

实验三 网络数据获取与显示

一、实验目的

进一步熟悉使用QtCreator进行程序开发和调试的基本方法；

掌握使用界面编辑器进行用户界面的创建、布局方法；

掌握文本文件的读取和写入方法，

利用QNetworkAccessManager类访问网络和利用QXmlStreamReader类进行简单HTML页面解析的基本方法；

掌握图表控件简单使用方法。

二、实验软件

- Qt 5.7.0
- MinGW 5.3.0 32bit
- Windows 7 系统及以上

三、实验任务

编写一个具备基本网络访问能力，获取网络数据并将数据绘制成图表的简单数据处理程序。用户可以选择需要显示数据的时间、设定图注风格及数据点显示风格。同时，获得的数据以时间为文件名的文本文件保存于程序所在的数据文件夹data路径下。当选定时间的数据文件已经存在时，直接通过解析本地文件绘制图表。

程序基本界面如图1所示：

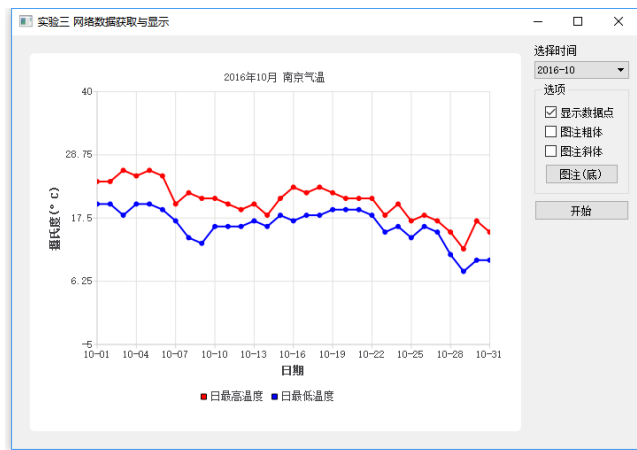


图 1 网络数据获取与显示

四、实验原理与说明

1. 使用 Qt Creator 界面编辑器创建用户界面

Qt Creator 的界面编辑器可以很方便的创建用户界面并布局，大大减少使用代码创建用户界面的工作量。对一个实际工程来说，采用界面编辑器制作的用户界面可能无法完全满足需求，此时需要使用代码对界面进行进一步的精细控制。使用代码创建用户界面，详见实验二内容。

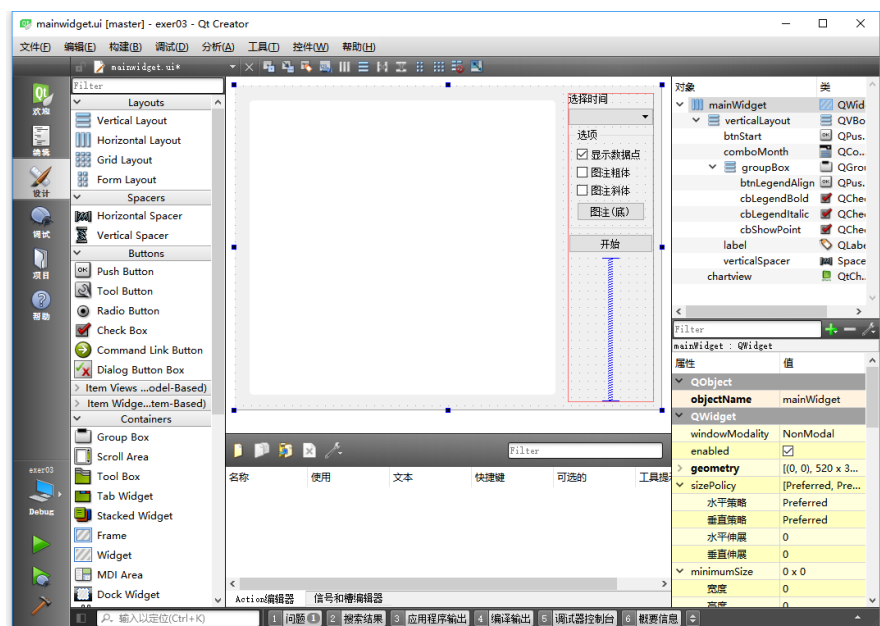


图 2 界面编辑器

Qt Creator 的界面编辑器如图 2 所示，使用界面编辑器创建用户界面的一般流程如下：

首先，使用模板创建一个带有 ui 的基于 QWidget（或 QMainWindow）的应用程序。

其次，双击 ui 文件进入界面编辑器。

最后，按照需要进行具体的界面设计。

2. 网络访问与 HTTP 协议

(1) QNetworkAccessManager 类与网络访问

随着网络对人们日常生活的渗透，具备访问网络、从网络获取数据的应用程序非常常见。

Qt 的网络模块提供了完整的网络访问功能，并且使用起来非常方便。在使用网络模块时，必须要在工程文件中添加网络模块，即打开工程的.pro 文件，添加“QT += network”语句。本实验中，我们将使用 Qt 的网络模块进行最基本的网络访问操作。

网络访问的核心类是 QNetworkAccessManager 类，Qt 的整个访问网络 API 都是围绕这个类进行的。QNetworkAccessManager 类允许应用程序发送网络请求以及接受服务器的响应，保存发送请求的基本配置信息，包括代理和缓存等的设置。该类设计为异步工作模式，

在进行网络访问时，不需要为其开启线程，防止界面锁死。

关于线程的相关内容，本课程限于时间关系并不涉及，基本说明如下：

每个应用程序在运行时都会存在于一个线程内，一般称之为主线程。Qt 规定 GUI 界面相关操作必须在主线程中进行。当程序在处理一个如访问网络、大规模计算等较为耗时的工作时，如果该工作位于主线程内进行，由于程序尚未处理完成，此时用户界面无法对用户操作进行及时反应（也称为主线程被阻塞），即界面被锁死。从人机交互的角度来说，这种设计非常不友好。QNetworkAccessManager 类使用异步工作模式的这种设计避免了主线程阻塞等一系列问题，但要求使用更多的代码来监听返回（使用信号槽方式实现）。

一个应用程序仅需要一个 QNetworkAccessManager 类的实例，应避免创建多个实例对象。创建完成后，我们就可以使用它发送网络请求。这些请求都返回 QNetworkReply 对象作为响应。这个对象一般会包含有服务器响应及返回等数据。请求完成时，QNetworkAccessManager 对象发射 finished 信号，表示网络请求已经结束。我们可以在该信号的槽函数中对服务器返回的数据进行读取和处理。

(2) HTTP 协议

HTTP（超文本传输协议）是一个基于请求与响应模式的、无状态的、应用层的协议，常基于 TCP 的连接方式，HTTP1.1 版本中给出一种持续连接的机制，绝大多数的 Web 应用都是构建在 HTTP 协议之上。

HTTP 链接的格式如下：`http://host[":"port][abs_path]`。当我们访问一个网站时，前缀 `http` 表示该协议为 HTTP 协议，`host` 为一个合法的 Internet 主机域名或者 IP 地址；`port` 表示访问端口（省略时为默认值 80），`abs_path` 指定请求资源的 URI；如果 URL 中没有给出 `abs_path`，那么当它作为请求 URI 时，必须以“/”的形式给出，通常这个工作浏览器自动帮我们完成。

HTTP 请求由请求行、消息报头、请求正文三部分组成，命令格式如下：

```
Method Request-URI HTTP-Version CRLF
```

其中 Method 表示请求方法；Request-URI 是一个统一资源标识符（即访问页面的相对路径）；HTTP-Version 表示请求的 HTTP 协议版本；CRLF 表示回车和换行。

请求方法有多种（必须为大写），如下列出几个常见方法：

GET	请求获取 Request-URI 所标识的资源
POST	在 Request-URI 所标识的资源后附加新的数据
HEAD	请求获取由 Request-URI 所标识的资源的响应消息报头
PUT	请求服务器存储一个资源，并用 Request-URI 作为其标识
DELETE	请求服务器删除 Request-URI 所标识的资源

对于普通的 HTTP 应用来说,最为常见的是 GET 和 POST。前者用于从服务器获取数据,后者用于向服务器发送数据(一般是用户参数、认证等小型数据)。

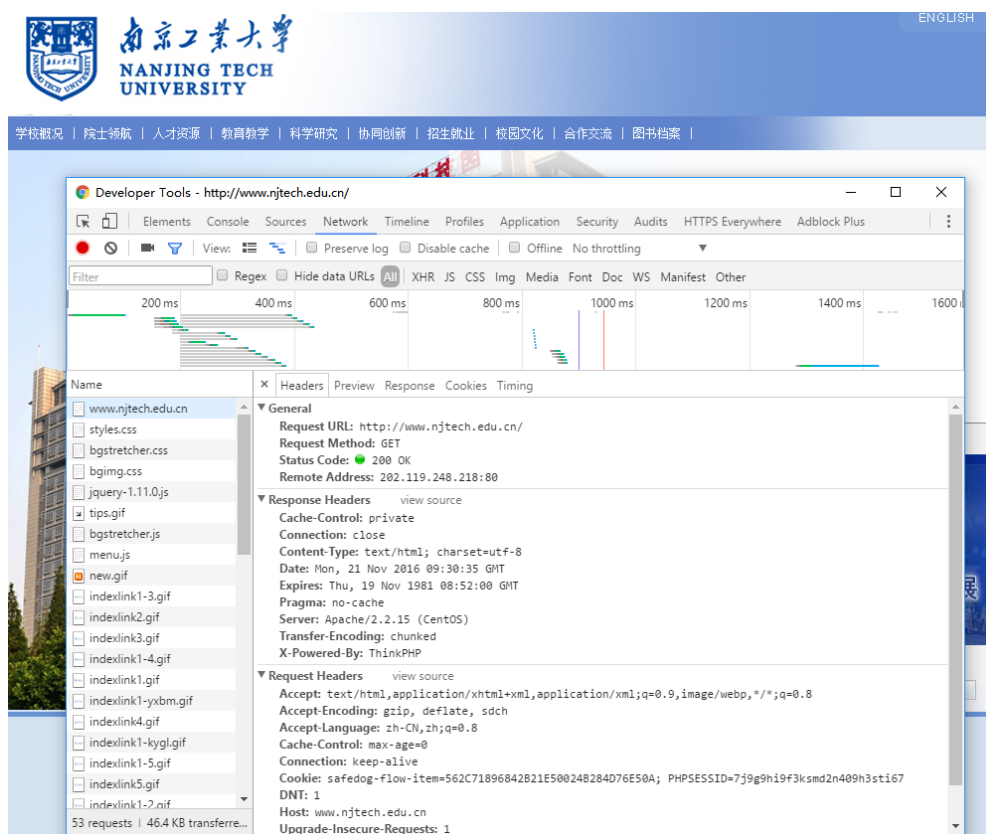


图3 使用 Chrome 的开发者工具显示 HTTP 请求与响应数据

观察 HTTP 请求与响应数据,可使用浏览器自带的开发人员工具(IE 11)或开发者工具(Google Chrome),在 Firefox 浏览器中有著名的 Firebug 插件。上述工具可以显示详细的请求数据以及服务器的应答和返回数据。如图 3 所示为使用 Chrome 的开发者工具显示的访问 <http://www.njtech.edu.cn> 页面的 HTTP 请求与响应数据。图 3 中,Headers 页面下为 HTTP 请求头数据(Request Headers)和服务器的响应头数据(Response Headers),General 栏目下为请求地址、方法和状态(Request URL、Request Method 和 Status Code)。在 Response 页面下为服务器根据 HTTP 请求所发回的响应数据,这里就是南京工业大学主页的 HTTP 代码。

对于一个 HTTP 请求,服务器会根据请求结果,返回相应的状态码。状态码由三位数字构成,可分为 5 大类别,如下所示:

- 1xx: 指示信息--表示请求已接收,继续处理
- 2xx: 成功--表示请求已被成功接收、理解、接受
- 3xx: 重定向--要完成请求必须进行更进一步的操作
- 4xx: 客户端错误--请求有语法错误或请求无法实现
- 5xx: 服务器端错误--服务器未能实现合法的请求

常见状态代码、状态描述、说明:

200 OK	//请求成功, 服务器正确返回请求的资源
400 Bad Request	//客户端请求有语法错误, 不能被服务器所理解
401 Unauthorized	//请求未经授权, 该状态代码必须和 WWW-Authenticate 报头域一起使用
403 Forbidden	//请求收到, 但服务器拒绝提供服务
404 Not Found	//请求资源不存在
500 Internal Server Error	//服务器错误
503 Server Unavailable	//服务器无法处理请求

关于 HTTP 请求的详细内容, 本实验不作说明, 请自行参考其他资料。

(3) 使用 QNetworkAccessManager 发起 HTTP 请求, 并获得网页数据

使用 QNetworkAccessManager 实现 HTTP 访问的方法比较简单。其主要步骤有 3 个:

第一、创建 QNetworkAccessManager 实例对象, 连接 QNetworkAccessManager 的 finished 信号与请求结束时的数据处理槽函数;

第二、组装访问请求, 使用 QNetworkAccessManager 相应的函数发起 HTTP 请求 (如发起 HTTP 的 GET 请求时, 使用 get 函数; 发起 HTTP 的 POST 请求时, 使用 post 函数)。

第三、在处理槽函数中对获得的数据进行处理 (包括访问错误处理、返回数据处理等)

如下代码片段显示了如何创建一个 QNetworkAccessManager 对象、连接相应的信号与槽并发起 HTTP 请求 (使用 GET 方法)。

```
manager = new QNetworkAccessManager(this);
connect(manager, &QNetworkAccessManager::finished, this,
        &dataWorker::httpsFinished);

manager->get(QNetworkRequest(QUrl("http://www.njtech.edu.cn")));
```

在响应槽函数中, QNetworkReply 对象包括了服务器返回的响应头和响应数据。通常需要在槽函数中将各种可能出现的错误情况排除, 在获得正确的响应数据后, 对数据按照需要进行处理, 槽函数基本结构如下所示。

```
void dataWorker::httpsFinished(QNetworkReply *reply)
{
    if (reply->error()) {
        qDebug() <<reply->errorString();
        reply->deleteLater();
        reply = Q_NULLPTR;
        return;
    }
    int v =
reply->attribute(QNetworkRequest::HttpStatusCodeAttribute).toInt();
    if(v != 200) {
        qDebug() <<"HTTP 返回代码: " <<v;
        reply->deleteLater();
        reply = Q_NULLPTR;
    }
}
```

```

        return;
    }

    qDebug() << "开始读取返回的数据: ";
    QString html = QString::fromLocal8Bit(reply->readAll());

    reply->deleteLater();
    reply = Q_NULLPTR;

    qDebug() << "返回数据长度: " << html.size();

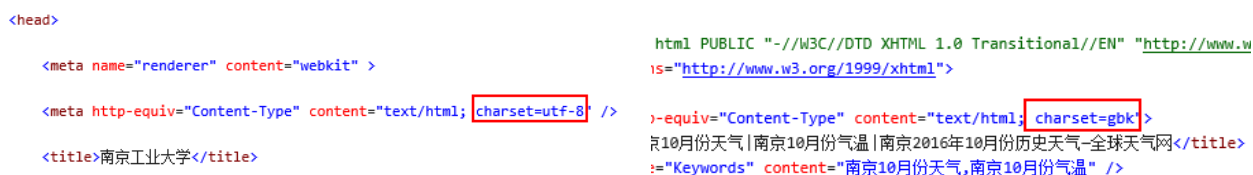
    ...

    parseHTML(html); // 解析获得的 HTML 代码
}

```

QNetworkReply 对象的 error 函数可以返回当前网络访问时发生的问题，以网络故障为主，如网络无法连接、无法找到服务器、网络超时等。HttpStatusCodeAttribute 属性则为服务器返回的状态码。前一节中我们可以看到，状态码为 200 时表示网络请求正常完成。因此这里我们做一个简单判断，对所有未正常完成的请求简单的进行返回，而对正确完成的请求再继续向下进行数据处理。

当所有的请求正确处理后，那么需要对服务器返回的数据进行读取。对于非下载类的简单 HTTP 请求，数据量一般较小，可以使用 readAll 函数，将所有返回数据读入内存。



```

<head>
  <meta name="renderer" content="webkit" >
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>南京工业大学</title>

  html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w
  is="http://www.w3.org/1999/xhtml">
  <meta http-equiv="Content-Type" content="text/html; charset=gbk" />
  <title>南京10月份天气|南京10月份气温|南京2016年10月份历史天气-全球天气网</title>
  <meta name="Keywords" content="南京10月份天气,南京10月份气温" />

```

图 4 网页字符集

对于中文网页的解析来说，有一个经常遇到的问题是网页的字符编码问题。如图 4 所示，HTML 中 meta 标签下的 charset 表明了网页使用的字符集，常用的中文字符集是 UTF-8 和 GBK（或 GB2312）。由于 Qt 5 的默认编码是 UTF-8，因此对使用 UTF-8 字符集的中文网页直接读取不会出现乱码情况，而使用 GBK（或 GB2312）编码的网页则必须进行一次转码处理，由 GBK 编码转为 UTF-8。对中文操作系统来说，可以使用一个简单的 QString::fromLocal8Bit 函数进行。

注意：在使用 QNetworkReply 对象完成数据获取工作后，程序员必须对该对象进行释放。释放时不能直接使用 delete 函数，而必须使用 deleteLater 函数。在 Qt 文档中，对 deleteLater 函数是这样描述：

该函数标记对象为需删除对象，在事件循环结束后，该对象会被删除。

这是一种较为安全的对象删除方法，特别是针对多线程应用程序，新线程的事件循环尚未结束。如果此时直接删除对象，会出现“Fatal Error: Access Violation”内存访问错误的問題，而使用 `deleteLater` 函数之后，对象将在事件循环结束后删除，避免了上述问题。

3. QXmlStreamReader 类与 XML 解析

XML (eXtensible Markup Language) 是一种 World Wide Web Consortium (W3C) 发布的通用的文本标记语言规范，广泛运用于数据交换和数据存储。XML 是 SGML (Standard Generalized Markup Language) 的精简版本，允许用户自行定义所需的标签(tags)。XML 语法规类似于 HTML，都是用标签对标记文本的内容进行描述。两者的最大区别在于：HTML 语言的标签为固定非可扩展标签，用于固有标记描述文档内容，多用于页面内容和外观的显示。比如表示需要显示一个图像，其含义是固定的。相对的，XML 则没有固定的标记，只描述内容的数据形式和结构。

因此，对于一个格式良好（标签成对、标签结构完整且正常）的 HTML 文档，我们可以将其视为一个 XML 文件，利用 XML 的解析器对网页数据进行解析。

在 Qt 中，对 XML 类型文档的解析一般有两类：一类是基于流式读写、使用 QXmlStream 开头的 QXmlStreamReader / QXmlStreamWriter 类，另外一类是基于 DOM 技术的 QDom 开头 QDomXXX 类。在 Qt 5.7.0 版本中，原有的 Qt XML 模块不再维护，QXmlStreamReader 和 QXmlStreamWriter 类移入 Qt Core 模块，同时新增了 Qt XML Patterns 模块（该模块支持用于 XML 数据文件检索的 XQuery 语言，可用于 HTML 页面解析）。本实验采用 QXmlStreamReader 解析 HTML。

QXmlStreamReader 类提供了一个针对良好格式化的基于流方式的 XML 快速解析器，适合于实现一次性解析任务（即在 XML 解析时，文档以流的形式从头到尾顺序遍历并处理，对已读取内容，无法返回解析）。同时为了更加高效的实现页面解析，在原始 HTML 文档中，我们首先做一次初步处理，将无用部分删除，并去除非可视空白字符“\t\r\n”等，只保留需要解析的 HTML 文本块（一个完整的 tag 对内包含的文本，如<div>....</div>块）

```
// 先做一个简单处理，获取包含内容的完整<div>..</div>标签内的文本内容，
// 并滤除其中的空白字符"\r\n\t"
int begin = html.indexOf("<div class=\"tqtongji2\">");
int end = html.indexOf("<div class=\"lishicity03\">");
html = html.mid(begin,end-begin);
html = html.left(html.indexOf("<div style=\"clear:both\">"));
html = html.simplified().trimmed();
```

在 HTML 解析时，采用一个循环来遍历整个 HTML 文档，并将获得标签内部的数据读取出来。

```

// 使用 QXmlStreamReader 解析 Html 文档
QXmlStreamReader reader(sourceText.simplified().trimmed());

QStringList strData;
while (!reader.atEnd()) {
    reader.readNext();
    if (reader.isStartElement()) {           // 起始的 token
        if (reader.name() == "ul"){         // 查找 Html 标签: ul
            strData<<reader.readElementText(
                QXmlStreamReader::IncludeChildElements).trimmed();
        }
    }
}
}

```

每次 QXmlStreamReader 的 readNext()函数调用，解析器都会读取下一个元素，按照下表中展示的类型进行处理。我们通过表中所列的有关函数即可获得相应的数据值。

enum QXmlStreamReader::TokenType

This enum specifies the type of token the reader just read.

Constant	Value	Description
QXmlStreamReader::NoToken	0	The reader has not yet read anything.
QXmlStreamReader::Invalid	1	An error has occurred, reported in <code>error()</code> and <code>errorString()</code> .
QXmlStreamReader::StartDocument	2	The reader reports the XML version number in <code>documentVersion()</code> , and the encoding as specified in the XML document in <code>documentEncoding()</code> . If the document is declared standalone, <code>isStandaloneDocument()</code> returns <code>true</code> ; otherwise it returns <code>false</code> .
QXmlStreamReader::EndDocument	3	The reader reports the end of the document.
QXmlStreamReader::StartElement	4	The reader reports the start of an element with <code>namespaceUri()</code> and <code>name()</code> . Empty elements are also reported as StartElement, followed directly by EndElement. The convenience function <code>readElementText()</code> can be called to concatenate all content until the corresponding EndElement. Attributes are reported in <code>attributes()</code> , namespace declarations in <code>namespaceDeclarations()</code> .
QXmlStreamReader::EndElement	5	The reader reports the end of an element with <code>namespaceUri()</code> and <code>name()</code> .
QXmlStreamReader::Characters	6	The reader reports characters in <code>text()</code> . If the characters are all white-space, <code>isWhitespace()</code> returns <code>true</code> . If the characters stem from a CDATA section, <code>isCDATA()</code> returns <code>true</code> .
QXmlStreamReader::Comment	7	The reader reports a comment in <code>text()</code> .
QXmlStreamReader::DTD	8	The reader reports a DTD in <code>text()</code> , notation declarations in <code>notationDeclarations()</code> , and entity declarations in <code>entityDeclarations()</code> . Details of the DTD declaration are reported in <code>dtdName()</code> , <code>dtdPublicId()</code> , and <code>dtdSystemId()</code> .
QXmlStreamReader::EntityReference	9	The reader reports an entity reference that could not be resolved. The name of the reference is reported in <code>name()</code> , the replacement text in <code>text()</code> .
QXmlStreamReader::ProcessingInstruction	10	The reader reports a processing instruction in <code>processingInstructionTarget()</code> and <code>processingInstructionData()</code> .

当查找到符合要求的标签后，调用 readTextElement 函数会获得当前标签内的所有非标签数据，而处于两个尖括号(<>)内的标签部分并不会被读取。如图 5 所示的 ...标签对内数据，readTextElement 函数会返回“2016-10-01 24 20 阵雨 北风 微风”字符串。分别将这些字符串存入一个字符串列表对象中，分别解析出前三个数据即可完成数据解析工作。

```

<ul>
  <li>
    <a href="http://nanjing.tianqi.com/20161001.html">2016-10-01</a>
  </li>
  <li>24</li>
  <li>20</li>
  <li>阵雨</li>
  <li>北风</li>
  <li>微风</li>
</ul>

```

图 5 一组 ul 标签文本

4. QChart 与数据显示

Qt 5.7.0 中新增了一个专用于图表绘制的 Qt Charts 模块,该模块提供了一套易于使用的图表组件。Qt Charts 模块在架构上使用了 Qt 的图形视图框架(Graphics View Framework),可以很容易地集成到现代用户界面中,实现非常美观图表的绘制。

Qt 的图形视图框架是一个基于元素 (item) 的模型视图架构的框架,由三部分组成:图形项(item)、场景(scene)和视图(view)。场景 scene 负责整个图形的管理, item 是具体的图形单元,视图可以理解为一个观察窗口,能够观察整个图像的场景。本实验不深入说明图形视图框架具体内容,如需进一步了解,请参考 Qt 文档 (Graphics View Framework)。

在 Qt 应用程序中使用 Qt Charts 模块,首先必须在工程文件.pro 文件中添加 charts 模块:

```
QT += charts
```

然后在头文件中加入下面的语句后,就可使用 Qt Charts 模块的各种类库

```
#include <QtCharts>
QT_CHARTS_USE_NAMESPACE
```

常见的 QChart 使用方法有三个步骤:

- (1) 创建一个 QChart 对象和 QChartView 对象,并设置 QChartView 当前的图表为新创建的 QChart 对象;

```
QChart * chart = new QChart();
// 创建一个 QChartView, 并设置当前的 QChart 为 chart
QChartView * view = new QChartView(chart, this);

// 或使用如下方法创建并设置当前的 QChart
QChartView * view = new QChartView(this);
view->setChart(chart);
```

- (2) 创建序列(Series)、坐标轴(Axes),设置该序列的坐标轴、外观和图注(Legend)等,并向序列中添加需要显示的数据;

```
chart->createDefaultAxes();
chart->setBackgroundVisible(true);
chart->setBackgroundPen(QPen(Qt::lightGray)); // 边框
chart->setBackgroundBrush(QBrush(QColor(240, 240, 240))); // 背景
QLineSeries* series = new QLineSeries(); // 创建一个序列
series->append(5, 5); // 添加数据
...
```

- (3) 使用 addSeries 方法将新建的序列添加到 QChart 中。

```
chart->addSeries(series); // 将序列加入图表中
```

QChart 中还有其他一些方法,如 addAxis、setAxisX、setAxisY 等,可对坐标轴及其他一些如背景、图例等图表元素进行配置,详细内容请参阅 Qt 文档。

对于使用 Qt Creator 的界面编辑器创建的应用程序来说，可以省去 QChart 的创建过程，直接调用 `chartView->chart()` 函数来获得当前的图表对象。

5. 文件保存与读取

文件操作是应用程序必不可少的功能，几乎所有的应用程序都要在文件系统中进行文件访问和读写操作。Qt 通过 `QDir`、`QFile` 和 `QFileInfo` 等文件、文件夹相关类对文件进行操作和管理。Qt 中 `QDir` 和 `QFileInfo` 提供了文件/文件夹信息以及文件/文件夹管理等操作，而派生于 `QIODevice` 的 `QFile` 类提供了对文件进行读写操作的能力。`QIODevice` 是进行数据输入/输出(I/O)的抽象基类。

`QDir` 类用于实现对文件夹的相关操作，包括目录路径、文件等信息。`QDir` 能够操作真实的底层文件系统，也能够对程序内资源文件以同样的方式进行操作（内部资源系统以 `"/"` 为根目录），如实现文件夹的创建(`mkdir/mkpath`)、删除(`rmdir/remove`)、重命名(`rename`)、判断路径是否存在(`exists`)等。

需要注意的是，Qt 中的文件路径遵循 Unix/Linux 风格，以斜杠（`"/"`）作为路径分隔符，非 Windows 系统下的反斜杠（`"\"`），例如路径 `D:/Dev/Qt`。

`QFileInfo` 类提供关于文件在文件系统上的相关信息，包括访问权限、是否为文件夹或符号链接（快捷方式），文件的大小和最后修改/读取时间等。同样，`QFileInfo` 也能够获取程序内的资源系统文件信息。如下代码片段说明了，如何判断一个给定路径是否存在、且为文件。

```
// 检查给定路径名称是否存在，且是否为文件
// 当 path 为存在的文件时，返回 true；其余返回 false。
bool fileExists(QString path)
{
    QFileInfo check_file(path);
    return (check_file.exists() && check_file.isFile());
}
```

`QFile` 类是基本文件读写类，提供文件操作功能，如打开文件、关闭文件、刷新文件、读写文件等。`QFile` 类本身提供的文件读写函数功能较为单一，仅能够面向字节数据进行读写。因此，在大多数情况下 `QFile` 类并不单独使用，而是与 `QTextStream` 类（负责文本文件读写）或 `QDataStream` 类（负责二进制文件读写）结合使用。

`QTextStream` 类和 `QDataStream` 类在初始化时，以 `QFile` 指针对象为参数进行构造，将文件作为流输入（保存）/输出（读取）的目标。通过输入输出流操作符 `>>` 和 `<<` 进行读写操作。`QTextStream` 和 `QDataStream` 可以对 C++ 基本类型(`int`、`double`、`char` 等)和 Qt 类型(Qt 的类和 Qt 自定义类等)实现流式读写操作。需要注意的是，在使用这两个类之前文件必须正

确打开。使用 QFile 和 QTextStream 写入文本文件的示例代码如下：

```
QDir dir;
// 首先判断路径是否存在, dataPath 为路径
if( ! dir.exists(dataPath) )
    dir.mkdir(dataPath);

QString fName = QString("%1/sampleFile.txt").arg(dataPath);
// 以文件名为参数, 创建 QFile 对象
QFile f(fName);
if(f.open(QIODevice::WriteOnly|QIODevice::Text)){
    // 以 QFile 指针为参数, 初始化 QTextStream
    QTextStream stream (&f);
    // 循环写入字符串, data 为 QStringList 对象, 是一个字符串列表
    for( QString d : data)
        stream << d <<"\n";
}else{
    qDebug()<<"打开文件错误";
}
```

上述代码中, 首先判断需要保存的目录是否存在, 然后指定 QFile 对象 f 为该路径下的文件 “sampleFile.txt” 并打开该文件。在文件打开时, 设置打开模式为只写、文本模式。当文件成功打开后, 使用 QTextStream 对象 stream 进行文件的写入操作。

当需要进行文件导入时, 我们可以使用类似的方法进行, 示例代码如下:

```
QStringList dataList;
QFile f(fName);

if(f.open(QIODevice::ReadOnly|QIODevice::Text)){
    // 成功打开数据文件, 则由文件中读取
    QTextStream stream (&f);
    while(!stream.atEnd())
        dataList<<stream.readLine();
    // 数据导入完成,
    // 后续数据处理...
}
```

如果文件没有成功打开, 则不执行操作。

五、系统设计

1. 创建用户界面

本实验中, 根据需要分别拖入 1 个 QLabel、1 个下拉框(QComboBox)、1 个分组框(QGroupBox), 3 个复选框(QCheckBox)、2 个按钮(QPushButton)、1 个图表视图(QChartView) 和一个垂直布局弹簧。如图 6 所示:



图 6 使用编辑器设计用户界面

对上一步骤中拖入的控件设置属性、进行命名并布局。

控件名称	控件类型	控件用途	控件名称	控件类型	控件用途
btnStart	QPushButton	开始按钮	comboMonth	QComboBox	选取数据时间
btnLegendAlign	QPushButton	图注对齐按钮	默认	QLabel	显示“选取时间”
cbShowPoint	QCheckBox	显示数据点	默认	Spacer	垂直弹簧
cbLegendBold	QCheckBox	设置图注为粗体	chartView	QChartView	图表显示窗口
cbLegendItalic	QCheckBox	设置图注为斜体	默认	QGroupBox	选项群组

布局时，首先框选需要布局的控件。然后使用水平、垂直或表格布局对界面进行设计。本实验中，第一步先将 3 个复选框和图注按钮拖入选项群组 QGroupBox 控件中，再单击 QGroupBox，对其使用垂直布局。第二步，框选图 5 中除图表窗口外的其他控件，并对其使用垂直布局。最后，单击界面空白处，对整个界面使用水平布局。最终的布局如图 7 所示。

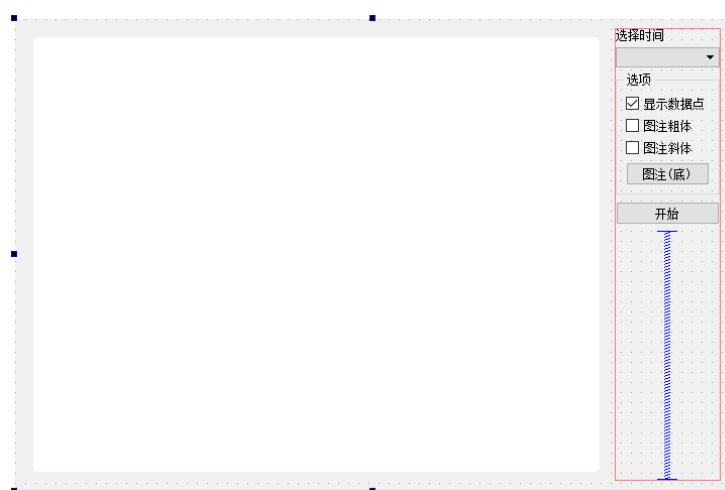


图 7 用户界面布局

2. 程序结构设计

本程序引入一个专门负责数据获取与处理的 dataWorker 类，该类需要响应有窗口界面

mainWidget 类的信号，以进行网络访问/本地数据文件载入、数据解析等工作。主窗口 mainWidget 类为 QWidget 窗口类的派生类，是整个程序的主窗口界面，该类主要负责用户交互与数据展示部分，包括用户点击、选择以及图表绘制后展示等。类基本结构如图 8 所示。



图 8 本程序类派生关系

dataWorker 类是本程序后台数据处理的自定义核心功能类，需要响应来自 mainWidget 窗口类的信号并将数据处理结果以信号形式发送出去提醒界面更新。

在进行程序编写时，很多功能需要自定义类实现。Qt 中很容易实现具有 Qt 特性的自定义类。按照实际用途，自定义类可划分为需要使用信号槽机制的功能类，以及作为信号槽函数参数或文件存储的数据类两大部分。

对于功能类，如果实现接收和发射信号，必须满足以下条件：

- (1) 该类必须派生于 QObject 类或其子类；
- (2) 在类声明的私有区声明 Q_OBJECT 宏
- (3) 使用元对象编译器(Meta-Object Compiler, moc)对程序进行预处理。

实际上，上述条件是自定义类能够使用 Qt 元对象系统所提供特性的必要条件。Qt 中的元对象系统提供了如下特性：

- (1) 支持 QObject 及其子类的对象间信号槽通信机制；
- (2) 返回与该类绑定的元对象 (QObject::metaObject 函数)；
- (3) 在运行时获得当前类名 (QMetaObject::className 函数)；
- (4) 返回类的继承信息，即该类在 QObject 类派生树上的实例 (QObject::inherits 函数)；
- (5) 提供国际化支持 (QObject::tr 函数)；
- (6) 构造该类的一个新实例 (QMetaObject::newInstance 函数)；
- (7) 动态类型转换 (qobject_cast 函数)；
- (8) 动态设置或获取对象的属性。

在满足上述 3 个基本条件后，我们可以在 Qt 应用程序中使用自定义的信号和槽。自定义信号必须以关键字 signals 声明，是一个以 void 为返回类型的函数声明（注意不需要实现该函数的定义）；自定义槽函数必须以 slots 声明，与普通成员函数一样，槽函数具有 public、private、protected 访问限定，以及完整的函数定义；

自定义信号与槽和普通信号槽在使用方面没有区别：首先，使用 `connect()` 函数连接信号和槽；其次，在需要的位置使用关键字 `emit` 发射信号。如下代码显示了 `dataWorker` 类的声明。

```
class dataWorker : public QObject
{
    Q_OBJECT
public:
    explicit dataWorker(QObject *parent = 0);
    explicit dataWorker(QString date, QObject *parent = 0);
    void setRequestDate(QString newDate);
    QString requestDate();
    void doRequest();

protected:
    QString requestUrl();
    void httpGet(QString url);
    void initNetwork();
    void parseHTML(const QString sourceText);
    void parseData(const QString sourceText);
    void exportDataToFile(const QString dataText);

protected slots:
    void httpsFinished(QNetworkReply *reply);

private:
    QNetworkAccessManager *manager;           //!< 网络访问管理类对象
    QString _requestDate;                     //!< 请求年月

    QList<QDateTime> dataDate;                 //!< 日期
    QList<qreal> dataHigh;                     //!< 最高温度
    QList<qreal> dataLow;                      //!< 最低温度

    const QString splitter;                   //!< 数据分隔符
    const QString dataPath;                   //!< 数据保存路径

signals:
    /**
     * @brief 数据解析完成信号
     * @param date 所获取数据年月列表
     * @param high 所获取数据最高温度列表
     * @param low 所获取数据最低温度列表
     *
     * 该信号在数据解析完成，将解析的数据以 3 个列表（QList）的形式作为信号参数发射，<br/>
     * 提醒界面开始更新图表数据。
     */
    void dataParseFinished(QList<QDateTime> date, QList<qreal> high,
        QList<qreal> low);
```

对于数据类，如果希望该自定义类能够作为信号槽函数的参数，必须满足以下 4 个条件：

- (1) 该类必须提供一个公有的默认构造函数；
- (2) 该类必须提供一个公有的拷贝构造函数；

(3) 该类必须提供一个公有的析构函数。

(4) 该类必须使用宏 `Q_DECLARE_METATYPE` 进行注册；

满足上述 4 点要求后，该自定义类可以存储在 `QVariant` 对象内，并且可以在直接连接的信号槽中作为数据参数使用。注意，直接连接的信号槽意味着该类无法在跨线程中使用。

```
class Message
{
public:
    Message();
    Message(const Message &other);
    ~Message();

    Message(const QString &body, const QStringList &headers);

    QString body() const;
    QStringList headers() const;

private:
    QString m_body;
    QStringList m_headers;
};

Q_DECLARE_METATYPE(Message)
```

(5) 该类必须在程序中使用 `qRegisterMetaType` 函数进行注册。

```
int main(int argc, char *argv[])
{
    ...
    qRegisterMetaType< QMessage >("QMessage");    //!< 注册自定义类型
    ...
    return a.exec();
}
```

使用 `qRegisterMetaType` 函数对自定义类进行注册后，该自定义类即可在跨线程的信号槽中（作为参数）使用。

如果希望该自定义类能够使用 `QDebug()` 函数输出，还需要满足如下条件：

(6) 重载 `QDebug` 的 “<<” 运算符；

```
QDebug operator<< (QDebug d, const UClass &userClass) {
    d.nospace()<< userClass.member1<<"\t"<< userClass.member2
    << ... ;
    return d;
}
```

如果希望该自定义类能够使用 “<<” 和 “>>” 进行流式输入输出，需要满足如下条件：

(7) 重载 `QDataStream` 的运算符 “<<” 和 “>>” ；

```
QDataStream &operator<<(QDataStream &out, const MyClass &myObj);
QDataStream &operator>>(QDataStream &in, MyClass &myObj);
```

(8) 在程序中（一般是在 `main` 函数内）注册流操作运算符。

使用 `qRegisterMetaTypeStreamOperators` 函数进行注册，

```
qRegisterMetaTypeStreamOperators<MyClass>("MyClass");
```

自定义类型一览表

实现

1. `Type::Type()` – 公有缺省构造函数
2. `Type::Type(const Type &other)` – 公有拷贝构造函数
3. `Type::~~Type()` – 公有析构函数
4. `QDebug operator<<` – 方便的调试
5. `QDataStream operator<<` 和 `>>` – 流式输入输出

注册

1. `Q_DECLARE_METATYPE` – 在头文件中
2. `qRegisterMetaType` – 在主函数main中
3. `qRegisterMetaTypeStreamOperators` – 在主函数main中

图 8 自定义类型实现一览表

对 `dataWorker` 类，该类派生于 `QObject`，并且在类定义的第一行加入了私有的 `Q_OBJECT` 宏，因此该类可以使用信号和槽。在 `dataWorker` 类定义的最后部分，自定义了一个 `dataParseFinished` 的信号，该信号有 3 个数据参数。对于 `QList<QDateTime>` 类型的参数，需要在程序开始时，使用 `qRegisterMetaType<T>()` 函数将其注册到 Qt 中。

```
qRegisterMetaType< QList<QDateTime> >("QList<QDateTime>");
```

`dataWorker` 类也定义了一个用于响应 `QNetworkAccessManager` 完成网络请求的 `finished` 信号的 `httpsFinished` 槽函数，该函数为 `protected` 类型的槽函数。

```
protected slots:
    void httpsFinished(QNetworkReply *reply);
```

函数 `parseHTML` 和 `parseData` 分布负责解析含 HTML 标签的数据文本和纯数据文本。将这两部分功能分开，可以方便对已保存的本地数据进行解析（本地数据不含 HTML 标签，只需使用 `parseData` 函数即可）。

在 `parseData` 函数完成数据解析后，程序发射 `dataParseFinished` 信号，表示数据处理完成，通知图表进行更新操作。

```
dataList.removeFirst(); // 第一条数据是表头，删除
for (QString s : dataList){
    QStringList dataList = s.split(" ",QString::SkipEmptyParts);
    QDateTime momentInTime =
        QDateTime::fromString(dataList.at(0), "yyyy-MM-dd");
    dataDate.append(momentInTime);
    dataHigh.append(dataList.at(1).toDouble());
    dataLow.append(dataList.at(2).toDouble());
}
emit dataParseFinished(dataDate, dataHigh, dataLow);
```

六、预习及实验要求

1. 认真阅读本说明及参考示例；
2. 阅读 Qt 文档：Network Programming with Qt，并将其按照原文档格式译为中文（代码不需翻译）。
3. 模仿示例代码，添加能够查询不少于 5 个城市气温的功能，如"南京"、"北京"、"上海"、"杭州"、"哈尔滨"等。
4. 使用 QChart 绘制某城市指定年月的空气质量情况图表。空气质量数据来源网址：<http://www.tianqihoubao.com/aqi>，绘图至少绘制 2 个曲线，如 AQI、PM2.5 等。

查询 URL 格式如下：

[http://www.tianqihoubao.com/aqi/城市\(拼音\)-年月.html](http://www.tianqihoubao.com/aqi/城市(拼音)-年月.html)

城市：需要查询的城市名称，如“nanjing”，

年月：需要查询的时间，格式(yyyyMM，如 201710)

2017年8月南京空气质量指数AQI_PM2.5历史数据

南京08月份空气质量指数(AQI)数据: 数值单位: $\mu\text{g}/\text{m}^3$ (CO为 mg/m^3)									
日期	质量等级	AQI指数	当天AQI排名	PM2.5	PM10	So2	No2	Co	O3
2017-08-01	优	36	66	7	27	9	18	0.77	68
2017-08-02	优	33	43	11	27	9	29	0.75	57
2017-08-03	优	40	107	14	35	13	34	0.99	51
2017-08-04	优	48	129	22	47	18	37	1.36	62
2017-08-05	良	56	186	24	54	14	49	1.18	66
2017-08-06	良	72	258	35	69	12	43	1.12	92
2017-08-07	良	73	281	35	75	14	49	1.22	85
2017-08-08	优	30	31	8	16	9	27	0.82	54
2017-08-09	优	35	79	12	26	9	24	0.83	69
2017-08-10	良	52	174	18	41	12	28	1.07	82

图 9 部分网页数据

问题 4 提示：

(1) 大多数网站，如果链接 URL 中含有中文，那么需要对中文进行转码（称为 urlencode）。可以使用在线的转码工具进行查询（搜索关键字：在线 urlencode）。例如“南京”的 urlencode 为“%E5%8D%97%E4%BA%AC”（不区分大小写）。Qt 中可以分别使用静态函数 QUrl::toPercentEncoding 和 QUrl::fromPercentEncoding 进行中文与 url 编码的相互转换。注意，urlencode 对英文和数字无效果（保持不变）。

(2) 使用 QXmlStreamReader 解析 HTML 时，要求解析对象须严格按照 XML 规范进行。例如类似“<tr height=38px>”的内容，不符合 XML 规范，需修改为“<tr height=’38px’>”。

(3) 解析时 tag 设为“tr”，表格的基本 HTML 语法如下：

```

<table>
  <tr>
    <td> <b>日期</b></td>
    <td><b>质量等级</b></td>
    <td><b>AQI 指数</b></td>
    <td><b>当天 AQI 排名</b></td>
    <td><b>PM2.5</b></td>
    <td><b>PM10</b></td>
    <td><b>So2</b></td>
    <td><b>No2</b></td>
    <td><b>Co</b></td>
    <td><b>O3</b></td>
  </tr>
</table>

```

其中，table 表示是一个表格，tr 表示表格的一行，td 表示一列。td 如果嵌套在 tr 中表示是该行的某一列。详情请参阅相关资料。

- (4) 注意修改纵坐标的范围，符合数据变化幅度。
- (5) 由于城市名称为拼音，注意 AQI 数据文件名与气温数据文件名需有所区别。
- (6) 数据保存，请在当前目录下创建一个 data 目录供数据保存使用，空气质量以 aqi 开头，如 aqi_nanjing-201809.txt，天气以 weather 开头，如 weather_nanjing-201809.txt，
- (7) 时间，请从用户使用时的月份前一月开始，向前 10 个月，如当前日期为 2018 年 10 月 30 日，则时间跨度为 2018-09，2018-08，...
- (8) 参考实现界面如图 10 所示

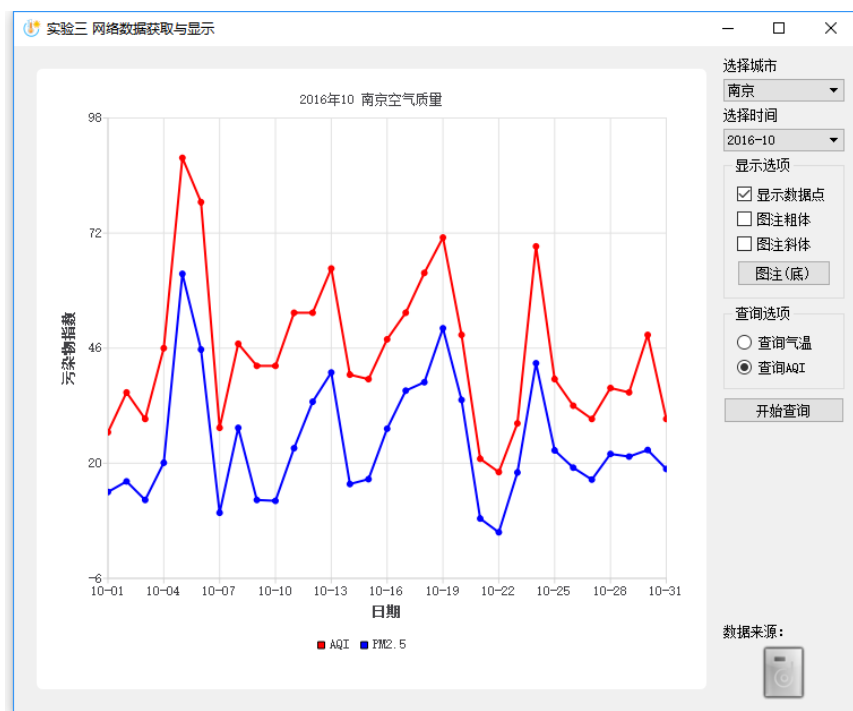


图 10 参考设计界面

七、PPT撰写要求

认真理解代码后，请围绕以下几个方面撰写汇报 PPT。

- (1) 如何使用布局编辑器进行布局（layout）（基本流程）？
- (2) Qt 中使用网络功能的基本过程是怎样的？
- (3) HTTP 请求中的 Get 和 Post 方法有何区别？如何根据 HTTP 请求的返回值判定网络状态？如何理解 HTML 中的 Tag？
- (4) 使用 QtCharts 绘制图表的基本流程是怎样的？
- (5) 如何自定义信号和槽？如何发射信号并在信号中使用自定义数据类型？结合代码说明。
- (6) 什么是运算符重载？如何撰写一个运算符重载函数，使得 qDebug 能够将所有 `QList<QDate>` 列表以（2017-11-01,2017-11-02,...）的形式输出？
- (7) 使用流式文本读写访问的基本流程是什么？如何配合使用 `QStringList` 进行文本解析？注意 `simplified` 函数的功能。