



Qt程序设计方法

数据库和XML

School of Computer Science
and Engineering

2024-2025-暑期课程

主要内容

- 数据库
- XML



数据库

Qt中的**Qt SQL**模块提供了对数据库的支持，该模块中的众多类基本上可以分为三层：

- 驱动层为具体的数据库和**SQL**接口层之间提供了底层的桥梁；
- **SQL**接口层提供了对数据库的访问，其中的**QSqlDatabase**类用来创建连接，**QSqlQuery**类可以使用**SQL**语句来实现与数据库交互，其他几个类对该层提供了支持；
- 用户接口层的几个类实现了将数据库中的数据链接到窗口部件上，这些类是使用前一章的模型/视图框架实现的，它们是更高层次的抽象，即便不熟悉**SQL**也可以操作数据库。

用户接口层	QSqlQueryModel 、 QSqlTableModel 和 QSqlRelationalTableModel
SQL 接口层	QSqlDatabase 、 QSqlQuery 、 QSqlError 、 QSqlField 、 QSqlIndex 和 QSqlRecord
驱动层	QSqlDriver 、 QSqlDriverCreator 、 QSqlDriverCreatorBase 、 QSqlDriverPlugin 和 QSqlResult



数据库驱动

- **Qt SQL**模块使用数据库驱动插件来和不同的数据库接口进行通信。由于**Qt SQL**模块的接口是独立于数据库的，所以所有数据库特定的代码都包含在了这些驱动中。**Qt**默认支持一些驱动：

驱动名称	数据库
QDB2	IBM DB2（7.1 版或者以上版本）
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Oracle Call Interface Driver
QODBC	Open Database Connectivity(ODBC)- 微软 SQL Server 和其他 ODBC 兼容数据库
QPSQL	PostgreSQL（7.3 版本或者更高）
QSQLITE2	SQLite 版本 2
QSQLITE	SQLite 版本 3
QTDS	Sybase Adaptive Server 注：从 Qt 4.7 开始已经过时

- 这里要重点提一下**SQLite**数据库，它是一款轻型的文件型数据库，无需数据库服务器，主要应用于嵌入式领域，支持跨平台，而且**Qt**对它提供了很好的默认支持。

示例：遍历输出驱动列表

```
#include <QApplication>
```

```
#include <QSqlDatabase>
```

```
#include <QDebug>
```

```
#include <QStringList>
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    QApplication a(argc, argv);
```

```
    qDebug() << "Available drivers:";
```

```
    QStringList drivers = QSqlDatabase::drivers();
```

```
    foreach(QString driver, drivers)
```

```
        qDebug() << driver;
```

```
    return a.exec();
```

```
}
```

创建数据库连接

- 要想使用**QSqlQuery**或者**QSqlQueryModel**来访问数据库，那么先要创建并打开一个或者多个数据库连接。数据库连接使用连接名来定义，而不是使用数据库名，可以向相同的数据库创建多个连接。**QSqlDatabase**也支持默认连接的概念，默认连接就是一个没有命名的连接。在使用**QSqlQuery**或者**QSqlQueryModel**的成员函数时需要指定一个连接名作为参数，如果没有指定，那么就会使用默认连接。如果在应用程序中只需要有一个数据库连接，那么使用默认连接是很方便的。示例：

原型：**QSqlDatabase QSqlDatabase::addDatabase(const QString &type, const QString &connectionName = QLatin1String(defaultConnection))**

```
QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");  
db.setHostName("bigblue");  
db.setDatabaseName("flightdb");  
db.setUserName("acarlson");  
db.setPassword("1uTbSbAs");  
bool ok = db.open();
```

同时创建多个连接

- 下面的示例代码中创建了两个名为“first”和“second”的连接：

```
QSqlDatabase firstDB = QSqlDatabase::addDatabase("QMYSQL", "first");  
QSqlDatabase secondDB = QSqlDatabase::addDatabase("QMYSQL",  
"second");
```

- 创建完连接后，可以在任何地方使用**QSqlDatabase::database()**静态函数通过连接名称获取指向数据库连接的指针，如果调用该函数时没有指明连接名称，那么会返回默认连接，例如：

```
QSqlDatabase defaultDB = QSqlDatabase::database();  
QSqlDatabase firstDB = QSqlDatabase::database("first");  
QSqlDatabase secondDB = QSqlDatabase::database("second");
```

- 要移除一个数据库连接，需要先使用**QSqlDatabase::close()**关闭数据库，然后使用静态函数**QSqlDatabase::removeDatabase()**移除该连接。

执行SQL语句

- **QSqlQuery**类提供了一个接口，用于执行**SQL**语句和浏览查询的结果集。要执行一个**SQL**语句，只需要简单的创建一个**QSqlQuery**对象，然后调用**QSqlQuery::exec()**函数即可。例如：

```
QSqlQuery query;
```

```
query.exec("select * from student");
```

- **QSqlQuery**提供了对结果集的访问，可以一次访问一条记录。当执行完**exec()**函数后，**QSqlQuery**的内部指针会位于第一条记录前面的位置。必须调用一次**QSqlQuery::next()**函数来使其前进到第一条记录，然后可以重复使用**next()**函数来访问其他的记录，直到该函数的返回值为**false**，例如可以使用以下代码来遍历一个结果集：

```
while(query.next())
```

```
{ qDebug() << query.value(0).toInt() << query.value(1).toString(); }
```

- 在**QSqlQuery**类中提供了多个函数来实现在结果集中进行定位，比如**next()**定位到下一条记录，**previous()**定位到前一条记录，**first()**定位的第一条记录，**last()**定位到最后一条记录，**seek(n)**定位到第**n**条记录。当前行的索引可以使用**at()**返回；**record()**函数可以返回当前指向的记录；如果数据库支持，那么可以使用**size()**来返回结果集中的总行数。

使用占位符

- 插入一条记录: `query2.exec("insert into student (id, name) values (100, 'ChenYun')");`
- 如果想在同一时间插入多条记录, 那么一个有效的方法就是将查询语句和真实的值分离, 这可以使用占位符来完成。Qt支持两种占位符: 名称绑定和位置绑定。

名称绑定:

```
query2.prepare("insert into student (id, name) values  
(:id, :name)");  
  
int idValue = 100;  
  
QString nameValue = "ChenYun";  
  
query2.bindValue(":id", idValue);  
  
query2.bindValue(":name", nameValue);  
  
query2.exec();
```

位置绑定:

```
query2.prepare("insert into student (id, name)  
values (?, ?)");  
  
int idValue = 100;  
  
QString nameValue = "ChenYun";  
  
query2.addBindValue(idValue);  
  
query2.addBindValue(nameValue);  
  
query2.exec();
```

批处理

- 当要插入多条记录时，只需要调用 **QSqlQuery::prepare()** 一次，然后使用多次 **bindValue()** 或者 **addBindValue()** 函数来绑定需要的数据，最后调用一次 **exec()** 函数就可以了。其实，进行多条数据插入时，还可以使用批处理进行：

```
query2.prepare("insert into student (id, name) values (?, ?)");  
QVariantList ids;  
ids << 20 << 21 << 22;  
query2.addBindValue(ids);  
QVariantList names;  
names << "xiaoming" << "xiaoliang" << "xiaogang";  
query2.addBindValue(names);  
if(!query2.execBatch()) qDebug() << query2.lastError();
```

事务

- 事务可以保证一个复杂的操作的原子性，就是对于一个数据库操作序列，这些操作要么全部做完，要么一条也不做，是不可分割的工作单位。如果底层的数据库引擎支持事务，
QSqlDriver::hasFeature(QSqlDriver::Transactions)会返回**true**。可以使用**QSqlDatabase::transaction()**来启动一个事务，然后编写希望在事务中执行的SQL语句，最后调用**QSqlDatabase::commit()**提交或者**QSqlDatabase::rollback()**回滚。使用事务必须在创建查询以前就开始事务。

```
QSqlDatabase::database().transaction();
```

```
QSqlQuery query;
```

```
query.exec("SELECT id FROM employee WHERE name = 'Torild  
Halvorsen'");
```

```
if (query.next()) {
```

```
    int employeeld = query.value(0).toInt();
```

```
    query.exec("INSERT INTO project (id, name, ownerid) "  
              "VALUES (201, 'Manhattan Project', "  
              + QString::number(employeeld) + ')');
```

```
}
```

```
QSqlDatabase::database().commit();
```

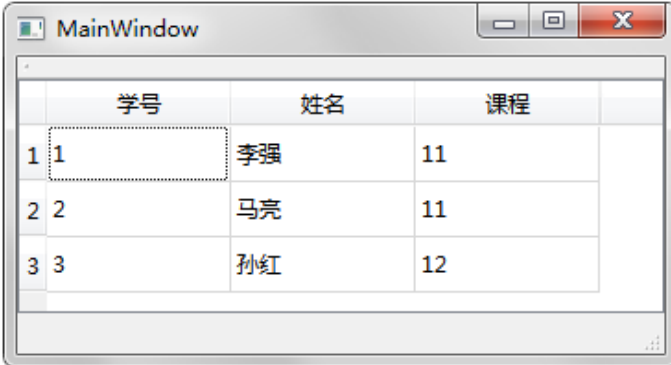
使用SQL模型类

- 除了**QSqlQuery**，Qt还提供了3个更高层的类来访问数据库，分别是**QSqlQueryModel**、**QSqlTableModel**和**QSqlRelationalTableModel**。
- 这3个类都是从**QAbstractTableModel**派生来的，可以很容易地实现将数据库中的数据在**QListView**和**QTableView**等项视图类中进行显示。使用这些类的另一个好处是，这样可以使编写的代码很容易的适应其他的数据源。例如，如果开始使用了**QSqlTableModel**，而后来要改为使用**XML**文件来存储数据，这样需要做的仅是更换一个数据模型。

SQL查询模型 QSqlQueryModel

QSqlQueryModel提供了一个基于SQL查询的只读模型。例如：

```
QSqlQueryModel *model = new QSqlQueryModel(this)
model->setQuery("select * from student");
model->setHeaderData(0, Qt::Horizontal, tr("学号"));
model->setHeaderData(1, Qt::Horizontal, tr("姓名"));
model->setHeaderData(2, Qt::Horizontal, tr("课程"));
QTableView *view = new QTableView(this);
view->setModel(model);
setCentralWidget(view);
```



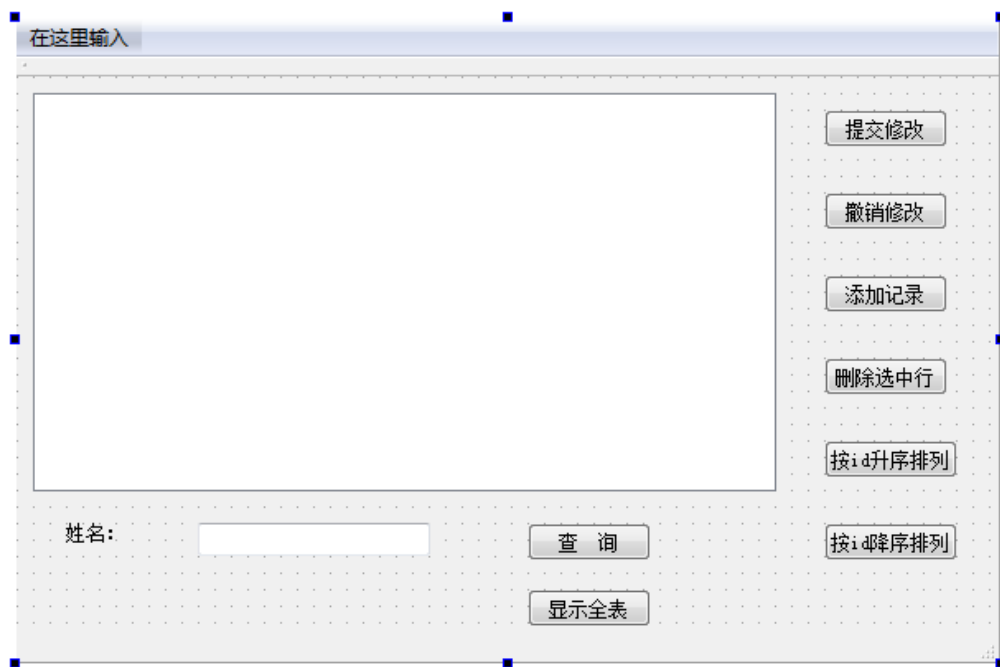
	学号	姓名	课程
1	1	李强	11
2	2	马亮	11
3	3	孙红	12

这里先创建了QSqlQueryModel对象，然后使用setQuery()来执行SQL语句查询整张student表，并使用setHeaderData()来设置显示的标头。后面创建了视图，并将QSqlQueryModel对象作为其要显示的模型。这里要注意，其实QSqlQueryModel中存储的是执行完setQuery()函数后的结果集，所以视图中显示的是结果集的内容。

QSqlQueryModel中还提供了columnCount()返回一条记录中字段的个数；rowCount()返回结果集中记录的条数；record()返回第n条记录；index()返回指定记录的指定字段的索引；clear()可以清空模型中的结果集。

SQL表格模型 QSqlTableModel

- **QSqlTableModel**提供了一个一次只能操作一个**SQL**表的读写模型，它是**QSqlQuery**的更高层次的替代品，可以浏览和修改独立的**SQL**表，并且只需编写很少的代码，而且不需要了解**SQL**语法。例如：



■ 创建数据表

QSqlQuery query;

// 创建student表

```
query.exec("create table student (id int primary key, "  
            "name varchar, course int)");
```

```
query.exec("insert into student values(1, '李强', 11)");
```

```
query.exec("insert into student values(2, '马亮', 11)");
```

```
query.exec("insert into student values(3, '孙红', 12)");
```

// 创建course表

```
query.exec("create table course (id int primary key, "  
            "name varchar, teacher varchar)");
```

```
query.exec("insert into course values(10, '数学', '王老师')");
```

```
query.exec("insert into course values(11, '英语', '张老师')");
```

```
query.exec("insert into course values(12, '计算机', '白老师')");
```

■ 显示表:

```
model = new QSqlTableModel(this);  
model->setTable("student");  
model->select();  
// 设置编辑策略  
model->setEditStrategy(QSqlTableModel::OnManualSubmit);  
ui->tableView->setModel(model);
```

这里创建一个**QSqlTableModel**后，只需使用**setTable()**来为其指定数据库表，然后使用**select()**函数进行查询，调用这两个函数就等价于执行了“**select * from student**”语句。这里还可以使用**setFilter()**来指定查询时的条件。在使用该模型以前，一般还要设置其编辑策略，它由**QSqlTableModel::EditStrategy**枚举类型定义。

常量	描述
QSqlTableModel::OnFieldChange	所有对模型的改变都会立即应用到数据库
QSqlTableModel::OnRowChange	对一条记录的改变会在用户选择另一条记录时被应用
QSqlTableModel::OnManualSubmit	所有的改变都会在模型中进行缓存，直到调用 submitAll() 或者 revertAll() 函数

■ 修改操作

// 提交修改按钮

```
void MainWindow::on_pushButton_clicked()
{ // 开始事务操作
    model->database().transaction();
    if (model->submitAll()) {
        if(model->database().commit()) // 提交
            QMessageBox::information(this, tr("tableModel"),
                                     tr("数据修改成功! "));
    } else {
        model->database().rollback(); // 回滚
        QMessageBox::warning(this, tr("tableModel"),
                             tr("数据库错误: %1").arg(model->lastError().text()),
                             QMessageBox::Ok);
    }
}
```

// 撤销修改按钮

```
void MainWindow::on_pushButton_2_clicked()
{
    model->revertAll();
}
```

■ 筛选操作

// 查询按钮，进行筛选

```
void MainWindow::on_pushButton_5_clicked()
{
    QString name = ui->lineEdit->text();
    // 根据姓名进行筛选，一定要使用单引号
    model->setFilter(QString("name = '%1'").arg(name));
    model->select();
}

// 显示全表按钮

void MainWindow::on_pushButton_6_clicked()
{
    model->setTable("student");
    model->select();
}
```

■ 排序操作

// 按id升序排列按钮

```
void MainWindow::on_pushButton_7_clicked()
{
    //id字段，即第0列，升序排列
    model->setSort(0, Qt::AscendingOrder);
    model->select();
}
```

// 按id降序排列按钮

```
void MainWindow::on_pushButton_8_clicked()
{
    model->setSort(0, Qt::DescendingOrder);
    model->select();
}
```

■ 删除记录

// 删除选中行按钮

```
void MainWindow::on_pushButton_4_clicked()
```

```
{
```

```
    // 获取选中的行
```

```
    int curRow = ui->tableView->currentIndex().row();
```

```
    // 删除该行
```

```
    model->removeRow(curRow);
```

```
    int ok = QMessageBox::warning(this, tr("删除当前行!"),  
        tr("你确定删除当前行吗? "), QMessageBox::Yes, QMessageBox::No);
```

```
    if(ok == QMessageBox::No)
```

```
    { // 如果不删除，则撤销
```

```
        model->revertAll();
```

```
    } else { // 否则提交，在数据库中删除该行
```

```
        model->submitAll();
```

```
    }
```

```
}
```

■ 添加记录

// 添加记录按钮

```
void MainWindow::on_pushButton_3_clicked()
```

```
{
```

```
    // 获得表的行数
```

```
    int rowNum = model->rowCount();
```

```
    int id = 10;
```

```
    // 添加一行
```

```
    model->insertRow(rowNum);
```

```
    model->setData(model->index(rowNum,0), id);
```

```
    // 可以直接提交
```

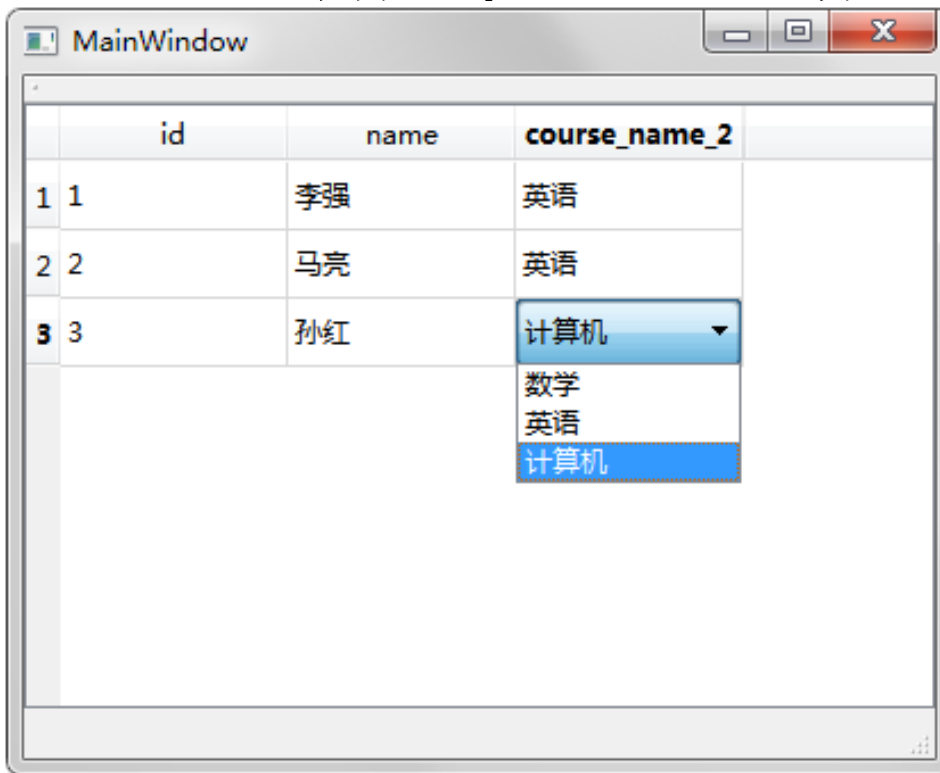
```
    //model->submitAll();
```

```
}
```

SQL关系表格模型 QSqlRelationalTableModel

- **QSqlRelationalTableModel**继承自**QSqlTableModel**，并且对其进行了扩展，提供了对外键的字段之间的一对一关系。他表中的主键对应的就是这里的表格模型，就可以将它显示为

```
QSqlRelationalTableModel *model = new QSqlRelationalTableModel(view, db, QSqlRelationalTableModel::SetEditStrategy(QSqlRelationalTableModel::OnFieldChange));  
model->setTable("course");  
model->setRelation("course_id", "course_name_2");  
model->select();  
QTableView *view = new QTableView;  
view->setModel(model);  
setCentralWidget(view);
```



- Qt中还提供了一个**QSqlRelationalDelegate**委托类，它可以为**QSqlRelationalTableModel**显示和编辑数据。这个委托为一个外键提供了一个**QComboBox**部件来显示所有可选的数据，这样就显得更加人性化了。
`view->setItemDelegate(new QSqlRelationalDelegate(view));`

XML

- **XML(Extensible Markup Language, 可扩展标记语言)**, 是一种类似于**HTML**的标记语言, 设计目的是用来传输数据, 而不是显示数据。**XML**的标签没有被预定义, 用户需要在使用时自行进行定义。**XML**是**W3C**(万维网联盟)的推荐标准。相对于数据库表格的二维表示, **XML**使用的树形结构更能表现出数据的包含关系, 作为一种文本文件格式, **XML**简单明了的特性使得它在信息存储和描述领域非常流行。
- **Qt**中提供了**Qt XML**模块来进行**XML**文档的处理, 这里主要提供了两种解析方法: **DOM**方法, 可以进行读写; **SAX**方法, 可以进行读取。但是, 从**Qt 5**开始**Qt XML**模块不再提供维护, 而是推荐使用**Qt Core**模块中的**QXmlStreamReader**和**QXmlStreamWriter**进行**XML**读取和写入, 这是一种基于流的方法。
- 如果要使用**Qt XML**模块, 需要在项目文件 (**.pro**文件) 中添加**QT += xml**一行代码。



标准的XML文档示例

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<library>
```

```
  <book id="01">
```

```
    <title>Qt</title>
```

```
    <author>shiming</author>
```

```
  </book>
```

```
  <book id="02">
```

```
    <title>Linux</title>
```

```
    <author>yafei</author>
```

```
  </book>
```

```
</library>
```


DOM

- DOM（Document Object Model，文档对象模型），是W3C的推荐标准。它提供了一个接口来访问和改变一个XML文件的内容和结构，可以将XML文档表示为一个存储在内存中具有层次的树视图。文档本身由QDomDocument对象来表示，而文档树中所有的DOM节点都是QDomNode类的子类。
- 在Qt中使用QDomProcessingInstruction类来表示XML说明。元素对应QDomElement类。属性对应QDomAttr类。文本内容由QDomText类表示。所有的DOM节点，比如这里的说明、元素、属性和文本等，都使用QDomNode来表示。

使用DOM读取XML文档

```
// 新建QDomDocument类对象，它代表一个XML文档
QDomDocument doc;
QFile file("../myDOM1/my.xml");
if (!file.open(QIODevice::ReadOnly)) return 0;
if (!doc.setContent(&file)) { // 将文件内容读到doc中
    file.close();
    return 0;
}
```

```
file.close();// 关闭文件
```

```
QDomNode firstNode = doc.firstChild(); // 获得第一个结点，即XML说明
```

```
// 输出XML说明,nodeName()为“xml”,nodeValue()为版本和编码信息
qDebug() << qPrintable(firstNode.nodeName())
          << qPrintable(firstNode.nodeValue());
```

```
QDomElement docElem = doc.documentElement(); // 返回根元素
```

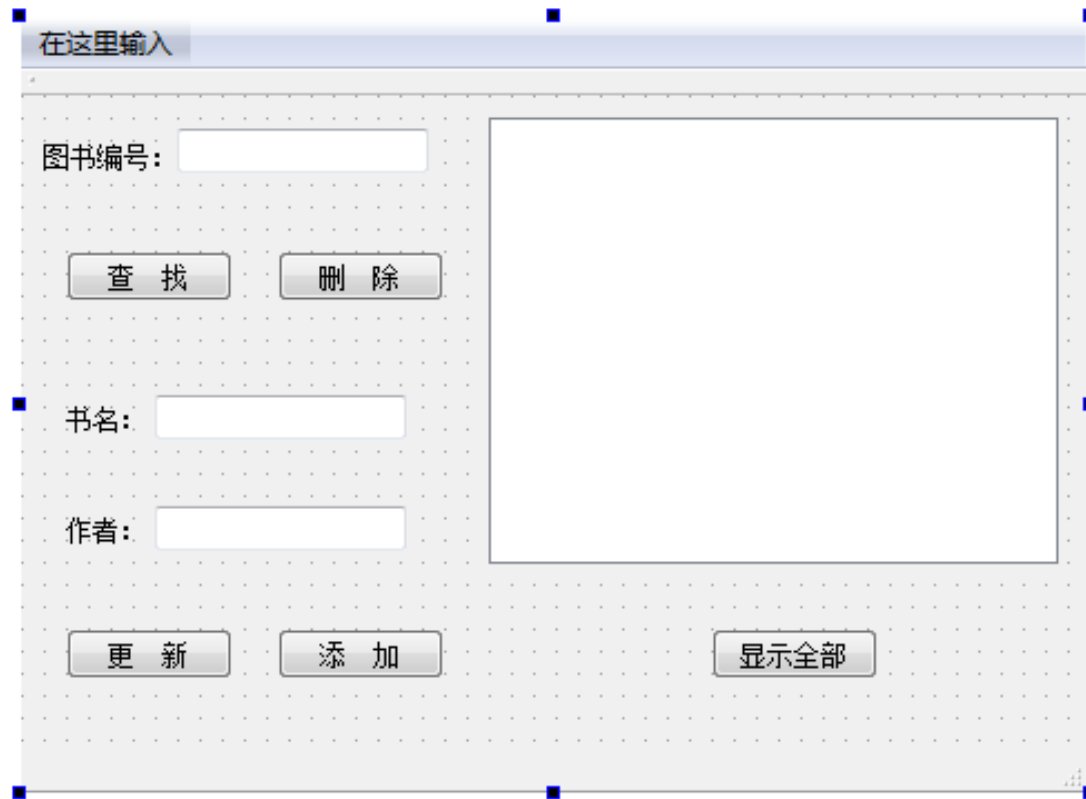
```
QDomNode n = docElem.firstChild();// 返回根节点的第一个子结点
```

```
// 如果结点不为空，则转到下一个节点
while(!n.isNull())
{
    if (n.isElement()) // 如果结点是元素
    {
        QDomElement e = n.toElement(); // 将其转换为元素

        // 返回元素标记和id属性的值
        qDebug() << qPrintable(e.tagName())
                << qPrintable(e.attribute("id"));
        // 获得元素e的所有子结点的列表
        QDomNodeList list = e.childNodes();
        // 遍历该列表
        for(int i=0; i<list.count(); i++)
        {
            QDomNode node = list.at(i);
            if(node.isElement())
                qDebug() << " " << qPrintable(node.toElement().tagName())
                        << qPrintable(node.toElement().text());
        }
    }
    n = n.nextSibling(); // 转到下一个兄弟结点
}
```

使用DOM创建和操作XML文档

示例：设计界面，向界面拖入**Push Button**、**List Widget**、**Label**和**Line Edit**等部件设计界面，最终效果如图所示。



■ 生成XML文件

```
QDomDocument doc;  
QDomProcessingInstruction instruction; // 添加处理指令即XML说明  
instruction = doc.createProcessingInstruction("xml",  
                                             "version=\"1.0\" encoding=\"UTF-8\"");  
  
doc.appendChild(instruction);  
QDomElement root = doc.createElement("书库"); // 添加根元素  
doc.appendChild(root);  
// 添加图书元素及其子元素  
QDomElement book = doc.createElement("图书");  
QDomAttr id = doc.createAttribute("编号");  
QDomElement title = doc.createElement("书名");  
QDomElement author = doc.createElement("作者");  
QDomText text;  
id.setValue(QString("1"));  
book.setAttributeNode(id);  
text = doc.createTextNode("Qt");  
title.appendChild(text);  
text = doc.createTextNode("shiming");  
author.appendChild(text);  
book.appendChild(title);  
book.appendChild(author);  
root.appendChild(book);
```

```
// 添加第二个图书元素及其子元素
book = doc.createElement("图书");
id = doc.createAttribute("编号");
title = doc.createElement("书名");
author = doc.createElement("作者");
id.setValue(QString("2"));
book.setAttributeNode(id);
text = doc.createTextNode("Linux");
title.appendChild(text);
text = doc.createTextNode("yafei");
author.appendChild(text);
book.appendChild(title);
book.appendChild(author);
root.appendChild(book);
```

```
QFile file("my.xml");
if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate)) return ;
QTextStream out(&file);
// 将文档保存到文件，4为子元素缩进字符数
doc.save(out, 4);
file.close();
```

■ 显示全部

```
ui->listWidget->clear(); // 先清空显示
QFile file("my.xml");
if (!file.open(QIODevice::ReadOnly)) return ;
QDomDocument doc;
if (!doc.setContent(&file)) { file.close(); return ; }
file.close();
QDomElement docElem = doc.documentElement();
QDomNode n = docElem.firstChild();
while(!n.isNull()) {
    if (n.isElement()) {
        QDomElement e = n.toElement();
        ui->listWidget->addItem(e.tagName() + e.attribute("编号"));
        QDomNodeList list = e.childNodes();
        for (int i=0; i<list.count(); i++) {
            QDomNode node = list.at(i);
            if(node.isElement())
                ui->listWidget->addItem(" " + node.toElement().tagName()
                    + " : " + node.toElement().text()); }
        n = n.nextSibling();
    }
}
```

■ 添加一个元素

// 先清空显示，然后显示“无法添加！”，

// 这样如果添加失败则会显示“无法添加！”

```
ui->listWidget->clear();
```

```
ui->listWidget->addItem(tr("无法添加！"));
```

```
QFile file("my.xml");
```

```
if (!file.open(QIODevice::ReadOnly)) return;
```

```
QDomDocument doc;
```

```
if (!doc.setContent(&file)) { file.close(); return; }
```

```
file.close();
```

```
QDomElement root = doc.documentElement();
```

```
QDomElement book = doc.createElement("图书");
```

```
QDomAttr id = doc.createAttribute("编号");
```

```
QDomElement title = doc.createElement("书名");
```

```
QDomElement author = doc.createElement("作者");
```

```
QDomText text;
```



```
// 我们获得了最后一个孩子结点的编号，然后加1，便是新的编号
QString num = root.lastChild().toElement().attribute("编号");
int count = num.toInt() + 1;
id.setValue(QString::number(count));
book.setAttributeNode(id);
text = doc.createTextNode(ui->lineEdit_2->text());
title.appendChild(text);
text = doc.createTextNode(ui->lineEdit_3->text());
author.appendChild(text);
book.appendChild(title);
book.appendChild(author);
root.appendChild(book);
if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate)) return ;
QTextStream out(&file);
doc.save(out, 4);
file.close();
ui->listWidget->clear(); // 最后更改显示为“添加成功！”
ui->listWidget->addItem(tr("添加成功！"));
```

■ 查找、删除和更新内容

```
void MainWindow::doXml(const QString operate)
```

```
{
    ui->listWidget->clear();
    ui->listWidget->addItem(tr("没有找到相关内容! "));
    QFile file("my.xml");
    if (!file.open(QIODevice::ReadOnly)) return ;
    QDomDocument doc;
    if (!doc.setContent(&file)) { file.close(); return ; }
    file.close();
    QDomNodeList list = doc.elementsByTagName("图书"); // 以标签名进行查找
    for(int i=0; i<list.count(); i++)
    {
        QDomElement e = list.at(i).toElement();
        if(e.attribute("编号") == ui->lineEdit->text())
        { // 如果元素的“编号”属性值与我们所查的相同
            if (operate == "delete") { // 如果是删除操作
                QDomElement root = doc.documentElement();
                root.removeChild(list.at(i)); // 从根节点上删除该节点
                QFile file("my.xml");
                if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate)) return ;
                QTextStream out(&file);
                doc.save(out,4);
                file.close();
                ui->listWidget->clear();
                ui->listWidget->addItem(tr("删除成功! "));
            }
        }
    }
}
```

```
else if (operate == "update") { // 如果是更新操作
    QDomNodeList child = list.at(i).childNodes();
    // 将它子节点的首个子节点（就是文本节点）的内容更新
    child.at(0).toElement().firstChild()
        .setNodeValue(ui->lineEdit_2->text());
    child.at(1).toElement().firstChild()
        .setNodeValue(ui->lineEdit_3->text());
    QFile file("my.xml");
    if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate)) return ;
    QTextStream out(&file);
    doc.save(out,4);
    file.close();
    ui->listWidget->clear();
    ui->listWidget->addItem(tr("更新成功! "));
} else if (operate == "find") { // 如果是查找操作
    ui->listWidget->clear();
    ui->listWidget->addItem(e.tagName()
        + e.attribute("编号"));
    QDomNodeList list = e.childNodes();
    for(int i=0; i<list.count(); i++)
    {
        QDomNode node = list.at(i);
        if(node.isElement())
            ui->listWidget->addItem(" "
                + node.toElement().tagName() + " : "
                + node.toElement().text());
    }
}
```

SAX

SAX（**Simple API for XML**）为XML解析器提供了一个基于事件的标准接口。在前面讲解的**DOM**方法需要在一个树结构中读入和存储整个XML文档，这样会耗费大量内存，不过使用它来操作文档结构是很容易的。如果不需要对文档进行操作，而只需要读取整个XML文档，那么使用**SAX**方法就很高效了。**SAX2**接口是一种事件驱动机制，用来为用户提供文档信息。这里的事件是由解析器发出的，例如它遇到了一个开始标签或者一个结束标签等。下面来看一个例子：

<quote>A quotation.</quote>

当解析上面这行文档时会触发三个事件：

- 遇到开始标签（**<quote>**），这时会调用**startElement()**事件处理函数；
- 发现字符数据“**A quotation.**”，这时会调用**characters()**事件处理函数；
- 一个结束标签被解析（**</quote>**），这时会调用**endElement()**事件处理函数。

- 每当发生一个事件时，都可以在相应的事件处理函数中进行操作来完成对文档的自定义解析。比如可以在**startElement()**中获得元素名和属性，在**characters()**中获得元素中的文本，在**endElement()**中进行一些结束读取该元素时想要进行的操作。这些事件处理函数都可以通过继承**QXmlDefaultHandler**类来重新实现，Qt中提供的事件处理类如表所列，它们都是继承自**QXmlDefaultHandler**类的。

处理类	描述
QXmlContentHandler	报告与文档内容相关的事件（例如起始标签或字符）
QXmlDTDHandler	报告与 DTD 相关的事件
QXmlErrorHandler	报告在解析过程中发生的错误或警告
QXmlEntityResolver	报告解析过程中的外部实体，允许用户解析外部实体
QXmlDeclHandler	报告 XML 数据的声明内容
QXmlLexicalHandler	报告与文档的词法结构相关的事件

■ 自定义MySAX类

```
class MySAX : public QXmlDefaultHandler
```

```
{
```

```
public:
```

```
    MySAX();
```

```
    ~MySAX();
```

```
    bool readFile(const QString &fileName);
```

```
protected:
```

```
    bool startElement(const QString &namespaceURI, const QString &localName,  
                    const QString &qName, const QXmlAttributes &atts);
```

```
    bool endElement(const QString &namespaceURI, const QString &localName,  
                  const QString &qName);
```

```
    bool characters(const QString &ch);
```

```
    bool fatalError(const QXmlParseException &exception);
```

```
private:
```

```
    QListWidget *list;
```

```
    QString currentText;
```

```
};
```

```
bool MySAX::readFile(const QString &fileName)
{
    QFile file(fileName);
    QDomInputSource inputSource(&file); // 读取文件内容
    QDomSimpleReader reader; // 建立QDomSimpleReader对象
    reader.setContentHandler(this); // 设置内容处理器
    reader.setErrorHandler(this); // 设置错误处理器
    return reader.parse(inputSource); // 解析文件
}

// 已经解析完一个元素的起始标签
bool MySAX::startElement(const QString &namespaceURI, const QString &localName,
                        const QString &qName, const QDomAttributes &atts)
{
    if (qName == "library")
        list->addItem(qName);
    else if (qName == "book")
        list->addItem("  " + qName + atts.value("id"));
    return true;
}
```

// 已经解析完一块字符数据

bool MySAX::characters(const QString &ch)

```
{  
    currentText = ch;  
    return true;
```

```
}
```

// 已经解析完一个元素的结束标签

**bool MySAX::endElement(const QString &namespaceURI, const QString &localName,
 const QString &qName)**

```
{  
    if (qName == "title" || qName == "author")  
        list->addItem("      " + qName + " : " + currentText);  
    return true;
```

```
}
```

// 错误处理

bool MySAX::fatalError(const QDomParseException &exception)

```
{  
    qDebug() << exception.message();  
    return false;
```

```
}
```

XML流

常量	描述
<code>QXmlStreamReader::NoToken</code>	没有读到任何内容
<code>QXmlStreamReader::Invalid</code>	发生了一个错误，在 <code>error()</code> 和 <code>errorString()</code> 中报告
<code>QXmlStreamReader::StartDocument</code>	在 <code>documentVersion()</code> 中报告 XML 版本号，在 <code>documentEncoding()</code> 中指定文档的编码
<code>QXmlStreamReader::EndDocument</code>	报告文档结束
<code>QXmlStreamReader::StartElement</code>	使用 <code>namespaceUri()</code> 和 <code>name()</code> 来报告元素开始，可以使用 <code>attributes()</code> 来获取属性
<code>QXmlStreamReader::EndElement</code>	使用 <code>namespaceUri()</code> 和 <code>name()</code> 来报告元素结束
<code>QXmlStreamReader::Characters</code>	使用 <code>text()</code> 来报告字符，如果字符是空白，那么 <code>isWhitespace()</code> 返回 <code>true</code> ，如果字符源于 <code>CDATA</code> 部分，那么 <code>isCDATA()</code> 返回 <code>true</code>
<code>QXmlStreamReader::Comment</code>	使用 <code>text()</code> 报告一个注释
<code>QXmlStreamReader::DTD</code>	使用 <code>text()</code> 来报告一个 DTD，符号声明在 <code>notationDeclarations()</code> 中，实体声明在 <code>entityDeclarations()</code> 中，具体的 DTD 声明通过 <code>dtdName()</code> 、 <code>dtdPublicId()</code> 和 <code>dtdSystemId()</code> 来报告
<code>QXmlStreamReader::EntityReference</code>	报告一个无法解析的实体引用，引用的名字由 <code>name()</code> 获取， <code>text()</code> 可以获取替换文本
<code>QXmlStreamReader::ProcessingInstruction</code>	使用 <code>processingInstructionTarget()</code> 和 <code>processingInstructionData()</code> 来报告一个处理指令

使用QXmlStreamReader解析XML文档

```
QFile file("../myxmlstream/my.xml");
if (!file.open(QFile::ReadOnly | QFile::Text)) {
    qDebug() << "Error: cannot open file"; return 1; }
QXmlStreamReader reader;
reader.setDevice(&file); // 设置文件，这时会将流设置为初始状态
while (!reader.atEnd()) { // 如果没有读到文档结尾，而且没有出现错误
    // 读取下一个记号，它返回记号的类型
    QXmlStreamReader::TokenType type = reader.readNext();
    // 下面便根据记号的类型来进行不同的输出
    if (type == QXmlStreamReader::StartDocument)
        qDebug() << reader.documentEncoding() << reader.documentVersion();
    if (type == QXmlStreamReader::StartElement) {
        qDebug() << "<" << reader.name() << ">";
        if (reader.attributes().hasAttribute("id"))
            qDebug() << reader.attributes().value("id"); }
    if (type == QXmlStreamReader::EndElement)
        qDebug() << "</" << reader.name() << ">";
    if (type == QXmlStreamReader::Characters && !reader.isWhitespace())
        qDebug() << reader.text(); }
// 如果读取过程中出现错误，那么输出错误信息
if (reader.hasError()) { qDebug() << "error: " << reader.errorString();}
file.close();
```

使用QXmlStreamWriter写入XML文档

```
QFile file("../myxmlstream/my2.xml");  
if (!file.open(QFile::WriteOnly | QFile::Text))  
{  
    qDebug() << "Error: cannot open file";  
    return 1;  
}
```

```
QXmlStreamWriter stream(&file);  
stream.setAutoFormatting(true);  
stream.writeStartDocument();  
stream.writeStartElement("bookmark");  
stream.writeAttribute("href", "http://www.qt.io/");  
stream.writeTextElement("title", "Qt Home");  
stream.writeEndElement();  
stream.writeEndDocument();  
file.close();
```

小结

数据库和XML在很多程序中经常用到，它们的使用也总是和数据的显示联系起来，所以学习好前面一章的知识也是很重要的，这两章可以说是密不可分的。在这一章只是讲解了数据库和XML最简单的应用，要深入研究，还需要去学习相关专业知识。



- Qt支持哪些数据库？
- Qt数据库模型类有哪些？
- Qt中常用的处理XML的方法？

