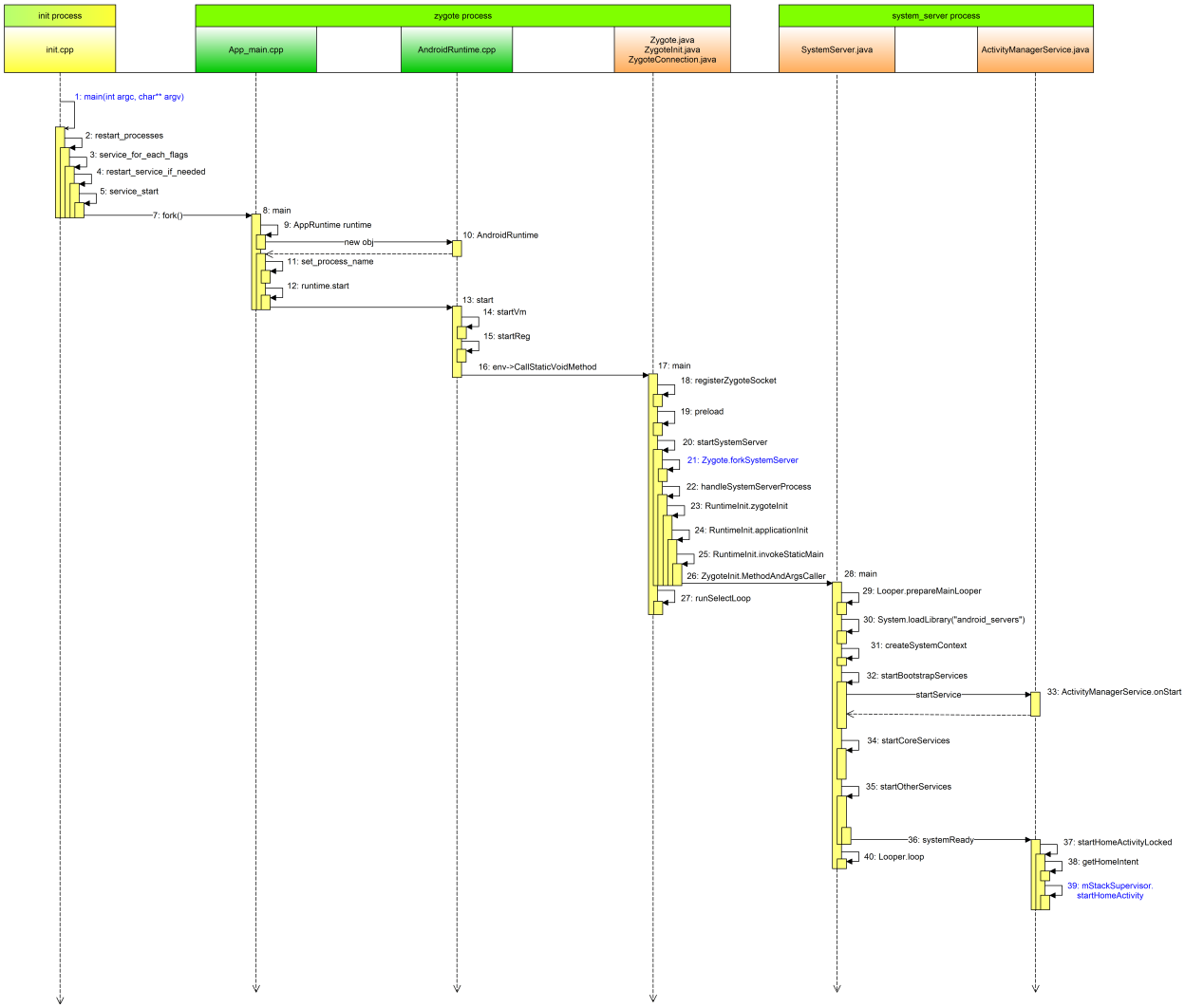
Android M 启动源码分析笔记之 - Zygote 孵化

zygote 到 Home 程序启动源码分析：

先上总时序图：



下面开始每一个阶段详细源码分析。

时序图【1-5】

```
/* init进程的main函数服务启动代码段 */
int main(int argc, char** argv) {
...
    while (true) {
        if (!waiting_for_exec) {
            execute_one_command();
            restart_processes();
        }
...
    }
    return 0;
}
```

展开：

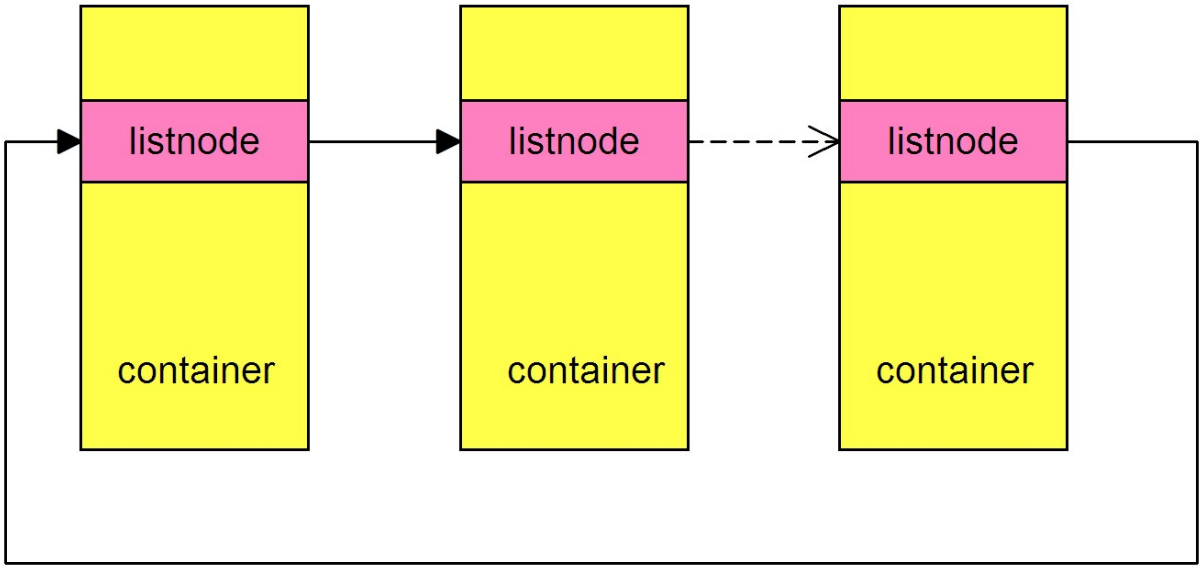
```
/* 传入服务的SVC_RESTARTING 的标记，匹配此标记的才执行启动 */
static void restart_processes()
{
    process_needs_restart = 0;
    service_for_each_flags(SVC_RESTARTING,
                           restart_service_if_needed);
}
```

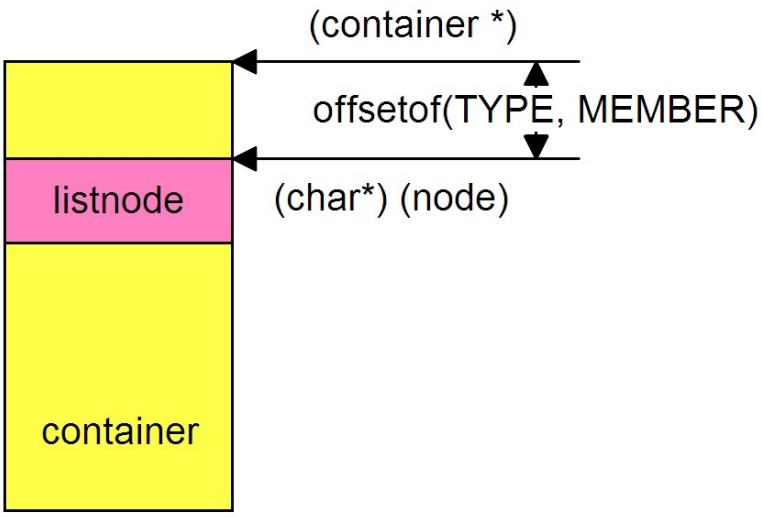
```c
/* 从服务容器链表service_list中取出每一个svc，
   匹配到标记的执行func动作，service_list 是把每个svc结构内存
   链接起来的listnode链表头指针，只要找到该svc数据结构的slist，
   就可以通过node_to_item指针操作找到对应的svc数据结构首地址.
*/
void service_for_each_flags(unsigned matchflags,
                            void (*func)(struct service *svc))
{
    struct listnode *node;
    struct service *svc;
    list_for_each(node, &service_list) {
        svc = node_to_item(node, struct service, slist);
        if (svc->flags & matchflags) {
            func(svc);
        }
    }
}

/* 宏展开 */
#define node_to_item(node, container, member) \
    (container *) (((char*) (node)) - offsetof(container, member))

#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

这里有一个重要的数据结构需要特别注意下， 这就是内核中常见的container_of提取容器数据结构大法.
node_to_item是一个宏，通过指针操作很方便实现提取容器数据结构实例，很巧妙， 简单而又深刻的展现了指针的强大和一些特定场合的不可替代性.这里有必要深入理解分析下改方法的实现，因为这个东西使用 实在是太普遍了，特别是内核源代码中到处都是.初步看上去，这个宏 数据结构相当复杂，难以理解，这里可以分解几步来看：
1、&((TYPE *)0)->MEMBER --> 将0转换为TYPE类型的指针，那么->MEMBER则是相对0这个指针的偏移地址;
2、offsetof(TYPE, MEMBER) --> 所以就是返回成员MEMBER在TYPE结构中的内存偏移量.
3、((char*) (node)) - offsetof(Container, member) --> 则是由member的具体地址（node指针）减去偏移量内存地址反推到container的首地址.再强制转换成container类型，实现提取容器数据结构的操作！
内核中的定义是类似的：

```c
#define container_of(ptr, type, member) ({   \
 const typeof( ((type *)0)->member ) *__mptr = (ptr); \
 (type *)( (char *)__mptr - offsetof(type,member) );})
```

图解：





(container *) == (char*) (node) - offsetof(TYPE, MEMBER)

有了这个理解就好了，言归正传，继续分析service_for_each_flags函数，如果匹配到了允许启动的标记，那么久会继续执行：

```
static void restart_service_if_needed(struct service *svc)
{
    time_t next_start_time = svc->time_started + 5;

    if (next_start_time <= gettime()) {
        svc->flags &= (~SVC_RESTARTING);

// 进入svc启动入口.
        service_start(svc, NULL);
        return;
    }


/* svc 启动函数*/
void service_start(struct service *svc, const char *dynamic_args)
{
// flags 复位.
```

```
    svc->flags &= (~(SVC_DISABLED|SVC_RESTARTING|SVC_RESET|SVC_RESTART|SVC_DISABLED_START));

// 初始化启动时间,开始计时.
    svc->time_started = 0;

// 接下来一堆条件、权限检查，略过..
...

/* 看到了熟悉的fork函数，直接fork一个新的进程.fork函数的执行一次
   返回两次，pid = 0 执行child proc路径，pid > 0 执行parent proc代码路径.
*/
    pid_t pid = fork();
    if (pid == 0) {
// 执行子进程代码:
        struct socketinfo *si;
        struct svcenvinfo *ei;
        char tmp[32];
        int fd, sz;

...
// 创建socket 监听，用于父子进程通信.
        for (si = svc->sockets; si; si = si->next) {
            int socket_type = (
                    !strcmp(si->type, "stream") ? SOCK_STREAM :
                        (!strcmp(si->type, "dgram") ? SOCK_DGRAM : SOCK_SEQPACKET));
            int s = create_socket(si->name, socket_type,
                                  si->perm, si->uid, si->gid, si->socketcon ?: scon);
            if (s >= 0) {
                publish_socket(si->name, s);
            }
        }
..
        if (!dynamic_args) {
/* 真正开始执行子进程系统调用入口代码，svc->args[0]为进程名称，
   svc->args为传入参数.
*/
            if (execve(svc->args[0], (char**) svc->args, (char**) ENV) < 0) {
                ERROR("cannot execve('%s'): %s\n", svc->args[0], strerror(errno));
            }
        } else {
...
        }
        _exit(127);
    }

    freecon(scon);

    if (pid < 0) {
        ERROR("failed to start '%s'\n", svc->name);
        svc->pid = 0;
        return;
    }

    svc->time_started = gettime();

// 父进程执行路径，记录pid,标记为运行状态.
    svc->pid = pid;
    svc->flags |= SVC_RUNNING;

    if ((svc->flags & SVC_EXEC) != 0) {
        INFO("SVC_EXEC pid %d (uid %d gid %d+%zu context %s) started; waiting...\n",
             svc->pid, svc->uid, svc->gid, svc->nr_supp_gids,
             svc->seclabel ? : "default");
        waiting_for_exec = true;
    }

// 更新设置该服务为“running”状态.
    svc->NotifyStateChange("running");
}
```

小结：

1、从service_list链表中依次取出符合restarting启动条件的svc；

2、svc启动权限等一系列要求检查；

3、fork()子进程，创建socket监听，execve子进程程序；

4、记录子进程pid，发通知标记为“runing”状态；

至此，一个新的服务启动完成,也可以抓开机trace看具体有哪些进程被依次fork出来：

```
Line 4974: [    3.123846] <0>.(2)[155:init]init: >>start execve(/sbin/ueventd): /sbin/ueventd
 Line 5460: [    6.995649] <1>.(0)[200:init]init: >>start execve(/system/bin/debuggerd): /system/bin/debuggerd
 Line 5464: [    7.002603] <0>.(0)[201:init]init: >>start execve(/system/bin/vold): /system/bin/vold
 Line 5516: [    8.123684] <0>.(0)[209:init]init: >>start execve(/system/bin/logd): /system/bin/logd
 Line 5584: [    8.246659] <1>.(1)[221:init]init: >>start execve(/system/bin/servicemanager): /system/bin/servicemanager
 Line 5590: [    8.276387] <0>.(1)[222:init]init: >>start execve(/system/bin/surfaceflinger): /system/bin/surfaceflinger
 Line 5676: [    8.328844] <3>.(1)[228:init]init: >>start execve(/vendor/bin/nvram_daemon): /vendor/bin/nvram_daemon
 Line 5735: [    8.357851] <3>.(1)[246:init]init: >>start execve(/vendor/bin/batterywarning): /vendor/bin/batterywarning
 Line 5769: [    8.376744] <3>.(1)[252:init]init: >>start execve(/system/bin/cameraserver): /system/bin/cameraserver
 Line 5779: [    8.384829] <3>.(0)[255:init]init: >>start execve(/system/bin/keystore): /system/bin/keystore
 Line 5803: [    8.400396] <3>.(1)[258:init]init: >>start execve(/system/bin/mediaserver): /system/bin/mediaserver
 Line 5805: [    8.401733] <3>.(2)[251:init]init: >>start execve(/system/bin/audioserver): /system/bin/audioserver
 Line 5815: [    8.417515] <3>.(2)[250:init]init: >>start execve(/system/bin/app_process): /system/bin/app_process
 Line 5817: [    8.423806] <2>.(2)[224:init]init: >>start execve(/system/bin/sh): /system/bin/sh
 Line 5823: [    8.436931] <3>.(3)[256:init]init: >>start execve(/system/bin/mediadrmserver): /system/bin/mediadrmserver
 Line 6329: [   10.084763] <1>.(3)[384:init]init: >>start execve(/sbin/adbd): /sbin/adbd
 Line 7039: [   11.081987] <3>.(0)[442:init]init: >>start execve(/system/bin/bootanimation): /system/bin/bootanimation
```

上面看到的第一个被init启动的进程是ueventd, 而 app_process 就是后面的zygote 进程, app_process 进程入口代码在：

时序图【8-16】

frameworks/base/cmsd/app_process/App_main.cpp

```
int main(int argc, char* const argv[])
{
...
/* AppRuntime 继承于AndroidRuntime,这里本质是new了一个
   AndroidRuntime的实例.
*/
    AppRuntime runtime(argv[0], computeArgBlockSize(argc, argv));
...
    if (!niceName.isEmpty()) {

// 这里的niceName为ZYGOTE_NICE_NAME，所以进程名被改为zygote
        runtime.setArgv0(niceName.string());
        set_process_name(niceName.string());
    }

// 通过AndroidRuntime 启动 java world 函数 ZygoteInit 入口.
    if (zygote) {
        runtime.start("com.android.internal.os.ZygoteInit", args, zygote);
    }
...
}
```

小结：

1、new一个AndroidRuntime的实例；

2、更改进程名为zygote；

3、进入 AndroidRuntime.start,创建VM，切换到Java world;

继续分析 AndroidRuntime：

```
void AndroidRuntime::start(const char* className, const Vector<String8>& options, bool zygote)
{
  ...

// 初始化JNI，创建、启动虚拟机.
    JniInvocation jni_invocation;
    jni_invocation.Init(NULL);
    JNIEnv* env;
    if (startVm(&mJavaVM, &env, zygote) != 0) {
        return;
    }
    onVmCreated(env);

// 这里注册JNI，所以java world才可以无障碍的使用JNI功能.
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }
```

```
 ...

// 转换类型.
    char* slashClassName = toSlashClassName(className);
    jclass startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
        /* keep going */
    } else {

// 获取startClass中的 main 方法.
        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 {

// JNI 调用 com.android.internal.os.ZygoteInit 类的main函数入口,传入参数.
            env->CallStaticVoidMethod(startClass, startMeth, strArray);
        }
    }
...
}
```

小结：

1、初始化jni引擎，创建启动VM虚拟机；

2、注册jni，这样java world才可以无障碍使用jni支持；

3、jni回调ZygoteInit类的main函数，切换到 java workd;

然后继续分析 ZygoteInit：时序图【17-26】

```
public static void main(String argv[]) {
        try {
    ...
    // 注册socket监听，socket名字为“zygote”，用于接受子进程创建req.
            registerZygoteSocket(socketName);
           ...
    // 预加载资源、opengl等各种必要类库.
            preload();
     ...
    // 进入启动system_server 进程
            if (startSystemServer) {
                startSystemServer(abiList, socketName);
            }

    // 进入while(1)循环等待创建子进程的socket连接请求.
            runSelectLoop(abiList);
            closeServerSocket();
        } catch (MethodAndArgsCaller caller) {

    /* 这个地方设计很巧妙，通过try catch 实现了goto语句的效果,每一个进程fork完成
       后都是通过这个函数反射启动新进程入口. */
            caller.run();
        } catch (RuntimeException ex) {
            closeServerSocket();
            throw ex;
        }
    }
```

zygoteInit.main干的事情小结：

1、 注册名为zygote的socket监听；

2、预加载各种必要类库资源；

3、启动system_server进程；

4、进入while(1)循环监听来自创建子进程的请求， 巧妙运用异常铺货实现goto语句效果；

下面重点分析 startSystemServer 流程：

```
private static boolean startSystemServer(String abiList, String socketName)
            throws MethodAndArgsCaller, RuntimeException {
       // 设置启动system_server进程的各种参数.
```

```
        String args[] = {
            "--setuid=1000",
            "--setgid=1000",
            "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,1021,1032,3001,3002,3003,3006,3007",
            "--capabilities=" + capabilities + "," + capabilities,
            "--nice-name=system_server",
            "--runtime-args",
            "com.android.server.SystemServer",
        };
...

        try {
...
           /* fork 出system_server 子进程 ，这里封装了fork()的实现，本质都是系统调用内核的fork实现进程复制 */
            pid = Zygote.forkSystemServer(
                    parsedArgs.uid, parsedArgs.gid,
                    parsedArgs.gids,
                    parsedArgs.debugFlags,
                    null,
                    parsedArgs.permittedCapabilities,
                    parsedArgs.effectiveCapabilities);
        } catch (IllegalArgumentException ex) {
            throw new RuntimeException(ex);
        }

        /* For child process */
        if (pid == 0) {
...
        /* 处理system server子进程 */
            handleSystemServerProcess(parsedArgs);
        }

        return true;
    }

    private static void handleSystemServerProcess(
            ZygoteConnection.Arguments parsedArgs)
            throws ZygoteInit.MethodAndArgsCaller {
...
            /* 进入RuntimeInit */
            RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.remainingArgs, cl);
        }
    }

    public static final void zygoteInit(int targetSdkVersion, String[] argv, ClassLoader classLoader)
            throws ZygoteInit.MethodAndArgsCaller {
...
    /* 公共部分初始化：handler、timezone、user agent等 */
        commonInit();

    /* 调用native函数启动binder线程池用于支持binder通信 */
        nativeZygoteInit();
        applicationInit(targetSdkVersion, argv, classLoader);
    }


  private static void applicationInit(int targetSdkVersion, String[] argv, ClassLoader classLoader)
            throws ZygoteInit.MethodAndArgsCaller {
  ...
        invokeStaticMain(args.startClass, args.startArgs, classLoader);
    }
```

那么关键部分就来了：

```
private static void invokeStaticMain(String className, String[] argv, ClassLoader classLoader)
            throws ZygoteInit.MethodAndArgsCaller {
        Class<?> cl;

/* 通过className获取需要反射启动的class<>参数： */
        try {
            cl = Class.forName(className, true, classLoader);
        } catch (ClassNotFoundException ex) {
            throw new RuntimeException(
                    "Missing class when invoking static main " + className,
                    ex);
```

```
        }

/* 获取该类的main 方法作为入口：*/
        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);
        }
...
        /* 抛出异常，然后被zygoteInit.main里的try catch捕获到，执行run()函数。*/
        throw new ZygoteInit.MethodAndArgsCaller(m, argv);
    }
```

下面就来看看这个类的原型：

```
public static class MethodAndArgsCaller extends Exception
            implements Runnable {

        private final Method mMethod;
        private final String[] mArgs;

        public MethodAndArgsCaller(Method method, String[] args) {
        /* 传入反射调用参数：main, com.android.server.SystemServer */
            mMethod = method;
            mArgs = args;
        }

        public void run() {
            try {
            /* 执行反射调用：com.android.server.SystemServer.main */
                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);
            }
        }
    }
```

到这一步执行路径就直接切到SystemServer类的main方法了：
时序图【28-36】

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

main函数很简单，就是new一个对象，然后重点来看 run() 的源码分析：

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

    /* system_server 进程就正式开始执行了，下面这句log比较关键，用于分析
       定位问题很有标志意义 ，分析system_server卡住问题标志性log.
    */
        Slog.i(TAG, "Entered the Android system server!");

    /* 设置当前线程的优先级等。*/
        android.os.Process.setThreadPriority(
                android.os.Process.THREAD_PRIORITY_FOREGROUND);
        android.os.Process.setCanSelfBackground(false);

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

```
    /* 初始化加载 native jni库. */
        System.loadLibrary("android_servers");

    /* 检查上一次关机是否成功，如果失败就进入关键流程 */
        performPendingShutdown();

        // 初始化创建系统上下文.
        createSystemContext();

        // 创建系统服务管理服务.
        mSystemServiceManager = new SystemServiceManager(mSystemContext);
        LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);

    // 依次进入启动各种系统框架服务:
        try {

    /* 启动跟boot依赖较大的服务，比如：ActivityManagerService、
       PowerManagerService、PackageManagerService、UserManagerService 、
       SensorManagerService ext. */
            startBootstrapServices();

    /* 启动核心服务：BatteryService、UsageStatsService、WebViewUpdateService */
            startCoreServices();

    /* 其他服务:CameraService、VibratorService、InputManagerService、
       NetworkManagementService ..启动 watchdog线程 等 */
            startOtherServices();
        } catch (Throwable ex) {
        ...
        }
...
    // 进入线程消息循环，永不退出.
        Looper.loop();
        throw new RuntimeException("Main thread loop unexpectedly exited");
    }
```

那么在什么时候启动Home 应用程序呢？继续看：
时序图【35-39】

```
private void startOtherServices() {
    ...
    // 从这里进入开始启动Home程序:
        mActivityManagerService.systemReady(new Runnable() {
            @Override
            public void run() {
                ...
            }
        }
    }

    // 走
    public void systemReady(final Runnable goingCallback) {
        ...
            if (!PowerOffAlarmUtility.isAlarmBoot()) {
                startHomeActivityLocked(mCurrentUserId, "systemReady");
            }
    }

    // 继续走
    boolean startHomeActivityLocked(int userId, String reason) {
        ...
        // 获取home的intent
        Intent intent = getHomeIntent();
        ActivityInfo aInfo =
            resolveActivityInfo(intent, STOCK_PM_FLAGS, userId);
        if (aInfo != null) {
           ...
                // start
                mStackSupervisor.startHomeActivity(intent, aInfo, reason);
            }
        }

        return true;
```

```
    }

    void startHomeActivity(Intent intent, ActivityInfo aInfo, String reason) {
    // 将Home的activity挪到栈顶.
        moveHomeStackTaskToTop(HOME_ACTIVITY_TYPE, reason);

    // 启动Home程序的主Activity:
        startActivityLocked(null /* caller */, intent, null /* resolvedType */, aInfo,
                null /* voiceSession */, null /* voiceInteractor */, null /* resultTo */,
                null /* resultWho */, 0 /* requestCode */, 0 /* callingPid */, 0 /* callingUid */,
                null /* callingPackage */, 0 /* realCallingPid */, 0 /* realCallingUid */,
                0 /* startFlags */, null /* options */, false /* ignoreTargetSecurity */,
                false /* componentSpecified */,
                null /* outActivity */, null /* container */,  null /* inTask */);
       ...
        }
    }
```

SystemServer 小结：

1、创建主线程,初始化native库，创建系统上下文Context；

2、启动跟boot依赖大的服务、核心服务、其它服务，包括watchdog线程；

3、启动Home程序主Activity入口，这里会涉及到app进程的创建等一系列过程；

4、进入线程消息循环；

从zygote到home启动流程就分析完了，那么回到最开始的问题：zygote如何实现孵化作用，成为java world所有进程的父进程呢？本文可以看到system_server进程是由zygote创建，那么其它app程序呢？所以，这个问题还得等分析完app的启动流程就可以完整的回答这个问题。