Базы данных

Лекция 01 – 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, 2024

2024.09.02-16

Оглавление

libpq — библиотека для языка C

libpq — это интерфейс Postgres для программирования приложений на языке C.

Библиотека libpq содержит набор функций, используя которые клиентские программы могут передавать запросы серверу Postgres и принимать результаты этих запросов.

libpq также является базовым механизмом для других прикладных интерфейсов Postgres, в частности, для C++, Perl, Python и Tcl.

Примеры использования `libpq`, в том числе несколько завершённых приложений, можно найти в каталоге `src/test/examples` дистрибутива в исходных текстах.

Клиентские программы, которые используют `libpq`, должны включать заголовочный файл `libpq-fe.h` и должны компоноваться с библиотекой `libpq`.

Интерфейс для C++ обычно находится в `libpqxx (libpqxx-devel)`.

Основные функции подразделяются на несколько категорий:

- управление подключением к базе данных;
- исполнения команд;
- извлечение информации, связанной с результатом запроса;
- получение дополнительной информации о результате;
- экранирование строковых значений для включения в SQL-команды.

Функции управления подключением к базе данных

Прикладная программа может иметь несколько подключений к серверу, открытых одновременно. Это нужно для доступа к более чем одной базе данных.

Каждое соединение представляется объектом типа **PGconn**, который можно получить от функций:

```
PQconnectdbParams;
PQconnectdb;
PQsetdbLogin.
```

В случае успеха эти функции возвращают ненулевой указатель на объект.

Прежде чем передавать запросы через объект подключения, следует вызвать функцию **PQstatus** для проверки возвращаемого значения в случае успешного подключения.

PQconnectdbParams – создаёт новое подключение к серверу баз данных

```
PGconn *PQconnectdbParams(const char *const *keywords,
                          const char *const *values,
                          int expand_dbname);
```

Функция открывает новое соединение с базой данных, используя параметры, содержащиеся в двух массивах, завершающихся символом NULL.

`keywords` – массив строк, каждая из которых представляет собой ключевое слово.

`values` – значение для каждого ключевого слова.

Набор параметров может быть расширен без изменения сигнатуры функции.

Ключевые слова

`host` – имя компьютера для подключения.

Если это имя начинается с косой черты, оно выбирает подключение через Unix-сокет, а не через TCP/IP, и задаёт имя каталога, содержащего файл сокета.

По умолчанию, если параметр `host` отсутствует или пуст, выполняется подключение к Unix-сокету в `/tmp`.

В системах, где Unix-сокеты не поддерживаются, по умолчанию выполняется подключение к `localhost`.

Может принимать разделённый запятыми список имён узлов. В данном случае имена будут перебираться по порядку.

Для пустых элементов списка применяется поведение по умолчанию.

`hostaddr` – числовой IP-адрес компьютера для подключения.

Он должен быть представлен в стандартном формате адресов IPv4, например, 172.28.40.9.

Есть поддержка IPv6, можно использовать и эти адреса.

Если в качестве этого параметра передана непустая строка, всегда используется связь по протоколу TCP/IP.

Использование `hostaddr` вместо `host` позволяет приложению избежать поиска на сервере имён, что может быть важно для приложений, имеющих ограничения по времени.

Тем не менее, имя компьютера требуется для некоторых методов аутентификации, а также для проверки полномочий на основе SSL-сертификатов.

Применяются следующие правила:

1) Если адрес `host` задаётся без `hostaddr`, осуществляется разрешение имени[1].

2) Если указан `hostaddr`, а `host` нет, значение `hostaddr` задает сетевой адрес сервера. Однако, если метод аутентификации требует наличия имени компьютера, попытка подключения завершится неудачей.

3) Если указаны как `host`, так и `hostaddr`, значение `hostaddr` задает сетевой адрес сервера, а значение `host` игнорируется. Если метод аутентификации его требует, оно будет использоваться в качестве имени компьютера.

Аутентификация может завершится неудачей, если `host` не является именем сервера с сетевым адресом `hostaddr`. Когда указывается и `host`, и `hostaddr`, соединение в файле паролей идентифицируется по значению `host`.

4) Если `hostaddr` является списком значений, разделенных запятыми, узлы будут перебираться по порядку.

5) Если не указаны ни имя компьютера, ни его адрес, `libpq` будет производить подключение, используя локальный Unix-сокет. В системах, не поддерживающих Unix-сокеты, будет попытка подключиться к `localhost`.

`port` – номер порта, к которому нужно подключаться на сервере.

Если в параметрах `host` или `hostaddr` задано несколько серверов, в данном параметре может задаваться через запятую список портов такой же длины, либо может указываться один номер порта для всех узлов.

Пустая строка или пустой элемент в списке через запятую воспринимается как номер порта по умолчанию, установленный при сборке PostgreSQL (5432).

---

1) Есть нюансы при использовании PQconnectPoll()

dbname – имя базы данных.
По умолчанию оно совпадает с именем пользователя.

user – имя пользователя Postgres, используемое для подключения.
По умолчанию используется то же имя, которое имеет в операционной системе пользователь, от лица которого выполняется приложение.

password – пароль, используемый в случае, когда сервер требует аутентификации по паролю.

passfile – имя файла, в котором будут храниться пароли.
По умолчанию это `~/.pgpass` или `%APPDATA%\postgresql\pgpass.conf` в Microsoft Windows. Файл может отсутствовать.

connect_timeout – максимальное время ожидания подключения.
Задаётся десятичным целым числом.
При нуле, отрицательном или если не указано, ожидание будет бесконечным.
Минимальный допустимый тайм-аут равен 2 секундам, таким образом, значение 1 воспринимается как 2 секунды.
Этот тайм-аут применяется для каждого отдельного IP-адреса или имени сервера.
Если заданы адреса двух серверов и значение `connect_timeout` равно 5, тайм-аут при неудачной попытке подключения к каждому серверу произойдёт через 5 секунд, а общее время ожидания подключения может достигать 10 секунд.

client_encoding – параметр устанавливает конфигурационный параметр client_encoding для данного подключения.

В дополнение к значениям, которые принимает соответствующий параметр сервера, можно использовать значение auto. В этом случае правильная кодировка определяется на основе текущей локали на стороне клиента.

options – задаёт параметры командной строки, которые будут отправлены серверу при установлении соединения.

application_name – устанавливает значение для конфигурационного параметра application_name.

fallback_application_name – устанавливает альтернативное значение для конфигурационного параметра application_name.

Это значение будет использоваться, если для параметра application_name не было передано никакого значения с помощью параметров подключения или переменной системного окружения PGAPPNAME.

Задание альтернативного имени полезно для универсальных программ-утилит, которые желают установить имя приложения по умолчанию, но позволяют пользователю изменить его.

keepalives – управляет использованием сообщений keepalive протокола TCP на стороне клиента. Значение по умолчанию равно 1, что означает использование сообщений. Можно изменить его на 0, если эти сообщения не нужны.

keepalives_idle — управляет длительностью периода отсутствия активности, выраженного числом секунд, по истечении которого TCP должен отправить сообщение keepalive серверу.

При значении 0 действует системная величина.

Этот параметр игнорируется для соединений, установленных через Unix-сокеты, или если сообщения keepalive отключены.

Он поддерживается только в системах, воспринимающих параметр сокета TCP_KEEPIDLE или равнозначный. В других системах он не оказывает влияния.

keepalives_count — задаёт количество сообщений keepalive протокола TCP, которые могут быть потеряны, прежде чем соединение клиента с сервером будет признано нерабо-тающим.

Нулевое значение этого параметра указывает, что будет использоваться системное значение по умолчанию.

Этот параметр игнорируется для соединений, установленных через Unix-сокеты, или если сообщения keepalive отключены.

Он поддерживается только в системах, воспринимающих параметр сокета TCP_KEEPCNT или равнозначный; в других системах он не оказывает влияния.

sslmode – определяет, будет ли согласовываться с сервером защищённое SSL-соединение по протоколу TCP/IP, и если да, то в какой очередности.

Всего предусмотрено шесть режимов:

`disable` – следует пытаться установить только соединение без использования SSL;

`allow` – сначала следует попытаться установить соединение без использования SSL, но если попытка будет неудачной, нужно попытаться установить SSL-соединение;

`prefer` (по умолчанию) – сначала следует попытаться установить SSL-соединение, но если попытка будет неудачной, нужно попытаться установить соединение без использования SSL;

`verify-ca` – следует попытаться установить только SSL-соединение, при этом проконтролировать, чтобы сертификат сервера был выпущен доверенным центром сертификации (CA);

`require` – следует попытаться установить только SSL-соединение. Если присутствует файл корневого центра сертификации, то нужно верифицировать сертификат таким же способом, как будто был указан параметр verify-ca;

`verify-full` – следует попытаться установить только SSL-соединение, при этом проконтролировать, чтобы сертификат сервера был выпущен доверенным центром сертификации (CA) и чтобы имя запрошенного сервера соответствовало имени в сертификате;

!!! Замечание !!!

`sslmode` игнорируется при использовании Unix-сокетов. В случаях, когда Postgres скомпилирован без поддержки SSL, использование параметров `require`, `verify-ca` или `verify-full` приведёт к ошибке. Параметры `allow` и `prefer` будут приняты, но `libpq` не будет пытаться установить SSL-соединение.

`sslcompression` – если установлено значение 1 (по умолчанию), данные, пересылаемые через SSL-соединения, будут сжиматься. Если установлено значение 0, сжатие будет отключено.

Параметр игнорируется, если выполнено подключение без SSL, или если используемая версия OpenSSL не поддерживает его.

Сжатие требует процессорного времени, но может улучшить пропускную способность, если узким местом является сеть.

Отключение сжатия может улучшить время отклика и пропускную способность, если ограничивающим фактором является производительность CPU.

`sslcert` – указывает имя файла для SSL-сертификата клиента, заменяющего файл по умолчанию `~/.postgresql/postgresql.crt`. Этот параметр игнорируется, если SSL-подключение не выполнено.

`sslkey` – указывает местоположение секретного ключа, используемого для сертификата клиента.

Он может либо указывать имя файла, которое будет использоваться вместо имени по умолчанию `~/.postgresql/postgresql.key`, либо он может указывать ключ, полученный от внешнего криптомодуля (загружаемого модуля OpenSSL).

`sslrootcert` – указывает имя файла, содержащего SSL-сертификаты, выданные Центром сертификации (CA). Если файл существует, сертификат сервера будет проверен на предмет его подписания одним из этих центров. Имя по умолчанию – `~/.postgresql/root.crt`.

`sslcrl` – указывает имя файла, содержащего список отозванных серверных сертификатов (CRL) для SSL. Сертификаты, перечисленные в этом файле, если он существует, будут отвергаться при попытке установить подлинность сертификата сервера. Имя по умолчанию такое `~/.postgresql/root.crl`.

`requirepeer` – указывает имя пользователя операционной системы, предназначенное для сервера, например, `requirepeer=postgres`.

При создании подключения через Unix-сокет, если этот параметр установлен, клиент проверяет в самом начале процедуры подключения, что серверный процесс запущен от имени указанного пользователя и, если это не так, соединение аварийно прерывается с ошибкой.

Этот параметр можно использовать, чтобы обеспечить аутентификацию сервера, подобную той, которая доступна с помощью SSL-сертификатов при соединениях по протоколу TCP/IP.

`krbsrvname` – имя сервиса Kerberos, предназначенное для использования при аутентификации на основе GSSAPI. Оно должно соответствовать имени сервиса, указанному в конфигурации сервера, чтобы аутентификация на основе Kerberos прошла успешно.

`gsslib` – библиотека GSS, предназначенная для использования при аутентификации на основе GSSAPI.

В настоящее время это действует только в сборках для Windows, поддерживающих одновременно и GSSAPI, и SSPI. Значение gssapi в таких сборках позволяет указать, что libpq

должна использовать для аутентификации библиотеку GSSAPI, а не подразумеваемую по умолчанию SSPI.

`service` – имя сервиса, используемое для задания дополнительных параметров. Оно указывает имя сервиса в файле `pg_service.conf`, который содержит дополнительные параметры подключения. Это позволяет приложениям указывать только имя сервиса, поскольку параметры подключения могут поддерживаться централизованно.

`target_session_attrs` – если этот параметр равен `read-write`, по умолчанию будут приемлемы только подключения, допускающие транзакции на чтение/запись.

При успешном подключении будет отправлен запрос `SHOW transaction_read_only`; если он вернёт on, соединение будет закрыто.

Если в строке подключения указано несколько серверов, будут перебираться остальные серверы, как и при неудачной попытке подключения. Со значением по умолчанию (any) приемлемыми будут все подключения.

PQconnectdb – создаёт новое подключение к серверу баз данных

```
PGconn *PQconnectdb(const char *conninfo);
```

Эта функция открывает новое соединение с базой данных, используя параметры, полученные из строки `conninfo`.

Передаваемая строка может быть пустой. В этом случае используются все параметры по умолчанию. Она также может содержать одно или более значений параметров, разделённых пробелами, или URI.

Пример:

```
host=localhost port=5432 dbname=mydb connect_timeout=10
```

PQsetdbLogin – создаёт новое подключение к серверу баз данных

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,      // не используется в н.в.
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

Предшественница функции PQconnectdb( ) с фиксированным набором параметров.

Она имеет такую же функциональность, за исключением того, что непереданные параметры (NULL или пустая строка) всегда принимают значения по умолчанию.

PQsetdb – создаёт новое подключение к серверу баз данных.

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

Это макрос. Он вызывает `PQsetdbLogin()` с нулевыми указателями в качестве значений параметров `login` и `pwd`.

Обеспечивает обратную совместимость с очень старыми программами.

Подключение к серверу баз данных неблокирующим способом

```
PGconn *PQconnectStartParams(const char *const *keywords,
                             const char *const *values,
                             int expand_dbname);

PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

Эти функции используются для того, чтобы открыть подключение к серверу баз данных таким образом, чтобы вызывающий их поток исполнения не был заблокирован при выполнении удаленной операции ввода/вывода во время подключения.

Суть этого подхода в том, чтобы ожидание завершения операций ввода/вывода могло происходить в главном цикле приложения, а не в внутри функций `PQconnectdbParams()` или `PQconnectdb()`, с тем, чтобы приложение могло управлять этой операцией параллельно с другой работой.

С помощью функции `PQconnectStartParams()` подключение к базе данных выполняется, используя параметры, взятые из массивов `keywords` и `values`.

С помощью функции PQconnectStart подключение к базе данных выполняется, используя параметры, взятые из строки `conninfo`.

Эти функции не блокируются до тех пор, пока выполняется ряд ограничений:

1) Параметр hostaddr должен использоваться так, чтобы для разрешения заданного имени не требовалось выполнять запросы DNS.

2) Если вызывается PQtrace, необходимо обеспечить, чтобы поток, в который выводится трассировочная информация, не заблокировался.

Детали в документации.

```
typedef enum {
    PGRES_POLLING_FAILED = 0,
    PGRES_POLLING_READING,    // These two indicate that one may
    PGRES_POLLING_WRITING,    // use select before polling again
    PGRES_POLLING_OK,
    PGRES_POLLING_ACTIVE         // не используется; обратная совместимость
} PostgresPollingStatusType;
```

PQconndefaults – возвращает значения по умолчанию для парам-ров подключения

Возвращает массив параметров подключения.

```
PQconninfoOption *PQconndefaults(void);

typedef struct {
    char   *keyword;   // Ключевое слово для данного параметра
    char   *envvar;    // Имя альтернативной переменной окружения
    char   *compiled;  // Альтернативное значение по умолчанию,
                       // назначенное при компиляции
    char   *val;       // Текущее значение параметра или NULL
    char   *label;     // Обозначение этого поля в диалоге подключения
    char   *dispchar;  // Показывает, как отображать это поле
                       // в диалоге подключения. Значения следующие:
                       // ""  Отображать введённое значение "как есть"
                       // "*" Поле пароля – скрывать значение
                       // "D" Параметр отладки
    int     dispsize;  // Размер поля в символах для диалога
} PQconninfoOption;
```

Массив параметров подключения может использоваться для определения всех возможных параметров PQconnectdb и их текущих значений по умолчанию.

Возвращаемое значение указывает на массив структур PQconninfoOption, который завершается элементом, имеющим нулевой указатель keyword.

Если выделить память не удалось, то возвращается нулевой указатель.

PQconninfo – Возвращает параметры подключения действующего соединения

```
PQconninfoOption *PQconninfo(PGconn *conn);
```

Возвращает массив параметров подключения.

Используется для определения всех возможных параметров PQconnectdb и значений, которые были использованы для подключения к серверу.

Возвращаемое значение указывает на массив структур PQconninfoOption, который завершается элементом, имеющим нулевой указатель keyword.

```
typedef struct _PQconninfoOption {
    char *keyword;  // The keyword of the option
    char *envvar;   // Fallback environment variable name */
    char *compiled; // Fallback compiled in default value */
    char *val;      // Option's current value, or NULL     */
    char *label;    // Label for field in connect dialog  */
    char *dispchar; // Indicates how to display this field in a connect
                    // dialog. Values are:
                    // "" Display entered value as is
                    // "*" Password field - hide value
                    // "D"  Debug option - don't show by default
    int  dispsize;  // Field size in characters for dialog
} PQconninfoOption;
```

PQfinish – Закрывает соединение с сервером

```
void PQfinish(PGconn *conn);
```

Освобождает память, используемую объектом PGconn.

Приложение в любом случае должно вызвать PQfinish( ), чтобы освободить память, используемую объектом PGconn, даже если попытка подключения к серверу потерпела неудачу (как показывает PQstatus).

После того, как была вызвана функция PGfinish, указатель PGconn не должен использоваться повторно.

PQreset – переустанавливает канал связи с сервером

```
void PQreset(PGconn *conn);
```

Эта функция закроет подключение к серверу, а потом попытается установить новое подключение, используя все те же параметры, которые использовались прежде. Это может быть полезным для восстановления после ошибки, если работающее соединение было разорвано.

Переустановка канала связи с сервером неблокирующим способом

PQresetStart
PQresetPoll

```
int PQreset Start(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

Эти функции закроют подключение к серверу, после чего попытаются установить новое подключение, используя все те же параметры, которые использовались прежде.

Это может быть полезным для восстановления после ошибки, если работающее соединение оказалось потерянным.

Они отличаются от PQreset тем, что действуют неблокирующим способом.

На эти функции налагаются те же ограничения, что и на PQconnectStartParams(), PQconnectStart() и PQconnectPoll().

Переустановка подключения PQresetStart() – если она возвратит 0, переустановка завершилась неудачно.

Если она возвратит 1, следует опросить результат переустановки, используя PQresetPoll().

Детали в документации

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.

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.

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 ... (как в unix shell)

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[2]

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

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

```
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 указывается имя оператора 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.

---

2) 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.

PREPARE — подготовить оператор к выполнению

```
PREPARE name [ ( data_type [, ...] ) ] AS statement
```

name – уникальное в рамках сеанса имя оператора;
data_type – тип данных параметра подготовленного оператора;
statement – любой оператор SELECT, INSERT, UPDATE, DELETE, MERGE или VALUES.

PREPARE создаёт подготовленный оператор. Это объект на стороне сервера, позволяющий оптимизировать производительность приложений.

Когда выполняется PREPARE, указанный оператор разбирается, анализируется и переписывается. При последующем выполнении команды EXECUTE подготовленный оператор планируется и исполняется. Такое разделение действий исключает повторный разбор запроса, при этом позволяет выбрать наилучший план выполнения в зависимости от определённых значений параметров.

Подготовленные операторы могут принимать параметры — значения, которые подставляются в оператор, когда он собственно выполняется. При создании подготовленного оператора к этим параметрам можно обращаться по порядковому номеру, используя запись $1, $2 и т. д. Дополнительно можно указать список соответствующих типов данных параметров. Если тип данных параметра не указан или объявлен как unknown (неизвестный), тип выводится из контекста при первом обращении к этому параметру (если это возможно). При выполнении оператора фактические значения параметров передаются команде EXECUTE.

Подготовленные операторы существуют только в рамках текущего сеанса работы с БД.

Когда сеанс завершается, система забывает подготовленный оператор, так что его надо будет создать снова, чтобы использовать дальше.

Это также означает, что один подготовленный оператор не может использоваться одновременно несколькими клиентами базы данных; но каждый клиент может создать собственный подготовленный оператор и использовать его.

Освободить подготовленный оператор можно вручную, выполнив команду DEALLOCATE.

Подготовленные операторы потенциально дают наибольший выигрыш в производительности, когда в одном сеансе выполняется большое число однотипных операторов.

Отличие в производительности особенно значительно, если операторы достаточно сложны для планирования или перезаписи, например, когда в запросе объединяется множество таблиц или необходимо применить несколько правил.

Если оператор относительно прост в этом плане, но сложен для выполнения, выигрыш от использования подготовленных операторов вряд ли будет заметным.

EXECUTE — выполнить подготовленный оператор

```
EXECUTE name [ ( parameter [, ...] ) ]
```

name – имя подготовленного оператора, подлежащего выполнению;

parameter – фактическое значение параметр{а|ов} подготовленного оператора. Это может быть выражение, выдающее значение, совместимое с типом данных соответствующего параметра, определённого при создании подготовленного оператора.

PQexecPrepared – запрос на исполнение подготовленного оператора[3]

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

```
PGresult *PQexecPrepared(PGconn     *conn,
                         const char *stmtName, // имя подготовленного оп
                         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.

---

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

PQdescribePrepared – получение информации о подготовленном операторе[4]

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.

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

PQclear – освободить память, связанную с PGresult

Освобождает область памяти, связанную с PGresult.

Результат выполнения каждой команды должен быть освобождён, если он больше не нужен, с помощью PQclear.

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

Можно держать объект PGresult под рукой до тех пор, пока он нужен.

Объект PGresult не исчезает, ни когда выдаётся новая команда, ни даже если соединение закрывается.

Чтобы от него избавиться, необходимо вызвать PQclear.

Если этого не делать, то в результате будут иметь место утечки памяти в приложении.

Дополнительные запросы о состоянии выполнения команды

PQresStatus – преобразовать код статуса в строку

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

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

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 ошибка.

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.

Извлечение информации, связанной с результатом запроса

Эти функции служат для извлечения информации из объекта `PGresult`, который представляет результат успешного запроса.

Только если 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`[5] 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);
```

---

5) `limits.h: #define INT_MAX  2147483647` (2^{31} - 1)

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.

PQfnumber – номер столбца по имени

Возвращает номер столбца, соответствующий данному имени столбца.

```
int PQfnumber(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 FOO, 2 AS "BAR";
```

будет получено:

```
PQfname(res, 0)              foo
PQfname(res, 1)              BAR
PQfnumber(res, "FOO")        0
PQfnumber(res, "foo")        0
PQfnumber(res, "BAR")       -1
PQfnumber(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 `InvalidOid`:
- 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 `InvalidOid` будут определены только в том случае, если будет включен заголовок для libpq.

Они будут принадлежать к одному из целочисленных типов.

PQftablecol – номер столбца в таблице 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 явно скопировать данные из `PGresult` 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);  // опци 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. Соответственно, эту операцию выполняет `PQescapeLiteral()`.

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

Эту память нужно освобождать с помощью функции `PQfreemem()`[6], 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()`.

---

6) `libpq-fe.h: extern void PQfreemem(void *ptr);`

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.

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.

`PQescapeIdentifier()` возвращает версию параметра `str`, экранированную как SQL-идентификатор, и размещённую в области памяти, распределённой с помощью функции `malloc()`.

Эту память нужно освобождать с помощью функции `PQfreemem()`, когда возвращённое значение больше не требуется.

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. Таким образом, поведение функции напоминает `strncpy()`.)

В возвращённой строке все специальные символы заменены таким образом, что она станет правильным SQL-идентификатором.

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.

Подсказка

Как и в случае со строковыми литералами, для того чтобы предотвратить атаки с помощью SQL-инъекций, SQL-идентификаторы должны экранироваться, когда они получены из ненадёжного источника.

PQescapeStringConn – экранировать строковые литералы

```
size_t PQescapeStringConn(PGconn *conn,
                          char *to, const char *from, size_t length,
                          int *error);
```

`PQescapeStringConn()` экранирует строковые литералы наподобие `PqescapeLiteral()`.

Но, 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 Pro – они должны быть отдельно 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 в параметре `length` не должен учитываться. Если завершающий нулевой байт был найден до того, как были обработаны `length` байт, то `PQescapeStringConn()` останавливает работу на нулевом байте. Таким образом, поведение функции напоминает `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.

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.

PQescapeByteaConn – экранировать двоичные данные

Экранирует двоичные данные для их использования внутри SQL-команды с типом данных bytea. Как и в случае с PQescapeStringConn( ), эта функция применяется 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);
```

Байты, имеющие определённые значения, должны экранироваться, когда они используются в качестве составной части литерала, имеющего тип bytea, в SQL-операторе. PQescapeByteaConn( ) экранирует байты, используя либо hex-кодирование, либо экранирование с помощью обратной косой черты. См. Раздел 8.4 для получения дополнительной информации.

from – указывает на первый байт строки, которая должна экранироватьсяю.

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 результирующей строки.

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( ), когда результирующая строка больше не нужна.

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

Одинарные кавычки, которые должны окружать строковые литералы Postgres Pro, не являются частью результирующей строки.

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

В настоящее время единственной возможной ошибкой может быть нехватка памяти для результирующей строки.

PQunescapeBytea – строковое представление → в двоичные данные

Преобразует строковое представление двоичных данных в двоичные данные.

Функция является обратной функцией к функции `PQescapeBytea()`.

Она нужна, когда данные типа `bytea` извлекаются в текстовом формате, но не когда они извлекаются в двоичном формате.

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

from – указывает на строку, такую, какую могла бы возвратить функция `PQgetvalue()`, применённая к столбцу типа `bytea`. `PQunescapeBytea()` преобразует это строковое представление в его двоичное представление. Она возвращает указатель на буфер, распределённый с помощью функции `malloc()` (или NULL в случае ошибки) и помещает размер буфера по адресу `to_length`. Когда результат не будет нужен, необходимо освободить его память, вызвав `PQfreemem()`.

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