждого студента. Следует сделать ряд замечаний относительно создания хранимых изменяемых массивов этого примера:

- Тип должен быть известен базе данных и храниться в словаре данных, поэтому оператор CREATE TYPE обязателен (тип не может быть локальным по отношению к блоку PL/SQL).
- В каждой строке таблицы **checked_out** будет находиться изменяемый массив, содержащий сведения не более чем о 10 книгах. Место хранения массива совпадает с местом хранения строки базы данных, и информация изменяемого массива встраивается для хранения в табличные данные.

### Работа с хранимыми изменяемыми массивами

Для модификации хранимого изменяемого массива необходимо сначала считать его в переменную PL/SQL. Затем можно изменить эту переменную и ввести ее информацию обратно в таблицу. Такой метод показан на примере процедуры **CheckedOut**:

```
-- Этот пример содержится в файле checkout.sql.
CREATE OR REPLACE PROCEDURE CheckOut(
  p_StudentID IN NUMBER,
  p_NewBook IN BookObj) AS

  v_Books BookList2;
  v_Found BOOLEAN := FALSE;
  v_Book BookObj;
BEGIN
  -- Сначала определим текущий список книг, отмеченных этим студентом.
  BEGIN
      SELECT books
        INTO v_Books
```

```
        FROM checked_out
        WHERE student_id = p_StudentID;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
          -- Студент не отметил ни одной книги.
          v_Books := BookList2(NULL);
    END;

    -- Просмотрим список и определим, нет ли у этого студента данной книги.
    FOR v_Counter IN 1..v_Books.COUNT LOOP
        v_Book := v_Books(v_Counter);

        IF v_Book.catalog_number = p_NewBook.catalog_number THEN
          RAISE_APPLICATION_ERROR(-20001, 'Book is already checked out');
        END IF;
    END LOOP;

    -- Убедимся, что еще есть место для книг.
    IF v_Books.COUNT = v_Books.LIMIT THEN
        RAISE_APPLICATION_ERROR(-20001, 'Cannot check out any more books');
    END IF;

    -- Отметим книгу, добавив ее к списку
    v_Books.EXTEND;
    v_Books(v_Books.COUNT) := p_NewBook;

    -- и вернув обратно в базу данных.
    UPDATE checked_out
        SET books = v_Books
        WHERE student_id = p_StudentID;
    IF SQL%NOTFOUND THEN
        INSERT INTO checked_out (student_id, books)
          VALUES (p_StudentID, v_Books);
    END IF;
  END CheckOut;
```

**Можно вызвать CheckOut из блока, например, так:**

```
  -- Этот пример содержится в файле co.sql.
  DECLARE
    v_RequiredBooks BookList;
    v_Book BookObj;

  BEGIN
    SELECT required_reading
        INTO v_RequiredBooks
        FROM course_material
        WHERE department = 'CS'
        AND course = 101;

    FOR v_Counter IN 1..v_RequiredBooks.COUNT LOOP
        v_Book := v_RequiredBooks(v_Counter);
        CheckOut(1005, v_Book);
    END LOOP;
  END;
```

## Изменяемые массивы и вложенные таблицы

Как изменяемые массивы, так и вложенные таблицы являются сборными конструкциями, и поэтому многие их свойства достаточно схожи:

- Оба типа обеспечивают доступ к отдельным элементам при помощи индексов.
- Оба типа можно хранить в таблицах базы данных.

Однако имеются и некоторые отличия:

- Для изменяемых массивов задается максимальный размер, а для вложенных таблиц — нет.
- Изменяемые массивы встраиваются в содержащую их таблицу, а вложенные таблицы хранятся в отдельной таблице, которая может иметь собственные, отличные от других, характеристики хранения информации.
- При хранении в базе данных изменяемые массивы сохраняют упорядочение, а также значения индексов своих элементов, а вложенные таблицы — нет.
- Отдельные элементы вложенной таблицы могут быть удалены (с помощью метода TRIM, описанного в следующем разделе), что приводит к сокращению ее размера. Размер же массива всегда постоянен.

# Методы сборных конструкций

Сборные конструкции — это объектные типы, и поэтому для них определяется ряд методов, в частности атрибуты, используемые для таблиц PL/SQL (версии 2.3 и выше; см. главу 2). Эти методы можно применять как для вложенных таблиц, так и для изменяемых массивов, если явно не указано иное. Методы сборных конструкций можно вызывать только из процедурных, а не из SQL-операторов.

Во всех рассматриваемых ниже примерах считается, что существуют следующие объявления:

```
-- Этот пример является частью файла tables8.sql.
CREATE OR REPLACE TYPE NumTab AS TABLE OF NUMBER;
CREATE OR REPLACE TYPE NumVar AS VARRAY(25) OF NUMBER;
```

Методы перечислены в таблице 12.1 и описаны в последующих разделах.

**ТАБЛИЦА 12.1. Методы сборных конструкций**

| Метод | Описание |
|---|---|
| EXISTS | Определяет, существует ли некоторый элемент сборной конструкции |
| COUNT | Возвращает число элементов сборной конструкции |
| LIMIT | Возвращает максимальное число элементов сборной конструкции |
| FIRST и LAST | Возвращает первый (последний) элемент сборной конструкции |
| NEXT и PRIOR | Возвращает элемент сборной конструкции, следующий (предыдущий) по отношению к данному элементу |
| EXTEND | Добавляет элементы в сборную конструкцию |
| TRIM | Удаляет элементы, начиная с конца\ сборной конструкции |
| DELETE | Удаляет указанные элементы из сборной конструкции |

## EXISTS

Метод EXISTS используется для определения реального существования элемента, на который производится ссылка. Синтаксис этого метода следующий:

EXISTS(*n*)

где *n* — целочисленное выражение. Если элемент с индексом *n* существует, возвращается TRUE (даже если элемент является NULL-значением); если же значение n лежит вне требуемого диапазона, EXISTS не устанавливает исключительную ситуацию SUBCRIPT_OUTSIDE_LIMIT, а возвращает FALSE. Методы

EXISTS и DELETE часто применяются для работы с разреженными вложенными таблицами. Проиллюстрируем использование EXISTS на примере:

```
-- Этот пример содержится в файле exists.sql.
DECLARE
  v_Table NumTab := NumTab(-7, 14.3, 3.14159, NULL, 0);
  v_Count BINARY_INTEGER := 1;
BEGIN
  -- Последовательно просмотрим v_Table и выведем на экран список
  -- элементов, указав с помощью EXISTS конец цикла.
  LOOP
    IF v_Table.EXISTS(v_Count) THEN
      DBMS_OUTPUT.PUT_LINE('v_Table(' || v_Count || '): ' ||
                           v_Table(v_Count));
      v_Count := v_Count + 1;
    ELSE
      EXIT;
    END IF;
  END LOOP;
END;
```

Получим следующий результат:

```
v_Table(1): -7
v_Table(2): 14.3
v_Table(3): 3.14159
v_Table(4):
v_Table(5): 0
```

Метод EXISTS применим и для NULL-конструкции; в этом случае будет всегда возвращаться FALSE.

## COUNT

Метод COUNT возвращает текущее число элементов сборной конструкции в виде целого числа. COUNT не имеет аргументов и может применяться везде, где разрешено использовать целочисленные выражения. Проиллюстрируем использование этого метода на примере:

```
-- Этот пример содержится в файле count.sql.
DECLARE
  v_Table NumTab := NumTab(1, 2, 3);
  v_Varray NumVar := NumVar(-1, -2, -3, -4);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Table Count: ' || v_Table.COUNT);
  DBMS_OUTPUT.PUT_LINE('Varray Count: ' || v_Varray.COUNT);
END;
```

Получим следующий результат:

```
Table Count: 3
Varray Count: 4
```

Для изменяемых массивов метод COUNT эквивалентен методу LAST (см. ниже в этом разделе), так как элементы изменяемых массивов удалять нельзя. Однако из вложенной таблицы удалять элементы можно, поэтому для таблицы методы COUNT и LAST могут работать по-разному. COUNT наиболее полезен при выборе вложенной таблицы в базе данных, поскольку в этот момент число элементов таблицы неизвестно. При вычислении итогового значения COUNT игнорирует удаленные элементы.

## LIMIT

Метод LIMIT возвращает текущее максимальное число элементов сборной конструкции. Для вложенных таблиц максимальный размер не задается, поэтому для них LIMIT всегда возвращает NULL-значение. Проиллюстрируем работу метода LIMIT на примере:

```
-- Этот пример содержится в файле limit.sql.
DECLARE
  v_Table  NumTab  := NumTab(1,  2,  3);
  v_Varray NumVar  := NumVar(1234,  4321);
BEGIN
  -- Выведем максимальное и текущее число элементов сборных конструкций.
  DBMS_OUTPUT.PUT_LINE('Varray limit: ' || v_Varray.LIMIT);
  DBMS_OUTPUT.PUT_LINE('Varray count: ' || v_Varray.COUNT);
  IF v_Table.LIMIT IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Table limit is NULL');
  ELSE
      DBMS_OUTPUT.PUT_LINE('Table limit: ' || v_Table.LIMIT);
  END IF;
  DBMS_OUTPUT.PUT_LINE('Table count: ' || v_Table.COUNT);
END;
```

Получим следующий результат:

```
Varray limit: 25
Varray count: 2
Table limit is NULL
Table count: 3
```

Обратите внимание: пределом для изменяемого массива является 25 элементов, что определено в операторе CREATE TYPE, хотя в данный момент массив **v_Varray** содержит только 2 элемента. COUNT возвращает текущее число элементов (см. предыдущий раздел).

## FIRST и LAST

Метод FIRST возвращает индекс первого элемента сборной конструкции, а метод LAST – индекс последнего. Для изменяемого массива FIRST всегда возвращает 1, а LAST – значение COUNT, так как массив является плотным и его элементы удалять нельзя. Методы FIRST и LAST используются совместно с методами NEXT и PRIOR для последовательного просмотра сборных конструкций, как показывает пример из следующего раздела.

## NEXT и PRIOR

Методы NEXT и PRIOR возвращают увеличенное или уменьшенное ключевое значение сборной конструкции. Их синтаксис таков:

NEXT(*n*)
PRIOR(*n*)

где *n* – целочисленное выражение. PRIOR(*n*) возвращает ключ элемента, непосредственно предшествующего элементу *n*, а NEXT(*n*) – ключ элемента, следующего сразу же за элементом *n*. Если предшествующего или следующего элемента нет, NEXT или PRIOR возвращает NULL. В приведенном ниже примере показано, как можно использовать NEXT и PRIOR совместно с FIRST и LAST для последовательного просмотра вложенной таблицы:

```
-- Этот пример содержится в файле loops.sql.
DECLARE
  TYPE t_CharTab IS TABLE OF CHAR(1);
  v_Characters t_CharTab := t_CharTab('M', 'a', 'd', 'a', 'm',
                ',', ' ', 'I', '''', 'm', ' ', 'A', 'd', 'a', 'm');

  v_Index INTEGER;
```

```
BEGIN
   -- Последовательно просмотрим таблицу от начала до конца.
   v_Index := v_Characters.FIRST;
   WHILE v_Index <= v_Characters.LAST LOOP
       DBMS_OUTPUT.PUT(v_Characters(v_Index));
       v_Index := v_Characters.NEXT(v_Index);
   END LOOP;
   DBMS_OUTPUT.NEW_LINE;

   -- Последовательно просмотрим таблицу от конца до начала.
   v_Index := v_Characters.LAST;
   WHILE v_Index >= v_Characters.FIRST LOOP
       DBMS_OUTPUT.PUT(v_Characters(v_Index));
       v_Index := v_Characters.PRIOR(v_Index);
   END LOOP;
   DBMS_OUTPUT.NEW_LINE;
END;
```

## EXTEND

Метод EXTEND используется для добавления элементов в конец вложенной таблицы. Он имеет три формы:

EXTEND
EXTEND(*n*)
EXTEND(*n*,*i*)

EXTEND без аргументов просто добавляет NULL-элемент в конец таблицы, присваивая ему индекс LAST + 1. EXTEND(*n*) добавляет в конец таблицы *n* NULL-элементов, а EXTEND(*n*,*i*) добавляет в конец таблицы *n* копий элемента *i*. Если таблица была создана с ограничением NOT NULL, то можно применять только последнюю форму, так как при этом NULL-элементы не добавляются. Проиллюстрируем использование EXTEND на примере:

```
-- Этот пример содержится в файле extend.sql.
DECLARE
   v_Numbers NumTab := NumTab(1, 2, 3, 4, 5);
BEGIN
   -- При выполнении этой операции присваивания устанавливается
   -- SUBSCRIPT_BEYOND_COUNT, так как в v_Numbers содержится только
   -- 5 элементов.
   v_Numbers(26) := -7;
EXCEPTION
   WHEN SUBSCRIPT_BEYOND_COUNT THEN
       DBMS_OUTPUT.PUT_LINE('ORA-6533 raised');

       -- Можно решить эту проблему, добавив 30 элементов в v_Numbers.
       V_Numbers.EXTEND(30);

       -- А теперь выполним присваивание.
       v_Numbers(26) := -7;
END;
```

▼ ВНИМАНИЕ **Поскольку изменяемые массивы имеют фиксированный размер, метод EXTEND не оказывает на них никакого воздействия. Тем не менее использовать его не запрещено.**

Метод EXTEND работает с внутренним размером сборной конструкции, в котором учитываются все удаленные элементы. При удалении элемента (с помощью метода DELETE, описанного ниже) удаляются его данные но его ключ остается. Проиллюстрируем взаимодействие EXTEND и DELETE на примере:

```
-- Этот пример содержится в файле extdel.sql.
DECLARE
  -- Инициализируем таблицу 5-ю элементами.
  v_Numbers NumTab := NumTab(-2, -1, 0, 1, 2);

  -- Локальная процедура для вывода таблицы на экран.
  PROCEDURE Print(p_Table IN NumTab) IS
      v_Index INTEGER;
  BEGIN
      v_Index := p_Table.FIRST;
      WHILE v_Index <= p_Table.LAST LOOP
         DBMS_OUTPUT.PUT('Element ' || v_Index || ': ');
         DBMS_OUTPUT.PUT_LINE(p_Table(v_Index));
         v_Index := p_Table.NEXT(v_Index);
      END LOOP;
  END Print;

BEGIN
  DBMS_OUTPUT.PUT_LINE('At initialization, v_Numbers contains');
  Print(v_Numbers);

  -- Удалим элемент 3. При этом удаляется значение 0, но место его
  -- хранения остается.
  V_Numbers.DELETE(3);

  DBMS_OUTPUT.PUT_LINE('After delete, v_Numbers contains');
  Print(v_Numbers);

  -- Добавим в таблицу две копии элемента 1. При этом добавятся
  -- элементы 6 и 7.
  v_Numbers.EXTEND(2, 1);

  DBMS_OUTPUT.PUT_LINE('After extend, v_Numbers contains');
  Print(v_Numbers);

  DBMS_OUTPUT.PUT_LINE('v_Numbers.COUNT = ' || v_Numbers.COUNT);
  DBMS_OUTPUT.PUT_LINE('v_Numbers.LAST = ' || v_Numbers.LAST);
END;
```

Ниже приведен результат выполнения этого примера. Обратите внимание на значение COUNT и LAST после выполнения операций DELETE и EXTEND.

```
At initialization, v_Numbers contains
Element 1: -2
Element 2: -1
Element 3: 0
Element 4: 1
Element 5: 2
After delete, v_Numbers contains
Element 1: -2
Element 2: -1
Element 4: 1
Element 5: 2
After extend, v_Numbers contains
Element 1: -2
```

```
Element 2: -1
Element 4: 1
Element 5: 2
Element 6: -2
Element 7: -2
v_Numbers.COUNT = 6
v_Numbers.LAST = 7
```

## TRIM

TRIM используется для удаления элементов из конца вложенной таблицы. Поскольку размеры изменяемых массивов фиксированы, метод TRIM не оказывает на массивы воздействия. Этот метод имеет две формы:

TRIM
TRIM(*n*)

Без аргументов TRIM удаляет один последний элемент сборной конструкции. Во втором случае удаляются последние *n* элементов. Если *n* больше COUNT, устанавливается исключительная ситуация SUBSCRIPT_BEYOND_COUNT. После выполнения метода TRIM значение COUNT уменьшается, так как TRIM удаляет элементы конструкции.

Как и EXTEND, метод TRIM работает с внутренним размером сборной конструкции, т.е. учитывает все элементы, удаленные с помощью DELETE. Проиллюстрируем вышесказанное на примере:

```
-- Этот пример содержится в файле trim.sql.
DECLARE
  -- Инициализируем таблицу 7-ю элементами.
  v_Numbers NumTab := NumTab(-3, -2, -1, 0, 1, 2, 3);

  -- Локальная процедура для вывода таблицы на экран.
  PROCEDURE Print(p_Table IN NumTab) IS
     v_Index INTEGER;
  BEGIN
     v_Index := p_Table.FIRST;
     WHILE v_Index <= p_Table.LAST LOOP
       DBMS_OUTPUT.PUT('Element ' || v_Index || ': ');
       DBMS_OUTPUT.PUT_LINE(p_Table(v_Index));
       v_Index := p_Table.NEXT(v_Index);
     END LOOP;
     DBMS_OUTPUT.PUT_LINE('COUNT = ' || p_Table.COUNT);
     DBMS_OUTPUT.PUT_LINE('LAST = ' || p_Table.LAST);
  END Print;

BEGIN
  DBMS_OUTPUT.PUT_LINE('At initialization, v_Numbers contains');
  Print(v_Numbers);

  -- Удалим элемент 6.
  v_Numbers.DELETE(6);
  DBMS_OUTPUT.PUT_LINE('After delete , v_Numbers contains');
  Print(v_Numbers);

  -- Удалим последние 3 элемента. При этом будут удалены 1 и 3,
  -- а также место хранения (теперь пустое) 2.
  v_Numbers.TRIM(3);
  DBMS_OUTPUT.PUT_LINE('After trim, v_Numbers contains');
  Print(v_Numbers);
END;
```

Этот пример дает следующий результат:

```
At initialization, v_Numbers contains
Element 1: -3
Element 2: -2
Element 3: -1
Element 4: 0
Element 5: 1
Element 6: 2
Element 7: 3
COUNT = 7
LAST = 7
After delete, v_Numbers contains
Element 1: -3
Element 2: -2
Element 3: -1
Element 4: 0
Element 5: 1
Element 7: 3
COUNT = 6
LAST = 7
After trim, v_Numbers contains
Element 1: -3
Element 2: -2
Element 3: -1
Element 4: 0
COUNT = 4
LAST = 4
```

## DELETE

Метод DELETE удаляет один или более элементов из вложенной таблицы. Подобно TRIM, DELETE не оказывает воздействия на изменяемые массивы, так как их размеры фиксированы. Метод DELETE имеет три формы:

DELETE
DELETE(*n*)
DELETE(*m*,*n*)

Без аргументов DELETE удаляет все элементы таблицы; DELETE(*n*) удаляет элемент с индексом *n*, а DELETE(*m*,*n*) — все элементы, находящиеся между элементами с индексами *m* и *n* включительно. После выполнения DELETE значение COUNT уменьшается, отражая новый размер вложенной таблицы. Если удаляемый элемент не существует, DELETE не устанавливает какую-либо исключительную ситуацию, а просто пропускает его. Проиллюстрируем использование DELETE на примере:

```
-- Этот пример содержится в файле delete.sql.
DECLARE
  -- Инициализируем таблицу 10-ю элементами.
  v_Numbers NumTab := NumTab(10, 20, 30, 40, 50, 60, 70, 80, 90, 100);

  -- Локальная процедура для вывода таблицы на экран.
  PROCEDURE Print(p_Table IN NumTab) IS
     v_Index INTEGER;
  BEGIN
     v_Index := p_Table.FIRST;
     WHILE v_Index <= p_Table.LAST LOOP
        DBMS_OUTPUT.PUT('Element ' || v_Index || ': ');
        DBMS_OUTPUT.PUT_LINE(p_Table(v_Index));
```

```
        v_Index := p_Table.NEXT(v_Index);
      END LOOP;
      DBMS_OUTPUT.PUT_LINE('COUNT = ' || p_Table.COUNT);
      DBMS_OUTPUT.PUT_LINE('LAST = ' || p_Table.LAST);
  END Print;

BEGIN
  DBMS_OUTPUT.PUT_LINE('At initialization, v_Numbers contains');
  Print(v_Numbers);

  -- Удалим элемент 6.
  DBMS_OUTPUT.PUT_LINE('After delete(6), v_Numbers contains');
  V_Numbers.DELETE(6);
  Print(v_Numbers);

  -- Удалим элементы с 7 по 9.
  DBMS_OUTPUT.PUT_LINE('After delete(7,9), v_Numbers contains');
  v_Numbers.DELETE(7,9);
  Print(v_Numbers);
END;
```

Получим следующий результат:

```
At initialization, v_Numbers contains
Element 1: 10
Element 2: 20
Element 3: 30
Element 4: 40
Element 5: 50
Element 6: 60
Element 7: 70
Element 8: 80
Element 9: 90
Element 10: 100
COUNT = 10
LAST = 10
After delete (6), v_Numbers contains
Element 1: 10
Element 2: 20
Element 3: 30
Element 4: 40
Element 5: 50
Element 7: 70
Element 8: 80
Element 9: 90
Element 10: 100
COUNT = 9
LAST = 10
After delete (7,9), v_Numbers contains
Element 1: 10
Element 2: 20
Element 3: 30
Element 4: 40
Element 5: 50
Element 10: 100
```

```
COUNT = 6
LAST = 10
```

# Итоги

Сборные конструкции — вложенные таблицы и изменяемые массивы — полезны в любом языке программирования. Пользователь сам решает, какой тип сборной конструкции выбрать для собственных нужд. На этом обсуждение программных конструкций PL/SQL завершается. Теперь, когда получено представление о строительных блоках этого языка, перейдем к рассмотрению различных сред выполнения программ PL/SQL.

# Глава 13

## Среды выполнения программ PL/SQL

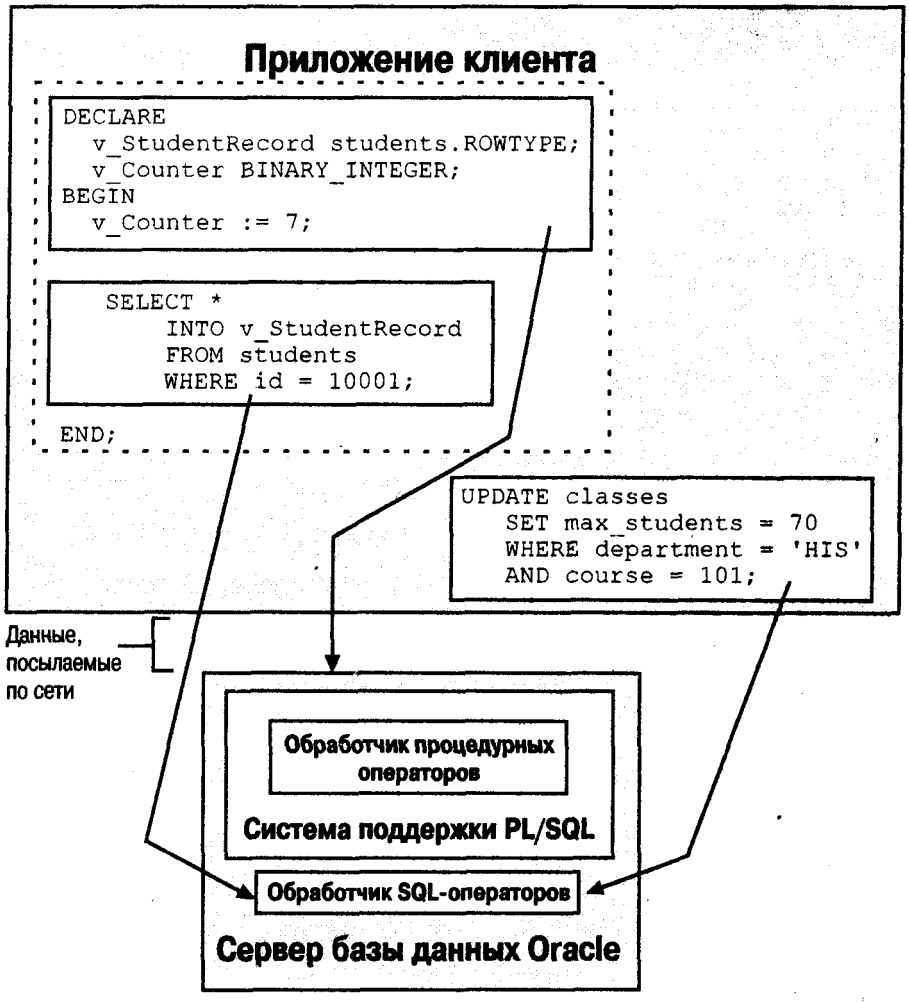
В главах 2 — 12 были изложены основные принципы построения PL/SQL. В этой главе рассматриваются различные среды выполнения программ, созданных на PL/SQL. В некоторых средах (например, в Oracle Forms или Procedure Builder) блоки PL/SQL можно выполнять на станции клиента, не взаимодействуя с сервером базы данных. В других средах (например, в предкомпиляторах или SQL-Station) эти блоки можно запускать на выполнение из клиентской программы, а выполнять на сервере. Использование средств для управления блоками PL/SQL полностью зависит от среды.

# Различные системы поддержки PL/SQL

Начиная с Oracle версии 6 с PL/SQL можно работать на сервере базы данных. Это означает, что как SQL-операторы, так и блоки PL/SQL можно посылать в базу данных и там обрабатывать. Как было показано в главе 1, в Oracle7 содержится PL/SQL версии 2, а в Oracle8 — PL/SQL версии 8. Приложение клиента, написанное с помощью средств разработки программ Oracle или средств других производителей, может направлять серверу как SQL-операторы, так и блоки PL/SQL. Примером такого клиентского приложения, в котором SQL-операторы и блоки PL/SQL вводятся в диалоговом режиме в ответ на подсказку SQL, а затем посылаются на сервер для выполнения, является программа SQL*Plus.

Такой метод демонстрируется на рис. 13.1. Приложение клиента посылает по сети на сервер блок PL/SQL (содержащий как процедурные, так и SQL-операторы) и отдельный SQL-оператор. На сервере SQL-оператор

**Рис. 13.1.**
*Система PL/SQL на сервере*

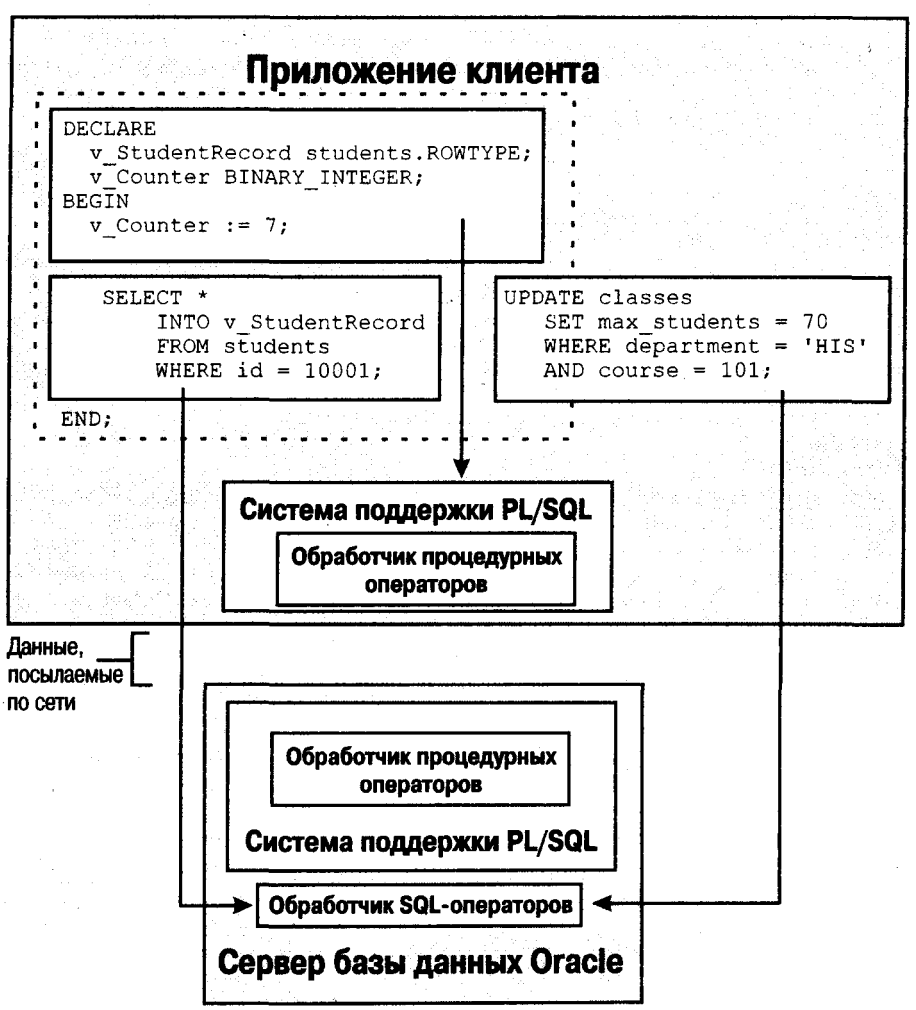


направляется непосредственно обработчику SQL-операторов, а для процедурных операторов блока (например, операции присваивания) выполняется синтаксический анализ и обработка системой поддержки PL/SQL. Все SQL-операторы блока (например, оператор SELECT) также направляются обработчику.

В дополнение к системе поддержки PL/SQL, размещенной на сервере, некоторые инструментальные средства Oracle обладают собственными такими системами. Средство разработки программ выполняется не на сервере, а на станции клиента, и здесь же запускается система поддержки PL/SQL этого средства. Когда на станции клиента установлена система поддержки PL/SQL, процедурные операторы блоков PL/SQL выполняются на стороне клиента и не направляются серверу. Например, программа Oracle Forms (часть программного пакета Developer 2000 корпорации Oracle) обладает собственной системой поддержки PL/SQL. Другие инструментальные средства из этого пакета, например Oracle Reports или Oracle Graphics, также включают такие системы. Клиентская система поддержки PL/SQL отличается от системы поддержки, установленной на сервере. Блоки PL/SQL содержатся в приложении клиента, создаваемом с помощью средств разработки программ. Например, в приложении Oracle Forms находятся триггеры и процедуры. Такие приложения выполняются на станции клиента, и только содержащиеся в них SQL-операторы посылаются для обработки на сервер. Все процедурные операторы выполняются локальной системой поддержки PL/SQL, находящейся на клиентской стороне (рис. 13.2).

Как и прежде, SQL-операторы, выдаваемые приложением (оператор UPDATE), посылаются по сети непосредственно на сервер обработчику SQL-операторов. Однако блоки PL/SQL обрабатываются локально клиентом. Все процедурные операторы (например, операция присваивания) обрабатываются лока-

**Рис. 13.2.**
*Система поддержки PL/SQL на станции клиента*



льно, а SQL-операторы блока PL/SQL (например, SELECT) посылаются на сервер. Логическое обоснование рассмотренной ситуации приводится ниже в разделе "PL/SQL на станции клиента".

## Замечания относительно PL/SQL на станции клиента

В архитектуре, приведенной на рис. 13.2, существуют две отдельные системы поддержки PL/SQL, взаимодействующие между собой. Например, триггер формы (запущенный в PL/SQL на станции клиента) может вызывать хранимую процедуру в базе данных (запущенной в PL/SQL на сервере). Такое взаимодействие осуществляется через вызовы удаленных процедур (RPC – remote procedure calls). Аналогичный алгоритм используется при взаимодействии двух систем поддержки PL/SQL, расположенных на двух различных серверах; здесь серверы обращаются друг к другу при помощи связей баз данных. Зависимости, существующие между различными объектами PL/SQL, рассмотрены в главе 7.

Две системы поддержки PL/SQL могут быть разных версий. К примеру, Developer 2000 версии 1.2 использует PL/SQL версии 1, в то время как сервер использует PL/SQL версии 2 (8 в случае Oracle8). Это значит, что средства, предлагаемые в PL/SQL версии 2 и выше, например определяемые пользователями таблицы и записи или тип данных CHAR фиксированной длины, невозможно использовать в PL/SQL на стороне клиента.

▼ СОВЕТУЕМ

Когда для приложений клиентов становится доступен PL/SQL 2, содержащиеся в этих приложениях блоки необходимо модернизировать в соответствии с синтаксисом и семантикой версии 2. Одно из основных отличий версий 1 и 2 заключается в работе с данными, имеющими тип CHAR: в PL/SQL версии 1 переменные типа CHAR и VARCHAR2 представляют собой последовательности символов переменной длины, а в версии 2 переменные типа CHAR имеют фиксированную длину, а данные типа VARCHAR2 — переменную. Для упрощения перехода от одной версии к другой не используйте переменные типа CHAR в PL/SQL на стороне клиента; применяйте вместо них переменные типов VARCHAR или VARCHAR2 (см. главу 3).

# PL/SQL на сервере

В этом разделе обсуждается выполнение блоков PL/SQL на сервере. Блоки могут передаваться серверу различными средствами клиентов. В состав таких средств входят SQL*Plus, предкомпиляторы Oracle и OCI (Oracle Call Interface – интерфейс вызовов Oracle). Кроме того, блоки могут формироваться приложениями третьих фирм, например программой SQL-Station. Обращаться к PL/SQL на сервере могут также приложения, созданные с помощью инструментальных средств, в состав которых входит система поддержки PL/SQL, например с помощью Oracle Forms.

## SQL*Plus

SQL*Plus дает возможность пользователям вводить SQL-операторы и блоки PL/SQL в диалоговом режиме в ответ на предлагаемую подсказку. Эти операторы направляются непосредственно базе данных, а результаты выводятся на экран. Благодаря тому что программа SQL*Plus работает в диалоговом режиме, она является, возможно, самым удобным способом управления PL/SQL на сервере. (За более детальной информацией о SQL*Plus, а также о командах, не рассмотренных в этом разделе, обращайтесь к руководству SQL*Plus User's Guide and Reference.)

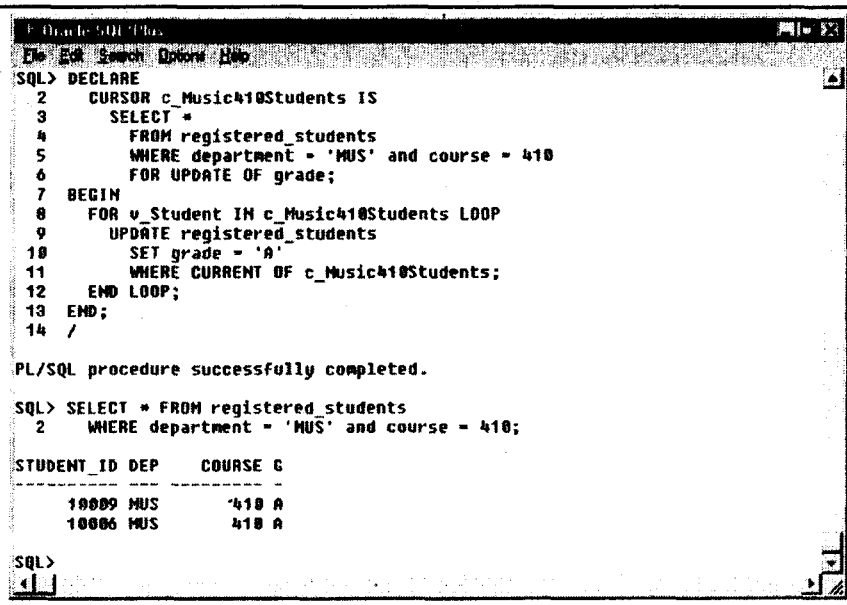
Команды SQL*Plus не учитывают регистра символов. Например, все эти команды используются для объявления переменных привязки:

```
SQL> VARIABLE v_Num NUMBER
SQL> variable v_Char char(3)
SQL> vaRIAbLe v_Varchar VarCHAR2(5)
```

### Управление блоками в SQL*Plus

При выполнении SQL-оператора в SQL*Plus нужно заканчивать оператор точкой с запятой. Точка с запятой не является элементом собственно оператора – это признак его окончания. Когда SQL*Plus считывает точку с запятой, программа узнает о завершении оператора и посылает его базе данных. В блоке же PL/SQL точка с запятой является синтаксическим элементом самого блока, а не признаком конца некоторого оператора. При вводе пользователем ключевого слова DECLARE или BEGIN SQL*Plus распознает это и понимает, что пользователь запускает на выполнение блок PL/SQL, а не SQL-оператор. Однако SQL*Plus должен знать еще и о том, когда блок завершится. Для этого применяется косая черта – сокращенное обозначение команды RUN SQL*Plus.

**Рис. 13.3.**
*PL/SQL в SQL*Plus*



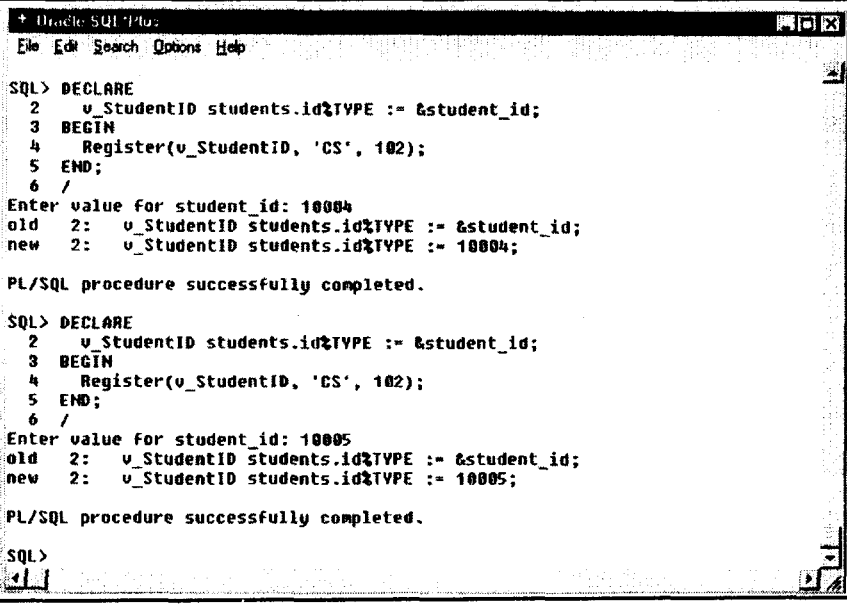

Обратите внимание на косую черту после блока PL/SQL, с помощью которого обновляется таблица **registered_students** (рис. 13.3). Для оператора SELECT, находящегося после блока, косая черта не требуется, поскольку после этого оператора указана точка с запятой.

## Переменные подстановки

В PL/SQL фактически не существует средств для ввода данных пользователями и вывода информации на экран. Программный модуль DBMS_OUTPUT обеспечивает ограниченный вывод информации в SQL*Plus (см. главу 14). В PL/SQL 2.3 предлагается также модуль UTL_FILE (см. главу 18), обеспечивающий обмен файлами с операционной системой. Однако в самой программе SQL*Plus имеется механизм ввода пользовательских данных, который реализуется через *переменные подстановки* (substitution variables). Подстановка переменных в тексте осуществляется в SQL*Plus перед передачей блока PL/SQL или SQL-оператора серверу и напоминает макросы языка C. Переменная подстановки обозначается символом амперсенда (&).

Использование переменных подстановки показано на рис. 13.4. Один и тот же блок выполняется дважды, каждый раз инициализируя переменную **v_StudentID** разным значением. Пользователь вводит значения 10004 и 10005, а в тексте блока они замещают переменную **&student.id**.

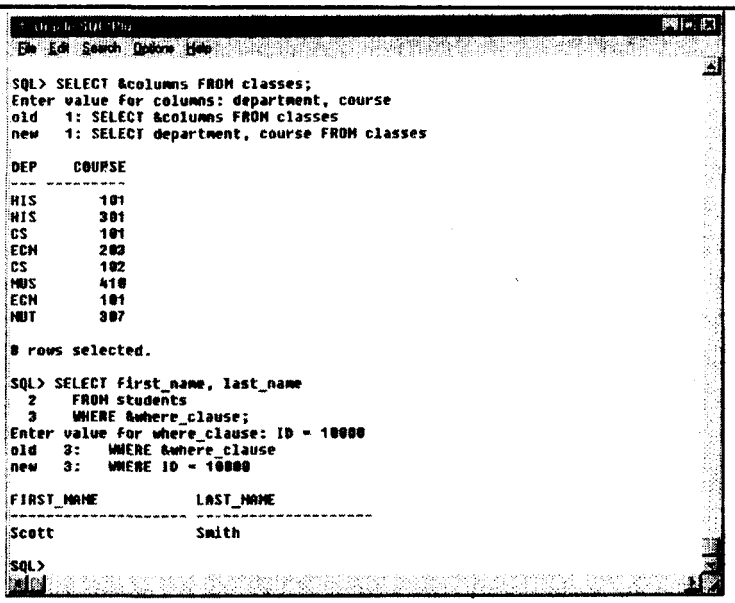
**Рис. 13.4.**
*Переменные подстановки SQL*Plus*



Для переменных подстановки память реально не выделяется. Перед передачей блока в базу данных для обработки SQL*Plus замещает переменную подстановки значением, вводимым пользователем. Поэтому переменные подстановки применяются только при вводе информации, а переменные привязки можно использовать как для ввода, так и для вывода.

Хотя переменные подстановки используются только при вводе данных, их можно указывать в любом месте SQL-оператора блока PL/SQL, как продемонстрировано на рис. 13.5. Переменные подстановки **&columns** и **&where_clause**, например, являются структурными элементами собственно операторов — именами столбцов и условием WHERE. Единственный способ выполнить ту же задачу в чистом PL/SQL — воспользоваться модулем DBMS_SQL (см. главу 15).

**Рис. 13.5.**
*Еще один пример использования переменных подстановки*




▼ СОВЕТУЕМ

Предположим, что в ответ на подсказку SQL> вводится следующий SQL-оператор:

```
SQL> SELECT *
       FROM students
       WHERE first_name = &first_name;
```

В этом случае, когда SQL*Plus предлагает ввести некоторое значение, нужно заключать это значение в одиночные кавычки, например 'SCOTT'. Далее введем такой оператор:

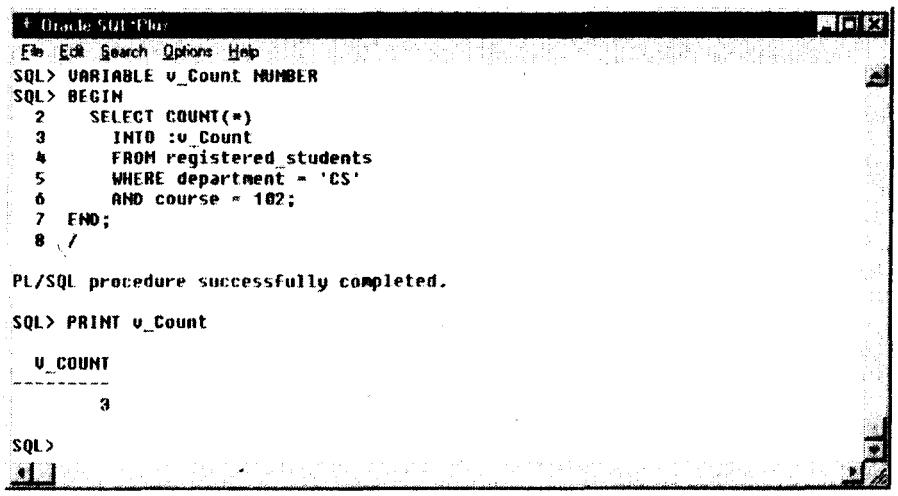
```
SQL> SELECT *
       FROM students
       WHERE first_name = '&first_name';
```

Теперь при вводе значения не нужно будет указывать кавычки, поскольку они уже являются частью оператора.

## Переменные привязки SQL*Plus

В SQL*Plus можно также выделять область памяти для хранения некоторой информации. Такая область используется внутри блоков PL/SQL и SQL-операторов, однако находится вне блоков, поэтому можно по очереди выделять ее разным блокам и после выполнения каждого из них выводить ее содержимое. Область памяти, выделяемая блоку, называется *переменной привязки* — bind variable (рис. 13.6). Переменная привязки (в данном случае **v_Count**) в SQL*Plus выделяется с помощью команды VARIABLE, которая действует только тогда, когда указана в ответ на подсказку SQL, но не внутри блока PL/SQL. Внутри него переменная привязки ограничивается двоеточием, а не знаком амперсенда, указываемым перед ней. После завершения блока команда PRINT выводит содержимое этой переменной. Для переменных привязки SQL*Plus разрешено задавать только следующие типы: VARCHAR2, CHAR и NUMBER (в SQL*Plus 3.2 и выше еще и REFCURSOR). Если для переменных привязки типа VARCHAR2 или CHAR длина не указана, то по умолчанию она принимается равной 1. Числовые (NUMBER) переменные привязки нельзя ограничивать точностью и масштабом.

**Рис. 13.6.**
*Переменные привязки SQL*Plus*




## Вызов хранимых процедур с помощью EXECUTE

Хранимые процедуры можно вызывать только из выполняемого раздела или из раздела исключительных ситуаций блока PL/SQL. В SQL*Plus предлагается удобный и более простой способ вызова хранимых процедур – команда EXECUTE. Она принимает заданные аргументы, размещает BEGIN перед ними и END; после них. Затем полученный блок передается в базу данных. Например, если ввести из подсказки SQL

```
SQL> EXECUTE Register(10006, 'CS', 102)
```

то блок PL/SQL

```
BEGIN Register(10006, 'CS', 102); END;
```

будет послан в базу данных. Точка с запятой после EXECUTE необязательна, что, кстати, справедливо для всех команд SQL*Plus: если точка с запятой указана, она игнорируется. Как и команды PRINT и VARIABLE, EXECUTE является командой SQL*Plus, следовательно, в блоке PL/SQL ее использовать нельзя.

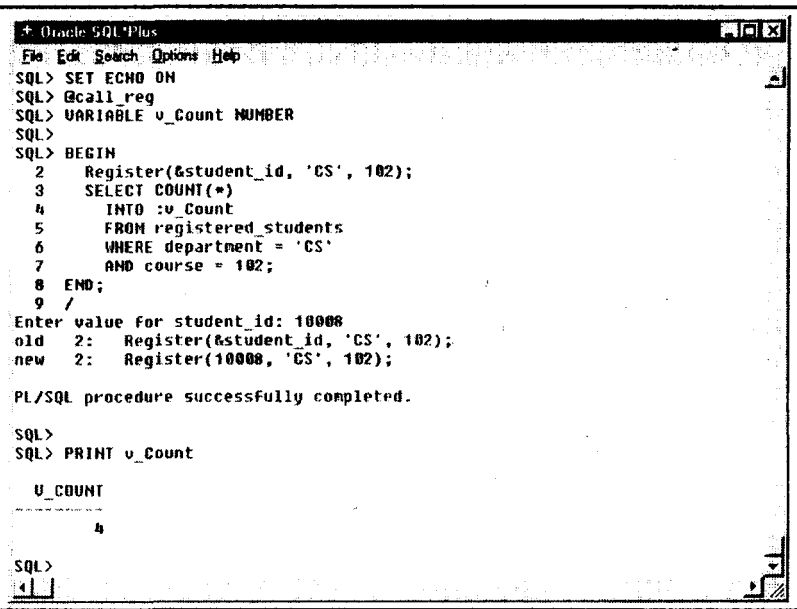
## Использование файлов

SQL*Plus дает возможность сохранить текущий блок PL/SQL или SQL-оператор в файле, который затем можно считать и выполнить. Эта возможность полезна как при разработке, так и при выполнении программ PL/SQL. Например, можно сохранить в файле команду CREATE OR REPLACE, а затем внести нужные изменения в процедуру или функцию, отредактировав этот файл. Для сохранения сделанных изменений в базе данных нужно всего лишь считать файл в SQL*Plus.

С помощью команды GET SQL*Plus считывает файл с диска в локальный буфер. Если указать символ косой черты, файл будет запущен на выполнение, как если бы он был введен непосредственно с клавиатуры. Если же символ косой черты уже содержится в файле, то такой файл можно считать и запустить, указав символ @. Предположим, что содержимое файла **call_reg.sql** выглядит следующим образом:

```
-- Этот пример содержится в файле call_reg.sql.
VARIABLE v_Count NUMBER

BEGIN
  Register(&student_id, 'CS', 102); SQL
  SELECT COUNT(*)
      INTO :v_Count
      FROM registered_students
      WHERE department = 'CS'
      AND course = 102;
  END;
/
PRINT v_Count
```

**Рис. 13.7.**
*Использование файла в SQL*Plus*




Теперь можно выполнить этот файл из подсказки SQL:

```
SQL> @call_reg
```

Полученный результат представлен на рис. 13.7. Команда SET ECHO ON сообщает SQL*Plus, что нужно вывести содержимое файла на экран.

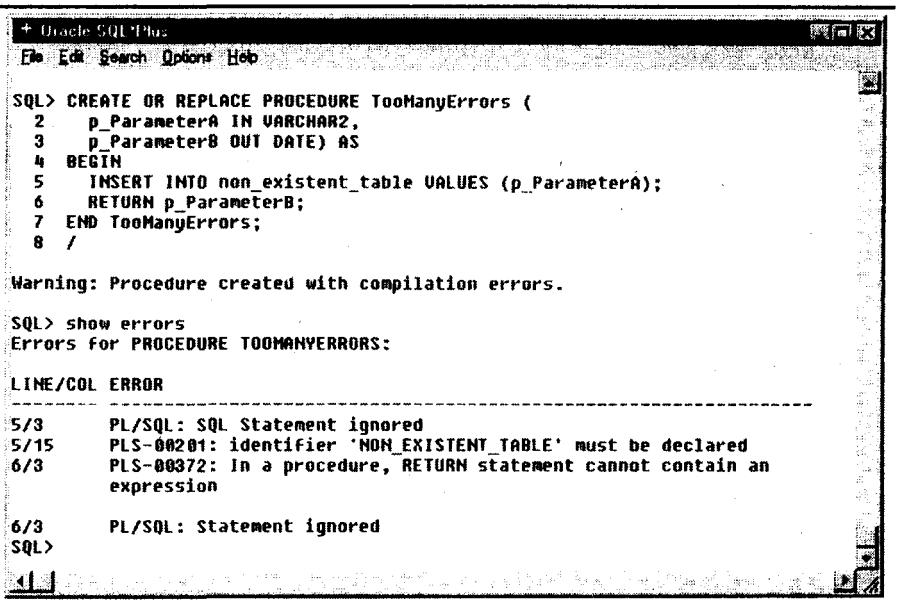
## Использование команды SHOW ERRORS

Как было сказано в главе 7, при создании хранимой подпрограммы информация о ней помещается в словарь данных. В частности, сведения об ошибках компиляции хранятся в представлении словаря данных **user_errors**. В SQL*Plus имеется команда SHOW ERRORS, которая обращается к этому представлению с запросами и сообщает об ошибках (рис. 13.8). Командой SHOW ERRORS можно воспользоваться после того, как SQL*Plus выдаст следующее сообщение:

```
Warning: Procedure created with compilation errors.
```

(Предупреждение! Процедура создана с ошибками компиляции. — *Прим. пер.*)

**Рис. 13.8.**
*Использование SHOW ERRORS*



# Предкомпиляторы Oracle

Предкомпиляторы Oracle используются для включения SQL-операторов и блоков PL/SQL в программы, созданные на других языках программирования третьего поколения: C, C++, COBOL, Pascal, FORTRAN, PL/1 и Ada. Предкомпиляторы называются соответственно Pro*C, Pro*Cobol, Pro*Pascal и т.д. Такое использование PL/SQL называют *встроенным PL/SQL* (enbeddedPL/SQL), а язык, в который встраивается PL/SQL, — *базовым языком* (host language). Предкомпилятор транслирует встроенные SQL- и PL/SQL-операторы в вызовы библиотеки поддержки (run-time library) предкомпилятора. Затем результаты его работы необходимо скомпилировать и скомпоновать с этой библиотекой, чтобы получить в итоге исполняемую программу. При использовании предкомпиляторов, как и при работе с SQL*Plus, на станции клиента система поддержки PL/SQL не устанавливается — все SQL-операторы и блоки PL/SQL посылаются для выполнения на сервер. На стороне клиента размещается только сама программа.

Взаимодействие между программой и базой данных осуществляется с помощью *базовых переменных* (host variables), которые создаются в соответствии с правилами базового языка, но располагаются в специальном разделе объявлений (declare section) программы. Этот раздел ограничивается в исходном тексте программы следующим образом:

```
EXEC SQL BEGIN DECLARE SECTION;
```

и

```
EXEC SQL END DECLARE SECTION;
```

▼ ВНИМАНИЕ **В Pro*C/C++ версии 2.0 и выше раздел объявлений больше не нужен. Во встроенном SQL-операторе или блоке PL/SQL можно использовать любую базовую переменную независимо от того, объявлена она или нет в разделе объявлений.**

Между встроенным PL/SQL и диалоговым PL/SQL в SQL*Plus существует ряд различий. Они касаются использования переменных привязки, признаков конца операторов, а также требований, предъявляемых предкомпиляторами. (За более детальной информацией о предкомпиляторах Oracle обращайтесь к руководствам Programmer's Guide to the Pro*C/C++ Precompiler или Programmer's Guide to the Oracle Precompilers.)

## Переменные привязки в предкомпиляторах

Переменные, описываемые в разделе объявлений, можно использовать во встроенных блоках PL/SQL и во встроенных SQL-операторах. Внутри встроенного оператора переменная привязки ограничивается начальным двоеточием. Например, в следующем фрагменте программы Pro*C вызывается хранимая процедура **Register.** Для тех читателей, кто, возможно, не знаком с языком C, текст снабжен более подробными комментариями, чем обычно.

```
-- Этот пример содержится в файле call_reg.pc.
EXEC SQL BEGIN DECLARE SECTION;
  /* Объявим переменные C . */
  VARCHAR v_Department[4];    /* Псевдотип VARCHAR доступен только
                                 в Pro*C и преобразуется в тип записи
                                 с двумя полями — .arr and .len */
  int v_Course;               /* v_Course — целое число. */
  int v_StudentID;            /* v_StudentID — также целое число. */
EXEC SQL END DECLARE SECTION;

  /* Инициализируем базовые переменные. Им лишь присваиваются
     некоторые значения, однако пользователь может считывать эти
     переменные из файла, обращаться к ним при вводе данных и т. д.
     Что касается переменной типа VARCHAR, то ее строка символов
     копируется в поле .arr, а длина строки (в данном случае 3)
     присваивается полю .len. */
  strcpy(v_Department.arr, "ECN");
  v_Department.len = 3;
  v_Course = 101;
  v_StudentID = 10006;
```

```
    /* Начнем встроенный блок PL/SQL. Обратите внимание на
       использование EXEC SQL EXECUTE и END-EXEC - ключевых слов,
       ограничивающих блок для предкомпилятора. */
    EXEC SQL EXECUTE
        BEGIN
          Register(:v_Department, :v_Course, :v_StudentID);
    END;
END-EXEC;
```

Внутри встроенного блока базовые переменные **v_Department**, **v_Course** и **v_StudentID** предварены двоеточиями. Если этого не сделать, программа не будет прекомпилироваться и на экран будет выдано сообщение об ошибке:

```
PLS-201: identifier must be declared
```

(идентификатор должен быть описан. — *Прим. пер.*)

## Встраивание блока в программу

Обратите внимание, как в рассмотренном фрагменте программы ограничивается сам блок PL/SQL: он начинается ключевыми словами

```
EXEC SQL EXECUTE
```

а заканчивается

```
END-EXEC;
```

Точка с запятой после END-EXEC обязательна. Между этими ключевыми словами располагается весь блок PL/SQL, в том числе и конечная точка с запятой после оператора END. Во встроенном блоке могут также находиться собственный раздел объявлений и раздел исключительных ситуаций.

## Переменные-индикаторы

В переменных PL/SQL, как и в столбцах базы данных, могут содержаться или конкретные, или NULL-значения. Однако в языках третьего поколения, например в C, неизвестно, что такое NULL-значения. Для строк символов они имитируются пустыми строками, однако, скажем, NULL-значение целого типа описать нельзя. Для устранения возможных проблем используется *переменная-индикатор* (indicator variable). Это просто 2-байтовое целое число, добавляемое к ссылке на базовую переменную во встроенном блоке или в SQL-операторе. Ниже приведен встроенный блок, в котором информация считывается из таблицы **registered_students**. В столбце grade этой таблицы могут содержаться NULL-значения, поэтому для их обнаружения необходима переменная-индикатор.

```
-- Этот пример является частью файла indicator.pc.
EXEC SQL BEGIN DECLARE SECTION;
  char v_Grade;    /*  v_Grade - это один символ. */
  short i_Grade;   /*  Заметьте, что индикатор объявлен с типом
                       short, т.е. 2-байтовое целое число. */
EXEC SQL END DECLARE SECTION;
EXEC SQL EXECUTE
  BEGIN
      SELECT grade
        INTO :v_Grade INDICATOR :i_Grade
        FROM registered_students
        WHERE student_id = 10006
        AND department = 'ECN'
        AND course = 101;
  END;
END-EXEC;
if (i_Grade != 0)
  printf("No grade recorded for this student\n");
else
  printf("The grade recorded is % c\n", v_Grade);
```

Между базовой переменной и переменной-индикатором можно указать ключевое слово INDICATOR, как сделано в предыдущем примере. Это слово необязательно, поэтому блок можно переписать следующим образом:

```
-- Этот пример является частью файла indicator.pc.
EXEC SQL EXECUTE
  BEGIN
      SELECT grade
         INTO :v_Grade:i_Grade
         FROM registered_students
         WHERE student_id = 10006
         AND department = 'ECN'
         AND course = 101;
END;
```

В этом примере переменная v_Grade является *выходной переменной* (output variable), значение которой присваивается в блоке. Значения, принимаемые индикатором для выходных переменных, описаны в таблице 13.1.

**ТАБЛИЦА 13.1.**

| Значение переменной-индикатора | Смысл |
|---|---|
| 0 | В базовую переменную успешно считано некоторое значение. |
| -1 | Базовой переменной присвоено NULL-значение. |
| >0 | Базовая переменная слишком мала для хранения возвращаемого значения, поэтому оно усечено. В переменной-индикаторе содержится первоначальная длина результата. Это значение применимо только для символьных базовых переменных. |
| -2 | Базовая переменная слишком мала для хранения возвращаемого значения, поэтому оно усечено. Однако первоначальная длина результата слишком велика и не умещается в 2-х байтах. Это значение применимо только для символьных базовых переменных. |

*Входные переменные* (input variable) считываются из встроенных блоков. Для них также можно использовать переменные-индикаторы, значения которых аналогичны значениям индикаторов, применяемых для выходных переменных (что показано в таблице 13.2.).

**ТАБЛИЦА 13.2.**

| Значение переменной-индикатора | Смысл |
|---|---|
| 0 | Следует использовать соответствующую базовую переменную. |
| -1 | Следует использовать NULL-значение. |

## Обработка ошибок

Обработка ошибок в программе Pro*C осуществляется с помощью либо структуры sqlca, либо переменных состояния SQLCODE и/или SQLSTAT. После выполнения каждого встроенного SQL-оператора в переменных состояния будет содержаться или код ошибки, или ноль, если оператор выполнен успешно. Сказанное относится и к встроенным блокам PL/SQL. Если в блоке установлена и не обработана исключительная ситуация, то в переменную состояния записывается код ошибки. Если все установленные в блоке исключительные ситуации обработаны, он завершается успешно и в переменную состояния заносится ноль. (Правила передачи исключительных ситуаций см. в главе 10.)

Например, в следующем фрагменте программы содержится встроенный блок, в котором вызывается процедура **RecordFullClasses** (см. главу 7). После выполнения блока программа проверяет переменную состояния **sqlca.sqlcode**, чтобы убедиться в его успешном выполнении. Если блок выполнен неуспешно, выводится сообщение об ошибке.

```
-- Этот пример содержится в файле error.pc.
EXEC SQL INCLUDE SQLCA; /* В этом операторе находится структура
                           SQLCA, поля которой используются
                           для обработки ошибок. */
EXEC SQL EXECUTE
  BEGIN
      RecordFullClasses;
  END;
END-EXEC;
/* В sqlca.sqlcode будет содержаться ноль, если оператор выполнен
    успешно, и код ошибки, если оператор выполнен с ошибкой. В случае
    ошибки в sqlca.sqlerrm.sqlerrmc будет содержаться текст сообщения
    об ошибке, а в sqlca.sqlerrm.sqlerrml — длина этого сообщения. */
if (sqlca.sqlcode != 0) {
  printf("Error during execution of RecordFullClasses.\n");
  printf("%.70s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  }
else
  printf("Execution successful.\n");
```

## Параметры, необходимые для работы предкомпилятора

Для прекомпиляции программы с встроенным блоком PL/SQL параметр предкомпилятора SQLCHECK следует установить как SEMANTICS. Если SQLCHECK=SEMANTICS, предкомпилятор будет пытаться соединиться с базой данных во время прекомпиляции для проверки синтаксиса и семантики тех объектов базы данных, на которые ссылается программа. Для этого предкомпилятору требуется знать имя и пароль пользователя, которые можно указать с помощью параметра предкомпилятора USERID. Во время прекомпиляции нужно указывать те же имя и пароль, которые используются при выполнении программы. Например, если программа соединяется с базой данных как пользователь Oracle с именем example, параметры предкомпилятора должны быть установлены следующим образом:

```
SQLCHECK=SEMANTICS USERID=example/example
```

Для параметра USERID можно указывать как имя с паролем (см. предыдущий пример), так и одно имя пользователя. Если пароль не указан, предкомпилятор попросит пользователя ввести его.

Если параметр USERID не установлен, необходимо описать структуру таблиц, на которые производятся ссылки, с помощью встроенных операторов DECLARE TABLE. Однако оператора DECLARE PROCEDURE не существует, поэтому, если в программе вызываются хранимые процедуры, параметры USERID и SQLCHECK=SEMANTICS обязательны.

**▼ ВНИМАНИЕ** **Если параметр USERID не указан и если для таблиц, на которые ссылается программа, не заданы операторы DECLARE TABLE, предкомпилятор возвращает сообщение об ошибке:**

```
PLS-201: identifier must be declared
```

**(идентификатор должен быть описан. — *Прим. пер.*)**

**которое может привести пользователя в замешательство. Таблица создана, и пользователю предоставлены полномочия на доступ к ней, однако данное сообщение говорит об ином. Эта ошибка является результатом того, что пропущен параметр USERID, а не объявление таблицы. Чтобы избежать путаницы, придерживайтесь правила: когда SQLCHECK=SEMANTICS, USERID всегда должен быть указан.**

# OCI

Интерфейс вызовов Oracle (OCI — Oracle Call Interface) предлагает еще один способ обращения к базе данных из программы, написанной на одном из языков третьего поколения. В этом случае вместо встраивания блоков PL/SQL и SQL-операторов вызываются функции, описанные в библиотеке OCI. В ней содержатся функции для синтаксического анализа SQL-операторов, привязки входных переменных, для описания выходных переменных, выполнения операторов и считывания результатов. Исходная программа целиком пишется на одном из языков третьего поколения, и предкомпилятор здесь не нужен. (Более подробно об OCI рассказано в руководстве Programmer's Guide to the Oracle Call Interface).

Использование PL/SQL с OCI достаточно просто. Вместо синтаксического анализа отдельного SQL-оператора выполняется синтаксический анализ анонимного блока PL/SQL, как показано в следующем фрагменте программы:

```
char *plsql_block =
  "BEGIN \
     Register (:v_StudentID, :v_Department, :v_Course); \
  END;";
int return_val;
Cda_Def cda;

return_val = oparse(&cda, plsql_block, -1, 1, 2);
```

▼ ВНИМАНИЕ В примерах этого раздела используется OCI, определенный для Oracle7. Интерфейс для Oracle8 имеет ряд дополнительных свойств: манипулирование объектами и клиентским объектным кэшем, а также возможность формирования SQL-операторов. Тем не менее в библиотеку OCI Oracle8 включены все вызовы OCI Oracle7, которые можно применять и при работе с базой данных Oracle8.

## Рекомендации по использованию блоков PL/SQL в OCI

Вызов oparse принимает последовательность символов, содержащую SQL-оператор, который необходимо выполнить. Для выполнения блока PL/SQL в такую последовательность следует включить весь блок, в том числе конечную точку с запятой (как показано в предыдущем примере).

▼ ОСТОРОЖНО Соблюдайте осторожность при сопровождении блока комментариями. В последовательности, передаваемой вызову **oparse**, символы перехода на новую строку не учитываются, поэтому комментарии типа --... будут превращать в комментарий всю оставшуюся часть блока, а не только символы, находящиеся между --... и символом конца строки. Для гарантии корректного выполнения программ используйте комментарии типа /* ... */, применяемые также в C.

## Структура вызовов OCI

Блоки PL/SQL выполняются так же, как и операторы DML. Особенно важно учесть, что считывать данные из блока PL/SQL нельзя. Далее приведен алгоритм выполнения блока PL/SQL:

1. Синтаксический анализ с помощью **oparse**.
2. Привязка всех областей хранения данных с помощью **obndrv** или **obndra**.
3. Выполнение блока при помощи **oexec**.

Для блока PL/SQL запрещается использовать **odefin** или **ofetch**; эти вызовы разрешены только для операторов SELECT. Кроме того, все области хранения данных должны быть привязаны по именам при помощи **obndrv** или **obndra** – **obndrn** использовать нельзя.

Ниже приведен пример программы, работающей в OCI и вызывающей процедуру **Register**. Эта программа написана для системы Unix и поэтому может не компилироваться на другой платформе.

```
-- Этот пример содержится в файле oci.c.
/* Добавим стандартные файлы заголовков, а также заголовки OCI. */
#include <stdio.h>
#include <oratypes.h>
#include <ocidfn.h>
#include <ociapr.h>

/* Объявим LDA, HDA и CDA для последующего использования в операторах. */
Lda_Def lda;
ub1 HDA[512];
Cda_Def cda;

/* Объявим переменные, которые будут использоваться при вводе. */
char v_Department[4] = "ECN";
```

```
int v_Course = 101;
int v_StudentID = 10006;

/* Строка символов, которая содержит блок, вызывающий Register.
   Обратите внимание: символы возврата заменены обратными косыми
   чертами, чтобы весь текст содержался в одной строке С.
   В строке символов указана конечная точка с запятой, поскольку
   она является синтаксическим элементом блока. */
char *plsqlBlock =
  "BEGIN
     Register(:v_StudentID, :v_Department, :v_Course);
  END;";

/* Имя и пароль пользователя для соединения с базой данных. */
char *username = "example";
char *password = "example";

/* Функция сообщения об ошибках. С помощью oerhms получает
   полную информацию об ошибках и выводит ее на экран. */
void print_error(Lda_Def *lda, Cda_Def *cda) {
  int v_ReturnChars;
  char v_Buffer[1000];

  v_ReturnChars = oerhms(lda, cda->rc, (text *) v_Buffer,
                         (sword) sizeof(v_Buffer));
  printf("Oracle error occurred!\n");
  printf("%s\n", v_Buffer);
}

main() {
  /* Соединимся с базой данных. */
  if (orlon(&lda, HDA, (text *) username, -1,
            (text *) password, -1, 0)) {
     print_error(&lda, &lda);
     exit(-1);
  }
  printf("Connected to Oracle\n");

  /* Откроем курсор для последующего использования. */
  if (oopen(&cda, &lda, (text *) 0, -1, -1,
           (text *) 0, -1)) {
     print_error(&lda, &cda);
     exit(-1);
  }

  /* Выполним синтаксический анализ блока PL/SQL. */
  if (oparse(&cda, (text *) plsqlBlock,
            (sb4) -1, 1, (ub4) 2)) {
     print_error(&lda, &cda);
     exit(-1);
  }

  /* Выполним привязку переменной v_Department, используя тип 5, STRING. */
  if (obndrv(&cda, (text *) ":v_Department", -1,
```

```
                (ub1 *) v_Department, sizeof(v_Department),
                5, -1, (sb2 *) 0, 0, -1, -1)) {
            print_error(&lda, &cda);
            exit(-1);
        }

        /* Выполним привязку переменной v_Course, используя тип 3, INTEGER. */
        if (obndrv(&cda, (text *) ":v_Course", -1,
                (ub1 *) &v_Course, sizeof(v_Course),
                3, -1, (sb2 *) 0, 0, -1, -1)) {
            print_error(&lda, &cda);
            exit(-1);
        }

        /* Выполним привязку переменной v_StudentID, используя тип 3, INTEGER. */
        if (obndrv(&cda, (text *) ":v_StudentID", -1,
                (ub1 *) &v_StudentID, sizeof(v_StudentID),
                3, -1, (sb2 *) 0, 0, -1, -1)) {
            print_error(&lda, &cda);
            exit(-1);
        }

        /* Выполним оператор. */
        if (oexec(&cda)) {
            print_error(&lda, &cda);
            exit(-1);
        }

        /* Завершим работу. */
        if (ocom(&lda)) {
            print_error(&lda, &cda);
            exit(-1);
        }

        /* Закроем курсор. */
        if (oclose(&cda)) {
            print_error(&lda, &cda);
            exit(-1);
        }

        /* Отключимся от базы данных. */
        if (ologof(&lda)) {
            print_error(&lda, &cda);
            exit(-1);
        }
    }
```

## SQL-Station

Формировать PL/SQL-операторы и выполнять их в базе данных можно не только с помощью программных продуктов корпорации Oracle, таких как пакет Developer 2000 или предкомпиляторы, но и с помощью программных средств третьих фирм. Подобно предкомпиляторам, эти средства не имеют в своем составе локальных систем поддержки PL/SQL, поэтому вся обработка информации должна выполняться на сервере.

Одним из таких средств является программный пакет SQL-Station корпорации Platinum. SQL-Station состоит из трех компонентов: SQL-Station Coder (кодировщик), SQL-Station Debugger (отладчик) и

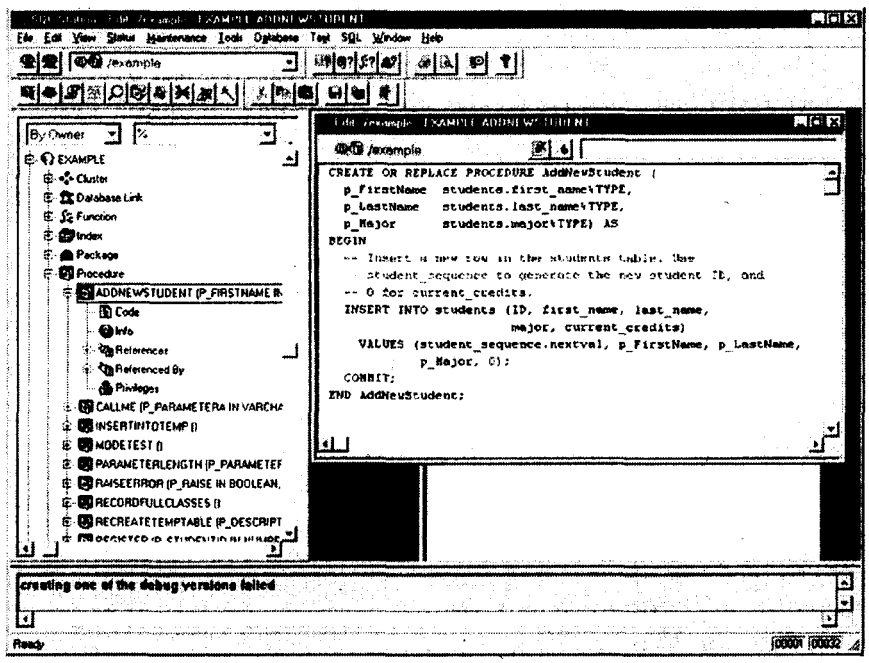
SQL-Station Plan Analyzer (анализатор планов). Кодировщик используется для создания объектов PL/SQL и выполнения SQL-сценариев. Отладчик дает возможность выполнить программу PL/SQL на сервере в пошаговом режиме и проанализировать содержимое переменных во время ее выполнения. Анализатор планов позволяет просмотреть и настроить план выполнения оператора. SQL-Station Coder рассматривается в этом разделе, SQL-Station Debugger – в главе 14, а SQL-Station Plan Analyzer – в главе 22.

▼ ВНИМАНИЕ Демонстрационный вариант SQL-Station содержится на прилагаемом компакт-диске, в каталоге station, и там же описан процесс инсталляции этого продукта. (Более подробную информацию об использовании и приобретении SQL-Station можно найти на web-узле корпорации Platinum *www.platinum.com.*

## Среда функционирования кодировщика

Среда функционирования кодировщика SQL-Station показана на рис. 13.9. Рабочий экран разделяется на несколько окон. Слева расположено окно просмотра каталогов, в котором можно видеть все объекты базы данных, упорядоченные по типам. На этом рисунке развернута процедура **AddNewStudent.** Для получения различных сведений об этом объекте просто дважды щелкните мышью на его имени.

**Рис. 13.9.**
*Кодировщик SQL-Station Coder*



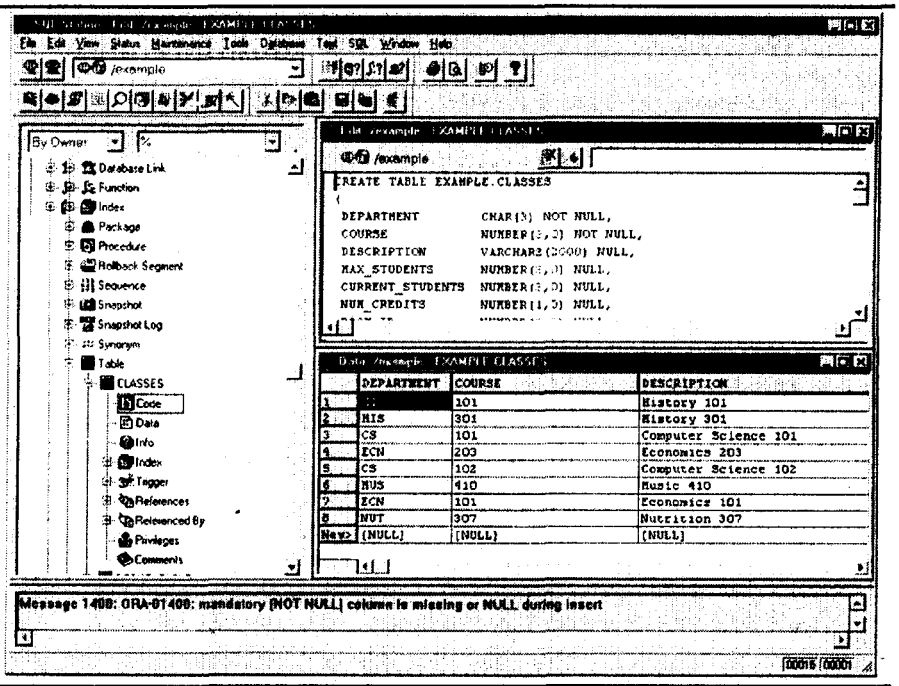

Справа от окна просмотра каталогов на рабочем экране имеется область редактирования. На рис. 13.9 показаны два окна редактирования – одно открыто для текста процедуры **AddNewStudent**, а другое в данный момент пусто. Внизу экрана находится окно сообщений, а вверху – панель инструментов.

**Окно просмотра каталогов** Окно просмотра каталогов используется для просмотра объектов базы данных. Просмотреть и модифицировать программный текст объекта PL/SQL можно в окне редактирования. Описание и данные таблиц можно просматривать в отдельных окнах редактирования. Для этого нужно дважды щелкнуть мышью на соответствующих указателях в окне просмотра. На рис. 13.10 продемонстрированы два окна редактирования, открытых для таблицы **classes**.

**Окна редактирования** Окна редактирования используются в различных целях. Например, на рис. 13.10 в одном окне находится текст, написанный на SQL, а в другом – данные, содержащиеся в таблице. Окна редактирования располагаются с правой стороны рабочего экрана; при желании можно размещать их в виде мозаики либо перекрывая друг друга.

**Окно сообщений** В окне сообщений, расположенном внизу экрана, отображаются сообщения об ошибках SQL-Station и/или Oracle.

**Рис. 13.10.**
*Просмотр таблицы classes*




## Вызов хранимой процедуры

В окне Procedure Execution (выполнение процедур) можно вызвать хранимую процедуру и ввести ее параметры в диалоговом режиме. Это окно открывается в меню Tools (инструменты) или щелчком правой клавиши мыши на нужном объекте в окне просмотра каталогов. Окно обработки процедур представляет собой заполненную экранную форму, в которой можно указать значения каждого параметра процедуры. Окно выполнения процедур для AddNewStudent с введенными параметрами показано на рис. 13.11. После ввода параметров можно выполнить процедуру, щелкнув мышью на кнопке Execute в панели инструментов или выбрав Execute в меню File. Результаты выполнения данной процедуры приведены на рис. 13.12. Обратите внимание на сообщение о процессе выполнения – здесь сказано, что процедура была выполнена успешно.

**Рис. 13.11.**
*Вызов AddNewStudent*

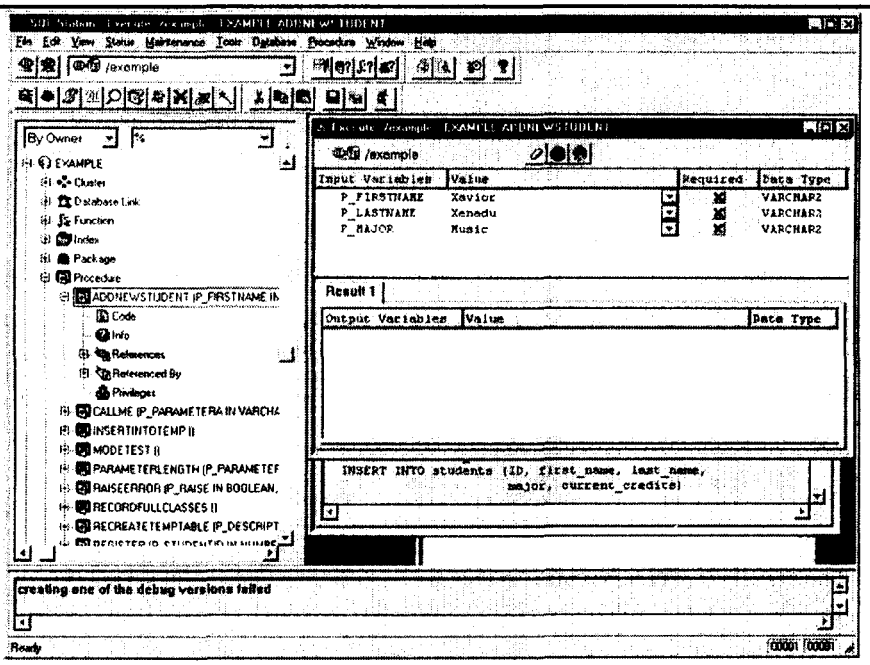
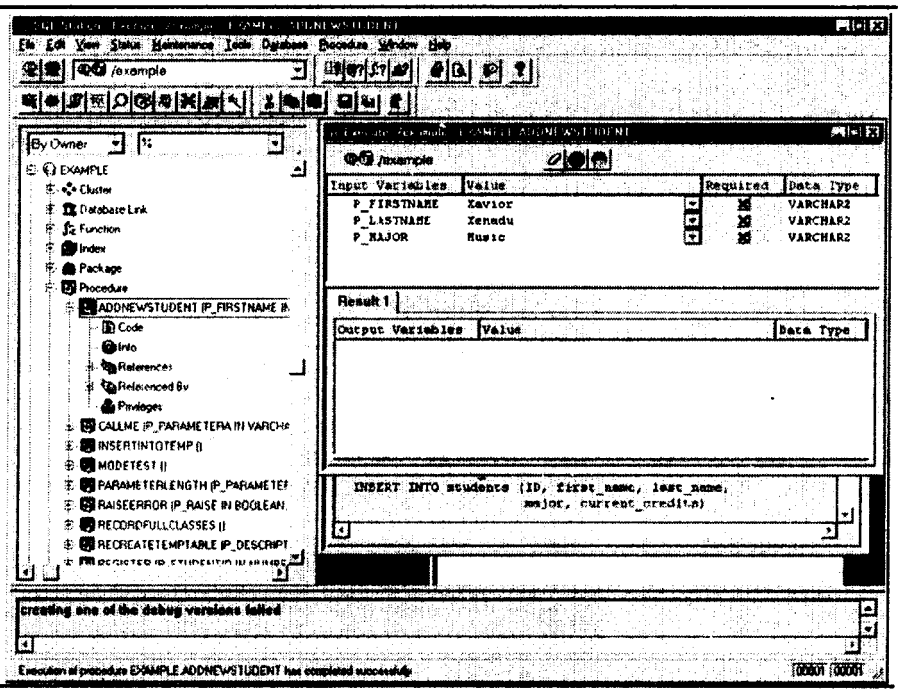


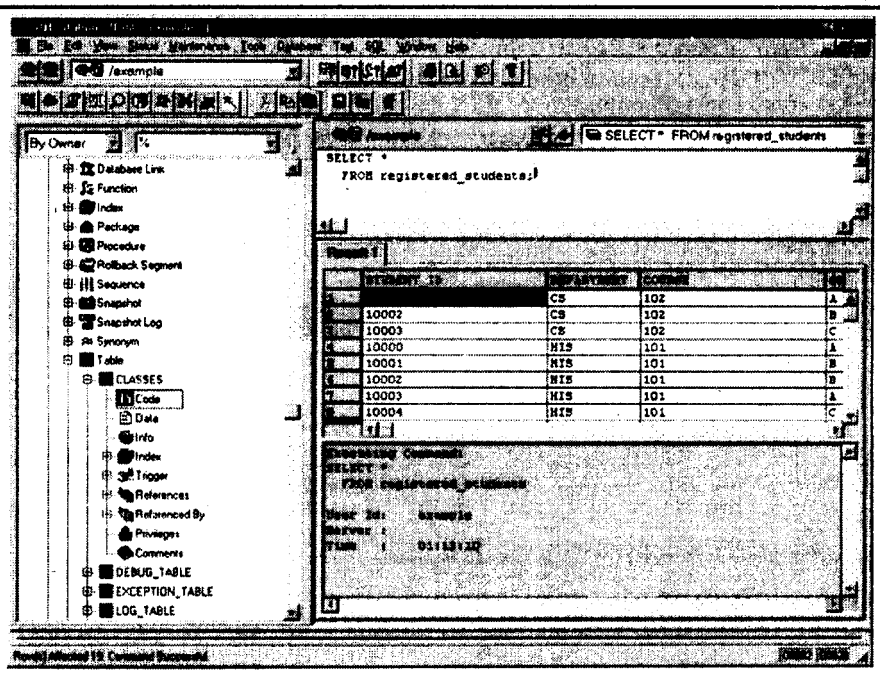
**Рис. 13.12.**
*Результаты вызова AddNewStudent*




## Выполнение SQL-сценария

Для выполнения SQL-сценария нужно просто считать его в окно редактирования и щелкнуть мышью на кнопке Execute. Сценарий будет запущен на выполнение, и результаты отобразятся в отдельном окне. На рис. 13.13 в окне редактирования (максимально увеличенном) показаны результаты работы оператора SELECT.

**Рис. 13.13.**
*Выполнение SQL-оператора*



# PL/SQL на станции клиента

В число средств Oracle, содержащих систему поддержки PL/SQL на станции клиента, входят программные пакеты Developer 2000, Designer 2000 и Discoverer 2000. Каждый из этих пакетов включает несколько программных средств разработки программ и выдачи запросов, объединенных в одно целое. Для каждого пакета на станции клиента устанавливается локальная система поддержки PL/SQL, используемая для управления обработкой приложений (рис. 13.2).

Если объект PL/SQL создается с помощью клиентского PL/SQL, обращение к нему возможно только из того объекта, который создал данный объект. Например, процедура, созданная в Oracle Forms, доступна только единственной форме и недоступна другим процедурам базы данных или другим формам. В таблице 13.3 сравниваются хранимые и клиентские процедуры, а в таблице 13.4 — триггеры базы данных и клиентские триггеры. (О хранимых процедурах и триггерах базы данных см. главы 7 и 9.)

**ТАБЛИЦА 13.3. Хранимые и клиентские процедуры**

| Хранимые процедуры | Клиентские процедуры |
|---|---|
| Хранятся в словаре данных как объекты базы данных | Хранятся как часть приложения клиента; не являются объектами базы данных |
| Могут быть выполнены любым пользователем, установившим соединение с базой данных и имеющим привилегию EXECUTE для процедур | Могут быть выполнены только создавшими их приложениями |
| Могут вызывать другие хранимые процедуры | Могут вызвать клиентские процедуры в том же приложении и хранимые процедуры в базе данных |
| Могут работать с данными, хранимыми в таблицах Oracle, с помощью SQL-операторов | Могут работать с данными, хранимыми в таблицах Oracle, а также с переменными в приложениях |

**ТАБЛИЦА 13.4. Триггеры базы данных и клиентские триггеры**

| Триггеры базы данных | Клиентские триггеры |
|---|---|
| Активизируются, когда над базой данных выполняется оператор DML | Активизируются, когда пользователь нажимает какую-либо клавишу или переходит на экране от одного поля к другому |
| Могут быть строковыми или операторными | Между строковыми и операторными триггерами различия нет |
| Могут работать с данными, хранимыми в таблицах Oracle, с помощью SQL-операторов | Могут работать с данными, хранимыми в таблицах Oracle, а также с переменными в формах |
| Могут активизироваться любым соединением, которое выполняет оператор DML, вызывающий активизацию триггера | Могут активизироваться только из той формы, которая их создает |
| Могут вызывать активизацию других триггеров базы данных | Могут вызывать активизацию триггеров базы данных, но не триггеров другой формы |

## Назначение клиентской системы

Большая часть функций приложений, разрабатываемых с помощью средств Developer 2000, Designer 2000 и Discoverer 2000, весьма удобна для работы с PL/SQL. Особенно интенсивно конструкции PL/SQL используются в Oracle Forms. Например, каждый триггер формы представляет собой блок PL/SQL, а поля формы трактуются как переменные привязки. На первый взгляд кажется, что размещение системы поддержки PL/SQL на станции клиента сводит на нет все преимущества PL/SQL. В конце концов одним из назначений языка программирования является снижение сетевого трафика, (см. главу 1). Когда система поддержки PL/SQL размещается на станции клиента, SQL-операторы передаются по сети, а процедурные операторы обрабатываются клиентом. Может показаться, что при этом нагрузка на сеть увеличивается и замедляет выполнение приложений.

Однако большая часть работы, выполняемой приложениями Oracle Forms, по природе своей процедурна. Когда пользователь переходит от формы к форме или от блока к блоку либо нажимает клавишу, — автоматически активизируются триггеры и в поля формы заносятся другие значения. Все эти операции являются процедурными и поэтому выполняются на станции клиента, что повышает производительность системы.

По отношению к PL/SQL все вышеперечисленные программные пакеты ведут себя точно так же. В этой главе будут рассмотрены средства Oracle Forms и Procedure Builder.

# Oracle Forms

Forms Designer (проектировщик форм) является средой GUI (графического пользовательского интерфейса) и поэтому предоставляет в распоряжение пользователей ряд различных окон. К PL/SQL имеют отношение окна Object Navigator (навигация объектов) и PL/SQL Editor (редактор PL/SQL). Более подробно о средстве Oracle Forms и о его применении рассказано в руководстве Oracle Forms User's Guide.

## PL/SQL Editor

PL/SQL Editor используется для модификации процедур и функций как на станции клиента, так и на сервере. В зависимости от местонахождения модифицируемого объекта окно PL/SQL Editor выглядит по-разному. На рис. 13.14 показано это окно для процедуры **Register**, хранимой на сервере. Кнопка Save (сохранить) редактора аналогична команде CREATE OR REPLACE PROCEDURE: при нажатии этой кнопки процедура создается в словаре данных. Кнопка Drop (удалить) эквивалентна команде DROP PROCEDURE: при ее нажатии процедура удаляется из словаря данных.

**Рис. 13.14.**
*Окно PL/SQL Editor для хранимой процедуры*

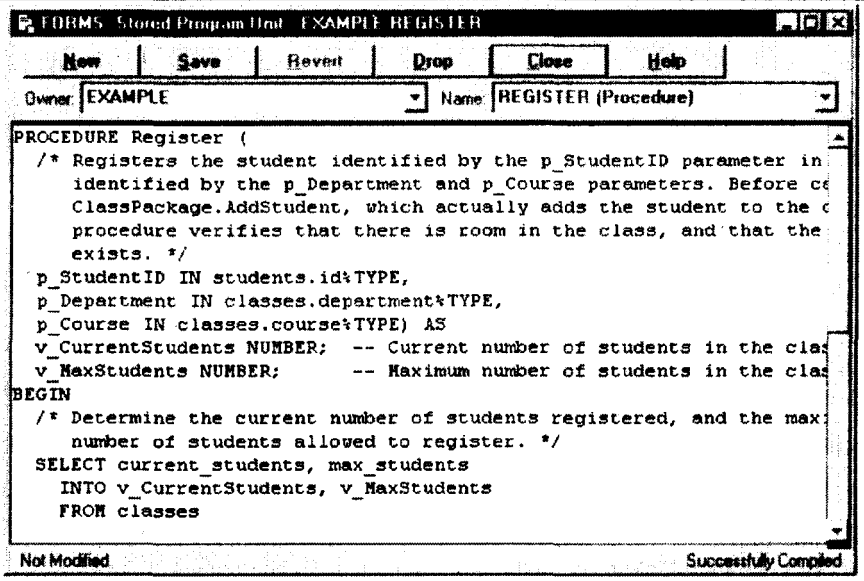
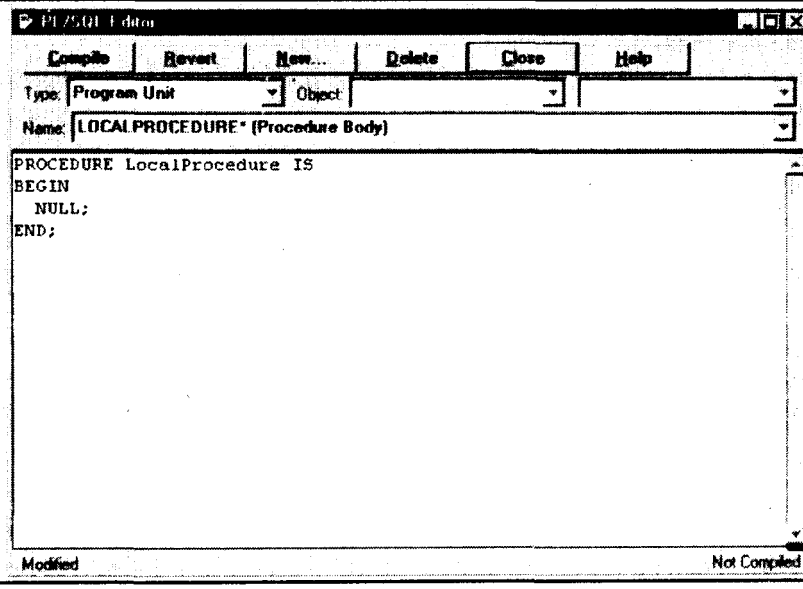



**Рис. 13.15.**
*Окно PL/SQL Editor для локальной процедуры*

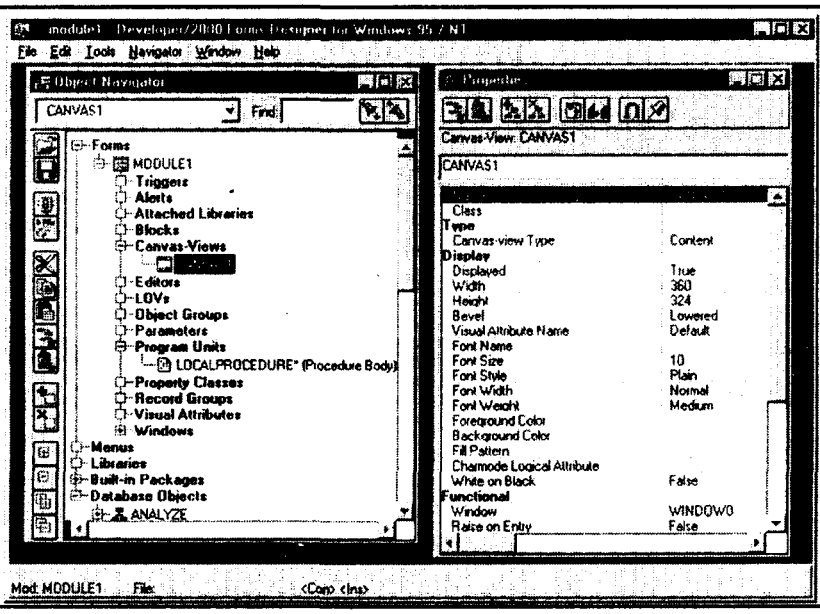


Теперь сравним рис. 13.14 и 13.15. На рис. 13.15 изображено окно PL/SQL Editor с локальной процедурой. Кнопка Save заменена на Compile (компилировать), с помощью которой процедура будет сохраняться не на сервере, а в системе PL/SQL на станции клиента. При этом словарь данных в базе данных не модифицируется. Теперь процедура удаляется с помощью кнопки Delete, а не Drop. Кнопка New позволяет создавать новые процедуры, функции и триггеры формы.

## Object Navigator

Окно Object Navigator используется для работы со всеми объектами формы. Щелкнув мышью на значке любого объекта, затем можно отредактировать его характеристики. Объекты можно перемещать на экране, например передавать процедуры со станции клиента на сервер, и наоборот. На рис. 13.16 изображено окно Object Navigator со списком Property (свойства) для некоторого объекта-основы.

**Рис. 13.16.**
*Object Navigator*




# Procedure Builder

Procedure Builder (построитель процедур) является частью набора инструментальных средств Developer 2000, в состав которого входят также средства Oracle Forms, Oracle Reports и Oracle Graphics. Procedure Builder расширяет возможности Oracle Forms по работе с объектами, добавляя средство PL/SQL Interpreter (интерпретатор PL/SQL). Окно интерпретатора является основным элементом Procedure Builder, так как в нем можно выполнять команды клиентского PL/SQL в пошаговом режиме, строка за строкой, анализируя при этом содержание локальных переменных. Данное средство очень похоже на программы-отладчики, применяемые в других языках третьего поколения, например в C, а также на программу SQL-Station Debugger.

Окно PL/SQL Interpreter, изображенное на рис. 13.17, поделено на две области: просмотра и командной строки.

## Область просмотра

В области просмотра интерпретатора отображается блок, процедура или функция, выполняемые в данный момент. Блок показывается целиком, причем все строки нумеруются. Кнопки, расположенные вверху окна, используются для пошагового выполнения программного кода. Если дважды щелкнуть мышью на номере одной из строк, то в этом месте будет установлена точка прерывания. На рис. 13.18 показан тот же блок, что и на рис. 13.17, но с точкой прерывания, установленной в строке 9.

## Область командной строки

В разделе командной строки интерпретатора можно выполнять отдельные PL/SQL-операторы. Подсказка о возможности ввода SQL-оператора в этом разделе та же, что и в SQL*Plus. Однако здесь можно вводить и процедурные операторы PL/SQL, в том числе операторы присваивания, и вызовы процедур (рис. 13.18).

**Рис. 13.17.**
*PL/SQL Interpreter*

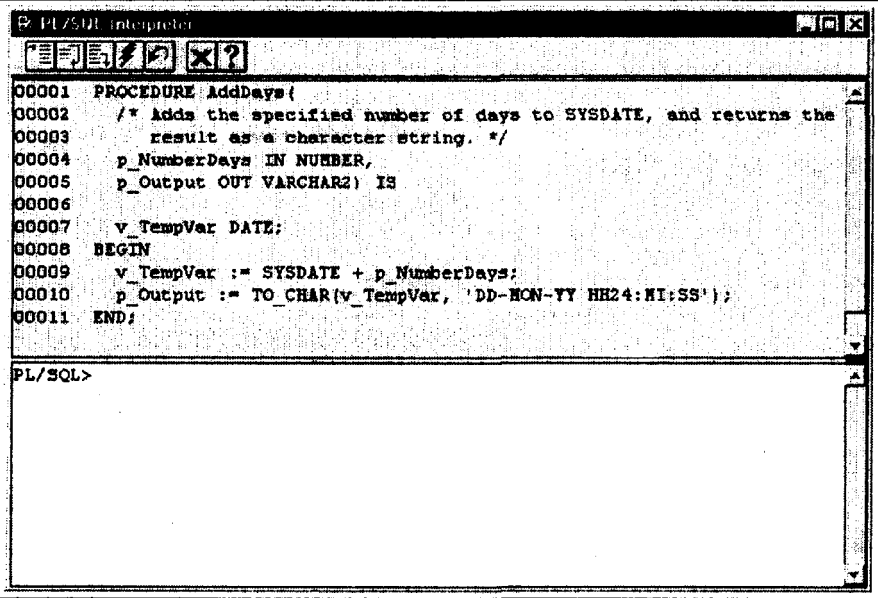
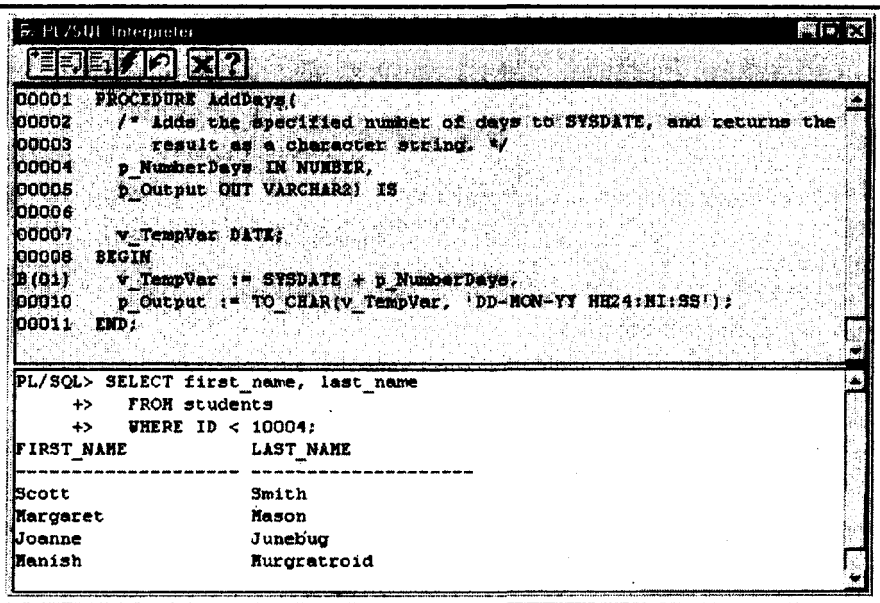



**Рис. 13.18.**
*Отладка блока PL/SQL*




## Отладка PL/SQL в диалоговом режиме

Procedure Builder дает возможность отлаживать программы PL/SQL в диалоговом режиме, поэтому данное средство просто незаменимо при разработке приложений. Все достоинства Procedure Builder более детально описаны в главе 14, а сейчас перечислим лишь некоторые из них:

- Возможность анализа значений локальных переменных во время выполнения блока. Не будь этой возможности, пришлось бы вводить в базу данных и считывать переменные только после окончания выполнения блока или выводить их содержимое на экран с помощью программного модуля DBMS_OUTPUT.
- Возможность изменения значений локальных переменных во время выполнения блока. Например, во время обработки процедуры **AddDays** можно изменить переменную **p_NumberDays** и проанализировать полученные результаты.

- Возможность выполнения команд PL/SQL и хранимых процедур из командной строки интерпретатора во время выполнения программы. Хотя это можно делать и в SQL*Plus, Procedure Builder предлагает намного больше средств для решения данной задачи.

# Оболочка PL/SQL

Когда процедура, функция или модуль хранится в базе данных, исходный текст этого объекта находится в представлении словаря данных **user_source**. Это очень удобно при разработке приложений, так как всегда можно узнать о содержимом процедуры в конкретный момент времени. Кроме того, в Procedure Builder и в других клиентских программных средствах данное представление используется для работы с хранимыми процедурами. Однако это не всегда желательно. Единственный способ ввести в эксплуатацию приложение, написанное на PL/SQL, – предоставить заказчику исходный текст этого приложения, после чего текст загружается в базу данных и компилируется на компьютере заказчика. Но это не всегда удобно, поскольку в исходном тексте могут содержаться описания алгоритмов и структур данных.

**PL/SQL 2.2 ... и ВЫШЕ**

В Oracle существует средство, позволяющее разрешить эту проблему, – оболочка (wrapper) PL/SQL (доступна в PL/SQL 2.2 и выше). Она кодирует написанный на PL/SQL исходный текст, превращая его в шестнадцатиричный код, который пользователи прочитать не могут. В базе же данных закодированные процедуры могут быть декодированы и сохранены.

## Запуск оболочки

Оболочка представляет собой программу, выполняемую операционной системой. Местонахождение и имя этой программы зависят от используемой системы, однако в большинстве систем оболочка называется WRAP. Формат вызова оболочки выглядит следующим образом:

WRAP INAME=*входной_файл* [ONAME=*выходной_файл*]

где *входной_файл* – это имя файла, содержащего оператор CREATE OR REPLACE. Для имени файла может быть указано любое расширение; по умолчанию устанавливается расширение .sql. Можно указать также параметр *выходной_файл* – имя выходного файла. Если выходной файл не указан, по умолчанию ему присваивается имя входного файла, но с расширением .plb.

Ниже приведен ряд верных командных строк WRAP. В качестве входного файла в них используется файл **register.sql**, а в качестве выходного – **register.plb**.

```
WRAP  INAME=register.sql
WRAP  INAME=register
WRAP  INAME=register.sql  ONAME=register.plb
```

Параметры INAME и ONAME не учитывают регистра символов, однако используемая операционная система может его учитывать. При этом в имени самой программы WRAP, как, впрочем, и в именах файлов, регистр символов может учитываться.

## Входные и выходные файлы

Во входном файле оболочки могут содержаться только следующие SQL-команды (помимо комментариев):

```
CREATE  [OR  REPLACE]  PROCEDURE
CREATE  [OR  REPLACE]  PACKAGE
CREATE  [OR  REPLACE]  PACKAGE  BODY
CREATE  [OR  REPLACE]  FUNCTION
```

Если входной файл, выглядит, например, так:

```
CREATE  OR  REPLACE  PROCEDURE  Register(...)  AS
   ...
BEGIN
   ...
END  Register;
```

то выходной файл будет иметь следующий вид:

```
CREATE OR REPLACE PROCEDURE Register WRAPPED
   012ba779f...
```

Весь исходный текст процедуры преобразуется в шестнадцатиричные цифры. Файл с расширением **.plb** можно загрузить из SQL*Plus точно так же, как и файл с расширением **.sql**, и это тоже приведет к созданию процедуры **Register**. Однако в представлении user_source будет храниться кодированный вариант текста, что защищает настоящий алгоритм процедуры.

Размер выходного файла обычно значительно превышает размер входного. Однако размер скомпилированного p-кода одинаков для обоих файлов, поскольку он не изменяется.

## Проверка синтаксиса и семантики

Оболочка проверяет синтаксис входного файла, но не его семантику. Это значит, что, если в файле встречаются ссылки на несуществующие объекты, — сообщения об ошибках выдаваться не будут до конца выполнения файла. Например, если закодировать файл:

```
CREATE OR REPLACE PROCEDURE CountStudents
   (p_Major IN students.major%TYPE,
   p_TotalNumber OUT NUMBER) AS
BEGIN
   SELECT COUNT(*)
      INTO p_TotalNumber
      FROM student
      WHERE major = p_Major;
END CountStudents;
/
```

то сообщения об ошибках выдаваться не будут, хотя таблица **student** и не существует (таблица должна называться **students**). Однако, если попытаться выполнить выходной файл с расширением **.plb**, будет возвращена следующая ошибка:

```
PLS-201: identifier 'student' must be declared
```

(идентификатор 'student' должен быть описан. — *Прим. пер.*)

При этом отредактировать выходной файл и устранить ошибку невозможно, так как исходный текст недоступен. Нужно модифицировать файл с расширением **.sql**, а затем закодировать его вновь.

## Рекомендации по работе с оболочкой

После кодирования процедуры, функции или модуля закодированный файл редактировать нельзя. Его содержимое невозможно прочитать и, следовательно, изменить. Единственным способом модифицировать кодированный объект является изменение файла, содержащего исходный текст. Поэтому не следует кодировать те объекты, разработка которых еще не закончена.

Тела модулей, функционирование которых поддерживает Oracle (DBMS_OUTPUT, DBMS_PIPE и др.), поставляются в закодированном виде (см. следующие главы и приложение B). Однако заголовки модулей не закодированы. Этого правила следует придерживаться и при создании собственных модулей. Часть модуля, с которой работают все пользователи (заголовок), должна быть видима, чтобы пользователи знали имена и параметры процедур, но тело модуля следует закодировать. При этом все алгоритмы его функционирования будут скрыты и доступным останется только интерфейс.

# Итоги

В этой главе были рассмотрены различные среды выполнения программ PL/SQL. Системы поддержки PL/SQL разделяются на клиентские и серверные. В число серверных сред входят SQL*Plus, предкомпиляторы Oracle, а также программные средства других фирм, например SQL-Station. С PL/SQL на станции клиента можно работать с помощью программных пакетов Oracle Forms и Procedure Builder. Кроме того, в Procedure Builder можно выполнять отладку и тестирование программ PL/SQL в диалоговом режиме. И, наконец, утилита, называемая оболочкой PL/SQL, позволяет кодировать процедуры, преобразуя их в двоичный код, и в таком виде поставлять конечным пользователям. Обсуждение методов разработки программ PL/SQL продолжается в главе 14, где также приводятся рекомендации по тестированию и отладке программ.

# Глава 14

## Тестирование и отладка

Очень немногие программы начинают правильно работать сразу же после создания. К тому же требования, предъявляемые к написанной программе, нередко меняются во время ее внедрения, и в результате приходится вносить в нее серьезные изменения. Так или иначе, необходимо тщательно протестировать созданную программу, чтобы быть уверенным, что программа работает правильно и делает то, что и было задумано. В этой главе описаны средства, применяемые при тестировании и отладке программ PL/SQL: тестовые таблицы, модуль DBMS_OUTPUT и программы-отладчики PL/SQL — Procedure Builder и SQL-Station. Помимо этих средств, рассматриваются методологии надежного программирования, помогающие избежать ошибок.

# Диагностика неисправностей

Каждая новая ошибка, возникающая во время работы программы, отличается от предыдущей, и это усложняет процессы отладки и тестирования. Конечно, можно выполнять тестирование и анализ программы на этапе ее разработки, что уменьшит общее число ошибок, но, если программа разрабатывается достаточно быстро, в ее тексте почти наверняка обнаружатся различные дефекты и неточности.

## Рекомендации по отладке программ

Хотя все дефекты различны и для устранения каждого из них существует множество способов, необходимо в общих чертах описать процесс поиска и ликвидации неисправностей. Автор предлагает ряд рекомендаций по выяснению причин возникновения ошибок. Эти рекомендации основаны на его личном опыте и на опыте других программистов и применимы к программам, написанным не только на PL/SQL, но и на любом другом языке программирования.

### Поиск места возникновения ошибки

Определение места возникновения ошибки является крайне важной задачей. Если ошибка происходит в большой сложной программе, первым делом необходимо определить, в каком именно месте она произошла. Это не всегда просто и зависит от сложности текста конкретной программы. Самым легким способом обнаружить причину ошибки является трассировка программы на этапе ее выполнения с анализом значений, содержащихся в структурах данных.

### Определение вида ошибки

Узнав, в каком месте программы произошла ошибка, нужно точно определить ее вид. Является ли она ошибкой Oracle? Может быть, к неверному результату привели какие-то вычисления? Или в базу данных вводилась неточная информация? Чтобы выйти из сложившейся ситуации, необходимо иметь представление, как проявляет себя данная ошибка.

### Сокращение программы для построения тестового примера

Когда местонахождение ошибки неизвестно, рекомендуется постепенно сокращать программу до построения тестового примера. Удалив какой-либо фрагмент, перезапустите программу. Если ошибка сохранилась, значит, не данный фрагмент был ее причиной. Если же ошибка больше себя не проявляет, внимательно проанализируйте удаленный фрагмент.

Имейте в виду, что иногда причиной ошибки является одна область программы, а проявляет себя эта ошибка совсем в другом фрагменте. Например, некоторая процедура может возвращать неверное значение, которое затем используется в основной программе. Источником ошибки в этом случае является не основная программа (где возникает ошибка), а процедура. Если из программы убрать вызов данной процедуры и заменить его операцией прямого присваивания возвращаемого значения, причина неисправности станет очевидна. Такая ситуация рассмотрена ниже в этой главе.

### Создание среды тестирования

В идеале тестирование и отладка программ не должны выполняться в производственной среде. Для тестирования следует создать среду, максимально воспроизводящую производственную, — например такую же базу данных, — но с меньшим объемом хранимой в ней информации. Это позволит разрабатывать и проверять новые варианты прикладных программ, не изменяя эксплуатируемую производственную систему. Если в производственной среде проявилась какая-либо неисправность, в первую очередь попытайтесь воспроизвести эту неисправность в тестовой системе, что, кстати, отвечает правилу сокращения программы, сформулированному выше. Сокращать можно не только текст программы: PL/SQL в большой степени зависит от структуры базы данных и от вида информации, поэтому можно упростить и их.

## Модуль Debug

Язык PL/SQL создан для работы в первую очередь с информацией, хранимой в базах данных Oracle, и структура PL/SQL полностью отвечает этому требованию. Однако для выполнения практических задач необходимы дополнительные средства, помогающие создавать и отлаживать программы.

Далее в этой главе подробно рассматриваются методы отладки программ PL/SQL. В каждом разделе анализируется определенный метод выявления дефекта с учетом рекомендаций, приведенных выше, причем сначала описывается общий метод отладки, а затем способ устранения конкретной проблемы. Одновременно рассматриваются различные варианты модуля Debug, используемого для отладки программ. Выбор необходимого варианта полностью зависит от применяемой системы и потребностей пользователей.

# Ввод данных в тестовые таблицы

Самым простым способом отладки программ является ввод значений локальных переменных во временную таблицу на этапе выполнения программы. После завершения работы программы можно обратиться к такой таблице и определить значения локальных переменных. Этот метод требует минимума усилий и применим в любой среде, поскольку в программу добавляется лишь несколько операторов INSERT.

## Ситуация 1

Предположим, что необходимо создать функцию, которая будет возвращать среднюю оценку успеваемости для каждой учебной группы исходя из текущего числа зарегистрированных в группе студентов. Создадим такую функцию:

```
-- Этот пример содержится в файле avgrade1.sql.
CREATE OR REPLACE FUNCTION AverageGrade (
/* Определяет среднюю оценку для указанной группы. Оценки
    сохраняются в таблице registered_students в виде одиночных
    символов от A до E. Средняя оценка возвращается также в виде
    одиночной буквы. Если в группе не зарегистрировано ни одного
    студента, устанавливается исключительная ситуация. */
  p_Department IN VARCHAR2,
  p_Course IN NUMBER) RETURN VARCHAR2 AS

  v_AverageGrade VARCHAR2(1);
  v_NumericGrade NUMBER;
  v_NumberStudents NUMBER;
  CURSOR c_Grades IS
      SELECT grade
        FROM registered_students
        WHERE department = p_Department
        AND course = p_Course;
BEGIN
  /* Сначала нужно узнать число студентов данной группы. Если
      студентов нет, устанавливается ошибка. */
  SELECT COUNT(*)
      INTO v_NumberStudents
      FROM registered_students
      WHERE department = p_Department
        AND course = p_Course;

  IF v_NumberStudents = 0 THEN
      RAISE_APPLICATION_ERROR(-20001, 'No students registered for ' ||
        p_Department || ' ' || p_Course);
  END IF;
```

```
    /* Поскольку оценки представляют собой буквы, функцию AVG
       использовать по отношению к ним нельзя. Вместо этого
       преобразуем буквы в числовые значения с помощью функции DECODE
       и определим среднее для полученных чисел. */
    SELECT AVG(DECODE(grade, 'A', 5,
                             'B', 4,
                             'C', 3,
                             'D', 2,
                             'E', 1))
        INTO v_NumericGrade
        FROM registered_students
        WHERE department = p_Department
        AND course = p_Course;

    /* В v_NumericGrade теперь содержится средняя оценка в виде
       числа от 1 до 5. Необходимо преобразовать ее обратно в букву.
       Для этого вновь воспользуемся функцией DECODE. Обратите
       внимание: результат считывается в v_AverageGrade, а не
       присваивается этой переменной, так как функцию DECODE можно
       применять только в SQL-операторах. */
    SELECT DECODE(ROUND(v_NumericGrade), 5, 'A',
                                         4, 'B',
                                         3, 'C',
                                         2, 'D',
                                         1, 'E')
        INTO v_AverageGrade
        FROM dual;
    RETURN v_AverageGrade;
END AverageGrade;
```

**Предположим, что содержимое таблицы registered_students таково:**

```
SQL> select * from registered_students;

  STUDENT_ID   DEP  COURSE  G
-----------  ----   -----  ---
       10000   CS       102  A
       10002   CS       102  B
       10003   CS       102  C
       10000   HIS      101  A
       10001   HIS      101  B
       10002   HIS      101  B
       10003   HIS      101  A
       10004   HIS      101  C
       10005   HIS      101  C
       10006   HIS      101  E
       10007   HIS      101  B
       10008   HIS      101  A
       10009   HIS      101  D
       10010   HIS      101  A
       10008   NUT      307  A
       10010   NUT      307  A
       10009   MUS      410  B
       10006   MUS      410  E
18 rows selected.
```

▼ ВНИМАНИЕ Таблица **registered_students** заполнена этими 18-ю строками в соответствии с *файлом-сценарием tables.sql. (Более подробно о файле tables.sql см. главу 1.)*

Студенты зарегистрированы в четырех группах: Computer Science 102, History 101, Nutrition 307 и Music 410. Следовательно, для каждой из этих групп можно вызвать функцию **AverageGrade**. Для всех других групп будет устанавливаться исключительная ситуация "No sudents registered" (студенты не зарегистрированы). Ниже приведен пример результатов, полученных с помощью SQL*Plus.

```
SQL> VARIABLE v_AveGrade VARCHAR2(1)
SQL> exec :v_AveGrade := AverageGrade('HIS', 101)
PL/SQL procedure successfully completed.

SQL> print v_AveGrade

V_AVEGRADE
----------------------------------------------------------------
B

SQL> exec :v_AveGrade := AverageGrade('NUT', 307)

PL/SQL procedure successfully completed.

SQL> print v_AveGrade

V_AVEGRADE
----------------------------------------------------------------
A

SQL> exec :v_AveGrade := AverageGrade('MUS', 410)

PL/SQL procedure successfully completed.

SQL> print v_AveGrade

V_AVEGRADE
----------------------------------------------------------------
C

SQL> exec :v_AveGrade := AverageGrade('CS', 102)
begin :v_AveGrade := AverageGrade('CS', 102); end;

*
ERROR at line 1:
ORA-20001: No sudents registered for CS 102
ORA-06512: at "EXAMPLE.AVERAGEGRADE", line 21
```

Последний вызов демонстрирует возможный дефект программы. В результате выполнения этого вызова возвращается ошибка ORA-20001, хотя в группе Computer Science 102 студенты зарегистрированы.

## Ситуация 1: модуль Debug

Ниже показан один из вариантов модуля Debug, используемый для выявления данного дефекта. В этом *модуле основной процедурой является* **Debug.Debug**. *Она имеет два параметра, которые представляют* описание ошибки и некоторое значение. Эти параметры конкатенируются и вносятся в таблицу **debug_table**. Процедуру **Debug.Reset** следует вызывать в начале программы для инициализации таблицы и внутреннего счетчика строк, который нужен для гарантии выбора строк таблицы **debug_table** в том порядке, в котором они были введены.

```
-- Этот пример содержится в файле debug1.sql.
CREATE OR REPLACE PACKAGE Debug AS
```

```
    /* Первый вариант модуля Debug. Задачей этого модуля является ввод
       данных в таблицу debug_table. Чтобы увидеть результат, нужно
       считать информацию debug_table в SQL*Plus следующим образом:
  SELECT debug_str FROM debug_table ORDER BY linecount; */

    /* Это основная процедура отладки. p_Description будет
       конкатенироваться с p_Value и вводиться в таблицу debug_table. */
    PROCEDURE Debug(p_Description IN VARCHAR2, p_Value IN VARCHAR2);

    /* Начальная установка среды отладки. Процедура Reset вызывается при
       первой конкретизации модуля, а также для удаления содержимого
       таблицы debug_table при установке нового соединения. */
    PROCEDURE Reset;
  END Debug;

  CREATE OR REPLACE PACKAGE BODY Debug AS
    /* v_LineCount используется для упорядочения строк таблицы debug_table. */
    v_LineCount NUMBER;

    PROCEDURE Debug(p_Description IN VARCHAR2, p_Value IN VARCHAR2) IS
    BEGIN
        INSERT INTO debug_table (linecount, debug_str)
          VALUES (v_LineCount, p_Description || ': ' || p_Value);
        COMMIT;
        v_LineCount := v_LineCount + 1;
    END Debug;

    PROCEDURE Reset IS
    BEGIN
        v_LineCount := 1;
        DELETE FROM debug_table;
    END Reset;

  BEGIN /* Инициализация модуля. */
    Reset;
  END Debug;
```

## Ситуация 1: использование модуля Debug

Чтобы установить причину неверной работы процедуры **AverageGrade**, необходимо проанализировать значения используемых ею переменных. Внесем в текст процедуры несколько отладочных операторов. В данном варианте **Debug** нужно вызывать **Debug.Reset** при запуске **AverageGrade**, а **Debug.Debug** – при необходимости просмотреть содержимое одной из переменных. Модифицируем модуль следующим образом (для сокращения текста программы некоторые комментарии удалены):

```
-- Этот пример содержится в файле avgrade2.sql.
CREATE OR REPLACE FUNCTION AverageGrade (
  p_Department IN VARCHAR2,
  p_Course IN NUMBER) RETURN VARCHAR2 AS

  v_AverageGrade VARCHAR2(1);
  v_NumericGrade NUMBER;
  v_NumberStudents NUMBER;

  CURSOR c_Grades IS
      SELECT grade
        FROM registered_students
```

```
        WHERE department = p_Department
        AND course = p_Course;
BEGIN
  Debug.Reset;
  Debug.Debug('p_Department', p_Department);
  Debug.Debug('p_Course', p_Course);

  /* Сначала нужно узнать число студентов данной группы. Если
     студентов нет, устанавливается ошибка. */
  SELECT COUNT(*)
      INTO v_NumberStudents
      FROM registered_students
      WHERE department = p_Department
      AND course = p_Course;

  Debug.Debug('After select, v_NumberStudents', v_NumberStudents);
  IF v_NumberStudents = 0 THEN
      RAISE_APPLICATION_ERROR(-20001, 'No students registered for ' ||
        p_Department || ' ' || p_Course);
  END IF;
  SELECT AVG(DECODE(grade, 'A', 5,
                           'B', 4,
                           'C', 3,
                           'D', 2,
                           'E', 1))
      INTO v_NumericGrade
      FROM registered_students
      WHERE department = p_Department
      AND course = p_Course;

  SELECT DECODE(ROUND(v_NumericGrade), 5, 'A',
                                       4, 'B',
                                       3, 'C',
                                       2, 'D',
                                       1, 'E')
      INTO v_AverageGrade
      FROM dual;
  RETURN v_AverageGrade;
END AverageGrade;
```

Вновь вызовем **AverageGrade** и проанализируем содержимое таблицы **debug_table**:

```
SQL> EXEC :v_AveGrade := AverageGrade('CS', 102)
begin :v_AveGrade := AverageGrade('CS', 102); end;

*
ERROR at line 1:
ORA-20001: No sudents registered for CS 102
ORA-06512: at "EXAMPLE.AVERAGEGRADE", line 25
ORA-06512: at line 1

SQL> SELECT debug_str FROM debug_table ORDER BY linecount;
```

```
DEBUG_STR
-----------------------------------------
p_Department: CS
p_Course: 102
After select, v_NumberStudents: 0
```

Итак, в переменной **v_NumberStudents** находится 0, чем и объясняется сообщение об ошибке ORA-20001. Таким образом, все дело в операторе SELECT, который не отбирает ни одной строки. Следовательно, нужно внимательно проанализировать условие WHERE этого оператора:

```
SELECT COUNT(*)
  INTO v_NumberStudents
  FROM registered_students
  WHERE department = p_Department
  AND course = p_Course;
```

Похоже, что значения для **p_Department** и **p_Course**, выводимые на экран, правильные, однако в SQL*Plus не указывается, где может находиться символ новой строки, поэтому эти значения могут быть неверны. Изменим вызов **Debug.Debug,** заключив **p_Department** и **p_Course** в кавычки и тем самым устранив все начальные и конечные пробелы:

```
-- Этот пример содержится в файле avgrade3.sql.
CREATE OR REPLACE FUNCTION AverageGrade (
  ...
BEGIN
  Debug.Reset;
  Debug.Debug('p_Department', '''' || p_Department || '''');
  Debug.Debug('p_Course', '''' || p_Course || '''');

  /* Сначала нужно узнать число студентов данной группы. Если
     студентов нет, устанавливается ошибка. */
  SELECT COUNT(*)
      INTO v_NumberStudents
      FROM registered_students
      WHERE department = p_Department
      AND course = p_Course;

Debug.Debug('After select, v_NumberStudents', v_NumberStudents);
...
```

Теперь, если запустить на выполнение **AverageGrade** и просмотреть таблицу **debug_table**, будет получен такой результат:

```
SQL> exec :v_AveGrade := AverageGrade('CS', 102)
begin :v_AveGrade := AverageGrade('CS', 102); end;

*
ERROR at line 1:
ORA-20001: No sudents registered for CS 102
ORA-06512: at "EXAMPLE.AVERAGEGRADE", line 25
ORA-06512: at line 1

SQL> SELECT debug_str FROM debug_table ORDER BY linecount;

DEBUG_STR
-----------------------------------------------------------------
p_Department: 'CS'
p_Course: '102'
After select, v_NumberStudents: 0
```

Видно, что **p_Department** не содержит конечного пробела. Это неправильно, поскольку тип столбца **department** таблицы **registered_students** задан как CHAR(3), а **p_Department** имеет тип VARCHAR2. Таким образом, в столбце базы данных записано значение 'CS ' (с конечным пробелом), и именно по этой причине оператор SELECT не возвратил ни одной строки, а переменной **v_NumberStudents** было присвоено значение 0.

▼ СОВЕТУЕМ Для исследования содержимого столбца базы данных можно воспользоваться встроенной функцией DUMP. Для примера просмотрим информацию, хранящуюся в столбце **department** таблицы **registered_students**:

```
SQL> SELECT DISTINCT DUMP(department)
  2    FROM registered_students
  3    WHERE department = 'CS';

DUMP(DEPARTMENT)
-----------------------------------------
Typ=96 Len=3: 67,83,32
```

Тип отображается как 96, что соответствует типу CHAR, а последним байтом столбца является 32 (код ASCII для символа пробела). Таким образом, информация этого столбца дополняется пробелами. (Более подробно о функции DUMP и о других встроенных функциях см. в главе 5.) Коды типов данных приведены в руководстве Oracle SQL Reference.

Решением проблемы может стать изменение типа переменной **p_Department** на CHAR:

```
CREATE OR REPLACE FUNCTION AverageGrade (
  p_Department IN CHAR,
  p_Course IN NUMBER) RETURN VARCHAR2 AS
  ...
BEGIN
  ...
END AverageGrade;
```

После этого выдается правильный результат для **AverageGrade**:

```
SQL> exec :v_AveGrade := AverageGrade('CS', 102)

PL/SQL procedure successfully completed.

SQL> print v_AveGrade

V_AVEGRADE
-----------------------------------------------------------
B
```

Функция была выполнена правильно благодаря тому, что оба значения в условии WHERE имеют тип CHAR и что при сравнении этих значений применяется метод дополнения пробелами, помогающий установить соответствие. (Более подробно о семантике сравнения символов см. в главе 2.)

▼ СОВЕТУЕМ Если бы при описании функции был использован атрибут %TYPE, тип переменной **p_Department** был бы правильным. Это еще один довод в пользу применения %TYPE. Кроме того, поскольку значение **AverageGrade** является строкой символов длиной 1, и всегда будет такой длины, то типом данных фиксированной длины (CHAR) можно воспользоваться и в конструкции RETURN. В этом случае описание **AverageGrade** будет выглядеть следующим образом:

```
-- Этот пример содержится в файле avgrade4.sql.
CREATE OR REPLACE FUNCTION AverageGrade (
  p_Department IN registered_students.department%TYPE,
  p_Course IN registered_students.course%TYPE) RETURN CHAR AS

  v_AverageGrade CHAR(1);
```

```
    v_NumericGrade  NUMBER;
    v_NumberStudents  NUMBER;
  ...
BEGIN
  ...
END  AverageGrade;
```

## Ситуация 1: комментарии

Данный вариант модуля Debug очень прост. Его задачей является лишь ввод информации в таблицу **debug_table**, однако с его помощью был обнаружен дефект в тексте программы **AverageGrade**. У такого метода есть ряд преимуществ:

- В модуле Debug используются только SQL-операторы, поэтому его можно выполнять в любой среде. Оператор SELECT, применяемый для вывода результатов, может быть запущен из SQL*Plus или из другого средства.;
- Процесс отладки достаточно прост, поэтому в отлаживаемую процедуру не вносится слишком много дополнительных операторов.

Однако у этого метода имеются и недостатки:

- При выполнении **AverageGrade** устанавливается исключительная ситуация, что приводит к откату всех операций SQL. Следовательно, в процедуре **Debug.Debug** нужно задать оператор COMMIT для гарантии того, что вводимая в таблицу **debug_table** информация не будет откатываться. Это может породить трудности, если не нужно выполнять откат других операций в отлаживаемой процедуре. Кроме того, оператор завершения становится недействительным при открытии курсоров типа SELECT FOR UPDATE;
- Модуль Debug не работает корректно, когда его одновременно используют несколько соединений, так как оператор SELECT будет возвращать результаты для всех соединений. Этот недостаток можно устранить, если изменить модуль Debug и таблицу **debug_table**, включив в ее состав столбцы, однозначно идентифицирующие каждое соединение.

Недостатки данного варианта модуля Debug устраняются с помощью модуля DBMS_OUTPUT, описанного в следующем разделе.

# DBMS_OUTPUT

Первый вариант модуля Debug, рассмотренный выше, по существу, реализует ограниченный ввод/вывод информации. В PL/SQL преднамеренно не предусмотрены средства, обеспечивающие ввод/вывод, так как для работы с данными, хранимыми в базе данных, выводить значения переменных и структуры данных не нужно. Тем не менее при отладке программ средства ввода/вывода весьма полезны. Именно поэтому в состав PL/SQL 2.0 был включен модуль DBMS_OUTPUT, обеспечивающий вывод информации. Использование этого модуля демонстрируется во втором варианте модуля Debug, описанном в данном разделе.

Средства ввода данных в языке PL/SQL до сих пор не существует. Однако вводить информацию можно с помощью переменных подстановки SQL*Plus (см. главу 13). В PL/SQL 2.3 имеется модуль UTL_FILE, используемый для обмена информацией с файлами операционной системы (см. главу 18).

## Модуль DBMS_OUTPUT

Опишем некоторые свойства модуля DBMS_OUTPUT. Его владельцем, как и других модулей DBMS (СУБД), является пользователь Oracle с именем SYS. В файле-сценарии создания DBMS_OUTPUT роли PUBLIC предоставляется полномочие EXECUTE на этот модуль; кроме того, для модуля создается общий синоним. Это означает, что любой пользователь Oracle может вызывать подпрограммы модуля DBMS_OUTPUT, не указывая перед его именем SYS.

Каковы же принципы работы DBMS_OUTPUT? С помощью процедур этого модуля реализованы две базовые операции — GET (получить) и PUT (поместить). Операция PUT берет свои аргументы и помещает их во внутренний буфер для хранения. Операция GET читает этот буфер и возвращает его содержимое процедуре в качестве аргумента. Размер буфера устанавливается с помощью процедуры ENABLE (разрешить).

## Процедуры в DBMS_OUTPUT

Выполнение операции PUT обеспечивают процедуры PUT, PUT_LINE и NEW_LINE, а выполнение операции GET – процедуры GET_LINE и GET_LINES. Управляют буфером процедуры ENABLE и DISABLE.

**PUT и PUT_LINE** Процедуры PUT и PUT_LINE вызываются следующим образом:

```
PROCEDURE PUT(a VARCHAR2);
PROCEDURE PUT(a NUMBER);
PROCEDURE PUT(a DATE);

PROCEDURE PUT_LINE(a VARCHAR2);
PROCEDURE PUT_LINE(a NUMBER);
PROCEDURE PUT_LINE(a DATE);
```

где *a* – аргумент, помещаемый в буфер. Заметьте, что эти процедуры переопределяются типом параметра (см. главу 7). Существует по три различных варианта PUT и PUT_LINE, поэтому в буфере могут содержаться значения типов VARCHAR2, NUMBER и DATE, причем эти значения хранятся в буфере в первоначальном формате. Процедуры же GET_LINE и GET_LINES прочитывают буфер и возвращают только строки символов. При выполнении операции GET содержимое буфера преобразуется в строку символов в соответствии с правилами преобразования типов данных, установленными по умолчанию. Если при преобразовании необходимо указать формат, следует явно вызвать функцию TO_CHAR для PUT, а не для GET.

Буфер организован в виде строк, состоящих не более чем из 255 байт каждая. PUT_LINE добавляет к аргументу символ новой строки, указывая тем самым конец строки; PUT же этого не делает. Вызов PUT_LINE аналогичен вызову PUT с последующим вызовом NEW_LINE.

**NEW_LINE** Процедура NEW_LINE вызывается следующим образом:

```
PROCEDURE NEW_LINE;
```

NEW_LINE помещает в буфер символ новой строки, указывая тем самым конец строки. При этом ограничения на число строк в буфере не существует и его размер ограничивается лишь значением, указанным в ENABLE.

**GET_LINE** Процедура GET_LINE вызывается следующим образом:

```
PROCEDURE GET_LINE(line OUT VARCHAR2, status OUT INTEGER);
```

где *line* – это последовательность символов одной строки буфера, а *status* указывает, успешно или нет была считана эта строка. Максимальная длина строки – 255 байт. Если строка считана, в переменной status будет находиться 0, а если в буфере строк больше нет – то 1.

▼ ВНИМАНИЕ **Хотя максимальный размер строки буфера равен 255 байт, в строке выходной переменной может содержаться более 255 байт. Например, в буфере могут находиться значения типа DATE, каждое из которых занимает 7 байт памяти буфера, но преобразуется в последовательность символов длиной, как правило, более 7 байт.**

**GET_LINES** Аргументом процедуры GET_LINES является таблица PL/SQL. Тип таблицы и вызов данной процедуры выглядят следующим образом:

```
TYPE CHARARR IS TABLE OF VARCHAR2(255)
INDEX BY BINARY INTEGER;
PROCEDURE GET_LINES(lines OUT CHARARR,
numlines IN OUT INTEGER);
```

где *lines* – это таблица PL/SQL, в которую будут записаны строки из буфера, а *numlines* – число запрошенных строк. На входе GET_LINES в переменной *numlines* указывается число запрошенных строк, а на выходе – фактическое число возвращаемых строк, которое будет меньше или равно числу запрошенных. Вызов GET_LINES заменяет несколько вызовов GET_LINE.

Тип CHARARR определен в модуле DBMS_OUTPUT. Поэтому, если GET_LINES вызывается явным образом, нужно объявить какую-либо переменную с типом DBMS_OUTPUT.CHARARR. Например:

```
-- Этот пример содержится в файле output.sql.
DECLARE
  /* Демонстрирует использование PUT_LINE и GET_LINE. */
  V_Data      DBMS_OUTPUT.CHARARR;
  V_NumLines  NUMBER;
BEGIN
```

```
    -- Сначала разрешим работу с буфером.
    DBMS_OUTPUT.ENABLE(1000000);

    -- Поместим в буфер данные, чтобы процедура GET_LINES
    -- смогла что-либо оттуда считать.
    DBMS_OUTPUT.PUT_LINE('Line One');
    DBMS_OUTPUT.PUT_LINE('Line Two');
    DBMS_OUTPUT.PUT_LINE('Line Three');

    -- Установим максимальное число строк, которые нужно считать.
    v_NumLines := 3;

    /* Считаем содержимое буфера. Обратите внимание: переменная
       v_Data объявлена с типом DBMS_OUTPUT.CHARARR, так что ее
       описание соответствует описанию, требуемому при работе с
       DBMS_OUTPUT.GET_LINES. */
    DBMS_OUTPUT.GET_LINES(v_Data, v_NumLines);

    /* Последовательно просмотрим возвращаемый буфер и внесем его
       содержимое в таблицу temp_table. */
    FOR v_Counter IN 1..v_NumLines LOOP
       INSERT INTO temp_table (char_col)
          VALUES (v_Data(v_Counter));
    END LOOP;
END;
```

**ENABLE и DISABLE** Процедуры ENABLE и DISABLE вызываются следующим образом:

```
PROCEDURE ENABLE(buffer_size IN INTEGER DEFAULT 20000);
PROCEDURE DISABLE;
```

где *buffer_size* – это первоначальный размер внутреннего буфера в байтах. По умолчанию задается размер 20 000 байтов, а максимальный размер – 1 000 000 байтов. Впоследствии в этот буфер будут записываться аргументы процедур PUT и PUT_LINE. Эти аргументы хранятся в своем внутреннем формате, занимая столько пространства буфера, сколько определяет их структура. Если объявлена процедура DISABLE, то содержимое буфера уничтожается, и последующие вызовы PUT и PUT_LINE не будут иметь никакого эффекта.

## Использование модуля DBMS_OUTPUT

В самом модуле DBMS_OUTPUT не предусмотрено какого-либо механизма вывода данных. По существу, в нем реализуется алгоритм "первым пришел – первым обслужен". Так как же с его помощью осуществить вывод информации? Для этого в каждой из утилит SQL*Plus, SQL*DBA и Server Manager имеется средство, называемое SERVEROUTPUT (серверный вывод). Кроме того, некоторые продукты других фирм (в том числе и в SQL-Station) обладают возможностью отображать на экране данные, используемые при работе с DBMS_OUTPUT. Когда эта возможность разрешена, SQL*Plus будет автоматически вызывать DBMS_OUTPUT.GET_LINES при завершении блока PL/SQL и выводить полученные результаты на экран (рис. 14.1).

Команда SQL*Plus, называемая SET SERVEROUTPUT ON, неявно вызывает процедуру DBMS_OUTPUT.ENABLE, устанавливающую внутренний буфер. Если нужно, можно указать указать размер буфера с помощью команды

```
SET SERVEROUTPUT ON SIZE размер_буфера
```

где *размер_буфера* – это первоначальный размер буфера (аргумент процедуры DBMS_OUTPUT.ENABLE). При разрешенном серверном выводе SQL*Plus будет вызывать процедуру DBMS_OUTPUT.GET_LINES после окончания блока PL/SQL. Это значит, что результаты будут выводиться на экран *после* завершения блока, а не во время его выполнения. Как правило, это не вызывает каких-либо затруднений в работе с модулем DBMS_OUTPUT при отладке программ.

▼ ОСТОРОЖНО Модуль DBMS_OUTPUT предназначен в первую очередь для отладки программ, а не для вывода на экран каких-либо сообщений. Если нужно получить на экране результаты запросов в удобном виде, используйте не DBMS_OUTPUT и SQL*Plus, а специальные средства, например Oracle Reports.

**Рис. 14.1.** *Использование SERVEROUTPUT и PUT_LINE*

```
File Edit Search Options Help
SQL> SET serveroutput on SIZE 1000000
SQL> BEGIN
  2    DBMS_OUTPUT.PUT_LINE('Before loop');
  3    FOR v_Counter IN 1..10 LOOP
  4      DBMS_OUTPUT.PUT_LINE('Inside loop, counter = ' || v_Counter);
  5    END LOOP;
  6    DBMS_OUTPUT.PUT_LINE('After loop');
  7  END;
  8  /
Before loop
Inside loop, counter = 1
Inside loop, counter = 2
Inside loop, counter = 3
Inside loop, counter = 4
Inside loop, counter = 5
Inside loop, counter = 6
Inside loop, counter = 7
Inside loop, counter = 8
Inside loop, counter = 9
Inside loop, counter = 10
After loop

PL/SQL procedure successfully completed.

SQL>
```

Для внутреннего буфера задается максимальный размер (указывается в DBMS_OUTPUT.ENABLE), а максимальная длина каждой строки составляет 255 байт. Поэтому вызовы процедур DBMS_OUTPUT.PUT, DBMS_OUTPUT.PUT_LINE и DBMS_OUTPUT.NEW_LINE могут привести к установлению исключительных ситуаций:

- ORA-20000: ORU-10027: buffer overflow, limit of <предельный_размер_буфера> bytes.
  (переполнение буфера, ограничение <предельный_размер_буфера> байт. — *Прим. пер.*),

или

- ORA-20000: ORU-10028: line length overflow, limit of 255 bytes per line.
  (переполнение строки, ограничение 255 байт на строку. — *Прим. пер.*)

Тип возвращаемого сообщения зависит от того, какое ограничение нарушено.
Более подробно о модуле DBMS_OUTPUT о других модулях DBMS рассказано в приложении В.

▼ СОВЕТУЕМ Рекомендуется всегда указывать размер буфера в команде SET SERVEROUTPUT ON. Если этого не сделать, SQL*Plus будет вызывать DBMS_OUTPUT.ENABLE с размером буфера 2000 байт, хотя значение по умолчанию для DBMS_OUTPUT.ENABLE равно 20 000 байт.

## Ситуация 2

В таблице students предусмотрен столбец для хранения текущего числа зачетов по всем курсам, на которые зарегистрировался каждый студент. Процедура **Register** не обновляет этот столбец, оставляя в нем текущую информацию. Для устранения этого дефекта можно создать функцию, которая будет подсчитывать общее число зачетов по всем курсам, на которые студент зарегистрирован. После этого процедура **Register** может обновить столбец **current_credits** таблицы **students.** Создадим функцию **CountCredits**:

```
-- Этот пример содержится в файле cntcred1.sql.
CREATE OR REPLACE FUNCTION CountCredits (
  /* Возвращает число зачетов, на которое зарегистрирован студент,
     указанный в p_StudentID. */
  p_StudentID IN students.ID%TYPE)
  RETURN NUMBER AS
```

```
    v_TotalCredits NUMBER;      -- Общее число зачетов.
    v_CourseCredits NUMBER;     -- Зачеты для одного курса обучения.
    CURSOR c_RegisteredCourses IS
        SELECT department, course
          FROM registered_students
          WHERE student_id = p_StudentID;
BEGIN
    FOR v_CourseRec IN c_RegisteredCourses LOOP
        -- Определим число зачетов для данной группы.
        SELECT num_credits
          INTO v_CourseCredits
          FROM classes
          WHERE department = v_CourseRec.department
          AND course = v_CourseRec.course;

        -- Добавим полученное число к тому, что было раньше.
        v_TotalCredits := v_TotalCredits + v_CourseCredits;
    END LOOP;

    RETURN v_TotalCredits;
END CountCredits;
```

Поскольку **CountCredits** не изменяет состояния базы данных или модуля, в PL/SQL 2.1 и выше можно вызывать эту функцию непосредственно из SQL-операторов (см. главу 8). Таким образом, можно узнать общее число зачетов для всех студентов, если просмотреть таблицу **students**. Выдается следующий результат:

```
SQL> SELECT ID, CountCredits(ID) "Total Credits"
  2    FROM students;

        ID Total Credits
---------- --------------------
     10000
     10001
     10002
     10003
     10004
     10005
     10006
     10007
     10008
     10009
     10010
11 rows selected.
```

Можно видеть, что результата выполнения **CountCredits** как такового нет, т.е. функция возвращает NULL-значения, а это неверно.

## Ситуация 2: модуль Debug

Для выявления дефекта в **CountCredits** воспользуемся модулем DBMS_OUTPUT и создадим новый вариант модуля Debug с тем же интерфейсом, что и предыдущий; поэтому нужно изменить лишь его тело.

```
-- Этот пример содержится в файле debug2.sql.
CREATE OR REPLACE PACKAGE BODY Debug AS
  PROCEDURE Debug(p_Description IN VARCHAR2,
                  p_Value IN VARCHAR2) IS
  BEGIN
```

```
        DBMS_OUTPUT.PUT_LINE(p_Description || ': ' || p_Value);
    END Debug;

    PROCEDURE Reset IS
    BEGIN
          /* Сначала запретим буфер, а затем разрешим его создание
             с максимальным размером. DISABLE уничтожает буфер, поэтому
             при вызове Reset буфер всегда будет новым. */
        DBMS_OUTPUT.DISABLE;
        DBMS_OUTPUT.ENABLE(1000000);
    END Reset;
BEGIN /* Инициализация модуля */
    Reset;
END Debug;
```

Таблица **debug_table** более не используется; вместо нее применяется модуль DBMS_OUTPUT. Следовательно, этот вариант Debug будет работать только в SQL*Plus, SQL*DBA, Server Manager или SQL-Station, так как данные средства автоматически вызывают DBMS_OUTPUT.GET_LINES и распечатывают содержимое буфера. Кроме того, перед использованием Debug необходимо разрешить SERVEROUTPUT.

## Ситуация 2: использование модуля Debug

Функция **CountCredits** возвращает NULL-значения. Чтобы убедиться в этом, а также узнать, какое значение добавляется в цикле к **v_TotalCredits**, внесем в описание функции несколько вызовов Debug:

```
-- Этот пример содержится в файле cntcred2.sql.
CREATE OR REPLACE FUNCTION CountCredits (
  /* Возвращает число зачетов, на которое зарегистрирован студент,
     указанный в p_StudentID. */
  p_StudentID IN students.ID%TYPE)
  RETURN NUMBER AS

  v_TotalCredits NUMBER;      -- Общее число зачетов
  v_CourseCredits NUMBER;     -- Зачеты для одного курса обучения
  CURSOR c_RegisteredCourses IS
      SELECT department, course
        FROM registered_students
        WHERE student_id = p_StudentID;
BEGIN
  Debug.Reset;
  FOR v_CourseRec IN c_RegisteredCourses LOOP
      -- Определим число зачетов для данной группы.
      SELECT num_credits
        INTO v_CourseCredits
        FROM classes
        WHERE department = v_CourseRec.department
        AND course = v_CourseRec.course;

      Debug.Debug('Inside loop, v_CourseCredits', v_CourseCredits);
      -- Добавим полученное число к тому, что было раньше.
      v_TotalCredits := v_TotalCredits + v_CourseCredits;
  END LOOP;

  Debug.Debug('After loop, returning', v_TotalCredits);
  RETURN v_TotalCredits;
END CountCredits;
```

Получается следующий результат:

```
SQL> VARIABLE v_Total NUMBER
SQL> SET SERVEROUTPUT ON
SQL> exec :v_Total := CountCredits(10006);
Inside loop, v_CourseCredits: 4
Inside loop, v_CourseCredits: 3
After loop, returning:

PL/SQL procedure successfully completed.

SQL> print v_Total

  V_TOTAL
-------------
SQL>
```

▼ ВНИМАНИЕ

Тестирование функции **CountCredits** выполняется с использованием переменных привязки SQL*Plus, а не путем выбора значения функции в таблице **students.** Дело в том, что **CountCredits**, который не является функцией в чистом виде, теперь вызывает DBMS_OUTPUT. Если бы функция **CountCredits** была применена в SQL-операторе, было бы выдано сообщение об ошибке ORA-6571. Об этой ошибке и о вызове хранимых функций в SQL-операторах более подробно рассказано в главе 8.

Проанализировав результаты, можно решить, что число зачетов, определенных для каждой группы, правильно: цикл был выполнен дважды, и были получены значения 4 и 3 соответственно. Однако ясно, что эти значения не добавились к общему числу. Добавим еще несколько отладочных операторов:

```
-- Этот пример содержится в файле cntcred3.sql.
CREATE OR REPLACE FUNCTION CountCredits (
  /* Возвращает число зачетов, на которое зарегистрирован студент,
     указанный в p_StudentID. */
  p_StudentID IN students.ID%TYPE)
  RETURN NUMBER AS

  v_TotalCredits NUMBER;  -- Общее число зачетов.
  v_CourseCredits NUMBER; -- Зачеты для одного курса обучения.
CURSOR c_RegisteredCourses IS
  SELECT department, course
     FROM registered_students
     WHERE student_id = p_StudentID;
BEGIN
  Debug.Reset;
  Debug.Debug('Before loop, v_TotalCredits', v_TotalCredits);
  FOR v_CourseRec IN c_RegisteredCourses LOOP
     -- Определим число зачетов для данной группы.
     SELECT num_credits
        INTO v_CourseCredits
        FROM classes
        WHERE department = v_CourseRec.department
        AND course = v_CourseRec.course;

     Debug.Debug('Inside loop, v_CourseCredits', v_CourseCredits);
     -- Добавим полученное число к тому, что было раньше.
     v_TotalCredits := v_TotalCredits + v_CourseCredits;
     Debug.Debug('Inside loop, v_TotalCredits', v_TotalCredits);
```

```
    END LOOP;

    Debug.Debug('After loop, returning', v_TotalCredits);
    RETURN v_TotalCredits;
END CountCredits;
```

Результат выполнения последнего варианта CountCredits таков:

```
SQL> exec :v_Total := CountCredits(10006);
Before loop, v_TotalCredits:
Inside loop, v_CourseCredits: 4
Inside loop, v_TotalCredits:
Inside loop, v_CourseCredits: 3
Inside loop, v_TotalCredits:
After loop, returning:

PL/SQL procedure successfully completed.
```

Проанализируем полученный результат. Обратите внимание: в **v_TotalCredits** содержится NULL-значение как до начала цикла, так и во время его выполнения. Дело в том, что в данном описании переменная **v_TotalCredits** не была инициализирована. В последнем варианте **CountCredits** инициализируем эту переменную, а также уберем отладочные операторы:

```
-- Этот пример содержится в файле cntcred4.sql.
CREATE OR REPLACE FUNCTION CountCredits (
  /* Возвращает число зачетов, на которое зарегистрирован студент,
     указанный в p_StudentID. */
  p_StudentID IN students.ID%TYPE)
  RETURN NUMBER AS

  v_TotalCredits NUMBER := 0; -- Общее число зачетов
  v_CourseCredits NUMBER;     -- Зачеты для одного курса обучения
  CURSOR c_RegisteredCourses IS
      SELECT department, course
        FROM registered_students
        WHERE student_id = p_StudentID;
BEGIN
  FOR v_CourseRec IN c_RegisteredCourses LOOP
      -- Определим число зачетов для данной группы.
      SELECT num_credits
        INTO v_CourseCredits
        FROM classes
        WHERE department = v_CourseRec.department
        AND course = v_CourseRec.course;

      -- Добавим полученное число к тому, что было раньше.
      v_TotalCredits := v_TotalCredits + v_CourseCredits;
  END LOOP;

  RETURN v_TotalCredits;
END CountCredits;
```

Получим следующий результат:

```
SQL> exec :v_Total := CountCredits (10006);
PL/SQL procedure successfully completed.
```

```
SQL> print v_Total
  V_TOTAL
-----------
          7
SQL> SELECT ID, CountCredits(ID) "Total Credits"
  2    FROM students;

       ID   Total Credits
--------   -------------------
    10000                8
    10001                4
    10002                8
    10003                8
    10004                4
    10005                4
    10006                7
    10007                4
    10008                8
    10009                7
    10010                8
11 rows selected.
```

Теперь функция **CountCredits** работает правильно как для одного студента, так и для всей таблицы. Если переменная при объявлении не инициализируется, ей присваивается NULL-значение. Оно проходит через все операции, предусмотренные в функции, и обрабатывается в соответствии с правилами вычисления NULL-выражений (см. главу 2).

## Ситуация 2: комментарии

Некоторые характеристики данного варианта модуля Debug отличны от характеристик его первого варианта. В частности, устранена зависимость от таблицы **debug_table**, что дает ряд преимуществ:

- Возможность одновременной работы нескольких соединений с базой даных, не мешающих друг другу; у каждого соединения будет свой собственный внутренний буфер DBMS_OUTPUT.
- В **Debug.Debug** больше не требуется выполнять оператор COMMIT.
- Пока разрешен (установлен в ON) SERVEROUTPUT, для вывода результатов на экран не нужно задавать дополнительные операторы SELECT. Кроме того, можно запретить отладку, просто отменив SERVEROUTPUT (установив для него OFF).

С другой стороны, при работе с этим вариантом следует учитывать некоторые моменты:

- Если не используются SQL*Plus, SQL*DBA или Server Manager, результаты отладки не будут автоматически выводиться на экран. Модуль будет работать и в других средах выполнения программ PL/SQL (например, в Pro*C или Oracle Forms), однако при этом, возможно, придется явно вызывать процедуру DBMS_OUTPUT.GET_LINE или DBMS_OUTPUT.GET_LINES и отображать результаты вручную.
- Объем выводимой отладочной информации ограничен размером буфера DBMS_OUTPUT. Если информации слишком много, а буфер недостаточно велик, то, скорее всего, лучшим решением будет использование первого варианта модуля.

# Отладчики PL/SQL

В состав ряда программных пакетов, предназначенных для разработки программ PL/SQL, входят отладчики. Это очень полезное средство, позволяющее выполнять программы PL/SQL в пошаговом режиме, строка за строкой, и выявлять причины ошибок. В этом разделе рассматриваются два таких средства: Procedure Builder корпорации Oracle и SQL-Station корпорации Platinum.

## Procedure Builder

Procedure Builder (построитель процедур) – это среда разработки программ PL/SQL, которая входит в состав программного пакета Developer 2000. Procedure Builder дает возможность выполнять программы PL/SQL в пошаговом режиме, анализируя при этом содержимое переменных. Как и при работе с отладчиками, применяемыми в других языках программирования, в Procedure Builder можно устанавливать *точки прерывания* (breakpoints) и изменять значения переменных на этапе выполнения программы. Procedure Builder был рассмотрен в главе 13, а в этом разделе будут затронуты те его возможности, которые имеют непосредственное отношение к отладке программ. В данном разделе модуль Debug создаваться не будет, так как в самой среде Procedure Builder имеются нужные средства, позволяющие осуществлять отладку программ.

▼ ВНИМАНИЕ

При работе с Procedure Builder программы PL/SQL можно отлаживать только на станции клиента, а не на сервере. Поэтому для отладки некоторого объекта нужно сначала перенести его на станцию клиента (с помощью Object Navigator). Скажем, процедуру можно отредактировать на станции клиента, а затем скопировать обратно на сервер.

## Ситуация 3

Рассмотрим следующую процедуру:

```
-- Этот пример содержится в файле crloop1.sql.
CREATE OR REPLACE PROCEDURE CreditLoop AS
  /* Введем идентификационный номер и значение, соответствующие
     текущему числу зачетов каждого студента, в таблицу temp_table. */
  v_StudentID students.ID%TYPE;
  v_Credits students.current_credits%TYPE;
  CURSOR c_Students IS
     SELECT ID
       FROM students;
BEGIN
  OPEN c_Students;
  LOOP
     FETCH c_Students INTO v_StudentID;
     v_Credits := CountCredits(v_StudentID);
     INSERT INTO temp_table (num_col, char_col)
       VALUES (v_StudentID, 'Credits = ' || TO_CHAR(v_Credits));
     EXIT WHEN c_Students%NOTFOUND;
  END LOOP;
  CLOSE c_Students;
END CreditLoop;
```

Процедура **CreditLoop** просто записывает число зачетов каждого студента в таблицу **temp_table**. Если в SQL*Plus запустить **CreditLoop** и обратиться с запросом к таблице **temp_table**, будет получен такой результат:

```
SQL> exec CreditLoop;
PL/SQL procedure successfully completed.
SQL> SELECT * FROM temp_table
  2    ORDER BY num_col;

   NUM_COL  CHAR_COL
---------- -----------------
     10000  Credits = 8
     10001  Credits = 4
     10002  Credits = 8
     10003  Credits = 8
     10004  Credits = 4
```

```
10005 Credits = 4
10006 Credits = 7
10007 Credits = 4
10008 Credits = 8
10009 Credits = 7
10010 Credits = 8
10010 Credits = 8
12 rows selected.
```

Проблема заключается в том, что последняя строка вводится дважды, т.е. для студента с идентификатором 10010 введено две строки, а для всех остальных — по одной.

## Ситуация 3: отладка с помощью Procedure Builder

В первую очередь нужно скопировать процедуру **CreditLoop** в клиентскую систему поддержки PL/SQL (в Procedure Builder можно вызывать серверные хранимые процедуры, но отлаживать их нельзя). Для этого в окне Object Navigator переместим процедуру из списка Stored Program Units в список Program Units, как показано на рис. 14.2.

**Рис. 14.2.**
*Копирование CreditLoop на станцию клиента*

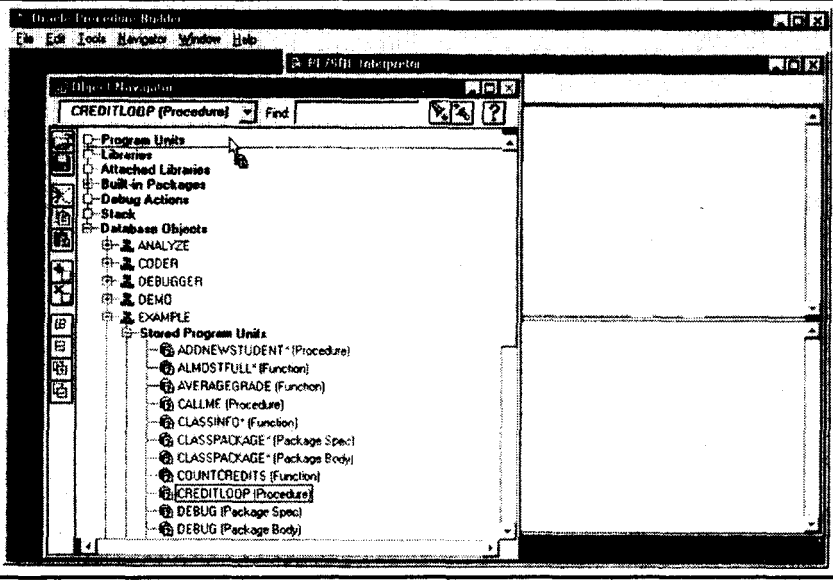



**Рис. 14.3.**
*Установка точки прерывания*

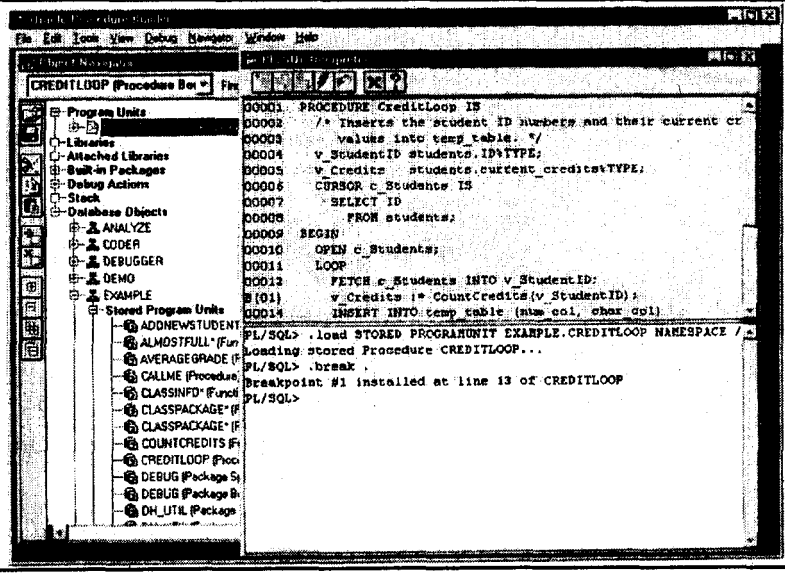
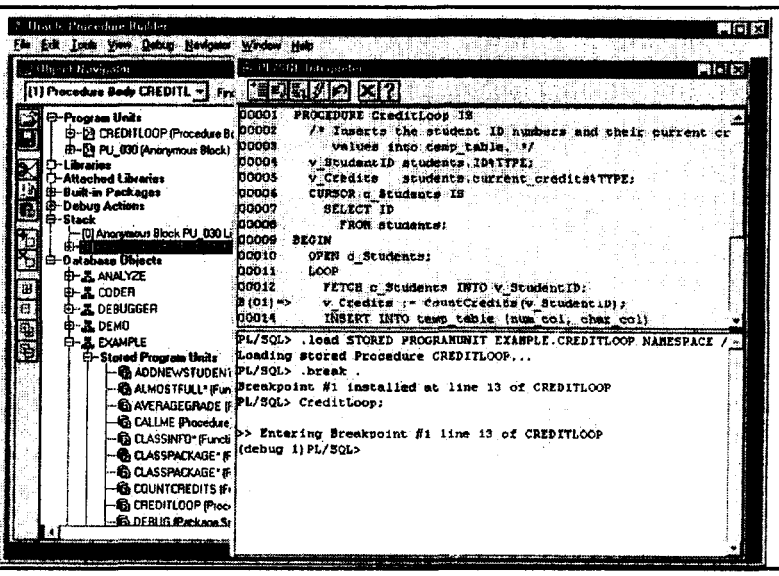


**Рис. 14.4.**
*Остановка в точке прерывания*




Когда **CreditLoop** оказывается на станции клиента, можно просмотреть текст этой процедуры в окне PL/SQL Interpreter. С помощью именно этого окна осуществляется управление процессом обработки процедуры. Сначала установим точку прерывания. После запуска процедуры на выполнение в этой точке оно будет остановлено и можно будет проанализировать значения, содержащиеся в локальных переменных, в окне Object Navigator. Для установки точки прерывания нужно дважды щелкнуть мышью на номере требуемой строки. Для **CreditLoop** это строка 13, следующая сразу же за оператором FETCH. Проанализируем содержание переменной **v_StudentID** и посмотрим, откуда поступает повторяющееся значение. Окно интерпретатора с установленной точкой прерывания показано на рис. 14.3.

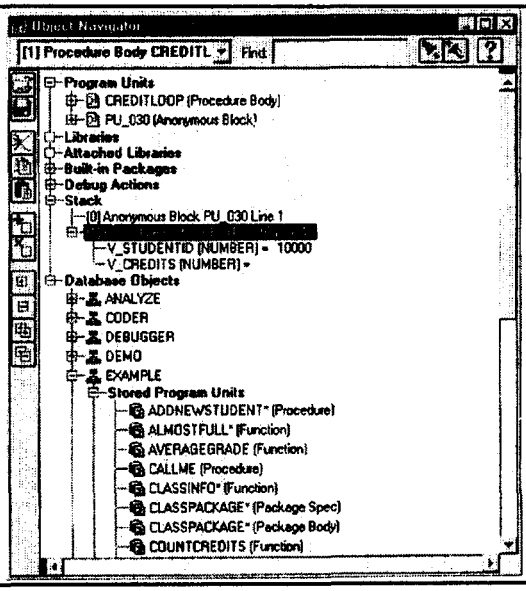
Запустим процедуру **CreditLoop** на выполнение:

```
CreditLoop;
```

в подсказке PL/SQL, предлагаемой интерпретатором. После запуска процедура остановится в указанной точке. Эта ситуация демонстрируется на рис. 14.4.

В этой точке предлагается несколько вариантов дальнейших действий. Можно продолжить выполнение процедуры с помощью кнопок, расположенных вверху экрана интерпретатора. Первые три кнопки позволяют продолжить выполнение в пошаговом режиме (точнее, в разных пошаговых режимах), а четвертая (с зигзагообразной стрелкой) — выполнить процедуру до ее полного завершения. Кроме того, можно просмотреть значения локальных переменных, обратившись к разделу Stack в окне Object Navigator. На рис. 14.5 показаны значения двух локальных переменных, видимых в данной точке преры-

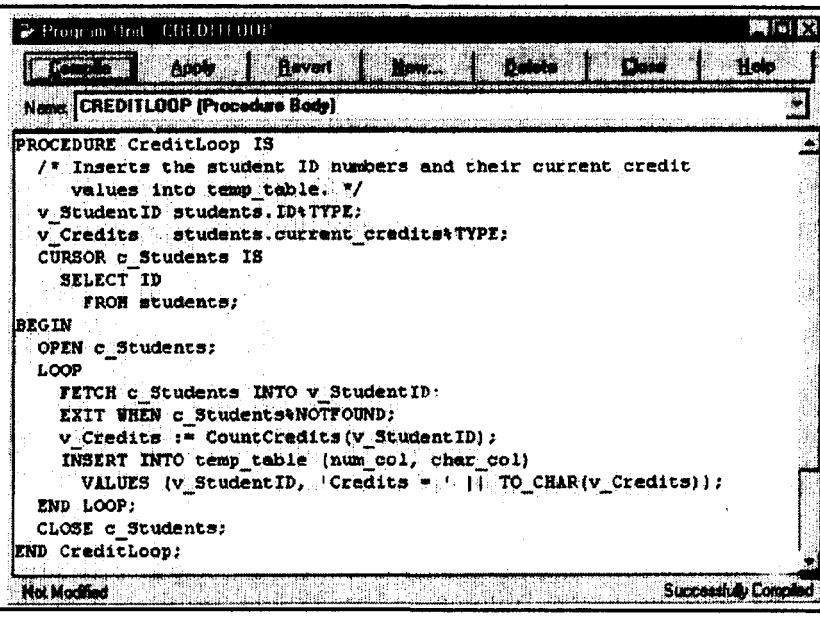
**Рис. 14.5.**
*Исследование локальных переменных в Object Navigator*



вания: **v_StudentID** и **v_Credits**. Видно, что в **v_StudentID** находится первый идентификатор, полученный из оператора FETCH, а в **v_Credits** – NULL. Именно этого и следовало ожидать.

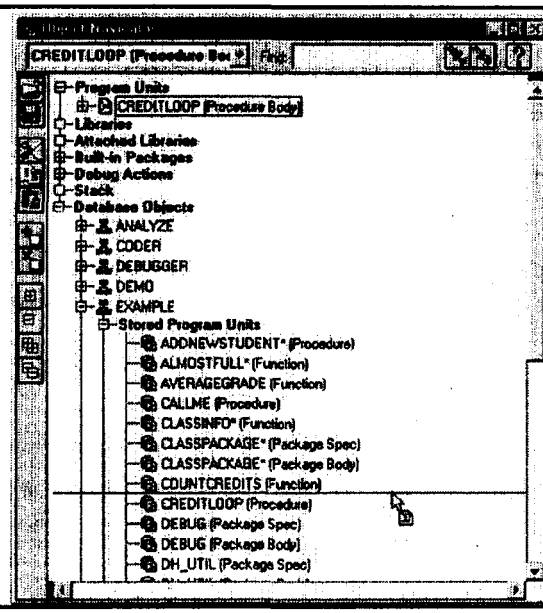
Теперь начнем выполнять процедуру в пошаговом режиме и просматривать **v_StudentID** после каждого оператора FETCH. Видно, что **v_StudentID** изменяется так, как и должно быть. Это продолжается до последнего оператора FETCH, возвращающего идентификатор 10010. Это значение вводится в таблицу **temp_table**, и цикл повторяется снова. Следующий оператор FETCH не изменяет значение **v_StudentID** – оно остается равным 10010, т.е. вводится дважды. После второй операции ввода цикл заканчивается, так как значение **c_Students%NOTFOUND** становится истинным. Таким образом, причина ошибки заключается в том, что оператор EXIT не находится сразу же после оператора FETCH. Модифицируем процедуру **CreditLoop** в окне редактора Program Unit Editor и протестируем ее. Правильный вариант процедуры показан на рис. 14.6.

**Рис. 14.6.**
*Правильный вариант CreditLoop*




Теперь, когда дефект устранен, нужно скопировать **CreditLoop** обратно на сервер, чтобы другие соединения также могли вызывать эту процедуру. Перенесем CreditLoop в окне Object Navigator из раздела Program Units в раздел Stored Program Units, как показано на рисунке 14.7. После этого можно снова протестировать процедуру с помощью SQL*Plus. Получаются следующие результаты:

**Рис. 14.7.**
*Копирование CreditLoop обратно на сервер*



```
SQL> exec CreditLoop
PL/SQL procedure successfully completed.

SQL> SELECT * FROM temp_table
  2   ORDER BY num_col;

   NUM_COL  CHAR_COL
---------  --------------
     10000  Credits = 8
     10001  Credits = 4
     10002  Credits = 8
     10003  Credits = 8
     10004  Credits = 4
     10005  Credits = 4
     10006  Credits = 7
     10007  Credits = 4
     10008  Credits = 8
     10009  Credits = 7
     10010  Credits = 8
11 rows selected.
```

### Ситуация 3: комментарии

Применение Procedure Builder дает ряд преимуществ:

- Не нужно включать в процедуру какие-либо отладочные операции, так как она выполняется в среде, управляющей отладкой.
- Отладка программы достаточно удобна, так как процедуру не нужно изменять, затем перекомпилировать и только потом анализировать содержимое различных переменных.
- В Procedure Builder предлагается интегрированная среда разработки программ, в состав которой входят PL/SQL Editor и Object Navigator. Для разработки программ PL/SQL другие средства не требуются.

Тем не менее при работе с Procedure Builder существуют определенные неудобства. Основным является то, что в текущей версии Procedure Builder (1.5) можно отлаживать программы только в клиентской системе PL/SQL. На станциях клиентов до сих пор используется PL/SQL версии 1.0 в отличие от сереверного PL/SQL версии 2.0 и выше. Это значит, что программы, написанные на PL/SQL версии 2, например те, в которых применяются таблицы PL/SQL и записи, определяемые пользователями, отлаживать нельзя.

## SQL-Station

Как и в Procedure Builder, в SQL-Station поддерживается интегрированная среда разработки программ PL/SQL, с редактором и отладчиком. Однако в SQL-Station можно отлаживать хранимые процедуры на сервере, не копируя их на станцию клиента. В SQL-Station фактически нет клиентской системы поддержки PL/SQL, поэтому все программы PL/SQL выполняются на сервере.

В главе 13 уже говорилось, что SQL-Station состоит из трех компонентов: Coder (кодировщика), Debugger (отладчика) и Plan Analyzer (анализатора планов). В этом разделе рассматривается отладчик. Для отладки подпрограммы, написанной на PL/SQL, SQL-Station автоматически создает ее специальный отладочный вариант — копию исходной подпрограммы с дополнительным программным кодом, позволяющим SQL-Station следить за состоянием процесса отладки; такой вариант создается при отладке каждый раз.

Отладочные варианты всегда начинаются с символов X#, за которыми следуют порядковый номер и имя объекта. При желании можно установить режим Clear debug objects after session (удаление отладочных объектов после окончания сеанса работы), и все эти символы будут автоматически удаляться при выходе из SQL-Station. Можно также удалить их явно с помощью позиции меню Maintenance. За более подробной информацией о работе с отладочными вариантами обращайтесь к встроенной документации по SQL-Station.

## Ситуация 4

Очень часто причинами ошибок, возникающих в программах PL/SQL, являются не сами программы, а данные, с которыми они работают. Для примера рассмотрим SQL-сценарий, с помощью которого данные копируются из таблицы **source** (источник) в таблицу **destination** (место назначения):

```
-- Этот пример содержится в файле copytab1.sql.
CREATE OR REPLACE PROCEDURE CopyTables AS
  V_Key    source.key%TYPE;
  V_Value  source.value%TYPE;

  CURSOR c_AllData IS
      SELECT *
        FROM source;
BEGIN
  OPEN c_AllData;

  LOOP
      FETCH c_AllData INTO v_Key, v_Value;
      EXIT WHEN c_AllData%NOTFOUND;

      INSERT INTO destination (key, value)
        VALUES (v_Key, TO_NUMBER(v_Value));
  END LOOP;

  CLOSE c_AllData;
END CopyTables;
```

Таблицы **source** и **destination** создаются следующим образом:

```
-- Этот пример является частью файла tables.sql.
CREATE TABLE source (
  key NUMBER(5),
  value VARCHAR2(50) );

CREATE TABLE destination (
  key NUMBER(5),
  value NUMBER);
```

Обратите внимание, что столбец **value** (значение) таблицы **source** имеет тип VARCHAR2, однако тип столбца **value** таблицы **destination** – NUMBER. Заполним таблицу source информацией с помощью приведенного ниже блока PL/SQL, вводящего в эту таблицу 500 строк. 499 вводимых строк представляют собой правильные последовательности символов (которые могут быть преобразованы к типу NUMBER), а в одной строке (выбираемой случайным образом при помощи модуля **Random**, который описан в главе 8) содержится неверное значение.

```
-- Этот пример содержится в файле populate.sql.
DECLARE
  v_RandomKey source.key%TYPE;
BEGIN
  -- Сначала заполним таблицу source верными значениями.
  FOR v_Key IN 1..500 LOOP
      INSERT INTO source (key, value)
        VALUES (v_Key, TO_CHAR(v_Key));
  END LOOP;

  -- Теперь выберем случайное число в диапазоне от 1 до 500 и
  -- введем в строку с таким номером неверное значение.
```

```
    v_RandomKey := Random.RandMax(500);
    UPDATE source
        SET value = 'Oops, not a number!'
        WHERE key = v_RandomKey;

    COMMIT;
END;
```

Если вызвать процедуру **CopyTables,** будет выдано сообщение об ошибке ORA-1722:

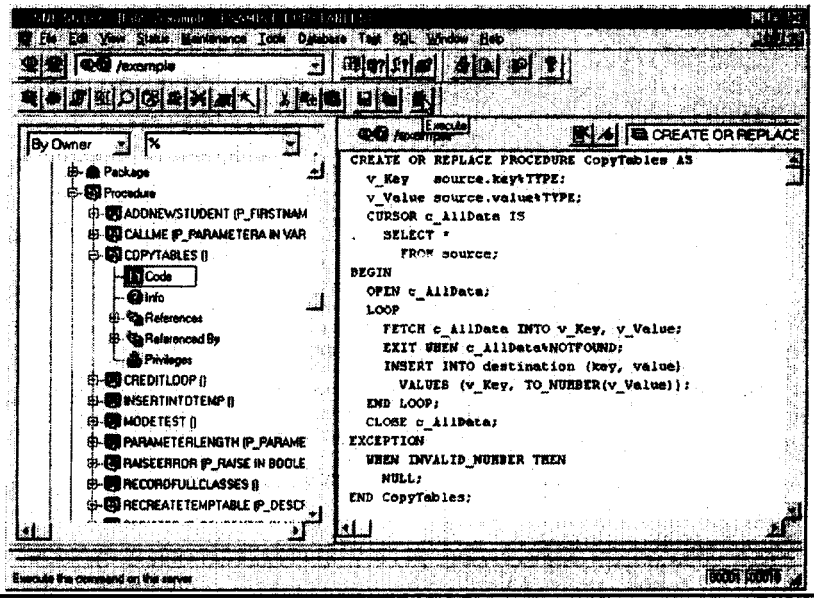
```
SQL> exec CopyTables
begin CopyTables; end;
*
ERROR at line 1:
ORA-01722: invalid number
ORA-06512: at "EXAMPLE.COPYTABLES", line 12
ORA-06512: at line 1
```

Понятно, что ошибка возникает при выполнении оператора INSERT. Вопрос в том, какое именно значение неверно. Чтобы ответить на этот вопрос, воспользуемся программой-отладчиком.

## Ситуация 4: Отладчик SQL-Station

В первую очередь необходимо создать в **CopyTables** обработчик исключительной ситуации. Тогда в нем можно будет установить точку прерывания и исследовать **v_Key** в поисках строки, содержащей ошибку. На рис. 14.8 продемонстрирован измененный вариант процедуры **CopyTables** в окне редактирования кодировщика. Можно сохранить этот вариант, щелкнув мышью на кнопке Execute, как показано на рисунке.

**Рис. 14.8.**
*Изменение CopyTables*




После сохранения этого варианта на сервере откроем для **CopyTables** окно отладки, выбрав Debug в меню Tools или щелкнув мышью на Debug в панели инструментов. SQL-Station создаст отладочный вариант **CopyTables** и инициализирует среду отладки (рис. 14.9). Затем следует установить точку прерывания в обработчике исключительной ситуации, щелкнув мышью на кнопке Set Breakpoint в тот момент, когда текст обработчика выделен (рис. 14.10).

Теперь можно запустить процедуру на выполнение. Она будет выполняться до контрольной точки в обработчике исключительной ситуации. В этот момент содержимое локальных переменных показывается в нижней области окна отладки. Можно видеть, что ключ строки, являющейся причиной ошибки, имеет значение 409 (рис. 14.11).

**Рис. 14.9.**
*CopyTables в окне отладки*

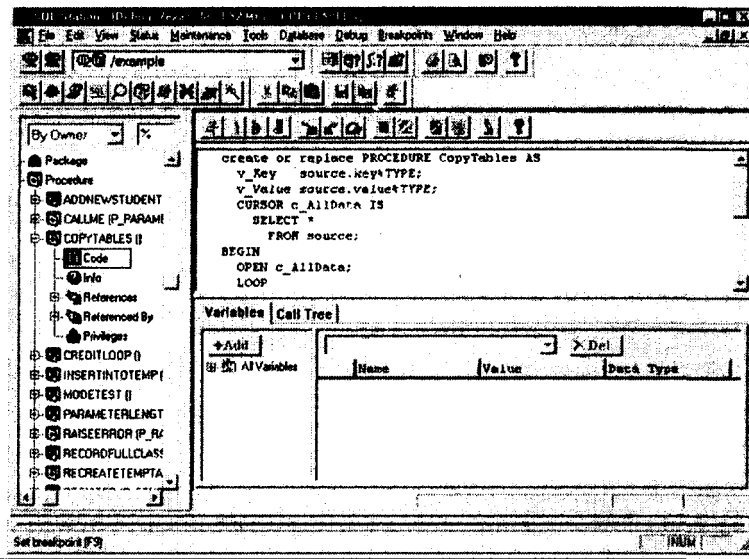
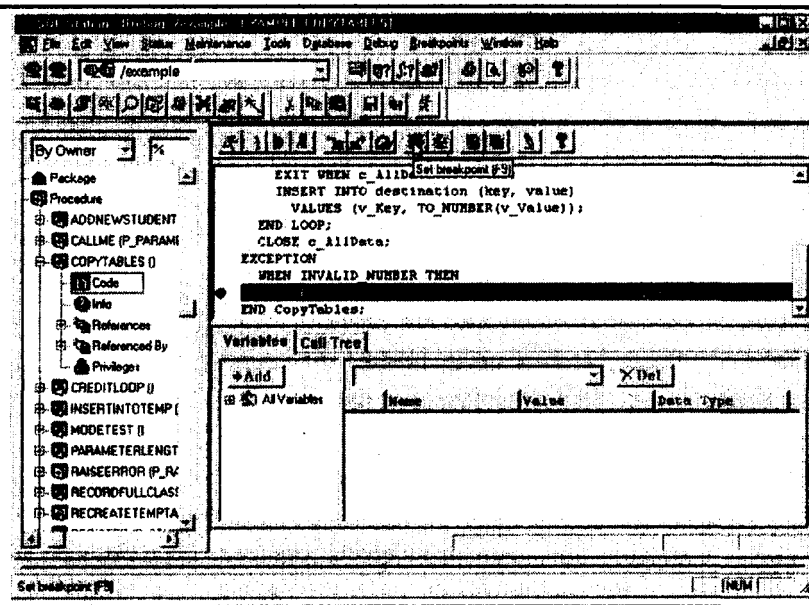



**Рис. 14.10.**
*Установка точки прерывания*

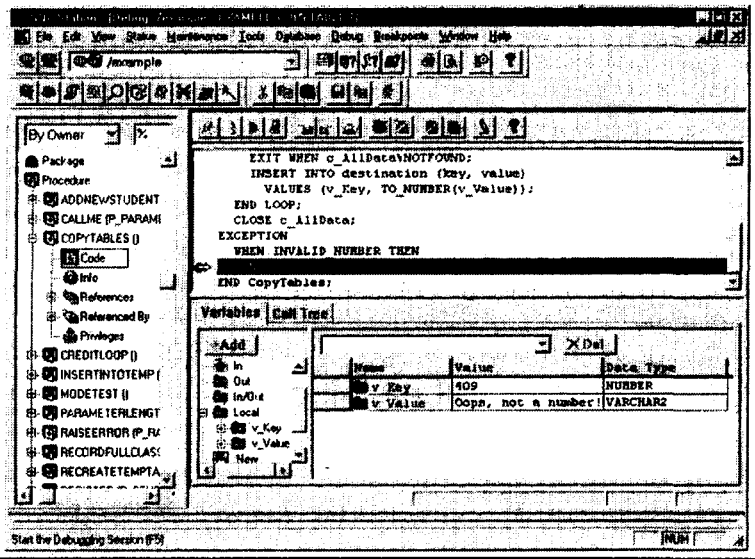



## Ситуация 4: комментарии

Выше был рассмотрен достаточно простой пример, однако, в зависимости от сложности используемого приложения, ошибочность данных может проявляться и в более запутанных и непонятных ситуациях. В нашем случае неверным был тип данных, и поэтому информацию нельзя было преобразовать. Может оказаться, например, что данные не попадают в указанный диапазон. Заметьте, что неверные данные не всегда приводят к установлению исключительных ситуаций; результат зависит от метода обработки ошибок и от характера несогласованности информации.

Важно отметить, что в рассмотренном примере подпрограмма была написана без ошибок. Причина ошибки заключалась не в программном коде, а в обрабатываемых данных. Поэтому вполне возможно, что неверные данные проще обнаружить, обратившись напрямую к таблице, а не отлаживая программу.

**Рис. 14.11.**
*Остановка в точке прерывания*




# Сравнение Procedure Builder и SQL-Station

Procedure Builder и SQL-Station являются эффективными программными средствами и обладают схожими возможностями. Оба продукта:

- Позволяют редактировать и отлаживать объекты PL/SQL в одной и той же среде.
- Обладают похожими возможностями отладки — устанавливать точки прерывания и анализировать (а также модифицировать) значения, содержащиеся в локальных переменных.
- Поддерживают навигацию объектов, позволяющую просматривать и модифицировать серверные (и клиентские — в случае Procedure Builder) объекты различных типов.
- Используют многооконную среду, в отличие от модуля Debug и других более простых методов отладки программ.

Однако у этих средств есть и различия:

- В Procedure Builder содержится клиентская система поддержки PL/SQL. Для отладки объекта необходимо сначала скопировать его на станцию клиента. SQL-Station же позволяет отлаживать серверные объекты на месте. Это удобно, например, при отладке триггеров;
- Procedure Builder может работать непосредственно с программным кодом PL/SQL, не создавая специальный отладочный вариант. SQL-Station же при отладке подпрограммы обязательно создает дополнительный вариант.

*Конечно, в данном разделе рассмотрены не все свойства этих программных продуктов. Если необходима дополнительная информация о Procedure Builder и/или SQL-Station, обратитесь к встроенной документации.*

# Методологии программирования

В этой главе был рассмотрен ряд различных методов отладки программ PL/SQL. Число возникающих в программах ошибок можно свести к минимуму, если придерживаться надежной стратегии программирования и использовать эффективные методологии проектирования программ. Предлагаемые концепции применимы не только к PL/SQL, но и к другим языкам программирования.

## Модульное программирование

*Модульным программированием* (modular programming) называется разбиение программы на отдельные фрагменты, или модули (modules; не путайте с программными модулями (packages). — *Прим. пер.*), каждый из которых выполняет конкретную функцию. После разбиения программы можно настроить и протестировать каждый модуль в отдельности, а затем скомпоновать их в единую итоговую программу.

Кроме того, отдельные модули можно повторно использовать в других программах. Проиллюстрируем вышесказанное на примере. Предположим, что нужно написать программу, отображающую различную информацию о студентах, – так называемые характеристики (transcripts). Разобьем эту задачу на несколько этапов и составим алгоритм работы программы:

1. Определим идентификатор того студента, характеристику которого нужно отобразить.
2. Определим учебные группы, в которых зарегистрирован данный студент.
3. Для каждой группы узнаем факультет, учебный курс, описание курса и оценку студента.
4. Определим среднюю оценку успеваемости (GPA – grade point ave\rage) студента и выведем ее.

Каждый из этих этапов можно выполнить с помощью отдельной программной конструкции PL/SQL, а затем объединить все конструкции в итоговую программу. Например:

1. Передадим идентификатор студента процедуре в качестве параметра.
2. Узнаем, в каких группах зарегистрирован данный студент, из таблицы **registered_students**.
3. Получим сведения о каждой группе из таблицы **classes.**
4. Вычислим оценку GPA для студента с помощью другой процедуры – **CalculateGPA**.

Определив общую структуру процедуры, можно ее зафиксировать:

```
CREATE OR REPLACE PROCEDURE PrintTranscript (
  /* Выводит характеристику данного студента. В ней указываются группы,
     в которых зарегистрирован студент, и его оценки, полученные
     в каждой группе. В конце характеристики выводится оценка GPA. */
  p_StudentID IN students.ID%TYPE) AS

  v_StudentGPA NUMBER; – Средняя оценка успеваемости студента.
  CURSOR CurrentClasses IS
      SELECT *
        FROM registered_students
        WHERE student_id = p_StudentID;
BEGIN
  -- Выведем общую информацию о студенте: имя, фамилию,
  -- профилирующую дисциплину и т.д.

  FOR v_ClassesRecord IN CurrentClasses LOOP
      -- Выведем информацию о каждой группе.
      NULL;
  END LOOP;

  -- Вычислим оценку GPA.
  CalculateGPA (p_StudentID, v_StudentGPA);

  -- Выведем оценку GPA.
END PrintTranscript;
```

Итак, создана структура процедуры **PrintTranscript**. Теперь можно по очереди разрабатывать отдельные компоненты процедуры, убеждаясь в правильности предыдущего компонента перед началом работы над следующим. Это будет выполнено в главе 18.

## Нисходящее проектирование

*Нисходящее проектирование* (top-down design) дополняет модульное программирование. При использовании этого метода сначала создается оболочка программы, а затем разрабатываются все детали. Перед началом разработки можно точно указать, для чего предназначена каждая подпрограмма. В PL/SQL это можно сделать с помощью процедур-заглушек. Обратимся к процедуре **CalculateGPA**, которая нужна для функционирования **PrintTranscript**. Создадим процедуру-заглушку, не разрабатывая реального алгоритма для **CalculateGPA**:

```
CREATE OR REPLACE PROCEDURE CalculateGPA (
  /* Записывает в p_GPA среднюю оценку успеваемости студента,
     указанного в p_StudentID. */
  p_StudentID IN students.ID%TYPE,
  p_GPA OUT NUMBER) AS
BEGIN
  NULL;
END CalculateGPA;
```

Эту процедуру можно скомпилировать, что позволит скомпилировать и заготовку для **PrintTranscript**. Оператор NULL служит в данном случае вместилищем реального программного кода процедуры. Создав вначале процедуру-заглушку, можно затем продолжить разработку процедуры **PrintTranscript**, не заботясь о деталях **CalculateGPA**.

Таким образом, разработка программы начинается с самого верхнего уровня, а заканчивается деталями. Этот принцип и является основной концепцией нисходящего проектирования. Если же выбрать метод *восходящего проектирования* (bottom-up design), **CalculateGPA** будет создаваться прежде, чем **PrintTranscript**. Нисходящее проектирование гибче, чем восходящее. Например, при разработке **PrintTranscript** может оказаться, что **CalculateGPA** будет лучше работать как функция или что для этой процедуры необходимы какие-то дополнительные параметры. Можно просто внести изменения в процедуру-заглушку **CalculateGPA** и продолжить разработку **PrintTranscript**, а завершить процесс создания **CalculateGPA** позже, после того как она будет настроена надлежащим образом.

## Абстрактное представление данных

*Абстрактное представление данных* (data abstraction) – еще один эффективный метод программирования. Оно позволяет скрыть некоторые детали реализации конкретного алгоритма, оставив лишь интерфейс, т.е. средство доступа к нему. Процедура **CalculateGPA** является хорошим примером абстрактного представления. Процедуре **PrintTranscript** незачем знать, что представляет из себя **CalculateGPA**. Пользователь вправе как угодно изменять текст **CalculateGPA**, не изменяя форму вызова этой процедуры.

Объекты, содержащиеся в теле модуля и являющиеся его частными объектами, также могут быть использованы для реализации абстрактного представления данных. В заголовке модуля описывается внешний интерфейс для данных, а работа с ними реально выполняется в теле этого модуля. В модуле DBMS_OUTPUT используется именно такой метод. Интерфейс связи, например, с GET_LINES описан при объявлении этой процедуры и типа CHARARR в заголовке модуля. В теле процедуры создан цикл, в котором просматривается буфер, а содержимое буфера возвращается в параметре строки. Однако, как выполняется это процесс и даже как устроен буфер, неизвестно. Чтобы работать с модулем DBMS_OUTPUT, не нужно знать, каково внутреннее устройство буфера (он может быть таблицей PL/SQL, таблицей базы данных или любой другой структурой).

**PL/SQL 8.0 ... и ВЫШЕ**

В PL/SQL версии 8 объектные типы также реализованы через абстрактное представление данных. Методы и атрибуты в спецификации типа описывают общий интерфейс для работы с ним, а реализуется такой интерфейс в теле типа. Можно как угодно изменять тело типа, не изменяя интерфейс связи с данным типом (более подробно об этом см. главу 12).

# Итоги

В этой главе были рассмотрены три различных способа отладки программ PL/SQL: ввод данных в выходную таблицу, использование модуля DBMS_OUTPUT, а также использование отладчиков PL/SQL – Procedure Builder и SQL-Station. Следует выбирать тот метод, который лучше всего отвечает требованиям, выдвигаемым при создании приложений, а также подходит для вашей рабочей среды. При анализе этих методов были разобраны четыре ошибочных ситуации, типичных при работе с PL/SQL, а также приведены примеры устранения возможных ошибок, таких как некорректное сравнение символов, использование неинициализированных переменных, неправильное условие выхода из цикла и неверные данные. Кроме того, были кратко описаны эффективные методологии программирования, применимые при разработке различных программ, а не только написанных на PL/SQL.

# Глава 15

## Динамический PL/SQL

В главе 2 было сказано о том, что в PL/SQL применяется ранняя привязка переменных. Поэтому в программах, написанных на PL/SQL, могут содержаться только операторы DML – операторы DDL запрещены. В PL/SQL версии 2.1 и выше это ограничение устраняется благодаря использованию модуля DBMS_SQL. С его помощью реализуются динамические свойства SQL и PL/SQL, т.е. программные конструкции этих языков динамически вызываются из других блоков PL/SQL. В этой главе говорится о применении и назначении модуля DBMS_SQL. Кроме того, дается анализ новых возможностей DBMS_SQL, доступных в PL/SQL 8.0, в том числе характеристики процесса обработки массивов. Завершается глава сравнением DBMS_SQL с динамическими средствами, применяемыми в других программных продуктах Oracle: в предкомпиляторах и OCI.

# Введение

PL/SQL 2.1 ... и ВЫШЕ

Модуль DBMS_SQL применяется в PL/SQL версии 2.1 (Oracle7, релиз 7.1) и выше. По существу, DBMS_SQL делает доступным пользователю обычный процесс выполнения конструкций SQL и PL/SQL и дает ему возможность управлять этим процессом. Поскольку всем управляет программист, модуль DBMS_SQL можно применять по отношению как к операторам DDL, так и к операторам DML. Чтобы понять основные принципы работы DBMS_SQL, следует выяснить, в чем же различие между статическим и динамическим SQL.

## Статический и динамический SQL

Все приведенные выше программы PL/SQL являются статическими. Это означает, что структура операторов известна уже во время компиляции программы. Для примера рассмотрим следующий блок PL/SQL:

```
DECLARE
  v_Department classes.department%TYPE := 'ECN';
  v_NumCredits classes.num_credits%TYPE := 5;
BEGIN
  UPDATE classes
      SET num_credits = v_NumCredits
      WHERE department = v_Department;
END;
```

Оператор UPDATE этого блока является примером статического SQL-оператора. При компиляции блока известно, что данный оператор – это оператор обновления данных, что в нем происходит ссылка на столбцы **num_credits** и **department** таблицы **classes**, а также известно, что задано в условии WHERE. Теперь рассмотрим такой блок:

```
DECLARE
  v_SQLString VARCHAR2(100);
  v_SetClause VARCHAR2(100);
  v_WhereClause VARCHAR2(100);
BEGIN
  v_SetClause := 'SET num_credits = :num_credits WHERE ';
  v_WhereClause := 'department = :department';
  v_SQLString := 'UPDATE classes ' || v_SetClause ||
  v_WhereClause;
  DoIt(v_SQLString);
END;
```

*Это блок нельзя запустить на выполнение сразу же после его написания, так как предварительно нужно создать процедуру* **DoIt** с помощью DBMS_SQL, чтобы в ней выполнялись *динамические* SQL-операторы. Для таких операторов ограничений меньше – многое из того, что необходимо знать во время компиляции, вовсе не обязательно описывать до этапа выполнения программы. Полное описание SQL-оператора можно задать во время его выполнения.

## Обзор модуля DBMS_SQL

Алгоритм выполнения оператора с помощью DBMS_SQL состоит из следующих шагов:

1. Преобразование SQL-оператора или блока PL/SQL в строку символов.
2. Выполнение грамматического разбора этой строки символов с помощью DBMS_SQL.PARSE.
3. Привязка всех входных переменных с помощью DBMS_SQL.BIND_VARIABLE.
4. Если оператор не является запросом – выполнение его с помощью DBMS_SQL.EXECUTE и/или DBMS_SQL.VARIABLE_VALUE. Если оператор является запросом – переход к шагу 5.
5. Если оператор является запросом – описание выходных переменных с помощью DBMS_SQL.DEFINE_COLUMN.
6. Выполнение запроса и считывание результатов с помощью DBMS_SQL.EXECUTE, DBMS_SQL.FETCH_ROWS, DBMS_SQL.COLUMN_VALUE и DBMS_SQL.VARIABLE_VALUE.

Рассмотрим в качестве примера процедуру **RecreateTempTable**, которая сначала удаляет, а затем повторно создает таблицу **temp_table**; описание таблицы передается процедуре как аргумент.

```
-- Этот пример содержится в файле. recrtemp.sql.
CREATE OR REPLACE PROCEDURE RecreateTempTable (
  /* Удаляет таблицу temp_table и повторно создает ее. Описаниетаблицы
     передается в процедуру с помощью p_Description и должно быть текстом
     оператора CREATE TABLE, расположенным после имени таблицы.
     Например, ниже приведен правильный вызов процедуры:
     RecreateTempTable('(num_col NUMBER, char_col VARCHAR2(2000))');
  */
  p_Description IN VARCHAR2) IS

  V_Cursor          NUMBER;
  V_CreateString    VARCHAR2(100);
  V_DropString      VARCHAR2(100);
  V_NumRows         INTEGER;
BEGIN
  /* Откроем курсор для обработки данных. */
  v_Cursor := DBMS_SQL.OPEN_CURSOR;

  /* Удалим таблицу. */
  v_DropString := 'DROP TABLE temp_table';

  /* Проведем грамматический разбор команды 'DROP TABLE' и выполним
     ее. Если таблица не существует, установим ошибку ORA-942. */
  BEGIN
     -- DBMS_SQL.V7 – константа, описанная в заголовке модуля.
     DBMS_SQL.PARSE(v_Cursor, v_DropString, DBMS_SQL.V7);
     v_NumRows := DBMS_SQL.EXECUTE(v_Cursor);
  EXCEPTION
     WHEN OTHERS THEN
        IF SQLCODE != -942 THEN
           RAISE;
        END IF;
  END;
  /* Теперь создадим таблицу. Сначала нужно создать строку символов
     CREATE TABLE, а затем провести ее грамматический разбор и
     обработать ее. */
  v_CreateString := 'CREATE TABLE temp_table ' || p_Description;
  DBMS_SQL.PARSE(v_Cursor, v_CreateString, DBMS_SQL.V7);
```

```
    v_NumRows := DBMS_SQL.EXECUTE(v_Cursor);
    /* Работа закончена, поэтому закроем курсор. */
    DBMS_SQL.CLOSE_CURSOR(v_Cursor);
EXCEPTION
  WHEN OTHERS THEN
      /* Сначала закроем курсор, затем переустановим ошибку так, чтобы
         передать ее вне блока. */
      DBMS_SQL.CLOSE_CURSOR(v_Cursor);
      RAISE;
END RecreateTempTable;
```

▼ ОСТОРОЖНО **Для выполнения этого примера необходимо иметь системные полномочия CREATE TABLE и DROP TABLE, предоставленные непосредственно, а не через роль. (Более подробно об этом см. в разделе "Привилегии и DBMS_SQL" ниже в данной главе.)**

При анализе приведенного примера обратите внимание на следующее:

- Строка символов, грамматический разбор которой выполняется, может быть константой (здесь **v_DropString**), а также может создаваться программой динамически, с помощью символьных функций, например функции конкатенации (здесь **v_CreateString**).
- Обработка ошибок осуществляется так же, как и в статическом SQL: ошибки устанавливаются и обрабатываются с помощью исключительных ситуаций (см. главу 10). Отличие состоит в том, что теперь во время выполнения блока могут возвращаться ошибки компиляции (например, ORA-942). В статическом PL/SQL такие ошибки обнаруживаются на этапе компиляции, еще до запуска блока на выполнение.
- Пользователю предоставляются большие возможности по управлению курсорами, в частности *определение моментов открытия и закрытия курсоров. Все открытые курсоры должны быть* закрыты, даже в случае установления исключительной ситуации. В данном примере закрытие курсора обеспечивается вызовом DBMS_SQL.CLOSE_CURSOR в обработчике второй исключительной ситуации.

На рис. 15.1 приведена блок-схема, иллюстрирующая порядок, в котором обычно вызывается DBMS_SQL.

С помощью DBMS_SQL можно обрабатывать операторы трех видов: операторы DML и DDL, запросы и анонимные блоки PL/SQL. Каждый из видов обрабатывается различными процедурами. Ниже приведено краткое описание этих процедур, а более детальное объяснение будет дано позже.

**OPEN_CURSOR** Как и в статическом SQL, в динамическом PL/SQL каждый SQL-оператор выполняется в границах курсора. Кроме того, в динамическом PL/SQL можно управлять процессом обработки курсора. OPEN_CURSOR возвращает идентификационный номер курсора, используемый для обозначения контекстной области, в которой будет выполняться оператор. Этот номер указывается во всех последующих вызовах данного курсора.

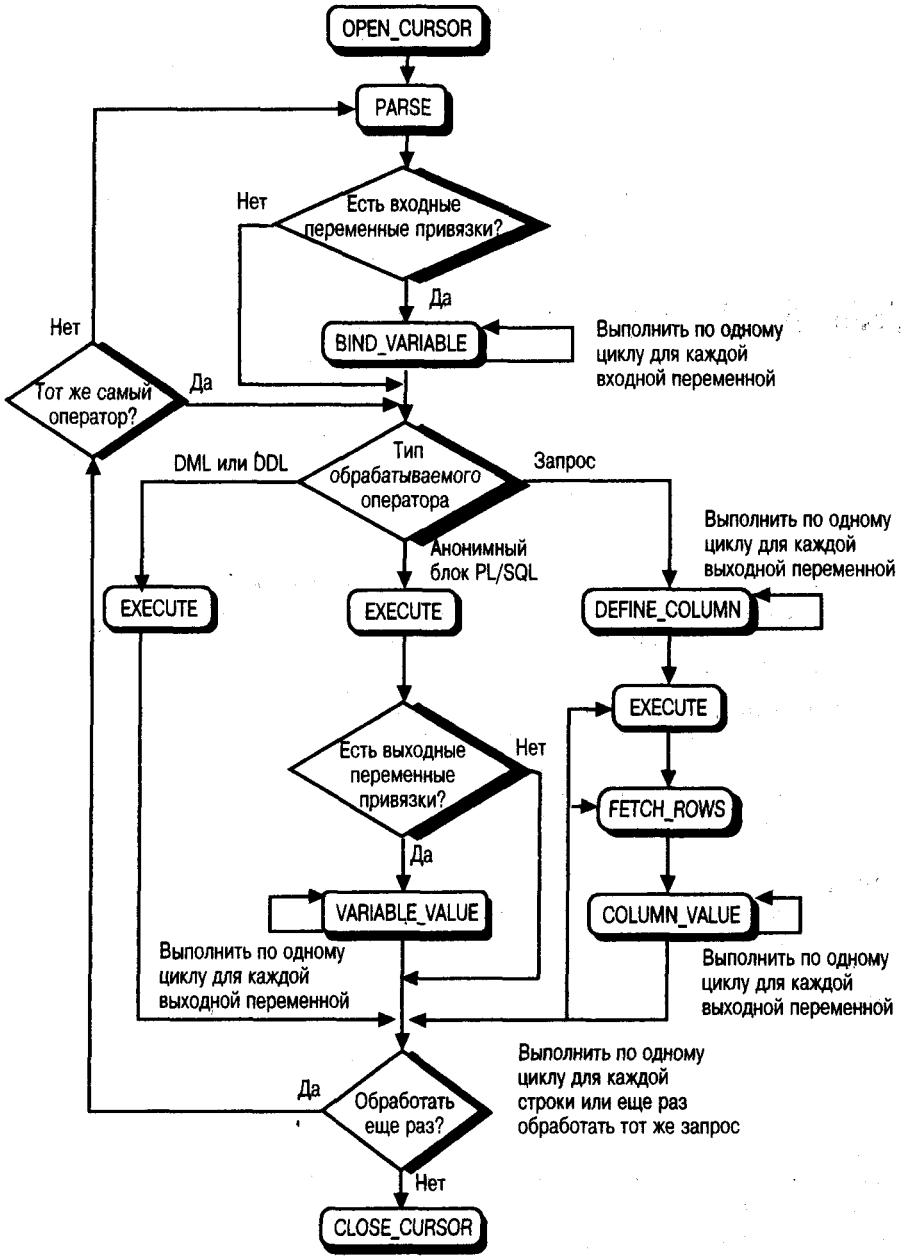
**PARSE** Грамматический разбор оператора заключается в пересылке его на сервер, где проверяются его синтаксис и семантика. Если оператор является запросом, то на этом этапе определяется и план его выполнения.

**BIND_VARIABLE** Привязка переменной к заполнителю аналогична процессу привязки, используемому в PL/SQL для статических SQL-операторов. Заполнитель (placeholder) – это специальный идентификатор в последовательности символов оператора. В процессе привязки устанавливается соответствие между заполнителем и фактической переменной, а модулю DBMS_SQL сообщается о ее типе и размере. Привязка выполняется для входных переменных.

**DEFINE_COLUMN** Определение выходной переменной аналогично привязке входной переменной. В данном случае выходными переменными являются результаты запроса. Процедура DEFINE_COLUMN определяет тип и размер переменных PL/SQL, в которые будут записываться данные при считывании информации процедурой FETCH_ROWS. DEFINE_COLUMN используется только для выходных переменных (результатов запросов), а BIND_VARIABLE – для входных переменных (заполнителей, указанных в операторах).

**EXECUTE** Эта функция исполняет операторы, не являющиеся запросами, и возвращает число обработанных строк. Для запросов же EXECUTE определяет активный набор. После этого данные считывают-

**Рис. 15.1.**
*Алгоритм обработки данных в DBMS_SQL*




ся процедурой FETCH_ROWS. Для любых операторов переменные привязки анализируются во время работы функции EXECUTE.

**FETCH_ROWS** С каждым вызовом процедуры FETCH_ROWS с сервера считывается все больший и больший объем данных. Затем полученная информация преобразуется к типам данных, указанным процедурой DEFINE_COLUMN. EXECUTE_AND_FETCH позволяет объединить операции обработки и считывания данных в одном вызове.

**VARIABLE_VALUE** Данная подпрограмма используется для определения значения переменной привязки, если эта переменная модифицируется оператором. VARIABLE_VALUE применяется только тогда, когда оператор является блоком PL/SQL (в котором может вызываться хранимая процедура).

**COLUMN_VALUE** После вызова FETCH_ROWS процедура COLUMN_VALUE применяется для фактического возврата данных. В ней используются переменные того же типа, который указан в процедуре DEFINE_COLUMN. COLUMN_VALUE применяется только для запросов.

**CLOSE_CURSOR** После окончания обработки данных курсор закрывается. При этом ресурсы, использовавшиеся курсором, освобождаются.

# Обработка операторов DML, не являющихся запросами, и операторов DDL

▼ ВНИМАНИЕ В этом и в последующих двух разделах описан DBMS_SQL для PL/SQL версий с 2.1 по 2.3. В PL/SQL 8.0 возможности ряда процедур расширены. Они обсуждаются в разделе "Новые возможности DBMS_SQL в PL/SQL 8.0" ниже.

В этом разделе подробно описаны шаги, выполняемые при обработке операторов INSERT, UPDATE, DELETE, а также операторов DDL. Обработка анонимных блоков PL/SQL рассмотрена в разделе "Обработка блоков PL/SQL", а обработка запросов – в разделе "Обработка запросов" этой главы.

Для обработки оператора DML, не являющегося запросом, или оператора DDL необходимо выполнить следующие шаги:

1. Открыть курсор (OPEN_CURSOR).
2. Выполнить грамматический разбор оператора (PARSE).
3. Выполнить привязку всех входных переменных (BIND_VARIABLE).
4. Обработать оператор (EXECUTE).
5. Закрыть курсор (CLOSE_CURSOR).

## Открытие курсора

Каждый SQL-оператор или блок PL/SQL (статический либо динамический) обрабатывается в границах некоторого курсора. Обработкой большинства статических SQL-операторов управляет система поддержки PL/SQL. Пользователь может управлять процессом обработки запросов с помощью команд OPEN и CLOSE (см. главу 6).

В этом смысле динамический SQL не составляет исключения. Каждый вызов процедуры OPEN_CURSOR возвращает целое число, являющееся идентификационным номером курсора. Этот номер используется в последующих вызовах курсора. В границах одного курсора можно по очереди обработать несколько SQL-операторов или выполнить один и тот же оператор несколько раз.

Для каждого вызова OPEN_CURSOR должен существовать соответствующий вызов CLOSE_CURSOR, чтобы ресурсы, используемые курсором, освобождались. Процедура без параметров OPEN_CURSOR описывается следующим образом:

```
OPEN_CURSOR RETURN INTEGER;
```

## Грамматический разбор оператора

Грамматический разбор оператора выполняется на сервере, который проверяет синтаксис и семантику оператора и возвращает ошибку (устанавливая исключительную ситуацию), если нарушены требования грамматики. Кроме того, во время разбора определяется план выполнения оператора. Аналогичный вызов в OCI – **oparse** – позволяет отложить проведение грамматического разбора оператора, записав его в буфер и сохраняя там до этапа обработки. Затем вместе с обработкой выполняется грамматический разбор, что снижает нагрузку на сеть. Однако в текущей версии DBMS_SQL отложенное выполнение грамматического разбора не поддерживается. Обычно это не приводит к каким-либо отрицательным последствиям, поскольку весь блок PL/SQL может быть обработан на сервере (например, в хранимой процедуре), поэтому передачи данных по сети часто не требуется.

Процедура PARSE описывается следующим образом:

```
PROCEDURE PARSE(c IN INTEGER,
                statement IN VARCHAR2,
                language_flag IN INTEGER);
```

Описание параметров процедуры PARSE приведено в таблице 15.1.

**ТАБЛИЦА 15.1.**

| Параметр | Тип | Описание |
|---|---|---|
| *c* | INTEGER | Идентификационный номер курсора, в котором выполняется грамматический разбор оператора. Курсор должен быть предварительно открыт с помощью OPEN_CURSOR. |
| *statement* | VARCHAR2 | SQL-оператор, грамматический разбор которого выполняется. Если это оператор DML или DDL, то в нем не нужно указывать конечную точку с запятой. Если это анонимный блок PL/SQL, то после заключительного оператора END должна быть указана точка с запятой. |
| *language_flag* | INTEGER | Указывает, как трактовать оператор. SQL-оператор можно обработать в режиме версии 6 или 7. Этот параметр может принимать три значения:<br>V6: режим версии 6<br>V7: режим версии 7<br>NATIVE: режим, установленный в той базе данных, с которой соединена данная программа. |

В параметре language_flag с помощью модульных констант DBMS_SQL.V6, DBMS_SQL.V7 или DBMS_SQL.NATIVE указывается режим версии 6 или 7. Параметра DBMS_SQL.V8 для Oracle8 не существует; в этом отношении Oracle8 ведет себя так же, как и Oracle7.

▼ ВНИМАНИЕ Единственная ситуация, когда DBMS_SQL используется для базы данных версии 6, — включение связи баз данных в состав оператора. Модуль DBMS_SQL является элементом языка PL/SQL версии 2.1 или выше, для которого требуется, по меньшей мере, Oracle7.

## Привязка входных переменных

При выполнении этой операции заполнители, указанные в операторе, связываются с фактическими переменными PL/SQL. Заполнитель определяется наличием двоеточия перед идентификатором. К примеру, в операторе

```
INSERT INTO temp_table (num_col, char_col)
VALUES (:number_value, :char_value);
```

содержится два заполнителя — **:number_value** и **:char_value**. Имена заполнителей могут быть произвольными. Если имя одного и того же заполнителя используется в нескольких операторах, то это значение связывается со всеми вхождениями этого имени. Если заполнители не присутствуют в операторе — привязка не требуется.

Процедура BIND_VARIABLE используется для привязки и указания имен заполнителей. Например, можно привязать заполнители, приведенные в предыдущем операторе INSERT, так:

```
DBMS_SQL.BIND_VARIABLE(v_CursorID, ':number_value', -7);
DBMS_SQL.BIND_VARIABLE(v_CursorID, ':char_value', 'Hello');
```

▼ ВНИМАНИЕ Двоеточие перед именем переменной привязки необязательно. Однако для соблюдения полноты изложения материала во всех примерах этой главы двоеточие указывается.

Размер и тип данных фактической переменной также определяются процедурой BIND_VARIABLE, через набор переопределенных вызовов. Для привязки переменных типа NUMBER используется следующий вызов:

```
PROCEDURE BIND_VARIABLE(c IN INTEGER,
                        name IN VARCHAR2,
                        value IN NUMBER);
```

для привязки переменных типа VARCHAR2:

```
PROCEDURE BIND_VARIABLE(c IN INTEGER,
                        name IN VARCHAR2,
                        value IN VARCHAR2);
```

```
PROCEDURE BIND_VARIABLE(c IN INTEGER,
                        name IN VARCHAR2,
                        value IN VARCHAR2,
                        out_value_size IN INTEGER);
```

для привязки переменных типа DATE:

```
PROCEDURE BIND_VARIABLE(c IN INTEGER,
                        name IN VARCHAR2,
                        value IN DATE);
```

Ниже приведены вызовы, использующиеся для привязки переменных типа CHAR. Эти вызовы выглядят по-другому, так как в PL/SQL запрещено переопределение для данных CHAR и VARCHAR2.

```
PROCEDURE BIND_VARIABLE_CHAR(c IN INTEGER,
                             name IN VARCHAR2,
                             value IN CHAR);
PROCEDURE BIND_VARIABLE_CHAR(c IN INTEGER,
                             name IN VARCHAR2,
                             value IN CHAR,
                             out_value_size IN INTEGER);
```

То же самое справедливо для привязки переменных типа RAW:

```
PROCEDURE BIND_VARIABLE_RAW (c IN INTEGER,
                             name IN VARCHAR2,
                             value IN RAW);
PROCEDURE BIND_VARIABLE_RAW (c IN INTEGER,
                             name IN VARCHAR2,
                             value IN RAW,
                             out_value_size IN INTEGER);
```

Такой вызов используется для привязки переменных типа MLSLABEL:

```
PROCEDURE BIND_VARIABLE (c IN INTEGER,
                         name IN VARCHAR2,
                         value IN MLSLABEL);
```

а такой — для привязки переменных типа ROWID:

```
PROCEDURE BIND_VARIABLE_ROWID(c IN INTEGER,
                              name IN VARCHAR2,
                              value IN ROWID);
```

Параметры, используемые в этих вызовах, описаны в таблице 15.2.

**ТАБЛИЦА 15.2.**

| Параметр | Тип | Описание |
|---|---|---|
| *c* | INTEGER | Идентификационный номер курсора. Этот курсор должен быть предварительно открыт с помощью OPEN_CURSOR, а также с помощью PARSE должен быть выполнен грамматический разбор оператора, для обработки которого открыт курсор. |
| *name* | VARCHAR2 | Имя заполнителя, с которым будет связана эта переменная (следует указывать двоеточие). |
| *value* | NUMBER, CHAR, VARCHAR2, DATE, ROWID, RAW | Реальные данные, которые будут привязываться. Тип и размер этой переменной также считываются. При необходимости содержащиеся в ней данные будут преобразованы. |
| *out_value_size* | INTEGER | Необязательный параметр, задаваемый при привязке переменных только типов CHAR и ROWID. Если он указан, то обозначает максимальный ожидаемый размер значения OUT в байтах. Если не указан — используется размер, заданный в параметре *value*. Этот параметр следует задавать только при обработке анонимных блоков PL/SQL, когда в переменную привязки может быть записанонекоторое значение. Для входных переменных размер, заданный в параметре *value*, не изменится (см. пример, приведенный ниже в этой главе) |

## Выполнение оператора

Функция EXECUTE используется для выполнения оператора и возвращает число обработанных строк. Значение, возвращаемое функцией EXECUTE, достоверно только для операторов DML. Для запросов, операторов DDL и анонимных блоков PL/SQL это значение не определено и поэтому игнорируется. EXECUTE является функцией, поэтому ее нужно вызывать из выражения. Описание EXECUTE выглядит следующим образом:

```
FUNCTION EXECUTE(c IN INTEGER) RETURN INTEGER;
```

Параметр и возвращаемое значение этой функции описаны в таблице 15.3.

**ТАБЛИЦА 15.3.**

| Параметр | Тип | Описание |
|---|---|---|
| *c* | INTEGER | Идентификатор курсора, содержащего выполняемый оператор. Этот курсор должен быть предварительно открыт, оператор — грамматически разобран, а все заполнители привязаны. |
| возвращаемое значение | INTEGER | Число строк, обрабатываемых оператором. Это значение аналогично курсорному атрибуту %ROWCOUNT. Определено только тогда, когда исполняемый оператор — это INSERT, UPDATE или DELETE. |

## Закрытие курсора

Когда обработка закончена, курсор должен быть закрыт. Это высвобождает ресурсы, выделенные курсору, и является сигналом того, что данный курсор больше не используется. После закрытия курсора работать с ним нельзя; для этого его нужно снова открыть. Синтаксис процедуры CLOSE_CURSOR таков:

```
PROCEDURE CLOSE_CURSOR(c IN OUT INTEGER);
```

Значение, передаваемое в процедуру CLOSE_CURSOR, должно быть достоверным идентификатором курсора. После такого вызова фактический параметр устанавливается в NULL, что свидетельствует о закрытии курсора.

## Пример

Процедура UpdateClasses обновляет число зачетов для учебных групп указанного факультета. Хотя эту процедуру можно написать и с помощью статического SQL, данный пример показывает, какие действия необходимы для обработки данных.

```
-- Этот пример содержится в файле updclass.sql.
CREATE OR REPLACE PROCEDURE UpdateClasses(
  /* Для обновления таблицы classes и установки указанного числа зачетов
     для всех групп данного факультета используется модуль DBMS_SQL. */
  p_Department IN classes.department%TYPE,
  p_NewCredits IN classes.num_credits%TYPE,
  p_RowsUpdated OUT INTEGER) AS

  v_CursorID INTEGER;
  v_UpdateStmt VARCHAR2(100);
BEGIN
  -- Откроем курсор для обработки данных.
  v_CursorID := DBMS_SQL.OPEN_CURSOR;

  -- Определим строку символов SQL-оператора.
  v_UpdateStmt :=
    'UPDATE classes
      SET num_credits = :nc
      WHERE department = :dept';

  -- Выполним грамматический разбор оператора.
```

```
    DBMS_SQL.PARSE(v_CursorID, v_UpdateStmt, DBMS_SQL.V7);

    -- Привяжем p_NewCredits к заполнителю :nc.
    DBMS_SQL.BIND_VARIABLE(v_CursorID, ':nc', p_NewCredits);

    -- Привяжем p_Department к заполнителю :dept.
    DBMS_SQL.BIND_VARIABLE(v_CursorID, ':dept', p_Department);

    -- Выполним оператор.
    p_RowsUpdated := DBMS_SQL.EXECUTE(v_CursorID);

    -- Закроем курсор.
    DBMS_SQL.CLOSE_CURSOR(v_CursorID);
  EXCEPTION
    WHEN OTHERS THEN
        -- Закроем курсор, затем установим ошибку вновь.
        DBMS_SQL.CLOSE_CURSOR(v_CursorID);
        RAISE;
  END UpdateClasses;
```

В следующем фрагменте сеанса работы с SQL*Plus показаны результаты выполнения **UpdateClasses**:

```
SQL> VARIABLE v_RowsUpdated NUMBER
SQL> exec UpdateClasses('MUS', 5, :v_RowsUpdated)

PL/SQL procedure successfully completed.

SQL> print v_RowsUpdated

V_ROWSUPDATED
--------------
            1
```

## Обработка операторов DDL

Обработка операторов DDL ненамного отличается от обработки операторов DML. Различия между этими процессами заключаются в следующем:

- В операторах DDL нельзя использовать переменные привязки, поэтому после грамматического разбора оператора не нужно вызывать процедуру BIND_VARIABLE.
- Операторы DDL выполняются при вызове процедуры PARSE. Вызов EXECUTE не нужен, так как он не будет иметь никакого эффекта.

Для примера перепишем процедуру **RecreateTempTable**, рассмотренную выше, следующим образом:

```
-- Этот пример содержится в файле recrtmp2.sql.
CREATE OR REPLACE PROCEDURE RecreateTempTable (
  /* Удаляет таблицу temp_table и повторно создает ее. Описание таблицы
     передается в процедуру с помощью p_Description и должно быть
     содержимым оператора CREATE TABLE, расположенным после имени таблицы.
     Например, ниже приведен правильный вызов процедуры:
     RecreateTempTable('(num_col NUMBER, char_col VARCHAR2(2000))');
  */
  p_Description IN VARCHAR2) IS

  v_Cursor NUMBER;
  v_CreateString VARCHAR2(100);
  v_DropString VARCHAR2(100);
```

```
BEGIN
  /* Откроем курсор для обработки данных. */
  v_Cursor := DBMS_SQL.OPEN_CURSOR;

  /* Удалим таблицу. */
  v_DropString := 'DROP TABLE temp_table';

  /* Проведем грамматический разбор команды 'DROP TABLE' и выполним ее.
     Если таблица не существует, установим ошибку ORA-942. */
  BEGIN
     -- DBMS_SQL.V7 - константа, описанная в заголовке модуля.
     DBMS_SQL.PARSE(v_Cursor, v_DropString, DBMS_SQL.V7);
  EXCEPTION
     WHEN OTHERS THEN
       IF SQLCODE != -942 THEN
         RAISE;
       END IF;
  END;
  /* Теперь создадим таблицу. Сначала нужно создать строку символов
     CREATE TABLE, а затем провести ее грамматический разбор и
     обработать ее. */
  v_CreateString := 'CREATE TABLE temp_table ' || p_Description;
  DBMS_SQL.PARSE(v_Cursor, v_CreateString, DBMS_SQL.V7);

  /* Работа закончена, поэтому закроем курсор. */
  DBMS_SQL.CLOSE_CURSOR(v_Cursor);
EXCEPTION
  WHEN OTHERS THEN
     /* Сначала закроем курсор, затем переустановим ошибку, чтобы
       передать ее вне блока. */
     DBMS_SQL.CLOSE_CURSOR(v_Cursor);
     RAISE;
END RecreateTempTable;
```

*За исключением двух указанных моментов, обработка операторов DDL абсолютно идентична обработке операторов DML:* открывается курсор (DBMS_SQL.OPEN_CURSOR), выполняется грамматический разбор оператора, причем на этом шаге оператор еще и исполняется (DBMS_SQL.PARSE), затем курсор закрывается (DBMS_SQL.CLOSE_CURSOR).

## Обработка запросов

Процесс обработки запросов с помощью DBMS_SQL практически совпадает с обработкой операторов DML и DDL до шага, на котором вызывается функция EXECUTE. Перед исполнением запроса необходимо определить типы и размер выходных переменных с помощью процедуры DEFINE_COLUMN. После исполнения нужно вызвать FETCH_ROWS и COLUMN_VALUE, чтобы считать полученные результаты. Алгоритм обработки запроса выглядит следующим образом:

1. Открыть курсор (OPEN_CURSOR).
2. Выполнить грамматический разбор оператора (PARSE).
3. Выполнить привязку всех входных переменных (BIND_VARIABLE).
4. Описать выходные переменные (DEFINE_COLUMN).
5. Выполнить запрос (EXECUTE).
6. Считать строки (FETCH_ROWS).
7. Записать результаты в переменные PL/SQL (COLUMN_VALUE).
8. Закрыть курсор (CLOSE_CURSOR).

Операции, аналогичные рассмотренным выше (OPEN_CURSOR, PARSE, BIND_VARIABLE, EXECUTE и CLOSE_CURSOR), выполняются точно так же, как и в случае обработки операторов DML и DDL. Далее в этом разделе подробно рассматриваются только те вызовы, которые не использовались ранее.

## Грамматический разбор оператора

Процедура PARSE выполняется так же, как и для операторов DML, однако строка символов запроса должна отвечать определенным условиям. Необходимо, чтобы запрос был одиночным оператором SELECT, а не в составе блока PL/SQL. Если запрос находится внутри блока, то он обрабатывается в соответствии с правилами, установленными для обработки блоков PL/SQL (см. следующий раздел). Кроме того, запрос не должен содержать конструкции INTO. Вместо нее используются процедуры DEFINE_COLUMN и COLUMN_VALUE. Наконец, как и в случае операторов DML и DDL, конечную точку с запятой указывать не надо. Например, все приведенные ниже запросы можно использовать в процедуре DBMS_SQL.PARSE.

```
SELECT * FROM students

SELECT COUNT(*) "Number of Students", department || course
  FROM registered_students
  WHERE department IN (:d1, :d2)
  GROUP BY department || course

SELECT FullName(ID), ID
  FROM students
  WHERE ID = :student_id
```

## Определение выходных переменных

Процесс определения очень напоминает процесс привязки, только входные переменные должны быть привязаны, а выходные – лишь определены. В процедуре DEFINE_COLUMN указываются типы и размер выходных переменных. Каждый пункт списка выбора (см. главу 4) преобразуется к типу соответствующей выходной переменной.

В отличие от заполнителей в BIND_VARIABLE, пункты списка выбора идентифицируются не по имени, а по позиции. Номер первой позиции равен 1, второй – 2 и т.д. Например, если выполнить грамматический разбор запроса

```
SELECT first_name, last_name, num_credits
FROM students
```

то вызов DEFINE_COLUMN будет выглядеть примерно так:

```
DECLARE
  V_FirstName    students.first_name%TYPE,
  V_LastName     students.last_name%TYPE,
  V_NumCredits   students.num_credits%TYPE,
  V_CursorID     INTEGER;
BEGIN
  ...
  DBMS_SQL.DEFINE_COLUMN(v_CursorID, 1, v_FirstName);
  DBMS_SQL.DEFINE_COLUMN(v_CursorID, 2, v_LastName);
  DBMS_SQL.DEFINE_COLUMN(v_CursorID, 3, v_NumCredits);
  ...
END;
```

Как и BIND_VARIABLE, DEFINE_COLUMN переопределяется типом переменной, в данном случае выходной переменной. Для определения переменных типа NUMBER используется следующий вызов:

```
PROCEDURE DEFINE_COLUMN(c IN INTEGER,
                        position IN INTEGER,
                        column IN NUMBER);
```

для определения переменных типа VARCHAR2:

```
PROCEDURE DEFINE_COLUMN(c IN INTEGER,
                        position IN INTEGER,
                        column IN VARCHAR2,
                        column_size IN INTEGER);
```

для определения переменных типа DATE:

```
PROCEDURE DEFINE_COLUMN(c IN INTEGER,
                        position IN INTEGER,
                        column IN DATE,
                        column_size IN INTEGER);
```

для определения переменных типа CHAR:

```
PROCEDURE DEFINE_COLUMN_CHAR(c IN INTEGER,
                             position IN INTEGER,
                             column IN CHAR,
                             column_size IN INTEGER);
```

для определения переменных типа RAW:

```
PROCEDURE DEFINE_COLUMN_RAW(c IN INTEGER,
                            position IN INTEGER,
                            column IN RAW,
                            column_size IN INTEGER);
```

для определения переменных типа MLSLABEL:

```
PROCEDURE DEFINE_COLUMN(c IN INTEGER,
                        position IN INTEGER,
                        column IN MLSLABEL);
```

и, наконец, для определения переменных типа ROWID:

```
PROCEDURE DEFINE_COLUMN(c IN INTEGER,
                        position IN INTEGER,
                        column IN ROWID);
```

▼ ВНИМАНИЕ При вызове DEFINE_COLUMN для переменных VARCHAR2, CHAR и RAW нужно обязательно указывать параметр column_size. Дело в том, что PL/SQL должен знать максимальный размер этих переменных во время выполнения программы. В отличие от NUMBER, DATE, MLSLABEL и ROWID, данные вышеуказанных типов не имеют фиксированной длины, известной компилятору PL/SQL.

Параметры DEFINE_COLUMN очень похожи на параметры BIND_VARIABLE и приведены в таблице 15.4.

**ТАБЛИЦА 15.4.**

| Параметр | Тип | Описание |
|---|---|---|
| *c* | INTEGER | Идентификационный номер курсора. Предварительно должен быть выполнен грамматический разбор запроса, для обработки которого открыт курсор, а входные переменные должны быть привязаны. |
| *position* | INTEGER | Относительная позиция пункта списка выбора. Позиция первого пункта списка равна 1. |
| *column* | NUMBER, VARCHAR2, CHAR, DATE, MLSLABEL, RAW, ROWID | Переменная, определяющая тип и размер выходной переменной. Имя переменной не играет особой роли, однако ее тип и размер важны. Тем не менее в DEFINE_COLUMN и BIND_VARIABLE обычно используются одни и те же переменные. |
| *column_size* | INTEGER | Максимальный ожидаемый размер выходных данных. Если не указан, то используется размер, заданный в параметре column. |

## Считывание строк

Строки, соответствующие условию WHERE запроса, считываются в буфер функцией FETCH_ROWS. После этого вызывается процедура COLUMN_VALUE, считывающая реальные данные из буфера в переменные PL/SQL. Описание функции FETCH_ROWS выглядит следующим образом:

```
FUNCTION FETCH_ROWS(c IN INTEGER) RETURN INTEGER;
```

Единственным параметром этой функции является идентификационный номер курсора. FETCH_ROWS возвращает число считываемых строк. FETCH_ROWS и COLUMN_VALUE обычно вызываются в цикле по нескольку раз до тех пор, пока FETCH_ROWS не возвратит ноль.

▼ ВНИМАНИЕ

Условием выхода из цикла является возврат функцией FETCH_ROWS нуля, а не исключительная ситуация NO_DATA_FOUND или курсорный атрибут %NOTFOUND.

Вызов функции EXECUTE и первый вызов FETCH_ROWS можно объединить в один вызов — EXECUTE_AND_FETCH. При работе с удаленной базой данных это помогает снизить нагрузку на сеть и тем самым повысить производительность системы. Синтаксис EXECUTE_AND_FETCH таков:

```
FUNCTION EXECUTE_AND_FETCH(c IN INTEGER,
                           exact IN BOOLEAN DEFAULT FALSE)
                           RETURN INTEGER;
```

Параметры описаны в таблице 15.5.

**ТАБЛИЦА 15.5.**

| Параметр | Тип | Описание |
|---|---|---|
| *c* | INTEGER | Идентификационный номер курсора. Предварительно должен быть выполнен грамматический разбор запроса, для обработки которого открыт курсор, входные переменные должны быть привязаны, а выходные — определены. |
| *exact* | BOOLEAN | Если TRUE, то в случае возврата запросом более одной строки устанавливается исключительная ситуация TOO_MANY_ROWS. Однако даже при этом строки выбираются и считываются. |
| возвращаемое значение | INTEGER | Число строк, выбранных ранее; оно аналогично числу, возвращаемому FETCH_ROWS. |

## Запись результатов в переменные PL/SQL

После того как с помощью FETCH_ROWS данные считаны в локальный буфер, нужно воспользоваться процедурой COLUMN_VALUE и записать их в переменные PL/SQL. Как правило, в COLUMN_VALUE используются те же переменные, что и в DEFINE_COLUMN. Каждому вызову DEFINE_COLUMN должен соответствовать вызов COLUMN_VALUE.

Как и BIND_VARIABLE и DEFINE_COLUMN, COLUMN_VALUE переопределяется типом переменной, в данном случае выходной переменной. Для переменных типа NUMBER используется следующий вызов:

```
PROCEDURE COLUMN_VALUE(c IN INTEGER,
                       position IN INTEGER,
                       value OUT NUMBER);

PROCEDURE COLUMN_VALUE(c IN INTEGER,
                       position IN INTEGER,
                       value OUT NUMBER,
                       column_error OUT NUMBER,
                       actual_length OUT INTEGER);
```

для переменных типа VARCHAR2:

```
PROCEDURE COLUMN_VALUE(c IN INTEGER,
                       position IN INTEGER,
                       value OUT VARCHAR2);
```

```
PROCEDURE COLUMN_VALUE(c IN INTEGER,
                       position IN INTEGER,
                       value OUT VARCHAR2,
                       column_error OUT NUMBER,
                       actual_length OUT INTEGER);
```

для переменных типа DATE:

```
PROCEDURE COLUMN_VALUE(c IN INTEGER,
                       position IN INTEGER,
                       value OUT DATE);
PROCEDURE COLUMN_VALUE(c IN INTEGER,
                       position IN INTEGER,
                       value OUT DATE,
                       column_error OUT NUMBER,
                       actual_length OUT INTEGER);
```

для переменных типа CHAR:

```
PROCEDURE COLUMN_VALUE_CHAR(c IN INTEGER,
                            position IN INTEGER,
                            value OUT CHAR);
PROCEDURE COLUMN_VALUE_CHAR(c IN INTEGER,
                            position IN INTEGER,
                            value OUT CHAR,
                            column_error OUT NUMBER,
                            actual_length OUT INTEGER);
```

для переменных типа RAW:

```
PROCEDURE COLUMN_VALUE_RAW(c IN INTEGER,
                           position IN INTEGER,
                           value OUT RAW);
PROCEDURE COLUMN_VALUE_RAW(c IN INTEGER,
                           position IN INTEGER,
                           value OUT RAW,
                           column_error OUT NUMBER,
                           actual_length OUT INTEGER);
```

для переменных типа MLSLABEL:

```
PROCEDURE COLUMN_VALUE(c IN INTEGER,
                       position IN INTEGER,
                       value OUT MLSLABEL);
PROCEDURE COLUMN_VALUE(c IN INTEGER,
                       position IN INTEGER,
                       value OUT MLSLABEL,
                       column_error OUT NUMBER,
                       actual_length OUT INTEGER);
```

и, наконец, для переменных типа ROWID:

```
PROCEDURE COLUMN_VALUE_ROWID(c IN INTEGER,
                             position IN INTEGER,
                             value OUT ROWID);
PROCEDURE COLUMN_VALUE_ROWID(c IN INTEGER,
                             position IN INTEGER,
                             value OUT ROWID,
                             column_error OUT NUMBER,
                             actual_length OUT INTEGER);
```

Параметры этих вызовов описаны в таблице 15.6.

**ТАБЛИЦА 15.6.**

| Параметр | Тип | Описание |
|---|---|---|
| *c* | INTEGER | Идентификационный номер курсора. Предварительно должен быть выполнен грамматический разбор запроса, для обработки которого открыт курсор, и все заполнители должны быть привязаны; запрос должен быть обработан, а нужная информация считана. |
| *position* | INTEGER | Относительная позиция в списке выбора. Как и в DEFINE_COLUMN, позиция первого пункта списка равна 1. |
| *value* | NUMBER, DATE, MLSLABEL, CHAR, VARCHAR2, RAW, ROWID | Выходная переменная, в которую будет записываться содержимое буфера для указанных строки и столбца. Если тип параметра *value* отличен от типа, указанного в DEFINE_COLUMN, возникает ошибка "ORA-6562: type of OUT argument must match type of column or bind variable" (тип аргумента OUT должен соответствовать типу столбца или переменной привязки), что равносильно установлению исключительной ситуации DBMS_SQL.INCONSISTENT_TYPES. |
| *column_error* | NUMBER | Код ошибки столбца. Если указан, то в переменной будет возвращаться код ошибки, подобной ошибке "ORA-1406: fetched *column value* is truncated" (выбранное значение столбца усечено). Код выдается в виде отрицательного значения, как в функции SQLCODE. Ошибка будет устанавливать исключительную ситуацию, но *column_error* позволяет определить, какой из столбцов стал причиной конкретной ошибки. Если столбец считан успешно, значение параметра *column_error* равно нулю. |
| *actual_length* | INTEGER | Если указан, то в данной переменной будет находиться исходный размер столбца (т.е. размер перед считыванием). Это удобно в том случае, когда размер выходной переменной недостаточен и значение усекается (что приводит к ошибке ORA-1406) |

# Пример

Процедура **DynamicQuery** заносит передаваемые ей значения имен, фамилий и профилирующих дисциплин студентов в таблицу **temp_table**. Хотя такую процедуру можно написать при помощи статического SQL, проиллюстрируем процесс обработки запросов посредством DBMS_SQL.

```
-- Этот пример содержится в файле dquery.sql.
CREATE OR REPLACE PROCEDURE DynamicQuery (
  /* DBMS_SQL используется для выдачи запроса к таблице students и
     для записи результатов в таблицу temp_table. Имена, фамилии и
     профилирующие дисциплины вносятся в таблицу, включающую не более
     двух профилирующих дисциплин. */
  p_Major1 IN students.major%TYPE DEFAULT NULL,
  p_Major2 IN students.major%TYPE DEFAULT NULL) AS

  V_CursorID     INTEGER;
  V_SelectStmt   VARCHAR2(500);
  V_FirstName    students.first_name%TYPE;
  V_LastName     students.last_name%TYPE;
  V_Major        students.major%TYPE;
  V_Dummy        INTEGER;
BEGIN
  -- Откроем курсор для обработки данных.
  v_CursorID := DBMS_SQL.OPEN_CURSOR;

  -- Создадим строку символов запроса.
  v_SelectStmt := 'SELECT first_name, last_name, major
                   FROM students
```

```
                    WHERE major IN (:m1, :m2)
                    ORDER BY major, last_name';

    -- Выполним грамматический разбор запроса.
    DBMS_SQL.PARSE(v_CursorID, v_SelectStmt, DBMS_SQL.V7);

    -- Привяжем входные переменные.
    DBMS_SQL.BIND_VARIABLE(v_CursorID, ':m1', p_Major1);
    DBMS_SQL.BIND_VARIABLE(v_CursorID, ':m2', p_Major2);

    -- Определим выходные переменные.
    DBMS_SQL.DEFINE_COLUMN(v_CursorID, 1, v_FirstName, 20);
    DBMS_SQL.DEFINE_COLUMN(v_CursorID, 2, v_LastName, 20);
    DBMS_SQL.DEFINE_COLUMN(v_CursorID, 3, v_Major, 30);

    -- Обработаем оператор. Сейчас необязательно знать о том, какое
    -- возвращается значение, однако нужно объявить для него переменную.
    v_Dummy := DBMS_SQL.EXECUTE(v_CursorID);

    -- Цикл выборки.
    LOOP
        -- Выберем строки в буфер, а также проверим, не выполняется ли
        -- условие выхода из цикла.
        IF DBMS_SQL.FETCH_ROWS(v_CursorID) = 0 THEN
          EXIT;
        END IF;

    -- Считаем строки из буфера и запишем их в переменные PL/SQL.
    DBMS_SQL.COLUMN_VALUE(v_CursorID, 1, v_FirstName);
    DBMS_SQL.COLUMN_VALUE(v_CursorID, 2, v_LastName);
    DBMS_SQL.COLUMN_VALUE(v_CursorID, 3, v_Major);

    -- Внесем считанные данные в таблицу temp_table.
    INSERT INTO temp_table (char_col)
        VALUES (v_FirstName || ' ' || v_LastName || ' is a ' ||
                v_Major || ' major.');
    END LOOP;

    -- Закроем курсор.
    DBMS_SQL.CLOSE_CURSOR(v_CursorID);

    -- Завершим работу.
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        -- Закроем курсор, а затем повторно установим ошибку.
        DBMS_SQL.CLOSE_CURSOR(v_CursorID);
        RAISE;
END DynamicQuery;
```

**Результат выполнения DynamicQuery в SQL*Plus выглядит следующим образом:**

```
SQL> exec DynamicQuery('History', 'Computer Science')
PL/SQL procedure successfully completed.

SQL> SELECT char_col FROM temp_table;
```

```
CHAR_COL
--------------------------------------------------
Joanne Juneburg is a Computer Science major.
Scott Smith is a Computer Science major.
Margaret Mason is a History major.
Patrick Poll is a History major.
Timothy Taller is a History major.
```

# Обработка блоков PL/SQL

Третьим и последним типом операторов являются анонимные блоки PL/SQL. Они часто используются для вызова хранимых процедур. Их обработка схожа с обработкой операторов DML и DDL в том, что в обоих случаях отсутствуют циклы выборки. Однако в отличие от операторов DML и DDL в блоках PL/SQL можно присваивать значения заполнителям. Для выбора выходных значений переменных привязки применяется дополнительный вызов – VARIABLE_VALUE (похожий на COLUMN_VALUE). Ниже приведен порядок вызова процедур и функций:

1. Открыть курсор (OPEN_CURSOR).
2. Выполнить грамматический разбор оператора (PARSE).
3. Выполнить привязку всех входных переменных (BIND_VARIABLE).
4. Выполнить оператор (EXECUTE).
5. Считать значения всех выходных переменных (VARIABLE_VALUE).
6. Закрыть курсор (CLOSE_CURSOR).

Рассмотрим только те вызовы, которые используются по-другому или применяются для обработки только анонимных блоков PL/SQL. Прочие процедуры и функции вызываются так же, как и ранее.

## Грамматический разбор оператора

В строке символов, передаваемой процедуре PARSE, должен содержаться полный анонимный блок PL/SQL. Конечную точку с запятой после заключительного оператора END следует обязательно указывать, поскольку она является синтаксическим элементом блока (для запросов и операторов DML/DDL точка с запятой необязательна, так как она не является синтаксическим элементом – это просто признак конца оператора). В блоке могут содержаться заполнители, которые привязываются с помощью BIND_VARIABLE. Если они используются в качестве выходных переменных (например, в качестве параметров OUT хранимой процедуры), то их новые значения могут быть считаны с помощью VARIABLE_VALUE после обработки блока. Ниже приведены правильные последовательности символов, которые можно передать процедуре PARSE:

```
BEGIN :placeholder := 7; END;
```

```
DECLARE
  v_Numeric NUMBER := :p1;
  v_Character VARCHAR2(50) := :p2;
BEGIN
  INSERT INTO temp_table VALUES (v_Numeric, v_Character);
END;
```

```
BEGIN
  SELECT first_name, last_name
      INTO :first_name, :last_name
      FROM students
      WHERE ID = :id;
END;
```

▼ ОСТОРОЖНО Не используйте комментарии, начинающиеся с --, внутри блока PL/SQL, выполняемого с помощью DBMS_SQL. Символы тире делают комментариями все последующие символы вплоть до символа новой строки. Таким образом, вся оставшаяся часть блока будет закомментирована, поскольку он весь включается в одну строку символов. Если нужны комментарии, применяйте ограничители комментариев /* и */, как в языке C.

## Считывание значений выходных переменных

После обработки блока значения всех выходных переменных можно считать с помощью процедуры VARIABLE_VALUE. Как и при обработке запросов, эти значения сначала сохраняются с помощью EXECUTE в буфере, а затем считываются из буфера процедурой VARIABLE_VALUE. Считывать нужно только те переменные привязки, которые используются в качестве выходных. Как правило, в VARIABLE_VALUE указываются те же переменные, что и в BIND_VARIABLE, однако это необязательно.

Как и BIND_VARIABLE и COLUMN_VALUE, VARIABLE_VALUE переопределяется типом выходной переменной. Для переменных типа NUMBER используется следующий вызов:

```
PROCEDURE VARIABLE_VALUE(c IN INTEGER,
                         name IN VARCHAR2,
                         value OUT NUMBER);
```

для переменных типа VARCHAR2:

```
PROCEDURE VARIABLE_VALUE(c IN INTEGER,
                         name IN VARCHAR2,
                         value OUT VARCHAR2);
```

для переменных типа DATE:

```
PROCEDURE VARIABLE_VALUE(c IN INTEGER,
                         name IN VARCHAR2,
                         value OUT DATE);
```

для переменных типа CHAR:

```
PROCEDURE VARIABLE_VALUE_CHAR(c IN INTEGER,
                              name IN VARCHAR2,
                              value OUT CHAR);
```

для переменных типа RAW:

```
PROCEDURE VARIABLE_VALUE_RAW(c IN INTEGER,
                             name IN VARCHAR2,
                             value OUT RAW);
```

для переменных типа MLSLABEL:

```
PROCEDURE VARIABLE_VALUE(c IN INTEGER,
                         name IN VARCHAR2,
                         value OUT MLSLABEL);
```

для переменных типа ROWID:

```
PROCEDURE VARIABLE_VALUE_ROWID(c IN INTEGER,
                               name IN VARCHAR2,
                               value OUT ROWID);
```

Параметры, используемые в этих вызовах, описаны в таблице 15.7.

**ТАБЛИЦА 15.7.**

| Параметр | Тип | Описание |
|---|---|---|
| *c* | INTEGER | Идентификационный номер курсора. Этот курсор должен быть уже открыт, блок PL/SQL — грамматически разобран и обработан с предварительной привязкой всех заполнителей. |
| *name* | VARCHAR2 | Имя заполнителя (включая двоеточие), значение которого должно быть считано. |

**ТАБЛИЦА 15.7. (продолжение)**

| Параметр | Тип | Описание |
|---|---|---|
| *value* | NUMBER, CHAR, VARCHAR2, DATE, ROWID, MLSLABEL | Выходная переменная для считывания результата. Если тип параметра *value* не соответствует типу, указанному в BIND_VARIABLE, возникает ошибка "ORA-6562:type of OUT argument must match type of column or bind variable" (тип аргумента OUT должен соответствовать типу столбца или переменной привязки), что равносильно установлению исключительной ситуации DBMS_SQL.INCONSISTENT_TYPES. Этот параметр аналогичен параметру *value* в COLUMN_VALU |

# Пример

Процедура **DynamicPLSQL** выполняет блок PL/SQL, в котором запрашивается таблица students. Заметьте, что процедуре нужно передать максимальный размер выходных заполнителей **:first_name** и **:last_name**. До начала обработки блока в этих заполнителях не содержится никаких значений, поэтому их максимальный размер не может быть определен автоматически.

```
-- Этот пример содержится в файле dynplsql.sql.
CREATE OR REPLACE PROCEDURE DynamicPLSQL (
  /* Динамически выполняет блок PL/SQL. В блоке выбирается
     информация в таблице students, а p_StudentID используется в
     качестве входного заполнителя. */
  p_StudentID IN students.ID%TYPE) IS

  v_CursorID INTEGER;
  v_BlockStr VARCHAR2(500);
  v_FirstName students.first_name%TYPE;
  v_LastName students.last_name%TYPE;
  v_Dummy INTEGER;

BEGIN
  -- Откроем курсор для обработки данных.
  v_CursorID := DBMS_SQL.OPEN_CURSOR;

  -- Создадим строку символов, содержащую блок PL/SQL. В этой строке
  -- заполнители :first_name и :last_name являются выходными
  -- переменными, а заполнитель :ID — входной переменной.
  v_BlockStr :=
      'BEGIN
        SELECT first_name, last_name
          INTO :first_name, :last_name
          FROM students
          WHERE ID = :ID;
      END;';

  -- Выполним грамматический разбор оператора.
  DBMS_SQL.PARSE(v_CursorID, v_BlockStr, DBMS_SQL.V7);

  -- Выполним привязку заполнителей к переменным. Обратите внимание, что
  -- привязка производится как для входных, так и для выходных переменных.
  -- Передадим максимальный размер для :first_name и :last_name.
  DBMS_SQL.BIND_VARIABLE(v_CursorID, ':first_name', v_FirstName, 20);
  DBMS_SQL.BIND_VARIABLE(v_CursorID, ':last_name', v_LastName, 20);
  DBMS_SQL.BIND_VARIABLE(v_CursorID, ':ID', p_StudentID);
```

```
    -- Обработаем оператор. Сечас необязательно знать о том, какое
    -- возвращается значение, однако нужно объявить для него переменную.
    v_Dummy := DBMS_SQL.EXECUTE(v_CursorID);

    -- Считаем значения для выходных переменных.
    DBMS_SQL.VARIABLE_VALUE(v_CursorID, ':first_name', v_FirstName);
    DBMS_SQL.VARIABLE_VALUE(v_CursorID, ':last_name', v_LastName);

    -- Внесем эти значения в таблицу temp_table.
    INSERT INTO temp_table (num_col, char_col)
        VALUES (p_StudentID, v_FirstName || ' ' || v_LastName);

    -- Закроем курсор.
    DBMS_SQL.CLOSE_CURSOR(v_CursorID);

    -- Завершим работу.
    COMMIT;
  EXCEPTION
    WHEN OTHERS THEN
        -- Закроем курсор, а затем повторно установим ошибку.
        DBMS_SQL.CLOSE_CURSOR(v_CursorID);
        RAISE;
  END DynamicPLSQL;
```

Запустим **DynamicPLSQL** на выполнение из SQL*Plus:

```
SQL> exec DynamicPLSQL (10010)
PL/SQL procedure successfully completed.

SQL> exec DynamicPLSQL (10003)
PL/SQL procedure successfully completed.

SQL> SELECT * FROM temp_table
  NUM_COL  CHAR_COL
----------  ----------
    10010  Rita Razmataz
    10003  Manish Murgratroid
```

## Использование параметра out_value_size

При использовании BIND_VARIABLE для выходных символьных переменных важно указывать значение параметра **out_value_size**. Модифицируем вызовы привязки в процедуре **DynamicPLSQL** следующим образом:

```
    -- Выполним привязку заполнителей к переменным. Обратите внимание,
    -- что привязка производится как для входных, так и для выходных
    -- переменных. Не будем передавать максимальный размер для
    -- v_FirstName и v_LastName.
    DBMS_SQL.BIND_VARIABLE(v_CursorID, ':first_name', v_FirstName);
    DBMS_SQL.BIND_VARIABLE(v_CursorID, ':last_name', v_LastName);
    DBMS_SQL.BIND_VARIABLE(v_CursorID, ':ID', p_StudentID);
```

Теперь, после запуска **DynamicPLSQL** на выполнение, выдается сообщение об ошибке числа или значения:

```
SQL> exec DynamicPLSQL (10010)
begin DynamicPLSQL (10010); end;
```

```
*
ERROR at line 1:
ORA-6502: PL/SQL: numeric or value error
ORA-06512: at "EXAMPLE.DYNAMICPLSQL", line 51
ORA-06512: at line 1
```

В чем причина этой ошибки? Все дело в том, как переменные привязки обрабатываются в базе данных Oracle. Они не анализируются до тех пор, пока оператор не будет выполнен, после чего на основании значений переменных определяется их фактический размер. В процедуре **DynamicPLSQL** обе переменных – **v_FirstName** и **v_LastName** – не инициализируются программой, и, как следствие, переменным привязки назначается нулевой размер. Поэтому, когда процедура VARIABLE_VALUE пытается присвоить некоторое значение переменной **v_FirstName**, происходит ошибка. Если же процедуре BIND_VARIABLE передается параметр *out_value_size*, как в первоначальном примере, то вместо фактического размера переменной используется указанное значение.

Альтернативой является инициализация переменных привязки строками символов максимальной длины, например:

```
-- Выполним привязку заполнителей к переменным. Обратите внимание,
-- что привязка производится как для входных, так и для выходных
-- переменных. Сначала инициализируем переменные значениями,
-- соответствующими максимальному размеру каждой из них.
v_FirstName := '12345678901234567890';
v_LastName := '12345678901234567890';
DBMS_SQL.BIND_VARIABLE(v_CursorID, ':first_name', v_FirstName);
DBMS_SQL.BIND_VARIABLE(v_CursorID, ':last_name', v_LastName);
DBMS_SQL.BIND_VARIABLE(v_CursorID, ':ID', p_StudentID);
```

# PL/SQL в работе: выполнение произвольных хранимых процедур

PL/SQL 2.3 ... и ВЫШЕ

Модуль DBMS_SQL в сочетании с модулем DBMS_DESCRIBE – очень удобное средство выполнения произвольных хранимых процедур, которое можно использовать в динамических программах, допускающих ввод информации пользователями, или в качестве оболочки модуля DBMS_SQL.

DBMS_DESCRIBE – это модуль, в состав которого входит одна процедура – DESCRIBE_PROCEDURE. Она возвращает информацию о параметрах процедур, в том числе их имена, виды параметров и типы. Интерфейс этой процедуры схож с интерфейсом процедуры DBMS_SQL.DESCRIBE_COLUMNS, описанной ниже. (Более подробно о модуле DBMS_DESCRIBE см. в приложении В.)

Ниже приведен программный текст модуля **ExecuteAny**. Поскольку при работе этого модуля используются таблицы записей, для него необходим PL/SQL версии 2.3 или выше.

```
-- Этот пример содержится в файле execany.sql.
CREATE OR REPLACE PACKAGE ExecuteAny AS
  -- Базовый параметр для хранимой процедуры или функции.
  TYPE t_Parameter IS RECORD (
      Actual_type     VARCHAR2(8),    -- Один из типов 'NUMBER',
                                      -- 'VARCHAR2', 'DATE', 'CHAR'
      Actual_length   INTEGER,
      Name            VARCHAR2(50),
      Num_param       NUMBER,
      Vchar_param     VARCHAR2(500),
      Char_param      CHAR(500),
      Date_param      sDATE);

  -- Список базовых параметров.
  TYPE t_ParameterList IS TABLE OF t_Parameter
```

```
        INDEX BY BINARY_INTEGER;
    -- Выполняет произвольную процедуру. Во всех параметрах IN в
    -- p_Parameters должны быть заполнены хотя бы поля _param и
    -- actual_type, а во всех параметрах OUT - хотя бы поле
    -- actual_type. При выводе заполняется поле name.
    PROCEDURE RunProc(p_NumParams IN NUMBER,
                      p_ProcName IN VARCHAR2,
                      p_Parameters IN OUT t_ParameterList);

    -- Заполняет внутренние структуры данных описанием процедуры,
    -- указанным в p_ProcName. Если значение p_Print истинно, то
    -- эта информация выводится с помощью DBMS_OUTPUT.
    PROCEDURE DescribeProc(p_ProcName IN VARCHAR2,
                           p_Print IN BOOLEAN);

    -- Выводит на экран при помощи DBMS_OUTPUT параметры списка
    -- p_Parameters.
    PROCEDURE Printparams(p_Parameters IN t_ParameterList,
                          p_NumParams IN NUMBER);

END ExecuteAny;

CREATE OR REPLACE PACKAGE BODY ExecuteAny AS
    -- Внутренние переменные процедуры DBMS_DESCRIBE.DESCRIBE_PROCEDURE.
    V_Overload       DBMS_DESCRIBE.NUMBER_TABLE;
    V_Position       DBMS_DESCRIBE.NUMBER_TABLE;
    V_Level          DBMS_DESCRIBE.NUMBER_TABLE;
    V_ArgumentName   DBMS_DESCRIBE.VARCHAR2_TABLE;
    V_Datatype       DBMS_DESCRIBE.NUMBER_TABLE;
    V_DefaultValue   DBMS_DESCRIBE.NUMBER_TABLE;
    V_InOut          DBMS_DESCRIBE.NUMBER_TABLE;
    V_Length         DBMS_DESCRIBE.NUMBER_TABLE;
    V_Precision      DBMS_DESCRIBE.NUMBER_TABLE;
    V_Scale          DBMS_DESCRIBE.NUMBER_TABLE;
    V_Radix          DBMS_DESCRIBE.NUMBER_TABLE;
    V_Spare          DBMS_DESCRIBE.NUMBER_TABLE;

    -- Локальная функция для преобразования кодов типов данных в
    -- строки символов.
    FUNCTION ConvertDatatype(p_Code IN NUMBER)
        RETURN VARCHAR2 IS
        v_Output VARCHAR2(20);
    BEGIN
        SELECT DECODE(p_Code,  0, ' ',
                               1, 'VARCHAR2',
                               2, 'NUMBER',
                               3, 'BINARY_INTEGER',
                               8, 'LONG',
                              11, 'ROWID',
                              12, 'DATE',
                              23, 'RAW',
                              24, 'LONG RAW',
                              96, 'CHAR',
                             106, 'MLSLABEL',
```

```
                                250,  'RECORD',
                                251,  'TABLE',
                                252,  'BOOLEAN')
    INTO v_Output
    FROM dual;

    RETURN v_Output;
END ConvertDatatype;

-- Локальная функция для преобразования видов параметров в
-- строки символов.
FUNCTION ConvertMode(p_Code IN NUMBER)
    RETURN VARCHAR2 IS
    v_Output VARCHAR2(10);
BEGIN
    SELECT DECODE(p_Code,  0,  'IN',
                           1,  'IN OUT',
                           2,  'OUT')
      INTO v_Output
      FROM dual;
    RETURN v_Output;
END ConvertMode;

PROCEDURE DescribeProc(p_ProcName IN VARCHAR2,
                       p_Print IN BOOLEAN) IS
    v_ArgCounter NUMBER := 1;
BEGIN
    -- Сначала вызовем DESCRIBE_PROCEDURE для заполнения внутренних
    -- переменных информацией о процедуре.
    DBMS_DESCRIBE.DESCRIBE_PROCEDURE(
      p_ProcName,
      null,
      null,
      v_Overload,
      v_Position,
      v_Level,
      v_ArgumentName,
      v_Datatype,
      v_DefaultValue,
      v_InOut,
      v_Length,
      v_Precision,
      v_Scale,
      v_Radix,
      V_Spare);

IF NOT p_Print THEN
    RETURN;
END IF;

-- Выходные заголовки.
DBMS_OUTPUT.PUT_LINE('Description of ' || p_ProcName || ':');
DBMS_OUTPUT.PUT('Overload Position Argument Name Level ');
DBMS_OUTPUT.PUT('Datatype Mode Length Precision Scale');
```

```
    DBMS_OUTPUT.NEW_LINE;
    DBMS_OUTPUT.PUT('--------- ------- ------------- ------');
    DBMS_OUTPUT.PUT('---------- ----- ------ ---------- -------');
    DBMS_OUTPUT.NEW_LINE;

    -- Выходная информация о параметрах.
    LOOP
        BEGIN
           DBMS_OUTPUT.PUT(RPAD(TO_CHAR(v_Overload(v_ArgCounter)), 9));
           DBMS_OUTPUT.PUT(RPAD(TO_CHAR(v_Position(v_ArgCounter)), 9));
           DBMS_OUTPUT.PUT(RPAD(v_ArgumentName(v_ArgCounter), 14));
           DBMS_OUTPUT.PUT(RPAD(TO_CHAR(v_Level(v_ArgCounter)), 6));
           DBMS_OUTPUT.PUT(RPAD(ConvertDatatype(v_Datatype
                                                (v_ArgCounter)), 15));
           DBMS_OUTPUT.PUT(RPAD(ConvertMode(v_InOut(v_ArgCounter)), 7));
           DBMS_OUTPUT.PUT(RPAD(TO_CHAR(v_Length(v_ArgCounter)), 7));
           DBMS_OUTPUT.PUT(RPAD(TO_CHAR(v_Precision(v_ArgCounter)), 10));
           DBMS_OUTPUT.PUT(RPAD(TO_CHAR(v_Scale(v_ArgCounter)), 5));
           DBMS_OUTPUT.NEW_LINE;
           v_ArgCounter := v_ArgCounter + 1;
        EXCEPTION
           WHEN NO_DATA_FOUND THEN
              EXIT;
        END;
    END LOOP;

END DescribeProc;
PROCEDURE RunProc(p_NumParams IN NUMBER,
                  p_ProcName IN VARCHAR2,
                  p_Parameters IN OUT t_ParameterList) IS

    -- Переменные DBMS_SQL.
    v_Cursor NUMBER;
    v_NumRows NUMBER;

    v_ProcCall VARCHAR2(500);
    v_FirstParam BOOLEAN := TRUE;
BEGIN
    -- Сначала опишем процедуру.
    DescribeProc(p_ProcName, TRUE);

    -- Теперь создадим последовательность символов вызова процедуры
    -- в виде: 'BEGIN <имя_процедуры>(:p1, :p2, ...); END;'
    v_ProcCall := 'BEGIN ' || p_ProcName || '(';

    FOR v_Counter IN 1..p_NumParams LOOP
        IF v_FirstParam THEN
           v_ProcCall := v_ProcCall || ':' || v_ArgumentName(v_Counter);
           v_FirstParam := FALSE;
        ELSE
           v_ProcCall := v_ProcCall || ', :' || v_ArgumentName(v_Counter);
        END IF;
    END LOOP;
```

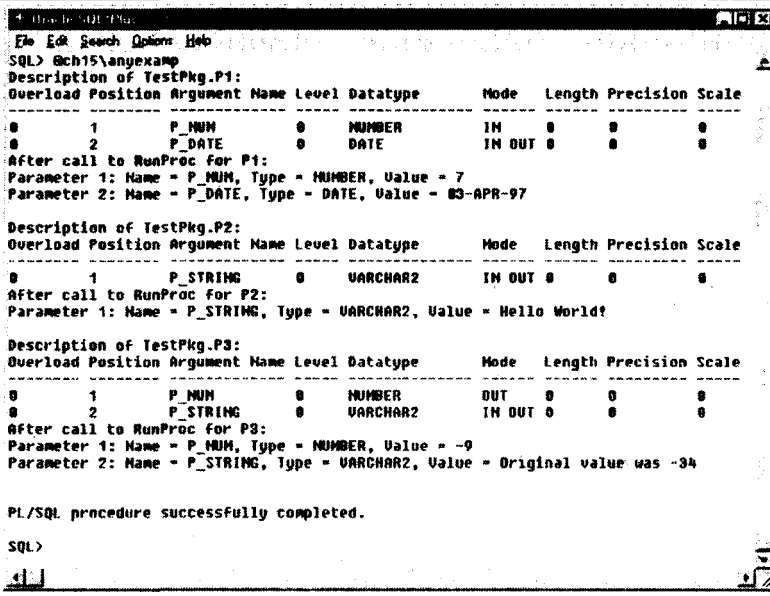
```
v_ProcCall := v_ProcCall || '); END;';
-- Откроем курсор и выполним грамматический разбор оператора.
v_Cursor := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(v_Cursor, v_ProcCall, DBMS_SQL.V7);

-- Выполним привязку параметров процедуры.
FOR v_Counter IN 1..p_NumParams LOOP

    -- Зададим имя процедуры.
    p_Parameters(v_Counter).name := v_ArgumentName(v_Counter);

    -- Выполним привязку на основании типа параметра.
    IF p_Parameters(v_Counter).actual_type = 'NUMBER' THEN
       DBMS_SQL.BIND_VARIABLE(v_Cursor, p_Parameters(v_Counter).name,
                              p_Parameters(v_Counter).num_param);
    ELSIF p_Parameters(v_Counter).actual_type = 'VARCHAR2' THEN
       DBMS_SQL.BIND_VARIABLE(v_Cursor, p_Parameters(v_Counter).name,
                              p_Parameters(v_Counter).vchar_param, 500);
    ELSIF p_Parameters(v_Counter).actual_type = 'DATE' THEN
       DBMS_SQL.BIND_VARIABLE(v_Cursor, p_Parameters(v_Counter).name,
                              p_Parameters(v_Counter).date_param);
    ELSIF p_Parameters(v_Counter).actual_type = 'CHAR' THEN
       DBMS_SQL.BIND_VARIABLE_CHAR(v_Cursor, p_Parameters(v_Counter).name,
                                   p_Parameters(v_Counter).char_param, 500);
    ELSE
       RAISE_APPLICATION_ERROR(-20001, 'Invalid type');
    END IF;
END LOOP;

-- Выполним процедуру.
v_NumRows := DBMS_SQL.EXECUTE(v_Cursor);

-- Вызовем VARIABLE_VALUE для всех параметров OUT и IN OUT.
FOR v_Counter IN 1..p_NumParams LOOP
    IF v_InOut(v_Counter) = 1 OR v_InOut(v_Counter) = 2 THEN
       IF p_Parameters(v_Counter).actual_type = 'NUMBER' THEN
          DBMS_SQL.VARIABLE_VALUE(v_Cursor, ':' ||
                                  p_Parameters(v_Counter).name
                                  p_Parameters(v_Counter).num_param);
       ELSIF p_Parameters(v_Counter).actual_type = 'VARCHAR2' THEN
          DBMS_SQL.VARIABLE_VALUE(v_Cursor, ':' ||
                                  p_Parameters(v_Counter).name
                                  p_Parameters(v_Counter).vchar_param);
       ELSIF p_Parameters(v_Counter).actual_type = 'DATE' THEN
          DBMS_SQL.VARIABLE_VALUE(v_Cursor, ':' ||
                                  p_Parameters(v_Counter).name
                                  p_Parameters(v_Counter).date_param);
       ELSIF p_Parameters(v_Counter).actual_type = 'CHAR' THEN
          DBMS_SQL.VARIABLE_VALUE_CHAR(v_Cursor, ':' ||
                                  p_Parameters(v_Counter).name,
                                  p_Parameters(v_Counter).char_param);
       ELSE
          RAISE_APPLICATION_ERROR(-20001, 'Invalid type');
       END IF;
```

```
        END IF;
    END LOOP;
END RunProc;

    PROCEDURE Printparams(p_Parameters IN t_ParameterList,
                          p_NumParams IN NUMBER) IS
    BEGIN
        -- Последовательно просмотрим все параметры и выведем имя, тип и
        -- значение каждого из них.
        FOR v_Counter IN 1..p_NumParams LOOP
            DBMS_OUTPUT.PUT('Parameter ' || v_Counter || ': Name = ');
            DBMS_OUTPUT.PUT(p_Parameters(v_Counter).name || ', Type = ');
            DBMS_OUTPUT.PUT(p_Parameters(v_Counter).actual_type ||
                            ', Value = ');
            IF p_Parameters(v_Counter).actual_type = 'NUMBER' THEN
                DBMS_OUTPUT.PUT_LINE(p_Parameters(v_Counter).num_param);
            ELSIF p_Parameters(v_Counter).actual_type = 'VARCHAR2' THEN
                DBMS_OUTPUT.PUT_LINE(p_Parameters(v_Counter).vchar_param);
            ELSIF p_Parameters(v_Counter).actual_type = 'DATE' THEN
                DBMS_OUTPUT.PUT_LINE(p_Parameters(v_Counter).date_param);
            ELSE
                DBMS_OUTPUT.PUT_LINE(p_Parameters(v_Counter).char_param);
            END IF;
        END LOOP;
    END PrintParams;
END ExecuteAny;
```

Далее следует пример использования **ExecuteAny**. Создадим модуль **TestPkg**:

```
-- Этот пример содержится в файле testpkg.sql.
CREATE OR REPLACE PACKAGE TestPkg AS
    -- Это достаточно простой модуль, причем для лучшей иллюстрации работы
    -- модуля ExecuteAny аргументы процедур имеют различные типы.
    PROCEDURE P1(p_Num IN NUMBER, p_Date OUT DATE);
    PROCEDURE P2(p_String OUT VARCHAR2);
    PROCEDURE P3(p_Num IN OUT NUMBER, p_String OUT VARCHAR2);
END TestPkg;

CREATE OR REPLACE PACKAGE BODY TestPkg AS
    PROCEDURE P1(p_Num IN NUMBER, p_Date OUT DATE) IS
    BEGIN
        p_Date := SYSDATE;
    END P1;

    PROCEDURE P2(p_String OUT VARCHAR2) IS
    BEGIN
        p_String := 'Hello World!';
    END P2;

    PROCEDURE P3(p_Num IN OUT NUMBER, p_String OUT VARCHAR2) IS
    BEGIN
        p_String := 'Original value was ' || TO_NUMBER(p_Num);
        p_Num := p_Num + 25;
    END P3;
END TestPkg;
```

**Рис. 15.2.**
*Результаты работы модуля ExecuteAny*



```
File Edit Search Options Help
SQL> @ch15\anyexamp
Description of TestPkg.P1:
Overload Position Argument Name Level Datatype       Mode   Length Precision Scale
-------- -------- ------------- ----- -------------- ------ ------ --------- -----
0        1        P_NUM         0     NUMBER         IN     0      0         0
0        2        P_DATE        0     DATE           IN OUT 0      0         0
After call to RunProc for P1:
Parameter 1: Name = P_NUM, Type = NUMBER, Value = 7
Parameter 2: Name = P_DATE, Type = DATE, Value = 03-APR-97

Description of TestPkg.P2:
Overload Position Argument Name Level Datatype       Mode   Length Precision Scale
-------- -------- ------------- ----- -------------- ------ ------ --------- -----
0        1        P_STRING      0     VARCHAR2       IN OUT 0      0         0
After call to RunProc for P2:
Parameter 1: Name = P_STRING, Type = VARCHAR2, Value = Hello World!

Description of TestPkg.P3:
Overload Position Argument Name Level Datatype       Mode   Length Precision Scale
-------- -------- ------------- ----- -------------- ------ ------ --------- -----
0        1        P_NUM         0     NUMBER         OUT    0      0         0
0        2        P_STRING      0     VARCHAR2       IN OUT 0      0         0
After call to RunProc for P3:
Parameter 1: Name = P_NUM, Type = NUMBER, Value = -9
Parameter 2: Name = P_STRING, Type = VARCHAR2, Value = Original value was -34


PL/SQL procedure successfully completed.

SQL>
```

Можно воспользоваться модулем **ExecuteAny** для вызова процедур модуля **TestPkg** с помощью следующего блока. Результаты показаны на рис. 15.2.

```
-- Этот пример содержится в файле anyexamp.sql.
DECLARE
  -- В v_Params будет храниться список параметров вызываемых процедур.
  v_Params  ExecuteAny.t_ParameterList;
BEGIN
  -- Заполним v_Params информацией о TestPkg.P1. Поскольку вид первого
  -- параметра IN, а второго – OUT, то для параметра 1 нужно заполнить
  -- только значение _param. Все параметры OUT заполнит ExecuteAny.RunProc.
  -- Однако необходимо указать тип каждого параметра.
  v_Params(1).actual_type  := 'NUMBER';
  v_Params(1).num_param  := 7;
  v_Params(2).actual_type  := 'DATE';
  ExecuteAny.RunProc(2, 'TestPkg.P1', v_Params);
  DBMS_OUTPUT.PUT_LINE('After call to RunProc for P1:');
  ExecuteAny.PrintParams(v_Params, 2);
  DBMS_OUTPUT.NEW_LINE;

  -- Заполним v_Params для TestPkg.P2. Поскольку в P2 используется
  -- только один параметр OUT, нужно указать лишь его тип.
  v_Params(1).actual_type  := 'VARCHAR2';
  ExecuteAny.RunProc(1, 'TestPkg.P2', v_Params);
  DBMS_OUTPUT.PUT_LINE('After call to RunProc for P2:');
  ExecuteAny.PrintParams(v_Params, 1);
  DBMS_OUTPUT.NEW_LINE;

  -- Заполним v_Params для TestPkg.P3.
  v_Params(1).actual_type  := 'NUMBER';
  v_Params(1).num_param  := -34;
  v_Params(2).actual_type  := 'VARCHAR2';
  ExecuteAny.RunProc(2, 'TestPkg.P3', v_Params);
  DBMS_OUTPUT.PUT_LINE('After call to RunProc for P3:');
```

```
    ExecuteAny.PrintParams(v_Params,  2);
    DBMS_OUTPUT.NEW_LINE;
END;
```

# Новые возможности DBMS_SQL в PL/SQL 8.0

PL/SQL 8.0 ... и ВЫШЕ

В модуль DBMS_SQL были внесены изменения, расширяющие его возможности и позволяющие применять его в Oracle8 и PL/SQL 8.0. В число новых средств входят: средство грамматического разбора больших строк символов SQL, средство обработки массивов данных и средство привязки и описания типов Oracle8, таких как объекты и объекты LOB, а также процедура DESCRIBE_COLUMNS. Для выполнения всех примеров, приведенных в этой главе, требуется Oracle8.

## Грамматический разбор больших строк символов SQL

Поскольку максимальная длина данных типа VARCHAR2 равна 32 767 байт, а параметр statement в процедуре PARSE имеет тип VARCHAR2, максимальный размер SQL-операторов, обрабатываемых с помощью модуля DBMS_SQL, ограничен. Это ограничение можно устранить, если вызвать PARSE следующим образом:

```
PROCEDURE DBMS_SQL.PARSE(c IN INTEGER,
                         statement IN VARCHAR2S,
                         lb IN INTEGER,
                         ub IN INTEGER,
                         lfflg IN BOOLEAN,
                         language_flag IN INTEGER);
```

Тип VARCHAR2S описывается так:

```
TYPE VARCHAR2S IS TABLE OF VARCHAR2(256)
  INDEX BY BINARY_INTEGER;
```

Теперь SQL-оператор передается в таблицу PL/SQL, поэтому можно выполнять грамматический разбор операторов произвольной длины (не превышающей предела, определяемого сервером). Параметры этой версии процедуры PARSE описаны в таблице 15.8.

**ТАБЛИЦА 15.8.**

| Параметр | Тип | Описание |
|---|---|---|
| *c* | INTEGER | Номер курсора. |
| *statement* | DBMS_SQL.VARCHAR2S | Строка символов, подвергаемая грамматическому разбору. Ее следует разбить на части с максимальной длиной в 256 символов каждая. |
| *lb* | INTEGER | Нижняя граница в таблице *statement*. |
| *ub* | INTEGER | Верхняя граница в таблице *statement*. |
| *lfflg* | BOOLEAN | Если TRUE, то после каждого элемента таблицы *statement* вводится символ перевода строки. |
| *language_flag* | INTEGER | Определяет, каким образом Oracle обрабатывает оператор. Аналогичен параметру *language_flag*, рассмотренному выше. |

Для работы с данной версией PARSE нужно скопировать оператор в таблицу *statement*, располагая каждый фрагмент оператора последовательно в элементах таблицы, начиная с элемента *lb* и заканчивая элементом *ub*. Затем модуль DBMS_SQL объединит все фрагменты в один оператор, который будет выглядеть следующим образом:

```
statement(lb) || statement(lb + 1) || ... || statement(ub)
```

В качестве примера перепишем раздел грамматического разбора процедуры DynamicPLSQL:

```
CREATE OR REPLACE PROCEDURE DynamicPLSQL (
  p_StudentID IN students.ID%TYPE) IS

  v_CursorID INTEGER;
  v_BlockStr DBMS_SQL.VARCHAR2S;
  ...
BEGIN
  -- Откроем курсор для обработки данных.
  v_CursorID := DBMS_SQL.OPEN_CURSOR;

  -- Создадим строку символов, содержащую блок PL/SQL. В этой строке
  -- заполнители :first_name и :last_name являются выходными
  -- переменными, а заполнитель :ID - входной переменной.
  v_BlockStr(1) := 'BEGIN';
  v_BlockStr(2) := 'SELECT first_name, last_name';
  v_BlockStr(3) := ' INTO :first_name, :last_name';
  v_BlockStr(4) := '   FROM students';
  v_BlockStr(5) := '   WHERE ID = :ID;';
  v_BlockStr(6) := 'END;';

  -- Выполним грамматический разбор оператора.
  DBMS_SQL.PARSE(v_CursorID, v_BlockStr, 1, 6, TRUE, DBMS_SQL.V7);
  ...
END DynamicPLSQL;
```

## Обработка массивов с помощью DBMS_SQL

Средство обработки массивов (array processing) позволяет выполнять различные операции с большими объемами данных при помощи одного SQL-оператора. К примеру, можно ввести в базу данных 100 строк, и для этого потребуется всего одна передача данных по сети от клиента к серверу. Без средства обработки массивов в такой ситуации потребовалось бы выполнить 100 передач данных.

В DBMS_SQL обработка массивов выполняется с помощью процедур BIND_ARRAY (для пакетного ввода, обновления и удаления информации) и DEFINE_ARRAY ( для пакетных запросов).

### BIND_ARRAY

Операции, выполняемые процедурой BIND_ARRAY, и аргументы этой процедуры очень похожи на функционирование и аргументы процедуры BIND_VARIABLE. Однако теперь привязываемая переменная является таблицей PL/SQL, а не скалярной переменной. Ниже приведены типы, разрешенные для использования в процедуре BIND_ARRAY:

```
TYPE NUMBER_TABLE IS TABLE OF NUMBER
  INDEX BY BINARY_INTEGER;
TYPE VARCHAR2_TABLE IS TABLE OF VARCHAR2(2000)
  INDEX BY BINARY_INTEGER;
TYPE DATE_TABLE IS TABLE OF DATE
  INDEX BY BINARY_INTEGER;
TYPE BLOB_TABLE IS TABLE OF BLOB
  INDEX BY BINARY_INTEGER;
TYPE CLOB_TABLE IS TABLE OF CLOB
  INDEX BY BINARY_INTEGER;
TYPE BFILE_TABLE IS TABLE OF BFILE
  INDEX BY BINARY_INTEGER;
```

Еще одно различие между BIND_ARRAY и BIND_VARIABLE заключается в том, что в BIND_ARRAY можно указывать диапазон привязываемых значений таблицы PL/SQL. Поэтому BIND_ARRAY переопределяется типом и индексами таблицы:

```
PROCEDURE BIND_ARRAY(c IN INTEGER,
                     name IN VARCHAR2,
```

*table_variable* IN *table_datatype*);

PROCEDURE BIND_ARRAY(*c* IN INTEGER,
*name* IN VARCHAR2,
*table_variable* IN *table_datatype*,
*index1* IN INTEGER,
*index2* IN INTEGER);

Параметры BIND_ARRAY описаны в таблице 15.9.

**ТАБЛИЦА 15.9.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *c* | INTEGER | Номер курсора |
| *name* | VARCHAR2 | Имя заполнителя в операторе |
| *tabie_variable* | Один из типов NUMBER_TABLE, VARCHAR2_TABLE, DATE_TABLE, BLOB_TABLE, CLOB_TABLE, BFILE_TABLE | Таблица PL/SQL, в которой содержатся привязываемые данные |
| *index1* | INTEGER | Индекс элемента таблицы, являющегося нижней границей привязываемого диапазона |
| *index2* | INTEGER | Индекс элемента таблицы, являющегося верхней границей привязываемого диапазона |

Если *index1* и *index2* не указаны — будет использоваться вся таблица PL/SQL. Если к одному и тому же оператору (к различным заполнителям) привязывается несколько разных по размеру таблиц PL/SQL, то будет использоваться массив наименьшего размера. Пример работы с процедурами BIND_ARRAY и DEFINE_ARRAY приведен в конце следующего раздела.

## DEFINE_ARRAY

Процедура DEFINE_ARRAY выполняет операции, схожие с операциями процедуры DEFINE_COLUMN, только аргументами DEFINE_ARRAY являются таблицы PL/SQL. Типы таблиц те же, что и для процедуры BIND_ARRAY:

PROCEDURE DEFINE_ARRAY(*c* IN INTEGER,
*position* IN INTEGER,
*table_variable* IN *table_datatype*,
*cnt* IN INTEGER,
*indx* IN INTEGER);

Параметры описаны в таблице 15.10.

**ТАБЛИЦА 15.10.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *c* | INTEGER | Номер курсора. |
| *position* | INTEGER | Позиция данного столбца в списке выбора. Позиция первого столбца равна 1. |
| *table_variable* | Один из типов NUMBER_TABLE, VARCHAR2_TABLE, DATE_TABLE, BLOB_TABLE, CLOB_TABLE, BFILE_TABLE | Табличная переменная PL/SQL, в которой будут размещены данные при выполнении последующей функции FETCH_ROWS. |
| *cnt* | INTEGER | Максимальное число строк, считываемых при каждом вызове FETCH_ROWS. |
| *indx* | INTEGER | Начальный индекс результирующего набора данных. В table_variable будет записано не более cnt строк, начиная с индекса indx. |

Пример использования процедур BIND_ARRAY и DEFINE_ARRAY приведен в следующем разделе.

## Пример обработки массивов

Процедура CopyRegisteredStudents копирует таблицу registered_students. Хотя это можно выполнить и без DBMS_SQL, данный пример иллюстрирует концепцию обработки массивов данных.

```
-- Этот пример содержится в файле copyrs.sql.
CREATE OR REPLACE PROCEDURE CopyRegisteredStudents(
  p_NewName IN VARCHAR2) AS
  /* Создает новую таблицу с именем, указанным в p_NewName, той же самой
     структуры, что и таблица registered_students. Затем содержимое
     registered_students считывается в таблицы PL/SQL и
     вносится в новую таблицу. */
  v_BatchSize CONSTANT INTEGER := 5;
  v_IDs DBMS_SQL.NUMBER_TABLE;
  v_Departments DBMS_SQL.VARCHAR2_TABLE;
  v_Courses DBMS_SQL.NUMBER_TABLE;
  v_Grades DBMS_SQL.VARCHAR2_TABLE;

  v_Cursor1 INTEGER;
  v_Cursor2 INTEGER;
  v_ReturnCode INTEGER;
  v_NumRows INTEGER;
  v_SQLStatement VARCHAR2(200);
  v_SelectStmt VARCHAR2(200);
  v_InsertStmt VARCHAR2(200);
BEGIN
  v_Cursor1 := DBMS_SQL.OPEN_CURSOR;
  v_Cursor2 := DBMS_SQL.OPEN_CURSOR;

  -- Сначала удалим новую таблицу. Ошибку ORA-942 (таблица или
  -- представление не существует) будем игнорировать.
  BEGIN
      v_SQLStatement := 'DROP TABLE ' || p_NewName;
      DBMS_SQL.PARSE(v_Cursor1, v_SQLStatement, DBMS_SQL.V7);
      v_ReturnCode := DBMS_SQL.EXECUTE(v_Cursor1);
  EXCEPTION
      WHEN OTHERS THEN
        IF SQLCODE != -942 THEN
          RAISE;
        END IF;
  END;

  -- Создадим новую таблицу.
  v_SQLStatement := 'CREATE TABLE ' || p_NewName || '(';
  v_SQLStatement := v_SQLStatement || 'student_id NUMBER(5),';
  v_SQLStatement := v_SQLStatement || 'department CHAR(3),';
  v_SQLStatement := v_SQLStatement || 'course NUMBER(3),';
  v_SQLStatement := v_SQLStatement || 'grade CHAR(1))';
  DBMS_SQL.PARSE(v_Cursor1, v_SQLStatement, DBMS_SQL.V7);
  v_ReturnCode := DBMS_SQL.EXECUTE(v_Cursor1);

  -- Выполним грамматический разбор операторов выбора и операторов ввода.
  v_SelectStmt := 'SELECT * FROM registered_students';
  v_InsertStmt := 'INSERT INTO ' || p_NewName || ' VALUES ';
```

```
    v_InsertStmt := v_InsertStmt || '(:ID, :department, :course, :grade)';
    DBMS_SQL.PARSE(v_Cursor1, v_SelectStmt, DBMS_SQL.V7);
    DBMS_SQL.PARSE(v_Cursor2, v_InsertStmt, DBMS_SQL.V7);

    -- Чтобы указать выходные переменные для операции выбора,
    -- воспользуемся процедурой DEFINE_ARRAY.
    DBMS_SQL.DEFINE_ARRAY(v_Cursor1, 1, v_IDs, v_BatchSize, 1);
    DBMS_SQL.DEFINE_ARRAY(v_Cursor1, 2, v_Departments, v_BatchSize, 1);
    DBMS_SQL.DEFINE_ARRAY(v_Cursor1, 3, v_Courses, v_BatchSize, 1);
    DBMS_SQL.DEFINE_ARRAY(v_Cursor1, 4, v_Grades, v_BatchSize, 1);

    -- Выполним оператор SELECT.
    v_ReturnCode := DBMS_SQL.EXECUTE(v_Cursor1);

    -- Цикл выборки. С каждым вызовом FETCH_ROWS будет считываться
    -- v_BatchSize строк данных. Цикл закончится, когда FETCH_ROWS
    -- вернет значение, меньшее, чем v_BatchSize.
    LOOP
        v_NumRows := DBMS_SQL.FETCH_ROWS(v_Cursor1);
        DBMS_SQL.COLUMN_VALUE(v_Cursor1, 1, v_IDs);
        DBMS_SQL.COLUMN_VALUE(v_Cursor1, 2, v_Departments);
        DBMS_SQL.COLUMN_VALUE(v_Cursor1, 3, v_Courses);
        DBMS_SQL.COLUMN_VALUE(v_Cursor1, 4, v_Grades);

        -- Если это последняя операция считывания, FETCH_ROWS вернет число
        -- строк меньше v_BatchSize. Однако могут остаться выбранные строки,
        -- и их число равно v_NumRows. Поэтому для их привязки нужно
        -- воспользоваться v_NumRows вместо v_BatchSize. Нужно проверить
        -- особую ситуацию, когда v_NumRows = 0. Это значит, что последняя
        -- операция считывания вернула все оставшиеся строки
        -- и поэтому цикл можно завершить.
        IF v_NumRows = 0 THEN
          EXIT;
        END IF;

        -- Чтобы указать входные переменные для операции ввода,
        -- воспользуемся процедурой BIND_ARRAY.
        -- Использоваться будут только элементы 1..v_NumRows.
        DBMS_SQL.BIND_ARRAY(v_Cursor2, ':ID', v_IDs, 1, v_NumRows);
        DBMS_SQL.BIND_ARRAY(v_Cursor2, ':department', v_Departments, 1,
                            v_NumRows);
        DBMS_SQL.BIND_ARRAY(v_Cursor2, ':course', v_Courses, 1, v_NumRows);
        DBMS_SQL.BIND_ARRAY(v_Cursor2, ':grade', v_Grades, 1, v_NumRows);

        -- Выполним оператор INSERT.
        v_ReturnCode := DBMS_SQL.EXECUTE(v_Cursor2);
          -- Условие выхода из цикла. Обратите внимание: обработка данных
          -- в цикле завершена до этой проверки.
          EXIT WHEN v_NumRows < v_BatchSize;
        END LOOP;
    COMMIT;
    DBMS_SQL.CLOSE_CURSOR(v_Cursor1);
    DBMS_SQL.CLOSE_CURSOR(v_Cursor2);
END CopyRegisteredStudents;
```

**ВНИМАНИЕ** Семантика процедуры FETCH_ROWS немного меняется при считывании массивов. Оператор EXIT WHEN цикла выборки выполняется только после ввода строк в новую таблицу, а не сразу после оператора FETCH_ROWS (сравните данный цикл выборки с циклом в процедуре **DynamicQuery,** рассмотренной выше). Это обязательно, поскольку после считывания функцией FETCH_ROWS числа строк, меньшего чем запрошенное число, могут остаться строки, которые нужно обработать. Т.е. может получиться так, что в базе данных все строки считаны, но не все еще обработаны.

## Описание списка выбора

Если оператор SELECT является полностью динамическим, то во время компиляции программы нет никакой информации о возвращаемых столбцах. Ее можно получить с помощью процедуры DESCRIBE_COLUMNS, которая вызывается в любое время после грамматического разбора запроса. Ниже приведен синтаксис этой процедуры и используемых в ней типов:

```
TYPE DESC_REC IS RECORD (
   col_type BINARY_INTEGER := 0;
   col_max_len BINARY_INTEGER := 0;
   col_name VARCHAR2(32) := '';
   col_name_len BINARY_INTEGER := 0;
   col_schema_name VARCHAR2(32) := '';
   col_schema_name_len BINARY_INTEGER := 0;
   col_precision BINARY_INTEGER := 0;
   col_scale BINARY_INTEGER := 0;
   col_charsetid BINARY_INTEGER := 0;
   col_charsetform BINARY_INTEGER := 0;
   col_null_ok BOOLEAN := TRUE);

TYPE DESC_TAB IS TABLE OF DESC_REC
   INDEX BY BINARY_INTEGER;

PROCEDURE DBMS_SQL.DESCRIBE_COLUMNS(c IN INTEGER,
                                    col_cnt OUT INTEGER,
                                    desc_t OUT DESC_TYPE);
```

Параметры DESCRIBE_COLUMNS описаны в таблице 15.11.

**ТАБЛИЦА 15.11.**

| Параметр | Тип | Описание |
|---|---|---|
| *c* | INTEGER | Номер курсора |
| *col_cnt* | INTEGER | Число столбцов в списке выбора |
| *desc_t* | DESC_TYPE | Таблица PL/SQL, в которой содержится описание столбцов |

Описание полей типа DESC_REC приведено в таблице 15.12.

**ТАБЛИЦА 15.12.**

| Поле | Тип | Описание |
|---|---|---|
| col_type | BINARY_INTEGER | Код типа описываемого столбца (коды приведены в таблице 15.13) |
| col_max_len | BINARY_INTEGER | Максимальный размер столбца |
| col_name | VARCHAR2(32) | Имя столбца |
| col_name_len | BINARY_INTEGER | Длина имени столбца |

**ТАБЛИЦА 15.12 (продолжение)**

| Поле | Тип | Описание |
|---|---|---|
| col_schema_name | VARCHAR2(32) | Имя схемы, в которой объявлен тип столбца (применяется только для объектных типов) |
| col_schema_name_len | BINARY_INTEGER | Длина имени схемы |
| col_precision | BINARY_INTEGER | Точность столбца (применяется только для столбцов типа NUMBER) |
| col_scale | BINARY_INTEGER | Масштаб столбца (применяется только для столбцов типа NUMBER) |
| col_charsetid | BINARY_INTEGER | Идентификатор набора символов столбца |
| col_charsetform | BINARY_INTEGER | Форма набора символов столбца |
| col_null_ok | BOOLEAN | TRUE, если в столбце разрешаются NULL-значения, в противном случае — FALSE |

**ТАБЛИЦА 15.13. Внутренние коды типов данных**

| Код | Описание |
|---|---|
| 1 | VARCHAR2 |
| 2 | NUMBER |
| 8 | LONG |
| 12 | DATE |
| 23 | RAW |
| 24 | LONG RAW |
| 69 | ROWID |
| 96 | CHAR |
| 106 | MLSLABEL |
| 112 | CLOB |
| 113 | BLOB |
| 114 | BFILE |

В качестве примера использования DESCRIBE_COLUMNS рассмотрим процедуру DescribeTable, в которую вводится имя таблицы, а на выходе на экран выдается (с помощью DBMS_OUTPUT) описание этой таблицы.

```
-- Этот пример содержится в файле descrtab.sql.
CREATE OR REPLACE PROCEDURE DescribeTable(p_Table IN VARCHAR2) AS
  V_Cursor          INTEGER;
  V_SQLStatement    VARCHAR2(100);
  V_DescribeInfo    DBMS_SQL.DESC_TAB;
  V_DRec            DBMS_SQL.DESC_REC;
  V_ReturnCode      INTEGER;
  V_NumColumns      INTEGER;

  FUNCTION ConvertDatatype (v_Datatype IN NUMBER)
      RETURN VARCHAR2 IS
      v_Output VARCHAR2(20);
```

```
    BEGIN
        SELECT DECODE(v_Datatype, 1,  'VARCHAR2',
                                  2,  'NUMBER',
                                  8,  'LONG',
                                 12,  'DATE',
                                 23,  'RAW',
                                 24,  'LONG RAW',
                                 69,  'ROWID',
                                 96,  'CHAR',
                                106,  'MLSLABEL',
                                112,  'CLOB',
                                113,  'BLOB',
                                114,  'BFILE')
          INTO v_Output
          FROM dual;

        RETURN v_Output;
    END ConvertDatatype;

    BEGIN
        v_Cursor := DBMS_SQL.OPEN_CURSOR;

        -- Проведем грамматический разбор оператора SELECT для этой
        -- таблицы. Выполнять данный оператор не нужно.
          v_SQLStatement := 'SELECT * FROM ' || p_Table;
          DBMS_SQL.PARSE(v_Cursor, v_SQLStatement, DBMS_SQL.V7);

          -- Опишем оператор, получив в результате описание таблицы.
          DBMS_SQL.DESCRIBE_COLUMNS(v_Cursor, v_NumColumns, v_DescribeInfo);

        -- Информация выходного заголовка.
        DBMS_OUTPUT.PUT_LINE('Description of ' || p_Table || ':');
        DBMS_OUTPUT.PUT('Column Name Datatype Length Precision Scale ');
        DBMS_OUTPUT.PUT_LINE('Null?');
        DBMS_OUTPUT.PUT('----------- ------ ---- ------ ------ ');
        DBMS_OUTPUT.PUT_LINE('------');

        -- Последовательно просмотрим все столбцы и выведем описание каждого
        -- из них.
        FOR v_Col IN 1..v_NumColumns LOOP
          v_DRec := v_DescribeInfo(v_Col);
          DBMS_OUTPUT.PUT(RPAD(v_DRec.col_name, 16));
          DBMS_OUTPUT.PUT(RPAD(ConvertDatatype(v_DRec.col_type), 9));
          DBMS_OUTPUT.PUT(RPAD(v_DRec.col_max_len, 7));
          DBMS_OUTPUT.PUT(RPAD(v_DRec.col_precision, 10));
          DBMS_OUTPUT.PUT(RPAD(v_DRec.col_scale, 6));
          IF v_DescribeInfo(v_Col).col_null_ok THEN
            DBMS_OUTPUT.NEW_LINE;
          ELSE
            DBMS_OUTPUT.PUT_LINE('NOT NULL');
          END IF;
        END LOOP;

END DescribeTable;
```

# Другие процедуры

В модуле DBMS_SQL содержатся и другие процедуры, используемые для считывания данных типа LONG и для расширенной обработки ошибок.

## Считывание данных типа LONG

PL/SQL 2.2 ... и ВЫШЕ

В поле столбца типа LONG может храниться до 2 Гбайт данных, а в переменной PL/SQL LONG – только 32 Кбайт, поэтому в DBMS_SQL предусмотрена возможность считывания данных типа LONG в сегменты большего размера, которые гораздо удобнее обрабатывать. Это осуществляется с помощью двух процедур: DEFINE_COLUMN_LONG и COLUMN_VALUE_LONG, доступных начиная с PL/SQL версии 2.2.

Данные процедуры используются так же, как и процедуры DEFINE_COLUMN и COLUMN_VALUE, но COLUMN_VALUE_LONG обычно вызывается в цикле, что позволяет считать все фрагменты данных.

### DEFINE_COLUMN_LONG

Описание DEFINE_COLUMN_LONG выглядит следующим образом:

```
PROCEDURE DEFINE_COLUMN_LONG(c IN INTEGER,
                             position IN INTEGER);
```

Параметры процедуры описаны в таблице 15.14.

**ТАБЛИЦА 15.14.**

| Параметр | Тип | Описание |
|---|---|---|
| *c* | INTEGER | Идентификационный номер курсора. Предварительно должен быть открыт курсор и выполнен грамматический разбор запроса, в котором указан столбец типа LONG. Все заполнители должны быть привязаны. |
| *position* | INTEGER | Относительная позиция пункта LONG в списке выбора. Позиция первого пункта списка равна 1. |

### COLUMN_VALUE_LONG

Описание COLUMN_VALUE_LONG выглядит следующим образом:

```
PROCEDURE COLUMN_VALUE_LONG(c IN INTEGER,
                            position IN INTEGER,
                            length IN INTEGER,
                            offset IN INTEGER,
                            value OUT VARCHAR2,
                            value_length OUT INTEGER);
```

Параметры процедуры описаны в таблице 15.15.

Целесообразнее всего начинать считывание столбца LONG с первой позиции, а не с середины или конца. С каждым вызовом COLUMN_VALUE_LONG возвращается фрагмент значения LONG длиной *length* байт, начиная со смещения *offset*. Этот фрагмент записывается в *value*, а его размер – в *value_length*. Если value_length меньше, чем *length*, значит, считаны все данные. Пример использования COLUMN_VALUE_LONG приведен в разделе "PL/SQL в работе".

**ТАБЛИЦА 15.15.**

| Параметр | Тип | Описание |
|---|---|---|
| *c* | INTEGER | Идентификационный номер курсора. Предварительно курсор должен быть открыт, запрос — грамматически разобран, входные заполнители — привязаны, столбец LONG — описан с помощью DEFINE_COLUMN_LONG, а другие столбцы — с помощью DEFINE_COLUMN. Запрос должен быть обработан, а столбцы — считаны. |
| *position* | INTEGER | Относительная позиция пункта LONG в списке выбора. Позиция первого пункта списка равна 1. |
| *length* | INTEGER | Размер данного сегмента в байтах. |
| *offset* | INTEGER | Смещение данного фрагмента в байтах. Размер фрагмента будет равен length байт. Нулевое *смещение* обозначает первый фрагмент. |
| *value* | VARCHAR2 | Выходная переменная, в которую будет записан данный фрагмент. |
| *value_length* | INTEGER | Фактический размер возвращаемого фрагмента. Когда *value_length* меньше *length*, считывается весь столбец. |

# Дополнительные функции по обработке ошибок

Эти функции предосталяют дополнительные возможности для обработки ошибок курсоров DBMS_SQL, а также для сообщения о таких ошибках. Некоторые функции применяются только в особых случаях, о чем сообщается при их описании.

## LAST_ERROR_POSITION

Эта функция возвращает смещение в байтах, указывающее место в SQL-операторе, где произошла ошибка. Функцию LAST_ERROR_POSITION удобно использовать при обработке ошибок грамматического разбора, например ошибки ORA-911 (неверный символ). Описание этой функции выглядит следующим образом:

```
FUNCTION LAST_ERROR_POSITION RETURN INTEGER;
```

Эту функцию нужно вызывать после вызова процедуры PARSE DBMS_SQL, перед вызовом других процедур и функций DBMS_SQL, и только в том случае, когда вызов PARSE завершился неуспешно.

## LAST_ROW_COUNT

Эта функция возвращает суммарное число строк, считанных в курсоре до ее вызова, и в этом она похожа на курсорный атрибут %ROWCOUNT. Описание данной функции выглядит следующим образом:

```
FUNCTION LAST_ROW_COUNT RETURN INTEGER;
```

Ее нужно вызвать после FETCH_ROWS или EXECUTE_AND_FETCH. Если LAST_ROW_COUNT вызывается после EXECUTE, но до первого вызова FETCH_ROWS, то она всегда возвращает ноль, поскольку на данный момент ни одной строки еще не считано.

## LAST_ROW_ID

Эта функция возвращает идентификатор строки, обработанной последней. Описание функции LAST_ROW_ID выглядит следующим образом:

```
FUNCTION LAST_ROW_ID RETURN ROWID;
```

Ее нужно вызвать следом за FETCH_ROWS или EXECUTE_AND_FETCH. После выполнения запроса значение этой функции не определено, поскольку на данный момент ни одной строки еще не считано. Значение не определено и после исполнения оператора DML, так как в нем может обрабатываться несколько строк.

## LAST_SQL_FUNCTION_CODE

Эта функция возвращает код функции для SQL-оператора, исполняемого в данный момент. Коды функций изменяются от версии к версии PL/SQL. Перечень кодов функций для конкретной версии можно найти в руководстве Programmer's Guide to the Oracle Call Interfaces. Синтаксис этой функции таков:

```
FUNCTION LAST_SQL_FUNCTION_CODE RETURN INTEGER;
```

Эту функцию можно вызывать сразу после вызова EXECUTE. В другое время возвращаемое значение не определено.

### IS_OPEN

*Это логическая функция. Она возвращает* TRUE, *когда курсор, обозначенный как c, уже открыт, и* FALSE – *если он не открыт. Описание этой функции выглядит следующим образом:*

```
FUNCTION IS_OPEN(c IN INTEGER) RETURN BOOLEAN;
```

Применение функции IS_OPEN при обработке ошибок делает этот процесс более надежным:

```
DECLARE
  v_CursorID  INTEGER;
  ...
BEGIN;
  ...
EXCEPTION
  WHEN  OTHERS  THEN
      IF  DBMS_SQL.IS_OPEN(v_CursorID)  THEN
         DBMS_SQL.CLOSE_CURSOR(v_CursorID);
      END  IF;
  RAISE;
END;
```

# PL/SQL в работе: запись данных LONG в файл

**PL/SQL 2.3 ... и ВЫШЕ**

При работе с Oracle7 в строке, имеющей тип данных LONG, можно хранить до 2 Гбайт символьных данных. Однако размер самой большой переменной PL/SQL (типа VARCHAR2, CHAR или LONG) равен 32 Кбайт, и раньше это налагало определенные ограничения на объем считываемой информации.

Теперь в PL/SQL версии 2.2 можно за один раз считать все содержимое столбца LONG. При этом информация считывается порциями (chunks), как в вызове **oflng** OCI V7, когда для указания точки начала считывания используется параметр offset (смещение).

Этот пример показывает, как можно считать все содержимое столбца LONG (с помощью модуля DBMS_SQL) и затем записать полученную информацию в файл. Для такой записи применяется модуль UTL_FILE, поставляемый с PL/SQL версии 2.3 (см. главу 18).

**▼ ВНИМАНИЕ**

**Чтобы этот пример заработал, в параметре UTL_FILE_DIR файла init.ora нужно указать тот же каталог, что и в переменной location (местоположение) функции dump_doc.**

Предположим, что создана таблица ASCII_DOCS, имеющая следующую структуру:

```
Name                           Null?    Type
-----------------------------  -------  --------
ID                                      NUMBER
DOCUMENT                                LONG
```

ID – *это первичный ключ, применяемый для определения того документа, который необходимо извлечь.* Функция, описанная ниже, имеет 2 параметра: первичный ключ записываемого документа и имя файла для сохранения этого документа. Функция создает SQL-оператор, выполняет его грамматический разбор и привязывает базовые переменные, а затем исполняет оператор. После этого для считывания нужной строки один раз вызывается FETCH_ROWS (если требуется записать несколько документов/строк, можно организовать цикл для считывания данных). Затем с различными смещениями (**cur_pos**) несколько раз вызывается COLUMN_VALUE_LONG; при этом смещение показывает, откуда начинать считывание данных LONG. Потом считанное записывается в файл, и процесс продолжается до тех пор, пока COLUMN_VALUE_LONG не вернет ноль строк (т.е. пока параметр chunk_size_returned не станет равным нулю). После этого файл закрывается и устанавливается флаг успешного завершения операции. Если возникают неожиданные исключительные ситуации, файл закрывается и устанавливается флаг неуспешного завершения.

Размер для порции данных в 254 байт был выбран не случайно: он удобен для отображения данных LONG модуля DBMS_OUTPUT в SQL*Plus. Тем не менее можно установить и другой размер, скажем, 2000 байт или даже 32 Кбайт, и это даже ускорит работу. Кроме того, обратите внимание, что PUT и

PUT_LINE используются при записи каждой порции информации. При этом в выходные данные не добавляются лишние символы возврата каретки; сохранены все символы возврата каретки исходного документа.

Ниже приведен текст функции **dump_doc**.

```
-- Этот пример содержится в файле dumplong.sql.
CREATE OR REPLACE FUNCTION dump_doc(docid IN NUMBER,
                                    filename IN VARCHAR2)
                            RETURN VARCHAR2
IS
   Data_chunk             VARCHAR2(254);
   Chunk_size             NUMBER:=254;
   Chunk_size_returned    NUMBER;
   -- Зададим местоположение каталога, в котором будет находиться файл.
   Location               VARCHAR2(20)  := 'c:\temp';
   Mycursor               NUMBER;
   Stmt                   VARCHAR2(1024);
   Cur_pos                NUMBER:=0;
   Rows                   NUMBER;
   Dummy                  NUMBER;
   File_handle            UTL_FILE.FILE_TYPE;
   Status                 VARCHAR2(50);
BEGIN
   -- Откроем файл на запись.
   file_handle := utl_file.fopen(location, filename, 'w');
   -- Свяжем базовую переменную doctoget с параметром docid,
   -- который передается функции.
   stmt := 'SELECT DOCUMENT FROM ASCII_DOCS WHERE ID = :doctoget';
   mycursor := dbms_sql.open_cursor;
   dbms_sql.parse(mycursor, stmt, dbms_sql.v7);
   dbms_sql.bind_variable(mycursor, ':doctoget', docid);

   -- Для данного первичного ключа считаем лишь одну строку, так как
   -- предполагается, что весь документ хранится в одной строке.
   dbms_sql.define_column_long(mycursor,1);
   dummy := dbms_sql.execute(mycursor);
   rows := dbms_sql.fetch_rows(mycursor);

   loop
      -- Будем считывать порции данных long до тех пор, пока не
      -- считаем все.
      dbms_sql.column_value_long(mycursor,
                                 1,
                                 chunk_size,
                                 cur_pos,
                                 data_chunk,
                                 chunk_size_returned);
      utl_file.put(file_handle, data_chunk);
      cur_pos := cur_pos + chunk_size;
      exit when chunk_size_returned = 0;
   end loop;
   dbms_sql.close_cursor(mycursor);
   utl_file.fclose(file_handle);
   return('Success');
```

```
EXCEPTION
      WHEN OTHERS THEN
         utl_file.fclose(file_handle);
         return ('Failure');
END dump_doc;
```

# Привилегии и DBMS_SQL

При использовании модуля DBMS_SQL возникает ряд вопросов, связанных с привилегиями. В частности, какие привилегии необходимы для выполнения самого модуля DBMS_SQL и как он взаимодействует с ролями.

## Привилегии, необходимые для работы с DBMS_SQL

Для работы с DBMS_SQL необходимо иметь привилегию EXECUTE на этот модуль. Владельцем DBMS_SQL, как и других модулей DBMS, является пользователь SYS. В инсталляционном файле-сценарии, с помощью которого создается этот модуль, привилегия EXECUTE на DBMS_SQL обычно предоставляется роли PUBLIC, поэтому все пользователи могут работать с этим модулем. Данную привилегию можно отменить у PUBLIC и предоставить ее только некоторым пользователям.

Как правило, процедуры выполняются в рамках набора привилегий своих владельцев. В данном случае владельцем DBMS_SQL является SYS, т.е. все команды, выполняемые с помощью этого модуля, должны были бы запускаться из схемы SYS, а это представляет серьезную угрозу безопасности информации. Для обеспечения необходимой защиты данных процедуры и функции модуля DBMS_SQL выполняются в рамках набора привилегий тех пользователей, которые их вызывают, а не SYS. Если пользователь, скажем, **UserA** вызывает процедуру **RecreateTempTable**, то таблица **temp_table** будет создана в схеме **UserA**, хотя владельцем DBMS_SQL является SYS.

## Роли и DBMS_SQL

Как было сказано в главе 7, внутри хранимых процедур все роли запрещены. Это относится и к модулю DBMS_SQL. Однако с его помощью можно выполнять произвольные команды, поэтому пользователь, вызывающий DBMS_SQL, должен иметь не только привилегию EXECUTE на сам модуль, но и привилегии на выполнение динамических команд. Более того, привилегии на выполнение динамических команд должны быть предоставлены явным образом, а не через роль. Внутри DBMS_SQL роли запрещены, поэтому привилегии будут недоступны.

Например, пользователям часто предоставляется роль RESOURCE, в состав которой входит системная привилегия CREATE TABLE. Предположим, что роль RESOURCE предоставлена пользователю **UserA** и поэтому он может задать такие команды:

```
DROP TABLE temp_table;
CREATE TABLE temp_table (num_col NUMBER, char_col VARCHAR2(50));
```

Однако, когда те же самые команды выполняются с помощью DBMS_SQL (например, в процедуре **RecreateTempTable**), роль запрещается и выдается сообщение об ошибке:

```
ORA-1031: insufficient privileges
```

(недостаточные привилегии. — *Прим. пер.*)

Таким образом, пользователю **UserA** необходимо предоставить привилегию CREATE TABLE явно.

▼ СОВЕТУЕМ

Когда при работе с DBMS_SQL выдается ошибка ORA-1031, проверьте SQL-оператор или блок PL/SQL, обрабатываемый этим модулем. Убедитесь, что пользователь, выполняющий оператор или блок, имеет соответствующие системные и объектные привилегии, предоставленные явно, а не через роль. Именно это чаще всего является причиной ошибки.

# Сравнение DBMS_SQL с другими динамическими средствами

В Oracle существует три различных инструментальных средства, с помощью которых можно выполнять динамическую обработку SQL-операторов и блоков PL/SQL: модуль DBMS_SQL, OCI и предкомпиляторы. Они выполняют одинаковые операции: открытие курсоров, грамматический разбор операторов, привязку входных переменных, описание выходных переменных, выполнение операторов, считывание данных и закрытие курсоров. Однако работа с динамическим SQL в каждом из интерфейсов выполняется по-разному.

Основные различия между этими интерфейсами приведены в таблице 15.16 и детально описаны в последующих разделах.

**ТАБЛИЦА 15.16. Сравнение методов динамической обработки**

| Метод | DBMS_SQL | Предкомпиляторы | OCI |
|---|---|---|---|
| Описание списка выбора | Доступен в версии 8.0 и выше | Доступен во всех версиях | Доступен во всех версиях |
| Обработка массивов | Доступен в версии 8.0 и выше | Доступен во всех версиях | Доступен во всех версиях |
| Считывание фрагментов данных LONG | Доступен в версии 2.2 и выше | Недоступен | Доступен во всех версиях |
| Введение/обновление фрагментов данных LONG | Недоступен | Недоступен | Доступен в версии 7.3 и выше |

## Описание списка выбора

Описание списка выбора запроса позволяет во время выполнения программы определить размер и тип данных каждого возвращаемого пункта этого списка. Средство описания списка выбора применяется в предкомпиляторах и OCI, но в модуле DBMS_SQL это средство доступно только начиная с версии 8.0 (процедура DESCRIBE_COLUMNS).

Даже при наличии процедуры DESCRIBE_COLUMNS средство описания пунктов выбора модуля DBMS_SQL менее эффективно, чем аналогичные средства других программных пакетов. Дело в том, что в настоящее время пользователи PL/SQL не имеют возможности динамически выделять память и отменять выделение памяти (как это делается с помощью функций **malloc** и **free** языка C). В предкомпиляторах и OCI после описания оператора часто динамически выделяются выходные переменные, размер которых соответствует объему ожидаемых данных. В программе же PL/SQL приходится имитировать это посредством таблиц и изменяемых массивов PL/SQL.

Тем не менее в PL/SQL можно получить информацию о столбцах и без использования процедуры DESCRIBE_COLUMNS. Сведения о типах и размере столбцов таблиц базы данных содержатся в представлении словаря данных **user_tab_columns**, поэтому можно обратиться из программы PL/SQL к конкретной таблице и определить структуру списка выбора.

## Обработка массивов

Как в OCI, так и в предкомпиляторах можно использовать интерфейс массивов Oracle (Oracle Array Interface). Такой метод позволяет вводить информацию в базу данных непосредственно из массивов языка C или считывать информацию из базы данных в такие массивы. При этом нагрузка на сеть снижается во много раз. Данное средство реализуется в PL/SQL 8.0 через процедуры BIND_ARRAY и DEFINE_ARRAY.

## Операции над фрагментами данных типа LONG

С помощью предкомпиляторов в настоящее время нельзя выполнять операции над фрагментами данных LONG, и только OCI позволяет это делать. Многие вызовы в модуле DBMS_SQL реализованы через вызовы эквивалентных операций OCI, в число которых входит процедура **oflng**. С ее помощью можно считывать фрагменты данных LONG; в DBMS_SQL ей эквивалентны процедуры DEFINE_COLUMN_LONG и FETCH_COLUMN_LONG.

Ввод и обновление фрагментов данных LONG в OCI более сложны, а в DBMS_SQL эти операции не поддерживаются.

▼ ВНИМАНИЕ Хотя в DBMS_LOB (и в предкомпиляторах) нельзя вводить и обновлять фрагменты данных LONG, с помощью модуля DBMS_LOB можно достаточно просто работать с данными типа LOB. (Более подробно об этом см. главу 21.)

## Различия в интерфейсах

В PL/SQL (даже в версии 8.0) до сих пор не создан интерфейс, обеспечивающий работу пользователей с переменными-указателями, поэтому вся информация сначала записывается в локальный буфер и только потом считывается с помощью процедур COLUMN_VALUE и VARIABLE_VALUE. В OCI и предкомпиляторах такие дополнительные вызовы не нужны, поскольку при работе с этими средствами можно передавать адреса программных переменных непосредственно на сервер. При исполнении оператора база данных записывает информацию прямо в программные переменные, поэтому буфер не нужен.

# Советы и рекомендации

В этом разделе предлагается несколько рекомендаций по работе с DBMS_SQL. Модуль DBMS_SQL является мощным программным средством, но он достаточно сложен. Автор рекомендует использовать DBMS_SQL лишь при необходимости. Когда приложение может быть разработано другим способом (например, с помощью курсорных переменных), это, возможно, будет наилучшим вариантом.

## Повторное использование курсоров

По мере возможности курсоры следует использовать повторно. Старайтесь избегать ненужных вызовов процедур OPEN_CURSOR и CLOSE_CURSOR. В одном и том же курсоре разрешается обрабатывать SQL-операторы различных видов, поэтому можно, используя один курсор, уменьшить расход ресурсов системы на повторное открытие и закрытие курсоров. Если один и тот же оператор обрабатывается повторно, не нужно снова выполнять его грамматический разбор, – просто еще раз выполните этот оператор.

## Полномочия

Внутри модульных процедур (в том числе и в DBMS_SQL) роли запрещены, и это может привести к выдаче непонятных сообщений об ошибках, например ORA-1031. (Более подробно об этом см. в разделе "Привилегии и DBMS_SQL".)

## Операции DDL и зависание

*Неосторожное использование DBMS_SQL для динамической обработки операторов* DDL может привести к зависанию системы. Так, например, при вызове модульной процедуры устанавливается ее блокировка, которая снимается только после завершения вызова. Если попытаться динамически удалить модуль в то время, когда еще один пользователь выполняет процедуру этого модуля, операция EXECUTE зависнет. Максимальная длительность периода зависания 5 мин.

# Итоги

В этой главе говорилось о модуле DBMS_SQL, с помощью которого можно динамически обрабатывать SQL-операторы и блоки PL/SQL в программах, написанных на PL/SQL. В зависимости от вида обрабатываемого оператора (запрос, оператор DDL, оператор DML или блок PL/SQL) используются те или иные процедуры модуля DBMS_SQL. Кроме того, было проведено сравнение DBMS_SQL с другими динамическими методами, применяемыми в OCI и предкомпиляторах, а также предложены советы и рекомендации по использованию динамического SQL при разработке программ.

# Глава 16

## *Взаимодействие между соединениями*

В дополнение к средствам чтения таблиц базы данных и записи в эти таблицы, в PL/SQL предлагаются два встроенных модуля, предназначенных для взаимодействия между соединениями, или сеансами (sessions). Это модули DBMS_PIPE и DBMS_ALERT. Их можно использовать для обмена сообщениями между сеансами, соединенными с одним и тем же экземпляром базы данных. Эти модули чрезвычайно удобны и выполняют множество полезных функций. В этой главе детально рассмотрен каждый из них и, кроме того, проведено их сравнение, чтобы пользователь смог выбрать модуль, более удобный для работы.

**PL/SQL 8.0 ... и ВЫШЕ**

Модуль Oracle8 Advanced Queuing, или Oracle/AQ (средство улучшенной организации очередей), описанный в главе 17, также можно использовать для связи между сеансами. Свойства Oracle/AQ схожи со свойствами обоих модулей DBMS_PIPE и DBMS_ALERT, но это средство является гораздо более мощной и сложной коммуникационной системой.

# DBMS_PIPE

Посредством модуля DBMS_PIPE реализуются *программные каналы базы данных* (database pipes), которые аналогичны программным каналам операционной системы Unix, но применяются исключительно в Oracle. Следовательно, канал базы данных не зависит от операционной системы и будет работать в любой среде, в которой функционирует Oracle. Различные сеансы, соединенные с одним и тем же экземпляром Oracle, могут посылать и получать сообщения по такому каналу. Канал могут использовать несколько сеансов, получающих сообщения, или *получателей* (readers), и несколько сеансов, посылающих сообщения, или *отправителей* (writers). Все эти сеансы могут находиться на разных машинах и работать в различных средах выполнения программ PL/SQL. Все, что требуется от сеансов, — соединиться с одним и тем же экземпляром Oracle, а также иметь возможность выполнять блоки PL/SQL.

Программные каналы *асинхронны* — они функционируют независимо от транзакций. После того как сообщение послано по каналу, вернуть его нельзя, даже если был выполнен откат той транзакции, которая выдала сообщение.

Отправитель упаковывает группу элементов данных в локальном буфере сообщений. Затем содержимое этого буфера посылается по каналу в буфер сообщений принимающего сеанса. Принимающий сеанс распаковывает буфер и получает в результате реальную информацию. Например, триггер **LogRSInserts** регистрирует операции ввода данных в таблицу **registered_students**. Информация о вносимых изменениях посылается по программному каналу, поэтому триггер в данном случае является отправителем.

```
-- Этот пример содержится в файле logrsins.sql.
CREATE OR REPLACE TRIGGER LogRSInserts
  BEFORE INSERT ON registered_students
  FOR EACH ROW
DECLARE
  v_Status INTEGER;
BEGIN

  /* Сначала упакуем в буфере описание операции. */
  DBMS_PIPE.PACK_MESSAGE('I');

  /* Упакуем информацию о текущем пользователе и временную метку. */
  DBMS_PIPE.PACK_MESSAGE(user);
  DBMS_PIPE.PACK_MESSAGE(sysdate);

  /* Упакуем новые значения. */
  DBMS_PIPE.PACK_MESSAGE(:new.student_ID);
  DBMS_PIPE.PACK_MESSAGE(:new.department);
  DBMS_PIPE.PACK_MESSAGE(:new.course);
  DBMS_PIPE.PACK_MESSAGE(:new.grade);

  /* Пошлем сообщение по каналу 'RSInserts'. */
  v_Status := DBMS_PIPE.SEND_MESSAGE('RSInserts');
```

```
    /* Если посылка неудачна, установим ошибку так, чтобы изменение не
       было внесено. */
    IF v_Status != 0 THEN
       RAISE_APPLICATION_ERROR(-20010, 'LogRSInserts trigger ' ||
         'couldn''t send the message, status = ' || v_Status);
    END IF;
END LogRSInserts;
```

Триггер является лишь одним из компонентов системы регистрации изменений. Этот триггер – отправитель для канала **RSInserts**, но нужен и получатель. С помощью Pro*C создадим получателя – программу, принимающую сообщения по каналу и записывающую изменения в файл операционной системы.

```
/* Этот пример содержится в файле rsinsert.pc. */
/* Эта программа принимает сообщения по каналу RSInserts и
   регистрирует их в файле. */

/* Файлы заголовков C и SQL. */
#include <stdio.h>
EXEC SQL INCLUDE sqlca;

EXEC SQL BEGIN DECLARE SECTION;
  /* Имя и пароль пользователя для соединения с базой данных. */
  char *v_Connect = "example/example";

  /* Переменные состояния, используемые в вызовах DBMS_PIPE. */
  int v_Status;
  VARCHAR v_Code[5];

  /* Переменные, посылаемые по каналу; именно их информация
     будет регистрироваться. */
  VARCHAR v_Userid[9];
  VARCHAR v_Changedate[10];
  int v_StudentID;
  VARCHAR v_Department[4];
  int v_Course;
  VARCHAR v_Grade[2];
  short v_Grade_ind;
EXEC SQL END DECLARE SECTION;

/* Указатель на файл регистрации. */
FILE *outfile;

void sqlerror();

int main() {

/* Установим обработку ошибок. */
EXEC SQL WHENEVER SQLERROR DO sqlerror();
/* Соединимся с базой данных. */
EXEC SQL CONNECT :v_Connect;
/* Откроем файл регистрации. */
outfile = fopen("rs.log", "w");
/* Основной цикл. Он будет прерван только в случае получения
   сообщения 'STOP' или в случае ошибки. */
for (;;) {
```

```
/* Подождем, пока сообщение, передаваемое по каналу 'RSInserts',
   не будет получено. Длительность задержки не указана, поэтому
   используется значение по умолчанию. */
EXEC SQL EXECUTE
    BEGIN
       :v_Status := DBMS_PIPE.RECEIVE_MESSAGE('RSInserts');
    END;
END-EXEC;

if (v_Status == 0) {
    /* Сообщение успешно считано. Теперь необходимо получить
       первый элемент данных, чтобы решить, что с ними делать. */
    v_Code.len = 5;
    EXEC SQL EXECUTE
       BEGIN
          DBMS_PIPE.UNPACK_MESSAGE(:v_Code);
       END;
    END-EXEC;
    v_Code.arr[v_Code.len] = '\0';

    if (!strcmp(v_Code.arr, "STOP")) {
       /* Получено сообщение об остановке. Прервем цикл. */
       break;
    }

    /* Считаем остальную часть сообщения, куда входит информация
       об идентификаторе пользователя, дата и новые значения. */
    v_Userid.len = 9;
    v_Changedate.len = 10;
    v_Department.len = 4;
    v_Grade.len = 2;
    EXEC SQL EXECUTE
       DECLARE
          v_ChangeDate DATE;
       BEGIN
          DBMS_PIPE.UNPACK_MESSAGE(:v_Userid);
          DBMS_PIPE.UNPACK_MESSAGE(v_ChangeDate);
          :v_Changedate := TO_CHAR(v_ChangeDate, 'DD-MON-YY');
          DBMS_PIPE.UNPACK_MESSAGE(:v_StudentID);
          DBMS_PIPE.UNPACK_MESSAGE(:v_Department);
          DBMS_PIPE.UNPACK_MESSAGE(:v_Course);
          DBMS_PIPE.UNPACK_MESSAGE(:v_Grade:v_Grade_ind);
       END;
    END-EXEC;

    /* Завершим строки символов нулями. */
    v_Userid.arr[v_Userid.len] = '\0';
    v_Changedate.arr[v_Changedate.len] = '\0';
    v_Department.arr[v_Department.len] = '\0';

    if (v_Grade_ind == -1)
       v_Grade.arr[0] = '\0';
    else
       v_Grade.arr[v_Grade.len] = '\0';
```

```
            /* Запишем данные в файл регистрации. */
            fprintf(outfile, "User: %s Timestamp: %s",
              v_Userid.arr, v_Changedate.arr);
            fprintf(outfile, " ID: %d Course: %s %d Grade: %s\n",
              v_StudentID, v_Department.arr, v_Course, v_Grade.arr);
          }
          else if (v_Status == 1) {
            /* Задержка вызова RECEIVE_MESSAGE. Вернемся назад и подождем. */
            continue;
          }
          else {
            /* Вызов RECEIVE_MESSAGE завершен с ошибкой. Зафиксируем это и
                выйдем из цикла. */
            printf("RECEIVE_MESSAGE Error! Status = %d\n", v_Status);
            EXEC SQL ROLLBACK WORK RELEASE;
            exit(1);
          }

        } /* Конец основного цикла. */

        /* Закроем файл. */
        Fclose(outfile);

        /* Отсоединимся от базы данных. */
        EXEC SQL COMMIT WORK RELEASE;
      }
      /* Функция обработки ошибок. Выведем ошибку на экран и выйдем из программы. */
      void sqlerror() {

        printf("SQL Error!\n");
        printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);

        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXEC SQL ROLLBACK RELEASE;
      }
```

Поскольку программный канал работает асинхронно, операция ввода данных в таблицу **registered_students** будет зарегистрирована даже в случае отката транзакции. Таким образом, этот триггер будет фиксировать как попытки изменения информации базы данных, так и реальные изменения.

## Посылка сообщений

Сообщения посылаются в два этапа. Сначала данные упаковываются в локальном буфере сообщений, а затем содержимое буфера посылается по каналу. Упаковка данных осуществляется с помощью процедуры PACK_MESSAGE, а посылка – с помощью функции SEND_MESSAGE.

### PACK_MESSAGE

Процедура PACK_MESSAGE переопределяется различными типами элементов данных. На принимающей стороне канала процедура UNPACK_MESSAGE также переопределяется различными типами. Описание процедуры PACK_MESSAGE выглядит следующим образом:

```
PROCEDURE PACK_MESSAGE(item IN VARCHAR2);
PROCEDURE PACK_MESSAGE(item IN NUMBER);
PROCEDURE PACK_MESSAGE(item IN DATE);
PROCEDURE PACK_MESSAGE_RAW(item IN RAW);
PROCEDURE PACK_MESSAGE_ROWID(item IN ROWID);
```

Размер буфера равен 4096 байт. Если общий размер упакованных данных превышает это значение, генерируется ошибка ORA-6558. Для каждого элемента данных в буфере отводится 1 байт для типа дан-

ных, 2 байта — для их длины, а также пространство для самих данных. Для завершения сообщения необходим один дополнительный байт. Размер буфера ограничен, поэтому по программному каналу нельзя пересылать данные типа LONG или LONG RAW.

### SEND_MESSAGE

Когда с помощью одного или нескольких вызовов PACK_MESSAGE локальный буфер сообщений заполнен данными, с помощью SEND_MESSAGE его содержимое посылается по каналу:

```
FUNCTION SEND_MESSAGE(pipename IN VARCHAR2,
                      timeout IN INTEGER DEFAULT MAXWAIT,
                      maxpipesize IN INTEGER DEFAULT 8192)
    RETURN INTEGER;
```

Если программный канал еще не создан, функция SEND_MESSAGE создает его. Каналы можно создавать и с помощью процедуры CREATE_PIPE, которая применяется в PL/SQL версии 2.2 и выше и описана в разделе "Создание программных каналов и управление ими" ниже. Параметры функции SEND_MESSAGE описаны в таблице 16.1.

**ТАБЛИЦА 16.1.**

| Параметр | Тип | Описание |
|---|---|---|
| *pipename* | VARCHAR2 | Имя канала. Имена каналов ограничены 30-ю символами и не учитывают регистра символов. Они начинаются с ORA$ и зарезервированы для использования базой данных. |
| *timeout* | INTEGER | Задержка в секундах. Если в силу различных причин (указанных возвращаемым кодом) сообщение не может быть послано, вызов возвращает число секунд после задержки. Значение по умолчанию DBMS_PIPE.MAXWAIT равно 86 400 000 с (1000 дней) |
| *maxpipesize* | INTEGER | Общий размер канала в байтах. По умолчанию устанавливается равным 8192 байт (два сообщения максимального размера). Суммарная длина всех сообщений в канале не может превышать этого значения (когда сообщение получается с помощью RECEIVE_MESSAGE, оно удаляется из канала). Максимальный размер канала является элементом его описания и сохраняется на все время его существования. В разных вызовах SEND_MESSAGE можно указывать разные значения параметра *maxpipesize*. Если новое значение больше существующего размера, то размер канала увеличивается. Если же новое значение меньше — сохраняется существующий размер. |

Значения, возвращаемые функцией SEND_MESSAGE, описаны в таблице 16.2.

**ТАБЛИЦА 16.2.**

| Возвращаемое значение | Смысл |
|---|---|
| 0 | Сообщение успешно послано, и при вызове RECEIVE_MESSAGE будет считано. |
| 1 | Вызов задержан. Это может произойти в том случае, если канал слишком заполнен и сообщение передаться не может, либо если нельзя установить блокировку данного канала. |
| 3 | Вызов прерван из-за внутренней ошибки. |

## Получение сообщений

Функции RECEIVE_MESSAGE и NEXT_ITEM_TYPE и процедура UNPACK_MESSAGE модуля DBMS_PIPE используются для приема посланных по каналу сообщений, а также для их распаковки и превращения в исходные элементы данных.

## RECEIVE_MESSAGE

Функция RECEIVE_MESSAGE обратна по отношению к функции SEND_MESSAGE. Она считывает сообщение из канала и помещает его в локальный буфер сообщений. После этого вызывается процедура UNPACK_MESSAGE, которая считывает данные уже из буфера. Описание функции RECEIVE_MESSAGE выглядит следующим образом:

```
FUNCTION RECEIVE_MESSAGE(pipename IN VARCHAR2,
                         timeout IN INTEGER DEFAULT MAXWAIT)
  RETURN INTEGER;
```

Обычно RECEIVE_MESSAGE вызывается из принимающей программы. Если сообщения не ожидаются, RECEIVE_MESSAGE блокируется до тех пор, пока не считает сообщение. Таким образом, сеанс-получатель будет находиться в режиме ожидания до посылки некоторого сообщения по каналу. Принимающие программы весьма похожи на демоны операционной системы Unix: они так же засыпают и просыпаются только тогда, когда получено сообщение по программному каналу. Параметры функции RECEIVE_MESSAGE описаны в таблице 16.3.

### ТАБЛИЦА 16.3.

| Параметр | Тип | Описание |
|---|---|---|
| *pipename* | VARCHAR2 | Имя канала. Это должно быть то же имя, что и в SEND_MESSAGE. На имя канала налагаются те же ограничения (длина менее 30 символов, регистр символов не учитывается). |
| *timeout* | INTEGER | Максимальное время ожидания сообщения в секундах. Как и в SEND_MESSAGE, значение по умолчанию — DBMS_PIPE.MAXWAIT (1000 дней). Если параметр timeout равен 0, то значение RECEIVE_MESSAGE возвращается немедленно со значением 0 (сообщение считано) или 1 (задержка) |

Коды, возвращаемые функцией RECEIVE_MESSAGE, описаны в таблице 16.4.

### ТАБЛИЦА 16.4.

| Возвращаемое значение | Смысл |
|---|---|
| 0 | Сообщение получено успешно. Оно было считано в локальный буфер и может быть распаковано с помощью UNPACK_MESSAGE. |
| 1 | Задержка. За время ожидания RECEIVE_MESSAGE по каналу не было передано ни одного сообщения. |
| 2 | Сообщение в канале слишком велико для буфера. Это внутренняя ошибка, которая обычно не происходит. |
| 3 | Вызов прерван из-за внутренней ошибки. |

## NEXT_ITEM_TYPE

Функция NEXT_ITEM_TYPE возвращает тип данных следующего элемента в буфере. На основании этого значения можно решить, в какую переменную записать данные. Если тип известен заранее, то вызывать NEXT_ITEM_TYPE необязательно. Информация об использовании этой функции приведена в разделе "Установление протокола связи" ниже. Функция NEXT_ITEM_TYPE описывается следующим образом:

```
FUNCTION NEXT_ITEM_TYPE RETURN INTEGER;
```

Возвращаемые коды описаны в таблице 16.5.

▼ ВНИМАНИЕ

В этой таблице приведены все типы, которые можно использовать при посылке данных по программным каналам базы данных. Данные, которые имеют типы, определяемые пользователями, например таблицы и записи PL/SQL или объектные типы Oracle8, посылать нельзя.

**ТАБЛИЦА 16.5.**

| Возвращаемое значение | Смысл |
|---|---|
| 0 | Элементов больше нет |
| 6 | NUMBER |
| 9 | VARCHAR2 |
| 11 | ROWID |
| 12 | DATE |
| 23 | RAW |

### UNPACK_MESSAGE

Процедура UNPACK_MESSAGE обратна по отношению к процедуре PACK_MESSAGE. Как и PACK_MESSAGE, эта процедура переопределяется типами считываемых элементов. Описание процедуры UNPACK_MESSAGE выглядит следующим образом:

```
PROCEDURE UNPACK_MESSAGE(item OUT VARCHAR2);
PROCEDURE UNPACK_MESSAGE(item OUT NUMBER);
PROCEDURE UNPACK_MESSAGE(item OUT DATE);
PROCEDURE UNPACK_MESSAGE_RAW(item OUT RAW);
PROCEDURE UNPACK_MESSAGE_ROWID(item OUT ROWID);
```

В *item* записывается элемент данных из буфера. Если в нем больше нет данных или если следующий элемент имеет не тот тип, который был запрошен, то генерируется ошибка Oracle ORA-6556 или ORA-6559. Перед установлением ошибки PL/SQL попытается преобразовать тип следующего элемента к нужному типу, используя формат преобразования, заданный по умолчанию (см. главу 2).

## Создание программных каналов и управление ими

Если канал не существует, то при первом указании его имени в SEND_MESSAGE он создается неявным образом. В PL/SQL версии 2.2 и выше каналы можно создавать и удалять явно с помощью процедур CREATE_PIPE и REMOVE_PIPE.

### Программные каналы и разделяемый пул

Программный канал представляет собой структуру данных, находящуюся в разделяемом пуле системной глобальной области (system global area, SGA) памяти. Поэтому он не занимает память, в которую могут помещаться другие объекты базы данных при их считывании с диска. Когда необходимо освободить дополнительное пространство разделяемого пула, те каналы, в которых нет непрочитанных сообщений, автоматически уничтожаются. Алгоритмом уничтожения программных каналов является алгоритм LRU (least recently used – использовавшийся раньше всех): уничтожается тот канал, который последний раз использовался раньше всех. (Более подробно о разделяемом пуле и о его влиянии на производительность системы см. в главе 22.)

Максимальный размер канала и, следовательно, размер структуры данных в разделяемом пуле задается параметром *maxpipesize* функций SEND_MESSAGE и CREATE_PIPE.

### Общие и частные программные каналы

Каналы, создаваемые неявно с помощью SEND_MESSAGE, называются *общими* (public). Любой пользователь, имеющий полномочие EXECUTE на модуль DBMS_PIPE или знающий имя канала, может посылать и принимать по нему сообщения. Доступ к *частному* (private) каналу ограничивается пользователем, создавшим его, а также хранимыми процедурами, выполняющимися на основании набора привилегий владельца канала, и пользователями, присоединившимися как SYSDBA или INTERNAL.

**PL/SQL 2.2 ... и ВЫШЕ**

В PL/SQL версий 2.0 и 2.1 применяются только общие каналы, создаваемые неявно. В PL/SQL версии 2.2 и выше можно уже создавать каналы явно, используя функцию CREATE_PIPE. Эта функция является единственным способом создания частного канала, однако при желании можно воспользоваться ею и для создания общего канала. Каналы, создаваемые посредством CREATE_PIPE, остаются в разделяемом пуле до тех пор, пока не будут явно удалены с помощью функции REMOVE_PIPE или пока экземпляр базы данных не будет останов-

лен. Такие каналы автоматически не уничтожаются в SGA. Описание CREATE_PIPE выглядит следующим образом:

```
FUNCTION CREATE_PIPE(pipename IN VARCHAR2,
                     maxpipesize IN INTEGER DEFAULT 8192,
                     private IN BOOLEAN DEFAULT TRUE)
RETURN INTEGER;
```

Если канал создан успешно, CREATE_PIPE возвращает ноль. Если канал уже существует, а текущий пользователь имеет привилегии, необходимые для доступа к нему, возвращается ноль и данные, находящиеся в канале, остаются без изменений. Если существует общий канал с указанным именем или если существует частный канал, имеющий то же имя и принадлежащий другому пользователю, устанавливается ошибка

```
ORA-23322: insufficient privilege to access pipe
```

(привилегии, недостаточные для доступа к каналу. — *Прим. пер.*)

и функция завершается неуспешно. Параметры этой функции описаны в таблице 16.6.

**ТАБЛИЦА 16.6.**

| Параметр | Тип | Описание |
|---|---|---|
| *pipename* | VARCHAR2 | Имя создаваемого канала. Имена каналов ограничены 30-ю символами. Имена, начинающиеся с ORA$, зарезервированы для внутреннего использования. |
| *maxpipesize* | INTEGER | Максимальный размер канала в байтах. Это тот же самый параметр, что и параметр, используемый в SEND_MESSAGE для неявного создания канала. Значение по умолчанию — 8192 байт. Если SEND_MESSAGE вызывается со значением, превышающим maxpipesize, размер канала увеличивается. Если же SEND_MESSAGE вызывается со значением меньшим, чем maxpipesize, сохраняется существующий размер. |
| *private* | BOOLEAN | TRUE, если канал должен быть частным; в противном случае FALSE. Значение по умолчанию — TRUE. Общие каналы создаются неявно функцией SEND_MESSAGE, поэтому, как правило, не имеет смысла устанавливать для параметра private значение FALSE. |

Каналы, создаваемые явно функцией CREATE_PIPE, удаляются с помощью функции REMOVE_PIPE. Если при этом в нем находятся какие-либо сообщения, они также удаляются. Эта функция — единственный способ (за исключением случая остановки экземпляра) удалить те каналы, которые были созданы явно. Описание функции REMOVE_PIPE выглядит следующим образом:

```
FUNCTION REMOVE_PIPE(pipename IN VARCHAR2)
  RETURN INTEGER;
```

Единственным параметром этой функции является имя удаляемого канала. Если канал существует, а текущий пользователь имеет на него привилегии, то канал удаляется и функция возвращает ноль. Если канал не существует, также возвращается ноль. Если канал существует, но текущий пользователь не имеет привилегий на доступ к нему, устанавливается ошибка ORA-23322 (как и в функции CREATE_PIPE).

### Процедура PURGE

Процедура PURGE уничтожает содержимое канала, не уничтожая его самого. Если канал создан неявно, то, поскольку теперь он пуст, его можно удалить из разделяемого пула в соответствии с алгоритмом LRU. Процедура PURGE повторно вызывает RECEIVE_MESSAGE, поэтому содержимое локального буфера сообщений может быть переписано. PURGE описывается следующим образом:

```
PROCEDURE PURGE(pipename IN VARCHAR2);
```

## Привилегии и безопасность

Для модуля DBMS_PIPE установлено три различных уровня безопасности. Первый — привилегия EXECUTE на сам модуль. По умолчанию при создании модуля эта привилегия не предоставляется всем пользователям. Поэтому обращаться к модулю DBMS_PIPE смогут только те пользователи, которые имеют системную привилегию EXECUTE ANY PROCEDURE. Чтобы разрешить доступ к модулю DBMS_PIPE

другим пользователям базы данных, с помощью оператора GRANT нужно предоставить им привилегии EXECUTE на этот модуль.

▼ ВНИМАНИЕ

Системная привилегия EXECUTE ANY PROCEDURE входит в роль DBA. Следовательно, те пользователи, кому предоставлена роль DBA (например, SYSTEM), смогут по умолчанию обращаться к модулю DBMS_PIPE из анонимных блоков PL/SQL. Но, поскольку эта привилегия предоставляется через роль, пользователи не смогут обращаться к модулю DBMS_PIPE из хранимых процедур или триггеров, так как в этих случаях роли запрещены (более подробно о ролях и об их взаимодействии с хранимыми подпрограммами см. главу 7).

Для обеспечения первого уровня безопасности информации можно предоставить привилегию EXECUTE на DBMS_PIPE только конкретным пользователям базы данных. После этого нужно создать свой собственный модуль для управления доступом к базовым каналам. Затем привилегию EXECUTE на этот модуль можно предоставить и другим пользователям.

Второй уровень безопасности — имя программного канала. Не зная этого имени, пользователи не смогут посылать и принимать сообщения по данному каналу. Имя можно выбрать произвольно или создать имя, уникальное для двух сеансов, взаимодействующих при помощи этого канала. Последний способ реализуется посредством функции UNIQUE_SESSION_NAME (он описан в разделе "Установление протокола связи" ниже).

### Частные каналы

PL/SQL 2.2 ... и ВЫШЕ

Самым надежным уровнем безопасности является использование частных каналов, которые можно создавать в PL/SQL 2.2. Частный канал доступен только пользователю, создавшему его, и пользователям, присоединившимся как SYSDBA или INTERNAL, поэтому доступ к нему существенно ограничен. Даже если пользователь имеет привилегию EXECUTE на модуль DBMS_PIPE и знает имя программного канала, при попытке доступа он получит сообщение об ошибке Oracle:

```
ORA-23322: insufficient privilege to access pipe
```

(недостаточно привилегий для доступа к каналу. — *Прим. пер.*)

Такая ошибка устанавливается в ситуациях, описанных в таблице 16.7. Обратите внимание, что ошибка возникает только при создании или удалении канала либо при попытке послать/получить сообщение. Другие вызовы в DBMS_PIPE не обращаются непосредственно к программным каналам.

**ТАБЛИЦА 16.7. Ситуации, в которых устанавливается ORA-23322**

| Процедура или функция | Ситуация, вызывающая ошибку ORA-23322 |
|---|---|
| CREATE_PIPE | Существует частный канал с тем же именем, но текущий пользователь не имеет привилегий на доступ к нему. Если этот пользователь имеет необходимые привилегии, CREATE_PIPE возвращает 0 и владелец канала не изменяется. |
| SEND_MESSAGE | Текущий пользователь не имеет привилегий на доступ к каналу. |
| RECEIVE_MESSAGE | Текущий пользователь не имеет привилегий на доступ к каналу. |
| REMOVE_PIPE | Текущий пользователь не имеет привилегий на доступ к каналу. Канал будет по-прежнему существовать, и все сообщения, содержащиеся в нем в данный момент, останутся без изменений. |

Лучшим способом использования частных каналов является создание хранимых процедур или модулей, которые, в свою очередь, вызывают модуль DBMS_PIPE. Поскольку хранимые подпрограммы выполняются на основании набора привилегий своего владельца, к частным каналам можно обращаться и из хранимых подпрограмм.

## Установление протокола связи

Программные каналы используются так же, как и другие низкоуровневые модули связи, такие как TCP/IP. Пользователь имеет возможность самостоятельно определить способы форматирования данных и их пересылки. Кроме того, он может решить, кто должен получить сообщение. Однако, чтобы воспользоваться гибкостью каналов связи, обратите особое внимание на рекомендации, предлагаемые в следующих разделах.

## Форматирование данных

Каждое сообщение, посылаемое по программному каналу, состоит из одного или нескольких элементов данных. Они заносятся в буфер сообщений с помощью процедуры PACK_MESSAGE, а затем все содержимое буфера посылается по каналу с помощью функции SEND_MESSAGE. На другой стороне канала буфер принимается функцией RECEIVE_MESSAGE, а элементы данных считываются с помощью функции NEXT_ITEM_TYPE и процедуры UNPACK_MESSAGE.

Операции, выполняемые принимающей программой, обычно зависят от содержимого получаемого сообщения. Например, программа Pro*C, посредством которой реализуется приемная часть триггера **LogRSInserts** (см. выше), использует первый элемент данных для форматирования сообщения, регистрируемого в файле. По сути, этот элемент данных является кодом операции, или командой, которая сообщает принимающей программе, как интерпретировать остальные данные. В зависимости от вида информации в сообщении могут содержаться данные различных типов или разное количество элементов данных.

▼ СОВЕТУЕМ Рекомендуется, помимо других команд, включать в сообщение команду STOP. Ее можно использовать, к примеру, для сообщения ожидающей программе о необходимости отсоединиться от базы данных и нормально завершить работу. Без такого сообщения ожидающая программа будет уничтожена операционной системой и/или базой данных, а это нежелательно. Использование команд STOP продемонстрировано в примере, приведенном в следующем разделе.

## Адресация данных

Один и тот же канал могут использовать несколько получателей и отправителей, однако принять сообщение может только один получатель. Более того, заранее неизвестно, какой именно получатель действительно примет сообщение. Поэтому рекомендуется адресовать сообщения конкретной программе-получателю. Это можно сделать, сгенерировав для канала уникальное имя, которое будет использоваться только двумя сеансами – одним получателем и одним отправителем. Для этого применяется функция UNIQUE_SESSION_NAME:

```
FUNCTION UNIQUE_SESSION_NAME RETURN VARCHAR2;
```

Каждый вызов UNIQUE_SESSION_NAME возвращает строку символов, максимальная длина которой равна 30 символам. Если эту функцию вызывает один и тот же сеанс базы данных, всякий раз будет выдаваться одна и та же строка, уникальная среди всех сеансов, соединенных с базой данных в конкретный момент времени. Однако, если сеанс отсоединяется, его имя может быть позже задействовано другим сеансом.

Строка символов, возвращаемая UNIQUE_SESSION_NAME, может быть использована в качестве имени канала, что обеспечит прием сообщения только одним получателем. Для этого существует специальный метод. По каналу с заранее определенным именем посылается начальное сообщение с указанием имени канала, по которому посылается ответ. Принимающая программа расшифровывает начальное сообщение и посылает ответ по новому каналу, который затем будет использоваться только этими двумя сеансами. С новым каналом будут работать только один получатель и один отправитель, поэтому устраняется неопределенность сеанса – получателя информации. Этот метод демонстрируется в примере, приведенном в следующем разделе.

# Пример

В этом примере описывается еще один вариант модуля Debug (см. главу 14). Как и в примере, приведенном в начале данной главы, одним из двух взаимодействующих сеансов является программа Pro*C. Другой сеанс – это модуль Debug.

## Debug.pc

```
/* Этот пример содержится в файле debug.pc. */
/* Эта программа является приемной частью того варианта модуля
   Debug, в котором используется DBMS_PIPE. Ее следует запускать не
   в том окне, в котором работает отлаживаемый сеанс PL/SQL. */

/* Файлы заголовков C и SQL. */
#include <stdio.h>
EXEC SQL INCLUDE sqlca;

EXEC SQL BEGIN DECLARE SECTION;
```

```
    /* Имя и пароль пользователя для соединения с базой данных. */
    char *v_Connect = "example/example";

    /* Переменные состояния, используемые в вызовах DBMS_PIPE. */
    int v_Status;
    VARCHAR v_Code[6];

    /* Переменные, посылаемые и принимаемые по каналам. */
    VARCHAR v_ReturnPipeName[31];
    VARCHAR v_Description[100];
    VARCHAR v_Value[100];
EXEC SQL END DECLARE SECTION;

/* Функция обработки ошибок. */
void sqlerror();

int main() {

    /* Установим обработку ошибок. */
    EXEC SQL WHENEVER SQLERROR DO sqlerror();

    /* Соединимся с базой данных. */
    EXEC SQL CONNECT :v_Connect;

    printf("Debug ready for input.\n");

    /* Основной цикл. Он будет прерван только в случае получения
       сообщения 'STOP' или в случае ошибки. */
    for (;;) {
        /* Подождем, пока сообщение, передаваемое по каналу 'DebugPipe',
           не будет получено. Длительность задержки не указана, поэтому
           используется значение по умолчанию. */
    EXEC SQL EXECUTE
        BEGIN
          :v_Status := DBMS_PIPE.RECEIVE_MESSAGE('DebugPipe');
        END;
    END-EXEC;
    if (v_Status == 0) {
    /* Сообщение успешно считано. Теперь необходимо получить
       первый элемент данных, чтобы решить, что с ними делать. */
    v_Code.len = 6;
    EXEC SQL EXECUTE
        BEGIN
          DBMS_PIPE.UNPACK_MESSAGE(:v_Code);
        END;
    END-EXEC;
    v_Code.arr[v_Code.len] = '\0';

    if (!strcmp(v_Code.arr, "STOP")) {
        /* Получено сообщение STOP. Прервем цикл. */
        break;
    } /* Конец обработки сообщения STOP. */
    else if (!strcmp(v_Code.arr, "TEST")) {
        /* Получено сообщение TEST. Пошлем подтверждение по тому же каналу. */
        EXEC SQL EXECUTE
```

```
        BEGIN
           DBMS_PIPE.PACK_MESSAGE('Handshake');
           :v_Status := DBMS_PIPE.SEND_MESSAGE('DebugPipe');
        END;
     END-EXEC;

     if (v_Status != 0) {
        /* Сообщение об ошибке. Выведем его на экран. */
        printf("Error %d while responding to TEST message\n",
               v_Status);
     }
  } /* Конец обработки сообщения TEST. */

  else if (!strcmp(v_Code.arr, "DEBUG")) {
     /* Получено сообщение DEBUG. Распакуем данные, содержащие
        сведения о канале возврата, описание и выходное значение. */
     v_ReturnPipeName.len = 30;
     v_Description.len = 100;
     v_Value.len = 100;
     EXEC SQL EXECUTE
        BEGIN
           DBMS_PIPE.UNPACK_MESSAGE(:v_ReturnPipeName);
           DBMS_PIPE.UNPACK_MESSAGE(:v_Description);
           DBMS_PIPE.UNPACK_MESSAGE(:v_Value);
        END;
     END-EXEC;

        /* Завершим входные переменные нулями. */
        v_Description.arr[v_Description.len] = '\0';
        v_Value.arr[v_Value.len] = '\0';

        /* Выведем на экран отладочную информацию. */
        printf("%s: %s\n", v_Description.arr, v_Value.arr);

        /* Пошлем подтверждающее сообщение назад. */
        EXEC SQL EXECUTE
           BEGIN
              DBMS_PIPE.PACK_MESSAGE('Processed');
              :v_Status := DBMS_PIPE.SEND_MESSAGE(:v_ReturnPipeName);
           END;
        END-EXEC;

        if (v_Status != 0) {
           /* Сообщение об ошибке. Выведем его на экран. */
           printf("Error %d while sending handshake message\n",
                  v_Status);
        }
     } /* Конец обработки сообщения DEBUG. */
  } /* Конец успешного считывания сообщения. */

  else if (v_Status == 1) {
     /* Задержка вызова RECEIVE_MESSAGE. Вернемся назад и подождем. */
     continue;
  }
```

```
        else {
          /* Вызов RECEIVE_MESSAGE завершен с ошибкой. Зафиксируем это и
             выйдем из цикла. */
          printf("Main loop RECEIVE_MESSAGE Error. Status = %d\n",
                 v_Status);
          EXEC SQL ROLLBACK WORK RELEASE;
          exit(1);
        }

    } /* Конец основного цикла. */

    /* Отсоединимся от базы данных. */
    EXEC SQL COMMIT WORK RELEASE;
}
/* Функция обработки ошибок. Выведем ошибку на экран и выйдем из программы. */
void sqlerror() {

    printf("SQL Error!\n");
    Printf("%.*s\n", sqlca.sqlerrm.sqlerrml,
                     Sqlca.sqlerrm.sqlerrmc);

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    EXEC SQL ROLLBACK RELEASE;
}
```

## Модуль Debug

Сам модуль Debug определяется следующим образом:

```
-- Этот пример содержится в файле debug.sql.
CREATE OR REPLACE PACKAGE Debug AS
  -- Максимальное время ожидания подтверждающего сообщения (в секундах).
  v_TimeOut NUMBER := 10;

  -- Основная процедура Debug.
  PROCEDURE Debug(p_Description IN VARCHAR2,
                  p_Value IN VARCHAR2);

  -- Устанавливает среду Debug.
  PROCEDURE Reset;

  -- Прекращает работу демона.
  PROCEDURE Exit;
END Debug;

CREATE OR REPLACE PACKAGE BODY Debug as
  v_CurrentPipeName VARCHAR2(30);

  PROCEDURE Debug(p_Description IN VARCHAR2,
                  p_Value IN VARCHAR2) IS
    v_ReturnCode NUMBER;
    v_Handshake VARCHAR2(10);
  BEGIN
    /* Если у канала нет имени, то определим для него имя. */
    IF v_CurrentPipeName IS NULL THEN
```

```
      v_CurrentPipeName := DBMS_PIPE.UNIQUE_SESSION_NAME;
    END IF;

    /* Пошлем сообщение 'DEBUG' вместе с:
         - именем канала для подтверждения;
         - описанием;
         - значением.
    */
    DBMS_PIPE.PACK_MESSAGE('DEBUG');
    DBMS_PIPE.PACK_MESSAGE(v_CurrentPipeName);
    DBMS_PIPE.PACK_MESSAGE(p_Description);
    DBMS_PIPE.PACK_MESSAGE(p_Value);
    v_ReturnCode := DBMS_PIPE.SEND_MESSAGE('DebugPipe');

    IF v_ReturnCode != 0 THEN
      RAISE_APPLICATION_ERROR(-20210,
        'Debug.Debug: SEND_MESSAGE failed with ' || v_ReturnCode);
    END IF;

    /* Подождем подтверждающего сообщения по обратному каналу. */
    v_ReturnCode := DBMS_PIPE.RECEIVE_MESSAGE(v_CurrentPipeName);

    IF v_ReturnCode = 1 THEN
      -- Задержка.
      RAISE_APPLICATION_ERROR(-20211,
        'Debug.Debug: No handshake message received');
    ELSIF v_ReturnCode != 0 THEN
      -- Другая ошибка.
      RAISE_APPLICATION_ERROR(-20212,
        'Debug.Debug: RECEIVE_MESSAGE failed with ' ||
        v_ReturnCode);
    ELSE
      -- Проверим наличие подтверждающего сообщения.
      DBMS_PIPE.UNPACK_MESSAGE(v_Handshake);
      IF v_Handshake = 'Processed' THEN
        -- Выход обработан.
        NULL;
      ELSE
        -- Нет подтверждения.
        RAISE_APPLICATION_ERROR(-20213,
          'Debug.Debug: Incorrect handshake message received');
      END IF;
    END IF;
END Debug;
PROCEDURE Reset IS
    /* Убедимся, что демон работает, послав тестовое сообщение
       по каналу. Если не работает, установим ошибку. */
    v_ReturnCode NUMBER;
BEGIN
    DBMS_PIPE.PACK_MESSAGE('TEST');
    v_ReturnCode := DBMS_PIPE.SEND_MESSAGE('DebugPipe');

    IF v_ReturnCode != 0 THEN
      RAISE_APPLICATION_ERROR(-20200,
```

```
            'Debug.Reset: SEND_MESSAGE failed with ' || v_ReturnCode);
        END IF;

        /* Демон будет отвечать по тому же каналу. Если этот вызов
           задерживается, значит, демон не готов, поэтому нужно
           установить ошибку. */
        v_ReturnCode :=
          DBMS_PIPE.RECEIVE_MESSAGE('DebugPipe', v_TimeOut);
        IF v_ReturnCode = 1 THEN
          -- Задержка.
          RAISE_APPLICATION_ERROR(-20201,
            'Debug.Reset: Daemon not ready');
        ELSIF v_ReturnCode != 0 THEN
          -- Другая ошибка.
          RAISE_APPLICATION_ERROR(-20202,
            'Debug.Reset: RECEIVE_MESSAGE failed with ' ||
            v_ReturnCode);
        ELSE
          -- Демон готов.
          NULL;
        END IF;
    END Reset;

    PROCEDURE Exit IS
        v_ReturnCode NUMBER;
    BEGIN
        -- Пошлем сообщение 'STOP'.
        DBMS_PIPE.PACK_MESSAGE('STOP');
        v_ReturnCode := DBMS_PIPE.SEND_MESSAGE('DebugPipe');

        IF v_ReturnCode != 0 THEN
          RAISE_APPLICATION_ERROR(-20230,
          'Debug.Exit: SEND_MESSAGE failed with ' || v_ReturnCode);
        END IF;
    END Exit;
END Debug;
```

## Комментарии

Следует сделать ряд замечаний о данном варианте модуля Debug. Прежде всего нужно отметить, что программа Pro*C служит для вывода получаемых результатов. При вызове **Debug.Debug** результат отображается на экране именно программой Pro*C, а не сеансом PL/SQL. Поэтому нужно запускать эту программу и сеанс PL/SQL в разных окнах. По существу, программа функционирует как демон: большую часть времени она находится в режиме ожидания сообщения, посланного по каналу.

**Коды операций** Первое сообщение, посланное по каналу **DebugPipe**, является кодом операции для демона. Этим сообщением может быть STOP, TEST или DEBUG. Демон будет отвечать по-разному, в зависимости от полученного кода операции, работая как процесс-диспетчер. В более сложной ситуации демон выполняет функции порождающего процесса, создавая на основании кода операции другие процессы, которые обрабатывают данные, в то время как сам демон ожидает другого сообщения.

**Протоколы связи** Процедура **Debug.Debug** передает в начальном сообщении уникальное имя канала, сгенерированное при помощи функции UNIQUE_SESSION_NAME. После этого **Debug.Debug** прослушивает новый канал. Такой метод весьма удобен, так как позволяет одновременно выполняться нескольким демонам. Первое сообщение получается ожидающим демоном. Затем возвращается имя канала, которое будет использоваться только одним сеансом, поскольку оно однозначно идентифицирует сеанс, пославший данное сообщение. Таким образом, проблема работы нескольких получателей с одним и тем же каналом устраняется. После установления уникального имени канала оба сеанса могут посы-

лать и получать по нему сообщения в полной уверенности, что ни один другой сеанс не сможет их подслушать.

**Подтверждающие сообщения** И модуль PL/SQL, и демон Pro*C — это одновременно и получатель, и отправитель. Рассмотрим в качестве примера сообщение TEST, посылаемое процедурой **Debug.Debug**. Модуль Debug посылает сообщение, а затем ждет ответа. Задержка ответа свидетельствует о некорректном приеме начального сообщения. Использование подтверждающих сообщений в протоколах связи является очень важным и ценным методом.

# Модуль DBMS_ALERT

Посредством модуля DBMS_ALERT реализуется система оповещений базы данных. *Оповещение* (alert) — это сообщение, посылаемое после завершения транзакции. В отличие от программных каналов, являющихся асинхронными, оповещения синхронизированы с транзакциями. Они обычно применяются в односторонних линиях связи, а программные каналы — в двусторонних.

Посылающий сеанс формирует вызов процедуры SIGNAL для конкретного оповещения. Этот вызов фиксирует сигнал об оповещении в словаре данных, но реально оповещение не посылает. Оно посылается после завершения транзакции, содержащей вызов SIGNAL. Если производится откат этой транзакции, оповещение не посылается. Принимающий сеанс заранее фиксирует свою заинтересованность в конкретных оповещениях с помощью процедуры REGISTER. Получены будут только те сообщения, которые были предварительно зарегистрированы. Затем принимающий сеанс ожидает сигнала об оповещениях, используя процедуру WAITONE или WAITANY.

## Посылка оповещений

Оповещения посылаются с помощью процедуры SIGNAL, которая записывает информацию о них в словарь данных. Синтаксис процедуры SIGNAL таков:

```
PROCEDURE SIGNAL(name IN VARCHAR2,
                 message IN VARCHAR2);
```

SIGNAL имеет только два параметра: имя оповещения, о котором сигнализируется, и сообщение. Максимальная длина имен оповещений составляет 30 символов, и эти имена не учитывают регистра символов. Имена оповещений, как и программных каналов, начинаются с ORA$ и зарезервированы для Oracle, поэтому их не следует указывать в пользовательских приложениях. Максимальная длина сообщения равна 1800 байт.

Каждое конкретное оповещение может находиться только в двух состояниях: в сигнальном и несигнальном. При вызове SIGNAL состояние оповещения изменяется на сигнальное. Это изменение записывается в таблицу **dbms_alert_info** словаря данных (см. раздел "Оповещения и словарь данных" ниже). Таким образом, в конкретный момент времени сигнализировать об оповещении может только один сеанс. Об одном и том же оповещении в разные моменты времени могут сигнализировать несколько сеансов, однако первый сеанс заблокирует работу всех остальных.

При посылке оповещения все сеансы, ожидающие его в данный момент, получат сообщение. Если таких сеансов нет, то оповещение немедленно получит сеанс, вновь зврегистрировавшийся в списке ожидающих сеансов.

## Получение оповещений

Процесс получения оповещений состоит из двух этапов: регистрации заинтересованности в оповещении и ожидании его. Принимающий сеанс получит только те оповещения, на которые он зарегистрировался.

### Регистрация

Процедура REGISTER применяется для регистрации заинтересованности в конкретном оповещении. Любой сеанс базы данных может регистрироваться на сколь угодно большое число оповещений и будет оставаться зарегистрированным до тех пор, пока не отсоединится от базы данных или пока не вызовет процедуру REMOVE (см. ниже) с целью указать, что больше не заинтересован в этом оповещении. Описание процедуры REGISTER выглядит следующим образом:

```
PROCEDURE REGISTER(name IN VARCHAR2);
```

Единственным параметром этой процедуры является имя оповещения. Регистрация на оповещение не приводит к блокированию сеанса — он лишь фиксирует свою заинтересованность в оповещении.

## Ожидание конкретного оповещения

Процедура WAITONE используется для ожидания конкретного оповещения. Если о нем уже было сигнализировано, WAITONE немедленно возвращает в параметре состояния значение 0, что свидетельствует о получении оповещения. Если сигнала об оповещении не поступало, WAITONE блокируется до тех пор, пока не получит такой сигнал или пока не истечет время задержки. Описание процедуры WAITONE выглядит следующим образом:

```
PROCEDURE WAITONE(name IN VARCHAR2,
                  message IN VARCHAR2,
                  status OUT INTEGER,
                  timeout IN NUMBER DEFAULT MAXWAIT);
```

Как и RECEIVE_MESSAGE, WAITONE переводит принимающий сеанс в режим ожидания до получения сигнала об оповещении.

Один и тот же сеанс может сигнализировать об оповещении, а затем ожидать его. В такой ситуации обязательно завершайте транзакцию в промежутке между вызовами SIGNAL и WAITONE. В противном случае вызов WAITONE будет всегда переходить в состояние задержки, поскольку оповещение никогда не будет послано. Параметры процедуры WAITONE описаны таблице 16.8.

**ТАБЛИЦА 16.8.**

| Параметр | Тип | Описание |
|---|---|---|
| *name* | VARCHAR2 | Имя ожидаемого оповещения. Перед вызовом WAITONE сеанс должен зарегистрировать свою заинтересованность в этом оповещении с помощью REGISTER. |
| *message* | VARCHAR2 | Текст сообщения, включенный посылающим сеансом в вызов SIGNAL. Если процедура SIGNAL вызывалась для одного и того же оповещения несколько раз до его получения, то считывается только последнее сообщение. Более ранние сообщения отменяются. |
| *status* | INTEGER | Возвращает индикатор получения оповещения. Значение 0 свидетельствует, что оповещение получено, а 1 — что вызов перешел в состояние задержки. |
| *timeout* | NUMBER | Максимальное время ожидания оповещения в секундах. Если *timeout* не задан, то по умолчанию устанавливается значение DBMS_ALERT.MAXWAIT, равное 1000 дней. Если оповещение не получено за *timeout* секунд, вызов возвращается с состоянием 0. |

## Ожидание любого из оповещений

Сеанс также может ожидать любого из оповещений, на которые он зарегистрировался. Для этого применяется процедура WAITANY. В отличие от WAITONE, WAITANY будет успешно выполнена, если получен сигнал о любом из оповещений, а не только о конкретном. Описание процедуры WAITANY выглядит следующим образом:

```
PROCEDURE WAITANY(name OUT VARCHAR2,
                  message OUT VARCHAR2,
                  status OUT INTEGER,
                  timeout IN NUMBER DEFAULT MAXWAIT);
```

Параметры этой процедуры аналогичны параметрам процедуры WAITONE и имеют тот же смысл. Единственное отличие заключается в том, что параметр name — это параметр вида OUT, указывающий на оповещение, о котором получен сигнал.

Как и прежде, один и тот же сеанс может сигнализировать об оповещении и ожидать его, используя в данном случае процедуру WAITANY. Если в промежутке между моментом выдачи сигнала об оповещении и вызовом WAITANY не будет указан оператор COMMIT, процедура WAITANY перейдет в состояние задержки, поскольку оповещение никогда не будет послано. Параметры процедуры WAITANY описаны таблице 16.9.

**ТАБЛИЦА 16.9.**

| Параметр | Тип | Описание |
|---|---|---|
| *name* | VARCHAR2 | Имя оповещения, о котором сигнализируется. Сеанс получит только те оповещения, на которые он зарегистрировался до вызова WAITANY. |
| *message* | VARCHAR2 | Текст сообщения, включенный посылающим сеансом в вызов SIGNAL. Если процедура SIGNAL вызывалась для одного и того же оповещения несколько раз до его получения, то считывается только последнее сообщение. Более ранние сообщения отменяются. |
| *status* | INTEGER | Возвращает индикатор получения оповещения. Значение 0 свидетельствует, что оповещение получено, а 1 — что вызов перешел в состояние задержки. |
| *timeout* | NUMBER | Максимальное время ожидания оповещения в секундах. Если *timeout* не задан, то по умолчанию устанавливается значение DBMS_ALERT.MAXWAIT, равное 1000 дней. Если оповещение не получено за *timeout* секунд, вызов возвращается с состоянием 0. |

# Другие процедуры

Для работы с оповещениями используются две дополнительные процедуры: REMOVE и SET_DEFAULTS.

## Отмена регистрации

Когда сеанс более не заинтересован в некотором оповещении, он может отменить свою регистрацию на него. Это помогает сберечь ресурсы, используемые для выдачи сигналов об оповещении и для получения оповещения. Отмена регистрации осуществляется с помощью процедуры REMOVE:

```
PROCEDURE REMOVE(name IN VARCHAR2);
```

Единственным параметром этой процедуры является имя оповещения. Процедура REMOVE обратна по отношению к REGISTER.

## Интервалы опроса

В большинстве случае Oracle управляется событиями. Это значит, что сеанс, ожидающий какого-либо события, не должен циклически проверять возникновение этого события, а будет уведомляться о нем. Если же циклическая проверка нужна, то строится цикл, называемый *циклом опроса* (polling loop). Интервал времени между проверками называется *интервалом опроса* (polling interval). При использовании системы оповещений циклический опрос необходим в двух случаях:

1. Если база данных работает в разделяемом режиме (shared mode), — для поиска оповещений, сигналы о которых выдаются другим экземпляром. Интервал опроса для такого цикла можно установить с помощью процедуры SET_DEFAULTS.
2. Если при вызове WAITANY не поступило ни одного сигнала о зарегистрированных оповещениях, — для поиска сигнальных оповещений. Интервал опроса начинается с 1 с и экспоненциально увеличивается до максимального значения 30 с. Этот интервал не устанавливается пользователями.

Описание процедуры SET_DEFAULTS выглядит следующим образом:

```
PROCEDURE SET_DEFAULTS(polling_interval IN NUMBER);
```

Единственным параметром этой процедуры является интервал опроса, задаваемый в секундах; значение по умолчанию равно 5 с.

# Оповещения и словарь данных

Система оповещений реализуется с помощью представления словаря данных **dbms_alert_info**. Для каждого оповещения, на которые зарегистрировался сеанс, в эту таблицу вводится отдельная строка. Если на одно и то же оповещение регистрируется несколько сеансов, то для каждого сеанса вводится собственная строка. Представление **dbms_alert_info** состоит из четырех столбцов, структура которых описана в таблице 16.10.

**ТАБЛИЦА 16.10.**

| Столбец | Тип данных | Разрешены ли NULL-значения? | Описание |
|---|---|---|---|
| *name* | VARCHAR2(30) | NOT NULL | Имя зарегистрированного оповещения |
| *SID* | VARCHAR2(30) | NOT NULL | Идентификатор сеанса, зарегистрировавшего заинтересованность в оповещении |
| *changed* | VARCHAR2(1) | | Y, если оповещение является сигнальным; N — если не является |
| *message* | VARCHAR2(1800) | | Сообщение, передаваемое с вызовом сигнала |

Работа этого представления рассматривается на рис. 16.1. Здесь показаны три сеанса базы данных и выдаваемые ими команды. В каждый указанный момент времени демонстрируется содержимое **dbms_alert_info**. Ниже приведено описание происходящих событий:

T1: Сеанс B регистрирует заинтересованность в оповещении. В этот момент в **dbms_alert_info** вводится строка, фиксируя данное событие. В поле Changed заносится N, т.е. сигнал об оповещении еще не поступал.

T2: Сеанс A сигнализирует об оповещении с сообщением 'Message A'. Поскольку сеанс не завершил работу, **dbms_alert_info** не изменяется.

T3: Сеанс C сигнализирует о том же оповещении с сообщением 'Message C'. Сеанс A еще не завершил работу. В конкретный момент времени об оповещении может сигнализировать только один сеанс, поэтому вызов SIGNAL, выданный сеансом C, блокируется.

T4: Сеанс A завершает свою работу. Происходит два события. Вызов SIGNAL, выданный сеансом C, выполняется, а сообщение сеанса A записывается в **dbms_alert_info**. В поле Changed устанавливается Y, что свидетельствует о выдаче сигнала о данном оповещении.

T5: Сеанс C завершает свою работу. Сообщение в **dbms_alert_info** заменяется. Сообщение сеанса A потеряно.

T6: Сеанс B решает ожидать оповещения. Вызов WAITONE возвращается немедленно с сообщением 'Message C'. В поле Changed устанавливается N, т.е. оповещение более не является сигнальным.

# Сравнение модулей DBMS_PIPE и DBMS_ALERT

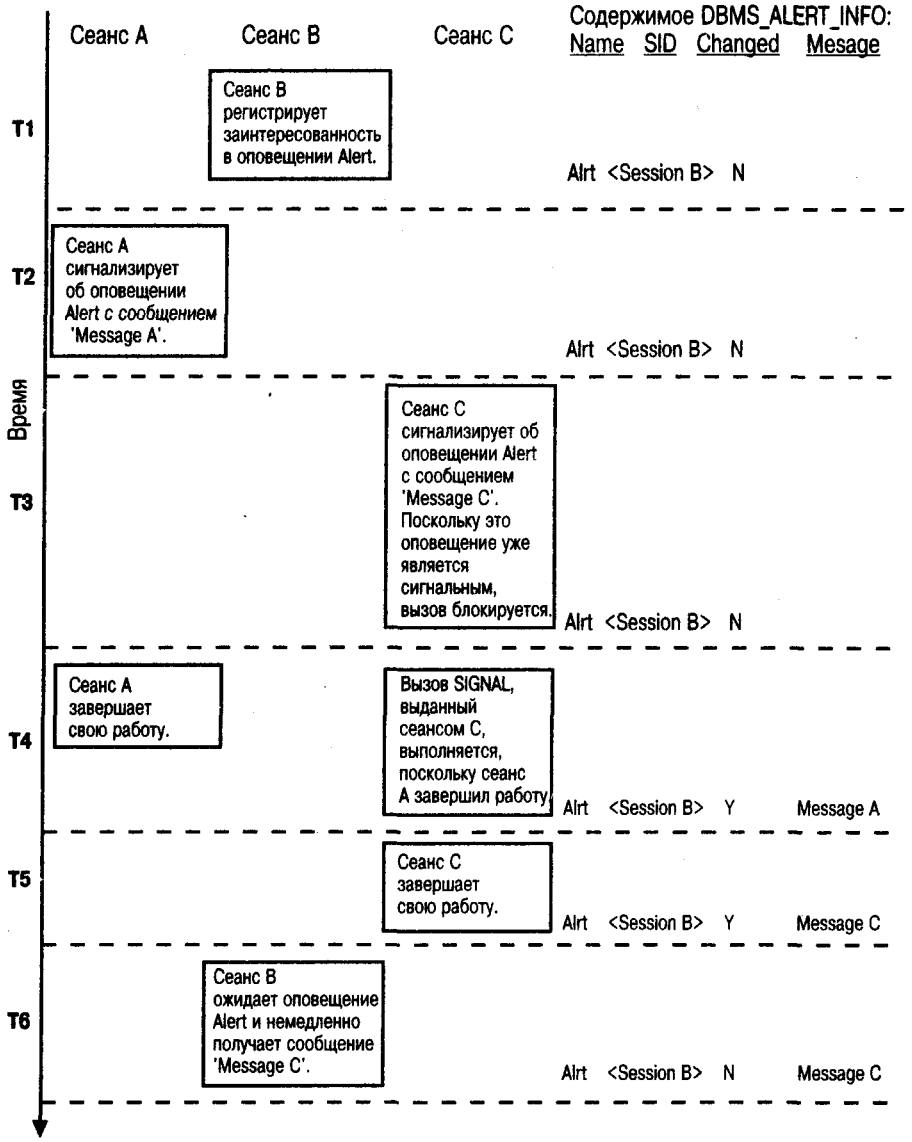
Некоторые свойства модулей DBMS_PIPE и DBMS_ALERT достаточно схожи:

- Оба реализованы в виде модулей PL/SQL. Это значит, что функциональными возможностями каждого из них можно воспользоваться в любой среде выполнения программ PL/SQL. Для модуля DBMS_PIPE одна из таких возможностей была рассмотрена — последний вариант модуля Debug, посылающего выходные результаты демону Pro*C. Обе программы, как вызывающая, так и получающая, используют в своей работе PL/SQL, но в различных средах.
- Оба модуля служат одной цели — пересылке сообщений между сеансами, соединенными с одним и тем же экземпляром базы данных. Пользователь в своем приложении может применять любой из этих модулей.
- В PL/SQL версии 2 не предусмотрена возможность непосредственного взаимодействия с программами на языке C. К примеру, нельзя вызывать программу из хранимых процедур PL/SQL. Единственный способ обойти это ограничение — воспользоваться программными каналами или оповещениями и с их помощью посылать сообщения ожидающему демону на языке C. В Oracle8 это ограничение снято благодаря использованию внешних процедур (см. главу 20) .

Хотя оба модуля выполняют аналогичные функции, в их работе имеется ряд различий. Применяйте тот модуль, который лучше подходит для вашей системы. Ниже приведены различия между модулями DBMS_PIPE и DBMS_ALERT:

- Оповещения синхронизированы с транзакциями, а программные каналы — нет. Оповещение не будет послано до тех пор, пока транзакция, в которой находится вызов DBMS_ALERT.SIGNAL, не будет завершена. Если выполняется откат транзакции, оповещение не посылается. Программные же каналы асинхронны. Сообщение посылается при каждом вызове DBMS_PIPE.SEND_MESSAGE, и вернуть его нельзя, даже если транзакция откатывается.

**Рис. 16.1.**
*Сигнализация об оповещении и получение оповещения*




- При выдаче сигнала об оповещении все сеансы, зарегистрировавшие заинтересованность в этом оповещении и ожидающие его, получат сообщение. Если оповещение ожидается несколькими сеансами, сообщение получит каждый из них. При работе с программными каналами все иначе: сообщение получает только один из ожидающих сеансов. Когда ожидающих сеансов несколько, неизвестно, какой из них получит сообщение.
- Методы пересылки информации также различны. В сообщении оповещения нельзя передавать более одной строки символов. С другой стороны, когда сообщение посылается по каналу, то передается все содержимое буфера сообщений. Передаваться могут фрагменты информации или данные различных типов.

- По программному каналу можно посылать более сложные сообщения, чем через оповещение, поэтому каналы проще использовать в двусторонних соединениях. Следовательно, при работе с программными каналами очень важным элементом являются протоколы связи. Оповещения же обычно применяются в односторонних соединениях.

# Итоги

В этой главе были рассмотрены два различных способа взаимодействия между сеансами. Программные каналы базы данных, реализованные посредством модуля DBMS_PIPE, обеспечивают установление двусторонних соединений и передачу сложных сообщений. Оповещения базы данных, реализованные посредством модуля DBMS_ALERT, обеспечивают установление односторонних соединений, функционирующих синхронно с транзакциями. Эти модули работают по-разному и применяются в приложениях разного типа. В следующей главе рассматривается средство усовершенствованной организации очередей (Advanced Queuing), доступное в Oracle8.

# Глава 17

## *Улучшенная организация очередей Oracle*

В предыдущей главе были рассмотрены средства взаимодействия сеансов, применяемые в Oracle7: программные каналы и оповещения, с помощью которых реализуются основные возможности обмена сообщениями. Однако в состав Oracle8 входит гораздо более мощная система обмена сообщениями, называемая Oracle Advanced Queuing (улучшенная организация очередей).

# Введение

Любое приложение можно рассматривать как набор программ и модулей, взаимодействующих друг с другом. Взаимодействие заключается в обмене сообщениями между программами. Для достаточно надежной работы системы средство, осуществляющее пересылку сообщений, должно обладать следующими качествами:

- **Надежность.** Должна быть твердая уверенность в том, что конкретное сообщение будет доставлено нужному получателю (получателям) только один раз.
- **Масштабируемость.** С увеличением числа программ и/или сообщений производительность системы не должна падать.
- **Восстанавливаемость.** В результате сбоя системы не должно быть потеряно ни одного сообщения. Необходимо, чтобы в случае сбоя все посланные и завершенные сообщения можно было восстановить.

PL/SQL 8.0 ... и ВЫШЕ

Средство Oracle Advanced Queuing обладает всеми вышеперечисленными качествам. Оно состоит из набора модулей PL/SQL и фоновых процессов, реализующих систему организации очередей. Подобные системы (например, мониторы транзакций (TP-мониторы)) производятся многими компаниями, однако Oracle Advanced Queuing входит в состав СУБД Oracle8. Данное средство создано с использованием SQL и таблиц Oracle, поэтому оно имеет те же преимущества, что и другие объекты базы данных:

- Очереди сохраняются в таблицах базы данных, поэтому они включаются в стандартный процесс резервного копирования и восстановления информации, поддерживаемый в Oracle. Очереди можно также экспортировать и импортировать.
- Во всех операциях, выполняемых над очередями, применяется SQL, что обеспечивает полноценное управление транзакциями.
- Сообщения являются объектными типами, поэтому они могут иметь различные атрибуты. Одно сообщение может состоять из нескольких атрибутов, посланных как единое целое.

Таким образом, Oracle Advanced Queuing удовлетворяет критериям надежности, масштабируемости и восстанавливаемости. Средства обмена сообщениями, применяемые в Oracle7, далеко не столь эффективны, и поэтому Advanced Queuing является наилучшей системой для разработки новых приложений.

## Компоненты средства Advanced Queuing

Известно, что *очередь* (queue) представляет собой упорядоченный список элементов. С помощью операции *постановки в очередь* (enqueue) в список вносятся новые элементы, а с помощью операции *вывода из очереди* (dequeue) элементы удаляются из этого списка. В Oracle Advanced Queuing элементами очереди являются сообщения, в состав которых, в свою очередь, входят данные и информация о маршрутах. Очереди не отделены от объектов словаря данных и хранятся в *таблицах очередей* (queue tables). Компоненты Oracle Advanced Queuing описаны в последующих разделах.

### Операция ENQUEUE

С помощью операции ENQUEUE сообщение ставится в конкретную очередь. При этом вместе с данными (которые могут быть либо объектными типами, либо данными RAW) вводится информация о маршруте, где указывается способ ввода и предполагаемые получатели этого сообщения. В состав информации о маршруте могут быть включены следующие параметры:

- **Идентификатор корреляции** (correlation identifier). Он создается пользователем и указывает на конкретное сообщение. С помощью идентификатора корреляции можно вывести это сообщение из очереди.
- **Список абонентов, или подписчиков** (subscription list), и список получателей (recipient list). Для очереди может быть создан список получателей, в котором указываются получатели сообщения, поставленного в эту очередь.Список получателей отменяется списком абонентов сообщения.

- **Приоритет** (priority). Сообщению может быть назначен определенный приоритет. Сообщения с более высоким приоритетом будут ставиться в очередь раньше сообщений с более низким приоритетом.
- **Группирование** (grouping). Сообщения можно группировать. Группа сообщений будет считываться некоторым процессом как одна единица, а затем этот процесс может выводить сообщения из очереди, выбирая последовательно каждое сообщение группы. Другие процессы, выполняя операции по выводу из очереди, будут получать последующие сообщения, а не те, которые входят в состав группы.
- **Временные показатели** (time specification). Можно установить время истечения срока действия сообщения, указав момент, когда оно должно быть выведено из очереди. Если срок действия сообщения истекает, то оно переносится в очередь исключительных ситуаций. Кроме того, можно задать минимальное время, в течение которого сообщение не выводится из очереди.
- **Защита транзакций** (transaction protection). Операция ENQUEUE может являться частью текущей транзакции или быть отдельной транзакцией, завершающейся сразу после своего выполнения, что позволяет немедленно выводить сообщение из очереди с помощью операции DEQUEUE.

## Операция DEQUEUE

С помощью операции DEQUEUE сообщение считывается из указанной очереди. Если сообщений нет, то операция DEQUEUE блокируется до тех пор, пока не будет считано некоторое сообщение или пока не истечет лимит времени, заданный для этой операции. Для DEQUEUE можно установить следующие параметры:

- **Режим вывода из очереди** (dequeue mode). Можно задать режим, в котором сообщение после считывания будет оставаться в очереди (режим просмотра) или удаляться из нее.
- **Тайм-аут** (time-out). Если ожидающих сообщений нет, то в операции DEQUEUE можно указать время их ожидания.
- **Защита транзакций** (transaction protection). Операция DEQUEUE может являться частью текущей транзакции или быть отдельной транзакцией, завершающейся сразу же после своего выполнения.
- **Описание сообщения** (message specification). Операция DEQUEUE может считать первое сообщение, сообщение, следующее после указанного, или выбрать сообщение на основании идентификатора корреляции.
- **Повторные попытки с задержкой** (retries with delays). Если выполнение DEQUEUE заканчивается неудачей и осуществляется откат транзакции, то пользователь может указать временную задержку, после которой сообщение будет повторно обработано, а также задать необходимое число попыток.

## Очереди

Существует два вида очередей: пользовательские очереди и очереди исключительных ситуаций. В *пользовательской очереди* (user queue) содержатся обычные сообщения. Если срок действия сообщения истекает (т.е. сообщение не выводится из очереди до срока истечения своего действия), то оно переносится в *очередь исключительных ситуаций* (exception queue), в которой содержатся ошибочные сообщения.

## Таблицы очередей

Очереди хранятся в таблицах очередей. По умолчанию в таблице очередей находится одна очередь исключительных ситуаций, хотя при необходимости в ней может содержаться и несколько таких очередей. Вначале создаются таблицы очередей, затем операциям ENQUEUE и DEQUEUE разрешается работа с очередями, находящимися в таблицах.

## Агенты

*Агент* (agent) – это пользователь очереди. *Производящий агент* (producing agent), или *производитель* (producer), ставит сообщения в очередь, а *потребляющий агент* (consuming agent), или *потребитель* (consumer), выводит сообщения из очереди. Один и тот же агент может быть и производителем, и потребителем.

## Менеджер времени

Менеджер времени (time manager) – это фоновый процесс Oracle, который используется для установления срока действия сообщений, а также для определения временных задержек и числа повторных попыток обработки сообщений. Если этот процесс не запущен, то данные параметры задавать нельзя.

▼ ВНИМАНИЕ Для запуска менеджера времени установите параметр AQ_TM_PROCESSES файла init.ora в 1. Значением по умолчанию является 0; при этом ни один процесс менеджера времени не запускается. Значение 1 указывает на необходимость запустить один такой процесс. В Oracle8 релиза 8.0.3 может работать только один процесс менеджера времени.

## Реализация Advanced Queuing

Средство Oracle Advanced Queuing реализовано с помощью двух модулей PL/SQL: DBMS_AQ и DBMS_AQADM. В модуле DBMS_AQ описаны операции ENQUEUE и DEQUEUE, а в модуле DBMS_AQADM – административные функции. Подпрограммы этих модулей перечислены в таблице 17.1 и подробно описаны в данной главе. Более детальную информацию о DBMS_AQ и DBMS_AQADM можно получить в руководстве Oracle8 Server Application Developer's Guide.

### ТАБЛИЦА 17.1. Модули DBMS_AQ и DBMS_AQADM

| Модуль | Подпрограмма | Описание |
|---|---|---|
| DBMS_AQ | ENQUEUE | Операция постановки в очередь. Используется для ввода сообщений в очередь и описания информации о маршруте |
| DBMS_AQ | DEQUEUE | Операция вывода из очереди. Используется для считывания сообщений в очереди вместе с указанной информацией |
| DBMS_AQADM | CREATE_QUEUE_TABLE | Создает таблицу очередей с указанными параметрами |
| DBMS_AQADM | CREATE_QUEUE | Создает очередь в существующей таблице очередей |
| DBMS_AQADM | DROP_QUEUE | Удаляет указанную очередь в ее таблице очередей |
| DBMS_AQADM | DROP_QUEUE_TABLE | Удаляет таблицу очередей вместе с содержащимися в ней очередями |
| DBMS_AQADM | ALTER_QUEUE | Изменяет параметры указанной очереди |
| DBMS_AQADM | ADD_SUBSCRIBER | Добавляет абонента в указанную очередь |
| DBMS_AQADM | REMOVE_SUBSCRIBER | Удаляет абонента из указанной очереди |
| DBMS_AQADM | GRANT_TYPE_ACCESS | Предоставляет полномочия администратору очередей |
| DBMS_AQADM | QUEUE_SUBSCRIBERS | Возвращает абонентов указанной очереди |
| DBMS_AQADM | START_TIME_MANAGER | Разрешает фоновый процесс менеджера времени |
| DBMS_AQADM | STOP_TIME_MANAGER | Запрещает фоновый процесс менеджера времени |

# Операции над очередями

В модуле DBMS_AQ содержится описание вспомогательных типов, а также операций ENQUEUE и DEQUEUE. Операции, связанные с управлением очередями (например, создание и удаление очередей и таблиц очередей), реализованы в модуле DBMS_AQADM.

## Вспомогательные типы

В модуле DBMS_AQ описан ряд вспомогательных типов, которые используются в последующих вызовах различных операций. Эти типы перечислены в таблице 17.2 и описаны ниже. Вспомогательные типы представляют собой индексные таблицы и записи PL/SQL.

### SYS.AQ$_AGENT

Этот тип идентифицирует агента, который может быть как производителем, так и потребителем очереди. Реализован тип в виде записи PL/SQL:

```
TYPE SYS.AQ$_AGENT IS RECORD (
```

```
    name VARCHAR2(30),
    address VARCHAR2(30),
    protocol NUMBER);
```

В поле *name* содержится имя агента. В Oracle8 варианта 8.0 поля *address* и *protocol* не применяются. Они зарезервированы для будущего использования.

**ТАБЛИЦА 17.2. Вспомогательные типы модуля DBMS_AQ**

| Имя типа | Описание | Где применяется |
|---|---|---|
| SYS.AQ$_AGENT | Производитель или потребитель сообщения | В ENQUEUE и DEQUEUE, так как является частью типа AQ$_RECIPIENT_LIST_T |
| AQ$_RECIPIENT_LIST_T | Список агентов, получающих сообщение | В ENQUEUE и DEQUEUE, так как является частью типа MESSAGE_PROPERTIES_T |
| MESSAGE_PROPERTIES_T | Характеристики сообщения | В ENQUEUE и DEQUEUE |
| ENQUEUE_OPTIONS_T | Параметры постановки в очередь | В ENQUEUE |
| DEQUEUE_OPTIONS_T | Параметры вывода из очереди | В DEQUEUE |

## AQ$_RECIPIENT_LIST_T

Этот тип представляет собой список агентов, которые могут получить сообщение. Он является характеристикой сообщения и поэтому используется как в операции ENQUEUE, так и в операции DEQUEUE. Список агентов-получателей реализован в виде индексной таблицы:

```
TYPE AQ$_RECIPIENT_LIST_T IS TABLE OF SYS.AQ$_AGENT
  INDEX BY BINARY_INTEGER;
```

## MESSAGE_PROPERTIES_T

Тип MESSAGE_PROPERTIES_T используется в операциях ENQUEUE и DEQUEUE и описывает параметры данного сообщения. Этот тип реализован в виде записи PL/SQL:

```
TYPE MESSAGE_PROPERTIES_T IS RECORD (
  priority BINARY_INTEGER DEFAULT 1,
  delay BINARY_INTEGER DEFAULT NO_DELAY,
  expiration BINARY_INTEGER DEFAULT NEVER,
  correlation VARCHAR2(128) DEFAULT NULL,
  attempts BINARY_INTEGER,
  recipient_list AQ$_RECIPIENT_LIST_T,
  exception_queue VARCHAR2(51) DEFAULT NULL,
  enqueue_time DATE,
  state BINARY_INTEGER);
```

Поля этого типа описаны в таблице 17.3.

**ТАБЛИЦА 17.3.**

| Поле | Тип данных | Описание |
|---|---|---|
| *priority* | BINARY_INTEGER | Приоритет данного сообщения. Чем меньше число, тем выше приоритет. Приоритет может быть любым целочисленным значением (в том числе и отрицательным) |
| *delay* | BINARY_INTEGER | Задержка данного сообщения. Значением *delay* может быть либо NO_DELAY (без задержки), означающее, что сообщение можно немедленно вывести из очереди, либо время ожидания в секундах. При выводе сообщения с указанием его идентификатора эта задержка отменяется. Задержанное сообщение ставится в очередь в состоянии WAITING (ожидания) и через *delay* секунд переходит в состояние READY (готовности) |

**ТАБЛИЦА 17.3. (продолжение)**

| Поле | Тип данных | Описание |
|---|---|---|
| *expiration* | BINARY_INTEGER | Время в секундах, по истечении которого действие сообщения, не выведенного из очереди, прекращается. Если для параметра *expiration* установлено значение NEVER (никогда), то время действия сообщения никогда не истечет. Отсчет времени действия сообщения начинается после задержки, если она задана. По истечении срока действия сообщение переводится в очередь исключительных ситуаций в состояние EXPIRED (время действия истекло) |
| *correlation* | VARCHAR2(128) | Идентификатор корреляции. При необходимости сообщения можно считывать по значению этого идентификатора |
| *attempts* | BINARY_INTEGER | Число попыток вывода сообщения из очереди. Этот параметр устанавливается только во время выполнения операции DEQUEUE |
| *recipient_list* | AQ$_RECIPIENT_LIST_T | Список получателей данного сообщения. Этот параметр считывается во время выполнения операции ENQUEUE и не возвращается операцией DEQUEUE |
| *exception_queue* | VARCHAR2(51) | Очередь исключительных ситуаций для данного сообщения. Если время действия сообщения истекает или число попыток вывода его из очереди превышает *max_retries*, это сообщение переносится в очередь исключительных ситуаций в состояние EXPIRED. В случае если параметр *exception_queue* не указан, сообщение переносится в очередь исключительных ситуаций, установленную для данной таблицы очередей по умолчанию. Когда параметр *exception_queue* указан, но нужная очередь не доступна во время переноса, данное сообщение переносится в очередь исключительных ситуаций, установленную для данной таблицы очередей по умолчанию, и сообщение об этом записывается в журнал оповещений |
| *enqueue_time* | DATE | Время постановки сообщения в очередь. Этот параметр возвращается операцией DEQUEUE, а во время выполнения операции ENQUEUE автоматически устанавливается системой |
| *state* | BINARY_INTEGER | Состояние сообщения во время выполнения операции DEQUEUE. Этот параметр не может быть установлен операцией ENQUEUE и при необходимости автоматически обновляется системой. Его значения: WAITING (ожидание — срок задержки еще не истек), READY (готовность — сообщение готово к обработке), PROCESSED (обработка завершена — сообщение обработано, но осталось в очереди) и EXPIRED (время действия истекло — сообщение перенесено в очередь исключительных ситуаций) |

## ENQUEUE_OPTIONS_T

Этот тип используется для указания параметров операции ENQUEUE, отличных от тех, которые относятся непосредственно к сообщению. Он реализуется как запись PL/SQL:

```
TYPE ENQUEUE_OPTIONS_T IS RECORD (
   visibility BINARY_INTEGER DEFAULT ON_COMMIT,
   relative_msgid RAW(16) DEFAULT NULL,
   sequence_deviation BINARY_INTEGER DEFAULT NULL);
```

Поля этого типа описаны в таблице 17.4.

**ТАБЛИЦА 17.4.**

| Поле | Тип данных | Описание |
|---|---|---|
| *visibility* | BINARY_INTEGER | Указывает режим функционирования транзакций. Если значением параметра *visibility* является значение по умолчанию ON_COMMIT (по завершении), то постановка сообщения в очередь выполняется по завершении текущей транзакции. Если же значением параметра *visibility* является IMMEDIATE (немедленно), то операция постановки в очередь образует отдельную транзакцию, которая завершается немедленно. Сообщение ставится в очередь даже в том случае, когда выполняется откат текущей транзакции |
| *relative_msgid* | RAW | Если в параметре *sequence_deviation* указано BEFORE, то данное сообщение вводится до сообщения, определяемого значением *relative_msgid*. Если значение *sequence_deviation* не указано, то параметр *relative_msgid* игнорируется |
| *sequence_deviation* | BINARY_INTEGER | Указывает местоположение данного сообщения в очереди. Значения этого параметра: BEFORE (до — сообщение должно быть поставлено в очередь впереди сообщения, указанного параметром *relative_msgid*), TOP (верх — сообщение должно быть поставлено в очередь впереди всех других сообщений) и NULL (позицию в очереди указывает приоритет сообщения). Значением по умолчанию является NULL |

## DEQUEUE_OPTIONS_T

Эта запись используется для указания параметров операции DEQUEUE, а не параметров самого сообщения:

```
TYPE DEQUEUE_OPTIONS_T IS RECORD (
  consumer_name VARCHAR2(30) DEFAULT NULL,
  dequeue_mode BINARY_INTEGER DEFAULT REMOVE,
  navigation BINARY_INTEGER DEFAULT NEXT_MESSAGE,
  visibility BINARY_INTEGER DEFAULT ON_COMMIT,
  wait BINARY_INTEGER DEFAULT FOREVER,
  msgid RAW(16) DEFAULT NULL,
  correlation VARCHAR2(30) DEFAULT NULL);
```

Поля этого типа описаны в таблице 17.5.

**ТАБЛИЦА 17.5.**

| Поле | Тип данных | Описание |
|---|---|---|
| *consumer_name* | VARCHAR2(30) | Имя потребителя, получающего сообщение. Если имя указано, то выводиться из очереди будут только те сообщения, которые соответствуют данному потребителю. Если очередь не установлена для работы с несколькими потребителями, параметр consumer_name должен содержать NULL-значение |
| *dequeue_mode* | BINARY_INTEGER | Указывает режим для операции вывода из очереди. Значения: BROWSE (режим просмотра — сообщение считывается без блокирования или исключения из очереди; аналогичен операции SELECT), LOCKED (режим блокирования — сообщение считывается, и для него устанавливается блокировка записи; аналогичен операции SELECT FOR UPDATE) и REMOVE (режим удаления — сообщение считывается, а затем обновляется или удаляется, в зависимости от свойств хранения, установленных для очереди ). Значение по умолчанию — REMOVE |

**ТАБЛИЦА 17.5. (продолжение)**

| Поле | Тип данных | Описание |
|---|---|---|
| *navigation* | BINARY_INTEGER | Указывает позицию считываемого сообщения. После определения позиции применяется критерий поиска (состоящий из параметров *consumer_name*, *msgid и correlation*). Значения этого параметра: NEXT_MESSAGE (следующее сообщение — считывается следующее доступное сообщение, удовлетворяющее критерию поиска), NEXT_TRANSACTION (следующая транзакция — оставшаяся часть текущей группы сообщений пропускается, и считывается первое доступное сообщение в следующей группе сообщений) и FIRST_MESSAGE (первое сообщение — считывается первое с начала очереди доступное сообщение, удовлетворяющее критерию поиска). Значение по умолчанию — NEXT_MESSAGE |
| *visibility* | BINARY_INTEGER | Указывает, является ли операция DEQUEUE частью текущей транзакции. Если значение этого параметра — ON_COMMIT (значение по умолчанию), то операция DEQUEUE является частью текущей транзакции, а если IMMEDIATE, то операция вывода из очереди образует отдельную транзакцию, которая завершается немедленно |
| *wait* | BINARY_INTEGER | Время ожидания (в секундах) сообщения, удовлетворяющего критерию поиска. Если выводятся сообщения одной группы, этот параметр игнорируется. Значения: FOREVER (бесконечно — ограничение по времени не задано; значение по умолчанию), NO_WAIT (без ожидания — если нет доступных сообщений, немедленный возврат операции) или определенное число секунд |
| *msgid* | RAW(16) | Идентификатор сообщения, выводимого из очереди |
| *correlation* | VARCHAR2(30) | Идентификатор корреляции сообщения, выводимого из очереди |

## Константы перечислимого типа

Многие поля вспомогательных типов DBMS_AQ являются константами перечислимого типа (enumerated constants). Такие поля описаны с типом BINARY_INTEGER, а принимаемые ими значения определяются модулем DBMS_AQ. При задании этих параметров необходимо предварять их имена именем модуля - например DBMS_AQ.IMMEDIATE. В таблице 17.6 приведен список констант перечислимого типа, их значений и областей применения.

**ТАБЛИЦА 17.6.**

| Поле | Значение | Где применяется |
|---|---|---|
| *sequence_deviation* | BEFORE, TOP | ENQUEUE_OPTIONS_T |
| *visibility* | IMMEDIATE, ON_COMMIT | ENQUEUE_OPTIONS_T |
| *dequeue_mode* | BROWSE, LOCKED, REMOVE | DEQUEUE_OPTIONS_T |
| *navigation* | FIRST_MESSAGE, NEXT_MESSAGE, NEXT_TRANSACTION | DEQUEUE_OPTIONS_T |
| *wait* | FOREVER, NO_WAIT | DEQUEUE_OPTIONS_T |
| *state* | WAITING, READY, PROCESSED, EXPIRED | MESSAGE_PROPERTIES_T |
| *delay* | NO_DELAY | MESSAGE_PROPERTIES_T |
| *expiration* | NEVER | MESSAGE_PROPERTIES_T |

## ENQUEUE

Операция DBMS_AQ.ENQUEUE принимает полезную нагрузку (payload) сообщения, которая является объектным типом или данными типа RAW:

```
PROCEDURE ENQUEUE (
```

*queue_name* IN VARCHAR2,
*enqueue_options* IN ENQUEUE_OPTIONS_T,
*message_properties* IN MESSAGE_PROPERTIES_T,
*payload* IN *тип_сообщения*,
*msgid* OUT RAW);

Параметры этой процедуры описаны в таблице 17.7.

**ТАБЛИЦА 17.7.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *queue_name* | VARCHAR2 | Очередь, в которую вводится сообщение. |
| *enqueue_options* | ENQUEUE_OPTIONS_T | Параметры, используемые при вводе сообщения. в очередь. Поля этого типа описаны в таблице 17.4. |
| *message_properties* | MESSAGE_PROPERTIES_T | Характеристики сообщения. Эти характеристики возвращаются последующей операцией DEQUEUE. Поля этого типа описаны в таблице 17.3. |
| *Payload* | тип_сообщения | Данные, содержащиеся в сообщении, могут быть либо данными RAW, либо объектным типом. Если это объектный тип, то он должен соответствовать типу, указанному при создании очереди. |
| *Msgid* | RAW | Возвращаемый идентификатор сообщения. Этот идентификатор можно использовать для вывода из очереди конкретного сообщения, вне зависимости от его приоритета и установленной задержки. |

Поле *sequence_deviation* в параметрах enqueue_options определяет соотношение между двумя сообщениями. Если это поле указано, то для сообщения, вводимого в очередь, задаются два условия:

- Задержка для вновь вводимого в очередь сообщения должна быть не больше задержки, установленной для сообщения, перед которым нужно поставить в очередь данное сообщение.
- Приоритет вновь вводимого в очередь сообщения должен быть не ниже приоритета сообщения, перед которым его надлежит поставить в очередь.

## DEQUEUE

Процедура DBMS_AQ.DEQUEUE также переопределяется типами сообщения (RAW или объектный тип):

PROCEDURE DEQUEUE (
*queue_name* IN VARCHAR2,
*dequeue_options* IN DEQUEUE_OPTIONS_T,
*message_properties* OUT MESSAGE_PROPERTIES_T,
*payload* OUT *тип_сообщения*,
*msgid* OUT RAW);

Параметры этой процедуры описаны в таблице 17.8.

Критерий поиска для вывода сообщения из очереди определяется полями *consumer_name*, *msgid* и/или *correlation* в параметрах *dequeue_options*. Без указания *msgid* из очереди выводятся только сообщения, находящиеся в состоянии READY.

**ТАБЛИЦА 17.8.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *queue_name* | VARCHAR2 | Имя очереди, в которой запрашиваются сообщения |
| *dequeue_options* | DEQUEUE_OPTIONS_T | Параметры, используемые при выводе сообщения из очереди. Поля этого типа описаны в таблице 17.5 |

**ТАБЛИЦА 17.8. (продолжение)**

| Параметр | Тип данных | Описание |
|---|---|---|
| *message_properties* | MESSAGE_PROPERTIES_T | Характеристики считываемого сообщения. Описание полей этого типа см. в таблице 17.3 |
| *payload* | *тип_сообщения* | *Сообщение, содержащее либо данные RAW, либо объектный тип* |
| *msgid* | RAW | Идентификатор сообщения |

# Администрирование очередей

В этом разделе будут рассмотрены модуль DBMS_AQADM и привилегии, необходимые для работы с очередями. Здесь анализируются представления словаря данных, имеющие отношение к очередям.

## Подпрограммы модуля DBMS_AQADM

В модуле DBMS_AQADM содержатся подпрограммы, применяемые для управления очередями и таблицами очередей. Они описаны в данном разделе.

### CREATE_QUEUE_TABLE

Процедура CREATE_QUEUE_TABLE используется для создания таблицы очередей и для указания характеристик по умолчанию, которые будут присваиваться всем очередям, создаваемым позднее в этой таблице. В число этих характеристик входит порядок сортировки очередей. CREATE_QUEUE_TABLE создает следующие объекты (здесь *queue_table* обозначает имя создаваемой таблицы):

- Очередь исключительных ситуаций по умолчанию для этой таблицы, называемая **aq$_queue_table_e**
- Представление "только для чтения", используемое для обращения к очереди и называемое **aq$_queue_table**
- Индекс для менеджера времени, называемый **aq$_queue_table_t**
- Индекс или таблицу, организованную по индексу для нескольких очередей потребителей, называемые **aq$_queue_table_i**

Ниже приведен синтаксис создания процедуры CREATE_QUEUE_TABLE, а ее параметры описаны в таблице 17.9.

```
PROCEDURE CREATE_QUEUE_TABLE (
   queue_table IN VARCHAR2,
   queue_payload_type IN VARCHAR2,
   storage_clause IN VARCHAR2 DEFAULT NULL,
   sort_list IN VARCHAR2 DEFAULT NULL,
   multiple_consumers IN BOOLEAN DEFAULT FALSE,
   message_grouping IN BINARY_INTEGER DEFAULT NONE,
   comment IN VARCHAR2 DEFAULT NULL,
   auto_commit IN BOOLEAN DEFAULT TRUE);
```

**ТАБЛИЦА 17.9.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *queue_table* | VARCHAR2 | Имя создаваемой таблицы очередей |
| *queue_payload_type* | VARCHAR2 | Тип пользовательских данных в этой очереди (RAW или объектный тип) |

**ТАБЛИЦА 17.9. (продолжение)**

| Параметр | Тип данных | Описание |
|---|---|---|
| *storage_clause* | VARCHAR2 | Параметр хранения информации для использования в операторе CREATE TABLE. Сюда могут входить конструкции TABLESPACE, PCTFREE, PCTUSED, LOB, INITTRANS и MAXTRANS. Полное описание команд, используемых при хранении информации, приведено в руководстве SQL Reference Guide |
| *sort_list* | VARCHAR2 | Определяет столбцы, предназначенные для использования в качестве первичных ключей, и тем самым задает порядок сортировки очередей, создаваемых в этой таблице. Более подробно об этом параметре рассказано ниже |
| *multiple_consumers* | BOOLEAN | Если TRUE, очереди, создаваемые в этой таблице, могут иметь по нескольку потребителей на одно сообщение. Пользователю с помощью процедуры GRANT_TYPE_ACCESS должно быть предоставлено право на доступ к типам. Если FALSE (значение по умолчанию), то у очередей, создаваемых в этой таблице, может быть только один потребитель на каждое сообщение |
| *message_grouping* | BINARY_INTEGER | Если NONE (нет; значение по умолчанию), каждое сообщение в очередях, создаваемых в данной таблице очередей, рассматривается отдельно. Если TRANSACTIONAL (транзакционный), то все сообщения, поставленные в очередь одной транзакцией, считаются элементами одной группы сообщений и поэтому выводятся из очереди одновременно |
| *comment* | VARCHAR2 | Комментарий для создаваемой таблицы. Он записывается в каталог очередей |
| auto_commit | BOOLEAN | Если TRUE (значение по умолчанию) — текущая транзакция завершается до создания таблицы очередей и операция фиксируется сразу же после обработки процедуры CREATE_QUEUE_TABLE. Если FALSE, процесс создания таблицы становится частью текущей транзакции и фиксируется при ее завершении |

**SORT_LIST** Параметр *sort_list* (список сортировки) процедуры CREATE_QUEUE_TABLE характеризует свойства сортировки данных таблицы очередей и, следовательно, всех очередей, создаваемых в этой таблице. Он представляет собой список столбцов сортировки, отделенных друг от друга запятыми. Разрешенными для сортировки столбцами являются priority и enq_time. Если указаны оба столбца, то первый столбец задает более важный порядок. Если *sort_list* не указан, по умолчанию задается сортировка по времени постановки в очередь, т.е. создается очередь типа FIFO

Тем не менее отдельные сообщения могут выводиться из очереди независимо от установленного порядка сортировки. Это можно сделать с помощью идентификатора корреляции или идентификатора сообщения при выполнении операции DEQUEUE.

## DROP_QUEUE_TABLE

С помощью этой процедуры удаляется указанная таблица очередей. Перед удалением таблицы все очереди, содержащиеся в ней, должны быть остановлены, а затем удалены:

```
PROCEDURE DROP_QUEUE_TABLE (
   queue_table IN VARCHAR2,
   force IN BOOLEAN DEFAULT FALSE,
   auto_commit IN BOOLEAN DEFAULT TRUE);
```

Параметры процедуры DROP_QUEUE_TABLE описаны в таблице 17.10.

**ТАБЛИЦА 17.10.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *queue_table* | VARCHAR2 | Имя удаляемой таблицы очередей |

**ТАБЛИЦА 17.10. (продолжение)**

| Параметр | Тип данных | Описание |
|---|---|---|
| *force* | BOOLEAN | Если TRUE, то все очереди в таблице останавливаются и удаляются автоматически. Если FALSE (значение по умолчанию) — при наличии очередей возвращается ошибка |
| *auto_commit* | BOOLEAN | Если TRUE (значение по умолчанию), то текущая транзакция завершается до удаления таблицы очередей и операция фиксируется сразу же после обработки процедуры DROP_QUEUE_TABLE. Если FALSE, операция фиксируется при завершении текущей транзакции |

## CREATE_QUEUE

Процедура CREATE_QUEUE используется для создания очереди в указанной таблице очередей. Имя очереди должно быть уникально в текущей схеме; по умолчанию операции постановки и вывода из очереди для данной очереди запрещены. После создания очереди ее необходимо сделать доступной при помощи процедуры START_QUEUE. Синтаксис создания CREATE_QUEUE таков:

```
PROCEDURE CREATE_QUEUE (
   queue_name IN VARCHAR2,
   queue_table IN VARCHAR2,
   queue_type IN BINARY_INTEGER DEFAULT NORMAL_QUEUE,
   max_retries IN NUMBER DEFAULT 0,
   retry_delay IN NUMBER DEFAULT 0,
   retention_time IN NUMBER DEFAULT 0,
   dependency_tracking IN BOOLEAN DEFAULT FALSE,
   comment IN VARCHAR2 DEFAULT NULL,
   auto_commit IN BOOLEAN DEFAULT TRUE);
```

Параметры этой процедуры описаны в таблице 17.11.

**ТАБЛИЦА 17.11.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *queue_name* | VARCHAR2 | Имя создаваемой очереди |
| *queue_table* | VARCHAR2 | Таблица очередей, в которой будет содержаться создаваемая очередь. Свойства новой очереди определяются на основании свойств таблицы очередей, таких как порядок сортировки и тип полезной нагрузки |
| *queue_type* | BINARY INTEGER | Тип очереди. Допустимые значения: NORMAL_QUEUE (стандартная очередь) и EXCEPTION_QUEUE (очередь исключительных ситуаций). Значение по умолчанию — NORMAL_QUEUE |
| *max_retries* | NUMBER | Максимальное число попыток вывода сообщения из очереди, указанное в REMOVE. Если некоторый агент выводит сообщение из очереди, а затем отменяет эту операцию, то число попыток увеличивается. Когда оно достигает значения *max_retries*, сообщение пересылается в очередь исключительных ситуаций. По умолчанию устанавливается 0, т.е. ни одной попытки вывода |
| *retry_delay* | NUMBER | Длительность (в секундах) задержки между попытками. Если значение *max_retries* равно 0, то значение *retry_delay* не имеет смысла. При наличии у очереди нескольких потребителей параметр *retry_delay* задавать нельзя |
| *retention_time* | NUMBER | Длительность (в секундах) сохранения сообщения в таблице очередей после его вывода из очереди. Допустимые значения: INFINITE (бесконечно — сообщение будет сохраняться бесконечно) или число секунд. Значение по умолчанию — 0 |

**ТАБЛИЦА 17.11. (продолжение)**

| Параметр | Тип данных | Описание |
|---|---|---|
| *dependency_tracking* | BOOLEAN | Этот параметр зарезервирован для будущего использования. Значение по умолчанию — FALSE, и указание TRUE в настоящее время является ошибкой |
| *comment* | VARCHAR2 | Описание создаваемой таблицы. Этот комментарий вносится в каталог очередей |
| *auto_commit* | BOOLEAN | Если TRUE (значение по умолчанию), текущая транзакция завершается до создания очереди и операция фиксируется сразу же после обработки процедуры CREATE_QUEUE. Если FALSE, то операция фиксируется при завершении текущей транзакции |

## DROP_QUEUE

С помощью процедуры DROP_QUEUE существующая очередь удаляется из таблицы очередей, в которой она содержится. Перед удалением очередь должна быть остановлена с помощью процедуры STOP_QUEUE. При выполнении операции DROP_QUEUE удаляются все данные, находящиеся в очереди. Описание этой процедуры выглядит так:

```
PROCEDURE DROP_QUEUE (
   queue_name IN VARCHAR2,
   auto_commit IN BOOLEAN DEFAULT TRUE);
```

где *queue_name* — имя удаляемой очереди, а *auto_commit* определяет режим работы транзакций при выполнении данной операции. Если *auto_commit* установлен в TRUE (значение по умолчанию), то сначала завершается текущая транзакция, а если в FALSE — очередь удаляется с завершением текущей транзакции.

## ALTER_QUEUE

Эта процедура используется для модификации характеристик очереди *max_retries*, *retry_delay* и *retention_time*. Для изменения других характеристик таблицу нужно сначала удалить, а затем создать заново. Ниже приведен синтаксис процедуры ALTER_QUEUE, а ее параметры описаны в таблице 17.12.

```
PROCEDURE ALTER_QUEUE (
   queue_name IN VARCHAR2,
   max_retries IN NUMBER DEFAULT NULL,
   retry_delay IN NUMBER DEFAULT NULL,
   retention_time IN NUMBER DEFAULT NULL,
   auto_commit IN BOOLEAN DEFAULT TRUE);
```

**ТАБЛИЦА 17.12.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *queue_name* | VARCHAR2 | Имя изменяемой очереди |
| *max_retries* | NUMBER | Максимальное число попыток вывода сообщения из очереди, указанное в REMOVE. Если некоторый агент выводит сообщение из очереди, а затем отменяет эту операцию, то число попыток увеличивается. Когда оно достигает значения *max_retries*, сообщение пересылается в очередь исключительных ситуаций |
| *retry_delay* | NUMBER | Длительность (в секундах) задержки между попытками. Если значение *max_retries* равно 0, значение *retry_delay* не имеет смысла. Параметр *retry_delay* задавать нельзя, если у очереди есть несколько потребителей |
| *retention_time* | NUMBER | Длительность (в секундах) сохранения сообщения в таблице очередей после его вывода из очереди. Допустимые значения: INFINITE (бесконечно — сообщение будет сохраняться бесконечно) или число секунд. Значение по умолчанию — 0 |
| *auto_commit* | BOOLEAN | Если TRUE, текущая транзакция завершается до создания очереди и операция фиксируется сразу же после обработки процедуры CREATE_QUEUE. Если FALSE — операция фиксируется при завершении текущей транзакции |

## START_QUEUE

Процедура START_QUEUE используется для разрешения операций постановки в данную очередь и/или вывода из этой очереди. После создания очередь должна быть запущена с помощью START_QUEUE. Для очереди исключительных ситуации можно выполнять только операции вывода из очереди, поэтому разрешение ввода сообщений в очередь исключительных ситуаций не имеет никакого эффекта. Очередь запускается сразу же после выполнения этой процедуры, даже если производится откат текущей транзакции. Описание START_QUEUE выглядит следующим образом:

```
PROCEDURE START_QUEUE (
   queue_name IN VARCHAR2,
   enqueue IN BOOLEAN DEFAULT TRUE,
   dequeue IN BOOLEAN DEFAULT TRUE);
```

где *queue_name* – имя запускаемой очереди, а *enqueue* и *dequeue* разрешают соответственно постановку сообщений в очередь и вывод сообщений из нее.

## STOP_QUEUE

Процедура STOP_QUEUE используется для запрещения операций постановки в данную очередь и/или вывода из нее. Нельзя останавливать очередь, если ее в этот момент используют транзакции. Очередь останавливается сразу же после выполнения этой процедуры, даже когда производится откат текущей транзакции. Описание STOP_QUEUE выглядит следующим образом:

```
PROCEDURE STOP_QUEUE (
   queue_name IN VARCHAR2,
   enqueue IN BOOLEAN DEFAULT TRUE,
   dequeue IN BOOLEAN DEFAULT TRUE,
   wait IN BOOLEAN DEFAULT TRUE);
```

где *queue_name* – это имя останавливаемой очереди, а *enqueue* и *dequeue* запрещают соответственно постановку и вывод сообщений из очереди. Если значением параметра *wait* является TRUE, вызов будет блокироваться до тех пор, пока все работающие с этой очередью транзакции не будут завершены или пока не будет выполнен их откат. Если значением параметра *wait* является FALSE, то вызов процедуры немедленно возвращается (успешно или неуспешно).

## ADD_SUBSCRIBER

Процедура ADD_SUBSCRIBER добавляет к очереди абонента по умолчанию. Использовать эту процедуру можно только для тех очередей, которые имеют несколько потребителей. ADD_SUBSCRIBER начинает действовать немедленно, и текущая транзакция завершается. Для работы с данной очередью пользователю необходимо предоставить доступ к ней с помощью GRANT_TYPE_ACCESS. Синтаксис процедуры ADD_SUBSCRIBER таков:

```
PROCEDURE ADD_SUBSCRIBER (
   queue_name IN VARCHAR2,
   subscriber IN SYS.AQ$_AGENT);
```

где *queue_name* – это имя модифицируемой очереди, а *subscriber* – новый абонент.

## REMOVE_SUBSCRIBER

С помощью процедуры REMOVE_SUBSCRIBER данный абонент удаляется из указанной очереди. Процедура REMOVE_SUBSCRIBER начинает действовать немедленно, и текущая транзакция завершается. Во время процесса удаления в существующих сообщениях уничтожаются все ссылки на вышеуказанного абонента. Для работы с этой очередью пользователю необходимо предоставить доступ к ней с помощью GRANT_TYPE_ACCESS. Синтаксис процедуры REMOVE_SUBSCRIBER таков:

```
PROCEDURE REMOVE_SUBSCRIBER (
   queue_name IN VARCHAR2,
   subscriber IN SYS.AQ$_AGENT);
```

где *queue_name* – это имя модифицируемой очереди, а *subscriber* – удаляемый абонент.

## QUEUE_SUBSCRIBERS

Эта функция возвращает таблицу, состоящую из абонентов данной очереди:

```
FUNCTION QUEUE_SUBSCRIBERS (
   queue_name IN VARCHAR2,
   RETURN AQ$_SUBSCRIBER_LIST_T;
```

### GRANT_TYPE_ACCESS

Процедура GRANT_TYPE_ACCESS используется для разрешения пользователю выполнять операции по управлению очередями. Если этого не сделать, то он не сможет выполнять операции CREATE_QUEUE_TABLE, CREATE_QUEUE, ADD_SUBSCRIBER и REMOVE_SUBSCRIBER:

```
PROCEDURE GRANT_TYPE_ACCESS (
   user_name IN VARCHAR2);
```

где *user_name* – пользователь базы данных, которому предоставляются полномочия.

### START_TIME_MANAGER

Эта процедура запускает процесс менеджера времени. Параметр AQ_TM_PROCESS файла init.ora должен быть предварительно установлен в 1, тогда этот процесс запускается во время старта базы данных. Менеджер времени запускается при завершении вызова этой процедуры, даже когда производится откат текущей транзакции. Процедура START_TIME_MANAGER не имеет никаких аргументов и описывается следующим образом:

```
PROCEDURE START_TIME_MANAGER;
```

### STOP_TIME_MANAGER

Процедура STOP_TIME_MANAGER останавливает процесс менеджера времени, но не уничтожает его, а лишь прекращает выполняемые им операции. Менеджер времени останавливается при возвращении вызова этой процедуры, даже когда производится откат текущей транзакции. Процедура STOP_TIME_MANAGER не имеет никаких аргументов и описывается следующим образом:

```
PROCEDURE STOP_TIME_MANAGER;
```

## Привилегии на работу с очередями

Для обеспечения нормальной работы с модулем DBMS_AQ или DBMS_AQADM системный администратор должен выполнить ряд определенных действий: предоставить пользователям две роли, а затем доступ к объектным типам, применяемым в Oracle Advanced Queuing.

### AQ_ADMINISTRATOR_ROLE

Роль AQ_ADMINISTRATOR_ROLE является стандартной ролью Oracle. В ее состав входят все те операторы GRANT, которые необходимы для предоставления полномочия EXECUTE на модули DBMS_AQ и DBMS_AQADM. Эту роль следует предоставлять пользователям, которые будут управлять очередями.

### AQ_USER_ROLE

С помощью роли AQ_USER_ROLE можно предоставить привилегию EXECUTE лишь на модуль DBMS_AQ. Эту роль следует предоставлять пользователям, которые будут ставить сообщения в очереди и выводить их оттуда, но не выполнять функций по управлению очередями.

### Доступ к объектным типам Oracle Advanced Queuing

Наконец, администратору очередей необходим доступ к объектным типам, используемым в очередях. Такой доступ предоставляется с помощью процедуры DBMS_AQADM.GRANT_TYPE_ACCESS, которую нужно вызывать из схемы SYS. Обладатель таких полномочий получает право выполнять подпрограммы CREATE_QUEUE_TABLE, CREATE_QUEUE, ADD_SUBSCRIBER и REMOVE_SUBSCRIBER.

## Очереди и словарь данных

В словаре данных находится ряд представлений, с помощью которых можно получать необходимые сведения об очередях, существующих в системе. В состав этих представлений входят представления для каждой таблицы очередей, а также для всех очередей и для всех таблиц очередей.

### Представление для таблицы очередей

Всякий раз при создании таблицы очередей (с помощью DBMS_AQADM.CREATE_QUEUE_TABLE) в текущей схеме создается представление, в котором содержится информация о сообщениях, находящихся в этой таблице. В каждой строке представлены сведения об одном сообщении. Имя такого представления ***aq$queue_table_name*** (где table_name – имя таблицы); его структура приведена в таблице 17.13.

**ТАБЛИЦА 17.13.**

| Столбец | Тип данных | Описание |
|---|---|---|
| queue | VARCHAR2(30) | Имя очереди, в которой находится данное сообщение |
| msg_id | RAW(16) | Идентификатор сообщения |
| corr_id | VARCHAR2(30) | Идентификатор корреляции |
| msg_priority | NUMBER | Приоритет сообщения |
| msg_state | VARCHAR2(9) | Состояние сообщения: READY, DELAYED, PROCESSED или EXPIRED |
| delay | DATE | Задержка, указанная при постановке сообщения в очередь |
| expiration | NUMBER | Время истечения срока действия сообщения, указанное при постановке его в очередь |
| enq_time | DATE | Время постановки сообщения в очередь |
| enq_user_id | NUMBER | Идентификатор пользователя, поставившего сообщение в очередь |
| enq_txn_id | VARCHAR2(30) | Идентификатор транзакции постановки в очередь |
| deq_time | DATE | Время вывода сообщения из очереди |
| deq_user_id | NUMBER | Идентификатор пользователя, исключившего сообщение из очереди |
| deq_txn_id | VARCHAR2(30) | Идентификатор транзакции вывода из очереди |
| retry_count | NUMBER | Число попыток вывода данного сообщения из очереди |
| exception_queue_owner | VARCHAR2(30) | Владелец очереди исключительных ситуаций, указанный при постановке сообщения в очередь (если имеется) |
| exception_queue | VARCHAR2(30) | Очередь исключительных ситуаций, указанная при постановке сообщения в очередь (если имеется) |
| user_data | *объектный_тип* или RAW | Данные, поставленные в очередь |

## DBA_QUEUE_TABLES/USER_QUEUE_TABLES

В представлении **dba_queue_tables** содержится информация обо всех таблицах очередей базы данных, а в представлении **user_queue_tables** — о таблицах очередей текущего пользователя. Структура каждого из них приведена в таблице 17.14.

**ТАБЛИЦА 17.14.**

| Столбец | Тип данных | Описание |
|---|---|---|
| owner | VARCHAR2(30) | Владелец таблицы очередей (только в dba_queue_tables) |
| queue_table | VARCHAR2(30) | Имя таблицы очередей |
| type | VARCHAR2(7) | Тип полезной нагрузки: OBJECT или RAW |
| sort_order | VARCHAR2(22) | Порядок сортировки, указанный при создании таблицы очередей |
| recipients | VARCHAR2(8) | SINGLE (один получатель) или MULTIPLE (несколько получателей) |
| message_grouping | VARCHAR2(13) | NONE или TRANSACTIONAL |
| user_comment | VARCHAR2(50) | Комментарий, указанный при создании таблицы очередей |

## DBA_QUEUES/USER_QUEUES

В представлении **dba_queues** содержится информация обо всех очередях базы данных, а в представлении **user_queues** – об очередях текущего пользователя. Структура каждого из них приведена в таблице 17.15.

**ТАБЛИЦА 17.15.**

| Столбец | Тип данных | Описание |
|---|---|---|
| owner | VARCHAR2(30) | Владелец очереди (только в **dba_queues**) |
| name | VARCHAR2(30) | Имя очереди |
| queue_table | VARCHAR2(30) | Таблица очередей, в которой размещается данная очередь |
| qid | NUMBER | Уникальный идентификатор очереди, генерируемый при ее создании |
| queue_type | VARCHAR2(15) | NORMAL_QUEUE или EXCEPTION_QUEUE |
| max_retries | NUMBER | Число разрешенных попыток вывода из очереди |
| retry_delay | NUMBER | Задержка между попытками |
| enqueue_enabled | VARCHAR2(7) | YES или NO |
| dequeue_enabled | VARCHAR2(7) | YES или NO |
| retention | VARCHAR2(40) | Период (в секундах), в течение которого сообщение сохраняется в очереди после вывода |
| user_comment | VARCHAR2(50) | Комментарий, указанный при создании очереди |

# Подробные примеры

В этом разделе на примерах будут продемонстрированы многие возможности средства Oracle Advanced Queuing:

- Создание очередей и таблиц очередей
- Простая постановка в очередь и простой вывод из очереди
- Очистка очереди
- Постановка в очередь и вывод из очереди по приоритету
- Постановка в очередь и вывод из очереди при помощи идентификатора сообщения или идентификатора корреляции
- Просмотр очереди
- Использование очередей исключительных ситуаций
- Удаление очередей

## Создание очередей и таблиц очередей

Сначала необходимо предоставить соответствующие полномочия владельцу очереди. Предположим, что очередь нужно создать в схеме **example.** Для этого вначале предоставим схеме **example** роль AQ_ADMINISTRATOR_ROLE, а затем доступ к типам. Итак, соединяясь с базой данной как **system** или как другой пользователь DBA (администратор базы данных), выполним следующие операторы:

```
GRANT AQ_ADMINISTRATOR_ROLE TO example;
BEGIN DBMS_AQADM.GRANT_TYPE_ACCESS('example'); END;
```

Теперь можно приступить к созданию таблиц очередей и самих очередей. Ниже приведен пример, в котором создаются и запускаются все очереди, используемые в этом и последующих примерах.

```
-- Этот пример содержится в файле createq.sql.
CREATE OR REPLACE TYPE MessageObj AS OBJECT (
  title VARCHAR2(30),
  data1 NUMBER,
  data2 VARCHAR2(100),
  data3 DATE,

  MEMBER PROCEDURE Print(v_Message IN VARCHAR2)
);

CREATE OR REPLACE TYPE BODY MessageObj AS
  MEMBER PROCEDURE Print(v_Message IN VARCHAR2) IS
  BEGIN
      DBMS_OUTPUT.PUT_LINE(v_Message || ': ' || title);
      DBMS_OUTPUT.PUT('Data 1: ' || data1);
      DBMS_OUTPUT.PUT(' Data 2: ' || data2);
      DBMS_OUTPUT.PUT_LINE(' Data 3: ' || data3);
  END Print;
END;

BEGIN
  -- Создадим простую таблицу, все значения которой заданы по
  -- умолчанию. Разрешим использование очередей FIFO без
  -- группирования сообщений и с одним потребителем.
  DBMS_AQADM.CREATE_QUEUE_TABLE(
      queue_table => 'SimpleQTab',
      queue_payload_type => 'MessageObj',
      comment => 'Simple Queue Table');
  -- Создадим простую очередь, содержащуюся в SimpleQTab. Вновь
  -- воспользуемся параметрами по умолчанию.
      DBMS_AQADM.CREATE_QUEUE(
         queue_name => 'SimpleQ',
         queue_table => 'SimpleQTab',
         comment => 'Simple Queue');

  -- Разрешим для SimpleQ операции постановки в очередь и вывода из очереди.
  DBMS_AQADM.START_QUEUE('SimpleQ');
  -- Создадим в SimpleQTab очередь исключительных ситуаций.
  DBMS_AQADM.CREATE_QUEUE(
      queue_name => 'ExceptionQ',
      queue_table => 'SimpleQTab',
      queue_type => DBMS_AQADM.EXCEPTION_QUEUE,
      comment => 'Exception Queue');

  -- Разрешим для ExceptionQ операции постановки в очередь и вывода
  -- из очереди.
  DBMS_AQADM.START_QUEUE('ExceptionQ', FALSE, TRUE);
END;
BEGIN
  -- Создадим приоритетную таблицу очередей, указав порядок сортировки.
  -- Не разрешим группирование сообщений и наличие
  -- нескольких потребителей для очередей.
  DBMS_AQADM.CREATE_QUEUE_TABLE(
      queue_table => 'PriorityQTab',
```

```
        queue_payload_type => 'MessageObj',
        sort_list => 'priority,enq_time',
        comment => 'Priority Queue Table');

  -- Создадим в PriorityQTab приоритетную очередь. Вновь
  -- воспользуемся параметрами по умолчанию.

    DBMS_AQADM.CREATE_QUEUE(
        queue_name => 'PriorityQ',
        queue_table => 'PriorityQTab',
        comment => 'Priority Queue');
    -- Разрешим для PriorityQ операции постановки в очередь и вывода
    -- из очереди.
    DBMS_AQADM.START_QUEUE('PriorityQ');
END;
```

## Простая постановка в очередь и простой вывод из очереди

Ниже приведен пример, иллюстрирующий серию операций постановки в очередь и вывода из очереди для **SimpleQ**.

```
-- Этот пример содержится в файле simple.sql.
DECLARE
  v_Message MessageObj;
  v_EnqueueOptions DBMS_AQ.ENQUEUE_OPTIONS_T;
  v_DequeueOptions DBMS_AQ.DEQUEUE_OPTIONS_T;
  v_MessageProperties DBMS_AQ.MESSAGE_PROPERTIES_T;
  v_MsgID RAW(16);
  c_NumMessages CONSTANT INTEGER := 10;

  e_QTimeOut EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_QTimeOut, -25228);
BEGIN
  FOR v_Counter IN 1..c_NumMessages LOOP
  -- Создадим сообщение для постановки в очередь.
  v_Message :=
      MessageObj('Message ' || v_Counter, v_Counter * 10,
                 'abcdefghijklmnopqrstuvwxyz', SYSDATE + v_Counter);

  -- Поставим его в очередь с параметрами по умолчанию.
  DBMS_AQ.ENQUEUE(
      queue_name => 'SimpleQ',
      enqueue_options => v_EnqueueOptions,
      message_properties => v_MessageProperties,
      payload => v_Message,
      msgid => v_MsgID);
END LOOP;

-- Завершим все операции постановки в очередь.
COMMIT;

-- Последовательно выведем все сообщения.
BEGIN
  LOOP
      -- Выведем первое сообщение в v_Message, ожидая максимум — 1 секунду.
```

```
        v_DequeueOptions.wait := 1;
        DBMS_AQ.DEQUEUE(
           queue_name => 'SimpleQ',
           dequeue_options => v_DequeueOptions,
           message_properties => v_MessageProperties,
           payload => v_Message,
           msgid => v_MsgID);

        -- Отобразим это сообщение.
        v_Message.Print('After dequeue');
    END LOOP;
  EXCEPTION
    WHEN e_QTimeOut THEN
        -- Достигнут конец очереди.
        NULL;
    END;
    -- Завершим все операции вывода из очереди.
    COMMIT;
  END;
```

При выполнении этого сценария в SQL*Plus получаются следующие результаты:

```
After dequeue: Message 1
Data 1: 10 Data 2: abcdefghijklmnopqrstuvwxyz Data 3: 15-AUG-97
After dequeue: Message 2
Data 1: 20 Data 2: abcdefghijklmnopqrstuvwxyz Data 3: 16-AUG-97
After dequeue: Message 3
Data 1: 30 Data 2: abcdefghijklmnopqrstuvwxyz Data 3: 17-AUG-97
After dequeue: Message 4
Data 1: 40 Data 2: abcdefghijklmnopqrstuvwxyz Data 3: 18-AUG-97
After dequeue: Message 5
Data 1: 50 Data 2: abcdefghijklmnopqrstuvwxyz Data 3: 19-AUG-97
After dequeue: Message 6
Data 1: 60 Data 2: abcdefghijklmnopqrstuvwxyz Data 3: 20-AUG-97
After dequeue: Message 7
Data 1: 70 Data 2: abcdefghijklmnopqrstuvwxyz Data 3: 21-AUG-97
After dequeue: Message 8
Data 1: 80 Data 2: abcdefghijklmnopqrstuvwxyz Data 3: 22-AUG-97
After dequeue: Message 9
Data 1: 90 Data 2: abcdefghijklmnopqrstuvwxyz Data 3: 23-AUG-97
After dequeue: Message 10
Data 1: 100 Data 2: abcdefghijklmnopqrstuvwxyz Data 3: 24-AUG-97
```

Сообщения были выведены из очереди в том порядке, в каком были поставлены, так как это очередь типа FIFO.

## Очистка очереди

Можно превратить вторую часть рассмотренного примера в процедуру, которая будет удалять все сообщения из очереди, выводя их до тех пор, пока ни одного не останется:

```
-- Этот пример содержится в файле clearq.sql.
CREATE OR REPLACE PROCEDURE ClearQueue(p_QueueName IN VARCHAR2) AS
  v_Message MessageObj;
  v_DequeueOptions DBMS_AQ.DEQUEUE_OPTIONS_T;
  v_MessageProperties DBMS_AQ.MESSAGE_PROPERTIES_T;
  v_MsgID RAW(16);
```

```
    e_QTimeOut EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_QTimeOut, -25228);
  BEGIN
    -- Последовательно выведем все сообщения.
    BEGIN
        LOOP
          -- Выведем первое сообщение в v_Message, ожидая максимум 1с екунду.
          v_DequeueOptions.wait := 1;
          DBMS_AQ.DEQUEUE(
            queue_name => 'SimpleQ',
            dequeue_options => v_DequeueOptions,
            message_properties => v_MessageProperties,
            payload => v_Message,
            msgid => v_MsgID);
        END LOOP;
    EXCEPTION
        WHEN e_QTimeOut THEN
          -- Достигнут конец очереди.
          NULL;
    END;
    -- Завершим все операции вывода из очереди.
    COMMIT;
  END;
```

## Постановка в очередь и вывод из очереди по приоритету

Ниже приведен пример, иллюстрирующий последовательность операций постановки в очередь и вывода из очереди по приоритету.

```
-- Этот пример содержится в файле priority.sql.
DECLARE
  v_Message MessageObj;
  v_EnqueueOptions DBMS_AQ.ENQUEUE_OPTIONS_T;
  v_DequeueOptions DBMS_AQ.DEQUEUE_OPTIONS_T;
  v_MessageProperties DBMS_AQ.MESSAGE_PROPERTIES_T;
  v_MsgID RAW(16);

  c_NumMessages CONSTANT INTEGER := 10;

  e_QTimeOut EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_QTimeOut, -25228);
BEGIN
  FOR v_Counter IN 1..c_NumMessages LOOP
  -- Создадим сообщение для постановки в очередь.
  v_Message :=
      MessageObj('Message ' || v_Counter, v_Counter * 10,
                 'zyxwvutsrqponmlkjihgfedcba', SYSDATE + v_Counter);

  -- Поставим его в очередь с приоритетом, равным -v_Counter.
  -- Таким образом, последнее вводимое в очередь сообщение будет
  -- иметь наивысший приоритет.
  v_MessageProperties.priority := -v_Counter;
  DBMS_AQ.ENQUEUE(
      queue_name => 'PriorityQ',
      enqueue_options => v_EnqueueOptions,
```

```
        message_properties => v_MessageProperties,
        payload => v_Message,
        msgid => v_MsgID);

    END LOOP;

    -- Завершим все операции постановки в очередь.
    COMMIT;

    -- Последовательно выведем все сообщения.
    BEGIN
        LOOP
          -- Выведем первое сообщение в v_Message, ожидая максимум
          -- 1 секунду.
          v_DequeueOptions.wait := 1;
          DBMS_AQ.DEQUEUE(
             queue_name => 'PriorityQ',
             dequeue_options => v_DequeueOptions,
             message_properties => v_MessageProperties,
             payload => v_Message,
             msgid => v_MsgID);

          -- Отобразим это сообщение.
          v_Message.Print('After dequeue');
        END LOOP;
    EXCEPTION
        WHEN e_QTimeOut THEN
          -- Достигнут конец очереди.
          NULL;
    END;

    -- Завершим все операции вывода из очереди.
    COMMIT;
END;
```

**При выполнении этого сценария в SQL*Plus получаются следующие результаты:**

```
After dequeue: Message 10
Data 1: 100 Data 2: zyxwvutsrqponmlkjihgfedcba Data 3: 24-AUG-97
After dequeue: Message 9
Data 1: 90 Data 2: zyxwvutsrqponmlkjihgfedcba Data 3: 23-AUG-97
After dequeue: Message 8
Data 1: 80 Data 2: zyxwvutsrqponmlkjihgfedcba Data 3: 22-AUG-97
After dequeue: Message 7
Data 1: 70 Data 2: zyxwvutsrqponmlkjihgfedcba Data 3: 21-AUG-97
After dequeue: Message 6
Data 1: 60 Data 2: zyxwvutsrqponmlkjihgfedcba Data 3: 20-AUG-97
After dequeue: Message 5
Data 1: 50 Data 2: zyxwvutsrqponmlkjihgfedcba Data 3: 19-AUG-97
After dequeue: Message 4
Data 1: 40 Data 2: zyxwvutsrqponmlkjihgfedcba Data 3: 18-AUG-97
After dequeue: Message 3
Data 1: 30 Data 2: zyxwvutsrqponmlkjihgfedcba Data 3: 17-AUG-97
After dequeue: Message 2
Data 1: 20 Data 2: zyxwvutsrqponmlkjihgfedcba Data 3: 16-AUG-97
```

```
After dequeue: Message 1
Data 1: 10 Data 2: zyxwvutsrqponmlkjihgfedcba Data 3: 15-AUG-97
```

Обратите внимание на разницу в результатах обработки простой очереди и приоритетной очереди. В последней сообщения выводятся по приоритету, что дало в итоге список, обратный полученному для простой очереди.

## Постановка в очередь и вывод из нее при помощи идентификатора сообщения или идентификатора корреляции

С помощью идентификатора корреляции пользователь может указать сообщение или последовательность сообщений определенной строкой символов. Этот идентификатор передается в качестве характеристики сообщения во время постановки в очередь и во время вывода его из очереди. В приведенном ниже блоке иллюстрируется использование идентификатора корреляции при выводе серии связанных между собой сообщений.

При помощи идентификатора сообщения выводится лишь одно соответствующее сообщение. Идентификатор сообщения возвращается операцией постановки в очередь и может быть передан в качестве одного из параметров операции вывода из очереди во время ее выполнения. Это иллюстрируется в приведенном ниже блоке.

```
-- Этот пример содержится в файле ident.sql.
DECLARE
  v_Message MessageObj;
  v_EnqueueOptions DBMS_AQ.ENQUEUE_OPTIONS_T;
  v_DequeueOptions DBMS_AQ.DEQUEUE_OPTIONS_T;
  v_MessageProperties DBMS_AQ.MESSAGE_PROPERTIES_T;
  v_MsgID RAW(16);
  v_TigerMsgID RAW(16);

  c_NumMessages CONSTANT INTEGER := 10;

  TYPE t_Correlations IS TABLE OF VARCHAR2(30)
      INDEX BY BINARY_INTEGER;
  v_Correlations t_Correlations;

  e_QTimeOut EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_QTimeOut, -25228);
BEGIN
  -- Инициализируем массив идентификаторов корреляции. Всего будет 5
  -- различных идентификаторов, причем каждый получит два сообщения.
  FOR v_Counter IN 1..c_NumMessages LOOP
      IF MOD(v_Counter, 5) = 1 THEN
        v_Correlations(v_Counter) := 'Lion';
      ELSIF MOD(v_Counter, 5) = 2 THEN
        v_Correlations(v_Counter) := 'Tiger';
      ELSIF MOD(v_Counter, 5) = 3 THEN
        v_Correlations(v_Counter) := 'Bear';
      ELSIF MOD(v_Counter, 5) = 4 THEN
        v_Correlations(v_Counter) := 'Fish';
      ELSE
        v_Correlations(v_Counter) := 'Horse';
      END IF;
  END LOOP;

  FOR v_Counter IN 1..c_NumMessages LOOP
```

```
    -- Создадим сообщение для постановки в очередь.
    v_Message :=
       MessageObj('Message ' || v_Counter, v_Counter * 10,
                  'abcdefghijklmnopqrstuvwxyz', SYSDATE + v_Counter);

    v_MessageProperties.correlation := v_Correlations(v_Counter);
    DBMS_OUTPUT.PUT_LINE('Enqueing message ' || v_Counter ||
       ' with correlation ID ' || v_Correlations(v_Counter));
    DBMS_AQ.ENQUEUE(
       queue_name => 'SimpleQ',
       enqueue_options => v_EnqueueOptions,
       message_properties => v_MessageProperties,
       payload => v_Message,
       msgid => v_MsgID);

    -- Сохраним один из идентификаторов сообщения о тигре (Tiger).
    IF v_Correlations(v_Counter) = 'Tiger' THEN
        v_TigerMsgID := v_MsgID;
    END IF;
END LOOP;

-- Завершим все операции постановки в очередь.
COMMIT;

-- Выведем из очереди только сообщения с идентификатором
-- корреляции 'Fish'.
BEGIN
    LOOP
       -- Выведем первое сообщение в v_Message, ожидая максимум 1 секунду.
       v_DequeueOptions.wait := 1;
       v_DequeueOptions.correlation := 'Fish';
       DBMS_AQ.DEQUEUE(
          queue_name => 'SimpleQ',
          dequeue_options => v_DequeueOptions,
          message_properties => v_MessageProperties,
          payload => v_Message,
          msgid => v_MsgID);

       -- Отобразим это сообщение.
       v_Message.Print('After dequeue with correlation ID Fish');
    END LOOP;
EXCEPTION
    WHEN e_QTimeOut THEN
       -- Достигнут конец очереди.
       NULL;
END;

-- Выведем из очереди только сообщение с сохраненным идентификатором.
v_DequeueOptions.correlation := NULL;
v_DequeueOptions.msgid := v_TigerMsgID;
DBMS_AQ.DEQUEUE(
    queue_name => 'SimpleQ',
    dequeue_options => v_DequeueOptions,
    message_properties => v_MessageProperties,
```

```
        payload => v_Message,
        msgid => v_MsgID);

    -- Отобразим это сообщение.
    v_Message.Print('After dequeue with saved message ID');

    -- Сбросим все оставшиеся сообщения.
    ClearQueue('SimpleQ');

    -- Завершим все операции вывода из очереди.
    COMMIT;
END;
```

Ниже приведены результаты выполнения этого сценария.

```
Enqueing message 1 with correlation ID Lion
Enqueing message 2 with correlation ID Tiger
Enqueing message 3 with correlation ID Bear
Enqueing message 4 with correlation ID Fish
Enqueing message 5 with correlation ID Horse
Enqueing message 6 with correlation ID Lion
Enqueing message 7 with correlation ID Tiger
Enqueing message 8 with correlation ID Bear
Enqueing message 9 with correlation ID Fish
Enqueing message 10 with correlation ID Horse
After dequeue with correlation ID Fish: Message 4
Data 1: 40 Data 2: abcdefghijklmnopqrstuvwxyz Data 3: 18-AUG-97
After dequeue with correlation ID Fish: Message 9
Data 1: 90 Data 2: abcdefghijklmnopqrstuvwxyz Data 3: 23-AUG-97
After dequeue with saved message ID: Message 7
Data 1: 70 Data 2: abcdefghijklmnopqrstuvwxyz Data 3: 21-AUG-97
```

## Просмотр очереди

Установив для параметра вывода dequeue_mode значение BROWSE, можно исследовать содержимое очереди, не удаляя ее элементы. Такой метод удобен, например, при просмотре очереди в поисках конкретного сообщения. Именно эту операцию выполняет функция **SearchQueue**:

```
-- Этот пример является частью файла searchq.sql.
CREATE OR REPLACE FUNCTION SearchQueue(
  /* Просматривает очередь в поисках первого экземпляра сообщения с
     заголовком p_MessageTitle и возвращает идентификатор этого экземпляра
     сообщения. Если сообщение не найдено, возвращается NULL. */
  p_QueueName IN VARCHAR2,
  p_MessageTitle IN VARCHAR2)
  RETURN RAW AS
  v_Message MessageObj;
  v_DequeueOptions DBMS_AQ.DEQUEUE_OPTIONS_T;
  v_MessageProperties DBMS_AQ.MESSAGE_PROPERTIES_T;
  v_MsgID RAW(16);

  e_QTimeOut EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_QTimeOut, -25228);
BEGIN
  -- Будем выполнять цикл до тех пор, пока не останется сообщений
  -- или пока не будет найдено нужное сообщение.
```

```
    BEGIN
        LOOP
           -- Выведем из очереди (в режиме просмотра) первое сообщение в
           -- v_Message, ожидая максимум 1 секунду. В этом режиме
           -- сообщение не удаляется из очереди.
           v_DequeueOptions.wait := 1;
           v_DequeueOptions.dequeue_mode := DBMS_AQ.BROWSE;
           DBMS_AQ.DEQUEUE(
              queue_name => p_QueueName,
              dequeue_options => v_DequeueOptions,
              message_properties => v_MessageProperties,
              payload => v_Message,
              msgid => v_MsgID);

           -- Проверим заголовки сообщений.
           IF v_Message.title = p_MessageTitle THEN
              -- Найдено соответствие, возвращаем идентификатор сообщения.
              COMMIT;
              RETURN v_MsgID;
           END IF;
        END LOOP;
    EXCEPTION
        WHEN e_QTimeOut THEN
           -- Достигнут конец очереди.
           NULL;
    END;

    -- Завершим все операции вывода из очереди и возвратим NULL,
    -- что означает отсутствие соответствия.
    COMMIT;
    RETURN NULL;
  END;
```

Ниже приведен блок, иллюстрирующий вызов функции **SearchQueue**.

```
-- Этот пример является частью файла searchq.sql.
DECLARE
  v_Message MessageObj;
  v_EnqueueOptions DBMS_AQ.ENQUEUE_OPTIONS_T;
  v_DequeueOptions DBMS_AQ.DEQUEUE_OPTIONS_T;
  v_MessageProperties DBMS_AQ.MESSAGE_PROPERTIES_T;
  v_MsgID RAW(16);

  c_NumMessages CONSTANT INTEGER := 10;

  e_QTimeOut EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_QTimeOut, -25228);
BEGIN
  FOR v_Counter IN 1..c_NumMessages LOOP
     -- Создадим сообщение для постановки в очередь.
     v_Message :=
      MessageObj('Message ' || v_Counter, v_Counter * 10,
                 'abcdefghijklmnopqrstuvwxyz', SYSDATE + v_Counter);

     -- Введем его с параметрами по умолчанию.
```

```
        DBMS_AQ.ENQUEUE(
          queue_name => 'SimpleQ',
          enqueue_options => v_EnqueueOptions,
          message_properties => v_MessageProperties,
          payload => v_Message,
          msgid => v_MsgID);
    END LOOP;

    -- Завершим все операции постановки в очередь.
    COMMIT;

    -- Поиск сообщения 4.
    v_MsgID := SearchQueue('SimpleQ', 'Message 4');

    -- Если сообщение найдено, выведем его из очереди и отобразим.
    IF v_MsgID IS NOT NULL THEN
        v_DequeueOptions.wait := 1;
        v_DequeueOptions.msgid := v_MsgID;
        DBMS_AQ.DEQUEUE(
          queue_name => 'SimpleQ',
          dequeue_options => v_DequeueOptions,
          message_properties => v_MessageProperties,
          payload => v_Message,
          msgid => v_MsgID);
        v_Message.Print('After search');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Message 4 not found');
    END IF;

    -- Очистим очередь.
    ClearQueue('SimpleQ');
END;
```

## Работа с очередями исключительных ситуаций

Сообщение автоматически переносится в очередь исключительных ситуаций в следующих случаях:

1. Сообщение не выведено из очереди до момента истечения срока своего действия, указанного как параметр сообщения во время его постановки в очередь.
2. Число попыток вывода сообщения из очереди превышает значение max_retries, указанное для очереди.

В приведенном ниже блоке иллюстрируется вторая ситуация.

```
-- Этот пример содержится в файле except.sql.
DECLARE
  v_Message MessageObj;
  v_EnqueueOptions DBMS_AQ.ENQUEUE_OPTIONS_T;
  v_DequeueOptions DBMS_AQ.DEQUEUE_OPTIONS_T;
  v_MessageProperties DBMS_AQ.MESSAGE_PROPERTIES_T;
  v_MsgID RAW(16);
  v_NormalCount INTEGER;
  v_ExceptionCount INTEGER;

  c_NumMessages CONSTANT INTEGER := 3;
```

```
  e_QTimeOut EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_QTimeOut, -25228);
BEGIN
  -- Введем в очередь 3 сообщения, установив очередь
  -- исключительных ситуаций.
  FOR v_Counter IN 1..c_NumMessages LOOP
      -- Создадим сообщение для постановки в очередь.
      v_Message :=
       MessageObj('Message ' || v_Counter, v_Counter * 10,
                  'abcdefghijklmnopqrstuvwxyz', SYSDATE + v_Counter);
      -- Поставим его в очередь.
      v_MessageProperties.exception_queue := 'ExceptionQ';
      DBMS_AQ.ENQUEUE(
        queue_name => 'SimpleQ',
        enqueue_options => v_EnqueueOptions,
        message_properties => v_MessageProperties,
        payload => v_Message,
        msgid => v_MsgID);
  END LOOP;

  -- Завершим все операции постановки в очередь.
  COMMIT;

  -- Убедимся, что в обычной очереди находятся три сообщения, а в
  -- очереди исключительных ситуаций сообщений нет.
  SELECT COUNT(*)
      INTO v_NormalCount
      FROM aq$SimpleQTab
      WHERE queue = UPPER('Simpleq');
  SELECT COUNT(*)
      INTO v_ExceptionCount
      FROM aq$SimpleQTab
      WHERE queue = UPPER('ExceptionQ');
  DBMS_OUTPUT.PUT('After initial enqueues, count(simple) = ' ||
                  v_NormalCount);
  DBMS_OUTPUT.PUT_LINE(', count(exception) = ' || v_ExceptionCount);

  -- Выведем из очереди первое сообщение, затем выполним откат. При
  -- этом произойдет откат операции вывода из очереди, а также
  -- увеличится значение поля attempts в характеристиках сообщения.
  -- Значение max_retries для SimpleQ равно 0, поэтому сообщение
  -- будет перенесено в очередь исключительных ситуаций.
  v_DequeueOptions.wait := 1;
  DBMS_AQ.DEQUEUE(
      queue_name => 'SimpleQ',
      dequeue_options => v_DequeueOptions,
      message_properties => v_MessageProperties,
      payload => v_Message,
      msgid => v_MsgID);

  ROLLBACK;

  -- Убедимся, что в обычной очереди находятся два сообщения, а в
```

```
    -- очереди исключительных ситуаций сообщений нет.
    SELECT COUNT(*)
        INTO v_NormalCount
        FROM aq$SimpleQTab
        WHERE queue = UPPER('Simpleq');
    SELECT COUNT(*)
        INTO v_ExceptionCount
        FROM aq$SimpleQTab
        WHERE queue = UPPER('ExceptionQ');
    DBMS_OUTPUT.PUT('After dequeue and rollback, count(simple) = ' ||
                    v_NormalCount);
    DBMS_OUTPUT.PUT_LINE(', count(exception) = ' || v_ExceptionCount);

    -- Теперь можно выбрать сообщение в очереди исключительных ситуаций.
    -- Обратите внимание: необходимо использовать идентификатор сообщения,
    -- поскольку состояние этого сообщения определено как EXPIRED,
    -- а сообщения в состоянии, отличном от READY, не будут выводиться
    -- обычным образом.
    v_DequeueOptions.msgid := v_MsgID;
    DBMS_AQ.DEQUEUE(
        queue_name => 'ExceptionQ',
        dequeue_options => v_DequeueOptions,
        message_properties => v_MessageProperties,
        payload => v_Message,
        msgid => v_MsgID);
    v_Message.Print('After exception dequeuing');

    -- Убедимся, что в обычной очереди находятся два сообщения, а в
    -- очереди исключительных ситуаций сообщений нет.
    SELECT COUNT(*)
        INTO v_NormalCount);
        FROM aq$SimpleQTab
        WHERE queue = UPPER('Simpleq');
    SELECT COUNT(*)
        INTO v_ExceptionCount
        FROM aq$SimpleQTab
        WHERE queue = UPPER('ExceptionQ');
    DBMS_OUTPUT.PUT('After exception dequeue, count(simple) = ' ||
                    v_NormalCount);
    DBMS_OUTPUT.PUT_LINE(', count(exception) = ' || v_ExceptionCount);

    -- Очистим очередь и завершим работу.
    ClearQueue('SimpleQ');
    COMMIT;
END;
```

## Удаление очередей

В сценарии, приведенном ниже, все очереди останавливаются, затем удаляются, а после этого удаляются таблицы, в которых содержались эти очереди.

```
-- Этот пример содержится в файле dropq.sql.
BEGIN
   DBMS_AQADM.STOP_QUEUE('SimpleQ');
   DBMS_AQADM.DROP_QUEUE('SimpleQ');
```

```
    DBMS_AQADM.STOP_QUEUE('ExceptionQ');
    DBMS_AQADM.DROP_QUEUE('ExceptionQ');
    DBMS_AQADM.DROP_QUEUE_TABLE('SimpleQTab');
END;

BEGIN
    DBMS_AQADM.STOP_QUEUE('PriorityQ');
    DBMS_AQADM.DROP_QUEUE('PriorityQ');
    DBMS_AQADM.DROP_QUEUE_TABLE('PriorityQTab');
END;

DROP  TYPE  MessageObj;
```

# Итоги

В этой главе была рассмотрена система Oracle Advanced Queuing. Это надежное и эффективное средство является не отдельным программным продуктом, а встроенным компонентом системы базы данных. Поэтому все полезные свойства, которыми обладает Oracle, можно использовать и в Oracle Advanced Queuing. Были рассмотрены операции, необходимые для работы с очередями и управления ими, а также модули PL/SQL, обеспечивающие выполнение этих операций. В следующей главе рассматриваются еще два модуля DBMS, применяемые в Oracle: DBMS_JOB и UTL_FILE.

# Глава 18

## Задания для баз данных и файловый ввод/вывод

Помимо модулей DBMS, рассмотренных выше, в состав PL/SQL входят другие модули, в том числе – DBMS_JOB и UTL_FILE. Модуль DBMS_JOB, доступный в PL/SQL версии 2.2 и выше, дает возможность системе самостоятельно, без вмешательства пользователей, периодически выполнять хранимые процедуры. С помощью модуля UTL_FILE, доступного в PL/SQL версии 2.3 и выше, реализуется возможность чтения и записи файлов операционной системы. Данные модули расширяют возможности PL/SQL и предоставляют в распоряжение пользователей функции, которые применяются в других языках программирования третьего поколения.

# Задания для баз данных

**PL/SQL 2.2 ... и ВЫШЕ**

В PL/SQL версии 2.2 и выше можно планировать выполнение подпрограмм PL/SQL в заранее установленное время при помощи модуля DBMS_JOB, который создает *очереди заданий* (job queues). Для того чтобы выполнить задание, его необходимо внести в очередь, указав параметры, определяющие частоту выполнения задания. Информацию о заданиях, обрабатываемых в конкретный момент времени, а также сведения об успешном или неуспешном выполнении предыдущих заданий можно найти в словаре данных. Более подробно о заданиях для баз данных рассказано в руководстве Server Administrator's Guide релиза 7.2 (или более позднего).

**PL/SQL 8.0 ... и ВЫШЕ**

Система Oracle Advanced Queuing, применяемая в PL/SQL 8.0 и описанная в главе 17, намного расширяет возможности PL/SQL по организации очередей по сравнению с модулем DBMS_JOB.

## Фоновые процессы

В экземпляре Oracle одновременно функционирует множество процессов. За выполнение различных операций в базе данных, например считывание блоков данных в память, запись блоков на диск или архивирование данных для длительного хранения, несут ответственность разные процессы (см. главу 22). Помимо процессов, управляющих базой данных, существуют так называемые SNP-процессы. Они обеспечивают получение моментальных снимков (snapshots) базы данных, а также создание очередей заданий.

SNP-процессы, как и другие процессы базы данных, выполняются в фоновом режиме. Однако в отличие от всех других процессов, в случае сбоя в работе SNP-процесса Oracle перезапускает его, и это не оказывает никакого влияния на функционирование остальных процессов базы данных. Если же отказывают другие процессы, то это обычно приводит к остановке всей базы данных. Время от времени SNP-процесс возобновляет свою работу (просыпается) и проверяет, не появились ли какие-либо задания. При выполнении задания SNP-процесс обрабатывает его и вновь переходит в режим ожидания (засыпает). В каждый момент времени один SNP-процесс может выполнять только одно задание. В Oracle7 может существовать максимум десять SNP-процессов (с SNP0 по SNP9), поэтому одновременно может быть выполнено не более десяти заданий. В Oracle8 число этих процессов увеличено до 36 (с SNP0 по SNP9 и с SNPA по SNPZ).

Функционированием SNP-процессов управляют три параметра файла инициализации INIT.ORA; их описание приведено в таблице 18.1. Обратите внимание: если параметр JOB_QUEUE_PROCESSES установлен в ноль, то задания обрабатываться не будут. Каждый процесс пребывает в режиме ожидания JOB_QUEUE_INTERVAL секунд; таким образом, этот параметр определяет минимальную длительность промежутка времени между операциями исполнения заданий.

**ТАБЛИЦА 18.1. Параметры инициализации заданий**

| Параметр | Значение по умолчанию | Диапазон значений | Описание |
|---|---|---|---|
| JOB_QUEUE_PROCESSES | 0 | 0..10 | Число запускаемых процессов |
| JOB_QUEUE_INTERVAL | 60 | 1..3600 | Интервал между моментами возобновления работы процесса. Перед проверкой наличия нового задания процесс находится в режиме ожидания в течение указанного числа секунд |
| JOB_QUEUE_KEEP_CONNECTIONS | FALSE | TRUE, FALSE | Управляет режимом закрытия SNP-процессом соединений, установленных с удаленными базами данных. Если TRUE, все соединения сохраняются до остановки процесса. Если FALSE — соединения сохраняются только при наличии исполняемых заданий |

## Выполнение заданий

Задание можно выполнить двумя способами: поставить (submit) его в очередь заданий или выполнить немедленно. Когда задание ставится в очередь, SNP-процесс обрабатывает его в запланированное время. Можно задать такой режим, при котором задание с определенного момента выполняется автоматически. Если задание выполняется немедленно, то оно обрабатывается только один раз.

### SUBMIT

Задание ставится в очередь заданий с помощью процедуры SUBMIT:

```
PROCEDURE SUBMIT(job OUT BINARY_INTEGER,
                 what IN VARCHAR2,
                 next_date IN DATE DEFAULT SYSDATE,
                 interval IN VARCHAR2 DEFAULT NULL,
                 no_parse IN BOOLEAN DEFAULT FALSE);
```

Параметры процедуры SUBMIT описаны в таблице 18.2.

**ТАБЛИЦА 18.2.**

| Параметр | Тип | Описание |
|---|---|---|
| *job* | BINARY_INTEGER | Номер, присваиваемый заданию во время его создания. Пока существует задание, его номер не изменяется. Номера заданий уникальны в рамках всего экземпляра |
| *what* | VARCHAR2 | Программный текст задания на PL/SQL. Обычно это вызов хранимой процедуры |
| *next_date* | DATE | Дата следующего выполнения задания |
| *interval* | VARCHAR2 | Функция, определяющая момент следующего выполнения задания |
| *no_parse* | BOOLEAN | Если TRUE, то текст задания не будет подвергаться грамматическому разбору до первой обработки этого задания. Если FALSE (значение по умолчанию), текст задания подвергается грамматическому разбору при внесении задания в очередь. Удобно использовать этот параметр, когда объекты базы данных, на которые ссылается задание, еще не существуют, но задание должно быть внесено в очередь |

В качестве примера создадим процедуру **TempInsert**:

```
-- Этот пример содержится в файле tmpins.sql.
CREATE SEQUENCE temp_seq
  START WITH 1
  INCREMENT BY 1;

CREATE OR REPLACE PROCEDURE TempInsert AS
BEGIN
  INSERT INTO temp_table (num_col, char_col)
      VALUES (temp_seq.nextval, TO_CHAR(SYSDATE,
              'DD-MON-YY HH24:MI:SS'));
  COMMIT;
END TempInsert;
```

Зададим запуск процедуры TempInsert каждые 10 секунд с помощью следующего сценария SQL*Plus:

```
SQL> VARIABLE v_JobNum NUMBER
SQL> BEGIN
  2      DBMS_JOB.SUBMIT(:v_JobNum, 'TempInsert;', SYSDATE,
  3                      'sysdate + (10/(24*60*60))');
  4    END;
  5    /
```

```
PL/SQL procedure successfully completed.
SQL> print v_JobNum
  V_JOBNUM
--------------
         2
```

**Номера заданий** Заданию присваивается номер при первой постановке в очередь. Номера заданий генерируются в последовательности SYS.JOBSEQ. После того как заданию присвоен номер, его можно изменить, только удалив задание и затем вновь внести его в очередь.

▼ ОСТОРОЖНО Задания, как и другие объекты базы данных, можно экспортировать и импортировать. При этом номер задания не изменяется. При попытке импортировать задание с номером, который уже существует, возвращается сообщение об ошибке, и задание не импортируется. В такой ситуации вновь внесите задание в очередь, и его номер изменится.

**Описания заданий** С помощью параметра *what* указывается программный текст задания. В заданиях вызываются, как правило, хранимые процедуры, и поэтому в параметре *what* указывается символьная строка с вызовом некоторой процедуры. Такая процедура может иметь сколько угодно параметров, причем все они должны иметь вид IN, поскольку не могут существовать параметры фактические, получающие значения формальных параметров вида OUT или IN OUT. Единственным исключением из этого правила являются специальные идентификаторы *next_date* и *broken*, описанные ниже.

▼ ОСТОРОЖНО После того как задание внесено в очередь, оно будет запускаться одним из SNP-процессов в фоновом режиме. Обязательно поместите оператор COMMIT в конце текста процедуры этого задания для того, чтобы увидеть результаты.

При описании задания можно использовать три специальных идентификатора, которые приведены в таблице 18.3. Параметр *job* имеет вид IN, поэтому задание может лишь считывать это значение. Параметры *next_date* и *broken* имеют вид IN OUT, поэтому само задание может их изменять. Изменим текст процедуры **TempInsert**:

```
-- Этот пример содержится в файле tmpins1.sql.
CREATE OR REPLACE PROCEDURE TempInsert
  (p_NextDate IN OUT DATE) AS
  v_SeqNum NUMBER;
  v_StartNum NUMBER;
  v_SQLCode NUMBER;
  v_Errmsg VARCHAR2(60);
BEGIN
  SELECT temp_seq.nextval
      INTO v_SeqNum
      FROM dual;
  -- Проверим, является ли этот вызов первым.
BEGIN
  SELECT num_col
      INTO v_StartNum
      FROM temp_table
      WHERE char_col = 'TempInsert Start';

  -- Вызовы были и ранее, поэтому введем новое значение.
  INSERT INTO temp_table (num_col, char_col)
      VALUES (v_SeqNum, TO_CHAR(SYSDATE, 'DD-MON-YY HH24:MI:SS'));
EXCEPTION
  WHEN NO_DATA_FOUND THEN
      -- Вызов первый, поэтому введем в таблицу следующее:
      INSERT INTO temp_table (num_col, char_col)
        VALUES (v_SeqNum, 'TempInsert Start');
```

```
    END;

    -- Если число вызовов превышает 15, выйдем из подпрограммы.
    IF v_SeqNum - V_StartNum > 15 THEN
        p_NextDate := NULL;
    END IF;
    COMMIT;
END TempInsert;
```

и запустить ее на выполнение:

```
BEGIN
    DBMS_JOB.SUBMIT(:v_JobNum, 'TempInsert(next_date);', SYSDATE,
                    'sysdate + (5/(24*60*60))');
END;
```

Задание будет автоматически удаляться само из очереди заданий (установкой **p_NextDate** в NULL), когда его порядковый номер превысит 15. В результате того что задание может возвращать значения *next_date* и *broken*, оно может при необходимости удалять себя из очереди.

## ТАБЛИЦА 18.3. Идентификаторы управления заданиями

| Идентификатор | Тип | Описание |
|---|---|---|
| *job* | BINARY_INTEGER | Определяет номер текущего задания |
| *next_date* | DATE | Определяет дату следующего выполнения задания |
| *broken* | BOOLEAN | Определяет состояние задания — TRUE, если задание неработоспособно, и FALSE — в противном случае |

Параметр *what* — это строка символов типа VARCHAR2. Поэтому все символьные литералы, используемые в вызове процедуры задания, должны быть ограничены двумя одиночными кавычками. Кроме того, вызов процедуры должен заканчиваться двоеточием. Для примера вызовем процедуру **Register** с помощью следующей строки *what*:

```
'Register(10006, ''MUS'', 410);'
```

**Интервалы между обработкой** Первый раз задание будет запущено на выполнение через время, определенное параметром *next_date* процедуры SUBMIT. Непосредственно перед выполнением собственно задания обрабатывается функция, указанная параметром *interval*. При успешном решении задания результат, возвращаемый функцией *interval*, становится новым значением *next_date*. Если задание выполнено успешно, а *interval* возвращает NULL, задание удаляется из очереди. Выражение, указанное в *interval*, передается как строка символов, но его вычисление должно давать дату. Часто используемые выражения и результаты их вычисления приведены в таблице 18.4.

## ТАБЛИЦА 18.4.

| Значение для интервала | Результат |
|---|---|
| 'SYSDATE + 7' | Семь дней с момента последнего выполнения задания. Например, задание вносится в очередь во вторник, значит в следующий раз оно будет выполнено в следующий вторник. Если во второй раз задание выполняется неуспешно, а затем успешно выполняется в среду, то в следующий раз оно будет выполняться в среду |
| 'NEXT_DAY(TRUNC(SYSDATE), ''FRIDAY'') + 12' | Каждую пятницу в полдень. Обратите внимание на то, что в строке символов литерал 'FRIDAY' заключен в двойные кавычки |
| 'SYSDATE + 1/24' | Каждый час |

## RUN

Процедура DBMS_JOB.RUN выполняет задание немедленно:

```
RUN(job IN BINARY_INTEGER);
```

Задание должно быть предварительно создано с помощью процедуры SUBMIT. Независимо от того, в каком состоянии находится задание, оно немедленно запускается текущим процессом. Обратите внимание: задание не запускается фоновым SNP-процессом.

## Неработоспособные задания

В случае неудачного выполнения задания Oracle автоматически пытается еще раз выполнить его. Первая попытка делается спустя одну минуту после первого сбоя. Если и эта попытка терпит неудачу, то следующая производится через две минуты. Каждый раз интервал удваивается: четыре минуты, восемь и т.д. Если этот интервал становится больше указанного интервала между обработками задания, то используется последний. Задание помечается как неработоспособное (broken) после 16 неудачных попыток выполнения. Такие задания автоматически повторно не выполняются.

Тем не менее с помощью процедуры RUN неработоспособное задание можно выполнить. Если такой вызов завершается успехом, то счетчик ошибок сбрасывается в ноль и задание помечается как работоспособное. Для изменения состояния задания можно воспользоваться также процедурой BROKEN:

```
BROKEN(job IN BINARY_INTEGER,
       broken IN BOOLEAN,
       next_date IN DATE DEFAULT SYSDATE);
```

Параметры процедуры BROKEN описаны в таблице 18.5.

**ТАБЛИЦА 18.5.**

| Параметр | Тип | Описание |
|---|---|---|
| *job* | BINARY_INTEGER | Номер задания, состояние которого изменяется |
| *broken* | BOOLEAN | Новое состояние задания. Если TRUE, задание помечено как неработоспособное. Если FALSE — задание помечено как работоспособное и будет выполнено в момент, указанный параметром **next_date** |
| *next_date* | DATE | Дата следующего выполнения задания. Значение по умолчанию — SYSDATE |

## Удаление задания

Задание можно удалить из очереди явным образом. Для этого служит процедура REMOVE:

```
REMOVE(job IN BINARY_INTEGER);
```

в которой единственным параметром является номер задания. Если значение *next_date* равно NULL (в результате выполнения задания или вычисления значения функции *interval*), то задание удаляется после выполнения. Если во время вызова REMOVE задание выполняется, то оно также будет удалено из очереди после выполнения.

## Изменение задания

После постановки задания в очередь его параметры можно изменить. Это осуществляется с помощью одной из следующих процедур:

```
PROCEDURE CHANGE(job IN BINARY_INTEGER,
                 what IN VARCHAR2,
                 next_date IN DATE,
                 interval IN VARCHAR2);
PROCEDURE WHAT(job IN BINARY_INTEGER,
               what IN VARCHAR2);
PROCEDURE NEXT_DATE(job IN BINARY_INTEGER,
                    next_date IN DATE);
PROCEDURE INTERVAL(job IN BINARY_INTEGER,
                   interval IN VARCHAR2);
```

Процедура CHANGE используется для одновременного изменения нескольких характеристик, а процедуры WHAT, NEXT_DATE и INTERVAL – для изменения характеристик, обозначенных соответствующими аргументами.

Аргументы этих процедур аналогичны аргументам процедуры SUBMIT. Если с помощью процедур CHANGE или WHAT изменяется параметр *what*, то текущей становится новая среда выполнения задания. Более подробно о средах выполнения заданий рассказано в разделе "Среды выполнения заданий".

## Просмотр заданий в словаре данных

Информация о заданиях регистрируется в нескольких представлениях словаря данных. Представления **dba_jobs** и **user_jobs** содержат такие сведения о заданиях, как *what*, *next_date* и *interval*, а также информацию о среде выполнения. Представление **dba_jobs_running** описывает задания, выполняемые в данный момент. (Об этих представлениях см. в приложении D.)

## Среды выполнения заданий

Во время внесения задания в очередь запоминается текущее состояние среды. В число фиксируемых показателей входят установки параметров NLS, например NLS_DATE_FORMAT. Установки, записанные в процессе формирования задания, используются всякий раз при его последующем выполнении. При изменении характеристики what (с помощью CHANGE или WHAT) эти установки также изменяются.

▼ ВНИМАНИЕ

Можно изменить среду выполнения задания, если вызвать команду ALTER SESSION из модуля DBMS_SQL. Это воздействует только на текущее выполнение задания; на то, как впоследствии будет выполняться задание, это не влияет. Модуль DBMS_SQL описан в главе 15.

# Файловый ввод/вывод

PL/SQL 2.3 ... и ВЫШЕ

Выше было рассказано о том, что в PL/SQL не предусмотрены встроенные средства для ввода и вывода информации, однако эти функции реализуются через вспомогательные программные модули. Ввод/вывод данных на экран осуществляется с помощью модуля DBMS_OUTPUT, описанного в главе 14. PL/SQL 2.3 расширяет эту функциональную возможность, обеспечивая ввод/вывод текстовых файлов посредством модуля UTL_FILE. С помощью этой версии UTL_FILE невозможно осуществлять вывод информации непосредственно в двоичные файлы, однако в будущих версиях PL/SQL данное ограничение будет устранено.

PL/SQL 8.0 ... и ВЫШЕ

В Oracle8 можно считывать двоичные файлы при помощи объектов BFILE, которые представляют собой особую форму внешних объектов LOB. Объекты BFILE, как и другие объекты LOB, рассматриваются в главе 21.

В данном разделе рассказывается о том, как функционирует модуль UTL_FILE. В конце раздела приведены три примера, демонстрирующие его работу.

## Безопасность

В клиентском PL/SQL имеется модуль, подобный UTL_FILE и называемый TEXT_IO. Однако в отношении требований безопасности информации эти модули различны. Файлы, создаваемые с помощью клиентского модуля TEXT_IO, можно размещать в любом месте на станции клиента, при условии наличия необходимых привилегий для работы с операционной системой. При выполнении клиентских операций файлового ввода/вывода каких-либо привилегий для работы с PL/SQL и собственно базой данных не требуется.

### Безопасность базы данных

На сервере необходимо поддерживать безопасность информации на более высоком уровне, чем на станции клиента. Для этого выделяется ограниченный набор специальных каталогов, в которые модуль UTL_FILE может записывать данные. Такие каталоги называются доступными каталогами (accessible directories) и определяются параметром UTL_FILE_DIR инициализационного файла INIT.ORA. Каждый доступный каталог указывается в файле INIT.ORA отдельной строкой:

UTL_FILE_DIR = *имя_каталога*

Вид спецификации *имя_каталога* зависит от применяемой операционной системы. Если операционная система учитывает регистр символов, имя_каталога также учитывает регистр. Приведенные ниже строки файла INIT.ORA корректны в операционной системе Unix при условии, что указанные каталоги существуют на самом деле.

```
UTL_FILE_DIR = /tmp
UTL_FILE_DIR = /home/oracle/output_files
```

Для того чтобы обратиться к файлу с помощью модуля UTL_FILE, нужно передать функции FOPEN имя каталога и имя файла в виде отдельных параметров. Имя каталога сопоставляется со списком доступных каталогов и, если оно там найдено, операция разрешается. Если в FOPEN указано имя недоступного каталога, выдается сообщение об ошибке. Подкаталоги доступных каталогов можно использовать только тогда, когда они также указаны как доступные явным образом. На основании приведенного выше примера составим таблицу 18.6, в которой будут продемонстрированы примеры верных и неверных комбинаций каталог/файл.

**ТАБЛИЦА 18.6. Верные и неверные спецификации файлов**

| Имя каталога | Имя файла | Комментарии |
|---|---|---|
| /tmp | myfile.out | Верно |
| /home/oracle/output_files | students.list | Верно |
| /tmp/1995 | january.results | Неверно — подкаталог /tmp/1995 недоступен |
| /home/oracle/ | output_files/classes.list | Неверно — подкаталог передается как часть имени файла |
| /TMP | myfile.out | Неверно — другой регистр символов |

**ВНИМАНИЕ** Даже если операционная система не учитывает регистр символов, сопоставление указанного каталога и доступных каталогов всегда производится с учетом регистра символов.

Если в файле INIT.ORA имеется строка

```
UTL_FILE_DIR = *
```

то полномочия на работу с базой данных отменяются. При этом все каталоги становятся доступны модулю UTL_FILE.

**ОСТОРОЖНО** Отмену полномочий на работу с базой данных следует производить крайне осторожно. Oracle не рекомендует использовать эту возможность в производственных системах, так как в такой ситуации могут быть отменены и полномочия операционной системы. Кроме того, не следует в качестве элемента списка доступных каталогов указывать "." (текущий каталог в системах Unix). Рекомендуется всегда использовать явное указание каталогов.

## Безопасность операционной системы

Операции файлового ввода/вывода выполняются модулем UTL_FILE в режиме пользователя Oracle (пользователь Oracle — это владелец файлов, обеспечивающих функционирование базы данных, и процессов, составляющих экземпляр базы данных). Следовательно, пользователь Oracle должен иметь привилегии операционной системы на чтение из всех доступных каталогов и на запись в эти каталоги. Если пользователю Oracle не предоставлены соответствующие привилегии на доступные каталоги, то выполнение всех операций над этими каталогами запрещается операционной системой.

Все файлы, создаваемые в результате работы модуля UTL_FILE, будут принадлежать пользователю Oracle. Кроме того, файлы будут создаваться с привилегиями, установленными операционной системой для пользователя Oracle по умолчанию. Если с этими файлами должны работать другие пользователи, обращаясь к ним не из модуля UTL_FILE, системный администратор обязан изменить полномочия на эти файлы.

**ОСТОРОЖНО** Рекомендуется запрещать операции записи в каталоги, входящие в список доступных каталогов. Единственным пользователем, которому следует предоставить полномочие записи в доступные каталоги, должен быть пользователь Oracle. Если пользователям разрешается запись в эти каталоги, то они получают возможность создавать символические связи с другими каталогами, тем самым обходя проверку привилегий, выполняемую операционной системой.

## Исключительные ситуации, устанавливаемые в UTL_FILE

Если при выполнении некоторой процедуры или функции модуля UTL_FILE возникает ошибка, то устанавливается исключительная ситуация. Возможные исключительные ситуации приведены в таблице 18.7. Обратите внимание на то, что в состав этих исключительных ситуаций входят семь определенных в модуле UTL_FILE и две стандартных исключительных ситуации (NO_DATA_FOUND и VALUE_ERROR). Исключительные ситуации UTL_FILE можно распознать по имени или с помощью обработчика исключительной ситуации OTHERS. Стандартные исключительные ситуации определяются по значениям, возвращаемым для них функцией SQLCODE.

**ТАБЛИЦА 18.7. Исключительные ситуации, устанавливаемые в UTL_FILE**

| Исключительная ситуация | Когда устанавливается | Чем устанавливается |
|---|---|---|
| INVALID_PATH | Имя каталога или имя файла неверно или недоступно | FOPEN |
| INVALID_MODE | Для режима файла указана неверная строка символов | FOPEN |
| INVALID_FILEHANDLE | Описатель файла не указывает на открытый файл | FCLOSE, GET_LINE, PUT, PUT_LINE NEW_LINE, PUTF, FFLUSH |
| INVALID_OPERATION | Файл нельзя открыть так, как это было запрошено. Возможная причина — полномочия операционной системы. Устанавливается также при попытке записи в файл, открытый на чтение, или при попытке чтения файла, открытого на запись | GET_LINE, PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH |
| READ_ERROR | Ошибка операционной системы во время операции чтения | GET_LINE |
| WRITE_ERROR | Ошибка операционной системы во время операции записи | PUT, PUT_LINE, NEW_LINE, FFLUSH, FCLOSE, FCLOSE_ALL |
| INTERNAL_ERROR | Неопределенная внутрисистемная ошибка | Все функции |
| NO_DATA_FOUND | Во время чтения достигнут конец файла | GET_LINE |
| VALUE_ERROR | Вводимая строка слишком велика для буфера, указанного в GET_LINE | GET_LINE |

## Открытие и закрытие файлов

Во всех операциях модуля UTL_FILE фигурирует понятие описателя файла. *Описатель, или логический номер, файла* (file handle) — это значение, которое используется в PL/SQL для идентификации файла, подобно идентификатору курсора в модуле DBMS_SQL. Все описатели файлов имеют тип UTL_FILE.FILE_TYPE, который определяется в спецификации модуля UTL_FILE. Описатели файлов возвращаются функцией FOPEN.

### FOPEN

Функция FOPEN открывает файл на ввод или на вывод. В любой момент времени файл может быть открыт или только на ввод, или только на вывод; одновременно на ввод и на вывод файл открывать нельзя. Описание FOPEN выглядит следующим образом:

```
FUNCTION FOPEN(location IN VARCHAR2,
               filename IN VARCHAR2,
               open_mode IN VARCHAR2);
  RETURN FILE_TYPE;
```

Каталог, путь к которому указывается, должен существовать на момент выполнения этой функции — FOPEN его не создает. Однако, если установить режим open_mode как 'w', то существующий файл будет перезаписываться. Параметры и возвращаемое значение функции FOPEN описаны в таблице 18.8. FOPEN может устанавливать любую из следующих исключительных ситуаций:

- UTL_FILE.INVALID_PATH

- UTL_FILE.INVALID_MODE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.INTERNAL_ERROR

**ТАБЛИЦА 18.8.**

| Параметр | Тип | Описание |
|---|---|---|
| *location* | VARCHAR2 | Путь к каталогу, в котором расположен файл. Если этот каталог не входит в число доступных, то устанавливается UTL_FILE.INVALID_PATH |
| *filename* | VARCHAR2 | Имя открываемого файла. Если задан режим 'w', то существующий файл перезаписывается |
| *open_mode* | VARCHAR2 | Используемый режим. Возможные значения:<br>'r' — чтение текста;<br>'w' — запись текста;<br>'a' — добавление текста.<br>Этот параметр не учитывает регистр символов. Если установлен режим 'a', а файл не существует, то он создается в режиме 'w' |
| *возвращаемое значение* | UTL_FILE.FILE_TYPE | Описатель файла, используемый впоследствии в функциях |

## FCLOSE

После окончания чтения из файла или записи в него этот файл должен быть закрыт с помощью процедуры FCLOSE. В результате освобождаются ресурсы, используемые модулем UTL_FILE для работы с файлом. Описание FCLOSE выглядит следующим образом:

```
PROCEDURE FCLOSE(file_handle IN OUT FILE_TYPE);
```

Единственным параметром этой процедуры является описатель файла. Все изменения, которые должны быть записаны в файл, завершаются перед его закрытием. В случае ошибки записи устанавливается UTL_FILE.WRITE_ERROR. Если описатель не идентифицирует корректный открытый файл, то устанавливается UTL_FILE.INVALID_FILEHANDLE.

## IS_OPEN

Эта логическая функция возвращает TRUE, если указанный файл открыт, и FALSE — в противном случае. Описание функции IS_OPEN таково:

```
FUNCTION IS_OPEN(file_handle IN FILE_TYPE)
RETURN BOOLEAN;
```

Даже когда IS_OPEN возвращает TRUE, при работе с данным файлом могут происходить ошибки операционной системы.

## FCLOSE_ALL

Процедура FCLOSE_ALL закрывает все открытые файлы. Она предназначена для удаления ставшей ненужной информации, что особенно полезно в обработчиках исключительных ситуаций:

```
PROCEDURE FCLOSE_ALL;
```

Эта процедура не имеет никаких аргументов. Все ожидающие записи изменения сбрасываются на диск перед закрытием файлов. Поэтому в случае ошибки записи FCLOSE_ALL может устанавливать исключительную ситуацию UTL_FILE.WRITE_ERROR.

▼ ОСТОРОЖНО Процедура FCLOSE_ALL закрывает файлы и высвобождает ресурсы, используемые модулем UTL_FILE, но не помечает файлы как закрытые — IS_OPEN будет по-прежнему возвращать TRUE после выполнения FCLOSE_ALL. После FCLOSE_ALL все операции чтения и записи будут заканчиваться неудачей, если только файл не открыт по-новому с помощью функции FOPEN.

## Файловый вывод

Для вывода данных в файлы используются пять процедур: PUT, PUT_LINE, NEW_LINE, PUTF и FFLUSH. Работа процедур PUT, PUT_LINE и NEW_LINE сходна с работой их аналогов, входящих в состав модуля DBMS_OUTPUT (см. главу 14). Максимальный размер выходной записи равен 1023 байтам.

### PUT

Процедура PUT выводит указанную строку символов в указанный файл, причем данный файл должен быть открыт на запись:

```
PROCEDURE PUT(file_handle IN FILE_TYPE,
              buffer IN VARCHAR2);
```

Символ новой строки в файл не вводится. Для того чтобы включить в файл признак конца строки, используйте процедуры PUT_LINE или NEW_LINE. В случае ошибки записи устанавливается UTL_FILE.WRITE_ERROR. Параметры процедуры PUT описаны в таблице 18.9.

**ТАБЛИЦА 18.9.**

| Параметр | Тип | Описание |
|---|---|---|
| *file_handle* | UTL_FILE.FILE_TYPE | Описатель файла, возвращаемый функцией FOPEN. Если этот описатель некорректен, то устанавливается UTL_FILE.INVALID_FILEHANDLE |
| *buffer* | VARCHAR2 | Текстовая строка символов, выводимая в файл. Если файл не был открыт в режиме 'w' или 'a', то устанавливается UTL_FILE.INVALID_OPERATION |

### NEW_LINE

Процедура NEW_LINE записывает один или несколько признаков конца строки в указанный файл:

```
PROCEDURE NEW_LINE(file_handle IN FILE_TYPE,
                   lines IN NATURAL := 1);
```

Признак конца строки зависит от используемой системы — в разных операционных системах признаки конца строки различны. В случае ошибки записи устанавливается UTL_FILE.WRITE_ERROR. Параметры процедуры NEW_LINE описаны в таблице 18.10.

**ТАБЛИЦА 18.10.**

| Параметр | Тип | Описание |
|---|---|---|
| *file_handle* | UTL_FILE.FILE_TYPE | Описатель файла, возвращаемый функцией FOPEN. Если описатель некорректен, то устанавливается UTL_FILE.INVALID_FILEHANDLE |
| *lines* | NATURAL | Число выводимых признаков конца строки. Значение по умолчанию равно 1, что соответствует выводу одного символа новой строки. Если файл не был открыт в режиме 'w' или 'a', то устанавливается UTL_FILE.INVALID_OPERATION |

### PUT_LINE

Процедура PUT_LINE выводит указанную последовательность символов в данный файл, который должен быть открыт на запись. После вывода последовательности выводится символ новой строки:

```
PROCEDURE PUT_LINE(file_handle IN FILE_TYPE,
                   buffer IN VARCHAR2);
```

Параметры процедуры PUT_LINE описаны в таблице 18.11. Вызов PUT_LINE эквивалентен вызову PUT с последующим вызовом NEW_LINE для вывода символа новой строки. В случае ошибки записи устанавливается UTL_FILE.WRITE_ERROR.

**ТАБЛИЦА 18.11.**

| Параметр | Тип | Описание |
|---|---|---|
| *file_handle* | UTL_FILE.FILE_TYPE | Описатель файла, возвращаемый функцией FOPEN. Если описатель некорректен, то устанавливается UTL_FILE.INVALID_FILEHANDLE |
| *buffer* | VARCHAR2 | Текстовая строка символов, выводимая в файл. Если файл не был открыт в режиме 'w' или 'a', то устанавливается UTL_FILE.INVALID_OPERATION |

## PUTF

Процедура PUTF подобна PUT, но позволяет форматировать выходную строку символов. PUTF – это сокращенный вариант функции *printf()*, применяемой в C, и имеет аналогичный синтаксис:

```
PROCEDURE PUTF(file_handle IN FILE_TYPE,
               format IN VARCHAR2,
               arg1 IN VARCHAR2 DEFAULT NULL,
               arg2 IN VARCHAR2 DEFAULT NULL,
               arg3 IN VARCHAR2 DEFAULT NULL,
               arg4 IN VARCHAR2 DEFAULT NULL,
               arg5 IN VARCHAR2 DEFAULT NULL);
```

В строке format содержится обычный текст и специальные символы форматирования %s и \n. Каждое вхождение %s в этой строке замещается одним из необязательных аргументов, а каждое вхождение \n – символом новой строки. Параметры процедуры PUTF описаны в таблице 18.12. В случае ошибки записи устанавливается UTL_FILE.WRITE_ERROR.

Например, если выполнить блок

```
DECLARE
  v_OutputFile UTL_FILE.FILE_TYPE;
  v_Name VARCHAR2(10) := 'Scott';
BEGIN
  v_OutputFile := UTL_FILE.FOPEN(...);
  UTL_FILE.PUTF(v_OutputFile,
      'Hi there!\nMy name is %s, and I am a %s major.\n',
      v_Name, 'Computer Science');
  FCLOSE(v_OutputFile);
END;
```

то результат будет выглядеть следующим образом:

```
Hi there! My name is Scott, and I am a Computer Science major.
```

**ТАБЛИЦА 18.12.**

| Параметр | Тип | Описание |
|---|---|---|
| *file_handle* | UTL_FILE.FILE_TYPE | Описатель файла, возвращаемый функцией FOPEN. Если описатель некорректен, то устанавливается UTL_FILE.INVALID_FILEHANDLE |
| *format* | VARCHAR2 | Строка символов, содержащая обычный текст и, возможно, специальные символы форматирования %s и \n. Если файл не был открыт в режиме 'w' или 'a', то устанавливается UTL_FILE.INVALID_OPERATION |
| *arg1...arg5* | VARCHAR2 | От одного до пяти необязательных аргументов. Каждый из них будет замещать соответствующий символ форматирования %s. Если число символов форматирования больше числа аргументов, то вместо символа %s будет вводиться пустая строка (NULL) |

### FFLUSH

С помощью процедур PUT, PUT_LINE, PUTF или NEW_LINE данные выводятся через буфер. Когда буфер заполняется, данные физически сбрасываются (flush) в файл. Процедура FFLUSH записывает содержимое буфера в файл немедленно. Описание этой процедуры выглядит следующим образом:

```
PROCEDURE FFLUSH(file_handle IN FILE_TYPE);
```

FFLUSH может устанавливать любую из следующих исключительных ситуаций:

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.WRITE _ERROR

## Файловый ввод

Процедура GET_LINE используется для чтения из файла, а не для записи в файл. Одна строка текста считывается из указанного файла и помещается в параметр *buffer*. Символ новой строки в возвращаемую последовательность не включается. Описание процедуры GET_LINE выглядит следующим образом:

```
PROCEDURE GET_LINE(file_handle IN FILE_TYPE,
                   buffer OUT VARCHAR2);
```

Когда из файла считывается последняя строка, устанавливается исключительная ситуация NO_DATA_FOUND. Если строка не помещается в буфер, размер которого указан как фактический параметр, устанавливается VALUE_ERROR. При чтении пустой строки возвращается пустая последовательность символов (NULL). В случае ошибки записи устанавливается UTL_FILE.READ_ERROR. Параметры процедуры GET_LINE описаны в таблице 18.13.

**ТАБЛИЦА 18.13.**

| Параметр | Тип | Описание |
|---|---|---|
| *file_handle* | UTL_FILE.FILE_TYPE | Описатель файла, возвращаемый функцией FOPEN. Если описатель некорректен, то устанавливается UTL_FILE.INVALID_FILEHANDLE |
| *buffer* | VARCHAR2 | Буфер, в который записывается строка. Если файл не был открыт на чтение ('r'), то устанавливается UTL_FILE.INVALID_OPERATION |

## Примеры

В этом разделе описано три примера использования модуля UTL_FILE. Первый является еще одним вариантом рассматривавшегося ранее модуля Debug. Во втором примере показано, как информация о студентах считывается в файле и загружается в таблицу. В третьем примере отображаются характеристики студентов.

### Модуль Debug

Создадим еще один вариант модуля Debug, воспользовавшись модулем UTL_FILE:

```
-- Этот пример содержится в файле debug.sql.
CREATE OR REPLACE PACKAGE Debug AS
  /* Глобальные переменные для хранения имени отладочного файла и
     имени каталога. */
  v_DebugDir VARCHAR2(50);
  v_DebugFile VARCHAR2(20);
  PROCEDURE Debug(p_Description IN VARCHAR2,
                  p_Value IN VARCHAR2);
  PROCEDURE Reset(p_NewFile IN VARCHAR2 := v_DebugFile,
                  p_NewDir IN VARCHAR2 := v_DebugDir) ;
  /* Закрывает отладочный файл. */
  PROCEDURE Close;
END Debug;

CREATE OR REPLACE PACKAGE BODY Debug AS
```

```
    v_DebugHandle  UTL_FILE.FILE_TYPE;

    PROCEDURE  Debug(p_Description  IN  VARCHAR2,
                     p_Value  IN  VARCHAR2)  IS
    BEGIN
        /*  Выведем информацию в буфер и запишем ее в файл.  */
        UTL_FILE.PUTF(v_DebugHandle,  '%s:  %s\n',
                      p_Description,  p_Value);
        UTL_FILE.FFLUSH(v_DebugHandle);
    EXCEPTION
        WHEN  UTL_FILE.INVALID_OPERATION  THEN
          RAISE_APPLICATION_ERROR(-20102,
                                  'Debug:  Invalid  Operation');
        WHEN  UTL_FILE.INVALID_FILEHANDLE  THEN
          RAISE_APPLICATION_ERROR(-20103,
                                  'Debug:  Invalid  File  Handle');
        WHEN  UTL_FILE.WRITE_ERROR  THEN
          RAISE_APPLICATION_ERROR(-20104,
                                  'Debug:  Write  Error');
    END  Debug;

    PROCEDURE  Reset(p_NewFile  IN  VARCHAR2  :=  v_DebugFile,
                     p_NewDir  IN  VARCHAR2  :=  v_DebugDir)  IS

    BEGIN
        /*  Вначале убедимся в том, что файл закрыт.  */
        IF  UTL_FILE.IS_OPEN(v_DebugHandle)  THEN
          UTL_FILE.FCLOSE(v_DebugHandle);
        END  IF;

        /*  Откроем файл на запись.  */
        v_DebugHandle  :=  UTL_FILE.FOPEN(p_NewDir,  p_NewFile,  'w');

        /*  Установим в модульных переменных только что полученные значения.  */
        v_DebugFile  :=  p_NewFile;
        v_DebugDir  :=  p_NewDir;
    EXCEPTION
        WHEN  UTL_FILE.INVALID_PATH  THEN
          RAISE_APPLICATION_ERROR(-20100,  'Reset:  Invalid  Path');
        WHEN  UTL_FILE.INVALID_MODE  THEN
          RAISE_APPLICATION_ERROR(-20101,  'Reset:  Invalid  Mode');
        WHEN  UTL_FILE.INVALID_OPERATION  THEN
          RAISE_APPLICATION_ERROR(-20101,  'Reset:  Invalid  Operation');
    END  Reset;

    PROCEDURE  Close  IS
    BEGIN
        UTL_FILE.FCLOSE(v_DebugHandle);
    END  Close;

    BEGIN
        v_DebugDir  :=  '/tmp';  v_DebugFile  :=  'debug.out';
    Reset;
END  Debug;
```

Этот вариант модуля Debug работает практически так же, как и варианты, рассмотренные в главах 13 и 14, но с незначительными изменениями. В процедуре **Debug.Reset** имя и местонахождение отладочного файла задаются как параметры; если же эти параметры не указываются, то по умолчанию выходным файлом будет файл /tmp/debug.out. Каждый отладочный оператор будет добавлять в этот файл отдельную строку. Кроме того, создана новая процедура – **Debug.Close.** Эту процедуру следует вызывать для закрытия отладочного файла. Хотя процедура **Debug.Debug** сбрасывает информацию в выходной файл, его нужно закрывать, что высвобождает занятые им ресурсы.

▼ СОВЕТУЕМ Обратите внимание на обработчики исключительных ситуаций в разных подпрограммах. В них указаны реальные ошибки, которые могут произойти, а также процедуры, их устанавливающие. При работе с модулем UTL_FILE рекомендуется всегда следовать этому правилу, иначе для обнаружения ошибок придется использовать обработчик WHEN OTHERS.

## Загрузка информации о студентах

Процедура **LoadStudents** вводит данные, считываемые из файла, в таблицу **students.** Содержимое файла разграничено запятыми, т.е. каждая запись содержится в отдельной строке текста, а поля отделены одно от другого запятой. Такой формат является обычным для текстовых файлов. Создадим процедуру **LoadStudents**:

```
-- Этот пример содержится в файле loadstud.sql.
CREATE OR REPLACE PROCEDURE LoadStudents (
  /* Загружает информацию, считываемую из разграниченного запятыми файла,
     в таблицу students. Строки файла выглядят следующим образом:

     first_name,last_name,major

     Идентификаторы студентов генерируются последовательностью
     student_sequence. Общее число вводимых строк возвращается в
     p_TotalInserted. */
  p_FileDir IN VARCHAR2,
  p_FileName IN VARCHAR2,
  p_TotalInserted IN OUT NUMBER) AS

  v_FileHandle UTL_FILE.FILE_TYPE;
  v_NewLine VARCHAR2(100); -- Входная строка.
  v_FirstName students.first_name%TYPE;
  v_LastName students.last_name%TYPE;
  v_Major students.major%TYPE;
  /* Позиции запятых во входном файле. */
  v_FirstComma NUMBER;
  v_SecondComma NUMBER;

BEGIN
  -- Откроем указанный файл на чтение.
  v_FileHandle := UTL_FILE.FOPEN(p_FileDir, p_FileName, 'r');

  -- Инициализируем выходную переменную для числа студентов.
  p_TotalInserted := 0;

  -- Последовательно просмотрим содержимое файла, считывая каждую строку.
  -- В конце GET_LINE установит NO_DATA_FOUND, поэтому воспользуемся
  -- этой исключительной ситуацией в качестве условия выхода из цикла.
  LOOP
    BEGIN
      UTL_FILE.GET_LINE(v_FileHandle, v_NewLine);
    EXCEPTION
```

```
        WHEN NO_DATA_FOUND THEN
          EXIT;
      END;

      -- Каждое поле входной записи ограничено запятыми. Нужно определить
      -- местоположение двух запятых в строке и использовать полученные
      -- результаты для выбора полей из v_NewLine. Для определения позиций
      --запятых воспользуемся функцией INSTR.
      v_FirstComma := INSTR(v_NewLine, ',', 1, 1);
      v_SecondComma := INSTR(v_NewLine, ',', 1, 2);
      -- Теперь для выделения полей воспользуемся функцией SUBSTR.
      v_FirstName := SUBSTR(v_NewLine, 1, v_FirstComma - 1);
      v_LastName := SUBSTR(v_NewLine, v_FirstComma + 1,
                           v_SecondComma - v_FirstComma - 1);
      v_Major := SUBSTR(v_NewLine, v_SecondComma + 1);

      -- Введем новую запись в таблицу students.
      INSERT INTO students (ID, first_name, last_name, major)
        VALUES (student_sequence.nextval, v_FirstName,
                v_LastName, v_Major);

      p_TotalInserted := p_TotalInserted + 1;
    END LOOP;

    -- Закроем файл.
    UTL_FILE.FCLOSE(v_FileHandle);

    COMMIT;
  EXCEPTION
    -- Обработаем исключительные ситуации UTL_FILE, указав необходимую
    -- информацию, и убедимся, что файл закрыт надлежащим образом.
    WHEN UTL_FILE.INVALID_OPERATION THEN
      UTL_FILE.FCLOSE(v_FileHandle);
        RAISE_APPLICATION_ERROR(-20051,
                                'LoadStudents: Invalid Operation');
    WHEN UTL_FILE.INVALID_FILEHANDLE THEN
      UTL_FILE.FCLOSE(v_FileHandle);
        RAISE_APPLICATION_ERROR(-20052,
                                'LoadStudents: Invalid File Handle');
    WHEN UTL_FILE.READ_ERROR THEN
      UTL_FILE.FCLOSE(v_FileHandle);
        RAISE_APPLICATION_ERROR(-20053,
                                'LoadStudents: Read Error');
    WHEN OTHERS THEN
      UTL_FILE.FCLOSE(v_FileHandle);
        RAISE;
  END LoadStudents;
```

**Ниже приведен возможный результат выполнения LoadStudents.**

```
Scott, Smith, Computer Science
Margaret, Mason, History
Joanne, Junebug, Computer Science
Manish, Murgratroid, Economics
Patrick, Poll, History
```

```
Timothy, Taller, History
Barbara, Blues, Economics
David, Dinsmore, Music
Ester, Elegant, Nutrition
Rose, Riznit, Music
Rita, Razmataz, Nutrition
```

▼ ВНИМАНИЕ Для определения идентификаторов студентов в процедуре **LoadStudents** используются порядковые номера последовательности **student_sequence.** Если эту последовательность не инициализировать, то она может возвращать значения, которые уже содержатся в таблице **students**. В этом случае нарушается ограничение первичного ключа данной таблицы.

## Печать характеристик студентов

Процедура **PrintTranscript** впервые была рассмотрена в главе 13. Теперь, когда известно, как работать с модулем UTL_FILE, создание этой процедуры завершается. Сначала нужно описать процедуру **CalculateGPA**:

```
-- Этот пример содержится в файле calcgpa.sql.
CREATE OR REPLACE PROCEDURE CalculateGPA (
  /* Записывает в p_GPA среднюю оценку успеваемости студента,
     указанного в p_StudentID. . */
  p_StudentID IN students.ID%TYPE,
  p_GPA OUT NUMBER) AS

  CURSOR c_ClassDetails IS
      SELECT classes.num_credits, rs.grade
        FROM classes, registered_students rs
        WHERE classes.department = rs.department
        AND classes.course = rs.course
        AND rs.student_id = p_StudentID;

  v_NumericGrade NUMBER;
  v_TotalCredits NUMBER := 0;
  v_TotalGrade NUMBER := 0;
BEGIN
  FOR v_ClassRecord in c_ClassDetails LOOP
      -- Определим числовое значение оценки.
        SELECT DECODE(v_ClassRecord.grade,  'A', 4,
                                            'B', 3,
                                            'C', 2,
                                            'D', 1,
                                            'E', 0)
          INTO v_NumericGrade
        FROM dual;

        v_TotalCredits := v_TotalCredits + v_ClassRecord.num_credits;
        v_TotalGrade := v_TotalGrade +
                        (v_ClassRecord.num_credits * v_NumericGrade);
      END LOOP;
      p_GPA := v_TotalGrade / v_TotalCredits;
END CalculateGPA;
```

Теперь создадим **PrintTranscript**:

```
-- Этот пример содержится в файле printran.sql.
CREATE OR REPLACE PROCEDURE PrintTranscript (
```

```
    /* Выводит характеристику указанного студента. В ней указываются группы,
       в которых зарегистрирован студент, и его оценки, полученные в каждой
       из групп. В конце характеристики выводится оценка GPA. */
    p_StudentID IN students.ID%TYPE,
    P_FileDir IN VARCHAR2,
    p_FileName IN VARCHAR2) AS

    v_StudentGPA NUMBER;
    v_StudentRecord students%ROWTYPE;
    v_FileHandle UTL_FILE.FILE_TYPE;
    v_NumCredits NUMBER;

    CURSOR c_CurrentClasses IS
        SELECT *
          FROM registered_students
          WHERE student_id = p_StudentID;
BEGIN
    -- Откроем файл в режиме добавления.
    v_FileHandle := UTL_FILE.FOPEN(p_FileDir, p_FileName, 'w');

    SELECT *
        INTO v_StudentRecord
        FROM students
        WHERE ID = p_StudentID;

    -- Информация выходного заголовка: текущие дата и время, а также
    -- сведения о студенте.

    UTL_FILE.PUTF(v_FileHandle, 'Student ID: %s\n',
        v_StudentRecord.ID);
    UTL_FILE.PUTF(v_FileHandle, 'Student Name: %s %s\n',
        v_StudentRecord.first_name, v_StudentRecord.last_name);
    UTL_FILE.PUTF(v_FileHandle, 'Major: %s\n',
        v_StudentRecord.major);
    UTL_FILE.PUTF(v_FileHandle, 'Transcript Printed on: %s\n\n\n',
        TO_CHAR(SYSDATE, 'Mon DD,YYYY HH24:MI:SS'));

    UTL_FILE.PUT_LINE(v_FileHandle, 'Class Credits Grade');
    UTL_FILE.PUT_LINE(v_FileHandle, '------ ------ -----');
    FOR v_ClassesRecord in c_CurrentClasses LOOP
        -- Определим число зачетов для этой группы.
        SELECT num_credits
          INTO v_NumCredits
          FROM classes
          WHERE course = v_ClassesRecord.course
          AND department = v_ClassesRecord.department;

        -- Выведем информацию о данной группе.
        UTL_FILE.PUTF(v_FileHandle, '%s %s %s\n',
      '    RPAD(v_ClassesRecord.department || ' ' ||
              v_ClassesRecord.course, 7),
          LPAD(v_NumCredits, 7),
          LPAD(v_ClassesRecord.grade, 5));
    END LOOP;
```

```
    -- Вычислим оценку GPA.
    CalculateGPA(p_StudentID, v_StudentGPA);

    -- Выведем оценку GPA.
    UTL_FILE.PUTF(v_FileHandle, '\n\nCurrent GPA: %s\n',
        TO_CHAR(v_StudentGPA, '9.99'));

    -- Закроем файл.
    UTL_FILE.FCLOSE(v_FileHandle);

  EXCEPTION
    -- Обработаем исключительные ситуации UTL_FILE, указав необходимую
    -- информацию, и убедимся, что файл закрыт надлежащим образом.
    WHEN UTL_FILE.INVALID_OPERATION THEN
        UTL_FILE.FCLOSE(v_FileHandle);
          RAISE_APPLICATION_ERROR(-20061,
                                  'PrintTranscript: Invalid Operation');
    WHEN UTL_FILE.INVALID_FILEHANDLE THEN
        UTL_FILE.FCLOSE(v_FileHandle);
          RAISE_APPLICATION_ERROR(-20062,
                                  'PrintTranscript: Invalid File Handle');
    WHEN UTL_FILE.WRITE_ERROR THEN
        UTL_FILE.FCLOSE(v_FileHandle);
          RAISE_APPLICATION_ERROR(-20063,
                                  'PrintTranscript: Write Error');
    WHEN OTHERS THEN
        UTL_FILE.FCLOSE(v_FileHandle);
          RAISE;
  END PrintTranscript;
```

Если таблица **registered_students** выглядит следующим образом:

```
SQL> SELECT * FROM registered_students;
  STUDENT_ID  DEP   COURSE  G
-------------------  ------  -
       10002   CS       102  B
       10002   HIS      101  B
       10002   ECN      203  A
       10002   CS       101  A
       10009   HIS      101  D
       10009   MUS      410  B
       10009   HIS      301  C
       10009   MUS      410  B

8 rows selected.
```

и если вызвать **PrintTranscript** для студентов 10002 и 10009, то будет получено два выходных файла:

```
Student ID: 10002
Student Name: Joanne Junebug
Major: Computer Science
Transcript Printed on: Jan 27,1996 17:37:43

Class    Credits Grade
-------  ------- -----
CS 102         4     B
```

```
HIS 101       4    B
ECN 203       3    A
CS 101        4    A

Current GPA: 3.47

Student ID: 10009
Student Name: Rose Riznit
Major: Music
Transcript Printed on: Jan 27,1996 17:38:56

Class    Credits Grade
------   ------- ------
HIS 101       4    D
MUS 410       3    B
HIS 301       4    C
MUS 410       3    B

Current GPA: 2.14
```

# Итоги

В этой главе было рассказано еще о двух служебных модулях: DBMS_JOB и UTL_FILE. Задания для баз данных позволяют автоматически выполнять процедуры в заранее установленное время. UTL_FILE расширяет возможности PL/SQL, предоставляя средство файлового ввода/вывода при условии обеспечения необходимой безопасности информации на сервере. Каждая из этих утилит реализует полезное функциональное средство, отсутствующее в данном языке.

# Глава 19

## *Программа Oracle WebServer*

Программа Oracle WebServer имеет больше функциональных возможностей, чем обычные Web-серверы. Интегрированная с базой данных Oracle, она позволяет создавать страницы формата HTML посредством хранимых процедур PL/SQL. С ее помощью можно генерировать динамические Web-страницы, используя как вводимую пользователями информацию, так и хранящиеся в базе данные. В этой главе рассмотрена структура Web-сервера Oracle и работа с ним PL/SQL, а также используемые при этом структуры данных PL/SQL. Кроме того, рассказано о программе SQL-Station Code, предназначенной для выполнения программ PL/SQL в глобальной гипертекстовой Web-системе.

# Среда WebServer

▼ ВНИМАНИЕ

Вся эта глава, в том числе и примеры, написана для работы с Web-сервером Oracle версии 2.1 (примеры подготовлены в этой же версии). Пробную версию Web-сервера для систем Solaris и Windows NT можно найти на сервере www.oracle.com и на компакт-диске, прилагаемом к этой книге. Информация, приведенная в этой главе, справедлива и для Web-сервера Oracle версии 3.0 (называемого Web-сервером приложений).

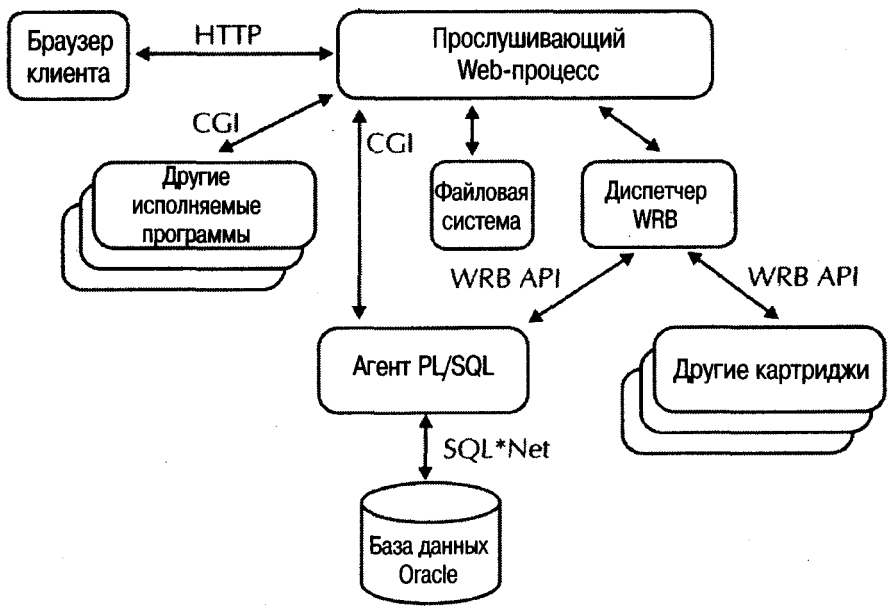
Oracle WebServer является более сложной и более мощной программой, чем обычные Web-серверы. В этой главе рассказывается о программных компонентах, составляющих Web-сервер, и об их взаимодействии между собой и с базой данных Oracle. Полное обсуждение Web-сервера и интерфейсов связи с ним не является предметом этой книги; более подробную информацию о работе с Web-сервером можно найти во встроенной документации, доступной при инсталляции сервера.

Ниже перечислены компоненты программы Oracle WebServer, которые изображены на рис. 19.1 и описаны в последующих разделах:

- Прослушивающий Web-процесс (web listener)
- Интерфейс CGI (Common Gateway Interface – общий шлюзовый интерфейс)
- Интерфейс WRB (Web Request Broker – программа – посредник обработки Web-запросов)
- Диспетчер WRB (WRB dispatcher)
- Агент PL/SQL (PL/SQL Agent)

**Прослушивающий Web-процесс** Прослушивающий Web-процесс отвечает за получение указателя URL (Uniform Resource Locator – унифицированный указатель ресурсов) от программы-браузера клиента и посылку назад соответствующей выходной информации. После обработки запрошенного

**Рис. 19.1**
*Компоненты программы Oracle WebServer*



указателя URL определяется, какой дополнительный компонент Web-сервера более всего подходит для получения нужного результата. К примеру, результат может быть текстовым файлом, хранимым на диске средствами операционной системы, или выходными данными некоторой программы. Кроме того, результат может быть выдан интерфейсом WRB.

**Общий шлюзовый интерфейс (CGI)** Web-сервер предназначен для обслуживания запросов, выдаваемых программой-браузером клиента. Ответом, как правило, являются страницы формата HTML. CGI – это интерфейс для программ, работающих на сервере и создающих документы HTML, а не просто считывающих статичные текстовые файлы операционной системы. С помощью CGI Web-страницы можно создавать динамически, поэтому CGI часто используется для реализации форм HTML.

**Интерфейс-посредник обработки Web-запросов (WRB)** Интерфейс WRB – это вспомогательный интерфейс, разрешающий прослушивающему Web-процессу вызывать исполняемые программы. Когда прослушивающий процесс обнаруживает, что нужен интерфейс WRB, он передает управление обработкой диспетчеру WRB и возвращается к обработке входящих запросов. Интерфейс WRB используется только в Oracle WebServer, в то время как CGI является компонентом всех Web-серверов.

**Диспетчер WRB** Диспетчер WRB управляет входящими запросами WRB с помощью группы процессов WRBX (WRB executable engines – исполняемых систем WRB). Система WRBX взаимодействует с внутренним приложением, называемым картриджем. Картридж (cartridge) – это специальное приложение, разработанное для выполнения конкретной задачи и взаимодействующее с WRBX (следовательно, и с клиентом – через WRBX и прослушивающий процесс) при помощи открытого интерфейса прикладных программ (API) WRB. Диспетчер также используется только в программе Oracle WebServer.

**Агент PL/SQL** Агент PL/SQL – это последнее звено в цепочке, связывающей браузер клиента и Oracle-сервер. Агент вызывает хранимую процедуру Oracle, результатом работы которой является динамическая страница HTML, после чего он передает полученный результат обратно клиенту через прослушивающий Web-процесс. Агент может использовать в работе либо интерфейс CGI, либо интерфейс WRB.

# Агент PL/SQL

Агент PL/SQL является основным средством организации доступа к базе данных Oracle. Когда прослушивающий Web-процесс получает указатель URL, где сказано о том, что нужно вызвать агента PL/SQL, агент устанавливает соединение с базой данных и вызывает хранимую процедуру PL/SQL. Информацию можно передавать процедуре через ее параметры, причем в эту информацию можно включать сведения, указанные в URL, или информацию, генерируемую формой HTML. Процедура, в свою очередь, генерирует выходные данные HTML с помощью группы специальных модулей, которые поставляются вместе с Web-сервером.

## Описатель соединения с базой данных

Для того чтобы вызвать хранимую процедуру, агент PL/SQL должен установить соединение с базой данных. Для этого ему необходимо знать нужную информацию: имя и пароль пользователя, а также системный идентификатор базы данных или строку соединения SQL*Net. Такая информация хранится в *описателе соединения с базой данных* (database connection descriptor, DCD). Поля DCD представлены в таблице 19.1.

**ТАБЛИЦА 19.1. Поля описателя соединения с базой данных**

| Поле | Описание |
|---|---|
| username | Пользователь Oracle, с которым устанавливается соединение. Этот пользователь должен быть владельцем всех хранимых процедур, вызываемых с данным DCD |
| password | Пароль этого пользователя |
| ORACLE_HOME | Корневой каталог для программ Oracle в операционной системе сервера |
| ORACLE_SID | Системный идентификатор (SID) локальной базы данных, если DCD указывает на эту базу данных |
| SQL*Net V2 connect string | Если DCD указывает на удаленную базу данных, то строка соединения SQL*Net версии 2 |
| owa_error_page | Полный маршрут к странице HTML, которую Web-агент будет возвращать в случае ошибки операционной системы или базы данных. Заметьте, что это не виртуальный маршрут для Web-сервера, а маршрут, указываемый в стандарте операционной системы |

**ТАБЛИЦА 19.1. Поля описателя соединения с базой данных (продолжение)**

| Поле | Описание |
|---|---|
| owa_valid_ports | Перечень действующих портов, через которые будет посылать ответы прослушивающий процесс. В URL можно указывать различные порты |
| owa_log_dir | Каталог регистрации, куда агент PL/SQL будет записывать файл регистрации. Этим файлом можно пользоваться для диагностики ошибок, а также действий процесса регистрации |
| owa_nls_lang | Среда NLS_LANG для сеанса Oracle. Это поле указывается в том же формате, что и для стандартного соединения Oracle, например в формате AMERICAN_AMERICA.US7ASCII. Сведения о NLS_LANG приведены в руководстве Oracle Server Reference |

Значения для DCD можно устанавливать на странице Administration Web-сервера, генерируемой при инсталляции Web-сервера и позволяющей создавать и модифицировать описатели DCD в диалоговом режиме. Поля DCD можно видоизменять, редактируя файл конфигурации в файловой системе сервера. Более подробно об описателях DCD и о файлах конфигурации рассказано в документации по Oracle WebServer.

## Сравнение CGI и WRB

Агент PL/SQL можно вызвать с помощью одного из интерфейсов: CGI и WRB. В любом случае для указания информации о соединении с базой данных применяется описатель DCD. Какой же из этих интерфейсов лучше? При использовании интерфейса WRB процесс WRBX устанавливает соединение в два этапа:

1. Соединение устанавливается сразу же после создания WRBX.
2. Для обработки каждого запроса WRBX регистрируется в базе данных. Когда обработка запроса завершается, WRBX отключается. Соединение остается активным.

При использовании интерфейса CGI агент PL/SQL должен выполнять оба этапа соединения с базой данных для каждого запроса. Дело в том, что каждый запрос CGI порождает процесс, который должен начинаться заново. Процесс же WRBX остается активен, поэтому нужно выполнить только второй этап.

Поскольку первый этап более продолжителен, использование интерфейса WRB значительно повышает быстродействие всей системы; поэтому рекомендуется работать именно с этим интерфейсом.

# Указание параметров процедур

Процедуру PL/SQL можно вызывать через Web-агент одним из методов: GET или POST. Если используется метод GET, то параметры передаются непосредственно в URL. Метод POST применяется в тех случаях, когда процедура является адресатом формы HTML. Web-агент автоматически определяет используемый метод и вызывает процедуру. Ниже приведен пример, в котором применяется метод GET. Метод POST используется автоматически при работе с формами HTML и описан далее в этой главе. Рассмотрим следующую хранимую процедуру:

```
-- Этот пример содержится в файле hello.sql.
CREATE OR REPLACE PROCEDURE hello(p_Greeting IN VARCHAR2) AS
BEGIN
  HTP.htmlOpen;
  HTP.headOpen;
  HTP.Title('Hello World!');
  HTP.headClose;
  HTP.bodyOpen;
  HTP.print(p_Greeting);
  HTP.bodyClose;
  HTP.htmlClose;
END hello;
```

Вызовы модуля HTP генерируют реальный код HTML (об этом рассказано в разделе "Web-пакет PL/SQL"). Для того чтобы вызвать эту процедуру из URL, необходимо выполнить следующие действия:

1. Создать процедуру в конкретной схеме. Предположим, что в нашем случае имя пользователя базы данных — **example**.

2. Создать для прослушивающего Web-процесса описатель DCD, в котором **example** является идентификатором пользователя. Предположим, что DCD также называется **example**.
3. Создать WRB и указать в качестве вызова Web-агента Oracle виртуальный маршрут /example/owa.

Теперь можно вызвать процедуру **hello,** например, так:

```
http://хост_машина:порт/example/owa/hello?p_Greeting=Welcome!
```

где *хост_машина* и *порт* – это соответственно машина и порт, в которых функционирует прослушивающий процесс. Этот процесс выполнит синтаксический анализ предоставленного ему указателя URL и определит дальнейший ход обработки на основании компонентов URL, каждый из которых можно считывать с помощью соответствующей переменной среды CGI. Компоненты URL описаны в таблице 19.2.

**ТАБЛИЦА 19.2.**

| Компонент URL | Переменная среды CGI | Назначение |
|---|---|---|
| /example/owa | SCRIPT_NAME | Сообщает прослушивающему процессу о том, что следует использовать DCD example. WRB указывает виртуальный маршрут как /example/owa, поэтому Web-процесс обрабатывает оставшуюся часть URL |
| hello | PATH_INFO | Идентифицирует вызываемую процедуру. Web-агент будет ожидать, что эта процедура выдаст выходную информацию HTML с помощью модулей Web-пакета |
| p_Greeting=Welcome! | QUERY_STRING | Идентифицирует параметры процедуры. Обратите внимание: имя параметра указано вместе со значением. Если параметров несколько, то их нужно разделять знаками вопроса |

Когда процедура **hello** вызывается в Web-браузере из этого URL, результат отображается в окне программы, например Netscape Navigator 3.0, в виде, представленном на рис. 19.2. Код HTML, генерируемый процедурой **hello**, будет выглядеть примерно так:

```
<HTML>
<HEAD>
<TITLE>Hello World!</TITLE>
</HEAD>
```

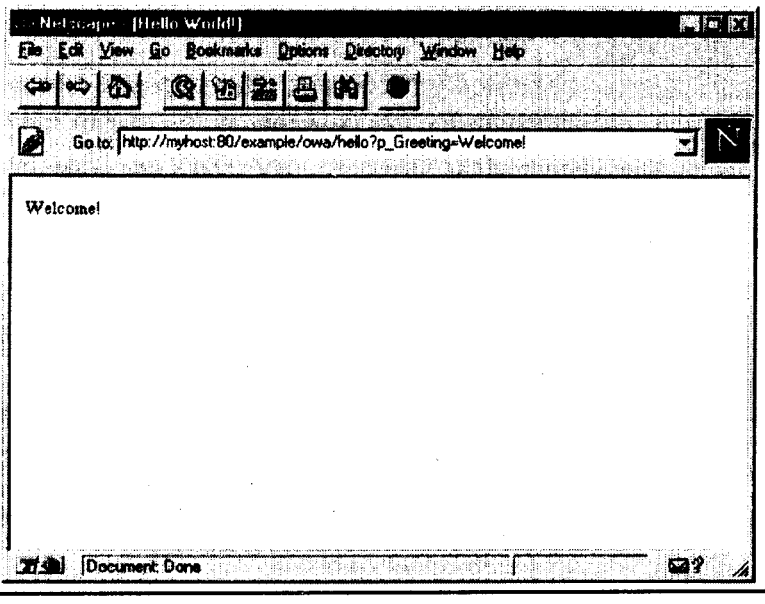



**Рис. 19.2.**
*Результат выполнения hello в Netscape Navigator*

```
<BODY>
Welcome!
</BODY>
</HTML>
```

▼ ВНИМАНИЕ Создаваемые страницы HTML, как и любую другую страницу HTML, можно просматривать в различных программах-браузерах. Так, на рис. 19.2 использована программа Netscape Navigator версии 3.0. На рисунках, изображенных в этой главе, представлены различные браузеры.

# Web-пакет PL/SQL

Web-пакет PL/SQL (Oracle Web Toolkit) — это набор программных модулей, с помощью которых процедуры PL/SQL генерируют выходные данные формата HTML. Модули этого пакета описаны в таблице 19.3. Примеры работы многих модулей рассмотрены в последующих разделах, а более подробно о них, в том числе и о составляющих их подпрограммах, рассказано в руководстве PL/SQL Web Toolkit Reference, которое устанавливается при инсталляции программы WebServer.

**ТАБЛИЦА 19.3. Модули программного Web-пакета PL/SQL**

| Модуль | Описание |
|---|---|
| HTP | Гипертекстовые процедуры. В этом модуле содержатся процедуры, генерирующие многие из тегов HTML |
| HTF | Гипертекстовые функции. Этот модуль реализует те же функциональные средства, что и HTP, но не в качестве подпрограмм, а как функции |
| OWA* | В этом модуле содержатся внутренние процедуры, которые вызываются самим агентом PL/SQL. Они не предназначены для внешнего использования |
| OWA_UTIL | Служебные функции для работы с временной информацией и динамическим SQL, а также для получения значений переменных среды CGI |
| OWA_OPT_LOCK* | Предназначен для установления блокировок, предотвращающих потерю изменений |
| OWA_PATTERN* | Функции, реализующие сопоставление регулярных выражений с образцами |
| OWA_TEXT* | Процедуры, функции и типы данных, позволяющие манипулировать большими текстовыми строками. OWA_TEXT используется при функционировании OWA_PATTERN |
| OWA_IMAGE | Утилиты для работы с серверными картами образов |
| OWA_COOKIE | Утилиты для работы с домашними данными HTML |

* Эти модули не рассматриваются в последующих разделах; информация о них и примеры их использования приведены во встроенной документации.

▼ ВНИМАНИЕ Назначением модулей Web-пакета является создание кода HTML. Далее рассматривается этот пакет и генерируемый им код HTML, но не рассказывается о назначении самого кода HTML. Более подробная информация о HTML приведена в документации по Oracle WebServer.

## HTP и HTF

Эти модули формируют основу выходных данных HTML. Модуль HTP состоит только из процедур. С каждым вызовом процедуры HTP создается один тег (признак) HTML. Когда процедура вызывается через агент PL/SQL, этот тег возвращается как элемент генерируемой страницы. Модуль же HTF состоит лишь из функций. Эти функции идентичны процедурам HTP, за исключением того что они возвращают строку символов с тегом. При необходимости такую строку можно затем отобразить с помощью процедуры HTP.PRINT.

Процедуры HTP и функции HTF можно разбить на категории, перечисленные в таблице 19.4. Далее будут приведены примеры работы с подпрограммами каждой из категорий. Полную информацию о синтаксисе и параметрах каждой из этих процедур и функций можно найти во встроенной документации по пакету Oracle Web Toolkit.

## ТАБЛИЦА 19.4. Подпрограммы модулей HTP и HTF

| Категория | Подпрограммы |
|---|---|
| Печать* | p, print, prn, prints, ps |
| Константы | bodyOpen, bodyClose, htmlOpen, htmlClose, headOpen, headClose |
| Заголовок | base, isindex, linkrel, linkrev, meta, title |
| Тело | address, anchor, anchor2, area, base, basefont, bgsound, big, blockquoteOpen, blockquoteClose, br, center, centerOpen, centerClose, comment, dfn, div, fontOpen, fontClose, header, hr, img, img2, line, listingOpen, listingClose, mailto, mapOpen, mapClose, nl, nobr, para, paragraph, plaintext, preOpen, preClose, s, small, strike, sub, sup, wbr |
| Списки | dirlistClose, dirlistOpen, dlistClose, dlistDef, dlistOpen, dlistTerm, listHeader, listItem, menulistClose, menulistOpen, olistClose, olistOpen, ulistClose, ulistOpen |
| Форматирование символов | cite, code, emphasis, em, keyboard, kbd, sample, strong, variable |
| Физическое форматирование | bold, italics, teletype |
| Формы | formOpen, formClose, formCheckbox, formHidden, formImage, formPassword, formRadio, formReset, formSubmit, formText, formSelectOpen, formSelectOption, formSelectClose, formTextarea, formTextarea2, formTextareaOpen, formTextareaOpen2, formTextareaClose |
| Таблицы | tableOpen, tableClose, tableCaption, tableRowOpen, tableRowClose, tableHeader, tableData |

* Подпрограммы печати являются лишь процедурами и содержатся только в модуле HTP. Все другие подпрограммы могут быть как процедурами (HTP), так и функциями (HTF).

## Печать

Процедуры печати находятся только в модуле HTP, в HTF их нет. Основной процедурой является PRINT, которая используется для отображения строк кода HTML. Ее аргументы возвращаются клиенту через прослушивающий процесс как фрагмент страницы. Эта процедура переопределяется типами данных:

```
PROCEDURE HTP.PRINT(cbuf IN VARCHAR2);
PROCEDURE HTP.PRINT(dbuf IN DATE);
PROCEDURE HTP.PRINT(nbuf IN NUMBER);
```

В *cbuf*, *dbuf* и *nbuf* содержится выводимое значение. Если процедуре передается дата или число, то это значение преобразуется в строку символов в соответствии с форматом, заданным по умолчанию. После передачи строки символов процедура PRINT выводит символ новой строки. Другие процедуры этой категории (P, PRN, PRINTS, PS) функционируют аналогично. PRN не включает в выводимую информацию символ новой строки, а PRINTS дает возможность замещать специальные символы. Использование PRINT демонстрируется на примере процедуры **PrintDemo**:

```
-- Этот пример содержится в файле prntdemo.sql.
CREATE OR REPLACE PROCEDURE PrintDemo AS
BEGIN
  -- В этой процедуре PRINT используется для вывода всего кода HTML.
  -- В модулях HTP и HTF содержатся служебные процедуры, которые
  -- обладают практически теми же функциональными возможностями.
  HTP.print('<HTML>');
  HTP.print('This is a demonstration of the ');
  HTP.print('<STRONG>HTP.print</STRONG> procedure. It can be ');
  HTP.print('used to output additional tags that are not in ');
```

```
    HTP.print('HTP and HTF, such as the ');
    HTP.print('<BLINK>BLINK</BLINK> tag.');
    OWA_UTIL.signature('PrintDemo'); HTP.print('</HTML>');
END PrintDemo;
```

Результат выполнения PrintDemo выглядит следующим образом:

```
<HTML>
This is a demonstration of the
<STRONG>HTP.print</STRONG> procedure. It can be
used to output additional tags that are not in
HTP and HTF, such as the
<BLINK>BLINK</BLINK> tag.
</HTML>
```

▼ ВНИМАНИЕ В **PrintDemo** вызывается процедура **OWA_UTIL.signature**, которая печатает внизу страницы строку, где указаны текущая дата и связь с представлением исходного текста PL/SQL. Более подробно об этом рассказано ниже, в разделе "OWA_UTIL".

## Константы

В модуле HTF имеется шесть констант, которые описаны в таблице 19.5. Эти константы отображаются также с помощью эквивалентных процедур HTP.

**ТАБЛИЦА 19.5.**

| Константа | Результат HTML | Описание |
|---|---|---|
| htmlOpen | <HTML> | Начинает страницу HTML. Должна быть первым тегом на каждой странице (исключая комментарии) |
| htmlClose | </HTML> | Заканчивает страницу HTML. Должна быть последним тегом на каждой странице |
| headOpen | <HEAD> | Открывает заголовок страницы. Заголовок необязателен, однако рекомендуется указывать эту константу на страницах HTML |
| headClose | </HEAD> | Закрывает заголовок |
| bodyOpen | <BODY> | Открывает тело |
| bodyClose | </BODY> | Закрывает тело |

## Заголовок

Подпрограммы этой категории создают теги, которые действуют в заголовке страницы HTML и, следовательно, должны указываться между вызовами HTP.headOpen и HTP.headClose. В этих тегах содержится информация о названии страницы. Использование некоторых тегов заголовков демонстрируется в следующем разделе на примере процедуры **BodyDemo**.

## Тело

Заголовки тела используются в основном фрагменте страницы HTML. Рекомендуется ограничивать тело тегами <BODY> и </BODY>, которые генерируются подпрограммами **HTP.bodyOpen** и **HTP.bodyClose**. С помощью подпрограмм этой категории, которые могут быть как процедурами, так и функциями, вводятся образы, точки привязки и листинги. Использование ряда тегов тела демонстрируется на примере процедуры **BodyDemo**:

```
-- Этот пример содержится в файле bodydemo.sql.
CREATE OR REPLACE PROCEDURE BodyDemo AS
BEGIN
  HTP.htmlOpen;
  HTP.headOpen;
  HTP.title('Body tags demo');
```

```
    HTP.headClose;

    HTP.bodyOpen;
    HTP.print('This page demonstrates several of the tags ');
    HTP.print('available for formatting the text in the body ');
    HTP.print('of a web page, and for inserting additional ');
    HTP.print('tags.' || HTF.para);

    HTP.header(2, 'Links');
    HTP.print('Here are some links to check out. These are ');
    HTP.print('generated using the HTP.anchor procedure.');
    HTP.para;
    HTP.anchor('//www.oracle.com/', 'Oracle Corporation');
    HTP.para;
    HTP.anchor('//www.osborne.com/oracle/index.htm', 'Oracle Press');
    HTP.para;
    HTP.anchor('//www.platinum.com/', 'Platinum Corporation');
    HTP.para;
    HTP.header(2, 'Formats');
    HTP.print(HTF.centerOpen || 'These lines are centered, using ');
    HTP.br;
    HTP.print('the HTF.centerOpen and HTF.centerClose functions.');
    HTP.centerClose;

    HTP.strike('Here''s some strikethrough text.');
    OWA_UTIL.signature('BodyDemo');
    HTP.bodyClose;
    HTP.htmlClose;
END BodyDemo;
```

Код HTML, создаваемый процедурой **BodyDemo**, приведен ниже, а сама страница (в Netscape Navigator 4.0) показана на рис. 19.3.

```
<HTML>
<HEAD>
```

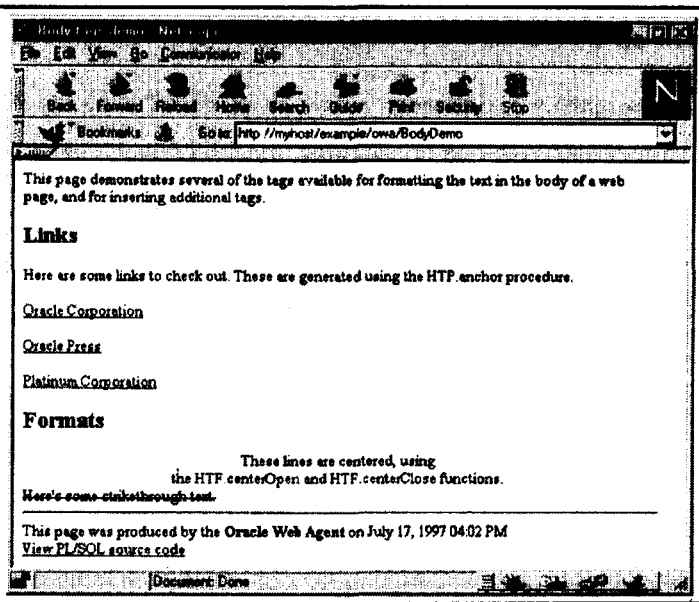



**Рис. 19.3.**
*Результат выполнения BodyDemo в Netscape Navigator*

```
<TITLE>Body tags demo</TITLE>
</HEAD>
<BODY>
This page demonstrates several of the tags
available for formatting the text in the body
of a web page, and for inserting additional
tags.<P>
<H2>Links</H2>
Here are some links to check out. These are
generated using the HTP.anchor procedure.
<P>
<A HREF="//www.oracle.com/">Oracle Corporation</A>
<P>
<A HREF="//www.osborne.com/oracle/index.htm">Oracle Press</A>
<P>
<A HREF="//www.platinum.com/">Platinum Corporation</A>
<P>
<H2>Formats</H2>
<CENTER>These lines are centered, using
<BR>
the HTF.centerOpen and HTF.centerClose functions.
</CENTER>
<STRIKE>Here's some strikethrough text.</STRIKE>
</BODY>
</HTML>
```

## Списки

В HTML существует несколько различных видов списков. Каждый список состоит из группы элементов, обычно по одному на строку, причем перед каждым элементом указывается тег форматирования (например, жирная точка или некоторое число). Применяемые списки перечислены ниже:

- *Упорядоченный список* (ordered list) с пронумерованными элементами
- *Неупорядоченный список* (unordered list) с элементами, предваряемыми жирными точками
- *Список меню* (menu list) (аналогичен неупорядоченному списку, но обычно более компактен)
- *Список определений* (definition list), в котором за каждым элементом следует его определение
- *Список файлов каталога* (directory list), организованный обычно в столбцы шириной 24 символа. Ширина каждого элемента списка не должна превышать 20 символов.

Список, как правило, начинается с тега открытия списка, за которым следуют элементы списка (разграниченные тегами элементов списка), и заканчивается тегом закрытия списка. Различные виды списков показаны на примере модуля **ListDemo**, приведенном ниже. На рис. 19.4 изображен упорядоченный список, полученный в Microsoft Internet Explorer 3.0.

```
-- Этот пример содержится в файле listdemo.sql.
CREATE OR REPLACE PACKAGE ListDemo AS
  PROCEDURE Go;
  PROCEDURE ShowList(p_ListType IN CHAR);
END ListDemo;

CREATE OR REPLACE PACKAGE BODY ListDemo AS
  -- Установим c_OWAPath для виртуального маршрута (включая DCD),
  -- указывающего место, где инсталлирован Oracle Web Agent.
  c_OWAPath CONSTANT VARCHAR2(50) := '/example/owa/';

  -- Представляет информацию о пользователе посредством различных списков.
  PROCEDURE ShowChoices IS
```

```
    c_ListPath VARCHAR2(100) :=
      c_OWAPATH || 'ListDemo.ShowList?p_ListType=';
BEGIN
    HTP.line;
    HTP.p('Click on any of the following links to see the');
    HTP.p('students in the specified list type.' || HTF.para);
    HTP.anchor(c_ListPath || 'U', 'Unordered');
    HTP.anchor(c_ListPath || 'O', 'Ordered');
    HTP.anchor(c_ListPath || 'M', 'Menu');
    HTP.anchor(c_ListPath || 'D', 'Directory');
END ShowChoices;

-- Представляет имена и фамилии студентов в списке указанного типа.
PROCEDURE ShowList(p_ListType IN CHAR) IS
    v_Title VARCHAR2(20);
    v_ListOpen VARCHAR2(10);
    v_ListClose VARCHAR2(10);
    CURSOR c_Names IS
      SELECT first_name || ' ' || last_name name
        FROM students
        ORDER BY last_name;
BEGIN
    IF p_ListType = 'U' THEN
      v_Title := 'Unordered List';
      v_ListOpen := HTF.ulistOpen;
      v_ListClose := HTF.ulistClose;
    ELSIF p_ListType = 'O' THEN
      v_Title := 'Ordered List';
      v_ListOpen := HTF.olistOpen;
      v_ListClose := HTF.olistClose;
    ELSIF p_ListType = 'M' THEN
      v_Title := 'Menu List';
      v_ListOpen := HTF.menulistOpen;
      v_ListClose := HTF.menulistClose;
    ELSIF p_ListType = 'D' THEN
      v_Title := 'Directory List';
      v_ListOpen := HTF.dirlistOpen;
      v_ListClose := HTF.dirlistClose;
    END IF;
    HTP.htmlOpen;
    HTP.headOpen;
    HTP.title(v_Title);
    HTP.headClose;

    HTP.bodyOpen;
    -- Отобразим список.
    HTP.p(v_ListOpen);
    FOR v_NamesRec IN c_Names LOOP
      HTP.listItem(v_NamesRec.name);
    END LOOP;
    HTP.p(v_ListClose);

    -- Отобразим варианты выбора.
    ShowChoices;
```

```
        OWA_UTIL.signature('ListDemo.ShowList');
        HTP.bodyClose;
    END  ShowList;

    PROCEDURE  Go  IS
    BEGIN
        HTP.htmlOpen;
        HTP.headOpen;
        HTP.title('List  Demo');
        HTP.headClose;

        HTP.bodyOpen;
        HTP.header(2,  'Welcome  to  the  list  demo');
        ShowChoices;
        OWA_UTIL.signature('ListDemo.Go');
        HTP.bodyClose;
        HTP.htmlClose;
    END  Go;
END  ListDemo;
```

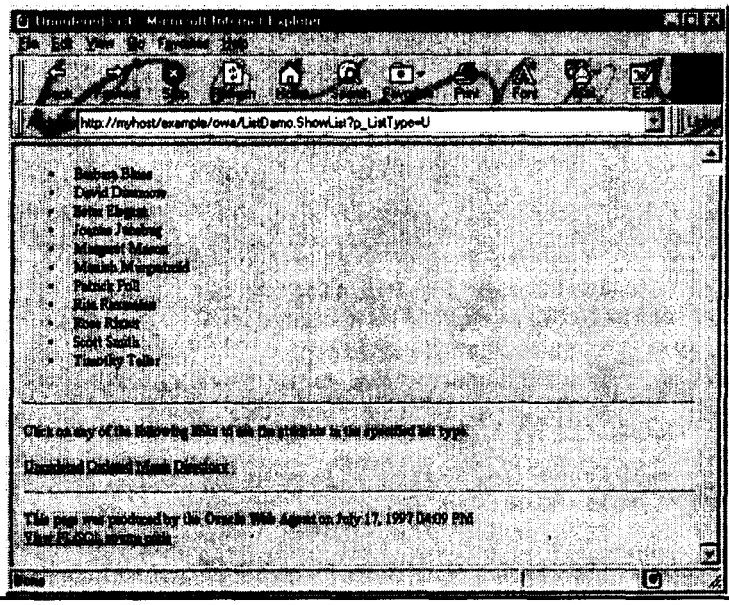



**Рис. 19.4.**
*Результат выполнения ListDemo в Internet Explorer*

## Форматирование символов

Теги форматирования символов указывают, как форматировать символы, однако не дают точных характеристик формата. Каждая программа-браузер может передавать символы в том стиле, который принят. Например, тег STRONG обычно воспроизводит текст, набранный жирным шрифтом, однако в некоторых браузерах этот тег может передавать мерцающий текст. Специфические теги форматирования описаны в следующем разделе, "Физическое форматирование". В приведенном ниже примере представлены некоторые теги форматирования символов. Результат выполнения процедуры **CharDemo** в NCSA Mosaic 3.0 показан на рис.19.5.

```
-- Этот пример содержится в файле listdemo.sql.
CREATE  OR  REPLACE  PROCEDURE  CharDemo  AS
BEGIN
  HTP.htmlOpen;
  HTP.headOpen;
```

```
    HTP.title('Character Formatting Tags');
    HTP.headClose;

    HTP.bodyOpen;
    HTP.header(2, 'This page demonstrates character formatting
               tags.');
    HTP.para;
    HTP.cite('This text is a citation, usually rendered as
             italics.');
    HTP.para;
    HTP.code('This text is code, usually rendered as a monospace
             font.');
    HTP.para;
    HTP.strong('This text is strong, usually rendered as bold.');
    HTP.para;

    OWA_UTIL.signature('CharDemo');
    HTP.bodyClose;
    HTP.htmlClose;
END CharDemo;
```

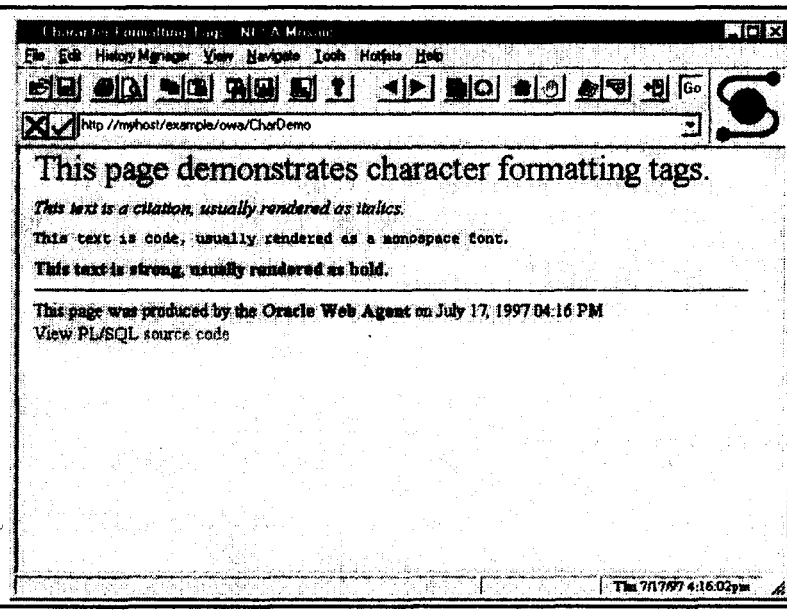



**Рис. 19.5.**
*Результат выполнения CharDemo в NCSA Mosaic*

## Физическое форматирование

Теги физического форматирования указывают физические атрибуты текста. К этой категории относятся три подпрограммы: BOLD, ITALIC и TELETYPE. BOLD указывает на то, что аргументы процедуры должны быть выделены жирным шрифтом, ITALIC – курсивом, а TELETYPE сообщает об использовании шрифта типа пишущей машинки, например шрифта Courier.

## Формы

Теги форм применяются для создания *форм HTML*. Форма разрешает ввод информации пользователем в виде элементов GUI (графического пользовательского интерфейса), например в виде полей ввода текста или в виде переключателей. Входная информация передается серверу методом POST. Web-агент автоматически определяет используемый протокол и передает вводимые данные процедуре PL/SQL. В качестве примера рассмотрим модуль **FormDemo**:

```
-- Этот пример содержится в файле formdemo.sql.
CREATE OR REPLACE PACKAGE FormDemo AS
  PROCEDURE Go;
  PROCEDURE Process(p_Checkbox IN VARCHAR2 DEFAULT 'off',
                    p_Password IN VARCHAR2 DEFAULT NULL,
                    p_Radio IN VARCHAR2 DEFAULT NULL);
END FormDemo;

CREATE OR REPLACE PACKAGE BODY FormDemo AS
  -- Установим c_OWAPath для виртуального маршрута (включая DCD),
  -- указывающего на место, где инсталлирован Oracle Web Agent.
  c_OWAPath CONSTANT VARCHAR2(50) := '/example/owa/';

  PROCEDURE ShowForm IS
  BEGIN
     HTP.line;
     -- Сначала откроем форму, указав в качестве адресата URL
     -- процедуру Process.
     HTP.formOpen(curl => c_OWAPath || 'FormDemo.Process');

     -- Теперь отобразим текст, а затем средства ввода.
     HTP.p('Welcome to a HTML form. Fill out some information ');
     HTP.p('below, and press the ''Submit'' button to process it.');
     HTP.p('Press ''Reset'' to clear your entries.');
     HTP.para;

     HTP.p('Here is a checkbox: ');
     HTP.formCheckbox(cname => 'p_CheckBox');
     HTP.para;
     HTP.p('Here is a password entry: ');
     HTP.formPassword(cname => 'p_Password',
                      csize => 10);
     HTP.para;
     HTP.p('Select one of the following radio buttons:');
     HTP.nl;
     HTP.formRadio(cname => 'p_Radio', cvalue => 'One');
     HTP.p('One');
     HTP.formRadio(cname => 'p_Radio', cvalue => 'Two');
     HTP.p('Two');
     HTP.formRadio(cname => 'p_Radio', cvalue => 'Three');
     HTP.p('Three' || HTF.nl);
     HTP.formRadio(cname => 'p_Radio', cvalue => 'Four');
     HTP.p('Four');
     HTP.formRadio(cname => 'p_Radio', cvalue => 'Five');
     HTP.p('Five' || HTF.para);
     HTP.formSubmit;
     HTP.formReset;
     HTP.formClose;
  END ShowForm;

  PROCEDURE Process(p_Checkbox IN VARCHAR2 DEFAULT 'off',
                    p_Password IN VARCHAR2 DEFAULT NULL,
                    p_Radio IN VARCHAR2 DEFAULT NULL) IS
  BEGIN
```

```
        HTP.htmlOpen;
        HTP.headOpen;
        HTP.title('Form Results');
        HTP.headClose;

        HTP.bodyOpen;
        HTP.header(2, 'Form Results:');
        HTP.p('p_Checkbox = ' || p_Checkbox || HTF.nl);
        HTP.p('p_Password = ' || p_Password || HTF.nl);
        HTP.p('p_Radio = ' || p_Radio || HTF.para);
        ShowForm;
        OWA_UTIL.signature('FormDemo.Process');
        HTP.bodyClose;
        HTP.htmlClose;
    END Process;

    PROCEDURE Go IS
    BEGIN
        HTP.htmlOpen;
        HTP.headOpen;
        HTP.title('Forms Demo');
        HTP.headClose;
        HTP.bodyOpen;
        ShowForm;
        OWA_UTIL.signature('FormDemo.Go');
        HTP.bodyClose;
        HTP.htmlClose;
    END Go;
END FormDemo;
```

Основной для установки формы является процедура HTP.FORMOPEN. Ее синтаксис приведен ниже, а параметры описаны в таблице 19.6. На рис. 19.6 показан результат выполнения **FormDemo.Go** в Internet Explorer.

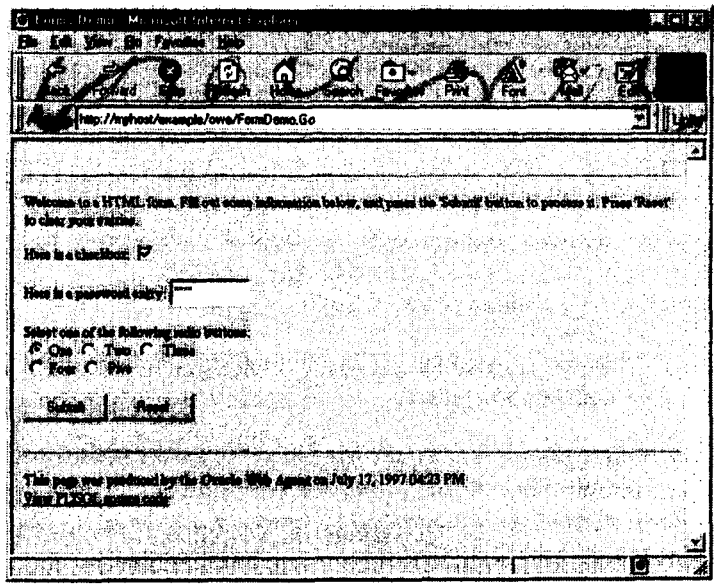



**Рис. 19.6.**
*Результат выполнения FormDemo.Go в Internet Explorer*

```
PROCEDURE FORMOPEN(curl IN VARCHAR2,
                   cmethod IN VARCHAR2 DEFAULT 'POST',
                   ctarget IN VARCHAR2 DEFAULT NULL,
                   cenctype IN VARCHAR2 DEFAULT NULL,
                   cattributes IN VARCHAR2 DEFAULT NULL);
```

**ТАБЛИЦА 19.6. Параметры процедуры FORMOPEN**

| Параметр | Тип данных | Описание |
|---|---|---|
| *curl* | VARCHAR2 | Указатель URL, который будет обрабатывать форму. Это обычно еще одна хранимая процедура PL/SQL, вызываемая с помощью Web-агента |
| *cmethod* | VARCHAR2 | Используемый метод HTTP: или 'GET', или 'POST' |
| *ctarget* | VARCHAR2 | Атрибут TARGET (адресат) - если он установлен, то curl будет открыт в новом окне |
| *cenctype* | VARCHAR2 | Тип кодирования, применяемый при посылке данных. Если не указан, то протокол будет определяться, как правило, параметром *cmethod* |
| *cattributes* | VARCHAR2 | Все необходимые дополнительные атрибуты |

FORMOPEN генерирует код HTML следующего вида:

```
<FORM ACTION="curl"METHOD="cmethod"TARGET="ctarget"
   ENCTYPE="cenctype"cattributes>
```

Параметры процедуры, указанной в curl, должны соответствовать именам элементов формы. В примере, рассмотренном выше, процедура **FormDemo.Process** имеет три параметра: **p_Checkbox, p_Password** и **p_Radio.** Именно такие имена имеют элементы, используемые в процедуре **FormDemo.ShowForm.**

Можно использовать как метод GET, так и метод POST. Когда применяется метод GET, браузер клиента компонует URL таким образом, чтобы процедура, указанная параметром curl, вызывалась так, как если бы она вызывалась напрямую. В состав URL включаются все параметры. В некоторых системах число возможных параметров ограничено размером URL, поэтому по умолчанию применяется метод POST. При использовании метода POST параметры посылаются браузером после посылки URL. Web-агент считывает их и вызывает соответствующую процедуру.

▼ СОВЕТУЕМ Для всех параметров рабочей процедуры рекомендуется задавать значения по умолчанию. Если пользователь не вводит значение для элемента формы, то соответствующий параметр не передается. При отсутствии значения по умолчанию происходит ошибка, и процедура выполняется неуспешно.

## Таблицы

Подпрограммы форматирования таблиц используются для создания таблиц HTML. Таблицы состоят из строк и столбцов, в которых может содержаться информация HTML. Применение некоторых из процедур форматирования таблиц демонстрируется на примере процедуры **TableDemo,** приведенном ниже. Результат выполнения этой процедуры в Netscape 4.0 показан на рис. 19.7.

```
-- Этот пример содержится в файле tabledemo.sql.
CREATE OR REPLACE PROCEDURE TableDemo AS
  CURSOR c_Students IS
      SELECT *
        FROM students
        ORDER BY ID;
BEGIN
  HTP.htmlOpen;
  HTP.headOpen;
  HTP.title('Table Demo');
  HTP.headClose;
```

```
    HTP.bodyOpen;
    HTP.tableOpen(cborder => 'BORDER=1');

    -- Последовательно просмотрим информацию о всех студентах и для
    -- каждого из них выведем по две строки таблицы следующего вида:
    -- +--------------------------------+
    -- |     | first_name last_name     |
    -- | ID  +-------+-----------------+
    -- |     | major | current_credits |
    -- +-----+-------+-----------------+
    FOR v_StudentRec in c_Students LOOP
        HTP.tableRowOpen;
        HTP.tableData(crowspan => 2,
                      cvalue => HTF.bold(v_StudentRec.ID));
        HTP.tableData(ccolspan => 2,
                      calign => 'CENTER',
                      cvalue => v_StudentRec.first_name || ' ' ||
                                v_StudentRec.last_name);
        HTP.tableRowClose;
        HTP.tableRowOpen;
        HTP.tableData(cvalue => 'Major: ' || v_StudentRec.major);
        HTP.tableData(cvalue => 'Credits: ' ||
                                v_StudentRec.current_credits);
        HTP.tableRowClose;
    END LOOP;

    HTP.tableClose;
    OWA_UTIL.signature('TableDemo');
    HTP.bodyClose;
    HTP.htmlClose;
END TableDemo;
```

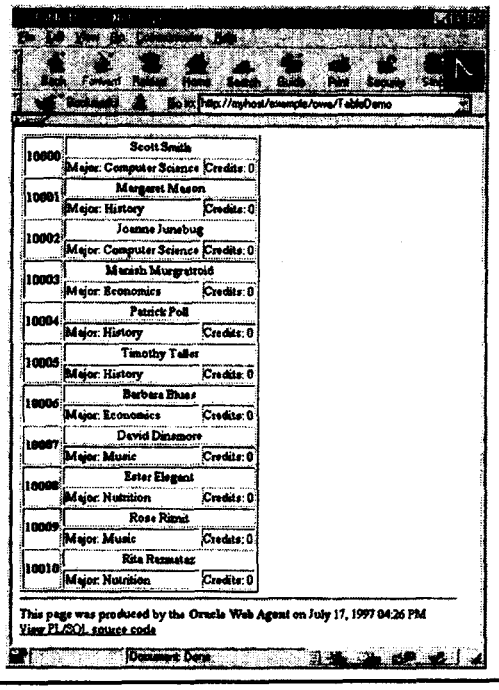



**Рис. 19.7.**
*Результат выполнения TableDemo в Netscape 4.0*

# OWA_UTIL

В модуле OWA_UTIL, который создан с использованием модулей HTP и HTF, содержатся дополнительные функциональные средства. Ниже описаны три различных набора подпрограмм модуля OWA_UTIL:

- Утилиты HTML
- Утилиты динамического SQL
- Временные утилиты

▼ СОВЕТУЕМ Для работы с OWA_UTIL необходимо установить значение FALSE для параметра Protect_OWA_Pkg в конфигурации OWA для Web Request Broker. Более подробно об этом рассказано во встроенной документации.

## Утилиты HTML

Подпрограммы этого набора перечислены в таблице 19.7. Их функции весьма разнообразны, начиная с вывода подписей и заканчивая отображением таблиц Oracle в виде таблиц HTML. Использование некоторых из этих подпрограмм продемонстрировано на примере процедуры **UtilDemo**. Более подробная информация о каждой из них приведена во встроенной документации Web-сервера.

```
-- Этот пример содержится в файле utildemo.sql.
CREATE OR REPLACE PROCEDURE UtilDemo AS
  v_Temp BOOLEAN;
BEGIN
  HTP.htmlOpen;
  HTP.headOpen;
  HTP.title('OWA_UTIL HTML Demos');
  HTP.headClose;

  HTP.bodyOpen;
  HTP.p('This page demonstrates several of the HTML utilities ');
  HTP.p('in the OWA_UTIL package.' || HTF.para);

  HTP.header(2, 'print_cgi_env');
  HTP.p('The following is generated by ');
  HTP.bold('OWA_UTIL.print_cgi_env');
  HTP.p(':' || HTF.nl);
  OWA_UTIL.print_cgi_env;

  HTP.header(2, 'get_owa_service_path');
  HTP.p('The following is generated by ');
  HTP.bold('OWA_UTIL.get_owa_service_path');
  HTP.p(':' || HTF.nl);
  HTP.p(OWA_UTIL.get_owa_service_path);

  HTP.header(2, 'tableprint');
  HTP.p('The following table (containing information from ');
  HTP.p('registered_students) is generated by ');
  HTP.bold('OWA_UTIL.tableprint');
  HTP.p(':' || HTF.nl);
  v_Temp := OWA_UTIL.tableprint(
              ctable => 'registered_students',
              cattributes => 'BORDER=1',
              ntable_type => OWA_UTIL.html_table,
              cclauses => 'ORDER BY student_id');

  HTP.p('Here is the same data, as a preformatted table instead:');
```

```
    HTP.nl;
    v_Temp  :=  OWA_UTIL.tableprint(
                    ctable  =>  'registered_students',
                    cattributes  =>  'BORDER=1',
                    ntable_type  =>  OWA_UTIL.pre_table,
                    cclauses  =>  'ORDER  BY  student_id');
    OWA_UTIL.signature('UtilDemo');
    HTP.bodyClose;
END  UtilDemo;
```

## ТАБЛИЦА 19.7. Утилиты OWA_UTIL

| Подпрограмма | Описание |
|---|---|
| signature | Выдает горизонтальную линию, за которой следует строка примерно такого содержания: "This page was produced by **Oracle PL/SQL Agent** on August 9, 1995 09:30" (эта страница была создана Oracle PL/SQL Agent 9 августа 1995 года в 09:30). Кроме того, здесь можно указывать ссылку на исходный текст PL/SQL |
| showsource | Выдает исходный текст модуля или процедуры. Если указана модульная процедура, то выдает текст всего модуля |
| showpage | Выдает (с помощью DBMS_OUTPUT) последнюю страницу, созданную с использованием Web-пакета. Это удобно для тестирования результатов выполнения процедуры в SQL*Plus или SQL-Station. Более подробно об этом рассказано в разделе "Среды разработки процедур OWA" в конце данной главы |
| get_cgi_env | Возвращает значение указанной переменной среды CGI. Если данная переменная не установлена — возвращается NULL |
| print_cgi_env | Выдает значения всех переменных среды CGI. Эта функция полезна при отладке программ |
| mime_header | Используется для изменения заданного по умолчанию заголовка MIME, который возвращается агентом. Эту процедуру необходимо вызывать до обращения к HTP.PRINT или HTP.PRN; в противном случае Web-агент возвращает заголовок по умолчанию |
| redirect_url | Указывает, что WebServer просматривает другой URL. Поскольку генерируемый код является частью заголовка, эту процедуру необходимо вызывать до вызова HTP.PRINT или HTP.PRN |
| status_line | Посылает клиенту стандартный код состояния HTTP. Поскольку генерируемый код является частью заголовка, эту процедуру необходимо вызывать до вызова HTP.PRINT или HTP.PRN |
| header_close | Закрывает заголовок явным образом. Если процедуры MIME_HEADER, REDIRECT_URL или STATUS_LINE не вызывались со значением TRUE параметра *bclose_header*, то эту процедуру необходимо вызывать до HTP.PRINT или HTP.PRN |
| get_owa_service_path | Возвращает полный сервисный маршрут (в том числе DCD) для процедуры, исполняемой в данный момент |
| tableprint | Распечатывает содержимое таблицы базы данных в виде таблицы HTML или в виде текста стандартного формата |
| who_called_me | Возвращает информацию об обрабатываемом объекте PL/SQL, в том числе имя и тип этого объекта |

## Утилиты динамического SQL

С помощью этих подпрограмм можно генерировать таблицы и списки HTML из запросов SQL. Существуют три таких переопределяемых подпрограммы: BIND_VARIABLES, CELLSPRINT и LISTPRINT. Краткое описание каждой из них, а также пример их использования приведены ниже.

**BIND_VARIABLES** Эта функция готовит оператор запроса к выполнению, используя в качестве входных аргументов сам запрос и необязательные переменные привязки. При этом производится грамматический разбор оператора и привязка всех входных переменных с помощью процедуры DBMS_SQL.BIND_VARIABLES, которая возвращает номер курсора, используемый для хранения подготовленного оператора. Более подробно о модуле DBMS_SQL и о переменных привязки рассказано в главе 15.

▼ СОВЕТУЕМ Переменная BIND_VARIABLES в своей работе обращается к модулю DBMS_SQL, поэтому нельзя включать во входную строку символов точку с запятой.

**CELLSPRINT** Эта процедура форматирует результаты запроса, превращая их в таблицу HTML. Она переопределяется, допуская ввод запроса либо как строки символов, либо в виде курсора, подготовленного функцией BIND_VARIABLES. Кроме того, она позволяет просматривать результирующий набор запроса и возвращать только выбранные строки.

**LISTPRINT** LISTPRINT форматирует результаты запроса, превращая их во всплывающий список выбора HTML, или список указания (picklist), который разработан специально для форм. При использовании LISTPRINT в запросе должно выбираться три столбца:

- Столбец 1: значение, возвращаемое списком указания
- Столбец 2: строка символов, отображаемая в списке указания
- Столбец 3: этот пункт должен быть выбран заранее (если это не NULL-значение)

Использование этих трех утилит демонстрируется на примере модуля **DynamicDemo**. Для начала обработки демонстрационного примера выполните процедуру **DynamicDemo.Go**.

```
-- Этот пример содержится в файле dyndemo.sql.
CREATE OR REPLACE PACKAGE DynamicDemo AS
  PROCEDURE Go;
  PROCEDURE Process(p_Query IN VARCHAR2 DEFAULT NULL,
                    p_Type IN VARCHAR2);
  PROCEDURE ListProcess(p_Value IN VARCHAR2 DEFAULT NULL);
END DynamicDemo;

CREATE OR REPLACE PACKAGE BODY DynamicDemo AS
  -- Установим c_OWAPath для виртуального маршрута (включая DCD),
  -- указывающего на место, где инсталлирован Oracle Web Agent.
  c_OWAPath CONSTANT VARCHAR2(50) := '/example/owa/';

  PROCEDURE ShowForm IS
  BEGIN
      HTP.line;
      HTP.p('Enter a query in the box below, and select the output ');
      HTP.p('type, then click ''Submit''. If you choose the picklist, ');
      HTP.p('then your query should be of the following form: ');
      HTP.nl;
      HTP.p('Column 1: Result returned for this item' || HTF.nl);
      HTP.p('Column 2: String displayed for this item' || HTF.nl);
      HTP.p('Column 3: NULL or non-NULL. If non-NULL, the current ');
      HTP.p('field will be flagged as selected.'); HTP.para;
      -- Сначала откроем форму, указав в качестве адресата URL
      -- процедуру Process.
      HTP.formOpen(curl => c_OWAPath || 'DynamicDemo.Process');
      HTP.formTextArea(cname => 'p_Query',
                       nrows => 5,
                       ncolumns => 40);
      HTP.nl;
      HTP.p('Output type: ');
      HTP.p('HTML Table');
      HTP.formRadio(cname => 'p_Type',
                    cvalue => 'table',
                    cchecked => 'CHECKED');
      HTP.p('Picklist');
```

```
    HTP.formRadio(cname => 'p_Type',
                  cvalue => 'list');
    HTP.formSubmit;
END ShowForm;

/* Основная процедура ввода. */
PROCEDURE Go IS
BEGIN
    HTP.htmlOpen;
    HTP.headOpen;
    HTP.title('OWA_UTIL Dynamic SQL Utilities Demo');
    HTP.headClose;

    HTP.bodyOpen;
    ShowForm;
    OWA_UTIL.signature('DynamicDemo.Go');
    HTP.bodyClose;
    HTP.htmlClose;
END Go;

/* Обрабатывает основную форму. */
PROCEDURE Process(p_Query IN VARCHAR2 DEFAULT NULL,
                  p_Type IN VARCHAR2) IS
    v_CursorID INTEGER;
BEGIN
    HTP.htmlOpen;
    HTP.headOpen;
    HTP.title('Query Results');
    HTP.headClose;

    HTP.bodyOpen;
    v_CursorID := OWA_UTIL.bind_variables(p_Query);
    HTP.p('Output from query ');
    HTP.bold(p_Query);
    HTP.p(':' || HTF.para);
    IF p_Type = 'table' THEN
       HTP.tableOpen(cborder => 'BORDER=1');
       OWA_UTIL.cellsprint(v_CursorID);
       HTP.tableClose;
    ELSE
       HTP.formOpen(curl => c_OWAPath || 'DynamicDemo.ListProcess');
       OWA_UTIL.listprint(v_CursorID, 'p_Value', 10);
       HTP.formSubmit;
       HTP.formClose;
    END IF;
    ShowForm;
    OWA_UTIL.signature('DynamicDemo.Process');
       HTP.bodyClose;
       HTP.htmlClose;
END Process;

/* Обрабатывает форму, содержащую список указания. */
PROCEDURE ListProcess(p_Value IN VARCHAR2 DEFAULT NULL) IS
BEGIN
```

```
        HTP.htmlOpen;
        HTP.headOpen;
        HTP.title('List Results');
        HTP.headClose;

        HTP.bodyOpen;
        HTP.p('You picked ' || p_Value || '.');
        ShowForm;
        OWA_UTIL.signature('DynamicDemo.ListProcess');
        HTP.bodyClose;
        HTP.htmlClose;
    END ListProcess;
END DynamicDemo;
```

## Временные утилиты

Для упрощения работы на Web-страницах со значениями, представляющими собой даты, в модуле OWA_UTIL предусмотрены две временные утилиты. Процедура CALENDARPRINT отображает в виде таблицы HTML календарь, созданный на основе результатов выполнения запроса, а процедура CHOOSE_DATE генерирует код, позволяющий вводить даты в виде "день, месяц и год".

**CALENDARPRINT** Входными данными для CALENDARPRINT является строка символов, содержащая запрос, или курсор, подготовленный OWA_UTIL.BIND_VARIABLES, а выходными — календарь, созданный на основе результатов выполнения запроса. В запросе можно указывать два или три столбца:

- Столбец 1: значение DATE, используемое для установления связи между возвращаемыми данными и отображаемым календарем.
- Столбец 2: текст, отображаемый для этой даты.
- Столбец 3: если указан этот столбец, то второй столбец будет отображаться как связь, адресат которой указан столбцом 3. Если используются всего два столбца, то значения будут отображаться только в виде текста.

**CHOOSE_DATE** CHOOSE_DATE отображает три поля, которые используются для ввода года, месяца и дня. Адресатом этих полей должна быть процедура с параметром типа OWA_UTIL.DATETYPE. Тип данных DATETYPE можно преобразовать к стандартному типу Oracle DATE с помощью подпрограммы OWA_UTIL.TO_DATE.

Использование этих утилит продемонстрировано на примере модуля **DateDemo**, а результат выполнения **DateDemo.Go** в Internet Explorer показан на рисунке 19.8. Для того чтобы создать этот модуль, предварительно нужно создать таблицу **calendar**:

```
-- Этот пример является частью файла datedemo.sql.
CREATE TABLE calendar (
  Today         DATE,
  Description   VARCHAR2(25),
  Link          VARCHAR2(20)
  );

INSERT INTO calendar (today, description, link)
  VALUES (SYSDATE - 5, 'Hello!', '//www.oracle.com');
INSERT INTO calendar (today, description, link)
  VALUES (SYSDATE - 4, 'Getting there...', '//www.oracle.com');
INSERT INTO calendar (today, description, link)
  VALUES (SYSDATE - 3, 'How''s it going?', '//www.oracle.com');
INSERT INTO calendar (today, description, link)
  VALUES (SYSDATE - 2, 'Soon...', '//www.oracle.com');
INSERT INTO calendar (today, description, link)
  VALUES (SYSDATE - 1, 'Tomorrow, tomorrow', '//www.oracle.com');
INSERT INTO calendar (today, description, link)
```

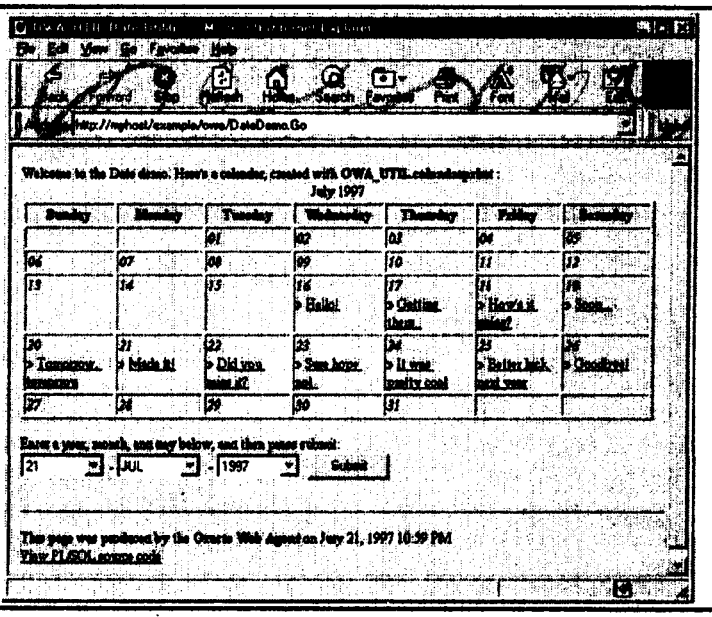



**Рис. 19.8.**
*Результат выполнения DateDemo.Go в Internet Explorer*

```
    VALUES (SYSDATE, 'Made it!', '//www.oracle.com');
INSERT INTO calendar (today, description, link)
    VALUES (SYSDATE + 1, 'Did you miss it?', '//www.oracle.com');
INSERT INTO calendar (today, description, link)
    VALUES (SYSDATE + 2, 'Sure hope not...', '//www.oracle.com');
INSERT INTO calendar (today, description, link)
    VALUES (SYSDATE + 3, 'It was pretty cool', '//www.oracle.com');
INSERT INTO calendar (today, description, link)
    VALUES (SYSDATE + 4, 'Better luck next year',
            '//www.oracle.com');
INSERT INTO calendar (today, description, link)
    VALUES (SYSDATE + 5, 'Goodbye!', '//www.oracle.com');
```

**Ниже приведено описание собственно модуля:**

```
-- Этот пример является частью файла datedemo.sql.
CREATE OR REPLACE PACKAGE DateDemo AS
  PROCEDURE Go;
  PROCEDURE Process(p_DateVal IN OWA_UTIL.dateType);
END DateDemo;

CREATE OR REPLACE PACKAGE BODY DateDemo AS
  -- Установим c_OWAPath для виртуального маршрута (включая DCD),
  -- указывающего на место, где инсталлирован Oracle Web Agent.
  c_OWAPath CONSTANT VARCHAR2(50) := '/example/owa/';

  -- Принимает выбранные данные.
  PROCEDURE Process(p_DateVal IN OWA_UTIL.dateType) IS
  BEGIN
      HTP.htmlOpen;
      HTP.headOpen;
      HTP.title('Results');
      HTP.headClose;
      HTP.bodyOpen;
      HTP.p('You picked ');
```

```
        HTP.p(TO_CHAR(OWA_UTIL.todate(p_DateVal)));
        HTP.nl;
        OWA_UTIL.signature('DateDemo.Process');
        HTP.bodyClose;
        HTP.htmlClose;
    END Process;

    -- Основная процедура ввода.
    PROCEDURE Go IS
    BEGIN
        HTP.htmlOpen;
        HTP.headOpen;
        HTP.title('OWA_UTIL Date Utilities');
        HTP.headClose;

        HTP.bodyOpen;
        HTP.p('Welcome to the Date demo. Here''s a calendar, ');
        HTP.p('created with ' || HTF.bold('OWA_UTIL.calendarprint'));
        HTP.p(':' || HTF.nl);
        OWA_UTIL.calendarprint(
           'SELECT * FROM calendar ORDER BY today');

        HTP.para;
        HTP.p('Enter a year, month, and day below, and then press');
        HTP.p(' submit:' || HTF.nl);
        HTP.formOpen(curl => c_OWAPath || 'DateDemo.Process');
        OWA_UTIL.choose_date(p_name => 'p_DateVal');
        HTP.formSubmit;
        HTP.formClose;
        OWA_UTIL.signature('DateDemo.Go');
        HTP.bodyClose;
        HTP.htmlClose;
    END Go;
END DateDemo;
```

# OWA_IMAGE

С помощью модуля OWA_IMAGE можно обрабатывать карты образов. *Карта образа* (image map) – это график, находящийся на Web-странице. В качестве точки привязки (anchor) для связи может быть использован любой график, однако карта образов чувствует место на графике, на котором пользователь щелкает клавишей мыши. Беря за основу координаты x и y графика, Web-сервер возвращает соответствующий URL. В отличие от других Web-серверов, в которых карты образов обрабатываются при помощи специальной программы CGI, Web-сервер Oracle может обрабатывать их внутрисистемно. Процедура, обращающаяся к модулю OWA_IMAGE, используется в качестве адресата карты образов и обрабатывает координаты x и y. В OWA_IMAGE определен один тип данных – POINT, с помощью которого задаются значения x и y. Для считывания этих значений применяются две функции:

```
FUNCTION GET_X(p IN POINT) RETURN INTEGER;
FUNCTION GET_Y(p IN POINT) RETURN INTEGER;
```

где *p* – это указатель, выбранный с картой образа. Один из способов использования OWA_IMAGE иллюстрируется на примере модуля **Imagemap**:

```
-- Этот пример содержится в файле imagemap.sql.
CREATE OR REPLACE PACKAGE Imagemap AS
  PROCEDURE Go;
  PROCEDURE Process(p_Img IN OWA_IMAGE.POINT);
END Imagemap;
```

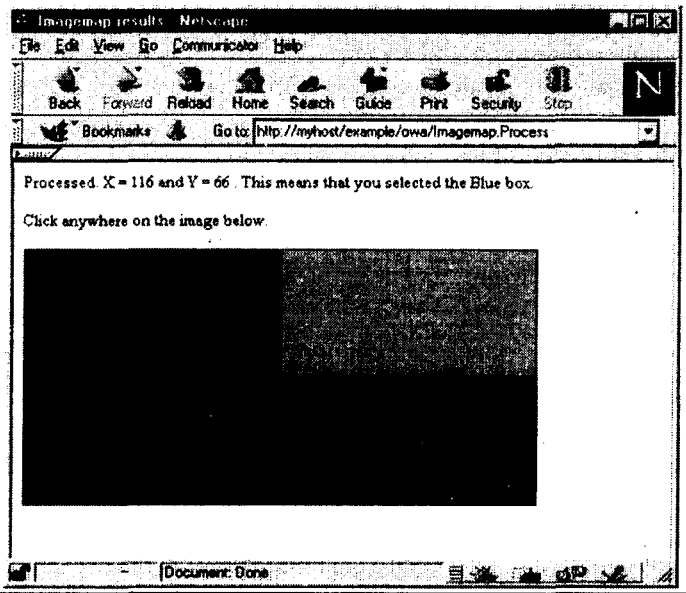
```
CREATE OR REPLACE PACKAGE BODY Imagemap AS
  -- Установим c_ImagePath для виртуального маршрута, указывающего
  -- место хранения на сервере файла boxes.gif.
  c_ImagePath CONSTANT VARCHAR2(50) := '/ows-img/boxes.gif';

  -- Установим c_OWAPath для виртуального маршрута (включая DCD),
  -- указывающего место, где инсталлирован Oracle Web Agent.
  c_OWAPath CONSTANT VARCHAR2(50) := '/example/owa/';
  PROCEDURE ShowBoxes IS
  BEGIN
      HTP.p('Click anywhere on the image below.' || HTF.para);
      -- Создадим форму с одним образом, вызываемым щелчком мыши.
      -- Адресатом этого образа является процедура Process. Поскольку
      -- тип этого поля формы — image, параметр p_Img передает
      -- значение типа OWA_IMAGE.POINT.
      HTP.formOpen(curl => c_OWAPath || 'Imagemap.Process');
      HTP.formImage(cname => 'p_Img',
                    csrc => c_ImagePath,
                    cattributes => 'BORDER=0');
      HTP.formClose;
  END ShowBoxes;

  PROCEDURE Go IS
  BEGIN
      HTP.htmlOpen;
      HTP.headOpen;
      HTP.title('Imagemap test');
      HTP.headClose;

      HTP.bodyOpen;
      ShowBoxes;
      HTP.bodyClose;
      HTP.htmlClose;
  END Go;

  PROCEDURE Process(p_Img IN OWA_IMAGE.POINT) IS
      v_x INTEGER := OWA_IMAGE.GET_X(p_Img);
      v_y INTEGER := OWA_IMAGE.GET_Y(p_Img);
      v_Color VARCHAR2(5);
  BEGIN
      IF v_x < 200 THEN
         IF v_y < 100 THEN
            v_Color := 'Blue';
         ELSE
            v_Color := 'Red';
         END IF;
      ELSE
         IF v_y < 100 THEN
            v_Color := 'Green';
         ELSE
            v_Color := 'Black';
         END IF;
      END IF;
      HTP.htmlOpen;
```

```
        HTP.headOpen;
        HTP.title('Imagemap results');
        HTP.headClose;

        HTP.bodyOpen;
        HTP.print('Processed. X = ' || v_x || ' and Y = ' || v_y);
        HTP.print('. This means that you selected the ' || v_Color);
        HTP.print(' box.' || HTF.para);

        ShowBoxes;
        HTP.bodyClose;
        HTP.htmlClose;
    END Process;
END Imagemap;
```

**▼ ВНИМАНИЕ** Для того чтобы выполнить рассмотренный модуль в своей системе, скопируйте файл boxes.gif (находящийся на прилагаемом компакт-диске) в каталог образов Web-сервера и укажите маршрут в **c_ImagePath**.

Для установления карты образа необходимо создать форму с одним образом, вызываемым щелчком мыши. Адресатом этого образа является процедура **ImagemapProcess**. Форма создается процедурой **ShowBoxes**. Образ boxes.gif представляет собой четыре квадрата разного цвета. Результаты вызова **Imagemap.Go** и щелчка мышью на голубом квадрате (в Netscape Navigator 4.0) показаны на рис. 19.9.

**Рис. 19.9.**
*Результаты щелчка мыши на карте образа*




# OWA_COOKIE

Модуль OWA_COOKIE дает пользователям возможность работать с так называемыми домашними данными HTML. Домашние данные (cookie) – это фрагмент данных, хранимый не на сервере, а на станции клиента. Домашние данные используются для обеспечения работы системы в перерыве между обращениями к серверу. Во время обычного процесса обработки информации клиент не знает о существовании домашних данных, однако во многих браузерах устанавливается специальный параметр, позволяющий уведомлять пользователя о передаче домашних данных и, при желании, отказываться от них.

## Типы данных

В спецификации HTTP на домашние данные особо оговаривается, что они представляют собой пару имя-значение. Значение может быть установлено или считано по конкретному имени. Размер имени и

значения ограничен 4096 байт, и под одним именем может храниться несколько значений. Для выполнения этого условия в модуле OWA_COOKIE созданы следующие типы данных:

```
TYPE VC_ARR IS TABLE OF VARCHAR2(4096)
   INDEX BY BINARY_INTEGER;
TYPE COOKIE IS RECORD (
   name      VARCHAR2(4096),
   vals      VC_ARR,
   num_vals  INTEGER);
```

Тип данных OWA_COOKIE.COOKIE реализован в виде записи, содержащей таблицу PL/SQL, поэтому с помощью данного типа можно хранить несколько значений.

## SEND

Процедура SEND используется для передачи домашних данных на станцию клиента для хранения:

```
PROCEDURE SEND(name IN VARCHAR2,
               value IN VARCHAR2,
               expires IN DATE DEFAULT NULL,
               path IN VARCHAR2 DEFAULT NULL,
               domain IN VARCHAR2 DEFAULT NULL,
               secure IN VARCHAR2 DEFAULT NULL);
```

Процедуру SEND необходимо вызывать из заголовка страницы HTTP (см. пример в конце этого раздела). SEND генерирует строку заголовка HTTP, которая выглядит следующим образом:

```
Set-Cookie:   name=value expires=expires path=path
              domain=domain [secure]
```

Обязательными являются параметры *name* (имя) и *value* (значение); если не указан какой-либо другой параметр, то не создается соответствующая фраза. Параметр *expires* указывает момент истечения срока действия домашних данных; после указанного момента эти данные не могут быть считаны. Если этот параметр не задан, то срок действия домашних данных истекает с окончанием сеанса работы клиента (т.е. при закрытии браузера).

## GET

Функция GET возвращает домашние данные с указанным именем. Если на станции клиента нет данных с указанным именем, то значение параметра *num_vals* в возвращаемых домашних данных будет равно нулю. Описание этой функции выглядит следующим образом:

```
FUNCTION GET(name IN VARCHAR2) RETURN COOKIE;
```

где *name* — имя нужных домашних данных.

## GET_ALL

Процедура GET_ALL возвращает все пары имя-значение домашних данных, считываемых браузером, в том порядке, в котором они были посланы:

```
PROCEDURE GET_ALL(names OUT VC_ARR,
                  vals OUT VC_ARR,
                  num_vals OUT INTEGER);
```

## REMOVE

Процедура REMOVE форсирует истечение срока существующих домашних данных. Как и SEND, эту процедуру нужно вызывать из заголовка HTTP:

```
PROCEDURE REMOVE(name IN VARCHAR2,
                 val IN VARCHAR2,
                 path IN VARCHAR2 DEFAULT NULL);
```

Она создает строку следующего вида:

```
Set-Cookie: name=value path=path expires=01-JAN-1990
```

Если не указан параметр *path*, то фраза, определяющая маршрут, опускается. Прекращение срока действия вызывается установкой в качестве срока действия некоторой даты, которая была передана (здесь 1 января 1990 года).

## Пример

Использование модуля OWA_COOKIE демонстрируется на примере процедуры CookieDemo.

```
-- Этот пример содержится в файле cookie.sql.
CREATE OR REPLACE PROCEDURE CookieDemo(
  p_NewVal IN NUMBER DEFAULT NULL) AS
  -- Установим c_OWAPath для виртуального маршрута (включая DCD),
  -- указывающего место, где инсталлирован Oracle Web Agent.
  c_OWAPath CONSTANT VARCHAR2(50) := '/example/owa/';

  v_Cookie OWA_COOKIE.COOKIE;
BEGIN
  -- Считаем текущее значение домашних данных и на основании
  -- полученных параметров пошлем новое значение.
  v_Cookie := OWA_COOKIE.GET('Count');
  OWA_UTIL.MIME_HEADER('text/html', FALSE);
  IF p_NewVal IS NOT NULL THEN
      OWA_COOKIE.SEND('Count', p_NewVal + 1);
  ELSIF (v_Cookie.num_vals = 0) THEN
      OWA_COOKIE.SEND('Count', 0);
  ELSE
      OWA_COOKIE.SEND('Count', v_Cookie.vals(1) + 1);
  END IF;
  OWA_UTIL.HTTP_HEADER_CLOSE;

  HTP.htmlOpen;
  HTP.headOpen;
  HTP.title('Cookie Demo');
  HTP.headClose;

  HTP.bodyOpen;
  HTP.print('Welcome to the cookie demo, using the OWA_COOKIE ');
  HTP.print('package.' || HTF.para);
  IF p_NewVal IS NOT NULL THEN
      HTP.print('The cookie value has been reset to ');
      HTP.bold(p_NewVal);
      HTP.print('.' || HTF.para);
  ELSIF v_Cookie.num_vals > 0 THEN
      HTP.print('The cookie value is now ');
      HTP.bold(v_Cookie.vals(1));
      HTP.print('.' || HTF.para);
  ELSE
      HTP.print('The cookie value has been set to zero.');
  END IF;
  HTP.para;
  HTP.print('Click ');
  HTP.anchor(c_OWAPath || 'CookieDemo', 'here');
  HTP.print(' to increment the value, or enter a new value in');
  HTP.print(' the box below and click Submit to reset it.');
  HTF.para;

  HTP.formOpen(curl => c_OWAPath || 'CookieDemo');
  HTP.formText(cname => 'p_NewVal',
               csize => 10);
```

```
    HTP.formSubmit;
    HTP.formClose;
    OWA_UTIL.signature('CookieDemo');
    HTP.bodyClose;
    HTP.htmlClose;
END CookieDemo;
```

# Среды разработки процедур OWA

Разработка хранимых процедур для Web-агента Oracle во многом похожа на процесс создания других хранимых процедур – пользователь имеет возможность выбрать желаемую среду разработки. В двух средах – SQL*Plus и SQL-Station – имеются утилиты, которые можно использовать при создании процедур такого рода.

## OWA_UTIL.SHOWPAGE

Результат выполнения процедуры, которая вызывается из Web-агента, можно наблюдать, как правило, только в Web-браузере. Поэтому такие инструментальные средства, как SQL*Plus, в данном случае не очень полезны. Однако процедура OWA_UTIL.SHOWPAGE, обращаясь к модулю DBMS_OUTPUT, отображает на экране результат выполнения последней процедуры в формате HTML. Пример использования этой процедуры в SQL*Plus приведен ниже.

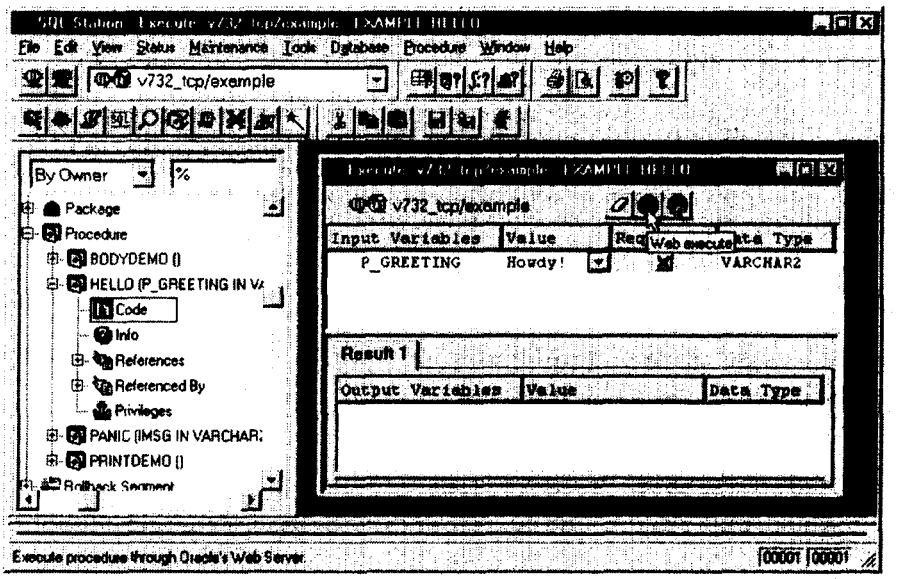
```
SQL> exec BodyDemo
PL/SQL procedure successfully completed.

SQL> set SERVEROUTPUT ON
SQL> exec OWA_UTIL.SHOWPAGE
<HTML>
<HEAD>
<TITLE>Body tags demo</TITLE>
</HEAD>
<BODY>
This page demonstrates several of the tags .
available for formatting the text in the body
of a web page, and for inserting additional
tags.<P>
<H2>Links</H2>
Here are some links to check out. These are
generated using the HTP.anchor procedure.
<P>
<A HREF="//www.oracle.com/">Oracle Corporation</A>
<P>
<A HREF="//www.osborne.com/oracle/index.htm">Oracle Press</A>
<P>
<A HREF="//www.platinum.com/">Platinum Corporation</A>
<P>
<H2>Formats</H2>
<CENTER>These lines are centered, using
<BR>
the HTF.centerOpen and HTF.centerClose functions.
</CENTER>
<STRIKE>Here's some strikethrough text.</STRIKE>
</BODY>
</HTML>
PL/SQL procedure successfully completed.
```

С помощью SHOWPAGE можно проверить правильность процедуры, проанализировав результаты ее выполнения как текст в формате HTML.

## SQL-Station Coder

Процедуры OWA можно выполнять и в SQL-Station Coder. В окне выполнения процедуры находятся три кнопки. С помощью первой кнопки процедура выполняется обычным образом, а следующие две кнопки используются для вызова процедуры с помощью Web-агента (см. рис. 19.10). Кнопка Web execute вызывает браузер на станции клиента и передает ему правильный URL для исполнения процедуры с указанными параметрами методом GET. Кнопка Generate URL дает пользователю возможность увидеть (и при желании отредактировать) создаваемый URL и скопировать его в буфер обмена. Таким образом, с помощью SQL-Station Coder можно создавать Web-процедуры так же, как и обычные процедуры. Более подробно о программе SQL-Station Coder рассказано в главе 13, а также во встроенной документации на SQL-Station.

**Рис. 19.10.**
*Выполнение процедуры OWA в SQL-Station Coder*




# Итоги

В этой главе была рассмотрена программа Oracle WebServer, а также обсуждены способы работы с ней при помощи PL/SQL, который является неотъемлемой частью данной программы. С помощью Web-агента PL/SQL хранимые процедуры PL/SQL могут генерировать выходные данные формата HTML, что позволяет создавать динамические Web-страницы, на которых представлена информация базы данных. Кроме того, здесь было рассказано о модулях программного Web-пакета, которые применяются для создания Web-страниц, в том числе и о модуле OWA_UTIL. Завершается глава обсуждением сред разработки, в которых создаются процедуры Web-агента.

# Глава 20

## *Внешние процедуры*

PL/SQL – это чрезвычайно эффективный язык программирования, в состав которого входит много различных средств, но некоторые возможности в нем не предусмотрены. К таким возможностям относятся взаимодействие с файловой системой и другими компонентами операционной системы, а также действия над комплексными числами. В таких ситуациях удобнее использовать другие языки программирования третьего поколения, например C. В Oracle8 предусмотрена возможность вызова процедур, написанных на C, в качестве внешних процедур непосредственно из PL/SQL, что позволяет интегрировать в среду PL/SQL различные дополнительные возможности. В этой главе обсуждены принципы работы внешних процедур, в том числе и требования, необходимые для их функционирования.

# Понятие внешней процедуры

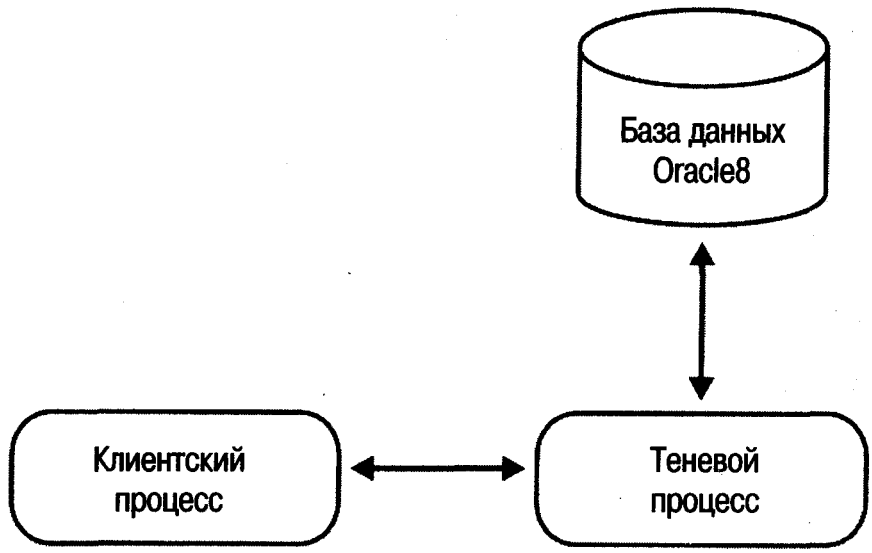
PL/SQL 8.0 ... и ВЫШЕ

*Внешняя процедура* (external procedure) – это процедура, написанная на языке программирования, отличном от PL/SQL, которую можно вызывать из любой программы PL/SQL. Внешние процедуры впервые появились в Oracle8. В Oracle8 версии 8.0.3 единственным языком программирования, на котором можно создавать внешние процедуры, является язык C, однако в следующих версиях Oracle8 будет предусмотрена поддержка и других языков (таких, как Java или C++). Поэтому синтаксис объявления и вызова внешних процедур является обобщенным – он шире, чем это необходимо в текущей версии PL/SQL.

Каким же образом вызываются внешние процедуры? Взаимодействие между клиентским процессом и сервером Oracle8 представлено на рисунке 20.1. Клиентский процесс взаимодействует с так называемым теневым процессом, который, в свою очередь, общается с остальной частью базы данных. Клиентский процесс посылает теневому процессу SQL-операторы и блоки PL/SQL, а тот выполняет их и отправляет полученные результаты назад. Более подробно о клиентских и теневых процессах, а также о взаимодействии этих процессов с другими процессами Oracle рассказано в главе 22.

Для выполнения внешней процедуры необходим ряд дополнительных процессов и библиотек, кото-

**Рис. 20.1.**
*Клиентский и теневой процессы*

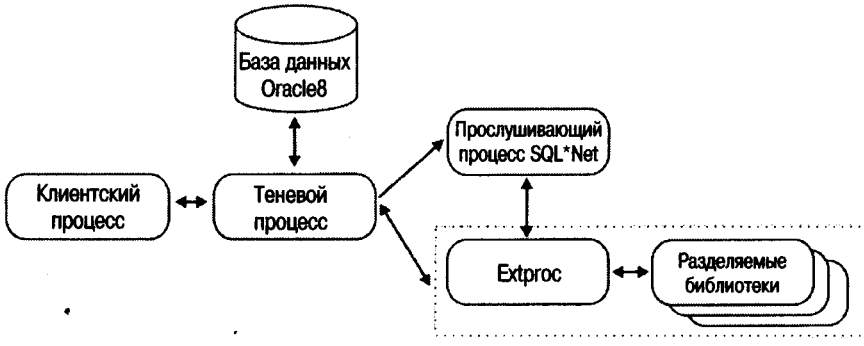



рые представлены на рисунке 20.2. Когда теневой процесс вызывает внешнюю процедуру первый раз, прослушивающий процесс SQL*Net порождает специальный процесс extproc. Этот процесс динамически загружает разделяемую библиотеку (или несколько разделяемых библиотек), в которой находится процедура C, а затем выполняет эту процедуру, посылая результаты обратно теневому процессу, который направляет их клиенту. После порождения процесса extproc он функционирует в течение всего сеанса работы. Для того чтобы выполнить дополнительные процедуры, теневой процесс и процесс extproc могут взаимодействовать напрямую, без участия прослушивающего процесса.

При использовании внешних процедур следует выполнить ряд требований:

- Внешняя процедура должна находиться в разделяемой библиотеке. При вызове процедуры процесс extproc динамически загружает библиотеку. Если затем в течение данного сеанса вызывается внешняя процедура, находящаяся в другой библиотеке, то extproc загружает и эту библиотеку. Таким

**Рис. 20.2.**
*Прослушивающий процесс SQL*Net и процесс extproc*




образом, выполнение внешних процедур обеспечивается только в том случае, когда в базовой операционной системе поддерживается функционирование разделяемых библиотек. Например, в Windows NT внешняя процедура может быть скомпилирована в виде DLL, а в системе Solaris – в виде разделяемого объекта (.so).

- Для каждого сеанса, вызывающего внешнюю процедуру, прослушивающий процесс SQL*Net порождает отдельный процесс extproc.
- Прослушивающий процесс и процесс extproc должны функционировать на той же машине, что и база данных. Как будет показано далее, синтаксис создания библиотечного объекта словаря данных не имеет средств для задания хост-имени.
- Внешняя процедура может, в свою очередь, выполнять обратные вызовы базы данных для того, чтобы обрабатывать SQL-операторы или блоки PL/SQL. Обратные вызовы выполняются с помощью интерфейса вызовов Oracle8 (OCI – Oracle8 Call Interface) и обсуждаются в разделе "Обратные вызовы базы данных" далее в этой главе.

## Порядок вызова внешней процедуры

Для вызова внешней процедуры необходимо выполнить следующие операции:

1. Написать процедуру на языке C и скомпилировать ее в разделяемую библиотеку.
2. Сконфигурировать файлы параметров SQL*Net и запустить прослушивающий процесс.
3. Создать библиотечный объект словаря данных для представления библиотеки операционной системы.
4. Создать в PL/SQL процедуру-оболочку, устанавливающую соответствие между параметрами PL/SQL и параметрами C.

Теперь подробно рассмотрим каждую из этих операций.

### Создание процедуры

Первый шаг – это создание самой процедуры. Предположим, что необходимо создать файл операционной системы и записать в него строку символов. Это можно сделать с помощью процедуры **OutputString**:

```
/* Этот пример является частью файла outstr.c. */
#include <stdio.h>
```

```
/* Записывает строку символов, которая содержится в сообщении, в
   файл, находящийся по указанному маршруту. Если файл
   не существует, он будет создан./
void OutputString(path, message)
char *path;
char *message; {

FILE *file_handle;

/* Откроем файл на запись. */
file_handle = fopen(path, "w");

/* Выведем строку, завершив ее символом новой строки. */
fprintf(file_handle, "%s\n", message);

/* Закроем файл. */
fclose(file_handle);
}
```

После создания файла необходимо скомпилировать его в разделяемую библиотеку. В системе Solaris это можно сделать с помощью команды

```
cc -G -o /home/utils/stringlib.so outstr.c
```

которая создаст разделяемую библиотеку /home/utils/stringlib.so. Команды, применяемые для этой цели в других операционных системах, сильно различаются — за более подробной информацией обращайтесь к документации на операционную систему и/или к документации на соответствующий компилятор.

▼ СОВЕТУЕМ В системах Unix целевые объекты для компонующих разделяемых библиотек содержатся в формирующем файле **$ORACLE_HOME/rdbms/demo/demo_rdbms.mk.** С помощью объекта **extproc_nocallback** будет создаваться библиотека, в которой не выполняются обратные вызовы базы данных, а с помощью объекта **extproc_callback** — библиотека, в которой обратные вызовы выполняются (обратные вызовы обсуждаются ниже, в разделе "Обратные вызовы базы данных"). Инструкции по использованию этого формирующего файла и примеры работы с ним приведены в тексте этого файла.

## Конфигурирование прослушивающего процесса SQL*Net

Конфигурацию прослушивающего процесса необходимо устанавливать только однажды. После того как он сконфигурирован и запущен на выполнение, при необходимости автоматически порождается процесс extproc. Для установления конфигурации прослушивающего процесса нужны два файла: listener.ora и tnsnames.ora. После создания этих файлов прослушивающий процесс может быть запущен.

Местонахождение конфигурационных файлов SQL*Net может быть различно и зависит от применяемой операционной системы. К примеру, в Unix каталогом по умолчанию для этих файлов является каталог $ORACLE_HOME/network/admin. Местонахождение конфигурационных файлов может быть изменено указанием переменной среды TNS_ADMIN.

**listener.ora** В этом файле указаны параметры прослушивающего процесса, например:

```
#Этот пример содержится в файле listener.ora.
#Пример файла listener.ora для внешних процедур.
#Заменим <<ORACLE_HOME>> на каталог $ORACLE_HOME,
#заменим <<LISTENER_KEY>> на ключ IPC,
#заменим <<EXTPROC_SID>> на идентификатор процесса extproc.
listener =
  (ADDRESS_LIST =
     (ADDRESS =
        (PROTOCOL = ipc)
        (KEY = <<LISTENER_KEY>>)
     )
  )
```

```
sid_list_listener =
   (SID_LIST =
      (SID_DESC =
         (SID_NAME = <<EXTPROC_SID>>)
         (ORACLE_HOME = <<ORACLE_HOME>>)
         (PROGRAM = extproc)
      )
   )
```

Ключевым местом этого файла является раздел (PROGRAM=extproc) в списке SID. Этот фрагмент означает, что данное соединение должно использоваться для порождения процесса extproc, а не для установления связи с базой данных.

**tnsnames.ora** Этот файл используется для указания строк соединения SQL*Net. Для процесса extproc существует специальная строка соединения – extproc_connection_data, которая определяется в tnsnames.ora, например:

```
#Этот пример содержится в файле tnsnames.ora.
#Пример файла tnsnames.ora для внешних процедур.
#Заменим <<LISTENER_KEY>> на ключ IPC,
#заменим <<EXTPROC_SID>> на идентификатор процесса extproc.
extproc_connection_data =
   (DESCRIPTION =
      (ADDRESS =
         (PROTOCOL = ipc)
         (KEY = <<LISTENER_KEY>>)
      )
   (CONNECT_DATA =
      (SID = <<EXTPROC_SID>>)
   )
)
```

Системный идентификатор (SID) в разделе CONNECT_DATA обоих файлов, tnsnames.ora и listener.ora, должен быть один и тот же. Он может быть системным идентификатором базы данных, хотя это вовсе не обязательно. Кроме того, одинаковым для обоих файлов должен быть и раздел ADDRESS.

**▼ ВНИМАНИЕ** Прослушивающий процесс (или группа прослушивающих процессов) может ждать сигнала как для внешних процедур, так и для новых соединений с базой данных, поэтому в файлах конфигурации могут быть указаны дополнительные параметры для соединений с базой данных. Более подробно о дополнительных параметрах файлов tnsnames.ora и listener.ora рассказано в руководстве Net8 Administrator's Guide.

**Запуск прослушивающего процесса** Прослушивающий процесс запускается (и останавливается) с помощью утилиты lsnrctl (или lsnrctl80). Эта утилита запускается из командной строки операционной системы, и ее символьный интерфейс похож на интерфейс SQL*Plus. Предположим, что файл listener.ora сконфигурирован так, как это показано в предыдущих примерах, с V803 в качестве системного идентификатора внешней процедуры и V803_extproc в качестве ключа IPC. Тогда прослушивающий процесс может быть запущен следующим образом:

```
$ lsnrctl start
LSNRCTL for Solaris: Version 8.0.3.0.0 - Production on 12-AUG-97
01:03:55

(c) Copyright 1997 Oracle Corporation. All rights reserved.
Starting /oracle/app/oracle/product/8.0.3/bin/tnslsnr: please wait...

TNSLSNR for Solaris: Version 8.0.3.0.0 - Production
System parameter file is
  /oracle/app/oracle/product/8.0.3/network/admin/listener.ora
```

```
Log messages written to
  /oracle/app/oracle/product/8.0.3/network/log/listener.log
Listening on: (ADDRESS=(PROTOCOL=ipc) (DEV=9) (KEY=V803_extproc))
Connecting to (ADDRESS=(PROTOCOL=ipc) (KEY=V803_extproc))
STATUS of the LISTENER
------------------------
Alias                LISTENER
Version              TNSLSNR for Solaris: Version 8.0.3.0.0
- Production
Start Date           12-AUG-97 01:04:00
Uptime               0 days 0 hr. 0 min. 1 sec
Trace Level          off
Security             OFF
SNMP                 OFF
Listener Parameter File
  /oracle/app/oracle/product/8.0.3/network/admin/listener.ora
Listener Log File
  /oracle/app/oracle/product/8.0.3/network/log/listener.log
Services Summary...
  V803 has 1 service handler(s)
The command completed successfully
```

Более подробно об использовании утилиты lsnrctl рассказано в руководстве Net8 Administrator's Guide.

## Создание библиотеки

*Библиотека* (library) — это объект словаря данных, в котором содержится информация о местонахождении на диске разделяемой библиотеки. Поскольку разделяемая библиотека располагается вне базы данных, в базе данных должен существовать объект, делающий эту библиотеку доступной для PL/SQL. Библиотеки создаются с помощью команды DDL, называемой CREATE LIBRARY:

CREATE LIBRARY *имя_библиотеки* {IS | AS}
*'маршрут_операционной_системы';*

где *имя_библиотеки* — это имя новой библиотеки, а *маршрут_операционной_системы* — полный путь к разделяемой библиотеке в файловой системе. Например, можно создать библиотеку **stringlib** при помощи следующего оператора:

```
CREATE LIBRARY stringlib AS
'/home/utils/stringlib.so';
```

В этот момент база данных не проверяет фактическое наличие разделяемой библиотеки операционной системы. Однако, если библиотека не существует, при попытке вызова процедуры, описанной с использованием этой библиотеки, возвращается сообщение об ошибке.

Библиотеку можно удалить с помощью следующей команды:

DROP LIBRARY *имя_библиотеки*;

где *имя_библиотеки* — это имя удаляемой библиотеки.

**Привилегии на библиотеки** Для того чтобы создать библиотеку, необходимо иметь системную привилегию CREATE LIBRARY, которая аналогична привилегиям, требуемым для выполнения других команд DDL. После создания библиотеки можно позволить другим пользователям работать с ней, если предоставить им привилегию EXECUTE на эту библиотеку.

**Библиотеки в словаре данных** Информация о библиотеках, как и о других объектах базы данных, хранится в словаре данных в таблицах **user_libraries, all_libraries** и **dba_libraries**. Основные столбцы этих таблиц описаны в таблице 20.1.

Сведения о библиотеке **stringlib** отображаются с помощью следующего запроса:

```
SQL> SELECT * FROM user_libraries
  2      WHERE library_name = 'STRINGLIB';
LIBRARY_NAME     FILE_SPEC                    D STATUS
------------     ----------------------       ------------------
STRINGLIB        /home/utils/stringlib.so     Y VALID
```

Более подробно о словаре данных и о другой информации, хранимой в нем, рассказано в приложении D.

**Таблица 20.1.**

| Столбец | Тип данных | Описание |
|---|---|---|
| library_name | VARCHAR2(30) | Имя библиотеки |
| file_spec | VARCHAR2(2000) | Маршрут к разделяемой библиотеке в операционной системе, указанный оператором CREATE LIBRARY |
| dynamic | VARCHAR2(1) | Если Y, то библиотека является динамической, если N — то нет. В Oracle8 варианта 8.0 можно создавать только динамические библиотеки |
| status | VARCHAR2(7) | VALID (достоверна) или INVALID (недостоверна); этот столбец аналогичен столбцам status других представлений словаря |

## Создание процедуры-оболочки

Для того чтобы вызвать внешнюю процедуру, необходимо создать *процедуру-оболочку* (wrapper procedure). При создании такой процедуры преследуются три цели: отобразить параметры PL/SQL на параметры C, создать заполнитель, с помощью которого вызывающие процедуры будут определять существующие зависимости, и сообщить PL/SQL имя внешней библиотеки. Процедура-оболочка состоит из спецификации, или описания с параметрами, за которой следует конструкция EXTERNAL:

```
CREATE [OR REPLACE] PROCEDURE имя_процедуры [список_параметров]
  AS EXTERNAL
  LIBRARY имя_библиотеки
  [NAME внешнее_имя]
  [LANGUAGE название_языка]
  [CALLING STANDARD {C | PASCAL}]
  [WITH CONTEXT]
  [PARAMETERS (список_внешних_параметров)];
```

Конструкц
ии вызова этой процедуры рассмотрены в следующих разделах.

**LIBRARY** Это единственная обязательная конструкция, и она используется для указания библиотеки, которая представляет разделяемую библиотеку операционной системы, содержащую процедуру C. Эта библиотека должна находиться в текущей схеме либо пользователь должен иметь на библиотеку привилегию EXECUTE.

**NAME** Эта конструкция указывает имя процедуры C. Если имя не указано, то по умолчанию процедуре присваивается имя процедуры-оболочки PL/SQL. Обратите внимание на то, что имена PL/SQL хранятся в виде прописных символов, поэтому, если имя процедуры C хранится в виде строчных символов или символов обоих регистров, то идентификатор нужно обязательно заключать в двойные кавычки.

**LANGUAGE** Конструкция LANGUAGE используется для указания языка, на котором написана внешняя процедура. Если язык не указан, то языком по умолчанию считается язык C, который является единственным языком внешних процедур в версии 8.0.3.

**CALLING STANDARD** Это стандарт вызова, который может быть либо C, либо PASCAL. Стандарт вызова определяет порядок размещения параметров в стеке. Стандартом по умолчанию является C, который следует использовать для внешних процедур, написанных на C. Когда стандартом вызова является PASCAL, внешние процедуры при считывании параметров отвечают за явное извлечение их из стека, а сами параметры расположены в обратном порядке.

**WITH CONTEXT** Эта конструкция добавляет в процедуру C параметр, который можно использовать для получения описателей OCI, применяемых при обратных вызовах базы данных. Конструкция WITH CONTEXT подробно рассмотрена в разделе "Обратные вызовы базы данных" далее в этой главе.

**PARAMETERS** Конструкция PARAMETERS указывает способ отображения параметров PL/SQL процедуры-оболочки на параметры C. Если этот способ не указан, то используется отображение, установленное по умолчанию. Конструкция PARAMETERS подробно рассмотрена в следующем разделе — "Отображение параметров".

Ниже приведен пример процедуры-оболочки для **OutputString**.

```
-- Этот пример является частью файла outstr.sql.
CREATE OR REPLACE PROCEDURE OutputString(
  p_Path IN VARCHAR2,
  p_Message IN VARCHAR2) AS EXTERNAL

  LIBRARY stringlib
  NAME "OutputString"
  PARAMETERS (p_Path STRING,
              p_Message STRING);
```

После создания процедуры-оболочки можно вызывать **OutputString** непосредственно из блока PL/SQL. Например, ниже показано, как с помощью SQL*Plus в каталоге /tmp создается файл output.txt, состоящий из одной строки "Hello World!".

```
SQL> BEGIN
  2      OutputString ('/tmp/output.txt', 'Hello World!');
  3      END;
  4      /
PL/SQL procedure successfully completed.

SQL> exit
Disconnected from Oracle8 Server Release 8.0.3.0.0 - Production
With the Partitioning and Objects options
PL/SQL Release 8.0.3.0.0 - Production
$ cat /tmp/output.txt
Hello World!
```

## Отображение параметров

Одним из самых важных вопросов взаимодействия двух языков является преобразование типов данных. Это справедливо и в случае внешних процедур: необходимо указать, как типы данных PL/SQL, используемые для параметров процедуры-оболочки, отображаются на типы данных C, используемые для параметров внешней процедуры. Помимо соответствия самих типов данных необходимо разрешить ряд других вопросов:

- В PL/SQL любая переменная может иметь NULL-значение. В языке C понятия NULL-значения не существует, поэтому для указания того, является ли параметр NULL-значением или нет, можно использовать дополнительную переменную, называемую *индикатором* (indicator).
- В PL/SQL существуют типы данных, отсутствующие в C, а в C существуют типы данных, отсутствующие в PL/SQL. Например, в C нет типов данных DATE и BOOLEAN, а в PL/SQL не определено различие между одно-, двух- и четырехбайтовыми целыми числами.
- В PL/SQL версии 8 разрешается использовать символьные переменные в различных наборах символов. Эти наборы символов нужно каким-то образом связать с C.
- В C необходимо знать текущую и/или максимальную длину строк символов, передаваемых из PL/SQL.
- В PL/SQL необходимо знать текущую и/или максимальную длину строк символов, передаваемых из C.

Для устранения возможных проблем применяется конструкция PARAMETERS:

PARAMETERS (*список_внешних_параметров*);

где *список_внешних_параметров* — это список параметров, каждый из которых имеет следующую структуру:

{*имя_параметра* | RETURN} *свойство* [BY REF] [*внешний_тип_данных*]

Здесь *имя_параметра* — это имя параметра, *внешний_тип_данных* — тип данных C, присваиваемый этому параметру, а *свойство* — одно или несколько свойств из следующих: INDICATOR, LENGTH, MAXLEN, CHARSETID и CHARSETFORM. Все эти свойства будут рассмотрены ниже.

## Сравнение типов данных PL/SQL и типов данных C

Поскольку наборы типов данных в PL/SQL и C различны, в Oracle8 для каждого типа данных PL/SQL предусмотрен тип данных C по умолчанию. При желании эти установки можно изменить. Для облегчения этого процесса в Oracle определен набор внешних типов данных, которые описаны в таблице 20.2. *Внешний тип данных* (external datatype) – это мнемоническое обозначение часто применяемого типа данных C. Внешние типы данных удобно использовать также в программах Pro*C и OCI. Кроме того, в Oracle определены дополнительные внешние типы данных, которые не являются стандартными для C, например OCILOBLOCATOR. Все доступные внешние типы данных для каждого из типов данных PL/SQL приведены в таблице 20.3.

### Таблица 20.2. Внешние типы данных

| Внешний тип данных | Описание | Тип данных C |
|---|---|---|
| CHAR[1] | Однобайтовая величина, используемая для хранения символа или целого числа в диапазоне от -128 до +127 | char |
| UNSIGNED CHAR[1] | Однобайтовая величина, используемая для хранения целого числа в диапазоне от 0 до 255 | unsigned char |
| SHORT[1,2] | Обычно двухбайтовая величина, используемая для хранения целого числа в диапазоне от $-2^{15}$ до $+2^{15}-1$ | short |
| UNSIGNED SHORT[1,2] | Обычно беззнаковая двухбайтовая величина, используемая для хранения целого числа в диапазоне от 0 до $+2^{16}$ | unsigned short |
| INT[1,2] | Обычно четырехбайтовая величина, используемая для хранения целого числа в диапазоне от $-2^{31}$ до $+2^{31}-1$ | int |
| UNSIGNED INT[1,2] | Обычно беззнаковая четырехбайтовая величина, используемая для хранения целого числа в диапазоне от 0 до $2^{32}$ | unsigned int |
| LONG[1,2] | Обычно четырехбайтовая величина, используемая для хранения целого числа в диапазоне от $-2^{31}$ до $+2^{31}-1$ | long |
| UNSIGNED LONG[1,2] | Обычно беззнаковая четырехбайтовая величина, используемая для хранения целого числа в диапазоне от 0 до $2^{32}$ | unsigned long |
| SIZE_T[1,2] | Число байтов, указанное операционной системой и компилятором | size_t |
| SB1[3] | Знаковая однобайтовая величина | sb1 |
| UB1[3] | Беззнаковая однобайтовая величина | ub1 |
| SB2[3] | Знаковая двухбайтовая величина | sb2 |
| UB2[3] | Беззнаковая двухбайтовая величина | ub2 |
| SB4[3] | Знаковая четырехбайтовая величина | sb4 |
| UB4 | Беззнаковая четырехбайтовая величина | ub4 |
| FLOAT[1,2] | Обычно четырехбайтовая величина, используемая для хранения числа с плавающей точкой | float |
| DOUBLE[1,2] | Обычно четырехбайтовая величина, используемая для хранения числа с плавающей точкой | double |
| STRING[1] | Используется для хранения строк символов переменной длины с завершающими нулями | char* |
| RAW[1] | Используется для хранения байтовых строк переменной длины | unsigned char* |

**Таблица 20.2. Внешние типы данных(продолжение)**

| Внешний тип данных | Описание | Тип данных C |
|---|---|---|
| OCILOBLOCATOR[4] | Используется в функциях OCI для работы с объектами LOB базы данных | OCILobLocater* |

[1] Стандартный тип данных C.

[2] Размер значений этого типа может меняться в зависимости от операционной системы и компилятора. Поэтому данные этого типа не всегда переносимы.

[3] Тип данных Oracle, определяемый в файле заголовка oratypes.h. В каждой системе этот тип определяется таким образом, чтобы он соответствовал своему исходному определению, что обеспечивает его переносимость.

[4] Тип данных Oracle, определяемый в файле заголовка oci.h. Этот тип используется только в функциях OCI, которые работают с объектами LOB базы данных.

**Таблица 20.3. Соответствие типов данных PL/SQL и внешних типов данных**

| Тип данных PL/SQL | Вспомогательные внешние типы данных |
|---|---|
| CHAR, CHARACTER, LONG, ROWID, VARCHAR, VARCHAR2 | STRING* |
| BINARY_INTEGER, BOOLEAN, PLS_INTEGER | INT*, CHAR, UNSIGNED CHAR, SHORT, UNSIGNED SHORT, UNSIGNED INT, LONG, UNSIGNED LONG, SB1, UB1, SB2, UB2, SB4, UB4, SIZE_T |
| NATURAL, NATURALN, POSITIVE, POSITIVEN, SIGNTYPE | UNSIGNED INT*, CHAR, UNSIGNED CHAR, SHORT, UNSIGNED SHORT, INT, LONG, UNSIGNED LONG, SB1, UB1, SB2, UB2, SB4, UB4, SIZE_T |
| FLOAT, REAL | FLOAT* |
| DOUBLE PRECISION | DOUBLE* |
| LONG RAW, RAW | RAW* |
| BFILE, BLOB, CLOB | OCILOBLOCATOR* |

*Внешний тип данных по умолчанию.

Каждый элемент конструкции PARAMETERS представляет собой параметр внешней процедуры C. Для примера возьмем ту же процедуру **OutputString**. Конструкция PARAMETERS процедуры-оболочки выглядит следующим образом:

```
CREATE OR REPLACE PROCEDURE OutputString(
  p_Path IN VARCHAR2,
  p_Message IN VARCHAR2) AS EXTERNAL
  ...
  PARAMETERS (p_Path STRING,
              p_Message STRING);
```

а процедура C определяется так:

```
void OutputString(path, message)
char *path;
char *message; {
  ...
```

В этой конструкции указано, что **p_Path** и **p_Message** должны быть переданы в C в виде строк символов, завершающихся нулями, типа **char***. Поскольку внешним типом данных по умолчанию для VARCHAR2 является STRING, то в данном случае конструкцию PARAMETERS можно было бы и опустить.

В качестве альтернативы рассмотрим процедуру **OutputString2**, параметр которой имеет тип **int**. Этот параметр указывает, сколько раз нужно повторить передачу сообщения. Процедура C определяется следующим образом:

```
/* Этот пример является частью файла outstr.c. */
/* Записывает строку символов, которая содержится в сообщении, в файл,
   находящийся по указанному маршруту. Передача сообщения повторяется
   num_times раз. Если файл не существует, то он будет создан. */
void OutputString2(path, message, num_times)
char *path;
char *message;
int num_times; {

  FILE *file_handle;
  int counter;

  /* Откроем файл на запись. */
  file_handle = fopen(path, "w");

  for (counter = 0; counter < num_times; counter++)
      /* Выведем строку, завершив ее символом новой строки. */
      fprintf(file_handle, "%s\n", message);

  /* Закроем файл. */
  fclose(file_handle);
}
```

а процедура-оболочка:

```
-- Этот пример является частью файла outstr.sql.
CREATE OR REPLACE PROCEDURE OutputString2(
  p_Path IN VARCHAR2,
  p_Message IN VARCHAR2,
  p_NumLines IN BINARY_INTEGER) AS EXTERNAL

  LIBRARY stringlib
  NAME "OutputString2"
  PARAMETERS (p_Path STRING,
              p_Message STRING,
              p_NumLines INT);
```

Обратите внимание на то, что для **p_NumLines** в качестве внешнего типа данных указан тип INT.

## Виды параметров

Для параметров процедуры-оболочки (а следовательно, и внешней процедуры) можно задавать только вид, или режим, установленный в PL/SQL: IN, OUT или IN OUT. Если вид параметра OUT или IN OUT, то соответствующий параметр C должен передаваться по ссылке, а не по значению, вне зависимости от применяемого типа данных. Единственным исключением из этого правила является внешний тип данных STRING, значения которого передаются по ссылке; при этом вид соответствующего параметра значения не имеет. Проиллюстрируем вышесказанное на примере процедуры **OutputString3.** Обратите внимание: вид параметра **p_NumLinesWritten** – OUT, и поэтому соответствующий параметр C передается как **int***, а не как **int.** Ниже приведен программный текст процедуры-оболочки и внешней процедуры.

```
/* Этот пример является частью файла outstr.c. */
/* Записывает строку символов, которая содержится в сообщении, в файл,
   находящийся по указанному маршруту. Передача сообщения повторяется
   num_times раз. Если файл не существует, то он будет создан.
   Фактическое число записанных строк возвращается в num_lines_written. */
void OutputString3(path, message, num_times, num_lines_written)
char *path;
char *message;
```

```
int num_times;
ub2 *num_lines_written; {

  FILE *file_handle;
  int counter;
  /* Откроем файл на запись. */
  file_handle = fopen(path, "w");
  for (counter = 0; counter < num_times; counter++) {
      /* Выведем строку, завершив ее символом новой строки. */
      fprintf(file_handle, "%s\n", message);
              (*num_lines_written)++;
  }
  /* Закроем файл. */
  fclose(file_handle);
}
```

```
-- Этот пример является частью файла outstr.sql.
CREATE OR REPLACE PROCEDURE OutputString3(
  p_Path IN VARCHAR2,
  p_Message IN VARCHAR2,
  p_NumLines IN BINARY_INTEGER,
  p_NumLinesWritten OUT NATURAL) AS EXTERNAL

LIBRARY stringlib
NAME "OutputString3"
PARAMETERS (p_Path STRING,
            p_Message STRING,
            p_NumLines INT,
            p_NumLinesWritten UB2);
```

## Свойства параметров

Конструкция PARAMETERS используется для указания не только типа данных, но и дополнительной информации, называемой свойствами (properties), для каждого из параметров. Каждое свойство отображается на другой параметр во внешней процедуре, но не в процедуре-оболочке. Применяемые свойства описаны ниже, а используемые при этом типы данных – в таблице 20.4.

### Таблица 20.4. Свойства параметров

| Свойство | Вспомогательные внешние типы данных |
|---|---|
| INDICATOR | SHORT*, INT, LONG |
| LENGTH | INT*, SHORT, UNSIGNED SHORT, UNSIGNED INT, LONG, UNSIGNED LONG |
| MAXLEN | INT*, SHORT, UNSIGNED SHORT, UNSIGNED INT, LONG, UNSIGNED LONG |
| CHARSETID, CHARSETFORM | UNSIGNED INT*, UNSIGNED SHORT, UNSIGNED LONG |

*Внешний тип данных по умолчанию.

**INDICATOR** Индикатор используется в том случае, когда соответствующий параметр может являться NULL-значением. Переменные PL/SQL могут содержать NULL-значения, а переменные C – не могут. Именно поэтому необходим индикатор, поскольку внешней процедуре или самой реляционной СУБД, возможно, будет нужно знать о том, что возвращаемый параметр – это NULL-значение. Внешняя процедура может проверить значение индикатора – если этим значением является OCI_IND_NULL, то параметр – NULL, а если OCI_IND_NOTNULL – то параметр – не NULL. Эти значения можно устанавливать и для выходных параметров.

**LENGTH и MAXLEN** Свойство LENGTH используется для хранения текущего размера, а MAXLEN — для хранения максимального размера символьного параметра или параметра типа RAW. Если вид параметра OUT или IN OUT, то возвращаемое значение не должно превышать MAXLEN.

**CHARSETID и CHARSETFORM** Эти свойства используются для указания соответственно идентификатора и формы набора символов. Они применяются в средах NLS для параметров типов CHAR, CLOB и VARCHAR2. Эквивалентными атрибутами OCI являются OCI_ATTR_CHARSET_ID и OCI_ATTR_CHARSET_FORM.

Использование этих свойств иллюстрируется на примере процедуры **OutputString4.** Программный текст внешней процедуры и процедуры-оболочки приведен ниже. Обратите внимание на то, что для **num_lines_written** индикатор не нужен, так как этот параметр всегда будет возвращать значение, отличное от NULL.

```
/* Этот пример является частью файла outstr.c. */
/* Записывает строку символов, которая содержится в сообщении, в файл,
   находящийся по указанному маршруту. Передача сообщения повторяется
   num_times раз. Если файл не существует, то он будет создан.
   Фактическое число записанных строк возвращается в num_lines_written.
   Каждый параметр проверяется на наличие в нем NULL-значения. */
void OutputString4(path, path_ind,
                   message, message_ind,
                   num_times, num_times_ind,
                   num_lines_written)
char *path;
short path_ind;
char *message;
short message_ind;
int num_times;
short num_times_ind;
ub2 *num_lines_written; {

  FILE *file_handle;
  int counter;

  /* Если какие-либо из входных параметров содержат NULL-значения,
     возвращается ноль и нечего не выводится. */
  if (path_ind == OCI_IND_NULL || message_ind == OCI_IND_NULL ||
      num_times_ind == OCI_IND_NULL) {
    *num_lines_written = 0;
    return;
  }
  /* Откроем файл на запись. */
  file_handle = fopen(path, "w");
  for (counter = 0; counter < num_times; counter++) {
    /* Выведем строку, завершив ее символом новой строки. */
    fprintf(file_handle, "%s\n", message);
    (*num_lines_written)++;
  }
  /* Закроем файл. */
  fclose(file_handle);
}
```

```
-- Этот пример является частью файла outstr.sql.
CREATE OR REPLACE PROCEDURE OutputString4(
  p_Path IN VARCHAR2,
  p_Message IN VARCHAR2,
```

```
p_NumLines IN BINARY_INTEGER,
p_NumLinesWritten OUT NATURAL) AS EXTERNAL

LIBRARY stringlib
NAME "OutputString4"
PARAMETERS (p_Path STRING,
            p_Path INDICATOR,
            p_Message STRING,
            p_Message INDICATOR,
            p_NumLines INT,
            p_NumLines INDICATOR,
            p_NumLinesWritten UB2);
```

## Внешние функции и модульные процедуры

Помимо внешних процедур можно создавать и внешние функции. Внешние процедуры и функции могут включаться в состав программных модулей, что позволяет использовать преимущества модулей, например свойство переопределения.

### Значения, возвращаемые функциями

Внешние функции создаются точно так же, как и внешние процедуры. Значение, возвращаемое функцией, указывается с помощью ключевого слова RETURN в конструкции PARAMETERS. С возвращаемым значением могут быть связаны определенные свойства, что показано на примере функции **OutputString5**:

```
/* Этот пример является частью файла outstr.c. */
/* Записывает строку символов, которая содержится в сообщении, в
   файл, находящийся по указанному маршруту. Передача сообщения
   повторяется num_times раз. Если файл не существует, то он будет
   создан. Возвращается фактическое число записанных строк. Каждый
   параметр проверяется на наличие в нем NULL-значения. */
ub2 OutputString5(path, path_ind,
                  message, message_ind,
                  num_times, num_times_ind,
                  retval_ind)
char *path;
short path_ind;
char *message;
short message_ind;
int num_times;
short num_times_ind;
short *retval_ind; {

  FILE *file_handle;
  ub2 counter;
  /* Если какие-либо из входных параметров содержат NULL-значения,
     возвращается ноль и нечего не выводится. */
  If (path_ind == OCI_IND_NULL || message_ind == OCI_IND_NULL ||
      num_times_ind == OCI_IND_NULL) {
    *retval_ind = OCI_IND_NULL;
    return 0;
  }
  /* Откроем файл на запись. */
  file_handle = fopen(path, "w");

  for (counter = 0; counter < num_times; counter++) {
```

```
        /* Выведем строку, завершив ее символом новой строки. */
        fprintf(file_handle, "%s\n", message);
    }
    /* Закроем файл. */
    Fclose(file_handle);

    /* Установим возвращаемые значения. */
    *retval_ind = OCI_IND_NOTNULL;
    return counter;
}
```

```
-- Этот пример является частью файла outstr.sql.
CREATE OR REPLACE FUNCTION OutputString5(
  p_Path IN VARCHAR2,
  p_Message IN VARCHAR2,
  p_NumLines IN BINARY_INTEGER)
  RETURN NATURAL AS EXTERNAL

  LIBRARY stringlib
  NAME "OutputString5"
  PARAMETERS (p_Path STRING,
              p_Path INDICATOR,
              p_Message STRING,
              p_Message INDICATOR,
              p_NumLines INT,
              p_NumLines INDICATOR,
              RETURN INDICATOR,
              RETURN UB2);
```

## Переопределение

Процедуры-оболочки (или функции-оболочки) могут входить в состав модулей, а также храниться отдельно. В нашем случае процедуру-оболочку следует включить в тело модуля, предварительно объявив ее в заголовке, как и обычную процедуру. Этот способ проиллюстрирован на примере модуля **debug_extproc**, описанного ниже.

## RESTRICT_REFERENCES

Если внешняя функция находится в модуле, можно создать прагму RESTRICT_REFERENCES для того, чтобы использовать функцию в SQL-операторах. Однако компилятор PL/SQL не может проверить, нарушает ли внешняя функция ограничения, установленные прагмой, поскольку функция написана на языке, отличном от PL/SQL. Соблюдение этих ограничений является обязанностью пользователя. Если внешняя функция нарушает прагму, могут возникать внутрисистемные ошибки.

# Обратные вызовы базы данных

Для установления ошибок или выполнения SQL-команд во внешней процедуре могут производиться обратные вызовы базы данных. Это осуществляется при помощи интерфейса OCI Oracle8. Для выполнения всех SQL-операторов во внешней процедуре используются существующее соединение и транзакция, установленные в процедуре-оболочке PL/SQL.

## Служебные подпрограммы

Служебные подпрограммы используются для установления исключительных ситуации в базе данных, для выделения памяти и для считывания описателей OCI с целью выполнения SQL-операторов. Одним из параметров каждой служебной подпрограммы является контекст. В C контекст имеет тип **OCIExtProcContext** и обозначается ключевым словом CONTEXT в списке параметров. Кроме того, обязательна конструкция WITH CONTEXT.

## OCIExtProcRaiseExcp

Эта служебная подпрограмма устанавливает стандартную исключительную ситуацию, подобно тому как это делается с помощью оператора RAISE:

```
int OCIExtProcRaiseExcp(with_context, error_number)
OCIExtProcContext *with_context;
size_t error_number;
```

При вызове OCIExtProcRaiseExcp, как и при вызове RAISE, параметрам OUT и IN OUT значения не присваиваются, и процедура должна возвращаться немедленно. Использование этой служебной подпрограммы иллюстрируется на примере функции **OutputString6.** Здесь в случае передачи NULL-значения устанавливается исключительная ситуация ORA-6502. Обратите внимание на использование WITH CONTEXT в конструкции PARAMETERS.

```
/* Этот пример является частью файла outstr.c. */
/* Записывает строку символов, которая содержится в сообщении, в файл,
   находящийся по указанному маршруту. Передача сообщения повторяется
   num_times раз. Если файл не существует, то он будет создан.
   Возвращается фактическое число записанных строк. Каждый параметр
   проверяется на наличие в нем NULL-значения, и, если хотя бы один
   входной параметр содержит NULL-значение, устанавливается ORA-6502. */
ub2 OutputString6(context, path, path_ind,
                  message, message_ind,
                  num_times, num_times_ind,
                  retval_ind)
OCIExtProcContext  *context;
Char               *path;
Short              path_ind;
Char               *message;
Short              message_ind;
Int                num_times;
Short              num_times_ind;
Short              *retval_ind; {

  FILE *file_handle;
  ub2 counter;

  /* Если хотя бы один из входных параметров содержит NULL-значение,
     устанавливается ORA-6502, и функция возвращается немедленно. */
  if (path_ind == OCI_IND_NULL || message_ind == OCI_IND_NULL ||
     num_times_ind == OCI_IND_NULL) {
   OCIExtProcRaiseExcp(context, 6502);
   return 0;
  }

  /* Откроем файл на запись. */
  file_handle = fopen(path, "w");

  for (counter = 0; counter < num_times; counter++) {
     /* Выведем строку, завершив ее символом новой строки. */
     fprintf(file_handle, "%s\n", message);
  }

  /* Закроем файл. */
  Fclose(file_handle);
```

```
    /* Установим возвращаемые значения. */
    *retval_ind = OCI_IND_NOTNULL;
    return counter;
}
```

```
-- Этот пример является частью файла outstr.sql.
CREATE OR REPLACE FUNCTION OutputString6(
  p_Path IN VARCHAR2,
  p_Message IN VARCHAR2,
  p_NumLines IN BINARY_INTEGER)
  RETURN NATURAL AS EXTERNAL

  LIBRARY stringlib
  NAME "OutputString6"
  WITH CONTEXT
  PARAMETERS (CONTEXT,
              p_Path STRING,
              p_Path INDICATOR,
              p_Message STRING,
              p_Message INDICATOR,
              p_NumLines INT,
              p_NumLines INDICATOR,
              RETURN INDICATOR,
              RETURN UB2);
```

## OCIExtProcRaiseExcpWithMsg

Эта служебная подпрограмма аналогична оператору RAISE_APPLICATION_ERROR. В отличие от OCIExtProcRaiseExcp, она позволяет передавать сообщение пользователя вместе с ошибкой. Как и в случае RAISE_APPLICATION_ERROR, номер ошибки должен лежать в диапазоне от 20 000 до 20 999:

```
int OCIExtProcRaiseExcpWithMsg(
  with_context, error_number, error_message, len)
OCIExtProcContext  *with_context;
Size_t             error_number;
Text               *error_message;
Size_t             len;
```

Если **error_message** является строкой символов, завершенной нулем, то значение **len** должно быть равно 0. В противном случае значение **len** должно соответствовать длине строки. Использование этой служебной подпрограммы иллюстрируется в следующем разделе на примере функции **OutputString7**.

## OCIExtProcAllocCallMemory

Если во внешней процедуре необходимо выделить память, то можно воспользоваться подпрограммой OCIExtProcAllocCallMemory. Выделение памяти сохраняется на протяжении всего вызова; память автоматически освобождается системой PL/SQL после завершения работы внешней процедуры:

```
dvoid *OCIExtProcAllocCallMemory(with_context, amount);
OCIExtProcContext *with_memory;
size_t amount;
```

где *amount* — число выделяемых байтов. В *OutputString7* эта подпрограмма используется для выделения памяти под текст сообщения об ошибке.

```
/* Этот пример является частью файла outstr.c. */
/* Записывает строку символов, которая содержится в сообщении, в файл,
   находящийся по указанному маршруту. Передача сообщения повторяется
   num_times раз. Если файл не существует, то он будет создан.
   Возвращается фактическое число записанных строк. Каждый параметр
```

```
   проверяется на наличие в нем NULL-значения, и, если хотя бы один
   входной параметр содержит NULL-значение, устанавливается ORA-6502. Если
   файл невозможно открыть, возвращается ошибка, определенная пользователем. */
ub2 OutputString7(context, path, path_ind,
                  message, message_ind,
                  num_times, num_times_ind,
                  retval_ind)
OCIExtProcContext  *context;
Char               *path;
Short              path_ind;
Char               *message;
Short              message_ind;
Int                num_times;
Short              num_times_ind;
Short              *retval_ind; {

  FILE *file_handle;
  ub2 counter;

  /* Если хотя бы один из входных параметров содержит NULL-значение,
     устанавливается ORA-6502, и функция возвращается немедленно. */
  if (path_ind == OCI_IND_NULL || message_ind == OCI_IND_NULL ||
      num_times_ind == OCI_IND_NULL) {
    OCIExtProcRaiseExcp(context, 6502);
   return 0;
  }

  /* Откроем файл на запись. */
  file_handle = fopen(path, "w");

  /* Проверим успешность выполнения функции. Если функция выполнена
     неуспешно, установим ошибку. */
  if (!file_handle) {
     text *initial_msg = (text *)"Cannot open file ";
     text *error_msg;

  /* Выделим и установим память для текста сообщения об ошибке.
     Освобождать эту память не нужно — PL/SQL сделает это автоматически. */
  error_msg = OCIExtProcAllocCallMemory(context,
                  strlen(path) + strlen(initial_msg) + 1);
  strcpy((char *)error_msg, (char *)initial_msg);
  strcat((char *)error_msg, path);
  OCIExtProcRaiseExcpWithMsg(context, 20001, error_msg, 0);
  return 0;
}
for (counter = 0; counter < num_times; counter++) {
  /* Выведем строку, завершив ее символом новой строки. */
  fprintf(file_handle, "%s\n", message);
}
/* Закроем файл. */
Fclose(file_handle);

/* Установим возвращаемые значения. */
```

```
    *retval_ind = OCI_IND_NOTNULL;
  return counter;
}
```

## Выполнение SQL-операторов во внешней процедуре

Для выполнения SQL-операторов во внешней процедуре применяется интерфейс Oracle8 OCI. Для работы с OCI необходимо установить ряд описателей OCI. Это можно сделать с помощью подпрограммы OCIExtProcGetEnv.

### OCIExtProcGetEnv

Описание этой служебной подпрограммы выглядит следующим образом:

```
sword OCIExtProcGetEnv(with_context, envh, svch, errh)
OCIExtProcContext    *with_context;
OCIEnv              **envh;
OCISvcCtx           **svch;
OCIError            **errh;
```

Среду возвращения значений (envh), контекст сервиса (svch) и описатели ошибок (errh) можно указывать в последующих вызовах OCI, выполняющих SQL-операторы. Эти описатели можно использовать только в обратных вызовах – в стандартных вызовах OCI их использовать нельзя. Более подробно об описателях OCI и об их применении рассказано в руководстве Programmer's Guide to the Oracle Call Interface.

### Ограничения

При задании SQL-операторов во внешних процедурах необходимо соблюдать ряд условий. Во внешних процедурах нельзя:

- Использовать команды управления транзакциями, например COMMIT или ROLLBACK
- Использовать команды DDL, вызывающие неявное выполнение оператора COMMIT
- Вызывать другие внешние процедуры

Кроме того, во внешних процедурах запрещены некоторые вызовы OCI. Полный перечень таких вызовов OCI приведен в руководстве Oracle8 PL/SQL User's Guide and Reference.

# Советы, рекомендации и ограничения

В последней части этой главы предлагаются некоторые рекомендации по отладке и анализу внешних процедур, а также ограничения на их использование.

## Отладка внешних процедур

Внешние процедуры являются достаточно полезным и эффективным средством. Однако они взаимодействуют с двумя различными языками программирования (PL/SQL и C) и с интерфейсом связи этих языков, поэтому отладка внешних процедур является довольно сложным процессом. Далее рассмотрен ряд методов отладки внешних процедур.

### Прямой вызов из C

Пожалуй, самый простой способ отладки – это вызов внешней процедуры непосредственно из C, без участия PL/SQL. Вызвать процедуру можно или из статически, или из динамически скомпонованной программы, а затем для отладки процедуры воспользоваться стандартной программой-отладчиком C, применяемой в конкретной системе (или другими средствами). Хотя такой способ не поможет устранить проблему, связанную с интерфейсом PL/SQL и C, он полезен при поиске ошибок непосредственно в тексте внешней процедуры. Данный метод можно применять только в том случае, если во внешней процедуре не производится обратных вызовов базы данных.

### Соединение с процессом extproc с помощью отладчика

Для того чтобы с помощью отладчика выполнить процедуру C в пошаговом режиме, практически всегда нужно запускать программу под управлением этого отладчика. Однако для внешних процедур прослушивающий процесс SQL*Net автоматически запускает процесс extproc, что не позволяет запускать программу под управлением отладчика. Решением является предварительный запуск процесса extproc

(посредством вызова фиктивной внешней процедуры) и последующее соединение с функционирующим процессом через отладчик.

**▼ ВНИМАНИЕ** Для использования этого метода операционная система должна поддерживать работу отладчика, позволяющего соединяться с функционирующим процессом. Во многих системах Unix, например, можно применять отладчик gdb.

Для упрощения этого процесса в Oracle существует модуль, называемый debug_extproc. Сценарий создания этого модуля можно найти в демонстрационном каталоге PL/SQL в $ORACLE_HOME, а также на компакт-диске, прилагаемом к данной книге. Ниже приведен текст модуля debug_extproc.

```
-- Этот пример содержится в файле dbgextp.sql.
CREATE OR REPLACE PACKAGE debug_extproc IS
  -- Запуск агента-процесса extproc в данном сеансе. При выполнении
  -- этой процедуры запускается агент-процесс extproc, который позволяет
  -- узнать идентификатор выполняющегося процесса. Этот идентификатор
  -- нужен для соединения с функционирующим процессом через отладчик.
  PROCEDURE startup_extproc_agent;
END debug_extproc;

CREATE OR REPLACE LIBRARY debug_extproc_library IS STATIC;

CREATE OR REPLACE PACKAGE BODY debug_extproc IS
  extproc_lib_error EXCEPTION;
  PRAGMA EXCEPTION_INIT (extproc_lib_error, -6520);

  extproc_func_error EXCEPTION;
  PRAGMA EXCEPTION_INIT (extproc_func_error, -6521);

  PROCEDURE local_startup_extproc_agent IS EXTERNAL
      LIBRARY debug_extproc_library;

  PROCEDURE startup_extproc_agent is
  BEGIN
      -- Вызовем фиктивную процедуру, фиксируя все ошибки.
      local_startup_extproc_agent;
  EXCEPTION
      -- Если функция или библиотека не найдена, ошибки игнорируются.
      WHEN extproc_func_error then NULL;
      WHEN extproc_lib_error then NULL;
  END startup_extproc_agent;
END debug_extproc;
```

Ниже приведена последовательность работы с модулем debug_extproc.

1. Убедиться в том, что программа C скомпилирована и скомпонована в виде разделяемой библиотеки с символами отладки. В системах Unix это обычно выполняется путем запуска компилятора с флагом -g.
2. В SQL*Plus или в другом соединении с базой данных вызвать процедуру **debug_extproc.startup_extproc_agent**. Хотя с этой процедурой-оболочкой не связана какая-либо реальная процедура C, процесс extproc запускается. В **startup_extproc_agent** имеются обработчики исключительных ситуаций, поэтому сообщений об ошибках не поступает.
3. Определить идентификатор процесса extproc, устанавливаемый операционной системой. В Unix это выполняется с помощью команды ps.
4. Запустить отладчик и соединиться с функционирующим процессом, определенным на шаге 3.
5. Установить в функции точку прерывания extproc и продолжить работу отладчика.

6. В исходном сеансе вызвать процедуру-оболочку. При этом будет вызван процесс extproc, который затем остановится в точке pextproc.
7. В этой точке внешняя процедура обнаружена процессом extproc, поэтому можно установить для нее точку прерывания. После динамической загрузки символа некоторые отладчики не распознают его сразу же, поэтому может потребоваться загрузить этот символ повторно (в отладчике gdb это выполняется с помощью команды share).
8. Продолжить работу отладчика, который затем остановится в соответствующей процедуре. После этого можно выполнять процедуру в пошаговом режиме, осуществляя ее отладку.

## Рекомендации

При работе с внешними процедурами необходимо учитывать ряд моментов:

- Вызов внешней процедуры приводит к расходу ресурсов, особенно при первом вызове за время работы сеанса, так как прослушивающий процесс должен обработать запрос и породить процесс extproc. Поэтому применяйте внешние процедуры только в тех ситуациях, когда выгоднее затратить больший объем ресурсов, необходимых на порождение нового процесса, чем лишиться преимуществ, предоставляемых языком C.
- Процессу extproc предоставляются те же привилегии на работу с операционной системой, которые имеет порождающий его прослушивающий процесс. Поэтому для внешних процедур рекомендуется запускать собственный прослушивающий процесс, который отличен от пользовательского и которому предоставлены ограниченные права. Если же прослушивающий процесс будет функционировать в качестве процесса пользователя Oracle, то любой пользователь, обладающий полномочиями на создание библиотек, сможет написать на C программу, модифицирующую файлы в каталоге $ORACLE_HOME, что, вполне возможно, приведет к повреждению базы данных.
- Обязательно записывайте информацию во все параметры OUT и IN OUT, а также в значения, возвращаемые функциями. В PL/SQL это не проверяется, и, если таким параметрам не присвоить значения, это может привести к внутренним ошибкам системы.
- Аналогично, не записывайте информацию в параметры IN и не превышайте размеры (указанные свойством MAXLEN) параметров OUT.
- При обработке всех символьных параметров используйте свойства LENGTH и INDICATOR. Если возвращается NULL-значение, устанавливайте параметр LENGTH в ноль.
- Если для функции-оболочки указан оператор PRAGMA RESTRICT_REFERENCES, то внешняя процедура не должна нарушать эту прагму. В PL/SQL это не проверяется, и, если внешняя функция модифицирует базу данных в нарушение прагмы, это может привести к внутренним ошибкам системы.

## Ограничения

В настоящее время (в Oracle8 версии 8.0) на использование внешних процедур налагается ряд ограничений:

- Если в базовой системе не поддерживаются динамическая компоновка и разделяемые библиотеки, то внешние процедуры применять нельзя.
- Прослушивающий процесс SQL*Net и процесс extproc должны выполняться на той же машине, что и база данных. Аналогично, нельзя использовать в обороте LIBRARY связь баз данных для указания библиотеки, находящейся в удаленной базе данных.
- В настоящее время для внешних процедур используется только один язык программирования — C. Если необходимо вызвать процедуру, написанную на другом языке, то нужно написать процедуру, внешнюю по отношению к C, для которой C будет выступать в роли посредника. После этого можно вызывать в PL/SQL внешнюю процедуру C, которая будет, в свою очередь, вызывать процедуру, написанную на другом языке. Последняя процедура должна быть доступна из разделяемой библиотеки.
- Параметры внешних процедур не могут быть курсорными переменными, записями, сборными конструкциями или экземплярами объектных типов PL/SQL. Однако эти объекты могут считываться из внешних процедур через обратные вызовы.

- Oracle-сервер не может работать в режиме MTS (многопоточного сервера). Процесс extproc должен взаимодействовать с процессом выделенного сервера.
- Максимальное число параметров, передаваемых процедуре C, равно 128. В число этих параметров входят параметры, используемые для считывания свойств, таких, как INDICATOR или LENGTH. Однако, если параметры типа float или double передаются по ссылке, то их максимальное число меньше 128. Для каждого параметра, передаваемого таким образом, требуется приблизительно столько памяти, сколько необходимо для двух параметров, передаваемых по значению.

# Итоги

В этой главе были рассмотрены внешние процедуры, которые позволяют вызывать из программ PL/SQL процедуры C. Обсуждалось отображение типов данных PL/SQL на типы данных C, а также использование свойств параметров. Кроме того, был дан анализ ряда методов отладки внешних процедур. В следующей главе будет рассмотрено еще одно новое средство Oracle8 – большие объекты (LOB).

# Глава 21

## *Большие объекты*

Большие объекты (LOB – Large OBjects) полезны в любой реляционной базе данных. Они используются для хранения больших объемов информации, будь то текстовые или двоичные данные. В этой главе рассказывается о программных средствах, применяемых в Oracle8 для работы с объектами LOB, в том числе о способах хранения таких объектов в базе данных и вне ее, о различных типах больших объектов и об интерфейсе PL/SQL, предназначенном для работы с объектами LOB, – модуле DBMS_LOB.

# Понятие объекта LOB

LOB – это всего лишь поле базы данных, в котором сохраняется большой объем данных, например графический файл или объемный текстовый документ. Какие типы данных Oracle более всего подходят для хранения информации такого вида? В Oracle7 в поле типа VARCHAR2 можно хранить до 2000 байтов. В поле столбца LONG или LONG RAW можно записать до 2 гигабайтов, однако для данных типа LONG установлено множество ограничений, например возможность создания в таблице базы данных только одного столбца LONG. Кроме того, с фрагментами данных LONG или LONG RAW можно работать только с помощью интерфейса OCI (Oracle Call Interface), применяемого в Oracle 7.3.

**PL/SQL 8.0 ... и ВЫШЕ**

В Oracle8 этой проблемы не существует. Столбцы LONG и LONG RAW по-прежнему доступны (с теми же самыми ограничениями), однако теперь можно использовать типы данных нового семейства – семейства LOB. Существует четыре вида объектов LOB, созданных для различных видов данных: CLOB, NCLOB, BLOB и BFILE. Эти типы данных описаны в таблице 21.1.

**Таблица 21.1. Типы LOB**

| Тип LOB | Описание |
|---|---|
| CLOB | Подобно типу LONG в Oracle7, тип CLOB дает возможность хранить однобайтовые символьные данные |
| NCLOB | В объектах NCLOB хранятся многобайтовые данные фиксированного размера, представляющие национальные наборы символов |
| BLOB | Подобно типу LONG RAW в Oracle7, тип BLOB дает возможность хранить неструктурированные двоичные данные. Oracle8 не воспринимает эти данные, но может хранить и считывать их фрагментами |
| BFILE | Объекты BFILE позволяют обращаться в режиме "только чтение" к большим двоичным файлам, находящимся вне базы данных Oracle. В отличие от трех других типов LOB, данные типа BFILE хранятся в отдельных файлах, которые не обслуживаются Oracle. Следовательно, операции над объектами BFILE не входят в состав транзакций Oracle; они выполняются асинхронно и не могут быть завершены (COMMIT) или отменены (ROLLBACK) |

Объекты CLOB, NCLOB и BLOB в совокупности называются *внутренними объектами LOB* (internal LOB), а объекты BFILE – *внешними объектами LOB* (external LOB). Объекты LOB имеют следующие характеристики:

- Максимальный размер объекта LOB составляет 4 гигабайта, то есть ограничение в 2 гигабайта, установленное в Oracle (для данных LONG и LONG RAW), отменяется.
- Управлять объектами LOB можно либо через интерфейс вызовов Oracle8 (OCI), либо через PL/SQL с помощью модуля DBMS_LOB. Оба этих интерфейса обеспечивают случайный доступ к LOB как для чтения, так и для записи (исключая объекты BFILE, которые доступны только для чтения).
- Ограничения, налагаемые на данные LONG и LONG RAW, для объектов LOB недействительны. Например, в отличие от столбцов LONG и LONG RAW, в таблице базы данных можно создавать неограниченное число столбцов LOB (вплоть до максимального числа столбцов таблицы), и данными LOB можно управлять при помощи триггеров.
- Объекты могут иметь атрибуты типа LOB, а методы могут принимать LOB-аргументы. Однако атрибут объекта не может иметь тип NCLOB, хотя объект NCLOB может быть указан в качестве аргумента метода.
- Объекты LOB можно использовать в качестве переменных привязки.
- Работа с внутренними объектами LOB осуществляется либо с помощью SQL-операторов DML, либо посредством модуля DBMS_LOB. В любом случае обеспечивается полная поддержка функционирования транзакций и согласованность чтения информации, как и при работе с данными, не являющимися объектами LOB.

## Хранение объектов LOB

В отличие от данных LONG и LONG RAW, данные LOB не хранятся в таблицах базы данных. Они содержатся в отдельном месте, а в исходной таблице находится только *локатор LOB* (LOB locator), который указывает на фактические данные. При необходимости такая информация может содержаться в собственной табличной области, параметры хранения которой отличаются от параметров таблицы. Место хранения данных LOB можно указать в операторе CREATE TABLE с помощью специальной конструкции, называемой *хранением LOB* (LOB storage clause). В такой конструкции можно указать табличную область и/или параметры хранения, отличные от табличной области и параметров самой таблицы, что позволяет достаточно гибко и эффективно управлять данными LOB. Хранение LOB определяется следующим образом:

LOB(*столбец_LOB, столбец_LOB,* ...) STORE AS
[*имя_сегмента_LOB*][ *хранение*]

где *столбец_LOB* — это столбец таблицы, имеющий тип LOB, *имя_сегмента_LOB* — имя сегмента LOB в нужной табличной области, а *хранение* определяет табличную область и другие параметры. Более подробно о хранении LOB рассказано в руководствах Oracle8 Application Developer's Guide и Oracle8 Server SQL Reference.

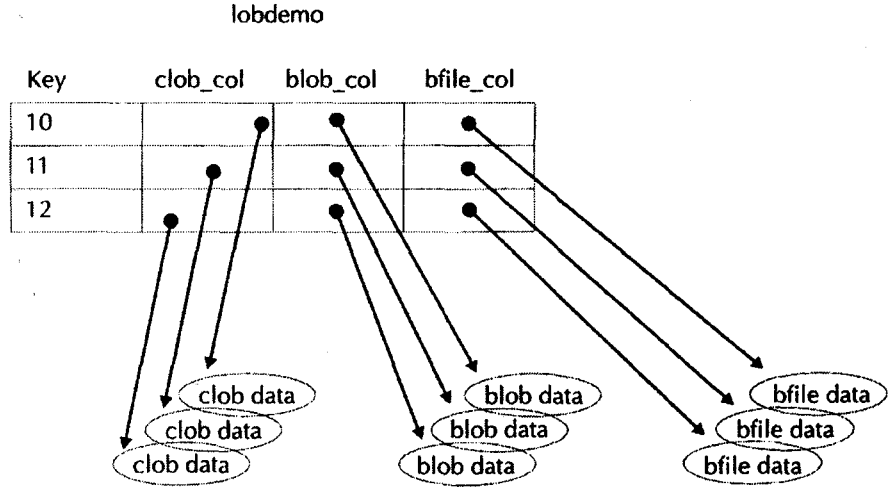
В качестве примера рассмотрим таблицу **lobdemo**, создаваемую с помощью следующего оператора:

```
-- Этот пример является частью файла tables8.sql.
CREATE TABLE lobdemo (
  key NUMBER PRIMARY KEY,
  clob_col CLOB,
  blob_col BLOB,
  bfile_col BFILE
);
```

Реальные данные для каждого из столбцов LOB хранятся отдельно от остальных данных таблицы **lobdemo**, что проиллюстрировано на рисунке 21.1.

**▼ ВНИМАНИЕ** Объекты BFILE, как и объекты CLOB, BLOB и NCLOB, управляются при помощи локаторов LOB. Однако данные типа BFILE хранятся в файлах, которые не управляются Oracle. Более подробно об объектах BFILE рассказано в разделе "Работа с объектами BFILE" далее в этой главе.

**Рис. 21.1.**
*Хранение LOB в базе данных*



## Объекты LOB в DML

Объекты LOB как единое целое управляются операторами DML. К примеру, можно обновить (UPDATE) LOB новым значением. Фрагментами объектов LOB можно манипулировать с помощью модуля DBMS_LOB, подробно описанного ниже, в разделе "Подпрограммы DBMS_LOB". Во всех подпрограммах DBMS_LOB для работы с данными используются локаторы LOB.

### Инициализация столбца LOB

Значение поля столбца LOB можно установить в NULL, что продемонстрировано в следующем операторе INSERT:

```
INSERT INTO lobdemo(key, clob_col, blob_col, bfile_col)
  VALUES (10, NULL, NULL, NULL);
```

Здесь в столбец вводится не локатор LOB, а NULL-значение. При этом реальное место для хранения данных LOB не выделяется, поскольку на них не указывает какой-либо локатор. Поэтому для NULL-значений модуль DBMS_LOB использовать нельзя. Предварительно в строку необходимо ввести правильный локатор.

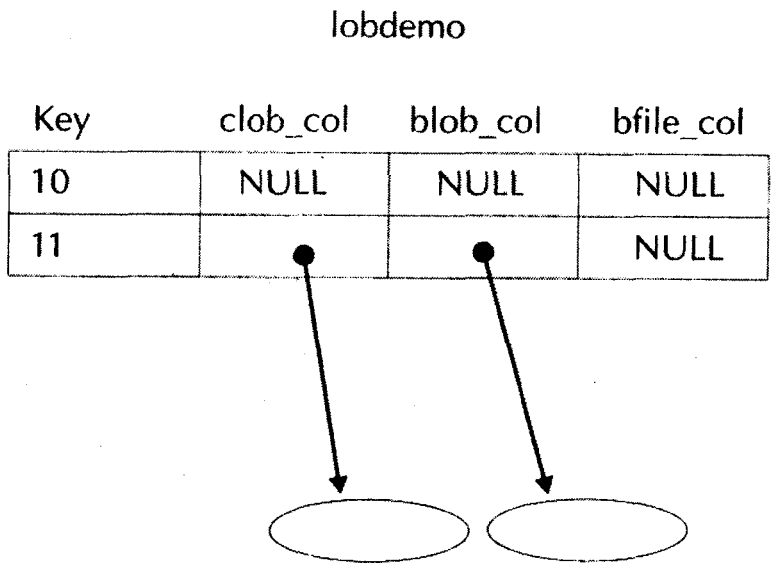
Одним из способов введения локатора является использование функций EMPTY_BLOB() и EMPTY_CLOB(). Эти функции не имеют каких-либо аргументов, а возвращают правильный локатор LOB. После этих функций можно воспользоваться модулем DBMS_LOB и ввести данные LOB. Для примера рассмотрим такой оператор INSERT:

```
INSERT INTO lobdemo(key, clob_col, blob_col, bfile_col)
  VALUES (11, empty_clob(), empty_blob(), NULL);
```

**▼ ВНИМАНИЕ** Для функций EMPTY_BLOB() и EMPTY_CLOB() пустые круглые скобки обязательны.

После выполнения примеров, рассмотренных выше, таблица **lobdemo** будет выглядеть так, как показано на рисунке 21.2. После инициализации столбцов можно считывать (SELECT) локаторы LOB в переменные PL/SQL и работать с ними в DBMS_LOB. Другим способом выбора локатора является использование конструкции RETURNING оператора INSERT.

**Рис. 21.2.** *Таблица lobdemo после ввода данных*




Объекты BFILE не инициализируются функциями EMPTY_BLOB() и EMPTY_CLOB(); вместо этих функций нужно применять функцию BFILENAME (раздел "Работа с объектами BFILE" этой главы).

### Пример

В следующем далее примере демонстрируется использование операторов DML для управления объектами LOB. Обратите внимание на то, что без DBMS_LOB можно ссылаться только на весь объект LOB.

Кроме того, заметьте, что с помощью оператора SELECT из столбца LOB будет выбираться локатор, а не сами данные.

```
-- Этот пример содержится в файле lobdml.sql.
DECLARE
  v_CLOBlocator CLOB;
  v_BLOBlocator BLOB;
BEGIN
  -- Инициализирует clob_col указанной строкой символов и возвращает
  -- локатор в v_LOBlocator.
  INSERT INTO lobdemo (key, clob_col)
      VALUES (20, 'abcdefghijklmnopqrstuvwxyz')
      RETURNING clob_col INTO v_CLOBlocator;

  -- Модифицирует значение blob_col, содержащееся в той же самой строке.
  UPDATE lobdemo
      SET blob_col = HEXTORAW('00FF00FF00FF')
      WHERE key = 20;

  -- Считывает локатор для обновленного значения, а не само значение.
  SELECT blob_col
      INTO v_BLOBlocator
      FROM lobdemo
      WHERE key = 20;
END;
```

# Работа с объектами BFILE

Основной характеристикой объектов BFILE является то, что они хранятся вне базы данных. Для управления файлами, как и другими LOB, можно использовать локаторы. Однако относительно объектов BFILE следует сделать ряд замечаний:

- Файлы находятся вне Oracle, поэтому для операций, выполняемых над объектами BFILE, не производится управление транзакциями. Oracle не обеспечивает целостности и долговечности внешних файлов, так что эти функции должна брать на себя соответствующая операционная система.
- Объекты BFILE доступны только для чтения. Как с помощью модуля DBMS_LOB, так и через OCI Oracle8 можно считывать информацию этих файлов, но модифицировать их нельзя.
- Для объектов BFILE не поддерживается средство переноса сеансов Oracle8 версии 8.0 в режиме MTS (многопоточного сервера). Следовательно, объекты BFILE перестают быть открытыми после окончания вызова сервера MTS. Для их обработки сеанс должен быть связан с одним разделяемым сервером.

## Каталоги

Для обращения к внешнему файлу сервер Oracle должен знать местонахождение этого файла в операционной системе. Местонахождение файла можно определить при помощи нового типа объекта словаря данных, называемого каталогом. *Каталог* (directory) — это логический псевдоним реального маршрута операционной системы. Каталоги создаются с помощью оператора CREATE DIRECTORY:

CREATE DIRECTORY *имя_каталога* AS *маршрут_ос*;

где *имя_каталога* — это имя нового каталога, а *маршрут_ос* — полный маршрут к каталогу операционной системы. Например, с помощью следующего оператора создается каталог, который указывает на каталог операционной системы /home/utils:

```
CREATE DIRECTORY utils AS '/home/utils';
```

### Привилегии, необходимые для работы с каталогами

Для того чтобы выполнить команду CREATE DIRECTORY, пользователь должен иметь системную привилегию CREATE DIRECTORY. Эта привилегия по умолчанию предоставляется роли DBA и может быть предоставлена другим пользователям администратором базы данных. После создания каталога можно предоставить привилегию READ на этот каталог другому пользователю. Наличие этой привилегии проверяется как в DBMS_LOB, так и в OCI.

Кроме того, указанный каталог операционной системы должен быть предварительно создан и быть доступен для чтения пользователем Oracle. За соблюдение всех этих условий несет ответственность системный администратор.

### Каталоги в словаре данных

В представлениях словаря данных **dba_directories** и **all_directories** описаны каталоги, которые созданы в системе. В **dba_directories** находится информация обо всех каталогах базы данных, а в *all_directories* – информация обо всех каталогах, которые доступны текущему пользователю. Структура этих представлений приведена в таблице 21.2.

**Таблица 21.2.**

| Столбец | Тип данных | Описание |
|---|---|---|
| OWNER | VARCHAR2(30) | Схема базы данных, владеющая каталогом |
| DIRECTORY_NAME | VARCHAR2(30) | Имя каталога |
| DIRECTORY_PATH | VARCHAR2(4000) | Путь к этому каталогу в операционной системе |

### Удаление каталогов

Каталоги удаляются с помощью команды DROP DIRECTORY:

```
DROP DIRECTORY имя_каталога;
```

где *имя_каталога* – удаляемый каталог. Для использования этой команды необходимо иметь системную привилегию DROP ANY DIRECTORY.

При удалении каталогов следует соблюдать осторожность. Если во время удаления каталога сеанс считывает информацию из файла, указанного этим каталогом, то операции, выполняемыми данным сеансом над LOB, окончатся неудачей. Более того, ресурсы, расходуемые на выполнение этих операций, не освободятся до тех пор, пока сеанс не будет уничтожен или пока не будет использована процедура DBMS_LOB.FILECLOSEALL (описанная в разделе "Модуль DBMS_LOB" далее в этой главе).

## Открытие и закрытие объектов BFILE

Операции, выполняемые над объектами BFILE, очень похожи на операции модуля UTL_FILE (главу 18). В модуле DBMS_LOB содержатся подпрограммы для открытия и закрытия объектов BFILE, а также для чтения содержащейся в них информации. Как и в UTL_FILE, следует закрывать все объекты BFILE, открытые в процедуре, даже тогда, когда устанавливается исключительная ситуация.

Кроме того, существует ограничение на число объектов BFILE, которые могут быть открыты в одном сеансе. Это ограничение указывается параметром SESSION_MAX_OPEN_FILES в файле init.ora. Значением по умолчанию является 10, то есть сеанс не может одновременно работать с более чем 10 открытыми файлами. Можно увеличить это число, если изменить значение параметра SESSION_MAX_OPEN_FILES.

## Объекты BFILE в DML

С объектами BFILE, как и с внутренними объектами LOB, можно работать в операторах DML. Однако для выполнения важных операций (например для считывания фрагментов данных из файла) необходимо использовать модуль DBMS_LOB или интерфейс OCI.

### Инициализация столбцов BFILE

Поля столбцов BFILE можно инициализировать NULL-значениями. Однако для того чтобы инициализировать их правильными локаторами, нужно использовать функцию BFILENAME:

```
FUNCTION BFILENAME(псевдоним_каталога IN VARCHAR2,
                   имя_файла IN VARCHAR2)
  RETURN BFILE;
```

где *псевдоним_каталога* – это имя ранее созданного объекта каталога, а *имя_файла* – файл операционной системы. Функцию BFILENAME можно использовать в операторах INSERT и UPDATE, а также для инициализации переменных PL/SQL типа BFILE.

Функция BFILENAME не проверяет, существует ли на самом деле каталог и/или файл, – эта проверка выполняется при обращении к локатору из DBMS_LOB. Например, с помощью следующего оператора INSERT создается локатор LOB, указывающий на файл /home/utils/file1:

```
INSERT INTO lobdemo(key, bfile_col)
  VALUES (-1, BFILENAME('utils', 'file1'));
```

## Сравнение семантики копирования и ссылочной семантики

Создадим вторую таблицу, имеющую ту же структуру, что и таблица **lobdemo**:

```
-- Этот пример является частью файла tables8.sql.
CREATE TABLE lobdemo2 (
  key NUMBER PRIMARY KEY,
  clob_col CLOB,
  blob_col BLOB,
  bfile_col BFILE
);
```

и выполним следующий SQL-оператор:

```
INSERT INTO lobdemo2
  SELECT key, clob_col, blob_col, NULL FROM lobdemo;
```

С помощью этого оператора и локаторы, и данные будут скопированы из **lobdemo** в **lobdemo2.** Семантика такой операции называется *семантикой копирования* (copy semantics), поскольку после выполнения операции будет существовать два набора данных. Семантика копирования применима для операций DML, выполняемых над внутренними объектами LOB, – в операциях DML над внешними объектами LOB используется *ссылочная семантика* (reference semantics). В этом случае копируется только локатор. Для примера рассмотрим такой блок LOB PL/SQL:

```
-- Этот пример содержится в файле bfiledml.sql.
BEGIN
  -- Создадим строку в lobdemo, указывающую на внешний файл.
  INSERT INTO lobdemo (key, bfile_col)
      VALUES (5, BFILENAME('util', 'file1'));
  -- Скопируем этот локатор в lobdemo2.
  INSERT INTO lobdemo2
      SELECT *
        FROM lobdemo
        WHERE key = 5;
END;
```

После выполнения этого блока в обеих таблицах, **lobdemo** и **lobdemo2**, будет находиться по строке, указывающей на один и тот же файл операционной системы. Такая ситуация проиллюстрирована на рисунке 21.3.

## Удаление локаторов объектов BFILE

При удалении строки, содержащей локатор BFILE, удаляется только локатор; сам же файл остается и, если на него указывают другие локаторы, может быть доступен для выполнения различных операций. Если после выполнения блока, рассмотренного выше, выполнить следующий оператор DELETE:

```
DELETE FROM lobdemo
  WHERE key = 5;
```

то к файлу file1 можно по-прежнему обращаться с помощью локатора, находящегося в таблице **lobdemo2.**

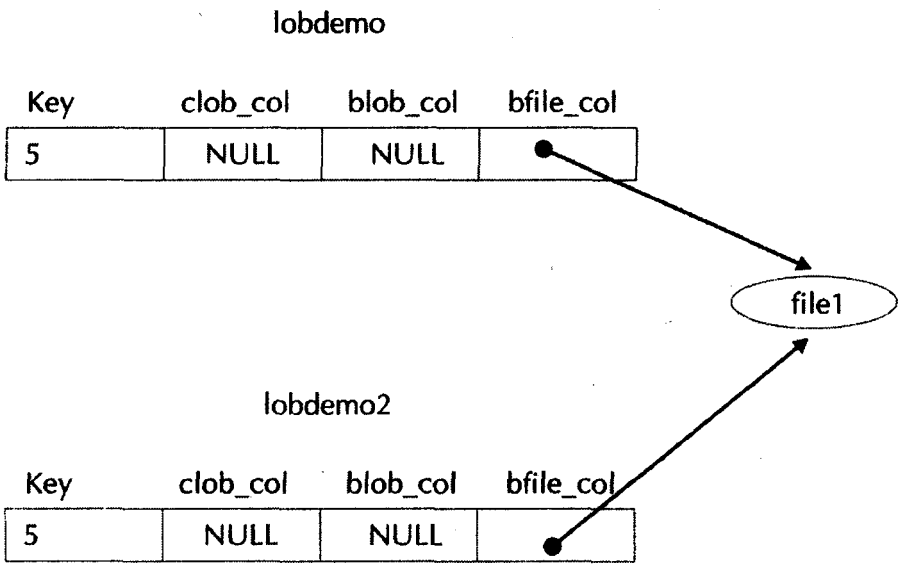



**Рис. 21.3.** *Таблицы lobdemo и lobdemo2*

# Модуль DBMS_LOB

С помощью SQL можно работать только с целыми объектами LOB, фрагментами же данных LOB манипулировать нельзя. Модуль DBMS_LOB устраняет это ограничение, позволяя выполнять операции над фрагментами объектов LOB (считывание и запись для внутренних LOB и только считывание для внешних LOB). В этом разделе модуль DBMS_LOB подробно описан; кроме того, здесь рассказывается о взаимодействии операций DBMS_LOB и операций DML.

## Подпрограммы DBMS_LOB

Для чтения значений LOB и для получения информации о значениях LOB предназначено пять подпрограмм модуля DBMS_LOB, причем ни одна из этих подпрограмм не модифицирует считываемые значения:

COMPARE
GETLENGTH
INSTR
READ
SUBSTR

Кроме того, в DBMS_LOB имеется шесть подпрограмм для записи значений LOB:

APPEND
COPY
ERASE
LOADFROMFILE
TRIM
WRITE

И наконец, следующие шесть подпрограмм можно использовать для работы с объектами BFILE:

FILECLOSE
FILECLOSEALL
FILEEXISTS
FILEGETNAME
FILEISOPEN
FILEOPEN

Все подпрограммы DBMS_LOB описаны ниже в алфавитном порядке. За описанием подпрограмм следует перечень исключительных ситуаций, устанавливаемых ими. Во всех приводимых примерах предполагается, что таблица **lobdemo** уже создана и заполнена с помощью следующего сценария:

```
-- Этот пример содержится в файле lobtest.sql.
INSERT INTO lobdemo (key, clob_col, blob_col, bfile_col)
  VALUES (-1, EMPTY_CLOB(), EMPTY_BLOB(), NULL);
INSERT INTO lobdemo (key, clob_col, blob_col, bfile_col)
  VALUES (1, 'abcdefghijklmnopqrstuvwxyz',
             HEXTORAW('000102030405060708090A0B0C0D0E0F'),
             NULL);
INSERT INTO lobdemo (key, clob_col, blob_col, bfile_col)
  VALUES (2, 'A Quick Brown Fox Jumps Over the Lazy Dog',
             HEXTORAW('FFFEFDFCFBFAF9F8F7F6F5F4F3F2F1F0'),
             NULL);
COMMIT;
```

Более подробно о модуле DBMS_LOB рассказано в руководстве Oracle8 Server Application Developer's Guide.

## APPEND

Эта подпрограмма используется для добавления содержимого LOB-источника к содержимому LOB-приемника; при этом LOB-источник не изменяется. Процедура APPEND является переопределяемой: в ней допускается использование двоичных и символьных LOB:

```
PROCEDURE APPEND(
   dest_lob IN OUT BLOB,
   src_lob IN BLOB);
PROCEDURE APPEND(
   dest_lob IN OUT CLOB CHARACTER SET ANY_CS;
   src_lob IN CLOB CHARACTER SET dest_lob%CHARSET);
```

Здесь *dest_lob* – это LOB-приемник, а *src_lob* – LOB-источник. Обратите внимание: APPEND нельзя использовать для объектов BFILE, так как их можно только считывать. Если в *src_lob* или *в dest_lob* содержится NULL-значение, устанавливается исключительная ситуация VALUE_ERROR (полный перечень исключительных ситуаций, устанавливаемых подпрограммами DBMS_LOB, можно найти в конце этого раздела).

**▼ ВНИМАНИЕ** В процедуре APPEND, как и во многих других подпрограммах DBMS_LOB, применяется синтаксическая конструкция CHARACTER SET ANY_CS, которая указывает на возможность использования как типа CLOB, так и типа NCLOB. Атрибут %CHARSET возвращает набор символов указанного аргумента и обеспечивает применение для *src_lob* и *dest_lob* одного и того же набора символов.

## COMPARE

Эта функция используется для сравнения значений двух целых объектов LOB или значений фрагментов двух LOB. Если значения LOB тождественны, то COMPARE возвращает ноль, а если не тождественны, то не ноль. Функция COMPARE переопределяется всеми четырьмя типами LOB:

```
FUNCTION COMPARE(
   lob_1 IN BLOB,
   lob_2 IN BLOB,
   amount IN INTEGER := 4294967295,
   offset_1 IN INTEGER := 1,
   offset_2 IN INTEGER := 1)
   RETURN INTEGER;
PRAGMA RESTRICT_REFERENCES(compare, WNDS, RNDS, WNPS, RNPS);

FUNCTION COMPARE(
   lob_1 IN CLOB CHARACTER SET ANY_CS,
   lob_2 IN CLOB CHARACTER SET lob_1%CHARSET,
   amount IN INTEGER := 4294967295,
   offset_1 IN INTEGER := 1,
   offset_2 IN INTEGER := 1)
```

```
    RETURN INTEGER;
  PRAGMA RESTRICT_REFERENCES(compare, WNDS, RNDS, WNPS, RNPS);

  FUNCTION COMPARE(
    file_1 IN BFILE,
    file_2 IN BFILE,
    amount IN INTEGER := 4294967295,
    offset_1 IN INTEGER := 1,
    offset_2 IN INTEGER := 1)
    RETURN INTEGER;
  PRAGMA RESTRICT_REFERENCES(compare, WNDS, RNDS, WNPS, RNPS);
```

Заметьте, что функцию COMPARE можно применять только для сравнения значений LOB одного типа: двух объектов BLOB, двух объектов BFILE и т.д. Параметры и возвращаемое значение COMPARE описаны в таблице 21.3.

**Таблица 21.3.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *lob_1, file_1* | BLOB, CLOB, NCLOB, BFILE | Локатор первого сравниваемого LOB |
| *lob_2, file_2* | BLOB, CLOB, *NCLOB, BFILE* | Локатор второго сравниваемого LOB |
| *amount* | INTEGER | Максимальное число сравниваемых символов (CLOB, NCLOB) или байтов (BLOB, BFILE) |
| *offset_1* | INTEGER | Смещение в символах (CLOB, NCLOB) или в байтах (BLOB, BFILE) в первом LOB перед началом сравнения. Первый байт или символ имеет смещение 1 |
| *offset_2* | INTEGER | Смещение в символах (CLOB, NCLOB) или в байтах (BLOB, BFILE) во втором LOB перед началом сравнения. Первый байт или символ имеет смещение 1 |
| возвращаемое значение | INTEGER | Ноль, если значения двух LOB тождественны; не ноль, если не тождественны. Если *amount, offset_1* или *offset_2* < 1 либо > LOBMAXSIZE, то возвращается NULL-значение |

## COPY

Процедура COPY используется для копирования всего или части LOB-источника в LOB-приемник. Если указанное для LOB-приемника смещение превышает число элементов данных, находящихся в этом LOB, то вводятся символы нулевых байтов (для BLOB) или пробелы (для CLOB и NCLOB). Если указанное для приемника смещение меньше, чем текущий размер этого LOB, то существующие данные перезаписываются. Если указанное значение amount больше, чем размер LOB-источника, то копируется весь LOB-источник, и ошибка не устанавливается. Процедуру COPY можно применять для копирования данных, содержащихся в LOB одного типа:

```
  PROCEDURE COPY(
    dest_lob IN OUT BLOB,
    src_lob IN BLOB,
    amount IN INTEGER,
    dest_offset IN INTEGER := 1,
    src_offset IN INTEGER := 1);

  PROCEDURE COPY(
    dest_lob IN OUT CLOB CHARACTER SET ANY_CS,
    src_lob IN CLOB CHARACTER SET dest_lob%CHARSET,
    amount IN INTEGER,
    dest_offset IN INTEGER := 1,
    src_offset IN INTEGER := 1);
```

Параметры процедуры COPY описаны в таблице 21.4.

**Таблица 21.4.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *dest_lob* | BLOB, CLOB, NCLOB | Локатор LOB-приемника |
| *src_lob* | BLOB, CLOB, NCLOB | Локатор LOB-источника; должен быть того же типа, что и dest_lob |
| *amount* | INTEGER | Число копируемых байтов (BLOB) или символов (CLOB, NCLOB) |
| *dest_offset* | INTEGER | Смещение (относительно 1) в байтах (BLOB) или символах (CLOB, NCLOB) в LOB-приемнике, указывающее место копирования данных |
| *src_offset* | INTEGER | Смещение (относительно 1) в байтах (BLOB) или символах (CLOB, NCLOB) в LOB-источнике, указывающее начало копируемых данных |

Если *src_offset* или *dest_offset* меньше 1 либо больше LOBMAXSISE или если *amount* меньше 1 либо больше LOBMAXSISE, то устанавливается исключительная ситуация INVALID_ARGVAL. Использование процедуры COPY иллюстрируется на следующем ниже примере. Поскольку amount больше, чем размер v_Lob2, вводятся дополнительные пробелы.

```
-- Этот пример содержится в файле lobcopy.sql.
DECLARE
  v_Lob1  CLOB;
  v_Lob2  CLOB;
BEGIN
  -- Считаем локатор LOB-источника.
  SELECT clob_col
      INTO v_Lob1
      FROM lobdemo
      WHERE key = 1;

  -- Считаем локатор LOB-приемника.
  SELECT clob_col
      INTO v_Lob2
      FROM lobdemo
      WHERE key = 2;

  -- Скопируем все 26 символов из v_Lob1 в конец v_Lob2, начиная со
  -- смещения 50. Поскольку размер v_Lob2 не равен 50 символам,
  -- вводятся пробелы.
  DBMS_LOB.COPY(v_Lob2, v_Lob1, 26, 50, 1);

  -- Выведем результат.
  LOBPrint(v_Lob2);
END;
```

**▼ ВНИМАНИЕ** В этом примере используется процедура **LOBPrint**, о которой рассказывается ниже. Для вывода на экран символьных LOB эта процедура обращается к DBMS_LOB.READ и к DBMS_OUTPUT.

## ERASE

Эта функция используется для стирания информации всего LOB или его части. Стираемый фрагмент LOB заполняется пробелами (CLOB, NCLOB) или нулевыми байтами (BLOB). Обратите внимание: размер LOB остается прежним (для фактического удаления значения LOB применяется процедура TRIM):

```
PROCEDURE ERASE(
  lob_loc IN OUT BLOB,
  amount IN OUT INTEGER,
  offset IN INTEGER := 1);
```

```
PROCEDURE ERASE(
   lob_loc IN OUT CLOB CHARACTER SET ANY_CS,
   amount IN OUT INTEGER,
   offset IN INTEGER := 1);
```

Параметры процедуры ERASE описаны в таблице 21.5.

**Таблица 21.5.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *lob_loc* | BLOB, CLOB, NCLOB | Локатор стираемого LOB |
| *amount* | INTEGER | Число стираемых байтов (BLOB) или символов (CLOB, NCLOB). Если сумма *offset* и *amount* больше, чем размер значения LOB, то информация LOB стирается только до границы, указывающей его максимальный размер. В *amount* возвращается фактическое число стертых байтов или символов |
| *offset* | INTEGER | Смещение (относительно 1) в значении LOB, указывающее, откуда начинать стирание |

## FILECLOSE

Эта подпрограмма используется для закрытия BFILE, открытого с помощью FILEOPEN:

```
PROCEDURE FILECLOSE(file_loc IN OUT BFILE);
```

где *file_loc* – локатор закрываемого BFILE. Применение FILECLOSE демонстрируется на примере процедуры **FileExec** в разделе "LOADFROMFILE" далее в этой главе.

## FILECLOSEALL

Эта подпрограмма используется для закрытия всех объектов BFILE, открытых в данный момент сеансом. Ее удобно применять в обработчиках исключительных ситуаций для того, чтобы обеспечивать закрытие всех файлов перед выдачей сообщения об ошибке (как в процедуре UTL_FILE.FCLOSEALL):

```
PROCEDURE FILECLOSEALL;
```

Процедура FILECLOSEALL не имеет каких-либо аргументов. Если ни одного BFILE в это время не открыто, устанавливается исключительная ситуация UNOPENED_FILE.

## FILEEXISTS

Эта функция используется для проверки, существует ли на самом деле указанный файл операционной системы и можно ли его считать:

```
FUNCTION FILEEXISTS(file_loc IN BFILE)
   RETURN INTEGER;
PRAGMA RESTRICT_REFERENCES(fileexists,
   WNDS, RNDS, WNPS, RNPS);
```

где *file_loc* – локатор нужного BFILE. Если файл существует и читаем, то FILEEXISTS возвращает 1; в противном случае возвращается 0. Если *file_loc* является NULL-значением, не имеет соответствующих привилегий на каталог или на операционную систему, а также в случае ошибки операционной системы возвращается NULL.

## FILEGETNAME

Процедура FILEGETNAME возвращает имена каталога и файла, связанные с локатором BFILE при его инициализации (с помощью BFILENAME):

```
PROCEDURE FILEGETNAME(
   file_loc IN BFILE,
   dir_alias OUT VARCHAR2,
   filename OUT VARCHAR2);
```

где *file_loc* – это локатор BFILE, а в *dir_alias* и *filename* записываются имена соответственно каталога и файла. Максимальный размер *dir_alias* равен 30, а максимальный размер *filename* – 2000.

## FILEISOPEN

Эта функция используется для определения, открыт ли данный BFILE (с помощью FILEOPEN) для данного локатора:

```
FUNCTION FILEISOPEN(file_loc IN BFILE)
   RETURN INTEGER;
PRAGMA RESTRICT_REFERENCES(fileisopen,
   WNDS, RNDS, WNPS, RNPS);
```

где *file_loc* – локатор нужного файла. Если файл был открыт с этим локатором, то функция возвращает 1; в противном случае возвращается 0. FILEISOPEN проверяет состояние только передаваемого ей локатора; если она возвращает 0, то вполне возможно, что тот же самый внешний файл открыт для другого локатора.

## FILEOPEN

Эта процедура используется для открытия BFILE с целью последующей обработки:

```
PROCEDURE FILEOPEN(
   file_loc IN OUT BFILE,
   open_mode IN BINARY_INTEGER := FILE_READONLY);
```

где *file_loc* – локатор нужного файла. Параметр *open_mode* применяется для указания режима работы с BFILE (как и в модуле UTL_FILE). В Oracle8 версии 8.0 единственным возможным режимом является режим "только чтение". Именно это обозначает передаваемая константа DBMS_LOB.FILE_READONLY. В будущих вариантах Oracle8, скорее всего, можно будет использовать другие режимы и, соответственно, указывать другие константы. Когда для параметра *open_mode* передается значение, отличное от DBMS_LOB.FILE_READONLY, устанавливается исключительная ситуация INVALID_ARGVAL. Применение FILEOPEN демонстрируется на примере процедуры **FileExec** в разделе "LOADFROMFILE" ниже в этой главе.

## GETLENGTH

Функция GETLENGTH возвращает текущий размер указанного LOB, в байтах (BLOB, BFILE) или в символах (CLOB, NCLOB). В это значение включаются все пробелы или нулевые байты (появившиеся после выполнения операций ERASE или WRITE). Размер пустого LOB равен 0. Функция GETLENGTH переопределяется всеми четырьмя типами LOB:

```
FUNCTION GETLENGTH(lob_loc IN BLOB)
   RETURN INTEGER;
   PRAGMA RESTRICT_REFERENCES(getlength,
        WNDS, RNDS, WNPS, RNPS);

FUNCTION GETLENGTH(lob_loc IN CLOB CHARACTER SET ANY_CS)
   RETURN INTEGER;
   PRAGMA RESTRICT_REFERENCES(getlength,
        WNDS, RNDS, WNPS, RNPS);

FUNCTION GETLENGTH(file_loc IN BFILE)
   RETURN INTEGER;
   PRAGMA RESTRICT_REFERENCES(getlength,
        WNDS, RNDS, WNPS, RNPS);
```

Здесь *lob_loc* или *file_loc* указывает локатор того LOB, размер которого возвращается. NULL возвращается в случае истинности любого из следующих условий:

- Во входном LOB содержится NULL-значение
- *file_loc* не указывает на корректный открытый файл
- *file_loc* не имеет необходимых привилегий на каталог или на операционную систему
- В результате ошибки операционной системы невозможно считать *file_loc*

Обратите внимание: исключительные ситуации в этих случаях не устанавливаются.

Применение GETLENGTH демонстрируется на примере процедуры **FileExec** в разделе "LOADFROMFILE" далее в этой главе.

## INSTR

Функция DBMS_LOB.INSTR аналогична функции INSTR, определенной в модуле STANDARD для строк символов типов CHAR и VARCHAR2. Она применяется для поиска позиции n-ого вхождения указанного шаблона в данном LOB. Функция INSTR переопределяется всеми четырьмя типами LOB:

```
FUNCTION INSTR(
   lob_loc IN BLOB,
   pattern IN RAW,
   offset IN INTEGER := 1,
   nth IN INTEGER := 1);
PRAGMA RESTRICT_REFERENCES(instr,
   WNDS, RNDS, WNPS, RNPS);

FUNCTION INSTR(
   file_loc IN BFILE,
   pattern IN RAW,
   offset IN INTEGER := 1,
   nth IN INTEGER := 1);
PRAGMA RESTRICT_REFERENCES(instr,
   WNDS, RNDS, WNPS, RNPS);

FUNCTION INSTR(
   lob_loc IN CLOB CHARACTER SET ANY_CS,
   pattern IN VARCHAR2 CHARACTER SET lob_loc% CHARSET,
   offset IN INTEGER := 1,
   nth IN INTEGER := 1);
PRAGMA RESTRICT_REFERENCES(instr,
   WNDS, RNDS, WNPS, RNPS);
```

Параметры и возвращаемое значение функции INSTR описаны в таблице 21.6.

**Таблица 21.6.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *lob_loc, file_loc* | BLOB, BFILE, CLOB, NCLOB | Локатор исследуемого LOB |
| *pattern* | VARCHAR2, RAW | Искомая строка байтов (BLOB, BFILE) или символов (CLOB, NCLOB) (шаблон). При указании параметра *pattern* для NCLOB необходимо использовать тот же набор символов, что и для *lob_loc* |
| *offset* | INTEGER | То смещение (относительно 1) в байтах (BLOB, BFILE) или в символах (CLOB, NCLOB) в *lob_loc или file_loc*, с которого начинается поиск соответствия шаблону |
| *nth* | INTEGER | Номер возвращаемого вхождения (начиная с 1) |
| возвращаемое значение | INTEGER | Позиция в байтах (BLOB, BFILE) или в символах (CLOB, NCLOB) в LOB, с которой начинается *n*-ое вхождение шаблона, или 0, если ни одно вхождение шаблона не найдено |

*Применение INSTR демонстрируется на примере процедуры* FileExec *в разделе* "LOADFROMFILE" *далее в этой главе.*

## LOADFROMFILE

*Эту процедуру можно применять для заполнения двоичных или символьных объектов* LOB *информацией файлов операционной системы. Можно указать часть* LOB*-источника, которую нужно считать, и смещение в* LOB*-приемнике, определяющее место записи данных. Процедура* LOADFROMFILE *переопределяется типами приемников (*BLOB, CLOB *и* NCLOB*):*

```
PROCEDURE LOADFROMFILE(
   dest_lob IN OUT BLOB,
   src_lob IN BFILE,
   amount IN INTEGER,
   dest_offset IN INTEGER := 1,
   src_offset IN INTEGER := 1);
```

```
PROCEDURE LOADFROMFILE(
   dest_lob IN OUT NCLOB CHARACTER SET ANY_CS,
   src_lob IN BFILE,
   amount IN INTEGER,
   dest_offset IN INTEGER := 1,
   src_offset IN INTEGER := 1);
```

Параметры процедуры LOADFROMFILE описаны в таблице 21.7.

**Таблица 21.7.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *dest_lob* | BLOB, CLOB, NCLOB | LOB, в который записываются данные |
| *src_lob* | BFILE | Считываемый файл; перед вызовом LOADFROMFILE он должен быть уже открыт и инициализирован (с помощью FILEOPEN) |
| *amount* | INTEGER | Число байтов, считываемых из исходного файла |
| *dest_offset* | INTEGER | Смещение (относительно 1) в байтах для BLOB или в символах для CLOB и NCLOB, указывающее место начала записи в *dest_lob* |
| *src_offset* | INTEGER | Смещение (в байтах, относительно 1) в исходном файле, указывающее место начала считывания |

Относительно использования процедуры LOADFROMFILE следует сделать ряд замечаний:

- Если *dest_offset* превышает текущий размер *dest_lob*, то для заполнения разницы вводятся нулевые байты или пробелы (как в процедуре COPY).
- Перед вызовом LOADFROMFILE LOB-приемник должен быть блокирован, то есть для этого LOB нужно выполнить оператор SELECT FOR UPDATE. Кроме того, LOB-приемник не должен содержать NULL-значение.
- Если информация загружается в CLOB, то для файла операционной системы необходимо использовать тот же набор символов, что и применяемый в базе данных. Проверка этого соответствия не производится.

Использование процедуры LOADFROMFILE, а также ряда других функций, предназначенных для работы с объектами BFILE, иллюстрируется на примере процедуры **FileExec**. Она служит для считывания информации в файл, состоящий из SQL-операторов, а также для выполнения этого файла.

```
-- Этот пример содержится в файле fileexec.sql.
CREATE OR REPLACE PROCEDURE FileExec(
  -- Выполняет SQL-операторы в файле, указанном переменными
  -- p_Directory and p_FileName. Каждый из операторов не должен
  -- заканчиваться двоеточием (если он не является блоком
  -- PL/SQL block) и должен отделяться символом p_SeparationChar.
  p_Directory IN VARCHAR2,
  p_FileName IN VARCHAR2,
  p_SeparationChar IN CHAR) AS

  v_FileLocator BFILE;
  v_CLOBLocator CLOB;
  v_SQLCursor INTEGER;
  v_StartPoint INTEGER := 1;
  v_EndPoint INTEGER;
  v_SQLStatement VARCHAR2(32000);
  v_StatementLength INTEGER;
  v_RC INTEGER;
BEGIN
```

```
    -- Инициализируем символьный локатор для записи. Обратите внимание:
    -- CLOB нужно считывать из таблицы в режиме FOR UPDATE, что блокирует
    -- строку (это является обязательным условием для LOADFROMFILE).
    SELECT clob_col
        INTO v_CLOBLocator
        FROM lobdemo
        WHERE key = -1
        FOR UPDATE;

    -- Инициализируем локатор BFILE для чтения.
    v_FileLocator := BFILENAME(p_Directory, p_FileName);
    DBMS_LOB.FILEOPEN(v_FileLocator, DBMS_LOB.FILE_READONLY);

    -- Откроем курсор.
    v_SQLCursor := DBMS_SQL.OPEN_CURSOR;

    -- Загрузим весь файл в символьный LOB. Это обязательно, поскольку
    -- данные находятся в символьных переменных, а не в переменных типа RAW.
    DBMS_LOB.LOADFROMFILE(v_CLOBLocator, v_FileLocator,
                          DBMS_LOB.GETLENGTH(v_FileLocator));

    -- Последовательно просмотрим LOB, отмечая каждый экземпляр
    -- символа-разделителя.
    LOOP
        v_EndPoint := DBMS_LOB.INSTR(v_CLOBLocator, p_SeparationChar,
                                     v_StartPoint, 1);
        EXIT WHEN v_EndPoint = 0;

        -- Выделим информацию, находящуюся между начальной и конечной
        -- точками. Это и есть SQL-оператор, который нужно выполнить.
        v_StatementLength := v_EndPoint - v_StartPoint;
        v_SQLStatement := DBMS_LOB.SUBSTR(v_CLOBLocator,
          v_StatementLength, v_StartPoint);

        -- Отобразим оператор на экране, а затем выполним его с помощью
        -- DBMS_SQL.
        DBMS_OUTPUT.PUT_LINE(v_SQLStatement);
        DBMS_SQL.PARSE(v_SQLCursor, v_SQLStatement, DBMS_SQL.V7);
        v_RC := DBMS_SQL.EXECUTE(v_SQLCursor);

        -- Увеличим значение указателя операторов, установив его на
        -- следующий оператор.
        v_StartPoint := v_EndPoint + 1;
    END LOOP;

    -- Закроем файл и курсор.
    DBMS_LOB.FILECLOSE(v_FileLocator);
    DBMS_SQL.CLOSE_CURSOR(v_SQLCursor);
EXCEPTION
    WHEN OTHERS THEN
        -- Закроем файл и курсор и переустановим исключительную ситуацию.
        DBMS_LOB.FILECLOSE(v_FileLocator);
        DBMS_SQL.CLOSE_CURSOR(v_SQLCursor);
        RAISE;
END FileExec;
```

Если был создан каталог statements, в котором находится следующий файл:

```
-- Этот пример содержится в файле inserts.sql.
INSERT INTO temp_table (num_col, char_col)
  VALUES (1, 'hello')
/
INSERT INTO temp_table (num_col, char_col)
  VALUES (2, 'hello')
/
INSERT INTO temp_table (num_col, char_col)
  VALUES (3, 'hello')
/
INSERT INTO temp_table (num_col, char_col)
  VALUES (4, 'hello')
/
INSERT INTO temp_table (num_col, char_col)
  VALUES (5, 'hello')
/
INSERT INTO temp_table (num_col, char_col)
  VALUES (6, 'hello')
/
INSERT INTO temp_table (num_col, char_col)
  VALUES (7, 'hello')
/
```

то можно вызвать FileExec и ввести 7 строк в таблицу temp_table с помощью такого блока PL/SQL:

```
BEGIN
  FileExec('STATEMENTS', 'inserts.sql', '/');
END;
```

## READ

Процедура READ используется для считывания фрагмента LOB, начиная с указанного смещения. Если во время чтения достигается конец LOB, то в *amount* возвращается ноль и устанавливается исключительная ситуация NO_DATA_FOUND. Процедура READ переопределяется всеми четырьмя типами LOB:

```
PROCEDURE READ(lob_loc IN BLOB,
               amount IN OUT BINARY_INTEGER,
               offset IN INTEGER,
               buffer OUT RAW);

PROCEDURE READ(lob_loc IN CLOB CHARACTER SET ANY_CS,
               amount IN OUT BINARY_INTEGER,
               offset IN INTEGER,
               buffer OUT VARCHAR2);

PROCEDURE READ(lob_loc IN BFILE,
               amount IN OUT BINARY_INTEGER,
               offset IN INTEGER,
               buffer OUT RAW);
```

Параметры процедуры READ описаны в таблице 21.8.

Использование READ для просмотра LOB целиком продемонстрировано на примере процедуры **LOBPrint**. Содержимое CLOB выводится на экран с помощью модуля DBMS_OUTPUT, в виде строк по 80 символов каждая.

```
-- Этот пример содержится в файле lobprint.sql.
CREATE OR REPLACE PROCEDURE LOBPrint(p_CLOB IN CLOB) AS
  v_Buffer VARCHAR2(80);
```

```
    v_Offset  INTEGER  := 1;
    v_Amount  INTEGER  := 80;
BEGIN
    LOOP
        -- Считаем и отобразим следующие 80 символов.
        DBMS_LOB.READ(p_CLOB, v_Amount, v_Offset, v_Buffer);
        DBMS_OUTPUT.PUT_LINE(v_Buffer);

        v_Offset  := v_Offset + v_Amount;
    END LOOP;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        -- Конец цикла и возврат.
        NULL;
END LOBPrint;
```

**Таблица 21.8.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *lob_loc* | BLOB, CLOB, NCLOB, BFILE | LOB, из которого считываются данные |
| *amount* | BINARY_INTEGER | Число считываемых байтов (BLOB, BFILE) или символов (CLOB, NCLOB). Возвращается фактическое число байтов или символов |
| *offset* | INTEGER | Смещение (относительно 1) в байтах для BLOB и BFILE или в символах для CLOB и NCLOB, указывающее место начала считывания в *lob_loc* |
| *buffer* | VARCHAR2, RAW | Выходной буфер для получения данных |

## SUBSTR

Функция DBMS_LOB.SUBSTR аналогична функции SUBSTR, определенной в модуле STANDARD. Она, как и READ, применяется для считывания фрагментов данных LOB. Функция SUBSTR переопределяется всеми четырьмя типами LOB:

```
FUNCTION SUBSTR(lob_loc IN BLOB,
                amount IN INTEGER := 32767,
                offset IN INTEGER := 1)
  RETURN RAW;
PRAGMA RESTRICT_REFERENCES(substr,
  WNDS, RNDS, WNPS, RNPS);

FUNCTION SUBSTR(lob_loc IN CLOB CHARACTER SET ANY_CS,
                amount IN INTEGER := 32767,
                offset IN INTEGER := 1)
  RETURN VARCHAR2;
PRAGMA RESTRICT_REFERENCES(substr,
  WNDS, RNDS, WNPS, RNPS);

FUNCTION SUBSTR(lob_loc IN BFILE,
                amount IN INTEGER := 32767,
                offset IN INTEGER := 1)
  RETURN RAW;
PRAGMA RESTRICT_REFERENCES(substr,
  WNDS, RNDS, WNPS, RNPS);
```

Параметры процедуры SUBSTR описаны в таблице 21.9.

**Таблица 21.9.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *lob_loc* | BLOB, CLOB, NCLOB, BFILE | LOB, из которого считываются данные |
| *amount* | INTEGER | Число считываемых байтов (BLOB, BFILE) или символов (CLOB, NCLOB) |
| Offset | INTEGER | Смещение (относительно 1) в байтах для BLOB и BFILE или в символах для CLOB и NCLOB, указывающее место начала считывания в lob_loc |
| возвращаемое значение | VARCHAR2, RAW | Выбранный фрагмент LOB |

Применение SUBSTR демонстрируется на примере процедуры **FileExec** в разделе "LOADFROMFILE" выше в этой главе.

## TRIM

Процедура TRIM используется для удаления данных из конца LOB. В отличие от процедуры ERASE, которая замещает фрагменты LOB пробелами или NULL-значениями, TRIM удаляет данные и уменьшает внутренний размер. Она переопределяется типами BLOB и CLOB:

```
PROCEDURE TRIM(lob_loc IN CLOB,
               newlen IN INTEGER);
PROCEDURE TRIM(lob_loc IN BLOB,
               newlen IN INTEGER);
```

где *lob_loc* – локатор сокращаемого LOB, а *newlen* – новый размер LOB. Данные, находящиеся после newlen, удаляются. Если значение newlen превышает размер LOB, устанавливается ошибка "ORA-22926: specified trim length is greater than current LOB value's length" (указанная длина сокращаемых данных превышает текущий размер LOB. – *Прим. пер.*). Если lob_loc является NULL-значением, устанавливается исключительная ситуация VALUE_ERROR. Однако если TRIM вызывается для пустого LOB, ошибка не устанавливается и LOB не изменяется. Использование процедуры TRIM иллюстрируется на следующем примере:

```
-- Этот пример содержится в файле lobtrim.sql.
DECLARE
  v_BLOB  BLOB;
BEGIN
  SELECT blob_col
      INTO v_BLOB
      FROM lobdemo
      WHERE key = 1
      FOR UPDATE;

  DBMS_OUTPUT.PUT_LINE('Before trim, length = ' ||
                       DBMS_LOB.GETLENGTH(v_BLOB))
  DBMS_LOB.TRIM(v_BLOB, 5);
  DBMS_OUTPUT.PUT_LINE('After trim, length = ' ||
                       DBMS_LOB.GETLENGTH(v_BLOB));

  -- Размер равен 5, поэтому устанавливается ORA-22926.
  DBMS_LOB.TRIM(v_BLOB, 10);
END;
```

## WRITE

Процедура DBMS_LOB.WRITE используется для записи данных во фрагменты LOB. Все данные, находящиеся в указанном диапазоне, перезаписываются. Если величина смещения превышает размер LOB-приемника, то в качестве заполнителя вводятся пробелы или нулевые байты. Процедура WRITE переопределяется типами CLOB и NCLOB:

```
PROCEDURE WRITE(lob_loc IN OUT BLOB,
                amount IN BINARY_INTEGER,
```

*offset* IN INTEGER,
*buffer* IN **RAW**);
PROCEDURE WRITE(*lob_loc* IN OUT **CLOB CHARACTER SET ANY_CS**,
*amount* IN BINARY_INTEGER,
*offset* IN INTEGER,
*buffer* IN **VARCHAR2**);

Параметры процедуры WRITE описаны в таблице 21.10.

**Таблица 21.10.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *lob_loc* | BLOB, CLOB, NCLOB | LOB, в который записываются данные |
| *amount* | BINARY_INTEGER | Число записываемых байтов (BLOB) или символов (CLOB, NCLOB). Значение *amount* не должно превышать размер буфера |
| *offset* | INTEGER | Смещение (относительно 1) в байтах для BLOB или в символах для CLOB и NCLOB в *lob_loc* |
| *buffer* | RAW, VARCHAR2 | Данные, записываемые в LOB-приемник |

## Исключительные ситуации, устанавливаемые подпрограммами модуля DBMS_LOB

В таблице 21.11 описаны исключительные ситуации, которое могут устанавливаться подпрограммами модуля DBMS_LOB. Более подробно о каждой из исключительных ситуаций рассказано выше, в разделах, посвященных подпрограммам DBMS_LOB. Если это не оговорено особо, все исключительные ситуации определяются в модуле DBMS_LOB.

**Таблица 21.11. Исключительные ситуации в подпрограммах BFILE модуля DBMS_LOB**

| Исключительная ситуация | Код ошибки | Сообщение об ошибке |
|---|---|---|
| ACESS_ERROR | 22925 | В ходе выполнения операции будет превышен максимальный размер, разрешенный для LOB |
| INVALID_ARGVAL | 21590 | Аргумент является NULL-значением, неверен или вне диапазона |
| INVALID_DIRECTORY | 22287 | Неверный или модифицированный каталог |
| NO_DATA_FOUND* | 1403 | Данные не найдены |
| NOEXIST_DIRECTORY | 22285 | Каталог не существует |
| NOPRIV_DIRECTORY | 22286 | Недостаточные привилегии для работы с каталогом |
| OPEN_TOOMANY | 22290 | Достигнут установленный предел на число открытых файлов |
| OPERATION_FAILED | 22288 | Операция, выполняемая над файлом, завершена неудачно |
| UNOPENED_FILE | 22289 | Невозможно выполнить операцию над неоткрытым файлом |
| VALUE_ERROR* | 6502 | Ошибка числа или значения |

* Эти исключительные ситуации определены в модуле STANDARD, а не в DBMS_LOB.

## Сравнение DBMS_LOB и OCI

Со значениями LOB можно работать не только с помощью модуля DBMS_LOB, но и через интерфейс OCI Oracle8. Полный анализ функций OCI для работы с объектами LOB не является предметом этой книги. За более подробной информацией обращайтесь к руководству Programmer's Guide to the Oracle8 Call Interface, Volume 1: OCI Concepts. Однако для справки в таблице 21.12 перечислены функции

DBMS_LOB и эквивалентные им функции OCI. Обратите внимание: некоторые функции доступны только в OCI, а некоторые — только в DBMS_LOB.

**Таблица 21.12. Функции OCI для работы с объектами LOB**

| Функция OCI | Эквивалентная функция DBMS_LOB | Описание |
|---|---|---|
| OCILobAppend() | APPEND | Добавляет содержимое одного LOB в другой LOB |
| OCILobAssign() | — | Присваивает локатор одного LOB другому |
| OCILobCharSetForm() | — | Возвращает форму набора символов для данного LOB |
| OCILobCharSetId() | — | Возвращает идентификатор набора символов для данного LOB |
| OCILobCopy() | COPY | Копирует фрагмент данных LOB-источника в LOB-приемник |
| OCILobErase() | ERASE | Стирает часть LOB, начиная с указанного смещения |
| OCILobGetLength() | GETLENGTH | Возвращает размер LOB или BFILE |
| OCILobEqual() | — | Проверяет, указывают ли два локатора LOB на одно и то же значение LOB |
| OCILobLocatorIsInit() | — | Проверяет, инициализирован ли данный LOB |
| OCILobLocatorSize() | — | Возвращает размер локатора LOB |
| OCILobRead() | READ | Считывает указанный фрагмент LOB- или BFILE-источника во вспомогательный буфер |
| OCILobTrim() | TRIM | Усекает LOB |
| OCILobWrite() | WRITE | Записывает данные из буфера в LOB, начиная с указанного смещения и перезаписывая существующую информацию |
| OCILobFileOpen() | FILEOPEN | Открывает BFILE |
| OCILobFileIsOpen() | FILEISOPEN | Возвращает TRUE, если BFILE открыт |
| OCILobExists() | FILEEXISTS | Проверяет существование BFILE |
| OCILobFileClose() | FILECLOSE | Закрывает BFILE |
| OCILobFileCloseAll() | FILECLOSEALL | Закрывает все объекты BFILE, открытые в это время |
| OCILobFileSetName() | — | Устанавливает имя BFILE в локаторе |
| OCILobFileGetName() | FILEGETNAME | Возвращает имя BFILE |
| — | COMPARE | Сравнивает фрагменты значений двух LOB |
| — | INSTR | Используется для установления соответствия шаблону в символьном LOB |
| — | SUBSTR | Возвращает указанный фрагмент значения LOB |
| — | LOADFROMFILE | Копирует BFILE в CLOB или BLOB |

# PL/SQL в работе: копирование данных LONG в объект LOB

Как было показано в этой главе, объекты LOB имеют достаточно много преимуществ по сравнению с данными типа LONG или LONG RAW, применяемыми в Oracle7. Поэтому может потребоваться преобразовать информацию, хранимую в столбцах LONG, в эквивалентные объекты LOB. Именно для этого применяется процедура Long2Lob, позволяющая выполнить поставленную задачу исключительно средст-

вами PL/SQL. Для считывания фрагментов данных LONG в **Long2Lob** используется модуль DBMS_SQL (глава 15), а для ввода этих фрагментов в объект LOB — модуль DBMS_LOB:

```
-- Этот пример содержится в файле long2lob.sql.
CREATE OR REPLACE PROCEDURE Long2Lob(
  -- Для считывания столбца LONG, указанного в p_LongQuery, применяется
  -- DBMS_SQL. Выбранная информация записывается в p_CLob.
  p_LongQuery IN VARCHAR2,
  p_CLob IN OUT CLOB) AS

  c_ChunkSize CONSTANT INTEGER := 100;

  v_CursorID INTEGER;
  v_RC INTEGER;
  v_Chunk VARCHAR2(100);
  v_ChunkLength INTEGER;
  v_Offset INTEGER := 0;
BEGIN
  -- Откроем курсор, опишем столбец, выполним оператор и считаем информацию.
  v_CursorID := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(v_CursorID, p_LongQuery, DBMS_SQL.V7);
  DBMS_SQL.DEFINE_COLUMN_LONG(v_CursorID, 1);
  v_RC := DBMS_SQL.EXECUTE_AND_FETCH(v_CursorID);

  -- Последовательно просмотрим LONG, по одному считывая символы
  -- c_ChunkSize из LONG и внося их в LOB.
  LOOP
      DBMS_SQL.COLUMN_VALUE_LONG(v_CursorID, 1, c_ChunkSize,
        v_Offset, v_Chunk, v_ChunkLength);
      DBMS_LOB.WRITE(p_CLob, v_ChunkLength, v_Offset + 1,
        v_Chunk);
      IF v_ChunkLength < c_ChunkSize THEN
        EXIT;
      ELSE
        v_Offset := v_Offset + v_ChunkLength;
      END IF;
  END LOOP;

  DBMS_SQL.CLOSE_CURSOR(v_CursorID);
EXCEPTION
  WHEN OTHERS THEN
      -- Закроем курсор и переустановим исключительную ситуацию.
      DBMS_SQL.CLOSE_CURSOR(v_CursorID);
      RAISE;
END Long2Lob;
```

Предположим, что существует таблица, созданная и заполненная следующим образом:

```
-- Этот пример является частью файла 121test.sql.
CREATE TABLE long_tab (
  key NUMBER,
  long_col LONG
);
INSERT INTO long_tab (key, long_col)
  VALUES (100,
```

```
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmn
opqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzABC
DEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR
STUVWXYZabcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZabcdefg
hijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuv
wxyzABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzABCDEFGHIJK
LMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmno
pqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzABCD
EFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRS
TUVWXYZabcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZabcdefgh
ijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw
xyzABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzABCDEFGHIJKL
MNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
);
```

Теперь можно перенести данные из столбца long_col в таблицу lobdemo, вызвав процедуру Long2Lob:

```
-- Этот пример является частью файла l2ltest.sql.
DECLARE
  v_Query VARCHAR2(200);
  v_NewLOB CLOB;
BEGIN
  v_Query := 'SELECT long_col FROM long_tab where key = 100';
  INSERT INTO lobdemo (key, clob_col)
      VALUES (100, EMPTY_CLOB())
      RETURNING clob_col INTO v_NewLOB;
  Long2Lob(v_Query, v_NewLOB);
  LOBPrint(v_NewLOB);
  COMMIT;
END;
```

Относительно использования процедуры **Long2Lob** следует сделать ряд замечаний:

- Переменная **p_LongQuery** должна быть оператором SELECT, который отбирает одну строку и возвращает один столбец типа LONG.
- Если локатор LOB выбирается в таблице, а не является новым локатором, необходимо использовать оператор SELECT FOR UPDATE для блокировки соответствующей строки, так как процедура **Long2Lob** ее обновляет.
- Метод, применявшийся в **Long2Lob**, можно усовершенствовать и преобразовывать всю таблицу, используя DBMS_SQL для выбора значений полей столбца LONG из каждой строки, а затем последовательно просматривая каждую считанную строку и записывая данные в LOB.

# Итоги

В этой главе было рассмотрено средство Oracle8, называемое большими объектами (LOB – Large OBjects). Для хранения различных видов информации размером до 4 гигабайтов используется четыре типа LOB: CLOB, NCLOB, BLOB и BFILE. Данные могут храниться внутри Oracle и управляться базой данных, а могут находиться вне Oracle и управляться базовой операционной системой. Был дан анализ модуля DBMS_LOB, который является интерфейсом PL/SQL, предназначенным для работы с информацией объектов LOB. В следующей, последней главе будут рассмотрены способы повышения производительности приложений, созданных на PL/SQL.

# Глава 22

## *Производительность и настройка*

Крайне важно, чтобы созданное приложение функционировало должным образом и обрабатывало данные максимально быстро и эффективно. Создание приложения PL/SQL не станет слишком сложным, если всегда помнить о нескольких основных аспектах в этих процессах. Разделяемый пул, грамотно написанные SQL-операторы и надлежащим образом настроенная сеть помогут разработать оптимально функционирующее приложение. Каждое из этих понятий подробно анализируется в данной главе. Кроме этого, здесь рассмотрен третий, и последний, компонент SQL-Station – SQL-Station Plan Analyzer.

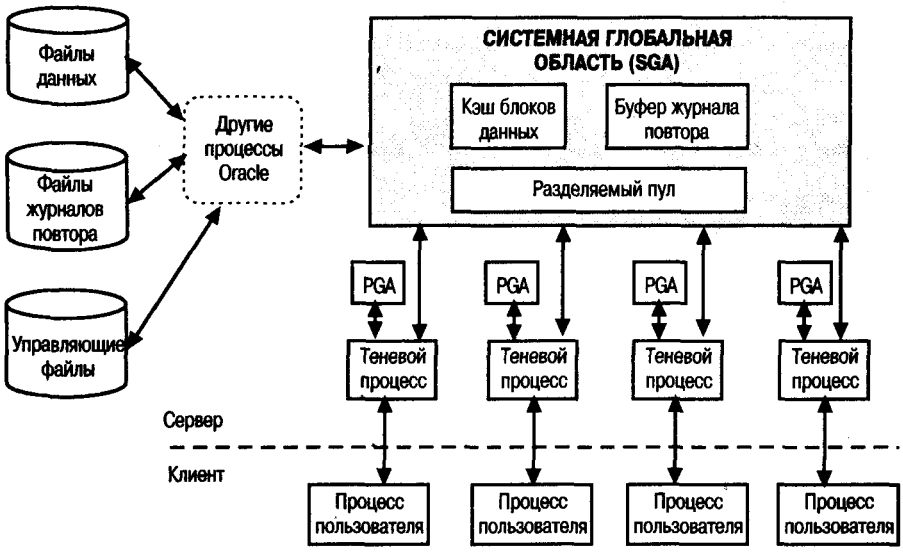
# Разделяемый пул

Разделяемый пул (shared pool) – это область памяти, в которой Oracle сохраняет информацию об SQL-операторах и хранимых подпрограммах. Установка нужного размера разделяемого пула и использование его надлежащим образом являются особенно важными в процессе создания оптимально работающего приложения PL/SQL.

## Структура экземпляра Oracle

Перед тем как перейти к обсуждению работы разделяемого пула и его влияния на процесс программирования на PL/SQL, проанализируем структуру выполняющегося экземпляра Oracle. Экземпляр (instance) состоит из процессов, оперативной памяти и файлов операционной системы, функционирующих на сервере. Кроме того, с базой данных могут соединяться процессы клиентов. Структура экземпляра приведена на рис. 22.1.

**Рис. 22.1.**
*Экземпляр Oracle*




▼ ВНИМАНИЕ В некоторых системах (например в Windows 3.1) в состав экземпляра Oracle входит только один процесс. Этот процесс несет ответственность за все операции по администрированию базы данных и за обслуживание теневого процесса. Разделяемый пул ведет себя точно так же.

## Процессы

При старте экземпляра запускаются несколько фоновых процессов Oracle. Они взаимодействуют между собой через разделяемую память, называемую системной глобальной областью (system global area, SGA). Все процессы Oracle могут считывать информацию, содержащуюся в различных структурах данных SGA, а также записывать информацию в эти структуры данных. Каждый процесс отвечает за определенный аспект функционирования базы данных. Некоторые из этих процессов описаны в таблице 22.1. Полный анализ данных процессов не является предметом данной книги; за более подробной информацией обращайтесь к руководству Oracle Server Concepts. В этой главе будут обсуждены процессы пользователей и теневые процессы.

### Таблица 22.1. Процессы базы данных

| Процесс | Описание |
|---|---|
| Процесс записи в базу данных (Database Writer, DBWR) | Переписывает измененную информацию из буферного кэша SGA в файлы базы данных. |
| Процесс монитора процессов (Process Monitor, PMON) | Устраняет последствия работы прерванных процессов пользователей: снимает блокировки и откатывает все незавершенные SQL-операторы. |
| Процесс системного монитора (System Monitor, SMON) | При необходимости восстанавливает экземпляр во время его запуска. Несет ответственность за освобождение временных сегментов и сращивание свободного пространства SGA. |
| Процесс восстановления (Recover, RECO) | Устраняет проблемы, связанные с распределенными транзакциями. Функционирует только в распределенной среде. |
| Процесс записи в журнал (Log Writer, LGWR) | Переписывает буфер журнала повтора в журнал повтора на диске. Процессы LGWR и DBWR работают совместно, обеспечивая надлежащую запись всех завершенных транзакций перед их успешным завершением. |
| Процесс архивации (Archiver, ARCH) | Копирует заполненный журнал повтора в автономное устройство хранения информации, например на магнитную ленту; работает только в том случае, если для базы данных установлен режим ARCHIVELOG. |
| Процесс регенерации моментальных снимков (Snapshot Refresh, SNP) | Регенерирует моментальные снимки таблиц и выполняет задания для базы данных (см. главу 18). |
| Теневой процесс (Shadow Process) | Управляет обменом информацией между процессом пользователя и другими ресурсами базы данных. |
| Процесс пользователя (User Process) | Выполняет пользовательское приложение, посылая базе данных SQL-операторы и блоки PL/SQL. |
| Процесс диспетчера (Dispatcher Process) | Позволяет нескольким пользователям совместно работать с одним теневым процессом как частью многопоточного сервера. Процесс пользователя соединяется с диспетчером, который, в свою очередь, передает информацию теневому процессу. Соединение пользователя с процессом диспетчера должно устанавливаться при помощи SQL*Net версии 2. |

**Процессы пользователей** Процесс пользователя – это приложение, посылающее SQL-операторы и блоки PL/SQL на сервер для обработки. Приложения можно создавать с помощью таких программных средств Oracle, как предкомпиляторы, OCI, SQL*Plus, пакет Developer 2000, а также средств разработки, созданных другими компаниями, например программой SQL-Station. Процесс пользователя через SQL*Net взаимодействует по сети с теневым процессом (см. в разделе "Работа с сетью").

**Теневые процессы** Теневой процесс отвечает за передачу информации остальной части экземпляра от процесса пользователя, и обратно. Если база данных не работает в режиме многопоточного сервера, то для каждого процесса пользователя существует собственный теневой процесс. В многопоточном сервере один теневой процесс может обслуживать несколько пользователей с помощью процессов диспетчеров. Более подробно о многопоточном сервере и способах его настройки рассказано в руководстве Oracle Server Concepts.

## Память

Память, используемая экземпляром, подразделяется на различные области, с которыми работают разные процессы. В число этих областей входят разделяемый пул, кэш блоков данных, буфер журнала повтора и глобальные области процессов (PGA). Все области, кроме PGA, выделяются как части SGA и поэтому доступны всем процессам. Область SGA целиком выделяется при запуске экземпляра. Ее размер определяется различными параметрами файла INIT.ORA, в том числе SHARED_POOL_SIZE, LOG_BUFFER и DB_BLOCK_BUFFERS. Более подробно об этих и других параметрах файла INIT.ORA рассказано в руководстве Oracle Server Reference.

**Разделяемый пул (SHARED POOL)** Разделяемый пул – это структура памяти, которая более других влияет на производительность программ PL/SQL. В разделяемом пуле содержатся текст SQL-операторов и блоков PL/SQL, посылаемых базе данных, а также их варианты, подвергшиеся грамматическому разбору. Программные элементы не остаются в разделяемом пуле навсегда; с течением

времени, становясь ненужными, они удаляются из пула. О функционировании разделяемого пула будет вкратце рассказано ниже.

**Кэш блоков данных (DB BLOCK CACHE)** Данные считываются из базы данных и записываются туда *блоками* (blocks). Для максимальной эффективности размер блоков обычно задается кратным размеру блоков операционной системы и указывается в параметре DB_BLOCK_SIZE файла INIT.ORA. Стандартными значениями для параметра DB_BLOCK_SIZE являются 2048 и 4096, так как блоки операционной системы часто бывают равны 1024 байт. Блоки данных считываются из файлов данных теневыми процессами, после чего хранятся и обрабатываются в кэше блоков данных. Если содержимое некоторого блока изменяется, процесс DBWR записывает этот блок обратно в файл.

**Буфер журнала повтора (REDO LOG BUFFER)** Буфер журнала повтора работает в том же режиме, что и кэш блоков данных, но в нем содержатся не блоки данных, а блоки повтора. Эта информация создается при обработке SQL-операторов и блоков PL/SQL и используется в случае сбоя экземпляра, а также для обеспечения правильности считываемых данных. За переписывание блоков из буфера журнала повтора в автономные журналы повтора отвечает процесс LGWR.

**Глобальная область процесса (PGA)** Для функционирования каждого теневого процесса выделяется память, называемая *глобальной областью процесса* (process global area, PGA). В PGA хранятся активные наборы всех открытых в это время курсоров, указатели на подвергшиеся грамматическому разбору, и находящиеся в разделяемом пуле SQL-операторы и блоки PL/SQL; кроме того, в PGA выделяется память для обработки операторов PL/SQL. Память PGA выделяется при запуске теневого процесса и сохраняется в течение всего времени существования соединения. При работе в режиме многопоточного сервера (MTS) некоторая часть информации, обычно находящейся в PGA, хранится в SGA. Дело в том, что в режиме MTS управление сеансом может передаваться от одного теневого процесса к другому.

## Файлы

В работе базы данных используются файлы трех видов: файлы данных, файлы журналов повтора и управляющие файлы, за обслуживание которых отвечают различные процессы. Эти файлы отличаются от внешних файлов, с которыми работают подпрограммы модуля UTL_FILE (см. главу 18), и от внешних файлов, используемых для хранения объектов BFILE (см. главу 21).

**Файлы данных (DATABASE FILES)** В этих файлах содержится реальная информация базы данных: табличные и индексные данные, текст хранимых подпрограмм, описание представлений и информация последовательностей, а также собственно словарь данных. Записью информации в эти файлы управляет процесс DBWR, а данные считываются теневыми процессами.

**Файлы журналов повтора (REDO LOG FILES)** В журнале повтора регистрируются все изменения, вносимые в объекты базы данных. Эта информация применяется для создания правильного представления данных при обработке запросов, а также для выполнения отката транзакций. Записью в журнал повтора управляет процесс LGWR. Если разрешено средство архивирования (база данных работает в режиме ARCHIVELOG), то за чтение информации журналов повтора и за копирование этих журналов в автономные устройства хранения данных несет ответственность процесс ARCH.

**Управляющие файлы (CONTROL FILES)** Информация о текущем состоянии базы данных хранится в одном или нескольких управляющих файлах. Если таких файлов несколько, то все они находятся в одинаковом состоянии, т.е. *зеркально отображены* (mirrored), что упрощает процесс их восстановления в случае потери одного из управляющих файлов. Состояние базы данных определяется числом файлов данных и файлов журналов повтора, а также объемом записанной в эти файлы информации. Процесс LGWR модифицирует управляющие файлы, а процесс ARCH отвечает за регистрацию изменений в автономных устройствах хранения данных.

# Функционирование разделяемого пула

При получении базой данных SQL-оператора производится его грамматический разбор и определяется план выполнения оператора. Затем эта информация сохраняется в разделяемом пуле. Если позже база данных получает такой же SQL-оператор (выданный тем же или другим процессом пользователя), выполнять повторный грамматический разбор не требуется, так как в разделяемом пуле оператор уже хранится в нужном виде. Это намного повышает быстродействие всей системы, поскольку снижается нагрузка на сервер, особенно в тех средах, где много пользователей, работающих с одним и тем же приложением.

Когда база данных получает от клиентского приложения SQL-оператор, просматривается разделяемый пул и, если в нем уже находится такой оператор, тотчас же используются подвергшаяся грамматическому разбору форма этого оператора и план его выполнения. Если оператора нет в разделяемом пуле, то после грамматического разбора он записывается в пул. Важно отметить, что до проведения такого сравнения база данных должна получить оператор. Даже находясь в разделяемом пуле, оператор

все равно каждый раз передается по сети. Хотя после приема оператора база данных, возможно, не будет его обрабатывать, поскольку он уже в разделяемом пуле, тем не менее, сначала она должна получить весь этот оператор. Передача больших SQL-операторов может стать весьма серьезной проблемой. Указатель на местонахождение оператора, подвергшегося грамматическому разбору, возвращается клиенту после первой посылки этого оператора, поэтому при разработке клиентских приложений следует абсолютно исключить возможность проведения повторного анализа операторов. Более подробно об этом рассказано в разделе "Недопущение повторного грамматического разбора".

Кроме разобранных форм SQL-операторов, в разделяемом пуле хранятся скомпилированный p-код хранимых подпрограмм и содержимое программных каналов базы данных. При первом вызове хранимой подпрограммы p-код считывается с диска и записывается в разделяемый пул. При последующей ссылке на тот же объект обращения к диску не происходит, поскольку p-код уже находится в памяти.

Когда разделяемый пул заполняется, содержащиеся в нем объекты со временем удаляются в соответствии с алгоритмом LRU (least recently used — использовавшийся раньше всех). Объект, обращения к которому не было дольше, чем к другим, первым удаляется из пула, а занимаемое им пространство отдается другому объекту. Удаляются только объекты, не использующиеся в момент выполнения этой операции. Если, например, подпрограмма продолжает обрабатываться, то удалить ее нельзя. Различие в размерах объектов приводит к фрагментации разделяемого пула — пространства для нового объекта может быть вполне достаточно, однако оно не будет непрерывным. Если новому объекту не хватает памяти разделяемого пула, ошибка Oracle возвращается.

- `ORA-4031: unable to allocate X bytes of shared memory`
  (невозможно выделить X байт разделяемой памяти. — *Прим. пер.*)

Это происходит из-за недостаточного размера разделяемого пула или из-за того, что пространство пула стало фрагментированным.

## Сбрасывание содержимого разделяемого пула

При получении ошибки ORA-4031 возможным решением проблемы является сбрасывание (flush) содержимого разделяемого пула на диск. При этом из пула удаляются все объекты, не использующиеся в данный момент. Команда, применяемая для выполнения такой операции, выглядит так:

```
ALTER SYSTEM FLUSH SHARED POOL;
```

Выдать такую команду может любой пользователь, имеющий системную привилегию ALTER SYSTEM. Сбрасывание содержимого разделяемого пула во время работы базы данных не будет влиять на работу текущих приложений. Команда ALTER SYSTEM выдается, как правило, не из приложения, а из сеанса DBA.

Содержимое разделяемого пула сбрасывается и при остановке экземпляра. Во время запуска экземпляра разделяемый пул пуст, так как ни один SQL-оператор к этому моменту еще не обработан.

## Триггеры и разделяемый пул

P-код кэшируется только тогда, когда подпрограмма является хранимой. В версиях Oracle, предшествующих Oracle7 релиза 7.3, триггеры в число таких подпрограмм не входят. В этих версиях скомпилированный p-код триггеров не хранится в словаре данных — хранится только исходный текст. Поэтому при первой активизации триггера он должен быть скомпилирован. Затем скомпилированный код сохраняется в разделяемом пуле и используется при последующей активизации этого триггера. Однако, если триггер через некоторое время удаляется из разделяемого пула, то перед повторным выполнением он должен быть снова скомпилирован.

Рекомендуется сокращать программный текст триггеров для сведения к минимуму времени, затрачиваемого на их компиляцию. Для этого функции триггера можно передать модульной подпрограмме и из триггера вызывать модуль. В результате в триггере будет содержаться только короткий вызов модуля.

**PL/SQL 2.3 ... и ВЫШЕ**

В Oracle7 релиза 7.3 и PL/SQL 2.3 триггеры хранятся в скомпилированном виде, поэтому использовать предлагаемый выше способ необязательно. Тем не менее, вызов модульных процедур, причем не только из триггера, является достаточно удобным методом.

## Разделяемый пул и многопоточный сервер

При работе в режиме многопоточного сервера некоторая информация, относящаяся к работе конкретного сеанса и хранящаяся в обычных условиях в PGA, переносится в разделяемый пул SGA. Дело в том, что сеансы могут управляться различными теневыми процессами. Если управление сеансом переходит к другому теневому процессу, то новому теневому процессу нужна информация об этом сеансе. Таким образом, информация о сеансе должна быть доступна обоим процессам и храниться в разделяемом пуле. Следовательно, при работе в режиме многопоточного сервера разделяемый пул должен быть больше. Подробно об этом рассказано в руководстве Oracle Server Reference.

# Определение размера разделяемого пула

Размер разделяемого пула задается параметром SHARED_POOL_SIZE файла INIT.ORA. Значение по умолчанию – 3,5 Мбайт. Именно такой объем памяти выделяется при запуске экземпляра, и этот объем не будет расти или сокращаться. Если во время запуска непрерывной памяти недостаточно, база данных работать не сможет.

Поскольку размер разделяемого пула фиксирован, крайне важно устанавливать нужное значение для этого размера. Рекомендуется определить размеры часто используемых объектов и убедиться в том, что они все смогут одновременно разместиться в разделяемом пуле. Если это условие не соблюдается, то объекты будут периодически удаляться из пула и считываться в него, что ведет к фрагментации разделяемого пула и возникновению ошибок ORA-4031.

Для определения надлежащего размера разделяемого пула существуют различные методы (здесь они обсуждаться не будут – за более детальной информацией обращайтесь к руководству Oracle Server Tuning). Можно воспользоваться самым простым способом: определить размер каждого из объектов, обращение к которым производится чаще всего, и сложить полученные значения. Результатом будет минимальный размер разделяемого пула. Если применяется многопоточный сервер, то к этому результату следует добавить еще и размер памяти, в которой хранится информация о сеансах.

## Размеры хранимых подпрограмм

Размеры хранимых объектов PL/SQL фиксируются в представлении словаря данных **dba_object_size**. Ниже приведен запрос, с помощью которого определяются размеры ряда внутренних модулей.

```
SQL> SELECT name, type, code_size
  2      FROM dba_object_size
  3      WHERE name IN ('DBMS_PIPE', 'STANDARD', 'DBMS_OUTPUT')
  4      ORDER BY name, type;

NAME                           TYPE          CODE_SIZE
-----------------------------  ------------  ----------
DBMS_OUTPUT                    PACKAGE              388
DBMS_OUTPUT                    PACKAGE BODY        6217
DBMS_OUTPUT                    SYNONYM                0
DBMS_PIPE                      PACKAGE              699
DBMS_PIPE                      PACKAGE BODY        6427
DBMS_PIPE                      SYNONYM                0
STANDARD                       PACKAGE            10494
STANDARD                       PACKAGE BODY       22400
```

В столбце **code_size** этого представления содержатся сведения о размерах объектов. Можно обратиться к **dba_object_size** и запросить информацию обо всех часто используемых объектах.

## Память сеанса

О том, каким образом используется память конкретного сеанса связи с базой данных, можно узнать из представлений **v$sesstat** и **v$statname**. Для этого нужно вначале определить идентификатор сеанса с помощью запроса такого типа:

```
SQL> SELECT sid
  2      FROM v$process p, v$session s
  3      WHERE p.addr = s.paddr
  4      AND s.username = 'SYSTEM';
       SID
----------
         6
```

Запрос возвращает идентификатор сеанса, присоединившегося как пользователь Oracle с именем SYSTEM (при желании можно указать и другого пользователя). Теперь можно узнать объем памяти, используемой сеансом:

```
SQL> SELECT value
  2      FROM v$sesstat s, v$statname n
  3      WHERE s.statistic# = n.statistic#
```

```
4       AND n.name = 'session uga memory max'
5       AND SID = 6;
 VALUE
----------
 94704
```

Таким образом, объем памяти, использованной сеансом к этому моменту, составляет 94 704 байт. Такой запрос следует выдавать по прошествии некоторого времени с начала сеанса, так как запрос возвращает число, соответствующее максимальному объему той памяти, которая использовалась на данный момент времени, а не той, которая будет использоваться. Для определения необходимого размера разделяемого пула нужно умножить полученное значение на общее число сеансов.

# Закрепление объектов

Задание правильного размера — это лишь первый шаг в процессе настройки разделяемого пула. Объекты, применяющиеся особенно часто, следует *закреплять* (pin) в разделяемом пуле. Закрепленный объект не удалится из пула до тех пор, пока не будет выдана соответствующая команда. При этом неважно, насколько заполнен пул и как часто происходит обращение к находящимся в нем объектам. Закрепление объектов осуществляется с помощью модуля DBMS_SHARED_POOL. В этом модуле содержатся три процедуры: DBMS_SHARED_POOL.KEEP, DBMS_SHARED_POOL.UNKEEP и DBMS_SHARED_POOL.SIZES.

## KEEP

Процедура DBMS_SHARED_POOL.KEEP используется для закрепления объектов в пуле. Можно закреплять модули и SQL-операторы, а в Oracle8 — еще последовательности и триггеры. Описание процедуры KEEP выглядит следующим образом:

```
PROCEDURE KEEP(name VARCHAR2,
               flag CHAR DEFAULT 'P');
```

Параметры этой процедуры описаны в таблице 22.2. После того как объект закреплен, он не будет удален из пула до остановки базы данных или до выдачи команды DBMS_SHARED_POOL.UNKEEP.

**Таблица 22.2.**

| Параметр | Тип | Описание |
|---|---|---|
| *name* | VARCHAR2 | Имя объекта. Может быть именем модуля или идентификатором SQL-оператора. Идентификатор SQL-оператора — это конкатенация значений полей **address** и **hash_value** представления **v$sqlarea**; он возвращается процедурой SIZES. |
| *flag* | CHAR | Определяет тип объекта. Если значение *flag* равно 'P' (значение по умолчанию), то *name* должно соответствовать имени модуля; если же значение *flag* — 'C' (курсор), то в параметре *name* должен содержаться текст SQL-оператора. Если 'S', то *name* — это последовательность, а если 'R' — то триггер. Значения 'C', 'S' и 'R' можно задавать только в Oracle8. |

## UNKEEP

Процедура UNKEEP — это единственное средство, позволяющее удалить объект из разделяемого пула. Закрепленные объекты никогда не удаляются автоматически. Описание этой процедуры выглядит следующим образом:

```
PROCEDURE UNKEEP(name VARCHAR2,
                 flag CHAR DEFAULT 'P');
```

Аргументы аналогичны аргументам процедуры KEEP. Если указанного объекта нет в разделяемом пуле, то возвращается сообщение об ошибке.

## SIZES

С помощью этой процедуры содержимое разделяемого пула выводится на экран. Описание процедуры SIZES выглядит так:

```
PROCEDURE SIZES(minsize NUMBER);
```

Возвращаются объекты, размер которых превышает значение minsize. Для выбора информации процедура SIZES использует модуль DBMS_OUTPUT, поэтому перед ее вызовом обязательно задавайте команду SET SERVEROUTPUT ON в SQL*Plus или Server Manager. (Подробнее о модуле DBMS_OUTPUT см. в главе 13).

# Настройка SQL-операторов

Для обработки SQL-оператора необходимо определить *план его выполнения* (execution plan) – метод, посредством которого база данных будет фактически обрабатывать этот оператор. В плане выполнения определяется, какие таблицы и индексы использовать, нужно ли проводить сортировку данных и т.д. План выполнения существенно влияет на время обработки SQL-оператора. В этом разделе описаны общие способы настройки SQL-операторов, независимо от того, является оператор частью блока PL/SQL или нет.

## Определение плана выполнения

Существует несколько различных способов определения плана выполнения оператора. Самый простой – с помощью оператора EXPLAIN PLAN. При работе с утилитой TKPROF тоже выдается план выполнения, а кроме того, и дополнительные статистические показатели, касающиеся обработки SQL-оператора. Для определения плана можно воспользоваться также программой SQL-Station Plan Analyzer.

### EXPLAIN PLAN

С помощью SQL-оператора EXPLAIN PLAN определяется план выполнения указанного оператора, а полученные результаты записываются в заданную таблицу базы данных. Формат оператора EXPLAIN PLAN таков:

```
EXPLAIN PLAN [SET STATEMENT_ID = 'информация_об_операторе']
    [INTO таблица_плана] FOR sql_оператор;
```

где *sql_оператор* – оператор, план выполнения которого необходимо получить. Этот план вводится в таблицу *таблица_плана*. Если *таблица_плана* не указана, то по умолчанию ею становится таблица *plan_table*, создаваемая следующим образом (здесь statement_id – идентификатор_оператора, timestamp – временная_метка, remarks – примечания, operation – операция, options – параметры, object_node – узел_объекта, object_owner – владелец_объекта, object_name – имя_объекта, object_instance – экземпляр_объекта, object_type – тип_объекта, search_columns – столбцы_поиска, id – идентификатор, parent_id – родительский_идентификатор, position – позиция, other – другое. – *Прим. пер.*):

```
CREATE TABLE plan_table (
    Statement_id      VARCHAR2(30),
    Timestamp         DATE,
    Remarks           VARCHAR2(80),
    Operation         VARCHAR2(30),
    Options           VARCHAR2(30),
    Object_node       VARCHAR2(30),
    Object_owner      VARCHAR2(30),
    Object_name       VARCHAR2(30),
    Object_instance   NUMBER,
    Object_type       VARCHAR2(30),
    Search_columns    NUMBER,
    Id                NUMBER,
    Parent_id         NUMBER,
    Position          NUMBER,
    Other             LONG);
```

Для использования команды EXPLAIN PLAN необходимо либо иметь в своей схеме таблицу **plan_table**, либо указать другую таблицу для плана (предварительно создав ее) в EXPLAIN PLAN. Если в операторе EXPLAIN PLAN указан идентификатор оператора, то он будет записан в столбец **statement_id** таблицы **plan_table**. Это удобно для хранения нескольких планов в одной таблице – ключом каждого оператора будет его идентификатор. Если указанный идентификатор уже имеется в таблице плана, то план этого оператора заменяется.

▼ СОВЕТУЕМ

Создать таблицу plan_table можно с помощью файла **utlxplan.sql**, местонахождение которого зависит от операционной системы. В системах Unix он вместе с другими сценариями словаря данных находится в каталоге $ORACLE_HOME/rdbms/admin.

Для примера определим план выполнения запроса, в котором происходит обращение к таблицам **registered_students** и **classes**:

```
EXPLAIN PLAN
```

```
SET STATEMENT_ID = 'Query 1' FOR
    SELECT rs.course, rs.department, students.ID
      FROM registered_students rs, students
      WHERE rs.student_id = students.id
      AND students.last_name = 'Razmataz';
```

После задания этого оператора можно обратиться к таблице плана со следующим SQL-запросом:

```
SELECT LPAD(' ', 2 * (LEVEL - 1)) || operation ||
        ' ' || options || ' ' || object_name || ' ' ||
        DECODE(id, 0, 'Cost = ' || position) "Execution Plan"
FROM plan_table
START WITH id = 0
    AND statement_id = 'Query 1'
CONNECT BY PRIOR id = parent_id
    AND statement_id = 'Query 1';
```

Обратите внимание, что для оператора, указанного в EXPLAIN PLAN, и для запроса, обращенного к таблице плана, используется один и тот же идентификатор. Результат этого запроса выглядит следующим образом:

```
Execution Plan
----------------------------------------------------
SELECT STATEMENT Cost =
  NESTED LOOPS
      TABLE ACCESS FULL REGISTERED_STUDENTS
      TABLE ACCESS BY ROWID STUDENTS
        INDEX UNIQUE SCAN SYS_C00859
```

О том, как интерпретировать план выполнения оператора, будет рассказано ниже, в разделе "Использование плана".

## Утилита TKPROF

Итак, оператор EXPLAIN PLAN весьма удобен для определения плана выполнения оператора, однако с помощью утилиты TKPROF также можно получить различные статистические показатели о том, насколько хорошо SQL-оператор был выполнен на самом деле. Сначала для своего сеанса нужно создать *файл трассировки* (trace file), который затем с помощью TKPROF форматируется и преобразуется в вид, доступный для чтения. Для создания файла трассировки перед исследуемым SQL-оператором используйте SQL-команду:

```
ALTER SESSION SET SQL_TRACE=TRUE;
```

При этом будет запущен файл трассировки, и в процессе работы в него запишется информация обо всех SQL-операторах и блоках PL/SQL, передаваемых на выполнение базе данных. Информация будет загружаться в файл трассировки до завершения работы сеанса или до отключения трассировки с помощью команды

```
ALTER SESSION SET SQL_TRACE = FALSE;
```

Местонахождение файла трассировки определяется параметром USER_DUMP_DEST файла INIT.ORA. Имя файла трассировки зависит от используемой операционной системы; обычно оно начинается с ora и включает в свой состав идентификатор теневого процесса, например, выглядит так: ora_12345.trc. Самый простой способ узнать имя файла трассировки — просмотреть параметр USER_DUMP_DEST и определить имя самого нового файла после выполнения трассировки.

▼ ОСТОРОЖНО В блоке PL/SQL нельзя использовать оператор ALTER SESSION так как он не является оператором DML. Однако ALTER SESSION можно применять перед выполнением блока, а также трассировать SQL-операторы вне блока.

В качестве примера выполним в SQL*Plus следующий оператор:

```
SQL> ALTER SESSION SET SQL_TRACE = TRUE;
Session altered.
SQL> SELECT rs.course, rs.department, students.ID
        FROM registered_students rs, students
        WHERE rs.student_id = students.id
```

```
        AND students.last_name = 'Razmataz';
   COURSE DEP         ID
--------- ---- -----------
      101 HIS      10010
      307 NUT      10010
SQL> ALTER SESSION SET SQL_TRACE = FALSE;
Session altered.
```

Результатом выполнения оператора будет создание файла ora_29338.trc в USER_DUMP_DEST. Теперь нужно отформатировать этот файл трассировки, чтобы его можно было прочитать. Для этого используем утилиту TKPROF, запускаемую на выполнение из операционной системы. Ее местонахождение зависит от системы, но обычно она находится в том же каталоге, что и другие исполняемые программы Oracle, такие как SQL*Plus. Для TKPROF используется следующий формат:

TKPROF *входной_файл выходной_файл* [SORT = *параметры_сортировки*]
[PRINT = *отображаемое_число*]
[EXPLAIN = *пользователь/пароль*]

где *входной_файл* — имя сгенерированного файла трассировки (в данном случае ora_29338.trc), а *выходной_файл* — это отформатированный файл трассировки. Если параметры SORT не указаны, то SQL-операторы будут отображаться в заданном порядке. Иначе, можно отсортировать их с помощью одного или нескольких параметров, перечень которых приведен в таблице 22.3. Для указания нескольких параметров сортировки используйте следующий синтаксис:

SORT = (*параметр1, параметр2, ...*)

## Таблица 22.3. Параметры сортировки TKPROF

| Параметр сортировки | Описание |
|---|---|
| PRSCNT | Число грамматических разборов (PARSE) |
| PRSCPU | Время ЦП, потраченное на грамматический разбор |
| PRSELA | Полное время грамматического разбора |
| PRSDSK | Число операций чтения с диска во время грамматического разбора |
| PRSQRY | Число правильных операций чтения блоков во время грамматического разбора |
| PRSCU | Число операций чтения текущего блока во время грамматического разбора |
| PRSMIS | Число кэш-промахов при обращении к библиотечному кэшу во время грамматического разбора |
| EXECNT | Число обработок (EXECUTE) |
| EXECPU | Время ЦП, потраченное на обработку |
| EXEELA | Полное время обработки |
| EXEDSK | Число операций физического чтения с диска во время обработки |
| EXEQRY | Число правильных операций чтения блоков во время обработки |
| EXECU | Число операций чтения текущего блока во время обработки |
| EXEROW | Число обрабатываемых строк |
| EXEMIS | Число кэш-промахов при обращении к библиотечному кэшу во время обработки |
| FCHCNT | Число считываний (FETCH) |
| FCHCPU | Время ЦП, потраченное на считывание |
| FCHELA | Полное время считывания |
| FCHDSK | Число операций физического чтения с диска во время считывания |
| FCHQRY | Число правильных операций чтения блоков во время считывания |
| FCHCU | Число операций чтения текущего блока во время считывания |
| FCHROW | Число считанных строк |

Для включения плана выполнения в файл трассировки нужно указать имя и пароль пользователя в параметре EXPLAIN. TKPROF создаст таблицу плана, выполнит EXPLAIN PLAN, запишет полученные результаты в эту таблицу, считает полученный результат в файл и удалит таблицу. Если указан параметр *отображаемое_число*, то именно столько операторов будет включено в этот файл после сортировки.

Отформатируем файл трассировки:

```
TKPROF ora_29338.trc trace.out EXPLAIN=example/example
```

Результатом выполнения этой команды будет файл trace.out, фрагмент которого приведен ниже.

```
SELECT rs.course, rs.department, students.ID
      FROM registered_students rs, students
      WHERE rs.student_id = students.id
      AND students.last_name = 'Razmataz';
Call     count  cpu    elapsed    disk    query    current    rows
------   ------ -----  -------    -----   -----    -------    -----
Parse        1  0.00   0.00       0       0        0          0
Execute      1  0.00   0.00       0       0        0          0
Fetch        1  0.00   0.00       0       55       3          2
------   ------ -----  -------    -----   -----    -------    -----
Total        3  0.00   0.00       0       55       3          2

Misses in library cache during parse: 1
Optimizer hint: CHOOSE
Parsing user id: 9 (EXAMPLE)
Rows     Execution Plan
-------  ---------------------------------------------------
0 SELECT STATEMENT OPTIMIZER HINT: CHOOSE
      2    NESTED LOOPS
     18       TABLE ACCESS (FULL) OF 'REGISTERED_STUDENTS'
     18       TABLE ACCESS (BY ROWID) OF 'STUDENTS'
     18          INDEX (UNIQUE SCAN) OF 'SYS_C00859' (UNIQUE)
********************************************************************
```

В результирующий файл включены сам оператор, статистические показатели его выполнения, а также план его выполнения. В данном случае на проведение грамматического разбора время ЦП (cpu) не затрачивалось, поскольку оператор был уже обработан и находился в разделяемом пуле (о том, какие операторы находятся в определенный момент в разделяемом пуле, можно узнать из представлений словаря данных **v$sql** и **v$sqlarea**). Однако заметьте, что число грамматических разборов равно 1, так как оператор грамматического разбора был вызван в базе данных.

**▼ ВНИМАНИЕ** В файле трассировки будут всегда отображаться рекурсивные SQL-операторы, автоматически генерируемые рассматриваемым оператором. Для любого оператора Oracle может создать до шести рекурсивных операторов, выполняющих такие операции, как проверка установок NLS и контроль соответствия объектных привилегий. Поскольку в файле трассировки отображаются все SQL-операторы, рекурсивные и все остальные, он может быть полезен для отладки программ.

## SQL-Station Plan Analyzer

Один из компонентов программы SQL-Station, называемый Plan Analyzer (анализатор планов), сочетает в себе функциональные возможности оператора EXPLAIN PLAN и утилиты TKPROF. При первом запуске Plan Analyzer и соединении с базой данных на экране появляются два окна. В верхнем окне можно ввести SQL-оператор, а в нижнем отображается план выполнения этого оператора. Выше окна SQL находится панель инструментов для выбора различных режимов работы оптимизатора. Экран Plan Analyzer с введенным SQL-оператором показан на рис. 22.2.

**▼ СОВЕТУЕМ** Вводя оператор, не завершайте его точкой с запятой. Если указана точка с запятой, она посылается базе данных вместе с оператором. В результате возвращается сообщение об ошибке ORA-911 (неверный символ).

Для просмотра плана щелкните клавишей мыши на соответствующей кнопке панели инструментов или выберите нужный план в меню Plan. Можно выбрать один из трех заданных по умолчанию режи-

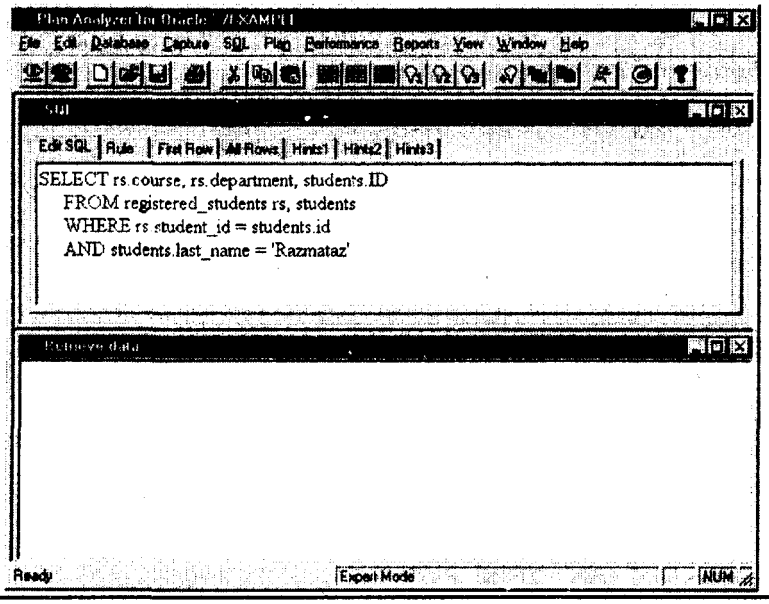



**Рис. 22.2.**
*Plan Analyzer с введенным SQL-оператором*

мов работы оптимизатора или предоставить свой собственный план с помощью кнопок-подсказок (Hint). Режимы по умолчанию – это режим работы оптимизатора на основе правил (Rule), а также стоимостные режимы First Row (первая строка) и All Rows (все строки). План оптимизатора для выполнения SQL-оператора в режиме работы на основе правил показан на рис. 22.3.

Для просмотра статистических показателей выполнения оператора выберите пункт Statistics в меню Performance. Это позволит просмотреть показатели работы оптимизатора в разных режимах и сравнить результаты либо графически, либо в текстовой форме. Окно Statistics с результатами работы оптимизатора в режиме Rule показано на рис. 22.4. Результаты работы в другом режиме можно увидеть, щелкнув мышью на соответствующей вкладке в этом окне. Затем с помощью кнопки Compare можно провести сравнение разных режимов.

Здесь продемонстрированы лишь немногие из средств, предоставленных Plan Analyzer. В их число входят средство анализа объектов базы данных, используемых в конкретном операторе, и пошаговое выполнение плана с объяснением каждого шага. Более подробную информацию о Plan Analyzer и о его возможностях можно получить во встроенной документации на программу SQL-Station.

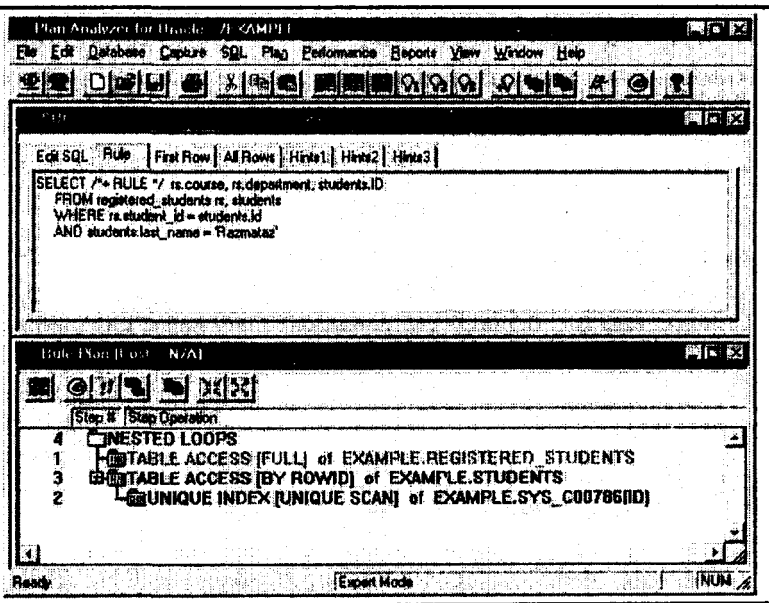



**Рис. 22.3.**
*План выполнения в режиме работы на основе правил*

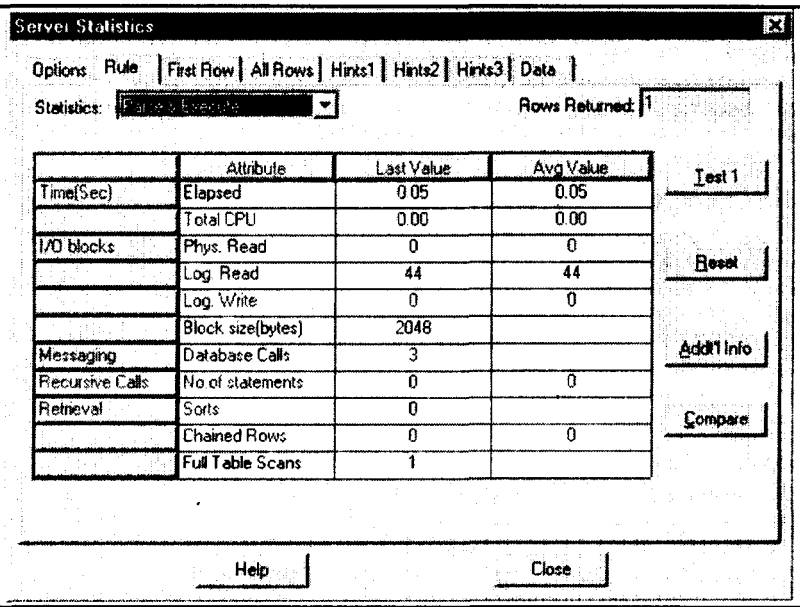



**Рис. 22.4.**
*Статистические показатели выполнения оператора*

## Использование плана

План выполнения для заданного SQL-оператора и фиксированного режима оптимизации не зависит от того, как он был получен. План состоит из отдельных *операций* (operations). Операции (будь то полный просмотр таблицы или просмотр таблицы, организованной по индексу) выполняются по отдельности. Результатом выполнения каждой операции является набор строк, определяющийся обращением к таблице или индексу либо использованием результатов, полученных после выполнения другой операции. Результаты последней операции – это *результирующий набор* (result set) запроса. В процессе выполнения запроса каждая из операций вносит свой определенный вклад. Полное обсуждение операций и их эффективности не является предметом данной книги. Здесь дается лишь краткий анализ основных операций. За более подробной информацией обращайтесь к руководству Oracle Server Tuning.

Вышеописанный план выполнения состоит из трех операций: NESTED LOOP (вложенный цикл) и двух разновидностей операции TABLE ACCESS (обращение к таблице).

### NESTED LOOP

Результатом выполнения операции NESTED LOOP является комбинация результирующих наборов двух других операций: **a** и **b**. Строки в операциях a и b сопоставляются с некоторым условием и те, которые удовлетворяют условию, сохраняются. В данном случае условием является условие соединения (join), заданное в операторе: rs.student_id = students.id. Всякий раз при использовании соединения для его обработки требуется операция NESTED LOOP.

### TABLE ACCESS(FULL)

При полном (FULL) просмотре таблицы считываются все строки этой таблицы. В данном случае для таблицы **registered_students** полный просмотр необходим, так как для нее не создано ни одного индекса.

### TABLE ACCESS(BY ROWID)

Эта операция является самым быстрым способом выбора конкретной строки. В данном случае результаты операции INDEX SCAN (индексный просмотр) передаются операции TABLE ACCESS BY ROWID (обращение к таблице по идентификатору строки). Индексный просмотр разрешен, поскольку на столбце **ID** таблицы **students** определен индекс.

# Работа с сетью

После создания SQL-оператора его структура определяет план выполнения оператора и результирующую производительность системы. Но сначала SQL-оператор необходимо послать базе данных, т.е. передать от процесса пользователя теневому процессу. Даже если процесс пользователя и теневой процесс функционируют на одной и той же машине, SQL-оператор или блок PL/SQL должен быть передан от одного другому. В большинстве приложений основная часть времени тратится на пересылку

данных по сети, поэтому снижение нагрузки на сеть является первоочередной задачей при настройке приложения. Для уменьшения интенсивности сетевого трафика применяется три способа: использование клиентского PL/SQL, недопущение повторного грамматического разбора операторов и применение интерфейса массивов Oracle. Хотя последние два способа применяются в основном при работе с OCI и предкомпиляторами Oracle, тем не менее, они вполне пригодны и для PL/SQL.

## Использование клиентского PL/SQL

Написание приложения при помощи программного пакета Developer или Designer 2000 означает, что на станции клиента установлена собственная система поддержки PL/SQL. Эта среда выполнения программ подробно рассмотрена в главе 13. Любая работа, выполняемая на клиентской стороне, снижает нагрузку на сервер. Кроме того, если с одним приложением одновременно работают несколько пользователей, они могут обрабатывать данные на своих клиентских станциях параллельно, не замедляя функционирования сервера.

Клиентскую систему поддержки PL/SQL следует использовать максимально эффективно. Скажем, проверку достоверности входных данных можно проводить и до посылки информации на сервер.

## Недопущение повторного грамматического разбора

Когда SQL-оператор или блок PL/SQL передается со станции клиента на сервер, клиент может сохранять ссылку на оператор, подвергшийся грамматическому разбору. Такой ссылкой является область данных курсора в случае использования OCI или элемент кэша курсора при работе с предкомпиляторами. Если в приложении некоторый оператор выполняется неоднократно, то требуется провести только его один — начальный — грамматический разбор. Впоследствии можно использовать преобразованную форму этого оператора, выполняя его с различными значениями переменных привязки.

Этот метод применяется в основном в OCI и предкомпиляторах, поскольку они обеспечивают достаточно высокий уровень управления процессом обработки курсоров. В OCI курсоры управляются непосредственно через область данных курсора (cursor data area). Можно явным образом провести грамматический разбор (**oparse**) оператора и выполнить (**oxec**) его. В предкомпиляторах процесс обработки операторов управляется параметрами HOLD_CURSOR и RELEASE_CURSOR.

Непосредственно в PL/SQL удобнее всего пользоваться этим методом, работая с модулем DBMS_SQL, интерфейс которого схож с OCI. После грамматического разбора оператора (DBMS_SQL.PARSE) он может быть выполнен несколько раз (см. главу 15).

## Обработка массивов

В предкомпиляторах и OCI данные могут посылаться и считываться с помощью базовых массивов. Этот метод называется интерфейсом массивов Oracle. Интерфейс массивов является очень удобным средством, поскольку с его помощью можно пересылать по сети большие объемы информации как единое целое, вместо нескольких посылок выполняя всего лишь одну. Например, считать 100 строк можно за одну операцию, которая вернет все необходимые 100 строк; в противном случае пришлось бы выполнять 100 операций считывания, каждая из которых возвращала бы только одну строку. Такой подход применяется в SQL*Plus.

Непосредственно в PL/SQL интерфейс массивов Oracle не применяется, поскольку здесь массивы хранятся не так, как базовые. Таблицы PL/SQL реализованы иначе (см. главу 3). Однако по мере возможности следует использовать этот интерфейс. Если команды PL/SQL задаются из OCI или из предкомпиляторов, применяйте интерфейс массивов для обработки других SQL-операторов приложения. За более подробной информацией о работе с базовыми массивами обращайтесь к документации на предкомпиляторы и/или OCI.

# Итоги

Грамотное приложение разрабатывается с учетом обеспечения высокой производительности системы. В этой главе были рассмотрены различные аспекты настройки и производительности приложений. В начале главы обсуждены структура экземпляра Oracle и процесс выполнения SQL-операторов. Попадая на сервер, оператор обрабатывается в соответствии с планом, определяющимся с помощью оператора EXPLAIN PLAN и утилиты TKPROF. Кроме того, здесь было рассказано о том, как минимизировать сетевой трафик, используя клиентский PL/SQL, избегая повторного грамматического разбора операторов и применяя интерфейс массивов Oracle.

# Приложение А

## *Зарезервированные слова PL/SQL*

Слова, перечисленные в данном приложении, являются зарезервированными словами PL/SQL; они имеют специальное синтаксическое значение в этом языке и поэтому не могут быть использованы в качестве идентификаторов (имен переменных, имен процедур и т.д.). Некоторые из этих слов зарезервированы также и в SQL, поэтому их нельзя указывать в качестве имен объектов базы данных, таких как таблицы, последовательности и представления.

В таблице А.1 перечислены зарезервированные слова, применяемые в PL/SQL вплоть до версии 8 включительно. В более ранних версиях зарезервированными являются не все эти слова, однако их использования следует избегать, поскольку они будут конфликтовать при работе с версией 8.

**Таблица А.1**

| | | | |
|---|---|---|---|
| ABORT | ACCEPT | ACCESS* | ADD* |
| ALL* | ALTER* | AND* | ANY* |
| ARRAY | ARRAYLEN | AS* | ASC* |
| ASSERT | ASSIGN | AT | AUDIT* |
| AUTHORIZATION | AVG | BASE_TABLE | BEGIN |
| BETWEEN* | BINARY_INTEGER | BODY | BOOLEAN |
| BY* | CASE | CHAR* | CHAR_BASE |
| CHECK* | CLOSE | CLUSTER* | CLUSTERS |
| COLAUTH | COLUMN* | COMMENT* | COMMIT |
| COMPRESS* | CONNECT* | CONSTANT | CRASH |
| CREATE* | CURRENT* | CURRVAL | CURSOR |
| DATABASE | DATA_BASE | DATE* | DBA |
| DEBUGOFF | DEBUGON | DECLARE | DECIMAL* |
| DEFAULT* | DEFINITION | DELAY | DELETE* |
| DESC* | DIGITS | DISPOSE | DISTINCT* |
| DO | DROP* | ELSE* | ELSIF |
| END | ENTRY | EXCEPTION | EXCEPTION_INIT |
| EXCLUSIVE* | EXISTS* | EXIT | FALSE |
| FETCH | FILE* | FLOAT* | FOR* |
| FORM* | FROM* | FUNCTION | GENERIC |
| GOTO | GRANT* | GROUP* | HAVING* |
| IDENTIFIED* | IF | IMMEDIATE* | IN* |
| INCREMENT* | INDEX* | INDEXES | INDICATOR |
| INITIAL* | INSERT* | INTEGER* | INTERFACE |
| INTERSECT* | INTO* | IS5* | LEVEL* |
| LIKE* | LIMITED | LOCK* | LONG* |
| LOOP | MAX | MAXEXTENTS* | MIN |
| MINUS | MLSLABEL | MOD | MODE* |
| NATURAL | NATURALN | NEW | NEXTVAL |
| NOAUDIT* | NOCOMPRESS* | NOT* | NOWAIT* |
| NULL* | NUMBER* | NUMBER_BASE | OF* |
| OFFLINE* | ON* | ONLINE* | OPEN |
| OPTION* | OR* | ORDER* | OTHERS |
| OUT | PACKAGE | PARTITION | PCTFREE* |
| PLS_INTEGER | POSITIVE | POSITIVEN | PRAGMA |
| PRIOR* | PRIVATE | PRIVILEGES* | PROCEDURE |
| PUBLIC* | RAISE | RANGE | RAW* |
| REAL | RECORD | REF | RELEASE |
| REMR | RENAME* | RESOURCE* | RETURN |
| REVERSE | REVOKE* | ROLLBACK | ROW* |
| ROWID* | ROWLABEL* | ROWNUM | ROWS* |
| ROWTYPE | RUN | SAVEPOINT | SCHEMA |

* Зарезервировано и в SQL

**Таблица A.1 (продолжение)**

| | | | |
|---|---|---|---|
| SELECT* | SEPARATE | SESSION* | SET* |
| SHARE* | SMALLINT* | SPACE | SQL |
| SQLCODE | SQLERRM | STATE* | STATEMENT |
| STTDEV | SUBTYPE | SUCCESSFUL* | SUM |
| SYNONYM* | SYSDATE* | TABAUTH | TABLE* |
| TABLES* | TASK | TERMINATE | THEN* |
| TO* | TRIGGER* | TRUE | TYPE |
| UID* | UNION* | UNIQUE* | UPDATE* |
| USE | USER* | VALIDATE* | VALUES* |
| VARCHAR* | VARCHAR2* | VARIANCE | VIEW* |
| VIEWS | WHEN | WHENEVER* | WHERE* |
| WHILE | WITH* | WORK | WRITE |
| XOR | | | |

* Зарезервировано и в SQL

Если необходимо воспользоваться зарезервированным словом, его нужно заключить в двойные кавычки. Например, этот блок вполне корректен:

```
DECLARE
  "BEGIN" NUMBER;
BEGIN
  "BEGIN" := 7;
END;
```

Хотя этот блок и правильный, создавать такие блоки не рекомендуется (см. главу 2).

# Приложение В

## *Руководство по работе со встроенными модулями DBMS*

В этом приложении кратко описаны встроенные модули PL/SQL, а также процедуры и функции, содержащиеся в них.

## Создание модулей

Все вспомогательные модули принадлежат пользователю базы данных с именем SYS, однако для них созданы общие синонимы, поэтому модули можно вызывать, не предваряя имя префиксом SYS. Для вызова процедур и функций этих модулей пользователь, который не является пользователем SYS, должен иметь полномочие EXECUTE на конкретный модуль. Создаются модули с помощью сценария создания словаря данных – **catproc.sql**. Дополнительную информацию можно найти и во встроенной документации на Oracle – каждый из модулей создается в отдельном файле. Местонахождение этих файлов зависит от применяемой операционной системы; в системах Unix, например, они расположены в каталоге $ORACLE_HOME/rdbms/admin. Файлы создания модулей снабжены комментариями, сообщающими о правилах использования каждого из модулей.

Каждый из модулей обычно хранится в двух файлах операционной системы – в одном находится заголовок модуля, а во втором – тело. Программный текст тела модуля, как правило, кодирован посредством программы-оболочки.

Перечень модулей, описанных в этом приложении, приведен в таблице B-1.

**Таблица B.1.**

| Имя модуля | Описание |
|---|---|
| DBMS_ALERT [1] | Синхронное взаимодействие соединений (сеансов) |
| DBMS_APPLICATION_INFO | Регистрация приложений для трассировки |
| DBMS_AQ и DBMS_AQADM | Управление средством Oracle8 Advanced Queuing |
| DBMS_DEFER, DBMS_DEFER_SYS и DBMS_DEFER_QUERY [2] | Создание отложенных вызовов удаленных процедур и управление этими вызовами |
| DBMS_DDL | PL/SQL-эквиваленты для некоторых команд DDL |
| DBMS_DESCRIBE | Описание хранимых подпрограмм |
| DBMS_JOB [1] | Планирование выполнения процедур PL/SQL |
| DBMS_LOB | Работа с объектами LOB в Oracle8 |
| DBMS_LOCK | Блокировки, определяемые пользователями |
| DBMS_OUTPUT [1] | Вывод информации на экран в SQL*Plus или в Server Manager |
| DBMS_PIPE [1] | Асинхронное взаимодействие соединений (сеансов) |
| DBMS_REFRESH и DBMS_SNAPSHOT [2] | Работа с моментальными снимками |
| DBMS_REPCAT, DBMS_REPCAT_AUTH и DBMS_REPCAT_ADMIN [2] | Работа со средством симметричного тиражирования Oracle |
| DBMS_ROWID [2] | Получение информации из идентификаторов строк (ROWID). Преобразование ROWID Oracle7 в ROWID Oracle8, и наоборот |
| DBMS_SESSION | PL/SQL-эквивалент команды ALTER SESSION |
| DBMS_SHARED_POOL [1] | Управление разделяемым пулом |
| DBMS_SQL [1] | Динамические PL/SQL и SQL |
| DBMS_TRANSACTION | Команды управления транзакциями |
| DBMS_UTILITY | Дополнительные служебные процедуры |
| UTL_FILE [1] | Файловый ввод/вывод |

[1] Этот модуль рассмотрен выше; здесь дано только краткое описание.
[2] Полный анализ этого модуля не является предметом данной книги; здесь дано только краткое описание.

## Описание модулей

Далее описан каждый из модулей; за описанием модуля следует описание составляющих его процедур и функций. Некоторые из модулей, а также некоторые из процедур этих модулей применяются только в конкретных версиях PL/SQL, что отмечено особо.

## DBMS_ALERT

Модуль DBMS_ALERT применяется для обмена сообщениями между сеансами, соединенными с одной и той же базой данных. Оповещения (alerts) синхронны, то есть они посылаются при завершении транзакций. При откате транзакции оповещение не посылается (см. главу 16).

## DBMS_APPLICATION_INFO

Модуль DBMS_APPLICATION_INFO применяется для регистрации информации (info) о конкретной программе в системной таблице **v$session**. Затем с помощью этой информации можно определить, какие приложения (applications) функционируют в некоторый момент времени и какие действия они выполняют. В частности, модуль DBMS_APPLICATION_INFO можно использовать для считывания и модификации столбцов **module, action** и **client_info** таблицы **v$session**, содержащих сведения о текущем сеансе.

В столбце **module** хранится имя приложения. Например, SQL*Plus устанавливает в этом столбце значение 'SQL*Plus'. В столбце **action** хранится фрагмент приложения, выполняющийся в данный момент. Можно установить определенное значение **action** перед введением, скажем, фрагмента программы Pro*C. В столбец **client_info** можно вносить произвольные строки символов.

### SET_MODULE

Эта процедура используется для установки значений module и action в текущем сеансе:

```
PROCEDURE SET_MODULE(module_name IN VARCHAR2,
                     action_name IN VARCHAR2);
```

где *module_name* — имя текущего приложения, а *action_name* — имя выполняемого фрагмента. Значение module ограничено 48 байтами, а значение action — 32 байтами; значения, превышающие эти пределы, усекаются.

### READ_MODULE

Эта процедура используется для считывания значений **module** и **action** для текущего сеанса. Такую же информацию можно получить, если обратиться с запросом к таблице **v$session**, но по умолчанию просматривать непосредственно эту таблицу имеют право только пользователи с привилегией DBA. Описание процедуры READ_MODULE выглядит следующим образом:

```
PROCEDURE READ_MODULE(module_name OUT VARCHAR2,
                      action_name OUT VARCHAR2);
```

где *module_name* — это значение module, установленное последним с помощью SET_MODULE, а *action_name* — значение action, установленное последним с помощью SET_MODULE или SET_ACTION.

### SET_ACTION

Процедура SET_ACTION используется для установки значения **action** только в поле, соответствующем текущему сеансу. Эту процедуру следует применять перед вводом фрагмента программы:

```
PROCEDURE SET_ACTION(action_name IN VARCHAR2);
```

где *action_name* — имя текущего фрагмента, ограниченное 32 байтами; значения, превышающие этот предел, усекаются. Когда выполнение данного фрагмента программы заканчивается, рекомендуется устанавливать **action** в NULL.

### SET_CLIENT_INFO

Эта процедура используется для записи значения **client_info** текущего сеанса в таблицу **v$session**. В Oracle это значение не применяется, однако оно может быть запрошено администратором базы данных (DBA). Значением может стать имя текущего пользователя или имя машины, определяемое клиентской программой. Описание этой процедуры выглядит следующим образом:

```
PROCEDURE SET_CLIENT_INFO(client_info IN VARCHAR2);
```

где *client_info* — новое значение для этого поля. Оно ограничено 64 байтами; значения, превышающие этот предел, усекаются.

### READ_CLIENT_INFO

Процедура READ_CLIENT_INFO возвращает значение, установленное последним с помощью SET_CLIENT_INFO:

PROCEDURE READ_CLIENT_INFO(*client_info* OUT VARCHAR2);

где *client_info* – значение, установленное последним с помощью SET_CLIENT_INFO.

## DBMS_AQ и DBMS_AQADM

PL/SQL 8.0 ... и ВЫШЕ

Эти модули применяются для управления системой улучшенной организации очередей (Oracle Advanced Queuing). Посредством DBMS_AQ реализуются операции постановки в очередь и вывода из очереди, а DBMS_AQADM позволяет управлять очередями и таблицами очередей (см. главу 17).

## DBMS_DEFER, DBMS_DEFER_SYS и DBMS_DEFER_QUERY

PL/SQL 8.0 ... и ВЫШЕ

Эти модули позволяют создавать отложенные (defer) вызовы удаленных процедур и управлять этими вызовами. Полное обсуждение данных модулей не является темой этой книги (см. руководство Oracle8 Server Replication).

## DBMS_DDL

В модуле DBMS_DDL предлагаются PL/SQL-эквиваленты для некоторых наиболее полезных команд DDL, которые нельзя использовать непосредственно в PL/SQL. Хотя для выполнения таких команд можно применять модуль DBMS_SQL, несколько проще они описаны в DBMS_DDL.

### ALTER_COMPILE

Эта процедура эквивалентна SQL-командам ALTER PROCEDURE COMPILE, ALTER PACKAGE COMPILE, ALTER PACKAGE BODY COMPILE и ALTER FUNCTION COMPILE:

```
PROCEDURE ALTER_COMPILE(
   type VARCHAR2,
   schema VARCHAR2,
   name VARCHAR2);
```

Параметры этой процедуры описаны в таблице B.2. ALTER_COMPILE может устанавливать одну из следующих ошибок:

- ORA-20000: Insufficient privileges or object does not exist.
  (недостаточные привилегии, или объект не существует. – *Прим. пер.*)
- ORA-20001: Remote object, cannot compile.
  (удаленный объект; компиляция невозможна. – *Прим. пер.*)
- ORA-20002: Bad value for object type.
  (неверное значение для объектного типа. – *Прим. пер.*)

**Таблица B.2.**

| Параметр | Тип | Описание |
|---|---|---|
| *type* | VARCHAR2 | Тип компилируемого объекта: 'PROCEDURE', 'FUNCTION' или 'PACKAGE BODY' |
| *schema* | VARCHAR2 | Схема, которой принадлежит объект (с учетом регистра символов) |
| *name* | VARCHAR2 | Имя компилируемого объекта (с учетом регистра символов) |

### ANALYZE_OBJECT

Эта процедура эквивалентна SQL-командам ANALYZE TABLE, ANALYZE CLUSTER и ANALYZE INDEX:

```
PROCEDURE ANALYZE_OBJECT(
   type VARCHAR2,
   schema VARCHAR2,
   name VARCHAR2,
   method VARCHAR2,
   estimate_rows NUMBER DEFAULT NULL,
   estimate_percent NUMBER DEFAULT NULL);
```

Параметры этой процедуры описаны в таблице B.3.

**Таблица В.3.**

| Параметр | Тип | Описание |
|---|---|---|
| *type* | VARCHAR2 | Тип анализируемого объекта: 'TABLE', 'CLUSTER' или 'INDEX' |
| *schema* | VARCHAR2 | Схема, которой принадлежит объект (с учетом регистра символов) |
| *name* | VARCHAR2 | Имя анализируемого объекта (с учетом регистра символов) |
| *method* | VARCHAR2 | Метод анализа: NULL или 'ESTIMATE' (оценка). Если 'ESTIMATE', то одно из значений *estimate_rows* и *estimate_percent* должно быть ненулевым |
| *estimate_rows* | NUMBER | Число оцениваемых строк |
| *estimate_percent* | NUMBER | Процент оцениваемых строк. Если значение *estimate_rows* не равно нулю, этот параметр игнорируется |

# DBMS_DESCRIBE

В модуле DBMS_DESCRIBE содержится только одна процедура – DESCRIBE_PROCEDURE. Она при указании имени хранимой процедуры или функции возвращает информацию о параметрах данной подпрограммы. Если подпрограмма переопределяется как часть модуля, возвращается информация обо всех переопределяемых вариантах. В описании процедуры DESCRIBE_PROCEDURE используются два табличных типа:

```
TYPE varchar2_table IS TABLE OF VARCHAR2(30)
   INDEX BY BINARY_INTEGER;
TYPE number_table IS TABLE OF NUMBER
   INDEX BY BINARY_INTEGER;
```

## DESCRIBE_PROCEDURE

```
PROCEDURE DESCRIBE_PROCEDURE(
   object_name IN VARCHAR2,
   reserved1 IN VARCHAR2,
   reserved2 IN VARCHAR2,
   overload OUT number_table,
   position OUT number_table,
   level OUT number_table,
   argument_name OUT varchar2_table,
   datatype OUT number_table,
   default_value OUT number_table,
   in_out OUT number_table,
   length OUT number_table,
   precision OUT number_table,
   scale OUT number_table,
   radix OUT number_table,
   spare OUT number_table);
```

Имя характеризуемой процедуры указывается параметром *object_name*. Сведения обо всех параметрах возвращаются в таблицу PL/SQL. Так, имя первого параметра – *argument_name*(1), а его тип данных – *datatype*(1); имя второго параметра *argument_name*(2) и т.д. Такие же задачи выполняет процедура OCI, называемая **odessp**. Параметры процедуры DESCRIBE_PROCEDURE описаны в таблице В.4.

**Таблица В.4.**

| Параметр | Тип данных | Описание |
|---|---|---|
| *object_name* | VARCHAR2 | Характеризуемая процедура или функция. Она может принадлежать другой схеме или находиться в составе модуля. |
| *reserved1* | VARCHAR2 | В настоящее время не используется; передается как NULL. |
| *reserved2* | VARCHAR2 | В настоящее время не используется; передается как NULL. |

## Таблица B.4. (продолжение)

| Параметр | Тип данных | Описание |
|---|---|---|
| *overload* | number_table | Порядковый номер для переопределяемых процедур. Для информации о первом варианте overload = 0, о втором варианте — *overload* = 1 и т.д. |
| *position* | number_table | Позиция в списке аргументов. Значения, возвращаемые функциями, имеют позицию 0. |
| *level* | number_table | Для составных типов, например для записей и таблиц, значение *level* увеличивается на 1 с каждым уровнем вложенности. Если объявление записи находится на уровне 2, то поля этой записи располагаются на уровне 3. |
| *argument_name* | varchar2_table | Имя данного параметра |
| *datatype* | number_table | Код типа данных для параметра. Применяемые коды перечислены в таблице B.5. Обратите внимание на то, что коды подтипов равны кодам их базовых типов; так, INTEGER и REAL являются подтипами типа NUMBER и код каждого из них равен 2. |
| *default_value* | number_table | 1, если аргумент имеет значение по умолчанию; 0, если не имеет. |
| *in_out* | number_table | Вид параметра: 0 — IN, 1 — OUT, 2 — IN OUT. |
| *length* | number_table | Размер аргумента (для CHAR и VARCHAR2). |
| *precision* | number_table | Точность аргумента (для NUMBER). |
| *scale* | number_table | Масштаб аргумента (для NUMBER). |
| *radix* | number_table | Основание системы счисления аргумента (для NUMBER). |
| *spare* | number_table | В настоящее время не применяется — этот параметр зарезервирован для будущего использования. |

## Таблица B.5. Коды типов данных, используемые в процедуре DBMS_DESCRIBE.DESCRIBE_PROCEDURE

| Код типа данных | Тип данных |
|---|---|
| 1 | VARCHAR2 |
| 2 | NUMBER |
| 3 | BINARY_INTEGER |
| 8 | LONG |
| 11 | ROWID |
| 12 | DATE |
| 23 | RAW |
| 24 | LONG RAW |
| 96 | CHAR |
| 106 | MLSLABEL |
| 250 | Запись PL/SQL |
| 251 | Таблица PL/SQL |
| 252 | BOOLEAN |

## DBMS_JOB

PL/SQL 2.2 ... и выше

Модуль DBMS_JOB используется для планирования выполнения заданий PL/SQL в определенное время в фоновом режиме, то есть фоновыми процессами. Задание (job) – это хранимая процедура. Если задание выполняется неудачно, то PL/SQL пытается выполнить это задание до 16 раз, пока оно не завершится успешно (см. главу 18).

## DBMS_LOB

PL/SQL 8.0 ... и выше

Модуль DBMS_LOB используется для работы с четырьмя типами больших объектов (LOBs – Large OBjects) Oracle8: CLOB, BLOB, NCLOB и BFILE. Функциям модуля DBMS_LOB эквивалентен ряд функций Oracle8 OCI (см. главу 21).

## DBMS_LOCK

Модуль DBMS_LOCK используется для создания пользовательских блокировок (locks). Управление такими блокировками производится точно так же, как и другими блокировками Oracle. Это значит, что их можно просматривать в постоянных представлениях словаря данных. Пользовательские блокировки начинаются с UL, поэтому они не конфликтуют с блокировками Oracle (см. руководства Oracle Server Application Developer's Guide и Oracle Server Reference).

В некоторых из процедур этого модуля задаются режимы блокирования, указываемые определенными номерами. Эти номера и их значения приведены в таблице В.6.

### Таблица В.6. Идентификаторы режимов блокирования

| Идентификатор | Смысл |
|---|---|
| 1 | Режим NULL |
| 2 | Режим разделения строк (UL Row Share, ULRS) |
| 3 | Исключающий режим блокирования строк (UL Row Exclusive, ULRX) |
| 4 | Режим разделения (UL Share, ULS) |
| 5 | Исключающий режим разделения строк (UL Row Share Exclusive, ULRSX) |
| 6 | Исключающий режим (Exclusive, ULX) |

### ALLOCATE_UNIQUE

Процедура по заданному имени блокировки порождает уникальный идентификатор блокировки.

```
PROCEDURE ALLOCATE_UNIQUE(
   lockname IN VARCHAR2,
   lockhandle OUT VARCHAR2,
   expiration_secs IN INTEGER DEFAULT 864000);
```

Описатель идентификатора блокировки возвращается параметром *lockhandle*, размер значения которого лежит в пределах до 128 байт. Идентификаторы блокировок – это числа в диапазоне от 0 до 1073741823. В последующих вызовах используются либо идентификаторы блокировок, либо описатели блокировок. Эта процедура всегда выполняет оператор COMMIT. Значение *expiration_secs* указывает минимальное время в секундах, после которого блокировка снимается.

▼ ОСТОРОЖНО

Имена блокировок, начинающиеся с $ORA, зарезервированы для использования Oracle. Имя блокировки имеет максимальный размер в 128 байт и учитывает регистр символов.

### REQUEST

```
FUNCTION REQUEST(id IN INTEGER,
   lockmode IN INTEGER DEFAULT X_MODE,
   timeout IN INTEGER DEFAULT MAXWAIT,
   release_on_commit IN BOOLEAN DEFAULT FALSE)
 RETURN INTEGER;
```

```
FUNCTION REQUEST(lockhandle IN VARCHAR2,
   lockmode IN INTEGER DEFAULT X_MODE,
   timeout IN INTEGER DEFAULT MAXWAIT,
   release_on_commit IN BOOLEAN DEFAULT FALSE)
 RETURN INTEGER;
```

Эта функция используется для запроса блокировки конкретного вида. Функция REQUEST переопределяется первым параметром — описателем или идентификатором блокировки. Параметры этой функции и значения, возвращаемые ей, описаны в таблицах B.7 и B.8.

**Таблица B.7.**

| Параметр | Тип | Описание |
|---|---|---|
| *id* | INTEGER | Идентификатор запрашиваемой блокировки; диапазон значений — от 0 до 1073741823. |
| *lockhandle* | VARCHAR2 | Описатель блокировки, возвращаемый процедурой ALLOCATE_UNIQUE. Можно указывать либо *id*, *либо lockhandle*, но не оба параметра сразу. |
| *lockmode* | INTEGER | Запрашиваемый режим блокировки. Разрешенные значения перечислены в таблице B.6. |
| *timeout* | INTEGER | Максимальное время (в секундах) ожидания предоставления блокировки. Если за этот период времени блокировка не может быть предоставлена, REQUEST возвращает 1. |
| *release_on_commit* | BOOLEAN | Если TRUE, блокировка снимается после завершения (COMMIT) транзакции; если FALSE, блокировка сохраняется до тех пор, пока не будет снята явно. |

**Таблица B.8.**

| Значение, возвращаемое функцией REQUEST | Смысл |
|---|---|
| 0 | Успешное выполнение |
| 1 | Тайм-аут |
| 2 | Обнаружен тупик (взаимоблокировка) |
| 3 | Ошибка параметра |
| 4 | Уже владеет указанной блокировкой |
| 5 | Запрещенный описатель блокировки |

## CONVERT

```
FUNCTION CONVERT(id IN INTEGER,
   lockmode IN INTEGER,
   timeout IN NUMBER DEFAULT MAXWAIT,
 RETURN INTEGER;
FUNCTION CONVERT(lockhandle IN VARCHAR2,
   lockmode IN INTEGER,
   timeout IN NUMBER DEFAULT MAXWAIT)
 RETURN INTEGER;
```

Эта функция используется для изменения режима блокирования. Аргументы и возвращаемое значение те же, что и в REQUEST, и описаны в таблицах B.9 и B.10. Как и REQUEST, эта продукция переопределяется первым параметром.

**Таблица B.9.**

| Параметр | Тип | Описание |
|---|---|---|
| *id* | INTEGER | Идентификатор блокировки, присвоенный пользователем; диапазон значений — от 0 до 1073741823. |
| *lockhandle* | VARCHAR2 | Описатель блокировки, возвращаемый процедурой ALLOCATE_UNIQUE. Можно указывать либо *id*, либо *lockhandle*, но не оба параметра сразу. |

**Таблица B.9. (продолжение)**

| Параметр | Тип | Описание |
|---|---|---|
| *lockmode* | INTEGER | Запрашиваемый режим блокировки (см. таблицу B.6). |
| *timeout* | NUMBER | Максимальное время (в секундах) ожидания изменения режима. |

**Таблица B.10.**

| | |
|---|---|
| 0 | Успешное выполнение |
| 1 | Тайм-аут |
| 1 | Тайм-аут |
| 2 | Обнаружен тупик (взаимоблокировка) |
| 3 | Ошибка параметра |
| 4 | Не владеет указанной блокировкой |
| 5 | Запрещенный описатель блокировки |

### RELEASE

FUNCTION RELEASE(*id* IN INTEGER) RETURN INTEGER;
FUNCTION RELEASE(*lockhandle* IN VARCHAR2) RETURN INTEGER;

Эта функция снимает блокировку, установленную функцией REQUEST. Функция RELEASE переопределяется типом аргумента – блокировка может быть указана либо по идентификатору, либо по описателю. Возвращаемые значения приведены в таблице B.11.

**Таблица B.11.**

| Значение, возвращаемое функцией RELEASE | Смысл |
|---|---|
| 0 | Успешное выполнение |
| 3 | Ошибка параметра |
| 4 | Не владеет указанной блокировкой |
| 5 | Запрещенный описатель блокировки |

### SLEEP

PROCEDURE SLEEP(*seconds* IN NUMBER);

Процедура SLEEP переводит текущее соединение в режим ожидания на указанное время (в секундах). Максимальное разрешение составляет сотые доли секунды, поэтому значение параметра seconds может быть дробным.

## DBMS_OUTPUT

Посредством модуля DBMS_OUTPUT (в комбинации с SQL*Plus или Server Manager) для PL/SQL реализуются ограниченные возможности по выводу информации, что удобно при тестировании и отладке программ PL/SQL (см. главу 14).

## DBMS_PIPE

Модуль DBMS_PIPE похож на модуль DBMS_ALERT и также дает возможность сеансам взаимодействовать друг с другом. Однако сообщения, посылаемые по программным каналам (pipes), асинхронны. Если сообщение послано, оно будет доставлено по адресу даже в случае отката транзакции, пославшей это сообщение (см. главу 16).

## DBMS_REFRESH и DBMS_SNAPSHOT

Эти модули используются для работы с моментальными снимками (snapshots) и группами моментальных снимков в распределенной среде. DBMS_REFRESH применяется для создания групп моментальных снимков, регенерация (refresh) которых может выполняться одновременно, а DBMS_SNAPSHOT позволяет регенерировать моментальные снимки, не являющиеся частью групп (см. документацию по Oracle).

## DBMS_REPCAT, DBMS_REPCAT_AUTH и DBMS_REPCAT_ADMIN

Эти модули используются для управления средством симметричного тиражирования Oracle. DBMS_REPCAT позволяет работать с этим средством, а DBMS_REPCAT_AUTH и DBMS_REPCAT_ADMIN – управлять им (см. документацию по Oracle).

## DBMS_ROWID

Модуль DBMS_ROWID используется для преобразования форматов ROWID (идентификаторов строк) Oracle7 и Oracle8 (см. руководство Oracle8 Server SQL Reference).

## DBMS_SESSION

Команда ALTER SESSION является оператором DDL, поэтому ее нельзя применять непосредственно в PL/SQL. Модуль DBMS_SESSION реализует некоторые из вариантов ALTER SESSION, позволяя вызывать их из блоков PL/SQL. В качестве альтернативы DBMS_SESSION можно использовать модуль DBMS_SQL, поскольку он дает возможность выполнять произвольные операторы, в том числе и ALTER SESSION.

### SET_ROLE

PROCEDURE SET_ROLE(*role_cmd* VARCHAR2);

Процедура SET_ROLE эквивалентна SQL-команде SET ROLE. Текст, находящийся в *role_cmd*, добавляется к 'SET ROLE', а затем полученная строка символов выполняется. Поскольку в хранимых процедурах роли запрещены, вызов SET_ROLE в хранимой подпрограмме или в триггере никаких действий не вызовет. Если для роли задан пароль, его нужно включить в вызов. Например, с помощью следующего оператора разрешается роль **Administrator** с паролем admin:

```
DBMS_SESSION.SET_ROLE('Administrator IDENTIFIED BY admin');
```

### SET_SQL_TRACE

PROCEDURE SET_SQL_TRACE(*sql_trace* BOOLEAN);

Эта процедура используется для разрешения и запрещения SQL-трассировки и эквивалентна команде ALTER SESSION SET SQL_TRACE = *sql_трассировка* (см. главу 22).

### SET_NLS

PROCEDURE SET_NLS(*param* VARCHAR2, *value* VARCHAR2);

Эта процедура эквивалентна команде ALTER SESSION SET *параметр = значение*, где *параметр* – это правильный параметр NLS, а *значение* – значение, устанавливаемое для этого параметра. Применение этой процедуры в триггерах запрещено. Аргументы *param и value* используются непосредственно в результирующей команде ALTER SESSION, поэтому, когда значение *value* является текстовым литералом, оно должно включать в свой состав дополнительные одиночные кавычки. Например, можно изменить формат дат следующим образом:

```
DBMS_SESSION.SET_NLS('nls_date_format' ,
                     '''DD-MON-YY HH24:MI:SS''');
```

*Обратите внимание на одиночные кавычки в описании строки формата.*

### CLOSE_DATABASE_LINK

PROCEDURE CLOSE_DATABASE_LINK(*dblink* VARCHAR2);

Эта процедура эквивалентна команде ALTER SESSION CLOSE DATABASE LINK связь_баз_данных. Она закрывает неявное соединение с удаленной базой данных.

### SET_LABEL

PROCEDURE SET_LABEL(*lbl* VARCHAR2);

Эта процедура применяется в Trusted Oracle и эквивалентна команде ALTER SESSION SET LABEL = *метка*. Значением аргумента *lbl* может быть 'DBHIGH', 'DBLOW' или другая текстовая метка.

### SET_MLS_LABEL_FORMAT

```
PROCEDURE SET_MLS_LABEL_FORMAT(fmt VARCHAR2);
```

Эта процедура также применяется в Trusted Oracle и эквивалентна команде ALTER SESSION SET MLS_LABEL_FORMAT = *формат*. Она изменяет формат по умолчанию для текущего сеанса.

### RESET_PACKAGE

```
PROCEDURE RESET_PACKAGE;
```

Для процедуры RESET_PACKAGE эквивалентной SQL-команды не существует. Она освобождает память, используемую для хранения состояния модуля, и отменяет подтверждение всех модулей для сеанса. Такая ситуация возникает во время начала сеанса.

### UNIQUE_SESSION_ID

```
FUNCTION UNIQUE_SESSION_ID RETURN VARCHAR2;
```

Эта функция возвращает строку размером максимум 24 байта, которая уникальна среди всех сеансов, соединенных с базой данных в этот момент. Несколько вызовов UNIQUE_SESSION_ID из одного и того же сеанса всегда возвращают одинаковый результат. SQL-эквивалента не существует.

IS_ROLE_ENABLED

```
FUNCTION IS_ROLE_ENABLED(rolename VARCHAR2)
  RETURN BOOLEAN;
```

Эта функция возвращает TRUE, если роль с именем *rolename* разрешена для сеанса, и FALSE в противном случае. Если функция IS_ROLE_ENABLED вызывается из хранимой подпрограммы или из триггера, то она всегда возвращает FALSE, так как использование ролей здесь запрещено.

## DBMS_SHARED_POOL

Модуль DBMS_SHARED_POOL используется для работы с *разделяемым пулом* (shared pool). В разделяемом пуле можно закреплять модули и процедуры так, что они с течением времени из пула не удаляются. Настройка разделяемого пула — ключевой момент процесса оптимизации среды PL/SQL (см. главу 22).

## DBMS_SQL

PL/SQL 2.1 ... и ВЫШЕ

Посредством модуля DBMS_SQL реализуется динамический PL/SQL. С помощью этого модуля можно во время выполнения программы конструировать SQL-операторы и блоки PL/SQL, а затем обрабатывать их. Кроме того, DBMS_SQL можно использовать для выполнения в PL/SQL операторов DDL, выполнение которых другим способом запрещено (см. главу 15).

## DBMS_TRANSACTION

В модуле DBMS_TRANSACTION находятся процедуры, управляющие транзакциями. Многим из них соответствуют эквивалентные SQL-команды PL/SQL, которые также приведены здесь для полноты обсуждения.

### Команды SET TRANSACTION

```
PROCEDURE READ_ONLY;
PROCEDURE READ_WRITE;
PROCEDURE USE_ROLLBACK_SEGMENT(rb_name VARCHAR2);
```

Эти процедуры эквивалентны SQL-командам SET TRANSACTION READ ONLY, SET TRANSACTION READ WRITE и SET TRANSACTION USE ROLLBACK SEGMENT *имя_сегмента_отката*. Каждая из них должна выполняться как первый оператор транзакции.

### Команды ALTER SESSION ADVISE

```
PROCEDURE ADVISE_COMMIT;
PROCEDURE ADVISE_ROLLBACK;
PROCEDURE ADVISE_NOTHING;
```

Эти процедуры эквивалентны командам ALTER SESSION ADVISE COMMIT, ALTER SESSION ADVISE ROLLBACK и ALTER SESSION ADVISE NOTHING. Они применяются для посылки совета (advice) распределенной транзакции. В случае неопределенного состояния транзакции такой совет будет находиться в столбце **advice** представления словаря данных **dba_2pc_pending** удаленной базы данных.

## Команды COMMIT

```
PROCEDURE COMMIT;
PROCEDURE COMMIT_COMMENT(cmnt VARCHAR2);
PROCEDURE COMMIT_FORCE(
   xid VARCHAR2,
   scn VARCHAR2 DEFAULT NULL);
```

Эти процедуры эквивалентны SQL-командам COMMIT, COMMIT COMMENT комментарий и COMMIT FORCE *xid, scn*. COMMIT COMMENT и COMMIT FORCE используются обычно в распределенных транзакциях.

## Команды ROLLBACK и SAVEPOINT

```
PROCEDURE SAVEPOINT(savept VARCHAR2);
PROCEDURE ROLLBACK;
PROCEDURE ROLLBACK_SAVEPOINT(savept VARCHAR2);
PROCEDURE ROLLBACK_FORCE(xid VARCHAR2);
```

Эти процедуры эквивалентны SQL-командам SAVEPOINT точка_сохранения, ROLLBACK, ROLLBACK TO SAVEPOINT *точка_сохранения* и ROLLBACK FORCE *xid*. *xid* – это идентификатор локальной или глобальной транзакции. ROLLBACK FORCE используется обычно в распределенных транзакциях.

## BEGIN_DISCRETE_TRANSACTION

```
PROCEDURE BEGIN_DISCRETE_TRANSACTION;
```

Эта процедура применяется для пометки текущей транзакции как дискретной. Дискретная транзакция может выполняться быстрее, чем обычная, так как информация отмены не записывается. Все изменения, вносимые в базу данных, буферизуются и реально используются во время выполнения оператора COMMIT. На дискретные транзакции налагается ряд ограничений, поэтому применять их следует с осторожностью (см. руководство Oracle Server Application Developer's Guide).

## PURGE_MIXED

```
PROCEDURE PURGE_MIXED(xid VARCHAR2);
```

Эта процедура используется для уничтожения смешанных транзакций. Смешанные транзакции – это такие распределенные транзакции, в которых работа, выполненная некоторыми узлами, завершена, а для других узлов произведен откат. Администратор базы данных (DBA) должен применять процедуру PURGE_MIXED с осторожностью. Для параметра xid следует устанавливать идентификатор транзакции, хранящийся в столбце **local_tran_id** представления **dba_2pc_pending**.

## LOCAL_TRANSACTION_ID

```
FUNCTION LOCAL_TRANSACTION_ID(
   create_transaction BOOLEAN DEFAULT FALSE)
   RETURN VARCHAR2;
```

Эта функция возвращает уникальный идентификатор текущей транзакции или NULL, если текущая транзакция отсутствует. Идентификатор уникален для локального экземпляра. Если параметр *create_transaction* истинен (TRUE), то транзакция создается (если она уже не существует).

## STEP_ID

```
FUNCTION STEP_ID RETURN NUMBER;
```

Эта функция возвращает уникальное положительное целое число, которое упорядочивает операции DML текущей транзакции. Это значение уникально только для текущей транзакции.

# DBMS_UTILITY

В модуле DBMS_UTILITY предоставляются дополнительные функциональные возможности для управления процедурами, выдачи сообщений об ошибках и другой информации.

## COMPILE_SCHEMA

```
PROCEDURE COMPILE_SCHEMA(schema VARCHAR2);
```

Эта функция компилирует все процедуры, функции и модули в указанной схеме и эквивалентна SQL-командам ALTER PROCEDURE COMPILE, ALTER FUNCTION COMPILE и ALTER PACKAGE COMPILE. Если пользователю не предоставлены привилегии ALTER на один или несколько объектов схемы *schema*, то устанавливается ошибка ORA-20000.

## ANALYZE_SCHEMA

```
PROCEDURE ANALYZE_SCHEMA(
   schema VARCHAR2,
   method VARCHAR2,
   estimate_rows NUMBER DEFAULT NULL,
   estimate_percent NUMBER DEFAULT NULL);
```

Эта процедура анализирует все таблицы, кластеры и индексы в схеме. Ее параметры описаны в таблице В.12.

**Таблица В.12.**

| Параметр | Тип | Описание |
|---|---|---|
| *schema* | VARCHAR2 | Схема, объекты которой анализируются |
| *method* | VARCHAR2 | Метод анализа: NULL или 'ESTIMATE' (оценка). Если 'ESTIMATE', то одно из значений *estimate_rows* и *estimate_percent* должно быть ненулевым |
| *estimate_rows* | NUMBER | Число оцениваемых строк |
| *estimate_percent* | NUMBER | Процент оцениваемых строк. Если указано значение *estimate_rows*, то этот параметр игнорируется |

## FORMAT_ERROR_STACK

```
FUNCTION FORMAT_ERROR_STACK RETURN VARCHAR2;
```

Эта функция возвращает весь стек ошибок, максимальный размер которого составляет 2000 байт. Ее удобно применять в обработчиках исключительных ситуаций.

## FORMAT_CALL_STACK

```
FUNCTION FORMAT_CALL_STACK RETURN VARCHAR2;
```

Эта функция возвращает строку символов, содержащую текущий стек вызовов — все выполняемые в это время процедуры. Максимальный размер стека — 2000 байт.

## IS_PARALLEL_SERVER

```
FUNCTION IS_PARALLEL_SERVER RETURN BOOLEAN;
```

Эта функция возвращает TRUE, если экземпляр работает в режиме параллельного сервера, и FALSE — в противном случае.

## GET_TIME

```
FUNCTION GET_TIME RETURN NUMBER;
```

Возвращает время в сотых долях секунды. Полезна для хронометрирования процедуры. Например:

```
DECLARE
   v_Start NUMBER;
   v_End NUMBER;
BEGIN
   v_Start := DBMS_UTILITY.GET_TIME;
   /* Выполним некоторые операции. */
   v_End := DBMS_UTILITY.GET_TIME;
   /* Выполнение работы заняло (v_Start - v_End) * 100 секунд. */
END;
```

## NAME_RESOLVE

```
PROCEDURE NAME_RESOLVE(
   name IN VARCHAR2,
   context IN NUMBER,
   schema OUT VARCHAR2,
   part1 OUT VARCHAR2,
   part2 OUT VARCHAR2,
   dblink OUT VARCHAR2,
```

*part1_type* OUT NUMBER,
*object_number* OUT NUMBER);

Процедура NAME_RESOLVE используется для разбора указанной ссылки на элементы. Если ввести, скажем, example.Debug.Reset@dblink, то результатом будут отдельные значения 'example', 'Debug', 'Reset' и dblink, возвращаемые соответственно в *schema, part1, part2* и *dblink*. Параметры этой процедуры описаны в таблице B.13.

**Таблица B.13.**

| Параметр | Тип данных | Описание |
|---|---|---|
| name | VARCHAR2 | Имя обрабатываемого объекта |
| context | NUMBER | Зарезервировано для дальнейшего использования; должно быть равно 1 |
| schema | VARCHAR2 | Схема объекта, если указана в имени name |
| part1 | VARCHAR2 | Первая часть имени. Если объект представляет собой модульную процедуру, то это имя модуля. Если объект — не модульная процедура, то это полное имя. Оно определяется также типом *part1_type* |
| part2 | VARCHAR2 | Вторая часть имени (если применяется) |
| dblink | VARCHAR2 | Связь баз данных (если применяется) |
| part1_type | NUMBER | Определяет смысл значения part1:<br>5 — синоним;<br>7 — процедура;<br>8 — функция;<br>9 — модуль |
| object_number | NUMBER | Если объект успешно разобран, то это номер объекта, записанный в словаре данных |

## PORT_STRING

FUNCTION PORT_STRING RETURN VARCHAR2;

Эта функция возвращает строку символов, которая однозначно идентифицирует версию Oracle и версию операционной системы. Максимальная длина строки зависит от операционной системы.

## UTL_FILE

PL/SQL 2.3 ... и ВЫШЕ

Модуль UTL_FILE реализует в PL/SQL файловый ввод/вывод. При помощи этого модуля программы PL/SQL могут считывать информацию из файлов операционной системы, находящихся на сервере, и записывать информацию в такие файлы. В целях безопасности доступные файлы и каталоги ограничены параметрами файла инициализации базы данных INIT.ORA (см. главу 18).

# Приложение С

## Глоссарий средств PL/SQL

В этом приложении кратко описаны основные средства PL/SQL; для каждого средства дается ссылка на главу, в которой оно описано более детально. Перечень средств PL/SQL приведен в таблице C.1 (все пункты глоссария расположены в алфавитном порядке (английского алфавита. — *Прим. пер.*).

## Таблица C.1. Средства PL/SQL

| | | |
|---|---|---|
| ACCESS_INTO_NULL — исключительная ситуация | ACCESS_INTO_NULL — исключительная ситуация | Blocks (блоки) |
| COLLECTION_IS_NULL — исключительная ситуация | Collections (сборные конструкции) | Comments (комментарии) |
| COMMIT — оператор | Conditions (условия) | CURSOR_ALREADY_OPEN — исключительная ситуация |
| Cursors (курсоры) | Cursor Variables (курсорные переменные) | Datatypes (типы данных) |
| DDL | DELETE — оператор | DML |
| DUP_VAL_ON_INDEX — исключительная ситуация | EXCEPTION_INIT — прагма | Exceptions (исключительные ситуации) |
| EXIT — оператор | Expressions (выражения) | External procedures (внешние процедуры) |
| extproc | FETCH — оператор | %FOUND — атрибут |
| Functions (функции) | GOTO — оператор | Identifiers (идентификаторы) |
| IF — оператор | INSERT — оператор | INVALID_CURSOR — исключительная ситуация |
| INVALID_NUMBER — исключительная ситуация | I/O (ввод/вывод) | %ISOPEN — атрибут |
| Literals (литералы) | LOBs (большие объекты) | LOCK TABLE — оператор |
| LOGIN_DENIED — исключительная ситуация | Loops (циклы) | MAP & ORDERS — методы |
| Nested Tables (вложенные таблицы) | NO_DATA_FOUND — исключительная ситуация | %NOTFOUND — атрибут |
| NOT_LOGGED_ON — исключительная ситуация | NULL — оператор | NULL — значение |
| OPEN — оператор | Packages (модули) | Procedural Statements (процедурные операторы) |
| Procedures (процедуры) | PROGRAM_ERROR — исключительная ситуация | RAISE — оператор |
| Records (записи) | RESTRICT_REFERENCES — прагма | RETURN — оператор |
| ROLLBACK — оператор | %ROWCOUNT — атрибут | %ROWTYPE — атрибут |
| ROWTYPE_MISMATCH — исключительная ситуация | SAVEPOINT — оператор | SELECT..INTO — оператор |
| SET TRANSACTION — оператор | SQLCODE — функция | SQL Cursor (SQL-курсор) |
| SQLERRM — функция | SQL Statements (SQL-операторы) | STANDARD — модуль |
| Statements (операторы) | STORAGE_ERROR — исключительная ситуация | SUBSCRIPT_BEYOND_COUNT — исключительная ситуация |
| SUBSCRIPT_OUTSIDE_LIMIT — исключительная ситуация | Tables (таблицы) | TIMEOUT_ON_RESOURCE — исключительная ситуация |
| TOO_MANY_ROWS — исключительная ситуация | TRANSACTION_BACKED_OUT — исключительная ситуация | Triggers (триггеры) |
| %TYPE — атрибут | UPDATE — оператор | VALUE_ERROR — исключительная ситуация |
| Variables & Constants (переменные и константы) | VARRAYs (изменяемые массивы) | ZERO_DIVIDE — исключительная ситуация |

PL/SQL 8.0 ... и ВЫШЕ

**ACCESS_INTO_NULL — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-6530: Reference to uninitialized composite" (ссылка на неинициализированный составной тип). Она устанавливается при попытке обращения к атрибуту или методу объекта, который является NULL-значением (см. главу 11).

**Assignment (присваивание)** Операторы присваивания используются для замещения переменной PL/SQL некоторым значением:

*переменная := выражение*;

где *переменная* – это идентификатор переменной PL/SQL, а *выражение* – выражение PL/SQL. Выражение называется *значением выражения* (rvalue), а переменная – *именующим значением* (lvalue). Если выражение и переменная имеют разные типы, PL/SQL пытается преобразовать тип выражения к типу переменной. Если преобразование невозможно, устанавливается ошибка (см. главу 2).

**Blocks (блоки)** Все программы PL/SQL построены из блоков. Блок PL/SQL состоит из раздела объявлений, выполняемого раздела и раздела исключительных ситуаций:

```
DECLARE
   — раздел объявлений
BEGIN
   — выполняемый раздел
EXCEPTION
   — раздел исключительных ситуаций
END;
```

Разделы ограничиваются ключевыми словами DECLARE, BEGIN, EXCEPTION и END. Обязателен только выполняемый раздел (см. главу 2).

PL/SQL 8.0 ... и ВЫШЕ

**COLLECTION_IS_NULL — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-6531: Reference to uninitialized collection" (ссылка на неинициализированную сборную конструкцию). Она устанавливается при попытке добавить или модифицировать элемент вложенной таблицы или изменяемого массива (см. главу 12).

**Collections (сборные конструкции)** Сборная конструкция – это группа переменных PL/SQL, с которыми можно работать либо по отдельности, либо как с единым целым. В PL/SQL версии 2 существует только один тип сборных конструкций – таблица PL/SQL. В PL/SQL 8.0 к набору сборных конструкций добавлены вложенные таблицы и изменяемые массивы (см. главы 3 и 12).

**Comments (комментарии)** Комментарии используются для документирования программного текста и для его удобочитаемости. Компилятор PL/SQL игнорирует комментарии. Существует два вида комментариев: однострочные и многострочные, или комментарии С-типа. Однострочные комментарии начинаются с двух символов тире (--) и продолжаются до конца строки, ограниченной символом новой строки. Многострочные комментарии начинаются с /* и заканчиваются */ (см. главу 2). Например:

```
-- Это однострочный комментарий.
/* Эо многострочный комментарий,
   продолжтающийся на второй строке. */
```

**COMMIT — оператор** SQL-оператор COMMIT применяется для завершения транзакции и фиксации сделанных изменений. До завершения транзакции другие пользователи не могут видеть изменения, внесенные этой транзакцией в базу данных. COMMIT снимает также все блокировки, установленные транзакцией. Синтаксис оператора COMMIT таков:

```
COMMIT [WORK];
```

причем ключевое слово WORK необязательно (см. главу 4).

**Conditions (условия)** Условие – это выражение, которое принимает логическое значение. Условия применяются в операторах IF..THEN, EXIT..WHEN, WHILE..LOOP, а также в условии WHERE SQL-операторов. Условия можно объединять посредством логических операций AND, OR и NOT. Условия создаются с использованием логических операций, например =, >=, LIKE, IN или BEETWEEN (см. главу 2).

**CURSOR_ALREADY_OPEN — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-6511:PL/SQL: cursor already open" (курсор уже открыт). Она устанавливается при попытке открыть уже открытый курсор. Определить, открыт ли курсор, можно с помощью атрибута %ISOPEN (см. главу 6).

**Cursors (курсоры)** Курсоры используются для управления процессом обработки запросов, возвращающих несколько строк. Курсор объявляется с помощью оператора CURSOR..IS, а обрабатывается посредством операторов OPEN, FETCH и CLOSE. Курсорные атрибуты применяются для определения текущего состояния курсора, получения сведений о числе строк, возвращаемых курсором, о том, открыт ли курсор и успешно ли завершилась последняя операция считывания данных (см. главу 6).

PL/SQL 8.0 ... и ВЫШЕ

**Cursor Variables (курсорные переменные)** Курсорные переменные — это динамические курсоры. С помощью оператора OPEN..FOR курсорную переменную можно открыть для различных запросов. Обычно курсорная переменная открывается на сервере, затем из нее считывается информация; после этого курсорная переменная закрывается на станции клиента. Курсорные переменные применяются в PL/SQL версии 2.2 и выше. В PL/SQL 2.2 для обращения к курсорным переменным необходимо применять клиентскую программу, например SQL*Plus или программу, написанную с помощью предкомпилятора или OCI. В PL/SQL 2.3 курсорные переменные можно полностью обрабатывать на сервере. Они объявляются с типом REF CURSOR — единственным типом-указателем, применяемым в PL/SQL версий, предшествующих 2.3 (см. главу 6).

**Datatypes (типы данных)** В PL/SQL поддерживаются все типы данных, применяемые сервером Oracle, а также ряд дополнительных типов. Типы подробно описаны в главе 2, а в таблице C.2 они перечислены по категориям и семействам. Элементы этой таблицы, обозначенные *, применяются только в PL/SQL версии 8.0 и выше.

**Таблица C.2.**

| Категория | Семейство | Типы |
|---|---|---|
| Ссылочные типы | — | REF CURSOR, REF объектный тип* |
| Составные типы | — | RECORD, TABLE, VARRAY* |
| Типы LOB* | | BFILE, BLOB, CLOB, NCLOB |
| Скалярные типы | Числовое | BINARY_INTEGER, DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INT, INTEGER, NATURAL, NATURALN*, NUMBER, NUMERIC, PLS_INTEGER, POSITIVE, POSITIVEN*, REAL, SIGNTYPE*, SMALLINT |
| | Логическое | BOOLEAN |
| | Trusted | MLSLABEL |
| | Символьное | CHAR, CHARACTER, LONG, NCHAR*, NVARCHAR2*, STRING, VARCHAR, VARCHAR2 |
| | Типы без обработки | RAW, LONG RAW |
| | Временное | DATE |
| | Rowid | ROWID |

**DDL (data definition language — язык определения данных)** С помощью операторов DDL *модифицируются объекты словаря данных, а не информация, содержащаяся в этих объектах.* Такими операторами являются, например, операторы CREATE TABLE и DROP PROCEDURE. В PL/SQL операторы DDL нельзя использовать непосредственно, так как компилятор работает по методу ранней привязки. Однако в PL/SQL 2.1 можно применять операторы DDL в модуле DBMS_SQL, позволяющем создавать операторы не на этапе компиляции, а на этапе выполнения программы (см. главу 15).

**DELETE — оператор** SQL-оператор DELETE применяется для удаления строк данных из таблицы:

DELETE [FROM] *таблица* [*псевдоним*]
WHERE *условие* | CURRENT OF *курсор*;

где *таблица* — это таблица, строки которой удаляются. При необходимости можно указать псевдоним для имени таблицы. *Условие* определяет, какие строки нужно удалить. Если используется конструкция CURRENT OF *курсор*, то удаляться будет последняя строка, считанная из курсора. После удаления одной или нескольких строк в атрибуте SQL%NOTFOUND содержится FALSE, в SQL%FOUND — TRUE, а в SQL%ROWCOUNT — число удаленных строк. Если условие не соответствует ни одной строке, то они не

удаляются, и теперь в SQL%NOTFOUND содержится TRUE, в SQL%FOUND – FALSE, а значение SQL%ROWCOUNT равно 0 (см. главу 4).

**DML (data manipulation language — язык манипулирования данными)** С помощью операторов DML (SELECT, UPDATE, DELETE, INSERT, EXPLAIN PLAN) модифицируются данные, содержащиеся в объектах Oracle, а не сами объекты. Операторы DML можно использовать непосредственно в PL/SQL (см. главу 4).

**DUP_VAL_ON_INDEX — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-1: unique constraint violation" (нарушение ограничения уникальности). Она устанавливается в том случае, когда в таблицу, для некоторого поля которой создан уникальный индекс, вводится строка, где для этого поля указано значение, уже существующее в таблице.

**EXCEPTION_INIT — прагма** Прагма EXCEPTION_INIT используется для связывания именованной исключительной ситуации с ошибкой Oracle. При этом именованная исключительная ситуация добавляется к стандартным:

PRAGMA EXCEPTION_INIT (*имя_исключительной_ситуации, номер_ошибки*);

где *имя_исключительной_ситуации* – это исключительная ситуация в области своего действия, а *номер_ошибки* – значение функции SQLCODE, соответствующее ошибке Oracle (см. главу 10). Ниже приведен пример установления исключительной ситуации **e_NonExistentTable** при возникновении ошибки "ORA-942: table or view does not exist" (таблица или представление не существует).

```
DECLARE
  e_NonExistentTable  EXCEPTION;
  PRAGMA  EXCEPTION_INIT(e_NonExistentTable,  -942);
  ...
```

**Exceptions (исключительные ситуации)** Исключительные ситуации используются для обнаружения ошибок во время выполнения программ. Когда происходит ошибка, устанавливается исключительная ситуация, и управление программой немедленно передается в раздел обработки исключительных ситуаций блока. Если в текущем блоке такого раздела нет, то исключительная ситуация передается во внешний блок. Стандартные исключительные ситуации определены в модуле STANDARD, и, кроме того, пользователи могут создавать собственные исключительные ситуации (см. главу 10).

**EXIT — оператор** Оператор EXIT применяется для передачи управления программой из цикла, выполняющегося в данный момент:

EXIT [*имя_цикла*] [WHEN *условие*];

где *условие* – некоторое логическое выражение. Если условия WHEN нет, то выход из цикла производится немедленно. Если условие WHEN указано, то выход из цикла производится только когда *условие* истинно. Если указано *имя_цикла*, то оно должно соответствовать метке в начале цикла (см. главу 2).

**Expressions (выражения)** Выражение – это комбинация переменных, констант, литералов, знаков операций, других выражений и вызовов функций. Используется вычисленное значение выражения (rvalue). Выражения можно применять в различных операторах PL/SQL, в том числе в условии WHERE операторов SELECT, UPDATE и DELETE. Выражения комбинируются с помощью знаков операций: +, -, NOT, OR, ||. Если элементы выражения имеют разные типы, то сначала эти типы преобразуются к одному, и только потом выполняются нужные операции (см. главу 2).

PL/SQL 8.0 ... и ВЫШЕ

**External procedures (внешние процедуры)** Внешние процедуры – это новое средство Oracle, которое позволяет создавать процедуры и функции на языке C и вызывать их непосредственно из PL/SQL. Для этого необходимы процедуры-оболочки PL/SQL, создающиеся с помощью оператора CREATE [OR REPLACE] PROCEDURE...AS EXTERNAL. Кроме того, в операторе CREATE нужно устанавливать соответствие между типами данных PL/SQL и C (см. главу 20).

PL/SQL 8.0 ... и ВЫШЕ

**extproc** Внешние процедуры вызываются специальным процессом Oracle, называемым extproc. Процесс extproc стартует в тот момент, когда сеанс первый раз вызывает внешнюю процедуру, и остается активным в течение всего времени работы сеанса. Процесс extproc загружает разделяемую библиотеку, содержащую внешнюю процедуру, а затем вызывает процедуру. Он взаимодействует с исходным теневым процессом при помощи SQL*Net (см. главу 20).

**FETCH — оператор** Оператор FETCH используется для считывания строк из курсора или курсорной переменной в переменные PL/SQL или в запись PL/SQL:

```
FETCH курсор | курсорная переменная
INTO запись | список_переменных;
```

где *курсор* — это имя ранее открытого курсора, а *курсорная переменная* — имя ранее открытой курсорной переменной (в PL/SQL 2.2 и выше). *Список переменных* или поля *записи* должны соответствовать списку выбора запроса. Обычно FETCH вызывается в цикле, условие выхода из которого задается атрибутом %NOTFOUND. С каждым вызовом FETCH считывается по одной строке; при этом значение курсор%ROWCOUNT увеличивается на единицу (см. главу 6).

**%FOUND — атрибут** Этот логический курсорный атрибут используется для определения, считалась ли из курсора или курсорной переменной строка во время последней операции FETCH (в PL/SQL 2.2 и выше):

```
курсор%FOUND | курсорная переменная%FOUND
```

где *курсор* — это имя курсора, а *курсорная переменная* — имя курсорной переменной. Если строка считалась, то значение %FOUND истинно. После открытия курсора, но перед первым считыванием значением %FOUND является NULL. Атрибут %FOUND имеет смысл, противоположный смыслу атрибута %NOTFOUND (см. главу 6).

**Functions (функции)** Функции — это именованные блоки PL/SQL, которые возвращают значения и могут вызываться с различными аргументами. Функции можно указывать в выражениях. Их использование разрешено во всех процедурных операторах, а некоторые функции можно применять и в SQL-операторах (в PL/SQL 2.1 и выше). Функции сохраняются в базе данных с помощью команды CREATE [OR REPLACE] FUNCTION, а также могут располагаться в разделе объявлений другого блока (см. главу 7).

**GOTO — оператор** Оператор GOTO передает управление программой оператору, обозначенному меткой:

```
GOTO метка;
```

причем *метка* должна быть ограничена двойными угловыми скобками << и >>. Запрещено передавать управление в обработчики исключительных ситуаций, операторы IF, циклы и подблоки, а также из них (см. главу 2).

**Identifiers (идентификаторы)** Имя любого объекта PL/SQL является идентификатором. Идентификаторы не учитывают регистр символов и состоят из не более чем 30 символов применяемого в PL/SQL набора символов. Идентификаторы должны начинаться с букв (см. главу 2).

**IF — оператор** Оператор IF используется для условного выполнения последовательности операторов:

```
IF условие1 THEN
   последовательность_операторов1;
[ELSIF условие2 THEN
   последовательность_операторов2;]
...
[ELSE
   последовательность_операторов3;]
END IF;
```

*Последовательность операторов* выполняется в случае истинности первого же условия. Выполняться может лишь одна *последовательность операторов* (см. главу 2).

**INSERT — оператор** Оператор INSERT применяется для добавления строк в таблицу:

```
INSERT INTO таблица [(список_столбцов)]
   VALUES (список_выражений);
```

где *таблица* — ссылка на таблицу, в которую вводятся строки, а *список_выражений* — список разделенных запятыми выражений, являющихся полями новой строки. Если указан *список_столбцов*, то он определяет столбцы, в которые будут введены значения. В те столбцы, которые здесь не указаны, будут записаны NULL-значения. Если *список_столбцов* не задан, то *список_выражений* должен соответствовать всем столбцам таблицы (см. главу 4).

**INVALID_CURSOR — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-1001: invalid cursor" (неверный курсор). Она устанавливается при попытке использования неверного курсора. Это может произойти в том случае, когда пользователь пытается считать данные из неоткрытого курсора или из курсора, созданного для обновления (FOR UPDATE), после завершения работы (см. главу 6).

**INVALID_NUMBER — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-1722: invalid number" (неверное число). Она устанавливается при неудачной попытке преобразования некоторого значения к типу NUMBER. Например, строку символов 'Nineteen Hundred and Ninety Five' нельзя преобразовать к типу NUMBER, так как в ней не содержатся только цифры, десятичная точка или необязательный знак (см. главы 2 и 4).

**I/O (ввод/вывод)** Непосредственно в PL/SQL не обеспечивается поддержка ввода/вывода. Это функциональное средство реализуется при помощи модуля DBMS_OUTPUT, используемого в SQL*Plus, Server Manager или SQL*DBA. В SQL*Plus возможность ввода данных обеспечивается посредством переменных подстановки. В PL/SQL 2.3 предлагается новое средство — модуль UTL_FILE, реализующий файловый ввод/вывод (о DBMS_OUTPUT см. главу 14, а о UTL_FILE — главу 18).

**%ISOPEN — атрибут** Курсорный атрибут %ISOPEN используется для определения, открыт ли курсор (курсорная переменная (PL/SQL 2.2 и выше)):

*курсор*%ISOPEN | *курсорная переменная*%ISOPEN

где *курсор* — это имя явного курсора, а *курсорная переменная* — имя курсорной переменной. Если курсор (курсорная переменная) открыт посредством оператора OPEN или OPEN..FOR и еще не закрыт, то %ISOPEN возвращает TRUE. SQL%ISOPEN всегда возвращает FALSE, так как неявный курсор после выполнения SQL-оператора всегда закрыт (см. главу 6).

**Literals (литералы)** Литералы делятся на числовые и символьные. Числовой литерал состоит из цифр (от 0 до 9), с необязательным знаком и/или десятичной точкой. Числовые литералы можно использовать при экспоненциальном представлении данных. Типом данных для всех числовых литералов является тип NUMBER. Символьный литерал — это любая последовательность символов, заключенных в одиночные кавычки. Строки символов, ограниченные двойными кавычками, трактуются не как литералы, а как идентификаторы с учетом регистра символов. Тип данных для символьных литералов — CHAR, но не VARCHAR2 (см. главу 2).

PL/SQL 8.0 ... и ВЫШЕ

**LOBs (большие объекты)** Объекты LOB, применяемые в Oracle8, намного удобнее, чем данные типов LONG и LONG RAW. Объектами LOB можно управлять либо с помощью модуля DBMS_LOB, либо через интерфейс OCI. В объекте LOB можно хранить до 4 Гбайт символьных или двоичных данных (см. главу 21).

**LOCK TABLE — оператор** LOCK TABLE — это оператор DML, с помощью которого можно блокировать целую таблицу. Для блокирования отдельных строк таблицы применяется оператор SELECT..FOR UPDATE. Синтаксис LOCK TABLE таков:

LOCK TABLE *таблица* IN *режим_блокирования* [NOWAIT];

где *таблица* — это ссылка на нужную таблицу, а *режим_блокирования* определяет используемый режим. Существуют следующие режимы блокирования: ROW EXCLUSIVE, ROW SHARE, SHARE, SHARE UPDATE, SHARE ROW EXCLUSIVE и EXCLUSIVE. Если указано ключевое слово NOWAIT, оператор немедленно возвращается. При этом устанавливается либо блокировка, либо исключительная ситуация TIMEOUT_ON_RESOURCE (если блокировка не может быть установлена). Если NOWAIT не указано, LOCK TABLE будет ожидать установления блокировки (см. Oracle Server SQL Reference).

**LOGIN_DENIED — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-1017: invalid username/password; logon denied" (неверные имя/пароль пользователя; в регистрации отказано). Она устанавливается при указании неправильной комбинации имя/пароль пользователя.

**Loops (циклы)** В PL/SQL существуют четыре различных вида циклических структур. Простые циклы ограничиваются ключевыми словами LOOP и END LOOP. Циклы WHILE начинаются с ключевых слов WHILE..LOOP и заканчиваются словами END LOOP. Для числовых и курсорных циклов FOR используется синтаксис FOR индекс IN..LOOP с ключевыми словами END LOOP. Все циклы могут начинаться с метки, которую затем можно использовать в операторе EXIT внутри цикла (о простых циклах, циклах WHILE и числовых циклах FOR см. главу 2; о курсорных циклах FOR см. главу 6).

PL/SQL 8.0 ... и ВЫШЕ

**MAP & ORDERS — методы** Методы MAP и ORDERS — это специальные методы для объектных типов, используемые с целью определения порядка сортировки экземпляров объектного типа. У каждого объектного типа может быть либо метод MAP, либо метод ORDERS, но не оба сразу (см. главу 12).

PL/SQL 8.0 ... и выше

**Nested Tables (вложенные таблицы)** Вложенные таблицы расширяют функции таблиц PL/SQL, применяемых в версии 2. В отличие от таблиц PL/SQL версии 2, вложенные таблицы можно хранить в таблицах базы данных. С помощью SQL-операторов можно работать с отдельными строками вложенных таблиц или с целыми таблицами. Кроме того, для вложенных таблиц применимы такие методы сборных конструкций, как FIRST, LAST, NEXT и PRIOR (см. главу 12).

**NO_DATA_FOUND — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-1403: no data found" (данные не найдены). Она устанавливается, когда оператор SELECT..INTO не отбирает ни одной строки, а также при ссылке на такую строку таблицы PL/SQL, которой еще не присвоено какое-либо значение (о SELECT..INTO см. главу 4; о таблицах PL/SQL см. главу 3).

**%NOTFOUND — атрибут** Этот курсорный атрибут используется для определения, считалась ли из курсора или курсорной переменной (в PL/SQL 2.2 и выше) строка во время последней операции FETCH:

*курсор*%NOTFOUND | *курсорная переменная*%NOTFOUND

где *курсор* — это идентификатор курсора, а *курсорная переменная* — идентификатор курсорной переменной (в PL/SQL 2.2 и выше). Если в результате достижения конца активного набора строка не считалась, то %NOTFOUND возвращает TRUE. Атрибут %NOTFOUND имеет смысл, противоположный смыслу атрибута %FOUND (см. главу 6).

**NOT_LOGGED_ON — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-1012: not coonected to Oracle" (нет соединения с Oracle). Она устанавливается при выдаче SQL-оператора до соединения с Oracle.

**NULL — оператор** Оператор NULL, задаваемый следующим образом:

NULL;

не выполняет никаких действий. Он удобен для указания того, что в некоторой точке, где требуется именно оператор PL/SQL, не нужно выполнять какие-либо операции (см. главу 2).

**NULL — значение** Все выражения PL/SQL могут принимать NULL-значения, если только они не ограничены соответствующим образом при объявлении переменной. NULL трактуется как пропущенное или неизвестное значение. Многие выражения принимают NULL-значение, если один из операндов становится NULL-значением. В их число входят логические выражения — логические операции в PL/SQL реализуют трех-, а не двузначную логику (см. главу 2).

**OPEN — оператор** Оператор OPEN используется для открытия курсора (или курсорной переменной (в PL/SQL 2.2 и выше)). При открытии курсора оцениваются все переменные привязки в условии WHERE и определяется активный набор. Переменные привязки не проверяются, а активный набор изменяется, если только закрыть и повторно открыть курсор:

OPEN *имя_курсора*; | OPEN *курсорная_переменная* FOR *оператор_выбора*;

где *имя_курсора* обозначает ранее открытый статический курсор, а *курсорная_переменная* указывает предварительно описанную курсорную переменную. Для курсорных переменных указывается *оператор_выбора* (см. главу 6).

**Packages (модули)** Модули определяются как два различных объекта словаря данных: заголовок (или спецификация) и тело модуля. Заголовок создается с помощью команды CREATE OR REPLACE PACKAGE, а тело — с помощью команды CREATE OR REPLACE PACKAGE BODY. Модули должны храниться в базе данных; их нельзя располагать в разделе объявлений, как процедуры или функции. В самих модулях могут содержаться процедуры, функции, переменные, типы, курсоры и исключительные ситуации. Элементы, объявленные в заголовке модуля, будут видимы вне пределов модуля, в то время как элементы, описанные в теле модуля, являются частными, или локальными по отношению к этому модулю. Модули разрывают существующие зависимости, поскольку тело модуля можно перекомпилировать без воздействия на его спецификацию (см. главу 8).

**Procedural Statements (процедурные операторы)** Процедурные операторы, в противоположность SQL-операторам, управляют обработкой блоков PL/SQL. Эти операторы выполняются системой поддержки PL/SQL и не направляются программе-обработчику в базу данных. В состав процедурных операторов входят вызовы процедур, операторы присваивания, условные операторы, например IF..THEN, и циклы (см. главу 2).

**Procedures (процедуры)** Процедуры можно сохранять в базе данных с помощью оператора CREATE OR REPLACE PROCEDURE, а также создавать в разделе объявлений блока. Процедура — это

именованный блок, который можно вызывать с различными параметрами. Параметры могут принимать значения, вводимые вызывающей средой (IN), возвращать значения в вызывающую среду (OUT) или совмещать две эти функции (IN OUT) (см. главу 7).

**PROGRAM_ERROR — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "PL/SQL: internal error mmm, arguments [mmm], [mmm], [mmm], [mmm], [mmm], [mmm]" (PL/SQL: внутренняя ошибка), где mmm представляет собой код и аргументы, соответствующие данной внутренней ошибке. Эта исключительная ситуация устанавливается при возникновении внутренней ошибки PL/SQL.

**RAISE — оператор** Оператор RAISE используется для установления исключительной ситуации, сигнализирующей о возникновении ошибки. Исключительная ситуация может быть стандартной или определяемой пользователем:

RAISE [*имя_исключительной_ситуации*];

*имя_исключительной_ситуации* указывает устанавливаемую исключительную ситуацию. Единственное место, где можно задать оператор RAISE без именованной исключительной ситуации, — это обработчик исключительной ситуации. В таком случае текущая исключительная ситуация установится вновь (см. главу 10).

**Records (записи)** Записи PL/SQL используются для группирования логически связанной информации различного типа. Как и в таблицах PL/SQL, сначала нужно описать тип записи и только потом объявить переменную этого типа. Ниже приведен пример объявления записи для хранения некоторой информации о студентах.

```
DECLARE
TYPE t_StudentType IS RECORD (
  FirstName students.first_name%TYPE,
  LastName students.last_name%TYPE,
  ID students.ID%TYPE);
v_StudentInfo t_StudentType;
```

Атрибут %ROWTYPE также возвращает запись PL/SQL (см. главу 3).

PL/SQL 2.1 ... и ВЫШЕ

**RESTRICT_REFERENCES — прагма** Прагма RESTRICT_REFERENCES используется для задания уровней строгости функций и методов, определяемых пользователями. Чтобы использовать функцию в SQL-операторе, необходимо дать системе PL/SQL гарантию того, что функция не изменит состояние базы данных или модуля:

PRAGMA RESTRICT_REFERENCES(*имя_функции*,
[RNDS], [,WNDS], [,RNPS], [,WNPS]);

где *имя_функции* — это функция, уровень строгости которой задается. Уровни строгости, приведенные в таблице С.3, можно указывать в любом порядке.

**Таблица С.3.**

| Уровень строгости | Значение |
|---|---|
| RNDS (Read no database state) | Не читать состояние базы данных |
| WNDS (Write no database state) | Не записывать состояние базы данных |
| RNPS (Read no package state) | Не читать состояние модуля |
| WNPS (Write no package state) | Не записывать состояние модуля |

**RETURN — оператор** Оператор RETURN можно применять в двух случаях: для возврата из функции и для возврата из процедуры:

RETURN [*возвращаемое_значение*];

RETURN передает управление программой из функции или процедуры в вызывающую среду. Для функции необходимо указывать *возвращаемое_значение;* для процедуры это необязательно. Функция должна передавать в вызывающую среду некоторое значение (см. главу 7).

**ROLLBACK — оператор** Оператор ROLLBACK используется для окончания транзакции и для отката всей работы, выполненной данной транзакцией, то есть для возврата к тому состоянию, которое было перед началом транзакции:

ROLLBACK [WORK] [TO SAVEPOINT *точка_сохранения*];

Ключевое слово WORK необязательно. Как и COMMIT, ROLLBACK снимает все блокировки, установленные транзакцией. Если указана *точка сохранения*, то откатывается только та работа, которая была выполнена после этой точки (см. главу 4).

**%ROWCOUNT — атрибут** Этот курсорный атрибут возвращает число строк, считанных ранее для явного курсора, или число строк, на которые оказал воздействие последний оператор неявного SQL-курсора. С каждой явной операцией считывания значение %ROWCOUNT возрастает на единицу (см. главу 6).

**%ROWTYPE — атрибут** Этот атрибут применяется для таблиц базы данных. Он возвращает тип записи PL/SQL, состоящей из всех столбцов таблицы, в том порядке, в котором они были указаны при создании таблицы (см. главу 2). Например, в следующем примере объявляется запись, в которой может храниться строка таблицы **classes**:

```
DECLARE
  v_ClassInfo classes%ROWTYPE;
```

PL/SQL 2.2 ... и ВЫШЕ

**ROWTYPE_MISMATCH — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-6504: PL/SQL: return types of result set variables or query do not match" (PL/SQL: не установлено соответствие между возвращаемыми типами переменных результирующего набора и запросом). Эта исключительная ситуация устанавливается при открытии курсорной переменной с помощью оператора OPEN..FOR для запроса, тип которого отличен от того, для которого переменная была создана (см. главу 6).

**SAVEPOINT — оператор** Оператор SAVEPOINT используется для отметки определенного места транзакции:

SAVEPOINT *имя_точки_сохранения*;

где *имя_точки_сохранения* – это имя определяемой точки сохранения. После создания точки сохранения можно откатывать транзакцию до этой точки с помощью команды ROLLBACK TO SAVEPOINT (см. главу 4).

**SELECT..INTO — оператор** Оператор SELECT..INTO используется для считывания в базе данных одной строки:

SELECT *список_выбора*
  INTO *список_переменных* | *запись*
  FROM *таблица*
  [WHERE *условие*
  [GROUP BY *группирование*]
  [ORDER BY *упорядочивание*]
  [HAVING *имея*];

*Список переменных* должен соответствовать пунктам списка выбора или полям *записи*. Строка считывается из таблицы, указанной как *таблица*. Остальные конструкции определяют, какая именно строка должна быть выбрана. Если запрос возвращает несколько строк, нужно использовать явный курсор. Если запрос не возвращает ни одной строки, устанавливается исключительная ситуация NO_DATA_FOUND (см. главу 4).

**SET TRANSACTION — оператор** Оператор SET TRANSACTION используется для задания характеристик транзакции. Этот оператор необходимо вызывать как первый оператор транзакции:

SET TRANSACTION
  USE ROLLBACK SEGMENT *сегмент* | READ ONLY;

SET TRANSACTION можно применять как для назначения транзакции определенного сегмента отката, так и для разрешения операций "только для чтения" (запросов и операторов LOCK TABLE). Все запросы в транзакции с типом "только для чтения" будут согласованы по чтению данных с начала транзакции, а не для каждого оператора (см. руководство Oracle Server SQL Reference).

**SQL Cursor (SQL-курсор)** Каждый SQL-оператор обрабатывается в курсоре. Для многострочных запросов используются явные курсоры, в которых для обработки данных задаются команды OPEN, FETCH и CLOSE. Для других SQL-операторов применяются неявные курсоры, называемые SQL-курсорами. Существуют четыре курсорных атрибута: SQL%FOUND, SQL%NOTFOUND, SQL%ROWCOUNT и SQL%ISOPEN. SQL%ISOPEN всегда возвращает FALSE, так как до проверки SQL%ISOPEN неявный курсор открывается, обрабатывает оператор и закрывается (см. главу 6).

**SQL Statements (SQL-операторы)** SQL-операторы, в отличие от процедурных операторов, применяются для выдачи команд базе данных. Непосредственно в PL/SQL разрешены только операторы DML и операторы управления транзакциями. Для выдачи в PL/SQL команд DDL можно использовать модуль DBMS_SQL, применяемый в PL/SQL 2.1 и выше (см. главу 4).

**SQLCODE — функция** Эта функция используется для получения кода текущей ошибки. Она обычно применяется в обработчике WHEN OTHERS для определения ошибки Oracle, вызвавшей установление исключительной ситуации. SQLCODE возвращает значение типа INTEGER (см. главу 10).

**SQLERRM — функция** Эта функция возвращает текст сообщения об ошибке, соответствующий коду ошибки Oracle. Если код ошибки не указан, SQLERRM возвращает текст сообщения для текущей ошибки. Функция SQLERRM обычно применяется вместе с функцией SQLCODE в обработчике WHEN OTHERS для определения текста ошибки, вызвавшей установление исключительной ситуации. Максимальный размер текста ошибки составляет 512 символов (см. главу 10).

**STANDARD — модуль** В модуле PL/SQL, называемом STANDARD, определены все стандартные исключительные ситуации (например NO_DATA_FOUND или INVALID_CURSOR), типы (например NUMBER или DATE) и функции (например TO_CHAR или ADD_MONTHS). Владельцем этого модуля является пользователь SYS, а создается STANDARD при создании словаря данных посредством сценария catproc.sql, в котором роли PUBLIC предоставляется полномочие EXECUTE на STANDARD. В отличие от других модулей PL/SQL, к объектам модуля STANDARD можно обращаться, не указывая перед именем объекта *имя модуля*.

**Statements (операторы)** Существуют два вида операторов: SQL-операторы и процедурные. Процедурные операторы — это, например, операторы присваивания, вызовы процедур, циклы и операторы IF. SQL-операторы подразделяются на операторы DML, операторы DDL, операторы управления транзакциями, операторы управления сеансами и операторы управления системой (см. главы 2 и 4).

**STORAGE_ERROR — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-6500: PL/SQL: storage error" (ошибка хранения данных). Она устанавливается в том случае, когда PL/SQL не может выделить память, достаточную для продолжения своей работы. Это внутренняя ошибка, которая не происходит при обычных условиях.

PL/SQL 8.0 ... и ВЫШЕ

**SUBSCRIPT_BEYOND_COUNT — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-6533: subscript beyond count" (слишком большой индекс). Она устанавливается при ссылке на индексное значение сборной конструкции, превышающее число элементов изменяемого массива или слишком большое для вложенной таблицы (см. главу 12).

PL/SQL 8.0 ... и ВЫШЕ

**SUBSCRIPT_OUTSIDE_LIMIT — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-6532: subscript outside limit" (индекс вне установленных пределов). Она устанавливается при ссылке на индексное значение, превышающее границы изменяемого массива, либо на неположительное индексное значение изменяемого массива или вложенной таблицы (см. главу 12).

**Tables (таблицы)** Таблицы PL/SQL синтаксически схожи с массивами, применяемыми в других языках третьего поколения. Для того чтобы объявить таблицу PL/SQL, сначала необходимо описать новый табличный тип, а затем переменную этого типа. Ниже приведен пример создания таблицы, состоящей из дат:

```
DECLARE
   TYPE t_DateTable IS TABLE OF DATE
      INDEX BY BINARY_INTEGER;
   v_Dates t_DateTable;
```

Таблица синтаксически похожа на массив, но реализация ее отличается от реализации массива (см. главу 3).

**TIMEOUT_ON_RESOURCE — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-51: timeout occured while waiting for resource" (тайм-аут при ожидании ресурса). Ее можно установить, указав в операторе SELECT..FOR UPDATE конструкции NOWAIT, когда другой сеанс уже установил блокировку запрошенных строк.

**TOO_MANY_ROWS — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-1422: exact fetch returns more than requested numbers of rows" (операция явного считывания возвращает больше строк чем запрошено). Она устанавливается, если оператор SELECT..INTO возвращает несколько строк. В такой ситуации для считывания всего активного набора следует применять курсор (см. главу 6).

**TRANSACTION_BACKED_OUT — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-61: another instance has a different DML_LOCKS setting" (еще один экземпляр установил DML_LOCKS). Эта исключительная ситуация предопределена только в PL/SQL версий 2.0 и 2.1, но не в 2.2 и более поздних. Она устанавливается, если в результате тупиковой ситуации (взаимоблокировки) должен быть выполнен откат транзакции.

**Triggers (триггеры)** Триггер, как и процедура, является именованным, хранимым в базе данных блоком, и его можно вызывать из других блоков. Однако триггер не вызывается явным образом. Он активизируется неявно, всякий раз при наступлении соответствующего события. Таким событием является операция DML, выполняемая над базой данных. Существуют 12 видов триггеров, отличающихся типом оператора (INSERT, UPDATE, DELETE), типом триггера (ROW, STATEMENT) и временем активизации (BEFORE, AFTER). В Oracle8 применяется еще один вид триггеров – триггеры INSTEAD OF (см. главу 9).

**%TYPE — атрибут** Атрибут %TYPE можно указывать вместе с переменной или со столбцом таблицы. Он возвращает тип заданного объекта, что позволяет создавать более гибкие программы. В качестве примера ниже приведен блок, в котором переменная **v_FirstName** объявляется с типом VARCHAR2(20), а переменная **v_CurrentCredits** – с типом NUMBER(3), так как именно такие типы имеют столбцы **student.first_name** и **student.current_credits**.

```
DECLARE
   V_FirstName          student.first_name%TYPE;
   V_CurrentCredits     student.current_credits%TYPE;
```

При этом учитываются только размер или точность/масштаб столбца. Даже если столбец ограничен как NOT NULL, в переменной могут содержаться NULL-значения (если, конечно, она тоже не ограничена) (см. главу 2).

**UPDATE — оператор** Оператор UPDATE используется для модификации существующих строк таблицы базы данных:

UPDATE *таблица* SET *столбец1 = значение1, столбец2 = значение2, ...*
[WHERE *условие*]

где *таблица* – это ссылка на изменяемую таблицу, а *условие* указывает, какие строки следует изменить. В столбцах, обозначенных как *столбец1, столбец2* и т.д., устанавливаются соответствующие значения. Если оператор UPDATE оказывает воздействие на одну или несколько строк, то значение SQL%FOUND – TRUE, значение SQL%NOTFOUND – FALSE, а в SQL%ROWCOUNT содержится число строк, модифицированных после выполнения оператора. Если соответствия строк в операторе не установлено, то в SQL%FOUND находится FALSE, в SQL%NOTFOUND – TRUE, а в SQL%ROWCOUNT – ноль (см. главу 4).

**VALUE_ERROR — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-6502: numeric or value error" (ошибка числа или значения). Она устанавливается при неудачной попытке преобразования символьного значения к типу NUMBER. VALUE_ERROR обычно устанавливается для процедурных операторов; для SQL-операторов вместо нее устанавливается исключительная ситуация INVALID_NUMBER (см. главы 2 и 4).

**Variables & Constants (переменные и константы)** Переменные и константы описываются в разделе объявлений блока PL/SQL:

*имя_переменной тип* [CONSTANT] [NOT NULL] [:= *начальное_значение*];

где *имя_переменной* – это имя новой переменной или константы, а *тип* – ее тип, который может быть как стандартным типом, например DATE, так и определяемым пользователем. Если указано ключевое слово CONSTANT, то значение переменной изменяться не будет – это константа. Переменной или константе будет присваиваться *начальное_значение*; если оно не задано, то присваивается NULL (см. главу 2).

PL/SQL 8.0 ... и ВЫШЕ

**VARRAYs (изменяемые массивы)** Изменяемый массив – это упорядоченная совокупность объектов, для которой установлен максимальный размер. Работать с изменяемыми массивами можно при помощи таких методов сборных конструкций, как DELETE, COUNT, NEXT и PRIOR (см. главу 12). В приведенном ниже примере создается тип изменяемого массива, в котором будут храниться даты:

```
DECLARE
   TYPE t_Dates IS VARRAY(25) OF DATE;
```

**ZERO_DIVIDE — исключительная ситуация** Эта стандартная исключительная ситуация соответствует ошибке Oracle "ORA-1476: divisor is equal to zero" (деление не ноль). Она устанавливается при попытке деления на ноль.

# Приложение D

## *Словарь данных*

В этом приложении описаны представления словаря данных, применяющиеся в процессе программирования на PL/SQL. Здесь приведены не все представления, а только используемые чаще других. Кроме того, дается краткое описание словаря данных и принципов его функционирования.

# Понятие словаря данных

Словарь данных (data dictionary) — это область, где Oracle хранит информацию о структуре базы данных. Сами данные находятся в других местах — в словаре данных описано, как организована фактическая информация. Словарь состоит из таблиц и представлений, к которым можно обращаться с запросами так же, как и любым другим таблицам и представлениям базы данных. Владельцем этих представлений является пользователь Oracle с именем SYS.

При создании и инсталляции базы данных, как правило, формируется словарь данных, без которого программы PL/SQL работать не смогут. В большинстве систем представления словаря данных создаются с помощью сценария **catproc.sql**, запускающегося на выполнение либо пользователем SYS, либо из внутреннего соединения SQL*DBA или Server Manager.

В дополнение к собственно словарю данных **catproc.sql** создает стандартные модули PL/SQL и DBMS, хранящиеся в словаре. Более подробно о представлениях (в том числе и о тех представлениях, которые не рассмотрены здесь, а также о представлениях функционирования v$) рассказано в руководстве Oracle7 Sever Reference. Информация о встроенных модулях дается в приложении B.

## Соглашения по именованию

Многие представления имеют по три различных экземпляра, обозначаемых как **user_***, **all_*** и **dba_***. Например, в трех экземплярах представлены сведения об исходном программном тексте хранимых объектов: в виде представлений **user_source**, **all_source** и **dba_source.** Вообще говоря, в представлениях типа **user_*** содержится информация об объектах, владельцем которых является текущий пользователь, в представлениях типа **all_*** — информация об объектах, доступных текущему пользователю (не всегда ему принадлежащих), а в представлениях типа **dba_*** — информация обо всех объектах базы данных.

SQL и PL/SQL не учитывают регистр символов, поскольку перед сохранением информации об объектах она преобразуется в прописные символы. Поэтому именно их следует использовать при обращении с запросом к словарю данных. Например, в представлении **user_objects** имеется столбец **object_name**, в котором находятся имена объектов. Эти имена хранятся в прописных символах. К представлению **user_objects** нужно обращаться так:

```
SQL> SELECT object_type, status
  2      FROM user_objects
  3      WHERE object_name = UPPER('ClassPackage');
OBJECT_TYPE   STATUS
-----------   ----------
PACKAGE       VALID
PACKAGE BODY  VALID
```

Обратите внимание на использование функции UPPER: благодаря ей запрос возвращает нужные строки (модуль **ClassPackage** описан в главе 8).

## Полномочия

Владельцем представлений словаря данных является пользователь SYS. По умолчанию все представления могут видеть только SYS и пользователи с системной привилегией DBA. Пользователи без привилегии DBA могут просматривать представления типа **user_*** и **all_***, а также некоторые другие. Просматривать представления типа **dba_*** такие пользователи смогут только в случае предоставления им привилегии SELECT на конкретное представление.

Представления словаря данных не следует изменять даже пользователю SYS. Они обновляются автоматически базой данных при изменении соответствующей информации. В Oracle предлагаются файлы-сценарии для изменения таблиц словаря данных при модернизации (или наоборот, при переходе к более ранней версии) базы данных. Например, файл **cat7302.sql** применяется при переходе от базы данных версии 7.3.1 к базе данных версии 7.3.2. Эти сценарии находятся в том же каталоге, что и **catproc.sql.**

# Представления словаря all_*/user_*/ dba_*

В этом разделе описаны представления словаря данных, имеющие все три разновидности: **user_***, **dba_*** и **all_***. Поскольку столбцы представлений одного класса схожи, они описываются в одной таблице. Для справки в таблице D.1 перечислены все классы представлений.

## Таблица D.1. Представления словаря данных, описанные в этом приложении

| Класс | Представления |
|---|---|
| Dependencies (зависимости) | all_dependencies, dba_dependencies, user_dependencies |
| Collections (сборные конструкции)* | all_coll_types, dba_coll_types, user_coll_types |
| Compile Errors (ошибки компиляции) | all_errors, dba_errors, user_errors |
| Directories (каталоги)* | all_directories, dba_directories, user_directories |
| Jobs (задания) | all_jobs, dba_jobs, user_jobs |
| Libraries (библиотеки)* | all_libraries, dba_libraries, user_libraries |
| LOBs (большие объекты)* | all_lobs, dba_lobs, user_lobs |
| Object Methods (объектные методы)* | all_type_methods, dba_type_methods, user_type_methods |
| Object Method Parameters (параметры объектных методов)* | all_method_params, dba_method_params, user_method_params |
| Object Method Results (результаты объектных методов) — возвращаемые значения* | all_method_results, dba_method_results, user_method_results |
| Object References (ссылки на объекты)* | all_refs, dba_refs, user_refs |
| Object Types (объектные типы)* | all_types, dba_types, user_types |
| Object Type Attributes (атрибуты объектных типов)* | all_type_attrs, dba_type_attrs, user_type_attrs |
| Schema Objects (объекты схем) | all_objects, dba_objects, user_objects |
| Source Code (исходный программный текст) | all_source, dba_source, user_source |
| Tables (таблицы) | all_tables, dba_tables, user_tables, all_catalog, dba_catalog, user_catalog |
| Table Columns (столбцы таблиц) | all_tab_columns, dba_tab_columns, user_tab_columns |
| Triggers (триггеры) | all_triggers, dba_triggers, user_triggers |
| Trigger Columns (столбцы триггеров) | all_trigger_cols, dba_trigger_cols, user_trigger_cols |

*Доступны в Oracle8 и выше.

## Dependencies (зависимости)

Представления **all_dependencies, dba_dependencies** и **user_dependencies** документируют отношения зависимости, существующие между хранимыми объектами.

### Таблица D.2.

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема-владелец объекта (только в **all_dependencies** и **dba_dependencies**) |
| NAME | NOT NULL | VARCHAR2(30) | Имя объекта (в прописных символах) |

**Таблица D.2. (продолжение)**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| TYPE | — | VARCHAR2(12) | Тип объекта: PROCEDURE, FUNCTION, PACKAGE или PACKAGE BODY |
| REFERENCED_OWNER | — | VARCHAR2(30) | Схема-владелец объекта, на который производится ссылка |
| REFERENCED_NAME | — | VARCHAR2(30) | Имя объекта, на который производится ссылка |
| REFERENCED_TYPE | — | VARCHAR2(12) | Тип объекта, на который производится ссылка: PROCEDURE, FUNCTION, PACKAGE или PACKAGE BODY |
| REFERENCED_LINK_NAME | — | VARCHAR2(128) | Имя связи баз данных с объектом, на который производится ссылка (если этот объект расположен в удаленной базе данных) |

## Collections (сборные конструкции)

PL/SQL 8.0 ... и ВЫШЕ

В представлениях **all_coll_types, dba_coll_types** и **user_coll_types** содержится информация о сборных конструкциях, доступных пользователю. В словаре данных хранятся только сборные конструкции, созданные при помощи оператора CREATE TYPE (в отличие от локальных для блока PL/SQL сборных конструкций).

**Таблица D.3.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема-владелец сборной конструкции (только в **all_coll_types** и **dba_coll_types**) |
| TYPE_NAME | NOT NULL | VARCHAR2(30) | Имя сборной конструкции |
| COLL_TYPE | NOT NULL | VARCHAR2(30) | Тип сборной конструкции: либо TABLE, либо VARYING ARRAY |
| UPPER_BOUND | — | NUMBER | Максимальное число элементов для типов изменяемых массивов |
| ELEM_TYPE_MODE | — | VARCHAR2(7) | Модификатор типа элемента (например REF) |
| ELEM_TYPE_OWNER | — | VARCHAR2(30) | Владелец типа элемента (если отличается от владельца собственно сборной конструкции) |
| ELEM_TYPE_NAME | — | VARCHAR2(30) | Имя типа элемента |
| LENGTH | — | NUMBER | Размер элемента, если его тип CHAR или VARCHAR2 |
| PRECISION | — | NUMBER | Точность элемента, если его тип NUMBER |
| SCALE | — | NUMBER | Масштаб элемента, если его тип NUMBER |
| CHARACTER_SET_NAME | — | VARCHAR2(44) | Имя набора символов: CHAR_CS или NCHAR_CS |

## Compile Errors (ошибки компиляции)

В представлениях **all_errors, dba_errors** и **user_errors** содержится текст ошибок компиляции для хранимых объектов и представлений. Если в представлениях **_errors** находится хотя бы одна запись, то объект недостоверен (что показано в представлениях **all_objects, dba_objects** и **user_objects**).

Типичный запрос, обращенный к user_errors, выглядит следующим образом:

```
SELECT line, position, text
  FROM user_errors
  WHERE name = object_name
  ORDER BY sequence;
```

**Таблица D.4.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема-владелец объекта (только в **all_errors** и **dba_errors**) |
| NAME | NOT NULL | VARCHAR2(30) | Имя объекта (в прописных символах) |
| TYPE | — | VARCHAR2(12) | Тип объекта: VIEW, PROCEDURE, FUNCTION, PACKAGE или PACKAGE BODY |
| SEQUENCE | NOT NULL | NUMBER | Номер последовательности; используется для упорядочивания ошибок |
| LINE | NOT NULL | NUMBER | Номер строки, в которой произошла ошибка |
| POSITION | NOT NULL | NUMBER | Смещение (относительно 0) в строке, в которой произошла ошибка |
| TEXT | NOT NULL | VARCHAR2(2000) | Текст ошибки, содержащий код ошибки и сообщение об ошибке |

## Directories (каталоги)

PL/SQL 8.0 ... и ВЫШЕ

В представлениях **all_directories, dba_directories** и **user_directories** содержится информация о каталогах, используемых для указания каталогов операционной системы (для объектов BFILE — см. главу 21).

**Таблица D.5.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец каталога (только в all_directories и dba_directories) |
| DIRECTORY_NAME | NOT NULL | VARCHAR2(30) | Имя каталога |
| DIRECTORY_PATH | NOT NULL | VARCHAR2(4000) | Маршрут к каталогу в файловой системе |

## Jobs (задания)

PL/SQL 8.0 ... и ВЫШЕ

В представлениях **all_jobs, dba_jobs** и **user_jobs** содержится информация о заданиях для базы данных.

**Таблица D.6.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| JOB | NOT NULL | NUMBER | Номер идентификатора задания. Идентификатор не изменяется, пока существует задание |
| LOG_USER | NOTNULL | VARCHAR2(30) | Пользователь, поставивший задание в очередь |
| PRIV_USER | NOT NULL | VARCHAR2(30) | Пользователь, чьих привилегий по умолчанию достаточно для работы с этим заданием |
| SCHEMA_USER | NOT NULL | VARCHAR2(30) | Схема по умолчанию для задания |

**Таблица D.6. (продолжение)**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| LAST_DATE | — | DATE | Дата последнего успешного выполнения задания |
| LAST_SEC | — | VARCHAR2(8) | То же, что и LAST_DATE, но в формате HH24:MI:SS |
| THIS_DATE | — | DATE | Дата начала выполнения задания; NULL, если задание в данный момент не выполняется |
| THIS_SEC | — | VARCHAR2(8) | То же, что и THIS_DATE, но в формате HH24:MI:SS |
| NEXT_DATE | NOT NULL | DATE | Дата следующего выполнения задания |
| NEXT_SEC | — | VARCHAR2(8) | То же, что и NEXT_DATE, но в формате HH24:MI:SS |
| TOTAL_TIME | — | NUMBER | Время в секундах, затраченное системой на это задание |
| BROKEN | — | VARCHAR2(1) | Y, если задание неработоспособно; N в противном случае |
| INTERVAL | NOT NULL | VARCHAR2(200) | Временная функция, используемая для вычисления следующего значения NEXT_DATE |
| FAILURES | — | NUMBER | Число неудачных попыток выполнения задания со времени его последнего успешного выполнения |
| WHAT | — | VARCHAR2(2000) | Тело анонимного блока PL/SQL, составляющего задание |
| CURRENT_SESSION_LABEL | — | RAW MLSLABEL | Метка сервера Trusted Oracle7 для текущего сеанса задания |
| CLEARANCE_HI | — | RAW MLSLABEL | Высший уровень гашения, применяемый для задания (только в Trusted Oracle7) |
| CLEARANCE_LO | — | RAW MLSLABEL | Низший уровень гашения, применяемый для задания (только в Trusted Oracle7) |
| NLS_ENV | — | VARCHAR2(2000) | Среда NLS для задания (как указано в ALTER SESSION) |
| MISC_ENV | — | RAW(32) | Другие параметры сеанса для задания |

## Libraries (библиотеки)

PL/SQL 8.0 ... и ВЫШЕ

В представлениях **all_libraries, dba_libraries** и **user_libraries** указывается местонахождение разделяемых библиотек, используемых в работе внешних процедур, в операционной системе (см. главу 20).

**Таблица D.7.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец библиотеки (только в **all_libraries** и **dba_libraries**) |
| LIBRARY_NAME | NOT NULL | VARCHAR2(30) | Имя библиотеки |
| FILE_SPEC | — | VARCHAR2(2000) | Маршрут к библиотеке в файловой системе |
| DYNAMIC | — | VARCHAR2(1) | Y, если библиотека динамическая; N в противном случае |
| STATUS | — | VARCHAR2(7) | Состояние библиотеки: VALID или INVALID |

## LOBs (большие объекты)

PL/SQL 8.0 ... и ВЫШЕ

В представлениях **all_lobs, dba_lobs** и **user_lobs** содержится информация об объектах LOB, находящихся в таблицах, доступных текущему пользователю.

### Таблица D.8.

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец таблицы, содержащей объект LOB (только в **all_lobs** и **dba_lobs**) |
| TABLE_NAME | NOT NULL | VARCHAR2(30) | Имя таблицы, содержащей объект LOB |
| COLUMN_NAME | — | VARCHAR2(4000) | Имя столбца или атрибута объекта LOB |
| SEGMENT_NAME | NOT NULL | VARCHAR2(30) | Имя сегмента объекта LOB |
| INDEX_NAME | NOT NULL | VARCHAR2(30) | Имя индекса объекта LOB |
| CHUNK | — | NUMBER | Размер порции данных LOB как единицы выделения или управления (в байтах) |
| PCTVERSION | NOT NULL | NUMBER | Максимальный процент области хранения LOB, используемый для отслеживания версий |
| CACHE | — | VARCHAR2(3) | YES, если обращение к LOB производится через буферный кэш; NO в противном случае |
| LOGGING | — | VARCHAR2(3) | YES, если объект LOB зарегистрирован; NO в противном случае |
| IN_ROW | — | VARCHAR2(3) | YES, если часть LOB хранится в строке базы данных; NO в противном случае |

## Object Methods (объектные методы)

PL/SQL 8.0 ... и ВЫШЕ

В представлениях **all_type_methods, dba_type_methods** и **user_type_methods** содержится информация о методах объектных типов, доступных текущему пользователю.

### Таблица D.9.

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец объектного типа (только в **all_type_methods** и **dba_type_methods**) |
| TYPE_NAME | NOT NULL | VARCHAR2(30) | Имя объектного типа |
| METHOD_NAME | NOT NULL | VARCHAR2(30) | Имя метода |
| METHOD_NO | NOT NULL | NUMBER | Номер метода (используется для отличия переопределяемых методов) |
| METHOD_TYPE | — | VARCHAR2(6) | Тип метода: MAP, ORDER или PUBLIC |
| PARAMETERS | NOT NULL | NUMBER | Число параметров метода |
| RESULTS | NOT NULL | NUMBER | Число результатов, возвращаемых методом |

## Object Method Parameters (параметры объектных методов)

PL/SQL 8.0 ... и ВЫШЕ

Представления **all_method_params, dba_method_params** и **user_method_params** описывают параметры методов доступных объектных типов. Применяемые методы полностью описывает совокупность представлений ***_method_params, *_method_results** и ***_type_methods**.

**Таблица D.10.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец объектного типа (только в **all_method_params** и **dba_method_params**) |
| TYPE_NAME | NOT NULL | VARCHAR2(30) | Имя объектного типа |
| METHOD_NAME | NOT NULL | VARCHAR2(30) | Имя метода |
| METHOD_NO | NOT NULL | NUMBER | Номер метода (используется для отличия переопределяемых методов) |
| PARAM_NAME | NOT NULL | VARCHAR2(30) | Имя параметра |
| PARAM_NO | NOT NULL | NUMBER | Номер или позиция параметра |
| PARAM_MODE | — | VARCHAR2(6) | Вид параметра (IN, OUT или IN OUT) |
| PARAM_TYPE_MODE | — | VARCHAR2(7) | Модификатор типа параметра (например REF) |
| PARAM_TYPE_OWNER | — | VARCHAR2(30) | Владелец типа параметра (если тип определяется пользователем) |
| PARAM_TYPE_NAME | — | VARCHAR2(30) | Имя типа параметра |
| CHARACTER_SET_NAME | — | VARCHAR2(44) | Набор символов для параметра |

## Object Method Results (результаты объектных методов)

PL/SQL 8.0 ... и ВЫШЕ

В представлениях **all_method_results, dba_method_results** и **user_method_results**, как и в представлениях ***_method_params**, содержится информация о методах объектных типов, доступных пользователю. Однако в представлениях ***_method_results** находятся сведения о значениях, возвращаемых методами, являющимися функциями.

**Таблица D.11.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец объектного типа (только в **all_method_results** и **dba_method_results**) |
| TYPE_NAME | NOT NULL | VARCHAR2(30) | Имя объектного типа |
| METHOD_NAME | NOT NULL | VARCHAR2(30) | Имя метода |
| METHOD_NO | NOT NULL | NUMBER | Номер метода (используется для отличия переопределяемых методов) |
| RESULT_TYPE_MODE | — | VARCHAR2(7) | Модификатор типа возвращаемого значения (например REF) |
| RESULT_TYPE_OWNER | — | VARCHAR2(30) | Владелец типа возвращаемого значения (если тип определяется пользователем) |
| RESULT_TYPE_NAME | — | VARCHAR2(30) | Имя типа возвращаемого значения (если тип определяется пользователем) |
| CHARACTER_SET_NAME | — | VARCHAR2(44) | Набор символов, используемый для возвращаемого значения |

## Object References (ссылки на объекты)

PL/SQL 8.0 ... и ВЫШЕ

В представлениях **all_refs, dba_refs** и **user_refs** содержится информация о столбцах типа REF таблиц, доступных пользователю.

**Таблица D.12.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец таблицы (только в **all_refs** и **dba_refs**) |
| TABLE_NAME | NOT NULL | VARCHAR2(30) | Имя таблицы, содержащей столбец REF |
| COLUMN_NAME | — | VARCHAR2(4000) | Имя столбца типа REF |
| WITH_ROWID | — | VARCHAR2(3) | YES, если столбец REF хранится с ROWID; NO в противном случае |
| IS_SCOPED | — | VARCHAR2(3) | YES, если столбец REF находится в области своего действия; NO в противном случае |
| SCOPE_TABLE_OWNER | — | VARCHAR2(30) | Если столбец REF находится в области своего действия, то схема — владелец таблицы, находящейся в области действия столбца |
| SCOPE_TABLE_NAME | — | VARCHAR2(30) | Если столбец REF находится в области своего действия, то имя таблицы, находящейся в области действия столбца |

## Object Type Attributes (атрибуты объектных типов)

PL/SQL 8.0 ... и ВЫШЕ

В представлениях **all_type_attrs, dba_type_attrs** и **user_type_attrs** содержится информация об атрибутах объектных типов, доступных пользователю.

**Таблица D.13.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | — | VARCHAR2(30) | Схема — владелец объектного типа (только в **all_type_attrs** и **dba_type_attrs**) |
| TYPE_NAME | NOT NULL | VARCHAR2(30) | Имя объектного типа |
| ATTR_NAME | NOT NULL | VARCHAR2(30) | Имя атрибута |
| ATTR_TYPE_MODE | — | VARCHAR2(7) | Модификатор атрибута (например REF) |
| ATTR_TYPE_OWNER | — | VARCHAR2(30) | Владелец типа атрибута (если тип определяется пользователем) |
| LENGTH | — | NUMBER | Размер атрибута CHAR или VARCHAR2 |
| PRECISION | — | NUMBER | Точность атрибута NUMBER |
| SCALE | — | NUMBER | Масштаб атрибута NUMBER |
| CHARACTER_SET_NAME | — | VARCHAR2(44) | Набор символов, используемый для атрибута |

## Schema Objects (объекты схем)

В представлениях **all_objects, dba_objects** и **user_objects** содержится информация обо всех типах объектов схем, в том числе о таблицах, хранимых подпрограммах, представлениях, последовательностях и

индексах. Объектные типы описаны в представлениях ***_type_methods, *_methods_params, *_methods_results, *_type_attrs** и ***_types.**

**Таблица D.14.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец объекта (только в **all_objects** и **dba_objects**) |
| OBJECT_NAME | NOT NULL | VARCHAR2(30) | Имя объекта (в символах верхнего регистра) |
| OBJECT_ID | NOT NULL | NUMBER | Номер объекта. Каждому объекту базы данных присваивается уникальный идентификатор. |
| OBLECT_TYPE | — | VARCHAR2(12) | Тип объекта (TABLE, PACKAGE BODY, SEQUENCE, PROCEDURE и т.д.) |
| CREATED | NOT NULL | DATE | Временная метка создания объекта |
| LAST_DDL_TIME | NOT NULL | DATE | Временная метка последней операции DDL (например ALTER), выполненной над объектом. Операции GRANT и REVOKE также изменяют эту временную метку |
| TIMESTAMP | — | VARCHAR2(75) | Временная метка создания в формате YYYY-MM-DD:HH24:MI:SS |
| STATUS | — | VARCHAR2(7) | Состояние объекта: VALID, INVALID или N/A |

## Source Code (исходный программный текст)

В представлениях **all_source, dba_source** и **user_source** содержится исходный текст хранимых процедур, функций, модулей и тел модулей. Исходный программный текст триггеров находится в представлениях **all_triggers, dba_triggers** и **user_triggers.** Если хранимый объект кодирован, то в этих представлениях содержится кодированный, а не исходный текст.

Типичный запрос, обращенный к user_source, выглядит следующим образом:

```
SELECT text
   FROM user_source
   WHERE name = object_name
   ORDER BY line;
```

**Таблица D.15.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец объекта (только в **all_source** и **dba_source**) |
| NAME | NOT NULL | VARCHAR2(30) | Имя хранимого объекта |
| TYPE | — | VARCHAR2(12) | Тип объекта: PACKAGE, PACKAGE BODY, PROCEDURE или FUNCTION |
| LINE | NOT NULL | NUMBER | Номер строки исходного текста |
| TEXT | — | VARCHAR2(2000) | Исходный текст в данной строке |

## Tables (таблицы)

В представлениях **all_tables, dba_tables** и **user_tables** содержится информация о таблицах базы данных. Информация о столбцах таблиц находится в представлениях **all_tab_columns, dba_tab_columns** и **user_tab_columns.**

## Таблица D.16.

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец таблицы (только в **all_tables** и **dba_tables**) |
| TABLE_NAME | NOT NULL | VARCHAR2(30) | Имя таблицы (в прописных символах) |
| TABLESPACE_NAME | NOT NULL | VARCHAR2(30) | Имя табличной области, содержащей эту таблицу |
| CLUSTER_NAME | — | VARCHAR2(30) | Имя кластера, которому принадлежит таблица. Если таблица некластеризована, то NULL |
| PCT_FREE | NOT NULL | NUMBER | Минимальный процент свободного пространства блока, указанный при создании таблицы или во время последней модификации таблицы оператором ALTER |
| PCT_USED | NOT NULL | NUMBER | Минимальный процент используемого пространства блока, указанный при создании таблицы или во время последней модификации таблицы оператором ALTER |
| INI_TRANS | NOT NULL | NUMBER | Начальное число транзакций, указанное при создании таблицы или во время последней модификации таблицы оператором ALTER |
| MAX_TRANS | NOT NULL | NUMBER | Максимальное число транзакций, указанное при создании таблицы или во время последней модификации таблицы оператором ALTER |
| INITIAL_EXTENT | — | NUMBER | Размер начального экстента в байтах (если указан) |
| NEXT_EXTENT | — | NUMBER | Размер следующего экстента в байтах (если указан) |
| MIN_EXTENTS | — | NUMBER | Минимальное число экстентов, разрешенное для сегмента (если указано) |
| MAX_EXTENTS | — | NUMBER | Максимальное число экстентов (если указано) |
| PCT_INCREASE | — | NUMBER | Разрешенный процент увеличения размера экстентов (если указано) |
| FREELISTS | — | NUMBER | Число выделенных сегменту списков свободных блоков (если указано) |
| FREELIST_GROUPS | — | NUMBER | Число выделенных сегменту групп списков свободных блоков (если указано) |
| BACKED_UP | — | VARCHAR2(1) | Y, если со времени последнего изменения таблицы выполнялось ее резервное копирование; N в противном случае |
| NUM_ROWS | — | NUMBER | Число строк в таблице |
| BLOCKS | | NUMBER | Число блоков данных, выделенных таблице |
| EMPTY_BLOCKS | — | NUMBER | Число выделенных блоков данных, не содержащих информацию. Отношение (**EMPTY_BLOCKS/BLOCKS**) * 100 представляет собой процент использования блоков |
| AVG_SPACE | — | NUMBER | Средний объем свободного пространства в выделенном блоке (в байтах) |
| CHAIN_CNT | — | NUMBER | Число сцепленных строк в таблице, распространяющихся более чем на один блок |

**Таблица D.16. (продолжение)**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| AVG_ROW_LEN | — | NUMBER | Средняя длина строк (в байтах) |
| DEGREE | — | VARCHAR2(10) | Число потоков на экземпляр для просмотра таблицы (только для параллельного сервера) |
| INSTANCES | — | VARCHAR2(10) | Число экземпляров, в которых просматривается таблица (только для параллельного сервера) |
| CACHE | — | VARCHAR2(5) | YES, если таблица кэшируется в буферном кэше; NO в противном случае |
| TABLE_LOCK | — | VARCHAR2(8) | ENABLED, если разрешено блокирование таблицы; DISABLED в противном случае |

Представления **all_catalog, dba_catalog** и **user_catalog** являются подмножеством представлений **all_tables, dba_tables** и **user_tables**.

**Таблица D.17.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец объекта (только в **all_catalog** и **dba_catalog**) |
| TABLE_NAME | NOT NULL | VARCHAR2(30) | Имя объекта (в прописных символах) |
| TABLE_TYPE | — | VARCHAR2(11) | Тип объекта (TABLE, VIEW, SYNONYM, SEQUENCE) |

## Table Columns (столбцы таблиц)

В представлениях **all_tab_columns, dba_tab_columns** и **user_tab_columns** содержится информация о столбцах таблиц, представлений и кластеров базы данных.

**Таблица D.18.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец объекта (только в **all_tab_columns** и **dba_tab_columns**) |
| TABLE_NAME | NOT NULL | VARCHAR2(30) | Имя таблицы, представления или кластера |
| COLUMN_NAME | NOT NULL | VARCHAR2(30) | Имя столбца |
| DATA_TYPE | — | VARCHAR2(9) | Тип данных столбца (NUMBER, CHAR, DATE и т.д.) |
| DATA_LENGTH | NOT NULL | NUMBER | Максимальный размер столбца в байтах |
| DATA_PRECISION | — | NUMBER | Десятичная точность для столбцов NUMBER, двоичная точность для столбцов FLOAT. Для всех других типов данных или в том случае, когда точность не указана, NULL |
| DATA_SCALE | — | NUMBER | Масштаб столбцов NUMBER. Для всех других типов данных или в том случае, когда масштаб не указан, NULL |
| NULLABLE | — | VARCHAR2(1) | Y, если в столбце разрешены NULL-значения; N в противном случае |
| COLUMN_ID | NOT NULL | NUMBER | Уникальное значение, присвоенное столбцу. Идентификаторы назначаются всем столбцам |
| DEFAULT_LENGTH | — | NUMBER | Размер значения по умолчанию для столбца (если указан) |
| DATA_DEFAULT | — | LONG | Значение по умолчанию для столбца (если указано) |

### Таблица D.18. (продолжение)

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| NUM_DISTINCT | — | NUMBER | Число неповторяющихся значений в столбце |
| LOW_VALUE | — | RAW(32) | Второе по меньшинству значение в таблице (4 строки или более) или младшее значение в таблице (3 строки или менее). Хранится как внутреннее представление первых 32 байтов столбца |
| HIGH_VALUE | — | RAW(32) | Второе по старшинству значение в таблице (4 строки или более) или старшее значение в таблице (3 строки или менее). Хранится как внутреннее представление первых 32 байт столбца |
| DENSITY | — | NUMBER | Плотность столбца |
| NUM_NULLS | — | NUMBER | Число строк, содержащих NULL-значения |
| NUM_BUCKETS | — | NUMBER | Число областей, использованных при анализе (ANALYZE) таблицы |
| LAST_ANALYZED | — | DATE | Временная метка последнего анализа таблицы |
| SAMPLE_SIZE | — | NUMBER | Выборочный размер, использовавшийся во время последнего анализа |

## Triggers (триггеры)

В представлениях **all_triggers, dba_triggers** и **user_triggers** описаны триггеры базы данных, доступные пользователю. Столбцами этих представлений являются различные компоненты оператора CREATE TRIGGER.

### Таблица D.19.

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | OT NULL | VARCHAR2(30) | Схема — владелец триггера (только в **all_triggers** и **dba_triggers**) |
| TRIGGER_NAME | NOT NULL | VARCHAR2(30) | Имя триггера (в прописных символах) |
| TRIGGER_TYPE | — | VARCHAR2(16) | Тип триггера: BEFORE ROW, BEFORE STATEMENT, AFTER ROW, AFTER STATEMENT |
| TRIGGER_EVENT | — | VARCHAR2(26) | Оператор DML, активизирующий триггер: один или несколько операторов INSERT, UPDATE, DELETE |
| TABLE_OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец таблицы, для которой создан триггер |
| TABLE_NAME | NOT NULL | VARCHAR2(30) | Имя таблицы, для которой создан триггер |
| REFERENCING_NAMES | — | VARCHAR2(87) | Имена, используемые для ссылок **:old** и **:new** в строковых триггерах, если указаны |
| WHEN_CLAUSE | — | VARCHAR2(2000) | Условие WHEN для триггера (если указано) |
| STATUS | — | VARCHAR2(8) | ENABLED, если триггер разрешен; DISABLED в противном случае |
| DESCRIPTION | — | VARCHAR2(2000) | Строка символов, содержащая имя триггера, тип триггера, условие WHEN и ссылку (как указано в операторе CREATE TRIGGER). Удобно для воссоздания оператора в случае потери файла с исходным текстом |
| TRIGGER_BODY | — | LONG | Блок PL/SQL, составляющий тело триггера |

## Trigger Columns (столбцы триггеров)

В представлениях **all_trigger_cols, dba_trigger_cols** и **user_trigger_cols** показано использование столбцов триггеров базы данных. Эти представления дополняют представления **all_triggers, dba_triggers** и **user_triggers.**

### Таблица D.20.

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| TRIGGER_OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец триггера |
| TRIGGER_NAME | NOT NULL | VARCHAR2(30) | Имя триггера (в прописных символах) |
| TABLE_OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец таблицы, для которой создан триггер |
| TABLE_NAME | NOT NULL | VARCHAR2(30) | Имя таблицы, для которой создан триггер (в прописных символах) |
| COLUMN_NAME | NOT NULL | VARCHAR2(30) | Имя столбца, применяемого в триггере |
| COLUMN_LIST | — | VARCHAR2(3) | YES, если столбец указан в конструкции UPDATE; NO в противном случае |
| COLUMN_USAGE | — | VARCHAR2(17) | Показывает, каким образом триггер ссылается на столбец. Все возможные комбинации из NEW, OLD, IN, OUT и IN OUT |

## Views (представления)

В представлениях **all_views, dba_views** и **user_views** описываются все представления базы данных.

### Таблица D.21.

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| OWNER | NOT NULL | VARCHAR2(30) | Схема — владелец представления (только в **all_views** и **dba_views**) |
| VIEW_NAME | NOT NULL | VARCHAR2(30) | Имя представления (в прописных символах) |
| TEXT_LENGTH | — | NUMBER | Размер текста представления |
| TEXT | — | LONG | Текст представления — тело оператора CREATE VIEW |

# Другие представления словаря

Помимо представлений, описанных выше, есть еще два представления, полезных во время программирования на PL/SQL.

## dbms_alert_info

В представлении **dbms_alert_info** содержится информация о сеансах, зарегистрировавших свою заинтересованность в оповещениях.

### Таблица D.22.

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| NAME | NOT NULL | VARCHAR2(30) | Имя оповещения, на которое зарегистрирован сеанс |
| SID | NOT NULL | VARCHAR2(30) | Идентификатор сеанса |
| CHANGED | — | VARCHAR2(30) | Y, если сигнал об оповещении выдан; N в противном случае |
| MESSAGE | — | VARCHAR2(1800) | Сообщение, переданное при вызове SIGNAL |

# dict_columns

В представлении **dict_columns** описаны все столбцы словаря данных.

**Таблица D.23.**

| Столбец | NULL? | Тип | Описание |
|---|---|---|---|
| TABLE_NAME | — | VARCHAR2(30) | Имя представления словаря данных |
| COLUMN_NAME | — | VARCHAR2(30) | Столбец представления |
| COMMENTS | — | VARCHAR2(2000) | Описание столбца |

# *Содержание*

# Информация о прилагаемом компакт-диске

К этой книге прилагается компакт-диск, содержимое которого состоит из трех разделов:

1. Программный текст примеров, используемых в книге.
2. Демонстрационная версия первой редакции SQL-Station для Windows 95 и Windows NT производства корпорации Platinum Technologies обладает всеми свойствами обычной версии; срок ее использования истекает через 30 дней.
3. Демонстрационная версия программы Oracle WebServer 2.1 для систем Windows NT и Solaris обладает всеми свойствами обычной версии; срок ее использования истекает через 90 дней.

Компакт-диск организован следующим образом:

| | |
|---|---|
| readme.txt | Файл, в котором содержится информация, приведенная на этой странице |
| code/ | Каталог, в котором содержится программный текст всех примеров, используемых в книге. В этом каталоге находятся подкаталоги с примерами для каждой главы. Кроме того, в каталоге code расположен файл, называемый readme.txt, с полным описанием содержимого этого каталога. С помощью файла code/ch01/tables.sql создаются все таблицы, используемые в примерах. Весь программный текст распространяется в формате текстовых файлов ASCII, которые читаются стандартными текстовыми редакторами в системах Windows и Unix |
| station/ | Каталог, в котором содержится демонстрационная версия SQL-Station. Для ее инсталляции выполните программу setup.exe, находящуюся в каталоге station. Оперативная документация (в формате HTML) расположена в каталоге station/info. Более подробную информацию об SQL-Station, в том числе сведения о ценах и о порядке приобретения программы, можно найти на web-узле Platinum www.platinum.com. Описание SQL-Station дано в главах 13, 14, 19 и 22 этой книги |
| ows-nt/<br>ows-sol/ | Каталоги, содержащие демонстрационные версии Oracle WebServer 2.1. В каталоге ows-nt находится версия для NT, а в каталоге ows-sol — Solaris. Для инсталляции NT-версии выполните программу setup.exe, находящуюся в каталоге ows-nt. После инсталляции установите дополнение к версии 2.1.0.3.2, содержащееся в каталоге ows-nt/patch (см. файл readme.txt в каталоге ows-nt/patch). Для инсталляции Solaris-версии следуйте указаниям файла install.txt, находящегося в каталоге ows-sol. Более подробную информацию об Oracle WebServer, в том числе сведения о ценах и о порядке приобретения программы, найдете на web-узле Oracle www.oracle.com. Oracle WebServer обсуждается в главе 19 этой книги. |