华清远见 FARSIGHT 嵌入式培训专家

嵌入式学院 华清远见旗下品牌

Android核心入门分析

Jack.fan

主要内容：

- 1、android系统启动流程分析
- 2、android系统JNI和Binder使用简介
- 3、android系统输入子系统模型分析

1.1 android系统启动流程分析:

- 1).
- init进程启动控制台进程
- init进程启动servicemanager进程(即runtime进程)
- => 打开/dev/binder等,并设置自己为runtime(context),用于对系统中的所有服务进行统一管理
- init进程启动vold/debuggerd/rild进程
- 2).
- init进程启动Zygote进程

1.2 android系统启动流程分析:

- 3).
- runtime进程请求Zygote启动SystemServer进程
- 4).
- SystemServer进程启动两个本地服务:SurfaceFlinger/AudioFlinger
- SurfaceFlinger/AudioFlinger向ServiceManager注册
- 5).
- SystemServer进程启动其他Android服务(如WindowManager)
- 所有的Android服务向ServiceManager注册

1.3 android系统启动流程分析:

- android_src/system/core/init.c

- parse_config_file("/init.rc")
- snprintf(tmp, sizeof(tmp), "/init.%s.rc", hardware)
- parse_config_file(tmp)
- action_for_each_trigger("init", action_add_queue_tail) // 运行脚本文件中的on init段
- action_for_each_trigger("early-boot", action_add_queue_tail);
- action_for_each_trigger("boot", action_add_queue_tail);

1.4 android系统启动流程分析:

- init.rc
- service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
- frameworks/base/cmds/app_process/app_main.cpp
- main()
- Step 1 => 解析虚拟机的运行参数
- Step 2 => 解析dexopt运行参数
- Step 3 => 初始化VM虚拟机(Initialize the VM)　: 这里启动虚拟机后,这以后就可以运行java代码了
- Step 4 => 初始化JNI模块(注册android函数) : 这以后,CPP和JAVA代码之间就可以互相函数调用了
- Step 5 => 启动VM虚拟机(Start VM. This thread becomes the main thread of the VM, and will not return until the VM exits)

1.5 android系统启动流程分析:

- Step 6 => 启动进程system_server => 该进程会启动android的后续全部进程
- Step 7 => 启动SurfaceFlinger和AudioFlinger
  - 打开/dev/pmem设备:
  - 开始android机器人开机动画的显示:
- Step 8 => 启动其他的各种服务并将这些服务添加到ServiceManager中: 如PowerManager,ActivityManager,WindowManager,InputMethodManagerService,...
  - 这里将启动android系统上电后用户看到的第一个锁屏显示界面(HomeApp) => 最后SystemServer::init2 将会调用ActivityManagerService.systemReady 通过发送Intent.CATEGORY_HOME intent来启动第一个 activity

  然后开始等待新的android应用启动请求(提供fork()服务)

1.6 android系统启动流程分析:

```
# ps
USER     PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
root      1     0     300    208   c00bec44 0000c93c S /init
root      2     0     0      0     c0073f8c 00000000 S kthreadd
root      3     2     0      0     c006601c 00000000 S ksoftirqd/0
root      4     2     0      0     c00887b8 00000000 S watchdog/0
root      5     2     0      0     c0070ca4 00000000 S events/0
root      6     2     0      0     c0070ca4 00000000 S khelper
root      12    2     0      0     c0070ca4 00000000 S suspend
root      156   2     0      0     c0070ca4 00000000 S kblockd/0
root      162   2     0      0     c01ec304 00000000 S kseriod
root      169   2     0      0     c0070ca4 00000000 S twl4030-irqchip
root      170   2     0      0     c01ff210 00000000 S twl4030-irq
root      186   2     0      0     c0070ca4 00000000 S omap2_mcspi
root      193   2     0      0     c0070ca4 00000000 S ksuspend_usbd
root      199   2     0      0     c025f4d8 00000000 S khubd
root      204   2     0      0     c0070ca4 00000000 S kmmcd
root      229   2     0      0     c0093cd4 00000000 S pdflush
root      230   2     0      0     c0093cd4 00000000 S pdflush
root      231   2     0      0     c0098108 00000000 S kswapd0
root      233   2     0      0     c0070ca4 00000000 S aio/0
root      234   2     0      0     c0070ca4 00000000 S nfsiod
root      385   2     0      0     c0070ca4 00000000 S zd1211rw
root      406   2     0      0     c02470fc 00000000 S mtdblockd
root      474   2     0      0     c0070ca4 00000000 S rpciod/0
root      495   2     0      0     c0297258 00000000 S mmcqd
root      755   1     728    324   c0063c5c afe0d6ac S /system/bin/sh
```

1.7 android系统启动流程分析:


```
root      755   1     728    324   c0063c5c afe0d6ac S /system/bin/sh
system    757   1     796    260   c029ee64 afe0ca7c S /system/bin/servicemanager
root      759   1     832    376   c00bec44 afe0cba4 S /system/bin/vold
root      760   1     656    248   c02c4608 afe0d40c S /system/bin/debuggerd
radio     762   1     3372   676   ffffffff afe0d0ec S /system/bin/rild
root      763   1     79384  24908 c00bec44 afe0cba4 S zygote
media     764   1     18040  3912  ffffffff afe0ca7c S /system/bin/mediaserver
bluetooth 766   1     1092   524   c00bec44 afe0d87c S /system/bin/dbus-daemon
root      768   1     784    288   c031bda8 afe0c7dc S /system/bin/installd
keystore  769   1     1616   404   c02c4608 afe0d40c S /system/bin/keystore
system    778   763   109492 23820 ffffffff afe0ca7c S system_server
graphics  793   1     19392  13128 ffffffff afe0ca7c S /system/bin/bootanimation
root      801   755   872    336   00000000 afe0c7dc R ps
```

2.1 android系统JNI和Binder使用简介:

- 用于Java代码和本地代码（C/C++）的互相调用

2.2 android系统JNI和Binder使用简介:

- android java调用CPP函数: 原理 => 相当于java的那个class里面有的函数使用CPP代码来实现了

- 1)通过结构JNINativeMethod描述java代码调用函数和CPP函数的对应关系:
- typedef struct {
- const char* name; // java代码调用CPP函数的入口
- const char* signature; // CPP函数的返回值
- void* fnPtr; // CPP的函数名
- } JNINativeMethod;

2.3 android系统JNI和Binder使用简介:

- => 例如: java代码通过IBinder.transact()来调用CPP的函数android_os_BinderProxy_transact()
- {"transact", "(ILandroid/os/Parcel;Landroid/os/Parcel;I)Z", (void*)android_os_BinderProxy_transact},

- 2)将CPP函数注册到java的某个class中: 使用函数AndroidRuntime::registerNativeMethods()来注册
  - => 这之后,java代码就可以调用CPP函数了
- 3)java代码调用CPP函数方法:
  - IBinder.transact()

2.4 android系统JNI和Binder使用简介:

- andorid CPP调用java函数: 原理 => 相当于CPP代码找到java的那个class里面的函数的入口地址,然后在CPP代码中调用java代码

- 1)通过结构JNINativeInterface描述CPP代码调用java函数的对应关系:
- CallStaticVoidMethod

2.5 android系统JNI和Binder使用简介:

- 2)到java的那个class(如android.os.Binder)中找到java函数(如execTransact())的入口:
- jclass clazz = env->FindClass(kBinderPathName) // const char* const kBinderPathName = "android/os/Binder";
- gBinderOffsets.mExecTransact = env->GetMethodID(clazz, "execTransact", "(IIII)Z")
- 3)在CPP代码中调用java函数:
- env->CallBooleanMethod(mObject, gBinderOffsets.mExecTransact, code, (int32_t)&data, (int32_t)reply, flags

3.1 android输入子系统模型分析:

- Step 1 => WindowManagerService运行线程 InputDeviceReader 用于读取如下消息: 按键消息,触摸屏消息,轨迹球消息
- Step 2 => WindowManagerService运行线程PolicyThread : 结合 PhoneWindowManager.java/KeyguardViewMediator.java 来管理当前窗口显示等
- Step 3 => WindowManagerService运行线程 InputDispatcherThread : 将线程InputDeviceReader()放在事件队列里面的消息分发出去

3.2 android输入子系统模型分析:

- InputDeviceReader() 运行流程:
- Step 1: 调用底层函数,从/dev/input/下面的输入设备读入输入事件
- Step 2: 对读入的输入事件作预处理 -> 用于决定是否要分发给系统的其他模块(不分发的输入事件要么自己处理,要么丢弃,这需要让PhoneWindowManager来决定)
- Step 3: 将读入的输入事件放到各自对应的事件队列中

3.3 android输入子系统模型分析:

- 寻找kl文件的原理如下:
- 1)首先寻找/system/usr/keylayout/gpio-keys.kl => I/EventHub(698): New keyboard: publicID=65537 device->id=65537 devname='gpio-keys' propName='hw.keyboards.65537.devname' keylayout='/system/usr/keylayout/qwerty.kl'
- 2)如果没有,就默认使用/system/usr/keylayout/qwerty.kl

3.4 android输入子系统模型分析:

- 从驱动读取到输入事件的信息:
- 然后将CPP层读取到的输入事件写入java层代码中:
- env->SetIntField(event, gInputOffsets.mDeviceId, (jint)deviceId);
- env->SetIntField(event, gInputOffsets.mType, (jint)type);
- env->SetIntField(event, gInputOffsets.mScancode, (jint)scancode);
- env->SetIntField(event, gInputOffsets.mKeycode, (jint)keycode);
- env->SetIntField(event, gInputOffsets.mFlags, (jint)flags);
- env->SetIntField(event, gInputOffsets.mValue, value);

www.farsight.com.cn

Thanks !