Работа с базами данных в Qt

Qt*

Qt дает возможность создания платформо-независимых приложений для работы с базами данных, используя стандартные СУБД. Qt включает «родные» драйвера для Oracle, Microsoft SQL Server, Sybase Adaptive Server, IBM DB2, PostgreSQL, MySQL и ODBC-совместимых баз данных. Qt включает специфичные для баз данных виджеты, а также поддерживает расширение для работы с базами данных любых встроенных или отдельно написанных виджетов.

Введение

Работа с базами данных в Qt происходит на различных уровнях:
1.Слой драйверов — Включает классы QSqlDriver, QSqlDriverCreator, QSqlDriverCreatorBase, QSqlDriverPlugin и QSqlResult. Этот слой предоставляет низкоуровневый мост между определенными базами данных и слоем SQL API.
2.Слой SQL API — Этот слой предоставляет доступ к базам данных. Соединения устанавливаются с помощью класса QSqlDatabase. Взаимодействие с базой данных осуществляется с помощью класса QSqlQuery. В дополнение к классам QSqlDatabase и QSqlQuery слой SQL API опирается на классы QSqlError, QSqlField, QSqlIndex и QsqlRecord.
3.Слой пользовательского интерфейса — Этот слой связывает данные из базы данных с дата-ориентированными виджетами. Сюда входят такие классы, как QSqlQueryModel, QSqlTableModel и QSqlRelationalTableModel.

Соединение с базой данных

Чтобы получить доступ к базе данных с помощью QSqlQuery и QSqlQueryModel, необходимо создать и открыть одно или более соединений с базой данных.
Qt может работать со следующими базами данных (из-за несовместимости с GPL лицензией, не все плагины поставляются с Qt Open Source Edition):

1. QDB2 — IBM DB2 (версия 7.1 и выше
2. QIBASE — Borland InterBase
3. QMYSQL — MySQL
4. QOCI — Драйвер Oracle Call Interface
5. QODBC — Open Database Connectivity (ODBC) — Microsoft SQL Server и другие ODBC-совместимые базы данных
6. QPSQL — PostgreSQL (версия 7.3 и выше)
7. QSQLITE2 — SQLite версии 2
8. QSQLITE — SQLite версии 3
9. QTDS — Драйвер Sybase Adaptive Server

Для сборки плагина драйвера, которые не входят в поставку Qt нужно иметь соответствующую клиентскую библиотеку для используемой СУБД.

Соединиться с базой данных можно вот так:

```
1.    QSqlDatabase db = QsqlDatabase::addDatabase("QMYSQL", "mydb");
2.    db.setHostName("bigblue");
3.    db.setDatabaseName("flightdb");
4.    db.setUserName("acarlson");
5.    db.setPassword("luTbSbAs");
6.    bool ok = db.open();
* This source code was highlighted with Source Code Highlighter.
```

Первая строка создает объект соединения, а последняя открывает его. В промежутке инициализируется некоторая информация о соединении, включая имя соединения, имя базы данных, имя узла, имя пользователя, пароль. В этом примере происходит соединение с базой данных MySQL flightdb на узле bigblue. Аргумент «QMYSQL» в addDatabase() указывает тип драйвера базы данных, чтобы использовать для соединения, а «mydb» — имя соединения.
Как только соединение установлено, можно вызвать статическую функцию QSqlDatabase::database() из любого места программы с указанием имени соединения, чтобы получить указатель на это соединение. Если не передать имя соединения, она вернет соединение по умолчанию.
Если open() потерпит неудачу, он вернет false. В этом случае, можно получить информацию об ошибке, вызвав QSqlDatabase::lastError().
Для удаления соединения с базой данных, надо сначала закрыть базу данных с помощью QSqlDatabase::close(), а затем, удалить соединение с помощью статического метода QSqlDatabase::removeDatabase().

Выполнение инструкций SQL

Класс QSqlQuery обеспечивает интерфейс для выполнения SQL запросов и навигации по результирующей выборке.
Для выполнения SQL запросов, просто создают объект QSqlQuery и вызывают QSqlQuery::exec(). Например, вот так:

```
1.    QSqlQuery query;
2.    query.exec("SELECT name, salary FROM employee WHERE salary > 50000");
* This source code was highlighted with Source Code Highlighter.
```

Конструктор QSqlQuery принимает необязательный аргумент QSqlDatabase, который уточняет, какое соединение с базой данных используется. Если его не указать, то используется соединение по умолчанию.

Конструктор QSqlQuery принимает необязательный аргумент QSqlDatabase, который уточняет, какое соединение с базой данных используется. Если его не указать, то используется соединение по умолчанию.
Если возникает ошибка, exec() возвращает false. Доступ к ошибке можно получить с помощью QSqlQuery::lastError().
QSqlQuery предоставляет единовременный доступ к результирующей выборке одного запроса. После вызова exec(), внутренний указатель QSqlQuery указывает на позицию перед первой записью. Если вызвать метод QSqlQuery::next() один раз, то он переместит указатель к первой записи. После этого необходимо повторять вызов next(), чтобы получать доступ к другим записям, до тех пор пока он не вернет false. Вот типичный цикл, перебирающий все записи по порядку:

```
1.    while (query.next()) {
2.      QString name = query.value(0).toString();
3.      int salary = query.value(1).toInt();
4.      qDebug() << name << salary;
5.    }
* This source code was highlighted with Source Code Highlighter.
```

QSqlQuery может выполнять не только SELECT, но также и любые другие запросы. Следующий пример вставляет запись в таблицу, используя INSERT:

```
1.    QSqlQuery query;
2.    query.exec("INSERT INTO employee (id, name, salary) "
3.         "VALUES (1001, 'Thad Beaumont', 65000)");
* This source code was highlighted with Source Code Highlighter.
```

Если надо одновременно вставить множество записей, то зачастую эффективней отделить запрос от реально вставляемых значений. Это можно сделать с помощью вставки значений через параметры. Qt поддерживает два синтаксиса вставки значений: поименованые параметры и позиционные параметры. В следующем примере показана вставка с помощью поименованного параметра:

```
1.    QSqlQuery query;
2.    query.prepare("INSERT INTO employee (id, name, salary) "
3.           "VALUES (:id, :name, :salary)");
4.    query.bindValue(":id", 1001);
5.    query.bindValue(":name", "Thad Beaumont");
6.    query.bindValue(":salary", 65000);
7.    query.exec();
* This source code was highlighted with Source Code Highlighter.
```

В этом примере показана вставка с помощью позиционного параметра:

```
1.    QSqlQuery query;
2.    query.prepare("INSERT INTO employee (id, name, salary) "
3.           "VALUES (?, ?, ?)");
4.    query.addBindValue(1001);
5.    query.addBindValue("Thad Beaumont");
6.    query.addBindValue(65000);
7.    query.exec();
* This source code was highlighted with Source Code Highlighter.
```

При вставке множества записей требуется вызвать QSqlQuery::prepare() только однажды. Далее можно вызвать bindValue() или addBindValue() с последующим вызовом exec() столько раз, сколько потребуется.

Отображение данных в таблице-представлении

Классы QSqlQueryModel, QSqlTableModel и QSqlRelationalTableModel могут использоваться в качестве источников данных для классов представлений Qt, таких как QListView, QTableView и QTreeView. На практике наиболее часто используется QTableView в связи с тем, что результирующая SQL выборка, по существу, представляет собой двумерную структуру данных.
В следующем примере создается представление основанное на модели данных SQL:

```
1.    QSqlTableModel model;
2.    model.setTable("employee");
3.    QTableView *view = new QTableView;
4.    view->setModel(&model);
5.    view->show();
* This source code was highlighted with Source Code Highlighter.
```

Если модель является моделью для чтения-записи (например, QSqlTableModel), то представление позволяет редактировать поля. Это можно отключить с помощью следующего кода

```
1.    view->setEditTriggers(QAbstractItemView::NoEditTriggers);
* This source code was highlighted with Source Code Highlighter.
```

```
1.    view->setEditTriggers(QAbstractItemView::NoEditTriggers);
* This source code was highlighted with Source Code Highlighter.
```

Можно использовать одну и ту-же модель в качестве источника данных для нескольких представлений. Если пользователь изменяет данные модели с помощью одного из представлений, другие представления немедленно отобразят изменения. Классы-представления для обозначения колонок наверху отображают заголовки. Для изменения текста заголовка, используется функция setHeaderData() модели. Например:

```
1.    model->setHeaderData(0, Qt::Horizontal, QObject::tr("ID"));
2.    model->setHeaderData(1, Qt::Horizontal, QObject::tr("Name"));
3.    model->setHeaderData(2, Qt::Horizontal, QObject::tr("City"));
4.    model->setHeaderData(3, Qt::Horizontal, QObject::tr("Country"));
* This source code was highlighted with Source Code Highlighter.
```

Заключение

В данной статье изложены базовые принципы работы с базами данных в Qt. Однако кроме указанных здесь возможностей еще много интересного, например, транзакции, работа с внешними ключами или создание дата-ориентированых форм. К сожалению эти темы достаточно обширны для одной статьи.