# 第一章 引言

## 1.1 系统背景

随着互联网技术的飞适发展，现在的软件开发、WEB开发、游戏开发……都离不开互联网，但是在开发与调试与网络相关的功能时，需要自已写一些测试代码，如果出了问题之后还需要确定问题究竟是出在服务端程序还是客户端程序上，这些都需要耗费一定的时间和精力，大大降低开发的效率；网络功能测试、故障排察，针对这两个方面，需要一个高效的工具来帮助开发人员来完成这两项繁琐而又重复的工作；而现在网上有关的工具都太过于专业且单一，对于开发人员来说不太友好，所以将在本次毕业设计中做这样一个网络数据构造的工具，它能够帮助开发人员完成一些简单而又繁琐的工作；

## 1.2 意义

帮助开发人员完成一些简单而又繁琐的工作；

# 第二章 软件需求分析

## 2.1 软件应用场景

涉及到网络相关的开发：比如web开发、游戏开发、APP开发、普通的软件……一系列需要发送或接收TCP数据的应用；

## 2.2 需求分析

既然是开发中会遇到的工作，就先要分析我们在实际的开发中到底会进行哪些操作，首先是web开发，涉及到的基本上是HTTP协议，发送数据有两种方式：POST和GET，请求的内容是URL+参数，响应的数据比较多样化，有html、xml、json、以及各种格式的图片、视频、音乐文件，所以对于web开发者而言，做的事情相对较少，只需要完成POST和GET两种传输数据的方法，以及把收到的数据用适当的形式显示出来即可；

其次是对于普通软件开发和游戏开发而言，现在大多数的开发而言使用的都是面向连接的可靠协议TCP，特别是游戏应用中，对网络的连接要求最高，玩家的数据需要即时而又准确地传输到服务器，所以在这些应用中，为了保证数据传输的可靠性，都是采用的TCP协议，而TCP协议可以传输的数据类型更为多，比如TCP可以传输原始的二进制数据，这一点是HTTP不具备的，但是原始的二进制数据游戏需要考虑的方面实在是太多（图片，基本数据类型、纯文本、音频数据……），在数据太大需要分片时，更加难以处理，所以这里无法对所有的数据类型进行处理；好消息是在使用TCP时，几本上会传输太过复杂的数据，因为复杂的数据都有更上层来处理，比如文件传输使用FTP，网页使用HTTP、邮件传输使用SMTP……作了这些分析后发现在一般的软件和游戏中传输的数据类型都是纯二进制或json、xml数据，而json和xml都属于纯文本的范畴，二进制都是些自定义的数据类型，也就是c/c++、java中的struct/class，这些都是由char、int、long、float、double等基本数据类型构成；

但是对于在程序运行的时候，需要查看一下程序到底发送/接收到了什么数据，这个时候就需要数据抓取的功能，功能也就是网上所说的抓包软件wireshark之类的，但是wireshark对于抓取的数据包是把数据链路层、ip层、tcp层的协议数据也显示出来了，而这些对于应用开发人员来说是完全没有必要 的，所以在我所设计的这个软件中，不再显示这些底层协议，而是直接把我们发送的具体内容显示出来，而这些接收到的数据也是原始的TCP数据，处理方式和发送TCP数据时的处理方式一样，这样可以将它的数据解析部分与发送TCP数据部分的数据解析功能独立出来，设为公共部分。

## 2.3 功能总结

综合上面需求分析里面提到的内容，这个软件的功能具体如下：

1. 发送HTTP请求：请求方式可以选择POST或GET；
2. 发送TCP数据：发送的数据可以是纯文本、二进制数据流；
3. 抓取与指定主机+端口的通信数据，并且可以查看单条数据的详细信息；
4. 整个软件都带图形界面，方便用户操作；
5. 对于TCP数据，支持加密解密，压缩解压功能，应对多变的环境；

## 2.4 开发环境的选择；

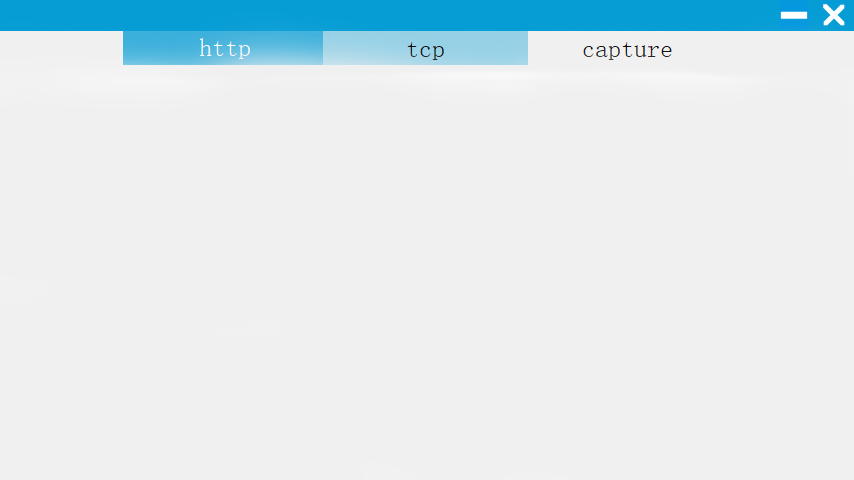
现在主流的操作系统是windows，linux和mac用户只是极少数，所以不作考虑，软件的图形界面使用强大的开源图形框架Qt，与网络交互的部分也使用Qt中封装的网络部分；数据抓取使用开源软件wireshark 的底层库winpcap，数据压缩暂时使用现阶段应用较为广泛的quicklz算法，数据加密解密使用对称加密算法RC4；

# 第三章 软件框架设计

## 3.1 子系统划分

### 3.1.1 主界面

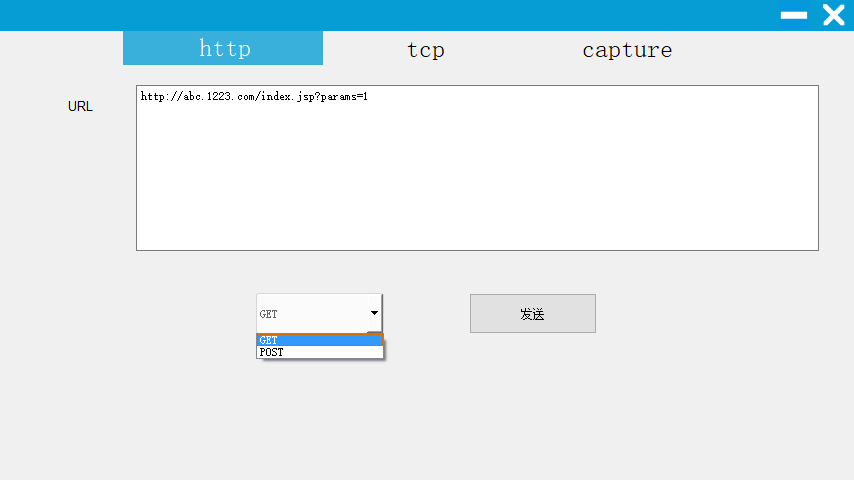
主界面直接将所有的功能放在主界面下，通过Qt提供的TableWidget来回切换



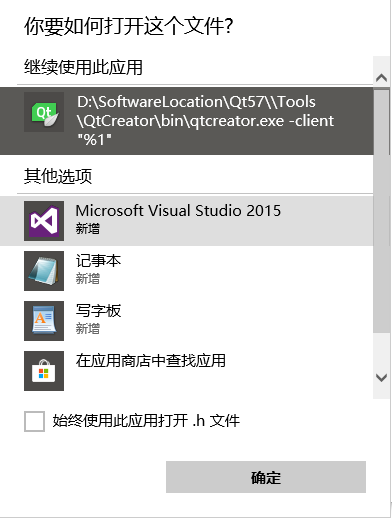
软件中的三个主要功能：发送HTTP数据、发送TCP数据、数据抓取，通过这个TabelWidget来回切换，可以三个功能同时使用，且相互之间不会干扰,主界面的窗口使用QT重写，这样相比于不重写的界面在相同大小的平面上，可以放更多的内容，对于屏幕较小的用户来说无疑是很大的福音

### 3.1.2 HTTP数据包发送模块

如图，为HTTP数据发送模块的界面，



用户可以在名为URL的文本输入框内输入要访问的链接，格式为标准的GET请求格式，如果需要以POST的方式请求，则直接在下方的下拉列表框中，选择POST请求方式即可，格式相同，在输入URL和选择请求方式后，就可以点击发送按钮发送HTTP请求；收到请求后，会根据根据HTTP响应头中的Content-Type来确定是哪种类型的文件，然后根据文件的类型，将收到的数据保存到本地，文件名的后缀与Content-Type类型相对，然后通过系统的文件资源管理器来选择对应的默认应用程序来打开此文件，若没有设置打开此类型文件的默认应用程序，系统则会弹出对应的提示框，提示用户选择打开此类型文件的应用



选择应用后就可以打开该应用了，不过一般的都会使用浏览器打开应用；

当然，请求的内容只能是文本的格式，这样的功能看起来虽然是单一了点，却是完全够用的，想想我们平时浏览网页的时候，最多也只是点几下鼠标、输入英文字符以及汉字，要说发送其它的数据，可能也就是在特殊的应用场景了，已经脱离绝大多数软件开发者的实际了，而且做起来难度也要大很多，由于时间有限，所以这里不做实现；

### 3.1.3 TCP数据发送模块

TCP数据发送这个模块，相比HTTP数据发送的模块要复杂得多，因为使用TCP数据传输的时候，多数时间是出于性能、安全性等特殊的原因考虑的，而且还有一个因素是绝大多数使用TCP协议来传输数据的软件，都对数据进行了压缩、加密的操作，且数据完全是二进制的格式，需要将它转化为char、int、double、string等数据类型、所以要完全实现与开发者的应用相接轨，是有相当的难度的，基于以上的原因，需要将这个模块拆分为以下几个模块：（1）数据包的构造；（2）数据包的加密与解密、压缩与解压缩；（3）对于收到的网络数据包的解析；（4）网络数据包的发送与接收（5）几个界面的独立设计；

要将这些功能全部拆分开来看的话，整个软件系统设计得过分完美，同时也是设计过度的，所以需要对这几个功能进行一下合并，将其它设计为适当当的大小；但对于各个模块又完全地解耦合，对于数据包的构造和网络数据的传输，可以放在一个模块，也就是主界面中去，加密解密和压缩解压又作为一个独立的模块，数据的解析也作为一个独立的模块，与后面的数据监听模块共用；

d. 数据监听模块：

数据监听这个模块主要是监听指定网卡，指定IP端口上的数据，也可以对网上上的数据进行简单的筛选，当然也只是对数据包的长度，目的IP和目的端口来进行筛选，对于收到的网络中的数据，显示摘要数据，详细的数据用文件保存起来，需要的时候再对文件进行读取，因为如果将网络中的数据全部通过内存的形式缓存起来，可能会因为数据太多，而出现内存不够用的情况，这种情况对于内存较小的老机器尤其严重；另外，对于数据监听这一块，由于它需要一直循环，且不会退出监听函数，如果要把它与界面放在一个线程里面的话，界面中的函数不会执行，应付出现假死的情况，所以需要把数据监听的功能单独放到一个线程里面去，边监听边保存，并将收到的数据的简要信息发送到主界面的线程中，显示出来，如果需要查看数据包的详细信息，再读取本地文件中的数据，加载到主界面中，但文件读取也是一个耗时的操作，当文件太大时，读取文件的时间也相对变长了，也会使主界面出现假死的情况，所以为了避免此类情况的出现，需要单独开辟一个线程来完成文件读取的操作，这样主界面也就只是进行的数据的显示和控制的作用，真正的数据在另外两个子线程中，这也是现阶段MVC出现最多的形态，控制层和显示层混合在一起；当然数据抓取只针对会用到的TCP协议，与TCP数据发送模块相同，也需要数据加密解密，压缩解压的功能；



如图所示，为数据抓取模块的设计，从上方的网络接口的QComboBox下拉菜单中选取一个本机的网络接口，再输入要抓取数据的IP地址、端口、最小数据长度，点击开始即可进行网络中的数据抓取，如果需要加密解密、压缩解压的功能，那么点击高级选项的按钮，就可以弹出一个子对话框，在子对话框中，可以选取压缩数据的算法，加密解密的算法，从而对将要发送的数据进行加密，对接收到的数据进行解密，操作相当简单直观；

对于接收到的网络数据要相查看详细信息，双击单条记录，就可以显示信息的详细信息，此部分与TCP数据发送部分的数据接收模块公用，将在下一部分介绍；

### 3.1.4 数据解析模块

上面介绍了TCP数据发送模块和网络数据监听的模块，一直在提数据解析这个模块，但是一直没有给出实现，下面将在这一部分作详细的分析，数据解析模块的功能看起来比较直观，它的功能就只有一个，那就是显示数据：但是我们不仅仅需要显示ascii字符，还要让它可以显示编程中的基本数据类型，比如short、int、long long int、float、double基本数据类型，这些数据类型都是有着统一的标准的，与语言的实现无关，所以就算是用其它编程语言写的程序中使用到这些数据，都是可以正常显示的；有时候我们只需要看数据指定位置上的数据的值，这时可以能过程序中读取string的长度来设置数据偏移，然后再读取指定位置上的数据；

### 3.1.5 数据压缩解压、加密解密模块

数据的压缩和加密，这两个功能涵盖的面实在是太广了，由于时间的和个人能力的限制，这里不可能将所有的加密算法和压缩算法全部加进去，所以只选取两个应用比较广泛且具有代表性的算法来实现；

网络数据传输中的压缩与解压，主要的作用是减小发送数据的大小，减小数据中的冗余、空白部分，使数据在传输过程当中更加快速可靠，但是由于处理的时间很短，所以对算法的效率要求非常之高，但对于能压缩多少的空间这方面并不是特别重要，要选取网络数据压缩算法时，最重要的因素就是要求压缩的时间短，至于压缩比例，只要不是特殊的应用场景，要求都不高；quicklz号称是最快的数据压缩算法，在数据压缩率上相对其它的算法也毫不逊色，所以这次就只选取了quicklz来做为测试算法，完成数据压缩和解压这一功能；

软件中的数据采用透明传输？数据被窃取了怎么办？为了保证用户的合法信息，现在的软件在网络传输中，对于关键的数据都是进行了加密处理的，也就是说发送的数据是已经加密过的数据，服务端接收到数据后，会按照加密的钥匙进行解密，如果数据没有经过加密，解密出来的数据到底是什么估计没有人会知道，还很有可以导致服务端崩溃，所有说数据的发送在特殊的场景是需要进行加密处理的；服务端向我们发来加密的数据之后，我们不能解密，看到的只是一串不明不白的数据，也就是所说的无效信息，这当然不是我们所看到的，所以对于接收到的数据还需要进行解密操作；

上面探讨了数据加密解密这个功能的必要性，那么接下来就需要分析到底采用什么样的数据加密算法来做（同数据压缩算法一样，数据加密算法各类众多，且还有一些新的算法在出现，所以不可能通通实现，这里也只是选取一种具有代表性的加密算法来使用）；现阶段的数据加密算法主要分为两大类，对称加密算法和非对称加密算法，现阶段主流的对称加密算法有RC4、DES、AES……，非对称加密算法应用广泛的也就是大名鼎鼎的RSA算法了，而在一般的应用上不会用到RSA这种非对称加密算法，所以这次实现的加密算法采用对称加密算法RC4，加密算法本身比较简单，且难以破解，应用广泛，所以软件中首先加入RC4加密算法；

## 3.2 功能模块设计

### 3.2.1 主流程

用户打开程序后，可以在主界面选择使用HTTP数据发送、TCP数据发送、网络数据抓取这三个功能，这三个功能可以同使用，但是在界面上只能显示一个；

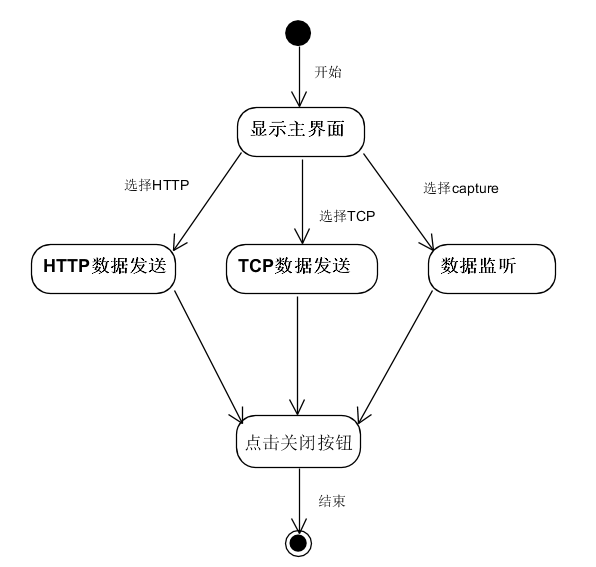
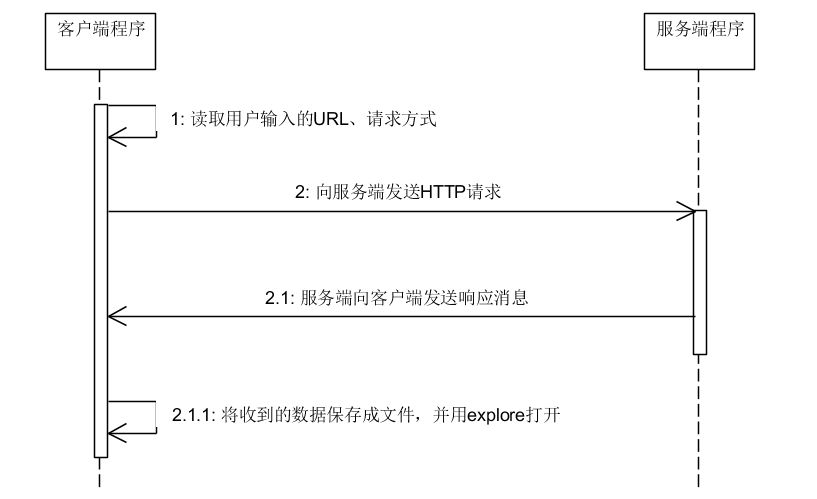


图3.1主流程图

### 3.2.2 HTTP数据与接收

用户输入URL、选择请求方式，并点击发送按钮，开始发送HTTP数据，接收到HTTP数据后，再显示数据内容；



3.2 HTTP数据发送时序图

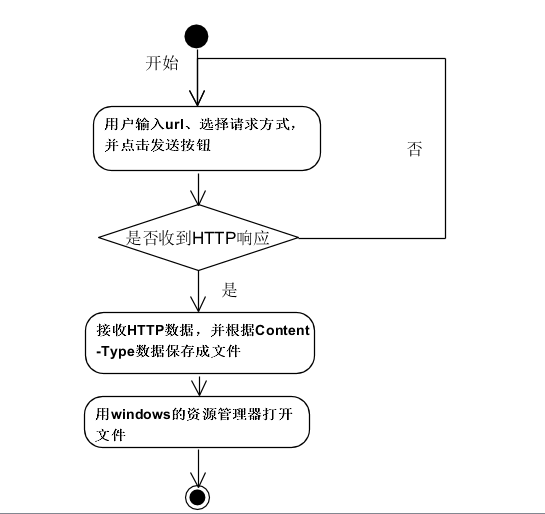


图3.3 HTTP数据发送活动图

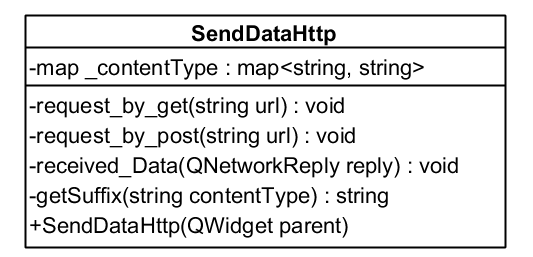


图3.4 HTTP数据发送类图

### 3.2.3 TCP数据发送

用户输入目标IP地址、目标端口，再通过程序中的数据构造功能，创建自己想要发送的数据，（如果需要，可以加密压缩数据）之后点击发送按钮即开始发送数据，如果自己与目标主机网络是畅通且目标主机开启了监听，那么数据的连接将会开启，并向目标主机发送数据；

接收到数据后，将数据传给TCP数据解析模块，解析数据（如果需要，对接收到的数据进行解密、解压）；

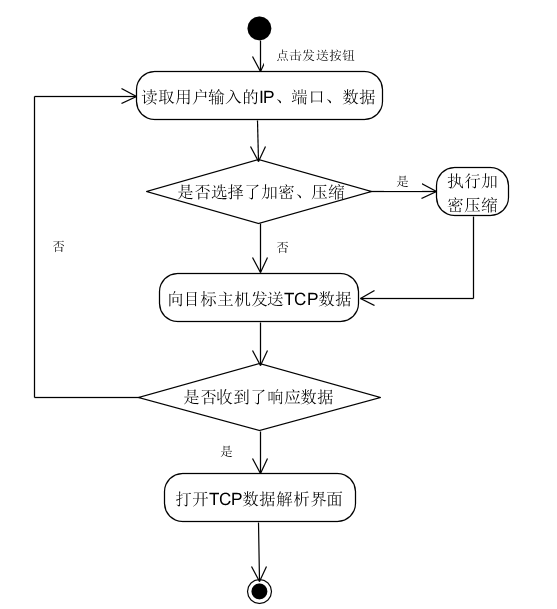


图3.5 TCP数据发送活动图

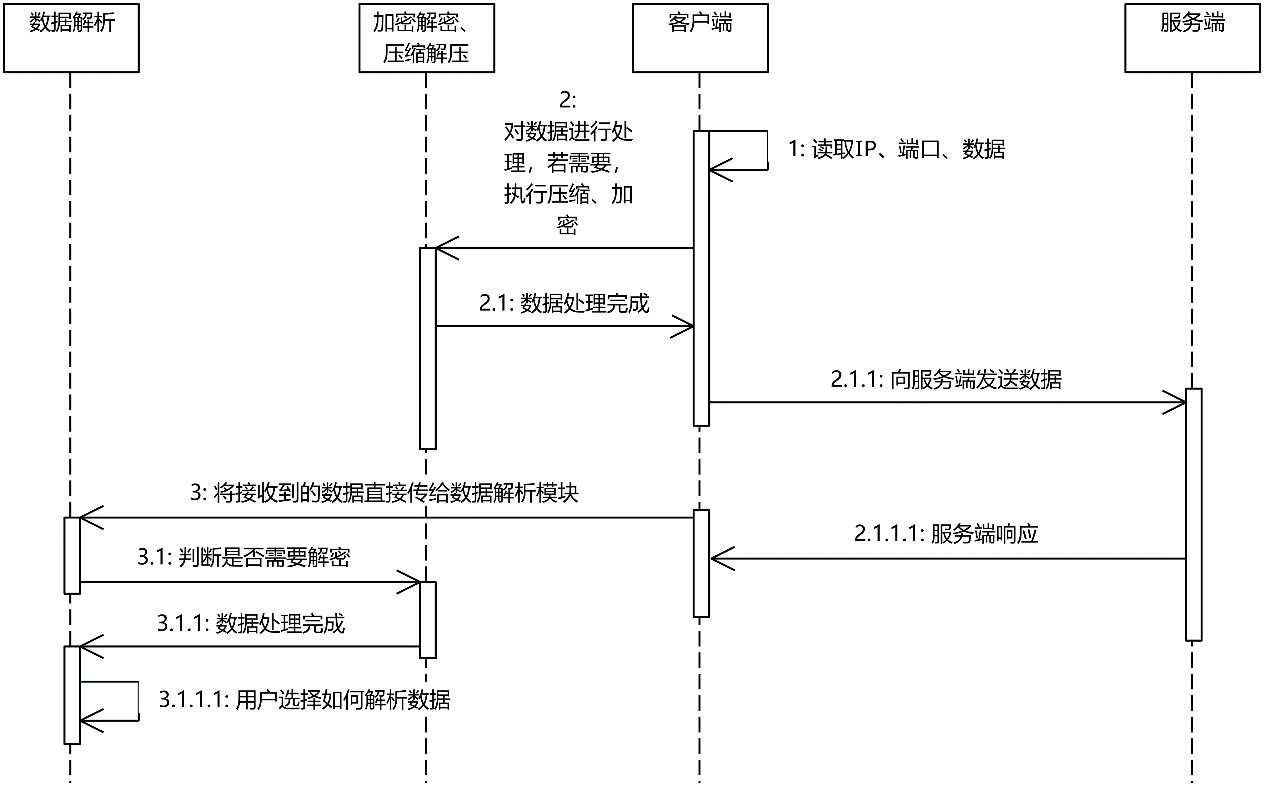


图3.6 TCP数据发送流程图

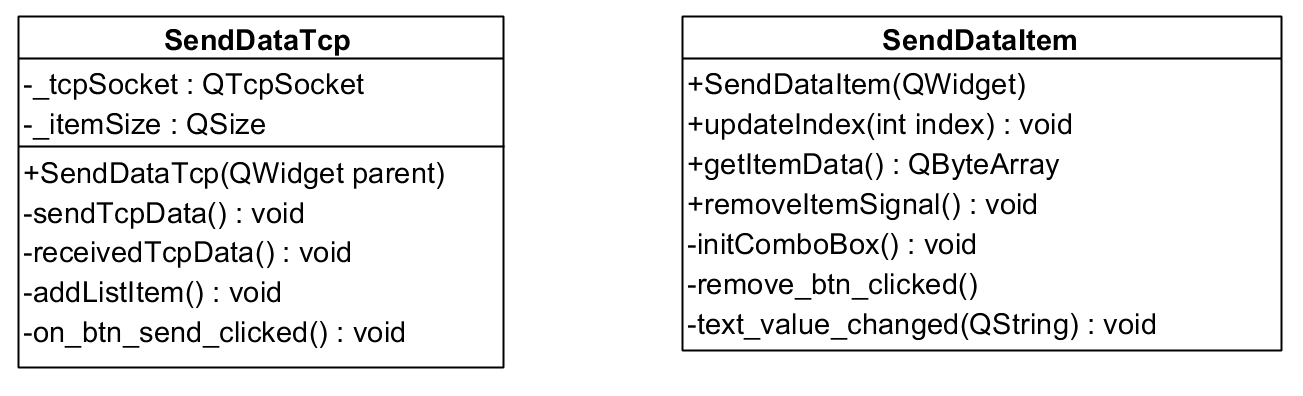


图3.7 TCP数据发送类图

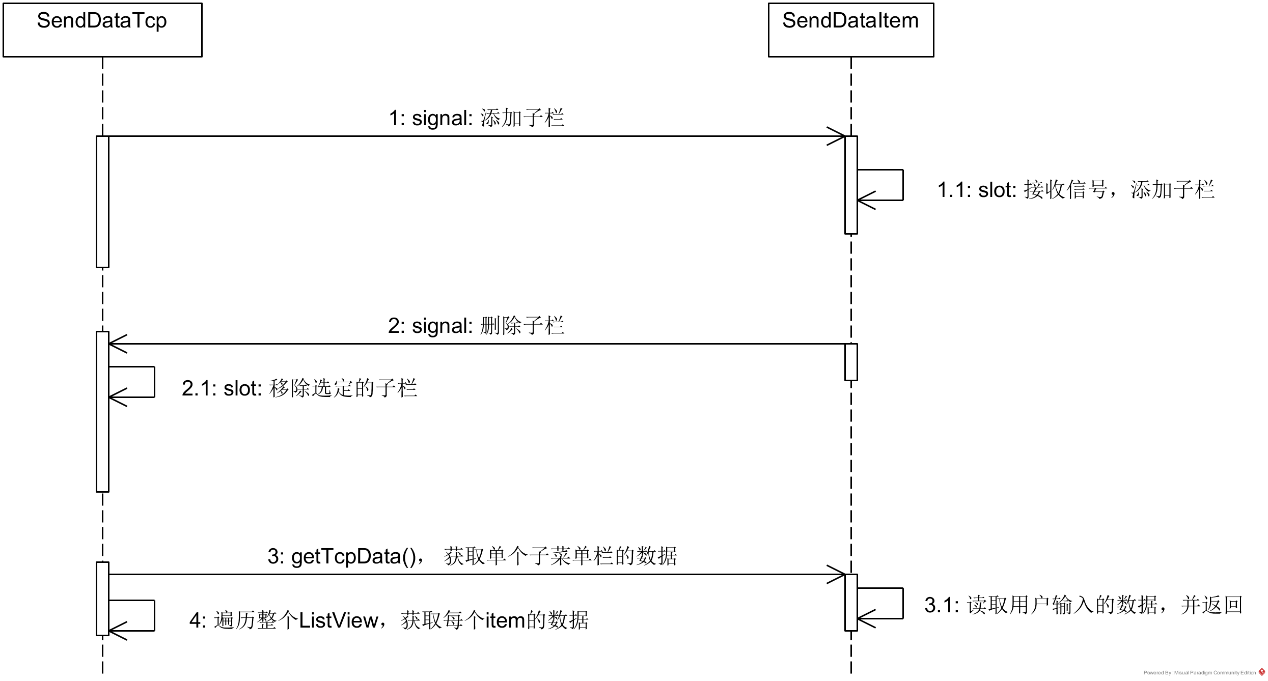


图3.8 TCP数据构建流程图

### 3.2.4 TCP数据解析

在用户发送一条TCP消息后，收到回应会自动弹出TCP数据解析的界面，或者在后面的数据抓取功能中，双击单条数据，查看详细信息，也会弹出TCP数据解析的界面在解析TCP数据之前，如果发现用户已经选择或配置了压缩加密的算法，则先执行解密和解压的操作，且顺序不能乱，因为在加密和压缩一块使用的时候，是先进行压缩，再进行加密的，如果顺序弄反了，将不会得到正确的结果；解析之后的数据，默认会以字符串的形式出现’\0’字符默认以ASCII字符’.’来表示，如果用户需要更加查看数据中单个或多个字节，那么可以使用TCP数据查看选项来查看；

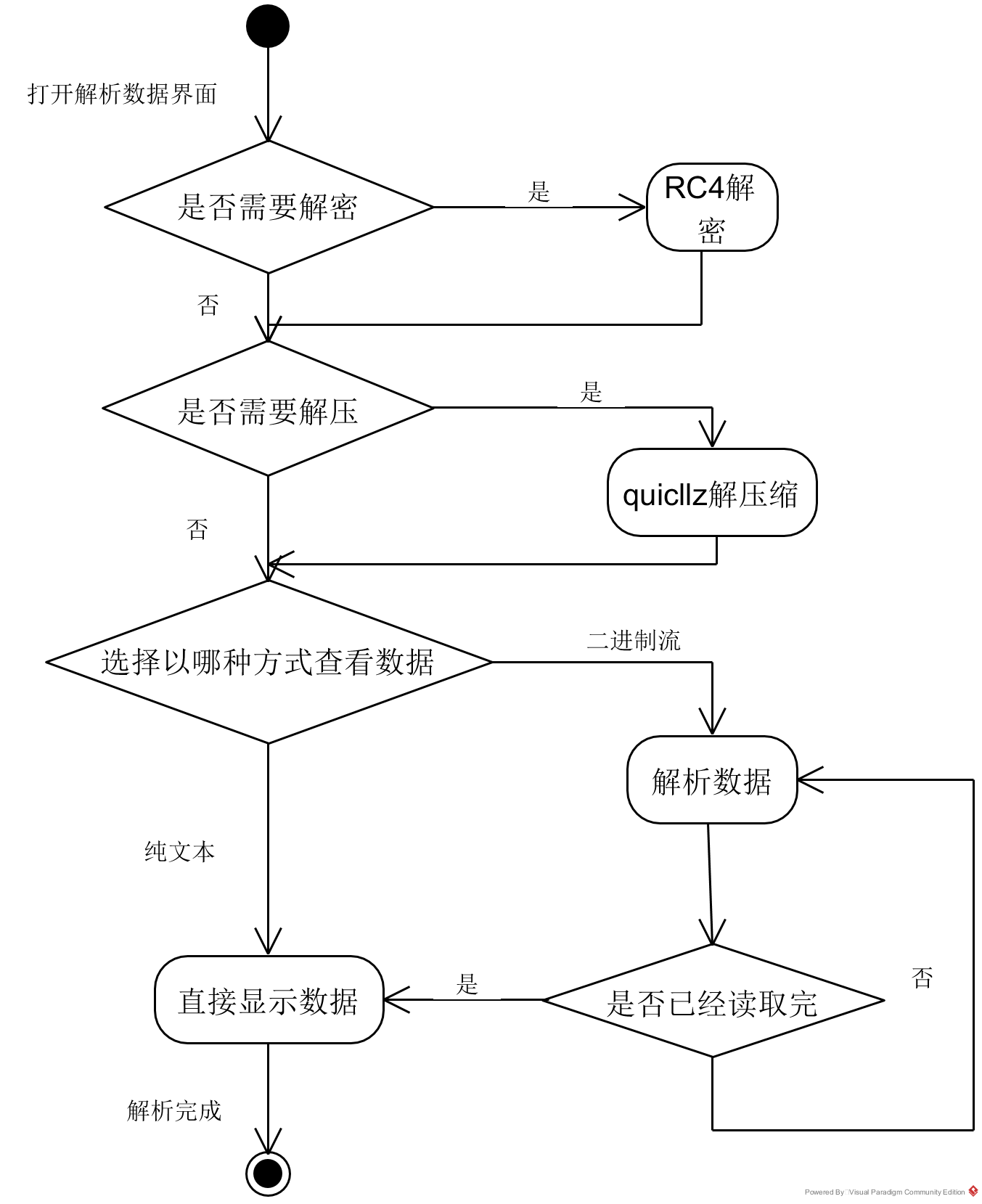


图3.9 数据解析活动图

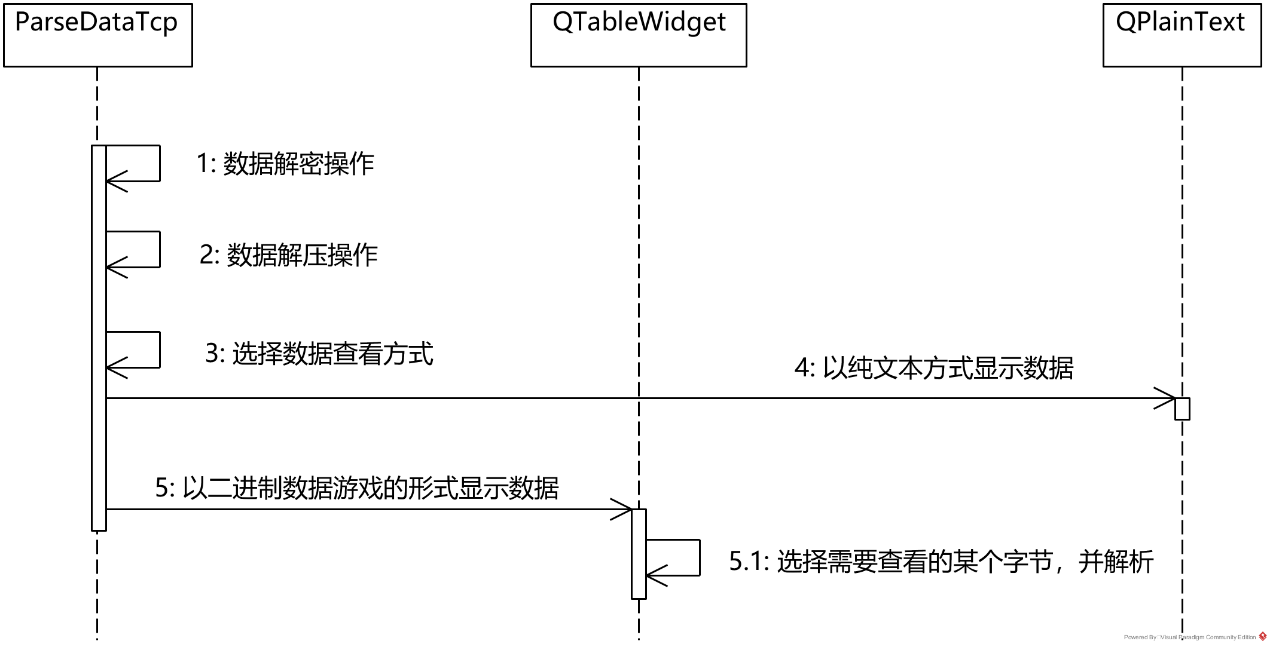


图3.10 数据解析流程图

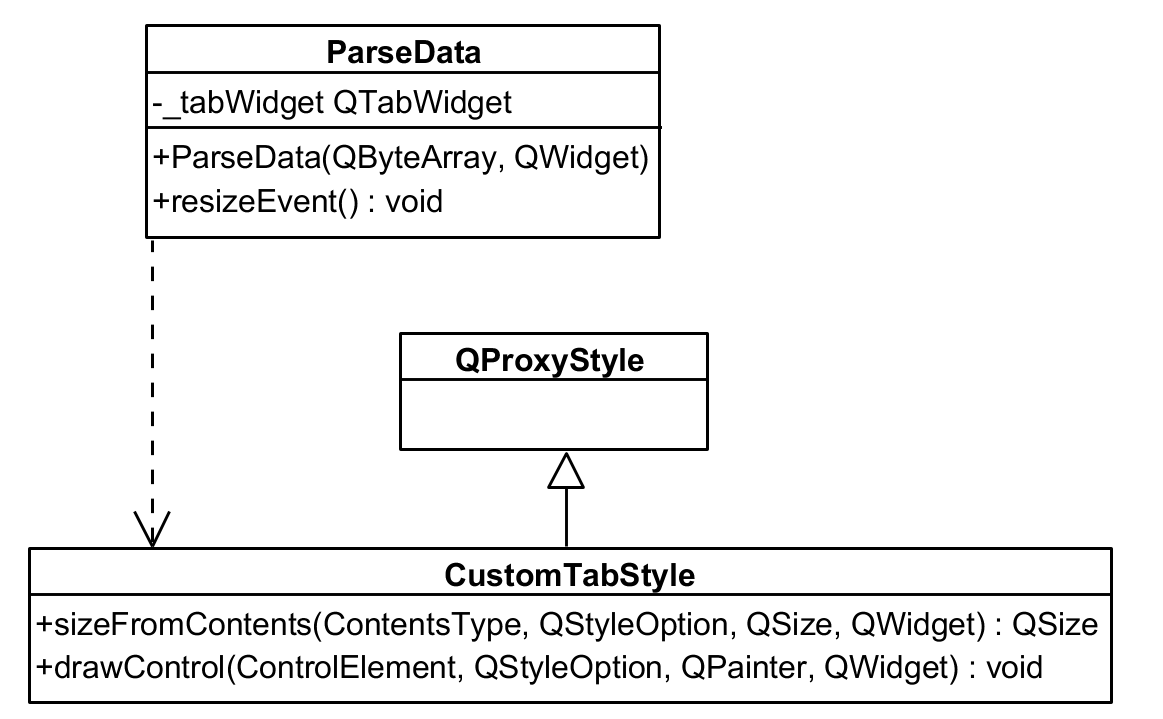


图3.11 数据解析主界面类图

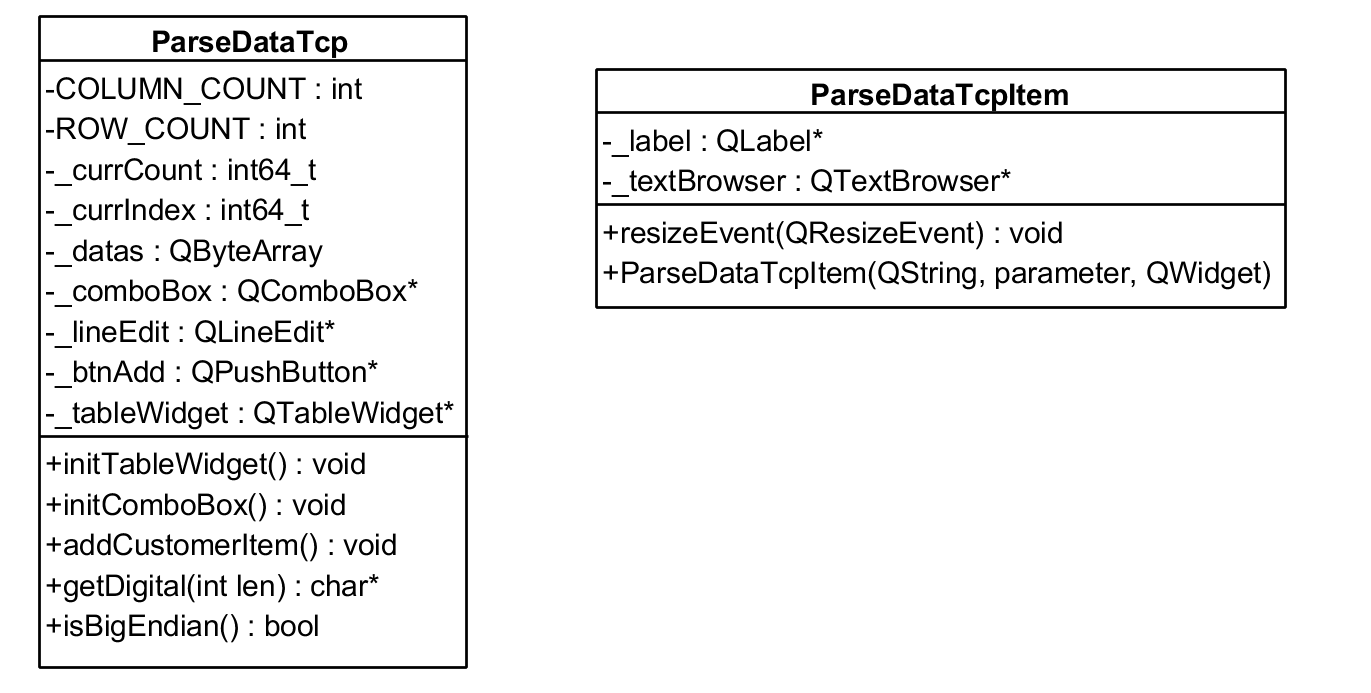


图3.12 TCP数据以二进制数据解析类图

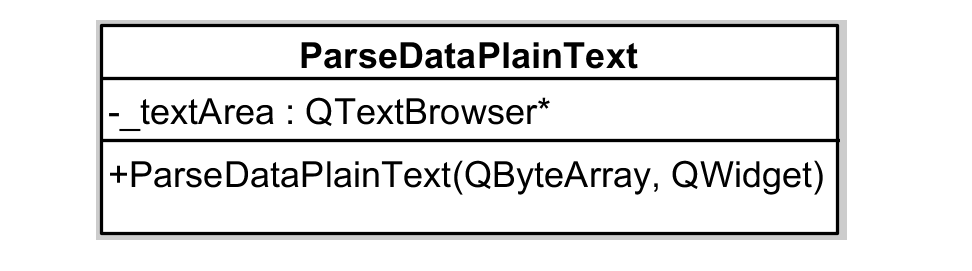


图3.13 以纯文本数据解析类型

### 3.2.5 加密与压缩

要进行加密解密、压缩解压的操作，首先要确定是否需要进行此类型的操作，如果在不需要进行加密或压缩的情况下强行进行压缩操作，那么原本正常的数据就会变得不可预测，所以加密压缩这些操作要在需要的时候才会执行；

这里的选择单例模式，可以在整个程序中共享一个全局的唯一对象，在这个对象中保存加密解密、压缩解压信息，在发送和解析数据的时候，通过这个全局唯一对象中存储的数据判断是否需要进行相应的操作，默认情况下是没有压缩与加密的操作，需要用户通过给出的图形接口来进行配置相应的加密与压缩算法；

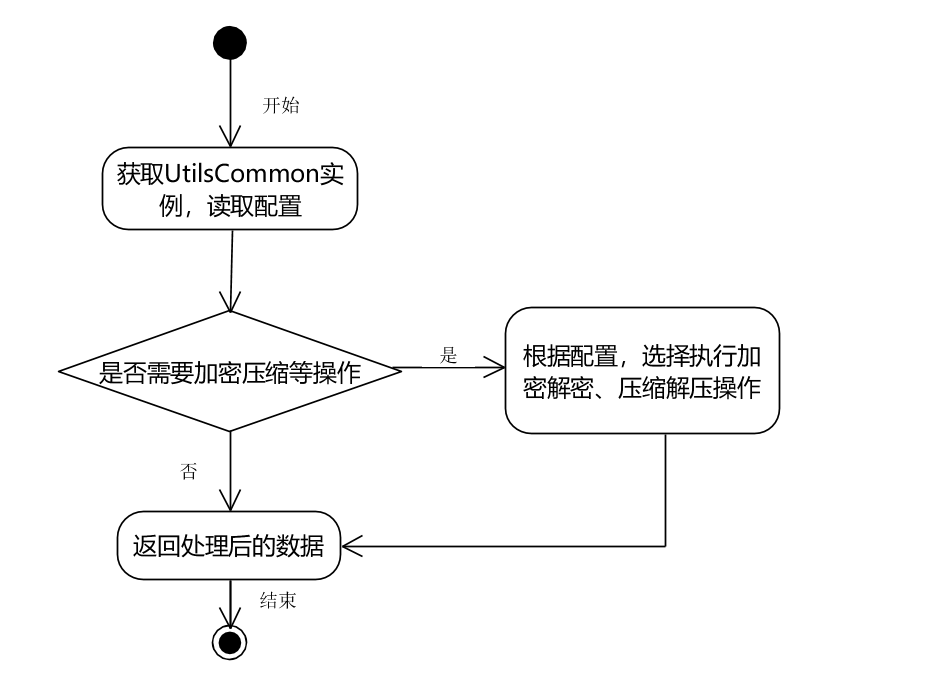


图3.14 加密压缩主流程活动图

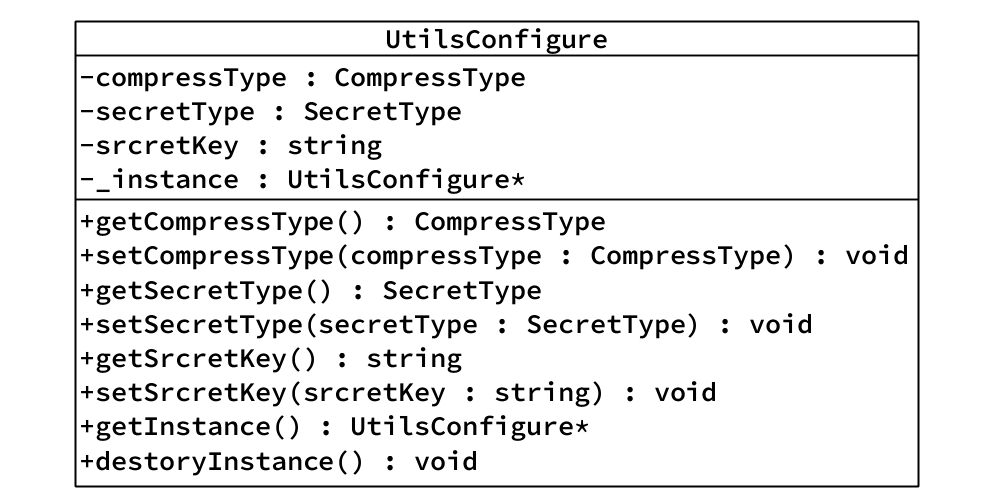


图3.15 UtilsConfigure单例类图

Quicklz算法直接调用函数库，不需要进行多余的设计

RC4没有专程的函数库，所以需要自己实现并封装好，方便使用；因为RC4加密与解密的操作完全一致，所以类型的设计中虽然有encrpyt和decrpyt两个方法，但是这两方法都是调用同一个方法来实现其它对应的功能的；

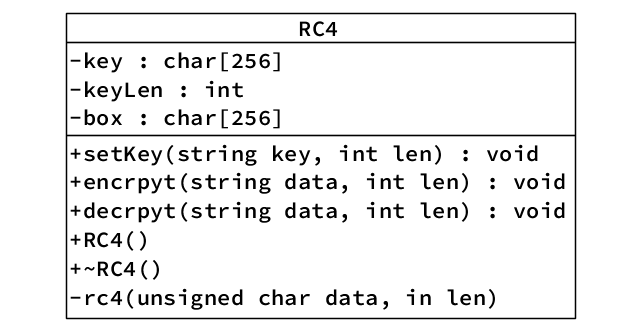


图3.16 RC4加密类的类图

### 3.2.6 数据抓取功能

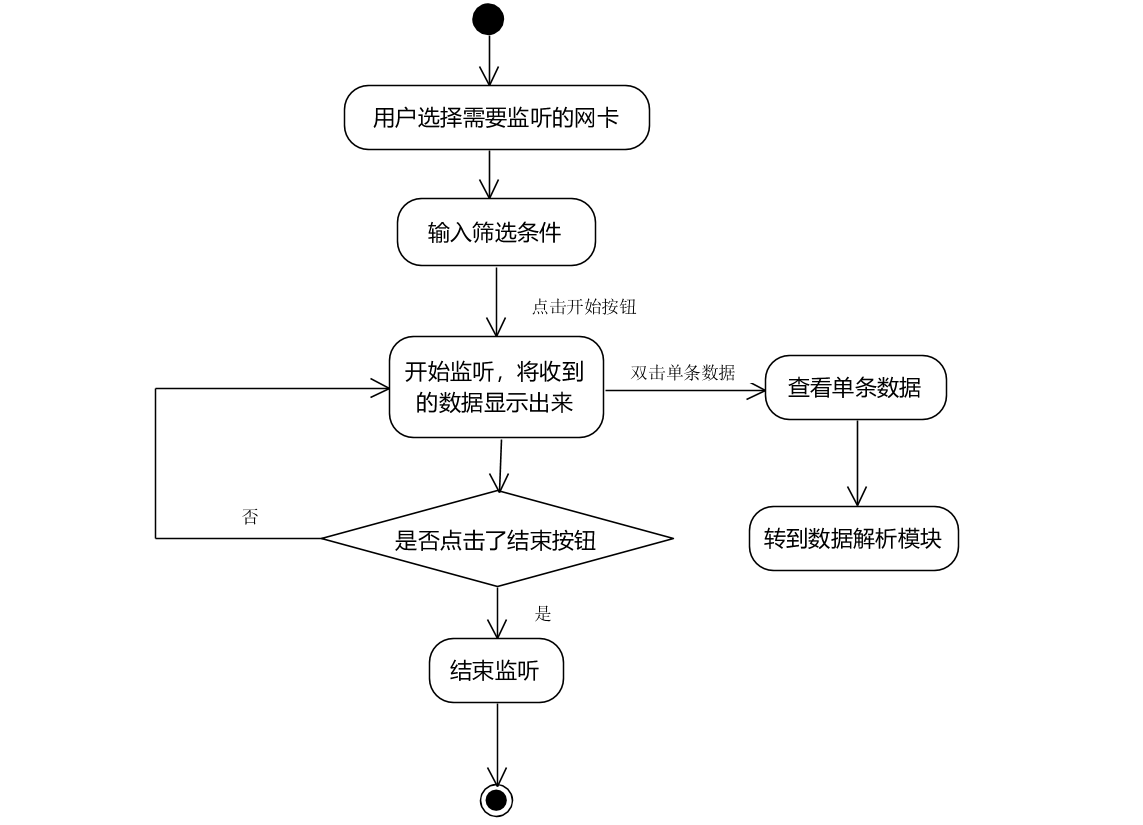


图3.17 数据监听主流程图

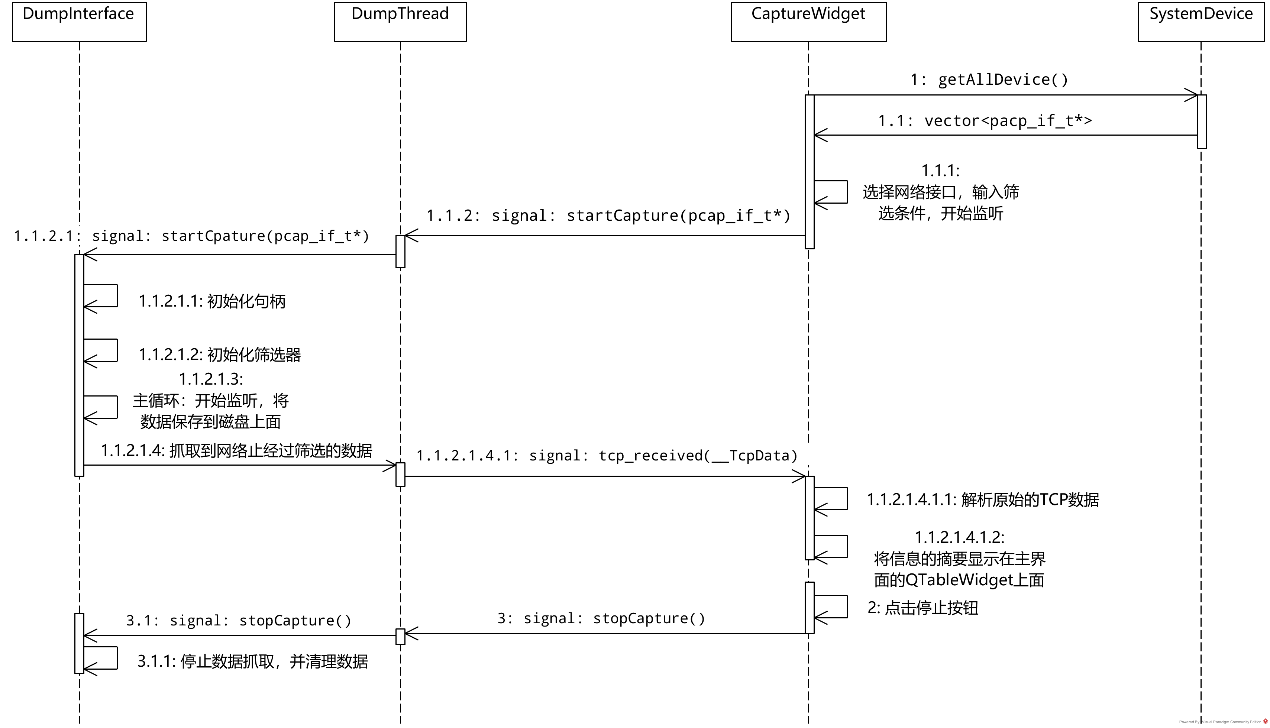


图3.18 网络数据监听时序图

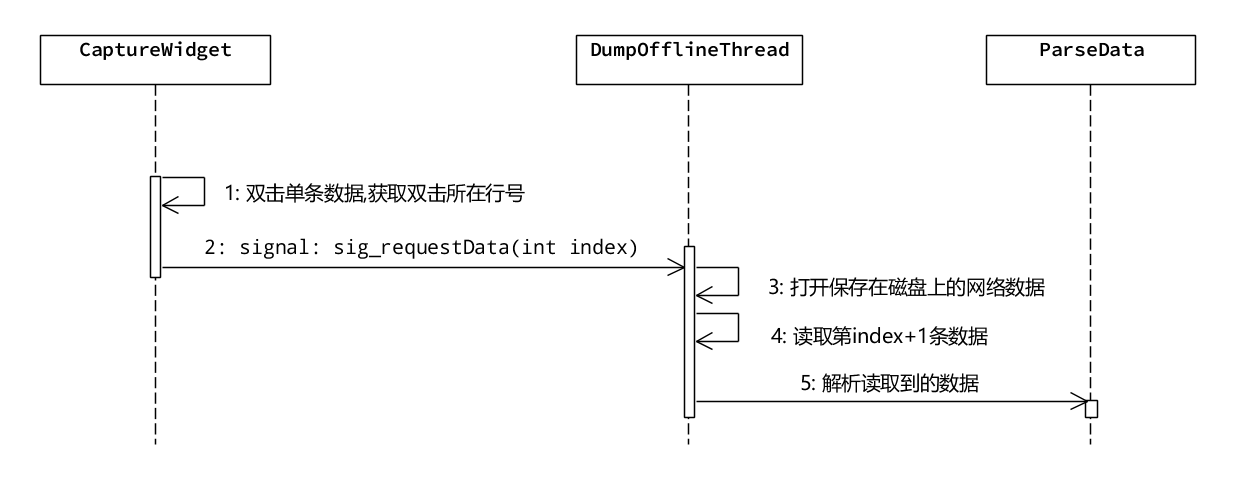


图3.19 查看单条数据时序图

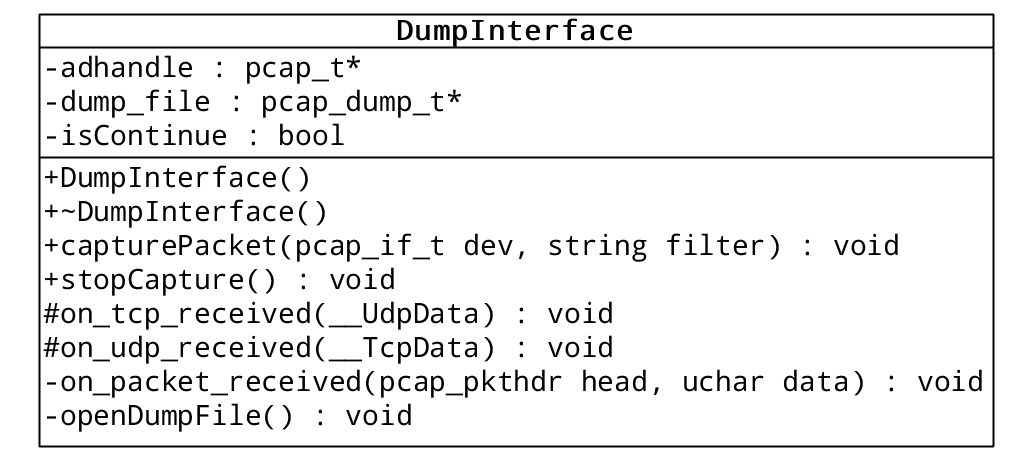


图3.20 数据监听抽象接口DumpInterface

数据监听线程类，继承自DumpInterface抽象类和QObject类

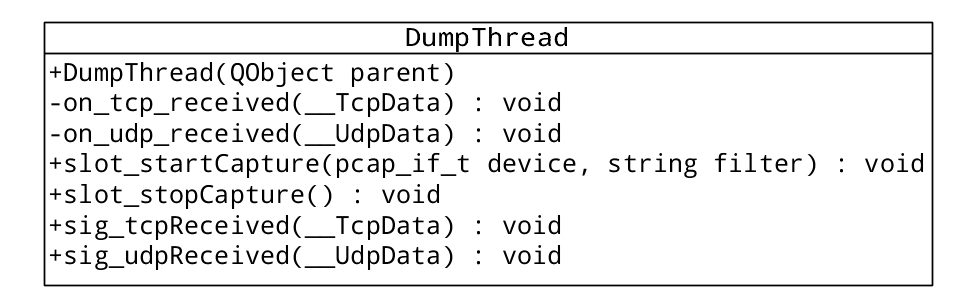


图3.21 数据监听线程类DumpThread

新开一个线程类，读取保存在磁盘上的网络数据，继承自QObject类

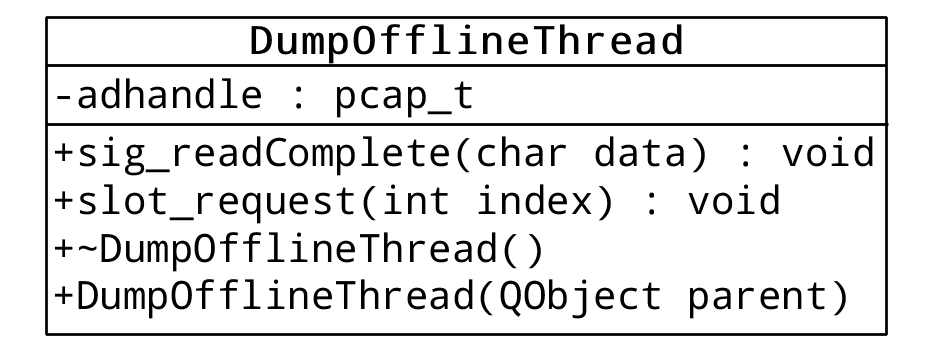


图3.22 读取保存在磁盘上的网络数据类图

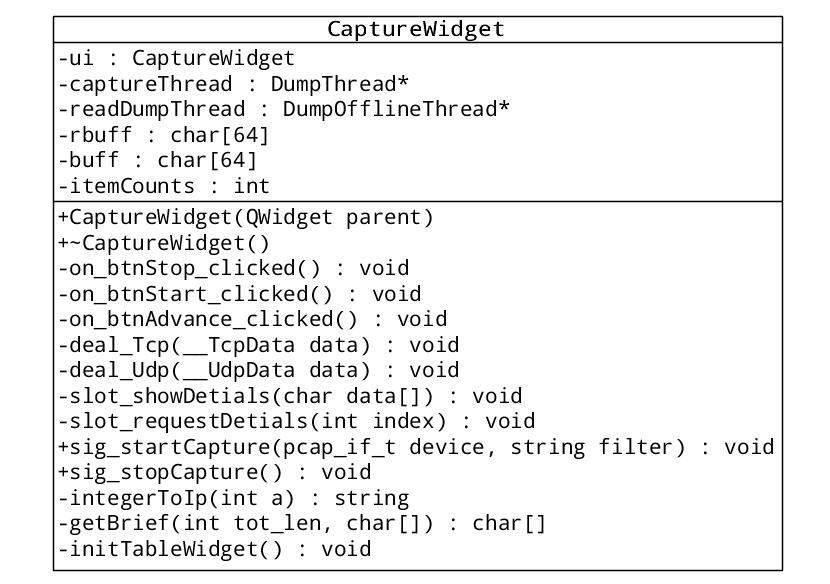


图3.23 网络数据监听主界面类图

# 第四章 软件详细设计

## 4.1 QT中基础知识简介

**信号与槽(signal and slot)**

信号和槽是QT的核心机制，也是与标准C++不一样的地方，它是一种高级接口，应用于对象之间的通信；

信号：当某个对象的内部状态发生变动，信号被一个对象发射（只有定义这个信号的类能及其派生类能够发射这个信号），当信号被发射时，与其关联的槽将被立马执行，就像正常调用一个函数一样，如果一个signal与多个slot关联，那么这些slot会一个接一个无序地执行，执行完所有的slot后，signal都会返回；

槽：槽是普通的c++成员函数，能被正常调用，它们唯一特别的就是能够与信号相关联，当与其关联的信号发射时，这个槽就会被调用，槽可以有参数，但是不能有缺省值；

要想使用信号与槽，就必须要继承QT提供的QObject这个父类，并且在类的开始加上Q\_OBJECT这个关键字，再在类中定义相就的signal和slot，并将signal和slot通过connect关联起来，那么在定义的signal被发送(emit)后，应付执行相应的slot函数；

以下有一个小例子来说明信号与槽该如何定义以及使用；

class A {

……

signals:

void sig\_print();

slots:

void slot\_print();

……

}

……

connect(a, &A::sig\_print, a, &A::slot\_print);

……

emit sig\_print();

……

当emit sig\_print()这条语句被执行量，与之相关联的slot\_print()也会执行，connect可以用于相同对象，还可以用于不同对象，不同线程之间，connect函数除了前四个参数外，还有第五个可选的参数,类型为Qt::ConnectionType的枚举变量，以下四个取值：

Qt::AutoConnection：默认值，使用这个值则连接类型会在信号发送时决定。如果接收者和发送者在同一个线程，则自动使用Qt::DirectionConnection类型，如果接收者和发送者不在同一个线程中，则自动使用Qt::QueuedConnection类型；

Qt::DircetConnection：槽函数会在信号发送的时候直接调用，槽函数运行于信号发送所在的线程，效果看上去就像是直接在信号发送位置调用了槽函数，这个在多线程环境下比较危险，可能会导致崩溃；

Qt::QueuedConnection：槽函数会在控制回到接收者所在事件循环时被调用，槽函数运行于信号接收者所在线程。发送信号之后，槽函数不会立刻被调用，而是等待接收者当前函数执行完成，进行事件循环之后，槽函数才会执行。多用于多线程的环境之中；

Qt::BlockingQueuedConnection：槽函数的运行机制和Qt::QueuedConnection一样，不过在发完信号后，发送者所在的线程会被阻塞，直到槽函数运行完。接收和和发送者绝对不能在同一个线程中，否则会死锁。在多线程同步的场合可能会用到这个 ；

Qt::UniqueConnection：这个flag可以通过按位或(|)与以上四个一起使用，当使用这个flag时，当某个信号和槽已经连接时，再进行重复的连接就会失败。也就是避免重复连接；

**元对象编译器(MOC)**

MOC的实现是一个预处理器，在代码的编译时期，MOC会读取C++源代码，如果它发现其中包含一或多个类的声明中含有Q\_OBJECT宏，它就会给含有Q\_OBJECT宏的类生成另一个含有元对象代码的C++源文件，这个生成的源文件可以被类包含(#include)或者和这个类的实现一起编译和链接；

如果要作用QT中的信号与槽应必须要在类的开关加上Q\_OBJECT宏，在信号声明前加上signals:的关键字，在槽的声明前加上slots:的关键字，这样才能够正常使用信号与槽的特性。

## 4.2 HTTP数据发送

在QT的网络模块中，要实现HTTP数据的发送并不是通过网络连接来直接发送的，而是通过专门的管理类QNetworkAccessManager来实现网络数据的发送与接收，要请求的数据放在QNetworkRequest对象中，这样的设计可以让用户更加专注与应用的开发而无需过多的关心底层协议的实现；

首先完成如图4.1界面设计



图4.1 HTTP数据发送界面

最大的那个白色的框是一个QPlainTextEdit对象，主要用于处理文本输入，在这个功能中的作用是读取用户输入的URL请求附带的参数；左下方的小下拉列表是一个QComboBox对象，一共有两个选项:GET和POST，用于让用户选择使用哪一种请求方式，右下角的发送按钮，是一个QPushButton对象，在用户输入好URL和参数，选择好请求方式，后点击发送按钮；这里程序会读取URL输入框的内容，以及QComboBox中的选择，以确定要使用哪一种请求方式；

发送POST请求：

void SendDataHttp::request\_by\_post(const QString &text)

{

std::string str = text.toStdString();

int pos = str.find\_first\_of('?');

std::string url = std::string(str, 0, pos);

std::string params = std::string(str, pos+1, str.size()-url.size()-1);

QNetworkAccessManager \*manager = new QNetworkAccessManager(this);

QNetworkRequest request;

request.setUrl(QUrl(QString(tr(url.c\_str()))));

manager->post(request, QByteArray(params.c\_str(), params.size()));

connect(manager, &QNetworkAccessManager::finished, this, &SendDataHttp::received\_Data);

}

发送GET请求，

void SendDataHttp::request\_by\_get(const QString &url)

{

QNetworkAccessManager \*manager =

new QNetworkAccessManager(this);

QNetworkRequest request;

request.setUrl(QUrl(url));

manager->get(request);

connect(manager, &QNetworkAccessManager::finished, this, &SendDataHttp::received\_Data);

}

收到HTTP响应后的处理，收到HTTP数据响应扣，根据HTTP头中所携带的HttpContent-Type信息来选择以哪一种文件格式来保存所收到的数据，当保存完数据后，新建一个进程，使用Windows系统的资源管理器来打开保存在磁盘上的文件，至于到底以哪一种方式来打开该文件，完全由用户所使用的操作系统确定，若没有安装打开此类型文件的应用，Windows系统会提示用户选择以哪一种应用来打开此文件，若实在找不到，也可以选择联网查找；比如说收到的数据是html文件，那么Windows系统就会使用默认浏览器来打开该文件；

void SendDataHttp::received\_Data(QNetworkReply \*reply)

{

if(!reply) {

qDebug() << "QNetworkReply is nullptr";

} else if(reply->error() == QNetworkReply::NoError){

QString filename = "text";

std::string all\_header = reply->header(QNetworkRequest::ContentTypeHeader).toString().toStdString();

/\* http contentType 也可能分为多段，用分号隔开 \*/

std::string header = all\_header.substr(0, all\_header.find\_first\_of(';'));

/\* getSuffix函数可以根据HTTP响应头中的Content-Type来确定文件以哪一种后缀的文件名来保存 \*/

filename.append(getSuffix(header).c\_str());

QByteArray data = reply->readAll();

QFile recv(filename);

if(recv.exists()){

recv.remove();

}

if(!recv.*open*(QFile::WriteOnly)){

qDebug() << "open file error";

}

recv.write(data);

recv.*close*();

QProcess::execute("explorer.exe", QStringList() << filename);

} else {

qDebug() << reply->errorString();

}

}

## 4.3 TCP数据发送

与传统的TCP数据发送方法一样，QT的TCP数据发送功能也是通过新建一个套接字来实现的，并用该套接字来发送网络数据，但是与之不同的是QT的套接字实现了自己的封装，且在入了信号与槽，更加容易使用；

如图4.2为TCP数据发送的界面图，主机输入框是一个QLineEdit对象，用于输入目标IP地址，端口输入框也是一个QLineEdit对象，右边的发送按钮和高级按钮都是QPushButton对象，点击高级按钮后，会弹出让用户选择加密和压缩算法的一个对话框，点击确定后，就会使用当前的加密和压缩生效；点击发送按钮后就会读取当前的配置，用户输入的目标主机、端口、以及构建的数据（若需要再进行压缩、加密操作），之后向目标主机发送数据，收到响应的数据后，会自动打开数据解析界面（数据解析界面将会在后面介绍）；

界面的再下面一点，标有数字1、2和一个添加按钮的地方是一个QListWidget对象，在QListWidget面板中，放入了一个item，item中有一个大的添加按钮，一起在QListWidget的最下方，点击添加按钮则在添加按钮的上面一格插入一个item其格式与标号1、2的一样，点击item右边的删除按钮，则会删除对应的item，并更新item最左方的序号；

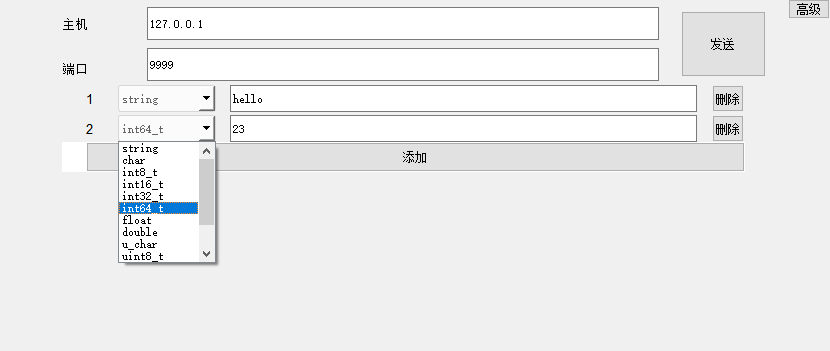


图4.2 TCP数据发送界面

底部添加按钮实现

\_btnAdd = new QPushButton(tr("添加"));

QListWidgetItem \*item\_buttom = new QListWidgetItem(\_listWidget);

\_listWidget->setItemWidget(item\_buttom, \_btnAdd);

\_listWidget->addItem(item\_buttom);

connect(\_btnAdd, &QPushButton::clicked, this, &SendDataTcp::addListItem);

添加item函数（只有核心部分的代码，细节见源码）

void SendDataTcp::addListItem()

{

QListWidgetItem \*item = new QListWidgetItem();

SendDataItem \*widget =

new SendDataItem(\_listWidget->count(), \_itemSize);

\_listWidget->insertItem(\_listWidget->count()-1, item);

\_listWidget->setItemWidget(item, widget);

\_listWidget->scrollToBottom();

connect(widget, &SendDataItem::removeItemSignal, this, [=](){

\_listWidget->removeItemWidget(item);

if(nullptr != item) delete item;

if(nullptr != widget) delete widget;

});

}

网络数据传输部分

/\* socket初始化 \*/

\_tcpSocket = new QTcpSocket(this);

connect(\_tcpSocket, &QTcpSocket::connected,

this, &SendDataTcp::sendTcpData);

connect(\_tcpSocket, &QTcpSocket::readyRead,

this, &SendDataTcp::receivedTcpData);

……

/\* 连接到指定主机：端口 \*/

\_tcpSocket->connectToHost(QHostAddress(host) , port);

……

/\* 发送用户构造的数据 \*/

QByteArray bytes;

for(int idx = 0; idx < \_listWidget->count(); idx++) {

SendDataItem \*item = dynamic\_cast<SendDataItem\*>(

\_listWidget->itemWidget(\_listWidget->item(idx)));

if(nullptr == item) continue;

bytes.append(item->getItemData());

}

/\* 如果需要，这里执行压缩加密的操作 \*/

\_tcpSocket->write(bytes);

……

接收数据，并调用ParseData模块，解析收到的数据，并关闭连接

QByteArray bytes = \_tcpSocket->readAll();

ParseData \*parseData = new ParseData(bytes);

parseData->show();

\_tcpSocket->close();

单条数据构造部分：在本次设计中，为单条数据的构造专门创建了一个类，用来 构造数据，在需要的时候，返回构造完成的单条数据，



图4.3 单条数据界面

如图4.3 最左边为序号，方便用户查看当前一共输入了多少条数据，具体第几条的数据是多少，这一点非常重要，而且界面做过了特殊处理，在用户删除单条item时，会更新所有的序号，使其变得连续且有序，紧跟序号之后的是一个下拉菜单，用户可以通过这个下拉菜单来选择需要发送的数据类型，其中包括了现在会用到的所有基本数据类型char、short、int、long、float、double，当然也加入了字符串的支持；在中间的输入框中，需要输入用户相应的数据，记住，创建了一个item是必须要输入数据的，不然发送的数据可能会与预期的不一样，而软件不会做出提示；当用户点击发送按钮后，程序会遍历所有的item，通过item的对象获取存放在item中的用户输入的数据，并将所有的数据存起来，再处理，发送；

但是这里有一个很大的问题，那就是float的数据类型，与其它数据会有所不同，比如说0x12345678和0x11223344这两个数据在windows上是这样存储的（数据高位在内存低位，数据低位在内存高位，也就是小端字节序）



图4.4 小端字节充在内存中的存放

但是float数据的存取与这个存法完全相反，它是数据高位在内存地址高位，数据低位在内存地址低位，完全是按照大端序的存法来的，这里我不知道是QByteArray和QDataStream的问题还是float本身是这样的（或者是编译器实现的问题），所以我对float的数据类型做了特殊处理，在返回字节流的时候，我把它的数据反过来存：

char ptr[4]; memset(ptr, 0x00, sizeof(ptr));

float f = value.toFloat();

memcpy(ptr, &f, sizeof(float));

for(int i=3; i>=0; i--) bytes.append(ptr[i]);

关于字节序的大端与小端，在TCP数据解析的时候会专门说明；

## 4.4 TCP数据解析

TCP数据解析这个部分，相对而言要简单一些，它的功能就是给它一个QByteArray，然后按照指定的方式显示出来就可以了，不过显示的方式分两种：第一种是直接以纯文本字符串来显示，不可见字符或‘\0‘的结束符用’.’来代替：

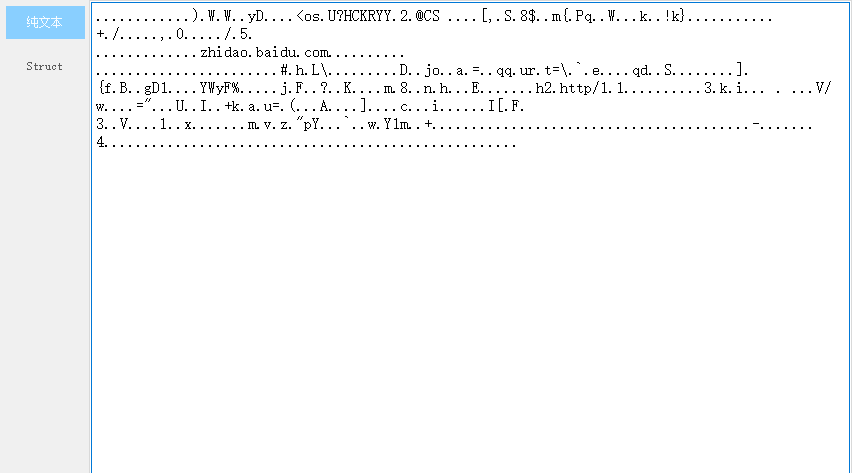


图4.5 以纯文本的方式解析数据

另一种方式是接收到的数据如果是原始的非字符串的二进制数据流，那么就需要用专用的显示方法来显示了，点击界面左侧栏上的struct，就进入了二进制数据流解析的过程中了：

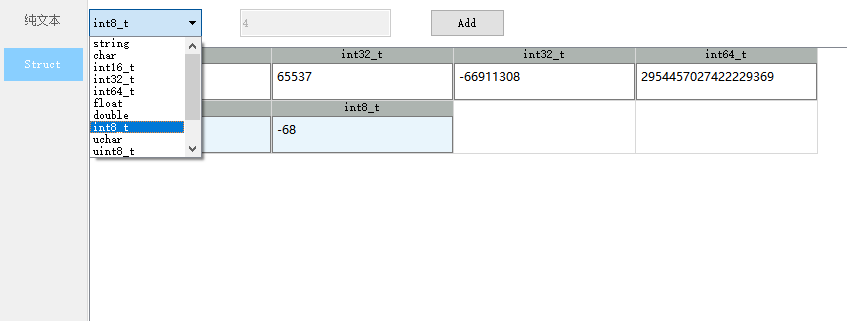


图4.6 解析非字符串的二进制数据流

数据解析的界面如图4.6，与TCP数据的构造类似，用户也可以选择将要解析的数据类型，选择之后点击add添加，下方的QTableWidget就会多出一个Iteme用来显示解析也来的对应数据，如果要查看指定位上的数据，可以先读取一个string，设置好长度，点击添加，这个时候，数据就偏移了刚刚设定的位数，再读取自己想要的数据就可以了；

但是在实现的时候会有大问题，因为数据在网络中的义传输作用的是大端字节序(big endian)，而在windows等主流的操作系统上，数据在内存中是以小端字节序(little endian)来存放的，直接举个例子：

两个四字节的数据0x12345678和0x11223344以小端的方式来存储，在内存中是这样的：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0x78 | 0x56 | 0x34 | 0x12 | 0x44 | 0x33 | 0x22 | 0x11 |

内存增长的方向

而以大端方式来存是这样的：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0x12 | 0x34 | 0x56 | 0x78 | 0x11 | 0x22 | 0x33 | 0x44 |

矛盾显而易见，网络中传输用大端，计算机内传输用小端，这两个读出来根本不是同一个东西，所以我们需要对机器的大小端进行判断，之后再选择以大端还是小端来处理数据，isLittle函数，如果机器是小端字节序，那么返回true，如果机器是大端字节序，返回false；

inline bool isLittleEndian ()

{

int32\_t a = 1;

char \*p = (char \*)&a;

return (1 == \*p);

}

要解析成指定的数据，首先要从原来的数据中读取指定长度位的数据，才能进行解析，同时在读取指定长度时，还要考虑大端字节序和小端字节序的问题，所以这里封装了一个函数，专门用来处理这个问题，该函数首先判断数据是否已经解析完了，如果已经解析完，直接返回，没有解析完成，才进行接下来的操作，从网络数据中读取指定位数，然后完成转化，返回一个char\*的指针，里面存放了解析完成的原始数据：

char\* ParseDataTcp::getDigital(int length)

{

if(length + \_currIndex > \_datas.size()) return NULL;

int r\_idx;

char \*ret = new char[16];

memset(ret, 0x00, 16);

if(!isLittleEndian()) { /\* big endian \*/

for(r\_idx=0; r\_idx<length; r\_idx++) {

ret[r\_idx] = \_datas.at(\_currIndex + r\_idx);

}

} else { /\* little endian \*/

for(r\_idx = length-1; r\_idx>=0; r\_idx--) {

ret[r\_idx] = \_datas.at(\_currIndex + length - r\_idx -1);

}

}

\_currIndex += length;

return ret;

}

如果要解析int类型的数据，直接使用字符串转int函数atoi即可完成，最后再将数据和数据类型在QTableWidget上面显示出来；

## 4.5 数据压缩解压、加密解密

RC4算法的特点是算法简单，运行速度快，而且密钥长度可变(1-256字节)，在如今的技术下，当密钥的长度达到16字节时，用暴力搜索算法已经不太可行，所以RC4在现阶段而言是相对比较安全的对称加密算法；

在实现RC4算法之前有必要先介绍一下算法中涉及的基本东西；

密钥流：RC4算法的关键是根据明文和密钥生成相应的密钥流，密钥流的长度和明文的长度是一样的，加密生成的密文长度也是一样的，因为密文的第i字节=明文第i字节^密钥流的第i字节；

状态变量S：长度为256，第个单元都是一个字节，算法运行在任何时候，S都包括0~255的9比特数的排列组合，只不过数值的位置发生了变化；

密钥K：度为1~256字节，也就是所说的密码，通常取16位；

加密步骤：设置密钥和密钥流

void RC4::setKey(const char \*key, int keyLen)

{

\_keyLen = keyLen;

memcpy(\_key, key, keyLen);

unsigned char k[MAX\_LEN] = { 0 };

for (int i = 0; i < MAX\_LEN; i++) {

\_box[i] = i;

k[i] = \_key[i % \_keyLen];

}

for (int i = 0, j = 0; i < MAX\_LEN; i++) {

j = (j + \_box[i] + k[i]) % MAX\_LEN;

std::swap(\_box[i], \_box[j]);

}

}

RC加密与解密，因为RC4加密算法的原因，其实它的加密解密完全是同一个步骤：

void RC4::rc4(unsigned char \* data, int len)

{

unsigned char tBox[MAX\_LEN] = { 0 };

memcpy(tBox, \_box, MAX\_LEN);

int i = 0, j = 0, t = 0;

unsigned long k = 0;

for (k = 0; k < len; k++) {

i = (i + 1) % MAX\_LEN;

j = (j + tBox[i]) % MAX\_LEN;

std::swap(tBox[i], tBox[j]);

t = (tBox[i] + tBox[j]) % MAX\_LEN;

data[k] ^= tBox[t];

}

}

至于数据压缩算法，直接从网上下载相应的加密库在程序中调用即可，下面列出压缩算法所会用到的方法：

数据压缩算法：destination必需是已经分配内存的指针

size\_t qlz\_compress(const void \*source, char \*destination, size\_t size, qlz\_state\_compress \*state);

传入压缩后的数据，即可获得解压后数据的大小，可以决定分配多大的内存：

size\_t qlz\_size\_decompressed(const char \*source);

数据解压函数：destination必须是已经分配内存的指针，并用分配空间的大小必须要大于或等于qlz\_size\_decompressed的返回值：

size\_t qlz\_decompress(const char \*source, void \*destination, qlz\_state\_decompress \*state);

## 4.6 数据监听

数据监听这个功能是这个软件实现中最为复杂的一个部分，数据监听函数一旦开始执行，那么它的主循环便会一直执行，并不会退出，听起来好像没有什么问题，如果把数据监听的线程与主界面放在同一个线程中的话，那就意味着主界面中的代码会一直处于阻塞状态，在图形界面中这个会弹出程序未响应的提示框，用户点什么操作它都不会响应；唯一的解决办法就是把这个会阻塞主线程的函数放到一个单独的子线程中，有什么状态需要改变的时候，再发送信号通知子线程进行相应的操作，或者子线程发送消息给主线程；另一个问题是网络中的数据可能会很大，如果要把这些数据全部存放在内存中，那么数据太多的时候，内存会完全不够用，从而导致系统崩溃，所以这里的解决方法是每收到一条数据，全将数据保存到磁盘中，需要的时候再去读取；最后一个问题是这个文件是两个线程共用的，所以可能会出现等待的问题，也有可能使主界面假死的情况，所以又需要新加一个线程专门用来读取在磁盘中的文件；

综合上面的分析，除主线程外，还需要单独创建两个线程，用与数据监听和磁盘文件读取；

现在程序的框架已经出来了，那就是在主界面创建完成之后，再新开两个线程，一个用于数据监听，另一个用于读取保存在本地的网络数据文件；数据监听该怎么来实现呢？直接从硬件获取？还好已经有现成的API可以帮我们完成这一件事情——winpcap，作为一个开源的程序库，winpcap提供了几个非常强大的功能：捕获原始的数据包，包括在共享网络中各主机发送/接收的以及相互之间交换的数据包；对于网络中发来的数据，可以对数据进行自定义的筛选；也可以发送原始的数据包；对网络中的数据进行分析与统计；

在这次实现的软件中，主要使用的是winpcap的数据捕获能力和网络数据筛选的功能；

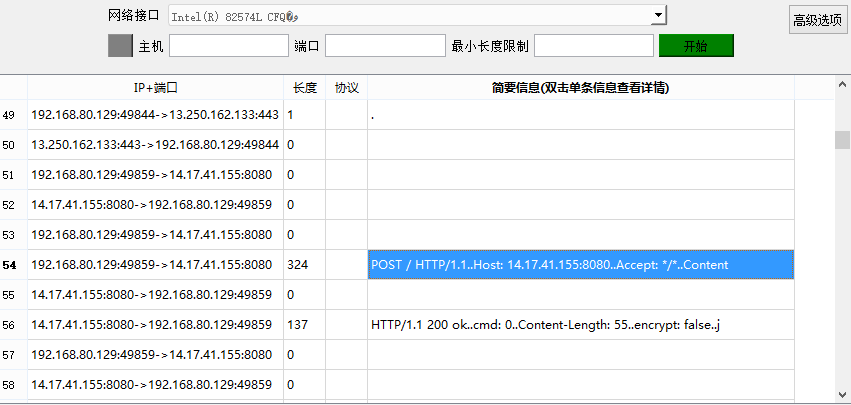


图4.7 网络数据监听界面

界面中显示了数据包的数量，数据包是第几个收到的，数据报文的源地址和目的地址，TCP上层应用层协议的数据长度，以及数据的简要信息；所有的数据都放在一个QTableWidget中间，双击单条数据即可查看数据的详细信息，该功能屏蔽了网络底层实现的细节，让用户更加专注于应用层传输的内容；

winpcap是一个第三方的库，使用前一定要include头文件，在工程文件.pro中加入.lib库，这样才能正常编译：

# winpcap

INCLUDEPATH += $$PWD/winpcap/Include

LIBS += $$PWD/winpcap/Lib/wpcap.lib

LIBS += $$PWD/winpcap/Lib/Packet.lib

另外，由于winpcap还引用了winsock的内容，所以需要链接winsock的lib库，在.pro文件中加入以下内容（有#号的行代表注释）：

# ws2\_32.lib Windows Socket

LIBS += -lws2\_32

要抓取网络数据，首先通过winpcap提供的API获取所有的网络设备：

vector<pcap\_if\_t\*> SystemDevice::init()

{

pcap\_if\_t \*devs;

char errbuff[PCAP\_ERRBUF\_SIZE + 1];

if(-1 == pcap\_findalldevs(&devs, errbuff)) {

return false;

}

vector<pcap\_if\_t\*> \_allDevs;

for(pcap\_if\_t \*dev = devs; dev; dev = dev->next) {

\_allDevs.push\_back(dev);

}

return \_allDevs;

}

当网络接口获取完成后，所有的网络接口信息将会在界面上的下拉菜单中显示，这时用户可以通过下拉菜单来选取抓取哪个接口上的数据；选择数据后在下面的筛选选项中可以对网络中的数据进行筛选，筛选完成后，就可以点击开始按钮进行数据抓取了，下面给出了数据抓取的核心代码：

void DumpInterface::capturePacket(const pcap\_if\_t \*dev)

{

const char \*devName = dev->name;

char errbuf[PCAP\_ERRBUF\_SIZE + 1];

\_adhandle = pcap\_open\_live(devName, 65536, 1, 1000,errbuf);

/\* build filter \*/

struct bpf\_program fcode;

pcap\_compile(\_adhandle, &fcode, "tcp and ip", 1, 0x00ffffff);

pcap\_setfilter(\_adhandle, &fcode);

/\* start capture \*/

int res = 0;

pcap\_pkthdr \*pkt\_header;

const u\_char \*pkt\_data;

while(\_isContinue && 0 <= (res = pcap\_next\_ex(\_adhandle, &pkt\_header, &pkt\_data))) {

if(0 == res) continue;

on\_packet\_received(pkt\_header, pkt\_data);

}

}

从网络上抓取到数据之后，就会调用on\_packet\_received(pcap\_pkthdr \*,u\_char\*)这个函数，处理发来的数据，根据网络中发来的不同类型的协议，作不同的处理，它会自动忽略掉除TCP和UDP以外的数据报文，当接收到数据后，它会先根据14位的MAC长度计算出IP数据头的起始位置，再根据IP头的头部长度信息计算也TCP数据报的头部，再根据TCP的数据头部长度计算出真实的数据起始位置，TCP上层协议的数据长度可以用IP数据报的长度减去IP头部长度+TCP头部长度；

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

|Version| IHL |Type of Service| Total Length |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Identification |Flags| Fragment Offset |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Time to Live | Protocol | Header Checksum |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Source Address |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Destination Address |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Options | Padding |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

图4.8 IP数据报头部

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Source Port | Destination Port |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Sequence Number |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Acknowledgment Number |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Data | |U|A|P|R|S|F| |

| Offset| Reserved |R|C|S|S|Y|I| Window |

| | |G|K|H|T|N|N| |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Checksum | Urgent Pointer |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Options | Padding |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| data |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

图4.9 TCP数据报文头部

跟之前一样，因为在X86体系的机器上，都是小端字节序，所以对网络中发回来的数据，需要将大端转为小端，以下分别是大端16位转小端和32位转小端的高效方法（核心代码，具体细节见源码）：

inline uint16\_t \_\_ntohs(uint16\_t x) {

return ((x >> 8) & 0x00ff) | ((x & 0x00ff) << 8);

}

inline uint32\_t \_\_ntohl(uint32\_t x) {

return ((((x) & 0xff000000) >> 24) | (((x) & 0x00ff0000) >> 8) |

(((x) & 0x0000ff00) << 8) | (((x) & 0x000000ff) << 24) );

}

还有就是在此功能模块中用到的多线程技术，QT中多线程有多种使用方法，这里采用QT官方较为推崇的写法：

将要执行的耗时操作放到一个继承自QThread类的子类中；

关联必要的信号与槽，用于和主线程之间通信；

创建一个新的QThread类对象和我们的线程类对象；

将要执行的线程移交到QThread对象中去；

调用QThread::start()方法，开始执行子线程；

\_captureThread = new DumpThread(); // DumpThread继承自QThread

connect(\_captureThread, &DumpThread::sig\_tcpReceived, this, &CaptureWidget::deal\_Tcp, Qt::QueuedConnection); //主线程与子线程通信

QThread \*mainLoopThread = new QThread(); // 创建一个子线程

\_captureThread->moveToThread(mainLoopThread); // 将子线程移交到新的线程

mainLoopThread->start(); // 开始新的线程

剩下的就都一些界面方面的代码，太大的技术难点要看实现细节的参照源码；

# 第五章 系统测试

5.1 测试环境简介

5.2 各功能模块的测试

# 第六章 总结

内容

参考文献