



ИНВЕРСИЯ

PostgresPro

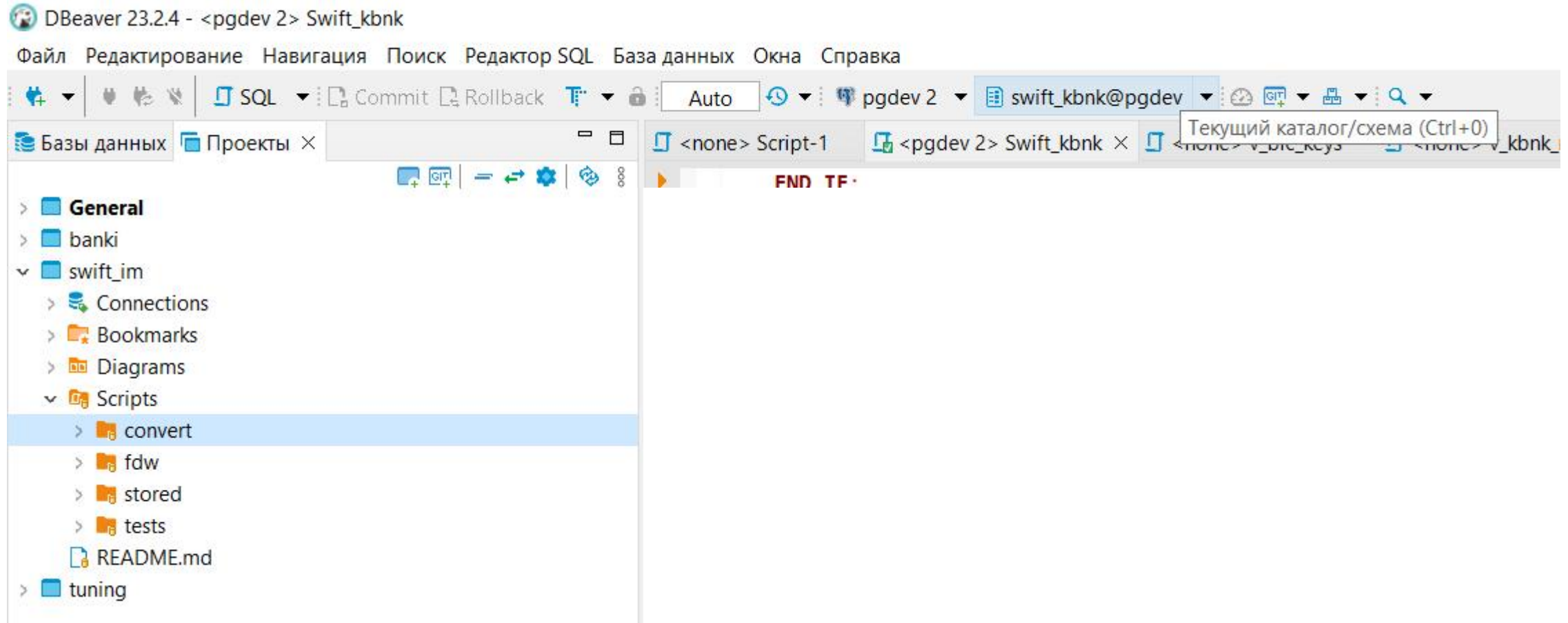
POSTGRES. НАЧАЛО РАБОТЫ, ОБЩИЕ ПРИНЦИПЫ МИГРАЦИИ

Лекция 1

Докладчик:

Буров Александр

Рекомендации по структурированию рабочих файлов



Стиль написания

Вне пакетов создаются только функции и процедуры, нацеленные на наиболее широкое и повсеместное использование.

Функции:

```
FUNCTION func(a IN  
VARCHAR2(2))  
RETURN VARCHAR  
IS  
    variable1 VARCHAR2(5);  
BEGIN  
    RETURN variable1;  
END;
```

```
CREATE FUNCTION func(a IN  
varchar)  
RETURNS varchar  
LANGUAGE PLPGSQL  
AS $$  
DECLARE  
    variable1 varchar;  
BEGIN  
    RETURN variable1;  
END;  
$$
```

Процедуры:

```
procedure proc  
is  
    variable1 number;  
begin  
    variable1 := 1;  
end;
```

```
CREATE PROCEDURE proc()  
LANGUAGE PLPGSQL  
AS $$  
DECLARE  
    variable1 numeric;  
BEGIN  
    variable1 := 1;  
END;  
$$
```

Стиль написания

Вне пакетов создаются только функции и процедуры, нацеленные на наиболее широкое и повсеместное использование.

Функции:

```
FUNCTION func(a IN  
VARCHAR2(2))  
RETURN VARCHAR  
IS  
    variable1 VARCHAR2(5);  
BEGIN  
    RETURN variable1;  
END;
```

```
CREATE FUNCTION func(a IN  
varchar)  
RETURNS varchar  
LANGUAGE PLPGSQL  
AS $$  
DECLARE  
    variable1 varchar;  
BEGIN  
    RETURN variable1;  
END;  
$$
```

Процедуры:

```
procedure proc  
is  
    variable1 number;  
begin  
    variable1 := 1;  
end;
```

```
CREATE PROCEDURE proc()  
LANGUAGE PLPGSQL  
AS $$  
DECLARE  
    variable1 numeric;  
BEGIN  
    variable1 := 1;  
END;  
$$
```

!!! Именованное одноимённых таблиц и представлений в Oracle и Postgres отличаются !!!

Принято решение в Oracle таблицы именовать в low_case, view – UPPER_CASE.

В Postgres наоборот – таблицы UPPER_CASE, представления – low_case.

Подход к написанию и использованию

В PostgreSQL **нельзя**:

- Указывать новые параметры после параметров со значением по умолчанию;
- Создавать функции с выходными параметрами так, как привыкли работать с ними в Oracle.

!!! Если функция возвращает *record*, при `select func().*` функция будет вызвана для каждого поля, возвращаемого *record*'ом **!!!**

Вариант 1: Возврат собственной структуры

```
function varinit(a in varchar2,  
                b out number,  
                c out number)  
  
    return number  
is  
    some_variable number;  
begin  
    b := 11;  
    c := 12;  
    if a = 'H' then  
        return some_variable;  
    else  
        return 1;  
    end if;  
  
    return 2;  
exception  
    when others then  
        b := 144;  
        return 0;  
end;
```

```
CREATE TYPE t_rec AS (b    NUMERIC,  
                      c    NUMERIC,  
                      vret NUMERIC)  
  
CREATE FUNCTION varinit(a IN VARCHAR)  
RETURNS t_rec AS $$  
DECLARE  
    res          t_rec;  
    some_variable NUMERIC;  
BEGIN  
    res.b := 11;  
    res.c := 12;  
    IF a = 'H' THEN  
        res.vret := some_variable;  
        RETURN res;  
    ELSE  
        res.vret := 1;  
        RETURN res;  
    END IF;  
    res.vret := 2;  
    RETURN res;  
EXCEPTION  
    WHEN OTHERS THEN  
        res.b := 144;  
        res.vret := 0;  
        RETURN res;  
END;  
$$
```

ВЫЗОВ

Соответственно, ВЫЗОВ ИЗМЕНИТСЯ:

```
declare
  varr varchar2(1) := 'H';
  res1 number;
  res2 number;
begin
  res := varinit (varr, res1, res2);
end;
```

```
DO $$
DECLARE
  varr varchar := 'H';
  res some_pkg.t_rec;
BEGIN
  res := varinit (varr);
END;
$$
```

Вариант 2: Преобразование функции в процедуру

```
function varinit(a in varchar2,  
                b out number,  
                c out number)  
  
    return number  
is  
    some_variable number;  
begin  
    b := 11;  
    c := 12;  
    if a = 'H' then  
        return some_variable;  
    else  
        return 1;  
    end if;  
  
    return 2;  
exception  
    when others then  
        b := 144;  
        return 0;  
end;
```

```
CREATE PROCEDURE varinit(a    IN varchar,  
                        b    OUT NUMERIC,  
                        c    OUT NUMERIC,  
                        iret OUT NUMERIC)  
  
AS $$  
DECLARE  
    some_variable NUMERIC;  
BEGIN  
    b := 11;  
    c := 12;  
    IF a = 'H' THEN  
        iret := some_variable;  
        RETURN;  
    ELSE  
        iret := 1;  
        RETURN;  
    END IF;  
  
    iret := 2;  
EXCEPTION  
    WHEN OTHERS THEN  
        b := 144;  
        iret := 0;  
END;  
$$
```


ВЫЗОВ

Соответственно, ВЫЗОВ ИЗМЕНИТСЯ:

```
declare
    varr varchar2(1) := 'H';
    res1 number;
    res2 number;
begin
    res := varinit (varr,
res1, res2);
end;
```

```
DO $$
DECLARE
    varr varchar := 'H';
    res1 NUMERIC;
    res2 NUMERIC;
    res3 NUMERIC;
BEGIN
    CALL varinit (varr, res1, res2, res3);
END;
$$
```

Быстродействие

- **Oracle** обычно быстрее выполняет PL/SQL-код благодаря оптимизации на уровне сервера и компиляции кода в родной код.
- **PostgreSQL** может быть медленнее, особенно при использовании языков программирования, отличных от PL/pgSQL. Однако, для многих задач разница в производительности может быть незначительной.

Внутреннее Устройство и Стек Вызовов

- **Oracle** использует PL/SQL Engine, который обеспечивает эффективное выполнение кода на сервере. Вложенные вызовы функций и процедур управляются стеком вызовов.
- **PostgreSQL** использует Executor для выполнения запросов и вызовов функций. Архитектура позволяет гибко управлять вызовами функций с помощью стека вызовов и поддерживает расширяемость за счет дополнительных языков программирования.

Вложенные функции/процедуры

PostgreSQL **НЕ** предоставляет возможность объявить внутри блока declare еще одну функцию/процедуру для локального использования.

```
procedure test_proc_2(cnt out number)
is
    f number := 3;
    c number := 44;

    procedure lp_proc(cnt2 in number) is
    begin
        for i in 1 .. 100 loop
            cnt := cnt + 1;
        end loop;
        cnt := cnt - cnt2;
    end lp_proc;

begin
    lp_proc(f);
    --some code--
    lp_proc(c);
end;
```

```
CREATE PROCEDURE lp_proc(cnt2 IN    NUMERIC,
                        cnt INOUT NUMERIC)
AS $$
BEGIN
    FOR i IN 1 .. 100 loop
        cnt := cnt + 1;
    END LOOP;
    cnt := cnt - cnt2;
END;
$$

CREATE PROCEDURE test_proc_2(cnt out NUMERIC)
AS $$
DECLARE
    f NUMERIC := 3;
    c NUMERIC := 44;
BEGIN
    CALL lp_proc(f, cnt);
    --some code--
    CALL lp_proc(c, cnt);
END;
$$
```

Наименование вложенных функций/процедур

Для того чтобы пометить функцию или процедуру, как используемую только внутри пакета, рекомендуется начинать ее имя с подчеркивания или использовать маску:

parentFuncName\$funcName

```
FUNCTION reg_faktImpl (...)  
IS ...  
    FUNCTION getAcShipMentBill (...) IS  
        BEGIN ...  
    End;  
BEGIN  
    ...  
END reg_faktImpl;
```

CREATE FUNCTION reg_faktimpl(...)

CREATE FUNCTION _getacshipmentbill(...)

ИЛИ

CREATE FUNCTION reg_faktimpl\$getacshipmentbill(...)

Применение функций и процедур. Использование в SQL

- В обеих СУБД функции могут быть использованы в SQL-запросах, что позволяет возвращать результаты в SELECT, WHERE и других SQL-конструкциях.
- Процедуры не могут быть использованы в SQL-запросах напрямую, поскольку они не возвращают значение. Эта особенность чревата определенными последствиями в случаях замены функции на процедуру из-за особенностей языка. Однако, для использования результатов бывшей функции внутри запроса можно, например, ввести переменную и через out параметр процедуры записать в нее нужный результат.

Применение функций и процедур. Архитектурные особенности

- **Oracle** имеет более оптимизированный механизм выполнения PL/SQL кода на уровне сервера, что может влиять на производительность.
- **PostgreSQL** предлагает большую гибкость за счет поддержки различных языков программирования для написания функций и процедур.
- В **PostgreSQL** процедуры быстрее на больших количествах данных.

Процедуры и функции

Функции следует использовать, когда нужно вернуть значение и этот результат может быть использован в SQL-запросах. Например, для вычислений, преобразования данных или получения агрегированных результатов.

Процедуры лучше использовать, когда нужно выполнить сложные операции, которые включают множество шагов или транзакций, и когда нет необходимости возвращать значение. Процедуры удобны для операций обновления, вставки или удаления данных.

Вопросы для самопроверки

Какую проблему создает наличие локальных функций/процедур внутри функций в Oracle при переносе в PostgreSQL, и какое решение предлагается для ее обхода?

- a) Локальные функции/процедуры не могут быть перенесены в PostgreSQL.
- b) В PostgreSQL локальные функции/процедуры автоматически преобразуются в глобальные.
- c) Необходимо вынести локальную функцию/процедуру в отдельную, глобальную процедуру и использовать параметры INOUT для передачи изменяемых переменных.
- d) Необходимо переписать всю функцию, используя только SQL-запросы.

Вопросы для самопроверки

Какую проблему создает наличие локальных функций/процедур внутри функций в Oracle при переносе в PostgreSQL, и какое решение предлагается для ее обхода?

- a) Локальные функции/процедуры не могут быть перенесены в PostgreSQL.
- b) В PostgreSQL локальные функции/процедуры автоматически преобразуются в глобальные.
- c) **Необходимо вынести локальную функцию/процедуру в отдельную, глобальную процедуру и использовать параметры INOUT для передачи изменяемых переменных.**
- d) Необходимо переписать всю функцию, используя только SQL-запросы.

Вопросы для самопроверки

В чем состоит основное отличие в подходе к параметрам функций между Oracle и PostgreSQL, которое необходимо учитывать при переносе кода?

- a) В Oracle порядок входных параметров не имеет значения, а в PostgreSQL имеет значение.
- b) В PostgreSQL порядок входных параметров не имеет значения, а в Oracle имеет значение.
- c) Оба языка требуют строгого соблюдения порядка входных параметров.
- d) Оба языка позволяют игнорировать порядок входных параметров.

Вопросы для самопроверки

В чем состоит основное отличие в подходе к параметрам функций между Oracle и PostgreSQL, которое необходимо учитывать при переносе кода?

- a) **В Oracle порядок входных параметров не имеет значения, а в PostgreSQL имеет значение.**
- b) В PostgreSQL порядок входных параметров не имеет значения, а в Oracle имеет значение.
- c) Оба языка требуют строгого соблюдения порядка входных параметров.
- d) Оба языка позволяют игнорировать порядок входных параметров.

Вопросы к первому блоку

Работа с параметрами

```
CREATE FUNCTION sample_function(a IN INT,  
                                b INOUT INT,  
                                c OUT INT)  
  
RETURNS RECORD  
AS $$  
  
BEGIN  
    c := a + b;  
    b := b * 2;  
END;  
$$ LANGUAGE plpgsql;
```

Работа с параметрами

```
CREATE FUNCTION sample_function(a IN INT,  
                                b INOUT INT,  
                                c OUT INT)  
RETURNS RECORD --опционально  
AS $$  
  
BEGIN  
    c := a + b;  
    b := b * 2;  
END;  
$$ LANGUAGE plpgsql;
```

Если функция возвращает один параметр, то также можно не писать RETURNS...

Работа с параметрами

```
CREATE FUNCTION sample_function(a IN INT,  
                                b INOUT INT,  
                                c OUT INT)  
  
RETURNS RECORD --опционально  
AS $$  
  
BEGIN  
    c := a + b;  
    b := b * 2;  
END;  
$$ LANGUAGE plpgsql;
```

Если функция возвращает один параметр, то также можно не писать RETURNS...

!!! Невозможно создать функцию, в которой есть OUT параметры и возвращаемые через return значения **!!!**

Варианты написания параметров

```
CREATE FUNCTION func(a varchar)
RETURNS varchar AS $$
BEGIN
    RETURN a;
END;
$$
```

```
----в стиле Oracle
CREATE FUNCTION func(a IN varchar)
RETURNS varchar AS $$
BEGIN
    RETURN a;
END;
$$
```

```
CREATE FUNCTION func(IN a varchar)
RETURNS varchar AS $$
BEGIN
    RETURN a;
END;
$$
```

IN/OUT параметры

Нельзя указывать входные параметры без значения по умолчанию после параметров с объявленными значениями по умолчанию, а OUT параметры нельзя указывать после входящих параметров со значением по умолчанию.

```
create procedure proc(a IN varchar2,  
                     b IN number DEFAULT 2,  
                     c OUT number)  
  
is  
BEGIN  
    IF a = 'b' THEN  
        c := b;  
    ELSE  
        c := b + 1;  
    END IF;  
END;
```

```
CREATE PROCEDURE proc (a IN varchar2,  
                       c OUT NUMERIC,  
                       b IN NUMERIC DEFAULT 2)  
  
LANGUAGE PLPGSQL  
AS $$  
BEGIN  
    IF a = 'b' THEN  
        c := b;  
    ELSE  
        c := b + 1;  
    END IF;  
END;  
$$
```

Передача переменной record, предопределенной через таблицу.

```
FUNCTION test_func(rec IN acc%rowtype)
RETURN varchar
is
a number := 1;
BEGIN
    a := a + 1;
    return 'b';
END;
```

```
CREATE FUNCTION test_func(rec IN acc)
RETURNS varchar
LANGUAGE PLPGSQL
AS $$
DECLARE
a NUMERIC := 1;
BEGIN
    a := a + 1;
    return 'b'
END;
$$
```

!!! Неявное приведение типов для in параметров возможно, а для out параметров нет !!!

Неявное приведение типов и иерархии типов, присваивание

Тип Oracle	Тип Postgres	Комментарий
NUMBER(<=4)	numeric(N)	
NUMBER(<=9)	numeric(N)	
NUMBER(<=18)	numeric(N)	
NUMBER	numeric(N)/numeric(N,M)	Указывать numeric без размерности нельзя! (Внутреннее правило компании)
NUMBER(X,X)	numeric(N,M)	
INTEGER	numeric(38)	
VARCHAR2	varchar	
CHAR(X)	char(X)	
DATE	date	Без времени
DATE	timestamp	Со временем (вариант, в который конвертеры преобразуют по умолчанию)
TIMESTAMP	timestamp	
CLOB	text	
BLOB	bytea	
RAW	bytea	
	xml	

Строковые преобразования

!!! При переносе все строковые типы преобразовывать в varchar, кроме:

- clob -> text
- char -> char

Неявное приведение типов

Oracle:

- В PL/SQL приведение типов может происходить **неявно**, когда это возможно.

- Пример:

```
DECLARE
```

```
  num NUMBER;
```

```
BEGIN
```

```
  num := '123'; -- Строка будет неявно преобразована в число
```

```
END;
```

- Если строку невозможно преобразовать в число, PL/SQL выбросит ошибку времени выполнения.
- **Строгость приведения типов:** преобразование DATE в NUMBER не будет работать неявно.

Неявное приведение типов

PostgreSQL:

- PostgreSQL также поддерживает **неявное приведение типов**, но его возможности ограничены.
- Пример:

```
DO $$  
DECLARE  
  num NUMERIC;  
BEGIN  
  num := '123'; -- Строка будет неявно преобразована в число  
END;  
$$
```

- PostgreSQL в некоторых случаях требует **явного приведения** типов для операций с различными типами данных.
- **Типизация и строгая проверка:** PostgreSQL значительно более строг в отношении типизации.

```
SELECT '123'::int; -- Явное преобразование строки в число.
```

Иерархия типов

Oracle (PL/SQL)

- В Oracle типы данных имеют определённую **иерархию** и совместимость между собой.
- **Скалярные типы:** Например, такие типы как NUMBER, VARCHAR2, DATE имеют четкую иерархию, и их преобразование в более общий тип может происходить автоматически.
- **Объектные типы и коллекции:** В PL/SQL поддерживаются пользовательские объектные типы, наследование и переопределение методов, что добавляет гибкость в работе с типами данных.

PostgreSQL:

- В базе данных PostgreSQL есть множество типов данных, таких как INTEGER, TEXT, TIMESTAMP, но их иерархия реализована через приведение и расширяемость типов.
- В PostgreSQL поддерживаются **пользовательские типы** и расширения, что позволяет создавать новые типы данных. Но нет наследования типов, как в Oracle.

Пакеты и схемы

```
CREATE OR REPLACE PACKAGE example_pkg
IS
    -- объявление переменных, типов и пр. глобальной видимости
    TYPE t_test_pkg IS RECORD (id number,
                               vret varchar2(15));
    --объявление спес 'а функций и процедур глобальной видимости
END example_pkg;

CREATE OR REPLACE PACKAGE BODY example_pkg
IS

    vtext varchar2(25) := 'a';
    vnum number;
    --
    function get_vtext return varchar2
    is
    begin
        return vtext;
    end get_vtext;
    --
    procedure set_vnum
    is
    begin
        if get_vtext = 'a' then
            vnum := 5;
        else
            vnum := 0;
        end if;
    end set_vnum;
    --
BEGIN
    dbms_output.put_line('package initialised');
END example_pkg;
```

```
CREATE OR REPLACE PACKAGE example_pkg

CREATE TYPE t_test_pkg AS (id NUMERIC,
                           vret VARCHAR)

CREATE OR REPLACE FUNCTION __init__()
RETURNS void AS $$ -- package initialization
-- здесь выполняется импорт других пакетов
-- это необходимо, если внутри создаваемого пакета используются переменные из
другого
    #IMPORT example_pkg2
DECLARE
    --объявление глобальных переменных пакета
    vtext VARCHAR := 'a';
    vnum NUMERIC;
BEGIN
    --код, исполняемый при инициализации пакета
    RAISE NOTICE 'package test_pkg intialised';
END;
$$
----
CREATE FUNCTION get_vtext()
RETURNS VARCHAR AS $$
BEGIN
    RETURN vtext;
END;
$$
----
CREATE PROCEDURE set_vnum() AS $$
BEGIN
    IF example_pkg.get_vtext() = 'a' THEN
        vnum := 5;
    ELSE
        vnum := 0;
    END;
END;
$$
```

Пакеты и схемы

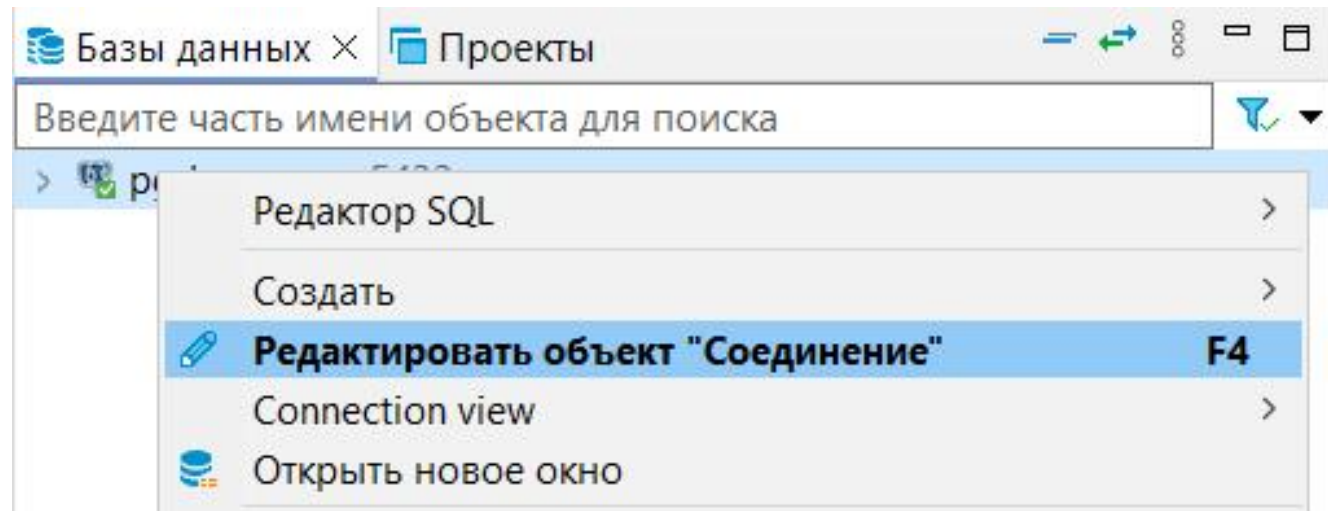
!!! В пакетах PostgreSQL не поддерживается тип `enum`. Также в пакетах не поддерживаются `domain` и для использования их следует создавать схему. Примером является Ts.

Пакеты и схемы

!!! В пакетах PostgreSQL не поддерживается тип `enum`. Также в пакетах не поддерживаются `domain` и для использования их следует создавать схему. Примером является `Ts`.

!!! Использование `raise_notice` в нашем коде запрещено. Альтернатива – `raise_debug`.

Настройка raise_debug



Настройка raise_debug

Конфигурация соединения "pgdev"

Инициализация

Настройки инициализации соединения

Настройки соединения

- Инициализация**
- Команды ОС
- Идентификация клиента
- Transactions
- Общее
- Метаданные
- Обработка ошибок
- Data Transfer
- Редактор данных
- Редактор SQL

Соединение

Авто-фиксация транзакций: ☒

Уровень изоляции:

База по умолчанию:

Схема по умолчанию:

Авто поддержка связи (в секундах):

☒ Close idle connection after (seconds)

See "[Типы соединений](#)" for transactions settings related to the connection types.
Data source settings have an advantage over other settings.

Начальная загрузка SQL: [Конфигурация ..](#)

Чтобы просмотреть уровни изоляции транз. SQL запросы выполняемые после соединения нажмите "Тест соединения"

[Read more in the documentation.](#)

Настройка raise_debug

Выполнить SQL при после соединения

SQL запросы

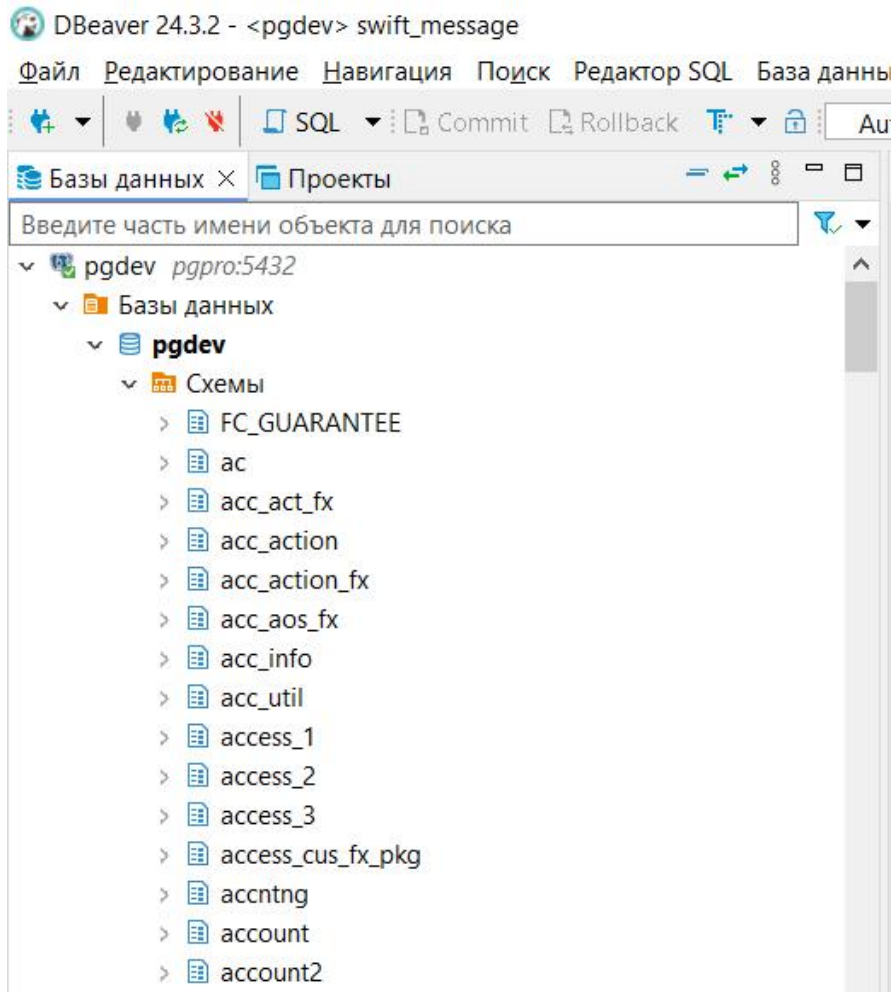
set client_min_messages='debug1'	Добавить
	Удалить

☐ Игнорировать ошибки SQL

[Вы можете использовать переменные в командах. Кликните чтобы посмотреть список.](#)

OK Отмена

Пакеты и схемы



- > xtpack
- ▼ xxi
 - > Таблицы
 - > Внешние таблицы
 - > Представления
 - > Мат. представления
 - > Индексы
 - > Функции
 - > Последовательности
 - > Типы данных
 - > Агрегатные функции
- > xxi_context
- > xxi_dbms_alert
- > xxi_dbms_pipe
- > xxi_dev_test
- > xxi_service
- > zsk_pkg
- > Событийные триггеры
- > Расширения
- > Хранилище
- > Системные объекты
- > Роли

Пакеты и схемы

xxi	
Таблицы	
A2R	136K
AAC	88K
ACA	136K
ACB	56K
ACC	474M
ACC40911	64K
ACC_DEFER_BACK_TRN	144K
ACC_FLT_LST	80K
ACC_GACC	874M
ACC_GACC_SPOOL	72K
ACC_GACC_TMP	24K
ACC_GRP	72K
ACC_OKACCEPT	96K

xxi	
Таблицы	
Внешние таблицы	
Представления	
V_TRN_PART_CURRENT	
a2r	
a2rtype	
aac	
abb_ml2	
ac_code	
ac_parameter	
ac_result	
aca	
acb	
acb_mf	
acc	

xxi	
Таблицы	
Внешние таблицы	
Представления	
Мат. представления	
Индексы	
pl_service.i_pl_service_proc	
pl_service.pk_pl_service	
pl_service.uk_pl_serv_idproc	
ora2pg_stress_cus_result_hist.ora2pg_stress_cus_re	
ora2pg_stress_cus_result_hist.ora2pg_stress_cus_re	
ple.i_ple_agr_typ	
ple.i_ple_agrid_part_type	
ple.i_ple_atm	
ple.i_ple_date	
ple.i_ple_ioprecid	

xxi	
Таблицы	
Внешние таблицы	
Представления	
Мат. представления	
Индексы	
Функции	
f_init_0	
f_t_a_idu_dj_docd()	
f_t_a_idu_dj_md1()	
f_t_a_idu_dj_md2()	
f_t_a_idu_dj_md3()	
f_t_a_idu_dj_mp1()	
f_t_a_idu_dj_mp2()	
f_t_a_idu_dj_mp3()	
f_t_a_idu_dj_mp4()	

Вопросы для самопроверки

```
FUNCTION calculate_sum (num1 VARCHAR2, num2 NUMBER) RETURN NUMBER IS
    result NUMBER;
BEGIN
    result := num1 + num2; -- Неявное преобразование VARCHAR2 в NUMBER
    RETURN result;
END;
```

a)

```
FUNCTION calculate_sum (num1 VARCHAR, num2 INTEGER)
    RETURNS INTEGER AS $$
BEGIN
    RETURN CAST(num1 AS INTEGER) + num2;
END;
$$ LANGUAGE plpgsql;
```

b)

```
FUNCTION calculate_sum (num1 VARCHAR, num2 INTEGER)
    RETURNS INTEGER AS $$
BEGIN
    RETURN num1 + num2;
END;
$$ LANGUAGE plpgsql;
```

c)

```
FUNCTION calculate_sum (num1 VARCHAR, num2 INTEGER)
    RETURNS INTEGER AS $$
BEGIN
    RETURN num1::INTEGER + num2;
END;
$$ LANGUAGE plpgsql;
```

Вопросы для самопроверки

```
FUNCTION calculate_sum (num1 VARCHAR2, num2 NUMBER) RETURN NUMBER IS
    result NUMBER;
BEGIN
    result := num1 + num2; -- Неявное преобразование VARCHAR2 в NUMBER
    RETURN result;
END;
```

a)

```
FUNCTION calculate_sum (num1 VARCHAR, num2 INTEGER)
    RETURNS INTEGER AS $$
BEGIN
    RETURN CAST(num1 AS INTEGER) + num2;
END;
$$ LANGUAGE plpgsql;
```

c)

```
FUNCTION calculate_sum (num1 VARCHAR, num2 INTEGER)
    RETURNS INTEGER AS $$
BEGIN
    RETURN num1::INTEGER + num2;
END;
$$ LANGUAGE plpgsql;
```

Вопросы для самопроверки

```
CREATE OR REPLACE FUNCTION get_data(user_id INT, OUT username VARCHAR)
  RETURNS VARCHAR AS $$
BEGIN
  SELECT u.username INTO username FROM users u WHERE u.id = user_id;
  RETURN username;
END;
$$ LANGUAGE plpgsql;
```

- a) Функция успешно создастся и будет работать корректно, возвращая имя пользователя.
- b) Функция успешно создастся, но при вызове будет возвращать NULL.
- c) PostgreSQL выдаст ошибку, поскольку нельзя одновременно использовать OUT параметры и RETURNS отличный от record.
- d) Функция успешно создастся, но параметр username будет игнорироваться.

Вопросы для самопроверки

```
CREATE OR REPLACE FUNCTION get_data(user_id INT, OUT username VARCHAR)
  RETURNS VARCHAR AS $$
BEGIN
  SELECT u.username INTO username FROM users u WHERE u.id = user_id;
  RETURN username;
END;
$$ LANGUAGE plpgsql;
```

- a) Функция успешно создастся и будет работать корректно, возвращая имя пользователя.
- b) Функция успешно создастся, но при вызове будет возвращать NULL.
- c) **PostgreSQL выдаст ошибку, поскольку нельзя одновременно использовать OUT параметры и RETURNS отличный от record.**
- d) Функция успешно создастся, но параметр username будет игнорироваться.

Вопросы ко второму блоку

Управление транзакцией

- В PostgreSQL транзакционность (возможность commit и rollback) доступна только в процедурах.
 - Для вызова процедур с commit из клиентской части необходим autocommit.

<pre>begin ... -- Здесь производится DML или DDL операция, которая требует commit или rollback exception; ... end;</pre>	<pre>BEGIN ... -- Здесь производится DML или DDL операция, которая требует commit или rollback commit; -- Так нельзя! exception ... END;</pre>
--	--

Текущее использование, Oracle

```
FOR Obj IN SELECT....  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
    COMMIT;  
  EXCEPTION  
    WHEN OTHERS THEN  
      ok := false;  
  END;  
  
  IF ok THEN  
    ...  
  ELSE  
    ROLLBACK;  
  END IF;  
END LOOP;
```

Текущее использование, Oracle

```
FOR Obj IN SELECT....  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
    COMMIT;  
  EXCEPTION  
    WHEN OTHERS THEN  
      ok := false;  
  END;  
  
  IF ok THEN  
    ...  
  ELSE  
    ROLLBACK;  
  END IF;  
END LOOP;
```


Текущее использование, Oracle

```
FOR Obj IN SELECT....  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
    COMMIT;  
  EXCEPTION  
    WHEN OTHERS THEN  
      ok := false;  
  END;  
  
  IF ok THEN  
    ...  
  ELSE  
    ROLLBACK;  
  END IF;  
END LOOP;
```

Текущее использование, Oracle

```
FOR Obj IN SELECT....  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
    COMMIT;  
    EXCEPTION  
      WHEN OTHERS THEN  
        ok := false;  
    END;  
  
    IF ok THEN  
      ...  
    ELSE  
      ROLLBACK;  
    END IF;  
  END LOOP;
```

Текущее использование, Oracle

```
FOR Obj IN SELECT....
LOOP
  BEGIN
    ok := f(Obj)
    IF NOT ok THEN
      RAISE ;
    END IF;
    COMMIT;
  EXCEPTION
    WHEN OTHERS THEN
      ok := false;
  END;

  IF ok THEN
    ...
  ELSE
    ROLLBACK;
  END IF;
END LOOP;
```

Проблемы миграции

```
FOR Obj IN SELECT....  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
    COMMIT;  
  EXCEPTION  
    WHEN OTHERS THEN  
      ok := false;  
  END;  
  
  IF ok THEN  
    ...  
  ELSE  
    ROLLBACK;  
  END IF;  
END LOOP;
```

Проблемы:

1) COMMIT внутри BEGIN-EXCEPTION-END

Проблемы миграции

```
FOR Obj IN SELECT....  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
    COMMIT;  
  EXCEPTION  
    WHEN OTHERS THEN  
      ok := false;  
  END;  
  
  IF ok THEN  
    ...  
  ELSE  
    ROLLBACK;  
  END IF;  
END LOOP;
```

Проблемы:

1) COMMIT внутри BEGIN-EXCEPTION-END

Проблемы миграции

```
FOR Obj IN SELECT....  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
    COMMIT;  
  EXCEPTION  
    WHEN OTHERS THEN  
      ok := false;  
  END;  
  
  IF ok THEN  
    ...  
  ELSE  
    ROLLBACK;  
  END IF;  
END LOOP;
```

Проблемы:

1) COMMIT внутри BEGIN-EXCEPTION-END

Проблемы миграции

```
FOR Obj IN SELECT....  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
    COMMIT;  
  EXCEPTION  
    WHEN OTHERS THEN  
      ok := false;  
  END;  
  
  IF ok THEN  
    ...  
  ELSE  
    ROLLBACK;  
  END IF;  
END LOOP;
```

Проблемы:

1) COMMIT внутри BEGIN-EXCEPTION-END

Проблемы миграции

```
FOR Obj IN SELECT...  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
  EXCEPTION  
    WHEN OTHERS THEN  
      ok := false;  
  END;  
  COMMIT;  
  
  IF ok THEN  
    ...  
  ELSE  
    ROLLBACK;  
  END IF;  
END LOOP;
```

Проблемы:

1) COMMIT внутри BEGIN-EXCEPTION-END

Проблемы миграции

```
FOR Obj IN SELECT....  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
  EXCEPTION  
    WHEN OTHERS THEN  
      ok := false;  
  END;  
  COMMIT;  
  
  IF ok THEN  
    ...  
  ELSE  
    ROLLBACK;  
  END IF;  
END LOOP;
```

Проблемы:

1) COMMIT внутри BEGIN-EXCEPTION-END

Решение:

Автономные транзакции

Проблемы миграции

```
FOR Obj IN SELECT...  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
  EXCEPTION  
    WHEN OTHERS THEN  
      ok := false;  
  END;  
  COMMIT;  
  
  IF ok THEN  
    ...  
  ELSE  
    ROLLBACK;  
  END IF;  
END LOOP;
```

Проблемы:

1) COMMIT внутри BEGIN-EXCEPTION-END

Решение:

Автономные транзакции

2) Курсор все время открыт

Проблемы миграции

```
FOR Obj IN SELECT...  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
  EXCEPTION  
    WHEN OTHERS THEN  
      ok := false;  
  END;  
  COMMIT;  
  
  IF ok THEN  
    ...  
  ELSE  
    ROLLBACK;  
  END IF;  
END LOOP;
```

Проблемы:

1) COMMIT внутри BEGIN-EXCEPTION-END

Решение:

Автономные транзакции

2) Курсор все время открыт

Варианты решения:

- **HOLD Cursor (Материализация при commit)?**

Проблемы миграции

```
FOR Obj IN SELECT...  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
  EXCEPTION  
    WHEN OTHERS THEN  
      ok := false;  
  END;  
  COMMIT;  
  
  IF ok THEN  
    ...  
  ELSE  
    ROLLBACK;  
  END IF;  
END LOOP;
```

Проблемы:

1) COMMIT внутри BEGIN-EXCEPTION-END

Решение:

Автономные транзакции

2) Курсор все время открыт

Варианты решения:

- ~~HOLD Cursor (Материализация при commit)~~

Проблемы миграции

```
FOR Obj IN SELECT...  
LOOP  
  BEGIN  
    ok := f(Obj)  
    IF NOT ok THEN  
      RAISE ;  
    END IF;  
  EXCEPTION  
    WHEN OTHERS THEN  
      ok := false;  
  END;  
  COMMIT;  
  
  IF ok THEN  
    ...  
  ELSE  
    ROLLBACK;  
  END IF;  
END LOOP;
```

Проблемы:

1) COMMIT внутри BEGIN-EXCEPTION-END

Решение:

Автономные транзакции

2) Курсор все время открыт


Варианты решения:

- ~~HOLD Cursor (Материализация при commit)~~
- **Чтение порциями (батчи)**

Чтение порциями (батчи)

```
DECLARE
    ...
BEGIN
    CALL ...
    <<l_Main>>
    LOOP
        ...
        FOREACH ID_Mail IN ARRAY
        TabID_Mail
        LOOP
            BEGIN AUTONOMOUS
                ...
            EXCEPTION
                ...
            END;
            ...
        END LOOP;
    END LOOP l_Main;

    CALL UTIL.LO_End (iTotal);
END;
```



```
CALL UTIL.LO_Start ( cOpName => ML2.GetText (
    'UF_REQ.DeleteMarked#OpName', 'Удаление УФЭБС сообщений'),
    cTargetDesc => ML2.GetText (
    'UF_REQ.DeleteMarked#TargetDesc', 'Обработано сообщений') );


<<l_Main>>
LOOP
    TabID_Mail := UTIL.Mrk_Next1 (ID_Marker, ID_Mail);
    EXIT WHEN TabID_Mail IS NULL;

    FOREACH ID_Mail IN ARRAY TabID_Mail
    LOOP
```

Чтение порциями (батчи)

```
DECLARE
...
BEGIN
  CALL ...
  <<l_Main>>
  LOOP
    ...
    FOREACH ID_Mail IN ARRAY
      TabID_Mail
    LOOP
      BEGIN AUTONOMOUS
        ...
      EXCEPTION
        ...
      END;
    ...
  END LOOP;
END LOOP l_Main;

CALL UTIL.LO_End (iTotal);
END;
```



```
CALL UTIL.LO_Start ( cOpName => ML2.GetText (
  'UF_REQ.DeleteMarked#OpName', 'Удаление УФЭБС сообщений'),
  cTargetDesc => ML2.GetText (
    'UF_REQ.DeleteMarked#TargetDesc', 'Обработано сообщений') );

<<l_Main>>
LOOP
  TabID_Mail := UTIL.Mrk_Next1 (ID_Marker, ID_Mail);
  EXIT WHEN TabID_Mail IS NULL;

  FOREACH ID_Mail IN ARRAY TabID_Mail
  LOOP
```

Чтение порциями (батчи)

```
DECLARE
...
BEGIN
  CALL ...
  <<l_Main>>
  LOOP
    ...
    FOREACH ID_Mail IN ARRAY
      TabID_Mail
      LOOP
        BEGIN AUTONOMOUS
          ...
          EXCEPTION
            ...
          END;
          ...
        END LOOP;
      END LOOP l_Main;

      CALL UTIL.LO_End (iTotal);
    END;
```

BEGIN AUTONOMOUS


```
rc := UF_REQ.DeleteMail ( ID_Mail => ID_Mail,
                          cMessage => cMessage );

IF rc.bRetCode THEN
  CALL UTIL.Mrk_Update2 (ID_Mail, NULL, ID_Marker, ID_Success);
  CALL MBUNCH.Kill (cBomProc, TO_CHAR (ID_Mail));
END IF;
```


Чтение порциями (батчи)

```
DECLARE
...
BEGIN
  CALL ...
  <<l_Main>>
  LOOP
    ...
    FOREACH ID_Mail IN ARRAY
      TabID_Mail
      LOOP
        BEGIN AUTONOMOUS
          EXCEPTION
          ...
        END;
        ...
      END LOOP;
    END LOOP l_Main;

    CALL UTIL.LO_End (iTotal);
  END;
```

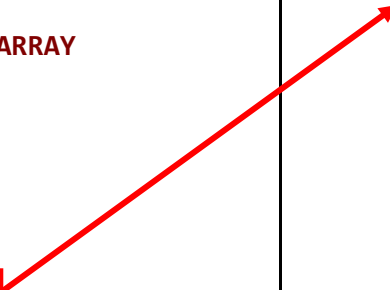


```
EXCEPTION
  WHEN OTHERS THEN
    DECLARE
      ex TS.T_StackedDiagnostics;
    BEGIN
      GET STACKED DIAGNOSTICS
        ex.RETURNED_SQLSTATE = RETURNED_SQLSTATE,
        ex.MESSAGE_TEXT = MESSAGE_TEXT,
        ex.PG_EXCEPTION_DETAIL = PG_EXCEPTION_DETAIL,
        ex.PG_EXCEPTION_HINT = PG_EXCEPTION_HINT,
        ex.PG_EXCEPTION_CONTEXT = PG_EXCEPTION_CONTEXT;
      rc := ROW (FALSE, TS.WhenOthersError ('UF_REQ.DeleteMarked', ex));
    END;
  END;
```

Чтение порциями (батчи)

```
DECLARE
...
BEGIN
  CALL ...
  <<l_Main>>
  LOOP
    ...
    FOREACH ID_Mail IN ARRAY
      TabID_Mail
      LOOP
        BEGIN AUTONOMOUS
          ...
          EXCEPTION
            ...
            END;
          ...
        END LOOP;
      END LOOP l_Main;

      CALL UTIL.LO_End (iTotal);
    END;
```



```
IF rc.bRetCode THEN
  iCntOk := iCntOk + 1;
ELSE
  CALL MBUNCH.Put_AT (cBomProc, TO_CHAR (ID_Mail), rc.cErrorMsg);
END IF;

iCurrent := iCurrent + 1;
IF UTIL.LO_Cancel (iCurrent, iTotal) THEN
  EXIT l_Main;
END IF;
```

Автономные транзакции

Если всё-таки commit и rollback нужен, то можно пользоваться автономной транзакцией:

```
begin autonomous
...
exception
...
end;
```

```
begin autonomous
...
exception
... -- Производим новую DML/DDI операцию, например,
вызываем функцию логирования в какую-нибудь таблицу
    commit;
end;
```

Важно, что для автономной транзакции в блоке обработки ошибок rollback уже произведён. Если необходимо внутри блока exception совершать новые DML/DDI операции, то их необходимо коммитить отдельно: здесь уже можно писать слово commit.

Фиксация транзакции по условию

Если внутри автономной транзакции надо производить commit / rollback по условию, то rollback лучше превращать в исключение с его последующим перехватом.

```
begin
...
if ... then
    log_something (...) ;
    rollback;
else
    commit;
end;
exception
...
end;
```

```
begin autonomous
...
    if ... then
        raise exception ...;
    end if;
exception
    when raise exception then
        log_something (...) ;
end;
```

Обработка ошибок. Общий принцип

В Oracle уровень изоляции по умолчанию выше, чем в PostgreSQL.

```
declare
tmess varchar2(100);
begin
    tmess := 'some error';
    if tmess is not null then
        raise_application_error(-
20077, tmess);
    end if;
end;
```

```
DO $$
DECLARE
tmess varchar;
BEGIN
    tmess := 'some error';
    IF tmess IS NOT NULL THEN
        RAISE SQLSTATE '50078'
        USING HINT = -20077,
        MESSAGE = tmess;
    END IF;
END;
$$
```

Обработка ошибок. Объявленные ошибки

```
declare
    v_id number := 142;
    name VARCHAR2(20) := 'Mike';
    tmsg VARCHAR2(200);
    bb exception;
begin
    some_proc(v_id, name, tmsg);
    if tmsg is not null then
        raise bb;
    end if;
exception
    when bb then
        tracepkg.txtout('error');
end;
```

```
DO $$
DECLARE
    v_id NUMERIC := 142;
    name VARCHAR := 'Mike';
    tmsg VARCHAR;
BEGIN
    CALL some_proc(v_id, name, tmsg);
    IF tmsg IS NOT NULL THEN
        RAISE SQLSTATE '57778';
    END IF;
EXCEPTION
    WHEN SQLSTATE '57778' THEN
        CALL tracepkg.txtout('error');
END;
$$
```

Обработка ошибок. Предопределенные исключения

```
declare
    v_id    number := 142;
    v_name  varchar2(20) := 'Mike';
begin
    insert into tab_users (v_id, v_name)
    values (v_id, v_name);
exception
    when dup_val_on_index THEN
        tracepkg.txtout('error');
end;
```

```
DO $$
DECLARE
    v_id    NUMERIC := 142;
    v_name  varchar(20) := 'Mike';
BEGIN
    INSERT INTO tab_users (v_id, v_name)
    VALUES (v_id, v_name);
EXCEPTION
    WHEN unique_violation THEN
        CALL tracepkg.txtout('error');
END;
$$
```

Неявный откат изменений после возникновения исключения

В PL/pgSQL при перехвате исключения в секции EXCEPTION все изменения в базе данных с начала блока автоматически откатываются.

```
BEGIN
    SAVEPOINT s1;
    ... здесь код ...
EXCEPTION
    ... THEN
        ROLLBACK TO s1;
    ... здесь код ...
WHEN ... THEN
    ROLLBACK TO s1;
    ... здесь код ...
END;
```


Неявный откат изменений после возникновения исключения

В PL/pgSQL при перехвате исключения в секции EXCEPTION все изменения в базе данных с начала блока автоматически откатываются.

```
BEGIN
    SAVEPOINT s1;
    ... здесь код ...
EXCEPTION
    ... THEN
        ROLLBACK TO s1;
    ... здесь код ...
WHEN ... THEN
    ROLLBACK TO s1;
    ... здесь код ...
END;
```

Для PG достаточно убрать операторы SAVEPOINT и ROLLBACK TO

Подробная информация об ошибке

```
DO $$
BEGIN
    -- код, в котором может возникнуть исключение
EXCEPTION
    WHEN OTHERS THEN
        DECLARE
            ex TS.T_StackedDiagnostics;
        BEGIN
            GET STACKED DIAGNOSTICS
                ex.RETURNED_SQLSTATE = RETURNED_SQLSTATE, -- код ошибки
                ex.MESSAGE_TEXT = MESSAGE_TEXT, -- текст ошибки
                ex.PG_EXCEPTION_DETAIL = PG_EXCEPTION_DETAIL, -- контекст исключения
                ex.PG_EXCEPTION_HINT = PG_EXCEPTION_HINT, -- подробный текст ошибки
                ex.PG_EXCEPTION_CONTEXT = PG_EXCEPTION_CONTEXT; -- текст подсказки к исключению

            EODG.Error_Message := ts.whenotherserror ('text', ex);
        END;
END;
$$
```

GET STACKED DIAGNOSTICS:

Name	Тип	Описание
RETURNED_SQLSTATE	text	код исключения, возвращаемый SQLSTATE
COLUMN_NAME	text	имя столбца, относящегося к исключению
CONSTRAINT_NAME	text	имя ограничения целостности, относящегося к исключению
PG_DATATYPE_NAME	text	имя типа данных, относящегося к исключению
MESSAGE_TEXT	text	текст основного сообщения исключения
TABLE_NAME	text	имя таблицы, относящейся к исключению
SCHEMA_NAME	text	имя схемы, относящейся к исключению
PG_EXCEPTION_DETAIL	text	текст детального сообщения исключения (если есть)
PG_EXCEPTION_HINT	text	текст подсказки к исключению (если есть)
PG_EXCEPTION_CONTEXT	text	строки текста, описывающие стек вызовов в момент исключения

Вопросы для самопроверки

Что из перечисленного ниже является верным относительно автономных транзакций в PostgreSQL и их обработки ошибок?

- a) Автономные транзакции в PostgreSQL не поддерживают автоматический откат в случае исключения, требуется явный rollback.
- b) В случае успешного выполнения автономной транзакции она не коммитится автоматически.
- c) После того, как автономная транзакция словила исключение, внутри блока обработки ошибок rollback уже произведён, и для новых DML/DDl операций внутри этого блока требуется явный commit.
- d) Использование автономных транзакций не рекомендуется внутри циклов.

Вопросы для самопроверки

Что из перечисленного ниже является верным относительно автономных транзакций в PostgreSQL и их обработки ошибок?

- a) Автономные транзакции в PostgreSQL не поддерживают автоматический откат в случае исключения, требуется явный `rollback`.
- b) В случае успешного выполнения автономной транзакции она не коммитится автоматически.
- c) **После того, как автономная транзакция словила исключение, внутри блока обработки ошибок `rollback` уже произведён, и для новых DML/DDDL операций внутри этого блока требуется явный `commit`.**
- d) Использование автономных транзакций не рекомендуется внутри циклов.

Вопросы для самопроверки

Учитывая, что в PL/pgSQL при перехвате исключения в секции EXCEPTION все изменения с начала блока автоматически откатываются, какой эквивалентный фрагмент кода из Oracle PL/SQL следует портировать в PostgreSQL, чтобы получить аналогичное поведение отката при исключении?

- a) Нужно просто скопировать блок EXCEPTION из Oracle в PostgreSQL, так как поведение идентично.
- b) Можно просто убрать операторы SAVEPOINT и ROLLBACK TO из оригинального Oracle PL/SQL кода.
- c) Перед блоком EXCEPTION необходимо добавить SAVEPOINT my_savepoint; , а в блоке EXCEPTION вызвать ROLLBACK TO my_savepoint;
- d) Oracle PL/SQL изначально не поддерживает автоматический откат при исключении, в отличие от PostgreSQL, поэтому поведение воспроизвести невозможно.

Вопросы для самопроверки

Учитывая, что в PL/pgSQL при перехвате исключения в секции EXCEPTION все изменения с начала блока автоматически откатываются, какой эквивалентный фрагмент кода из Oracle PL/SQL следует портировать в PostgreSQL, чтобы получить аналогичное поведение отката при исключении?

- a) Нужно просто скопировать блок EXCEPTION из Oracle в PostgreSQL, так как поведение идентично.
- b) **Можно просто убрать операторы SAVEPOINT и ROLLBACK TO из оригинального Oracle PL/SQL кода.**
- c) Перед блоком EXCEPTION необходимо добавить SAVEPOINT my_savepoint; , а в блоке EXCEPTION вызвать ROLLBACK TO my_savepoint;
- d) Oracle PL/SQL изначально не поддерживает автоматический откат при исключении, в отличие от PostgreSQL, поэтому поведение воспроизвести невозможно.

Вопросы к третьему блоку



ИНВЕРСИЯ

Тел.: +7 495 721-19-57

www.inversion.ru

Буров Александр,
Ведущий программист направления разработки
«Валютные переводы», ИНВЕРСИЯ

Тел.: +7 495 721-19-57 (доб.375)
e-mail: Alexander.Burov@inversion.ru