

# **Базы данных**

## **Лекция 17 – Интерфейс Postgres**

**Преподаватель: Поденок Леонид Петрович, 505а-5**

**+375 17 293 8039 (505а-5)**

**+375 17 320 7402 (ОИПИ НАНБ)**

**prep@lsi.bas-net.by**

**ftp://student:2ok\*uK2@Rwox@lsi.bas-net.by**

**Кафедра ЭВМ, 2023**

2023.12.15

## Оглавление

Функции исполнения команд.....	3
Основные функции.....	3
PQexec() – передаёт команду серверу и ожидает результата.....	3
PQexecParams() -- отправляет команду серверу и ожидает результата.....	5
PQprepare – запрос на создание подготовленного оператора.....	8
PQexecPrepared – запрос на исполнение подготовленного оператора.....	10
PQdescribePrepared – получение информации о подготовленном операторе.....	11
PQresultStatus – статус результата выполнения команды.....	12
PQresStatus – преобразовать код статуса в строку.....	13
PQresultErrorMessage – сообщение об ошибке.....	13
PQresultVerboseErrorMessage – более полное сообщения об ошибке.....	14
PQresultErrorField – поле из отчёта об ошибке.....	15
PQclear – освободить память, связанную с PGresult.....	18
Извлечение информации, связанной с результатом запроса.....	19
PQntuples – число строк (кортежей) в полученной выборке.....	19
PQnfields – число столбцов (полей) в каждой строке полученной выборки.....	19
PQfname – имя столбца с номером.....	20
PQfnumber – номер столбца по имени.....	21
PQftable – OID таблицы для данного столбца.....	22
PQftablecol – номер столбца в таблицы для столбца в выборке.....	23
PQfformat – код формата столбца.....	23
PQftype – тип данных по номеру столбца.....	24
PQfmod – модификатор типа для столбца.....	24
PQfsize – размер в байтах для столбца.....	25
PQgetvalue – значение поля из строки в PGresult.....	26
PQgetisnull – проверка поля на null.....	27
PQgetlength – фактическая длина значения поля в байтах.....	27
PQnparams – число параметров подготовленного оператора.....	28
PQparamtype – тип данных для параметра оператора.....	28
PQprint – вывод строк и имен столбцов в поток вывода.....	29
Получение другой информации о результате.....	30
PQcmdStatus – статус для исполненной SQL-команды.....	30
PQoidValue – OID вставленной строки.....	30
PQcmdTuples – число строк затронутых SQL-командой.....	31
Экранирование строковых значений для включения в SQL-команды.....	32
PQescapeLiteral – экранировать строковое значение.....	32
PQescapeStringConn – экранировать строковые литералы.....	34
PQescapeIdentifier – экранировать строку – идентификатор SQL.....	36
PQescapeByteaConn – экранировать двоичные данные.....	37
PQunescapeBytea – строковое представление → в двоичные данные.....	39

# Функции исполнения команд

## Основные функции

**PQexec()** – передаёт команду серверу и ожидает результата.

```
#include <libpq-fe.h> // -lpq -- компоновка с libpq.so

PGresult *PQexec(PGconn      *conn,      // соединение
                 const char *command); // команда
```

Возвращает указатель на объект PGresult или, возможно, пустой указатель NULL в случае нехватки памяти или возникновения серьёзной ошибки, например такой, как невозможность отправки команды серверу.

Для проверки возврата на наличие ошибок БД следует вызывать PQresultStatus( ), которая в случае нулевого указателя возвратит PGRES\_FATAL\_ERROR.

PGRES_TUPLES_OK	-- результат в PGresult
PGRES_EMPTY_QUERY	-- пустой запрос
PGRES_COMMAND_OK	-- команда подготовлена к исполнению
PGRES_NONFATAL_ERROR	-- уведомление или предупреждение (warning)
PGRES_BAD_RESPONSE	-- неожиданный ответ от движка
PGRES_COPY_OUT	-- идет копирование
PGRES_COPY_IN	-- идет копирование

Для получения дополнительной информации об ошибках следует использовать функцию PQerrorMessage( ).

Строка команды может включать в себя более одной SQL-команды.

**Команды разделяются точкой с запятой.**

Несколько запросов, отправленных с помощью одного вызова PQexec( ), обрабатываются в рамках одной транзакции.

Если команды BEGIN/COMMIT явно включены в строку запроса, он будет разделен на несколько транзакций.

**!!! ACHTUNG !!!**

PGresult описывает только результат последней из выполненных команд из строки запроса.

Если одна из команд завершается сбоем, то обработка строки запроса на этом останавливается, и возвращённая структура PGresult содержит состояние ошибки.

## PQexecParams() -- отправляет команду серверу и ожидает результата.

Имеет возможность передать параметры отдельно от текста SQL-команды.

```
PGresult *PQexecParams(PGconn      *conn,
                        const char *command,    // параметризованная к-да
                        int          nParams,    // кол-во параметров
                        const Oid   *paramTypes, // oid-типы параметров
                        const char * const *paramValues, // значения
                        const int   *paramLengths, // размеры данных
                        const int   *paramFormats, // текстовый/двоичный
                        int          resultFormat); // текстовый/двоичный
```

PQexecParams( ) похожа на PQexec( ), но предлагает дополнительную функциональность:

- отдельно от самой строки-команды могут быть указаны значения параметров;
- результаты запроса могут быть получены как в текстовом, так и в двоичном формате.

**conn** – подключение, через которое пересылается команда.

**command** – строка SQL-команды, которую следует выполнить. Если используются параметры, то в строке команды на них ссылаются, как \$1, \$2 ...

**nParams** – число предоставляемых параметров. Оно равно размеру массивов paramTypes[], paramValues[], paramLengths[] и paramFormats[].

Если nParams равно нулю, указатели на массивы могут быть равны NULL.

**paramTypes[]** – предписывает, используя OID, типы данных, которые должны быть назначены параметрам.

**paramValues[ ]** – содержит фактические значения параметров.

Нулевой указатель в этом массиве означает, что соответствующий параметр равен null; в противном случае указатель указывает на текстовую строку, завершающуюся нулевым символом (для текстового формата), или на двоичные данные в формате, которого ожидает сервер (для двоичного формата).

**paramLengths[ ]** – содержит фактические размеры данных для параметров, представленных в двоичном формате.

Он игнорируется для параметров, имеющих значение null, и для параметров, представленных в текстовом формате. Если нет двоичных параметров, указатель на массив может быть нулевым.

**paramFormats[ ]** – указывает, что параметр текстовый (0) или двоичный (1).

Если указатель на массив является нулевым, тогда все параметры считаются текстовыми строками.

Значения, переданные в двоичном формате, требуют знания внутреннего представления, которого ожидает сервер. Например, целые числа должны передаваться с использованием сетевого порядка байтов (hton{l|s}(), ntoh{l|s}()).

**Передача значений типа numeric требует знания формата, в котором их хранит сервер.**

Информацию об этом можно получить из функций numeric\_send() и numeric\_recv(), располагающихся в src/backend/utils/adt/numeric.c

Данные функции преобразуют numeric в двоичный формат и обратно.

**resultFormat** – формат возврата.

0 – результаты в текстовом формате;

1 – результаты в двоичном формате.

Основное преимущество PQexecParams( ) от PQexec( ) – значения параметров могут быть отделены от строки с командой. Это позволяет избежать использования кавычек и экранирующих символов, что часто приводит к ошибкам.

В отличие от PQexec( ), PQexecParams( ) позволяет включать в строку запроса только одну SQL-команду.

В ней могут содержаться точки с запятой, однако может присутствовать не более одной непустой команды.

### !!! ВАЖНО !!!

Можно избежать указания типов параметров с помощью OID, для чего, чтобы показать, какой тип данных отправляется, в строке SQL-команды следует добавить явное приведение типа для этого параметра. Например:

```
SELECT * FROM mytable WHERE x = $1::bigint;
```

Это заставит считать параметр \$1 имеющим тип bigint, хотя по умолчанию ему был бы назначен тот же самый тип, что и x.

Если значения параметров отправляются в двоичном формате, строго рекомендуется явное указание о типе параметра либо с помощью описанного метода, либо путём задания числового OID.

## PQprepare – запрос на создание подготовленного оператора<sup>1</sup>

Отправляет запрос, чтобы создать подготовленный оператор с конкретными параметрами, и ожидает завершения.

```
PGresult *PQprepare(PGconn      *conn,
                    const char *stmtName,    // подготовленный оператор
                    const char *query,       // команда SQL
                    int         nParams,     // кол-во параметров
                    const Oid   *paramTypes); // oid-типы параметров
```

PQprepare( ) создаёт подготовленный оператор, который может быть впоследствии исполнен с помощью PQexecPrepared( ).

Благодаря этому, команды, которые будут выполняться многократно, серверу не потребуется разбирать и планировать каждый раз.

**query** – команда SQL.

**stmtName** – здесь создаётся подготовленный оператор с именем stmtName из строки query, которая должна содержать единственную SQL-команду.

Если stmtName пустая строка "", то будет создан *неименованный* оператор.

Любой уже существующий неименованный оператор будет автоматически заменён.

Если имя оператора уже определено в текущем сеансе работы, будет ошибка.

Если используются параметры, то в запросе к ним обращаются таким образом: \$1, \$2.

---

1) PQprepare() поддерживается только с подключениями по протоколу 3.0 и новее.



**nParams** – число параметров, типы данных для которых указаны в массиве `paramTypes[]`.

Если значение `nParams` равно нулю, указатель на массив может быть равен `NULL`.

**paramTypes[]** – OID-типы данных, которые будут назначены параметрам.

Если `paramTypes` равен `NULL` или какой-либо элемент в этом массиве равен нулю, то сервер назначает тип данных соответствующему параметру точно таким же способом, как он сделал бы для литеральной строки, не имеющей типа.

Также в запросе можно использовать параметры с номерами, большими, чем `nParams`, в этом случае сервер сможет подобрать типы данных для них самостоятельно.

Результатом вызова является объект `PGresult`, содержимое которого показывает успех или сбой на стороне сервера.

Нулевой указатель означает нехватку памяти или невозможность вообще отправить команду. Для получения дополнительной информации о таких ошибках следует использовать `PQerrorMessage()`.

Подготовленные операторы для использования с `PQexecPrepared()` можно также создать путём исполнения SQL-команд `PREPARE`.

## PQexecPrepared – запрос на исполнение подготовленного оператора<sup>2</sup>

Отправляет запрос на исполнение подготовленного оператора с данными параметрами и ожидает результата.

```
PGresult *PQexecPrepared(PGconn      *conn,
                          const char *stmtName, // имя подготовленного
                          int         nParams,  //
                          const char *const *paramValues,
                          const int  *paramLengths,
                          const int  *paramFormats,
                          int         resultFormat);
```

PQexecPrepared( ) подобна PQexecParams( ), но команда, подлежащая исполнению, указывается путём передачи имени предварительно подготовленного оператора вместо передачи строки запроса.

Эта возможность позволяет командам, которые вызываются многократно, подвергаться разбору и планированию только один раз, а не при каждом их исполнении.

Оператор должен быть предварительно подготовлен в рамках текущего сеанса работы.

Параметры идентичны PQexecParams( ), за исключением того, что вместо строки SQL-запроса передаётся имя подготовленного оператора, а параметр paramTypes[ ] отсутствует, поскольку типы данных для параметров были определены при создании подготовленного оператора.

---

2) PQexecPrepared() поддерживается только в соединениях по протоколу версии 3.0 или более поздних версий.

## PQdescribePrepared – получение информации о подготовленном операторе<sup>3</sup>

Передаёт запрос на получение информации об указанном подготовленном операторе и ожидает завершения.

```
PGresult *PQdescribePrepared(PGconn      *conn,  
                             const char *stmtName);
```

PQdescribePrepared( ) позволяет приложению получить информацию о предварительно подготовленном операторе.

Для ссылки на неименованный оператор значение stmtName может быть пустой строкой "" или NULL, в противном случае оно должно быть именем существующего подготовленного оператора.

В случае успеха возвращается PGresult со статусом PGRES\_COMMAND\_OK.

Функции PQnparams( ) и PQparamtype( ) позволяют извлечь из PGresult информацию о параметрах подготовленного оператора, а функции PQnfields( ), PQfname( ), PQftype( ) и т. п. предоставляют информацию о результирующих столбцах (если они есть) данного оператора.

---

3) PQdescribePrepared() поддерживается только в соединениях по протоколу версии 3.0 или более поздних версий.

## **PQresultStatus – статус результата выполнения команды**

Возвращает статус результата выполнения команды.

```
ExecStatusType PQresultStatus(const PGresult *res);
```

PQresultStatus может возвращать одно из следующих значений:

**PGRES\_EMPTY\_QUERY** – строка, отправленная серверу, была пустой.

**PGRES\_COMMAND\_OK** – успешное завершение команды, не возвращающей никаких данных.

Ответ PGRES\_COMMAND\_OK предназначен для команд, которые никогда не возвращают строки (INSERT или UPDATE без использования предложения RETURNING и др.).

**PGRES\_TUPLES\_OK** – успешное завершение команды, возвращающей данные (такой, как SELECT или SHOW), и SELECT не возвративший никаких данных тоже.

**PGRES\_COPY\_OUT** – начат перенос данных Copy Out (с сервера).

**PGRES\_COPY\_IN** – начат перенос данных Copy In (на сервер).

**PGRES\_BAD\_RESPONSE** – ответ сервера не был распознан.

**PGRES\_NONFATAL\_ERROR** – произошла не фатальная ошибка (уведомление или предупреждение).

**PGRES\_FATAL\_ERROR** – произошла фатальная ошибка.

**PGRES\_COPY\_BOTH** – начат перенос данных Copy In/Out (на сервер и с сервера).

**PGRES\_SINGLE\_TUPLE** – структура PGresult содержит только одну результирующую строку, возвращённую текущей командой.

Этот статус имеет место только тогда, когда для данного запроса был выбран режим построчного вывода.

## **PQresStatus – преобразовать код статуса в строку**

Преобразует значение перечислимого типа, возвращённое функцией PQresultStatus, в строковую константу, описывающую код статуса.

```
char *PQresStatus(ExecStatusType status);
```

## **PQresultErrorMessage – сообщение об ошибке**

Возвращает сообщение об ошибке, связанное с командой, или пустую строку, если ошибки не произошло.

```
char *PQresultErrorMessage(const PGresult *res);
```

Если произошла ошибка, то возвращённая строка будет включать завершающий символ новой строки.

Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель – она будет освобождена, когда соответствующий указатель PGresult будет передан функции PQclear( ).

## **PQresultVerboseErrorMessage – более полное сообщения об ошибке**

Возвращает более полную версию сообщения об ошибке из объекта PGresult.

```
char *PQresultVerboseErrorMessage(const PGresult      *res,  
                                  PGVerbosity         verbosity,  
                                  PGContextVisibility show_context);
```

В некоторых ситуациях клиент может захотеть получить более подробную версию ранее выданного сообщения об ошибке.

PQresultVerboseErrorMessage( ) формирует сообщение, которое было бы выдано функцией PQresultErrorMessage( ), если бы заданный уровень детализации был текущим для соединения в момент заполнения PGresult.

Если в PGresult ошибки нет, выдаётся сообщение «PGresult is not an error result» (PGresult — не результат с ошибкой).

Возвращаемое этой функцией сообщение завершается переводом строки.

В отличие от многих других функций, извлекающих данные из PGresult, результат этой функции — новая размещённая в памяти строка.

Когда эта строка будет не нужна, вызывающий код должен освободить её место, вызвав PQfreemem( ).

При нехватке памяти может быть возвращен NULL.

## PQresultErrorField – поле из отчёта об ошибке

Возвращает индивидуальное поле из отчёта об ошибке.

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

**fieldcode** – идентификатор поля ошибки.

Если PGresult не содержит ошибки или предупреждения или не включает указанное поле, то возвращается NULL.

Значения полей обычно не включают завершающий символ новой строки.

Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель – она будет освобождена, когда соответствующий указатель PGresult будет передан функции PQclear( ).

Доступны следующие коды полей:

**PG\_DIAG\_SEVERITY** – серьёзность; поле может содержать ERROR, FATAL или PANIC (в сообщении об ошибке) либо WARNING, NOTICE, DEBUG, INFO или LOG (в сообщении-уведомлении) либо локализованный перевод одного из этих значений. Присутствует всегда.

**PG\_DIAG\_SEVERITY\_NONLOCALIZED** – серьёзность; поле может содержать ERROR, FATAL или PANIC (в сообщении об ошибке) либо WARNING, NOTICE, DEBUG, INFO или LOG (в сообщении-уведомлении). Это поле подобно PG\_DIAG\_SEVERITY, но его содержимое никогда не переводится. Присутствует только в отчётах, выдаваемых Postgres Pro версии 9.6 и новее.

**PG\_DIAG\_SQLSTATE** – код ошибки в соответствии с соглашением о кодах SQLSTATE.

Код SQLSTATE идентифицирует тип случившейся ошибки. Он может использоваться клиентскими приложениями, чтобы выполнять конкретные операции (обработка ошибок) в ответ на конкретную ошибку базы данных. Это поле всегда присутствует.

**PG\_DIAG\_MESSAGE\_PRIMARY** – главное сообщение об ошибке, предназначенное для прочтения пользователем.

Как правило составляет всего одну строку. Это поле всегда присутствует.

**PG\_DIAG\_MESSAGE\_DETAIL** – необязательное дополнительное сообщение об ошибке, передающее более детальную информацию о проблеме.

Может занимать несколько строк.

**PG\_DIAG\_MESSAGE\_HINT** – подсказка: необязательное предположение о том, что можно сделать в данной проблемной ситуации.

Оно отличается от детальной информации в том смысле, что оно предлагает совет (возможно, и неподходящий), а не просто факты. Может занимать несколько строк.

**PG\_DIAG\_STATEMENT\_POSITION** – строка, содержащая десятичное целое число, указывающее позицию расположения ошибки в качестве индекса в оригинальной строке оператора. Первый символ имеет позицию 1, при этом позиции измеряются в символах а не в байтах.

**PG\_DIAG\_INTERNAL\_POSITION** – это поле определяется точно так же, как и поле PG\_DIAG\_STATEMENT\_POSITION, но оно используется, когда позиция местонахождения ошибки относится к команде, сгенерированной внутренними модулями, а не к команде, представленной клиентом. Когда появляется это поле, то всегда появляется и поле PG\_DIAG\_INTERNAL\_QUERY.



**PG\_DIAG\_INTERNAL\_QUERY** – текст команды, сгенерированной внутренними модулями, завершившейся сбоем (например, SQL-запрос, выданный функцией на PL/pgSQL).

**PG\_DIAG\_CONTEXT** – характеристика контекста, в котором произошла ошибка.

Включает вывод стека вызовов активных функций процедурного языка и запросов, сгенерированных внутренними модулями.

**PG\_DIAG\_SCHEMA\_NAME** – если ошибка была связана с конкретным объектом базы данных, то в это поле будет записано имя схемы, содержащей данный объект.

**PG\_DIAG\_TABLE\_NAME** – если ошибка была связана с конкретной таблицей, то в это поле будет записано имя таблицы.

**PG\_DIAG\_COLUMN\_NAME** – если ошибка была связана с конкретным столбцом таблицы, то в это поле будет записано имя столбца. Чтобы идентифицировать таблицу, следует обратиться к полям, содержащим имена схемы и таблицы.

**PG\_DIAG\_DATATYPE\_NAME** – если ошибка была связана с конкретным типом данных, то в это поле будет записано имя типа данных.

**PG\_DIAG\_CONSTRAINT\_NAME** – если ошибка была связана с конкретным ограничением, то в это поле будет записано имя ограничения.

**PG\_DIAG\_SOURCE\_FILE** – имя файла, содержащего позицию в исходном коде, для которой было выдано сообщение об ошибке.

**PG\_DIAG\_SOURCE\_LINE** – номер строки той позиции в исходном коде, для которой было выдано сообщение об ошибке.

**PG\_DIAG\_SOURCE\_FUNCTION** – имя функции в исходном коде, сообщающей об ошибке.

## **PQclear – освободить память, связанную с PGresult**

Освобождает область памяти, связанную с PGresult. Результат выполнения каждой команды должен быть освобождён с помощью PQclear, когда он больше не нужен.

```
void PQclear(PGresult *res);
```

Вы можете держать объект PGresult под рукой до тех пор, пока он вам нужен; он не исчезает, ни когда вы выдаёте новую команду, ни даже если вы закрываете соединение. Чтобы от него избавиться, вы должны вызвать PQclear( ). Если этого не делать, то в результате будут иметь место утечки памяти в вашем приложении.

## Извлечение информации, связанной с результатом запроса

Эти функции служат для извлечения информации из объекта `PGresult`, который представляет результат успешного запроса (имеет статус `PGRES_TUPLES_OK` или `PGRES_SINGLE_TUPLE`).

### **PQntuples** – число строк (кортежей) в полученной выборке

Возвращает число строк (кортежей) в полученной выборке.

Объекты `PGresult` не могут содержать более чем `INT_MAX` строк, так что для результата достаточно типа `int`.

```
int PQntuples(const PGresult *res);
```

### **PQnfields** – число столбцов (полей) в каждой строке полученной выборки

Возвращает число столбцов (полей) в каждой строке полученной выборки.

```
int PQnfields(const PGresult *res);
```

## **PQfname – имя столбца с номером**

Возвращает имя столбца, соответствующего данному номеру столбца.

Номера столбцов начинаются с 0.

Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель – она будет освобождена, когда соответствующий указатель на PGresult будет передан функции PQclear( ).

```
char *PQfname(const PGresult *res,  
              int             column_number);
```

Если номер столбца выходит за пределы допустимого диапазона, то возвращается NULL.

## PQfname – номер столбца по имени

Возвращает номер столбца, соответствующий данному имени столбца.

```
int PQfname(const PGresult *res,  
            const char      *column_name);
```

Если данное имя не совпадает с именем ни одного из столбцов, то возвращается -1.

Данное имя интерпретируется, как идентификатор в SQL-команде.

Это означает, что оно переводится в нижний регистр, если только оно не заключено в двойные кавычки.

Например, для выборки, сгенерированной с помощью такой SQL-команды:

```
SELECT 1 AS F00, 2 AS "BAR";
```

будет получено:

PQfname(res, 0)	foo
PQfname(res, 1)	BAR
PQfname(res, "F00")	0
PQfname(res, "foo")	0
PQfname(res, "BAR")	-1
PQfname(res, "\"BAR\"")	1

## PQftable – OID таблицы для данного столбца

Возвращает OID таблицы, из которой был получен данный столбец. Номера столбцов начинаются с 0.

```
Oid PQftable(const PGresult *res,  
             int             column_number);
```

В следующих случаях возвращается Invalid0id:

- если номер столбца выходит за пределы допустимого диапазона;
- если указанный столбец не является простой ссылкой на столбец таблицы;
- когда используется протокол версии более ранней, чем 3.0.

Чтобы точно определить, к какой таблице было произведено обращение, можно сделать запрос к системной таблице pg\_class.

Тип данных Oid и константа Invalid0id будут определены только в том случае, если будет включен заголовок для libpq.

Они будут принадлежать к одному из целочисленных типов.

## **PQftablecol – номер столбца в таблицы для столбца в выборке**

Возвращает номер столбца (в пределах его таблицы) для указанного столбца в полученной выборке. Номера столбцов в полученной выборке начинаются с 0, но столбцы в таблице имеют ненулевые номера.

```
int PQftablecol(const PGresult *res,  
                int          column_number);
```

В следующих случаях возвращается ноль:

- если номер столбца выходит за пределы допустимого диапазона;
- если указанный столбец не является простой ссылкой на столбец таблицы;
- когда используется протокол версии более ранней, чем 3.0.

## **PQfformat – код формата столбца**

Возвращает код формата, показывающий формат данного столбца. Номера столбцов начинаются с 0.

```
int PQfformat(const PGresult *res,  
              int          column_number);
```

Значение кода формата, равное нулю, указывает на текстовое представление данных, в то время, как значение, равное единице, означает двоичное представление.

## **PQftype – тип данных по номеру столбца**

Возвращает тип данных, соответствующий данному номеру столбца.

Возвращаемое целое значение является внутренним номером OID для этого типа.

Номера столбцов начинаются с 0.

```
Oid PQftype(const PGresult *res,  
            int             column_number);
```

Чтобы получить имена и свойства различных типов данных, можно сделать запрос к системной таблице pg\_type. Значения OID для встроенных типов данных определены в файле include/server/catalog/pg\_type.h в каталоге установленного сервера.

## **PQfmod – модификатор типа для столбца**

Возвращает модификатор типа для столбца, соответствующего данному номеру. Номера столбцов начинаются с 0.

```
int PQfmod(const PGresult *res,  
           int             column_number);
```

Интерпретация значений модификатора зависит от типа; они обычно показывают точность или предельные размеры. Значение -1 используется, чтобы показать «нет доступной информации». Большинство типов данных не используют модификаторов, в таком случае значение всегда будет -1.



## **PQfsize – размер в байтах для столбца**

Возвращает размер в байтах для столбца, соответствующего данному номеру. Номера столбцов начинаются с 0.

```
int PQfsize(const PGresult *res,  
            int           column_number);
```

PQfsize( ) возвращает размер пространства, выделенного для этого столбца в строке базы данных.

Это размер внутреннего представления этого типа данных на сервере.

Отрицательное значение говорит о том, что тип данных имеет переменную длину.

## PQgetvalue – значение поля из строки в PGresult

Возвращает значение одного поля из одной строки, содержащейся в PGresult.

Номера строк и столбцов начинаются с 0.

Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель – она будет освобождена, когда соответствующий указатель на PGresult будет передан функции PQclear( ).

```
char *PQgetvalue(const PGresult *res,  
                 int             row_number,  
                 int             column_number);
```

Для данных в текстовом формате значение, возвращаемое функцией PQgetvalue( ), является значением поля, представленным в виде символьной строки с завершающим нулевым символом.

Для данных в двоичном формате используется двоичное представление значения.

Оно определяется функциями typsend( ) и typreceive( ) для конкретного типа данных.

Когда значение поля отсутствует (null), возвращается пустая строка.

Указатель, возвращаемый функцией PQgetvalue( ), указывает на область хранения, которая является частью структуры PGresult.

Данные, на которые указывает этот указатель, не следует модифицировать. Вместо этого, если предполагается их использовать за пределами времени жизни самой структуры PGresult, нужно их явно скопировать данные в другую область хранения.

## **PQgetisnull – проверка поля на null**

Проверяет поле на предмет отсутствия значения (null).

Номера строк и столбцов начинаются с 0.

```
int PQgetisnull(const PGresult *res,  
                int          row_number,  
                int          column_number);
```

Эта функция возвращает 1, если значение в поле отсутствует (null), и 0, если поле содержит непустое (non-null) значение.

PQgetvalue( ) же, если значение в поле отсутствует, возвратит пустую строку, а не нулевой указатель.

## **PQgetlength – фактическая длина значения поля в байтах**

Возвращает фактическую длину значения поля в байтах.

Номера строк и столбцов начинаются с 0.

```
int PQgetlength(const PGresult *res,  
                int row_number,  
                int column_number);
```

Это фактическая длина данных для конкретного значения данных, то есть размер объекта, на который указывает PQgetvalue( ).

Для текстового формата данных это то же самое, что strlen( ).

## **PQnparams – число параметров подготовленного оператора**

Возвращает число параметров подготовленного оператора.

```
int PQnparams(const PGresult *res);
```

Эта функция полезна только при исследовании результата работы функции PQdescribePrepared( ).

Для других типов запросов она возвратит ноль.

## **PQparamtype – тип данных для параметра оператора**

Возвращает тип данных для указанного параметра оператора.

Номера параметров начинаются с 0.

```
Oid PQparamtype(const PGresult *res, int param_number);
```

Эта функция полезна только при исследовании результата работы функции PQdescribePrepared( ).

Для других типов запросов она возвратит ноль.

## PQprint – вывод строк и имен столбцов в поток вывода

Выводит все строки и, по выбору, имена столбцов в указанный поток вывода.

```
void PQprint(FILE          *fout, // поток вывода
              const PGresult *res, //
              const PQprintOpt *po); // опции печати

typedef struct {
    pqbool header;      // печатать заголовки полей и счётчик строк
    pqbool align;       // выравнивать поля
    pqbool standard;    // старый формат
    pqbool html3;       // выводить HTML-таблицы
    pqbool expanded;    // расширять таблицы
    pqbool pager;       // использовать программу для постраничного
                        // просмотра, если нужно
    char *fieldSep;     // разделитель полей
    char *tableOpt;     // атрибуты для HTML-таблицы
    char *caption;      // заголовок HTML-таблицы
    char **fieldName;   // массив заменителей для имён полей,
                        // завершающийся нулевым символом
} PQprintOpt;
```

Эту функцию прежде использовала утилита `psql` для вывода результатов запроса, но больше она её не использует.

Предполагается, что все данные представлены в текстовом формате.

## Получение другой информации о результате

Эти функции используются для получения остальной информации из объектов PGresult.

### **PQcmdStatus – статус для исполненной SQL-команды**

Возвращает дескриптор статуса для SQL-команды, которая сгенерировала PGresult.

```
char *PQcmdStatus(PGresult *res);
```

Как правило, это просто имя команды, но могут быть включены и дополнительные сведения, такие, как число обработанных строк.

Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель – она будет освобождена, когда соответствующий указатель на PGresult будет передан функции PQclear( ).

### **PQoidValue – OID вставленной строки**

Возвращает OID вставленной строки, если SQL-команда была командой INSERT, которая вставила ровно одну строку в таблицу, имеющую идентификаторы OID, или командой EXECUTE, которая выполнила подготовленный запрос, содержащий соответствующий оператор INSERT. В противном случае эта функция возвращает InvalidOid. Эта функция также возвратит InvalidOid, если таблица, затронутая командой INSERT, не содержит идентификаторов OID.

```
Oid PQoidValue(const PGresult *res);
```

## **PQcmdTuples – число строк затронутых SQL-командой**

Возвращает число строк, которые затронула SQL-команда.

```
char *PQcmdTuples(PGresult *res);
```

Эта функция возвращает строковое значение, содержащее число строк, которые затронул SQL-оператор, сгенерировавший данный PGresult.

Эту функцию можно использовать только сразу после выполнения команд SELECT, CREATE TABLE AS, INSERT, UPDATE, DELETE, MOVE, FETCH или COPY, а также после оператора EXECUTE, выполнившего подготовленный запрос, содержащий команды INSERT, UPDATE или DELETE.

Если команда, которая сгенерировала PGresult, была какой-то иной, то функция PQcmdTuples( ) возвращает пустую строку.

Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель – она будет освобождена, когда соответствующий указатель на PGresult будет передан функции PQclear( ).

# Экранирование строковых значений для включения в SQL-команды

## PQescapeLiteral – экранировать строковое значение

```
char *PQescapeLiteral(PGconn      *conn,  
                      const char *str,      // исх. версия строки  
                      size_t      length); // ее длина
```

PQescapeLiteral( ) экранирует строковое значение для использования внутри SQL-команды. Она полезна при вставке в SQL-команды значений данных в виде литеральных констант.

Определённые символы (такие, как кавычки и символы обратной косой черты) должны экранироваться, чтобы предотвратить их специальную интерпретацию синтаксическим анализатором языка SQL. PQescapeLiteral( ) выполняет эту операцию.

**str** – неэкранированная версия строки, размещённая в области памяти, распределённой с помощью функции malloc( ).

Эту память нужно освободить с помощью функции PQfreemem( ), когда возвращённое значение больше не требуется. Завершающий нулевой байт не нужен и не должен учитываться в параметре length.

**length** – длина строки.

Если до того, как были обработаны length байт, был найден завершающий нулевой байт, то PQescapeLiteral( ) останавливает работу на нулевом байте. Таким образом, поведение функции напоминает strncpy( ).



В возвращённой строке все специальные символы будут заменены таким образом, что синтаксический анализатор строковых литералов Postgres сможет обработать их должным образом.

Также будет добавлен завершающий нулевой байт.

Одинарные кавычки, которые должны окружать строковые литералы в Postgres, включаются в результирующую строку.

В случае ошибки `PQescapeLiteral( )` возвращает `NULL`, и в объект `conn` помещается соответствующее сообщение.

### **!!! ВАЖНО !!!**

Особенно важно выполнять надлежащее экранирование при обработке строк, полученных из ненадёжных источников.

В противном случае безопасность подвергается риску из-за уязвимости в отношении атак с использованием «SQL-инъекций», с помощью которых в базу данных направляются нежелательные SQL-команды.

Экранировать значения данных, передаваемых в виде отдельных параметров в функцию `PQexecParams( )` или родственные ей функции, нет необходимости. Мало того, это будет некорректно.

## PQescapeStringConn – экранировать строковые литералы

```
size_t PQescapeStringConn(PGconn      *conn,  
                           char        *to,  
                           const char *from, // исходная строка  
                           size_t      length, // длина исходной строки  
                           int         *error); //
```

PQescapeStringConn( ) экранирует строковые литералы наподобие PQescapeLiteral( ). Однако, в отличие от PQescapeLiteral(), за предоставление буфера надлежащего размера отвечает вызывающая функция.

Более того, PQescapeStringConn( ) не добавляет одинарные кавычки, которые должны окружать строковые литералы Postgres – они должны быть включены в SQL-команду, в которую вставляется результирующая строка.

**from** – указывает на первый символ строки, которая должна экранироваться.

**length** – задаёт число байт в этой строке.

Завершающий нулевой байт не требуется и не должен учитываться в параметре length.

Если завершающий нулевой байт был найден до того, как были обработаны length байт, PQescapeStringConn( ) останавливает работу на нулевом байте (напоминает поведение strncpy).

**to** – должен указывать на буфер, который сможет вместить как минимум на один байт больше, чем предписывает удвоенное значение параметра length, в противном случае поведение функции не определено.

Поведение будет также не определено, если строки to и from перекрываются.

**error** – если параметр `error` не равен `NULL`, тогда значение `*error` устанавливается равным нулю в случае успешной работы и не равным нулю в случае ошибки.

В настоящее время единственным возможным условием возникновения ошибки является неверная мультибайтовая кодировка в исходной строке.

Выходная строка формируется даже при наличии ошибки, но можно ожидать, что сервер отвергнет её как неверно сформированную.

В случае ошибки в объект `conn` записывается соответствующее сообщение независимо от того, равно ли `NULL` значение параметра `error`.

`PQescapeStringConn( )` возвращает число байт, записанных по адресу `to`, не включая завершающий нулевой байт (похоже по поведению на `getline( )`).

## PQescapeIdentifier – экранировать строку – идентификатор SQL

```
char *PQescapeIdentifier(PGconn      *conn,  
                        const char *str,  
                        size_t      length);
```

PQescapeIdentifier( ) экранирует строку, предназначенную для использования в качестве идентификатора SQL, такого, как таблица, столбец или имя функции. Это полезно, когда идентификатор, выбранный пользователем, может содержать специальные символы, которые в противном случае не интерпретировались бы синтаксическим анализатором SQL, как часть идентификатора, или когда идентификатор может содержать символы верхнего регистра, и этот регистр требуется сохранить.

**str** – исходная строка.

PQescapeIdentifier( ) возвращает str, экранированную как SQL-идентификатор. Память под возврат распределяется с помощью функции malloc( ). Если возвращенное значение больше не требуется, эту память нужно освободить с помощью функции PQfreemem( ),

Завершающий нулевой байт не нужен и не должен учитываться в параметре length.

Если завершающий нулевой байт был найден до того, как были обработаны length байт, то PQescapeIdentifier( ) останавливает работу на нулевом байте.

В возвращённой строке все специальные символы будут заменены таким образом, что она будет надлежащим образом обработана, как SQL-идентификатор. Также будет добавлен завершающий нулевой байт.

Возвращённая строка также будет заключена в двойные кавычки.

В случае ошибки PQescapeIdentifier( ) возвращает NULL, и в объект conn помещается соответствующее сообщение.

## PQescapeByteaConn – экранировать двоичные данные

Экранирует двоичные данные для их использования внутри SQL-команды с типом данных bytea.

Эту функцию следует применять только тогда, когда данные вставляются непосредственно в строку SQL-команды.

```
unsigned char *PQescapeByteaConn(PGconn          *conn,  
                                const unsigned char *from,  
                                size_t              from_length,  
                                size_t              *to_length);
```

Если байты, имеющие определенные значения, используются в качестве составной части литерала, имеющего тип bytea, они должны в SQL операторе экранироваться.

Эта функция экранирует байты, используя либо hex-кодирование, либо экранирование с помощью обратной косой черты.

**from** – указывает на первый байт строки, которая должна экранироваться.

**from\_length** – задаёт число байт в этой двоичной строке. Завершающий нулевой байт не нужен и не учитывается.

**to\_length** – указывает на переменную, которая будет содержать длину результирующей экранированной строки. Эта длина включает завершающий нулевой байт.

PQescapeByteaConn( ) возвращает экранированную версию двоичной строки, на которую указывает параметр from, и размещает её в памяти, распределённой с помощью malloc( ). Эта память должна быть освобождена с помощью функции PQfreemem( ).

В возвращаемой строке все специальные символы заменены так, чтобы синтаксический анализатор литеральных строк Postgres и функция ввода для типа `bytea` могли обработать их надлежащим образом. Также добавляется завершающий нулевой байт.

Одинарные кавычки, которые должны окружать строковые литералы Postgres, не добавляются.

В случае ошибки возвращается нулевой указатель, и соответствующее сообщение об ошибке записывается в объект `conn`.

## PQunescapeBytea – строковое представление → в двоичные данные

Преобразует строковое представление двоичных данных в двоичные данные.

Является обратной функцией к функции PQescapeBytea( ).

Она нужна, когда данные типа bytea извлекаются в текстовом формате.

```
unsigned char *PQunescapeBytea(const unsigned char *from,  
                                size_t                *to_length);
```

**from** – указывает на строку, такую, какую могла бы вернуть функция PQgetValue( ), применённая к столбцу типа bytea. Функция преобразует это строковое представление в его двоичное представление.

Возвращает указатель на буфер, распределённый с помощью функции malloc( ) или NULL в случае ошибки, и помещает размер буфера по адресу to\_length.

Когда не будет нужен результат, необходимо освободить память, вызвав PQfreemem( ).

Это преобразование не является точной инверсией для PQescapeBytea( ), поскольку ожидается, что строка, полученная от PQgetValue( ), не будет «экранированной».