

Расширяемость Создание расширений



Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

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

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.

Расширения в PostgreSQL

Создание расширений

Версии расширений и обновление

Особенности работы утилиты pg_dump

Расширяемость PostgreSQL

Возможности

- функции и языки программирования
- типы данных, операторы, методы доступа
- обертки сторонних данных (FDW)

Механизмы

- изменяемый системный каталог
- API для подключения внешних обработчиков
- загрузка и выполнение пользовательского кода

3

Расширяемость — важнейшая черта PostgreSQL — это возможность подключать «на лету» новый функционал без изменения кода сервера.

Таким образом можно добавлять языки программирования и разрабатывать на них функции, определять новые типы данных и операторы для работы с ними, создавать новые методы доступа для типов данных, разрабатывать обертки сторонних данных для подключения к внешним источникам.

Для того чтобы это было возможным, системный каталог PostgreSQL хранит большое количество информации об объектах БД. Эта информация не зашита жестко в код сервера. Пользователи могут изменять содержимое таблиц системного каталога, тем самым добавляя новые объекты и связанный с ними функционал.

Кроме того, в исходном коде PostgreSQL встроено большое количество хуков и различных API для подключения пользовательских функций. Это дает возможность разрабатывать такие расширения как `pg_stat_statements`, `auto_explain`, `pldebugger` и многие, многие другие.

Завершает картину возможность загружать в серверные процессы пользовательский код. Например, можно написать разделяемую библиотеку и подключать ее по ходу работы.

<https://postgrespro.ru/docs/postgresql/16/extend-how>

В качестве предостережения следует отметить, что выполнение процессами сервера неправильно написанного пользовательского кода может привести к катастрофическим последствиям. Следует доверять только проверенному коду из надежных источников.

Группа взаимосвязанных объектов БД

- установка всех объектов одной командой
- невозможность удалить отдельный объект
- сохранение связи при выгрузке с помощью `pg_dump`
- инструменты для перехода на новую версию

Источники

- в составе дистрибутива (`contrib`)
- внешние расширения
- возможность создания собственных расширений

Бывают ситуации, когда несколько объектов базы данных логически связаны между собой. Например, несколько типов данных, функции и операторы для работы с ними, классы операторов. Такую связь можно сделать явной с помощью механизма расширений.

Это облегчает управление объектами:

- все объекты устанавливаются одной командой;
- невозможно удалить отдельный объект — расширение можно удалить только полностью;
- связь между объектами сохраняется и при создании резервной копии с помощью утилиты `pg_dump`;
- есть инструменты для управления версиями расширений.

В состав PostgreSQL входит значительное количество полезных расширений, частью из которых мы уже пользовались.

<https://postgrespro.ru/docs/postgresql/16/contrib>

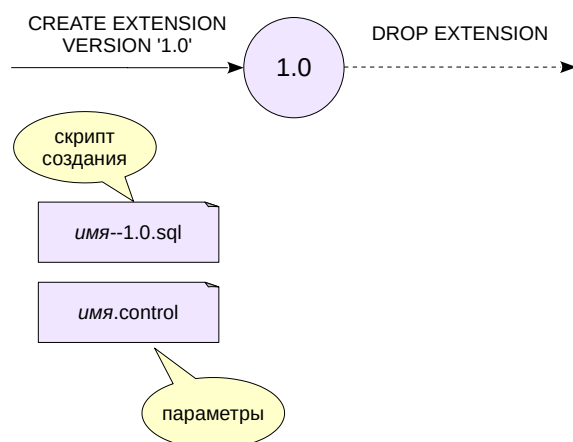
<https://postgrespro.ru/docs/postgresql/16/contrib-prog>

Другой источник — PostgreSQL Extension Network (PGXN) — сеть расширений по аналогии с CPAN для Perl: <https://pgxn.org/>

Расширения могут распространяться и другими способами, в том числе через пакетные репозитории дистрибутивов ОС.

В этой теме мы рассмотрим, как создавать собственные расширения.

Создание расширения



Расширение устанавливается в базу данных командой CREATE EXTENSION. При этом должны существовать два файла:

- **управляющий файл** «имя.control» с параметрами расширения;
- **скрипт создания** объектов расширения «имя--версия.sql».

Версия традиционно имеет вид «1.0», «1.1» и т. д., но это не обязательно: ее имя может состоять из любых символов (но не должно содержать «--» и начинаться или заканчиваться на «--»).

Обычно версию не указывают в команде CREATE EXTENSION, поскольку текущая актуальная версия записана в управляющем файле (параметр default_version) и используется по умолчанию.

Другие параметры расширения указывают зависимости от других расширений (requires), возможность перемещения объектов расширения между схемами (relocatable), возможность установки только суперпользователем (superuser) и др.

<https://postgrespro.ru/docs/postgresql/16/extend-extensions>

<https://postgrespro.ru/docs/postgresql/16/sql-createextension>

Расширение удаляется командой DROP EXTENSION. Скрипт для удаления объектов писать не нужно.

<https://postgrespro.ru/docs/postgresql/16/sql-dropextension>

Создание расширения

Создадим простое расширение — конвертер единиц измерения, и назовем его uom (units of measure).

Начнем с каталога, в котором будем создавать необходимые файлы:

```
student$ mkdir /home/student/tmp/uom
```

Сначала создадим управляющий файл с настройками (мы можем сделать это в любом текстовом редакторе).

- default_version определяет версию по умолчанию, без этого параметра версию придется указывать явно;
- relocatable говорит о том, что расширение можно перемещать из схемы в схему (мы поговорим об этом чуть позже);
- encoding требуется, если используются символы, отличные от ASCII;
- comment определяет комментарий к расширению.

```
/home/student/tmp/uom/uom.control
```

```
default_version = '1.0'
relocatable = true
encoding = UTF8
comment = 'Единицы измерения'
```

Это не все возможные параметры; полный список можно узнать из документации.

Теперь займемся файлом с командами, создающими объекты расширения.

- Первая строка файла предотвращает случайный запуск скрипта вручную.
- Все команды будут выполнены в одной транзакции — неявном блоке BEGIN ... END. Поэтому команды управления транзакциями (и служебные команды, такие, как VACUUM) здесь не допускаются.
- Путь поиска (параметр search_path) будет установлен на единственную схему — ту, в которой создаются объекты расширения.

```
/home/student/tmp/uom/uom--1.0.sql
```

```
\echo Use "CREATE EXTENSION uom" to load this file. \quit

-- Справочник единиц измерения
CREATE TABLE uoms (
    uom text PRIMARY KEY,
    k numeric NOT NULL
);
GRANT SELECT ON uoms TO public;
INSERT INTO uoms(uom,k) VALUES ('м',1), ('км',1000), ('см',0.01);

-- Функция для перевода значения из одной единицы в другую
CREATE FUNCTION convert(value numeric, uom_from text, uom_to text) RETURNS numeric
LANGUAGE sql STABLE STRICT
RETURN convert.value *
    (SELECT k FROM uoms WHERE uom = convert.uom_from) /
    (SELECT k FROM uoms WHERE uom = convert.uom_to);
```

Чтобы PostgreSQL нашел созданные нами файлы, они должны оказаться в каталоге SHAREDIR/extension. Значение SHAREDIR можно узнать так:

```
student$ pg_config --sharedir
```

```
/usr/share/postgresql/16
```

Например, посмотрим на файлы расширения pg_background:

```
student$ ls `pg_config --sharedir`/extension/pg_background*
```

```
/usr/share/postgresql/16/extension/pg_background--1.0--1.2.sql
/usr/share/postgresql/16/extension/pg_background--1.1--1.2.sql
/usr/share/postgresql/16/extension/pg_background--1.2.sql
/usr/share/postgresql/16/extension/pg_background.control
```

Конечно, файлы расширения можно скопировать вручную, но стандартный способ — воспользоваться утилитой make. Ей понадобится Makefile, который должен выглядеть, как показано ниже.

- Переменная EXTENSION задает имя расширения;
- Переменная DATA определяет список файлов, которые надо скопировать в SHAREDIR (кроме управляющего);
- Последние строки не меняются. Они подключают специальный Makefile для расширений, который содержит всю необходимую логику сборки и установки. Важно, чтобы утилита pg_config была доступна — иначе

неизвестны пути, по которым установлен PostgreSQL.

/home/student/tmp/uom/Makefile

```
EXTENSION = uom
DATA = uom--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Теперь выполним `make install` в каталоге расширения:

```
student$ sudo make install -C /home/student/tmp/uom
```

```
make: Entering directory '/home/student/tmp/uom'
/bin/mkdir -p '/usr/share/postgresql/16/extension'
/bin/mkdir -p '/usr/share/postgresql/16/extension'
/usr/bin/install -c -m 644 ./uom.control '/usr/share/postgresql/16/extension/'
/usr/bin/install -c -m 644 ./uom--1.0.sql '/usr/share/postgresql/16/extension/'
make: Leaving directory '/home/student/tmp/uom'
```

Создадим базу данных и подключимся к ней:

```
=> CREATE DATABASE ext_extensions;
```

```
CREATE DATABASE
```

```
=> \c ext_extensions
```

You are now connected to database "ext_extensions" as user "student".

Проверим, доступно ли наше расширение?

```
=> SELECT * FROM pg_available_extensions WHERE name = 'uom';
```

name	default_version	installed_version	comment
uom	1.0		Единицы измерения

(1 row)

Попробуем создать в новой базе расширение uom:

```
=> CREATE EXTENSION uom;
```

```
CREATE EXTENSION
```

Мы не указали версию, поэтому было взято значение из управляющего файла (1.0).

```
=> SELECT * FROM uoms;
```

uom	k
м	1
км	1000
см	0.01

(3 rows)

```
=> SELECT convert(2, 'км', 'м');
```

convert
2000.0000000000000000

(1 row)

Все работает.

Само расширение не относится к какой-либо схеме, но объекты расширения — относятся. В какой схеме они созданы?

```
=> \dt uoms
```

```
      List of relations
 Schema | Name | Type  | Owner
-----+-----+-----+-----
 public | uoms | table | student
(1 row)
```

Объекты установлены в схему, в которой они были бы созданы по умолчанию; в данном случае — `public`. При создании расширения мы можем указать эту схему явно:

```
CREATE EXTENSION uom SCHEMA public;
```

Поскольку мы указали в управляющем файле, что расширение переносимо (relocatable), его можно переместить в другую схему:

```
=> CREATE SCHEMA uom;
```

```
CREATE SCHEMA
```

```
=> ALTER EXTENSION uom SET SCHEMA uom;
```

```
ALTER EXTENSION
```

Теперь все объекты находятся в схеме uom:

```
=> \dt uom.*
```

```
      List of relations
 Schema | Name | Type  | Owner
-----+-----+-----+-----
 uom    | uoms | table | student
(1 row)
```

```
=> \df uom.*
```

```
      List of functions
 Schema | Name   | Result data type | Argument data types | Type
-----+-----+-----+-----+-----
 uom    | convert | numeric          | value numeric, uom_from text, uom_to text | func
(1 row)
```

Можно ли прочитать данные из таблицы без указания схемы, где она расположена?

```
=> SELECT * FROM uoms;
```

```
ERROR: relation "uoms" does not exist
LINE 1: SELECT * FROM uoms;
                        ^
```

Нет, потому что теперь таблица не находится в пути поиска.

А будет ли работать функция, если при ее вызове мы явно укажем схему? Напомним, что в определении тела функции обращение к таблице не включало название схемы.

```
=> SELECT uom.convert(2, 'км', 'м');
```

```
      convert
-----
 2000.000000000000000000
(1 row)
```

Да, потому что код функции был оформлен в современном стиле стандарта SQL, а значит еще на этапе создания был выполнен его разбор и теперь обращение к таблице производится по ее идентификатору, а не по символическому имени.

Позаботимся о путях поиска; данные из таблицы читаются:

```
=> SET search_path = uom, public;
```

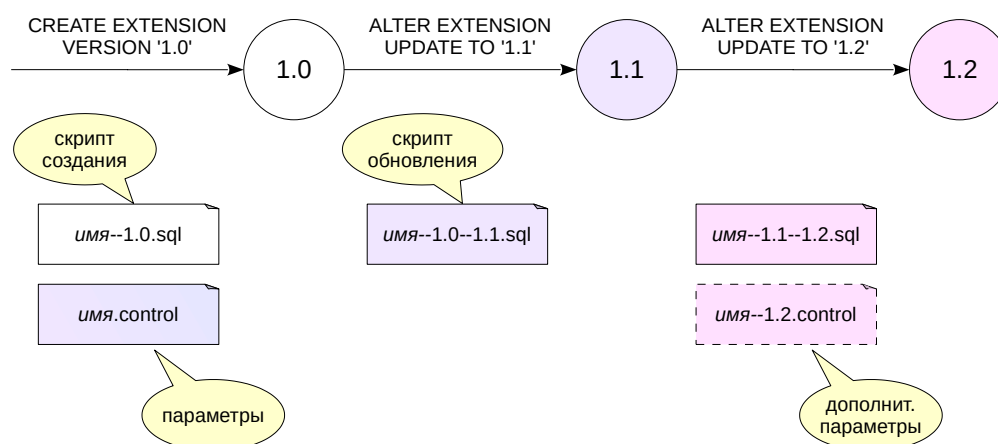
```
SET
```

```
=> SELECT * FROM uoms;
```

```
 uom | k
-----+-----
  м   | 1
 км   | 1000
 см   | 0.01
(3 rows)
```

Отметим, что некоторые расширения не допускают перемещения, но это бывает нечасто.

Обновление расширения



7

Обновление версии расширения выполняется командой `ALTER EXTENSION UPDATE`. При этом должен существовать **скрипт обновления** «имя--старая-версия--новая-версия.sql», содержащий необходимые для обновления команды.

Также необходимо изменить управляющий файл «имя.control», обновив актуальную версию и, возможно, другие параметры.

При необходимости может существовать и отдельный управляющий файл, привязанный к версии. Например, если в версии 1.2 появилась зависимость от другого расширения, то эту зависимость неправильно указывать в основном управляющем файле. Параметры, указанные в дополнительном управляющем файле, более приоритетны, чем параметры основного управляющего файла.

В примере, приведенном на слайде, имеются скрипты обновления 1.0 → 1.1 и 1.1 → 1.2. Можно создать и скрипт 1.0 → 1.2, но, как правило, это не требуется: механизм расширений сам берет на себя выбор пути с учетом доступных переходов между версиями. Например, если установлена версия 1.0, то ее можно обновить сразу до 1.2: сначала автоматически применится скрипт 1.0 → 1.1, а затем 1.1 → 1.2.

Как и при создании, при обновлении номер версии обычно не указывают — в этом случае обновление происходит до последней актуальной версии, записанной в основном управляющем файле.

<https://postgrespro.ru/docs/postgresql/16/sql-alterextension>

Версии расширения и обновление

При некотором размышлении мы можем сообразить, что не любые единицы допускают преобразование. Например, метры нельзя пересчитать в килограммы. Создадим версию 1.1 нашего расширения, которая это учитывает.

В управляющем файле исправим версию на 1.1:

/home/student/tmp/uom/uom.control

```
default_version = '1.1'
relocatable = true
encoding = UTF8
comment = 'Единицы изменения'
```

И создадим файл с командами для обновления:

/home/student/tmp/uom/uom--1.0--1.1.sql

```
\echo Use "CREATE EXTENSION uom" to load this file. \quit

-- Все, что было, отнесем к мерам длины
ALTER TABLE uoms ADD uom_class text NOT NULL DEFAULT 'длина';

-- Добавим единицы измерения массы
INSERT INTO uoms(uom,k,uom_class) VALUES
    ('г', 1, 'масса'), ('кг', 1_000, 'масса'),
    ('ц', 100_000, 'масса'), ('т', 1_000_000, 'масса');

-- Функция для перевода значения из одной единицы в другую
CREATE OR REPLACE FUNCTION convert(
    value numeric,
    uom_from text,
    uom_to text
)
RETURNS numeric AS $$
DECLARE
    uoms_from uoms;
    uoms_to uoms;
BEGIN
    SELECT * INTO uoms_from FROM uoms WHERE uom = convert.uom_from;
    SELECT * INTO uoms_to FROM uoms WHERE uom = convert.uom_to;
    IF uoms_from.uom_class != uoms_to.uom_class THEN
        RAISE EXCEPTION 'Невозможно преобразовать : % -> %',
            uoms_from.uom_class, uoms_to.uom_class;
    END IF;
    RETURN convert.value * uoms_from.k / uoms_to.k;
END;
$$ LANGUAGE plpgsql STABLE STRICT;
```

Добавим в Makefile новый файл в список DATA:

/home/student/tmp/uom/Makefile

```
EXTENSION = uom
DATA = uom--1.0.sql uom--1.0--1.1.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Выполним make install, чтобы разместить файлы расширения:

```
student$ sudo make install -C /home/student/tmp/uom
```

```
make: Entering directory '/home/student/tmp/uom'
/bin/mkdir -p '/usr/share/postgresql/16/extension'
/bin/mkdir -p '/usr/share/postgresql/16/extension'
/usr/bin/install -c -m 644 ./uom.control '/usr/share/postgresql/16/extension/'
/usr/bin/install -c -m 644 ./uom--1.0.sql ./uom--1.0--1.1.sql
'/usr/share/postgresql/16/extension/'
make: Leaving directory '/home/student/tmp/uom'
```

Какие версии расширения нам доступны?

```
=> SELECT name, version, installed
FROM pg_available_extension_versions
WHERE name = 'uom';

name | version | installed
-----+-----+-----
uom  | 1.0     | t
uom  | 1.1     | f
(2 rows)
```

Какие пути обновления доступны?

```
=> SELECT * FROM pg_extension_update_paths('uom');

source | target | path
-----+-----+-----
1.0     | 1.1     | 1.0--1.1
1.1     | 1.0     |
(2 rows)
```

Очевидно, путь один. Заметьте, что если бы мы создали файл «uom--1.1--1.0.sql», можно было бы «понизить версию». Для механизма расширений имена версий ничего не значат.

Выполним обновление:

```
=> ALTER EXTENSION uom UPDATE;
```

ALTER EXTENSION

Теперь нам доступен новый функционал:

```
=> SELECT convert(2, 'ц', 'кг');
```

```
convert
-----
200.0000000000000000
(1 row)
```

```
=> SELECT convert(1, 'м', 'кг');
```

```
ERROR:  Невозможно преобразовать : длина -> масса
CONTEXT:  PL/pgSQL function convert(numeric,text,text) line 9 at RAISE
```

Утилита pg_dump

Что попадает в резервную копию базы данных, созданную с помощью утилиты pg_dump?

```
student$ pg_dump ext_extensions | grep -v '^--'
```

```
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;
```

```
CREATE SCHEMA uom;
```

```
ALTER SCHEMA uom OWNER TO student;
```

```
CREATE EXTENSION IF NOT EXISTS uom WITH SCHEMA uom;
```

```
COMMENT ON EXTENSION uom IS 'Единицы измерения';
```

- Вначале идут установки различных параметров сервера;
- Объекты расширения не попадают в резервную копию, вместо этого выполняется команда CREATE EXTENSION — это позволяет сохранить зависимости между объектами.

В процессе работы с расширением пользователь может захотеть расширить справочник единиц измерения:

```
=> INSERT INTO uomс(uom,k,uom_class) VALUES
    ('верста',1066.8,'длина'), ('сажень',2.1336,'длина');
```

```
INSERT 0 2
```

Что теперь попадает в резервную копию?

```
student$ pg_dump ext_extensions | grep -v '^--'
```

```
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;
```

```
CREATE SCHEMA uom;
```

```
ALTER SCHEMA uom OWNER TO student;
```

```
CREATE EXTENSION IF NOT EXISTS uom WITH SCHEMA uom;
```

```
COMMENT ON EXTENSION uom IS 'Единицы измерения';
```

Сделанные пользователем изменения будут потеряны.

Но этого можно избежать, если мы сможем разделить предустановленные значения и пользовательские. Подготовим версию 1.2 расширения.

```
=> DELETE FROM uomс WHERE uom IN ('верста', 'сажень');
```

```
DELETE 2
```

В управляющем файле исправим версию на 1.2:

```
/home/student/tmp/uom/uom.control
```

```
default_version = '1.2'
relocatable = true
encoding = UTF8
comment = 'Единицы измерения'
```

Создадим файл с командами для обновления. Вызов функции pg_extension_config_dump определяет, какие строки таблицы требуют выгрузки.

```
/home/student/tmp/uom/uom--1.1--1.2.sql
```

```
\echo Use "CREATE EXTENSION uom" to load this file. \quit

-- Добавляем признак предустановленных данных
ALTER TABLE uomс ADD seeded boolean NOT NULL DEFAULT false;
UPDATE uomс SET seeded = true;

SELECT pg_extension_config_dump('uomс', 'WHERE NOT seeded');
```

Добавим в Makefile новый файл в список DATA:

```
/home/student/tmp/uom/Makefile
```

```
EXTENSION = uom
DATA = uom--1.0.sql uom--1.0--1.1.sql uom--1.1--1.2.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Выполним make install, чтобы разместить файлы расширения:

```
student$ sudo make install -C /home/student/tmp/uom
```

```
make: Entering directory '/home/student/tmp/uom'
/bin/mkdir -p '/usr/share/postgresql/16/extension'
/bin/mkdir -p '/usr/share/postgresql/16/extension'
/usr/bin/install -c -m 644 ./uom.control '/usr/share/postgresql/16/extension/'
/usr/bin/install -c -m 644 ./uom--1.0.sql ./uom--1.0--1.1.sql ./uom--1.1--1.2.sql
'/usr/share/postgresql/16/extension/'
make: Leaving directory '/home/student/tmp/uom'
```

И выполним обновление:

```
=> ALTER EXTENSION uom UPDATE;
```

```
ALTER EXTENSION
```

Повторим эксперимент:

```
=> INSERT INTO uoms(uom, k, uom_class) VALUES
    ('верста', 1066.8, 'длина'), ('сажень', 2.1336, 'длина');
```

```
INSERT 0 2
```

Что теперь попадает в резервную копию?

```
student$ pg_dump ext_extensions | grep -v '^--'
```

```
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;
```

```
CREATE SCHEMA uom;
```

```
ALTER SCHEMA uom OWNER TO student;
```

```
CREATE EXTENSION IF NOT EXISTS uom WITH SCHEMA uom;
```

```
COMMENT ON EXTENSION uom IS 'Единицы измерения';
```

```
COPY uom.uoms (uom, k, uom_class, seeded) FROM stdin;
верста 1066.8 длина f
сажень 2.1336 длина f
\.
```

На этот раз все правильно: после создания расширения в таблицу добавляются строки, созданные пользователем.

Расширения — упаковка взаимосвязанных объектов БД, предназначенных для решения какой-либо задачи

Расширения упрощают использование функционала

Имеются средства для разработки собственных расширений



1. Создайте расширение bookfmt, содержащее все необходимое для работы с типом данных для формата изданий. Установите расширение, чтобы объединить имеющиеся в базе данных разрозненные объекты.
2. Воспользуйтесь стандартным расширением isn для того чтобы проверить корректность кодов ISBN у имеющихся в магазине книг.

1. Объекты базы данных, относящиеся к книжному формату — тип данных `book_format`, приведение типов, функции, операторы и класс операторов — мы создавали в темах «Пользовательские типы данных» и «Классы операторов».

Если создать расширение так, как это показывалось в демонстрации, то для его установки придется удалить из базы существующие объекты. Что, конечно, нежелательно для работающей системы.

Поэтому сначала создайте и установите пустое расширение, а затем напишите скрипт для его обновления, добавляющий к расширению уже имеющиеся в базе данных объекты с помощью команды `ALTER EXTENSION ADD`

2. В достаточно старых книгах используется 10-значный код ISBN. С 2007 года для совместимости со штрихкодами используют 13-значный код ISBN (первые три цифры всегда равны 978).

Код имеет формат: *978 страна изд-во издание контрольная_цифра*.

Между группами цифр обычно ставятся дефисы, но это незначащий символ. Контрольная цифра вычисляется по специальным правилам.

В расширении `isn` имеются функции `isbn(text)` для 10-значного кода и `isbn13(text)` для 13-значного. Если контрольная цифра неверна, функции вызывают ошибку.

1. Расширения для книжного формата

Сначала создадим «пустое» расширение без объектов.

```
student$ mkdir bookfmt
```

bookfmt/bookfmt.control

```
default_version = '0'
relocatable = true
encoding = UTF8
comment = 'Формат издания'
```

bookfmt/bookfmt--0.sql

```
\echo Use "CREATE EXTENSION bookfmt" to load this file. \quit
```

bookfmt/Makefile

```
EXTENSION = bookfmt
DATA = bookfmt--0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

```
student$ sudo make install -C bookfmt
```

```
make: Entering directory '/home/student/bookfmt'
/bin/mkdir -p '/usr/share/postgresql/16/extension'
/bin/mkdir -p '/usr/share/postgresql/16/extension'
/usr/bin/install -c -m 644 ./bookfmt.control '/usr/share/postgresql/16/extension/'
/usr/bin/install -c -m 644 ./bookfmt--0.sql '/usr/share/postgresql/16/extension/'
make: Leaving directory '/home/student/bookfmt'
```

Установим расширение.

```
=> CREATE EXTENSION bookfmt;
```

CREATE EXTENSION

Теперь напишем скрипт для обновления, в котором добавим в расширение уже существующие в базе объекты.

bookfmt/bookfmt.control

```
default_version = '1.0'
relocatable = true
encoding = UTF8
comment = 'Формат издания'
```

bookfmt/bookfmt--0--1.0.sql

```
\echo Use "CREATE EXTENSION bookfmt" to load this file. \quit

ALTER EXTENSION bookfmt ADD TYPE book_format;
ALTER EXTENSION bookfmt ADD FUNCTION book_format_to_text;
ALTER EXTENSION bookfmt ADD CAST (book_format AS text);
ALTER EXTENSION bookfmt ADD FUNCTION book_format_area;
ALTER EXTENSION bookfmt ADD FUNCTION book_format_cmp;
ALTER EXTENSION bookfmt ADD FUNCTION book_format_lt;
ALTER EXTENSION bookfmt ADD OPERATOR < (book_format, book_format);
ALTER EXTENSION bookfmt ADD FUNCTION book_format_le;
ALTER EXTENSION bookfmt ADD OPERATOR <= (book_format, book_format);
ALTER EXTENSION bookfmt ADD FUNCTION book_format_eq;
ALTER EXTENSION bookfmt ADD OPERATOR = (book_format, book_format);
ALTER EXTENSION bookfmt ADD FUNCTION book_format_gt;
ALTER EXTENSION bookfmt ADD OPERATOR > (book_format, book_format);
ALTER EXTENSION bookfmt ADD FUNCTION book_format_ge;
ALTER EXTENSION bookfmt ADD OPERATOR >= (book_format, book_format);
ALTER EXTENSION bookfmt ADD OPERATOR CLASS book_format_ops USING btree;
```

bookfmt/Makefile

```
EXTENSION = bookfmt
DATA = bookfmt--0.sql bookfmt--0--1.0.sql

PG_CONFIG = pg_config
```



```
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

```
student$ sudo make install -C bookfmt
```

```
make: Entering directory '/home/student/bookfmt'
/bin/mkdir -p '/usr/share/postgresql/16/extension'
/bin/mkdir -p '/usr/share/postgresql/16/extension'
/usr/bin/install -c -m 644 ../bookfmt.control '/usr/share/postgresql/16/extension/'
/usr/bin/install -c -m 644 ../bookfmt--0.sql ../bookfmt--0--1.0.sql
'/usr/share/postgresql/16/extension/'
make: Leaving directory '/home/student/bookfmt'
```

Выполним обновление:

```
=> ALTER EXTENSION bookfmt UPDATE;
```

```
ALTER EXTENSION
```

Теперь все объекты объединены в расширение, которые мы при необходимости сможем развивать.

А для того чтобы расширением могли воспользоваться другие, надо подготовить файл «bookfmt--1.0.sql» с командами, создающими все необходимые объекты.

2. Проверка кодов ISBN

Установим расширение:

```
=> CREATE EXTENSION isn;
```

```
CREATE EXTENSION
```

```
=> DO $$
DECLARE
    b_id bigint;
    i text;
    i10 isbn;
    i13 isbn13;
BEGIN
    FOR b_id, i IN SELECT book_id, additional->'ISBN' FROM books LOOP
        BEGIN
            IF length(translate(i, '-', '')) = 10 THEN
                i10 := isbn(i);
            ELSE
                i13 := isbn13(i);
            END IF;
        EXCEPTION
            WHEN others THEN
                RAISE NOTICE 'book_id=?: %', b_id, sqlerrm;
        END;
    END LOOP;
END;
$$;
```

```
NOTICE: book_id=20: invalid check digit for ISBN number: "5-7490-0068-9", should be 0
```

```
NOTICE: book_id=61: invalid input syntax for ISBN number: "5-9000242-17-x"
```

```
NOTICE: book_id=19: invalid check digit for ISBN number: "5-7490-0065-1", should be 6
```

```
DO
```

Проблемные данные часто встречаются в реальных системах. В данном случае они могут быть вызваны не только ошибками при вводе, но и неправильно указанным кодом в самой книге. Расширение isn имеет возможность работы в нестрогом режиме, допуская ошибки (но предупреждая о них).

1. Создайте расширение с функцией для подготовки текста к публикации. Функция должна применить к текстовому параметру последовательность правил, выполняющих замену по регулярному выражению, и вернуть результат. Правила должны храниться в таблице и применяться в порядке их вставки. К предопределенным правилам пользователь может добавлять собственные. Работа функции не должна зависеть от настройки пути поиска, но должна позволять выбрать схему при установке.
2. Установите расширение в схему `туро`, добавьте в таблицу пользовательское правило. Корректно ли выгружает копию базы данных утилита `pg_dump`? Проверьте возможность добавить правило после восстановления из резервной копии.

11

1. В качестве предопределенных правил используйте, например:

`(^|s)"(S) → \1«\2`

`(S)"(s|$) → \1»\2`

`(^|s)-(s|$) → \1—\2`

Эти правила заменяют (не всегда корректно) обычные символы на кавычки-елочки и тире:

- Буквы "р" нет, - сказал я. → — Буквы «р» нет, — сказал я.

Чтобы обеспечить независимость от пути поиска, расширение придется сделать переносимым. Укажите соответствующий параметр в управляющем файле, а в теле функции используйте макрос `@extschema@` для указания схемы, в которой находится таблица правил. Этот макрос будет заменен на выбранную схему при установке.

2. Чтобы утилита `pg_dump` правильно выгружала пользовательские правила, используйте функцию `pg_extension_config_dump` и для самой таблицы, и для последовательности, созданной для первичного ключа.

При вызове `pg_dump` можно указать ключи `--clean` и `--create`, чтобы копия включала в себя команды для удаления и создания базы данных.

1. Расширение для подготовки текста

```
student$ mkdir typo
```

В управляющем файле указываем relocatable = false:

```
typo/typo.control
```

```
default_version = '1.0'
relocatable = false
encoding = UTF8
comment = 'Подготовка текста по настраиваемым правилам'
```

Создаем файл с командами.

```
typo/typo--1.0.sql
```

```
\echo Use "CREATE EXTENSION typo" to load this file. \quit

-- В таблице предусматриваем столбец seeded, чтобы отличать предустановленные правила от
пользовательских.

CREATE TABLE typo_rules (
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    pattern text NOT NULL,
    replace_by text NOT NULL,
    seeded boolean DEFAULT false
);
GRANT SELECT ON typo_rules TO public;
INSERT INTO typo_rules(pattern, replace_by, seeded) VALUES
    ('(^|\s)\s', '\1\2', true),
    ('\s)\s($|)', '\1\2', true),
    ('(^|\s)-(\s|$)', '\1\2', true);

-- Добавляем функцию, в ней квалифицируем таблицу именем схемы, чтобы не зависеть от настройки пути
поиска. Имя схемы задается макросом.

CREATE FUNCTION typo(INOUT s text) AS $$
DECLARE
    r record;
BEGIN
    FOR r IN (
        SELECT pattern, replace_by
        FROM @extschema@.typo_rules
        ORDER BY id
    )
    LOOP
        s := regexp_replace(s, r.pattern, r.replace_by, 'g');
    END LOOP;
END;
$$ LANGUAGE plpgsql STABLE;

-- Для того чтобы утилита pg_dump корректно выгружала пользовательские правила, вызываем специальную
функцию не только для таблицы, но и для последовательности, которая используется для первичного
ключа.

SELECT pg_extension_config_dump('typo_rules', 'WHERE NOT seeded');
SELECT pg_extension_config_dump('typo_rules_id_seq', '');
```

Makefile и установка расширения в систему:

```
typo/Makefile
```

```
EXTENSION = typo
DATA = typo--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

```
student$ sudo make install -C typo
```

```
make: Entering directory '/home/student/typo'
/bin/mkdir -p '/usr/share/postgresql/16/extension'
```

```
/bin/mkdir -p '/usr/share/postgresql/16/extension'
/usr/bin/install -c -m 644 ./typo.control '/usr/share/postgresql/16/extension/'
/usr/bin/install -c -m 644 ./typo--1.0.sql '/usr/share/postgresql/16/extension/'
make: Leaving directory '/home/student/typo'
```

2. Проверка

Создаем базу данных и схему, и устанавливаем расширение:

```
=> CREATE DATABASE ext_extensions;
```

```
CREATE DATABASE
```

```
=> \c ext_extensions
```

You are now connected to database "ext_extensions" as user "student".

```
=> CREATE SCHEMA typo;
```

```
CREATE SCHEMA
```

```
=> CREATE EXTENSION typo SCHEMA typo;
```

```
CREATE EXTENSION
```

При установке макрос был автоматически заменен на имя выбранной схемы:

```
=> \sf typo.typo
```

```
CREATE OR REPLACE FUNCTION typo.typo(INOUT s text)
  RETURNS text
  LANGUAGE plpgsql
  STABLE
AS $function$
DECLARE
  r record;
BEGIN
  FOR r IN (
    SELECT pattern, replace_by
    FROM typo.typo_rules
    ORDER BY id
  )
  LOOP
    s := regexp_replace(s, r.pattern, r.replace_by, 'g');
  END LOOP;
END;
$function$
```

Проверим:

```
=> SELECT typo.typo(
  'Вдруг попугай заорал: "Овер-рсан! Овер-рсан!" - и все замерли.'
);
```

```

              typo
-----
Вдруг попугай заорал: «Овер-рсан! Овер-рсан!» — и все замерли.
(1 row)
```

Добавим правило:

```
=> INSERT INTO typo.typo_rules(pattern, replace_by)
  VALUES ('+', ' ');
```

```
INSERT 0 1
```

```
=> SELECT typo.typo(
  '- Будет, - сказал Дрозд. - Я уже букву "к" нарисовал.'
);
```

```

              typo
-----
- Будет, — сказал Дрозд. — Я уже букву «к» нарисовал.
(1 row)
```

```
=> \q
```

Выгружаем копию базы данных и восстанавливаемся из нее (при восстановлении база данных будет удалена и создана заново):

```
student$ pg_dump --clean --create ext_extensions > ext_extensions.dump
```

```
student$ psql -f ext_extensions.dump
```

```
SET
SET
SET
SET
SET
set_config
-----
```

```
(1 row)
```

```
SET
SET
SET
SET
DROP DATABASE
CREATE DATABASE
ALTER DATABASE
You are now connected to database "ext_extensions" as user "student".
SET
SET
SET
SET
set_config
-----
```

```
(1 row)
```

```
SET
SET
SET
SET
CREATE SCHEMA
ALTER SCHEMA
CREATE EXTENSION
COMMENT
COPY 1
setval
-----
```

```
4
```

```
(1 row)
```

Обратите внимание, что последней командой было установлено корректное значение последовательности. Если бы функция `pg_extension_config_dump` была вызвана только для таблицы, этого бы не произошло.

```
student$ psql ext_extensions
```

Проверяем:

```
=> INSERT INTO typo.typo_rules(pattern, replace_by)
VALUES ('\.\.\.', '...');
```

```
INSERT 0 1
```

```
=> SELECT typo.typo(
'Как это там... Соус пикан. Полстакана уксусу, две луковицы... и перчик.'
);
```

```
typo
```

```
-----
Как это там... Соус пикан. Полстакана уксусу, две луковицы... и перчик.
(1 row)
```