Qt и SQLite и вообще, программирование БД в Qt tutorial

SQLite*, Qt*

Добрый день.

Ниже пойдет речь о том, как использовать SQLite в Qt. Автор постарался как можно подробнее рассматривать программирование баз данных в Qt.

Об этих двух замечательных продуктах можно прочитать следуя приведенным выше ссылкам, а мы будем конкретно рассматривать программирование БД в Qt, в частности, на примере SQLite. Скажу только, что SQLite несколько отличается от «обычных» баз данных, таких как MySQL тем, что «не обладает» клиент-серверной архитектурой. То есть движок БД не является отдельно работающим процессом, с которым взаимодействует программа. SQLite представляет собой библиотеку, с которой компонуется ваша программа и, таким образом, движок становится составной частью программы. То есть представьте вы решили сохранять все данные, с которыми «сталкивается» ваша программа в обычный файл. В один прекрасный день вы решаете сохранять данные в файле, но организовав это с «реляционной» точки зрения. После этого вы поняли, что новая структура файла должна «распознаваться особым образом». С этого, как минимум, следует, что вам нужно предоставить некоторый API, обеспечивающий связь между этим файлом данных с приложением. В общем, следуя логической постановке приведенного сценария у вас рождается система БД, не требующая сервера БД и собственно, клиента. Получается достаточно быстрая по сравнению с «клиент-серверной» БД система, и сама программа упрощается.

Я состою в дружеских отношениях с Qt и недавно мне понадобилось ее БД функциональность. С MySQL я тоже в достаточно дружеских отношениях и попытался использовать Qt с MySQL в программе, в то время разрабатываемой мной. Имея нехватку времени и нервов, чтоб «связать» MySQL с Qt, решил воспользоваться SQLite, для чего в Qt есть, так сказать, встроенная поддержка, то есть ничего нового установить/конфигурировать не надо было (это не относиться к случаю, если ваш Qt собран с поддержкой «считанных» модулей, без подключения модуля QtSql). И еще, если мне придется установить программу в другой компьютер, я не буду «вынужден» установить сервер MySQL и т.д. (спорная тема — знаю).

FIY

На данный момент я использую программу SQLiteManager для создания БД, таблиц и т.д., использую недавно, но программа сразу понравилась. В моей «рабочей лошадке» установлен(а?) Qt Windows SDK и я использую QtCreator, сразу скажу — просто блеск (*не ИМХО, и вправду отличная IDE*).

И так, Qt и базы данных

Как уже неявно упомянулось выше, в Qt есть отдельный модуль, предоставляющий удобный «сервис» использования БД — **QtSql**. Если у вас есть опыт работы с Qt, то о файле .pro вам известно, если нет — познакомьтесь. Помните только, что нужно добавить следующую строку в .pro файл:

```
QT += sql
```

Это, чтоб использовать модуль QtSql, а для работы с ее классами, нужно включать одноименный заголовок.

```
#include <QtSql>
```

В книгах по Qt говорится о трех уровнях модуля QtSql:

1. Уровень драйверов
2. Программный уровень
3. Уровень пользовательского интерфейса

Уровень драйверов

К уровню драйверов относятся классы для получения данных на физическом уровне, такие, как:

- QSqlDriver
- QSqlDriverCreator<T*>
- QSqlDriverCreatorBase,
- QSqlDriverPlugin
- QSqlResult

QSqlDriver является абстрактным базовым классом, предназначенный для доступа к специфичным БД. Важно, что класс не должен быть использован «прямо», взамен нужно/можно воспользоваться *QSqlDatabase*. Хотя, если вы хотите создать свой собственный драйвер SQL, то можете наследовать от *QSqlDriver* и реализовать чисто виртуальные, и нужные вам виртуальные функции.
QSqlDriverCreator — шаблонный класс, предоставляющий фабрику SQL драйвера для специфичного типа драйвера. Шаблонный параметр должен быть подклассом *QSqlDriver*.
QSqlCreatorBase — базовый класс для фабрик SQL драйверов, чтобы возвращать экземпляр специфичного поскласса класса *QSqlDriver*, который вы хотите предоставить, нужно «перефразировать» метод *createObject()*.
QSqlDatabase несет ответственность за загрузку и управление плагинов драйверов баз данных. Когда база данных добавлена (это делается функцией *QSqlDatabase::addDatabase()*), необходимый плагин драйвера загружается (используя *QSqlDriverPlugin*). QSqlDriverPlugin предоставляет собой абстрактный базовый класс для пользовательских *QSqlDriver* плагинов.
QSqlResult сам говорит о себе (как и все Qt-шные классы), этот класс предоставляет абстрактный интерфейс для доступа к данным специфичных БД. С практической точки зрения мы будем использовать *QSqlQuery* вместо *QSqlResult*, поскольку *QSqlQuery* предоставляет обертку («обобщенную») для БД-специфичных реализации *QSqlResult*.
Так, поскольку уровень драйверов, как оказалось, актуально использовать при создании собственного драйвера, то привож

данным специфичных БД. С практической точки зрения мы будем использовать *QSqlQuery* вместо *QSqlResult*, поскольку *QSqlQuery* предоставляет обертку («обобщенную») для БД-специфичных реализации *QSqlResult*.
Так, поскольку уровень драйверов, как оказалось, актуально использовать при создании собственного драйвера, то привожу пример кода (для наиболее заинтересованных), который может быть использован как каркасс для драйвера:

```
class XyzResult : public QSqlResult
 {
 public:
     XyzResult(const QSqlDriver *driver)
         : QSqlResult(driver) {}
     ~XyzResult() {}

 protected:
     QVariant data(int /* index */) { return QVariant(); }
     bool isNull(int /* index */) { return false; }
     bool reset(const QString & /* query */) { return false; }
     bool fetch(int /* index */) { return false; }
     bool fetchFirst() { return false; }
     bool fetchLast() { return false; }
     int size() { return 0; }
     int numRowsAffected() { return 0; }
     QSqlRecord record() const { return QSqlRecord(); }
 };

 class XyzDriver : public QSqlDriver
 {
 public:
     XyzDriver() {}
     ~XyzDriver() {}

     bool hasFeature(DriverFeature /* feature */) const { return false; }
     bool open(const QString & /* db */, const QString & /* user */,
               const QString & /* password */, const QString & /* host */,
               int /* port */, const QString & /* options */)
         { return false; }
     void close() {}
     QSqlResult *createResult() const { return new XyzResult(this); }
 };
```

Программный уровень

Для соединения с базой данных прежде всего нужно активизировать драйвер используя статический метод *QSqlDatabase::addDatabase()*. Метод получает строку как аргумент, обозначающий идентификатор драйвер СУБД. Нам понадобится «QSQLITE».

```
QSqlDatabase sdb = QSqlDatabase::addDatabase("QSQLITE");
sdb.setDatabaseName("db_name.sqlite");

if (!sdb.open()) {
       //....
}
```

У статической функции addDatabase есть перегруженный «брат», получающий не имя драйвера, а сам драйвер(*QSqlDriver**). Соединение осуществляется методом open(). Класс *QSqlDatabase* представляет соединение с БД. Соединение предоставляет доступ к БД через поддерживаемый драйвер БД. Важно, что можно иметь несколько соединений к одной БД.
Если при соединении (метод *open()*) возникла ошибка, то получить информацию об ошибке можно через метод *QSqlDatabase::lastError()* (возвращает *QSqlError*).

```
if (!sdb.open()) {
      qDebug() << sdb.lastError().text();
}
```

Рассмотрим как Qt позволяет исполнять команды SQL. Для этого можно воспользоваться классом *QSqlQuery*. Класс может быть использована не только для исполнения DML (Data Manipulation Language) выражений, таких, как **SELECT**, **INSERT**, **UPDATE** и **DELETE**, но и DDL (Data Definition Language) выражений, таких, как **CREATE TABLE**. Обратите внимание, что может быть выполнена и БД-специфичная команда, не являющийся стандартом SQL (например, для PSQL — «SET DATESTYLE=ISO»).
Удачно выполненные запросы устанавливают состояние запроса в «активный», так, *isActive()* возвратит *true*, в противоположном случае состояние устанавливается в неактивное. Запросы оформляются в виде обычной строки, которая передается в конструктор или в метод *QSqlQuery::exec()*. В первом случае, при передаче конструктору, запуск команды будет производиться автоматически (при конструировании объекта).
Что очень интересно, так это возможность навигации, предоставляемый *QSqlQuery*. Например, после запроса *SELECT* можно

противоположном случае состояние устанавливается в неактивное. Запросы оформляются в виде обычной строки, которая передается в конструктор или в метод *QSqlQuery::exec()*. В первом случае, при передаче конструктору, запуск команды будет производиться автоматически (при конструировании объекта).
Что очень интересно, так это возможность навигации, предоставляемый *QSqlQuery*. Например, после запроса *SELECT* можно перемещаться по собранным данным при помощи методов *next(), previous(), first(), last()* и *seek()*.

```
QSqlQuery query("SELECT country FROM artist");
while (query.next()) {
        QString country = query.value(0).toString();
        do_something(country);
}
```

Метод *next()* позволяет перемащатся на следующую строку данных, а вызов *previous()* на предыдущую строку, соответственно. *first(), last()* извлекают, соответственно, первую запись из результата. *seek()* получает целочисленный индекс, извлекая запись из результата по полученному индексу и «позиционирует запрос» на извлеченную запись. Проверить размер, вернее количество строк данных (результата) можно методом *size()*. Важно помнить, что первая запись находится в позиции 0, запрос должен быть в активном состоянии, а *isSelect*() возвращать *true* (это происходит, если последним запросом был *SELECT*) перед вызовом метода seek(). О методе *seek*() более подробно советую прочитать в официальной документации.
Выше упомянули, что если передавать строку запроса в конструктор класса *QSqlQuery*, то запрос выполнится при создании объекта — при конструировании. Используя метод *exec*() можно, так сказать, следить за временем выполнения запросов.
Конструкция

```
QSqlQuery query("SELECT country FROM artist");
```

может быть представлена также так:

```
QSqlQuery query;
query.exec("SELECT country FROM artist");
```

Так, *exec*() получает запрос в виде *QString*. Выполняя запрос, в случае удачи этот метод возвращает true и устанавливает состояние в активное, в противоположном случае все «противоположное» указанным операциям. Конечно, следует еще и помнить, что строка запроса должна подчиняться синтаксическим правилам запрашиваемой БД (в частности, стандарту SQL).
Что интересно, так после исполнения, запрос позиционируется на инвалидный(ую?) запись, то есть для адекватного использования результатов, необходимо воспользоваться, скажем, методом *next*().
У метода *exec*() перегруженная альтернатива, не получающая никаких аргументов. Вызов этого варианта *exec*() исполняет до этого подготовленный запрос. Обратите внимание — «подготовленный». Для этого предназначен метод *prepare*(), который возвращает *true* в случае удачной подготовки запроса.
Важность или, можно сказать, уникальность метода в том, что запрос может содержать «заполнители» для связывания со значениями используая *bindValue*().

```
QSqlQuery my_query;
my_query.prepare("INSERT INTO my_table (number, address, age)"
                         "VALUES (:number, :address, :age);");
my_query.bindValue(":number", "14");
my_query.bindValue(":address", "hello world str.");
my_query.bindValue(":age", "37");
```

Еще можно использовать вариант безымянных параметров:

```
QSqlQuery my_query;
my_query.prepare("INSERT INTO my_table (number, address, age)"
                         "VALUES (?, ?, ?);");
my_query.bindValue("14");
my_query.bindValue("hello world str.");
my_query.bindValue("37");
```

И наконец, можно просто использовать подставляемые аргументы, которые предоставляет QString:

```
QSqlQuery my_query;
my_query.prepare(
          QString("INSERT INTO my_table (number, address, age) VALUES (%1, '%2', %3);")
                         .arg("14").arg("hello world str.").arg("37")
          );
```

Компилируемый (copy-paste-to-your-ide) пример:

```
#include <QtGui/QApplication>
#include <QtSql>

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    QSqlDatabase dbase = QSqlDatabase::addDatabase("QSQLITE");
    dbase.setDatabaseName("my_db.sqlite");
    if (!dbase.open()) {
        qDebug() << "Что-то пошло не так!";
```

```
    dbase.setDatabaseName("my_db.sqlite");
    if (!dbase.open()) {
        qDebug() << "Что-то пошло не так!";
        return -1;
    }

    QSqlQuery a_query;
    // DDL query
    QString str = "CREATE TABLE my_table ("
            "number integer PRIMARY KEY NOT NULL, "
            "address VARCHAR(255), "
            "age integer"
            ");";
    bool b = a_query.exec(str);
    if (!b) {
        qDebug() << "Вроде не удается создать таблицу, провертье карманы!";
    }

    // DML
    QString str_insert = "INSERT INTO my_table(number, address, age) "
            "VALUES (%1, '%2', %3);";
    str = str_insert.arg("14")
            .arg("hello world str.")
            .arg("37");
    b = a_query.exec(str);
    if (!b) {
        qDebug() << "Кажется данные не вставляются, проверьте дверь, может она закрыта?";
    }
    //.....
    if (!a_query.exec("SELECT * FROM my_table")) {
        qDebug() << "Даже селект не получается, я пас.";
        return -2;
    }
    QSqlRecord rec = a_query.record();
    int number = 0,
            age = 0;
    QString address = "";

    while (a_query.next()) {
        number = a_query.value(rec.indexOf("number")).toInt();
        age = a_query.value(rec.indexOf("age")).toInt();
        address = a_query.value(rec.indexOf("address")).toString();

        qDebug() << "number is " << number
                 << ". age is " << age
                 << ". address" << address;
    }

    return app.exec();
}
```

Для получения результата запроса следует вызвать метод *QSqlQuery::value()*, в котором необходимо передать номер столбца, для чего в примере воспользовались методом *record*(). Этот метод возвращает объект класса *QSqlRecord*, который содержит информацию, относящуюся к запросу *SELECT*. С помощью вызова *QSqlRecord::indexOf()* получаем индекс столбца.
Метод *value()* возвращает значения типа *QVariant* (класс, объекты которого могут содержать в себе значения разных типов), поэтому вы и преобразовали полученное значение, воспользовавшись методами *QVariant::toInt()* и *QVariant::toString()*.

Уровень пользовательского интерфейса

Модуль *QtSql* поддерживает концепцию «Интервью», предоставляя ряд моделей для использования их в представлениях. Чтобы хорошенько познакомиться с этой концепцией — загляните сюда.
В качестве примера, класс *QSqlTableModel* позволяет отображать данные в табличной и иерархической форме. Как утверждается в литературе, интервью — самый простой способ отобразить данные таблицы, здесь не потребуется цикла для прохождения по строкам таблицы. Вот малюсенький пример:

```
#include <QtGui/QApplication>
#include <QtSql>
#include <QTableView>

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
```

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    QSqlDatabase dbase = QSqlDatabase::addDatabase("QSQLITE");
    dbase.setDatabaseName("my_db.sqlite");
    if (!dbase.open()) {
        qDebug() << "Что-то не так с соединением!";
        return -1;
    }

    QTableView view;
    QSqlTableModel model;

    model.setTable("my_table");
    model.select();
    model.setEditStrategy(QSqlTableModel::OnFieldChange);

    view.setModel(&model);
    view.show();

    return app.exec();
}
```

После соединения создается объект табличного представления *QTableView* и объект табличной модели *QSqlTableModel*.
Методом *setTable*() устанавливается актуальная база в модели, а вызов *select*() производит заполнение данными.
Класс *QSqlTableModel* предоставляет следующие стратегии редактирования (устанавливаемые с помощью *setEditStrategy*()):

- **OnRowChange** — производит запись данных, как только пользователь перейдет к другой строке таблицы.
- **OnFieldChange** — производит запись после того, как пользователь перейдет к другой ячейке таблицы.
- **OnManualSubmit** — записывает данные по вызову слота *submitAll*(). Если вызывается слот *revertAll*(), то данные возвращаются в исходное состояние.