11. 客户端接收识别人脸和优化

基本步骤：

1. 使用封装好的 `qfaceobject` 模块中的查询 `face_query` 。
2. 接收端模块 `attendancewin` 添加头文件 `#include <qfaceobject.h>`
3. 对接收到数据进行解码，解码函数 `imdecde` 在7小节介绍
4. 接收端 `read_data` 函数中添加识别显示人脸部分

```
//识别人脸
    cv::Mat faceimage;//人脸图片
    std::vector<uchar> decode;//存储解码后数据
    decode.resize(data.size());
    memcpy(decode.data(),data.data(),data.size());// 将数据从data中复制到
decode
    faceimage = cv::imdecode(decode,cv::IMREAD_COLOR);// 解码，IMREAD_COLOR代
表解码成3通道BGR图像
    int faceid = fobj.face_query(faceimage);// 查询faceid
    qDebug()<<faceid;
```

5. 查询到faceid后从数据库中寻找个人信息
6. 导入数据库模块，新建 `QsqlTableModel` 类型，并绑定 `employee` 数据库，在数据库中查询到员工信息后发送到客户端显示。

```
/*在.h文件中*/
QSqlTableModel model;
/*在AttendanceWin构造函数中*/
model.setTable("employee");
/*在read_data()函数中*/
// 从数据库中查询faceid对应的人脸数据
    // 给模型设置过滤器
    model.setFilter(QString("faceID=%1").arg(faceid));
    // 查询
    model.select();
    // 判断是否查询到数据
    if(model.rowCount()==1){
        // 工号，姓名，部门，时间
        // {"employeeID":%1,"name":%2,"department":软件,"time":%3}
        QSqlRecord record = model.record(0);
        QString sdmsg =QString("
```

```
{\"employeeID\":\"%1\",\"name\":\"%2\",\"department\":\"软件
\",\"time\":\"%3\"}").
arg(record.value("employeeID").toString()).arg(record.value("name").toString
())
.arg(QDateTime::currentDateTime().toString("yyyy-MM-dd hh:mm:ss"));
        // 打包发送给客户端
        msocket->write(sdmsg.toUtf8());

        // 把数据写入数据库

    }
```

7. 客户端进行接收。

```
/*在faceattendance客户端的构造函数中*/
// 关联接收数据的槽函数
    connect(&msocket, &QTcpSocket::readyRead, this,
&FaceAttendance::recv_data);

void FaceAttendance::recv_data(){
    QString msg = msocket.readAll();
    qDebug()<<msg;
}
```

优化

优化发送次数

- 思路：设置一个flag来记录检测到人脸的次数，当检测到一定值后在发送图像；
- 规定：
    1. flag的初值为 0，表示初始状态下没有检测到人脸。
    2. 如果连续检测到人脸，flag_onepersion 会逐步增加，表示连续检测到人脸的次数。
    3. 当flag达到一定值（大于 2）时，才会执行发送人脸图像的操作。
    4. 一旦发送了人脸图像后，会将flag重置为 -2，以便下一次检测到人脸时再次触发发送操作，相当于检测到的是同一个人的时候后就不在发送，只有他移动了脱离了检测区域才会进行下一次检测。
    5. 如果在一帧图像中没有检测到人脸，则会将flag重置为 0，表示重新开始检测人脸。

```
/*在构造函数中赋初始值*/
flag_faceSend = 0;
/*在定时器事件中timerEvent(QTimerEvent *e)*/
std::vector<Rect> faceRects;
```

```
    cascade.detectMultiScale(srcImage, faceRects);
    if(faceRects.size()>0 && flag_faceSend ≥ 0)//检测到人脸
    {
        Rect rect = faceRects.at(0);// 第一个人脸的矩形框
        // 移动人脸框（图片--QLabel）
        ui->headpic->move(rect.x,rect.y);
        if(flag_faceSend > 2){
            // Mat 转换为能够发送的数据QByteArray
            // 编码成jpg格式
            std::vector<uchar> buf;
            cv::imencode(".jpg", srcImage, buf);
            // 新建传输的数据格式
            QByteArray byte((const char*)buf.data(),buf.size());
            // 准备发送
            // 获取数据大小
            quint64 backsize = byte.size();
            // 创建发送对象
            QByteArray sendData;
            // 将用户定义的一些变量保存到文件的模块
            QDataStream stream(&sendData, QIODevice::WriteOnly);
            // 设置QDataStream版本
            stream.setVersion(QDataStream::Qt_5_14);
            // 将数据大小和字节写入sendData
            stream << backsize << byte;
            // 发送
            msocket.write(sendData);
            flag_faceSend = -2;
        }
        flag_faceSend++;
    }
    if(faceRects.size() == 0){//当检测不到人脸时，回到初始位。
        //把人脸框移动到中心位置
        ui->headpic->move(100,60);
        flag_faceSend = 0;
    }
```

优化人脸查询

- 用线程实现查询

```
/*头文件中创建信号*/
signals:
    void query(cv::Mat& image);
```

```
/*在接收端attendancewin构造函数中*/
// 创建一个线程
    QThread *thread = new QThread();
    // 把QFaceObject对象移动到thread线程中
    fobj.moveToThread(thread);
    // 启动线程,线程不能直接调用，需要信号触发
    thread->start();
    connect(this,&AttendanceWin::query, &fobj, &QFaceObject::face_query);
    // 关联QFaceObject对象里面的send_faceid信号
    connect(&fobj, &QFaceObject::send_faceid, this,
&AttendanceWin::recv_faceid);
```

此时，在接受人脸后不用直接调用`face_query()`函数，而是发送一个信号，让进程去处理查询的功能。

```
//识别人脸
    cv::Mat faceimage;//人脸图片
    std::vector<uchar> decode;//存储解码后数据
    decode.resize(data.size());
    memcpy(decode.data(),data.data(),data.size());// 将数据从data中复制到
decode
    faceimage = cv::imdecode(decode,cv::IMREAD_COLOR);
    //int faceid = fobj.face_query(faceimage);// 查询faceid
    emit query(faceimage);
```

但是，这会导致无法返回faceid，所以需要在进行一次信号返回faceid。

```
/*connect(&fobj, &QFaceObject::send_faceid, this,
&AttendanceWin::recv_faceid);*/
/*face_query()应该为信号发送*/
int QFaceObject::face_query(cv::Mat &faceimage)
{
    // 把Mat数据转为seetaface的数据
    SeetaImageData simage;
    simage.data = faceimage.data;
    simage.width = faceimage.cols;
    simage.height = faceimage.rows;
    simage.channels = faceimage.channels();
    float similarity = 0;
    int64_t faceid = this->fengineptr->Query(simage, &similarity);
    if(similarity>0.5){
        emit send_faceid(faceid);
    }
```

```
    else{
        emit send_faceid(-1);
    }
    return faceid;
}
/**/
void AttendanceWin::recv_faceid(int64_t faceid){
    if(faceid<0){
        QString sdmsg = QString("{\"employeeID\":,\"name\": ,\"department\":
,\"time\": }");
        msocket->write(sdmsg.toUtf8());
        return;
    }
    // 从数据库中查询faceid对应的人脸数据
    // 给模型设置过滤器
    model.setFilter(QString("faceID=%1").arg(faceid));
    // 查询
    model.select();
    // 判断是否查询到数据
    if(model.rowCount()==1){
        // 工号，姓名，部门，时间
        // {"employeeID":%1,"name":%2,"department":软件,"time":%3}
        QSqlRecord record = model.record(0);
        QString sdmsg =QString("
{\"employeeID\":\"%1\",\"name\":\"%2\",\"department\":\"软件
\",\"time\":\"%3\"}")

.arg(record.value("employeeID").toString()).arg(record.value("name").toStrin
g())
                .arg(QDateTime::currentDateTime().toString("yyyy-MM-dd
hh:mm:ss"));
        // 打包发送给客户端
        msocket->write(sdmsg.toUtf8());
    }
}
```

Qt默认发送的信号类型并不支持发送 `Mat` 和 `int64_t` 类型，需要在 `main` 函数中注册。

```
    // 自定义类型需要注册才能传递
    qRegisterMetaType<cv::Mat>("cv::Mat&");
```

```
    qRegisterMetaType<cv::Mat>("cv::Mat");
    qRegisterMetaType<int64_t>("int64_t");
```