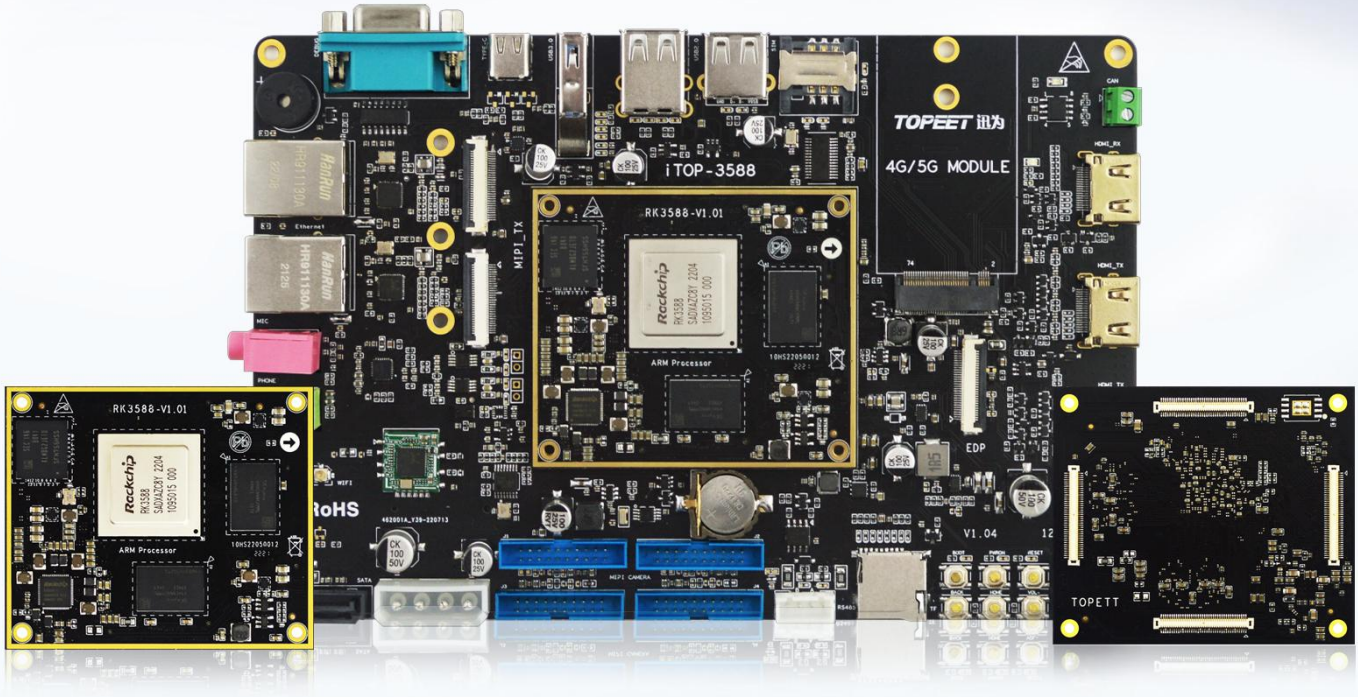
TOPEET 北京迅为

强大的 AI 能力 更快更强

超长供货周期 | 7X24 小时稳定运行 | 8K 视频编解码



iTOP-RK3588 开发板使用手册

八核 64 位 CPU | 主频 2.4GHz | NPU 算力 6T | 4800 安防级别 ISP

更新记录

更新版本	修改内容
V1.0	初版

目录

版权声明

更多帮助

注意事项与维护

- ❖ 请注意和遵循标注在产品上的所有警示和指引信息；
- ❖ 请勿带电插拔核心板及外围模块；
- ❖ 使用产品之前，请仔细阅读本手册，并妥善保管，以备将来参考；
- ❖ 请使用配套电源适配器，以保证电压、电流的稳定；
- ❖ 请勿在冷热交替环境中使用本产品，避免结露损坏元器件；
- ❖ 请保持产品干燥，如果不慎被任何液体泼溅或浸润，请立刻断电并充分晾干；
- ❖ 请勿使用有机溶剂或腐蚀性液体清洗本产品；
- ❖ 请勿在多尘、脏乱的环境中使用本产品，如果长期不使用，请包装好本产品；
- ❖ 如果在震动场景使用，请做好核心板与底板的固定，避免核心板跌落损坏；
- ❖ 请勿在通电情况下，插拔核心板及外围模块(特别是串口模块)；
- ❖ 请勿自行维修、拆解本产品，如产品出现故障应及时联系本公司进行维修；
- ❖ 请勿自行修改或使用未经授权的配件，由此造成的损坏将不予保修；

资料的更新

为了确保您的资料是最新状态，请密切关注我们的动态，我们将会通过微信公众号和 QQ 群推送。

关注“迅为电子”微信公众号，不定期分享教程、资料和行业干货及产品一线资料。

迅为新媒体账号

官网：https://www.topeetboard.com

知乎 https://www.zhihu.com/people/topeetabc123

CSDN：https://blog.csdn.net/BeiJingXunWei

售后服务政策

1. 如产品使用过程中出现硬件故障可根据售后服务政策进行维修
2. 服务政策：参见官方网售后服务说明 https://www.topeetboard.com/sydymfl/Service/bx.html

送修地址：

1. 地址：北京市海淀区永翔北路 9 号中国航发大厦三层
2. 联系人：迅为开发板售后服务部
3. 电话：010-85270716
4. 邮编：100094
5. 邮寄须知：建议使用顺丰、圆通或韵达，且不接受任何到付

技术支持范围

1. 了解产品的软、硬件资源提供情况咨询
2. 产品的软、硬件手册使用过程中遇到的问题
3. 下载和烧写更新系统过程中遇到的问题
4. 产品用户的资料丢失、更新后重新获取
5. 产品的故障判断及售后维修服务。

PS：（由于嵌入式系统知识范围广泛，我们无法保证对各种问题都能一一解答，部分内容无法供技术支持，只能提供建议。 ）

技术支持

1. 周一至周五：（法定节假日除外）
上午 9:00 ~ 11:30 / 下午 13:30 ~ 17:30

2. QQ 技术交流群： 824412014
822183461
95631883
861311530

第 1 章 安卓应用开发说明

1.1 读者对象

这份文档适合以下人群：

- **Android 开发初学者**：如果您对 Android 应用开发流程不太熟悉，推荐先学习《13【北京迅为】itop-3588 开发板 android 系统和应用开发手册》。这本手册将帮助您掌握在 Android 平台上开发应用程序以及 JNI 基础。

  此外，北京迅为电子录制了 Android JNI 开发课程，帮助初学者快速入门。视频链接如下所示：

  jni 教程(一)https://www.bilibili.com/video/BV18v411i7Dx?p=1

  jni 教程(二)https://www.bilibili.com/video/BV18v411i7Dx?p=2

  jni 教程(三)https://www.bilibili.com/video/BV18v411i7Dx?p=3

  JNI 允许在 java 代码中调用 C/C++函数，从而访问开发板更底层的系统接口。学习完 JNI 视频之后，您可以开始阅读本文档，通过创建 Android JNI 项目来熟悉在 Android Studio 中使用 JNI 方法。

- **嵌入式系统工程师、Android 系统工程师、Android 应用工程师**：如果您需要与外部设备（如传感器、控制器等）进行通信，本文档提供了相关的应用案例及测试方法，以帮助开发更加高效。

1.2 文档内容

本文档具有极高的操作性，第 1 章详细介绍了安卓应用开发文档的受众群体，内容概述以及开发前的准备工作。第 2 章至第 6 章专注于 GPIO ，串口，RS485，CAN，PWM 和音视频编解码器的应用开发。为了精简内容，文档省略了部分 Android 应用工程的代码，仅对核心功能进行了深入解析。您可以通过本文档的附带资料查看完整的源代码，所有的源码均有详尽的注释。

1.3 前提准备

Android 系统是一个开放式的移动互联网操作系统，可以被安装在各种嵌入式产品上，例如 iTOP-RK3588 开发板。要在 iTOP-RK3588 上进行安卓应用开发，首先需要在开发板上烧写 Android 系统，本文档将在 Android13 系统上进行 Android 应用开发。

首先我们将开发板硬件连接好，使用 USB-TypeC 连接线连接到 OTG 端口。串口连接是可选的，同时可以选择连接迅为的屏幕或者 HDMI 屏幕。如果您没有屏幕，可以使用软件 QtScrcpy-win-x64-v1.9.0 进行显示。

连接好之后，开发板烧写 Android13 的系统镜像。并且在 Windows 上安装 IDE——Android Studio。

最后，在进行 Android 应用开发之前，建议先了解 ADB 工具的使用，可以查阅相关手册《18【北京迅为】itop-3588 开发板 ADB 使用手册》。

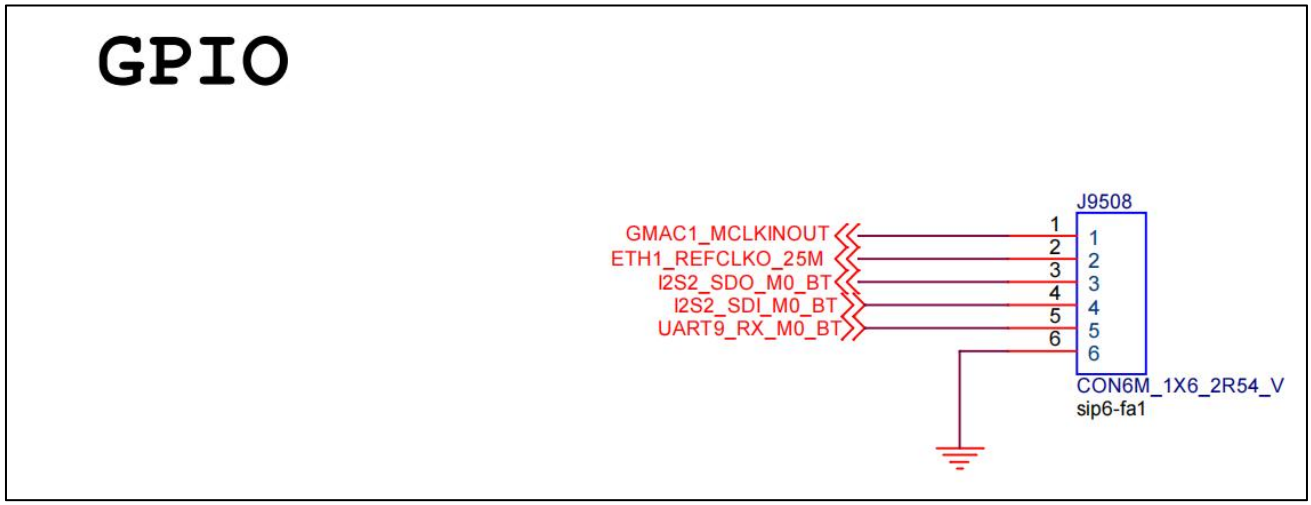
第 2 章 安卓 GPIO 调用

2.1 前置说明

Android GPIO 源码可以在“iTOP-3588 开发板\02_【iTOP-RK3588 开发板】开发资料\14_应用程序测试例程\01_AndroidStudioExample\01_gpio_test”获取，如何使用 Android Studio 加载 app 源码可以参考手册“13【北京迅为】itop-3588 开发板 android 系统和应用开发手册 v1.1”。

2.2 硬件连接

iTOP-RK3588 底板上 gpio 插槽原理图如下所示：



底板有 5 个 gpio 引脚引出，其中前四个引脚默认有其他复用须在设备树里取消掉对应的节点，才可进行控制，具体如下表所示：

Num	核心板连接器网络名称	GPIO	Vol	默认复用功能	引脚号
1	GMAC1_MCLKINOUT	GPIO3_B6_d	3.3V	ETH1	110
2	ETH1_REFCLKO_25M	GPIO3_A6_d	3.3V	ETH1	102
3	I2S2_SDO_M0_BT	GPIO4_C3_d	1.8V	ETH0	147
4	I2S2_SDI_M0_BT	GPIO2_C3_d	1.8V	ETH0	83
5	UART9_RX_M0_BT	GPIO2_C4_d	1.8V	无复用，为通用 IO	84
6	GND	--	GND	地	

本章中以操作 GPIO2_C4_d 引脚为例编写 Android GPIO 调用例程。GPIO2_C4_d 引脚的引脚号为 84。

2.3 Android 源码配置

在进行测试之前，请按照以下命令顺序修改 GPIO 接口的访问权限，否则可能由于权限问题导致错误。

```
adb root                # 以 root 用户的权限运行 adb
adb remount             # 重新挂载 Android 设备的文件系统，使系统分区变为可读写状态
adb shell               # 进入 adb shell，方便与 Android 设备交互
chmod 777 -R /sys/class/gpio    # 修改 GPIO 文件接口访问权限
```

```
PS F:\> adb devices
List of devices attached
68b639b85b2b1018        device

PS F:\>
PS F:\> adb root
restarting adbd as root
PS F:\> adb remount
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
Using overlayfs for /system
Using overlayfs for /vendor
Using overlayfs for /odm
Using overlayfs for /product
Using overlayfs for /system_ext
Now reboot your device for settings to take effect
remount succeeded
PS F:\>
```

```
PS F:\> adb shell
adb server is out of date.  killing...
* daemon started successfully *
topeet_rk3588:/ #

topeet_rk3588:/ # chmod 777 /sys/class/gpio
chmod 777 /sys/class/gpio
topeet_rk3588:/ #
```

我们进入 gpio 设备目录，输入下面的命令：

echo 84 > export //84 代表 gpio 号

```
topeet_rk3588:/ # cd /sys/class/gpio
cd /sys/class/gpio
topeet_rk3588:/sys/class/gpio # echo 84 > export
echo 84 > export
1|topeet_rk3588:/sys/class/gpio #
```

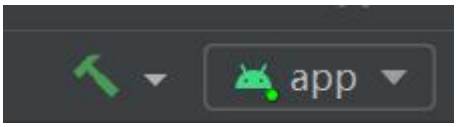
然后输入下面的命令修改 GPIO 文件接口访问权限

chmod 777 -R /sys/class/gpio/gpio84

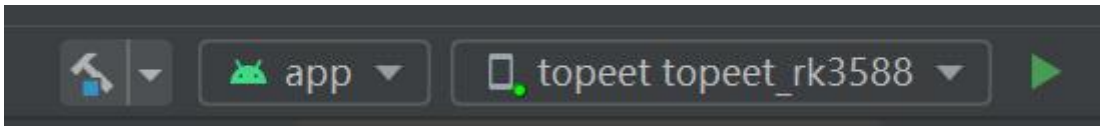
```
1|topeet_rk3588:/ # chmod 777 /sys/class/gpio/gpio84/*
chmod 777 /sys/class/gpio/gpio84/*
topeet_rk3588:/ #

topeet_rk3588:/ #
```

2.4 APP 运行测试

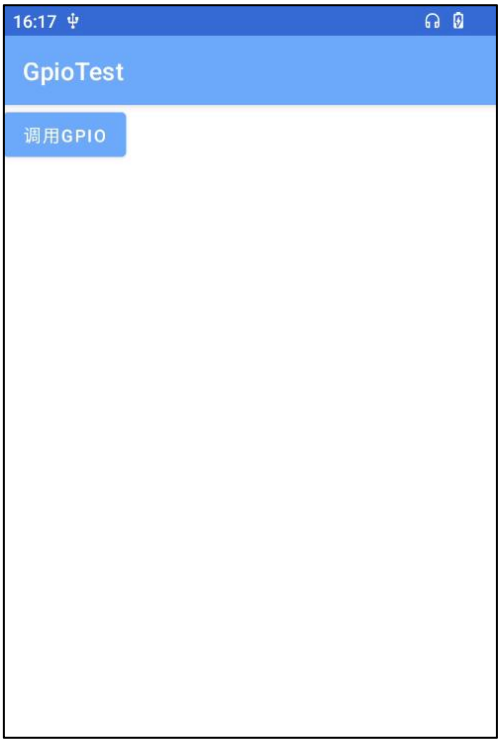
将网盘上的安卓工程文件复制到 Windows 电脑上。确保工程路径中使用英文字符，不包含中文。接着，启动 Android Studio，点击“Open”按钮选择应用工程文件夹，然后点击“OK”。由于下载 Gradle 和各种 Jar 包可能需要一段时间，Android Studio 加载工程可能会耗时较长甚至编译失败。如果编译失败，可以尝试多次点击工具栏上的绿色“小锤子”按钮重新编译，“小锤子”按钮如下图所示：



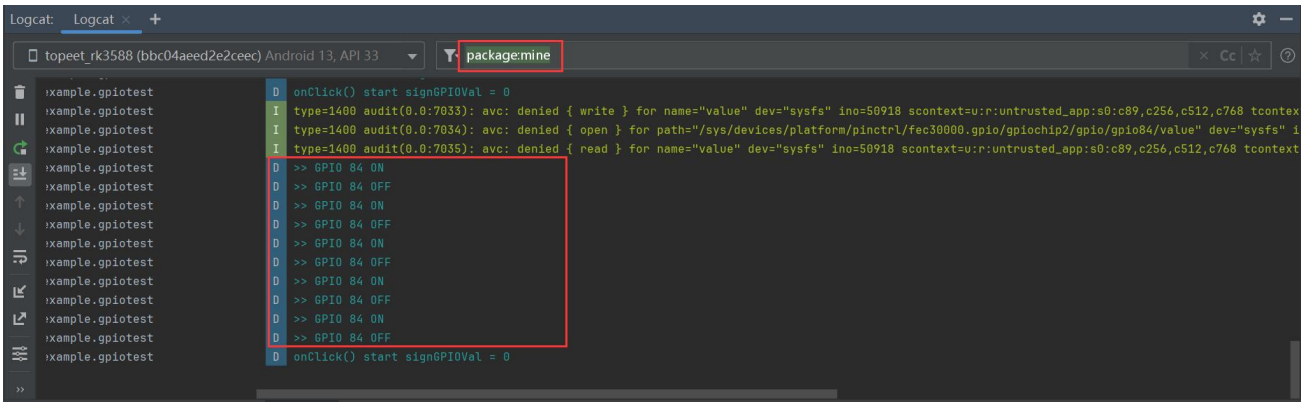
一旦源代码成功编译，选择目标设备后点击工具栏上的绿色三角形按钮即可运行应用程序，如下图所示：



如果 APP 运行成功，在开发板连接的屏幕上显示 App 界面，如下图所示：



打开 Android Studio 的 locat 日志打印窗口，筛选打印“package:mine”，然后点击 APP 界面的“调用 GPIO”按钮，会循环打印 GPIO 引脚打开和 GPIO 引脚关闭，如下图所示：



到此，安卓 GPIO 测试 App 的操作步骤就完成了。

2.5 核心功能代码解析

本小节主要对安卓 GPIO APP 的例程源码的核心功能进行解析，完整的代码可以通过本章第一小节描述的方式获取。

首先看一下 GPIO 的 JNI 本地方法，在 GpioTest\app\src\main\cpp\test_gpio.c 文件中，具体代码如下所示：

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <fcntl.h>
5   #include <unistd.h>
6   #include <sys/select.h>
7   #include <sys/stat.h>
8   #include <jni.h>
9   #include<android/log.h>
10
11  #define GPIO_PIN 84 //  定义要使用的 GPIO 引脚编号
12  #define   TAG "GPIO_JNI"     // 定义日志标签用于 Android 日志输出
13
14  typedef enum {
15      GPIO_OUTPUT = 1, // GPIO 输出模式
16      GPIO_OUTPUT_HIGH, // GPIO 输出高电平模式
17      GPIO_OUTPUT_LOW, // GPIO 输出低电平模式
18      GPIO_INPUT,    // GPIO 输入模式
19  } GPIO_DIRECT;
20
21  // 设置 GPIO 引脚方向（输入或输出）
```

```
22      int gpio_direction(int gpio, int dir) {
23          int ret = 0;
24          char buf[128];
25          // 构造 GPIO 方向文件路径
26          sprintf(buf, "/sys/class/gpio/gpio%d/direction", gpio);
27          int gpiofd = open(buf, O_WRONLY);
28          if (gpiofd < 0) {
29              __android_log_print(ANDROID_LOG_ERROR,     TAG,"Couldn't open gpio file");
30              ret = -1;
31              return -1;
32          }
33          // 根据 dir 参数设置 GPIO 方向
34          if (dir == GPIO_OUTPUT && gpiofd) {
35              if (3 != write(gpiofd, "out", 3)) {
36                  __android_log_print(ANDROID_LOG_ERROR,     TAG,"Couldn't set GPIO direction to out");
37                  ret = -2;
38              }
39          } else if (dir == GPIO_OUTPUT_HIGH && gpiofd) {
40              if (4 != write(gpiofd, "high", 4)) {
41                  __android_log_print(ANDROID_LOG_ERROR,     TAG,"Couldn't set GPIO direction to out
42  high");
43                  ret = -3;
44              }
45          } else if (dir == GPIO_OUTPUT_LOW && gpiofd) {
46              if (3 != write(gpiofd, "low", 3)) {
47                  __android_log_print(ANDROID_LOG_ERROR,     TAG,"Couldn't set GPIO direction to out
48  low");
49                  ret = -4;
50              }
51          } else if (gpiofd) {
52              if (2 != write(gpiofd, "in", 2)) {
53                  __android_log_print(ANDROID_LOG_ERROR,     TAG,"Couldn't set GPIO directio to in");
54                  ret = -5;
55              }
56          }
57
58          close(gpiofd);
59          return ret;
60      }
61      // 设置 GPIO 引脚的中断边沿检测
62      int gpio_set_edge(int gpio, int rising, int falling) {
63          int ret = 0;
64          char buf[128];
65          // 构造 GPIO 边沿检测文件路径
```

```
66              sprintf(buf, "/sys/class/gpio/gpio%d/edge", gpio);
67              int gpiofd = open(buf, O_WRONLY);
68              if (gpiofd < 0) {
69                      __android_log_print(ANDROID_LOG_ERROR,      TAG,"Couldn't open IRQ file");
70                  ret = -1;
71              }
72
73              if (gpiofd && rising && falling) {
74                  if (4 != write(gpiofd, "both", 4)) {
75                          __android_log_print(ANDROID_LOG_ERROR,      TAG,"Failed to set IRQ to both falling &
76  rising");
77                      ret = -2;
78                  }
79              } else {
80                  if (rising && gpiofd) {
81                      if (6 != write(gpiofd, "rising", 6)) {
82                              __android_log_print(ANDROID_LOG_ERROR,      TAG,"Failed to set IRQ to rising");
83                          ret = -2;
84                      }
85                  } else if (falling && gpiofd) {
86                      if (7 != write(gpiofd, "falling", 7)) {
87                              __android_log_print(ANDROID_LOG_ERROR,      TAG,"Failed to set IRQ to falling");
88                          ret = -3;
89                      }
90                  }
91              }
92
93              close(gpiofd);
94
95              return ret;
96          }
97          // 导出 GPIO 引脚到用户空间
98          int gpio_export(int gpio) {
99              int efd;
100             char buf[50];
101             int gpiofd, ret;
102
103             /* Quick test if it has already been exported */
104             sprintf(buf, "/sys/class/gpio/gpio%d/value", gpio);
105             efd = open(buf, O_WRONLY);
106             if (efd != -1) {
107                 close(efd);
108                 return 0;
109             }
```

```
110
111         efd = open("/sys/class/gpio/export", O_WRONLY);
112
113         if (efd != -1) {
114             sprintf(buf, "%d", gpio);
115             ret = write(efd, buf, strlen(buf));
116             if (ret < 0) {
117                 __android_log_print(ANDROID_LOG_ERROR,     TAG, "Export failed");
118                 return -2;
119             }
120             close(efd);
121         } else {
122             // If we can't open the export file, we probably
123             // dont have any gpio permissions
124             // 如果无法打开导出文件，可能是由于没有 GPIO 权限
125             return -1;
126         }
127         return 0;
128     }
129     // 取消导出 GPIO 引脚
130     void gpio_unexport(int gpio) {
132         int gpiofd, ret;
133         char buf[50];
134         gpiofd = open("/sys/class/gpio/unexport", O_WRONLY);
135         sprintf(buf, "%d", gpio);
136         ret = write(gpiofd, buf, strlen(buf));
137         close(gpiofd);
138     }
139     // 获取 GPIO 引脚值文件描述符
140     int gpio_getfd(int gpio) {
141         char in[3] = {0, 0, 0};
142         char buf[50];
143         int gpiofd;
144         sprintf(buf, "/sys/class/gpio/gpio%d/value", gpio); // 构造 GPIO 值文件路径
145         gpiofd = open(buf, O_RDWR);
146         if (gpiofd < 0) {
147             fprintf(stderr, "Failed to open gpio %d value\n", gpio);
148             __android_log_print(ANDROID_LOG_ERROR,     TAG,"gpio failed");
149         }
150         return gpiofd;
151     }
152
153     // 读取 GPIO 引脚的值
154     int gpio_read(int gpio) {
```

```
155             char in[3] = {0, 0, 0};
156             char buf[50];
157             int nread, gpiofd;
158             sprintf(buf, "/sys/class/gpio/gpio%d/value", gpio);// 构造 GPIO 值文件路径
159             gpiofd = open(buf, O_RDWR);
160             if (gpiofd < 0) {
161                 fprintf(stderr, "Failed to open gpio %d value\n", gpio);
162                 __android_log_print(ANDROID_LOG_ERROR,     TAG,"gpio failed");
163             }
164             // 循环读取直到成功
165             do {
166                 nread = read(gpiofd, in, 1);
167             } while (nread == 0);
168             if (nread == -1) {
169                 __android_log_print(ANDROID_LOG_ERROR,     TAG,"GPIO Read failed");
170                 return -1;
171             }
172
173             close(gpiofd);
174             return atoi(in);
175         }
176
177         // 向 GPIO 引脚写入值
178         int gpio_write(int gpio, int val) {
179             char buf[50];
180             int nread, ret, gpiofd;
181             sprintf(buf, "/sys/class/gpio/gpio%d/value", gpio);// 构造 GPIO 值文件路径
182             gpiofd = open(buf, O_RDWR);
183             if (gpiofd > 0) {
184                 snprintf(buf, 2, "%d", val);
185                 ret = write(gpiofd, buf, 2);
186                 if (ret < 0) {
187                     __android_log_print(ANDROID_LOG_ERROR,     TAG,"failed to set gpio");
188                     return 1;
189                 }
190
191                 close(gpiofd);
192                 if (ret == 2) return 0;
193             }
194             return 1;
195         }
196
197         // 使用 select 等待 GPIO 引脚的中断
197         int gpio_select(int gpio) {
```

```
198             char gpio_irq[64];
199             int ret = 0, buf, irqfd;
200             fd_set fds;
201             FD_ZERO(&fds);
202
203             snprintf(gpio_irq, sizeof(gpio_irq), "/sys/class/gpio/gpio%d/value", gpio);
204             irqfd = open(gpio_irq, O_RDONLY);
205             if (irqfd < 1) {
206                 __android_log_print(ANDROID_LOG_ERROR,      TAG,"Couldn't open the value file");
207                 return -13;
208             }
209
210             // 首次读取以清除任何初始状态
211             ret = read(irqfd, &buf, sizeof(buf));
212
213             while (1) {
214                 FD_SET(irqfd, &fds);
215                 // 使用 select 等待中断
216                 ret = select(irqfd + 1, NULL, NULL, &fds, NULL);
217                 if (FD_ISSET(irqfd, &fds)) {
218                     FD_CLR(irqfd, &fds);   // 从集合中移除文件描述符
219                     // Clear the junk data in the IRQ file
220                     ret = read(irqfd, &buf, sizeof(buf));
221                     return 1;
222                 }
223             }
224         }
225
226         JNIEXPORT jint JNICALL
227         Java_com_example_gpiotest_MainActivity_invokeGPIO( JNIEnv* env,
228                                 jobject mainActivity)
229                                 {
230             // 定义 GPIO 引脚号
231             int gpio_pin = GPIO_PIN;
232             // 导出 GPIO 引脚
233             gpio_export(gpio_pin);
234             // 设置 GPIO 引脚方向为输出
235             gpio_direction(gpio_pin, 1);
236             // 循环控制 GPIO 引脚输出状态
237             for (int i = 0; i < 5; i++) {
238                 // 设置 GPIO 引脚输出高电平（1）
239                 gpio_write(gpio_pin, 1);
240                 // 打印调试信息，显示 GPIO 引脚打开
241                 __android_log_print(ANDROID_LOG_DEBUG,      TAG,">> GPIO %d ON\n", gpio_pin);
```

```
242                 // 等待 1 秒钟
243                 sleep(1);
244                 // 设置 GPIO 引脚输出低电平（0）
245                 gpio_write(gpio_pin, 0);
246                 // 打印调试信息，显示 GPIO 引脚关闭
247                 __android_log_print(ANDROID_LOG_DEBUG,     TAG,">> GPIO %d OFF\n", gpio_pin);
248                 // 再次等待 1 秒钟
249                 sleep(1);
250             }
251             // 返回操作成功的标志（0 表示成功）
252             return 0;
253 }
```

在上面的代码中首先引入了头文件，接下来 GPIO_PIN 定义了要操作的 GPIO 引脚号。TAG 用于 Android 日志输出。GPIO_DIRECT 是一个枚举，定义了 GPIO 的几种操作模式，包括输出、输出高电平、输出低电平和输入。

- 代码的第 21-224 行定义了 gpio 函数，如下所示：

  gpio_direction(int gpio, int dir)：设置 GPIO 引脚的方向（输入或输出）。

  gpio_set_edge(int gpio, int rising, int falling)：设置 GPIO 引脚的中断边沿检测。

  gpio_export(int gpio)：将 GPIO 引脚导出到用户空间。

  gpio_unexport(int gpio)：取消导出 GPIO 引脚。

  gpio_getfd(int gpio)：获取 GPIO 引脚值的文件描述符。

  gpio_read(int gpio)：读取 GPIO 引脚的值。

  gpio_write(int gpio, int val)：向 GPIO 引脚写入值。

  gpio_select(int gpio)：使用 select 函数等待 GPIO 引脚的中断。

- 代码的 226 行-250 行是 JNI 方法，用于在 Android 应用中控制 GPIO 引脚。通过调用 gpio_export 导出 GPIO 引脚到用户空间。通过调用 gpio_direction 将 GPIO 引脚设置为输出模式。循环中通过 gpio_write 函数控制 GPIO 引脚的输出状态，然后打印相应的日志信息，并使用 sleep 函数进行等待。

总之，上面的代码展示了如何在 Android 平台上使用 JNI 来控制 GPIO 引脚的输出状态，通过文件操作实现对 GPIO 的控制和状态读取。

到此，核心功能代码分析完毕。

第 3 章 安卓串口 RS485 App 开发

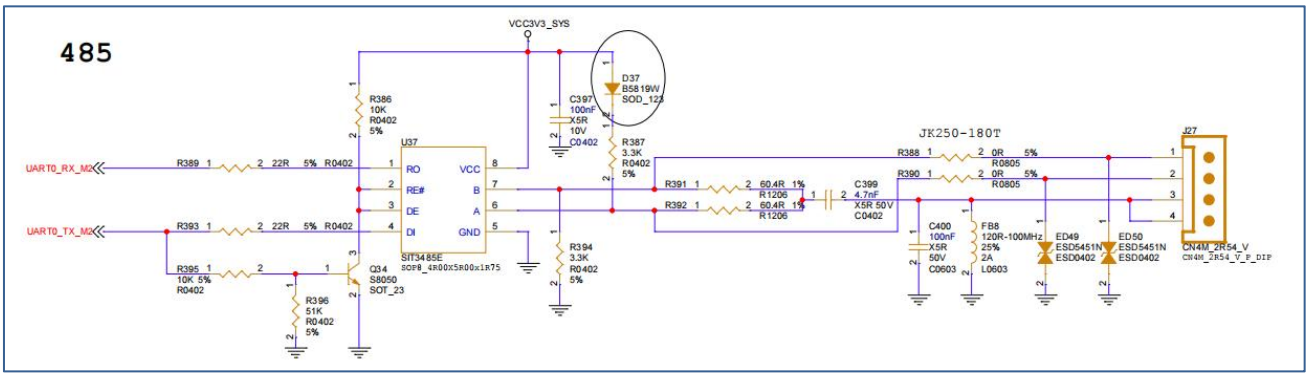
3.1 前置说明

迅为 iTOP-RK3588 底板上有 2 路串口，一路用做调试串口，一路用做 RS485。测试串口和 RS485 可以使用同一个应用程序。

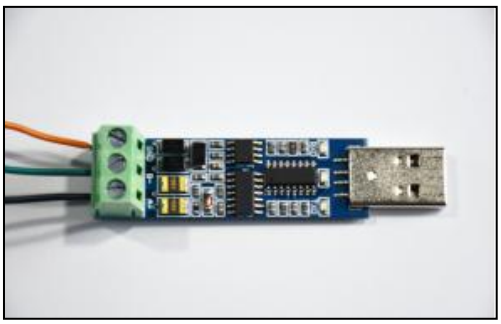
Android 串口和 RS485 App 源码可以在“iTOP-3588 开发板\02_【iTOP-RK3588 开发板】开发资料\14_应用程序测试例程\01_AndroidStudioExample\02_uart_485_test”获取，如何使用 Android Studio 加载 app 源码可以参考手册“13【北京迅为】itop-3588 开发板 android 系统和应用开发手册 v1.1”。

3.2 硬件连接

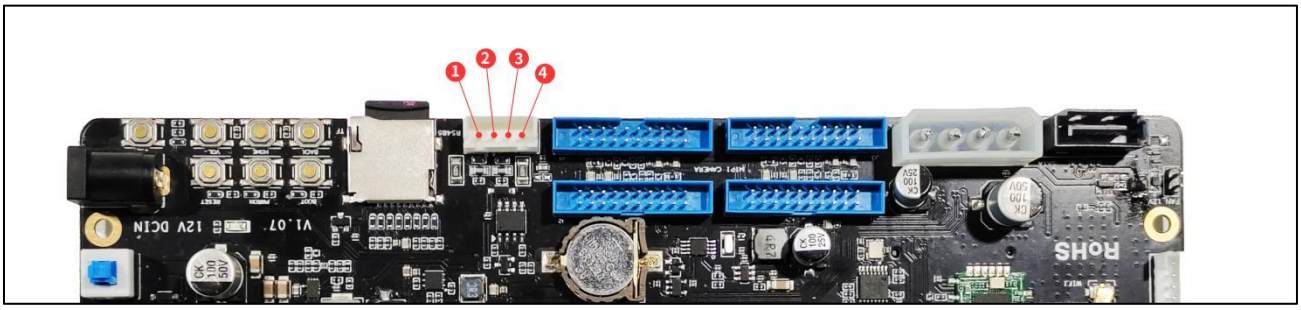
485 电路图如下图所示：



由原理图可知，485 使用的串口 0，可以通过/dev/ttyS0 来控制。接下来使用 USB 转 RS485 模块进行测试（需要自行准备）。USB 转 RS485 模块如下图所示：



由原理图可知，开发板底板上 1 号引脚是 B，2 号引脚是 A ，4 号引脚连接地，然后将 1 2 引脚连接到 USB 转 RS485 模块上，A 接 A，B 接 B，如下图所示：



USB 转 RS485 模块的 usb 端口连接到电脑上。

3.3 Android 源码配置

迅为提供的 Android13 源码默认配置上了调试串口和 RS485。如果要测试调试串口，首先要将调试串口修改为普通串口。RS485 对应的设备节点为/dev/ttyS0。

在进行测试之前，请按照以下命令顺序修改 RS485 接口的访问权限，否则可能由于权限问题导致错误。

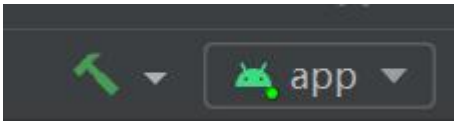
```
adb root            # 以 root 用户的权限运行 adb
adb remount         # 重新挂载 Android 设备的文件系统，使系统分区变为可读写状态
adb shell           # 进入 adb shell，方便与 Android 设备交互
chmod 777 /dev/ttyS0    # 修改串口 RS485 文件接口访问权限
```

```
PS F:\> adb devices
List of devices attached
68b639b85b2b1018        device

PS F:\>
PS F:\> adb root
restarting adbd as root
PS F:\> adb remount
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
Using overlayfs for /system
Using overlayfs for /vendor
Using overlayfs for /odm
Using overlayfs for /product
Using overlayfs for /system_ext
Now reboot your device for settings to take effect
remount succeeded
PS F:\>
```

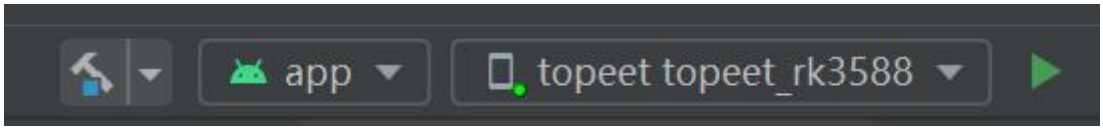
```
PS F:\> adb shell
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
topeet_rk3588:/ # chmod 777 /dev/ttyS0
chmod 777 /dev/ttyS0
topeet_rk3588:/ #
```

3.4 APP 运行测试

将网盘上的安卓工程文件复制到 Windows 电脑上。确保工程路径中使用英文字符，不包含中文。接着，启动 Android Studio，点击“Open”按钮选择应用工程文件夹，然后点击“OK”。由于下载 Gradle 和各种 Jar 包可能需要一段时间，Android Studio 加载工程可能会耗时较长甚至编译失败。如果编译失败，可以尝试多次点击工具栏上的绿色“小锤子”按钮重新编译，“小锤子”按钮如下图所示：



一旦源代码成功编译，选择目标设备后点击工具栏上的绿色三角形按钮即可运行应用程序，如下图所示：

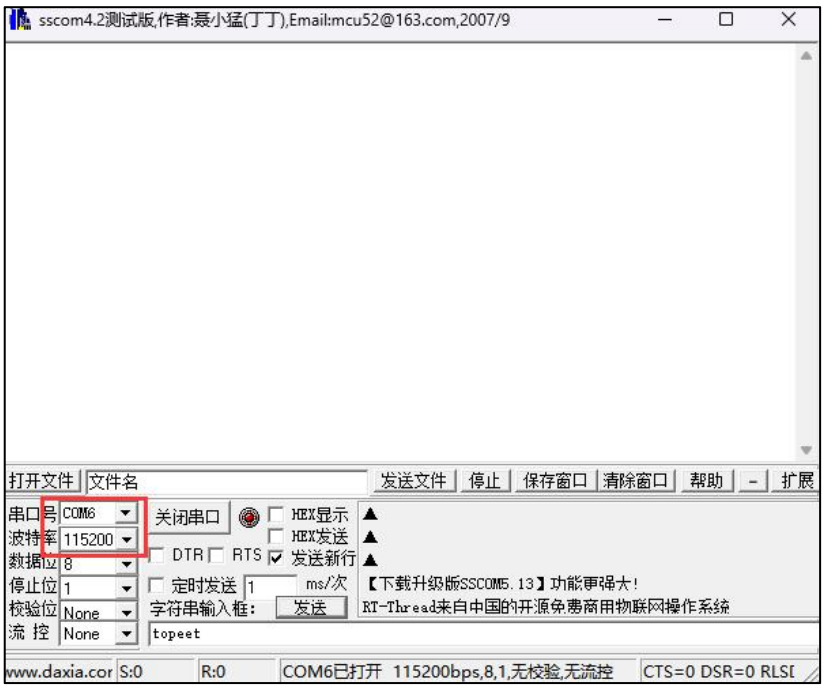


如果 APP 运行成功，在开发板连接的屏幕上显示 App 界面，如下图所示：

我们可以选择测试的串口的波特率，这里我们选择/dev/ttyS0(RS485)和 115200 波特率。选择好之后，点击“打开串口”按钮，如果串口打开成功，打开按钮会置灰表示不可以使用点击状态，“关闭串口”和“SEND”这俩个按钮会从灰色变成蓝色表示可以使用点击状态，如下图所示：

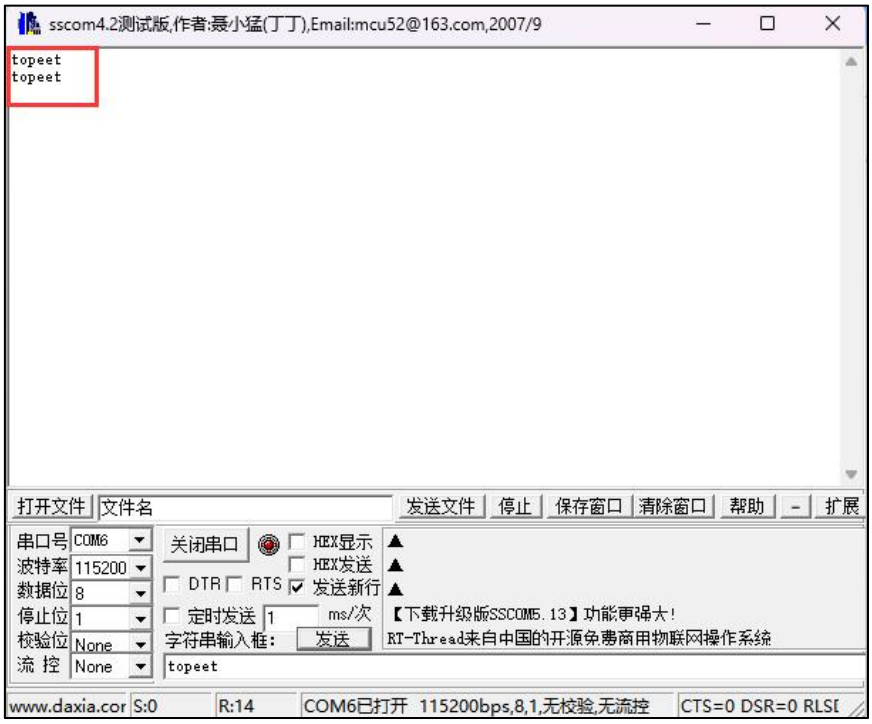
在电脑上打开串口助手，选择对应的串口号和波特率，注意：默认波特率为 115200！打

开串口，如下图所示：

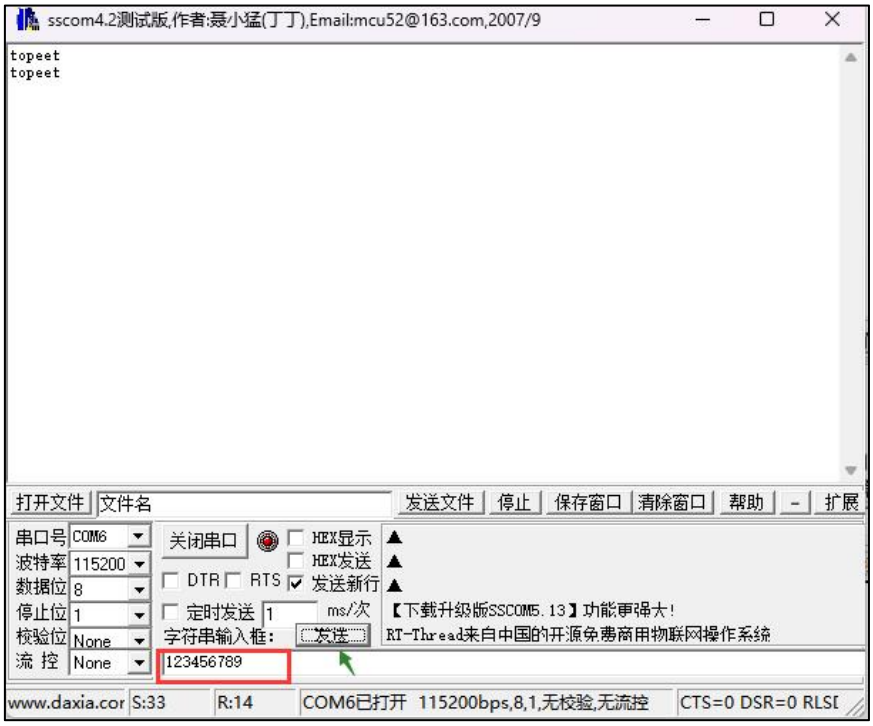


打开串口之后，在底部的文本框中输入自定义的字符，然后点击旁边的发送按钮，效果如下图所示：

电脑端接收到信息，如下图所示：



电脑端发送数据，如下图所示：



开发板收到数据，如下图所示：

到此，安卓串口 485 测试 App 的操作步骤就完成了。

3.5 核心功能代码解析

本小节主要对安卓串口 485 APP 的例程源码的核心功能进行解析，完整的代码可以通过本章第一小节描述的方式获取。

首先看一下串口设备的 JNI 本地方法，方法声明在 uartRs485Test\app\src\main\java\com\example\uartrs485test\serial.java 文件中，具体代码如下所示：

```
1   // 声明一个名为 serial 的 Java 类
2   public class serial {
3
4   // 声明 native 方法 open，用于打开串口连接
5   public native int open(int Port, int Rate);
6
7   // 声明 native 方法 close，用于关闭串口连接
8   public native int close();
9
10  // 声明 native 方法 read，用于从串口读取数据
11  public native byte[] read();
12
13  // 声明 native 方法 write，用于向串口写入数据
```

```
14      public native int write(byte[] buffer);
15
16  }
```

上面这 4 个本地方法的声明按顺序分别是打开串口设备、关闭串口设备、从串口读数据、向串口写入数据，上述 4 个本地方法都是在 uartRs485Test\app\src\main\cpp\native-lib.cpp 文件里面实现。

接着，我们先来看打开 Can 设备的本地方法的实现代码，具体的代码如下：

```
1   // 打开串口
2   extern "C"
3   JNIEXPORT jint JNICALL
4   Java_com_example_uartrs485test_serial_open(JNIEnv *env, jobject thiz, jint port, jint rate) {
5       // 如果串口文件描述符小于等于 0，则表示串口未打开
6       if (fd <= 0) {
7           // 根据传入的端口号打开对应的串口设备文件
8           const char* portName;
9           switch(port) {
10              case 0: portName = "/dev/ttyS0"; break;
11              case 1: portName = "/dev/ttyS1"; break;
12              case 2: portName = "/dev/ttyS2"; break;
13              case 3: portName = "/dev/ttyS3"; break;
14              case 4: portName = "/dev/ttyS4"; break;
15              case 5: portName = "/dev/ttyS5"; break;
16              case 6: portName = "/dev/ttyS6"; break;
17              case 7: portName = "/dev/ttyS7"; break;
18              case 8: portName = "/dev/ttyS8"; break;
19              case 9: portName = "/dev/ttyS9"; break;
20              default: {
21                  __android_log_print(ANDROID_LOG_INFO, "serial", "Parameter Error serial not found");
22                  fd = 0;
23                  return -1;
24              }
25          }
26
27          // 打开串口设备文件
28          __android_log_print(ANDROID_LOG_INFO, "serial", "open fd %s", portName);
29          fd = open(portName, O_RDWR | O_NOCTTY | O_NONBLOCK);
30
31          // 判断串口是否成功打开
32          if (fd > 0) {
33              __android_log_print(ANDROID_LOG_INFO, "serial", "serial open ok fd=%d", fd);
34              struct termios ios;
```

```
35                      tcgetattr(fd, &ios);
36                      ios.c_oflag &= ~(INLCR | IGNCR | ICRNL);
37                      ios.c_oflag &= ~(ONLCR | OCRNL);
38                      ios.c_iflag &= ~(ICRNL | IXON);
39                      ios.c_iflag &= ~(INLCR | IGNCR | ICRNL);
40                      ios.c_iflag &= ~(ONLCR | OCRNL);
41                      tcflush(fd, TCIFLUSH);
42
43                      // 根据传入的波特率设置串口参数
44                      speed_t speed;
45                      switch(rate) {
46                          case 0: speed = B2400; break;
47                          case 1: speed = B4800; break;
48                          case 2: speed = B9600; break;
49                          case 3: speed = B19200; break;
50                          case 4: speed = B38400; break;
51                          case 5: speed = B57600; break;
52                          case 6: speed = B115200; break;
53                          default: speed = B9600; break;
54                      }
55                      cfsetospeed(&ios, speed);
56                      cfsetispeed(&ios, speed);
57
58                      ios.c_cflag |= (CLOCAL | CREAD);
59                      ios.c_cflag &= ~PARENB;
60                      ios.c_cflag &= ~CSTOPB;
61                      ios.c_cflag &= ~CSIZE;
62                      ios.c_cflag |= CS8;
63                      ios.c_lflag = 0;
64                      tcsetattr(fd, TCSANOW, &ios);
65                  } else {
66                      __android_log_print(ANDROID_LOG_INFO, "serial", "serial open failed fd=%d", fd);
67                  }
68              }
69              // 返回串口文件描述符
70              return fd;
71  }
```

上述代码的第 10-19 行设置了要打开的串口设备节点，如果需要设置其他的参数，可以自己手动修改。

上述代码的第 46-52 行设置了串口的波特率，如果需要设置其他的参数，可以自己手动修改。

打开串口的本地方法 Java_com_example_uartrs485test_serial_open 的具体解析可以查看代码注释，现在 接着看关闭串口的本地方法，代码如下所示：

```
1     // 关闭串口
2     extern "C"
3     JNIEXPORT jint JNICALL
4     Java_com_example_uartrs485test_serial_close(JNIEnv *env, jobject thiz) {
5         // 如果串口文件描述符大于 0，则关闭串口
6         if (fd > 0) close(fd);
7         // 将串口文件描述符设为 -1，表示串口已关闭
8         fd = -1;
9         // 返回 1，表示关闭成功
10        return 1;
11    }
```

在上述代码中主要调用了 close 函数关闭释放了串口设备的文件描述，现在本地方法还剩下俩个本地方法，一个是读取串口数据，一个是发送串口数据，先来看一下读取串口数据的本地方法 Java_com_example_uartrs485test_serial_read，具体代码如下所示：

```
1     // 从串口读取数据
2     extern "C"
3     JNIEXPORT jbyteArray JNICALL
4     Java_com_example_uartrs485test_serial_read(JNIEnv *env, jobject thiz) {
5         unsigned char buffer[512];  // 用于存储读取的数据
6         int len = 0;  // 实际读取的数据长度
7         memset(buffer, 0, sizeof(buffer));  // 清空缓冲区
8
9         // 循环读取数据
10        while (fd > 0) {
11            len = read(fd, buffer, 512);  // 从串口读取数据
12
13            // 如果读取失败或没有数据可读，则等待一段时间后继续读取
14            if (len <= 0) {
15                usleep(500);  // 等待 500 毫秒
16                continue;
17            }
18
19            // 创建一个新的字节数组，并将读取的数据复制到该数组中
20            jbyteArray array = (*env).NewByteArray(len);
21            (*env).SetByteArrayRegion(array, 0, len, reinterpret_cast<const jbyte *>(buffer));
22
23            // 释放数组元素，并返回数组
24            env->ReleaseByteArrayElements(array, env->GetByteArrayElements(array, JNI_FALSE), 0);
25            return array;
```

```
26          }
27
28          // 如果没有成功读取数据，则返回空数组
29          return nullptr;
30      }
```

上面的代码我们主要看 while 循环里面的代码即第 10-26 行。

第 10 行代码用于检查文件描述符 fd 是否大于 0，如果是，则进入循环体，这标明串口已经打开且有效。

第 11 行代码使用 read 函数尝试从串口文件描述符 fd 中读取最多 512 字节的数据到 buffer 中，并将实际读取的字节保存在 len 中

第 14-17 行代码检查读取结果

第 20-21 行处理读取成功的情况，如果成功读取到数据（len>0）,则会创建一个新的 jbyteArray 对象 array，并将从 buffer 中读取的数据复制到这个数组中。

第 24 行代码，释放 jbyteArray 的内存。

现在本地方法里还有一个发送串口数据的本地方法 Java_com_example_uartrs485test_serial_write，具体代码如下所示：

```
1
2       // 向串口写入数据
3       extern "C"
4       JNIEXPORT jint JNICALL
5       Java_com_example_uartrs485test_serial_write(JNIEnv *env, jobject thiz, jbyteArray buf) {
6           // 获取字节数组的长度
7           jsize encoded_length = env->GetArrayLength(buf);
8           // 创建一个缓冲区，用于存储字节数组中的数据
9           jbyte *buffer = new jbyte[encoded_length];
10          // 将字节数组中的数据复制到缓冲区中
11          env->GetByteArrayRegion(buf, 0, encoded_length, buffer);
12          // 将数据写入到串口中
13          return write(fd, buffer, encoded_length);
14      }
```

上面代码允许从 Java 层向 C/C++层传递字节数组数据，并通过系统调用 write 将这些数据写入到串口中，实现了 Java 应用与底层串口通信的功能。

到此，核心功能代码分析完毕。

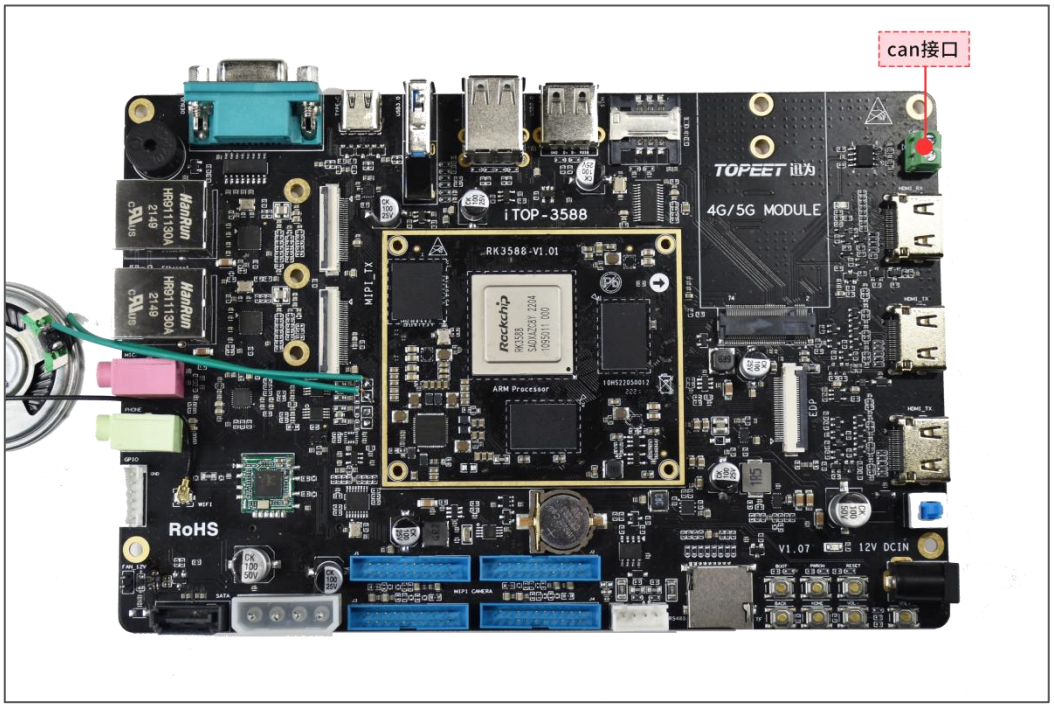
第 4 章 安卓 CAN 通信 APP 开发

4.1 前置说明

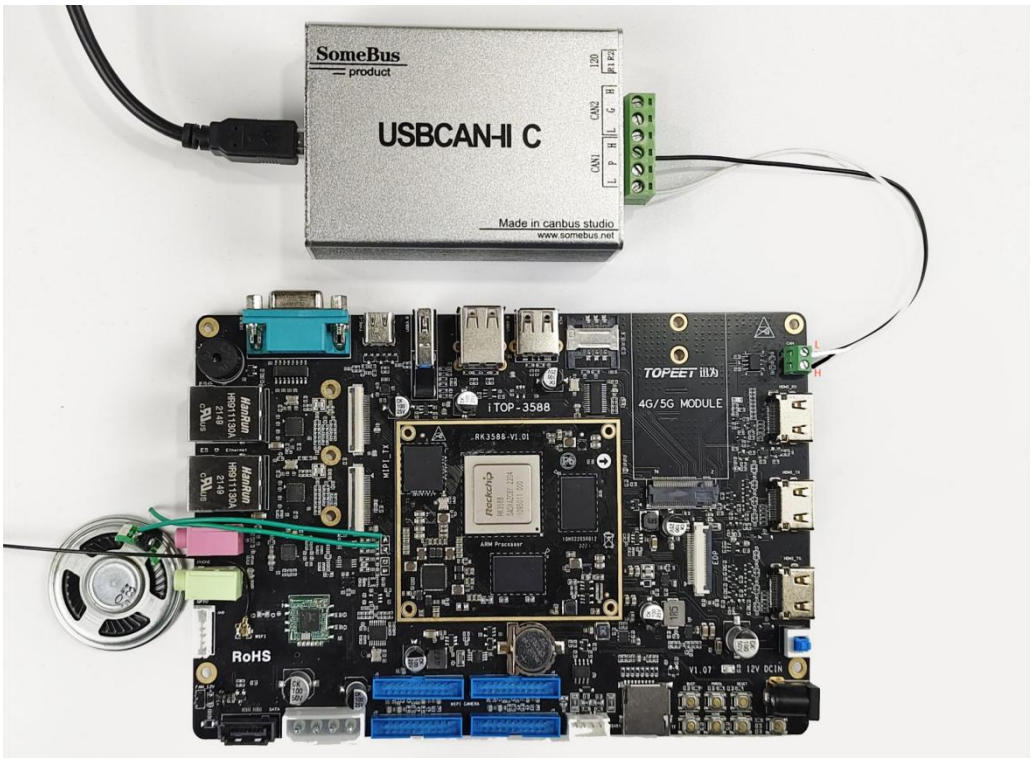
Android can 通信源码可以在“iTOP-3588 开发板\02_【iTOP-RK3588 开发板】开发资料\14_应用程序测试例程\01_AndroidStudioExample\03_can_test”获取，如何使用 Android Studio 加载 app 源码可以参考手册“13【北京迅为】itop-3588 开发板 android 系统和应用开发手册 v1.1”。

4.2 硬件连接

iTOP-3588 开发板支持 can 接口，底板上 can 接口如下图所示：



本次测试使用了 usbcan 调试分析仪作为测试设备，将 usbcan 调试分析仪通道一的 L 连到开发板的 L， 将通道一的 H 连到开发板的 H，如下图所示：



4.3 Android 源码配置

迅为提供的 Android13 源码默认配置上了 can0。

在进行测试之前，请按照以下命令顺序设置 can0 ，否则可能由于 can 设备配置导致通信错误。

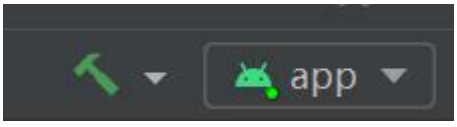
```
adb root            # 以 root 用户的权限运行 adb
adb remount         # 重新挂载 Android 设备的文件系统，使系统分区变为可读写状态
adb shell           # 进入 adb shell，方便与 Android 设备交互
ip link set can0 down #关闭 can 设备
ip link set can0 up type can bitrate 1000000 dbitrate 3000000 fd on     # 设置 can0 仲裁段 1M 波特率，数据段 3M 波特率，并启动 can0
```

```
PS F:\> adb devices
List of devices attached
68b639b85b2b1018        device

PS F:\>
PS F:\> adb root
restarting adbd as root
PS F:\> adb remount
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
Using overlayfs for /system
Using overlayfs for /vendor
Using overlayfs for /odm
Using overlayfs for /product
Using overlayfs for /system_ext
Now reboot your device for settings to take effect
remount succeeded
PS F:\>
```

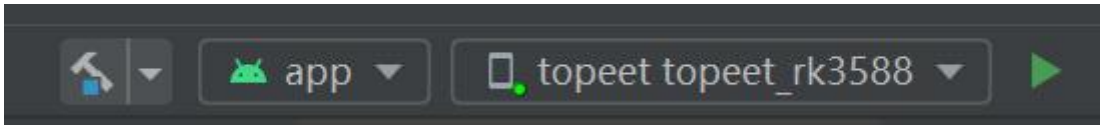
```
PS F:\> adb shell
adb server is out of date.  killing...
* daemon started successfully *
topeet_rk3588:/ # ip link set can0 down
ip link set can0 down
topeet_rk3588:/ # ip link set can0 up type can bitrate 1000000 dbitrate 3000000 fd on
 set can0 up type can bitrate 1000000 dbitrate 3000000 fd on                    <
topeet_rk3588:/ #

topeet_rk3588:/ #
```

4.4 APP 运行测试

将网盘上的安卓工程文件复制到 Windows 电脑上。确保工程路径中使用英文字符，不包含中文。接着，启动 Android Studio，点击“Open”按钮选择应用工程文件夹，然后点击“OK”。由于下载 Gradle 和各种 Jar 包可能需要一段时间，Android Studio 加载工程可能会耗时较长甚至编译失败。如果编译失败，可以尝试多次点击工具栏上的绿色“小锤子”按钮重新编译，“小锤子”按钮如下图所示：



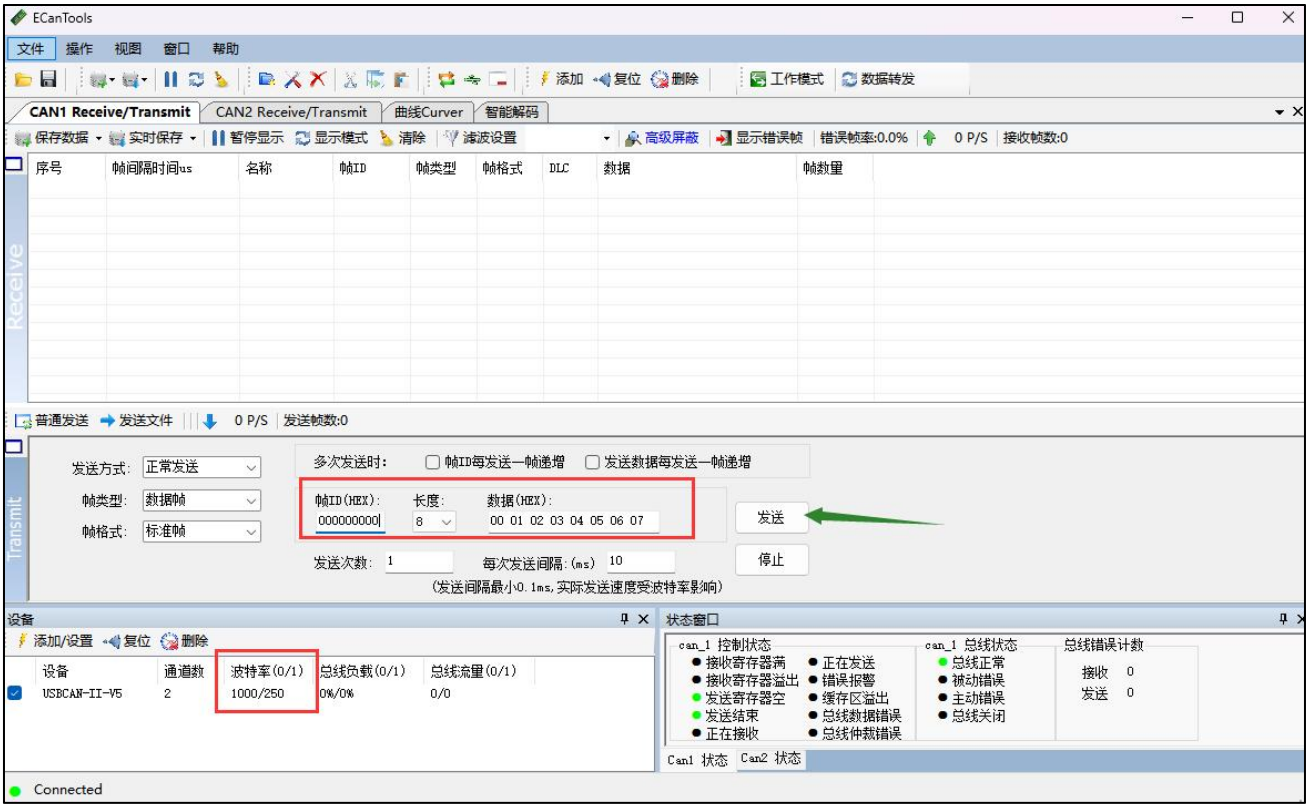
一旦源代码成功编译，选择目标设备后点击工具栏上的绿色三角形按钮即可运行应用程序，如下图所示：

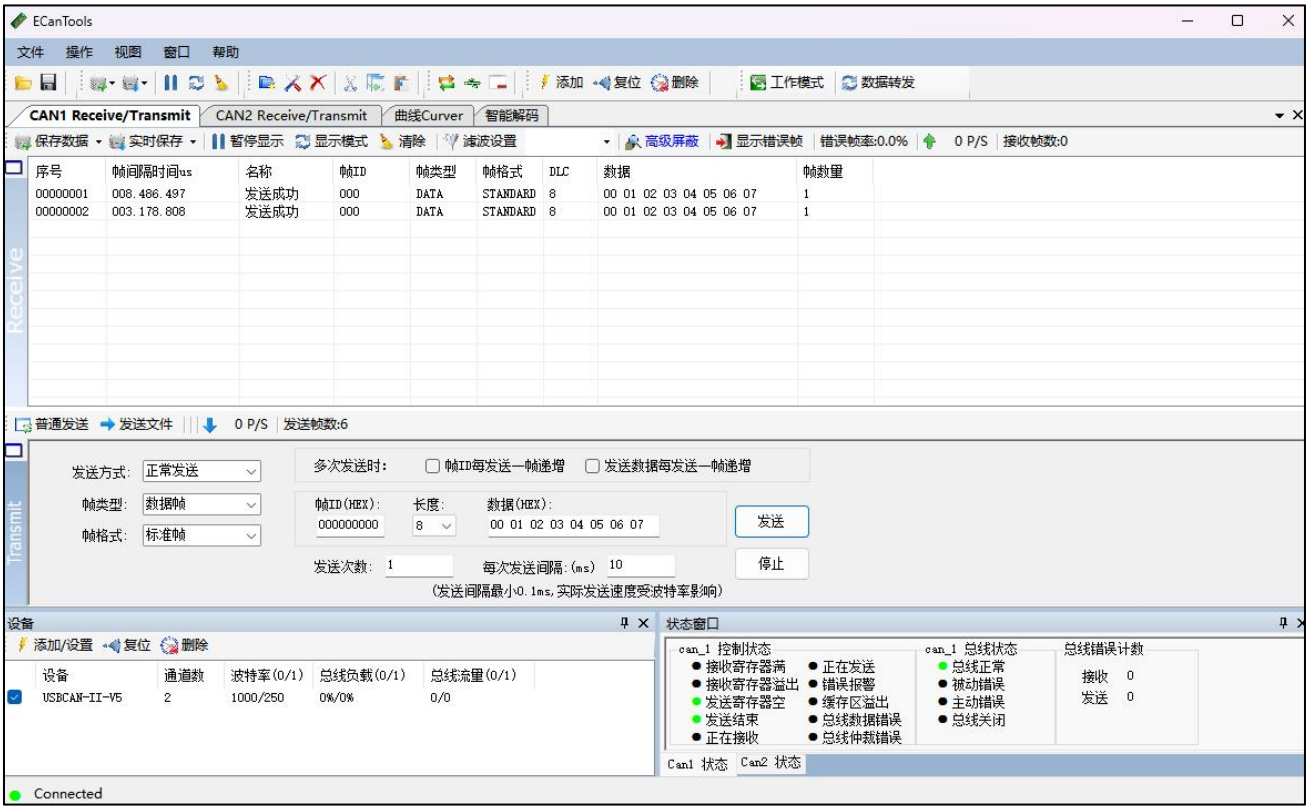


如果 APP 运行成功，在开发板连接的屏幕上显示 App 界面，如下图所示：

我们可以选择测试的 CAN 设备，这里我们选择 can0 设备。选择好之后，点击“打开设备”按钮，如果 CAN 设备打开成功，打开按钮会置灰表示不可以使用点击状态，“关闭设备”和“发送”这俩个按钮会从灰色变成蓝色表示可以使用点击状态，如下图所示：

然后在 PC 端 usbcan 的上位机发送数据如下图所示：



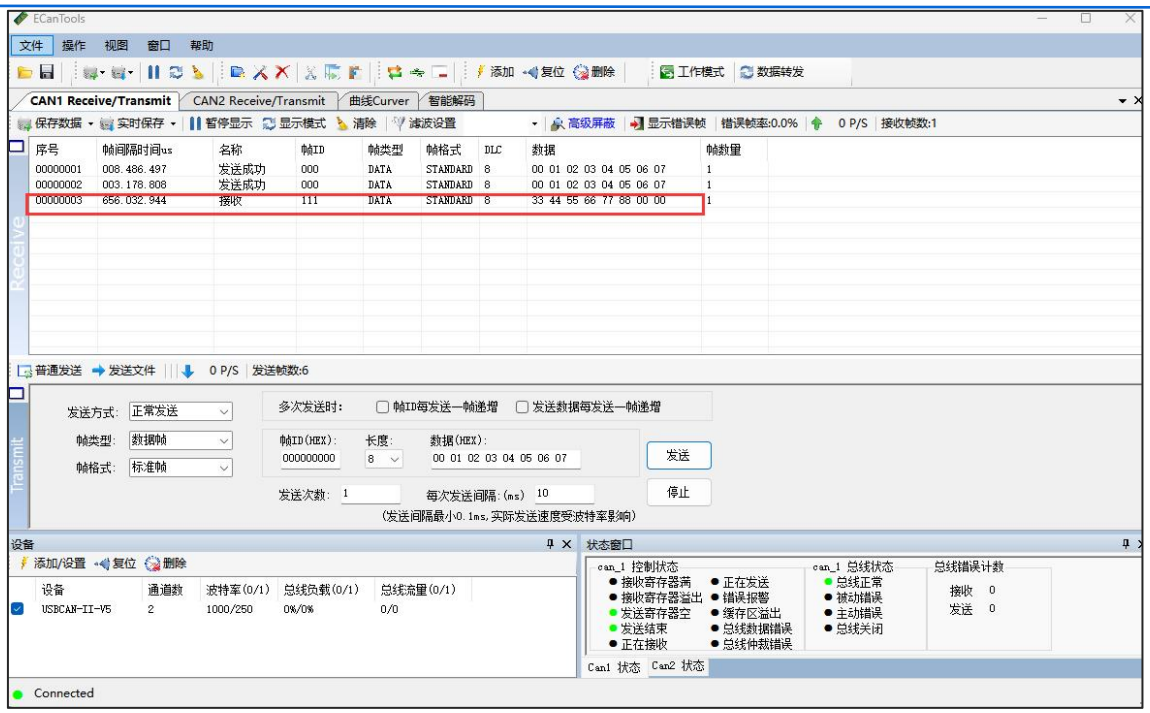


上位机的数据发送后，开发板的 APP 端会收到相应的数据，如下图所示：

接下来将 iTOP-3588 开发板作为发送端，在 APP 上发送相应的数据，如下图所示：

发送数据时，请确保数据格式正确：前三位为帧 ID，其中 111 为帧 ID，后面的数据为 CAN 发送的数据（例如：334455667788）。输入值必须为有效的 16 进制数字字符串，并且数据位数不超过 16 位。

然后在 usncan 的 Windows 上位机会返回如下信息：



至此，can 功能测试完毕。

4.5 核心功能代码解析

本小节主要对安卓 CAN 通信 APP 的例程源码的核心功能进行解析，完整的代码可以通过本章第一小节描述的方式获取。

首先看一下 CAN 设备的 JNI 本地方法，方法声明在 can_test\app\src\main\java\com\example\cantest\CanUtil.java 文件中，具体代码如下所示：

```
1    public class CanUtil {
2        public static native int openCan(String canDevName);
3        public static native int senddata(int[] data);
4        public static native int[] recvdata();
5        public static native void closeCan();
     }
```

上面这 4 个本地方法的声明按顺序分别是打开 Can 设备、Can 设备发送数据、Can 设备读取数据和关闭 Can 设备，上述 4 个本地方法都是在 can_test\app\src\main\cpp\native-lib.cpp 文件里面实现。

接着，我们先来看打开 Can 设备的本地方法的实现代码，具体的代码如下：

```
1    extern "C" JNIEXPORT jint JNICALL
2    Java_com_example_cantest_CanUtil_openCan(JNIEnv *env,jclass clazz,jstring can_dev_name){
```

```
3           //打开套接字
4           sockfd = socket(PF_CAN, SOCK_RAW, CAN_RAW);
5           if(sockfd < 0){
6               LOGE("打开 can 失败");
7               return -1;
8           }
9           // 将接口名设置为 "can0"
10          strcpy(ifr.ifr_name, "can0");
11          // 获取接口索引
12          ioctl(sockfd,SIOGIFINDEX,&ifr);
13          // 设置 CAN 套接字地址族和接口索引
14          can_addr.can_family = AF_CAN;
15          can_addr.can_ifindex = ifr.ifr_ifindex;
16          // 将 can0 与套接字进行绑定
17          ret = bind (sockfd, (struct sockaddr *)(&can_addr), sizeof(can_addr));
18          if(ret<0){
19              LOGE("绑定 can 设备失败");
20              close(sockfd);
21              return -1;
22          }
23
24          return 0;
25  }
```

上面代码第 4 行使用 socket 函数创建了一个 CAN 套接字，PF_CAN 表示协议族为 CAN,SOCK_RAW 表示原始套接字，CAN_RAW 表示原始 CAN 协议。

第 5-8 行代码检查套接字创建是否成功

第 10-11 行代码将接口名设置为 "can0"，这是 CAN 设备的默认名称。使用 ioctl 函数获取 CAN 接口的索引存储在 ifr.ifr_ifindex 中。SIOGIFINDEX 是一个 ioctl 操作，用于获取网络接口的索引。

第 14-15 行代码设置 CAN 套接字地址族为 AF_CAN，表示使用 CAN 协议。将之前获取到的 CAN 接口索引设置到 can_addr.can_ifindex 中。

第 17 行代码使用 bind() 函数将 CAN 套接字 sockfd 与指定的 CAN 设备（使用 can_addr 结构体描述）进行绑定。

第 18-22 行代码如果绑定失败，则记录错误日志，关闭套接字并返回-1。

打开 CAN 设备的本地方法 Java_com_example_cantest_CanUtil_openCan 的具体解析可以查看代码注释，现在 接着看关闭 CAN 设备的本地方法，代码如下所示：

```
1   extern "C"
2   JNIEXPORT void JNICALL
```

```
3       Java_com_example_cantest_CanUtil_closeCan(JNIEnv *env, jclass clazz) {
4           if (sockfd >= 0) {
5               close(sockfd);
6               LOGD("关闭 can 成功");
7           }
8   }
```

在上述代码中主要调用了 close 函数关闭释放了 CAN 设备的文件描述，现在本地方法还剩下俩个本地方法，一个是 Can 设备发送数据，一个是 Can 设备读取数据，先来看一下 Can 设备发送数据的本地方法 Java_com_example_cantest_CanUtil_senddata，具体代码如下所示：

```
1       extern "C"
2       JNIEXPORT jint JNICALL
3       Java_com_example_cantest_CanUtil_senddata(JNIEnv *env, jclass clazz,jintArray data){
4           if(sockfd < 0){
5               LOGE("sendData: 打开 can 设备失败");
6               return -1;
7           }
8
9           jsize length = env->GetArrayLength(data);
10          jint *ptr = env->GetIntArrayElements(data,JNI_FALSE);
11          if(length <1 || length > 9){
12              LOGE("senddata: 数据长度无效");
13              env->ReleaseIntArrayElements(data,ptr,0);
14              return -1;
15          }
16
17
18          //填充 can 的帧 ID 和数据个数
19          frame.can_dlc = length -1;
20          frame.can_id = ptr[0];
21          //填充 can 帧的数据
22          for(int i = 1;i<length;i++){
23              frame.data[i-1] = ptr[i];
24
25          }
26          LOGD("打开的 can 设备：%d",sockfd);
27          ret = write(sockfd, &frame, sizeof(frame));
28          if (ret == sizeof(frame)) {
29              LOGD("CAN 数据发送成功");
30
31          } else {
32              LOGE("发送数据失败错误码: %s", strerror(errno));
33          }
34          env->ReleaseIntArrayElements(data, ptr, 0);
```

```
35
36          return 0;
37      }
```

上面代码第 4-7 行首先检查套接字的状态，如果 sockfd 小于 0 则打印打开 can 设备失败。

第 9 行代码使用 GetArrayLength() 获取传入 jintArray data 的长度 length。

第 10 行代码使用 GetIntArrayElements() 获取 data 数组的指针 ptr，并传入 JNI_FALSE 表示不复制数据。

第 19 行设置 CAN 帧的数据长度 frame.can_dlc 为 length - 1，表示除去帧 ID 之外的数据字节数。

第 20 行将 ptr[0] 设置为 CAN 帧的 ID frame.can_id。

第 22-25 行循环填充 CAN 帧的数据部分 frame.data，从 ptr[1] 开始。

第 27 行使用 write() 函数向 CAN 套接字 sockfd 写入数据，数据内容为 frame 结构体，大小为 sizeof(frame) 字节。

将 write() 的返回值保存在 ret 中，表示实际写入的字节数。

第 28-33 行处理发送结果，如果 write() 成功写入整个 frame 结构体的大小，记录信息日志表示 CAN 数据发送成功。如果写入失败，记录错误日志并输出具体的错误信息（使用 strerror(errno) 获取）。

现在本地方法里还有一个 Can 设备读取数据的本地方法 Java_com_example_cantest_CanUtil_recvdata，具体代码如下所示：

```
1   extern "C"
2   JNIEXPORT jintArray JNICALL
3   Java_com_example_cantest_CanUtil_recvdata(JNIEnv *env, jclass clazz){
4       if(sockfd < 0){
5           LOGE("sendData: 打开 can 设备失败");
6           return nullptr;
7       }
8       LOGD("准备读取 can 数据");
9
10      //接收数据
11      ret = read(sockfd,&frame,sizeof(struct can_frame));
12      if (ret < 0)
13      {
14          LOGE("接收数据失败： %s", strerror(errno));
15          return nullptr;
16      }
```

```
17          //检验是否接收到数据帧
18          if (frame.can_id & CAN_ERR_FLAG){
19              LOGE("接收到错误帧：%s", strerror(errno));
20              return nullptr;
21
22          }
23          //校验帧格式
24          if(frame.can_id & CAN_EFF_FLAG){
25              LOGD("扩展帧 <0x%08x>", frame.can_id & CAN_EFF_MASK);
26          }
27          else{
28              LOGD("标准帧 <0x%03x>", frame.can_id & CAN_SFF_MASK);
29          }
30          //校验帧类型，数据帧还是远程帧
31          if(frame.can_id & CAN_RTR_FLAG){
32              LOGD("这是远程帧");
33              return nullptr;
34          }
35          //打印数据长度
36          LOGD("帧的数据长度[%d]",frame.can_dlc);
37          //创建包含帧 ID 和数据的数据
38          jintArray   result = env->NewIntArray(frame.can_dlc +1);
39          if(result == nullptr){
40              LOGE("分配 int 数组失败");
41              return nullptr;
42          }
43
44          jint buffer[frame.can_dlc +1];
45          buffer[0] = frame.can_id;   //帧 ID
46          for(int i=0;i<frame.can_dlc;++i){
47              buffer[i+1] = frame.data[i];
48          }
49
50          //将 C 数组复制到 Java 数组中
51          env->SetIntArrayRegion(result, 0, frame.can_dlc + 1, buffer);
52
53          return result;
54      }
```

上面代码用于从 CAN 设备接收数据并将接收到的数据返回给 Java 层。

第 4-7 行代码首先检查套接字状态，如果 sockfd 是否小于 0，则记录错误日志并返回空指针（nullptr），表示 CAN 设备打开失败。

第 11 行代码使用 read() 函数从 CAN 套接字 sockfd 中读取数据，数据大小为 sizeof(struct can_frame) 字节。将实际读取的字节数保存在 ret 中。

第 12-16 行如果 read() 返回值小于 0，表示读取数据失败。记录错误日志并返回空指针。

第 18-22 行根据帧 ID 的标志位判断帧的格式是扩展帧还是标准帧，并记录相应的信息日志。

第 31-34 行检查帧 ID 的标志位，判断帧类型是数据帧还是远程帧。如果是远程帧，记录信息日志并返回空指针。

第 36 行记录信息日志，打印接收到的帧的数据长度 frame.can_dlc。

第 38-51 行创建并填充返回的数据数组。使用 env->NewIntArray() 创建一个长度为 frame.can_dlc + 1 的 jintArray，用于存储帧 ID 和数据。如果创建失败，记录错误日志并返回空指针。将帧 ID 和数据填充到 buffer 数组中，然后使用 env->SetIntArrayRegion() 将 buffer 数组复制到 result 中。

第 53 行将帧 ID 和数据填充到 buffer 数组中，然后使用 env->SetIntArrayRegion() 将 buffer 数组复制到 result 中。

到此，核心功能代码分析完毕。

第 5 章 安卓 PWM 调速风扇 APP 开发

5.1 前置说明

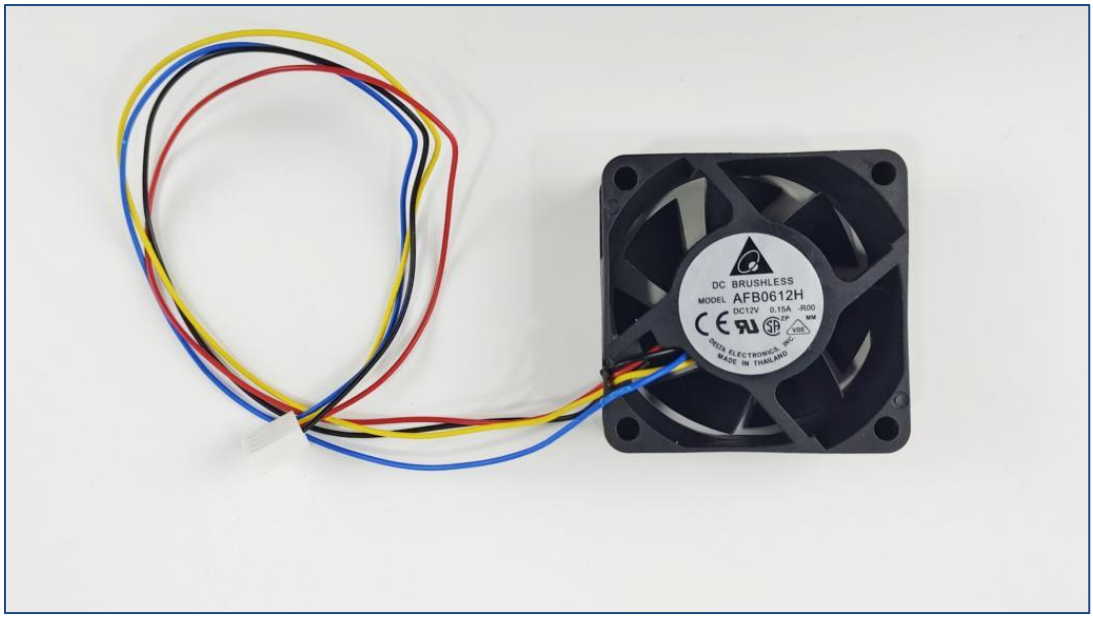
由于 iTOP-RK3588 底板的风扇接口为两线接口，并且考虑到部分客户要使用调速风扇，本章节将编写一个 Android 示例，使用 PWM 控制风扇速度。鉴于 iTOP-RK3568 底板的风扇接口为四线，因此我们选择使用 iTOP-RK3568 底板来进行验证。在 iTOP-RK3568 底板上，我们烧写了 Android 11 镜像，并且 Android 11 源码已经默认配置了风扇。

Android PWM 调速风扇源码可以在“iTOP-3588 开发板\02_【iTOP-RK3588 开发板】开发资料\14_应用程序测试例程\01_AndroidStudioExample\04_pwmfan_test”获取，如何使用 Android Studio 加载 app 源码可以参考手册“13【北京迅为】itop-3588 开发板 android 系统和应用开发

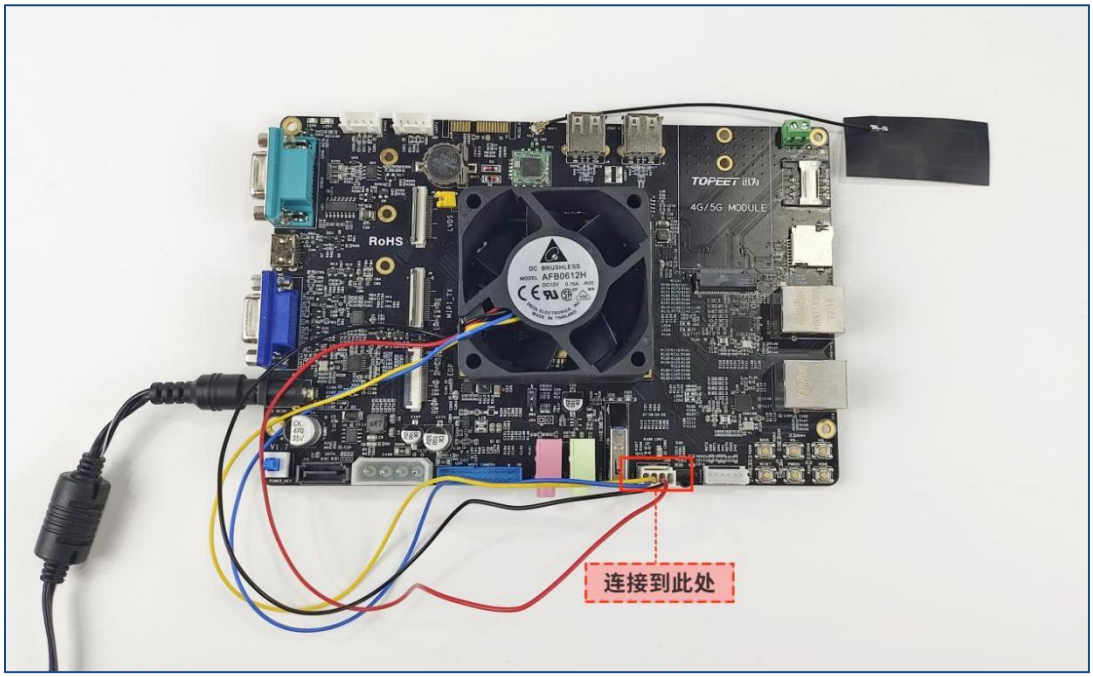
手册 v1.1”。

5. 2 硬件连接

风扇模块如下所示，黑线是负，红线是正，黄线是测速，蓝线是温控。



风扇接口位置如下图所示，底板 U56 插针的 1-4 引脚依次连接风扇模块的黑红黄蓝线。



5.3 Android 源码配置

迅为提供的 iTOP-RK3568 配套的 Android11 源码默认配置了调速风扇，在进行测试之前，请按照以下命令顺序修改风扇文件接口的访问权限，否则可能由于权限问题导致错误。

```
adb root          # 以 root 用户的权限运行 adb
adb remount       # 重新挂载 Android 设备的文件系统，使系统分区变为可读写状态
adb shell         # 进入 adb shell，方便与 Android 设备交互

chmod 777  /sys/class/hwmon/hwmon1/pwm1 # 修改风扇文件接口访问权限
```

```
PS F:\> adb devices
List of devices attached
68b639b85b2b1018        device

PS F:\>
PS F:\> adb root
restarting adbd as root
PS F:\> adb remount
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
Using overlayfs for /system
Using overlayfs for /vendor
Using overlayfs for /odm
Using overlayfs for /product
Using overlayfs for /system_ext
Now reboot your device for settings to take effect
remount succeeded
PS F:\>
```

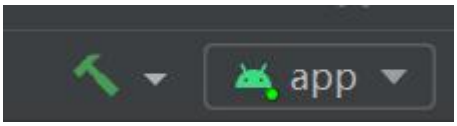
```
PS F:\> adb shell
adb server is out of date.  killing...
* daemon started successfully *
rk3568_r:/ # chmod 777  /sys/class/hwmon/hwmon1/pwm1
chmod 777  /sys/class/hwmon/hwmon1/pwm1
rk3568_r:/ #
```

修改完成之后，再运行 APP 进行测试。

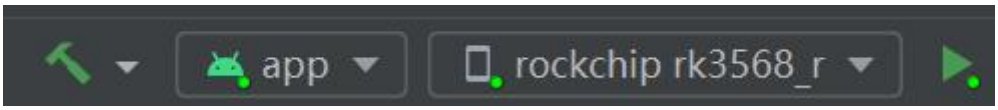
5.4 APP 运行测试

将网盘上的安卓工程文件复制到 Windows 电脑上。确保工程路径中使用英文字符，不包含中文。接着，启动 Android Studio，点击“Open”按钮选择应用工程文件夹，然后点击“OK”。由于下载 Gradle 和各种 Jar 包可能需要一段时间，Android Studio 加载工程可能会耗时较长

甚至编译失败。如果编译失败，可以尝试多次点击工具栏上的绿色“小锤子”按钮重新编译，“小锤子”按钮如下图所示：



一旦源代码成功编译，选择目标设备后点击工具栏上的绿色三角形按钮即可运行应用程序，如下图所示：



如果 APP 运行成功，在开发板连接的屏幕上显示 App 界面，如下图所示：

通过拖动滑块来调节风扇转速，向右拖动滑块会使风扇速度增加。

到此，安卓 PWM 调速风扇 App 的操作步骤就完成了。

5.5 核心功能代码解析

本小节主要对安卓 PWM 调速风扇 APP 的例程源码的核心功能进行解析，完整的代码可以通过本章第一小节描述的方式获取。

首先看一下跟 PWM 相关的 JNI 本地方法，方法声明在 pwmfan_test\app\src\main\java\com\example\pwmfantest\pwmfan.java 文件中，具体代码如下

所示：

```
1  public class pwmfan
2  {
3          public static native int changeThermalCoolingCurState(int i);
4  }
5
```

上面的代码声明了一个静态的 native 方法 **changeThermalCoolingCurState**，用于改变散热风扇的当前状态，接着我们先来看 changeThermalCoolingCurState 方法的实现代码，具体代码如下所示：

```
1   extern "C"
2   JNIEXPORT jint JNICALL
3   Java_com_example_pwmfantest_pwmfan_changeThermalCoolingCurState(JNIEnv *env, jclass clazz, jint i) {
4       std::ofstream file("/sys/class/hwmon/hwmon1/pwm1");
5       if (file.is_open()) {
6           file << i;
7           file.close();
8           return 0;
9       }
10      return -1;
11  }
```

上面代码声明一个 JNI 函数，用于在 Java 代码中调用改动散热风扇状态的方法。

第 4 行代码使用 ofstream 打开/sys/class/hwmon/hwmon1/pwm1 文件，如果您使用其他的 pwm，/sys/class/hwmon/hwmon1/pwm1 文件可以根据情况进行修改。比如 iTOP_3588 开发板的蜂鸣器也可以使用此源码例程，只需将节点修改为“/sys/class/pwm/pwmchip2/pwm0/”即可。

第 5-8 行代码，如果成功打开文件，写入传入的状态值 i，然后关闭文件。

布局文件 pwmfan_test\app\src\main\res\layout\activity_main.xml 中放置了一个文本控件 TextView 和 SeekBar 控件，SeekBar 用于显示一个可拖动的滑块，用户可以通过拖动滑块来调整数值。在 XML 布局文件中添加 SeekBar 控件，如下所示：

```
1   <SeekBar
2       android:id="@+id/seek_bar"
3       android:layout_width="match_parent"
4       android:layout_height="wrap_content"
5       android:progress="80"
6       app:layout_constraintBottom_toBottomOf="parent"
7       app:layout_constraintEnd_toEndOf="parent"
8       app:layout_constraintHorizontal_bias="0.4"
9       app:layout_constraintStart_toStartOf="parent"
10      app:layout_constraintTop_toBottomOf="@+id/speed_set"
```

```
11          app:layout_constraintVertical_bias="0.05" />
```

接下来在 03_pwmfan_test\app\src\main\java\com\example\pwmfantest\MainActivity.java 文件中设置监听器，如下所示：

```
1   SeekBar seekbar = findViewById(R.id.seek_bar);
2   seekbar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
3               @Override
4               public void onProgressChanged(SeekBar seekBar, int i, boolean b) {
5                   //当 seekbar 进度改变时调用
6                  //可在此处处理进度改变的逻辑
7                  // 调用 changeThermalCoolingCurState 方法，将当前进度值 i 传入
8                   int result = changeThermalCoolingCurState(i);
9                  // 如果返回值为 0，表示设置成功
10                  if (result == 0) {
11                   // 更新 textview 显示风扇转速设置
12                      textview.setText("风扇转速设置：  " + i);
13                  }
14              }
15
16              @Override
17              public void onStartTrackingTouch(SeekBar seekBar) {
18                  //当用户开始拖动 seekbar 时调用
19
20              }
21
22              @Override
23              public void onStopTrackingTouch(SeekBar seekBar) {
24                  //当用户停止拖动 seekbar 时调用
25
26              }
27          });
```

到此，核心功能代码分析完毕。

第 6 章 安卓音视频编解码器 MediaCodec

6.1 前言

在 Android 平台上，我们经常需要处理音视频数据，比如播放视频、录制音频等。音视频数据的原始形式十分庞大，使得它们在存储和传输过程中面临重大挑战。为了应对这个问题，必须采用压缩技术对音视频数据进行处理，这种技术被称为音视频编码。编码的主要目标是在尽可能减少图像或音频信息丢失的情况下实现最大的压缩效果，而解码则旨在尽可能还原原始图像或声音信息。编解码技术的核心作用在于方便数据的高效传输和有效存储。比如直播，小视频，视频会议，一对一通话，视频录制，播放器等项目都大量使用了编解码技术。音视频编解码格式非常多（h264，h265，vp8，vp9，aac...）,实现每种编解码都需要引入外部库，导致项目臃肿 运行性能差。

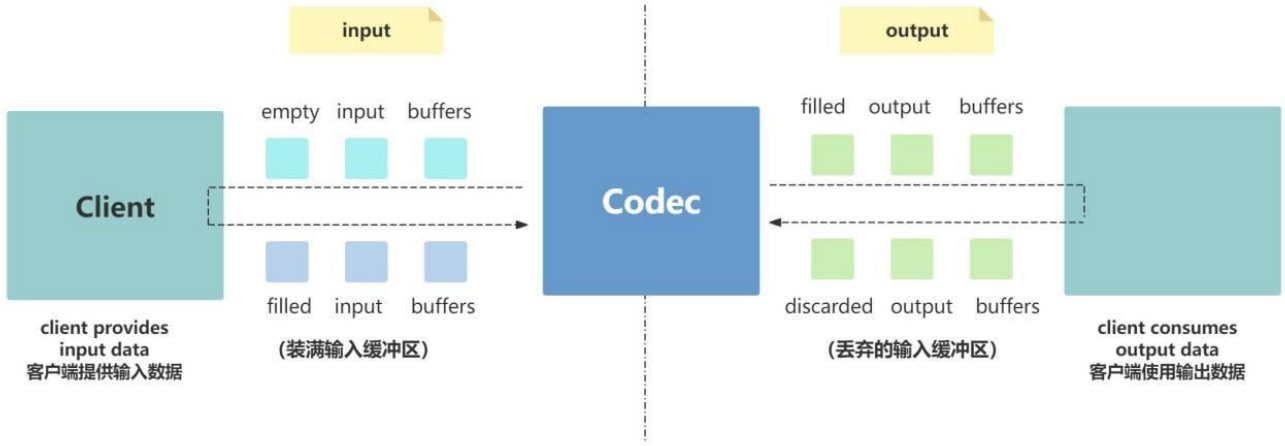
为了高效处理这些数据，Android 提供了 MediaCodec 类。在 Android 开发中，MediaCodec 扮演着关键角色，它是连接多媒体数据与硬件编解码器的桥梁。通过 Mediacodec，开发者能够轻松实现对音频和视频数据的实时处理，包括可以解码和编码操作，以确保用户获得流畅的媒体体验。

MediaCodec 的强大之处在于它广泛支持多种音视频格式，如 H.264、H.265 和 AAC 等。这意味着开发者可以利用设备内置的硬件解码器或编码器，对这些格式的媒体数据进行高效处理，显著提升应用的性能和响应速度。

6.2 MediaCodec 概述

MediaCodec 是 Android 提供的一个音视频编解码器，它允许应用程序对音频和视频数据进行编码（压缩）和解码（解压缩）。通过 MediaCodec，我们可以实现音视频的播放、录制、转码等功能。

Android 官网对 MediaCodec 工作原理如下图所示。

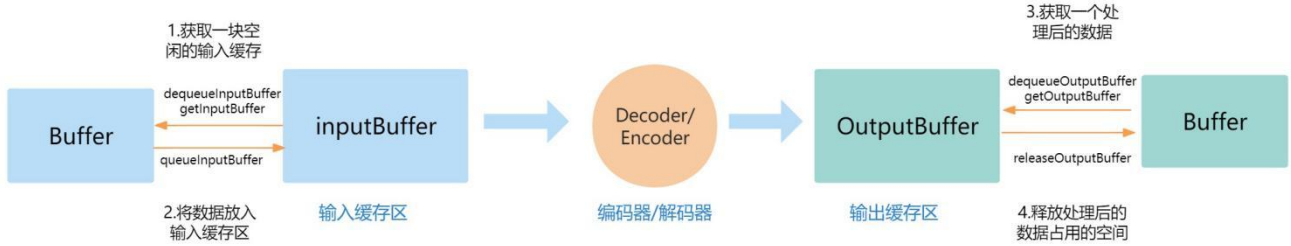


上图是 Android 官网对 MediaCodec 工作原理的描述，编解码器（codec）可以看做是一种处理输入数据以及生成输出数据的工具。MediaCodec 运行过程中的数据传输，可以看成两路数据流：Input 流、Output 流。 Input 流表示客户端输入的待编码或解码的数据。Output 流表示客户端输出的已编码或解码的数据。

在上图工作流的核心是缓冲区，MediaCodec 编解码都是通过缓冲区进行数据处理的。MediaCodec 使用两组 Buffer 队列，通过同步或异步方式处理两路数据流。包含一组 InputBuffer(格式 ByteBuffer)的 InputBufferQueue 和一组 OutputBuffer(格式 ByteBuffer)的 OutputBufferQueue

MediaCodec 可以类比为一个工厂的生产线，输入缓冲区就像是原材料仓库，输出缓冲区就像是成品仓库。原材料（待编解码的数据）首先被送入原材料仓库（输入缓冲区），然后工厂（即 MediaCodec）根据生产需求，从原材料仓库中取出原材料进行加工处理（即编解码操作）。加工处理完成之后，成品（即编解码后的数据）被放入成品仓库（即输出缓冲区）。最后，消费者（即应用程序）从成品仓库中取出成品进行使用。

使用 MediaCodec 的编解码流程如下图所示：



1）Client 也就是调用者从 MediaCodec 的 InputBufferQueue 中获取空的 InputBuffer，写入要编码或解码的数据，提交给 MediaCodec

2）MediaCodec 收到数据进行处理，处理完毕后，将其转存到 OutputBufferQueue 中空的

OutputBuffer(拷贝过程由 mediacodec 内部完成，调用者只需要关注 OutputBufferQueue 中空闲 Buffer 索引的变化)，同时释放 InputBuffer，将它重新放回到 InputBufferQueue

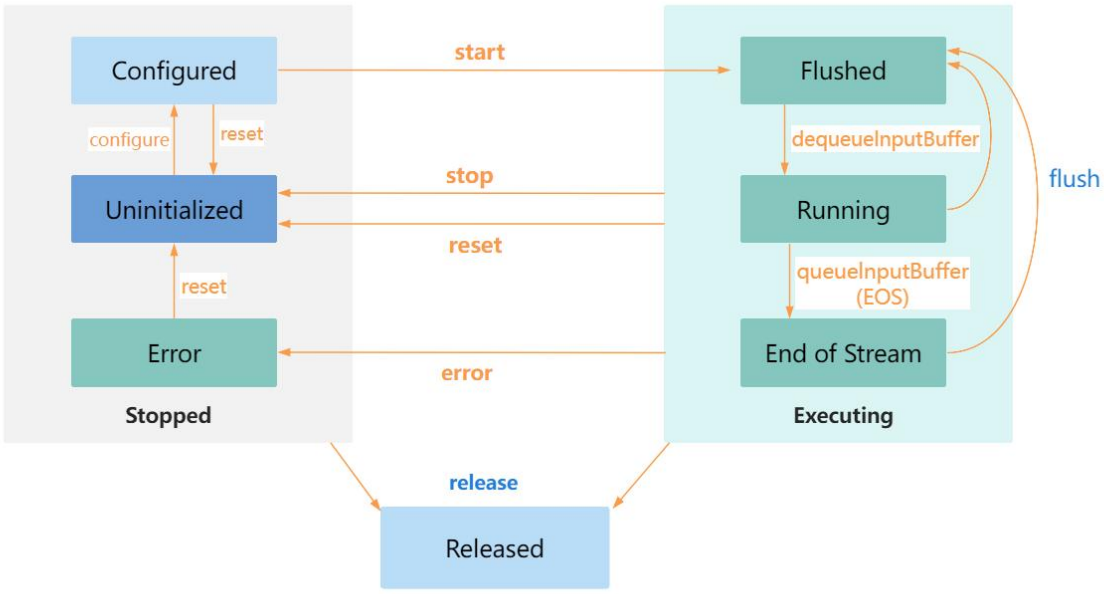
3）Client 从 MediaCodec 的 OutputBufferQueue 中获取到一个空的 OutputBuffer，读取数据进行使用，待 Client 处理完数据后，释放 OutputBuffer 并将其放回 OutputBufferQueue

上面讲解的 MediaCodec 同步模式，在实际使用中，MediaCodec 通常会有多个输入缓冲区和输出缓冲区，也就是异步模式。通过多个缓冲区，MediaCodec 可以在一个缓冲区正在被处理（比如说正在进行编解码操作）的同时，另一个缓冲区可以被填充数据，这样可以提高效率。本章节只讨论同步模式，异步模式大家可以自行学习。

接下来我们学下下 MediaCodec 的生命周期。

6.3 MediaCodec 生命周期

MediaCodec 状态流转图如下图所示：



MediaCodec 的生命周期大致分为三类: Sotpped、Executing、Released

(1) Stopped（停止）：包含了三个小状态：Error、Uninitialized、Configured

- 在调用 MediaCodec.createEncoderByType 创建编解码器类型时，编解码器处于未初始化（Uninitialized）状态。
- 调用 encoderCodec.configure() 后，编解码器进入配置（Configured）状态。

(2) Executing（执行）：同样包含 3 个小状态：Flushed、Running、End of Stream

❖ 调用 encoderCodec.start() 后，编解码器进入执行（Running）状态。

❖ 在执行状态时，可以通过调用 flush() 方法返回到 Flushed 子状态。

❖ 最后，当解码/编码结束时，进入 End of Stream（EOF）状态。

(3) Released（释放）:

❖ 调用 release() 方法释放完整的编解码器资源，进入最终的释放（Released）状态。

这些状态包括了从编解码器创建到配置、执行到最终释放资源的完整生命周期。

6. 4 同步模式解码实例

```
1   private fun decodeToBitmap() {
2       // 创建并配置 MediaExtractor 对象，用于提取媒体数据
3       val mediaExtractor = MediaExtractor()
4       // 打开原始资源文件 (R.raw.h264_720p) 并设置为 MediaExtractor 的数据源
5       resources.openRawResourceFd(R.raw.h264_720p).use {
6           mediaExtractor.setDataSource(it)
7       }
8       // 选择要提取的音视频轨道，这里假设视频轨道的索引为 0
9       val videoTrackIndex = 0
10      mediaExtractor.selectTrack(videoTrackIndex)
11      // 获取选定轨道的格式信息
12      val videoFormat = mediaExtractor.getTrackFormat(videoTrackIndex)
13
14      // 创建并配置 MediaCodec 对象，用于解码视频数据
15      // 获取可以用于解码指定格式的编解码器名称
16      val codecList = MediaCodecList(MediaCodecList.REGULAR_CODECS)
17      val codecName = codecList.findDecoderForFormat(videoFormat)
18      // 根据编解码器名称创建 MediaCodec 实例
19      val codec = MediaCodec.createByCodecName(codecName)
20      // 配置 MediaCodec，指定解码格式和输出到一个空的 Surface 以便获取解码后的图像数据
21      codec.configure(videoFormat, null, null, 0)
22
23      // 开始解码过程
24      // 获取视频格式中的最大输入缓冲区大小
25      val maxInputSize = videoFormat.getInteger(MediaFormat.KEY_MAX_INPUT_SIZE)
26      // 分配用于存储视频数据的 ByteBuffer
27      val inputBuffer = ByteBuffer.allocate(maxInputSize)
28      // 用于存储解码输出的缓冲区信息
29      val bufferInfo = MediaCodec.BufferInfo()
```

```
30          // 设置解码的超时时间为 10ms
31          val timeoutUs = 10000L // 10ms
32          // 标记是否读取完输入数据
33          var inputEnd = false
34          // 标记是否解码完成
35          var outputEnd = false
36
37          // 启动 MediaCodec 开始解码
38          codec.start()
39          while (!outputEnd && !stopDecoding) {
40              // 从 MediaExtractor 中读取输入数据到 inputBuffer，并更新 bufferInfo
41              val isExtractorReadEnd =
42                  getInputBufferFromExtractor(mediaExtractor, inputBuffer, bufferInfo)
43              if (isExtractorReadEnd) {
44                  // 如果读取到数据末尾，标记输入结束
45                  inputEnd = true
46              }
47
48              // 获取 MediaCodec 的输入缓冲区 ID，并将数据填充到缓冲区中
49              // timeoutUs 为 -1L 表示无限等待
50              val inputBufferId = codec.dequeueInputBuffer(-1L)
51              if (inputBufferId >= 0) {
52                  if (inputEnd) {
53                      // 如果输入结束，则将结束流标记添加到输入队列中
54                      codec.queueInputBuffer(inputBufferId, 0, 0, 0, BUFFER_FLAG_END_OF_STREAM)
55                  } else {
56                      // 获取输入缓冲区并将数据从 inputBuffer 中复制到输入缓冲区
57                      val codecInputBuffer = codec.getInputBuffer(inputBufferId)
58                      codecInputBuffer!!.put(inputBuffer)
59                      codec.queueInputBuffer(
60                          inputBufferId,
61                          0,
62                          bufferInfo.size,
63                          bufferInfo.presentationTimeUs,
64                          0
65                      )
66                  }
67              }
68
69              // 从 MediaCodec 中获取解码后的输出缓冲区，并将其渲染到 ImageView 中
70              // NOTE! dequeueOutputBuffer 使用 -1L 可能会阻塞，因此在这里等待 10ms
71              val outputBufferId = codec.dequeueOutputBuffer(bufferInfo, timeoutUs)
72              if (outputBufferId >= 0) {
73                  if (bufferInfo.flags and BUFFER_FLAG_END_OF_STREAM != 0) {
```

```
                        // 如果解码完成标志被设置，则结束解码
                        outputEnd = true
                    }
                    if (bufferInfo.size > 0) {
                        // 从解码器中获取解码后的 YUV 图像
                        val outputImage = codec.getOutputImage(outputBufferId)
                        // 将 YUV 图像转换为 Bitmap 以便渲染到 ImageView
                        val bitmap = yuvImage2Bitmap(outputImage!!)
                        // 将 Bitmap 发送到主线程中更新 ImageView
                        imageView.post {
                            imageView.setImageBitmap(bitmap)
                        }
                        // 渲染完成后记得释放输出缓冲区
                        codec.releaseOutputBuffer(outputBufferId, false)
                        // 为了模拟 30fps 的视频播放，睡眠 30ms
                        Thread.sleep(30)
                    }
                }

                // 处理下一个数据帧
                mediaExtractor.advance()
            }

            // 释放 MediaExtractor 和 MediaCodec 对象
            mediaExtractor.release()
            codec.stop()
            codec.release()
        }
```

在上述的同步模式示例代码中，功能实现如下所示：

1 创建并配置媒体提取器（MediaExtractor）：媒体提取器用于从媒体文件中提取音频和视频数据。这里，它从资源文件 h264_720p 中提取数据。

2 选择要处理的轨道：这里选择的是视频轨道，其索引为 0。

3 获取视频格式：通过 getTrackFormat 方法获取视频轨道的格式。

4 创建并配置媒体编解码器（MediaCodec）：首先，通过 MediaCodecList 获取适合视频格式的解码器名称，5 然后通过该名称创建解码器。接着，使用视频格式和空的 Surface（这样可以更容易地获取解码后的位图）来配置解码器。

6 开始解码：首先，从视频格式中获取最大输入大小，并创建一个相应大小的 ByteBuffer。然

后，创建一个 MediaCodec.BufferInfo 对象，用于保存解码后的数据信息。最后，定义两个标志位，分别表示输入和输出是否结束。

7 循环解码：在循环中，首先从媒体提取器中获取输入缓冲区的数据。然后，从解码器中获取输入缓冲区，并将提取器中的数据填充到解码器的输入缓冲区中。接着，从解码器中获取输出缓冲区，并将其转换为位图，然后在主线程中更新 ImageView。最后，释放输出缓冲区，并使线程休眠 30 毫秒，以模拟 30fps 的帧率。

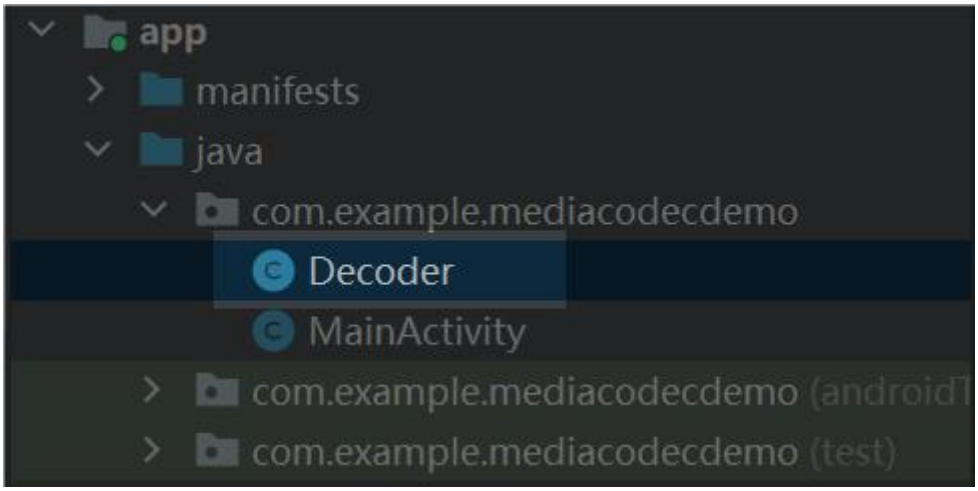
8 结束解码：在循环结束后，释放媒体提取器和解码器。

6.5 MediaCodec demo

在前面的几个小节中，我们学习了 MediaCodec 的基础知识，本小节将编写一个简易视频播放器，其中会用到 MediaCodec 的知识。

Android 音频编解码 demo 源码可以在“iTOP-3588 开发板\02_【iTOP-RK3588 开发板】开发资料\14_应用程序测试例程\01_AndroidStudioExample\05_mediacodecdemo”获取，如何使用 Android Studio 加载 app 源码可以参考手册“13【北京迅为】itop-3588 开发板 android 系统和应用开发手册 v1.1”。

首先新建一个 Android 空工程，然后在工程下新建一个 Decoder 类，如下图所示：



Decoder 类的主要功能是从 MP4 文件中解码 H.264 编码的视频数据，将视频帧渲染到一个 Surface 上，并提供了视频画面尺寸变化的回调接口。以下是对 Decoder 类的详细解释。

首先 Decoder 类里面定义了下面代码的成员变量。

```
1     public class Decoder {
2         // TAG 用于在日志中标识当前类的简单名称
3         private static final String TAG = Decoder.class.getSimpleName();
4         // 媒体数据提取器，用于从媒体文件中提取音视频数据
5         private final MediaExtractor mMediaExtractor = new MediaExtractor();
6         // 媒体解码器，用于解码音视频数据
7         private MediaCodec mMediaCodec;
8         // 媒体格式，包含了音视频流的格式信息
9         private MediaFormat mMediaFormat;
10        // MP4 解码线程，负责将从 MP4 文件中读取的数据填充到解码器的输入缓冲区
11        private DecoderMP4Thread mDecodeMp4Thread;
12        // 渲染线程，负责从解码器获取解码后的视频帧数据并渲染到 Surface 上
13        private RenderThread mRenderThread;
14        // 视频画面尺寸变化监听器，用于监听视频画面尺寸的改变事件
15    private OnSizeChangedListener mOnSizeChangedListener;
16    ........
17    }
```

然后定义了接口 OnSizeChangedListener ，OnSizeChangedListener 用于监听视频画面尺寸的变化，提供 onChanged 方法以便在尺寸变化时得到通知。

```
1     public class Decoder {
2     ........
3     public interface OnSizeChangedListener {
4         void onChanged(int width, int height);
5     }
6
7     ........
      }
```

设置画面尺寸变化监听器，以便在视频画面尺寸发生变化时得到通知，如下所示：

```
1     public class Decoder {
2     ........
3     public void setOnSizeChangedListener(OnSizeChangedListener onSizeChangedListener) {
4         this.mOnSizeChangedListener = onSizeChangedListener;
5     }
6
7     ........
      }
```

接下来定义了开始播放视频 play 方法，如下所示：

```
1     public class Decoder {
2     ........
3     public void play(AssetFileDescriptor mp4, Surface surface) {
```

```
4                 try {
5                     // 创建视频解码器
6                     mMediaCodec = MediaCodec.createDecoderByType("video/avc");
7                     // mMediaFormat = MediaFormat.createVideoFormat("video/avc", 1280, 720);
8                     // 设置媒体数据源为 MP4 文件
9                     mMediaExtractor.setDataSource(mp4);
10                    Log.d(TAG, "getTrackCount: " + mMediaExtractor.getTrackCount());
11                    // 遍历媒体文件的所有轨道，找到视频轨道并选择
12                    for (int i = 0; i < mMediaExtractor.getTrackCount(); i++) {
13                        MediaFormat format = mMediaExtractor.getTrackFormat(i);
14                        String mime = format.getString(MediaFormat.KEY_MIME);
15                        Log.d(TAG, "mime: " + mime);
16                        assert mime != null;
17                        if (mime.startsWith("video")) {
18                            Log.d(TAG, "found video");
19                            mMediaFormat = format;
20                            mMediaExtractor.selectTrack(i);
21                        }
22                    }
23                } catch (IOException e) {
24                    e.printStackTrace();
25                }
26                // 配置解码器，设置输出的 Surface
27                mMediaCodec.configure(mMediaFormat, surface, null, 0);
28                mMediaCodec.start();
29                // 启动 MP4 解码线程
30                mDecodeMp4Thread = new DecoderMP4Thread();
31                mDecodeMp4Thread.start();
32                //启动解码输出线程
33                mRenderThread = new RenderThread();
34                mRenderThread.start();
35            }
36
37        ........
38        }
```

上述代码中主要是初始化 MediaCodec 和 MediaExtractor，配置解码器，并启动解码和渲染线程。具体步骤如下所示：

1 创建一个 MediaCodec 解码器。

2 设置 MediaExtractor 的数据源为 MP4 文件。

3 遍历 MP4 文件中的轨道，找到视频轨道并选择。

4 配置 MediaCodec，设置输出到 Surface。

5 启动 DecoderMP4Thread 和 RenderThread 线程，分别处理解码数据和渲染视频帧

再接下来定义了关闭解码器 close 方法，如下所示：

```
1   public class Decoder {
2   ........
3
4   // 关闭解码器
5       public void close() {
6           try {
7               Log.d(TAG, "close start");
8               if (mMediaCodec != null) {
9                   // 停止解码线程和输出线程
10                  mDecodeMp4Thread.interrupt();
11                  mRenderThread.interrupt();
12                  try {
13                      mDecodeMp4Thread.join(2000);
14                      mRenderThread.join(200);
15                  } catch (InterruptedException e) {
16                      Log.e(TAG, "InterruptedException " + e);
17                  }
18                  boolean isAlive = mDecodeMp4Thread.isAlive();
19                  Log.d(TAG, "close end isAlive :" + isAlive);
20                  // 释放解码器资源
21                  mMediaCodec.stop();
22                  mMediaCodec.release();
23                  mMediaCodec = null;
24              }
25          } catch (IllegalStateException e) {
26              e.printStackTrace();
27          }
28  }
29  ........
30  }
```

上面的 stop 方法功能是停止解码线程和渲染线程，释放 MediaCodec 的资源。

再接下来是解码输入线程的 DecoderMP4Thread 方法，负责从 MP4 文件中读取数据并传递给 MediaCodec 进行解码，代码如下所示：

```
1   public class Decoder {
2   ........
3
4   // 解码输入线程，负责从 MP4 文件读取数据并传递给解码器解码
5       private class DecoderMP4Thread extends Thread {
6           long pts = 0;
7
```

```
8                   @Override
9                   public void run() {
10                      super.run();
11                      long starttime = 0;
12                      while (!interrupted()) {
13                          int inputIndex = mMediaCodec.dequeueInputBuffer(-1);
14                          if (inputIndex >= 0) {
15                              //从媒体提取器中读取数据填充解码器的输入缓冲区
16                              ByteBuffer byteBuffer = mMediaCodec.getInputBuffer(inputIndex);
17                              //读取一片或者一帧数据
18                              int sampSize = mMediaExtractor.readSampleData(byteBuffer, 0);
19                              if (sampSize >= 0) {
20                                  //读取时间戳
21                                  if (starttime == 0)
22                                      starttime = System.currentTimeMillis() * 1000;
23                                  long time = mMediaExtractor.getSampleTime();
24                                  // 将数据发送给解码器进行解码
25                                  mMediaCodec.queueInputBuffer(inputIndex, 0, sampSize, starttime + time,
26  0);
27                                  //读取一帧后必须调用，提取下一帧
28                                  mMediaExtractor.advance();
29                              } else {
30                                  // 媒体文件读取完毕，发送结束标志
31                                  mMediaCodec.queueInputBuffer(inputIndex, 0, 0, 0,
32  MediaCodec.BUFFER_FLAG_END_OF_STREAM);
33                              }
34                              // }
35                          }
36                      }
37                  }
38      }
39      ........
40      }
```

在上述代码的方法中，主要功能如下所述：

1 读取 MP4 文件中的数据到解码器的输入缓冲区。

2 将数据传递给解码器，并提取下一帧数据

3 当文件读取完毕时，发送结束留标志。

再接下来是解码输出线程 RenderThread 方法，负责从解码器获取解码后的视频数据并渲染，代码如下所示：

```
1       public class Decoder {
2       ........
```

```
3       // 解码输出线程，负责从解码器获取解码后的视频帧数据并渲染
4           private class RenderThread extends Thread {
5               private void vsleep(long pts) {
6                   // 根据帧的展示时间戳，休眠适当的时间
7                   if (pts > System.currentTimeMillis() * 1000) {
8                       try {
9                           Thread.sleep((pts - System.currentTimeMillis() * 1000) / 1000);
10                      } catch (InterruptedException e) {
11                          throw new RuntimeException(e);
12                      }
13                  }
14              }
15
16              @Override
17              public void run() {
18                  super.run();
19                  long starttime = System.currentTimeMillis();
20                  long count = 0;
21                  while (!interrupted()) {
22                      // 从解码器获取输出的视频帧数据
23                      MediaCodec.BufferInfo bufferInfo = new MediaCodec.BufferInfo();
24                      int outIndex = mMediaCodec.dequeueOutputBuffer(bufferInfo, -1);
25                      if (outIndex >= 0) {
26                          // 将帧数据渲染到 Surface 上
27                          mMediaCodec.releaseOutputBuffer(outIndex, bufferInfo.presentationTimeUs);
28                          count++;
29                          // 如果还没有到显示时间，则休眠一会儿。
30                          vsleep(bufferInfo.presentationTimeUs);
31                          long currenttime = System.currentTimeMillis();
32                          // 计算当前帧率
33                          if (currenttime - starttime > 1000) {
34                              double fps = (double) count / (double) (currenttime - starttime) * 1000;
35                              count = 0;
36                              starttime = currenttime;
37                              Log.d(TAG, "fps: " + fps);
38                          }
39
40                      } else if (outIndex == MediaCodec.INFO_OUTPUT_FORMAT_CHANGED) {
41                          Log.d(TAG, "INFO_OUTPUT_FORMAT_CHANGED ");
42  //                              mSurfaceView.post(() -> reCalcVideoRect(getWidth(), getHeight()));
43                          // 解码器输出格式发生变化，通知监听器更新视频尺寸
44                          MediaFormat format = mMediaCodec.getOutputFormat();
45                          int width = format.getInteger("width");
46                          int height = format.getInteger("height");
```

```
                    if (mOnSizeChangedListener != null) {
                        mOnSizeChangedListener.onChanged(width, height);
                    }
//                    if (onCompleteListener != null) {
//                    }
                }
            }
        }
    ........
    }
```

在上述方法中，主要功能如下所述：

1 vsleep 方法根据时间戳调整帧显示的时间

2 run 方法从 MediaCodec 中获取帧数据，渲染到 surface,并计算 FPS

3 监听解码器输出格式变化，更新视频尺寸。

在 MediaCodecDemo\app\src\main\res\layout\activity_main.xml 布局文件中添加 TextureView 控件，此控件用来显示准备好的 mp4 视频文件。

TextureView 是 Android 中用于显示图形数据的一个视图组件，它通常与 SurfaceTexture 一起使用，以实现高性能的图形渲染。TextureView 的使用场景主要包括与媒体播放和自定义图形渲染相关。TextureView 控件的使用方法主要包括在布局文件中定义 TextureView 控件，获取 TextureView 实例，并设置 SurfaceTextureListener 监听器。

首先，在布局文件中定义 TextureView 控件，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <LinearLayout
```

```
13                      android:id="@+id/videoPlayerBox"
14                      android:layout_gravity="center"
15                      android:gravity="center"
16                      android:background="#000000"
17                      android:layout_width="match_parent"
18                      android:layout_height="match_parent">
19                      <TextureView
20                          android:layout_gravity="center"
21                          android:id="@+id/textureView"
22                          android:layout_width="match_parent"
23                          android:layout_height="match_parent"
24                          />
25                  </LinearLayout>
26
27              </LinearLayout>
28
29
30  </androidx.constraintlayout.widget.ConstraintLayout>
```

接着，在 MediaCodecDemo\app\src\main\java\com\example\mediacodecdemo\MainActivity.java 中获取 TextureView 实例，代码如下所示：

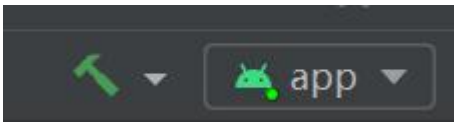
```
1
2       public class MainActivity extends AppCompatActivity {
3
4           private static final String TAG = MainActivity.class.getSimpleName();
5
6           private TextureView mTextureView;
7           private LinearLayout mVideoPlayBox;
8
9           @Override
10          protected void onCreate(Bundle savedInstanceState) {
11              super.onCreate(savedInstanceState);
12              // 设置 Activity 的布局
13              setContentView(R.layout.activity_main);
14              // 查找布局中的视频播放容器和 TextureView
15              mVideoPlayBox = findViewById(R.id.videoPlayerBox);
16              mTextureView = findViewById(R.id.textureView);
17              // 创建解码器实例
18              Decoder decoder = new Decoder();
19              // 监听视频画面尺寸的改变事件，通过视频的宽高比计算显示画布的大小，以保持正确的
20  宽高比
21              decoder.setOnSizeChangedListener(new Decoder.OnSizeChangedListener() {
```

```
22                      @Override
23                      public void onChanged(int width, int height) {
24                          // 在 UI 线程中执行调整大小的操作
25                          mVideoPlayBox.post(new Runnable() {
26                              @Override
27                              public void run() {
28                                  // 计算宽高比例，调整 TextureView 的大小
29                                  double scale1 = (double) mVideoPlayBox.getWidth() / width;
30                                  double scale2 = (double) mVideoPlayBox.getHeight() / height;
31                                  Log.d(TAG, "VideoPlayBox {width=" + mVideoPlayBox.getWidth() + ",height="
32  + mVideoPlayBox.getHeight() + "}");
33                                  double scale = Math.min(scale1, scale2);
34                                  ViewGroup.LayoutParams videoParams = mTextureView.getLayoutParams();
35                                  videoParams.width = (int) (width * scale);
36                                  videoParams.height = (int) (height * scale);
37                                  mTextureView.setLayoutParams(videoParams);
38                                  Log.d(TAG, "TextureView {width=" + videoParams.width + ",height=" +
39  videoParams.height + "}");
40                              }
41                          });
42
43                      }
44                  });
45
46                  // 当 TextureView（显示画布）准备就绪后，开始播放视频
47                  mTextureView.post(() -> {
48                      try {
49                          // 使用解码器播放指定资源的视频，并将视频渲染到 TextureView 上
50                          decoder.play(getAssets().openFd("input.mp4"), new
51  Surface(mTextureView.getSurfaceTexture()));
52                      } catch (IOException e) {
53                          // 捕获播放异常并抛出运行时异常
54                          throw new RuntimeException(e);
55                      }
56                  });
57
58              }
59
60
61  }
62
```

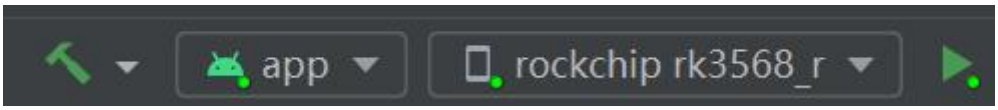
好，代码工程编写完成，接下来我们运行测试一下。

6.6 APP 运行测试

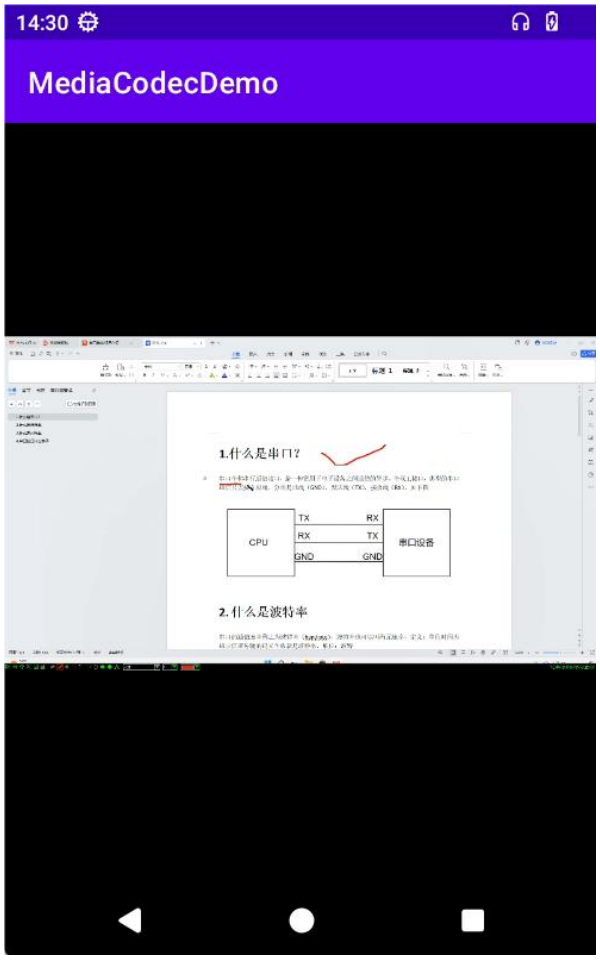
点击工具栏上的绿色“小锤子”按钮重新编译，“小锤子”按钮如下图所示：



一旦源代码成功编译，选择目标设备后点击工具栏上的绿色三角形按钮即可运行应用程序，如下图所示：



如果 APP 运行成功，在开发板连接的屏幕上显示 App 界面，如下图所示：



我们可以看到内置的 input.mp4 音频文件正在播放，且显示正常。到此，安卓音视频编解码 App 的操作步骤就完成了。