Авторы: Егор Рогов, Павел Лузанов

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Темы

Процедуры и их отличие от функций

Входные и выходные параметры

Перегрузка и полиморфизм

Подпрограммы

Функции

- вызываются в контексте выражения
- не могут управлять транзакциями
- возвращают результат

Процедуры

- вызываются оператором CALL
- могут управлять транзакциями
- могут возвращать результат

Процедуры были введены в PostgreSQL 11. Основная причина их появления состоит в том, что функции не могут управлять транзакциями. Функции вызываются в контексте какого-либо выражения, которое вычисляется как часть уже начатого оператора (например SELECT) в уже начатой транзакции. Нельзя завершить транзакцию и начать новую «посередине» выполнения оператора.

Процедуры всегда вызываются специальным оператором CALL. Если этот оператор сам начинает новую транзакцию (а не вызывается из уже начатой), то в процедуре можно использовать команды управления транзакциями.

К сожалению, процедуры, написанные на языке SQL, лишены возможности использовать команды COMMIT и ROLLBACK. Поэтому пример процедуры, управляющей транзакциями, придется отложить до темы «PL/pgSQL. Выполнение запросов».

Иногда можно услышать, что процедура отличается от функции тем, что не возвращает результат. Но это не так — процедуры тоже могут возвращать результат, если необходимо.

Вместе функции и процедуры называются *подпрограммами (routine)*.

https://postgrespro.ru/docs/postgresql/12/sql-createprocedure

https://postgrespro.ru/docs/postgresql/12/sql-call

Процедуры без параметров

Начнем с примера простой процедуры без параметров.

```
=> CREATE TABLE t(a float);
```

```
CREATE TABLE
```

```
=> CREATE PROCEDURE fill()
AS $$
    TRUNCATE t;
    INSERT INTO t SELECT random() FROM generate_series(1,3);
$$ LANGUAGE sql;
```

```
CREATE PROCEDURE
```

Чтобы вызвать подпрограмму, необходимо использовать специальный оператор:

```
=> CALL fill();
```

```
CALL
```

Результат работы виден в таблице:

```
=> SELECT * FROM t;
```

```
          a
---------------------
 0.34413614061116604
  0.8777533932723642
 0.18337840060280897
(3 rows)
```

Тот же эффект можно получить и с помощью функции. Функция на языке SQL тоже может состоять из нескольких операторов (не обязательно SELECT); возвращаемое значение определяется последним оператором. Можно объявить тип результата void, если фактически функция ничего не возвращает, или вернуть что-то осмысленное:

```
=> CREATE FUNCTION fill_avg() RETURNS float
AS $$
    TRUNCATE t;
    INSERT INTO t SELECT random() FROM generate_series(1,3);
    SELECT avg(a) FROM t;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

В любом случае функция вызывается в контексте какого-либо выражения:

```
=> SELECT fill_avg();
```

```
      fill_avg
--------------------
 0.4615903155866062
(1 row)
```

```
=> SELECT * FROM t;
```

```
          a
---------------------
  0.6529629672924955
  0.6394871008896068
 0.09232087857771631
(3 rows)
```

Чего нельзя достичь с помощью функции — это управления транзакциями. Но и в процедурах на языке SQL это не поддерживается (зато поддерживается при использовании других языков).

Процедуры с параметрами

Добавим в процедуру входной параметр — число строк:

```
=> DROP PROCEDURE fill();
```

```
DROP PROCEDURE
```

```
=> CREATE PROCEDURE fill(nrows integer)
AS $$
    TRUNCATE t;
    INSERT INTO t SELECT random() FROM generate_series(1,nrows);
$$ LANGUAGE sql;
```

```
CREATE PROCEDURE
```

Точно так же, как и в случае функций, при вызове процедур фактические параметры можно передавать позиционным способом или по имени:

```
=> CALL fill(nrows => 5);
```

```
CALL
```

```
=> SELECT * FROM t;
```

```
          a
---------------------
  0.8701404833498891
  0.9593929021918797
  0.4200245938564784
  0.8680510881248082
 0.11433961530488901
(5 rows)
```

Процедуры могут также иметь INOUT-параметры, позволяющие возвращать значение. OUT-параметры пока не поддерживаются (но будут в PostgreSQL 14).

```
=> DROP PROCEDURE fill(integer);
```

```
DROP PROCEDURE
```

```
=> CREATE PROCEDURE fill(IN nrows integer, INOUT average float)
AS $$
    TRUNCATE t;
    INSERT INTO t SELECT random() FROM generate_series(1,nrows);
    SELECT avg(a) FROM t; -- как в функции
$$ LANGUAGE sql;
```

```
CREATE PROCEDURE
```

Попробуем:

Точно так же, как и в случае функций, при вызове процедур фактические параметры можно передавать позиционным способом или по имени:

```
=> CALL fill(5, NULL /* входное значение не используется*/);
```

```
      average
--------------------
 0.6374691082887736
(1 row)
```

Перегрузка

Несколько подпрограмм с одним и тем же именем

- подпрограммы различаются типами входных параметров;
тип возвращаемого значения и выходные параметры игнорируются
- подходящая подпрограмма выбирается во время выполнения
в зависимости от фактических параметров

Команда CREATE OR REPLACE

- при несовпадении типов входных параметров создаст новую
перегруженную подпрограмму
- при совпадении — изменит существующую подпрограмму,
но поменять тип возвращаемого значения нельзя

Перегрузка — это возможность использования одного и того же имени для нескольких подпрограмм (функций или процедур), отличающихся типами параметров IN и INOUT. Иными словами, *сигнатура подпрограммы* — ее имя и типы входных параметров.

При вызове подпрограммы PostgreSQL находит ту подпрограмму, которая соответствует переданным фактическим параметрам. Возможны ситуации, когда подходящую подпрограмму невозможно определить однозначно; в таком случае во время выполнения возникнет ошибка.

Перегрузку надо учитывать при использовании команды CREATE OR REPLACE (FUNCTION или PROCEDURE). Дело в том, что при несовпадении типов входных параметров будет создана новая — перегруженная — подпрограмма. Кроме того, для функций эта команда не позволяет изменить тип возвращаемого значения и типы выходных параметров.

Поэтому при необходимости следует удалять подпрограмму и создавать ее заново, но это будет уже новая подпрограмма. При удалении старой функции потребуется также удалить зависящие от нее представления, триггеры и т. п. (DROP FUNCTION ... CASCADE).

https://postgrespro.ru/docs/postgresql/12/xfunc-overload

Полиморфизм

Подпрограмма, принимающая параметры разных типов

- формальные параметры используют полиморфные псевдотипы (например, `anyelement`)
- конкретный тип данных выбирается во время выполнения по типу фактических параметров

В некоторых случаях удобно не создавать несколько перегруженных подпрограмм для разных типов, а написать одну, принимающую параметры любого (или почти любого) типа.

Для этого в качестве типа формального параметра указывается специальный *полиморфный псевдотип*. Пока мы ограничимся одним типом — anyelement, который соответствует любому базовому типу, — но позже познакомимся и с другими.

Конкретный тип, с которым будет работать подпрограмма, выбирается во время выполнения по типу фактического параметра.

Если у подпрограммы определено несколько полиморфных параметров, то типы соответствующих фактических параметров должны совпадать. Иными словами, anyelement при каждом вызове функции обозначает какой-то один конкретный тип.

Если функция объявлена с возвращаемым значением полиморфного типа, то она должна иметь по крайней мере один входной полиморфный параметр. Конкретный тип возвращаемого значения также определяется исходя из типа фактического входного параметра.

https://postgrespro.ru/docs/postgresql/12/extend-type-system#EXTEND-TYPES-POLYMORPHIC

https://postgrespro.ru/docs/postgresql/12/xfunc-sql#id-1.8.3.8.18

Перегруженные подпрограммы

Перегрузка работает одинаково и для функций, и для процедур. Они имеют общее пространство имен.

В качестве примера напишем функцию, возвращающую большее из двух целых чисел. (Похожее выражение есть в SQL и называется greatest, но мы напишем собственную функцию.)

```
=> CREATE FUNCTION maximum(a integer, b integer) RETURNS integer
AS $$
    SELECT CASE WHEN a > b THEN a ELSE b END;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Проверим:

```
=> SELECT maximum(10, 20);
```

```
 maximum
---------
      20
(1 row)
```

Допустим, мы решили сделать аналогичную функцию для трех чисел. Благодаря перегрузке, не надо придумывать для нее какое-то новое название:

```
=> CREATE FUNCTION maximum(a integer, b integer, c integer)
RETURNS integer
AS $$
SELECT CASE
        WHEN a > b THEN maximum(a,c)
        ELSE maximum(b,c)
    END;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Теперь у нас две функции с одним именем, но разным числом параметров:

```
=> \df maximum
```

```
                               List of functions
 Schema |  Name   | Result data type |       Argument data types       | Type
--------+---------+------------------+---------------------------------+------
 public | maximum | integer          | a integer, b integer            | func
 public | maximum | integer          | a integer, b integer, c integer | func
(2 rows)
```

И обе работают:

```
=> SELECT maximum(10, 20), maximum(10, 20, 30);
```

```
 maximum | maximum
---------+---------
      20 |      30
(1 row)
```

Команда CREATE OR REPLACE позволяет создать подпрограмму или заменить существующую, не удаляя ее. Поскольку в данном случае функция с такой сигнатурой уже существует, она будет заменена:

```
=> CREATE OR REPLACE FUNCTION maximum(a integer, b integer, c integer)
RETURNS integer
AS $$
SELECT CASE
        WHEN a > b THEN
            CASE WHEN a > c THEN a ELSE c END
        ELSE
            CASE WHEN b > c THEN b ELSE c END
    END;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Пусть наша функция работает не только для целых чисел, но и для вещественных.

Как этого добиться? Можно было бы определить еще такую функцию:

```
=> CREATE FUNCTION maximum(a real, b real) RETURNS real
AS $$
    SELECT CASE WHEN a > b THEN a ELSE b END;
$$ LANGUAGE sql;

CREATE FUNCTION
```

Теперь у нас три функции с одинаковым именем:

```
=> \df maximum

                              List of functions
 Schema |  Name   | Result data type |       Argument data types       | Type
--------+---------+------------------+---------------------------------+------
 public | maximum | integer          | a integer, b integer            | func
 public | maximum | integer          | a integer, b integer, c integer | func
 public | maximum | real             | a real, b real                  | func
(3 rows)
```

Две из них имеют одинаковое количество параметров, но отличаются их типами:

```
=> SELECT maximum(10, 20), maximum(1.1, 2.2);

 maximum | maximum
---------+---------
      20 |     2.2
(1 row)
```

Но дальше нам придется определить функции и для всех остальных типов данных, при том, что тело функции не меняется. И затем придется повторить все то же самое для варианта с тремя параметрами.

Полиморфные функции

Здесь нам поможет полиморфный тип anyelement.

Удалим все три наши функции...

```
=> DROP FUNCTION maximum(integer, integer);

DROP FUNCTION

=> DROP FUNCTION maximum(integer, integer, integer);

DROP FUNCTION

=> DROP FUNCTION maximum(real, real);

DROP FUNCTION
```

...и затем создадим новую:

```
=> CREATE FUNCTION maximum(a anyelement, b anyelement)
RETURNS anyelement
AS $$
    SELECT CASE WHEN a > b THEN a ELSE b END;
$$ LANGUAGE sql;

CREATE FUNCTION
```

Такая функция должна принимать любой тип данных (а работать будет с любым типом, для которого определен оператор «больше»).

Получится?

```
=> SELECT maximum('A', 'B');

ERROR:  could not determine polymorphic type because input has type unknown
```

Увы, нет. В данном случае строковые литералы могут быть типа char, varchar, text — конкретный тип нам неизвестен. Но можно применить явное приведение типов:

```
=> SELECT maximum('A'::text, 'B'::text);

 maximum
---------
 B
(1 row)
```

Еще пример с другим типом:

```
=> SELECT maximum(now(), now() + interval '1 day');
```

```
            maximum
-------------------------------
 2022-11-18 11:16:18.182105+03
(1 row)
```

Тип результата функции всегда будет тот же, что и тип параметров.

Важно, чтобы типы обоих параметров совпадали, иначе будет ошибка:

```
=> SELECT maximum(1, 'A');
```

```
ERROR:  invalid input syntax for type integer: "A"
LINE 1: SELECT maximum(1, 'A');
                          ^
```

В этом примере такое ограничение выглядит естественно, хотя в некоторых случаях оно может оказаться и неудобным.

Определим теперь функцию с тремя параметрами, но так, чтобы третий можно было не указывать.

```
=> CREATE FUNCTION maximum(
    a anyelement,
    b anyelement,
    c anyelement DEFAULT NULL
) RETURNS anyelement
AS $$
SELECT CASE
        WHEN c IS NULL THEN
            x
        ELSE
            CASE WHEN x > c THEN x ELSE c END
    END
FROM (
    SELECT CASE WHEN a > b THEN a ELSE b END
) max2(x);
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Попробуем:

```
=> SELECT maximum(10, 20, 30);
```

```
 maximum
---------
      30
(1 row)
```

Так работает. А так?

```
=> SELECT maximum(10, 20);
```

```
ERROR:  function maximum(integer, integer) is not unique
LINE 1: SELECT maximum(10, 20);
               ^
HINT:  Could not choose a best candidate function. You might need to add explicit type casts.
```

А так произошел конфликт перегруженных функций:

```
=> \df maximum
```

```
                                          List of functions
 Schema |  Name   | Result data type |                      Argument data types                       | Type
--------+---------+------------------+----------------------------------------------------------------+------
 public | maximum | anyelement       | a anyelement, b anyelement                                     | func
 public | maximum | anyelement       | a anyelement, b anyelement, c anyelement DEFAULT NULL::unknown | func
(2 rows)
```

Невозможно понять, имеем ли мы в виду функцию с двумя параметрами, или с тремя (но просто не указали последний).

Мы решим этот конфликт просто — удалим первую функцию за ненадобностью.

```
=> DROP FUNCTION maximum(anyelement, anyelement);
```

```
DROP FUNCTION
```

```
=> SELECT maximum(10, 20), maximum(10, 20, 30);
```

```
 maximum | maximum
---------+---------
      20 |      30
(1 row)
```

Теперь все работает. А в теме «PL/pgSQL. Массивы» мы узнаем, как определять подпрограммы с произвольным числом параметров.

Итоги

Можно создавать и использовать собственные процедуры

В отличие от функций, процедуры вызываются оператором CALL и могут управлять транзакциями

Для процедур и функций поддерживаются перегрузка и полиморфизм

Практика

1. В таблице authors имена, фамилии и отчества авторов по смыслу должны быть уникальны, но это условие никак не проверяется. Напишите процедуру, удаляющую возможные дубликаты авторов.
2. Чтобы необходимость в подобной процедуре не возникала, создайте ограничение целостности, которое не позволит появляться дубликатам в будущем.

1. В приложении возможность добавлять авторов появится в теме «PL/pgSQL. Выполнение запросов». А пока для проверки можно добавить дубликаты в таблицу вручную.

1. Устранение дубликатов

В целях проверки добавим второго Пушкина:

```
=> INSERT INTO authors(last_name, first_name, middle_name)
    VALUES ('Пушкин', 'Александр', 'Сергеевич');
```

```
INSERT 0 1
```

```
=> SELECT last_name, first_name, middle_name, count(*)
FROM authors
GROUP BY last_name, first_name, middle_name;
```

```
 last_name  | first_name | middle_name | count
------------+------------+-------------+-------
 Свифт      | Джонатан   |             |     1
 Стругацкий | Борис      | Натанович   |     1
 Пушкин     | Александр  | Сергеевич   |     2
 Стругацкий | Аркадий    | Натанович   |     1
 Толстой    | Лев        | Николаевич  |     1
 Тургенев   | Иван       | Сергеевич   |     1
(6 rows)
```

Задачу устранения дубликатов можно решить разными способами. Например, так:

```
=> CREATE PROCEDURE authors_dedup()
AS $$
DELETE FROM authors
WHERE author_id IN (
    SELECT author_id
    FROM (
        SELECT author_id,
               row_number() OVER (
                   PARTITION BY first_name, last_name, middle_name
                   ORDER BY author_id
               ) AS rn
        FROM authors
    ) t
    WHERE t.rn > 1
);
$$ LANGUAGE sql;
```

```
CREATE PROCEDURE
```

```
=> CALL authors_dedup();
```

```
CALL
```

```
=> SELECT last_name, first_name, middle_name, count(*)
FROM authors
GROUP BY last_name, first_name, middle_name;
```

```
 last_name  | first_name | middle_name | count
------------+------------+-------------+-------
 Свифт      | Джонатан   |             |     1
 Стругацкий | Борис      | Натанович   |     1
 Пушкин     | Александр  | Сергеевич   |     1
 Стругацкий | Аркадий    | Натанович   |     1
 Толстой    | Лев        | Николаевич  |     1
 Тургенев   | Иван       | Сергеевич   |     1
(6 rows)
```

2. Ограничение целостности

Создать подходящее ограничение целостности мешает тот факт, что отчество может быть неопределенным (NULL). Неопределенные значения считаются различными, поэтому ограничение

```
UNIQUE(first_name, last_name, middle_name)
```

не помешает добавить второго Джонатана Свифта без отчества.

Задачу можно решить, создав уникальный индекс:

```
=> CREATE UNIQUE INDEX authors_full_name_idx ON authors(
    last_name, first_name, coalesce(middle_name,'')
);
```

```
CREATE INDEX
```

Проверим:

```
=> INSERT INTO authors(last_name, first_name)
    VALUES ('Свифт', 'Джонатан');

ERROR:  duplicate key value violates unique constraint "authors_full_name_idx"
DETAIL:  Key (last_name, first_name, COALESCE(middle_name, ''::text))=(Свифт, Джонатан, ) already exists.

=> INSERT INTO authors(last_name, first_name, middle_name)
    VALUES ('Пушкин', 'Александр', 'Сергеевич');

ERROR:  duplicate key value violates unique constraint "authors_full_name_idx"
DETAIL:  Key (last_name, first_name, COALESCE(middle_name, ''::text))=(Пушкин, Александр, Сергеевич) already exists.
```

Практика

1. Получится ли создать в одной и той же схеме и имеющие одно и то же имя: 1) процедуру с одним входным параметром, 2) функцию с одним входным параметром того же типа, возвращающую некоторое значение? Проверьте.
2. В таблице хранятся вещественные числа (например, результаты каких-либо измерений). Напишите процедуру нормализации данных, которая умножает все числа на определенный коэффициент так, чтобы все значения попали в интервал от −1 до 1.
   Процедура должна возвращать выбранный коэффициент.

2. В качестве коэффициента возьмите максимальное абсолютное значение из таблицы.

1. Перегрузка процедур и функций

Не получится, так как в сигнатуру подпрограммы входит только имя и тип входных параметров (возвращаемое значение игнорируется), и при этом процедуры и функции имеют общее пространство имен.

```
=> CREATE PROCEDURE test(IN x integer)
AS $$
    SELECT 1;
$$ LANGUAGE sql;
```

```
CREATE PROCEDURE
```

```
=> CREATE FUNCTION test(IN x integer) RETURNS integer
AS $$
    SELECT 1;
$$ LANGUAGE sql;
```

```
ERROR:  function "test" already exists with same argument types
```

В некоторых сообщениях, как и в этом, вместо слова «процедура» используется «функция», поскольку во многом они устроены одинаково.

2. Нормализация данных

Таблица с тестовыми данными:

```
=> CREATE TABLE samples(a float);
```

```
CREATE TABLE
```

```
=> INSERT INTO samples(a)
    SELECT (0.5 - random())*100 FROM generate_series(1,10);
```

```
INSERT 0 10
```

Процедуру можно написать, используя один SQL-оператор:

```
=> CREATE PROCEDURE normalize_samples(INOUT coeff float)
AS $$
    WITH c(coeff) AS (
        SELECT 1/max(abs(a))
        FROM samples
    ),
    upd AS (
        UPDATE samples
        SET a = a * c.coeff
        FROM c
    )
    SELECT coeff FROM c;
$$ LANGUAGE sql;
```

```
CREATE PROCEDURE
```

```
=> CALL normalize_samples(NULL);
```

```
        coeff
--------------------
 0.02092074269393098
(1 row)


=> SELECT * FROM samples;

          a
----------------------
   0.5091623540495687
   0.7086300677160096
                    1
   0.6567707165234902
   0.8733148166896652
  0.34300769190836694
  -0.8690290824590934
 -0.20579432178047002
 -0.19847790349106587
  -0.8641454301141649
(10 rows)
```