



企业实训QT方向

-第8课：Qt网络通信编程

2021合肥工业大学软件学院实训项目 | 2021

时代红

2021-07-05



目录

CONTENTS

01 HTTP客户端

02 FTP客户端

03 网络接口信息获取

04 UDP/TCP编程

05 课后练习作业

01

HTTP客户端

Strive to become a better version of yourself

QtNetwork模块 Network Introduction

Qt提供了QtNetwork模块来进行网络编程。该模块提供了诸如QFtp等类来实现特定的应用层协议；有较低层次的类，例如：QTcpSocket/QTcpServer和QUdpSocket等来表示低层的网络概念；还有高层次的类，例如：QNetworkRequest/QNetworkReply和QNetworkAccessManager使用相同的协议来进行网络操作；也提供了QNetworkConfiguration/QNetworkConfigurationManager和QNetworkSession等类来实现负载管理。如果要使用QtNetwork模块中的类，需要在项目文件中添加“QT += network”一行代码。

我们也可以在帮助系统中查找Network Programming关键字，来查看相关帮助文档。最后我们会结合讲解一个网络模块的综合应用实例程序。



HTTP消息结构(1/3)

HTTP消息结构

HTTP Protocol Introduction

HTTP (HyperText Transfer Protocol, 超文本传输协议) 是一个客户端和服务端请求和应答的标准。

The screenshot displays the 'Raw' tab of a web browser's developer tools, showing an HTTP POST request and its corresponding response. The request is to `http://httpbin.org/post` with a `Content-Type: application/json` header and a JSON body. The response is a `200 OK` status with a `Content-Type: application/json` header and a JSON body. The JSON body of the response contains the details of the request, including the headers, the request body, and the origin.

Labels and arrows pointing to the request and response structure:

- 请求行** (Request Line): Points to the first line of the request, `POST http://httpbin.org/post HTTP/1.1`.
- 请求头** (Request Header): Points to the headers section of the request, including `Content-Type: application/json`, `Content-Length: 35`, `Connection: Keep-Alive`, `Accept-Encoding: gzip, deflate`, `Accept-Language: zh-CN,en,*`, `User-Agent: Mozilla/5.0`, and `Host: httpbin.org`.
- 空行** (Empty Line): Points to the blank line separating the request headers from the request body.
- 请求体** (Request Body): Points to the JSON body of the request, `{"Password": "123456", "User": "Qter"}`.
- 状态行** (Status Line): Points to the first line of the response, `HTTP/1.1 200 OK`.
- 响应头** (Response Header): Points to the headers section of the response, including `Server: nginx`, `Date: Thu, 15 Sep 2016 05:22:59 GMT`, `Content-Type: application/json`, `Content-Length: 468`, `Connection: keep-alive`, `Access-Control-Allow-Origin: *`, and `Access-Control-Allow-Credentials: true`.
- 空行** (Empty Line): Points to the blank line separating the response headers from the response body.
- 响应体** (Response Body): Points to the JSON body of the response, which contains the details of the request.

HTTP消息结构(2/3)

HTTP消息结构

HTTP Protocol Introduction

- Request

请求行：Request 消息中的第一行，由请求方式、请求URL、HTTP协议及版本三部分组成。

请求头：其中 Content-Type 指定了客户端发送的内容格式。例如：`Content-Type: application/json`，指客户端发送的内容格式为 Json。

请求体：要发送的表单数据。

HTTP消息结构(3/3)

HTTP消息结构

HTTP Protocol Introduction

- Response

状态行：Response 消息中的第一行，由 HTTP 协议版本号、状态码、状态消息三部分组成。状态码用来告诉 HTTP 客户端，HTTP 服务器是否产生了预期的 Response。HTTP/1.1 中定义了 5 类状态码，状态码由三位数字组成，第一个数字定义了响应的类别：

- 1XX：提示信息 - 表示请求已被成功接收，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处理。
- 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。

响应头：其中 Content-Type 指定了服务器返回的内容格式。例如：`Content-Type: application/json`，指服务器返回的内容格式为 Json。

响应体：服务器返回的内容。

在Qt的网络模块中提供了网络接口来实现HTTP编程，网络访问接口是执行一般网络操作的类集合，该接口在特性的操作和使用的协议，如通过HTTP进行获取和发送数据，在此协议之上提供了一个抽象层，只为外部暴露除了类、函数和信号。网络请求由`QNetworkRequest`类来表示，也作为与请求有关的信息（例如：任何头信息和使用加密）的容器。在创建请求对象时指定的URL决定了请求使用的协议，目前支持HTTP、FTP和本地文件URLs的上传和下载。

`QNetworkAccessManager`类用来协调网络操作，每当一个请求创建后，该类用来调度它，并发射信号来报告进度。该类还协调cookies的使用、身份验证请求及其代理的使用。对于网络请求的应答使用`QNetworkReply`类来表示，它会在请求被完成调度时由`QNetworkAccessManager`来创建。`QNetworkReply`提供的信号可以用来单独监视每一个应答，当然也可以使用`QNetworkAccessManager`的信号来实现。因为`QNetworkReply`是`QIODevice`的子类，应答可以使用同步或者异步的方式来处理，例如阻塞或者非阻塞的操作。每一个应用程序或者库文件都可以创建一个或者多个`QNetworkAccessManager`实例来处理网络通信。

在进行网络请求之前，首先，要查看 `QNetworkAccessManager` 支持的协议：

```
1 QNetworkAccessManager *manager = new QNetworkAccessManager(this);
2 qDebug() << manager->supportedSchemes();
```

通过调用 `supportedSchemes()`，列出了支持的所有 URL schemes：

```
("ftp", "file", "qrc", "http", "https", "data")
```


创建HTTP请求

构建一个请求非常简单，本例中，我们尝试获取某个网页，以 CSDN 为例：

```
1 QString baseUrl = "http://www.csdn.net/";  
2  
3 QNetworkRequest request;  
4 request.setUrl(QUrl(baseUrl));
```

现在，名为 request 的 QNetworkRequest 请求对象就构建成功了，为了获取所有想要的信息，我们需要将该请求发送出去。

```
1 QNetworkReply *pReplay = manager->get(request);
```

这时，会获取一个名为 pReplay 的 QNetworkReply 响应对象。等待响应完成，就可以从这个对象中获取所有想要的信息。

```
1 QByteArray bytes = pReplay->readAll();  
2 qDebug() << bytes;
```

创建HTTP请求

注意：开启事件循环的同时，程序界面将不会响应用户操作（界面被阻塞）。因为请求的过程是异步的，所以此时使用 QEventLoop 开启一个局部的事件循环，等待响应结束，事件循环退出。

简便的 API 意味着所有 HTTP 请求类型都是显而易见的。那么其他 HTTP 请求类型：POST，PUT 又是如何的呢？都是一样的简单。

```
QNetworkReply *pPostReplay = manager->post(request, QByteArray());
```

```
QNetworkReply *pPutReplay = manager->put(request, QByteArray());
```

参考代码：

```
// URL
QString baseUrl = "http://www.csdn.net/";
// 构造请求
QNetworkRequest request;
request.setUrl(QUrl(baseUrl));
QNetworkAccessManager *manager = new QNetworkAccessManager(this);
// 发送请求
QNetworkReply *pReplay = manager->get(request);
// 开启一个局部的事件循环，等待响应结束，退出
QEventLoop eventLoop;
QObject::connect(manager, &QNetworkAccessManager::finished, &eventLoop, &QEventLoop::quit);
eventLoop.exec();
// 获取响应信息
QByteArray bytes = pReplay->readAll();
qDebug() << bytes;
```

传递URL参数

你也许经常想为 URL 的查询字符串（query string）传递某种数据。如果你是手工构建 URL，那么数据会以键/值对的形式置于 URL 中，跟在一个问号的后面。

例如：<http://www.zhihu.com/search?type=content&q=Qt>。

Qt 提供了 `QUrlQuery` 类，可以很便利地提供这些参数。更多参考：[Qt之QUrlQuery](#)

举例来说，如果你想传递 `type=content`和`q=Qt`到<http://www.zhihu.com/search>，可以使用如下代码：

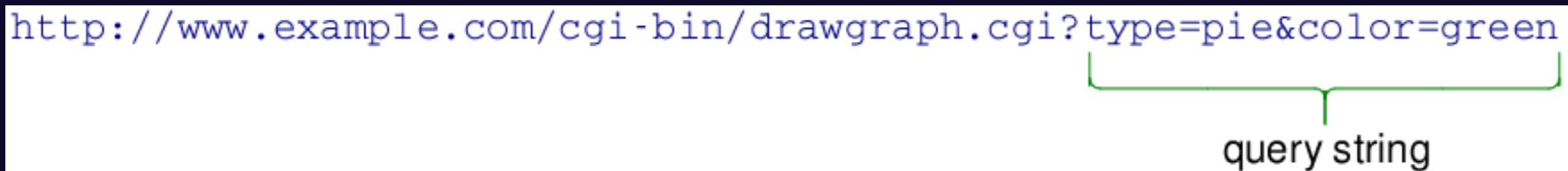
```
// URL  
  
QString baseUrl = "http://www.zhihu.com/search";  
QUrl url(baseUrl);
```

```
// key-value 对  
  
QUrlQuery query;  
query.addQueryItem("type", "content");  
query.addQueryItem("q", "Qt");
```

```
url.setQuery(query);  
  
通过打印输出该 URL，你能看到 URL 已被正确编码：  
  
QDebug() << url.url();  
  
// http://www.zhihu.com/search?type=content&q=Qt
```

QUrlQuery

描述：QUrlQuery 用来解析 URL 中的查询字符串，像下面这样：



The diagram shows the URL `http://www.example.com/cgi-bin/drawgraph.cgi?type=pie&color=green`. A green bracket is drawn under the query string `?type=pie&color=green`, with the text "query string" centered below the bracket.

上述的查询字符串在 URL 中 被用来传输选项，通常解码为多个 key-value 对。其列表包含了的两个条目，键为 “type” 和 “color”。QUrlQuery 也适用于从查询的各个组成部分创建一个查询字符串，为了 在 `QUrl::setQuery()` 中使用。

解析一个查询字符串最常见的方式是在构造函数中始化它，通过传递一个查询字符串。否则，可以使用 `setQuery()` 函数来设置要解析的查询。该函数也可用于解析具有非标准分割符的查询，在设置它们之后使用 `setQueryDelimiters()` 函数。

编码的查询字符串可以再次使用 `query()` 获得，这需要所有的内部存储项，并使用分隔符编码字符串。

编码：

QUrlQuery 中的所有 getter 函数均支持一个可选参数 `QUrl::ComponentFormattingOptions` 类型，包括 `query()`，它决定如何编码数据。除了 `QUrl::FullyDecoded`，返回值必须被视为一个百分比编码字符串。由于某些值不能在解码形式（如控制字符，字节序列不能被解码为 UTF-8）来表达。出于这个原因，百分比字符总是由字符串 “%25” 表示。

处理空格和加号（“+”）：空格应该被编码成加号（“+”），而如果字符本身就是加号（“+”），则应该被编码成百分比编码格式（%2B）然而，互联网规范管理 URL 不认为空格和加号字符等价。

由于这样，QUrlQuery 不会将空格字符编码为 “+”，也不会将 “+” 解码为一个空格字符。相反，空格字符将在编码形式中呈现 “%20”。为了支持这样的 HTML 表单编码，QUrlQuery 既不会将 “%2B” 序列解码为一个加号，也不会编码一个加号。事实上，任何键、值、查询字符串中的 “%2B” 或 “+” 序列完全像写的一样（除了 “%2b” 到 “%2B” 大写转换）。

全解码：

使用 `QUrl::FullyDecoded` 格式化，所有百分比编码序列将被完全解码，并且 `'%'` 字符用于表示本身。应小心使用 `QUrl::FullyDecoded`，因为可能会导致数据丢失。这种格式化模式应该只在这种情况下使用：当在不期望百分比编码的上下文中处理呈献给用户的文本时。

注意：

`QUrlQuery` `setters` 和 `query` 函数不支持对应 `QUrl::DecodedMode` 解析，所以使用 `QUrl::FullyDecoded` 获得 `keys` 列表可能导致在对象不能找到 `keys` 。

非标准分隔符：

默认情况下，`QUrlQuery` 使用等号 (`"="`) 来分隔 `key` 和 `value`，符号 (`"&"`) 分割彼此的 `key-value` 对。通过调用 `setQueryDelimiters()`，可以改变 `QUrlQuery` 用于解析和重构查询的分隔符。

非标准分隔符在 RFC 3986 的“sub-delimiters”中：

```
sub-delims = "!" / "$" / "&" / "'" / "(" / ")"  
            / "*" / "+" / "," / ";" / "="
```

不支持使用其他字符，可能会导致意外的行为。`QUrlQuery` 不验证是否传递了一个有效的分隔符。

```
http://www.zhihu.com/search?type=content&q=Qt
```

如果要构造一个这样的 URL，按照传统方式，一般是手动拼接字符串：

```
1 // 基本 URL
2 QString baseUrl = "http://www.zhihu.com/search?";
3
4 // 设置发送的数据
5 QByteArray bytes;
6 bytes.append("type=content&");
7 bytes.append(QString("q=%1").arg("Qt")); // Qt 作为变量输入
8
9 // 组合 URL
10 baseUrl += bytes;
11
12 QUrl url(baseUrl);
13
14 qDebug() << url;
```

显然可以实现，但相比之下，QUrlQuery 更为简单、方便。而且QUrlQuery 还提供了很多其他便利的接口。

QUrlQuery

使用 addQueryItem :

```
1 // 基本 URL
2 QString baseUrl = "http://www.zhihu.com/search";
3 QUrl url(baseUrl);
4
5 // key-value 对
6 QUrlQuery query;
7 query.addQueryItem("type", "content");
8 query.addQueryItem("q", "Qt");
9
10 url.setQuery(query);
11
12 qDebug() << url;
```

使用 setQueryItems :

```
1 // 基本 URL
2 QString baseUrl = "http://www.zhihu.com/search";
3 QUrl url(baseUrl);
4
5 QUrlQuery query;
6
7 // key-value 对
8 QPair<QString, QString> pair;
9 pair.first = "type";
10 pair.second = "content";
11
12 QPair<QString, QString> pair2;
13 pair2.first = "q";
14 pair2.second = "Qt";
15
16 QList<QPair<QString, QString> > items;
17 items << pair << pair2;
18
19 query.setQueryItems(items);
20 url.setQuery(query);
21
22 qDebug() << url;
```


使用 setQuery :

```
1 // 基本 URL
2 QString baseUrl = "http://www.zhihu.com/search";
3 QUrl url(baseUrl);
4
5 // 查询字符串
6 QUrlQuery query;
7 query.setQuery("type=content&q=Qt");
8
9 url.setQuery(query);
10
11 qDebug() << url;
```

分隔符

默认情况下，各个 key-value 对之间的分隔符为 `'&'`，而 key-value 之间的分隔符为 `'='`。

```
1 QChar pair = query.queryPairDelimiter(); // '&'  
2 QChar value = query.queryValueDelimiter(); // '='
```

如果要改变默认的分隔符，使用 `setQueryDelimiters()`：

```
1 query.setQueryDelimiters('(', ')');
```

这时，URL 就会变为这样：

```
QUrl("http://www.zhihu.com/search?type(content)q(Qt")
```

查询

查询所有的 key-value 对：

```
1 QList<QPair<QString, QString> > list = query.queryItems();  
2 // (QPair("type", "content"), QPair("q", "Qt"))
```

查询指定 key 对应的 value：

```
1 QString value = query.queryItemValue("q");  
2 // "Qt"
```

查询 key-value 对合并的字符串：

```
1 QString queryString = url.query();  
2 // "type=content&q=Qt"
```

查询指定 key 是否存在：

```
1 bool exist = query.hasQueryItem("q");  
2 // true
```

如果存在返回 true，否则返回 false.

删除

删除指定 key 对应的 key-value 对：

```
1 query.removeQueryItem("q");
```

清空当前存储的所有 key-value 对：

```
1 query.clear();
```

是否为空

如果 QUrlQuery 对象不包含 key-value 对，则返回 true；否则，返回 false。

```
1 query.isEmpty();
```

代理

在发送请求的时候，如果要通过 Fiddler 分析，就必须设置代理，主要使用 QNetworkProxy：

```
1 QNetworkProxy proxy;  
2 proxy.setType(QNetworkProxy::HttpProxy);  
3 proxy.setHostName("127.0.0.1");  
4 proxy.setPort(8888);  
5 manager->setProxy(proxy);
```

更多参考：[Qt之QNetworkProxy（网络代理）](#)

HTTP-复杂的POST请求

通常，你想要发送一些编码为表单形式的数据 – 非常像一个 HTML 表单。要实现这个，只需简单地传递一个QByteArray 给 data 参数。

你的数据在发出请求时会自动编码为表单形式：

```
// URL
QString baseUrl = "http://httpbin.org/post";
QUrl url(baseUrl);

// 表单数据
QByteArray dataArray;
dataArray.append("key1=value1&");
dataArray.append("key2=value2");

// 构造请求
QNetworkRequest request;
request.setHeader(QNetworkRequest::ContentTypeHeader, "application/x-www-form-urlencoded");
request.setUrl(url);
QNetworkAccessManager *manager = new QNetworkAccessManager(this);

// 发送请求
manager->post(request, dataArray);
```

HTTP-复杂的POST请求

还可以使用 json 参数直接传递，然后它就会被自动编码：为了不阻塞界面，我们不再使用 QEventLoop，而用 QNetworkAccessManager 对应的信号，当响应结束就会发射 finished() 信号，将其链接到对应的槽函数上即可。

```
// URL
QString baseUrl = "http://httpbin.org/post";
QUrl url(baseUrl);

// Json数据
QJsonObject json;
json.insert("User", "Qter");
json.insert("Password", "123456");

QJsonDocument document;
document.setObject(json);

QByteArray dataArray = document.toJson(QJsonDocument::Compact);

// 构造请求
QNetworkRequest request;
request.setHeader(QNetworkRequest::ContentTypeHeader, "application/json");
request.setUrl(url);

QNetworkAccessManager *manager = new QNetworkAccessManager(this);

// 连接信号槽
connect(manager, SIGNAL(finished(QNetworkReply *)), this, SLOT(replyFinished(QNetworkReply *)));

// 发送请求
manager->post(request, dataArray);
```

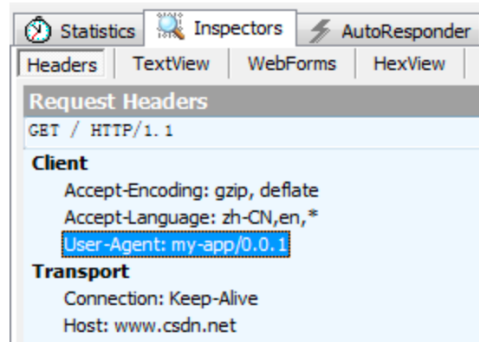
HTTP-定义请求头

如果你想为请求添加 HTTP 头部，只要简单地调用 `setHeader()` 就可以了。

例如，发送的请求时，使用的 User-Agent 是 `Mozilla/5.0`，为了方便以后追踪版本信息，可以将软件的版本信息写入到 User-Agent 中。

```
1 QNetworkRequest request;  
2 request.setHeader(QNetworkRequest::UserAgentHeader, "my-app/0.0.1");
```

User-Agent：包含发出请求的用户信息。



当然，除了 User-Agent 之外，`QNetworkRequest::KnownHeaders` 还包含其他请求头，它就是为 HTTP 头部而生的。根据 [RFC 2616](#)，HTTP 头部是大小写不敏感。

如果 `QNetworkRequest::KnownHeaders` 不满足需要，使用 `setRawHeader()`。

HTTP-响应内容

要获取响应的内容，可以调用 `readAll()`，由于上述的 POST 请求返回的数据为 Json 格式，将响应结果先转化为 Json，然后再对其解析：

```
1 void replyFinished(QNetworkReply *reply)
2 {
3     // 获取响应信息
4     QByteArray bytes = reply->readAll();
5
6     QJsonParseError jsonError;
7     QJsonDocument doucment = QJsonDocument::fromJson(bytes, &jasonError);
8     if (jsonError.error != QJsonParseError::NoError) {
9         qDebug() << QStringLiteral("解析Json失败");
10        return;
11    }
12
13    // 解析Json
14    if (doucment.isObject()) {
15        QJsonObject obj = doucment.object();
16        QJsonValue value;
17        if (obj.contains("data")) {
18            value = obj.take("data");
19            if (value.isString()) {
20                QString data = value.toString();
21                qDebug() << data;
22            }
23        }
24    }
25 }
```

[复制](#)

响应的内容可以是 HTML 页面，也可以是字符串、Json、XML等。最上面所发送的 GET 请求 获取的就是 CSDN 的首页 HTML。

响应状态码

我们可以检测响应状态码：

```
1 QVariant variant = pReplay->attribute(QNetworkRequest::HttpStatusCodeAttribute);  
2 if (variant.isValid())  
3     qDebug() << variant.toInt();  
4 // 200
```

HTTP 状态码请参考：

- [Status codes](#)
- [List of HTTP status codes](#)

最常见的是 200 OK，表示请求已成功，请求所希望的响应头或数据体将随此响应返回。

HTTP-响应头

可以看到包含很多，和上面一样，使用任意 `QNetworkRequest::KnownHeaders` 来访问这些响应头字段。例如，Content Type：

```
1 QVariant variant = pReply->header(QNetworkRequest::ContentTypeHeader);
2 if (variant.isValid())
3     qDebug() << variant.toString();
4 // "text/html; charset=utf-8"
```

如果 `QNetworkRequest::KnownHeaders` 不满足需要，可以使用 `rawHeader()`。例如，响应包含了登录后的TOKEN，位于原始消息头中：

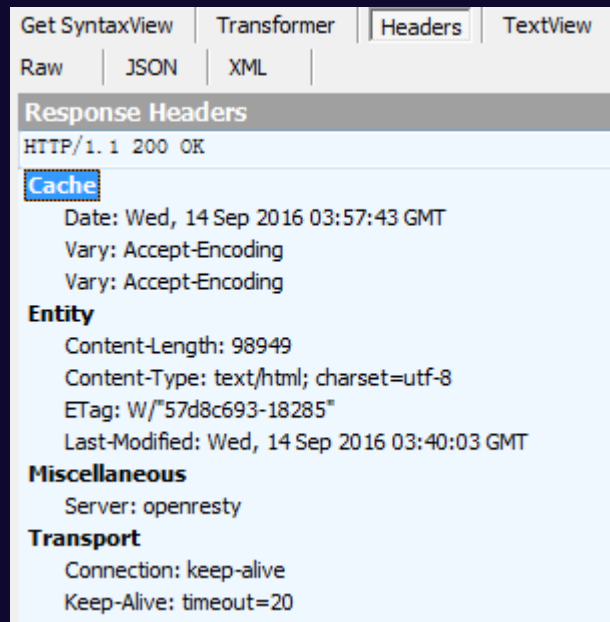
```
1 if (reply->hasRawHeader("TOKEN"))
2     QByteArray byte = reply->rawHeader("TOKEN");
```

复制

它还有一个特殊点，那就是服务器可以多次接受同一 header，每次都使用不同的值。Qt 会将它们合并，这样它们就可以用一个映射来表示出来，参见 [RFC 7230](#)：

A recipient MAY combine multiple header fields with the same field name into one “field-name: field-value” pair, without changing the semantics of the message, by appending each subsequent field value to the combined field value in order, separated by a comma.

接收者可以合并多个相同名称的 header 栏位，把它们合为一个 “field-name: field-value” 配对，将每个后续的栏位值依次追加到合并的栏位值中，用逗号隔开即可，这样做不会改变信息的语义。



Get SyntaxView | Transformer | Headers | TextView

Raw | JSON | XML

Response Headers

HTTP/1.1 200 OK

Cache

Date: Wed, 14 Sep 2016 03:57:43 GMT
Vary: Accept-Encoding
Vary: Accept-Encoding

Entity

Content-Length: 98949
Content-Type: text/html; charset=utf-8
ETag: W/"57d8c693-18285"
Last-Modified: Wed, 14 Sep 2016 03:40:03 GMT

Miscellaneous

Server: openresty

Transport

Connection: keep-alive
Keep-Alive: timeout=20

HTTP-请求错误处理

如果请求的处理过程中遇到错误（如：DNS 查询失败、拒绝连接等）时，则会产生一个 `QNetworkReply::NetworkError`。

例如，我们将 URL 改为这样：

```
1 QString baseUrl = "http://www.csdnQter.net/";
```

发送请求后，由于主机名无效，必然会发生错误，根据 `error()` 来判断：

```
1 QNetworkReply::NetworkError error = pReplay->error();
2 switch (error) {
3     case QNetworkReply::ConnectionRefusedError:
4         qDebug() << QStringLiteral("远程服务器拒绝连接");
5         break;
6     case QNetworkReply::HostNotFoundError:
7         qDebug() << QStringLiteral("远程主机名未找到（无效主机名）");
8         break;
9     case QNetworkReply::TooManyRedirectsError:
10        qDebug() << QStringLiteral("请求超过了设定的最大重定向次数");
11        break;
12    default:
13        qDebug() << QStringLiteral("未知错误");
14    }
15    // "远程主机名未找到（无效主机名）"
```

这时，会产生一个 `QNetworkReply::HostNotFoundError` 错误。

注意：`QNetworkReply::TooManyRedirectsError` 是 5.6 中引入的，默认限制是 50，可以使用 `QNetworkRequest::setMaxRedirectsAllowed()` 来改变。

02

FTP客户端

Strive to become a better version of yourself

引言

为了方便网络编程，Qt 提供了 Network 模块。该模块包含了许多类，例如：QFtp - 能够更加轻松使用 FTP 协议进行网络编程。

但是，从 Qt5.x 之后，Qt Network 发生了很大的变化，助手中关于此部分描述如下：

The QFtp and QUrlInfo classes are no longer exported. Use QNetworkAccessManager instead. Programs that require raw FTP or HTTP streams can use the Qt FTP and Qt HTTP compatibility add-on modules that provide the QFtp and QHttp classes as they existed in Qt 4.

意思是说：不再导出 QFtp 和 QUrlInfo 类，改用 QNetworkAccessManager。

Linux 下实现 FTP 服务的软件很多，最常见的有：vsftpd、Wu-ftp 和 Proftpd 等。

关于FTP服务端的搭建，属于配置的范畴，请大家自行网络搜索配置方法。

实现FTP上传/下载

FTPManager.h:

```
class FtpManager : public QObject
{
    Q_OBJECT
public:
    explicit FtpManager(QObject *parent = 0);
    // 设置地址和端口
    void setHostPort(const QString &host, int port = 21);
    // 设置登录 FTP 服务器的用户名和密码
    void setUserInfo(const QString &userName, const QString &password);
    // 上传文件
    void put(const QString &fileName, const QString &path);
    // 下载文件
    void get(const QString &path, const QString &fileName);
signals:
    void error(QNetworkReply::NetworkError);
    // 上传进度
    void uploadProgress(qint64 bytesSent, qint64 bytesTotal);
    // 下载进度
    void downloadProgress(qint64 bytesReceived, qint64 bytesTotal);
private slots:
    // 下载过程中写文件
    void finished();
private:
    QUrl m_pUrl;
    QFile m_file;
    QNetworkAccessManager m_manager;
};
```

实现FTP上传/下载

FTPManager.cpp:

```
FtpManager::FtpManager(QObject *parent)
    : QObject(parent)
{
    // 设置协议
    m_pUrl.setScheme("ftp");
}

// 设置地址和端口
void FtpManager::setHostPort(const QString &host, int port)
{
    m_pUrl.setHost(host);
    m_pUrl.setPort(port);
}

// 设置登录FTP服务器的用户名和密码
void FtpManager::setUserInfo(const QString &userName, const QString &password)
{
    m_pUrl.setUserName(userName);
    m_pUrl.setPassword(password);
}
```


实现FTP上传/下载

FTPManager.cpp:

// 上传文件

```
void FtpManager::put(const QString &fileName, const QString &path)
{
    QFile file(fileName);
    file.open(QIODevice::ReadOnly);

    QByteArray data = file.readAll();

    m_pUrl.setPath(path);

    QNetworkReply *pReply = m_manager.put(QNetworkRequest(m_pUrl), data);

    connect(pReply, SIGNAL(uploadProgress(qint64, qint64)), this, SIGNAL(uploadProgress(qint64, qint64)));

    connect(pReply, SIGNAL(error(QNetworkReply::NetworkError)), this, SIGNAL(error(QNetworkReply::NetworkError)));
}
```

// 下载文件

```
void FtpManager::get(const QString &path, const QString &fileName)
{
    QFileInfo info;
    info.setFile(fileName);
    m_file.setFileName(fileName);
    m_file.open(QIODevice::WriteOnly | QIODevice::Append);
    m_pUrl.setPath(path);

    QNetworkReply *pReply = m_manager.get(QNetworkRequest(m_pUrl));

    connect(pReply, SIGNAL(finished()), this, SLOT(finished()));

    connect(pReply, SIGNAL(downloadProgress(qint64, qint64)), this, SIGNAL(downloadProgress(qint64, qint64)));

    connect(pReply, SIGNAL(error(QNetworkReply::NetworkError)), this, SIGNAL(error(QNetworkReply::NetworkError)));
}
```

实现FTP上传/下载

注意：

下载过程中文件写入是在主线程中进行的，如果文件过大，频繁写入会造成主线程卡顿现象。要避免此种情况，请在工作线程中进行。

FTPManager.cpp:

```
// 下载过程中写文件
void FtpManager::finished()
{
    QNetworkReply *pReply = qobject_cast<QNetworkReply *>(sender());
    switch (pReply->error()) {
    case QNetworkReply::NoError : {
        m_file.write(pReply->readAll());
        m_file.flush();
    }
        break;
    default:
        break;
    }

    m_file.close();
    pReply->deleteLater();
}
```

使用FtpManager

主要代码如下：

其中，m_ftp 是类变量 FtpManager。

```
// 构建需要的控件
QPushButton *pUploadButton = new QPushButton(this);
QPushButton *pDownloadButton = new QPushButton(this);
m_pUploadBar = new QProgressBar(this);
m_pDownloadBar = new QProgressBar(this);

pUploadButton->setText(QString::fromLocal8Bit("上传"));
pDownloadButton->setText(QString::fromLocal8Bit("下载"));

// 接信号槽
connect(pUploadButton, SIGNAL(clicked(bool)), this, SLOT(upload()));
connect(pDownloadButton, SIGNAL(clicked(bool)), this, SLOT(download()));

// 设置 FTP 相关信息
m_ftp.setHostPort("192.168.***.***", 21);
m_ftp.setUser Info("wang", "123456");
```

使用FtpManager

// 上传文件

```
void MainWindow::upload()
{
    m_ftp.put("E:\\Qt.zip", "/home/wang/Qt.zip");
    connect(&m_ftp, SIGNAL(error(QNetworkReply::NetworkError)), this, SLOT(error(QNetworkReply::NetworkError)));
    connect(&m_ftp, SIGNAL(uploadProgress(qint64, qint64)), this, SLOT(uploadProgress(qint64, qint64)));
}
```

// 下载文件

```
void MainWindow::download()
{
    m_ftp.get("/home/wang/Qt.zip", "F:\\Qt.zip");
    connect(&m_ftp, SIGNAL(error(QNetworkReply::NetworkError)), this, SLOT(error(QNetworkReply::NetworkError)));
    connect(&m_ftp, SIGNAL(downloadProgress(qint64, qint64)), this, SLOT(downloadProgress(qint64, qint64)));
}
```

// 更新上传进度

```
void MainWindow::uploadProgress(qint64 bytesSent, qint64 bytesTotal)
{
    m_pUploadBar->setMaximum(bytesTotal);
    m_pUploadBar->setValue(bytesSent);
}
```

使用FtpManager

// 更新下载进度

```
void MainWindow::downloadProgress(qint64 bytesReceived, qint64 bytesTotal)
{
    m_pDownloadBar->setMaximum(bytesTotal);
    m_pDownloadBar->setValue(bytesReceived);
}
```

// 错误处理

```
void MainWindow::error(QNetworkReply::NetworkError error)
{
    switch (error) {
        case QNetworkReply::HostNotFoundError :
            qDebug() << QString::fromLocal8Bit("主机没有找到");
            break;
            // 其他错误处理
        default:
            break;
    }
}
```

关于验证:

- 1) 需要确保服务端FTP服务正常且目录具有下载和上传权限。
- 2) 通过MD5SUM工具, 可比较上传/下载前后的文件MD5值, 保证操作正确性。

```
[root@localhost wang]# md5sum Qt.zip
8d010354447515d55c65d733bbba2682  Qt.zip
```

03

网络接口信息获取

Strive to become a better version of yourself

QHostInfo 类为主机名查找提供了静态函数。

QHostInfo 利用操作系统提供的查询机制来查询与特定主机名相关联的主机的 IP 地址，或者与一个IP地址相关联的主机名。这个类提供了两个静态的便利函数：一个以异步方式工作，一旦找到主机就发射一个信号；另一个以阻塞方式工作，并且最终返回一个 QHostInfo 对象。

要使用异步方式查询主机的 IP 地址，调用 lookupHost() 即可，该函数包含 3 个参数，依次是：主机名/ IP 地址、接收的对象、接收的槽函数，并返回一个查询ID。以查询ID为参数，通过调用 abortHostLookup() 函数的来中止查询。

当获得查询结果后就会调用槽函数，查询结果被存储到 QHostInfo 对象中。可通过调用 addresses() 函数来获得主机的 IP 地址列表，同时可通过调用 hostName() 函数来获得查询的主机名。

查本机主机名称：

```
QString strLocalHostName = QHostInfo::localHostName();  
qDebug() << "Local Host Name:" << strLocalHostName;
```

QHostInfo

查询主机信息（异步方式）：使用 `lookupHost()`，实际的查询在一个单独的线程中完成，利用操作系统的方法来执行名称查找。

根据域名查询主机信息：

```
int nID = QHostInfo::lookupHost("qt-project.org", this, SLOT(lookedUp(QHostInfo)));  
void MainWindow::lookedUp(const QHostInfo &host)  
{  
    if (host.error() != QHostInfo::NoError) {  
        qDebug() << "Lookup failed:" << host.errorString();  
        return;  
    }  
    foreach (const QHostAddress &address, host.addresses()) {  
        // 输出IPV4、IPv6地址  
        if (address.protocol() == QAbstractSocket::IPv4Protocol)  
            qDebug() << "Found IPv4 address:" << address.toString();  
        else if (address.protocol() == QAbstractSocket::IPv6Protocol)  
            qDebug() << "Found IPv6 address:" << address.toString();  
        else  
            qDebug() << "Found other address:" << address.toString();  
    }  
}
```

输出结果：

```
Found IPv4 address: "5.254.113.102"  
Found IPv4 address: "178.32.152.214"
```


QHostInfo

根据IP查询主机信息：

```
int nID = QHostInfo::lookupHost("5.254.113.102", this, SLOT(lookedUp(QHostInfo)));
```

```
void MainWindow::lookedUp(const QHostInfo &host)
{
    if (host.error() != QHostInfo::NoError) {
        qDebug() << "Lookup failed:" << host.errorString();
        return;
    }

    qDebug() << "Found hostName:" << host.hostName();
}
```

输出结果：

```
Found hostName: "webredirects.cloudns.net"
```

QHostInfo

根据IP查询主机信息（阻塞方式）：如果要使用阻塞查找使用 `QHostInfo::fromName()` 函数。

这种情况下，名称查询的执行与调用者处于相同的线程中。

这对于非 GUI 应用程序或在一个单独的、非 GUI 线程中做名称查找是比较有用的（在 GUI 线程中调用这个函数可能会导致用户界面冻结）。

```
QHostInfo host = QHostInfo::fromName("5.254.113.102");  
if (host.error() != QHostInfo::NoError) {  
    qDebug() << "Lookup failed:" << host.errorString();  
    return;  
}
```

```
qDebug() << "Found hostName:" << host.hostName();
```

输出结果：

```
Found hostName: "webredirects.cloudns.net"
```

中止查询

lookupHost() 查询主机信息时，会返回一个查询 ID，以此 ID 为参数，通过调用 abortHostLookup() 来中止查询：

```
1 QHostInfo::abortHostLookup(nId);
```

错误处理

如果查询失败，error() 返回发生错误的类型，errorString() 返回一个能够读懂的查询错误描述。

枚举 QHostInfo::HostInfoError：

常量	值	描述
QHostInfo::NoError	0	查找成功
QHostInfo::HostNotFound	1	没有发现主机对应的IP地址
QHostInfo::UnknownError	2	未知错误

QHostAddress

QHostAddress类提供一个IP地址。这个类提供一种独立于平台和协议的方式来保存IPv4和IPv6地址。

QHostAddress通常与QTcpSocket、QTcpServer、QUdpSocket一起使用，来连接到主机或建立一个服务器。

可以通过setAddress()来设置一个主机地址，使用toIPv4Address()、toIPv6Address()或toString()来检索主机地址。你可以通过protocol()来检查协议类型。

注意：

- ✓ QHostAddress不做DNS查询，而QHostInfo是有必要的。
- ✓ 这个类还支持通用的预定义地址：Null、LocalHost、LocalHostIPv6、Broadcast和Any。

常用接口方法：

枚举 QHostAddress::SpecialAddress：		
常量	值	描述
QHostAddress::Null	0	空地址对象，相当于QHostAddress()。
QHostAddress::LocalHost	2	IPv4本地主机地址，相当于QHostAddress("127.0.0.1")。
QHostAddress::LocalHostIPv6	3	IPv6本地主机地址，相当于 QHostAddress("::1")。
QHostAddress::Broadcast	1	Pv4广播地址，相当于QHostAddress("255.255.255.255")。
QHostAddress::AnyIPv4	6	IPv4 any-address，相当于QHostAddress("0.0.0.0")。与该地址绑定的socket将只监听IPv4接口。
QHostAddress::AnyIPv6	5	IPv6 any-address，相当于QHostAddress("::")。与该地址绑定的socket将只监听IPv4接口。
QHostAddress::Any	4	双any-address栈，与该地址绑定的socket将侦听IPv4和IPv6接口。

QHostAddress

常用接口方法：

- ① 如果地址是IPv6的环回地址，或任何IPv4的环回地址，则返回true。

```
bool isLoopback() const
```

- ② 判断地址是否为空：如果主机地址为空（INADDR_ANY 或 in6addr_any），返回true。默认的构造函数创建一个空的地址，这个地址对于任何主机或接口是无效的。

```
bool isNull() const
```

- ③ 获取主机地址的网络层协议

```
QAbstractSocket::NetworkLayerProtocol protocol() const
```

- ④ 返回IPv4地址为一个数字。如果protocol()是IPv4Protocol，该值是有效的；如果是IPv6Protocol，并且IPv6地址是一个IPv4映射的地址，（RFC4291）。在这种情况下，ok将被设置为true；否则，它将被设置为false。

```
quint32 toIPv4Address() const
```

- ⑤ 返回的IPv6地址为Q_IPV6ADDR结构。该结构由16位无符号字符组成。

```
Q_IPV6ADDR toIPv6Address() const
```

- ⑥ 返回一个地址字符串

返回地址为一个字符串，例如，如果地址是IPv4地址127.0.0.1，返回的字符串为“127.0.0.1”。对于IPv6字符串格式将按照RFC5952建议。对于QHostAddress::Any，IPv4地址将返回（“0.0.0.0”）。

QHostAddress-简单应用1

获取所有主机地址：（QNetworkInterface类中提供了一个便利的静态函数allAddresses()，用于返回一个QHostAddress主机地址列表。）

```
QList<QHostAddress> list = QNetworkInterface::allAddresses();
foreach (QHostAddress address, list) {
    if (address.isNull())
        continue;

    qDebug() << "*****";

    QAbstractSocket::NetworkLayerProtocol nProtocol = address.protocol();

    QString strScopeId = address.scopeId();
    QString strAddress = address.toString();
    bool bLoopback = address.isLoopback();

    // 如果是IPv4
    if (nProtocol == QAbstractSocket::IPv4Protocol) {
        bool bOk = false;
        quint32 nIPv4 = address.toIPv4Address(&bOk);
        if (bOk)
            qDebug() << "IPv4 : " << nIPv4;
    }
    else if (nProtocol == QAbstractSocket::IPv6Protocol) {
        QStringList IPV6List("");
        Q_IPV6ADDR IPV6 = address.toIPv6Address();
        for (int i = 0; i < 16; ++i) {
            quint8 nC = IPV6[i];
            IPV6List << QString::number(nC);
        }
        qDebug() << "IPv6 : " << IPV6List.join(" ");
    }

    qDebug() << "Protocol : " << nProtocol;    qDebug() << "ScopeId : " << strScopeId;
    qDebug() << "Address : " << strAddress;    qDebug() << "IsLoopback : " << bLoopback;
}
```

QHostAddress-简单应用2

判断一个地址是否为私有IPV4地址：

10. 0. 0. 0 – 10. 255. 255. 255 172. 16. 0. 0 – 172. 31. 255. 255 192. 168. 0. 0 – 192. 168. 255. 255

```
bool isLocalIP(QHostAddress addr) {
    quint32 nIPv4 = addr.toIPv4Address();
    quint32 nMinRange1 = QHostAddress("10. 0. 0. 0").toIPv4Address();
    quint32 nMaxRange1 = QHostAddress("10. 255. 255. 255").toIPv4Address();
    quint32 nMinRange2 = QHostAddress("172. 16. 0. 0").toIPv4Address();
    quint32 nMaxRange2 = QHostAddress("172. 31. 255. 255").toIPv4Address();
    quint32 nMinRange3 = QHostAddress("192. 168. 0. 0").toIPv4Address();
    quint32 nMaxRange3 = QHostAddress("192. 168. 255. 255").toIPv4Address();
    if ((nIPv4 >= nMinRange1 && nIPv4 <= nMaxRange1)
        || (nIPv4 >= nMinRange2 && nIPv4 <= nMaxRange2)
        || (nIPv4 >= nMinRange3 && nIPv4 <= nMaxRange3)) {
        return true;
    } else {
        return false;
    }
}
```

QNetworkAddressEntry

QNetworkAddressEntry类由网络接口支持，存储了一个IP地址，子网掩码和广播地址。

这个类代表一个这样的组，每个网络接口可以包含零个或多个IP地址，进而可以关联到一个子网掩码和/或一个广播地址（取决于操作系统的支持）。

常用接口：

- ① 返回IPv4地址和子网掩码相关联的广播地址。对于IPv6地址来说，返回的总是空，因为广播的概念已被抛弃，为了系统支持多播。

```
QHostAddress broadcast() const
```

- ② 返回一个网络接口中存在的IPv4或IPv6地址。

```
QHostAddress ip() const
```

- ③ 返回与IP地址相关联的子网掩码。子网掩码是一个IP地址的形式表示，如255. 255. 0. 0。对于IPv6地址，前缀长度被转换成一个地址，其中设置为1的位数等于前缀长度。前缀长度为64位（最常见的值），子网掩码将被表示为一个地址为FFFF:FFFF:FFFF:FFFF::的QHostAddress。

```
QHostAddress netmask() const
```

- ④ 返回此IP地址的前缀长度。前缀长度和子网掩码中设置为1的位数相匹配。IPv4地址的值在0 - 32之间。IPv6地址的值在0 - 128之间，是表示数据的首选。

```
int prefixLength() const
```


QNetworkAddressEntry使用

获取本机所有网卡网络接口信息：

QNetworkInterface类中提供了一个便利的静态函数allInterfaces()，用于返回所有的网络接口。

通过遍历每一个网络接口QNetworkInterface，根据其addressEntries()函数，我们可以很容易的获取到所有的QNetworkAddressEntry，然后通过ip()、netmask()、broadcast()函数获取对应的IP地址、子网掩码以及广播地址。

```
QList<QNetworkInterface> list = QNetworkInterface::allInterfaces();
foreach (QNetworkInterface netInterface, list) {
    QList<QNetworkAddressEntry> entryList = netInterface.addressEntries();
    foreach(QNetworkAddressEntry entry, entryList) { // 遍历每一个IP地址
        qDebug() << "*****";
        qDebug() << "IP Address:" << entry.ip().toString(); // IP地址
        qDebug() << "Netmask:" << entry.netmask().toString(); // 子网掩码
        qDebug() << "Broadcast:" << entry.broadcast().toString(); // 广播地址
        qDebug() << "Prefix Length:" << entry.prefixLength(); // 前缀长度
    }
}
```

QNetworkInterface

QNetworkInterface类负责提供主机的IP地址和网络接口的列表。

QNetworkInterface表示了当前程序正在运行时与主机绑定的一个网络接口。每个网络接口可能包含0个或多个IP地址，每个IP地址都可选择性地与一个子网掩码和/或一个广播地址相关联。这样的列表可以通过addressEntries()方法获得。当子网掩码或者广播地址不必要时，可以使用allAddresses()便捷函数来仅仅获得IP地址。

QNetworkInterface使用hardwareAddress()方法获取接口的硬件地址。

注意：
不是所有的操作系统都支持这些所有的特性。只有IPv4地址可以保证在所有平台上都能被这个类列举出来，尤其是IPv6地址的列举目前只支持Windows XP及相关版本、Linux、MacOS和BSDs。

枚举 QNetworkInterface::InterfaceFlag
标识 QNetworkInterface::InterfaceFlags

指定网络接口相关的标识，可能的值：

常量	值	描述
QNetworkInterface::IsUp	0x1	网络接口处于活动状态
QNetworkInterface::IsRunning	0x2	网络接口已分配资源
QNetworkInterface::CanBroadcast	0x4	网络接口工作在广播模式
QNetworkInterface::IsLoopBack	0x8	网络接口是环回接口：也就是说，它是一个虚拟接口，其目的是主机本身
QNetworkInterface::IsPointToPoint	0x10	网络接口是一个点对点接口：也就是说，有一个，单一的其他地址可以直接由它到达。
QNetworkInterface::CanMulticast	0x20	网络接口支持多播

QNetworkInterface

常用接口：

1、这个便利函数返回主机上面发现的所有IP地址。相当于allInterfaces() 返回的所有对象调用addressEntries() 来获取QHostAddress对象列表，然后对每一个对象调用QHostAddress::ip()方法。

```
QList<QHostAddress> allAddresses() [static]
```

2、返回的主机上找到的所有的网络接口的列表。在失败情况下，它会返回一个空列表。

```
QList<QNetworkInterface> allInterfaces() [static]
```

3、返回IP地址列表，这个接口具备连同与其相关的网络掩码和广播地址。如果不需要子网掩码或广播地址的信息，可以调用allAddresses()函数来只获取IP地址。

```
QList<QNetworkAddressEntry> addressEntries() const
```

4、返回与此网络接口关联的标志。

```
InterfaceFlags flags() const
```

5、返回此接口的底层硬件地址。在以太网接口上，这将是表示MAC地址的字符串，用冒号分隔。其他接口类型可能有硬件地址的其他类型。不应该依赖于实现这个函数返回一个有效的MAC地址。

```
QString hardwareAddress() const
```

6、如果名称可确定，在Windows上，返回网络接口的人类可读的名称，例如：“本地连接”；如果不能，这个函数返回值与name()相同。用户可以在Windows控制面板中修改人类可读的名称，因此它可以在程序的执行过程中变化的名称。在Unix上，此函数目前返回值总是和name()相同，因为Unix系统不存储人类可读的名称的配置。

```
QString humanReadableName() const
```

7、如果此QNetworkInterface对象包含一个的有效网络接口，则返回true。

```
bool isValid() const
```

8、返回网络接口的名称。在Unix系统中，这是一个包含接口的类型和任选的序列号的字符串，例如：“eth0”、“lo”或者“pcn0”；在Windows中，这是一个内部ID，用户不能更改。

```
QString QNetworkInterface::name() const
```

QNetworkInterface使用

获取所有的IP地址：

通过调用便利的静态函数allAddresses()，可以返回一个QHostAddress地址列表（只能获取IP地址，没有子网掩码和广播地址的信息）。

```
QList<QHostAddress> list = QNetworkInterface::allAddresses();  
foreach (QHostAddress address, list) {  
    if (!address.isNull())  
        qDebug() << "Address : " << address.toString();  
}
```

```
Address : "fe80::f864:a962:7219:f98e%16"  
Address : "192.168.17.1"  
Address : "fe80::8169:691f:148e:d3cb%17"  
Address : "192.168.178.1"  
Address : "fe80::5996:27a3:83b5:2ae7%18"  
Address : "192.168.56.1"  
Address : "::1"  
Address : "127.0.0.1"
```

QNetworkInterface使用

获取网络接口列表：

通过调用便利的静态函数allInterfaces()，可以返回一个QNetworkInterface网络接口列表（通过QNetworkAddressEntry，可以获取IP地址、子网掩码和广播地址等信息）。

```
QList<QNetworkInterface> list = QNetworkInterface::allInterfaces();
foreach (QNetworkInterface netInterface, list) {

    if (!netInterface.isValid())
        continue;
    qDebug() << "*****";
    QNetworkInterface::InterfaceFlags flags = netInterface.flags();
    if (flags.testFlag(QNetworkInterface::IsRunning)
        && !flags.testFlag(QNetworkInterface::IsLoopBack)) { // 网络接口处于活动状态
        qDebug() << "Device : " << netInterface.name(); // 设备名
        qDebug() << "HardwareAddress : " << netInterface.hardwareAddress(); // 硬件地址
        qDebug() << "Human Readable Name : " << netInterface.humanReadableName(); // 人类可读的名字
    }
    // QList<QNetworkAddressEntry> entryList = netInterface.addressEntries();
    // foreach(QNetworkAddressEntry entry, entryList) { // 遍历每一个IP地址
        // qDebug() << "IP Address:" << entry.ip().toString(); // IP地址
        // qDebug() << "Netmask:" << entry.netmask().toString(); // 子网掩码
        // qDebug() << "Broadcast:" << entry.broadcast().toString(); // 广播地址
    //}
}
```

04

UDP/TCP编程

Strive to become a better version of yourself

UDP报文的发送和接收

QUdpSocket类可以用来发送和接收UDP数据报。UDP是一种不可靠，面向数据报的协议。一些应用层的协议使用UDP，因为它比TCP更加小巧轻便。

采用UDP，数据是以数据报的形式从一个主机发送到另一个主机的。这里并没有连接的概念，而且如果UDP包没有被成功投递，他不会向发送者报告任何错误。

我们将通过Weather Balloon和Weather Station这两个实例来看看Qt应用程序中如何使用UDP。

Weather Balloon应用模拟气象气球，每隔2秒就发送一个包含当天天气情况的UDP数据报，Weather Station应用程序接收这些数据报并将这些信息展示在屏幕上。

主要代码：

```
WeatherBalloon::WeatherBalloon(QWidget *parent)
    : QPushButton(tr("Quit"), parent)
{
    ...

    connect(&timer, SIGNAL(timeout()), this, SLOT(sendDatagram()));
    timer.start(2 * 1000);

    setWindowTitle(tr("Weather Balloon"));

    ...
}
```

在构造函数中，利用一个QTimer来实现每2秒钟调用一次sendDatagram()。

UDP报文的发送和接收

在sendDatagram()中，生成并发送一个包含当前日期、时间、温度、湿度和高度的数据报：

```
void WeatherBalloon::sendDatagram()
{
    QByteArray datagram;
    QDataStream out(&datagram, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_3);
    out <<QDateTime::currentDateTime() <<temperature() <<humidity() <<altitude();

    udpSocket.writeDatagram(datagram, QHostAddress::LocalHost, 5824);
}
```

WeatherStation的UDP服务端，需要监听同一个端口：

```
WeatherStation::WeatherStation(QWidget *parent):QDialog(parent)
{
    udpSocket.bind(5824);
    connect(&udpSocket, SIGNAL(readyRead()), this, SLOT(processPendingDatagrams));
    ...
}
```

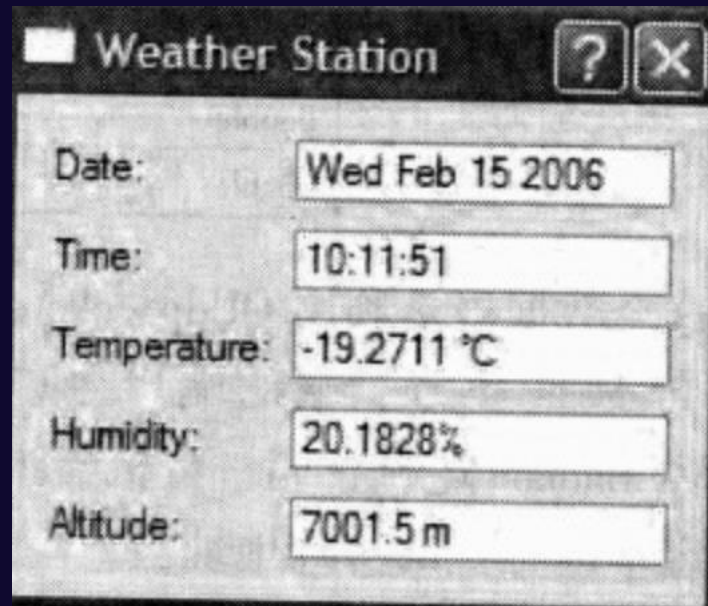

UDP报文的发送和接收

服务端收到数据并将收到的包含当前日期、时间、温度、湿度和高度的数据报，展示到界面中：

```
void WeatherStation::processPendingDatagrams()  
{  
    QByteArray datagram;  
    do{  
        datagram.resize(udpsocket.pendingDatagramSize());  
        udpSocket.readDatagram(datagram.data(), datagram.size());  
    }while (udpSocket.hasPendingDatagrams());  
    QDateTime dateTime; double temperature; double humidity; double altitude;  
    QDataStream in(&datagram, QIODevice::ReadOnly);  
    in.setVersion(QDataStream::Qt_4_3);  
    in>>dateTime>>temperature>>humidity>>altitude;  
    ...  
}
```

其中：

pendingDatagramSize() 函数返回第一个待处理的数据报大小，readDatagram把一个待处理数据报的内容复制到指定的缓冲区中，如果缓冲区太小会造成数据截断，并且前移至下一个待处理的数据报。



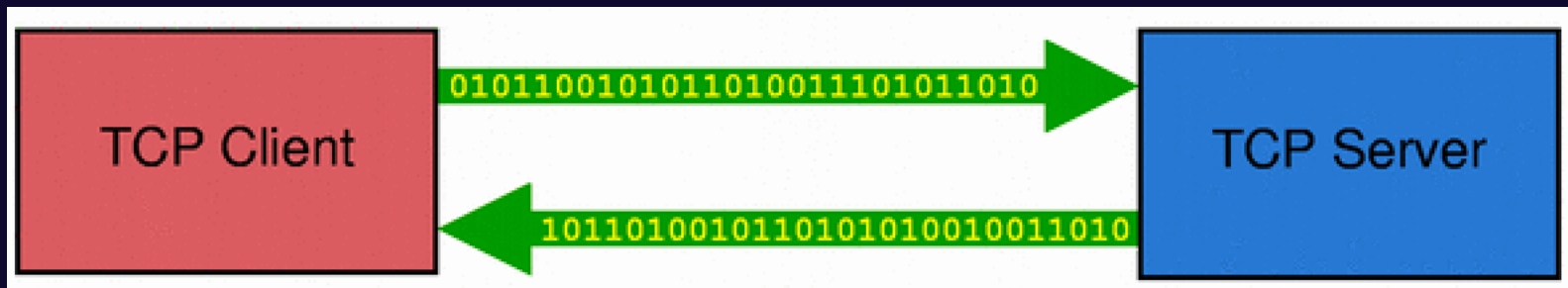
TCP客户端与服务端应用

TCP协议-传输控制协议是一个用于数据传输的低层网络协议，多个互联网协议（包括HTTP/FTP）都是基于TCP协议的。

TCP是一个面向数据流和连接的可靠传输协议。`QTcpSocket`类为TCP提供了一个接口，可以使用`QTcpSocket`来实现POP3、SMTP等标准的网络协议，也可以实现自定义的网络协议。

`QTcpSocket`与`QUdpSocket`传输的数据报不同，它适合用来传输连续的数据流，尤其适合于连续的数据传输。

TCP采用典型的C/S模型，一般分为客户端和服务端。



在任何数据传输之前，必须建立一个TCP连接到远程的主机和端口上。一旦建立连接，客户端的IP地址和端口可分别使用`QTcpSocket::peerAddress()`和`QTcpSocket::peerPort()`来获取。当客户端关闭连接后，数据传输也会立即停止。

注意：`QTcpSocket`是异步工作的，通过发射对应的信号来报告状态的改变和错误信息。它依靠事件循环来检测到来的数据，并且自动刷新输出数据。

我们可以通过`QTcpSocket::write()`来写入数据，使用`QTcpSocket::read()`来读取数据。

`QTcpSocket`本身继承自`QIODevice`，有二个独立的数据流来分别用于读取和写入。

从`QTcpSocket`中读取数据前，必须先调用`QTcpSocket::bytesAvailable()`函数来确保已经有足够的数据可用。

TCP客户端与服务端应用

在服务端应用中，可以使用QTcpServer类来处理到来的TCP连接。调用QTcpServer::listen()可以监听某一个端口，每当有客户端连接时均会触发QTcpServer::newConnection()信号，每当有客户端连接时都会发射该信号，在槽中，调用QTcpServer::nextPendingConnection()来接收这个连接，该方法返回一个QTcpSocket对象，可使用该对象与客户端通信。

注意：QTcpSocket大部分函数都是异步工作的，但仍然可以使用waitFor开头的函数，来实现一些阻塞行为，他们会挂起调用的线程，直到一个信号被发射。如：在调用connectToHost函数后，调用waitForConnected来阻塞线程，指导conneted信号被发射。

但，在一个GUI应用中，会占用GUI线程，导致用户界面的冻结。因此，**一般建议在非GUI线程中使用同步的套接字。**

和之前一样，还是要在工程文件中，添加QT+=network来使用相关的类。

TCP客户端与服务端应用示例-发送消息

Tcp客户端程序：

主要代码：

在构造函数中定义，socket声明及相关信号和槽的定义。

私有对象和变量定义：

```
QTcpSocket tcpSocket;//socket套接字  
quint16 blockSize;//用来存储数据的大小信息
```

构造函数中添加定义：

```
tcpSocket = new QTcpSocket(this);  
connect(tcpSocket, SIGNAL(readyRead()), this, SLOT(readMessage()));  
connect(tcpSocket, SIGNAL(error(QAbstractSocket::SocketError)), this, SLOT(displayError(QAbstractSocket::SocketError)));
```

连接按钮响应槽函数：

```
void Client::newConnect()  
{  
    //初始化数据大小信息为0  
    blockSize = 0;  
    //取消已有的连接  
    tcpSocket->abort();  
    tcpSocket->connectToHost(ui->hostLineEdit->text(), ui->portLineEdit->text().toInt());  
}
```

TCP客户端与服务端应用示例-发送消息

读取数据的槽函数定义：

在读取数据时，先读取了数据的大小信息，然后使用该大小信息来判断是否已经读取到了所有的数据。

```
void Client::readMessage()  
{  
    QDataStream in(tcpSocket);  
    //设置数据流版本，这里要和服务器相同  
    in.setVersion(QDataStream::Qt_5_12);  
    //如果是刚开始接收数据  
    if(blockSize == 0){  
        //判断接收的数据是否大于两字节，也就是文件的大小信息所占的空间  
        //如果是则保存到blockSize变量中，否则直接返回，继续接收数据  
        if(tcpSocket->bytesAvailable() < (int)sizeof(quint16)) return;  
        in>>blockSize;  
    }  
    //如果没有得到全部的数据，则返回继续接收数据  
    if(tcpSocket->bytesAvailable() < blockSize) return;  
    //将接收到的数据存放到变量中  
    in>>message;  
    //显示接收到数据  
    qDebug() <<message;  
    ui->messageLabel->setText(message);  
}
```

TCP客户端与服务端应用示例-发送消息

TCP服务端程序：调用listen()函数来监听客户端连接，这里监听了本地主机6666的端口，这样可以实现客户端和服务端在同一个机器上通信。

主要代码：

私有对象定义：

```
QTcpServer *tcpServer;//服务端对象
```

私有槽函数声明：

```
private slots:
```

```
    void sendMessage();
```

在构造函数中添加如下代码：

```
    tcpServer = new QTcpServer(this);
```

```
    //使用了IPv4的本地主机地址
```

```
    if(!tcpServer->listen(QHostAddress::LocalHost, 6666))
```

```
{
```

```
        qDebug() << tcpServer->errorString();
```

```
        close();
```

```
}
```

```
    connect(tcpServer, SIGNAL(newConnection()), this, SLOT(sendMessage()));
```

TCP客户端与服务端应用示例-发送消息

发送信息槽函数主要代码：

```
void Server::sendMessage()  
{  
    //用于暂存要发送的数据  
    QByteArray block;  
    QDataStream out(&block, QIODevice::WriteOnly);  
  
    //设置数据流的版本，客户端和服务端使用的版本要相同  
    out.setVersion(QDataStream::Qt_5_12);  
    out<<(quint16)0; out<<tr("Hello TcpSocket!");  
    out.device()->seek(0); out<<(quint16)(block.size()-sizeof(quint16));  
  
    //获取已经建立的连接的套接字  
    QTcpSocket *clientConnection = tcpServer->nextPendingConnection();  
    connect(clientConnection, SIGNAL(disconnected()), clientConnection, SLOT(deleteLater()));  
    clientConnection->write(block);  
    clientConnection->disconnectFromHost();  
    //发送数据成功后，显示提示  
    ui->label->setText("Send message successful!");  
}
```

TCP客户端与服务端应用示例-文件传输

实现一个示例，完成客户端向服务端发送文件，并能显示进度。

客户端主要代码如下：

私有对象和变量定义：

```
QTcpSocket *tcpClient;  
QFile *localFile;//要发送的文件  
qint64 totalBytes;//发送数据的总大小  
qint64 bytesWritten;//已经发送数据大小  
qint64 bytesToWrite;//剩余数据大小  
qint64 payloadSize;//每次发送数据的大小  
QString fileName;//保存文件路径  
QByteArray outBlock;//数据缓冲区，即存放每次要发送的数据块
```

私有槽声明：

```
private slots:  
    void openFile();  
    void send();  
    void startTransfer();  
    void updateClientProgress(qint64);  
    void displayError(QAbstractSocket::SocketError);
```


TCP客户端与服务端应用示例-文件传输

客户端主要代码如下：

在构造函数中，初始化变量。

```
payloadSize = 64*1024; //64KB
```

```
totalBytes = 0;
```

```
bytesWritten = 0;
```

```
bytesToWrite = 0;
```

```
tcpClient = new QTcpSocket(this);
```

```
//当连接服务器成功时，发出connected()信号，开始传送文件
```

```
connect(tcpClient, SIGNAL(connected), this, SLOT(startTransfer()));
```

```
connect(tcpClient, SIGNAL(bytesWritten(qint64)), this, SLOT(updateClientProgress(qint64)));
```

```
connect(tcpClient, SIGNAL(error(QAbstractSocket::SocketError)), this, SLOT(displayError(QAbstractSocket::SocketError)));
```

```
ui->sendButton->setEnabled(false);
```

调用QFileDialog来打开一个本地文件，使用connectToHost来连接到服务器。

TCP客户端与服务端应用示例-文件传输

客户端主要代码如下：

在实际的文件传输前，需要将整个传输的数据大小、文件名大小、文件名等信息放在数据开头进行传输，形成文件头部结构进行传输。

```
void Client::startTransfer ()
{
    QFile localFile (filename);
    if (!localFile->open (QFile::ReadOnly))
    {
        qDebug () << "Client: open file error!";
        return;
    }
    //获取文件大小
    totalBytes = localFile->size ();
    QDataStream sendOut (&outBlock, QIODevice::WriteOnly);
    sendOut.setVersion (QDataStream::Qt_5_12);
    QString currentFileName = fileName.right (fileName.size () - fileName.lastIndexOf ("/") - 1);
    //保留总大小信息空间、文件名大小信息空间，然后输入文件名
    sendOut << qint64 (0) << qint64 (0) << currentFileName;
    //这里的总大小是总大小信息、文件名大小信息、文件名和实际文件消息总和
    totalBytes += outBlock.size ();
    sendOut.device ()->seek (0);
    //返回outBlock的开始，用实际的大小信息代替二个初始qint64 (0)空间
    sendOut << totalBytes << qint64 ((outBlock.size () - sizeof (qint64) * 2));
    //发送完文件头结构后剩余数据大小
    bytesToWrite = totalBytes - tcpClient->write (outBlock);
    ui->clientStatusLabel->setText (tr ("已连接"));
    outBlock.resize (0);
}
```

TCP客户端与服务端应用示例-文件传输

客户端主要代码如下：

发送完文件头结构以后，发送实际文件内容，在更新进度的槽函数中实现。

```
void Client::updateClientProgress(qint64 numBytes)
{
    //已经发送数据的大小
    bytesWritten += (int)numBytes;
    //如果已经发送了数据
    if(bytesToWrite > 0) {
        //每次发送payloadSize 大小数据，这里设置的大小是64KB，如果剩余数据不足64KB就发送剩余数据大小
        outBlock = localFile->read(qMin(bytesToWrite, payloadSize));
        //发送完一次数据后还剩余数据的大小
        bytesToWrite -= (int)tcpClient->write(outBlock);
        //清空发送缓冲区
        outBlock.resize(0);
    }else{ //如果没有发送任何数据，则关闭文件
        localFile->close();
    }
    //更新进度条
    ui->clientProgressBar->setMaximum();
    ui->clientProgressBar->setValue(bytesWritten);
    //如果发送完毕
    if(bytesWritten == totalBytes) {
        ui->clientStatusLabel->setText(tr("传送文件%1成功").arg(fileName));
        localFile->close();
        tcpClient->close();
    }
}
```

TCP客户端与服务端应用示例-文件传输

服务端主要代码如下：

私有变量和对象定义：

```
QTcpServer tcpServer;  
QTcpSocket *tcpServerConnection;  
qint64 totalBytes; //存放总大小信息  
qint64 bytesReceived; //已收到数据的大小  
qint64 fileNameSize; //文件名的大小信息  
QString fileName; //存放文件名  
QFile *localFile; //本地文件  
QByteArray inBlock; //数据缓冲区
```

再添加几个私有槽声明：

```
void start();  
void acceptConnection();  
void updateServerProgress();  
void displayError(QAbstractSocket::SocketError socketError);
```

构造函数中添加信号和槽关系：

```
connect(&tcpServer, SIGNAL(newConnection), this, SLOT(acceptConnection));
```

TCP客户端与服务端应用示例-文件传输

服务端主要代码如下：

```
void Server::updateServerProgress()
{
    QDataStream in(tcpServerConnection);
    in.setVersion(QDataStream::Qt_5_12);
    //如果接收到的数据小于16个字节，保存到来的文件头结构
    if(bytesReceived <= sizeof(qint64)*2) {
        if((tcpServerConnection->bytesAvailable() >= sizeof(qint64)*2) && (fileNameSize == 0))
        {
            //接收数据总大小信息和文件名大小信息
            in>>totalBytes>>fileNameSize;
            bytesReceived += sizeof(qint64) * 2;
        }
        if((tcpServerConnection->bytesAvailable() >= fileNameSize)
            && (fileNameSize != 0)) {
            //接收文件名,并建立文件
            in>>fileName;
            ui->serverStatusLabel->setText(tr("接收文件 %1 ...")
                .arg(fileName));
            bytesReceived += fileNameSize;
            localFile = new QFile(fileName);
            if (! localFile->open(QFile::WriteOnly)) {
                qDebug() << " server: open file error!" ;
                return ;
            }
        }else {
            return;
        }
    }
    .....
}
```

TCP客户端与服务端应用示例-文件传输

服务端主要代码如下：

```
void Server::updateServerProgress()
{
    .....

    //如果接收的数据小于总数据, 那么写入文件
    if (bytesReceived < totalBytes) {
        bytesReceived += tcpServerConnection->bytesAvailable();
        inBlock = tcpServerConnection->readAll();
        localFile->write(inBlock);
        inBlock.resize(0);
    }

    ui->serverProgressBar->setMaximum(totalBytes);
    ui->serverProgressBar->setValue(bytesReceived);

    //接收数据完成时
    if (bytesReceived == totalBytes) {
        tcpServerConnection->close();
        localFile->close();
        ui->startButton->setEnabled(true);
        ui->serverStatusLabel->setText(tr("接收文件%1成功!").arg(fileName));
    }
}
```

05

课后练习

Qt exercise assignment

课后练习

序号	作业内容
1	实现一个HTTP客户端，要求实现HTTP上传/下载,上传下载进度展示，实现一个POST请求，按照xml/JSON格式进行数据的展示/更新。
2	实现一个FTP客户端，可遍历和展示整个主目录，能对文件进行上传、下载、删除等常见操作，上传/下载过程有进度展示。
3	创建一个应用，可获取本机所有网卡详细信息并展示，通过界面操作可实现指定修改某一个网卡的ip、添加ip等操作。
4	实现一个UDP的客户端和服务端，实现一个简易的聊天应用、实现点对点的文件传输。
5	实现一个TCP的客户端和服务端，服务端要求支持多客户端连接，可接收不同客户端信息并展示，客户端要求支持手动或定时发送文本信息、十六进制信息。



感谢！Thank You.

2021合肥工业大学软件学院实训项目 | 2021

时代红

2021-07-05

