C++ 聊天室 - 客户端代码开发

版权说明

对比特项目感兴趣，可以联系这个微信。

代码 & 板书链接

https://gitee.com/bitedu-tech/cpp-chatsystem

项目概况

基于 C++ 实现一个 客户端-服务器 结构的聊天程序.

- 客户端: 基于 Qt 实现. 由汤老湿负责开发和讲解.
- 服务器: 基于 C++ 的分布式微服务架构 + 主流后端组件. 由超哥负责开发和讲解.

服务器微服务个数	7 个
服务器组件个数	17 个
业务功能点	40+
前后端交互接口	40+
数据库表个数	6 个
总代码量	1.8w

整体来说, 这是一个综合性比较强的项目. 具有一定的挑战性.

前置要求

本项目具有一定的难度和综合性, 对于学习本项目的同学, 提出以下要求:

- 充分掌握了直播课堂的所有知识, 并学有余力.
- 已经完成基础项目 2 个及以上.
- 热爱技术, 有钻研精神, 不怕困难.
- 有充分的学习时间, 愿意花较长的时间来学习编写这个项目.

> 💡
> 1. 这个项目并不适合所有同学去看. 对于基础一般或者薄弱的同学, 建议还是优先做班级中的基础项目.
> 2. 简历上的项目不是越难, 越复杂就越好. 一切的前提是能否给面试官介绍清楚.

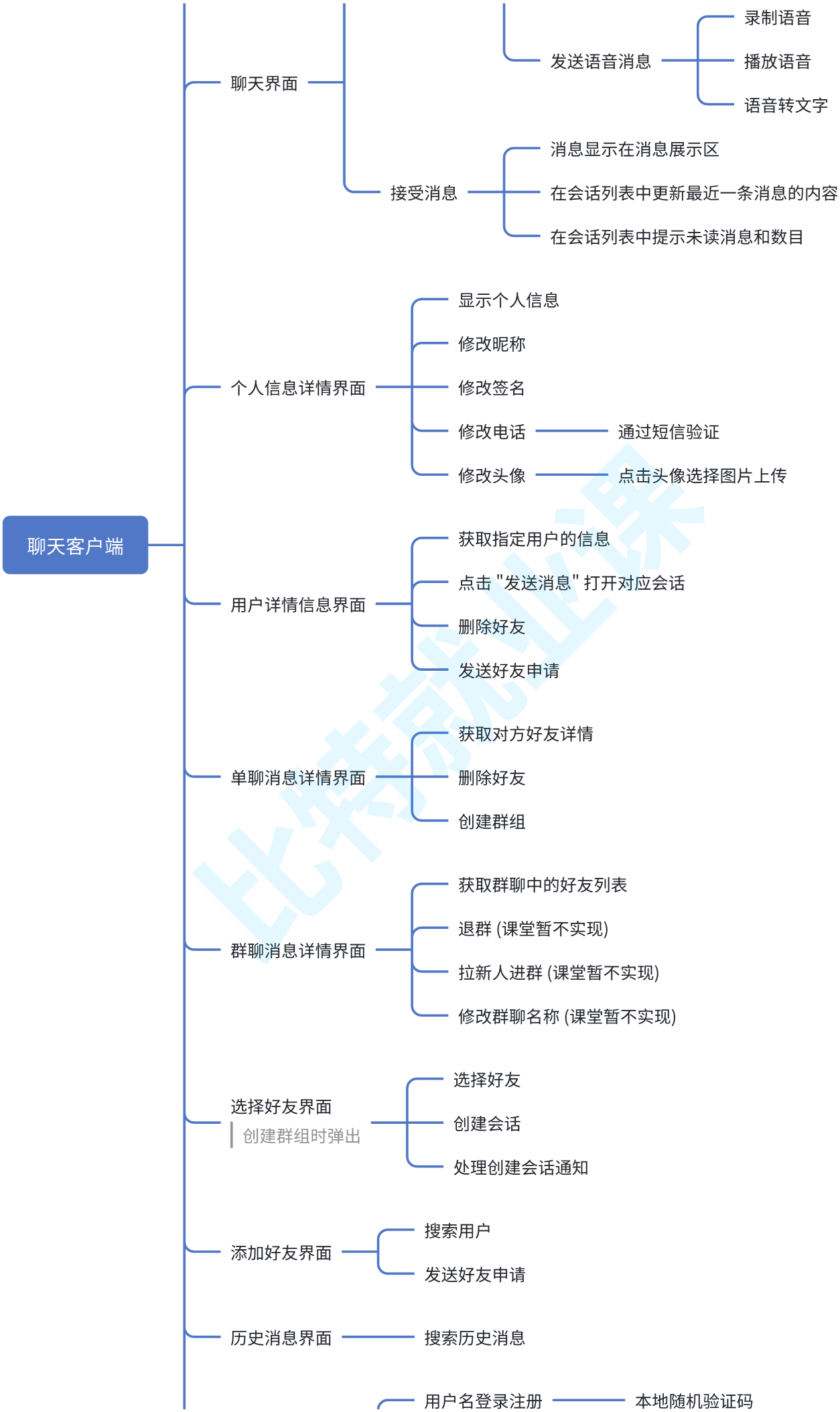
另外,

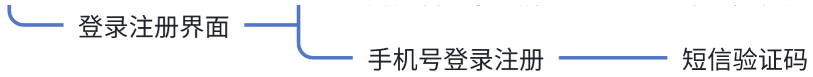
- 对于专注于做 Qt 的同学, 可以只看 "客户端开发部分"
- 对于专注于做 后端 的同学, 也可以只看 "服务器开发部分"

当然, 能全部掌握当然是更好的选择.

界面预览 & 功能介绍

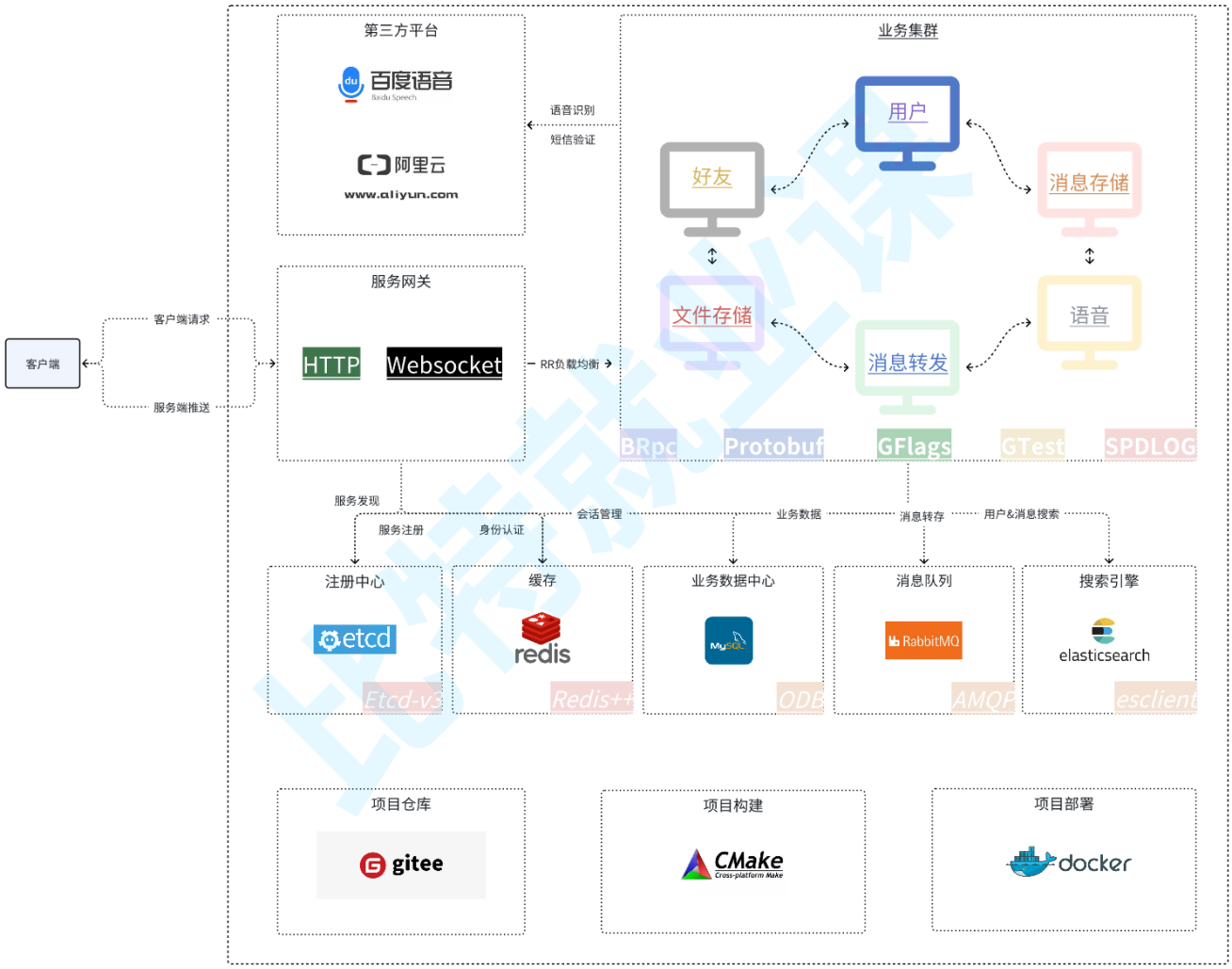
实现一个聊天程序客户端.


聊天客户端
聊天界面
发送语音消息
录制语音
播放语音
语音转文字
接受消息
消息显示在消息展示区
在会话列表中更新最近一条消息的内容
在会话列表中提示未读消息和数目
个人信息详情界面
显示个人信息
修改昵称
修改签名
修改电话
通过短信验证
修改头像
点击头像选择图片上传
用户详情信息界面
获取指定用户的信息
点击 "发送消息" 打开对应会话
删除好友
发送好友申请
单聊消息详情界面
获取对方好友详情
删除好友
创建群组
群聊消息详情界面
获取群聊中的好友列表
退群 (课堂暂不实现)
拉新人进群 (课堂暂不实现)
修改群聊名称 (课堂暂不实现)
选择好友界面
创建群组时弹出
选择好友
创建会话
处理创建会话通知
添加好友界面
搜索用户
发送好友申请
历史消息界面
搜索历史消息
用户名登录注册
本地随机验证码

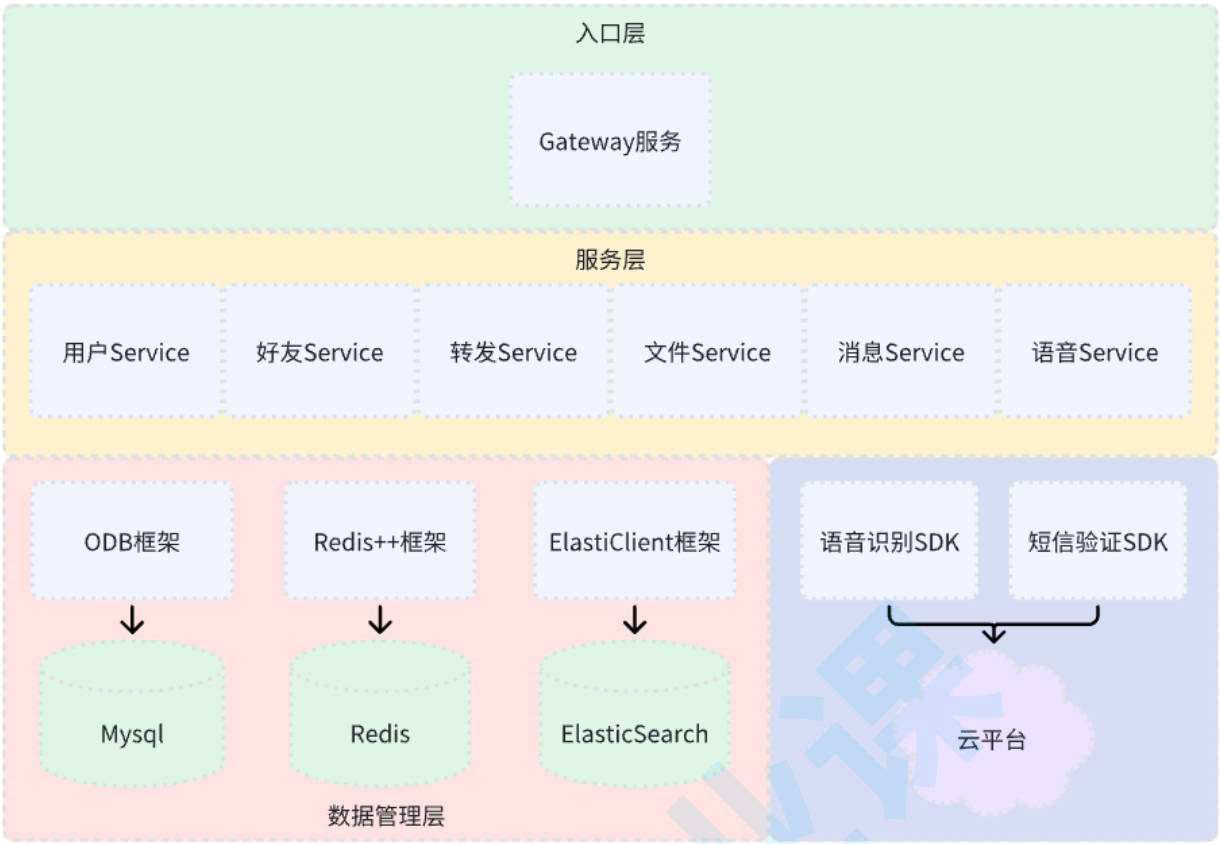


技术要点 & 服务器架构

该项目在设计的时候采用微服务框架设计，指将一个大的业务拆分称为多个子业务，分别在多台不同的机器节点上提供对应的服务，由网关服务统一接收多个客户端的各种不同请求，然后将请求分发到不同的子服务节点上进行处理，获取响应后，再转发给客户端。



模块层次



服务拆分：

- 入口网关服务器：主要用于与客户端直接交互，接收客户端的各项请求提供服务。
- 用户管理子服务：主要用于管理用户的数据，以及关于用户信息的各项操作。
- 好友管理子服务：主要用于管理好友与聊天会话管理相关的数据与操作。
- 转发管理子服务：主要用于封装消息进行转存，然后告诉网关服务器一条消息应该发给谁。
- 消息存储子服务：主要用于进行消息元信息的存储与搜索功能。
- 文件管理子服务：主要用于管理系统中文件类型数据的存储，比如用户头像，文件消息等。
- 语音转换子服务：用于调用语音识别SDK，进行语音识别，将语音转换为文字。

技术要点：

- gflags：针对程序运行所需的运行参数解析/配置文件解析框架。
- gtest：针对程序编写到一定阶段后，进行的单元测试框架。
- spdlog：针对项目中进行日志输出的框架。
- protobuf：针对项目中的网络通信数据所采用的序列化和反序列化框架。
- brpc：项目中的rpc调用使用的框架。
- redis：高性能键值存储系统，用于项目中进行用户登录会话信息的存储管理。

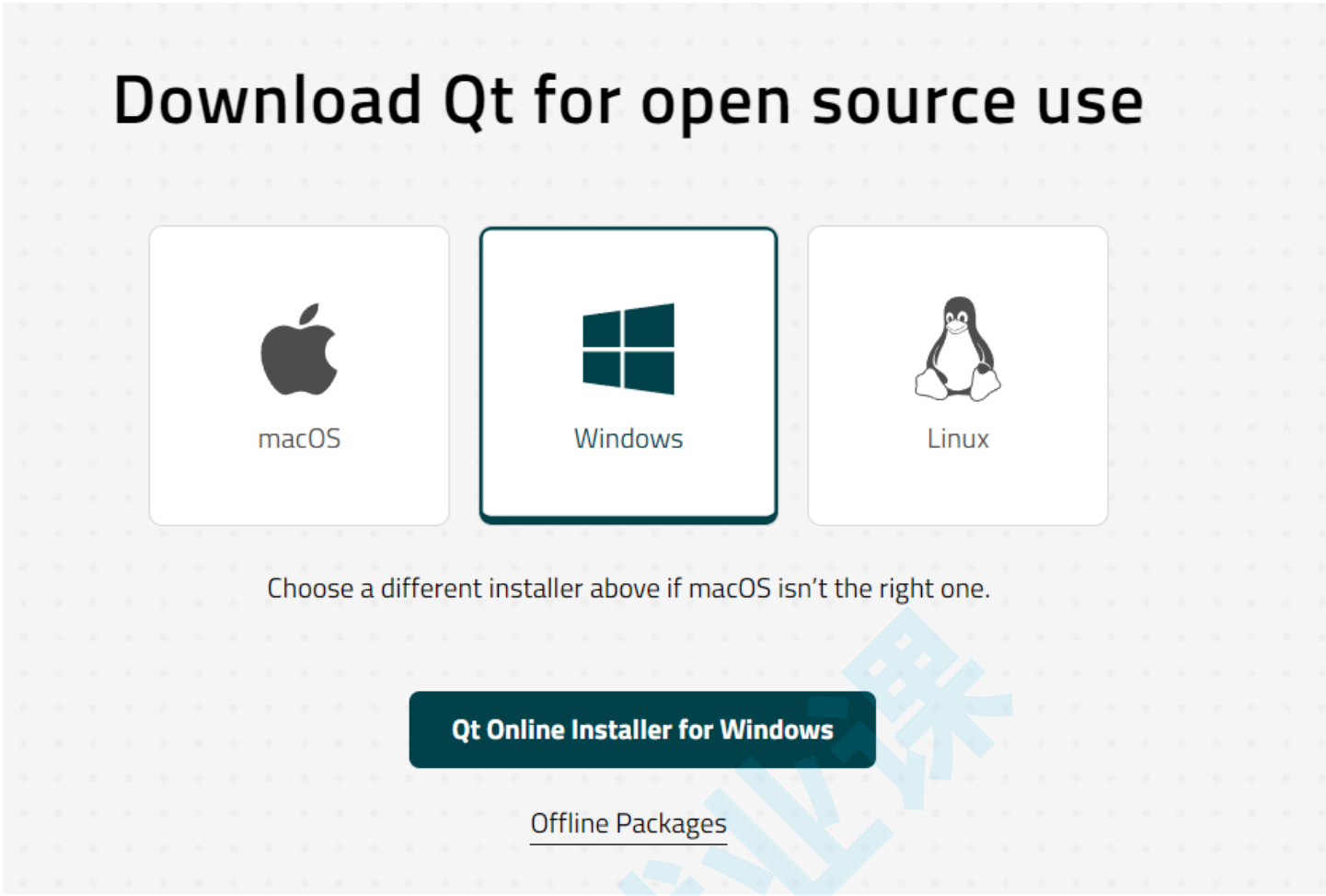
- mysql：关系型数据库系统，用于项目中的业务数据的存储管理。
- ODB：项目中mysql数据库操作的ORM框架（Object-Relational Mapping，对象关系映射）
- Etcd：分布式、高可用的一致性键值存储系统，用于项目中实现服务注册与发现功能的框架。
- cpp-httplib：用于搭建简单轻量HTTP服务器的框架。
- websocketpp：用于搭建Websocket服务器的框架。
- rabbitMQ：用于搭建消息队列服务器，用于项目中持久化消息的转发消费。
- elasticsearch：用于搭建文档存储/搜索服务器，用于项目中历史消息的存储管理
- 语音云平台：采用百度语音识别技术云平台实现语音转文字功能。
- 短信云平台：采用阿里云短信云平台实现手机短信验证码通知功能。
- cmake：项目工程的构建工具。
- docker：项目工程的一键式部署工具。

环境搭建

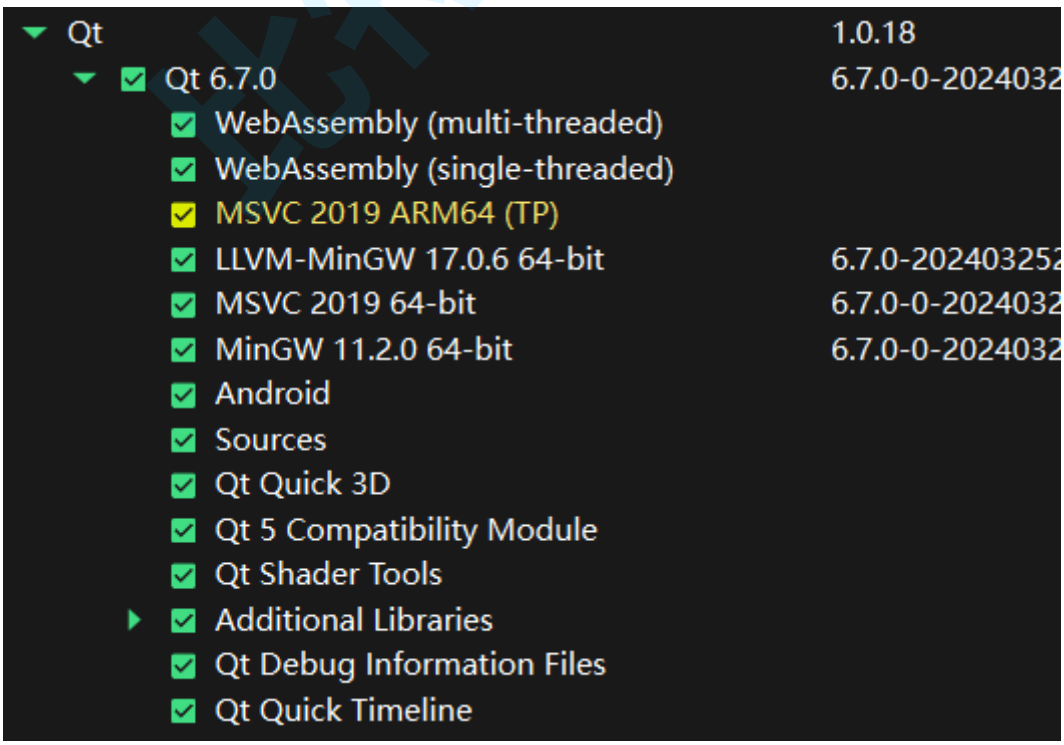
安装 Qt6

https://www.qt.io/download-qt-installer-oss?hsCtaTracking=99d9dd4f-5681-48d2-b096-470725510d34%7C074ddad0-fdef-4e53-8aa8-5e8a876d6ab4

下载在线安装包



- 选择 MSVC 2019 套件
- 机器上要能包含 VS2019 或者更高版本
- 后续创建项目要使用 cmake 作为构建工具



> 要勾选所有. 尤其是 Additional Libraries. 这里包含 protobuf 的 cmake 插件.

💡 如果下载过程不稳定, 一直出现网络错误, 可以更换成清华镜像作为下载源, 能有所改善.

1. 进入到在线安装包所在目录
2. 把安装包名字改成 installer.exe (或者随便什么简单的名字).
3. Shift + 右键 目录空白区域, 选择 "在此处打开 powershell"
4. 运行 `.\installer.exe –mirror https://mirrors.tuna.tsinghua.edu.cn/qt`

安装 vcpkg

```
git clone https://github.com/microsoft/vcpkg.git
cd vcpkg && bootstrap-vcpkg.bat
```

参考文档:

https://learn.microsoft.com/zh-cn/vcpkg/get_started/get-started?pivots=shell-cmd

安装 protobuf

```
vcpkg.exe install protobuf protobuf:x64-windows
```

参考文档:

https://doc.qt.io/qt-6/qtgrpc-examples.html

https://doc.qt.io/qt-6/qtprotobuf-installation-windows-vcpkg.html

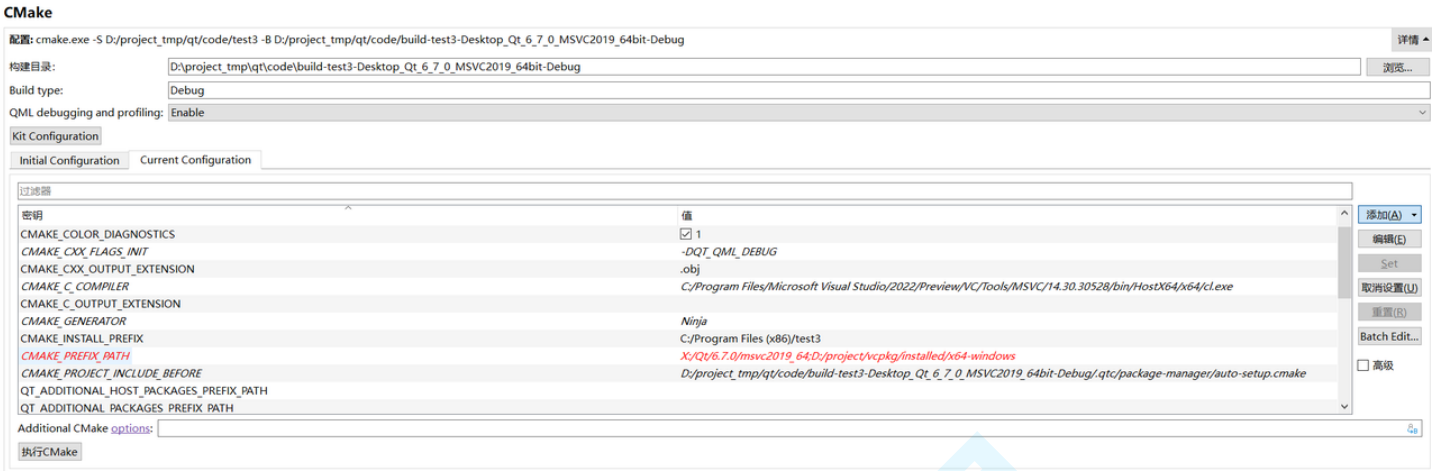
https://doc.qt.io/qt-6/qtgrpc-index.html

配置 cmake 属性

点击左侧边栏 "项目", 然后修改 cmake 的配置项: `CMAKE_PREFIX_PATH`

添加上 protoc 和 grpc 的路径前缀.

> **注意:**
>
> - 切换 debug release 时也要同时修改上述配置.
> - 添加到 current configuration 标签页, 而不是 Initial Configuration 标签页.



> 如果未能正确配置, 会出现形如
>
> ```
> CMake Error at client/CMakeLists.txt:13 (find_package):
>   Found package configuration file:
>
>     X:/Qt/6.7.0/msvc2019_64/lib/cmake/Qt6/Qt6Config.cmake
>
>   but it set Qt6_FOUND to FALSE so package "Qt6" is considered to be NOT
>   FOUND.
> ```
>
> 这样的错误

创建项目

基于 Qt6, 创建 cmake 项目. 编译套件选择基于 MSVC.

- 后续需要用到 protobuf, 依赖 Qt6 + cmake
- Qt6 支持的组件更丰富更强大.
- 网上关于 Qt6 的资料也越来越多. 继续使用 Qt5 意味着很多网上查到的内容和代码会存在差异
- 使用 MSVC 而不是使用 mingw. 否则编译速度可能会很慢.

核心数据结构 (1)

创建核心类

创建 `data.h` 存放核心的类.

```cpp
////////////////////////////////////////////////////
/// 用户信息
////////////////////////////////////////////////////

class UserInfo {
public:
    QString userId;
    QString nickname;
    QString description;
    QString phone;
    QIcon avatar;
};

////////////////////////////////////////////////////
/// 消息信息
////////////////////////////////////////////////////

enum MessageType {
    TEXT_TYPE,                  // 文本消息
    IMAGE_TYPE,                 // 图片消息
    FILE_TYPE,                  // 文件消息
    SPEECH_TYPE,                // 语音消息
    UNKNOWN_TYPE                // 错误类型
};

class Message {
public:
    QString messageId = "";
    QString chatSessionId = "";
    QString time = "";  // 格式化时间
    MessageType messageType = TEXT_TYPE;
    UserInfo sender;
    // 实际内容取决于 messageType
    QByteArray content;
    // 如果是图片，文件，语音类型，表示对应的文件 id
    QString fileId = "";
    // 如果是文件消息，表示文件名
    QString fileName = "";

    // 生成消息 ID
    static QString makeId() {
        return "M" + QUuid::createUuid().toString().sliced(25, 12);
    }

    static Message makeMessage(MessageType messageType, const QString&
chatSessionId,
```

```
46                                const UserInfo& sender, const QByteArray&
content, const QString& extraInfo) {
47          if (messageType == TEXT_TYPE) {
48              return makeTextMessage(chatSessionId, sender, content);
49          } else if (messageType == SPEECH_TYPE) {
50              return makeSpeechMessage(chatSessionId, sender, content);
51          } else if (messageType == IMAGE_TYPE) {
52              return makeImageMessage(chatSessionId, sender, content);
53          } else if (messageType == FILE_TYPE) {
54              return makeFileMessage(chatSessionId, sender, content, extraInfo);
55          } else {
56              return Message();
57          }
58      }
59
60      // 生成一个文本消息
61      static Message makeTextMessage(const QString& chatSessionId, const
UserInfo& sender, const QByteArray& content) {
62          Message message;
63          message.messageId = makeId();
64          message.chatSessionId = chatSessionId;
65          message.sender = sender;
66          message.time = formatTime(getTime());
67          message.messageType = TEXT_TYPE;
68          message.content = content;
69          return message;
70      }
71
72      // 生成一个语音消息
73      static Message makeSpeechMessage(const QString& chatSessionId, const
UserInfo& sender, const QByteArray& content) {
74          Message message;
75          message.messageId = makeId();
76          message.chatSessionId = chatSessionId;
77          message.sender = sender;
78          message.time = formatTime(getTime());
79          message.messageType = SPEECH_TYPE;
80          message.content = content;
81          // fileId 在创建空消息时不需要.
82          // message.fileId = "";
83          return message;
84      }
85
86      static Message makeImageMessage(const QString& chatSessionId, const
UserInfo& sender, const QByteArray& content) {
87          Message message;
88          message.messageId = makeId();
```

```
        message.chatSessionId = chatSessionId;
        message.sender = sender;
        message.time = formatTime(getTime());
        message.messageType = IMAGE_TYPE;
        message.content = content;
        return message;
    }

    static Message makeFileMessage(const QString& chatSessionId, const UserInfo& sender, const QByteArray& content, const QString& fileName) {
        Message message;
        message.messageId = makeId();
        message.chatSessionId = chatSessionId;
        message.sender = sender;
        message.time = formatTime(getTime());
        message.messageType = FILE_TYPE;
        message.content = content;
        message.fileName = fileName;
        return message;
    }
};


//////////////////////////////////////////////////////
/// 聊天会话信息
//////////////////////////////////////////////////////

class ChatSessionInfo {
public:
    QString chatSessionId = "";
    QString chatSessionName = "";
    Message prevMessage;
    QIcon avatar;
    // 当会话为单聊时，表示对方的用户 id
    // 如果为群聊，则该字段为 ""
    QString userId = "";
};
```

创建工具函数

```
//////////////////////////////////////////////////////
/// 工具函数
//////////////////////////////////////////////////////
```

```cpp

// 根据 QByteArray 转成 QIcon
static inline QIcon makeIcon(const QByteArray& byteArray) {
    QPixmap pixmap;
    pixmap.loadFromData(byteArray);
    QIcon icon(pixmap);
    return icon;
}

// 获取到图片的二进制数据
static inline QByteArray loadImageToByteArray(const QString& fileName) {
    // 1. 加载为 QImage
    QImage image(fileName);
    if (image.isNull()) {
        LOG() << "图片无法加载!";
        return QByteArray();
    }
    // 2. 转换成 QPixmap
    QPixmap pixmap = QPixmap::fromImage(image);

    // 3. 获取图片类型，是 png 还是 jpg 还是啥别的
    QFileInfo fileInfo(fileName);
    QString type = fileInfo.suffix();

    // 4. 进行转换
    QByteArray byteArray;
    QBuffer buffer(&byteArray);
    bool ok = false;
    if (type == "png") {
        // 此处的 save 支持 BMP, JPG, JPEG, PNG
        ok = pixmap.save(&buffer, "PNG");
    } else if (type == "jpg" || type == "jpeg") {
        ok = pixmap.save(&buffer, "JPG");
    } else if (type == "bmp") {
        ok = pixmap.save(&buffer, "BMP");
    } else {
        LOG() << "图片类型不支持!";
        return QByteArray();
    }
    if (!ok) {
        // 如果保存失败，返回空的字节数组
        return QByteArray();
    }
    return byteArray;
}

// 获取到文件的二进制内容
```

```cpp
static inline QByteArray loadFileToByteArray(const QString& filename) {
    QFile file(filename);
    bool ok = file.open(QFile::ReadOnly);
    if (!ok) {
        LOG() << "文件读取失败!";
        return QByteArray();
    }
    QByteArray content = file.readAll();
    file.close();
    return content;
}

// 把二进制数据写入文件
static inline void writeByteArrayToFile(const QString& filename, const QByteArray& content) {
    QFile file(filename);
    bool ok = file.open(QFile::WriteOnly);
    if (!ok) {
        LOG() << "文件写入失败!";
        return;
    }
    file.write(content);
    file.flush();
    file.close();
    return;
}

// 把时间戳转成 01-01 18:00:00 这样的格式化时间
static inline QString formatTime(int64_t timestamp) {
    QDateTime dateTime = QDateTime::fromSecsSinceEpoch(static_cast<time_t>(timestamp));
    return dateTime.toString("MM-dd HH:mm:ss");
}

// 获取当前秒级时间戳
static inline int64_t getTime() {
    return QDateTime::currentSecsSinceEpoch();
}
```

创建编译开关和日志

创建 `debug.h`

```cpp
#include <QString>
#include <QFileInfo>
```

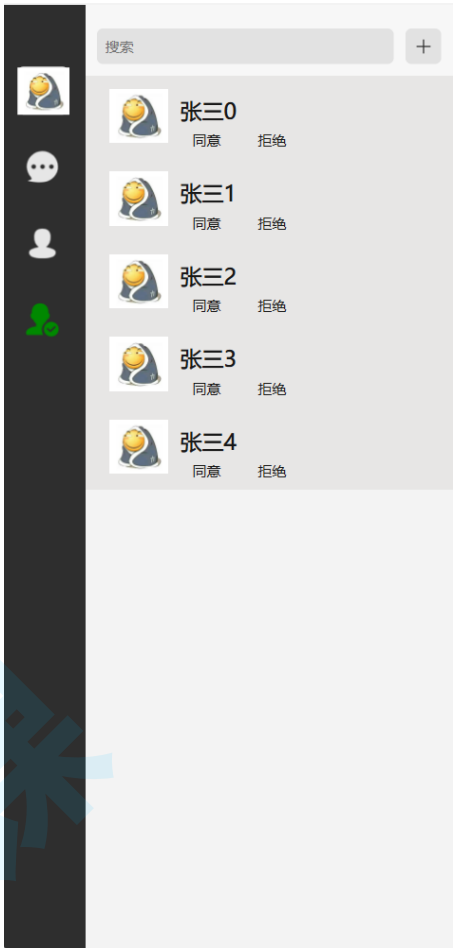
```
3
4  // 辅助打印日志的宏定义
5  static inline QString getFileName(const QString& path) {
6      QFileInfo fileInfo(path);
7      return fileInfo.fileName();
8  }
9
10 #define TAG QString("[%1:%2]").arg(getFileName(__FILE__),
   QString::number(__LINE__))
11 // 使用 .noquote() 设置 qDebug 针对字符串不输出 ""
12 #define LOG() qDebug().noquote() << TAG
13
14 // 测试 UI , 显示构造的假数据
15 #define TEST_UI 0
16 // 测试群组会话详情窗口
17 #define TEST_GROUP_SESSION_DETAIL 1
18 // 测试跳过登录窗口
19 #define TEST_SKIP_LOGIN 0
20 // 测试网络连通性
21 #define TEST_NETWORK 0
22 // 从网络获取数据
23 #define LOAD_DATA_FROM_NETWORK 1
24 // 是否连接测试服务器
25 #define CONNECT_TEST_SERVER 0
```

界面布局

实现主界面

程序的主界面.

界面效果:



1) 创建 `MainWidget` 表示主窗口

```
class MainWidget : public QWidget
{
    Q_OBJECT

public:
    ~MainWidget();

    static MainWidget* getInstance();

private:
    Ui::MainWidget *ui;

    static MainWidget* instance;
    MainWidget(QWidget *parent = nullptr);

private:
    QWidget* windowLeft;
    QWidget* windowMid;
    QWidget* windowRight;

    // 用户头像
    QPushButton* userAvatar;
```

```
23
24      // tab 按钮
25      QPushButton* sessionTabBtn;
26      QPushButton* friendTabBtn;
27      QPushButton* applyTabBtn;
28
29      // 搜索框
30      QLineEdit* searchEdit;
31
32      // 添加好友按钮
33      QPushButton* addFriendBtn;
34
35      // 显示会话列表和好友列表的
36      SessionFriendArea* sessionFriendArea;
37
38      // 右侧窗口上方的会话标题
39      QLabel* sessionTitle;
40
41      // 右侧上方的会话详情按钮
42      QPushButton* extraButton;
43
44      // 右侧窗口中间的消息显示区
45      MessageShowArea* messageShowArea;
46
47      // 右侧窗口下方的消息编辑区
48      MessageEditArea* messageEditArea;
49
50      // 描述当前激活的标签页
51      enum ActiveTab {
52          SESSION_LIST,               // 会话列表
53          FRIEND_LIST,                // 好友列表
54          APPLY_LIST                  // 好友申请列表
55      };
56      ActiveTab activeTab = SESSION_LIST;
57      ActiveTab getActiveTab();
58
59      void initUI();
60      void initMainWindow();
61      void initWindowLeft();
62      void initWindowMid();
63      void initWindowRight();
64  }
```

```
1  MainWidget::MainWidget(QWidget *parent)
2      : QWidget(parent)
```

```
    , ui(new Ui::MainWidget)
{
    ui->setupUi(this);

    this->setWindowTitle("我的聊天室");
    this->setWindowIcon(QIcon(":/image/logo.png"));

    // 初始化界面
    initUI();
}
```

```
void MainWidget::initUI()
{
    // 1. 初始化整体窗口布局
    initMainWindow();
    // 2. 初始化左侧
    initWindowLeft();
    // 3. 初始化中间
    initWindowMid();
    // 4. 初始化右侧
    initWindowRight();
}

void MainWidget::initMainWindow()
{
    QHBoxLayout* layout = new QHBoxLayout();
    layout->setSpacing(0);
    layout->setContentsMargins(0, 0, 0, 0);
    this->setLayout(layout);

    windowLeft = new QWidget();
    windowMid = new QWidget();
    windowRight = new QWidget();

    windowLeft->setObjectName("windowLeft");
    windowMid->setObjectName("windowMid");
    windowRight->setObjectName("windowRight");

    windowLeft->setFixedWidth(70);
    windowMid->setFixedWidth(310);
    // windowRight 宽度自适应，不指定固定值，只指定最小值
    windowRight->setMinimumWidth(900);

    layout->addWidget(windowLeft);
    layout->addWidget(windowMid);
```

```
35      layout->addWidget(windowRight);
36
37      QString style = "#windowLeft { background-color: rgb(46, 46, 46); }
    #windowMid { background-color: rgb(247, 247, 247); } #windowRight {background-
    color: rgb(245, 245, 245);} ";
38      this->setStyleSheet(style);
39  }
40
41  void MainWidget::initWindowLeft()
42  {
43      QVBoxLayout* layout = new QVBoxLayout();
44      layout->setContentsMargins(0, 50, 0, 0);
45      layout->setSpacing(20);
46      windowLeft->setLayout(layout);
47
48      // 初始化用户头像
49      userAvatar = new QPushButton();
50      userAvatar->setFixedSize(45, 45);
51      userAvatar->setIconSize(QSize(45, 45));
52  #if TEST_UI
53      userAvatar->setIcon(QIcon(":/image/defaultAvatar.png"));
54  #endif
55      layout->addWidget(userAvatar, 1, Qt::AlignTop | Qt::AlignHCenter);
56
57      // 初始化 tab 按钮
58      sessionTabBtn = new QPushButton();
59      sessionTabBtn->setFixedSize(45, 45);
60      sessionTabBtn->setIcon(QIcon(":/image/session_active.png"));
61      sessionTabBtn->setIconSize(QSize(30, 30));
62      sessionTabBtn->setStyleSheet("QPushButton { background-color: transparent;
    }");
63      layout->addWidget(sessionTabBtn, 1, Qt::AlignTop | Qt::AlignHCenter);
64
65      friendTabBtn = new QPushButton();
66      friendTabBtn->setFixedSize(45, 45);
67      friendTabBtn->setIcon(QIcon(":/image/friend_inactive.png"));
68      friendTabBtn->setIconSize(QSize(30, 30));
69      friendTabBtn->setStyleSheet("QPushButton { background-color: transparent;
    }");
70      layout->addWidget(friendTabBtn, 1, Qt::AlignTop | Qt::AlignHCenter);
71
72      applyTabBtn = new QPushButton();
73      applyTabBtn->setFixedSize(45, 45);
74      applyTabBtn->setIcon(QIcon(":/image/apply_inactive.png"));
75      applyTabBtn->setIconSize(QSize(30, 30));
76      applyTabBtn->setStyleSheet("QPushButton { background-color: transparent;
    }");
```

```
    layout->addWidget(applyTabBtn, 1, Qt::AlignTop | Qt::AlignHCenter);


    // 下方添加空隙，把上述控件挤上去.
    layout->addStretch(20);
}
```

2) 实现左侧 tab 页

添加信号槽

```
// 处理 tab 按钮的点击操作
connect(sessionTabBtn, &QPushButton::clicked, this,
&MainWidget::switchTabToSession);
connect(friendTabBtn, &QPushButton::clicked, this,
&MainWidget::switchTabToFriend);
connect(applyTabBtn, &QPushButton::clicked, this,
&MainWidget::switchTabToApply);
```

槽函数的实现

```
void MainWidget::switchTabToSession()
{
    // 1. 设置 activeTab
    activeTab = SESSION_LIST;
    // 2. 调整图标高亮
    sessionTabBtn->setIcon(QIcon(":/image/session_active.png"));
    friendTabBtn->setIcon(QIcon(":/image/friend_inactive.png"));
    applyTabBtn->setIcon(QIcon(":/image/apply_inactive.png"));
    // 3. TODO 加载会话列表数据
    this->loadSessionList();
}

void MainWidget::switchTabToFriend()
{
    // 1. 设置 activeTab
    activeTab = FRIEND_LIST;
    // 2. 调整图标高亮
    sessionTabBtn->setIcon(QIcon(":/image/session_inactive.png"));
    friendTabBtn->setIcon(QIcon(":/image/friend_active.png"));
    applyTabBtn->setIcon(QIcon(":/image/apply_inactive.png"));
    // 3. TODO 加载好友列表数据
```

```
    this->loadFriendList();
}

void MainWidget::switchTabToApply()
{
    // 1. 设置 activeTab
    activeTab = APPLY_LIST;
    // 2. 调整图标高亮
    sessionTabBtn->setIcon(QIcon(":/image/session_inactive.png"));
    friendTabBtn->setIcon(QIcon(":/image/friend_inactive.png"));
    applyTabBtn->setIcon(QIcon(":/image/apply_active.png"));
    // 3. TODO 加载好友列表数据
    this->loadApplyList();
}
```

3) 创建中间上方的搜索框和搜索按钮

```
void MainWidget::initWindowMid()
{
    QGridLayout* layout = new QGridLayout();
    layout->setContentsMargins(0, 20, 0, 0);
    layout->setVerticalSpacing(10);
    layout->setHorizontalSpacing(0);
    windowMid->setLayout(layout);
    // 去除默认边框. 否则边框会对后续布局产生影响.
    windowMid->setStyleSheet("QWidget { border: none; }");

    searchEdit = new QLineEdit();
    searchEdit->setFixedHeight(30);
    searchEdit->setPlaceholderText("搜索");
    searchEdit->setStyleSheet("QLineEdit { border-radius: 5px; background-color: rgb(226, 226, 226); padding-left: 5px;}");

    addFriendBtn = new QPushButton();
    addFriendBtn->setFixedSize(30, 30);
    addFriendBtn->setIcon(QIcon(":/image/cross.png"));
    QString style = "QPushButton { border-radius: 5px; background-color: rgb(226, 226, 226); } ";
    style += "QPushButton:pressed { background-color: rgb(240, 240, 240); } ";
    addFriendBtn->setStyleSheet(style);

    sessionFriendArea = new SessionFriendArea();

```

```
    // 由于 GridLayout 不能够给不同的行设置不同的边距
    // 因此通过添加空白 widget 来控制搜索框和搜索按钮之间的边距
    QWidget* spacer1 = new QWidget();
    spacer1->setFixedWidth(10);
    QWidget* spacer2 = new QWidget();
    spacer2->setFixedWidth(10);
    QWidget* spacer3 = new QWidget();
    spacer3->setFixedWidth(10);

    layout->addWidget(spacer1, 0, 0);
    layout->addWidget(searchEdit, 0, 1);
    layout->addWidget(spacer2, 0, 2);
    layout->addWidget(addFriendBtn, 0, 3);
    layout->addWidget(spacer3, 0, 4);
    // 参数 1, 0, 1, 5 表示放在第 1 行, 第 0 列, 行方向跨一行, 列方向跨五列.
    layout->addWidget(sessionFriendArea, 1, 0, 1, 5);
}

```

4) 实现中间的会话列表, 消息列表, 好友申请列表

创建 `SessionFriendArea` 类

```
////////////////////////////////////////////////////
/// 表示整个 会话-好友 区域
////////////////////////////////////////////////////

class SessionFriendArea : public QScrollArea
{
    Q_OBJECT
public:
    SessionFriendArea();

    // 清空所有元素
    void clear();

    // 添加一个元素, text 可能表示签名, 也可能表示最后一条消息.
    void addItem(ItemType itemType, const QString& id, const QIcon& avatar,
const QString& name, const QString& text);

    // 点击指定下标的元素
    void clickItem(int index);

private:
```

```
    QWidget* container;
};
```

实现界面布局

```
SessionFriendArea::SessionFriendArea() {
    // 1. 设置基础属性
    // 一定要添加这个设置，否则无法正确显示.
    this->setWidgetResizable(true);
    // 设置滚动条宽度
    this->verticalScrollBar()->setStyleSheet("QScrollBar:vertical { width: 2px; background-color: rgb(46, 46, 46); }");
    this->horizontalScrollBar()->setStyleSheet("QScrollBar:horizontal { height: 0px;}");

    // 2. 创建核心 Widget
    container = new QWidget(this);
    container->setFixedWidth(310);
    this->setWidget(container);

    // 3. 创建布局
    QVBoxLayout* layout = new QVBoxLayout();
    layout->setContentsMargins(0, 0, 0, 0);
    layout->setSpacing(0);
    layout->setAlignment(Qt::AlignTop);
    container->setLayout(layout);

    // 测试滚动效果
    // for (int i = 0; i < 50; i++) {
    //     QPushButton* btn = new QPushButton("hello button");
    //     layout->addWidget(btn);
    //     LOG() << "---> " << i;
    // }
}
```

实现核心操作

```
void SessionFriendArea::clear()
{
    QLayout* layout = container->layout();
```

```
    // 从后往前遍历. 否则删除前面的元素会影响后面元素的下标
    for (int i = layout->count() - 1; i >= 0; i--) {
        QLayoutItem* item = layout->takeAt(i);
        if (item->widget()) {
            delete item->widget();
        }
        delete item;
    }
}

void SessionFriendArea::addItem(ItemType itemType, const QString &id, const QIcon& avatar, const QString &name, const QString &text)
{
    SessionFriendItem* itemWidget = nullptr;
    if (itemType == FriendItemType) {
        itemWidget = new FriendItem(this, id, avatar, name, text);
    } else if (itemType == SessionItemType) {
        itemWidget = new SessionItem(this, id, avatar, name, text);
    } else if (itemType == ApplyItemType) {
        itemWidget = new ApplyItem(this, id, avatar, name);
    }

    // 添加到 container 中
    container->layout()->addWidget(itemWidget);
}

void SessionFriendArea::clickItem(int index)
{
    if (index >= container->layout()->count()) {
        qCritical() << TAG << "点击的元素下标超出范围! index=" << index;
        return;
    }
    QLayoutItem* top = container->layout()->itemAt(index);
    if (top->widget() == nullptr) {
        qCritical() << TAG << "指定元素不存在! index=" << index;
        return;
    }
    SessionFriendItem* item = dynamic_cast<SessionFriendItem*>(top->widget());
    item->select();
}
```

5) 创建列表元素 - 父类

类声明

```
class SessionFriendItem : public QWidget
{
    Q_OBJECT

public:
    SessionFriendItem(QWidget* owner, const QIcon& avatar, const QString &name, const QString &lastMessage);

    void mousePressEvent(QMouseEvent *event) override;
    // 选中这一项要做的事情. 单独提出一个函数, 为了后续能直接调用.
    void select();

    void enterEvent(QEnterEvent* event) override;
    void leaveEvent(QEvent* event) override;
    void paintEvent(QPaintEvent *event) override;

    // 如果该元素被点击时, 要做哪些事情
    virtual void active() { }

protected:
    QGridLayout* layout;
    // 显示最后一条消息/描述信息
    QLabel* messageLabel;

    // 是否被选中
    bool selected;

    // 该 Item 所属的 QWidget. 此处填写 SessionFriendArea
    QWidget* owner;
};
```

样式布局

```
SessionFriendItem::SessionFriendItem(QWidget* owner, const QIcon& avatar, const QString &name, const QString &lastMessage)
    : owner(owner)
{
    this->selected = false;
    this->setStyleSheet("QWidget {background-color: rgb(231, 230, 229);}");
    this->setFixedHeight(70);

    layout = new QGridLayout();
    layout->setContentsMargins(20, 0, 0, 0);
    layout->setHorizontalSpacing(10);
```

```
    layout->setVerticalSpacing(0);
    this->setLayout(layout);

    // 创建头像
    QPushButton* avatarBtn = new QPushButton();
    avatarBtn->setFixedSize(50, 50);
    avatarBtn->setIcon(avatar);
    avatarBtn->setIconSize(QSize(50, 50));
    avatarBtn->setStyleSheet("QPushButton {background-color: rgb(231, 230, 229);}");
    avatarBtn->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);

    // 创建名字
    QLabel* nameLabel = new QLabel();
    nameLabel->setText(name);
    nameLabel->setStyleSheet("QLabel { font-size: 18px; font-weight: 600; }");
    nameLabel->setAlignment(Qt::AlignBottom); // 调整下文字对齐位置，看起来更好看一些.
    nameLabel->setFixedHeight(35);
    nameLabel->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);

    // 创建消息预览
    messageLabel = new QLabel();
    messageLabel->setText(lastMessage);
    messageLabel->setFixedHeight(35);
    messageLabel->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);

    layout->addWidget(avatarBtn, 0, 0, 2, 2);
    layout->addWidget(nameLabel, 0, 2, 1, 8);
    layout->addWidget(messageLabel, 1, 2, 1, 8);
}
```

实现鼠标悬停/选中元素效果

```
void SessionFriendItem::mousePressEvent(QMouseEvent *event)
{
    (void) event;
    select();
}

void SessionFriendItem::select()
{
    // 清空其他兄弟元素的样式
    const QObjectList& children = this->parentWidget()->children();
```

```
    for (QObject* child : children) {
        if (!child->isWidgetType()) {
            // 不是 QWidget 直接跳过.
            continue;
        }
        SessionFriendItem* widget = dynamic_cast<SessionFriendItem*>(child);
        widget->selected = false;
        widget->setStyleSheet("QWidget {background-color: rgb(231, 230, 229);}");
    }
    // 应用自己的样式.
    this->setStyleSheet("QWidget { background-color: rgb(210, 209, 209); }");
    this->selected = true;

    // TODO 进一步的加载详情. 按照多态的方式调用到对应子类的 active
    this->active();
}

void SessionFriendItem::enterEvent(QEnterEvent *event)
{
    (void) event;
    if (!this->selected) {
        this->setStyleSheet("QWidget { background-color: rgb(215, 215, 215); }");
    }
}

void SessionFriendItem::leaveEvent(QEvent *event)
{
    (void) event;
    if (!this->selected) {
        this->setStyleSheet("QWidget {background-color: rgb(231, 230, 229);}");
    }
}

void SessionFriendItem::paintEvent(QPaintEvent *event)
{
    // 这里是官方文档给出的要求. 原因在文档上没有解释.
    (void) event;
    QStyleOption opt;
    opt.initFrom(this);
    QPainter p(this);
    style()->drawPrimitive(QStyle::PE_Widget, &opt, &p, this);
}
```

> 出处在 Qt 文档 "style sheet reference" 章节的 QWidget 的详细描述中.

6) 创建列表元素 - 聊天会话

```
enum ItemType {
    SessionItemType,
    FriendItemType,
    ApplyItemType
};
```

```
 class SessionItem : public SessionFriendItem {
    Q_OBJECT
private:
    QString chatSessionId;

public:
    SessionItem(QWidget* owner, const QString& chatSessionId, const QIcon&
avatar, const QString &name, const QString &lastMessage);
    void active() override;
};
```

```
SessionItem::SessionItem(QWidget* owner, const QString &chatSessionId, const
QIcon &avatar, const QString &name, const QString &lastMessage)
    : SessionFriendItem(owner, avatar, name, lastMessage),
chatSessionId(chatSessionId)
{
}

void SessionItem::active()
{
    LOG() << "SessionItem active. chatSessionId=" << chatSessionId;
    // TODO
}

```

7) 创建列表元素 - 好友

```
class FriendItem : public SessionFriendItem {
    Q_OBJECT
private:
    QString userId;

public:
    FriendItem(QWidget* owner, const QString& userId, const QIcon& avatar,
const QString &name, const QString &description);
    void active() override;
};
```

```
FriendItem::FriendItem(QWidget* owner, const QString &friendId, const QIcon
&avatar, const QString &name, const QString &lastMessage)
    : SessionFriendItem(owner, avatar, name, lastMessage), userId(friendId)
{
}

void FriendItem::active()
{
    LOG() << "FriendItem active. userId=" << userId;
    // TODO
}
```

8) 创建列表元素 - 好友申请

```
class ApplyItem : public SessionFriendItem {
    Q_OBJECT
private:
    // 申请人的用户 id
    QString userId;

public:
    ApplyItem(QWidget* owner, const QString& userId, const QIcon& avatar, const
 QString& name);
    void active() override;
};
```

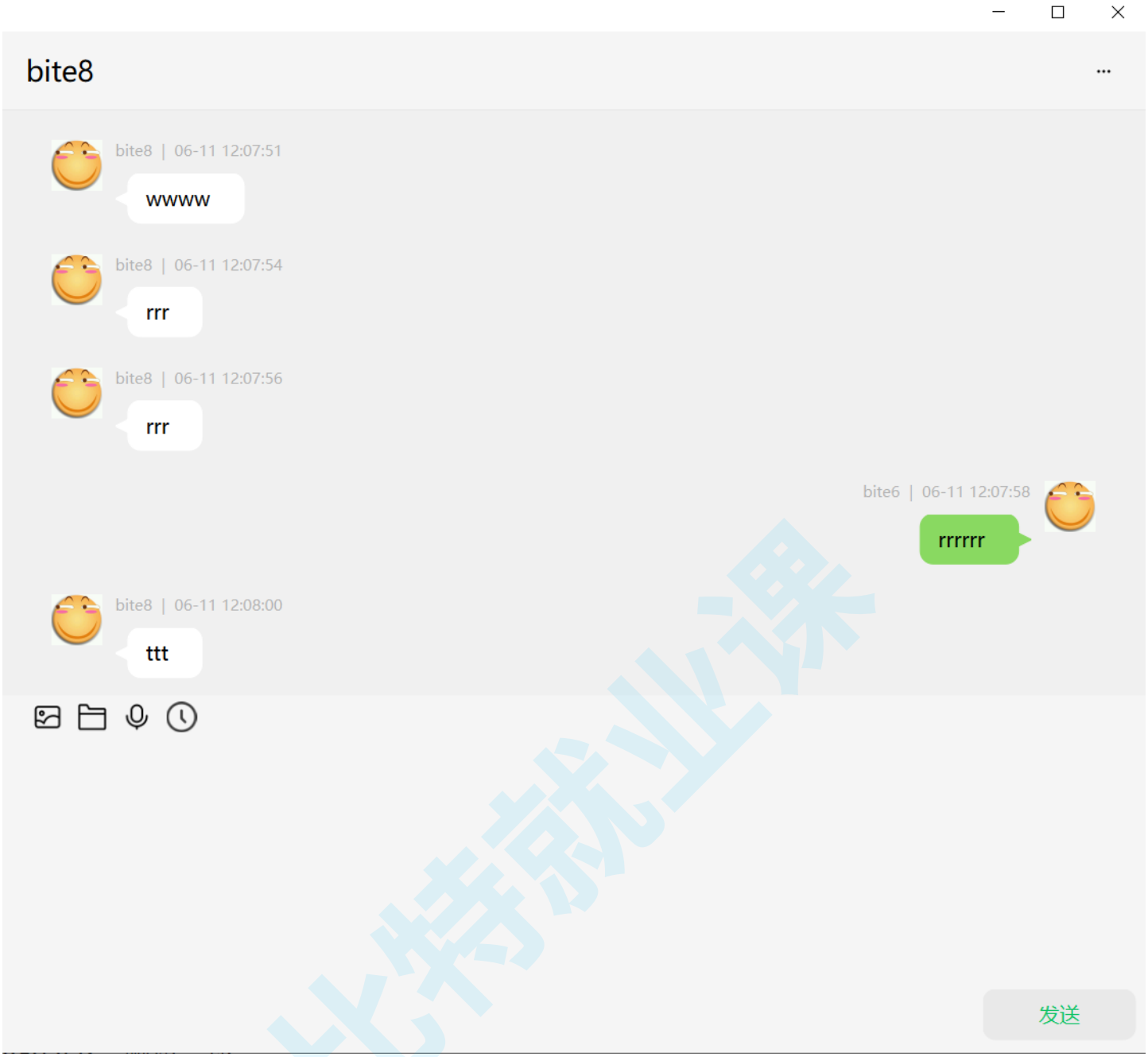
```
ApplyItem::ApplyItem(QWidget *owner, const QString& userId, const QIcon&
avatar, const QString& name)
```

```
    : SessionFriendItem(owner, avatar, name, ""), userId(userId)
{
    // 1. 移除 messageLabel
    layout->removeWidget(messageLabel);
    messageLabel->deleteLater();

    // 2. 创建两个按钮
    QPushButton* acceptBtn = new QPushButton();
    acceptBtn->setText("同意");
    QPushButton* rejectBtn = new QPushButton();
    rejectBtn->setText("拒绝");

    // 3. 添加到布局中
    // layout->addWidget(messageLabel, 1, 2, 1, 8);
    layout->addWidget(acceptBtn, 1, 2, 1, 2);
    layout->addWidget(rejectBtn, 1, 4, 1, 2);
}

void ApplyItem::active()
{
    LOG() << "ApplyItem active.";
    // 这里不需要做额外的操作, 空函数即可.
}
```

实现聊天界面

界面效果:

- 点击右上角 `...` 按钮, 打开新的窗口, 显示这个消息会话的详情.
- 点击用户头像, 打开新窗口, 显示用户详细信息.
- 左侧下方四个按钮, 分别是发送图片, 发送文件, 发送语音, 查看历史消息 (打开新的窗口)



1) 实现会话标题栏

回到 `MainWidget`

```
1  void MainWidget::initWindowRight()
2  {
3      QVBoxLayout* layout = new QVBoxLayout();
4      layout->setContentsMargins(0, 0, 0, 0);
5      layout->setSpacing(0);
6      windowRight->setLayout(layout);
7
8      ////////////////////////////////////////////////////
9      /// 初始化右侧上方，会话标题区
10     ////////////////////////////////////////////////////
```

```cpp
    QWidget* widget = new QWidget();
    widget->setObjectName("titleWidget");
    widget->setFixedHeight(62);
    widget->setStyleSheet("#titleWidget { border-bottom: 1px solid rgb(231, 231, 231); border-left: 1px solid rgb(231, 231, 231); }");
    layout->addWidget(widget, 0, Qt::AlignTop);

    QHBoxLayout* hlayout = new QHBoxLayout();
    hlayout->setContentsMargins(20, 0, 20, 0);
    widget->setLayout(hlayout);

    sessionTitle = new QLabel();
    sessionTitle->setStyleSheet("QLabel { font-size: 22px; }");
#if TEST_UI
    sessionTitle->setText("这里是会话标题");
#endif
    hlayout->addWidget(sessionTitle);

    extraButton = new QPushButton();
    extraButton->setIcon(QIcon(":/image/more.png"));
    extraButton->setFixedSize(30, 30);
    extraButton->setStyleSheet("QPushButton { background-color: rgb(245, 245, 245); border: none;} QPushButton:pressed { background-color: rgb(222, 222, 223); }");
    extraButton->setCursor(Qt::PointingHandCursor);
    hlayout->addWidget(extraButton);


    ////////////////////////////////////////////////////
    /// 初始化右侧中间, 消息显示区
    ////////////////////////////////////////////////////
    messageShowArea = new MessageShowArea();
    layout->addWidget(messageShowArea);
#if TEST_UI
    Message* message = new Message();
    message->sender.avatar = QIcon(":/image/defaultAvatar.png");
    message->sender.nickname = "张三";
    message->time = "2024-04-16 17:00";
    message->messageType = MessageType::TEXT_TYPE;
    QString text = "今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥今晚吃啥";
    message->content = text.toUtf8();

    messageShowArea->addMessage(false, *message);
    for (int i = 0; i < 30; i++) {
        Message* message = new Message();
```

```
        message->sender.avatar = QIcon(":/image/defaultAvatar.png");
        message->sender.nickname = "张三" + QString::number(i);
        message->time = "2024-04-16 17:00";
        message->messageType = MessageType::TEXT_TYPE;
        QString text = "今晚吃啥";
        message->content = text.toUtf8();
        messageShowArea->addMessage(true, *message);
    }
#endif

    ////////////////////////////////////////////////////
    /// 初始化右侧下方，消息编辑区
    ////////////////////////////////////////////////////
    messageEditArea = new MessageEditArea(this);
    layout->addWidget(messageEditArea, 0, Qt::AlignBottom);
}
```

2) 实现消息展示区域

1) 创建 `MessageShowArea`

```
class MessageShowArea : public QScrollArea
{
    Q_OBJECT
public:
    explicit MessageShowArea(QWidget *parent = nullptr);

    // 新增一个 Message (尾插)
    void addMessage(bool isLeft, const Message& message);
    // 新增一个 Message (头插)
    void addFrontMessage(bool isLeft, const Message& message);

    // 清空所有 Message
    void clear();

    // 设置滚动条滚动到末尾
    void scrollToEndLater();

signals:

private:
    QWidget* container;
};
```

```cpp
MessageShowArea::MessageShowArea(QWidget *parent)
    : QScrollArea{parent}
{
    // 1. 初始化基础属性
    this->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
    // 一定要添加这个设置，否则无法正确显示.
    this->setWidgetResizable(true);
    // 隐藏水平滚动条. 把垂直滚动条设置的细一些.
    this->verticalScrollBar()->setStyleSheet("QScrollBar:vertical { width: 2px; background-color: rgb(240, 240, 240); } QScrollBar::handle:vertical {background-color: rgb(46, 46, 46);}");

    this->horizontalScrollBar()->setStyleSheet("QScrollBar:horizontal { height: 0px;}");
    this->setStyleSheet("QScrollArea { border: none;}");

    // 2. 创建核心 Widget
    //    此处设置个最小宽度. 而不设置固定宽度
    container = new QWidget();
    this->setWidget(container);

    // 3. 给核心 Widget 添加布局
    QVBoxLayout* layout = new QVBoxLayout();
    layout->setSpacing(0);
    layout->setContentsMargins(0, 0, 0, 0);
    container->setLayout(layout);
}

void MessageShowArea::addMessage(bool isLeft, const Message& message)
{
    MessageItem* messageItem = MessageItem::makeMessageItem(isLeft, message);
    container->layout()->addWidget(messageItem);
}

void MessageShowArea::addFrontMessage(bool isLeft, const Message &message)
{
    MessageItem* messageItem = MessageItem::makeMessageItem(isLeft, message);
    QVBoxLayout* layout = dynamic_cast<QVBoxLayout*>(container->layout());
    layout->insertWidget(0, messageItem);
}

void MessageShowArea::clear()
{
    QLayout* layout = container->layout();
    // 从后往前遍历. 否则删除前面的元素会影响后面元素的下标
    for (int i = layout->count() - 1; i >= 0; i--) {
        QLayoutItem* item = layout->takeAt(i);
```

```
        if (item->widget()) {
            delete item->widget();
        }
        delete item;
    }
}

// 如果直接进行滚动到最后，可能得不到正确结果. 因为添加控件到布局管理器，触发界面大小 resize 是一个异步过程.
// 此处要个花招，通过定时器，延时 500ms 之后才真正进行滚动.
void MessageShowArea::scrollToEndLater()
{
    LOG() << "scrollToEndLater";
    QTimer* timer = new QTimer();
    connect(timer, &QTimer::timeout, this, [=]() {
        // 执行滚动操作
        int maxValue = this->verticalScrollBar()->maximum();
        this->verticalScrollBar()->setValue(maxValue);

        // 销毁定时器
        timer->stop();
        timer->deleteLater();
    });

    timer->start(500);
}
```

2) 创建消息对象 `MessageItem`

```
// 表示一个消息条目
class MessageItem : public QWidget
{
    Q_OBJECT
public:
    MessageItem(bool isLeft);

    // 通过工厂设计模式构造 Message
    static MessageItem* makeMessageItem(bool isLeft, const Message& message);

private:
    static QWidget* makeTextMessageItem(const QByteArray& content, bool isLeft);
    static QWidget* makeImageMessageItem(const QString& fileId, const QByteArray& content, bool isLeft);
```

```
    static QWidget* makeFileMessageItem(const QString& fileId, const QByteArray& content, bool isLeft, const QString& fileName);
    static QWidget* makeSoundMessageItem(const QString& fileId, const QByteArray& content, bool isLeft);

    bool isLeft;
};
```

```
MessageItem::MessageItem(bool isLeft) : isLeft(isLeft)
{

}

MessageItem* MessageItem::makeMessageItem(bool isLeft, const Message& message)
{
    // 1. 创建基本骨架
    MessageItem* messageItem = new MessageItem(isLeft);
    QGridLayout* layout = new QGridLayout();
    layout->setContentsMargins(40, 10, 40, 0);
    layout->setHorizontalSpacing(10);
    layout->setVerticalSpacing(10);
    // layout->setColumnStretch(0, 0);
    // layout->setColumnStretch(1, 10);
    messageItem->setMinimumHeight(100);
    messageItem->setLayout(layout);

    // 2. 创建头像
    QPushButton* avatarBtn = new QPushButton();
    avatarBtn->setFixedSize(40, 40);
    avatarBtn->setIconSize(QSize(40, 40));
    avatarBtn->setIcon(message.sender.avatar);
    if (isLeft) {
        layout->addWidget(avatarBtn, 0, 0, 2, 1, Qt::AlignTop | Qt::AlignLeft);
    } else {
        layout->addWidget(avatarBtn, 0, 1, 2, 1, Qt::AlignTop | Qt::AlignLeft);
    }

    // 3. 创建名字 和 时间
    QLabel* nameLabel = new QLabel();
    nameLabel->setText(message.sender.nickname + "  |  " + message.time);
    nameLabel->setAlignment(Qt::AlignBottom);
    nameLabel->setStyleSheet("QLabel { font-size: 12px; color: rgb(178, 178, 178); }");
    if (isLeft) {
        layout->addWidget(nameLabel, 0, 1);
```

```
    } else {
        layout->addWidget(nameLabel, 0, 0, Qt::AlignRight);
    }

    // 4. 创建消息体
    QWidget* contentWidget = nullptr;
    switch (message.messageType) {
    case TEXT_TYPE:
        contentWidget = makeTextMessageItem(message.content, isLeft);
        break;
    case IMAGE_TYPE:
        contentWidget = makeImageMessageItem(message.fileId, message.content,
isLeft);
        break;
    case FILE_TYPE:
        contentWidget = makeFileMessageItem(message.fileId, message.content,
isLeft, message.fileName);
        break;
    case SPEECH_TYPE:
        contentWidget = makeSoundMessageItem(message.fileId, message.content,
isLeft);
        break;
    default:
        LOG() << "error messageType! messageType = " << message.messageType;
    }
    if (isLeft) {
        layout->addWidget(contentWidget, 1, 1);
    } else {
        layout->addWidget(contentWidget, 1, 0);
    }

    // 5. 关联信号槽
    // 5.1 点击头像，弹出窗口显示用户信息.
    connect(avatarBtn, &QPushButton::clicked, messageItem, [=]() {
        UserInfoWidget* userInfoWidget = new UserInfoWidget(message);
        userInfoWidget->show();
    });

    // 5.2 用户修改了昵称，则修改昵称内容. TODO

    return messageItem;
}
```

3) 实现工厂方法

```
QWidget* MessageItem::makeTextMessageItem(const QByteArray &content, bool isLeft)
{
    // 文本消息不需要 fileId
    MessageContentLabel* item = new MessageContentLabel(TEXT_TYPE, QString(content), "", content, isLeft);
    return item;
}

QWidget* MessageItem::makeImageMessageItem(const QString& fileId, const QByteArray &content, bool isLeft)
{
    MessageImageLabel* item = new MessageImageLabel(fileId, content, isLeft);
    return item;
}

QWidget* MessageItem::makeFileMessageItem(const QString& fileId, const QByteArray &content, bool isLeft, const QString& fileName)
{
    MessageContentLabel* item = new MessageContentLabel(FILE_TYPE, "[文件] " + fileName, fileId, content, isLeft);
    return item;
}

QWidget* MessageItem::makeSoundMessageItem(const QString& fileId, const QByteArray &content, bool isLeft)
{
    MessageContentLabel* item = new MessageContentLabel(SPEECH_TYPE, "[语音]", fileId, content, isLeft);
    return item;
}
```

4) 创建文本消息类. 同时也可以用作显示文件消息和语音消息.

```
class MessageContentLabel : public QWidget
{
public:
    MessageContentLabel(MessageType type, const QString& text, const QString& fileId, const QByteArray& content, bool isLeft);

    void paintEvent(QPaintEvent* ) override;
    void mousePressEvent(QMouseEvent * ) override;
    void contextMenuEvent(QContextMenuEvent* event) override;

```

```
private:
    QLabel* label;
    // 保存消息类型
    MessageType messageType;
    // 文件id
    QString fileId;
    // 保存消息内容(比如文件消息，语音消息的二进制内容)，为后续的 "播放" / "另存为" 做准备
    QByteArray content;
    // 是否是左侧消息
    bool isLeft;
};
```

```
MessageContentLabel::MessageContentLabel(MessageType messageType, const QString& text, const QString& fileId, const QByteArray& content, bool isLeft)
    : messageType(messageType), fileId(fileId), content(content), isLeft(isLeft)
{
    this->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);

    QFont font;
    font.setFamily("微软雅黑");
    font.setPixelSize(16);

    this->label = new QLabel(this);
    this->label->setText(text);
    this->label->setFont(font);
    this->label->setAlignment(Qt::AlignVCenter | Qt::AlignLeft);
    this->label->setWordWrap(true);
    this->label->setStyleSheet("QLabel { padding: 0 10px; line-height: 1.2; }");
}

void MessageContentLabel::paintEvent(QPaintEvent *)
{
    // 1. 拿到父元素的宽度
    QObject* object = this->parent();
    if (!object->isWidgetType()) {
        return;
    }
    QWidget* parent = dynamic_cast<QWidget*>(object);
    // 宽度设置为父元素的 60%
    int width = parent->width() * 0.6;

    // 2. 计算文本的行数，进一步得到高度
```

```
30      QFontMetrics metrics(this->label->font());
31      // 显示这些文本一共需要多宽
32      int totalWidth = metrics.horizontalAdvance(this->label->text());
33      int rows = (totalWidth / (width - 40)) + 1; // - 40 是给左右两侧留下足够的空隙
34      if (totalWidth + 40 < width) {
35          // 文本只有一行，且不足一行，就直接调整成更小的宽度
36          width = totalWidth + 40;
37      }
38      int height = rows * (this->label->font().pixelSize() * 1.2) + 20; // + 20 是
    给上下留下点空白控件，1.2 是考虑到行间距的问题
39      // LOG() << "width: " << width << ", height: " << height << ", totalWidth:
    " << totalWidth << ", rows: " << rows;
40
41      // 4. 绘制圆角矩形
42      QPainter painter(this);
43      // 设置抗锯齿
44      painter.setRenderHint(QPainter::Antialiasing);
45      // 在这里画个小箭头 + 矩形
46      QPainterPath path;
47      if (isLeft) {
48          painter.setPen(QPen(QColor(255, 255, 255), 1));
49          painter.setBrush(QColor(255, 255, 255));
50
51          path.addRoundedRect(10, 0, width, height, 10, 10);
52          path.moveTo(10, 15);
53          path.lineTo(0, 20);
54          path.lineTo(10, 25);
55          path.closeSubpath();
56
57          painter.drawPath(path);
58
59          // 5. 把 QLabel 移动到合适的位置
60          this->label->setGeometry(10, 0, width, height);
61      } else {
62          painter.setPen(QPen(QColor(137, 217, 97), 1));
63          painter.setBrush(QColor(137, 217, 97));
64
65          // x 轴的起始位置. 此处用自身的 宽度 减去 label 的宽度，再往左挪 10px
66          int leftPos = this->width() - width - 10;
67          int rightPos = this->width() - 10;
68          path.addRoundedRect(leftPos, 0, width, height, 10, 10);
69          path.moveTo(rightPos, 15);
70          path.lineTo(rightPos + 10, 20);
71          path.lineTo(rightPos, 25);
72          path.closeSubpath();
73
74          painter.drawPath(path);
```

```
75
76          // 5. 把 QLabel 移动到合适的位置
77          this->label->setGeometry(leftPos, 0, width, height);
78      }
79
80      // 6. 设置父元素的高度，确保父元素能够把当前整个文本都显示完整.
81      //    50 这个数字是拍脑门拍的. 根据实际情况微调
82      parent->setFixedHeight(height + 50);
83  }
84
85  void MessageContentLabel::mousePressEvent(QMouseEvent *event)
86  {
87      // TODO
88  }
89
90  void MessageContentLabel::contextMenuEvent(QContextMenuEvent *event)
91  {
92      // TODO
93  }
```

5) 图片消息的表示, 后面再实现.

```
1  class MessageImageLabel : public QWidget
2  {
3      Q_OBJECT
4
5      // TODO
6  };
```

3) 实现消息编辑区域

创建 `MessageEditArea`

```
1  class MessageEditArea : public QWidget
2  {
3      Q_OBJECT
4  public:
5      explicit MessageEditArea(QWidget* owner);
6
7      // 消息输入框
```

```
    QPlainTextEdit* textEdit;
    // 提示信息 label
    QLabel* tipLabel;
    // 发送消息按钮
    QPushButton* sendBtn;
    // 发送图片消息
    QPushButton* sendImageBtn;
    // 发送文件消息
    QPushButton* sendFileBtn;
    // 显示历史消息按钮
    QPushButton* showHistoryBtn;
    // 发送语音按钮
    QPushButton* sendSoundBtn;

private:
    // 被哪个窗口持有. 指向 MainWidget 的指针
    QWidget* owner;

signals:
};
```

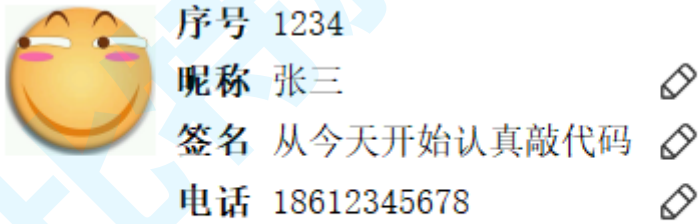
```
MessageEditArea::MessageEditArea(QWidget *owner) : owner(owner)
{
    // 1. 设置基础属性
    this->setFixedHeight(280);
    this->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);

    // 2. 创建纵向布局管理器
    QVBoxLayout* layout = new QVBoxLayout();
    layout->setContentsMargins(10, 0, 10, 10);
    layout->setSpacing(0);
    layout->setAlignment(Qt::AlignTop);
    this->setLayout(layout);

    // 3. 创建工具按钮容器 Widget
    QWidget* toolsContainer = new QWidget();
    toolsContainer->setFixedHeight(35);
    layout->addWidget(toolsContainer);

    // 3.1 创建工具按钮的布局管理器.
    QHBoxLayout* hlayout = new QHBoxLayout();
    hlayout->setContentsMargins(10, 0, 0, 0);
    hlayout->setSpacing(0);
    hlayout->setAlignment(Qt::AlignLeft);
    toolsContainer->setLayout(hlayout);
```

```
25
26      // 3.2 创建工具按钮
27      QString buttonStyle = "QPushButton { background-color: rgb(245, 245, 245);
    border: none; } QPushButton:pressed { background-color: rgb(255, 255, 255);}";
28      QSize buttonSize(35, 35);
29      QSize iconSize(25, 25);
30
31      sendImageBtn = new QPushButton();
32      sendImageBtn->setFixedSize(buttonSize);
33      sendImageBtn->setIconSize(iconSize);
34      sendImageBtn->setIcon(QIcon(":/image/image.png"));
35      sendImageBtn->setStyleSheet(buttonStyle);
36      hlayout->addWidget(sendImageBtn);
37
38      sendFileBtn = new QPushButton();
39      sendFileBtn->setFixedSize(buttonSize);
40      sendFileBtn->setIconSize(iconSize);
41      sendFileBtn->setIcon(QIcon(":/image/file.png"));
42      sendFileBtn->setStyleSheet(buttonStyle);
43      hlayout->addWidget(sendFileBtn);
44
45      sendSoundBtn = new QPushButton();
46      sendSoundBtn->setFixedSize(buttonSize);
47      sendSoundBtn->setIconSize(iconSize);
48      sendSoundBtn->setIcon(QIcon(":/image/sound.png"));
49      sendSoundBtn->setStyleSheet(buttonStyle);
50      hlayout->addWidget(sendSoundBtn);
51
52      showHistoryBtn = new QPushButton();
53      showHistoryBtn->setFixedSize(buttonSize);
54      showHistoryBtn->setIconSize(iconSize);
55      showHistoryBtn->setIcon(QIcon(":/image/history.png"));
56      showHistoryBtn->setStyleSheet(buttonStyle);
57      hlayout->addWidget(showHistoryBtn);
58
59      // 4. 创建编辑框
60      textEdit = new QPlainTextEdit();
61      textEdit->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
62      textEdit->setStyleSheet("QPlainTextEdit { border: none; background-color:
    transparent; font-size: 14px; padding: 10px; } ");
63      textEdit->verticalScrollBar()->setStyleSheet("QScrollBar:vertical { width:
    2px; background-color: rgb(46, 46, 46); }");
64      layout->addWidget(textEdit);
65
66      // 5. 创建一个提示信息 label, 默认状态为隐藏
67      tipLabel = new QLabel();
68      tipLabel->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
```

```
    tipLabel->setText("录音中...");
    tipLabel->setAlignment(Qt::AlignCenter);
    tipLabel->setFont(QFont("微软雅黑", 24, 600));
    layout->addWidget(tipLabel);
    tipLabel->hide();

    // 6. 创建发送按钮
    sendBtn = new QPushButton();
    sendBtn->setText("发送");
    sendBtn->setFixedSize(120, 40);
    QString style = "QPushButton { font-size: 16px; color: rgb(7, 193, 96); border: none; background-color: rgb(233, 233, 233); border-radius: 10px; }";
    style += "QPushButton:hover { background-color: rgb(210, 210, 210); } ";
    style += "QPushButton:pressed { background-color: rgb(198, 198, 198);} ";
    sendBtn->setStyleSheet(style);
    layout->addWidget(sendBtn, 0, Qt::AlignRight | Qt::AlignVCenter);
}
```

实现个人信息详细界面

点击自己头像时打开的界面.



创建 `SelfInfoWidget` 实现个人信息窗口.

```
// 用来显示/修改个人信息的窗口
class SelfInfoWidget : public QDialog
{
    Q_OBJECT
public:
    explicit SelfInfoWidget(QWidget *parent = nullptr);

signals:

private:
```

```
    // 头像
    QPushButton* avatar;
    // 名字
    QLineEdit* nameEdit;
    QLabel* nameLabel;
    QPushButton* nameModifyBtn;
    QPushButton* nameSubmitBtn;
    // 签名
    QLineEdit* descEdit;
    QLabel* descLabel;
    QPushButton* descModifyBtn;
    QPushButton* descSubmitBtn;
    // 电话
    QLineEdit* phoneEdit;
    QLabel* phoneLabel;
    QPushButton* phoneModifyBtn;
    QPushButton* phoneSubmitBtn;
    QLineEdit* verityCodeEdit;
    QPushButton* sendVerityCodeBtn;

    // 布局管理器
    QGridLayout* gridLayout;

    // 控制发送验证码按钮的倒计时
    int leftTime = 0;

    // 即将要修改的新的手机号.
    // 这里要通过这个成员来记录一下.
    // 确保发送验证码的手机, 和最终修改的手机是同一个手机号码.
    QString phoneToChange;
};
```

```
SelfInfoWidget::SelfInfoWidget(QWidget *parent) : QDialog(parent)
{
    // 1. 设置基础属性
    this->setFixedSize(450, 250);
    this->setWindowIcon(QIcon(":/image/logo.png"));
    this->setWindowTitle("个人信息");
    // 隐藏标题栏
    // this->setWindowFlags(Qt::FramelessWindowHint);
    // 设置关闭后自动销毁
    this->setAttribute(Qt::WA_DeleteOnClose);
    // 设置窗口显示位置为鼠标所在位置.
    this->move(QCursor::pos());
    // 设置背景色
```

```
14      this->setStyleSheet("QWidget { background-color: rgb(255, 255, 255); font-
    size: 18px;} QPushButton { border: none; } QPushButton:pressed {background-
    color: rgb(235, 235, 235);}");
15
16      // 2. 创建界面布局
17      QHBoxLayout* layout = new QHBoxLayout();
18      layout->setContentsMargins(30, 20, 30, 20);
19      layout->setSpacing(10);
20      this->setLayout(layout);
21
22      // 3. 添加用户头像
23      avatar = new QPushButton();
24      avatar->setFixedSize(75, 75);
25      avatar->setIconSize(QSize(75, 75));
26      avatar->setStyleSheet("QPushButton { border: none; background-color:
    white; }");
27      layout->addWidget(avatar, 0, Qt::AlignTop);
28
29      // 4. 添加用户其他信息
30      //    这里实现的效果是, 使用一个 N 行 4 列的 gridLayout 表示布局
31      //    第 0 列是一个 QLabel
32      //    第 1 列是一个 QLineEdit
33      //    第 2 列是一个 编辑按钮
34      //    第 3 列是一个 确认按钮
35      //    界面默认是显示 QLabel 和 编辑按钮.
36      //    点击编辑之后, 就会切换成显示 QLineEdit 和 确认按钮.
37      //    点击确认之后, 又会切换回 QLabel 和 编辑按钮.
38      //    (通过 hide 和 show 操作来切换).
39      gridLayout = new QGridLayout();
40      gridLayout->setContentsMargins(0, 0, 0, 0);
41      gridLayout->setSpacing(10);
42      layout->addLayout(gridLayout);
43
44      // 作为下方输入框的宽度
45      QSize size(200, 30);
46
47      // 4.1 用户 id (不能编辑, 无需切换)
48      QLabel* idTag = new QLabel(this);
49      idTag->setStyleSheet("QLabel { font-weight: 800; }");
50      idTag->setText("序号");
51      QLabel* idLabel = new QLabel(this);
52
53      QIcon modifyIcon(":/image/modify.png");
54      QIcon submitIcon(":/image/submit.png");
55
56      // 4.2 用户昵称
57      QLabel* nameTag = new QLabel(this);
```

```
58      nameTag->setText("昵称");
59      nameTag->setStyleSheet("QLabel { font-weight: 800; }");
60      nameLabel = new QLabel(this);
61      nameLabel->setFixedSize(size);
62      nameEdit = new QLineEdit(this);
63      nameEdit->setFixedSize(size);
64      nameModifyBtn = new QPushButton(this);
65      nameModifyBtn->setIcon(modifyIcon);
66      nameSubmitBtn = new QPushButton(this);
67      nameSubmitBtn->setIcon(submitIcon);
68
69      nameEdit->hide();
70      nameSubmitBtn->hide();
71
72      // 4.3 用户签名
73      QLabel* descTag = new QLabel(this);
74      descTag->setText("签名");
75      descTag->setStyleSheet("QLabel { font-weight: 800; }");
76      descLabel = new QLabel(this);
77      descLabel->setFixedSize(size);
78      descEdit = new QLineEdit(this);
79      descEdit->setFixedSize(size);
80      descModifyBtn = new QPushButton(this);
81      descModifyBtn->setIcon(modifyIcon);
82      descSubmitBtn = new QPushButton(this);
83      descSubmitBtn->setIcon(submitIcon);
84
85      descEdit->hide();
86      descSubmitBtn->hide();
87
88      // 4.4 用户电话
89      QLabel* phoneTag = new QLabel(this);
90      phoneTag->setText("电话");
91      phoneTag->setStyleSheet("QLabel { font-weight: 800; }");
92      phoneLabel = new QLabel(this);
93      phoneLabel->setFixedSize(size);
94      phoneEdit = new QLineEdit(this);
95      phoneEdit->setFixedSize(size);
96      phoneModifyBtn = new QPushButton(this);
97      phoneModifyBtn->setIcon(modifyIcon);
98      phoneSubmitBtn = new QPushButton(this);
99      phoneSubmitBtn->setIcon(submitIcon);
100
101     phoneEdit->hide();
102     phoneSubmitBtn->hide();
103
104     verityCodeEdit = new QLineEdit(this);
```

```
105     verityCodeEdit->setFixedSize(size);
106     verityCodeEdit->setPlaceholderText("验证码");
107     verityCodeEdit->hide();
108     // 通过这个按钮来显示发送间隔时间的倒计时.
109     sendVerityCodeBtn = new QPushButton(this);
110     sendVerityCodeBtn->setText("获取");
111     sendVerityCodeBtn->hide();
112
113 #if TEST_UI
114     avatar->setIcon(QIcon(":/image/defaultAvatar.png"));
115     idLabel->setText("1234");
116     nameLabel->setText("张三");
117     descLabel->setText("从今天开始认真敲代码");
118     phoneLabel->setText("18612345678");
119 #endif
120
121     // 5. 添加到布局管理器中.
122     gridLayout->setRowStretch(0, 1);
123     gridLayout->setRowStretch(1, 1);
124     gridLayout->setRowStretch(2, 1);
125     gridLayout->setRowStretch(3, 1);
126     gridLayout->setRowStretch(4, 1);
127
128     gridLayout->addWidget(idTag, 0, 0);
129     gridLayout->addWidget(idLabel, 0, 1);
130
131     gridLayout->addWidget(nameTag, 1, 0);
132     gridLayout->addWidget(nameLabel, 1, 1);
133     gridLayout->addWidget(nameModifyBtn, 1, 2);
134
135     gridLayout->addWidget(descTag, 2, 0);
136     gridLayout->addWidget(descLabel, 2, 1);
137     gridLayout->addWidget(descModifyBtn, 2, 2);
138
139     gridLayout->addWidget(phoneTag, 3, 0);
140     gridLayout->addWidget(phoneLabel, 3, 1);
141     gridLayout->addWidget(phoneModifyBtn, 3, 2);
142
143     gridLayout->addWidget(sendVerityCodeBtn, 4, 0);
144     gridLayout->addWidget(verityCodeEdit, 4, 1);
145
146     // 6. 设置第一行的 SizePolicy
147     QSizePolicy sizePolicy(QSizePolicy::Preferred, QSizePolicy::Fixed);
148     idTag->setSizePolicy(sizePolicy);
149     idLabel->setSizePolicy(sizePolicy);
150
151 }
```

修改 `MainWidget::initData`, 添加弹出该窗口的逻辑

```
//////////////////////////////////////////
/// 点击用户自己头像, 显示出用户详细信息
//////////////////////////////////////////
connect(userAvatar, &QPushButton::clicked, this, [=]() {
    SelfInfoWidget* selfInfoWidget = new SelfInfoWidget();
    // 按照模态方式弹出对话框
    selfInfoWidget->exec();
});
```

实现用户详细信息界面

点击其他用户头像时打开的界面.

创建 `UserInfoWidget` 实现其他用户信息窗口.

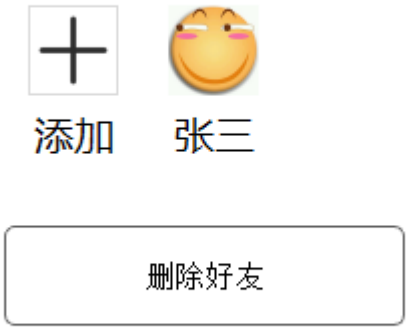
```
// 用于显示其他人的用户信息的窗口
class UserInfoWidget : public QDialog
{
    Q_OBJECT
public:
    explicit UserInfoWidget(const Message& message, QWidget *parent = nullptr);

private:
    const Message& message;

signals:

};
```

```cpp
UserInfoWidget::UserInfoWidget(const Message& message, QWidget *parent) : message(message), QDialog(parent)
{
    // 1. 设置基本属性
    this->setFixedSize(400, 200);
    this->setStyleSheet("QWidget {background-color: rgb(255, 255, 255);} QLabel {font-size: 18px;}");
    this->setWindowTitle("用户详情");
    this->setWindowIcon(QIcon(":/image/logo.png"));
    // 隐藏标题栏
    // this->setWindowFlags(Qt::FramelessWindowHint);
    // 设置关闭后自动销毁
    this->setAttribute(Qt::WA_DeleteOnClose);
    // 设置窗口显示位置为鼠标所在位置.
    this->move(QCursor::pos());

    // 2. 设置主布局管理器
    QGridLayout* layout = new QGridLayout();
    layout->setContentsMargins(50, 30, 50, 30);
    layout->setSpacing(10);
    this->setLayout(layout);

    // 3. 设置头像
    QPushButton* avatar = new QPushButton();
    avatar->setFixedSize(75, 75);
    avatar->setIconSize(QSize(75, 75));
    avatar->setIcon(message.sender.avatar);
    avatar->setStyleSheet("QPushButton { border: none; background-color: white; }");

    // 4. 设置用户序号
    QLabel* idTag = new QLabel();
    idTag->setText("序号");
    idTag->setAlignment(Qt::AlignCenter);
    idTag->setStyleSheet("QLabel { font-weight: 800; }");
    QLabel* idLabel = new QLabel();
    idLabel->setText(message.sender.userId);

    // 5. 设置昵称
    QLabel* nameTag = new QLabel();
    nameTag->setText("昵称");
    nameTag->setAlignment(Qt::AlignCenter);
    nameTag->setStyleSheet("QLabel { font-weight: 800; }");
    QLabel* nameLabel = new QLabel();
```

```
42      nameLabel->setText(message.sender.nickname);
43
44      // 6. 设置电话
45      QLabel* phoneTag = new QLabel();
46      phoneTag->setText("电话");
47      phoneTag->setAlignment(Qt::AlignCenter);
48      phoneTag->setStyleSheet("QLabel { font-weight: 800; }");
49      QLabel* phoneLabel = new QLabel();
50      phoneLabel->setText(message.sender.phone);
51
52      // 7. 设置底部按钮(申请添加好友, 发送信息, 删除好友)
53      QString btnStyle = "QPushButton { border: 1px solid rgb(90, 90, 90);
    border-radius: 5px; } QPushButton:pressed { background-color: rgb(235, 235,
    235);}";
54      QPushButton* addFriendBtn = new QPushButton();
55      addFriendBtn->setStyleSheet(btnStyle);
56      addFriendBtn->setFixedSize(80, 30);
57      addFriendBtn->setText("申请好友");
58
59      QPushButton* sendMessageBtn = new QPushButton();
60      sendMessageBtn->setText("发送消息");
61      sendMessageBtn->setFixedSize(80, 30);
62      sendMessageBtn->setStyleSheet(btnStyle);
63
64      QPushButton* deleteFriendBtn = new QPushButton();
65      deleteFriendBtn->setText("删除好友");
66      deleteFriendBtn->setFixedSize(80, 30);
67      deleteFriendBtn->setStyleSheet(btnStyle);
68
69      // 8. 添加到布局管理器中
70      layout->addWidget(avatar, 0, 0, 3, 2);
71
72      layout->addWidget(idTag, 0, 2, 1, 2);
73      layout->addWidget(idLabel, 0, 4, 1, 2);
74
75      layout->addWidget(nameTag, 1, 2, 1, 2);
76      layout->addWidget(nameLabel, 1, 4, 1, 2);
77
78      layout->addWidget(phoneTag, 2, 2, 1, 2);
79      layout->addWidget(phoneLabel, 2, 4, 1, 3);
80
81      layout->addWidget(addFriendBtn, 3, 0, 1, 2);
82      layout->addWidget(sendMessageBtn, 3, 2, 1, 2);
83      layout->addWidget(deleteFriendBtn, 3, 4, 1, 2);
84  }
```

实现单聊消息会话详细信息界面

点击单聊 `...` 时打开的界面.



1) 实现会话详情窗口

创建 `SessionDetailWidget`

```
// 单聊消息会话的详情窗口
class SessionDetailWidget : public QDialog
{
    Q_OBJECT
public:
    explicit SessionDetailWidget(const ChatSessionInfo& chatSessionInfo);

signals:

private:
    const ChatSessionInfo& chatSessionInfo;
};
```

```
SessionDetailWidget::SessionDetailWidget(const ChatSessionInfo&
chatSessionInfo)
    : chatSessionInfo(chatSessionInfo)
{
    // 1. 设置基本属性
    this->setWindowTitle("会话详情");
    this->setAttribute(Qt::WA_DeleteOnClose);
```

```
    this->setWindowIcon(QIcon(":/image/logo.png"));
    this->setFixedSize(300, 300);
    this->setStyleSheet("QWidget {background-color: rgb(255, 255, 255); }");
    // 隐藏标题栏
    // this->setWindowFlags(Qt::FramelessWindowHint);

    // 2. 创建核心布局
    QGridLayout* layout = new QGridLayout();
    layout->setContentsMargins(50, 50, 50, 50);
    layout->setSpacing(10);
    layout->setAlignment(Qt::AlignTop | Qt::AlignLeft);
    this->setLayout(layout);

    // 3. 添加 "添加按钮"
    AvatarItem* addBtn = new AvatarItem(QIcon(":/image/cross.png"), "添加");
    layout->addWidget(addBtn, 0, 0);

    // 4. 添加当前用户头像
#if TEST_UI
    AvatarItem* currentUser = new AvatarItem(QIcon(":/image/defaultAvatar.png"), "张三");
    layout->addWidget(currentUser, 0, 1);
#endif

    // 5. 添加空白
    QSpacerItem* spacerItem = new QSpacerItem(70, 70);
    layout->addItem(spacerItem, 0, 2);

    // 6. 添加 删除好友 按钮
    QPushButton* deleteFriendBtn = new QPushButton();
    deleteFriendBtn->setFixedHeight(50);
    deleteFriendBtn->setText("删除好友");
    deleteFriendBtn->setStyleSheet("QPushButton { border: 1px solid rgb(90, 90, 90); border-radius: 5px; } QPushButton:pressed { background-color: rgb(235, 235, 235); }");
    layout->addWidget(deleteFriendBtn, 1, 0, 1, 3);
}
```

2) 创建 头像+昵称 组合控件

创建 `AvatarItem` 类

```
class AvatarItem : public QWidget {
    Q_OBJECT
```

```
3
4  public:
5      AvatarItem(const QIcon& avatar, const QString& name);
6
7      void setClicked(std::function<void(void)> slotFunc);
8
9  private:
10     QPushButton* avatarBtn;
11     QLabel* nameLabel;
12 };
```

```
1  AvatarItem::AvatarItem(const QIcon &avatar, const QString &name)
2  {
3      // 1. 设置基本属性
4      this->setFixedSize(70, 80);
5
6      // 2. 创建核心布局
7      QVBoxLayout* layout = new QVBoxLayout();
8      layout->setContentsMargins(0, 0, 0, 0);
9      layout->setSpacing(5);
10     this->setLayout(layout);
11
12     // 3. 创建头像. 使用按钮表示. 方便后续实现点击操作.
13     avatarBtn = new QPushButton(this);
14     avatarBtn->setIcon(avatar);
15     avatarBtn->setFixedSize(45, 45);
16     avatarBtn->setIconSize(QSize(45, 45));
17     avatarBtn->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);
18     avatarBtn->setStyleSheet("QPushButton {border: 1px solid rgb(218, 218,
   218);}");
19
20     // 4. 创建名字
21     nameLabel = new QLabel();
22     QFont font("微软雅黑", 12);
23     nameLabel->setFont(font);
24     nameLabel->setAlignment(Qt::AlignCenter);
25     nameLabel->setText(name);
26
27     // 5. 名字截断逻辑
28     QFontMetrics metrics(font);
29     int totalWidth = metrics.horizontalAdvance(name);
30     const int MAX_WIDTH = 65;
31     if (totalWidth >= MAX_WIDTH) {
32         // 宽度太长了, 截断名字
33         QString ellipsis = "...";
```

```
        int ellipsisWidth = metrics.horizontalAdvance(ellipsis);
        int availableWidth = MAX_WIDTH - ellipsisWidth;
        // 取左侧 n 个字符
        QString truncatedName = name.left(name.length() * availableWidth / totalWidth);
        nameLabel->setText(truncatedName + ellipsis);
    }

    // 6. 添加到布局管理器中
    layout->addWidget(avatarBtn, 0, Qt::AlignCenter);
    layout->addWidget(nameLabel, 0, Qt::AlignCenter);
}
```

3) 实现弹出对话框

在 `MainWidget::initData` 中

```
connect(extraButton, &QPushButton::clicked, this, [=]() {
    // 测试阶段的代码
   #if TEST_GROUP_SESSION_DETAIL
        GroupSessionDetailWidget* groupSessionDetailWidget = new GroupSessionDetailWidget();
        groupSessionDetailWidget->show();
   #else
        SessionDetailWidget* sessionDetailWidget = new SessionDetailWidget();
        sessionDetailWidget->show();
   #endif
}
```

实现选择好友界面

点击 "添加按钮" 弹出对话框, 选择已有好友进入群聊.

1) 实现好友选择窗口

创建 `ChooseFriendDialog`

```
1  class ChooseFriendDialog : public QDialog
2  {
3      Q_OBJECT
4  public:
5      ChooseFriendDialog(const QString& userId, QWidget* parent = nullptr);
6
7      void addFriend(const QString& userId, const QIcon avatar, const QString&
   name, bool checked);
8      void addSelectedFriend(const QString& userId, const QIcon avatar, const
   QString& name);
9      void deleteSelectedFriend(const QString& userId);
10
11 private:
12     // 保存左侧待选中好友列表的 container
13     QWidget* totalContainer;
14     // 保存右侧已选中好友列表的 container
15     QWidget* selectedContainer;
16     // 保存弹出当前窗口的好友是哪个
```

```
    QString userId;

    void initLeft(QHBoxLayout* layout);
    void initRight(QHBoxLayout* layout);
};
```

```
ChooseFriendDialog::ChooseFriendDialog(const QString& userId, QWidget* parent)
    : QDialog(parent), userId(userId)
{
    // 1. 设置基本属性
    this->setWindowTitle("选择好友");
    this->setWindowIcon(QIcon(":/image/logo.png"));
    // 隐藏标题栏
    // this->setWindowFlags(Qt::FramelessWindowHint);
    this->setFixedSize(755, 550);
    this->setStyleSheet("QDialog {background-color: rgb(255, 255, 255);}");

    // 2. 创建核心布局(左右结构).
    QHBoxLayout* layout = new QHBoxLayout();
    layout->setContentsMargins(0, 0, 0, 0);
    layout->setSpacing(0);
    this->setLayout(layout);

    // 3. 创建左侧筛选好友列表
    initLeft(layout);

    // 4. 创建右侧已选中好友列表
    initRight(layout);
}
```

2) 实现筛选好友列表

```
// 初始化左侧区域
void ChooseFriendDialog::initLeft(QHBoxLayout *layout)
{
    // 1. 创建滚动区
    QScrollArea* scrollArea = new QScrollArea();
    scrollArea->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
    // 一定要添加这个设置, 否则无法正确显示.
    scrollArea->setWidgetResizable(true);
    // 隐藏水平滚动条. 把垂直滚动条设置的细一些.
```

```
    scrollArea->verticalScrollBar()->setStyleSheet("QScrollBar:vertical { width: 2px; background-color: rgb(255, 255, 255); } QScrollBar::handle:vertical {background-color: rgb(205, 205, 205);}");
    scrollArea->horizontalScrollBar()->setStyleSheet("QScrollBar:horizontal { height: 0px;}");
    scrollArea->setStyleSheet("QScrollArea { border: none; }");
    layout->addWidget(scrollArea, 1);

    totalContainer = new QWidget();
    totalContainer->setObjectName("totalContainer");
    totalContainer->setStyleSheet("#totalContainer {background-color: rgb(255, 255, 255);}");
    scrollArea->setWidget(totalContainer);

    // 2. 创建子布局管理器
    QVBoxLayout* vlayout = new QVBoxLayout();
    vlayout->setContentsMargins(0, 0, 0, 0);
    vlayout->setSpacing(0);
    vlayout->setAlignment(Qt::AlignTop);  // 设置靠上方对齐
    totalContainer->setLayout(vlayout);

    // 3. 添加所有待选择的好友
#if TEST_UI
    for (int i = 0; i < 30; i++) {
        this->addFriend("", QIcon(":/image/defaultAvatar.png"), "张三");
    }
#endif
}
```

```
void ChooseFriendDialog::addFriend(const QString& userId, const QIcon avatar, const QString &name, bool checked)
{
    ChooseFriendItem* item = new ChooseFriendItem(this, userId, avatar, name, checked);
    QVBoxLayout* layout = dynamic_cast<QVBoxLayout*>(totalContainer->layout());
    layout->addWidget(item);
}
```

3) 创建好友元素

创建 `ChooseFriendItem` 类

```cpp
class ChooseFriendItem : public QWidget
{
    Q_OBJECT
public:
    QCheckBox* checkBox;

    // 该对象的持有者
    ChooseFriendDialog* owner;
    QString userId;
    QIcon avatar;
    QString name;
    bool isHover;

    ChooseFriendItem(ChooseFriendDialog* owner, const QString& userId, const QIcon& avatar, const QString& name, bool checked = false);

    // 此处为了设置背景色，采取手绘的方式.
    void enterEvent(QEnterEvent *event) override;
    void leaveEvent(QEvent *event) override;
    void paintEvent(QPaintEvent *event) override;

private:
    void toggleCheckBox(bool checked);
};
```

```cpp
ChooseFriendItem::ChooseFriendItem(ChooseFriendDialog* owner, const QString& userId, const QIcon& avatar, const QString& name, bool checked)
    : owner(owner), userId(userId), avatar(avatar), name(name), isHover(false)
{
    this->setFixedHeight(50);
    this->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);

    QHBoxLayout* layout = new QHBoxLayout();
    layout->setContentsMargins(10, 5, 10, 5);
    layout->setSpacing(10);
    this->setLayout(layout);

    // 创建头像
    QPushButton* avatarBtn = new QPushButton();
    avatarBtn->setFixedSize(40, 40);
    avatarBtn->setIconSize(QSize(40, 40));
    avatarBtn->setIcon(avatar);

    // 创建名字
    QLabel* nameLabel = new QLabel();
```

```
20      nameLabel->setText(name);
21      nameLabel->setAlignment(Qt::AlignLeft | Qt::AlignVCenter);
22
23      // 创建复选框
24      checkBox = new QCheckBox();
25      checkBox->setFixedSize(25, 25);
26      checkBox->setChecked(checked);
27      checkBox->setStyleSheet("QCheckBox::indicator { width: 20px; height:20px;
    image: url(:/image/unchecked.png); } QCheckBox::indicator:checked { image:
    url(:/image/checked.png); }");
28      connect(checkBox, &QCheckBox::toggled, this,
    &ChooseFriendItem::toggleCheckBox);
29
30      // 添加到布局管理器中
31      layout->addWidget(checkBox);
32      layout->addWidget(avatarBtn);
33      layout->addWidget(nameLabel);
34  }
35
36  void ChooseFriendItem::enterEvent(QEnterEvent *event)
37  {
38      (void) event;
39      isHover = true;
40      repaint();
41  }
42
43  void ChooseFriendItem::leaveEvent(QEvent *event)
44  {
45      (void) event;
46      isHover = false;
47      repaint();
48  }
49
50  // 通过重写这个方法来实现背景色的设置.
51  void ChooseFriendItem::paintEvent(QPaintEvent *event)
52  {
53      (void) event;
54      QPainter painter(this);
55      if (isHover) {
56          painter.fillRect(rect(), QColor(231, 231, 231));
57      } else {
58          painter.fillRect(rect(), QColor(255, 255, 255));
59      }
60
61      // 如果直接使用 QSS 实现, 需要添加下列代码(来自于 Qt 官方文档)
62  //    QStyleOption opt;
63  //    opt.init(this);
```

```
//    QPainter p(this);
//    style()->drawPrimitive(QStyle::PE_Widget, &opt, &p, this);
}

void ChooseFriendItem::toggleCheckBox(bool checked)
{
    if (checked) {
        // 添加到右侧区域中
        owner->addSelectedFriend(userId, avatar, name);
    } else {
        // 从右侧区域中删除
        owner->deleteSelectedFriend(userId);
    }
}
```

4) 实现已选中好友列表

```
// 初始化右侧区域
void ChooseFriendDialog::initRight(QHBoxLayout *layout)
{
    // 1. 创建网格布局
    QGridLayout* gridLayout = new QGridLayout();
    gridLayout->setContentsMargins(20, 20, 20, 20);
    gridLayout->setSpacing(10);

    // 2. 创建顶部提示语
    QLabel* tipLabel = new QLabel();
    tipLabel->setText("选择联系人");
    tipLabel->setFixedHeight(30);
    tipLabel->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    tipLabel->setAlignment(Qt::AlignLeft | Qt::AlignVCenter);
    tipLabel->setStyleSheet("QLabel { font-size:16px; font-weight: 700; }");

    // 3. 创建中间滚动区
    QScrollArea* scrollArea = new QScrollArea();
    scrollArea->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
    // 一定要添加这个设置，否则无法正确显示.
    scrollArea->setWidgetResizable(true);
    // 隐藏水平滚动条. 把垂直滚动条设置的细一些.
    scrollArea->verticalScrollBar()->setStyleSheet("QScrollBar:vertical {
width: 2px; background-color: rgb(255, 255, 255); }
QScrollBar::handle:vertical {background-color: rgb(205, 205, 205);}");
```

```
24      scrollArea->horizontalScrollBar()->setStyleSheet("QScrollBar:horizontal {
    height: 0px;}");
25      scrollArea->setStyleSheet("QScrollArea { border: none; }");
26
27      selectedContainer = new QWidget();
28      selectedContainer->setObjectName("selectedContainer");
29      selectedContainer->setStyleSheet("#selectedContainer {background-color:
    rgb(255, 255, 255);}");
30      scrollArea->setWidget(selectedContainer);
31
32      QVBoxLayout* vlayout = new QVBoxLayout();
33      vlayout->setContentsMargins(0, 0, 0, 0);
34      vlayout->setSpacing(0);
35      vlayout->setAlignment(Qt::AlignTop);  // 设置靠上方对齐
36      selectedContainer->setLayout(vlayout);
37
38      // 4. 创建底部按钮
39      QString style = "QPushButton { color: rgb(7, 193, 96); background-color:
    rgb(243, 243, 243); border: none; border-radius: 5px;}";
40      style += "QPushButton:hover {background-color: rgb(219, 219, 219);}
    QPushButton:pressed {background-color: rgb(206, 206, 206);}";
41
42      QPushButton* okBtn = new QPushButton();
43      okBtn->setText("完成");
44      okBtn->setFixedHeight(40);
45      okBtn->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
46      okBtn->setStyleSheet(style);
47      // okBtn 的信号槽后续再处理.
48
49      QPushButton* cancelBtn = new QPushButton();
50      cancelBtn->setText("取消");
51      cancelBtn->setStyleSheet(style);
52      cancelBtn->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
53      cancelBtn->setFixedHeight(40);
54      connect(cancelBtn, &QPushButton::clicked, this, [=]() {
55          // 关闭对话框, 并返回 "QDialog::Rejected"
56          this->reject();
57      });
58
59      // 5. 放到布局中. 整体分成 9 个列. tipLabel 和 scrollArea 占满 9 列. 下面的按钮各
    自占 3 列. 剩下 3 列是左中右的空隙
60      gridLayout->addWidget(tipLabel, 0, 0, 1, 9);
61      gridLayout->addWidget(scrollArea, 1, 0, 1, 9);
62      gridLayout->addWidget(okBtn, 2, 1, 1, 3);
63      gridLayout->addWidget(cancelBtn, 2, 5, 1, 3);
64
65      layout->addLayout(gridLayout, 1);
```

```
66
67      // 6. 创建测试数据
68  #if TEST_UI
69      for (int i = 0; i < 30; i++) {
70          this->addSelectedFriend(QIcon(":/image/defaultAvatar.png"), "张三");
71      }
72  #endif
73  }
74
75  void ChooseFriendDialog::addSelectedFriend(const QString& userId, const QIcon
    avatar, const QString &name)
76  {
77      ChooseFriendItem* item = new ChooseFriendItem(this, userId, avatar, name,
    true);
78      QVBoxLayout* layout = dynamic_cast<QVBoxLayout*>(selectedContainer-
    >layout());
79      layout->addWidget(item);
80  }
81
82  void ChooseFriendDialog::deleteSelectedFriend(const QString &userId)
83  {
84      // 1. 删除已选中列表中的元素
85      QVBoxLayout* layoutSelected = dynamic_cast<QVBoxLayout*>(selectedContainer-
    >layout());
86      // 遍历 + 删除，需要从后往前遍历.
87      for (int i = layoutSelected->count() - 1; i >= 0; --i) {
88          auto* item = layoutSelected->itemAt(i);
89          if (item == nullptr || item->widget() == nullptr) {
90              continue;
91          }
92          ChooseFriendItem* chooseFriendItem = dynamic_cast<ChooseFriendItem*>
    (item->widget());
93          if (chooseFriendItem->userId != userId) {
94              continue;
95          }
96          // 从布局管理器中移除元素
97          layoutSelected->removeItem(item);
98          // 不要忘记释放内存
99          chooseFriendItem->deleteLater();
100     }
101     // 2. 确认好友列表中的 checkbox 的值为未选中状态.
102     //    这个逻辑用来处理只在右侧选中列表取消 checkBox 后，在左侧列表不能自动取消的
    bug
103     QVBoxLayout* layoutTotal = dynamic_cast<QVBoxLayout*>(totalContainer-
    >layout());
104     for (int i = 0; i < layoutTotal->count(); ++i) {
105         auto* item = layoutTotal->itemAt(i);
```

```
106         if (item == nullptr || item->widget() == nullptr) {
107             continue;
108         }
109         ChooseFriendItem* chooseFriendItem = dynamic_cast<ChooseFriendItem*>
    (item->widget());
110         if (chooseFriendItem->userId != userId) {
111             continue;
112         }
113         // 如果该选项的 checkBox 值修改为 false
114         chooseFriendItem->checkBox->setChecked(false);
115     }
116 }
```

实现群聊消息会话详细信息界面

点击群聊 `...` 时打开的界面.

创建 `GroupSessionDetailWidget` 实现群组会话详情窗口

```
class GroupSessionDetailWidget : public QDialog
{
    Q_OBJECT
public:
    explicit GroupSessionDetailWidget(QWidget *parent = nullptr);

    void addMember(AvatarItem* avatarItem);

signals:

private:
    // 保存当前群组会话中的成员列表
    QWidget* container;
    QLabel* groupNameLabel;

    // 记录当前成员放到了第几行，第几列
    int currentRow = 0;
    int currentCol = 0;
    const int MAX_COL = 4;
};
```

```
GroupSessionDetailWidget::GroupSessionDetailWidget(QWidget *parent) :
QDialog(parent)
{
    // 1. 设置基本属性
    this->setAttribute(Qt::WA_DeleteOnClose);
    this->setFixedSize(410, 600);
    this->setStyleSheet("QWidget {background-color: rgb(255, 255, 255); }");
    this->setWindowTitle("群组详情");
    this->setWindowIcon(QIcon(":/image/logo.png"));

    // 2. 创建布局管理器
    QVBoxLayout* layout = new QVBoxLayout();
    layout->setContentsMargins(50, 50, 50, 50);
    layout->setSpacing(10);
    this->setLayout(layout);

    // 3. 创建成员列表区
    QScrollArea* scrollArea = new QScrollArea();
    scrollArea->setFixedSize(310, 350);
    scrollArea->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    // 一定要添加这个设置，否则无法正确显示.
```

```
21      scrollArea->setWidgetResizable(true);
22      // 隐藏水平滚动条. 把垂直滚动条设置的细一些.
23      scrollArea->verticalScrollBar()->setStyleSheet("QScrollBar:vertical {
    width: 2px; background-color: rgb(255, 255, 255); }
    QScrollBar::handle:vertical {background-color: rgb(205, 205, 205);}");
24      scrollArea->horizontalScrollBar()->setStyleSheet("QScrollBar:horizontal {
    height: 0px;}");
25      scrollArea->setStyleSheet("QScrollArea { border: none;}");
26      layout->addWidget(scrollArea);
27
28      container = new QWidget();
29      scrollArea->setWidget(container);
30
31      QGridLayout* gridLayout = new QGridLayout();
32      gridLayout->setContentsMargins(0, 0, 0, 0);
33      gridLayout->setSpacing(10);
34      gridLayout->setAlignment(Qt::AlignTop | Qt::AlignLeft);
35      container->setLayout(gridLayout);
36
37      // 4. 添加邀请按钮
38      AvatarItem* addBtn = new AvatarItem(QIcon(":/image/cross.png"), "添加");
39      gridLayout->addWidget(addBtn, currentRow, currentCol);
40      currentCol++;
41
42      // 5. 添加群聊名称
43      QLabel* groupNameTag = new QLabel();
44      groupNameTag->setText("群聊名称");
45      groupNameTag->setStyleSheet("QLabel {font-weight: 700;}");
46      layout->addWidget(groupNameTag);
47
48      QHBoxLayout* hlayout = new QHBoxLayout();
49      groupNameLabel = new QLabel();
50      hlayout->addWidget(groupNameLabel);
51
52      QPushButton* modifyBtn = new QPushButton();
53      modifyBtn->setIcon(QIcon(":/image/modify.png"));
54      modifyBtn->setFixedSize(30, 30);
55      modifyBtn->setIconSize(QSize(20, 20));
56      modifyBtn->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);
57      modifyBtn->setStyleSheet("QPushButton { border: none; }
    QPushButton:pressed { background-color: rgb(235, 235, 235);}" );
58      hlayout->addWidget(modifyBtn);
59      layout->addLayout(hlayout);
60
61      // 6. 添加退群按钮
62      QPushButton* exitGroupBtn = new QPushButton();
63      exitGroupBtn->setText("退出群聊");
```

```
64      exitGroupBtn->setFixedSize(310, 50);
65      exitGroupBtn->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
66      exitGroupBtn->setStyleSheet("QPushButton { border: 1px solid rgb(90, 90,
    90); border-radius: 5px; } QPushButton:pressed { background-color: rgb(235,
    235, 235); }");
67      layout->addWidget(exitGroupBtn, 0, Qt::AlignCenter);
68
69      // 7. 添加群聊成员数据
70  #if TEST_UI
71      for (int i = 0; i < 20; i++) {
72          AvatarItem* avatarItem = new
    AvatarItem(QIcon(":/image/defaultAvatar.png"), "张三");
73          this->addMember(avatarItem);
74      }
75      groupNameLabel->setText("人类吃喝行为研究小组");
76  #endif
77  }
78
79  void GroupSessionDetailWidget::addMember(AvatarItem *avatarItem)
80  {
81      QGridLayout* layout = dynamic_cast<QGridLayout*>(this->container-
    >layout());
82
83      // 判定是否要换行
84      if (currentCol >= MAX_COL) {
85          currentCol = 0;
86          currentRow++;
87      }
88
89      layout->addWidget(avatarItem, currentRow, currentCol);
90      currentCol++;
91  }
```

实现添加好友界面

点击主界面上方的 "+" 按钮, 弹出添加好友界面.

1) 实现添加好友窗口

创建 `AddFriendDialog`

```
class AddFriendDialog : public QDialog
{
    Q_OBJECT
public:
    AddFriendDialog(QWidget* parent = nullptr);

    // 添加一个结果
    void addResult(const UserInfo& userInfo);

    // 清空结果
    void clear();

    // 设置输入框显示内容
    void setSearchKey(const QString& searchKey);

private:
    QLineEdit* searchEdit;
    // 保存搜索结果的容器
    QWidget* resultContainer;
```

```
20
21      void initResultArea(QGridLayout* layout);
22  };
```

```
1   AddFriendDialog::AddFriendDialog(QWidget* parent) : QDialog(parent)
2   {
3       // 1. 设置基本属性
4       this->setWindowTitle("添加好友");
5       this->setWindowIcon(QIcon(":/image/logo.png"));
6       this->setFixedSize(500, 500);
7       this->setStyleSheet("QDialog {background-color: rgb(255, 255, 255);}");
8
9       // 2. 设置核心布局
10      QGridLayout* layout = new QGridLayout();
11      layout->setContentsMargins(30, 30, 30, 30);
12      layout->setSpacing(10);
13      this->setLayout(layout);
14
15      // 3. 创建搜索框
16      searchEdit = new QLineEdit();
17      searchEdit->setFixedHeight(50);
18      searchEdit->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
19      searchEdit->setPlaceholderText("按手机号/用户序号查找");
20      searchEdit->setStyleSheet("QLineEdit { border-radius: 10px; background-
    color: rgb(243, 243, 243); border: none; padding-left: 10px; font-size: 16px;
    }");
21
22      // 搜索框横跨 8 列
23      layout->addWidget(searchEdit, 0, 0, 1, 8);
24
25      // 4. 创建搜索按钮
26      QPushButton* searchBtn = new QPushButton();
27      searchBtn->setFixedSize(50, 50);
28      searchBtn->setIcon(QIcon(":/image/search.png"));
29      searchBtn->setIconSize(QSize(30, 30));
30      QString style = "QPushButton { background-color: rgb(243, 243, 243);
    border: none; border-radius: 10px;}";
31      style += "QPushButton:hover {background-color: rgb(219, 219, 219);}
    QPushButton:pressed {background-color: rgb(206, 206, 206);}";
32      searchBtn->setStyleSheet(style);
33      connect(searchBtn, &QPushButton::clicked, this,
    &AddFriendDialog::clickSearchBtn);
34
35      // 搜索按钮占 1 行
36      layout->addWidget(searchBtn, 0, 8, 1, 1);
```

```
37
38      // 5. 创建结果展示区
39      initResultArea(layout);
40  }
41
42  void AddFriendDialog::initResultArea(QGridLayout *layout)
43  {
44      // 1. 创建滚动区域
45      QScrollArea* scrollArea = new QScrollArea();
46      scrollArea->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
47      // 一定要添加这个设置，否则无法正确显示.
48      scrollArea->setWidgetResizable(true);
49      // 隐藏水平滚动条. 把垂直滚动条设置的细一些.
50      scrollArea->verticalScrollBar()->setStyleSheet("QScrollBar:vertical {
    width: 2px; background-color: rgb(255, 255, 255); }
    QScrollBar::handle:vertical {background-color: rgb(205, 205, 205);}");
51      scrollArea->horizontalScrollBar()->setStyleSheet("QScrollBar:horizontal {
    height: 0px;}");
52      scrollArea->setStyleSheet("QScrollArea { border: none; }");
53
54      // 2. 创建滚动区核心 Widget
55      resultContainer = new QWidget();
56      resultContainer->setObjectName("resultContainer");
57      resultContainer->setStyleSheet("#resultContainer {background-color:
    rgb(255, 255, 255);}");
58      scrollArea->setWidget(resultContainer);
59
60      // 3. 给核心 Widget 添加布局管理器
61      QVBoxLayout* vlayout = new QVBoxLayout();
62      vlayout->setContentsMargins(0, 0, 0, 0);
63      vlayout->setSpacing(10);
64      vlayout->setAlignment(Qt::AlignTop);
65      resultContainer->setLayout(vlayout);
66
67      // 4. 添加到布局管理器中，占据 9 列空间.
68      layout->addWidget(scrollArea, 1, 0, 1, 9);
69
70      // 5. 添加结果元素
71  #if TEST_UI
72      for (int i = 0; i < 20; i++) {
73          UserInfo* userInfo = new UserInfo();
74          userInfo->avatar = QIcon(":/image/defaultAvatar.png");
75          userInfo->nickName = "张三";
76          userInfo->description = "我的个性签名"
77          this->addResult(*userInfo);
78      }
79  #endif
```

```
}

// 给结果中添加一个元素
void AddFriendDialog::addResult(const UserInfo& userInfo)
{
    FriendResultItem* item = new FriendResultItem(userInfo);
    resultContainer->layout()->addWidget(item);
}

void AddFriendDialog::clear()
{
    QVBoxLayout* layout = dynamic_cast<QVBoxLayout*>(resultContainer->layout());
    for (int i = layout->count() - 1; i >= 0; --i) {
        QWidget* w = layout->itemAt(i)->widget();
        if (w == nullptr) {
            continue;
        }
        layout->removeWidget(w);
        delete w;
    }
}

void AddFriendDialog::setSearchKey(const QString &searchKey)
{
    searchEdit->setText(searchKey);
}
```

2) 实现好友结果元素

创建 `FriendResultItem` 类

```
class FriendResultItem : public QWidget
{
public:
    FriendResultItem(const UserInfo& userInfo);

private:
    // 存储该好友的信息
    const UserInfo& userInfo;

    // 添加好友按钮
    QPushButton* addBtn;
};
```

```cpp
FriendResultItem::FriendResultItem(const UserInfo& userInfo) :
userInfo(userInfo)
{
    // 1. 设置基本属性
    this->setFixedHeight(70);
    this->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);

    // 2. 创建核心布局
    QGridLayout* layout = new QGridLayout();
    layout->setContentsMargins(0, 0, 30, 0);
    layout->setSpacing(10);
    this->setLayout(layout);

    // 3. 创建头像
    QPushButton* avatarBtn = new QPushButton();
    avatarBtn->setFixedSize(50, 50);
    avatarBtn->setIconSize(QSize(50, 50));
    avatarBtn->setIcon(userInfo.avatar);

    // 4. 创建名字
    QLabel* nameLabel = new QLabel();
    nameLabel->setFixedHeight(35);
    nameLabel->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    nameLabel->setAlignment(Qt::AlignLeft | Qt::AlignVCenter);
    nameLabel->setStyleSheet("QLabel { font-size: 16px; font-weight: 600;}");
    nameLabel->setText(userInfo.nickname);

    // 5. 创建签名
    QLabel* signatureLabel = new QLabel();
    signatureLabel->setFixedHeight(35);
    signatureLabel->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    signatureLabel->setAlignment(Qt::AlignLeft | Qt::AlignVCenter);
    signatureLabel->setText(userInfo.description);

    // 6. 创建添加按钮
    addBtn = new QPushButton();
    addBtn->setFixedSize(100, 40);
    addBtn->setText("添加好友");
    addBtn->setStyleSheet("QPushButton { border: none; background-color:
rgb(137, 217, 97); color: rgb(255, 255, 255); border-radius: 10px; }
QPushButton:pressed { background-color: rgb(198, 198, 198); }");
    connect(addBtn, &QPushButton::clicked, this,
&FriendResultItem::clickAddBtn);

```

```
    // 7. 添加到布局管理器中
    layout->addWidget(avatarBtn, 0, 0, 2, 1);
    layout->addWidget(nameLabel, 0, 1, 1, 1);
    layout->addWidget(signatureLabel, 1, 1, 1, 1);
    layout->addWidget(addBtn, 0, 2, 2, 1);
}

```

3) 弹出添加好友对话框

在 `MainWidget::initData` 中

a) 处理 `+` 按钮的点击

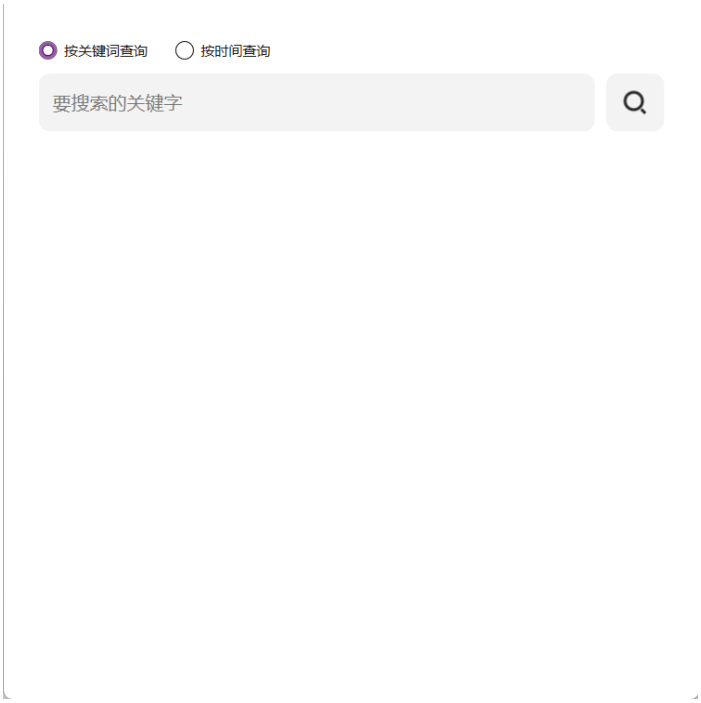
```
 connect(addFriendBtn, &QPushButton::clicked, this, [=](){
    AddFriendDialog* dialog = new AddFriendDialog(this);
    dialog->exec();
    delete dialog;
});
```

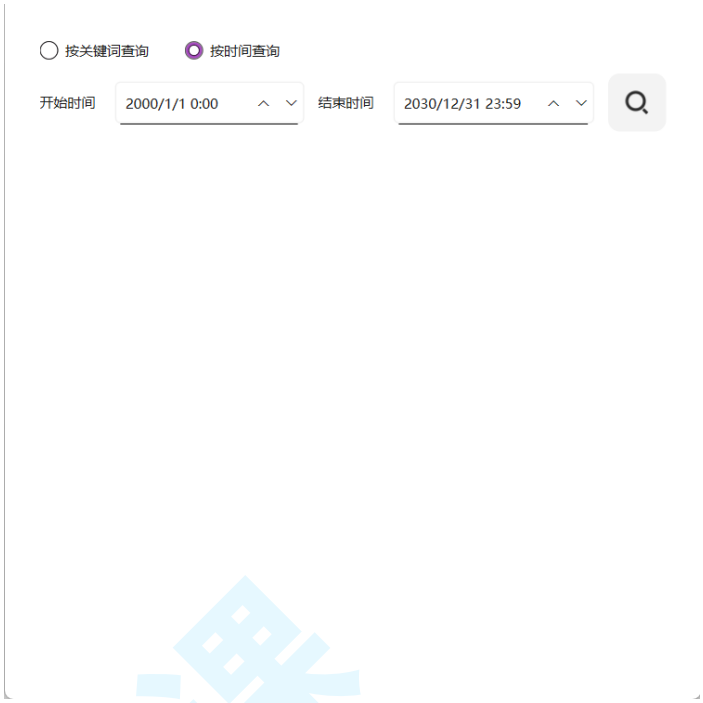
b) 处理输入框输入

```
// 此处要使用 textEdit 信号, 在 setText 的时候不会触发.
// 不要使用 textChanged!
connect(searchEdit, &QLineEdit::textEdited, this, [=]() {
    // 获取到输入框内容并清空
    const QString& text = searchEdit->text();
    searchEdit->setText("");
    // 弹出次级对话框
    AddFriendDialog* dialog = new AddFriendDialog();
    dialog->setSearchKey(text);
    dialog->exec();
    delete dialog;
});
```

实现历史消息界面

点击查看历史消息按钮时弹出该窗口





1) 实现历史消息窗口

创建 `HistoryMessageWidget`

```
class HistoryMessageWidget : public QDialog
{
    Q_OBJECT
public:
    explicit HistoryMessageWidget(QWidget *parent = nullptr);
    QScrollArea* initScrollArea();

    // 清空显示的搜索结果
    void clear();

    // 新增一条结果
    void addHistoryMessage(const Message& message);

private:
    QRadioButton* radioBtn1;
    QRadioButton* radioBtn2;
    QLineEdit* searchEdit;
    QPushButton* searchBtn;
    QLabel* begLabel;
    QLabel* endLabel;
    QDateTimeEdit* begTimeEdit;
    QDateTimeEdit* endTimeEdit;

    // 保存历史消息条目
```

```
    QWidget* container;

    void clickSearchBtn();
    void clickSearchBtnDone();

signals:
};
```

```
HistoryMessageWidget::HistoryMessageWidget(QWidget *parent) : QDialog(parent)
{
    // 1. 设置自身属性.
    this->setFixedSize(600, 600);
    this->setWindowTitle("历史消息");
    this->setWindowIcon(QIcon(":/image/logo.png"));
    // 设置关闭后自动销毁
    this->setAttribute(Qt::WA_DeleteOnClose);
    // 设置背景色
    this->setStyleSheet("QWidget { background-color: rgb(255, 255, 255); } ");

    // 2. 创建核心布局
    QGridLayout* layout = new QGridLayout();
    layout->setContentsMargins(30, 30, 30, 30);
    layout->setSpacing(10);
    this->setLayout(layout);

    // 创建单选框
    radioBtn1 = new QRadioButton();
    radioBtn2 = new QRadioButton();
    radioBtn1->setText("按关键词查询");
    radioBtn2->setText("按时间查询");
    layout->addWidget(radioBtn1, 0, 0, 1, 2);
    layout->addWidget(radioBtn2, 0, 2, 1, 2);
    radioBtn1->setChecked(true);   // 默认按关键词查询

    // 3. 创建搜索框
    searchEdit = new QLineEdit();
    searchEdit->setFixedHeight(50);
    searchEdit->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    searchEdit->setPlaceholderText("要搜索的关键字");
    searchEdit->setStyleSheet("QLineEdit { border-radius: 10px; background-
color: rgb(243, 243, 243); border: none; padding-left: 10px; font-size: 16px;
}");

    // 搜索框横跨 8 列
    layout->addWidget(searchEdit, 1, 0, 1, 8);
```

```
36
37      // 4. 创建搜索按钮
38      searchBtn = new QPushButton();
39      searchBtn->setFixedSize(50, 50);
40      searchBtn->setIcon(QIcon(":/image/search.png"));
41      searchBtn->setIconSize(QSize(30, 30));
42      QString style = "QPushButton { background-color: rgb(243, 243, 243);
    border: none; border-radius: 10px;}";
43      style += "QPushButton:hover {background-color: rgb(219, 219, 219);}
    QPushButton:pressed {background-color: rgb(206, 206, 206);}";
44      searchBtn->setStyleSheet(style);
45
46      // 搜索按钮占 1 行
47      layout->addWidget(searchBtn, 1, 8, 1, 1);
48
49      // 5. 创建日期选择器
50      begLabel = new QLabel();
51      begLabel->setText("开始时间");
52      endLabel = new QLabel();
53      endLabel->setText("结束时间");
54      begTimeEdit = new QDateTimeEdit();
55      endTimeEdit = new QDateTimeEdit();
56      auto endDateTime = QDateTime::currentDateTime();
57      auto begDateTime = endDateTime.addDays(-30);
58      begTimeEdit->setDateTime(begDateTime);
59      endTimeEdit->setDateTime(endDateTime);
60      begTimeEdit->setFixedHeight(40);
61      endTimeEdit->setFixedHeight(40);
62      // 初始情况隐藏.
63      begLabel->hide();
64      endLabel->hide();
65      begTimeEdit->hide();
66      endTimeEdit->hide();
67
68      // 6. 创建历史消息显示区域
69      QScrollArea* scrollArea = initScrollArea();
70      // 直接占据 9 列.
71      layout->addWidget(scrollArea, 2, 0, 1, 9);
72
73      // 7. 实现切换搜索模式
74      connect(radioBtn1, &QRadioButton::clicked, this, [=]() {
75          layout->removeWidget(begLabel);
76          layout->removeWidget(endLabel);
77          layout->removeWidget(begTimeEdit);
78          layout->removeWidget(endTimeEdit);
79          begLabel->hide();
80          endLabel->hide();
```

```
        begTimeEdit->hide();
        endTimeEdit->hide();

        layout->addWidget(searchEdit, 1, 0, 1, 8);
        searchEdit->show();
    });

    connect(radioBtn2, &QRadioButton::clicked, this, [=]() {
        layout->removeWidget(searchEdit);
        searchEdit->hide();

        layout->addWidget(begLabel, 1, 0, 1, 1);
        layout->addWidget(begTimeEdit, 1, 1, 1, 3);
        layout->addWidget(endLabel, 1, 4, 1, 1);
        layout->addWidget(endTimeEdit, 1, 5, 1, 3);
        begLabel->show();
        endLabel->show();
        begTimeEdit->show();
        endTimeEdit->show();
    });


    // 8. 添加数据
#if TEST_UI
    QString text = "很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息很长的消息";
    addHistoryMessage(QIcon(":/image/defaultAvatar.png"), "张三", "2024-04-25 18:30:00", TEXT_TYPE, text.toUtf8());
    for (int i = 0; i < 30; i++) {
        addHistoryMessage(QIcon(":/image/defaultAvatar.png"), "张三", "2024-04-25 18:30:00", TEXT_TYPE, QString("测试消息").toUtf8());
    }
#endif
}
```

2) 创建展示消息内容区域

```
QScrollArea* HistoryMessageWidget::initScrollArea()
{
    // 创建成员列表区
    QScrollArea* scrollArea = new QScrollArea();
    scrollArea->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
```

```
    // 一定要添加这个设置，否则无法正确显示.
    scrollArea->setWidgetResizable(true);
    // 隐藏水平滚动条. 把垂直滚动条设置的细一些.
    scrollArea->verticalScrollBar()->setStyleSheet("QScrollBar:vertical { width: 2px; background-color: rgb(255, 255, 255); } QScrollBar::handle:vertical {background-color: rgb(205, 205, 205);}");
    scrollArea->horizontalScrollBar()->setStyleSheet("QScrollBar:horizontal { height: 0px;}");
    scrollArea->setStyleSheet("QScrollArea { border: none;}");

    container = new QWidget();
    scrollArea->setWidget(container);

    QVBoxLayout* layout = new QVBoxLayout();
    layout->setContentsMargins(0, 0, 0, 0);
    layout->setSpacing(0);
    layout->setAlignment(Qt::AlignTop);
    container->setLayout(layout);

    return scrollArea;
}

void HistoryMessageWidget::clear()
{
    QVBoxLayout* layout = dynamic_cast<QVBoxLayout*>(container->layout());
    for (int i = layout->count() - 1; i >= 0; --i) {
        QWidget* w = layout->itemAt(i)->widget();
        if (w == nullptr) {
            continue;
        }
        layout->removeWidget(w);
        delete w;
    }
}

void HistoryMessageWidget::addHistoryMessage(const Message& message)
{
    HistoryMessageItem* item = HistoryMessageItem::makeHistoryMessageItem(message);
    container->layout()->addWidget(item);
}
```

3) 实现历史消息条目

创建 `HistoryMessageItem` 类

```
class HistoryMessageItem : public QWidget {
public:
    HistoryMessageItem();

    static HistoryMessageItem* makeHistoryMessageItem(const Message& message);

private:
};
```

```
HistoryMessageItem::HistoryMessageItem()
{

}

HistoryMessageItem *HistoryMessageItem::makeHistoryMessageItem(const Message&
message)
{
    // 1. 设置核心属性
    HistoryMessageItem* item = new HistoryMessageItem();
    item->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Preferred);

    // 2. 创建主布局
    QGridLayout* gridLayout = new QGridLayout();
    gridLayout->setContentsMargins(0, 0, 0, 0);
    gridLayout->setSpacing(10);
    item->setLayout(gridLayout);

    // 3. 创建头像
    QPushButton* avatarBtn = new QPushButton();
    avatarBtn->setFixedSize(40, 40);
    avatarBtn->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);
    avatarBtn->setIconSize(QSize(40, 40));
    avatarBtn->setIcon(message.sender.avatar);
    avatarBtn->setStyleSheet("QPushButton { border: none; }");

    // 4. 创建名字 + 时间
    QLabel* nameLabel = new QLabel();
    nameLabel->setText(message.sender.nickname + " | " + message.time);
    nameLabel->setFixedHeight(20);
    nameLabel->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);

    // 5. 创建消息体
    QWidget* contentWidget = nullptr;
    if (message.messageType == TEXT_TYPE) {
```

```
        QLabel *label = new QLabel();
        label->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
// 设置大小策略
        label->setWordWrap(true); // 启用文本换行
        label->adjustSize(); // 调整大小以适应内容
        label->setAlignment(Qt::AlignTop | Qt::AlignLeft);
        label->setText(QString(message.content));

        contentWidget = label;
    } else if (message.messageType == IMAGE_TYPE) {
        contentWidget = new ImageButton(message.fileId, message.content);
    } else if (message.messageType == FILE_TYPE) {
        contentWidget = new FileLabel(message.fileId, message.content,
message.fileName);
    } else if (message.messageType == SPEECH_TYPE) {
        contentWidget = new SoundLabel(message.fileId, message.content);
    } else {
        LOG() << "错误的 messageType = " << message.messageType;
    }

    // 6. 把上述控件放入布局中.
    gridLayout->addWidget(avatarBtn, 0, 0, 2, 1);
    gridLayout->addWidget(nameLabel, 0, 1, 1, 1);
    // 这里先设置占 5 行
    gridLayout->addWidget(contentWidget, 1, 1, 5, 1);

    return item;
}
```

4) 图片消息, 文件消息, 语音消息 放到后面再实现

5) 弹出历史消息对话框

在 `MessageEditArea::initSignalSlot` 中连接信号槽.

```
connect(showHistoryBtn, &QPushButton::clicked, this, [=]() {
    HistoryMessageWidget* widget = new HistoryMessageWidget();
    widget->show();
});
```

实现用户名登录/注册界面

程序启动, 会先打开登录注册窗口.

1) 实现用户注册登录窗口

创建 `LoginWidget`

```
class LoginWidget : public QWidget
{
    Q_OBJECT
public:
    explicit LoginWidget(QWidget *parent = nullptr);


private:
    QLabel* tipLabel;
    QLineEdit* usernameEdit;
    QLineEdit* passwordEdit;
    QLineEdit* verifyCodeEdit;
    VerifyCodeWidget* verifyCodeWidget;
    QPushButton* submitBtn;
    QPushButton* phoneModeBtn;
    QPushButton* switchModeBtn;

    // 是否是登录模式
    bool isLoginMode;

    // 切换注册/登录
    void switchMode();

    // 切换到电话登录模式
    void switchToPhone();
```

```
26
27 signals:
28 };
```

```
1  LoginWidget::LoginWidget(QWidget *parent) : QWidget(parent), isLoginMode(true)
2  {
3      // 1. 设置基本属性
4      this->setFixedSize(400, 350);
5      this->setWindowTitle("登录");
6      this->setWindowIcon(QIcon(":/image/logo.png"));
7      this->setStyleSheet("QWidget { background-color: rgb(255, 255, 255); }");
8      // 如果是在栈上定义的变量，不能 delete，会使程序崩溃.
9      this->setAttribute(Qt::WA_DeleteOnClose);
10
11     // 2. 创建核心布局
12     QGridLayout* layout = new QGridLayout();
13     layout->setContentsMargins(50, 0, 50, 0);
14     layout->setSpacing(0);
15     this->setLayout(layout);
16
17     // 3. 创建标题
18     tipLabel = new QLabel();
19     tipLabel->setText("登录");
20     tipLabel->setAlignment(Qt::AlignCenter);
21     tipLabel->setFixedHeight(50);
22     tipLabel->setStyleSheet("QLabel {font-size: 40px;}");
23
24     // 4. 创建输入框
25     usernameEdit = new QLineEdit();
26     usernameEdit->setPlaceholderText("输入用户名");
27     usernameEdit->setFixedHeight(40);
28     usernameEdit->setStyleSheet("QLineEdit { padding-left: 10px; font-size:
   20px; border-radius: 5px; background-color: rgb(235, 235, 235); }");
29
30     passwordEdit = new QLineEdit();
31     passwordEdit->setPlaceholderText("输入密码");
32     passwordEdit->setFixedHeight(40);
33     passwordEdit->setStyleSheet("QLineEdit { padding-left: 10px; font-size:
   20px; border-radius: 5px; background-color: rgb(235, 235, 235); }");
34     passwordEdit->setEchoMode(QLineEdit::Password);
35
36     verifyCodeEdit = new QLineEdit();
37     verifyCodeEdit->setPlaceholderText("输入验证码");
38     verifyCodeEdit->setFixedHeight(40);
```

```
39      verifyCodeEdit->setStyleSheet("QLineEdit { padding-left: 10px; font-size:
    20px; border-radius: 5px; background-color: rgb(235, 235, 235); }");
40      verifyCodeEdit->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
41
42      verifyCodeWidget = new VerifyCodeWidget();
43      verifyCodeWidget->setFixedSize(100, 40);
44      // verifyCodeWidget->setStyleSheet("QPushButton {border: none;}");
45      // verifyCodeWidget->setText("验证码");
46
47      // 5. 创建提交按钮
48      submitBtn = new QPushButton();
49      submitBtn->setFixedHeight(40);
50      submitBtn->setStyleSheet("QPushButton { color: rgb(255, 255, 255);
    background-color: rgb(44, 182, 61); border: none; border-radius: 5px;}
    QPushButton:pressed {background-color: rgb(235, 235, 235);}");
51      submitBtn->setText("登录");
52
53      // 6. 创建切换手机号登录按钮
54      phoneModeBtn = new QPushButton();
55      phoneModeBtn->setFixedHeight(40);
56      phoneModeBtn->setStyleSheet("QPushButton { border: none; border-radius:
    5px;} QPushButton:pressed {background-color: rgb(235, 235, 235);}");
57      phoneModeBtn->setText("切换到手机号");
58
59      connect(phoneModeBtn, &QPushButton::clicked, this,
    &LoginWidget::switchToPhone);
60
61      // 7. 创建切换注册按钮
62      switchModeBtn = new QPushButton();
63      switchModeBtn->setFixedHeight(40);
64      switchModeBtn->setStyleSheet("QPushButton { border: none; border-radius:
    5px;} QPushButton:pressed {background-color: rgb(235, 235, 235);}");
65      switchModeBtn->setText("注册");
66
67      connect(switchModeBtn, &QPushButton::clicked, this,
    &LoginWidget::switchMode);
68
69      // 8. 添加到布局管理器中
70      layout->addWidget(tipLabel, 0, 0, 1, 5);
71      layout->addWidget(usernameEdit, 1, 0, 1, 5);
72      layout->addWidget(passwordEdit, 2, 0, 1, 5);
73      layout->addWidget(verifyCodeEdit, 3, 0, 1, 4);
74      layout->addWidget(verifyCodeWidget, 3, 4, 1, 1);
75      layout->addWidget(submitBtn, 4, 0, 1, 5);
76      layout->addWidget(phoneModeBtn, 5, 0, 1, 1);
77      layout->addWidget(switchModeBtn, 5, 4, 1, 1);
78  }
```

2) 实现界面切换

```
void LoginWidget::switchMode()
{
    if (isLoginMode) {
        // 切换到注册模式
        this->setWindowTitle("注册");
        tipLabel->setText("注册");
        submitBtn->setText("注册");
        switchModeBtn->setText("登录");
        isLoginMode = false;
    } else {
        // 切换到登录模式
        this->setWindowTitle("登录");
        tipLabel->setText("登录");
        submitBtn->setText("登录");
        switchModeBtn->setText("注册");
        isLoginMode = true;
    }
}

// 打开通过手机号注册登录页面
void LoginWidget::switchToPhone()
{
    PhoneLoginWidget* widget = new PhoneLoginWidget();
    widget->show();

    // 关闭当前窗口
    this->close();
}
```

实现手机号登录/注册界面

1) 实现手机号注册登录窗口

创建 `PhoneLoginWidget`

```
PhoneLoginWidget::PhoneLoginWidget(QWidget *parent) : QWidget(parent), isLoginMode(true)
{
    // 1. 设置基本属性
    this->setFixedSize(400, 350);
    this->setWindowTitle("登录");
    this->setWindowIcon(QIcon(":/image/logo.png"));
    this->setStyleSheet("QWidget { background-color: rgb(255, 255, 255); }");
    // 如果是在栈上定义的变量，不能 delete，会使程序崩溃.
    this->setAttribute(Qt::WA_DeleteOnClose);

    // 2. 创建核心布局
    QGridLayout* layout = new QGridLayout();
    layout->setContentsMargins(50, 0, 50, 0);
    layout->setSpacing(0);
    this->setLayout(layout);

    // 3. 创建标题
    tipLabel = new QLabel();
    tipLabel->setText("登录");
    tipLabel->setAlignment(Qt::AlignCenter);
    tipLabel->setFixedHeight(50);
    tipLabel->setStyleSheet("QLabel {font-size: 40px;}");

    // 4. 创建输入框
    phoneEdit = new QLineEdit();
    phoneEdit->setPlaceholderText("输入电话");
```

```
27      phoneEdit->setFixedHeight(40);
28      phoneEdit->setStyleSheet("QLineEdit { padding-left: 10px; font-size: 20px;
    border-radius: 5px; background-color: rgb(235, 235, 235); }");
29
30      verifyCodeEdit = new QLineEdit();
31      verifyCodeEdit->setPlaceholderText("输入验证码");
32      verifyCodeEdit->setFixedHeight(40);
33      verifyCodeEdit->setStyleSheet("QLineEdit { padding-left: 10px; font-size:
    20px; border-radius: 5px; background-color: rgb(235, 235, 235); }");
34      verifyCodeEdit->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
35
36      verifyCodeBtn = new QPushButton();
37      verifyCodeBtn->setFixedSize(100, 40);
38      verifyCodeBtn->setStyleSheet("QPushButton {border: none;}");
39      verifyCodeBtn->setText("发送验证码");
40
41      // 5. 创建提交按钮
42      submitBtn = new QPushButton();
43      submitBtn->setFixedHeight(40);
44      submitBtn->setStyleSheet("QPushButton { color: rgb(255, 255, 255);
    background-color: rgb(44, 182, 61); border: none; border-radius: 5px;}
    QPushButton:pressed {background-color: rgb(235, 235, 235);}");
45      submitBtn->setText("登录");
46
47      // 6. 创建切换手机号登录按钮
48      switchToUsernameBtn = new QPushButton();
49      switchToUsernameBtn->setFixedHeight(40);
50      switchToUsernameBtn->setStyleSheet("QPushButton { border: none; border-
    radius: 5px;} QPushButton:pressed {background-color: rgb(235, 235, 235);}");
51      switchToUsernameBtn->setText("切换到用户名");
52
53      connect(switchToUsernameBtn, &QPushButton::clicked, this,
    &PhoneLoginWidget::switchToUsername);
54
55      // 7. 创建切换注册按钮
56      switchModeBtn = new QPushButton();
57      switchModeBtn->setFixedHeight(40);
58      switchModeBtn->setStyleSheet("QPushButton { border: none; border-radius:
    5px;} QPushButton:pressed {background-color: rgb(235, 235, 235);}");
59      switchModeBtn->setText("注册");
60
61      connect(switchModeBtn, &QPushButton::clicked, this,
    &PhoneLoginWidget::switchMode);
62
63      // 8. 添加到布局管理器中
64      layout->addWidget(tipLabel, 0, 0, 1, 5);
65      layout->addWidget(phoneEdit, 1, 0, 1, 5);
```

```
    layout->addWidget(verifyCodeEdit, 3, 0, 1, 4);
    layout->addWidget(verifyCodeBtn, 3, 4, 1, 1);
    layout->addWidget(submitBtn, 4, 0, 1, 5);
    layout->addWidget(switchToUsernameBtn, 5, 0, 1, 1);
    layout->addWidget(switchModeBtn, 5, 4, 1, 1);

    // 9. 初始化 QTimer
    timer = new QTimer(this);
    connect(timer, &QTimer::timeout, this, &PhoneLoginWidget::countDown);
}
```

2) 实现界面切换

```
// 切换注册/登录模式
void PhoneLoginWidget::switchMode()
{
    if (isLoginMode) {
        // 切换到注册模式
        this->setWindowTitle("注册");
        tipLabel->setText("注册");
        submitBtn->setText("注册");
        switchModeBtn->setText("登录");
        isLoginMode = false;
    } else {
        // 切换到登录模式
        this->setWindowTitle("登录");
        tipLabel->setText("登录");
        submitBtn->setText("登录");
        switchModeBtn->setText("注册");
        isLoginMode = true;
    }
}

// 切换到用户名登录注册
void PhoneLoginWidget::switchToUsername()
{
    LoginWidget* widget = new LoginWidget();
    widget->show();

    this->close();
}
```

实现全局通知

创建 `Toast` 类

```
class Toast : public QDialog {
    Q_OBJECT
public:
    Toast(const QString& text);

    static void showMessage(const QString& text);

};
```

```
Toast::Toast(const QString &text) : QDialog(nullptr) {
    // 1. 设置基本参数
    this->setFixedSize(500, 100);
    this->setWindowFlags(Qt::FramelessWindowHint);
    this->setWindowIcon(QIcon(":/image/logo.png"));
    this->setAttribute(Qt::WA_DeleteOnClose);
    this->setStyleSheet("QDialog { background-color: rgb(255, 255, 255);
border-radius: 10px; }");

    // 2. 计算摆放位置
    QScreen *screen = QApplication::primaryScreen();
    int width = screen->size().width();
    int height = screen->size().height();
    int x = (width - this->width()) / 2;
    int y = height - this->height() - 100;
    this->move(x, y);

    // 3. 添加内容
    QHBoxLayout* layout = new QHBoxLayout();
    layout->setContentsMargins(20, 20, 20, 20);
    this->setLayout(layout);

    QLabel* label = new QLabel();
    label->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
    label->setAlignment(Qt::AlignCenter);
    label->setText(text);
    QFont font;
    font.setPointSize(18);
    label->setFont(font);
```

```
    layout->addWidget(label);

    // 4. 实现 2s 后自动关闭窗口
    QTimer* timer = new QTimer(this);
    connect(timer, &QTimer::timeout, this, [=]() {
        timer->stop();
        this->close();
    });
    timer->start(2000);
}

void Toast::showMessage(const QString &text) {
    Toast* toast = new Toast(text);
    toast->show();
}
```

构建界面注意事项

1) 直接通过 QSS 给 QWidget 设置背景色, 有时候会失效. 尤其是 QWidget 的子类的时候.

具体原因还不清楚.

官方文档说:

QWidget

Supports only the background, background-clip and background-origin properties.
If you subclass from QWidget, you need to provide a paintEvent for your custom QWidget as below:

```
void CustomWidget::paintEvent(QPaintEvent *)
{
    QStyleOption opt;
    opt.init(this);
    QPainter p(this);
    style()->drawPrimitive(QStyle::PE_Widget, &opt, &p, this);
}
```

The above code is a no-operation if there is no stylesheet set.
Warning: Make sure you define the Q_OBJECT macro for your custom widget.

原因没有解释.

```
void CustomWidget::paintEvent(QPaintEvent *)
  {
      QStyleOption opt;
      opt.init(this);
      QPainter p(this);
      style()->drawPrimitive(QStyle::PE_Widget, &opt, &p, this);
  }
```

2) QScrollArea 不能通过 QSS 直接设置背景色.

要给 QScrollArea 中持有的 QWidget 设置.

3) QCheckBox 不能通过 QSS 的 border-radius 设置圆形.

形如下列代码, 不能生效.

```
QString style ="QCheckBox { border-radius: 12.5px; background-color: white; }
QCheckBox::indicator { width:25px; height: 25px; border-radius: 12.5px;} ";
style += "QCheckBox::indicator:checked{ color: white; background-color: rgb(7,
193, 96); }";
checkBox->setStyleSheet(style);
```

需要使用替换背景图的方式来完成.

4) 滚动区域代码示例

- 使用 QScrollBar:vertical 设置垂直滚动条样式.
- 使用 QScrollBar:horizontal 设置水平滚动条样式.
- 使用 QScrollBar::handle:vertical 设置垂直滚动条滑块样式.
- setWidgetResizable(true) 务必要添加.

```
QScrollArea* scrollArea = new QScrollArea();
scrollArea->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
// 一定要添加这个设置，否则无法正确显示.
scrollArea->setWidgetResizable(true);
// 隐藏水平滚动条. 把垂直滚动条设置的细一些.
scrollArea->verticalScrollBar()->setStyleSheet("QScrollBar:vertical { width:
2px; background-color: rgb(255, 255, 255); } QScrollBar::handle:vertical
{background-color: rgb(205, 205, 205);}");
scrollArea->horizontalScrollBar()->setStyleSheet("QScrollBar:horizontal {
height: 0px;}");
scrollArea->setStyleSheet("QScrollArea { border: none; }");

selectedContainer = new QWidget();
selectedContainer->setObjectName("selectedContainer");
```

```
12 selectedContainer->setStyleSheet("#selectedContainer {background-color:
   rgb(255, 255, 255);}");
13 scrollArea->setWidget(selectedContainer);
```

5) 针对登录窗口进行 delete 后程序崩溃

使用 deleteLater 也不行.

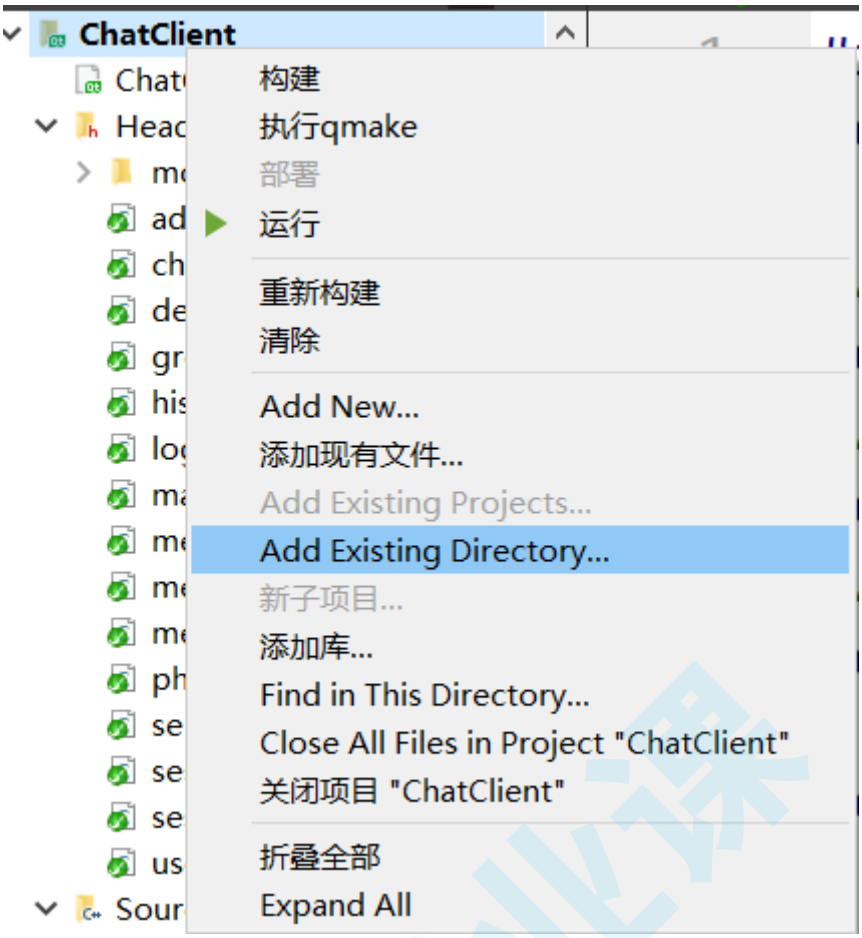
使用 this->setAttribute(Qt::WA_DeleteOnClose); 也不行.

原因是这个变量 `LoginWidget` 是在 main 中定义在栈上的. 不能 delete !

```
1  void LoginWidget::switchToPhone()
2  {
3      PhoneLoginWidget* widget = new PhoneLoginWidget();
4      widget->show();
5
6      // 关闭当前窗口
7      this->close();
8
9      // 注意!!! 此处不能 delete, 否则程序会崩溃.
10     // 因为该 LoginWidget 是在 main 中定义在栈上的变量, 是不能 delete 的!
11     delete this;
12     // this->deleteLater();
13 }
```

6) 使用 Qt Creator 添加目录

创建好目录 -> 右键项目 -> Add Existing Directory



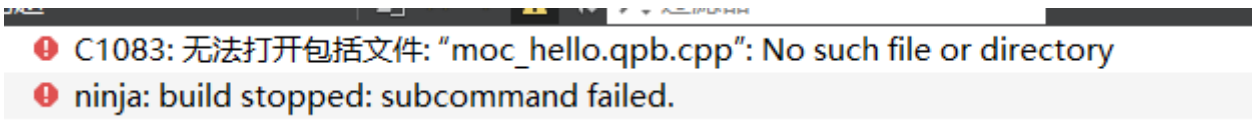
核心数据结构

引入 protobuf

1) 修改 CMake 最小版本为 3.16

```
1  cmake_minimum_required(VERSION 3.16)
```

> 默认的最小版本是 3.5 . 此处务必要修改成高版本, 否则后面可能会生成失败!



2) `find_package` 时引入 **`Protobuf`**

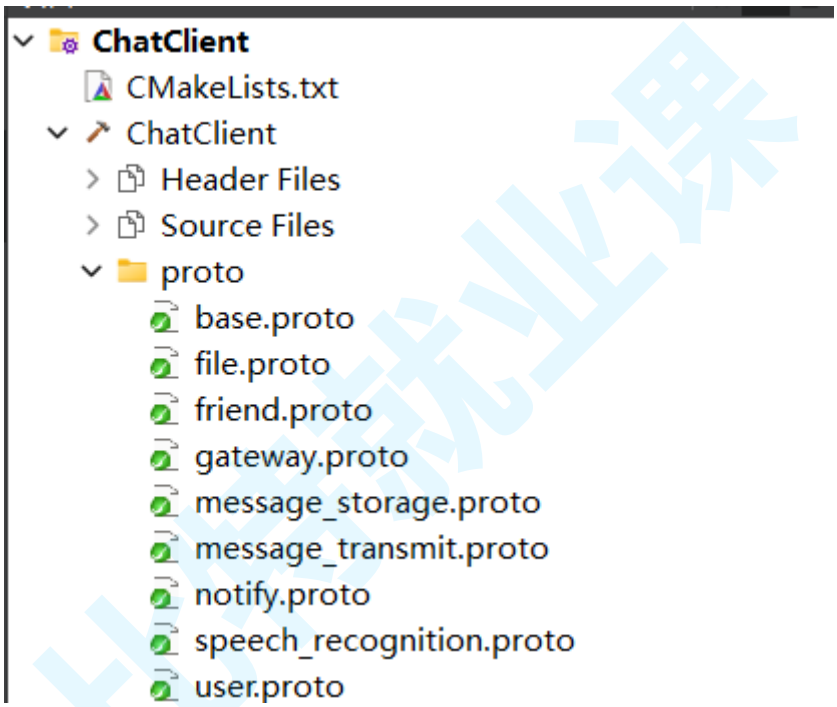
```
1  find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets Protobuf)
```

3) 创建 proto 目录, 并把服务器提供的 proto 拷贝过来

4) 修改 `CMakeLists.txt`

```
file(GLOB PB_FILES "./proto/*.proto")
...
qt_add_protobuf(ChatClient PROTO_FILES ${PB_FILES})
```

最终结果形如



前后端交互接口定义

1) 核心 PB 类

用户信息

```
//用户信息结构
message UserInfo {
    string user_id = 1;//用户ID
    string nickname = 2;//昵称
    string description = 3;//个人签名/描述
    string phone = 4; //绑定手机号
    bytes  avatar = 5;//头像照片，文件内容使用二进制
}

```

```
10
```

会话信息

```
//聊天会话信息
message ChatSessionInfo {
    optional string single_chat_friend_id = 1;//群聊会话不需要设置，单聊会话设置为对方ID
    string chat_session_id = 2; //会话ID
    string chat_session_name = 3;//会话名称git
    optional MessageInfo prev_message = 4;//会话上一条消息，新建的会话没有最新消息
    optional bytes avatar = 5;//会话头像 --群聊会话不需要，直接由前端固定渲染，单聊就是对方的头像
}
```

消息信息

```
//消息类型
enum MessageType {
    STRING = 0;
    IMAGE = 1;
    FILE = 2;
    SPEECH = 3;
}
message StringMessageInfo {
    string content = 1;//文字聊天内容
}
message ImageMessageInfo {
    optional string file_id = 1;//图片文件id,客户端发送的时候不用设置，由transmit服务器进行设置后交给storage的时候设置
    optional bytes image_content = 2;//图片数据，在ES中存储消息的时候只要id不要文件数据，服务端转发的时候需要原样转发
}
message FileMessageInfo {
    optional string file_id = 1;//文件id,客户端发送的时候不用设置
    int64 file_size = 2;//文件大小
    string file_name = 3;//文件名称
    optional bytes file_contents = 4;//文件数据，在ES中存储消息的时候只要id和元信息，不要文件数据，服务端转发的时候也不需要填充
}
```

```
message SpeechMessageInfo {
    optional string file_id = 1;//语音文件id,客户端发送的时候不用设置
    optional bytes file_contents = 2;//文件数据，在ES中存储消息的时候只要id不要文件数据，服务端转发的时候也不需要填充
}
message MessageContent {
    MessageType message_type = 1; //消息类型
    oneof msg_content {
        StringMessageInfo string_message = 2;//文字消息
        FileMessageInfo file_message = 3;//文件消息
        SpeechMessageInfo speech_message = 4;//语音消息
        ImageMessageInfo image_message = 5;//图片消息
    };
}
//消息结构
message MessageInfo {
    string message_id = 1;//消息ID
    string chat_session_id = 2;//消息所属聊天会话ID
    int64 timestamp = 3;//消息产生时间
    UserInfo sender = 4;//消息发送者信息
    MessageContent message = 5;
}

message Message {
    string request_id = 1;
    MessageInfo message = 2;
}
```

2) HTTP 接口定义

请求响应基本格式.

```
//通信接口统一采用POST请求实现,正文采用protobuf协议进行组织
/*
    HTTP HEADER:
    POST /service/xxxxx
    Content-Type: application/x-protobuf
    Content-Length: 123

    xxxxxx

    ------------------------------------------------------

    HTTP/1.1 200 OK
```

```
    Content-Type: application/x-protobuf
    Content-Length: 123

    xxxxxxxxxx
*/
```

约定路径

> 每个接口都提供对应的请求响应的 proto 对象. 具体到后面实现中再展开.

```
//在客户端与网关服务器的通信中，使用HTTP协议进行通信
//  通信时采用POST请求作为请求方法
//  通信时，正文采用protobuf作为正文协议格式，具体内容字段以前边各个文件中定义的字段格式为准
/*  以下是HTTP请求的功能与接口路径对应关系：
SERVICE HTTP PATH:
{
    获取随机验证码                  /service/user/get_random_verify_code
    获取短信验证码                  /service/user/get_phone_verify_code
    用户名密码注册                  /service/user/username_register
    用户名密码登录                  /service/user/username_login
    手机号码注册                    /service/user/phone_register
    手机号码登录                    /service/user/phone_login
    获取个人信息                    /service/user/get_user_info
    修改头像                        /service/user/set_avatar
    修改昵称                        /service/user/set_nickname
    修改签名                        /service/user/set_description
    修改绑定手机                    /service/user/set_phone

    获取好友列表                    /service/friend/get_friend_list
    获取好友信息                    /service/friend/get_friend_info
    发送好友申请                    /service/friend/add_friend_apply
    好友申请处理                    /service/friend/add_friend_process
    删除好友                        /service/friend/remove_friend
    搜索用户                        /service/friend/search_friend
    获取指定用户的消息会话列表      /service/friend/get_chat_session_list
    创建消息会话                    /service/friend/create_chat_session
    获取消息会话成员列表            /service/friend/get_chat_session_member
    获取待处理好友申请事件列表      /service/friend/get_pending_friend_events

    获取历史消息/离线消息列表       /service/message_storage/get_history
    获取最近N条消息列表             /service/message_storage/get_recent
    搜索历史消息                    /service/message_storage/search_history

```

```
    发送消息                        /service/message_transmit/new_message

    获取单个文件数据                /service/file/get_single_file
    获取多个文件数据                /service/file/get_multi_file
    发送单个文件                    /service/file/put_single_file
    发送多个文件                    /service/file/put_multi_file

    语音转文字                      /service/speech/recognition
}
```

3) websocket 接口定义

a) 身份认证

```
/*
    消息推送使用websocket长连接进行
    websocket长连接转换请求：ws://host:ip/ws
    长连建立以后，需要客户端给服务器发送一个身份验证信息
*/
message ClientAuthenticationReq {
    string request_id = 1;
    string session_id = 2;
}
message ClientAuthenticationRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
}
```

b) 消息推送

当前存在五种消息推送.

- 申请好友通知
- 好友申请处理通知 (同意/拒绝)
- 创建消息会话通知
- 收到消息通知
- 删除好友通知

```
enum NotifyType {
    FRIEND_ADD_APPLY_NOTIFY = 0;
    FRIEND_ADD_PROCESS_NOTIFY = 1;
    CHAT_SESSION_CREATE_NOTIFY = 2;
    CHAT_MESSAGE_NOTIFY = 3;
    FRIEND_REMOVE_NOTIFY = 4;
}

message NotifyFriendAddApply {
    UserInfo user_info = 1;  //申请人信息
}
message NotifyFriendAddProcess {
    bool agree = 1;
    UserInfo user_info = 2;  //处理人信息
}
message NotifyFriendRemove {
    string user_id = 1; //删除自己的用户ID
}
message NotifyNewChatSession {
    ChatSessionInfo chat_session_info = 1; //新建会话信息
}
message NotifyNewMessage {
    MessageInfo message_info = 1; //新消息
}

message NotifyMessage {
    optional string notify_event_id = 1;//通知事件操作id（有则填无则忽略）
    NotifyType notify_type = 2;//通知事件类型
    oneof notify_remarks {      //事件备注信息
        NotifyFriendAddApply friend_add_apply = 3;
        NotifyFriendAddProcess friend_process_result = 4;
        NotifyFriendRemove friend_remove = 7;
        NotifyNewChatSession new_chat_session_info = 5;//会话信息
        NotifyNewMessage new_message_info = 6;//消息信息
    }
}
```

核心数据结构和 PB 之间的转换

```
class UserInfo {
public:
    QString userId;
    QString nickname;
```

```
    QString description;
    QString phone;
    QIcon avatar;

    void load(const bite_im::UserInfo& userInfo) {
        userId = userInfo.userId();
        nickname = userInfo.nickname();
        phone = userInfo.phone();
        description = userInfo.description();
        if (userInfo.avatar().isEmpty()) {
            // 如果服务器上没有拿到头像，则使用默认头像
            avatar = QIcon(":/image/defaultAvatar.png");
        } else {
            avatar = makeIcon(userInfo.avatar());
        }
    }

};
```

```
class Message {
public:
    QString messageId = "";
    QString chatSessionId = "";
    QString time = "";  // 格式化时间
    MessageType messageType = TEXT_TYPE;
    UserInfo sender;
    // 实际内容取决于 messageType
    QByteArray content;
    // 如果是图片，文件，语音类型，表示对应的文件 id
    QString fileId = "";
    // 如果是文件消息，表示文件名
    QString fileName = "";
    // 如果是文件/语音，使用这个字段保存本地缓存的路径.
    // QString localPath = "";

    // 从 PB 中加载构造出 Message
    void load(const bite_im::MessageInfo& messageInfo) {
        messageId = messageInfo.messageId();
        chatSessionId = messageInfo.chatSessionId();
        time = formatTime(messageInfo.timestamp());
        sender.load(messageInfo.sender());

        auto type = messageInfo.message().messageType();
        if (type == bite_im::MessageTypeGadget::MessageType::STRING) {
            messageType = TEXT_TYPE;
```

```
            content = messageInfo.message().stringMessage().content().toUtf8();
        } else if (type == bite_im::MessageTypeGadget::MessageType::FILE) {
            messageType = FILE_TYPE;
            fileId = messageInfo.message().fileMessage().fileId();
            if (messageInfo.message().fileMessage().hasFileContents()) {
                content = messageInfo.message().fileMessage().fileContents();
            }
            fileName = messageInfo.message().fileMessage().fileName();
        } else if (type == bite_im::MessageTypeGadget::MessageType::IMAGE) {
            messageType = IMAGE_TYPE;
            fileId = messageInfo.message().imageMessage().fileId();
            if (messageInfo.message().imageMessage().hasImageContent()) {
                content = messageInfo.message().imageMessage().imageContent();
            }
        } else if (type == bite_im::MessageTypeGadget::MessageType::SPEECH){
            messageType = SPEECH_TYPE;
            fileId = messageInfo.message().speechMessage().fileId();
            if (messageInfo.message().speechMessage().hasFileContents()) {
                content = messageInfo.message().speechMessage().fileContents();
            }
        } else {
            messageType = UNKNOWN_TYPE;
        }
    }

     // ... 省略一系列 makeXXX
}
```

```
class ChatSessionInfo {
public:
    QString chatSessionId = "";
    QString chatSessionName = "";
    Message prevMessage;
    QIcon avatar;
    // 当会话为单聊时，表示对方的用户 id
    // 如果为群聊，则该字段为 ""
    QString userId = "";

    void load(const bite_im::ChatSessionInfo& chatSessionInfo) {
        chatSessionId = chatSessionInfo.chatSessionId();
        chatSessionName = chatSessionInfo.chatSessionName();
        if (chatSessionInfo.hasSingleChatFriendId()) {
            userId = chatSessionInfo.singleChatFriendId();
        }
        // 新的会话没有 prevMessage，需要先判定再 load
```

```
        if (chatSessionInfo.hasPrevMessage()) {
            prevMessage.load(chatSessionInfo.prevMessage());
        }
        if (!chatSessionInfo.hasAvatar() ||
chatSessionInfo.avatar().isEmpty()) {
            // 响应中没有会话头像
            // 单聊和群聊设置不同的默认头像
            if (userId != "") {
                // 设置单聊头像
                avatar = QIcon(":/image/defaultAvatar.png");
            } else {
                // 设置群聊头像
                avatar = QIcon(":/image/groupAvatar.png");
            }
        } else {
            avatar = makeIcon(chatSessionInfo.avatar());
        }
    }
};
```

设计数据中心类

通过 `model/datacenter.h` 中的 `DataCenter` 类来管理所有客户端需要的数据. 这是一个单例类.

```
class DataCenter : public QObject {
    Q_OBJECT
private:
    // 登录会话id
    // 这个信息是登录成功后，服务器返回的.
    // 返回之后这个数据需要被客户端持久化保存在文件中.
    // 后续每次启动，都从文件中拿到这个内容.
    // 当用户显式点击 "退出登录"，则删除这个信息.
    QString loginSessionId = "";

    // 当前用户信息
    UserInfo* myself = nullptr;

    // 当前用户的好友列表
    QList<UserInfo>* friendList = nullptr;

    // 当前用户的会话列表
    QList<ChatSessionInfo>* chatSessionList = nullptr;
```

```
    // 当前用户选择的消息会话 id
    QString currentChatSessionId = "";

    // 待处理的好友申请列表
    QList<UserInfo>* applyList = nullptr;

    // 每个会话中的用户列表
    QHash<QString, QList<UserInfo>>* memberList = nullptr;

    // 当前用户的消息内容，key 为 ChatSessionInfo 的 chatSessionId
    QHash<QString, QList<Message>>* recentMessages = nullptr;

    // 未读消息数目的统计，key 为 ChatSessionInfo 的 chatSessionId，value 为未读消息个数
    QHash<QString, int>* unreadMessageCount = nullptr;

    // 用户搜索结果
    QList<UserInfo>* searchUserResult = nullptr;

    // 当前历史消息搜索结果
    QList<Message>* searchMessageResult = nullptr;


    // 当前短信验证码的验证 id
    QString currentVerifyCodeId = "";

    // 持有网络通信客户端
    NetClient netClient;

    // 构造单例模式
    static DataCenter* instance;
    DataCenter();

public:
    static DataCenter* getInstance();
    ~DataCenter();

    // 初始化数据文件
    void initDataFile();
    // 保存必要的数据到文件
    void saveDataFile();
    // 从文件加载必要的数据
    void loadDataFile();
}
```

```cpp
// 初始化实例
DataCenter* DataCenter::instance = nullptr;

DataCenter::DataCenter() : netClient(this)
{
    // 这几个 哈希表 提前把对象 new 好
    recentMessages = new QHash<QString, QList<Message>>();
    memberList = new QHash<QString, QList<UserInfo>>();
    unreadMessageCount = new QHash<QString, int>();

    loadDataFile();
}


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

DataCenter::~DataCenter()
{
    delete myself;
    delete friendList;
    delete chatSessionList;
    delete memberList;
    delete recentMessages;
    delete searchMessageResult;
    delete unreadMessageCount;
    delete searchUserResult;
}
```

> NetClient 的实现后续完成.

数据持久化

使用文件存储 sessionId 和 未读消息信息.

```cpp
void DataCenter::initDataFile()
{
    // 拼装一个文件目录. 使用 appData 作为目录
```

```
    QString filePath = QStandardPaths::writableLocation(QStandardPaths::AppDataLocation) + "/ChatClient.json";

    // 打开文件
    QFile file(filePath);
    if (!file.open(QIODevice::WriteOnly | QIODevice::Text)) {
        qCritical() << TAG << "Cannot open file:" << file.errorString() << filePath;
        return;
    }

    // 写入初始内容
    QString data = "{\n\n}";
    file.write(data.toUtf8());
    file.close();
}

void DataCenter::saveDataFile()
{
    // 拼装一个文件目录. 使用 appData 作为目录
    QString filePath = QStandardPaths::writableLocation(QStandardPaths::AppDataLocation) + "/ChatClient.json";

    // 打开文件
    QFile file(filePath);
    if (!file.open(QIODevice::WriteOnly | QIODevice::Text)) {
        qCritical() << TAG << "Cannot open file:" << file.errorString() << filePath;
        return;
    }

    // 写入 loginSessionId
    QJsonObject jsonObj;
    jsonObj["loginSessionId"] = loginSessionId;

    // 写入未读消息次数
    QJsonObject jsonUnread;
    for (auto it = unreadMessageCount->begin(); it != unreadMessageCount->end(); ++it) {
        jsonUnread[it.key()] = it.value();
    }
    jsonObj["unread"] = jsonUnread;

    // 写入文件
    QJsonDocument jsonDoc(jsonObj);
```

```
    QString s = jsonDoc.toJson();
    // LOG() << "saveData: s=" << s;
    file.write(s.toUtf8());

    // 关闭文件. QFile 能在析构的时候自动关闭
    file.close();
}

void DataCenter::loadDataFile()
{
    // 拼装一个文件目录. 使用 appData 作为目录
    QString basePath =
QStandardPaths::writableLocation(QStandardPaths::AppDataLocation);
    QString filePath = basePath + "/ChatClient.json";

    // 如果目录不存在, 就先创建目录
    QDir dir;
    if (!dir.exists(basePath)) {
        dir.mkpath(basePath);
    }

    // 如果文件不存在, 就先创建文件
    QFileInfo fileInfo(filePath);
    if (!fileInfo.exists()) {
        initDataFile();
    }

    // 读方式打开文件.
    QFile file(filePath);
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        qCritical() << TAG << "Cannot open file:" << file.errorString() <<
filePath;
        return;
    }

    // 读取文件内容, 解析为 JSON
    QJsonDocument jsonDoc = QJsonDocument::fromJson(file.readAll());
    if (jsonDoc.isNull()) {
        qCritical() << TAG << "Invalid JSON format";
        return;
    }

    // 获取到 JSON 中的属性
    QJsonObject jsonObj = jsonDoc.object();
    loginSessionId = jsonObj["loginSessionId"].toString();

    this->unreadMessageCount->clear();
```

```
    QJsonObject unread = jsonObj["unread"].toObject();
    for (auto it = unread.constBegin(); it != unread.constEnd(); ++it) {
        // qDebug() << "key:" << it.key() << " value:" << it.value().toInt();
        (*this->unreadMessageCount)[it.key()] = it.value().toInt();
    }

    // 检查读到的内容是否正确
    if (loginSessionId == "") {
        qCritical() << TAG << "读取到的 sessionId 为空!";
        return;
    }

    // 关闭文件. QFile 能在析构的时候自动关闭
    // file.close();
}
```

> 未读消息的实现放到后面完成.

网络通信

定义 NetClient 类

通过 `network/NetClient.h` 中的 `NetClient` 类来管理所有的和服务器通信的内容.

`NetClient` 内部又分成 `httpClient` 和 `websocketClient` 两个部分.

`DataCenter` 中会持有 `NetClient` 的指针.

CMakeLists.txt 中添加

```
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets Network
WebSockets Protobuf)

target_link_libraries(ChatClient PRIVATE Qt${QT_VERSION_MAJOR}::Widgets
Qt6::Network Qt6::WebSockets Qt6::Protobuf )
```

```
class NetClient : public QObject {
    Q_OBJECT

private:
#if CONNECT_TEST_SERVER
```

```
    const QString HTTP_URL = "http://127.0.0.1:8000";
    const QString WEBSOCKET_URL = "ws://127.0.0.1:8001/ws";
#else
    const QString HTTP_URL = "http://47.108.169.126:9001";
    const QString WEBSOCKET_URL = "ws://47.108.169.126:9000/ws";
#endif

    QNetworkAccessManager httpClient;
    QWebSocket websocketClient;

    model::DataCenter* dataCenter;

    // 序列化器
    QProtobufSerializer serializer;
}
```

```
NetClient::NetClient(DataCenter* dataCenter) : dataCenter(dataCenter)
{
}
```

引入 HTTP

1) 进行网络测试

```
void NetClient::ping()
{
    QNetworkRequest httpReq;
    httpReq.setUrl(QUrl(HTTP_URL + "/ping"));
    QNetworkReply* httpResp = httpClient.get(httpReq);
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        if (httpResp->error() != QNetworkReply::NoError) {
            LOG()<< "HTTP 请求失败: " << httpResp->errorString();
            return;
        }
        // 获取到响应 body
        QByteArray respBody = httpResp->readAll();
        LOG() << "resp=" << QString(respBody);
        httpResp->deleteLater();
    });
}
```

2) 封装构造 HTTP 请求和处理响应

```
// 通过这个方法统一构造 HTTP 请求.
// apiPath 应该要以 / 开头
QNetworkReply* sendHttpRequest(const QString& apiPath, const QByteArray& body)
{
    // 1. 构造 http 请求
    QNetworkRequest httpReq;
    httpReq.setUrl(QUrl(HTTP_URL + apiPath));
    httpReq.setHeader(QNetworkRequest::ContentTypeHeader, "application/x-
protobuf");

    // 2. 发送 http 请求
    QNetworkReply* httpResp = httpClient.post(httpReq, body);
    return httpResp;
}

// 通过这个方法统一处理 HTTP 响应.
// 这个是通用接口
template<typename T>
std::shared_ptr<T> handleHttpResponse(QNetworkReply* httpResp) {
    if (httpResp->error() != QNetworkReply::NoError) {
        LOG()<< "HTTP 请求失败: " << httpResp->errorString();
        httpResp->deleteLater();
        return std::shared_ptr<T>();
    }
    // a) 获取到响应 body
    QByteArray respBody = httpResp->readAll();

    // b) 反序列化
    std::shared_ptr<T> respObj = std::make_shared<T>();
    respObj->deserialize(&serializer, respBody);

    // c) 判定响应结果是否正确
    if (!respObj->success()) {
        LOG() << "request_id=" << respObj->requestId() << ", errmsg=" <<
respObj->errmsg();
        httpResp->deleteLater();
        return std::shared_ptr<T>();
    }

    // c) 打印日志
    LOG() << "request_id=" << respObj->requestId() << " 响应完成";

```

```
    // d) 释放响应对象
    httpResp->deleteLater();

    return respObj;
}

// 这个接口能够获取到具体的原因，并进行判定
template<typename T>
std::shared_ptr<T> handleHttpResponseWithReason(QNetworkReply* httpResp) {
    if (httpResp->error() != QNetworkReply::NoError) {
        LOG()<< "HTTP 请求失败: " << httpResp->errorString();
        httpResp->deleteLater();
        return std::shared_ptr<T>();
    }
    // a) 获取到响应 body
    QByteArray respBody = httpResp->readAll();

    // b) 反序列化
    std::shared_ptr<T> respObj = std::make_shared<T>();
    respObj->deserialize(&serializer, respBody);

    // c) 判定响应结果是否正确
    if (!respObj->success()) {
        LOG() << "request_id=" << respObj->requestId() << ", errmsg=" <<
respObj->errmsg();
        httpResp->deleteLater();
        return respObj;
    }

    // c) 打印日志
    LOG() << "request_id=" << respObj->requestId() << " 响应完成";

    // d) 释放响应对象
    httpResp->deleteLater();

    return respObj;
}
```

引入 websocket

Websocket 在主窗口加载后, 才和服务器建立连接. 并且在建立连接后给服务器发送一个 认证请求 之后, 才能收到后续数据.

1) 初始化 websocket

```
void NetClient::initWebsocket()
{
    // 1. 连接信号槽
    connect(&websocketClient, &QWebSocket::connected, this, [=]() {
        LOG() << "connected";
        // 通过 websocket 发送身份信息
        sendAuthentication()
    });
    connect(&websocketClient, &QWebSocket::disconnected, this, [=]() {
        LOG() << "disconnected";
    });
    connect(&websocketClient, &QWebSocket::errorOccurred, this, [=]
(QAbstractSocket::SocketError error) {
        LOG() << "error: " << error;
    });
    connect(&websocketClient, &QWebSocket::textMessageReceived, this, [=](const
 QString &message) {
        LOG() << "websocket text receive: " << message;
    });
    connect(&websocketClient, &QWebSocket::binaryMessageReceived, this, [=]
(const QByteArray& byteArray) {
        LOG() << "websocket binary receive!";
        // 通过二进制方式收到响应数据，响应数据是 pb 的 NotifyMessage 结构
        std::shared_ptr<bite_im::NotifyMessage> respObj =
std::make_shared<bite_im::NotifyMessage>();
        respObj->deserialize(&serializer, byteArray);
        handleWsResponse(respObj);
    });

    // 2. 建立 websocket 连接
    websocketClient.open(WEBSOCKET_URL);
}
```

2) 初始化身份信息

```
void NetClient::sendAuthentication()
{
    if (!websocketClient.isValid()) {
        // 当前连接无效
        LOG() << "websocket 无效!";
```

```
        return;
    }
    bite_im::ClientAuthenticationReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(dataCenter->getLoginSessionId());
    QByteArray body = req.serialize(&serializer);
    websocketClient.sendBinaryMessage(body);
    LOG() << "[WS身份认证] requestId=" << req.requestId() << ", loginSessionId="
 << req.sessionId();

```

3) 搭建 websocket 消息推送的逻辑

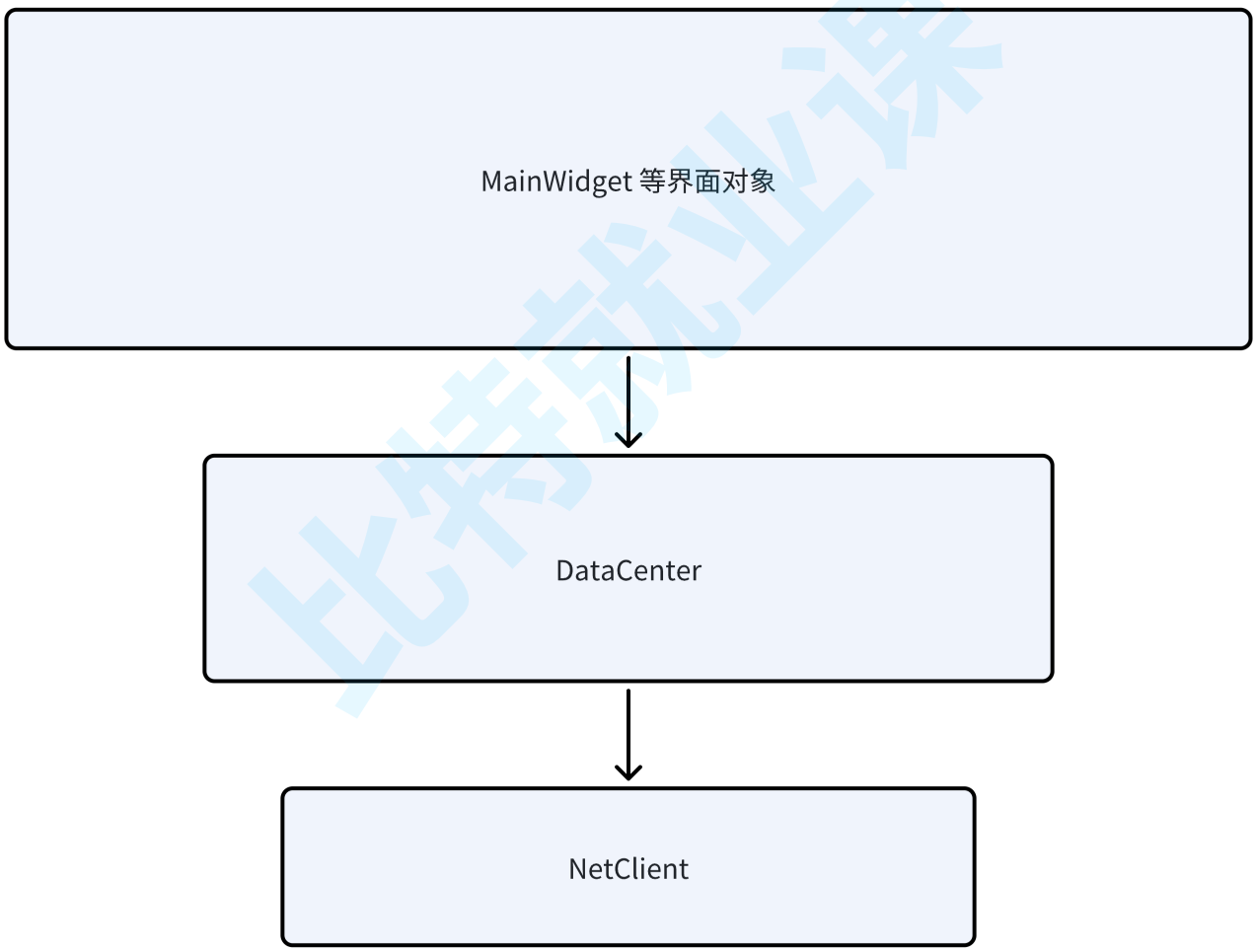
```
void NetClient::handleWsResponse(std::shared_ptr<bite_im::NotifyMessage>
respObj)
{
    if (respObj->notifyType() ==
bite_im::NotifyTypeGadget::NotifyType::CHAT_MESSAGE_NOTIFY) {
        // 收到消息. 由于有 "创建会话" 的操作, 所以这里收到消息时会话是一定存在的.
        model::Message message;
        message.load(respObj->newMessageInfo().messageInfo());
        handleWsMessage(message);
    } else if (respObj->notifyType() ==
bite_im::NotifyTypeGadget::NotifyType::CHAT_SESSION_CREATE_NOTIFY) {
        // 这个响应在每次创建会话的时候, 都会通知给会话的所有客户端.
        model::ChatSessionInfo chatSessionInfo;
        chatSessionInfo.load(respObj->newChatSessionInfo().chatSessionInfo());
        handleWsSessionCreate(chatSessionInfo);
    } else if (respObj->notifyType() ==
bite_im::NotifyTypeGadget::NotifyType::FRIEND_ADD_APPLY_NOTIFY) {
        model::UserInfo userInfo;
        userInfo.load(respObj->friendAddApply().userInfo());
        handleWsAddFriendApplyReq(userInfo);
    } else if (respObj->notifyType() ==
bite_im::NotifyTypeGadget::NotifyType::FRIEND_ADD_PROCESS_NOTIFY) {
        model::UserInfo userInfo;
        userInfo.load(respObj->friendProcessResult().userInfo());
        handleWsAddFriendResp(userInfo, respObj-
>friendProcessResult().agree());
    } else if (respObj->notifyType() ==
bite_im::NotifyTypeGadget::NotifyType::FRIEND_REMOVE_NOTIFY) {
        const QString& userId = respObj->friendRemove().userId();
        handleWsRemoveFriend(userId);
```

```
24      } else {
25          LOG() << "unknown notifyType! notifyType=" << respObj->notifyType();
26      }
27  }
```

4) 针对上述每种消息的处理实现, 后续再进一步完成

小结

三个层次



NetClient 从网络拿到数据, 只交给 DataCenter

通过网络收到的数据, DataCenter 负责发送信号给 MainWidget, 从而异步通知界面更新.

搭建测试服务器

以 "获取个人信息" 为例, 先把流程跑通.

再逐渐加上其他的接口. 比如 "获取好友列表", "获取会话列表", "获取指定会话的消息列表".

直接基于 Qt 在 windows 上搭建 HTTP 服务器.

1) 创建项目

基于 CMake 创建 Qt 项目

> 虽然使用控制台项目也可以(创建成 Qt Core Application), 但是使用图形界面更合适一些.
>
> 尤其是后面构造一些测试数据, 图形界面更方便进行操作.
>
> 比如在界面上提供不同的按钮, 按下不同按钮就可以给客户端推送不同的数据.

```
cmake_minimum_required(VERSION 3.16)

find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets HttpServer
WebSockets Protobuf)

file(GLOB PB_FILES "../ChatClient/proto/*.proto")

qt_add_protobuf(ChatServerMock PROTO_FILES ${PB_FILES})

target_link_libraries(ChatServerMock PRIVATE Qt${QT_VERSION_MAJOR}::Widgets
Qt6::HttpServer Qt6::WebSockets Qt6::Protobuf)
```

2) 引入 http

```
class HttpServer : public QObject {
    Q_OBJECT

private:
    QHttpServer httpServer;

    // 序列化器
    QProtobufSerializer serializer;

    static HttpServer* instance;
    HttpServer() {}
public:
    static HttpServer* getInstance();
```

```
    bool init();
}
```

```
HttpServer* HttpServer::instance = nullptr;

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

bool HttpServer::init() {
    // 返回值表示绑定成功的端口号的值.
    int ret = httpServer.listen(QHostAddress::Any, 8000);

    httpServer.route("/ping", [](const QHttpServerRequest& req) {
        qDebug() << "[http] 收到 ping 请求";
        return "pong";
    })

    return ret == 8000;
}
```

3) 引入 websocket

```
class WebsocketServer : public QObject {
    Q_OBJECT

private:
    QWebSocketServer websocketServer;

    // 序列化器
    QProtobufSerializer serializer;

    static WebsocketServer* instance;
    WebsocketServer() : websocketServer("Qt WebSocket Server",
QWebSocketServer::NonSecureMode) { }
public:
    static WebsocketServer* getInstance();
```

```
    bool init()
}
```

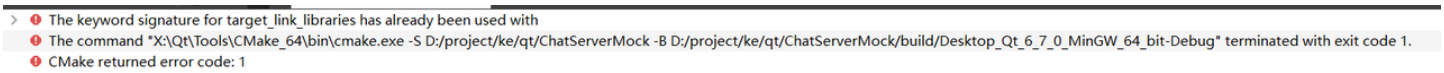
```
bool WebsocketServer::init() {
    bool ok = websocketServer.listen(QHostAddress::Any, 8001);

    QByteArray avatar = loadImageToByteArray(":/image/defaultAvatar.png");

    connect(&websocketServer, &QWebSocketServer::newConnection, this, [=](){
        QWebSocket *socket = websocketServer.nextPendingConnection();
        qDebug() << "[websocket] New Connection";

        connect(socket, &QWebSocket::disconnected, this, [=]() {
            qDebug() << "[websocket] Disconnected";

            socket->deleteLater();
        });

        connect(socket, &QWebSocket::errorOccurred, this, [=]
(QAbstractSocket::SocketError error) {
            qDebug() << "[websocket] error: " << error;
        });

        connect(socket, &QWebSocket::textMessageReceived, this, [=](const
QString& message) {
            qDebug() << "[websocket] Received message:" << message;
            if (message == "ping") {
                socket->sendTextMessage("pong");
                return;
            }
        });
    }
    return ok;
}
```

4) 引入 protobuf

```
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets HttpServer
WebSockets Protobuf)

file(GLOB PB_FILES "../ChatClient/proto/*.proto")
```

> 直接从 ChatClient 项目中引入 proto 文件.

如果出现下列报错:



则给 `target_link_libraries` 引入 `PRIVATE`

从

```
target_link_libraries(ChatServerMock Qt${QT_VERSION_MAJOR}::Core Qt6::Network
Qt6::WebSockets)
```

修改为

```
target_link_libraries(ChatServerMock PRIVATE Qt${QT_VERSION_MAJOR}::Core
Qt6::Network Qt6::WebSockets)
```

5) 编写工具函数和构造数据函数

工具函数

```
// 获取到图片的二进制数据
static inline QByteArray loadImageToByteArray(const QString& fileName) {
    // 1. 加载为 QImage
    QImage image(fileName);
    if (image.isNull()) {
        // 如果图片无法加载，返回空的字节数组
        return QByteArray();
    }
    // 2. 转换成 QPixmap
    QPixmap pixmap = QPixmap::fromImage(image);

    // 3. 获取图片类型，是 png 还是 jpg 还是啥别的
    QFileInfo fileInfo(fileName);
    QString type = fileInfo.suffix();

```

```
    // 4. 进行转换
    QByteArray byteArray;
    QBuffer buffer(&byteArray);
    bool ok = false;
    if (type == "png") {
        // 此处的 save 支持 BMP, JPG, JPEG, PNG
        ok = pixmap.save(&buffer, "PNG");
    } else if (type == "jpg" || type == "jpeg") {
        ok = pixmap.save(&buffer, "JPG");
    } else if (type == "bmp") {
        ok = pixmap.save(&buffer, "BMP");
    } else {
        return QByteArray();
    }
    if (!ok) {
        // 如果保存失败，返回空的字节数组
        return QByteArray();
    }
    return byteArray;
}

// 获取到文件的二进制内容
static inline QByteArray loadFileToByteArray(const QString& filename) {
    QFile file(filename);
    bool ok = file.open(QFile::ReadOnly);
    if (!ok) {
        LOG() << "文件读取失败!";
        return QByteArray();
    }
    QByteArray content = file.readAll();
    file.close();
    return content;
}
```

构造数据函数

```
// 构造一个 UserInfo
bite_im::UserInfo makeUserInfo(int index, const QByteArray& avatar) {
    bite_im::UserInfo userInfo;
    userInfo.setUserId(QString::number(1000 + index));
    userInfo.setNickname("张三" + QString::number(index));
    userInfo.setDescription("这是个性签名" + QString::number(index));
    userInfo.setPhone("18612345678");
    userInfo.setAvatar(avatar);
```

```
    return userInfo;
}

bite_im::UserInfo makeEmptyUserInfo(const QByteArray& avatar) {
    bite_im::UserInfo userInfo;
    userInfo.setUserId("");
    userInfo.setNickname("");
    userInfo.setDescription("");
    userInfo.setPhone("");
    userInfo.setAvatar(avatar);
    return userInfo;
}

// 构造一个文本消息对象
bite_im::MessageInfo makeMessageInfo(int index, const QString& chatSessionId,
const QByteArray& avatar) {
    bite_im::MessageInfo messageInfo;
    messageInfo.setChatSessionId(chatSessionId);
    messageInfo.setTimestamp(QDateTime::currentMSecsSinceEpoch() / 1000);
    messageInfo.setMessageId(QString::number(3000 + index));

    bite_im::UserInfo sender = makeUserInfo(index, avatar);
    messageInfo.setSender(sender);

    bite_im::MessageContent messageContent;

messageContent.setMessageType(bite_im::MessageTypeGadget::MessageType::STRING);
    bite_im::StringMessageInfo stringMessageInfo;
    stringMessageInfo.setContent("这是一条消息" + QString::number(index) + ", "
+ chatSessionId);
    messageContent.setStringMessage(stringMessageInfo);
    messageInfo.setMessage(messageContent);

    return messageInfo;
}

bite_im::MessageInfo makeEmptyMessageInfo(const QString& chatSessionId, const
QByteArray& avatar) {
    bite_im::MessageInfo messageInfo;
    messageInfo.setChatSessionId(chatSessionId);
    messageInfo.setTimestamp(QDateTime::currentMSecsSinceEpoch() / 1000);
    messageInfo.setMessageId("");

    bite_im::UserInfo sender = makeEmptyUserInfo(avatar);
    messageInfo.setSender(sender);

    bite_im::MessageContent messageContent;
```

```
52
   messageContent.setMessageType(bite_im::MessageTypeGadget::MessageType::STRING);
53      bite_im::StringMessageInfo stringMessageInfo;
54      stringMessageInfo.setContent("");
55      messageContent.setStringMessage(stringMessageInfo);
56      messageInfo.setMessage(messageContent);
57
58      return messageInfo;
59  }
```

6) 验证网络连通性

修改客户端的 `main.cpp` , 添加网络测试代码.

```
1  // 测试网络联通
2  #if TEST_NETWORK
3      network::NetClient netClient(nullptr);
4      netClient.ping();
5  #endif
```

运行客户端, 连接测试服务器, 并验证是否 HTTP / Websocket 网络能连通.

7) 网络通信注意事项

1) 不能使用两个 Qt Creator 分别启动服务器和客户端. 后启动的程序 qDebug 会失效.

> 提示: "无法获取调试输出".

2) websocket 客户端代码要编写完整, 再连接服务器. 否则会直接崩溃, 而没有任何具体提示.

3) 一定要确保 websocket 的 `connected` 信号触发之后, 才能 `sendTextMessage` . 否则不会有任何提示, 但是消息发送不成功.

> Qt 这一套信号槽, 用起来和 Node.js 非常相似的. 时刻注意 "异步" 的问题.

4) 每次更新完 PB, 一定要记得 服务器 和 客户端 都需要重新编译运行!!

否则程序会出现不可预期的错误.

主界面逻辑 (1)

获取个人信息

1) 客户端发送请求

a) 编写 `MainWidget::initData` 函数

```
connect(dataCenter, &DataCenter::getMyselfDone, this, [=]() {
    const auto* myself = dataCenter->getMyself();
    this->userAvatar->setIcon(myself->avatar);
});
dataCenter->getMyselfAsync();
```

b) 编写 `DataCenter::getMyselfAsync`

```
void DataCenter::getMyselfAsync()
{
    netClient.getMyself(loginSessionId);
}
```

c) 编写 `NetClient::getMyself`

接口定义

```
//个人信息获取-这个只用于获取当前登录用户的信息
//  客户端传递的时候只需要填充session_id即可
//其他个人/好友信息的获取在好友操作中完成
message GetUserInfoReq {
    string request_id = 1;
    optional string user_id = 2;
    optional string session_id = 3;
}
message GetUserInfoRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
```

```
    UserInfo user_info = 4;
}
```

代码实现

```
void NetClient::getMyself(const QString &loginSessionId)
{
    // 1. 通过 proto 构造 body
    bite_im::GetUserInfoReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    QByteArray body = req.serialize(&serializer);
    LOG() << "[获取个人信息] requestId=" << req.requestId() << ", sessionId=" << loginSessionId;

    // 2. 发送 http 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/user/get_user_info", body);

    // 3. 处理 http 响应
    connect(httpResp, &QNetworkReply::finished, this, [=] () {
        // a) 解析响应
        auto userInfoRsp = this->handleHttpResponse<bite_im::GetUserInfoRsp>(httpResp);
        if (!userInfoRsp) {
            // shared_ptr 可以直接通过 bool 运算判定是否是空对象.
            return;
        }
        // b) 设置 DataCenter 中的数据
        dataCenter->resetMyself(userInfoRsp);
        // c) 发送信号
        emit dataCenter->getMyselfDone();
    });
}
```

2) 客户端处理响应

a) 实现 `DataCenter::resetMyself`

```
void DataCenter::resetMyself(std::shared_ptr<bite_im::GetUserInfoRsp>
```

```
    userInfoResp)
2   {
3       if (myself == nullptr) {
4           myself = new UserInfo();
5       }
6       const bite_im::UserInfo& userInfo = userInfoResp->userInfo();
7       myself->load(userInfo);
8   }
```

b) 定义 `DataCenter` 信号

```
1   signals:
2       // 获取个人信息完成
3       void getMyselfDone();
```

3) 服务器处理请求

a) 编写 `HttpServer::init` 注册路由

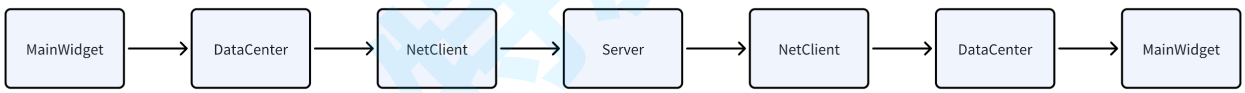
```
1   httpServer.route("/service/user/get_user_info", [=](const QHttpServerRequest&
    req) {
2       return this->getUserInfo(req);
3   });
```

b) 实现处理函数

```
1   QHttpServerResponse HttpServer::getUserInfo(const QHttpServerRequest &req)
2   {
3       // 解析请求
4       bite_im::GetUserInfoReq pbReq;
5       pbReq.deserialize(&serializer, req.body());
6       LOG() << "[REQ 获取用户信息] request_id=" << pbReq.requestId() << ",
    sessionId=" << pbReq.sessionId();
7
8       // 构造响应
9       bite_im::GetUserInfoRsp pbRsp;
10      pbRsp.setRequestId(pbReq.requestId());
```

```
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");

    bite_im::UserInfo userInfo;
    userInfo.setUserId("1234");
    userInfo.setNickname("张三");
    userInfo.setDescription("这是个性签名");
    userInfo.setPhone("18612345678");
    userInfo.setAvatar(loadImageToByteArray(":/image/defaultAvatar.png"));
    pbRsp.setUserInfo(userInfo);

    // 序列化
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

4) 整体流程小结



后续每个前后端交互都是类似的逻辑链路.

获取好友列表

1) 客户端发送请求

a) 编写 `MainWidget::initData`

```
//////////////////////////////////////////
/// 获取好友列表
//////////////////////////////////////////
loadFriendList();
```

b) 实现 `loadFriendList`

```
void MainWidget::loadFriendList()
{
#if LOAD_DATA_FROM_NETWORK
    // 1. 获取到实例
    DataCenter* dataCenter = DataCenter::getInstance();

    if (dataCenter->getFriendList() != nullptr) {
        // 2. 尝试从内存读数据
        updateFriendList();
    } else {
        // 3. 尝试从网络加载数据
        connect(dataCenter, &DataCenter::getFriendListDone, this,
&MainWidget::updateFriendList, Qt::UniqueConnection);
        dataCenter->getFriendListAsync();
    }
#endif
}
```

注意:

> loadFriendList 不仅仅会在初始化时调用, 也会在后续切换标签页时调用.
>
> 多次 connect 虽然不会报错, 但是会导致槽函数被一个信号触发多次.
>
> 可以在 connect 的时候使用 `Qt::UniqueConnection` 参数(第五个参数), 避免触发多次的情况.

c) 实现 `DataCenter` 中的 `getFriendList`, `getFriendListAsync`

```
QList<UserInfo>* DataCenter::getFriendList()
{
    return friendList;
}

void DataCenter::getFriendListAsync()
{
    netClient.getFriendList(loginSessionId);
}
```

d) 实现 `NetClient::getFriendList`

接口定义

```
//--------------------------------------
//好友列表获取
message GetFriendListReq {
    string request_id = 1;
    optional string user_id = 2;
    optional string session_id = 3;
}
message GetFriendListRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    repeated UserInfo friend_list = 4;
}
```

代码实现

```
void NetClient::getFriendList(const QString &loginSessionId)
{
    // 1. 通过 proto 构造 body
    bite_im::GetFriendListReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    QByteArray body = req.serialize(&serializer);
    LOG() << "[获取好友列表] requestId=" << req.requestId() << ", sessionId=" << loginSessionId;

    // 2. 发送 http 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/friend/get_friend_list", body);

    // 3. 处理 http 响应
    connect(httpResp, &QNetworkReply::finished, this, [=] () {
        // a) 解析响应
        auto friendListRsp = this->handleHttpResponse<bite_im::GetFriendListRsp>(httpResp);
        if (!friendListRsp) {
            // shared_ptr 可以直接通过 bool 运算判定是否是空对象.
            return;
        }
        // b) 设置 DataCenter 中的数据
        dataCenter->resetFriendList(friendListRsp);
```

```
        // c) 发送信号
        emit dataCenter->getFriendListDone();
    });
}
```

2) 客户端处理响应

a) 编写 `DataCenter::resetFriendList`

```
void DataCenter::resetFriendList(std::shared_ptr<bite_im::GetFriendListRsp> friendListRsp)
{
    // 1. 清空原来的好友列表数据
    if (friendList == nullptr) {
        friendList = new QList<UserInfo>();
    }
    friendList->clear();

    // 2. 遍历响应结果，添加到数据结构中. Qt 的 protobuf 接口和原生 protobuf 存在差别.
    auto& friendListPB = friendListRsp->friendList();
    for (auto& f : friendListPB) {
        UserInfo userInfo;
        userInfo.load(f);
        friendList->push_back(userInfo);
    }
}
```

b) 定义 DataCenter 信号

```
void getFriendListDone()
```

c) 实现 `MainWidget::updateFriendList`

```
void MainWidget::updateFriendList()
{
```

```
    if (getActiveTab() != FRIEND_LIST) {
        // 如果当前选中的标签页不是好友列表，就不更新界面.
        return;
    }
    DataCenter* dataCenter = DataCenter::getInstance();
    const auto* friendList = dataCenter->getFriendList();

    // 1) 清空之前界面的好友列表
    sessionFriendArea->clear();

    // 2) 添加新的好友到界面上
    for (const auto& f : *friendList) {
        sessionFriendArea->addItem(FriendItemType, f.userId, f.avatar,
f.nickname, f.description);
    }
}
```

3) 服务器处理请求

a) 编写 `HttpServer::init` 注册路由

```
httpServer.route("/service/friend/get_friend_list", [=](const
QHttpServerRequest& req) {
    return this->getFriendList(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::getFriendList(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::GetFriendListReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 获取好友列表] request_id=" << pbReq.requestId() << ",
sessionId=" << pbReq.sessionId();

    // 构造响应
    bite_im::GetFriendListRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
```

```
    pbRsp.setErrmsg("");

    QByteArray avatar = loadImageToByteArray(":/image/defaultAvatar.png");

    for (int i = 0; i < 30; i++) {
        bite_im::UserInfo userInfo = makeUserInfo(i, avatar);
        pbRsp.friendList().push_back(userInfo);
    }

    // 序列化
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

4) 整体流程小结

再次强调, 下列流程是前后端交互最核心流程.

获取会话列表

1) 客户端发送请求

a) 编写 `MainWidget::init`

```
//////////////////////////////////////////
/// 获取会话列表
//////////////////////////////////////////
loadSessionList();
```

b) 编写 `loadSessionList()`

```
void MainWidget::loadSessionList()
```

```
{
#if LOAD_DATA_FROM_NETWORK
    // 1. 获取到 DataCenter 实例
    DataCenter* dataCenter = DataCenter::getInstance();

    if (dataCenter->getChatSessionList() != nullptr) {
        // 2. 尝试从内存读数据
        updateChatSessionList();
    } else {
        // 3. 内存没有数据，从网络获取
        //    为了避免触发多次槽函数，使用 Qt::UniqueConnection 来处理
        connect(dataCenter, &DataCenter::DataCenter::getChatSessionListDone,
                this, &MainWidget::updateChatSessionList,
Qt::UniqueConnection);
        dataCenter->getChatSessionListAsync();
    }
#endif
}
```

c) 编写 `DataCenter`

```
QList<ChatSessionInfo>* DataCenter::getChatSessionList()
{
    return chatSessionList;
}

void DataCenter::getChatSessionListAsync()
{
    netClient.getChatSessionList(loginSessionId);
}
```

d) 编写 `NetClient`

接口定义

```
//--------------------------------------
//会话列表获取
message GetChatSessionListReq {
    string request_id = 1;
    optional string session_id = 2;
    optional string user_id = 3;
```

```
}
message GetChatSessionListRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    repeated ChatSessionInfo chat_session_info_list = 4;
}
```

函数实现

```
void NetClient::getChatSessionList(const QString &loginSessionId)
{
    // 1. 通过 proto 构造 body
    bite_im::GetChatSessionListReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    QByteArray body = req.serialize(&serializer);
    LOG() << "[获取会话列表] requestId=" << req.requestId() << ", sessionId=" << loginSessionId;

    // 2. 发送 http 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/friend/get_chat_session_list", body);

    // 3. 处理 http 响应
    connect(httpResp, &QNetworkReply::finished, this, [=] () {
        // a) 解析响应
        auto chatSessionListRsp = this->handleHttpResponse<bite_im::GetChatSessionListRsp>(httpResp);
        if (!chatSessionListRsp) {
            // shared_ptr 可以直接通过 bool 运算判定是否是空对象.
            return;
        }
        // b) 设置 DataCenter 中的数据
        dataCenter->resetChatSessionList(chatSessionListRsp);
        // c) 发送信号
        emit dataCenter->getChatSessionListDone();
    });
}
```

2) 客户端处理响应

a) 修改 `DataCenter`

```
void
DataCenter::resetChatSessionList(std::shared_ptr<bite_im::GetChatSessionListRsp
> chatSessionListRsp)
{
    // 1. 清空原来的会话列表
    if (chatSessionList == nullptr) {
        chatSessionList = new QList<ChatSessionInfo>();
    }
    chatSessionList->clear();

    // 2. 遍历响应结果，添加到数据结构中.
    auto& chatSessionListPB = chatSessionListRsp->chatSessionInfoList();
    for (auto& c : chatSessionListPB) {
        ChatSessionInfo chatSessionInfo;
        chatSessionInfo.load(c);
        chatSessionList->push_back(chatSessionInfo);
    }
}
```

b) 定义 `DataCenter` 信号

```
// 获取会话列表完成
void getChatSessionListDone();
```

c) 实现 `MainWidget::updateChatSessionList`

```
void MainWidget::updateChatSessionList()
{
    // 当前标签页不是会话列表，则不必更新界面
    if (getActiveTab() != SESSION_LIST) {
        return;
    }

    DataCenter* dataCenter = DataCenter::getInstance();
    const auto* chatSessionList = dataCenter->getChatSessionList();

    // 1) 清空之前界面的会话列表
```

```
    sessionFriendArea->clear();

    // 2) 添加新的会话到界面上
    for (const auto& c : *chatSessionList) {
        if (c.prevMessage.messageType == model::TEXT_TYPE) {
            sessionFriendArea->addItem(SessionItemType, c.chatSessionId,
c.avatar, c.chatSessionName, QString(c.prevMessage.content));
        } else if (c.prevMessage.messageType == model::FILE_TYPE) {
            sessionFriendArea->addItem(SessionItemType, c.chatSessionId,
c.avatar, c.chatSessionName, "[文件]");
        } else if (c.prevMessage.messageType == model::IMAGE_TYPE) {
            sessionFriendArea->addItem(SessionItemType, c.chatSessionId,
c.avatar, c.chatSessionName, "[图片]");
        } else if (c.prevMessage.messageType == model::SPEECH_TYPE) {
            sessionFriendArea->addItem(SessionItemType, c.chatSessionId,
c.avatar, c.chatSessionName, "[语音]");
        } else {
            qCritical() << TAG << "messageType 错误! messageType=" <<
c.prevMessage.messageType;
        }
    }
}
```

3) 服务器处理请求

a) 编写 `HttpServer::init` 注册路由

```
httpServer.route("/service/friend/get_chat_session_list", [=](const
QHttpServerRequest& req) {
    return this->getChatSessionList(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::getChatSessionList(const QHttpServerRequest
&req)
{
    // 解析请求
    bite_im::GetChatSessionListReq pbReq;
    pbReq.deserialize(&serializer, req.body());
```

```
    LOG() << "[REQ 获取会话列表] request_id=" << pbReq.requestId() << ", sessionId=" << pbReq.sessionId();

    // 构造响应
    bite_im::GetChatSessionListRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");

    QByteArray avatar = loadImageToByteArray(":/image/defaultAvatar.png");

    for (int i = 0; i < 30; i++) {
        bite_im::ChatSessionInfo chatSessionInfo;
        chatSessionInfo.setChatSessionId(QString::number(2000 + i));
        chatSessionInfo.setChatSessionName("张三" + QString::number(i));
        if (i == 0) {
            // 把这个元素设置成群聊
            chatSessionInfo.setSingleChatFriendId("");
        } else {
            // 其他元素设置为单聊
            chatSessionInfo.setSingleChatFriendId(QString::number(1000 + i));
        }
        chatSessionInfo.setAvatar(avatar);

        bite_im::MessageInfo messageInfo = makeMessageInfo(i, chatSessionInfo.chatSessionId(), avatar);
        chatSessionInfo.setPrevMessage(messageInfo);

        pbRsp.chatSessionInfoList().push_back(chatSessionInfo);
    }

    // 序列化
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

获取好友申请列表

1) 客户端发送请求

a) 编写 `MainWidget::initData`

```
loadApplyList();
```

b) 实现 `loadApplyList`

```
void MainWidget::loadApplyList()
{
#if LOAD_DATA_FROM_NETWORK
    // 1. 获取到实例
    DataCenter* dataCenter = DataCenter::getInstance();

    if (dataCenter->getApplyList() != nullptr) {
        // 2. 尝试从内存读数据
        updateApplyList();
    } else {
        // 3. 尝试从网络加载数据
        connect(dataCenter, &DataCenter::getApplyListDone, this,
&MainWidget::updateApplyList, Qt::UniqueConnection);
        dataCenter->getApplyListAsync();
    }
#endif
}
```

c) 实现 `getApplyList` 和 `getApplyListAsync`

```
QList<UserInfo> *DataCenter::getApplyList()
{
    return applyList;
}

void DataCenter::getApplyListAsync()
{
    netClient.getApplyList(loginSessionId);
}
```

d) 实现 `NetClient::getApplyList`

接口定义

```
//获取待处理的，申请自己好友的信息列表
message GetPendingFriendEventListReq {
    string request_id = 1;
    optional string session_id = 2;
    optional string user_id = 3;
}

message FriendEvent {
    string event_id = 1;
    UserInfo sender = 3;
}
message GetPendingFriendEventListRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    repeated FriendEvent event = 4;
}
```

函数实现

```
void NetClient::getApplyList(const QString &loginSessionId)
{
    // 1. 通过 proto 构造 body
    bite_im::GetPendingFriendEventListReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    QByteArray body = req.serialize(&serializer);
    LOG() << "[获取好友申请列表] requestId=" << req.requestId() << ", sessionId="
 << loginSessionId;

    // 2. 发送 http 请求
    QNetworkReply* httpResp = this-
>sendHttpRequest("/service/friend/get_pending_friend_events", body);

    // 3. 处理 http 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto pbRsp = this-
>handleHttpResponse<bite_im::GetPendingFriendEventListRsp>(httpResp);
```

```
        if (!pbRsp) {
            // shared_ptr 可以直接通过 bool 运算判定是否是空对象.
            return;
        }

        // b) 设置 DataCenter 中的数据
        dataCenter->resetApplyList(pbRsp);

        // c) 发送信号
        emit dataCenter->getApplyListDone();
    });
}
```

2) 客户端处理响应

a) 实现 `DataCenter::resetApplyList`

```
void DataCenter::resetApplyList(std::shared_ptr<bite_im::GetPendingFriendEventListRsp> resp)
{
    // 1. 清空列表
    if (applyList == nullptr) {
        applyList = new QList<UserInfo>();
    }
    applyList->clear();

    // 2. 遍历响应结果, 添加到列表中
    auto& eventList = resp->event();
    for (auto& event : eventList) {
        UserInfo userInfo;
        userInfo.load(event.sender());
        applyList->push_back(userInfo);
    }
}
```

b) 定义 `DataCenter` 信号

```
void getApplyListDone();
```

c) 实现 `MainWidget::updateApplyList`

```
void MainWidget::updateApplyList()
{
    if (getActiveTab() != APPLY_LIST) {
        // 如果当前选中的标签页不是好友申请列表，就不更新界面.
        return;
    }

    DataCenter* dataCenter = DataCenter::getInstance();
    const auto* applyList = dataCenter->getApplyList();

    // 1) 清空之前界面的好友列表
    sessionFriendArea->clear();

    // 2) 添加新的好友到界面上
    for (const auto& f : *applyList) {
        sessionFriendArea->addItem(ApplyItemType, f.userId, f.avatar,
f.nickname, "");
    }
}
```

3）服务器逻辑实现

a) 注册路由

```
httpServer.route("/service/friend/get_pending_friend_events", [=](const
QHttpServerRequest& req) {
    return this->getApplyList(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::getApplyList(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::GetPendingFriendEventListReq pbReq;
    pbReq.deserialize(&serializer, req.body());
```

```
    LOG() << "[REQ 获取好友申请列表] request_id=" << pbReq.requestId() << ", loginSessionId=" << pbReq.sessionId();
    // 构造响应
    bite_im::GetPendingFriendEventListRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");

    QByteArray avatar = loadImageToByteArray(":/image/defaultAvatar.png");

    for (int i = 0; i < 5; i++) {
        bite_im::FriendEvent event;
        event.setEventId("event" + QString::number(i));
        event.setSender(makeUserInfo(i, avatar));

        pbRsp.event().push_back(event);
    }

    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

标签页切换

把加载会话列表和好友列表的逻辑, 分别放到点击按钮的槽函数中调用.

> 这个功能点不涉及到前后端交互

a) 编写 `MainWidget::initWindowLeft`

```
// 处理 tab 按钮的点击操作
connect(sessionTabBtn, &QPushButton::clicked, this, &MainWidget::switchTabToSession);
connect(friendTabBtn, &QPushButton::clicked, this, &MainWidget::switchTabToFriend);
connect(applyTabBtn, &QPushButton::clicked, this, &MainWidget::switchTabToApply);
```

b) 实现上述三个槽函数

```
void MainWidget::switchTabToSession()
{
    // 1. 设置 activeTab
    activeTab = SESSION_LIST;
    // 2. 调整图标高亮
    sessionTabBtn->setIcon(QIcon(":/image/session_active.png"));
    friendTabBtn->setIcon(QIcon(":/image/friend_inactive.png"));
    applyTabBtn->setIcon(QIcon(":/image/apply_inactive.png"));
    // 3. 加载会话列表数据
    this->loadSessionList();
}

void MainWidget::switchTabToFriend()
{
    // 1. 设置 activeTab
    activeTab = FRIEND_LIST;
    // 2. 调整图标高亮
    sessionTabBtn->setIcon(QIcon(":/image/session_inactive.png"));
    friendTabBtn->setIcon(QIcon(":/image/friend_active.png"));
    applyTabBtn->setIcon(QIcon(":/image/apply_inactive.png"));
    // 3. 加载好友列表数据
    this->loadFriendList();
}

void MainWidget::switchTabToApply()
{
    // 1. 设置 activeTab
    activeTab = APPLY_LIST;
    // 2. 调整图标高亮
    sessionTabBtn->setIcon(QIcon(":/image/session_inactive.png"));
    friendTabBtn->setIcon(QIcon(":/image/friend_inactive.png"));
    applyTabBtn->setIcon(QIcon(":/image/apply_active.png"));
    // 3. 加载好友列表数据
    this->loadApplyList();
}
```

获取指定会话的近期消息

点击会话列表中的列表项, 获取该会话的最后 N 个历史消息, 并展示到界面上.

1) 客户端发送请求

a) 编写 `SessionItem::active`

> 此处的 active 在 `select` 中已经通过多态的方式调用到了. 只要用户点击, 就能触发这个逻辑.

```
void SessionItem::active()
{
    LOG() << "SessionItem active. chatSessionId=" << chatSessionId;

    // TODO 后续在这里添加针对未读消息的处理.

    // 加载当前会话的消息内容
    MainWidget* mainWidget = MainWidget::getInstance();
    mainWidget->loadRecentMessages(chatSessionId);
}
```

b) 编写 `MainWidget::loadRecentMessages`

```
void MainWidget::loadRecentMessages(const QString &chatSessionId)
{
    // 1. 获取到 DataCenter
    DataCenter* dataCenter = DataCenter::getInstance();

    // 确认要更新界面
    if (dataCenter->getRecentMsgList(chatSessionId) != nullptr) {
        // 3. 通过内存更新界面
        updateRecentMessages(chatSessionId);
    } else {
        // 4. 通过网络获取数据
        //    使用 Qt::UniqueConnection 避免重复关联
        connect(dataCenter, &DataCenter::getRecentMsgListDone,
                this, &MainWidget::updateRecentMessages, Qt::UniqueConnection);
        dataCenter->getRecentMsgListAsync(chatSessionId, true);
    }

}
```

c) 编写 `DataCenter`

```cpp
QList<Message> *DataCenter::getRecentMsgList(const QString &chatSessionId)
{
    if (!recentMessages->contains(chatSessionId)) {
        return nullptr;
    }
    return &(*recentMessages)[chatSessionId];
}

void DataCenter::getRecentMsgListAsync(const QString &chatSessionId, bool updateUI)
{
    netClient.getRecentMsgList(loginSessionId, chatSessionId, updateUI);
}
```

d) 编写 `NetClient`

接口定义

```
message GetRecentMsgReq {
    string request_id = 1;
    string chat_session_id = 2;
    int64 msg_count = 3;
    optional int64 cur_time = 4;//用于扩展获取指定时间前的n条消息
    optional string user_id = 5;
    optional string session_id = 6;
}
message GetRecentMsgRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    repeated MessageInfo msg_list = 4;
}
```

代码实现

```cpp
void NetClient::getRecentMsgList(const QString &loginSessionId, const QString &chatSessionId, bool updateUI)
{
    // 1. 通过 proto 构造 body
```

```
    bite_im::GetRecentMsgReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setChatSessionId(chatSessionId);
    req.setMsgCount(50);                    // 获取最近 50 条消息. 不足 50 则按照实际条数展示.
    QByteArray body = req.serialize(&serializer);
    LOG() << "[获取最近消息] requestId=" << req.requestId() << ", sessionId=" << loginSessionId << ", chatSessionId=" << chatSessionId;

    // 2. 发送 http 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/message_storage/get_recent", body);

    // 3. 处理 http 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto recentMsgRsp = this->handleHttpResponse<bite_im::GetRecentMsgRsp>(httpResp);
        if (!recentMsgRsp) {
            // shared_ptr 可以直接通过 bool 运算判定是否是空对象.
            return;
        }

        // b) 设置 DataCenter 中的数据
        dataCenter->resetRecentMsgList(chatSessionId, recentMsgRsp);

        // c) 发送信号
        if (updateUI) {
            emit dataCenter->getRecentMsgListDone(chatSessionId);
        } else {
            emit dataCenter->getRecentMsgListDoneNoUI(chatSessionId);
        }
    });
}
```

2) 客户端处理响应

a) 实现 `DataCenter::resetRecentMsgList`

```
void DataCenter::resetRecentMsgList(const QString& chatSessionId, std::shared_ptr<bite_im::GetRecentMsgRsp> recentMsgRsp)
{
    // 1. 清空原来的会话列表
```

```
    QList<Message>& messageList = (*recentMessages)[chatSessionId];
    messageList.clear();

    // 2. 遍历响应结果，添加到数据结构中
    auto& messageListPB = recentMsgRsp->msgList();
    for (auto& m : messageListPB) {
        Message message;
        message.load(m);
        messageList.push_back(message);
    }
}
```

b) 定义 `DataCenter` 信号

```
// 获取近期消息完成
void getRecentMsgListDone(const QString& chatSessionId);                // 更新 UI
void getRecentMsgListDoneNoUI(const QString& chatSessionId);        // 不更新 UI
```

c) 实现 `MainWidget::updateRecentMessages`

```
void MainWidget::updateRecentMessages(const QString& chatSessionId)
{
    DataCenter* dataCenter = DataCenter::getInstance();

    // 1) 获取到数据
    auto* recentMsgList = dataCenter->getRecentMsgList(chatSessionId);

    // 2) 设置会话消息
    messageShowArea->clear();
    // for (const auto& message : *recentMsgList) {
    //     bool isLeft = message.sender.userId != dataCenter->getMyself()->userId;
    //     messageShowArea->addMessage(isLeft, message);
    // }
    // 改成从后往前插入
    for (int i = recentMsgList->size() - 1; i >= 0; --i) {
        const Message& message = recentMsgList->at(i);
        bool isLeft = message.sender.userId != dataCenter->getMyself()->userId;
        messageShowArea->addFrontMessage(isLeft, message);
```

```
    }

    // 3) 滚动条滚动到末尾
    messageShowArea->scrollToEndLater();

    // 4) 设置会话标题
    auto* chatSessionInfo = dataCenter->findChatSessionBySessionId(chatSessionId);
    if (chatSessionInfo != nullptr) {
        sessionTitle->setText(chatSessionInfo->chatSessionName);
    } else {
        qCritical() << TAG << "chatSessionInfo 查找失败! chatSessionId=" << chatSessionId;
    }

    // 5) 保存当前被选中的会话
    dataCenter->setCurrentChatSessionId(chatSessionId);
}
```

d) 实现 `DataCenter::findChatSessionBySessionId`

```
ChatSessionInfo *DataCenter::findChatSessionBySessionId(const QString &chatSessionId)
{
    if (chatSessionList == nullptr) {
        return nullptr;
    }
    for (auto& info : *chatSessionList) {
        if (info.chatSessionId == chatSessionId) {
            return &info;
        }
    }
    return nullptr;
}
```

e) 实现 `DataCenter::setCurrentChatSessionId` 和 `DataCenter::getCurrentChatSessionId`

```
const QString &DataCenter::getCurrentChatSessionId() const
{
```

```
    return currentChatSessionId;
}

void DataCenter::setCurrentChatSessionId(const QString &chatSessionId)
{
    this->currentChatSessionId = chatSessionId;
}
```

3) 服务器处理请求

a) 注册路由

```
httpServer.route("/service/message_storage/get_recent", [=](const QHttpServerRequest& req) {
    return this->getRecent(req);
});
```

b) 实现处理函数

```
// 获取近期的历史消息
QHttpServerResponse HttpServer::getRecent(const QHttpServerRequest &req)
{
    int64_t beg = QDateTime::currentMSecsSinceEpoch();
    // 解析请求
    bite_im::GetRecentMsgReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 获取近期消息] request_id=" << pbReq.requestId() << ", sessionId=" << pbReq.sessionId();

    // 构造响应
    bite_im::GetRecentMsgRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");

    // 这个操作会耗时较多，大概 60ms 左右，不应该在循环中进行
    QByteArray avatar = loadImageToByteArray(":/image/defaultAvatar.png");

    for (int i = 0; i < 30; i++) {
```

```
        bite_im::MessageInfo messageInfo = makeMessageInfo(i, pbReq.chatSessionId(), avatar);
        pbRsp.msgList().push_back(messageInfo);
    }

    // 序列化
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");

    int64_t end = QDateTime::currentMSecsSinceEpoch();
    LOG() << "[RESP] request_id=" << pbReq.requestId() << ", sessionId=" << pbReq.sessionId() << ", time=" << (end - beg);
    return httpResp;
}
```

点击某个好友项

1) 切换到会话列表

编写 `FriendItem::active`

> `active` 已经在 `select` 方法中通过多态的方式调用到了.

```
void FriendItem::active()
{
    LOG() << "FriendItem active. userId=" << userId;
    // 切换到当前会话. 如果没有就创建会话
    MainWidget* mainWidget = MainWidget::getInstance();
    mainWidget->switchToSession(userId);
}
```

2) 该会话置顶并被选中

a) 实现 `MainWidget::switchToSession`

```
void MainWidget::switchToSession(const QString &userId)
{
```

```
    DataCenter* dataCenter = DataCenter::getInstance();
    // 1. 先根据好友 id 找到对应的会话
    auto* chatSessionInfo = dataCenter->findChatSessionByUserId(userId);
    if (chatSessionInfo == nullptr) {
        qCritical() << TAG << "用户对应的会话不存在! userId=" << userId;
        return;
    }
    // 2. 把会话置顶
    dataCenter->topChatSessionInfo(*chatSessionInfo);

    // 3. 切换到会话标签页
    //    内部会根据 DataCenter 中的 chatSessionList 的顺序来排列元素.
    switchTabToSession();

    // 4. 点击一下这个会话
    sessionFriendArea->clickItem(0);
}
```

> `switchTabToSession` 已经在前面实现过了.

b) 实现 `DataCenter::findChatSessionByUserId`

```
ChatSessionInfo *DataCenter::findChatSessionByUserId(const QString &userId)
{
    if (chatSessionList == nullptr) {
        return nullptr;
    }
    for (auto& info : *chatSessionList) {
        if (info.userId == userId) {
            return &info;
        }
    }
    return nullptr;
}
```

c) 实现 `DataCenter::topChatSessionInfo`

```
// 把指定的会话信息排列到第一个位置
void DataCenter::topChatSessionInfo(const ChatSessionInfo &chatSessionInfo)
{
    if (chatSessionList == nullptr) {
```

```
        return;
    }
    auto it = chatSessionList->cbegin();
    for (; it != chatSessionList->cend(); ++it) {
        if (it->chatSessionId == chatSessionInfo.chatSessionId) {
            break;
        }
    }
    if (it == chatSessionList->cend()) {
        return;
    }
    // 备份一个 ChatSessionInfo
    ChatSessionInfo backup = chatSessionInfo;
    // 删除该元素
    chatSessionList->erase(it);
    // 插入到前头
    chatSessionList->push_front(backup);
}
```

d) 实现 `SessionFriendArea::clickItem`

```
void SessionFriendArea::clickItem(int index)
{
    if (index >= container->layout()->count()) {
        qCritical() << TAG << "点击的元素下标超出范围！ index=" << index;
        return;
    }
    QLayoutItem* top = container->layout()->itemAt(index);
    if (top->widget() == nullptr) {
        qCritical() << TAG << "指定元素不存在！ index=" << index;
        return;
    }
    SessionFriendItem* item = dynamic_cast<SessionFriendItem*>(top->widget());
    item->select();
}
```

3) 加载该会话的最近消息并显示

在上述 `clickItem` 中会调用 `item->select()`, 进一步调用到 `active` 方法, 从而触发加载最近消息的逻辑.

注意:

- 每个会话中的用户列表, 应该是按需加载的, 不应该是程序启动全都加载进来!
- 创建会话操作放到同意好友申请时. 换而言之, 每个用户都一定存在一个和他对应的会话.

聊天界面逻辑

发送消息

1) 客户端发送消息请求

a) 在 `MessageEditArea` 中创建 `initSignalSlot` 方法.

关联上 `sendBtn` 的槽函数

```
void MessageEditArea::initSignalSlot()
{
    DataCenter* dataCenter = DataCenter::getInstance();

    // 处理按钮点击
    connect(sendBtn, &QPushButton::clicked, this,
&MessageEditArea::sendTextMessage);
}
```

b) 实现 `MessageEditArea::sendTextMessage`

```
void MessageEditArea::sendTextMessage()
{
    DataCenter* dataCenter = DataCenter::getInstance();
    // 1. 如果当前未选中任何会话, 则不发送任何内容
    if (dataCenter->getCurrentChatSessionId().isEmpty()) {
        LOG() << "当前未选中会话, 不发送任何消息!";
        Toast::showMessage("当前未选中会话, 不发送任何消息!");
        return;
    }

    // 2. 获取到输入框中的字符串. 并对这个字符串进行 trim 操作
```

```
    QString content = textEdit->toPlainText().trimmed();
    if (content.isEmpty()) {
        LOG() << "输入框为空，不发送任何消息!";
        return;
    }
    // 3. 清空输入框
    textEdit->setPlainText("");

    // 4. 通过网络发送数据给服务器
    dataCenter->sendTextMessageAsync(dataCenter->getCurrentChatSessionId(),
content);
}
```

c) 实现 `DataCenter::sendTextMessageAsync`

```
void DataCenter::sendTextMessageAsync(const QString& chatSessionId, const
QString &content)
{
    netClient.sendMessage(loginSessionId, chatSessionId,
MessageType::TEXT_TYPE, content.toUtf8());
}
```

d) 实现 `NetClient::sendMessage`

接口定义

```
message NewMessageReq {
    string request_id = 1;
    optional string user_id = 2;
    optional string session_id = 3;
    string chat_session_id = 4;
    MessageContent message = 5;
}
message NewMessageRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
}
```

方法实现

> 此方法同时支持四种消息的发送.

```
void NetClient::sendMessage(const QString &loginSessionId, const QString& chatSessionId,
                            model::MessageType messageType, const QByteArray& content, const QString& extraInfo)
{
    // 1. 通过 proto 构造 body
    bite_im::NewMessageReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setChatSessionId(chatSessionId);

    bite_im::MessageContent messageContent;
    if (messageType == model::MessageType::TEXT_TYPE) {
        messageContent.setMessageType(bite_im::MessageTypeGadget::MessageType::STRING);
        bite_im::StringMessageInfo stringMessageInfo;
        stringMessageInfo.setContent(QString(content));
        messageContent.setStringMessage(stringMessageInfo);
    } else if (messageType == model::MessageType::IMAGE_TYPE) {
        messageContent.setMessageType(bite_im::MessageTypeGadget::MessageType::IMAGE);
        bite_im::ImageMessageInfo imageMessageInfo;
        imageMessageInfo.setFileId("");
        imageMessageInfo.setImageContent(content);
        messageContent.setImageMessage(imageMessageInfo);
    } else if (messageType == model::MessageType::FILE_TYPE) {
        messageContent.setMessageType(bite_im::MessageTypeGadget::MessageType::FILE);
        bite_im::FileMessageInfo fileMessageInfo;
        fileMessageInfo.setFileId("");
        fileMessageInfo.setFileName(extraInfo);
        fileMessageInfo.setFileSize(content.size());
        fileMessageInfo.setFileContents(content);
        messageContent.setFileMessage(fileMessageInfo);
    } else if (messageType == model::MessageType::SPEECH_TYPE) {
        messageContent.setMessageType(bite_im::MessageTypeGadget::MessageType::SPEECH);
        bite_im::SpeechMessageInfo speechMessageInfo;
        speechMessageInfo.setFileId("");
        speechMessageInfo.setFileContents(content);
        messageContent.setSpeechMessage(speechMessageInfo);
```

```
    } else {
        qCritical() << TAG << "未知的 MessageType! messageType=" << messageType;
        return;
    }
    req.setMessage(messageContent);

    QByteArray body = req.serialize(&serializer);
    LOG() << "[发送消息] requestId=" << req.requestId() << ", sessionId=" << loginSessionId << ", chatSessionId=" << chatSessionId;

    // 2. 发送 http 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/message_transmit/new_message", body);

    // 3. 处理 http 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto resp = this->handleHttpResponseWithReason<bite_im::NewMessageRsp>(httpResp);
        if (!resp) {
            LOG() << "发送消息失败";
            return;
        }
        if (!resp->success()) {
            LOG() << "发送消息失败! " << resp->errmsg();
            emit dataCenter->sendMessageFailed(resp->errmsg());
            return;
        }
        // b) 此处不需要设置数据，直接发信号即可
        emit dataCenter->sendMessageDone(messageType, content, extraInfo);
    });
}
```

2) 客户端收到消息响应

a) 定义 `DataCenter` 信号

```
// 发送消息完成
void sendMessageDone(MessageType messageType, const QByteArray& content, const QString& extraInfo);
void sendMessageFailed(const QString& reason);
```

b) 修改 `MessageEditArea::initSignalSlot`，新增信号槽连接

```
// 处理发送消息的网络相应，把自己发的内容添加到消息展示区
connect(dataCenter, &DataCenter::sendMessageDone, this,
&MessageEditArea::addSelfMessage);
connect(dataCenter, &DataCenter::sendMessageFailed, this, [=](const QString&
reason) {
    Toast::showMessage("发送消息失败！ " + reason);
});
```

c) 新增函数 `MessageEditArea::addSelfMessage`

把消息添加到消息显示区

```
void MessageEditArea::addSelfMessage(model::MessageType messageType, const
QByteArray &content, const QString& extraInfo)
{
    DataCenter* dataCenter = DataCenter::getInstance();

    // 1. 构造消息对象，并添加到消息列表中
    const Message& message = dataCenter->addMessage(
        Message::makeMessage(messageType, dataCenter-
>getCurrentChatSessionId(), *dataCenter->getMyself(), content, extraInfo)
    );

    // 2. 向消息展示区添加消息
    MainWidget* mainWidget = dynamic_cast<MainWidget*>(owner);
    MessageShowArea* messageShowArea = mainWidget->getMessageShowArea();
    messageShowArea->addMessage(false, message);

    // 3. 设置消息显示区的滚动条滚动到末尾
    messageShowArea->scrollToEndLater();

    // 4. 更新会话列表的最后一条消息
    emit dataCenter->updateLastMessage(dataCenter->getCurrentChatSessionId());
}
```

d) 给 `DataCenter` 定义信号

更新会话列表中的 "最后一条消息"

```
// 更新会话列表中的最后一条消息
void updateLastMessage(const QString& chatSessionId);
```

e) 在 `SessionArea` 的 `SessionItem` 构造函数中, 连接上述信号并处理

```
SessionItem::SessionItem(QWidget* owner, const QString &chatSessionId, const QIcon &avatar, const QString &name, const QString &lastMessage)
    : SessionFriendItem(owner, avatar, name, lastMessage), chatSessionId(chatSessionId)
{
    DataCenter* dataCenter = DataCenter::getInstance();
    // 连接信号槽
    connect(dataCenter, &DataCenter::updateLastMessage, this, &SessionItem::updateLastMessage);
}
```

```
void SessionItem::updateLastMessage(const QString &chatSessionId)
{
    // 1. 如果发来的信号的会话不是这个 item 当前的会话, 就跳过
    if (this->chatSessionId != chatSessionId) {
        return;
    }

    // 2. 获取到最后一条消息
    DataCenter* dataCenter = DataCenter::getInstance();
    QList<Message>* msgList = dataCenter->getRecentMsgList(chatSessionId);
    if (msgList == nullptr || msgList->size() == 0) {
        return;
    }
    const Message& lastMessage = msgList->back();

    // 3. 确定显示的文本内容
    QString text;
    if (lastMessage.messageType == model::TEXT_TYPE) {
        text = lastMessage.content;
    } else if (lastMessage.messageType == model::IMAGE_TYPE) {
        text = "[图片]";
    } else if (lastMessage.messageType == model::FILE_TYPE) {
        text = "[文件]";
    } else if (lastMessage.messageType == model::SPEECH_TYPE) {
        text = "[语音]";
    } else {
```

```
        text = "[未知类型消息]";
    }

    // 4. 显示文本
    if (chatSessionId != dataCenter->getCurrentChatSessionId()) {
        // 当前会话不是正在选中的会话，写入未读消息数目
        int unreadCount = dataCenter->getUnread(chatSessionId);
        this->messageLabel->setText(QString("[未读%1条] ").arg(unreadCount) +
text);
    } else {
        // 是正在选中的会话，不需要写入未读消息数目
        this->messageLabel->setText(text);
    }
}
```

f) 实现对于未读消息数据的处理

```
void DataCenter::clearUnread(const QString &chatSessionId) {
    (*unreadMessageCount)[chatSessionId] = 0;

    // 保存到文件中
    saveDataFile();
}

void DataCenter::addUnread(const QString &chatSessionId) {
    ++(*unreadMessageCount)[chatSessionId];

    // 保存到文件中
    saveDataFile();
}

int DataCenter::getUnread(const QString &chatSessionId)
{
    return (*unreadMessageCount)[chatSessionId];
}
```

g) 补充 `SessionItem` 中的未读消息处理

```
SessionItem::SessionItem(QWidget* owner, const QString &chatSessionId, const
QIcon &avatar, const QString &name, const QString &lastMessage)
```

```
    : SessionFriendItem(owner, avatar, name, lastMessage),
chatSessionId(chatSessionId)
{
    DataCenter* dataCenter = DataCenter::getInstance();
    // 连接信号槽
    connect(dataCenter, &DataCenter::updateLastMessage, this,
&SessionItem::updateLastMessage);

    // 构造的时候先获取一下未读消息
    int unread = dataCenter->getUnread(chatSessionId);
    if (unread > 0) {
        this->messageLabel->setText(QString("[未读%1条] ").arg(unread) +
lastMessage);
    }
}
```

```
void SessionItem::active()
{
    LOG() << "SessionItem active. chatSessionId=" << chatSessionId;

    // 清空未读消息数目
    // 如果是消息列表可能还没加载，直接清空原来文本内容中的 "未读x条";
    QString text = messageLabel->text();
    static QRegularExpression reg(R"(\[\S+\])");
    messageLabel->setText(text.replace(reg, ""));

    DataCenter* dataCenter = DataCenter::getInstance();
    dataCenter->clearUnread(chatSessionId);

    // 加载当前会话的消息内容
    MainWidget* mainWidget = MainWidget::getInstance();
    mainWidget->loadRecentMessages(chatSessionId);
}
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/message_transmit/new_message", [=](const
QHttpServerRequest& req) {
    return this->sendMessage(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::sendMessage(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::NewMessageReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 发送消息] request_id=" << pbReq.requestId() << ", sessionId="
 << pbReq.sessionId();

    // 构造响应
    bite_im::NewMessageRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

接收消息

1) 客户端实现逻辑

a) 在 `NetClient` 中实现 `handleWsMessage` 处理 websocket 收到的数据.

```
void NetClient::handleWsMessage(const model::Message &message)
{
    QList<model::Message>* messageList = dataCenter-
>getRecentMsgList(message.chatSessionId);
    if (messageList == nullptr) {
        // 还没从网络上获取到消息列表，就先从网络加载
        connect(dataCenter, &DataCenter::getRecentMsgListDoneNoUI, dataCenter,
&DataCenter::receiveMessage,
                Qt::UniqueConnection);
        dataCenter->getRecentMsgListAsync(message.chatSessionId, false);
```

```
    } else {
        // 本地已经有消息列表了，直接添加进去
        messageList->push_back(message);
        dataCenter->receiveMessage(message.chatSessionId);
    }
}
```

b) 实现 `DataCenter::receiveMessage`

```
void DataCenter::receiveMessage(const QString& chatSessionId)
{
    // 1. 如果消息是来自于当前被选中会话，那么直接把消息展示到消息展示区.
    if (chatSessionId == currentChatSessionId) {
        // 消息添加到消息展示区
        // a) 先获取到最后一条消息
        const Message& lastMessage = (*recentMessages)[chatSessionId].back();
        // b) 通过信号告知界面要添加一个新消息
        emit this->receiveMessageDone(lastMessage);
    } else {
        // 2. 在会话列表中给出未读消息提示
        //    需要在 DataCenter 中创建一个 hash，key 为 chatSessionId，value 为未读消息个数
        //    然后通知给界面，让界面更新未读消息数字.
        //    当对应的会话被点击，则对应的未读消息的数字
        //    未读消息数目需要在硬盘保存，以备在重启后重新获得
        //    对于当前会话的窗口，不需要保存这个数据
        addUnread(chatSessionId);
    }

    // 3. 再发个信号，更新会话列表中的显示的 lastMessage
    emit this->updateLastMessage(chatSessionId);
}
```

c) 定义 `DataCenter` 信号

```
// 收到消息
void receiveMessageDone(const Message& message);
```

d) 修改 `MessageEditArea::initSignalSlot`, 添加信号槽

```
// 处理收到网络上来自别人的响应情况
connect(dataCenter, &DataCenter::receiveMessageDone, this,
&MessageEditArea::addOtherMessage);
```

e) 实现 `MessageEditArea::addOtherMessage`

```
void MessageEditArea::addOtherMessage(const model::Message &message)
{
    // 把这个消息显示到界面上
    MainWidget* mainWidget = dynamic_cast<MainWidget*>(owner);
    MessageShowArea* messageShowArea = mainWidget->getMessageShowArea();

    messageShowArea->addMessage(true, message);

    // 设置消息显示区的滚动条滚动到末尾
    messageShowArea->scrollToEndLater();

    // 提示收到一个消息
    Toast::showMessage("收到一条新消息!");
}
```

2) 服务器实现逻辑

a) 在界面上创建一个按钮, 表示 "发送文本消息", 并实现信号槽

```
void Widget::on_pushButton_clicked()
{
    WebsocketServer* websocketServer = WebsocketServer::getInstance();
    emit websocketServer->sendTextResp();
}
```

b) 给 `WebsocketServer` 创建信号 `sendTextResp`

```
signals:
```

```
    void sendTextResp();
```

c) 实现处理函数.

注意此处的 connect 要放到 `connect(&websocketServer, &QWebSocketServer::newConnection, this, [=] () { } )` 中

这样才能捕获到 socket 对象.

```
connect(this, &WebsocketServer::sendTextResp, this, [=]() {
    if (socket == nullptr || !socket->isValid()) {
        LOG() << "socket 对象无效!";
        return;
    }
    bite_im::NotifyMessage notifyMessage;
    notifyMessage.setNotifyEventId("");

notifyMessage.setNotifyType(bite_im::NotifyTypeGadget::NotifyType::CHAT_MESSAGE_NOTIFY);

    bite_im::NotifyNewMessage newMessage;
    bite_im::MessageInfo messageInfo = makeMessageInfo(this->messageIndex++, "2000", avatar);
    newMessage.setMessageInfo(messageInfo);

    notifyMessage.setNewMessageInfo(newMessage);
    QByteArray body = notifyMessage.serialize(&serializer);

    socket->sendBinaryMessage(body);

    LOG() << "发送文本消息响应" << messageInfo.message().stringMessage().content();
});
```

d) 在 `QWebSocket::disconnected` 处理函数中, 添加解除信号槽的逻辑

```
disconnect(this, &WebsocketServer::sendTextResp, this, nullptr);
```

> 此处的 disconnect 非常重要. 否则如果客户端重复连接服务器, 服务器就会尝试针对上次已经释放的 socket 对象进行处理, 就会使程序崩溃.

个人信息详情逻辑

加载个人信息

直接从 DataCenter 中读取数据

在 `SelfInfoWidget` 构造函数中, 添加数据加载

```
// 通过网络获取到数据显示界面
if (myself != nullptr) {
    avatar->setIcon(myself->avatar);
    idLabel->setText(myself->userId);
    nameLabel->setText(myself->nickname);
    descLabel->setText(myself->description);
    phoneLabel->setText(myself->phone);
}
```

修改昵称

1) 客户端发送请求

a) 在 `SelfInfoWidget` 构造函数连接信号槽.

```
connect(nameModifyBtn, &QPushButton::clicked, this,
&SelfInfoWidget::clickNameModifyBtn);
connect(nameSubmitBtn, &QPushButton::clicked, this,
&SelfInfoWidget::clickNameSubmitBtn);
```

b) 实现 `SelfInfoWidget::clickNameModifyBtn`

切换显示状态.

```
void SelfInfoWidget::clickNameModifyBtn()
```

```
{
    // 隐藏 label, 显示 lineEdit
    gridLayout->removeWidget(nameLabel);
    nameLabel->hide();
    gridLayout->addWidget(nameEdit, 1, 1);
    nameEdit->show();
    nameEdit->setText(nameLabel->text());
    nameEdit->setFocus();

    // 切换按钮的显示
    gridLayout->removeWidget(nameModifyBtn);
    nameModifyBtn->hide();
    gridLayout->addWidget(nameSubmitBtn, 1, 2);
    nameSubmitBtn->show();
}
```

c) 实现 `SelfInfoWidget::clickNameSubmitBtn`

```
void SelfInfoWidget::clickNameSubmitBtn()
{
    // 1. 获取到新的名字
    const QString& newName = nameEdit->text();
    if (newName.isEmpty()) {
        return;
    }
    DataCenter* dataCenter = DataCenter::getInstance();
    // 2. 响应结果到了之后, 把界面进行改变, 恢复成 label 来显示
    connect(dataCenter, &DataCenter::changeNickNameDone, this,
&SelfInfoWidget::clickNameSubmitBtnDone, Qt::UniqueConnection);
    // 3. 发送修改名字请求
    dataCenter->changeNickNameAsync(newName);
}
```

d) 实现 `DataCenter::changeNickNameAsync`

```
// 修改昵称
void DataCenter::changeNickNameAsync(const QString &nickName)
{
    netClient.changeNickName(loginSessionId, nickName);
}
```

e) 实现 `NetClient::changeNickName`

接口定义

```
//----------------------------
//用户昵称修改
message SetUserNicknameReq {
    string request_id = 1;
    optional string user_id = 2;
    optional string session_id = 3;
    string nickname = 4;
}
message SetUserNicknameRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
}
```

函数实现

```
void NetClient::changeNickName(const QString &loginSessionId, const QString &nickName)
{
    // 1. 构造请求
    bite_im::SetUserNicknameReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setNickname(nickName);
    QByteArray body = req.serialize(&serializer);
    LOG() << "[修改用户昵称] requestId=" << req.requestId() << ", sessionId=" << loginSessionId << ", nickName=" << nickName;

    // 2. 发送 http 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/user/set_nickname", body);

    // 3. 处理 http 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
```

```
        auto resp = this->handleHttpResponse<bite_im::SetUserNicknameRsp>
(httpResp);
        if (!resp) {
            // shared_ptr 可以直接通过 bool 运算判定是否是空对象.
            return;
        }
        // b) 修改数据
        dataCenter->resetNickName(nickName);

        // c) 发送信号
        emit dataCenter->changeNickNameDone();
    });
}
```

2) 客户端处理响应

a) 实现 `DataCenter::resetNickName`

```
void DataCenter::resetNickName(const QString& nickName)
{
    myself->nickname = nickName;
}
```

b) 定义 `DataCenter` 信号

```
void changeNickNameDone();
```

c) 实现 `SelfInfoWidget::clickNameSubmitBtnDone`

```
void SelfInfoWidget::clickNameSubmitBtnDone()
{
    // 隐藏 lineEdit, 显示 label
    gridLayout->removeWidget(nameEdit);
    nameEdit->hide();
    gridLayout->addWidget(nameLabel, 1, 1);
    nameLabel->show();
```

```
    nameLabel->setText(nameEdit->text());

    // 切换按钮的显示
    gridLayout->removeWidget(nameSubmitBtn);
    nameSubmitBtn->hide();
    gridLayout->addWidget(nameModifyBtn, 1, 2);
    nameModifyBtn->show();
}
```

d) 修改 `MessageShowArea` 的 `MessageItem::makeMessageItem`，自动更新消息展示区的消息中显示的昵称.

```
// 5.2 用户修改了昵称，则修改昵称内容
if (!isLeft) {
    // 只针对自己的消息做出自动调整
    DataCenter* dataCenter = DataCenter::getInstance();

    // 调整名字
    connect(dataCenter, &DataCenter::changeNickNameDone, messageItem, [=]() {
        nameLabel->setText(dataCenter->getMyself()->nickname + " | " +
message.time);
    });
}
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/user/set_nickname", [=](const QHttpServerRequest&
req) {
    return this->setNickName(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::setNickName(const QHttpServerRequest &req)
```

```
{
    // 解析请求
    bite_im::SetUserNicknameReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 设置昵称] request_id=" << pbReq.requestId() << ", sessionId="
 << pbReq.sessionId();

    // 构造响应
    bite_im::SetUserNicknameRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

修改签名

1) 客户端发送请求

a) 在 `SelfInfoWidget` 构造函数连接信号槽.

```
connect(descModifyBtn, &QPushButton::clicked, this,
&SelfInfoWidget::clickSignatureModifyBtn);
connect(descSubmitBtn, &QPushButton::clicked, this,
&SelfInfoWidget::clickSignatureSubmitBtn);
```

b) 实现 `SelfInfoWidget::clickSignatureModifyBtn`

```
void SelfInfoWidget::clickSignatureModifyBtn()
{
    // 隐藏 label, 显示 lineEdit
    gridLayout->removeWidget(descLabel);
    descLabel->hide();
    gridLayout->addWidget(descEdit, 2, 1);
    descEdit->show();
```

```
    descEdit->setText(descLabel->text());
    descEdit->setFocus();

    // 切换按钮的显示
    gridLayout->removeWidget(descModifyBtn);
    descModifyBtn->hide();
    gridLayout->addWidget(descSubmitBtn, 2, 2);
    descSubmitBtn->show();
}
```

c) 实现 `SelfInfoWidget::clickSignatureSubmitBtn`

```
void SelfInfoWidget::clickSignatureSubmitBtn()
{
    // 1. 获取到新的签名
    const QString& newDesc = descEdit->text();
    if (newDesc.isEmpty()) {
        return;
    }
    DataCenter* dataCenter = DataCenter::getInstance();
    // 2. 相应结果到了之后，把界面进行改变，恢复成 label 来显示
    connect(dataCenter, &DataCenter::changeDescriptionDone, this,
&SelfInfoWidget::clickSignatureSubmitBtnDone, Qt::UniqueConnection);
    // 3. 发送修改名字请求
    dataCenter->changeDescriptionAsync(newDesc);
}
```

d) 实现 `DataCenter::changeDescriptionAsync`

```
void DataCenter::changeDescriptionAsync(const QString &description)
{
    netClient.changeDescription(loginSessionId, description);
}
```

e) 实现 `NetClient::changeDescription`

接口定义

```
//----------------------------
//用户签名修改
message SetUserDescriptionReq {
    string request_id = 1;
    optional string user_id = 2;
    optional string session_id = 3;
    string description = 4;
}
message SetUserDescriptionRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
}
```

函数实现

```
void NetClient::changeDescription(const QString &loginSessionId, const QString &description)
{
    // 1. 构造请求
    bite_im::SetUserDescriptionReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setDescription(description);
    QByteArray body = req.serialize(&serializer);
    LOG() << "[修改用户昵称] requestId=" << req.requestId() << ", sessionId=" << loginSessionId << ", description=" << description;

    // 2. 发送 http 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/user/set_description", body);

    // 3. 处理 http 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto resp = this->handleHttpResponse<bite_im::SetUserDescriptionRsp>(httpResp);
        if (!resp) {
            // shared_ptr 可以直接通过 bool 运算判定是否是空对象.
            return;
        }
        // b) 修改数据
        dataCenter->resetDescription(description);

```

```
        // c) 发送信号
        emit dataCenter->changeDescriptionDone();
    });
}
```

2) 客户端处理响应

a) 实现 `DataCenter::resetDescription`

```
void DataCenter::resetDescription(const QString &description)
{
    myself->description = description;
}
```

b) 定义 `DataCenter` 信号

```
void changeDescriptionDone();
```

c) 实现 `SelfInfoWidget::clickSignatureSubmitBtnDone`

```
void SelfInfoWidget::clickSignatureSubmitBtnDone()
{
    // 隐藏 lineEdit, 显示 label
    gridLayout->removeWidget(descEdit);
    descEdit->hide();
    gridLayout->addWidget(descLabel, 2, 1);
    descLabel->show();
    descLabel->setText(descEdit->text());

    // 切换按钮的显示
    gridLayout->removeWidget(descSubmitBtn);
    descSubmitBtn->hide();
    gridLayout->addWidget(descModifyBtn, 2, 2);
    descModifyBtn->show();
}
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/user/set_description", [=](const
QHttpServerRequest& req) {
    return this->setDescription(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::setDescription(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::SetUserDescriptionReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 设置昵称] request_id=" << pbReq.requestId() << ", sessionId="
 << pbReq.sessionId();

    // 构造响应
    bite_im::SetUserDescriptionRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

修改电话 (1) - 发起短信验证码

1) 客户端发送请求

a) 在 `SelfInfoWidget` 构造函数连接信号槽.

```
connect(phoneModifyBtn, &QPushButton::clicked, this,
&SelfInfoWidget::clickPhoneModifyBtn);
connect(phoneSubmitBtn, &QPushButton::clicked, this,
&SelfInfoWidget::clickPhoneSubmitBtn);
connect(sendVerityCodeBtn, &QPushButton::clicked, this,
&SelfInfoWidget::clickSendVerityCodeBtn);
```

b) 实现 `clickPhoneModifyBtn`

```
void SelfInfoWidget::clickPhoneModifyBtn()
{
    // 隐藏 label, 显示 lineEdit
    gridLayout->removeWidget(phoneLabel);
    phoneLabel->hide();
    gridLayout->addWidget(phoneEdit, 3, 1);
    phoneEdit->show();
    phoneEdit->setText(phoneLabel->text());
    phoneEdit->setFocus();

    // 切换按钮的显示
    gridLayout->removeWidget(phoneModifyBtn);
    phoneModifyBtn->hide();
    gridLayout->addWidget(phoneSubmitBtn, 3, 2);
    phoneSubmitBtn->show();

    // 显示验证码输入框
    verityCodeEdit->show();
    sendVerityCodeBtn->show();
}
```

c) 实现 `clickSendVerityCodeBtn`

发送验证码

> **注意:**
>
> 需要在 SelfInfoWidget 中把发送验证码的手机号存起来, 并在后续发送修改请求的时候使用这一个号码来请求.
>
> 确保重新绑定的手机号码和发送验证码的手机号码一致.
>
> (发送修改手机号请求的时候手机号码不能从输入框读取!)

```cpp
void SelfInfoWidget::clickSendVerityCodeBtn()
{
    // 1. 获取到手机号
    const QString& phone = phoneEdit->text();
    if (phone.isEmpty()) {
        return;
    }
    // 2. 发送请求
    DataCenter* dataCenter = DataCenter::getInstance();
    dataCenter->getVerifyCodeAsync(phone);

    // 3. 记录要绑定的新手机号码. 必须在此处记录
    //    使得后续发送修改请求时候的手机号码, 和验证的手机号码一致.
    this->phoneToChange = phone;

    // 4. 禁用发送按钮
    sendVerityCodeBtn->setEnabled(false);

    // 5. 启动定时器, 在一定时间之后恢复按钮可用.
    leftTime = 30;
    QTimer* timer = new QTimer(this);
    connect(timer, &QTimer::timeout, this, [=]() {
        if (leftTime <= 1) {
            // 计时结束.
            sendVerityCodeBtn->setEnabled(true);
            sendVerityCodeBtn->setText("获取");
            timer->stop();
            timer->deleteLater();
            return;
        }
        leftTime--;
        sendVerityCodeBtn->setText(QString::number(leftTime) + "s");
    });
    timer->start(1000);
}
```

d) 实现 `DataCenter::getVerifyCodeAsync`

```cpp
void DataCenter::getVerifyCodeAsync(const QString& phone)
{
    netClient.getVerifyCode(phone);
}
```

e) 实现 `NetClient::getVerifyCode`

接口定义

```
//----------------------------
//手机号验证码获取
message PhoneVerifyCodeReq {
    string request_id = 1;
    string phone_number = 2;
}
message PhoneVerifyCodeRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    string verify_code_id = 4;
}
```

> 服务器会在 redis 中保存 `verify_code_id` 和 `verify_code`，给后续的验证提供支持.

函数实现

```
void NetClient::getVerifyCode(const QString &phone)
{
    // 1. 构造请求
    bite_im::PhoneVerifyCodeReq req;
    req.setRequestId(makeRequestId());
    req.setPhoneNumber(phone);
    QByteArray body = req.serialize(&serializer);
    LOG() << "[获取验证码] requestId=" << req.requestId() << " phoneNumber=" << phone;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/user/get_phone_verify_code", body);

    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto resp = this->handleHttpResponse<bite_im::PhoneVerifyCodeRsp>(httpResp);
        if (!resp) {
            return;
```

```
        }

        // b) 修改数据
        dataCenter->resetVerifyCodeId(resp);

        // c) 发送信号
        emit dataCenter->getVerifyCodeDone();
    });
}
```

2) 客户端处理响应

a) 实现 `DataCenter::resetVerifyCodeId`

```
void
DataCenter::resetVerifyCodeId(std::shared_ptr<bite_im::PhoneVerifyCodeRsp>
resp)
{
    this->currentVerifyCodeId = resp->verifyCodeId();
}
```

b) 定义 `DataCenter` 信号

```
void getVerifyCodeDone();
```

> 这个信号暂时不使用. 会在后续的 "手机号登录" 功能中使用.

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/user/get_phone_verify_code", [=](const
QHttpServerRequest& req) {
    return this->getVerifyCode(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::getVerifyCode(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::PhoneVerifyCodeReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 获取手机验证码] request_id=" << pbReq.requestId() << ", phone=" << pbReq.phoneNumber();

    // 构造响应
    bite_im::PhoneVerifyCodeRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");
    pbRsp.setVerifyCodeId("testVerifyCodeId");
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

修改电话 (2) - 修改电话逻辑

1) 客户端发送请求

a) 实现 `SelfInfoWidget::clickPhoneSubmitBtn`

> **注意:**
>
> 需要在 SelfInfoWidget 中把发送验证码的手机号存起来, 并在后续发送修改请求的时候使用这一个号码来请求.
>
> 确保重新绑定的手机号码和发送验证码的手机号码一致.
>
> (发送修改手机号请求的时候手机号码不能从输入框读取!)

```
void SelfInfoWidget::clickPhoneSubmitBtn()
{
```

```
    DataCenter* dataCenter = DataCenter::getInstance();

    // 1. 判定下 verifyCodeId 是否获取到
    const QString& verifyCodeId = dataCenter->getVerifyCodeId();
    if (verifyCodeId.isEmpty()) {
        LOG() << "验证码尚未获取成功! 稍后再试!";
        return;
    }

    // 2. 获取到验证码输入框内容
    const QString& verifyCode = verityCodeEdit->text();
    if (verifyCode.isEmpty()) {
        return;
    }

    // 3. 发送请求. 这里的手机号码不能从输入框读取!
    connect(dataCenter, &DataCenter::changePhoneDone, this,
            &SelfInfoWidget::clickPhoneSubmitBtnDone, Qt::UniqueConnection);
    dataCenter->changePhoneAsync(this->phoneToChange, verifyCodeId,
verifyCode);

    // 4. 结束获取按钮的倒计时
    leftTime = 1;
}
```

b) 实现 `DataCenter::changePhoneAsync`

```
void DataCenter::changePhoneAsync(const QString &phone, const QString&
verifyCodeId, const QString& verifyCode)
{
    netClient.changePhone(loginSessionId, phone, verifyCodeId, verifyCode);
}
```

c) 实现 `NetClient::changePhone`

接口定义

```
//---------------------------
//用户手机修改
message SetUserPhoneNumberReq {
    string request_id = 1;
```

```
    optional string user_id = 2;
    optional string session_id = 3;
    string phone_number = 4;
    string phone_verify_code_id = 5;
    string phone_verify_code = 6;
}
message SetUserPhoneNumberRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
}
```

函数实现

```
void NetClient::changePhone(const QString &loginSessionId, const QString &phone, const QString& verifyCodeId,
                            const QString& verifyCode)
{
    // 1. 构造请求
    bite_im::SetUserPhoneNumberReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setPhoneNumber(phone);
    req.setPhoneVerifyCodeId(verifyCodeId);
    req.setPhoneVerifyCode(verifyCode);
    QByteArray body = req.serialize(&serializer);
    LOG() << "[修改手机号] requestId=" << req.requestId() << " phoneNumber=" << phone << " verifyCodeId=" << verifyCodeId
          << " verifyCode=" << verifyCode;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/user/set_phone", body);

    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto resp = this->handleHttpResponse<bite_im::SetUserPhoneNumberRsp>(httpResp);
        if (!resp) {
            return;
        }

        // b) 修改数据
```

```
        dataCenter->resetPhone(phone);

        // c) 发送信号
        emit dataCenter->changePhoneDone();
    });

}
```

2) 客户端处理响应

a) 实现 `DataCenter::resetPhone`

```
void DataCenter::resetPhone(const QString &phone)
{
    myself->phone = phone;
}
```

b) 定义 `DataCenter` 信号

```
// 修改手机号完成
void changePhoneDone();
```

c) 实现 `SelfInfoWidget::clickPhoneSubmitBtnDone`

```
void SelfInfoWidget::clickPhoneSubmitBtnDone()
{
    // 隐藏 lineEdit, 显示 label
    gridLayout->removeWidget(phoneEdit);
    phoneEdit->hide();
    gridLayout->addWidget(phoneLabel, 3, 1);
    phoneLabel->show();
    phoneLabel->setText(phoneEdit->text());
    phoneLabel->setFocus();

    // 切换按钮的显示
    gridLayout->removeWidget(phoneSubmitBtn);
    phoneSubmitBtn->hide();
```

```
    gridLayout->addWidget(phoneModifyBtn, 3, 2);
    phoneModifyBtn->show();

    // 隐藏验证码输入框
    verityCodeEdit->hide();
    sendVerityCodeBtn->hide();
}
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/user/set_phone", [=](const QHttpServerRequest& req)
{
    return this->setPhone(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::setPhone(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::SetUserPhoneNumberReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 修改手机号码] request_id=" << pbReq.requestId() << ", phone="
<< pbReq.phoneNumber()
          << ", verifyCodeId=" << pbReq.phoneVerifyCodeId() << ", verifyCode="
<< pbReq.phoneVerifyCode();

    // 构造响应
    bite_im::SetUserPhoneNumberRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
```

```
20 }
```

修改头像

1) 客户端发送请求

a) 在 `SelfInfoWidget` 构造函数连接信号槽.

```
1 connect(avatar, &QPushButton::clicked, this, &SelfInfoWidget::clickAvatar);
```

b) 实现 `SelfInfoWidget::clickAvatar`

```
1 void SelfInfoWidget::clickAvatar()
2 {
3     // 1. 设置过滤器, 只打开图片
4     QString filter = "Image files (*.png *.jpeg *.jpg *.bmp)";
5
6     // 2. 弹出一个文件选择对话框
7     QString imagePath = QFileDialog::getOpenFileName(this, "选择头像",
  QDir::homePath(), filter);
8     if (imagePath.isEmpty()) {
9         LOG() << "clickAvatar 未选择图片";
10        return;
11    }
12
13    // 3. 读取图片内容
14    QByteArray imageBytes = loadImageToByteArray(imagePath);
15
16    // 4. 发送请求
17    DataCenter* dataCenter = DataCenter::getInstance();
18    connect(dataCenter, &DataCenter::changeAvatarDone, this,
  &SelfInfoWidget::clickAvatarDone, Qt::UniqueConnection);
19    dataCenter->changeAvatarAsync(imageBytes);
20 }
```

c) 实现 `DataCenter::changeAvatarAsync`

```cpp
void DataCenter::changeAvatarAsync(const QByteArray &avatar)
{
    netClient.changeAvatar(loginSessionId, avatar);
}
```

d) 实现 `NetClient::changeAvatar`

接口定义

```proto
//---------------------------
//用户头像修改
message SetUserAvatarReq {
    string request_id = 1;
    optional string user_id = 2;
    optional string session_id = 3;
    bytes avatar = 4;
}
message SetUserAvatarRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
}
```

函数实现

```cpp
void NetClient::changeAvatar(const QString &loginSessionId, const QByteArray &avatar)
{
    // 1. 构造请求
    bite_im::SetUserAvatarReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setAvatar(avatar);
    QByteArray body = req.serialize(&serializer);
    LOG() << "[修改头像] requestId=" << req.requestId() << " loginSessionId=" << loginSessionId;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/user/set_avatar", body);

```

```
    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto resp = this->handleHttpResponse<bite_im::SetUserAvatarRsp>
(httpResp);
        if (!resp) {
            return;
        }

        // b) 修改数据
        dataCenter->resetAvatar(avatar);

        // c) 发送信号
        emit dataCenter->changeAvatarDone();
    });
}
```

2) 客户端处理响应

a) 实现 `DataCenter::resetAvatar`

```
void DataCenter::resetAvatar(const QByteArray &avatar)
{
    myself->avatar = makeIcon(avatar);
}
```

b) 定义 `DataCenter` 信号

```
void changeAvatarDone();
```

c) 实现 `SelfInfoWidget::clickAvatarDone`

修改用户详情界面的头像

```
void SelfInfoWidget::clickAvatarDone()
{
    DataCenter* dataCenter = DataCenter::getInstance();
```

```
    UserInfo* myself = dataCenter->getMyself();
    avatar->setIcon(myself->avatar);
}
```

d) 修改主界面的头像

在 `MainWidget::initData` 中处理 `changeAvatarDone` 信号.

```
// 在用户详情页修改头像时更新
connect(dataCenter, &DataCenter::changeAvatarDone, this, [=]() {
    const auto* myself = dataCenter->getMyself();
    this->userAvatar->setIcon(myself->avatar);
});
```

e) 修改消息显示区的头像

在 `ShowMessageArea` 的 `MessageItem::makeMessageItem` 中处理 `changeAvatarDone` 信号.

> 和修改名字放在一起.

```
if (!isLeft) {
    // 只针对自己的消息做出自动调整
    DataCenter* dataCenter = DataCenter::getInstance();

    // 调整名字
    connect(dataCenter, &DataCenter::changeNickNameDone, messageItem, [=]() {
        nameLabel->setText(dataCenter->getMyself()->nickname + " | " +
message.time);
    });

    // 调整头像
    connect(dataCenter, &DataCenter::changeAvatarDone, messageItem, [=]() {
        avatarBtn->setIcon(dataCenter->getMyself()->avatar);
    });
}
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/user/set_avatar", [=](const QHttpServerRequest&
req) {
    return this->setAvatar(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::setAvatar(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::SetUserAvatarReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 修改头像] request_id=" << pbReq.requestId() << ",
loginSessionId=" << pbReq.sessionId();
    // 构造响应
    bite_im::SetUserAvatarRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

用户详细信息界面逻辑

获取指定用户的信息

客户端处理逻辑

从对应的 Message 对象中获取到用户详细信息.

> 不需要从服务器获取数据.

a) 在 `UserInfoWidget` 的构造函数中, 添加逻辑获取数据.

```
DataCenter* dataCenter = DataCenter::getInstance();
auto* myFriend = dataCenter->getFriendById(message.sender.userId);
QString btnDisableStyle = "QPushButton { border: none; border-radius: 5px;
background-color: rgb(204, 204, 204); color: rgb(60, 60, 60);}";
if (myFriend == nullptr) {
    // 不是好友
    sendMessageBtn->setEnabled(false);
    sendMessageBtn->setStyleSheet(btnDisableStyle);
    deleteFriendBtn->setEnabled(false);
    deleteFriendBtn->setStyleSheet(btnDisableStyle);
} else {
    // 是好友
    addFriendBtn->setEnabled(false);
    addFriendBtn->setStyleSheet(btnDisableStyle);
}
```

b) 实现 `DataCenter::getFriendById`

```
UserInfo *DataCenter::getFriendById(const QString &userId)
{
    if (friendList == nullptr) {
        return nullptr;
    }
    for (auto& f : *friendList) {
        if (f.userId == userId) {
            return &f;
        }
    }
    return nullptr;
}
```

点击 "发送消息" 打开对应会话

客户端处理逻辑

直接调用之前的逻辑即可. 在选中好友时有类似的逻辑, 直接调用即可.

> 不需要和服务器交互

a) 在 `UserInfoWidget` 构造函数中, 连接信号槽

```
connect(sendMessageBtn, &QPushButton::clicked, this,
&UserInfoWidget::clickSendMessageBtn);
```

b) 实现 `UserInfoWidget::clickSendMessageBtn`

```
void UserInfoWidget::clickSendMessageBtn()
{
    // 1. 切换到会话列表, 调用 MainWidget 的 switchToSession 即可.
    MainWidget* mainWidget = MainWidget::getInstance();
    mainWidget->switchToSession(message.sender.userId);

    // 2. 关闭本窗口
    this->close();
}
```

删除好友

1) 客户端发送请求

a) 在 `UserInfoWidget` 构造函数中连接信号槽

```
connect(deleteFriendBtn, &QPushButton::clicked, this,
&UserInfoWidget::clickDeleteFriendBtn);
```

b) 实现 `UserInfoWidget::clickDeleteFriendBtn`

```
void UserInfoWidget::clickDeleteFriendBtn()
{
    // 1. 弹出对话框确认是否要删除
    auto result = QMessageBox::warning(this, "确认删除", "确认删除该好友?",
QMessageBox::Ok | QMessageBox::Cancel);
    if (result != QMessageBox::Ok) {
```

```
        LOG() << "删除好友取消";
        return;
    }
    // 2. 发送请求删除好友
    DataCenter* dataCenter = DataCenter::getInstance();
    dataCenter->deleteFriendAsync(message.sender.userId);
    // 3. 关闭本窗口
    this->close();
}
```

c) 实现 `DataCenter::deleteFriendAsync`

```
void DataCenter::deleteFriendAsync(const QString &userId)
{
    netClient.deleteFriend(loginSessionId, userId);
}
```

d) 实现 `NetClient::deleteFriend`

接口定义

```
//--------------------------------------
//好友删除
message FriendRemoveReq {
    string request_id = 1;
    optional string user_id = 2;
    optional string session_id = 3;
    string peer_id = 4;
}
message FriendRemoveRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
}
```

函数实现

```cpp
void NetClient::deleteFriend(const QString& loginSessionId, const QString &userId)
{
    // 1. 构造请求
    bite_im::FriendRemoveReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setPeerId(userId);
    QByteArray body = req.serialize(&serializer);
    LOG() << "[删除好友] requestId=" << req.requestId() << " loginSessionId=" << loginSessionId << " userId=" << userId;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/friend/remove_friend", body);

    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto resp = this->handleHttpResponse<bite_im::FriendRemoveRsp>(httpResp);
        if (!resp) {
            return;
        }

        // b) 修改数据
        dataCenter->removeFriend(userId);

        // c) 发送信号
        emit dataCenter->deleteFriendDone(userId);
    });
}
```

2) 客户端处理响应

a) 实现 `DataCenter::removeFriend`

删除好友和会话.

```cpp
void DataCenter::removeFriend(const QString &userId)
{
    friendList->removeIf([=](const UserInfo& userInfo) {
        return userInfo.userId == userId;
    });
```

```
    chatSessionList->removeIf([=](const ChatSessionInfo& chatSessionInfo) {
        if (chatSessionInfo.userId == "") {
            // 群聊会话不受影响.
            return false;
        }
        if (chatSessionInfo.userId != userId) {
            return false;
        }
        if (chatSessionInfo.chatSessionId == this->currentChatSessionId) {
            // 如果当前要删除的会话就是正在选中的会话, 则清空当前会话内容
            emit this->clearCurrentSession();
        }
        return true;
    });
}
```

b) 定义 `DataCenter` 信号

```
// 删除好友完成
void deleteFriendDone(const QString& userId);
void clearCurrentSession();
```

c) 处理 `deleteFriendDone` 信号和 `clearCurrentSession` 信号.

在 `MainWidget::initData` 中, 更新界面显示.

- 更新会话列表
- 更新好友列表
- 更新当前会话的消息列表

```
////////////////////////////////////////////
/// 处理删除好友
////////////////////////////////////////////
connect(dataCenter, &DataCenter::deleteFriendDone, this, [=] () {
    this->updateChatSessionList();
    this->updateFriendList();
    LOG() << "删除好友完成";
});

connect(dataCenter, &DataCenter::clearCurrentSession, this, [=]() {
```

```
    sessionTitle->setText("");
    messageShowArea->clear();
    LOG() << "删除好友为当前会话，清空完成";
});
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/friend/remove_friend", [=](const
QHttpServerRequest& req) {
    return this->removeFriend(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::removeFriend(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::FriendRemoveReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 删除好友] request_id=" << pbReq.requestId() << ",
loginSessionId=" << pbReq.sessionId()
          << ", peerId=" << pbReq.peerId();
    // 构造响应
    bite_im::FriendRemoveRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

删除好友推送处理

A 删除 B 好友时, B 也会收到一个 websocket 的推送信息.

1) 客户端处理推送

实现 `NetClient::handleWsRemoveFriend`

> 此处 deleteFriendDone 信号已经被处理过了.

```
void NetClient::handleWsRemoveFriend(const QString &userId)
{
    // 服务器告知客户端, 某个好友把你删了.
    // 1. 从 DataCenter 中删除好友
    dataCenter->removeFriend(userId);
    // 2. 通知界面变化
    emit dataCenter->deleteFriendDone(userId);
}
```

2) 服务器实现逻辑

a) 在界面上添加按钮 "发送删除好友推送", 并实现槽函数.

```
void Widget::on_pushButton_9_clicked()
{
    WebsocketServer* websocketServer = WebsocketServer::getInstance();
    emit websocketServer->sendFriendRemove();
}
```

b) 定义 `WebsocketServer` 信号, 并处理

```
 void sendFriendRemove();
```

```
connect(this, &WebsocketServer::sendFriendRemove, this, [=]() {
    if (socket == nullptr || !socket->isValid()) {
        LOG() << "socket 对象无效!";
        return;
    }
```

```
6
7       bite_im::NotifyMessage notifyMessage;
8       notifyMessage.setNotifyEventId("");
9
    notifyMessage.setNotifyType(bite_im::NotifyTypeGadget::NotifyType::FRIEND_REMOV
    E_NOTIFY);
10
11      bite_im::NotifyFriendRemove friendRemove;
12      QString userId = "1000";
13      friendRemove.setUserId(userId);
14      notifyMessage.setFriendRemove(friendRemove);
15
16      QByteArray body = notifyMessage.serialize(&serializer);
17
18      socket->sendBinaryMessage(body);
19
20      LOG() << "通知对方好友通过, userId=" << userId;
21  });
```

c) 断开连接时断开信号槽

在 `connect(socket, &QWebSocket::disconnected, this, [=] () {}` 中调用.

```
1  disconnect(this, &WebsocketServer::sendFriendRemove, this, nullptr);
```

发送好友申请

点击 "申请好友按钮" , 触发

1) 客户端发送请求

a) 在 `UserInfoWidget` 构造函数中, 连接信号槽.

```
1  connect(addFriendBtn, &QPushButton::clicked, this,
   &UserInfoWidget::clickAddFriendBtn);
```

b) 实现 `UserInfoWidget::clickAddFriendBtn`

```
void UserInfoWidget::clickAddFriendBtn()
{
    // 1. 发送好友申请请求
    DataCenter* dataCenter = DataCenter::getInstance();
    dataCenter->addFriendApplyAsync(message.sender.userId);
    // 2. 关闭本窗口
    this->close();
}
```

c) 实现 `DataCenter::addFriendApplyAsync`

```
void DataCenter::addFriendApplyAsync(const QString &userId)
{
    netClient.addFriend(loginSessionId, userId);
}
```

d) 实现 `NetClient::addFriend`

接口定义

```
//添加好友--发送好友申请
message FriendAddReq {
    string request_id = 1;
    optional string session_id = 2;
    optional string user_id = 3;//申请人id
    string respondent_id = 4;//被申请人id
}
message FriendAddRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    string notify_event_id = 4;//通知事件id
}
```

函数实现

```
void NetClient::addFriend(const QString &loginSessionId, const QString &userId)
```

```
{
    // 1. 构造请求
    bite_im::FriendAddReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setRespondentId(userId);
    QByteArray body = req.serialize(&serializer);
    LOG() << "[添加好友] requestId=" << req.requestId() << " loginSessionId="
<< loginSessionId << " userId=" << userId;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this-
>sendHttpRequest("/service/friend/add_friend_apply", body);

    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto resp = this->handleHttpResponse<bite_im::FriendAddRsp>(httpResp);
        if (!resp) {
            return;
        }

        // b) 发送信号
        emit dataCenter->addFriendApplyDone();
    });
}
```

2) 客户端处理响应

a) 定义 `DataCenter` 信号

```
void addFriendApplyDone();
```

b) 在 `MainWidget::initData` 中, 处理 `addFriendApplyDone` 信号.

```
connect(dataCenter, &DataCenter::addFriendApplyDone, this, [=]() {
    Toast::showMessage("好友申请已发送!");
});
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/friend/add_friend_apply", [=](const
QHttpServerRequest& req) {
    return this->addFriendApply(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::addFriendApply(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::FriendAddReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 发送好友申请] request_id=" << pbReq.requestId() << ",
loginSessionId=" << pbReq.sessionId()
          << ", respondentId=" << pbReq.respondentId();
    // 构造响应
    bite_im::FriendAddRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

主界面逻辑 (2)

收到好友申请

1) 客户端处理推送

通过 websocket 收到 "好友申请通知", 并进行处理

a) 实现 `NetClient::handleWsAddFriendApplyReq`

```
void NetClient::handleWsAddFriendApplyReq(const model::UserInfo &userInfo)
{
    // 添加数据到 applyList 中
    QList<UserInfo>* applyList = dataCenter->getApplyList();
    if (applyList == nullptr) {
        qCritical() << TAG << "本地不存在 applyList";
        return;
    }
    // 添加到列表头部
    applyList->push_front(userInfo);
    // 通知界面新增一个内容
    emit dataCenter->receiveFriendApplyDone();
}
```

b) 实现 `DataCenter::getApplyList`

```
QList<UserInfo> *DataCenter::getApplyList()
{
    return applyList;
}
```

c) 定义 `DataCenter` 信号

```
void receiveFriendApplyDone();
```

d) 在 `MainWidget::initData` 中, 处理上述信号.

```
connect(dataCenter, &DataCenter::receiveFriendApplyDone, this, [=]() {
    Toast::showMessage("收到新的好友申请!");
    // 如果当前选中的标签页正好是好友申请, 则更新申请列表
```

```
    updateApplyList();
});
```

2) 服务器实现逻辑

a) 在界面上创建一个按钮, "发送好友申请推送", 并实现槽函数

```
void Widget::on_pushButton_2_clicked()
{
    WebsocketServer* websocketServer = WebsocketServer::getInstance();
    emit websocketServer->sendFriendApply();
}
```

b) 在 websocket 逻辑中处理上述信号

```
connect(this, &WebsocketServer::sendFriendApply, this, [=]() {
    if (socket == nullptr || !socket->isValid()) {
        LOG() << "socket 对象无效!";
        return;
    }
    bite_im::NotifyMessage notifyMessage;
    notifyMessage.setNotifyEventId("");

notifyMessage.setNotifyType(bite_im::NotifyTypeGadget::NotifyType::FRIEND_ADD_APPLY_NOTIFY);

    QByteArray avatar = loadImageToByteArray(":/image/defaultAvatar.png");

    bite_im::NotifyFriendAddApply friendAddApply;
    bite_im::UserInfo userInfo = makeUserInfo(100, avatar);
    friendAddApply.setUserInfo(userInfo);

    notifyMessage.setFriendAddApply(friendAddApply);
    QByteArray body = notifyMessage.serialize(&serializer);

    socket->sendBinaryMessage(body);

    LOG() << "发送好友申请, userId=" << userInfo.userId();
});
```

c) 在断开 websocket 连接时断开上述信号槽

```
disconnect(this, &WebsocketServer::sendFriendApply, this, nullptr);
```

同意好友申请

点击 "同意" 按钮, 触发下列逻辑.

1) 客户端发送请求

a) 在 `ApplyItem::ApplyItem` 的构造函数中, 连接信号槽

```
connect(acceptBtn, &QPushButton::clicked, this, &ApplyItem::acceptFriend);
```

b) 实现 `ApplyItem::acceptFriend`

```
void ApplyItem::acceptFriend()
{
    DataCenter* dataCenter = DataCenter::getInstance();
    // 1. 设置回调
    connect(dataCenter, &DataCenter::acceptFriendApplyDone, this,
&ApplyItem::acceptFriendDone,
            Qt::UniqueConnection);
    // 2. 发送网络请求, 同意好友申请
    dataCenter->acceptFriendApplyAsync(this->userId);
}
```

c) 实现 `DataCenter::acceptFriendApplyAsync`

```
void DataCenter::acceptFriendApplyAsync(const QString &userId)
{
    netClient.acceptFriendApply(loginSessionId, userId);
}
```

d) 实现 `NetClient::acceptFriendApply`

接口定义

```
//好友申请的处理
message FriendAddProcessReq {
    string request_id = 1;
    string notify_event_id = 2;//通知事件id
    bool agree = 3;//是否同意好友申请
    string apply_user_id = 4; //申请人的用户id
    optional string session_id = 5;
    optional string user_id = 6;
}
//   ++++++++++++++++++++++++++++++++
message FriendAddProcessRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    optional string new_session_id = 4; // 同意后会创建会话，向网关返回会话信息，用于通知双方会话的建立，这个字段客户端不需要关注
}
```

函数实现

```
void NetClient::acceptFriendApply(const QString &loginSessionId, const QString &userId)
{
    // 1. 构造请求
    bite_im::FriendAddProcessReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setNotifyEventId("");
    req.setAgree(true);
    req.setApplyUserId(userId);
    QByteArray body = req.serialize(&serializer);
    LOG() << "[同意好友申请] requestId=" << req.requestId() << " loginSessionId=" << loginSessionId << " userId=" << userId;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/friend/add_friend_process", body);
```

```
15
16      // 3. 处理 HTTP 响应
17      connect(httpResp, &QNetworkReply::finished, this, [=]() {
18          // a) 解析响应
19          auto resp = this->handleHttpResponseWithReason<bite_im::FriendAddProcessRsp>(httpResp);
20          if (!resp) {
21              return;
22          }
23
24          if (resp->success()) {
25              // b) 如果成功，就把这个用户从申请列表中删除，添加到好友列表中.
26              UserInfo applyUser = dataCenter->removeFromApplyList(userId);
27              QList<UserInfo>* friendList = dataCenter->getFriendList();
28              friendList->push_front(applyUser);
29          } else {
30              // c) 如果失败，则单纯从用户申请列表删除即可.
31              dataCenter->removeFromApplyList(userId);
32          }
33
34          // d) 发送信号，更新界面
35          emit dataCenter->acceptFriendApplyDone(userId, resp->errmsg());
36      });
37  }
```

2) 客户端处理响应

a) 实现 `DataCenter::removeFromApplyList`

```
1  UserInfo DataCenter::removeFromApplyList(const QString &userId)
2  {
3      if (applyList == nullptr) {
4          return UserInfo();
5      }
6      for (int i = 0; i < applyList->size(); ++i) {
7          if (applyList->at(i).userId == userId) {
8              UserInfo toRemove = applyList->at(i);
9              applyList->removeAt(i);
10             return toRemove;
11         }
12     }
13     return UserInfo();
14 }
```

b) 定义 `DataCenter` 信号

```
// 发送同意好友申请完成
void acceptFriendApplyDone(const QString& userId, const QString& reason);
```

c) 实现 `ApplyItem::acceptFriendDone`

```
void ApplyItem::acceptFriendDone(const QString& userId, const QString& reason)
{
    // 1. 同意的不是当前的选项，直接跳过
    if (userId != this->userId) {
        return;
    }
    // 2. 删除当前元素
    QWidget* parent = this->parentWidget();
    parent->layout()->removeWidget(this);

    // 3. 进行全局通知
    if (reason.isEmpty()) {
        Toast::showMessage("好友申请接受成功!");
    } else {
        Toast::showMessage("好友申请接受失败!" + reason);
    }

    // 4. 释放当前对象
    this->deleteLater();
}
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/friend/add_friend_process", [=](const
QHttpServerRequest& req) {
    return this->addFriendProcess(req);
});
```

b) 处理函数实现

```
QHttpServerResponse HttpServer::addFriendProcess(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::FriendAddProcessReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 处理好友申请] request_id=" << pbReq.requestId() << ", loginSessionId=" << pbReq.sessionId()
          << ", applyUserId=" << pbReq.applyUserId();
    // 构造响应
    bite_im::FriendAddProcessRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

📍 这里其实有个遗留问题:

当点击同意之后, 该用户被加入到 "好友列表" 中之后, 点击这个好友, 就会提示 "找不到对应的会话".

这个时候是因为对应的会话还没有创建.

创建会话的工作是由服务器自行完成的, 创建完毕后会通过 websocket 通知给客户端.

会话有了自然上述问题就迎刃而解. 这部分的具体处理逻辑在后续 "创建群聊" 中进一步完成.

拒绝好友申请

1) 客户端发送请求

a) 在 `ApplyItem::ApplyItem` 的构造函数中, 连接信号槽

```
connect(rejectBtn, &QPushButton::clicked, this, &ApplyItem::rejectFriend);
```

b) 实现 `ApplyItem::rejectFriend`

```
void ApplyItem::rejectFriend()
{
    DataCenter* dataCenter = DataCenter::getInstance();
    // 1. 设置回调
    connect(dataCenter, &DataCenter::rejectFriendApplyDone, this,
&ApplyItem::rejectFriendDone,
            Qt::UniqueConnection);
    // 2. 发送网络请求, 同意好友申请
    dataCenter->rejectFriendApplyAsync(this->userId);
}
```

c) 实现 `DataCenter::rejectFriendApplyAsync`

```
void DataCenter::rejectFriendApplyAsync(const QString &userId)
{
    netClient.rejectFriendApply(loginSessionId, userId);
}
```

d) 实现 `NetClient::rejectFriendApply`

接口定义 (和刚才的同意好友申请是同一套接口)

```
//好友申请的处理
message FriendAddProcessReq {
    string request_id = 1;
    string notify_event_id = 2;//通知事件id
    bool agree = 3;//是否同意好友申请
    string apply_user_id = 4; //申请人的用户id
    optional string session_id = 5;
    optional string user_id = 6;
}
//   ++++++++++++++++++++++++++++++++
```

```
message FriendAddProcessRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    optional string new_session_id = 4; // 同意后会创建会话，向网关返回会话信息，用于通知双方会话的建立，这个字段客户端不需要关注
}
```

函数实现

```
void NetClient::rejectFriendApply(const QString &loginSessionId, const QString &userId)
{
    // 1. 构造请求
    bite_im::FriendAddProcessReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setNotifyEventId("");
    req.setAgree(false);
    req.setApplyUserId(userId);
    QByteArray body = req.serialize(&serializer);
    LOG() << "[拒绝好友申请] requestId=" << req.requestId() << " loginSessionId=" << loginSessionId << " userId=" << userId;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/friend/add_friend_process", body);

    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto resp = this->handleHttpResponseWithReason<bite_im::FriendAddProcessRsp>(httpResp);
        if (!resp) {
            return;
        }

        // b) 把该用户从 applyList 中删除
        dataCenter->removeFromApplyList(userId);

        // c) 发送信号
        emit dataCenter->rejectFriendApplyDone(userId, resp->errmsg());
    });
}
```

2) 客户端处理响应

a) 实现 `DataCenter::removeFromApplyList`

> 这个函数上面 "同意好友申请" 中已经实现过, 直接调用即可.

b) 定义 `DataCenter` 信号

```
// 发送拒绝好友申请完成
void rejectFriendApplyDone(const QString& userId, const QString& reason);
```

c) 实现 `ApplyItem::rejectFriendDone` 的信号处理函数

```
void ApplyItem::rejectFriendDone(const QString& userId, const QString& reason)
{
    // 1. 同意的不是当前的选项, 直接跳过
    if (userId != this->userId) {
        return;
    }
    // 2. 删除当前元素
    QWidget* parent = this->parentWidget();
    parent->layout()->removeWidget(this);

    // 3. 进行全局通知
    if (reason.isEmpty()) {
        Toast::showMessage("好友申请已拒绝!");
    } else {
        Toast::showMessage("好友申请拒绝失败!" + reason);
    }

    // 4. 释放对象
    this->deleteLater();
}
```

3) 服务器实现逻辑

此处的逻辑和 "同意好友申请" 的服务器逻辑是一致的.

获取到好友申请处理结果

服务器通过 websocket 推送数据

1) 客户端处理推送

a) 实现 `NetClient::handleWsAddFriendResp`

```
void NetClient::handleWsAddFriendResp(const model::UserInfo &userInfo, bool agree)
{
    // 告诉客户端是否对方是否同意好友申请
    if (agree) {
        // 1. 添加数据到好友列表中
        QList<UserInfo>* friendList = dataCenter->getFriendList();
        if (friendList == nullptr) {
            qCritical() << TAG << "本地不存在 friendList";
            return;
        }
        friendList->push_front(userInfo);
        // 2. 告知界面同意好友申请
        emit dataCenter->receiveFriendProcessAccept(userInfo);
    } else {
        emit dataCenter->receiveFriendProcessReject(userInfo);
    }
}
```

b) 定义 `DataCenter` 信号

```
// 收到好友同意通知
void receiveFriendProcessAccept(const UserInfo& userInfo);

// 收到好友拒绝通知
void receiveFriendProcessReject(const UserInfo& userInfo);
```

c) 在 `MainWidget::initData` 中, 处理上述信号.

```
connect(dataCenter, &DataCenter::receiveFriendProcessAccept, this, [=](const UserInfo& userInfo) {
    // 更新好友列表数据
    updateFriendList();
    // 发送全局通知
    Toast::showMessage(userInfo.nickname + " 已经同意了好友申请!");
});

connect(dataCenter, &DataCenter::receiveFriendProcessReject, this, [=](const UserInfo& userInfo) {
    // 发送全局通知
    Toast::showMessage(userInfo.nickname + " 已经拒绝了好友申请!");
});
```

2) 服务器实现逻辑

a) 创建按钮, "发送好友通过通知" 和 "发送好友拒绝通知", 并创建槽函数

```
void Widget::on_pushButton_3_clicked()
{
    WebsocketServer* websocketServer = WebsocketServer::getInstance();
    emit websocketServer->sendFriendProcess(true);
}


void Widget::on_pushButton_4_clicked()
{
    WebsocketServer* websocketServer = WebsocketServer::getInstance();
    emit websocketServer->sendFriendProcess(false);
}
```

b) 定义上述信号

```
void sendFriendProcess(bool)
```

c) 在 websocket 逻辑中, 处理上述信号

```
connect(this, &WebsocketServer::sendFriendProcess, this, [=](bool agree) {
    if (socket == nullptr || !socket->isValid()) {
        LOG() << "socket 对象无效!";
        return;
    }

    bite_im::NotifyMessage notifyMessage;
    notifyMessage.setNotifyEventId("");

notifyMessage.setNotifyType(bite_im::NotifyTypeGadget::NotifyType::FRIEND_ADD_PROCESS_NOTIFY);

    QByteArray avatar = loadImageToByteArray(":/image/defaultAvatar.png");

    bite_im::NotifyFriendAddProcess friendAddProcess;
    bite_im::UserInfo userInfo = makeUserInfo(100, avatar);
    friendAddProcess.setAgree(agree);
    friendAddProcess.setUserInfo(userInfo);

    notifyMessage.setFriendProcessResult(friendAddProcess);
    QByteArray body = notifyMessage.serialize(&serializer);

    socket->sendBinaryMessage(body);

    LOG() << "通知对方好友通过, userId=" << userInfo.userId() << ", agree=" << agree;
});
```

d) 在 websocket 断开连接的逻辑中, 断开上述信号槽

```
disconnect(this, &WebsocketServer::sendFriendProcess, this, nullptr);
```

单聊消息会话详细信息界面逻辑

判定会话详情为单聊还是群聊

在 MainWidget 的 extraButton 的槽函数中做一个条件判断即可.

> 不需要和服务器通信

```cpp
connect(extraButton, &QPushButton::clicked, this, [=]() {
    // 测试阶段的代码
// #if TEST_GROUP_SESSION_DETAIL
//         GroupSessionDetailWidget* groupSessionDetailWidget = new GroupSessionDetailWidget();
//         groupSessionDetailWidget->show();
// #else
//         SessionDetailWidget* sessionDetailWidget = new SessionDetailWidget();
//         sessionDetailWidget->show();
// #endif

    // 1. 获取到当前会话信息
    ChatSessionInfo* sessionInfo = dataCenter->findChatSessionBySessionId(dataCenter->getCurrentChatSessionId());
    if (sessionInfo == nullptr) {
        LOG() << "当前选中的会话不存在, 不弹出详情对话框";
        return;
    }
    // 2. 判定是单聊还是群聊
    if (sessionInfo->userId != "") {
        // 单聊
        UserInfo* userInfo = dataCenter->getFriendById(sessionInfo->userId);
        if (userInfo == nullptr) {
            LOG() << "当前用户信息不存在, 不弹出详情对话框!";
            return;
        }
        SessionDetailWidget* detailWidget = new SessionDetailWidget(*sessionInfo);
        detailWidget->show();
    } else {
        // 群聊
        GroupSessionDetailWidget* detailWidget = new GroupSessionDetailWidget();
        detailWidget->show();
    }
});
```

获取对方好友详情

通过 friendList 查询即可.

> 不需要和服务器通信

在 `SessionDetailWidget` 构造函数中, 添加数据加载逻辑.

```
#if LOAD_DATA_FROM_NETWORK
    // 获取到当前的对方用户信息. 对方一定是咱们的好友.
    DataCenter* dataCenter = DataCenter::getInstance();
    UserInfo* userInfo = dataCenter->getFriendById(chatSessionInfo.userId);
    if (userInfo != nullptr) {
        AvatarItem* currentUser = new AvatarItem(userInfo->avatar, userInfo->nickname);
        layout->addWidget(currentUser, 0, 1);
    }
#endif
```

删除好友

和 UserInfoWidget 中的删除好友是一样的逻辑.

a) 在 `SessionDetailWidget` 构造函数中, 绑定信号槽

```
connect(deleteFriendBtn, &QPushButton::clicked, this, &SessionDetailWidget::clickDeleteFriendBtn);
```

b) 实现 `SessionDetailWidget::clickDeleteFriendBtn`

```
void SessionDetailWidget::clickDeleteFriendBtn()
{
    // 1. 弹出对话框, 确认是否要删除
    auto result = QMessageBox::warning(this, "确认删除", "确认删除该好友?", QMessageBox::Ok | QMessageBox::Cancel);
    if (result != QMessageBox::Ok) {
        LOG() << "删除好友取消";
        return;
    }
    // 2. 发送请求删除好友
    DataCenter* dataCenter = DataCenter::getInstance();
    dataCenter->deleteFriendAsync(chatSessionInfo.userId);
    // 3. 关闭本窗口
    this->close();
```

```
14 }
```

> 后续的 `deleteFriendAsync` 以及响应的处理, 已经在前面实现过了. 此处直接复用即可.

选择好友界面逻辑

选择联系人

> 纯客户端操作

1) 弹出选择联系人界面

a) 在 `SessionDetailWidget` 构造函数中, 给 `addBtn` 注册槽函数.

```
addBtn->setClicked([=]() {
    ChooseFriendDialog* dialog = new
ChooseFriendDialog(chatSessionInfo.userId);
    // 弹出模态对话框
    auto result = dialog->exec();
    if (result == QDialog::Accepted) {
        // 关闭当前窗口
        this->close();
    }
    delete dialog;
});
```

b) 实现 `AvatarItem::setClicked`

```
void AvatarItem::setClicked(std::function<void ()> slotFunc)
{
    connect(avatarBtn, &QPushButton::clicked, this, slotFunc);
}
```

2) 初始化待选择好友列表

在 `ChooseFriendDialog` 构造函数中, 新增加载数据逻辑

```
#if LOAD_DATA_FROM_NETWORK
    DataCenter* dataCenter = DataCenter::getInstance();
    QList<UserInfo>* friendList = dataCenter->getFriendList();
    if (friendList == nullptr) {
        return;
    }
    for (auto it = friendList->begin(); it != friendList->end(); ++it) {
        if (it->userId == this->userId) {
            // 对于当前会话的好友，直接就默认选中
            this->addFriend(it->userId, it->avatar, it->nickname, true);
            this->addSelectedFriend(it->userId, it->avatar, it->nickname);
        } else {
            this->addFriend(it->userId, it->avatar, it->nickname, false);
        }
    }
#endif
```

3) 在待选择好友列表中勾选某个元素, 添加到已选择列表

这里的逻辑已经在前面实现过

创建群聊会话

1) 客户端发送请求

a) 点击 "完成" 按钮, 发送创建会话请求

在 `ChooseFriendDialog::initRight` 中, 连接信号槽

```
connect(okBtn, &QPushButton::clicked, this, &ChooseFriendDialog::clickOkBtn);
```

b) 实现 `ChooseFriendDialog::clickOkBtn`

```
void ChooseFriendDialog::clickOkBtn()
{
    // 1. 生成会话成员列表
    QList<QString> userIdList = generateMemberList();
    if (userIdList.size() <= 2) {
        LOG() << "选中成员数目不足 2 个，不能创建群聊!";
```

```
        Toast::showMessage("选中成员数目不足 2 个，不能创建群聊!");
        return;
    }
    // 2. 发送请求
    DataCenter* dataCenter = DataCenter::getInstance();
    dataCenter->createChatSessionAsync(userIdList);
    // 3. 关闭窗口，返回 "QDialog::Accepted"
    this->accept();
}
```

c) 实现 `ChooseFriendDialog::generateMemberList`

```
QList<QString> ChooseFriendDialog::generateMemberList()
{
    QList<QString> result;
    // 1. 先把自己加进去
    DataCenter* dataCenter = DataCenter::getInstance();
    UserInfo* userInfo = dataCenter->getMyself();
    if (userInfo == nullptr) {
        qCritical() << TAG << "获取个人信息失败!";
        return result;
    }
    result.push_back(dataCenter->getMyself()->userId);

    // 2. 遍历 selectedContainer，获取到选中的好友id
    QVBoxLayout* layoutSelected = dynamic_cast<QVBoxLayout*>(selectedContainer->layout());
    for (int i = 0; i < layoutSelected->count(); ++i) {
        auto* item = layoutSelected->itemAt(i);
        if (item == nullptr || item->widget() == nullptr) {
            continue;
        }
        ChooseFriendItem* chooseFriendItem = dynamic_cast<ChooseFriendItem*>(item->widget());
        result.push_back(chooseFriendItem->userId);
    }
    return result;
}
```

d) 实现 `DataCenter::createChatSessionAsync`

```cpp
void DataCenter::createChatSessionAsync(const QList<QString> &userIdList)
{
    netClient.createChatSession(loginSessionId, userIdList);
}
```

e) 实现 `NetClient::createChatSession`

接口定义

```
//创建会话
message ChatSessionCreateReq {
    string request_id = 1;
    optional string session_id = 2;
    optional string user_id = 3;
    string chat_session_name = 4;
    //需要注意的是，这个列表中也必须包含创建者自己的用户ID
    repeated string member_id_list = 5;
}
message ChatSessionCreateRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    //这个字段属于后台之间的数据，给前端回复的时候不需要这个字段，会话信息通过通知进行发送
    optional ChatSessionInfo chat_session_info = 4;
}
```

函数实现

```cpp
void NetClient::createChatSession(const QString &loginSessionId, const
QList<QString> &userIdList)
{
    // 1. 构造请求
    bite_im::ChatSessionCreateReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setChatSessionName("新的群聊");
    req.memberIdList() = userIdList;

    QByteArray body = req.serialize(&serializer);
    LOG() << "[创建群聊会话] requestId=" << req.requestId() << " loginSessionId="
 << loginSessionId;
```

```
12
13      // 2. 发送 HTTP 请求
14      QNetworkReply* httpResp = this->sendHttpRequest("/service/friend/create_chat_session", body);
15
16      // 3. 处理 HTTP 响应
17      connect(httpResp, &QNetworkReply::finished, this, [=]() {
18          // a) 解析响应
19          auto resp = this->handleHttpResponse<bite_im::ChatSessionCreateRsp>(httpResp);
20          if (!resp) {
21              return;
22          }
23
24          // b) 修改会话列表操作不需要在这里进行. 虽然在 HTTP 响应这里会返回一个会话对象详情.
25          //    但是我们忽略这个响应
26          //    并且服务器会在创建会话完成后, 通过 ws 给会话中所有成员推送会话创建信息.
27
28          // c) 发送信号
29          emit dataCenter->createChatSessionDone();
30      });
31  }
```

2) 客户端处理响应

a) 定义 `DataCenter` 信号

```
1  void createChatSessionDone();
```

b) 在 `MainWidget::initData` 中, 处理上述信号

```
1  connect(dataCenter, &DataCenter::createChatSessionDone, this, [=]() {
2      // 发送全局通知
3      Toast::showMessage("创建群聊会话请求已经发送!");
4  });
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/friend/create_chat_session", [=](const
QHttpServerRequest& req) {
    return this->createChatSession(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::createChatSession(const QHttpServerRequest
&req)
{
    // 解析请求
    bite_im::ChatSessionCreateReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 处理创建会话] request_id=" << pbReq.requestId() << ",
loginSessionId=" << pbReq.sessionId();
    // 构造响应
    bite_im::ChatSessionCreateRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");

    bite_im::ChatSessionInfo chatSessionInfo;
    pbRsp.setChatSessionInfo(chatSessionInfo);
    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

4) 客户端实现 "取消" 按钮

在 `ChooseFriendDialog::initRight` 中注册信号槽.

```
 connect(cancelBtn, &QPushButton::clicked, this, [=]() {
    // 关闭对话框, 并返回 "QDialog::Rejected"
    this->reject();
});
```

收到群聊会话创建通知

当有用户创建会话时, 服务器会通过 websocket, 给所有会话成员的客户端, 发送会话创建通知. `NotifyNewChatSession`

1) 客户端处理推送

a) 实现 `NetClient::handleWsSessionCreate`

```
void NetClient::handleWsSessionCreate(const model::ChatSessionInfo &chatSessionInfo)
{
    QList<ChatSessionInfo>* chatSessionList = dataCenter->getChatSessionList();
    if (chatSessionList == nullptr) {
        qCritical() << TAG << "本地不存在 chatSessionList";
        return;
    }
    // 添加到列表头部
    chatSessionList->push_front(chatSessionInfo);
    // 通知界面新增一个内容
    emit dataCenter->receiveSessionCreateDone();
}
```

b) 定义 `DataCenter` 信号

```
void receiveSessionCreateDone();
```

c) 处理 `receiveSessionCreateDone` 信号

在 `MainWidget::initData` 中, 处理信号.

```
connect(dataCenter, &DataCenter::receiveSessionCreateDone, this, [=]() {
    // 更新会话列表数据
    updateChatSessionList();
    // 发送全局通知
    Toast::showMessage("您被拉入新的群聊中!");
});
```

2) 服务器实现逻辑

a) 创建按钮 "发送创建会话通知", 并定义槽函数.

```
void Widget::on_pushButton_5_clicked()
{
    WebsocketServer* websocketServer = WebsocketServer::getInstance();
    emit websocketServer->sendNewChatSession();
}
```

b) 定义 `WebsocketServer` 信号

```
void sendNewChatSession();
```

c) 在 websocket 处理逻辑中, 处理上述信号

```
connect(this, &WebsocketServer::sendNewChatSession, this, [=]() {
    if (socket == nullptr || !socket->isValid()) {
        LOG() << "socket 对象无效!";
        return;
    }

    QByteArray avatar = loadImageToByteArray(":/image/groupAvatar.png");

    bite_im::NotifyMessage notifyMessage;
    notifyMessage.setNotifyEventId("");

notifyMessage.setNotifyType(bite_im::NotifyTypeGadget::NotifyType::CHAT_SESSION
_CREATE_NOTIFY);
```

```
12
13      bite_im::NotifyNewChatSession newChatSession;
14
15      bite_im::ChatSessionInfo chatSessionInfo;
16      chatSessionInfo.setChatSessionId("2100");
17      chatSessionInfo.setSingleChatFriendId("");
18      chatSessionInfo.setChatSessionName("新的群聊");
19      // bite_im::MessageInfo messageInfo =
    makeEmptyMessageInfo(chatSessionInfo.chatSessionId(), avatar);
20      // chatSessionInfo.setPrevMessage(messageInfo);
21      chatSessionInfo.setAvatar(avatar);
22
23      newChatSession.setChatSessionInfo(chatSessionInfo);
24      notifyMessage.setNewChatSessionInfo(newChatSession);
25      QByteArray body = notifyMessage.serialize(&serializer);
26
27      socket->sendBinaryMessage(body);
28
29      LOG() << "通知新增消息会话, chatSessionId=" <<
    chatSessionInfo.chatSessionId();
30  });
```

d) 在 websocket 断开连接时, 断开信号槽连接

```
1  disconnect(this, &WebsocketServer::sendNewChatSession, this, nullptr);
```

实现群聊消息会话详细信息界面

获取群聊成员列表

1) 客户端发送请求

a) 在 `GroupSessionDetailWidget` 构造函数中, 加载数据

```
1  #if LOAD_DATA_FROM_NETWORK
2      DataCenter* dataCenter = DataCenter::getInstance();
3      connect(dataCenter, &DataCenter::getMemberListDone, this,
    &GroupSessionDetailWidget::initMembers);
4      dataCenter->getMemberListAsync(dataCenter->getCurrentChatSessionId());
```

```
#endif
```

b) 实现 `DataCenter::getMemberListAsync`

```
void DataCenter::getMemberListAsync(const QString &chatSessionId)
{
    netClient.getMemberList(loginSessionId, chatSessionId);
}
```

c) 实现 `NetClient::getMemberList`

接口定义

```
//获取会话成员列表
message GetChatSessionMemberReq {
    string request_id = 1;
    optional string session_id = 2;
    optional string user_id = 3;
    string chat_session_id = 4;
}
message GetChatSessionMemberRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    repeated UserInfo member_info_list = 4;
}
```

函数实现

```
void NetClient::getMemberList(const QString &loginSessionId, const QString
&chatSessionId)
{
    // 1. 构造请求
    bite_im::GetChatSessionMemberReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setChatSessionId(chatSessionId);
```

```
8
9       QByteArray body = req.serialize(&serializer);
10      LOG() << "[获取群聊成员列表] requestId=" << req.requestId() << "
    loginSessionId=" << loginSessionId
11            << " chatSessionId=" << chatSessionId;
12
13      // 2. 发送 HTTP 请求
14      QNetworkReply* httpResp = this-
    >sendHttpRequest("/service/friend/get_chat_session_member", body);
15
16      // 3. 处理 HTTP 响应
17      connect(httpResp, &QNetworkReply::finished, this, [=]() {
18          // a) 解析响应
19          auto resp = this->handleHttpResponse<bite_im::GetChatSessionMemberRsp>
    (httpResp);
20          if (!resp) {
21              return;
22          }
23
24          // b) 修改会话列表操作不需要在这里进行. 虽然在 HTTP 响应这里会返回一个会话对象详
    情.
25          //    但是我们忽略这个响应
26          //    并且服务器会在创建会话完成后, 通过 ws 给会话中所有成员推送会话创建信息.
27          dataCenter->resetMemberList(chatSessionId, resp);
28
29          // c) 发送信号
30          emit dataCenter->getMemberListDone(chatSessionId);
31      });
32  }
```

2) 客户端处理响应

a) 实现 `DataCenter::resetMemberList`

```
1   void DataCenter::resetMemberList(const QString& chatSessionId,
    std::shared_ptr<bite_im::GetChatSessionMemberRsp> resp)
2   {
3       // 先清空原有数据
4       QList<UserInfo>& curMemberList = (*memberList)[chatSessionId];
5       curMemberList.clear();
6       // 添加元素到成员列表中
7       for (auto& m : resp->memberInfoList()) {
8           UserInfo userInfo;
9           userInfo.load(m);
```

```
        curMemberList.push_back(userInfo);
    }
}
```

b) 定义 `DataCenter` 信号

```
void getMemberListDone();
```

c) 处理 `getMemberListDone` 信号

实现 `GroupSessionDetailWidget::initMembers`

```
void GroupSessionDetailWidget::initMembers(const QString &chatSessionId)
{
    DataCenter* dataCenter = DataCenter::getInstance();
    QList<UserInfo>* memberList = dataCenter->getMemberList(chatSessionId);
    if (memberList == nullptr) {
        LOG() << "获取成员列表失败! chatSessionId=" << chatSessionId;
        return;
    }
    for (auto it = memberList->begin(); it != memberList->end(); ++it) {
        AvatarItem* avatarItem = new AvatarItem(it->avatar, it->nickname);
        this->addMember(avatarItem);
    }
    // 群聊名称先写成固定值
    groupNameLabel->setText("新建群聊");
}
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/friend/get_chat_session_member", [=](const
QHttpServerRequest& req) {
    return this->getChatSessionMember(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::getChatSessionMember(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::GetChatSessionMemberReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 获取会话成员列表] request_id=" << pbReq.requestId() << ", loginSessionId=" << pbReq.sessionId()
          << ", chatSessionId=" << pbReq.chatSessionId();
    // 构造响应
    bite_im::GetChatSessionMemberRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");

    QByteArray avatar = loadImageToByteArray(":/image/defaultAvatar.png");

    auto& memberInfoList = pbRsp.memberInfoList();
    for (int i = 0; i < 10; ++i) {
        bite_im::UserInfo userInfo = makeUserInfo(1000 + i, avatar);
        memberInfoList.push_back(userInfo);
    }

    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

扩展功能

下列功能暂时不实现. 同学们可以自行扩展.

- 点击群成员头像查看详情
- 新增群聊成员
- 退出群聊
- 修改群聊名称

添加好友界面逻辑

搜索用户

1) 客户端发送请求

a) 在 `AddFriendDialog` 构造函数中, 连接信号槽.

```
connect(searchBtn, &QPushButton::clicked, this,
&AddFriendDialog::clickSearchBtn);
```

b) 实现 `clickSearchBtn`

```
void AddFriendDialog::clickSearchBtn()
{
    // 1. 获取到输入框的内容
    const QString& searchKey = searchEdit->text();
    if (searchKey.isEmpty()) {
        return;
    }
    DataCenter* dataCenter = DataCenter::getInstance();
    // 2. 创建处理响应的回调
    connect(dataCenter, &DataCenter::searchUserDone, this,
&AddFriendDialog::clickSearchBtnDone,
            Qt::UniqueConnection);
    // 3. 发送请求
    dataCenter->searchUserAsync(searchKey);
}
```

c) 实现 `DataCenter::searchUserAsync`

```
void DataCenter::searchUserAsync(const QString &searchKey)
{
    netClient.searchUser(loginSessionId, searchKey);
}
```

d) 实现 `NetClient::searchUser`

接口定义

```
//好友搜索
message FriendSearchReq {
    string request_id = 1;
    string search_key = 2;//就是名称模糊匹配关键字
    optional string session_id = 3;
    optional string user_id = 4;
}
message FriendSearchRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    repeated UserInfo user_info = 4;
}
```

函数实现

```
void NetClient::searchUser(const QString &loginSessionId, const QString &searchKey)
{
    // 1. 构造请求
    bite_im::FriendSearchReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setSearchKey(searchKey);

    QByteArray body = req.serialize(&serializer);
    LOG() << "[搜索用户] requestId=" << req.requestId() << " loginSessionId=" << loginSessionId
          << " searchKey=" << searchKey;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/friend/search_friend", body);

    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
```

```
        auto resp = this->handleHttpResponse<bite_im::FriendSearchRsp>
(httpResp);
        if (!resp) {
            return;
        }

        // b) 获取到的结果数据
        dataCenter->resetSearchUserResult(resp);

        // c) 发送信号
        emit dataCenter->searchUserDone();
    });
}
```

2) 客户端处理响应

a) 实现 `DataCenter::resetSearchUserResult` 和 `getSearchUserResult`

```
void
DataCenter::resetSearchUserResult(std::shared_ptr<bite_im::FriendSearchRsp>
resp)
{
    // 1. 创建出实例
    if (searchUserResult == nullptr) {
        searchUserResult = new QList<UserInfo>();
    }
    // 2. 清空之前的内容
    searchUserResult->clear();
    // 3. 添加当前响应的结果
    for (const auto& u : resp->userInfo()) {
        UserInfo userInfo;
        userInfo.load(u);
        searchUserResult->push_back(userInfo);
    }
}

QList<UserInfo> *DataCenter::getSearchUserResult()
{
    return searchUserResult;
}
```

b) 定义 `DataCenter` 的信号

```
void searchUserDone();
```

c) 处理 `searchUserDone` 信号

实现 `AddFriendDialog::clickSearchBtnDone`

```
void AddFriendDialog::clickSearchBtnDone()
{
    DataCenter* dataCenter = DataCenter::getInstance();
    // 1. 获取到搜索结果数据
    QList<UserInfo>* searchResult = dataCenter->getSearchUserResult();
    if (searchResult == nullptr) {
        LOG() << "获取好友搜索结果失败!";
        return;
    }
    // 2. 清空原有数据
    this->clear();
    // 3. 构造新数据
    for (const auto& userInfo : *searchResult) {
        this->addResult(userInfo);
    }
}
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/friend/search_friend", [=](const
QHttpServerRequest& req) {
    return this->searchFriend(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::searchFriend(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::FriendSearchReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 搜索用户] request_id=" << pbReq.requestId() << ",
loginSessionId=" << pbReq.sessionId()
          << ", searchKey=" << pbReq.searchKey();
    // 构造响应
    bite_im::FriendSearchRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");

    QByteArray avatar = loadImageToByteArray(":/image/defaultAvatar.png");

    auto& userInfoList = pbRsp.userInfo();
    for (int i = 0; i < 10; ++i) {
        bite_im::UserInfo userInfo = makeUserInfo(1000 + i, avatar);
        userInfoList.push_back(userInfo);
    }

    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

发送好友申请

1) 客户端发送请求

a) "添加好友" 按钮, 在 `FriendResultItem` 的构造函数中, 添加信号槽.

```
connect(addBtn, &QPushButton::clicked, this, &FriendResultItem::clickAddBtn);
```

b) 实现 `FriendResultItem::clickAddBtn`

```
void FriendResultItem::clickAddBtn()
{
    DataCenter* dataCenter = DataCenter::getInstance();
    // 1. 发送好友申请
    dataCenter->addFriendApplyAsync(userInfo.userId);
    // 2. 发送按钮设置为 "禁用状态"
    addBtn->setEnabled(false);
    addBtn->setText("已申请");
    addBtn->setStyleSheet("QPushButton { color: rgb(255, 255, 255); background-color: rgb(198, 198, 198); border-radius: 10px; }");
}
```

c) 实现 `DataCenter::addFriendApplyAsync`

> 前面已经实现过了. 此处直接调用.

2) 客户端处理响应

> 前面已经实现过了. 此处直接调用.

3) 服务器实现逻辑

> 前面已经实现过了. 此处直接调用.

历史消息界面逻辑 (1)

搜索历史消息 (1) - 按查询词搜索

1) 添加弹出对话框条件: 当前会话id 不为 "" 才弹出

```
connect(showHistoryBtn, &QPushButton::clicked, this, [=]() {
    const QString& currentChatSessionId = dataCenter->getCurrentChatSessionId();
    // 如果当前未选中任何会话, 不弹出历史消息对话框
    if (currentChatSessionId.isEmpty()) {
        return;
    }
    HistoryMessageWidget* widget = new HistoryMessageWidget();
    widget->show();
});
```

2) 客户端发送请求

a) 在 `HistoryMessageWidget` 构造函数中, 连接信号槽.

```
#if LOAD_DATA_FROM_NETWORK
    connect(searchBtn, &QPushButton::clicked, this,
&HistoryMessageWidget::clickSearchBtn);
#endif
```

b) 实现 `HistoryMessageWidget::clickSearchBtn`

```
void HistoryMessageWidget::clickSearchBtn()
{
    // 1. 处理响应
    DataCenter* dataCenter = DataCenter::getInstance();
    connect(dataCenter, &DataCenter::searchMessageDone, this,
&HistoryMessageWidget::clickSearchBtnDone);

    if (radioBtn1->isChecked()) {
        // 2. 按照词查询. 获取到输入框的查询词
        const QString& searchKey = searchEdit->text();
        if (searchKey.isEmpty()) {
            return;
        }
        // 3. 发送请求
        dataCenter->searchMessageAsync(searchKey);
    } else {
        // TODO
    }
}
```

c) 实现 `DataCenter::searchMessageAsync`

```
void DataCenter::searchMessageAsync(const QString &searchKey)
{
    netClient.searchMessage(loginSessionId, currentChatSessionId, searchKey);
```

```
}
```

d) 实现 `NetClient::searchMessage`

接口定义

```
message MsgSearchReq {
    string request_id = 1;
    optional string user_id = 2;
    optional string session_id = 3;
    string chat_session_id = 4;
    string search_key = 5;
}
message MsgSearchRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    repeated MessageInfo msg_list = 4;
}
```

函数实现

```
void NetClient::searchMessage(const QString &loginSessionId, const QString&
chatSessionId, const QString &searchKey)
{
    // 1. 构造请求
    bite_im::MsgSearchReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setChatSessionId(chatSessionId);
    req.setSearchKey(searchKey);

    QByteArray body = req.serialize(&serializer);
    LOG() << "[按词搜索消息] requestId=" << req.requestId() << " loginSessionId="
 << loginSessionId
          << " searchKey=" << searchKey;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this-
>sendHttpRequest("/service/message_storage/search_history", body);

```

```
    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto resp = this->handleHttpResponse<bite_im::MsgSearchRsp>(httpResp);
        if (!resp) {
            return;
        }

        // b) 获取到的结果数据
        dataCenter->resetSearchMessageResult(resp->msgList());

        // c) 发送信号
        emit dataCenter->searchMessageDone();
    });
}
```

3) 客户端处理响应

a) 实现 `DataCenter::resetSearchMessageResult`

```
void DataCenter::resetSearchMessageResult(const QList<bite_im::MessageInfo>&
msgList)
{
    // 1. 创建出实例
    if (searchMessageResult == nullptr) {
        searchMessageResult = new QList<Message>();
    }
    // 2. 清空之前的结果
    searchMessageResult->clear();
    // 3. 添加当前数据到结果中
    for (const auto& m : msgList) {
        Message message;
        message.load(m);
        searchMessageResult->push_back(message);
    }
}
```

b) 定义 `DataCenter` 信号

```
void searchMessageDone();
```

c) 处理 `searchMessageDone` 信号

实现 `HistoryMessageWidget::clickSearchBtnDone`

```
void HistoryMessageWidget::clickSearchBtnDone()
{
    DataCenter* dataCenter = DataCenter::getInstance();

    // 1. 获取到搜索结果列表
    QList<Message>* searchMessageResult = dataCenter->getSearchMessageResult();
    if (searchMessageResult == nullptr) {
        LOG() << "获取搜索结果为空";
        return;
    }
    // 2. 清空已有列表内容
    this->clear();
    // 3. 添加响应结果的内容
    for (const auto& message : *searchMessageResult) {
        this->addHistoryMessage(message);
    }
}
```

4) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/message_storage/search_history", [=](const
QHttpServerRequest& req) {
    return this->searchHistory(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::searchHistory(const QHttpServerRequest &req)
{
    // 解析请求
```

```
    bite_im::MsgSearchReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 搜索历史消息] request_id=" << pbReq.requestId() << ", loginSessionId=" << pbReq.sessionId()
          << ", chatSessionId=" << pbReq.chatSessionId() << ", searchKey=" << pbReq.searchKey();
    // 构造响应
    bite_im::MsgSearchRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");

    QByteArray avatar = loadImageToByteArray(":/image/defaultAvatar.png");

    auto& messageList = pbRsp.msgList();
    for (int i = 0; i < 10; ++i) {
        bite_im::MessageInfo messageInfo = makeMessageInfo(3000 + i, pbReq.chatSessionId(), avatar);
        messageList.push_back(messageInfo);
    }

    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

搜索历史消息(2) - 按时间范围搜索

1) 客户端发送请求

a) 实现 `HistoryMessageWidget::clickSearchBtn`

```
void HistoryMessageWidget::clickSearchBtn()
{
    // 1. 处理响应
    DataCenter* dataCenter = DataCenter::getInstance();
    connect(dataCenter, &DataCenter::searchMessageDone, this, &HistoryMessageWidget::clickSearchBtnDone);

    if (radioBtn1->isChecked()) {
```

```
        // 2. 按照词查询. 获取到输入框的查询词
        // ....
    } else {
        // 2. 按照时间查询. 获取输入框的时间日期
        auto begTime = begTimeEdit->dateTime();
        auto endTime = endTimeEdit->dateTime();
        if (begTime >= endTime) {
            Toast::showMessage("时间错误! 开始时间大于等于结束时间!");
            return;
        }
        // 3. 发送请求
        dataCenter->searchMessageByTimeAsync(begTime, endTime);
    }
}
```

b) 实现 `DataCenter::searchMessageByTimeAsync`

```
void DataCenter::searchMessageByTimeAsync(const QDateTime &begTime, const QDateTime &endTime)
{
    netClient.searchMessageByTime(loginSessionId, currentChatSessionId, begTime, endTime);
}
```

c) 实现 `NetClient::searchMessageByTime`

接口定义

```
message GetHistoryMsgReq {
    string request_id = 1;
    string chat_session_id = 2;
    int64 start_time = 3;
    int64 over_time = 4;
    optional string user_id = 5;
    optional string session_id = 6;
}
message GetHistoryMsgRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    repeated MessageInfo msg_list = 4;
```

```
}
```

函数实现

```
void NetClient::searchMessageByTime(const QString &loginSessionId, const QString &chatSessionId, const QDateTime &begTime, const QDateTime &endTime)
{
    // 1. 构造请求
    bite_im::GetHistoryMsgReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setChatSessionId(chatSessionId);
    req.setStartTime(begTime.toSecsSinceEpoch());
    req.setOverTime(endTime.toSecsSinceEpoch());

    QByteArray body = req.serialize(&serializer);
    LOG() << "[按时间搜索消息] requestId=" << req.requestId() << " loginSessionId=" << loginSessionId
          << " begTime=" << begTime << " endTime=" << endTime;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/message_storage/get_history", body);

    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto resp = this->handleHttpResponse<bite_im::GetHistoryMsgRsp>(httpResp);
        if (!resp) {
            return;
        }

        // b) 获取到的结果数据
        dataCenter->resetSearchMessageResult(resp->msgList());

        // c) 发送信号
        emit dataCenter->searchMessageDone();
    });
}
```

2) 客户端处理响应

> 此处已经实现过了, 直接复用.

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/message_storage/get_history", [=](const
QHttpServerRequest& req) {
    return this->getHistory(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::getHistory(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::GetHistoryMsgReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 获取历史消息] request_id=" << pbReq.requestId() << ",
loginSessionId=" << pbReq.sessionId()
          << ", chatSessionId=" << pbReq.chatSessionId();
    // 构造响应
    bite_im::GetHistoryMsgRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");

    QByteArray avatar = loadImageToByteArray(":/image/defaultAvatar.png");

    auto& messageList = pbRsp.msgList();
    for (int i = 0; i < 10; ++i) {
        bite_im::MessageInfo messageInfo = makeMessageInfo(3000 + i,
pbReq.chatSessionId(), avatar);
        messageList.push_back(messageInfo);
    }

    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
```

```
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

用户名登录/注册界面

生成验证码

创建 `VerifyCodeWidget` 类

```
class VerifyCodeWidget : public QWidget
{
    Q_OBJECT
public:
    explicit VerifyCodeWidget(QWidget *parent = nullptr);

    // 重写绘图事件. 绘制验证码
    void paintEvent(QPaintEvent *event) override;

    // 鼠标点击时也刷新验证码
    void mousePressEvent(QMouseEvent *event) override;

    // 刷新验证码
    void refreshVerifyCode();

    // 检测验证码是否匹配
    bool checkVerifyCode(const QString& code);
private:
    // 随机数生成器
    QRandomGenerator randomGenerator;

    // 保存验证码的值
    QString verifyCode = "";

    // 生成验证码
    QString generateVerifyCode();
signals:
};
```

生成验证码核心逻辑

- 生成随机字符串.
- 按照随机的颜色和位置绘制.
- 引入噪点和噪线.

```
VerifyCodeWidget::VerifyCodeWidget(QWidget *parent)
    : QWidget{parent}, randomGenerator(QDateTime::currentMSecsSinceEpoch()), verifyCode(generateVerifyCode())
{
    // 启动时也生成一个随机的验证码
    verifyCode = generateVerifyCode();
}

QString VerifyCodeWidget::generateVerifyCode()
{
    QString code;
    // 生成随机验证码序列
    for(int i = 0;i < 4;++i)
    {
        // 验证码只有大写字母，避免出现一些容易混淆的结果，比如 o 和 0，比如 1 I l 等.
        int init = 'A';
        code += static_cast<QChar>(init + randomGenerator.generate() % 26);
    }
    return code;
}

void VerifyCodeWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    QPen pen;
    QFont font("楷体",25,QFont::Bold,true);
    painter.setFont(font);

    // 画点：添加随机噪点
    for(int i = 0; i < 100; i++)
    {
        pen = QPen(QColor(randomGenerator.generate() % 256, randomGenerator.generate() % 256, randomGenerator.generate() % 256));
        painter.setPen(pen);
        painter.drawPoint(randomGenerator.generate() % 180, randomGenerator.generate() % 80);
    }

    // 画线：添加随机干扰线
    for(int i = 0; i < 5; i++)
    {
```

```
39          pen = QPen(QColor(randomGenerator.generate() % 256,
    randomGenerator.generate() % 256, randomGenerator.generate() % 256));
40          painter.setPen(pen);
41          painter.drawLine(randomGenerator.generate() % 180,
    randomGenerator.generate() % 80, randomGenerator.generate() % 180,
    randomGenerator.generate() % 80);
42      }
43
44      // 绘制验证码
45      for(int i = 0; i < verifyCode.size(); i++)
46      {
47          pen = QPen(QColor(randomGenerator.generate() % 255,
    randomGenerator.generate() % 255, randomGenerator.generate() % 255));
48          painter.setPen(pen);
49          painter.drawText(5+20*i, randomGenerator.generate() % 10, 30, 30,
    Qt::AlignCenter, QString(verifyCode[i]));
50      }
51  }
52
53  void VerifyCodeWidget::mousePressEvent(QMouseEvent *event)
54  {
55      refreshVerifyCode();
56  }
57
58  void VerifyCodeWidget::refreshVerifyCode()
59  {
60      verifyCode = generateVerifyCode();
61      update();
62  }
63
64  bool VerifyCodeWidget::checkVerifyCode(const QString &code)
65  {
66      // 按照大小写不敏感的方式进行匹配
67      return verifyCode.compare(code, Qt::CaseInsensitive) == 0;
68  }
```

登录逻辑

1) 客户端发送请求

a) 在 `LoginWidget` 构造函数中注册信号槽.

```
1  connect(submitBtn, &QPushButton::clicked, this, &LoginWidget::clickSubmitBtn);
```

b) 实现 `LoginWidget::clickSubmitBtn`

```
void LoginWidget::clickSubmitBtn()
{
    // 1. 获取到必要的输入框内容
    const QString& username = usernameEdit->text();
    const QString& password = passwordEdit->text();
    const QString& verifyCode = verifyCodeEdit->text();
    if (username.isEmpty() || password.isEmpty()) {
        // QMessageBox::warning(this, "提示", "用户名/密码不能为空!");
        Toast::showMessage("用户名/密码不能为空!");
        return;
    }
    // 2. 验证验证码是否匹配
    if (!verifyCodeWidget->checkVerifyCode(verifyCode)) {
        // QMessageBox::warning(this, "提示", "验证码不正确!");
        Toast::showMessage("验证码不正确!");
        // 同时刷新验证码
        // verifyCodeWidget->refreshVerifyCode();
        return;
    }
    // 3. 真正发起网络请求
    DataCenter* dataCenter = DataCenter::getInstance();
    if (isLoginMode) {
        connect(dataCenter, &DataCenter::userLoginDone, this,
&LoginWidget::userLoginDone, Qt::UniqueConnection);
        dataCenter->userLoginAsync(username, password);
    } else {
        connect(dataCenter, &DataCenter::userRegisterDone, this,
&LoginWidget::userRegisterDone, Qt::UniqueConnection);
        dataCenter->userRegisterAsync(username, password);
    }
}
```

c) 实现 `DataCenter::userLoginAsync`

```
void DataCenter::userLoginAsync(const QString &username, const QString
&password)
{
    netClient.userLogin(username, password);
}
```

d) 实现 `NetClient::userLogin`

接口定义

```
//用户名登录
message UserLoginReq {
    string request_id = 1;
    string nickname = 2;
    string password = 3;
    string verify_code_id = 4;
    string verify_code = 5;
}
message UserLoginRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    string login_session_id = 4;
}
```

函数实现

```
void NetClient::userLogin(const QString &username, const QString &password)
{
    // 1. 构造请求
    bite_im::UserLoginReq req;
    req.setRequestId(makeRequestId());
    req.setNickname(username);
    req.setPassword(password);

    QByteArray body = req.serialize(&serializer);
    LOG() << "[用户名登录] requestId=" << req.requestId() << " username=" << username << " password=" << password;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/user/username_login", body);

    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
```

```
18          auto resp = this->handleHttpResponseWithReason<bite_im::UserLoginRsp>
    (httpResp);
19          if (!resp) {
20              // 失败也发送一个信号，告知原因
21              emit dataCenter->userLoginDone(false, "网络故障");
22              return;
23          }
24          if (!resp->success()) {
25              emit dataCenter->userLoginDone(false, resp->errmsg());
26              return;
27          }
28
29          // b) 获取到的结果数据
30          dataCenter->resetLoginSessionId(resp->loginSessionId());
31
32          // c) 发送信号
33          emit dataCenter->userLoginDone(true, "");
34      });
35  }
```

2) 客户端处理响应

a) 实现 `DataCenter::resetLoginSessionId`

```
1  void DataCenter::resetLoginSessionId(const QString &loginSessionId)
2  {
3      this->loginSessionId = loginSessionId;
4      saveDataFile();
5  }
```

b) 定义 `DataCenter` 信号

```
1  // 用户名登录完成，参数表示成功失败
2  void userLoginDone(bool ok, const QString reason);
```

c) 处理 `userLoginDone` 信号

实现 `LoginWidget::userLoginDone`

```
void LoginWidget::userLoginDone(bool ok, QString reason)
{
    if (!ok) {
        Toast::showMessage("登录失败! " + reason);
        return;
    }
    // 1. 登录成功, 跳转到主界面
    MainWidget* mainWidget = MainWidget::getInstance();
    mainWidget->show();
    // 2. 关闭当前窗口
    this->close();
}
```

d) 修改 debug.h, 不再跳过登录窗口.

```
// 测试跳过登录窗口
#define TEST_SKIP_LOGIN 0
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/user/username_login", [=](const QHttpServerRequest&
req) {
    return this->userLogin(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::userLogin(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::UserLoginReq pbReq;
    pbReq.deserialize(&serializer, req.body());
```

```
    LOG() << "[REQ 用户名登录] request_id=" << pbReq.requestId() << ", username="
 << pbReq.nickname()
          << ", password=" << pbReq.password();
    // 构造响应
    bite_im::UserLoginRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");
    pbRsp.setLoginSessionId("testSessionId");

    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

注册逻辑

1) 客户端发送请求

a) 实现 `DataCenter::userRegisterAsync`

```
void DataCenter::userRegisterAsync(const QString &username, const QString
&password)
{
    netClient.userRegister(username, password);
}
```

b) 实现 `NetClient::userRegister`

接口定义

```
//用户名注册
message UserRegisterReq {
    string request_id = 1;
    string nickname = 2;
    string password = 3;
    string verify_code_id = 4;
```

```
    string verify_code = 5;
}
message UserRegisterRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
}
```

函数实现

```
void NetClient::userRegister(const QString &username, const QString &password)
{
    // 1. 构造请求
    bite_im::UserRegisterReq req;
    req.setRequestId(makeRequestId());
    req.setNickname(username);
    req.setPassword(password);

    QByteArray body = req.serialize(&serializer);
    LOG() << "[用户名注册] requestId=" << req.requestId() << " username=" << username << " password=" << password;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/user/username_register", body);

    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto resp = this->handleHttpResponseWithReason<bite_im::UserRegisterRsp>(httpResp);
        if (!resp) {
            emit dataCenter->userRegisterDone(false, "网络故障");
            return;
        }
        if (!resp->success()) {
            emit dataCenter->userRegisterDone(false, resp->errmsg());
            return;
        }

        // b) 获取到的结果数据.
        //    对于注册来说，不需要任何数据

        // c) 发送信号
```

```
        emit dataCenter->userRegisterDone(true, "");
    });
}
```

2) 客户端处理响应

a) 定义 `DataCenter` 信号

```
// 用户名注册完成, 参数表示成功失败
void userRegisterDone(bool ok, const QString reason);
```

b) 实现 `LoginWidget::userRegisterDone`

```
void LoginWidget::userRegisterDone(bool ok, QString reason)
{
    if (!ok) {
        Toast::showMessage("注册失败! " + reason);
        return;
    }
    // 提示注册成功
    // QMessageBox::warning(this, "提示", "注册成功!");
    Toast::showMessage("注册成功!");
    // 跳转到登录界面
    switchMode();
    // 清空输入框
    usernameEdit->setText("");
    passwordEdit->setText("");
    verifyCodeEdit->setText("");
    // 更新验证码
    verifyCodeWidget->refreshVerifyCode();
}
```

3) 实现服务器逻辑

a) 注册路由

```
httpServer.route("/service/user/username_register", [=](const
```

```
    QHttpServerRequest& req) {
    return this->userRegister(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::userRegister(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::UserRegisterReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 用户名注册] request_id=" << pbReq.requestId() << ", username="
 << pbReq.nickname()
          << ", password=" << pbReq.password();
    // 构造响应
    bite_im::UserRegisterRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");

    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

手机号登录/注册界面

获取短信验证

> 直接调用之前封装好的接口即可.

1) 客户端发送请求

a) 在 `PhoneLoginWidget` 构造函数中连接信号槽.

```
connect(verifyCodeBtn, &QPushButton::clicked, this,
&PhoneLoginWidget::sendVerifyCode);
```

b) 实现 `PhoneLoginWidget::sendVerifyCode`

```
void PhoneLoginWidget::sendVerifyCode()
{
    // 1. 获取到输入框内容
    const QString& phone = phoneEdit->text();
    if (phone.isEmpty()) {
        LOG() << "电话号码为空!";
        return;
    }
    // 2. 开始定时器
    timer->start(1000);
    // 3. 给服务器发请求, 获取验证码
    DataCenter* dataCenter = DataCenter::getInstance();
    connect(dataCenter, &DataCenter::getVerifyCodeDone, this,
&PhoneLoginWidget::sendVerifyCodeDone, Qt::UniqueConnection);
    dataCenter->getVerifyCodeAsync(phone);
}
```

c) 实现 `DataCenter::getVerifyCodeAsync`

> 前面已经实现过.

2) 客户端处理响应

实现 `PhoneLoginWidget::sendVerifyCodeDone`

```
void PhoneLoginWidget::sendVerifyCodeDone()
{
    // 给出提示即可
    Toast::showMessage("验证码请求已发送!");
}
```

3) 服务器实现逻辑

> 前面已经实现过.

登录逻辑

1) 客户端发送请求

a) 在 `PhoneLoginWidget` 构造函数中连接信号槽.

```
connect(submitBtn, &QPushButton::clicked, this,
&PhoneLoginWidget::clickSubmitBtn);
```

b) 实现 `PhoneLoginWidget::clickSubmitBtn` 处理函数

```
void PhoneLoginWidget::clickSubmitBtn()
{
    // 1. 获取到必要的内容
    const QString& phone = phoneEdit->text();
    const QString& verifyCode = verifyCodeEdit->text();
    if (phone.isEmpty() || verifyCode.isEmpty()) {
        Toast::showMessage("电话号码/短信验证码不能为空!");
        return;
    }
    // 2. 发送请求
    DataCenter* dataCenter = DataCenter::getInstance();
    if (isLoginMode) {
        connect(dataCenter, &DataCenter::phoneLoginDone, this,
&PhoneLoginWidget::phoneLoginDone, Qt::UniqueConnection);
        dataCenter->phoneLoginAsync(phone, verifyCode);
    } else {
        connect(dataCenter, &DataCenter::phoneRegisterDone, this,
&PhoneLoginWidget::phoneRegisterDone, Qt::UniqueConnection);
        dataCenter->phoneRegisterAsync(phone, verifyCode);
    }
}
```

c) 实现 `DataCenter::phoneLoginAsync`

```
void DataCenter::phoneLoginAsync(const QString &phone, const QString
&verifyCode)
```

```
{
    netClient.phoneLogin(phone, verifyCode);
}
```

d) 实现 `NetClient::phoneLogin`

接口定义

```
//手机号登录
message PhoneLoginReq {
    string request_id = 1;
    string phone_number = 2;
    string verify_code_id = 3;
    string verify_code = 4;
}
message PhoneLoginRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    string login_session_id = 4;
}
```

函数实现

```
void NetClient::phoneLogin(const QString &phone, const QString &verifyCode)
{
    // 1. 构造请求
    bite_im::PhoneLoginReq req;
    req.setRequestId(makeRequestId());
    req.setPhoneNumber(phone);
    req.setVerifyCodeId(dataCenter->getVerifyCodeId());
    req.setVerifyCode(verifyCode);

    QByteArray body = req.serialize(&serializer);
    LOG() << "[手机号登录] requestId=" << req.requestId() << " phone=" << phone
<< " verifyCodeId=" << req.verifyCodeId()
          << " verifyCode=" << req.verifyCode();

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this-
>sendHttpRequest("/service/user/phone_login", body);
```

```
16
17      // 3. 处理 HTTP 响应
18      connect(httpResp, &QNetworkReply::finished, this, [=]() {
19          // a) 解析响应
20          auto resp = this->handleHttpResponseWithReason<bite_im::PhoneLoginRsp>
    (httpResp);
21          if (!resp) {
22              emit dataCenter->phoneLoginDone(false, "网络故障");
23              return;
24          }
25          if (!resp->success()) {
26              emit dataCenter->phoneLoginDone(false, resp->errmsg());
27              return;
28          }
29
30          // b) 获取到的结果数据
31          dataCenter->resetLoginSessionId(resp->loginSessionId());
32
33          // c) 发送信号
34          emit dataCenter->phoneLoginDone(true, "");
35      });
36  }
```

2) 客户端处理响应

a) 实现 `DataCenter::resetLoginSessionId`

> 前面已经实现过了.

b) 定义 `DataCenter` 信号

```
1  // 电话登录完成, 参数表示成功失败
2  void phoneLoginDone(bool ok, const QString reason);
```

c) 实现 `PhoneLoginWidget::phoneLoginDone`

```
1  void PhoneLoginWidget::phoneLoginDone(bool ok, const QString reason)
2  {
3      if (!ok) {
```

```
        Toast::showMessage("登录失败！ " + reason);
        return;
    }
    // 1. 登录成功，跳转到主界面
    MainWidget* mainWidget = MainWidget::getInstance();
    mainWidget->show();
    // 2. 关闭当前窗口
    this->close();
}
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/user/phone_login", [=](const QHttpServerRequest&
req) {
    return this->phoneLogin(req);
});
```

b) 实现处理逻辑

```
QHttpServerResponse HttpServer::phoneLogin(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::PhoneLoginReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 手机号登录] request_id=" << pbReq.requestId() << ", phone="
<< pbReq.phoneNumber()
          << ", verifyCodeId=" << pbReq.verifyCodeId() << ", verifyCode=" <<
pbReq.verifyCode();
    // 构造响应
    bite_im::PhoneLoginRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");
    pbRsp.setLoginSessionId("testSessionId");

    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
```

```
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

注册逻辑

1) 客户端发送请求

a) 实现 `DataCenter::phoneRegisterAsync`

```
void DataCenter::phoneRegisterAsync(const QString &phone, const QString &verifyCode)
{
    netClient.phoneRegister(phone, verifyCode);
}
```

b) 实现 `NetClient::phoneRegister`

```
void NetClient::phoneRegister(const QString &phone, const QString &verifyCode)
{
    // 1. 构造请求
    bite_im::PhoneRegisterReq req;
    req.setRequestId(makeRequestId());
    req.setPhoneNumber(phone);
    req.setVerifyCodeId(dataCenter->getVerifyCodeId());
    req.setVerifyCode(verifyCode);

    QByteArray body = req.serialize(&serializer);
    LOG() << "[手机号登录] requestId=" << req.requestId() << " phone=" << phone << " verifyCodeId=" << req.verifyCodeId()
          << " verifyCode=" << req.verifyCode();

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/user/phone_register", body);

    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
```

```
        auto resp = this->handleHttpResponseWithReason<bite_im::PhoneRegisterRsp>(httpResp);
        if (!resp) {
            emit dataCenter->phoneRegisterDone(false, "网络故障");
            return;
        }
        if (!resp->success()) {
            emit dataCenter->phoneRegisterDone(false, resp->errmsg());
            return;
        }

        // b) 获取到的结果数据
        //    注册操作不需要获取到结果

        // c) 发送信号
        emit dataCenter->phoneRegisterDone(true, "");
    });
}
```

2) 客户端处理响应

a) 定义 `DataCenter` 信号

```
// 电话注册完成, 参数表示成功失败
void phoneRegisterDone(bool ok, const QString reason);
```

b) 处理 `phoneRegisterDone` 信号

实现 `PhoneLoginWidget::phoneRegisterDone`

```
void PhoneLoginWidget::phoneRegisterDone(bool ok, const QString reason)
{
    if (!ok) {
        Toast::showMessage("注册失败! " + reason);
        return;
    }
    // 提示注册成功
    Toast::showMessage("注册成功!");
    // 跳转到登录界面
    switchMode();
    // 清空输入框
```

```
    phoneEdit->setText("");
    verifyCodeEdit->setText("");
    // 更新定时器，恢复发送按钮的状态
    timer->stop();
    verifyCodeBtn->setText("发送验证码");
    verifyCodeBtn->setEnabled(true);
}

```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/user/phone_register", [=](const QHttpServerRequest& req) {
    return this->phoneRegister(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::phoneRegister(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::PhoneRegisterReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 电话注册] request_id=" << pbReq.requestId() << ", phone=" << pbReq.phoneNumber()
          << ", verifyCodeId=" << pbReq.verifyCodeId() << ", verifyCode=" << pbReq.verifyCode();
    // 构造响应
    bite_im::PhoneRegisterRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");

    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
```

```
    return httpResp;
}
```

聊天界面逻辑 (2)

异步获取文件内容

1) 客户端发送请求

a) 如果 `content` 为空 (比如这个消息是服务器推送来的), 还需要异步的从服务器获取到图片内容.

实现 `DataCenter::getSingleFileAsync`

```
void DataCenter::getSingleFileAsync(const QString &fileId)
{
    netClient.getSingleFile(loginSessionId, fileId);
}
```

b) 实现 `NetClient::getSingleFile`

接口定义

```
message GetSingleFileReq {
    string request_id = 1;
    string file_id = 2;
    optional string user_id = 3;
    optional string session_id = 4;
}
message GetSingleFileRsp {
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    FileDownloadData file_data = 4;
}
message FileDownloadData {
    string file_id = 1;
    bytes file_content = 2;
}
```

函数实现

```
void NetClient::getSingleFile(const QString &loginSessionId, const QString &fileId)
{
    // 1. 构造请求
    bite_im::GetSingleFileReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setFileId(fileId);

    QByteArray body = req.serialize(&serializer);
    LOG() << "[获取单个文件] requestId=" << req.requestId() << " loginSessionId=" << loginSessionId
          << " fileId=" << fileId;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/file/get_single_file", body);

    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto resp = this->handleHttpResponse<bite_im::GetSingleFileRsp>(httpResp);
        if (!resp) {
            return;
        }

        // b) 获取到的结果数据
        //    此处不需要单独设置到 DataCenter 中

        // c) 发送信号
        emit dataCenter->getSingleFileDone(fileId, resp->fileData().fileContent());
    });
}
```

2) 客户端响应

> 会在接下来的 图片消息/文件消息/语音消息中分别实现.

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/file/get_single_file", [=](const
QHttpServerRequest& req) {
    return this->getSingleFile(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::getSingleFile(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::GetSingleFileReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 获取单个文件] request_id=" << pbReq.requestId() <<
pbReq.fileId();
    // 构造响应
    bite_im::GetSingleFileRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");

    bite_im::FileDownloadData fileData;
    fileData.setFileId(pbReq.fileId());
    if (pbReq.fileId() == "imageFileId") {
        fileData.setFileContent(loadFileToByteArray(":/image/cat.jpg"));
    } else if (pbReq.fileId() == "fileFileId") {
        fileData.setFileContent(loadFileToByteArray(":/file/test.txt"));
    } else if (pbReq.fileId() == "soundFileId") {
        fileData.setFileContent(loadFileToByteArray(":/file/sound.pcm"));
    }

    pbRsp.setFileData(fileData);

    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
```

```
    return httpResp;
}
```

图片消息

1) 客户端发送请求

a) 在 `MessageEditArea::initSignalSlot` 中, 连接信号槽

```
// 处理点击发送图片
connect(sendImageBtn, &QPushButton::clicked, this,
&MessageEditArea::clickSendImageBtn);
```

b) 实现 `MessageEditArea::clickSendImageBtn`

```
void MessageEditArea::clickSendImageBtn()
{
    DataCenter* dataCenter = DataCenter::getInstance();

    // 1. 判定是否选中会话
    if (dataCenter->getCurrentChatSessionId().isEmpty()) {
        LOG() << "未选中任何会话, 不能发送图片";
        return;
    }

    // 2. 弹出对话框, 要求用户选择图片
    QString filter = "Image files (*.png *.jpeg *.jpg *.bmp)";
    QString imagePath = QFileDialog::getOpenFileName(this, "选择图片",
QDir::homePath(), filter);
    if (imagePath.isEmpty()) {
        LOG() << "取消图片选择";
        return;
    }

    // 3. 读取图片内容
    QByteArray content = loadImageToByteArray(imagePath);

    // 4. 发送消息
    dataCenter->sendImageMessageAsync(dataCenter->getCurrentChatSessionId(),
content);
```

```
24
25      // 5. 处理消息回调，仍然通过最上方的 addSelfMessage 来处理
26  }
```

c) 实现 `DataCenter::sendImageMessageAsync`

```
1  void DataCenter::sendImageMessageAsync(const QString &chatSessionId, const
   QByteArray &content)
2  {
3      netClient.sendMessage(loginSessionId, chatSessionId,
   MessageType::IMAGE_TYPE, content);
4  }
```

2) 客户端处理响应

发送消息之后, 服务器的相应最终会通过信号槽, 调用到 `addSelfMessage`.

此处重点是在 `addSelfMessage` 中 `MessageShowArea::addMessage` 内部对图片消息的适配

实现 `MessageShowArea` 中的 `MessageImageLabel`

```
1  // 表示一个内容区域
2  // 满足下列条件:
3  // 1. 根据图片内容，识别出图片的尺寸
4  // 2. 根据图片尺寸，进行缩放成合适显示的尺寸
5  // 3. 根据计算出的图片尺寸，设置父元素的高度
6  class MessageImageLabel : public QWidget
7  {
8      Q_OBJECT
9  public:
10     MessageImageLabel(const QString& fileId, const QByteArray& content, bool
   isLeft);
11     void updateUI(const QString& fileId, const QByteArray& content);
12
13     void paintEvent(QPaintEvent* ) override;
14
15 private:
16     QPushButton* imageBtn;
```

```
    QString fileId;
    QByteArray content;
    bool isLeft;
};
```

```
MessageImageLabel::MessageImageLabel(const QString& fileId, const QByteArray &content, bool isLeft)
    : fileId(fileId), content(content), isLeft(isLeft)
{
    this->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);

    imageBtn = new QPushButton(this);
    imageBtn->setStyleSheet("QPushButton { border: none;}");

#if LOAD_DATA_FROM_NETWORK
    if (content.isEmpty()) {
        DataCenter* dataCenter = DataCenter::getInstance();
        connect(dataCenter, &DataCenter::getSingleFileDone, this, &MessageImageLabel::updateUI, Qt::UniqueConnection);
        dataCenter->getSingleFileAsync(fileId);
    }
#endif
}

void MessageImageLabel::updateUI(const QString& fileId, const QByteArray& content)
{
    if (fileId != this->fileId) {
        // 需要更新的不是当前的控件.
        return;
    }
    this->content = content;
    this->update();
}

void MessageImageLabel::paintEvent(QPaintEvent *)
{
    // 1. 拿到父元素的宽度
    QObject* object = this->parent();
    if (!object->isWidgetType()) {
        return;
    }
    QWidget* parent = dynamic_cast<QWidget*>(object);
    // 宽度设置为父元素的 60% (这里就设定为固定值也可以)
    int width = parent->width() * 0.6;
```

```
38
39      // 2. 构造 QImage 对象. 如果 content 为空, 则需要从网络加载. 此时给一个默认的图片.
40      QImage image;
41      if (content.isEmpty()) {
42          QByteArray tmpContent = loadFileToByteArray(":/image/image.png");
43          image.loadFromData(tmpContent);
44      } else {
45          image.loadFromData(content);  // 这里 QImage 会猜 content 是 png 还是
    jpg 等格式.
46      }
47
48      // 3. 图片进行缩放
49      int height = ((double)image.height() / image.width()) * width;
50      QPixmap pixmap = QPixmap::fromImage(image);
51      imageBtn->setIconSize(QSize(width, height));
52      imageBtn->setIcon(QIcon(pixmap));
53
54      // 4. 设置父元素的高度
55      parent->setFixedHeight(height + 50);
56
57      // 5. 设置图片位置
58      if (isLeft) {
59          imageBtn->setGeometry(10, 0, width, height);
60      } else {
61          int leftPos = this->width() - width - 10;
62          imageBtn->setGeometry(leftPos, 0, width, height);
63      }
64  }
```

3) 服务器实现逻辑

a) 在界面上添加按钮, "发送图片消息", 并实现槽函数

```
1  void Widget::on_pushButton_7_clicked()
2  {
3      WebsocketServer* websocketServer = WebsocketServer::getInstance();
4      emit websocketServer->sendImageResp();
5  }
```

b) 定义 `WebsocketServer` 信号

```
void sendImageResp();
```

c) 在 websocket 逻辑中, 添加发送图片逻辑

```
connect(this, &WebsocketServer::sendImageResp, this, [=]() {
    if (socket == nullptr || !socket->isValid()) {
        LOG() << "socket 对象无效!";
        return;
    }
    bite_im::NotifyMessage notifyMessage;
    notifyMessage.setNotifyEventId("");

notifyMessage.setNotifyType(bite_im::NotifyTypeGadget::NotifyType::CHAT_MESSAGE
_NOTIFY);

    bite_im::NotifyNewMessage newMessage;
    bite_im::MessageInfo messageInfo = makeImageMessageInfo(this-
>messageIndex++, "2000", avatar);
    newMessage.setMessageInfo(messageInfo);

    notifyMessage.setNewMessageInfo(newMessage);
    QByteArray body = notifyMessage.serialize(&serializer);

    socket->sendBinaryMessage(body);

    LOG() << "发送图片消息响应";
});
```

```
// 构造一个图片消息对象
bite_im::MessageInfo makeImageMessageInfo(int index, const QString&
chatSessionId, const QByteArray& avatar) {
    bite_im::MessageInfo messageInfo;
    messageInfo.setChatSessionId(chatSessionId);
    messageInfo.setTimestamp(QDateTime::currentMSecsSinceEpoch() / 1000);
    messageInfo.setMessageId(QString::number(3000 + index));

    bite_im::UserInfo sender = makeUserInfo(index, avatar);
    messageInfo.setSender(sender);

    bite_im::MessageContent messageContent;
```

```
12
    messageContent.setMessageType(bite_im::MessageTypeGadget::MessageType::IMAGE);
13      bite_im::ImageMessageInfo imageMessageInfo;
14      imageMessageInfo.setFileId("testFileId");
15      QByteArray content = loadFileToByteArray(":/image/cat.jpg");
16      imageMessageInfo.setImageContent(content);
17      messageContent.setImageMessage(imageMessageInfo);
18      messageInfo.setMessage(messageContent);
19
20      return messageInfo;
21  }
```

d) 在断开 websocket 连接的逻辑中断开上述信号槽

```
1  disconnect(this, &WebsocketServer::sendImageResp, this, nullptr);
```

文件消息

1) 客户端发送请求

a) 在 `MessageEditArea::initSignalSlot` 添加信号槽

```
1  // 处理点击发送文件
2  connect(sendFileBtn, &QPushButton::clicked, this,
   &MessageEditArea::clickSendFileBtn);
```

b) 实现 `MessageEditArea::clickSendFileBtn`

```
1  void MessageEditArea::clickSendFileBtn()
2  {
3      DataCenter* dataCenter = DataCenter::getInstance();
4
5      // 1. 判定是否选中会话
6      if (dataCenter->getCurrentChatSessionId().isEmpty()) {
7          LOG() << "未选中任何会话, 不能发送文件";
```

```
        return;
    }

    // 2. 弹出对话框, 要求用户选择文件
    QString filter = "*";
    QString path = QFileDialog::getOpenFileName(this, "选择文件",
QDir::homePath(), filter);
    if (path.isEmpty()) {
        LOG() << "取消文件选择";
        return;
    }

    // 3. 读取文件内容
    QByteArray content = loadFileToByteArray(path);

    // 4. 获取到文件名
    QFileInfo fileInfo(path);
    const QString& fileName = fileInfo.fileName();

    // 5. 发送消息
    dataCenter->sendFileMessageAsync(dataCenter->getCurrentChatSessionId(),
fileName, content);
}
```

c) 实现 `DataCenter::sendFileMessageAsync`

```
void DataCenter::sendFileMessageAsync(const QString &chatSessionId, const
QString &fileName, const QByteArray &content)
{
    netClient.sendMessage(loginSessionId, chatSessionId,
MessageType::FILE_TYPE, content, fileName);
}
```

2) 客户端处理响应

发送消息之后, 服务器的相应最终会通过信号槽, 调用到 `addSelfMessage`.

此处重点是在 `addSelfMessage` 中 `MessageShowArea::addMessage` 内部对图片消息的适配

a) 在 `MessageContentLabel` 构造函数中添加逻辑, 异步加载文件内容.

```
#if LOAD_DATA_FROM_NETWORK
    if (messageType == TEXT_TYPE) {
        return;
    }
    if (content.isEmpty()) {
        DataCenter* dataCenter = DataCenter::getInstance();
        connect(dataCenter, &DataCenter::getSingleFileDone, this,
&MessageContentLabel::updateUI);
        dataCenter->getSingleFileAsync(fileId);
    }
#endif
```

b) 实现 `MessageContentLabel::updateUI`

```
void MessageContentLabel::updateUI(const QString &fileId, const QByteArray
&content)
{
    if (fileId != this->fileId) {
        // 要修改的内容不是这个控件
        return;
    }
    this->content = content;
    this->update();
}
```

c) 实现鼠标左键点击文件消息, 触发文件另存为.

```
void MessageContentLabel::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        if (messageType == SPEECH_TYPE) {
            // TODO
        } else if (messageType == FILE_TYPE) {
            // 如果是文件, 点击后触发另存为
            if (content.isEmpty()) {
                Toast::showMessage("数据加载中, 请稍后");
                return;
```

```
            }
            saveAsFile(content);
        } else {
            // 文本消息啥都不做.
        }
    } else if (event->button() == Qt::RightButton) {
        // 弹出菜单
        // 通过 contextMenuEvent 实现.
    } else {
        LOG() << "不响应其他鼠标按键";
    }
}
```

```
void MessageContentLabel::saveAsFile(const QByteArray& content)
{
    // 1. 弹出对话框, 选择位置
    QString filePath = QFileDialog::getSaveFileName(this, "另存为",
QDir::homePath(), "*");
    if (filePath.isEmpty()) {
        LOG() << "取消选择保存文件的路径";
        return;
    }
    // 2. 保存文件
    writeByteArrayToFile(filePath, content);
}
```

3) 服务器实现逻辑

a) 在界面上添加按钮, "发送文件消息", 并实现槽函数

```
void Widget::on_pushButton_8_clicked()
{
    WebsocketServer* websocketServer = WebsocketServer::getInstance();
    emit websocketServer->sendFileResp();
}
```

b) 定义 `WebsocketServer` 信号

```
void sendFileResp();
```

c) 在 websocket 逻辑中, 添加发送文件逻辑

```
connect(this, &WebsocketServer::sendFileResp, this, [=]() {
    if (socket == nullptr || !socket->isValid()) {
        LOG() << "socket 对象无效!";
        return;
    }
    bite_im::NotifyMessage notifyMessage;
    notifyMessage.setNotifyEventId("");

notifyMessage.setNotifyType(bite_im::NotifyTypeGadget::NotifyType::CHAT_MESSAGE
_NOTIFY);

    bite_im::NotifyNewMessage newMessage;
    bite_im::MessageInfo messageInfo = makeFileMessageInfo(this-
>messageIndex++, "2000", avatar);
    newMessage.setMessageInfo(messageInfo);

    notifyMessage.setNewMessageInfo(newMessage);
    QByteArray body = notifyMessage.serialize(&serializer);

    socket->sendBinaryMessage(body);

    LOG() << "发送文件消息响应";
});
```

```
// 构造一个文件消息对象
bite_im::MessageInfo makeFileMessageInfo(int index, const QString&
chatSessionId, const QByteArray& avatar) {
    bite_im::MessageInfo messageInfo;
    messageInfo.setChatSessionId(chatSessionId);
    messageInfo.setTimestamp(QDateTime::currentMSecsSinceEpoch() / 1000);
    messageInfo.setMessageId(QString::number(3000 + index));

    bite_im::UserInfo sender = makeUserInfo(index, avatar);
    messageInfo.setSender(sender);

    bite_im::MessageContent messageContent;

messageContent.setMessageType(bite_im::MessageTypeGadget::MessageType::FILE);
```

```
    bite_im::FileMessageInfo fileMessageInfo;
    fileMessageInfo.setFileId("testFileId");
    fileMessageInfo.setFileName("test.txt");
    QByteArray content = loadFileToByteArray(":/file/test.txt");
    fileMessageInfo.setFileContents(content);
    fileMessageInfo.setFileSize(content.size());
    messageContent.setFileMessage(fileMessageInfo);
    messageInfo.setMessage(messageContent);

    return messageInfo;
}
```

d) 在 websocket 断开连接时, 释放信号槽

```
disconnect(this, &WebsocketServer::sendFileResp, this, nullptr);
```

语音消息

1) 录制语音

Qt 录制语音提供两个方案:

- QMediaRecorder
- QAudioSource

其中 QMediaRecorder 方案是 Qt6 新增方案, 目前使用体验, 感觉存在一些不好处理的 bug (比如设置采样率, 声道等参数不生效).

因此我们使用 QAudioSource 实现.

考虑到语音识别需求, 需要使咱们录制的声音符合百度语音识别 SDK 的要求.

注意：

1. 关于参数：如果相关音频参数不符合要求，可以使用ffmpeg工具进行转码
   - 采样率： 百度语音识别一般仅支持16000的采样率。即1秒采样16000次。
   - 位深： 无损音频格式pcm和wav可以设置，百度语音识别使用16bits 小端序 ， 即2个字节记录1/16000 s的音频数据。
   - 声道： 百度语音识别仅支持单声道。
2. 语音识别返回结果与音频内容不匹配，例如："嗨嗨嗨"、"嗯嗯嗯嗯嗯"、"什么"等错误返回
   - **解决方法**：排查音频采样率、声道、格式等参数是否符合接口规范。如与要求不符，需要用工具对音频进行转码。
3. 在使用之前一定先过一遍官方文档：https://ai.baidu.com/ai-doc/SPEECH/dlbxfrs5o

创建 `SoundRecorder` 类

```
1  class SoundRecorder : public QObject
2  {
3      Q_OBJECT
4
5      const QString RECORD_PATH =
   QStandardPaths::writableLocation(QStandardPaths::AppDataLocation) +
   "/sound/tmpRecord.pcm";
6      const QString PLAY_PATH =
   QStandardPaths::writableLocation(QStandardPaths::AppDataLocation) +
   "/sound/tmpPlay.pcm";
7
8  public:
9      ~SoundRecorder();
10     static SoundRecorder* getInstance();
11
12 private:
13     static SoundRecorder* instance;
14     explicit SoundRecorder(QObject *parent = nullptr);
15
16 public:
17     // 开始录制
18     void startRecord();
19     // 停止录制
20     void stopRecord();
21
22 signals:
23     // 录制完毕后发送这个信号
24     void soundRecordDone(const QString& path);
25
```

```
private:
    QFile soundFile;
    QAudioSource* audioSource;
};
```

```
////////////////////////////////////////////
/// 单例模式
////////////////////////////////////////////
SoundRecorder* SoundRecorder::instance = nullptr;

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

// 播放参考 https://www.cnblogs.com/tony-yang-flutter/p/16477212.html
// 录制参考 https://doc.qt.io/qt-6/qaudiosource.html
SoundRecorder::SoundRecorder(QObject *parent)
    : QObject{parent} {
    // 1. 创建目录
    QDir soundRootPath(QStandardPaths::writableLocation(QStandardPaths::AppDataLocation));
    soundRootPath.mkdir("sound");

    // 2. 初始化录制模块
    soundFile.setFileName(RECORD_PATH);

    QAudioFormat inputFormat;
    inputFormat.setSampleRate(16000);
    inputFormat.setChannelCount(1);
    inputFormat.setSampleFormat(QAudioFormat::Int16);

    QAudioDevice info = QMediaDevices::defaultAudioInput();
    if (!info.isFormatSupported(inputFormat)) {
        LOG() << "录制设备，格式不支持!";
        return;
    }

    audioSource = new QAudioSource(inputFormat, this);
    connect(audioSource, &QAudioSource::stateChanged, this, [=](QtAudio::State state) {
```

```
        if (state == QtAudio::StoppedState) {
            // 录制完毕
            if (audioSource->error() != QAudio::NoError) {
                LOG() << audioSource->error();
            }
        }
    });
}

void SoundRecorder::startRecord() {
    soundFile.open( QIODevice::WriteOnly | QIODevice::Truncate );
    audioSource->start(&soundFile);
}

void SoundRecorder::stopRecord() {
    audioSource->stop();
    soundFile.close();
    emit this->soundRecordDone(RECORD_PATH);
}
```

此时录制的语音文件是 pcm 格式的原始音频数据. 还不能通过第三方播放器播放.

只能通过下列代码来实现播放功能.

2) 播放语音

继续在 `SoundRecorder` 上添加代码

```
private:
    QAudioSink *audioSink;
    QMediaDevices *outputDevices;
    QAudioDevice outputDevice;
    QFile inputFile;
signals:
    // 播放完毕发送这个信号
    void soundPlayDone();
public:
    // 开始播放
    void startPlay(const QByteArray& content);
    // 停止播放
    void stopPlay();
```

```cpp
SoundRecorder::SoundRecorder(QObject *parent)
    : QObject{parent} {
    // 1. 创建目录
    QDir soundRootPath(QStandardPaths::writableLocation(QStandardPaths::AppDataLocation));
    soundRootPath.mkdir("sound");

    // 2. 初始化播放模块
    outputDevices = new QMediaDevices(this);
    outputDevice = outputDevices->defaultAudioOutput();
    QAudioFormat outputFormat;
    outputFormat.setSampleRate(16000);
    outputFormat.setChannelCount(1);
    outputFormat.setSampleFormat(QAudioFormat::Int16);
    if (!outputDevice.isFormatSupported(outputFormat)) {
        LOG() << "播放设备，格式不支持";
        return;
    }
    audioSink = new QAudioSink(outputDevice, outputFormat);

    connect(audioSink, &QAudioSink::stateChanged, this, [=](QtAudio::State state) {
        if (state == QtAudio::IdleState) {
            LOG() << "IdleState";
            this->stopPlay();
            emit this->soundPlayDone();
        } else if (state == QAudio::ActiveState) {
            LOG() << "ActiveState";
        } else if (state == QAudio::StoppedState) {
            LOG() << "StoppedState";
            if (audioSink->error() != QtAudio::NoError) {
                LOG() << audioSink->error();
            }
        }
    });

    // 3. 初始化录制模块
    soundFile.setFileName(RECORD_PATH);

    QAudioFormat inputFormat;
    inputFormat.setSampleRate(16000);
    inputFormat.setChannelCount(1);
    inputFormat.setSampleFormat(QAudioFormat::Int16);
```

```
42
43      QAudioDevice info = QMediaDevices::defaultAudioInput();
44      if (!info.isFormatSupported(inputFormat)) {
45          LOG() << "录制设备, 格式不支持!";
46      }
47
48      audioSource = new QAudioSource(inputFormat, this);
49      connect(audioSource, &QAudioSource::stateChanged, this, [=](QtAudio::State
    state) {
50          if (state == QtAudio::StoppedState) {
51              // 录制完毕
52              if (audioSource->error() != QAudio::NoError) {
53                  LOG() << audioSource->error();
54              }
55          }
56      });
57  }
58
59   void SoundRecorder::startPlay(const QByteArray& content) {
60      if (content.isEmpty()) {
61          Toast::showMessage("数据加载中, 请稍后播放");
62          return;
63      }
64      // 1. 把数据写入到临时文件
65      model::writeByteArrayToFile(PLAY_PATH, content);
66
67      // 2. 播放语音
68      inputFile.setFileName(PLAY_PATH);
69      inputFile.open(QIODevice::ReadOnly);
70      audioSink->start(&inputFile);
71      // LOG() << "startPlay! n=" << n << " error: " << device->errorString();
72  }
73
74  void SoundRecorder::stopPlay() {
75      audioSink->stop();
76      inputFile.close();
77  }
```

3) 客户端发送请求

a) 在 `MessageEditArea::initSignalSlot` 注册信号槽.

按下录音按钮开始录制, 释放录音按钮则停止录制.

```
// 处理录制语音
SoundRecorder* soundRecorder = SoundRecorder::getInstance();
connect(sendSoundBtn, &QPushButton::pressed, this,
&MessageEditArea::soundRecordPressed);
connect(sendSoundBtn, &QPushButton::released, this,
&MessageEditArea::soundRecordReleased);
connect(soundRecorder, &SoundRecorder::soundRecordDone, this,
&MessageEditArea::sendSound);
```

b) 实现 `MessageEditArea::soundRecordPressed`

```
void MessageEditArea::soundRecordPressed()
{
    // LOG() << "按下";
    sendSoundBtn->setIcon(QIcon(":/image/sound_active.png"));
    DataCenter* dataCenter = DataCenter::getInstance();
    if (dataCenter->getCurrentChatSessionId().isEmpty()) {
        LOG() << "未选中会话, 不能发送语音消息!";
        return;
    }
    SoundRecorder* soundRecorder = SoundRecorder::getInstance();
    soundRecorder->startRecord();

    // 隐藏输入框, 显示提示 label
    textEdit->hide();
    tipLabel->show();
}
```

c) 实现 `MessageEditArea::soundRecordReleased`

```
void MessageEditArea::soundRecordReleased()
{
    // LOG() << "释放";
    sendSoundBtn->setIcon(QIcon(":/image/sound.png"));
    DataCenter* dataCenter = DataCenter::getInstance();
    if (dataCenter->getCurrentChatSessionId().isEmpty()) {
        LOG() << "未选中会话, 不能发送语音消息!";
        return;
    }
    SoundRecorder* soundRecorder = SoundRecorder::getInstance();
```

```
    soundRecorder->stopRecord();

    // 显示输入框，隐藏提示 label
    textEdit->show();
    tipLabel->hide();
}
```

在 `stopRecord` 中会触发 `soundRecordDone` 信号, 进一步的触发 `MessageEditArea::sendSound`

d) 实现 `MessageEditArea::sendSound`

```
void MessageEditArea::sendSound(const QString &path)
{
    DataCenter* dataCenter = DataCenter::getInstance();
    // 1. 读取文件内容
    QByteArray content = loadFileToByteArray(path);
    // 2. 通过网络发送消息
    dataCenter->sendSpeechMessageAsync(dataCenter->getCurrentChatSessionId(), content);
    // 3. 消息被服务器接受后，仍然通过上面的 addSelfMessage 来处理
}
```

e) 实现 `DataCenter::sendSpeechMessageAsync`

```
void DataCenter::sendSpeechMessageAsync(const QString &chatSessionId, const QByteArray &content)
{
    netClient.sendMessage(loginSessionId, chatSessionId, MessageType::SPEECH_TYPE, content);
}
```

4) 客户端处理响应

发送消息之后, 服务器的相应最终会通过信号槽, 调用到 `addSelfMessage`.

此处重点是在 `addSelfMessage` 中 `MessageShowArea::addMessage` 内部对图片消息的适配

在 `MessageContentLabel` 的 `mousePressEvent` 实现点击播放语音.

> 此处要考虑到文本提示的切换. 点击播放切换为 "播放中...", 播放完毕切换回 "[语音]"

```
void MessageContentLabel::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        if (messageType == SPEECH_TYPE) {
            if (content.isEmpty()) {
                Toast::showMessage("数据加载中, 请稍后");
                return;
            }
            // 如果是语音, 点击后触发播放
            SoundRecorder* soundRecorder = SoundRecorder::getInstance();
            connect(soundRecorder, &SoundRecorder::soundPlayDone, this, [=]() {
                if (label->text() == "播放中...") {
                    label->setText("[语音]");
                }
            });
            label->setText("播放中...");
            soundRecorder->startPlay(content);
        } else if (messageType == FILE_TYPE) {
            // 如果是文件, 点击后触发另存为
            // .......
        } else {
            // 文本消息啥都不做.
        }
    } else if (event->button() == Qt::RightButton) {
        // 弹出菜单
        // 通过 contextMenuEvent 实现.
    } else {
        LOG() << "不响应其他鼠标按键";
    }
}
```

4) 服务器实现逻辑

a) 在界面上添加按钮, "发送语音消息", 并实现槽函数

```
void Widget::on_pushButton_6_clicked()
{
    WebsocketServer* websocketServer = WebsocketServer::getInstance();
```

```
    emit websocketServer->sendSoundResp();
}
```

b) 定义 `WebsocketServer` 信号

```
void sendSoundResp();
```

c) 在 websocket 逻辑中, 添加发送文件逻辑

```
connect(this, &WebsocketServer::sendSoundResp, this, [=]() {
        if (socket == nullptr || !socket->isValid()) {
            LOG() << "socket 对象无效!";
            return;
        }
        bite_im::NotifyMessage notifyMessage;
        notifyMessage.setNotifyEventId("");

notifyMessage.setNotifyType(bite_im::NotifyTypeGadget::NotifyType::CHAT_MESSAGE_NOTIFY);

        bite_im::NotifyNewMessage newMessage;
        bite_im::MessageInfo messageInfo = makeSoundMessageInfo(this->messageIndex++, "2000", avatar);
        newMessage.setMessageInfo(messageInfo);

        notifyMessage.setNewMessageInfo(newMessage);
        QByteArray body = notifyMessage.serialize(&serializer);

        socket->sendBinaryMessage(body);

        LOG() << "发送语音消息响应";
    });
}
```

```
bite_im::MessageInfo makeSoundMessageInfo(int index, const QString& chatSessionId, const QByteArray& avatar) {
    bite_im::MessageInfo messageInfo;
    messageInfo.setChatSessionId(chatSessionId);
```

```
    messageInfo.setTimestamp(QDateTime::currentMSecsSinceEpoch() / 1000);
    messageInfo.setMessageId(QString::number(3000 + index));

    bite_im::UserInfo sender = makeUserInfo(index, avatar);
    messageInfo.setSender(sender);

    bite_im::MessageContent messageContent;

messageContent.setMessageType(bite_im::MessageTypeGadget::MessageType::SPEECH);
    bite_im::SpeechMessageInfo speechMessageInfo;
    speechMessageInfo.setFileId("testFileId");
    QByteArray content = loadFileToByteArray(":/file/sound.pcm");
    speechMessageInfo.setFileContents(content);
    messageContent.setSpeechMessage(speechMessageInfo);
    messageInfo.setMessage(messageContent);

    return messageInfo;
}
```

d) 在 websocket 断开连接时, 释放信号槽

```
disconnect(this, &WebsocketServer::sendSoundResp, this, nullptr);
```

语音识别文字

1) 客户端发送请求

a) 给 `MessageContentLabel` 添加右键菜单.

只针对语音消息才生效.

```
void MessageContentLabel::contextMenuEvent(QContextMenuEvent *event)
{
    if (messageType != model::SPEECH_TYPE) {
        LOG() << "非语音消息, 暂时不提供右键菜单";
        return;
    }
    QMenu* menu = new QMenu(this);
    QAction* action = menu->addAction("语音转文字");
    connect(action, &QAction::triggered, this, [=]() {
```

```
        LOG() << "语音转文字";
        speechRecognition();
    });
    menu->exec(event->globalPos());
    delete menu;
}
```

b) 实现 `speechRecognition`

```
void MessageContentLabel::speechRecognition()
{
    DataCenter* dataCenter = DataCenter::getInstance();
    connect(dataCenter, &DataCenter::speechRecognitionDone, this,
&MessageContentLabel::speechRecognitionDone, Qt::UniqueConnection);
    dataCenter->speechRecognitionAsync(fileId, content);
}
```

c) 实现 `DataCenter::speechRecognitionAsync`

```
void DataCenter::speechRecognitionAsync(const QString& fileId, const
QByteArray &content)
{
    netClient.speechRecognition(loginSessionId, fileId, content);
}
```

d) 实现 `NetClient::speechRecognition`

接口定义

```
message SpeechRecognitionReq {
    string request_id = 1;
    bytes speech_content = 2;
    optional string user_id = 3;
    optional string session_id = 4;
}

message SpeechRecognitionRsp {
```

```
    string request_id = 1;
    bool success = 2;
    string errmsg = 3;
    string recognition_result = 4;
}
```

函数实现

```
void NetClient::speechRecognition(const QString &loginSessionId, const QString& fileId, const QByteArray &content)
{
    // 1. 构造请求
    bite_im::SpeechRecognitionReq req;
    req.setRequestId(makeRequestId());
    req.setSessionId(loginSessionId);
    req.setSpeechContent(content);

    QByteArray body = req.serialize(&serializer);
    LOG() << "[语音识别] requestId=" << req.requestId() << " loginSessionId=" << loginSessionId;

    // 2. 发送 HTTP 请求
    QNetworkReply* httpResp = this->sendHttpRequest("/service/speech/recognition", body);

    // 3. 处理 HTTP 响应
    connect(httpResp, &QNetworkReply::finished, this, [=]() {
        // a) 解析响应
        auto resp = this->handleHttpResponse<bite_im::SpeechRecognitionRsp>(httpResp);
        if (!resp) {
            emit dataCenter->speechRecognitionDone(fileId, false, "语音识别服务调用失败", "");
            return;
        }

        // b) 获取到的结果数据
        //    此处不需要单独设置到 DataCenter 中

        // c) 发送信号
        emit dataCenter->speechRecognitionDone(fileId, true, "", resp->recognitionResult());
    });
}
```

2) 客户端处理响应

a) 定义 `DataCenter` 信号

```
// 语音识别完成
void speechRecognitionDone(const QString& fileId, bool ok, const QString reason, const QString text);
```

b) 处理 `speechRecognitionDone` 信号

实现 `MessageContentLabel::speechRecognitionDone`

```
void MessageContentLabel::speechRecognitionDone(const QString& fileId, bool ok, const QString reason, const QString text)
{
    if (fileId != this->fileId) {
        return;
    }
    if (!ok) {
        Toast::showMessage("语音转文字失败！ " + reason);
        return;
    }
    label->setText("[语音转文字] " + text);
    this->update();
}
```

3) 服务器实现逻辑

a) 注册路由

```
httpServer.route("/service/speech/recognition", [=](const QHttpServerRequest& req) {
    return this->recognition(req);
});
```

b) 实现处理函数

```
QHttpServerResponse HttpServer::recognition(const QHttpServerRequest &req)
{
    // 解析请求
    bite_im::SpeechRecognitionReq pbReq;
    pbReq.deserialize(&serializer, req.body());
    LOG() << "[REQ 语音转文字] request_id=" << pbReq.requestId();
    // 构造响应
    bite_im::SpeechRecognitionRsp pbRsp;
    pbRsp.setRequestId(pbReq.requestId());
    pbRsp.setSuccess(true);
    pbRsp.setErrmsg("");
    pbRsp.setRecognitionResult("这是一段录音这是一段录音");

    QByteArray body = pbRsp.serialize(&serializer);

    // 发送响应给客户端
    QHttpServerResponse httpResp(body, QHttpServerResponse::StatusCode::Ok);
    httpResp.setHeader("Content-Type", "application/x-protobuf");
    return httpResp;
}
```

历史消息界面逻辑 (2)

增加对图片, 文件, 语音的适配.

适配图片消息

a) 定义 `ImageButton` 类

此处要针对拿到的图片适当缩放, 使图片能正确显示.

```
class ImageButton : public QPushButton {
public:
    ImageButton(const QString& fileId, const QByteArray& content);

private:
    QString fileId;

    void updateUI(const QString& fileId, const QByteArray& content);
```

```
};
```

```
ImageButton::ImageButton(const QString &fileId, const QByteArray &content)
    : fileId(fileId)
{
    this->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);
    this->setStyleSheet("QPushButton { border:none; }");
    if (!content.isEmpty()) {
        // 2. 数据如果非空，则直接设置数据
        updateUI(fileId, content);
    } else {
        DataCenter* dataCenter = DataCenter::getInstance();
        connect(dataCenter, &DataCenter::getSingleFileDone, this,
&ImageButton::updateUI);
        dataCenter->getSingleFileAsync(fileId);
    }
}

void ImageButton::updateUI(const QString& fileId, const QByteArray& content)
{
    if (fileId != this->fileId) {
        return;
    }
    QImage image;
    image.loadFromData(content);
    QPixmap pixmap = QPixmap::fromImage(image);
    if (pixmap.width() >= 300) {
        // 适当进行缩放
        pixmap = pixmap.scaledToWidth(300);
    }
    this->resize(pixmap.size());
    this->setIconSize(pixmap.size());
    this->setIcon(QIcon(pixmap));
}
```

b) 修改 `HistoryMessageItem::makeHistoryMessageItem`

```
// 5. 创建消息体
QWidget* contentWidget = nullptr;
if (message.messageType == TEXT_TYPE) {
    // ......
} else if (message.messageType == IMAGE_TYPE) {
```

```
    contentWidget = new ImageButton(message.fileId, message.content);
} else if (message.messageType == FILE_TYPE) {
    // TODO
} else if (message.messageType == SPEECH_TYPE) {
    // TODO
} else {
    LOG() << "错误的 messageType = " << message.messageType;
}
```

适配文件消息

a) 创建 `FileLabel` 类

此处要实现点击另存为的功能.

```
class FileLabel : public QLabel {
public:
    FileLabel(const QString& fileId, const QByteArray& content, const QString& fileName);
    void mousePressEvent(QMouseEvent* event) override;

private:
    QString fileId;
    QByteArray content;
    QString fileName;
    bool loadDone = false;
};
```

```
FileLabel::FileLabel(const QString& fileId, const QByteArray &content, const QString& fileName)
    : fileId(fileId), content(content), fileName(fileName)
{
    this->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding); // 设置大小策略
    this->setWordWrap(true); // 启用文本换行
    this->adjustSize(); // 调整大小以适应内容
    this->setAlignment(Qt::AlignTop | Qt::AlignLeft);
    this->setText("[文件] " + fileName);

    if (!content.isEmpty()) {
        return;
    }
```

```
    DataCenter* dataCenter = DataCenter::getInstance();
    connect(dataCenter, &DataCenter::getSingleFileDone, this, [=](const QString& fileId, const QByteArray& content) {
        if (fileId != this->fileId) {
            return;
        }
        this->content = content;
        this->loadDone = true;
    });
    dataCenter->getSingleFileAsync(fileId);
}

void FileLabel::mousePressEvent(QMouseEvent *event)
{
    // 1. 判定是否加载完毕
    if (!loadDone) {
        Toast::showMessage("文件内容加载中，请稍后");
        return;
    }
    // 2. 弹出对话框，选择位置
    QString filePath = QFileDialog::getSaveFileName(this, "另存为", QDir::homePath(), "*");
    if (filePath.isEmpty()) {
        LOG() << "取消选择保存文件的路径";
        return;
    }
    // 3. 保存文件
    writeByteArrayToFile(filePath, content);
}
```

b) 修改 `HistoryMessageItem::makeHistoryMessageItem`

```
// 5. 创建消息体
QWidget* contentWidget = nullptr;
if (message.messageType == TEXT_TYPE) {
    // ......
} else if (message.messageType == IMAGE_TYPE) {
    // ......
} else if (message.messageType == FILE_TYPE) {
    contentWidget = new FileLabel(message.fileId, message.content, message.fileName);
} else if (message.messageType == SPEECH_TYPE) {
    // TODO
} else {
```

```
    LOG() << "错误的 messageType = " << message.messageType;
}
```

适配语音消息

a) 创建 `SoundLabel` 类

```
class SoundLabel : public QLabel {
public:
    SoundLabel(const QString& fileId, const QByteArray& content);
    void mousePressEvent(QMouseEvent* event) override;

private:
    QString fileId;
    QByteArray content;
    bool loadDone = false;
};
```

```
SoundLabel::SoundLabel(const QString& fileId, const QByteArray &content) : fileId(fileId), content(content)
{
    this->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding); // 设置大小策略
    this->setWordWrap(true); // 启用文本换行
    this->adjustSize(); // 调整大小以适应内容
    this->setAlignment(Qt::AlignTop | Qt::AlignLeft);
    this->setText("[语音]");

    if (!content.isEmpty()) {
        return;
    }
    DataCenter* dataCenter = DataCenter::getInstance();
    connect(dataCenter, &DataCenter::getSingleFileDone, this, [=](const QString& fileId, const QByteArray& content) {
        if (fileId != this->fileId) {
            return;
        }
        this->content = content;
        this->loadDone = true;
    });
    dataCenter->getSingleFileAsync(fileId);
```

```
}

void SoundLabel::mousePressEvent(QMouseEvent *event)
{
    if (!loadDone) {
        Toast::showMessage("文件内容加载中, 请稍后");
        return;
    }
    SoundRecorder* soundRecorder = SoundRecorder::getInstance();
    soundRecorder->startPlay(content);
}
```

b) 修改 `HistoryMessageItem::makeHistoryMessageItem`

```
// 5. 创建消息体
QWidget* contentWidget = nullptr;
if (message.messageType == TEXT_TYPE) {
    // ......
} else if (message.messageType == IMAGE_TYPE) {
    // ......
} else if (message.messageType == FILE_TYPE) {
    // ......
} else if (message.messageType == SPEECH_TYPE) {
    contentWidget = new SoundLabel(message.fileId, message.content);
} else {
    LOG() << "错误的 messageType = " << message.messageType;
}
```

重定向日志到文件中

```
FILE *output = NULL;

void msgHandler(QtMsgType type, const QMessageLogContext &context, const
QString &msg)
{
    QByteArray localMsg = msg.toUtf8();
    fprintf(output, "[%s:%d] %s\n", context.file, context.line,
localMsg.constData());
}
```

```
8
9  int main(int argc, char **argv)
10 {
11     output = fopen("log.txt", "a"); //重定向于文件
12     qInstallMessageHandler(msgHandler);
13
14     // ......
15 }
```

发布程序

1) 按照 release 的方式构建程序

2) 找到 exe 所在目录, 并单独拷贝出来.

3) 使用 windeployqt 获取到依赖的 dll.

```
1 C:\Qt\6.7.1\msvc2019_64\bin\windeployqt.exe .\ChatClient.exe
```

> `windeployqt` 是 Qt SDK 自带的工具.
>
> 注意 `windeployqt` 所在的路径, 根据个人机器的实际路径来设置.

扩展功能

群聊操作

- 添加成员
- 退出群聊
- 群管理员
- ……

持久登录

- 登录一次之后, 关闭程序, 下次可以不必重新登录.

> 需要服务器提供一个验证 loginSessionId 是否有效的 api

退出登录

- 主界面上增加退出登录按钮, 点击后主动退出登录状态.

创建系统托盘图标

- 创建托盘图标
- 右键菜单
- 左键显示窗口
- 收到消息图标闪烁

播放声音

- 收到消息播放声音

切换主题

- 内置多种主题, 可以进行切换

系统设置

- 设置窗口大小, 声音大小, 字体大小等.