Android 双应用进程工控方案（二）

------Android 双应用进程 Demo 程序设计

英创公司

2017 年 12 月

Android 是移动设备的主流操作系统，近年来越来越多的工业领域的客户开始关注基于 Android 操作系统的设备在工控领域的应用。鉴于 Android 是基于 Linux 内核的事实，我们发展了一种以双应用进程为特色的 Android 工控应用方案，并在 ESM6802 工控主板上加以实现。具体说来，就是在 Linux 平台上运行一个直接操作硬件接口的控制通讯管理进程，为保证运行效率，该进程采用 C/C++语言编写（以下简称 C 进程或控制进程）；另一方面在 Android 平台采用标准 Java 语言编写一个人机界面进程（以下简称 Java 进程）。底层的控制进程并不依赖与上层的 Java 进程而独立运行，两个进程之间通过本地 IP 进行通讯，控制进程处于服务器侦听模式，Java 进程则为客户端模式。本方案的主要优点是客户可以直接继承已有的现成应用程序作为底层控制进程的基础，仅仅增加标准的 Socket 侦听功能，即可快速完成新的底层应用程序的设计。而界面的 Java 程序，由于不再涉及具体的工控硬件接口，属于单纯的 Android 程序，编程难度也大大降低。

设计 Android 双应用进程 Demo 程序的目的就是验证“双应用进程”Android 工控应用方案的可行性，同时起到一个抛砖引玉的作用。本设计文档将具体讲解设计思路，约束通信协议和接口。

一、总体描述

总体要求如下：

- 采用 C/S（客户端/服务端）模式，通过 socket 连接，通信需要自定义通信协议；
- 客户端使用 Java 语言开发，主要是人机交互，查询以及简单的设置功能；
- 服务端使用 C/C++语言开发，主要是各个功能模块的业务逻辑正常运行，以及接收处理客户端的人机交互请求；
- 服务端各个功能模块的业务逻辑部分始终正常运行，客户端连接与否，不影响各个模块

的工作；

- 客户端接入后，固定周期向服务端发出刷新数据请求，服务端响应后，客户端仅刷新有改变的 UI 部件。

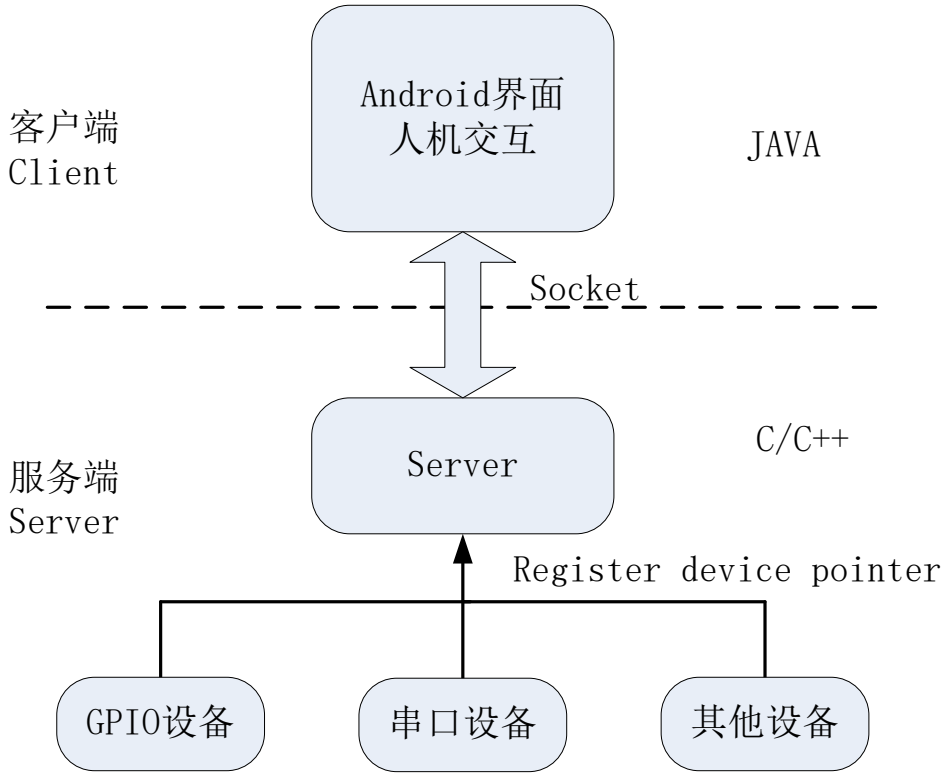

图 1 C/S 工作模式示意图

如图 1 所示，Android 开机后，服务端 C/C++程序自动运行，此时整个控制系统已经运转起来，各个功能模块下的设备都已进入正常工作模式。这里以 GPIO 和串口设备作为示例，其中 GPIO 模拟开关状态变量，串口模拟通信设备。server 模块监听 client 的连接，当有 client 接入后，响应 client 的请求，将 GPIO 状态、串口的数据统计信息，按照自定义的通信协议封装成协议包，通过 socket 发送给 client。如果没有 client 接入，或者 client 接入断开，各个功能模块依然正常工作。**也就是说，服务端 C/C++程序完全可以在没有客户端的情况下，自动运转各个功能模块；客户端的接入，主要是为了方便人为监控**。

二、模拟业务描述

在上一节的描述下，我们模拟某个控制系统（控制进程）的实际控制设备如下：

- **设备 1：**两个 GPIO，其中 GPIO0 作为警报输入，GPIO1 作为消防输出。控制进

程启动后一直监听 GPIO0，当 GPIO0 输入高电平时，控制进程设置 GPIO1 输出低电平；当 GPIO0 输入低电平时，GPIO1 输出高电平。

- **设备 2：**1 个串口设备，控制进程开启后，串口设备周期性自动发送固定字符串，接收线程一直开启。

在 Android 的人机界面进程（Java 进程）中，需要做的有：

- 查询并显示 GPIO0 和 GPIO1 的状态；
- 查询并显示串口设备的发送/接收计数值、底层串口的自动发送周期值
- 设置底层串口的自动发送周期
- 清零底层串口的发送/接收计数值

三、自定义通信协议

客户端与服务端之间通过 socket 连接通信，由于 socket 流传输是没有边界概念的，可能存在“分包”或“黏包”的情况，这就要求用户自定义通信协议，用以从 socket 字节流中提取出完整的通信包，解析此通信包，完成通信。

根据模拟业务描述，在此 Demo 程序中，采用下图所示的通信包结构：

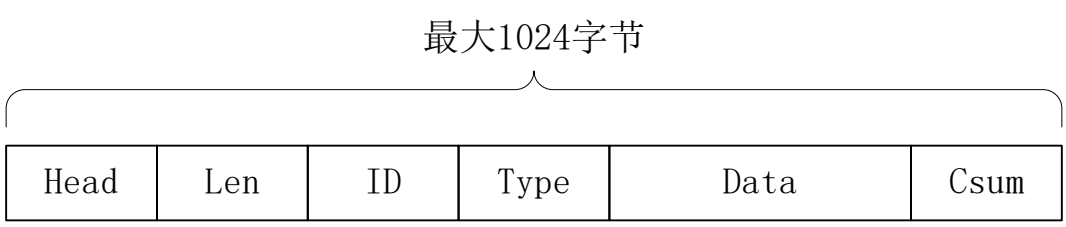

图 2 自定义通信协议包

Head：包头，2 字节，固定 0xFF0x02；

Len：包剩余（ID+Type+Data+Csum）长度，4 字节，范围 0~1018（1024-2-4）；

ID：设备标号，1 字节，表明此包来自于客户端还是服务端（可以不用）

‘S’ -- 服务端；

‘C’ -- 客户端；

Type：数据类型，1 字节，表明此包的具体指令目的

‘q’-- 来自客户端，查询所有参数（gpio 状态、串口发送/接收字节数、自动发送周期）；

‘r’-- 来自服务端，应答客户端‘q’查询；

‘s’-- 来自客户端，设置底层串口自动发送的周期；

‘c’-- 来自客户端，底层串口发送接收计数清零；

Data：数据内容，长度不定字节，长度范围 0~1015 字节

Type=‘q’时，data 没有意义，长度可以为 0；

Type=‘r’时，data 长度 14 字节，gpio0 状态（1 字节）+gpio1 状态（1 字节）+串口 rx 计数（4 字节）+ 串口 tx 计数（4 字节）+串口自动发送周期（4 字节）；

Type=‘s’时，data 长度 4 字节，表示设置底层串口自动发送的周期；

Type=‘c’时，data 没有意义，长度可以为 0；

Csum：校验和，1 字节，从 ID 到 Data 结束的校验和（反码），接收端 ID+Type+data+Csum=0 即为正确接收。

四、服务端 C/C++程序

服务端 C/C++程序是“双应用进程”模式中的控制进程，其主要功能有两个方面：一是能够自动运转各个功能模块，控制完成主要的业务逻辑；二是监听客户端连接，解析响应客户端请求。

4.1 自动运转各个功能模块

在此 demo 程序中，功能模块主要是 gpio 和串口，gpio 模拟开关量，串口模拟通信设备。因此，创建 MyGPIO 和 MySerial 两个类。下面简单介绍下这两个类的 public 接口。

```
class MyGPIO{
public:
    MyGPIO();
    ~MyGPIO();
    /**
     * init gpio device , open /dev/esm6800_gpio;
     * return 0 if success; -1 failed;
     */
    int init();

    /**
     * start a new thread to run gpio working task
     * @return
     */
    int run();

    /**
     * stop gpio working task
     * @return
     */
    void stop();

    /**
     * read the state of gpio-index, no matter whether gpiox is input or not
     * @param index gpio index 0~31
     * @return -1 -- failed; 0 -- low; 1 -- high
     */
    int getGPIOStateByIndex(int index);


private:
    int m_GPIOFd;
    int m_stopFlag;
    //bitx of m_pinDir indecate gpiox is input(0) or output(1);
    unsigned int m_pinDir;
    //only when bitx in m_pinDir is input, the bitx of m_pinInputState is valid,
    unsigned int m_pinInputState;
    //only when bitx in m_pinDir is output, the bitx of m_pinOutputState is valid,
    unsigned int m_pinOutputState;

    pthread_t m_workingThread;

    int GPIO_OutEnable(unsigned int dwEnBits);
    int GPIO_OutDisable(unsigned int dwDisBits);
    int GPIO_OpenDrainEnable(unsigned int dwODBits);
    int GPIO_OutSet(unsigned int dwSetBits);
    int GPIO_OutClear(unsigned int dwClearBits);
    int GPIO_PinState(unsigned int* pPinState);


    static int workingThreadFunc(void* lparam);

};
```

图 3、MyGPIO 类

MyGPIO 的 init 接口主要是打开/dev/esm6800_gpio 这个设备节点。run 接口主要是新建线程开启 GPIO 相关的业务工作，这里假设 GPIO0 作为输入，GPIO1 作为输出，当检测到 GPIO0 为低电平状态时，GPIO1 输出高电平；反之，当 GPIO0 位高电平时，GPIO1 输出低电平。stop 接口为停止 GPIO 业务，退出 GPIO 业务线程。getGPIOStateByIndex 接口可以读取 gpio-index 的输入/输出电平状态。

同样，对于 MySerial 类，也提供 init、run、stop 接口，另外提供了 getCountInfo 接口，读取传送计数信息，clearCount 接口对计数清零。run 接口开启串口业务线程，这里模拟业务为周期性的向串口发送数据（接收线程一直接收串口数据），因此还提供了一个设置周期的公共接口 setPeriod 和查询接口 getPeriod。

```
49
50      /**
51       * read rx/tx count of the serial device
52       * @param rcnt pointer of rx cnt
53       * @param tcnt pointer of tx cnt
54       * @return
55       */
56      void getCountInfo(int *rcnt, int *tcnt);
57
58      /**
59       * clear count of rx/tx
60       * @return
61       */
62      void clearCount();
63
64      /**
65       * read serial port send period
66       * @return period
67       */
68      int getPeriod();
69
70      /**
71       * set serial port send period
72       * @param per period
73       * @return
74       */
75      void setPeriod(int per);
```

图 4　MySerial 类部分公共接口

4.2 监听客户端连接，解析响应客户端请求

此部分主要对应图 1 中的 Server 模块，为此创建一个 MyServer 类，其头文件如图 5 所示。其中，registerDev 接口用于向 MyServer 类注册设备，主要是将设备类（MyGPIO、MySerial）的指针传递给 MyServer 类，当 MyServer 类解析通信包后，可以通过设备类的指针调用其公共接口，查询/设置相关参数。run 函数开启服务器，监听本地网络（127.0.0.1）的默认端口 9733，进入 accept 等待连接状态。

```
#define DEV_TYPE_GPIO 0
#define DEV_TYPE_UART 1

class MyServer{
public:
    MyServer();
    ~MyServer();

    /**
     * register the device pointer to the server, in order to use pointer
     * to invoke public method to get info
     * @param dev device pointer
     * @param type device type
     */
    void registerDev(void * dev, int type);
    /**
     * start to accept connection
     */
    void run();
    void stop();
    int isStopped();
private:
    int m_hasUIConnect;
    int m_clientSocket;

    //pointer to GPIO dev;
    MyGPIO* m_pGPIO;
    //pointer to serial dev;
    MySerial* m_pSerial;

    int m_exitFlag;
    int m_defaultPort = 9733;
    char m_defaultIP[] = "127.0.0.1";

    pthread_mutex_t m_writeMutex;

    QUEUE m_queueSend2UI;

    static int _init_server(void *param);

    static void *write_to_ui_thread_func(void *p);
    static void *recv_from_ui_thread_func(void *p);
    int write2UI(int type, char* data, int len);
};
```

图 5 MyServer 类

4.3 main 函数工作流程

在 main 函数中，首先将 GPIO 设备和串口设备注册到 Server 模块中；然后初始化并运行 GPIO 和串口的具体工作任务；最后运行 Server 模块，开始监听本地端口 9733，如果有连接到来，就建立新连接，解析来自客户端的协议帧，根据协议栈中的信息，调用 GPIO、串口设备的公共接口得到响应信息，响应客户端请求。main 函数的流程如下图所示：

```
11  int main(int argc, char** argv)
12  {
13      printf("enter main\n");
14      MyGPIO myGPIO;
15      MySerial mySerial;
16      MyServer myServer;
17
18      myServer.registerDev(&myGPIO,DEV_TYPE_GPIO);
19      myServer.registerDev(&mySerial,DEV_TYPE_UART);
20
21      myGPIO.init();
22      myGPIO.run();
23
24      mySerial.init(1);
25      mySerial.run();
26
27      myServer.run();
28
29
30      while (!myServer.isStopped()){
31          sleep(5);
32      }
33      printf("exit main\n");
34      return 0;
35  }
```

图 6 main 函数

五、客户端 Android Java 程序

```
266     private void init() {
267         mIvAlert = (ImageView) findViewById(R.id.imageView_alert);
268         mTvAlertState = (TextView) findViewById(R.id.tv_arlert_state);
269
270         mIvHydrant = (ImageView) findViewById(R.id.imageView_fire_hydrant);
271         mTvHydrantState = (TextView) findViewById(R.id.tv_fire_hydrant_state);
272
273         mBtUartSettings = (Button) findViewById(R.id.bt_uart_settings);
274         mTvUartTxCnt = (TextView) findViewById(R.id.tv_uart_tx_cnt);
275         mTvUartRxCnt = (TextView) findViewById(R.id.tv_uart_rx_cnt);
276         mTvUartPeroid = (TextView) findViewById(R.id.tv_uart_period_value);
277
278         mBtUISettings = (Button) findViewById(R.id.bt_ui_settings);
279         mTvFreshPeroid = (TextView) findViewById(R.id.tv_refresh_peroid_value);
280
281         mIP = "127.0.0.1";
282         mPort = 9733;
283     }
```

图 7 客户端初始化

```
102     private void connect2server() {
103         mClientThread = new  SocketClientThread(mIP,mPort,mHandler);
104         mClientThread.start();
105     }
```

图 8 客户端连接本地端口

客户端启动后，新建 socket 连接本地 IP 地址的默认端口（127.0.0.1:9733），如图 7 和图 8 所示。建立连接后，以固定周期向服务器发送查询请求，然后刷新界面。

图 9 客户端界面（正常状态）

图 10 客户端界面（报警状态）

如图 9 和图 10 所示，“警报输入状态”和“消防栓输出状态”分别读取的 GPIO0 和 GPIO1 的状态，当 GPIO0 输入高电平时（报警输入状态 OFF），服务端自动控制 GPIO1 输出低电平（消防栓输出状态 OFF）；反之，当 GPIO0 输入低电平时（报警输入状态 ON），服务端控制 GPIO1 输出高电平（消防栓输出状态 ON）。

“串口设备”模拟通信设备，这里仅获取了发送和接收计数值。为了模拟通信设备的自动独立运行，我们让串口周期自动发送字符串，而这个周期值可以在点击“串口设备”弹出的对话框中的设置。如图 11 所示，在弹出的对话框中，还可以选择清零底层串口的发送/接收计数值。这样就加入了人为的控制。

最后，“设置”项主要设置 UI 进程相关的属性，比如访问服务器获取信息的“刷新周期”等。

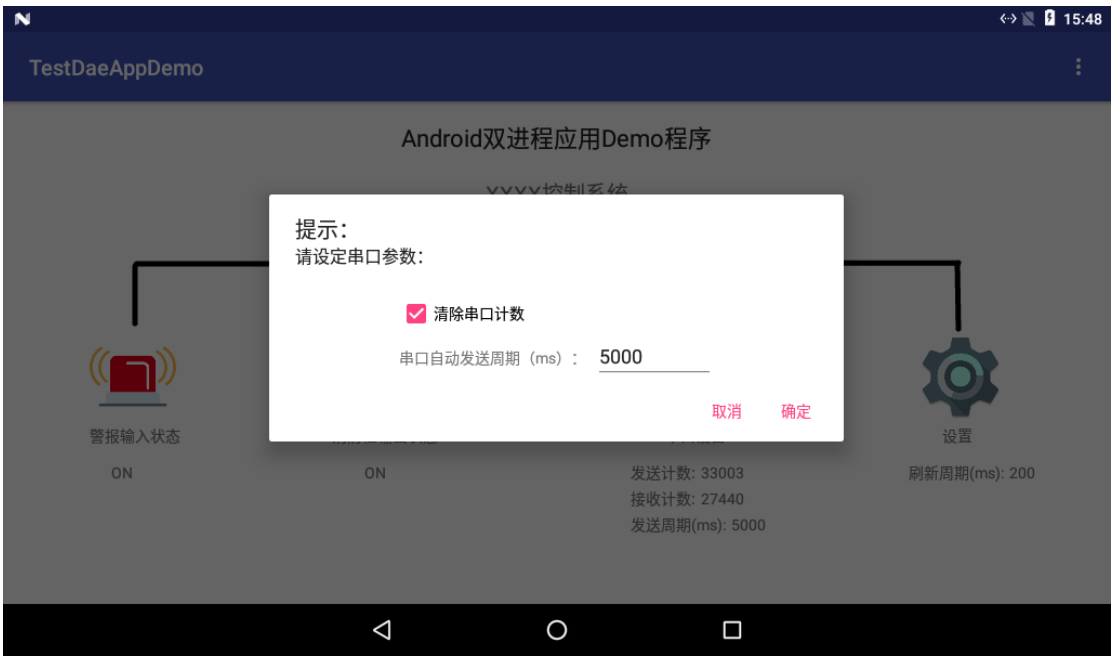

图 11 串口设置对话框

六、总结

“双应用进程”方案设计的应用程序，在原来的 C/C++程序基础上，添加一个 server 模块，将工作设备（GPIO、串口）的运转信息通过本地网络（127.0.0.1）的 socket 传送给了 Android UI 端显示；同时，server 模块又能接收 UI 端的人机交互命令，并设置到对应工作设备。这其中，主要工作是抽象出 server 与各工作设备间的通信方式，以及 server 与 UI 端的自定义通信协议及解析。此方案充分利用了原有的 C/C++程序，加快了底层业务逻辑的开发进度；同时，也降低了 Java 端界面开发的难度。

本文在 server 与各工作设备间的通信方式，以及 server 与 UI 端的自定义通信协议及解析方面只是针对第二节中简单模拟业务而设定的，其目的在于验证方案的可行性，起到抛砖引玉的作用。如果用户对此方案感兴趣，英创会提供此 demo 程序的源码，供客户参考。