开发指南

清单：

1.linux

（1）librxtxSerial.so linux 相关的串口操作 so 文件（64 位 linux），

（2）RXTXcomm.jar 与 so 文件对应的 jar 包 RXTXcomm.jar，

（3）RFID.jar 与模块交互的上层接口

2.Windows

（4）librxtxSerial.dll windows 相关的串口操作 dll 文件（64 位 windows），

（5）RXTXcomm.jar 与 so 文件对应的 jar 包 RXTXcomm.jar，

（6）RFID.jar 与模块交互的上层接口

3.RFID-api RFID.jar api 文档

4.Test.java 简单的示例代码

5.开发指南 SDK 的开发指南

1. RFID java 开发

1. 首先要连接读写器以实现与读写器的通信，打开连接之后台会启动一个线程启动监听读写器返回数据，并且如果连接成功会返回一个 RFIDReaderHelper 对象他是与读写器交互的核心类。

示例代码：

```
ModuleConnector connector = new ReaderConnector();//构建连接器

connector.connectCom("COM7",115200);//连接指定串口，返回 RFIDReaderHelper 对象表示成功，返回空表示失败 false 失败
```

3. RFIDReaderHelper 对象可以发送指令到读写器还可以通过注册观察者 RXObserver 对象监听读写器返回数据

示例代码：

```
mReaderHelper.realTimeInventory((byte) 0xFF,(byte)0x01);//发送实时盘存指令，更多指令参考 API 文档
```

4.获取 RFID 模块的数据返回，继承 RXObserver 类覆盖相应的的方法，通过 RFIDReaderHelper 的 registerObserver 方法注册到 RFIDReaderHelper 中，后台线程在读取到 RFID 模块返回的相应数据的时候会回调对应的方法，作为参数传递出来。因此 RXObserver 中的各种回调方法运行在子线程中。你没必有覆盖所有的方法，只需覆盖你用到的方法即可。（以下提到的发送指令的函数，均为 RFIDReaderHelper 中的函数）

示例代码

```
RXObserver rxObserver = new RXObserver() {

    @Override

    protected void onExeCMDStatus(byte cmd, byte status) {

        //如果指令没有返回额外数据仅包含命令执行的状态码（例如 RFIDReaderHelper 中的各种以 set 开头的设置指令函数，）会回调该方法

        //如果指令返回数据异常一定会回调该方法 status 为异常码

        //cmd 可以用来区分具体是哪条命令的返回，命令参考 CMD 类文档，status 指令执行状态码，参考 ERROR 类文档

    }



    @Override

    protected void refreshSetting(ReaderSetting readerSetting) {

       //当发送查询读写器设置指令（例如 RFIDReaderHelper 中的各种以 get 开头的查询指令函数）会回调该方法，若有返回值会存储在 readerSetting 相应字段中

       //具体可以参考 API 文档中 ReaderSetting 各个字段的含义

    }
```

```
    @Override

    protected void onInventoryTag(RXInventoryTag tag) {

       //当发送盘存指令的时候该方法将会回调，盘存指令包括 RFIDReaderHelper 中 inventory，
realTimeInventory，customizedSessionTargetInventory,fastSwitchAntInventory 等函数以及
扣扳机

       //inventory 函数盘存到的标签会先缓存到 RFID 模块的缓存中，只有调用 getInventoryBuffer
或 getAndResetInventoryBuffer 函数是才会回调该方法将数据上传，上传的标签数据无重复

       //当盘存到多张标签的时，该方法会多次回调，标签可以重复

    }



    @Override

    protected void onInventoryTagEnd(RXInventoryTag.RXInventoryTagEnd tagEnd) {

      //当一条盘存指令执行结束的时候该方法会回调（fastSwitchAntInventory 除外
fastSwitchAntInventory 结束时回调 onFastSwitchAntInventoryTagEnd），tagEnd 为指令结束时的
返回数据，具体各个字段的含义

      //可以参考文档中 RXInventoryTag.RXInventoryTagEnd 各个字段的含义

    }

    @Override

    protected void
onFastSwitchAntInventoryTagEnd(RXInventoryTag.RXFastSwitchAntInventoryTagEnd tagEnd)
{

    //因为 fastSwitchAntInventory 函数返回的结束数据特殊，因此其单独回调这个函数

    // RXInventoryTag.RXFastSwitchAntInventoryTagEnd 中各字段的含义参考 API 文档

    }



    @Override

    protected void onGetInventoryBufferTagCount(int nTagCount) {

       //通过函数 getInventoryBufferTagCount 得到缓存中盘存标签的数量，数据是通过 inventory
盘存到读写器缓存区中标签数量，无重复标签的数量
```

```
    }

    @Override
    protected void onOperationTag(RXOperationTag tag) {
       //当执行 readTag,writeTag,lockTag 或者 killTag 等操作标签指令函数时会回调该方法，当一次操作多张标签时会多次回调
       //返回数据 RXOperationTag tag 参考 API 文档
    }

    @Override
    protected void onOperationTagEnd(int operationTagCount) {
      //当执行 readTag,writeTag,lockTag 或者 killTag 等操作标签指令函数结束时会回调该方法
      //operationTagCount 为操作的标签数量
    }
    @Override
    protected void onInventory6BTag(byte nAntID, String strUID) {
      //当执行 iso180006BInventory 时会回调该方法，如果盘存到多张标签会回调多次
      //nAntID 盘存的标签的天线号，strUID 盘存到 6B 标签的 UID
    }

    @Override
    protected void onInventory6BTagEnd(int nTagCount) {
       //当 iso180006BInventory 函数执行结束，所有盘存到的 6B 标签数据上传完毕，会回调该方法，并传回盘存的 6B 标签数量
       //nTagCount 为盘存到 6B 标签的数量
    }
```

```
@Override
protected void onRead6BTag(byte antID, String strData) {
   //当执行 iso180006BReadTag 函数时该方法会回调
   //
}

@Override
protected void onWrite6BTag(byte nAntID, byte nWriteLen) {
   //当执行 iso180006BWriteTag 函数时该方法会回调
}

@Override
protected void onLock6BTag(byte nAntID, byte nStatus) {
   //当执行 iso180006BLockTag 函数时该方法会回调
   //nAntID 天线号 nStatus 标签 Lock 状态
}

@Override
protected void onLockQuery6BTag(byte nAntID, byte nStatus) {
   //当执行 iso180006BQueryLockTag 函数时该方法会回调
   //nAntID 天线号 nStatus 标签 Lock 状态
}
@Override
protected void onConfigTagMask(MessageTran msgTran) {
```

```
    //当执行 setTagMask，getTagMask，clearTagMask 函数时改方法会回调

    //返回数据 msgTran 具体数据参考 MessageTran API 说明与 Select 指令格式

    }

};



//注册 RXObserver 对象到 RFIDReaderHelper，只有这样一旦 RFID 模块有数据返回才会回调
RXObserver 中的相应方法。

mReader.registerObserver(rxObserver);
```

5. 释放资源

退出应用的时候一定要释放相应的资源 示例代码：

```
  //移除所有的 RXObserver 监听

  mReader.unRegisterObserver(rxObserver);

  //停止相应的线程，关闭相应 I/O 资源,连接断开无法与模块交互只有重新连接再次获取
RFIDReaderHelper 才能与模块交互

   mConnector.disConnect();

   //释放读写器上电掉电控制设备
```

6. 高级

(1). 监听发送和接收数据，以及模块的链接状态。实现 RXTXListener 接口将其设置到 RFIDReaderHelper 类中

示例代码： //实现 RXTXListener 接口 RXTXListener mListener = new RXTXListener() {

```
    @Override

    public void reciveData(byte[] btAryReceiveData) {

        // TODO Auto-generated method stub

        //获取从 RFID 模块接收到的数据

    }
```

```
    @Override
    public void sendData(byte[] btArySendData) {
        // TODO Auto-generated method stub
        //获取发送到 RFID 模块的数据
    }

    @Override
    public void onLostConnect() {
        // TODO Auto-generated method stub
        //链接断开会回调该方法。
    }

};
//将 RXTXListener 注册到 RFIDReaderHelper 以便监听相关的数据
mReader.setRXTXListener(mListener);
```

（2).自定义相关的实现，如果你想自定义类去实现与 RFID 模块的交互，可以继承或实现 com.module.interaction 包中的类或接口定义自己的实现，具体可以参考文档以及我们的实现。