СИСТЕМА
УПРАВЛЕНИЯ
БАЗАМИ
ДАННЫХ

ЛИНТЕР®

ЛИНТЕР БАСТИОН

ЛИНТЕР СТАНДАРТ

Qt-интерфейс

НАУЧНО-ПРОИЗВОДСТВЕННОЕ ПРЕДПРИЯТИЕ

РЕЛЭКС

Товарные знаки

РЕЛЭКС™, ЛИНТЕР® являются товарными знаками, принадлежащими АО НПП «Реляционные экспертные системы» (далее по тексту – компания РЕЛЭКС). Прочие названия и обозначения продуктов в документе являются товарными знаками их производителей, продавцов или разработчиков.

Интеллектуальная собственность

Правообладателем продуктов ЛИНТЕР® является компания РЕЛЭКС (1990-2026). Все права защищены.

Данный документ является результатом интеллектуальной деятельности, права на который принадлежат компании РЕЛЭКС.

Все материалы данного документа, а также его части/разделы могут свободно размещаться на любых сетевых ресурсах при условии указания на них источника документа и активных ссылок на сайты компании РЕЛЭКС: relex.ru и linter.ru.

При использовании любого материала из данного документа несетевым/печатным изданием обязательно указание в этом издании источника материала и ссылок на сайты компании РЕЛЭКС: relex.ru и linter.ru.

Цитирование информации из данного документа в средствах массовой информации допускается при обязательном упоминании первоисточника информации и компании РЕЛЭКС.

Любое использование в коммерческих целях информации из данного документа, включая (но не ограничиваясь этим) воспроизведение, передачу, преобразование, сохранение в системе поиска информации, перевод на другой (в том числе компьютерный) язык в какой-либо форме, какими-либо средствами, электронными, механическими, магнитными, оптическими, химическими, ручными или иными, запрещено без предварительного письменного разрешения компании РЕЛЭКС.

О документе

Материал, содержащийся в данном документе, прошел доскональную проверку, но компания РЕЛЭКС не гарантирует, что документ не содержит ошибок и пропусков, поэтому оставляет за собой право в любое время вносить в документ исправления и изменения, пересматривать и обновлять содержащуюся в нем информацию.

Контактные данные

394006, Россия, г. Воронеж, ул. Бахметьева, 2Б.

Тел./факс: (473) 2-711-711, 2-778-333.

e-mail: info@linter.ru.

Техническая поддержка

С целью повышения качества программного продукта ЛИНТЕР и предоставляемых услуг в компании РЕЛЭКС действует автоматизированная система учёта и обработки пользовательских рекламаций. Обо всех обнаруженных недостатках и ошибках в программном продукте и/или документации на него просим сообщать нам в раздел Поддержка на сайте ЛИНТЕР.

Содержание

Предисловие 3
Назначение документа 3
Для кого предназначен документ 3
Необходимые предварительные знания 3
Дополнительные документы 3
Общие сведения о Qt-библиотеке 4
Назначение 4
Условия применения 4
Характеристики программы 5
Обращение к программе 6
Использование в ОС Linux 6
Сборка QtLinter-драйвера 6
Установка QtLinter-драйвера 6
QtLinter-интерфейс 8
Управление соединением 8
Установка соединения с БД 8
Инициализация соединения 8
Создание объекта-соединения 8
Создание копии объекта-соединения 8
Уничтожение объекта-соединения 9
Конфигурирование соединения 9
Подключить драйвер базы данных 9
Установить параметры соединения 9
Установить имя ЛИНТЕР-сервера 11
Установить имя пользователя 12
Установить пароль пользователя 12
Просмотр параметров соединения 13
Получить параметры соединения 13
Получить имя ЛИНТЕР-сервера 14
Получить имя пользователя 14
Получить пароль пользователя 15
Получить имя драйвера 15
Проверить доступность драйвера 16
Проверить функциональность драйвера 16
Проверить состояние соединения 18
Проверить результат соединения с БД 18
Проверить существование соединения 19
Установить сконфигурированное соединение 20
Установить параметризуемое соединение 20
Закрыть соединение 21
Начать транзакцию 22
Подтвердить транзакцию 22
Удалить именованное соединение 23
Откатить транзакцию 24
Поддерживаемые типы данных 24
Обработка SQL-запросов 25
Получить номер последней обработанной записи 29
Работа с BLOB-данными 31
Получение BLOB-данных 31
Стандартный способ 31
Получение с помощью псевдозапроса 31
Получение с помощью устаревшего метода 34
Добавление BLOB-данных 34

Стандартный способ 34
Добавление с помощью псевдозапроса 35
Добавление с помощью устаревшего метода 37
Удаление BLOB-данных 37
Стандартный способ 37
Удаление с помощью псевдозапроса 37
Удаление с помощью устаревшего метода 38
Получение информации об объектах БД 38
Получить список таблиц БД 38
Получить описание первичного ключа таблицы 39
Получить описание строки таблицы 40
Обработка кодов завершения 41
Получить последний код завершения 41
Приложение. Устаревшие функции для работы с BLOB-данными 42
Получить порцию BLOB-данных 42
Добавить порцию BLOB-данных 43
Удалить BLOB-данные 44

 © Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

Предисловие

Назначение документа

Документ содержит описание QtLinter-драйвера, выполняющего доступ к СУБД ЛИНТЕР из приложений, разработанных с использованием среды разработки Qt.

Документ предназначен для СУБД ЛИНТЕР БАСТИОН 6.0 сборка 20.8, далее по тексту СУБД ЛИНТЕР.

Для кого предназначен документ

Документ предназначен для программистов, разрабатывающих приложения в среде разработки Qt с использованием СУБД ЛИНТЕР.

Необходимые предварительные знания

Для работы необходимо владеть:

- основами реляционных баз данных и языка баз данных SQL;
- классами и методами модуля SQL среды разработки Qt;
- навыками работы в соответствующей операционной системе на уровне простого пользователя.

Дополнительные документы

- Установка СУБД ЛИНТЕР в среде ОС Linux
- Установка СУБД ЛИНТЕР в среде ОС Windows
- Сетевые средства
- Архитектура СУБД
- Справочник по SQL
- Справочник кодов завершения
- Документация на систему разработки Qt.

Общие сведения о Qt-библиотеке

Назначение

Qt-библиотека – мультиплатформенная C++ библиотека для разработки высококачественных графических интерфейсов приложений. Qt является полностью объектно-ориентированной, легко расширяемой и простой в применении библиотекой. Qt включает в себя большой набор средств, облегчающих процесс разработки приложений, наиболее важными из которых являются Qt Designer, средство для визуального создания графического интерфейса приложений, и QtSQL, средство для создания платформенно-независимых приложений для работы с базами данных.

Qt включает «родные» драйвера для Oracle, Microsoft SQL Server, Sybase Adaptive Server, IBM DB2, PostgreSQL, MySQL и ODBC-совместимых баз данных.

Функциональность QtSQL полностью интегрирована с Qt Designer, который может отображать данные из БД «вживую». Qt включает специфичные для БД виджеты, а также поддерживает расширение для работы с БД любых встроенных или отдельно написанных виджетов.

Приложения на Qt могут расширять свою функциональность с помощью модулей («plugins») и динамических библиотек. Модули включают дополнительные кодеки, драйвера баз данных, форматы изображений, стили и виджеты. Библиотеки могут предложить неограниченное расширение функциональности.

Условия применения

Для выполнения Qt-приложения на компьютере должен быть установлен полный пакет среды разработки Qt.

Для корректной работы драйвера необходимо указать в переменной окружения PATH местоположение используемых драйвером библиотек СУБД ЛИНТЕР `inter325.dll`, `lapi325.dll` и `dectic32.dll`. Библиотеки располагаются в подкаталоге `/bin` установочного каталога СУБД ЛИНТЕР.

Если в системе определена переменная окружения `LINTER_HOME`, достаточно добавить в переменную окружения `PATH` путь `%LINTER_HOME%\bin`.

Примеры

1)

```
set PATH=C:\Linter\bin;%PATH%
```

2)

```
set LINTER_HOME=C:\Linter
set PATH=%LINTER_HOME%\bin;%PATH%
```


© Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

Характеристики программы

Драйвер Qt для СУБД ЛИНТЕР расширяет список баз данных, с которыми поддерживается Qt-интерфейс. Для работы с СУБД ЛИНТЕР драйвер использует модуль QtSQL.

Драйвер предназначен для работы только с Qt 4.x, Qt 5.x, Qt 6.x.

Версии драйвера Qt 2.x и Qt 3.x не поддерживаются.

Примечание
Если необходимая Вам версия отсутствует в перечне поддерживаемых версий, следует обратиться в раздел Поддержка на сайте ЛИНТЕР.

Обращение к программе

Использование в ОС Linux

Сборка QtLinter-драйвера

Сборка QtLinter-драйвера выполняется вручную.

Для сборки QtLinter-драйвера в составе дистрибутива СУБД ЛИНТЕР поставляются исходные файлы драйвера.

Для сборки драйвера необходимо:

1) установить среду разработки Qt, если это не было выполнено ранее;
2) установить дистрибутив СУБД ЛИНТЕР, выбрав компонент драйвера Qt в процессе установки (см. документ «Установка СУБД ЛИНТЕР в среде ОС Linux»);
3) проверить наличие переменной окружения `QTDIR`. Переменная должна содержать путь к установочному каталогу среды разработки Qt;
4) убедиться, что переменная окружения `PATH` содержит путь к подкаталогу `\bin` установочного каталога Qt;
5) из подкаталога `\linqt` установочного каталога СУБД ЛИНТЕР выполнить команды:

```
qmake linqt.pro
make
```

В результате в подкаталоге `\linqt` будет собрана разделяемая библиотека `libqsqllinter.so`.

Установка QtLinter-драйвера

Установка QtLinter-драйвера выполняется вручную.

Для видимости собранного драйвера средой разработки Qt необходимо разместить библиотеку `libqsqllinter.so` в специальном каталоге расширений SQL-драйверов Qt – `$QTDIR\plugins\sqldrivers`.

Использование в среде Windows

Сборка QtLinter-драйвера

Сборка QtLinter-драйвера выполняется вручную.

Для сборки QtLinter-драйвера в составе дистрибутива СУБД ЛИНТЕР поставляются исходные файлы драйвера. Сборку драйвера необходимо осуществлять тем компилятором, который соответствует сборке пакета Qt (MinGw или MSVC).

Для сборки драйвера необходимо:

1) установить среду разработки Qt, если это не было произведено ранее;
2) установить компилятор в соответствии со сборкой Qt (MSVC или набор инструмента разработки MinGw);

© Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

3) установить дистрибутив СУБД ЛИНТЕР, выбрав компонент драйвера Qt в процессе установки (см. документ «Установка СУБД ЛИНТЕР в среде ОС Windows»);

4) проверить наличие переменной окружения `QTDIR`. Переменная должна содержать путь к установочному каталогу среды разработки Qt;

5) убедиться, что переменная окружения `PATH` содержит путь к подкаталогу `\bin` установочного каталога Qt, а также к подкаталогу `\bin` установочного каталога СУБД ЛИНТЕР. В том числе при использовании пакета MinGw убедиться, что переменная окружения `PATH` также содержит путь к подкаталогу `\bin` инструмента разработки MinGw;

6) при сборке драйвера должна быть указана спецификация компилятора для утилиты `qmake` либо в переменной окружения `QMAKESPEC`, либо в параметре командной строки `-spec` непосредственно при вызове утилиты `qmake`.

   Например:

   - при использовании инструмента разработки MinGw:

```
QMAKESPEC=win32-g++
```

   - при использовании компилятора MSVC:

```
QMAKESPEC=win32-msvc
```

7) из подкаталога `\linqt` установочного каталога СУБД ЛИНТЕР выполнить команды:

   - при использования инструмента разработки MinGw:

```
qmake linqt.pro
mingw32-make
```

   - при использовании компилятора MSVC (команды необходимо выполнять из командной строки разработчика Visual Studio):

```
qmake linqt.pro
nmake
```

   Если переменная окружения `QMAKESPEC` не определена, то, как было сказано ранее, можно задать спецификацию компилятора через аргумент командной строки `-spec`. Например:

```
qmake linqt.pro -spec win32-g++
```

В результате в подкаталоге `\linqt` будет собрана динамическая библиотека `qsqllinter.dll`.

Установка QtLinter-драйвера

Установка QtLinter-драйвера выполняется вручную.

Для видимости собранного драйвера средой разработки Qt необходимо разместить библиотеку `qsqllinter.dll` в специальном каталоге расширений SQL-драйверов Qt – `%QTDIR%\plugins\sqldrivers`.

QtLinter-интерфейс

Управление соединением

Установка соединения с БД

Логический доступ к БД ЛИНТЕР из Qt-приложения реализуется с помощью методов (функций) абстрактного класса QSqlDatabase, входящего в состав среды разработки Qt. Реальный доступ к данным БД и управление пользовательскими SQL-запросами выполняет драйвер QtLinter, входящий в состав СУБД ЛИНТЕР.

Инициализация соединения

Создание объекта-соединения

Синтаксические правила

```
QSqlDatabase::QSqlDatabase();
```

Описание

Создает пустой QSqlDatabase-объект. Для дальнейшего использования объекта к нему необходимо применить методы `addDatabase()`, `removeDatabase()` и `database()`.

Возвращаемое значение

Указатель на созданный QSqlDatabase-объект или NULL.

См. также: addDatabase(), removeDatabase() и `database()`.

Пример

```
QSqlDatabase db = QSqlDatabase::QSqlDatabase(); //Qt 4.x
```

Создание копии объекта-соединения

Синтаксические правила

```
QSqlDatabase::QSqlDatabase(<объект>)
<объект>::= указатель на существующий QSqlDatabase-объект
```

Описание

Создает копию заданного QSqlDatabase-объекта.

Возвращаемое значение

Указатель на созданную копию QSqlDatabase-объекта.

Пример

```
QSqlDatabase * conn_user1 = QSqlDatabase::QSqlDatabase();
QSqlDatabase * conn_user2 =
 QSqlDatabase::QSqlDatabase(conn_user1);
```


© Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

Уничтожение объекта-соединения

Синтаксические правила

```
QSqlDatabase::~QSqlDatabase()
```

Описание

Уничтожает созданный объект-соединение и освобождает все выделенные ему ресурсы. Если уничтоженный объект был последним объектом-соединением, который использовался для доступа к БД, то соединение с БД автоматически закрывается.

Возвращаемое значение

Нет.

См. также: close().

Конфигурирование соединения

Подключить драйвер базы данных

Синтаксические правила

```
QSqlDatabase::addDatabase(<имя драйвера>[, <имя соединения>]);
<имя драйвера>::= символьный литерал или символьная переменная
<имя соединения>::= внутреннее имя данного соединения в Qt, в СУБД
 ЛИНТЕР не используется
```

Описание

Устанавливает тип драйвера, через который должен выполняться доступ к БД по данному соединению. Для СУБД ЛИНТЕР имя драйвера должно быть «QLINTER».

Возвращаемое значение

Нет.

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" ); //Qt 4.x
```

См. также: `database()`, removeDatabase().

Установить параметры соединения

Синтаксические правила

```
void QSqlDatabase::setConnectOptions(<параметры>)
<параметры>::= символьный литерал или символьная переменная
```

Описание

Устанавливает (или изменяет) специфичные для СУБД параметры соединения с БД. Эта операция должна выполняться до открытия соединения с БД, иначе она не имеет смысла.

Изменить параметры соединения можно и таким образом:

- закрыть соединение (функция close());

- вызвать данную функцию для установки нужных параметров соединения;
- заново открыть соединение (функция open()).

Символьная строка, задающая значение аргумента <параметры>, должна иметь формат:

<опция>[; <опция>…]

Значение <опция> зависит от СУБД, к которой выполняется соединение.

Для СУБД ЛИНТЕР допустимы следующие значения <опций>:

Обозначение опции	Значение опции	Примечание
AUTOCOMMIT	Задает режим транзакций	Значение по умолчанию
OPTIMISTIC	Задает режим транзакций **Примечание** Режим OPTIMISTIC устарел (использовать не рекомендуется).	
EXCLUSIVE	Задает режим транзакций	
ANSI	Задает кодировку соединения	
KOI8	Задает кодировку соединения	Значение по умолчанию

Примечание
Если аргумент <параметры> содержит несколько однотипных опций (например, "PESSIMISTIC;AUTOCOMMIT;READ_COMMITTED"), то будет использоваться последняя опция.

Возвращаемое значение

Нет.

Примеры

```
  ...
// Соединение с СУБД ЛИНТЕР
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
db.setUserName( "SYSTEM" );
db.setPassword( "MANAGER8" );
db.setConnectOptions("OPTIMISTIC;CHARSET=KOI8"); // установить
 опции соединения
if (!db.open()) {
  db.setConnectOptions(); // очистить строку опций соединения
}

// Соединение с MySQL
```

 © Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

```
db.setConnectOptions("CLIENT_SSL=1;CLIENT_IGNORE_SPACE=1");
if (!db.open()) {
  db.setConnectOptions();
}

// Соединение через ODBC
db.setConnectOptions("SQL_ATTR_ACCESS_MODE=SQL_MODE_READ_ONLY;
 SQL_ATTR_TRACE=SQL_OPT_TRACE_ON");
if (!db.open()) {
  db.setConnectOptions();
}
```

См. также: connectOptions().

Установить имя ЛИНТЕР-сервера

Синтаксические правила

```
void QSqlDatabase::setDatabaseName(<имя>)
<имя>::= символьный литерал или символьная переменная длиной не
 более 8 символов
```

Описание

Задает ЛИНТЕР-сервер, к которому должен быть установлен доступ через соединение. ЛИНТЕР-сервер должен быть определен в файле сетевой конфигурации `nodetab` (см. документ «Сетевые средства»).

Назначение ЛИНТЕР-сервера должно выполняться до открытия соединения, в противном случае делать это не имеет смысла. Если же соединение уже открыто, необходимо выполнить метод close(), затем данную функцию, и после этого снова открыть соединение с помощью функции open().

Если ЛИНТЕР-сервер не был установлен, соединение будет установлено с локальным ЛИНТЕР-сервером по умолчанию.

Возвращаемое значение

Нет.

Пример

```
QSqlDatabase db;
db.setDatabaseName("conn_DB_SALE"); // Установить имя сервера
db.setUserName("SYSTEM");
db.setPassword("MANAGER8");
db.open();

QSqlQuery query(db);
query.exec("SELECT count(*) FROM AUTO WHERE make = 'BMW'");
```

См. также: databaseName(), setUserName(), setPassword(), `setHostName()`, `setPort()`, setConnectOptions(), open().

Установить имя пользователя

Синтаксические правила

```
void QSqlDatabase::setUserName(<пользователь>)
<пользователь>::= символьный литерал или символьная переменная
 длиной не более 66 символов
```

Описание

Задает пользователя, от имени которого должен выполняться доступ к БД (может быть пустым).

Назначение пользователя соединения должно выполняться до открытия соединения, в противном случае делать это не имеет смысла. Если же соединение уже открыто, необходимо вызвать функцию close(), затем данную функцию и после этого снова открыть соединение с помощью функции open().

Значения по умолчанию нет.

Возвращаемое значение

Нет.

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
db.setUserName( "SYSTEM" );
cout << "User name " << db.userName().toLocal8Bit().data() <<
 "\n";
db.setPassword( "MANAGER8" );
cout << "Password " << db.password().toLocal8Bit().data() << "\n";

if( !db.open() )
{
cout << db.lastError().driverText().toLocal8Bit().data() << endl;
return 1;
}
```

См. также: userName(), setDatabaseName(), setPassword(), `setHostName()`.

Установить пароль пользователя

Синтаксические правила

```
void QSqlDatabase::setPassword(<пароль>)
<пароль>::= символьный литерал или символьная переменная длиной не
 более 18 символов
```

Описание

Задает пароль, который должен использоваться при авторизации доступа к БД по заданному соединению (может быть пустым).

© Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

Назначение пароля должно выполняться до открытия соединения, в противном случае делать это не имеет смысла. Если же соединение уже открыто, необходимо вызвать функцию close(), затем данную функцию и после этого снова открыть соединение с помощью функции open().

Значения по умолчанию нет.

Примечание
Использование данной функции приводит к запоминаю пароля внутри Qt-интерфейса, что чревато его вскрытием. Чтобы избежать этого, необходимо использовать функцию open(<пользователь>, <пароль>), в которой пароль передается как параметр. В этом случае значение пароля в Qt-интерфейсе задействовано только на время выполнения операции открытия соединения.

Возвращаемое значение

Нет.

Пример

См. setDatabaseName().

См. также: password(), setUserName(), setDatabaseName(), `setHostName()`, `setPort()`, setConnectOptions(), open().

Просмотр параметров соединения

Получить параметры соединения

Синтаксические правила

```
QSqlDatabase::connectOptions() const
```

Описание

Предоставляет параметры указанного соединения.

Возвращаемое значение

Символьная строка переменной длины, содержащая список опций соединения, разделенных знаком «;» (точка с запятой).

Возможные значения опций приведены в описании функции setConnectOptions().

Возвращаемая строка может быть пустой (это означает, что используются опции по умолчанию).

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
db.setUserName( "SYSTEM" );
db.setPassword( "MANAGER8" );
db.setConnectOptions("EXCLUSIVE;CHARSET=KOI8");
cout << "options " << db.connectOptions().toLocal8Bit().data()<<
 "\n";
```

На консоль будет выведена строка `EXCLUSIVE;CHARSET=KOI8`

Получить имя ЛИНТЕР-сервера

Синтаксические правила

```
QSqlDatabase::databaseName();
```

Описание

Предоставляет имя ЛИНТЕР-сервера, к которому должен выполняться доступ по данному соединению.

Возвращаемое значение

Символьная строка длиной не более 8 символов. Если соединение установлено с ЛИНТЕР-сервером по умолчанию, строка содержит пробелы.

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
db.setDatabaseName("conn_DB_SALE ");
db.setUserName( "SYSTEM" );
db.setPassword( "MANAGER8" );
cout << "databaseName " << db.databaseName().toLocal8Bit().data()
 << "\n";
```

См. также: setDatabaseName().

Получить имя пользователя

Синтаксические правила

```
QSqlDatabase::userName();
```

Описание

Предоставляет значение, которое должно использоваться в качестве имени пользователя БД при открытии соединения (т.е. имя пользователя, заданное ранее при помощи функции setUserName()).

Возвращаемое значение

Символьное значение длиной до 66 символов (может быть пустым).

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
db.setUserName( "SYSTEM" );
cout << "User name " << db.userName().toLocal8Bit().data() <<
 "\n";
```

На консоль будет выведена строка: `User name SYSTEM`

См. также: setUserName().

© Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

Получить пароль пользователя

Синтаксические правила

```
QString QSqlDatabase::password() const
```

Описание

Предоставляет пароль пользователя, от имени которого выполнено (или должно выполняться) соединение с БД.

Возвращаемое значение

Символьная строка длиной до 18 символов.

Пустая строка возвращается в следующих случаях:

- пароль не был предварительно задан с помощью функции setPassword();
- пароль использовался временно (как параметр функции open(<пользователь>, <пароль>) при открытии соединения);
- пользователь, установивший соединение с БД, не имеет пароля.

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
db.setPassword( "MANAGER8" );
cout << "Password " << db.password().toLocal8Bit().data() << "\n";

На консоль будет выведена строка:

Password MANAGER8
```

См. также: setPassword(), userName().

Получить имя драйвера

Синтаксические правила

```
QSqlDatabase::driverName();
```

Описание

Предоставляет имя драйвера, используемого для доступа к БД по указанному соединению.

Возвращаемое значение

Символьная строка длиной 8 символов.

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
cout << "driver name" << db.driverName().toLocal8Bit().data() <<
 "\n"; //на консоль будет выведена строка QLINTER
```

См. также: addDatabase(), driver().

Проверить доступность драйвера

Синтаксические правила

```
QSqlDatabase::isDriverAvailable(<имя драйвера>);
<имя драйвера>::= символьный литерал или символьная переменная
```

Описание

Предоставляет информацию о доступности указанного драйвера (метод статический).

Возвращаемое значение

Логическое значение:

- `true` – драйвер доступен (установлен в Qt-интерфейсе);
- `false` – драйвер недоступен (неверное имя драйвера или драйвер не установлен в Qt-интерфейсе).

Пример

```
QSqlDatabase db = QSqlDatabase::QSqlDatabase();
cout << "Linter driver is available: " <<
 QSqlDatabase::isDriverAvailable("QLINTER")<< "\n";
```

См. также: `drivers()`.

Проверить функциональность драйвера

Синтаксические правила

```
bool QSqlDriver::hasFeature(<параметр>);
<параметр>::= идентификатор проверяемой функциональности
```

(элемент перечисления `QSqlDriver::DriverFeature` в таблице 1)

Таблица 1. Список доступных для проверки функциональных возможностей драйвера

Элемент перечисления QSqlDriver::DriverFeature	Результат проверки	Описание
QSqlDriver::Transactions	true	Поддержка SQL транзакций, т.е. поддерживаются методы: QSqlDatabase::transaction(); QSqlDatabase::commit(); QSqlDatabase::rollback()
QSqlDriver::QuerySize	true	Поддержка получения информации о результате выполнения запроса, то есть поддержка методов: QSqlQuery::size() – возвращает количество записей в выборке данных, QSqlQuery::numRowsAffected() – возвращает количество реально добавленных, измененных или удаленных записей при выполнении операции insert, update, delete
QSqlDriver::BLOB	false	Автоматическая загрузка BLOB-данных в оперативную память не

Элемент перечисления QSqlDriver::DriverFeature	Результат проверки	Описание
		поддерживается. Работа с BLOB-данными должна выполняться только с помощью эксклюзивных методов QtLinter-драйвера (см. раздел Работа с BLOB-данными)
QSqlDriver::Unicode	false	Поддержка драйвером данных в Unicode-кодировке при работе с СУБД, которые не поддерживают Unicode-кодировку. Так как СУБД ЛИНТЕР поддерживает Unicode-кодировку, то данная функциональная возможность драйвера не требуется
QSqlDriver::PreparedQueries	true	Поддержка использования претранслированных запросов
QSqlDriver::NamedPlaceholders	true	Поддержка именованных контейнеров (placeholder), заполняемых значениями именованных параметров претранслированного запроса
QSqlDriver::PositionalPlaceholders	true	Поддержка именованных контейнеров (placeholder), заполняемых значениями неименованных (позиционных) параметров претранслированного запроса
QSqlDriver::LastInsertId	true	Поддержка предоставления информации о номере последней обработанной записи, то есть поддержка метода QVariant QSqlQuery::lastInsertId()
QSqlDriver::BatchOperations	true	Поддержка выполнения нескольких операций в виде пакета
QSqlDriver::SimpleLocking	false	Поддержка запрета на запись в таблицу во время чтения из неё данных
QSqlDriver::LowPrecisionNumbers	false	Поддержка выборки численных значений с низкой точностью на уровне драйвера. В СУБД ЛИНТЕР точность значений обеспечивается непосредственно ядром СУБД с помощью указания соответствующего типа данных, поэтому данная функциональность не требуется
QSqlDriver::EventNotifications	false	Поддержка уведомлений СУБД о событиях при обработке данных
QSqlDriver::FinishQuery	false	Поддержка выполнения низкоуровневого освобождения ресурсов при вызове QSqlQuery::finish()
QSqlDriver::MultipleResultSets	false	Поддержка доступа к результирующим наборам данных, возвращаемым из пакетных операторов или хранимых процедур

Описание

Предоставляет информацию о поддержке драйвером QtLinter запрошенной функциональной возможности (метод статический).

Возвращаемое значение

Логическое значение:

- `true` – драйвер поддерживает проверяемую функциональную возможность;
- `false` – задана проверка неизвестной функциональной возможности, или проверяемая функциональная возможность не поддерживается.

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
db.driver()->hasFeature(QSqlDriver::Transactions);
db.driver()->hasFeature(QSqlDriver::QuerySize);
db.driver()->hasFeature(QSqlDriver::BLOB);
db.driver()->hasFeature(QSqlDriver::Unicode);
db.driver()->hasFeature(QSqlDriver::PreparedQueries);
```

Проверить состояние соединения

Синтаксические правила

```
bool QSqlDatabase::isOpen() const
```

Описание

Предоставляет информацию о текущем состоянии указанного соединения.

Возвращаемое значение

Логическое значение:

- `true` – соединение открыто (активно);
- `false` – соединение закрыто (не активно). QSqlDatabase-объект, соответствующий этому соединению, не уничтожен, и может быть использован повторно.

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
db.setUserName( "SYSTEM" );
db.setPassword( "MANAGER8" );
db.open();
cout << "Linter connection is open: " << db.isOpen() << "\n"; //
 выдает 1
db.close();
cout << "Linter connection is open: " << db.isOpen() << "\n"; //
 выдает 0
db.open();
cout << "Linter connection is open: " << db.isOpen() << "\n"; //
 выдает 1
```

Проверить результат соединения с БД

Синтаксические правила

```
bool QSqlDatabase::isOpenError() const
```

 © Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

Описание

Предоставляет информацию о результате выполнения операции соединения с БД.

Возвращаемое значение

Логическое значение:

- `true` – соединение установлено успешно;
- `false` – соединение не установлено. Причину ошибки можно узнать с помощью функции lastError().

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
db.setUserName( "SYSTEM" );
db.setPassword( "MANAGER8" );
db.open();
cout << "is OpenError: " << db.isOpenError()<< "\n";
```

Проверить существование соединения

Синтаксические правила

```
bool QSqlDatabase::isValid() const
```

Описание

Предоставляет информацию о существовании именованного соединения.

Возвращаемое значение

Логическое значение:

- `true` – соединение существует;
- `false` – соединение не существует.

Пример

```
{
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
db.setUserName( "SYSTEM" );
db.setPassword( "MANAGER8" );

db.open();
cout << "connection valid: " << db.isValid()<< "\n";

if( !db.open() )
{
  cout << db.lastError().driverText().toLocal8Bit().data() <<
 endl;
  return 1;
```

```
}

cout << "connectionName " <<
 db.connectionName().toLocal8Bit().data()<< "\n";
QString str = db.connectionName();

}db.removeDatabase(str);
cout << "connection valid: " << db.isValid()<< "\n";
```

Установить сконфигурированное соединение

Синтаксические правила

```
bool QSqlDatabase::open()
```

Описание

Открывает соединение с БД в соответствии с параметрами, установленными при инициализации (конфигурировании) объекта-соединения.

Возвращаемое значение

Логическое значение:

- `true` – соединение с БД установлено;
- `false` – ошибка открытия соединения. Подробную информацию о причине ошибки можно получить с помощью функции lastError().

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
db.setUserName( "SYSTEM" );
db.setPassword( "MANAGER8" );
if( !db.open() )
{
  cout << db.lastError().driverText().toLocal8Bit().data() <<
 endl;
  return 1;
}
```

См. также: lastError(), setDatabaseName(), setUserName(), setPassword(), `setHostName()`, `setPort()`, setConnectOptions().

Установить параметризуемое соединение

Синтаксические правила

```
bool QSqlDatabase::open(<пользователь>, <пароль>)
<пользователь>::= символьный литерал или символьная переменная
 длиной до 66 символов
<пароль>::= символьный литерал или символьная переменная длиной до
 18 символов
```

 © Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

Описание

Открывает соединение с БД в соответствии с текущими параметрами QSqlDatabase-объекта, но с новыми регистрационными данными `<пользователь>` и `<пароль>`. Параметры `<пользователь>` и `<пароль>` не запоминаются в QSqlDatabase-объекте, а передаются непосредственно драйверу для открытия соединения, после чего становятся недоступными.

Возвращаемое значение

Логическое значение:

- `true` – соединение с БД установлено;
- `false` – ошибка открытия соединения. Подробную информацию о причине ошибки можно получить с помощью функций lastError().

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER");

if( !db.open("SYSTEM", "MANAGER8") )
{
  cout << db.lastError().driverText().toLocal8Bit().data() <<
 endl;
  return 1;
}
```

См. также: lastError().

Закрыть соединение

Синтаксические правила

```
void QSqlDatabase::close();
```

Описание

Закрывает соединение с БД, освобождая все задействованные ресурсы.

Возвращаемое значение

Нет.

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER");

if( !db.open("SYSTEM", "MANAGER8") )
{
  cout << db.lastError().driverText().toLocal8Bit().data() <<
 endl;
  return 1;
}

db.close();
```

См. также: removeDatabase().

Начать транзакцию

Синтаксические правила

```
bool QSqlDatabase::transaction();
```

Описание

Объявляет о начале транзакции. Текущая транзакция (если есть) завершается с фиксацией изменений в БД (функция commit()).

Возвращаемое значение

Логическое значение:

- `true` – успешный старт транзакции;
- `false` – транзакция не стартовала.

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
db.setUserName( "SYSTEM" );
db.setPassword( "MANAGER8" );

db.open();db.transaction();
QSqlQuery query;
query.exec("DELETE FROM AUTO WHERE make = 'FIAT'");
query.exec("SELECT model FROM AUTO WHERE make = 'FIAT'");
if (query.next())
// что-то не так – откатываем транзакцию
  db.rollback();
else
  db.commit();
```

См. также: hasFeature(), commit(), rollback().

Подтвердить транзакцию

Синтаксические правила

```
bool QSqlDatabase::commit();
```

Описание

Завершает текущую транзакцию в соединении (если транзакция была ранее инициирована при помощи функции transaction()).

Возвращаемое значение

Логическое значение:

- `true` – транзакция завершена успешно;
- `false` – ошибка при завершении транзакции.

© Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

Пример

См. функцию transaction().

См. также: hasFeature(), rollback().

Удалить именованное соединение

Синтаксические правила

```
void QSqlDatabase::removeDatabase(<имя соединения>)
<имя соединения>::= символьный литерал или символьная переменная
```

Описание

Удаляет объект-соединение с заданным <именем соединения> из списка декларированных соединений.

В момент вызова данной функции соединение не должно содержать обрабатываемых SQL-запросов, в противном случае возможна потеря выделенных для обработки запроса ресурсов.

Возвращаемое значение

Нет.

Пример

```
// Неправильный код
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER",
 "connection_name");
if( !db.open("SYSTEM", "MANAGER8") )
{
  cout << db.lastError().driverText().toLocal8Bit().data() <<
 endl;
  return 1;
}
QSqlQuery query("SELECT count(*) FROM AUTO", db);
QSqlDatabase::removeDatabase("connection_name");
// будет выдано предупреждение
// "db" теперь ссылается на несуществующее соединение,
// "query" содержит неправильные данные

// Правильный код:
{
  QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER",
 "connection_name");
  if( !db.open("SYSTEM", "MANAGER8") )
  {
    cout << db.lastError().driverText().toLocal8Bit().data() <<
 endl;
    return 1;
```

```
  }
  QSqlQuery query("SELECT count(*) FROM AUTO", db);
}
// Объекты "db" и "query" уничтожены, потому что они больше не
 нужны
QSqlDatabase::removeDatabase("connection_name");
// правильная операция
```

См. также: `database()`.

Откатить транзакцию

Синтаксические правила

```
bool QSqlDatabase::rollback();
```

Описание

Выполняет откат текущей транзакции в заданном соединении. Транзакция должна быть предварительно инициирована с помощью функции transaction().

Возвращаемое значение

Логическое значение:

- `true` – откат транзакции выполнен успешно;
- `false` – при откате транзакции зафиксирована ошибка.

Пример

См. функцию transaction().

См. также: hasFeature(), commit().

Поддерживаемые типы данных

Соответствие типов данных СУБД ЛИНТЕР и стандартного Qt-интерфейса приведено в таблицах 2, 3.

Таблица 2. Соответствие типов данных при выборке из БД

Тип данных LINAPI-интерфейса	Тип данных Qt-интерфейса
tSmallInt	QVariant::Int
tInt	QVariant::Int
tBigInt	QVariant::LongLong QVariant::Double
tByte	QVariant::ByteArray
tVarByte	
tChar	QVariant::String
tVarChar	
tNChar	QVariant::String

© Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

Тип данных LINAPI-интерфейса	Тип данных Qt-интерфейса
tNVarChar	
tDate	QVariant::DateTime
tReal	QVariant::Double
tDouble	QVariant::Double
tDecimal	QVariant::Double
tBlob	QVariant::ByteArray
tBoolean	QVariant::Bool
tExtFile	QVariant::String

Таблица 3. Соответствие типов данных при добавлении (модификации) данных БД

Тип данных Qt-интерфейса	Тип данных LINAPI-интерфейса
QVariant::Int	tInt
QVariant::LongLong	tBigInt
QVariant::ByteArray	tByte
QVariant::String	tChar
QVariant::String	tNChar
QVariant::DateTime	tDate
QVariant::Double	tDouble
QVariant::Bool	tBoolean

Обработка SQL-запросов

Синтаксические правила

```
QSqlQuery QSqlDatabase::exec(<SQL-запрос>)
<SQL-запрос>::= символьный литерал или символьная переменная
```

Описание

Выполняет подготовленный (без параметров) SQL-запрос.

Для получения кода завершения выполненного SQL-запроса используется функция lastError(). Если текст <SQL-запроса> пуст, то запрос не выполняется и код завершения не генерируется.

Возвращаемое значение

Указатель на QSqlQuery-объект.

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER");
db.setUserName( "SYSTEM" );
db.setPassword( "MANAGER8" );

if( !db.open() )
{
```

```
  cout << db.lastError().driverText().toLocal8Bit().data() <<
 endl;
  return 1;
 }

QSqlQuery q;
QSqlQuery q = db.exec( QString( "select name from person;" ) );

while ( q.next() )
{
  cout << q.value(0).toString().toLocal8Bit().data() << "\n";
}
```

Первый вызов `next()` позиционирует `QSqlQuery` на **первую** запись в наборе данных. Последующие вызовы `next()` передвигают указатель на следующую запись и так до тех пор, пока не будет достигнут конец выборки. В этой точке `next()` вернет **false**.

Функция `value()` возвращает значение поля в виде `QVariant`. Поля нумеруются, начиная с 0, в порядке их следования в предложении `SELECT`. Класс `QVariant` может хранить большое количество типов данных языка C++ и Qt, в том числе `int` и `QString`. Различные типы данных, которые могут храниться в БД, переводятся в соответствующие типы C++ и Qt, и сохраняются в виде `QVariant`. Например, `VARCHAR` представляется в виде `QString`, а `DATETIME` – как `QDateTime`.

Класс `QSqlQuery` предоставляет целый набор функций для навигации по набору данных: `first()`, `last()`, `prev()`, `seek()` и `at()`. Они удобны в использовании, но при больших объемах выборки могут оказаться медлительными и ресурсоемкими. С целью оптимизации при работе с большими наборами данных можно вызвать `QSqlQuery::setForwardOnly(true)` перед `exec()`, а затем выполнять просмотр набора данных с помощью `next()` (в этом случае получаем однонаправленные наборы данных, т.е. такие наборы, навигация по которым может осуществляться только вперед, с помощью `next()`).

SQL-запрос можно передавать не только как аргумент функции `exec()`, но и напрямую, конструктору `QSqlQuery` (в этом случае функция `exec()` вызывается автоматически из конструктора):

```
QSqlQuery query("SELECT title, year FROM cd WHERE year >= 1998");
```

Проверку на наличие ошибок и выдачу диагностического сообщения можно выполнить следующим образом:

```
if (!query.isActive())
  cout << query.lastError().driverText().toLocal8Bit().data() <<
 endl;
```

Выполнение SQL-запросов манипулирования данными аналогично запросам выборки данных:

```
QSqlQuery query("INSERT INTO cd (id, artistid, title, year) "
 "VALUES (203, 102, 'Living in America', 2002)");
```

После выполнения такого запроса `QSqlQuery::numRowsAffected()` возвращает количество добавленных записей.

 © Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

При необходимости вставить в SQL-запрос значения переменных или, когда нежелательно, или невозможно перевести аргументы SQL-предложения в строковый вид, можно построить параметризованный запрос с помощью функции `prepare()`. Текст параметризованного запроса вместо реальных значений содержит параметры, которые заполняются фактическими значениями после создания запроса, например:

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
db.setUserName( "SYSTEM" );
db.setPassword( "MANAGER8" );
if( !db.open() )
{
  cout << db->lastError().driverText().toLocal8Bit().data() <<
 endl;
  return 1;
}
QSqlQuery query(db);
query.prepare("INSERT INTO cd (year, make, model, color) "
              "VALUES (?, ?, ?, ?)");
query.addBindValue(70);
query.addBindValue(QString ("FORD"));
query.addBindValue(QString("PANTERA"));
query.addBindValue(QString("BLACK"));
query.exec();
```

После создания запроса (вызовом `prepare()`), параметры запроса заполняются фактическими значениями с помощью функции `bindValue()` или `addBindValue()`, после чего запрос исполняется вызовом `exec()`. Параметризованные запросы можно выполнять в цикле. Перед началом цикла создается запрос, а в теле цикла производится заполнение параметров новыми значениями и исполнение запроса.

Параметризованные запросы очень часто используются в тех случаях, когда в БД нужно записать двоичные данные или строки, которые содержат символы из наборов, не принадлежащих диапазону ASCII или Unicode.

Каждое соединение с БД может поддерживать только одну активную транзакцию, поэтому множественные подключения могут оказаться полезными в том случае, когда необходимо одновременно запустить несколько транзакций. При использовании нескольких соединений в Qt-приложении имеется одно неименованное соединение, которое используется по умолчанию объектами `QSqlQuery`, если им явно не указать с каким соединением они должны работать.

В дополнение к `QSqlQuery`, Qt предоставляет класс `QSqlCursor`, производный от `QSqlQuery`. Этот класс расширяет функциональность предка большим числом дополнительных методов, которые позволяют отказаться от написания SQL-запросов для SQL-операций, таких как: `SELECT`, `INSERT`, `UPDATE` и `DELETE`.

Следующий пример демонстрирует выполнение и обработку результата SELECT-запроса с помощью методов класса `QSqlCursor`:

```
QSqlCursor cursor("cd");
cursor.select("year >= 1998");
```

Эквивалентный вариант с использованием `QSqlQuery`:

```
QSqlQuery query("SELECT id, artistid, title, year FROM cd " "WHERE
 year >= 1998");
```

Навигация по выборке выполняется точно так же, как и в `QSqlQuery`, за одним исключением – вместо порядкового номера поля функции `value()` можно передать его имя:

```
while (cursor.next()) {
  cout << cursor.value("title").toString().toLocal8Bit().data() <<
 "\n";
}
```

Для вставки записи в таблицу предварительно нужно создать новую запись `QSqlRecord` с помощью вызова `primeInsert()`, а затем для каждого из полей вызвать функцию `setValue()`. После всего этого можно выполнить вставку записи функцией `insert()`:

```
QSqlCursor cursor("cd");
QSqlRecord buffer = cursor.primeInsert();
buffer.setValue("year", 71);
buffer.setValue("model", "BUICK");
buffer.setValue("make", "FIAT");
buffer.setValue("color", "BLACK");
cursor.insert();
```

Чтобы изменить запись, нужно:

- позиционировать `QSqlCursor` на запись, которая должна подвергнуться изменениям (например, с помощью `select()` и `next()`);
- получить указатель на `QSqlRecord` вызовом `primeUpdate()`;
- записать новые значения функцией `setValue()`;
- вызвать `update()`, чтобы отправить сделанные изменения в базу данных:

```
QSqlCursor cursor("cd");
cursor.select("personid = 125");
if (cursor.next()) {
  QSqlRecord buffer = cursor.primeUpdate();
  buffer.setValue("color", "BLACK");
  buffer.setValue("year", buffer.value("year").toInt() + 1);
  cursor.update();
}
```

Процедура удаления записи похожа на процедуру изменения:

```
QSqlCursor cursor("cd");
cursor.select("personid = 128");
if (cursor.next()) {
  cursor.primeDelete();
  cursor.del();
}
```

См. также: lastError().

© Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

Получить номер последней обработанной записи

Синтаксические правила

```
QVariant QSqlQuery::lastInsertId() const;
```

Описание

Метод предоставляет информацию о номере последней обработанной записи при выполнении операции `insert/update/delete`.

Возвращаемое значение

Возможные значения:

1) указатель на QVariant-объект при успешном обновлении данных. Значение QVariant содержит RowId последней обработанной записи при выполнении операции `insert/update/delete`;
2) указатель на недействительный QVariant-объект в случае ошибки доступа к БД;
3) 0, если реального добавления/изменения/удаления записей не произошло.

Метод возвращает корректное значение RowId только в случае его вызова сразу после выполнения операции `insert/update/delete`. Если после указанных операций был выполнен select-запрос, то возвращаемое значение RowId будет некорректным.

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );
  db.setUserName( "SYSTEM" );
  db.setPassword( "MANAGER8" );

  if( !db.open() )
  {
    cout << db.lastError().driverText().toLocal8Bit().data() <<
 endl;
    return 1;
  }
  QSqlQuery query;
  query.exec("create or replace table Colors (color
 varchar(20));");
  query.exec("insert into Colors (color) values ('Red'),
('Green');");
  if (query.lastError().isValid())
  {
    cout << query.lastError().driverText().toLocal8Bit().data() <<
 endl;
    return 1;
  }
  QVariant sKey = query.lastInsertId();
  if (!sKey.isValid())
  {
```

```
    cout << db.lastError().driverText().toLocal8Bit().data() <<
 endl;
    return 1;
  }
  cout <<"RowId добавленной записи " <<
 sKey.toString().toLocal8Bit().data() <<
 endl;

  query.exec("update Colors set color ='Black' where
 color='Red';");
  if (query.lastError().isValid())
  {
    cout << query.lastError().driverText().toLocal8Bit().data() <<
 endl;
    return 1;
  }
  sKey = query.lastInsertId();
  if (!sKey.isValid())
  {
    cout << db.lastError().driverText().toLocal8Bit().data() <<
 endl;
    return 1;
  }
  cout <<"RowId измененной записи " <<
 sKey.toString().toLocal8Bit().data() <<
 endl;

  query.exec("delete from Colors where color = '12345';");
  if (query.lastError().isValid())
  {
    cout << query.lastError().driverText().toLocal8Bit().data() <<
 endl;
    return 1;
  }
  sKey = query.lastInsertId();
  if (!sKey.isValid())
  {
    cout << db.lastError().driverText().toLocal8Bit().data() <<
 endl;
    return 1;
  }
  cout <<"RowId удаленной записи " <<
 sKey.toString().toLocal8Bit().data() <<
 endl;
```

Результат выполнения примера:

© Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

```
RowId добавленной записи 2
RowId измененной записи 1
RowId удаленной записи 0
```

Работа с BLOB-данными

Для работы с BLOB-данными могут использоваться следующие средства:

1) обычные SQL-операторы манипулирования данными (`select`, `insert`, `update`) с привязкой параметра типа `QByteArray` для получения/добавления BLOB-данных;
2) специальный псевдооператор для работы с BLOB-данными;
3) набор методов (устаревших) для работы с BLOB-данными (см. приложение).

Получение BLOB-данных

Стандартный способ

Стандартный способ получения BLOB-данных предполагает использование SELECT-запроса для загрузки данных в поле типа `QByteArray` с помощью метода `value` класса QSqlQuery, например,

```
value(0).toByteArray()
```

В этом случае в оперативную память загружается всё содержимое BLOB-столбца. Это может оказаться неприемлемо при больших размерах BLOB-данных.

```
q.exec("select * from test;");
cout<<"Blob data:"<<q.value(2).toString()<<endl;
```

При использовании параметризованного запроса стандартный метод предполагает подготовку параметризованного запроса к выполнению, привязку параметров и последующее выполнение запроса, например,

```
…
q.prepare("INSERT INTO test (i, bl) VALUES (?, ?)");
q.bindValue(0, 6);
q.bindValue(1, "af2367aadecd");
q.exec();
```

Получение с помощью псевдозапроса

Синтаксические правила

```
BLOB GET
{ROWID=<rowid записи> [<имя схемы>.]<имя таблицы>.<имя столбца>
|[COLUMN=<номер столбца>]}
[FILE=<спецификация файла>]
```

Все имена в команде могут задаваться в двойных кавычках.

```
blob get rowid=250 "Админ"."Авто"."Модель";
```

Описание

Опция ROWID задает внутренний идентификатор строки в таблице, к которой относится BLOB-команда. При её использовании требуется обязательное указание имени столбца таблицы (не выборки данных !) с указанием имени таблицы и опционально имени схемы. При обработке данной опции СУБД самостоятельно выполняет новый SELECT-запрос, для получения записи с указанным ROWID, в результате чего текущая выборка данных будет испорчена.

```
blob get rowid=250 auto.make;
```

Если задана опция COLUMN, все операции с BLOB-данными применяются к указанному столбцу текущей записи последней выборки данных. Отсчет столбцов начинается с 1. Если опция не задана, по умолчанию используется первый BLOB-столбец выборки.

```
blob get column;
blob get column=3;
```

Опция FILE задает спецификацию файла (в терминах текущей ОС), в который должны быть выгружены BLOB-данные. Возможность доступна только при сборке Qt-драйвера с установленным макросом ENABLE_ACCESS_BLOBFILE (по умолчанию выключен).

Если спецификация файла не задана, BLOB-данные будут извлекаться СУБД ЛИНТЕР в текстовом виде в стандартный выходной поток с помощью функции printf.

Примечание

В случае претрансляции запроса (псевдозапроса) существует возможность модифицировать его поведение путем привязки параметров. Это возможно только в случае операций над выборкой. На запрос с опцией ROWID привязка параметров не действует.

Для команды GET также возможна привязка 4 параметров:

- параметр с номером 0 задает адрес массива типа QByteArray, который будет использован в качестве буфера для извлекаемых BLOB-данных. Размер буфера подразумевается равным размеру массива QByteArray (метод size()). Если параметр не привязан, используется внутренний буфер QByteArray, выделяемый по размеру порции BLOB-значения. Если порция данных больше, чем массив байт, то заполняется весь массив, иначе только часть буфера с начала. Реальный размер возвращаемой порции можно получить с помощью метода size QByteArray после выполнения метода exec() или в параметре с номером 1;
- параметр с номером 1 указывает размер буфера для извлекаемых BLOB-данных. Используется наименьший размер между заданным размером и размером массива QByteArray. Если параметр не привязан, подразумевается размер, равный размеру массива QByteArray;
- параметр с номером 2 указывает смещение требуемой порции BLOB-данных в BLOB-столбце. Если параметр не привязан, подразумевается смещение 0;
- параметр с номером 3 указывает номер BLOB-столбца в выборке данных (отсчет начинается с 1). Если параметр не указан, подразумевается первый BLOB-столбец в выборке данных.

Команда GET возвращает порцию BLOB-данных в привязанных параметрах с номерами 0 и 1 после выполнения метода exec(). Значения параметров могут быть получены с помощью метода boundValue. В параметре с номером 1 возвращается реальная

 © Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

длина полученной порции BLOB-данных, в параметре с номером 0 возвращаются буфер данных, даже если он не был явно привязан.

По умолчанию результаты выборки BLOB-значения будут помещены во вновь созданную переменную типа `QByteArray`, которая может быть получена методом `boundValue` с аргументом 0.

```
q.exec("select * from test;");
q.exec("blob get column=2");
cout<<"Blob data:"<<q.boundValue(0).toString()<<endl;
```

Имеется возможность получить данные в заранее выделенный массив данных типа `QByteArray`. Для этого необходимо выполнить `prepare()` для запроса типа `blob get`, привязать массив типа `QByteArray` к нулевому параметру этого псевдозапроса методом `QSqlQuery.bindValue(int, QVariant)`, и выполнить его методом exec() без аргументов. В этом случае полученные BLOB-значения BLOB-столбца будут сохранены в привязанном массиве `QByteArray`.

```
QByteArray ba = QByteArray(128);
     q.prepare("blob get");
     q.bindValue(0, ba); //QByteArray to return the data
     q.exec();
     cout<<"Blob data in QByteArray:"<<ba<<"  size
 QByteArray="<<ba.size()<<endl
```

Максимальная длина получаемых данных будет равна длине переменной `QByteArray`, которую можно получить путем вызова методов `size()` или `length()`.

Имеется возможность установить явно максимальную длину запрашиваемых BLOB-данных. Для этого ее необходимо привязать в параметре номер 1. В этом случае максимальная длина данных выбирается как минимальная величина между указанной явно длиной и размером буфера `QByteArray`. С помощью этого же параметра может быть получена реальная длина полученных BLOB-данных после выполнения псевдозапроса

```
blob get(QSqlQuery.exec(), QSqlQuery.boundValue(1).toInt())
```

Реальная длина прочитанной порции данных может быть получена также методом `size()` буфера `QByteArray`.

```
q.bindValue(0, ba); //Can be skipped
q.bindValue(1, 64); //max data length
q.exec();
cout<<"Blob data in QByteArray:"<<ba<<"  size
 ="<<q.boundValue(1).toInt()<<endl
```

Такое поведение параметра номер 1 (длины буфера) сохраняется и в случае, если параметр номер 0 (`QByteArray`) был не привязан: привязывание параметра ограничивает максимальную длину получаемых данных, просмотр привязанных данных после исполнения позволяет получить реальную длину полученных данных. Последнее действительно даже в том случае, если параметр номер 1 не был привязан явно, максимальная длина возвращаемых данных соответствует длине BLOB-данных в данном столбце выборки.

```
q.exec("blob get"); // no any value was bound
```

```
cout<<"Blob data length got by
 boundValue:"<<q.boundValue(1).toInt()
```

Имеются еще 2 параметра, которые также являются опциональными. Смещение получаемых BLOB-данных указывается в параметре номер 2: нулевое смещение соответствует началу BLOB-данных и подразумевается по умолчанию. Параметр номер 3 позволяет устанавливать номер столбца с BLOB-данными, так же, как это делает модификатор `COLUMN` псевдозапроса.

Таким образом, команда `blob get` может принимать до 4 параметров, каждый из которых является опциональным. Нулевой и первый параметры являются как входными, так и выходными и могут быть получены, даже если не были привязаны явно.

Получаемые BLOB-данные могут быть сохранены в заданный файл с помощью указания опции `FILE`. При сохранении в файл продолжают действовать ранее выполненные привязки параметров, однако действуют они только на внутренний промежуточный буфер хранения данных, который, в частном случае, может быть задан явно привязкой нулевого параметра (с сохранением в нем данных).

```
q.exec("select * from test");
q.exec("blob get file=/home/user/blobfile.txt");
```

Получение с помощью устаревшего метода

Получение порции BLOB-данных возможно также с помощью устаревшего метода `FetchBlob` (см. приложение).

Данный метод аналогичен запросу типа blob get за исключением:

1) не поддерживаются опции запроса, поэтому запрос всегда состоит из одного слова "FetchBlob". Следовательно, невозможно сохранение BLOB-данных в файл и спецификация BLOB-столбца возможна только с помощью параметра. Также не поддерживается работа с BLOB-данными через `ROWID`;
2) другое функциональное назначение параметров:

   0 – номер столбца;

   1 – смещение порции BLOB-данных;

   2 – адрес выходного буфера;

   3 – размер выходного буфера.

Добавление BLOB-данных

Стандартный способ

Вставка BLOB-значения может осуществляться с помощью обычного `insert` (`update`) SQL-запроса с привязкой к полю BLOB-типа массива `QByteArray`

```
q.prepare( "insert into test(id, b)  values(?, ?)" );
q.bindValue( 0, 1 );
q.bindValue( 1, ba );
```

 © Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

```
q.exec();
```

В этом случае BLOB-столбец вставляется вместе с данными.

Добавление с помощью псевдозапроса

Синтаксические правила

```
BLOB INSERT | APPEND
{ROWID=<rowid записи> [<имя схемы>.]<имя таблицы>.<имя столбца>
|COLUMN=<номер столбца>}
{FILE=<спецификация файла> | <BLOB-значение>}
<BLOB-значение>::=<строка шестнадцатеричных символов>|
 '<символьная строка>'
```

Все имена в команде могут задаваться в двойных кавычках.

```
blob insert rowid=250 "Админ"."Авто"."Модель" '1234567890';
```

Описание

Опция `ROWID` задает внутренний идентификатор строки в таблице, к которой относится BLOB-команда. При её использовании требуется обязательное указание имени столбца таблицы (не выборки данных !) с указанием имени таблицы и опционально имени схемы. При обработке данной опции СУБД самостоятельно выполняет новый SELECT-запрос, для получения записи с указанным `ROWID`, в результате чего текущая выборка данных будет испорчена.

```
blob insert t rowid=250 tst.blb 342f45aaf0;
```

Если задана опция `COLUMN`, все операции с BLOB-данными применяются к указанному столбцу текущей записи последней выборки данных. Отсчет столбцов начинается с 1. Если опция не задана, по умолчанию используется первый BLOB-столбец выборки.

```
blob append ffcc0011;
blob append column 3 'блог пользователя …';
```

Опция `FILE` задает спецификацию файла (в терминах текущей ОС), содержащего добавляемые BLOB-данные. Возможность доступна только при сборке Qt-драйвера с установленным макросом ENABLE_ACCESS_BLOBFILE (по умолчанию выключен).

<Строка шестнадцатеричных символов> – BLOB-данные для вставки/добавления, представленные в виде непрерывной последовательности шестнадцатеричных цифр.

<Символьная строка> – BLOB-данные для вставки/добавления, представленные в виде символьной строки в нужной кодировке.

```
blob append column=3 file=c:\blob\my_foto.png;
```

Команды `INSERT` и `APPEND` функционально идентичны друг другу, отличаются они только тем, что при выполнении команды `INSERT` предварительно выполняется очистка BLOB-значения.

```
q.exec("blob append 'text data'");
q.exec("blob insert A2C3BF12");
```

BLOB-данные могут быть взяты из массива QByteArray. Для этого он должен быть привязан в качестве нулевого параметра.

```
QByteArray ba("abcd", 4);
q.exec("select * from test");
q.prepare("blob insert");
q.bindValue(0, ba);
```

Длина порции вставляемых/добавляемых BLOB-данных принимается равной длине массива QByteArray. Она может быть также задана явно привязкой в параметре номер 1. Тогда в качестве размера порции будет выбрано минимальное значение из размера массива QByteArray и явно указанной длины порции.

В параметрах с номерами 2 и 3 может быть указан тип добавляемого BLOB-значения и столбец в выборке данных, к BLOB-данным которого должна быть добавлена порция данных. Если значения этих параметров не заданы, то по умолчанию тип добавляемых BLOB-данных берется равным 0, а столбец, к которому добавляется порция данных – первый BLOB-столбец выборки данных.

```
q.prepare("blob append");
q.bindValue(0, ba); // Data array
q.bindValue(1, 1); // Data size
q.bindValue(2, 14); // type
q.bindValue(3, 3); // column
```

Если указать опцию ROWID и имя столбца с таблицей, то возможно добавление BLOB-данных без предварительной выборки данных, по имени таблицы, столбца и ROWID. В этом случае выборка данных будет осуществлена в процессе выполнения запроса.

Команда APPEND добавляет порцию BLOB-данных в конец BLOB-значения.

Команда INSERT эквивалентна последовательности команд CLEAR и APPEND.



Примечание

В случае претрансляции запроса (псевдозапроса) существует возможность модифицировать его поведение путем привязки параметров. Это возможно только в случае операций над выборкой. На запрос с опцией ROWID привязка параметров не действует.

Для команд INSERT, APPEND возможна привязка 4-х параметров, каждый из которых опционален:

- параметр с номером 0 задает адрес массива типа QByteArray, содержимое которого будет вставлено (добавлено) в BLOB-поле. Размер данных подразумевается равным размеру QByteArray (метод size()). Если параметр не привязан, используется пустой внутренний буфер QByteArray;
- параметр с номером 1 указывает размер вставляемой (добавляемой) порции BLOB-данных. Для вставки выбирается наименьший размер между заданным и размером массива QByteArray. Если параметр не привязан, используется размер данных, равный размеру массива QByteArray;
- параметр с номером 2 указывает тип вставляемых BLOB-данных. Если параметр не указан, подразумевается 0 (в команде APPEND параметр игнорируется);
- параметр с номером 3 указывает номер BLOB-столбца в выборке данных, в который должны быть вставлены (добавлены) BLOB-данные. Если параметр не указан, подразумевается первый BLOB-столбец в выборке данных.

© Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

Тип добавляемых (вставляемых) BLOB-данных равен 0.

Добавление с помощью устаревшего метода

Добавление порции BLOB-данных возможно также с помощью устаревшего метода `AddBlob`. Поведение его аналогично псевдозапросу blob append без опций. Метод может иметь также до 4 параметров, но их функциональное назначение другое:

0 – номер столбца;

1 – тип добавляемых BLOB-данных;

2 – адрес буфера с добавляемой порцией BLOB-данных;

3 – размер добавляемой порции BLOB-данных.

```
q.prepare("AddBlob");
q.bindValue(0, 3); // column
q.bindValue(1, 14); // type
q.bindValue(2, ba); // Data array
q.bindValue(3, 1); // Data size
```

Удаление BLOB-данных

Стандартный способ

Удалить BLOB-данные можно двумя способами:

1) с помощью удаления всей записи, содержащей удаляемые BLOB-данные, соответствующим SQL-запросом delete (с последующим добавлением копии удаленной записи, но уже без BLOB-данных);
2) с помощью присвоения удаляемым BLOB-данным NULL-значения соответствующим SQL-запросом update.

Удаление с помощью псевдозапроса

Синтаксические правила

```
BLOB CLEAR
[{ROWID=<rowid записи> [<имя схемы>.]<имя таблицы>.<имя столбца>
|COLUMN=<номер столбца>}]
```

Все имена в команде могут задаваться в двойных кавычках.

```
blob clear rowid=250 "Админ"."Авто"."Модель";
```

Описание

Опция `ROWID` задает внутренний идентификатор строки в таблице, к которой относится BLOB-команда. При её использовании требуется обязательное указание имени столбца таблицы (не выборки данных !) с указанием имени таблицы и опционально имени схемы.

Предварительное выполнение запроса для установления текущей записи не требуется.

```
q.exec("blob clear rowid=126 SYSTEM.test.b");
```

Если задана опция COLUMN, все операции с BLOB-данными применяются к указанному столбцу текущей записи последней выборки данных. Отсчет столбцов начинается с 1. Если опция не задана, по умолчанию используется первый BLOB-столбец выборки.

```
q.exec("select * from test;");
q.exec("blob clear column=3");
```

Примечание
В случае претрансляции запроса (псевдозапроса) существует возможность модифицировать его поведение путем привязки параметров. Это возможно только в случае операций над выборкой. На запрос с опцией ROWID привязка параметров не действует.

При использовании команды CLEAR возможна привязка нулевого параметра, в котором указывается номер столбца в выборке данных (отсчет начинается с 1). Без привязки этого параметра по умолчанию подразумевается первый BLOB-столбец выборки данных.

Опции можно также не задавать. В этом случае очищен будет первый BLOB-столбец в текущей записи выборки. Привязкой нулевого параметра можно в этом случае указать номер столбца в выборке последнего выполненного запроса.

```
q.prepare("blob clear");
q.bindValue(0, 2);
q.exec();
```

Удаление с помощью устаревшего метода

Удаление BLOB-данных возможно также с помощью устаревшего метода PurgeBlob (см. приложение). Поведение его аналогично псевдозапросу blob clear без опций.

Примечание
В случае претрансляции запроса (псевдозапроса) существует возможность модифицировать его поведение путем привязки параметров. Это возможно только в случае операций над выборкой. На запрос с опцией ROWID привязка параметров не действует.

В нулевом параметре можно задать номер столбца в текущей выборке данных с удаляемым BLOB-значением. Если параметр не привязан, по умолчанию используется первый столбец.

Получение информации об объектах БД

Получить список таблиц БД

Синтаксические правила

```
QStringList QSqlDatabase::tables(<тип>QSql::TableType type =
 QSql::Tables) const
<тип>:: = тип запрашиваемых таблиц
```

Типы таблиц определены в перечисленном типе данных QSql::TableType (таблица 4).

 © Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

Таблица 4. Типы таблиц БД, о которых можно запрашивать информацию

Тип данных QSql::TableType	Значение	Описание
QSql::Tables	0x01	Все таблицы, владельцем которых является пользователь, от имени которого выполнено соединение с БД
QSql::SystemTables	0x02	Системные таблицы
QSql::Views	0x04	Все таблицы и представления, владельцем которых является пользователь, от имени которого выполнено соединение с БД
QSql::AllTables	0xff	Все системные таблицы БД, а также таблицы и представления, владельцем которых является пользователь, от имени которого выполнено соединение с БД

Описание

Предоставляет список пользовательских или системных таблиц (представлений) БД.

Возвращаемое значение

Список таблиц в виде QStringList-объекта.

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER" );

if( !db.open("SYSTEM", "MANAGER8") )
{
  cout << db.lastError().driverText().toLocal8Bit().data() <<
 endl;
  return 1;
}

QStringList list = db.tables(QSql::Views);
cout << "Views list\n " << list.join(",").toLocal8Bit().data() <<
 "\n";
```

См. также: primaryIndex(), record().

Получить описание первичного ключа таблицы

Синтаксические правила

```
QSqlIndex QSqlDatabase::primaryIndex([<имя схемы>.]<таблица>)
 const
<таблица>::= символьный литерал или символьная переменная в виде
[<имя схемы>.]<имя таблицы>
```

Описание

Предоставляет информацию о первичном ключе указанной таблицы.

Возвращаемое значение

Указатель на QSqlIndex-объект. Если первичный ключ не создан, возвращается пустой QSqlIndex-объект.

Пример

```
// Получить список столбцов, входящих в первичный ключ
  QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER");

  db.setUserName( "SYSTEM" );
  db.setPassword( "MANAGER8" );

  if( !db.open() )
  {
    cout << db.lastError().driverText().toLocal8Bit().data() <<
 endl;
    return 1;
  }

  QSqlIndex ind = db.primaryIndex("PERSON");
  cout << "Primary index of table person " <<
 ind.name().toLocal8Bit().data()<< "\n";
```

См. также: tables(), record().

Получить описание строки таблицы

Синтаксические правила

```
QSqlRecord QSqlDatabase::record(<таблица>) const
<таблица>::= символьный литерал или символьная переменная в виде
[<имя схемы>.]<имя таблицы>
```

Описание

Предоставляет описание строки таблицы или представления. Порядок расположения полей таблицы – произвольный.

Возвращаемое значение

Указатель на QSqlRecord-объект. Если аргумент `<таблица>` задает несуществующую в БД таблицу (представление), возвращается пустой QSqlRecord-объект (`isEmpty` будет `true`).

Пример

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QLINTER");

  db.setUserName( "SYSTEM" );
  db.setPassword( "MANAGER8" );
```

 © Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

```
  if( !db.open() )
  {
    cout << db.lastError().driverText().toLocal8Bit().data() <<
endl;
    return 1;
  }

  QSqlRecord rec = db.record("PERSON");
  int i;
  cout << "count " << rec.count() << "\n";
  for (i = 0; i < rec.count(); i++ )
  {
    cout << "record " << rec.fieldName(i).toLocal8Bit().data()<<
"\n";
  }
```

См. также: tables().

Обработка кодов завершения

Получить последний код завершения

Синтаксические правила

```
QSqlError QSqlDatabase::lastError() const
```

Описание

Предоставляет информацию о коде завершения последней выполненной по соединению операции.

Возвращаемое значение

Указатель на QSqlError-объект. Возможные значения кода завершения определены в перечислимом типе данных `QSqlError::ErrorType` (таблица 5).

Таблица 5. Коды завершения интерфейса

Код завершения QSqlError::ErrorType	Значение	Описание
`QSqlError::NoError`	0x00	Нормальное завершение операции
`QSqlError::ConnectionError`	0x01	Ошибка при соединении с БД
`QSqlError::StatementError`	0x02	Синтаксическая ошибка SQL-оператора
`QSqlError::TransactionError`	0x03	Ошибка при обработке транзакции
`QSqlError::UnknownError`	0x04	Неизвестная ошибка

Пример

См. раздел Получить описание строки таблицы.

Приложение
Устаревшие функции для работы с BLOB-данными

Примечание
Приведенные ниже функции для работы с BLOB-данными являются устаревшими и не рекомендуются для использования в новых клиентских приложениях.

QtLinter-драйвер поддерживает два способа работы с BLOB-данными:

1) с помощью стандартных средств QSql-интерфейса;
2) с помощью специальных функций QtLinter-драйвера.

В первом способе работа с BLOB-данными выполняется так же, как и с данными любого другого типа:

- выборка – с помощью SELECT-запроса;
- извлечение – с помощью функции `QSqlQuery::data()`;
- добавление – с помощью претранслированного INSERT-запроса и привязкой в качестве параметра значения типа QVariant::ByteArray.

Единственный недостаток первого способа – необходимость достаточного количества оперативной памяти для размещения BLOB-значения целиком.

Во втором способе для работы с BLOB-данными используются специальные функции, встроенные в QtLinter-драйвер. Последовательность работы в этом случае должна быть следующей:

- создание экземпляра класса QSqlQuery;
- выполнение SELECT-запроса;
- переход с помощью функции `fetch()` на строку выборки, содержащую нужные BLOB-данные;
- с помощью этого же экземпляра класса выполнение функции `prepare()` для BLOB-функции QtLinter-драйвера;
- привязка необходимых параметров (с помощью функции `bind()`).

Оперативная память при этом расходуется более экономно.

Получить порцию BLOB-данных

Синтаксические правила

```
q.prepare("FetchBlob")  | q.prepare("FetchBlob(?,?,?.?)");
q.bindValue( 0, 2 );
q.bindValue( 1, 3 );
q.bindValue( 2, ba );
q.bindValue( 3, 6 );
```

Описание

Функция предоставляет заданную порцию BLOB-значения.

© Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

После выполнения `q.prepare("FetchBlob")` необходимо привязать к этому запросу 4 параметра:

- 0-й параметр – порядковый номер BLOB-столбца в строке таблицы (отсчет начинается с 0);
- 1-й параметр – смещение (в байтах) запрашиваемой порции BLOB-данных (отсчет начинается с 0);
- 2-й параметр – адрес буфера для приема порции BLOB-данных (переменная типа QVariant::ByteArray). Размер буфера должен соответствовать размеру запрашиваемой порции данных;
- 3-й параметр – размер считываемой порции BLOB-данных (в байтах).

Возвращаемое значение

`QSqlQuery::prepare()` и exec() возвращают `true` в случае нормального завершения, и `false` при ошибках.

Чтобы узнать реальную длину полученной порции BLOB-данных, необходимо проверить длину буфера. Она будет меньше или равна заданному размеру буфера.

Пример

```
q.exec(Qstring("select blobcolumn from test;"));
q.first();
q.prepare( "FetchBlob" );
ba.resize( 6 );
q.bindValue( 0, 1 );  //Номер BLOB-столбца (1)
q.bindValue( 1, 1 );  //Смещение порции BLOB-данных (1)
q.bindValue( 2, ba ); //Адрес буфера для приема BLOB-данных
q.bindValue( 3, ba.size() ); //Размер порции BLOB-данных
if ( !q.exec() )
{
   cout << q.lastError().driverText();
   return 1;
 }
```

Добавить порцию BLOB-данных

Синтаксические правила

```
q.prepare("AddBlob")  | q.prepare("AddBlob(?,?,?.?)");
q.bindValue( 0, 2 );
q.bindValue( 1, 3 );
q.bindValue( 2, ba );
q.bindValue( 3, 6 );
```

Описание

Функция добавляет порцию BLOB-данных в конец существующего BLOB-значения.

После выполнения `q.prepare("AddBlob")` необходимо привязать к этому запросу 4 параметра:

- 0-й параметр – порядковый номер BLOB-столбца в строке таблицы (отсчет начинается с 0);
- 1-й параметр – тип добавляемых BLOB-данных (произвольное число, выбирается и используется пользователем);
- 2-й параметр – адрес буфера, в котором содержится добавляемая порция BLOB-данных (переменная типа QVariant::ByteArray);
- 3-й параметр – размер добавляемой порции BLOB-данных (в байтах).

Возвращаемое значение

`QSqlQuery::prepare()` возвращают `true` в случае нормального завершения, и `false` при ошибках.

Пример

```
q.exec(Qstring("select blobcolumn from test;"));
q.first();
q.prepare( "AddBlob" );
// Заполнения буфера ba BLOB-данными

ba = QByteArray::duplicate("TESTSTRING", strlen("TESTSTRING"));

q.bindValue( 0, 1 );  //Номер BLOB-столбца (1)
q.bindValue( 2, ba ); //Адрес буфера с добавляемыми BLOB-данными
q.bindValue( 3, ba.size() ); //Размер порции добавляемых BLOB-
данных
if ( !q.exec() )
{
   cout << q.lastError().driverText();
   return 1;
 }
```

Удалить BLOB-данные

Синтаксические правила

```
q.prepare("PurgeBlob")  | q.prepare("PurgeBlob(?)");
q.bindValue( 0, 2 );
```

Описание

Функция удаляет все BLOB-данные в указанном столбце таблицы.

После выполнения `q.prepare("PurgeBlob")` необходимо привязать к этому запросу 0-й параметр: порядковый номер BLOB-столбца в строке таблицы (отсчет начинается с 0).

Возвращаемое значение

`QSqlQuery::prepare()` возвращают `true` в случае нормального завершения, и `false` при ошибках.

 © Qt-интерфейс. АО НПП «РЕЛЭКС», 1990-2026

Пример

```
q.exec(Qstring("select blobcolumn from test;"));
q.first();
q.prepare( "PurgeBlob(?)" );
q.bindValue( 0, 1 );  //Номер BLOB-столбца (1)
if ( !q.exec() )
{
   cout << q.lastError().driverText();
   return 1;
 }
```