

Oracle Core

Тема 6

Exceptions & Dynamic SQL

Содержание

1. Исключительные ситуации

- Управление выдачей предупреждающих сообщений при компиляции
- Общая схема обработки исключений
 - категории исключений;
 - преимущества обработчиков исключений;
- Стандартные исключения
- Создание и обработка собственных исключений
- Обработка системных исключений
- Функции обработки ошибок
- Эскалация исключений
- Продолжение работы после обработки исключения - примеры

2. Динамический SQL

- Для чего нужен динамический SQL
- EXECUTE IMMEDIATE
- OPEN FOR
- Пакет DBMS_SQL
- SQL Injection
 - техника SQL Injection;
 - защита от SQL Injection.

Часть 1

Exceptions

Обработка исключений

Многие программисты, как правило, пишут приложения, предназначенные для работы в «лучшем из миров», где в программах не бывает ошибок, пользователи вводят лишь правильные данные и только должным образом, а все системы - и аппаратные, и программные - всегда в полном порядке.

Но жестокая реальность свидетельствует о том, что как бы вы ни старались, в приложении останутся ошибки. И пользователь обязательно воспользуется именно той непредусмотренной возможностью, на которую ваша программа отреагирует сбоем.

К счастью, PL/SQL предлагает достаточно мощный и гибкий механизм перехвата и обработки ошибок. И вполне возможно написать на языке PL/SQL такое приложение, которое полностью защитит от ошибок и пользователей, и базу данных.

Как в PL/SQL обрабатываются исключения

В языке PL/SQL ошибки всех видов интерпретируются как исключения.

К числу исключений относятся:

- ошибки, которые генерируются системой (в частности такие, как нехватка памяти или повторяющееся значение индекса);
- ошибки, которые генерируются приложением (невыполнение каких-либо условий и проверок);
- ошибки, вызванные действиями пользователя.

Концепция обработки исключений и терминология

Существуют следующие виды исключений:

- **Системные исключения**

Определены в Oracle и обычно инициируется ядром PL/SQL, обнаружившим ошибку.

Системные исключения можно также поделить на две категории:

- **неименованные исключения** - имеют только номера (ORA-02292)
- **именованные исключения** – имеют как номера, так и названия (например, ORA-01403: NO_DATA_FOUND)

Имена присваиваются наиболее используемым исключениям.

- **Исключения, определяемые программистом**

Определяется программистом, а следовательно, специфично для конкретного приложения. Имя исключения можно связать с конкретной ошибкой Oracle с помощью директивы компилятора EXCEPTION_INIT. Ошибке можно присвоить номер и создать для нее текстовое описание, воспользовавшись процедурой RAISE_APPLICATION_ERROR.

Определение исключений

- Прежде чем исключение можно будет инициировать и обрабатывать, его сначала нужно определить.
- В Oracle заранее определены тысячи исключений, большинство из которых имеют только номера и поясняющие сообщения. Имена присваиваются только самым распространенным исключениям.
- Можно определить собственные исключения и использовать их в своих приложениях.

Предупреждающие сообщения при компиляции

Но прежде, чем начать рассмотрение обработки ошибок, давайте затронем небольшой вопрос, связанный с возможностью выдачи предупреждающих сообщений при компиляции хранимых программных модулей – например, при попытке использования уже неподдерживаемых (deprecated) возможностей PL/SQL.

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

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 сервера БД	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

Предупреждающие сообщения при компиляции

Посредством установки параметра окружения 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*] ...) } }

Примеры

Включение всех предупреждений внутри сессии (полезно при разработке):

```
SQL> ALTER SESSION SET PLSQL_WARNINGS='ENABLE:ALL';
```

Включение сообщений PERFORMANCE для сессии:

```
SQL> ALTER SESSION SET PLSQL_WARNINGS='ENABLE:PERFORMANCE';
```

Включение сообщений PERFORMANCE для процедуры loc_var:

```
SQL> ALTER PROCEDURE loc_var COMPILE PLSQL_WARNINGS='ENABLE:PERFORMANCE';
```

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

```
SQL> ALTER SESSION SET PLSQL_WARNINGS='ENABLE:SEVERE', 'DISABLE:PERFORMANCE', 'ERROR:06002';  
Session altered
```

```
SQL> alter procedure unreachable_code compile;  
Warning: Procedure altered with compilation errors
```

Отключение всех предупреждающих сообщений для текущей сессии:

```
SQL> ALTER SESSION SET PLSQL_WARNINGS='DISABLE:ALL';
```

Для просмотра текущего значения PLSQL_WARNINGS следует обратиться к представлению ALL_PLSQL_OBJECT_SETTINGS.

Примеры

Создадим процедуру:

```
CREATE OR REPLACE PROCEDURE unreachable_code AUTHID DEFINER AS
  x CONSTANT BOOLEAN := TRUE;
BEGIN
  IF x
  THEN
    dbms_output.put_line( 'TRUE' );
  ELSE
    dbms_output.put_line( 'FALSE' );
  END IF;
END unreachable_code;
```

И выполним следующие команды:

```
SQL> ALTER SESSION SET PLSQL_WARNINGS='ENABLE:ALL';
Session altered
```

```
SQL> alter procedure unreachable_code compile;
Procedure altered
```

Теперь посмотрим на предупреждения:

```
SQL> show errors;
Errors for PROCEDURE UNREACHABLE_CODE:
LINE/COL ERROR
-----
8/9      PLW-06002: Unreachable code
```

Обработка исключений

Как только инициируется исключение, нормальное выполнение блока PL/SQL заканчивается и управление передается в раздел исключений.

Затем исключение либо обрабатывается обработчиком исключений в текущем блоке PL/SQL, либо передается в родительский блок.

Для того, чтобы обработать (или *перехватить*) исключение, нужно написать для него *обработчик*.

PL/SQL перехватывает ошибки и реагирует на них при помощи так называемых *обработчиков исключений*.

Обработка исключений

Механизм функционирования обработчиков исключений позволяет четко разделить код обработки ошибок от исполняемых операторов, а также 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 будет обработано одним и тем же обработчиком.

Обработчики исключений располагаются после исполняемой части блока, но перед завершающим его ключевым словом END.

Начало блок
EXCEPTION:

DECLARE

... объявления ...

BEGIN

... исполняемые операторы ...

[EXCEPTION

... обработчики исключений ...]

END;

ключевое слово

Как в PL/SQL обрабатываются исключения

Так выглядит программа, содержащая раздел обработки исключений:

```
BEGIN
.....
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    .....
  WHEN TOO_MANY_ROWS THEN
    .....
  WHEN OTHERS THEN
    .....
END;
```

Исполняемый блок

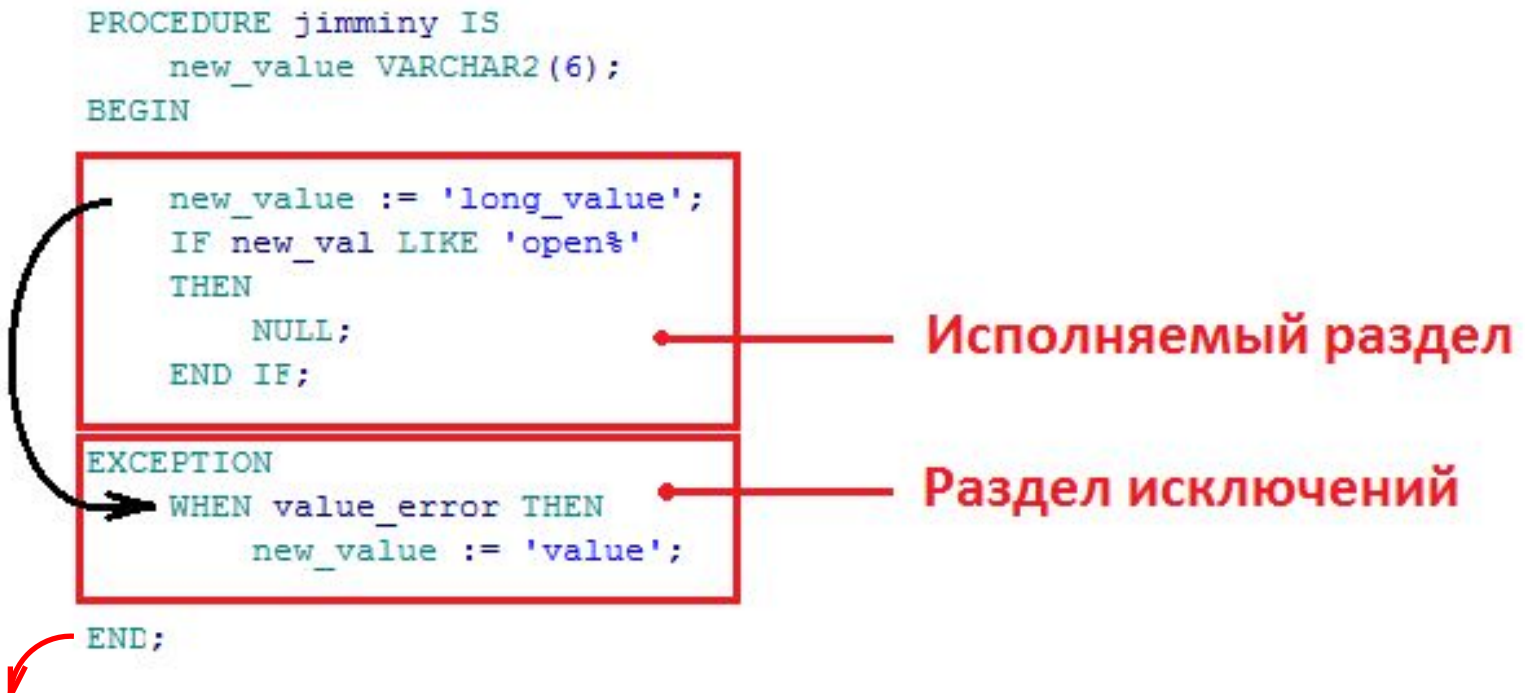
Блок обработки исключений

Обработчики исключений

Как в PL/SQL обрабатываются исключения

Если в блоке PL/SQL происходит ошибка, то инициируется исключение. В результате выполнение исполняемого блока прерывается и управление передается отдельному разделу исключений в текущем блоке, если таковой имеется.

После обработки исключения возврат в тот блок, из которого оно было инициировано, уже невозможен, поэтому управление переходит во внешний блок:



Обработка исключений

В одном разделе исключений может быть несколько обработчиков. Обработчики структурируются подобно условному оператору CASE:

EXCEPTION

```
WHEN NO_DATA_FOUND THEN
    исполняемые_операторы_1
-- Если инициировано исключение NO_DATA_FOUND,
-- то выполнить первый набор исполняемых операторов

WHEN payment_overdue THEN
    исполняемые_операторы_2
-- Если просрочена оплата (payment_overdue),
-- то выполнить второй набор исполняемых операторов

WHEN OTHERS THEN
    исполняемые_операторы_3
-- Если инициировано иное исключение,
-- то выполнить третий набор исполняемых операторов

END;
```

В одном предложении WHEN, используя оператор OR, можно объединить несколько исключений — точно так же, как с его помощью объединяются логические выражения:

```
WHEN имя_исключения [OR имя_исключения ... ]
THEN
    исполняемые_операторы
```

В одном обработчике можно также комбинировать имена пользовательских и системных исключений:

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


Обработка исключений

- Любая ошибка может быть перехвачена только одним обработчиком исключений.
- После выполнения операторов этого обработчика управление сразу же передается из текущего блока в родительский или вызывающий блок.
- Предложение WHEN OTHERS не является обязательным. Когда оно отсутствует, все необработанные исключения передаются в родительский блок, если таковой имеется.
- Если предложение WHEN OTHERS присутствует, то оно должно быть последним обработчиком в блоке.

Объявление собственных именованных исключений

Для того, чтобы обработать исключение, необходимо сначала задать ему имя.

Сделать это надо в разделе объявлений блока PL/SQL следующим образом:

DECLARE

ИМЯ_ИСКЛЮЧЕНИЯ EXCEPTION;

.....

Создание
исключений

```
PROCEDURE calc_annual_sales IS
```

```
invalid_company_id EXCEPTION;  
negative_balance    EXCEPTION;
```

```
duplicate_company BOOLEAN;
```

```
BEGIN
```

```
... исполняемые операторы ...
```

```
EXCEPTION
```

```
WHEN no_data_found THEN
```

```
.....
```

```
WHEN invalid_company_id THEN
```

```
.....
```

```
WHEN negative_balance THEN
```

```
.....
```

```
END;
```

Обработка
исключений

Объявление собственных именованных исключений

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;
```



Связываем имя исключения с кодом ошибки

Наличие в коде программы исключений без имен вполне допустимо, но такой код малопонятен и его трудно сопровождать.

Предположим, вы написали программу, при выполнении которой Oracle может сгенерировать ошибку, связанную 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, поэтому его обязательно нужно будет сопроводить комментарием.

* SQLCODE – встроенная функция, которая возвращает номер последней сгенерированной ошибки.

Использование директивы EXCEPTION_INIT

С помощью директивы компилятора EXCEPTION_INIT (команды, выполняемой во время компиляции программы) предложение WHEN, использовавшееся в предыдущем примере, можно изменить следующим образом:

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

Создаем пользовательское исключение

Связываем исключение с кодом -1843

После этого никакие литеральные номера ошибок, которые трудно запомнить, нам больше не понадобятся. Теперь имя ошибки говорит само за себя.

Установив такую связь, можно инициировать исключение по имени и использовать это имя в предложении WHEN обработчика ошибок.

Пример использования директивы

EXCEPTION INIT

Давайте рассмотрим пример возможного объявления исключения для обработки ошибки *ORA-2292 violated integrity constraining (OWNER.CONSTRAINT) - child record found*

```
PROCEDURE delete_company(company_id IN NUMBER) IS

    /* Объявляем исключение */
    still_have_employees EXCEPTION;

    /* Связываем имя исключения с номером ошибки */
    PRAGMA EXCEPTION_INIT(still_have_employees, -2292);

BEGIN
    /* Пытаемся удалить информацию о компании */
    DELETE FROM company
    WHERE id = company_id;

EXCEPTION

    /* Если найдена дочерняя запись, то будет инициировано это исключение */
    WHEN still_have_employees THEN
        dbms_output.put_line('Сначала нужно удалить данные о служащих компании');

END;
```

Об именованных системных исключениях

В Oracle для некоторых 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, 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 исключения в PL/SQL

Именованные системные исключения

Название	Код		Название	Код
ACCESS_INTO_NULL	-6530		PROGRAM_ERROR	-6501
CASE_NOT_FOUND	-6592		ROWTYPE_MISMATCH	-6504
COLLECTION_IS_NULL	-6531		SELF_IS_NULL	-30625
CURSOR_ALREADY_OPEN	-6511		STORAGE_ERROR	-6500
DUP_VAL_ON_INDEX	-1		SUBSCRIPT_BEYOND_COUNT	-6533
INVALID_CURSOR	-1001		SUBSCRIPT_OUTSIDE_LIMIT	-6532
INVALID_NUMBER	-1722		SYS_INVALID_ROWID	-1410
LOGIN_DENIED	-1017		TIMEOUT_ON_RESOURCE	-51
NO_DATA_FOUND	+100		TOO_MANY_ROWS	-1422
NO_DATA_NEEDED	-6548		VALUE_ERROR	-6502
NOT_LOGGED_ON	-1012		ZERO_DIVIDE	-1476

Как инициировать исключение?

Исключение может быть инициировано либо ядром Oracle при обнаружении ошибки, либо программистом.

Программист может инициировать исключения 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;
- RAISE имя_пакета.имя_исключения;
- RAISE.

Первая форма (без имени пакета) предназначена для инициирования исключений, определенных в текущем блоке, а также для инициирования системных исключений, объявленных в пакете STANDARD.

Если исключение объявлено в любом другом пакете, отличном от STANDARD, имя исключения нужно уточнять именем пакета.

Третья форма 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, Oracle предоставляет процедуру ***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.

При вызове этой процедуры выполнение текущего блока PL/SQL прекращается и любые изменения аргументов OUT и IN OUT (если таковые имеются) отменяются.

Изменения же, внесенные в глобальные структуры данных, такие как переменные пакетов и объекты баз данных с помощью инструкций INSERT, UPDATE или DELETE, **не отменяются**. Для отката DML-инструкций необходимо явно указать в разделе обработки исключений оператор ***rollback***.

```
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);

keeperrorstack – параметр указывает, хотите вы добавить ошибку к тем, что уже имеются в стеке (true), или заменить существующую ошибку (false).

По умолчанию параметр принимает значение false.

Пример использования процедуры RAISE_APPLICATION_ERROR

Следующая процедура по переданному ей коду сообщения выбирает текст сообщения об ошибке на языке, полученном из настроек NLS:

```
PROCEDURE raise_by_language(code_in IN PLS_INTEGER) IS

    l_message error_table.error_string;

BEGIN
    SELECT error_string
    INTO    l_message
    FROM    error_table,
           v$nls_parameters v
    WHERE   error_number = code_in
    AND     string_language = v.value
    AND     v.parameter = 'NLS_LANGUAGE';

    RAISE_APPLICATION_ERROR(code_in, l_message);

END;
```

Обзор функций обработки ошибок

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 текущей ошибки (значение 0 указывает, что в стеке ошибок нет ни одной ошибки).

Есть еще одна полезная функция – **SQLERRM**. Она возвращает поясняющее сообщение для текущей или для указанной ошибки:

SQLERRM – сообщение для текущей ошибки;

SQLERRM (n) – сообщение для указанной ошибки.

Недостаток функции SQLERRM в том, что она возвращает строку длиной не больше 512 символов.

В пакете **DBMS_UTILITY** также есть следующие полезные функции:

- **format_error_stack** – лучше использовать эту функцию вместо **SQLERRM** – она возвращает до 2000 символов;
- **format_call_stack** – возвращает текущий стек вызова;
- **format_error_backtrace** – возвращает полный стек вызова с момента возникновения исключительной ситуации.

Пример использования функции SQLCODE и SQLERRM

```
EXCEPTION
  WHEN OTHERS THEN
    /*
    || Анонимный блок внутри обработчика исключения позволяет
    || объявить локальные переменные для хранения информации об ошибке
    */
    DECLARE
      ERROR_CODE NUMBER := SQLCODE;
      /* Максимальная длина строки, возвращаемой функцией SQLERRM */
      error_msg VARCHAR2(512) := SQLERRM;
    BEGIN
      IF ERROR_CODE = -2292
      THEN
        /* Имеются дочерние записи - удалим их тоже */
        DELETE FROM employee
        WHERE  company_id = p_company_id;

        /* Теперь удалим родительскую запись */
        DELETE FROM company
        WHERE  id = p_company_id;

      ELSIF ERROR_CODE = -2291
      THEN
        /* Ключ родительской записи не найден */
        dbms_output.put_line('Компании с идентификатором ' || to_char(p_company_id) || ' не существует');
      ELSE
        /* Это что-то вроде WHEN OTHERS внутри WHEN OTHERS */
        dbms_output.put_line('Ошибка при удалении компании: ' || error_msg);
      END IF;
    END; -- Завершение анонимного блока
END; -- Завершение основного блока
```

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, а также не было и обработчика WHEN OTHERS, то исключение передается в родительский блок и PL/SQL пытается найти подходящий обработчик для этого исключения в родительском блоке. И так в каждом внешнем по отношению к другому блоке до тех пор, пока все они не будут исчерпаны. Если этот процесс завершился безрезультатно, то PL/SQL вернет необработанное исключение в среду приложения, из которого был выполнен самый внешний блок PL/SQL.



Продолжение работы после возникновения исключения

Когда в блоке PL/SQL инициируется исключение, нормальная работа программы прерывается и управление передается в раздел исключений. Однако и после обработки исключения управление не будет возвращено в тот блок, в котором оно было инициировано. А как же быть в тех случаях, когда необходимо обработать исключение и продолжить работу, начиная с того места, где одно произошло?

Предположим, нужно написать процедуру, которая выполняет последовательность DML-инструкций по отношению к разным таблицам (удаляет данные из одной таблицы, обновляет другую, добавляет строки в третью).

Первая версия этой процедуры может быть примерно такой:

```
PROCEDURE change_data IS
BEGIN
  1 DELETE FROM employee WHERE ... ;

  2 UPDATE company SET ... ;

  3 INSERT INTO company_history
    SELECT * FROM company WHERE ... ;
END;
```

Продолжение работы после возникновения исключения

Как же сделать так, чтобы выполнение программы происходило по необходимому нам сценарию?

Одним из вариантов решения может быть размещение каждой инструкции в собственном блоке обработки исключений

```
PROCEDURE change_data IS  
BEGIN
```

```
BEGIN  
    DELETE FROM employee WHERE ... ;  
EXCEPTION  
    WHEN OTHERS THEN  
        NULL;  
END;
```

```
BEGIN  
    UPDATE company SET ... ;  
EXCEPTION  
    WHEN OTHERS THEN  
        NULL;  
END;
```

```
BEGIN  
    INSERT INTO company_history  
        SELECT * FROM company WHERE ... ;  
EXCEPTION  
    WHEN OTHERS THEN  
        NULL;  
END;
```

```
END;
```

Продолжение работы после возникновения исключения

Было так:

Стало так:

Одно исключение – и выполнение программы прекращается:

```
PROCEDURE change_data IS
BEGIN


    DELETE FROM employee WHERE ... ;

    UPDATE company SET ... ;

    INSERT INTO company_history
        SELECT * FROM company WHERE ... ;

EXCEPTION
    WHEN OTHERS THEN
        NULL;

END;
```



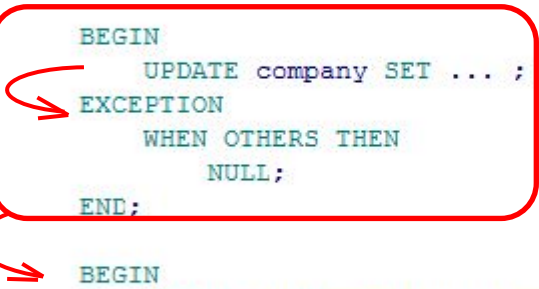
После возникновения исключения управление передается следующей инструкции:

```
PROCEDURE change_data IS
BEGIN
    BEGIN
        DELETE FROM employee WHERE ... ;
    EXCEPTION
        WHEN OTHERS THEN
            NULL;
    END;

    BEGIN
        UPDATE company SET ... ;
    EXCEPTION
        WHEN OTHERS THEN
            NULL;
    END;

    BEGIN
        INSERT INTO company_history
            SELECT * FROM company WHERE ... ;
    EXCEPTION
        WHEN OTHERS THEN
            NULL;
    END;

END;
```



Продолжение работы после возникновения исключения

Давайте теперь посмотрим на поведение БД в следующих примерах.

Создадим таблицу *test*:

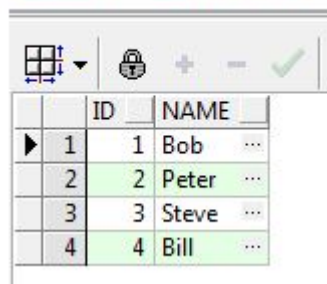
```
CREATE TABLE test(id NUMBER, name VARCHAR2(100));  
CREATE UNIQUE INDEX ui_test ON test(id);
```

Попытаемся добавить в нее в среде SQL пять записей:

```
SQL> insert into test(id, name) values(1, 'Bob');  
1 row inserted  
SQL> insert into test(id, name) values(2, 'Peter');  
1 row inserted  
SQL> insert into test(id, name) values(3, 'Steve');  
1 row inserted  
SQL> insert into test(id, name) values(4, 'Bill');  
1 row inserted  
SQL> insert into test(id, name) values(4, 'Jack');  
insert into test(id, name) values(4, 'Jack')  
~  
ORA-00001: unique constraint (DBADMIN.UI_TEST) violated
```

Пятая запись не была добавлена, поскольку нарушала условие уникальности поля ID.

`select * from test;` меет в таблице четыре записи:



The screenshot shows a database client interface with a table named 'test'. The table has two columns: 'ID' and 'NAME'. There are four rows of data. The first row has ID 1 and name Bob. The second row has ID 2 and name Peter. The third row has ID 3 and name Steve. The fourth row has ID 4 and name Bill. The table is displayed in a grid view with a toolbar above it.

	ID	NAME
1	1	Bob
2	2	Peter
3	3	Steve
4	4	Bill

Продолжение работы после возникновения исключения

Теперь очистим таблицу *test* и обернем добавляемые записи в PL/SQL-блок:

```
SQL> truncate table test;  
Table truncated
```

```
SQL>  
SQL> begin  
2   insert into test(id, name) values(1, 'Bob');  
3   insert into test(id, name) values(2, 'Peter');  
4   insert into test(id, name) values(3, 'Steve');  
5   insert into test(id, name) values(4, 'Bill');  
6   insert into test(id, name) values(4, 'Jack');  
7 end;  
8 /  
begin  
  insert into test(id, name) values(1, 'Bob');  
  insert into test(id, name) values(2, 'Peter');  
  insert into test(id, name) values(3, 'Steve');  
  insert into test(id, name) values(4, 'Bill');  
  insert into test(id, name) values(4, 'Jack');  
end;  
ORA-00001: unique constraint (DBADMIN.UI_TEST) violated  
ORA-06512: at line 6
```

```
select * from test;
```



ID	NAME
----	------

нет ни одной записи, поскольку PL/SQL-блок либо выполняется

Целиком,
либо не выполняется совсем.

Продолжение работы после возникновения исключения

Теперь снова очистим таблицу *test* и выполним PL/SQL-блок, добавив в него

```
O SQL> truncate table test;
Table truncated

SQL>
SQL> begin
  2     insert into test(id, name) values(1, 'Bob');
  3     insert into test(id, name) values(2, 'Peter');
  4     insert into test(id, name) values(3, 'Steve');
  5     insert into test(id, name) values(4, 'Bill');
  6     insert into test(id, name) values(4, 'Jack');
  7     exception
  8     when dup_val_on_index then
  9         raise;
 10 end;
 11 /
begin
insert into test(id, name) values(1, 'Bob');
insert into test(id, name) values(2, 'Peter');
insert into test(id, name) values(3, 'Steve');
insert into test(id, name) values(4, 'Bill');
insert into test(id, name) values(4, 'Jack');
exception
  when dup_val_on_index then
    raise;
end;
ORA-00001: unique constraint (DBADMIN.UI_TEST) violated
ORA-06512: at line 9
```

```
select * from test;
```



ID	NAME
----	------

В таблице ничего нет, поскольку PL/SQL-блок выполнялся с ошибкой и все изменения откатились.

Продолжение работы после возникновения исключения

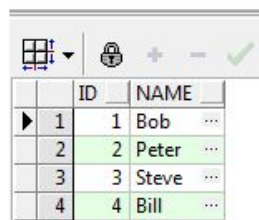
А теперь опять очистим таблицу **test** и выполним PL/SQL-блок, добавив в него обработчик:

```
SQL> truncate table test;
Table truncated

SQL>
SQL> begin
  2      insert into test(id, name) values(1, 'Bob');
  3      insert into test(id, name) values(2, 'Peter');
  4      insert into test(id, name) values(3, 'Steve');
  5      insert into test(id, name) values(4, 'Bill');
  6      insert into test(id, name) values(4, 'Jack');
  7  exception
  8      when dup_val_on_index then
  9          null;
 10 end;
 11 /
PL/SQL procedure successfully completed
```

Сейчас блок выполнен без ошибок, поскольку исключительная ситуация была обработана в разделе **EXCEPTION**.

```
select * from test;
```



	ID	NAME
1	1	Bob
2	2	Peter
3	3	Steve
4	4	Bill

Вывод: если выход из PL/SQL-блока происходит через раздел **EXCEPTION** и инициации повторного исключения не происходит, то считается, что PL/SQL- блок выполнен успешно и все операторы, выполненные до возникновения исключительной ситуации, не откатываются. Для их отката требуется явное выполнение оператора **rollback**.

Динамический SQL

Часть 2 Dynamic SQL

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 широкие возможности и позволяет писать универсальный код многократного использования.
- Динамический SQL может быть полезен при написании программ, когда при компиляции приложения еще неизвестен полный текст SQL-оператора или количество и типы данных входных и выходных переменных.
- Начиная с Oracle7 поддержка динамического SQL осуществляется с помощью встроенного пакета DBMS_SQL.
В Oracle 8i 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

Инструкции NDS

Главным достоинством NDS является его простота.

В отличие от пакета DBMS_SQL, для работы с которым требуется знание десятка процедур и множества правил их использования, NDS представлен в языке PL/SQL единственной инструкцией **EXECUTE IMMEDIATE**, немедленно выполняющей заданную SQL инструкцию, а также расширением существующей инструкции **OPEN FOR**, позволяющей выполнять сложные динамические запросы.

Инструкция EXECUTE IMMEDIATE

Инструкция **EXECUTE IMMEDIATE**, используемая для немедленного выполнении заданной SQL-инструкции, имеет следующий синтаксис:

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

Здесь

- **строка_SQL** — строковое выражение, содержащее SQL-инструкцию или блок PL/SQL;
- **переменная** — переменная, которой присваивается содержимое поля, возвращаемого запросом;
- **запись** — запись, основанная на типе данных который определяется пользователем или объявляется с помощью атрибута %ROWTYPE, и принимающая всю возвращаемую запросом строку;
- **аргумент** — выражение, значение которого передается SQL-инструкции или блоку 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

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

Инструкция EXECUTE IMMEDIATE

Несколько примеров использования:

- Создание индекса:

```
EXECUTE IMMEDIATE 'CREATE INDEX emp_u_1 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 хранимой процедуры создание того же индекса выглядит так:

```
BEGIN  
    execDDL('CREATE INDEX emp_u_1 ON employee  
(last_name)');  
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;

```
PROCEDURE show_parts_inventory(parts_table IN VARCHAR2,  
                               where_in    IN VARCHAR2 := NULL) IS  
    TYPE query_curtype IS REF CURSOR;  
    dyncur query_curtype;  
BEGIN  
    OPEN dyncur FOR  
        'select * from ' || parts_table ||  
        ' where ' || nvl(where_in, '1 = 1');  
    ...  
END;
```

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 (см. пример ниже)

Режимы использования параметров

```
PROCEDURE wrong_incentive(p_company_id  IN NUMBER,
                          p_new_layoffs IN NUMBER) IS

    sql_string      VARCHAR2(2000);
    sal_after_layoffs NUMBER;

BEGIN

    sql_string := 'update ceo_compensation
                   set     salary = salary + 10 * :layoffs
                   where  company_id = :company
                   returning salary into :newsal';

    EXECUTE IMMEDIATE sql_string
        USING p_new_layoffs, p_company_id, OUT sal_after_layoffs;

    dbms_output.put_line('CEO compensation after latest round of layoffs $' || sal_after_layoffs);

END;
```

В приведенном примере переменные *p_new_layoffs* и *p_company_id* используются для передачи значений в динамический SQL-оператор (IN), а *sal_after_layouts* – для получения обновленного значения (OUT)

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.

Дублирование формальных параметров

При выполнении SQL-инструкции параметр подстановки p_val повторяется:

```
PROCEDURE updnumval(p_col    VARCHAR2,  
                   p_start  DATE,  
                   p_end    DATE,  
                   p_val    NUMBER) IS  
    dml_str VARCHAR2(32767) :=  
        'update emp  
        set ' || p_col || ' = :val  
        where hiredate between :lodate and :hidate  
        and :val is not null';  
  
BEGIN  
    EXECUTE IMMEDIATE dml_str  
        USING p_val, p_start, p_end, p_val;  
END;
```

Дублирование формальных параметров

А при выполнении динамического блока PL/SQL параметр p_val задаем только один раз:

```
PROCEDURE updnumval(p_col  VARCHAR2,  
                   p_start DATE,  
                   p_end   DATE,  
                   p_val   NUMBER) IS  
    dml_str VARCHAR2(32767) :=  
        'begin  
          update emp  
          set ' || p_col || ' = :val  
          where hiredate between :lodate and :hidate  
          and :val is not null;  
        end;';  
BEGIN  
    EXECUTE IMMEDIATE dml_str  
        USING p_val, p_start, p_end;  
END;
```

Передача значений NULL

При попытке передать NULL в качестве параметра

```
П' EXECUTE IMMEDIATE
    'update employee
      set      sal = :newsal
      where hire_date is null'
    USING NULL;
```

произойдет ошибка.

Это происходит потому, что NULL не имеет типа данных и поэтому не может являться значением одного из типов данных SQL.

Способы передачи значения NULL

1. Можно использовать неинициализированную переменную:

```
DECLARE
    /* Если переменную ничем не инициализировать, то она имеет значение NULL */
    no_salary_when_fired NUMBER;

BEGIN
    EXECUTE IMMEDIATE
        'update employee
         set     sal = :newsal
         where hire_date is null'
        USING no_salary_when_fired;

END;
```

2. Можно преобразовать NULL в типизированное значение:

```
BEGIN
    EXECUTE IMMEDIATE
        'update employee
         set     sal = :newsal
         where hire_date is null'
        USING to_number(NULL);

END;
```

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:

Метод	Описание
OPEN_CURSOR	Открытие курсора для выполнения динамического оператора
PARSE	Связывание текста динамического оператора с курсором и его синтаксический анализ и разбор
BIND_VARIABLE BIND_ARRAY	Если используются входные аргументы, то связывание их с переменными, содержащими реальные значения
DEFINE_COLUMN DEFINE_COLUMN_LONG DEFINE_ARRAY	Связывание выходных значений с переменными вызывающего блока Указываем переменные, в которые будут попадать выходные значения
EXECUTE	Выполнение оператора
FETCH_ROWS	Извлечение строк
EXECUTE_AND_FETCH	Одновременно выполнение оператора и извлечение строк
VARIABLE_VALUE COLUMN_VALUE COLUMN_VALUE_LONG	Получение значения переменной, извлеченной запросом
CLOSE_CURSOR	Закрытие курсора

Пример использования динамического SQL

Реализуем выполнение следующего SQL-оператора

```
select  ename,  
        job,  
        hiredate,  
        sal  
from    emp  
where   deptno = 20;
```

возвращающего следующий результат

ENAME	JOB	HIREDATE	SAL
SMITH	CLERK	17.12.1980	800,00
JONES	MANAGER	02.04.1981	2975,00
SCOTT	ANALYST	09.12.1982	3000,00
ADAMS	CLERK	12.01.1983	1100,00
FORD	ANALYST	03.12.1981	3000,00

с помощью процедур пакета DBMS_SQL.

Пример использования динамического SQL

```
• DECLARE
•   cur          NUMBER;
•   v_ename      varchar2(10);
•   v_hiredate    date;
•   result       number;
```

Открываем курсор

```
• BEGIN
•   cur := dbms_sql.open_cursor;
```

Задаем SQL-оператор

```
•   dbms_sql.parse(cur,
•       'select ename, hiredate from empl where deptno = :deptno' ,
•       dbms_sql.native);
```

Задаем значение для связанной переменной

```
•   dbms_sql.bind_variable(cur,  ':deptno', 20);
•   dbms_sql.define_column(cur, 1, v_ename, 10);
•   dbms_sql.define_column(cur, 2, v_hiredate);
```

Связываем выходные поля с переменными

```
•   result := dbms_sql.execute(cur);
```

Выполняем SQL-оператор

```
•   LOOP
•       IF dbms_sql.fetch_rows(cur) > 0
•       THEN
•           dbms_sql.column_value(cur, 1, v_ename);
•           dbms_sql.column_value(cur, 2, v_hiredate);
```

Извлекаем строки

```
•           dbms_output.put( lpad(v_ename, 12));
•           dbms_output.put_line(to_char(v_hiredate, ' dd.mm.yyyy '));
```

Считываем значения

```
•       ELSE
•           EXIT;
•       END IF;
•   END LOOP;
```

Закрываем курсор

```
•   dbms_sql.close_cursor(cur);
```

```
•   END;
```


Summary of DBMS_SQL Subprograms

Ниже приведен полный перечень функций и процедур пакета DBMS_SQL:

Функции	
<u>EXECUTE</u>	Executes a given cursor
<u>EXECUTE_AND_FETCH</u>	Executes a given cursor and fetch rows
<u>FETCH_ROWS</u>	Fetches a row from a given cursor
<u>IS_OPEN</u>	Returns TRUE if given cursor is open
<u>LAST_ERROR_POSITION</u>	Returns byte offset in the SQL statement text where the error occurred
<u>LAST_ROW_COUNT</u>	Returns cumulative count of the number of rows fetched
<u>LAST_ROW_ID</u>	Returns ROWID of last row processed
<u>LAST_SQL_FUNCTION_CODE</u>	Returns SQL function code for statement
<u>OPEN_CURSOR</u>	Returns cursor ID number of new cursor
<u>TO_CURSOR_NUMBER</u>	Takes an OPENed strongly or weakly-typed ref cursor and transforms it into a DBMS_SQL cursor number
<u>TO_REFCURSOR</u>	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

Summary of DBMS_SQL Subprograms

Процедуры	
<u>BIND_ARRAY</u>	Binds a given value to a given collection
<u>BIND_VARIABLE</u>	Binds a given value to a given variable
<u>CLOSE_CURSOR</u>	Closes given cursor and frees memory
<u>COLUMN_VALUE</u>	Returns value of the cursor element for a given position in a cursor
<u>COLUMN_VALUE_LONG</u>	Returns a selected part of a LONG column, that has been defined using DEFINE_COLUMN_LONG
<u>DEFINE_ARRAY</u>	Defines a collection to be selected from the given cursor, used only with SELECT statements
<u>DEFINE_COLUMN</u>	Defines a column to be selected from the given cursor, used only with SELECT statements
<u>DEFINE_COLUMN_CHAR</u>	Defines a column of type CHAR to be selected from the given cursor, used only with SELECT statements
<u>DEFINE_COLUMN_LONG</u>	Defines a LONG column to be selected from the given cursor, used only with SELECT statements
<u>DEFINE_COLUMN_RAW</u>	Defines a column of type RAW to be selected from the given cursor, used only with SELECT statements
<u>DEFINE_COLUMN_ROWID</u>	Defines a column of type ROWID to be selected from the given cursor, used only with SELECT statements
<u>DESCRIBE_COLUMNS</u>	Describes the columns for a cursor opened and parsed through DBMS_SQL
<u>DESCRIBE_COLUMNS2</u>	Describes the specified column, an alternative to DESCRIBE_COLUMNS
<u>DESCRIBE_COLUMNS3</u>	Describes the specified column, an alternative to DESCRIBE_COLUMNS
<u>PARSE</u>	Parses given statement
<u>VARIABLE_VALUE</u>	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 понятный, но все же бывают случаи, когда приходится использовать пакет DBMS_SQL:

- Разбор очень длинных строк;
Если строка длиннее 32К, то EXECUTE IMMEDIATE ее не осилит.
- Получение информации о столбцах запроса;
- Минимальный разбор динамических курсоров.
При каждом выполнении EXECUTE IMMEDIATE динамическая строка разбирается заново, 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 процедурам пакета 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:

1. Statement modification
2. Statement injection
3. Data Type Conversion

Рассмотрим более подробно каждую группу, примеры SQL-инъекций, а также способы защиты от них.

Statement modification

Statement modification – изменение динамического SQL-оператора таким образом, что он будет выполняться не так, как планировал разработчик приложения.

Пусть имеется следующая процедура:

```
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;
```

При разработке предполагалось, что функция по имени сотрудника будет возвращать его должность:

```
SQL> begin
2  dbms_output.put_line(sql_injection(p_ename => 'SMITH'));
3 end;
4 /
select job from emp where ename = 'SMITH'
CLERK
PL/SQL procedure successfully completed
SQL>
```

Statement modification

Однако злоумышленник нашел этой функции иное применение. Выполнив вызов, приведенный ниже, он сможет узнать размер зарплаты любого сотрудника компании:

```
SQL> begin
  2  dbms_output.put_line(sql_injection(p_ename => 'AAA' union select to_char(sal) from emp where ename =
    "KING"));
  3 end;
  4 /
select job from scott.emp where ename = 'AAA' union select to_char(sal) from scott.emp where ename = 'KING'
5000
PL/SQL procedure successfully completed
SQL>
```

Statement injection

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

Рассмотрим такую процедуру:

```
CREATE OR REPLACE PROCEDURE stmt_injection_demo(user_name      IN VARCHAR2,
                                                service_type   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 || '');
        DBMS_OUTPUT.PUT_LINE(''service_type: ' || service_type || '');
    END;';
    dbms_output.put_line('PL/SQL Block: ' || v_block);
    EXECUTE IMMEDIATE v_block;
END stmt_injection_demo;
```


Statement injection

Вызов этой процедуры без использования SQL-injection:

```
begin
  stmt_injection_demo(user_name => 'Andy',
                      service_type => 'Waiter');
end;
/
```

Результат ее работы:

PL/SQL Block:

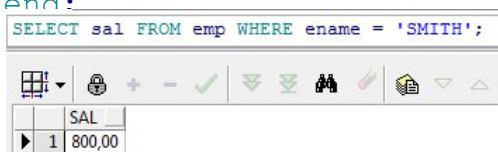
```
BEGIN
  DBMS_OUTPUT.PUT_LINE('user_name: Andy');
  DBMS_OUTPUT.PUT_LINE('service_type: Waiter');
END;
```

```
user_name: Andy
service_type: Waiter
```

Statement injection

А вот так выглядит вызов этой процедуры со **statement injection**:

```
begin
  stmt_injection_demo(user_name => 'Andy');
  update emp set sal = 2500 where ename = upper('SMITH',
    service_type => 'Waiter');
end;
```



	SAL
1	800,00

Результат такого вызова:

PL/SQL Block:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('user_name: Andy');
  update emp set sal = 2500 where ename = upper('SMITH');
  DBMS_OUTPUT.PUT_LINE('service_type: Waiter');
END;
```

user_name: Andy

service type: Waiter



	SAL
1	2500,00

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 возвращает 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:

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

```
RESULT
```

```
-----
MILLER
```

При этом фактически выполнялся такой запрос:

```
select ename from scott.emp where hiredate = '23-JAN-82'
```

Data Type Conversion

А теперь зададим формат даты, как указано ниже, и выполним тот же самый select-оператор:

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT=''' OR empno = "7499";  
Session altered
```

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

```
RESULT
```

```
-----  
ALLEN
```

Результат мы получили уже другой – 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 проверять значения всех переданных параметров на предмет соответствия тому, что именно ожидается.

Вообще следует использовать следующие методы:

- Связывание переменных
- Проверки на соответствие ожидаемым значениям
- Внутреннее преобразование формата

Рассмотрим их ниже более подробно

Связывание переменных

Модифицируем функцию SQL_INJECTION следующим образом:

```
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 using p_ename;
    return v_ret;
end SQL_INJECTION;
```

Теперь попытка выполнения вызова функции SQL_INJECTION с некорректным параметром приведет к ошибке:

```
SQL> begin
  2     dbms_output.put_line(sql_injection(p_ename => 'AAA' union select to_char(sal)
      from emp where ename = 'KING'));
  3 end;
  4 /
select job from scott.emp where ename = :p_ename
begin
    dbms_output.put_line(sql_injection(p_ename => 'AAA' union select to_char(sal) from emp
where ename = 'KING'));
end;
ORA-01403: no data found
ORA-06512: at "SQL_INJECTION", line 7
ORA-06512: at line 2
```

Проверки на соответствие ожидаемым значениям

Всегда следует сначала проверять значения всех переданных параметров.

Например, 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.

Внутреннее преобразование 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.

Внутреннее преобразование формата

Доработаем функцию data_type_conversion – сделаем внутреннее преобразование формата:

```
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   emp
              where  hiredate = date ''' || to_char(p_hiredate, 'YYYY-MM-DD') || ''';
    dbms_output.put_line(v_qry);
    EXECUTE IMMEDIATE v_qry INTO v_ret;
    RETURN v_ret;
END data_type_conversion;
```

Результат:

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

select ename from scott.emp where hiredate = date '1982-01-23'

SQL> ALTER SESSION SET NLS_DATE_FORMAT='''' OR empno = ''7499'';
Session altered

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

select ename from scott.emp where hiredate = date '1982-01-23'
```

Результат функции больше не зависит от NLS-настроек сеанса

Summary

В первой части лекции были освещены вопросы, связанные с выдачей предупреждающих сообщений при компиляции, рассмотрена общая схема обработки исключений, стандартные исключения, создание и обработка собственных исключений, а также рассмотрены несколько примеров.

Во второй части мы поговорили о возможности использования в программах динамического SQL и PL/SQL, рассмотрели работу операторов EXECUTE IMMEDIATE и OPEN FOR, познакомились с пакетом DBMS_SQL, техникой SQL Injection, а также методам защиты от нее.

Список использованных материалов

1. http://docs.oracle.com/cd/E11882_01/appdev.112/e25519/toc.htm
2. С. Фейерштейн, Б. Прибыл. "Oracle PL/SQL для профессионалов"