Оглавление

Исключения

Ошибки, возникающие при работе с СУБД, можно разделить на следующие группы:

- ошибки, генерируемые системой (например, нехватка памяти или повторяющееся значение индекса);
- ошибки, генерируемые приложением (например, невыполнение каких-либо условий и проверок).

В языке PL/SQL ошибки всех видов интерпретируются как исключительные ситуации, или исключения.

Исключения могут быть ***системными*** и ***пользовательскими***:

Системные исключения	Пользовательские исключения
Определены в СУБД.	Определяются программистом в

- ***неименованные исключения*** - имеют только номера (ORA-02292) - ***именованные исключения*** – имеют как номера, так и названия (например, ORA-01403: NO_DATA_FOUND)	приложении. Имеют номер в диапазоне от -20999 до -20000 и текстовое описание. Инициируются с помощью RAISE_APPLICATION_ERROR

Предупреждающие сообщения при компиляции

Исполняющую среду возможно сконфигурировать таким образом, чтобы при компиляции программных модулей происходила выдача сообщений, предупреждающих о моментах, на которые следует обратить внимание - например, при попытке использования в хранимой процедуре уже неподдерживаемых возможностей PL/SQL.

Категории предупреждающих сообщений:

Категория	Описание	Пример
SEVERE	Условия, которые могут привести к неожиданным последствиям или некорректным результатам	Использование INTO при объявлении курсора
PERFORMANCE	Код, приводящий к снижению производительности	Использование значения VARCHAR2 для поля с типом NUMBER в операторе INSERT
INFORMATIONAL	Условия, которые не влияют на производительность, но усложняют чтение кода	Код, который никогда не будет выполнен

Конфигурирование производится посредством установки значения параметра PLSQL_WARNINGS.

Посредством установки параметра PLSQL_WARNINGS можно:

- включать и отключать либо все предупреждающие сообщения, либо сообщения одной или нескольких категорий, либо конкретное сообщение;
- трактовать конкретные предупреждения как ошибки.

Значение этого параметра можно задавать для:

- всего экземпляра базы данных (ALTER SYSTEM);
- текущего сеанса (ALTER SESSION);
- хранимого PL/SQL-модуля (ALTER "PL/SQL block").

Во всех ALTER-операторах значение параметра PLSQL_WARNINGS задается в следующем виде:

SET PLSQL_WARNINGS = '*value_clause*' [, '*value_clause*'] ...

, где

value_clause::=

{ ENABLE | DISABLE | ERROR }:

{ ALL | SEVERE | INFORMATIONAL | PERFORMANCE | { integer | (integer [, integer] ...) } }

Для отображения предупреждающих сообщений, сгенерированных в процессе компиляции, можно либо опрашивать представления *_ERRORS (DBA_, USER_, ALL_), либо использовать команду SHOW ERRORS.

Несколько примеров настройки режима выдачи предупреждений

Включение всех предупреждений внутри сессии (полезно при разработке):

ALTER SESSION SET PLSQL_WARNINGS='ENABLE:ALL';

Включение сообщений PERFORMANCE для сессии:

ALTER SESSION SET PLSQL_WARNINGS='ENABLE:PERFORMANCE';

Включение сообщений PERFORMANCE для процедуры loc_var:

ALTER PROCEDURE loc_var COMPILE PLSQL_WARNINGS='ENABLE:PERFORMANCE';

Включение сообщений SEVERE, отключение сообщений PERFORMANCE и трактования сообщения PLW-06002 (unreachable code) как ошибки:

ALTER SESSION SET PLSQL_WARNINGS='ENABLE:SEVERE', 'DISABLE:PERFORMANCE', 'ERROR:06002';

Отключение всех предупреждающих сообщений для текущей сессии:

ALTER SESSION SET PLSQL_WARNINGS='DISABLE:ALL';

Для просмотра текущего значения PLSQL_WARNINGS следует обратиться к представлению ALL_PLSQL_OBJECT_SETTINGS.

Обработка исключений в PL/SQL

PL/SQL перехватывает ошибки и реагирует на них при помощи так называемых *обработчиков исключений*. Механизм функционирования обработчиков исключений позволяет четко отделить код обработки ошибок от исполняемых операторов, дает возможность реализовать обработку ошибок, управляемую событиями, отказавшись от устаревшей линейной модели программирования.

Независимо от того, как и по какой причине было инициировано конкретное исключение, оно обрабатывается одним и тем же обработчиком в разделе исключений.
Любая ошибка может быть обработана только одним обработчиком.

Для обработки исключений в блоке PL/SQL предназначается необязательный раздел **EXCEPTION**:

```
BEGIN
    операторы

EXCEPTION
    WHEN [исключение 1]THEN …..;
    WHEN [исключение 2]THEN …..;
    …
    WHEN [исключение N]THEN …..;
    WHEN OTHERS THEN …..;
```

END;

Если в исполняемом блоке PL/SQL инициируется исключение, то выполнение блока прерывается и управление передается в раздел обработки исключений (если таковой имеется). После обработки исключения возврат в исполняемый блок уже невозможен, поэтому управление передается в родительский блок.

Обработчик **WHEN OTHERS** должен быть последним обработчиком в блоке, иначе возникнет ошибка компиляции. Этот обработчик не является обязательным. Если он отсутствует, то все необработанные исключения передадутся в родительский блок, либо в вызывающую хост-систему.

В одном предложении WHEN, можно объединить несколько исключений, используя оператор OR:

```
WHEN invalid_company_id OR negative_balance THEN
```

Также в одном обработчике можно комбинировать имена пользовательских и системных исключений:

```
WHEN balance_too_low OR zero_divide OR dbms_ldap.invalid_session THEN
```

Создание собственных исключений

Внутри приложения можно определять свои собственные (пользовательские) исключения.
Сделать это можно в разделе объявлений блока PL/SQL следующим образом:

```
DECLARE
  INVALID_COMPANY_ID  EXCEPTION;
```

Для того, чтобы инициировать исключение, необходимо воспользоваться оператором RAISE:

```
raise INVALID_COMPANY_ID;
```

После этого выполнение программы переходит в раздел EXCEPTION на соответствующий обработчик:

```
BEGIN
        .....
        raise INVALID_COMPANY_ID;
EXCEPTION
        when DUP_VAL_ON_INDEX then
                ....
        when INVALID_COMPANY_ID   then
                ....
END;
```

Для того, чтобы присвоить ошибке номер и создать для нее текстовое описание, следует воспользоваться процедурой **RAISE_APPLICATION_ERROR**:

```
RAISE_APPLICATION_ERROR(-20000, 'My error!');
```

Связываем исключение с кодом ошибки

Предположим, у нас есть программа, при выполнении которой может сгенерироваться ошибка, связанная с данными, например ORA-01843: not a valid month.
Для перехвата этой ошибки в код программы потребуется поместить такой обработчик:

```
EXCEPTION
    WHEN OTHERS THEN
        IF SQLCODE = -1843 THEN /* not a valid month */
```

Но такой код малопонятен.

Конкретную ошибку Oracle можно привязать к именованному исключению с помощью директивы компилятора EXCEPTION_INIT:

```
DECLARE
    invalid_month EXCEPTION;
    PRAGMA EXCEPTION_INIT(invalid_month, -1843);
BEGIN
    ......
EXCEPTION
    WHEN invalid_month THEN .....
END;
```

Теперь имя ошибки говорит само за себя и никакие литеральные номера ошибок, которые трудно запомнить, не понадобятся.
Установив такую связь, можно инициировать исключение по имени и использовать это имя в предложении WHEN обработчика ошибок.

Именованные системные исключения

В Oracle для некоторых системных исключений определены стандартные имена, которые заданы с помощью директивы компилятора EXCEPTION_INIT во встроенных пакетах.
Наиболее важные и широко применяемые из них определены в пакете STANDARD.
То обстоятельство, что этот пакет используется по умолчанию, означает, что на определенные в нем исключения можно ссылаться без указания в качестве префикса имени пакета.
Например, если необходимо обработать в программе исключение NO_DATA_FOUND, то это можно сделать посредством любого из двух операторов:

```
WHEN NO_DATA_FOUND THEN
WHEN STANDARD.NO_DATA_FOUND THEN
```

Именованные системные исключения

Название	Код
ACCESS_INTO_NULL	-6530
CASE_NOT_FOUND	-6592

Название	Код
PROGRAM_ERROR	-6501
ROWTYPE_MISMATCH	-6504

COLLECTION_IS_NULL	-6531
CURSOR_ALREADY_OPEN	-6511
DUP_VAL_ON_INDEX	-1
INVALID_CURSOR	-1001
INVALID_NUMBER	-1722
LOGIN_DENIED	-1017
NO_DATA_FOUND	+100
NO_DATA_NEEDED	-6548
NOT_LOGGED_ON	-1012

SELF_IS_NULL	-30625
STORAGE_ERROR	-6500
SUBSCRIPT_BEYOND_COUNT	-6533
SUBSCRIPT_OUTSIDE_LIMIT	-6532
SYS_INVALID_ROWID	-1410
TIMEOUT_ON_RESOURCE	-51
TOO_MANY_ROWS	-1422
VALUE_ERROR	-6502
ZERO_DIVIDE	-1476

Инициирование исключений

Программно инициировать исключение можно посредством оператора RAISE или процедуры RAISE_APPLICATION_ERROR.

Оператор RAISE

С помощью оператора RAISE можно инициировать как собственные, так и системные исключения. Оператор имеет три формы:

RAISE имя_исключения	Инициирование исключения, определенного в текущем блоке, а также инициирование системных исключений, объявленных в пакете STANDARD
RAISE имя_пакета.имя_исключения	Если исключение объявлено в любом другом пакете, отличном от STANDARD, имя исключения нужно уточнять именем пакета
RAISE	Не требует указывать имя исключения, но используется только в предложении WHEN раздела исключений. Этой формой оператора следует пользоваться, когда в обработчике исключений нужно повторно инициировать то же самое исключение

Использование процедуры RAISE_APPLICATION_ERROR

Для инициирования исключений, специфических для приложения, в Oracle существует процедура RAISE_APPLICATION_ERROR. Ее преимущество перед оператором RAISE (который тоже может инициировать специфические для приложения явно объявленные исключения) заключается в том, что она позволяет связать с номером исключения некоторое текстовое сообщение об ошибке.

```
PROCEDURE RAISE_APPLICATION_ERROR(num BINARY_INTEGER,
                                  msg VARCHAR2,
                                  keeperrorstack boolean default false);
```

Здесь

num – это номер ошибки из диапазона от -20999 до -20000;

msg - это сообщение об ошибке, длина которого не должна превышать 2048 символов (символы, выходящие за эту границу, игнорируются);

keep errorstack – параметр указывает, хотите вы добавить ошибку к тем, что уже имеются в стеке (true), или заменить существующую ошибку (значение по умолчанию – false).

Использование функций обработки ошибок

SQLCODE

Предложение WHEN OTHERS используется для перехвата исключений, не указанных в предложениях WHEN. Однако в этом обработчике тоже нужна информация о том, какая именно ошибка произошла. Для ее получения можно воспользоваться функцией ***SQLCODE***, возвращающей номер возникшей ошибки (значение 0 указывает, что в стеке ошибок нет ни одной ошибки).

SQLERRM

Возвращает поясняющее сообщение для текущей или для указанной ошибки:

SQLERRM – возвратит описание для самой последней ошибки

SQLERRM(code NUMBER) – возвратит описание для ошибки с указанным кодом

DBMS_UTILITY.FORMAT_CALL_STACK

Функция возвращает отформатированную строку со стеком вызовов в приложении PL/SQL.

DBMS_UTILITY.FORMAT_ERROR_STACK

Эта функция, как и SQLERRM, возвращает сообщение, связанное с текущей ошибкой.

Ее отличия от SQLERRM:

- она возвращает до 2000 символов (SQLERRM возвращает 512 символов)
- этой функции нельзя в качестве аргумента передать код ошибки

DBMS_UTILITY.FORMAT_ERROR_BACKTRACE

Функция появилась в Oracle 10.

Она возвращает отформатированную строку с содержимым стека программ и номеров строк. Ее выходные данные позволяют отследить строку, в которой изначально была инициирована ошибка.

Продолжение работы после возникновения исключения

Если согласно бизнес-логики задачи необходимо обработать исключение и продолжить работу, начиная с того места, где одно произошло, то одним из вариантов решения может быть размещение каждой инструкции в собственном PL/SQL-блоке со своим обработчиком исключений. Тогда при возникновении исключения управление будет передано следующей инструкции.

Эскалация необработанного исключения

Инициированное исключение обрабатывается в соответствии с определенными правилами. Сначала PL/SQL ищет обработчик исключения в текущем блоке (анонимном блоке, процедуре или функции). Если такового не нашлось, исключение передается в родительский блок. Затем PL/SQL пытается обработать исключение, инициировав его еще раз в родительском блоке. И так в каждом внешнем по отношению к другому блоке до тех пор, пока все они не будут исчерпаны. После этого PL/SQL возвращает необработанное исключение в среду приложения, из которого был выполнен самый внешний блок PL/SQL.

На что стоит обратить внимание

1. Если при выполнении нескольких DML-операций в SQL-среде возникает исключительная ситуация, то все операции, предшествующие ошибочному оператору, считаются выполненными корректно и не откатываются.
2. Если те же самые DML-операции обернуть в блок BEGIN … END - тогда при возникновении исключительной ситуации на очередном DML-операторе все предыдущие успешно (!) выполненные операции откатываются. Откат происходит к моменту начала выполнения блока. Т.е. блок либо выполняется целиком, либо не выполняется совсем.
3. Если обработчик завершается с повторной инициацией исключительной ситуации (напр., WHEN OTHERS then **raise**), то все изменения, проделанные в блоке, откатываются.
4. Если же выход из блока происходит через обработку исключительной ситуации и повторной инициации исключительной ситуации не происходит (напр., WHEN OTHERS then **null**), то блок считается исполненным успешно и отката изменений, которые внутри него произошли, не будет (!!!). То есть результат работы операторов, предшествующих ошибочному оператору, останется в БД.
   Поэтому, если по бизнес-логике такого не нужно, то в обработчике исключения надо явно делать ROLLBACK.

Динамический SQL и динамический PL/SQL

Статическими называются жестко закодированные инструкции и операторы, которые не изменяются с момента компиляции программы.

Инструкции динамического SQL формируются, компилируются и вызываются непосредственно во время выполнения программы.

Следует отметить, что такая гибкость языка открывает перед программистами огромные возможности и

позволяет писать универсальный код многократного использования.

Начиная с Oracle7 поддержка динамического SQL осуществляется с помощью встроенного пакета DBMS_SQL. В Oracle 8i для этого появилась еще одна возможность — встроенный динамический SQL (Native Dynamic SQL, ***NDS***). NDS интегрируется в язык PL/SQL; пользоваться им намного удобнее, чем DBMS_SQL.

На практике NDS в подавляющем большинстве является более предпочтительным решением.

Инструкции NDS

Главным достоинством NDS является его простота.

NDS представлен в языке PL/SQL единственной инструкцией ***EXECUTE IMMEDIATE***, немедленно выполняющей заданную SQL инструкцию, а также расширением инструкции **OPEN FOR**, позволяющей выполнять сложные динамические запросы.

В отличие от пакета DBMS_SQL, для работы с которым требуется знание десятка процедур и множества правил их использования, при использовании NDS все очень просто.

Инструкция EXECUTE IMMEDIATE

Инструкция **EXECUTE IMMEDIATE**, используемая для выполнения необходимой SQL-инструкции, имеет следующий синтаксис:

```
EXECUTE IMMEDIATE строка_SQL
[INTO {переменная[, переменная]...| запись}]
[USING [ IN | OUT | IN OUT ] аргумент
    [, [ IN | OUT | IN OUT ] аргумент]...];
```

где

- ***строка_SQL*** — строковое выражение, содержащее SQL-инструкцию или блок PL/SQL;
- ***переменная*** — переменная, которой присваивается содержимое поля, возвращаемого запросом;
- ***запись*** — запись, основаниая на типе данных который определяется пользавателем или объявляется с помощью атрибута %ROWTYPE, и принимающая всю возвращаемую запросом строку;
- ***аргумент*** — выражение, значение которого передается SQL-инструкции или блоху PL/SQL, либо идентификатор, являющийся входной и/или выходной переменной для функции или процедуры, вызываемой из блока PL/SQL;
- ***INTO*** — предложение, используемое для однострочных запросов (для каждого возвращаемого запросом столбца в этом предложении должна быть задана

отдельная переменная или же ему должно соответствовать поле записи совместимого типа);

- **USING** - предложение, определяющее параметры SQL-инструкции и используемое как в динамическом SQL, так и в динамическом PL/SQL (способ передачи параметра дается только в PL/SQL, причем по умолчанию для него установлен режим передачи IN).

Инструкция EXECUTE IMMEDIATE может использоваться для выполнения любой SQL-инструкции или PL/SQL-блока, за исключением многострочных запросов.
Если SQL-строка заканчивается точкой с запятой, она интерпретируется как блок PL/SQL. В противном случае воспринимается как DML- или DDL-инструкция.
Строка может содержать формальные параметры, но с их помощью не могут быть заданы имена объектов схемы, скажем, такие, как имена столбцов таблицы.
При выполнении инструкции исполняющее ядро заменяет в SQL-строке формальные параметры (идентификаторы, начинающиеся с двоеточия) фактическими значениями параметров подстановки в предложении USING.
В инструкции EXECUTE IMMEDIATE не разрешается передача литерального значения NULL — вместо него следует указывать переменную соответствующего типа, содержащую это значение.

Несколько примеров:

- Создание индекса:

```
EXECUTE IMMEDIATE ‘CREATE INDEX emp_u_l ON employee (last_name)’;
```

- Хранимую процедуру, выполняющую любую инструкцию DDL, можно создать так:

```
CREATE OR REPLACE PROCEDURE execDDL(ddl_string in varchar2) is
BEGIN
        EXECUTE IMMEDIATE ddl_string;
END;
```

  При наличии процедуры создание того же индекса выглядит так:

```
BEGIN
        execDDL(‘CREATE INDEX emp_u_l ON employee (last_name)’);
END;
```

-
```
DECLARE
  v_emp_last_name    VARCHAR2(50);
  v_emp_first_name    VARCHAR2(50);
  v_birth       DATE;
BEGIN
  EXECUTE IMMEDIATE 'select emp_last_name, emp_first_name, birth ‘ ||
                    ’from EMPLOYEE where id = :id'
    INTO v_emp_last_name, v_emp_first_name, v_birth
```

```
    USING 178;
  dbms_output.put_line(v_emp_last_name);
  dbms_output.put_line(v_emp_first_name);
  dbms_output.put_line(to_char(v_birth, 'dd.mm.yyyy'));
END;
```

Инструкция OPEN FOR

Синтаксис инструкции **OPEN FOR** таков:

OPEN {переменная_курсор | :хост_переменная_курсор} FOR строка_SQL
[USING аргумент [, аргумент]...];

Здесь

- ***переменная_курсор*** – слаботипизированная переменная-курсор (SYS_REFCURSOR);
- ***:хост_переменная_курсор*** - переменная-курсор, объявленная в хост-среде PL/SQL;
- ***строка_SQL*** – инструкция SELECT, подлежащая динамическому выполнению;
- ***USING*** – такое же предложение, как в **EXECUTE IMMEDIATE**.

Режимы использования параметров

- При передаче значений параметров SQL-инструкции можно использовать один из трех режимов:

- IN (только чтение, задан по умолчанию);
- OUT (только запись);
- IN OUT (чтение и запись).

Когда выполняется динамический запрос, все параметры SQL-инструкции, за исключением параметра в предложении RETURNING, должны передаваться в режиме IN:

```
DECLARE
  v_emp_name1 VARCHAR2(50) := 'Марина';
  v_emp_name2 VARCHAR2(50) := 'Иванова';
  v_emp_name  VARCHAR2(50);
  v_id_emp  NUMBER := 1666;
BEGIN
  EXECUTE IMMEDIATE 'update ADM.EMPLOYEE ' ||
            'set    emp_name1 = :v_emp_name1, ' ||
                  'emp_name2 = :v_emp_name2 ' ||
            'where  id_emp = :v_id_emp ' ||
            'returning emp_name1 into :val'
    USING IN v_emp_name1, IN v_emp_name2, IN v_id_emp, OUT v_emp_name;
  dbms_output.put_line(v_emp_name);
END;
/
```

Дублирование формальных параметров

При выполнении динамической SQL-инструкции связь между формальными и фактическими параметрами устанавливается в соответствии с их позициями. Однако интерпретация одноименных параметров зависит от того, какой код, SQL или PL/SQL, выполняется с помощью оператора EXECUTE IMMEDIATE:

При выполнении динамической SQL-инструкции (DML- или DDL-строки, ***не*** оканчивающейся точкой с запятой) параметр подстановки нужно задать для каждого формального параметра, даже если их имена повторяются.

Когда выполняется динамический блок PL/SQL (строки, оканчивающейся точкой с запятой), нужно указать параметр подстановки для каждого уникального формального параметра.

Передача значений NULL

При попытке передать NULL в качестве параметра подстановки:

```
EXECUTE IMMEDIATE 'UPDATE employee SET salary = :newsal WHERE hire_date IS NULL'
USUNG NULL;
```

произойдет ошибка.

Дело в том, что NULL типа данных не имеет и поэтому не может являться значением одного из типов данных SQL.

Преодолеть это можно так:

1. Можно использовать неинициализированную переменную.

2. Можно преобразовать NULL в типизированное значение:

```
USING TO_NUMBER(NULL);
```

Использование пакета DBMS_SQL

Пакет DBMS_SQL предоставляет возможность использования в PL/SQL динамического SQL для выполнения DML- или DDL-операций.

Выполнение одного динамического оператора с использованием пакета DBMS_SQL состоит, как правило, из следующих шагов:

1. Связывание текста динамического оператора с курсором и его синтаксический анализ и разбор;
2. Связывание входных аргументов с переменными, содержащими реальные значения;
3. Связывание выходных значений с переменными вызывающего блока;

4. Указание переменных, в которые будут сохраняться выходные значения;
5. Выполнение оператора;
6. Извлечение строк;
7. Получение значений переменных, извлеченных запросом;
8. Закрытие курсора.

Ниже приведен перечень функций и процедур пакета DBMS_SQL:

Функции	
EXECUTE	Executes a given cursor
EXECUTE_AND_FETCH	Executes a given cursor and fetch rows
FETCH_ROWS	Fetches a row from a given cursor
IS_OPEN	Returns TRUE if given cursor is open
LAST_ERROR_POSITION	Returns byte offset in the SQL statement text where the error occurred
LAST_ROW_COUNT	Returns cumulative count of the number of rows fetched
LAST_ROW_ID	Returns ROWID of last row processed
LAST_SQL_FUNCTION_CODE	Returns SQL function code for statement
OPEN_CURSOR	Returns cursor ID number of new cursor
TO_CURSOR_NUMBER	Takes an OPENed strongly or weakly-typed ref cursor and transforms it into a DBMS_SQL cursor number
TO_REFCURSOR	Takes an OPENed, PARSEd, and EXECUTEd cursor and transforms/migrates it into a PL/SQL manageable REF CURSOR (a weakly-typed cursor) that can be consumed by PL/SQL native dynamic SQL switched to use native dynamic SQL
Процедуры	
BIND_ARRAY	Binds a given value to a given collection
BIND_VARIABLE	Binds a given value to a given variable
CLOSE_CURSOR	Closes given cursor and frees memory
COLUMN_VALUE	Returns value of the cursor element for a given position in a cursor
COLUMN_VALUE_LONG	Returns a selected part of a LONG column, that has been defined using DEFINE_COLUMN_LONG
DEFINE_ARRAY	Defines a collection to be selected from the given cursor, used only with SELECT statements
DEFINE_COLUMN	Defines a column to be selected from the given cursor, used only with SELECT statements
DEFINE_COLUMN_CHAR	Defines a column of type CHAR to be selected from the given cursor, used only with SELECT statements
DEFINE_COLUMN_LONG	Defines a LONG column to be selected from the given cursor, used only with SELECT statements
DEFINE_COLUMN_RAW	Defines a column of type RAW to be selected from the given cursor, used only with SELECT statements
DEFINE_COLUMN_ROWID	Defines a column of type ROWID to be selected from the given cursor, used only with SELECT statements
DESCRIBE_COLUMNS	Describes the columns for a cursor opened and parsed through DBMS_SQL
DESCRIBE_COLUMNS2	Describes the specified column, an alternative to DESCRIBE_COLUMNS
DESCRIBE_COLUMNS3	Describes the specified column, an alternative to DESCRIBE_COLUMNS
PARSE	Parses given statement
VARIABLE_VALUE	Returns value of named variable for given cursor

Когда следует использовать DBMS_SQL

Хотя встроенный динамический SQL гораздо проще применять, а программный код более короткий и понятный, но все же бывают случаи, когда приходится использовать пакет DBMS_SQL.
Это следующие случаи:

- Разбор очень длинных строк.
  Если строка длиннее 32К, то EXECUTE IMMEDIATE не сможет ее выполнить;
- Получение информации о столбцах запроса;
- Минимальный разбор динамических курсоров.
  При каждом выполнении EXECUTE IMMEDIATE динамическая строка разбирается заново (производится синтаксический анализ, оптимизация и построение плана выполнения запроса), поэтому в некоторых ситуациях это обходится слишком дорого, и тогда DBMS_SQL может оказаться эффективнее.

Новые возможности Oracle 11g

В Oracle 11g появились средства взаимодействия между встроенным динамическим SQL и DBMS_SQL: появилась возможность преобразования курсоров DBMS_SQL в курсорные переменные и наоборот.

- Функция **DBMS_SQL.TO_REFCURSOR**

Преобразует курсор, полученный вызовом DBMS_SQL.OPEN_CURSOR в курсорную переменную, объявленную с типом SYS_REFCURSOR.

- Функция **DBMS_SQL.TO_CURSOR**

Преобразует переменную REF CURSOR в курсор SQL, который затем может передаваться подпрограммам DBMS_SQL.

SQL Injection

SQL Injection – один из типов несанкционированного доступа к данным.
В результате выполнения SQL-инъекций становится возможным выполнять действия, которые не предполагались создателем процедуры.

Технику SQL Injection можно разделить на три группы:
- Statement modification;
- Statement injection;
- Data Type Conversion.

Statement modification

Statement modification – изменение динамического SQL-запроса таким образом, что он будет работать не так, как планировал разработчик.

Пусть имеется следующая функция:

```
create or replace function SQL_INJECTION(p_ename in varchar2) return varchar2 is
```

```
    v_ret varchar2(200);
    v_qry varchar2(200);
  begin
    v_qry := 'select job from scott.emp where ename = ''' || p_ename || '''';
    dbms_output.put_line(v_qry);
    execute immediate v_qry into v_ret;
    return v_ret;
  end SQL_INJECTION;
```

Если вызвать ее с параметром p_ename => **''' union select to_char(sal) from emp where ename = ''KING**', то получим доступ к зарплате сотрудника KING:

```
SQL> select sql_injection(p_ename => ''' union select to_char(sal) from scott.emp where ename = ''KING''')
king_salary from dual;
KING_SALARY
--------------------------------------------------------------------------------
5000
```

Запрос при этом будет выполняться такой:

```
select job from scott.emp where ename = ''
union
select to_char(sal) from scott.emp where ename = 'KING';
```

Statement injection

Statement injection - добавление еще одного DML- или DDL-оператора (или даже нескольких) к динамическому SQL-оператору.

Рассмотрим такую процедуру:

```
CREATE OR REPLACE PROCEDURE stmt_injection_demo(user_name    IN VARCHAR2) IS
  v_block VARCHAR2(4000);
BEGIN
  -- Следующий динамический блок уязвим для техники statement injection
  -- из-за использования конкатенации
  v_block := 'BEGIN
   DBMS_OUTPUT.PUT_LINE(''user_name: ' || user_name || ''');
  END;';
  dbms_output.put_line('PL/SQL Block: ' || v_block);
  EXECUTE IMMEDIATE v_block;
END stmt_injection_demo;
/
```

Если вызвать ее с параметром **user_name** => '**Andy''); update emp set sal = 2500 where ename = upper(''SMITH**', то в результате ее работы будет не только выведено на печать «user_name: Andy», но и еще будет увеличено значение поля sal у сотрудника SMITH.

Data Type Conversion

Еще один малоизвестный способ SQL-инъекций связан с использованием NLS-параметров сессии.

Создадим функцию data_type_conversion, которая по дате приема на работу выдает имя сотрудника:

```
CREATE OR REPLACE FUNCTION data_type_conversion(p_hiredate IN DATE) RETURN VARCHAR2 IS
  v_ret VARCHAR2(200);
  v_qry VARCHAR2(200);
BEGIN
  v_qry := 'select ename from scott.emp where hiredate = ''' || p_hiredate || '''';
  dbms_output.put_line(v_qry);
  EXECUTE IMMEDIATE v_qry INTO v_ret;
  RETURN v_ret;
END data_type_conversion;
```

Результат вызова этой функции:

```
SQL> select DATA_TYPE_CONVERSION(date '1982-01-23') result from dual;
RESULT
--------------------------------------------------------------------------------
MILLER
```

Если же задать формат даты, как указано ниже, и выполнить select-оператор:

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT='''' OR empno = ''7499''';
SQL> select DATA_TYPE_CONVERSION(date '1982-01-23') result from dual;
```

, то получим следующий результат:

```
RESULT
--------------------------------------------------------------------------------
ALLEN
```

Результат мы получим не тот, что ожидалось – из-за того, что наш запрос теперь стал выглядеть так:

```
select ename from scott.emp where hiredate = '' OR empno = '7499';
```

Методы защиты от SQL-инъекций

Если в приложении используется динамический SQL, то следует использовать следующие методы, которые не позволят злоумышленнику преодолеть наложенные ограничения:

Связывание переменных;

Если в функции SQL_INJECTION оператор для динамического выполнения конструировать не с помощью конкатенации, а с использоыванием связанной переменной, то это не позволит злоумышленнику изменить логику запроса:

```
v_qry := 'select job from scott.emp where ename = :p_ename';
execute immediate v_qry into v_ret using p_ename;
```

Проверка на соответствие ожидаемым значениям

Если пользователь передал номер департамента для выполнения операции DELETE, то сначала можно проверить, что такой департамент существует.

Аналогично, если в качестве значения параметра передается имя таблицы для удаления, пригодится проверка существования такой таблицы в базе данных путем выполнения обращения к представлению ALL_TABLES.

Для безопасного использования строковых литералов полезно использовать функцию DBMS_ASSERT.ENQUOTE_LITERAL, которая к переданной строке добавляет лидирующий и завершающий апострофы, одновременно контролируя отсутствие апострофов внутри строки.

Использование внутреннего преобразования формата

Если в процедуре, использующей динамический SQL, нет возможности использовать связанные переменные, и формирование оператора выполняется с помощью конкатенации, то в таком случае необходимо параметры преобразовывать в текст, используя внутреннее преобразование формата, которое не будет зависеть от настроек NLS, заданных внутри сессии.

Использование внутреннего преобразования рекомендуется не только с точки зрения безопасности, но и с точки зрения стабильной работоспособности приложения вне зависимости от национальных настроек окружения.

Преобразование в строковый формат следует использовать для переменных с типом DATE и NUMBER.