



### **Авторские права**

© Postgres Professional, 2017–2020

Авторы: Егор Рогов, Павел Лузанов

### **Использование материалов курса**

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

### **Обратная связь**

Отзывы, замечания и предложения направляйте по адресу:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

### **Отказ от ответственности**

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Проверки корректности

Отладчик PL/pgSQL

Служебные сообщения и различные способы их реализации

Трассировка сеансов

## Проверки времени компиляции и времени выполнения

*plpgsql.extra\_warnings*

*plpgsql.extra\_errors*

дополнительные проверки в расширении *plpgsql\_check*

## Проверки в коде

команда ASSERT

## Тестирование

Отладка предполагает выполнение программы и изучение возникающих проблем, обычно с помощью специализированного отладчика или с помощью вывода отладочных сообщений.

Но можно заранее исключить определенные классы ошибок, включив проверку исходного кода во время компиляции и выполнения. Это управляется параметрами *plpgsql.extra\_warnings* и *plpgsql.extra\_errors*, как рассматривалось в теме «PL/pgSQL. Выполнение запросов».

<https://postgrespro.ru/docs/postgrespro/12/plpgsql-development-tips#PLPGSQL-EXTRA-CHECKS>

Там же рассматривалось применение расширения *plpgsql\_check*, которое обеспечивает более широкий спектр проверок.

Еще одна возможность обезопасить свой код — добавить в него проверки условий, которые всегда должны выполняться (т. н. «sanity check»). Удобный способ для этого — SQL-команда ASSERT.

<https://postgrespro.ru/docs/postgresql/12/plpgsql-errors-and-messages#PLPGSQL-STATEMENTS-ASSERT>

Нельзя не сказать и о важности тестирования кода. Тестирование не только позволяет изначально убедиться, что код работает так, как задумано, но и облегчает его дальнейшую модификацию — дает уверенность в том, что изменения не поломали существующий функционал. Мы не будем затрагивать эту тему; отметим только, что тестирование кода, работающего с базой данных, может оказаться непростым делом из-за необходимости подготовки тестовых примеров.

## Проверки корректности

Команда ASSERT позволяет указать условия, нарушения которых является непредвиденной ошибкой. Можно провести определенную аналогию между такими условиями и ограничениями целостности в базе данных.

Пример: функция, возвращающее номер подъезда по номеру квартиры:

```
=> CREATE FUNCTION entrance(  
    floors integer,  
    flats_per_floor integer,  
    flat_no integer  
)  
RETURNS integer  
AS $$  
BEGIN  
    RETURN floor((flat_no - 1)::real / (floors * flats_per_floor)) + 1;  
END;  
$$ LANGUAGE plpgsql IMMUTABLE;
```

CREATE FUNCTION

Убедиться в правильности можно при помощи тестирования, проверив результат на некоторых «интересных» значениях:

```
=> SELECT entrance(9, 4, 1), entrance(9, 4, 36), entrance(9, 4, 37);
```

```
entrance | entrance | entrance  
-----+-----+-----  
1 | 1 | 2  
(1 row)
```

Но при некорректных входных значениях функция будет выдавать бессмысленный результат, который, например, может быть передан дальше в другие подпрограммы, которые из-за этого тоже могут повести себя некорректно. Тестирование только данной функции здесь никак не поможет.

```
=> SELECT entrance(9, 4, 0);
```

```
entrance  
-----  
0  
(1 row)
```

Можно обезопасить себя, добавив проверку:

```
=> CREATE OR REPLACE FUNCTION entrance(  
    floors integer,  
    flats_per_floor integer,  
    flat_no integer  
)  
RETURNS integer  
AS $$  
BEGIN  
    ASSERT floors > 0 AND flats_per_floor > 0 AND flat_no > 0,  
    'Некорректные входные параметры';  
    RETURN floor((flat_no - 1)::real / (floors * flats_per_floor)) + 1;  
END;  
$$ LANGUAGE plpgsql IMMUTABLE;
```

CREATE FUNCTION

```
=> SELECT entrance(9, 4, 0);
```

ERROR: Некорректные входные параметры

CONTEXT: PL/pgSQL function entrance(integer,integer,integer) line 3 at ASSERT

Теперь некорректный вызов сразу же приведет к ошибке.

## Состав

- расширение pldbgar1
- встроенная поддержка в некоторых графических средах

## Возможности

- установка точек прерывания
- пошаговое выполнение
- проверка и установка значений переменных
- не требуется изменение кода
- отладка работающих приложений

PL/pgSQL Debugger — это отладчик для PL/pgSQL. Представляет собой расширение pldbgar1, которое официально поддерживается разработчиками PostgreSQL.

Расширение pldbgar1 — это набор интерфейсных функций для сервера PostgreSQL, которые позволяют устанавливать точки прерывания, пошагово выполнять код программ, проверять и устанавливать значений переменных.

Исходный код отлаживаемых программ изменять не требуется, что дает возможность выполнять отладку работающих приложений. Т. е. процесс отладки не обязательно запускать отдельно, можно подключиться к уже работающему сеансу и начать его отладку.

Непосредственно пользоваться этими функциями неудобно, они в первую очередь предназначены для графических сред разработки. Некоторые из таких сред (включая pgAdmin) имеют встроенный интерфейс для отладки. Но чтобы им воспользоваться, сначала все равно придется установить расширение pldbgar1 в соответствующую базу данных.

Исходный код отладчика доступен по ссылке:

<https://git.postgresql.org/gitweb/?p=pldebugger.git;a=summary>

## Не только отладка кода

- мониторинг долго выполняющихся процессов
- ведение журнала приложения

## Подходы к реализации

- вывод на консоль или в журнал сервера
- запись в таблицу или в файл
- передача информации другим процессам

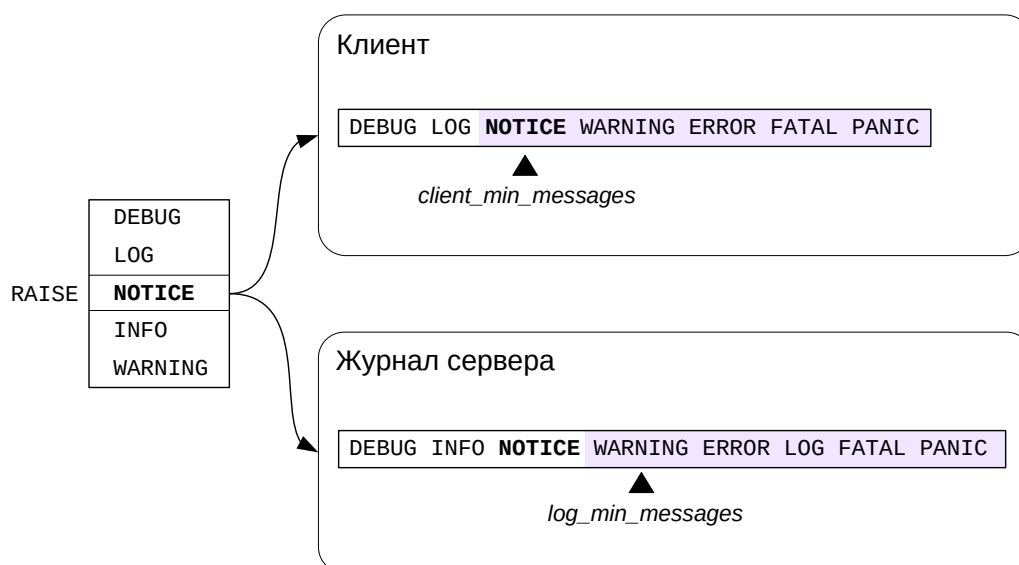
Второй способ отладки предполагает добавление в важные места кода служебных сообщений, содержащих текущий контекст. Анализируя сообщения можно понять, что именно пошло не так.

Помимо собственно отладки, служебные сообщения могут выполнять и другие функции. Сообщения помогут определить, на каком этапе выполнения находится долго выполняющийся процесс. По аналогии с журналом сообщений СУБД приложение может записывать в свой собственный журнал важную информацию. Например, данные о запуске отчетов: название отчета, пользователь, когда, с какими параметрами и т. д. Такая информация может сильно облегчить работу специалистов поддержки.

Можно выделить несколько подходов к реализации служебных сообщений в PL/pgSQL. Помимо уже знакомой команды RAISE, позволяющей выводить сообщения на консоль (а также в журнал сообщений сервера), можно отправлять сообщения другому процессу, записывать сообщения в таблицу или в файл.

При выборе того или иного подхода нужно обращать внимание на много вопросов. Являются ли сообщения транзакционными (отправляются не дожидаясь окончания транзакции или только при ее завершении)? Можно ли одновременно использовать в нескольких сеансах? Как организовать доступ к журналу сообщений и как очищать его от старых записей? Какое влияние оказывают журналирование на производительность? Требуется ли модификация исходного кода?

# Команда RAISE



7

Мы уже встречались с командой RAISE. Она служит и для вызова исключительных ситуаций, что подробно рассматривалось в теме «PL/pgSQL. Обработка ошибок», и для отправки сообщений. Причем сообщения можно не только отправлять клиенту, но и записывать в журнал сервера.

В простом случае для отладки нужно добавить вызовы RAISE NOTICE в код функции, запустить функцию на выполнение (например в сеансе psql) и проанализировать получаемые по ходу выполнения сообщения. Сообщения RAISE нетранзакционные: они отправляются асинхронно и не зависят от статуса завершения транзакции.

Для управления отправкой сообщений используются уровень сообщения (DEBUG, LOG, NOTICE, INFO, WARNING) и параметры сервера. Он значений параметров зависит, будет ли сообщение отправлено клиенту (*client\_min\_messages*) и/или записано в журнал сервера (*log\_min\_messages*). Сообщение будет отправлено, если уровень команды RAISE равен значению соответствующего параметра или больше его (находится правее на рисунке).

Значения параметров по умолчанию настроены так, что сообщения с уровнем NOTICE отправляются только клиенту, с уровнем LOG — только в журнал, а с уровнем WARNING — и клиенту, и в журнал.

Сообщения с уровнем INFO всегда отправляются клиенту, их нельзя перехватить параметром *client\_min\_messages*.

<https://postgrespro.ru/docs/postgresql/12/plpgsql-errors-and-messages>

## Команда RAISE

Создадим функцию для подсчета количества строк в таблице, имя которой передается параметром.

```
=> CREATE FUNCTION get_count(tabname text) RETURNS bigint
AS $$
DECLARE
    cmd text;
    retval bigint;
BEGIN
    cmd := 'SELECT COUNT(*) FROM ' || quote_ident(tabname);
    RAISE NOTICE 'cmd: %', cmd;
    EXECUTE cmd INTO retval;
    RETURN retval;
END;
$$ LANGUAGE plpgsql STABLE;
```

CREATE FUNCTION

Для динамического выполнения текст команды лучше предварительно записывать в переменную. В случае ошибок можно получить содержимое переменной.

```
=> SELECT get_count('pg_class');
```

```
NOTICE: cmd: SELECT COUNT(*) FROM pg_class
get_count
-----
          395
(1 row)
```

Строка, начинающаяся с «NOTICE» — наша отладочная информация.

RAISE можно использовать для отслеживания хода выполнения долгого запроса.

Предположим, что внутри кода четко выделяются три этапа выполнения. И по ходу работы подпрограммы мы хотим понимать, где сейчас находимся.

```
=> CREATE PROCEDURE long_running()
AS $$
BEGIN
    RAISE NOTICE 'long_running. Stage 1/3...';
    PERFORM pg_sleep(2);

    RAISE NOTICE 'long_running. Stage 2/3...';
    PERFORM pg_sleep(3);

    RAISE NOTICE 'long_running. Stage 3/3...';
    PERFORM pg_sleep(1);

    RAISE NOTICE 'long_running. Done.';
END;
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

Команда RAISE выдает сообщения сразу, а не в конце работы функции:

```
=> CALL long_running();
```

```
NOTICE: long_running. Stage 1/3...
NOTICE: long_running. Stage 2/3...
NOTICE: long_running. Stage 3/3...
NOTICE: long_running. Done.
CALL
```

Такой подход удобен, когда можно вызвать функцию в отдельном сеансе. Если же функция вызывается из приложения, то удобнее писать и смотреть в журнал сервера.

Напишем процедуру raise\_msg для выдачи сообщения с уровнем, установленным в пользовательском параметре app.raise\_level:

```
=> CREATE OR REPLACE PROCEDURE raise_msg(msg text)
AS $$
BEGIN
    CASE current_setting('app.raise_level', true)
        WHEN 'NOTICE' THEN RAISE NOTICE '%, %, %', user, clock_timestamp(), msg;
        WHEN 'DEBUG' THEN RAISE DEBUG '%, %, %', user, clock_timestamp(), msg;
        WHEN 'LOG' THEN RAISE LOG '%, %, %', user, clock_timestamp(), msg;
        WHEN 'INFO' THEN RAISE INFO '%, %, %', user, clock_timestamp(), msg;
        WHEN 'WARNING' THEN RAISE WARNING '%, %, %', user, clock_timestamp(), msg;
        ELSE NULL; -- все прочие значения отключают вывод сообщений
    END CASE;
END;
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

Для целей примера установим параметр на уровне сеанса и переделаем процедуру long\_running на использование raise\_msg:

```
=> SET app.raise_level TO 'NONE';
```



SET

```
=> CREATE OR REPLACE PROCEDURE long_running()
AS $$
BEGIN
    CALL raise_msg('long_running. Stage 1/3...');
    PERFORM pg_sleep(2);

    CALL raise_msg('long_running. Stage 2/3...');
    PERFORM pg_sleep(3);

    CALL raise_msg('long_running. Stage 3/3...');
    PERFORM pg_sleep(1);

    CALL raise_msg('long_running. Done.');
```

END;

```
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

Теперь в «обычной» жизни (app.raise\_level = NONE) отладочные сообщения не будут выдаваться:

```
=> CALL long_running();
```

CALL

Запуская функцию в отдельном сеансе, мы можем получить отладочные сообщения, выставив app.raise\_level в NOTICE:

```
=> SET app.raise_level TO 'NOTICE';
```

SET

```
=> CALL long_running();
```

```
NOTICE: student, 2021-01-12 20:34:04.253531+03, long_running. Stage 1/3...
NOTICE: student, 2021-01-12 20:34:06.255678+03, long_running. Stage 2/3...
NOTICE: student, 2021-01-12 20:34:09.259964+03, long_running. Stage 3/3...
NOTICE: student, 2021-01-12 20:34:10.261992+03, long_running. Done.
CALL
```

Если же мы хотим включить отладку в приложении с записью в журнал сервера, то переключаем app.raise\_level в LOG:

```
=> SET app.raise_level TO 'LOG';
```

SET

```
=> CALL long_running();
```

CALL

Смотрим в журнал сервера:

```
student$ tail -n 20 /var/log/postgresql/postgresql-12-main.log | grep long_running
```

```
2021-01-12 20:34:10.334 MSK [190182] student@plpgsql_debug LOG: student, 2021-01-12 20:34:10.334949+03, long_running. Stage 1/3...
SQL statement "CALL raise_msg('long_running. Stage 1/3...')"
PL/pgSQL function long_running() line 3 at CALL
2021-01-12 20:34:10.334 MSK [190182] student@plpgsql_debug STATEMENT: CALL long_running();
2021-01-12 20:34:12.338 MSK [190182] student@plpgsql_debug LOG: student, 2021-01-12 20:34:12.338158+03, long_running. Stage 2/3...
SQL statement "CALL raise_msg('long_running. Stage 2/3...')"
PL/pgSQL function long_running() line 6 at CALL
2021-01-12 20:34:12.338 MSK [190182] student@plpgsql_debug STATEMENT: CALL long_running();
2021-01-12 20:34:15.342 MSK [190182] student@plpgsql_debug LOG: student, 2021-01-12 20:34:15.34267+03, long_running. Stage 3/3...
SQL statement "CALL raise_msg('long_running. Stage 3/3...')"
PL/pgSQL function long_running() line 9 at CALL
2021-01-12 20:34:15.342 MSK [190182] student@plpgsql_debug STATEMENT: CALL long_running();
2021-01-12 20:34:16.344 MSK [190182] student@plpgsql_debug LOG: student, 2021-01-12 20:34:16.344511+03, long_running. Done.
SQL statement "CALL raise_msg('long_running. Done.')"
PL/pgSQL function long_running() line 12 at CALL
2021-01-12 20:34:16.344 MSK [190182] student@plpgsql_debug STATEMENT: CALL long_running();
```

Управляя параметрами app.raise\_level, log\_min\_messages и client\_min\_messages, можно добиться различного поведения отладочных сообщений.

Важно, что для этого не нужно менять код приложения.

## NOTIFY → LISTEN

команды SQL

транзакционное выполнение, неудобно для отладки

## Статус сеанса

параметр *application\_name*

виден в представлении `pg_stat_activity` и выводе команды `ps`

можно использовать в журнальных сообщениях

Серверные процессы в PostgreSQL могут обмениваться информацией между собой.

Среди встроенных решений можно отметить следующие.

- Использование команд `NOTIFY` для отправки сообщений в одном процессе и `LISTEN` для получения в другом. Но эти команды являются транзакционными, поэтому они неудобны для отладки:
  1. Сообщения отправляются только при фиксации транзакции, а не сразу после выполнения `NOTIFY`. Поэтому невозможно следить за ходом процесса.
  2. Если транзакция завершится неуспешно, то сообщения вообще не будут отправлены.
- Использование параметра *application\_name*.

Сеанс с долго выполняющимся процессом может периодически записывать статус выполнения в *application\_name*. В другом сеансе администратор может опрашивать представление `pg_stat_activity`, содержащем подробную информацию о всех выполняющихся сеансах. Также значение *application\_name* обычно видно в выводе команды `ps`.

Значение *application\_name* также можно записывать в журнал сервера (настраивая параметр *log\_line\_prefix*). Это облегчит поиск нужных строк в журнале.

<https://postgrespro.ru/docs/postgrespro/12/runtime-config-logging#RUNTIME-CONFIG-LOGGING-WHAT>

## Статус сеанса

Посмотрим, как использовать для отладки параметр `application_name`. Первый сеанс меняет значение этого параметра, второй — периодически опрашивает представление `pg_stat_activity`.

Новый вариант процедуры:

```
=> CREATE OR REPLACE PROCEDURE long_running()
AS $$
BEGIN
    SET LOCAL application_name TO "long_running. Stage 1/3...";
    PERFORM pg_sleep(2);

    SET LOCAL application_name TO "long_running. Stage 2/3...";
    PERFORM pg_sleep(3);

    SET LOCAL application_name TO "long_running. Stage 3/3...";
    PERFORM pg_sleep(1);

    SET LOCAL application_name TO "long_running. Done.";
END;
$$ LANGUAGE plpgsql;

CREATE PROCEDURE
```

Запускаем в первом сеансе:

```
=> CALL long_running();
```

Во втором с паузой в 2 секунды обновляем строку из `pg_stat_activity`:

```
=> SELECT pid, username, application_name
FROM pg_stat_activity
WHERE datname = 'plpgsql_debug' AND pid <> pg_backend_pid();
```

```
 pid | username | application_name
-----+-----+-----
190182 | student | long_running. Stage 1/3...
(1 row)
```

```
=> \g
```

```
 pid | username | application_name
-----+-----+-----
190182 | student | long_running. Stage 2/3...
(1 row)
```

```
=> \g
```

```
 pid | username | application_name
-----+-----+-----
190182 | student | long_running. Stage 3/3...
(1 row)
```

```
=> \g
```

```
 pid | username | application_name
-----+-----+-----
190182 | student | psql
(1 row)
```

CALL

### Расширение dblink

- входит в состав сервера
- накладные расходы на создание соединения

### Автономные транзакции

- коммерческий дистрибутив (Postgres Pro Enterprise)

Еще один способ сохранения отладочных сообщений — запись в таблицу базы данных.

К плюсам данного подхода относится то, что параллельная работа и доступ журналу обеспечиваются средствами самой СУБД.

Однако нужно позаботиться о том, чтобы вставка в таблицу была нетранзакционной. Для этого можно использовать расширение dblink, идущее в составе сервера PostgreSQL. Это расширение позволяет открыть новое соединение к той же самой базе данных, поэтому вставка в таблицу выполняется в другой — независимой — транзакции.

К минусам данного подхода относится то, что открытие нового соединения требует дополнительных ресурсов сервера.

Подробнее работа с dblink рассматривается в курсе DEV2.

<https://postgrespro.ru/docs/postgrespro/12/dblink>

В коммерческих дистрибутивах, например Postgres Pro Enterprise, доступен механизм автономных транзакций, не имеющий таких существенных накладных расходов, которые характерны для dblink.

## Запись в таблицу: расширение dblink

Установим расширение:

```
=> CREATE EXTENSION dblink;
```

```
CREATE EXTENSION
```

Создаем таблицу для записи сообщений.

В таблице полезно сохранять информацию о пользователе и времени вставки. Столбец id нужен для гарантированной сортировки результата в порядке добавления строк.

```
=> CREATE TABLE log (  
    id          integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,  
    username    text,  
    ts          timestampz,  
    message     text  
);
```

```
CREATE TABLE
```

Теперь создадим процедуру для удобного добавления записей в таблицу log. Процедура открывает новый сеанс, выполняет вставку в отдельной транзакции и закрывает сеанс.

```
=> CREATE PROCEDURE write_log(message text)  
AS $$  
DECLARE  
    cmd text;  
BEGIN  
    cmd := format(  
        'INSERT INTO log (username, ts, message)  
        VALUES (%L, %L::timestampz, %L)',  
        user, clock_timestamp()::text, write_log.message  
    );  
    PERFORM dblink('dbname=' || current_database(), cmd);  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Создаем новый вариант long\_running:

```
=> CREATE OR REPLACE PROCEDURE long_running()  
AS $$  
BEGIN  
    CALL write_log('long_running. Stage 1/3...');  
    PERFORM pg_sleep(2);  
  
    CALL write_log('long_running. Stage 2/3...');  
    PERFORM pg_sleep(3);  
  
    CALL write_log('long_running. Stage 3/3...');  
    PERFORM pg_sleep(1);  
  
    CALL write_log('long_running. Done.');
```

```
END;
```

```
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Для проверки запустим процедуру long\_running в отдельной транзакции, которую в конце откатим.

```
=> BEGIN;
```

```
BEGIN
```

```
=> CALL long_running();
```

```
CALL
```

```
=> ROLLBACK;
```

```
ROLLBACK
```

Убедимся, что в таблице сохранились все вызовы write\_log. По значениям ts можно проверить, сколько времени прошло между вызовами.

```
=> SELECT username, to_char(ts, 'HH24:MI:SS') as ts, message
FROM log
```

```
ORDER BY id;
```

username	ts	message
student	20:34:24	long_running. Stage 1/3...
student	20:34:26	long_running. Stage 2/3...
student	20:34:29	long_running. Stage 3/3...
student	20:34:30	long_running. Done.

(4 rows)

### Расширение adminpack

входит в состав сервера  
в том числе позволяет записывать текстовые файлы

### Недоверенные языки

например, PL/Perl

Вести запись сообщений можно и в файл операционной системы.

Для этого, например, можно использовать расширение adminpack, которое позволяет записывать данные в любой файл, доступ к которому есть у пользователя ОС postgres.

Другой вариант — написать функцию на недоверенном языке (таком, как PL/Perl — plperl), которая будет выполнять ту же задачу.

Различные языки серверного программирования рассматриваются в курсе DEV2.

<https://postgrespro.ru/docs/postgrespro/12/adminpack>

## Запись в файл: pg\_file\_write

Установим расширение:

```
=> CREATE EXTENSION adminpack;
```

```
CREATE EXTENSION
```

Теперь создадим процедуру write\_file, которая будет записывать отладочную информацию в файл. Пользователь postgres, запустивший экземпляр СУБД, должен иметь возможность записи в этот файл, поэтому расположим его в домашнем каталоге этого пользователя.

```
=> CREATE PROCEDURE write_file(message text)
AS $$
DECLARE
    filename CONSTANT text := '/var/lib/postgresql/log.txt';
    message text;
BEGIN
    message := format(E'%s, %s, %s\n',
        session_user, clock_timestamp()::text, write_file.message
    );
    PERFORM pg_file_write(filename, message, /* append */ true);
END;
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Функция записывает отдельной строкой в файл журнала сообщение, переданное параметром, вместе с информацией о том, кто и когда записал строку.

Создаем новый вариант long\_running.

```
=> CREATE OR REPLACE PROCEDURE long_running()
AS $$
BEGIN
    CALL write_file('long_running. Stage 1/3...');
    PERFORM pg_sleep(2);

    CALL write_file('long_running. Stage 2/3...');
    PERFORM pg_sleep(3);

    CALL write_file('long_running. Stage 3/3...');
    PERFORM pg_sleep(1);

    CALL write_file('long_running. Done.');
```

```
END;
```

```
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Для проверки запустим long\_running в отдельной транзакции, которую в конце откатим.

```
=> BEGIN;
```

```
BEGIN
```

```
=> CALL long_running();
```

```
CALL
```

```
=> ROLLBACK;
```

```
ROLLBACK
```

Проверим, что записи в журнале появились (от имени пользователя postgres операционной системы):

```
postgres$ cat /var/lib/postgresql/log.txt
```

```
student, 2021-01-12 20:34:30.315934+03, long_running. Stage 1/3...
student, 2021-01-12 20:34:32.318764+03, long_running. Stage 2/3...
student, 2021-01-12 20:34:35.320899+03, long_running. Stage 3/3...
student, 2021-01-12 20:34:36.32233+03, long_running. Done.
```

Чтобы пользователь student мог получить доступ к этому файлу, нужно использовать команду sudo.



## Стандартная трассировка в журнал сообщений

накладные расходы на запись в журнал  
большой размер файла журнала  
требуются инструменты для анализа  
нужен доступ к журналу (безопасность)

## Настройки

долго выполняющиеся команды	<i>log_min_duration_statement</i>
какие команды записывать	<i>log_statement</i>
контекст сообщения	<i>log_line_prefix</i>
...	

15

В некоторых случаях может оказаться полезной трассировка всего происходящего в коде. Штатными средствами можно получить в журнале сообщений сервера SQL-запросы сеансов. Надо учитывать:

- Высоконагруженное приложение может выполнять огромное число запросов. Их запись в файл журнала может оказывать влияние на производительность подсистемы ввода/вывода.
- В большинстве случаев требуются специальные инструменты для анализа такого объема данных. Стандарт де-факто — pgBadger.  
<https://github.com/darold/pgbadger>
- К файлу журнала на сервере может не быть доступа у разработчиков приложений. К тому же в журнале промышленной системе могут оказаться команды, содержащие конфиденциальную информацию.

Для настройки трассировки имеется ряд параметров, основные из которых приведены на слайде. Полный список:

<https://postgrespro.ru/docs/postgresql/12/runtime-config-logging>

Не обязательно устанавливать конфигурационные параметры для всего кластера. Их действие можно ограничить отдельными сеансами при помощи команд SET, ALTER DATABASE, ALTER ROLE (как рассказывалось в темах «Обзор базового инструментария. Установка и управление, psql» и «Организация данных. Логическая структура»).

## Расширение `auto_explain`

запись в журнал планов выполнения запросов  
трассировка вложенных запросов

## Настройки

планы долгих команд	<code>auto_explain.log_min_duration</code>
вложенные запросы	<code>auto_explain.log_nested_statements</code>
...	

При включении трассировки SQL-команды попадают в журнал в том виде, в каком они отправлены на сервер. Если была вызвана подпрограмма PL/pgSQL, то в журнале будет записан только вызов этой подпрограммы (например, оператор `SELECT` или `CALL`), но не будет тех команд, которые выполняются внутри подпрограммы.

Чтобы увидеть в журнале запросы не только верхнего уровня, но и вложенные, потребуется штатное расширение `auto_explain`.

Как следует из названия расширения, его основная задача — записывать в журнал не только текст команды, но и план ее выполнения. Это тоже может оказаться полезно, хотя относится не к трассировке как таковой, а к оптимизации запросов (которая рассматривается в курсе QPT).

<https://postgrespro.ru/docs/postgresql/12/auto-explain>

## Расширение `plpgsql_check`

накладные расходы на запись сообщений  
большой объем выдачи

## Основные настройки

включение трассировки	<code>plpgsql_check.enable_tracer</code>
	<code>plpgsql_check.tracer</code>
уровень сообщений	<code>plpgsql_check.tracer_errlevel</code>

Чтобы разобраться по журналу, какой процедурный код выполнялся, придется сопоставить SQL-запросы с подпрограммами на PL/pgSQL, а это может оказаться не просто. Стандартной возможности для трассировки PL/pgSQL-кода не предусмотрено, но расширение Павла Стехуле `plpgsql_check` (которое упоминалось в теме «PL/pgSQL. Выполнение запросов») позволяет ее выполнить.

Подобная трассировка вызывает большие накладные расходы и должна использоваться только для отладки, но не в промышленной эксплуатации.

[https://github.com/okbob/plpgsql\\_check](https://github.com/okbob/plpgsql_check)

## Трассировка сеансов

Простой пример включения трассировки — установка параметра `log_statement` в значение `all` (записывать все команды, включая DDL, модификацию данных и запросы).

```
=> SET log_statement = 'all';
```

SET

Выполним какой-нибудь запрос:

```
=> SELECT get_count('pg_views');
```

```
NOTICE: cmd: SELECT COUNT(*) FROM pg_views
get_count
-----
      124
(1 row)
```

И выключим трассировку:

```
=> RESET log_statement;
```

RESET

Информация о выполненных командах окажется в журнале сервера:

```
student$ tail -n 2 /var/log/postgresql/postgresql-12-main.log
```

```
2021-01-12 20:34:36.662 MSK [190182] student@plpgsql_debug LOG:  statement: SELECT get_count('pg_views');
2021-01-12 20:34:36.776 MSK [190182] student@plpgsql_debug LOG:  statement: RESET log_statement;
```

Однако в журнал попадает только команда верхнего уровня, но не запрос внутри функции `get_count`.

Воспользуемся расширением `auto_explain`. Это расширение не нужно устанавливать в базу данных, но требуется загрузить в память. Это можно сделать для всего экземпляра с помощью параметра `shared_preload_libraries`, либо только текущего процесса:

```
=> LOAD 'auto_explain';
```

LOAD

Установим трассировку всех команд независимо от длительности выполнения:

```
=> SET auto_explain.log_min_duration = 0;
```

SET

Включим трассировку вложенных запросов:

```
=> SET auto_explain.log_nested_statements = on;
```

SET

Сообщения выводятся с помощью того же механизма, что использует команда `RAISE`. По умолчанию используется уровень `LOG`, что обычно соответствует выводу в журнал. Изменив параметр, можно получать трассировку непосредственно в консоли:

```
=> SET auto_explain.log_level = 'NOTICE';
```

SET

Повторим запрос:

```
=> SELECT get_count('pg_views');
```

```
NOTICE: cmd: SELECT COUNT(*) FROM pg_views
NOTICE: duration: 0.268 ms plan:
Query Text: SELECT COUNT(*) FROM pg_views
Aggregate  (cost=18.25..18.26 rows=1 width=8)
  -> Seq Scan on pg_class c  (cost=0.00..17.94 rows=124 width=0)
    Filter: (relkind = 'v'::"char")
NOTICE: duration: 3.035 ms plan:
Query Text: SELECT get_count('pg_views');
Result  (cost=0.00..0.26 rows=1 width=8)
get_count
-----
      124
(1 row)
```

Видим не только вызов функции, но и вложенный запрос, вместе с планами выполнения.

PL/pgSQL Debugger — API отладчика, используется в графических средах разработки

Служебные сообщения — вывод на консоль, запись в журнал сообщений сервера, в таблицу или в файл, передача другим процессам

Возможность трассировки сеансов



1. Измените функцию `get_catalog` так, чтобы динамически формируемый текст запроса записывался в журнал сообщений сервера.  
В приложении выполните несколько раз поиск, заполняя разные поля, и убедитесь, что команды SQL формируются правильно.
2. Включите трассировку команд SQL на уровне сервера.  
Поработайте в приложении и проверьте, какие команды попадают в журнал сообщений.  
Выключите трассировку.

2. Для включения трассировки установите значение параметра `log_min_duration_statement` в 0 и перечитайте конфигурацию. В журнал будут записываться все команды и время их выполнения.

Проще всего это сделать командой `ALTER SYSTEM SET`. Другие способы рассматривались в теме «Обзор базового инструментария. Установка и управление, psql». Не забудьте перечитать конфигурационный файл.

После просмотра журнала следует вернуть значение параметра `log_min_duration_statement` в значение по умолчанию (-1), чтобы отключить трассировку. Удобный способ — команда `ALTER SYSTEM RESET`.

## 1. Функция get\_catalog

Текст динамического запроса формируем в отдельной переменной, которую перед выполнением запишем в журнал сервера. Для более полной информации включим в сообщение значения переданных в функцию параметров.

Отладочные строки в журнале можно найти по тексту «DEBUG get\_catalog».

После отладки команду RAISE LOG можно удалить или закомментировать.

```
=> CREATE OR REPLACE FUNCTION get_catalog(
    author_name text,
    book_title text,
    in_stock boolean
)
RETURNS TABLE(book_id integer, display_name text, onhand_qty integer)
AS $$
DECLARE
    title_cond text := '';
    author_cond text := '';
    qty_cond text := '';
    cmd text := '';
BEGIN
    IF book_title != '' THEN
        title_cond := format(
            ' AND cv.title ILIKE %L', '%' || book_title || '%'
        );
    END IF;
    IF author_name != '' THEN
        author_cond := format(
            ' AND cv.authors ILIKE %L', '%' || author_name || '%'
        );
    END IF;
    IF in_stock THEN
        qty_cond := ' AND cv.onhand_qty > 0';
    END IF;
    cmd := '
        SELECT cv.book_id,
               cv.display_name,
               cv.onhand_qty
        FROM   catalog_v cv
        WHERE  true'
        || title_cond || author_cond || qty_cond || '
        ORDER BY display_name';

    RAISE LOG 'DEBUG get_catalog (%, %, %): %',
        author_name, book_title, in_stock, cmd;
    RETURN QUERY EXECUTE cmd;
END;
$$ STABLE LANGUAGE plpgsql;
```

CREATE FUNCTION

## 2. Включение и выключение трассировки SQL-запросов

Чтобы включить трассировку всех запросов на уровне сервера, можно выполнить:

```
=> ALTER SYSTEM SET log_min_duration_statement = 0;
```

ALTER SYSTEM

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf
-----
t
(1 row)
```

Чтобы выключить:

```
=> ALTER SYSTEM RESET log_min_duration_statement;
```

ALTER SYSTEM

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf
-----
t
(1 row)
```

Последние две команды попали в журнал сообщений:

```
student$ tail -n 6 /var/log/postgresql/postgresql-12-main.log
```

```
2021-01-12 20:36:48.805 MSK [196592] LOG:  received SIGHUP, reloading configuration files
2021-01-12 20:36:48.806 MSK [196592] LOG:  parameter "log_min_duration_statement" changed to "0"
2021-01-12 20:36:48.925 MSK [206519] student@bookstore LOG:  duration: 7.516 ms statement: ALTER SYSTEM RESET log_min_duration_statement;
2021-01-12 20:36:48.974 MSK [196592] LOG:  received SIGHUP, reloading configuration files
2021-01-12 20:36:48.975 MSK [196592] LOG:  parameter "log_min_duration_statement" removed from configuration file, reset to default
2021-01-12 20:36:48.983 MSK [206519] student@bookstore LOG:  duration: 10.055 ms statement: SELECT pg_reload_conf();
```



1. Включите трассировку PL/pgSQL-кода средствами расширения `plpgsql_check` и проверьте ее работу на примере несколько подпрограмм, вызывающих одна другую.
2. При выводе отладочных сообщений из PL/pgSQL-кода удобно понимать, к какой подпрограмме они относятся. В демонстрации имя функции выводилось вручную. Реализуйте функционал, автоматически добавляющий к тексту сообщений имя текущей функции или процедуры.

1. Для включения трассировки загрузите расширение `plpgsql_check` в память сеанса командой `LOAD`, затем установите в сеансе оба параметра `plpgsql_check.enable_tracer` и `plpgsql_check.tracer` в значение «on».

2. Имя подпрограммы можно получить, разобрав стек вызовов. Воспользуйтесь результатами практического задания 3 к теме «Обработка ошибок».

## 1. Трассировка с помощью plpgsql\_check

```
=> CREATE DATABASE plpgsql_debug;
```

CREATE DATABASE

```
=> \c plpgsql_debug
```

You are now connected to database "plpgsql\_debug" as user "student".

Загрузим расширение (в данном случае устанавливать его в базу данных командой CREATE EXTENSION не нужно):

```
=> LOAD 'plpgsql_check';
```

LOAD

Включим трассировку:

```
=> SET plpgsql_check.enable_tracer = on;
```

SET

```
=> SET plpgsql_check.tracer = on;
```

SET

Несколько функций, вызывающих друг друга:

```
=> CREATE FUNCTION foo(n integer) RETURNS integer
AS $$
BEGIN
    RETURN bar(n-1);
END;
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

```
=> CREATE FUNCTION bar(n integer) RETURNS integer
AS $$
BEGIN
    RETURN baz(n-1);
END;
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

```
=> CREATE FUNCTION baz(n integer) RETURNS integer
AS $$
BEGIN
    RETURN n;
END;
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

Пример работы трассировки:

```
=> SELECT foo(3);
```

```
NOTICE: #0  -> start of function foo(integer) (oid=24851)
NOTICE: #0      "n" => '3'
NOTICE: #1  -> start of function bar(integer) (oid=24852)
NOTICE: #1      call by foo(integer) line 3 at RETURN
NOTICE: #1      "n" => '2'
NOTICE: #2  -> start of function baz(integer) (oid=24853)
NOTICE: #2      call by bar(integer) line 3 at RETURN
NOTICE: #2      "n" => '1'
NOTICE: #2      <<- end of function baz (elapsed time=0.113 ms)
NOTICE: #1      <<- end of function bar (elapsed time=0.540 ms)
NOTICE: #0  <<- end of function foo (elapsed time=1.282 ms)
foo
-----
      1
(1 row)
```

Выводятся не только события начала и окончания работы функций, но и значения параметров, а также затраченное время (в расширении есть и возможность профилирования, которую мы не рассматриваем).

Выключим трассировку:

```
=> SET plpgsql_check.tracer = off;
```

SET

## 2. Имя функции в отладочных сообщениях

Напишем процедуру, которая выводит верхушку стека вызовов (за исключением самой процедуры трассировки). Сообщение выводится с отступом, который соответствует глубине стека.

```
=> CREATE PROCEDURE raise_msg(msg text)
AS $$
DECLARE
    ctx text;
    stack text[];
BEGIN
    GET DIAGNOSTICS ctx = pg_context;
    stack := regexp_split_to_array(ctx, E'\n');
    RAISE NOTICE '%: %',
        repeat('. ', array_length(stack,1)-2) || stack[3], msg;
END;
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

Пример работы трассировки:

```
=> CREATE TABLE t(n integer);
```

CREATE TABLE

```
=> CREATE FUNCTION on_insert() RETURNS trigger
AS $$
BEGIN
    CALL raise_msg('NEW = ' || NEW::text);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

```
=> CREATE TRIGGER t_before_row
BEFORE INSERT ON t
FOR EACH ROW
EXECUTE FUNCTION on_insert();
```

CREATE TRIGGER

```
=> CREATE PROCEDURE insert_into_t()
AS $$
BEGIN
    CALL raise_msg('start');
    INSERT INTO t SELECT id FROM generate_series(1,3) id;
    CALL raise_msg('end');
END;
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

```
=> CALL insert_into_t();
```

```
NOTICE: . PL/pgSQL function insert_into_t() line 3 at CALL: start
NOTICE: . . . PL/pgSQL function on_insert() line 3 at CALL: NEW = (1)
NOTICE: . . . PL/pgSQL function on_insert() line 3 at CALL: NEW = (2)
NOTICE: . . . PL/pgSQL function on_insert() line 3 at CALL: NEW = (3)
NOTICE: . PL/pgSQL function insert_into_t() line 5 at CALL: end
CALL
```