

第 1 部分 数据库操作

Qt 提供了 QSql 模块来提供平台独立的基于 SQL 的数据库操作。这里我们所说的“平台独立”，既包括操作系统平台，有包括各个数据库平台。另外，我们强调了“基于 SQL”，因为 NoSQL 数据库至今没有一个通用查询方法，所以不可能提供一种通用的 NoSQL 数据库的操作。Qt 的数据库操作还可以很方便的与 model/view 架构进行整合。通常来说，我们对数据库的操作更多地在于对数据库表的操作，而这正是 model/view 架构的长项。

Qt 使用 QSqlDatabase 表示一个数据库连接。更底层上，Qt 使用驱动（drivers）来与不同的数据库 API 进行交互。Qt 桌面版本提供了如下几种驱动：

驱动	数据库
QDB2	IBM DB2 (7.1 或更新版本)
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Oracle Call Interface Driver
QODBC	Open Database Connectivity (ODBC) – Microsoft SQL Server 及其它兼容 ODBC 的数据库
QPSQL	PostgreSQL (7.3 或更新版本)
QSQLITE2	SQLite 2
QSQLITE	SQLite 3
QSYMSQL	针对 Symbian 平台的 SQLite 3
QTDS	Sybase Adaptive Server (自 Qt 4.7 起废除)

不过，由于受到协议的限制，Qt 开源版本并没有提供上面所有驱动的二进制版本，而仅仅以源代码的形式提供。通常，Qt 只默认搭载 QSqlite 驱动（这个驱动实际还包括 Sqlite 数据库，也就是说，如果需要使用 Sqlite 的话，只需要该驱动即可）。我们可以选择把这些驱动作为 Qt 的一部分进行编译，也可以当作插件编译。

如果习惯于使用 SQL 语句，我们可以选择 QSqlQuery 类；如果只需要使用高层次的数据库接口（不关心 SQL 语法），我们可以选择 QSqlTableModel 和 QSqlRelationalTableModel。本章我们介绍 QSqlQuery 类，在后面的章节则介绍 QSqlTableModel 和 QSqlRelationalTableModel。

在使用时，我们可以通过

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

找到系统中所有可用的数据库驱动的名字列表。我们只能使用出现在列表中的驱动。由于默认情况下，QtSql 是作为 Qt 的一个模块提供的。为了使用有关数据库的类，我们必须早 .pro 文件中添加这么一句：

```
QT += sql
```

这表示，我们的程序需要使用 Qt 的 core、gui 以及 sql 三个模块。注意，如果需要同时使用 Qt4 和 Qt5 编译程序，通常我们的 .pro 文件是这样的：

```
QT += core gui sql
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
```

这两句也很明确：Qt 需要加载 core、gui 和 sql 三个模块，如果主版本大于 4，则再添加 widgets 模块。

下面来看一个简单的函数：

```
bool connect(const QString &dbName)
{
    QSqlDatabase db = QSqlDatabase::addDatabase("SQLITE");
    // db.setHostName("host");
    // db.setDatabaseName("dbname");
    // db.setUserName("username");
    // db.setPassword("password");
    db.setDatabaseName(dbName);
    if (!db.open()) {
        QMessageBox::critical(0, QObject::tr("Database Error"),
                               db.lastError().text());
        return false;
    }
    return true;
}
```

我们使用 connect() 函数创建一个数据库连接。我们使用 QSqlDatabase::addDatabase() 静态函数完成这一请求，也就是创建了一个 QSqlDatabase 实例。注意，数据库连接使用自己的名字进行区分，而不是数据库的名字。例如，我们可以使用下面的语句：

```
QSqlDatabase db=QSqlDatabase::addDatabase("SQLITE", QString("conn%1").arg(dbName));
```

此时，我们是使用 addDatabase() 函数的第二个参数来给这个数据库连接一个名字。在这个例子中，用于区分这个数据库连接的名字是 QString("conn%1").arg(dbName)，而不是“SQLITE”。这个参数是可选的，如果不指定，系统会给出一个默认的名字 QSqlDatabase::defaultConnection，此时，Qt 会创建一个默认的连接。如果你给出的名字与已存在的名字相同，新的连接会替换掉已有的连接。通过这种设计，我们可以为一个数据库建立多个连接。

我们这里使用的是 sqlite 数据库，只需要指定数据库名字即可。如果是数据库服务器，比如 MySQL，我们还需要指定主机名、端口号、用户名和密码，这些语句使用注释进行了

简单的说明。

接下来我们调用了 QSqlDatabase::open()函数，打开这个数据库连接。通过检查 open()函数的返回值，我们可以判断数据库是不是正确打开。

QtSql 模块中的类大多具有 lastError()函数，用于检查最新出现的错误。如果你发现数据库操作有任何问题，应该使用这个函数进行错误的检查。这一点我们也在上面的代码中进行了体现。当然，这只是最简单的实现，一般来说，更好的设计是，不要在数据库操作中混杂界面代码（并且将这个 connect()函数放在一个专门的数据库操作类中）。

接下来我们可以在 main()函数中使用这个 connect()函数：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    if (connect("demo.db")) {
        QSqlQuery query;
        if (!query.exec("CREATE TABLE student ("
                        "id INTEGER PRIMARY KEY AUTOINCREMENT,"
                        "name VARCHAR,"
                        "age INT)")) {
            QMessageBox::critical(0, QObject::tr("Database Error"),
                                  query.lastError().text());
            return 1;
        }
    } else {
        return 1;
    }
    return a.exec();
}
```

main()函数中，我们调用这个 connect()函数打开数据库。如果打开成功，我们通过一个 QSqlQuery 实例执行了 SQL 语句。同样，我们使用其 lastError()函数检查了执行结果是否正确。

注意这里的 QSqlQuery 实例的创建。我们并没有指定是为哪一个数据库连接创建查询对象，此时，系统会使用默认的连接，也就是使用没有第二个参数的 addDatabase()函数创建的那个连接（其实就是名字为 QSqlDatabase::defaultConnection 的默认连接）。如果没有这么一个连接，系统就会报错。也就是说，如果没有默认连接，我们在创建 QSqlQuery 对象时必须指明是哪一个 QSqlDatabase 对象，也就是 addDatabase()的返回值。

我们还可以通过使用 QSqlQuery::isActive()函数检查语句执行正确与否。如果 QSqlQuery 对象是活动的，该函数返回 true。所谓“活动”，就是指该对象成功执行了 exec()函数，但是还没有完成。如果需要设置为不活动的，可以使用 finish()或者 clear()函数，或者直接释放掉这个 QSqlQuery 对象。这里需要注意的是，如果存在一个活动的 SELECT 语句，某些数据库

系统不能成功完成 connect()或者 rollback()函数的调用。此时，我们必须首先将活动的 SELECT 语句设置成不活动的。

创建过数据库表 student 之后，我们开始插入数据，然后将其独取出来：

```
if (connect("demo.db")) {
    QSqlQuery query;
    query.prepare("INSERT INTO student (name, age) VALUES (?, ?)");
    QVariantList names;
    names << "Tom" << "Jack" << "Jane" << "Jerry";
    query.addBindValue(names);
    QVariantList ages;
    ages << 20 << 23 << 22 << 25;
    query.addBindValue(ages);
    if (!query.execBatch()) {
        QMessageBox::critical(0, QObject::tr("Database Error"),
                               query.lastError().text());
    }
    query.finish();
    query.exec("SELECT name, age FROM student");
    while (query.next()) {
        QString name = query.value(0).toString();
        int age = query.value(1).toInt();
        qDebug() << name << ": " << age;
    }
} else {
    return 1;
}
```

依旧连接到我们创建的 demo.db 数据库。我们需要插入多条数据，此时可以使用 QSqlQuery::exec()函数一条一条插入数据，但是这里我们选择了另外一种方法：批量执行。首先，我们使用 QSqlQuery::prepare()函数对这条 SQL 语句进行预处理，问号？相当于占位符，预示着以后我们可以使用实际数据替换这些位置。简单说明一下，预处理是数据库提供的一种特性，它会将 SQL 语句进行编译，性能和安全性都要优于普通的 SQL 处理。在上面的代码中，我们使用一个字符串列表 names 替换掉第一个问号的位置，一个整型列表 ages 替换掉第二个问号的位置，利用 QSqlQuery::addBindValue()我们将实际数据绑定到这个预处理的 SQL 语句上。需要注意的是，names 和 ages 这两个列表里面的数据需要一一对应。然后我们调用 QSqlQuery::execBatch()批量执行 SQL，之后结束该对象。这样，插入操作便完成了。

另外说明一点，我们这里使用了 ODBC 风格的？占位符，同样，我们也可以使用 Oracle 风格的占位符：

```
query.prepare("INSERT INTO student (name, age) VALUES (:name, :age)");
```

此时，我们就需要使用

```
query.bindValue(":name", names);  
query.bindValue(":age", ages);
```

进行绑定。Oracle 风格的绑定最大的好处是，绑定的名字和值很清晰，与顺序无关。但是这里需要注意，bindValue()函数只能绑定一个位置。 比如

```
query.prepare("INSERT INTO test (name1, name2) VALUES (:name, :name)");  
// ...  
query.bindValue(":name", name);
```

只能绑定第一个 :name 占位符，不能绑定到第二个。

接下来我们依旧使用同一个查询对象执行一个 SELECT 语句。如果存在查询结果，QSqlQuery::next()会返回 true，直到到达结果最末，返回 false，说明遍历结束。我们利用这一点，使用 while 循环即可遍历查询结果。使用 QSqlQuery::value()函数即可按照 SELECT 语句的字段顺序获取到对应的数据库存储的数据。

对于数据库事务的操作，我们可以使用 QSqlDatabase::transaction() 开启事务，QSqlDatabase::commit() 或者 QSqlDatabase::rollback() 结束事务。使用 QSqlDatabase::database()函数则可以根据名字获取所需要的数据库连接

第 2 部分 使用模型操作数据库

前一章我们使用 SQL 语句完成了对数据库的常规操作，包括简单的 CREATE、SELECT 等语句的使用。我们也提到过，Qt 不仅提供了这种使用 SQL 语句的方式，还提供了一种基于模型的更高级的处理方式。这种基于 QSqlTableModel 的模型处理更为高级，如果对 SQL 语句不熟悉，并且不需要很多复杂的查询，这种 QSqlTableModel 模型基本可以满足一般的需求。本章我们将介绍 QSqlTableModel 的一般使用，对比 SQL 语句完成对数据库的增删改查等的操作。值得注意的是，QSqlTableModel 并不一定非得结合 QListView 或 QTableView 使用，我们完全可以用其作一般性处理。

首先我们来看看如何使用 QSqlTableModel 进行 SELECT 操作：

```
if (connect("demo.db")) {  
    QSqlTableModel model;  
    model.setTable("student");  
    model.setFilter("age > 20 and age < 25");  
    if (model.select()) {
```

```

    for (int i = 0; i < model.rowCount(); ++i) {
        QSqlRecord record = model.record(i);
        QString name = record.value("name").toString();
        int age = record.value("age").toInt();
        qDebug() << name << ": " << age;
    }
}
} else {
    return 1;
}

```

我们依旧使用了前一章的 connect() 函数。接下来我们创建了 QSqlTableModel 实例，使用 setTable() 函数设置所需要操作的表格；setFilter() 函数则是添加过滤器，也就是 WHERE 语句所需要的部分。例如上面代码中的操作实际相当于 SQL 语句

```
SELECT * FROM student WHERE age > 20 and age < 25
```

使用 QSqlTableModel::select() 函数进行操作，也就是执行了查询操作。如果查询成功，函数返回 true，由此判断是否发生了错误。如果没有错误，我们使用 record() 函数取出一行记录，该记录是以 QSqlRecord 的形式给出的，而 QSqlRecord::value() 则取出一个列的实际数据值。注意，由于 QSqlTableModel 没有提供 const_iterator 遍历器，因此不能使用 foreach 宏进行遍历。

另外需要注意，由于 QSqlTableModel 只是一种高级操作，肯定没有实际 SQL 语句方便。具体来说，我们使用 QSqlTableModel 只能进行 SELECT * 的查询，不能只查询其中某些列的数据。

下面一段代码则显示了如何使用 QSqlTableModel 进行插入操作：

```

QSqlTableModel model;
model.setTable("student");
int row = 0;
model.insertRows(row, 1);
model.setData(model.index(row, 1), "Cheng");
model.setData(model.index(row, 2), 24);

```

```
model.submitAll();
```

插入也很简单：model.insertRows(row, 1);说明我们想在索引 0 的位置插入 1 行新的数据。使用 setData()函数则开始准备实际需要插入的数据。注意这里我们向 row 的第一个位置写入 Cheng（通过 model.index(row, 1)，回忆一下，我们把 model 当作一个二维表，这个坐标相当于第 row 行第 1 列），其余以此类推。最后，调用 submitAll()函数提交所有修改。这里执行的操作可以用如下 SQL 表示：

```
INSERT INTO student (name, age) VALUES ('Cheng', 24)
```

当我们取出了已经存在的数据后，对其进行修改，然后重新写入数据库，即完成了一次更新操作：

```
QSqlTableModel model;
model.setTable("student");
model.setFilter("age = 25");
if (model.select()) {
    if (model.rowCount() == 1) {
        QSqlRecord record = model.record(0);
        record.setValue("age", 26);
        model.setRecord(0, record);
        model.submitAll();
    }
}
```

这段代码中，我们首先找到 age = 25 的记录，然后将 age 重新设置为 26，存入相同的位置（在这里都是索引 0 的位置），提交之后完成一次更新。当然，我们也可以类似其它模型一样的设置方式：setData()函数。具体代码片段如下：

```
if (model.select()) {
    if (model.rowCount() == 1) {
        model.setData(model.index(0, 2), 26);
        model.submitAll();
    }
}
```

注意我们的 age 列是第 3 列，索引值为 2，因为前面还有 id 和 name 两列。这里的更新操作则可以用如下 SQL 表示：

```
UPDATE student SET age = 26 WHERE age = 25
```

删除操作同更新类似：

```
QSqlTableModel model;
model.setTable("student");
model.setFilter("age = 25");
if (model.select()) {
    if (model.rowCount() == 1) {
        model.removeRows(0, 1);
        model.submitAll();
    }
}
```

如果使用 SQL 则是：

```
DELETE FROM student WHERE age = 25
```

当我们看到 `removeRows()` 函数就应该想到：我们可以一次删除多行。事实也正是如此，这里不再赘述。

第 3 部分 可视化显示数据库数据

前面我们用了两个章节介绍了 Qt 提供的两种操作数据库的方法。显然，使用 `QSqlQuery` 的方式更灵活，功能更强大，而使用 `QSqlTableModel` 则更简单，更方便与 `model/view` 结合使用（数据库应用很大一部分就是以表格形式显示出来，这正是 `model/view` 的强项）。本章我们简单介绍使用 `QSqlTableModel` 显示数据的方法。当然，我们也可以选择使用 `QSqlQuery` 获取数据，然后交给 `view` 显示，而这需要自己给 `model` 提供数据。鉴于我们前面已经详细介绍过如何使用自定义 `model` 以及如何使用 `QTableWidget`，所以我们这里不再详细说明这一方法。

我们还是使用前面一直在用的 `student` 表，直接来看代码：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    if (connect("demo.db")) {

        QSqlTableModel *model = new QSqlTableModel;

        model->setTable("student");

        model->setSort(1, Qt::AscendingOrder);
```



```

model->setHeaderData(1, Qt::Horizontal, "Name");
model->setHeaderData(2, Qt::Horizontal, "Age");
model->select();

QTableView *view = new QTableView;
view->setModel(model);
view->setSelectionMode(QAbstractItemView::SingleSelection);
view->setSelectionBehavior(QAbstractItemView::SelectRows);
// view->setColumnHidden(0, true);
view->resizeColumnsToContents();
view->setEditTriggers(QAbstractItemView::NoEditTriggers);

QHeaderView *header = view->horizontalHeader();
header->setStretchLastSection(true);

view->show();
} else {
    return 1;
}
return a.exec();
}

```

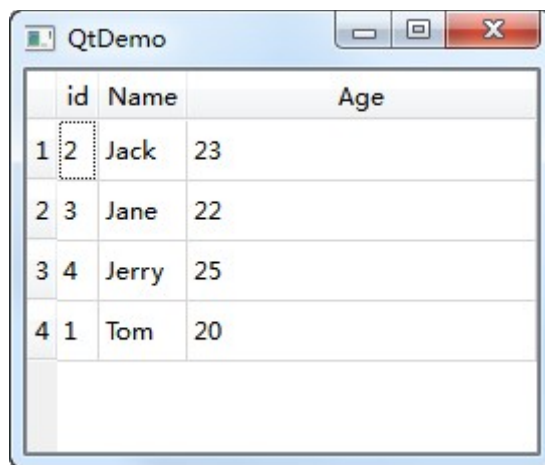
这里的 connect()函数还是我们前面使用过的，我们主要关注剩下的代码。

正如前一章的代码所示，我们在 main()函数中创建了 QSqlTableModel 对象，使用 student 表。student 表有三列：id，name 和 age，我们选择按照 name 排序，使用 setSort()函数达到这一目的。然后我们设置每一列的列头。这里我们只使用了后两列，第一列没有设置，所以依旧显示为列名 id。

在设置好 model 之后，我们又创建了 QTableView 对象作为视图。注意这里的设置：单行选择，按行选择。resizeColumnsToContents()说明每列宽度适配其内容；setEditTriggers()则禁用编辑功能。最后，我们设置最后一列要充满整个窗口。我们的代码中有一行注释，设置

第一列不显示。由于我们使用了 QSqlTableModel 方式，不能按列查看，所以我们在视图级别上面做文章：将不想显示的列隐藏掉。

接下来运行代码即可看到效果：



	id	Name	Age
1	2	Jack	23
2	3	Jane	22
3	4	Jerry	25
4	1	Tom	20

如果看到代码中很多“魔术数字”，更好的方法是，使用一个枚举将这些魔术数字隐藏掉，这也是一种推荐的方式：

```
enum ColumnIndex
{
    Column_ID = 0,
    Column_Name = 1,
    Column_Age = 2
};

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    if (connect("demo.db")) {
        QSqlTableModel *model = new QSqlTableModel;
        model->setTable("student");
        model->setSort(Column_Name, Qt::AscendingOrder);
        model->setHeaderData(Column_Name, Qt::Horizontal, "Name");
        model->setHeaderData(Column_Age, Qt::Horizontal, "Age");
    }
}
```

```

model->select();

QTableView *view = new QTableView;
view->setModel(model);
view->setSelectionMode(QAbstractItemView::SingleSelection);
view->setSelectionBehavior(QAbstractItemView::SelectRows);
view->setColumnHidden(Column_ID, true);
view->resizeColumnsToContents();
view->setEditTriggers(QAbstractItemView::NoEditTriggers);

QHeaderView *header = view->horizontalHeader();
header->setStretchLastSection(true);

view->show();
} else {
    return 1;
}
return a.exec();
}

```

第 4 部分 编辑数据库外键

前面几章我们介绍了如何对数据库进行操作以及如何使用图形界面展示数据库数据。本章我们将介绍如何对数据库的数据进行编辑。当然，我们可以选择直接使用 SQL 语句进行更新，这一点同前面所说的 model/view 的编辑没有什么区别。除此之外，Qt 还为图形界面提供了更方便的展示并编辑的功能。

普通数据的编辑很简单，这里不再赘述。不过，我们通常会遇到多个表之间存在关联的情况。首先我们要提供一个 city 表：

```

CREATE TABLE city (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name VARCHAR);

```

```
INSERT INTO city (name) VALUES ('Beijing');
INSERT INTO city (name) VALUES ('Shanghai');
INSERT INTO city (name) VALUES ('Nanjing');
INSERT INTO city (name) VALUES ('Tianjin');
INSERT INTO city (name) VALUES ('Wuhan');
INSERT INTO city (name) VALUES ('Hangzhou');
INSERT INTO city (name) VALUES ('Suzhou');
INSERT INTO city (name) VALUES ('Guangzhou');
```

由于 city 表是一个参数表，所以我们直接将所需要的城市名称直接插入到表中。接下来我们创建 student 表，并且使用外键连接 city 表：

```
CREATE TABLE student (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR,
    age INTEGER,
    address INTEGER,
    FOREIGN KEY(address) REFERENCES city(id));
```

我们重新创建 student 表（如果你使用的 RDBMS 支持 ALTER TABLE 语句直接修改表结构，就不需要重新创建了；否则的话只能先删除旧的表，再创建新的表，例如 sqlite）。

这里需要注意一点，如果此时我们在 Qt 中直接使用

```
INSERT INTO student (name, age, address) VALUES ('Tom', 24, 100);
```

语句，尽管我们的 city 中没有 ID 为 100 的记录，但还是可以成功插入的。这是因为虽然 Qt 中的 sqlite 使用的是支持外键的 sqlite3，但 Qt 将外键屏蔽掉了。为了启用外键，我们需要首先使用 QSqlQuery 执行：

```
PRAGMA foreign_keys = ON;
```

然后就会发现这条语句不能成功插入了。接下来我们插入一些正常的数据：

```
INSERT INTO student (name, age, address) VALUES ('Tom', 20, 2);
INSERT INTO student (name, age, address) VALUES ('Jack', 23, 1);
INSERT INTO student (name, age, address) VALUES ('Jane', 22, 4);
INSERT INTO student (name, age, address) VALUES ('Jerry', 25, 5);
```

下面，我们使用 model/view 方式来显示数据：

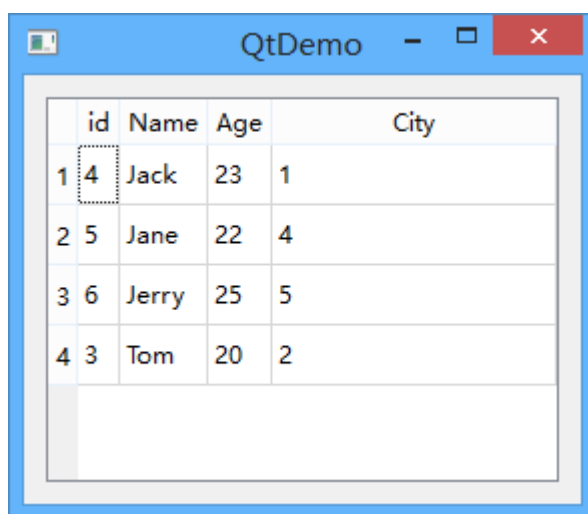
```
QSqlTableModel *model = new QSqlTableModel(this);
model->setTable("student");
model->setSort(ColumnID_Name, Qt::AscendingOrder);
model->setHeaderData(ColumnID_Name, Qt::Horizontal, "Name");
```

```
model->setHeaderData(ColumnID_Age, Qt::Horizontal, "Age");
model->setHeaderData(ColumnID_City, Qt::Horizontal, "City");
model->select();
```

```
QTableView *view = new QTableView(this);
view->setModel(model);
view->setSelectionMode(QAbstractItemView::SingleSelection);
view->setSelectionBehavior(QAbstractItemView::SelectRows);
view->resizeColumnsToContents();
```

```
QHeaderView *header = view->horizontalHeader();
header->setStretchLastSection(true);
```

这段代码和我们前面见到的没有什么区别。我们可以将其补充完整后运行一下看看：



	id	Name	Age	City
1	4	Jack	23	1
2	5	Jane	22	4
3	6	Jerry	25	5
4	3	Tom	20	2

注意外键部分：City 一列仅显示出了我们保存的外键。如果我们使用 QSqlQuery，这些都不是问题，我们可以将外键信息放在一个 SQL 语句中 SELECT 出来。但是，我们不想使用 QSqlQuery，那么现在可以使用另外的一个模型：

QSqlRelationalTableModel。QSqlRelationalTableModel 与 QSqlTableModel 十分类似，可以为一个数据库表提供可编辑的数据模型，同时带有外键的支持。下面我们修改一下我们的代码：

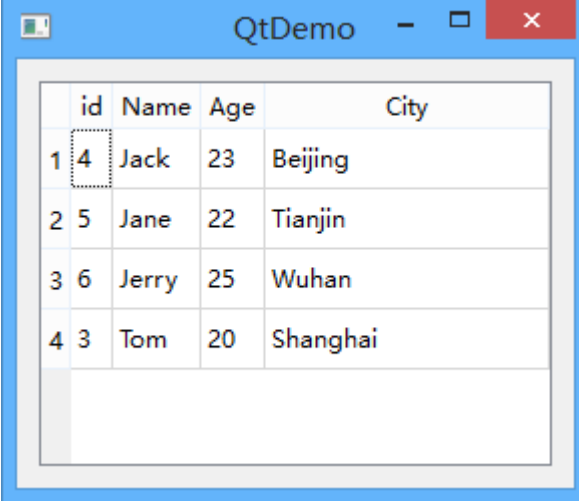
```
QSqlRelationalTableModel *model = new QSqlRelationalTableModel(this);
model->setTable("student");
model->setSort(ColumnID_Name, Qt::AscendingOrder);
model->setHeaderData(ColumnID_Name, Qt::Horizontal, "Name");
model->setHeaderData(ColumnID_Age, Qt::Horizontal, "Age");
model->setHeaderData(ColumnID_City, Qt::Horizontal, "City");
model->setRelation(ColumnID_City, QSqlRelation("city", "id", "name"));
model->select();
```

```
QTableView *view = new QTableView(this);
view->setModel(model);
view->setSelectionMode(QAbstractItemView::SingleSelection);
view->setSelectionBehavior(QAbstractItemView::SelectRows);
```

```
view->resizeColumnsToContents();
view->setItemDelegate(new QSqlRelationalDelegate(view));

QHeaderView *header = view->horizontalHeader();
header->setStretchLastSection(true);
```

这段代码同前面的几乎一样。我们首先创建一个 QSqlRelationalTableModel 对象。注意，这里我们有一个 setRelation() 函数的调用。该语句说明，我们将第 ColumnID_City 列作为外键，参照于 city 表的 id 字段，使用 name 进行显示。另外的 setItemDelegate() 语句则提供了一种用于编辑外键的方式。运行一下程序看看效果：



	id	Name	Age	City
1	4	Jack	23	Beijing
2	5	Jane	22	Tianjin
3	6	Jerry	25	Wuhan
4	3	Tom	20	Shanghai

此时，我们的外键列已经显示为 city 表的 name 字段的实际值。同时在编辑时，系统会自动成为一个 QComboBox 供我们选择。当然，我们需要自己将选择的外键值保存到实际记录中，这部分我们前面已经有所了解。