目录

22. 数据分析 - 唐代诗人的朋友圈 1
22.1 创建程序框架 2
22.2 数据整理与准备 4
22.2.1 sqlite 数据库 4
22.2.2 数据库连接 6
22.2.3 全唐诗数据整理 8
22.2.4 诗人名录及别名 14
22.2.5 引用关系 16
22.3 构建诗人关系网络 17
22.3.1 创建 Poem 及 Poet 类 18
22.3.2 进度条对话框 22
22.3.3 关系网络构建线程 23
22.3.4 创建线程对象并执行 29
22.4 引用查询 33
22.4.1 创建 Reference 类 33
22.4.2 引用关系的超文本展现 38
22.5 朋友圈 41
22.6 可视化关系网络 43

22. 数据分析 - 唐代诗人的朋友圈

“李杜文章在，光焰万丈长"，唐诗无疑是中国古代文学最灿烂的篇章之一。现代人发表论文，会互相引用，喝酒吃饭，也经常会谈及谁谁谁是我哥们。作为当时最重要的文学形式，唐代的诗人也经常会在诗文中提及自己的好朋友。杜甫比李白小十一岁，二者相识于杜甫父亲杜闲家中，彼时正是李白因触怒权贵放归山林之时。两人一见，杜秒变小迷弟。杜在《与李十二白同寻范十隐居》中描绘了两人的亲密关系："余亦东蒙客，怜君如兄弟。醉眠秋共被，携手日同行"。不仅如此，在两人各奔东西后，杜甫压抑不住对李白的思念，写了多首提及李白的诗。例如《梦李白》中云："三夜频梦君，情亲见君意"。能连续三个晚上做梦都梦到李白，可见交情不浅。

通过分析全唐诗中各位诗人之间的“引用”关系，可以描绘出当时诗坛的大致朋友圈图景：谁跟谁熟？ 谁是圈子里的带头大哥？ 全唐诗有 4 万多首，人工一首一首地筛查费时费力，这种重复的统计性质的工作正是计算机最擅长的。

本章的代码和数据整合在一个名为 C22_PoetsNetwork 的文件夹中。注意，本章节所依赖的全唐诗文本以及《中国历代人物传记资料库》使用了繁体中文，所以读者在运行代码查询时，如果使用简体中文输入诗人姓名，结果将与预期不符。

本章内容受开源项目 poetry_analyzer 的启发。为方便读者理解，作者整理了相关数据并重写了代码。

22.1 创建程序框架

请读者按照 21.1 及 21.3 节所介绍的方法，在 Qt Creator 中创建一个名为 PoetsNetwork 的项目，该项目的主窗口如图 22-1 所示，作者在该图上人工标注了各关键部件的名称。表 22-1 列出了该项目创建过程中的一些注意事项，以及其主窗口中部分部件的用途。

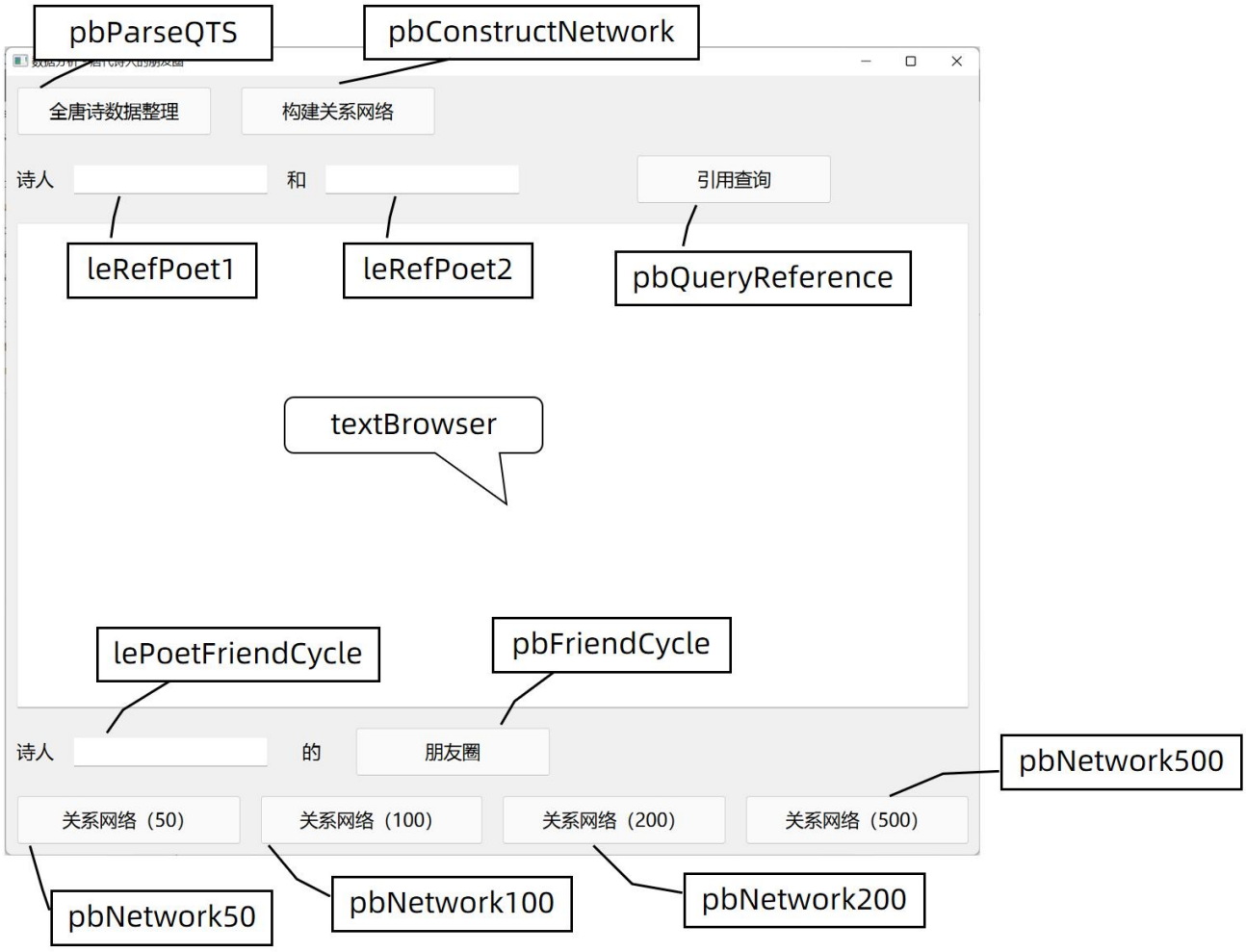


图 22-1 唐代诗人的朋友圈主窗口

表 22-1 创建 PoetsNetwork 项目的注意事项及主窗口的主要部件

名目	说明
项目模板	Application(Qt)/Qt Widgets Application (Qt 窗体应用程序)。
主窗口	类名：MainWidget；基类：QWidget。

pbParseQTS	类型：QPushButton；文字：全唐诗数据整理；用途：将全唐诗文本数据整理并写入到数据库。
pbConstructNetwork	类型：QPushButton；文字：构建关系网络；用途：对数据库中的全唐诗文本正行分析，找出诗人与诗人之间的引用关系并写入数据库。
leRefPoet1,leRefPoet2	类型：QLineEdit，单行输入框；用途：“引用查询”时输入第一/二个诗人的姓名。
pbQueryReference	类型：QPushButton；文字：引用查询；用途：查询两个诗人之间的引用数量以及相互之间的引用诗文。
textBrowser	类型：QTextBrowser，简单的文字型 HTML 浏览器；用途：显示程序的部分执行结果。
lePoetFriendCycle	类型：QLineEdit；用途：查询“朋友圈”时输入诗人的姓名。
pbFriendCycle	类型：QPushButton；用途：查询单个诗人的朋友圈，即找出与指定诗人存在引用关系的全部诗人，生成关系网络图，并用浏览器显示。
pbNetWork50/100/200/500	类型：QPushButton；用途：导出引用数量的前 50/100/200/500 行引用关系，将由相关诗人姓名及引用箭头组成的关系网络生成出来，并用浏览器显示。

图 22-2 则展示 MainWidget 主窗口的对象结构以及窗口组件之间的布局关系。如图所示，主窗口自身呈现竖向布局，其内包含 4 个横向布局以及 textBrowser。为了构造出期望的界面效果，读者可能需要：(1) 调整布局的 layoutSpacing（布局间隔）；(2) 修改按钮、单行输入框的 minimumSize（最小尺寸）。此外，在图 22-2 中，我们还看到了类型为 Spacer 的部件，这种类型的部件仅用于占据布局空间，其在最终的结果页面上不会有任何显示，它可以把别的部件“挤”到期望的位置。

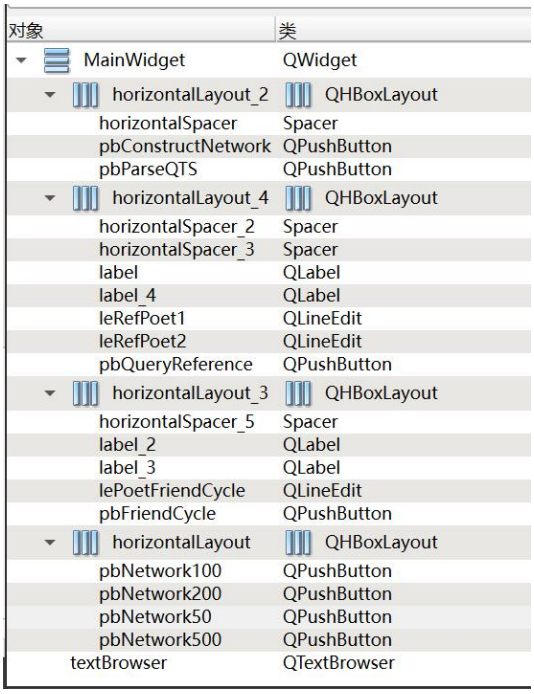


图 22-2 MainWidget 主窗口的对象结构

22.2 数据整理与准备

22.2.1 sqlite 数据库

为了便于统计分析以及向读者简单介绍数据库的入门知识和 C++访问数据库的方法，本章使用 sqlite 数据库来存储相关数据。

对于结构化的数据，如个人的身份信息、银行的交易流水、图书馆的借还记录等，通常都存储在数据库系统中。数据库系统通常运行在一个服务器或者由多个服务器构成的集群中，软件使用者的计算机或者终端直接或者间接地透过 TCP/IP 访问数据库、查询或存储数据。大型的数据库系统软件有阿里蚂蚁金服的 OceanBase、华为的 GaussDB、开源的 MySql 以及私有的 Oracle。

本章使用的 sqlite 是一个超级 mini 版的数据库系统，它本质上是一个运行于软件内部的 C 语言包。在本章的代码中，数据库的存储文件为 C22_PoetsNetwork/data 子目录下的 data.db。

注意　本章只能概要地介绍数据库系统的相关知识。对数据库系统的全面讨论是数据库系统课程的任务，读者如果对本实践所应用到的数据库技术感到疑惑，请查阅数据库系统课程的教材。

二维码 操作指南 SQLiteStudio 的下载与安装 http://codelearn.club/2022/04/installsqlitestudio/

为了便于查询数据库中的数据，请读者按照二维码链接所提供的方法，下载并安装一个名为 SQLiteStudio 的软件，SQLiteStudio 是遵从 GPL 协议的开源软件，它可以帮助我们创建、编辑和查询 sqlite 数据库。作者安装时，其版本号为 3.3.3。

如图 22-3 所示，运行 SQLiteStudio，选择 Database/Add a database（数据库/添加一个数据库）菜单项，将得到如图 22-4 所示的对话框。在该对话框中，将数据类型选择为 SQLite 3，点击浏览按钮（）定位到项目目录中已存在的 data.db 文件，此时，name(名称)被自动调整为 data，点击“测试连接”，在测试通过后点击“OK”按钮即可打开本实践的数据库文件（data.db）。

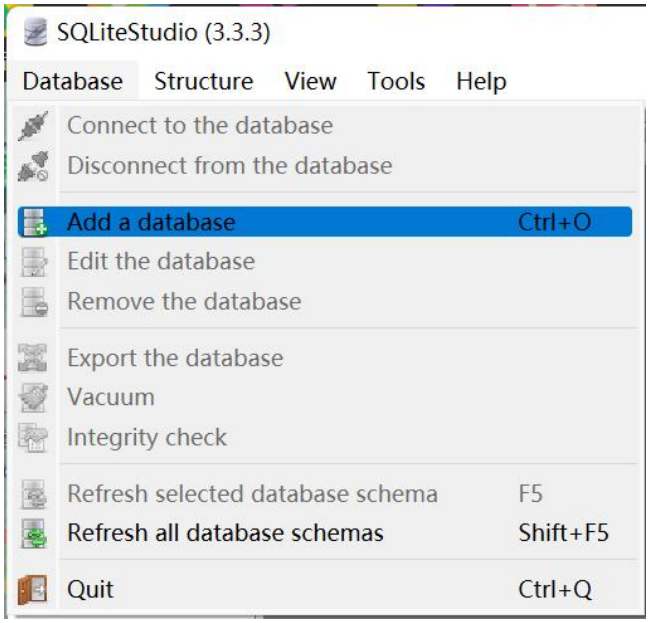


图 22-3 添加数据库

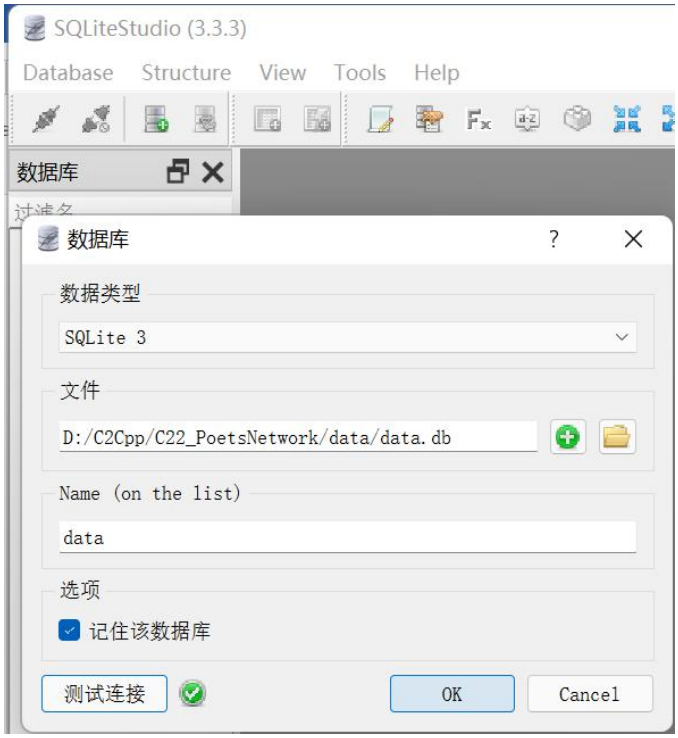


图 22-4 选择 data.db

数据库文件打开后，名为 data 数据库将显示在软件的左侧列表中，如图 22-5 所示。双击 data 数据库将其逐步展开，可以看到该数据库有 4 个表格，名称分别是 altname（别名）、peom(诗)、poet(诗人)和 reference(引用)。

选择 peom 表，可以看到表格中的数据，共有 42948 行，每一行存储了一首唐诗。这些数据来自于“全唐诗”，可以看到，排在最前面的是唐太宗李世民的诗。

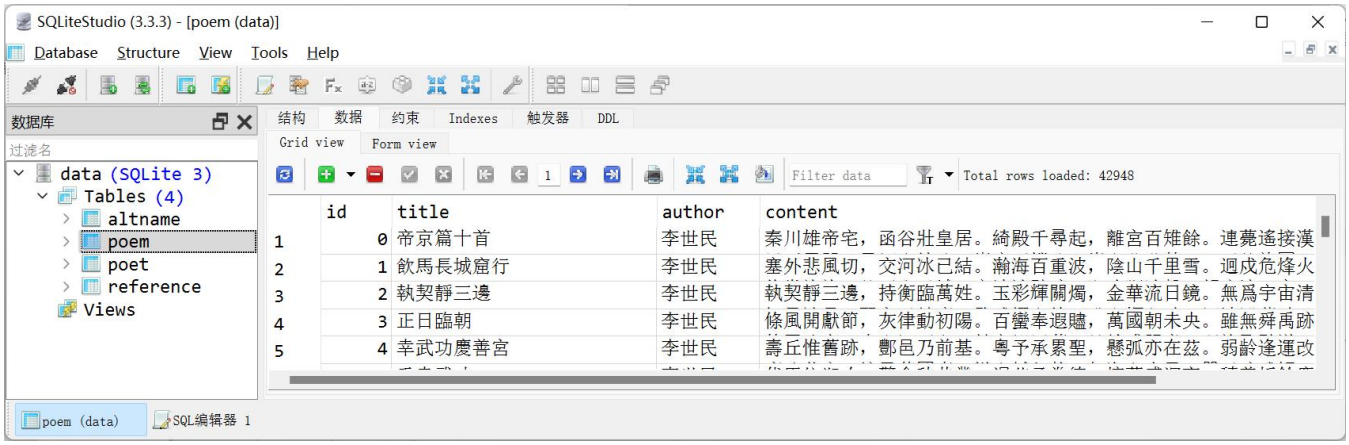


图 22-5 peom 表格中的数据

数据库中的表（Table）都是二维的，每一行称为一条记录（Record），在 poem 表中，一条记录存储一首唐诗。每一行又可以分为多列（Column），在数据库中，列也称为字段（Field）。peom 表的结构如表 22-2 所示。

表 22-2 诗人表 - poem

字段名	类型	用途
id	int	用一个数字来表示每首唐诗在表中的唯一编号,该字段不可重复
title	text	题名字符串
author	text	作者姓名
content	text	全文

text 是 sqlite 数据库中使用的字符串类型名，读者可以认为它就是 Qt 中的 QString 字符串类型。

22.2.2 数据库连接

应用程序与数据库之间的关系通常是客户机/服务器模式，读者可以把数据库想象成一个服务器，而应用程序则是客户机，数据库服务器通过网络向客户机提供数据存储和查询服务。当然，对于 sqlite 这种嵌入式数据库而言，数据库服务器并不真正存在，通信也不依赖于网络。

二维码：操作指南 创建并添加新类

http://codelearn.club/2021/03/qtnewclass/

为了访问数据库，需要建立数据库连接。按照上述二维码链接提供的方法，创建一个名为 DBHelper 的类并加入到项目 PoetsNetwork 中。

其中，头文件 dbhelper.h 的代码如下：

```
1  //dbhelper.h
2  #ifndef DBHELPER_H
3  #define DBHELPER_H
4  #include <QSqlDatabase>
5
6  class DBHelper {
7  public:
8      static QSqlDatabase db;
9      static bool openDatabase();
10     static void closeDatabase();
11     static QString sProjectPath;
12 };
13
14 #endif // DBHELPER_H
```

类 DBHelper 是一个所谓的“帮助类”，其数据成员和方法都是静态的，这意味着我们可以在不实例化 DBHelper 对象的前提下使用这些属性和方法。

第 8 行：QSqlDatabase 类型的对象 db 用于保存“数据库连接”。

第 9 行：openDatabase()用于“打开”/建立数据库连接，如果打开失败，则返回假。

第 10 行：closeDatabase()用于关闭数据库连接。

第 11 行：sProjectPath 用于存储程序的“工作目录”，整个应用程序依赖于该目录定位所有的数据文件，包括 sqlite 数据库文件。

代码文件 dbhelper.cpp 的内容如下：

```
1  //dbhelper.cpp
2  #include "dbhelper.h"
3  #include <QFile>
4  #include <QDebug>
5
6  QSqlDatabase DBHelper::db;
7
8  bool DBHelper::openDatabase() {
9      QString sDBFile = sProjectPath + "/data/data.db";
10     if (!(QFile::exists(sDBFile))) {
11         qDebug() << "Error: missing database file " << sDBFile;
12         return false;
13     }
14
15     db = QSqlDatabase::addDatabase("QSQLITE");
16     db.setDatabaseName(sDBFile);
17     if (db.open())
18         qDebug() << "Database opened successfully:" << sDBFile;
19     else
20         qDebug() << "Database open failed:" << sDBFile;
21
22     return db.isOpen();
23 }
24
25 void DBHelper::closeDatabase(){
26     if (db.isOpen())
27         db.close();
28 }
29
30 QString DBHelper::sProjectPath = "d:/C2Cpp/C22_PoetsNetwork";
```

第 30 行：如本书 14.5 节所述，类的静态数据成员必须在 cpp 文件中定义和初始化。这里的“d:/C2Cpp/C22_PoetsNetwork”是作者计算机上的项目工作目录，读者应根据自己计算机上的情况对该目录进行修改。在这个目录下，应有 data 和 html 两个子目录，其中，data 子目录下存储了全唐诗文本文件及 sqlite 数据库文件，html 子目录内则储存了后期用于关系网络可视化用的 JavaScript 及 html 文本文件。

第 6 行：对 db 静态数据成员进行定义。

第 8 ~ 23 行：openDatabase()的函数定义。其中，第 9 行借助于 sProjectPath 生成了数据库文件 data.db 的绝对路径，第 10 行则通过 QFile::exists()函数判定数据库文件是否存在，如果不存在，在向调试控制台输出错误信息后，返回 false。

第 25 ~ 28 行：closeDataBase()函数定义。如果当前数据库 db 处于打开状态，通过 close 成员函数关闭它。

> ⚠ 警告
>
> 本实践的程序运行时会打开和访问 sqlite 数据库文件，前一小节中提到的 SQLiteStudio 软件也需要打开和访问同一个数据库文件。读者必须避免两者同时运行，即读者试图构建并运行 PoetsNetwork 项目时，需要先行退出 SQLiteStudio 软件，否则程序运行可能出错。

22.2.3 全唐诗数据整理

在子目录 data 下有一个名为 qts_zht.txt 的文本文件，共收录唐诗 4 万多首。格式如下：

```
...
128_65      哭孟浩然 王維      故人不可見，漢水日東流。借問襄陽老，江山空蔡州。
...
```

可以看见，文本文件的每一行是一首唐诗，用空格（部分为\t 制表符）可分为四个部分，从前至后分别是编号、题名、作者和全文。为了便于后续的数据分析，作者使用下述程序将全唐诗导入 sqlite 数据库中的 poem 表 。

```
1  //mainwidget.h...
2  private:
3      int parsePoemsIntoDatabase();  //将全唐诗整理入库，返回成功整理入库的唐诗的数量
```

第 3 行：在 mainwidget.h 中，parsePoemsIntoDatabase()函数被定义为私有成员，因为该函数预期仅用于 MainWidget 内部。

```
1  //mainwidget.cpp ...
2  int MainWidget::parsePoemsIntoDatabase() {
3      QString sFile = DBHelper::sProjectPath + "/data/qts_zht.txt";
4      if (!(QFile::exists(sFile))) {
5          qDebug() << "Error: missing QTS file " << sFile;
6          return -1;
7      }
8
9      auto q = QSqlQuery();
10     q.prepare("DROP TABLE IF EXISTS poem; ");
11     if (!q.exec()){
12         qDebug() << q.lastError();
13         return -1;
14     }
15
16     q.clear();
17     q.prepare("CREATE TABLE poem(id INT PRIMARY KEY ASC, "
18               "title TEXT, author TEXT, content TEXT);");
```

```
    if (!q.exec()){
        qDebug() << q.lastError();
        return -1;
    }

    auto idx = 0;
    QVariantList ids, titles, authors, contents;

    QFile f(sFile);
    Q_ASSERT(f.open(QIODevice::ReadOnly));
    QTextStream fs(&f);
    fs.setEncoding(QStringConverter::Utf8);
    while (!fs.atEnd()){
        auto line = fs.readLine();
        line.replace("\t"," ");
        auto r = line.split(" ");
        if (r.size()==4){
            ids << idx++; titles << r[1]; authors << r[2]; contents << r[3];
        }
    }
    f.close();

    q.clear();
    q.prepare("INSERT INTO poem VALUES (?,?,?,?)");
    q.addBindValue(ids);     q.addBindValue(titles);
    q.addBindValue(authors); q.addBindValue(contents);

    DBHelper::db.transaction();    //开始数据库事务
    if (!q.execBatch()){
        qDebug() << q.lastError();
        DBHelper::db.rollback();   //回滚数据库事务
        return -1;
    }
    DBHelper::db.commit();         //提交数据库事务

    return int(ids.size());
}
```

这很可能是读者第一次接触数据库的相关应用，我们先解释代码中的 SQL 语名。SQL 是 Structured Query Language 的首字母拼写，是专门应用于关系数据库数据查询和操纵的“语言”。与函数 parsePoemsIntoDatabase()相关的 SQL 语句解析请见表 22-3。

表 22-3 SQL 语句解析 1

语句	DROP TABLE IF EXISTS poem;
说明	如果 poem 表存在，将其从数据库中删除（包括结构和数据）。当一个字符串中存在多条 SQL 语句时，语句之间使用分号作分隔。
语句	CREATE TABLE poem(id INT PRIMARY KEY ASC, title TEXT, author TEXT, content TEXT);

说明	创建 poem 表，id 字段的类型为 int，title、author、content 字段的类型为 text。请注意，id 字段同时也被指定为表的主键（PRIMARY KEY），这意味着在这个表里面，每一行的 id 字段的值不可以重复。对于关系数据库而言，为每个表指定一个主键通常是必要的。ASC 是 ASCENDING 的缩写，表示升序排列，这里就是指 poem 表内的各行依其主键键值在表中升序排列。
语句	INSERT INTO poem VALUES (?,?,?,?)
说明	用于向表格 poem 中插入一行数据，即一首唐诗。?在这里的作用类似于替换符，相关代码会将?号用实际数据替换，形成下述完整的 SQL 语句：INSERT INTO poem VALUES (0, '峨眉山月歌','李白','峨眉山月半轮秋,影入平羌江水流。...')

```
 3      QString sFile = DBHelper::sProjectPath + "/data/qts_zht.txt";
 4      if (!(QFile::exists(sFile))) {
 5          qDebug() << "Error: missing QTS file " << sFile;
 6          return -1;
 7      }
```

第 3 ~ 7 行：通过 DBHelper::sProjectPath 生成全唐诗文件文件的绝对路径 sFile。再通过 QFile::exists()函数判断该文件是否存在，如不存在，报错后返回-1。

```
 9      auto q = QSqlQuery();
10      q.prepare("DROP TABLE IF EXISTS poem; ");
11      if (!q.exec()){
12          qDebug() << q.lastError();
13          return -1;
14      }
```

第 9 ~ 14 行：执行 SQL 语句删除 poem 表（如果存在）。QSqlQuery 是 Qt 中的数据库查询对象，专门用于执行 SQL 语句。如果 q.exec()返回错误值，则报错并返回-1。在 QSqlQuery 类型的构造函数中，可以指定该查询对象所使用的数据库连接，如果没有指定，则使用默认数据库连接。由于本应用程序仅连接一个数据库，故总是使用默认数据库连接。

第 16 ~ 22 行：使用类似于第 9 ~ 14 行的代码新建 poem 表格。读者可能会好奇为什么要重建 poem 表，而不是简单清空其表内数据，作者经过测试，确认相较于删除表内数据，直接重建表的速度更快。

```
24      auto idx = 0;
25      QVariantList ids, titles, authors, contents;
26
27      QFile f(sFile);
28      Q_ASSERT(f.open(QIODevice::ReadOnly));
29      QTextStream fs(&f);
30      fs.setEncoding(QStringConverter::Utf8);
31      while (!fs.atEnd()){
32          auto line = fs.readLine();
33          line.replace("\t"," ");
34          auto r = line.split(" ");
```

```
35          if (r.size()==4){
36              ids << idx++; titles << r[1]; authors << r[2]; contents << r[3];
37          }
38      }
39      f.close();
```

第 24 ~ 39 行：使用 QFile 打开全唐诗文本文件，逐行读取其中的唐诗，将题名、作者、内容分拆后，连同顺序号，按顺序装入 QVariantList 类型的容器 ids、titles、authors 和 contents，为一次性地向数据库写入全部唐诗数据做好准备。

第 25 行：QVariantList 是用于存储 QVariant 类型的容器。在 Qt 中，QVariant 代表“不确定类型”的对象，该型对象可以存储 int、QString 以及任意其它类型的对象。

第 30 行：我们使用 utf-8 格式来解析文件文件，utf-8 是一种支持多国文字共存的文字编码方案。读者应该知道，任何一个文件本质上都是字节流，而文本文件中的字节流，预期应该被“解读”为文字，应该有确定的编码方案将其中的字节流与文字字符进行相互映射。

第 33 行：将行字符串 line 中的制表符\t 替换成空格。

第 34 行：line.split(" ")函数以空格作为分隔符，将字符串分成多个子串，r 的类型为 QStringList，是一个存储 QString 对象的容器。

```
41      q.clear();
42      q.prepare("INSERT INTO poem VALUES (?,?,?,?)");
43      q.addBindValue(ids);     q.addBindValue(titles);
44      q.addBindValue(authors); q.addBindValue(contents);
45
46      DBHelper::db.transaction();    //开始数据库事务
47      if (!q.execBatch()){
48          qDebug() << q.lastError();
49          DBHelper::db.rollback();   //回滚数据库事务
50          return -1;
51      }
52      DBHelper::db.commit();         //提交数据库事务
```

第 41 ~ 52 行：通过批量执行 SQL 语句将准备好的唐诗数据插入数据库的 poem 表。

第 43 ~ 44 行：将 ids、titles 等准备好的 QVariantList 绑定到 SQL 查询对象 q 上。

第 47 行：q.execBatch()按照第 42 行所设定的 SQL 语句模板，结合存储于 ids、titles、authors 及 contents 容量内的数据，批量生成并执行 SQL 语句，将所有数据一次性地写入数据库。

第 46、52 行：为了提交数据写入的效率，我们采用了数据库的事务（transaction）管理技术。第 46 行开始数据库事务，第 52 行提交数据库事务。只有事务提交后，相关的数据库修改才会被确认并写入硬盘文件。想象一笔银行转账交易，如果细分下来，其实包含了多处数据修改：包括减少转出账户的余额、增加转入账户的余额、添加流水日志等。上述多处修改，如果部分成功，部分失败（因断电或系统故障等导致），数据库里的数据就会出现不一致的情况。为了避免一个完整操作的细分动作部分成功，部分失败，数据库系统通常提供事务管理功能，只有在事务提交时，之前的一系列数据修改动作才被确认。在本例当中，应用事务管理技术的目的更多是为了避免数据库文件的多次重写。如果不使用事务管理，每执行一条 SQL 语句，数据库文件就可能需要写入一次，效率十分低下。

第 49 行：如果 SQL 语句的执行发生错误，回滚数据库事务。所谓回滚，就是把事务开始之后的全部操作作废，以避免一系列完整的数据库操作部分成功，部分失败。

按照 21.3.6 节所介绍的方法，我们为 MainWidget 里的 pbParseQTS 按钮的“released()”信号添加如下内容的槽函数。

```
1 //mainwidget.cpp
2 void MainWidget::on_pbParseQTS_released(){
3     auto r = parsePoemsIntoDatabase();
4     ui->textBrowser->setHtml(r<0?QString("解析全唐诗文本并导入数据库失败!")
5         :QString("%1 首全唐诗被解析并存入数据库 peom 表。").arg(r));
6 }
```

第 3 行：调用执行 parsePoemsIntoDatabase()函数。

第 4 行：如果返回的 r 小于 0，说明操作失败，大于等于 0 则表示解析成功的唐诗数量。textBrowser 是简单的 HTML 文本浏览器，在本例中用于显示部分程序执行结果。

第 5 行：arg()是 QString 对象的成员函数，本例中，它用参数值替换掉字符串中的占位符 1%，返回一个完成格式化的新字符串。

构建并执行 PoemsNetwork 程序，并点击“全唐诗整理”按钮，一切无误的话，将得到如图 22-6 的执行结果。该执行结果显示，共有 42948 首唐诗被整理入库。再次提醒，执行 PoemsNetwork 程序前，应退出 SQLiteStudio，以避免两个程序同时访问数据库文件 data.db。

图 22-6 全唐诗数据整理执行结果

在退出 PoetsNetwork 程序的执行后，读者可以使用 SQLiteStudio 查看全唐诗数据整理的成果。如图 22-7 所示，在左侧的数据库浏览框中，右键单击 poem 表，并在弹出的菜单中选择 Generate query for table（生成表查询）/SELECT（选择）。

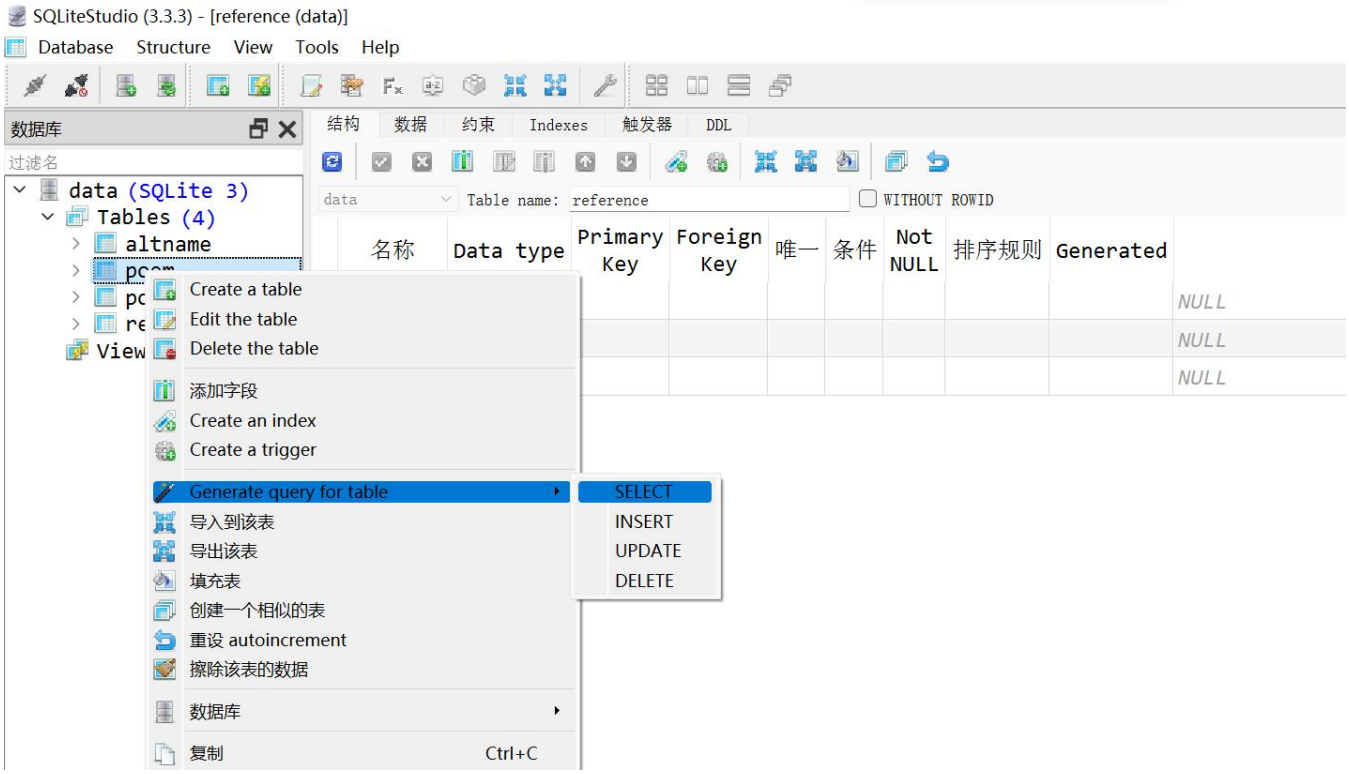


图 22-7 创建表查询

接着，在 SQL 语句输入框中录入如图 22-8 所示的 SQL 语句，然后单击执行语句按钮（蓝色三角形）。语句成功执行后，下方的结果框显示 poem 表的数据总行数为 42948。

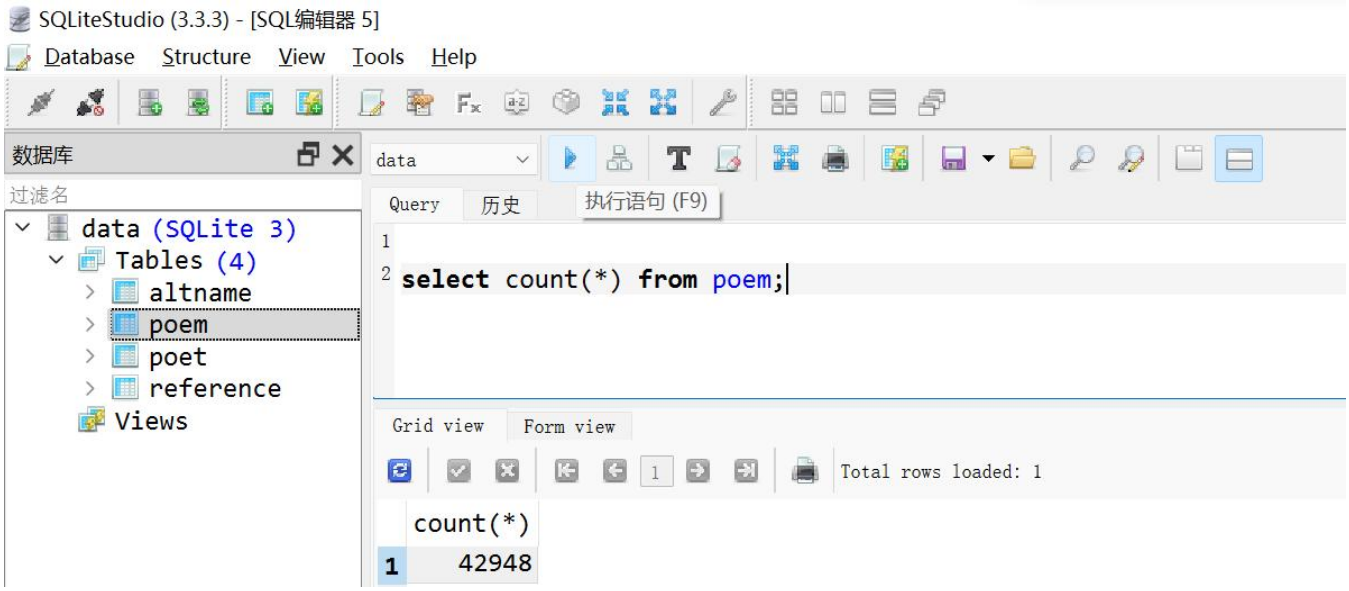


图 22-8 查询 poem 表的总记录数

如果把 SQL 语句修改成如图 22-9 所示，并再次执行，则可以查询并显示 poem 表数据的前 10 行。

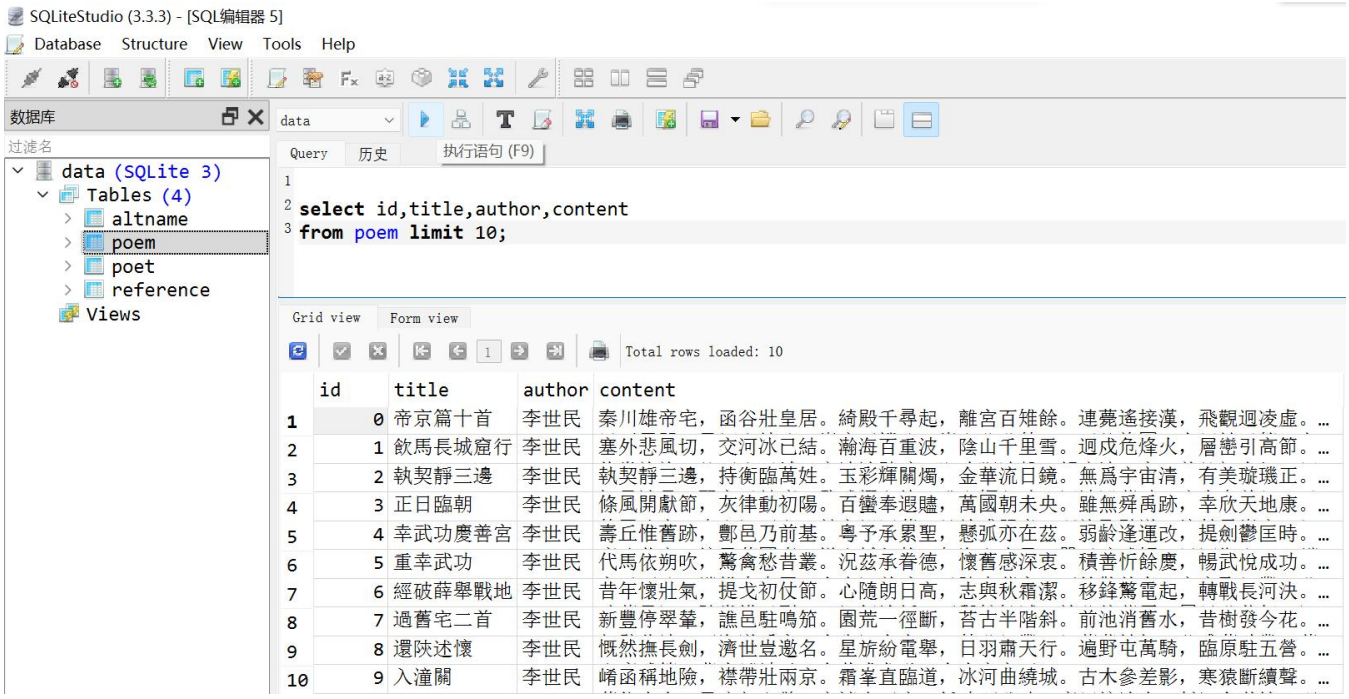


图 22-9 查询 poem 表前 10 行

此处涉及的两条 SQL 语句的语义解释请见表 22-4。SQL 语言本身是大小写不敏感的，即 SELECT 与 select 不作区分，等同使用。

表 22-4 SQL 语句解析 2

语句	SELECT count(*) FROM poem;
说明	从 poem 表统计并返回数据总行数。该语句的执行结果数据集包含一行数据且仅有一个名为 count(*)的字段，该字段预期为一个整数。
语句	SELECT id,title,author,content FROM poem LIMIT 10;
说明	从 poem 表中查询并返回前 10 行记录，包括 id、title、author 及 content 共 4 个字段。该语句的执行结果数据集包含 10 行数据，4 个字段。

22.2.4 诗人名录及别名

要通过对唐诗的检索确定诗人之间的引用关系，并不容易。最大的困难在于古代中国人的别名太多。比如，杜甫，按字称子美，按排行称杜二，按官职称杜工部，有时还甚至被称为老杜。为了解决上述问题，我们下载了哈佛大学编撰的《中国历代人物传记资料库》，关于这个资料库的信息，请访问下述二维链接。

二维码 资源下载 / 中国历代人物传记资料库

http://codelearn.club/2022/02/figurelib/

这个人物资料库中包含中国历代人物，并非特指唐代诗人，因此重名太多，例如可能存在多个王维、李良的情况。如果仅凭全唐诗的作者名，很难在人物资料库中准确定位那个作为诗

人的王维以及他的别名王右丞。还好，我们还有生卒年可以用。唐朝建立于618年，灭亡于907年。我们删除了那些生卒年明确且与唐朝没有交集的全部人名记录，也删除了仅记录有生年或卒年，但从生年或卒年看明显跟唐朝没关系的人名记录。顺便，我们也删除了全部生卒年均不明确的人名记录。经过整理，我们得到了两个数据表，诗人（poet）表（22-5）和别名表（altname）（22-6）。

表22-5 诗人(poet)表字段清单

字段名	类型	用途
id	int	用作主键，表示一个诗人的唯一编号
name	text	诗人的姓名，比如“李白”、“刘禹锡”
birthyear	int	诗人的出生年份，如果为0，表示生年不详
deathyear	int	诗人的死亡年份，如果为0，表示卒年不详

表22-6 别名（altname）表字段清单

字段名	类型	用途
id	int	人物在poet表中的id号，由于一个诗人可能拥有多个别名，因此，altname表中的id字段是允许重复的
name	text	人物的别名

接下来，我们在SQLiteStudio中通过SQL语句来查询一下杜甫的别名。第一步，执行下述SQL语句：SELECT * FROM poet WHERE name = '杜甫'，得到如图22-10所示的查询结果。请读者注意，与C/C++语言不同，SQL语言中的字符串使用单引号包裹。

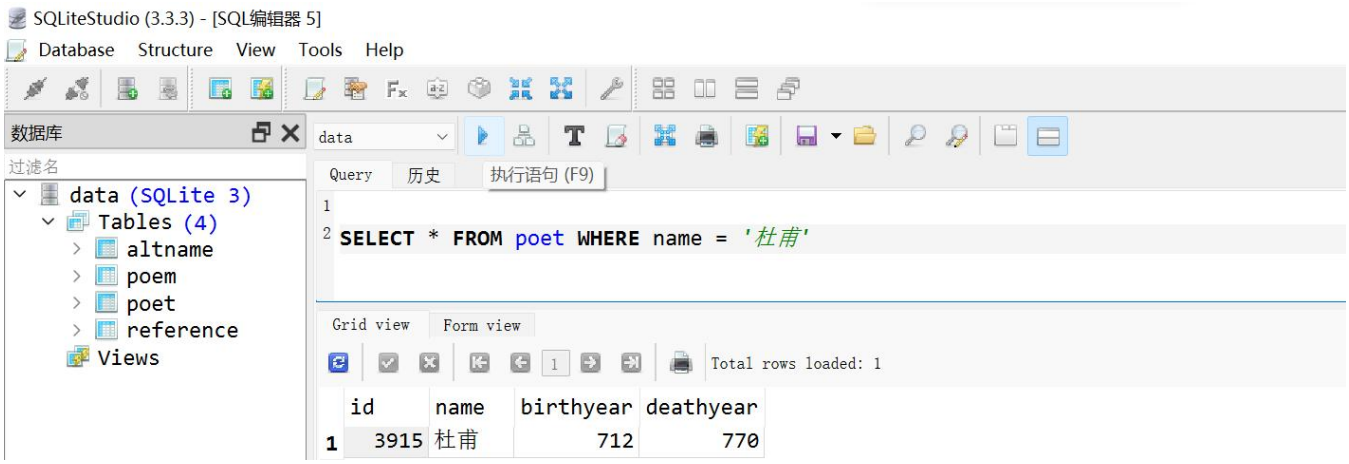


图22-10 poet表中的杜甫

在上述结果中可见，杜甫在poet表中的id号为3915，他生于公元712年，卒于公元770年。第二步，我们使用3915这个id去altname表中查询他的别名：SELECT * FROM altname WHERE id = 3915，查询结果如图22-11所示。杜甫的别名子美、工部都是我们所熟悉的，老杜这个别名有点出人意料。

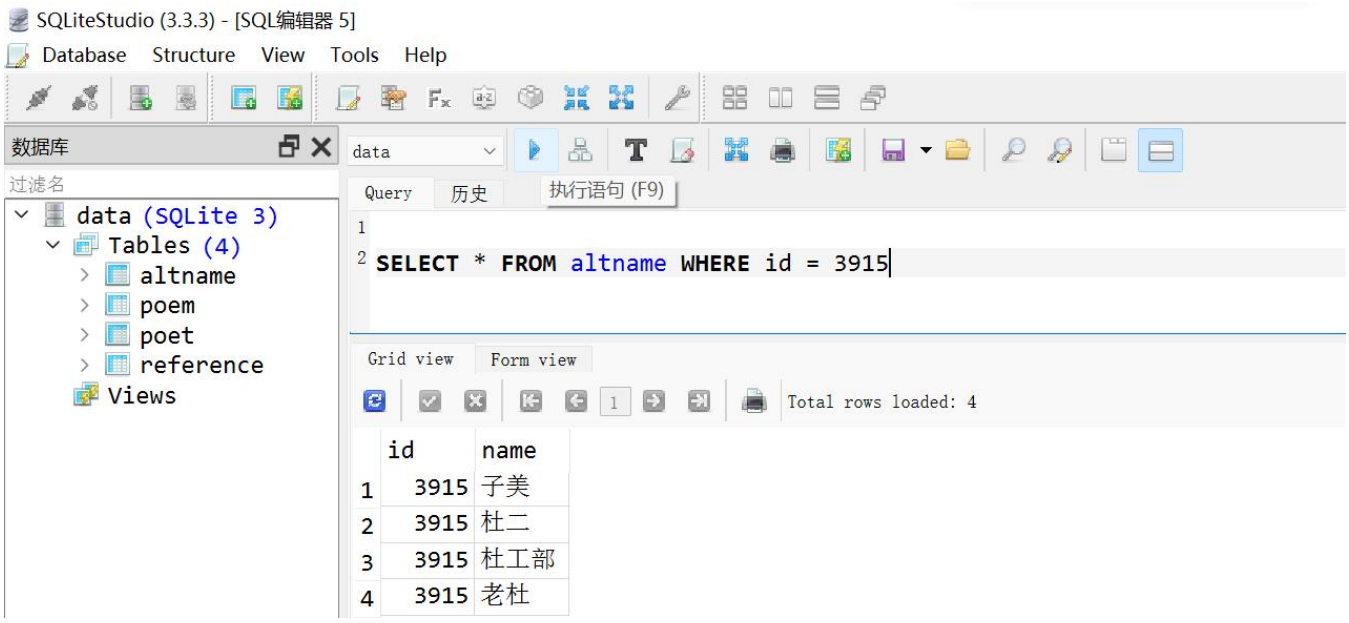


图 22-11 altname 表中的杜甫别名

前述两行 SQL 语句的解释详见表 22-7。

表 22-7 SQL 语句解析 3

语句	SELECT * FROM poet WHERE name = '杜甫'
说明	对 poet 表进行筛选，返回姓名（name）字段等于'杜甫'的所有行。*号代码返回行中应包括表格的全部字段（列），对于 poet 表，这些字段应为 id、name、birthyear 和 deathyear。
语句	SELECT * FROM altname WHERE id = 3915
说明	对 altname 表进行筛选，返回诗人 id 等于 3915 的所有行。同样地，*号代表结果数据集应包括 altname 表的全部字段，也就是 id 和 name。

22.2.5 引用关系

为了保存诗人之间的引用关系，我们还创建了一个名为 reference 的表（22-8）。

表 22-8 引用（reference）表字段清单

字段名	类型	作用
authorid	int	诗的作者在 poet 表中的 id
refid	int	被引用的诗人在 poet 表中的 id
poemid	int	诗在 poem 表中的 id

李白在《黄鹤楼送孟浩然之广陵》一诗中引用了孟浩然，李白在 poet 表中的 id 号为 32 540，孟浩然在 poet 表中的 id 号为 93 956，《黄鹤楼送孟浩然之广陵》在 poem 中的 id 号为 7 228，故 reference 表中存在一行记录，其值如图 22-12 所示。

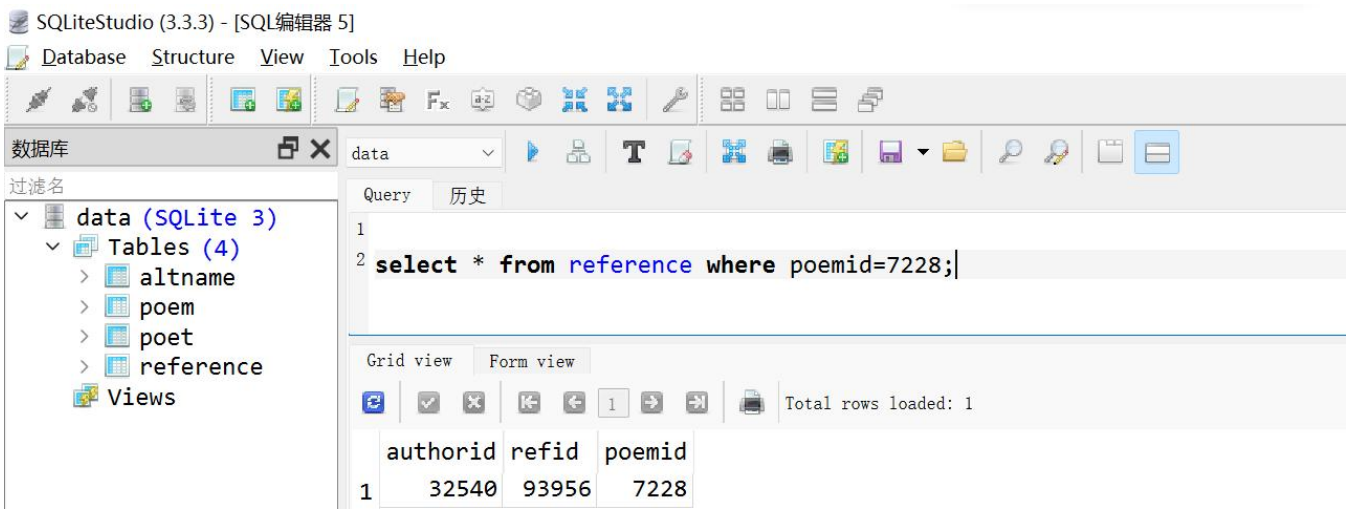


图 22-12 “黄鹤楼送孟浩然之广陵”所对应的李白对孟浩然的引用记录

上述李白、孟浩然和诗的 id 号可以通过下述 SQL 语句查询而得。其中，第三行的%表示通配符，它表示 0 到多个任意字符。注意，如果读者进行实际查询，SQL 语句中的第三行中的诗名应使用繁体中文。

```
select * from poet where name = '李白';
select * from poet where name = '孟浩然';
select * from poem where title like '%黄鹤楼送孟浩然之广陵%';
```

22.3 构建诗人关系网络

构建诗人关系网络，即是对 poem 表中的唐诗题名和内容进行分析，对照 poet 表和 altname 表，找出诗人之间的引用关系，并将引用关系记录存入 reference。对于每一首唐诗的题名和目录，我们均需要使用字符串匹配方法逐一检查所有诗人的姓名和别名，因此，诗人关系网络的构造运算量大，花费时间较长。为了设计出友好的用户界面，我们需要：(1) 当操作者点击主窗口中的“构建关系网络”按钮后，需要显示一个进度条对话框以避免操作者的焦虑，如图 22-13 所示；(2) 创建一个专门的线程（thread）来执行诗人关系网络的构建工作。如果我们在主线程也就是消息循环中进行耗时过长的操作，主线程将没有机会收取和处理来自操作者的操作信息，软件界面将“长时间”表现为“卡”住，即不回应操作者的任意操作。软件是否崩溃了？操作还要多久才能完成？没有预期的等待是对用户心理的极大折磨。

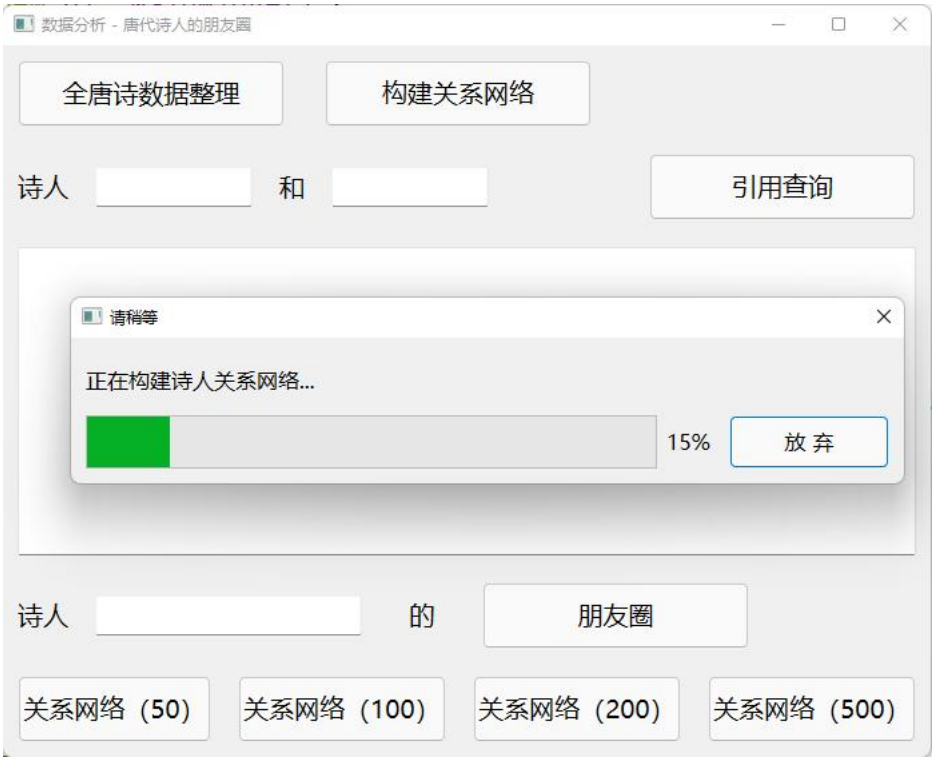


图 22-13 构建诗人关系网络进度对话框

22.3.1 创建 Poem 及 Poet 类

为了给后续工作做好准备，请读者新创 Poem 类并加入 PoetsNetwork 项目。其中，poem.cpp 中没有代码，头文件 poem.h 的内容如下。

```
1  #ifndef POEM_H
2  #define POEM_H
3  #include <QString>
4
5  class Poem {
6  public:
7      Poem(int id, const QString& title,
8           const QString& author, const QString& content):
9           iId(id),sTitle(title),sAuthor(author),sContent(content){ }
10
11     int iId = 0;
12     QString sTitle;
13     QString sAuthor;
14     QString sContent;
15 };
16
17 #endif // POEM_H
```

一个 Poem 对象存储一首诗，其数据成员的名称 iId、sTitle、sAuthor 和 sContent 分别与数据库 poem 表中的字段名 id、title、author 和 content 对应。出于简便，作者把 Poem 的构造函数的实现直接写在了头文件中，按照本书稍早的描述，这也是一种函数内联的方式。在构造函数初始化列表（第 9 行）中，构造函数对数据成员进行了初始化。

请读者新建 Poet 类并加入 PoetsNetwork 项目。头文件 poet.h 的内容如下。

```
1  #ifndef POET_H
2  #define POET_H
3  #include <QString>
4  #include <QVector>
5
6  class Poet {
7  public:
8      Poet(){}
9      Poet(int id, const QString& name, int birthyear, int deathyear);
10     int iId = 0;
11     QString sName;
12     int iBirthYear = 0;
13     int iDeathYear = 0;
14     QVector<QString> altNames;
15
16     static bool getPoetByName(const QString& name, Poet& poet);
17     static QVector<QString> getAltNamesById(int id);
18     static QString getPoetNameById(int id);
19 };
20
21 #endif // POET_H
```

一个 Poet 对象存储一位诗人，其数据成员 iId、sName、iBirthYear 和 iDeathYear 分别对应数据库 poet 表中的 id、name、birthyear 和 deathyear 字段；字符串向量 altNames 则存储诗人对象的全部别名，其数据预期来源于 altname 表。

第 16 行：静态成员函数 getPoetByName()从 poet 表中查询指定姓名的唐代诗人信息，如果找到“确定”的唐代诗人，将信息填入 poet 引用，并返回真。

第 17 行：静态成员函数 getAltNamesById()使用诗人 id 号从 altname 表中查询该诗人 id 所对应的全部别名。

第 18 行：静态成员函数 getPoetNameById()使用诗人 id 号从 poet 表中查询并返回诗人姓名。

poet.cpp 中的代码分段解读如下。

```
1  #include "poet.h"
2  #include <QSqlQuery>
3  #include <QSqlRecord>
4  #include <QSqlField>
5
6  Poet::Poet(int id, const QString& name, int birthyear, int deathyear){
7      iId = id; sName = name; iBirthYear = birthyear; iDeathYear = deathyear;
8  }
```

第 2 ~ 4 行：导入 QSqlQuery、QSqlRecord 及 QSqlField 头文件，这些类型是使用 SQL 进行数据库查询所必需的。

第 6 ~ 8 行：构造函数使用参数值初始化数据成员。

```
10 bool Poet::getPoetByName(const QString& name, Poet& poet){
11     int iTangStart = 618, iTangEnd = 907;  //唐建立及灭亡年份
12     QVector<Poet> candidates;
13
14     QSqlQuery q;
```

```
15      auto sSql = QString("SELECT id, name, birthyear, deathyear "
16                          "FROM poet WHERE name like '%%1%'").arg(name);
17      Q_ASSERT(q.exec(sSql));
18      while (q.next()){
19          auto r = q.record();
20          auto id = r.field(0).value().toInt();
21          auto name = r.field(1).value().toString();
22          auto birthyear = r.field(2).value().toInt();
23          auto deathyear = r.field(3).value().toInt();
24          if (birthyear && deathyear){
25              if (birthyear < deathyear and deathyear > iTangStart){
26                  poet = Poet(id,name,birthyear,deathyear);
27                  poet.altNames = getAltNamesById(id);
28                  return true;
29              }
30          }
31          else if (birthyear || deathyear){
32              auto year = birthyear?birthyear:deathyear;
33              if (year>iTangStart && year<iTangEnd)
34                  candidates.emplace_back(id,name,birthyear,deathyear);
35          }
36      }
37
38      if (candidates.size()!=1)
39          return false;
40
41      poet = candidates[0];
42      poet.altNames = getAltNamesById(poet.iId);
43      return true;
44  }
```

getPoetByName()函数用于从 poet 表中找出指定姓名的唐代诗人，如果成功找到，将信息填入 poet 引用，将返回真，否则返回假。考虑到 poet 表中同时存在中国历朝历代的历史人物，重名情况较多，需要结合生卒年加以鉴别。

第 15 ~ 16 行：假设 name 参数的值为“王维”，将产生如图 22-14 所示的 SQL 语句，其中的%在 SQL 语言里被用作通配符，代表 0 到多个任意字符。这里之所以采用模糊匹配，是因为在 poet 表，即《中国历代人物传记资料库》中，姓名后面常常包括备注，如李隆基，在 poet 表中其 name 字段内容实为：李隆基（唐玄宗）。如图 22-14 所示的查询结果，poet 表中存在两个王维，需要根据生卒年进行鉴别以返回正确的王维。

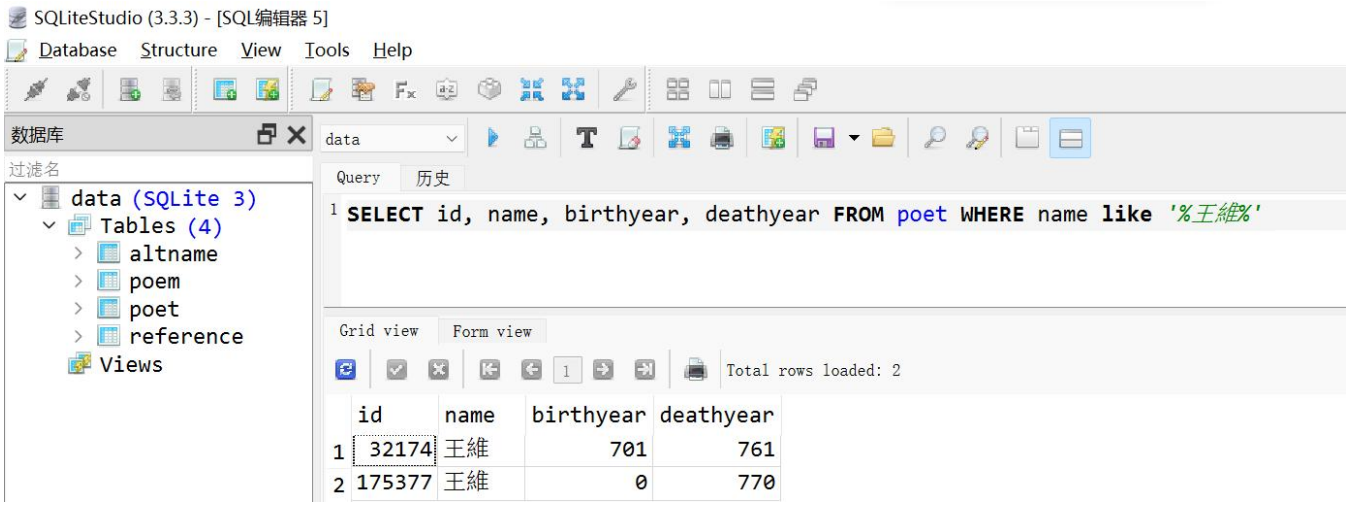


图 22-14 诗人姓名的模糊查询

第 17 行：通过 SQL 查询对象 q 执行 SQL 语句 sSql。该语句执行后，q 内将“包含”查询的结果数据集。

第 18 ~ 36 行：使用 while 循环对 q 中的结果数据集进行逐行遍历。

第 24 ~ 30 行：如果卒年明确且于唐代有交集，将信息填入 poet 引用，并通过 getAltNamesById()函数获取诗人的别名清单，然后返回真。第 24 行的条件判断使用了非零即真的原则，只要 birthyear 和 deathyear 的值不为 0，就表示生年或卒年是明确可考的。

第 31 ~ 35 行：如果生年和卒年之一是明确可考的，且生年或卒年与唐代有交集，将信息加入“候选人”（candidates）向量。

第 38 ~ 39 行：如果“候选人”数量不等于 1，表明没有到适配的诗人或者潜在的候选人多于 1 个，无法确定，返回 false 代表查询失败。

第 41 ~ 43 行：将唯一的“候选人”信息填入 poet 引用，并通过 getAltNamesById()函数获取其别名清单，然后返回真。

```
46 QVector<QString> Poet::getAltNamesById(int id) {
47     QSqlQuery q;
48     Q_ASSERT(q.exec(QString("SELECT name FROM altname WHERE id = %1").arg(id)));
49 
50     QVector<QString> r;
51     while (q.next()) {
52         auto n = q.record().field(0).value().toString();
53         if (n.size()>1) //别名至少要有两个字，单字别名为 刺，德
54             r << n;
55     }
56 
57     return r;
58 }
```

getAltNamesById()根据诗人 id 号查询并返回对应诗人的全部别名，返回值类型为一个字符串向量。

第 53 行：对别名的长度进行了限制，不允许单字别名被使用。如果允许象“刺”、“德”这样的单字别名被使用，最终程序分析得到的引用关系将是不可靠的，因为“德”字出现在诗的题名或者内容中，多数情况下并不代表人名。

第 54 行：r << n 执行的是向量 r 被重载的操作符函数<<，其用途就将字符串 n 添加至向量的尾部。

```
60 QString Poet::getPoetNameById(int id) {
61     QSqlQuery q;
62     Q_ASSERT(q.exec(QString("SELECT name FROM poet WHERE id = %1").arg(id)));
63     if (q.next())
64         return q.record().field(0).value().toString();
65     else
66         return "N/A";
67 }
```

getPoetNameById()根据诗人的 id 号查询并返回诗人的姓名，如果没找到，返回“N/A”。

22.3.2 进度条对话框

请读者按照本书 21.7.1 节所示的方法为项目 PoetsNetwork 创建 BuildNetwork 对话框，该对话框添加完成后，项目里将增加三个文件：buildnetwork.ui、buildnetwork.h 以及 buildnetwork.cpp。该对话框的运行效果如图 22-13 所示。

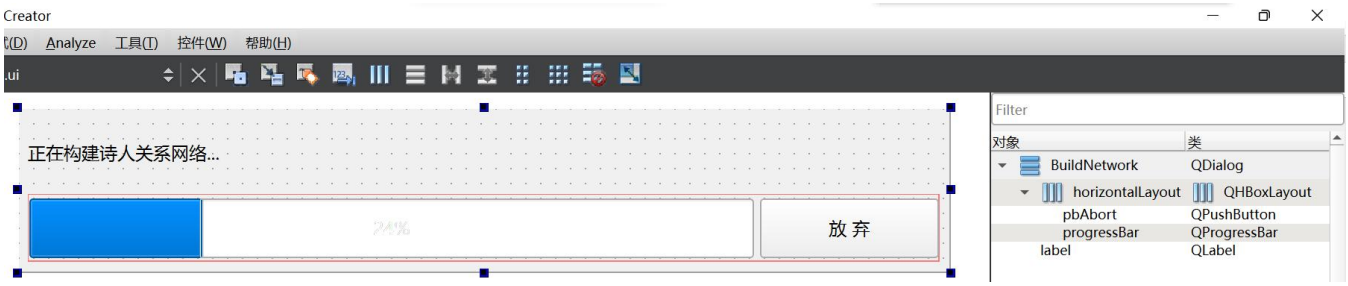


图 22-15 BuildNetwork 对话框的对象结构

图 22-15 展示了该对话框的对象结构。BuildNetwork 的父类是 QDialog，其自身是一个竖向布局。在该对话框内部，包括一个标签(QLabel)和一个横向布局，在横向布局内，包含进度条（QProgressBar）类型的部件 progressBar 以及按钮 pbAbort。显而易见，在诗人关系网络的“漫长”的构建过程中，如果点击放弃按钮，意味着要中断关系网络的构建过程。

接下来，我们在主窗口中为 pbConstructNetwork（“构建关系网络”）按钮的“released()”信号添加如下的槽函数。请读者注意，在当前阶段，暂时还不能录入下述代码的第 5 ~ 7 行，因为 BuildNetwork 对话框目前仅有一个框架，其并不存在名为 iReferenceCount 的公有数据成员。该成员预期用于返回成功发现并写入数据库的引用记录的总数量。

```
1  //mainwidget.cpp
2  void MainWidget::on_pbConstructNetwork_released() {
3      auto dlg = QSharedPointer<BuildNetwork>(new BuildNetwork(this));
4      dlg->exec();
5      auto r = dlg->iReferenceCount;
6      ui->textBrowser->setHtml(r<=0?QString("关系网络构建过程中途中断。")
7          :QString("关系网络构建完成，共发现%1 条引用关系。").arg(r));
8  }
```

第 6 ~ 7 行：根据对话框所返回的引用记录数判断关系网络的构建过程是否成功完成，并将相应结果信号填入 textBrower。

第 3 行：QSharedPointer 是 Qt 版本的共享智能指针（shared_ptr）。智能指针的采用，使得我们不必担心 dlg 对象的回收问题。除非智能指针的使用将严重影响程序的执行效率，作者尽可能多地使用智能指针以避免内存泄漏。

当前阶段，请读者仅录入上述槽函数的前两行（即第 3 ~ 4 行），确保对话框可以在点击“构建关系网络”按钮后被正常打开。此时，点击对话框右上角的关闭按钮（×），可结束对话框。程序设计不是一蹴而就的，程序员正常先搭出一个简易的可运行可验证的程序框架，然后再逐步添加程序的血和肉。

22.3.3 关系网络构建线程

如前所述，为了避免软件界面长时间“卡”住，我们不能在主线程的消息循环中去执行耗时的关系网络构建工作，也就是说，关系网络的构建工作不能放在槽函数中进行，应该建立一个专门的线程来构建关系网络。

请读者在 PoetsNetwork 中新建一个名为 ThreadNetworkBuilder 的新类。头文件 threadnetworkbuilder.h 的内容如下。

```
1  #ifndef THREADNETWORKBUILDER_H
2  #define THREADNETWORKBUILDER_H
3  #include <QThread>
4  #include <QVector>
5  #include <QMap>
6  #include "poem.h"
7  #include "poet.h"
8
9  class ThreadNetworkBuilder:public QThread {
10     Q_OBJECT
11 public:
12     ThreadNetworkBuilder();
13     void run() override;
14     volatile int iProgress {0};       //进度，0-100
15     void requestTerminate();          //主线程要求线程终止
16     volatile int iReferenceCount = 0; //成功写入数据库的引用记录数
17
18 signals:
19     void progressUpdate();
20
21 private:
22     QVector<Poem> poems;
23     QMap<QString,Poet> mapPoets;
24     void loadPoemsPoets();
25     void buildPoetsNetwork();
26     volatile bool bRequestTerminated {false};
27 };
28
29 #endif // THREADNETWORKBUILDER_H
```

第 5 行：QMap 是 Qt 版本的有序关联容器，其内存储着“键值对”，它可以帮助我们完成从键到值的快速映射。相关细节请回顾本书 19.6.1 节。

第 9 行：ThreadNetworkBuilder 的父类是 QThread，QThread 是 Qt 中的线程类型。

第 10 行：为了让 ThreaadNetworkBuilder 支持信号与槽机制，加入 Q_OBJECT 宏。

第 13 行：公用函数 run()是线程对象的运行实体，该函数由线程来执行。当 run()函数 return 时，即意味着线程执行的终结，线程对象的“finished()”信号随即被触发。

第 14 行：主线程可以随时访问线程对象的 iProgress 属性来获取关系网络构建的进度，其取值范围为 **0 ~ 100**。请读者注意，iProgress 的类型是 volatile int。关于 **volatile**，请读者留意后续说明。

第 15 行：主线程可以执行线程对象的 requestTerminate()函数要求线程提前终止。事实上，该函数的执行并不能直接终止 run()函数运行，它只是将第 26 行的 bRequestTerminated 属性设为

真。在 run()函数内部，我们会经常检查 bRequestTerminated 属性的值，如果为真，说明主线程已发出了终止执行的要求，run()函数主动 return 以结束线程。请读者注意，bRequestTerminated 属性也是 volatible 的。

第 16 行：线程执行结束后，主线程可从线程对象的 iReferenceCount 属性获取关系网络构建的结果：成功发现并写入数据库的引用记录的总数量。如果该值为-1，表示关系网络构建工作被中途中断。

第 **18 ~ 19** 行：progressUpdate()是 ThreadNetworkBuilder 类型的一个信号，当 run()函数认为关系网络构建工作获得了进展时，将会主动发送一个信号。该信号会进入主线程的消息循环，主线程将会调用与该信号相关联的槽函数，以更新对话框中的进度条显示。请读者注意，在 Qt 中，信号的声明格式类似于函数，但它本身并不是函数。在 threadnetworkbuilder.cpp 中，并不存在 progressUpdate()函数的定义。第 18 行的 signals 是一个宏，它是 Qt 对 C++语言的扩展，表示其之后的“函数声明”都是信号。

第 22 行：私有的 poems 向量用于存储读入内存中的唐诗。

第 23 行：私有的 mapPoets 是从诗人姓名到诗人对象的映射。在关系网络构建的过程中，它可以帮助我们快速地从诗人姓名映射至诗人对象，进而得到诗人 id、生卒年等信息，避免对数据库进行不必要的频繁访问。

第 24 行：私有函数 **loadPoemsPoets()**对唐诗和诗人进行筛选，符合要求的唐诗被装入 poems 向量，符合要求的诗人对象（作为值）连同其姓名（作为键）则被装入 mapPoets 映射。

第 25 行：私有函数 **buildPoetsNetwork()**对 poems 向量中的唐诗和 mapPoets 中的诗人对象进行匹配处理，发现诗人之间的引用关系，并存入数据库。该函数是关系网络构建工作的主体，稍后我们会看到，run()是通过调用该函数来完成关系网络的构建的。

要点

当一个对象被多个线程共享时，需要将其定义为 **volatile**。**volatile** 在英文中的意思为易变的，不稳定的，在 **C/C++**中，它是一个类型修饰符（**type specifier**），它要求编译器在生成机器指令时总是从内存中获其值，而不能使用位于寄存器中的副本。

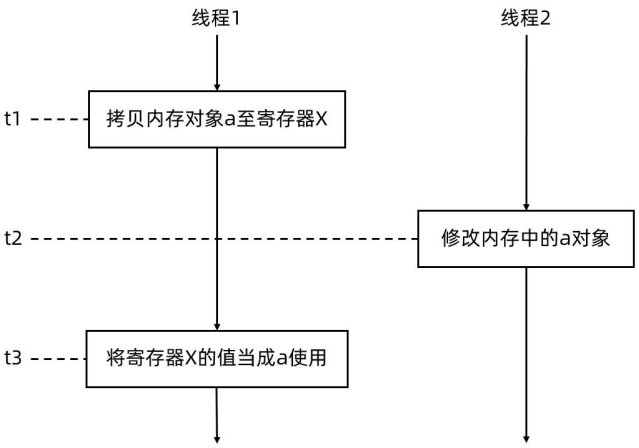


图 22-16 内存访问优化示例

我们结合图 22-16 来说明将多线程共享的对象设置为 volatile 的必要性。线程 1 在时间 t_1 将内存对象 a 读入到寄存器 X，从时间 t_1 到 t_3，线程 1 既没有修改 a 的值，也没有修改 X 的值，从线程 1 的角度看，a 和 X 相等。当时间 t_3 程序再次需要使用到对象 a 时，编译器会认为直接从寄存器 X 取值更为划算：因为寄存器的访问速度远高于内存。事实上，编译器确实会按照这一逻辑将那些不必要内存访问优化掉。

对于单个线程的程序，这样做是安全的。但对于多线程共享的对象，上述内存访问优化会有风险。如图 22-16 所示，线程 2 在 t_1 和 t_3 之间的中间时刻 t_2 修改了内存对象 a 的值，线程 1 在时间 t_3 从寄存器 X 所得到的，并不是对象 a 的最新值，寄存器 X 中的值是“脏”的。

通过将对象 a 设为 volatile，可以禁止编译器对 a 对象的内存访问进行优化，从而避免上述问题的发生。

接下来，分段讨论 threadnetworkbuilder.cpp 中的代码。

```
1  #include "threadnetworkbuilder.h"
2  #include <QSqlQuery>
3  #include <QSqlRecord>
4  #include <QSqlField>
5  #include <QSqlError>
6  #include "dbhelper.h"
7  #include "poet.h"
8
9  ThreadNetworkBuilder::ThreadNetworkBuilder() {
10    //按 Qt 文档，禁止通过 terminate()函数终止线程
11    setTerminationEnabled(false);
12 }
```

第 9 ~ 12 行：在构造函数中，我们执行 setTerminationEnabled(false)禁止其它线程通过 QThread::terminate()函数强行中止线程的执行。按照 Qt 文档的说法，这样做是有风险的，所以我们自定义了 requestTerminate()函数来达成类似目的。请读者注意，在本项目中，线程对象是由主线程创建的，这个构造函数本身也是在主线程中执行的。

```
14 void ThreadNetworkBuilder::run() {
15     DBHelper::openDatabase();  //打开线程数据库连接
16     buildPoetsNetwork();
17     DBHelper::closeDatabase(); //关闭线程数据库连接
18 }
```

第 14 ~ 18 行：run()函数是线程的执行实体，该函数由线程执行。run()函数 return 即表示线程的执行结束，线程结束时，其“finished()”信号会被发射。

第 15 行：Qt 中的 QSqlDatabase 数据库连接对象并不是“线程安全”（thread safe）的，这意味着该对象不允许跨线程访问。因此，线程需要建立自己的数据库连接，通过执行 DBHelper 的静态成员函数 openDatabase()打开数据库连接。稍后我们还会看到，在线程对象开始运行前，主线程会主动关闭数据库连接，以避免冲突。

第 16 行：执行 buildPoetsNetwork()函数构建诗人关系网络。

第 17 行：关闭数据库连接。

```
20 void ThreadNetworkBuilder::requestTerminate() {
21     //主线程要求线程终止
22     bRequestTerminated = true;
23     qDebug() << "ThreadNetworkBuilder - requestTerminate.";
24 }
```

第 20 ~ 24 行：定义公有的 requestTerminate()函数。主线程通过执行线程对象的 requestTerminate()函数来“中断”线程的执行。

第 22 行：将 volatile 的私有属性 bRequestTerminated 设为真。该标志表明线程对象被主线程要求中止执行。后面我们会看到，run()函数内的 buildPoetsNetwork()函数会经常检查该标志，发现其为真，便返回以结束线程的运行。

```
84  void ThreadNetworkBuilder::loadPoemsPoets() {
85      QSqlQuery q;
86      Q_ASSERT(q.exec("SELECT count(*) FROM poem"));
87      Q_ASSERT(q.next());
88      auto size = q.record().field(0).value().toInt();
89      Q_ASSERT(size>0);
90
91      int idx = 0;
92      q.clear();
93      Q_ASSERT(q.exec("SELECT id, title, author, content FROM poem"));
94      while (q.next()) {
95          if (bRequestTerminated)
96              return;
97
98          int t = 30 * (idx++) / size;
99          if (t>iProgress){
100             iProgress = t;
101             emit progressUpdate();
102         }
103
104         auto r = q.record();
105         auto id = r.field(0).value().toInt();
106         auto title = r.field(1).value().toString();
107         auto author = r.field(2).value().toString();
108         auto content = r.field(3).value().toString();
109
110         Poem m(id,title,author,content);
111         if (mapPoets.contains(author))
112             poems.append(m); //诗人 author 已存在于映射 mapPoets 中
113         else {
114             Poet t;
115             if (Poet::getPoetByName(author,t)){
116                 //在 poet 表中找到诗人 author
117                 poems.append(m);
118                 mapPoets[author] = t;
119             }
120         }
121     }
122 }
```

第 84 ~ 122 行：在进行关系网络构建之前，需要将符合要求的唐诗和诗人从数据库调入内存，这就是 loadPoemsPoets()函数要完成的任务。稍后我们会看到，buildPoetsNetwork()函数执行的第一步便是调用 loadPoemsPoets()。本函数将符合要求的唐诗装入 poems 向量，把符合要求的诗人对象装入 mapPoets 映射。

第 85 ~ 90 行：使用 SQL 语句查询并统计 poem 表中待筛选的唐诗的总数量（size）。size 变量在稍后被我们用于计算工作进度。

第 91 行：idx 变量表示当前正在处理的唐诗的编号，从 0 开始计数。

第 93 行：执行 SELECT 语句从 poem 表查询全部唐诗数据。

第 **94 ~ 121** 行：使用 while 循环逐行处理唐诗。

第 **95 ~ 96** 行：在循环中对 bRequestTerminated 标志进行检查，如果该标志为真，说明主线程要求线程中止执行，直接返回。

第 98 行：按作者的估计，唐诗和诗人的读取和筛选占总工作量的 30%，结合 idx 及 size 变量计算得到当前进度 t。

第 **99 ~ 102** 行：如果当前进度 t 大于 iProgress，发射“progressUpdate()”信号。主线程的消息循环在收到该信号后会调用对应的槽函数，刷新进度条对话框中的进度显示。

第 **104 ~ 108** 行：从数据库记录中获取 id、title、author 和 content 字段的值。

第 110 行：构建 Poem 对象 m。

第 **111 ~ 120** 行：如果诗人的姓名已存在于映射 mapPoets 中，说明这首唐诗的作者明确且可以对应到《中国历代人物传记资料库》中的某唐代人物，将该唐诗加入 poems 向量。否则，通过 Poet::getPoetByName()函数从 poet 表中筛查，如果能够对应到符合要求的唐代人物，该函数返回真，程序将唐诗装入 poems 向量，将诗人对象 t 加入映射 mapPoets。如果一首唐诗的作者无法对应到符合要求的唐代人物，程序会忽略这首唐诗，将其排除在统计范围之外。

```
26 void ThreadNetworkBuilder::buildPoetsNetwork() {
27     loadPoemsPoets();
28 
29     int idx = 0;
30     QVariantList authorids, refids, poemids;
31     for (auto& m:poems){
32         if (bRequestTerminated)
33             return;
34 
35         int t = 30 + 65 *(idx++)/poems.size();
36         if (t>iProgress){
37             iProgress = t;
38             emit progressUpdate();
39         }
40 
41         auto authorid = mapPoets[m.sAuthor].iId;
42         for (auto& t:mapPoets.values()) {
43             if (m.sTitle.contains(t.sName) || m.sContent.contains(t.sName)){
44                 //直接引用了诗人的本名
45                 authorids << authorid;
46                 refids << t.iId;
47                 poemids << m.iId;
48                 continue;
49             }
50 
51             //尝试别名
52             for (auto& altName:t.altNames){
53                 if (m.sTitle.contains(altName) || m.sContent.contains(altName)){
54                     authorids << authorid;
```

```
55                      refids << t.iId;
56                      poemids << m.iId;
57                      break;
58                  }
59              }
60          }
61      }
62
63      QSqlQuery q;
64      Q_ASSERT(q.exec("DELETE FROM reference"));
65
66      q.clear();
67      q.prepare("INSERT INTO reference VALUES (?,?,?)");
68      q.addBindValue(authorids); q.addBindValue(refids);
69      q.addBindValue(poemids);
70
71      DBHelper::db.transaction();    //开始数据库事务
72      if (!q.execBatch()){
73          qDebug() << q.lastError();
74          DBHelper::db.rollback();   //回滚数据库事务
75          return;
76      }
77      DBHelper::db.commit();         //提交数据库事务
78
79      iReferenceCount = int(refids.size());
80      iProgress = 100;
81      emit progressUpdate();
82  }
```

buildPoetsNetwork()函数对 poems 向量中的唐诗和 mapPoets 中的诗人对象进行匹配处理，发现诗人之间的引用关系，并存入数据库。

第 27 行：读入并筛选唐诗和诗人，该函数的执行成果是 poems 向量和 mapPoets 映射。

第 29 行：稍后第 31 行的 for 循环将逐行分析 poems 向量中的唐诗。变量 idx 表示当前正在处理的唐诗的序号。

第 30 行：authorids、refids 和 poemsids 是三个元素类型为 QVariant 的容器，其内分别存储 22.2.5 节所述的 reference 表的对应字段的值。第 31 行的 for 循环每发现一个引用关系，就会将作者 id、被引用人 id、相关唐诗的 id 依序存入这三个容器。

第 31 ~ 61 行：通过 for 循环逐一分析 poems 向量中的唐诗 m。

第 32 ~ 33 行：在循环内检查 bRequestTerminated 属性的值，真则表示主线程提出了中止执行的要求，return 以结束函数执行。

第 35 ~ 39 行：计算工作进度，必要则发射“progressUpdate()”信号。根据作者的估计，loadPoemsPoets()工作量占比总工作量的 30%， 引用关系分析占 65%，将引用关系写入数据库占 5%。

第 41 行：通过 mapPoets 映射找到 m.sAuthor 对应的诗人对象并得到作者 id。

第 42 ~ 60 行：mapPoets.values()函数返回映射内的全部诗人对象。逐一遍历这些诗人，分析这些诗人的本名或者别是否出现在诗的题名或者内容里，如果出现，则意味着发现了一条引用关系。

第 43 ~ 49 行：检查候选诗人 t 的本名是否出现在唐诗的题名或者内容里，如果出现，说明发现了一条引用关系，将 authorid（作者 id）、t.iId（被引用者 id）、m.iId（唐诗 id）存入相应的容器以备后续数据库写入之用。

第 52 ~ 59 行：逐一检查候选诗人 t 的别名是否出现在唐诗的题名或者内容里，如果出现，也认为是一条新的引用关系。

第 63 ~ 64 行：执行 SQL 语句，删除数据库 reference 表中的全部数据。SQL 语句“DELETE FROM reference”没有通过 WHERE 子句对 reference 表中拟删除的记录进行限制，这意味着其执行将清空 reference 表。

第 66 ~ 77 行：通过批量执行 SQL 语句将位于 authorids、refids 和 poemids 容器内的引用关系写入数据库的 reference 表。写入过程和方法同 mainwidget.cpp 中的 parsePoemsIntoDatabase()函数，请参考该函数的代码说明。关于 INSERT SQL 语句的语义，也请参阅 parsePoemsIntoDatabase()函数的相关解释。同样地，我们采用了数据库的事务管理技术来避免 data.db 文件的多次写入。

第 79 行：修改 iReferenceCount 属性的值，BuildNetwork 对话框将通过线程对象的这个属性来获取线程执行结果，该值代表了成功发现并写入数据库的引用关系的总数量。

第 80 ~ 81 行：关系网络构建完成，更新进度为 100%，并发射“progressUpdate()”信号。

22.3.4 创建线程对象并执行

接下来，我们需要在 BuildNetwork 对话框中创建 ThreadNetworkBuilder 线程对象并启动执行。头文件 buildnetwork.h 中的代码如下。

```
1  #ifndef BUILDNETWORK_H
2  #define BUILDNETWORK_H
3  #include <QDialog>
4  #include <QSharedPointer>
5  #include "threadnetworkbuilder.h"
6
7  namespace Ui {
8  class BuildNetwork;
9  }
10
11 class BuildNetwork : public QDialog {
12     Q_OBJECT
13 public:
14     explicit BuildNetwork(QWidget *parent = nullptr);
15     ~BuildNetwork();
16     int iReferenceCount = 0;
17
18 protected:
19     void reject();
20
21 private slots:
22     void progressUpdate();      //进度更新槽函数
23     void builderFinished();     //线程结束槽函数
24     void on_pbAbort_released();
25
26 private:
```

```
27      Ui::BuildNetwork *ui;
28      QSharedPointer<ThreadNetworkBuilder> builder = nullptr;
29      bool bCloseEnabled {false};
30  };
31
32  #endif // BUILDNETWORK_H
```

第 16 行：公有属性 iReferenceCount 表示成功发现并写入数据库的引用关系的总数。主窗口 MainWidget.cpp 通过该属性获取关系网络构建的结果。

第 19 行：重写 QDialog 的 reject()槽函数以禁止操作者通过对话框右上角的关闭按钮（×）结束对话框。

第 22 行：进度更新槽函数。BuildNetwork 的构造函数会将该槽函数与线程对象的“progressUpdate()”信号相关联。

第 23 行：线程结束槽函数。BuildNetwork 的构造函数会将该槽函数与线程的“finished()”信号相关联，当线程执行结束后，该槽函数会被主线程的消息循环调用。

第 24 行：对话框内 pbAbort 按钮的槽函数。操作者点击并释放“放弃”按钮后，该槽函数会执行线程对象的 requestTerminate()成员函数要求中止线程执行。

第 28 行：指向线程对象的智能指针。

第 29 行：bCloseEnabled 成员表示程序是否允许对话框关闭，其默认值为 false。

对于 buildnetwork.cpp，我们分段解释如下。

```
1   #include "buildnetwork.h"
2   #include "ui_buildnetwork.h"
3   #include "dbhelper.h"
4
5   BuildNetwork::BuildNetwork(QWidget *parent) :
6       QDialog(parent),
7       ui(new Ui::BuildNetwork)
8   {
9       ui->setupUi(this);
10
11      ui->progressBar->setRange(0, 100);
12      ui->progressBar->setValue(0);
13      setFixedSize(width(), height());
14
15      DBHelper::closeDatabase();  //关闭主线程中的数据库连接
16
17      builder = QSharedPointer<ThreadNetworkBuilder>(new ThreadNetworkBuilder());
18      connect(builder.get(), &ThreadNetworkBuilder::progressUpdate,
19              this, &BuildNetwork::progressUpdate);
20      connect(builder.get(), &ThreadNetworkBuilder::finished,
21              this, &BuildNetwork::builderFinished);
22      builder->start();
23  }
```

BuildNetwork 对话框对象是在 mainwidget.cpp 的 pbConstructNetwork 按钮的槽函数中实例化的，这意味着第 5 ~ 23 行的构造函数是在主线程中执行的。事实上，本实践中只有 ThreadNetworkBuilder 的 run()函数是在关系网络构建线程而不是主线程中执行的。

第 11 ~ 12 行：设置进度条的值范围以及当前值。

第 13 行：固定对话框大小，不允许操作者通过鼠标拖动改变对话框尺寸。

第 15 行：为了让关系网络构建线程建立自己的数据库连接，关闭主线程中的数据库连接。

第 17 行：创建 ThreadNetworkBuilder 线程对象。智能指针的使用可以确保该对象在 BuildNetwork 对话框被析构时回收。

第 18 ~ 19 行：连接线程对象的“progressUpdate()”信号与对话框对象的槽函数“progressUpdate()”。

第 20 ~ 21 行：连接线程对象的“finished()”信号与对话框对象的槽函数“builderFinished()”。线程对象的“finished()”信号来自于其父类 QThread。

第 22 行：执行 builder 的 start()成员函数启动线程执行。通过该函数，程序将会通过操作系统 API 创建一个单独的线程，该线程的执行主体即为 builder 的 run()成员函数。

```
25 BuildNetwork::~BuildNetwork() {
26     builder = nullptr;
27     delete ui;
28 }
```

第 25 ~ 28 行：进度对话框的析构函数。在第 26 行中，我们把智能指针 builder 主动置为空，这将导致线程对象的析构。事实上，第 26 行可以没有，智能指针对象 builder 作为 BuildNetwork 对话框的成员，其析构会自动销毁线程对象。

```
30 void BuildNetwork::progressUpdate() {
31     if (builder)
32         ui->progressBar->setValue(builder->iProgress);
33 }
```

第 30 ~ 33 行：进度更新槽函数。当关系网络构建线程发射相关信号后，主线程的消息循环将调用这个函数。如第 32 行所示，函数简单地将进度条的值设定为 builder->iProgress。事实上，通过线程对象的成员来获取其执行进度并不是一个好主意，这导致对话框与线程之间的“接口”变得复杂。在 Qt 中，信号是可以附带参数的，读者可以对相关代码进行改行，使得上述槽函数可以通过参数获得当前执行进度，而不是访问 builder 对象的属性。

```
35 void BuildNetwork::builderFinished() {
36     if (builder)
37         iReferenceCount = builder->iReferenceCount;
38     bCloseEnabled = true;
39
40     DBHelper::openDatabase();  //重新打开主线程的数据库连接
41     close();
42 }
```

第 35 ~ 42 行：线程结束槽函数。在线程结束运行后，该槽函数会被主线程的消息循环所调用。

第 37 行：从线程对象获取执行结果，即成功发现并写入数据库的引用关系的总数量。

第 38 行：打开 bCloseEnabled 开关，允许对话框关闭。稍后进一步解释其工作机制。

第 40 行：线程结束运行后，重新建立主线程的数据库连接。

第 41 行：执行 QDialog 的 close()函数以关闭对话框。

```
44 void BuildNetwork::on_pbAbort_released() {
45     if (builder)
46         builder->requestTerminate();
47 }
```

第 44 ~ 47 行：“放弃”按钮的框函数。如第 46 行所示，通过执行 builder->requestTerminate()，主线程要求关系网络构建线程中止。

```
49 void BuildNetwork::reject() {
50     if (bCloseEnabled){
51         QDialog::reject();
52     }
53 }
```

第 49 ~ 53 行：无论是操作者按下了对话框右上角的关闭按钮（×），还是程序主动执行了对话框 close()函数，Qt 都会通过 QDialog::reject()来完成对话框的真正关闭，并将对话框的执行结果设为 Rejected(已拒绝)。如上述代码所示，只有当 bCloseEnabled 标志为真时，QDialog::reject()才会被执行。这确保了操作者无法通过对话框的右上角关闭按钮关闭对话框。

重新构建程序并运行，如果代码无误，读者将得到如图 22-13 所示的执行界面，点击“构建关系网络”按钮后，将看到 BuildNetwork 进度条对话框被显示出来，其中包含一个进度条以及一个“放弃”按钮。在作者的计算机上，整个关闭网络构建过程大约耗时 20 秒。执行结束后，进度条对话框会自动关闭，如图 22-17 所示，主窗口的 textBrower 显示了执行结果：发现了 4729 条引用关系。

图 22-17 构建关系网络执行结果

读者会发现，关系网络的构建线程虽然在“后台”执行，但操作者仍然可以很顺畅地操作进度条对话框。如果读者点击“放弃”按钮，构建线程很快终结，textBrower 则显示构建被放弃的信息。

22.4 引用查询

运行作者编写好的成品程序，在行编辑框 leRefPoet1 和 leRefPoet2 中分别输入李白和杜甫，然后单击“引用查询”按钮，可得如图 22-18 所示的执行结果。如图中所示，textBrower 显示了两位诗的姓名以及别名、生卒年等信息，还列出了两者相互引用的全部唐诗的原文。本节讨论“引用查询”功能的代码实现。

图 22-18 李白与杜甫的相互引用

22.4.1 创建 Reference 类

为了实现“引用查询”、“朋友圈”、“可视化关系网络”等功能，我们创建一个名为 Reference 的类型，其头文件 reference.h 的内容如下。

```
#ifndef REFERENCE_H
#define REFERENCE_H
#include <QVector>
#include "poem.h"

class Reference {
public:
    int iAuthorId;    //引用人 id
    int iRefId;       //被引用人 id
    int iCount;       //引用数量
    Reference(int authorid, int refid, int cnt):
        iAuthorId(authorid),iRefId(refid),iCount(cnt){ }
    static QVector<Poem> getReferencePoems(int authorid, int refid);
    static QVector<Reference> getFriendCycle(int authorid);
    static QVector<Reference> getReferences(int iLimit);
};

```

```
18 #endif // REFERENCE_H
```

第 8 ~ 10 行：假设一个 Reference 对象表示李白在诗作中提到杜甫 3 次，那么李白就是引用人，杜甫是被引用人，引用数量则为 3；iAuthorId 保存李白在 poet 表中的 id，iRefId 保存杜甫在 poet 表中的 id，cnt 则保存引用数量 3。

第 11 ~ 12 行：Reference 对象的构造函数，其通过构造函数初始化列表初始化 3 个数据成员。

第 13 行：静态成员函数 getReferencePoems(int authorid, int refid)查询并返回某位指定的诗人引用另一位指定诗人的全部诗作。其中，authorid 是引用人的 id，refid 是被引用人的 id，函数的返回值类型为 QVector<Poem>。

第 14 行：静态成员函数 getFriendCycle(int authorid)查询并返回某位指定诗人的全部相关引用关系，其返回值类型为 QVector<Reference>。在这些引用关系中，这位诗人既可以是引用人，也可以是被引用人。在稍后的 22.5 节，该函数被用于显示一个诗人的朋友圈。

第 15 行：静态成员函数 getReferences(int iLimit)查询将返回所有唐代诗人相互引用关系的前 iLimit 行，该函数的返回值类型为 QVector<Reference>。在稍后的 22.6 节，该函数被用于可视化诗人关系网络。

接下来分段讨论 reference.cpp 中的代码实现。

```
1  #include "reference.h"
2  #include <QSqlQuery>
3  #include <QSqlRecord>
4  #include <QSqlField>
5
6  QVector<Poem> Reference::getReferencePoems(int authorid, int refid){
7      QSqlQuery q;
8      QString sSql =
9          QString("SELECT id, title, author, content FROM poem WHERE id IN "
10         "(SELECT poemid FROM reference WHERE authorid = %1 and refid = %2)")
11             .arg(authorid).arg(refid);
12     Q_ASSERT(q.exec(sSql));
13
14     QVector<Poem> t;
15     while (q.next()){
16         auto r = q.record();   //取得 QsqlRecord 记录
17         t << Poem(r.field(0).value().toInt(),         //字段 0 是 id
18                   r.field(1).value().toString(),  //字段 1 是 title
19                   r.field(2).value().toString(),  //字段 2 是 author
20                   r.field(3).value().toString()); //字段 3 是 content
21     }
22
23     return t;
24 }
```

如前所述，该函数用于查询并返回 id 为 authorid 的引用人引用 id 为 refid 的被引用人的全部唐诗。如代码所见，整个函数的执行过程分为两步。首先通过 SQL 查询获得结果数据集，然后遍历结果数据集，并逐行转换成 Poem 对象，加入向量 t 然后返回。我们重点解释相关的 SQL 语句。

第 9 ~ 11 行：通过连续两次的 arg()函数调用完成 SQL 语句字符串的格式化。本例中，authorid 替代了占位符 1%，refid 替代了占位符 2%。

对照前文，我们已知李白的 id 为 32 540，杜甫的 id 的是 3 915。如果要查询李白引用杜甫的所有唐诗，则经格式化后的上述 SQL 语句如下所示。为了便于阅读，作者把格式稍作了整理。

```
SELECT id, title, author, content
FROM poem
WHERE id IN (
    SELECT poemid
    FROM reference
    WHERE authorid = 32540 and refid = 3915
)
```

为了便于读者理解，作者把上述 SQL 语句分成两部分在 SQLiteStudio 中执行。图 22-19 展示了后一个 SELECT 子句的执行结果，该执行结果只有 1 个字段 3 行，其数据来源为 reference 表，引用人限定为李白，被引用人限定为杜甫。显然，执行结果中的 3 个整数就是李白引用杜甫的 3 首诗在 poem 表中的 id。

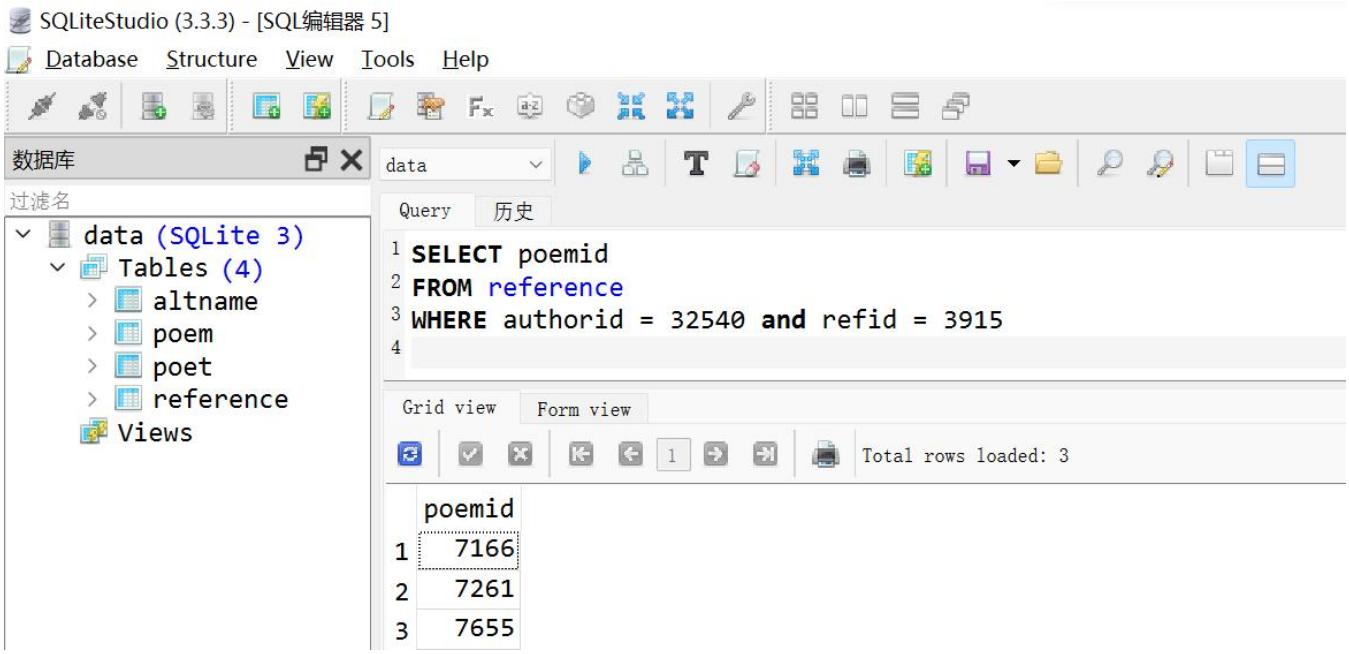


图 22-19 查询指定引用人和被引用人的引用关系中的唐诗 id

接下来，我们用查得的 3 首唐诗的 id 替换掉完整 SQL 语句中的第 2 个 SELECT 子句。查询结果如图 22-20 所示。这个 SQL 语句比较容易理解，它从 poem 表中进行记录筛选，要求 id 值必须是 7166、7261 和 7655 之一，最终的结果数据集包括 id、title、author 和 content 共 4 个字段。综上，前面所示的完整 SQL 语句由两个 SELECT 子句构成，其中，可以认为后面一个 SELECT 子句先执行，其执行结果构成了前一个 SELECT 子句的筛选条件。

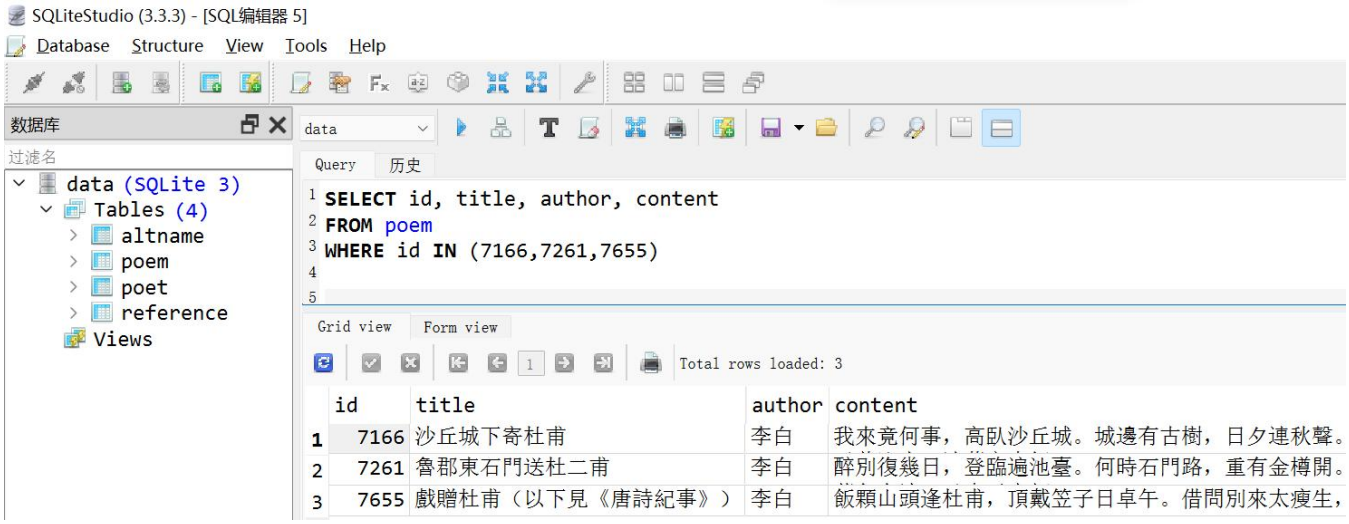


图 22-20 李白引用杜甫的全部唐诗

接下来讨论 reference.cpp 中的 getFriendCycle()函数。

```
26 QVector<Reference> Reference::getFriendCycle(int authorid) {
27     QSqlQuery q;
28     QString sSql = QString(
29         "SELECT authorid,refid,count(*) as cnt FROM reference "
30         "WHERE authorid=%1 or refid=%2 GROUP BY authorid, refid")
31             .arg(authorid).arg(authorid);
32
33     Q_ASSERT(q.exec(sSql));
34
35     QVector<Reference> t;
36     while (q.next()){
37         auto r = q.record();
38         t << Reference(r.field(0).value().toInt(),
39                        r.field(1).value().toInt(),
40                        r.field(2).value().toInt());
41     }
42
43     return t;
44 }
```

如前所述，getFriendCycle()函数用于查询与指定 id 的诗人相关的全部引用关系。同样地，函数先进行 SQL 查询，然后遍历 SQL 查询结果，逐行转换成 Reference 对象，再置入向量后返回。

假设我们查询的是与杜甫有关的全部引用关系，在经过占位符替换后，代码第 29 ~ 31 行生成的 SQL 语句如下所示（格式有调整）。

```
SELECT authorid,refid,count(*) as cnt
FROM reference
WHERE authorid=3915 or refid=3915
GROUP BY authorid, refid
```

在上述 SQL 语句中，被查询表格为 reference（第 2 行）；筛选条件为引用人或者被引用人 id 为 3915（第 3 行）；筛选而得的全部记录按引用人 id 和被引用人 id 进行分组（第 4 行），

即引用人 id 和被引用人 id 相同的记录分在同一组；分组之后的数据取 authorid、refid 和 cnt 三个字段，其中，cnt 的值为 count(*)，表示对应分组的记录数。

相关 SQL 语句在 SQLiteStudio 中的局部执行结果如图 22-21 所示。执行结果的第 2 行表示 id 为 3 702 的诗人引用了 id 为 3 915 的诗人（即杜甫）2 次；执行结果的第 8 行表示 id 为 3 915 的诗人（即杜甫）引用了 id 为 13 060 的诗人 8 次。带着这两个 id 去 poet 表中查询可得，皮日休（3 702）引用了杜甫 2 次，杜甫引用了李世民（13 060）8 次。

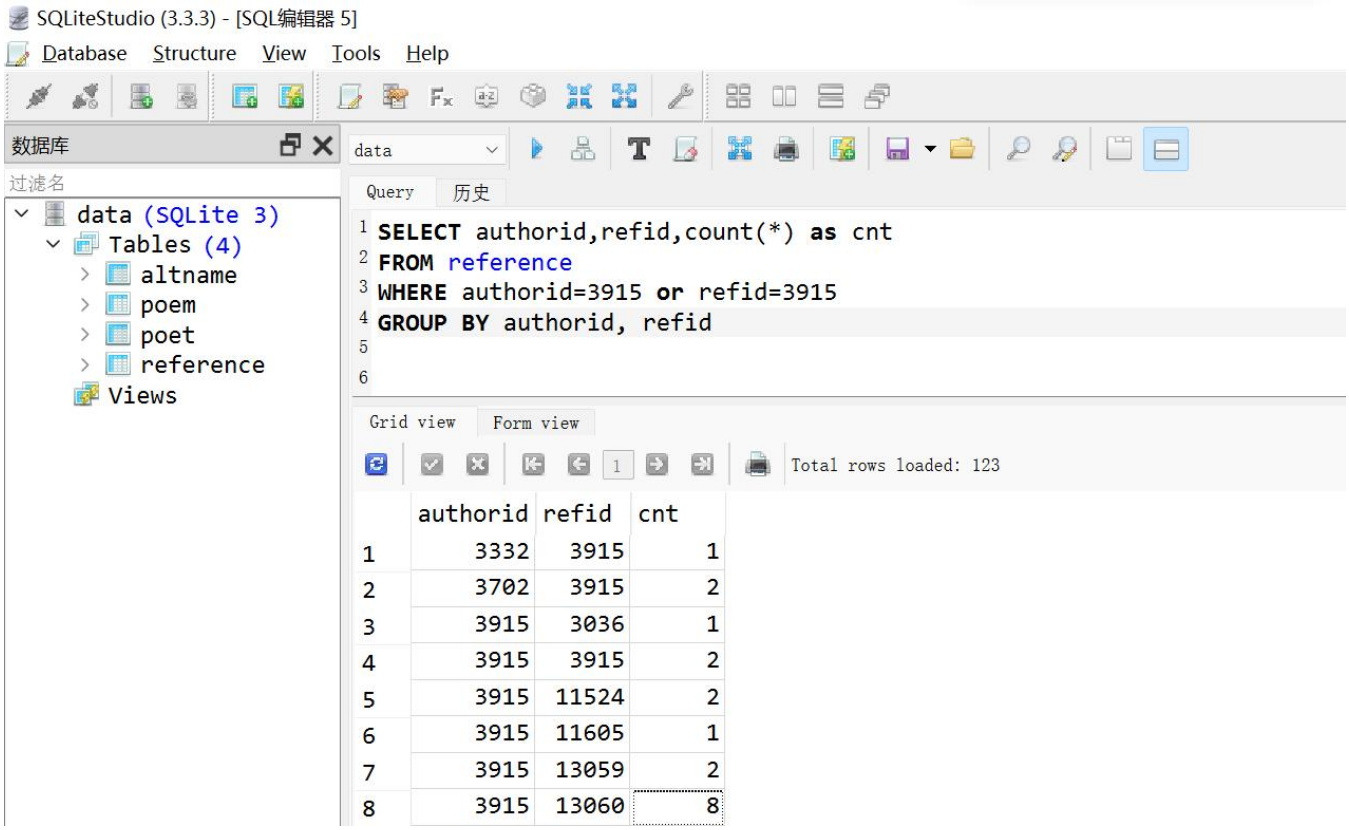


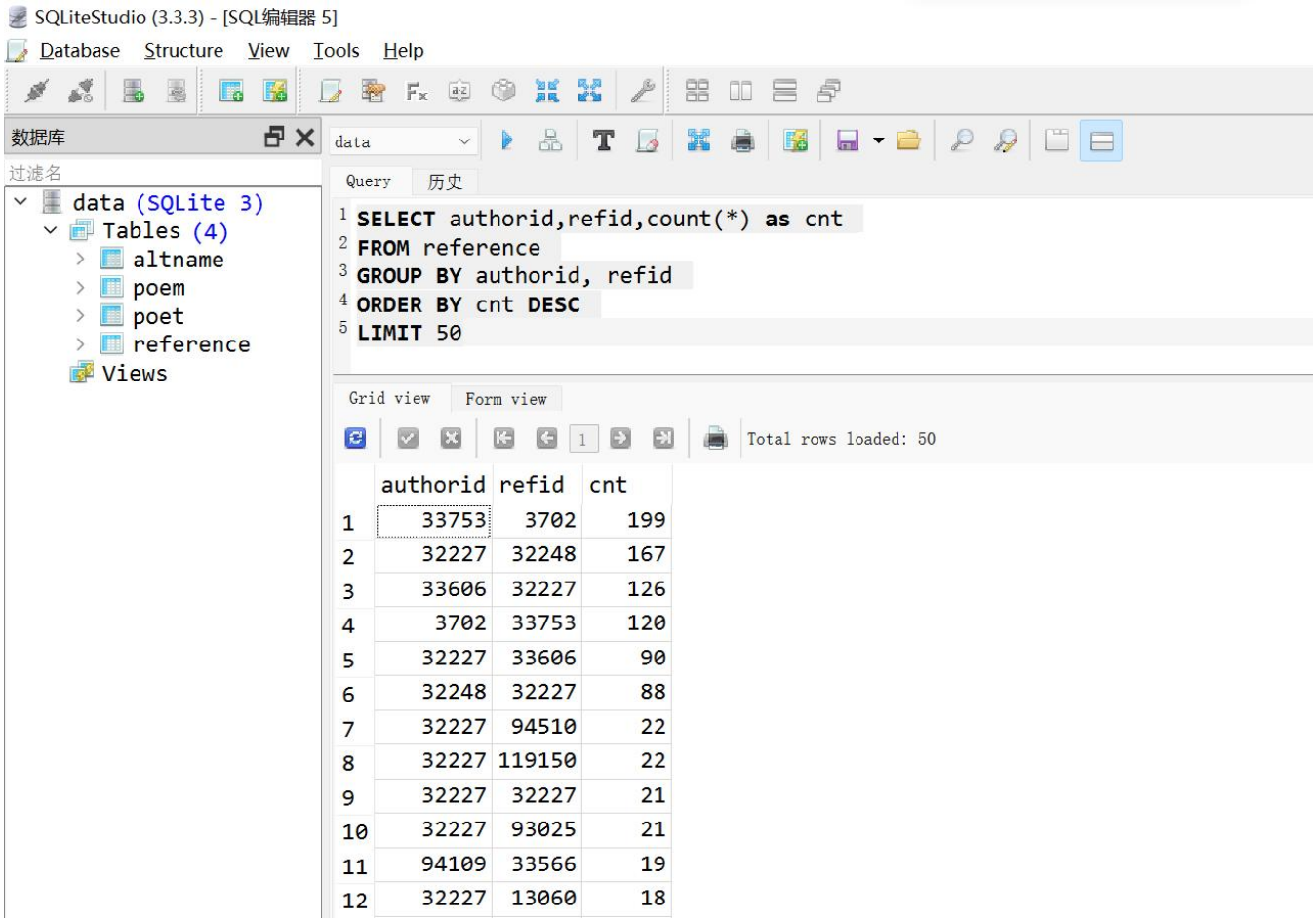
图 22-21 与杜甫相关的引用关系的部分查询结果

最后讨论 reference.cpp 中的最后一个函数 getReferences()。

```
46 QVector<Reference> Reference::getReferences(int iLimit) {
47     QSqlQuery q;
48     QString sSql = QString(
49         "SELECT authorid,refid,count(*) as cnt FROM reference "
50         "GROUP BY authorid, refid ORDER BY cnt DESC LIMIT %1").arg(iLimit);
51
52     Q_ASSERT(q.exec(sSql));
53
54     QVector<Reference> t;
55     while (q.next()){
56         auto r = q.record();
57         t << Reference(r.field(0).value().toInt(),
58                        r.field(1).value().toInt(),
59                        r.field(2).value().toInt());
60     }
61
62     return t;
63 }
```

如前所述，getReference()函数用于查询唐代诗人引用数量最多的前 iLimit 对。函数的执行过程和原理同前两个函数类似，我们重点讨论 SQL 语句。假设参数 iLimit 为 50，则相关 SQL 语句及其在 SQLiteStudio 中的执行结果如图 22-22 所示。

相关 SQL 语句中，被查询的表格为 reference；筛选条件无；全部记录按 authorid 以及 refid 进行分组；字段 cnt 值为 count(*)，表示每组中的记录条数；分组统计数据按照 cnt 降序排序（第 4 行）；只取全部查询结果的前 50 行（第 5 行）。



	authorid	refid	cnt
1	33753	3702	199
2	32227	32248	167
3	33606	32227	126
4	3702	33753	120
5	32227	33606	90
6	32248	32227	88
7	32227	94510	22
8	32227	119150	22
9	32227	32227	21
10	32227	93025	21
11	94109	33566	19
12	32227	13060	18

图 22-22 唐代诗人引用关系前 50 强

从图 22-22 可以看到，编号为 33 753 的诗人引用编号 3 702 的诗人 199 次，按 SQL 语句“SELECT * FROM poet WHERE id in (3702,33753)”进行查询可知，这对好朋友是陆龟蒙（~881）和皮日休（834 ~ 883）。

22.4.2 引用关系的超文本展现

我们对主窗口中的 pbQueryReference 按钮添加了如下的槽函数，该函数在操作者单击并释放“引用查询”按钮后被调用。

```
1 void MainWidget::on_pbQueryReference_released() {
2     Poet poet1;
3     if (!Poet::getPoetByName(ui->leRefPoet1->text().trimmed(),poet1))
4         return;
5     Poet poet2;
6     if (!Poet::getPoetByName(ui->leRefPoet2->text().trimmed(),poet2))
```

```
 7          return;
 8
 9      QString sHtml;
10      auto poems1 = Reference::getReferencePoems(poet1.iId,poet2.iId);
11      sHtml += "<table><tr><td bgcolor='#e8f4fe'>";
12      sHtml += toHtml(poet1);
13      sHtml += QString("<font size='4' color=#369ff3><center><br>"
14                       "%1 在%2 首诗中提到了%3<br></center></font>")
15              .arg(poet1.sName).arg(poems1.size()).arg(poet2.sName);
16      for (auto& m:poems1)
17          sHtml += toHtml(m);
18      sHtml += "</td><td bgcolor='#fcfdf8'>";
19
20      auto poems2 = Reference::getReferencePoems(poet2.iId,poet1.iId);
21      sHtml += toHtml(poet2);
22      sHtml += QString("<font size='4' color=#369ff3><center><br>"
23                       "%1 在%2 首诗中提到了%3<br></center></font>")
24              .arg(poet2.sName).arg(poems2.size()).arg(poet1.sName);
25      for (auto& m:poems2)
26          sHtml += toHtml(m);
27      sHtml += "</td></tr></table>";
28
29      ui->textBrowser->setHtml(sHtml);
30  }
```

第 2 ~ 4 行：从单行输入框 leRefPoet1 获取诗人 1 的姓名，并通过 Poet::getPoetByName()函数在《中国历代人物传记》中进行比对，如果无法准确定位到某位唐代诗人，放弃并返回。

第 5 ~ 7 行：使用相同方法比对诗人 2。

第 10 行：查询并得到诗人 1 引用诗人 2 的唐诗向量。

第 11 行：为 sHtml 字符串添加标记，这些标记属于超文本标记语言（Hyper Text Markup Language）的范畴，其简单解释见表 22-9。

第 12 行：toHtml()函数将诗人 poet1 转换成 HTML 格式的字符串，然后附加到 sHtml。

第 13 ~ 15 行：向 sHtml 字符串附加诗人 1 引用诗人 2 的摘要信息。

第 16 ~ 17 行：对诗人 1 引用诗人 2 的唐诗向量 poems1 进行遍历，使用 toHtml(m)函数逐一将唐诗对象转换成 HTML 格式的字符串，然后附加到 sHtml。

第 18 行：向 sHtml 字符串添加标记。

第 20 ~ 27 行：重复上述过程，向 sHtml 添加诗人 2 引用诗人 1 的唐诗及相关信息。

第 29 行：将字符串 sHtml 填入 textBrower。如前所述，textBrower 是文本浏览器，具备一定的超本文解析和展示能力。

前述代码中用到和 toHtml()实际上是两个函数名重载的同名函数，其分别完成 Poet 和 Poem 对象的 HTML 字符串转换工作。两个函数的代码如下，请读者结合表 22-9 列出的 HTML 标记综合理解。

```
1  QString MainWidget::toHtml(const Poet& t) {
2      QString s = t.sName + "( ";
3      for (auto& x:t.altNames)
```

```
        s = s + x + " ";
    s += ")";
    return QString("<center><font size='4'>%1</font><br>"
                   "<font size='2'>%2 ~ %3</font></center><hr>")
           .arg(s).arg(t.iBirthYear).arg(t.iDeathYear);
}

QString MainWidget::toHtml(const Poem& m) {
    QString t;
    t += QString("<center><h3>%1</h3></center>").arg(m.sTitle);
    t += QString("<center>%1</center>").arg(m.sAuthor);

    QString c = m.sContent;
    c.replace("。","。<br>");
    t += QString("<center>%1</center>").arg(c);
    return t;
}
```

表 22-9 案例中使用到的 HTML 标记说明

标记	说明
<table></table>	成对标记，中间的内容为一个表格。
<tr></tr>	成对标记，中间的内容为表格中的一行，tr 是 table row 的首字母简写。
<td></td>	成对标记，中间的内容为一个表格行中的一个单元格。
 	单一标记，表示内容换行。
	成对标记，中间的内容居中显示。
	成对标记，改变中间内容的字体。
<hr>	单一标记，显示一根分隔横线。
	成对标记，中间的内容按 3 号标题显示。

至此，引用查询功能完成。执行效果请回顾图 22-18。

22.5 朋友圈

图 22-23　单位诗人的朋友圈

如图 22-23 所示，在主窗口的 lePoetFriendCycle 单行编辑框中输入杜甫，然后单击“朋友圈”按钮，程序会打开浏览器，显示出如图 22-24 所示以杜甫为中心的朋友圈，看起来，杜圣是还是位社交达人。请读者注意图中顶部的文件名，该扩展名为 html 的超文本标记语言文件，就是俗称的网页文件，是由程序生成并保存至硬盘的。

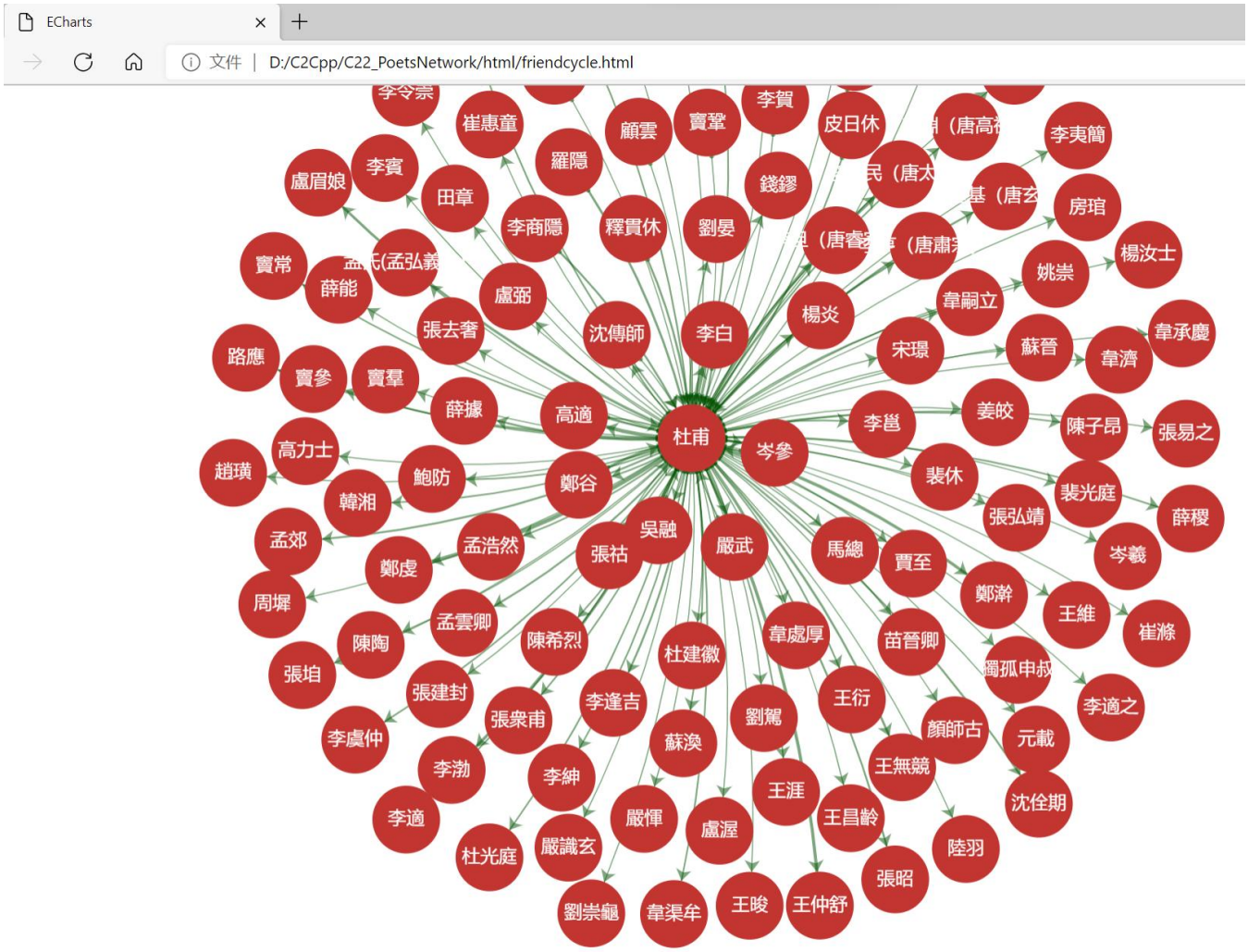


图 22-24　杜甫的朋友圈

在 mainwidget.cpp 中，“朋友圈”按钮的槽函数如下所示。

```
1  void MainWidget::on_pbFriendCycle_released() {
2      Poet m;
3      if (!Poet::getPoetByName(ui->lePoetFriendCycle->text().trimmed(),m)){
4          ui->textBrowser->setHtml(
5                      "<h3>诗人姓名不存在，注意数据库存储的姓名为繁体中文.</h3>");
6          return;
7      }
8
9      QVector<Reference> r = Reference::getFriendCycle(m.iId);
10     QString sFileName = DBHelper::sProjectPath + "/html/friendcycle.html";
11     writeHtmlFile(sFileName,toHtml(r));
12
13     QStringList t;
14     t << sFileName.replace("/","\\");
15     QProcess::startDetached("explorer",t);
16 }
```

第 2 ~ 7 行：从单行输入框 lePoetFriendCycle 读取操作者输入的诗人姓名，然后使用 Poet::getPoetByName()进行唐代诗人比对，如果比对失败，在 textBrower 中显示错误信息并返回。

第 9 行：使用 Reference::getFriendCycle()获取该诗人的全部引用关系，该函数代码细节见 22.4 节。

第 10 行：朋友圈的显示是通过浏览器实现的，程序需要往硬盘写入一个 html 格式的文件，本行代码生成该文件的完整路径。

第 11 行：调用 toHtml(r)将引用关系转换成 HTML 字符串，再经由 writeHtmlFile()将其写入到指定的文件。

第 13 ~ 16 行：使用 QProcess::startDetached()函数开始一个分离的外部进程，打开浏览器。“explorer”是浏览器的可执行文件名，QStringList 类型的变量 t 是命令行参数，它是一个容器，其包含了拟浏览的 HTML 文件在硬盘上的路径。

第 14 行：将文件路径放入容器 t 之前，提前将其中的“/”修改为“\”。Windows 操作系统使用“\”做为目录分隔符，如果读者使用的是 Linux 系统，则不应做这种替换。

接下来我们讨论上述代码中用到的 toHtml()函数的下述函数名重载版本，该函然将引用关系向量转换一个 HTML 字符串。这个生成出来的字符串将由 writeHtmlFile()函数写入文件。

```
1  QString MainWidget::toHtml(const QVector<Reference>& n) {
2      auto sHtmlHead = readFile(DBHelper::sProjectPath + "/html/html_head.txt");
3      auto sHtmlTail = readFile(DBHelper::sProjectPath + "/html/html_tail.txt");
4
5      QString sLinks = "links: [\n";
6      QString sLinkFormat =
7              "{source:'%1',target:'%2',lineStyle:{normal:{width: %3}}},";
8      QSet<QString> authorNames;
9      for (auto& x:n){
10         auto sAuthorName = Poet::getPoetNameById(x.iAuthorId);
11         auto sRefName = Poet::getPoetNameById(x.iRefId);
```

```
        sLinks += sLinkFormat.arg(sAuthorName).arg(sRefName).arg(sqrt(x.iCount));
        authorNames << sAuthorName <<  sRefName;
    }
    sLinks += "],\n";

    QString sNodes = "data:[\n";
    QString sItemFormat = "{name: '%1'},\n";
    for (auto& x:authorNames)
        sNodes += sItemFormat.arg(x);
    sNodes += "],\n";

    return sHtmlHead + sNodes + sLinks + sHtmlTail;
}
```

可以看到，整个 HTML 文件的内容由文件头（sHtmlHead）、文件尾(sHtmlTail)、链接（sLinks）和节点（sNodes）四个部分组成。该 HTML 文件使用了一个由 Java Script 编写的第三方库来渲染关系网络，字符串 sNodes 用于描述网络的节点构成，字符串 sLinks 则用于描述网络中节点间的连接。文件头和文件尾则包含了 HTML 文件的一些必要的语法构成部分，且提供了对第三方 Java Script 库的引用，它们都是从文件中读取出来的。

```
QString MainWidget::readFile(const QString filename) {
    if (!(QFile::exists(filename))) {
        qDebug() << "Error: missing file... " << filename;
        return "";
    }

    QFile f(filename);
    Q_ASSERT(f.open(QIODevice::ReadOnly));
    QTextStream fs(&f);
    fs.setEncoding(QStringConverter::Utf8);
    auto s = fs.readAll();
    f.close();
    return s;
}
```

readFile()函数则用于读取指定文件的全部内容，返回值类型为字符串。

```
void MainWidget::writeHtmlFile(const QString& filename,
                               const QString& sContent) {
    QFile f(filename);
    Q_ASSERT(f.open(QIODevice::WriteOnly));
    f.write(sContent.toUtf8());
    f.close();
}
```

writeHtmlFile()将字符串 sContent 写入文件 filename。请读者注意，无论是读文件文件还是写文本文件，我们都使用了 UTF-8 编码。

22.6 可视化关系网络

在主窗口的下方，还有四个分别名为“关系网络（50）”、“关系网络（100）”、“关系网络（200）”、“关系网络（500）”的按钮，其预期用于浏览诗人关系网络。由于唐代诗人

数量多，引用关系复杂，括号内的数字代表了对要浏览的关系网络的限制，比如 100 表示在该关系网络中，仅包含数量前 100 名的引用及其相关作者。

```
void MainWidget::on_pbNetwork50_released() {
    exploreNetwork(50);
}

void MainWidget::on_pbNetwork100_released() {
    exploreNetwork(100);
}

void MainWidget::on_pbNetwork200_released() {
    exploreNetwork(200);
}

void MainWidget::on_pbNetwork500_released() {
    exploreNetwork(500);
}
```

如上述代码所示，我们给这四个按钮添加了几乎相同的槽函数，这些槽函数都调用执行了 exploreNetowrk()。

```
void MainWidget::exploreNetwork(int iLimit) {
    QVector<Reference> r = Reference::getReferences(iLimit);
    QString sFileName = DBHelper::sProjectPath +
            QString("/html/network%1.html").arg(iLimit);
    writeHtmlFile(sFileName,toHtml(r));

    QStringList t;
    t << sFileName.replace("/","\\");
    QProcess::startDetached("explorer",t);
}
```

第 1 行：执行 Reference::getReferences()获取数量前 iLimit 的引用关系向量。相关代码细节请见 22.4 节。

第 3 ~ 5 行：将引用关系转换成 HTML，并写入文件。

第 7 ~ 9 行：打开外部浏览器渲染关系网络。

至此，本实践任务全部完成。图 22-25 展示了包含 200 行引用关系的唐代诗人关系网络的局部。在浏览器里，读者可以用鼠标按住诗人的姓名，然后拖动观察。在该网络中，诗人与诗人之间的连线的粗细表示引用的次数。可见，白居易与元稹，白居易与刘禹锡过从甚密。陆龟蒙和皮日休两位相互疯狂引用，但与其他诗人联络却相对较少。读者如果拖一下白居易和刘禹锡，可以看到他们两位以及元稹，几乎在中唐诗人中处于核心地位，属于带头大哥级别。熟悉唐诗的朋友可能知道，李杜时期的盛唐诗人以浪漫主义为主，而白居易、元稹为代表的中唐诗人逐渐转向现实主义。

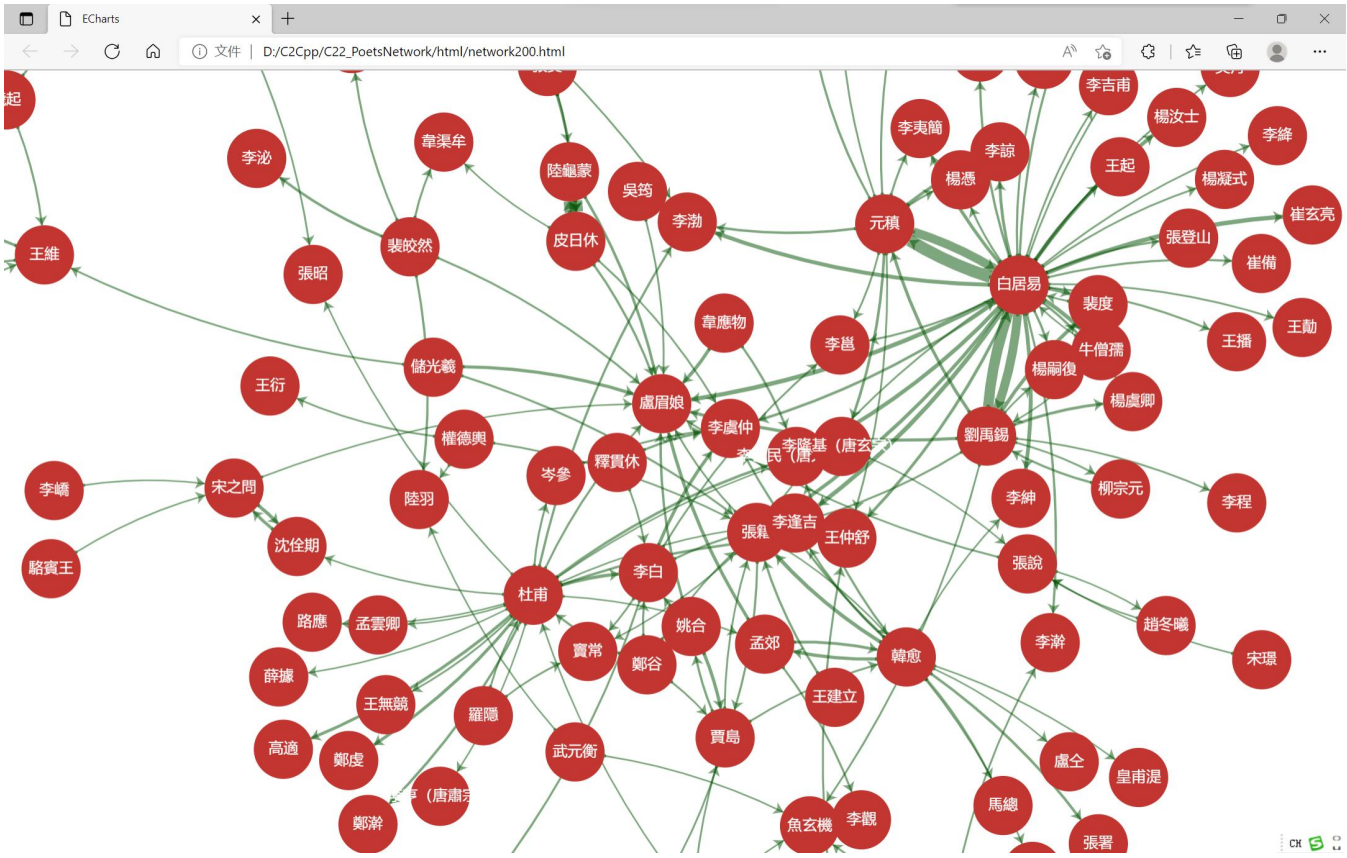


图 22-25 可视化诗人关系网络局部