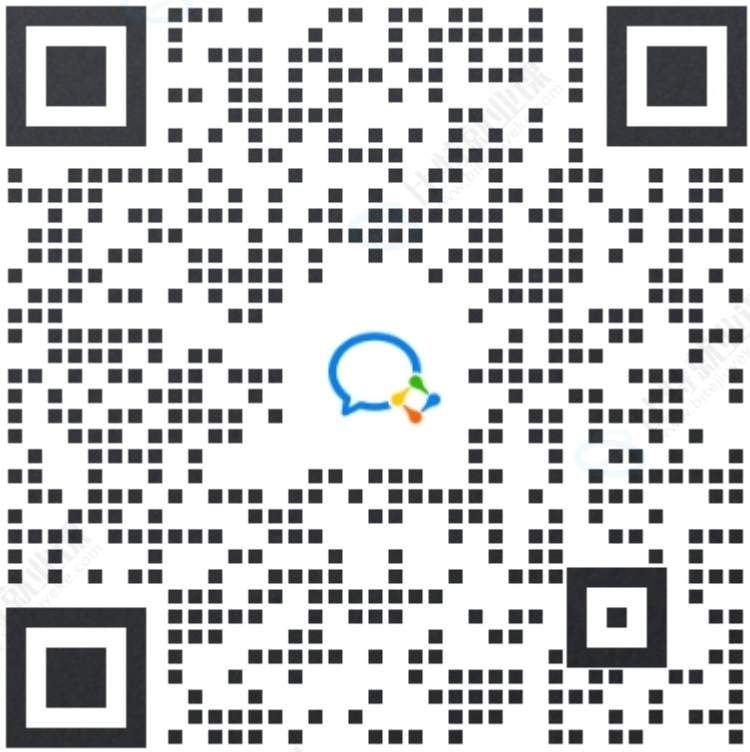
7. 网络通信

版权说明



代码 & 板书链接

> https://gitee.com/bitedu-cpp-team/cpp-microservice-videoplayer/tree/master/client

1. 网络通信

将来播放平台上几乎所有的数据，比如视频信息、用户信息、封面图片、视频文件等都在服务器上，客户端需要通过网络与服务器进行交互。在客户端和服务端进行交互的过程中，为了解决环境上的差异，减少冗余数据的传输，需要将通过网络传输的数据进行序列化和反序列化。

客户端和服务器的数据交互通过HTTP协议完成，数据的序列化和反序列化采用JSON来进行处理。

2. JSON

2.1 JSON介绍

JSON(JavaScript Object Notation, JavaScript对象表示)是一种轻量级的数据交换格式，它以文本形式表示由键值对组成的数据对象，用于数据的序列化和传输。由于其简单易读、易于解析和生成，JSON 已经成为现代 Web 开发和分布式系统中最常用的数据交换格式之一。

语法规则：

- JSON对象由键值对组成，所有键值对有一个大括号 `{}` 包围
- 键值对以 `"key" : value` 的形式出现，键和值之间以冒号 `:` 分割
- 多个键值对之间用逗号 `,` 分割，最后一个键值对之后不需要加逗号
- 所有键都必须是字符串，并且必须用双引号 `"` 包围
- JSON对象支持嵌套

数据类型

JSON的数据类型包括对象、数组、字符串、数字、布尔值(`true` / `false`)或 `null`

- 对象：使用花括号 `{}` 括起来的表示一个对象
- 使用中括号 `[]` 括起来的表示一个数组
- 使用常规双引号 `""` 括起来的表示一个字符串
- 包括整形和浮点型，直接使用

代码块

```
// 一个学生对象的JSon表示
{
    "name" : "张三",
    "age" : 18,
    "gender" : "男",
    "hobby" : ["篮球", "足球" , "编程"],
    "score" : {
        "C语言" : 82,
        "C++" : 76,
        "数据结构" : 80
```

```
    }
}
```

2.2 Qt中对JSON的支持

Qt 提供了一套完整的JSON支持，包括以下类：

- QJsonObject：封装了一个JSON对象，即键值对的集合
- QJsonArray：封装了一个JSON数组，可以存储一系列的QJsonValue对象，即值的有序集合
- QJsonValue：封装了一个JSON值，QJsonObject中key是字符串，value是QJsonValue对象
- QJsonDocument：用于解析和生产JSON文本，即对QJsonObject完成序列化和反序列化。

2.3 Qt中JSON序列化示例

代码块

```
QByteArray serialize()
{
    QJsonObject student;
    student.insert("name", "张三");
    student.insert("age", "18");
    student["gender"] = "男";

    // 创建一个JSON数组
    QJsonArray hobby;
    hobby.append("篮球");
    hobby.append("足球");
    hobby.append("编程");
    // 将JSON数组添加到JSON对象中
    student.insert("hobby", hobby);

    // 创建一个成绩JSON对象
    QJsonObject score;
    score.insert("C语言", 82);
    score.insert("C++", 76);
    score.insert("数据结构", 80);
    // 将score的JSON对象添加到student中
    student.insert("score", score);

    LOG()<<"Json中kv个数："<<student.count();
    LOG()<<student;

```

```
    // 对JSON对象进行序列化
    QJsonDocument jsonDoc(student);
    return jsonDoc.toJson();
}
```

2.4 Qt中JSON反序列化

代码块

```
// 对JSON字符串进行反序列化
QJsonObject Deserialize(QByteArray array){
    QJsonDocument jsonDoc = QJsonDocument::fromJson(array);
    QJsonObject student = jsonDoc.object();

    // 解析JSON对象
    LOG()<<"name: "<<student["name"].toString();
    LOG()<<"age: "<<student["age"].toInt();
    LOG()<<"gender: "<<student["gender"].toString();

    // 解析爱好，爱好是JSON数组
    LOG()<<"hobby: ";
    QJsonArray hobby = student["hobby"].toArray();
    for(int i = 0; i < hobby.count(); ++i){
        LOG()<<hobby[i].toString();
    }

    // 解析成绩，成绩是JSON对象
    LOG()<<"score:";
    QJsonObject score = student["score"].toObject();
    LOG()<<"C语言: "<<score["C语言"].toInt();
    LOG()<<"C++: "<<score["C++"].toInt();
    LOG()<<"数据结构: "<<score["数据结构"].toInt();
    LOG()<<"==============================";
    LOG()<<student;
    return student;
}

```

3. 搭建HTTP客户端

3.1 相关类介绍

在Qt中，想要使用网络进行通信，首先需要导入network模块：

代码块

```
CMake构建器：
find_package(Qt6 REQUIRED COMPONENTS Network)
target_link_libraries(mytarget PRIVATE Qt6::Network)

qmake构建器：
 qmake: QT += network
```

`QNetworkAccessManager` 是 Qt 网络模块中的一个核心类，用于发送请求、处理响应、管理会话等，使得开发者可以方便地发送 HTTP 请求、处理响应等。

代码块

```
// 发送get请求
QNetworkReply *get(const QNetworkRequest &request);

// 发送post请求
QNetworkReply *post(const QNetworkRequest &request, const QByteArray &data);
// ...
// QT将HTTP协议常见的方法都封装了，这里不在一一列举，需要用到时请翻阅Qt的官方文档。

 // 请求响应成功后，触发该信号
 void finished(QNetworkReply *reply)
```

注意：请求方法默认都是异步的，执行完成之后会立即返回，不会阻塞等待响应，响应到达时以finished信号通知

`QNetworkRequest` 是 Qt 网络模块中的一个核心类，用于封装网络请求的所有关键信息，包括请求的 URL、HTTP 头部信息、请求体等。

代码块

```
// 创建一个没有指定URL的请求对象，之后可以通过setUrl()设置URL
QNetworkRequest();

// 使用指定的URL构造请求
QNetworkRequest(const QUrl &url);

// 设置请求的URL
void setUrl(const QUrl &url);

// 设置已知的HTTP头部，比如 content-type
void setHeader(QNetworkRequest::KnownHeaders header, const QVariant &value);
```

利用 `QNetworkRequest` 可以完成请求的设置，之后就可以将请求适合的方式发送给服务器。

3.2 客户端搭建示例

代码块

```
//////////////////////////// CMakeLists.txt
//////////////////////////////////////
// ...
find_package(QT NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets Network)
// ...
set(MPV_DLL ${CMAKE_CURRENT_SOURCE_DIR}/mpv/dll/libmpv-2.dll)
target_link_libraries(MyBitPlayer PRIVATE Qt${QT_VERSION_MAJOR}::Widgets
                      ${MPV_DLL} Qt6::Network)
// ...

//////////////////////////// netclient.h
////////////////////////////////////
#include <QObject>
#include <QtNetwork/QNetworkAccessManager>

namespace network{
class NetClient : public QObject
{
    Q_OBJECT
public:
    NetClient(QObject *parent = nullptr);
    // 发送hello请求
    void hello();
private:
    const QString HTTP_URL = "http://127.0.0.1:8000";  // 默认本地回环，8000端口

    QNetworkAccessManager httpClient;
};

}  // end network
```

代码块

```
//////////////////////////// netclient.cpp
////////////////////////////////////
#include "netclient.h"
```

```cpp
#include <QtNetwork/QNetworkRequest>
#include <QtNetwork/QNetworkReply>
#include <QJsonObject>
#include <QJsonDocument>

#include "util.h"
namespace network {
NetClient::NetClient(QObject *parent)
    : QObject{parent}
{}

/*
* 请求URL post /hello
* {
*     // 请求中暂时不做什么事情
* }
*/
void NetClient::hello()
{
    // 1. 构造请求体 body
    QJsonObject reqBody;

    // 2. 发送请求
    QNetworkRequest httpReq;
    httpReq.setUrl(HTTP_URL + "/hello");
   httpReq.setHeader(QNetworkRequest::ContentTypeHeader, "application/json;
charset=utf8");

    QJsonDocument document(reqBody);
    QNetworkReply* httpResp = httpClient.post(httpReq, document.toJson());

    // 3. 异步处理响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // 判定Http层面是否出错
        if(httpResp->error() != QNetworkReply::NoError){
            LOG()<<httpResp->errorString();
            httpResp->deleteLater();
            return;
        }

        // 获取到响应的body
        QByteArray respBody = httpResp->readAll();

        // 针对body反序列化
        QJsonDocument jsonDoc = QJsonDocument::fromJson(respBody);
        if(jsonDoc.isNull()){
            LOG()<<"解析 JSON 文件失败! JSON 文件格式有错误!";
```

```
            httpResp->deleteLater();
            return;
        }

        // 判定业务上的逻辑是否正确
        QJsonObject respObj = jsonDoc.object();
        if(0 != respObj["errorCode"].toInt()){
            LOG()<<respObj["errorMsg"].toString();
            httpResp->deleteLater();
            return;
        }

        // 解析响应数据
        QJsonObject resoBody = respObj["data"].toObject();
        LOG()<<resoBody["hello"].toString();
        httpResp->deleteLater();
    });
}
}
```

在实际的生产环境中，为了区分不同请求，以及在出现问题时能够快速定位和排查，可以给每个HTTP请求唯一标识符Request ID。该Request ID可以在接口中约定，客户端可以利用UUID生成Request ID。

UUID（Universally Unique Identifier，**通用唯一标识符**）是一种软件构建中常用的标准化标识符，用于唯一标识信息的一个128位标识符，非常适合用于标识数据库记录、网络请求、日志条目等。

UUID通常以36个字符的字符串形式表示，`xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`，x表示十六进制，字符串由5组数字组成，分别包含8、4、4、4、12个十六进制数字，每组之间用字符`-`连接。

在Qt中可以使用QUuid类生成UUDI。

代码块

```
////////////////////////////////// netclient.h
//////////////////////////////////
class NetClient : public QObject
{
    // ...
    void hello();
private:
    // 生成请求id
    static QString makeRequestId();
    // ...
};

```

```cpp
////////////////////////////////// netclient.cpp
////////////////////////////////////
QString NetClient::makeRequestId()
{
    // 基本要求，确保每个请求的 id 都是不重复(唯一的)
    // 通过 UUID 来实现上述效果.
    // 请求ID以R开头，保留UUDI中的从第24未开始的后12位
    return "R" + QUuid::createUuid().toString().sliced(24, 12);
}

/*
* 请求URL post /hello
* {
*     requestId : "string"
*     // 请求中暂时不做什么事情
* }
*/
void NetClient::hello()
{
    // 1. 构造请求体 body
    QJsonObject reqBody;
        // 设置请求id
    reqBody["requestId"] = makeRequestId();

    // 2. 发送请求
    // ...
    });
}
```

4. 搭建MockServer

MockServer是能够提供Mock功能的服务器，通过模拟真实的服务器，提供对来自客户端请求的真实响应。它允许用户创建模拟HTTP服务以模拟后端服务的响应，从而在开发和测试过程中，无需依赖实际的后端服务，也能够进行接口测试和功能验证。实际就是搭建一个HTTP服务器，构造模拟数据对客户端接口进行测试。

4.1 相关类介绍

QHttpServer用来创建和管理HTTP服务器，该类提供简单易用的接口，通过route能够方便添加路由规则，为不同的URL路径指定不同的回调函数来处理请求，使用异步IO模型，能高效处理大量的并发连接。

代码块

```cpp
// 功能：创建QHttpServer实例
QHttpServer(QObject *parent = nullptr);

// 功能：绑定ip地址和端口号，将套接字设置为监听套接字
// 参数：
//       address: 指定服务器监听的IP地址。QHostAddress::Any表示监听所有可用的网络地址，
//                在 IPv4 中，它通常对应于 0.0.0.0，而在 IPv6 中，它对应于 ::
//       port:    指定服务器监听的端口号
// 返回值：如果服务器成功启动并开始监听指定的地址和端口时，返回端口号，否则返回0
quint16 listen(const QHostAddress &address = QHostAddress::Any, quint16 port = 0)

// 功能：定义HTTP请求的路由规则，将请求路径和方法与处理函数绑定，当服务器收到匹配的请求时，
//       会调用对应的处理函数
// Args: 可变参数包，最后一个参数是可调用对象(仿函数、lambda表达式、函数指针)，其余参数
//       用于创建新路由规则
// 返回值：新的路由规则创建成功返回true，否则返回false
template <typename Rule = QHttpServerRouterRule, typename... Args>
bool QHttpServer::route(Args &&... args)
// server.route("/hello", [] (QHttpServerRequest &request) { return ""; });

// 功能：构造http响应对象
// 参数：
//       data : 响应的正文内容，序列化之后的结果
//       status: http的状态码，默认是响应成功。200是OK  404是 Not Found...
QHttpServerResponse(const QByteArray &data,
                    const QHttpServerResponse::StatusCode status = StatusCode::Ok)
// 使用QHttpServer搭建Http服务器的步骤：
// 1. 创建QHttpServer对象
// 2. 调用route创建路由规则
// 3. 调用listen绑定ip地址和端口，并将套接字设置为监听套接字，然后等待客户端链接
// 4. 实现响应方法
```

4.2 服务端搭建示例

代码块

```cpp
////////////////////////////// CMakeLists.txt
//////////////////////////////////
//...
find_package(QT NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets HttpServer)
// ...
```

```
target_link_libraries(MyBitPlayerMockServer PRIVATE
                      Qt${QT_VERSION_MAJOR}::Widgets Qt6::HttpServer)
//...
////////////////////////////// util.h ////////////////////////////////
#include <QString>
#include <QDebug>
#include <QFileInfo>

static inline QString getFileName(const QString& path) {
    QFileInfo fileInfo(path);
    return fileInfo.fileName();
}

// 封装一个 "宏" 作为打印日志的方式.
#define TAG QString("[%1:%2]").arg(getFileName(__FILE__),
QString::number(__LINE__))
// #define TAG "[" << __LINE__ << "]"

// qDebug 打印字符串的时候, 就会自动加上 " "
#define LOG() qDebug().noquote() << TAG

////////////////////////////// server.h ////////////////////////////////
#include <QObject>
#include <QHttpServer>
#include <QHttpServerResponse>

class HttpServer : QObject
{
    Q_OBJECT
private:
     HttpServer();
     static HttpServer* instance;

private:
    QHttpServer httpServer;

public:
    bool init();
    static HttpServer* getInstance();

private:
    // 测试方法
    QHttpServerResponse hello(const QHttpServerRequest& req);
};
```

代码块

```cpp
////////////////////////////// server.cpp ///////////////////////////////////
#include "server.h"
#include "util.h"

#include <QJsonDocument>
#include <QJsonObject>

HttpServer* HttpServer::instance = nullptr;
HttpServer* HttpServer::getInstance()
{
    if(nullptr == instance){
        instance = new HttpServer();
    }

    return instance;
}

HttpServer::HttpServer()
{}

bool HttpServer::init()
{
    // 绑定ip地址和端口号
    int ret = httpServer.listen(QHostAddress::Any, 8000);

    // 配置路由
    httpServer.route("/hello", [=](const QHttpServerRequest& req){
        return this->hello(req);
    });

    return 8000 == ret;
}

QHttpServerResponse HttpServer::hello(const QHttpServerRequest &req)
{
    QJsonDocument docReq = QJsonDocument::fromJson(req.body());
    const QJsonObject& jsonReq = docReq.object();
    LOG() << "[hello] 收到 hello 请求, requestId = "
<<jsonReq["requestId"].toString();

    // 构造响应正文
    QJsonObject jsonBody;
    jsonBody["hello"] = "world";

    // 构造响应其他部分
    QJsonObject jsonResp;
    jsonResp["errmsg"] = "";
```

```
47      jsonResp["data"] = jsonBody;
48      QJsonDocument docResp;
49      docResp.setObject(jsonResp);
50
51      // 构造 HTTP 响应
52      QHttpServerResponse httpResp(docResp.toJson(),
    QHttpServerResponse::StatusCode::Ok);
53      httpResp.setHeader("Content-Type", "application/json; charset=utf-8");
54      return httpResp;
55  }
56  ////////////////////////////// main.cpp ////////////////////////////////
57  #include "mainwindow.h"
58  #include <QApplication>
59
60  #include "server.h"
61  #include "util.h"
62
63  int main(int argc, char *argv[])
64  {
65      QApplication a(argc, argv);
66
67      // 启动Http服务器
68      HttpServer* httpServer = HttpServer::getInstance();
69      if(!httpServer->init()){
70          LOG()<<"HTTP服务器启动失败!";
71      }
72
73      LOG()<<"HTTP服务器启动成功!";
74
75      MainWindow w;
76      w.show();
77      return a.exec();
78  }
79
```

5. 请求和请求响应封装

如果再实现一个ping请求，服务端收到ping请求之后返回pang。

代码块

```
1   ////////////////////////////// netclient.h
    ////////////////////////////////////////
2   class NetClient : public QObject
3   {
4       // ...
5       // hello请求
```

```
6       void hello();
7
8       // ping请求
9       void ping();
10      // ...
11  };
12  ////////////////////////////// netclient.cpp
    ////////////////////////////////////
13  void NetClient::ping()
14  {
15      // 1. 构造请求体 body
16      QJsonObject reqBody;
17
18      // 2. 发送请求
19      QNetworkRequest httpReq;
20      httpReq.setUrl(HTTP_URL + "/ping");
21     httpReq.setHeader(QNetworkRequest::ContentTypeHeader, "application/json;
    charset=utf8");
22
23      // 设置请求 id
24      reqBody["requestId"] = makeRequestId();
25
26      QJsonDocument document(reqBody);
27      QNetworkReply* httpResp = httpClient.post(httpReq, document.toJson());
28
29
30      // 3. 异步处理响应
31      connect(httpResp, &QNetworkReply::finished, this, [=]() {
32          // 判定Http层面是否出错
33          if(httpResp->error() != QNetworkReply::NoError){
34              LOG()<<httpResp->errorString();
35              httpResp->deleteLater();
36              return;
37          }
38
39          // 获取到响应的body
40          QByteArray respBody = httpResp->readAll();
41
42          // 针对body反序列化
43          QJsonDocument jsonDoc = QJsonDocument::fromJson(respBody);
44          if(jsonDoc.isNull()){
45              LOG()<<"解析 JSON 文件失败! JSON 文件格式有错误!";
46              httpResp->deleteLater();
47              return;
48          }
49
50          // 判定业务上的逻辑是否正确
```

```
        QJsonObject respObj = jsonDoc.object();
        if(0 != respObj["errorCode"].toInt()){
            LOG()<<respObj["errorMsg"].toString();
            httpResp->deleteLater();
            return;
        }

        // 解析响应数据
        QJsonObject resoBody = respObj["data"].toObject();
        LOG()<<resoBody["ping"].toString();
        httpResp->deleteLater();
    });
}
```

观察代码发现，hello()请求和ping()请求中大部分代码是相同的，只有构造请求体，和解析响应方法部分不同，其余基本相同，因此可以将封装请求的方法拆分，将重复代码提取出来减少代码冗余。

代码块

```
////////////////////////////// netclient.h ////////////////////////////////
class NetClient : public QObject
{
    // ...
    // 生成请求 id
    static QString makeRequestId();
    // 封装发送请求的逻辑
    QNetworkReply* sendHttpRequest(const QString& resourcePath, QJsonObject& body);

    // 封装处理响应的逻辑(包括判定 HTTP 正确性，反序列化，判定业务上的正确性)
    // 通过输出型参数，表示这次操作是成功还是失败，以及失败的原因.
    QJsonObject handleHttpResponse(QNetworkReply* httpResp, bool& ok, QString& reason);
    // ...
};
```

代码块

```
////////////////////////////// netclient.cpp ////////////////////////////////
// ...
QString NetClient::makeRequestId()
{
    // 基本要求，确保每个请求的 id 都是不重复(唯一的)
    // 通过 UUID 来实现上述效果.
    return "R" + QUuid::createUuid().toString().sliced(25, 12);
```

```cpp
}

// 通过这个函数，把发送 HTTP 请求操作封装一下.
QNetworkReply *NetClient::sendHttpRequest(const QString &resourcePath,
QJsonObject &jsonBody)
{
    QNetworkRequest httpReq;
    httpReq.setUrl(QUrl(HTTP_URL + resourcePath));
   httpReq.setHeader(QNetworkRequest::ContentTypeHeader, "application/json;
charset=utf8");

    // 设置请求 id
    jsonBody["requestId"] = makeRequestId();

    QJsonDocument document(jsonBody);
    QNetworkReply* httpResp = httpClient.post(httpReq, document.toJson());
    return httpResp;
}

QJsonObject NetClient::handleHttpResponse(QNetworkReply *httpResp, bool& ok,
                                          QString& reason) {
    // 1. 判定 HTTP 层面上，是否出错
    if (httpResp->error() != QNetworkReply::NoError) {
        ok = false;
        reason = httpResp->errorString();
        httpResp->deleteLater();
        return QJsonObject();
    }

    // 2. 获取到响应的 body
    QByteArray respBody = httpResp->readAll();

    // 3. 针对 body 反序列化
    QJsonDocument jsonDoc = QJsonDocument::fromJson(respBody);
    if (jsonDoc.isNull()) {
        ok = false;
        reason = "解析 JSON 文件失败！ JSON 文件格式有错误!";
        httpResp->deleteLater();
        return QJsonObject();
    }

    QJsonObject respObj = jsonDoc.object();
    // 4. 判定业务上的结果是否正确
    //    注意：错误码0表示没有错误
    if (0 != respObj["errorCode"].toInt()) {
        ok = false;
        reason = respObj["errorMsg"].toString();
```

```
        httpResp->deleteLater();
        return respObj;
    }

    // 5. 释放 httpResp 对象
    httpResp->deleteLater();
    ok = true;
    return respObj;
}
```

代码块

```
void NetClient::hello()
{
    // 1. 构造请求体 body
    QJsonObject reqBody;

    // 2. 发送请求
    QNetworkReply* httpResp = sendHttpRequest("/hello", reqBody);

    // 3. 异步处理响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // 1）解析HTTP响应
        bool ok = false;
        QString reason;
        QJsonObject respObj = handleHttpResponse(httpResp, ok, reason);

        // 2）判定响应是否出错
        if(!ok){
            LOG()<<"hello 请求出错，reason = "<<reason;
            return;
        }

        // 3）解析响应数据
        QJsonObject resoBody = respObj["data"].toObject();
        LOG()<<resoBody["hello"].toString();
    });
}

void NetClient::ping()
{
    // 1. 构造请求体 body
    QJsonObject reqBody;

    // 2. 发送请求
    QNetworkReply* httpResp = sendHttpRequest("/ping", reqBody);
```

```
35
36      // 3. 异步处理响应
37      connect(httpResp, &QNetworkReply::finished, this, [=]() {
38          // 1) 解析响应
39          bool ok;
40          QString reason;
41          QJsonObject respObj = handleHttpResponse(httpResp, ok, reason);
42
43          // 2) 判定响应是否出错
44          if(!ok){
45              LOG() << "ping 请求出错! reason = " << reason;
46              return;
47          }
48
49          // 3) 解析响应数据
50          QJsonObject resoBody = respObj["data"].toObject();
51          LOG()<<resoBody["ping"].toString();
52      });
53  }
54
```

6. 结构调整

界面层通常负责显示数据和处理用户输入，而与HTTP服务器交互属于数据访问和业务逻辑范畴，如果直接让界面与HTTP服务器交互，会导致界面成大过多职责，当项目复杂时，如果修改HTTP交互逻辑，可能需要修改大量界面代码，增加了可维护成本。

为了降低耦合性，在界面和HTTP服务之间引入中间层，用户需要发起HTTP请求时，通过中间层进行处理，中间层负责与HTTP服务器交互。在本项目中，可以利用DataCenter作为中间层，即DataCenter不仅承担了数据的缓存，还承担了界面和HTTP服务器交互的桥梁。