

Unisoc Confidential For hiar

Android 10.0 开机启动流程介绍

文档版本
发布日期

V1.0
2020-10-20

版权所有 © 紫光展锐（上海）科技有限公司。保留一切权利。

本文件所含数据和信息都属于紫光展锐（上海）科技有限公司（以下简称紫光展锐）所有的机密信息，紫光展锐保留所有相关权利。本文件仅为信息参考之目的提供，不包含任何明示或默示的知识产权许可，也不表示有任何明示或默示的保证，包括但不限于满足任何特殊目的、不侵权或性能。当您接受这份文件时，即表示您同意本文件中内容和信息属于紫光展锐机密信息，且同意在未获得紫光展锐书面同意前，不使用或复制本文件的整体或部分，也不向任何其他方披露本文件内容。紫光展锐有权在未经事先通知的情况下，在任何时候对本文件做任何修改。紫光展锐对本文件所含数据和信息不做任何保证，在任何情况下，紫光展锐均不负任何与本文件相关的直接或间接的、任何伤害或损失。

请参照交付物中说明文档对紫光展锐交付物进行使用，任何人对紫光展锐交付物的修改、定制化或违反说明文档的指引对紫光展锐交付物进行使用造成的任何损失由其自行承担。紫光展锐交付物中的性能指标、测试结果和参数等，均为在紫光展锐内部研发和测试系统中获得的，仅供参考，若任何人需要对交付物进行商用或量产，需要结合自身的软硬件测试环境进行全面的测试和调试。

Unisoc Confidential For hiar

紫光展锐（上海）科技有限公司



前言

概述

本文档结合源码对展锐 Android 10.0 平台开机启动流程进行介绍。

读者对象


本文档主要适用于 Android 10.0 平台系统开发人员。

缩略语

缩略语	英文全名	中文解释
AMS	Activity Manager Service	Activity 管理服务
JNI	Java Native Interface	Java Native 接口
ATM	Activity Task Manager Service	Activity 任务管理服务

符号约定

在本文中可能出现下列标志，它所代表的含义如下。

符号	说明
 说明	用于突出重要/关键信息、补充信息和小窍门等。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害。

变更信息

文档版本	发布日期	修改说明
V1.0	2020-10-20	第一次正式发布

关键字

开机启动、zygote。

Unisoc Confidential For hiar

目 录

1 概述.....	1
1.1 Android 架构图	1
1.2 Android 系统启动流程	2
2 Bootloader 及 Kernel 启动	5
2.1 Bootloader 启动	5
2.2 Kernel 启动	6
3 Init 启动流程	9
3.1 Init 进程介绍	9
3.2 Ueventd 子进程	10
3.3 Init 进程启动第一阶段	12
3.3.1 分区挂载	14
3.4 Andorid SELinux	15
3.5 Init 进程启动第二阶段	17
3.5.1 信号处理	17
3.5.2 属性服务	21
3.5.3 init.rc	25
4 zygote 启动流程	31
4.1 zygote.rc 解析	31
4.2 zygote 启动	32
4.2.1 启动入口	33
4.2.2 JNI 初始化	36
4.2.3 虚拟机创建	37
4.2.4 JNI 注册	39
4.3 Java 框架层启动	39
4.4 SystemServer 启动	43
5 SystemServer 启动流程	45
6 AMS 启动流程	50
7 Launcher 启动流程	51
8 常见问题分析	53
8.1 无法开机问题	53
8.2 开机慢问题	53

图目录

图 1-1 Android 框架图	2
图 1-2 Android 启动流程	3
图 2-1 Bootloader 及内核启动	5
图 2-2 Bootloader 启动模式	5
图 2-3 Kernel_init 初始化	7
图 3-1 Init 进程	9
图 3-2 Init 创建、终止以及重启子进程的流程	18
图 3-3 属性访问过程	24
图 3-4 init.rc 组成结构	26
图 3-5 时机类型	26
图 3-6 常用命令	27
图 3-7 常见可选项	28
图 3-8 Action 执行顺序与启动阶段的对应关系图	29
图 3-9 LoadBootScripts 加载 rc 文件顺序	30
图 4-1 zygote 进程	31
图 4-2 zygote 启动流程	32
图 5-1 SystemServer	45
图 5-2 服务分类	49
图 6-1 AMS 启动流程	50
图 7-1 Launcher 启动流程	51

1 概述

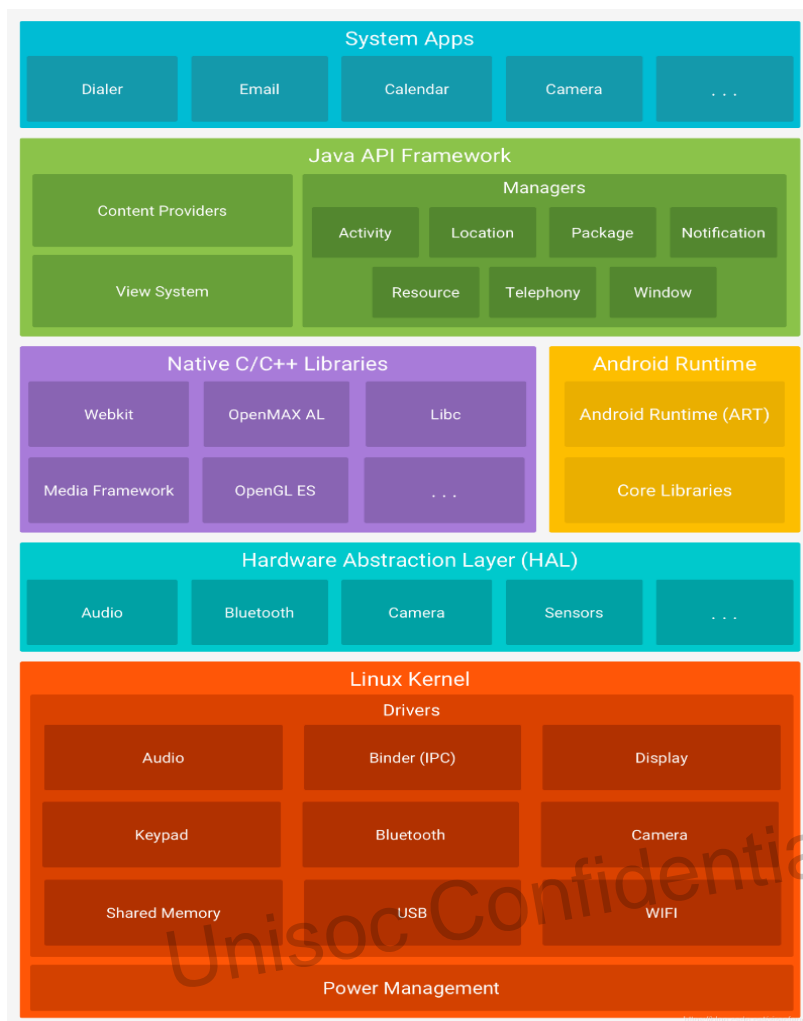
1.1 Android 架构图

图 1-1 是 Google 提供的经典 Android 框架图，该图清晰地展示了 Android 的五层架构，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）
- Linux 内核（Linux Kernel）

Unisoc Confidential For hiar

图1-1 Android 框架图



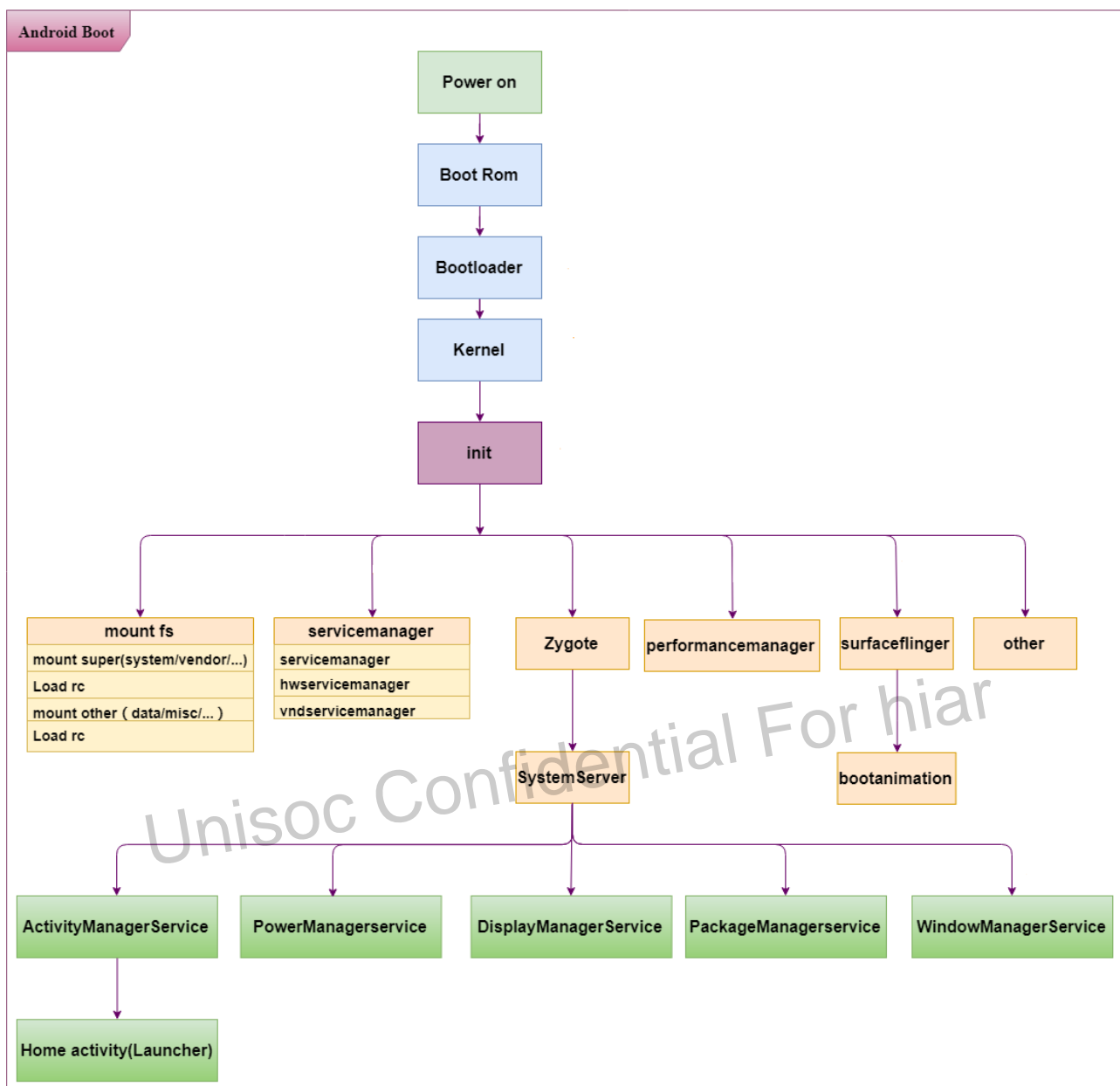
Android 启动流程是自下向上的，大体上可以分为三个阶段：

- BootLoader 引导
- 启动 Linux 内核
- 启动 Android 系统

1.2 Android 系统启动流程

Android 启动流程如图 1-2 所示。

图1-2 Android 启动流程



具体流程如下：

1. Boot Rom

当长按电源开机键开机时，引导芯片从固化在 ROM 的预设代码开始执行，加载引导程序到 RAM 中。

2. BootLoader

即引导程序，主要作用是初始化 flash、将内核 kernel 拷贝到内存中、启动内核。

3. 启动 Kernel

主要是启动内核线程 kernel_init。kernel_init 函数完成设备驱动程序的初始化，并调用 init_post 启动用户空间的 init 进程。

4. 启动 init 进程

在 Linux 系统中，所有的进程都是由 init 进程直接或间接 fork 出来的。init 进程负责创建系统中最关键的几个子进程，尤其是 Zygote 进程，另外，它还提供了 property service，类似于 Windows 系统的注册表服务。

5. 启动 Zygote 进程

当 init 进程创建之后，会 fork 出一个 Zygote 进程，这个进程是所有 Java 进程的父进程。

6. 启动 SystemServer 进程

SystemServer 进程由 Zygote 进程 fork，这个进程在整个 Android 系统中非常重要，系统里面的服务都是在这个进程里面开启的，例如 AMS、WMS 等。

7. 运行 Home Activity

AMS 会通过 Home intent 将 launcher 启动起来。

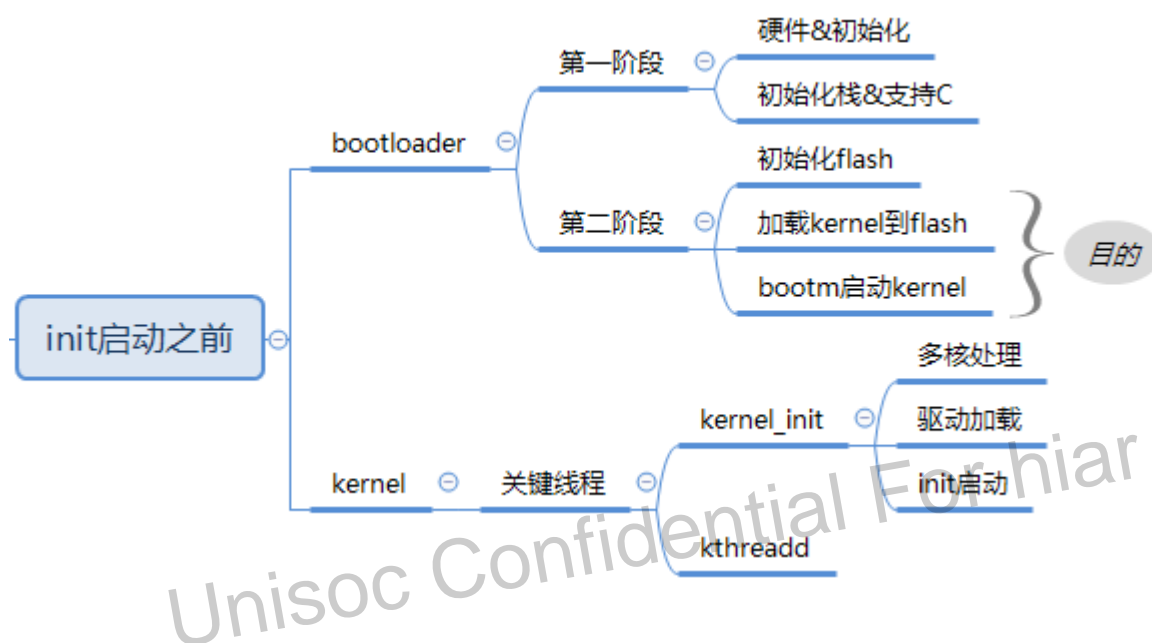
本文后续章节将对开机启动流程的关键任务逐一进行详细介绍。

Unisoc Confidential For hiar

2 Bootloader 及 Kernel 启动

开机启动 init 进程之前的流程包括 bootloader 以及 kernel 的启动。

图2-1 Bootloader 及内核启动



2.1 Bootloader 启动

图2-2 Bootloader 启动模式



上电后 CPU 将执行 bootloader 程序，如图 2-2 所示有如下三种 bootloader 启动模式：

- fastboot 模式

- recovery 模式
- normal 模式

正常启动时走 normal 模式，normal 模式下 bootloader 启动主要分两个阶段：

- 第一阶段
硬件初始化，SVC 模式，关闭中断，关闭看门狗，初始化栈，进入 C 代码。
- 第二阶段
CPU/board/中断初始化，初始化内存与 flash，将 kernel 从 flash 中拷贝到内存中，执行 kernel_init 内核线程来启动内核。

2.2 Kernel 启动

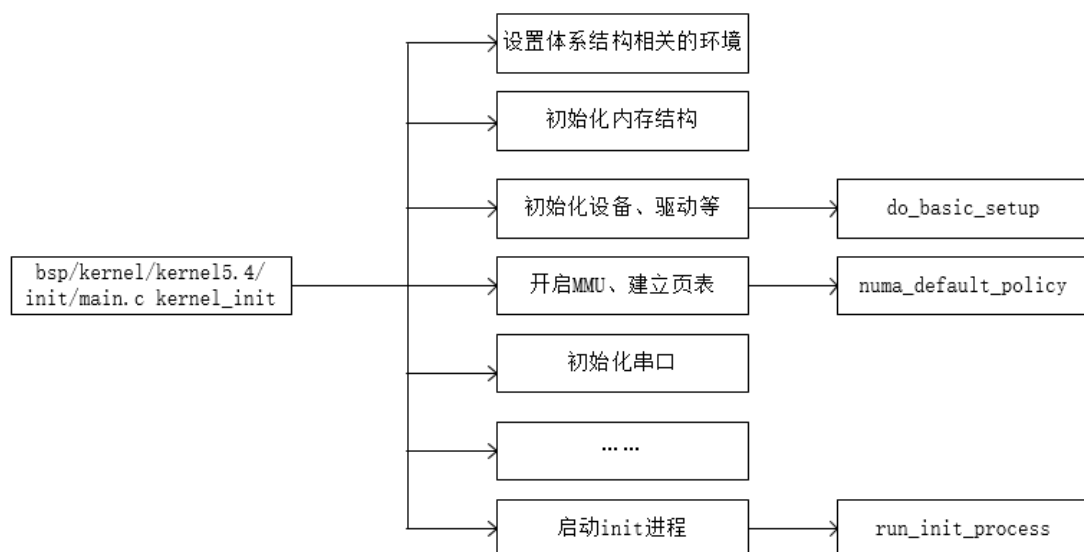
Kernel 启动代码流程如下：

```
start_kernel()
-->rest_init(){创建2个进程}
-->kernel_init
--->执行保存在__initcall_start与__early_initcall_end之间的函数
--->smp多核初始化处理
--->内核驱动模块初始化
--->init_post->run_init_process->kernel_execve启动init，一个用户态线程。
-->另一个kthreadd，一个内核态线程。
```

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，其它就不启动了。

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。

图2-3 Kernel_init 初始化



Kernel 初始化 init 进程的代码如下：

```

static int __ref kernel_init(void *unused)
{
    int ret;

    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，必须完成所有异步 __init 代码
    ftrace_free_init_mem();
    free_initmem();            //释放所有init.* 段中的内存
    mark_readonly();
    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

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

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

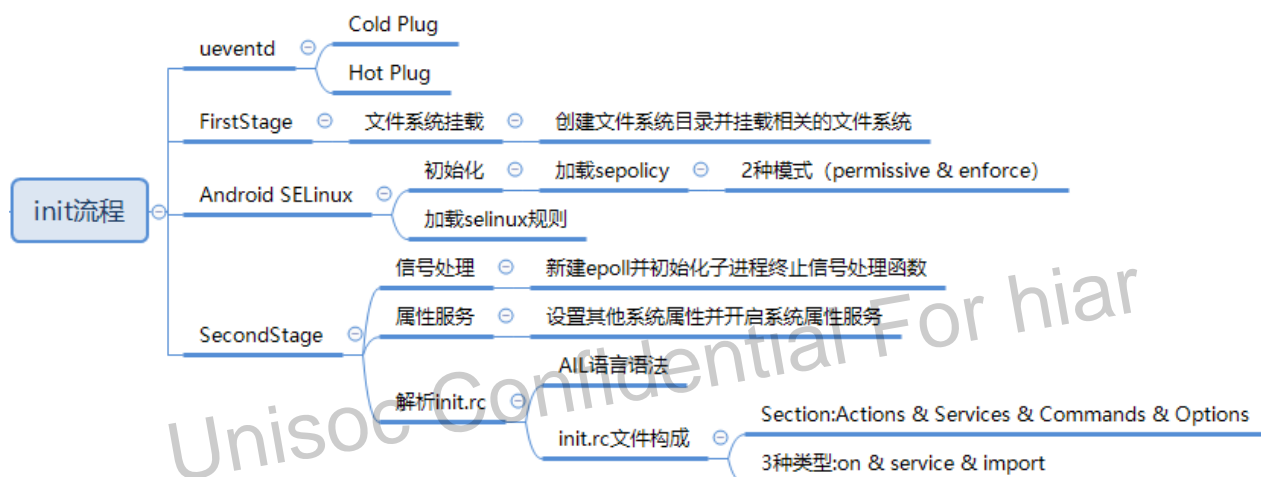
```
.....  
    if (execute_command) {        //execute_command的值如果有定义就去根目录下找对应的应用程序,然后启动  
        ret = run_init_process(execute_command);  
        if (!ret)  
            return 0;  
        panic("Requested init %s failed (error %d).",  
            execute_command, ret);  
    }  
/**  
Linux内核会依照下列的顺序寻找init服务:  
第一步检查/sbin/是否有init服务  
第二步检查/etc/是否有init服务  
第三步检查/bin/是否有init服务  
如果都找不到最后执行/bin/sh  
*/  
    if (!try_to_run_init_process("/sbin/init") ||  
        !try_to_run_init_process("/etc/init") ||  
        !try_to_run_init_process("/bin/init") ||  
        !try_to_run_init_process("/bin/sh"))  
        return 0;  
.....  
}
```

3 Init 启动流程

3.1 Init 进程介绍

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，从而启动其它相关进程以及其它操作。Init 进程执行内容如图 3-1 所示。

图3-1 Init 进程



Android 10.0 的 Init 主函数为 system/core/init/main.cpp，源码如下：

```
/*
 * 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) {
#ifdef __has_feature(address_sanitizer)
    __asan_set_error_report_callback(AsanReportCallback);
#endif

    //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创建、权限设定等一系列工作。
    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) {
        if(!strcmp(argv[1], "subcontext")) {
            //参数为subcontext，初始化日志系统
            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")) {
            android::mboot::mdb("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);
}

```

3.2 Ueventd 子进程

Android 根文件系统的镜像中不存在“/dev”目录，init 进程启动后动态创建该目录。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 代码路径：platform/system/core/init/ueventd.cpp

ueventd 创建设备节点文件有以下两种方式：

- “冷插拔”（Cold Plug）
即以预先定义的设备信息为基础，当 ueventd 启动后，统一创建设备节点文件。这一类设备节点文件也称为静态节点文件。
- “热插拔”（Hot Plug）
即在系统运行中，当有设备插入 USB 端口时，ueventd 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称为动态节点文件。

Ueventd 主函数如下：

```
int ueventd_main(int argc, char** argv) {
    //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
    umask(000);
    //初始化内核日志, 位于节点/dev/kmsg, 此时logd、logcat进程还没有起来,
    //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。
    android::base::InitLogging(argv, &android::base::KernelLogger);
    .....
    //注册selinux相关的用于打印log的回调函数
    SelinuxSetupKernelLogging();
    SelabelInitialize();
    .....
    //解析xml, 根据不同SOC厂商获取不同的hardware rc文件
    auto ueventd_configuration = ParseConfig({"ueventd.rc", "/vendor/ueventd.rc",
                                             "/odm/ueventd.rc", "ueventd." + hardware + ".rc"});
    .....
    //冷启动
    if (access(COLDBOOT_DONE, F_OK) != 0) {
        ColdBoot cold_boot(uevent_listener, uevent_handlers);
        cold_boot.Run();
    }

    for (auto& uevent_handler : uevent_handlers) {
        uevent_handler->ColdbootDone();
    }

    // We use waitpid() in ColdBoot, so we can't ignore SIGCHLD until now.
    //忽略子进程终止信号
    signal(SIGCHLD, SIG_IGN);
    //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
    while (waitpid(-1, nullptr, WNOHANG) > 0) {
    }
    //监听来自驱动的uevent, 进行“热插拔”处理
    uevent_listener.Poll([&uevent_handlers](const Uevent& uevent) {
```

```
    for (auto& uevent_handler : uevent_handlers) {  
        uevent_handler->HandleUevent(uevent);    //热启动，创建设备  
    }  
    return ListenerAction::kContinue;  
});  
return 0;  
}
```

3.3 Init 进程启动第一阶段

Init 进程第一个阶段完成以下内容：

1. 挂载文件系统并创建目录
2. 初始化日志输出、挂载分区设备
3. 启用 SELinux 安全策略

源码路径：system/core/init/first_stage_init.cpp

```
int FirstStageMain(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  
    //重启系统  
    if (REBOOT_BOOTLOADER_ON_PANIC) {  
        InstallRebootSignalHandlers();  
    }  
    .....  
    //清空文件权限  
    umask(0);  
    .....  
    //在RAM内存上获取基本的文件系统，剩余的被rc文件所用  
    CHECKCALL(mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755"));  
    CHECKCALL(mkdir("/dev/pts", 0755));  
    .....  
    //这对于日志包装器是必需的，它在ueventd运行之前被调用  
    CHECKCALL(mknod("/dev/ptmx", S_IFCHR | 0666, makedev(5, 2)));  
    CHECKCALL(mknod("/dev/null", S_IFCHR | 0666, makedev(1, 3)));  
  
    //在第一阶段挂载tmpfs、mnt/vendor、mount/product分区。其它分区不需要在第一阶段加载，  
    //只需要在第二阶段通过rc文件解析来加载。  
    CHECKCALL(mount("tmpfs", "/mnt", "tmpfs", MS_NOEXEC | MS_NOSUID | MS_NODEV,  
        "mode=0755,uid=0,gid=1000"));  
    //创建可供读写的vendor目录
```

```

CHECKCALL(mkdir("/mnt/vendor", 0755));
//创建可供读写的product目录
CHECKCALL(mkdir("/mnt/product", 0755));

//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，
//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
CHECKCALL(mount("tmpfs", "/apex", "tmpfs", MS_NOEXEC | MS_NOSUID | MS_NODEV,
                "mode=0755,uid=0,gid=0"));
.....

//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);
.....
/* 初始化一些必须的分区
 * 主要作用是去解析/proc/device-tree/firmware/android/fstab,
 * 然后得到"/system"、"/vendor"、"/product"等目录的挂载信息
 */
if (!DoFirstStageMount()) {
    LOG(FATAL) << "Failed to mount required partitions early ...";
}
.....
//此处应该是初始化安全框架：Android Verified Boot
//AVB主要用于防止系统文件本身被篡改，还包含了防止系统回滚的功能，
//以免有人试图回滚系统并利用以前的漏洞
SetInitAvbVersionInRecovery();
static constexpr uint32_t kNanosecondsPerMillisecond = 1e6;
uint64_t start_ms = start_time.time_since_epoch().count() / kNanosecondsPerMillisecond;
setenv("INIT_STARTED_AT", std::to_string(start_ms).c_str(), 1);
//启动init进程，传入参数selinux_steup
// 执行命令： /system/bin/init selinux_setup
const char* path = "/system/bin/init";
const char* args[] = {path, "selinux_setup", nullptr};
.....
PLOG(FATAL) << "execv(\"\" << path << "\") failed";

return 1;
}

```

3.3.1 分区挂载

Android 有很多分区，如 system、userdata、cache、vendor、odm 等分区。下面将介绍它们的挂载时机和挂载机制。

3.3.1.1 system/vendor/product 分区

在 Init First Stage 执行期间会解析和校验 Super 分区的元数据，并创建虚拟的块设备对应各个动态分区，从而对 system/vendor/product 等分区进行挂载，上述一切的发起者是 DoFirstStageMount()。

```
bool FirstStageMount::DoFirstStageMount() {
    if (!IsDmLinearEnabled() && fstab_.empty()) { //判断fstab中是否有logical标志，若无，则返回
        // Nothing to mount.
        LOG(INFO) << "First stage mount skipped (missing/incompatible/empty fstab in device tree)";
        return true;
    }
    if (!InitDevices()) return false;
    if (!CreateLogicalPartitions()) return false;
    if (!MountPartitions()) return false; //挂载system并切换为root、vendor、product、overlayfs
    return true;
}
```

文件系统挂载是需要挂载信息的，而这个信息通常保存在 fstab 文件中：fstab.\$(TARGET_BOARD)，对应 device board 里配置的是 fstab.ramdisk 文件。

```
#Dynamic partitions fstab file
#<dev> <mnt_point> <type> <mnt_flags options> <fs_mgr_flags>

system /system ext4 ro,barrier=1 wait,avb=vbmata_system,logical,first_stage_mount,avb_keys=/avb/q-gsi.avbpubkey:/avb/r-gsi.avbpubkey:/avb/s-gsi.avbpubkey
vendor /vendor ext4 ro,barrier=1 wait,avb=vbmata_vendor,logical,first_stage_mount
product /product ext4 ro,barrier=1 wait,avb=vbmata,logical,first_stage_mount
/dev/block/platform/soc/soc:ap-ahb/20600000.sdio/by-name/metadata /metadata ext4 nodev,noatime,nosuid,errors=panic
wait,formattable,first_stage_mount
```

需注意的是，必须包含 “avb=xxx, logical, first_stage_mount” 这样配置才可以在 first_stage 进行 mount。

3.3.1.2 其它分区

其它分区的挂载通过 do_mount_all 来实现，挂载流程如下：

init 进程会根据 init.rc 的规则启动进程或者服务。init.rc 通过 “import /init.xxx.rc” 语句导入平台的规则。

在 device/sprd/mpool/module/partition/main.rc 中就有如下规则：

```
on fs
    mount_all /vendor/etc/fstab.${ro.hardware}
on post-fs
    chmod 0775 /mnt/vendor
chown system system /mnt/vendor
```

mount_all 是一条命令，fstab.\${ro.hardware} 是 mount_all 命令的参数，xxx board 对应的 \${ro.hardware} 就是 xxx（项目名），如 sc9863a1h10 board 对应的 \${ro.hardware} 为 sc9863a1h10。

ActionManager 解析 mount_all 命令，找到对应命令执行函数，fstab.\${ro.hardware} 参数将传递给命令执行函数。

mount_all 命令与对应执行函数的映射关系定义在 system/core/init/builtins.cpp。

```
static const Map builtin_functions = {
.....
    {"mount_all",          {1,      kMax, {false,  do_mount_all}}},
.....
};
```

mount_all 命令对应执行函数为 do_mount_all，/vendor/etc/fstab.sc9863a1h10 是 do_mount_all 函数的传入参数。

开机 log 出现以下 log，说明文件系统挂载成功。

```
init: Command 'mount_all /vendor/etc/fstab.${ro.hardware} --early' action=fs (/vendor/etc/init/init.md.rc:679) took 1143ms and
succeeded
init: Command 'mount_all /vendor/etc/fstab.${ro.hardware} --late' action=late-fs (/vendor/etc/init/init.md.rc:684) took 2232ms and
succeeded
```

3.4 Andorid SELinux

SELinux（Security-Enhanced Linux）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 SELinux 访问控制体系，进程只能访问那些在它的任务中所需要文件。

SELinux 有如下两种工作模式：

- permissive
所有操作都被允许（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。
- 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

下面介绍 SELinux 的初始化及加载 SELinux 规则的过程。

SetupSELinux 函数初始化 SELinux，加载 SELinux 规则，配置 SELinux 相关 log 输出，并启动第二阶段。

SetupSELinux 函数源码路径：system\core\init\selinux.cpp

```
int SetupSELinux(char** argv) {
    InitKernelLogging(argv);                //初始化Kernel日志
    if(REBOOT_BOOTLOADER_ON_PANIC) {        //Debug版本init crash时重启引导加载程序
```

```

    InstallRebootSignalHandlers();
}

SelinuxSetupKernelLogging();           //注册回调，用来设置需要写入kmsg的selinux日志
SelinuxInitialize();                   //加载SELinux规则
/*
 *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，
 *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。
 *其实就是当前在内核域中，在加载Selinux后，需要重新执行init切换到C空间的用户态。
 */
if (selinux_android_restorecon("/system/bin/init", 0) == -1) {
    PLOG(FATAL) << "restorecon failed of /system/bin/init failed";
}

//准备启动init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));

PLOG(FATAL) << "execv(\"\" << path << "\") failed";           //执行 /system/bin/init second_stage, 进入第二阶段
return 1;
}

```

SelinuxInitialize()函数负责加载 SELinux 规则，源码如下：

```

void SelinuxInitialize() {
    Timer t;

    LOG(INFO) << "Loading SELinux policy";
    if (!LoadPolicy()) {
        LOG(FATAL) << "Unable to load SELinux policy";
    }

    bool kernel_enforcing = (security_getenforce() == 1); //获取当前Kernel的工作模式
    bool is_enforcing = IsEnforcing();                     //获取工作模式的配置
    if (kernel_enforcing != is_enforcing) {                //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 (security_setenforce(is_enforcing)) {
            PLOG(FATAL) << "security_setenforce(%s) failed" << (is_enforcing ? "true" : "false");
        }
    }

    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，它们最终都是调用selinux_android_load_policy_from_fd方法。
 */
bool LoadPolicy() {
    return IsSplitPolicyDevice() ? LoadSplitPolicy() : LoadMonolithicPolicy();
}
```

3.5 Init 进程启动第二阶段

Init 进程第二阶段完成以下主要内容：

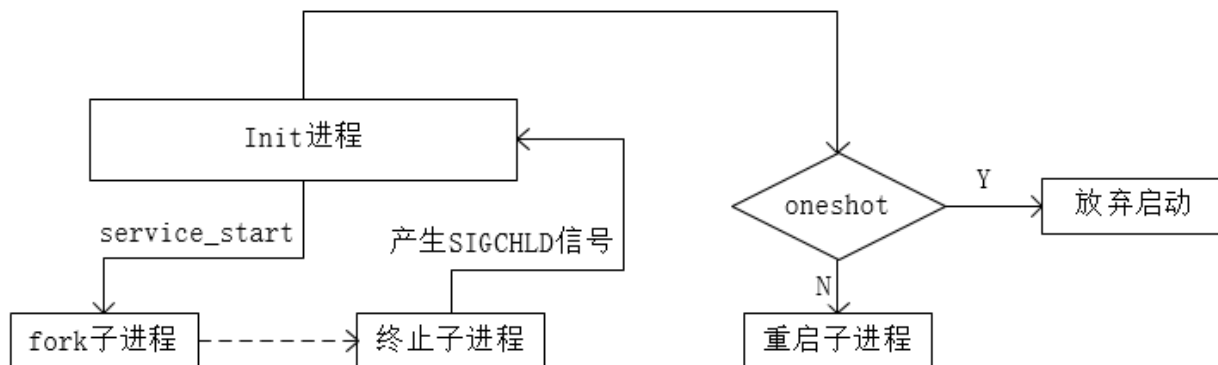
1. 创建进程会话密钥并初始化属性系统。
2. 进行 SELinux 第二阶段并恢复一些文件安全上下文。
3. 新建 `epoll` 并初始化子进程终止信号处理函数。
4. 启动匹配属性的服务端。
5. 解析 `init.rc` 等文件，建立 `rc` 文件的 `action`、`service`，启动其它进程。

3.5.1 信号处理

Init 是一个守护进程，为防止 Init 的子进程成为僵尸进程（zombie process），需要 Init 在子进程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，防止子进程成为僵尸进程（僵尸进程会一直占用程序表的空间，当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）。

Init 进程创建、终止以及重启子进程的流程如图 3-2 所示。

图3-2 Init 创建、终止以及重启子进程的流程



子进程信号处理包含以下内容：

- 初始化 Signal 句柄
- 循环处理子进程
- 注册 epoll 句柄
- 处理子进程终止

3.5.1.1 信号初始化

在 linux 中子进程终止时发出 SIGCHLD 信号，父进程捕捉到 SIGCHLD 信号得知子进程已结束运行。

Init 进程使用 InstallSignalFdHandler 函数初始化子进程的 SIGCHLD 信号，该函数源码路径如下：
system/core/init/init.cpp。

```

static void InstallSignalFdHandler(Epoll* epoll) {
// SA_NOCLDSTOP使init进程只有在其子进程终止时才会受到SIGCHLD信号
    const struct sigaction act { .sa_handler = SIG_DFL, .sa_flags = SA_NOCLDSTOP };
    sigaction(SIGCHLD, &act, nullptr);
    .....
    if (!IsRebootCapable()) {
        //如果init不具有CAP_SYS_BOOT的能力，则它此时正值容器中运行。
        //在这种场景下，接收SIGTERM将会导致系统关闭。
        sigaddset(&mask, SIGTERM);
    }
    .....
//注册处理程序以解除对子进程中的信号的阻止
    const int result = pthread_atfork(nullptr, nullptr, &UnblockSignals);
    if (result != 0) {
        LOG(FATAL) << "Failed to register a fork handler: " << strerror(result);
    }

//创建信号句柄

```



```
signal_fd = signalfd(-1, &mask, SFD_CLOEXEC);
if (signal_fd == -1) {
    PLOG(FATAL) << "failed to create signalfd";
}
//信号注册，当signal_fd收到信号时，触发HandleSignalFd。
if (auto result = epoll->RegisterHandler(signal_fd, HandleSignalFd); !result) {
    LOG(FATAL) << result.error();
}
}
```

3.5.1.2 信号注册

信号注册函数 `RegisterHandler` 的源码路径：`system/core/init/epoll.cpp`，该函数完成信号注册，把 `fd` 句柄加入 `epoll_fd` 的监听队列中。

```
Result<Success> Epoll::RegisterHandler(int fd, std::function<void()> handler, uint32_t events) {
    if (!events) {
        return Error() << "Must specify events";
    }
    .....
    //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<Success> result = ErrnoError() << "epoll_ctl failed to add fd";
        epoll_handlers_.erase(fd);
        return result;
    }
    return Success();
}
```

3.5.1.3 SIGCHLD 信号监控

`HandleSignalFd` 函数监控 `SIGCHLD` 信号，调用 `ReapAnyOutstandingChildren` 函数来终止出现问题的子进程。

`HandleSignalFd` 函数源码路径：`system/core/init/init.cpp`

```
static void HandleSignalFd() {
    signalfd_siginfo siginfo;
    ssize_t bytes_read = TEMP_FAILURE_RETRY(read(signal_fd, &siginfo, sizeof(siginfo)));
    .....
    //监控SIGCHLD信号
    switch (siginfo.ssi_signo) {
        case SIGCHLD:
            ReapAnyOutstandingChildren();
            break;
        case SIGTERM:
    }
```

```

        HandleSigtermSignal(signinfo);
        break;
    default:
        PLOG(ERROR) << "signal_fd: received unexpected signal " << signinfo.ssi_signo;
        break;
    }
}

```

3.5.1.4 信号处理

ReapOneProcess 是最终的处理函数，该函数处理大致如下：

1. 首先使用 waitpid 找出挂掉进程的 pid
2. 然后根据 pid 找到对应 Service
3. 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。

源码路径：system/core/init/sigchld_handle.cpp

```

static bool ReapOneProcess() {
    signinfo_t signinfo = {};
    //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 (TEMP_FAILURE_RETRY(waitid(P_ALL, 0, &signinfo, WEXITED | WNOHANG | WNOWAIT)) != 0) {
        PLOG(ERROR) << "waitid failed";
        return false;
    }
    .....
    //如果存在僵尸pid，使用scopeguard来清除僵尸pid
    auto reaper = make_scope_guard([pid] { TEMP_FAILURE_RETRY(waitpid(pid, nullptr, WNOHANG)); });
    .....
    if (SubcontextChildReap(pid)) {
        name = "Subcontext";
    } else {
        //通过pid找到对应的service
        service = ServiceList::GetInstance().FindService(pid, &Service::pid);
        .....
    }
    .....
    //没有找到service，说明已经结束了，退出
    if (!service) return true;

    service->Reap(signinfo);    //清除子进程相关的资源

    if (service->flags() & SVC_TEMPORARY) {

```

```
ServiceList::GetInstance().RemoveService(*service);    //移除该service
}

return true;
}
```

3.5.2 属性服务

Android 将属性设置统一交由 init 进程管理，其他进程不能直接修改属性，只能通知 init 进程来修改，在修改过程中，init 进程可以进行权限控制。大部分 property 是记录在某些文件中的，init 进程启动的时候，会加载这些文件，完成 property 系统初始化。

3.5.2.1 属性初始化

property_init 函数初始化属性系统，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。

源码路径：system/core/property_service.cpp。

```
void property_init() {
    //设置SELinux回调，进行权限控制
    selinux_callback cb;
    cb.func_audit = PropertyAuditCallback;
    selinux_set_callback(SELINUX_CB_AUDIT, cb);

    mkdir("/dev/_properties_", S_IRWXU | S_IXGRP | S_IXOTH);
    CreateSerializedPropertyInfo();
    if (__system_property_area_init()) {
        LOG(FATAL) << "Failed to initialize property area";
    }
    if (!property_info_area.LoadDefaultPath()) {
        LOG(FATAL) << "Failed to load serialized property info file";
    }
}
```

3.5.2.2 属性服务启动

StartPropertyService 函数启动属性服务。

该函数首先创建一个 socket 并返回文件描述符，然后设置最大并发数为 8，其它进程可以通过这个 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。

源码路径：system/core/init/property_service.cpp

```
void StartPropertyService(int* epoll_socket) {
    property_set("ro.property_service.version", "2");
```

```

.....
init_socket = sockets[1];
//建立socket连接
property_set_fd = CreateSocket(PROP_SERVICE_NAME, SOCK_STREAM | SOCK_CLOEXEC | SOCK_NONBLOCK,
                                false, 0666, 0, 0, nullptr);
.....

//最大监听8个并发
listen(property_set_fd, 8);

std::thread{PropertyServiceThread}.detach();
//注册property_set_fd，当收到句柄改变时，通过handle_property_set_fd来处理。
property_set = [](const std::string& key, const std::string& value) -> uint32_t {
    android::base::SetProperty(key, value);
    return 0;
};
}

```

3.5.2.3 属性处理

handle_property_set_fd 函数建立 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 的开始、停止、重启操作，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。

源码路径：system/core/init/property_service.cpp

```

static void handle_property_set_fd() {
    static constexpr uint32_t kDefaultSocketTimeout = 2000; /* ms */
    //等待客户端连接
    int s = accept4(property_set_fd, nullptr, nullptr, SOCK_CLOEXEC);
    if (s == -1) {
        return;
    }

    ucred cr;
    socklen_t cr_size = sizeof(cr);
    //获取连接到此socket的进程的凭据
    if (getsockopt(s, SOL_SOCKET, SO_PEERCRED, &cr, &cr_size) < 0) {
        close(s);
        PLOG(ERROR) << "sys_prop: unable to get SO_PEERCRED";
        return;
    }
}

```

```
//建立socket连接
SocketConnection socket(s, cr);
uint32_t timeout_ms = kDefaultSocketTimeout;

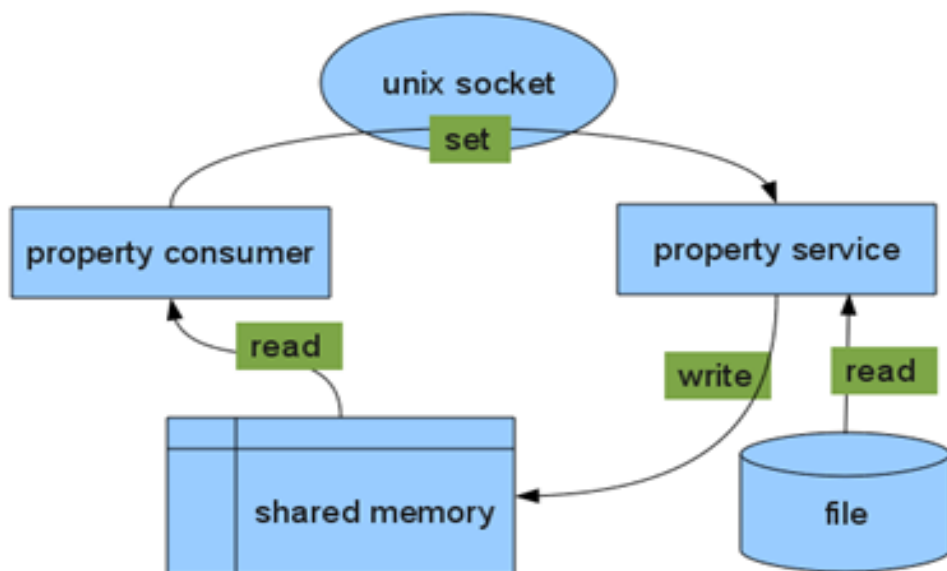
uint32_t cmd = 0;
//读取socket中的操作信息
if (!socket.RecvUint32(&cmd, &timeout_ms)) {
    PLOG(ERROR) << "sys_prop: error while reading command from the socket";
    socket.SendUint32(PROP_ERROR_READ_CMD);
    return;
}
//根据操作信息，执行对应处理,两者区别一个是以char形式读取，一个以String形式读取
switch (cmd) {
case PROP_MSG_SETPROP: {
.....
        break;
    }

case PROP_MSG_SETPROP2: {
.....
        break;
    }

default:
.....
        break;
    }
}
```

创建 `property_service` socket 用于监听进程修改 `property` 请求，通过 `handle_property_set_fd` 来处理请求，`set property msg` 分为两类处理，`msg name` 以 “ctl.” 为起始的 `msg` 通过 `handle_control_message` 处理，主要是启动、停止、重启服务。修改其它 `prop` 时会调用 `property_get`，然后通过 `bionic` 的 `__system_property_set` 函数来实现，而这个函数会通过 `socket` 与 `init` 的 `property service` 取得联系。整个 `property` 访问的过程可以用下图来表述：

图3-3 属性访问过程



3.5.2.4 属性加载顺序

property_load_boot_defaults 函数加载 property 文件，在整个启动流程中，property 的加载是在 Rc 文件加载之前的。

源码路径：system/core/init/property_service.cpp

```

void property_load_boot_defaults(bool load_debug_prop) {
    std::map<std::string, std::string> properties;
    if (!load_properties_from_file("/system/etc/prop.default", nullptr, &properties)) {
        // Try recovery path
        if (!load_properties_from_file("/prop.default", nullptr, &properties)) {
            // Try legacy path
            load_properties_from_file("/default.prop", nullptr, &properties);
        }
    }
    load_properties_from_file("/system/build.prop", nullptr, &properties);
    load_properties_from_file("/vendor/default.prop", nullptr, &properties);
    load_properties_from_file("/vendor/build.prop", nullptr, &properties);
    if (SelinuxGetVendorAndroidVersion() >= __ANDROID_API_Q__) {
        load_properties_from_file("/odm/etc/build.prop", nullptr, &properties);
    } else {
        load_properties_from_file("/odm/default.prop", nullptr, &properties);
        load_properties_from_file("/odm/build.prop", nullptr, &properties);
    }
    load_properties_from_file("/product/build.prop", nullptr, &properties);
    load_properties_from_file("/product_services/build.prop", nullptr, &properties);
}
    
```

```
load_properties_from_file("/factory/factory.prop", "ro.*", &properties);
.....

property_initialize_ro_product_props();
property_derive_build_fingerprint();

update_sys_usb_config();
}
```

3.5.3 init.rc

属性服务建立完成后，init 会启动其它进程，每个其它进程都有对应的二进制文件，通过 `exec` 命令执行指定进程对应的二进制文件来启动相应进程，例如执行 `/system/bin/init second_stage`，来启动 init 进程的第二阶段。

Android 系统有许多 Native 进程，因此 Android 推出 `init.rc` 机制（`init.rc` 是一个配置文件，是由 Android 初始化语言编写的脚本构成），通过类似读取配置文件的方式，启动不同的进程。

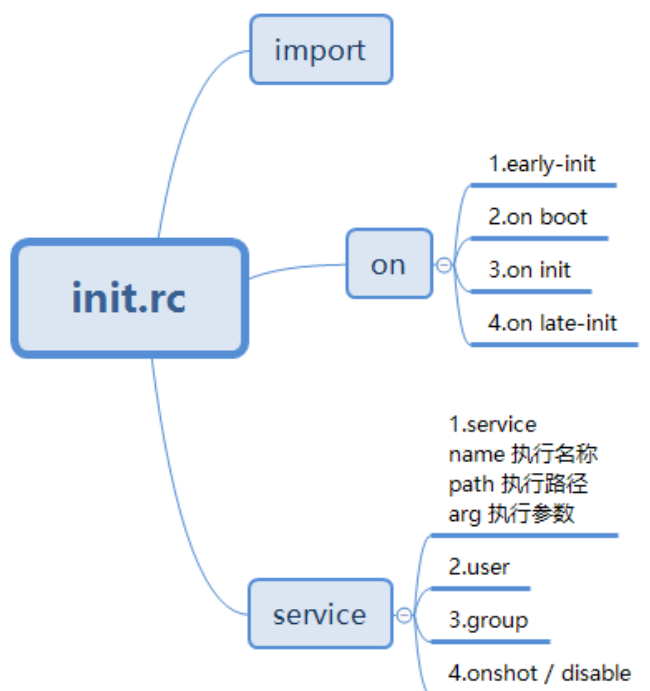
`init.rc` 文件路径： `./init.rc`

`init.rc` 常用的五种语句如下：

- Action
- Command
- Service
- Option
- Import

`init.rc` 组成结构如图 3-4 所示。

图3-4 init.rc 组成结构



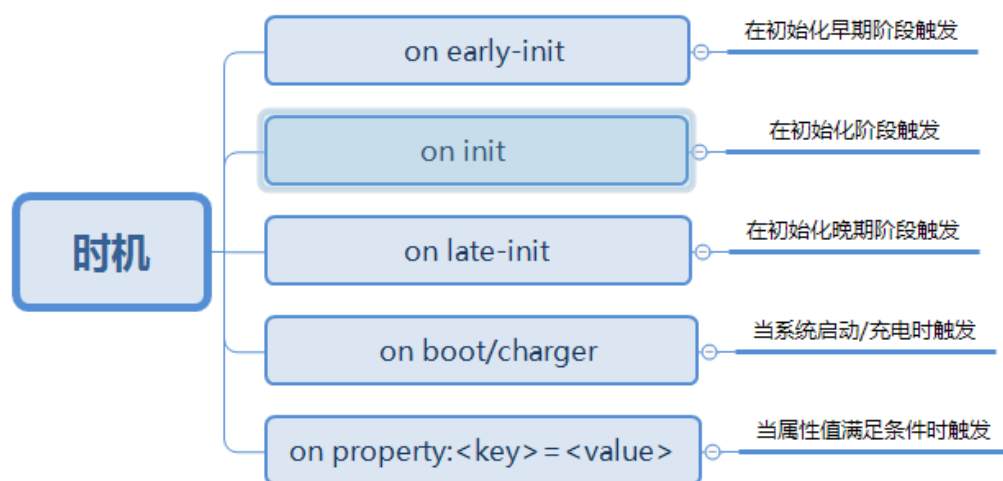
3.5.3.1 init.rc 语句

Action

Action 由一组命令（commands）组成。Action 包括一个触发器，决定何时执行这个动作。

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，时机类型如图 3-5 所示。

图3-5 时机类型

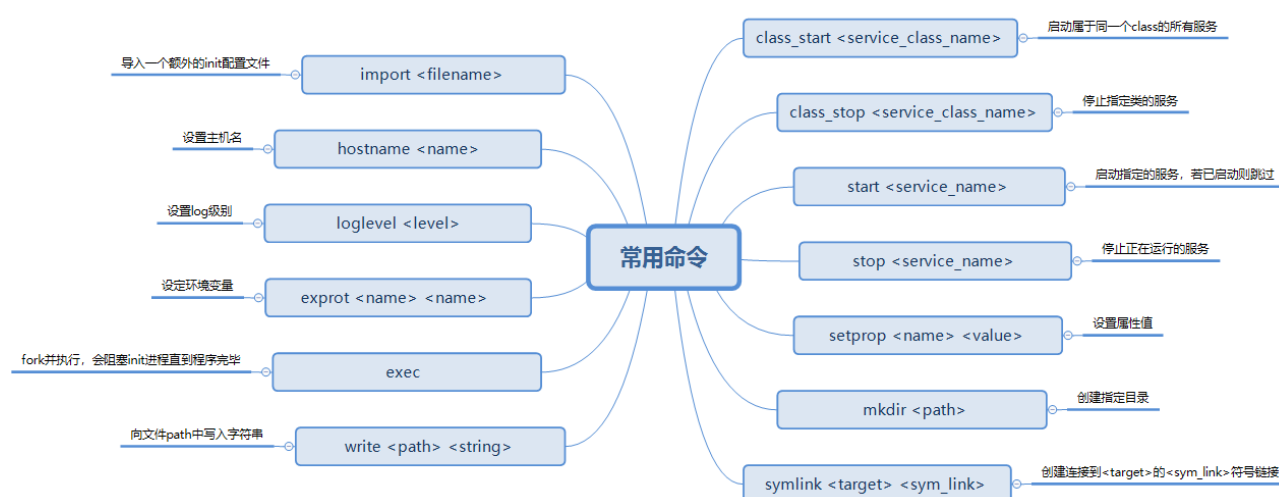


Command

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。

常用命令如图 3-6 所示。

图3-6 常用命令



Service

Service 以 service 开头，由 init 进程启动，一般运行在 init 的一个子进程，启动 service 前需要判断对应的可执行文件是否存在。

Service 语句格式：service <name><pathname> [<argument>]* <option> <option>

参数	含义
<name>	服务名称。
<pathname>	服务的可执行文件所在路径。
<argument>	启动服务所需参数。
<option>	服务的约束选项。

每个 service 在启动时会通过 fork 方式生成相应的子进程，init 生成的子进程定义在 rc 文件中。

例如：service servicemanager /system/bin/servicemanager

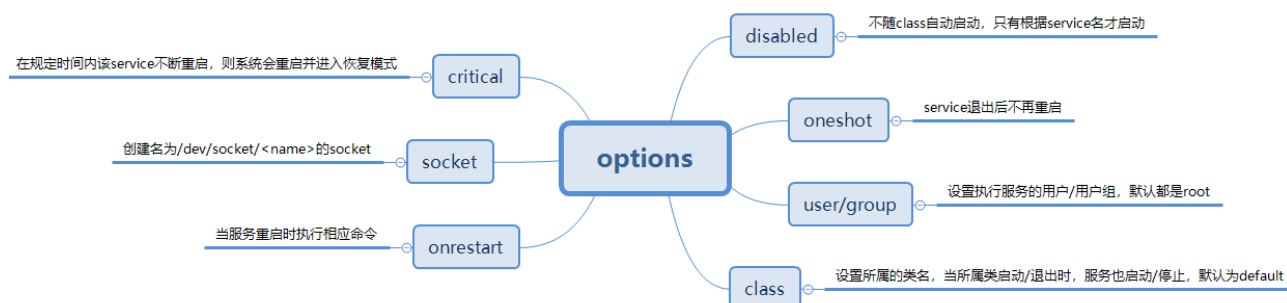
服务名称：servicemanager

服务的可执行文件路径：/system/bin/servicemanager

Option

Option 是 Service 的可选项，与 Service 配合使用，常见可选项如图 3-7 所示。

图3-7 常见可选项



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

import

import 命令用于导入其它的 rc 文件

import 命令格式: import <filename>

Action 执行顺序与启动阶段设置说明

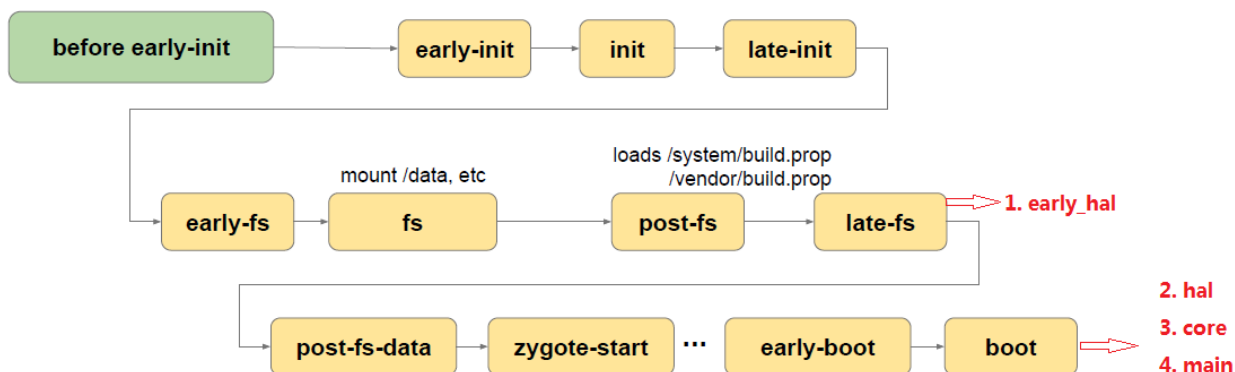
自定义 service 时通过声明 class xxx 来指定 service 的启动阶段，如何选取启动阶段呢？

系统目前常用的按执行先后顺序的启动阶段如下：

- early_hal (HALs required before storage encryption can get unlocked (FBE/FDE))
- hal (normal hal)
- core (system core base service)
- main (normal service)
- late_start (the last boot class, wait for all system resource ready)

Action 执行顺序与启动阶段的对应关系如图 3-8 所示，Action/Service 可以按照其时序结合自身需求添加对应 class。

图3-8 Action 执行顺序与启动阶段的对应关系图



3.5.3.2 init.rc 解析

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

完成 init.rc 解析后，解析 /system/etc/init、/product/etc/init、/product_services/etc/init、/odm/etc/init、/vendor/etc/init 等路径下的 rc 文件。

代码路径：system\core\init\init.cpp

```
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()) {
```

```
        std::string bootmode = GetProperty("ro.bootmode", "");
```

```
        if (bootmode == "charger") {
```

```
            parser.ParseConfig("/vendor/etc/init/charge.rc");
```

```
        } else {
```

```
            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");
```

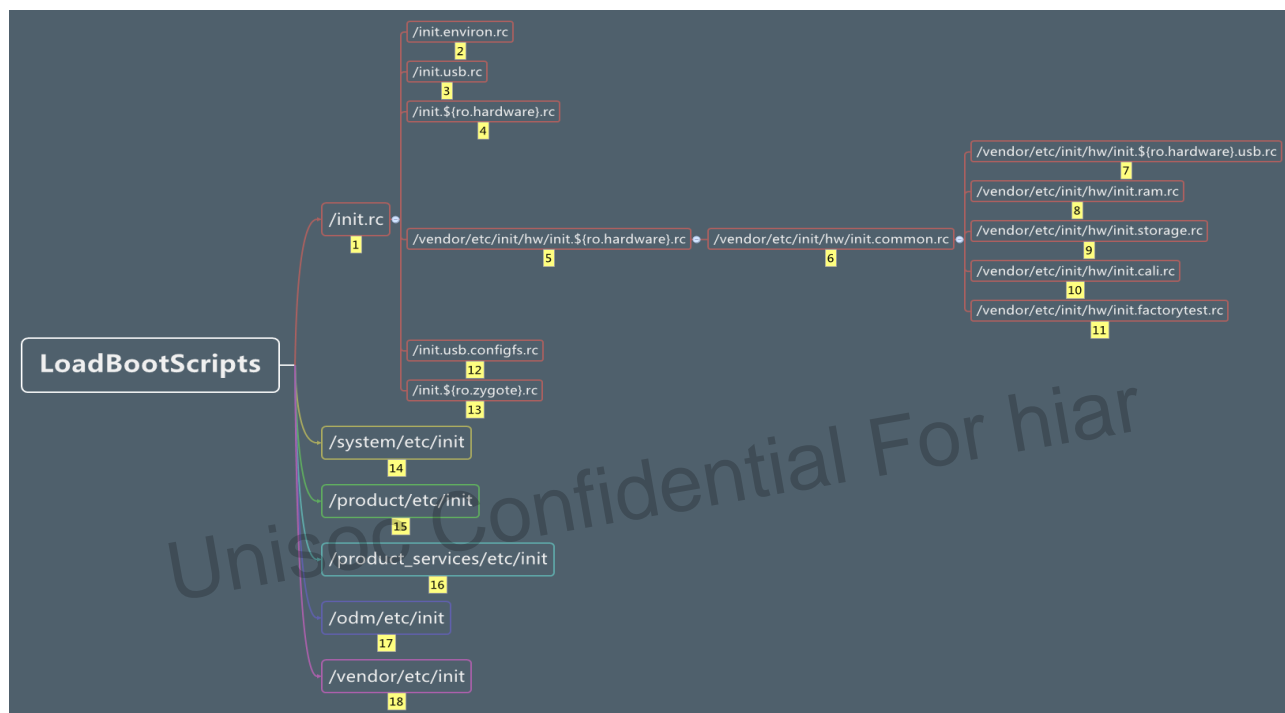
```
            }
```

```
            if (!parser.ParseConfig("/vendor/etc/init")) {
```

```
late_import_paths.emplace_back("/vendor/etc/init");
}
}
} else {
    parser.ParseConfig(bootscript);
}
}
```

在正常启动流程中，rc 文件加载顺序如图 3-9 所示。

图3-9 LoadBootScripts 加载 rc 文件顺序



每个 Service 都有自己的 rc 文件，这些 rc 文件基本都在/system/etc/init、/vendor/etc/init、/odm/etc/init 等目录下，完成 init.rc 解析后，会解析这些目录下的 rc 文件并执行相关动作。

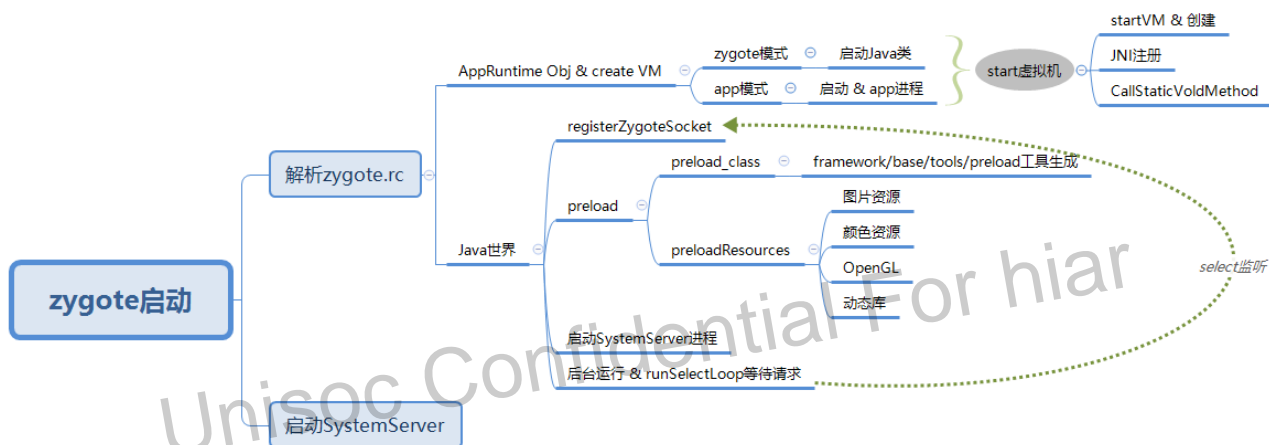
结合代码与开机 log 可以看到，init 会先加载/system/etc/init/init.rc，然后再加载其它路径下的 rc 文件。

4 zygote 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 和其它所有 Dalvik 虚拟机进程都是由 zygote fork 而来。

zygote 进程是 init 进程根据 init.rc 文件中的配置项创建的，实现了从 native 层到 java 层的调用。Zygote 最初的名字叫 app_process（在 Android.mk 文件中指定的），但 app_process 在运行过程中通过 pctrl 系统调用将自己的名字换成了 zygote，因此通过 ps 命令看到的进程名是 zygote。

图4-1 zygote 进程



4.1 zygote.rc 解析

init 进程启动后会解析 zygote.rc，在/system/core/rootdir/init.rc 文件中包含了 zygote.rc。

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

其中\${ro.zygote}是平台相关的参数，实际可对应到 init.zygote32.rc、init.zygote64.rc、init.zygote64_32.rc、init.zygote32_64.rc，前两个只会启动单一 app_process 进程，而后两个则会启动两个 app_process 进程：第二个 app_process 进程称为 secondary（有相应 secondary socket 的创建过程）。

以/system/core/rootdir/init.zygote64.rc 为例：

```
service zygote /system/bin/app_process64 -Xzygote /system/bin --zygote --start-system-server
    class main
    priority -20
    user root
    group root readproc reserved_disk
```

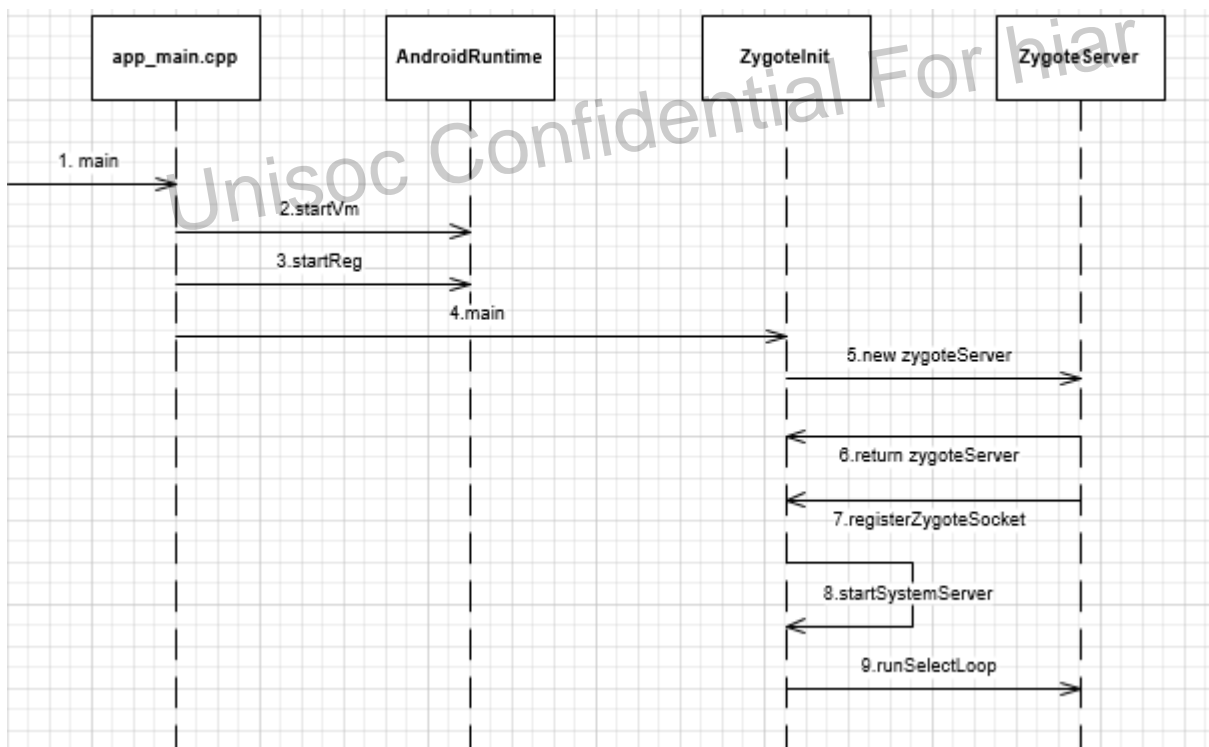
```
socket zygote stream 660 root system
socket usap_pool_primary stream 660 root system
onrestart write /sys/android_power/request_state wake
onrestart write /sys/power/state on
onrestart restart audioserver
onrestart restart cameraserver
onrestart restart media
onrestart restart netd
onrestart restart wificond
writepid /dev/cpuset/foreground/tasks
```

首先创建了名为 zygote 的进程，这个进程是通过 app_process64 的 main 启动并以对应参数作为 main 的入口参数。

4.2 zygote 启动

zygote 启动流程如图 4-2 所示。

图4-2 zygote 启动流程



zygote 启动具体流程说明如下：

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` 完成大部分工作，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。

4.2.1 启动入口

源码路径：frameworks/base/cmds/app_process/app_main.cpp

`zygote` 的原型 `app_process` 所对应的源文件是 `app_main.cpp`，核心代码如下所示：

//zygote进程由init通过fork而来，init.rc中设置的启动参数：

```
int main(int argc, char* const argv[])
```

```
{
```

```
.....
```

```
//zygote传入的参数argv为“-Xzygote /system/bin --zygote --start-system-server --socket-name=zygote”
```

```
//zygote_secondary传入的参数argv为“-Xzygote /system/bin --zygote --socket-name=zygote_secondary”
```

```
while (i < argc) {
```

```
    const char* arg = argv[i++];
```

```
    if (strcmp(arg, "--zygote") == 0) {
```

```
        zygote = true;
```

```
        //对于64位系统nice_name为zygote64; 32位系统为zygote
```

```
        niceName = ZYGOTE_NICE_NAME;
```

```
    } else if (strcmp(arg, "--start-system-server") == 0) {
```

```
        startSystemServer = true;    //是否需要启动system server
```

```
    } else if (strcmp(arg, "--application") == 0) {
```

```
        application = true;    //启动进入独立的程序模式
```

```
    } else if (strncmp(arg, "--nice-name=", 12) == 0) {
```

```
        niceName.setTo(arg + 12);    //niceName为当前进程别名，区别abi型号
```

```
    } else if (strncmp(arg, "--", 2) != 0) {
```

```
        className.setTo(arg);
```

```
        break;
```

```
    } else {
```

```
        --i;
```

```
        break;
```

```
}
```

```

}

Vector<String8> args;
if (!className.isEmpty())
{
.....
    //app mode,启动app时才不为空
    args.add(application ? String8("application") : String8("tool"));
    runtime.setClassNameAndArgs(className, argc - i, argv + i);
.....
}
else
{
    maybeCreateDalvikCache();    //进入zygote模式，新建Dalvik的缓存目录:/data/dalvik-cache
    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
    //参数的解析与处理，end
.....
}

//AppRuntime类继承了AndroidRuntime类，重载了onStarted、onZygoteInit和onExit函数
if (!niceName.isEmpty())
{
    //设置一个“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
{

```



```
fprintf(stderr, "Error: no class name or --zygote supplied.\n");
app_usage();    //没有指定类名或zygote, 参数错误
LOG_ALWAYS_FATAL("app_process: no class name or --zygote supplied.");
}
}
```

上述代码中 zygote 主要做了 2 件事情:

1. zygote 模式下参数初始化, 调用 runtime (AndroidRuntime) 的 start 方法启动 java 类。
2. app 启动模式下参数初始化, 调用 runtime (AndroidRuntime) 的 start 方法启动 app 进程。

runtime 的 start 方法的源码路径: frameworks/base/core/jni/AndroidRuntime.cpp

```
void AndroidRuntime::start(const char* className, const Vector<String8>& options, bool zygote)
{
.....
    JniInvocation jni_invocation;
    jni_invocation.Init(NULL);
    JNIEnv* env;
    // 虚拟机创建, 主要是设置虚拟机参数
    if (startVm(&mJavaVM, &env, zygote) != 0) {
        return;
    }
    onVmCreated(env);    //空函数, 没有任何实现

    //注册JNI函数
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }

    //启动Java类, 对于zygote模式是ZygoteInit类。
    jclass stringClass;
    jobjectArray strArray;
    jstring classNameStr;
.....
    //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);
    if (startClass == NULL) {        //找到Zygoteinit类
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
        /* keep going */
    } else {
```

```
//找到这个类后就继续找成员函数main方法的Method ID
jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
    "([Ljava/lang/String;)V");
if (startMeth == NULL) {
    ALOGE("JavaVM unable to find main() in '%s'\n", className);
    /* keep going */
} else {
    //通过反射调用ZygoteInit.main()方法
    env->CallStaticVoidMethod(startClass, startMeth, strArray);
.....
}
```

start()函数主要做了三件事情：

- 调用 startVm 开启虚拟机
- 调用 startReg 注册 JNI 方法
- 使用 JNI 把启动 Zygote 进程

相关 log 如下：

```
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
```

4.2.2 JNI 初始化

[JniInvocation.cpp] Init 函数主要作用是初始化 JNI。

JNI 初始化工作如下：

1. 通过 dlopen 加载 libart.so 获得其句柄。
2. 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。(这三个函数会在创建虚拟机时调用)

JNI 初始化源码路径：libnativehelper/JniInvocation.cpp

```
bool JniInvocationImpl::Init(const char* library) {
    /*
     * 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
     * 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
     * 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
     */
    library = GetLibrary(library, buffer);
    handle_ = OpenLibrary(library);    // 获取libart.so的句柄
    if (handle_ == NULL) {            //获取失败打印错误日志并尝试再次打开libart.so
        if (strcmp(library, kLibraryFallback) == 0) {
            // Nothing else to try.
```

```

    ALOGE("Failed to dlopen %s: %s", library, GetError().c_str());
    return false;
}
library = kLibraryFallback;
handle_ = OpenLibrary(library);
if (handle_ == NULL) {
    ALOGE("Failed to dlopen %s: %s", library, GetError().c_str());
    return false;
}
}
/*
 * 1.FindSymbol函数内部实际调用的是dlsym
 * 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<FUNC_POINTER*>(&JNI_GetDefaultJavaVMInitArgs_),
                "JNI_GetDefaultJavaVMInitArgs")) {
    return false;
}
.....
return true;
}

void* OpenLibrary(const char* filename) {
#ifdef _WIN32
    return LoadLibrary(filename);
#else
    const int kDlopenFlags = RTLD_NOW | RTLD_NODELETE;
    return dlopen(filename, kDlopenFlags);
#endif
}

```

4.2.3 虚拟机创建

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。

源码路径：frameworks/base/core/jni/AndroidRuntime.cpp

```

int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv, bool zygote)
{
    .....
    //这个函数绝大部分代码都是设置虚拟机的参数
    // 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;
    }
}
```

```
.....
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();
initArgs.nOptions = mOptions.size();
initArgs.ignoreUnrecognized = JNI_FALSE;
```

//调用之前JniInvocation初始化的JNI_CreateJavaVM_，参考[4.2.2]

```
if (JNI_CreateJavaVM(pJavaVM, pEnv, &initArgs) < 0) {
    ALOGE("JNI_CreateJavaVM failed\n");
    return -1;
}
```

```

}

return 0;
}

```

4.2.4 JNI 注册

函数 `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。

源码路径： `frameworks/base/core/jni/AndroidRuntime.cpp`

```

static const RegJNIRec gRegJNI[] = {
    REG_JNI(register_com_android_internal_os_RuntimeInit),
    REG_JNI(register_com_android_internal_os_ZygoteInit_nativeZygoteInit),
    .....
    REG_JNI(register_com_android_internal_os_FuseAppLoop),
};

int AndroidRuntime::startReg(JNIEnv* env)
{
    ATRACE_NAME("RegisterAndroidNatives");
    //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
    androidSetCreateThreadFunc((android_create_thread_fn) javaCreateThreadEtc);
    .....
    //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
    env->PushLocalFrame(200);
    //注册jni函数，gRegJNI是一个全局数组。
    if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
        env->PopLocalFrame(NULL);
        return -1;
    }
    env->PopLocalFrame(NULL);    //释放局部引用作用域
    return 0;
}

```

4.3 Java 框架层启动

通过 JNI 调用 `ZygoteInit` 的 `main` 函数后，`zygote` 便进入了 Java 框架层，换言之就是 `zygote` 开创了 Java 框架层。

源码路径： `frameworks/base/core/java/com/android/internal/os/ZygoteInit.java`

`ZygoteInit` 主函数的主要工作如下：

1. 调用 `preload()` 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。

```
public static void main(String argv[]) {
    // 1.创建ZygoteServer
    ZygoteServer zygoteServer = null;

    //调用native函数, 确保当前没有其它线程在运行。
    ZygoteHooks.startZygoteNoThreadCreation();

    //设置pid为0, Zygote进入自己的进程组。
    try {
        Os.setpgid(0, 0);
    } catch (ErrnoException ex) {
        throw new RuntimeException("Failed to setpgid(0,0)", ex);
    }

    Runnable caller;
    try {
        .....
        //得到systrace的监控TAG
        String bootTimeTag = Process.is64Bit() ? "Zygote64Timing" : "Zygote32Timing";
        TimingsTraceLog bootTimingsTraceLog = new TimingsTraceLog(bootTimeTag,
                                                                    Trace.TRACE_TAG_DALVIK);

        //通过systradce来追踪函数ZygoteInit, 可以通过systrace工具来进行分析。
        //traceBegin和traceEnd要成对出现, 而且需要使用同一个tag。
        bootTimingsTraceLog.traceBegin("ZygoteInit");
        //开启DDMS(Dalvik Debug Monitor Service)功能
        //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。
        RuntimeInit.enableDdms();
        .....
        //2. 解析app_main.cpp - start()传入的参数
        for (int i = 1; i < argv.length; i++) {
            if ("start-system-server".equals(argv[i])) {
                startSystemServer = true;    //启动zygote时, 才会传入参数: start-system-server
            } else if ("--enable-lazy-preload".equals(argv[i])) {

```

```

        enableLazyPreload = true;    //启动zygote_secondary时，才会传入参数：enable-lazy-preload
    } else if (argv[i].startsWith(ABI_LIST_ARG)) {
        //通过属性ro.product.cpu.abi64\ro.product.cpu.abi32从C空间传来的值
        abiList = argv[i].substring(ABI_LIST_ARG.length());    //会有两种值：zygote和zygote_secondary
    } else if (argv[i].startsWith(SOCKET_NAME_ARG)) {
        zygoteSocketName = argv[i].substring(SOCKET_NAME_ARG.length());
    } else {
        throw new RuntimeException("Unknown command line argument: " + argv[i]);
    }
}

//根据传入socket name来决定是创建socket还是zygote_secondary
final boolean isPrimaryZygote = zygoteSocketName.equals(Zygote.PRIMARY_SOCKET_NAME);

.....

//在第一次zygote启动时，enableLazyPreload为false，执行preload
if (!enableLazyPreload) {
    //systrace追踪ZygotePreload
    bootTimingsTraceLog.traceBegin("ZygotePreload");
    EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
        SystemClock.uptimeMillis());
    // 3.加载进程的资源 and 类
    preload(bootTimingsTraceLog);
    EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
        SystemClock.uptimeMillis());
    //systrace结束ZygotePreload的追踪
    bootTimingsTraceLog.traceEnd(); // ZygotePreload
} else {
    //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
    Zygote.resetNicePriority();
}

//结束ZygoteInit的systrace追踪
bootTimingsTraceLog.traceBegin("PostZygoteInitGC");

.....

//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
zygoteServer = new ZygoteServer(isPrimaryZygote);

```

```

    if (startSystemServer) {
        //5. fork出system server
        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);
    .....
    }
    // 7.在子进程中退出了选择循环。继续执行命令
    if (caller != null) {
        caller.run();
    }
}

```

开机 log 如下:

```

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

```

预加载（preload）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。

```

static void preload(TimingsTraceLog bootTimingsTraceLog) {
    .....
    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。
}

```



```

//从上面的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
// Ask the WebViewFactory to do any initialization that must run in the zygote process,
// for memory sharing purposes.
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;
}

```

4.4 SystemServer 启动

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。

```

private static Runnable forkSystemServer(String abiList, String socketName,
    ZygoteServer zygoteServer) {
.....

    /* Hardcoded command line to start the system server */
    //参数准备
    String args[] = {
        "--setuid=1000",
        "--setgid=1000",
.....

        "--target-sdk-version=" + VMRuntime.SDK_VERSION_CUR_DEVELOPMENT,
        "com.android.server.SystemServer",
    };
    ZygoteArguments parsedArgs = null;
    int pid;
    try {
        //将上面准备的参数，按照ZygoteArguments的风格进行封装
        parsedArgs = new ZygoteArguments(args);
        Zygote.applyDebuggerSystemProperty(parsedArgs);
        Zygote.applyInvokeWithSystemProperty(parsedArgs);
    }
}

```

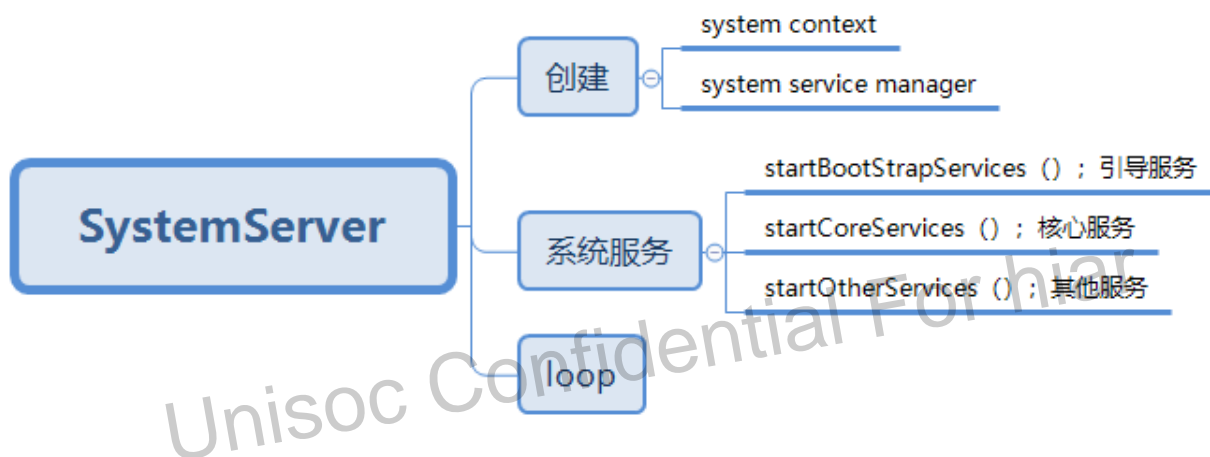
```
boolean profileSystemServer = SystemProperties.getBoolean(
    "dalvik.vm.profilesystemserver", false);
if (profileSystemServer) {
    parsedArgs.mRuntimeFlags |= Zygote.PROFILE_SYSTEM_SERVER;
}
//通过fork"分裂"出子进程system_server
pid = Zygote.forkSystemServer(
    parsedArgs.mUid, parsedArgs.mGid,
    .....,
    parsedArgs.mEffectiveCapabilities);
} catch (IllegalArgumentException ex) {
    throw new RuntimeException(ex);
}
/* For child process */
//进入子进程system_server
if (pid == 0) {
    // 处理32_64和64_32的情况
    if (hasSecondZygote(abiList)) {
        waitForSecondaryZygote(socketName);
    }
    // fork时会copy socket, system server需要主动关闭
    zygoteServer.closeServerSocket();
    // system server进程处理自己的工作
    return handleSystemServerProcess(parsedArgs);
}
return null;
}
```

5 SystemServer 启动流程

zygote 通过 forkSystemServer 进入到 SystemServer 对象的 run 方法中。

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

图5-1 SystemServer



run()方法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、核心服务和其它服务。

```
private void run() {  
    try {  
        traceBeginAndSlog("InitBeforeStartServices");  
  
        // Record the process start information in sys props.  
        //从属性中读取system_server进程的一些信息  
        SystemProperties.set(SYSPROP_START_COUNT, String.valueOf(mStartCount));  
        SystemProperties.set(SYSPROP_START_ELAPSED, String.valueOf(mRuntimeStartElapsedTime));  
        SystemProperties.set(SYSPROP_START_UPTIME, String.valueOf(mRuntimeStartUptime));  
  
        .....  
  
        //如果一个设备的时钟是在1970年之前(0年之前),  
        //那么很多api都会因为处理负数而崩溃，尤其是java.io.File#setLastModified  
        //时间设置为1970  
        if (System.currentTimeMillis() < EARLIEST_SUPPORTED_TIME) {
```

```
Slog.w(TAG, "System clock is before 1970; setting to 1970.");
SystemClock.setCurrentTimeMillis(EARLIEST_SUPPORTED_TIME);
}
//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);

// Ensure binder calls into the system always run at 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looper，即在当前线程运行
android.os.Process.setThreadPriority(
    android.os.Process.THREAD_PRIORITY_FOREGROUND);
android.os.Process.setCanSelfBackground(false);
```

```
Looper.prepareMainLooper();
Looper.getMainLooper().setSlowLogThresholdMs(
    SLOW_DISPATCH_THRESHOLD_MS, SLOW_DELIVERY_THRESHOLD_MS);

//加载android_servers.so库，初始化native service
System.loadLibrary("android_servers");

// Debug builds - allow heap profiling.
//如果是Debug版本，允许堆内存分析
if (Build.IS_DEBUGGABLE) {
    initZygoteChildHeapProfiling();
}

//检测上次关机过程是否失败，这个调用可能不会返回
performPendingShutdown();

//初始化系统上下文
createSystemContext();

//创建系统服务管理--SystemServiceManager
mSystemServiceManager = new SystemServiceManager(mSystemContext);
mSystemServiceManager.setStartInfo(mRuntimeRestart,
    mRuntimeStartElapsedTime, mRuntimeStartUptime);
//将mSystemServiceManager添加到本地服务的成员sLocalServiceObjects
LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);
//为可以并行化的init任务准备线程池
SystemServerInitThreadPool.get();
} finally {
    traceEnd(); // InitBeforeStartServices
}

//启动服务
try {
    traceBeginAndSlog("StartServices");
    startBootstrapServices(); // 启动引导服务
    startCoreServices(); // 启动核心服务
    startOtherServices(); // 启动其他服务
    SystemServerInitThreadPool.shutdown(); //停止线程池
} catch (Throwable ex) {
    Slog.e("System", "*****");
}
```

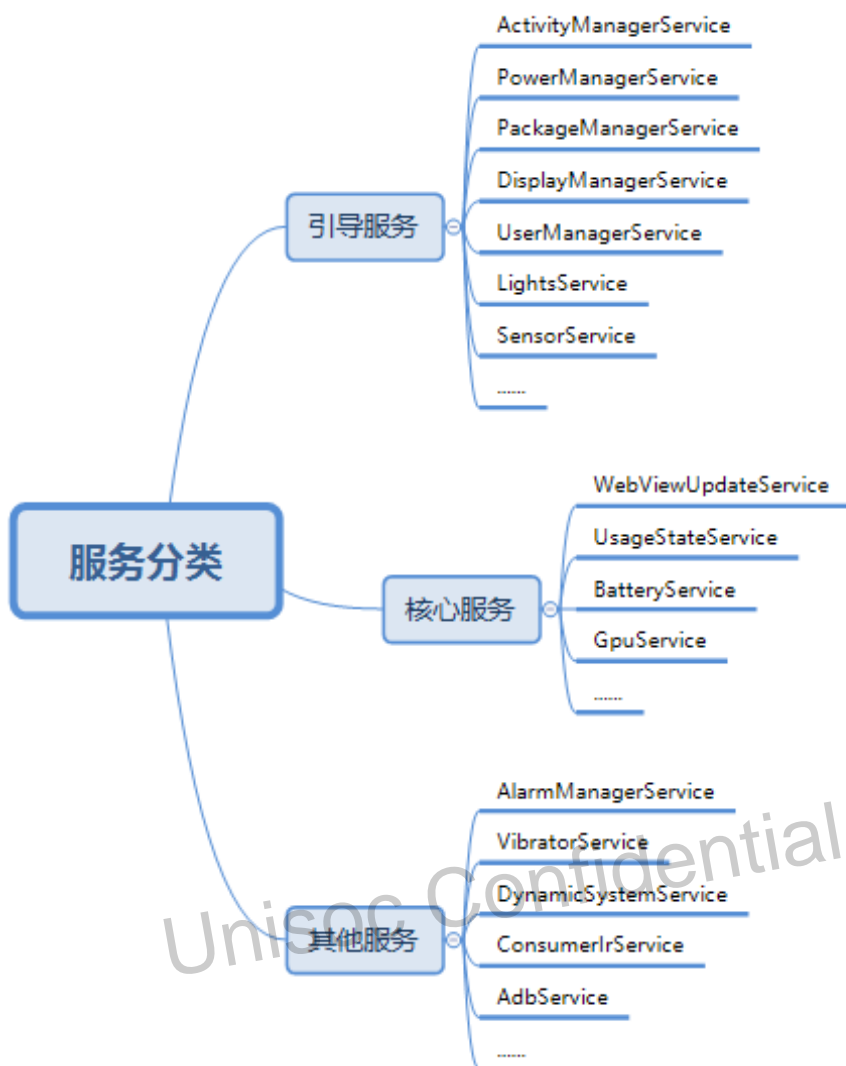
```
Slog.e("System", "***** Failure starting system services", ex);
throw ex;
} finally {
    traceEnd();
}
//为当前的虚拟机初始化VmPolicy
StrictMode.initVmDefaults(null);
.....
//死循环执行
Looper.loop();
throw new RuntimeException("Main thread loop unexpectedly exited");
}
```

服务共 90 多个，分为以下三大类：

- 引导服务(Boot Service)
- 核心服务(Core Service)
- 其它服务(Other Service)

Unisoc Confidential For hiar

图5-2 服务分类



6 AMS 启动流程

AMS（Activity Manager Service）具有管理 Activity 行为、控制 Activity 生命周期、派发消息事件、内存管理等功能。

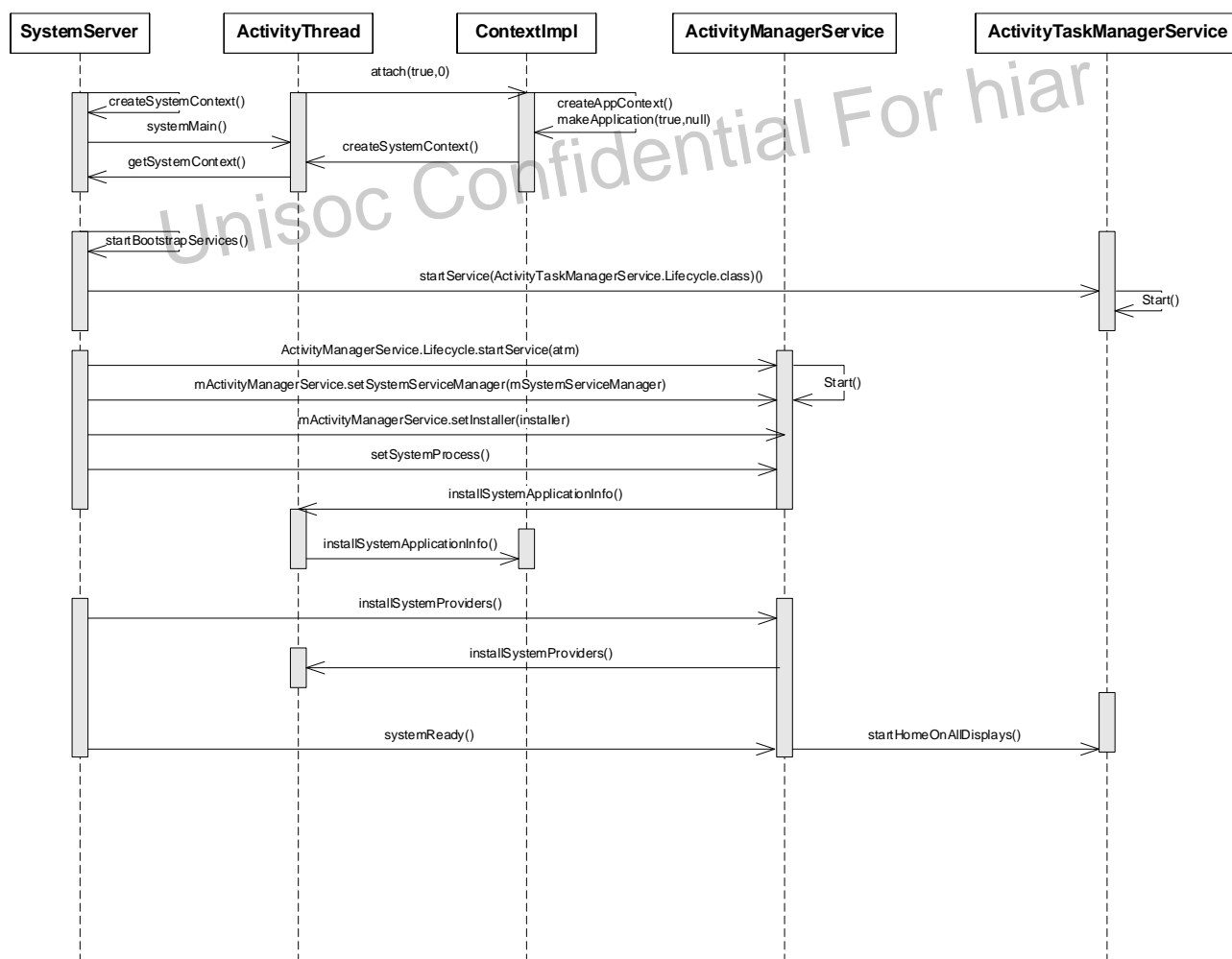
AMS 在 SystemServer 的 startBootstrapServices 中启动，主要是创建了一个 Lifecycle 对象创建 AMS。创建 AMS 后会调用 AMS 的 start 方法。setSystemServiceManager 方法是把 AMS 纳入 SystemServerManager 的管理。

AMS 的构造函数初始化了很多变量和一些服务，如管理广播的队列、电池和 CPU 等相关服务，服务会在 start 方法中启动，并等待启动完成。

最后，调用 AMS 的 systemReady 方法完成初始化，在 SystemReady 中启动桌面。

AMS 启动流程如图 6-1 所示，对具体流程不进行详解。

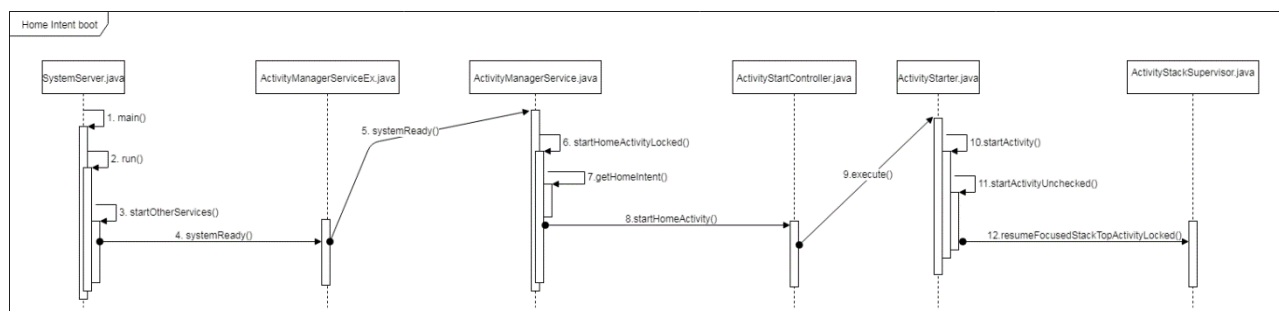
图6-1 AMS 启动流程



7 Launcher 启动流程

Launcher 是由 ActivityManagerService.SystemReady()启动的，Launcher 启动流程如图 7-1 所示。

图7-1 Launcher 启动流程



通过 getHomeIntent()源代码可以看出 getHomeIntent 是如何实现构造 Intent 对象。

源码路径：frameworks/base/services/core/java/com/android/server/wm/ActivityTaskManagerService.java

```
Intent getHomeIntent() {
    Intent intent = new Intent(mTopAction, mTopData != null ? Uri.parse(mTopData) : null);
    intent.setComponent(mTopComponent);
    intent.addFlags(Intent.FLAG_DEBUG_TRIAGED_MISSING);
    if (mFactoryTest != FactoryTest.FACTORY_TEST_LOW_LEVEL) {
        intent.addCategory(Intent.CATEGORY_HOME);
    }
    return intent;
}
```

可以发现，Intent 对象中添加了 Intent.CATEGORY_HOME 常量，这是 Home 应用的标志，一般系统的启动页面 Activity 都会在 manifest.xml 中配置这个标志：

在 Launcher 应用程序中可以找到这个标志：

```
<category android:name="android.intent.category.HOME" />
```

通过 getHomeIntent 来构建一个 category 为 CATEGORY_HOME 的 Intent，表明是 Home Activity；然后通过 resolveHomeActivity()从系统所用已安装的引用中，找到一个符合 HomeIntent 的 Activity，最终调用 startHomeActivity()来启动 Activity。

```
void startHomeActivity(Intent intent, ActivityInfo aInfo, String reason, int displayId) {
    .....
    //返回一个ActivityStarter对象，它负责Activity的启动
    //一系列setXXX()方法传入启动所需的各种参数，最后的execute()是真正的启动逻辑
```

```
//最后执行ActivityStarter的execute方法
mLastHomeActivityResult = obtainStarter(intent, "startHomeActivity: " + reason)
    .setOutActivity(tmpOutRecord)
    .setCallingUid(0)
    .setActivityInfo(aInfo)
    .setActivityOptions(options.toBundle())
    .execute();
mLastHomeActivityResultRecord = tmpOutRecord[0];
final ActivityDisplay display =
    mService.mRootActivityContainer.getActivityDisplay(displayId);
final ActivityStack homeStack = display != null ? display.getHomeStack() : null;
if (homeStack != null && homeStack.mInResumeTopActivity) {
    //如果home activity处于顶层的resume activity中，则Home Activity 将被初始化，但不会被恢复（以避免递归
    //恢复），
    //并将保持这种状态，直到有东西再次触发它。我们需要进行另一次恢复。
    mSupervisor.scheduleResumeTopActivities();
}
}
```

至此声明有 `Intent.CATEGORY_HOME` 的应用（包含 `Launcher`）便会启动。当 `Launcher` 启动时开机动画仍在不断的播放，`Launcher` 会进入 `idle` 状态。此后开机动画便会停止播放，当开机动画结束后，`AMS` 会唤醒栈顶的 `activity`。此时栈顶只有一个 `activity`，那便是 `Launcher`，这些都是 `AMS` 的一些处理流程（本文不进行展开），至此开机启动流程完成。

8 常见问题分析

8.1 无法开机问题

Android 的 log 一般保存在 data 分区，而开机过程出现问题时经常 data 分区还没有挂载好，分析开机过程中的问题（如开机定屏在 logo 界面、定屏在开机动画界面等）一般需要通过串口 log 来分析。

1. 首先检查 log 是否全：查看 kernel log，首先确认时间是不是从[0.000000]开始，若不是说明 log 不全，需要开机抓取串口 log 进行分析。
2. 相关分区是否都 mount 成功。
3. 是否有 kernel panic。
4. 是否有 service 被 kill 或 crash、died 等。
5. 是否有 waiting for xxx service。

找到问题点，请模块 owner 协助分析即可。

8.2 开机慢问题

主要是通过串口 log 或者 kernel log 分析 init 阶段的耗时情况：

1. 首先检查 log 是否全。查看 kernel log，确认时间是不是从[0.000000]开始，若不是说明 log 不全，需要开机抓取串口 log 进行分析。
2. 搜索“took”，确认是否有明显的大耗时情况。
3. 分析 main.log，搜索“Zygote : Preloading resource”，确认 zygote 耗时情况是否异常。

找到问题点，请模块 owner 协助分析即可。