

Технология ООП

Санкт-Петербургский государственный политехнический университет

29 ноября 2011

Qt обеспечивает интерфейс для общения с базами данных независимый от платформы и конкретной реализации базы данных.

Для использования баз данных, необходимо подключить модуль QtSql.

Добавить в проектный файл: QT += sql

Для использования классов этого модуля:

```
#include <QtSql>
```

Классы модуля QSql разделены на 3 уровня:

- уровень драйверов
- программный уровень
- уровень пользовательского интерфейса

К уровню драйверов относятся классы для получения данных на физическом уровне: QSqlDriver, QSqlDriverPlugin, QSqlResult.

Связь с базой данных обеспечивается объектом QSqlDatabase. Qt включает в себя следующие драйверы:

- QDB2 — DB2
- QIBASE — Interbase
- QMYSQL — MySQL
- QOCI — Oracle
- QODBC — ODBC драйвер для MS SQL Server, IBM DB2 и т.д.
- QPSQL — PostgreSQL
- QSQLITE — PostgreSQL
- QTDS — Sybase

Для выполнения запросов SQL необходимо сначала установить соединение с базой данных.

Обычно настройка выполняется отдельной функцией вызываемой при запуске приложения.

```
bool createConnection()  
{  
    QSqlDatabase *db = QSqlDatabase::addDatabase(""  
        QSQLITE"");  
    db->setDatabaseName("base.dat");  
    if (!db->open()) { qDebug("Error_open_database");  
        return false; }  
}
```

При использовании серверов баз данных, необходимо также задать `setHostName`, `setUserName`, `setPassword`.

Задав имя соединения, при подключении к базе данных, можно вернуть на него потом указатель.

```
QSqlDatabase *db = QSqlDatabase::addDatabase("
    QSQLITE", "MYDATABASE");

...
QSqlDatabase db1 = QSqlDatabase::database("
    MYDATABASE");
```

Программный интерфейс для обращения к базе данных:
QSqlDatabase, QSqlQuery, QSqlIndex, QSqlRecord.

Программный уровень

После установки соединения можно применять класс QSqlQuery для выполнения любых запросов SQL поддерживаемых используемой базой данных.

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

После выполнения запроса, можно посмотреть результат:

```
while (query.next()) {  
    QString country = query.value(0).toString();  
    int year = query.value(1).toInt();  
}
```

Функция next передвигает указатель на следующую возвращенную запись, когда записей не осталось, либо возвратилась ошибка, то возвращается false.

Функция value возвращает QVariant на ячейку.

Указание полей в шаблоне запроса при операциях вставки (используя имена):

```
QSqlQuery query;  
query.prepare("INSERT INTO person (id, forename, surname) "  
              "VALUES (:id, :forename, :surname)");  
query.bindValue(":id", 1001);  
query.bindValue(":forename", "Bart");  
query.bindValue(":surname", "Simpson");  
query.exec();
```

Указание полей в шаблоне запроса при операциях вставки (используя позиции):

```
QSqlQuery query;  
query.prepare("INSERT INTO person (id , forename ,  
    surname) "  
            "VALUES (:id , : forename , : surname)");  
query.bindValue(0, 1001);  
query.bindValue(1, "Bart");  
query.bindValue(2, "Simpson");  
query.exec();
```

Указание полей в шаблоне запроса при операциях вставки (используя позиции, версия 2):

```
QSqlQuery query;  
query.prepare("INSERT INTO person (id , forename ,  
    surname) "  
            "VALUES ( ? , ? , ? ) " );  
query.bindValue(0, 1001);  
query.bindValue(1, "Bart");  
query.bindValue(2, "Simpson");  
query.exec();
```

Указание полей в шаблоне запроса при операциях вставки (используя позиции, версия 3):

```
QSqlQuery query;  
query.prepare("INSERT INTO person (id , forename ,  
    surname) "  
    "VALUES ( ? , ? , ? )" );  
query.addBindValue(1001);  
query.addBindValue("Bart");  
query.addBindValue("Simpson");  
query.exec();
```

Указание полей в шаблоне запроса при операциях вставки (средствами подстановки Qt):

```
QSqlQuery query;  
QString str("INSERT INTO person (id , forename ,  
    surname) VALUES (%1,%2,%3)");  
str.arg(1001);  
str.arg("Bart");  
str.arg("Simpson");  
query.exec(str);
```

Передача значений в хранимые процедуры в базе данных:

```
QSqlQuery query;  
query.prepare("CALL AsciiToInt(?,?)");  
query.bindValue(0, "A");  
query.bindValue(1, 0, QSql::Out);  
query.exec();  
int i = query.boundValue(1).toInt();
```

Программный уровень

Qt поддерживает транзакции в тех базах где они предусмотрены, для активации транзакции используется функция `transaction()` класса `QSqlDatabase`.

```
QSqlDatabase::database().transaction();
QSqlQuery query("select id from artist where name =
    'Gluecifer'");
if(query.next())
{
    int artistId = query.value(0).toInt();
    query.exec("insert into cd(id, artistid, title,
        year) VALUES(201, " + QString::number(artistId)
        ) + ", 'Riding the Tiger', 1997)");
}
QSqlDatabase::database().commit();
```

Закрепление транзакции осуществляется функцией `commit()`, а отмена — `rollback()`.

Поддерживает ли база данных транзакции можно проверить функцией `hasFeature`

Представляет собой модели для отображения результата запросов: QSqlQueryModel, QSqlTableModel, QSqlRelationTableModel.

Концепция модель-представление

Концепция модель-представление позволяет предоставлять данные многократно без дублирования.

Возможность быстро адаптировать изменения под новую модель хранения данных

Удобство тестирования, позволяющие использовать разные тесты для любой модели реализующей заданный интерфейс.

Концепция модель-представление

Модель — отвечает за управление данными и предоставляет интерфейс для их чтения и записи.

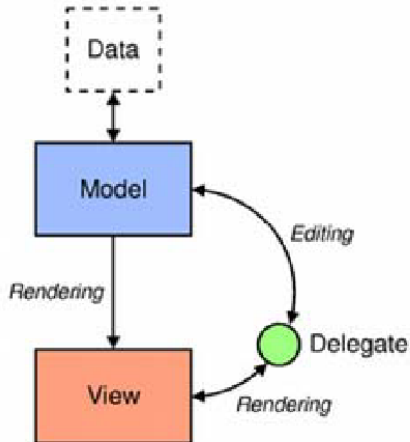
Представление — представление данных пользователю и их расположение.

Делегат — отвечает за рисование и редактирование каждого элемента.

Возможность быстро адаптировать изменения под новую модель хранения данных

Удобство тестирования, позволяющие использовать разные тесты для любой модели реализующей заданный интерфейс.

Архитектура модель-представление



Архитектура модель-представление

List Model

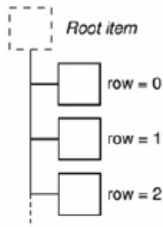
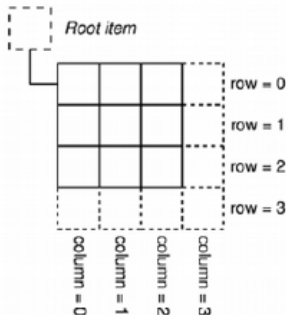
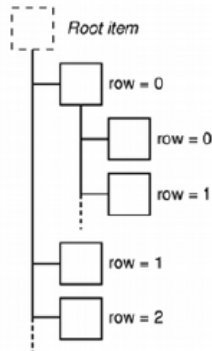


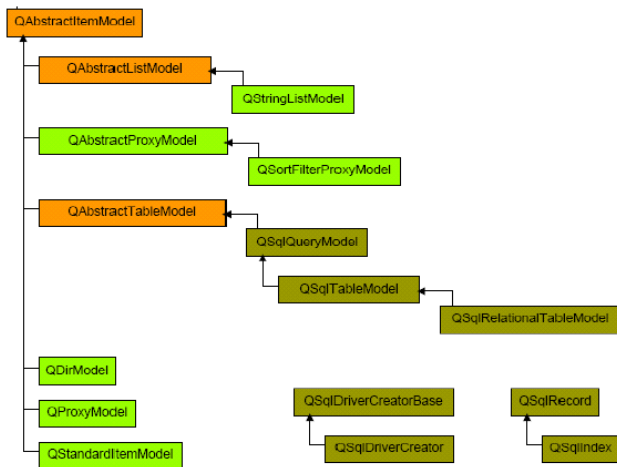
Table Model



Tree Model



Архитектура модель-представление



Уровень пользовательского интерфейса

Пример использование модели для осуществления выборки данных:

```
QSqlTableModel model;  
model.setTable("cd");  
model.setFilter("year >= 1998");  
model.select();
```

Аналогично запросу:

```
select * from cd where year >= 1998;
```

Уровень пользовательского интерфейса

Для просмотра результатов результирующей выборки используется функция `record()`.

```
for( int i = 0; i < model.rowCount(); i++)  
{  
    QSqlRecord record = model.record(i);  
    QString title = record.value("title").toString();  
    int year = record.value("year").toInt();  
    qDebug << title << "□" << year;  
}
```

`QSqlRecord::value()` может принимать имя поля, либо его индекс.

Уровень пользовательского интерфейса

Вставка данных в модель:

```
QSqlTableModel model;  
model.setTable("cd");  
model.insertRows(0, 1);  
model.setData(model.index(0, 0), 113);  
model.setData(model.index(0, 1), "Test");  
model.setData(model.index(0, 2), 224);  
model.setData(model.index(0, 3), 2004);  
model.submitAll();
```

Для вставки используется функция `insertRow()`, для вставки пустой строки и функция `setData()` — для задания значений столбцов, принимает в качестве параметров указатель на ячейку (`QModelIndex`).

После вызова `submitAll()` добавленная запись может быть перемещена в другую позицию, в зависимости от упорядоченности таблицы. А также для записи изменений в базу данных.

Обновление данных:

```
QSqlTableModel model;  
model.setTable("cd");  
model.setFilter("id=12");  
model.select();  
if(model.rowCount() == 1)  
{  
    QSqlRecord record = model.record(0);  
    record.setValue("title", "Test");  
    record.setValue("year", record.value("year").toInt  
        () + 1);  
    model.setRecord(0, record);  
    model.submitAll();  
}
```

Удаление данных:

```
QSqlTableModel model;  
model.setTable("cd");  
model.setFilter("id=12");  
model.select();  
if(model.rowCount() == 1)  
{  
    model.removeRows(0, 1);  
    model.submitAll();  
}
```

В `removeRows()` указывается номер строки с которой произвести удаление и число удаляемых строк.

Уровень пользовательского интерфейса

Просмотр модели и ее связь с отображением для пользователя. Для связи модели с отображением используются представления `QTableView` и функция `setModel`.

```
model->setHeaderData(0, Qt::Horizontal, "Name");  
QTableView *view = new QTableView;  
view->setModel(model);  
view->setColumnHidden(0, true);  
view->resizeColumnsToContents();  
view->setEditTriggers(AbstractItemView::  
    AllEditTriggers)
```
