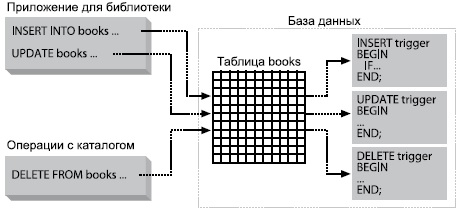
**Триггеры.**

Триггеры PL/SQL уровня команд DML (или просто триггеры DML) активизируются после вставки, обновления или удаления строк конкретной таблицы (рис. 1). Это самый распространенный тип триггеров, особенно часто применяемый разработчиками. Остальные триггеры используются преимущественно администраторами базы данных Oracle. В Oracle появилась *возможность объединения нескольких триггеров DML в один составной триггер*.

Прежде чем браться за написание триггера, необходимо ответить на следующие вопросы:

* Как триггер PL/SQL будет запускаться — по одному разу для каждой команды SQL или для каждой модифицируемой ею строки?
* Когда именно должен вызываться создаваемый триггер — до или после выполнения операции над строками?
* Для каких операций должен срабатывать триггер — вставки, обновления, удаления или их определенной комбинации?



**Рис. 1**. Схема срабатывания триггеров DML

**Основные концепции триггеров**

Прежде чем переходить к синтаксису и примерам использования триггеров DML, следует познакомиться с их концепциями и терминологией.

* **Триггер BEFORE**. Вызывается до внесения каких-либо изменений (например, BEFORE INSERT).
* **Триггер AFTER**. Выполняется для отдельной команды SQL, которая может обрабатывать одну или более записей базы данных (например, AFTER UPDATE).
* **Триггер уровня команды**. Выполняется для команды SQL в целом (которая может обрабатывать одну или несколько строк базы данных).
* **Триггер уровня записи**. Выполняется для отдельной записи, обрабатываемой командой SQL. Если, предположим, таблица books содержит 1000 строк, то следующая команда UPDATE модифицирует все эти строки:

UPDATE books SET title = UPPER (title);

И если для таблицы определен триггер уровня записи, он будет выполнен 1000 раз.

* **Псевдозапись NEW**. Структура данных с именем NEW так же выглядит и обладает (почти) такими же свойствами, как запись **PL/SQL**. Эта псевдозапись доступна только внутри триггеров обновления и вставки; она содержит значения модифицированной записи после внесения изменений.
* **Псевдозапись OLD**. Структура данных с именем OLD так же выглядит и обладает (почти) такими же свойствами, как запись PL/SQL. Эта псевдозапись доступна только внутри триггеров обновления и вставки; она содержит значения модифицированной записи до внесения изменений.
* **Секция WHEN**. Часть триггера DML, определяющая условия выполнения кода триггера (и позволяющая избежать лишних операций).

**Триггеры в транзакциях**

По умолчанию триггеры DML участвуют в транзакциях, из которых они запущены. Это означает, что:

* если триггер инициирует исключение, будет выполнен откат соответствующей части транзакции;
* если триггер сам выполнит команду DML (например, вставит запись в таблицу- журнал), она станет частью главной транзакции;
* в триггере DML нельзя выполнять команды COMMIT и ROLLBACK.

Если триггер DML определен как автономная транзакция PL/SQL, то все команды DML, выполняемые внутри триггера, будут сохраняться или отменяться (командой COMMIT или ROLLBACK) независимо от основной транзакции.

В следующем разделе описан синтаксис объявления триггера DML и приведен пример, в котором использованы многие компоненты и параметры триггеров этого типа.

**Создание триггера DML**

Команда создания (или замены) триггера DML имеет следующий синтаксис:

1 CREATE [OR REPLACE] TRIGGER имя\_триггера

2 {BEFORE | AFTER}

3 {INSERT | DELETE | UPDATE | UPDATE OF список\_столбцов } ON имя\_таблицы

4 [FOR EACH ROW]

5 [WHEN (...)]

6 [DECLARE ... ]

7 BEGIN

8 ...исполняемые команды...

9 [EXCEPTION ... ]

10 END [имя\_триггера];

Описание всех перечисленных элементов приведено в таблице.

|  |  |
| --- | --- |
| **Строки** | **Описание** |
| 1 | Создание триггера с заданным именем. Секция OR REPLACE не обязательна. Если триггер существует, а секция REPLACE отсутствует, попытка создания триггера приведет к ошибке ORA-4081. Вообще говоря, триггер и таблица могут иметь одинаковые имена (а также триггер и процедура), но мы рекомендуем использовать схемы выбора имен, предотвращающие подобные совпадения с неизбежной путаницей |
| 2 | Задание условий запуска триггера: до (BEFORE) или после (AFTER) выполнения команды либо обработки строки |
| 3 | Определение команды DML, с которой связывается триггер: INSERT, UPDATE или DELETE. Обратите внимание: триггер, связанный с командой UPDATE, может быть задан для всей строки или только для списка столбцов, разделенных запятыми. Столбцы можно объединять оператором OR и задавать в любом порядке. Кроме того, в строке 3 определяется таблица, с которой связывается данный триггер. Помните, что каждый триггер DML должен быть связан с одной таблицей |
| 4 | Если задана секция FOR EACH ROW, триггер будет запускаться для каждой обрабатываемой командой строки. Но если эта секция отсутствует, по умолчанию триггер будет запускаться только по одному разу для каждой команды (то есть будет создан триггер уровня команды) |
| 5 | Необязательная секция WHEN, позволяющая задать логику для предотвращения лишних выполнений триггера |
| 6 | Необязательный раздел объявлений для анонимного блока, составляющего код триггера. Если объявлять локальные переменные не требуется, это ключевое слово может отсутствовать. Никогда не объявляйте псевдозаписи NEW и OLD — они создаются автоматически |
| 7,8 | Исполняемый раздел триггера. Он является обязательным и должен содержать как минимум одну команду |
| 9 | Необязательный раздел исключений. В нем перехватываются и обрабатываются исключения, инициируемые только в исполняемом разделе |
| 10 | Обязательная команда END. Для наглядности в нее можно включить имя триггера |

**Рассмотрим пару примеров триггеров DML.**

* Первый триггер выполняет несколько проверок при добавлении или изменении строки в таблице сотрудников. В нем содержимое полей псевдозаписи NEW передается отдельным программам проверки:

TRIGGER validate\_employee\_changes

AFTER INSERT OR UPDATE

ON employees

FOR EACH ROW

BEGIN

check\_date (:NEW.hire\_date);

check\_email (:NEW.email);

END;

* Следующий триггер, запускаемый перед вставкой данных, проверяет изменения, производимые в таблице ceo\_compensation. Для сохранения новой строки таблицы аудита вне главной транзакции в нем используется технология автономных тран­закций:

TRIGGER bef\_ins\_ceo\_comp

BEFORE INSERT

ON ceo\_compensation

FOR EACH ROW

DECLARE

PRAGMA AUTONOMOUS\_TRANSACTION;

BEGIN

INSERT INTO ceo\_comp\_history

VALUES (:NEW.name,

:OLD.compensation, :NEW.compensation,

'AFTER INSERT', SYSDATE);

COMMIT;

END;

**Предложение WHEN**

Предложение WHEN предназначено для уточнения условий, при которых должен выполняться код триггера. В приведенном далее примере с его помощью мы указываем, что основной код триггера должен быть реализован только при изменении значения столбца salary:

TRIGGER check\_raise

AFTER UPDATE OF salary

ON employees

FOR EACH ROW

WHEN ((OLD.salary != NEW.salary) OR

(OLD.salary IS NULL AND NEW.salary IS NOT NULL) OR

(OLD.salary IS NOT NULL AND NEW.salary IS NULL))

BEGIN

Иными словами, если при обновлении записи пользователь по какой-то причине оставит salary текущее значение, триггер активизируется, но его основной код выполняться не будет. Проверяя это условие в предложении WHEN, можно избежать затрат, связанных с запуском соответствующего кода PL/SQL.

В большинстве случаев секция WHEN содержит ссылки на поля псевдозаписей NEW и OLD. В эту секцию разрешается также помещать вызовы встроенных функций, что и делается в следующем примере, где с помощью функции SYSDATE ограничивается время вставки новых записей:

TRIGGER valid\_when\_clause

BEFORE INSERT ON frame

FOR EACH ROW

WHEN ( TO\_CHAR(SYSDATE,'HH24') BETWEEN ‘9’ AND 17 )

При использовании WHEN следует соблюдать ряд правил:

* Все логические выражения всегда должны заключаться в круглые скобки. Эти скобки не обязательны в команде IF, но необходимы в секции WHEN триггера.
* Перед идентификаторами OLD и NEW не должно стоять двоеточие (:). В секции WHEN следует использовать только встроенные функции.
* Пользовательские функции и функции, определенные во встроенных пакетах (таких, как DBMS\_UTILITY), в нем вызывать нельзя. Чтобы вызвать такую функцию, переместите соответствующую логику в начало исполняемого раздела триггера.

Предложение WHEN может использоваться только в триггерах уровня записи. Поместив его в триггер уровня команды, вы получите сообщение об ошибке компиляции (ORA-04077).

**Работа с псевдозаписями NEW и OLD**

При запуске триггера уровня записи ядро PL/SQL создает и заполняет две структуры данных, имеющие много общего с записями. Речь идет о псевдозаписях NEW и OLD (префикс «псевдо» указывает на то, что они не обладают всеми свойствами записей PL/SQL). В псевдозаписи OLD хранятся исходные значения обрабатываемой триггером записи, а в псевдозаписи NEW — новые. Их структура идентична структуре записи, объявленной с атрибутом %ROWTYPE и создаваемой на основе таблицы, с которой связан триггер. Несколько правил, которые следует принимать во внимание при работе с псевдозаписями NEW и OLD:

* Для триггеров, связанных с командой INSERT, структура OLD не содержит данных, поскольку старого набора значений у операции вставки нет.
* Для триггеров, связанных с командой UPDATE, заполняются обе структуры, OLD и NEW. Структура OLD содержит исходные значения записи до обновления, а NEW — значения, которые будут содержаться в строке после обновления.
* Для триггеров, связанных с командой DELETE, заполняется только структура OLD, а структура NEW остается пустой, поскольку запись удаляется.
* Псевдозаписи NEW и OLD также содержат столбец ROWID, который в обеих псевдозаписях всегда заполняется одинаковыми значениями.
* Значения полей записи OLD изменять нельзя; попытка приведет к ошибке ORA- 04085. Значения полей структуры NEW модифицировать можно.
* Структуры NEW и OLD нельзя передавать в качестве параметров процедурам или функциям, вызываемым из триггера. Разрешается передавать лишь их отдельные поля. В сценарии gentrigrec.sp содержится программа, которая генерирует код копирования данных NEW и OLD в записи, передаваемые в параметрах.
* В ссылках на структуры NEW и OLD в анонимном блоке триггера перед соответствующими ключевыми словами необходимо ставить двоеточие:

IF :NEW.salary > 10000 THEN...

* Над структурами NEW и OLD нельзя выполнять операции уровня записи. Например, следующая команда вызовет ошибку компиляции триггера:

BEGIN :new := NULL; END;

С помощью секции REFERENCING в триггере можно менять имена псевдозаписей данных; это помогает писать самодокументированный код, ориентированный на конкретное приложение. Пример:

/\* Файл в Сети: full\_old\_and\_new.sql \*/

TRIGGER audit\_update

AFTER UPDATE

ON frame

REFERENCING OLD AS prior\_to\_cheat NEW AS after\_cheat

FOR EACH ROW

BEGIN

INSERT INTO frame\_audit

(bowler\_id,

game\_id,

old\_score,

new\_score,

change\_date,

operation)

VALUES(:after\_cheat.bowler\_id,

:after\_cheat.game\_id,

:prior\_to\_cheat.score,

:after\_cheat.score,

SYSDATE,

'UPDATE');

END;

**Идентификация команды DML в триггере**

Oracle предоставляет набор функций (также называемых операционными директивами) для идентификации команды DML, вызвавшей запуск триггера:

* INSERTING — возвращает TRUE, если триггер запущен в ответ на вставку записи в таблицу, с которой он связан, и FALSE в противном случае.
* UPDATING — возвращает TRUE, если триггер запущен в ответ на обновление записи в таблице, с которой он связан, и FALSE в противном случае.
* DELETING — возвращает TRUE, если триггер запущен в ответ на удаление записи из таблицы, с которой он связан, и FALSE в противном случае.

Пользуясь этими директивами, можно создать один триггер, который объединяет действия для нескольких операций. Пример:

/\* Файл в Сети: one\_trigger\_does\_it\_all.sql \*/

TRIGGER three\_for\_the\_price\_of\_one

BEFORE DELETE OR INSERT OR UPDATE ON account\_transaction

FOR EACH ROW

BEGIN

-- Сохранение информации о пользователе, вставившем новую строку

IF INSERTING

THEN

:NEW.created\_by := USER;

:NEW.created\_date := SYSDATE;

-- Сохранение информации об удалении с помощью специальной программы

ELSIF DELETING

THEN

audit\_deletion(USER,SYSDATE);

-- Сохранение информации о пользователе, который последним обновлял строку

ELSIF UPDATING

THEN

:NEW.UPDATED\_BY := USER;

:NEW.UPDATED\_DATE := SYSDATE;

END IF;

END;

|  |
| --- |
|  |

Функция UPDATING имеет перегруженную версию, которой в качестве аргумента передается имя конкретного столбца. Перегрузка функций является удобным способом изоляции операций обновления отдельных столбцов.

/\* Файл в Сети: overloaded\_update.sql \*/

TRIGGER validate\_update

BEFORE UPDATE ON account\_transaction

FOR EACH ROW

BEGIN

IF UPDATING ('ACCOUNT\_NO')

THEN

errpkg.raise('Account number cannot be updated');

END IF;

END;

В спецификации имени столбца игнорируется регистр символов. Имя столбца до за­пуска триггера не анализируется, и если в таблице, связанной с триггером, заданного столбца не оказывается, функция просто возвращает FALSE.

Операционные директивы можно вызывать из любого кода PL/SQL, а не только из триггеров. Однако значение TRUE они возвращают лишь при использовании в триггерах DML или вызываемых из них программах.

**Пример триггера DML**

Одной из идеальных областей для применения триггеров является аудит изменений. Допустим, Памела, хозяйка боулинга, стала получать жалобы на посетителей, которые жульничают со своими результатами. Памела недавно написала приложение для ведения счета в игре и теперь хочет усовершенствовать его, чтобы выявить нечестных игроков.

В приложении Памелы центральное место занимает таблица frame, в которой записывается результат конкретного фрейма конкретной партии конкретного игрока:

/\* Файл в Сети: bowlerama\_tables.sql \*/

TABLE frame

(bowler\_id NUMBER,

game\_id NUMBER,

frame\_number NUMBER,

strike VARCHAR2(1) DEFAULT 'N',

spare VARCHAR2(1) DEFAULT 'N',

score NUMBER,

CONSTRAINT frame\_pk

PRIMARY KEY (bowler\_id, game\_id, frame\_number))

Памела дополняет таблицу frame версией, в которой сохраняются все значения «до» и «после», чтобы она могла сравнить их и выявить несоответствия:

TABLE frame\_audit

(bowler\_id NUMBER,

game\_id NUMBER,

frame\_number NUMBER,

old\_strike VARCHAR2(1),

new\_strike VARCHAR2(1),

old\_spare VARCHAR2(1),

new\_spare VARCHAR2(1),

old\_score NUMBER,

new\_score NUMBER,

change\_date DATE,

operation VARCHAR2(6))

Для каждого изменения в таблице frame Памела хочет отслеживать состояние строки до и после изменения. Она создает простой триггер:

/\* Файл в Сети: bowlerama\_full\_audit.sql \*/

1 TRIGGER audit\_frames

2 AFTER INSERT OR UPDATE OR DELETE ON frame

3 FOR EACH ROW

4 BEGIN

5 IF INSERTING THEN

6 INSERT INTO frame\_audit(bowler\_id,game\_id,frame\_number,

7 new\_strike,new\_spare,new\_score,

8 change\_date,operation)

9 VALUES(:NEW.bowler\_id,:NEW.game\_id,:NEW.frame\_number,

10 :NEW.strike,:NEW.spare,:NEW.score,

11 SYSDATE,'INSERT');

12

13 ELSIF UPDATING THEN

14 INSERT INTO frame\_audit(bowler\_id,game\_id,frame\_number,

15 old\_strike,new\_strike,

16 old\_spare,new\_spare,

17 old\_score,new\_score,

18 change\_date,operation)

19 VALUES(:NEW.bowler\_id,:NEW.game\_id,:NEW.frame\_number,

20 :OLD.strike,:NEW.strike,

21 :OLD.spare,:NEW.spare,

22 :OLD.score,:NEW.score,

23 SYSDATE,'UPDATE');

24

25 ELSIF DELETING THEN

26 INSERT INTO frame\_audit(bowler\_id,game\_id,frame\_number,

27 old\_strike,old\_spare,old\_score,

28 change\_date,operation)

29 VALUES(:OLD.bowler\_id,:OLD.game\_id,:OLD.frame\_number,

30 :OLD.strike,:OLD.spare,:OLD.score,

31 SYSDATE,'DELETE');

32 END IF;

33 END audit\_frames;

В секции INSERTING (строки 6-11) для заполнения строки аудита используется псев­дозапись NEW. Для UPDATING (строки 14-23) используется сочетание информации NEW и OLD. Для DELETING (строки 26-31) доступна только информация OLD. Памела создает триггер и ждет результатов.

Конечно, она не распространяется о своей новой системе. Салли — амбициозный, но не очень искусный игрок — понятия не имеет, что ее действия могут отслеживаться. Салли решает, что в этом году она должна стать чемпионом, и она не остановится ни перед чем. У нее есть доступ к SQI\*Plus, и она знает, что ее идентификатор игрока равен 1. Салли располагает достаточной информацией, чтобы полностью обойти графический интерфейс, подключиться к SQL\*Plus и пустить в ход свое беспринципное «волшебство». Салли сходу выписывает себе страйк в первом фрейме:

SQL> INSERT INTO frame

2 (BOWLER\_ID,GAME\_ID,FRAME\_NUMBER,STRIKE)

3 VALUES(1,1,1,'Y');

1 row created.

Но затем она решает умерить аппетит и понижает результат первого фрейма, чтобы вызвать меньше подозрений:

SQL> UPDATE frame

2 SET strike = 'N',

3 spare = 'Y'

4 WHERE bowler\_id = 1

5 AND game\_id = 1

6 AND frame\_number = 1;

1 row updated.

Но что это? Салли слышит шум в коридоре. Она теряет самообладание и пытается замести следы:

SQL> DELETE frame

2 WHERE bowler\_id = 1

3 AND game\_id = 1

4 AND frame\_number = 1;

1 row deleted.

SQL> COMMIT;

Commit complete.

Она даже убеждается в том, что ее исправления были удалены:

SQL> SELECT \* FROM frame;

no rows selected

Вытирая пот со лба, Салли завершает сеанс, но рассчитывает вернуться и реализовать свои планы.

Бдительная Памела подключается к базе данных и моментально обнаруживает, что пыталась сделать Салли. Для этого она выдает запрос к таблице аудита (кстати, Памела также может настроить ежечасный запуск задания через DBMS\_JOB для автоматизации этой части аудита):

SELECT bowler\_id, game\_id, frame\_number

, old\_strike, new\_strike

, old\_spare, new\_spare

, change\_date, operation

FROM frame\_audit

Результат:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **BOWLER\_ID** | **GAME\_ID** | **FRAME\_NUMBER** | **O** | **N** | **O** | **N** | **CHANGE\_DA** | **OPERAT** |
| 1 | 1 | 1 |  | Y |  | N | 12-SEP-00 | INSERT |
| 1 | 1 | 1 | Y | N | N | Y | 12-SEP-00 | UPDATE |
| 1 | 1 | 1 | N |  | N |  | 12-SEP-00 | DELETE |

Салли поймана с поличным! Из записей аудита прекрасно видно, что она пыталась сделать, хотя в таблице frame никаких следов не осталось. Все три команды — исходная вставка записи, понижение результата и последующее удаление записи — были пере­хвачены триггером DMI.

**Применение секции WHEN**

После того как система аудита успешно проработала несколько месяцев, Памела при­нимает меры для дальнейшего устранения потенциальных проблем. Она просматривает интерфейсную часть своего приложения и обнаруживает, что изменяться могут только поля strike, spare и score. Следовательно, триггер может быть и более конкретным:

TRIGGER audit\_update

AFTER UPDATE OF strike, spare, score

ON frame

REFERENCING OLD AS prior\_to\_cheat NEW AS after\_cheat

FOR EACH ROW

BEGIN

INSERT INTO frame\_audit (...)

VALUES (...);

END;

Проходит несколько недель. Памеле не нравится ситуация, потому что новые записи создаются даже тогда, когда значения задаются равными сами себе. Обновления вроде следующего создают бесполезные записи аудита, которые показывают лишь отсутствие изменений:

SQL> UPDATE FRAME

2 SET strike = strike;

1 row updated.

SQL> SELECT old\_strike,

2 new\_strike,

3 old\_spare,

4 new\_spare,

5 old\_score,

6 new\_score

7 FROM frame\_audit;

O N O N OLD SCORE NEW SCORE

- - - - --------- ---------

Y Y N N

Триггер нужно дополнительно уточнить, чтобы он срабатывал только при фактическом изменении значений. Для этого используется секция WHEN:

/\* Файл в Сети: final\_audit.sql \*/

TRIGGER audit\_update

AFTER UPDATE OF STRIKE, SPARE, SCORE ON FRAME

REFERENCING OLD AS prior\_to\_cheat NEW AS after\_cheat

FOR EACH ROW

WHEN ( prior\_to\_cheat.strike != after\_cheat.strike OR

prior\_to\_cheat.spare != after\_cheat.spare OR

prior\_to\_cheat.score != after\_cheat.score )

BEGIN

INSERT INTO FRAME\_AUDIT ( ... )

VALUES ( ... );

END;

Теперь данные будут появляться в таблице аудита только в том случае, если данные действительно изменились, а Памеле будет проще выявить возможных мошенников. Небольшая завершающая проверка триггера:

SQL> UPDATE frame

2 SET strike = strike;

1 row updated.

SQL> SELECT old\_strike,

2 new\_strike,

3 old\_spare,

4 new\_spare,

5 old\_score,

6 new\_score

7 FROM frame\_audit;

no rows selected

**Использование псевдозаписей для уточнения триггеров**

Памела реализовала в системе приемлемый уровень аудита; теперь ей хотелось бы сделать систему более удобной для пользователя. Самая очевидная идея — сделать так, чтобы система сама увеличивала счет во фреймах, заканчивающихся страйком или спэром, на 10. Это позволяет счетчику отслеживать счет только за последующие броски, а счет за страйк будет начисляться автоматически:

/\* Файл в Сети: set\_score.sql \*/

TRIGGER set\_score

BEFORE INSERT ON frame

FOR EACH ROW

WHEN ( NEW.score IS NOT NULL )

BEGIN

IF :NEW.strike = 'Y' OR :NEW.spare = 'Y'

THEN

:NEW.score := :NEW.score + 10;

END IF;

END;

Помните, что значения полей в записях NEW могут изменяться только в BEFORE- триггерах строк.

Будучи человеком пунктуальным, Памела решает добавить проверку счета в ее набор триггеров:

/\* File on Сети: validate\_score.sql \*/

TRIGGER validate\_score

AFTER INSERT OR UPDATE

ON frame

FOR EACH ROW

BEGIN

IF :NEW.strike = 'Y' AND :NEW.score < 10

THEN

RAISE\_APPLICATION\_ERROR (

-20001,

'ERROR: Score For Strike Must Be >= 10'

);

ELSIF :NEW.spare = 'Y' AND :NEW.score < 10

THEN

RAISE\_APPLICATION\_ERROR (

-20001,

'ERROR: Score For Spare Must Be >= 10'

);

ELSIF :NEW.strike = 'Y' AND :NEW.spare = 'Y'

THEN

RAISE\_APPLICATION\_ERROR (

-20001,

'ERROR: Cannot Enter Spare And Strike'

);

END IF;

END;

Теперь любая попытка ввести строку, нарушающую это условие, будет отклонена:

SQL> INSERT INTO frame VALUES(1,1J1J'Y'JNULLJ5);

2 INSERT INTO frame \*

ERROR at line 1:

ORA-20001: ERROR: Score For Strike Must >= 10

|  |  |
| --- | --- |
| |  | | --- | |  | |

**Однотипные триггеры**

Oracle позволяет связать с таблицей базы данных несколько триггеров одного типа. Рассмотрим такую возможность на примере, связанном с игрой в гольф. Следующий триггер уровня строки вызывается при вставке в таблицу новой записи и добавляет в нее комментарий, текст которого определяется соотношением текущего счета с но­минальным значением 72:

/\* Файл в Сети: golf\_commentary.sql \*/

TRIGGER golf\_commentary

BEFORE INSERT

ON golf\_scores

FOR EACH ROW

DECLARE

c\_par\_score CONSTANT PLS\_INTEGER := 72;

BEGIN

:new.commentary :=

CASE

WHEN :new.score < c\_par\_score THEN 'Under'

WHEN :new.score = c\_par\_score THEN NULL

ELSE 'Over' || ' Par'

END;

END;

Эти же действия можно выполнить и с помощью трех отдельных триггеров уровня строки типа BEFORE INSERT с взаимоисключающими условиями, задаваемыми в секциях WHEN:

TRIGGER golf\_commentary\_under\_par

BEFORE INSERT ON golf\_scores

FOR EACH ROW

WHEN (NEW.score < 72)

BEGIN

:NEW.commentary := 'Under Par';

END;

TRIGGER golf\_commentary\_par

BEFORE INSERT ON golf\_scores

FOR EACH ROW

WHEN (NEW.score = 72)

BEGIN

:NEW.commentary := 'Par';

END;

TRIGGER golf\_commentary\_over\_par

BEFORE INSERT ON golf\_scores

FOR EACH ROW

WHEN (NEW.score > 72)

BEGIN

:NEW.commentary := 'Over Par';

END;

Обе реализации абсолютно допустимы, и каждая обладает своими достоинствами и не­достатками. Решение с одним триггером проще в сопровождении, поскольку весь код сосредоточен в одном месте, а решение с несколькими триггерами сокращает время синтаксического анализа и выполнения при необходимости более сложной обработки.

**Очередность вызова триггеров**

До выхода Oracle llg порядок срабатывания нескольких триггеров DML был непред­сказуемым. В рассмотренном примере он несущественен, но как показывает следующий пример, в других ситуациях могут возникнуть проблемы. Какой результат будет получен для последнего запроса?

/\* Файл в Сети: multiple\_trigger\_seq.sql \*/

TABLE incremented\_values

(value\_inserted NUMBER,

value\_incremented NUMBER);

TRIGGER increment\_by\_one

BEFORE INSERT ON incremented\_values

FOR EACH ROW

BEGIN

:NEW.value\_incremented := :NEW.value\_incremented + 1;

END;

TRIGGER increment\_by\_two

BEFORE INSERT ON incremented\_values

FOR EACH ROW

BEGIN

IF :NEW.value\_incremented > 1 THEN

:NEW.value\_incremented := :NEW.value\_incremented + 2;

END IF;

END;

INSERT INTO incremented\_values

VALUES(1,1);

SELECT \*

FROM incremented\_values;

SELECT \*

FROM incremented\_values;

Есть какие-нибудь предположения? Для моей базы данных результаты получились такими:

SQL> SELECT \*

2 FROM incremented\_values;

VALUE INSERTED VALUE INCREMENTED

-------------- -----------------

1 2

Это означает, что первым сработал триггер increment\_by\_two, который не выполнил ни­каких действий, потому что значение столбца value\_incremented не превышало 1; затем сработал триггер increment\_by\_one, увеличивший значение столбца value\_incremented на 1. А вы тоже получите такой результат? Вовсе не обязательно. Будет ли этот результат всегда одним и тем же? Опять-таки, ничего нельзя гарантировать. До выхода Oracle 11g в документации Oracle было явно указано, что порядок запуска однотипных триггеров, связанных с одной таблицей, не определен и произволен, поэтому задать его явно невоз­можно. На этот счет существуют разные теории, среди которых наиболее популярны две: триггеры запускаются в порядке, обратном порядку их создания или же в соответствии с идентификаторами их объектов, но полагаться на такие предположения не стоит. Начиная с Oracle11g гарантированный порядок срабатывания триггеров может опре­деляться при помощи условия FOLLOWS, как показано в следующем примере:

TRIGGER increment\_by\_two

BEFORE INSERT ON incremented\_values

FOR EACH ROW

FOLLOWS increment\_by\_one

BEGIN

IF :new.value\_incremented > 1 THEN

:new.value\_incremented := :new.value\_incremented + 2;

END IF;

END;

Теперь этот триггер заведомо будет активизирован раньше триггера increment\_by\_one. Тем самым гарантируется и результат вставки:

SQL> INSERT INTO incremented\_values

2 VALUES(1,1);

1 row created.

SQL> SELECT \*

2 FROM incremented\_values;

VALUE INSERTED VALUE INCREMENTED

-------------- -----------------

1 4

Триггер increment\_by\_one увеличил вставленное значение до 2, а триггер increment\_by\_two увеличил его до 4. Такое поведение гарантировано, потому что оно определяется на уровне самого триггера — нет необходимости полагаться на догадки и предположения. Связи последовательности триггеров можно просмотреть в представлении зависимостей словаря данных Oracle:

SQL> SELECT referenced\_name,

2 referenced\_type,

3 dependency\_type

4 FROM user\_dependencies

5 WHERE name = 'INCREMENT\_BY\_TWO'

6 AND referenced\_type = 'TRIGGER';

REFERENCED NAME REFERENCED TYPE DEPE

---------------- --------------- ----

INCREMENT\_BY\_ONE TRIGGER REF

Несмотря на поведение, описанное выше для Oracle Database 11g, при попытке отком­пилировать триггер, следующий за неопределенным триггером, выводится сообщение об ошибке:

Trigger "SCOTT"."BLIND\_FOLLOWER" referenced in FOLLOWS or PRECEDES clause may not

exist

**Ошибки при изменении таблицы**

Изменяющиеся объекты трудно анализировать и оценивать. Поэтому когда триггер уровня строки пытается прочитать или изменить данные в таблице, находящейся в со­стоянии изменения (с помощью команды INSERT, UPDATE или DELETE), происходит ошибка с кодом ORA-4091.

В частности, эта ошибка встречается тогда, когда триггер уровня строк пытается вы­полнить чтение или запись в таблицу, для которой сработал триггер. Предположим, для таблицы сотрудников требуется задать ограничение на значения в столбцах, которое заключается в том, что при повышении оклада сотрудника его новое значение не должно превышать следующее значение по отделу более чем на 20%.

Казалось бы, для проверки этого условия можно использовать следующий триггер:

TRIGGER brake\_on\_raises

BEFORE UPDATE OF salary ON employee

FOR EACH ROW

DECLARE

l\_curr\_max NUMBER;

BEGIN

SELECT MAX (salary) INTO l\_curr\_max

FROM employee;

IF l\_curr\_max \* 1.20 < :NEW.salary

THEN

errpkg.RAISE (

employee\_rules.en\_salary\_increase\_too\_large,

:NEW.employee\_id,

:NEW.salary

);

END IF;

END;

Однако при попытке удвоить, скажем, оклад программиста PL/SQL, Oracle выдаст сообщение об ошибке:

ORA-04091: table SCOTT.EMPLOYEE is mutating, trigger/function may not see it

Тем не менее некоторые приемы помогут предотвратить выдачу этого сообщения об ошибке:

* В общем случае триггер уровня строки не может считывать или записывать данные таблицы, с которой он связан. Но подобное ограничение относится только к тригге­рам уровня строки. Триггеры уровня команд могут и считывать, и записывать данные своей таблицы, что дает возможность произвести необходимые действия.
* Если триггер выполняется как автономная транзакция (директива PRAGMA AUTONOMOUS TRANSACTION и выполнение COMMIT в теле триггера), тогда в нем можно запрашивать содержимое таблицы. Однако модификация такой таблицы все равно будет запре­щена.

С каждым выпуском Oracle проблема ошибок изменения таблицы становится все менее актуальной, поэтому мы не станем приводить полное описание.

## Составные триггеры

По мере создания триггеров, содержащих все больший объем бизнес-логики, становится трудно следить за тем, какие триггеры связаны с теми или иными правилами и как триггеры взаимодействуют друг с другом. В предыдущем разделе было показано, как три типа команд DML (вставка, обновление, удаление) объединяются в одном триггере, но разве не удобно было бы разместить триггеры строк и команд вместе в одном объекте кода? В Oracle Database 11g появилась возможность использования составных триггеров для решения этой задачи. Следующий простой пример демонстрирует этот синтаксис:

/\* Файл в Сети: compound\_trigger.sql \*/

1 TRIGGER compounder

2 FOR UPDATE OR INSERT OR DELETE ON incremented\_values

3 COMPOUND TRIGGER

4

5 v\_global\_var NUMBER := 1;

6

7 BEFORE STATEMENT IS

8 BEGIN

9 DBMS\_OUTPUT.PUT\_LINE('Compound:BEFORE S:' || v\_global\_var);

10 v\_global\_var := v\_global\_var + 1;

11 END BEFORE STATEMENT;

12

13 BEFORE EACH ROW IS

14 BEGIN

15 DBMS\_OUTPUT.PUT\_LINE('Compound:BEFORE R:' || v\_global\_var);

16 v\_global\_var := v\_global\_var + 1;

17 END BEFORE EACH ROW;

18

19 AFTER EACH ROW IS

20 BEGIN

21 DBMS\_OUTPUT.PUT\_LINE('Compound:AFTER R:' || v\_global\_var);

22 v\_global\_var := v\_global\_var + 1;

23 END AFTER EACH ROW;

24

25 AFTER STATEMENT IS

26 BEGIN

27 DBMS\_OUTPUT.PUT\_LINE('Compound:AFTER S:' || v\_global\_var);

28 v\_global\_var := v\_global\_var + 1;

29 END AFTER STATEMENT;

30

31 END;

### Сходство с пакетами

Составные триггеры похожи на пакеты PL/SQL, не правда ли? Весь сопутствующий код и логика находятся в одном месте, что упрощает его отладку и изменение. Рассмотрим синтаксис более подробно.

Самое очевидное изменение — конструкция COMPOUND TRIGGER, сообщающая Oracle, что триггер содержит несколько триггеров, которые должны срабатывать вместе. Следующее (и пожалуй, самое долгожданное) изменение встречается в строке 5: глобальная переменная! Наконец-то глобальные переменные могут определяться вместе с кодом, который с ними работает, — специальные пакеты для них больше не нужны:

PACKAGE BODY yet\_another\_global\_package AS

v\_global\_var NUMBER := 1;

PROCEDURE reset\_global\_var IS

...

END;

В остальном синтаксис составных триггеров очень похож на синтаксис автономных триггеров, но не так гибок:

* BEFORE STATEMENT — код этого раздела выполняется до команды DML, как и в случае с автономным триггером BEFORE.
* BEFORE EACH ROW — код этого раздела выполняется перед обработкой каждой строки командой DML.
* AFTER EACH ROW — код этого раздела выполняется после обработки каждой строки командой DML.
* AFTER STATEMENT — код этого раздела выполняется после команды DML, как и в случае с автономным триггером AFTER.

Правила автономных триггеров также применимы и к составным триггерам — например, значения записей (OLD и NEW) не могут изменяться в триггерах уровня команд.

### Различия с пакетами

Итак, составные триггеры похожи на пакеты PL/SQL, но означает ли это, что они так же работают? Нет — они работают лучше! Рассмотрим следующий пример:

SQL> BEGIN

2 insert into incremented\_values values(1,1);

3 insert into incremented\_values values(2,2);

4 END;

5 /

Compound:BEFORE S:1

Compound:BEFORE R:2

Compound:AFTER R:3

Compound:AFTER S:4

Compound:BEFORE S:1

Compound:BEFORE R:2

Compound:AFTER R:3

Compound:AFTER S:4

PL/SQL procedure successfully completed.

Обратите внимание: при выполнении второй команды для глобальной переменной снова выводится 1. Это связано с тем, что область действия составного триггера огра­ничивается командой DML, которая его инициирует. После завершения этой команды составной триггер и его значения, хранящиеся в памяти, перестают существовать. Это обстоятельство упрощает логику.

Дополнительным преимуществом ограниченной области действия является упрощен­ная обработка ошибок. Чтобы продемонстрировать это обстоятельство, я определяю в таблице первичный ключ для последующего нарушения:

SQL> ALTER TABLE incremented\_values

2 add constraint a\_pk

3 primary key ( value\_inserted );

Теперь вставим одну запись:

SQL> INSERT INTO incremented\_values values(1,1);

Compound:BEFORE S:1

Compound:BEFORE R:2

Compound:AFTER R:3

Compound:AFTER S:4

1 row created.

Пока без сюрпризов. Но следующая команда INSERT выдает ошибку из-за нарушения нового первичного ключа:

SQL> INSERT INTO incremented\_values values(1,1);

Compound:BEFORE S:1

Compound:BEFORE R:2

insert into incremented\_values values(1,1)

\*

ERROR at line 1:

ORA-00001: unique constraint (SCOTT.A\_PK) violated

Следующая команда INSERT также снова выдает ошибку первичного ключа. Но в этом как раз ничего примечательного нет — примечательно то, что глобальная переменная была снова инициализирована значением 1 без написания дополнительного кода. Команда DML завершилась, составной триггер вышел из области действия, и со следующей командой все начинается заново:

SQL> INSERT INTO incremented\_values values(1,1);

Compound:BEFORE S:1

Compound:BEFORE R:2

insert into incremented\_values values(1,1)

\*

ERROR at line 1:

ORA-00001: unique constraint (DRH.A\_PK) violated

Теперь мне не нужно включать дополнительную обработку ошибок или пакеты только для сброса значений при возникновении исключения.

### FOLLOWS с составными триггерами

Составные триггеры также могут использоваться с синтаксисом FOLLOWS:

TRIGGER follows\_compounder

BEFORE INSERT ON incremented\_values

FOR EACH ROW

FOLLOWS compounder

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Following Trigger');

END;

Результат:

SQL> INSERT INTO incremented\_values

2 values(8,8);

Compound:BEFORE S:1

Compound:BEFORE R:2

Following Trigger

Compound:AFTER R:3

Compound:AFTER S:4

1 row created.

Конкретные триггеры, находящиеся внутри составного триггера, не могут определяться как срабатывающие после каких-либо автономных или составных триггеров.

Если автономный триггер определяется как следующий за составным триггером, который не содержит триггер, срабатывающий по той же команде или строке, секция FOLLOWS просто игнорируется.

# **Использование планировщика Oracle Scheduler**

Oracle Database 11g и 12c предлагает новое средство **Scheduler (Планировщик)**, которое помогает **автоматизировать задания** внутри **базы данных Oracle**. Пакет DBMS\_SCHEDULER содержит различные функции и процедуры, помогающие управлять планировщиком, хотя также можно очень легко планировать запуск заданий в интерфейсе Database Control. Наиболее важная архитектурная особенность Scheduler состоит в его модульном подходе к управлению заданиями, что позволяет повторно использовать сходные задания.

Использование Scheduler вместе с Database Resource Manager позволяет осуществлять тонкую настройку выделения ресурсов различным заданиям. Oracle Scheduler — не только инструмент описания заданий (jobs), он также помогает контролировать использование ресурсов и устанавливать приоритеты заданий внутри базы данных.

Одним из ограничений пакета DBMS\_JOB является то, что он может выполнять только задания на базе PL/SQL, и его нельзя использовать для планирования запуска системных сценариев либо исполняемых файлов. Чтобы запускать такие задания, не связанные с базой данных, до сих пор нужно было применять средство crontab в UNIX или at на серверах Windows или же обращаться к инструментам от независимых поставщиков. Oracle Scheduler позволяет использовать сценарии PL/SQL, сценарии оболочки операционной системы, программы Java, и родные двоичные исполняемые файлы для выполнения запланированных заданий.

## Базовые компоненты Scheduler

Scheduler состоит из пяти базовых компонентов — **заданий (jobs)**, **расписаний (schedules)**, **программ**, **событий** и **цепочек (chains)**. Задания очень похожи на задания в пакете DBMS\_JOB, но расписания, программы, события и цепочки — это новые концепции, формирующие модульный подход к управлению заданиями. Программа, например, позволяет нескольким пользователям выполнять сходные задачи.

Рассмотрим базовые компоненты Scheduler более подробно.

### *Задания (jobs)*

Задание — это задача, которая планируется для однократного или многократного автоматического запуска. Задание содержит спецификацию того, что и когда должно быть выполнено. Задание Scheduler может выполнять блок кода PL/SQL, двоичный родной исполняемый код, приложение Java или сценарий оболочки. Новое задание (jobs) создается за счет спецификации таких его деталей, как действия, которые должны быть выполнены, время и частота запуска — как это делается с помощью традиционного пакета DBMS\_JOB. В Scheduler можно абстрагировать все детали выполнения и времени запуска задания, используя модули программ (program) или расписаний (schedule).

### *Расписания (schedules)*

Расписание — это спецификация того, когда и как часто база данных должна выполнять задание. Одно и то же расписание может использоваться для нескольких заданий (jobs). Можно также запланировать выполнение задания при возникновении определенного события в базе данных.

### *Программы*

Программа содержит метаданные о задании Scheduler. Программа включает имя, тип (например, код PL/SQL или сценарий оболочки UNIX) и действие программы, которое представляет собой действительное имя процедуры или исполняемого сценария. Обратите внимание, что задание может специфицировать то, что именно следует выполнить, непосредственно в определении задания, или же использовать предварительно созданную программу для той же цели.

### *События*

Scheduler использует средство Oracle Streams Advanced Queuing для инициации событий и запуска заданий базы данных, основанных на событиях. Событие — это сообщение, посланное приложением или процессом в ответ на какое-то действие или условие.

Существуют два типа событий — события, инициируемые Scheduler, и события, инициируемые приложением. События, инициируемые Scheduler, вызваны изменениями в функционировании Scheduler, так что успешное завершение задания (jobs) Scheduler тоже может быть событием. Событие, инициируемые приложением, “потребляются” или используются Scheduler для запуска задания. Фактически, в качестве средства запуска задания есть возможность использовать событие вместо расписания. Можно также базировать расписание на событии; в этом случае расписание называется event schedule (событийное расписание).

### *Цепочки*

Концепция цепочки Scheduler служит для связи вместе взаимозависимых программ. Таким образом, запуск определенной программы может быть увязан с успешным выполнением некоторых других программ. Также можно запустить задание, основанное на цепочке, вместо единственной запланированной программы. Когда имеются взаимозависимые задания, цепочка облегчает запуск всех программ, необходимых для завершения целой транзакции.

## Типы запланированных заданий

Scheduler предлагает на выбор из следующих типов заданий: задания базы данных, задания-цепочки, внешние задания, отсоединенные задания и легковесные задания. Рассмотрим все эти типы заданий внимательней.

### *Задания базы данных*

Задания базы данных — наиболее распространенные задания Scheduler (jobs), и здесь они называются просто заданиями. Задания базы данных Oracle запускают такие модули, как анонимные блоки PL/SQL или хранимые процедуры, в дополнение к хранимым процедурам Java. Атрибут JOB\_TYPE потребуется установить в PLSQL\_BLOCK или STORED\_PROCEDURE. В качестве значения атрибута JOB\_ACTION нужно специфицировать имя анонимного блока или хранимой процедуры.

### *Задания-цепочки*

Задания-цепочки позволяют использовать условное планирование запуска заданий. Цепочка определяет набор программ с зависимостями между ними. Задание может указывать на такую цепочку, тем самым включая в себя целое множество заданий.

### *Внешние задания*

Внешние задания служат для запуска исполняемых программ операционной системы извне базы данных. Для внешних заданий JOB\_TYPE устанавливается в EXECUTABLE.

### *Отсоединенные задания*

Отсоединенные задания (jobs) служат для запуска сценария или приложения в виде независимого процесса. Значение атрибута DETACHED для отсоединенного задания равно TRUE.

### *Легковесные задания*

Легковесные задания отличаются от обычных заданий Scheduler в том, что зависят от шаблона, наследуя его привилегии и метаданные задания. Легковесные задания влекут за собой меньшие накладные расходы, они быстрее создаются и уничтожаются, что делает их незаменимыми в том случае, когда требуется запустить множество мелких заданий (jobs).

В дальнейшем если не оговаривается специально, то под заданиями базы данных имеются в виду обычные задания OracleScheduler, а не специализированный их тип, такой как внешнее или легковесное задание.

## Расширенные компоненты Scheduler

В дополнение к пяти базовым компонентам Scheduler — заданиям, расписаниям, программам, цепочкам и событиям — Scheduler также использует несколько расширенных компонентов: классы заданий, окна и группы окон. Эти расширенные средства отличают Scheduler от его предшественника — пакета DBMS\_JOB. Именно эти развитые концепции позволяют присваивать заданиям приоритеты и выделять ресурсы в соответствии с приоритетами предприятия. Давайте кратко познакомимся с расширенными компонентами Scheduler.

### *Классы заданий*

Классы заданий группируют общие их характеристики, вроде требований к ресурсам. Классы заданий позволяют выделять ресурсы заданиям, группируя сходные типы заданий вместе. Классы заданий используются для достижения следующих целей.

* Назначать уровни приоритетов индивидуальным заданиям, причем более высокоприоритетные задания выполняются перед низкоприоритетными.
* Специфицировать общие атрибуты для наборов заданий (jobs).

Для выделения дефицитных ресурсов в базе данных применяется Database Resource Manager в координации с Scheduler. В Database Resource Manager концепция группы потребителей ресурсов позволяет группировать пользователей в соответствии с потреблением ресурсов. Oracle контролирует выделение ресурсов, назначая каждый класс заданий в определенную группу потребителей. По умолчанию класс заданий назначается в группу потребителей по умолчанию.

### *Окна*

Окна Scheduler предлагают ссылку на Oracle Resource Manager. Окно в данном контексте представляет интервал времени, на который можно запланировать запуск задания; целью использования окон является изменение выделения ресурсов в определенный период времени. Каждое окно ассоциировано с определенным планом ресурсов, который создается с помощью Database Resource Manager. Используя окна, можно активизировать различные планы ресурсов во время разных периодов времени, предлагая различные приоритеты заданий (jobs).

### *Группы окон*

Группа окон — это коллекция сходных окон. Например, можно создать окно на выходные и окно на праздники, и сгруппировать оба окна в одну группу окон обслуживания.

## Архитектура планировщика

Архитектура Scheduler состоит из таблицы заданий, координатора заданий и исполнителей заданий (или подчиненных (slave), как их называет Oracle).

Таблица заданий содержит информацию о заданиях, такую как наименование задания, имя программы и владельца задания. Просмотреть данные этой таблицы можно через представление DBA\_SCHEDULER\_JOBS. Координатор заданий регулярно заглядывает в таблицу заданий, чтобы узнать, какие нужно выполнить задания. Координатор заданий создает и управляет процессами исполнителей заданий, которые, собственно, и делают работу.

При создании нового или запуске существующего задания фоновый процесс (cjqnnn) просыпается и координирует запуск задания. Когда координатор заданий указывает исполнителю выполнить задание, процесс исполнителя запускает новый сеанс базы данных и новую транзакцию. Он выполняет задание, и по завершении фиксирует результаты, завершая транзакцию и закрывая сеанс базы данных. Исполнитель задания обновляет таблицу заданий, счетчик запусков и таблицу журнала заданий.

## Привилегии планировщика

Oracle создает все jobs, программы и расписания в схеме пользователя, создавшего эти объекты, а кроме этого создает такие расширенные компоненты Scheduler, как классы заданий, окна и группы окон, на уровне базы данных, и их владельцем является схема SYS.

Роль SCHEDULER\_ADMIN содержит все системные привилегии Scheduler, с опцией WITH ADMIN OPTION. Роль администратора баз данных включает роль SCHEDULER\_ADMIN.

Системная привилегия MANAGE SCHEDULER позволяет выполнять следующие действия.

* Создание, удаление и изменение классов заданий, окон и групп окон.
* Останов любого jobs.
* Преждевременный запуск и останов окна.

**На заметку!** Все объекты Scheduler имеют форму [схема].имя. По умолчанию все имена объектов планировщика записываются в верхнем регистре, если только не заключить имя в нижнем регистре в кавычки, например, "test\_job".

Для создания компонентов Scheduler (заданий, расписаний, программ, цепочек и событий) необходимо иметь привилегию CREATE JOB. Чтобы использовать расширенные компоненты Scheduler (окна, группы окон и классы заданий), понадобится системная привилегия MANAGE SCHEDULER.

Выдавая другим пользователям привилегии EXECUTE на одном из своих компонентов, вы предоставляете им право пользоваться этим компонентом.

* Привилегия EXECUTE ANY PROGRAM разрешает пользователю выполнять любую программу в любой схеме.
* Привилегия EXECUTE ANY CLASS дает возможность назначать задание в любой класс заданий.

Чтобы пользователи могли модифицировать компоненты Scheduler, они должны применить SQL-оператор GRANT ALTER для каждого компонента Scheduler.

**На заметку!** Чтобы иметь возможность создавать задание (jobs) в сгенерированном вами классе заданий, необходимо располагать отдельной привилегией EXECUTE на этом классе заданий.

Относительно привилегий Scheduler следует отметить некоторые ключевые моменты.

* Чтобы создать задание, необходимо иметь привилегию CREATE JOB.
* Для спецификации расписания, окна, группа окон или программа, которыми вы владеете, никакие специальные привилегии не нужны.
* Для указания программы, принадлежащей другому пользователю, понадобится привилегия EXECUTE на эту программу или же системная привилегия EXECUTE ANY PROGRAM.

## Управление базовыми компонентами планировщика

Базовые компоненты Scheduler — задания (jobs), программы, расписания, цепочки и события — имеют несколько общих средств управляемости. Создание, изменение и уничтожение всех компонентов осуществляется одной и той же процедурой из пакета DBMS\_SCHEDULER. В следующих разделах будет описано управление этими компонентами.

### *Управление заданиями*

В основе средства Scheduler лежит создание и управление заданиями. Можно создавать и запускать задания независимо или же создавать задания, применяя расписания и программы. Использование готовых программ и расписаний избавляет от необходимости переопределения программы или расписания при каждом создании нового задания.

### *Создание заданий*

Задание Scheduler создается с помощью процедуры CREATE\_JOB из пакета DBMS\_SCHEDULER. В листинге 1 ниже показан пример создания базового задания Scheduler без применения программы или расписания. Это наиболее простой способ спецификации задания, при котором вся необходимая информация указывается в самом операторе создания задания (jobs), без использования готовых программ или расписаний.

SQL> BEGIN

2 DBMS\_SCHEDULER.CREATE\_JOB(

3 JOB\_NAME => 'test\_job',

4 JOB\_TYPE => 'PLSQL\_BLOCK',

5 JOB\_ACTION => 'insert into persons select \* from new\_persons;',

6 START\_DATE => '28-JUNE-08 07.00.00 PM ',

7 REPEAT\_INTERVAL => 'FREQ=DAILY; INTERVAL=2',

8 END\_DATE => '20-NOV-08 07.00.00 PM ',

9 COMMENTS => 'Insert new customers into the persons table',

10 ENABLED => TRUE,

11\* END;

/

PL/SQL procedure successfully completed.

SQL>

**На заметку!** Если вы создаете задание в собственной схеме, то становитесь его владельцем. Однако если задание создается в другой схеме, владельцем задания будет владелец этой другой схемы. Таким образом, тот факт, что именно вы создали задание, еще не означает, что вы обязательно будете его владельцем.

Давайте рассмотрим параметры процедуры CREATE\_JOB.

* JOB\_NAME — дает возможность указать имя задания.
* JOB\_TYPE — указывает тип задания. Задания могут включать блок PL/SQL, хранимую процедуру, исполняемую программу или программу Java.
* JOB\_ACTION — специфицирует конкретную процедуру, команду или сценарий, который будет запускать ваше задание.
* START\_DATE и END\_DATE — специфицируют дату запуска и завершения нового задания. (Если задание находится в процессе выполнения, оно может и не иметь параметра END\_DATE.)
* REPEAT\_INTERVAL — специфицирует, насколько часто Scheduler будет запускать задание. В листинге 1 выше интервал повторения указан 'FREQ=DAILY; INTERVAL=2'; это означает, что вы запускаете задание каждый второй день. Существуют два способа задания интервала повторения (оба обсуждаются в следующем разделе):
* использование календарного выражения базы данных;
* использование выражения даты/времени PL/SQL.
* COMMENTS — позволяет включить любые комментарии о запланированном задании.
* ENABLED — специфицирует, включено ли задание при его создании. Значением по умолчанию является FALSE, т.е. задание отключено; чтобы включить задание немедленно, установите ENABLED в TRUE.

#### **Установка интервала повторения**

Давайте рассмотрим два способа указания интервала повторения. Календарное выражение — простой способ, подобный выражению на английском языке; он состоит из трех компонентов, которые описаны ниже.

* Частота. Это обязательный компонент календарного выражения, который идентифицируется ключевым словом FREQ. Возможные значения — YEARLY, MONTHLY, DAILY, HOURLY, MINUTELY и SECONDLY.
* Интервал повторения. Этот интервал идентифицируется ключевым словом INTERVAL и указывает, насколько часто база данных должна повторять задание.
* Спецификаторы. Предоставляют детальную информацию о том, когда должно запускаться задание. Возможные значения — BYMONTH, BYWEEKNO, BYYEARDAY, BYMONTHDAY, BYDAY, BYHOUR, BYMINUTE и BYSECOND. Например, BYMONTHDAY указывает день месяца, когда должно быть запущено задание, а BYDAY — день недели.

Обратите внимание, что спецификаторы не обязательны, а компоненты интервала и частоты календарного выражения — обязательны. Вот некоторые типичные календарные выражения:

* FREQ=DAILY; INTERVAL=3 — выполняет задание каждые три дня;
* FREQ=HOURLY; INTERVAL=2 — выполняет задание каждый второй час;
* FREQ=WEEKLY; BYDAY=MON — выполняет задание каждый понедельник;
* FREQ=WEEKLY; INTERVAL=2; BYDAY=FRI — выполняет задание каждую вторую пятницу;
* FREQ=MONTHLY; BYMONTHDAY=1 — выполняет задание в последний день месяца.

Также можно задавать более сложные интервалы повторения, используя выражения PL/SQL, с условием, что все такие выражения должны в результате вычисления возвращать выражение типа даты или временной метки. Когда для задания интервала повторения применяется выражение даты/времени, в качестве значения интервала получается значение типа даты/времени:

repeat\_interval => 'FREQ=MINUTELY INTERVAL=30'

Приведенное выражение PL/SQL устанавливает, что Oracle будет выполнять задание каждые полчаса.

### Задания администрирования

Пакет DBMS\_SCHEDULER служит выполнения административных заданий. Включить и тем самым активизировать задание можно следующим образом:

SQL> EXEC DBMS\_SCHEDULER.ENABLE('TEST\_JOB1');

PL/SQL procedure successfully completed.

Отключается задание так:

SQL> EXEC DBMS\_SCHEDULER.DISABLE('TEST\_JOB1');

PL/SQL procedure successfully completed.

Уничтожить задание можно процедурой DROP\_JOB, как показано ниже:

SQL> BEGIN

DBMS\_SCHEDULER.DROP\_JOB( JOB\_NAME => 'TEST\_JOB1');

END;

Задание можно запустить вручную (в неурочное время), используя процедуру RUN\_JOB:

SQL> EXEC DBMS\_SCHEDULER.RUN\_JOB('TEST\_JOB');

И, наконец, с помощью процедуры STOP\_JOB можно немедленно остановить задание:

SQL> EXEC DBMS\_SCHEDULER.STOP\_JOB('TEST\_JOB');

**Совет.** В обеих процедурах — STOP\_JOB и RUN\_JOB — можно использовать атрибут FORCE, который указывает на то, может ли быть остановлено или удалено выполняющееся задание. Установив FORCE в TRUE, можно остановить или удалить работающее задание (jobs). Значением FORCE по умолчанию является FALSE.

## Управление легковесными заданиями

Когда нужно использовать Scheduler для частого запуска быстро выполняющегося задания, вместо заданий базы данных по умолчанию можно применять легковесные задания, что даст выигрыш в производительности. Легковесные задания не являются отдельными заданиями. Поскольку легковесные задания на самом деле не являются объектами схемы, с ними связаны намного меньшие накладные расходы. К тому же легковесные задания создаются быстрее, чем обычные; легковесные задания требуют меньше места для хранения метаданных и данных времени выполнения. Таким образом, за счет использования легковесных заданий получается выигрыш во времени и пространстве, если их нужно запускать в базе тысячи раз. Обычные задания обеспечивают большую гибкость и предполагают больше вариантов выполнения, и потому, если задание должно выполняться нечасто, то вместо легковесных заданий имеет смысл использовать обычные.

При создании легковесного задания необходимо использовать шаблон задания, причем такой шаблон должен содержать метаданные легковесного задания наряду с привилегиями, которые должны быть унаследованы этим легковесным заданием (jobs). В качестве шаблона задания можно применять хранимую процедуру или программу Scheduler. Чтобы специфицировать действие задания, нужно ссылаться на программу Scheduler. Типом программы должен быть PLSQL\_BLOCK или STORED\_PROCEDURE. Если у пользователя есть привилегии на программу, он автоматически получает привилегии на легковесное задание.

Для просмотра подробной информации о легковесном задании в базе данных применяется запрос, подобный показанному ниже:

SQL> SELECT job\_name, program\_name FROM dba\_scheduler\_jobs

WHERE job\_style='LIGHTWEIGHT';

JOB\_NAME PROGRAM\_NAME

----------- -------------

TEST\_JOB1 TEST\_PROG1

В отличие от обычных заданий базы данных, легковесные задания не отображаются в представлении DBA\_SCHEDULER\_JOBS, поскольку легковесные задания не являются объектами схемы как обычные задания (jobs).

Легковесное задание создается подобно обычному заданию — с помощью процедуры CREATE\_JOB. Просто укажите в параметре JOB\_STYLE значение LIGHTWEIGHT вместо REGULAR (которое является значением по умолчанию). Ниже приведен пример создания легковесного задания:

begin

dbms\_scheduler.create\_job (

job\_name => 'test\_ltwtjob1',

program\_name => 'test\_prog',

repeat\_interval => 'freq=daily,by\_hour=10',

end\_time => '31-DEC-08 06:00:00 AM Australia/Sydney',

job\_style => 'lightweight',

comments => 'A lightweight job based on a program');

end;

В данном примере программа test\_prog служит шаблоном для легковесного задания TEST\_LTWTJOB1. Вместо атрибутов REPEAT\_INTERVAL и END\_TIME можно также указать расписание.

Для создания набора легковесных заданий Scheduler можно использовать массив заданий. Массив заданий удобен, когда требуется создать большое количество заданий Scheduler. Следующий пример демонстрирует создание набора легковесных заданий посредством массива заданий.

1. Создайте две переменные — одну для определения задания Scheduler (jobs), а вторую для определения массива.

declare

testjob sys.job;

testjobarr sys.job\_array;

2. Воспользуйтесь конструктором sys.job\_array для инициализации массива заданий:

begin

testjobarr := sys.job\_array();

При инициализации массива заданий testobjarr, представляющего собой массив объектов типа JOB, база данных создает слот одиночного задания в этом массиве.

3. Установите размер массива заданий равным количеству заданий, которые собираетесь создать.

testjobarr.extend(500);

Показанный здесь оператор выделяет место в массиве заданий для хранения информации о 500 заданиях.

4. Следующий код создает 500 заданий и помещает их в массив заданий (jobs):

for I in 1 . . . 500 loop

testjob := sys.job(job\_name => 'TESTJOB'||TO\_CHAR(I),

job\_style => 'LIGHTWEIGHT',

job\_template => 'TEST\_PROG',

enabled => TRUE);

testjobarr(i) := TESTJOB;

end loop;

Приведенный здесь код создает 500 заданий с использованием шаблона TEST\_PROG. Эти задания добавляются в массив заданий операцией присваивания testjobarr(i).

5. Воспользуйтесь процедурой CREATE\_JOBS для фиксации массива из 500 заданий:

dbms\_scheduler.create\_jobs (testjobarr, 'transactional');

Процедура CREATE\_JOBS создает все 500 заданий за раз. В данном примере было решено создать легковесные задания как часть массива, специфицируя LIGHTWEIGHT в качестве значения параметра JOB\_STYLE при создании массива заданий. Если не указать параметр JOB\_STYLE, можно создать массив обычных заданий вместо легковесных, поскольку значением JOB\_STYLE по умолчанию является REGULAR.

## Управление внешними заданиями

Внешние задания — это исполняемые программы операционной системы, которые запускаются за пределами базы данных. Для внешнего задания в качестве значения параметра JOB\_TYPE указывается EXECUTABLE. При использовании для внешнего задания именованной программы в атрибуте JOB\_ACTION или PROGRAM\_ACTION потребуется указать полный путь к каталогу, где хранится эта исполняемая программа, например /usr/local/bin/perl.

Можно создавать локальные внешние задания и удаленные внешние задания. Локальное внешнее задание запускается на том же сервере, на котором находится база данных, а удаленное внешнее задание — на удаленном хосте. Удаленное внешнее задание применяется для управления заданиями по всей сети из единой базы данных. Что интересно — при этом вам даже не обязательно иметь запущенные экземпляры Oracle на удаленных хостах. Потребуется лишь инсталлировать Scheduler Agent на каждом из удаленных хостов, где нужно запускать внешние задания (jobs), чтобы он принимал запросы на задания от исходной базы данных, выполнял их на удаленном хосте и передавал результаты в исходную базу данных.

Запуск локальных внешних заданий прост. Все, что нужно — это указать значение EXECUTABLE для аргумента JOB\_TYPE или PROGRAM\_TYPE. Чтобы запускать удаленные внешние задания, необходимо инсталлировать и сконфигурировать агент планировщика (Scheduler Agent), а также назначить мандат для выполнения удаленных заданий. В последующих разделах описаны все шаги по настройке удаленного внешнего задания.

### Настройка базы данных

Настройка базы данных, которая будет издавать запросы на выполнение удаленных внешних заданий, осуществляется следующим образом.

1. Поскольку для запуска удаленного внешнего задания понадобится Oracle XML DB, сначала проверьте, успешно ли инсталлирована Oracle XML DB, запустив следующую команду DESCRIBE:

SQL> desc resource\_view

Name Null? Type

----------------- ----- ----------------------------

RES

XMLTYPE (XMLSchema "http://xm

lns.oracle.com/xdb/XDBResour

ce.xsd" Element "Resource")

ANY\_PATH VARCHAR2(4000)

RESID RAW(16)

SQL>

Команда DESCRIBE показывает, что Oracle XML DB корректно инсталлирована. Если отображается иное, прежде чем продолжить работу, нужно будет инсталлировать Oracle XML DB.

2. Выполните предоставленный Oracle сценарий prvtsch.plb, расположенный в каталоге $ORACLE\_HOME/rdbms/admin:

SQL> connect sys/sammyy1 as sysdba

SQL> @$ORACLE\_HOME/rdbms/admin/prvtrsch.plb

PL/SQL procedure successfully completed.

. . .

PL/SQL procedure successfully completed.

no rows selected

Package created.

Package body created.

No errors.

. . .

User altered.

SQL>

3. И, наконец, установите регистрационный пароль для Scheduler Agent:

SQL> EXEC dbms\_scheduler.set\_agent\_registration\_pass(

registration\_password => 'sammyy1'.-

expiration\_date => systimestamp + interval '7' day,-

max\_uses => 25)

PL/SQL procedure successfully completed.

SQL>

Scheduler Agent использует пароль для регистрации в базе данных. Параметры EXPIRATION\_DATE и MAX\_USES показывают, соответственно, срок действия пароля (дату, до которой он действителен) и сколько он может быть использован.

## Инсталляция и конфигурирование Scheduler Agent

Scheduler Agent потребуется инсталлировать на каждый удаленный хост, где планируется запускать внешние задания. Программное обеспечение для инсталляции можно либо загрузить с сайта Oracle, либо воспользоваться комплектом Database CD. В любом случае понадобится инсталляционный носитель для Oracle Database Gateway. Ниже перечислены шаги, которые необходимо выполнить для инсталляции Scheduler Agent.

1. Войдите в систему как владелец программного обеспечения Oracle (обычно пользователь oracle).
2. Перейдите в место хранения инсталляционных файлов Oracle Database Gateway и введите следующую команду для запуска универсального инсталлятора Oracle:

$ /oracle11g/gateways/runInstaller

1. На экране приглашения щелкните на кнопке Next (Далее).
2. На странице Select a Product (Выберите продукт) выберите Oracle Scheduler Agent 11.10.0.6.0 и щелкните на кнопке Next.
3. На странице Specify Home Details (Укажите детальную информацию о домашнем каталоге) выберите имя и укажите путь к домашнему каталогу Oracle Scheduler Agent. Щелкните на кнопке Next.
4. На странице Oracle Scheduler Agent (Агент планировщика Oracle) укажите имя хоста и номер порта, который должен использовать агент для взаимодействия с базой данных, выдавшей запрос на внешнее задание (jobs). Щелкните на кнопке Next.
5. На странице Summary (Сводка) проверьте установки и щелкните на кнопке Install (Инсталлировать).

**Внимание!** Для автоматизации процесса установки Scheduler Agent на большое количество хостов можно использовать молчаливую инсталляцию.

1. Когда инсталлятор предложит запустить сценарий root.sh от имени пользователя root, сделайте это и щелкните на кнопке OK.
2. Щелкните на кнопке Exit (Выход) на странице End of Installation (Конец инсталляции).

Для запуска Scheduler Agent служит исполняемый файл schagent. Однако сначала потребуется зарегистрировать агент в базе данных, откуда хотите запускать внешние задания на хосте, где был установлен Scheduler Agent. Вот как регистрируется Scheduler Agent в базе данных:

$ schagent –registerdatabase prod1 1522

В этом примере хост базы данных называется prod1, а номером порта, назначенного Scheduler Agent, является 1522. После выдачи этой команды появится приглашение указать пароль регистрации агента, созданного ранее:

$./schagent -registerdatabase localhost.localdomain 1522

Agent Registration Password ? \*\*\*\*\*\*

$

Запускается Scheduler Agent с помощью следующей команды:

$./schagent –start

Scheduler agent started

$

Остановить агент можно такой командой:

$./schagent –stop

Scheduler agent stopped

$

Приведенный выше пример демонстрирует работу с Scheduler Agent в системе UNIX/Linux. Прежде чем можно будет пользоваться агентом, потребуется установить службу OracleSchedulerExecutionAgent. Эта служба инсталлируется следующим образом:

$ schagent –installagentservice

Служба OracleSchedulerExecutionAgent отличается от службы Oracle, которая используется для запуска и останова экземпляра Oracle на сервере Windows.

### Создание и включение удаленных внешних заданий

Поскольку внешнее задание должно выполняться как задание пользователя операционной системы, Scheduler позволяет назначать мандаты операционной системы внешнему заданию. Используется мандат, который является объектом схемы, содержащим комбинацию имени и пароля пользователя, от имени которого будет выполняться это задание.

Для локальных внешних заданий указывать мандаты не обязательно, хотя в Oracle и рекомендуют это делать. Прежде чем можно будет создать удаленное внешнее задание, потребуется создать мандат. Объект мандата можно назначить пользователю, от имени которого будет запускаться внешняя исполняемая программа. Обратите внимание, что пользователь должен обладать привилегиями на выполнение объекта мандата, чтобы иметь возможность применять его для запуска задания (jobs).

Ниже перечислены шаги, которые необходимо выполнить для создания удаленного внешнего задания.

1. Сначала выполните процедуру CREATE\_CREDENTIAL для создания объекта мандата:

SQL> exec dbms\_scheduler.create\_credential('hrcredential, 'hr','sammyy1');

2. Выдайте привилегии на вновь созданный мандат пользователю, которому нужно будет его использовать:

SQL> grant execute on system.hrcrdential to sam;

Для просмотра всех мандатов в базе данных можно запросить представление DBA\_SCHEDULER\_VIEW.

3. Создайте удаленное внешнее задание, выполнив процедуру CREATE\_JOB:

SQL> begin

2 dbms\_scheduler.create\_job(

3 job\_name => 'remove\_logs',

4 job\_type => 'executable',

5 job\_action => '/u01/app/oracle/logs/removelogs',

6 repeat\_interval => 'freq=daily; byhour=23',

7 enabled => false);

8\* end;

SQL> /

PL/SQL procedure successfully completed.

SQL>

4. Создав удаленное внешнее задание REMOVE\_LOGS, с помощью процедуры SET\_ATTRIBUTE установите атрибут CREDENTIAL\_NAME этого удаленного задания:

SQL> exec dbms\_scheduler.set\_attribute('remove\_logs',

'credential\_name','hrcredential');

PL/SQL procedure successfully completed.

SQL>

5. Снова выполните процедуру SET\_ATTRIBUTE, на этот раз, чтобы установить атрибут DESTINATION:

SQL> exec dbms\_scheduler.set\_attribute('remove\_logs',

'destination', 'localhost.localdomain:1521');

PL/SQL procedure successfully completed.

SQL>

6. Выполните процедуру ENABLE для включения внешнего задания:

SQL> exec dbms\_scheduler.enable('remove\_logs');

PL/SQL procedure successfully completed.

SQL>

Удалив пользователя remote\_scheduler\_agent, создавшего сценарий prvtsch.plb, который вы запускали ранее, можно отключить возможность запуска внешних заданий в базе данных.

SQL> drop user remote\_scheduler\_agent cascade;

Затем, чтобы база данных снова могла запускать удаленные внешние задания (jobs), придется заново запустить сценарий prvtsch.plb.

### Управление программами

Программа содержит метаданные о том, что именно будет запускать Scheduler, включая имя и тип программы, и о том, какое задание будет выполняться. Разные задания могут разделять одну программу.

#### **Создание программы**

Новая программа создается с использованием процедуры CREATE\_PROGRAM из пакета DBMS\_SCHEDULER, как показано ниже:

SQL> BEGIN

2 DBMS\_SCHEDULER.CREATE\_PROGRAM(

3 PROGRAM\_NAME => 'MY\_PROGRAM',

4 PROGRAM\_ACTION => 'UPDATE\_SCHEMA\_STATS',

5 PROGRAM\_TYPE => 'STORED\_PROCEDURE',

6 enabled => TRUE);

7\* end;

SQL> /

PL/SQL procedure successfully completed.

SQL>

После создания программы можно упростить оператор создания задания, заменив атрибуты JOB\_TYPE и JOB\_ACTION именем программы, которая уже содержит спецификацию этих атрибутов. Атрибуты PROGRAM\_TYPE и PROGRAM\_ACTION заменяют атрибуты задания, которые обычно указываются при создании нового задания. Такого модульный подход дает преимущество — разные задания могут использовать одну и ту же программу, чем упрощается создание новых заданий.

В следующем примере пересоздается задание TEST\_JOB, которое было создано в листинге 1, но на этот раз с использованием компонента-программы:

SQL> BEGIN

2 DBMS\_SCHEDULER.CREATE\_JOB(

3 JOB\_NAME => 'TEST\_JOB',

4 PROGRAM\_NAME => 'TEST\_PROGRAM',

5 REPEAT\_INTERVALl => 'FREQ=DAILY;BYHOUR=12',

6 ENABLED => TRUE);

7\* END;

SQL> /

PL/SQL procedure successfully completed.

SQL>

В приведенном примере использование программы позволяет избежать необходимости указывать параметры JOB\_TYPE и JOB\_ACTION в операторе CREATE\_JOB.

#### **Администрирование программ**

Вы можете включать, отключать и удалять программы Scheduler с помощью различных процедур из пакета DBMS\_SCHEDULER, как показано в последующих примерах.

Процедура ENABLE используется для включения программы Scheduler:

SQL> EXEC DBMS\_SCHEDULER.ENABLE('TEST\_PROGRAM');

PL/SQL procedure successfully completed.

Соответственно, для отключения программы применяется процедура DISABLE:

SQL> EXEC DBMS\_SCHEDULER.DISABLE('TEST\_PROGRAM');

PL/SQL procedure successfully completed.

SQL>

Процедура DROP\_PROGRAM служит для удаления программы:

SQL> EXEC DBMS\_SCHEDULER.DROP\_PROGRAM('TEST\_PROGRAM');

PL/SQL procedure successfully completed.

SQL>

### Управление расписаниями

Предположим, что имеется множество заданий (jobs), выполняемых в одно и то же время. Используя общее расписание, можно упростить создание и управление такими заданиями. В следующем разделе будет объяснено, как управлять расписаниями.

#### **Создание расписания**

Для создания расписания применяется процедура CREATE\_SCHEDULER из пакета DBMS\_SCHEDULER:

SQL> BEGIN

2 DBMS\_SCHEDULER.CREATE\_SCHEDULE(

3 SCHEDULE\_NAME => 'TEST\_SCHEDULE',

4 START\_DATE => SYSTIMESTAMP,

5 END\_DATE => SYSTIMESTAMP + 90,

6 REPEAT\_INTERVAL => 'FREQ=HOURLY;INTERVAL= 4',

7 COMMENTS => 'Every 4 hours');

8\* END;

SQL> /

PL/SQL procedure successfully completed

SQL>

Расписание TEST\_SCHEDULE говорит о том, что задание с этим расписанием будет выполнено немедленно, а затем станет повторяться каждые 4 часа на протяжении 90 дней. Обратите внимание на следующие моменты, касающиеся этого нового расписания.

* Процедура CREATE\_SCHEDULE имеет три важных параметра: START\_DATE, END\_DATE и REPEAT\_INTERVAL.
* Время начала и конца расписания задается с использованием типа данных TIMESTAMP WITH TIME ZONE.
* Для указания интервала повторения необходимо применять календарное выражение.

Создав расписание TEST\_SCHEDULE, можно еще более упростить процесс создания заданий, используя и программу, и расписание, как показано ниже:

SQL> BEGIN

2 DBMS\_SCHEDULER.CREATE\_JOB(

3 JOB\_NAME => 'MY\_JOB',

4 PROGRAM\_NAME => 'MY\_PROGRAM',

5 SCHEDULE\_NAME => 'MY\_SCHEDULE');

6 END;

7 /

PL/SQL procedure successfully completed.

SQL>

Как видите, применение сохраняемых расписаний и программ превращает создание нового задания в пару пустяков.

#### **Администрирование расписаний**

Изменять различные атрибуты расписания можно с помощью процедуры SET\_ATTRIBUTE из пакета DBMS\_SCHEDULER. Модификации доступны любые атрибуты, за исключением имени самого расписания.

Удаляется расписание процедурой DROP\_SCHEDULE, как показано ниже:

SQL> BEGIN

2 DBMS\_SCHEDULER.DROP\_SCHEDULE (SCHEDULE\_NAME => 'TEST\_SCHEDULE');

3 END;

4 /

PL/SQL procedure successfully completed.

SQL>

Если расписание, которое планируется удалить, используется окном или заданием, то попытка удаления такого расписания по умолчанию закончится ошибкой. Тем не менее, можете заставить базу данных все равно удалить это расписание, указав в предыдущем примере дополнительный параметр FORCE со значением TRUE.

**Совет.** При создании расписания Oracle предоставляет к нему PUBLIC-доступ, тем самым по умолчанию разрешая всем пользователям применять это расписание.

### Управление цепочками

Цепочка Scheduler состоит из набора взаимосвязанных программ, которые запускаются в заданной последовательности. Относительные позиции в цепочке называются “шагами”, причем каждый шаг указывает на другую цепочку, программу или событие. Цепочка включает “правила”, определяющие то, что должно быть сделано на каждом шаге цепочки.

Давайте создадим простую цепочку Scheduler, начиная с создания объекта цепочки, после чего добавим к ней шаги и правила.

#### **Создание цепочки**

Поскольку цепочки Scheduler используют объекты Oracle Streams Rules Engine, для создания цепочки пользователь должен иметь как привилегии CREATE JOB, так и привилегии Rules Engine. Выдать все необходимые привилегии Rules Engine можно с помощью приведенного ниже оператора, который выдает привилегии пользователю nina:

SQL> BEGIN

DBMS\_RULE\_ADM.GRANT\_SYSTEM\_PRIVILEGE(DBMS\_RULE\_ADM.CREATE\_RULE\_OBJ, 'nina'),

DBMS\_RULE\_ADM.GRANT\_SYSTEM\_PRIVILEGE (

DBMS\_RULE\_ADM.CREATE\_RULE\_SET\_OBJ, 'nina'),

DBMS\_RULE\_ADM.GRANT\_SYSTEM\_PRIVILEGE (

DBMS\_RULE\_ADM.CREATE\_EVALUATION\_CONTEXT\_OBJ, 'nina')

END;

Теперь, имея необходимые привилегии, создадим цепочку Scheduler по имени TEST\_CHAIN с использованием процедуры CREATE\_CHAIN:

SQL> BEGIN

DBMS\_SCHEDULER.CREATE\_CHAIN (

chain\_name => 'test\_chain',

rule\_set\_name => NULL,

evaluation\_interval => NULL,

comments => NULL);

END;

Затем определим шаги новой цепочки, используя процедуру DEFINE\_CHAIN\_STEP. Обратите внимание, что шаг цепочки может указывать на программу, событие или другую цепочку:

SQL> BEGIN

DBMS\_SCHEDULER.DEFINE\_CHAIN\_STEP('test\_chain', 'step1', 'program1');

DBMS\_SCHEDULER.DEFINE\_CHAIN\_STEP('test\_chain', 'step2', 'program2');

DBMS\_SCHEDULER.DEFINE\_CHAIN\_STEP('test\_chain', 'step3', 'program3');

END;

И, наконец, чтобы запустить цепочку в действие, с помощью процедуры DEFINE\_CHAIN\_RULE к ней нужно добавить правила. Правила цепочки определяют, когда цепочка запускается, и специфицируют условия, при которых запускается каждый шаг. Обычно правило, на котором базируется шаг цепочки, состоит в выполнении определенных условий. Вот пример:

SQL> BEGIN

DBMS\_SCHEDULER.DEFINE\_CHAIN\_RULE('test\_chain', 'TRUE', 'START step1');

DBMS\_SCHEDULER.DEFINE\_CHAIN\_RULE('test\_chain', 'step1 COMPLETED',

'Start step2, step3');

DBMS\_SCHEDULER.DEFINE\_CHAIN\_RULE('test\_chain',

'step2 COMPLETED AND step3 COMPLETED', 'END');

END;

Первое правило в приведенном примере указывает, что должен выполниться шаг step1, а это означает, что Scheduler запустит program1. Второе правило говорит о том, что шаги step2 (program2) и step3 (program3) будут запущены только при условии успешного завершения шага step1 ('step1 COMPLETED'). Последнее правило говорит о том, что по завершении step2 и step3 цепочка закончится.

#### **Включение цепочки**

Прежде чем использовать цепочку, ее необходимо включить. Вот как это делается:

SQL> BEGIN

DBMS\_SCHEDULER.ENABLE ('test\_chain');

END;

#### **Встраивание заданий в цепочку**

Для того чтобы запустить задание внутри цепочки Scheduler, нужно создать задание с атрибутом JOB\_TYPE, установленным в CHAIN, и атрибутом JOB\_ACTION, указывающим на имя определенной цепочки, которую необходимо использовать. Разумеется, это означает, что сначала следует создать цепочку.

Вот как выглядит синтаксис создания задания для цепочки Scheduler:

SQL> BEGIN

DBMS\_SCHEDULER.CREATE\_JOB (

JOB\_NAME => 'test\_chain\_job',

JOB\_TYPE => 'CHAIN',

JOB\_ACTION => 'test\_chain',

REPEAT\_INTERVAL => 'freq=daily;byhour=13;byminute=0;bysecond=0',

ENABLED => TRUE);

END;

С использованием процедуры RUN\_CHAIN можно запустить цепочку без предварительного создания задания. Эта процедура создаст временное задание и немедленно запустит цепочку. Ниже показано, как это делается:

SQL> BEGIN

DBMS\_SCHEDULER.RUN\_CHAIN (

CHAIN\_NAME => 'my\_chain1',

JOB\_NAME => 'quick\_chain\_job',

START\_STEPS => 'my\_step1, my\_step2');

END;

Как и с другими компонентами Scheduler, здесь существуют процедуры, позволяющие удалить цепочку, удалить правила из цепочки, отключить цепочку, изменить ее и т.д. За всеми подробностями обращайтесь в раздел руководства Oracle PL/SQL Packages and Types Reference (Справочник по пакетам и типам PL/SQL), посвященный DBMS\_SCHEDULER.

### Управление событиями

До сих пор было показано, как создаются задания с расписанием и без. При создании задания без расписания должно быть указано время запуска и частота повторений, в то время как применение расписаний позволяет исключить эти сведения из спецификации задания. В обоих случаях время задания базируется на календарном времени. Однако можно создавать также задания и расписания, основанные только на событиях, а не на календарном времени. В следующих разделах мы кратко рассмотрим задания и расписания на основе событий.

#### **Создание задания на базе события**

В следующем примере показано, как создается задание Scheduler с использованием программы и события. Задание будет запущено при возникновении события FILE\_ARRIVAL:

SQL> BEGIN

dbms\_scheduler.create\_job(

JOB\_NAME => test\_job,

PROGRAM\_NAME => test\_program,

START\_DATE => '01-AUG-08 5.00.00AM US/Pacific',

EVENT\_CONDITION => 'tab.user\_data.event\_name = ''FILE\_ARRIVAL''',

QUEUE\_SPEC => 'test\_events\_q'

ENABLED => TRUE,

COMMENTS => 'An event based job');

END;

В приведенной процедуре CREATE\_JOB есть два незнакомых параметра; оба они уникальны для заданий на основе событий.

* EVENT\_CONDITION. Атрибут EVENT\_CONDITION представляет собой условное выражение, которое берет свои значения из таблицы очереди источника событий и использует правила Oracle Streams Advanced Queuing. В этом выражении указываются атрибуты объектов, и они снабжаются префиксом tab.user\_data. Загляните в описание пакета DBMS\_AQADM, чтобы узнать больше о Oracle Streams Advanced Queuing и связанных с этим правилах.
* QUEUE\_SPEC. Атрибут QUEUE\_SPEC определяет очередь, в которую будет помещено событие, инициирующее задание. В приведенном примере test\_events\_q — имя очереди.

#### **Создание расписаний на базе событий**

В следующем примере показано, как создается расписание на основе события. Всякий раз, когда возникает событие (FILE\_ARRIVAL), Scheduler запускает задание, основанное на расписании, которое создано в этом примере. В данном случае событие указывает на поступление файла до полудня.

SQL> BEGIN

dbms\_scheduler.create\_event\_schedule(

SCHEDULE\_NAME => 'appowner.file\_arrival',

START\_DATE => systimestamp,

EVENT\_CONDITION => 'tab.user\_data.object\_owner = ''APPOWNER''

AND tab.user\_data.event\_name = ''FILE\_ARRIVAL''

AND extract hour FROM tab.user\_data.event\_timestamp < 12',

QUEUE\_SPEC => 'test\_events\_q');

END;

В предыдущем примере вы уже видели применение EVENT\_CONDITION и QUEUE\_SPEC.

## Управление расширенными компонентами Scheduler

До сих пор вы изучали управление базовыми компонентами Scheduler — заданиями, программами, расписаниями, цепочками и событиями. В этом же разделе мы рассмотрим управление расширенными компонентами Oracle Scheduler — классами заданий и окнами (а также группами окон).

Вы уже видели, как Scheduler использует средства Database Resource Manager, такие как группы потребителей ресурсов и планы ресурсов, чтобы эффективно распределять дефицитные ресурсы операционной системы и базы данных. Очень часто случается так, что сложные пакетные задания выходят за пределы выделенного им окна и распространяются на рабочее время, когда транзакции OLTP требуют львиной доли ресурсов. Назначение приоритетов заданиям для обеспечения адекватного использования ресурсов — важнейшее требование в рабочих базах данных. Scheduler для назначения приоритетов использует концепции классов заданий и окон.

### Управление классами заданий

С помощью классов заданий заданиям устанавливаются приоритеты, что позволяет дифференцированно выделять ресурсы различным группам заданий. Планировщик ассоциирует каждый класс заданий с группой потребителей ресурсов, что позволяет ему определять соответствующее выделение ресурсов для каждого из таких классов. Возможность ассоциировать классы заданий с группами потребителей ресурсов, созданных Database Resource Manager, помогает в назначении приоритетов заданиям.

**На заметку!** Все задания должны относиться к определенному классу. Существует класс заданий по умолчанию — DEFAULT\_JOB\_CLASS, — к которому по умолчанию относятся все задания, если они явно не отнесены к какому-то другому классу. Класс заданий будет ассоциирован по умолчанию с DEFAULT\_CONSUMER\_GROUP, если вы не назначите его явно в другую группу потребителей ресурсов.

#### **Создание класса заданий**

Все классы заданий создаются в схеме SYS, независимо от того, кто из пользователей создает их. В следующем примере используется процедура CREATE\_JOB\_CLASS для создания нового класса заданий по имени ADMIN\_JOBS:

SQL> BEGIN

DBMS\_SCHEDULER.CREATE\_JOB\_CLASS(

JOB\_CLASS\_NAME => 'admin\_jobs'

RESOURCE\_CONSUMER\_GROUP => 'admin\_group',

LOGGING\_LEVEL => dbms\_scheduler.logging\_runs

LOG\_HISTORY => 15);

END;

В приведенном примере используются перечисленные ниже атрибуты.

JOB\_CLASS\_NAME. Имя класса заданий.

RESOURCE\_CONSUMER\_GROUP. Этот атрибут указывает, что все задания, являющиеся членами этого класса, будут назначены в группу потребителей ресурсов ADMIN\_GROUP.

LOGGING\_LEVEL. Этот атрибут может принимать следующие три значения.

DBMS\_SCHEDULER.LOGGING\_OFF. Специфицирует отсутствие какого-либо ведения журналов для заданий этого класса.

DBMS\_SCHEDULER.LOGGING\_RUNS. Специфицирует детальные записи в журнале для каждого запуска задания.

DBMS\_SCHEDULER.LOGGING\_FULL. Специфицирует детальные записи о каждом запуске задания из данного класса, как и для всех прочих операций в задании, включая создание, удаление, изменение, включение и отключение заданий.

**На заметку!** Значение DBMS\_SCHEDULER.LOGGING\_FULL для атрибута LOGGING\_LEVEL предоставляет большую часть информации о заданиях класса; уровень ведения журналов по умолчанию — DBMS\_SCHEDULER.LOGGING\_RUNS.

* LOG\_HISTORY. Этот атрибут указывает количество дней, в течение которых база данных будет сохранять журналы, прежде чем очистить их автоматически запланированным заданием PURGE\_LOG. Очищать журналы можно также вручную, применяя процедуру PURGE\_LOG пакета DBMS\_SCHEDULER.

Процедура PURGE\_LOG пакета DBMS\_SCHEDULER принимает два важных параметра — LOG\_HISTORY и WHICH\_LOG. Параметр LOG\_HISTORY служит для спецификации количества дней, в течение которых будут храниться журналы перед тем, как Scheduler очистит их. Параметр WHICH\_LOG позволяет указать, хотите ли вы очищать задание. Например, чтобы очистить журналы всех заданий старше 14 дней, следует применить такой оператор:

SQL> EXEC DBMS\_SCHEDULER.PURGE\_LOG(LOG\_HISTORY=14, WHICH\_LOG='JOB\_LOG');

#### **Удаление класса заданий**

Класс заданий удаляется с помощью процедуры DROP\_JOB\_CLASS, как показано ниже:

SQL> BEGIN

DBMS\_SCHEDULER.DROP\_JOB\_CLASS('TEST\_CLASS');

END;

**Совет.** Для удаления классов заданий вместе с принадлежащими к ним заданиями потребуется специфицировать опцию FORCE=TRUE. Если задание в данный момент выполняется, ему будет позволено завершиться, прежде чем будет отключен удаляемый класс заданий.

#### **Изменение атрибутов класса заданий**

Изменять атрибуты класса заданий можно процедурой ALTER\_ATTRIBUTES. В следующем примере будет изменено значение атрибута START\_DATE, с указанием его нового значения в параметре VALUE:

SQL> BEGIN

2 DBMS\_SCHEDULER.ALTER\_ATTRIBUTES(

3 NAME => 'ADMIN\_JOBS',

4 ATTRIBUTE => 'START\_DATE',

5 VALUE => '01-JUL-2008 9:00:00 PM US/Pacific');

6\* END;

SQL>

### Изменение планов ресурсов, использующих окна

Окном называется интервал с определенным временем начала и конца, такой как “с полуночи до 6 утра”. Однако окно — это не просто хронологическая характеристика вроде расписания, которое указывает момент, когда будет запущено задание. Каждое окно ассоциировано с планом ресурсов. При создании окна в качестве параметра специфицируется план ресурсов. Возможность активизировать различные планы ресурсов в разное время — это то, что делает окна специальными объектами планирования, которые позволяет назначать приоритеты.

Основное назначение окна состоит в том, чтобы переключать активный план ресурсов в определенных временных рамках. Все задания, которые запускаются во время окна, будут управляться планом ресурсов, действующим для данного окна. Без окон пришлось бы вручную переключаться между разными планами управления ресурсами. Окна позволяет автоматически изменять планы ресурсов на основе расписания.

**На заметку!** Все окна создаются в схеме SYS, независимо от пользователя, создающего их. Для управления окнами необходимо располагать системной привилегией MANAGE SCHEDULER.

Окно Scheduler состоит из следующих трех основных атрибутов.

* Время начала, время конца и интервал повторения. Эти атрибуты определяют, когда и насколько часто будет открываться и закрываться окно (т.е. задают время действия окна).
* Длительность. Задает продолжительность времени, в течение которого окно остается открытым.
* План ресурсов. Задает приоритеты ресурсов между классами заданий.

**На заметку!** Представление V$RSRC\_PLAN предоставляет информацию о текущих активных планах ресурсов в базе данных.

На первый взгляд может показаться, что расписания и окна служат одной и той же цели, поскольку то и другое позволяет специфицировать время начала и конца, а также интервал повторения задания. Однако атрибут плана ресурсов отличает окно от простого расписания. Всякий раз, когда окно открывается, с ним ассоциируется определенный активный план ресурсов. Таким образом, определенному заданию будут выделены разные ресурсы, если оно запускается в разных окнах.

Специфицировать, какие ресурсы необходимо выделить различным классам заданий во время определенного периода времени, можно за счет ассоциации плана ресурсов с окном, которое создается на этот период. Когда окно открывается, база данных автоматически переключается к ассоциированному плану ресурсов, который становится активным планом. Общий для системы ресурсный план, ассоциированный с окном, будет управлять выделением ресурсов для всех заданий и сеансов, которые запланированы к запуску в период действия этого окна. Когда окно закрывается, происходит переключение к другому ресурсному плану, если никакое другое окно не открыто в это время.

Посмотреть, какое окно активно в данный момент, и какой план ресурсов ассоциирован с этим окном, можно с помощью следующего запроса:

SQL> SELECT window\_name, resource\_plan, enabled, active

2 FROM DBA\_SCHEDULER\_WINDOWS;

WINDOW\_NAME RESOURCE\_PLAN ENABLED ACTIVE

---------------- -------------------- ------- -------

TEST\_WINDOW TEST\_RESOURCEPLAN TRUE FALSE

. . .

SQL>

Здесь видно, что окно TEST\_WINDOW включено, но в данный момент не активно.

#### **Создание окна**

Окно создается с использованием процедуры CREATE\_WINDOW. Давайте рассмотрим два примера применения этой процедуры: один с встроенной спецификацией времени начала, конца и интервала повторения, и другой — с использованием сохраненного расписания вместо указания этих трех атрибутов планирования.

В первом примере оператор создания окна специфицирует расписание для окна:

SQL> BEGIN

DBMS\_SCHEDULER.CREATE\_WINDOW(

WINDOW\_NAME => 'MY\_WINDOW',

START\_DATE => '01-JUN-08 12:00:00AM',

REPEAT\_INTERVAL => 'FREQ=DAILY',

RESOURCE\_PLAN => 'TEST\_RESOURCEPLAN',

DURATION => interval '60' minute,

END\_DATE => '31-DEC-08 12:00:00AM',

WINDOW\_PRIORITY => 'HIGH',

COMMENTS => 'Test Window');

END;

Рассмотрим индивидуальные атрибуты нового окна, созданного в предыдущем операторе.

* RESOURCE\_PLAN. Этот атрибут специфицирует, что на то время, пока это окно открыто, выделение ресурсов для всех заданий, запускаемых в этом окне, будет управляться директивами плана ресурсов TEST\_RESOURCEPLAN.
* WINDOW\_PRIORITY. Этот атрибут установлен в HIGH, а уровнем приоритета по умолчанию является LOW; это единственная пара допустимых значений. Если два окна перекрываются, то окно с более высоким приоритетом имеет преимущество. В каждый данный момент времени может быть открыто только одно окно, и потому, когда они перекрываются, высокоприоритетное окно открывается, а низкоприоритетное — нет.
* START\_DATE. Установка этого атрибута специфицирует, что окно впервые активизируется в 12:00 дня 1 июня 2008 г. Можно также сказать, что окно открывается в это время.
* DURATION. Этот атрибут говорит о том, что окно останется открытым в течение 60-минутного периода, после чего будет закрыто.
* REPEAT\_INTERVAL. Этот атрибут специфицирует следующее время, когда окно будет вновь открыто. В данном примере это 12:00 2 июня 2008 г.
* END\_DATE. Этот атрибут специфицирует, что данное окно будет открыто последний раз 31 декабря 2008 г., после чего будет отключено и закрыто.

**На заметку!** Поскольку Scheduler не проверяет существование каких-либо окон для заданного расписания, иногда окна могут перекрываться.

В следующем примере окно создается с использованием сохраненного расписания. Очевидно, что создавать окно подобным образом намного проще:

SQL> BEGIN

DBMS\_SCHEDULER.CREATE\_WINDOW(

WINDOW\_NAME => 'TEST\_WINDOW',

SCHEDULE\_NAME => 'TEST\_SCHEDULE',

RESOURCE\_PLAN => 'TEST\_RESOURCEPLAN',

DURATION => interval '180' minute,

COMMENTS => 'Test Window');

END;

В приведенном вызове процедуры CREATE\_WINDOW использование расписания TEST\_SCHEDULE позволяет избежать необходимости в указании параметров START\_DATE, END\_DATE и REPEAT\_INTERVAL.

**На заметку!** Окно автоматически включается при создании.

После создания окно необходимо ассоциировать с заданием или классом заданий, чтобы эти задания могли использовать преимущества переключения активных ресурсных планов.

#### **Управление окнами**

Открывать, закрывать, изменять, включать, выключать или удалять окна можно с помощью соответствующих процедур из пакета DBMS\_SCHEDULER, и для выполнения всех этих заданий понадобится привилегия MANAGE SCHEDULER. Обратите внимание, что поскольку все окна создаются в схеме SYS, при обращении к окну всегда должен использоваться синтаксис [SYS].имя\_окна.

Окно будет автоматически открыто в момент времени, указанный атрибутом START\_TIME. В любой момент окно можно открыть вручную, воспользовавшись для этого процедурой OPEN\_WINDOW. Даже при ручном открытии окна оно также будет открыто в заданное время, определенное интервалом.

Ниже показан пример открытия окна вручную:

SQL> EXECUTE DBMS\_SCHEDULER.OPEN\_WINDOW(

WINDOW\_NAME => 'BACKUP\_WINDOW',

DURATION => '0 12:00:00');

SQL>

Взгляните на атрибут DURATION в предыдущем операторе. Когда вы специфицируете продолжительность, то можете указать дни, часы, минуты и секунды — именно в таком порядке. Таким образом, данная установка означает 0 дней, 12 часов, 0 минут и 0 секунд.

Можно также открывать уже открытое окно. Если сделать это, оно останется открытым в течение периода, заданного атрибутом DURATION. То есть если вы откроете окно, которое уже было открыто в течение 30 минут, и его продолжительность установлена в 60 минут, это окно останется открытым в течение начальных 30 минут плюс дополнительные 60 минут, итого — 90 минут.

Для закрытия окна используется процедура CLOSE\_WINDOW, как иллюстрирует следующий пример:

SQL> EXECUTE DBMS\_SCHEDULER.CLOSE\_WINDOW('BACKUP\_WINDOW');

Если на момент закрытия окна задание еще выполняется, оно будет продолжено до завершения. Однако если вы создадите задание с атрибутом STOP\_ON\_WINDOW\_CLOSE, установленным в TRUE, это работающее задание будет закрыто сразу при закрытии его окна. Для отключения окна служит процедура DISABLE:

SQL> EXECUTE DBMS\_SCHEDULER.DISABLE (NAME => 'BACKUP\_WINDOW');

Вы можете отключить окно только в том случае, если ни одно задание не использует его, или если окно не открыто. Если же окно открыто, его можно отключить с помощью процедуры DISABLE с атрибутом FORCE=TRUE.

Для удаления окна применяется процедура DROP\_WINDOW. Если задание, ассоциированное с окном, выполняется, то процедура DROP\_WINDOW продолжится до завершения, после чего только окно будет отключено. Однако если установить атрибут STOP\_ON\_WINDOW\_CLOSE в TRUE, задание остановится немедленно при удалении ассоциированного окна. При использовании установки FORCE=TRUE отключаются все задания, использующие это окно.

#### **Назначение приоритетов заданиям**

Каждый класс заданий Scheduler можно отобразить на определенную группу потребителей ресурсов. План ресурсов назначается группе потребителей ресурсов и, следовательно — каждому классу заданий, с помощью Database Resource Manager. Активный план ресурсов (определяемый текущим открытым окном) выделит ресурсы группе, с разными уровнями ресурсов для разных заданий, на основе их класса заданий.

Scheduler тесно взаимодействует с Database Resource Manager, обеспечивая правильное выделение ресурсов заданиям. Scheduler запустит задание только при наличии достаточного количества ресурсов для его выполнения.

Внутри каждого окна Scheduler можно иметь несколько работающих заданий, с разными уровнями приоритета. Приоритеты назначаются заданиям на двух уровнях —класса и задания. Назначение приоритетов на уровне класса основано на ресурсах, выделенных каждой группе потребителей ресурсов текущим активным планом распределения ресурсов. Например, класс FINANCE\_JOBS может иметь преимущество перед классом ADMIN\_JOBS в отношении выделяемых ресурсов, продиктованных его активным планом распределения ресурсов. Внутри классов FINANCE\_JOBS и ADMIN\_JOBS присутствуют несколько индивидуальных заданий. С помощью процедуры SET\_ATTRIBUTES можно изменять приоритет любого задания, как показано ниже:

SQL> BEGIN

dbms\_scheduler.SET\_ATTRIBUTE(

NAME => 'test\_job',

ATTRIBUTE => 'job\_priority',

VALUE => 1);

END;

Значение приоритета задания по умолчанию равно 3, что можно проверить следующим запросом:

SQL> SELECT job\_name, job\_priority FROM dba\_scheduler\_jobs;

JOB\_NAME JOB\_PRIORITY

------------------------------ --------------------

ADV\_SQLACCESS1523128 3

ADV\_SQLACCESS5858921 3

GATHER\_STATS\_JOB 3

PURGE\_LOG 3

TEST\_JOB03 3

TEST\_JOB1 3

6 rows selected

SQL>

Когда внутри одного и того же класса присутствует более одного задания, значение JOB\_PRIORITY каждого из них определяет, какое задание запускается первым.

#### **Приоритеты окон**

Поскольку окна могут иметь перекрывающиеся расписания, часто может быть открыто одновременно более одного окна, каждое со своим собственным планом ресурсов. В подобных случаях Scheduler закроет все окна за исключением одного, используя определенные правила старшинства. Ниже описано, как работают эти правила.

* Если два окна перекрываются, то окно с более высоким приоритетом открывается, а с более низким — закрывается.
* Если перекрываются окна с одинаковым приоритетом, открытым остается активное окно.
* Если вы находитесь в конце окна, и есть другие окна на тот же период с тем же приоритетом, то окно, имеющее больший процент времени, останется открытым.

#### **Группы окон**

Группой окон называется коллекция окон, являющаяся частью схемы SYS. Группы окон — необязательные сущности, и какое-нибудь окно можно сделать частью группы окон при ее создании, либо же добавить окна в группу позднее. В качестве расписания для задания можно специфицировать единственное окно или группу окон.

Как уже объяснялось ранее в этой главе, можно взять два или более окон с одинаковыми характеристиками, например, некоторые ночные окна и некоторые окна выходных, и сгруппировать их вместе, создавая группу окон времени обслуживания. Группы окон используются только для удобства, и их применение не обязательно.

## Управление атрибутами Scheduler

В начальных разделах этой главы уже было показано, как использовать процедуру SET\_ATTRIBUTE для модификации различных компонентов Scheduler. Такие атрибуты, как JOB\_NAME и PROGRAM\_NAME, уникальны для компонентов задания и программы.

Извлечь атрибуты любого компонента Scheduler можно с помощью процедуры GET\_SCHEDULER\_ATTRIBUTE из пакета DBMS\_SCHEDULER.

## Отмена установки атрибутов компонента

Процедура SET\_ATTRIBUTE\_NULL служит для установки атрибутов компонента Scheduler в NULL. Например, для отмены установки атрибута COMMENTS программы TEST\_PROGRAM можно воспользоваться следующим кодом:

SQL> EXECUTE dbms\_scheduler.SET\_ATTRIBUTE\_NULL('TEST\_PROGRAM', 'COMMENTS');

### Изменение атрибутов компонентов

Существуют некоторые атрибуты, общие для всех компонентов Scheduler. Процедура SET\_SCHEDULER\_ATTRIBUTE позволяет устанавливать эти общие, или глобальные, значения атрибутов, что затрагивает все компоненты Scheduler. Общие атрибуты включают часовой пояс по умолчанию, период хранения журнала хронологии и максимальное количество рабочих процессов задания.

### Мониторинг заданий Scheduler

Существует несколько динамических представлений, которые можно использовать для мониторинга заданий Scheduler, и ниже кратко описаны важнейшие из них.

#### **DBA\_SCHEDULER\_JOBS**

Представление DBA\_SCHEDULER\_JOBS показывает состояние и общую информацию о запланированных заданиях в базе данных. Вот простой запрос к этому представлению:

SQL> SELECT job\_name, program\_name

2 FROM DBA\_SCHEDULER\_JOBS;

JOB\_NAME PROGRAM\_NAME

----------------- -----------------

PURGE\_LOG PURGE\_LOG\_PROG

GATHER\_STATS\_JOB GATHER\_STATS\_PROG

. . .

SQL>

#### **DBA\_SCHEDULER\_RUNNING\_JOBS**

Представление DBA\_SCHEDULER\_RUNNING\_JOBS показывает информацию о текущих выполняющихся заданиях.

#### **DBA\_SCHEDULER\_JOB\_RUN\_DETAILS**

Представление DBA\_SCHEDULER\_JOB\_RUN\_DETAILS можно использовать для проверки состояния и длительности выполнения всех заданий в базе данных, например:

SQL> SELECT job\_name, status, run\_duration

2\* FROM DBA\_SCHEDULER\_JOB\_RUN\_DETAILS;

JOB\_NAME STATUS RUN\_DURATION

----------------- ---------- -------------

PURGE\_LOG SUCCEEDED +000 00:00:02

PURGE\_LOG SUCCEEDED +000 00:00:04

GATHER\_STATS\_JOB SUCCEEDED +000 00:31:18

SQL>

#### **DBA\_SCHEDULER\_SCHEDULES**

Представление DBA\_SCHEDULER\_SCHEDULES содержит информацию обо всех текущих расписаниях заданий в базе данных, как показано ниже:

SQL> SELECT schedule\_name, repeat\_interval

2\* FROM dba\_scheduler\_schedules;

SCHEDULE\_NAME REPEAT\_INTERVAL

--------------------- ------------------------------------------

DAILY\_PURGE\_SCHEDULE freq=daily;byhour=12;byminute=0;bysecond=0

SQL>

#### **DBA\_SCHEDULER\_JOB\_LOG**

Представление DBA\_SCHEDULER\_JOB\_LOG позволяет выполнять аудит деятельности, связанной с управлением заданиями в базе данных. Данные, содержащиеся в этом представлении, зависят от того, как установлены параметры ведения журналов для заданий и классов.

Ранее в разделе “Создание класса заданий” настоящей главы было показано, как устанавливается уровень ведения журналов на уровне задания или класса заданий. Для того настройки уровней ведения журналов для индивидуального задания используется процедура SET\_ATTRIBUTE из пакета DBMS\_SCHEDULER. В процедуре SET\_ATTRIBUTE можно установить атрибут LOGGING\_LEVEL в два разных значения:

* DBMS\_SCHEDULER.LOGGING\_FULL
* DBMS\_SCHEDULER.LOGGING\_RUNS

Опция DBMS\_SCHEDULER.LOGGING\_RUNS просто фиксирует запуски задания, в то время как опция DBMS\_SCHEDULER.LOGGING\_FULL включает полное ведение журналов задания.

Ниже приведен пример включения полного ведения журналов на уровне задания:

SQL> EXECUTE dbms\_scheduler.set\_attribute ('TESTJOB',

'LOGGING\_LEVEL', dbms\_scheduler.LOGGING\_FULL);

### Очистка журналов заданий

По умолчанию раз в день Scheduler будет очищать все журналы окон и журналы заданий, которые старше 30 дней. Очистить журналы можно также и вручную, выполнив процедуру PURGE\_LOG, как показано ниже:

SQL> EXECUTE DBMS\_SCHEDULER.PURGE\_LOG(

LOG\_HISTORY => 1,

JOB\_NAME => 'TEST\_JOB1');

### Здания Scheduler по умолчанию

По умолчанию все базы данных Oracle Database 11.1 используют Scheduler для запуска следующих заданий, хотя при желании можно отключить любое из них:

SQL> SELECT owner, job\_name, job\_type FROM dba\_scheduler\_jobs;

OWNER JOB\_NAME JOB\_TYPE

---------------------- ------------------------------ ----------------

SYS ADV\_SQLACCESS1821051 PLSQL\_BLOCK

SYS XMLDB\_NFS\_CLEANUP\_JOB STORED\_PROCEDURE

SYS FGR$AUTOPURGE\_JOB PLSQL\_BLOCK

SYS BSLN\_MAINTAIN\_STATS\_JOB

SYS DRA\_REEVALUATE\_OPEN\_FAILURES STORED\_PROCEDURE

SYS HM\_CREATE\_OFFLINE\_DICTIONARY STORED\_PROCEDURE

SYS ORA$AUTOTASK\_CLEAN

SYS PURGE\_LOG

ORACLE\_OCM MGMT\_STATS\_CONFIG\_JOB STORED\_PROCEDURE

ORACLE\_OCM MGMT\_CONFIG\_JOB STORED\_PROCEDURE

EXFSYS RLM$SCHDNEGACTION PLSQL\_BLOCK

EXFSYS RLM$EVTCLEANUP PLSQL\_BLOCK

12 rows selected.

SQL>

**Oracle Scheduler** — замечательное дополнение к арсеналу инструментов администратора баз данных Oracle. Предоставляя изощренные средства планирования сложных заданий, оно исключает необходимость в инструментах от независимых поставщиков или сложных сценариях оболочки для планирования заданий внутри базы данных.