**Оглавление**

**Исключения**

**Предупреждающие сообщения при компиляции**

**Обработка исключений в PL/SQL**

**Создание собственных исключений**

**Связываем исключение с кодом ошибки**

**Именованные системные исключения**

**Инициирование исключений**

Оператор RAISE

Использование процедуры RAISE\_APPLICATION\_ERROR

**Использование функций обработки ошибок**

**Продолжение работы после возникновения исключения**

**Эскалация необработанного исключения**

**На что стоит обратить внимание**

**Динамический SQL и динамический PL/SQL**

**Инструкции NDS**

**Инструкция EXECUTE IMMEDIATE**

**Инструкция OPEN FOR**

**Режимы использования параметров**

**Дублирование формальных параметров**

**Передача значений NULL**

**Использование пакета DBMS\_SQL**

**Когда следует использовать DBMS\_SQL**

**Новые возможности Oracle 11g**

**SQL Injection**

Statement modification

Statement injection

Data Type Conversion

**Методы защиты от SQL-инъекций**

Использование внутреннего преобразования формата

## 

## 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.  - ***неименованные*** ***исключения*** - имеют только номера (ORA-02292)  - ***именованные*** ***исключения*** – имеют как номера, так и названия (например, ORA-01403: NO\_DATA\_FOUND) | Определяются программистом в 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 о6ра6отчике можно ком6инировать 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 РL/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 ОRА-01843: not a valid month.

Для перехвата этой ошибки в код программы потребуется поместить такой обработчик:

EXCEPTION

WHEN OTHERS THEN

IF SQLCODE = -1843 ТНЕN /\* 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

B 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\_DАТА\_FOUND, то это можно сделать посредством любого из двух операторов:

WHEN NO\_DАТА\_FOUND THEN

WHEN STANDARD.NO\_DАТА\_FOUND THEN

**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** |

## Инициирование исключений

Программно инициировать исключение можно посредством оператора RAISE или процедуры RAISE\_АРРLICATIОN\_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 RAISЕ\_APPLICATION\_ERROR.  
Ее преимущество перед оператором RAISЕ (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);

**keepеrrorstасk** – 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 Огасlе7 поддержка динамического SQL осуществляется с помощью встроенного пакета DВМS\_SQL.  
В Оrасlе 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 DВМS\_SQL.

На практике NDS в подавляющем большинстве является более предпочтительным решением.

## Инструкции NDS

Главным достоинством NDS является его простота.

NDS представлен в языке РL/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 **ЕХЕСUТЕ 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 РL/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 РL/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 РL/SQL (способ передачи параметра дается только в РL/SQL, причем по умолчанию для него установлен режим передачи IN).

Инструкция ЕХЕСUТЕ 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 РL/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 ЕХЕСUТЕ 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;
* ***cтрока\_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](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Executes a given cursor |
| [EXECUTE\_AND\_FETCH](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Executes a given cursor and fetch rows |
| [FETCH\_ROWS](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Fetches a row from a given cursor |
| [IS\_OPEN](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Returns TRUE if given cursor is open |
| [LAST\_ERROR\_POSITION](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Returns byte offset in the SQL statement text where the error occurred |
| [LAST\_ROW\_COUNT](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Returns cumulative count of the number of rows fetched |
| [LAST\_ROW\_ID](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Returns ROWID of last row processed |
| [LAST\_SQL\_FUNCTION\_CODE](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Returns SQL function code for statement |
| [OPEN\_CURSOR](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Returns cursor ID number of new cursor |
| [TO\_CURSOR\_NUMBER](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Takes an OPENed strongly or weakly-typed ref cursor and transforms it into a DBMS\_SQL cursor number |
| [TO\_REFCURSOR](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | 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](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Binds a given value to a given collection |
| [BIND\_VARIABLE](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Binds a given value to a given variable |
| [CLOSE\_CURSOR](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Closes given cursor and frees memory |
| [COLUMN\_VALUE](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Returns value of the cursor element for a given position in a cursor |
| [COLUMN\_VALUE\_LONG](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Returns a selected part of a LONG column, that has been defined using DEFINE\_COLUMN\_LONG |
| [DEFINE\_ARRAY](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Defines a collection to be selected from the given cursor, used only with SELECT statements |
| [DEFINE\_COLUMN](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Defines a column to be selected from the given cursor, used only with SELECT statements |
| [DEFINE\_COLUMN\_CHAR](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Defines a column of type CHAR to be selected from the given cursor, used only with SELECT statements |
| [DEFINE\_COLUMN\_LONG](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Defines a LONG column to be selected from the given cursor, used only with SELECT statements |
| [DEFINE\_COLUMN\_RAW](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Defines a column of type RAW to be selected from the given cursor, used only with SELECT statements |
| [DEFINE\_COLUMN\_ROWID](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Defines a column of type ROWID to be selected from the given cursor, used only with SELECT statements |
| [DESCRIBE\_COLUMNS](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Describes the columns for a cursor opened and parsed through DBMS\_SQL |
| [DESCRIBE\_COLUMNS2](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Describes the specified column, an alternative to DESCRIBE\_COLUMNS |
| [DESCRIBE\_COLUMNS3](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Describes the specified column, an alternative to DESCRIBE\_COLUMNS |
| [PARSE](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | Parses given statement |
| [VARIABLE\_VALUE](http://docs.oracle.com/cd/E11882_01/appdev.112/e40758/d_sql.htm) | 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.