# 项目相关

## 1. 基于Android和Java的流式二维码数据传输系统

### 1.1. 简介

流式二维码数据传输系统，以动态二维码的形式在手机／电脑之间传送文件。发送端可以是手机、电脑等任意有屏幕的设备，接收端是带有摄像头的手机。发送端将文件数据编码后嵌入到多个二维码中，二维码动态显式于电脑／手机屏幕上。接收端APP通过摄像头连续捕捉发送的二维码，识别并解码还原出数据，完成文件传输过程。

### 1.2. 应用和优点

* 目前手机之间需要传送小文件（如照片）需要蓝牙或wifi进行配对连接，然后传输文件，这样有两个弊端：需要设备硬件支持，需要提前配对连接。而这种基于二维码的传输不需要硬件支持（手机几乎全都有摄像头），运行于软件层面上；且不需要提前建立连接，即用即传。
* 目前将文件从台式电脑传送到手机需要通过USB有线连接，或手机加入与台式机在同一局域网的wifi网络，这样很不方便。而通过基于二维码的传输只需要将二维码在电脑屏幕上显示，手机打开摄像头拍摄即可完成文件传输。
* 一些如企业等有保密需求的环境下，手机不能连接wifi加入同一局域网，通过基于二维码的传输则可方便解决。

### 1.3. 传输效率

在发送方最高为30fps的环境下，我们设计的黑白二维码可以达到200kbps，彩色（4色）二维码可以达到300kbps，应用机器学习后可以达到380kbps。

1080P每帧的处理时间在20ms左右。边界检测6ms，小方块采样8ms，识别2ms，纠错2ms。

### 1.4. 研究贡献

两个阶段

#### 1.4.1. 第一阶段

* 新的二维码布局：不同于一个小方块一种颜色，ShiftCode可以在小方块内部嵌套了一个更小的方块，以小方块的移动方向来嵌入数据。优点：嵌入数据更多，可扩展，为解决帧重叠问题提供可能。
* 解决帧重叠问题：对于30fps的接收方，当发送方帧速率超过15fps时就会出现帧重叠的问题。采用前后两帧相同位置小方块前景色和后景色相反的设计，可以在小方块发生帧重叠时正确区分前后帧小方块的移动方向，从而将重叠帧分离开。
* 可靠性解决方案：对于每一帧的数据，使用Reed-Solomon纠错编码，其优点是可以对连续出错有较好的纠错能力。对所有发送的帧数据使用RaptorQ冗余编码，因为这种传输是单向传输，并没有丢帧重传机制，RaptorQ可以将原始的N帧数据计算生成(N + K)帧数据，只要接收到任意的N帧即可还原出原数据。

#### 1.4.2. 第二阶段

* 基于机器学习的通用动态二维码识别方案：对于任何现有的基于规则的动态二维码方案，都可以将其识别步骤替换为机器学习方案，识别效率和准确度上都有很大提升。
* 统一的识别框架：只需要将小方块采样点颜色和其对应的真实数据交给机器学习训练，得到机器学习模型，后续真正传输时只需要采样小方块，交给模型即可得到真实数据。
* 解决帧重叠问题：以黑白为例，重叠帧中小方块可能有4个状态：【黑+黑】、【黑+白】、【白+黑】和【白+白】，通过人为制造【黑+黑】、【黑+白】、【白+黑】和【白+白】这4个状态，接下就只需要确定小方块属于其中哪个状态。这实际是一个分类问题，只需要提前训练好状态的分类，即可判断出小方块的重叠情况。更细节的需要涉及到rolling shutter现象。

### 1.5. 开发APP遇到的问题

#### 1.5.1. 不能实时解码

问题：接收端为30fps，即一秒钟要处理30张图片，每一张图片总的（边界检测、采样、识别、纠错）处理时间要在33ms内。前期没有专门优化导致每帧处理时间需要1秒以上。

解决方法：

* 使用Android Device Monitor的TraceView工具分析每一帧的处理耗时分布，找到耗时最长的函数进行优化，如此循环。
* 因Android提供的帧数据是NV21格式，最初是将图片由NV21转换为RGB，非常耗时。最初尝试将转换函数的浮点运算转换为整数运算，效果不好；之后尝试使用OpenGL ES的shader，即GPU来进行转换运算，有一定效果但耗时仍长；再尝试使用JNI即C++来运算，仍未达到预期。再后来转变思路，因为不会用到图片中所有像素点，所以可以在需要时再当场对单个像素点转换，这样暂时解决了问题。但后来发现因为采样点还是比较多，总体来看转换函数耗时还是很大，最后项目整体由RGB转到YUV，不再需要颜色转换，问题顺利解决。
* 二维码边界检测是使用一个小矩形逐渐放大直到框住二维码，这个过程需要逐像素行检查和扩展，很耗时。考虑到传输过程中摄像头不会大幅度移动，即二维码相对图片中的位置较为稳定，那么框住的矩形也会较为稳定，这样在确定一个矩形后，对于下一帧，只需要将矩形缩小1/3或1/5即可，节约了很多计算。
* 边界检测的结果是定位到二维码的四个顶点，最初是根据这四个顶点通过反透视变换生成一个标准的二维码，需要将原二维码中每个像素计算映射到标准二维码中，很耗时。后来考虑到反透视变换实际上只是计算出一个变换矩阵，而对于拍到二维码，只需要获得采样点的像素值即可，不需要将整个二维码中全部像素点都映射；所以只在需要某个采样点像素值时，将其标准化坐标通过透视变换矩阵转换为真实坐标，再获得像素值即可。
* 其他主要是Java语法层面优化，比如变量复用，减少new之类的。

#### 1.5.2. 代码扩展性差

问题：如有新的处理方法或新的二维码加入，需要修改代码时就有种牵一发而动全身的感觉，代码扩展性差。

解决方法：

* 恶补面向对象编程思想，设计模式。
* 充分利用继承和抽象复用代码，如不同形状二维码，不同颜色二维码。
* 使用工厂方法管理不同二维码的实例化，复用同一套处理框架。
* 使用策略模式根据不同需求调整使用不同的二维码定位算法。
* 将各种不同二维码的构造生成方法抽象为统一的平台，将二维码分割成border、padding、数据区等块，对于每一块独立指定二维码形状和数据等即可生成二维码。这样避免了为每种二维码写一个大同小异的生成方法，也避免了需要计算各种偏移量等。

### 1.6. 用到了哪些机器学习方法

* 实际是一个分类问题，features是小方块采样点颜色和对应参考颜色，class是小方块的真实数据。
* 最基本的是使用决策树（Decision Tree），分类效果较好。
* 为解决决策树学习时间长的问题，使用随机森林（Random Forest），学习速度有明显提高。
* 为解决每次传输需要当场学习出一个模型的问题，使用混合专家模型（Mixture of Experts），对不同场景进行几次传输得到多个模型，使用混合专家模型时只需要在最开始放上少量的学习帧，即可得到各模型的权重，形成一个新的模型。

### 1.7. MediaCodec应用在哪里？

* MediaCodec用来快速将视频文件解码为图片帧。
* 项目支持的输入包括摄像头预览、图片和视频。
* 视频的解码最初使用MediaMetadataRetriever，这样有两个主要缺点：软件解码，耗时长；只能使用getFrameAtTime()获取帧，不精确。
* MediaCodec是Android提供的硬件解码API，可以快速将视频文件解码为图片帧。
* 但MediaCodec存在一个问题：其仅支持处理器芯片支持的帧格式（YUV420中的众多格式），且没有通用支持的格式，即没有设备通用性。
* 利用Android API 21的新特性：COLOR\_FormatYUV420Flexible的支持，和新的Image类。
* 在解码时指定解码格式为COLOR\_FormatYUV420Flexible（这个所有设备都支持），解码生成的帧指定为Image类，其可以单独获取YUV各个分量的数据。拼接这些数据即可得到指定YUV420格式的图片帧。实现了设备通用性。
* 对1080P的视频，每个图片帧可以在30ms内得到。

### 1.8. 代码结构

#### 1.8.1. 发送方

* 二维码由Districts、District、Zone组成，其将二维码划分为不同区域（Zone）。每个区域指定其Block的类型，以及二进制数据。由底层处理数据向二维码图片的转换，包括位置、偏移、大小等。
* 不同的二维码可能有类似的边框样式或数据内容，通过继承关系组织不同的二维码，如BlackWhiteCode是最基础的二维码，其上可以衍生出ColorCode、ShiftCode等。

#### 1.8.2. 接收方

* 使用两个线程一个队列，一个线程用来从摄像头／视频文件获取图片帧，并放入队列；另一个线程从队列中取出图片帧，处理图片帧，直到再无图片帧或数据传输完成。
* 将图片帧数据封装为RawImage，提供像素值获取、二维码定位等功能。
* RawImage由MediatBarcode封装，MediatBarcode同时提供反透视变换功能，将二维码分割为Districts、District、Zone区域，提供获取区域像素值数据等功能。
* MediateBarcode供不同的二维码使用，这些二维码同样有继承关系，最基本的BlackWhiteCode提供有通用的Zone处理方法，如获取帧编号等。
* StreamDecode提供处理队列中二维码的骨架，其管理二维码间的关联关系，如二维码定位矩阵的保存、帧间冗余算法的纪录。

## 2. 基于Android、Flask和MySQL的用户情绪和隐私收集系统

### 2.1. 简介

APP主要包括两部分，一部分是填写调查问卷，另一部分是在后台持续收集用户隐私信息。调查问卷由用户向服务器请求得到，填写完成后提交给服务器；收集隐私是APP长驻后台，定时抓取用户数据并保存到本地数据库，定时将数据库发送给服务器。

### 2.2. 收集哪些隐私信息

收集的隐私信息包括录音（持续5秒）、通话记录、短信、联系人信息、地理位置（高德地图API）、手机信息、传感器信息（磁力计、陀螺仪等）、周围wifi、屏幕亮灭、前台运行APP、后台运行APP等。

### 2.3. 如何长驻后台

* 服务启动时即拿到PARTIAL\_WAKE\_LOCK，且不放弃。PARTIAL\_WAKE\_LOCK在屏幕熄灭的情况下仍保持CPU运行。
* 使用第三方云巴推送平台，定时向所有用户推送消息，用来唤醒APP。
* 用户配合关闭手机厂商的省电设置等。

### 2.4. 如何实现用户登录注册

因为是科研项目，并没有实现完整的登录注册功能。只需要填写学号、手机号，服务器后台查询不冲突即注册成功；只需要填写学号，服务器能够查询到相关信息即可登录。

### 2.5. 问卷内容是什么？

问卷内容我不关系。主要是询问用户是否开心、沮丧等，及其程度。

### 2.6. 调查问卷如何分发和收集

* 用户收到APP推送或主动打开APP，点击“填写调查问卷”即向服务器发送请求，服务器判断距离用户上次提交问卷时间间隔有没有达到6小时，若未达到则返回错误提示；若达到则进入分发问卷流程。
* 分发问卷时，服务器从数据库中读取题目，并拼装为json格式，同时针对每次分发生成一个唯一的id，用来记录问卷的提交。
* 用户填写完问卷点击提交时，同步提交问卷的唯一id；服务器首先验证唯一id是否有效，然后读取客户端发送的json格式回答，解析后保存到数据库中。

### 2.7. APP如何定时抓取用户信息

APP启动同步创建一个Service，Service启动时即实例化一个SingleThreadScheduledExecutor，并设定定时执行周期为一分钟。

本地数据库中新建一个表，每行纪录需要运行的任务（抓取哪些信息、心跳、上传数据库、清理数据库、切换新数据库）、执行间隔时间、下次运行时间。每当定时周期达到时，就查询数据库，一旦某个任务的下次运行时间小于当前时间，则执行任务，并根据执行间隔时间确定下次运行时间。

### 2.8. 如何上传用户数据

当执行上传用户数据任务时，则从数据库中读取还未上传的数据库的名字，并逐个通过HTTP POST请求发送本地压缩后的数据库，并在本地数据库中标记此数据库已上传。

### 2.9. APP崩溃如何分析恢复

使用第三方库ACRA进行崩溃信息的收集和上报。

当APP崩溃后，下次运行时对于不完整或异常的任务在撤销后重新执行并写数据库，保证数据库数据的一致性。

### 2.10. 后台提供哪些统计和管理

* 后台展示每个用户的心跳历史、问卷填写历史、上传用户数据时间等。
* 可以从后台向特定用户发送推送通知，可以针对不同用户指定不同的APP版本。

### 2.11. 用到哪些第三方库

Android上的HTTP通信部分使用了okhttp。

### 2.12. 后台是如何搭建的

后台是基于Python和flask的，flask提供了一套http的处理框架，只需要注册对应的url即可。对于数据的持久存储等则是利用到了MySQL。

### 2.13. 遇到过哪些问题

* APP经常退出前台即被杀死。最初考虑一般的保活方法，如使用AlarmManager，发现用处不大。后来才发现是国内厂商动了手脚，非白名单的APP一律不允许后台运行。所以后来除在技术上尽可能保证APP不被杀死外，还建议用户在手机的设置中将APP加入到白名单。
* 经常在抓取用户数据时crash。最初以为是本身代码有问题，一番排查后才发现国内厂商在权限管理上动了手脚。按照官方文档，如果执行某些代码时用户未授权则会返回null值或抛出异常，但国内厂商直接在framework层进行了修改，不知道在哪会去检查权限，一旦未授权则直接抛出异常，导致APP crash。最后不得已对一些敏感信息收集时使用大段的try catch，避免未知的crash。