ТЕМА 14. ОСНОВЫ СЕРВЕРНОГО ПРОГРАММИРОВАНИЯ

Типы данных

Числовые типы:

Имя	Описание	Диапазон
`integer`	типичный выбор для целых чисел	-2147483648 .. +2147483647
`decimal, numeric`	вещественное число с указанной точностью	до 131072 цифр до десятичной точки и до 16383 — после
`real, double`	вещественное число с переменной точностью	точность в пределах 6, 15 десятичных цифр
`serial`	целое с автоувеличением	1 .. 2147483647

Операции:

`+` сложение,
`−` вычитание,
`*` умножение,
`/` деление (при целочисленном делении остаток отбрасывается),
`%` остаток от деления,
`^` возведение в степень (вычисляется слева направо),
`|/` квадратный корень,
`||/` кубический корень,
`!` факториал,
`!!` факториал (префиксная форма),
`@` модуль числа (абсолютное значение)

Символьные типы

Имя	Описание
`character varying(`*n*`), varchar(`*n*`)`	строка ограниченной переменной длины
`character(`*n*`), char(`*n*`)`	строка фиксированной длины, дополненная пробелами
`text`	строка неограниченной переменной длины

Строковая константа в SQL — это обычная последовательность символов, заключённая в апострофы ('). Ещё один способ записи строковых констант — «заключение строк в доллары»: $$Некоторый текст$$ или $SomeTag$Некоторый $SomeTag$

Операции:

`||` конкатенация
`length` число символов
`substring` извлечение подстроки
`upper, lower`
`replace`
`left, right`
`format`

Типы даты/времени

Имя	Размер	Описание	Точность
`date`	4 байта	дата (без времени суток)	1 день
`time`	8 байт	время суток (без даты)	1 микросекунда / 14 цифр

Операторы:

`current_date` текущая дата
`current_time` текущее время суток
`now()` текущая дата и время на момент начала транзакции

```
make_date(year int, month int, dayint)
make_time(hour int, min int, secdouble precision)
EXTRACT(field FROM source)
```

Логический тип данных

Имя	Размер	Описание
`boolean`	1 байт	состояние: истина или ложь

Состояние «true» может задаваться следующими значениями: TRUE,'t', 'true','y','yes', 'on', '1'
Для состояния «false» можно использовать следующие варианты: FALSE, 'f', 'false', 'n', 'no', 'off', '0
Операторы: AND, OR, NOT

Геометрические типы: `point, line, lseg, box, path, polygon, circle`

Типы JSON (JavaScript Object Notation, NoSQL): `json и jsonb.`

Массивы:	`<тип>[][]…`
Ввод значения:	`'{{1,2,3},{4,5,6},{7,8,9}}'`
Обращение к массиву:	`SELECT f[1] FROM T`
Прямоугольные срезы массива (подмассивы)	`SELECT schedule[1:2][1:1] FROM T`
Текущие размеры значения массива (в текстовом виде) :	`SELECT array_dims(schedule) FROM T`
Верхняя и нижняя граница для указанной размерности:	`SELECT array_upper(schedule, 1) FROM T`
Вставка элементов в конец:	`array_append(anyarray,anyelement)`
Вставка элементов в начало:	`array_prepend(anyelement,anyarray)`
Удаление элемента:	`array_remove(anyarray,anyelement)`
Замена элемента:	`array_replace(anyarray,anyelement, anyelement)`
Массив в строку:	`array_to_string(anyarray,text [, text])`
Строка в массив:	`string_to_array(text,text [, text])`
Массив в набор строк:	`unnest(anyarray)`
только в секции FROM	`unnest(anyarray, anyarray, …)`

Преобразование типов

CAST (*выражение* AS *тип*)
выражение::*тип*
имя_типа (*выражение*)

Создание типов

Команда CREATE TYPE имеет пять форм: *составной тип*, *перечисление*, *диапазон*, *базовый тип* или *тип-пустышку*.

1. Составной тип

```
CREATE TYPE имя AS
( [ имя_атрибута тип_данных [ COLLATE правило_сортировки ] [, ... ] ] )

CREATE TYPE compfoo AS (f1 int, f2 text);
```

2. Перечисление

```
CREATE TYPE имя AS ENUM
( [ 'метка' [, ... ] ] )

CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');
```

Серверное программирование

В PostgreSQL представлены функции четырёх видов:
• функции на языке запросов (функции, написанные на SQL)
• функции на процедурных языках (функции, написанные, например, на PL/pgSQL или PL/Tcl)
• внутренние функции
• функции на языке C

Функции любых видов могут принимать в качестве аргументов (параметров) базовые типы, составные типы или их сочетания. Кроме того, любые функции могут возвращать значения базового или составного типа. Также можно определить функции, возвращающие наборы базовых или составных значений.

Структура функций PL/pgSQL

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'тело функции'
LANGUAGE plpgsql;
```

PL/pgSQL это блочно-структурированный язык. Текст тела функции должен быть *блоком*. Структура блока:

```
[ <<метка>> ]
[ DECLARE объявления ]
BEGIN
операторы
END [ метка ];
```

Каждое объявление и каждый оператор в блоке должны завершаться символом ";". Блок, вложенный в другой блок, должен иметь точку с запятой после END, как показано выше. Однако финальный END, завершающий тело функции, не требует точки с запятой.

Метка требуется только тогда, когда нужно идентифицировать блок в операторе EXIT, или дополнить имена переменных, объявленных в этом блоке. Если метка указана после END, то она должна совпадать с меткой в начале блока.

Комментарии в PL/pgSQL коде работают так же, как и в обычном SQL. Двойное тире (--) начинает комментарий, который завершается в конце строки. Блочный комментарий начинается с /* и завершается */. Блочные комментарии могут быть вложенными.

Любой оператор в выполняемой секции блока может быть *вложенным блоком*. Вложенные блоки используются для логической группировки нескольких операторов или локализации области действия переменных для группы операторов. Во время выполнения вложенного блока переменные, объявленные в нём, скрывают переменные внешних блоков с такими же именами. Чтобы получить доступ к внешним переменным, нужно дополнить их имена меткой блока.

Объявления

Все переменные, используемые в блоке, должны быть определены в секции объявления. Общий синтаксис объявления переменной:

```
имя [ CONSTANT ] тип [ NOT NULL ] [ { DEFAULT | := | = } выражение ];
```

Предложение DEFAULT, если присутствует, задаёт начальное значение, которое присваивается переменной при входе в блок. Если отсутствует, то переменная инициализируется SQL-значением NULL. Указание CONSTANT предотвращает изменение значения переменной после инициализации, таким образом, значение остаётся постоянным в течение всего блока. Все переменные, объявленные как NOT NULL, должны иметь непустые значения по умолчанию.

Примеры:

```
quantity numeric(5);
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

Присваивания

```
переменная { := | = } выражение;
```

Целевая переменная может быть простой переменной (возможно, дополненной именем блока), полем в переменной строкового типа или записи; или элементом массива, который является простой переменной или полем. Если тип данных результата выражения не соответствует типу данных переменной, это значение будет преобразовано к нужному типу с использованием приведения присваивания.

Результат SQL-команды, возвращающей одну строку (возможно из нескольких столбцов), может быть присвоен переменной типа record, переменной строкового типа или списку скалярных переменных:

```
SELECT выражения_select INTO цель FROM ...;

SELECT * INTO myrec FROM emp WHERE empname = myname;
```

Выполнение динамически формируемых команд

```
EXECUTE строка-команды [ INTO цель ] [ USING выражение [, ... ] ];
```

где *строка-команды* это выражение, формирующее строку (типа text) с текстом команды, которую нужно выполнить. Необязательная *цель* — это переменная-запись, переменная-кортеж или разделённый запятыми список простых переменных и полей записи/кортежа, куда будут помещены результаты команды. В тексте команды можно использовать значения параметров, ссылки на параметры обозначаются как $1, $2 и т. д. Эти символы указывают на значения, находящиеся в предложении USING.

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
INTO c
USING checked_user, checked_date;
```

Символы параметров можно использовать только вместо значений данных и только в командах SELECT, INSERT, UPDATE и DELETE. Если в предыдущем запросе необходимо динамически задавать имя таблицы, можно сделать следующее:

```
EXECUTE format('SELECT count(*) FROM %I 'WHERE inserted_by = $1 AND inserted <=
$2', tabname)
INTO c
USING checked_user, checked_date;
```

Возврат значения из функции

```
RETURN выражение;
```

Прекращает выполнение функции и возвращает значение выражения в вызывающую программу. Для функции с выходными параметрами просто используйте RETURN без выражения (будут возвращены текущие значения выходных параметров). Для функции, возвращающей void, RETURN можно использовать в любом месте, но без выражения после RETURN.

```
RETURN NEXT выражение;
RETURN QUERY запрос;
RETURN QUERY EXECUTE строка-команды [USING выражение [, ...]];
```

Для функций на PL/pgSQL, возвращающих SETOF *некий_тип:* отдельные элементы возвращаемого значения формируются командами RETURN NEXT или RETURN QUERY, а финальная команда RETURN без аргументов завершает выполнение функции. RETURN QUERY добавляет результат выполнения запроса к результату функции. RETURN NEXT и RETURN QUERY можно свободно смешивать в теле функции, в этом случае их результаты будут объединены. RETURN NEXT и RETURN QUERY не выполняют возврат из функции. Успешное выполнение RETURN NEXT и RETURN QUERY формирует множество строк результата. Для выхода из функции используется RETURN, обязательно без аргументов (или можно просто дождаться окончания выполнения функции).
RETURN QUERY EXECUTE используется для динамического выполнения запроса.

Управляющие структуры

Условные операторы
- IF ... THEN ... END IF
- IF ... THEN ... ELSE ... END IF
- IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF

и две формы CASE:

```
CASE выражение-поиска
WHEN выражение [, выражение [...]] THEN
операторы
[WHEN выражение [, выражение [...]] THEN операторы ...]
[ELSE операторы]
END CASE;
```

```
CASE x
WHEN 1, 2 THEN
msg := 'один или два';
ELSE
msg := 'значение, отличное от один или два';
END CASE;
```

```
CASE
WHEN логическое-выражение THEN
операторы
[WHEN логическое-выражение THEN операторы ...]
[ELSE операторы]
END CASE;
```

```
CASE
WHEN x BETWEEN 0 AND 10 THEN
msg := 'значение в диапазоне между 0 и 10';
WHEN x BETWEEN 11 AND 20 THEN
msg := 'значение в диапазоне между 11 и 20';
END CASE;
```

Простые циклы

1.
```
LOOP
[<<метка>>]
LOOP
операторы
END LOOP [ метка ];
```

```
EXIT [ метка ] [WHEN логическое-выражение];
CONTINUE [ метка ] [WHEN логическое-выражение];
```

2.

```
[<<метка>>]
WHILE логическое-выражение LOOP
операторы
END LOOP [ метка ];
```

3.

```
 [<<метка>>]
FOR имя IN [REVERSE] выражение .. выражение [BY выражение] LOOP
операторы
END LOOP [ метка ];
```

```
FOR i IN 1..10 LOOP
-- внутри цикла переменная i будет иметь значения 1,2,3,4,5,6,7,8,9,10
END LOOP;

FOR i IN REVERSE 10..1 LOOP
-- внутри цикла переменная i будет иметь значения 10,9,8,7,6,5,4,3,2,1
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
-- внутри цикла переменная i будет иметь значения 10,8,6,4,2
END LOOP;
```

4. Цикл по результатам запроса

```
[ <<метка>> ]
FOR цель IN запрос LOOP
операторы
END LOOP [ метка ];
```

Переменная *цель* может быть строковой переменной, переменной типа record или разделённым запятыми списком скалярных переменных. Переменной *цель* последовательно присваиваются строки результата запроса, и для каждой строки выполняется тело цикла.

[<<*метка*>>]
FOR *цель* IN EXECUTE *выражение_проверки* [USING *выражение* [, ...]] LOOP
операторы
END LOOP [*метка*];

Текст запроса указывается в виде строкового выражения, что даёт выбор между скоростью предварительно разобранного запроса и гибкостью динамического запроса.

5.
Перебор элементов массива.

```
[ <<метка>> ]
FOREACH цель [ SLICE число ] IN ARRAY выражение LOOP
операторы
END LOOP [ метка ];
```

Без указания SLICE, или если SLICE равен 0, цикл выполняется по всем элементам массива, полученного из *выражения*. Для SLICE выполняется итерации по срезам массива. Значение SLICE должно быть целым числом, не превышающим размерности массива.

Обработка ошибок

```
[ <<метка>> ]
[ DECLARE
объявления ]
BEGIN
операторы
EXCEPTION
WHEN условие [ OR условие ... ] THEN
операторы_обработчика
[ WHEN условие [ OR условие ... ] THEN
операторы_обработчика
... ]
END;
```

Если ошибок не было, то выполняются все *операторы* блока и управление переходит к следующему оператору после END. Но если при выполнении *оператора* происходит ошибка, то дальнейшая обработка прекращается и управление переходит к списку исключений в секции EXCEPTION. В этом списке ищется первое исключение, условие которого соответствует ошибке. Если исключение не найдено, то ошибка передаётся наружу, как будто секции EXCEPTION не было. Ошибку можно перехватить в секции EXCEPTION внешнего блока. Если ошибка так и не была перехвачена, то обработка функции прекращается.

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
      UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
      x := x + 1;
      y := x / 0;
      EXCEPTION
            WHEN division_by_zero THEN
                  RAISE NOTICE 'перехватили ошибку division_by_zero';
      RETURN x;
END;
```

При присвоении значения переменной y произойдёт ошибка division_by_zero. Она будет перехвачена в секции EXCEPTION. Оператор RETURN вернёт значение x, увеличенное на единицу, но изменения сделанные командой UPDATE будут отменены. Изменения, выполненные командой INSERT, которая предшествует блоку, не будут отменены. В результате, база данных будет содержать Tom Jones, а не Joe Jones.

Вывод сообщений и ошибок

```
RAISE [ уровень ] 'формат' [, выражение [, ... ]]

RAISE [ уровень ] имя_условия [ USING параметр = выражение [, ... ] ];
RAISE [ уровень ] SQLSTATE 'sqlstate' [ USING параметр = выражение [, ... ] ];
```

уровень задаёт уровень важности ошибки. Возможные значения: DEBUG, LOG, INFO, NOTICE, WARNING и EXCEPTION. По умолчанию используется EXCEPTION. EXCEPTION вызывает ошибку (что обычно прерывает текущую транзакцию), остальные значения *уровня* только генерируют сообщения с различными уровнями приоритета.

После указания *уровня*, если оно есть, можно задать *формат* (это должна быть простая строковая константа, не выражение). Строка формата определяет вид текста об ошибке, который будет выдан. За строкой формата могут следовать необязательные выражения аргументов, которые будут вставлены в сообщение. Внутри строки формата знак % заменяется строковым представлением значения очередного аргумента. Чтобы выдать символ % буквально, продублируйте его (как %%). Число аргументов должно совпадать с числом местозаполнителей % в строке формата, иначе при компиляции функции возникнет ошибка.

В следующем примере символ % будет заменён на значение v_job_id:

```
RAISE NOTICE 'Вызов функции cs_create_job(%)', v_job_id;
```