Android Startup Sequence

Reversion	Reversion History	Date	Modifier
V.01	First Draft	2012-5-9	Xiangfei_Meng

Contents

Basic Startup Sequence

In this section, it mainly talks about the basic startup sequence of Android system, it supposes you were familiar with Linux OS startup sequence, such as boot loader, Linux kernel bring up, etc. So let's begin with Init Process.

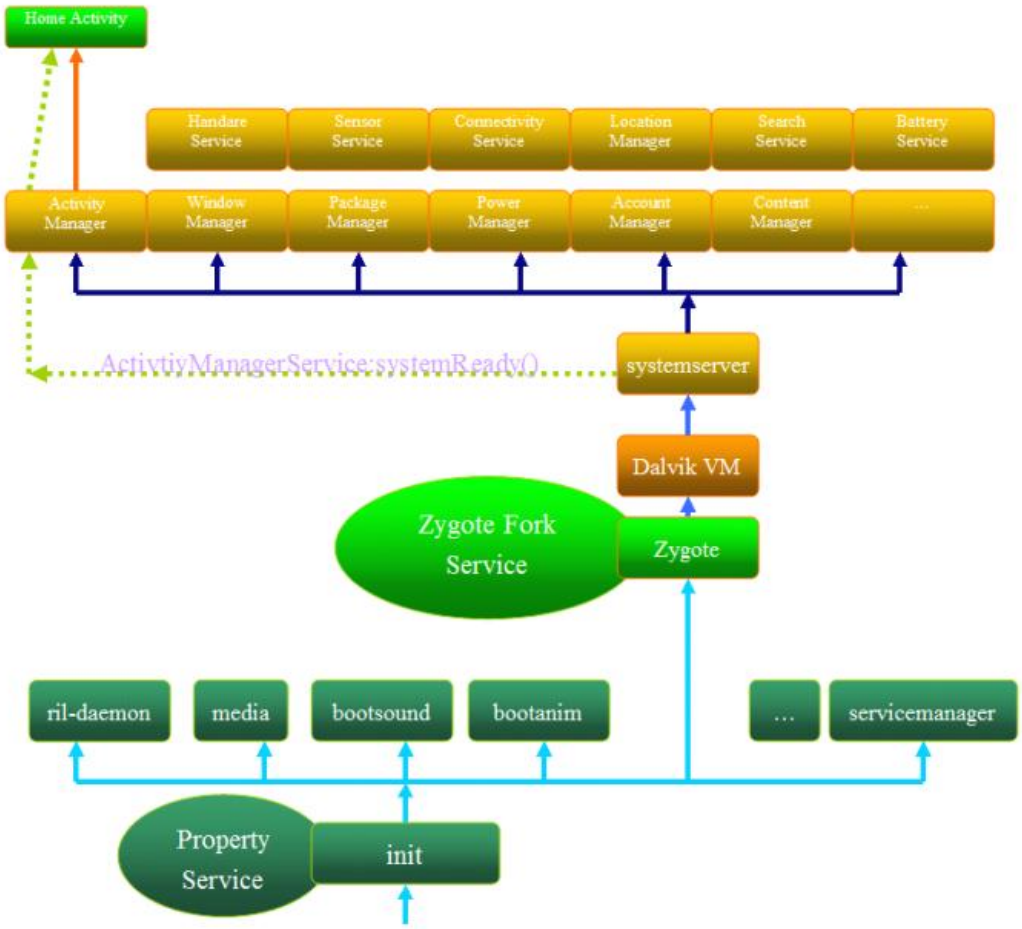

Figure 1: Main flow chart of Android startup sequence.

Init Process

Init process is a user level process which is startup by Linux kernel, and it is the first process of Android.

Reference code:

System/core/init/init.c

System/core/rootdir/init.rc

Main work of Init process

a) Get basic filesystem setup

```
mkdir("/dev", 0755);
mkdir("/proc", 0755);
mkdir("/sys", 0755);
mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755");
mkdir("/dev/pts", 0755);
mkdir("/dev/socket", 0755);
mount("devpts", "/dev/pts", "devpts", 0, NULL);
mount("proc", "/proc", "proc", 0, NULL);
mount("sysfs", "/sys", "sysfs", 0, NULL);
```

b) Log init

```
open_devnull_stdio();
klog_init();
```

c) Parse init.rc and init.xxx.rc

```
init_parse_config_file("/init.rc");
snprintf(tmp, sizeof(tmp), "/init.%s.rc", hardware);
init_parse_config_file(tmp);
```

After parse config files, add all services to service_list, add all actions to action_list.

d) Init process put all actions into several stages: early-init, init, early-fs, fs, post-fs, post-fs-data, early-boot, boot.

```
action_for_each_trigger("early-init", action_add_queue_tail);
action_for_each_trigger("init", action_add_queue_tail);
action_for_each_trigger("early-fs", action_add_queue_tail);
action_for_each_trigger("fs", action_add_queue_tail);
action_for_each_trigger("post-fs", action_add_queue_tail);
action_for_each_trigger("post-fs-data", action_add_queue_tail);
action_for_each_trigger("early-boot", action_add_queue_tail);
action_for_each_trigger("boot", action_add_queue_tail);
```

e) Execute command, all commands have been added into actions_list.

```
execute_one_command();
```

f) Start service, all services have been added into service_list.

```
restart_processes();
```

g) Monitor events from property server, socket, and keychord.

```
    for (i = 0; i < fd_count; i++) {
        if (ufds[i].revents == POLLIN) {
            if (ufds[i].fd == get_property_set_fd())
                handle_property_set_fd();
            else if (ufds[i].fd == get_keychord_fd())
                handle_keychord();
            else if (ufds[i].fd == get_signal_fd())
                handle_signal();
        }
}
```

Important services in init.rc

Servicemanager: it is a server that manages all services in Android system.

```
service servicemanager /system/bin/servicemanager
    class core
    user system
    group system
    critical
    onrestart restart zygote
    onrestart restart media
    onrestart restart surfaceflinger
    onrestart restart drm
```

Zygote: it is a server that can produce other service, such as service_server, etc.

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    class main
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

Surfaceflinger:

```
service surfaceflinger /system/bin/surfaceflinger
    class main
    user system
    group graphics
    onrestart restart zygote
```

Media:

```
service media /system/bin/mediaserver
    class main
```

```
    user media
    group audio camera inet net_bt net_bt_admin net_bw_acct drmrpc cw_access oma_drm_group
    ioprio rt 4
```

……

Servicemanager

ServiceManager is the manager of all services in Android system, it is startup by Init process.

Reference code

frameworks/base/cmds/servicemanager/service_manager.c

```
int main(int argc, char **argv)
{
    struct binder_state *bs;
    void *svcmgr = BINDER_SERVICE_MANAGER;

    bs = binder_open(128*1024);

    if (binder_become_context_manager(bs)) {
        LOGE("cannot become context manager (%s)\n", strerror(errno));
        return -1;
    }

    svcmgr_handle = svcmgr;
    binder_loop(bs, svcmgr_handler);
    return 0;
}
```

ServiceManager's work

1) Open Binder device
2) Set itself as a manager
3) Wait and dispose requests from clients

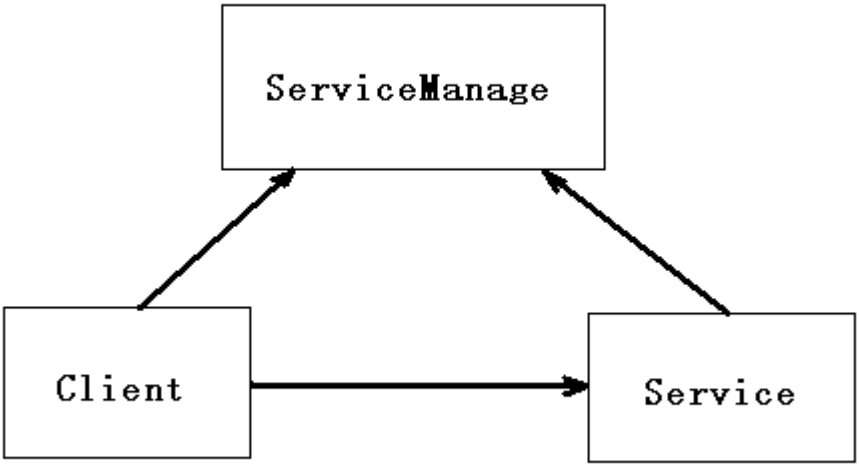

Figure 2: relationship between ServiceManager, Client and Service

First, Service must register itself to ServiceManager

Then, Client sent request to ServiceManager to get the related Service

When Client gets its related Service, the communication is built.

Note: not all the services can be register to ServiceManager

I) If the uid of the service is root or system, then it can be registered

II) Otherwise, it must be added into allowed[]

```
static struct {
        unsigned uid;
        const char *name;
} allowed[] = {
#ifdef LVMX
        { AID_MEDIA, "com.lifevibes.mx.ipc" },
#endif
        { AID_MEDIA, "media.pcm_service" }, //here must have!, or can not register...
        { AID_MEDIA, "media.audio_flinger" },
        { AID_MEDIA, "media.player" },
// U5G Jimmy.CM Chang add for Aricent VT. +
#ifdef CFLAG_VT
#ifdef CFLAG_VT_ARICENT
        { AID_MEDIA, "vtservice" },
#endif
#endif
// U5G Jimmy.CM Chang add for Aricent VT. -
        { AID_MEDIA, "media.camera" },
        { AID_MEDIA, "media.audio_policy" },
        { AID_DRM,      "drm.drmManager" },
```

```
        { AID_NFC,      "nfc" },
        { AID_RADIO, "radio.phone" },
        { AID_RADIO, "radio.sms" },
        { AID_RADIO, "radio.phonesubinfo" },
        { AID_RADIO, "radio.simphonebook" },
/* TODO: remove after phone services are updated: */
        { AID_RADIO, "phone" },
        { AID_RADIO, "sip" },
        { AID_RADIO, "isms" },
        { AID_RADIO, "iphonesubinfo" },
        { AID_RADIO, "simphonebook" },
//+[HTC_PHONE]: BVS, allow to add SIM Authentication Service.
        { AID_RADIO, "htc.sim_authentication" },
//-[HTC_PHONE]: BVS, allow to add SIM Authentication Service.
        { AID_GRAPHICS, "htc.abl" },
//@+Ausmus HtcTelephony
        { AID_RADIO, "htctelephony" },
        { AID_RADIO, "htctelephonyinternal" },
//@-Ausmus
};
```

Zygote

Zygote is the most important process for Android system, it is startup by Init process.

The main work of zygote

1) Start VM
2) Register JNI
3) Preload classes and resources
4) Fork service_server
5) Wait and response requests from activity, fork new process.

AppRuntime

AppRuntime is a Class, which extends from AndroidRuntime Class. Zygote call

androidruntime.start to set up zygote process.

```
frameworks/base/cmds/app_process/app_main.cpp
int main(int argc, const char* const argv[])
{
……
AppRuntime runtime;
……
// parse runtime parameters
……
    runtime.start("com.android.internal.os.ZygoteInit",
        startSystemServer ? "start-system-server" : "");
}
```

The parameters are set in init.rc:

service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server

```
class AppRuntime : public AndroidRuntime
{
public:
    AppRuntime()
        : mParentDir(NULL)
        , mClassName(NULL)
        , mClass(NULL)
        , mArgC(0)
        , mArgV(NULL)
    {
}
……
virtual void onStarted() {……}
virtual void onZygoteInit() {……}
virtual void onExit(int code) {……}
……
}
```

```
frameworks/base/include/android_runtime/AndroidRuntime.h
class AndroidRuntime
{
public:
    AndroidRuntime();
    virtual ~AndroidRuntime();

    enum StartMode {
        Zygote,
        SystemServer,
        Application,
```

```
        Tool,
};
.....
int addVmArguments(int argc, const char* const argv[]);
void start(const char *classname, const char* options);
.....
}
```

Start Android runtime

```
customize/base/core/jni/AndroidRuntime.cpp
void AndroidRuntime::start(const char* className, const char* options)
{
// className is “com.android.internal.os.ZygoteInit”, options is “start-system-server”
.....
    /* start the virtual machine */
    JNIEnv* env;
    if (startVm(&mJavaVM, &env) != 0) {
        return;
    }
    onVmCreated(env);
    .....
    /*
     * Register android functions.
     */
    if (startReg(env) < 0) {
        LOGE("Unable to register all android natives\n");
        return;
    }
    .....
    /*
     * Start VM.  This thread becomes the main thread of the VM, and will
     * not return until the VM exits.
     */
    char* slashClassName = toSlashClassName(className);
    jclass startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
        LOGE("JavaVM unable to locate class '%s'\n", slashClassName);
        /* keep going */
    } else {
        jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
```

```
            if (startMeth == NULL) {
                LOGE("JavaVM unable to find main() in '%s'\n", className);
                /* keep going */
            } else {
                env->CallStaticVoidMethod(startClass, startMeth, strArray);
            }
        }
        free(slashClassName);

}
.....
}
```

Step1: call startVM function to set up VM
Step2: call startReg function to register JNI function,
Step3: call JNI function *env->GetStaticMethodID* to get the jMethodId of static main function of class ZygoteInit
Step4: call JNI function *env->CallStaticVoidMethod*, then the VM will run ZygoteInit main() function, and zygote enters Java word from now.

ZygoteInit

Class ZygoteInit is the startup class for zygote process, its main() function is the entrance for Java VM.

```
frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
    public static void main(String argv[]) {
        try {
            // Start profiling the zygote initialization.
            SamplingProfilerIntegration.start();

            registerZygoteSocket();
            EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
                SystemClock.uptimeMillis());
            preload();
            EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
                SystemClock.uptimeMillis());

          .....
            if (argv[1].equals("start-system-server")) {
                startSystemServer();
            } else if (!argv[1].equals("")) {
```

```
                throw new RuntimeException(argv[0] + USAGE_STRING);
            }

            Log.i(TAG, "Accepting command socket connections");

            if (ZYGOTE_FORK_MODE) {
                runForkMode();
            } else {
                runSelectLoopMode();
            }

            closeServerSocket();
        } catch (MethodAndArgsCaller caller) {
            caller.run();
        } catch (RuntimeException ex) {
            Log.e(TAG, "Zygote died with exception", ex);
            closeServerSocket();
            throw ex;
        }
}
```

Step1: call registerZygoteSocket function to create the IPC communication server. It's a socket, Zygote use this socket to communicate with Clients.

Step2: call preload function to load classes and resources

preloadClasses() load cleasses, there are more than 1200 classes, and that will cost a long time to load.

preloadResources() load resources which from framework-res.apk.

Step3: call startSystemServer function to create system_server process.

Step4: call runSelectLoopMode function, this function is a while loop, it monitors the socket which be created in step1.

Step5: catch MethodAndArgsCaller exception, startSystemService will through an exception, and call run method to execute the main function of system_server process.

So far, Zygote process has accomplished its mission, it will go to sleep if there are no Clients resume it.

SystemServer

SystemServer is the first server created by Zygote, it's the core of framework.

```
    private static boolean startSystemServer()
            throws MethodAndArgsCaller, RuntimeException {
        /* Hardcoded command line to start the system server */

        ......
        int pid;

        try {
            ......
            /* Request to fork the system server process */
            pid = Zygote.forkSystemServer(
                    parsedArgs.uid, parsedArgs.gid,
                    parsedArgs.gids,
                    parsedArgs.debugFlags,
                    null,
                    parsedArgs.permittedCapabilities,
                    parsedArgs.effectiveCapabilities);
        } catch (IllegalArgumentException ex) {
            throw new RuntimeException(ex);
        }

        /* For child process */
        if (pid == 0) {
            handleSystemServerProcess(parsedArgs);
        }
          return true;
}
```

Step1: call Zygote.forkSystemServer to fork system_server process
Step2: call handleSystemServerProcess function to enter system_server process, child process system_server begin to work.

What does handleSystemServerProcess function do? Let's see its code.

```
    private static void handleSystemServerProcess(
            .......
            closeServerSocket();
            ......
            // Pass the remaining arguments to SystemServer.
                    RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion,
```

```
parsedArgs.remainingArgs);
        }
```

First, close the socket inherited from Zygote process

Then, call zygoteInit method of RuntimeInit class, pay attention to the arguments which include “com.android.server.SystemServer”

```
frameworks/base/core/java/com/android/internal/os/RuntimeInit.java
    public static final void zygoteInit(int targetSdkVersion, String[] argv)
            throws ZygoteInit.MethodAndArgsCaller {
        if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application from zygote");

        redirectLogStreams();

        commonInit();
        zygoteInitNative();

        applicationInit(targetSdkVersion, argv);
}
    private static void applicationInit(int targetSdkVersion, String[] argv)
            throws ZygoteInit.MethodAndArgsCaller {
        // We want to be fairly aggressive about heap utilization, to avoid
        // holding on to a lot of memory that isn't needed.
        VMRuntime.getRuntime().setTargetHeapUtilization(0.75f);
        VMRuntime.getRuntime().setTargetSdkVersion(targetSdkVersion);

        final Arguments args;
        try {
            args = new Arguments(argv);
        } catch (IllegalArgumentException ex) {
            Slog.e(TAG, ex.getMessage());
            // let the process exit
            return;
        }

        // Remaining arguments are passed to the start class's static main
        invokeStaticMain(args.startClass, args.startArgs);
}
```

There are two points should be pay attention on:

One is zygoteInitNative(), this function is a native function, which is defined in AndroidRuntime.cpp, in this function, it starts a thread used for Binder communication, we don’t talk about Binder communication in this section.

The other is invokeStaticMain(args.startClass, args.startArgs), let ‘s see its code

```
    private static void invokeStaticMain(String className, String[] argv)
        throws ZygoteInit.MethodAndArgsCaller {
……
    /*
     * This throw gets caught in ZygoteInit.main(), which responds
     * by invoking the exception's run() method. This arrangement
     * clears up all the stack frames that were required in setting
     * up the process.
     */
throw new ZygoteInit.MethodAndArgsCaller(m, argv);
}
```

This function throws an exception, do you remember Step5 in Zygoteinit.main() function, this exception will be caught by Zygoteinit.main(), and then the main() function of system_server process will run.

Now, the child process system_server begin running.

SystemServer's work

Now let's see what does SystemServer do?

frameworks/base/services/java/com/android/server/SystemServer.java

```
public static void main(String[] args) {
……
    System.loadLibrary("android_servers");
init1(args);
}
```

First, it loads libandroid_servers.so then call init1 function
Init1() function is native function, it is defined in com_android_server_SystemServer.cpp
frameworks/base/services/jni/com_android_server_SystemServer.cpp

```
    static void android_server_SystemServer_init1(JNIEnv* env, jobject clazz
    {
        system_init();
}
```

frameworks/base/cmds/system_server/library/system_init.cpp

```
extern "C" status_t system_init()
{
……
    char propBuf[PROPERTY_VALUE_MAX];
    property_get("system_init.startsurfaceflinger", propBuf, "1");
```

```
    if (strcmp(propBuf, "1") == 0) {
        // Start the SurfaceFlinger
        SurfaceFlinger::instantiate();
    }

    property_get("system_init.startsensorservice", propBuf, "1");
    if (strcmp(propBuf, "1") == 0) {
        // Start the sensor service
        SensorService::instantiate();
    }
    LOGI("System server: starting Android runtime.\n");
    AndroidRuntime* runtime = AndroidRuntime::getRuntime();

    jmethodID methodId = env->GetStaticMethodID(clazz, "init2", "()V");
    if (methodId == NULL) {
        return UNKNOWN_ERROR;
    }
    env->CallStaticVoidMethod(clazz, methodId);

    ......
}
```

Step1: start SurfaceFlinger service
Step2: start sensor service
Step3: start AndroidRumtime
Step4: call init2 function

```
    public static final void init2() {
        Slog.i(TAG, "Entered the Android system server!");
        Thread thr = new ServerThread();
        thr.setName("android.server.ServerThread");
        thr.start();
}
```

Init2 is java function, which is defined in SystemServer.java
It starts a thread ServerThread, ServerThread.run is a long function, almost all the services were startup in this function, and add them to ServerManager.

At the end of ServerThread.run(), ActivityManagerService call its systemReady()

```
    // We now tell the activity manager it is okay to run third party
    // code.    It will call back into us once it has gotten to the state
    // where third party code can really run (but before it has actually
    // started launching the initial applications), for us to complete our
    // initialization.
    ActivityManagerService.self().systemReady(new Runnable() {
```

```
        public void run() {
            Slog.i(TAG, "Making services ready");
    ......
    }
```

In ActivityManagerService.self().systemReady(), it calls mMainStack.resumeTopActivityLocked(null) at the end of this function.

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
    public void systemReady(final Runnable goingCallback) {
        synchronized(this) {
            if (mSystemReady) {
                if (goingCallback != null) goingCallback.run();
                return;
            }
            ......
            mMainStack.resumeTopActivityLocked(null);
     }
```

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
final boolean resumeTopActivityLocked(ActivityRecord prev) {
......
    // Find the first activity that is not finishing.
ActivityRecord next = topRunningActivityLocked(null);
......
    if (next == null) {
        // There are no more activities!    Let's just start up the
        // Launcher...
        if (mMainStack) {
            return mService.startHomeActivityLocked();
        }
}
......
}
```

Because the activity stack is NULL now, so it will call startHomeActivityLocked(), and this function will startup Launcher.

```
boolean startHomeActivityLocked() {
......
    Intent intent = new Intent(
        mTopAction,
        mTopData != null ? Uri.parse(mTopData) : null);
    intent.setComponent(mTopComponent);
```

```
        if (mFactoryTest != SystemServer.FACTORY_TEST_LOW_LEVEL) {
            intent.addCategory(Intent.CATEGORY_HOME);
}
……
}
```

In function startHomeActivityLocked(), it creates a intent with CATEGORY_HOME, and this will startup related activity----Launchar.

Till now, Android system has startup to HOME!

Logs during Android system startup

Logcat: we should use logcat to get the logs from frameworks, the logs are massy. I just list the useful log during system bring up.

D/AndroidRuntime(103): >>>>>> AndroidRuntime START com.android.internal.os.ZygoteInit <<<<<<

This log indicates that system has enter Zygote process, just before start VM

JNI_CreateJavaVM failed

This log means VM be created failed

Unable to register all android natives

This log means register native function failed

Shutting down VM

VM shut down

I/dalvikvm(103): System server process 256 has been created

This log indicates that system_server has been forked, PID is 256

System server process 256 has died. Restarting Zygote!

This log means that system_server died, Zygote should be restart

I/Zygote (103): Accepting command socket connections

This log indicates that system has enter ZygoteInit.main(), that is java layor, before this log, the system has create a socket for communication between Zygote and its clients; has load all classes and resources; system_server process has been forked.

Zygote died with exception

This log means Zygote died because of exception.

I/sysproc (256): Entered system_init()

This log indicates that system has enter Init1(), this function is in native layor.

I/sysproc (256): System server: starting Android runtime.
I/sysproc (256): System server: starting Android services.
I/sysproc (256): System server: entering thread pool.

Those logs are all print out in Init1()

I/SystemServer(256): Entered the Android system server!

This log indicates that system has enter init2(), it will create all the service next.

I/SystemServer(256): Making services ready

This log indicates that all the service will be ready

System ServerThread is exiting!

This log means that system_server exited, this won't happen normally.

I/ActivityManager(256): System now ready

This log indicates that system is ready

I/ActivityManager(256): START {act=android.intent.action.MAIN cat=[android.intent.category.HOME] flg=0x10000000 cmp=com.htc.launcher/.Launcher} from pid 0

This log indicates that Launcher has been startup, and system will be startup to HOME