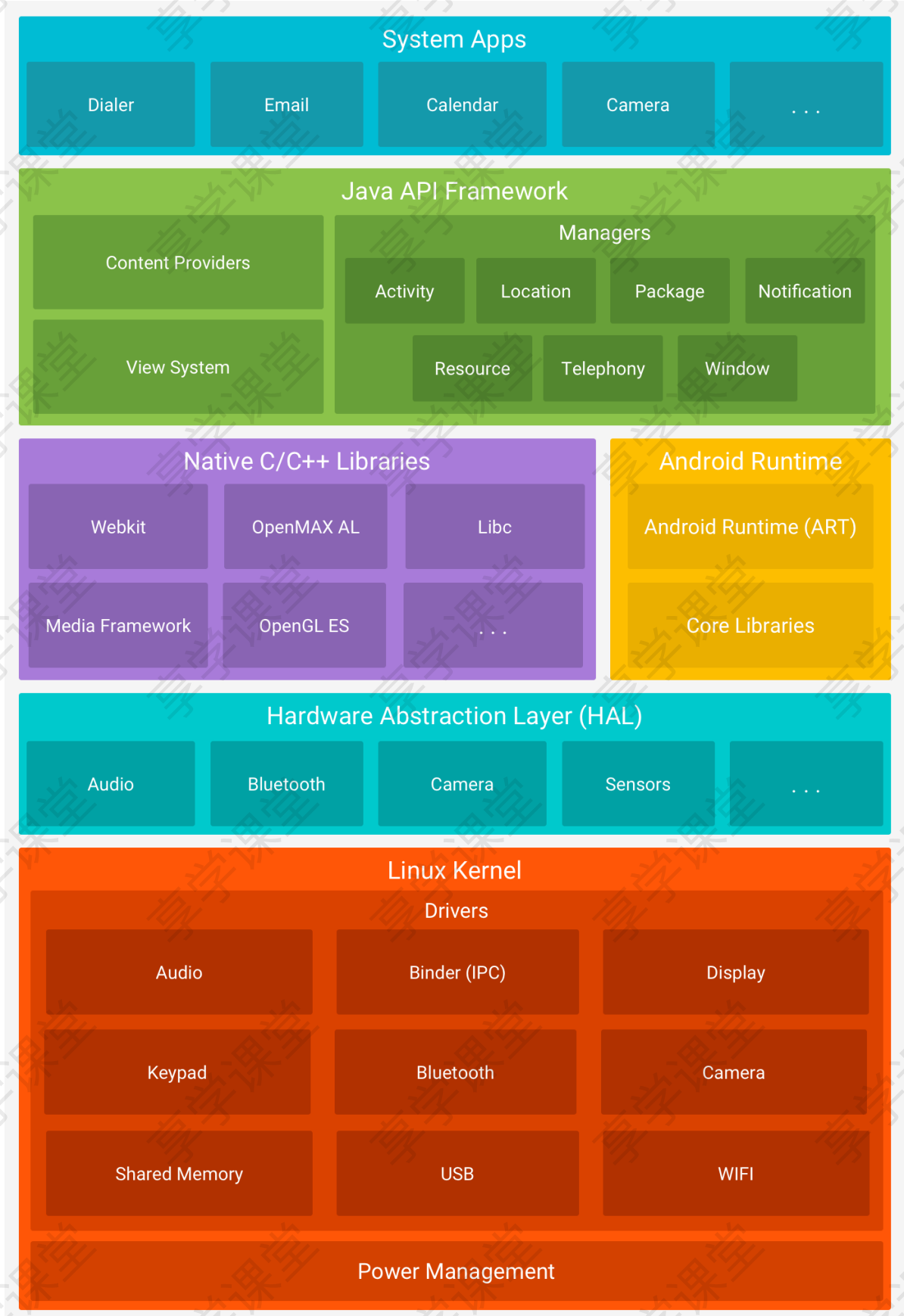
Android启动概览

众所周知，Android是谷歌开发的一款基于Linux的开源操作系统，下图所示为 Android 平台的主要组件



1. Linux 内核

   Android 平台的基础是 Linux 内核。例如，Android Runtime (ART) 依靠 Linux 内核来执行底层功

能，例如线程和低层内存管理。

使用 Linux 内核可让 Android 利用主要安全功能，并且允许设备制造商为著名的内核开发硬件驱动程序。

2. 硬件抽象层 (HAL)
   硬件抽象层 (HAL) 提供标准界面，向更高级别的 Java API 框架显示设备硬件功能。HAL 包含多个库模块，其中每个模块都为特定类型的硬件组件实现一个界面，例如相机或蓝牙模块。当框架 API 要求访问设备硬件时，Android 系统将为该硬件组件加载库模块。
3. Android Runtime
   对于运行 Android 5.0（API 级别 21）或更高版本的设备，每个应用都在其自己的进程中运行，并且有其自己的 Android Runtime (ART) 实例。ART 编写为通过执行 DEX 文件在低内存设备上运行多个虚拟机，DEX 文件是一种专为 Android 设计的字节码格式，经过优化，使用的内存很少。编译工具链（例如 Jack）将 Java 源代码编译为 DEX 字节码，使其可在 Android 平台上运行。

ART 的部分主要功能包括：

- 预先 (AOT) 和即时 (JIT) 编译
- 优化的垃圾回收 (GC)
- 在 Android 9（API 级别 28）及更高版本的系统中，支持将应用软件包中的 Dalvik Executable 格式 (DEX) 文件转换为更紧凑的机器代码。
- 更好的调试支持，包括专用采样分析器、详细的诊断异常和崩溃报告，并且能够设置观察点以监控特定字段
  在 Android 版本 5.0（API 级别 21）之前，Dalvik 是 Android Runtime。如果您的应用在 ART 上运行效果很好，那么它应该也可在 Dalvik 上运行，但反过来不一定。

Android 还包含一套核心运行时库，可提供 Java API 框架所使用的 Java 编程语言中的大部分功能，包括一些 Java 8 语言功能。

4. 原生 C/C++ 库
   许多核心 Android 系统组件和服务（例如 ART 和 HAL）构建自原生代码，需要以 C 和 C++ 编写的原生库。Android 平台提供 Java 框架 API 以向应用显示其中部分原生库的功能。例如，您可以通过 Android 框架的 Java OpenGL API 访问 OpenGL ES，以支持在应用中绘制和操作 2D 和 3D 图形。

如果开发的是需要 C 或 C++ 代码的应用，可以使用 Android NDK 直接从原生代码访问某些原生平台库。

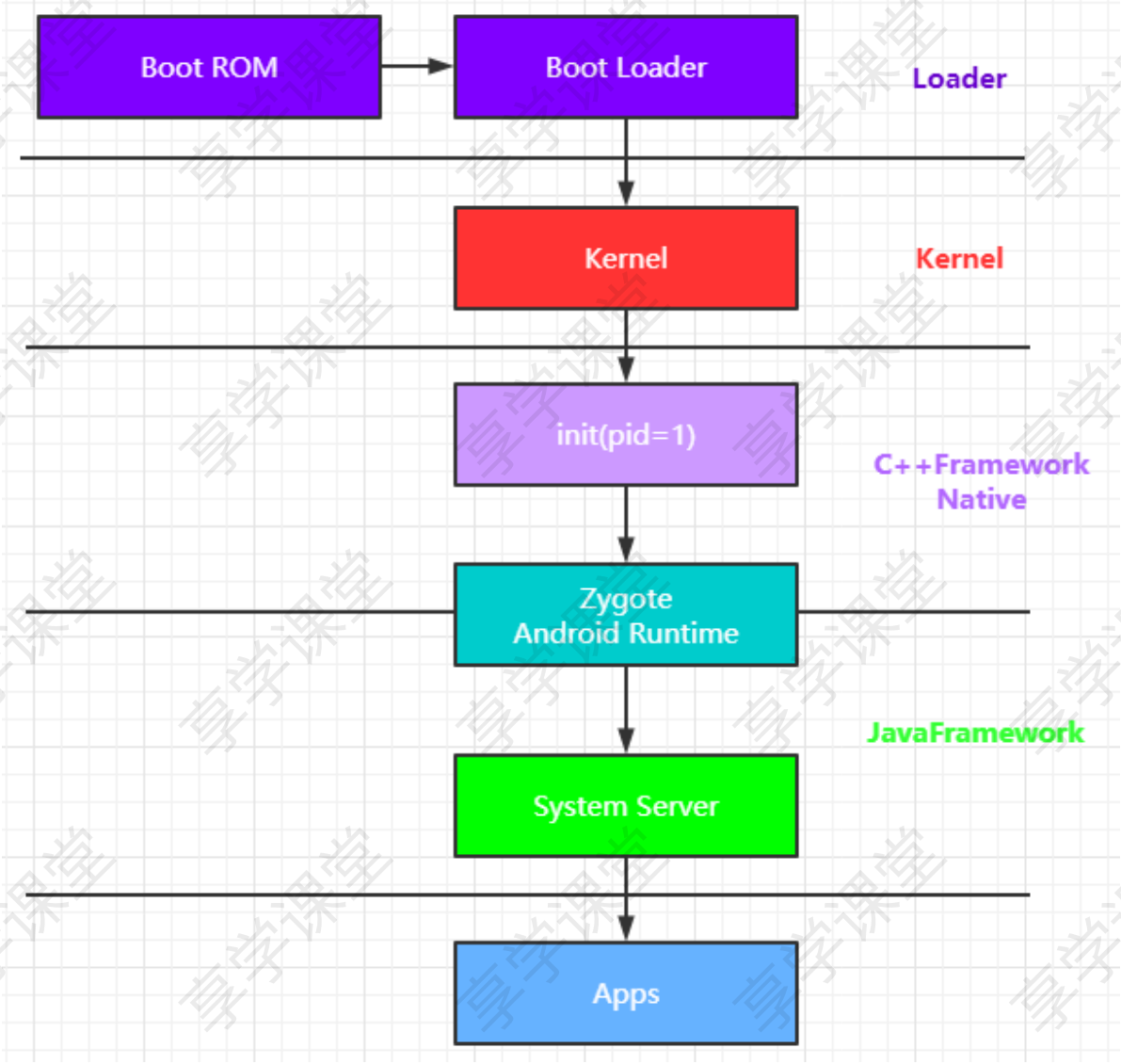
5. Java API 框架
   您可通过以 Java 语言编写的 API 使用 Android OS 的整个功能集。这些 API 形成创建 Android 应用所需的构建块，它们可简化核心模块化系统组件和服务的重复使用，包括以下组件和服务：

- 丰富、可扩展的视图系统，可用以构建应用的 UI，包括列表、网格、文本框、按钮甚至可嵌入的网络浏览器
- 资源管理器，用于访问非代码资源，例如本地化的字符串、图形和布局文件
- 通知管理器，可让所有应用在状态栏中显示自定义提醒
- Activity 管理器，用于管理应用的生命周期，提供常见的导航返回栈
- 内容提供程序，可让应用访问其他应用（例如“联系人”应用）中的数据或者共享其自己的数据
  开发者可以完全访问 Android 系统应用使用的框架 API。

6. 系统应用
   Android 随附一套用于电子邮件、短信、日历、互联网浏览和联系人等的核心应用。平台随附的应用与用户可以选择安装的应用一样，没有特殊状态。因此第三方应用可成为用户的默认网络浏览器、短信 Messenger 甚至默认键盘（有一些例外，例如系统的“设置”应用）。

系统应用可用作用户的应用，以及提供开发者可从其自己的应用访问的主要功能。例如，如果您的应用要发短信，您无需自己构建该功能，可以改为调用已安装的短信应用向您指定的接收者发送消息。

接下来，我们来看下Android系统启动的大概流程，如下图所示：



- 第一步： 启动电源以及系统启动

当电源按下，引导芯片代码开始从预定义的地方（固化在ROM）开始执行。加载引导程序到RAM，然后执行

- 第二步：引导程序

引导程序是在Android操作系统开始运行前的一个小程序。引导程序是运行的第一个程序，因此它是针对特定的主板与芯片的。设备制造商要么使用很受欢迎的引导程序比如redboot、uboot、qi bootloader或者开发自己的引导程序，它不是Android操作系统的一部分。引导程序是OEM厂商或者运营商加锁和限制的地方。
引导程序分两个阶段执行。
第一个阶段，检测外部的RAM以及加载对第二阶段有用的程序；
第二阶段，引导程序设置网络、内存等等。这些对于运行内核是必要的，为了达到特殊的目标，引导程序可以根据配置参数或者输入数据设置内核。
Android引导程序可以在\bootable\bootloader\legacy\usbloader找到。传统的加载器包含两个文件，需要在这里说明：
init.s初始化堆栈，清零BBS段，调用main.c的_main()函数；
main.c初始化硬件（闹钟、主板、键盘、控制台），创建linux标签

- 第三步：内核

Android内核与桌面linux内核启动的方式差不多。内核启动时，设置缓存、被保护存储器、计划列表，加载驱动。当内核完成系统设置，它首先在系统文件中寻找"init"文件，然后启动root进程或者系统的第一个进程

- 第四步：init进程

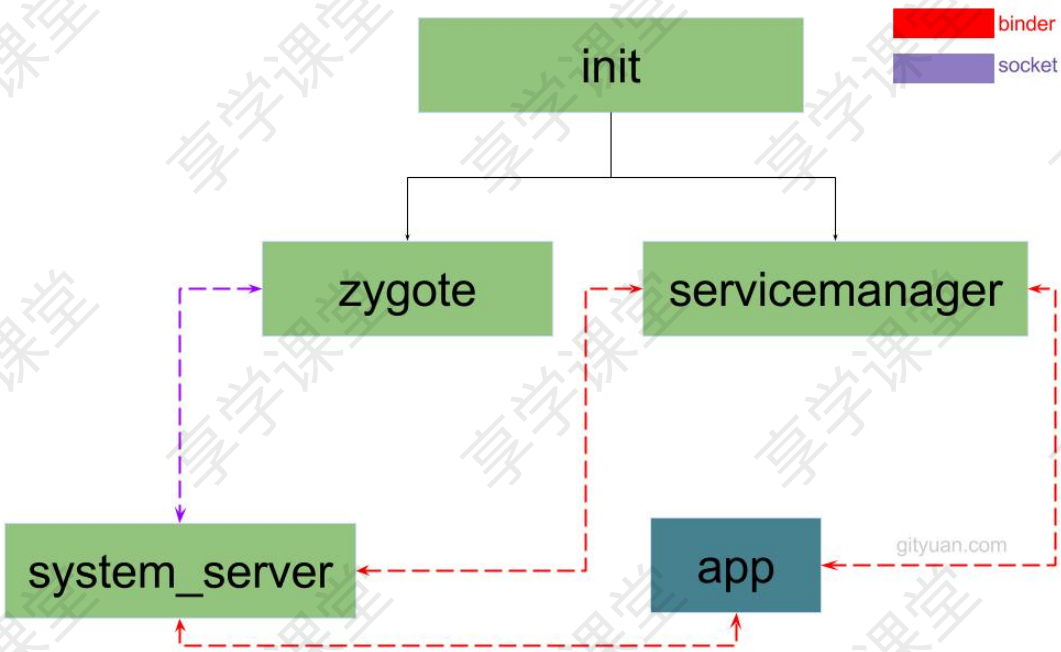
init进程是Linux系统中用户空间的第一个进程，进程号固定为1。Kernel启动后，在用户空间启动init进程，并调用init中的main()方法执行init进程的职责。

- 第五步：启动Lancher App

init进程分析

其中init进程是Android系统中及其重要的第一个进程，接下来我们来看下init进程注意做了些什么

1. 创建和挂载启动所需要的文件目录
2. 初始化和启动属性服务
3. 解析init.rc配置文件并**启动Zygote进程**




```
// \system\core\init\init.cpp main()  L545
/*
 * 1.C++中主函数有两个参数，第一个参数argc表示参数个数，第二个参数是参数列表，也就是具体的参数
 * 2.init的main函数有两个其它入口，一是参数中有ueventd，进入ueventd_main,二是参数中有watchdogd，进入watchdogd_main
 */
int main(int argc, char** argv) {
    /*
     * 1.strcmp是String的一个函数，比较字符串，相等返回0
     * 2.C++中0也可以表示false
     * 3.basename是C库中的一个函数，得到特定的路径中的最后一个'/'后面的内容，
     * 比如/sdcard/miui_recovery/backup，得到的结果是backup
     */
    if (!strcmp(basename(argv[0]), "ueventd")) {//当argv[0]的内容为ueventd时，strcmp的值为0,! strcmp为1
    //1表示true，也就执行ueventd_main,ueventd主要是负责设备节点的创建、权限设定等一些列工作
```

```
        return ueventd_main(argc, argv);
    }

    if (!strcmp(basename(argv[0]), "watchdogd")) {//watchdogd俗称看门狗，用于系统出问题时重启系统
        return watchdogd_main(argc, argv);
    }

    if (argc > 1 && !strcmp(argv[1], "subcontext")) {
        InitKernelLogging(argv);
        const BuiltinFunctionMap function_map;
        return SubcontextMain(argc, argv, &function_map);
    }

    if (REBOOT_BOOTLOADER_ON_PANIC) {
        InstallRebootSignalHandlers();//初始化重启系统的处理信号，内部通过 sigaction 注册信号，当监听到该信号时重启系统
    }

    bool is_first_stage = (getenv("INIT_SECOND_STAGE") == nullptr);//查看是否有环境变量INIT_SECOND_STAGE
    /*
     * 1.init的main方法会执行两次，由is_first_stage控制,first_stage就是第一阶段要做的事
     */
    if (is_first_stage) {
        boot_clock::time_point start_time = boot_clock::now();

        // Clear the umask.
        umask(0);//清空文件权限

        clearenv();
        setenv("PATH", _PATH_DEFPATH, 1);
        // Get the basic filesystem setup we need put together in the initramdisk
        // on / and then we'll let the rc file figure out the rest.
        //mount是用来挂载文件系统的，mount属于Linux系统调用
        mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755");
        mkdir("/dev/pts", 0755);//创建目录，第一个参数是目录路径，第二个是读写权限
        mkdir("/dev/socket", 0755);
        mount("devpts", "/dev/pts", "devpts", 0, NULL);
        #define MAKE_STR(x) __STRING(x)
        mount("proc", "/proc", "proc", 0, "hidepid=2,gid="
MAKE_STR(AID_READPROC));
        // Don't expose the raw commandline to unprivileged processes.
        chmod("/proc/cmdline", 0440);//用于修改文件/目录的读写权限
        gid_t groups[] = { AID_READPROC };
        setgroups(arraysize(groups), groups);// 用来将list 数组中所标明的组加入到目前进程的组设置中
        mount("sysfs", "/sys", "sysfs", 0, NULL);
        mount("selinuxfs", "/sys/fs/selinux", "selinuxfs", 0, NULL);
        //mknod用于创建Linux中的设备文件
        mknod("/dev/kmsg", S_IFCHR | 0600, makedev(1, 11));

        if constexpr (WORLD_WRITABLE_KMSG) {
            mknod("/dev/kmsg_debug", S_IFCHR | 0622, makedev(1, 11));
        }

```

```
66          mknod("/dev/random", S_IFCHR | 0666, makedev(1, 8));
67          mknod("/dev/urandom", S_IFCHR | 0666, makedev(1, 9));
68
69          // Mount staging areas for devices managed by vold
70          // See storage config details at
http://source.android.com/devices/storage/
71          mount("tmpfs", "/mnt", "tmpfs", MS_NOEXEC | MS_NOSUID | MS_NODEV,
72                "mode=0755,uid=0,gid=1000");
73          // /mnt/vendor is used to mount vendor-specific partitions that can
not be
74          // part of the vendor partition, e.g. because they are mounted
read-write.
75          mkdir("/mnt/vendor", 0755);
76
77          // Now that tmpfs is mounted on /dev and we have /dev/kmsg, we can
actually
78          // talk to the outside world...
79          InitKernelLogging(argv);//将标准输入输出重定向到"/sys/fs/selinux/null"
80
81          LOG(INFO) << "init first stage started!";
82
83          if (!DoFirstStageMount()) {
84              LOG(FATAL) << "Failed to mount required partitions early ...";
85          }
86
87          //Avb即Android Verfied boot,功能包括Secure Boot, verfying boot 和 dm-
verity,
88          //原理都是对二进制文件进行签名，在系统启动时进行认证，确保系统运行的是合法的二进
制镜像文件。
89          //其中认证的范围涵盖：bootloader，boot.img，system.img
90          SetInitAvbVersionInRecovery();//在刷机模式下初始化avb的版本,不是刷机模式
直接跳过
91
92          // Enable seccomp if global boot option was passed (otherwise it is
enabled in zygote).
93          global_seccomp();
94
95          // Set up SELinux, loading the SELinux policy.
96          SelinuxSetupKernelLogging();
97          SelinuxInitialize();//加载SELinux policy，也就是安全策略，
98
99          // We're in the kernel domain, so re-exec init to transition to the
init domain now
100         // that the SELinux policy has been loaded.
101         /*
102          * 1.这句英文大概意思是，我们执行第一遍时是在kernel domain，所以要重新执行
init文件，切换到init domain，
103          * 这样SELinux policy才已经加载进来了
104          * 2.后面的security_failure函数会调用panic重启系统
105          */
106         if (selinux_android_restorecon("/init", 0) == -1) {
107             PLOG(FATAL) << "restorecon failed of /init failed";
108         }
109
110         setenv("INIT_SECOND_STAGE", "true", 1);
111
112         static constexpr uint32_t kNanosecondsPerMillisecond = 1e6;
```

```
        uint64_t start_ms = start_time.time_since_epoch().count() /
kNanosecondsPerMillisecond;
        setenv("INIT_STARTED_AT", std::to_string(start_ms).c_str(), 1);

        char* path = argv[0];
        char* args[] = { path, nullptr };
        execv(path, args);//重新执行main方法，进入第二阶段

        // execv() only returns if an error happened, in which case we
        // panic and never fall through this conditional.
        PLOG(FATAL) << "execv(\"" << path << "\") failed";
    }

    // At this point we're in the second stage of init.
    InitKernelLogging(argv);
    LOG(INFO) << "init second stage started!";

    // Set up a session keyring that all processes will have access to. It
    // will hold things like FBE encryption keys. No process should
override
    // its session keyring.
    keyctl_get_keyring_ID(KEY_SPEC_SESSION_KEYRING, 1);

    // Indicate that booting is in progress to background fw loaders, etc.
    close(open("/dev/.booting", O_WRONLY | O_CREAT | O_CLOEXEC, 0000));

    property_init();//初始化属性系统，并从指定文件读取属性

    // If arguments are passed both on the command line and in DT,
    // properties set in DT always have priority over the command-line
ones.
    //接下来的一系列操作都是从各个文件读取一些属性，然后通过property_set设置系统属性

    // If arguments are passed both on the command line and in DT,
    // properties set in DT always have priority over the command-line
ones.
    /*
     * 1.这句英文的大概意思是，如果参数同时从命令行和DT传过来，DT的优先级总是大于命令行
的
     * 2.DT即device-tree，中文意思是设备树，这里面记录自己的硬件配置和系统运行参数，参
考http://www.wowotech.net/linux_kenrel/why-dt.html
     */
    process_kernel_dt();//处理DT属性
    process_kernel_cmdline();//处理命令行属性

    // Propagate the kernel variables to internal variables
    // used by init as well as the current required properties.
    export_kernel_boot_props();//处理其他的一些属性

    // Make the time that init started available for bootstat to log.
    property_set("ro.boottime.init", getenv("INIT_STARTED_AT"));
    property_set("ro.boottime.init.selinux", getenv("INIT_SELINUX_TOOK"));

    // Set libavb version for Framework-only OTA match in Treble build.
    const char* avb_version = getenv("INIT_AVB_VERSION");
    if (avb_version) property_set("ro.boot.avb_version", avb_version);

    // Clean up our environment.
```

```
    unsetenv("INIT_SECOND_STAGE");//清空这些环境变量，因为之前都已经存入到系统属性中去了
    unsetenv("INIT_STARTED_AT");
    unsetenv("INIT_SELINUX_TOOK");
    unsetenv("INIT_AVB_VERSION");

    // Now set up SELinux for second stage.
    SelinuxSetupKernelLogging();
    SelabelInitialize();
    SelinuxRestoreContext();

    epoll_fd = epoll_create1(EPOLL_CLOEXEC);//创建epoll实例，并返回epoll的文件描述符
    if (epoll_fd == -1) {
        PLOG(FATAL) << "epoll_create1 failed";
    }

    sigchld_handler_init();//主要是创建handler处理子进程终止信号，创建一个匿名socket并注册到epoll进行监听

    if (!IsRebootCapable()) {
        // If init does not have the CAP_SYS_BOOT capability, it is running in a container.
        // In that case, receiving SIGTERM will cause the system to shut down.
        InstallSigtermHandler();
    }

    property_load_boot_defaults();//从文件中加载一些属性，读取usb配置
    export_oem_lock_status();//设置ro.boot.flash.locked 属性
    start_property_service();//开启一个socket监听系统属性的设置
    set_usb_controller();//设置sys.usb.controller 属性

    const BuiltinFunctionMap function_map;//方法映射  “class_start”->"do_class_start"
     /*
     * 1.C++中::表示静态方法调用，相当于java中static的方法
     */
    Action::set_function_map(&function_map);//将function_map存放到Action中作为成员属性

    subcontexts = InitializeSubcontexts();

    ActionManager& am = ActionManager::GetInstance();
    ServiceList& sm = ServiceList::GetInstance();

    LoadBootScripts(am, sm);//解析xxx.rc

    // Turning this on and letting the INFO logging be discarded adds 0.2s to
    // Nexus 9 boot time, so it's disabled by default.
    if (false) DumpState();//打印一些当前Parser的信息，默认是不执行的

    am.QueueEventTrigger("early-init");//QueueEventTrigger用于触发Action,这里触发 early-init事件

    // Queue an action that waits for coldboot done so we know ueventd has set up all of /dev...
```

```
    am.QueueBuiltinAction(wait_for_coldboot_done_action,
"wait_for_coldboot_done");//QueueBuiltinAction用于添加Action，第一个参数是
Action要执行的Command,第二个是Trigger
    // ... so that we can start queuing up actions that require stuff from
/dev.
    am.QueueBuiltinAction(MixHwrngIntoLinuxRngAction,
"MixHwrngIntoLinuxRng");
    am.QueueBuiltinAction(SetMmapRndBitsAction, "SetMmapRndBits");
    am.QueueBuiltinAction(SetKptrRestrictAction, "SetKptrRestrict");
    am.QueueBuiltinAction(keychord_init_action, "keychord_init");
    am.QueueBuiltinAction(console_init_action, "console_init");

    // Trigger all the boot actions to get us started.
    am.QueueEventTrigger("init");

    // Repeat mix_hwrng_into_linux_rng in case /dev/hw_random or
/dev/random
    // wasn't ready immediately after wait_for_coldboot_done
    am.QueueBuiltinAction(MixHwrngIntoLinuxRngAction,
"MixHwrngIntoLinuxRng");

    // Don't mount filesystems or start core system services in charger
mode.
    std::string bootmode = GetProperty("ro.bootmode", "");
    if (bootmode == "charger") {
        am.QueueEventTrigger("charger");
    } else {
        am.QueueEventTrigger("late-init");
    }

    // Run all property triggers based on current state of the properties.
    am.QueueBuiltinAction(queue_property_triggers_action,
"queue_property_triggers");

    while (true) {
        // By default, sleep until something happens.
        int epoll_timeout_ms = -1; //epoll超时时间，相当于阻塞时间

        if (do_shutdown && !shutting_down) {
            do_shutdown = false;
            if (HandlePowerctlMessage(shutdown_command)) {
                shutting_down = true;
            }
        }

        if (!(waiting_for_prop || Service::is_exec_service_running())) {
            am.ExecuteOneCommand();//执行一个command
        }
        /*
          * 1.waiting_for_prop和IsWaitingForExec都是判断一个Timer为不为空，相当
于一个标志位
          * 2.waiting_for_prop负责属性设置，IsWaitingForExe负责service运行
          * 3.当有属性设置或Service开始运行时，这两个值就不为空，直到执行完毕才置为空
          * 4.其实这两个判断条件主要作用就是保证属性设置和service启动的完整性，也可以
说是为了同步
          */
        if (!(waiting_for_prop || Service::is_exec_service_running())) {
            if (!shutting_down) {
```

```
                auto next_process_restart_time = RestartProcesses();//重启服务

                // If there's a process that needs restarting, wake up in time for that.
                if (next_process_restart_time) {
                    epoll_timeout_ms = std::chrono::ceil<std::chrono::milliseconds>(
                                               *next_process_restart_time - boot_clock::now())
                                               .count();
                    if (epoll_timeout_ms < 0) epoll_timeout_ms = 0;
                }
            }

            // If there's more work to do, wake up again immediately.
            if (am.HasMoreCommands()) epoll_timeout_ms = 0;//当还有命令要执行时，将epoll_timeout_ms设置为0
        }

        epoll_event ev;
         /*
         * 1.epoll_wait与epoll_create1、epoll_ctl是一起使用的
         * 2.epoll_create1用于创建epoll的文件描述符，epoll_ctl、epoll_wait都把它创建的fd作为第一个参数传入
         * 3.epoll_ctl用于操作epoll，EPOLL_CTL_ADD：注册新的fd到epfd中，EPOLL_CTL_MOD：修改已经注册的fd的监听事件，EPOLL_CTL_DEL：从epfd中删除一个fd；
         * 4.epoll_wait用于等待事件的产生，epoll_ctl调用EPOLL_CTL_ADD时会传入需要监听什么类型的事件，
         *   比如EPOLLIN表示监听fd可读，当该fd有可读的数据时，调用epoll_wait经过epoll_timeout_ms时间就会把该事件的信息返回给&ev
         */
        int nr = TEMP_FAILURE_RETRY(epoll_wait(epoll_fd, &ev, 1, epoll_timeout_ms));
        if (nr == -1) {
            PLOG(ERROR) << "epoll_wait failed";
        } else if (nr == 1) {
            ((void (*)()) ev.data.ptr)();//当有event返回时，取出ev.data.ptr（之前epoll_ctl注册时的回调函数），直接执行
            //在signal_handler_init和start_property_service有注册两个fd的监听，一个用于监听SIGCHLD(子进程结束信号)，一个用于监听属性设置
        }
    }

    return 0;
}
```

init.rc解析

init.rc是一个非常重要的配置文件，它是由Android初始化语言（Android Init Language）编写的脚本，它主要包含五种类型语句：Action（Action中包含了一系列的Command）、Commands（init语言中的命令）、Services（由init进程启动的服务）、Options（对服务进行配置的选项）和Import（引入其他配置文件）。init.rc的配置代码如下所示。

```
# \system\core\rootdir\init.rc
on init # L41
```

```
    sysclktz 0

    # Mix device-specific information into the entropy pool
    copy /proc/cmdline /dev/urandom
    copy /default.prop /dev/urandom
    ...

on <trigger> [&& <trigger>]* //设置触发器
     <command>
     <command> //动作触发之后要执行的命令

service <name> <pathname> [ <argument> ]* //<service的名字><执行程序路径><传递参数>
     <option>  //Options是Services的参数配置. 它们影响Service如何运行及运行时机
     group <groupname> [ <groupname>\* ] //在启动Service前将group改为第一个groupname,第一个groupname是必须有的,
     //默认值为root（或许默认值是无），第二个groupname可以不设置，用于追加组（通过setgroups）
     priority <priority> //设置进程优先级. 在-20～19之间，默认值是0,能过setpriority实现
     socket <name> <type> <perm> [ <user> [ <group> [ <seclabel> ] ] ]//创建一个unix域的socket,名字叫/dev/socket/name , 并将fd返回给Service. type 只能是 "dgram", "stream" or "seqpacket".
       ...
```

Action

Action： 通过触发器trigger，即以on开头的语句来决定执行相应的service的时机，具体有如下时机：

- on early-init; 在初始化早期阶段触发；
- on init; 在初始化阶段触发；
- on late-init; 在初始化晚期阶段触发；
- on boot/charger： 当系统启动/充电时触发，还包含其他情况，此处不一一列举；
- on property:=: 当属性值满足条件时触发

Service

服务Service，以 service开头，由init进程启动，一般运行在init的一个子进程，所以启动service前需要判断对应的可执行文件是否存在。init生成的子进程，定义在rc文件，其中每一个service在启动时会通过fork方式生成子进程。

例如： `service servicemanager /system/bin/servicemanager` 代表的是服务名为servicemanager，服务执行的路径为/system/bin/servicemanager。

Command

下面列举常用的命令

- class_start <service_class_name>： 启动属于同一个class的所有服务；
- start <service_name>： 启动指定的服务，若已启动则跳过；
- stop <service_name>： 停止正在运行的服务
- setprop ： 设置属性值
- mkdir： 创建指定目录
- symlink <sym_link>： 创建连接到的<sym_link>符号链接；
- write ： 向文件path中写入字符串；
- exec： fork并执行，会阻塞init进程直到程序完毕；
- exprot ： 设定环境变量；

- loglevel： 设置log级别

Options

Options是Service的可选项，与service配合使用

- disabled: 不随class自动启动，只有根据service名才启动；
- oneshot: service退出后不再重启；
- user/group： 设置执行服务的用户/用户组，默认都是root；
- class：设置所属的类名，当所属类启动/退出时，服务也启动/停止，默认为default；
- onrestart:当服务重启时执行相应命令；
- socket: 创建名为 `/dev/socket/<name>` 的socket
- critical: 在规定时间内该service不断重启，则系统会重启并进入恢复模式

default: 意味着disabled=false，oneshot=false，critical=false。

```
service zygote /system/bin/app_process64 -Xzygote /system/bin --zygote --
start-system-server
    class main
    priority -20
    user root
    group root readproc reserved_disk
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart audioserver
    onrestart restart cameraserver
    onrestart restart media
    onrestart restart netd
    onrestart restart wificond
    writepid /dev/cpuset/foreground/tasks
```

service解析流程

```
// \system\core\init\init.cpp LoadBootScripts()  L110
static void LoadBootScripts(ActionManager& action_manager, ServiceList&
service_list) {
    Parser parser = CreateParser(action_manager, service_list);//创建解析器

    std::string bootscript = GetProperty("ro.boot.init_rc", "");
    if (bootscript.empty()) {
        parser.ParseConfig("/init.rc");
        if (!parser.ParseConfig("/system/etc/init")) {
            late_import_paths.emplace_back("/system/etc/init");
        }
        if (!parser.ParseConfig("/product/etc/init")) {
            late_import_paths.emplace_back("/product/etc/init");
        }
        if (!parser.ParseConfig("/odm/etc/init")) {
            late_import_paths.emplace_back("/odm/etc/init");
        }
        if (!parser.ParseConfig("/vendor/etc/init")) {
            late_import_paths.emplace_back("/vendor/etc/init");
        }
    } else {
        parser.ParseConfig(bootscript);//开始解析
    }
```

```
}
```

```
// \system\core\init\init.cpp CreateParser()  L100
Parser CreateParser(ActionManager& action_manager, ServiceList&
service_list) {
    Parser parser;

    parser.AddSectionParser("service", std::make_unique<ServiceParser>
(&service_list, subcontexts));//service解析
    parser.AddSectionParser("on", std::make_unique<ActionParser>
(&action_manager, subcontexts));
    parser.AddSectionParser("import", std::make_unique<ImportParser>
(&parser));

    return parser;
}
```

```
// \system\core\init\parser.cpp ParseData() L 42
void Parser::ParseData(const std::string& filename, const std::string& data,
size_t* parse_errors) {
    ...

    for (;;) {
        switch (next_token(&state)) {
            case T_EOF:
                end_section();
                return;
            case T_NEWLINE:
                ...
                if (section_parsers_.count(args[0])) {
                    end_section();
                    section_parser = section_parsers_[args[0]].get();
                    section_start_line = state.line;
                    if (auto result =
                            section_parser->ParseSection(std::move(args),
filename, state.line);// L95
                        !result) {
                        (*parse_errors)++;
                        LOG(ERROR) << filename << ": " << state.line << ": "
<< result.error();
                        section_parser = nullptr;
                    }
                } else if (section_parser) {
                    if (auto result = section_parser-
>ParseLineSection(std::move(args), state.line);// L102
                        !result) {
                        (*parse_errors)++;
                        LOG(ERROR) << filename << ": " << state.line << ": "
<< result.error();
                    }
                }
                args.clear();
                break;
            case T_TEXT:
                args.emplace_back(state.text);
                break;
```

```
        }
    }
}
```

```
// \system\core\init\service.cpp ParseSection() L1180
Result<Success> ServiceParser::ParseSection(std::vector<std::string>&& args,
                                            const std::string& filename, int
line) {
    if (args.size() < 3) {
        return Error() << "services must have a name and a program";
    }

    const std::string& name = args[1];
    if (!IsValidName(name)) {
        return Error() << "invalid service name '" << name << "'";
    }

    Subcontext* restart_action_subcontext = nullptr;
    if (subcontexts_) {
        for (auto& subcontext : *subcontexts_) {
            if (StartsWith(filename, subcontext.path_prefix())) {
                restart_action_subcontext = &subcontext;
                break;
            }
        }
    }

    std::vector<std::string> str_args(args.begin() + 2, args.end());
    service_ = std::make_unique<Service>(name, restart_action_subcontext,
str_args);//构建出一个service对象
    return Success();
}
```

```
// \system\core\init\service.cpp ParseLineSection() L1206
Result<Success> ServiceParser::ParseLineSection(std::vector<std::string>&&
args, int line) {
    return service_ ? service_->ParseLine(std::move(args)) : Success();
}
```

```
// \system\core\init\service.cpp EndSection() L1210
Result<Success> ServiceParser::EndSection() {
    if (service_) {
        Service* old_service = service_list_->FindService(service_->name());
        if (old_service) {
            if (!service_->is_override()) {
                return Error() << "ignored duplicate definition of service
'" << service_->name()
                               << "'";
            }

            service_list_->RemoveService(*old_service);
            old_service = nullptr;
        }

        service_list_->AddService(std::move(service_));
```

```
    }

    return Success();
}
```

```
// \system\core\init\service.cpp AddService() L1082
void ServiceList::AddService(std::unique_ptr<Service> service) {
    services_.emplace_back(std::move(service));
}
```

上面解析完成后，接下来就是启动Service,这里我们以启动Zygote来分析

```
# \system\core\rootdir\init.rc L680
on nonencrypted
    class_start main //class_start是一个命令，通过do_class_start函数处理
    class_start late_start
```

```
// \system\core\init\builtins..cpp do_class_start() L101
static Result<Success> do_class_start(const BuiltinArguments& args) {
    // Starting a class does not start services which are explicitly disabled.
    // They must  be started individually.
    for (const auto& service : ServiceList::GetInstance()) {
        if (service->classnames().count(args[1])) {
            if (auto result = service->StartIfNotDisabled(); !result) {
                LOG(ERROR) << "Could not start service '" << service->name()
                           << "' as part of class '" << args[1] << "': " << result.error();
            }
        }
    }
    return Success();
}
```

```
// \system\core\init\service.cpp StartIfNotDisabled() L977
Result<Success> Service::StartIfNotDisabled() {
    if (!(flags_ & SVC_DISABLED)) {
        return Start();
    } else {
        flags_ |= SVC_DISABLED_START;
    }
    return Success();
}
```

```
// \system\core\init\service.cpp Start() L785
Result<Success> Service::Start() {
   //如果service已经运行，则不启动
    if (flags_ & SVC_RUNNING) {
        if ((flags_ & SVC_ONESHOT) && disabled) {
            flags_ |= SVC_RESTART;
        }
        // It is not an error to try to start a service that is already running.
        return Success();
```

```
    }

    ...
    //判断需要启动的service的对应的执行文件是否存在，不存在则不启动service
    struct stat sb;
    if (stat(args_[0].c_str(), &sb) == -1) {
        flags_ |= SVC_DISABLED;
        return ErrnoError() << "Cannot find '" << args_[0] << "'";
    }

    std::string scon;
    if (!seclabel_.empty()) {
        scon = seclabel_;
    } else {
        auto result = ComputeContextFromExecutable(args_[0]);
        if (!result) {
            return result.error();
        }
        scon = *result;
    }

    LOG(INFO) << "starting service '" << name_ << "'...";
   //如果子进程没有启动，则调用fork函数创建子进程
    pid_t pid = -1;
    if (namespace_flags_) {
        pid = clone(nullptr, nullptr, namespace_flags_ | SIGCHLD, nullptr);
    } else {
        pid = fork();
    }

    if (pid == 0) {//当期代码逻辑在子进程中运行
        umask(077);


  //调用execv函数，启动sevice子进程
        if (!ExpandArgsAndExecv(args_)) {
            PLOG(ERROR) << "cannot execve('" << args_[0] << "')";
        }

        _exit(127);
    }
    return Success();
}
```

Zygote概叙

Zygote中文翻译为“受精卵”，正如其名，它主要用于孵化子进程。在Android系统中有以下两种程序：java应用程序，主要基于ART虚拟机，所有的应用程序apk都属于这类native程序，也就是利用C或C++语言开发的程序，如bootanimation。所有的Java应用程序进程及系统服务SystemServer进程都由Zygote进程通过Linux的fork()函数孵化出来的，这也就是为什么把它称为Zygote的原因，因为他就像一个受精卵，孵化出无数子进程，而native程序则由Init程序创建启动。Zygote进程最初的名字不是“zygote”而是“app_process”，这个名字是在Android.mk文件中定义的

Zgyote是Android中的第一个art虚拟机，他通过socket的方式与其他进程进行通信。这里的“其他进程”其实主要是系统进程——SystemServer

> Zygote是一个C/S模型，Zygote进程作为服务端，它主要负责创建Java虚拟机，加载系统资源，启动SystemServer进程，以及在后续运行过程中启动普通的应用程序，其他进程作为客户端向它发出“孵化”请求，而Zygote接收到这个请求后就“孵化”出一个新的进程。比如，当点击Launcher里的应用程序图标去启动一个新的应用程序进程时，这个请求会到达框架层的核心服务ActivityManagerService中，当AMS收到这个请求后，它通过调用Process类发出一个“孵化”子进程的Socket请求，而Zygote监听到这个请求后就立刻fork一个新的进程出来

Zygote触发过程

1. init.zygoteXX.rc

```
import /init.${ro.zygote}.rc
```

${ro.zygote} 会被替换成 ro.zyogte 的属性值，这个是由不同的硬件厂商自己定制的，

有四个值，

- zygote32： zygote 进程对应的执行程序是 app_process (纯 32bit 模式)
- zygote64： zygote 进程对应的执行程序是 app_process64 (纯 64bit 模式)
- zygote32_64： 启动两个 zygote 进程 (名为 zygote 和 zygote_secondary)，对应的执行程序分别是 app_process32 (主模式)
- zygote64_32： 启动两个 zygote 进程 (名为 zygote 和 zygote_secondary)，对应的执行程序分别是 app_process64 (主模式)、app_process32

2. start zygote
   位置：system\core\rootdir\init.rc 560

```
# It is recommended to put unnecessary data/ initialization from post-fs-data
# to start-zygote in device's init.rc to unblock zygote start.
on zygote-start && property:ro.crypto.state=unencrypted
    # A/B update verifier that marks a successful boot.
    exec_start update_verifier_nonencrypted
    start netd
    start zygote
    start zygote_secondary

on zygote-start && property:ro.crypto.state=unsupported
    # A/B update verifier that marks a successful boot.
    exec_start update_verifier_nonencrypted
    start netd
    start zygote
    start zygote_secondary

on zygote-start && property:ro.crypto.state=encrypted && property:ro.crypto.type=file
    # A/B update verifier that marks a successful boot.
    exec_start update_verifier_nonencrypted
    start netd
    start zygote
    start zygote_secondary
```

zygote-start 是在 on late-init 中触发的

```
# Mount filesystems and start core system services.
```

```
on late-init
    trigger early-fs

    # Mount fstab in init.{$device}.rc by mount_all command. Optional
parameter
    # '--early' can be specified to skip entries with 'latemount'.
    # /system and /vendor must be mounted by the end of the fs stage,
    # while /data is optional.
    trigger fs
    trigger post-fs

    # Mount fstab in init.{$device}.rc by mount_all with '--late' parameter
    # to only mount entries with 'latemount'. This is needed if '--early' is
    # specified in the previous mount_all command on the fs stage.
    # With /system mounted and properties form /system + /factory available,
    # some services can be started.
    trigger late-fs

    # Now we can mount /data. File encryption requires keymaster to decrypt
    # /data, which in turn can only be loaded when system properties are
present.
    trigger post-fs-data

    # Now we can start zygote for devices with file based encryption
    trigger zygote-start #
zygote-start 是在 on late-init 中触发的

    # Load persist properties and override properties (if enabled) from
/data.
    trigger load_persist_props_action

    # Remove a file to wake up anything waiting for firmware.
    trigger firmware_mounts_complete

    trigger early-boot
    trigger boot
```

```
    if (bootmode == "charger") {
        am.QueueEventTrigger("charger");
    } else {
        am.QueueEventTrigger("late-init");
    }
```

3. app_processXX
位置\frameworks\base\cmds\app_process\

```
app_process_src_files := \
    app_main.cpp \

LOCAL_SRC_FILES:= $(app_process_src_files)
...
LOCAL_MODULE:= app_process
LOCAL_MULTILIB := both
LOCAL_MODULE_STEM_32 := app_process32
LOCAL_MODULE_STEM_64 := app_process64
```

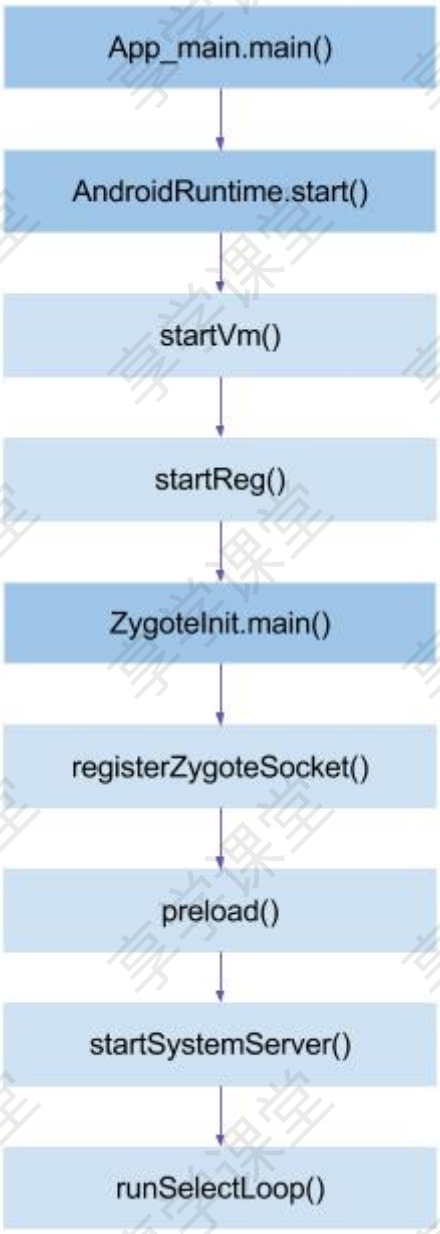
Zygote启动过程

位置\frameworks\base\cmds\app_process\app_main.cpp

在app_main.cpp的main函数中，主要做的事情就是参数解析. 这个函数有两种启动模式：

1. 一种是zygote模式，也就是初始化zygote进程，传递的参数有--start-system-server --socket-name=zygote，前者表示启动SystemServer，后者指定socket的名称
2. 一种是application模式，也就是启动普通应用程序，传递的参数有class名字以及class带的参数

两者最终都是调用AppRuntime对象的start函数，加载ZygoteInit或RuntimeInit两个Java类，并将之前整理的参数传入进去



app_process

```
// \frameworks\base\cmds\app_process\app_main.cpp main()  L280
if (strcmp(arg, "--zygote") == 0) {
            zygote = true;
            niceName = ZYGOTE_NICE_NAME;
        } else if (strcmp(arg, "--start-system-server") == 0) {
            startSystemServer = true;
        } else if (strcmp(arg, "--application") == 0) {
```

```
            application = true;
        }
 // L349
 if (zygote) {
       //这些Java的应用都是通过 AppRuntime.start（className)开始的
       //其实AppRuntime是AndroidRuntime的子类，它主要实现了几个回调函数，而start()方法是实现在AndroidRuntime这个方法类里
        runtime.start("com.android.internal.os.ZygoteInit", args, zygote);
    } else if (className) {
        runtime.start("com.android.internal.os.RuntimeInit", args, zygote);
    }
```

app_process 里面定义了三种应用程序类型：

1. Zygote: com.android.internal.os.ZygoteInit
2. System Server, 不单独启动，而是由Zygote启动
3. 其他指定类名的Java 程序

```
// \frameworks\base\core\jni\androidRuntime.cpp
class AppRuntime : public AndroidRuntime
{
public:
    AppRuntime(char* argBlockStart, const size_t argBlockLength)
        : AndroidRuntime(argBlockStart, argBlockLength)
        , mClass(NULL)
    {
    }

    void setClassNameAndArgs(const String8& className, int argc, char * const *argv) {
        mClassName = className;
        for (int i = 0; i < argc; ++i) {
             mArgs.add(String8(argv[i]));
        }
    }

    virtual void onVmCreated(JNIEnv* env)
    {
        if (mClassName.isEmpty()) {
            return; // Zygote. Nothing to do here.
        }

        /*
         * This is a little awkward because the JNI FindClass call uses the
         * class loader associated with the native method we're executing in.
         * If called in onStarted (from RuntimeInit.finishInit because we're
         * launching "am", for example), FindClass would see that we're calling
         * from a boot class' native method, and so wouldn't look for the class
         * we're trying to look up in CLASSPATH. Unfortunately it needs to,
         * because the "am" classes are not boot classes.
         *
         * The easiest fix is to call FindClass here, early on before we start
```

```
         * executing boot class Java code and thereby deny ourselves access
to
         * non-boot classes.
         */
        char* slashClassName = toSlashClassName(mClassName.string());
        mClass = env->FindClass(slashClassName);
        if (mClass == NULL) {
            ALOGE("ERROR: could not find class '%s'\n",
mClassName.string());
        }
        free(slashClassName);

        mClass = reinterpret_cast<jclass>(env->NewGlobalRef(mClass));
    }

    virtual void onStarted()
    {
        sp<ProcessState> proc = ProcessState::self();
        ALOGV("App process: starting thread pool.\n");
        proc->startThreadPool();

        AndroidRuntime* ar = AndroidRuntime::getRuntime();
        ar->callMain(mClassName, mClass, mArgs);

        IPCThreadState::self()->stopProcess();
        hardware::IPCThreadState::self()->stopProcess();
    }

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

    virtual void onExit(int code)
    {
        if (mClassName.isEmpty()) {
            // if zygote
            IPCThreadState::self()->stopProcess();
            hardware::IPCThreadState::self()->stopProcess();
        }

        AndroidRuntime::onExit(code);
    }


    String8 mClassName;
    Vector<String8> mArgs;
    jclass mClass;
};
}
```

什么是Runtime ?

> https://stackoverflow.com/questions/3900549/what-is-runtime
>
> As per Wikipedia: runtime library/run-time system.
>
> > In computer programming, a runtime library is a special program library used by a compiler, to implement functions built into a programming language, during the runtime (execution) of a computer program. This often includes functions for input and output, or for memory management.
>
> ---
>
> > A run-time system (also called runtime system or just runtime) is software designed to support the execution of computer programs written in some computer language. The run-time system contains implementations of basic low-level commands and may also implement higher-level commands and may support type checking, debugging, and even code generation and optimization. Some services of the run-time system are accessible to the programmer through an application programming interface, but other services (such as task scheduling and resource management) may be inaccessible.

归纳起来的意思就是，Runtime 是支撑程序运行的基础库，它是与语言绑定在一起的。比如：

- C Runtime：就是C standard lib, 也就是我们常说的libc。（有意思的是， Wiki会自动将“C runtime” 重定向到 “C Standard Library”).
- Java Runtime: 同样，Wiki将其重定向到” Java Virtual Machine”, 这里当然包括Java 的支撑类库（.jar).
- AndroidRuntime：显而易见，就是为Android应用运行所需的运行时环境。这个环境包括以下东东：
  - Dalvik VM: Android的Java VM, 解释运行Dex格式Java程序。每个进程运行一个虚拟机（什么叫运行虚拟机？说白了，就是一些C代码，不停的去解释Dex格式的二进制码(Bytecode)，把它们转成机器码(Machine code)，然后执行，当然，现在大多数的Java 虚拟机都支持JIT，也就是说，bytecode可能在运行前就已经被转换成机器码，从而大大提高了性能。过去一个普遍的认识是Java 程序比C，C++等静态编译的语言慢，但随着JIT的介入和发展，这个已经完全是过去时了，JIT的动态性运行允许虚拟机根据运行时环境，优化机器码的生成，在某些情况下，Java甚至可以比C/C++跑得更快，同时又兼具平台无关的特性，这也是为什么Java如今如此流行的原因之一吧）。
  - Android的Java 类库, 大部分来自于 Apache Hamony, 开源的Java API 实现，如 java.lang, java.util, java.net. 但去除了AWT, Swing 等部件。
  - JNI: C和Java互调的接口。
  - Libc: Android也有很多C代码，自然少不了libc，注意的是，Android的libc叫 bionic C

```
// \frameworks\base\core\jni\androidRuntime.cpp start() L1091
void AndroidRuntime::start(const char* className, const Vector<String8>&
options, bool zygote)
{
    ...
    JNIEnv* env;
    //JNI_CreateJavaVM L1015
    if (startVm(&mJavaVM, &env, zygote) != 0) {
        return;
    }

    onVmCreated(env);

    /*
     * Register android functions.
```

```
     */
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }
...
}
```

- Java虚拟机的启动大致做了以下一些事情：

1. 从property读取一系列启动参数。
2. 创建和初始化结构体全局对象（每个进程）gDVM，及对应与JavaVM和JNIEnv的内部结构体JavaVMExt, JNIEnvExt.
3. 初始化java虚拟机，并创建虚拟机线程
4. 注册系统的JNI，Java程序通过这些JNI接口来访问底层的资源。

```
    loadJniLibrary("javacore");
    loadJniLibrary("nativehelper");
```

5. 为Zygote的启动做最后的准备，包括设置SID/UID, 以及mount 文件系统
6. 返回JavaVM 给Native代码，这样它就可以向上访问Java的接口

```
// \frameworks\base\core\jni\androidRuntime.cpp startVm() L596
int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv, bool zygote)
{
 ...
 // L1015
   /*
     * Initialize the VM.
     *
     * The JavaVM* is essentially per-process, and the JNIEnv* is per-
thread.
     * If this call succeeds, the VM is ready, and we can start issuing
     * JNI calls.
     */
     //startVM的前半部分是在处理虚拟机的启动参数，处理完配置参数后，会调用libart.so提供
的一个接口：JNI_CreateJavaVM函数
    if (JNI_CreateJavaVM(pJavaVM, pEnv, &initArgs) < 0) {
        ALOGE("JNI_CreateJavaVM failed\n");
        return -1;
    }
 ...
```

```
// \art\runtime\java_vm_ext.cc JNI_CreateJavaVM() L1139
extern "C" jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void*
vm_args) {
  ScopedTrace trace(__FUNCTION__);
  const JavaVMInitArgs* args = static_cast<JavaVMInitArgs*>(vm_args);
  if (JavaVMExt::IsBadJniVersion(args->version)) {
    LOG(ERROR) << "Bad JNI version passed to CreateJavaVM: " << args-
>version;
    return JNI_EVERSION;
  }
  RuntimeOptions options;
  for (int i = 0; i < args->nOptions; ++i) {
    JavaVMOption* option = &args->options[i];
```

```
    options.push_back(std::make_pair(std::string(option->optionString),
option->extraInfo));
  }
  bool ignore_unrecognized = args->ignoreUnrecognized;
  //通过Runtime的create方法创建单例的Runtime对象
  if (!Runtime::Create(options, ignore_unrecognized)) {
    return JNI_ERR;
  }

  // Initialize native loader. This step makes sure we have
  // everything set up before we start using JNI.
  android::InitializeNativeLoader();

  Runtime* runtime = Runtime::Current();
  bool started = runtime->Start();
  if (!started) {
    delete Thread::Current()->GetJniEnv();
    delete runtime->GetJavaVM();
    LOG(WARNING) << "CreateJavaVM failed";
    return JNI_ERR;
  }

  *p_env = Thread::Current()->GetJniEnv();
  *p_vm = runtime->GetJavaVM();
  return JNI_OK;
}
```

首先通过Runtime的create方法创建单例的Runtime对象，runtime负责提供art虚拟机的运行时环境，然后调用其init方法来初始化虚拟机

```
// \art\runtime\runtime.cc Init() L1109
bool Runtime::Init(RuntimeArgumentMap&& runtime_options_in) {
...
// L1255 创建堆管理对象。
heap_ = new gc::Heap(runtime_options.GetOrDefault(Opt::MemoryInitialSize),
                     runtime_options.GetOrDefault(Opt::HeapGrowthLimit),
                     runtime_options.GetOrDefault(Opt::HeapMinFree),
                     runtime_options.GetOrDefault(Opt::HeapMaxFree),

runtime_options.GetOrDefault(Opt::HeapTargetUtilization),
                     foreground_heap_growth_multiplier,
                     runtime_options.GetOrDefault(Opt::MemoryMaximumSize),

runtime_options.GetOrDefault(Opt::NonMovingSpaceCapacity),
                     runtime_options.GetOrDefault(Opt::Image),

runtime_options.GetOrDefault(Opt::ImageInstructionSet),
                     // Override the collector type to CC if the read
barrier config.
                     kUseReadBarrier ? gc::kCollectorTypeCC :
xgc_option.collector_type_,
                     kUseReadBarrier ?
BackgroundGcOption(gc::kCollectorTypeCCBackground)
                                     :
runtime_options.GetOrDefault(Opt::BackgroundGc),
                     runtime_options.GetOrDefault(Opt::LargeObjectSpace),
```

```
20
runtime_options.GetOrDefault(Opt::LargeObjectThreshold),
21                      runtime_options.GetOrDefault(Opt::ParallelGCThreads),
22                      runtime_options.GetOrDefault(Opt::ConcGCThreads),
23                      runtime_options.Exists(Opt::LowMemoryMode),
24
runtime_options.GetOrDefault(Opt::LongPauseLogThreshold),
25
runtime_options.GetOrDefault(Opt::LongGCLogThreshold),
26                      runtime_options.Exists(Opt::IgnoreMaxFootprint),
27                      runtime_options.GetOrDefault(Opt::UseTLAB),
28                      xgc_option.verify_pre_gc_heap_,
29                      xgc_option.verify_pre_sweeping_heap_,
30                      xgc_option.verify_post_gc_heap_,
31                      xgc_option.verify_pre_gc_rosalloc_,
32                      xgc_option.verify_pre_sweeping_rosalloc_,
33                      xgc_option.verify_post_gc_rosalloc_,
34                      xgc_option.gcstress_,
35                      xgc_option.measure_,
36
runtime_options.GetOrDefault(Opt::EnableHSpaceCompactForOOM),
37
runtime_options.GetOrDefault(Opt::HSpaceCompactForOOMMinIntervalsMs));
38    ...
39    // L1408创建java虚拟机对象
40   std::string error_msg;
41   java_vm_ = JavaVMExt::Create(this, runtime_options, &error_msg);
42   if (java_vm_.get() == nullptr) {
43     LOG(ERROR) << "Could not initialize JavaVMExt: " << error_msg;
44     return false;
45   }
46
47   // Add the JniEnv handler.
48   // TODO Refactor this stuff.
49   java_vm_->AddEnvironmentHook(JNIEnvExt::GetEnvHandler);
50
51   Thread::Startup();
52
53   // ClassLinker needs an attached thread, but we can't fully attach a
thread without creating
54   // objects. We can't supply a thread group yet; it will be fixed later.
Since we are the main
55   // thread, we do not get a java peer.
56   // L1424 连接主线程
57   Thread* self = Thread::Attach("main", false, nullptr, false);
58   CHECK_EQ(self->GetThreadId(), ThreadList::kMainThreadId);
59   CHECK(self != nullptr);
60   ...
61   // L1437  创建类连接器
62   if (UNLIKELY(IsAotCompiler())) {
63     class_linker_ = new AotClassLinker(intern_table_);
64   } else {
65     class_linker_ = new ClassLinker(intern_table_);
66   }
67   if (GetHeap()->HasBootImageSpace()) {
68      //初始化类连接器
69     bool result = class_linker_->InitFromBootImage(&error_msg);
70     if (!result) {
```

```
      LOG(ERROR) << "Could not initialize from image: " << error_msg;
      return false;
    }
    ...
    }
  }
}
```

1. new gc::heap()，创建Heap对象，这是虚拟机管理对内存的起点。
2. new JavaVmExt(),创建Java虚拟机实例。
3. Thread::attach()，attach主线程
4. 创建ClassLinker
5. 初始化ClassLinker,成功attach到runtime环境后，创建ClassLinker实例负责管理java class

> 到这里，虚拟机的创建和初始化就完成了

```
//  \art\runtime\threed.cc Attach() L775
template <typename PeerAction>
Thread* Thread::Attach(const char* thread_name, bool as_daemon, PeerAction
peer_action) {
  Runtime* runtime = Runtime::Current();
  if (runtime == nullptr) {
    LOG(ERROR) << "Thread attaching to non-existent runtime: " <<
        ((thread_name != nullptr) ? thread_name : "(Unnamed)");
    return nullptr;
  }
  Thread* self;
  {
    MutexLock mu(nullptr, *Locks::runtime_shutdown_lock_);
    if (runtime->IsShuttingDownLocked()) {
      LOG(WARNING) << "Thread attaching while runtime is shutting down: " <<
          ((thread_name != nullptr) ? thread_name : "(Unnamed)");
      return nullptr;
    } else {
      Runtime::Current()->StartThreadBirth();
      self = new Thread(as_daemon);
      bool init_success = self->Init(runtime->GetThreadList(), runtime-
>GetJavaVM());
      Runtime::Current()->EndThreadBirth();
      if (!init_success) {
        delete self;
        return nullptr;
      }
    }
  }

  self->InitStringEntryPoints();

  CHECK_NE(self->GetState(), kRunnable);
  self->SetState(kNative);

  // Run the action that is acting on the peer.
  if (!peer_action(self)) {
    runtime->GetThreadList()->Unregister(self);
    // Unregister deletes self, no need to do this here.
    return nullptr;
  }
```

```
40
41    if (VLOG_IS_ON(threads)) {
42      if (thread_name != nullptr) {
43        VLOG(threads) << "Attaching thread " << thread_name;
44      } else {
45        VLOG(threads) << "Attaching unnamed thread.";
46      }
47      ScopedObjectAccess soa(self);
48      self->Dump(LOG_STREAM(INFO));
49    }
50
51    {
52      ScopedObjectAccess soa(self);
53      runtime->GetRuntimeCallbacks()->ThreadStart(self);
54    }
55
56    return self;
57  }
```

除了系统的JNI接口（"javacore", "nativehelper"), android framework 还有大量的Native实现，Android将所有这些接口一次性的通过start_reg()来完成

```
1  // \frameworks\base\core\jni\androidRuntime.cpp startReg() L1511
2   int AndroidRuntime::startReg(JNIEnv* env)
3  {
4      ATRACE_NAME("RegisterAndroidNatives");
5      /*
6       * This hook causes all future threads created in this process to be
7       * attached to the JavaVM.  (This needs to go away in favor of JNI
8       * Attach calls.)
9       */
10     androidSetCreateThreadFunc((android_create_thread_fn)
   javaCreateThreadEtc);
11
12     ALOGV("--- registering native functions ---\n");
13
14     /*
15      * Every "register" function calls one or more things that return
16      * a local reference (e.g. FindClass).  Because we haven't really
17      * started the VM yet, they're all getting stored in the base frame
18      * and never released.  Use Push/Pop to manage the storage.
19      */
20     env->PushLocalFrame(200);
21
22     if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
23         env->PopLocalFrame(NULL);
24         return -1;
25     }
26     env->PopLocalFrame(NULL);
27
28     //createJavaThread("fubar", quickTest, (void*) "hello");
29
30     return 0;
31 }
```

```cpp
// \system\core\libutils\Threads.cpp run() L662
status_t Thread::run(const char* name, int32_t priority, size_t stack)
{
    LOG_ALWAYS_FATAL_IF(name == nullptr, "thread name not provided to Thread::run");

    Mutex::Autolock _l(mLock);

    if (mRunning) {
        // thread already started
        return INVALID_OPERATION;
    }

    // reset status and exitPending to their default value, so we can
    // try again after an error happened (either below, or in readyToRun())
    mStatus = NO_ERROR;
    mExitPending = false;
    mThread = thread_id_t(-1);

    // hold a strong reference on ourself
    mHoldSelf = this;

    mRunning = true;

    bool res;
    // L685 Android native层有两种Thread的创建方式
    if (mCanCallJava) {
        res = createThreadEtc(_threadLoop,
                this, name, priority, stack, &mThread);
    } else {
        res = androidCreateRawThreadEtc(_threadLoop,
                this, name, priority, stack, &mThread);
    }

    if (res == false) {
        mStatus = UNKNOWN_ERROR;   // something happened!
        mRunning = false;
        mThread = thread_id_t(-1);
        mHoldSelf.clear();  // "this" may have gone away after this.

        return UNKNOWN_ERROR;
    }

    // Do not refer to mStatus here: The thread is already running (may, in fact
    // already have exited with a valid mStatus result). The NO_ERROR indication
    // here merely indicates successfully starting the thread and does not
    // imply successful termination/execution.
    return NO_ERROR;

    // Exiting scope of mLock is a memory barrier and allows new thread to run
}
```

它们的区别在是是否能够调用Java端函数，普通的thread就是对pthread_create的简单封装

```
//  \system\core\libutils\Threads.cpp run() L117
int androidCreateRawThreadEtc(android_thread_func_t entryFunction,
                              void *userData,
                              const char* threadName __android_unused,
                              int32_t threadPriority,
                              size_t threadStackSize,
                              android_thread_id_t *threadId)
{
    ...
    errno = 0;
    pthread_t thread;
    int result = pthread_create(&thread, &attr,
                    (android_pthread_entry)entryFunction, userData);
    ...
    return 1;
}
```

```
// \frameworks\base\core\jni\androidRuntime.cpp javaCreateThreadEtc() L1271
int AndroidRuntime::javaCreateThreadEtc(
                                android_thread_func_t entryFunction,
                                void* userData,
                                const char* threadName,
                                int32_t threadPriority,
                                size_t threadStackSize,
                                android_thread_id_t* threadId)
{
    void** args = (void**) malloc(3 * sizeof(void*));   // javaThreadShell must free
    int result;

    LOG_ALWAYS_FATAL_IF(threadName == nullptr, "threadName not provided to javaCreateThreadEtc");

    args[0] = (void*) entryFunction;
    args[1] = userData;
    args[2] = (void*) strdup(threadName);   // javaThreadShell must free

    result = androidCreateRawThreadEtc(AndroidRuntime::javaThreadShell, args,
        threadName, threadPriority, threadStackSize, threadId);
    return result;
}
```

```
// \frameworks\base\core\jni\androidRuntime.cpp javaThreadShell() L1242
int AndroidRuntime::javaThreadShell(void* args) {
    void* start = ((void**)args)[0];
    void* userData = ((void **)args)[1];
    char* name = (char*) ((void **)args)[2];        // we own this storage
    free(args);
    JNIEnv* env;
    int result;

    /* hook us into the VM */
    if (javaAttachThread(name, &env) != JNI_OK)
        return -1;

```

```
    /* start the thread running */
    result = (*(android_thread_func_t)start)(userData);

    /* unhook us */
    javaDetachThread();
    free(name);

    return result;
}
```

```
// \frameworks\base\core\jni\androidRuntime.cpp javaThreadShell() L1200
static int javaAttachThread(const char* threadName, JNIEnv** pEnv)
{
    JavaVMAttachArgs args;
    JavaVM* vm;
    jint result;

    vm = AndroidRuntime::getJavaVM();
    assert(vm != NULL);

    args.version = JNI_VERSION_1_4;
    args.name = (char*) threadName;
    args.group = NULL;

    result = vm->AttachCurrentThread(pEnv, (void*) &args);
    if (result != JNI_OK)
        ALOGI("NOTE: attach of thread '%s' failed\n", threadName);

    return result;
}
```

```
// \frameworks\base\core\java\com\android\internal\os\RuntimeInit.java
main() L325
public static final void main(String[] argv) {
        enableDdms();
        if (argv.length == 2 && argv[1].equals("application")) {
            if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application");

//将System.out 和 System.err 输出重定向到Android 的Log系统（定义在
android.util.Log)
            redirectLogStreams();
        } else {
            if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting tool");
        }

//commonInit(): 初始化了一下系统属性，其中最重要的一点就是设置了一个未捕捉异常的
handler，当代码有任何未知异常，就会执行它，调试过Android代码的同学经常看到的"*** FATAL
EXCEPTION IN SYSTEM PROCESS" 打印就出自这里
        commonInit();

        /*
         * Now that we're running in interpreted code, call back into native
code
         * to run the system.
         */
        nativeFinishInit();
```

```
21
22          if (DEBUG) Slog.d(TAG, "Leaving RuntimeInit!");
23      }
```

```
1  // \frameworks\base\core\jni\androidRuntime.cpp nativeFinishInit() L225
2  /*
3   * Code written in the Java Programming Language calls here from main().
4   */
5  static void com_android_internal_os_RuntimeInit_nativeFinishInit(JNIEnv* env,
   jobject clazz)
6  {
7      gCurRuntime->onStarted();
8  }
```

```
1  // \frameworks\base\cmds\app_process\app_main.cpp onStarted()  L78
2  virtual void onStarted()
3      {
4          sp<ProcessState> proc = ProcessState::self();
5          ALOGV("App process: starting thread pool.\n");
6          proc->startThreadPool();
7
8          AndroidRuntime* ar = AndroidRuntime::getRuntime();
9          ar->callMain(mClassName, mClass, mArgs);
10
11         IPCThreadState::self()->stopProcess();
12         hardware::IPCThreadState::self()->stopProcess();
13     }
```

ZygotInit

```
1  // \frameworks\base\core\java\com\android\internal\os\ZygotInit.java main()
   L750
2  public static void main(String argv[]) {
3          ZygoteServer zygoteServer = new ZygoteServer(); //新建Zygote服务器端
4
5           ...
6
7          final Runnable caller;
8          try {
9              ...
10             boolean startSystemServer = false;
11             String socketName = "zygote";Dalvik VM进程系统
12             String abiList = null;
13             boolean enableLazyPreload = false;
14             for (int i = 1; i < argv.length; i++) {
15                 //还记得app_main.cpp中传的start-system-server参数吗，在这里总有用
   到了
16                 if ("start-system-server".equals(argv[i])) {
17                     startSystemServer = true;
18                 } else if ("--enable-lazy-preload".equals(argv[i])) {
19                     enableLazyPreload = true;
20                 } else if (argv[i].startsWith(ABI_LIST_ARG)) {
21                     abiList = argv[i].substring(ABI_LIST_ARG.length());
22                 } else if (argv[i].startsWith(SOCKET_NAME_ARG)) {
23                     socketName =
   argv[i].substring(SOCKET_NAME_ARG.length());
```

```
                } else {
                    throw new RuntimeException("Unknown command line argument: " + argv[i]);
                }
            }

            if (abiList == null) {
                throw new RuntimeException("No ABI list supplied.");
            }

            zygoteServer.registerServerSocketFromEnv(socketName);//注册Socket
            // In some configurations, we avoid preloading resources and classes eagerly.
            // In such cases, we will preload things prior to our first fork.
             // 在有些情况下我们需要在第一个fork之前进行预加载资源
            if (!enableLazyPreload) {

                preload(bootTimingsTraceLog);

            } else {
                Zygote.resetNicePriority();
            }

            // Do an initial gc to clean up after startup
            bootTimingsTraceLog.traceBegin("PostZygoteInitGC");
            //主动进行一次资源GC
            gcAndFinalize();
            bootTimingsTraceLog.traceEnd(); // PostZygoteInitGC

            bootTimingsTraceLog.traceEnd(); // ZygoteInit
            // Disable tracing so that forked processes do not inherit stale tracing tags from
            // Zygote.
            Trace.setTracingEnabled(false, 0);

            Zygote.nativeSecurityInit();

            // Zygote process unmounts root storage spaces.
            Zygote.nativeUnmountStorageOnInit();

            ZygoteHooks.stopZygoteNoThreadCreation();

            if (startSystemServer) {
                Runnable r = forkSystemServer(abiList, socketName, zygoteServer);

                // {@code r == null} in the parent (zygote) process, and {@code r != null} in the
                // child (system_server) process.
                if (r != null) {
                    r.run();
                    return;
                }
            }

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

```

```
            // The select loop returns early in the child process after a fork and
            // loops forever in the zygote.
            caller = zygoteServer.runSelectLoop(abiList);
        } catch (Throwable ex) {
            Log.e(TAG, "System zygote died with exception", ex);
            throw ex;
        } finally {
            zygoteServer.closeServerSocket();
        }

        // We're in the child process and have exited the select loop. Proceed to execute the
        // command.
        if (caller != null) {
            caller.run();
        }
    }
```

preload() 的作用就是提前将需要的资源加载到VM中，比如class、resource等

```
 // \frameworks\base\core\java\com\android\internal\os\ZygotInit.java preload() L123
 static void preload(TimingsTraceLog bootTimingsTraceLog) {
        Log.d(TAG, "begin preload");
        bootTimingsTraceLog.traceBegin("BeginIcuCachePinning");
        beginIcuCachePinning();
        bootTimingsTraceLog.traceEnd(); // BeginIcuCachePinning
        bootTimingsTraceLog.traceBegin("PreloadClasses");
        //加载指定的类到内存并且初始化，使用的Class.forName(class, true, null);方式
        preloadClasses();
        bootTimingsTraceLog.traceEnd(); // PreloadClasses
        bootTimingsTraceLog.traceBegin("PreloadResources");
        //加载Android通用的资源，比如drawable、color...
        preloadResources();
        bootTimingsTraceLog.traceEnd(); // PreloadResources
        Trace.traceBegin(Trace.TRACE_TAG_DALVIK, "PreloadAppProcessHALs");
        nativePreloadAppProcessHALs();
        Trace.traceEnd(Trace.TRACE_TAG_DALVIK);
        Trace.traceBegin(Trace.TRACE_TAG_DALVIK, "PreloadOpenGL");
        //加载OpenGL...
        preloadOpenGL();
        Trace.traceEnd(Trace.TRACE_TAG_DALVIK);
        //加载共用的Library
        preloadSharedLibraries();
        //加载Text资源，字体等
        preloadTextResources();
        // Ask the WebViewFactory to do any initialization that must run in the zygote process,
        // for memory sharing purposes.
        // 为了内存共享，WebViewFactory进行任何初始化都要在Zygote进程中
        WebViewFactory.prepareWebViewInZygote();
        endIcuCachePinning();
        warmUpJcaProviders();
        Log.d(TAG, "end preload");

        sPreloadComplete = true;
```

```
    }
```

preloadClassess 将framework.jar里的preloaded-classes 定义的所有class load到内存里，preloaded-classes 编译Android后可以在framework/base下找到。而preloadResources 将系统的Resource(不是在用户apk里定义的resource）load到内存。资源preload到Zygoted的进程地址空间，所有fork的子进程将共享这份空间而无需重新load, 这大大减少了应用程序的启动时间，但反过来增加了系统的启动时间。通过对preload 类和资源数目进行调整可以加快系统启动。Preload也是Android启动最耗时的部分之一

```
// \frameworks\base\core\java\com\android\internal\os\ZygotInit.java gcAndFinalize() L439
static void gcAndFinalize() {
        final VMRuntime runtime = VMRuntime.getRuntime();

        /* runFinalizationSync() lets finalizers be called in Zygote,
         * which doesn't have a HeapWorker thread.
         */
        System.gc();
        runtime.runFinalizationSync();
        System.gc();
    }
```

gc()调用只是通知VM进行垃圾回收，是否回收，什么时候回收全由VM内部算法决定。GC的回收有一个复杂的状态机控制，通过多次调用，可以使得尽可能多的资源得到回收。gc()必须在fork之前完成（接下来的StartSystemServer就会有fork操作），这样将来被复制出来的子进程才能有尽可能少的垃圾内存没有释放

```
 // \frameworks\base\core\java\com\android\internal\os\ZygotInit.java gcAndFinalize() L657
 private static Runnable forkSystemServer(String abiList, String socketName,
            ZygoteServer zygoteServer) {
        long capabilities = posixCapabilitiesAsBits(
            OsConstants.CAP_IPC_LOCK,
            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_PTRACE,
            OsConstants.CAP_SYS_TIME,
            OsConstants.CAP_SYS_TTY_CONFIG,
            OsConstants.CAP_WAKE_ALARM,
            OsConstants.CAP_BLOCK_SUSPEND
        );
        /* Containers run without some capabilities, so drop any caps that are not available. */
        StructCapUserHeader header = new StructCapUserHeader(
                OsConstants._LINUX_CAPABILITY_VERSION_3, 0);
        StructCapUserData[] data;
        try {
            data = Os.capget(header);
        } catch (ErrnoException ex) {
            throw new RuntimeException("Failed to capget()", ex);
```

```
        }
        capabilities &= ((long) data[0].effective) | (((long)
data[1].effective) << 32);

        /* Hardcoded command line to start the system server */

        //启动SystemServer的命令行，部分参数写死
        String args[] = {
            "--setuid=1000",
            "--setgid=1000",
            "--
setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,1021,1023,1
024,1032,1065,3001,3002,3003,3006,3007,3009,3010",
            "--capabilities=" + capabilities + "," + capabilities,
            "--nice-name=system_server",
            "--runtime-args",
            "--target-sdk-version=" + VMRuntime.SDK_VERSION_CUR_DEVELOPMENT,
            "com.android.server.SystemServer",
        };
        ZygoteConnection.Arguments parsedArgs = null;

        int pid;

        try {
            parsedArgs = new ZygoteConnection.Arguments(args);
            ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);
            ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);//会设
置InvokeWith参数，这个参数在接下来的初始化逻辑中会有调用

            boolean profileSystemServer = SystemProperties.getBoolean(
                    "dalvik.vm.profilesystemserver", false);
            if (profileSystemServer) {
                parsedArgs.runtimeFlags |= Zygote.PROFILE_SYSTEM_SERVER;
            }

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

        /* For child process */
        if (pid == 0) {//如果是第一次创建的话pid==0
            if (hasSecondZygote(abiList)) {
                waitForSecondaryZygote(socketName);
            }

            zygoteServer.closeServerSocket();
            return handleSystemServerProcess(parsedArgs);
        }

```

```
        return null;
    }
```

ZygoteInit.forkSystemServer() 方法fork 出一个新的进程，这个进程就是SystemServer进程。fork出来的子进程在handleSystemServerProcess 里开始初始化工作，主要工作分为：

1. prepareSystemServerProfile（）方法中将SYSTEMSERVERCLASSPATH中的AppInfo加载到VM中。
2. 判断fork args中是否有invokWith参数，如果有则进行WrapperInit.execApplication（不进行深入讲解了）。如果没有则调用

```
// \frameworks\base\core\java\com\android\internal\os\ZygoteInit.java
handleSystemServerProcess() L453
private static Runnable handleSystemServerProcess(ZygoteConnection.Arguments
parsedArgs) {
        ...
            if (profileSystemServer && (Build.IS_USERDEBUG || Build.IS_ENG))
{
                try {
                    //将SYSTEMSERVERCLASSPATH中的AppInfo加载到VM中
                    prepareSystemServerProfile(systemServerClasspath);
                } catch (Exception e) {
                    Log.wtf(TAG, "Failed to set up system server profile",
e);
                }
            }
        }

        if (parsedArgs.invokeWith != null) {
            ...
            //判断fork args中是否有invokWith参数，如果有则进行
WrapperInit.execApplication
            WrapperInit.execApplication(parsedArgs.invokeWith,
                    parsedArgs.niceName, parsedArgs.targetSdkVersion,
                    VMRuntime.getCurrentInstructionSet(), null, args);

            throw new IllegalStateException("Unexpected return from
WrapperInit.execApplication");
        } else {

            ...
            /*
             * Pass the remaining arguments to SystemServer.
             */
             //调用zygoteInit
            return ZygoteInit.zygoteInit(parsedArgs.targetSdkVersion,
parsedArgs.remainingArgs, cl);
        }

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

```
// \frameworks\base\core\java\com\android\internal\os\RuntimeInit.java applicationInit() L345
    protected static Runnable applicationInit(int targetSdkVersion, String[] argv,
            ClassLoader classLoader) {
        ...

        // Remaining arguments are passed to the start class's static main
        //findStaticMain来运行args的startClass的main方法
        return findStaticMain(args.startClass, args.startArgs, classLoader);
    }
```

```
// \frameworks\base\core\java\com\android\internal\os\RuntimeInit.java findStaticMain() L287
 protected static Runnable findStaticMain(String className, String[] argv,
            ClassLoader classLoader) {
        Class<?> cl;

        try {
            cl = Class.forName(className, true, classLoader);
        } 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);
        }

        return new MethodAndArgsCaller(m, argv);
    }
```

很明显这是一个耗时操作所以使用线程来完成：

```
// \frameworks\base\core\java\com\android\internal\os\RuntimeInit.java MethodAndArgsCaller L479
 static class MethodAndArgsCaller 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);
            }
        }
    }
```

System Server 启动流程

System Server 是Zygote fork 的第一个Java 进程， 这个进程非常重要，因为他们有很多的系统线程，提供所有核心的系统服务

看到大名鼎鼎的WindowManager, ActivityManager了吗？对了，它们都是运行在system_server的进程里。还有很多“Binder-x”的线程，它们是各个Service为了响应应用程序远程调用请求而创建的。除此之外，还有很多内部的线程，比如 ”UI thread”, “InputReader”, “InputDispatch” 等等，我，现在我们只关心System Server是如何创建起来的。
SystemServer的main() 函数。

```
public static void main(String[] args) {
    new SystemServer().run();
}
```

记下来我分成4部分详细分析SystemServer run方法的初始化流程：
初始化必要的SystemServer环境参数，比如系统时间、默认时区、语言、load一些Library等等，
初始化Looper，我们在主线程中使用到的looper就是在SystemServer中进行初始化的
初始化Context，只有初始化一个Context才能进行启动Service等操作，这里看一下源码：

```
private void createSystemContext() {
     ActivityThread activityThread = ActivityThread.systemMain();
     mSystemContext = activityThread.getSystemContext();
     mSystemContext.setTheme(DEFAULT_SYSTEM_THEME);
     final Context systemUiContext = activityThread.getSystemUiContext();
     systemUiContext.setTheme(DEFAULT_SYSTEM_THEME);
}
```

看到没有ActivityThread就是这个时候生成的

继续看ActivityThread中如何生成Context：

```
public ContextImpl getSystemContext() {
    synchronized (this) {
      if (mSystemContext == null) {
         mSystemContext = ContextImpl.createSystemContext(this);
      }
      return mSystemContext;
    }
}
```

ContextImpl是Context类的具体实现，里面封装完成了生成几种常用的createContext的方法：

```
static ContextImpl createSystemContext(ActivityThread mainThread) {
      LoadedApk packageInfo = new LoadedApk(mainThread);
      //省略代码
      return context;
}

static ContextImpl createSystemUiContext(ContextImpl systemContext) {
     final LoadedApk packageInfo = systemContext.mPackageInfo;
     //省略代码
     return context;
}

static ContextImpl createAppContext(ActivityThread mainThread, LoadedApk
packageInfo) {
     if (packageInfo == null) throw new
IllegalArgumentException("packageInfo");
     //省略代码
     return context;
}

static ContextImpl createActivityContext(ActivityThread mainThread,
     LoadedApk packageInfo, ActivityInfo activityInfo, IBinder
activityToken, int displayId,
          Configuration overrideConfiguration) {
          //省略代码
          return context;
}
```

初始化SystemServiceManager,用来管理启动service，SystemServiceManager中封装了启动Service的startService方法启动系统必要的Service，启动service的流程又分成三步走：

```
// Start services.
try {
     traceBeginAndSlog("StartServices");
     startBootstrapServices();
     startCoreServices();
     startOtherServices();
     SystemServerInitThreadPool.shutdown();
} catch (Throwable ex) {
       //
} finally {
     traceEnd();
}
```

启动BootstrapServices,就是系统必须需要的服务，这些服务直接耦合性很高，所以干脆就放在一个方法里面一起启动，比如PowerManagerService、RecoverySystemService、DisplayManagerService、ActivityManagerService等等启动以基本的核心Service，很简单，只有三个BatteryService、UsageStatsService、WebViewUpdateService启动其它需要用到的Service，比如NetworkScoreService、AlarmManagerService

5. 善后工作是不是到此之后，Zygote的工作变得很轻松了，可以宜养天年了？可惜现代社会，哪个父母把孩子养大就可以撒手不管了？尤其是像Sytem Server 这样肩负社会重任的大儿子，出问题了父母还是要帮一把的。这里，Zygote会默默的在后台凝视这自己的大儿子，一旦发现System Server 挂掉了，将其回收，然后将自己杀掉，重新开始新的一生， 可怜天下父母心啊。这段实现在代码： dalvik/vm/native/dalvik_system_zygote.cpp 中，

```
static void Dalvik_dalvik_system_Zygote_forkSystemServer(
        const u4* args, JValue* pResult){
    ...
    pid_t pid;
    pid = forkAndSpecializeCommon(args, true);
    ...
    if (pid > 0) {
        int status;
        gDvm.systemServerPid = pid;
        /* WNOHANG 会让waitpid 立即返回，这里只是为了预防上面的赋值语句没有完成之前SystemServer就crash 了*/
        if (waitpid(pid, &status, WNOHANG) == pid) {
            ALOGE("System server process %d has died. Restarting Zygote!", pid);
            kill(getpid(), SIGKILL);
        }
    }
    RETURN_INT(pid);
}

/* 真正的处理在这里 */
static void sigchldHandler(int s){
    ...
    pid_t pid;
    int status;
    ...
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        ...
        if (pid == gDvm.systemServerPid) {
            ...
            kill(getpid(), SIGKILL);
        }
    }
    ...
}

static void Dalvik_dalvik_system_Zygote_fork(const u4* args, JValue* pResult){
    pid_t pid;
    ...
    setSignalHandler(); //signalHandler 在这里注册
    ...
    pid = fork();
    ...
    RETURN_INT(pid);
```

```
43 }
```

在Unix-like系统，父进程必须用 waitpid 等待子进程的退出，否则子进程将变成"Zombie" (僵尸）进程，不仅系统资源泄漏，而且系统将崩溃（没有system server，所有Android应用程序都无法运行）。但是waitpid() 是一个阻塞函数（WNOHANG参数除外），所以通常做法是在signal 处理函数里进行无阻塞的处理，因为每个子进程退出的时候，系统会发出 SIGCHID 信号。Zygote会把自己杀掉， 那父亲死了，所有的应用程序不就成为孤儿了？ 不会，因为父进程被杀掉后系统会自动给所有的子进程发生SIGHUP信号，该信号的默认处理就是将杀掉自己退出当前进程。但是一些后台进程（Daemon)可以通过设置SIG_IGN参数来忽略这个信号，从而得以在后台继续运行。

总结

1. init 根据init.rc 运行 app_process, 并携带'–zygote' 和 '–startSystemServer' 参数。
2. AndroidRuntime.cpp::start() 里将启动JavaVM，并且注册所有framework相关的系统JNI接口。
3. 第一次进入Java世界，运行ZygoteInit.java::main() 函数初始化Zygote. Zygote 并创建Socket的server 端。
4. 然后fork一个新的进程并在新进程里初始化SystemServer. Fork之前，Zygote是preload常用的Java类库，以及系统的resources，同时GC（）清理内存空间，为子进程省去重复的工作。
5. SystemServer 里将所有的系统Service初始化，包括ActivityManager 和 WindowManager, 他们是应用程序运行起来的前提。
6. 依次同时，Zygote监听服务端Socket，等待新的应用启动请求。
7. ActivityManager ready 之后寻找系统的"Startup" Application, 将请求发给Zygote。
8. Zygote收到请求后，fork出一个新的进程。
9. Zygote监听并处理SystemServer 的 SIGCHID 信号，一旦System Server崩溃，立即将自己杀死。init会重启Zygote.

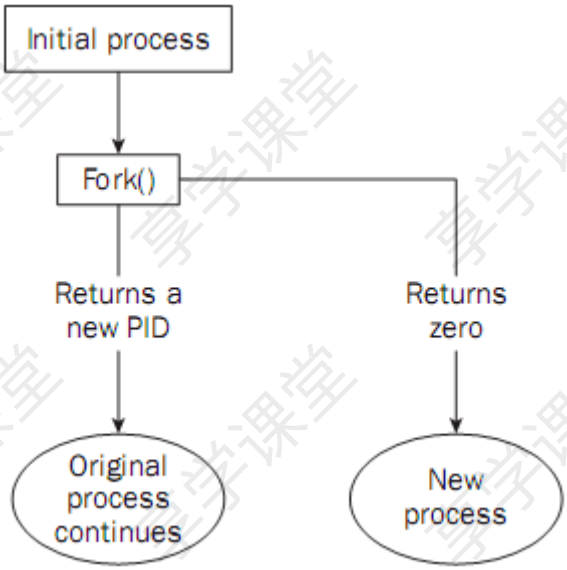
什么情况下Zygote进程会重启呢?

- servicemanager进程被杀;
- (onresart)surfaceflinger进程被杀;
- (onresart)Zygote进程自己被杀;
- (oneshot=false)system_server进程被杀; (waitpid)

fork函数

```
1 pid_t fork(void)
```

1. 参数：不需要参数
2. 需要的头文件 `<sys/types.h>` 和 `<unistd.h>`
3. 返回值分两种情况：
   - 返回0表示成功创建子进程，并且接下来进入子进程执行流程
   - 返回PID（>0），成功创建子进程，并且继续执行父进程流程代码
   - 返回非正数（<0），创建子进程失败，失败原因主要有：
     进程数超过系统所能创建的上限，errno会被设置为EAGAIN系统内存不足，errno会被设置为ENOMEM



> 使用 `fork()` 函数得到的子进程是父进程的一个复制品，它从父进程处继承了整个进程的地址空间：包括进程上下文（进程执行活动全过程的静态描述）、进程堆栈、打开的文件描述符、信号控制设定、进程优先级、进程组号等。子进程所独有的只有它的进程号，计时器等（只有小量信息）。因此，使用 `fork()` 函数的代价是很大的

子进程与父进程的区别

1. 除了文件锁以外,其他的锁都会被继承
2. 各自的进程ID和父进程ID不同
3. 子进程的未决告警被清除；
4. 子进程的未决信号集设置为空集。

写时拷贝 (copy- on-write)

Linux 的 `fork()` 使用是通过写时拷贝 (copy- on-write) 实现。写时拷贝是一种可以推迟甚至避免拷贝数据的技术。内核此时并不复制整个进程的地址空间，而是让父子进程共享同一个地址空间。只用在需要写入的时候才会复制地址空间，从而使各个进行拥有各自的地址空间。也就是说，资源的复制是在需要写入的时候才会进行，在此之前，只有以只读方式共享

孤儿进程、僵尸进程

fork系统调用之后，父子进程将交替执行，执行顺序不定。如果父进程先退出，子进程还没退出那么子进程的父进程将变为init进程（托孤给了init进程）。（注：任何一个进程都必须有父进程）如果子进程先退出，父进程还没退出，那么子进程必须等到父进程捕获到了子进程的退出状态才真正结束，否则这个时候子进程就成为僵进程（**僵尸进程**：只保留一些退出信息供父进程查询）

多线程进程的Fork调用

在 POSIX 标准中，fork 的行为是这样的：复制整个用户空间的数据（通常使用 copy-on-write 的策略，所以可以实现的速度很快）以及所有系统对象，然后仅复制当前线程到子进程。这里：所有父进程中别的线程，到了子进程中都是突然蒸发掉的

> 假设这么一个环境，在 fork 之前，有一个子线程 lock 了某个锁，获得了对锁的所有权。fork 以后，在子进程中，所有的额外线程都人间蒸发了。而锁却被正常复制了，在子进程看来，这个锁没有主人，所以没有任何人可以对它解锁。当子进程想 lock 这个锁时，不再有任何手段可以解开了。程序发生死锁

面试题

面试官：你了解 Android 系统启动流程吗?
A：当按电源键触发开机，首先会从 ROM 中预定义的地方加载引导程序 BootLoader 到 RAM 中，并执行 BootLoader 程序启动 Linux Kernel， 然后启动用户级别的第一个进程： init 进程。init 进程会解析 init.rc 脚本做一些初始化工作，包括挂载文件系统、创建工作目录以及启动系统服务进程等，其中系统服务进程包括 Zygote、service manager、media 等。在 Zygote 中会进一步去启动 system_server 进程，然后在 system_server 进程中会启动 AMS、WMS、PMS 等服务，等这些服务启动之后，AMS 中就会打开 Launcher 应用的 home Activity，最终就看到了手机的 "桌面"。

面试官：system_server 为什么要在 Zygote 中启动，而不是由 init 直接启动呢?
A：Zygote 作为一个孵化器，可以提前加载一些资源，这样 fork() 时基于 Copy-On-Write 机制创建的其他进程就能直接使用这些资源，而不用重新加载。比如 system_server 就可以直接使用 Zygote 中的 JNI 函数、共享库、常用的类、以及主题资源。

面试官：为什么要专门使用 Zygote 进程去孵化应用进程，而不是让 system_server 去孵化呢?
A：首先 system_server 相比 Zygote 多运行了 AMS、WMS 等服务，这些对一个应用程序来说是不需要的。另外进程的 fork() 对多线程不友好，仅会将发起调用的线程拷贝到子进程，这可能会导致死锁，而 system_server 中肯定是有很多线程的。

面试官：能说说具体是怎么导致死锁的吗?
在 POSIX 标准中，fork 的行为是这样的：复制整个用户空间的数据（通常使用 copy-on-write 的策略，所以可以实现的速度很快）以及所有系统对象，然后仅复制当前线程到子进程。这里：所有父进程中别的线程，到了子进程中都是突然蒸发掉的

对于锁来说，从 OS 看，每个锁有一个所有者，即最后一次 lock 它的线程。假设这么一个环境，在 fork 之前，有一个子线程 lock 了某个锁，获得了对锁的所有权。fork 以后，在子进程中，所有的额外线程都人间蒸发了。而锁却被正常复制了，在子进程看来，这个锁没有主人，所以没有任何人可以对它解锁。当子进程想 lock 这个锁时，不再有任何手段可以解开了。程序发生死锁

面试官：Zygote 为什么不采用 Binder 机制进行 IPC 通信?
A：Binder 机制中存在 Binder 线程池，是多线程的，如果 Zygote 采用 Binder 的话就存在上面说的 fork() 与 多线程的问题了。其实严格来说，Binder 机制不一定要多线程，所谓的 Binder 线程只不过是在循环读取 Binder 驱动的消息而已，只注册一个 Binder 线程也是可以工作的，比如 service manager 就是这样的。实际上 Zygote 尽管没有采取 Binder 机制，它也不是单线程的，但它在 fork() 前主动停止了其他线程，fork() 后重新启动了。