Android Kernel (3) - Kernel Bootstrapping Part 1 - Zygote

SEPTEMBER 06, 2014

android (/tags/android.html)

---

1. **init** starts daemon Services, including first Android Dalvik VM: **zygote**
2. zygote defines a Socket, used to accept request of starting app from ActivityManagerService.

3. zygote creates `system_server` process using `fork`
4. `system_server` starts Native System Service and Java System Service
5. new system Service will be registered to ServiceManager
6. ActivityManagerService enters systemReady state
7. ActivityManagerService communicates with Socket of zygote, then starting **Home** app.
8. zygote accepts the connect request from ActivityManagerService, and run `runSelectLoopMode` request
9. zygote uses `forkAndSpecialize` to start a new app process, and then starts Home.

Zygote

Zygote is a daemon service. All Dalvik VM process are forked from zygote. Compare to creating VM separately for each app process, it improves the performance. All app processes can share VM memory and framework layer resources.

The configure for zygote is in `init.rc`:

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --st
art-system-server
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

We can see there is a `socket` command there. And the actual bin file is `app_process`. We look for `MODULE_NAME` in all `Android.mk`, and found the source code of zygote is in `frameworks/base/cmds/app_process/app_main.cpp`.

```
 1 void app_usage()
 2 {
 3     fprintf(stderr,
 4         "Usage: app_process [java-options] cmd-dir start-class-name
[options]\n");
 5 }
 6
 7 int main(int argc, char* const argv[])
 8 {
 9 #ifdef __arm__
10     /*
11      * b/7188322 - Temporarily revert to the compat memory layout
12      * to avoid breaking third party apps.
13      *
14      * THIS WILL GO AWAY IN A FUTURE ANDROID RELEASE.
15      *
16      * http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.g
it;a=commitdiff;h=7dbaa466
17      * changes the kernel mapping from bottom up to top-down.
18      * This breaks some programs which improperly embed
19      * an out of date copy of Android's linker.
20      */
21     char value[PROPERTY_VALUE_MAX];
22     property_get("ro.kernel.qemu", value, "");
23     bool is_qemu = (strcmp(value, "1") == 0);
24     if ((getenv("NO_ADDR_COMPAT_LAYOUT_FIXUP") == NULL) && !is_qemu)
{
25         int current = personality(0xFFFFFFFF);
26         if ((current & ADDR_COMPAT_LAYOUT) == 0) {
27             personality(current | ADDR_COMPAT_LAYOUT);
28             setenv("NO_ADDR_COMPAT_LAYOUT_FIXUP", "1", 1);
29             execv("/system/bin/app_process", argv);
30             return -1;
31         }
32     }
33     unsetenv("NO_ADDR_COMPAT_LAYOUT_FIXUP");
34 #endif
35
36     // These are global variables in ProcessState.cpp
37     mArgC = argc;
38     mArgV = argv;
39
40     mArgLen = 0;
41     for (int i=0; i<argc; i++) {
```

```
42          mArgLen += strlen(argv[i]) + 1;
43      }
44      mArgLen--;
45
46      AppRuntime runtime;
47      const char* argv0 = argv[0];
48
49      // Process command line arguments
50      // ignore argv[0]
51      argc--;
52      argv++;
53
54      // Everything up to '--' or first non '-' arg goes to the vm
55
56      int i = runtime.addVmArguments(argc, argv);
57
58      // Parse runtime arguments.  Stop at first unrecognized option.
59      bool zygote = false;
60      bool startSystemServer = false;
61      bool application = false;
62      const char* parentDir = NULL;
63      const char* niceName = NULL;
64      const char* className = NULL;
65      while (i < argc) {
66          const char* arg = argv[i++];
67          if (!parentDir) {
68              parentDir = arg;
69          } else if (strcmp(arg, "--zygote") == 0) {
70              zygote = true;
71              niceName = "zygote";
72          } else if (strcmp(arg, "--start-system-server") == 0) {
73              startSystemServer = true;
74          } else if (strcmp(arg, "--application") == 0) {
75              application = true;
76          } else if (strncmp(arg, "--nice-name=", 12) == 0) {
77              niceName = arg + 12;
78          } else {
79              className = arg;
80              break;
81          }
82      }
83
84      if (niceName && *niceName) {
85          setArgv0(argv0, niceName);
86          set_process_name(niceName);
```

```
 87     }
 88
 89     runtime.mParentDir = parentDir;
 90
 91     if (zygote) {
 92         runtime.start("com.android.internal.os.ZygoteInit",
 93                 startSystemServer ? "start-system-server" : "");
 94     } else if (className) {
 95         // Remainder of args get passed to startup class main()
 96         runtime.mClassName = className;
 97         runtime.mArgC = argc - i;
 98         runtime.mArgV = argv + i;
 99         runtime.start("com.android.internal.os.RuntimeInit",
100                 application ? "application" : "tool");
101     } else {
102         fprintf(stderr, "Error: no class name or --zygote supplie
d.\n");
103         app_usage();
104         LOG_ALWAYS_FATAL("app_process: no class name or --zygote supp
lied.");
105         return 10;
106     }
107 }
```

`app_usage()` shows the command line usage of `app_process`. There are 3 steps of `main()`:

1. based on `-Xzygote`, sets options of VM
2. based on `--zygote`, name of the process
3. based on `--zygote` and `--start-system-server`, call `start()` of `AppRuntime` to start `ZygoteInit` or `RuntimeInit`

ZygoteInit

The most complex job is in `AppRuntime.start` (line 90). It is defined in `frameworks/base/core/jni/AndroidRuntime.cpp`:

```
  1 /*
  2 * Start the Android runtime.  This involves starting the virtual mach
ine
  3 * and calling the "static void main(String[] args)" method in the cla
ss
  4 * named by "className".
  5 *
  6 * Passes the main function two arguments, the class name and the spec
ified
  7 * options string.
  8 */
  9 void AndroidRuntime::start(const char* className, const char* option
s)
 10 {
 11         ALOGD("\n>>>>>> AndroidRuntime START %s <<<<<<\n",
 12                         className != NULL ? className : "(unknown)");
 13
 14         /*
 15        * 'startSystemServer == true' means runtime is obsolete and no
t run from
 16        * init.rc anymore, so we print out the boot start event here.
 17        */
 18         if (strcmp(options, "start-system-server") == 0) {
 19                 /* track our progress through the boot sequence */
 20                 const int LOG_BOOT_PROGRESS_START = 3000;
 21                 LOG_EVENT_LONG(LOG_BOOT_PROGRESS_START,
 22                                                 ns2ms(systemTime(SYSTEM_T
IME_MONOTONIC)));
 23         }
 24
 25         const char* rootDir = getenv("ANDROID_ROOT");
 26         if (rootDir == NULL) {
 27                 rootDir = "/system";
 28                 if (!hasDir("/system")) {
 29                         LOG_FATAL("No root directory specified, and
/android does not exist.");
 30                         return;
 31                 }
 32                 setenv("ANDROID_ROOT", rootDir, 1);
 33         }
 34
 35         //const char* kernelHack = getenv("LD_ASSUME_KERNEL");
 36         //ALOGD("Found LD_ASSUME_KERNEL='%s'\n", kernelHack);
 37
```

```
 38         /* start the virtual machine */
 39         JniInvocation jni_invocation;
 40         jni_invocation.Init(NULL);
 41         JNIEnv* env;
 42         if (startVm(&mJavaVM, &env) != 0) {
 43                 return;
 44         }
 45         onVmCreated(env);
 46
 47         /*
 48        * Register android functions.
 49        */
 50         if (startReg(env) < 0) {
 51                 ALOGE("Unable to register all android natives\n");
 52                 return;
 53         }
 54
 55         /*
 56        * We want to call main() with a String array with arguments in
it.
 57        * At present we have two arguments, the class name and an opti
on string.
 58        * Create an array to hold them.
 59        */
 60         jclass stringClass;
 61         jobjectArray strArray;
 62         jstring classNameStr;
 63         jstring optionsStr;
 64
 65         stringClass = env->FindClass("java/lang/String");
 66         assert(stringClass != NULL);
 67         strArray = env->NewObjectArray(2, stringClass, NULL);
 68         assert(strArray != NULL);
 69         classNameStr = env->NewStringUTF(className);
 70         assert(classNameStr != NULL);
 71         env->SetObjectArrayElement(strArray, 0, classNameStr);
 72         optionsStr = env->NewStringUTF(options);
 73         env->SetObjectArrayElement(strArray, 1, optionsStr);
 74
 75         /*
 76        * Start VM.  This thread becomes the main thread of the VM, an
d will
 77        * not return until the VM exits.
 78        */
 79         char* slashClassName = toSlashClassName(className);
```

```
 80         jclass startClass = env->FindClass(slashClassName);
 81         if (startClass == NULL) {
 82                 ALOGE("JavaVM unable to locate class '%s'\n", slashCl
assName);
 83                 /* keep going */
 84         } else {
 85                 jmethodID startMeth = env->GetStaticMethodID(startCla
ss, "main",
 86                         "([Ljava/lang/String;)V");
 87                 if (startMeth == NULL) {
 88                         ALOGE("JavaVM unable to find main() in
'%s'\n", className);
 89                         /* keep going */
 90                 } else {
 91                         env->CallStaticVoidMethod(startClass, startMe
th, strArray);
 92
 93 #if 0
 94                         if (env->ExceptionCheck())
 95                                 threadExitUncaughtException(env);
 96 #endif
 97                 }
 98         }
 99         free(slashClassName);
100
101         ALOGD("Shutting down VM\n");
102         if (mJavaVM->DetachCurrentThread() != JNI_OK)
103                 ALOGW("Warning: unable to detach main thread\n");
104         if (mJavaVM->DestroyJavaVM() != 0)
105                 ALOGW("Warning: VM did not shut down cleanly\n");
106 }
```

1. Create Dalvik VM

On line 42, `startVm()` is called.

```
/*
 * Start the Dalvik Virtual Machine.
 *
 * Various arguments, most determined by system properties, are passed i
n.
 * The "mOptions" vector is updated.
 *
 * Returns 0 on success.
 */
int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv)
{
    int result = -1;
    JavaVMInitArgs initArgs;
    JavaVMOption opt;
    char propBuf[PROPERTY_VALUE_MAX];

    // ...
    property_get("dalvik.vm.checkjni", propBuf, "");
    // ...
    property_get("dalvik.vm.stack-trace-file", stackTraceFileBuf, "");
    property_get("dalvik.vm.check-dex-sum", propBuf, "");
    // ...
    strcpy(jniOptsBuf, "-Xjniopts:");
    property_get("dalvik.vm.jniopts", jniOptsBuf+10, "");
    // ...
    /* enable verbose; standard options are { jni, gc, class } */
    // options[curOpt++].optionString = "-verbose:jni";
    opt.optionString = "-verbose:gc";
    mOptions.add(opt);

    /*
     * The default starting and maximum size of the heap.  Larger
     * values should be specified in a product property override.
     */
    strcpy(heapstartsizeOptsBuf, "-Xms");
    property_get("dalvik.vm.heapstartsize", heapstartsizeOptsBuf+4, "4
m");
    opt.optionString = heapstartsizeOptsBuf;
    mOptions.add(opt);
    strcpy(heapsizeOptsBuf, "-Xmx");
    property_get("dalvik.vm.heapsize", heapsizeOptsBuf+4, "16m");
    opt.optionString = heapsizeOptsBuf;
    mOptions.add(opt);
```

```
    // ...
    ALOGD("CheckJNI is %s\n", checkJni ? "ON" : "OFF");
    if (checkJni) {
        /* extended JNI checking */
        opt.optionString = "-Xcheck:jni";
        mOptions.add(opt);

        /* set a cap on JNI global references */
        opt.optionString = "-Xjnigreflimit:2000";
        mOptions.add(opt);

        /* with -Xcheck:jni, this provides a JNI function call trace */
        //opt.optionString = "-verbose:jni";
        //mOptions.add(opt);
    }
    // ...
    /* Set the properties for locale */
    {
        char langOption[sizeof("-Duser.language=") + 3];
        char regionOption[sizeof("-Duser.region=") + 3];
        strcpy(langOption, "-Duser.language=");
        strcpy(regionOption, "-Duser.region=");
        readLocale(langOption, regionOption);
        opt.extraInfo = NULL;
        opt.optionString = langOption;
        mOptions.add(opt);
        opt.optionString = regionOption;
        mOptions.add(opt);
    }
    /*
     * Initialize the VM.
     *
     * The JavaVM* is essentially per-process, and the JNIEnv* is per-thr
ead.
     * If this call succeeds, the VM is ready, and we can start issuing
     * JNI calls.
     */
    if (JNI_CreateJavaVM(pJavaVM, pEnv, &initArgs) < 0) {
        ALOGE("JNI_CreateJavaVM failed\n");
        goto bail;
    }

    result = 0;

bail:
```

```
    free(stackTraceFile);
    return result;
}
```

`startVm()` does two jobs:

1. Get VM configure info from Property System, and set VM arguments
2. Create VM by calling `JNI_CreateJavaVM`

VM options are inside `dalvik/docs/dexopt.html`, or by `adb shell dalvikvm` shell command.

2. Register JNI methods

Register JNI method for Java to call Native method. In line 50, we have `startReg()`:

```
/*
 * Register android native functions with the VM.
 */
/*static*/ int AndroidRuntime::startReg(JNIEnv* env)
{
    /*
     * This hook causes all future threads created in this process to be
     * attached to the JavaVM.  (This needs to go away in favor of JNI
     * Attach calls.)
     */
    androidSetCreateThreadFunc((android_create_thread_fn) javaCreateThrea
dEtc);

    ALOGV("--- registering native functions ---\n");

    /*
     * Every "register" function calls one or more things that return
     * a local reference (e.g. FindClass).  Because we haven't really
     * started the VM yet, they're all getting stored in the base frame
     * and never released.  Use Push/Pop to manage the storage.
     */
    env->PushLocalFrame(200);

    if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
        env->PopLocalFrame(NULL);
        return -1;
    }
    env->PopLocalFrame(NULL);

    //createJavaThread("fubar", quickTest, (void*) "hello");

    return 0;
}
```

The core of this method, is `register_jni_procs()`:

```
static int register_jni_procs(const RegJNIRec array[], size_t count, JNIE
nv* env)
{
    for (size_t i = 0; i < count; i++) {
        if (array[i].mProc(env) < 0) {
#ifndef NDEBUG
            ALOGD("----------!!! %s failed to load\n", array[i].mName);
#endif
            return -1;
        }
    }
    return 0;
}
```

`register_jni_procs()` wraps `gRegJNI`, which is a `RegJNIRec` array. It calls `mProc()` on every element of `gRegJNI`.

Here is `RegJNIRec`:

```
#ifdef NDEBUG
    #define REG_JNI(name)      { name }
    struct RegJNIRec {
        int (*mProc)(JNIEnv*);
    };
#else
    #define REG_JNI(name)      { name, #name }
    struct RegJNIRec {
        int (*mProc)(JNIEnv*);
        const char* mName;
    };
#endif
```

`RegJNIRec` is a struct, contains a function pointer.

Here is `gRegJNI` array:

```
static const RegJNIRec gRegJNI[] = {
    REG_JNI(register_android_debug_JNITest),
    REG_JNI(register_com_android_internal_os_RuntimeInit),
    REG_JNI(register_android_os_SystemClock),
    REG_JNI(register_android_util_EventLog),
    REG_JNI(register_android_util_Log),
    // ...
    REG_JNI(register_android_animation_PropertyValuesHolder),
    REG_JNI(register_com_android_internal_content_NativeLibraryHelper),
    REG_JNI(register_com_android_internal_net_NetworkStatsFactory),
};
```

It calls `REG_JNI` macro, assign method name to `mProc` function pointer. The method will finally be called as `mProc` method. An example `register_com_android_internal_os_RuntimeInit()`:

```
int register_com_android_internal_os_RuntimeInit(JNIEnv* env)
{
    return jniRegisterNativeMethods(env, "com/android/internal/os/Runtime
Init",
        gMethods, NELEM(gMethods));
}
```

Where are the JNI native methods?

They are in so shared libraries, `libandroid_runtime.so`. Check the `Android.mk` of `app_process`:

```
LOCAL_SHARED_LIBRARIES := \
    libcutils \
    libutils \
    liblog \
    libbinder \
    libandroid_runtime
```

`libandroid_runtime.so` is at `/system/lib/` of the release of android system, or the `out/target/product/generic/` of source code project.

3. Enter Java World

On line 91 of `AndroidRuntime::start()`, we call `CallStaticVoidMethod()`. This method calls `main()` of `ZygoteInit`. It is defined in `frameworks/base/java/com/android/internal/os/ZygoteInit.java`:

```
1 public static void main(String argv[]) {
2     try {
3         // Start profiling the zygote initialization.
4         SamplingProfilerIntegration.start();
5 
6         registerZygoteSocket();
7         EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
8             SystemClock.uptimeMillis());
9         preload();
10        EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
11            SystemClock.uptimeMillis());
12 
13        // Finish profiling the zygote initialization.
14        SamplingProfilerIntegration.writeZygoteSnapshot();
15 
16        // Do an initial gc to clean up after startup
17        gc();
18 
19        // Disable tracing so that forked processes do not inherit sta
le tracing tags from
20        // Zygote.
21        Trace.setTracingEnabled(false);
22 
23        // If requested, start system server directly from Zygote
24        if (argv.length != 2) {
25            throw new RuntimeException(argv[0] + USAGE_STRING);
26        }
27 
28        if (argv[1].equals("start-system-server")) {
29            startSystemServer();
30        } else if (!argv[1].equals("")) {
31            throw new RuntimeException(argv[0] + USAGE_STRING);
32        }
33 
34        Log.i(TAG, "Accepting command socket connections");
35 
36        runSelectLoop();
37 
38        closeServerSocket();
39    } catch (MethodAndArgsCaller caller) {
40        caller.run();
41    } catch (RuntimeException ex) {
42        Log.e(TAG, "Zygote died with exception", ex);
43        closeServerSocket();
```

```
44          throw ex;
45      }
46  }
```

1. register zygote Socket
2. pre-load **Class** and **Resource**
3. run system_server process
4. run `run()` of `MethodAndArgsCaller`
5. run runSelectLoopMode()`

1. Register zygote Socket

Line 4 calls `registerZygoteSocket()`:

```
    /**
     * Registers a server socket for zygote command connections
     *
     * @throws RuntimeException when open fails
     */
    private static void registerZygoteSocket() {
        if (sServerSocket == null) {
            int fileDesc;
            try {
                String env = System.getenv(ANDROID_SOCKET_ENV);
                fileDesc = Integer.parseInt(env);
            } catch (RuntimeException ex) {
                throw new RuntimeException(
                        ANDROID_SOCKET_ENV + " unset or invalid", ex);
            }

            try {
                sServerSocket = new LocalServerSocket(
                        createFileDescriptor(fileDesc));
            } catch (IOException ex) {
                throw new RuntimeException(
                        "Error binding to local socket '" + fileDesc +
"'", ex);
            }
        }
    }
```

2. Preload Class and Resource

Line 7 calls `preload()`:

```
    static void preload() {
        preloadClasses();
        preloadResources();
        preloadOpenGL();
    }

    /**
     * Performs Zygote process initialization. Loads and initializes
     * commonly used classes.
     *
     * Most classes only cause a few hundred bytes to be allocated, but
     * a few will allocate a dozen Kbytes (in one case, 500+K).
     */
    private static void preloadClasses() {
        final VMRuntime runtime = VMRuntime.getRuntime();

        InputStream is = ClassLoader.getSystemClassLoader().getResourceAs
Stream(
                PRELOADED_CLASSES);
        if (is == null) {
            Log.e(TAG, "Couldn't find " + PRELOADED_CLASSES + ".");
        } else {
            Log.i(TAG, "Preloading classes...");
            long startTime = SystemClock.uptimeMillis();

            // Drop root perms while running static initializers.
            setEffectiveGroup(UNPRIVILEGED_GID);
            setEffectiveUser(UNPRIVILEGED_UID);

            // Alter the target heap utilization.  With explicit GCs this
            // is not likely to have any effect.
            float defaultUtilization = runtime.getTargetHeapUtilizatio
n();
            runtime.setTargetHeapUtilization(0.8f);

            // Start with a clean slate.
            System.gc();
            runtime.runFinalizationSync();
            Debug.startAllocCounting();

            try {
                BufferedReader br
                    = new BufferedReader(new InputStreamReader(is), 256);
```

```
                int count = 0;
                String line;
                while ((line = br.readLine()) != null) {
                    // Skip comments and blank lines.
                    line = line.trim();
                    if (line.startsWith("#") || line.equals("")) {
                        continue;
                    }

                    try {
                        if (false) {
                            Log.v(TAG, "Preloading " + line + "...");
                        }
                        Class.forName(line);
                        if (Debug.getGlobalAllocSize() > PRELOAD_GC_THRES
HOLD) {
                            if (false) {
                                Log.v(TAG,
                                    " GC at " + Debug.getGlobalAllocSiz
e());
                            }
                            System.gc();
                            runtime.runFinalizationSync();
                            Debug.resetGlobalAllocSize();
                        }
                        count++;
                    } catch (ClassNotFoundException e) {
                        Log.w(TAG, "Class not found for preloading: " + l
ine);
                    } catch (Throwable t) {
                        Log.e(TAG, "Error preloading " + line + ".", t);
                        if (t instanceof Error) {
                            throw (Error) t;
                        }
                        if (t instanceof RuntimeException) {
                            throw (RuntimeException) t;
                        }
                        throw new RuntimeException(t);
                    }
                }

                Log.i(TAG, "...preloaded " + count + " classes in "
                        + (SystemClock.uptimeMillis()-startTime) + "m
s.");
            } catch (IOException e) {
```

```
                Log.e(TAG, "Error reading " + PRELOADED_CLASSES + ".",
e);
            } finally {
                IoUtils.closeQuietly(is);
                // Restore default.
                runtime.setTargetHeapUtilization(defaultUtilization);

                // Fill in dex caches with classes, fields, and methods b
rought in by preloading.
                runtime.preloadDexCaches();

                Debug.stopAllocCounting();

                // Bring back root. We'll need it later.
                setEffectiveUser(ROOT_UID);
                setEffectiveGroup(ROOT_GID);
            }
        }
    }
```

---

1. load class names from file called `preloaded-classes`
2. Load classes using Java Reflection

What's inside `preloaded-classes`?

```
# Classes which are preloaded by com.android.internal.os.ZygoteInit.
# Automatically generated by frameworks/base/tools/preload/WritePreloaded
ClassFile.java.
# MIN_LOAD_TIME_MICROS=1250
# MIN_PROCESSES=10
android.R$styleable
android.accounts.Account
android.accounts.Account$1
android.accounts.AccountManager
android.accounts.AccountManager$12
... // on my computer, 2783 lines
```

This file is generated by `frameworks/base/tools/preload/WritePreloadedClassFile.java`. `frameworks/base/tools/preload` package is the preload module. This tool does 2 things:

1. check whether there is a class load time exceeds `MIN_LOAD_TIME_MICROS` (by default 1250)
2. check whether there is a class has been loaded by at least `MIN_PROCESSES` (by default 10)

The classes listed in `preloaded-classes` will be kept in memory. When a new application start, it can re-use the resource.

```
    /**
     * Load in commonly used resources, so they can be shared across
     * processes.
     *
     * These tend to be a few Kbytes, but are frequently in the 20-40K
     * range, and occasionally even larger.
     */
    private static void preloadResources() {
        final VMRuntime runtime = VMRuntime.getRuntime();

        Debug.startAllocCounting();
        try {
            System.gc();
            runtime.runFinalizationSync();
            mResources = Resources.getSystem();
            mResources.startPreloading();
            if (PRELOAD_RESOURCES) {
                Log.i(TAG, "Preloading resources...");

                long startTime = SystemClock.uptimeMillis();
                TypedArray ar = mResources.obtainTypedArray(
                        com.android.internal.R.array.preloaded_drawable
s);
                int N = preloadDrawables(runtime, ar);
                ar.recycle();
                Log.i(TAG, "...preloaded " + N + " resources in "
                        + (SystemClock.uptimeMillis()-startTime) + "m
s.");

                startTime = SystemClock.uptimeMillis();
                ar = mResources.obtainTypedArray(
                        com.android.internal.R.array.preloaded_color_stat
e_lists);
                N = preloadColorStateLists(runtime, ar);
                ar.recycle();
                Log.i(TAG, "...preloaded " + N + " resources in "
                        + (SystemClock.uptimeMillis()-startTime) + "m
s.");
            }
            mResources.finishPreloading();
        } catch (RuntimeException e) {
            Log.w(TAG, "Failure preloading resources", e);
        } finally {
            Debug.stopAllocCounting();
```

```
        }
    }
```

`preloadResources` loads drawables and color. These system resources are defined in `frameworks/base/core/res/res/values/arrays.xml`, and will be built into `framework-res.apk`.

3. start system_server process

All the system service in Android are started by system_server process.

system_server is started in `startSystemServer()`, on line 29:

```
    /**
     * Prepare the arguments and fork for the system server process.
     */
    private static boolean startSystemServer()
            throws MethodAndArgsCaller, RuntimeException {
        long capabilities = posixCapabilitiesAsBits(
            OsConstants.CAP_KILL,
            OsConstants.CAP_NET_ADMIN,
            OsConstants.CAP_NET_BIND_SERVICE,
            OsConstants.CAP_NET_BROADCAST,
            OsConstants.CAP_NET_RAW,
            OsConstants.CAP_SYS_MODULE,
            OsConstants.CAP_SYS_NICE,
            OsConstants.CAP_SYS_RESOURCE,
            OsConstants.CAP_SYS_TIME,
            OsConstants.CAP_SYS_TTY_CONFIG
        );
        /* Hardcoded command line to start the system server */
        String args[] = {
            "--setuid=1000",
            "--setgid=1000",
            "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,101
0,1018,1032,3001,3002,3003,3006,3007",
            "--capabilities=" + capabilities + "," + capabilities,
            "--runtime-init",
            "--nice-name=system_server",
            "com.android.server.SystemServer",
        };
        ZygoteConnection.Arguments parsedArgs = null;

        int pid;

        try {
            parsedArgs = new ZygoteConnection.Arguments(args);
            ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);
            ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);

            /* 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;
    }
```

---

system_server is forked from zygote

1. create system_server in `forkSystemServer()`

Zygote class is defined in `libcore/dalvik/src/main/java/dalvik/system/Zygote.java`.

```
    /**
     * Special method to start the system server process. In addition to
the
     * common actions performed in forkAndSpecialize, the pid of the chil
d
     * process is recorded such that the death of the child process will
cause
     * zygote to exit.
     *
     * @param uid the UNIX uid that the new process should setuid() to af
ter
     * fork()ing and and before spawning any threads.
     * @param gid the UNIX gid that the new process should setgid() to af
ter
     * fork()ing and and before spawning any threads.
     * @param gids null-ok; a list of UNIX gids that the new process shou
ld
     * setgroups() to after fork and before spawning any threads.
     * @param debugFlags bit flags that enable debugging features.
     * @param rlimits null-ok an array of rlimit tuples, with the second
     * dimension having a length of 3 and representing
     * (resource, rlim_cur, rlim_max). These are set via the posix
     * setrlimit(2) call.
     * @param permittedCapabilities argument for setcap()
     * @param effectiveCapabilities argument for setcap()
     *
     * @return 0 if this is the child, pid of the child
     * if this is the parent, or -1 on error.
     */
    public static int forkSystemServer(int uid, int gid, int[] gids, int
debugFlags,
            int[][] rlimits, long permittedCapabilities, long effectiveCa
pabilities) {
        preFork();
        int pid = nativeForkSystemServer(
                uid, gid, gids, debugFlags, rlimits, permittedCapabilitie
s, effectiveCapabilities);
        postFork();
        return pid;
    }
```

Basicly, `forkSystemServer()` -> `nativeForkSystemServer()` -> `Dalvik_dalvik_system_Zygote_forkSystemServe()`, in `/dalvik/vm/native`:

```
/*
 * native public static int nativeForkSystemServer(int uid, int gid,
 *     int[] gids, int debugFlags, int[][] rlimits,
 *     long permittedCapabilities, long effectiveCapabilities);
 */
static void Dalvik_dalvik_system_Zygote_forkSystemServer(
        const u4* args, JValue* pResult)
{
    pid_t pid;
    pid = forkAndSpecializeCommon(args, true);

    /* The zygote process checks whether the child process has died or no
t. */
    if (pid > 0) {
        int status;

        ALOGI("System server process %d has been created", pid);
        gDvm.systemServerPid = pid;
        /* There is a slight window that the system server process has cr
ashed
         * but it went unnoticed because we haven't published its pid ye
t. So
         * we recheck here just to make sure that all is well.
         */
        if (waitpid(pid, &status, WNOHANG) == pid) {
            ALOGE("System server process %d has died. Restarting Zygot
e!", pid);
            kill(getpid(), SIGKILL);
        }
    }
    RETURN_INT(pid);
}
```

system_server will build up **Native System Service** and **Java System Service**. This creates the Java environment. Otherwise system need to restart zygote. `Dalvik_dalvik_system_Zygote_forkSystemServer()` does 2 jobs:

1. create system_server process

```
/*
 * Utility routine to fork zygote and specialize the child process.
 */
static pid_t forkAndSpecializeCommon(const u4* args, bool isSystemServer)
{
    pid_t pid;

    setSignalHandler();

    pid = fork();

    // ...

    return pid;
}
```

It forks the sub-routine system_server, and registers **signal handler**.

```
/*
 * configure sigchld handler for the zygote process
 * This is configured very late, because earlier in the dalvik lifecycle
 * we can fork() and exec() for the verifier/optimizer, and we
 * want to waitpid() for those rather than have them be harvested immedia
tely.
 *
 * This ends up being called repeatedly before each fork(), but there's
 * no real harm in that.
 */
static void setSignalHandler()
{
    int err;
    struct sigaction sa;

    memset(&sa, 0, sizeof(sa));

    sa.sa_handler = sigchldHandler;

    err = sigaction (SIGCHLD, &sa, NULL);

    if (err < 0) {
        ALOGW("Error setting SIGCHLD handler: %s", strerror(errno));
    }
}
```

2. monitor system_server starting log

```
/*
 * This signal handler is for zygote mode, since the zygote
 * must reap its children
 */
static void sigchldHandler(int s)
{ /* ... */ }
```

`sigchldHandler()` is called before sub-routine exits.

`forkSystemServer()` is cautious about create system_server process. It checks whether system_server is finished, and also monitors the state of system_server process after creating it. Anything wrong will cause zygote finish itself.

2. `handleSystemServerProcess()`

In the last part of `startSystemServer()`, `handleSystemServerProcess()` is called in sub-routine, defined in `ZygoteInit.java`:

```
    /**
     * Finish remaining work for the newly forked system server process.
     */
    private static void handleSystemServerProcess(
            ZygoteConnection.Arguments parsedArgs)
            throws ZygoteInit.MethodAndArgsCaller {

        closeServerSocket();

        // set umask to 0077 so new files and directories will default to
owner-only permissions.
        Libcore.os.umask(S_IRWXG | S_IRWXO);

        if (parsedArgs.niceName != null) {
            Process.setArgV0(parsedArgs.niceName);
        }

        if (parsedArgs.invokeWith != null) {
            WrapperInit.execApplication(parsedArgs.invokeWith,
                    parsedArgs.niceName, parsedArgs.targetSdkVersion,
                    null, parsedArgs.remainingArgs);
        } else {
            /*
             * Pass the remaining arguments to SystemServer.
             */
            RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArg
s.remainingArgs);
        }

        /* should never reach here */
    }
```

It does some clean up and initialization, and then calls `RuntimeInit.zygoteInit()`

```
    /**
     * The main function called when started through the zygote process.
This
     * could be unified with main(), if the native code in nativeFinishIn
it()
     * were rationalized with Zygote startup.<p>
     *
     * Current recognized args:
     * <ul>
     *   <li> <code> [--] &lt;start class name&gt;  &lt;args&gt;
     * </ul>
     *
     * @param targetSdkVersion target SDK version
     * @param argv arg strings
     */
    public static final void zygoteInit(int targetSdkVersion, String[] ar
gv)
            throws ZygoteInit.MethodAndArgsCaller {
        if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application from zy
gote");

        redirectLogStreams();

        commonInit();
        nativeZygoteInit();

        applicationInit(targetSdkVersion, argv);
    }
```

---

1. `redirectLogStreams()` redirect std IO operations

```
    /**
     * Redirect System.out and System.err to the Android log.
     */
    public static void redirectLogStreams() {
        System.out.close();
        System.setOut(new AndroidPrintStream(Log.INFO, "System.out"));
        System.err.close();
        System.setErr(new AndroidPrintStream(Log.WARN, "System.err"));
    }
```

---

2. `commonInit()` initialise common settings

This method does common initialization.

```
    private static final void commonInit() {
        if (DEBUG) Slog.d(TAG, "Entered RuntimeInit!");

        /* set default handler; this applies to all threads in the VM */
        Thread.setDefaultUncaughtExceptionHandler(new UncaughtHandler());

        /*
         * Install a TimezoneGetter subclass for ZoneInfo.db
         */
        TimezoneGetter.setInstance(new TimezoneGetter() {
            @Override
            public String getId() {
                return SystemProperties.get("persist.sys.timezone");
            }
        });
        TimeZone.setDefault(null);

        /*
         * Sets handler for java.util.logging to use Android log faciliti
es.
         * The odd "new instance-and-then-throw-away" is a mirror of how
         * the "java.util.logging.config.class" system property works. We
         * can't use the system property here since the logger has almost
         * certainly already been initialized.
         */
        LogManager.getLogManager().reset();
        new AndroidConfig();

        /*
         * Sets the default HTTP User-Agent used by HttpURLConnection.
         */
        String userAgent = getDefaultUserAgent();
        System.setProperty("http.agent", userAgent);

        /*
         * Wire socket tagging to traffic stats.
         */
        NetworkManagementSocketTagger.install();

        /*
         * If we're running in an emulator launched with "-trace", put th
e
         * VM into emulator trace profiling mode so that the user can hit
         * F9/F10 at any time to capture traces.  This has performance
```

```
         * consequences, so it's not something you want to do always.
         */
        String trace = SystemProperties.get("ro.kernel.android.tracing");
        if (trace.equals("1")) {
            Slog.i(TAG, "NOTE: emulator trace profiling enabled");
            Debug.enableEmulatorTraceOutput();
        }

        initialized = true;
    }
```

3. `onZygoteInit()` starting Binder communication

`nativeZygoteInit()` is a Native method, JNI implementation inside `AndroidRuntime.cpp`, by method `com_android_internal_os_RuntimeInit_nativeZygoteInit()`:

```
static void com_android_internal_os_RuntimeInit_nativeZygoteInit(JNIEnv*
env, jobject clazz)
{
    gCurRuntime->onZygoteInit();
}
```

Prefix `g` indicates a global variable. `onZygoteInit()` defined in `frameworks/base/cmds/app_process/app_main.cpp`:

```
    virtual void onZygoteInit()
       {
           // Re-enable tracing now that we're no longer in Zygote.
           atrace_set_tracing_enabled(true);

           sp<ProcessState> proc = ProcessState::self();
           ALOGV("App process: starting thread pool.\n");
           proc->startThreadPool();
       }
```

Simply `onZygoteInit` start the Binder communication channel of system_server.

4. `invokeStaticMain()` throws exception

`applicationInit()` defined in `frameworks/base/core/java/com/android/internal/os/RuntimeInit.java`:

```
    private static void applicationInit(int targetSdkVersion, String[] ar
gv)
            throws ZygoteInit.MethodAndArgsCaller {
        // If the application calls System.exit(), terminate the process
        // immediately without running any shutdown hooks.  It is not pos
sible to
        // shutdown an Android application gracefully.  Among other thing
s, the
        // Android runtime shutdown hooks close the Binder driver, which
can cause
        // leftover running threads to crash before the process actually
exits.
        nativeSetExitWithoutCleanup(true);

        // We want to be fairly aggressive about heap utilization, to avo
id
        // 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 mai
n
        invokeStaticMain(args.startClass, args.startArgs);
    }

    /**
     * Invokes a static "main(argv[]) method on class "className".
     * Converts various failing exceptions into RuntimeExceptions, with
     * the assumption that they will then cause the VM instance to exit.
     *
     * @param className Fully-qualified class name
     * @param argv Argument vector for main()
     */
    private static void invokeStaticMain(String className, String[] argv)
            throws ZygoteInit.MethodAndArgsCaller {
```

```
        Class<?> cl;

        try {
            cl = Class.forName(className);
        } catch (ClassNotFoundException ex) {
            throw new RuntimeException(
                    "Missing class when invoking static main " + className,
                    ex);
        }

        Method m;
        try {
            m = cl.getMethod("main", new Class[] { String[].class });
        } catch (NoSuchMethodException ex) {
            throw new RuntimeException(
                    "Missing static main on " + className, ex);
        } catch (SecurityException ex) {
            throw new RuntimeException(
                    "Problem getting static main on " + className, ex);
        }

        int modifiers = m.getModifiers();
        if (! (Modifier.isStatic(modifiers) && Modifier.isPublic(modifiers))) {
            throw new RuntimeException(
                    "Main method is not public and static on " + className);
        }

        /*
         * 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);
    }
```

It loads `com.android.server.SystemServer` class, then throws a `MethodAndroidArgsCaller` exception.

Notice that in `main()` of `ZygoteInit.java`:

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

Here `caller.run()` gets called. `MethodAndArgsCaller` is defined in `ZygoteInit.java`:

```
/**
 * Helper exception class which holds a method and arguments and
 * can call them. This is used as part of a trampoline to get rid of
 * the initial process setup stack frames.
 */
public static class MethodAndArgsCaller extends Exception
        implements Runnable {
    /** method to call */
    private final Method mMethod;

    /** argument array */
    private final String[] mArgs;

    public MethodAndArgsCaller(Method method, String[] args) {
        mMethod = method;
        mArgs = args;
    }

    public void run() {
        try {
            mMethod.invoke(null, new Object[] { mArgs });
        } catch (IllegalAccessException ex) {
            throw new RuntimeException(ex);
        } catch (InvocationTargetException ex) {
            Throwable cause = ex.getCause();
            if (cause instanceof RuntimeException) {
                throw (RuntimeException) cause;
            } else if (cause instanceof Error) {
                throw (Error) cause;
            }
            throw new RuntimeException(ex);
        }
    }
}
```

`MethodAndArgsCaller` is a **Exception** and a **Runnable**. Exception handling part calls `run()`, and `run()` runs the method passed in, which is `mMethod`.

Thus, `invokeStaticMain()` runs the `main()` of `com.android.server.SystemServer`, and poped from stack, go back to `main()` of `ZygoteInit`.

What does *`SystemServer.main()`* do?

Inside `frameworks/base/services/java/com/android/server/SystemServer.java`:

```
    public static void main(String[] args) {
        if (System.currentTimeMillis() < EARLIEST_SUPPORTED_TIME) {
            // If a device's clock is before 1970 (before 0), a lot of
            // APIs crash dealing with negative numbers, notably
            // java.io.File#setLastModified, so instead we fake it and
            // hope that time from cell towers or NTP fixes it
            // shortly.
            Slog.w(TAG, "System clock is before 1970; setting to 1970.");
            SystemClock.setCurrentTimeMillis(EARLIEST_SUPPORTED_TIME);
        }

        if (SamplingProfilerIntegration.isEnabled()) {
            SamplingProfilerIntegration.start();
            timer = new Timer();
            timer.schedule(new TimerTask() {
                @Override
                public void run() {
                    SamplingProfilerIntegration.writeSnapshot("system_ser
ver", null);
                }
            }, SNAPSHOT_INTERVAL, SNAPSHOT_INTERVAL);
        }

        // Mmmmmm... more memory!
        dalvik.system.VMRuntime.getRuntime().clearGrowthLimit();

        // The system server has to run all of the time, so it needs to b
e
        // as efficient as possible with its memory usage.
        VMRuntime.getRuntime().setTargetHeapUtilization(0.8f);

        Environment.setUserRequired(true);

        System.loadLibrary("android_servers");

        Slog.i(TAG, "Entered the Android system server!");

        // Initialize native services.
        nativeInit();

        // This used to be its own separate thread, but now it is
        // just the loop we run on the main thread.
        ServerThread thr = new ServerThread();
        thr.initAndLoop();
```

```
    }
```

1. apply for more memory
2. load android_server library
3. run `nativeInit()`

`nativeInit()` is the former `init1()`, defined in `com_android_server_SystemServer.cpp`

```
static void android_server_SystemServer_nativeInit(JNIEnv* env, jobject c
lazz) {
    char propBuf[PROPERTY_VALUE_MAX];
    property_get("system_init.startsensorservice", propBuf, "1");
    if (strcmp(propBuf, "1") == 0) {
        // Start the sensor service
        SensorService::instantiate();
    }
}
```

Run `runSelectLoop()`

The last step in `main()` of ZygoteInit, is run `runSelectLoop()`:

```
    /**
     * Runs the zygote process's select loop. Accepts new connections as
     * they happen, and reads commands from connections one spawn-reques
t's
     * worth at a time.
     *
     * @throws MethodAndArgsCaller in a child process when a main() shoul
d
     * be executed.
     */
    private static void runSelectLoop() throws MethodAndArgsCaller {
        ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();
        ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnectio
n>();
        FileDescriptor[] fdArray = new FileDescriptor[4];

        fds.add(sServerSocket.getFileDescriptor());
        peers.add(null);

        int loopCount = GC_LOOP_COUNT;
        while (true) {
            int index;

            /*
             * Call gc() before we block in select().
             * It's work that has to be done anyway, and it's better
             * to avoid making every child do it.  It will also
             * madvise() any free memory as a side-effect.
             *
             * Don't call it every time, because walking the entire
             * heap is a lot of overhead to free a few hundred bytes.
             */
            if (loopCount <= 0) {
                gc();
                loopCount = GC_LOOP_COUNT;
            } else {
                loopCount--;
            }


            try {
                fdArray = fds.toArray(fdArray);
                index = selectReadable(fdArray);
            } catch (IOException ex) {
```

```
                throw new RuntimeException("Error in select()", ex);
            }

            if (index < 0) {
                throw new RuntimeException("Error in select()");
            } else if (index == 0) {
                ZygoteConnection newPeer = acceptCommandPeer();
                peers.add(newPeer);
                fds.add(newPeer.getFileDesciptor());
            } else {
                boolean done;
                done = peers.get(index).runOnce();

                if (done) {
                    peers.remove(index);
                    fds.remove(index);
                }
            }
        }
    }
```

zygote has a infinite loop to listen to user request via Socket. It was registered in `registerZygoteSocket()`.

The Server part is the Socket, but what is the client part? What does client send to this Socket?

First let's look at how does the Home app start.

▸ *Home* start

Home starts from the `systemReady()` of `ActivityManagerService`:

```
/** Run all ActivityStacks through this */
ActivityStackSupervisor mStackSupervisor;

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

        mStackSupervisor.resumeTopActivitiesLocked();
        sendUserSwitchBroadcastsLocked(-1, mCurrentUserId);
    }
}
```

`systemReady()` at last calls `resumeTopActivityLocked()` of a `ActivityStackSupervisor`.

```
    boolean resumeTopActivitiesLocked() {
        return resumeTopActivitiesLocked(null, null, null);
    }

    ActivityStack getFocusedStack() {
        if (mFocusedStack == null) {
            return mHomeStack;
        }
        switch (mStackState) {
            case STACK_STATE_HOME_IN_FRONT:
            case STACK_STATE_HOME_TO_FRONT:
                return mHomeStack;
            case STACK_STATE_HOME_IN_BACK:
            case STACK_STATE_HOME_TO_BACK:
            default:
                return mFocusedStack;
        }
    }

    boolean resumeTopActivitiesLocked(ActivityStack targetStack, Activity
Record target,
            Bundle targetOptions) {
        if (targetStack == null) {
            targetStack = getFocusedStack();
        }
        boolean result = false;
        for (int stackNdx = mStacks.size() - 1; stackNdx >= 0; --stackNd
x) {
            final ActivityStack stack = mStacks.get(stackNdx);
            if (isFrontStack(stack)) {
                if (stack == targetStack) {
                    result = stack.resumeTopActivityLocked(target, target
Options);
                } else {
                    stack.resumeTopActivityLocked(null);
                }
            }
        }
        return result;
    }
```

It finally calls the `startHomeActivityLocked()`. This method starts Home

```
    boolean startHomeActivityLocked(int userId) {
        if (mHeadless) {
            // Added because none of the other calls to ensureBootComplet
ed seem to fire
            // when running headless.
            ensureBootCompleted();
            return false;
        }

        if (mFactoryTest == SystemServer.FACTORY_TEST_LOW_LEVEL
                && mTopAction == null) {
            // We are running in factory test mode, but unable to find
            // the factory test app, so just sit around displaying the
            // error message and don't try to start anything.
            return false;
        }
        Intent intent = getHomeIntent();
        ActivityInfo aInfo =
            resolveActivityInfo(intent, STOCK_PM_FLAGS, userId);
        if (aInfo != null) {
            intent.setComponent(new ComponentName(
                    aInfo.applicationInfo.packageName, aInfo.name));
            // Don't do this if the home app is currently being
            // instrumented.
            aInfo = new ActivityInfo(aInfo);
            aInfo.applicationInfo = getAppInfoForUser(aInfo.applicationIn
fo, userId);
            ProcessRecord app = getProcessRecordLocked(aInfo.processName,
                    aInfo.applicationInfo.uid, true);
            if (app == null || app.instrumentationClass == null) {
                intent.setFlags(intent.getFlags() | Intent.FLAG_ACTIVIT
Y_NEW_TASK);
                mStackSupervisor.startHomeActivity(intent, aInfo);
            }
        }

        return true;
    }
```

`startHomeActivityLocked()` starts application via **Intent**. Then enters `startActivityLocked()`:

```
    final int startActivityLocked(IApplicationThread caller,
            Intent intent, String resolvedType, ActivityInfo aInfo, IBind
er resultTo,
            String resultWho, int requestCode,
            int callingPid, int callingUid, String callingPackage, int st
artFlags, Bundle options,
            boolean componentSpecified, ActivityRecord[] outActivity) {
        int err = ActivityManager.START_SUCCESS;

        ActivityRecord sourceRecord = null;
        ActivityRecord resultRecord = null;
        // ...
        int launchFlags = intent.getFlags();
        // ...
        ActivityRecord r = new ActivityRecord(mService, callerApp, callin
gUid, callingPackage,
                intent, resolvedType, aInfo, mService.mConfiguration,
                resultRecord, resultWho, requestCode, componentSpecified,
this);
        if (outActivity != null) {
            outActivity[0] = r;
        }
        // ...
        err = startActivityUncheckedLocked(r, sourceRecord, startFlags, t
rue, options);

        if (allPausedActivitiesComplete()) {
            // If someone asked to have the keyguard dismissed on the nex
t
            // activity start, but we are not actually doing an activity
            // switch...  just dismiss the keyguard now, because we
            // probably want to see whatever is behind it.
            dismissKeyguard();
        }
        return err;
    }
```

It filters the arguments for a lot of steps, and at last calls `startActivityUncheckedLocked()`. This method is very complex. I will discuss

(/)

```
    final void startActivityLocked(ActivityRecord r, boolean newTask,
            boolean doResume, boolean keepCurTransition, Bundle options)
{
        TaskRecord rTask = r.task;
        final int taskId = rTask.taskId;

        // Place a new activity at top of stack, so it is next to interac
t
        // with the user.

        // If we are not placing the new activity frontmost, we do not wa
nt
        // to deliver the onUserLeaving callback to the actual frontmost
        // activity
        if (task == r.task && mTaskHistory.indexOf(task) != (mTaskHistor
y.size() - 1)) {
            mStackSupervisor.mUserLeaving = false;
            if (DEBUG_USER_LEAVING) Slog.v(TAG,
                    "startActivity() behind front, mUserLeaving=false");
        }

        task.addActivityToTop(r);

        r.putInHistory();
        r.frontOfTask = newTask;
        if (!isHomeStack() || numActivities() > 0) {
            // We want to show the starting preview window if we are
            // switching to a new task, or the next activity's process is
            // not currently running.


        if (doResume) {
            mStackSupervisor.resumeTopActivitiesLocked();
        }
    }
```

It at last calls `resumeTopActivitiesLocked()`:

```
    boolean resumeTopActivitiesLocked() {
        return resumeTopActivitiesLocked(null, null, null);
    }

    boolean resumeTopActivitiesLocked(ActivityStack targetStack, Activity
Record target,
            Bundle targetOptions) {
        if (targetStack == null) {
            targetStack = getFocusedStack();
        }
        boolean result = false;
        for (int stackNdx = mStacks.size() - 1; stackNdx >= 0; --stackNd
x) {
            final ActivityStack stack = mStacks.get(stackNdx);
            if (isFrontStack(stack)) {
                if (stack == targetStack) {
                    result = stack.resumeTopActivityLocked(target, target
Options);
                } else {
                    stack.resumeTopActivityLocked(null);
                }
            }
        }
        return result;
    }
```

-> `startSpecificActivityLocked()` -> `startProcessLocked()`

```
    void startSpecificActivityLocked(ActivityRecord r,
            boolean andResume, boolean checkConfig) {
        // Is this activity's application already running?
        ProcessRecord app = mService.getProcessRecordLocked(r.processNam
e,
                r.info.applicationInfo.uid, true);

        r.task.stack.setLaunchTime(r);

        if (app != null && app.thread != null) {
            try {
                app.addPackage(r.info.packageName, mService.mProcessStat
s);
                realStartActivityLocked(r, app, andResume, checkConfig);
                return;
            } catch (RemoteException e) {
                Slog.w(TAG, "Exception when starting activity "
                        + r.intent.getComponent().flattenToShortString(),
e);
            }

            // If a dead object exception was thrown -- fall through to
            // restart the application.
        }

        mService.startProcessLocked(r.processName, r.info.applicationInf
o, true, 0,
                "activity", r.intent.getComponent(), false, false, true);
    }


    final ProcessRecord startProcessLocked(String processName,
            ApplicationInfo info, boolean knownToBeDead, int intentFlags,
            String hostingType, ComponentName hostingName, boolean allowW
hileBooting,
            boolean isolated, boolean keepIfLarge) {
        ProcessRecord app;
        if (!isolated) {
            app = getProcessRecordLocked(processName, info.uid, keepIfLar
ge);
        } else {
            // If this is an isolated process, it can't re-use an existin
g process.
            app = null;
```

```
        }
        // We don't have to do anything more if:
        // (1) There is an existing application record; and
        //     object attached to it so we know it couldn't have crashed;
and
        // (3) There is a pid assigned to it, so it is either starting or
        //     already running.
        // ...

        if (app == null) {
            app = newProcessRecordLocked(info, processName, isolated);
            if (app == null) {
                Slog.w(TAG, "Failed making new process record for "
                        + processName + "/" + info.uid + " isolated=" + i
solated);
                return null;
            }
            mProcessNames.put(processName, app.uid, app);
            if (isolated) {
                mIsolatedProcesses.put(app.uid, app);
            }
        } else {
            // If this is a new package in the process, add the package t
o the list
            app.addPackage(info.packageName, mProcessStats);
        }

        // If the system is not ready yet, then hold off on starting this
        // process until it is.
        if (!mProcessesReady
                && !isAllowedWhileBooting(info)
                && !allowWhileBooting) {
            if (!mProcessesOnHold.contains(app)) {
                mProcessesOnHold.add(app);
            }
            if (DEBUG_PROCESSES) Slog.v(TAG, "System not ready, putting o
n hold: " + app);
            return app;
        }

        startProcessLocked(app, hostingType, hostingNameStr);
        return (app.pid != 0) ? app : null;
    }
```

Last part, it calls its overridden method `startProcessLocked` to create process:

```
    private final void startProcessLocked(ProcessRecord app,
            String hostingType, String hostingNameStr) {

// ...

        try {
            int uid = app.uid;

            int[] gids = null;
            int mountExternal = Zygote.MOUNT_EXTERNAL_NONE;
            if (!app.isolated) {
                /*
                 * Add shared application GID so applications can share s
ome
                 * resources like shared libraries
                 */
                if (permGids == null) {
                    gids = new int[1];
                } else {
                    gids = new int[permGids.length + 1];
                    System.arraycopy(permGids, 0, gids, 1, permGids.lengt
h);
                }
                gids[0] = UserHandle.getSharedAppGid(UserHandle.getAppI
d(uid));

            // Start the process.  It will either succeed and return a re
sult containing
            // the PID of the new process, or else throw a RuntimeExcepti
on.
            Process.ProcessStartResult startResult = Process.start("andro
id.app.ActivityThread",
                    app.processName, uid, uid, gids, debugFlags, mountExt
ernal,
                    app.info.targetSdkVersion, app.info.seinfo, null);
            app.setPid(startResult.pid);
```

`startProcessLocked` callse `Process.start`, which is located in `frameworks/base/core/java/android/os/Process.java`

```
    /**
     * Start a new process.
     *
     * <p>If processes are enabled, a new process is created and the
     * static main() function of a <var>processClass</var> is executed th
ere.
     * The process will continue running after this function returns.
     *
     * <p>If processes are not enabled, a new thread in the caller's
     * process is created and main() of <var>processClass</var> called th
ere.
     *
     * <p>The niceName parameter, if not an empty string, is a custom nam
e to
     * give to the process instead of using processClass.  This allows yo
u to
     * make easily identifyable processes even if you are using the same
base
     * <var>processClass</var> to start them.
     *
     * @param processClass The class to use as the process's main entry
     *                     point.
     * @param niceName A more readable name to use for the process.
     * @param uid The user-id under which the process will run.
     * @param gid The group-id under which the process will run.
     * @param gids Additional group-ids associated with the process.
     * @param debugFlags Additional flags.
     * @param targetSdkVersion The target SDK version for the app.
     * @param seInfo null-ok SE Android information for the new process.
     * @param zygoteArgs Additional arguments to supply to the zygote pro
cess.
     *
     * @return An object that describes the result of the attempt to star
t the process.
     * @throws RuntimeException on fatal start failure
     *
     * {@hide}
     */
    public static final ProcessStartResult start(final String processClas
s,
                                  final String niceName,
                                  int uid, int gid, int[] gids,
                                  int debugFlags, int mountExternal,
                                  int targetSdkVersion,
```

```
                                    String seInfo,
                                    String[] zygoteArgs) {
        try {
            return startViaZygote(processClass, niceName, uid, gid, gids,
                    debugFlags, mountExternal, targetSdkVersion, seInfo,
zygoteArgs);
        } catch (ZygoteStartFailedEx ex) {
            Log.e(LOG_TAG,
                    "Starting VM process through Zygote failed");
            throw new RuntimeException(
                    "Starting VM process through Zygote failed", ex);
        }
    }
```

Core method here is `startViaZygote()`:

```
    /**
     * Starts a new process via the zygote mechanism.
     *
     * @param processClass Class name whose static main() to run
     * @param niceName 'nice' process name to appear in ps
     * @param uid a POSIX uid that the new process should setuid() to
     * @param gid a POSIX gid that the new process shuold setgid() to
     * @param gids null-ok; a list of supplementary group IDs that the
     * new process should setgroup() to.
     * @param debugFlags Additional flags.
     * @param targetSdkVersion The target SDK version for the app.
     * @param seInfo null-ok SE Android information for the new process.
     * @param extraArgs Additional arguments to supply to the zygote proc
ess.
     * @return An object that describes the result of the attempt to star
t the process.
     * @throws ZygoteStartFailedEx if process start failed for any reason
     */
    private static ProcessStartResult startViaZygote(final String process
Class,
                                  final String niceName,
                                  final int uid, final int gid,
                                  final int[] gids,
                                  int debugFlags, int mountExternal,
                                  int targetSdkVersion,
                                  String seInfo,
                                  String[] extraArgs)
                                  throws ZygoteStartFailedEx {
        synchronized(Process.class) {
            ArrayList<String> argsForZygote = new ArrayList<String>();

            // --runtime-init, --setuid=, --setgid=,
            // and --setgroups= must go first
            argsForZygote.add("--runtime-init");
            argsForZygote.add("--setuid=" + uid);
            argsForZygote.add("--setgid=" + gid);
            if ((debugFlags & Zygote.DEBUG_ENABLE_JNI_LOGGING) != 0) {
                argsForZygote.add("--enable-jni-logging");
            }
            if ((debugFlags & Zygote.DEBUG_ENABLE_SAFEMODE) != 0) {
                argsForZygote.add("--enable-safemode");
            }
            if ((debugFlags & Zygote.DEBUG_ENABLE_DEBUGGER) != 0) {
                argsForZygote.add("--enable-debugger");
```

```
            }
            if ((debugFlags & Zygote.DEBUG_ENABLE_CHECKJNI) != 0) {
                argsForZygote.add("--enable-checkjni");
            }
            if ((debugFlags & Zygote.DEBUG_ENABLE_ASSERT) != 0) {
                argsForZygote.add("--enable-assert");
            }
            if (mountExternal == Zygote.MOUNT_EXTERNAL_MULTIUSER) {
                argsForZygote.add("--mount-external-multiuser");
            } else if (mountExternal == Zygote.MOUNT_EXTERNAL_MULTIUSER_A
LL) {
                argsForZygote.add("--mount-external-multiuser-all");
            }
            argsForZygote.add("--target-sdk-version=" + targetSdkVersio
n);

            return zygoteSendArgsAndGetResult(argsForZygote);
        }
    }
```

`startViaZygote` sets the arguments, then calls `zygoteSendArgsAndGetResult()` method:

```
    /**
     * Sends an argument list to the zygote process, which starts a new c
hild
     * and returns the child's pid. Please note: the present implementati
on
     * replaces newlines in the argument list with spaces.
     * @param args argument list
     * @return An object that describes the result of the attempt to star
t the process.
     * @throws ZygoteStartFailedEx if process start failed for any reason
     */
    private static ProcessStartResult zygoteSendArgsAndGetResult(ArrayLis
t<String> args)
            throws ZygoteStartFailedEx {
        openZygoteSocketIfNeeded();

        try {
            /**
             * See com.android.internal.os.ZygoteInit.readArgumentList()
             * Presently the wire format to the zygote process is:
             * a) a count of arguments (argc, in essence)
             * b) a number of newline-separated argument strings equal to
count
             *
             * After the zygote process reads these it will write the pid
of
             * the child or -1 on failure, followed by boolean to
             * indicate whether a wrapper process was used.
             */

            sZygoteWriter.write(Integer.toString(args.size()));
            sZygoteWriter.newLine();

            int sz = args.size();
            for (int i = 0; i < sz; i++) {
                String arg = args.get(i);
                if (arg.indexOf('\n') >= 0) {
                    throw new ZygoteStartFailedEx(
                            "embedded newlines not allowed");
                }
                sZygoteWriter.write(arg);
                sZygoteWriter.newLine();
            }
```

```
            sZygoteWriter.flush();

            // Should there be a timeout on this?
            ProcessStartResult result = new ProcessStartResult();
            result.pid = sZygoteInputStream.readInt();
            if (result.pid < 0) {
                throw new ZygoteStartFailedEx("fork() failed");
            }
            result.usingWrapper = sZygoteInputStream.readBoolean();
            return result;
        } catch (IOException ex) {
            try {
                if (sZygoteSocket != null) {
                    sZygoteSocket.close();
                }
            } catch (IOException ex2) {
                // we're going to fail anyway
                Log.e(LOG_TAG,"I/O exception on routine close", ex2);
            }

            sZygoteSocket = null;

            throw new ZygoteStartFailedEx(ex);
        }
    }
```

`zygoteSendArgsAndGetResult` calls `openZygoteSocketIfNeeded`:

```
    /**
     * Tries to open socket to Zygote process if not already open. If
     * already open, does nothing.  May block and retry.
     */
    private static void openZygoteSocketIfNeeded()
            throws ZygoteStartFailedEx {

        int retryCount;

        /*
         * See bug #811181: Sometimes runtime can make it up before zygot
e.
         * Really, we'd like to do something better to avoid this conditi
on,
         * but for now just wait a bit...
         */
        for (int retry = 0
                ; (sZygoteSocket == null) && (retry < (retryCount + 1))
                ; retry++ ) {

            try {
                sZygoteSocket = new LocalSocket();

                sZygoteSocket.connect(new LocalSocketAddress(ZYGOTE_SOCKE
T,
                        LocalSocketAddress.Namespace.RESERVED));

                sZygoteInputStream
                        = new DataInputStream(sZygoteSocket.getInputStrea
m());

                sZygoteWriter =
                    new BufferedWriter(
                            new OutputStreamWriter(
                                    sZygoteSocket.getOutputStream()),
                            256);

                Log.i("Zygote", "Process: zygote socket opened");

                sPreviousZygoteOpenFailed = false;
                break;
            } catch (IOException ex) {
            // ...
            }
```

```
        }

        if (sZygoteSocket == null) {
            sPreviousZygoteOpenFailed = true;
            throw new ZygoteStartFailedEx("connect failed");
        }
    }
```

From `ActivityManagerService systemReady()` to `zygote`. zygote is currently run in `runSelectLoopMode()` waiting for response. Client of zygote is `ActivityManagerService`.

Back to `runSelectLoopMode()`. After it receives the response from client, it calls `runOnce()`, inside `ZygoteConnection.java`:

```
    /**
     * Reads one start command from the command socket. If successful,
     * a child is forked and a {@link ZygoteInit.MethodAndArgsCaller}
     * exception is thrown in that child while in the parent process,
     * the method returns normally. On failure, the child is not
     * spawned and messages are printed to the log and stderr. Returns
     * a boolean status value indicating whether an end-of-file on the co
mmand
     * socket has been encountered.
     *
     * @return false if command socket should continue to be read from, o
r
     * true if an end-of-file has been encountered.
     * @throws ZygoteInit.MethodAndArgsCaller trampoline to invoke main()
     * method in child process
     */
    boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {

        String args[];
        Arguments parsedArgs = null;
        FileDescriptor[] descriptors;

        try {
            args = readArgumentList();
            descriptors = mSocket.getAncillaryFileDescriptors();
        } catch (IOException ex) {
            Log.w(TAG, "IOException on command socket " + ex.getMessag
e());
            closeSocket();
            return true;
        }

        if (args == null) {
            // EOF reached.
            closeSocket();
            return true;
        }

        /** the stderr of the most recent request, if avail */
        PrintStream newStderr = null;

        if (descriptors != null && descriptors.length >= 3) {
            newStderr = new PrintStream(
                    new FileOutputStream(descriptors[2]));
```

```
        }

        int pid = -1;
        FileDescriptor childPipeFd = null;
        FileDescriptor serverPipeFd = null;

        try {
            parsedArgs = new Arguments(args);

            applyUidSecurityPolicy(parsedArgs, peer, peerSecurityContex
t);
            applyRlimitSecurityPolicy(parsedArgs, peer, peerSecurityConte
xt);
            applyCapabilitiesSecurityPolicy(parsedArgs, peer, peerSecurit
yContext);
            applyInvokeWithSecurityPolicy(parsedArgs, peer, peerSecurityC
ontext);
            applyseInfoSecurityPolicy(parsedArgs, peer, peerSecurityConte
xt);

            applyDebuggerSystemProperty(parsedArgs);
            applyInvokeWithSystemProperty(parsedArgs);

            int[][] rlimits = null;

            if (parsedArgs.rlimits != null) {
                rlimits = parsedArgs.rlimits.toArray(intArray2d);
            }

            if (parsedArgs.runtimeInit && parsedArgs.invokeWith != null)
{
                FileDescriptor[] pipeFds = Libcore.os.pipe();
                childPipeFd = pipeFds[1];
                serverPipeFd = pipeFds[0];
                ZygoteInit.setCloseOnExec(serverPipeFd, true);
            }

            pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gi
d, parsedArgs.gids,
                    parsedArgs.debugFlags, rlimits, parsedArgs.mountExter
nal, parsedArgs.seInfo,
                    parsedArgs.niceName);
        } catch (IOException ex) {
            logAndPrintError(newStderr, "Exception creating pipe", ex);
        } catch (ErrnoException ex) {
```

```
            logAndPrintError(newStderr, "Exception creating pipe", ex);
        } catch (IllegalArgumentException ex) {
            logAndPrintError(newStderr, "Invalid zygote arguments", ex);
        } catch (ZygoteSecurityException ex) {
            logAndPrintError(newStderr,
                    "Zygote security policy prevents request: ", ex);
        }

        try {
            if (pid == 0) {
                // in child
                IoUtils.closeQuietly(serverPipeFd);
                serverPipeFd = null;
                handleChildProc(parsedArgs, descriptors, childPipeFd, new
Stderr);

                // should never get here, the child is expected to either
                // throw ZygoteInit.MethodAndArgsCaller or exec().
                return true;
            } else {
                // in parent...pid of < 0 means failure
                IoUtils.closeQuietly(childPipeFd);
                childPipeFd = null;
                return handleParentProc(pid, descriptors, serverPipeFd, p
arsedArgs);
            }
        } finally {
            IoUtils.closeQuietly(childPipeFd);
            IoUtils.closeQuietly(serverPipeFd);
        }
    }

    /**
     * Handles post-fork setup of child proc, closing sockets as appropri
ate,
     * reopen stdio as appropriate, and ultimately throwing MethodAndArgs
Caller
     * if successful or returning if failed.
     *
     * @param parsedArgs non-null; zygote args
     * @param descriptors null-ok; new file descriptors for stdio if avai
lable.
     * @param pipeFd null-ok; pipe for communication back to Zygote.
     * @param newStderr null-ok; stream to use for stderr until stdio
     * is reopened.
```

```
     *
     * @throws ZygoteInit.MethodAndArgsCaller on success to
     * trampoline to code that invokes static main.
     */
    private void handleChildProc(Arguments parsedArgs,
            FileDescriptor[] descriptors, FileDescriptor pipeFd, PrintStr
eam newStderr)
            throws ZygoteInit.MethodAndArgsCaller {

        closeSocket();
        ZygoteInit.closeServerSocket();

        if (descriptors != null) {
            try {
                ZygoteInit.reopenStdio(descriptors[0],
                        descriptors[1], descriptors[2]);

                for (FileDescriptor fd: descriptors) {
                    IoUtils.closeQuietly(fd);
                }
                newStderr = System.err;
            } catch (IOException ex) {
                Log.e(TAG, "Error reopening stdio", ex);
            }
        }

        if (parsedArgs.niceName != null) {
            Process.setArgV0(parsedArgs.niceName);
        }

        if (parsedArgs.runtimeInit) {
            if (parsedArgs.invokeWith != null) {
                WrapperInit.execApplication(parsedArgs.invokeWith,
                        parsedArgs.niceName, parsedArgs.targetSdkVersion,
                        pipeFd, parsedArgs.remainingArgs);
            } else {
                RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion,
                        parsedArgs.remainingArgs);
            }
        } else {
            String className;
            try {
                className = parsedArgs.remainingArgs[0];
            } catch (ArrayIndexOutOfBoundsException ex) {
                logAndPrintError(newStderr,
```

```
                    "Missing required class name argument", null);
                return;
            }

            String[] mainArgs = new String[parsedArgs.remainingArgs.lengt
h - 1];
            System.arraycopy(parsedArgs.remainingArgs, 1,
                    mainArgs, 0, mainArgs.length);

            if (parsedArgs.invokeWith != null) {
                WrapperInit.execStandalone(parsedArgs.invokeWith,
                        parsedArgs.classpath, className, mainArgs);
            } else {
                ClassLoader cloader;
                if (parsedArgs.classpath != null) {
                    cloader = new PathClassLoader(parsedArgs.classpath,
                            ClassLoader.getSystemClassLoader());
                } else {
                    cloader = ClassLoader.getSystemClassLoader();
                }

                try {
                    ZygoteInit.invokeStaticMain(cloader, className, mainA
rgs);
                } catch (RuntimeException ex) {
                    logAndPrintError(newStderr, "Error starting.", ex);
                }
            }
        }
    }
```

`runOnce` has some subroutines required by Home wihtin it. `handleChildProc()` calls `RuntimeInit.zygoteInit()` finally. We called `RuntimeInit.zygoteInit()` when starting system_server. At that time, it calls `redirectLogStreams()`, `commonInit()`, `zygoteInitNative()`, `applicationInit()` and finally calls `main()` of a class that `invokeStateMain`. Since we set the calling class to be `ActivityThread`, thus we calls `ActivityThread.main()` here.

After that, Home application will be started.

Code Summary

- `main()` in `app_main.cpp`
  - `AndroidRuntime::start()` : L90
  - `startVm()` : L42
  - `startReg()` : L50
  - `CallStaticVoidMethod()` : L91
    - `main()` of `ZygoteInit`
    - `registerZygoteSocket()` : L4
    - `preloads()` : L9
    - `startSystemServer()` : L29
      - call `forkSystemServer()`
      - call `handleSystemServerProcess()` -> `RuntimeInit.zygoteInit()`
      - `redirectLogStreams()`
      - `commitInit()`
      - `onZygoteInit()`
      - `applicationInit()` -> `invokeStaticMain()` : loads `SystemServer` class
        - `MethodAndArgsCaller.run()`
    - `runSelectLoop()` : L36
      - get response from client `ActivityManagerService`
      - `runOnce()`, in `ZygoteConnection.java`

Get response in `runSelectLoop()`

- `ActivityManagerService.systemReady()`
  - `ActivityStackSupervisor.resumeTopActivityLocked()`

- `startHomeActivityLocked()`, which starts **Home**
  - `startActivityLocked()`
  - `startActivityUncheckedLocked()`
    - another `startActivityLocked()`
    - `resumeTopActivitiesLocked()` -> `startSpecificActivityLocked()` -> `startProcessLocked()`
      - `Process.start()`
      - `startViaZygote()`
        -
        -

Share this article

ALSO ON ALLENLSY

WHAT'S THIS?

Ultimate Workspace Collection

4 comments • 2 years ago

allenlsy — 是IKEA的大爱杯

Continuous Testing environment setup in Rails

3 comments • 2 years ago

allenlsy — You are right, I don't need to use both at the same time. Without `spork` it actually

Google+ 团队的 Android UI 测试 — Simple, Not Easy

7 comments • 10 months ago

Li7tleMK — http://ww2.sinaimg.cn/mw690/6b.. You can see the image and

virtual keyword in C

1 comment • 3 years ago

zh0014in — 这个代码框是用的啥插件啊，可以改样式么。

嘿嘿参北斗哇 (http://www.baidu.com/p/嘿嘿参北斗哇)　　帐号管理


(http://duoshuo.com/settings/avatar/)

说点什么吧...

分享到:　发布

allenlsy正在使用多说 (http://duoshuo.com)

Blog (http://allenlsy.com) · Casts (http://cast.allenlsy.com) ·

(https://twitter.com/allenlsy)