Android 架构

作者：刘朝

日期：2020 年 8 月 2 日

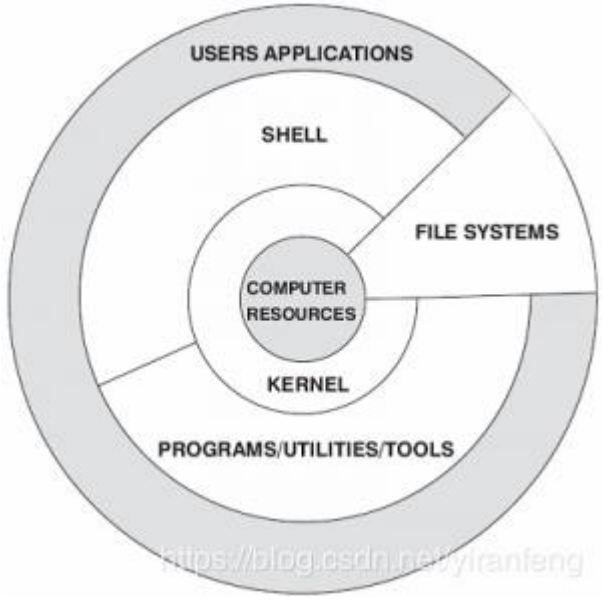
目录

一、架构

Android 的系统架构，Android 使用 linux 内核（Marco kernel），但是 Android 的架构又与 Linux 系统有所不同，因此在介绍 Android 系统架构之前，我们先一起来了解一下 Linux 系统的架构。

1.1 Linux 系统架构

Linux 架构如下图所示：



- Computer Resources：硬件资源
- Kernel：内核
- Shell：shell 是系统的用户界面，提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行，是一个命令解释器
- Programs/Utilities/Tools：库函数、工具等
- File systems：文件系统是文件存放在磁盘等存储设备上的组织方法。Linux 系统能支持多种目前流行的文件系统，如 EXT2、 EXT3、 FAT、 FAT32、 VFAT 和 ISO9660。
- User Application：Linux 应用，标准的 Linux 系统一般都有一套被称为应用程序的程序集，它包括文本编辑器、编程语言、X Window、办公套件、Internet 工具和数据库等

1.2 Android 系统架构

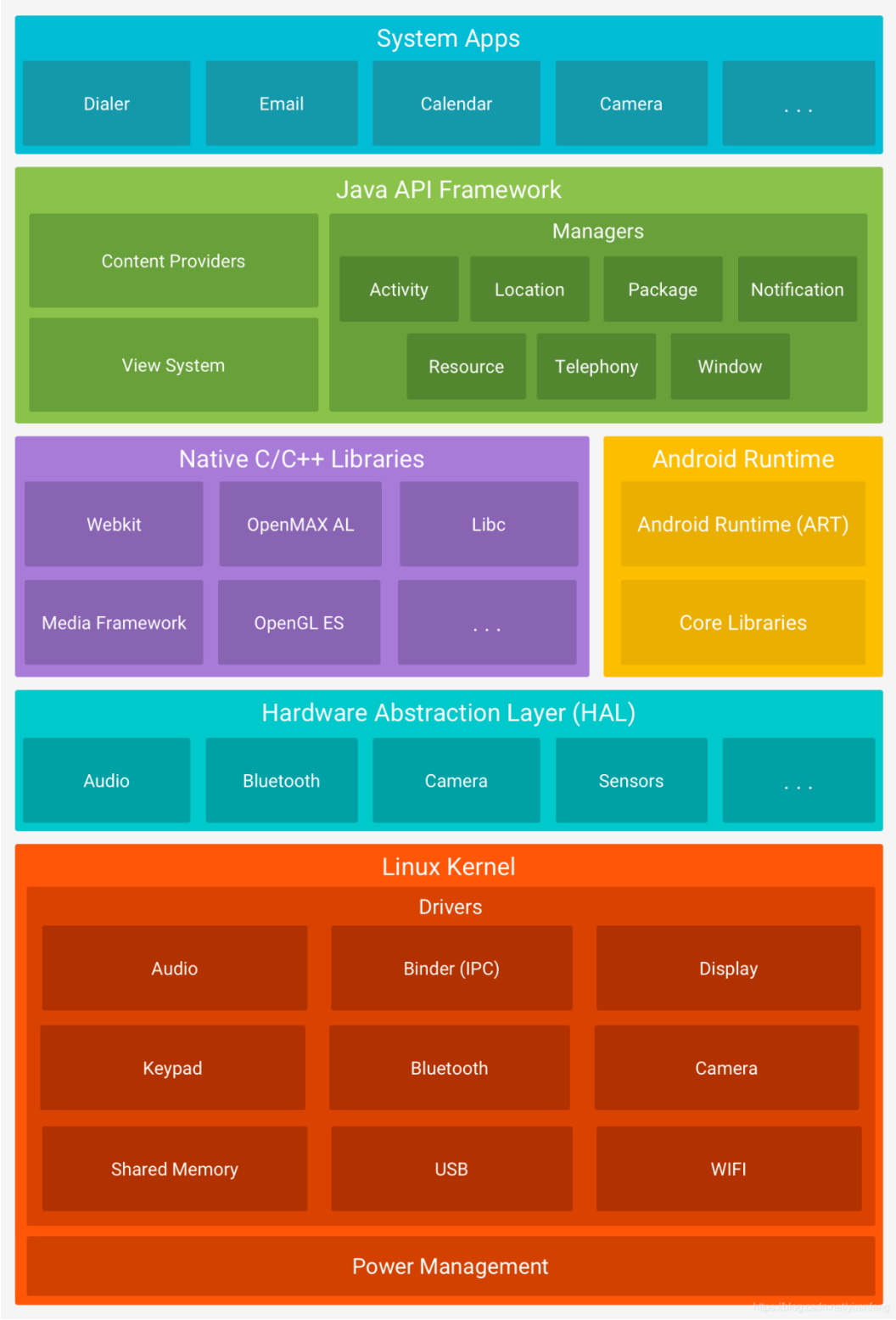
Linux 系统一般由 4 个组成部分：**内核 Kernel、shell、文件系统和应用程序。**内核、shell 和文件系统一起组成了基本的操作系统结构，它们让用户可以管理文件、运行程序并使用系统。

Linux 开机后，内核启动，激活内核空间，抽象硬件、初始化硬件参数等，运行并维护虚拟内存、调度器、信号及进程间通信(IPC)。

内核启动后，再加载 Shell 和用户应用程序，用户应用程序使用 C\C++编写，被编译成机器码，形成一个进程，通过系统调用（Syscall）与内核系统进行联通。进程间交流需要使用特殊的进程间通信(IPC)机制。

看完了 Linux 架构，我们再来一起看看 Android 系统架构。Android 的系统非常复杂和庞大，底层以 Linux 内核为基础，上层采用带有虚拟机的 JAVA 层，通过 JNI 技术，将上下层打通。

先来看一张Google提供经典Android架构图,从上往下依次为应用层(System Apps)、应用框架层（Java API Framework）、运行层（系统 Native 库和 Android 运行时环境）、硬件抽象层(HAL)、Liunx 内核(Marco Kernel)。每一层都有对应的进程、系统库。



- 应用层（System Apps）

该层中包含所有的 Android 应用程序，包括电话、相机、日历等，我们自己开发的 Android 应用程序也被安装在这层；大部分的应用使用 JAVA 开发，现在 Google 也开始力推 kotlin 进行开发

- 应用框架层（Java API Framework）

这一层主要提供构建应用程序是可能用到的各种 API，Android 自带的一些核心应用就是使用这些 API 完成的，开发者也可以通过使用 API 来构建自己的应用程序。

- 运行层

1）系统 Native 库

Android 包含一些 C/C++库，这些库能被 Android 系统中不同的组件使用

2）Android 运行时环境

Android 包括了一个核心库，该核心库提供了 Java 编程语言核心库的大多数功能。虚拟机也在该层启动。每个 Android 应用都有一个专有的进程，这些进程每个都有一个 Dalivik 虚拟机实例，并在该实例中运行。

- 硬件抽象层(HAL)

Android 的硬件驱动与 Linux 不同，传统的 Liunx 内核驱动完全存在于内核空间中。但是 Android 在内核外部增加了一个硬件抽象层(HAL-Hardware Abstraction Layer)，把一部分硬件驱动放到了 HAL 层。

为什么 Android 要这么做呢?

Linux 内核采用了 GPL 协议，如果硬件厂商需要支持 Linux 系统，就需要遵照 GPL 协议公开硬件驱动的源代码，这势必会影响到硬件厂家的核心利益。

Android 的 HAL 层运行在用户空间，HAL 是一个“空壳”，Android 会根据不同的需要，加载不同的动态库。这些动态库由硬件厂家提供。硬件厂家把相关硬件功能写入动态库，内核中只开放一些基本的读写接口操作。这样一些硬件厂家的驱动功能就由内核空间移动到了用户空间。

Android 的 HAL 层遵循 Apache 协议，并不要求它的配套程序，因此厂家提供的驱动库不需要进行开放，保护了硬件厂家的核心利益。

- **Liunx 内核(Marco Kernel)**

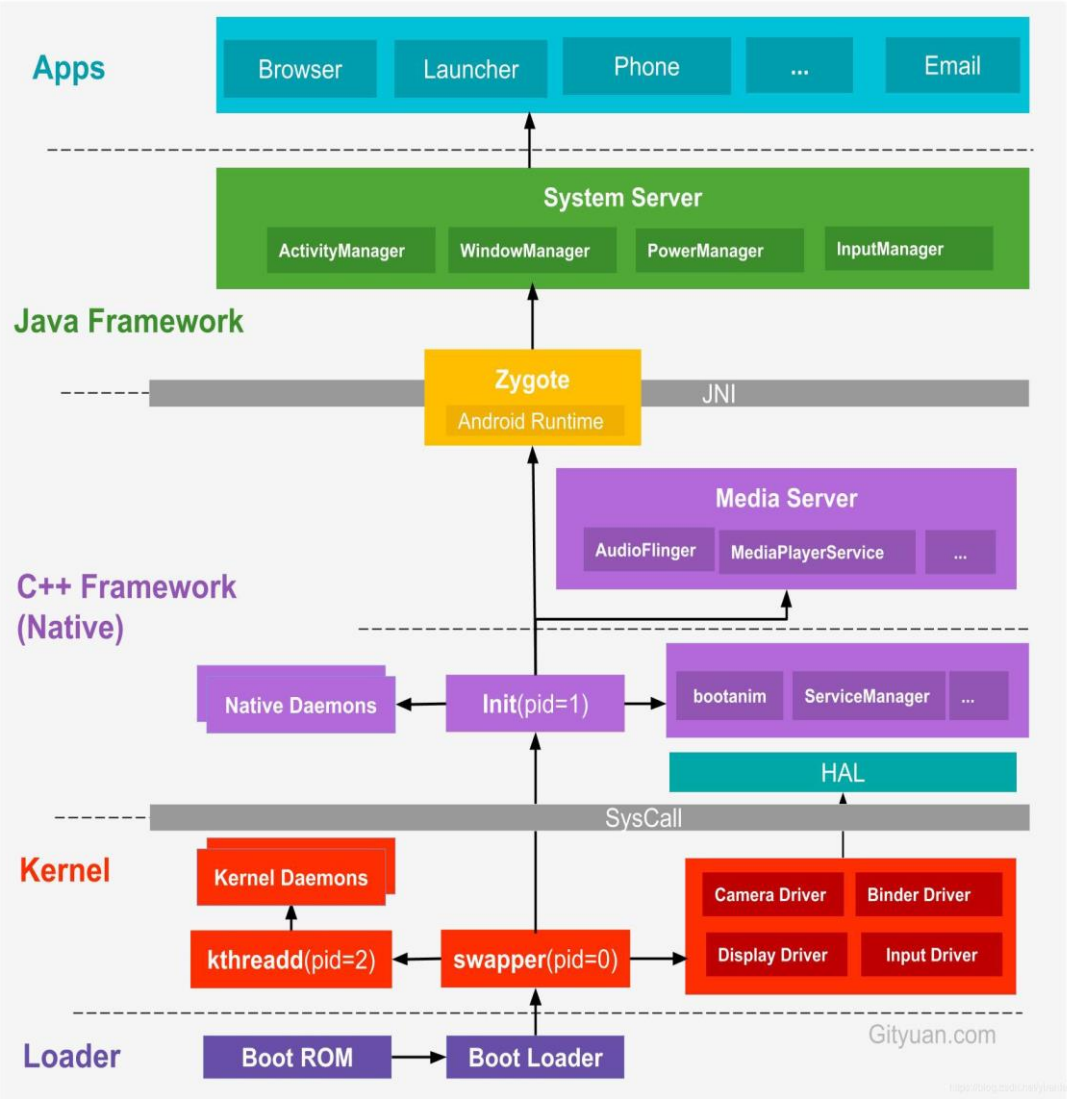
Android 平台的基础是 Linux 内核，比如 ART 虚拟机最终调用底层 Linux 内核来执行功能。Linux 内核的安全机制为 Android 提供相应的保障，也允许设备制造商为内核开发硬件驱动程序。

二、Android 启动

2.1 概述

- BootRom->BootLoader->Linux Kernel->Init->Zygote->SystemServer->Launcher
- BootLoader 层：主要包括 Boot Rom 和 Boot Loader
- Kernel 层：主要是 Android 内核层
- Native 层：主要是包括 init 进程以及其 fork 出 来的用户空间的守护进程、HAL 层、开机动画等
- JAVA Framework 层：主要是 AMS 和 PMS 等 Service 的初始化
- Application 层：主要指 SystemUI、Launcher 的启动

Android 架构图如下，图片来源于 Gityuan



2.2 Android 系统启动流程

第一步：手机开机后，引导芯片启动，引导芯片开始从固化在 ROM 里的预设代码执行，加载引导程序到到 RAM，bootloader 检查 RAM，初始化硬件参数等功能；

第二步：硬件等参数初始化完成后，进入到 Kernel 层，Kernel 层主要加载一些硬件设备驱动，初始化进程管理等操作。在 Kernel 中首先启动 swapper 进程（pid=0），用于初始化进程管理、内管管理、加载 Driver 等操作，再启动 kthread 进程(pid=2),这些 linux 系统的内核进程，kthread 是所有内核进程的鼻祖；

第三步：Kernel 层加载完毕后，硬件设备驱动与 HAL 层进行交互。初始化进程管理等操作会启动 INIT 进程 ，这些在 Native 层中；

第四步：init 进程(pid=1，init 进程是所有进程的鼻祖，第一个启动)启动后，会启动 adbd，logd 等用户守护进程，并且会启动 servicemanager(binder 服务管家)等重要服务，同时孵化出 zygote 进程，这里属于 C++ Framework，代码为 C++程序；

第五步：zygote 进程是由 init 进程解析 init.rc 文件后 fork 生成，它会加载虚拟机，启动 System Server(zygote 孵化的第一个进程)；System Server 负责启动和管理整个 Java Framework，包含 ActivityManager，WindowManager，PackageManager，PowerManager 等服务；

第六步：zygote 同时会启动相关的 APP 进程，它启动的第一个 APP 进程为 Launcher，然后启动 Email，SMS 等进程，所有的 APP 进程都有 zygote fork 生成

2.3 Android 系统启动之 init 进程

init 进程是 linux 系统中用户空间的第一个进程，进程号为 1.当 bootloader 启动后，

启动 kernel，kernel 启动完后，在用户空间启动 init 进程，再通过 init 进程，来读取 init.rc 中的相关配置，从而来启动其他相关进程以及其他操作。

2.3.1 概述

init 进程是 linux 系统中用户空间的第一个进程，进程号为 1.

当 bootloader 启动后，启动 kernel，kernel 启动完后，在用户空间启动 init 进程，再通过 init 进程，来读取 init.rc 中的相关配置，从而来启动其他相关进程以及其他操作。

init 进程被赋予了很多重要工作，init 进程启动主要分为两个阶段：

第一个阶段完成以下内容：

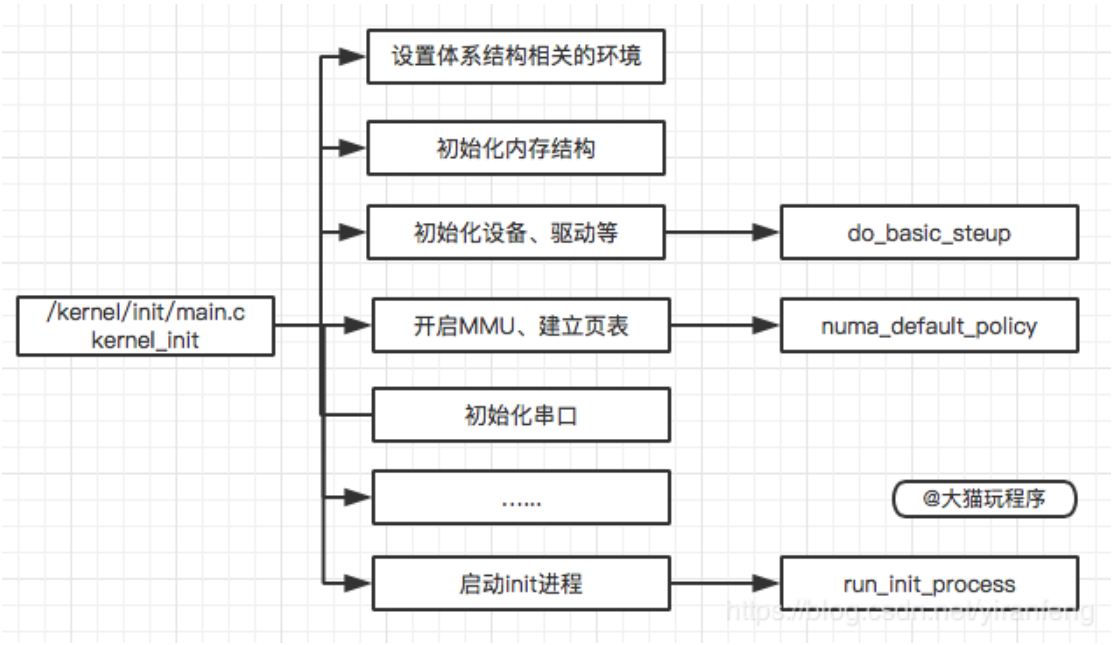
- ueventd/watchdogd 跳转及环境变量设置
- 挂载文件系统并创建目录
- 初始化日志输出、挂载分区设备
- 启用 SELinux 安全策略
- 开始第二阶段前的准备

第二个阶段完成以下内容：

- 初始化属性系统
- 执行 SELinux 第二阶段并恢复一些文件安全上下文
- 新建 epoll 并初始化子进程终止信号处理函数
- 设置其他系统属性并开启属性服务

2.3.2 架构

2.3.2.1 Init 进程如何被启动?

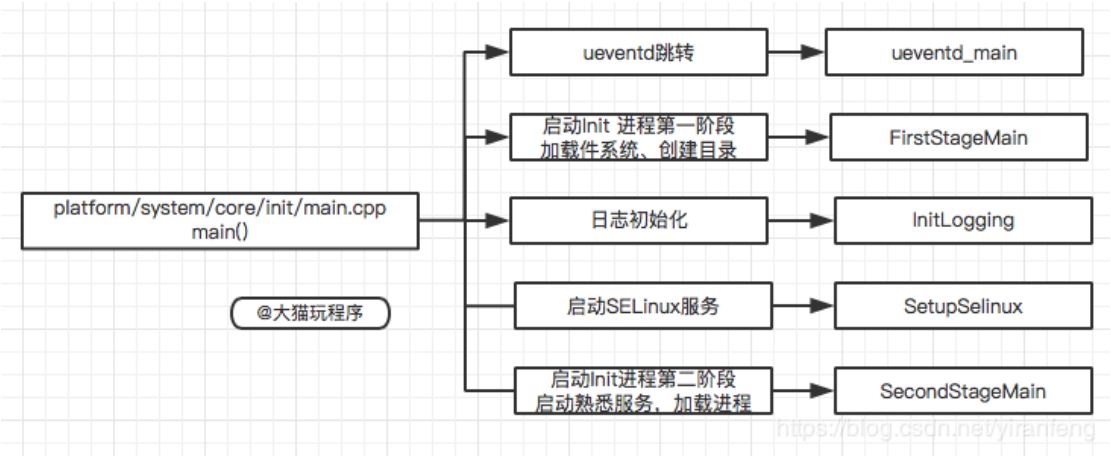


Init 进程是在 Kernel 启动后，启动的第一个用户空间进程，PID 为 1。

kernel_init 启动后，完成一些 init 的初始化操作，然后去系统根目录下依次找 ramdisk_execute_command 和 execute_command 设置的应用程序，如果这两个目录都找不到，就依次去根目录下找 /sbin/init，/etc/init，/bin/init,/bin/sh 这四个应用程序进行启动，只要这些应用程序有一个启动了，其他就不启动了。

Android 系统一般会在根目录下放一个 init 的可执行文件，也就是说 Linux 系统的 init 进程在内核初始化完成后，就直接执行 init 这个文件。

2.3.2.2 Init 进程启动后，做了哪些事?



Init 进程启动后，首先挂载文件系统、再挂载相应的分区，启动 SELinux 安全策略，启动属性服务，解析 rc 文件，并启动相应属性服务进程，初始化 epoll，依次设置 signal、property、keychord 这 3 个 fd 可读时相对应的回调函数。进入无线循环，用来响应各个进程的变化与重建。

2.3.3 kernel 启动 init 进程 源码分析

2.3.3.1 kernel_init

kernel/msm-4.19/init/main.c

```
kernel/msm-4.19/init/main.c
kernel_init()
  |
run_init_process(ramdisk_execute_command)  //运行可执行文件，启动 init 进程
```

```
static int __ref kernel_init(void *unused)
{
    kernel_init_freeable(); //进行 init 进程的一些初始化操作
    /* need to finish all async __init code before freeing the memory */
    async_synchronize_full();// 等待所有异步调用执行完成,，在释放内存前，必须完成所有的异步 __init 代码
```

```
    free_initmem();// 释放所有 init.* 段中的内存
    mark_rodata_ro(); //arm64 空实现
    system_state = SYSTEM_RUNNING;// 设置系统状态为运行状态
    numa_default_policy(); // 设定 NUMA 系统的默认内存访问策略

    flush_delayed_fput(); // 释放所有延时的 struct file 结构体

    if (ramdisk_execute_command) { //ramdisk_execute_command 的值为"/init"
        if (!run_init_process(ramdisk_execute_command)) //运行根目录下的 init 程序
            return 0;
        pr_err("Failed to execute %s\n", ramdisk_execute_command);
    }

    /*
     * We try each of these until one succeeds.
     *
     * The Bourne shell can be used instead of init if we are
     * trying to recover a really broken machine.
     */
    if (execute_command) { //execute_command 的值如果有定义就去根目录下找对应的应用程序,然后启动
        if (!run_init_process(execute_command))
            return 0;
        pr_err("Failed to execute %s.  Attempting defaults...\n",
            execute_command);
    }
    if (!run_init_process("/sbin/init") || //如果 ramdisk_execute_command 和 execute_command 定义的应用程序都没有找到,
    //就到根目录下找 /sbin/init，/etc/init，/bin/init,/bin/sh 这四个应用程序进行启动

        !run_init_process("/etc/init") ||
        !run_init_process("/bin/init") ||
        !run_init_process("/bin/sh"))
        return 0;

    panic("No init found.  Try passing init= option to kernel. "
          "See Linux Documentation/init.txt for guidance.");
}
```

2.3.3.2 do_basic_setup

```
kernel_init_freeable（）
```

```
2. |
3. do_basic_setup（）
```

```
1. static void __init do_basic_setup(void)
2. {
3.     cpuset_init_smp();//针对 SMP 系统，初始化内核 control group 的 cpuset 子系统。
4.     usermodehelper_init();// 创建 khelper 单线程工作队列，用于协助新建和运行用户空间程序
5.     shmem_init();// 初始化共享内存
6.     driver_init();// 初始化设备驱动
7.     init_irq_proc();//创建/proc/irq 目录，并初始化系统中所有中断对应的子目录
8.     do_ctors();// 执行内核的构造函数
9.     usermodehelper_enable();// 启用 usermodehelper
10.    do_initcalls();//遍历 initcall_levels 数组，调用里面的 initcall 函数，这里主要是对设备、驱动、文件系统进行初始化，
11.    //之所有将函数封装到数组进行遍历，主要是为了好扩展
12.
13.    random_int_secret_init();//初始化随机数生成池
14. }
```

2.3.3.3 Init 进程启动源码分析

我们主要是分析 Android Q(10.0) 的 init 的代码。

2.3.3.3.1 涉及源码文件

```
1. platform/system/core/init/main.cpp
2. platform/system/core/init/init.cpp
3. platform/system/core/init/ueventd.cpp
4. platform/system/core/init/selinux.cpp
5. platform/system/core/init/subcontext.cpp
6. platform/system/core/base/logging.cpp
7. platform/system/core/init/first_stage_init.cpp
8. platform/system/core/init/first_stage_main.cpp
9. platform/system/core/init/first_stage_mount.cpp
10. platform/system/core/init/keyutils.h
11. platform/system/core/init/property_service.cpp
```

```
platform/external/selinux/libselinux/src/label.c
platform/system/core/init/signal_handler.cpp
platform/system/core/init/service.cpp
```

2.3.3.3.2 Init 进程入口

前面已经通过 kernel_init,启动了 init 进程，init 进程属于一个守护进程，准确的说，它是 Linux 系统中用户控制的第一个进程，它的进程号为 1。它的生命周期贯穿整个 Linux 内核运行的始终。Android 中所有其它的进程共同的鼻祖均为 init 进程。

可以通过"**adb shell ps |grep init**" 的命令来查看 init 的进程号。

Android Q(10.0) 的 init 入口函数由原先的 init.cpp 调整到了 main.cpp，把各个阶段的操作分离开来，使代码更加简洁命令，接下来我们就从 main 函数开始学习。

[system/core/init/main.cpp]

```
/*
 * 1.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.main 函数有四个参数入口，
 *一是参数中有 ueventd，进入 ueventd_main
 *二是参数中有 subcontext，进入 InitLogging 和 SubcontextMain
 *三是参数中有 selinux_setup，进入 SetupSelinux
 *四是参数中有 second_stage，进入 SecondStageMain
 *3.main 的执行顺序如下：
   *  (1)ueventd_main     init 进程创建子进程 ueventd，
   *       并将创建设备节点文件的工作托付给 ueventd，ueventd 通过两种方式创建设备节点文件
   *  (2)FirstStageMain   启动第一阶段
   *  (3)SetupSelinux      加载 selinux 规则，并设置 selinux 日志,完成 SELinux 相关工作
   *  (4)SecondStageMain   启动第二阶段
 */
int main(int argc, char** argv) {
    //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
    if (!strcmp(basename(argv[0]), "ueventd")) {
```

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

   //当传入的参数个数大于 1 时，执行下面的几个操作
    if (argc > 1) {
        //参数为 subcontext，初始化日志系统，
        if (!strcmp(argv[1], "subcontext")) {
            android::base::InitLogging(argv, &android::base::KernelLogger);

            const BuiltinFunctionMap function_map;
            return SubcontextMain(argc, argv, &function_map);
        }

      //参数为“selinux_setup”,启动 Selinux 安全策略
        if (!strcmp(argv[1], "selinux_setup")) {
            return SetupSelinux(argv);
        }
       //参数为“second_stage”,启动 init 进程第二阶段
        if (!strcmp(argv[1], "second_stage")) {
            return SecondStageMain(argc, argv);
        }
    }
// 默认启动 init 进程第一阶段
    return FirstStageMain(argc, argv);
}
```

2.3.3.3.3 ueventd_main

代码路径：platform/system/core/init/ueventd.cpp

Android 根文件系统的镜像中不存在“/dev”目录，该目录是 init 进程启动后动态创建的。

因此，建立 Android 中设备节点文件的重任，也落在了 init 进程身上。为此，init 进程创建子进程 ueventd，并将创建设备节点文件的工作托付给 ueventd。

ueventd 通过两种方式创建设备节点文件。

第一种方式对应“冷插拔”（Cold Plug），即以预先定义的设备信息为基础，当uevent d 启动后，统一创建设备节点文件。这一类设备节点文件也被称为静态节点文件。

第二种方式对应“热插拔”（Hot Plug），即在系统运行中，当有设备插入 USB 端口时，uevent d 就会接收到这一事件，为插入的设备动态创建设备节点文件。这一类设备节点文件也被称为动态节点文件。

```
1.  int ueventd_main(int argc, char** argv) {
2.      //设置新建文件的默认值,这个与 chmod 相反,这里相当于新建文件后的权限为 666
3.      umask(000);
4.
5.      //初始化内核日志，位于节点/dev/kmsg，此时 logd、logcat 进程还没有起来，
6.      //采用 kernel 的 log 系统，打开的设备节点/dev/kmsg， 那么可通过 cat /dev/kmsg
    来获取内核 log。
7.      android::base::InitLogging(argv, &android::base::KernelLogger);
8.
9.      //注册 selinux 相关的用于打印 log 的回调函数
10.     SelinuxSetupKernelLogging();
11.     SelabelInitialize();
12.
13.     //解析 xml，根据不同 SOC 厂商获取不同的 hardware rc 文件
14.     auto ueventd_configuration = ParseConfig({"/ueventd.rc", "/vendor/uevent
    d.rc",
15.                                               "/odm/ueventd.rc", "/ueventd."
     + hardware + ".rc"});
16.
17.     //冷启动
18.     if (access(COLDBOOT_DONE, F_OK) != 0) {
19.         ColdBoot cold_boot(uevent_listener, uevent_handlers);
20.         cold_boot.Run();
21.     }
22.     for (auto& uevent_handler : uevent_handlers) {
23.         uevent_handler->ColdbootDone();
24.     }
25.
26.     //忽略子进程终止信号
27.     signal(SIGCHLD, SIG_IGN);
28.     // Reap and pending children that exited between the last call to waitpi
    d() and setting SIG_IGN
29.     // for SIGCHLD above.
```

```
30.         //在最后一次调用 waitpid（）和为上面的 sigchld 设置 SIG_IGN 之间退出的获取和挂起的子级
31.     while (waitpid(-1, nullptr, WNOHANG) > 0) {
32.     }
33.
34.     //监听来自驱动的 uevent,进行“热插拔”处理
35.     uevent_listener.Poll([&uevent_handlers](const Uevent& uevent) {
36.         for (auto& uevent_handler : uevent_handlers) {
37.             uevent_handler->HandleUevent(uevent); //热启动，创建设备
38.         }
39.         return ListenerAction::kContinue;
40.     });
41.     return 0;
42. }
```

2.3.3.3.4 init 进程启动第一阶段

代码路径：platform\system\core\init\first_stage_init.cpp

init 进程第一阶段做的主要工作是挂载分区,创建设备节点和一些关键目录,初始化日志输出系统,启用 SELinux 安全策略

第一阶段完成以下内容：

/* 01. 创建文件系统目录并挂载相关的文件系统 */

/* 02. 屏蔽标准的输入输出/初始化内核 log 系统 */

FirstStageMain

```
1.  int FirstStageMain(int argc, char** argv) {
2.      //init crash 时重启引导加载程序
3.      //这个函数主要作用将各种信号量，如 SIGABRT,SIGBUS 等的行为设置为 SA_RESTART,一旦监听到这些信号即执行重启系统
4.      if (REBOOT_BOOTLOADER_ON_PANIC) {
5.          InstallRebootSignalHandlers();
6.      }
7.      //清空文件权限
8.      umask(0);
```

```
9.
10.     CHECKCALL(clearenv());
11.     CHECKCALL(setenv("PATH", _PATH_DEFPATH, 1));
12.
13.     //在 RAM 内存上获取基本的文件系统，剩余的被 rc 文件所用
14.     CHECKCALL(mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755"));
15.     CHECKCALL(mkdir("/dev/pts", 0755));
16.     CHECKCALL(mkdir("/dev/socket", 0755));
17.     CHECKCALL(mount("devpts", "/dev/pts", "devpts", 0, NULL));
18. #define MAKE_STR(x) __STRING(x)
19.     CHECKCALL(mount("proc", "/proc", "proc", 0, "hidepid=2,gid=" MAKE_STR(AI
    D_READPROC)));
20. #undef MAKE_STR
21.
22.     // 非特权应用不能使用 Andrlid cmdline
23.     CHECKCALL(chmod("/proc/cmdline", 0440));
24.     gid_t groups[] = {AID_READPROC};
25.     CHECKCALL(setgroups(arraysize(groups), groups));
26.     CHECKCALL(mount("sysfs", "/sys", "sysfs", 0, NULL));
27.     CHECKCALL(mount("selinuxfs", "/sys/fs/selinux", "selinuxfs", 0, NULL));
28.
29.     CHECKCALL(mknod("/dev/kmsg", S_IFCHR | 0600, makedev(1, 11)));
30.
31.     if constexpr (WORLD_WRITABLE_KMSG) {
32.         CHECKCALL(mknod("/dev/kmsg_debug", S_IFCHR | 0622, makedev(1, 11)));
33.     }
34.
35.     CHECKCALL(mknod("/dev/random", S_IFCHR | 0666, makedev(1, 8)));
36.     CHECKCALL(mknod("/dev/urandom", S_IFCHR | 0666, makedev(1, 9)));
37.
38.
39.     //这对于日志包装器是必需的，它在 ueventd 运行之前被调用
40.     CHECKCALL(mknod("/dev/ptmx", S_IFCHR | 0666, makedev(5, 2)));
41.     CHECKCALL(mknod("/dev/null", S_IFCHR | 0666, makedev(1, 3)));
42.
43.
44.     //在第一阶段挂在 tmpfs、mnt/vendor、mount/product 分区。其他的分区不需要在第
    一阶段加载，
45.     //只需要在第二阶段通过 rc 文件解析来加载。
46.     CHECKCALL(mount("tmpfs", "/mnt", "tmpfs", MS_NOEXEC | MS_NOSUID | MS_NOD
    EV,
47.                     "mode=0755,uid=0,gid=1000"));
```

```
48.
49.     //创建可供读写的 vendor 目录
50.     CHECKCALL(mkdir("/mnt/vendor", 0755));
51.     // /mnt/product is used to mount product-
    specific partitions that can not be
52.     // part of the product partition, e.g. because they are mounted read-
    write.
53.     CHECKCALL(mkdir("/mnt/product", 0755));
54.
55.     // 挂载 APEX，这在 Android 10.0 中特殊引入，用来解决碎片化问题，类似一种组件方
    式，对 Treble 的增强，
56.     // 不写谷歌特殊更新不需要完整升级整个系统版本，只需要像升级 APK 一样，进行 APEX
    组件升级
57.     CHECKCALL(mount("tmpfs", "/apex", "tmpfs", MS_NOEXEC | MS_NOSUID | MS_NO
    DEV,
58.                     "mode=0755,uid=0,gid=0"));
59.
60.     // /debug_ramdisk is used to preserve additional files from the debug ra
    mdisk
61.     CHECKCALL(mount("tmpfs", "/debug_ramdisk", "tmpfs", MS_NOEXEC | MS_NOSUI
    D | MS_NODEV,
62.                     "mode=0755,uid=0,gid=0"));
63. #undef CHECKCALL
64.
65.     //把标准输入、标准输出和标准错误重定向到空设备文件"/dev/null"
66.     SetStdioToDevNull(argv);
67.     //在/dev 目录下挂载好 tmpfs 以及 kmsg
68.     //这样就可以初始化 /kernel Log 系统，供用户打印 log
69.     InitKernelLogging(argv);
70.
71.     ...
72.
73.     /* 初始化一些必须的分区
74.      *主要作用是去解析/proc/device-tree/firmware/android/fstab,
75.      * 然后得到"/system", "/vendor", "/odm"三个目录的挂载信息
76.      */
77.     if (!DoFirstStageMount()) {
78.         LOG(FATAL) << "Failed to mount required partitions early ...";
79.     }
80.
81.     struct stat new_root_info;
82.     if (stat("/", &new_root_info) != 0) {
83.         PLOG(ERROR) << "Could not stat(\"/\"), not freeing ramdisk";
84.         old_root_dir.reset();
```

```
85.     }
86.
87.     if (old_root_dir && old_root_info.st_dev != new_root_info.st_dev) {
88.         FreeRamdisk(old_root_dir.get(), old_root_info.st_dev);
89.     }
90.
91.     SetInitAvbVersionInRecovery();
92.
93.     static constexpr uint32_t kNanosecondsPerMillisecond = 1e6;
94.     uint64_t start_ms = start_time.time_since_epoch().count() / kNanoseconds
    PerMillisecond;
95.     setenv("INIT_STARTED_AT", std::to_string(start_ms).c_str(), 1);
96.
97.     //启动 init 进程，传入参数 selinux_steup
98.     // 执行命令:  /system/bin/init selinux_setup
99.     const char* path = "/system/bin/init";
100.      const char* args[] = {path, "selinux_setup", nullptr};
101.      execv(path, const_cast<char**>(args));
102.      PLOG(FATAL) << "execv(\"" << path << "\") failed";
103.
104.      return 1;
105. }
```

2.3.3.3.5 加载 SELinux 规则

SELinux 是「Security-Enhanced Linux」的简称，是美国国家安全局「NSA=The National Security Agency」

和 SCC（Secure Computing Corporation）开发的 Linux 的一个扩张强制访问控制安全模块。

在这种访问控制体系的限制下，进程只能访问那些在他的任务中所需要文件。

selinux 有两种工作模式：

1. permissive，所有的操作都被允许（即没有 MAC），但是如果违反权限的话，会记录日志,一般 eng 模式用

2. enforcing，所有操作都会进行权限检查。一般 user 和 user-debug 模式用

不管是 security_setenforce 还是 security_getenforce 都是去操作

/sys/fs/selinux/enforce 文件, **0 表示 permissive 1 表示 enforcing**

1、SetupSelinux

说明:初始化 selinux，加载 SELinux 规则，配置 SELinux 相关 log 输出，并启动第二阶段

代码路径： platform\system\core\init\selinux.cpp

```
1.  /*此函数初始化 selinux，然后执行 init 以在 init selinux 中运行*/
2.  int SetupSelinux(char** argv) {
3.          //初始化 Kernel 日志
4.      InitKernelLogging(argv);
5.
6.          // Debug 版本 init crash 时重启引导加载程序
7.      if (REBOOT_BOOTLOADER_ON_PANIC) {
8.          InstallRebootSignalHandlers();
9.      }
10.
11.     //注册回调，用来设置需要写入 kmsg 的 selinux 日志
12.     SelinuxSetupKernelLogging();
13.
14.      //加载 SELinux 规则
15.     SelinuxInitialize();
16.
17.     /*
18.         *我们在内核域中，希望转换到 init 域。在其 xattrs 中存储 selabel 的文件系统
    （如 ext4）不需要显式 restorecon，
19.         *但其他文件系统需要。尤其是对于 ramdisk，如对于 a/b 设备的恢复映像，这是必需
    要做的一步。
20.         *其实就是当前在内核域中，在加载 Seliux 后，需要重新执行 init 切换到 C 空间的用
    户态
21.         */
22.     if (selinux_android_restorecon("/system/bin/init", 0) == -1) {
23.         PLOG(FATAL) << "restorecon failed of /system/bin/init failed";
24.     }
25.
26.   //准备启动 innit 进程，传入参数 second_stage
```

```
    const char* path = "/system/bin/init";
    const char* args[] = {path, "second_stage", nullptr};
    execv(path, const_cast<char**>(args));

    /*
       *执行 /system/bin/init second_stage, 进入第二阶段
       */
    PLOG(FATAL) << "execv(\"" << path << "\") failed";

    return 1;
}
```

2、 SelinuxInitialize()

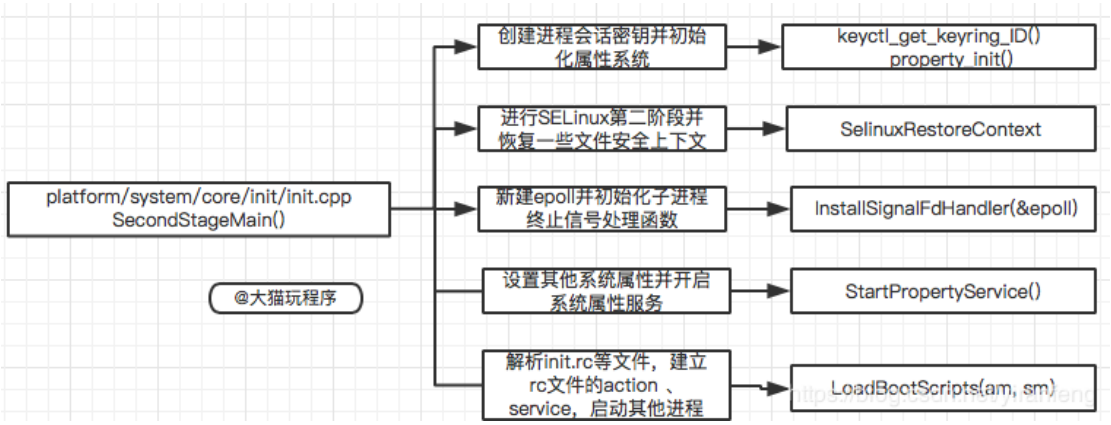
```
/*加载 selinux 规则*/
void SelinuxInitialize() {
    LOG(INFO) << "Loading SELinux policy";
    if (!LoadPolicy()) {
        LOG(FATAL) << "Unable to load SELinux policy";
    }

    //获取当前 Kernel 的工作模式
    bool kernel_enforcing = (security_getenforce() == 1);

    //获取工作模式的配置
    bool is_enforcing = IsEnforcing();

    //如果当前的工作模式与配置的不同，就将当前的工作模式改掉
    if (kernel_enforcing != is_enforcing) {
        if (security_setenforce(is_enforcing)) {
            PLOG(FATAL) << "security_setenforce(" << (is_enforcing ? "true" : "false")
                        << ") failed";
        }
    }

    if (auto result = WriteFile("/sys/fs/selinux/checkreqprot", "0"); !result) {
        LOG(FATAL) << "Unable to write to /sys/fs/selinux/checkreqprot: " << result.error();
    }
}
```

```
/*
 *加载 SELinux 规则
 *这里区分了两种情况,这两种情况只是区分从哪里加载安全策略文件,
 *第一个是从 /vendor/etc/selinux/precompiled_sepolicy 读取,
 *第二个是从 /sepolicy 读取,他们最终都是调用 selinux_android_load_policy_from_fd 方法
 */
bool LoadPolicy() {
    return IsSplitPolicyDevice() ? LoadSplitPolicy() : LoadMonolithicPolicy();
}
```

2.3.3.3.6 init 进程启动第二阶段

第二阶段主要内容:

1. 创建进程会话密钥并初始化属性系统
2. 进行 SELinux 第二阶段并恢复一些文件安全上下文
3. 新建 epoll 并初始化子进程终止信号处理函数，详细看第五节-信号处理
4. 启动匹配属性的服务端， 详细查看第六节-属性服务
5. 解析 init.rc 等文件，建立 rc 文件的 action 、service，启动其他进程，详细查看第七节-rc 文件解析



1、 SecondStageMain

```
int SecondStageMain(int argc, char** argv) {
    /* 01. 创建进程会话密钥并初始化属性系统 */
    keyctl_get_keyring_ID(KEY_SPEC_SESSION_KEYRING, 1);

    //创建 /dev/.booting 文件，就是个标记，表示 booting 进行中
```

```
    close(open("/dev/.booting", O_WRONLY | O_CREAT | O_CLOEXEC, 0000));

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

    /* 02. 进行 SELinux 第二阶段并恢复一些文件安全上下文 */
        SelinuxRestoreContext();

    /* 03. 新建 epoll 并初始化子进程终止信号处理函数 */
    Epoll epoll;
    if (auto result = epoll.Open(); !result) {
        PLOG(FATAL) << result.error();
    }

            InstallSignalFdHandler(&epoll);

    /* 04. 设置其他系统属性并开启系统属性服务*/
    StartPropertyService(&epoll);

            /* 05 解析 init.rc 等文件，建立 rc 文件的 action 、service，启动其他进程*/
    ActionManager& am = ActionManager::GetInstance();
    ServiceList& sm = ServiceList::GetInstance();
    LoadBootScripts(am, sm);
}
```

代码流程详细解析：

```
int SecondStageMain(int argc, char** argv) {
    /*
    *init crash 时重启引导加载程序
    *这个函数主要作用将各种信号量，如 SIGABRT,SIGBUS 等的行为设置为 SA_RESTART,一旦监听到这些信号即执行重启系统
    */
    if (REBOOT_BOOTLOADER_ON_PANIC) {
        InstallRebootSignalHandlers();
    }

    //把标准输入、标准输出和标准错误重定向到空设备文件"/dev/null"
    SetStdioToDevNull(argv);
    //在/dev 目录下挂载好 tmpfs 以及 kmsg
    //这样就可以初始化 /kernel Log 系统，供用户打印 log
    InitKernelLogging(argv);
    LOG(INFO) << "init second stage started!";
```

```
16.
17.     // 01. 创建进程会话密钥并初始化属性系统
18.     keyctl_get_keyring_ID(KEY_SPEC_SESSION_KEYRING, 1);
19.
20.     //创建 /dev/.booting 文件，就是个标记，表示 booting 进行中
21.     close(open("/dev/.booting", O_WRONLY | O_CREAT | O_CLOEXEC, 0000));
22.
23.     // 初始化属性系统，并从指定文件读取属性
24.     property_init();
25.
26.     /*
27.      * 1.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
28.      * 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，
29.      */
30.     process_kernel_dt(); // 处理 DT 属性
31.     process_kernel_cmdline(); // 处理命令行属性
32.
33.     // 处理一些其他的属性
34.     export_kernel_boot_props();
35.
36.     // Make the time that init started available for bootstat to log.
37.     property_set("ro.boottime.init", getenv("INIT_STARTED_AT"));
38.     property_set("ro.boottime.init.selinux", getenv("INIT_SELINUX_TOOK"));
39.
40.     // Set libavb version for Framework-only OTA match in Treble build.
41.     const char* avb_version = getenv("INIT_AVB_VERSION");
42.     if (avb_version) property_set("ro.boot.avb_version", avb_version);
43.
44.     // See if need to load debug props to allow adb root, when the device is
    unlocked.
45.     const char* force_debuggable_env = getenv("INIT_FORCE_DEBUGGABLE");
46.     if (force_debuggable_env && AvbHandle::IsDeviceUnlocked()) {
47.         load_debug_prop = "true"s == force_debuggable_env;
48.     }
49.
50.     // 基于 cmdline 设置 memcg 属性
51.     bool memcg_enabled = android::base::GetBoolProperty("ro.boot.memcg",fals
    e);
52.     if (memcg_enabled) {
53.        // root memory control cgroup
54.        mkdir("/dev/memcg", 0700);
55.        chown("/dev/memcg",AID_ROOT,AID_SYSTEM);
56.        mount("none", "/dev/memcg", "cgroup", 0, "memory");
```

```
57.         // app mem cgroups, used by activity manager, lmkd and zygote
58.         mkdir("/dev/memcg/apps/",0755);
59.         chown("/dev/memcg/apps/",AID_SYSTEM,AID_SYSTEM);
60.         mkdir("/dev/memcg/system",0550);
61.         chown("/dev/memcg/system",AID_SYSTEM,AID_SYSTEM);
62.     }
63.
64.     // 清空这些环境变量，之前已经存到了系统属性中去了
65.     unsetenv("INIT_STARTED_AT");
66.     unsetenv("INIT_SELINUX_TOOK");
67.     unsetenv("INIT_AVB_VERSION");
68.     unsetenv("INIT_FORCE_DEBUGGABLE");
69.
70.     // Now set up SELinux for second stage.
71.     SelinuxSetupKernelLogging();
72.     SelabelInitialize();
73.
74.     /*
75.      * 02. 进行 SELinux 第二阶段并恢复一些文件安全上下文
76.      * 恢复相关文件的安全上下文,因为这些文件是在 SELinux 安全机制初始化前创建的,
77.      * 所以需要重新恢复上下文
78.      */
79.     SelinuxRestoreContext();
80.
81.    /*
82.     * 03. 新建 epoll 并初始化子进程终止信号处理函数
83.     *  创建 epoll 实例，并返回 epoll 的文件描述符
84.     */
85.     Epoll epoll;
86.     if (auto result = epoll.Open(); !result) {
87.         PLOG(FATAL) << result.error();
88.     }
89.
90.     /*
91.      *主要是创建 handler 处理子进程终止信号，注册一个 signal 到 epoll 进行监听
92.      *进行子继承处理
93.      */
94.     InstallSignalFdHandler(&epoll);
95.
96.     // 进行默认属性配置相关的工作
97.     property_load_boot_defaults(load_debug_prop);
98.     UmountDebugRamdisk();
99.     fs_mgr_vendor_overlay_mount_all();
100.     export_oem_lock_status();
```

```
101.
102.     /*
103.      *04. 设置其他系统属性并开启系统属性服务
104.      */
105.     StartPropertyService(&epoll);
106.     MountHandler mount_handler(&epoll);
107.
108.     //为USB存储设置udc Contorller, sys/class/udc
109.     set_usb_controller();
110.
111.     // 匹配命令和函数之间的对应关系
112.     const BuiltinFunctionMap function_map;
113.     Action::set_function_map(&function_map);
114.
115.     if (!SetupMountNamespaces()) {
116.         PLOG(FATAL) << "SetupMountNamespaces failed";
117.     }
118.
119.     // 初始化文件上下文
120.     subcontexts = InitializeSubcontexts();
121.
122.    /*
123.      *05 解析init.rc等文件，建立rc文件的action 、service，启动其他进程
124.      */
125.     ActionManager& am = ActionManager::GetInstance();
126.     ServiceList& sm = ServiceList::GetInstance();
127.
128.     LoadBootScripts(am, sm);
129.
130.     // Turning this on and letting the INFO logging be discarded adds 0.2s
    to
131.     // Nexus 9 boot time, so it's disabled by default.
132.     if (false) DumpState();
133.
134.     // 当GSI脚本running时，确保GSI状态可用.
135.     if (android::gsi::IsGsiRunning()) {
136.         property_set("ro.gsid.image_running", "1");
137.     } else {
138.         property_set("ro.gsid.image_running", "0");
139.     }
140.
141.
142.     am.QueueBuiltinAction(SetupCgroupsAction, "SetupCgroups");
143.
```

```
144.     // 执行 rc 文件中触发器为 on early-init 的语句
145.     am.QueueEventTrigger("early-init");
146.
147.     // 等冷插拔设备初始化完成
148.     am.QueueBuiltinAction(wait_for_coldboot_done_action, "wait_for_coldboot
    _done");
149.
150.     // 开始查询来自 /dev 的 action
151.     am.QueueBuiltinAction(MixHwrngIntoLinuxRngAction, "MixHwrngIntoLinuxRng
    ");
152.     am.QueueBuiltinAction(SetMmapRndBitsAction, "SetMmapRndBits");
153.     am.QueueBuiltinAction(SetKptrRestrictAction, "SetKptrRestrict");
154.
155.     // 设备组合键的初始化操作
156.     Keychords keychords;
157.     am.QueueBuiltinAction(
158.         [&epoll, &keychords](const BuiltinArguments& args) -> Result<Succes
    s> {
159.             for (const auto& svc : ServiceList::GetInstance()) {
160.                 keychords.Register(svc->keycodes());
161.             }
162.             keychords.Start(&epoll, HandleKeychord);
163.             return Success();
164.         },
165.         "KeychordInit");
166.
167.     //在屏幕上显示 Android 静态 LOGO
168.     am.QueueBuiltinAction(console_init_action, "console_init");
169.
170.     // 执行 rc 文件中触发器为 on init 的语句
171.     am.QueueEventTrigger("init");
172.
173.     // Starting the BoringSSL self test, for NIAP certification compliance.

174.     am.QueueBuiltinAction(StartBoringSslSelfTest, "StartBoringSslSelfTest")
    ;
175.
176.     // Repeat mix_hwrng_into_linux_rng in case /dev/hw_random or /dev/rando
    m
177.     // wasn't ready immediately after wait_for_coldboot_done
178.     am.QueueBuiltinAction(MixHwrngIntoLinuxRngAction, "MixHwrngIntoLinuxRng
    ");
179.
180.     // Initialize binder before bringing up other system services
```

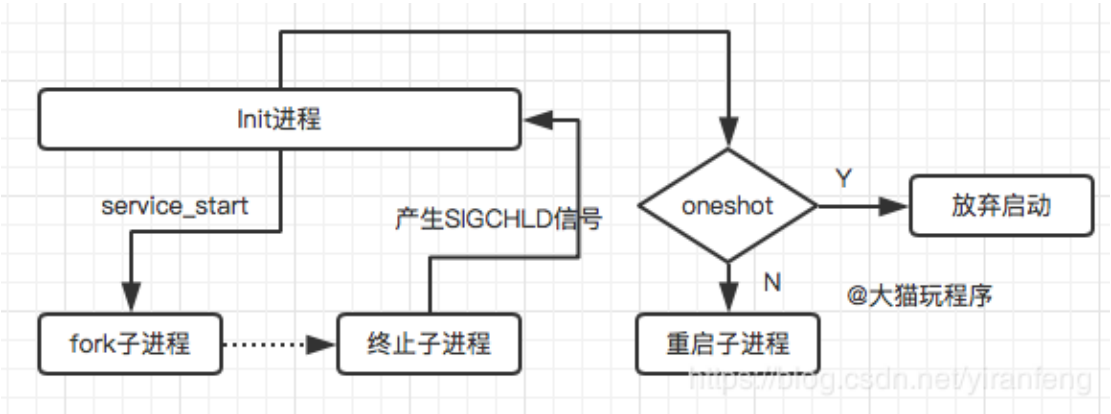
```
    am.QueueBuiltinAction(InitBinder, "InitBinder");

    // 当设备处于充电模式时，不需要 mount 文件系统或者启动系统服务
    // 充电模式下，将 charger 假如执行队列，否则把 late-init 假如执行队列
    std::string bootmode = GetProperty("ro.bootmode", "");
    if (bootmode == "charger") {
        am.QueueEventTrigger("charger");
    } else {
        am.QueueEventTrigger("late-init");
    }

    // 基于属性当前状态 运行所有的属性触发器.
    am.QueueBuiltinAction(queue_property_triggers_action, "queue_property_triggers");

    while (true) {
        // By default, sleep until something happens.
        auto epoll_timeout = std::optional<std::chrono::milliseconds>{};

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

//依次执行每个 action 中携带 command 对应的执行函数
        if (!(waiting_for_prop || Service::is_exec_service_running())) {
            am.ExecuteOneCommand();
        }
        if (!(waiting_for_prop || Service::is_exec_service_running())) {
            if (!shutting_down) {
                auto next_process_action_time = HandleProcessActions();

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

                    if (*epoll_timeout < 0ms) epoll_timeout = 0ms;
                }
            }
```

```
221.
222.             // If there's more work to do, wake up again immediately.
223.             if (am.HasMoreCommands()) epoll_timeout = 0ms;
224.         }
225.
226.         // 循环等待事件发生
227.         if (auto result = epoll.Wait(epoll_timeout); !result) {
228.             LOG(ERROR) << result.error();
229.         }
230.     }
231.
232.     return 0;
233. }
```

2.3.3.4 信号处理

init 是一个守护进程，为了防止 init 的子进程成为僵尸进程(zombie process)，需要 init 在子进程在结束时获取子进程的结束码，通过结束码将程序表中的子进程移除，防止成为僵尸进程的子进程占用程序表的空间（程序表的空间达到上限时，系统就不能再启动新的进程了，会引起严重的系统问题）。

子进程重启流程如下图所示：



信号处理主要工作：

- 初始化信号 signal 句柄
- 循环处理子进程
- 注册 epoll 句柄

- 处理子进程终止

注: EPOLL 类似于 POLL，是 Linux 中用来做事件触发的，跟 EventBus 功能差不多。linux 很长的时间都在使用 select 来做事件触发，它是通过轮询来处理的，轮询的 fd 数目越多，自然耗时越多，对于大量的描述符处理，EPOLL 更有优势

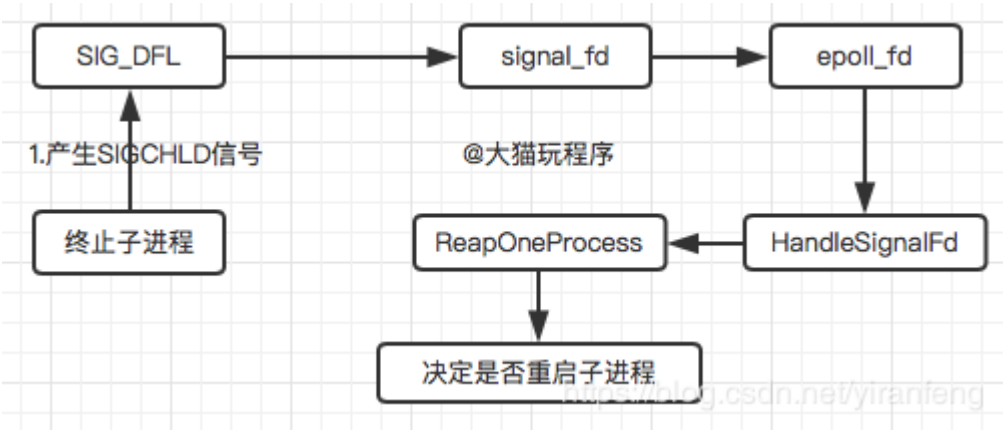
2.3.3.4.1 InstallSignalFdHandler

在 linux 当中，父进程是通过捕捉 SIGCHLD 信号来得知子进程运行结束的情况，SIGCHLD 信号会在子进程终止的时候发出，了解这些背景后，我们来看看 init 进程如何处理这个信号。

- 首先，新建一个 sigaction 结构体，sa_handler 是信号处理函数，指向内核指定的函数指针 SIG_DFL 和 Android 9.0 及之前的版本不同，这里不再通过 socket 的读写句柄进行接收信号，改成了内核的信号处理函数 SIG_DFL。
- 然后，sigaction(SIGCHLD, &act, nullptr) 这个是建立信号绑定关系，也就是说当监听到 SIGCHLD 信号时，由 act 这个 sigaction 结构体处理
- 最后，RegisterHandler 的作用就是 signal_read_fd（之前的 s[1]）收到信号，触发 handle_signal

终上所述，InstallSignalFdHandler 函数的作用就是，接收到 SIGCHLD 信号时触发 HandleSignalFd 进行信号处理

信号处理示意图：



代码路径： platform/system/core/init.cpp

说明： 该函数主要的作用是初始化子进程终止信号处理过程

```
1.  static void InstallSignalFdHandler(Epoll* epoll) {
2.
3.      // SA_NOCLDSTOP 使 init 进程只有在其子进程终止时才会受到 SIGCHLD 信号
4.      const struct sigaction act { .sa_handler = SIG_DFL, .sa_flags = SA_NOCLD
    STOP };
5.      sigaction(SIGCHLD, &act, nullptr);
6.
7.      sigset_t mask;
8.      sigemptyset(&mask);
9.      sigaddset(&mask, SIGCHLD);
10.
11.     if (!IsRebootCapable()) {
12.         // 如果 init 不具有 CAP_SYS_BOOT 的能力，则它此时正值容器中运行
13.         // 在这种场景下，接收 SIGTERM 将会导致系统关闭
14.         sigaddset(&mask, SIGTERM);
15.     }
16.
17.     if (sigprocmask(SIG_BLOCK, &mask, nullptr) == -1) {
18.         PLOG(FATAL) << "failed to block signals";
19.     }
20.
21.     // 注册处理程序以解除对子进程中的信号的阻止
22.     const int result = pthread_atfork(nullptr, nullptr, &UnblockSignals);
23.     if (result != 0) {
24.         LOG(FATAL) << "Failed to register a fork handler: " << strerror(resu
    lt);
25.     }
26.
27.     //创建信号句柄
28.     signal_fd = signalfd(-1, &mask, SFD_CLOEXEC);
29.     if (signal_fd == -1) {
30.         PLOG(FATAL) << "failed to create signalfd";
31.     }
32.
33.     //信号注册，当 signal_fd 收到信号时，触发 HandleSignalFd
34.     if (auto result = epoll->RegisterHandler(signal_fd, HandleSignalFd); !re
    sult) {
35.         LOG(FATAL) << result.error();
36.     }
37. }
```

2. 3. 3. 4. 2 RegisterHandler

代码路径：/platform/system/core/epoll.cpp

说明：信号注册,把 fd 句柄加入到 epoll_fd_的监听队列中

```
Result<void> Epoll::RegisterHandler(int fd, std::function<void()> handler, uint32_t events) {
    if (!events) {
        return Error() << "Must specify events";
    }
    auto [it, inserted] = epoll_handlers_.emplace(fd, std::move(handler));
    if (!inserted) {
        return Error() << "Cannot specify two epoll handlers for a given FD";
    }
    epoll_event ev;
    ev.events = events;
    // std::map's iterators do not get invalidated until erased, so we use the
    // pointer to the std::function in the map directly for epoll_ctl.
    ev.data.ptr = reinterpret_cast<void*>(&it->second);
    // 将 fd 的可读事件加入到 epoll_fd_的监听队列中
    if (epoll_ctl(epoll_fd_, EPOLL_CTL_ADD, fd, &ev) == -1) {
        Result<void> result = ErrnoError() << "epoll_ctl failed to add fd";
        epoll_handlers_.erase(fd);
        return result;
    }
    return {};
}
```

2. 3. 3. 4. 3 HandleSignalFd

代码路径：platform/system/core/init.cpp

说明：监控 SIGCHLD 信号，调用 ReapAnyOutstandingChildren 来 终止出现问题的子进程

```
static void HandleSignalFd() {
```

```
    signalfd_siginfo siginfo;
    ssize_t bytes_read = TEMP_FAILURE_RETRY(read(signal_fd, &siginfo, sizeof(siginfo)));
    if (bytes_read != sizeof(siginfo)) {
        PLOG(ERROR) << "Failed to read siginfo from signal_fd";
        return;
    }

   //监控 SIGCHLD 信号
    switch (siginfo.ssi_signo) {
        case SIGCHLD:
            ReapAnyOutstandingChildren();
            break;
        case SIGTERM:
            HandleSigtermSignal(siginfo);
            break;
        default:
            PLOG(ERROR) << "signal_fd: received unexpected signal " << siginfo.ssi_signo;
            break;
    }
}
```

2. 3. 3. 4. 4 ReapOneProcess

代码路径：/platform/system/core/sigchld_handle.cpp

说明：ReapOneProcess 是最终的处理函数了，这个函数先用 waitpid 找出挂掉进程的 pid,然后根据 pid 找到对应 Service，

最后调用 Service 的 Reap 方法清除资源,根据进程对应的类型，决定是否重启机器或重启进程

```
void ReapAnyOutstandingChildren() {
    while (ReapOneProcess()) {
    }
}

static bool ReapOneProcess() {
    siginfo_t siginfo = {};
    //用 waitpid 函数获取状态发生变化的子进程 pid
```

```
9.      //waitpid 的标记为 WNOHANG，即非阻塞，返回为正值就说明有进程挂掉了
10.     if (TEMP_FAILURE_RETRY(waitid(P_ALL, 0, &siginfo, WEXITED | WNOHANG | WN
    OWAIT)) != 0) {
11.         PLOG(ERROR) << "waitid failed";
12.         return false;
13.     }
14.
15.     auto pid = siginfo.si_pid;
16.     if (pid == 0) return false;
17.
18.     // 当我们知道当前有一个僵尸 pid，我们使用 scopeguard 来清楚该 pid
19.     auto reaper = make_scope_guard([pid] { TEMP_FAILURE_RETRY(waitpid(pid, n
    ullptr, WNOHANG)); });
20.
21.     std::string name;
22.     std::string wait_string;
23.     Service* service = nullptr;
24.
25.     if (SubcontextChildReap(pid)) {
26.         name = "Subcontext";
27.     } else {
28.         //通过 pid 找到对应的 service
29.         service = ServiceList::GetInstance().FindService(pid, &Service::pid)
    ;
30.
31.         if (service) {
32.             name = StringPrintf("Service '%s' (pid %d)", service->name().c_s
    tr(), pid);
33.             if (service->flags() & SVC_EXEC) {
34.                 auto exec_duration = boot_clock::now() - service->time_start
    ed();
35.                 auto exec_duration_ms =
36.                     std::chrono::duration_cast<std::chrono::milliseconds>(ex
    ec_duration).count();
37.                 wait_string = StringPrintf(" waiting took %f seconds", exec_
    duration_ms / 1000.0f);
38.             } else if (service->flags() & SVC_ONESHOT) {
39.                 auto exec_duration = boot_clock::now() - service->time_start
    ed();
40.                 auto exec_duration_ms =
41.                         std::chrono::duration_cast<std::chrono::milliseconds
    >(exec_duration)
42.                                 .count();
```

```
43.                 wait_string = StringPrintf(" oneshot service took %f seconds
    in background",exec_duration_ms / 1000.0f);
44.             }
45.         } else {
46.             name = StringPrintf("Untracked pid %d", pid);
47.         }
48.     }
49.
50.     if (siginfo.si_code == CLD_EXITED) {
51.         LOG(INFO) << name << " exited with status " << siginfo.si_status <<
    wait_string;
52.     } else {
53.         LOG(INFO) << name << " received signal " << siginfo.si_status << wai
    t_string;
54.     }
55.
56.     //没有找到 service，说明已经结束了，退出
57.     if (!service) return true;
58.
59.     service->Reap(siginfo);//清除子进程相关的资源
60.
61.     if (service->flags() & SVC_TEMPORARY) {
62.         ServiceList::GetInstance().RemoveService(*service); //移除该
    service
63.     }
64.
65.     return true;
66. }
```

2.3.3.5 属性服务

我们在开发和调试过程中看到通过 property_set 可以轻松设置系统属性，那干嘛这里还要启动一个属性服务呢？这里其实涉及到一些权限的问题，不是所有进程都可以随意修改任何的系统属性，

Android 将属性的设置统一交由 init 进程管理，其他进程不能直接修改属性，而只能通知 init 进程来修改，而在这过程中，init 进程可以进行权限控制，我们来看看具体的流程是什么

2. 3. 3. 5. 1 property_init

代码路径：platform/system/core/property_service.cpp

说明：初始化属性系统，并从指定文件读取属性，并进行 SELinux 注册，进行属性权限控制

清除缓存，这里主要是清除几个链表以及在内存中的映射，新建 property_filename 目录，这个目录的值为 /dev/_properties_

然后就是调用 CreateSerializedPropertyInfo 加载一些系统属性的类别信息，最后将加载的链表写入文件并映射到内存

```
1.  void property_init() {
2.
3.      //设置 SELinux 回调，进行权限控制
4.      selinux_callback cb;
5.      cb.func_audit = PropertyAuditCallback;
6.      selinux_set_callback(SELINUX_CB_AUDIT, cb);
7.
8.      mkdir("/dev/__properties__", S_IRWXU | S_IXGRP | S_IXOTH);
9.      CreateSerializedPropertyInfo();
10.     if (__system_property_area_init()) {
11.         LOG(FATAL) << "Failed to initialize property area";
12.     }
13.     if (!property_info_area.LoadDefaultPath()) {
14.         LOG(FATAL) << "Failed to load serialized property info file";
15.     }
16. }
```

通过 CreateSerializedPropertyInfo 来加载以下目录的 contexts：

1)与 SELinux 相关

```
1.  /system/etc/selinux/plat_property_contexts
2.
3.  /vendor/etc/selinux/vendor_property_contexts
```

```
4.
5.  /vendor/etc/selinux/nonplat_property_contexts
6.
7.  /product/etc/selinux/product_property_contexts
8.
9.  /odm/etc/selinux/odm_property_contexts
```

2)与 SELinux 无关

```
1.  /plat_property_contexts
2.
3.  /vendor_property_contexts
4.  /nonplat_property_contexts
5.
6.  /product_property_contexts
7.  /odm_property_contexts
```

2. 3. 3. 5. 2 StartPropertyService

代码路径： platform/system/core/init.cpp

说明：启动属性服务

首先创建一个 socket 并返回文件描述符，然后设置最大并发数为 8，其他进程可以通过这个 socket 通知 init 进程修改系统属性，

最后注册 epoll 事件，也就是当监听到 property_set_fd 改变时调用 handle_property_set_fd

```
1.  void StartPropertyService(Epoll* epoll) {
2.      property_set("ro.property_service.version", "2");
3.
4.      //建立 socket 连接
5.      if (auto result = CreateSocket(PROP_SERVICE_NAME, SOCK_STREAM | SOCK_CLO
    EXEC | SOCK_NONBLOCK,
6.                                     false, 0666, 0, 0, {})) {
7.          property_set_fd = *result;
8.      } else {
```

```
9.          PLOG(FATAL) << "start_property_service socket creation failed: " <<
    result.error();
10.     }
11.
12.     // 最大监听 8 个并发
13.     listen(property_set_fd, 8);
14.
15.     // 注册 property_set_fd，当收到句柄改变时，通过 handle_property_set_fd 来处
    理
16.     if (auto result = epoll->RegisterHandler(property_set_fd, handle_propert
    y_set_fd); !result) {
17.         PLOG(FATAL) << result.error();
18.     }
19. }
```

2. 3. 3. 5. 3 handle_property_set_fd

代码路径：platform/system/core/property_service.cpp

说明：建立 socket 连接，然后从 socket 中读取操作信息，根据不同的操作类型，调用 HandlePropertySet 做具体的操作

HandlePropertySet 是最终的处理函数，以"ctl"开头的 key 就做一些 Service 的 Start,Stop,Restart 操作，其他的就是调用 property_set 进行属性设置，不管是前者还是后者，都要进行 SELinux 安全性检查，只有该进程有操作权限才能执行相应操作

```
1.  static void handle_property_set_fd() {
2.      static constexpr uint32_t kDefaultSocketTimeout = 2000; /* ms */
3.
4.      // 等待客户端连接
5.      int s = accept4(property_set_fd, nullptr, nullptr, SOCK_CLOEXEC);
6.      if (s == -1) {
7.          return;
8.      }
9.
10.     ucred cr;
11.     socklen_t cr_size = sizeof(cr);
12.     // 获取连接到此 socket 的进程的凭据
13.     if (getsockopt(s, SOL_SOCKET, SO_PEERCRED, &cr, &cr_size) < 0) {
14.         close(s);
```

```
15.         PLOG(ERROR) << "sys_prop: unable to get SO_PEERCRED";
16.         return;
17.     }
18.
19.     // 建立 socket 连接
20.     SocketConnection socket(s, cr);
21.     uint32_t timeout_ms = kDefaultSocketTimeout;
22.
23.     uint32_t cmd = 0;
24.     // 读取 socket 中的操作信息
25.     if (!socket.RecvUint32(&cmd, &timeout_ms)) {
26.         PLOG(ERROR) << "sys_prop: error while reading command from the socke
    t";
27.         socket.SendUint32(PROP_ERROR_READ_CMD);
28.         return;
29.     }
30.
31.     // 根据操作信息，执行对应处理,两者区别一个是以 char 形式读取，一个以 String 形式
    读取
32.     switch (cmd) {
33.     case PROP_MSG_SETPROP: {
34.         char prop_name[PROP_NAME_MAX];
35.         char prop_value[PROP_VALUE_MAX];
36.
37.         if (!socket.RecvChars(prop_name, PROP_NAME_MAX, &timeout_ms) ||
38.             !socket.RecvChars(prop_value, PROP_VALUE_MAX, &timeout_ms)) {
39.           PLOG(ERROR) << "sys_prop(PROP_MSG_SETPROP): error while reading na
    me/value from the socket";
40.           return;
41.         }
42.
43.         prop_name[PROP_NAME_MAX-1] = 0;
44.         prop_value[PROP_VALUE_MAX-1] = 0;
45.
46.         std::string source_context;
47.         if (!socket.GetSourceContext(&source_context)) {
48.             PLOG(ERROR) << "Unable to set property '" << prop_name << "': ge
    tpeercon() failed";
49.             return;
50.         }
51.
52.         const auto& cr = socket.cred();
53.         std::string error;
```

```
54.         uint32_t result = HandlePropertySet(prop_name, prop_value, source_co
    ntext, cr, &error);
55.         if (result != PROP_SUCCESS) {
56.             LOG(ERROR) << "Unable to set property '" << prop_name << "' from
     uid:" << cr.uid
57.                        << " gid:" << cr.gid << " pid:" << cr.pid << ": " <<
    error;
58.         }
59.
60.         break;
61.       }
62.
63.     case PROP_MSG_SETPROP2: {
64.         std::string name;
65.         std::string value;
66.         if (!socket.RecvString(&name, &timeout_ms) ||
67.             !socket.RecvString(&value, &timeout_ms)) {
68.           PLOG(ERROR) << "sys_prop(PROP_MSG_SETPROP2): error while reading n
    ame/value from the socket";
69.           socket.SendUint32(PROP_ERROR_READ_DATA);
70.           return;
71.         }
72.
73.         std::string source_context;
74.         if (!socket.GetSourceContext(&source_context)) {
75.             PLOG(ERROR) << "Unable to set property '" << name << "': getpeer
    con() failed";
76.             socket.SendUint32(PROP_ERROR_PERMISSION_DENIED);
77.             return;
78.         }
79.
80.         const auto& cr = socket.cred();
81.         std::string error;
82.         uint32_t result = HandlePropertySet(name, value, source_context, cr,
     &error);
83.         if (result != PROP_SUCCESS) {
84.             LOG(ERROR) << "Unable to set property '" << name << "' from uid:
    " << cr.uid
85.                        << " gid:" << cr.gid << " pid:" << cr.pid << ": " <<
    error;
86.         }
87.         socket.SendUint32(result);
88.         break;
89.       }
```

```
90.
91.     default:
92.         LOG(ERROR) << "sys_prop: invalid command " << cmd;
93.         socket.SendUint32(PROP_ERROR_INVALID_CMD);
94.         break;
95.     }
96. }
```

2.3.3.6 第三阶段 init.rc

当属性服务建立完成后，init 的自身功能基本就告一段落，接下来需要来启动其他的进程。但是 init 进程如何其他其他进程呢？其他进程都是一个二进制文件，我们可以直接通过 exec 的命令方式来启动，例如 ./system/bin/init second_stage，来启动 init 进程的第二阶段。但是 Android 系统有那么多的 Native 进程，如果都通过传 exec 在代码中一个个的来执行进程，那无疑是一个灾难性的设计。

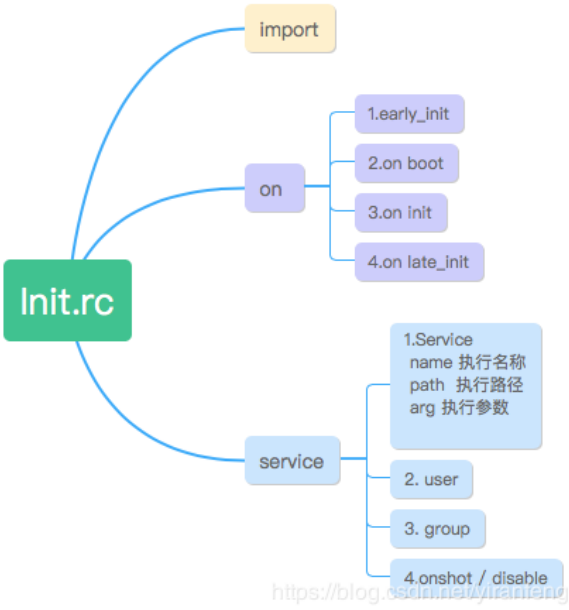
在这个基础上 Android 推出了一个 init.rc 的机制，即类似通过读取配置文件的方式，来启动不同的进程。接下来我们就来看看 init.rc 是如何工作的。

init.rc 是一个配置文件，内部由 Android 初始化语言编写（Android Init Language）编写的脚本。

init.rc 在手机的目录：./init.rc

init.rc 主要包含五种类型语句：

- Action
- Command
- Service
- Option
- Import



2.3.3.6.1 Action

动作表示了一组命令(commands)组成.动作包括一个触发器，决定了何时运行这个动作

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;
- on property:<key>=<value>: 当属性值满足条件时触发;

2.3.3.6.2 Command

command 是 action 的命令列表中的命令，或者是 service 中的选项 onrestart 的参数命令,命令将在所属事件发生时被一个个地执行.

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；
- class_stop <service_class_name> ：停止指定类的服务
- 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 <name> <value>：设置属性值
- mkdir <path>：创建指定目录
- symlink <target> <sym_link>： 创建连接到<target>的<sym_link>符号链接；
- write <path> <string>： 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 <name> <name>：设定环境变量；
- loglevel <level>：设置 log 级别
- hostname <name> ：设置主机名
- import <filename> ：导入一个额外的 init 配置文件

2.3.3.6.3 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。

命令：service <name><pathname> [<argument>]* <option> <option>

参数	含义
<name>	表示此服务的名称
<pathname>	此服务所在路径因为是可执行文件，所以一定有存储路径。
<argument>	启动服务所带的参数
<option>	对此服务的约束选项

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。

2.3.3.6.4 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。

2.3.3.6.5 import

用来导入其他的 rc 文件

命令：import <filename>

2.3.3.7 init.rc 解析过程

2.3.3.7.1 LoadBootScripts

代码路径：platform\system\core\init\init.cpp

说明：如果没有特殊配置 ro.boot.init_rc，则解析./init.rc

把/system/etc/init,/product/etc/init,/product_services/etc/init,/odm/etc/init,

/vendor/etc/init 这几个路径加入 init.rc 之后解析的路径，在 init.rc 解析完成后，解析这些目录里的 rc 文件

```
static void LoadBootScripts(ActionManager& action_manager, ServiceList& service_list) {
    Parser parser = CreateParser(action_manager, service_list);

    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("/product_services/etc/init")) {
            late_import_paths.emplace_back("/product_services/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);
    }
}
```

Android7.0 后，init.rc 进行了拆分，每个服务都有自己的 rc 文件，他们基本上都被加载到 /system/etc/init，/vendor/etc/init, /odm/etc/init 等目录，等 init.rc 解析完成后，会来解析这些目录中的 rc 文件，用来执行相关的动作。

代码路径：platform\system\core\init\init.cpp

说明：创建 Parser 解析对象，例如 service、on、import 对象

```
Parser CreateParser(ActionManager& action_manager, ServiceList& service_list
) {
    Parser parser;

    parser.AddSectionParser(
            "service", std::make_unique<ServiceParser>(&service_list, subcon
texts, std::nullopt));
    parser.AddSectionParser("on", std::make_unique<ActionParser>(&action_man
ager, subcontexts));
    parser.AddSectionParser("import", std::make_unique<ImportParser>(&parser
));

    return parser;
}
```

2.3.3.7.2 执行 Action 动作

按顺序把相关 Action 加入触发器队列，按顺序为 early-init -> init -> late-init. 然后在循环中，执行所有触发器队列中 Action 带 Command 的执行函数。

```
am.QueueEventTrigger("early-init");
am.QueueEventTrigger("init");
am.QueueEventTrigger("late-init");
...
while (true) {
if (!(waiting_for_prop || Service::is_exec_service_running())) {
            am.ExecuteOneCommand();
        }
}
```

2.3.3.7.3 Zygote 启动

从 Android 5.0 的版本开始，Android 支持 64 位的编译，因此 zygote 本身也支持 32 位和 64 位。通过属性 ro.zygote 来控制不同版本的 zygote 进程启动。

在 init.rc 的 import 段我们看到如下代码：

```
1. import /init.${ro.zygote}.rc // 可以看出 init.rc 不再直接引入一个固定的文件，而是根据属性 ro.zygote 的内容来引入不同的文件
```

init.rc 位于/system/core/rootdir 下。在这个路径下还包括四个关于 zygote 的 rc 文件。

分别是 init.zygote32.rc，init.zygote32_64.rc，init.zygote64.rc，init.zygote64_32.rc，由硬件决定调用哪个文件。

这里拿 64 位处理器为例，**init.zygote64.rc 的代码如下所示**：

```
1. service zygote /system/bin/app_process64 -Xzygote /system/bin --zygote --start-system-server
2.     class main      # class 是一个 option，指定 zygote 服务的类型为 main
3.             priority -20
4.             user root
5.         group root readproc reserved_disk
6.             socket zygote stream 660 root system  # socket 关键字表示一个 option，创建一个名为 dev/socket/zygote，类型为 stream，权限为 660 的 socket
7.         socket usap_pool_primary stream 660 root system
8.             onrestart write /sys/android_power/request_state wake # onrestart 是一个 option，说明在 zygote 重启时需要执行的 command
9.             onrestart write /sys/power/state on
10.    onrestart restart audioserver
11.    onrestart restart cameraserver
12.    onrestart restart media
13.    onrestart restart netd
14.    onrestart restart wificond
15.    writepid /dev/cpuset/foreground/tasks
```

service zygote /system/bin/app_process64 -Xzygote /system/bin --zygote --start-system-server 解析：

service zygote ： init.zygote64.rc 中定义了一个 zygote 服务。 init 进程就是通过这个 service 名称来创建 zygote 进程

/system/bin/app_process64 -Xzygote /system/bin --zygote --start-system-server

解析：

zygote 这个服务，通过执行进行/system/bin/app_process64 并传入 4 个参数进行运行：

- 参数 1：-Xzygote 该参数将作为虚拟机启动时所需的参数
- 参数 2：/system/bin 代表虚拟机程序所在目录
- 参数 3：--zygote 指明以 ZygoteInit.java 类中的 main 函数作为虚拟机执行入口
- 参数 4：--start-system-server 告诉 Zygote 进程启动 systemServer 进程

2.3.3.8 总结

init 进程第一阶段做的主要工作是挂载分区,创建设备节点和一些关键目录,初始化日志输出系统,启用 SELinux 安全策略。

init 进程第二阶段主要工作是初始化属性系统，解析 SELinux 的匹配规则，处理子进程终止信号，启动系统属性服务，可以说每一项都很关键，如果说第一阶段是为属性系统，SELinux 做准备，那么第二阶段就是真正去把这些功能落实。

init 进行第三阶段主要是解析 init.rc 来启动其他进程，进入无限循环，进行子进程实时监控。

2.4 Android 系统启动之 Zygote 进程

2.4.1 概要

解了 InIt 进程的整个启动流程。Init 进程启动后，最重要的一个进程就是 Zygote 进程,Zygote 是所有应用的鼻祖。SystemServer 和其他所有 Dalivik 虚拟机进程都是由 Zygote fork 而来。

Zygote 进程由 app_process 启动，Zygote 是一个 C/S 模型，Zygote 进程作为服务端，其他进程作为客户端向它发出“孵化-fork”请求，而 Zygote 接收到这个请求后就“孵化-fork”出一个新的进程。

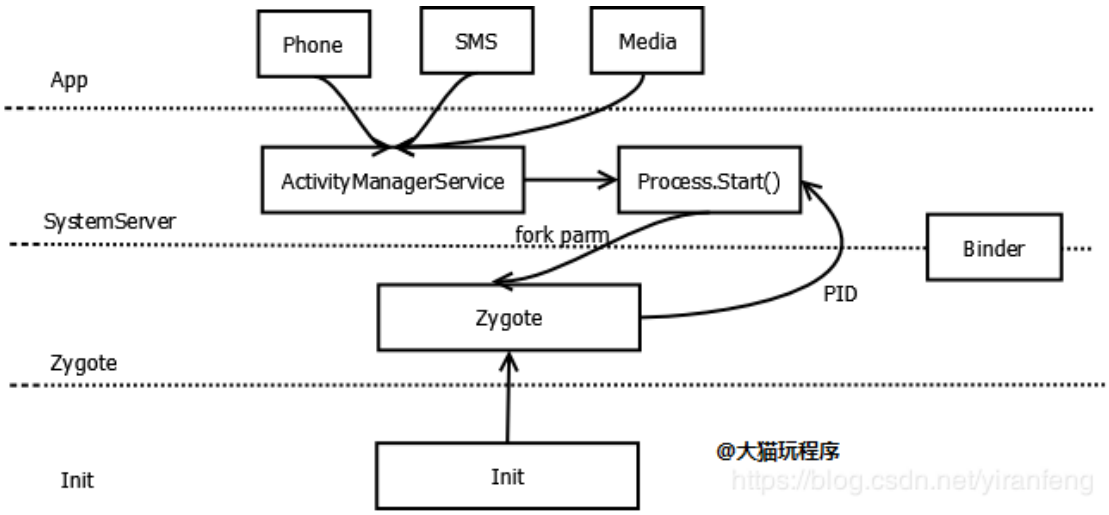
由于 Zygote 进程在启动时会创建 Java 虚拟机，因此通过 fork 而创建的应用程序进程和 SystemServer 进程可以在内部获取一个 Java 虚拟机的实例拷贝。

2.4.2 核心源码

```
1. /system/core/rootdir/init.rc
2. /system/core/init/main.cpp
3. /system/core/init/init.cpp
4. /system/core/rootdir/init.zygote64_32.rc
5. /frameworks/base/cmds/app_process/app_main.cpp
6. /frameworks/base/core/jni/AndroidRuntime.cpp
7. /libnativehelper/JniInvocation.cpp
8. /frameworks/base/core/java/com/android/internal/os/Zygote.java
9. /frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
10. /frameworks/base/core/java/com/android/internal/os/ZygoteServer.java
11. /frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java
12. /frameworks/base/core/java/com/android/internal/os/RuntimeInit.java
13. /frameworks/base/core/java/android/net/LocalServerSocket.java
14. /system/core/libutils/Threads.cpp
```

2.4.3 架构

2.4.3.1 架构图



2.4.3.2 Zygote 是如何被启动的

rc 解析和进程调用

Init 进程启动后，会解析 init.rc 文件，然后创建和加载 service 字段指定的进程。zygote 进程就是以这种方式，被 init 进程加载的。

在 /system/core/rootdir/init.rc 中，通过如下引用来 load Zygote 的 rc：

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

其中${ro.zygote} 由各个厂家使用，现在的主流厂家基本使用 zygote64_32，因此，我们的 rc 文件为 init.zygote64_32.rc

2.4.3.2.1 init.zygote64_32.rc

第一个 Zygote 进程：

进程名：zygote

进程通过 /system/bin/app_process64 来启动

启动参数：-Xzygote /system/bin --zygote --start-system-server --socket-name=zygote

socket 的名称：zygote

```
service zygote /system/bin/app_process64 -Xzygote /system/bin --zygote --start-system-server --socket-name=zygote
            class main
            priority -20
            user root                                   //用户为 root
            group root readproc reserved_disk           //访问组支持 root readproc reserved_disk
            socket zygote stream 660 root system        //创建一个 socket，名字叫 zygote，以 tcp 形式 ，可以在/dev/socket 中看到一个 zygote 的 socket
            socket usap_pool_primary stream 660 root system
            onrestart write /sys/android_power/request_state wake       // on restart 指当进程重启时执行后面的命令
            onrestart write /sys/power/state on
            onrestart restart audioserver
            onrestart restart cameraserver
            onrestart restart media
            onrestart restart netd
            onrestart restart wificond
            onrestart restart vendor.servicetracker-1-1
            writepid /dev/cpuset/foreground/tasks       // 创建子进程时，向 /dev/cpuset/foreground/tasks 写入 pid
```

第二个 Zygote 进程：

zygote_secondary

进程通过 /system/bin/app_process32 来启动

启动参数：-Xzygote /system/bin --zygote --socket-name=zygote_secondary --enable-lazy-preload

socket 的名称：zygote_secondary

```
service zygote_secondary /system/bin/app_process32 -Xzygote /system/bin --zygote --socket-name=zygote_secondary --enable-lazy-preload
            class main
            priority -20
            user root
            group root readproc reserved_disk
            socket zygote_secondary stream 660 root system      //创建一个 socket，名字叫 zygote_secondary，以 tcp 形式 ，可以在/dev/socket 中看到一个 zygote_secondary 的 socket
            socket usap_pool_secondary stream 660 root system
            onrestart restart zygote
            writepid /dev/cpuset/foreground/tasks
```

从上面我们可以看出，zygote 是通过进程文件 /system/bin/app_process64 和 /system/bin/app_process32 来启动的。对应的代码入口为:

frameworks/base/cmds/app_process/app_main.cpp

2.4.3.2.2 Zygote 进程在什么时候会被重启

Zygote 进程重启，主要查看 rc 文件中有没有 “restart zygote” 这句话。在整个 Android 系统工程中**搜索“restart zygote”**，会发现以下文件:

```
/frameworks/native/services/inputflinger/host/inputflinger.rc       对应进程： inputflinger
/frameworks/native/cmds/servicemanager/servicemanager.rc            对应进程： servicemanager
/frameworks/native/services/surfaceflinger/surfaceflinger.rc        对应进程： surfaceflinger
```

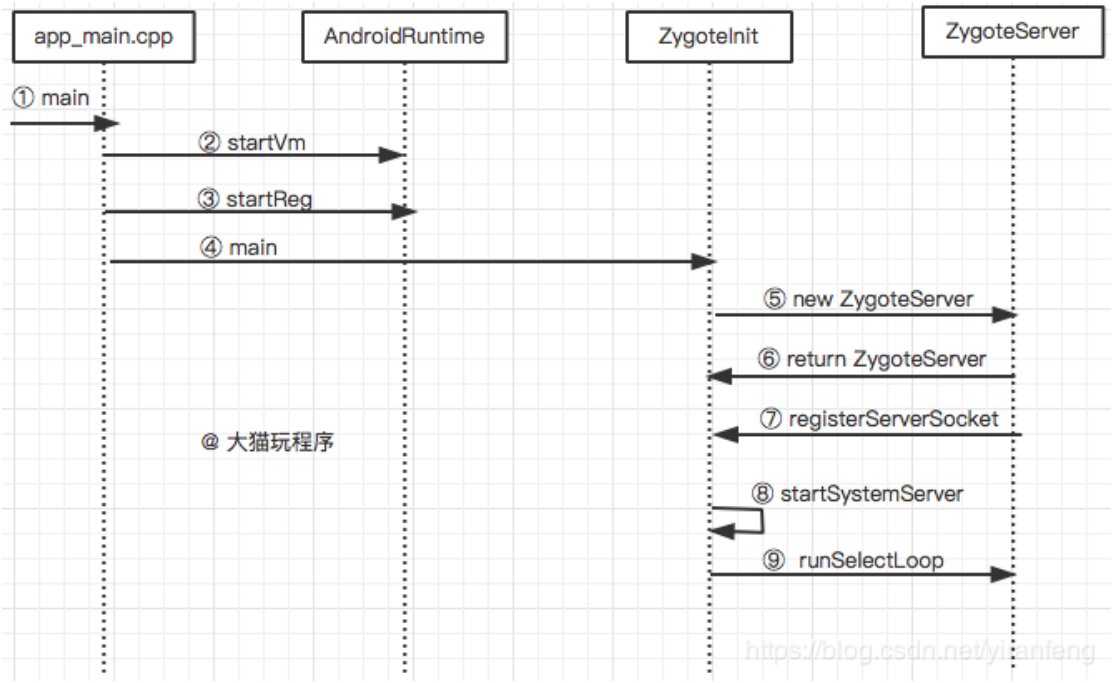
4. /system/netd/server/netd.rc 对应进程： netd

通过上面的文件可知，zygote 进程能够重启的时机：

1. inputflinger 进程被杀 （onrestart）
2. servicemanager 进程被杀 （onrestart）
3. surfaceflinger 进程被杀 （onrestart）
4. netd 进程被杀 （onrestart）
5. zygote 进程被杀 （oneshot=false）
6. system_server 进程被杀（waitpid）

2.4.3.2.3 Zygote 启动后做了什么

Zygote 启动时序图：



1. init 进程通过 init.zygote64_32.rc 来调用/system/bin/app_process64 来启动 zygote 进程，入口 app_main.cpp
2. 调用 AndroidRuntime 的 startVM()方法创建虚拟机，再调用 startReg()注册 JNI 函数；
3. 通过 JNI 方式调用 ZygoteInit.main()，第一次进入 Java 世界；

4. registerZygoteSocket()建立 socket 通道，zygote 作为通信的服务端，用于响应客户端请求；
5. preload()预加载通用类、drawable 和 color 资源、openGL 以及共享库以及 WebView，用于提高 app 启动效率；
6. zygote 完毕大部分工作，接下来再通过 startSystemServer()，fork 得力帮手 system_server 进程，也是上层 framework 的运行载体。
7. zygote 任务完成，调用 runSelectLoop()，随时待命，当接收到请求创建新进程请求时立即唤醒并执行相应工作。

2.4.3.2.4 Zygote 启动相关主要函数

C 空间：

```
1. [app_main.cpp] main()
2. [AndroidRuntime.cpp] start()
3. [JniInvocation.cpp] Init()
4. [AndroidRuntime.cpp] startVm()
5. [AndroidRuntime.cpp] startReg()
6. [Threads.cpp] androidSetCreateThreadFunc
7. [AndroidRuntime.cpp] register_jni_procs()    --> gRegJNI.mProc
```

Java 空间：

```
1. [ZygoteInit.java] main()
2. [ZygoteInit.java] preload()
3. [ZygoteServer.java] ZygoteServer
4. [ZygoteInit.java] forkSystemServer
5. [Zygote.java] forkSystemServer
6. [Zygote.java] nativeForkSystemServer
7. [ZygoteServer.java] runSelectLoop
```

2.4.3.3 Zygote 进程启动源码分析

分析 Android Q(10.0) 的 Zygote 启动的源码。

2.4.3.1 Nativate-C 世界的 Zygote 启动要代码调用流程

```
init
    |
init.zygote64_32.rc   //启动两个zygote，一个64位--/system/bin/app_process64;  另一个32位--/system/bin/app_process32
    |
[app_main.cpp]
   main()          //入口
    |
[AndroidRuntime.cpp]
start()
|   |   |       |
|   |   |   JniInvocation.Init(NULL);  //初始化JNI，加载libart.so，
|   |   |                              //调用dlsym，从libart.so中找到JNI_GetDefaultJavaVMInitArgs、
|   |   |                              //JNI_CreateJavaVM、JNI_GetCreatedJavaVMs三个函数地址，赋值给对应成员属性
|   |   |
|   |   startVm()   //启动虚拟机，配置虚拟机参数
|   |   |
|   |   JNI_CreateJavaVM()  //创建VM并返回JavaVM 和 JniEnv，pEnv对应于当前线程
|   |                                                                                @大猫玩程序
|   startReg()  //注册JNI函数
|   |   |
|   |   androidSetCreateThreadFunc()    //虚拟机启动后startReg()过程，会设置线程创建函数指针gCreateThreadFn指向javaCreateThreadEtc.
|   |
|   register_jni_procs()        //注册JNI函数数组  --gRegJNI
|
env->CallStaticVoidMethod()     //通过反射，调用JAVA的com.android.internal.os.ZygoteInit   ----ZygoteInit.java
    |
    [ZygoteInit.java]           //进入JAVA世界
       main()                                                                    https://blog.csdn.net/yiranfeng
```

- [app_main.cpp] main()

```
1.  int main(int argc, char* const argv[])
2.  {
3.      //zygote 传入的参数 argv 为“-Xzygote /system/bin --zygote --start-system-server --socket-name=zygote”
4.      //zygote_secondary 传入的参数 argv 为“-Xzygote /system/bin --zygote --socket-name=zygote_secondary”
5.      ...
6.      while (i < argc) {
7.          const char* arg = argv[i++];
8.          if (strcmp(arg, "--zygote") == 0) {
9.              zygote = true;
10.             //对于 64 位系统 nice_name 为 zygote64; 32 位系统为 zygote
11.             niceName = ZYGOTE_NICE_NAME;
12.         } else if (strcmp(arg, "--start-system-server") == 0) {
13.             //是否需要启动 system server
14.             startSystemServer = true;
15.         } else if (strcmp(arg, "--application") == 0) {
16.             //启动进入独立的程序模式
17.             application = true;
18.         } else if (strncmp(arg, "--nice-name=", 12) == 0) {
19.             //niceName 为当前进程别名，区别 abi 型号
20.             niceName.setTo(arg + 12);
21.         } else if (strncmp(arg, "--", 2) != 0) {
22.             className.setTo(arg);
23.             break;
```

```
        } else {
            --i;
            break;
        }
    }
    ..
    if (!className.isEmpty()) { //className 不为空，说明是 application 启动模式
        ...
    } else {
      //进入 zygote 模式，新建 Dalvik 的缓存目录:/data/dalvik-cache
        maybeCreateDalvikCache();
        if (startSystemServer) { //加入 start-system-server 参数
            args.add(String8("start-system-server"));
        }
        String8 abiFlag("--abi-list=");
        abiFlag.append(prop);
        args.add(abiFlag);  //加入--abi-list=参数
        // In zygote mode, pass all remaining arguments to the zygote
        // main() method.
        for (; i < argc; ++i) {  //将剩下的参数加入 args
            args.add(String8(argv[i]));
        }
    }
    ...
    if (!niceName.isEmpty()) {
//设置一个“好听的昵称” zygote\zygote64，之前的名称是 app_process
        runtime.setArgv0(niceName.string(), true /* setProcName */);
    }
    if (zygote) {    //如果是 zygote 启动模式，则加载 ZygoteInit
        runtime.start("com.android.internal.os.ZygoteInit", args, zygote);
    } else if (className) { //如果是 application 启动模式，则加载 RuntimeInit
        runtime.start("com.android.internal.os.RuntimeInit", args, zygote);
    } else {
        //没有指定类名或 zygote，参数错误
        fprintf(stderr, "Error: no class name or --zygote supplied.\n");
        app_usage();
        LOG_ALWAYS_FATAL("app_process: no class name or --zygote supplied.");
    }
}
```

Zygote 本身是一个 Native 的应用程序，刚开始的进程名称为“app_process”，运行过程中，通过调用 setArgv0 将名字改为 zygote 或者 zygote64(根据操作系统而来)，最后通

过 runtime 的 start()方法来真正的加载虚拟机并进入 JAVA 世界。

- [AndroidRuntime.cpp] start()

```
1.  void AndroidRuntime::start(const char* className, const Vector<String8>& opt
    ions, bool zygote)
2.  {
3.      ALOGD(">>>>>> START %s uid %d <<<<<<\n",
4.              className != NULL ? className : "(unknown)", getuid());
5.      ...
6.      JniInvocation jni_invocation;
7.      jni_invocation.Init(NULL);
8.      JNIEnv* env;
9.       // 虚拟机创建，主要是关于虚拟机参数的设置
10.     if (startVm(&mJavaVM, &env, zygote, primary_zygote) != 0) {
11.         return;
12.     }
13.     onVmCreated(env);    //空函数，没有任何实现
14.
15.     // 注册 JNI 方法
16.     if (startReg(env) < 0) {
17.         ALOGE("Unable to register all android natives\n");
18.         return;
19.     }
20.
21.     jclass stringClass;
22.     jobjectArray strArray;
23.     jstring classNameStr;
24.
25.     //等价 strArray= new String[options.size() + 1];
26.     stringClass = env->FindClass("java/lang/String");
27.     assert(stringClass != NULL);
28.
29.     //等价 strArray[0] = "com.android.internal.os.ZygoteInit"
30.     strArray = env->NewObjectArray(options.size() + 1, stringClass, NULL);
31.     assert(strArray != NULL);
32.     classNameStr = env->NewStringUTF(className);
33.     assert(classNameStr != NULL);
34.     env->SetObjectArrayElement(strArray, 0, classNameStr);
35.
36.     //strArray[1] = "start-system-server";
37.     //strArray[2] = "--abi-list=xxx";
38.     //其中 xxx 为系统响应的 cpu 架构类型，比如 arm64-v8a.
39.     for (size_t i = 0; i < options.size(); ++i) {
```

```
        jstring optionsStr = env->NewStringUTF(options.itemAt(i).string());

        assert(optionsStr != NULL);
        env->SetObjectArrayElement(strArray, i + 1, optionsStr);
    }

    //将"com.android.internal.os.ZygoteInit"转换为"com/android/internal/os/ZygoteInit"
    char* slashClassName = toSlashClassName(className != NULL ? className : "");
    jclass startClass = env->FindClass(slashClassName);
    //找到 Zygoteinit 类
    if (startClass == NULL) {
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
    } else {
        //找到这个类后就继续找成员函数 main 方法的 Mehtod ID
        jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
        if (startMeth == NULL) {
            ALOGE("JavaVM unable to find main() in '%s'\n", className);
        } else {
            // 通过反射调用 ZygoteInit.main()方法
            env->CallStaticVoidMethod(startClass, startMeth, strArray);
        }
    }
    //释放相应对象的内存空间
    free(slashClassName);
    ALOGD("Shutting down VM\n");
    if (mJavaVM->DetachCurrentThread() != JNI_OK)
        ALOGW("Warning: unable to detach main thread\n");
    if (mJavaVM->DestroyJavaVM() != 0)
        ALOGW("Warning: VM did not shut down cleanly\n");
}
```

start()函数主要做了三件事情，一调用 startVm 开启虚拟机，二调用 startReg 注册 JNI 方法，三就是使用 JNI 把 Zygote 进程启动起来。

相关 log：

01-10 11:20:31.369 722 722 D AndroidRuntime: >>>>>> START com.android.internal.os.ZygoteInit uid 0 <<<<<<

01-10 11:20:31.429 722 722 I AndroidRuntime: Using default boot image

01-10 11:20:31.429 722 722 I AndroidRuntime: Leaving lock profiling enabled

- [JniInvocation.cpp] Init()

  Init 函数主要作用是初始化 JNI，具体工作是首先通过 dlopen 加载 libart.so 获得其句柄，然后调用 **dlsym 从 libart.so** 中找到

  JNI_GetDefaultJavaVMInitArgs、**JNI_CreateJavaVM**、

  JNI_GetCreatedJavaVMs 三个函数地址，赋值给对应成员属性，这三个函数会在后续虚拟机创建中调用.

```
1.  bool JniInvocation::Init(const char* library) {
2.    char buffer[PROP_VALUE_MAX];
3.    const int kDlopenFlags = RTLD_NOW | RTLD_NODELETE;
4.    /*
5.     * 1.dlopen 功能是以指定模式打开指定的动态链接库文件，并返回一个句柄
6.     * 2.RTLD_NOW 表示需要在 dlopen 返回前，解析出所有未定义符号，如果解析不出来，在 dlopen 会返回 NULL
7.     * 3.RTLD_NODELETE 表示在 dlclose()期间不卸载库，并且在以后使用 dlopen()重新加载库时不初始化库中的静态变量
8.     */
9.    handle_ = dlopen(library, kDlopenFlags); // 获取 libart.so 的句柄
10.   if (handle_ == NULL) { //获取失败打印错误日志并尝试再次打开 libart.so
11.     if (strcmp(library, kLibraryFallback) == 0) {
12.       // Nothing else to try.
13.       ALOGE("Failed to dlopen %s: %s", library, dlerror());
14.       return false;
15.     }
16.     library = kLibraryFallback;
17.     handle_ = dlopen(library, kDlopenFlags);
18.     if (handle_ == NULL) {
19.       ALOGE("Failed to dlopen %s: %s", library, dlerror());
20.       return false;
21.     }
22.   }
23.   /*
24.    * 1.FindSymbol 函数内部实际调用的是 dlsym
25.    * 2.dlsym 作用是根据 动态链接库 操作句柄(handle)与符号(symbol)，返回符号对应的地址
```

```
    * 3.这里实际就是从 libart.so 中将 JNI_GetDefaultJavaVMInitArgs 等对应的地址存入&JNI_GetDefaultJavaVMInitArgs_中
    */
  if (!FindSymbol(reinterpret_cast<void**>(&JNI_GetDefaultJavaVMInitArgs_),

                  "JNI_GetDefaultJavaVMInitArgs")) {
    return false;
  }
  if (!FindSymbol(reinterpret_cast<void**>(&JNI_CreateJavaVM_),
                  "JNI_CreateJavaVM")) {
    return false;
  }
  if (!FindSymbol(reinterpret_cast<void**>(&JNI_GetCreatedJavaVMs_),
                  "JNI_GetCreatedJavaVMs")) {
    return false;
  }
  return true;
}
```

- [AndroidRuntime.cpp] startVm()

  该函数主要作用就是配置虚拟机的相关参数，再调用之前 JniInvocation 初始化得到的 JNI_CreateJavaVM_来启动虚拟机。

```
int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv, bool zygote, bool primary_zygote)
{
    JavaVMInitArgs initArgs;
    ...
     // JNI 检测功能，用于 native 层调用 jni 函数时进行常规检测，比较弱字符串格式是否符合要求，资源是否正确释放。
     //该功能一般用于早期系统调试或手机 Eng 版，对于 User 版往往不会开启，引用该功能比较消耗系统 CPU 资源，降低系统性能。
    bool checkJni = false;
    property_get("dalvik.vm.checkjni", propBuf, "");
    if (strcmp(propBuf, "true") == 0) {
        checkJni = true;
    } else if (strcmp(propBuf, "false") != 0) {
        /* property is neither true nor false; fall back on kernel parameter */
        property_get("ro.kernel.android.checkjni", propBuf, "");
        if (propBuf[0] == '1') {
            checkJni = true;
        }
```

```
    }
    ALOGV("CheckJNI is %s\n", checkJni ? "ON" : "OFF");
    if (checkJni) {
        /* extended JNI checking */
        addOption("-Xcheck:jni");
    }

    addOption("exit", (void*) runtime_exit); //将参数放入 mOptions 数组中

     //对于不同的软硬件环境，这些参数往往需要调整、优化，从而使系统达到最佳性能
    parseRuntimeOption("dalvik.vm.heapstartsize", heapstartsizeOptsBuf, "-Xms", "4m");
    parseRuntimeOption("dalvik.vm.heapsize", heapsizeOptsBuf, "-Xmx", "16m");
    parseRuntimeOption("dalvik.vm.heapgrowthlimit", heapgrowthlimitOptsBuf, "-XX:HeapGrowthLimit=");
    parseRuntimeOption("dalvik.vm.heapminfree", heapminfreeOptsBuf, "-XX:HeapMinFree=");
    parseRuntimeOption("dalvik.vm.heapmaxfree", heapmaxfreeOptsBuf, "-XX:HeapMaxFree=");

    ...

    //检索生成指纹并将其提供给运行时这样，anr 转储将包含指纹并可以解析。
    std::string fingerprint = GetProperty("ro.build.fingerprint", "");
    if (!fingerprint.empty()) {
        fingerprintBuf = "-Xfingerprint:" + fingerprint;
        addOption(fingerprintBuf.c_str());
    }

    initArgs.version = JNI_VERSION_1_4;
    initArgs.options = mOptions.editArray(); //将 mOptions 赋值给 initArgs
    initArgs.nOptions = mOptions.size();
    initArgs.ignoreUnrecognized = JNI_FALSE;

     //调用之前 JniInvocation 初始化的 JNI_CreateJavaVM_，参考[4.1.3]
    if (JNI_CreateJavaVM(pJavaVM, pEnv, &initArgs) < 0) {
        ALOGE("JNI_CreateJavaVM failed\n");
        return -1;
    }

    return 0;
}
```

- [AndroidRuntime.cpp] startReg()

  startReg 首先是设置了 Android 创建线程的处理函数，然后创建了一个 200 容量的局部引用作用域，用于确保不会出现 OutOfMemoryException，最后就是调用 register_jni_procs 进行 JNI 方法的注册

```
1.  int AndroidRuntime::startReg(JNIEnv* env)
2.  {
3.      ATRACE_NAME("RegisterAndroidNatives");
4.      //设置 Android 创建线程的函数 javaCreateThreadEtc，这个函数内部是通过 Linux 的
    clone 来创建线程的
5.      androidSetCreateThreadFunc((android_create_thread_fn) javaCreateThreadEt
    c);
6.
7.      ALOGV("--- registering native functions ---\n");
8.
9.      //创建一个 200 容量的局部引用作用域,这个局部引用其实就是局部变量
10.     env->PushLocalFrame(200);
11.
12.     //注册 JNI 方法
13.     if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
14.         env->PopLocalFrame(NULL);
15.         return -1;
16.     }
17.     env->PopLocalFrame(NULL); //释放局部引用作用域
18.     return 0;
19. }
```

- [Thread.cpp] androidSetCreateThreadFunc()

  虚拟机启动后 startReg()过程，会设置线程创建函数指针 gCreateThreadFn 指向 javaCreateThreadEtc.

```
1.  void androidSetCreateThreadFunc(android_create_thread_fn func) {
2.      gCreateThreadFn = func;
3.  }
```

- [AndroidRuntime.cpp] register_jni_procs()

它的处理是交给 RegJNIRec 的 mProc,RegJNIRec 是个很简单的结构体，mProc 是个函数指针

```
static int register_jni_procs(const RegJNIRec array[], size_t count, JNIEnv*
 env)
{
    for (size_t i = 0; i < count; i++) {
        if (array[i].mProc(env) < 0) {// /调用 gRegJNI 的 mProc，参考[4.1.8]
            return -1;
        }
    }
    return 0;
}
```

- [AndroidRuntime.cpp] gRegJNI()

```
static const RegJNIRec gRegJNI[] = {
    REG_JNI(register_com_android_internal_os_RuntimeInit),
    REG_JNI(register_com_android_internal_os_ZygoteInit_nativeZygoteInit),
    REG_JNI(register_android_os_SystemClock),
    REG_JNI(register_android_util_EventLog),
    REG_JNI(register_android_util_Log),
    ...
}

    #define REG_JNI(name)      { name, #name }
    struct RegJNIRec {
        int (*mProc)(JNIEnv*);
    };
gRegJNI 中是一堆函数指针，因此循环调用 gRegJNI 的 mProc，即等价于调用其所对应的函数指针。
例如调用： register_com_android_internal_os_RuntimeInit
这是一个 JNI 函数动态注册的标准方法。

int register_com_android_internal_os_RuntimeInit(JNIEnv* env)
{
    const JNINativeMethod methods[] = {
        { "nativeFinishInit", "()V",
            (void*) com_android_internal_os_RuntimeInit_nativeFinishInit },

        { "nativeSetExitWithoutCleanup", "(Z)V",
```

```
24.             (void*) com_android_internal_os_RuntimeInit_nativeSetExitWithout
    Cleanup },
25.     };
26.
27.     //跟 Java 侧的 com/android/internal/os/RuntimeInit.java 的函数
    nativeFinishInit() 进行一一对应
28.     return jniRegisterNativeMethods(env, "com/android/internal/os/RuntimeIni
    t",
29.         methods, NELEM(methods));
30. }
```

2.4.4 Java 世界的 Zygote 启动主要代码调用流程：

通过 JNI 调用 ZygoteInit 的 main 函数后，Zygote 便进入了 Java 框架层，此前没有任何代码进入过 Java 框架层，换句换说 Zygote 开创了 Java 框架层。

```
[ZygoteInit.java]
   main()
|   |   |   |
|   |   |   preload(bootTimingsTraceLog);      //预加载资源
|   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   beginPreload()
|   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   preloadClasses()    //预加载类的列表---/system/etc/preloaded-classes,
|   |   |   |   |   |   |   |   |   |                       //在版本：/frameworks/base/config/preloaded-classes 中，Android10.0中预计有7603左右个类
|   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   preloadResources()  //加载图片、颜色等资源文件
|   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   preloadSharedLibraries()   //加载 android、compiler_rt、jnigraphics等library
|   |   |   |   |   |   |   |
|   |   |   |   |   |   preloadTextResources()  //用于初始化文字资源
|   |   |   |   |   |
|   |   |   |   |   prepareWebViewInZygote()    //用于初始化webview;
|   |   |   |   |
|   |   |   |   endPreload()
|   |   |   |                                                          @大猫玩程序
|   |   |   [ZygoteServer.java]
|   |   |   ZygoteServer(isPrimaryZygote)   //根据传入的参数，决定是否创建socket：zygote_secondary
|   |   |       |   |
|   |   |       |   [Zygote.java]
|   |   |       |   createManagedSocketFromInitSocket("zygote")       //创建LocalServerSocket，名称为zygote
|   |   |       |
|   |   |       createManagedSocketFromInitSocket("zygote_secondary")  //创建LocalServerSocket，名称为zygote_secondary
|   |   |
|   |   forkSystemServer()  //fork出 System server
|   |       |
|   |       [Zygote.java]
|   |       forkSystemServer()
|   |           |
|   |           nativeForkSystemServer()    //通过JNI到C空间进行处理
|   |               |
|   |               [com_android_internal_os_Zygote.cpp]
|   |               com_android_internal_os_Zygote_nativeForkSystemServer
|
[ZygoteServer.java]
runSelectLoop()  //采用管道pipe-多路复用的方式，开启一个while死循环等待ActivityManagerService创建新进程的请求
                 //zygote功成身退，调用runSelectLoop()，随时待命，当接收到请求创建新进程请求时立即唤醒并执行相应工作。
```


2.4.4.1 [ZygoteInit.java]main.cpp

代码路径:frameworks\base\core\java\com\android\internal\os\ZygoteInit.java

main 的主要工作:

1. 调用 preload() 来预加载类和资源
2. 调用 ZygoteServer() 创建两个 Server 端的 Socket--/dev/socket/zygote 和 /dev/socket/zygote_secondary，Socket 用来等待 ActivityManagerService 来请求 Zygote 来创建新的应用程序进程。
3. 调用 forkSystemServer 来启动 SystemServer 进程，这样系统的关键服务也会由 SystemServer 进程启动起来。
4. 最后调用 runSelectLoop 函数来等待客户端请求

下面我们主要来分析这四件事。

```
1.  public static void main(String argv[]) {
2.          // 1.创建 ZygoteServer
3.          ZygoteServer zygoteServer = null;
4.
5.          // 调用 native 函数，确保当前没有其它线程在运行
6.          ZygoteHooks.startZygoteNoThreadCreation();
7.
8.          //设置 pid 为 0，Zygote 进入自己的进程组
9.          Os.setpgid(0, 0);
10.         ......
11.         Runnable caller;
12.         try {
13.             ......
14.             //得到 systrace 的监控 TAG
15.             String bootTimeTag = Process.is64Bit() ? "Zygote64Timing" : "Zyg
    ote32Timing";
16.             TimingsTraceLog bootTimingsTraceLog = new TimingsTraceLog(bootTi
    meTag,
17.                     Trace.TRACE_TAG_DALVIK);
18.             //通过 systradce 来追踪 函数 ZygoteInit， 可以通过 systrace 工具来进行
    分析
19.             //traceBegin 和 traceEnd 要成对出现，而且需要使用同一个 tag
20.             bootTimingsTraceLog.traceBegin("ZygoteInit");
21.
22.             //开启 DDMS(Dalvik Debug Monitor Service)功能
23.             //注册所有已知的 Java VM 的处理块的监听器。线程监听、内存监听、
    native 堆内存监听、debug 模式监听等等
24.             RuntimeInit.enableDdms();
25.
26.             boolean startSystemServer = false;
27.             String zygoteSocketName = "zygote";
28.             String abiList = null;
29.             boolean enableLazyPreload = false;
```

```
30.
31.             //2. 解析 app_main.cpp - start()传入的参数
32.             for (int i = 1; i < argv.length; i++) {
33.                 if ("start-system-server".equals(argv[i])) {
34.                     startSystemServer = true; //启动 zygote 时，才会传入参数：
    start-system-server
35.                 } else if ("--enable-lazy-preload".equals(argv[i])) {
36.                     enableLazyPreload = true; //启动 zygote_secondary 时，才会
    传入参数：enable-lazy-preload
37.                 } else if (argv[i].startsWith(ABI_LIST_ARG)) { //通过属性
    ro.product.cpu.abilist64\ro.product.cpu.abilist32 从 C 空间传来的值
38.                     abiList = argv[i].substring(ABI_LIST_ARG.length());
39.                 } else if (argv[i].startsWith(SOCKET_NAME_ARG)) {
40.                     zygoteSocketName = argv[i].substring(SOCKET_NAME_ARG.len
    gth()); //会有两种值：zygote 和 zygote_secondary
41.                 } else {
42.                     throw new RuntimeException("Unknown command line argumen
    t: " + argv[i]);
43.                 }
44.             }
45.
46.             // 根据传入 socket name 来决定是创建 socket 还是 zygote_secondary
47.             final boolean isPrimaryZygote = zygoteSocketName.equals(Zygote.P
    RIMARY_SOCKET_NAME);
48.
49.             // 在第一次 zygote 启动时，enableLazyPreload 为 false，执行 preload
50.             if (!enableLazyPreload) {
51.                 //systrace 追踪 ZygotePreload
52.                 bootTimingsTraceLog.traceBegin("ZygotePreload");
53.                 EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
54.                         SystemClock.uptimeMillis());
55.                 // 3.加载进程的资源和类,参考[4.2.2]
56.                 preload(bootTimingsTraceLog);
57.                 EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
58.                         SystemClock.uptimeMillis());
59.                 //systrae 结束 ZygotePreload 的追踪
60.                 bootTimingsTraceLog.traceEnd(); // ZygotePreload
61.             } else {
62.                 // 延迟预加载， 变更 Zygote 进程优先级为 NORMAL 级别，第一次 fork
    时才会 preload
63.                 Zygote.resetNicePriority();
64.             }
65.
66.             //结束 ZygoteInit 的 systrace 追踪
```

```
            bootTimingsTraceLog.traceEnd(); // ZygoteInit
            //禁用 systrace 追踪，以便 fork 的进程不会从 zygote 继承过时的跟踪标记
            Trace.setTracingEnabled(false, 0);

            // 4.调用 ZygoteServer 构造函数，创建 socket，会根据传入的参数，
            // 创建两个 socket：/dev/socket/zygote 和 /dev/socket/zygote_secondary
            //参考[4.2.3]
            zygoteServer = new ZygoteServer(isPrimaryZygote);

            if (startSystemServer) {
                //5. fork 出 system server，参考[4.2.4]
                Runnable r = forkSystemServer(abiList, zygoteSocketName, zygoteServer);

                // 启动 SystemServer
                if (r != null) {
                    r.run();
                    return;
                }
            }

            // 6.  zygote 进程进入无限循环，处理请求
            caller = zygoteServer.runSelectLoop(abiList);
        } catch (Throwable ex) {
            Log.e(TAG, "System zygote died with exception", ex);
            throw ex;
        } finally {
            if (zygoteServer != null) {
                zygoteServer.closeServerSocket();
            }
        }

        // 7.在子进程中退出了选择循环。继续执行命令
        if (caller != null) {
             caller.run();
         }
    }
```

日志:

```
01-10 11:20:32.219   722   722 D Zygote  : begin preload
```

```
01-10 11:20:32.219   722   722 I Zygote  : Calling ZygoteHooks.beginPreload()
01-10 11:20:32.249   722   722 I Zygote  : Preloading classes...
01-10 11:20:33.179   722   722 I Zygote  : ...preloaded 7587 classes in 926ms.

01-10 11:20:33.449   722   722 I Zygote  : Preloading resources...
01-10 11:20:33.459   722   722 I Zygote  : ...preloaded 64 resources in 17ms.
01-10 11:20:33.519   722   722 I Zygote  : Preloading shared libraries...
01-10 11:20:33.539   722   722 I Zygote  : Called ZygoteHooks.endPreload()
01-10 11:20:33.539   722   722 I Zygote  : Installed AndroidKeyStoreProvider in 1ms.
01-10 11:20:33.549   722   722 I Zygote  : Warmed up JCA providers in 11ms.
01-10 11:20:33.549   722   722 D Zygote  : end preload
01-10 11:20:33.649   722   722 D Zygote  : Forked child process 1607
01-10 11:20:33.649   722   722 I Zygote  : System server process 1607 has been created
01-10 11:20:33.649   722   722 I Zygote  : Accepting command socket connections

10-15 06:11:07.749   722   722 D Zygote  : Forked child process 2982
10-15 06:11:07.789   722   722 D Zygote  : Forked child process 3004
```

2.4.4.2 [ZygoteInit.java] preload()

```
static void preload(TimingsTraceLog bootTimingsTraceLog) {
        Log.d(TAG, "begin preload");

        beginPreload(); // Pin ICU Data, 获取字符集转换资源等

        //预加载类的列表---/system/etc/preloaded-classes， 在版本：/frameworks/base/config/preloaded-classes 中，Android10.0 中预计有 7603 左右个类
        //从下面的 log 看，成功加载了 7587 个类
        preloadClasses();

        preloadResources(); //加载图片、颜色等资源文件，部分定义在 /frameworks/base/core/res/res/values/arrays.xml 中
        ......
```

```
        preloadSharedLibraries();   // 加载 android、compiler_rt、jnigraphics 等 library
        preloadTextResources();    //用于初始化文字资源

        WebViewFactory.prepareWebViewInZygote();    //用于初始化 webview;
        endPreload();   //预加载完成，可以查看下面的 log
        warmUpJcaProviders();
        Log.d(TAG, "end preload");

        sPreloadComplete = true;
    }
```

什么是预加载：

预加载是指在 zygote 进程启动的时候就加载，这样系统只在 zygote 执行一次加载操作，所有 APP 用到该资源不需要再重新加载，减少资源加载时间，加快了应用启动速度，一般情况下，系统中 App 共享的资源会被列为预加载资源。

zygote fork 子进程时，根据 fork 的 copy-on-write 机制可知，有些类如果不做改变，甚至都不用复制，子进程可以和父进程共享这部分数据，从而省去不少内存的占用。

预加载的原理：

zygote 进程启动后将资源读取出来，保存到 Resources 一个全局静态变量中，下次读取系统资源的时候优先从静态变量中查找。

frameworks/base/config/preloaded-classes：

参考：

preloaded-classes

```
#·This·file·has·been·derived·for·mainline·phone·(and·tablet)·usage.
#
android.R$styleable
android.accessibilityservice.AccessibilityServiceInfo$1
android.accessibilityservice.AccessibilityServiceInfo
android.accounts.Account$1
android.accounts.Account
android.accounts.AccountManager$10
android.accounts.AccountManager$11
android.accounts.AccountManager$18
android.accounts.AccountManager$1
android.accounts.AccountManager$20
android.accounts.AccountManager$2
android.accounts.AccountManager$AmsTask$1
android.accounts.AccountManager$AmsTask$Response
android.accounts.AccountManager$AmsTask
android.accounts.AccountManager$BaseFutureTask$1
android.accounts.AccountManager$BaseFutureTask$Response
android.accounts.AccountManager$BaseFutureTask
android.accounts.AccountManager$Future2Task$1
android.accounts.AccountManager$Future2Task
android.accounts.AccountManager
```


相关日志：

```
01-10 11:20:32.219 722 722 D Zygote : begin preload
01-10 11:20:32.219 722 722 I Zygote : Calling ZygoteHooks.beginPreload()
01-10 11:20:32.249 722 722 I Zygote : Preloading classes...
01-10 11:20:33.179 722 722 I Zygote : ...preloaded 7587 classes in 926ms.
01-10 11:20:33.539 722 722 I Zygote : Called ZygoteHooks.endPreload()
01-10 11:20:33.549 722 722 D Zygote : end preload
```

2.4.4.3 [ZygoteServer.java] ZygoteServer()

path: frameworks\base\core\java\com\android\internal\os\ZygoteServer.java

作用：ZygoteServer 构造函数初始化时，根据传入的参数，利用 LocalServerSocket 创建了 4 个本地服务端的 socket，用来建立连接。

分别是：zygote、usap_pool_primary、zygote_secondary、usap_pool_secondary

```
private LocalServerSocket mZygoteSocket;
    private LocalServerSocket mUsapPoolSocket;

```

```
4.      //创建 zygote 的 socket
5.      ZygoteServer(boolean isPrimaryZygote) {
6.          mUsapPoolEventFD = Zygote.getUsapPoolEventFD();
7.
8.          if (isPrimaryZygote) {
9.              //创建 socket，并获取 socket 对象，socketname：  zygote
10.             mZygoteSocket = Zygote.createManagedSocketFromInitSocket(Zygote.
    PRIMARY_SOCKET_NAME);
11.             //创建 socket，并获取 socket 对象，socketname：usap_pool_primary
12.             mUsapPoolSocket =
13.                     Zygote.createManagedSocketFromInitSocket(
14.                             Zygote.USAP_POOL_PRIMARY_SOCKET_NAME);
15.         } else {
16.             //创建 socket，并获取 socket 对象，socketname：  zygote_secondary
17.             mZygoteSocket = Zygote.createManagedSocketFromInitSocket(Zygote.
    SECONDARY_SOCKET_NAME);
18.             //创建 socket，并获取 socket 对象，
    socketname：  usap_pool_secondary
19.             mUsapPoolSocket =
20.                     Zygote.createManagedSocketFromInitSocket(
21.                             Zygote.USAP_POOL_SECONDARY_SOCKET_NAME);
22.         }
23.         fetchUsapPoolPolicyProps();
24.         mUsapPoolSupported = true;
25.     }
26.
27.     static LocalServerSocket createManagedSocketFromInitSocket(String socket
    Name) {
28.         int fileDesc;
29.         // ANDROID_SOCKET_PREFIX 为"ANDROID_SOCKET_"
30.         //加入传入参数为 zygote，则 fullSocketName：ANDROID_SOCKET_zygote
31.         final String fullSocketName = ANDROID_SOCKET_PREFIX + socketName;
32.
33.         try {
34.             //init.zygote64_32.rc 启动时，指定了 4 个 socket：
35.             //分别是 zygote、usap_pool_primary、zygote_secondary、
    usap_pool_secondary
36.             // 在进程被创建时，就会创建对应的文件描述符，并加入到环境变量中
37.             // 这里取出对应的环境变量
38.             String env = System.getenv(fullSocketName);
39.             fileDesc = Integer.parseInt(env);
40.         } catch (RuntimeException ex) {
41.             throw new RuntimeException("Socket unset or invalid: " + fullSoc
    ketName, ex);
```

```
        }

        try {
            FileDescriptor fd = new FileDescriptor();
            fd.setInt$(fileDesc);   // 获取 zygote socket 的文件描述符
            return new LocalServerSocket(fd);   // 创建 Socket 的本地服务端
        } catch (IOException ex) {
            throw new RuntimeException(
                "Error building socket from file descriptor: " + fileDesc, e
x);
        }
    }

    path: \frameworks\base\core\java\android\net\LocalServerSocket.java
    public LocalServerSocket(FileDescriptor fd) throws IOException
    {
        impl = new LocalSocketImpl(fd);
        impl.listen(LISTEN_BACKLOG);
        localAddress = impl.getSockAddress();
    }
```

2.4.4.4 [ZygoteInit.java] forkSystemServer()

```
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
      );
      ......
      //参数准备
```

```
21.       /* Hardcoded command line to start the system server */
22.       String args[] = {
23.               "--setuid=1000",
24.               "--setgid=1000",
25.               "--
    setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010
26. ,1018,1021,1023,"
27.                       + "1024,1032,1065,3001,3002,3003,3006,3007,3009,3010",

28.               "--capabilities=" + capabilities + "," + capabilities,
29.               "--nice-name=system_server",
30.               "--runtime-args",
31.               "--target-sdk-version=" + VMRuntime.
32. SDK_VERSION_CUR_DEVELOPMENT,
33.               "com.android.server.SystemServer",
34.       };
35.       ZygoteArguments parsedArgs = null;
36.
37.       int pid;
38.
39.       try {
40.           //将上面准备的参数，按照 ZygoteArguments 的风格进行封装
41.           parsedArgs = new ZygoteArguments(args);
42.           Zygote.applyDebuggerSystemProperty(parsedArgs);
43.           Zygote.applyInvokeWithSystemProperty(parsedArgs);
44.
45.           boolean profileSystemServer = SystemProperties.getBoolean(
46.                   "dalvik.vm.profilesystemserver", false);
47.           if (profileSystemServer) {
48.               parsedArgs.mRuntimeFlags |= Zygote.PROFILE_SYSTEM_SERVER;
49.           }
50.
51.           //通过 fork"分裂"出子进程 system_server
52.           /* Request to fork the system server process */
53.           pid = Zygote.forkSystemServer(
54.                   parsedArgs.mUid, parsedArgs.mGid,
55.                   parsedArgs.mGids,
56.                   parsedArgs.mRuntimeFlags,
57.                   null,
58.                   parsedArgs.mPermittedCapabilities,
59.                   parsedArgs.mEffectiveCapabilities);
60.       } catch (IllegalArgumentException ex) {
61.           throw new RuntimeException(ex);
62.       }
```

```
63.
64.         //进入子进程 system_server
65.         /* For child process */
66.         if (pid == 0) {
67.             // 处理 32_64 和 64_32 的情况
68.             if (hasSecondZygote(abiList)) {
69.                 waitForSecondaryZygote(socketName);
70.             }
71.
72.             // fork 时会 copy socket，system server 需要主动关闭
73.             zygoteServer.closeServerSocket();
74.             // system server 进程处理自己的工作
75.             return handleSystemServerProcess(parsedArgs);
76.         }
77.
78.         return null;
79.     }
```

ZygoteInit。forkSystemServer()会在新 fork 出的子进程中调用 handleSystemServerProcess()，

主要是返回 Runtime.java 的 MethodAndArgsCaller 的方法，然后通过 r.run() 启动 com.android.server.SystemServer 的 main 方法

这个当我们后面的 SystemServer 的章节进行详细讲解。

```
1.  handleSystemServerProcess 代码流程：
2.  handleSystemServerProcess()
3.      |
4.      [ZygoteInit.java]
5.      zygoteInit()
6.          |
7.      [RuntimeInit.java]
8.      applicationInit()
9.          |
10.     findStaticMain()
11.         |
12.     MethodAndArgsCaller()
```

2.4.4.5 [ZygoteServer.java] runSelectLoop()

代码路径: frameworks\base\core\java\com\android\internal\os\ZygoteServer.java

```
1.  Runnable runSelectLoop(String abiList) {
2.          ArrayList<FileDescriptor> socketFDs = new ArrayList<FileDescriptor>(
    );
3.          ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>(
    );
4.
5.          // 首先将 server socket 加入到 fds
6.          socketFDs.add(mZygoteSocket.getFileDescriptor());
7.          peers.add(null);
8.
9.          while (true) {
10.             fetchUsapPoolPolicyPropsWithMinInterval();
11.
12.             int[] usapPipeFDs = null;
13.             StructPollfd[] pollFDs = null;
14.
15.             // 每次循环，都重新创建需要监听的 pollFds
16.             // Allocate enough space for the poll structs, taking into accou
    nt
17.             // the state of the USAP pool for this Zygote (could be a
18.             // regular Zygote, a WebView Zygote, or an AppZygote).
19.             if (mUsapPoolEnabled) {
20.                 usapPipeFDs = Zygote.getUsapPipeFDs();
21.                 pollFDs = new StructPollfd[socketFDs.size() + 1 + usapPipeFD
    s.
22. length];
23.             } else {
24.                 pollFDs = new StructPollfd[socketFDs.size()];
25.             }
26.
27.             /*
28.              * For reasons of correctness the USAP pool pipe and event FDs
29.              * must be processed before the session and server sockets.  Thi
    s
30.              * is to ensure that the USAP pool accounting information is
31.              * accurate when handling other requests like API blacklist
32.              * exemptions.
33.              */
34.
```

```
35.             int pollIndex = 0;
36.             for (FileDescriptor socketFD : socketFDs) {
37.                  // 关注事件到来
38.                 pollFDs[pollIndex] = new StructPollfd();
39.                 pollFDs[pollIndex].fd = socketFD;
40.                 pollFDs[pollIndex].events = (short) POLLIN;
41.                 ++pollIndex;
42.             }
43.
44.             final int usapPoolEventFDIndex = pollIndex;
45.
46.             if (mUsapPoolEnabled) {
47.                 pollFDs[pollIndex] = new StructPollfd();
48.                 pollFDs[pollIndex].fd = mUsapPoolEventFD;
49.                 pollFDs[pollIndex].events = (short) POLLIN;
50.                 ++pollIndex;
51.
52.                 for (int usapPipeFD : usapPipeFDs) {
53.                     FileDescriptor managedFd = new FileDescriptor();
54.                     managedFd.setInt$(usapPipeFD);
55.
56.                     pollFDs[pollIndex] = new StructPollfd();
57.                     pollFDs[pollIndex].fd = managedFd;
58.                     pollFDs[pollIndex].events = (short) POLLIN;
59.                     ++pollIndex;
60.                 }
61.             }
62.
63.             try {
64.                 // 等待事件到来
65.                 Os.poll(pollFDs, -1);
66.             } catch (ErrnoException ex) {
67.                 throw new RuntimeException("poll failed", ex);
68.             }
69.
70.             boolean usapPoolFDRead = false;
71.
72.             //倒序处理，即优先处理已建立链接的信息，后处理新建链接的请求
73.             while (--pollIndex >= 0) {
74.                 if ((pollFDs[pollIndex].revents & POLLIN) == 0) {
75.                     continue;
76.                 }
77.
```

```
78.                 // server socket 最先加入 fds， 因此这里是 server socket 收到数
    据
79.                 if (pollIndex == 0) {
80.                     // 收到新的建立通信的请求，建立通信连接
81.                     ZygoteConnection newPeer = acceptCommandPeer(abiList);
82.                     // 加入到 peers 和 fds，即下一次也开始监听
83.                     peers.add(newPeer);
84.                     socketFDs.add(newPeer.getFileDescriptor());
85.
86.                 } else if (pollIndex < usapPoolEventFDIndex) {
87.                     //说明接收到 AMS 发送过来创建应用程序的请求，调用
88. processOneCommand
89.                     //来创建新的应用程序进程
90.                     // Session socket accepted from the Zygote server socket

91.                     try {
92.                         //有 socket 连接，创建 ZygoteConnection 对象,并添加到
    fds。
93.                         ZygoteConnection connection = peers.get(pollIndex);

94.                         //处理连接，参考[4.2.6]
95.                         final Runnable command = connection.processOneComman
    d(
96. this);
97.
98.                         // TODO (chriswailes): Is this extra check necessary
    ?
99.                         if (mIsForkChild) {
100.                            // We're in the child. We should always have a

101. command to run at this
102.                            // stage if processOneCommand hasn't called "ex
    ec".
103.                            if (command == null) {
104.                                throw new IllegalStateException("command ==

105. null");
106.                            }
107.
108.                            return command;
109.                        } else {
110.                            // We're in the server - we should never have a
    ny
111. commands to run.
```

```
                            if (command != null) {
                                throw new IllegalStateException("command !=
null");
                            }

                            // We don't know whether the remote side of the
socket was closed or
                            // not until we attempt to read from it from
processOneCommand. This
                            // shows up as a regular POLLIN event in our
regular processing loop.
                            if (connection.isClosedByPeer()) {
                                connection.closeSocket();
                                peers.remove(pollIndex);
                                socketFDs.remove(pollIndex);    //处理完则
从
fds 中移除该文件描述符
                            }
                        }
                    } catch (Exception e) {
                        ......
                    } finally {
                        mIsForkChild = false;
                    }
                } else {
                    //处理 USAP pool 的事件
                    // Either the USAP pool event FD or a USAP reporting pi
pe.

                    // If this is the event FD the payload will be the numb
er
of USAPs removed.
                    // If this is a reporting pipe FD the payload will be t
he
PID of the USAP
                    // that was just specialized.
                    long messagePayload = -1;

                    try {
                        byte[] buffer = new byte[Zygote.
USAP_MANAGEMENT_MESSAGE_BYTES];
```

```
                        int readBytes = Os.read(pollFDs[pollIndex].fd, buff
er
, 0, buffer.length);

                        if (readBytes == Zygote.USAP_MANAGEMENT_MESSAGE_BYT
ES
) {
                            DataInputStream inputStream =
                                    new DataInputStream(new
ByteArrayInputStream(buffer));

                            messagePayload = inputStream.readLong();
                        } else {
                            Log.e(TAG, "Incomplete read from USAP managemen
t
FD of size "
                                    + readBytes);
                            continue;
                        }
                    } catch (Exception ex) {
                        if (pollIndex == usapPoolEventFDIndex) {
                            Log.e(TAG, "Failed to read from USAP pool event
 FD
: "
                                    + ex.getMessage());
                        } else {
                            Log.e(TAG, "Failed to read from USAP reporting

pipe: "
                                    + ex.getMessage());
                        }

                        continue;
                    }

                    if (pollIndex > usapPoolEventFDIndex) {
                        Zygote.removeUsapTableEntry((int) messagePayload);

                    }

                    usapPoolFDRead = true;
                }
            }

```

```
187.             // Check to see if the USAP pool needs to be refilled.
188.             if (usapPoolFDRead) {
189.                 int[] sessionSocketRawFDs =
190.                         socketFDs.subList(1, socketFDs.size())
191.                                 .stream()
192.                                 .mapToInt(fd -> fd.getInt$())
193.                                 .toArray();
194.
195.                 final Runnable command = fillUsapPool(sessionSocketRawFDs);
196.
197.                 if (command != null) {
198.                     return command;
199.                 }
200.             }
201.         }
202. }
```

2.4.4.6 [ZygoteConnection.java] processOneCommand()

```
1.  Runnable processOneCommand(ZygoteServer zygoteServer) {
2.      ...
3.      //fork 子进程
4.      pid = Zygote.forkAndSpecialize(parsedArgs.mUid, parsedArgs.mGid,
5.  parsedArgs.mGids,
6.              parsedArgs.mRuntimeFlags, rlimits, parsedArgs.mMountExternal,
7.  parsedArgs.mSeInfo,
8.              parsedArgs.mNiceName, fdsToClose, fdsToIgnore, parsedArgs.
9.  mStartChildZygote,
10.             parsedArgs.mInstructionSet, parsedArgs.mAppDataDir, parsedArgs
11. .mTargetSdkVersion);
12.     if (pid == 0) {
13.         // 子进程执行
14.         zygoteServer.setForkChild();
15.         //进入子进程流程，参考[4.2.7]
16.         return handleChildProc(parsedArgs, descriptors, childPipeFd,
17.                 parsedArgs.mStartChildZygote);
18.     } else {
19.         //父进程执行
20.         // In the parent. A pid < 0 indicates a failure and will be handled
    in
21.         //handleParentProc.
22.         handleParentProc(pid, descriptors, serverPipeFd);
23.         return null;
```

```
    }
    ...
}
```

2.4.4.7 [ZygoteConnection.java] handleChildProc()

```
private Runnable handleChildProc(ZygoteArguments parsedArgs, FileDescriptor[
] descriptors, FileDescriptor pipeFd, boolean isZygote) {
      ...
      if (parsedArgs.mInvokeWith != null) {
          ...
          throw new IllegalStateException("WrapperInit.execApplication
unexpectedly returned");
      } else {
          if (!isZygote) {
              // App 进程将会调用到这里，执行目标类的 main()方法
              return ZygoteInit.zygoteInit(parsedArgs.mTargetSdkVersion,
                      parsedArgs.mRemainingArgs, null /* classLoader */);
          } else {
              return ZygoteInit.childZygoteInit(parsedArgs.mTargetSdkVersion
,
                      parsedArgs.mRemainingArgs, null /* classLoader */);
          }
      }
  }
```

2.4.5 问题分析

2.4.5.1 为什么 SystemServer 和 Zygote 之间通信要采用 Socket

进程间通信我们常用的是 binder，为什么这里要采用 socket 呢。

主要是为了解决 fork 的问题：

UNIX 上 C++程序设计守则 3:多线程程序里不准使用 fork

Binder 通讯是需要多线程操作的，代理对象对 Binder 的调用是在 Binder 线程，需要再通过 Handler 调用主线程来操作。

比如 AMS 与应用进程通讯，AMS 的本地代理 IApplicationThread 通过调用 ScheduleLaunchActivity，调用到的应用进程 ApplicationThread 的 ScheduleLaunchActivity 是在 Binder 线程，

需要再把参数封装为一个 ActivityClientRecord，sendMessage 发送给 H 类（主线程 Handler，ActivityThread 内部类）

主要原因：害怕父进程 binder 线程有锁，然后子进程的主线程一直在等其子线程(从父进程拷贝过来的子进程)的资源，但是其实父进程的子进程并没有被拷贝过来，造成死锁。

所以 fork 不允许存在多线程。而非常巧的是 Binder 通讯偏偏就是多线程，所以干脆父进程（Zgote）这个时候就不使用 binder 线程

2.4.5.2 为什么一个 java 应用一个虚拟机?

1. android 的 VM(vm==Virtual Machine)也是类似 JRE 的东西，当然，各方面都截然不同，不过有一个作用都是一样的，为 app 提供了运行环境
2. android 为每个程序提供一个 vm，可以使每个 app 都运行在独立的运行环境，使稳定性提高。
3. vm 的设计可以有更好的兼容性。android apk 都被编译成字节码(bytecode)，在运行的时候，vm 是先将字节码编译真正可执行的代码，否则不同硬件设备的兼容是很大的麻烦。
4. android（非 ROOT）没有 windows 下键盘钩子之类的东西，每个程序一个虚拟机，各个程序之间也不可以随意访问内存，所以此类木马病毒几乎没有。

2.4.5.3 什么是 Zygote 资源预加载

预加载是指在 zygote 进程启动的时候就加载，这样系统只在 zygote 执行一次加载操作，所有 APP 用到该资源不需要再重新加载，减少资源加载时间，加快了应用启动速度，一般情况下，系统中 App 共享的资源会被列为预加载资源。

zygote fork 子进程时，根据 fork 的 copy-on-write 机制可知，有些类如果不做改变，甚至都不用复制，子进程可以和父进程共享这部分数据，从而省去不少内存的占用。

2.4.5.4 Zygote 为什么要预加载

应用程序都从 Zygote 孵化出来，应用程序都会继承 Zygote 的所有内容。

如果在 Zygote 启动的时候加载这些类和资源，这些孵化的应用程序就继承 Zygote 的类和资源，这样启动引用程序的时候就不需要加载类和资源了，启动的速度就会快很多。

开机的次数不多，但是启动应用程序的次数非常多。

2.4.5.5 Zygote 预加载的原理是什么?

zygote 进程启动后将资源读取出来，保存到 Resources 一个全局静态变量中，下次读取系统资源的时候优先从静态变量中查找。

2.4.6 总结

至此，Zygote 启动流程结束，Zygote 进程共做了如下几件事:

1. 解析 init.zygote64_32.rc，创建 AppRuntime 并调用其 start 方法，启动 Zygote 进程。
2. 创建 JavaVM 并为 JavaVM 注册 JNI.
3. 通过 JNI 调用 ZygoteInit 的 main 函数进入 Zygote 的 Java 框架层。
4. 通过 ZygoteServer 创建服务端 Socket，预加载类和资源，并通过 runSelectLoop 函数等待如 ActivityManagerService 等的请求。
5. 启动 SystemServer 进程。

2.5 Android 系统启动之 SystemServer 进程

解了 Zygote 进程的整个启动流程。Zygote 是所有应用的鼻祖。SystemServer 和其他所有 Dalivik 虚拟机进程都是由 Zygote fork 而来。Zygote fork 的第一个进程就是 SystemServer，其在手机中的进程名为 system_server。

2.5.1 概述

解了 Zygote 进程的整个启动流程。Zygote 是所有应用的鼻祖。SystemServer 和其他所有 Dalivik 虚拟机进程都是由 Zygote fork 而来。Zygote fork 的第一个进程就是 SystemServer，其在手机中的进程名为 system_server。

system_server 进程承载着整个 framework 的核心服务，例如创建 ActivityManagerService、PowerManagerService、DisplayManagerService、PackageManagerService、WindowManagerService、LauncherAppsService 等 80 多个核心系统服务。这些服务以不同的线程方式存在于 system_server 这个进程中。

接下来，就让我们透过 Android 系统源码一起来分析一下 SystemServer 的整个启动过程。

2.5.2 核心源码

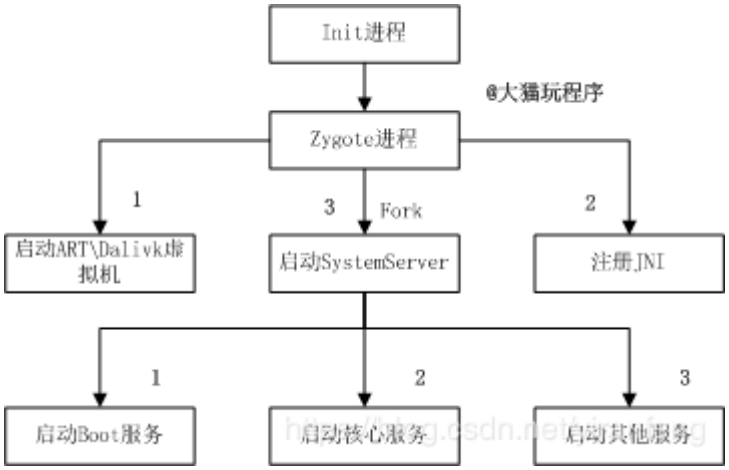
```
1. /frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
2. /frameworks/base/core/java/com/android/internal/os/RuntimeInit.java
3. /frameworks/base/core/java/com/android/internal/os/Zygote.java
4.
5. /frameworks/base/services/java/com/android/server/SystemServer.java
6. /frameworks/base/services/core/java/com/android/serverSystemServiceManager.j
   ava
7. /frameworks/base/services/core/java/com/android/ServiceThread.java
```

```
8.  /frameworks/base/services/core/java/com/android/server/am/ActivityManagerSer
    vice.java
9.
10. /frameworks/base/core/java/android/app/ActivityThread.java
11. /frameworks/base/core/java/android/app/LoadedApk.java
12. /frameworks/base/core/java/android/app/ContextImpl.java
13.
14. /frameworks/base/core/jni/AndroidRuntime.cpp
15. /frameworks/base/core/jni/com_android_internal_os_ZygoteInit.cpp
16. /frameworks/base/cmds/app_process/app_main.cpp
```

2.5.3 架构

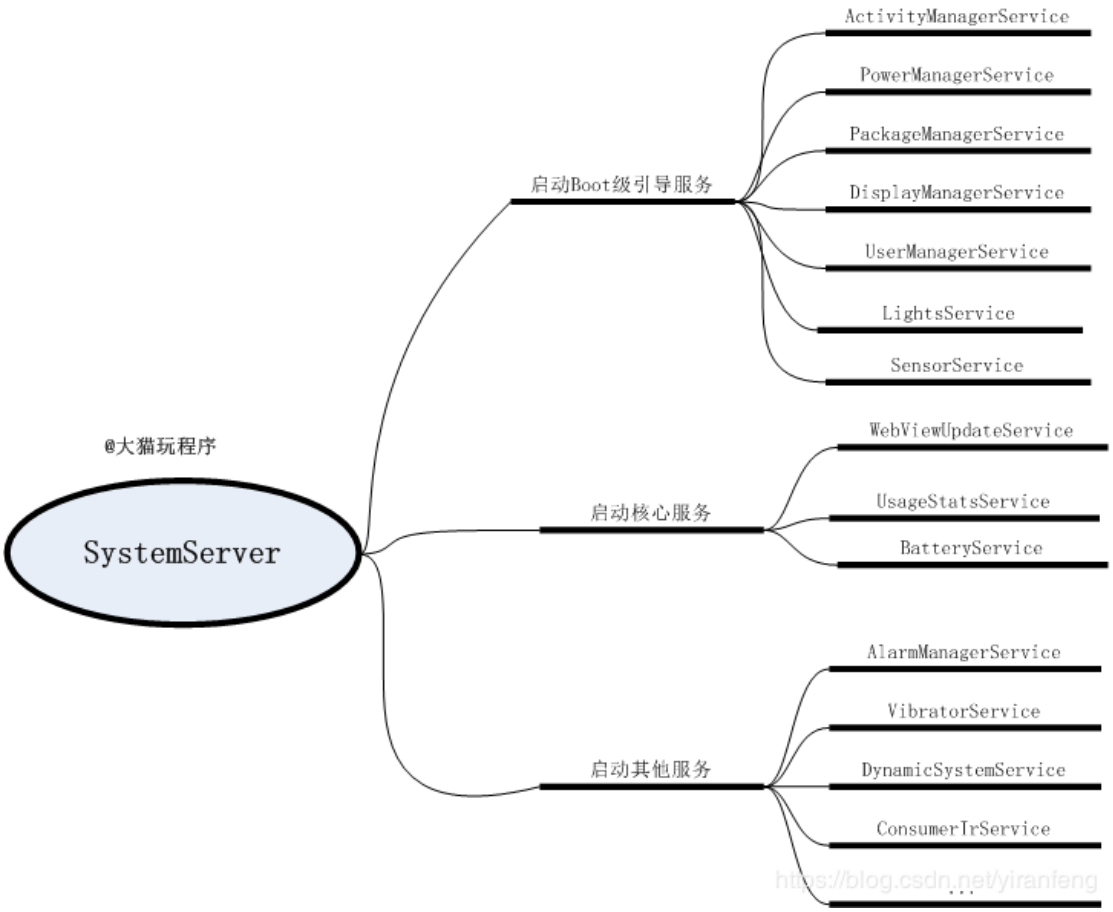
2.5.3.1 架构图

SystemServer 被 Zygote 进程 fork 出来后，用来创建 ActivityManagerService、PowerManagerService、DisplayManagerService、PackageManagerService、WindowManagerService、LauncherAppsService 等 90 多个核心系统服务。



2.5.3.2 服务启动

SystemServer 思维导图



2.5.3.3 源码分析

2.5.3.3.1 SystemServer fork 流程分析

```
SystemServer fork的流程:

[ZygoteInit.java]
  main()
|       |1.
|       Runnable r =forkSystemServer()
|2.     |   |   |(1)
r.run() |   |   [Zygote.java]
        |   |   forkSystemServer() //Fork子进程，这个进程是 system_server
        |   |   |   |(2)
        |   |   |   nativeForkSystemServer() //调用native的方法来fork system_server
        |   |   |   |(2.1)
        |   |   |   [com_android_internal_os_Zygote.cpp]
        |   |   |   |   com_android_internal_os_Zygote_nativeForkSystemServer()
        |   |   |   |   |(2.2)
        |   |   |   |   ForkCommon()
        |   |   |   |       |
        |   |   |   |       pid_t pid = fork();  //调用系统的fork()来孵化子进程
        |   |   |   |                            //fork()执行一次，会返回两次，之后会变成两个进程
        |   |   |   |                            //fork()返回值：0-fork成功，表示子进程在执行；-1：fork失败；大于0：父进程在执行，返回的是实际子进程的pid
        |   |   |   |(2.3)
        |   |   |   SpecializeCommon() //进入子进程system_server
        |   |   |       |   //gCallPostForkChildHooks 即是 Zygote.java的callPostForkChildHooks()
        |   |   |   env->CallStaticVoidMethod(gZygoteClass, gCallPostForkChildHooks,...)
        |   |   |       |
        |   |   |   [Zygote.java]
        |   |   |   callPostForkChildHooks()
        |   |   |       |
        |   |   |   [ZygoteHooks.java]
        |   |   |   postForkChild()
        |   |   |(3)
        |   |   [ZygoteHooks.java]
        |   |   postForkCommon()
        |   |(4)
        |   waitForSecondaryZygote() //如果 hasSecondZygote()成立，即参数中有第二个Zygote的话，需要等待第二个Zygote创建完成
        |(5)
        handleSystemServerProcess() //完成SystemServer的剩余工作
        |   |(5.1)
        |   performSystemServerDexOpt() //执行dex的优化操作
        |(5.2)
        [ZygoteInit.java]
        zygoteInit()
        |   |   |   |
        |   |   |   [RuntimeInit.java]
        |   |   |   redirectLogStreams() //重定向log输出
        |   |   |
        |   |   [RuntimeInit.java]
        |   |   commonInit()    ////通用的一些初始化
        |   |(5.2.1)
        |   [RuntimeInit.java]
        |   nativeZygoteInit()
        |       |
        |       [AndroidRuntime.cpp]    //JNI到Nativate空间
        |       com_android_internal_os_ZygoteInit_nativeZygoteInit()
        |           |
        |           [App_main.cpp]
        |           onZygoteInit()
        |               |
        |               proc->startThreadPool() //启动新的Binder线程
        |(5.2.2)
        [RuntimeInit.java]
        applicationInit()   //应用初始化
            |
            findStaticMain()
            |   |//得到SystemServer.java的main()方法
            |   m = cl.getMethod("main", new Class[] { String[].class })
            |//implements Runnable ,到runnerable的对象
            MethodAndArgsCaller()
            |
            run()  //ZygoteInit.java的main方法会调用该run方法
            |
            mMethod.invoke(null, new Object[] { mArgs });  //通过反射机制，启动 SystemServer.java的main()方法
```

2.5.3.3.2 [ZygoteInit.java] main()

说明：Zygote 进程，通过 fork()函数，最终孵化出 system_server 的进程，通过反射的方法启动

SystemServer.java 的 main()方法

源码：

```
1.  public static void main(String argv[]) {
2.      ZygoteServer zygoteServer = null;
3.      ...
4.      try {
5.          zygoteServer = new ZygoteServer(isPrimaryZygote);
6.          if (startSystemServer) {
7.              //fork system_server
8.              Runnable r = forkSystemServer(abiList, zygoteSocketName, zygoteS
    erver);
9.
10.             // {@code r == null} in the parent (zygote) process, and {@code
    r != null} in the
11.             // child (system_server) process.
12.             if (r != null) {
13.                 r.run(); //启动 SystemServer.java 的 main()
14.                 return; //Android 8.0 之前是通过抛异常的方式来启动，这里是直接
    return 出去，用来清空栈，提高栈帧利用率
15.             }
16.         }
17.         caller = zygoteServer.runSelectLoop(abiList);
18.     } catch (Throwable ex) {
19.         Log.e(TAG, "System zygote died with exception", ex);
20.         throw ex;
21.     } finally {
22.         if (zygoteServer != null) {
23.             zygoteServer.closeServerSocket();
24.         }
25.     }
26.     if (caller != null) {
27.         caller.run();
28.     }
29.     ...
30. }
```

2.5.3.3.3 [ZygoteInit.java] forkSystemServer()

说明：准备参数，用来进行 system_server 的 fork，从参数可知，pid=1000，gid=1000，进程名 nick-name=system_server

当有两个 Zygote 进程时，需要等待第二个 Zygote 创建完成。由于 fork 时会拷贝 socket，因此，在 fork 出 system_server 进程后，

需要关闭 Zygote 原有的 socket

源码：

```
1.  private static Runnable forkSystemServer(String abiList, String socketName,
2.          ZygoteServer zygoteServer) {
3.      ......
4.      //参数准备，uid 和 gid 都是为 1000
5.      String args[] = {
6.              "--setuid=1000",
7.              "--setgid=1000",
8.              "--
    setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,1021,1023,"
9.                      + "1024,1032,1065,3001,3002,3003,3006,3007,3009,3010",
10.             "--capabilities=" + capabilities + "," + capabilities,
11.             "--nice-name=system_server",
12.             "--runtime-args",
13.             "--target-sdk-
    version=" + VMRuntime.SDK_VERSION_CUR_DEVELOPMENT,
14.             "com.android.server.SystemServer",
15.     };
16.     ZygoteArguments parsedArgs = null;
17.     int pid;
18.     try {
19.         //将上面准备的参数，按照 ZygoteArguments 的风格进行封装
20.         parsedArgs = new ZygoteArguments(args);
21.         Zygote.applyDebuggerSystemProperty(parsedArgs);
22.         Zygote.applyInvokeWithSystemProperty(parsedArgs);
23.
24.         //通过 fork"分裂"出子进程 system_server
25.         /* Request to fork the system server process */
26.         pid = Zygote.forkSystemServer(
27.                 parsedArgs.mUid, parsedArgs.mGid,
28.                 parsedArgs.mGids,
29.                 parsedArgs.mRuntimeFlags,
30.                 null,
31.                 parsedArgs.mPermittedCapabilities,
32.                 parsedArgs.mEffectiveCapabilities);
33.     } catch (IllegalArgumentException ex) {
34.         throw new RuntimeException(ex);
35.     }
```

```
36.
37.     //进入子进程 system_server
38.     if (pid == 0) {
39.         // 处理 32_64 和 64_32 的情况
40.         if (hasSecondZygote(abiList)) {
41.             waitForSecondaryZygote(socketName);  //需要等待第二个 Zygote 创建完
    成
42.         }
43.
44.         // fork 时会 copy socket，Zygote 原有的 socket 需要关闭
45.         zygoteServer.closeServerSocket();
46.         // system server 进程处理自己的工作
47.         return handleSystemServerProcess(parsedArgs);
48.     }
49.     return null;
50. }
```

2.5.3.3.4 [Zygote.java] forkSystemServer()

说明：这里的 nativeForkSystemServer()最终是通过 JNI，调用 Nativate C 空间的

com_android_internal_os_Zygote_nativeForkSystemServer()

来 fork system_server

源码：

```
1.  public static int forkSystemServer(int uid, int gid, int[] gids, int runtime
    Flags,
2.          int[][] rlimits, long permittedCapabilities, long effectiveCapabilit
    ies) {
3.      ZygoteHooks.preFork();
4.      // Resets nice priority for zygote process.
5.      resetNicePriority();
6.      //调用 native 的方法来 fork system_server
7.      //最终调用 native 的方
    法:com_android_internal_os_Zygote_nativeForkSystemServer
8.      int pid = nativeForkSystemServer(
9.              uid, gid, gids, runtimeFlags, rlimits,
10.             permittedCapabilities, effectiveCapabilities);
11.     // Enable tracing as soon as we enter the system_server.
12.     if (pid == 0) {
```

```
13.         Trace.setTracingEnabled(true, runtimeFlags);
14.     }
15.     ZygoteHooks.postForkCommon();
16.     return pid;
17. }
```

2.5.3.3.5 [com_android_internal_os_Zygote.cpp]

说明：JNI 注册的映射关系

```
1.  static const JNINativeMethod gMethods[] = {
2.      { "nativeForkSystemServer", "(II[II[[IJJ)I",
3.        (void *) com_android_internal_os_Zygote_nativeForkSystemServer },
4.  }
```

2.5.3.3.6 [com_android_internal_os_Zygote.cpp]

com_android_internal_os_Zygote_nativeForkSystemServer()

说明：通过 SpecializeCommon 进行 fork，pid 返回 0 时，表示当前为 system_server 子进程

当 pid >0 时，是进入父进程，即 Zygote 进程，通过 waitpid 的 WNOHANG 非阻塞方式来监控

system_server 进程挂掉，如果挂掉后重启 Zygote 进程。

现在使用的 Android 系统大部分情况下是 64 位的，会存在两个 Zygote，当 system_server 挂掉后，

只启动 Zygote64 这个父进程

源码：

```
1.  static jint com_android_internal_os_Zygote_nativeForkSystemServer(
```

```
        JNIEnv* env, jclass, uid_t uid, gid_t gid, jintArray gids,
        jint runtime_flags, jobjectArray rlimits, jlong permitted_capabilities,
        jlong effective_capabilities) {


  pid_t pid = ForkCommon(env, true,
                         fds_to_close,
                         fds_to_ignore);
  if (pid == 0) {
      //进入子进程
      SpecializeCommon(env, uid, gid, gids, runtime_flags, rlimits,
                       permitted_capabilities, effective_capabilities,
                       MOUNT_EXTERNAL_DEFAULT, nullptr, nullptr, true,
                       false, nullptr, nullptr);
  } else if (pid > 0) {
      //进入父进程，即 zygote 进程
      ALOGI("System server process %d has been created", pid);

      int status;
      //用 waitpid 函数获取状态发生变化的子进程 pid
      //waitpid 的标记为 WNOHANG，即非阻塞，返回为正值就说明有进程挂掉了
      if (waitpid(pid, &status, WNOHANG) == pid) {
          //当 system_server 进程死亡后，重启 zygote 进程
          ALOGE("System server process %d has died. Restarting Zygote!", pid);
          RuntimeAbort(env, __LINE__, "System server process has died. Restarting Zygote!");
      }
      ...
  }
  return pid;
}
```

2.5.3.3.7 [com_android_internal_os_Zygote.cpp] ForkCommon

说明：从 Zygote 孵化出一个进程的使用程序

源码：

```
static pid_t ForkCommon(JNIEnv* env, bool is_system_server,
                        const std::vector<int>& fds_to_close,
                        const std::vector<int>& fds_to_ignore) {
```

```
    //设置子进程的 signal
    SetSignalHandlers();

    //在 fork 的过程中，临时锁住 SIGCHLD
    BlockSignal(SIGCHLD, fail_fn);

    //fork 子进程,采用 copy on write 方式，这里执行一次，会返回两次
    //pid=0 表示 Zygote  fork SystemServer 这个子进程成功
    //pid > 0 表示 SystemServer 的真正的 PID
    pid_t pid = fork();

    if (pid == 0) {
       //进入子进程
      // The child process.
      PreApplicationInit();

      // 关闭并清除文件描述符
      // Clean up any descriptors which must be closed immediately
      DetachDescriptors(env, fds_to_close, fail_fn);
      ...
    } else {
      ALOGD("Forked child process %d", pid);
    }

    //fork 结束，解锁
    UnblockSignal(SIGCHLD, fail_fn);

    return pid;
}
```

2.5.3.3.8 [Zygcom_android_internal_os_Zygoteote.cpp] SpecializeCommon

说明：system_server 进程的一些调度配置

源码：

```
static void SpecializeCommon(JNIEnv* env, uid_t uid, gid_t gid, jintArray gids,
                             jint runtime_flags, jobjectArray rlimits,
                             jlong permitted_capabilities, jlong effective_capabilities,
```

```
                                  jint mount_external, jstring managed_se_info,
                                  jstring managed_nice_name, bool is_system_server,
                                  bool is_child_zygote, jstring managed_instruction_set,
                                  jstring managed_app_data_dir) {
    ...
    bool use_native_bridge = !is_system_server &&
                             instruction_set.has_value() &&
                             android::NativeBridgeAvailable() &&
                             android::NeedsNativeBridge(instruction_set.value().c_str());

    if (!is_system_server && getuid() == 0) {
      //对于非 system_server 子进程，则创建进程组
      const int rc = createProcessGroup(uid, getpid());
      if (rc == -EROFS) {
        ALOGW("createProcessGroup failed, kernel missing CONFIG_CGROUP_CPUACCT?");
      } else if (rc != 0) {
        ALOGE("createProcessGroup(%d, %d) failed: %s", uid, /* pid= */ 0, strerror(-rc));
      }
    }

    SetGids(env, gids, fail_fn);  //设置设置 group
    SetRLimits(env, rlimits, fail_fn); //设置资源 limit

    if (use_native_bridge) {
      // Due to the logic behind use_native_bridge we know that both app_data_dir
      // and instruction_set contain values.
      android::PreInitializeNativeBridge(app_data_dir.value().c_str(),
                                         instruction_set.value().c_str());
    }

    if (setresgid(gid, gid, gid) == -1) {
      fail_fn(CREATE_ERROR("setresgid(%d) failed: %s", gid, strerror(errno)));

    }
     ...
     //selinux 上下文
    if (selinux_android_setcontext(uid, is_system_server, se_info_ptr, nice_name_ptr) == -1) {
```

```
40.     fail_fn(CREATE_ERROR("selinux_android_setcontext(%d, %d, \"%s\", \"%s\")
     failed",
41.                          uid, is_system_server, se_info_ptr, nice_name_ptr))
    ;
42.   }
43.
44.   //设置线程名为 system_server，方便调试
45.   if (nice_name.has_value()) {
46.     SetThreadName(nice_name.value());
47.   } else if (is_system_server) {
48.     SetThreadName("system_server");
49.   }
50.
51.   // Unset the SIGCHLD handler, but keep ignoring SIGHUP (rationale in SetSi
    gnalHandlers).
52.   //设置子进程的 signal 信号处理函数为默认函数
53.   UnsetChldSignalHandler();
54.
55.   if (is_system_server) {
56.
57.     //对应  Zygote.java 的 callPostForkSystemServerHooks()
58.     env->CallStaticVoidMethod(gZygoteClass, gCallPostForkSystemServerHooks);

59.     if (env->ExceptionCheck()) {
60.       fail_fn("Error calling post fork system server hooks.");
61.     }
62.
63.     //对应 ZygoteInit.java 的 createSystemServerClassLoader()
64.     //预取系统服务器的类加载器。这样做是为了尽早地绑定适当的系统服务器 selinux
    域。
65.     env->CallStaticVoidMethod(gZygoteInitClass, gCreateSystemServerClassLoad
    er);
66.     if (env->ExceptionCheck()) {
67.       // Be robust here. The Java code will attempt to create the classloade
    r
68.       // at a later point (but may not have rights to use AoT artifacts).
69.       env->ExceptionClear();
70.     }
71.     ...
72.   }
73.
74.    //等价于调用 zygote.java 的 callPostForkChildHooks()
75.   env->CallStaticVoidMethod(gZygoteClass, gCallPostForkChildHooks, runtime_f
    lags,
```

```
                                is_system_server, is_child_zygote, managed_instruction_set);

  if (env->ExceptionCheck()) {
    fail_fn("Error calling post fork hooks.");
  }
}
```

2.5.3.3.9 [ZygoteInit.java] handleSystemServerProcess

说明：创建类加载器，并赋予当前线程，其中环境变量 SYSTEMSERVERCLASSPATH，主要是 **service.jar、ethernet-service.jar 和 wifi-service.jar** 这三个 jar 包

```
export SYSTEMSERVERCLASSPATH=/system/framework/services.jar:/system/framework/ethernet-service.jar:/system/framework/wifi-service.jar
```

源码：

```
private static Runnable handleSystemServerProcess(ZygoteArguments parsedArgs) {

    if (parsedArgs.mNiceName != null) {
        Process.setArgV0(parsedArgs.mNiceName); //设置当前进程名为"system_server"
    }

    final String systemServerClasspath = Os.getenv("SYSTEMSERVERCLASSPATH");

    if (systemServerClasspath != null) {
       //执行 dex 优化操作
        if (performSystemServerDexOpt(systemServerClasspath)) {
            sCachedSystemServerClassLoader = null;
        }
        ...
    }

    if (parsedArgs.mInvokeWith != null) {
        String[] args = parsedArgs.mRemainingArgs;
        //如果我们有一个非空系统服务器类路径，我们将不得不复制现有的参数并将类路径附加到它。
        //当我们执行一个新进程时，ART 将正确地处理类路径。
```

```
20.         if (systemServerClasspath != null) {
21.             String[] amendedArgs = new String[args.length + 2];
22.             amendedArgs[0] = "-cp";
23.             amendedArgs[1] = systemServerClasspath;
24.             System.arraycopy(args, 0, amendedArgs, 2, args.length);
25.             args = amendedArgs;
26.         }
27.
28.         //启动应用进程
29.         WrapperInit.execApplication(parsedArgs.mInvokeWith,
30.                 parsedArgs.mNiceName, parsedArgs.mTargetSdkVersion,
31.                 VMRuntime.getCurrentInstructionSet(), null, args);
32.
33.         throw new IllegalStateException("Unexpected return from WrapperInit.
    execApplication");
34.     } else {
35.         // 创建类加载器，并赋予当前线程
36.         createSystemServerClassLoader();
37.         ClassLoader cl = sCachedSystemServerClassLoader;
38.         if (cl != null) {
39.             Thread.currentThread().setContextClassLoader(cl);
40.         }
41.
42.         //system_server 进入此分支
43.         return ZygoteInit.zygoteInit(parsedArgs.mTargetSdkVersion,
44.                 parsedArgs.mRemainingArgs, cl);
45.     }
46. }
```

2.5.3.3.10 [ZygoteInit.java] zygoteInit

说明：基础配置，并进行应用初始化，返回对象

源码：

```
1.  public static final Runnable zygoteInit(int targetSdkVersion, String[] argv,

2.          ClassLoader classLoader) {
3.
4.      Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "ZygoteInit");
5.      RuntimeInit.redirectLogStreams();  //重定向 log 输出
6.
7.      RuntimeInit.commonInit(); //通用的一些初始化
```

```
8.      ZygoteInit.nativeZygoteInit(); // zygote 初始化
9.      // 应用初始化
10.     return RuntimeInit.applicationInit(targetSdkVersion, argv, classLoader);

11. }
```

2.5.3.3.11 [RuntimeInit.java] commonInit

说明：配置 log、时区、http userAgent 等基础信息

源码：

```
1.  protected static final void commonInit() {
2.      LoggingHandler loggingHandler = new LoggingHandler();
3.
4.      // 设置默认的未捕捉异常处理方法
5.      RuntimeHooks.setUncaughtExceptionPreHandler(loggingHandler);
6.      Thread.setDefaultUncaughtExceptionHandler(new KillApplicationHandler(log
    gingHandler));
7.
8.      // 设置时区，通过属性读出中国时区为"Asia/Shanghai"
9.      RuntimeHooks.setTimeZoneIdSupplier(() -> SystemProperties.get("persist.s
    ys.timezone"));
10.
11.     //重置 log 配置
12.     LogManager.getLogManager().reset();
13.     new AndroidConfig();
14.
15.     //设置默认的 HTTP User-agent 格式,用于 HttpURLConnection
16.     String userAgent = getDefaultUserAgent();
17.     System.setProperty("http.agent", userAgent);
18.
19.     /*
20.      * Wire socket tagging to traffic stats.
21.      */
22.     //设置 socket 的 tag，用于网络流量统计
23.     NetworkManagementSocketTagger.install();
24.     ...
25. }
```

2.5.3.3.12 [ZygoteInit.java] nativeZygoteInit

说明：nativeZygoteInit 通过反射，进入

com_android_internal_os_ZygoteInit_nativeZygoteInit

源码：

[AndroidRuntime.cpp]

```
1. int register_com_android_internal_os_ZygoteInit_nativeZygoteInit(JNIEnv* env
   )
2. {
3.     const JNINativeMethod methods[] = {
4.         { "nativeZygoteInit", "()V",
5.             (void*) com_android_internal_os_ZygoteInit_nativeZygoteInit },
6.     };
7.     return jniRegisterNativeMethods(env, "com/android/internal/os/ZygoteInit
   ",
8.         methods, NELEM(methods));
9. }
10.
11. gCurRuntime = this;
12. static void com_android_internal_os_ZygoteInit_nativeZygoteInit(JNIEnv* env,
     jobject clazz)
13. {
14.     //此处的 gCurRuntime 为 AppRuntime，是在 AndroidRuntime.cpp 中定义的
15.     gCurRuntime->onZygoteInit();
16. }
17.
18. [app_main.cpp]
19. virtual void onZygoteInit()
20. {
21.     sp<ProcessState> proc = ProcessState::self();
22.     ALOGV("App process: starting thread pool.\n");
23.     proc->startThreadPool(); //启动新 binder 线程
24. }
```

2.5.3.3.13 [RuntimeInit.java] applicationInit

说明：通过参数解析，得到 args.startClass = com.android.server.SystemServer

源码：

```
protected static Runnable applicationInit(int targetSdkVersion, String[] argv,
        ClassLoader classLoader) {

    //true 代表应用程序退出时不调用 AppRuntime.onExit()，否则会在退出前调用
    nativeSetExitWithoutCleanup(true);

    // We want to be fairly aggressive about heap utilization, to avoid
    // holding on to a lot of memory that isn't needed.

    //设置虚拟机的内存利用率参数值为 0.75
    VMRuntime.getRuntime().setTargetHeapUtilization(0.75f);
    VMRuntime.getRuntime().setTargetSdkVersion(targetSdkVersion);

    final Arguments args = new Arguments(argv);  //解析参数
    ...
    // Remaining arguments are passed to the start class's static main
    //调用 startClass 的 static 方法 main()
    return findStaticMain(args.startClass, args.startArgs, classLoader);
}
```

2.5.3.3.14 [RuntimeInit.java] findStaticMain

说明：拿到 SystemServer 的 main()方法，并返回 MethodAndArgsCaller()对象

源码：

```
protected static Runnable findStaticMain(String className, String[] argv,
        ClassLoader classLoader) {
    Class<?> cl;

    try {
        //拿到 com.android.server.SystemServer 的类对象
        cl = Class.forName(className, true, classLoader);
    } catch (ClassNotFoundException ex) {
        throw new RuntimeException(
                "Missing class when invoking static main " + className,
                ex);
    }

```

```
    Method m;
    try {
        //得到 SystemServer 的 main()方法，
        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);
    }

    //把 MethodAndArgsCaller 的对象返回给 ZygoteInit.main()。这样做好处是能清空栈帧，提高栈帧利用率
    //清除了设置进程所需的所有堆栈帧
    return new MethodAndArgsCaller(m, argv);
}
```

2.5.3.3.15 [RuntimeInit.java] MethodAndArgsCaller

说明：最终在 ZygoteInit.java 的 main()，调用这里的 run()来启动 SystemServer.java 的 main()，真正进入 SystemServer 进程

源码：

```
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 {
          //根据传递过来的参数，可知此处通过反射机制调用的是 SystemServer.main()
    方法
            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);
        }
    }
}
```

2.5.3.4 SystemServer 启动后的流程

```
SystemServer 启动的流程：

[SystemServer.java]
main()
    |
    run()
|   |   |   |   |   |(1)
|   |   |   |   |   performPendingShutdown() //检测上次关机过程是否失败，这个调用可能不会返回
|   |   |   |   |       |
|   |   |   |   |     [ShutdownThread.java]
|   |   |   |   |     rebootOrShutdown() //当属性sys.shutdown.requested的值不为空时，会重启(值为1)或关机(值大于1)
|   |   |   |   |(2)
|   |   |   |   createSystemContext() //初始化系统上下文
|   |   |   |(3)
|   |   |   startBootstrapServices  // 启动系统Boot级服务
|   |   |(4)
|   |   startCoreServices()  // 启动核心服务
|   |(5)
|   startOtherServices()    // 启动其他服务：一些非紧要或者是非需要及时启动的服务
|(6)
Looper.loop()   //死循环执行，system_server进程不会退出
```


2.5.3.4.1 [SystemServer.java] main

说明：main 函数由 Zygote 进程 fork 后运行，作用是 new 一个 SystemServer 对象，再调用该对象的 run()方法

源码：

```
1.  public static void main(String[] args) {
2.      //new 一个 SystemServer 对象，再调用该对象的 run()方法
3.      new SystemServer().run();
4.  }
```

2.5.3.4.2 [SystemServer.java] run

说明：先初始化一些系统变量，加载类库，创建 Context 对象，创建

SystemServiceManager 对象等候再启动服务,启动引导服务、核心服务和其他服务

源码：

```
1.  private void run() {
2.      try {
3.          traceBeginAndSlog("InitBeforeStartServices");
4.
5.          // Record the process start information in sys props.
6.          //从属性中读取 system_server 进程的一些信息
7.          SystemProperties.set(SYSPROP_START_COUNT, String.valueOf(mStartCount
    ));
8.          SystemProperties.set(SYSPROP_START_ELAPSED, String.valueOf(mRuntimeS
    tartElapsedTime));
9.          SystemProperties.set(SYSPROP_START_UPTIME, String.valueOf(mRuntimeSt
    artUptime));
10.
11.         EventLog.writeEvent(EventLogTags.SYSTEM_SERVER_START,
12.                 mStartCount, mRuntimeStartUptime, mRuntimeStartElapsedTime);
13.
14.
15.         //如果一个设备的时钟是在 1970 年之前(0 年之前)，
16.         //那么很多 api 都会因为处理负数而崩溃，尤其是
    java.io.File#setLastModified
17.         //我把把时间设置为 1970
18.         if (System.currentTimeMillis() < EARLIEST_SUPPORTED_TIME) {
19.             Slog.w(TAG, "System clock is before 1970; setting to 1970.");
20.             SystemClock.setCurrentTimeMillis(EARLIEST_SUPPORTED_TIME);
21.         }
22.
```

```
        //如果时区不存在，设置时区为 GMT
        String timezoneProperty = SystemProperties.get("persist.sys.timezone");
        if (timezoneProperty == null || timezoneProperty.isEmpty()) {
            Slog.w(TAG, "Timezone not set; setting to GMT.");
            SystemProperties.set("persist.sys.timezone", "GMT");
        }

        //变更虚拟机的库文件，对于 Android 10.0 默认采用的是 libart.so
        SystemProperties.set("persist.sys.dalvik.vm.lib.2", VMRuntime.getRuntime().vmLibrary());

        // Mmmmmm... more memory!
        //清除 vm 内存增长上限，由于启动过程需要较多的虚拟机内存空间
        VMRuntime.getRuntime().clearGrowthLimit();
        ...
        //系统服务器必须一直运行，所以它需要尽可能高效地使用内存
        //设置内存的可能有效使用率为 0.8
        VMRuntime.getRuntime().setTargetHeapUtilization(0.8f);


        //一些设备依赖于运行时指纹生成，所以在进一步启动之前，请确保我们已经定义了它。
        Build.ensureFingerprintProperty();

        //访问环境变量前，需要明确地指定用户
        //在 system_server 中，任何传入的包都应该被解除，以避免抛出 BadParcelableException。
        BaseBundle.setShouldDefuse(true);

        //在 system_server 中，当打包异常时，信息需要包含堆栈跟踪
        Parcel.setStackTraceParceling(true);

        //确保当前系统进程的 binder 调用，总是运行在前台优先级 (foreground priority)
        BinderInternal.disableBackgroundScheduling(true);

        //设置 system_server 中 binder 线程的最大数量,最大值为 31
        BinderInternal.setMaxThreads(sMaxBinderThreads);

        //准备主线程 lopper，即在当前线程运行
        android.os.Process.setThreadPriority(
                android.os.Process.THREAD_PRIORITY_FOREGROUND);
        android.os.Process.setCanSelfBackground(false);
```

```
62.         Looper.prepareMainLooper();
63.         Looper.getMainLooper().setSlowLogThresholdMs(
64.                 SLOW_DISPATCH_THRESHOLD_MS, SLOW_DELIVERY_THRESHOLD_MS);
65.
66.         //加载 android_servers.so 库，初始化 native service
67.         System.loadLibrary("android_servers");
68.
69.         // Debug builds - allow heap profiling.
70.         //如果是 Debug 版本，允许堆内存分析
71.         if (Build.IS_DEBUGGABLE) {
72.             initZygoteChildHeapProfiling();
73.         }
74.
75.         //检测上次关机过程是否失败，这个调用可能不会返回
76.         performPendingShutdown();
77.
78.         //初始化系统上下文
79.         createSystemContext();
80.
81.         //创建系统服务管理--SystemServiceManager
82.         mSystemServiceManager = new SystemServiceManager(mSystemContext);
83.         mSystemServiceManager.setStartInfo(mRuntimeRestart,
84.                 mRuntimeStartElapsedTime, mRuntimeStartUptime);
85.         //将 mSystemServiceManager 添加到本地服务的成员 sLocalServiceObjects
86.         LocalServices.addService(SystemServiceManager.class, mSystemServiceM
    anager);
87.         // Prepare the thread pool for init tasks that can be parallelized
88.         //为可以并行化的 init 任务准备线程池
89.         SystemServerInitThreadPool.get();
90.     } finally {
91.         traceEnd();  // InitBeforeStartServices
92.     }
93.
94.     // Start services.
95.     //启动服务
96.     try {
97.         traceBeginAndSlog("StartServices");
98.         startBootstrapServices();   // 启动引导服务
99.         startCoreServices();        // 启动核心服务
100.         startOtherServices();       // 启动其他服务
101.         SystemServerInitThreadPool.shutdown(); //停止线程池
102.     } catch (Throwable ex) {
103.         Slog.e("System", "************************************************");
```

```
104.         Slog.e("System", "************ Failure starting system services", e
    x);
105.         throw ex;
106.     } finally {
107.         traceEnd();
108.     }
109.
110.     //为当前的虚拟机初始化 VmPolicy
111.     StrictMode.initVmDefaults(null);
112.     ...
113.     // Loop forever.
114.     //死循环执行
115.     Looper.loop();
116.     throw new RuntimeException("Main thread loop unexpectedly exited");
117. }
```

2.5.3.4.3 [SystemServer.java] performPendingShutdown

说明：检测上次关机过程是否失败，这个调用可能不会返回

源码：

```
1.  private void performPendingShutdown() {
2.      final String shutdownAction = SystemProperties.get(
3.              ShutdownThread.SHUTDOWN_ACTION_PROPERTY, "");
4.      if (shutdownAction != null && shutdownAction.length() > 0) {
5.          boolean reboot = (shutdownAction.charAt(0) == '1');
6.
7.          final String reason;
8.          if (shutdownAction.length() > 1) {
9.              reason = shutdownAction.substring(1, shutdownAction.length());
10.         } else {
11.             reason = null;
12.         }
13.
14.         //如果需要重新启动才能应用更新，一定要确保 uncrypt 在需要时正确执行。
15.         //如果'/cache/recovery/block.map'还没有创建，停止重新启动，它肯定会失
    败，
16.         //并有机会捕获一个 bugreport 时，这仍然是可行的。
17.         if (reason != null && reason.startsWith(PowerManager.REBOOT_RECOVERY
    _UPDATE)) {
18.             File packageFile = new File(UNCRYPT_PACKAGE_FILE);
19.             if (packageFile.exists()) {
```

```
20.                 String filename = null;
21.                 try {
22.                     filename = FileUtils.readTextFile(packageFile, 0, null);
23.                 } catch (IOException e) {
24.                     Slog.e(TAG, "Error reading uncrypt package file", e);
25.                 }
26.
27.                 if (filename != null && filename.startsWith("/data")) {
28.                     if (!new File(BLOCK_MAP_FILE).exists()) {
29.                         Slog.e(TAG, "Can't find block map file, uncrypt fail
    ed or " +
30.                                 "unexpected runtime restart?");
31.                         return;
32.                     }
33.                 }
34.             }
35.         }
36.         Runnable runnable = new Runnable() {
37.             @Override
38.             public void run() {
39.                 synchronized (this) {
40.                     //当属性 sys.shutdown.requested 的值为 1 时，会重启
41.                     //当属性的值不为空，且不为 1 时，会关机
42.                     ShutdownThread.rebootOrShutdown(null, reboot, reason);
43.                 }
44.             }
45.         };
46.
47.         // ShutdownThread must run on a looper capable of displaying the UI.
48.         //ShutdownThread 必须在一个能够显示 UI 的 looper 上运行
49.         //即 UI 线程启动 ShutdownThread 的 rebootOrShutdown
50.         Message msg = Message.obtain(UiThread.getHandler(), runnable);
51.         msg.setAsynchronous(true);
52.         UiThread.getHandler().sendMessage(msg);
53.
54.     }
55. }
```

2.5.3.4.4 [SystemServer.java] createSystemContext

说明：初始化系统上下文， 该过程会创建对象有 ActivityThread，Instrumentation，ContextImpl，LoadedApk，Application

源码：

```
1.  private void createSystemContext() {
2.      //创建 system_server 进程的上下文信息
3.      ActivityThread activityThread = ActivityThread.systemMain();
4.      mSystemContext = activityThread.getSystemContext();
5.      //设置主题
6.      mSystemContext.setTheme(DEFAULT_SYSTEM_THEME);
7.
8.      //获取 systemui 上下文信息，并设置主题
9.      final Context systemUiContext = activityThread.getSystemUiContext();
10.     systemUiContext.setTheme(DEFAULT_SYSTEM_THEME);
11. }
```

2.5.3.4.5 [SystemServer.java] startBootstrapServices

说明：用于启动系统 Boot 级服务，有 ActivityManagerService，PowerManagerService, LightsService, DisplayManagerService，PackageManagerService， UserManagerService， sensor 服务.

源码：

```
1.      traceBeginAndSlog("StartWatchdog");
2.      //启动 watchdog
3.      //尽早启动 watchdog，如果在早起启动时发生死锁，我们可以让 system_server
4.      //崩溃，从而进行详细分析
5.      final Watchdog watchdog = Watchdog.getInstance();
6.      watchdog.start();
7.      traceEnd();
8.
9.  ...
10.     //添加 PLATFORM_COMPAT_SERVICE，Platform compat 服务被
    ActivityManagerService、PackageManagerService
```

```
    //以及将来可能出现的其他服务使用。
    traceBeginAndSlog("PlatformCompat");
    ServiceManager.addService(Context.PLATFORM_COMPAT_SERVICE,
            new PlatformCompat(mSystemContext));
    traceEnd();

    //阻塞等待 installd 完成启动，以便有机会创建具有适当权限的关键目录，如
/data/user。
    //我们需要在初始化其他服务之前完成此任务。
    traceBeginAndSlog("StartInstaller");
    Installer installer = mSystemServiceManager.startService(Installer.class
);
    traceEnd();
...
    //启动服务 ActivityManagerService,并为其设置 mSystemServiceManager 和
installer
    traceBeginAndSlog("StartActivityManager");
    ActivityTaskManagerService atm = mSystemServiceManager.startService(
            ActivityTaskManagerService.Lifecycle.class).getService();
    mActivityManagerService = ActivityManagerService.Lifecycle.startService(
mSystemServiceManager, atm);
    mActivityManagerService.setSystemServiceManager(mSystemServiceManager);

    mActivityManagerService.setInstaller(installer);
    mWindowManagerGlobalLock = atm.getGlobalLock();
    traceEnd();

    //启动服务 PowerManagerService
    //Power manager 需要尽早启动，因为其他服务需要它。
    //本机守护进程可能正在监视它的注册，
    //因此它必须准备好立即处理传入的绑定器调用(包括能够验证这些调用的权限)
。
    traceBeginAndSlog("StartPowerManager");
    mPowerManagerService = mSystemServiceManager.startService(
PowerManagerService.class);
    traceEnd();

...
    //初始化 power management
    traceBeginAndSlog("InitPowerManagement");
    mActivityManagerService.initPowerManagement();
    traceEnd();

    //启动 recovery system，以防需要重新启动
```

```
50.     traceBeginAndSlog("StartRecoverySystemService");
51.     mSystemServiceManager.startService(RecoverySystemService.class);
52.     traceEnd();
53. ...
54.     //启动服务 LightsService
55.     //管理 led 和显示背光，所以我们需要它来打开显示
56.     traceBeginAndSlog("StartLightsService");
57.     mSystemServiceManager.startService(LightsService.class);
58.     traceEnd();
59. ...
60.     //启动服务 DisplayManagerService
61.     //显示管理器需要在包管理器之前提供显示指标
62.     traceBeginAndSlog("StartDisplayManager");
63.     mDisplayManagerService = mSystemServiceManager.startService(DisplayManag
    erService.class);
64.     traceEnd();
65.
66.     // Boot Phases: Phase100: 在初始化 package manager 之前，需要默认的显示.
67.     traceBeginAndSlog("WaitForDisplay");
68.     mSystemServiceManager.startBootPhase(SystemService.PHASE_WAIT_FOR_DEFAUL
    T_DISPLAY);
69.     traceEnd();
70.
71.     //当设备正在加密时，仅运行核心
72.     String cryptState = VoldProperties.decrypt().orElse("");
73.     if (ENCRYPTING_STATE.equals(cryptState)) {
74.         Slog.w(TAG, "Detected encryption in progress - only parsing core app
    s");
75.         mOnlyCore = true;
76.     } else if (ENCRYPTED_STATE.equals(cryptState)) {
77.         Slog.w(TAG, "Device encrypted - only parsing core apps");
78.         mOnlyCore = true;
79.     }
80. ...
81.     //启动服务 PackageManagerService
82.     traceBeginAndSlog("StartPackageManagerService");
83.     try {
84.         Watchdog.getInstance().pauseWatchingCurrentThread("packagemanagermai
    n");
85.         mPackageManagerService = PackageManagerService.main(mSystemContext,
    installer,
86.                 mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF, mOnlyCore)
    ;
87.     } finally {
```

```
88.         Watchdog.getInstance().resumeWatchingCurrentThread("packagemanagerma
    in");
89.     }
90. ...
91.     //启动服务 UserManagerService，新建目录/data/user/
92.     traceBeginAndSlog("StartUserManagerService");
93.     mSystemServiceManager.startService(UserManagerService.LifeCycle.class);

94.     traceEnd();
95.
96.     // Set up the Application instance for the system process and get  start
    ed.
97.     //为系统进程设置应用程序实例并开始。
98.     //设置 AMS
99.     traceBeginAndSlog("SetSystemProcess");
100.     mActivityManagerService.setSystemProcess();
101.     traceEnd();
102.
103.     //使用一个 ActivityManager 实例完成 watchdog 设置并监听重启，
104. //只有在 ActivityManagerService 作为一个系统进程正确启动后才能这样做
105.     traceBeginAndSlog("InitWatchdog");
106.     watchdog.init(mSystemContext, mActivityManagerService);
107.     traceEnd();
108.
109.      //传感器服务需要访问包管理器服务、app ops 服务和权限服务，
110.     //因此我们在它们之后启动它。
111.     //在单独的线程中启动传感器服务。在使用它之前应该检查完成情况。
112.     mSensorServiceStart = SystemServerInitThreadPool.get().submit(() -> {
113.         TimingsTraceLog traceLog = new TimingsTraceLog(
114.                 SYSTEM_SERVER_TIMING_ASYNC_TAG, Trace.
115. TRACE_TAG_SYSTEM_SERVER);
116.         traceLog.traceBegin(START_SENSOR_SERVICE);
117.         startSensorService(); //启动传感器服务
118.         traceLog.traceEnd();
119.     }, START_SENSOR_SERVICE);
120. }
```

2.5.3.4.6 [SystemServer.java] startCoreServices

说明：启动核心服务 BatteryService，UsageStatsService，WebViewUpdateService、BugreportManagerService、GpuService 等

源码：

```
1.  private void startCoreServices() {
2.      //启动服务 BatteryService，用于统计电池电量，需要 LightService.
3.      mSystemServiceManager.startService(BatteryService.class);
4.
5.      //启动服务 UsageStatsService，用于统计应用使用情况
6.      mSystemServiceManager.startService(UsageStatsService.class);
7.      mActivityManagerService.setUsageStatsManager(
8.              LocalServices.getService(UsageStatsManagerInternal.class));
9.
10.     //启动服务 WebViewUpdateService
11.     //跟踪可更新的 WebView 是否处于就绪状态，并监视更新安装
12.     if (mPackageManager.hasSystemFeature(PackageManager.FEATURE_WEBVIEW)) {
13.         mWebViewUpdateService = mSystemServiceManager.startService(WebViewUp
    dateService.class);
14.     }
15.
16.     //启动 CachedDeviceStateService，跟踪和缓存设备状态
17.     mSystemServiceManager.startService(CachedDeviceStateService.class);
18.
19.     //启动 BinderCallsStatsService，跟踪在绑定器调用中花费的 cpu 时间
20.     traceBeginAndSlog("StartBinderCallsStatsService");
21.     mSystemServiceManager.startService(BinderCallsStatsService.LifeCycle.cla
    ss);
22.     traceEnd();
23.
24.     //启动 LooperStatsService，跟踪处理程序中处理消息所花费的时间。
25.     traceBeginAndSlog("StartLooperStatsService");
26.     mSystemServiceManager.startService(LooperStatsService.Lifecycle.class);
27.     traceEnd();
28.
29.     //启动 RollbackManagerService，管理 apk 回滚
30.     mSystemServiceManager.startService(RollbackManagerService.class);
31.
32.     //启动 BugreportManagerService，捕获 bugreports 的服务
33.     mSystemServiceManager.startService(BugreportManagerService.class);
34.
35.     //启动 GpuService，为 GPU 和 GPU 驱动程序提供服务。
36.     mSystemServiceManager.startService(GpuService.class);
37. }
```

2.5.3.4.7 [SystemServer.java] startOtherServices

说明：启动其他的服务，开始处理一大堆尚未重构和整理的东西,这里的服务太多，大体启动过程类似，就不详细说明

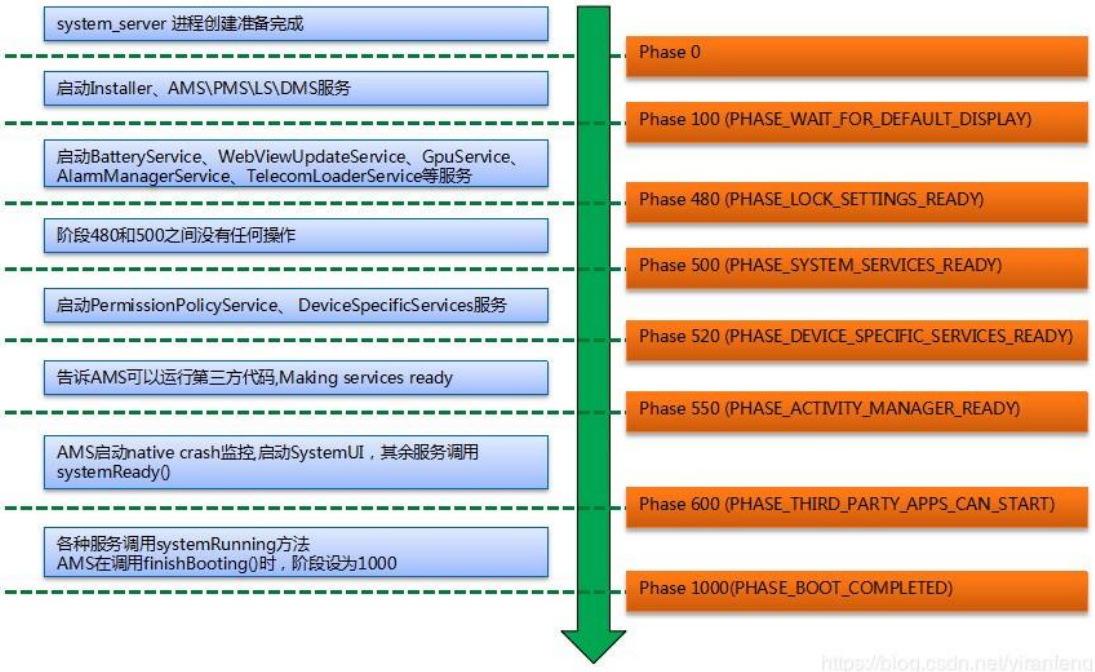
源码：

```
1.  private void startOtherServices() {
2.      ...
3.      //启动 TelecomLoaderService，通话相关核心服务
4.      mSystemServiceManager.startService(TelecomLoaderService.class);
5.
6.      //启动 TelephonyRegistry
7.      telephonyRegistry = new TelephonyRegistry(context);
8.      ServiceManager.addService("telephony.registry", telephonyRegistry);
9.      ...
10.     //启动 AlarmManagerService，时钟管理
11.     mSystemServiceManager.startService(new AlarmManagerService(context));
12.     ...
13.     //启动 InputManagerService
14.     inputManager = new InputManagerService(context);
15.     ServiceManager.addService(Context.INPUT_SERVICE, inputManager,
16.             /* allowIsolated= */ false, DUMP_FLAG_PRIORITY_CRITICAL);
17.     ...
18.     inputManager.setWindowManagerCallbacks(wm.getInputManagerCallback());
19.     inputManager.start();
20.     ...
21.     //Phase480:在接收到此启动阶段后，服务可以获得锁设置数据
22.     mSystemServiceManager.startBootPhase(SystemService.PHASE_LOCK_SETTINGS_R
    EADY);
23.
24.     //Phase500:在接收到这个启动阶段之后，服务可以安全地调用核心系统服务，
25.     //如 PowerManager 或 PackageManager。
26.     mSystemServiceManager.startBootPhase(SystemService.PHASE_SYSTEM_SERVICES
    _READY);
27.
28.     mActivityManagerService.systemReady(() -> {
29.         //Phase550:在接收到此引导阶段后，服务可以广播意图。
30.         mSystemServiceManager.startBootPhase(
31.                 SystemService.PHASE_ACTIVITY_MANAGER_READY);
32.
33.         //Phase600:在接收到这个启动阶段后，服务可以启动/绑定到第三方应用程序。
```

```
34.         //此时，应用程序将能够对服务进行绑定调用。
35.         mSystemServiceManager.startBootPhase(
36.         SystemService.PHASE_THIRD_PARTY_APPS_CAN_START);
37.     }
38. }
```

2.5.4 服务启动分析

服务启动流程如下，从阶段 0 到阶段 1000，一共 8 个阶段。



其中 PHASE_BOOT_COMPLETED=1000，该阶段是发生在 Boot 完成和 home 应用启动完毕。系统服务更倾向于监听该阶段，而不是注册广播 ACTION_BOOT_COMPLETED，从而降低系统延迟。

2.5.4.1 PHASE 0:

说明：startBootstrapServices() 启动引导级服务

主要启动以下 10 个服务：

- Installer
- DeviceIdentifiersPolicyService

- UriGrantsManagerService
- ActivityTaskManagerService
- ActivityManagerService
- PowerManagerService
- ThermalManagerService
- RecoverySystemService
- LightsService
- DisplayManagerService

启动完后，进入 PHASE_WAIT_FOR_DEFAULT_DISPLAY=100， 即 Phase100 阶段

源码：

```
...
    //1.启动 DeviceIdentifiersPolicyService
    mSystemServiceManager.startService(DeviceIdentifiersPolicyService.class);

    //2.启动 UriGrantsManagerService
    mSystemServiceManager.startService(UriGrantsManagerService.Lifecycle.class);

    //3.启动 ActivityTaskManagerService
    atm = mSystemServiceManager.startService(
                ActivityTaskManagerService.Lifecycle.class).getService();

    //4.启动 PowerManagerService
    mPowerManagerService = mSystemServiceManager.startService(PowerManagerService.class);

    //5.启动 ThermalManagerService
    mSystemServiceManager.startService(ThermalManagerService.class);

    //6.启动 RecoverySystemService
    mSystemServiceManager.startService(RecoverySystemService.class);

    //7.启动 LightsService
    mSystemServiceManager.startService(LightsService.class);

```

```
24.     //8.启动 DisplayManagerService
25.     mDisplayManagerService = mSystemServiceManager.startService(DisplayManag
    erService.class);
26.
27.     //执行回调函数 onBootPhase，把 PHASE_WAIT_FOR_DEFAULT_DISPLAY=100， 传入各
    个 service 的 onBootPhase
28.     mSystemServiceManager.startBootPhase(SystemService.PHASE_WAIT_FOR_DEFAUL
    T_DISPLAY);
29.         ...
30. }
```

2.5.4.2 PHASE 100 (阶段 100):

定义：public static final int PHASE_WAIT_FOR_DEFAULT_DISPLAY = 100;

说明： 启动阶段-Boot Phase, 该阶段需要等待 Display 有默认显示

进入阶段 PHASE_WAIT_FOR_DEFAULT_DISPLAY=100 回调服务: onBootPhase(100)

流程：startBootPhase(100) -> onBootPhase(100)

从以下源码可以看到这里遍历了一下服务列表，然后回调到各服务的 onBootPhase() 方法中了。每个服务的 onBootPhase()处理都不相同，这里不详细分析

源码：

```
1.  public void startBootPhase(final int phase) {
2.          ...
3.          mCurrentPhase = phase;
4.          ...
5.          final int serviceLen = mServices.size();
6.          for (int i = 0; i < serviceLen; i++) {
7.              final SystemService service = mServices.get(i);
8.              ...
9.              try {
10.                 service.onBootPhase(mCurrentPhase); // 轮训前面加过的
    service，把 phase 加入服务回调
11.             } catch (Exception ex) {
```

```
            ...
        }
        ...
    }
    ...
}
```

创建以下 80 多个服务：

- BatteryService
- UsageStatsService
- WebViewUpdateService
- CachedDeviceStateService
- BinderCallsStatsService
- LooperStatsService
- RollbackManagerService
- BugreportManagerService
- GpuService
- ....

2.5.4.3 PHASE 480 (阶段 480)：

定义：public static final int PHASE_LOCK_SETTINGS_READY = 480;

说明： 该阶段后， 服务可以获取到锁屏设置的数据了

480 到 500 之间没有任何操作，直接进入 500

2.5.4.4 PHASE 500 (阶段 500)：

定义：public static final int PHASE_SYSTEM_SERVICES_READY = 500;

说明：该阶段后，服务可以安全地调用核心系统服务，比如 PowerManager 或 PackageManager。

启动以下两个服务：

- PermissionPolicyService
- eviceSpecificServices

2.5.4.5 PHASE 520 (阶段 520)：

定义：public static final int PHASE_DEVICE_SPECIFIC_SERVICES_READY = 520;

说明：在接收到这个引导阶段之后，服务可以安全地调用特定于设备的服务。

告诉 AMS 可以运行第三方代码,Making services ready

mActivityManagerService.systemReady()

2.5.4.6 PHASE 550 (阶段 550)：

定义：public static final int PHASE_ACTIVITY_MANAGER_READY = 550;

说明：该阶段后，服务可以接收到广播 Intents

AMS 启动 native crash 监控，启动 SystemUI，其余服务调用 systemReady()

1）AMS 启动 native crash 监控：

```
1. mActivityManagerService.startObservingNativeCrashes();
```

2） 启动 systemUI：

startSystemUi()

3）其余服务调用 systemReady()：

- networkManagementF.systemReady()
- ipSecServiceF.systemReady()；

- networkStatsF.systemReady();
- connectivityF.systemReady();
- networkPolicyF.systemReady(networkPolicyInitReadySignal);

2.5.4.7 PHASE 600 (阶段 600):

定义：public static final int PHASE_THIRD_PARTY_APPS_CAN_START = 600;

说明：该阶段后，服务可以启动/绑定到第三方应用程序。此时，应用程序将能够对服务进行绑定调用。

各种服务调用 systemRunning 方法:

- locationF.systemRunning();
- countryDetectorF.systemRunning();
- networkTimeUpdaterF.systemRunning();
- inputManagerF.systemRunning();
- telephonyRegistryF.systemRunning();
- mediaRouterF.systemRunning();
- mmsServiceF.systemRunning();
- incident.systemRunning();
- touchEventDispatchServiceF.systemRunning();

2.5.4.8 PHASE 1000 (阶段 1000):

定义：public static final int PHASE_BOOT_COMPLETED = 1000;

说明： 该阶段后，服务可以允许用户与设备交互。此阶段在引导完成且主应用程序启动时发生。

系统服务可能更倾向于监听此阶段，而不是为完成的操作注册广播接收器，以减少总体延迟。

在经过一系列流程，再调用 AMS.finishBooting()时，则进入阶段 Phase1000。

到此，系统服务启动阶段完成就绪，system_server 进程启动完成则进入 Looper.loop()状态，随时待命，等待消息队列 MessageQueue 中的消息到来，则马上进入执行状态。

2.5.5 服务分类

system_server 进程启动的服务，从源码角度划分为引导服务、核心服务、其他服务 3 类。

Installer	DeviceIdentifiersPolicyService	UriGrantsManagerService	ActivityTaskManagerService	ActivityManagerService
PowerManagerService	ThermalManagerService	RecoverySystemService	LightsService	DisplayManagerService

核心服务 Core Service(9 个)：

BatteryService	UsageStatsService	WebViewUpdateService	CachedDeviceStateService	BinderCallsStatsService
LooperStatsService	RollbackManagerService	BugreportManagerService	GpuService	

其他服务 Other Service(70 个+)：

KeyChainSystemService	TelecomLoaderService	DropBoxManagerService	AlarmManagerService	BluetoothService
IpConnectivityMetrics	NetworkWatchlistService	PinnerService	MultiClientInputMethodManagerService	InputMethodManagerService
AccessibilityManagerService	StorageManagerService	UiModeManagerService	LockSettingsService	PersistentDataBlockService
TestHarnessModeService	OemLockService	DeviceIdleController	DevicePolicyManagerService	AppPredictionService
ContentSuggestionsService	TextServicesManagerService	TextClassificationManagerService	NetworkScoreService	NotificationManagerService
DeviceStorageMonitorService	TimeDetectorService	SearchManagerService	AudioService	BroadcastRadioService
DockObserver	ThermalObserver	MidiManager	AdbService	AdbService
SerialService	TwilightService	ColorDisplayService	JobSchedulerService	SoundTriggerService
TrustManagerService	BackupManager	AppWidgetService	RoleManagerService	VoiceRecognitionManager
GestureLauncher	SensorNotification	ContextHubSystemService	TimeZoneRulesManagerService	EmergencyAffordanceService
DreamManagerService	PrintManager	CompanionDeviceManager	RestrictionManager	MediaSessionService
HdmiControlService	TvInputManagerService	MediaResourceMonitorService	TvRemoteService	MediaRouterService
FaceService	IrisService	FingerprintService	BiometricService	ShortcutService
LauncherAppsService	CrossProfileAppsService	MediaProjectionManagerService	SliceManagerService	CameraServiceProxy
IoTSystemService	StatsCompanionService	IncidentCompanionService	MmsServiceBroker	AutoFillService
ClipboardService	AppBindingService	PermissionPolicyService	DeviceSpecificServices	

2.5.6 总结

- Zygote 启动后 fork 的第一个进程为 SystemServer,在手机中的进程别名为 "system_server"，主要用来启动系统中的服务
- .Zygote fork 后，进入 SystemServer 的 main()
- SystemServer 在启动过程中，先初始化一些系统变量，加载类库，创建 Context 对象，创建 SystemServiceManager 对象等候再启动服务
- 启动的服务分为 引导服务(Boot Service)、核心服务(Core Service)和其他服务(Other Service)三大类，共 90 多个服务
- SystemServer 在启动服务前，会尝试与 Zygote 建立 Socket 通信，通信成功后才去启动服务
- 启动的服务都单独运行在 SystemServer 的各自线程中，同属于 SystemServer 进程

2.6 Android 10.0 系统服务之 ActivityMnagerService-AMS 启动流程

2.6.1 概述

上一章讲完了 SystemServer 的启动过程，本章来讲解 ActivityManagerService 的启动过程。

ActivityManagerService 简称 AMS，具有管理 Activity 行为、控制 activity 的生命周期、派发消息事件、内存管理等功能。

2.6.2 核心源码

```
/frameworks/base/services/java/com/android/server/SystemServer.java
/frameworks/base/core/java/android/app/ActivityThread.java
/frameworks/base/core/java/android/app/Instrumentation.java
/frameworks/base/core/java/android/app/ContextImpl.java
/frameworks/base/core/java/android/app/LoadedApk.java
/frameworks/base/core/java/android/app/Application.java

/frameworks/base/core/java/com/android/server/LocalServices.java
/frameworks/base/services/core/java/com/android/server/ServiceThread.java

/frameworks/base/services/core/java/com/android/server/wm/ActivityTaskManagerService.java
/frameworks/base/services/core/java/com/android/server/wm/ActivityTaskManagerInternal.java
/frameworks/base/services/core/java/com/android/server/wm/RootActivityContainer.java
/frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java
```

2.6.3 架构

ActivityManagerService 启动共分为以下 4 个阶段：

阶段 1：为 SystemSerer 进程创建 Android 运行环境。AMS 运行与 SystemServer 进程中，它的许多工作依赖于该运行环境

```
1. createSystemContext() -> new ActvityThread()-
   ->attach ->getSystemContext ->createSystemContext
```

阶段 2：启动 AMS，主要进行一些初始化工作

```
1. new ActivityManagerService()
2.
3. start()
```

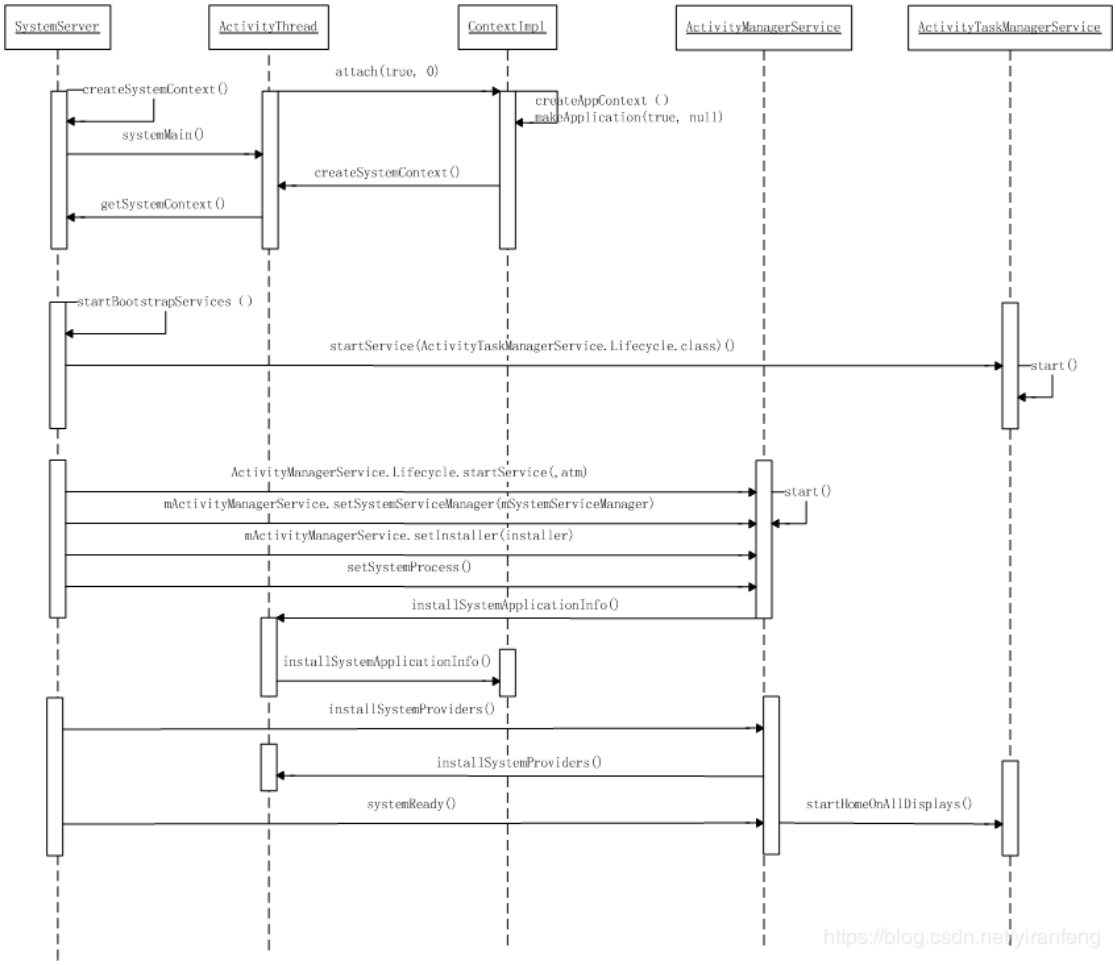
阶段 3：将 SystemServer 进程纳入到 AMS 的进程管理体系中

```
1. setSystemProcess() //将 framework-res.apk 的信息加入到 SystemServer 进程的
   LoadedApk 中；构建 SystemServer 进程的 ProcessRecord，保存到 AMS 中，以便 AMS 进程统
   一管理
2.
3. installSystemProvider() //安装 SystemServer 进程中的 SettingsProvider.apk
```

阶段 4：AMS 启动完成，通知服务或应用完成后续的工作，或直接启动一些进程

AMS.systemReady() //许多服务或应用进程必须等待 AMS 完成启动工作后，才能启动或进行一些后续工作；AMS 就是在 systemReady 中，通知或者启动这些等待的服务和应用进程，例如启动桌面等。

ActivityManagerService 时序图如下：



2.6.4 ActivityManagerService 启动流程-源码分析

```
ActivityManagerService启动流程:

[SystemServer.java]
 run()
|   |  |
|   |  [[SystemServer.java]]
|   |  createSystemContext()
|   |   |   |   |
|   |   |   |   [ActivityThread.java]
|   |   |   |   systemMain()
|   |   |   |       |
|   |   |   |   attach()
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   [Instrumentation.java]
|   |   |   |   |   |   |   new Instrumentation()
|   |   |   |   |   |   [ContextImpl.java]
|   |   |   |   |   |   createAppContext()
|   |   |   |   |   [LoadedApk.java]
|   |   |   |   |  makeApplication()
|   |   |   |   [Application.java]
|   |   |   |   onCreate()
|   |   |   |
|   |   |   |
|   |   |   [ActivityThread.java]
|   |   |   getSystemContext()
|   |   |   |       |
|   |   |   |       [ContextImpl.java]
|   |   |   |       createSystemContext()
|   |   |   |
|   |   |   [ActivityThread.java]
|   |   |   getSystemUiContext()
|   |   |
|   |   [SystemServer.java]
|   |   startBootstrapServices()
|   |   |   |   |
|   |   |   |   [ActivityTaskManagerService.java] //Android10中新引入功能，简称ATM，负责Activity的管理和调度作用
|   |   |   |   Lifecycle
|   |   |   |       |
|   |   |   |       onStart()   //启动服务ActivityTaskManagerService，并返回对象
|   |   |   |
|   |   |   [ActivityManagerService.java]
|   |   |   ActivityManagerService.Lifecycle.startService(
|   |   |                   mSystemServiceManager, atm)  //传入上面得到的atm对象，并返回AMS对象mActivityManagerService
|   |   |       |
|   |   |   Lifecycle
|   |   |       |
|   |   |       startService()  //启动服务 ActivityManagerService
|   |   |           |
|   |   |           [ActivityManagerService.java]
|   |   |           start()
|   |   |               |
|   |   |               removeAllProcessGroups() //移除所有的进程组
|   |   |
|   |   [ActivityManagerService.java]
|   |   setSystemProcess()  //设置AMS
|   |       |
|   |       [ActivityThread.java]
|   |       installSystemApplicationInfo() //安装系统app
|   |           |
|   |           [ContextImpl.java]
|   |           installSystemApplicationInfo()
|   |               |
|   |               [LoadedApk.java]
|   |               installSystemApplicationInfo()
|
[SystemServer.java]
startOtherServices()
|   |
|   [ActivityManagerService.java]
|   installSystemProviders()    //安装系统Provider
|       |
|       [ActivityThread.java]
|       installSystemProviders()
|           |
|           [ActivityThread.java]
|           installContentProviders()
|               |
|               [IActivityManagerSingleton]   //服务: IActivityManager --> Context.ACTIVITY_SERVICE = "activity"
|               publishContentProviders()
|
[ActivityManagerService.java]
systemReady(final Runnable goingCallback, TimingsTraceLog traceLog)
|   |
|   goingCallback.run()
|
[ActivityTaskManagerService.java]
startHomeOnAllDisplays(currentUserId, "systemReady") //启动Luncher
```

2.6.4.1 SystemServer 启动

Zygote 进行启动后，第一个 Fork 出 SystemServer 进程，SystemServer 的启动流程可以参考我之前的文章，这里只分析与 AMS 相关的启动流程：

1. 初始化 SystemContext
2. 创建 SystemServiceManager 对象，用来启动后面的服务
3. 启动系统服务，共分为三种系统服务：系统引导服务（Boot Service）、核心服务（Core Service）和其他服务（Other Service）
4. 在引导服务（Boot Service）中启动 ATM、AMS 服务
5. 在其他服务（Other Service）中完成 AMS 的最后工作 systemReady

源码：

[SystemServer.java]

```
1.  public static void main(String[] args) {
2.      new SystemServer().run();
3.  }
4.
5.  private void run() {
6.      ...
7.      //1.初始化 System Context
8.      createSystemContext(); //参考[4.2]
9.
10.     //2.创建 SystemServiceManager 对象，用来启动后面的服务
11.     mSystemServiceManager = new SystemServiceManager(mSystemContext);
12.     //参考[4.3]
```

```
13.     mSystemServiceManager.setStartInfo(mRuntimeRestart,
14.             mRuntimeStartElapsedTime, mRuntimeStartUptime);
15.     LocalServices.addService(SystemServiceManager.class, mSystemServiceManag
    er);
16.     ...
17.
18.     //3.启动服务
19.     startBootstrapServices();  //启动引导服务，参考[4.1.1]
20.     startCoreServices();       //启动核心服务
21.     startOtherServices();      //启动其他服务，参考[4.1.2]
22.     ...
23. }
```

2.6.4.2 [SystemServer.java] startBootstrapServices()

说明：启动引导服务，在其中启动了 ATM 和 AMS 服务，通过 AMS 安装 Installer、初始化 Power，设置系统进程等。

源码：

```
1.  private void startBootstrapServices() {
2.      ...
3.      //启动 ActivityTaskManagerService 服务，简称 ATM,Android10 新引入功能，用来管
    理 Activity 的启动、调度等功能
4.      //参考[4.4]
5.      atm = mSystemServiceManager.startService(
6.              ActivityTaskManagerService.Lifecycle.class).getService();
7.
8.      //参考[4.5]
9.      //启动服务 ActivityManagerService，简称 AMS
10.     mActivityManagerService = ActivityManagerService.Lifecycle.startService(

11.             mSystemServiceManager, atm);
12.
13.     //安装 Installer
14.     mActivityManagerService.setSystemServiceManager(mSystemServiceManager);

15.     mActivityManagerService.setInstaller(installer);
16.
17.     //初始化 PowerManager
18.     mActivityManagerService.initPowerManagement();
19.
```

```
20.     //设置系统进程，参考 4.6
21.     mActivityManagerService.setSystemProcess();
22.
23. }
```

2.6.4.3 [SystemServer.java] startOtherServices

说明：启动其他服务，AMS 启动后，完成后续桌面启动等操作

源码：

```
1.  private void startOtherServices() {
2.      ...
3.      //安装 SettingsProvider.apk
4.      mActivityManagerService.installSystemProviders();
5.      mActivityManagerService.setWindowManager(wm);
6.
7.      //AMS 启动完成，完成后续的工作，例如启动桌面等
8.      //参考[4.7]
9.      mActivityManagerService.systemReady(() -> {
10.         ...
11.     }, BOOT_TIMINGS_TRACE_LOG);
12.     ...
13. }
```

2.6.4.4 System Context 初始化

说明： 在 SystemServer 的 run 函数中，在启动 AMS 之前，调用了

createSystemContext 函，主要用来是初始化 System Context 和 SystemUi Context，

并设置主题

当 SystemServer 调用 createSystemContext()完毕后，完成以下两个内容：

1. 得到了一个 ActivityThread 对象，它代表当前进程（此时为系统进程）的主线程；
2. 得到了一个 Context 对象，对于 SystemServer 而言，它包含的 Application 运行环境与 framework-res.apk 有关。

源码：

[SystemServer.java]

```
1.  private void createSystemContext() {
2.      ActivityThread activityThread = ActivityThread.systemMain();  //参考
    [4.2.1]
3.
4.      //获取 system context
5.      mSystemContext = activityThread.getSystemContext();
6.      //设置系统主题
7.      mSystemContext.setTheme(DEFAULT_SYSTEM_THEME);
8.
9.      //获取 systemui context
10.     final Context systemUiContext = activityThread.getSystemUiContext();
11.     //设置 systemUI 主题
12.     systemUiContext.setTheme(DEFAULT_SYSTEM_THEME);
13. }
```

2.6.4.5 [ActivityThread.java] systemMain()

说明：systemMain 函数主要作用是： 创建 ActivityThread 对象，然后调用该对象的 attach 函数。

源码：

[ActivityThread.java]

```
1.  public static ActivityThread systemMain() {
2.      // The system process on low-memory devices do not get to use hardware
3.      // accelerated drawing, since this can add too much overhead to the
4.      // process.
5.      if (!ActivityManager.isHighEndGfx()) {
6.          ThreadedRenderer.disable(true);
7.      } else {
8.          ThreadedRenderer.enableForegroundTrimming();
9.      }
10.     //获取 ActivityThread 对象
11.     ActivityThread thread = new ActivityThread(); //参考[4.2.2]
12.     thread.attach(true, 0); 参考[4.2.3]
13.     return thread;
14. }
```

2.6.4.6 ActivityThread 对象创建

说明：ActivityThread 是 Android Framework 中一个非常重要的类，它代表一个应用进程的主线程，其职责就是调度及执行在该线程中运行的四大组件。

注意到此处的 ActivityThread 创建于 SystemServer 进程中。

由于 SystemServer 中也运行着一些系统 APK，例如 framework-res.apk、SettingsProvider.apk 等，因此也可以认为 SystemServer 是一个特殊的应用进程。

AMS 负责管理和调度进程，因此 AMS 需要通过 Binder 机制和应用进程通信。

为此，Android 提供了一个 IApplicationThread 接口，该接口定义了 AMS 和应用进程之间的交互函数。

ActivityThread 的构造函数比较简单，获取 ResourcesManager 的单例对象，比较关键的是它的成员变量：

源码：

```
1.  public final class ActivityThread extends ClientTransactionHandler {
2.
3.      ...
4.      //定义了 AMS 与应用通信的接口，拿到 ApplicationThread 的对象
5.      final ApplicationThread mAppThread = new ApplicationThread();
6.
7.      //拥有自己的 looper，说明 ActivityThread 确实可以代表事件处理线程
8.      final Looper mLooper = Looper.myLooper();
9.
10.     //H 继承 Handler，ActivityThread 中大量事件处理依赖此 Handler
11.     final H mH = new H();
12.
13.     //用于保存该进程的 ActivityRecord
14.     final ArrayMap<IBinder, ActivityClientRecord> mActivities = new ArrayMap
    <>();
```

```
15.
16.     //用于保存进程中的 Service
17.     final ArrayMap<IBinder, Service> mServices = new ArrayMap<>();
18.
19.     ////用于保存进程中的 Application
20.     final ArrayList<Application> mAllApplications
21.         = new ArrayList<Application>();
22.     //构造函数
23.     @UnsupportedAppUsage
24.     ActivityThread() {
25.         mResourcesManager = ResourcesManager.getInstance();
26.     }
27. }
```

2.6.4.7 [ActivityThread.java] attach

说明：对于系统进程而言，ActivityThread 的 attach 函数最重要的工作就是创建了 Instrumentation、Application 和 Context

调用：attach(true, 0)，传入的 system 为 0

源码：

```
1.  private void attach(boolean system, long startSeq) {
2.      mSystemThread = system;
3.      //传入的 system 为 0
4.      if (!system) {
5.          //应用进程的处理流程
6.          ...
7.      } else {
8.          //系统进程的处理流程，该情况只在 SystemServer 中处理
9.          //创建 ActivityThread 中的重要成员：
    Instrumentation、 Application 和 Context
10.         mInstrumentation = new Instrumentation();
11.         mInstrumentation.basicInit(this);
12.
13.         //创建系统的 Context，参考[4.2.4]
14.         ContextImpl context = ContextImpl.createAppContext(
15.                 this, getSystemContext().mPackageInfo);
16.
17.         //调用 LoadedApk 的 makeApplication 函数
```

```
18.         mInitialApplication = context.mPackageInfo.makeApplication(true, nul
    l);
19.         mInitialApplication.onCreate();
20.     }
21.
22. }
```

Instrumentation

Instrumentation 是 Android 中的一个工具类，当该类被启用时，它将优先于应用中其它的类被初始化。

此时，系统先创建它，再通过它创建其它组件。

```
1.  mInstrumentation = new Instrumentation();
2.  mInstrumentation.basicInit(this);
```

Context

Context 是 Android 中的一个抽象类，用于维护应用运行环境的全局信息。

通过 Context 可以访问应用的资源和类，甚至进行系统级的操作，例如启动 Activity、发送广播等。

ActivityThread 的 attach 函数中，通过下面的代码创建出系统应用对应的 Context:

```
1.  ContextImpl context = ContextImpl.createAppContext(this, getSystemContext().
    mPackageInfo);
```

Application:

Android 中 Application 类用于保存应用的全局状态。

在 ActivityThread 中，针对系统进程，通过下面的代码创建了初始的 Application：

```
1.  mInitialApplication = context.mPackageInfo.makeApplication(true, null);
```

```
2.  mInitialApplication.onCreate();
```

2.6.4.8 [ContextImpl.java] getSystemContext()

说明：创建并返回 System Context

源码：

```
1.  public ContextImpl getSystemContext() {
2.      synchronized (this) {
3.          if (mSystemContext == null) {
4.              //调用 ContextImpl 的静态函数 createSystemContext
5.              mSystemContext = ContextImpl.createSystemContext(this);
6.          }
7.          return mSystemContext;
8.      }
9.  }
```

说明：createSystemContext 的内容就是创建一个 LoadedApk，然后初始化一个 ContextImpl 对象。

注意到 createSystemContext 函数中，创建的 LoadApk 对应 packageName 为"android"，也就是 framwork-res.apk。

由于该 APK 仅供 SystemServer 进程使用，因此创建的 Context 被定义为 System Context。

现在该 LoadedApk 还没有得到 framwork-res.apk 实际的信息。

当 PKMS 启动，完成对应的解析后，AMS 将重新设置这个 LoadedApk。

```
1.  static ContextImpl createSystemContext(ActivityThread mainThread) {
2.      //创建 LoadedApk 类，代表一个加载到系统中的 APK
3.      //注意此时的 LoadedApk 只是一个空壳
4.      //PKMS 还没有启动，无法得到有效的 ApplicationInfo
5.      LoadedApk packageInfo = new LoadedApk(mainThread);
6.
7.      //拿到 ContextImpl 的对象
```

```
8.      ContextImpl context = new ContextImpl(null, mainThread, packageInfo, nul
    l, null, null, 0,
9.              null, null);
10.      //初始化资源信息
11.     context.setResources(packageInfo.getResources());
12.     context.mResources.updateConfiguration(context.mResourcesManager.getConf
    iguration(),
13.             context.mResourcesManager.getDisplayMetrics());
14.     return context;
15. }
```

2.6.4.9 SystemServiceManager 创建

说明：通过 SystemServiceManager 的构造方法创建一个 SystemServiceManager 对象，并将该对象添加到 LocalServices 中。

源码：

```
1.  private void run() {
2.      ...
3.      //1.创建 SystemServiceManager 对象，参考 [4.3.1]
4.      mSystemServiceManager = new SystemServiceManager(mSystemContext);
5.      mSystemServiceManager.setStartInfo(mRuntimeRestart,
6.              mRuntimeStartElapsedTime, mRuntimeStartUptime);
7.      //2.启动 SystemServiceManager 服务,参考[4.3.2]
8.      LocalServices.addService(SystemServiceManager.class, mSystemServiceManag
    er);
9.      ...
10. }
```

参阅：《系统启动篇》 文章首发微信公众号：IngresGe