第三章 认识系统组件

在上一章的学习中，我们成功编译了Android12，以及对应的系统内核，并且通过多种方式刷入手机。接下来需要先对Android源码的根结构有一定的了解，了解结构有助于更快地定位和分析源码，同时能让开发人员更好地理解Android系统。在修改系统时，有些简单的功能（例如native中的文件读写、java类型的转换c++类型等）并不需要我们重新实现，因为这些需求大多数在Android系统源码中都有类似的实现，熟练掌握Android系统源码，了解系统中常用的那些功能性函数，可以大大提高定制系统的效率。

在学习系统源码时，碰到问题，要学会暂时记录并跳过，经历过一遍遍学习和实践后，之前遇到的问题可能简单思考便会明白，这不仅节省了时间，也不会在学习过程中逐渐失去信心。

3.1 源码结构介绍

首先看看Android源码根目录下，各个目录的简单介绍。

1. art：该目录是在Android 5.0中新增加的，主要是实现Android RunTime (ART) 的目录，它作为Android 4.4中的Dalvik虚拟机的替代，主要处理Java字节码执行。
2. bionic：Android的C库，包含了很多标准的C库函数和头文件，还有一些Android特有的函数和头文件。
3. build：该目录包含了编译Android源代码所需要的脚本，包括makefile文件和一些构建工具。
4. compatibility：Android设备的兼容性测试套件 (CTS) 和兼容性实现 (Compatibility Implementation) 。
5. cts：Android设备兼容性测试套件 (CTS) ，主要用来测试设备是否符合Android标准。
6. dalvik：Dalvik虚拟机，它是Android 2.3版本之前的主要虚拟机，它主要处理Java字节码执行。
7. developers：Android开发者文档和样例代码。
8. development：调试工具，如systrace、monkey、ddms等。
9. device：特定的Android设备的驱动程序。
10. external：第三方库，如WebKit、OpenGL等。
11. frameworks：Android应用程序调用底层服务的API。
12. hardware：Android设备硬件相关的驱动代码，如摄像头驱动、蓝牙驱动等。
13. kernel：Android系统内核的源代码，它是Android系统的核心部分。
14. libcore：Android底层库，它提供了一些基本的API，如文件系统操作、网络操作等。
15. packages：Android系统中的系统应用程序的源码，例如短信、电话、浏览器、相机等
16. pdk：Android平台开发套件，它包含了一些工具和API，以便开发者快速开发Android应用程序。
17. platform_testing：测试工具，用于测试Android平台的稳定性和性能。
18. prebuilts：预先编译的文件，如编译工具、驱动程序等。
19. sdk：Android SDK的源代码，Android SDK的API文档、代码示例、工具等。
20. system：Android系统的核心部分，如系统服务、应用程序、内存管理机制、文件系统、网络协议等。
21. test：测试代码，用于测试Android系统的各个组件。
22. toolchain：编译器和工具链，如GCC、Clang等，用于编译Android源代码。
23. tools：开发工具，如Android SDK工具、Android Studio、Eclipse等。
24. vendor：硬件厂商提供的驱动程序，如摄像头驱动、蓝牙驱动等。

在上述目录中，并不需要全部记下，只需要记住几个重点即可，例如art、framework、libcore、system、build。在实践时，为了实现功能，查阅翻读源码时，就会不断加深你对这些目录划分的了解。

3.2 Android系统启动流程

Android系统启动主要分为四个阶段：Bootloader阶段、Kernel阶段、Init进程阶段和System Server启动阶段，下面看一下这几个阶段的启动流程。

1. Bootloader阶段： 当手机或平板电脑开机时，首先会执行引导加载程序（Bootloader），它会在手机的ROM中寻找启动内核（Kernel）的镜像文件，并将其加载进RAM。在这个阶段，Android系统并没有完全启动，只是建立了基本的硬件和内核环境。
2. Kernel阶段： Kernel阶段是Android启动的第二阶段，它主要负责初始化硬件设备、加载驱动程序、设置内存管理等。此外，Kernel还会加载initramfs，它是一个临时文件系统，包含了init程序和一些设备文件。
3. Init进程阶段： Kernel会启动init进程，它是Android系统中的第一个用户空间进程。Init进程的主要任务是读取init.rc文件，并根据该文件中的配置信息启动和配置Android系统的各个组件。在这个阶段中，系统会依次启动各个服务和进程，包括启动Zygote进程和创建System Server进程。
4. System Server启动阶段： System Server是Android系统的核心服务进程，它会启动所有的系统服务。其中包括Activity Manager、Package Manager、Window Manager、Location Manager、Telephony Manager、Wi-Fi Service、Bluetooth Service等。System Server启动后，Android系统就完全启动了，用户可以进入桌面，开始使用各种应用程序。

在开始启动流程代码追踪前，最重要的是不要试图了解所有细节过程，分析代码时要抓住需求重点，然后围绕着需求点来进行深入分析。尽管Android源码是一个非常庞大的体系，选择一个方向来熟悉代码，这样就能快速的达成目标，避免深陷代码泥沼。

3.3 内核启动

Bootloader其实是一段程序，这个程序的主要功能就是用来引导系统启动，也称之为引导程序，而这个引导程序是存放在一个只读的寄存器中，从物理地址0开始的一段空间分配给了这个只读存储器来存放引导程序。

Bootloader会初始化硬件设备并准备内存空间映射，为启动内核准备环境。然后寻找内核的镜像文件，验证boot分区和recovery分区的完整性，然后将其加载到内存中，最后开始执行内核。可以通过命令adb reboot bootloader直接重启进入引导程序。

Bootloader初始化完成后，会在特定的物理地址处查找EFI引导头（efi_head）。如果查找到EFI引导头，bootloader就会加载EFI引导头指定的EFI引导程序，然后开始执行EFI引导程序，以进行后续的EFI引导流程。而这个efi_head就是linux内核最早的入口了。

不做系统引导开发的朋友，并不需要完全看懂内核中的汇编部分代码，了解其执行的流程即可，因此不需要读者有汇编的功底，只需要能看懂简单的几个指令即可。打开编译内核源码时的目录，找到文件android-kernel/private/msm-google/arch/arm64/kernel/head.S，查看其汇编代码如下。

```
    __HEAD
_head:
    /*
     * DO NOT MODIFY. Image header expected by Linux boot-loaders.
     */
#ifdef CONFIG_EFI
    /*
     * This add instruction has no meaningful effect except that
     * its opcode forms the magic "MZ" signature required by UEFI.
```

```
     */
    add x13, x18, #0x16
    b   stext
#else
    b   stext               // branch to kernel start, magic
```

在arm指令集中，指令b表示跳转，所以，继续找到stext的定义。

```
    /*
     * The following callee saved general purpose registers are used on the
     * primary lowlevel boot path:
     *
     *  Register   Scope                      Purpose
     *  x21        stext() .. start_kernel()  FDT pointer passed at boot in x0
     *  x23        stext() .. start_kernel()  physical misalignment/KASLR offset
     *  x28        __create_page_tables()     callee preserved temp register
     *  x19/x20    __primary_switch()         callee preserved temp registers
     */
ENTRY(stext)
    bl  preserve_boot_args          // 把引导程序传的4个参数保存在全局数组boot_args
    bl  el2_setup           // Drop to EL1, w0=cpu_boot_mode
    adrp    x23, __PHYS_OFFSET
    and x23, x23, MIN_KIMG_ALIGN - 1    // KASLR offset, defaults to 0
    bl  set_cpu_boot_mode_flag
    bl  __create_page_tables        // 创建页表映射 x25=TTBR0, x26=TTBR1
    /*
     * The following calls CPU setup code, see arch/arm64/mm/proc.S for
     * details.
     * On return, the CPU will be ready for the MMU to be turned on and
     * the TCR will have been set.
     */
    bl  __cpu_setup         // // 初始化处理器 initialise processor
    b   __primary_switch
ENDPROC(stext)
```

能看到最后一行是跳转到__primary_switch，接下来继续看它的实现代码

```
__primary_switch:
#ifdef CONFIG_RANDOMIZE_BASE
    mov x19, x0             // preserve new SCTLR_EL1 value
    mrs x20, sctlr_el1          // preserve old SCTLR_EL1 value
#endif

    bl  __enable_mmu
#ifdef CONFIG_RELOCATABLE
    bl  __relocate_kernel
#ifdef CONFIG_RANDOMIZE_BASE
    ldr x8, =__primary_switched     //将x8设置成__primary_switched的地址
    adrp    x0, __PHYS_OFFSET
```

```
    blr x8                          //调用__primary_switched

    /*
     * If we return here, we have a KASLR displacement in x23 which we need
     * to take into account by discarding the current kernel mapping and
     * creating a new one.
     */
    msr sctlr_el1, x20          // disable the MMU
    isb
    bl  __create_page_tables        // recreate kernel mapping

    tlbi    vmalle1             // Remove any stale TLB entries
    dsb nsh
    isb

    msr sctlr_el1, x19          // re-enable the MMU
    isb
    ic  iallu               // flush instructions fetched
    dsb nsh             // via old mapping
    isb

    bl  __relocate_kernel
#endif
#endif
    ldr x8, =__primary_switched
    adrp    x0, __PHYS_OFFSET
    br  x8
ENDPROC(__primary_switch)
```

继续跟踪`__primary_switched`函数，就能看到调用重点函数`start_kernel`了。

```
__primary_switched:
    adrp    x4, init_thread_union
    add sp, x4, #THREAD_SIZE
    adr_l   x5, init_task
    msr sp_el0, x5          // Save thread_info

    adr_l   x8, vectors         // load VBAR_EL1 with virtual
    msr vbar_el1, x8            // vector table address
    isb

    stp xzr, x30, [sp, #-16]!
    mov x29, sp

#ifdef CONFIG_SHADOW_CALL_STACK
    adr_l   x18, init_shadow_call_stack // Set shadow call stack
#endif

    str_l   x21, __fdt_pointer, x5      // Save FDT pointer

    ldr_l   x4, kimage_vaddr        // Save the offset between
    sub x4, x4, x0          // the kernel virtual and
```

```
    str_l   x4, kimage_voffset, x5      // physical mappings

    // Clear BSS
    adr_l   x0, __bss_start
    mov x1, xzr
    adr_l   x2, __bss_stop
    sub x2, x2, x0
    bl  __pi_memset
    dsb ishst               // Make zero page visible to PTW
#ifdef CONFIG_KASAN
    bl  kasan_early_init
#endif
#ifdef CONFIG_RANDOMIZE_BASE
    tst x23, ~(MIN_KIMG_ALIGN - 1)  // already running randomized?
    b.ne    0f
    mov x0, x21             // pass FDT address in x0
    mov x1, x23             // pass modulo offset in x1
    bl  kaslr_early_init        // parse FDT for KASLR options
    cbz x0, 0f              // KASLR disabled? just proceed
    orr x23, x23, x0            // record KASLR offset
    ldp x29, x30, [sp], #16     // we must enable KASLR, return
    ret                 // to __primary_switch()
0:
#endif
    b   start_kernel        // 内核的入口函数
ENDPROC(__primary_switched)
```

上面能看到最后一个指令就是调用start_kernel了，这个函数是内核的入口函数，同时也是C语言部分的入口函数。接下来，查看文件android-kernel/private/msm-google/init/main.c，可以看到其中大量的init初始化各种子系统的函数调用。

```
asmlinkage __visible void __init start_kernel(void)
{
    // 加载各种子系统
    ...

    /* Do the rest non-__init'ed, we're now alive */
    rest_init();

    prevent_tail_call_optimization();
}
```

继续追踪关键的函数rest_init，在这里开启的内核初始化线程以及创建内核线程。

```
static noinline void __ref rest_init(void)
{
    ...
    kernel_thread(kernel_init, NULL, CLONE_FS);
    numa_default_policy();
```

```
    pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
    ...
}
```

继续看看kernel_init内核初始化线程的实现。

```
static int __ref kernel_init(void *unused)
{
    int ret;
    ...
    if (ramdisk_execute_command) {
        ret = run_init_process(ramdisk_execute_command);
        if (!ret)
            return 0;
        pr_err("Failed to execute %s (error %d)\n",
               ramdisk_execute_command, ret);
    }
}
```

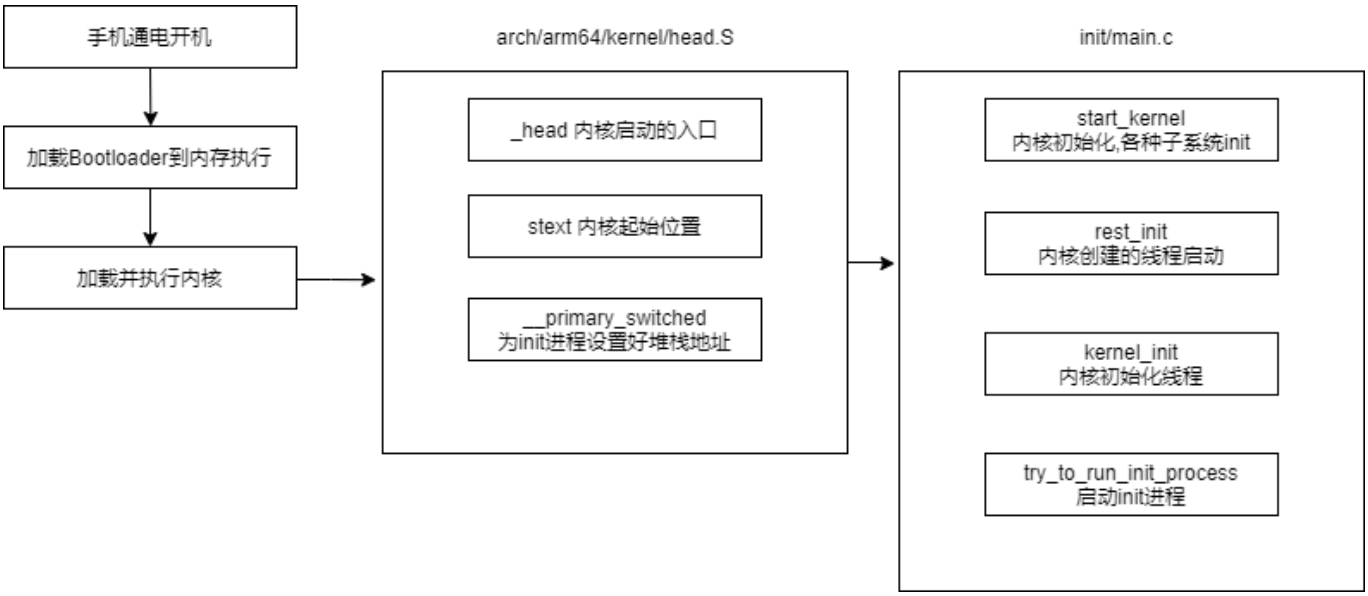
在这里，看到了原来init进程是用run_init_process启动的，ramdisk_execute_command被初始化为了"/init"。

```
static int try_to_run_init_process(const char *init_filename)
{
    int ret;

    ret = run_init_process(init_filename);

    if (ret && ret != -ENOENT) {
        pr_err("Starting init: %s exists but couldn't execute it (error %d)\n",
               init_filename, ret);
    }

    return ret;
}
```

这里简单包装调用的run_init_process，继续看下面的代码

```
static int run_init_process(const char *init_filename)
{
    argv_init[0] = init_filename;
    return do_execve(getname_kernel(init_filename),
        (const char __user *const __user *)argv_init,
        (const char __user *const __user *)envp_init);
}
```

这里能看到最后是通过execve拉起来了系统的第一个进程，init进程。总结内核启动的简单流程图如下。



3.4 Init进程启动

init进程是Android系统的第一个进程，它在系统启动之后就被启动，并且一直运行到系统关闭，它是Android系统的核心进程，隶属于系统进程，具有最高的权限，所有的其他进程都是它的子进程，它的主要功能有以下几点：

1、启动Android系统的基础服务：init进程负责启动Android系统的基础服务。

2、管理系统进程：init进程管理系统进程，比如启动和关闭系统进程。

3、加载设备驱动：init进程会加载设备的驱动，使设备可以正常使用。

4、加载系统环境变量：init进程会加载系统所需要的环境变量，如PATH、LD_LIBRARY_PATH等。

5、加载系统配置文件：init进程会加载系统所需要的配置文件。

6、启动用户进程：init进程会启动用户进程，如桌面程序、默认浏览器等。

init进程的入口是在Android源码的system/core/init/main.cpp。下面，看看入口函数的实现。

```
int main(int argc, char** argv) {
#if __has_feature(address_sanitizer)
    __asan_set_error_report_callback(AsanReportCallback);
#endif
    // Boost prio which will be restored later
    setpriority(PRIO_PROCESS, 0, -20);
    if (!strcmp(basename(argv[0]), "ueventd")) {
        return ueventd_main(argc, argv);
    }

    if (argc > 1) {
        if (!strcmp(argv[1], "subcontext")) {
            android::base::InitLogging(argv, &android::base::KernelLogger);
            const BuiltinFunctionMap& function_map = GetBuiltinFunctionMap();
```

```
            return SubcontextMain(argc, argv, &function_map);
        }
        // 第二步 装载selinux策略
        if (!strcmp(argv[1], "selinux_setup")) {
            return SetupSelinux(argv);
        }
        // 第三步
        if (!strcmp(argv[1], "second_stage")) {
            return SecondStageMain(argc, argv);
        }
    }
    // 第一步 挂载设备节点，初次进入没有参数将执行这里
    return FirstStageMain(argc, argv);
}
```

根据上一章的启动init的参数，可以判断第一次启动时执行的是FirstStageMain函数，继续看看这个函数的实现，可以看到初始化了一些基础系统支持的目录，以及使用mount进行挂载。

```
int FirstStageMain(int argc, char** argv) {
    ...
    CHECKCALL(clearenv());
    CHECKCALL(setenv("PATH", _PATH_DEFPATH, 1));
    // Get the basic filesystem setup we need put together in the initramdisk
    // on / and then we'll let the rc file figure out the rest.
    CHECKCALL(mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755"));
    CHECKCALL(mkdir("/dev/pts", 0755));
    CHECKCALL(mkdir("/dev/socket", 0755));
    CHECKCALL(mkdir("/dev/dm-user", 0755));
    CHECKCALL(mount("devpts", "/dev/pts", "devpts", 0, NULL));
#define MAKE_STR(x) __STRING(x)
    CHECKCALL(mount("proc", "/proc", "proc", 0, "hidepid=2,gid="
MAKE_STR(AID_READPROC)));
#undef MAKE_STR
    // Don't expose the raw commandline to unprivileged processes.
    CHECKCALL(chmod("/proc/cmdline", 0440));
    std::string cmdline;
    android::base::ReadFileToString("/proc/cmdline", &cmdline);
    // Don't expose the raw bootconfig to unprivileged processes.
    chmod("/proc/bootconfig", 0440);
    std::string bootconfig;
    android::base::ReadFileToString("/proc/bootconfig", &bootconfig);
    gid_t groups[] = {AID_READPROC};
    CHECKCALL(setgroups(arraysize(groups), groups));
    CHECKCALL(mount("sysfs", "/sys", "sysfs", 0, NULL));
    CHECKCALL(mount("selinuxfs", "/sys/fs/selinux", "selinuxfs", 0, NULL));
    CHECKCALL(mknod("/dev/kmsg", S_IFCHR | 0600, makedev(1, 11)));
    ...
    // 重新调用拉起init进程，并且参数设置为selinux_setup
    const char* path = "/system/bin/init";
```

```
    const char* args[] = {path, "selinux_setup", nullptr};
    auto fd = open("/dev/kmsg", O_WRONLY | O_CLOEXEC);
    dup2(fd, STDOUT_FILENO);
    dup2(fd, STDERR_FILENO);
    close(fd);
    // 使用execv再次调用init进程
    execv(path, const_cast<char**>(args));

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

    return 1;
}
```

在目录初始化完成后又拉起了一个init进程，并且传入参数selinux_setup，接下来，直接看前面main入口函数中判断出现该参数时调用的SetupSelinux函数。

```
int SetupSelinux(char** argv) {
    SetStdioToDevNull(argv);
    InitKernelLogging(argv);
    ...

    LOG(INFO) << "Opening SELinux policy";

    // Read the policy before potentially killing snapuserd.
    std::string policy;
    ReadPolicy(&policy);

    auto snapuserd_helper = SnapuserdSelinuxHelper::CreateIfNeeded();
    if (snapuserd_helper) {
        // Kill the old snapused to avoid audit messages. After this we cannot
        // read from /system (or other dynamic partitions) until we call
        // FinishTransition().
        snapuserd_helper->StartTransition();
    }

    LoadSelinuxPolicy(policy);

    if (snapuserd_helper) {
        // Before enforcing, finish the pending snapuserd transition.
        snapuserd_helper->FinishTransition();
        snapuserd_helper = nullptr;
    }

    SelinuxSetEnforcement();

    // We're in the kernel domain and want to transition to the init domain.  File
systems that
    // store SELabels in their xattrs, such as ext4 do not need an explicit
restorecon here,
```

```
    // but other file systems do.  In particular, this is needed for ramdisks such
as the
    // recovery image for A/B devices.
    if (selinux_android_restorecon("/system/bin/init", 0) == -1) {
        PLOG(FATAL) << "restorecon failed of /system/bin/init failed";
    }

    setenv(kEnvSelinuxStartedAt,
std::to_string(start_time.time_since_epoch().count()).c_str(), 1);
    // 继续再拉起一个init进程,参数设置second_stage
    const char* path = "/system/bin/init";
    const char* args[] = {path, "second_stage", nullptr};
    execv(path, const_cast<char**>(args));

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

    return 1;
}
```

上面的代码可以看到，在完成selinux的加载处理后，又拉起了一个init进程，并且传入参数second_stage。接下来，看第三步SecondStageMain函数。

```
int SecondStageMain(int argc, char** argv) {
    ...
    // 初始化属性系统
    PropertyInit();

    // 开启属性服务
    StartPropertyService(&property_fd);

    // 解析init.rc 以及启动其他相关进程
    LoadBootScripts(am, sm);

    ...
    return 0;
}
```

继续跟踪LoadBootScripts函数，了解它是如何解析执行init.rc文件的。（修改的意思是否正确？）

```
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()) {
```

```
        // 解析各目录中的init.rc
        parser.ParseConfig("/system/etc/init/hw/init.rc");
        if (!parser.ParseConfig("/system/etc/init")) {
            late_import_paths.emplace_back("/system/etc/init");
        }
        // late_import is available only in Q and earlier release. As we don't
        // have system_ext in those versions, skip late_import for system_ext.
        parser.ParseConfig("/system_ext/etc/init");
        if (!parser.ParseConfig("/vendor/etc/init")) {
            late_import_paths.emplace_back("/vendor/etc/init");
        }
        if (!parser.ParseConfig("/odm/etc/init")) {
            late_import_paths.emplace_back("/odm/etc/init");
        }
        if (!parser.ParseConfig("/product/etc/init")) {
            late_import_paths.emplace_back("/product/etc/init");
        }
    } else {
        parser.ParseConfig(bootscript);
    }
}
```

继续看看解析的逻辑，可以看到参数可以是目录或者文件。

```
bool Parser::ParseConfig(const std::string& path) {
    if (is_dir(path.c_str())) {
        return ParseConfigDir(path);
    }
    return ParseConfigFile(path);
}
```

如果是目录，则遍历所有文件再调用解析文件，所以直接看ParseConfigFile就好了。

```
bool Parser::ParseConfigFile(const std::string& path) {
    ...
    ParseData(path, &config_contents.value());
    ...
}
```

最后看看ParseData是如何解析数据的。

```
void Parser::ParseData(const std::string& filename, std::string* data) {
    ...

    for (;;) {
```

```
        switch (next_token(&state)) {
            case T_EOF:
                ...
                return;
            case T_NEWLINE: {
                ...
                else if (section_parsers_.count(args[0])) {
                    end_section();
                    // 从section_parsers_中获取出来的
                    section_parser = section_parsers_[args[0]].get();
                    section_start_line = state.line;
                    // 使用了ParseSection进行解析
                    if (auto result =
                                section_parser->ParseSection(std::move(args),
filename, state.line);
                        !result.ok()) {
                        parse_error_count_++;
                        LOG(ERROR) << filename << ": " << state.line << ": " <<
result.error();
                        section_parser = nullptr;
                        bad_section_found = true;
                    }
                } else if (section_parser) {
                    // 使用了ParseLineSection进行解析
                    if (auto result = section_parser-
>ParseLineSection(std::move(args), state.line);
                        !result.ok()) {
                        parse_error_count_++;
                        LOG(ERROR) << filename << ": " << state.line << ": " <<
result.error();
                    }
                }
                ...
            }
            case T_TEXT:
                args.emplace_back(state.text);
                break;
        }
    }
}
```

简单解读一下这里的代码，首先这里看到从section_parsers_中取出对应的节点解析对象section_parser，通过section_parser执行ParseSection或者ParseLineSection函数解析.rc文件中的数据。所以需要了解section_parsers_中存储的是什么，查看函数CreateParser就明白了。所谓的节点解析对象，就是ServiceParser、ActionParser、ImportParser。

```
void Parser::AddSectionParser(const std::string& name,
std::unique_ptr<SectionParser> parser) {
    section_parsers_[name] = std::move(parser);
}
```

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

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

    return parser;
}
```

如果了解过init.rc文件格式的，看到这里就很眼熟了，这就是.rc文件中配置时使用的节点名称了。它们的功能的简单描述如下。

1. service 定义一个服务
2. on 触发某个action时，执行对应的指令
3. import 表示导入另外一个rc文件

再解读上面的代码就是，根据rc文件的配置不同，使用ServiceParser、ActionParser、ImportParser这三种节点解析对象的ParseSection或者ParseLineSection函数来处理（这句不完整？）。继续看看这三个对象的解析函数实现。

```
// service节点的解析处理
Result<void> ServiceParser::ParseSection(std::vector<std::string>&& args,
                                         const std::string& filename, int line) {
    if (args.size() < 3) {
        return Error() << "services must have a name and a program";
    }

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

    filename_ = filename;

    Subcontext* restart_action_subcontext = nullptr;
    if (subcontext_ && subcontext_->PathMatchesSubcontext(filename)) {
        restart_action_subcontext = subcontext_;
    }

    std::vector<std::string> str_args(args.begin() + 2, args.end());

    if (SelinuxGetVendorAndroidVersion() <= __ANDROID_API_P__) {
        if (str_args[0] == "/sbin/watchdogd") {
            str_args[0] = "/system/bin/watchdogd";
```

```
        }
    }
    if (SelinuxGetVendorAndroidVersion() <= __ANDROID_API_Q__) {
        if (str_args[0] == "/charger") {
            str_args[0] = "/system/bin/charger";
        }
    }

    service_ = std::make_unique<Service>(name, restart_action_subcontext,
str_args, from_apex_);
    return {};
}


// on 节点的解析处理
Result<void> ActionParser::ParseSection(std::vector<std::string>&& args,
                                        const std::string& filename, int line) {
    std::vector<std::string> triggers(args.begin() + 1, args.end());
    if (triggers.size() < 1) {
        return Error() << "Actions must have a trigger";
    }

    Subcontext* action_subcontext = nullptr;
    if (subcontext_ && subcontext_->PathMatchesSubcontext(filename)) {
        action_subcontext = subcontext_;
    }

    std::string event_trigger;
    std::map<std::string, std::string> property_triggers;

    if (auto result =
                ParseTriggers(triggers, action_subcontext, &event_trigger,
&property_triggers);
        !result.ok()) {
        return Error() << "ParseTriggers() failed: " << result.error();
    }

    auto action = std::make_unique<Action>(false, action_subcontext, filename,
line, event_trigger,
                                           property_triggers);

    action_ = std::move(action);
    return {};
}

// import节点的解析处理
Result<void> ImportParser::ParseSection(std::vector<std::string>&& args,
                                        const std::string& filename, int line) {
    if (args.size() != 2) {
        return Error() << "single argument needed for import\n";
    }

    auto conf_file = ExpandProps(args[1]);
    if (!conf_file.ok()) {
```

```
        return Error() << "Could not expand import: " << conf_file.error();
    }

    LOG(INFO) << "Added '" << *conf_file << "' to import list";
    if (filename_.empty()) filename_ = filename;
    imports_.emplace_back(std::move(*conf_file), line);
    return {};
}
```

到这里大致的init进程的启动流程相信大家已经有了一定了解。明白init的原理后，对于init.rc相信大家已经有了简单的印象，接下来将详细展开讲解init.rc文件。

3.5 init.rc

init.rc是Android系统中的一个脚本文件而并非配置文件，是一种名为Android Init Language的脚本语言写成的文件，当然也可以简单当作配置文件来理解，主要用于启动和管理Android上的其他进程以对系统进行初始化工作。

将init.rc看作是init进程功能的动态延申，一些可能需要改动的初始化系统任务就放在配置文件中，然后读取配置解析后再进行初始化执行，如此可以提高一定的灵活性，相信很多开发人员在工作中都有做过类似的封装。而init.rc就是配置文件的入口，在init.rc中通过import节点来导入其他的配置文件，所以这些文件都可以算是init.rc的一部分。在上一章（确定是章？），通过了解init进程的工作流程，明白了解析init.rc文件的过程。

init.rc是由多个section节点组成的，而节点的类型分别主要是service、on、import三种。上一节中，有简单的介绍，它们的作用分别是定义服务、事件触发、导入其他rc文件。下面，来看init.rc文件中的几个例子，查看文件system/core/rootdir/init.rc。

```
// 导入另一个rc文件
import /init.environ.rc
import /system/etc/init/hw/init.usb.rc
import /init.${ro.hardware}.rc
import /vendor/etc/init/hw/init.${ro.hardware}.rc
import /system/etc/init/hw/init.usb.configfs.rc
import /system/etc/init/hw/init.${ro.zygote}.rc
...
// 当初始化触发时,执行section下的命令
on init
    sysclktz 0

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

    symlink /proc/self/fd/0 /dev/stdin
    symlink /proc/self/fd/1 /dev/stdout
    symlink /proc/self/fd/2 /dev/stderr

    # Create energy-aware scheduler tuning nodes
```

```
    mkdir /dev/stune/foreground
...

// 当属性ro.debuggable变更为1时触发section内的命令
on property:ro.debuggable=1
    # Give writes to anyone for the trace folder on debug builds.
    # The folder is used to store method traces.
    chmod 0773 /data/misc/trace
    # Give reads to anyone for the window trace folder on debug builds.
    chmod 0775 /data/misc/wmtrace
    # Give reads to anyone for the accessibility trace folder on debug builds.
    chmod 0775 /data/misc/a11ytrace
...

// 定义系统服务 服务名称ueventd  服务路径/system/bin/ueventd
// 服务类型core，关机行为critical，安全标签u:r:ueventd:s0
service ueventd /system/bin/ueventd
    class core
    critical
    seclabel u:r:ueventd:s0
    shutdown critical

// 定义系统服务 服务名称console 服务路径/system/bin/sh
// 服务类型core 服务状态disabled 服务所属用户shell 服务所属组shell log readproc
// 安全标签u:r:shell:s0 设置环境变量HOSTNAME console
service console /system/bin/sh
    class core
    console
    disabled
    user shell
    group shell log readproc
    seclabel u:r:shell:s0
    setenv HOSTNAME console
```

看完各种节点的样例后，大概了解init.rc中应该如何添加一个section了。import非常简单，只需要指定一个rc文件的路径即可。on节点在源码中，看到对应的处理是ActionParser，这个节点就是当触发了一个Action的事件后就自上而下，依次执行节点下的所有命令，所以，就得了解一下一共有哪些Action事件提供使用。详细介绍参考自http://www.gaohaiyan.com/4047.html

```
on boot                         #系统启动触发
on early-init                   #在初始化之前触发
on init                         #在初始化时触发（在启动配置文件/init.conf被装载之后）
on late-init                    #在初始化晚期阶段触发
on charger                      #当充电时触发
on property:<key>=<value>       #当属性值满足条件时触发
on post-fs                      #挂载文件系统
on post-fs-data                 #挂载data
on device-added-<path>          #在指定设备被添加时触发
on device-removed-<path>        #在指定设备被移除时触发
on service-exited-<name>        #在指定service退出时触发
on <name>=<value>               #当属性<name>等于<value>时触发
```

在触发Action事件后可以执行的命令如下。

```
chdir <dirc>                                    更改工作目录为<dirc>
chmod <octal-mode> <path>                       更改文件访问权限
chown <owner> <group> <path>                    更改文件所有者和组群
chroot <direc>                                  更改根目录位置
class_start <serviceclass>                      如果它们不在运行状态的话，启动由
<serviceclass>类名指定的所有相关服务
class_stop <serviceclass>                       如果它们在运行状态的话，停止
domainname <name>                               设置域名
exec <path> [ <argument> ]*                     fork并执行一个程序，其路径为<path>，这
条命令将阻塞直到该程序启动完成，因此它有可能造成init程序在某个节点不停地等待
export <name> <value>                           设置某个环境变量<name>的值为<value>，
这是对全局有效的，即其后所有进程都将继承这个变量
hostname <name>                                 设置主机名
ifup <interface>                                使网络接口<interface>成功连接
import <filename>                               引入一个名为<filename>的文件
insmod <path>                                   在<path>路径上安装一个模块
mkdir <path> [mode] [owner] [group]             在<path>路径上新建一个目录
mount <type> <device> <dir> [<mountoption>]*    尝试在指定路径上挂载一个设备
setprop <name> <value>                          设置系统属性<name>的值为<value>
setrlinit <resource> <cur> <max>                设置一种资源的使用限制。这个概念亦存在
于Linux系统中，<cur>表示软限制，<max>表示硬限制
start <service>                                 启动一个服务
stop <service>                                  停止一个服务
symlink <target> <path>                         创建一个<path>路径的软链接，目标为
<target>
sysclk <mins_west_of_gmt>                       设置基准时间，如果当前时间时GMT，这个值
是0
trigger <event>                                 触发一个事件
write <path> <string> [<string>]*               打开一个文件，并写入字符串
```

而service节点主要是将可执行程序作为服务启动，上面的例子，看到节点下面有一系列的参数，下面是这些参数的详细描述。

```
class <name>                                    为该服务指定一个class名，同一个
class的所有服务必须同时启动或者停止。
                                                默认情况下服务的class名是
“default”。另外还有core(其它服务依赖的基础性核心服务)、main(java须要的基本服务)、
late_start(厂商定制的服务)
critical                                        表示这是一个对设备至关重要的一个服
务，如果它在四分钟内退出超过四次，则设备将重启进入恢复模式
disabled                                        此服务不会自动启动，而是需要通过显
式调用服务名来启动
group <groupname> [<groupname>]*                在启动服务前将用户组切换为
<groupname>
oneshot                                         只启动一次，当此服务退出时，不要主
动去重启它
onrestart                                       当此服务重启时，执行某些命令
```

```
setenv <name> <value>                                    设置环境变量<name>为某个值
<value>
socket <name> <type> <perm> [ <user> [<group>]]    创建一个名为/dev/socket/<name>的
unix domain socket，然后将它的fd值传给启动它的进程，有效的<type>值包括dgram，stream和
seqacket，而user和group的默认值是0
user <username>                                          在启动服务前将用户组切换为
<username>，默认情况下用户都是root
```

到这里，相信大家应该能够看懂init.rc中的大多数section的含义了。下面的例子将组合使用，定义一个自己的服务，并且启动它。

```
service kservice /system/bin/app_process -
Djava.class.path=/system/framework/ksvr.jar /system/bin cn.ksvr.kSystemSvr svr
    class main
    user root
    group root
    oneshot
    seclabel u:r:su:s0

on property:sys.boot_completed=1
    bootchart stop
    start kservice
```

上面的案例中，我定义了一个kservice的服务，使用/system/bin/app_process作为进程启动，并设置目标jar作为应用的classpath，最后设置jar文件的入口类cn.ksvr.kSystemSvr，最后的svr是做为参数传递给kSystemSvr中的main函数。接下来是当属性sys.boot_completed变更为1时表示手机完成引导，执行节点下的命令启动刚刚定义的服务。

3.6 Zygote启动

了解init.rc定义的原理后，就可以继续阅读init.rc追踪后续的启动流程了。

```
# 导入含有zygote服务定义的rc文件，这个会根据系统所支持的对应架构导入
import /system/etc/init/hw/init.${ro.zygote}.rc

# init完成后触发zygote-start事件
on late-init
    ...
    # Now we can start zygote for devices with file based encryption
    trigger zygote-start
    ...

# zygote-start事件触发时执行的节点。最后启动了zygote和zygote_secondary
on zygote-start && property:ro.crypto.state=unencrypted
    wait_for_prop odsign.verification.done 1
    # A/B update verifier that marks a successful boot.
    exec_start update_verifier_nonencrypted
    start statsd
    start netd
```

```
    start zygote
    start zygote_secondary
```

zygote服务定义的rc文件在路径system/core/rootdir/中。分别是init.zygote32.rc、init.zygote64.rc、init.zygote32_64.rc、init.zygote64_32.rc，下面查看zygote64的是如何定义的。

```
// --zygote 传递给app_process程序的参数,表示这是启动一个孵化器。
// --start-system-server 传递给app_process程序的参数，表示进程启动后需要启动
system_server进程
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 exec_background - system system -- /system/bin/vdc volume abort_fuse
    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
    critical window=${zygote.critical_window.minute:-off} target=zygote-fatal
```

从定义中可以看到zygote进程实际启动的就是app_process进程。

app_process是Android系统的主要进程，它是其他所有应用程序的容器，它负责创建新的进程，并启动它们。此外，它还管理应用程序的生命周期，防止任何一个应用程序占用资源过多，或者做出不良影响。app_process还负责在应用运行时为它们提供上下文，以及管理应用进程之间的通信。

跟踪app_process的实现，它的入口是在目录frameworks/base/cmds/app_process/app_main.cpp中。

```
#if defined(__LP64__)
static const char ABI_LIST_PROPERTY[] = "ro.product.cpu.abilist64";
static const char ZYGOTE_NICE_NAME[] = "zygote64";
#else
static const char ABI_LIST_PROPERTY[] = "ro.product.cpu.abilist32";
static const char ZYGOTE_NICE_NAME[] = "zygote";
#endif

int main(int argc, char* const argv[])
{
    ...
    // Parse runtime arguments.  Stop at first unrecognized option.
```

```
    bool zygote = false;
    bool startSystemServer = false;
    bool application = false;
    String8 niceName;
    String8 className;

    ++i;  // Skip unused "parent dir" argument.
    // 参数的处理
    while (i < argc) {
        const char* arg = argv[i++];
        if (strcmp(arg, "--zygote") == 0) {
            zygote = true;
            niceName = ZYGOTE_NICE_NAME;
        } else if (strcmp(arg, "--start-system-server") == 0) {
            startSystemServer = true;
        } else if (strcmp(arg, "--application") == 0) {
            application = true;
        } else if (strncmp(arg, "--nice-name=", 12) == 0) {
            niceName.setTo(arg + 12);
        } else if (strncmp(arg, "--", 2) != 0) {
            className.setTo(arg);
            break;
        } else {
            --i;
            break;
        }
    }
    ...
    if (!niceName.isEmpty()) {
        runtime.setArgv0(niceName.string(), true /* setProcName */);
    }
    // 如果启动时设置--zygote，则启动ZygoteInit，否则启动RuntimeInit
    if (zygote) {
        const char* zygoteName="com.android.internal.os.ZygoteInit";
        runtime.start(zygoteName, args, zygote);
    } else if (className) {
        const char* zygoteName="com.android.internal.os.RuntimeInit";
        runtime.start(zygoteName, args, zygote);
    } else {
        fprintf(stderr, "Error: no class name or --zygote supplied.\n");
        app_usage();
        LOG_ALWAYS_FATAL("app_process: no class name or --zygote supplied.");
    }
}
```

从代码中可以看到主要是对参数进行处理包装后，然后根据是否携带`--zygote`选择启动`ZygoteInit`或者是`RuntimeInit`。

`ZygoteInit`负责加载和初始化`Android`运行时环境，例如应用程序运行器、垃圾收集器等，并且它启动`Android`系统中的所有核心服务。

RuntimeInit负责联系应用程序的执行环境与系统的运行环境，然后将应用程序的主类加载到运行时，最后将应用程序的控制权交给应用程序的主类。

下面继续看看runtime.start的实现，查看对应文件frameworks/base/core/jni/AndroidRuntime.cpp

```
void AndroidRuntime::start(const char* className, const Vector<String8>& options,
bool zygote)
{
    ...
    //const char* kernelHack = getenv("LD_ASSUME_KERNEL");
    //ALOGD("Found LD_ASSUME_KERNEL='%s'\n", kernelHack);

    /* 启动vm虚拟机 */
    JniInvocation jni_invocation;
    jni_invocation.Init(NULL);
    JNIEnv* env;
    if (startVm(&mJavaVM, &env, zygote, primary_zygote) != 0) {
        return;
    }
    onVmCreated(env);

    /*
     * 注册框架使用的JNI调用
     */
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }
    ...
    char* slashClassName = toSlashClassName(className != NULL ? className : "");
    jclass startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
        /* keep going */
    } else {
        // 这里调用ZygoteInit或者是RuntimeInit的main函数
        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 {
            env->CallStaticVoidMethod(startClass, startMeth, strArray);

#if 0
            if (env->ExceptionCheck())
                threadExitUncaughtException(env);
#endif
        }
    }
    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");
}
```

通过JNI函数CallStaticVoidMethod调用了ZygoteInit的main入口函数，现在就来到了java层中，查看文件代码frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

```
public static void main(String[] argv) {
        ZygoteServer zygoteServer = null;
        ...
        try {
            ...
            if (!enableLazyPreload) {
                bootTimingsTraceLog.traceBegin("ZygotePreload");
                EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
                        SystemClock.uptimeMillis());
                // 预加载资源，比如类、主题资源、字体资源等等
                preload(bootTimingsTraceLog);
                EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
                        SystemClock.uptimeMillis());
                bootTimingsTraceLog.traceEnd(); // ZygotePreload
            }
            ...

            Zygote.initNativeState(isPrimaryZygote);

            ZygoteHooks.stopZygoteNoThreadCreation();
            // 创建socket服务端
            zygoteServer = new ZygoteServer(isPrimaryZygote);
            // 前面在init.rc中有配置--start-system-server的进程则会进入fork启动
SystemServer
            if (startSystemServer) {
                Runnable r = forkSystemServer(abiList, zygoteSocketName,
zygoteServer);

                // {@code r == null} in the parent (zygote) process, and {@code r
!= null} in the
                // child (system_server) process.
                if (r != null) {
                    r.run();
                    return;
                }
            }
            Log.i(TAG, "Accepting command socket connections");
            // socket服务端等待AMS的请求，收到请求后就会由Zygote服务端来通过fork创建应
用程序的进程
            caller = zygoteServer.runSelectLoop(abiList);
        } catch (Throwable ex) {
```

```
            Log.e(TAG, "System zygote died with fatal exception", ex);
            throw ex;
        } finally {
            if (zygoteServer != null) {
                zygoteServer.closeServerSocket();
            }
        }

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

这里的重点是创建了zygoteServer，然后根据参数决定是否forkSystemServer，最后runSelectLoop等待AMS发送消息创建应用程序的进程。依次从代码观察它们的本质。首先是ZygoteServer的构造函数，可以看到，主要是创建Socket套接字。

```
ZygoteServer(boolean isPrimaryZygote) {
        mUsapPoolEventFD = Zygote.getUsapPoolEventFD();

        if (isPrimaryZygote) {
            mZygoteSocket =
Zygote.createManagedSocketFromInitSocket(Zygote.PRIMARY_SOCKET_NAME);
            mUsapPoolSocket =
                    Zygote.createManagedSocketFromInitSocket(
                            Zygote.USAP_POOL_PRIMARY_SOCKET_NAME);
        } else {
            mZygoteSocket =
Zygote.createManagedSocketFromInitSocket(Zygote.SECONDARY_SOCKET_NAME);
            mUsapPoolSocket =
                    Zygote.createManagedSocketFromInitSocket(
                            Zygote.USAP_POOL_SECONDARY_SOCKET_NAME);
        }

        mUsapPoolSupported = true;
        fetchUsapPoolPolicyProps();
    }
```

接着分析forkSystemServer，目的是了解返回值到底是什么，返回值的r.run()会调用到哪里。

```
private static Runnable forkSystemServer(String abiList, String socketName,
            ZygoteServer zygoteServer) {
        // 服务启动的相关参数，这里注意到类名是com.android.server.SystemServer
        String[] args = {
                "--setuid=1000",
                "--setgid=1000",
```

```
                "--
setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,1021,1023,"
                        +
"1024,1032,1065,3001,3002,3003,3006,3007,3009,3010,3011",
                "--capabilities=" + capabilities + "," + capabilities,
                "--nice-name=system_server",
                "--runtime-args",
                "--target-sdk-version=" + VMRuntime.SDK_VERSION_CUR_DEVELOPMENT,
                "com.android.server.SystemServer",
        };
        ZygoteArguments parsedArgs;

        int pid;

        try {
            ...
            // 使用fork创建一个SystemServer进程
            /* Request to fork the system server process */
            pid = Zygote.forkSystemServer(
                    parsedArgs.mUid, parsedArgs.mGid,
                    parsedArgs.mGids,
                    parsedArgs.mRuntimeFlags,
                    null,
                    parsedArgs.mPermittedCapabilities,
                    parsedArgs.mEffectiveCapabilities);
        } catch (IllegalArgumentException ex) {
            throw new RuntimeException(ex);
        }

        /* For child process */
        if (pid == 0) {
            if (hasSecondZygote(abiList)) {
                waitForSecondaryZygote(socketName);
            }

            zygoteServer.closeServerSocket();
            // pid为0的部分，就是由这里fork出来的SystemServer执行的了。
            return handleSystemServerProcess(parsedArgs);
        }

        return null;
    }


private static Runnable handleSystemServerProcess(ZygoteArguments parsedArgs) {
    ...
    ClassLoader cl = getOrCreateSystemServerClassLoader();
    if (cl != null) {
        Thread.currentThread().setContextClassLoader(cl);
    }
    // 初始化SystemServer
    return ZygoteInit.zygoteInit(parsedArgs.mTargetSdkVersion,
                                 parsedArgs.mDisabledCompatChanges,
                                 parsedArgs.mRemainingArgs, cl);
```

```
    ...
}

public static Runnable zygoteInit(int targetSdkVersion, long[]
disabledCompatChanges,
            String[] argv, ClassLoader classLoader) {
        ...
        // 继续跟进去
        return RuntimeInit.applicationInit(targetSdkVersion,
disabledCompatChanges, argv,
                classLoader);
    }


protected static Runnable applicationInit(int targetSdkVersion, long[]
disabledCompatChanges,
                                          String[] argv, ClassLoader classLoader)
{
    ...
    // 反射获取com.android.server.SystemServer的入口函数并返回
    return findStaticMain(args.startClass, args.startArgs, classLoader);
}

// 可以看到就是通过反射，获取到对应类的main函数，最后封装到MethodAndArgsCaller返回
protected static Runnable findStaticMain(String className, String[] argv,
                                         ClassLoader classLoader) {
    Class<?> cl;

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

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

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

    /*
```

```
             * This throw gets caught in ZygoteInit.main(), which responds
             * by invoking the exception's run() method. This arrangement
             * clears up all the stack frames that were required in setting
             * up the process.
             */
        return new MethodAndArgsCaller(m, argv);
    }

    // forkSystemServer最终返回的就是MethodAndArgsCaller对象
    static class MethodAndArgsCaller implements Runnable {
        /** method to call */
        private final Method mMethod;

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

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

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

forkSystemServer函数走到最后是通过反射获取com.android.server.SystemServer的入口函数main，并封装到MethodAndArgsCaller对象中返回。最后的返回结果调用run时，就会执行到SystemServer中的main函数。继续看看main函数的实现，查看文件 frameworks/base/services/java/com/android/server/SystemServer.java

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

    private void run() {
            ...
```

```
        // 创建主线程Looper
        Looper.prepareMainLooper();

        // 初始化系统Context上下文
        createSystemContext();

        // 创建SystemServiceManager，由它管理系统的所有服务
        mSystemServiceManager = new SystemServiceManager(mSystemContext);
        mSystemServiceManager.setStartInfo(mRuntimeRestart,
                                           mRuntimeStartElapsedTime,
mRuntimeStartUptime);
        mDumper.addDumpable(mSystemServiceManager);
        LocalServices.addService(SystemServiceManager.class,
mSystemServiceManager);
        ...

    // 启动各种服务
    try {
        t.traceBegin("StartServices");
        // 启动引导服务
        startBootstrapServices(t);
        // 启动核心服务
        startCoreServices(t);
        // 启动其他服务
        startOtherServices(t);
    } catch (Throwable ex) {
        Slog.e("System", "******************************************");
        Slog.e("System", "************ Failure starting system services", ex);
        throw ex;
    } finally {
        t.traceEnd(); // StartServices
    }
    ...
    // Loop forever.
    Looper.loop();
}

// 启动负责引导的服务
private void startBootstrapServices(@NonNull TimingsTraceAndSlog t) {
    t.traceBegin("startBootstrapServices");

    ...
    // 启动ActivityManagerService
    t.traceBegin("StartActivityManager");
    ActivityTaskManagerService atm = mSystemServiceManager.startService(
        ActivityTaskManagerService.Lifecycle.class).getService();
    mActivityManagerService = ActivityManagerService.Lifecycle.startService(
        mSystemServiceManager, atm);
    mActivityManagerService.setSystemServiceManager(mSystemServiceManager);
    mActivityManagerService.setInstaller(installer);
    mWindowManagerGlobalLock = atm.getGlobalLock();
    t.traceEnd();
    ...
}
```

```
private void startOtherServices(@NonNull TimingsTraceAndSlog t) {
    t.traceBegin("startOtherServices");
    ...
    // 从systemReady开始可以启动第三方应用
    mActivityManagerService.systemReady(() -> {
        ...
    }, t);
    ...
}

// 最后看看systemReady的处理
// frameworks/base/services/java/com/android/server/am/ActivitymanagerService.java
public void systemReady(final Runnable goingCallback, @NonNull TimingsTraceAndSlog
t) {
        ...
        Slog.i(TAG, "System now ready");
        ...
        // 启动多用户下的Home Activity，最终会开启系统应用Luncher桌面显示
        if (bootingSystemUser) {
            t.traceBegin("startHomeOnAllDisplays");
            mAtmInternal.startHomeOnAllDisplays(currentUserId, "systemReady");
            t.traceEnd();
        }
        ...
}
```

到这里大致的服务启动流程就清楚了，最后成功抵达了`Luncher`的启动，重新回到流程中，继续看看`runSelectLoop`函数是如何实现的。

```
Runnable runSelectLoop(String abiList) {
        ...
        socketFDs.add(mZygoteSocket.getFileDescriptor());
        peers.add(null);

        mUsapPoolRefillTriggerTimestamp = INVALID_TIMESTAMP;

        while (true) {
            fetchUsapPoolPolicyPropsWithMinInterval();
            mUsapPoolRefillAction = UsapPoolRefillAction.NONE;

            int[] usapPipeFDs = null;
            StructPollfd[] pollFDs;

            int pollReturnValue;
            try {
                pollReturnValue = Os.poll(pollFDs, pollTimeoutMs);
            } catch (ErrnoException ex) {
                throw new RuntimeException("poll failed", ex);
            }
```

```
            ...
            if (mUsapPoolRefillAction != UsapPoolRefillAction.NONE) {
                int[] sessionSocketRawFDs =
                        socketFDs.subList(1, socketFDs.size())
                                .stream()
                                .mapToInt(FileDescriptor::getInt$)
                                .toArray();

                final boolean isPriorityRefill =
                        mUsapPoolRefillAction == UsapPoolRefillAction.IMMEDIATE;

                final Runnable command =
                        fillUsapPool(sessionSocketRawFDs, isPriorityRefill);

                if (command != null) {
                    return command;
                } else if (isPriorityRefill) {
                    // Schedule a delayed refill to finish refilling the pool.
                    mUsapPoolRefillTriggerTimestamp = System.currentTimeMillis();
                }
            }
        }
    }
```

重点主要放在返回值的跟踪上，直接看`fillUsapPool`函数做了些什么

```
Runnable fillUsapPool(int[] sessionSocketRawFDs, boolean isPriorityRefill) {
        ...
        while (--numUsapsToSpawn >= 0) {
            Runnable caller =
                    Zygote.forkUsap(mUsapPoolSocket, sessionSocketRawFDs,
isPriorityRefill);

            if (caller != null) {
                return caller;
            }
        }
        ...
        return null;
    }

// 继续追踪关键返回值的函数forkUsap
// 对应文件frameworks/base/core/java/com/android/internal/os/Zygote.java
static @Nullable Runnable forkUsap(LocalServerSocket usapPoolSocket,
                                   int[] sessionSocketRawFDs,
                                   boolean isPriorityFork) {
        FileDescriptor readFD;
        FileDescriptor writeFD;

        try {
            FileDescriptor[] pipeFDs = Os.pipe2(O_CLOEXEC);
            readFD = pipeFDs[0];
```

```
            writeFD = pipeFDs[1];
        } catch (ErrnoException errnoEx) {
            throw new IllegalStateException("Unable to create USAP pipe.",
errnoEx);
        }
        // 这里fork出一个子进程并初始化信息，最后返回pid
        int pid = nativeForkApp(readFD.getInt$(), writeFD.getInt$(),
                                sessionSocketRawFDs, /*argsKnown=*/ false,
isPriorityFork);
        if (pid == 0) {
            IoUtils.closeQuietly(readFD);
            // 如果是子进程就调用childMain获取返回值
            return childMain(null, usapPoolSocket, writeFD);
        } else if (pid == -1) {
            // Fork failed.
            return null;
        } else {
            // readFD will be closed by the native code. See
removeUsapTableEntry();
            IoUtils.closeQuietly(writeFD);
            nativeAddUsapTableEntry(pid, readFD.getInt$());
            return null;
        }
    }

// 继续看childMain的实现
private static Runnable childMain(@Nullable ZygoteCommandBuffer argBuffer,
                                  @Nullable LocalServerSocket usapPoolSocket,
                                  FileDescriptor writePipe) {
        ...
        // 初始化应用程序环境，设置应用程序上下文，初始化应用程序线程等等
        specializeAppProcess(args.mUid, args.mGid, args.mGids,
                             args.mRuntimeFlags, rlimits, args.mMountExternal,
                             args.mSeInfo, args.mNiceName, args.mStartChildZygote,
                             args.mInstructionSet, args.mAppDataDir,
args.mIsTopApp,
                             args.mPkgDataInfoList, args.mAllowlistedDataInfoList,
                             args.mBindMountAppDataDirs,
args.mBindMountAppStorageDirs);

        Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
        // 又看到这个了，在SystemServer的启动中，之前追踪过
        // 这里最后是反射获取某个java类的main函数封装后返回
        return ZygoteInit.zygoteInit(args.mTargetSdkVersion,
                                     args.mDisabledCompatChanges,
                                     args.mRemainingArgs,
                                     null /* classLoader */);

    }
```

前面分析过了`zygoteInit`函数，所以这里就不需要再继续进去看了，看看孵化器进程是如何初始化应用程序环境的，追踪`specializeAppProcess`函数。

```
private static void specializeAppProcess(int uid, int gid, int[] gids, int
runtimeFlags,
            int[][] rlimits, int mountExternal, String seInfo, String niceName,
            boolean startChildZygote, String instructionSet, String appDataDir,
boolean isTopApp,
            String[] pkgDataInfoList, String[] allowlistedDataInfoList,
            boolean bindMountAppDataDirs, boolean bindMountAppStorageDirs) {

        // 参数传递到了native层进行初始化处理了。
        nativeSpecializeAppProcess(uid, gid, gids, runtimeFlags, rlimits,
mountExternal, seInfo,
                niceName, startChildZygote, instructionSet, appDataDir, isTopApp,
                pkgDataInfoList, allowlistedDataInfoList,
                bindMountAppDataDirs, bindMountAppStorageDirs);
        ...
    }

// 继续查看nativeSpecializeAppProcess
// 文件所在frameworks/base/core/jni/com_android_internal_os_Zygote.cpp
static void com_android_internal_os_Zygote_nativeSpecializeAppProcess(
        JNIEnv* env, jclass, jint uid, jint gid, jintArray gids, jint
runtime_flags,
        jobjectArray rlimits, jint mount_external, jstring se_info, jstring
nice_name,
        jboolean is_child_zygote, jstring instruction_set, jstring app_data_dir,
        jboolean is_top_app, jobjectArray pkg_data_info_list,
        jobjectArray allowlisted_data_info_list, jboolean mount_data_dirs,
        jboolean mount_storage_dirs) {
    jlong capabilities = CalculateCapabilities(env, uid, gid, gids,
is_child_zygote);

    SpecializeCommon(env, uid, gid, gids, runtime_flags, rlimits, capabilities,
capabilities,
                     mount_external, se_info, nice_name, false, is_child_zygote ==
JNI_TRUE,
                     instruction_set, app_data_dir, is_top_app == JNI_TRUE,
pkg_data_info_list,
                     allowlisted_data_info_list, mount_data_dirs == JNI_TRUE,
                     mount_storage_dirs == JNI_TRUE);
}

// 继续查看SpecializeCommon实现
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 is_top_app, jobjectArray pkg_data_info_list,
                             jobjectArray allowlisted_data_info_list, bool
mount_data_dirs,
```

```
                                bool mount_storage_dirs) {
    const char* process_name = is_system_server ? "system_server" : "zygote";
    auto fail_fn = std::bind(ZygoteFailure, env, process_name, managed_nice_name,
_1);
    auto extract_fn = std::bind(ExtractJString, env, process_name,
managed_nice_name, _1);

    auto se_info = extract_fn(managed_se_info);
    auto nice_name = extract_fn(managed_nice_name);
    auto instruction_set = extract_fn(managed_instruction_set);
    auto app_data_dir = extract_fn(managed_app_data_dir);
    // 在这里的nick_name就是应用的包名了
    const char* nice_name_ptr = nice_name.has_value() ? nice_name.value().c_str()
: nullptr;
    // 如果是系统服务，就初始化系统服务的classloader
    if (is_system_server) {
        // Prefetch the classloader for the system server. This is done early to
        // allow a tie-down of the proper system server selinux domain.
        env->CallStaticObjectMethod(gZygoteInitClass,
gGetOrCreateSystemServerClassLoader);
        if (env->ExceptionCheck()) {
            // Be robust here. The Java code will attempt to create the
classloader
            // at a later point (but may not have rights to use AoT artifacts).
            env->ExceptionClear();
        }
    }
    ...
    if (selinux_android_setcontext(uid, is_system_server, se_info_ptr,
nice_name_ptr) == -1) {
        fail_fn(CREATE_ERROR("selinux_android_setcontext(%d, %d, \"%s\", \"%s\")
failed", uid,
                             is_system_server, se_info_ptr, nice_name_ptr));
    }

    // Make it easier to debug audit logs by setting the main thread's name to the
    // nice name rather than "app_process".
    if (nice_name.has_value()) {
        SetThreadName(nice_name.value());
    } else if (is_system_server) {
        SetThreadName("system_server");
    }

    // 调用java层的callPostForkChildHooks函数
    // 这个函数主要用来在新创建的子进程中调用回调函数进行初始化。
    env->CallStaticVoidMethod(gZygoteClass, gCallPostForkChildHooks,
runtime_flags,
                              is_system_server, is_child_zygote,
managed_instruction_set);
    ...
}
```

可以在这里插入一个日志，看看在android启动完成时，孵化出了哪些进程。

```
env->CallStaticVoidMethod(gZygoteClass, gCallPostForkChildHooks, runtime_flags,
                              is_system_server, is_child_zygote,
managed_instruction_set);
ALOGW("start CallStaticVoidMethod current process:%s", nice_name_ptr);
```

然后编译aosp后刷入手机中。

```
// 执行脚本初始化编译环境
source ./build/envsetup.sh
// 选择要编译的版本
lunch aosp_blueline-userdebug
// 多线程编译
make -j$(nproc --all)
// 设置刷机目录
export ANDROID_PRODUCT_OUT=~/android_src/out/target/product/blueline
// 手机重启进入bootloader
adb reboot bootloader
// 查看手机是否已经进入bootloader了
fastboot devices
// 将刚刚编译的系统刷入手机
fastboot flashall -w
```

使用android studio的logcat查看日志，或者直接使用命令adb logcat > tmp.log将日志输出到文件中，再进行观察。

```
system_process                      W  start CallStaticVoidMethod current
process:(null)
com.android.bluetooth               W  start CallStaticVoidMethod current
process:com.android.bluetooth
com.android.systemui                W  start CallStaticVoidMethod current
process:com.android.systemui
pid-2292                            W  start CallStaticVoidMethod current
process:WebViewLoader-armeabi-v7a
pid-2293                            W  start CallStaticVoidMethod current
process:WebViewLoader-arm64-v8a
com.android.networkstack            W  start CallStaticVoidMethod current
process:com.android.networkstack.process
com.qualcomm.qti.telephonyservice   W  start CallStaticVoidMethod current
process:com.qualcomm.qti.telephonyservice
pid-2401                            W  start CallStaticVoidMethod current
process:webview_zygote
com.android.se                      W  start CallStaticVoidMethod current
process:com.android.se
com.android.phone                   W  start CallStaticVoidMethod current
process:com.android.phone
com.android.settings                W  start CallStaticVoidMethod current
process:com.android.settings
android.ext.services                W  start CallStaticVoidMethod current
```

```
process:android.ext.services
com.android.launcher3                W  start CallStaticVoidMethod current
process:com.android.launcher3
com....cellbroadcastreceiver.module  W  start CallStaticVoidMethod current
process:com.android.cellbroadcastreceiver.module
com.android.carrierconfig            W  start CallStaticVoidMethod current
process:com.android.carrierconfig
com.android.providers.blockednumber  W  start CallStaticVoidMethod current
process:android.process.acore
pid-2859                             W  start CallStaticVoidMethod current
process:com.android.deskclock
pid-2899                             W  start CallStaticVoidMethod current
process:com.android.nfc
pid-2927                             W  start CallStaticVoidMethod current
process:com.android.keychain
pid-2944                             W  start CallStaticVoidMethod current
process:com.android.providers.media.module
pid-3028                             W  start CallStaticVoidMethod current
process:com.android.quicksearchbox
pid-3059                             W  start CallStaticVoidMethod current
process:com.android.printspooler
pid-3077                             W  start CallStaticVoidMethod current
process:com.android.music
pid-3112                             W  start CallStaticVoidMethod current
process:com.android.traceur
pid-3145                             W  start CallStaticVoidMethod current
process:com.android.dialer
pid-3151                             W  start CallStaticVoidMethod current
process:android.process.media
pid-3213                             W  start CallStaticVoidMethod current
process:com.android.calendar
pid-3230                             W  start CallStaticVoidMethod current
process:com.android.imsserviceentitlement
pid-3256                             W  start CallStaticVoidMethod current
process:com.android.camera2
pid-3277                             W  start CallStaticVoidMethod current
process:com.android.contacts
pid-3302                             W  start CallStaticVoidMethod current
process:com.android.dynsystem
pid-3322                             W  start CallStaticVoidMethod current
process:com.android.dynsystem:dynsystem
pid-3337                             W  start CallStaticVoidMethod current
process:com.android.inputmethod.latin
pid-3359                             W  start CallStaticVoidMethod current
process:com.android.managedprovisioning
pid-3380                             W  start CallStaticVoidMethod current
process:com.android.messaging
pid-3413                             W  start CallStaticVoidMethod current
process:com.android.onetimeinitializer
pid-3436                             W  start CallStaticVoidMethod current
process:com.android.packageinstaller
pid-3455                             W  start CallStaticVoidMethod current
process:com.android.permissioncontroller
pid-3480                             W  start CallStaticVoidMethod current
```

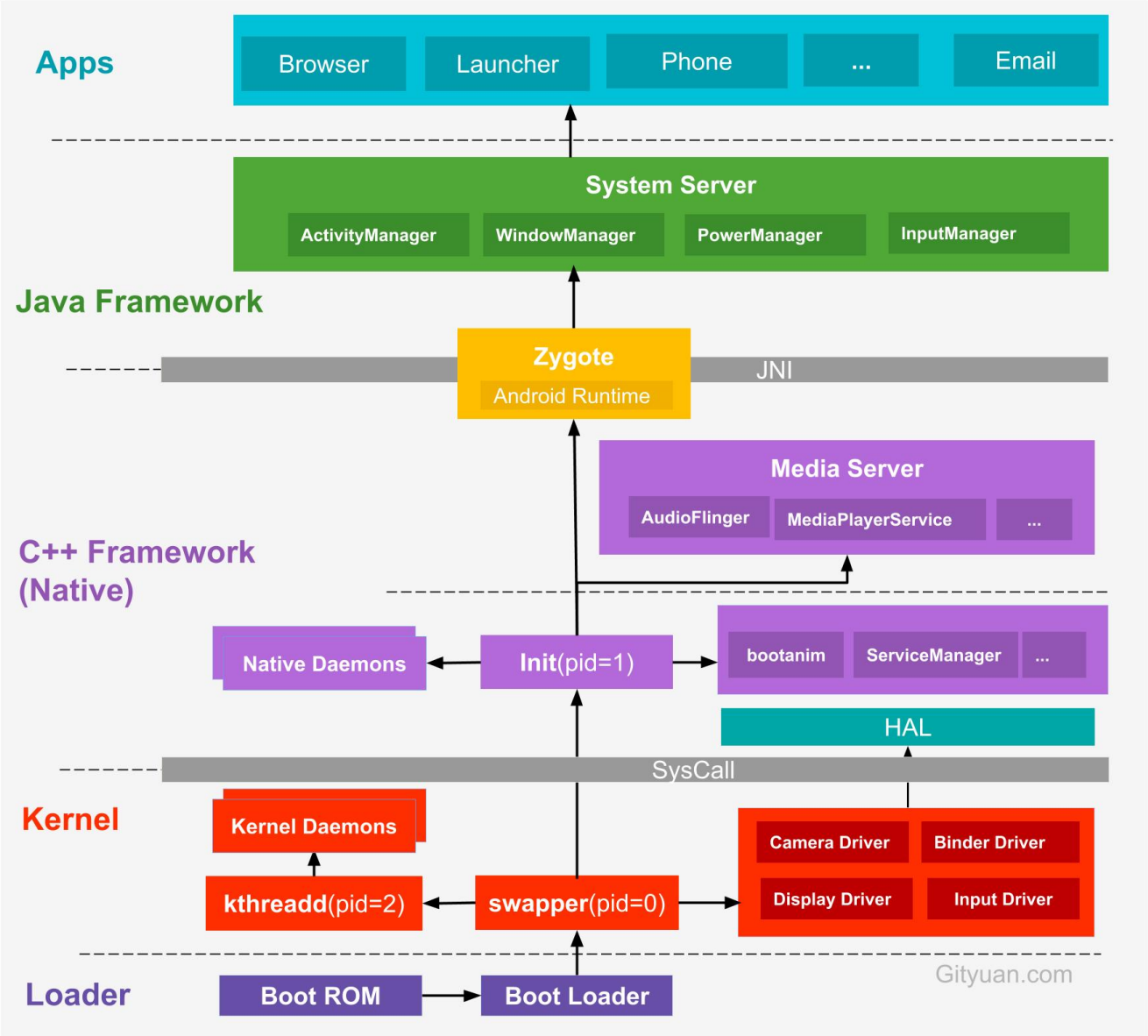
```
process:com.android.providers.calendar
pid-3503                              W  start CallStaticVoidMethod current
process:com.android.settings
pid-3504                              W  start CallStaticVoidMethod current
process:com.android.localtransport
pid-3545                              W  start CallStaticVoidMethod current
process:com.android.shell
pid-3568                              W  start CallStaticVoidMethod current
process:com.android.statementservice
pid-3595                              W  start CallStaticVoidMethod current
process:com.android.quicksearchbox
pid-3615                              W  start CallStaticVoidMethod current
process:com.android.cellbroadcastreceiver.module
pid-3638                              W  start CallStaticVoidMethod current
process:com.android.externalstorage
```

从日志中可以看到system_process进程是孵化出来的第一个进程，接着孵化了一堆系统相关的进程，包括launcher桌面应用管理的系统应用。

根据前文看到的一系列的源码，分析后得出以下几个结论

1. zygote启动实际是启动app_process进程。
2. 由init进程解析init.rc时启动了第一个zygote进程。
3. 在第一个zygote进程中创建的ZygoteServer，并开始监听消息。
4. 其他zygote进程是在ZygoteServer这个服务中收到消息后，再去fork出的新进程。
5. 所有进程均来自于zygote进程的fork而来，所以zygote是进程的始祖。

结合观测到的代码流程，再看下面的一个汇总图。不需要完全理解启动过程中的所有的处理，重点是在这里留下一个大致的印象以及简单的整理。



3.7 Android app应用启动

经过一系列的代码跟踪，学习了android是如何启动的，系统服务是如何启动的，进程是如何启动。相信大家也好奇，当点击打开一个应用后，系统做了一系列的什么工作，最终打开了这个app，调用到MainActivity的onCreate的呢。

当Android成功进入系统后，在主界面中显示的桌面是一个叫做Launcher的系统应用，它是用来显示系统中已经安装的应用程序，并将这些信息的图标作为快捷方式显示在屏幕上，当用户点击图标时，Launcher就会启动对应的应用。在前文中，从forkSystemServer的流程中，最后能看到系统启动准备就绪后拉起了Launcher的应用。

Launcher是如何打开一个应用的呢？其实Launcher本身就是作为第一个应用在系统启动后首先打开的，既然Launcher就是应用。那么在手机上看到各种应用的图标，就是它读取到需要展示的数据，然后布局展示出来的，点击后打开应用，就是给每个item设置的点击事件进行处理的。接着，来看看这个Launcher应用的源码。

查看代码frameworks/base/core/java/android/app/LauncherActivity.java。

```
public abstract class LauncherActivity extends ListActivity {
    ...
    @Override
    protected void onListItemClick(ListView l, View v, int position, long id) {
        Intent intent = intentForPosition(position);
        startActivity(intent);
    }
    ...
}
```

如果你是一名android开发人员，相信你对startActivity这个函数非常熟悉了，但是startActivity是如何打开一个应用的呢，很多人不会深入了解，有了前文中的一系列基础铺垫，这时你已经能尝试追踪调用链了。现在，继续深入挖掘startActivity的原理。

查看代码frameworks/base/core/java/android/app/Activity.java。

```
public void startActivity(Intent intent, @Nullable Bundle options) {
        ...
        if (options != null) {
            startActivityForResult(intent, -1, options);
        } else {
            // Note we want to go through this call for compatibility with
            // applications that may have overridden the method.
            startActivityForResult(intent, -1);
        }
    }
```

继续追踪startActivityForResult的实现。

```
// 继续追踪startActivityForResult
public void startActivityForResult(
            String who, Intent intent, int requestCode, @Nullable Bundle options)
{
        Uri referrer = onProvideReferrer();
        if (referrer != null) {
            intent.putExtra(Intent.EXTRA_REFERRER, referrer);
        }
        options = transferSpringboardActivityOptions(options);
        // 运行Activity
        Instrumentation.ActivityResult ar =
            mInstrumentation.execStartActivity(
                this, mMainThread.getApplicationThread(), mToken, who,
                intent, requestCode, options);
        if (ar != null) {
            mMainThread.sendActivityResult(
                mToken, who, requestCode,
```

```
                ar.getResultCode(), ar.getResultData());
        }
        cancelInputsAndStartExitTransition(options);
    }
```

接下来的关键函数是`execStartActivity`，继续深入

```
// 继续追踪execStartActivity
public ActivityResult execStartActivity(
            Context who, IBinder contextThread, IBinder token, Activity target,
            Intent intent, int requestCode, Bundle options) {
        ...
        try {
            intent.migrateExtraStreamToClipData(who);
            intent.prepareToLeaveProcess(who);
            // 启动Activity
            int result = ActivityTaskManager.getService().startActivity(whoThread,
                    who.getOpPackageName(), who.getAttributionTag(), intent,
                    intent.resolveTypeIfNeeded(who.getContentResolver()), token,
                    target != null ? target.mEmbeddedID : null, requestCode, 0,
null, options);
            checkStartActivityResult(result, intent);
        } catch (RemoteException e) {
            throw new RuntimeException("Failure from system", e);
        }
        return null;
    }
```

`ActivityTaskManager`下的`service`调用的`startActivity`。

查看代码

`frameworks/base/services/core/java/com/android/server/wm/ActivityTaskManagerService.java`

```
public final int startActivity(IApplicationThread caller, String callingPackage,
                               String callingFeatureId, Intent intent, String
resolvedType, IBinder resultTo,
                               String resultWho, int requestCode, int startFlags,
ProfilerInfo profilerInfo,
                               Bundle bOptions) {
    return startActivityAsUser(caller, callingPackage, callingFeatureId, intent,
resolvedType,
                               resultTo, resultWho, requestCode, startFlags,
profilerInfo, bOptions,
                               UserHandle.getCallingUserId());
}
```

```
private int startActivityAsUser(IApplicationThread caller, String callingPackage,
            @Nullable String callingFeatureId, Intent intent, String resolvedType,
            IBinder resultTo, String resultWho, int requestCode, int startFlags,
            ProfilerInfo profilerInfo, Bundle bOptions, int userId, boolean
validateIncomingUser) {
        ...

        return getActivityStartController().obtainStarter(intent,
"startActivityAsUser")
                .setCaller(caller)
                .setCallingPackage(callingPackage)
                .setCallingFeatureId(callingFeatureId)
                .setResolvedType(resolvedType)
                .setResultTo(resultTo)
                .setResultWho(resultWho)
                .setRequestCode(requestCode)
                .setStartFlags(startFlags)
                .setProfilerInfo(profilerInfo)
                .setActivityOptions(bOptions)
                .setUserId(userId)
                .execute();

    }
```

先看看obtainStarter返回的对象类型。

```
ActivityStarter obtainStarter(Intent intent, String reason) {
        return mFactory.obtain().setIntent(intent).setReason(reason);
    }
```

看到返回的是ActivityStarter类型，接着找到对应的excute的实现

```
// 处理 Activity 启动请求的接口
int execute() {
        ...
        res = executeRequest(mRequest);
        ...
    }

// 各种权限检查，合法的请求则继续
private int executeRequest(Request request) {
        ...
        mLastStartActivityResult = startActivityUnchecked(r, sourceRecord,
voiceSession,
                request.voiceInteractor, startFlags, true /* doResume */,
checkedOptions, inTask,
                restrictedBgActivity, intentGrants);
```

```
        ...
    }

private int startActivityUnchecked(final ActivityRecord r, ActivityRecord
sourceRecord,
                IVoiceInteractionSession voiceSession, IVoiceInteractor
voiceInteractor,
                int startFlags, boolean doResume, ActivityOptions options, Task
inTask,
                boolean restrictedBgActivity, NeededUriGrants intentGrants) {
       ...
       Trace.traceBegin(Trace.TRACE_TAG_WINDOW_MANAGER, "startActivityInner");
       result = startActivityInner(r, sourceRecord, voiceSession, voiceInteractor,
       startFlags, doResume, options, inTask, restrictedBgActivity, intentGrants);
       ...
    }


int startActivityInner(final ActivityRecord r, ActivityRecord sourceRecord,
            IVoiceInteractionSession voiceSession, IVoiceInteractor
voiceInteractor,
            int startFlags, boolean doResume, ActivityOptions options, Task
inTask,
            boolean restrictedBgActivity, NeededUriGrants intentGrants) {
        ...
        // 判断是否需要为 Activity 创建新的 Task
        mTargetRootTask.startActivityLocked(mStartActivity,
                topRootTask != null ? topRootTask.getTopNonFinishingActivity() :
null, newTask,
                mKeepCurTransition, mOptions, sourceRecord);
        // 如果需要恢复 Activity
        if (mDoResume) {
            final ActivityRecord topTaskActivity =
                    mStartActivity.getTask().topRunningActivityLocked();
            // 判断当前 Activity 是否可见以及是否需要暂停后台 Activity
            if (!mTargetRootTask.isTopActivityFocusable()
                    || (topTaskActivity != null && topTaskActivity.isTaskOverlay()
                    && mStartActivity != topTaskActivity)) {
                ...
            } else {
                // 如果当前 Activity 可见，则将其移动到前台
                if (mTargetRootTask.isTopActivityFocusable()
                        &&
!mRootWindowContainer.isTopDisplayFocusedRootTask(mTargetRootTask)) {
                    mTargetRootTask.moveToFront("startActivityInner");
                }
                // 恢复处于焦点状态的 Activity 的顶部 Activity
                mRootWindowContainer.resumeFocusedTasksTopActivities(
                        mTargetRootTask, mStartActivity, mOptions,
mTransientLaunch);
            }
        }
        ...
    }
```

```
// 恢复处于焦点状态的 Activity 的顶部 Activity。
boolean resumeFocusedTasksTopActivities(
            Task targetRootTask, ActivityRecord target, ActivityOptions
targetOptions,
            boolean deferPause) {
        ...
        // 遍历所有显示器
        for (int displayNdx = getChildCount() - 1; displayNdx >= 0; --displayNdx)
{
            final DisplayContent display = getChildAt(displayNdx);
            ...
            // 获取当前焦点所在的任务根节点
            final Task focusedRoot = display.getFocusedRootTask();
            // 如果有任务根节点，则恢复任务根节点中顶部的 Activity
            if (focusedRoot != null) {
                result |= focusedRoot.resumeTopActivityUncheckedLocked(target,
targetOptions);
            } else if (targetRootTask == null) {
                // 如果没有焦点任务根节点，并且目标任务根节点为空，则恢复 Home Activity
                result |= resumeHomeActivity(null /* prev */, "no-focusable-task",
                                             display.getDefaultTaskDisplayArea());
            }
        }
        return result;
    }

// 恢复位于任务根节点顶部的 Activity
boolean resumeTopActivityUncheckedLocked(ActivityRecord prev, ActivityOptions
options,
            boolean deferPause) {
        ...
        someActivityResumed = resumeTopActivityInnerLocked(prev, options,
deferPause);
        ...
    }

// 恢复位于任务根节点顶部的 Activity。
private boolean resumeTopActivityInnerLocked(ActivityRecord prev, ActivityOptions
options,
            boolean deferPause) {
        ...

        mTaskSupervisor.startSpecificActivity(next, true, true);
        ...

        return true;
    }
```

`startSpecificActivity`将启动指定的`Activity`。

```
void startSpecificActivity(ActivityRecord r, boolean andResume, boolean
checkConfig) {
        // 是否已经有进程在运行这个应用程序?
        final WindowProcessController wpc =
                mService.getProcessController(r.processName,
r.info.applicationInfo.uid);

        boolean knownToBeDead = false;
        // 如果应用程序正在运行，则直接启动 Activity
        if (wpc != null && wpc.hasThread()) {
            try {
                realStartActivityLocked(r, wpc, andResume, checkConfig);
                return;
            } catch (RemoteException e) {
                Slog.w(TAG, "Exception when starting activity "
                        + r.intent.getComponent().flattenToShortString(), e);
            }

            // If a dead object exception was thrown -- fall through to
            // restart the application.
            knownToBeDead = true;
        }
        // 通知 Keyguard 正在启动一个不确定的 Activity（仅在 Keyguard 转换期间使用）
        r.notifyUnknownVisibilityLaunchedForKeyguardTransition();
        // 如果应用程序未运行，则异步启动新进程
        final boolean isTop = andResume && r.isTopRunningActivity();
        mService.startProcessAsync(r, knownToBeDead, isTop, isTop ? "top-activity"
: "activity");
    }
```

主要关注开启一个新应用的流程，所以这里只追踪startProcessAsync调用即可。

```
void startProcessAsync(ActivityRecord activity, boolean knownToBeDead, boolean
isTop,
            String hostingType) {
        try {
            if (Trace.isTagEnabled(TRACE_TAG_WINDOW_MANAGER)) {
                Trace.traceBegin(TRACE_TAG_WINDOW_MANAGER,
"dispatchingStartProcess:"
                        + activity.processName);
            }
            // Post message to start process to avoid possible deadlock of calling
into AMS with the
            // ATMS lock held.
            final Message m =
PooledLambda.obtainMessage(ActivityManagerInternal::startProcess,
                    mAmInternal, activity.processName,
activity.info.applicationInfo, knownToBeDead,
                    isTop, hostingType, activity.intent.getComponent());
            mH.sendMessage(m);
        } finally {
```

```
            Trace.traceEnd(TRACE_TAG_WINDOW_MANAGER);
        }
    }
```

上面开启新进程的代码是异步发送消息给了ActivityManagerService。找到AMS中对应的startProcess。

```
@Override
public void startProcess(String processName, ApplicationInfo info, boolean
knownToBeDead,
                         boolean isTop, String hostingType, ComponentName
hostingName) {
    try {
        if (Trace.isTagEnabled(Trace.TRACE_TAG_ACTIVITY_MANAGER)) {
            Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "startProcess:"
                                + processName);
        }
        synchronized (ActivityManagerService.this) {
            // If the process is known as top app, set a hint so when the process
is
            // started, the top priority can be applied immediately to avoid cpu
being
            // preempted by other processes before attaching the process of top
app.
            startProcessLocked(processName, info, knownToBeDead, 0 /* intentFlags
*/,
                               new HostingRecord(hostingType, hostingName, isTop),
                               ZYGOTE_POLICY_FLAG_LATENCY_SENSITIVE, false /*
allowWhileBooting */,
                               false /* isolated */);
        }
    } finally {
        Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
    }
}

// 继续追踪startProcessLocked
final ProcessRecord startProcessLocked(String processName,
                                       ApplicationInfo info, boolean
knownToBeDead, int intentFlags,
                                       HostingRecord hostingRecord, int
zygotePolicyFlags, boolean allowWhileBooting,
                                       boolean isolated) {
    return mProcessList.startProcessLocked(processName, info, knownToBeDead,
intentFlags,
                                           hostingRecord, zygotePolicyFlags,
allowWhileBooting, isolated, 0 /* isolatedUid */,
                                           null /* ABI override */, null /*
entryPoint */,
                                           null /* entryPointArgs */, null /*
crashHandler */);
}
```

```
// 在这里初始化了一堆进程信息，然后调用了另一个重载
// 并且注意entryPoint赋值android.app.ActivityThread
boolean startProcessLocked(ProcessRecord app, HostingRecord hostingRecord,
                           int zygotePolicyFlags, boolean disableHiddenApiChecks,
boolean disableTestApiChecks,
                           String abiOverride) {
    ...
        // the PID of the new process, or else throw a RuntimeException.
        final String entryPoint = "android.app.ActivityThread";

    return startProcessLocked(hostingRecord, entryPoint, app, uid, gids,
                              runtimeFlags, zygotePolicyFlags, mountExternal,
seInfo, requiredAbi,
                              instructionSet, invokeWith, startTime);
}

//
boolean startProcessLocked(HostingRecord hostingRecord, String entryPoint,
ProcessRecord app,
                           int uid, int[] gids, int runtimeFlags, int
zygotePolicyFlags, int mountExternal,
                           String seInfo, String requiredAbi, String
instructionSet, String invokeWith,
                           long startTime) {
    ...
        final Process.ProcessStartResult startResult = startProcess(hostingRecord,
                                                                    entryPoint,
app,
                                                                    uid, gids,
runtimeFlags, zygotePolicyFlags, mountExternal, seInfo,
                                                                    requiredAbi,
instructionSet, invokeWith, startTime);
    handleProcessStartedLocked(app, startResult.pid, startResult.usingWrapper,
                               startSeq, false);
    ...
        return app.getPid() > 0;

}

// 继续查看startProcess
private Process.ProcessStartResult startProcess(HostingRecord hostingRecord,
String entryPoint,
                                                ProcessRecord app, int uid, int[]
gids, int runtimeFlags, int zygotePolicyFlags,
                                                int mountExternal, String seInfo,
String requiredAbi, String instructionSet,
                                                String invokeWith, long startTime)
{
    ...
        final Process.ProcessStartResult startResult;
    boolean regularZygote = false;
    // 这里根据应用情况使用不同类型的zygote来启动进程
    if (hostingRecord.usesWebviewZygote()) {
        startResult = startWebView(entryPoint,
```

```
                                    app.processName, uid, uid, gids, runtimeFlags,
mountExternal,
                                    app.info.targetSdkVersion, seInfo, requiredAbi,
instructionSet,
                                    app.info.dataDir, null, app.info.packageName,
                                    app.getDisabledCompatChanges(),
                                    new String[]{PROC_START_SEQ_IDENT +
app.getStartSeq()});
    } else if (hostingRecord.usesAppZygote()) {
        final AppZygote appZygote = createAppZygoteForProcessIfNeeded(app);

        // We can't isolate app data and storage data as parent zygote already did
that.
        startResult = appZygote.getProcess().start(entryPoint,
                                                   app.processName, uid, uid,
gids, runtimeFlags, mountExternal,
                                                   app.info.targetSdkVersion,
seInfo, requiredAbi, instructionSet,
                                                   app.info.dataDir, null,
app.info.packageName,
                                                   /*zygotePolicyFlags=*/
ZYGOTE_POLICY_FLAG_EMPTY, isTopApp,
                                                   app.getDisabledCompatChanges(),
pkgDataInfoMap, allowlistedAppDataInfoMap,
                                                   false, false,
                                                   new String[]
{PROC_START_SEQ_IDENT + app.getStartSeq()});
    } else {
        regularZygote = true;
        startResult = Process.start(entryPoint,
                                    app.processName, uid, uid, gids, runtimeFlags,
mountExternal,
                                    app.info.targetSdkVersion, seInfo,
requiredAbi, instructionSet,
                                    app.info.dataDir, invokeWith,
app.info.packageName, zygotePolicyFlags,
                                    isTopApp, app.getDisabledCompatChanges(),
pkgDataInfoMap,
                                    allowlistedAppDataInfoMap, bindMountAppsData,
bindMountAppStorageDirs,
                                    new String[]{PROC_START_SEQ_IDENT +
app.getStartSeq()});
    }

    if (!regularZygote) {
        // webview and app zygote don't have the permission to create the nodes
        if (Process.createProcessGroup(uid, startResult.pid) < 0) {
            Slog.e(ActivityManagerService.TAG, "Unable to create process group for
"
                    + app.processName + " (" + startResult.pid + ")");
        }
    }
    ...
        return startResult;
```

```
}
```

这里，看到了zygote有三种类型，根据启动的应用信息使用不同类型的zygote来启动。

1. regularZygote常规进程，zygote32/zygote64进程，是所有Android Java应用的父进程
2. appZygote应用进程，比常规进程多一些限制。
3. webviewZygote辅助zygote进程，渲染不可信的web内容，最严格的安全限制

三种zygote类型的启动流程差不多的，看常规进程启动即可。首先看getProcess返回的是什么类型

```
public ChildZygoteProcess getProcess() {
        synchronized (mLock) {
            if (mZygote != null) return mZygote;

            connectToZygoteIfNeededLocked();
            return mZygote;
        }
    }
```

应该找ChildZygoteProcess的start函数，然后找到类定义后，发现没有start，那么应该就是父类中的实现。

```
public class ChildZygoteProcess extends ZygoteProcess {
    private final int mPid;

    ChildZygoteProcess(LocalSocketAddress socketAddress, int pid) {
        super(socketAddress, null);
        mPid = pid;
    }

    public int getPid() {
        return mPid;
    }
}
```

继续找到父类ZygoteProcess的start函数，参数太长，这里省略掉参数的描述

```
public final Process.ProcessStartResult start(...) {
        ...
        return startViaZygote(processClass, niceName, uid, gid, gids,
                    runtimeFlags, mountExternal, targetSdkVersion, seInfo,
                    abi, instructionSet, appDataDir, invokeWith,
/*startChildZygote=*/ false,
```

```
                    packageName, zygotePolicyFlags, isTopApp,
disabledCompatChanges,
                    pkgDataInfoMap, allowlistedDataInfoList, bindMountAppsData,
                    bindMountAppStorageDirs, zygoteArgs);
        ...
    }


private Process.ProcessStartResult startViaZygote(...)
                                                    throws ZygoteStartFailedEx {
        ArrayList<String> argsForZygote = new ArrayList<>();
        // 前面是将前面准备的参数填充好
        // --runtime-args, --setuid=, --setgid=,
        // and --setgroups= must go first
        argsForZygote.add("--runtime-args");
        argsForZygote.add("--setuid=" + uid);
        argsForZygote.add("--setgid=" + gid);
        argsForZygote.add("--runtime-flags=" + runtimeFlags);
        if (mountExternal == Zygote.MOUNT_EXTERNAL_DEFAULT) {
            argsForZygote.add("--mount-external-default");
        } else if (mountExternal == Zygote.MOUNT_EXTERNAL_INSTALLER) {
            argsForZygote.add("--mount-external-installer");
        } else if (mountExternal == Zygote.MOUNT_EXTERNAL_PASS_THROUGH) {
            argsForZygote.add("--mount-external-pass-through");
        } else if (mountExternal == Zygote.MOUNT_EXTERNAL_ANDROID_WRITABLE) {
            argsForZygote.add("--mount-external-android-writable");
        }
        ...
        synchronized(mLock) {
            // The USAP pool can not be used if the application will not use the
systems graphics
            // driver.  If that driver is requested use the Zygote application
start path.
            return zygoteSendArgsAndGetResult(openZygoteSocketIfNeeded(abi),
                                              zygotePolicyFlags,
                                              argsForZygote);
        }
    }

private Process.ProcessStartResult zygoteSendArgsAndGetResult(
            ZygoteState zygoteState, int zygotePolicyFlags, @NonNull
ArrayList<String> args)
            throws ZygoteStartFailedEx {
        ...
        //是否用非特定的应用程序进程池进行处理，默认不使用
        if (shouldAttemptUsapLaunch(zygotePolicyFlags, args)) {
            try {
                return attemptUsapSendArgsAndGetResult(zygoteState, msgStr);
            } catch (IOException ex) {
                // If there was an IOException using the USAP pool we will log the
error and
                // attempt to start the process through the Zygote.
                Log.e(LOG_TAG, "IO Exception while communicating with USAP pool -
"
```

```
                        + ex.getMessage());
            }
        }

        return attemptZygoteSendArgsAndGetResult(zygoteState, msgStr);
    }


private Process.ProcessStartResult attemptZygoteSendArgsAndGetResult(
            ZygoteState zygoteState, String msgStr) throws ZygoteStartFailedEx {
        try {
            final BufferedWriter zygoteWriter = zygoteState.mZygoteOutputWriter;
            final DataInputStream zygoteInputStream =
zygoteState.mZygoteInputStream;
            // 这里实际就是连接SocketServer了，发送一个消息给zygote孵化出来的第一个进
程
            zygoteWriter.write(msgStr);
            zygoteWriter.flush();

            Process.ProcessStartResult result = new Process.ProcessStartResult();
            result.pid = zygoteInputStream.readInt();
            result.usingWrapper = zygoteInputStream.readBoolean();
            // ZygoteServer创建好进程后，返回pid
            if (result.pid < 0) {
                throw new ZygoteStartFailedEx("fork() failed");
            }

            return result;
        } catch (IOException ex) {
            zygoteState.close();
            Log.e(LOG_TAG, "IO Exception while communicating with Zygote - "
                    + ex.toString());
            throw new ZygoteStartFailedEx(ex);
        }
    }
```

到这里，回首看看前文中介绍ZygoteServer启动进程的流程，当时看到执行到最后是findStaticMain函数，是获取一个类名下的main函数，并返回后进行调用。现在启动进程时，在startProcessLocked函数中能看到类名赋值是android.app.ActivityThread，所以这里和ZygoteServer进行通信创建线程，最后调用的函数就是android.app.ActivityThread中的main函数。这样一来，启动流程就进入的应用的主线程。

ActivityThread是Android应用程序运行的UI主线程，负责处理应用程序的所有生命周期事件，接收系统消息并处理它们，main函数就是安卓应用的入口函数。prepareMainLooper函数将实例化一个Looper对象，然后由Looper对象创建一个消息队列，当loop函数调用时，UI线程就会进入消息循环，不断从消息队列获取到消息去进行相应的处理。

```
public static void main(String[] args) {
    ...
    Looper.prepareMainLooper();
    ...
    ActivityThread thread = new ActivityThread();
```

```
        thread.attach(false, startSeq);
        // 主线程消息循环处理的handler
        if (sMainThreadHandler == null) {
            sMainThreadHandler = thread.getHandler();
        }
        ...
        Looper.loop();
    }
    // 在loop函数中是一个死循环进行`loopOnce`调用
    public static void loop() {
        ...
        for (;;) {
            if (!loopOnce(me, ident, thresholdOverride)) {
                return;
            }
        }
    }

    private void attach(boolean system, long startSeq) {
        ...
        mgr.attachApplication(mAppThread, startSeq);
        ...
    }
```

继续看loopOnce的实现，看到了从队列中获取一条消息，并且将消息派发给对应的Handler来执行。

```
    private static boolean loopOnce(final Looper me,
                final long ident, final int thresholdOverride) {
            Message msg = me.mQueue.next(); // might block
            if (msg == null) {
                // No message indicates that the message queue is quitting.
                return false;
            }
            ...
            msg.target.dispatchMessage(msg);
            ...
            return true;
        }
```

对应的消息处理的Handler就是前面在入口函数main中看到的sMainThreadHandler对象，是通过getHandler函数获取的，跟进去寻找具体的对象。

```
    public Handler getHandler() {
            return mH;
        }

    final H mH = new H();
```

找到的这个H类型就是对应的主线程消息处理Handler了。看看相关实现。

```
class H extends Handler {
        public static final int BIND_APPLICATION        = 110;
        @UnsupportedAppUsage
        public static final int EXIT_APPLICATION        = 111;
        ...
        String codeToString(int code) {
            if (DEBUG_MESSAGES) {
                switch (code) {
                    case BIND_APPLICATION: return "BIND_APPLICATION";
                    case EXIT_APPLICATION: return "EXIT_APPLICATION";
                    ...
                }
            }
            return Integer.toString(code);
        }
        public void handleMessage(Message msg) {
            if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " +
codeToString(msg.what));
            switch (msg.what) {
                case BIND_APPLICATION:
                    Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER,
"bindApplication");
                    AppBindData data = (AppBindData)msg.obj;
                    handleBindApplication(data);
                    Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
                    break;
                case EXIT_APPLICATION:
                    if (mInitialApplication != null) {
                        mInitialApplication.onTerminate();
                    }
                    Looper.myLooper().quit();
                    break;
                ...
            }
            ...
        }
    }
```

再回头看看thread.attach中的处理，mgr就是AMS，所以来到ActivityManagerService查看attachApplication：

```
// 在main中调用的thread.attach函数
private void attach(boolean system, long startSeq) {
    ...
    mgr.attachApplication(mAppThread, startSeq);
    ...
}
```

```
// 将应用程序线程与 ActivityThread 绑定
public final void attachApplication(IApplicationThread thread, long startSeq) {
    if (thread == null) {
        throw new SecurityException("Invalid application interface");
    }
    synchronized (this) {
        int callingPid = Binder.getCallingPid();
        final int callingUid = Binder.getCallingUid();
        final long origId = Binder.clearCallingIdentity();
        attachApplicationLocked(thread, callingPid, callingUid, startSeq);
        Binder.restoreCallingIdentity(origId);
    }
}

// 继续追踪attachApplicationLocked
private boolean attachApplicationLocked(@NonNull IApplicationThread thread,
            int pid, int callingUid, long startSeq) {

            ...
            if (app.getIsolatedEntryPoint() != null) {
                // This is an isolated process which should just call an entry
point instead of
                // being bound to an application.
                thread.runIsolatedEntryPoint(
                        app.getIsolatedEntryPoint(),
app.getIsolatedEntryPointArgs());
            } else if (instr2 != null) {
                // 如果该应用程序未运行在隔离进程中，且有 Instrumentation
                thread.bindApplication(processName, appInfo, providerList,
                        instr2.mClass,
                        profilerInfo, instr2.mArguments,
                        instr2.mWatcher,
                        instr2.mUiAutomationConnection, testMode,
                        mBinderTransactionTrackingEnabled, enableTrackAllocation,
                        isRestrictedBackupMode || !normalMode, app.isPersistent(),
                        new
Configuration(app.getWindowProcessController().getConfiguration()),
                        app.getCompat(), getCommonServicesLocked(app.isolated),
                        mCoreSettingsObserver.getCoreSettingsLocked(),
                        buildSerial, autofillOptions, contentCaptureOptions,
                        app.getDisabledCompatChanges(), serializedSystemFontMap);
            } else {
                // 如果没有 Instrumentation
                thread.bindApplication(processName, appInfo, providerList, null,
profilerInfo,
                        null, null, null, testMode,
                        mBinderTransactionTrackingEnabled, enableTrackAllocation,
                        isRestrictedBackupMode || !normalMode, app.isPersistent(),
                        new
Configuration(app.getWindowProcessController().getConfiguration()),
                        app.getCompat(), getCommonServicesLocked(app.isolated),
                        mCoreSettingsObserver.getCoreSettingsLocked(),
                        buildSerial, autofillOptions, contentCaptureOptions,
                        app.getDisabledCompatChanges(), serializedSystemFontMap);
```

```
            }
            ...
    }
```

最后调用回ActivityThread的bindApplication，继续跟进去查看

```
public final void bindApplication(...) {
    ...
    //将应用程序和应用程序线程绑定所需的信息存储到AppBindData的各个字段中。
    AppBindData data = new AppBindData();
    data.processName = processName;
    data.appInfo = appInfo;
    data.providers = providerList.getList();
    data.instrumentationName = instrumentationName;
    data.instrumentationArgs = instrumentationArgs;
    data.instrumentationWatcher = instrumentationWatcher;
    data.instrumentationUiAutomationConnection = instrumentationUiConnection;
    data.debugMode = debugMode;
    data.enableBinderTracking = enableBinderTracking;
    data.trackAllocation = trackAllocation;
    data.restrictedBackupMode = isRestrictedBackupMode;
    data.persistent = persistent;
    data.config = config;
    data.compatInfo = compatInfo;
    data.initProfilerInfo = profilerInfo;
    data.buildSerial = buildSerial;
    data.autofillOptions = autofillOptions;
    data.contentCaptureOptions = contentCaptureOptions;
    data.disabledCompatChanges = disabledCompatChanges;
    data.mSerializedSystemFontMap = serializedSystemFontMap;
    // 发送消息给应用程序线程，调用 bindApplication 方法
    sendMessage(H.BIND_APPLICATION, data);
}
```

AppBindData数据绑定完成后，最后发送消息BIND_APPLICATION通知准备就绪，并将准备好的数据发送过去。查看消息循环的处理部分handleMessage函数，看这个数据传给哪个函数处理了。

```
public void handleMessage(Message msg) {
            if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " +
codeToString(msg.what));
            switch (msg.what) {
                case BIND_APPLICATION:
                    Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER,
"bindApplication");
                    AppBindData data = (AppBindData)msg.obj;
                    handleBindApplication(data);
                    Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
                    break;
```

```
            ...
}
```

发现调用到了handleBindApplication，继续跟进查看。

```
private void handleBindApplication(AppBindData data) {
    ...
    //前面准备好的data数据赋值给了mBoundApplication
    mBoundApplication = data;
    ...
    // 创建出了Context
    final ContextImpl appContext = ContextImpl.createAppContext(this, data.info);
    ...
    Application app;
    final StrictMode.ThreadPolicy savedPolicy =
StrictMode.allowThreadDiskWrites();
    final StrictMode.ThreadPolicy writesAllowedPolicy =
StrictMode.getThreadPolicy();
    try {
        // 创建出了Application
        app = data.info.makeApplication(data.restrictedBackupMode, null);
        ...
        // Application赋值给了mInitialApplication
        mInitialApplication = app;
        ...
        try {
            mInstrumentation.callApplicationOnCreate(app);
        } catch (Exception e) {
            ...
        }
    } finally {
        ...
    }
    ...
}

// 看看是如何创建出Application的
public Application makeApplication(boolean forceDefaultAppClass,
                                   Instrumentation instrumentation) {
    if (mApplication != null) {
        return mApplication;
    }
    ...
    app = mActivityThread.mInstrumentation.newApplication(
    cl, appClass, appContext);
    ...
    return app;
}

// 继续看newApplication的实现
static public Application newApplication(Class<?> clazz, Context context)
    throws InstantiationException, IllegalAccessException,
```

```
ClassNotFoundException {
    Application app = (Application)clazz.newInstance();
    // 最后发现调用了attach
    app.attach(context);
    return app;
}
```

在上面看到了Context的创建和Application的创建，继续看看怎么调用到自己开发的app中的onCreate的，追踪callApplicationOnCreate的实现

```
public void callApplicationOnCreate(Application app) {
    ...
    app.onCreate();
}
```

到这里，成功跟踪到最后调用app应用的onCreate函数，为什么很多人喜欢hook attach函数，因为在Application创建出来最早先调用了这个函数，该函数是一个较早hook时机。

3.8 了解Service

Service是一种运行在后台的组件也可以称之为服务，它不像Activity那样有前台显示用户界面的能力，而是一种更加抽象的组件，它可以提供后台服务，在后台定时执行某些任务。Service可以被应用程序绑定，也可以独立运行，它可以接收外部的命令，执行耗时的任务。

在Android启动流程中，就已经看到了很多Service的启动，前文代码看到当系统启动后通过forkSystemServer执行到SystemServer来启动一系列的Service。这些Service有着各自负责的功能，其中最关键的是ActivityManagerService，常常被简称为AMS。而启动了AMS的SystemServer也是一个服务，这个服务负责在Android完成启动后，加载和启动所有的系统服务，管理系统级别的资源。

AMS是Android系统中的一个核心服务，负责Android系统中的所有活动管理，包括应用程序的启动，暂停，恢复，终止，以及对系统资源的管理和分配。负责Android系统中所有活动的管理。它负责管理任务栈，并允许任务栈中的任务来回切换，以便在任务之间改变焦点。它还负责管理进程，并将进程启动，暂停，恢复，终止，以及分配系统资源。在启动流程中能看到，所有Service都是由它来启动的.

除了AMS外，还有其他重要的Service为Android应用提供基础的功能，下面简单介绍这些常见的Service。

WindowManagerService，它是负责管理系统上所有窗口的显示和操作，包括管理全屏窗口、小窗口、弹窗、菜单和其他应用程序的窗口，使窗口在手机屏幕上正确的显示。

PackageManagerService，Android系统中提供给应用程序访问Android软件包的主要服务。负责管理Android软件包的安装、删除和更新，以及软件包的查询和配置。它有一个名为Packages.xml的XML文档，该文档是Android系统中所有软件包的列表，其中包含了每个软件包的基本信息，如应用程序的版本，安装时间，文件大小等。

PowerManagerService，管理设备电源状态的服务，可以有效地管理设备的电源，从而大大提升设备的电池续航能力，也可以降低设备运行时的功耗。它负责处理设备上的所有电源相关操作，例如屏幕亮度、屏幕超时时间、电池和充电时的运行模式、设备锁以及设备唤醒功能。

InputMethodManagerService，输入法服务，它负责处理用户输入，管理输入法状态，以及向应用程序提供输入服务，例如可以安装、卸载和更新输入法，还可以管理系统的输入法开关，应用程序可以通过它来访问输入法的当前状态和内容，以及实时输入的文本内容，可以接收并处理用户的输入事件，包括按键、触摸屏、语音输入等。

NotificationManagerService，通知服务。它主要是用来管理系统的通知，包括消息、提醒、更新等，它实现了通知的管理，收集、组织、过滤通知，并将它们发送给用户。它能够管理所有应用程序发出的通知，包括系统通知、应用程序发出的通知，并可以根据用户的偏好，显示哪些通知。

LocationManagerService，位置管理服务。可以根据应用程序的要求调用GPS、网络和其他位置技术来获取当前设备的定位信息。根据设备的位置信息，控制应用程序的定位功能，以及设备的位置报警功能。

InputManagerService，负责输入设备的管理和控制，以及系统中所有输入事件的处理。例如触摸屏、虚拟按键、键盘、轨迹球等。会将输入事件传递给应用程序，以便处理和响应。

AlarmManagerService负责处理所有系统定时任务，如闹钟，定时器等。它可以安排可执行的任务，使它们在指定的时刻开始执行。监控系统中的各种时间事件，以执行指定的任务。可以发送唤醒广播，以启动指定的服务或应用程序。可以用于处理设备睡眠、唤醒等系统状态切换。

NetworkManagementService，网络管理服务。用于控制和管理Android系统中的网络连接，能够在不同的网络之间进行切换，检查和管理手机的网络状态，监控网络设备的连接状态，如WiFi、蓝牙、移动数据等。

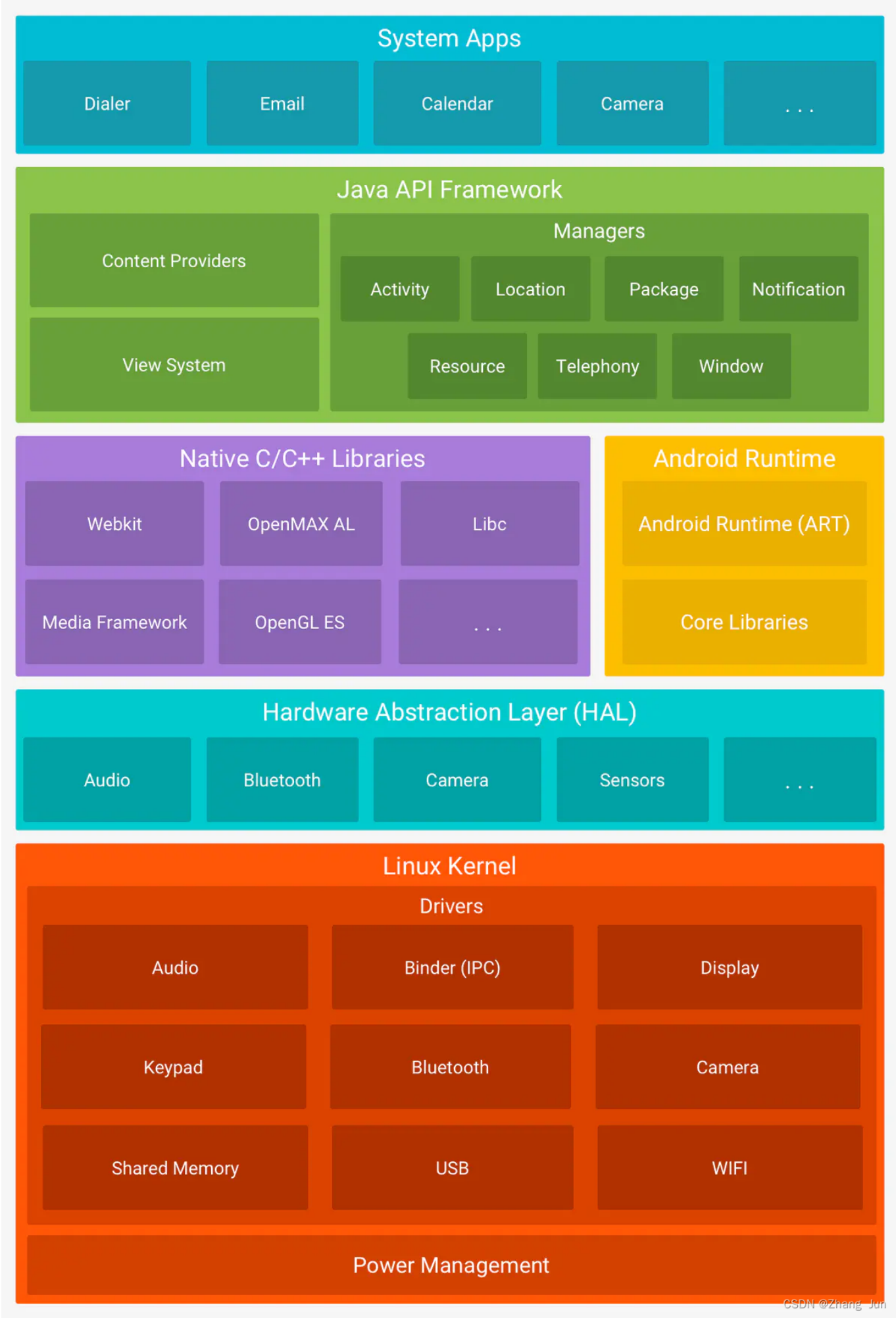
BluetoothService，蓝牙服务，它可以实现蓝牙设备之间的无线通信。它提供了一种方便的方式来建立和管理蓝牙连接，使蓝牙设备之间能够进行文件传输、远程打印、蓝牙键盘连接等活动。

还有更多的系统服务为Android的运行提供着各模块的基础功能，这里就不展开详细叙述了，当对某一个服务的功能实现感兴趣时，可以顺着启动服务的地方开始跟踪代码，分析实现的逻辑。也可以直接参考系统服务的定义方式来自定义系统服务来提供特殊需求的功能。

3.9 了解Framework

Framework指的是软件开发框架，由于系统处于内核中，无法直接对系统的功能进行请求，而是由框架层为开发的顶层应用提供接口调用，从而不必烦恼如何与底层交互，开发框架为开发人员提供各种功能，以及Android应用工具的支持来便于创建和管理Android应用程序，最终达到让用户能高效开发Android应用的目的，以生活中的事务为例，Framework就像是一个配套完善的小区，有高效的物业，周边配套有学校、医院、商场，各类设施非常齐全，而用户就像是小区内的业主。

看一张经典的Android架构图。


System Apps
Dialer
Email
Calendar
Camera
. . .
Java API Framework
Content Providers
View System
Managers
Activity
Location
Package
Notification
Resource
Telephony
Window
Native C/C++ Libraries
Webkit
OpenMAX AL
Libc
Media Framework
OpenGL ES
. . .
Android Runtime
Android Runtime (ART)
Core Libraries
Hardware Abstraction Layer (HAL)
Audio
Bluetooth
Camera
Sensors
. . .
Linux Kernel
Drivers
Audio
Binder (IPC)
Display
Keypad
Bluetooth
Camera
Shared Memory
USB
WIFI
Power Management
CSDN @Zhang_Jun

从上图中可以看到Framewok的组成部分，它们的功能分别是：

1. Activity Manager：用于管理和协调所有Android应用程序的活动和任务。
2. Content Providers：允许Android应用程序之间共享数据。
3. Package Manager：用于安装，升级和管理应用程序，以及处理应用程序的权限。
4. Resource Manager：管理应用程序使用的资源，例如图像，字符串，布局。
5. Notification Manager：处理Android系统的通知机制。
6. Telephony Manager：提供电话功能，例如拨打电话，接听电话等。
7. Location Manager：用于获取设备的位置信息。
8. View System：提供用户界面的基本组件或部件，例如按钮，文本框等。
9. Window Manager：处理屏幕上的窗口，例如在屏幕上绘制UI元素和管理窗口焦点。
10. Package Installer：用于在设备上安装应用程序的控制面板。
11. Resource Manager：管理所有允许应用程序访问的公共资源，例如铃声，照片和联系人信息。
12. Activity和Fragment：提供应用程序的用户界面和控制器。

可以看到前文中的各种系统服务就是属于Framework中的一部分，但是用户层并不能直接访问系统服务提供的功能，而是通过各服务对应的管理器来对系统服务进行调用。接下来开始跟踪，在开发应用中，当调用一个系统服务功能时发生了哪些调用，使用Android Studio创建一个项目，添加如下代码。

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    TelephonyManager tm = (TelephonyManager)
this.getSystemService(TELEPHONY_SERVICE);
    /*
         * 电话状态:
         * 1.tm.CALL_STATE_IDLE=0     无活动
         * 2.tm.CALL_STATE_RINGING=1  响铃
         * 3.tm.CALL_STATE_OFFHOOK=2  摘机
         */
    int state= tm.getCallState();//int
    Log.i("MainActivity","phone state "+state);
}
```

通过getSystemService函数提供了一个系统服务的名称，获取到了对应系统服务对应管理器，通过调用管理器的函数来触发对应系统服务的功能，看看具体是如何获取到系统服务的。找到Android源码中Activity.java文件。

```
public Object getSystemService(@ServiceName @NonNull String name) {
    if (getBaseContext() == null) {
        throw new IllegalStateException(
            "System services not available to Activities before onCreate()");
    }

    if (WINDOW_SERVICE.equals(name)) {
        return mWindowManager;
    } else if (SEARCH_SERVICE.equals(name)) {
        ensureSearchManager();
```

```
            return mSearchManager;
        }
        return super.getSystemService(name);
    }
```

如果是WINDOW_SERVICE或者SEARCH_SERVICE就快速的返回对应的管理器了，其他系统服务则继续调用父类的函数。Activity继承自ContextThemeWrapper，找到对应实现代码如下。

```
public Object getSystemService(String name) {
    if (LAYOUT_INFLATER_SERVICE.equals(name)) {
        if (mInflater == null) {
            mInflater =
LayoutInflater.from(getBaseContext()).cloneInContext(this);
        }
        return mInflater;
    }
    return getBaseContext().getSystemService(name);
}
```

找到ContextImpl中的对应实现

```
public Object getSystemService(String name) {
    ...
    return SystemServiceRegistry.getSystemService(this, name);
}
```

继续查看SystemServiceRegistry中的实现

```
public static Object getSystemService(ContextImpl ctx, String name) {
    if (name == null) {
        return null;
    }
    final ServiceFetcher<?> fetcher = SYSTEM_SERVICE_FETCHERS.get(name);
    if (fetcher == null) {
        if (sEnableServiceNotFoundWtf) {
            Slog.wtf(TAG, "Unknown manager requested: " + name);
        }
        return null;
    }
    final Object ret = fetcher.getService(ctx);
    ...
    return ret;
}
```

发现服务是从SYSTEM_SERVICE_FETCHERS中获取出来，然后返回的。看看这个对象的值是如何插进去的。搜索该对象的put函数调用处找到相关函数如下。

```
private static <T> void registerService(@NonNull String serviceName,
                                        @NonNull Class<T> serviceClass, @NonNull
ServiceFetcher<T> serviceFetcher) {
    SYSTEM_SERVICE_NAMES.put(serviceClass, serviceName);
    SYSTEM_SERVICE_FETCHERS.put(serviceName, serviceFetcher);
    SYSTEM_SERVICE_CLASS_NAMES.put(serviceName, serviceClass.getSimpleName());
}
```

从名字就能看的出来，这是一个注册系统服务的函数，在该函数中对大多数系统服务进行注册，想要查找到一个系统服务，可以顺着`registerService`注册函数进行跟踪，如果添加一个自定义的系统服务，同样也是需要在这里进行系统服务的注册。

下面继续观察`TelephonyManager`中`getCallState`函数的实现。

```
public @CallState int getCallState() {
    if (mContext != null) {
        TelecomManager telecomManager =
mContext.getSystemService(TelecomManager.class);
        if (telecomManager != null) {
            return telecomManager.getCallState();
        }
    }
    return CALL_STATE_IDLE;
}
```

这里又通过另一个管理器进行的函数调用，继续跟进去

```
public @CallState int getCallState() {
    ITelecomService service = getTelecomService();
    if (service != null) {
        try {
            return service.getCallStateUsingPackage(mContext.getPackageName(),
                                                    mContext.getAttributionTag());
        } catch (RemoteException e) {
            Log.d(TAG, "RemoteException calling getCallState().", e);
        }
    }
    return TelephonyManager.CALL_STATE_IDLE;
}
```

上述代码可以看出，`TelephonyManager`管理器不负责业务相关的处理，主要是调用对应的系统服务来获取结果。继续查看`getCallStateUsingPackage`函数实现

```
public int getCallStateUsingPackage(String callingPackage, String
callingFeatureId) {
    try {
```

```
            Log.startSession("TSI.getCallStateUsingPackage");
            if (CompatChanges.isChangeEnabled(
                TelecomManager.ENABLE_GET_CALL_STATE_PERMISSION_PROTECTION,
callingPackage,
                Binder.getCallingUserHandle())) {
                // Bypass canReadPhoneState check if this is being called from SHELL
UID
                if (Binder.getCallingUid() != Process.SHELL_UID && !canReadPhoneState(
                    callingPackage, callingFeatureId, "getCallState")) {
                    throw new SecurityException("getCallState API requires
READ_PHONE_STATE"
                                                + " for API version 31+");
                }
            }
            synchronized (mLock) {
                return mCallsManager.getCallState();
            }
        } finally {
            Log.endSession();
        }
    }
```

在系统服务中就看到了管理状态相关的具体业务代码了，继续观察`mCallsManager.getCallStae`的实现

```
int getCallState() {
    return mPhoneStateBroadcaster.getCallState();
}
```

最后是由`PhoneStateBroadcaster`对象维护着电话的状态信息了，`PhoneStateBroadcaster`是`Android`中的一个系统广播机制，它用于在电话状态发生变化时发出通知，以便其他组件和应用程序能够接收和处理这些变化。它可以发出包括新来电，挂断电话，拨号等状态变化的通知，以使系统中的其他组件能够更新和处理这些变化。`PhoneStateBroadcaster`还提供了一些其他的功能，例如电话状态监控，用于检测电话状态的变化，以便能够及时响应。简单的贴一下相关的代码如下。

```
final class PhoneStateBroadcaster extends CallsManagerListenerBase {
    ...
    @Override
    public void onCallStateChanged(Call call, int oldState, int newState) {
        if (call.isExternalCall()) {
            return;
        }
        updateStates(call);
    }

    @Override
    public void onCallAdded(Call call) {
        if (call.isExternalCall()) {
            return;
        }
```

```
            updateStates(call);

            if (call.isEmergencyCall() && !call.isIncoming()) {
                sendOutgoingEmergencyCallEvent(call);
            }
        }

        @Override
        public void onCallRemoved(Call call) {
            if (call.isExternalCall()) {
                return;
            }
            updateStates(call);
        }

        @Override
        public void onExternalCallChanged(Call call, boolean isExternalCall) {
            updateStates(call);
        }

        private void updateStates(Call call) {
            int callState = TelephonyManager.CALL_STATE_IDLE;
            if (mCallsManager.hasRingingOrSimulatedRingingCall()) {
                callState = TelephonyManager.CALL_STATE_RINGING;
            } else if (mCallsManager.getFirstCallWithState(CallState.DIALING,
    CallState.PULLING,
                    CallState.ACTIVE, CallState.ON_HOLD) != null) {
                callState = TelephonyManager.CALL_STATE_OFFHOOK;
            }
            sendPhoneStateChangedBroadcast(call, callState);
        }

        int getCallState() {
            return mCurrentState;
        }

        private void sendPhoneStateChangedBroadcast(Call call, int phoneState) {
            if (phoneState == mCurrentState) {
                return;
            }

            mCurrentState = phoneState;
            ...
        }
        ...
    }
```

3.10 了解libcore

libcore是Android平台下的Java核心库，主要提供与Java语言核心相关的类，如Object类、String类，Java集合类以及输入/输出流等。同时，libcore还包括了平台支持库，提供了一些用于Android平台特定功能

的实现，如Socket、SSL、File、URI等类的平台特定实现。在Android应用程序开发中，libcore库是必不可少的一部分，其提供的类和实现对于开发和调试应用程序都具有非常重要的作用。

在libcore库中，luni是其中的一个子库，是指Java的基础类库（LUNI = LANG + UTIL + NET + IO），而ojluni是OpenJDK的代码在Android中的实现，其目录结构与luni子库类似，包含了Java语言核心类、Java集合类和I/O类等。ojluni是在Java标准库的基础上进行了一些定制化的修改，以便更好地适配Android系统。下面看看ojluni的目录结构。

```
tree ./libcore/ojluni/src/main/java/ -d

├── com
│   └── sun
│       ├── net
│       │   └── ssl
│       │       └── internal
│       │           └── ssl
│       ├── nio
│       │   └── file
│       └── security
│           └── cert
│               └── internal
│                   └── x509
├── java
│   ├── awt
│   │   └── font
│   ├── beans
│   ├── io
│   ├── lang
│   │   ├── annotation
│   │   ├── invoke
│   │   ├── ref
│   │   └── reflect
│   ├── math
│   ├── net
│   ├── nio
│   │   ├── channels
│   │   │   └── spi
│   │   ├── charset
│   │   │   └── spi
│   │   └── file
│   │       ├── attribute
│   │       └── spi
│   ├── security
│   │   ├── acl
│   │   ├── cert
│   │   ├── interfaces
│   │   └── spec
│   ├── sql
│   ├── text
│   ├── time
│   │   ├── chrono
│   │   ├── format
```

```
│   │   ├── temporal
│   │   └── zone
│   └── util
│       ├── concurrent
│       │   ├── atomic
│       │   └── locks
│       ├── function
│       ├── jar
│       ├── logging
│       ├── prefs
│       ├── regex
│       ├── stream
│       └── zip
├── javax
│   ├── crypto
│   │   ├── interfaces
│   │   └── spec
│   ├── net
│   │   └── ssl
│   ├── security
│   │   ├── auth
│   │   │   ├── callback
│   │   │   ├── login
│   │   │   └── x500
│   │   └── cert
│   └── sql
│       └── rowset
├── jdk
│   ├── internal
│   │   ├── util
│   │   └── vm
│   │       └── annotation
│   └── net
└── sun
    ├── invoke
    │   └── util
    ├── misc
    ├── net
    │   ├── ftp
    │   │   └── impl
    │   ├── spi
    │   │   └── nameservice
    │   ├── util
    │   └── www
    │       └── protocol
    │           ├── file
    │           ├── ftp
    │           └── jar
    ├── nio
    │   ├── ch
    │   ├── cs
    │   └── fs
    ├── reflect
    │   └── misc
```

```
    ├── security
    │   ├── action
    │   ├── jca
    │   ├── pkcs
    │   ├── provider
    │   │   └── certpath
    │   ├── timestamp
    │   ├── util
    │   └── x509
    └── util
        ├── calendar
        ├── locale
        │   └── provider
        ├── logging
        └── resources
```

3.11 了解sepolicy

sepolicy主要用来存放SELinux策略的目录，SELinux是一种强制访问控制机制，Android系统中实现访问控制的一种安全机制，它在Linux内核的基础上实现，用于保证手机安全。

SELinux主要用于限制应用程序的权限，使其只能访问其所需的资源，并在需要时向用户请求权限。通过限制应用程序的权限，可以防止恶意软件和攻击者攻击系统。具体而言，sepolicy可以做到以下几点：

1. 限制应用程序访问系统的资源，例如系统设置、网络接口等。
2. 限制应用程序的使用权限，例如读取联系人、访问存储空间等。
3. 保护系统文件和目录，防止应用程序和攻击者修改和删除系统关键文件。

Type Enforcement是SELinux中的一个安全策略机制，用于对系统中每个对象和主体的访问进行强制访问控制。在Type Enforcement的模型中，每个对象和主体都被赋予了一个安全上下文（Security Context），该上下文由多个标签组成。

Role-Based Access Control (RBAC) 也是 SELinux 中的一种访问控制机制，用于对系统中多个用户、角色和对象的访问进行授权和限制。在RBAC模型中，每个用户都被分配了一个或多个角色，每个角色都有一组特定的权限和访问控制规则。与传统的访问控制方式不同，RBAC可以根据用户的职责和角色来授权和限制其访问，并且可以通过添加或删除角色等方式对访问控制进行动态管理。这种机制可以确保系统中不同用户之间的隔离和资源保护，并提高系统的安全性和可靠性。

在SELinux中，标签分为三种类型：用户标签（User ID）、角色标签（Role）和类型标签（Type）。每个安全上下文都包含了这三种标签的组合，如“u:r:system_app:s0”表示该上下文对应一个用户标签为“system_app”的进程，其角色标签和类型标签分别为“r”和“s0”。

通过安全上下文，SELinux可以对系统中的对象和主体进行细粒度控制，并限制它们之间的交互。例如，如果两个对象或主体的安全上下文不匹配，则它们不能相互通信或共享资源。这种机制可以有效地防止恶意应用程序或者攻击者对系统进行攻击或滥用。也可以通过修改 sepolicy 目录下的文件来调整安全策略，从而适应不同的应用程序和系统需求。

在ROM定制时，常会添加某些功能时由于权限问题导致系统输出警告信息提示错误，这种情况需要调整安全策略。调整策略的位置在Android源代码的 ./system/sepolicy/ 目录中，public、private目录下。

1. public：该目录包含Android系统与函数库等公共的sepolicy规则。这些规则是开发人员和厂商可以自由使用和修改的，因为这些规则涉及到的是公共区域的访问控制。
2. private：该目录包含硬编码到Android系统中的特定规则。这些规则用于控制既定的Android系统功能和应用程序，例如拨号应用程序、电源管理等，因此这些规则不能被修改或覆盖。

当修改Android系统的SELinux策略时，系统会使用prebuilts目录中的策略进行对比，这是因为prebuilts中包含了在 Android设备上预置的SELinux策略和规则。

对安全策略有一个大致的了解后，先看一个简单的例子，找到文件./system/sepolicy/public/adbd.te

```
# 定义类型
type adbd, domain;

# 允许adbd类型的进程，在类型shell_test_data_file中的目录内创建子目录
allow adbd shell_test_data_file:dir create_dir_perms;
```

这里使用了三个类型：

- adbd：指定进程的类型；
- domain：指定域的类型；
- shell_test_data_file：指定目录的类型。

规则使用了allow关键字，表示允许某些操作。具体来说，上述规则允许adbd类型的进程在shell_test_data_file类型的目录下创建目录，并且该目录将被赋予允许创建子目录的权限（由create_dir_perms定义）。

这个规则的实际意义是，当adbd进程需要在shell_test_data_file目录下创建子目录时，允许该操作，并为新创建的目录设置适当的权限。注意，这个规则只对该目录有效，不能用于其他目录。

通常情况下采用按需修改的方式调整安全策略，当添加的功能被安全策略拦住时，会输出警告提示。例如在文件com_android_internal_os_Zygote.cpp的SpecializeCommon函数中加入如下代码，访问data目录。

```
std::string filepath="/data/app/demo";
ReadFileToString(filepath,&file_contents)
```

然后就会被SELinux拦截并提示警告信息如下

```
avc: denied { search } for name="app" dev="dm-8" ino=100 scontext=u:r:zygote:s0
tcontext=u:object_r:apk_data_file:s0 tclass=dir permissive=0
```

在SELinux中，avc: denied是出现最频繁的提示之一，根据提示可以知道，进程zygote对安全上下文为u:object_r:apk_data_file:s0的目录进行search操作，该行为被拒绝了。除此之外，还有其他拒绝访问的提示消息如下。

- avc: denied {open} - 表示进程被禁止打开文件或设备。

- avc: denied {read} - 表示进程被禁止读取一个文件、设备或目录。
- avc: denied {write} - 表示进程被禁止写入一个文件、设备或目录。
- avc: denied {getattr} - 表示进程被禁止读取一个文件或目录的元数据（例如，所有权、组、权限等）。
- avc: denied {execute} - 表示进程被禁止执行一个文件或进程。
- avc: denied {create} - 表示进程被禁止创建一个文件。
- avc: denied {search} - 表示此进程被禁止在某目录中搜索文件的操作

除了 avc: denied之外，还有其他一些可能出现的提示信息。以下是一些常见提示信息以及它们的含义：

- avc: granted - 操作被允许。
- avc: audit - 正在监视执行上下文之间的交互，并将相关信息记录到审计日志中。
- avc: no audit - 没有记录此操作的详细信息，这通常是因为没有启用SELinux的审计功能。
- avc: invalid - 操作请求的权限非法或无效。
- avc: timeout - SELinux规则分析器超时无法确定操作是否应该允许。在这种情况下，操作通常会被拒绝。
- avc: failed - SELinux规则分析器无法确定操作是否应该被允许或拒绝。

在SELinux中，scontext代表系统中的安全上下文，tcontext代表对象的安全上下文。每个具有权限要求的进程和对象都有一个安全上下文。SELinux使用这些安全上下文来进行访问控制决策。

scontext和tcontext中的“u”，“r”和“s0”是安全上下文标记的不同部分。含义如下：

- u - 代表selinux中定义的用户，tcontext中的u代表对象所属用户。
- r - 代表进程的角色（role），tcontext中的r代表对象的角色。
- s0 - 代表进程的安全策略范围（security level），tcontext中的s0代表对象的安全策略范围。s0通常表示为默认值。

可以通过命令ps -eZ来查看进程的scontext。

```
ps -eZ

u:r:servicemanager:s0          system         672     1 10860740  3784 SyS_epoll+
0 S servicemanager
u:r:hwservicemanager:s0        system         673     1 10880928  4648 SyS_epoll+
0 S hwservicemanager
u:r:kernel:s0                  root           674     2       0      0 worker_th+
0 S [kworker/7:1H]
u:r:vndservicemanager:s0       system         675     1 10813436  2884 SyS_epoll+
0 S vndservicemanager
u:r:kernel:s0                  root           676     2       0      0 kthread_w+
0 S [psimon]
```

可以通过命令ls -Z来查看文件的scontext

```
cd /data/app
ls -Z -all

drwxrwxr-x  3 system system u:object_r:apk_data_file:s0          3488 2023-02-26
21:50:57.968696920 +0800 ~~QZ-rYHaywe6nr2ryYn3UoQ==
drwxrwxr-x  3 system system u:object_r:apk_data_file:s0          3488 2023-03-02
22:12:29.802016689 +0800 ~~W9dmzmphiDsjJm79RiBwdg==
```

重新对下面的这个提示进行一次解读。selinux拒绝搜索一个目录，目录名称为app，所在设备为dm-8，被拒绝的进程上下文特征是u:r:zygote:s0，角色是zygote，目标文件上下文特征是u:object_r:apk_data_file:s0，用户级别为object_r，文件的所属类型是apk_data_file，表示应用程序的数据文件。tclass表示请求对象的类型，dir为目录，file表示文件

```
avc: denied { search } for name="app" dev="dm-8" ino=100 scontext=u:r:zygote:s0
tcontext=u:object_r:apk_data_file:s0 tclass=dir permissive=0
```

解读完成后，可以开始调整安全策略了，找到文件system/sepolicy/private/zygote.te，然后添加策略如下

```
allow zygote apk_data_file:dir search;
```

修改完成后编译时，会报错，提示diff对比文件时发现内容不一致。最后再将文件system/sepolicy/prebuilts/api/31.0/private/zygote.te下添加相同的策略即可成功编译。

neverallow是SELinux策略语言中的一个规则，它用于指定某个操作永远不允许执行。neverallow规则用于设置一些强制访问控制规则，以在安全策略中明确禁止某些行为，从而提高其安全性。neverallow规则与allow规则在语法上非常相似，但在作用上截然不同。

有时按照警告信息提示，添加了对应策略后无法编译通过提示违反了neverallow。这种情况可以找到对应的neverallow，进行修改添加一个白名单来放过添加的规则。例如下面这个例子

```
  neverallow {
    coredomain
    -fsck
    -init
    -ueventd
    -zygote
  } device:{ blk_file file } no_rw_file_perms;
```

这个规则禁止上述进程以可读可写权限读写 device 类型的文件，其中-zygote，这种前面带有-表示排除掉这种进程，如果被设置了永不允许，只要找到对应的设置处，添加上排除对应进程即可成功编译了。

3.12 了解Linker

Linker是安卓中的一个系统组件，负责加载和链接系统动态库文件。

在Android源代码中，Linker源码的主要目录是bionic/linker。该目录包含Linker的核心实现，如动态加载、符号表管理、重定位、符号解析、SO文件搜索等。其中，linker.c是Linker的主要入口点，该文件中包含了大量的实现细节。linker_phdr.c是负责加载和处理ELF格式库文件的代码，linker_namespaces.cpp负责管理命名空间的代码，linker_relocs.cpp负责处理重定位的代码，linker_sleb128.cpp和linker_uleb128.cpp负责压缩和解压缩数据的实现等。除了bionic/linker目录外，Linker相关的代码还分散在其他系统组件中，例如系统服务和应用程序框架。

linker提供的一些函数来操作动态库，相关函数如下。

1. dlopen：打开一个动态链接库并返回句柄。
2. dlsym：查找动态链接库中符号的地址。
3. dlclose：关闭先前打开的动态链接库。
4. dlerror：返回最近的动态链接库错误。
5. dladdr：根据一个内存地址，返回映射到该地址的函数或变量的信息。
6. dl_iterate_phdr：遍历进程的动态链接库模块，可以获取模块地址、同名模块列表等信息。

在开始了解Linker如何加载动态库so文件前，需要先对so文件有一个简单的了解。

3.12.1 ELF文件格式

在Android中，so (Shared Object) 动态库是一种是一种基于ELF格式 (Executable and Linkable Format) 的可执行文件，它包含已编译的函数和数据，可以在运行时被加载到内存中，并被多个应用程序或共享库使用。

与静态库不同，动态库中的代码在可执行文件中并不存在，取而代之的是一些动态链接器（Linker）编译时不知道的外部引用符号。在运行时，Linker会根据动态库中的符号表来解析这些引用，并将动态库中的函数和数据链接到可执行程序中。

进程间共享动态库可以大大减少内存使用，提高代码重用性和可维护性。例如，如果多个应用程序都需要使用同一组件库，可以将其实现作为共享库提供。这样一来，每个应用程序都可以使用同一份库，而不必将代码重复添加到每个应用程序中。

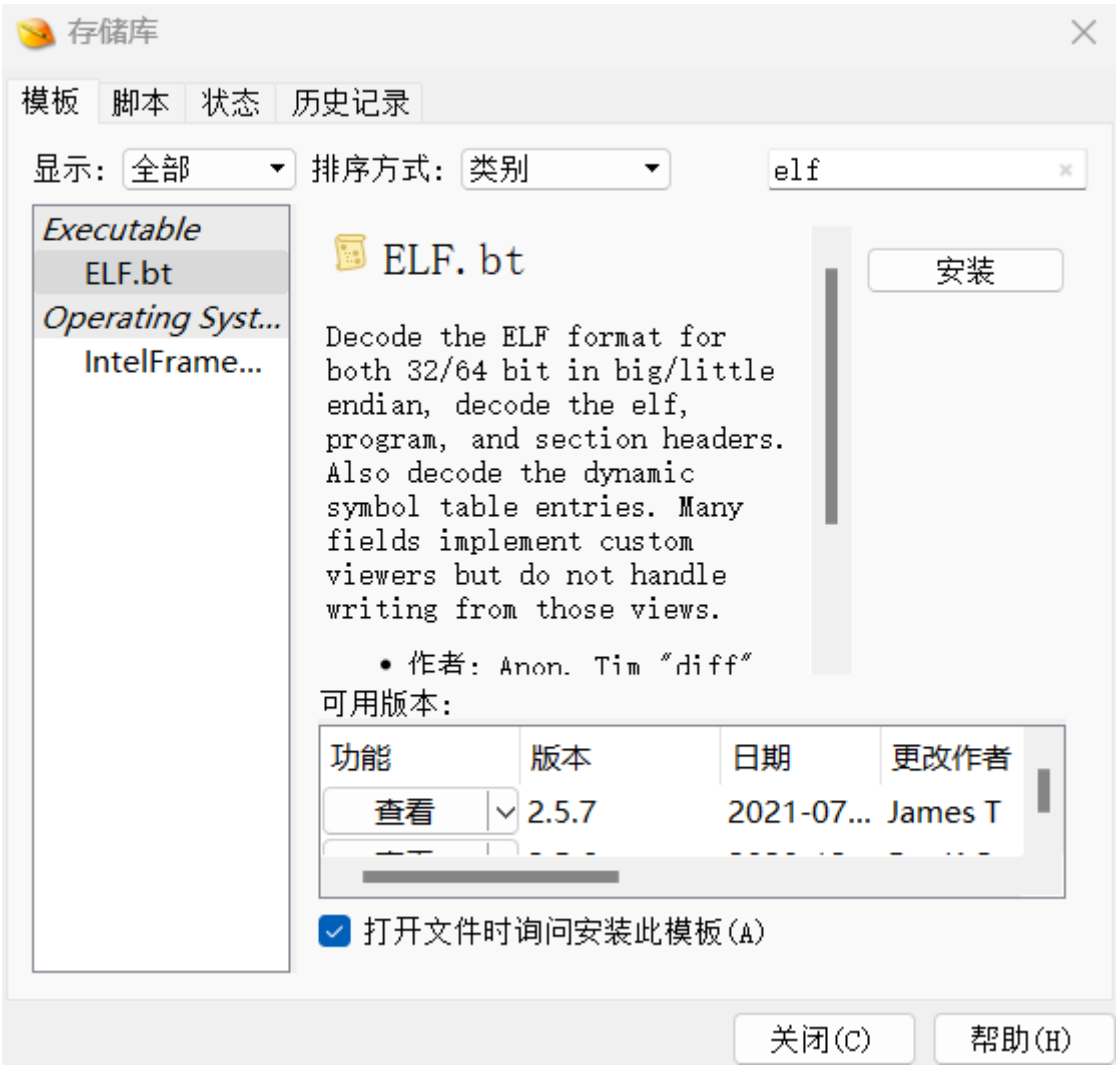
在ELF文件结构中，包含以下三个部分：

1. ELF Header，ELF文件头，包含了文件的基本信息，例如文件类型、程序入口地址、节表的位置和大小等。
2. Section Header，节头部分，描述了文件中各个节的大小、类型和位置等信息。ELF文件中的每个节都包含某种类型的信息，例如代码、数据、符号表、重定位表以及其他调试信息等。

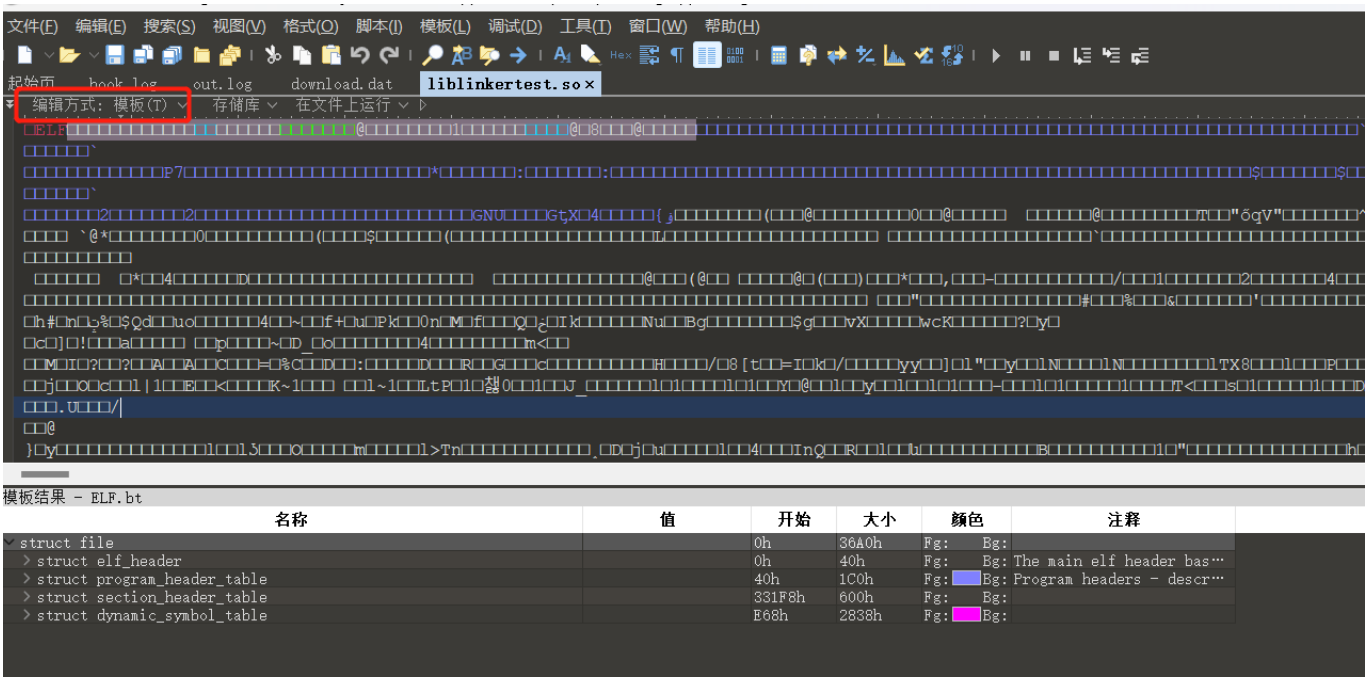
3. Program Header，段头部分，描述了可执行文件在内存中的布局。由于ELF文件的节可以以任意顺序排列，因此Linker在加载前需要使用Program Header来释放并映射虚拟内存，创建进程虚拟内存段布局。Program Header也包含了动态链接器所需的信息，例如动态库的位置、依赖关系和符号表位置等。

使用Android Studio创建一个Native C++的项目，成功编译后来到output目录中，解压app-debug.apk文件，然后进入app-debug\lib\arm64-v8a\目录，找到so文件将其拖入010 Editor编辑器工具中。

接着给010 Editor编辑器安装一个ELF格式解析的模板，在工具栏找到模板->模板存储库。搜索ELF，点击安装，操作见下图。



模板安装后，关闭文件，重新使用010 Editor打开后，将编辑方式切换为模板后，就能成功看到使用ELF格式解析so文件的结果了，如下图。



ELF头部定义了ELF文件的基本属性和结构，也为后续的段表和节表等信息提供了重要的指导作用。加载ELF文件的第一步就是解析ELF头部后，再根据头部信息去解析其他部分的数据，ELF头部（elf_header）结构包含以下成员：

- e_ident：长度为 16 字节的数组，用于标识文件类型和文件版本等信息。
- e_type: ELF文件类型，如可执行文件、共享库、目标文件等等。
- e_machine：目标硬件架构。
- e_version：ELF文件的版本，其一般为EV_CURRENT。
- e_entry：程序入口点的虚拟地址。
- e_phoff：程序头表（program header table）的偏移量（以字节为单位）。
- e_shoff：节头表（section header table）的偏移量（以字节为单位）。
- e_flags：表示一些标志，比如针对硬件进行微调的标志。
- e_ehsize：ELF头部的长度（以字节为单位）。
- e_phentsize：程序头表中一个入口的长度（以字节为单位）。
- e_phnum：程序头表中入口的数量。
- e_shentsize：节头表中一个入口的长度（以字节为单位）。
- e_shnum：节头表中入口的数量。
- e_shstrndx：节头表中节名称字符串表的索引。

下图是010 Edtior解析展示的结果图。

模板结果 - ELF.bt

名称	值	开始	大小	颜色	注释
struct file		0h	36A0h	Fg: Bg:	
struct elf_header		0h	40h	Fg: Bg:	The main elf header bas…
struct e_ident_t e_ident		0h	10h	Fg: Bg:	Magic number and other …
enum e_type64_e e_type	ET_DYN (3)	10h	2h	Fg: Bg:	Object file type
enum e_machine64_e e_machine	183	12h	2h	Fg: Bg:	Architecture
enum e_version64_e e_version	EV_CURRENT (1)	14h	4h	Fg: Bg:	Object file version
Elf64_Addr e_entry_START_ADDRESS	0x000000000000E4A0	18h	8h	Fg: Bg:	Entry point virtual add…
Elf64_Off e_phoff_PROGRAM_HEADER_OFFSET_IN_FILE	64	20h	8h	Fg: Bg:	Program header table fi…
Elf64_Off e_shoff_SECTION_HEADER_OFFSET_IN_FILE	209400	28h	8h	Fg: Bg:	Section header table fi…
Elf32_Word e_flags	0	30h	4h	Fg: Bg:	Processor-specific flags
Elf64_Half e_ehsize_ELF_HEADER_SIZE	64	34h	2h	Fg: Bg:	ELF Header size in bytes
Elf64_Half e_phentsize_PROGRAM_HEADER_ENTRY_SIZE_IN_FILE	56	36h	2h	Fg: Bg:	Program header table en…
Elf64_Half e_phnum_NUMBER_OF_PROGRAM_HEADER_ENTRIES	8	38h	2h	Fg: Bg:	Program header table en…
Elf64_Half e_shentsize_SECTION_HEADER_ENTRY_SIZE	64	3Ah	2h	Fg: Bg:	Section header table en…
Elf64_Half e_shnum_NUMBER_OF_SECTION_HEADER_ENTRIES	24	3Ch	2h	Fg: Bg:	Section header table en…
Elf64_Half e_shtrndx_STRING_TABLE_INDEX	23	3Eh	2h	Fg: Bg:	Section header string t…
struct program_header_table		40h	1C0h	Fg: Bg:	Program headers - descr…
struct section_header_table		331F8h	600h	Fg: Bg:	
struct dynamic_symbol_table		E68h	2838h	Fg: Bg:	

`program header table`是一种用于描述可执行文件和共享库的各个段（`section`）在进程内存中的映射关系的结构，也称为段表。每个程序头表入口表示一个段。在`Linux`系统中，它是被操作系统用于将`ELF`文件加载到进程地址空间的重要数据结构之一。每个`program header table`具有相同的固定结构，相关字段如下：

- `p_type`：指定该段的类型，如可执行代码、只读数据、可读写数据、动态链接表、注释等等。
- `p_offset`：该段在`ELF`文件中的偏移量（以字节为单位）。
- `p_vaddr`：该段在进程虚拟地址空间中的起始地址。
- `p_paddr`：该项通常与`p_vaddr`相等。用于操作系统在将`ELF`文件的一个段映射到进程地址空间前，进行虚拟地址和物理地址的转换等操作。
- `p_filesz`：该段在文件中的长度（以字节为单位）。
- `p_memsz`：该段在加到进程地址空间后的长度（以字节为单位）。
- `p_flags`：用于描述该段的标志，如可读、可写、可执行、不可缓存等等。
- `p_align`：对于某些类型的段，该字段用于指定段在地址空间中的对齐方式。

下图是编辑器中解析so看到的值

模板结果 - ELF.bt

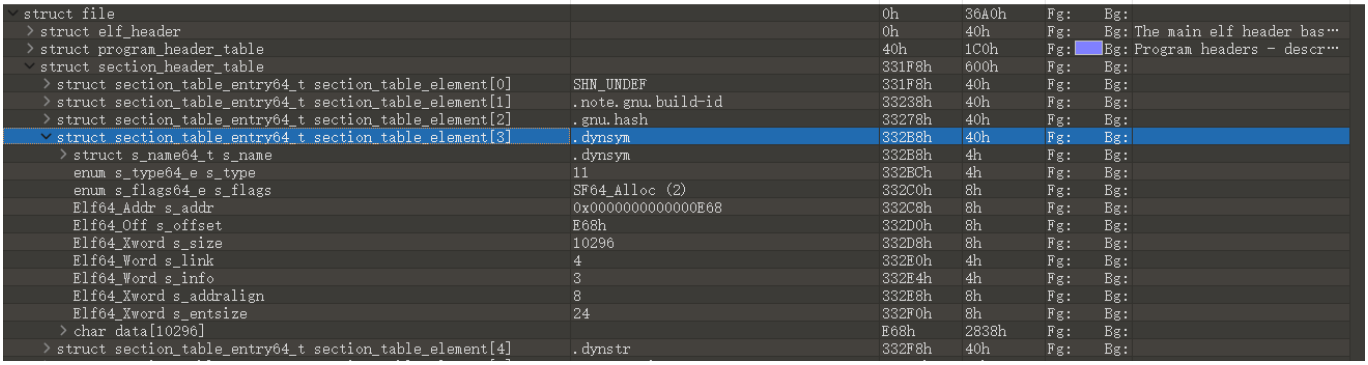
名称	值	开始	大小	颜色	注释
struct file		0h	36A0h	Fg: Bg:	
struct elf_header		0h	40h	Fg: Bg:	The main elf header bas…
struct program_header_table		40h	1C0h	Fg: Bg:	Program headers - descr…
struct program_table_entry64_t program_table_element[0]	(R_X) Loadable Segment	40h	38h	Fg: Bg:	
enum p_type64_e p_type	PT_LOAD (1)	40h	4h	Fg: Bg:	Segment type
enum p_flags64_e p_flags	PF_Read_Exec (5)	44h	4h	Fg: Bg:	Segment attributes
Elf64_Off p_offset_FROM_FILE_BEGIN	0h	48h	8h	Fg: Bg:	Segment offset in file
Elf64_Addr p_vaddr_VIRTUAL_ADDRESS	0x0000000000000000	50h	8h	Fg: Bg:	Segment virtual address
Elf64_Addr p_paddr_PHYSICAL_ADDRESS	0x0000000000000000	58h	8h	Fg: Bg:	Reserved (Segment physi…
Elf64_Xword p_filesz_SEGMENT_FILE_LENGTH	193532	60h	8h	Fg: Bg:	Segment size in file
Elf64_Xword p_memsz_SEGMENT_RAM_LENGTH	193532	68h	8h	Fg: Bg:	Segment size in ram
Elf64_Xword p_align	4096	70h	8h	Fg: Bg:	Segment alignment
struct program_table_entry64_t program_table_element[1]	(RW_) Loadable Segment	78h	38h	Fg: Bg:	
struct program_table_entry64_t program_table_element[2]	(RW_) Dynamic Segment	B0h	38h	Fg: Bg:	
struct program_table_entry64_t program_table_element[3]	(R__) Note	E8h	38h	Fg: Bg:	
struct program_table_entry64_t program_table_element[4]	(R__) Note	120h	38h	Fg: Bg:	
struct program_table_entry64_t program_table_element[5]	(R__) GCC .eh_frame_hdr Segment	158h	38h	Fg: Bg:	
struct program_table_entry64_t program_table_element[6]	(RW_) GNU Stack (executability)	190h	38h	Fg: Bg:	
struct program_table_entry64_t program_table_element[7]	(R__) GNU Read-only After Relocation	1C8h	38h	Fg: Bg:	
struct section_header_table		331F8h	600h	Fg: Bg:	
struct dynamic_symbol_table		E68h	2838h	Fg: Bg:	

`section header table`（节头表）是用于描述`ELF`文件中所有节（`section`）的元信息列表，也称为节表。它包含了每个节在文件中的位置、大小、类型、属性等信息。节头表的中相关字段如下：

- `sh_name`: 节的名字在`.shstrtab`节中的向偏移量。这个偏移量可以用于获取该节的名字。
- `sh_type`：节的类型（`type`），如代码段、数据段、符号表等。

- sh_flags：节的属性标志，如是否可读、可写、可执行等。
- sh_addr：节的内存地址（virtual address），当这个地址为零时，表示这个节没有被加载到内存中。
- sh_offset：节在ELF文件中的偏移量（offset）。
- sh_size：节的长度（size）属性。
- sh_link：节的连接节（linking section），可以帮助定位一些节，如符号表。
- sh_info：与sh_link一起使用，具体含义与sh_link的值有关。
- sh_addralign：节的对齐方式（alignment）。
- sh_entsize：节的entry的大小。

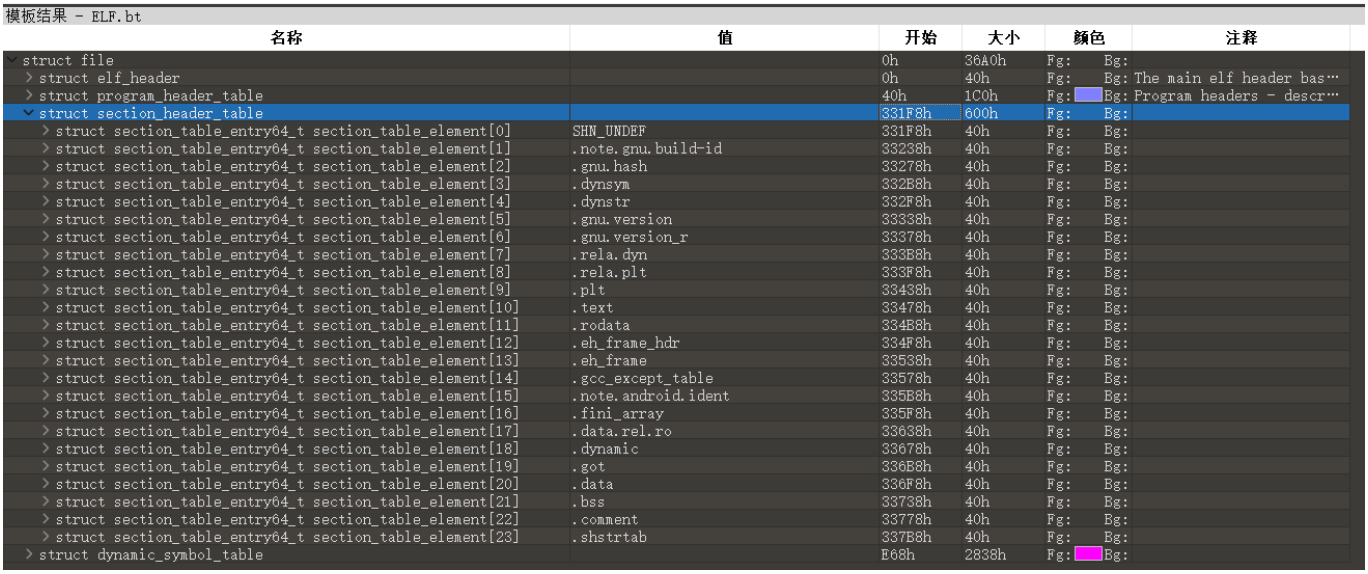
通过这些信息，section header table可以为执行链接和动态加载提供必要的元数据信息。样例数据看下图



struct file		0h	36A0h	Fg:	Bg:
struct elf_header		0h	40h	Fg:	Bg: The main elf header bas…
struct program_header_table		40h	1C0h	Fg:	Bg: Program headers - descr…
struct section_header_table		331F8h	600h	Fg:	Bg:
struct section_table_entry64_t section_table_element[0]	SHN_UNDEF	331F8h	40h	Fg:	Bg:
struct section_table_entry64_t section_table_element[1]	.note.gnu.build-id	33238h	40h	Fg:	Bg:
struct section_table_entry64_t section_table_element[2]	.gnu.hash	33278h	40h	Fg:	Bg:
struct section_table_entry64_t section_table_element[3]	.dynsym	332B8h	40h	Fg:	Bg:
struct s_name64_t s_name	.dynsym	332B8h	4h	Fg:	Bg:
enum s_type64_e s_type	11	332BCh	4h	Fg:	Bg:
enum s_flags64_e s_flags	SF64_Alloc (2)	332C0h	8h	Fg:	Bg:
Elf64_Addr s_addr	0x0000000000000E68	332C8h	8h	Fg:	Bg:
Elf64_Off s_offset	E68h	332D0h	8h	Fg:	Bg:
Elf64_Xword s_size	10296	332D8h	8h	Fg:	Bg:
Elf64_Word s_link	4	332E0h	4h	Fg:	Bg:
Elf64_Word s_info	3	332E4h	4h	Fg:	Bg:
Elf64_Xword s_addralign	8	332E8h	8h	Fg:	Bg:
Elf64_Xword s_entsize	24	332F0h	8h	Fg:	Bg:
char data[10296]		E68h	2838h	Fg:	Bg:
struct section_table_entry64_t section_table_element[4]	.dynstr	332F8h	40h	Fg:	Bg:

ELF文件中有各种节用于存放对应的信息，几个常见的节点存放数据的描述如下。

- .dynsym 节：该节包含动态链接符号表（dynamic symbol table），用于描述.so文件所包含的动态链接库中的符号。符号是程序中一些命名实体的名称，例如函数、变量、常量等等，描述了这些实体在程序中的地址和大小等信息。.dynsym 节可以协助动态加载器（Dynamic Linker）在程序运行时逐个查找符号。
- .dynstr 节：用于存放符号表中的字符串，包括函数名、变量名、库名等等。
- .plt 节：保存了远程函数调用实现的跳转代码。
- .rodata 节：包含程序中只读数据的代码段，如字符串常量、全局常量等等。
- .text 节：程序的主要代码存放在该节中。该节包含可执行代码的机器语言指令，例如函数代码、条件语句、循环语句等等。
- .bss 节点（Block Started by Symbol）存储未初始化的全局变量和静态变量，其大小在编译时无法确定。因此，.bss节点在ELF文件中只占据一些空间，该空间称为bss段。而在运行时，操作系统会分配实际的内存空间给这些变量。.bss节点的大小在 ELF文件头的e_shsize字段中给出。
- .shstrtab 节点（Section Header String Table）存储节名称字符串，即每个节的名称和节头表中的节名称偏移量。它包含了ELF文件中每个节的字符串名称，方便读取程序在加载时快速访问。在Android中，.shstrtab节点是一个特殊的节，它位于节头表的末尾，可以通过ELF文件头的e_shstrndx字段找到。



模板结果 - ELF.bt

名称	值	开始	大小	颜色	注释
struct file		0h	36A0h	Fg: Bg:	
struct elf_header		0h	40h	Fg: Bg:	The main elf header bas…
struct program_header_table		40h	1C0h	Fg: Bg:	Program headers - descr…
struct section_header_table		331F8h	600h	Fg: Bg:	
struct section_table_entry64_t section_table_element[0]	SHN_UNDEF	331F8h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[1]	.note.gnu.build-id	33238h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[2]	.gnu.hash	33278h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[3]	.dynsym	332B8h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[4]	.dynstr	332F8h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[5]	.gnu.version	33338h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[6]	.gnu.version_r	33378h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[7]	.rela.dyn	333B8h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[8]	.rela.plt	333F8h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[9]	.plt	33438h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[10]	.text	33478h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[11]	.rodata	334B8h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[12]	.eh_frame_hdr	334F8h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[13]	.eh_frame	33538h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[14]	.gcc_except_table	33578h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[15]	.note.android.ident	335B8h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[16]	.fini_array	335F8h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[17]	.data.rel.ro	33638h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[18]	.dynamic	33678h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[19]	.got	336B8h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[20]	.data	336F8h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[21]	.bss	33738h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[22]	.comment	33778h	40h	Fg: Bg:	
struct section_table_entry64_t section_table_element[23]	.shstrtab	337B8h	40h	Fg: Bg:	
struct dynamic_symbol_table		E68h	2838h	Fg: Bg:	

3.12.2 动态库加载流程

Linker动态库加载是把代码（函数、变量、数据结构等）从动态链接库（so文件）中加载到内存中，并建立起代码之间的相互引用关系的过程。在Android等操作系统中，Linker动态加载主要用于模块化开发，将程序分为多个独立的模块，以便于代码的管理和维护。下面是Linker动态加载的主要步骤：

1. 根据系统的运行时需求，将需要的库文件加载进内存中，实现代码重用和共享。此时，Linker会执行一些特定的逻辑，如依赖优化、so文件版本检查等。
2. 在进行动态链接的过程中，Linker会为每个库和每个函数生成全局唯一的标识符，以确定代码所在的地址。这个标识符会在编译过程中嵌入到库文件的头部，并且保存到动态链接库的符号表中。
3. 解析符号表。Linker会读取库文件的符号表，并把符号名和符号地址配对起来，以便于在程序运行期间在内存中动态地连接他们。
4. 检查符号表中的函数的其他库依赖项。如果当前库依赖于其他库，Linker就会递归地对这些依赖库进行加载、解析和链接。
5. 调整符号地址。Linker会修改符号表中的函数地址，将函数重定向到动态库中正确的位置，以确保函数调用能够正确地传递和接收数据。
6. 执行初始化和清理代码。在所有库和函数都被解析、链接和装载之后，Linker会执行全局构造函数来初始化代码，以及执行全局析构函数来清理代码。
7. Linker动态加载过程中还会涉及到如动态追加、卸载等操作。

以上是Linker动态加载的主要步骤及涉及到的主要逻辑。接着从源码层面跟踪动态加载的具体过程。打开前面创建的样例app。

```
public class MainActivity extends AppCompatActivity {
    // Used to load the 'linkertest' library on application startup.
    static {
        System.loadLibrary("linkertest");
    }
    ...
    public native String stringFromJNI();
}
```

在应用层直接通过调用loadLibrary就可以完成一系列的加载动态库的操作了，看看内部是如何实现的。首先是System下的loadLibrary函数，前文有介绍过libcore中存放着openjdk的核心库的实现，而java.lang.System就是其中，找到文件libcore/ojluni/src/main/java/java/lang/System.java查看函数实现如下。

```
public static void loadLibrary(String libname) {
    Runtime.getRuntime().loadLibrary0(Reflection.getCallerClass(), libname);
}
```

继续在ojluni的目录中搜索loadLibrary0的函数实现

```
void loadLibrary0(Class<?> fromClass, String libname) {
    ClassLoader classLoader = ClassLoader.getClassLoader(fromClass);
    loadLibrary0(classLoader, fromClass, libname);
}

private synchronized void loadLibrary0(ClassLoader loader, Class<?> callerClass,
String libname) {
    ...
    String libraryName = libname;
    // 如果classloader不是BootClassLoader
    if (loader != null && !(loader instanceof BootClassLoader)) {
        String filename = loader.findLibrary(libraryName);
        if (filename == null &&
            (loader.getClass() == PathClassLoader.class ||
             loader.getClass() == DelegateLastClassLoader.class)) {

            filename = System.mapLibraryName(libraryName);
        }
        ...
        String error = nativeLoad(filename, loader);
        if (error != null) {
            throw new UnsatisfiedLinkError(error);
        }
        return;
    }
    getLibPaths();
    String filename = System.mapLibraryName(libraryName);
    String error = nativeLoad(filename, loader, callerClass);
    if (error != null) {
        throw new UnsatisfiedLinkError(error);
    }
}

// 看到不管是哪个Classloader都是调用的nativeLoad，只是重载不一样。但是两个参数的实际也
是调用了三个参数重载的实现。
private static String nativeLoad(String filename, ClassLoader loader) {
    return nativeLoad(filename, loader, null);
}
```

```
// 三个参数重载的是一个native函数
private static native String nativeLoad(String filename, ClassLoader loader,
Class<?> caller);
```

继续搜索nativeLoad的相关实现如下

```
// 调用了JVM_NativeLoad
JNIEXPORT jstring JNICALL
Runtime_nativeLoad(JNIEnv* env, jclass ignored, jstring javaFilename,
                   jobject javaLoader, jclass caller)
{
    return JVM_NativeLoad(env, javaFilename, javaLoader, caller);
}
```

JVM_NativeLoad的代码在art目录中，继续查看相关实现

```
JNIEXPORT jstring JVM_NativeLoad(JNIEnv* env,
                                 jstring javaFilename,
                                 jobject javaLoader,
                                 jclass caller) {
  ScopedUtfChars filename(env, javaFilename);
  if (filename.c_str() == nullptr) {
    return nullptr;
  }

  std::string error_msg;
  {
    art::JavaVMExt* vm = art::Runtime::Current()->GetJavaVM();
    bool success = vm->LoadNativeLibrary(env,
                                         filename.c_str(),
                                         javaLoader,
                                         caller,
                                         &error_msg);
    if (success) {
      return nullptr;
    }
  }
  ...
}

// 继续找到相关实现
bool JavaVMExt::LoadNativeLibrary(JNIEnv* env,
                                  const std::string& path,
                                  jobject class_loader,
                                  jclass caller_class,
                                  std::string* error_msg) {
  error_msg->clear();
```

```
  SharedLibrary* library;
  Thread* self = Thread::Current();
  {
    MutexLock mu(self, *Locks::jni_libraries_lock_);
    library = libraries_->Get(path);
  }
  ...
  // 已经加载过的，存在则返回true了。
  if (library != nullptr) {
    ...
    return true;
  }

  ScopedLocalRef<jstring> library_path(env, GetLibrarySearchPath(env,
class_loader));

  Locks::mutator_lock_->AssertNotHeld(self);
  const char* path_str = path.empty() ? nullptr : path.c_str();
  bool needs_native_bridge = false;
  char* nativeloader_error_msg = nullptr;
  // 加载动态链接库
  void* handle = android::OpenNativeLibrary(
      env,
      runtime_->GetTargetSdkVersion(),
      path_str,
      class_loader,
      (caller_location.empty() ? nullptr : caller_location.c_str()),
      library_path.get(),
      &needs_native_bridge,
      &nativeloader_error_msg);
  VLOG(jni) << "[Call to dlopen(\"" << path << "\", RTLD_NOW) returned " << handle
<< "]";
  ...
  bool created_library = false;
  {
    std::unique_ptr<SharedLibrary> new_library(
        new SharedLibrary(env,
                          self,
                          path,
                          handle,
                          needs_native_bridge,
                          class_loader,
                          class_loader_allocator));

    MutexLock mu(self, *Locks::jni_libraries_lock_);
    library = libraries_->Get(path);
    // 将刚刚加载好的链接库保存起来
    if (library == nullptr) {  // We won race to get libraries_lock.
      library = new_library.release();
      libraries_->Put(path, library);
      created_library = true;
    }
  }
  ...
```

```
  bool was_successful = false;
  // 查找符号JNI_OnLoad
  void* sym = library->FindSymbol("JNI_OnLoad", nullptr);
  if (sym == nullptr) {
    ...
  } else {
    ScopedLocalRef<jobject> old_class_loader(env, env->NewLocalRef(self-
>GetClassLoaderOverride()));
    self->SetClassLoaderOverride(class_loader);

    VLOG(jni) << "[Calling JNI_OnLoad in \"" << path << "\"]";
    using JNI_OnLoadFn = int(*)(JavaVM*, void*);
    JNI_OnLoadFn jni_on_load = reinterpret_cast<JNI_OnLoadFn>(sym);
    // 调用JNI_OnLoad
    int version = (*jni_on_load)(this, nullptr);
    ...
  }
  library->SetResult(was_successful);
  return was_successful;
}
```

在这个函数中，看到使用OpenNativeLibrary来加载一个动态库，然后将加载动态库的信息包装成SharedLibrary对象，存入libraries_中，下次再加载时，会在libraries_查看是否存在，存在则直接返回。接着又通过函数FindSymbol查找JNI_OnLoad的符号地址，然后进行调用。继续跟踪加载动态库的具体实现，最后再回头看查找符号的实现。

```
void* OpenNativeLibrary(JNIEnv* env, int32_t target_sdk_version, const char* path,
                        jobject class_loader, const char* caller_location, jstring
library_path,
                        bool* needs_native_bridge, char** error_msg) {
#if defined(ART_TARGET_ANDROID)
  UNUSED(target_sdk_version);

  if (class_loader == nullptr) {
    ...
    void* handle = android_dlopen_ext(path, RTLD_NOW, &dlextinfo);
    if (handle == nullptr) {
        *error_msg = strdup(dlerror());
    }
    return handle;
    ...
    {
      Result<void*> handle = TryLoadNativeloaderExtraLib(path);
      if (!handle.ok()) {
        *error_msg = strdup(handle.error().message().c_str());
        return nullptr;
      }
      if (handle.value() != nullptr) {
```

```
          return handle.value();
        }
      }
      ...
      void* handle = OpenSystemLibrary(path, RTLD_NOW);
      if (handle == nullptr) {
        *error_msg = strdup(dlerror());
      }
      return handle;
    }
    ...
  #else
    for (const std::string& lib_path : library_paths) {
      ...
      void* handle = dlopen(path_arg, RTLD_NOW);
      if (handle != nullptr) {
        return handle;
      }
      if (NativeBridgeIsSupported(path_arg)) {
        *needs_native_bridge = true;
        handle = NativeBridgeLoadLibrary(path_arg, RTLD_NOW);
        if (handle != nullptr) {
          return handle;
        }
        *error_msg = strdup(NativeBridgeGetError());
      } else {
        *error_msg = strdup(dlerror());
      }
    }
    return nullptr;
  #endif
  }
```

在这里函数看到，使用多种方式尝试进行动态加载，分别是android_dlopen_ext、TryLoadNativeloaderExtraLib、OpenSystemLibrary。它们都是在Android平台上用来加载动态库的方法，但是它们各自的使用场景略有不同：

1. android_dlopen_ext：是一个供开发者使用的公开函数，它支持指定库的绝对路径和不同的标志（如RTLD_NOW、RTLD_LAZY等），并返回一个指向已加载库的指针，供后续调用函数的时候使用。
2. TryLoadNativeloaderExtraLib：是Android系统中的内部方法，用于加载额外的本地库。它被用于支持动态加载共享库的应用程序，例如使用反射实现的动态库加载方式。系统在应用程序启动时调用它，用于加载应用程序所需的额外本地库。使用该方法可以加载特定的本地库，并支持跨架构的执行。
3. OpenSystemLibrary：也是Android系统中的内部方法，用于加载Android系统的本地库。它不需要指定库的路径，而是使用系统库路径中的路径名来加载相应的库文件。该方法主要用于加载Android操作系统核心中的一些固定的系统库，例如 libz.so、liblog.so等。

总的来说，这三个方法都是用于加载动态库的方法，不同的是它们的使用场景略有不同。选一条路线分析即可，这里继续从android_dlopen_ext深入分析，该函数的相关代码在libdl.cpp中实现。

```
void* android_dlopen_ext(const char* filename, int flag, const android_dlextinfo*
extinfo) {
  const void* caller_addr = __builtin_return_address(0);
  return __loader_android_dlopen_ext(filename, flag, extinfo, caller_addr);
}
```

继续跟踪文件dlfcn.cpp中的实现

```
void* __loader_android_dlopen_ext(const char* filename,
                           int flags,
                           const android_dlextinfo* extinfo,
                           const void* caller_addr) {
  return dlopen_ext(filename, flags, extinfo, caller_addr);
}


static void* dlopen_ext(const char* filename,
                        int flags,
                        const android_dlextinfo* extinfo,
                        const void* caller_addr) {
  ScopedPthreadMutexLocker locker(&g_dl_mutex);
  g_linker_logger.ResetState();
  void* result = do_dlopen(filename, flags, extinfo, caller_addr);
  if (result == nullptr) {
    __bionic_format_dlerror("dlopen failed", linker_get_error_buffer());
    return nullptr;
  }
  return result;
}
```

到这里do_dlopen则执行到了Linker部分的实现了，找到linker.cpp文件查看

```
void* do_dlopen(const char* name, int flags,
                const android_dlextinfo* extinfo,
                const void* caller_addr) {
  ...
  soinfo* si = find_library(ns, translated_name, flags, extinfo, caller);
  loading_trace.End();

  if (si != nullptr) {
    void* handle = si->to_handle();
    LD_LOG(kLogDlopen,
           "... dlopen calling constructors: realpath=\"%s\", soname=\"%s\",
handle=%p",
           si->get_realpath(), si->get_soname(), handle);
    si->call_constructors();
    failure_guard.Disable();
    LD_LOG(kLogDlopen,
```

```
            "... dlopen successful: realpath=\"%s\", soname=\"%s\", handle=%p",
            si->get_realpath(), si->get_soname(), handle);
      return handle;
    }
    return nullptr;
  }
```

这里看到通过find_library进行查找的，找到后又调用了call_constructors函数。先看看call_constructors函数的处理。

```
void soinfo::call_constructors() {
  if (constructors_called || g_is_ldd) {
    return;
  }
  ...
  call_function("DT_INIT", init_func_, get_realpath());
  call_array("DT_INIT_ARRAY", init_array_, init_array_count_, false,
get_realpath());
  ...
}
```

根据上面代码发现这里就是.init和.initarray执行的地方，继续看加载的流程。

```
static soinfo* find_library(android_namespace_t* ns,
                            const char* name, int rtld_flags,
                            const android_dlextinfo* extinfo,
                            soinfo* needed_by) {
  soinfo* si = nullptr;

  if (name == nullptr) {
    si = solist_get_somain();
  } else if (!find_libraries(ns,
                             needed_by,
                             &name,
                             1,
                             &si,
                             nullptr,
                             0,
                             rtld_flags,
                             extinfo,
                             false /* add_as_children */)) {
    if (si != nullptr) {
      soinfo_unload(si);
    }
    return nullptr;
  }
```

```
  si->increment_ref_count();

  return si;
}

// 继续向下跟踪
bool find_libraries(...) {
  ...
  ZipArchiveCache zip_archive_cache;
  soinfo_list_t new_global_group_members;

  for (size_t i = 0; i<load_tasks.size(); ++i) {
    。。。
    if (!find_library_internal(const_cast<android_namespace_t*>(task-
>get_start_from()),
                               task,
                               &zip_archive_cache,
                               &load_tasks,
                               rtld_flags)) {
      return false;
    }

    soinfo* si = task->get_soinfo();
  }
  ...
  return true;
}

//追踪find_library_internal
static bool find_library_internal(android_namespace_t* ns,
                                  LoadTask* task,
                                  ZipArchiveCache* zip_archive_cache,
                                  LoadTaskList* load_tasks,
                                  int rtld_flags) {
  soinfo* candidate;
  ...
  if (load_library(ns, task, zip_archive_cache, load_tasks, rtld_flags,
                   true /* search_linked_namespaces */)) {
    return true;
  }
  ...
  return false;
}


static bool load_library(android_namespace_t* ns,
                         LoadTask* task,
                         ZipArchiveCache* zip_archive_cache,
                         LoadTaskList* load_tasks,
                         int rtld_flags,
                         bool search_linked_namespaces) {
  const char* name = task->get_name();
  soinfo* needed_by = task->get_needed_by();
  ...
```

```
  LD_LOG(kLogDlopen,
         "load_library(ns=%s, task=%s, flags=0x%x, search_linked_namespaces=%d):
calling "
         "open_library",
         ns->get_name(), name, rtld_flags, search_linked_namespaces);

  // Open the file.
  off64_t file_offset;
  std::string realpath;
  int fd = open_library(ns, zip_archive_cache, name, needed_by, &file_offset,
&realpath);
  ...
  return load_library(ns, task, load_tasks, rtld_flags, realpath,
search_linked_namespaces);
}

// open_library打开动态库文件将指定的共享库文件加载到当前进程的地址空间中，创建一个新的
动态链接对象，并返回其的句柄。
static int open_library(android_namespace_t* ns,
                        ZipArchiveCache* zip_archive_cache,
                        const char* name, soinfo *needed_by,
                        off64_t* file_offset, std::string* realpath) {
  TRACE("[ opening %s from namespace %s ]", name, ns->get_name());

  // If the name contains a slash, we should attempt to open it directly and not
search the paths.
  if (strchr(name, '/') != nullptr) {
    return open_library_at_path(zip_archive_cache, name, file_offset, realpath);
  }
  ...
  return fd;
}

// load_library加载解析ELF格式并将其链接到进程的地址空间中，将动态链接对象中的符号解析为
当前进程中的符号，从而创建动态链接的关系。
static bool load_library(android_namespace_t* ns,
                         LoadTask* task,
                         LoadTaskList* load_tasks,
                         int rtld_flags,
                         const std::string& realpath,
                         bool search_linked_namespaces) {
  ...
  soinfo* si = soinfo_alloc(ns, realpath.c_str(), &file_stat, file_offset,
rtld_flags);

  task->set_soinfo(si);

  // 读取elf header
  if (!task->read(realpath.c_str(), file_stat.st_size)) {
    task->remove_cached_elf_reader();
    task->set_soinfo(nullptr);
    soinfo_free(si);
    return false;
  }
```

```
    ...
    return true;
}

// 最后看看read函数，这个函数负责从elf文件格式的数据中读取内容
bool read(const char* realpath, off64_t file_size) {
    ElfReader& elf_reader = get_elf_reader();
    return elf_reader.Read(realpath, fd_, file_offset_, file_size);
  }
```

ElfReader是Android源文件中的工具，位于系统核心库libcore中，代码主要由C++编写。它可以读取ELF文件的所有信息，并将其解析为指定格式。

ElfReader具备以下特点：

- 读取ELF文件的头信息，包括ELF版本、目标体系结构、程序入口地址、节表偏移量等。
- 读取ELF文件的节表信息，包括节表名称、大小、偏移量、属性等。
- 通过节表信息可以获取符号表、重定位表、动态链接表等关键信息，如函数、变量、链接库、导出函数等。
- 支持通过指定节表名称获取某个节表的信息，如根据".rodata"获取只读数据节表的信息等。

小结

系统定制无论做怎样的修改，都要明白其原理，本章内容作为系统开发的内功心法，需要读者花费一些时间来吸收，在后面的内容的展开过程中，也可以随时重温本章内容，加深印象。

从设备开机到系统启动完成，整个启动链上涉及到的核心组件都在本节中进行了介绍。本节中介绍的系统组件，都是定制系统可能需要修改的地方。其中，Service与Framework的修改是最常见的，美化与安全定制都离不开它，读者朋友们可以重点阅读它们的代码来深入研究。