深入理解SystemServer

SystemServer的进程名实际上叫做“system\_server”，这里我们可将其简称为SS。SS做为Zygote的嫡长子，其重要性不言而喻。关于这一点，通过代码分析便可马上知晓。

4.3.1 SystemServer的诞生

我们先回顾一下SystemServer是怎么创建的。

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java中main()方法

**String args[] = {**

**"--setuid=1000",**

**"--setgid=1000",**

**"--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,1023,1032,3001,3002,3003,3006,3007",**

**"--capabilities=" + capabilities + "," + capabilities,**

**"--runtime-init",**

**"--nice-name=system\_server", //进程名，叫system\_server**

**"com.android.server.SystemServer", //启动的类名**

**};**

**ZygoteConnection.Arguments parsedArgs = null;**

**int pid;**

**parsedArgs = new ZygoteConnection.Arguments(args);**

**ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);**

**ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);**

**/\* Request to fork the system server process \*/**

**// fork出系统服务对应的进程**

**pid = Zygote.forkSystemServer(**

**parsedArgs.uid, parsedArgs.gid,**

**parsedArgs.gids,**

**parsedArgs.debugFlags,**

**null,**

**parsedArgs.permittedCapabilities,**

**parsedArgs.effectiveCapabilities);**

从上面的代码中可以看出，SystemServer是由Zygote通过Zygote.forkSystemServer函数fork诞生出来的。这里会有什么玄机吗？先来一起看看forkSystemServer的实现。它是一个native函数，实现在frameworks/base/core/jni/com\_android\_internal\_os\_Zygote.cpp中，如下所示：

**static jint com\_android\_internal\_os\_Zygote\_nativeForkSystemServer(**

**JNIEnv\* env, jclass, uid\_t uid, gid\_t gid, jintArray gids,**

**jint debug\_flags, jobjectArray rlimits, jlong permittedCapabilities,**

**jlong effectiveCapabilities) {**

**//根据参数，fork一个子进程**

**pid\_t pid = ForkAndSpecializeCommon(env, uid, gid, gids,**

**debug\_flags, rlimits,**

**permittedCapabilities, effectiveCapabilities,**

**MOUNT\_EXTERNAL\_NONE, NULL, NULL, true, NULL,**

**NULL, NULL);**

**if (pid > 0) {**

**// The zygote process checks whether the child process has died or not.**

**ALOGI("System server process %d has been created", pid);**

**gSystemServerPid = pid;//保存system\_server的进程id**

**// There is a slight window that the system server process has crashed**

**// but it went unnoticed because we haven't published its pid yet. So**

**// we recheck here just to make sure that all is well.**

**int status;**

**//函数退出前须先检查刚创建的子进程是否退出了。**

**if (waitpid(pid, &status, WNOHANG) == pid) {**

**//如果system\_server退出了，Zygote直接干掉了自己**

**//看来Zygote和SS的关系异常紧密，简直是生死与共！**

**ALOGE("System server process %d has died. Restarting Zygote!", pid);**

**RuntimeAbort(env);**

**}**

**}**

**return pid;**

**}**

下面，再看看forkAndSpecializeCommon，代码如下所示：

**static pid\_t ForkAndSpecializeCommon(JNIEnv\* env, uid\_t uid, gid\_t gid, jintArray javaGids,**

**jint debug\_flags, jobjectArray javaRlimits,**

**jlong permittedCapabilities, jlong effectiveCapabilities,**

**jint mount\_external,**

**jstring java\_se\_info, jstring java\_se\_name,**

**bool is\_system\_server, jintArray fdsToClose,**

**jstring instructionSet, jstring dataDir) {**

**uint64\_t start = MsTime();**

**//设置信号处理，待会儿要看看这个函数**

**SetSigChldHandler();**

**ckTime(start, "ForkAndSpecializeCommon:SetSigChldHandler");**

**pid\_t pid = fork(); //fork子进程**

**if (pid == 0) {**

**// The child process.**

**gMallocLeakZygoteChild = 1;**

**//对子进程要根据传入的参数做一些处理，例如设置进程名，设置各种id（用户id，组id等）**

**// Clean up any descriptors which must be closed immediately**

**DetachDescriptors(env, fdsToClose);**

**......**

**}**

**......**

**}**

最后看看setSignalHandler函数，它由Zygote在fork子进程前调用，代码如下所示：

**static void SetSigChldHandler() {**

**struct sigaction sa;**

**memset(&sa, 0, sizeof(sa));**

**sa.sa\_handler = SigChldHandler;**

**int err = sigaction(SIGCHLD, &sa, NULL);//设置信号处理函数，该信号是子进程死亡的信号**

**if (err < 0) {**

**ALOGW("Error setting SIGCHLD handler: %d", errno);**

**}**

**}**

//我们直接看这个信号处理函数**sigchldHandler**

**static void SigChldHandler(int /\*signal\_number\*/) {**

**pid\_t pid;**

**int status;**

**while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {**

**// Log process-death status that we care about. In general it is**

**// not safe to call LOG(...) from a signal handler because of**

**// possible reentrancy. However, we know a priori that the**

**// current implementation of LOG() is safe to call from a SIGCHLD**

**// handler in the zygote process. If the LOG() implementation**

**// changes its locking strategy or its use of syscalls within the**

**// lazy-init critical section, its use here may become unsafe.**

**if (WIFEXITED(status)) {**

**if (WEXITSTATUS(status)) {**

**ALOGI("Process %d exited cleanly (%d)", pid, WEXITSTATUS(status));**

**}**

**} else if (WIFSIGNALED(status)) {**

**if (WTERMSIG(status) != SIGKILL) {**

**ALOGI("Process %d exited due to signal (%d)", pid, WTERMSIG(status));**

**}**

**if (WCOREDUMP(status)) {**

**ALOGI("Process %d dumped core.", pid);**

**}**

**}**

**// If the just-crashed process is the system\_server, bring down zygote**

**// so that it is restarted by init and system server will be restarted**

**// from there.**

**if (pid == gSystemServerPid) {**

**ALOGE("Exit zygote because system server (%d) has terminated");**

**//如果死去的子进程是SS，则Zygote把自己也干掉了，这样就做到了生死与共！**

**kill(getpid(), SIGKILL);**

**}**

**}**

**// Note that we shouldn't consider ECHILD an error because**

**// the secondary zygote might have no children left to wait for.**

**if (pid < 0 && errno != ECHILD) {**

**ALOGW("Zygote SIGCHLD error in waitpid: %s", strerror(errno));**

**}**

**}**

OK，做为Zygote的嫡长子，SystemServer确实具有非常高的地位，竟然到了与Zygote生死与共的地步!它为什么这么重要呢？我们现在就从forkSystemServer来分析SystemServer究竟承担了怎样的工作使命。

4.3.2 SystemServer的重要使命

SystemServer诞生后，便和生父Zygote分道扬镳，它有了自己的历史使命。它的使命是什么呢？其代码如下所示：

**ZygoteInit.java->main()**

**pid =Zygote.forkSystemServer();**

**if(pid == 0) { //SystemServer进程返回0，那么下面这句话就是SystemServer的使命：**

**handleSystemServerProcess(parsedArgs);**

**}**

SystemServer调用handleSystemServerProcess来承担自己的职责。

**private static void handleSystemServerProcess(ZygoteConnection.Arguments parsedArgs)**

**throws ZygoteInit.MethodAndArgsCaller {**

**//关闭从Zygote那里继承下来的Socket。**

**closeServerSocket();**

**// set umask to 0077 so new files and directories will default to owner-only permissions.**

**Os.umask(S\_IRWXG | S\_IRWXO);**

**if (parsedArgs.niceName != null) {**

**Process.setArgV0(parsedArgs.niceName); //设置niceName**

**}**

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

**if (systemServerClasspath != null) {**

**performSystemServerDexOpt(systemServerClasspath);**

**}**

**if (parsedArgs.invokeWith != null) {**

**String[] args = parsedArgs.remainingArgs;**

**// If we have a non-null system server class path, we'll have to duplicate the**

**// existing arguments and append the classpath to it. ART will handle the classpath**

**// correctly when we exec a new process.**

**if (systemServerClasspath != null) {**

**String[] amendedArgs = new String[args.length + 2];**

**amendedArgs[0] = "-cp";**

**amendedArgs[1] = systemServerClasspath;**

**System.arraycopy(parsedArgs.remainingArgs, 0,**

**amendedArgs, 2, parsedArgs.remainingArgs.length);**

**}**

**WrapperInit.execApplication(parsedArgs.invokeWith,**

**parsedArgs.niceName, parsedArgs.targetSdkVersion,**

**null, args);**

**} else {**

**ClassLoader cl = null;**

**if (systemServerClasspath != null) {**

**cl = new PathClassLoader(systemServerClasspath,**

**ClassLoader.getSystemClassLoader());**

**Thread.currentThread().setContextClassLoader(cl);**

**}**

**// Pass the remaining arguments to SystemServer.**

**RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion,**

**parsedArgs.remainingArgs, cl);**

**}**

**/\* should never reach here \*/**

**}**

好了，SystemServer走到RuntimeInit了，它的代码在RuntimeInit.java中，如下所示

**public static final void zygoteInit(int targetSdkVersion, String[] argv,**

**ClassLoader classLoader)**

**throws ZygoteInit.MethodAndArgsCaller {**

**if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application from zygote");**

**redirectLogStreams();**

**//做一些常规初始化**

**commonInit();**

**//①native层的初始化。**

**nativeZygoteInit();**

**applicationInit(targetSdkVersion, argv, classLoader);**

**}**

**private static void applicationInit(int targetSdkVersion, String[] argv,**

**ClassLoader classLoader)**

**throws ZygoteInit.MethodAndArgsCaller {**

**nativeSetExitWithoutCleanup(true);**

**// We want to be fairly aggressive about heap utilization, to avoid**

**// holding on to a lot of memory that isn't needed.**

**VMRuntime.getRuntime().setTargetHeapUtilization(0.75f);**

**VMRuntime.getRuntime().setTargetSdkVersion(targetSdkVersion);**

**final Arguments args;**

**try {**

**args = new Arguments(argv);**

**} catch (IllegalArgumentException ex) {**

**Slog.e(TAG, ex.getMessage());**

**// let the process exit**

**return;**

**}**

**// Remaining arguments are passed to the start class's static main**

**//②调用startClass，也就是com.android.server.SystemServer类的main函数。**

**invokeStaticMain(args.startClass, args.startArgs, classLoader);**

**}**

对于上面列举出的两个关键点，我们一个一个地分析。

1. zygoteInitNative分析

先看zygoteInitNative，它是一个native函数，实现在AndroidRuntime.cpp中。

**[-->AndroidRuntime.cpp]**

**static void com\_android\_internal\_os\_RuntimeInit\_nativeZygoteInit(JNIEnv\* env,**

**jobject clazz) {**

**gCurRuntime->onZygoteInit();**

**}**

//gCurRuntime是什么？还记得我们在本章开始说的app\_process的main函数吗？

static AndroidRuntime\* gCurRuntime = NULL;

由于SystemSever是从Zygote fork出来的，所以它也拥有Zygote进程中定义的这个gCurRuntime，也就是AppRuntime对象。那么，它的onZygoteInit会干些什么呢？它的代码在App\_main.cpp中，我们一起来看：

**[-->App\_main.cpp]**

**virtual void onZygoteInit()**

**{**

**//下面这些东西和Binder有关系，但读者可以先不管它。**

**sp<ProcessState> proc = ProcessState::self();**

**if(proc->supportsProcesses()) {**

**proc->startThreadPool();//启动一个线程，用于Binder通信。**

**}**

**}**

一言以蔽之，SystemSever调用zygoteInitNative后，将和Binder通信系统建立联系，这样SystemServer就能够使用Binder了。关于Binder的知识，在第6章中将详细介绍，读者朋友现在不必关注。

2. invokeStaticMain分析

再来看第二个关键点invokeStaticMain。代码如下所示：

**[-->RuntimeInit.java]**

**private static void invokeStaticMain(String className, String[] argv,**

**ClassLoader classLoader) throws ZygoteInit.MethodAndArgsCaller {**

**Class<?> cl;**

**//注意我们的参数，className为"com.android.server.SystemServer"**

**try {**

**cl = Class.forName(className, true, classLoader);**

**} catch (ClassNotFoundException ex) {**

**throw new RuntimeException(**

**"Missing class when invoking static main " + className,**

**ex);**

**}**

**Method m;**

**try {**

**//找到com.android.server.SystemServer类的main函数，肯定有地方要调用它**

**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.**

**\*/**

**//抛出一个异常，为什么不在这里直接调用上面的main函数呢？**

**throw new ZygoteInit.MethodAndArgsCaller(m, argv);**

**}**

invokeStaticMain竟然抛出了一个异常，它是在哪里被截获呢？原来是在ZygoteInit的main函数中。请看这段代码：

注意：我们所在的进程是system\_server。

**[-->ZygoteInit.java]**

**....**

**if (argv[1].equals("true")) {**

**//SS进程中，抛出一个异常MethodAndArgsCaller**

**startSystemServer();**

**......**

**catch(MethodAndArgsCaller caller) {**

**//被截获，调用caller的run函数**

**caller.run();**

**}**

再来看看**MethodAndArgsCaller**的run函数。

**public void run() {**

**try {**

**//这个mMethod为com.android.server.SystemServer的main函数**

**mMethod.invoke(null, new Object[] { mArgs });**

**} catch (IllegalAccessException ex) {**

**throw new RuntimeException(ex);**

**......**

**}**

抛出的这个异常最后会导致**com.android.server.SystemServer**类的**main**函数被调用。不过这里有一个疑问，为什么不在invokeStaticMain那里直接调用，而是采用这种抛异常的方式呢？我对这个问题的看法是：

·这个调用是在ZygoteInit.main中，相当于Native的main函数，即入口函数，位于堆栈的顶层。如果不采用抛异常的方式，而是在invokeStaticMain那里调用，则会浪费之前函数调用所占用的一些调用堆栈。

关于这个问题的深层思考，读者可以利用**fork**和**exec**的知识。对这种抛异常的方式，我个人觉得是对**exec**的一种近似模拟，因为后续的工作将交给**com.android.server.SystemServer**类来处理

3. SystemServer的真面目

ZygoteInit分裂产生的SystemServer，其实就是为了调用com.android.server.SystemServer的main函数，这简直就是改头换面！下面就来看看这个真实的main函数，代码如下所示：

**[-->SystemServer.java]**

**public static void main(String[] args) {**

**new SystemServer().run();**

**}**

**private void run() {**

**// If a device's clock is before 1970 (before 0), a lot of**

**// APIs crash dealing with negative numbers, notably**

**// java.io.File#setLastModified, so instead we fake it and**

**// hope that time from cell towers or NTP fixes it shortly.**

**if (System.currentTimeMillis() < EARLIEST\_SUPPORTED\_TIME) {**

**Slog.w(TAG, "System clock is before 1970; setting to 1970.");**

**SystemClock.setCurrentTimeMillis(EARLIEST\_SUPPORTED\_TIME);**

**}**

**// Here we go!**

**Slog.i(TAG, "Entered the Android system server!");**

**EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_SYSTEM\_RUN, SystemClock.uptimeMillis());**

**/// M: BOOTPROF @{**

**mMTPROF\_disable = "1".equals(SystemProperties.get("ro.mtprof.disable"));**

**addBootEvent(new String("Android:SysServerInit\_START"));**

**/// @}**

**// In case the runtime switched since last boot (such as when**

**// the old runtime was removed in an OTA), set the system**

**// property so that it is in sync. We can't do this in**

**// libnativehelper's JniInvocation::Init code where we already**

**// had to fallback to a different runtime because it is**

**// running as root and we need to be the system user to set**

**// the property. <http://b/11463182>**

**//设置当前虚拟机的运行库路径**

**SystemProperties.set("persist.sys.dalvik.vm.lib.2", VMRuntime.getRuntime().vmLibrary());**

**// Enable the sampling profiler.**

**if (SamplingProfilerIntegration.isEnabled()) {**

**SamplingProfilerIntegration.start();**

**mProfilerSnapshotTimer = new Timer();**

**mProfilerSnapshotTimer.schedule(new TimerTask() {**

**@Override**

**public void run() {**

**SamplingProfilerIntegration.writeSnapshot("system\_server", null);**

**}**

**}, SNAPSHOT\_INTERVAL, SNAPSHOT\_INTERVAL);**

**}**

**// Mmmmmm... more memory!**

**VMRuntime.getRuntime().clearGrowthLimit();**

**// The system server has to run all of the time, so it needs to be**

**// as efficient as possible with its memory usage.**

**VMRuntime.getRuntime().setTargetHeapUtilization(0.8f);**

**// Some devices rely on runtime fingerprint generation, so make sure**

**// we've defined it before booting further.**

**Build.ensureFingerprintProperty();**

**// Within the system server, it is an error to access Environment paths without**

**// explicitly specifying a user.**

**Environment.setUserRequired(true);**

**// Ensure binder calls into the system always run at foreground priority.**

**BinderInternal.disableBackgroundScheduling(true);**

**// Prepare the main looper thread (this thread).**

**android.os.Process.setThreadPriority(**

**android.os.Process.THREAD\_PRIORITY\_FOREGROUND);**

**android.os.Process.setCanSelfBackground(false);**

**Looper.prepareMainLooper();**

**// Initialize native services.**

**System.loadLibrary("android\_servers");//装载libandroid\_servers.so**

**nativeInit();//启动native层的service，**

**///M:Add for low storage feature,to delete the reserver file.@{**

**try {**

**Runtime.getRuntime().exec("rm -r /data/piggybank");**

**} catch (IOException e) {**

**Slog.e(TAG, "system server init delete piggybank fail" + e);**

**}**

**///@}**

**// Check whether we failed to shut down last time we tried.**

**// This call may not return.**

**performPendingShutdown();**

**// Initialize the system context.**

**createSystemContext();**

**// Create the system service manager.**

**mSystemServiceManager = new SystemServiceManager(mSystemContext);**

**LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);**

**// Start services.**

**try {**

**//启动所有的java service**

**startBootstrapServices();**

**startCoreServices();**

**startOtherServices();**

**} catch (Throwable ex) {**

**Slog.e("System", "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");**

**Slog.e("System", "\*\*\*\*\*\*\*\*\*\*\*\* Failure starting system services", ex);**

**/// M: RecoveryManagerService @{**

**if (mRecoveryManagerService != null && ex instanceof RuntimeException) {**

**mRecoveryManagerService.handleException((RuntimeException)ex, true);**

**}**

**/// @}**

**}**

**// For debug builds, log event loop stalls to dropbox for analysis.**

**if (StrictMode.conditionallyEnableDebugLogging()) {**

**Slog.i(TAG, "Enabled StrictMode for system server main thread.");**

**}**

**/// M: BOOTPROF**

**addBootEvent(new String("Android:SysServerInit\_END"));**

**// Loop forever.**

**Looper.loop();**

**throw new RuntimeException("Main thread loop unexpectedly exited");**

**}**

main()方法的主要作用是：

（1）设置属性persist.sys.dalvik.vm.lib.2的值为当前虚拟机的运行库路径。

（2）调整时间。

（3）调整虚拟机堆的内存。设定虚拟机堆利用率为0.8，当实际的使用率偏离设定的比率时，虚拟机在垃圾回收的时候将调整堆的大小，使实际使用率接近设定的百分比。

（4）装载libandroid\_servers.so库。这个库的源文件位于frameworks\base\services\core\jni。

（5）调用nativeInit()方法初始化native层的Binder服务。这里native层的服务只有SensorService。

对应的native层函数是android\_server\_SystemServer\_nativeInit()，如下：

**frameworks/base/services/core/jni/com\_android\_server\_SystemServer.cpp**

**static void android\_server\_SystemServer\_nativeInit(JNIEnv\* env, jobject clazz) {**

**char propBuf[PROPERTY\_VALUE\_MAX];**

**property\_get("system\_init.startsensorservice", propBuf, "1");**

**if (strcmp(propBuf, "1") == 0) {**

**// Start the sensor service**

**SensorService::instantiate();**

**}**

**}**

如果属性system\_init.startsensorservice设置为1，就启动SensorService服务。SensorService提供的是各种传感器的服务，也是SystemServer中唯一的本地服务。

（6）调用createSystemContext()来获取Context，相当于创建了一个framework-res.apk的上下文环境。

（7）创建SystemServiceManager对象mSystemServiceManager。这个对象负责系统Service的启动。

（8）startBootstrapServices()、startCoreServices()、startOtherServices()创建并运行所有java服务。

（9）调用Loop.loop()，进入处理消息的循环。

SystemServer中的Watchdog

Android开发了WatchDog类作为软件看门狗来监控SystemServer进程，一旦发现问题，WatchDog会杀死SystemServer进程。SystemServer的父进程Zygote接收到SystemServer的死亡信号后，会杀死自己。Zygote进程的死亡信号传递到Init进程后，Init进程会杀死Zygote进程所有子进程并重启Zygote。这样整个手机相当于重启一遍。

启动WatchDog

在SystemServer中创建WatchDog对象，如下：

在startOtherService()方法中：

**private void startOtherServices() {**

**......**

**final Watchdog watchdog = Watchdog.getInstance();**

**watchdog.init(context, mActivityManagerService);**

**......**

**}**

WatchDog是单实例的运行模式。

**frameworks/base/services/core/java/com/android/server/Watchdog.java**

**public static Watchdog getInstance() {**

**if (sWatchdog == null) {**

**sWatchdog = new Watchdog();**

**}**

**return sWatchdog;**

**}**

**private Watchdog() {**

**super("watchdog");**

**// Initialize handler checkers for each common thread we want to check. Note**

**// that we are not currently checking the background thread, since it can**

**// potentially hold longer running operations with no guarantees about the timeliness**

**// of operations there.**

**// The shared foreground thread is the main checker. It is where we**

**// will also dispatch monitor checks and do other work.**

**mMonitorChecker = new HandlerChecker(FgThread.getHandler(),**

**"foreground thread", DEFAULT\_TIMEOUT);**

**mHandlerCheckers.add(mMonitorChecker);**

**// Add checker for main thread. We only do a quick check since there**

**// can be UI running on the thread.**

**mHandlerCheckers.add(new HandlerChecker(new Handler(Looper.getMainLooper()),**

**"main thread", DEFAULT\_TIMEOUT));**

**// Add checker for shared UI thread.**

**mHandlerCheckers.add(new HandlerChecker(UiThread.getHandler(),**

**"ui thread", DEFAULT\_TIMEOUT));**

**// And also check IO thread.**

**mHandlerCheckers.add(new HandlerChecker(IoThread.getHandler(),**

**"i/o thread", DEFAULT\_TIMEOUT));**

**mHandlerCheckers.add(new HandlerChecker(DisplayThread.getHandler(),**

**"display thread", DEFAULT\_TIMEOUT));**

**if (SystemProperties.get("ro.have\_aee\_feature").equals("1")) {**

**exceptionHWT = new ExceptionLog();**

**}**

**}**

WatchDog的构造方法主要工作是创建几个HandlerChecker对象，并把他们保存到数组列表mHandlerCheckers中。每个HandlerChecker对象对应一个被监控的线程。HandlerChecker类从Handler类派生，在构造时就和被监控的线程关联在一起。

WatchDog对象创建后，接下来会调用init()方法进行初始化，

**public void init(Context context, ActivityManagerService activity) {**

**mResolver = context.getContentResolver();**

**mActivity = activity;**

**context.registerReceiver(new RebootRequestReceiver(),**

**new IntentFilter(Intent.ACTION\_REBOOT),**

**android.Manifest.permission.REBOOT, null);**

**}**

注册了RebootRequestReceiver，监听重启的intent：ACTION\_REBOOT。

WatchDog监控的服务和线程

WatchDog主要监控线程。如果一个线程陷入了死循环或者和其他的线程相互死锁了，WatchDog需要有办法识别出它们。

WatchDog中提供的两个方法addThread和addMonitor()方法分别用来增加需要监控的线程和服务。addThread()实际是创建一个和受监控对象关联的HandlerChecker对象。

**public void addThread(Handler thread) {**

**addThread(thread, DEFAULT\_TIMEOUT);**

**}**

**public void addThread(Handler thread, long timeoutMillis) {**

**synchronized (this) {**

**if (isAlive()) {**

**throw new RuntimeException("Threads can't be added once the Watchdog is running");**

**}**

**final String name = thread.getLooper().getThread().getName();**

**mHandlerCheckers.add(new HandlerChecker(thread, name, timeoutMillis));**

**}**

**}**

对服务的监控也是由HandlerChecker对象来完成。一个HandlerChecker对象就可以检查所有服务。因此，WatchDog让mMonitorChecker对象来完成这个任务。

public void addMonitor(Monitor monitor) {

synchronized (this) {

if (isAlive()) {

throw new RuntimeException("Monitors can't be added once the Watchdog is running");

}

mMonitorChecker.addMonitor(monitor);

}

}

WatchDog的addMonitor()方法只是调用了mMonitorChecker对象的addMonitor方法。

在WatchDog的构造函数中，把5个公共线程加入到了监控列表中。

主线程

FgThread

UiThread。

IoThread。

DisplayThread。

SystemServer中一些重要的服务拥有专用的线程来处理消息。除了这5个公共线程外，一些重要服务的专用线程也加入到了监控中，包括：

ActivityManagerService的AThread线程。

PackageManagerService的mHandlerThread变量表示的线程。

PowerManagerService的线程。

WindowManagerService的wmHandlerThread变量表示的线程。

如果一个服务需要通过WatchDog来监控，它必须首先实现WatchDog的接口Moniter：

public interface Monitor {

void monitor();

}

WatchDog监控的原理

通过给现场发送消息可以判断一个线程是否运行正常，如果发送的消息不能在规定的时间内得到处理，就表明线程被不正常占用了。

WatchDog运行在一个单独的线程中，如下：

**public void run() {**

**boolean waitedHalf = false;**

**while (true) {**

**final ArrayList<HandlerChecker> blockedCheckers;**

**final String subject;**

**final boolean allowRestart;**

**int debuggerWasConnected = 0;**

**synchronized (this) {**

**long timeout = CHECK\_INTERVAL;**

**// Make sure we (re)spin the checkers that have become idle within**

**// this wait-and-check interval给监控的线程发送消息**

**for (int i=0; i<mHandlerCheckers.size(); i++) {**

**HandlerChecker hc = mHandlerCheckers.get(i);**

**hc.scheduleCheckLocked();**

**}**

**if (debuggerWasConnected > 0) {**

**debuggerWasConnected--;**

**}**

**// NOTE: We use uptimeMillis() here because we do not want to increment the time we**

**// wait while asleep. If the device is asleep then the thing that we are waiting**

**// to timeout on is asleep as well and won't have a chance to run, causing a false**

**// positive on when to kill things.**

**long start = SystemClock.uptimeMillis();**

**while (timeout > 0) {//睡眠一段时间**

**if (Debug.isDebuggerConnected()) {**

**debuggerWasConnected = 2;**

**}**

**try {**

**wait(timeout);**

**} catch (InterruptedException e) {**

**Log.wtf(TAG, e);**

**}**

**if (Debug.isDebuggerConnected()) {**

**debuggerWasConnected = 2;**

**}**

**timeout = CHECK\_INTERVAL - (SystemClock.uptimeMillis() - start);**

**}**

**//检查是否有线程或服务出问题**

**final int waitState = evaluateCheckerCompletionLocked();**

**if (waitState == COMPLETED) {**

**// The monitors have returned; reset**

**waitedHalf = false;**

**continue;**

**} else if (waitState == WAITING) {**

**// still waiting but within their configured intervals; back off and recheck**

**continue;**

**} else if (waitState == WAITED\_HALF) {**

**if (!waitedHalf) {**

**// We've waited half the deadlock-detection interval. Pull a stack**

**// trace and wait another half.**

**ArrayList<Integer> pids = new ArrayList<Integer>();**

**pids.add(Process.myPid());**

**ActivityManagerService.dumpStackTraces(true, pids, null, null,**

**NATIVE\_STACKS\_OF\_INTEREST);**

**waitedHalf = true;**

**}**

**continue;**

**}**

**// something is overdue!**

**blockedCheckers = getBlockedCheckersLocked();**

**subject = describeCheckersLocked(blockedCheckers);**

**allowRestart = mAllowRestart;**

**}**

**// If we got here, that means that the system is most likely hung.**

**// First collect stack traces from all threads of the system process.**

**// Then kill this process so that the system will restart.**

**EventLog.writeEvent(EventLogTags.WATCHDOG, subject);**

**ArrayList<Integer> pids = new ArrayList<Integer>();**

**pids.add(Process.myPid());**

**if (mPhonePid > 0) pids.add(mPhonePid);**

**// Pass !waitedHalf so that just in case we somehow wind up here without having**

**// dumped the halfway stacks, we properly re-initialize the trace file.**

**final File stack = ActivityManagerService.dumpStackTraces(**

**!waitedHalf, pids, null, null, NATIVE\_STACKS\_OF\_INTEREST);**

**// Give some extra time to make sure the stack traces get written.**

**// The system's been hanging for a minute, another second or two won't hurt much.**

**SystemClock.sleep(2000);**

**// Pull our own kernel thread stacks as well if we're configured for that**

**if (RECORD\_KERNEL\_THREADS) {**

**dumpKernelStackTraces();**

**}**

**// Trigger the kernel to dump all blocked threads, and backtraces on all CPUs to the kernel log**

**doSysRq('w');**

**doSysRq('l');**

**String tracesPath = SystemProperties.get("dalvik.vm.stack-trace-file", null);**

**String traceFileNameAmendment = "\_SystemServer\_WDT" + mTraceDateFormat.format(new Date());**

**if (tracesPath != null && tracesPath.length() != 0) {**

**File traceRenameFile = new File(tracesPath);**

**String newTracesPath;**

**int lpos = tracesPath.lastIndexOf (".");**

**if (-1 != lpos)**

**newTracesPath = tracesPath.substring (0, lpos) + traceFileNameAmendment + tracesPath.substring (lpos);**

**else**

**newTracesPath = tracesPath + traceFileNameAmendment;**

**traceRenameFile.renameTo(new File(newTracesPath));**

**tracesPath = newTracesPath;**

**}**

**final File newFd = new File(tracesPath);**

**// Try to add the error to the dropbox, but assuming that the ActivityManager**

**// itself may be deadlocked. (which has happened, causing this statement to**

**// deadlock and the watchdog as a whole to be ineffective)**

**Thread dropboxThread = new Thread("watchdogWriteToDropbox") {**

**public void run() {**

**mActivity.addErrorToDropBox(**

**"watchdog", null, "system\_server", null, null,**

**subject, null, newFd, null);**

**}**

**};**

**dropboxThread.start();**

**try {**

**dropboxThread.join(2000); // wait up to 2 seconds for it to return.**

**} catch (InterruptedException ignored) {}**

**// At times, when user space watchdog traces don't give an indication on**

**// which component held a lock, because of which other threads are blocked,**

**// (thereby causing Watchdog), crash the device to analyze RAM dumps**

**boolean crashOnWatchdog = SystemProperties**

**.getBoolean("persist.sys.crashOnWatchdog", false);**

**if (crashOnWatchdog) {**

**// wait until the above blocked threads be dumped into kernel log**

**SystemClock.sleep(3000);**

**// now try to crash the target**

**try {**

**FileWriter sysrq\_trigger = new FileWriter("/proc/sysrq-trigger");**

**sysrq\_trigger.write("c");**

**sysrq\_trigger.close();**

**} catch (IOException e) {**

**Slog.e(TAG, "Failed to write 'c' to /proc/sysrq-trigger");**

**Slog.e(TAG, e.getMessage());**

**}**

**}**

**IActivityController controller;**

**synchronized (this) {**

**controller = mController;**

**}**

**if (controller != null) {**

**Slog.i(TAG, "Reporting stuck state to activity controller");**

**try {**

**Binder.setDumpDisabled("Service dumps disabled due to hung system process.");**

**// 1 = keep waiting, -1 = kill system**

**int res = controller.systemNotResponding(subject);**

**if (res >= 0) {**

**Slog.i(TAG, "Activity controller requested to coninue to wait");**

**waitedHalf = false;**

**continue;**

**}**

**} catch (RemoteException e) {**

**}**

**}**

**// Only kill the process if the debugger is not attached.**

**if (Debug.isDebuggerConnected()) {**

**debuggerWasConnected = 2;**

**}**

**if (debuggerWasConnected >= 2) {**

**Slog.w(TAG, "Debugger connected: Watchdog is \*not\* killing the system process");**

**} else if (debuggerWasConnected > 0) {**

**Slog.w(TAG, "Debugger was connected: Watchdog is \*not\* killing the system process");**

**} else if (!allowRestart) {**

**Slog.w(TAG, "Restart not allowed: Watchdog is \*not\* killing the system process");**

**} else {**

**Slog.w(TAG, "\*\*\* WATCHDOG KILLING SYSTEM PROCESS: " + subject);**

**for (int i=0; i<blockedCheckers.size(); i++) {**

**Slog.w(TAG, blockedCheckers.get(i).getName() + " stack trace:");**

**StackTraceElement[] stackTrace**

**= blockedCheckers.get(i).getThread().getStackTrace();**

**for (StackTraceElement element: stackTrace) {**

**Slog.w(TAG, " at " + element);**

**}**

**}**

**Slog.w(TAG, "\*\*\* GOODBYE!");**

**Process.killProcess(Process.myPid());**

**System.exit(10);**

**}**

**waitedHalf = false;**

**}**

**}**

run()方法中有一个无限循环，每次循环主要做3件事。

（1）调用scheduleCheckLocked()方法给所有受监控的线程发送消息。

public void scheduleCheckLocked() {

if (mMonitors.size() == 0 && mHandler.getLooper().isIdling()) {

// If the target looper is or just recently was idling, then

// there is no reason to enqueue our checker on it since that

// is as good as it not being deadlocked. This avoid having

// to do a context switch to check the thread. Note that we

// only do this if mCheckReboot is false and we have no

// monitors, since those would need to be executed at this point.

mCompleted = true;

return;

}

if (!mCompleted) {

// we already have a check in flight, so no need

return;

}

mCompleted = false;

mCurrentMonitor = null;

mStartTime = SystemClock.uptimeMillis();

mHandler.postAtFrontOfQueue(this);

}

HandlerCHecker对象既要监控服务，又要监控某个线程；因此先判断mMonitors的size是否为0，如果为0，说明这个HandlerChecker没有监控服务，这时如果被监控线程的消息队列处于空闲状态（isIding()方法判断），则说明线程运行良好，把mCompleted设为true后就可以返回了。否则先把mCompleted设为false，并记录消息开始发送的时间，最后调用postAtFrontOfQueue()方法给被监控的线程发送一个消息。这个消息的处理方法是HandlerChecker类的run()方法，如下：

public void run() {

final int size = mMonitors.size();

for (int i = 0 ; i < size ; i++) {

synchronized (Watchdog.this) {

mCurrentMonitor = mMonitors.get(i);

}

mCurrentMonitor.monitor();

}

synchronized (Watchdog.this) {

mCompleted = true;

mCurrentMonitor = null;

}

}

如果消息处理方法run()能够执行，说明受监控的线程本身没有问题。但是，还要检查被监控服务的状态。检查是通过调用服务中实现的monitor()方法来完成的。通常monitor()方法的实现是获取服务中的锁，如果不能得到，线程就会挂起，这样mCompleted的值就不能被设为true。

mCompleted的值为true，表面HandlerChecker对象监控的线程或服务正常。否则就有空有问题。是否真有问题还要通过等待的时间是否超过规定时间来判断。

（2）、给受监控的线程发送完消息后，调用wait()方法让WatchDog线程睡眠一段时间。

（3）、逐个检查是否有线程或服务出问题了，一旦发现，马上杀死进程。

通过evaluateCheckerCompletionLocked()方法来检查，如下：

private int evaluateCheckerCompletionLocked() {

int state = COMPLETED;

for (int i=0; i<mHandlerCheckers.size(); i++) {

HandlerChecker hc = mHandlerCheckers.get(i);

state = Math.max(state, hc.getCompletionStateLocked());

}

return state;

}

evaluateCheckerCompletionLocked()通过调用每个检查对象的getCompletionStatelocked()方法来得到对象的状态值。状态值有4种。

COMPLETED：0，表示状态良好。

WAITING：1，表示正在等待消息处理的结果。

WAITED\_HALF：2，表示正在等待并且等待的时间超过了规定时间的一半。

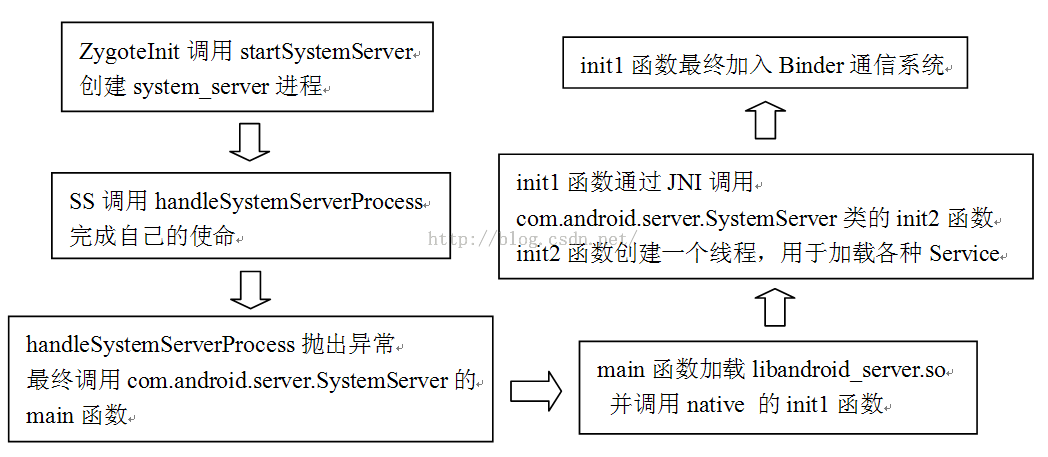
OVERDUE：3，表示等待时间已经超过了规定的时间。

evaluateCheckerCompletionLocked()方法希望知道的是最坏的情况，因此上面代码中使用Math.max()方法来计算出所有监控对象的最坏情况。

OVERDUE状态就会杀死SystemServer进程。

4.3.3 关于 SystemServer的总结

SystemServer曲折的调用流程真让人眼花缭乱，我们用图4-2来展示这一过程：



4.4 Zygote的分裂

前文已经讲道，Zygote分裂出嫡长子system\_server后，就通过runSelectLoop等待并处理来自客户的消息，那么，谁会向Zygote发送消息呢？这里，以一个Activity的启动为例，具体分析Zygote是如何分裂和繁殖的。

4.4.1 ActivityManagerService发送请求

ActivityManagerService也是由SystemServer创建的。假设通过startActivit来启动一个新的Activity，而这个Activity附属于一个还未启动的进程，那么这个进程该如何启动呢？先来看看ActivityManagerService中的startProcessLocked函数，代码如下所示：

**[-->ActivityManagerService.java]**

**protected void startProcessLocked(ProcessRecord app,**

**String hostingType, String hostingNameStr) {**

**startProcessLocked(app, hostingType, hostingNameStr, null /\* abiOverride \*/,**

**null /\* entryPoint \*/, null /\* entryPointArgs \*/);**

**}**

**protected void startProcessLocked(ProcessRecord app, String hostingType,**

**String hostingNameStr, String abiOverride,**

**String entryPoint, String[] entryPointArgs) {**

**......**

**// Start the process. It will either succeed and return a result containing**

**// the PID of the new process, or else throw a RuntimeException.**

**boolean isActivityProcess = (entryPoint == null);**

**if (entryPoint == null) entryPoint = "android.app.ActivityThread";**

**/// M: Add for launch time debug**

**Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER,"amProcessStart");**

**checkTime(startTime, "startProcess: asking zygote to start proc");**

**//这个Process类是Android提供的，并非JDK中的Process类**

**Process.ProcessStartResult startResult = Process.start(entryPoint,**

**app.processName, uid, uid, gids, debugFlags, mountExternal,**

**app.info.targetSdkVersion, app.info.seinfo, requiredAbi, instructionSet,**

**app.info.dataDir, entryPointArgs);**

**......**

**}**

接着来看看Process的start函数，这个Process类是android.os.Process，它的代码在Process.java中，代码如下所示：

**[-->Process.java]**

**public static final ProcessStartResult start(final String processClass,**

**final String niceName, int uid, int gid, int[] gids,**

**int debugFlags, int mountExternal,int targetSdkVersion,**

**String seInfo, String abi, String instructionSet,**

**String appDataDir, String[] zygoteArgs) {**

**try {**

**//调用startViaZygote。**

**return startViaZygote(processClass, niceName, uid, gid, gids,**

**debugFlags, mountExternal, targetSdkVersion, seInfo,**

**abi, instructionSet, appDataDir, zygoteArgs);**

**} catch (ZygoteStartFailedEx ex) {**

**Log.e(LOG\_TAG,**

**"Starting VM process through Zygote failed");**

**throw new RuntimeException(**

**"Starting VM process through Zygote failed", ex);**

**}**

**}**

**private static ProcessStartResult startViaZygote(final String processClass,**

**final String niceName, final int uid, final int gid,**

**final int[] gids, int debugFlags, int mountExternal,**

**int targetSdkVersion, String seInfo, String abi,**

**String instructionSet, String appDataDir,String[] extraArgs)**

**throws ZygoteStartFailedEx {**

**synchronized(Process.class) {**

**ArrayList<String> argsForZygote = new ArrayList<String>();**

**// --runtime-init, --setuid=, --setgid=,**

**// and --setgroups= must go first**

**//一些参数处理，最后调用zygoteSendArgsAndGetPid函数。**

**argsForZygote.add("--runtime-init"); //这个参数很重要**

**argsForZygote.add("--setuid=" + uid);**

**argsForZygote.add("--setgid=" + gid);**

**if ((debugFlags & Zygote.DEBUG\_ENABLE\_JNI\_LOGGING) != 0) {**

**argsForZygote.add("--enable-jni-logging");**

**}**

**if ((debugFlags & Zygote.DEBUG\_ENABLE\_SAFEMODE) != 0) {**

**argsForZygote.add("--enable-safemode");**

**}**

**if ((debugFlags & Zygote.DEBUG\_ENABLE\_DEBUGGER) != 0) {**

**argsForZygote.add("--enable-debugger");**

**}**

**if ((debugFlags & Zygote.DEBUG\_ENABLE\_CHECKJNI) != 0) {**

**argsForZygote.add("--enable-checkjni");**

**}**

**if ((debugFlags & Zygote.DEBUG\_ENABLE\_ASSERT) != 0) {**

**argsForZygote.add("--enable-assert");**

**}**

**if (mountExternal == Zygote.MOUNT\_EXTERNAL\_MULTIUSER) {**

**argsForZygote.add("--mount-external-multiuser");**

**} else if (mountExternal == Zygote.MOUNT\_EXTERNAL\_MULTIUSER\_ALL) {**

**argsForZygote.add("--mount-external-multiuser-all");**

**}**

**argsForZygote.add("--target-sdk-version=" + targetSdkVersion);**

**//TODO optionally enable debuger**

**//argsForZygote.add("--enable-debugger");**

**// --setgroups is a comma-separated list**

**if (gids != null && gids.length > 0) {**

**StringBuilder sb = new StringBuilder();**

**sb.append("--setgroups=");**

**int sz = gids.length;**

**for (int i = 0; i < sz; i++) {**

**if (i != 0) {**

**sb.append(',');**

**}**

**sb.append(gids[i]);**

**}**

**argsForZygote.add(sb.toString());**

**}**

**if (niceName != null) {**

**argsForZygote.add("--nice-name=" + niceName);**

**}**

**if (seInfo != null) {**

**argsForZygote.add("--seinfo=" + seInfo);**

**}**

**if (instructionSet != null) {**

**argsForZygote.add("--instruction-set=" + instructionSet);**

**}**

**if (appDataDir != null) {**

**argsForZygote.add("--app-data-dir=" + appDataDir);**

**}**

**argsForZygote.add(processClass);**

**if (extraArgs != null) {**

**for (String arg : extraArgs) {**

**argsForZygote.add(arg);**

**}**

**}**

**return zygoteSendArgsAndGetResult(openZygoteSocketIfNeeded(abi), argsForZygote);**

**}**

**}**

**private static ZygoteState openZygoteSocketIfNeeded(String abi) throws ZygoteStartFailedEx**

**{**

**if (primaryZygoteState == null || primaryZygoteState.isClosed()) {**

**try {**

**primaryZygoteState = ZygoteState.connect(ZYGOTE\_SOCKET);**

**} catch (IOException ioe) {**

**throw new ZygoteStartFailedEx("Error connecting to primary zygote", ioe);**

**}**

**}**

**if (primaryZygoteState.matches(abi)) {**

**return primaryZygoteState;**

**}**

**// The primary zygote didn't match. Try the secondary.**

**if (secondaryZygoteState == null || secondaryZygoteState.isClosed()) {**

**try {**

**secondaryZygoteState = ZygoteState.connect(SECONDARY\_ZYGOTE\_SOCKET);**

**} catch (IOException ioe) {**

**throw new ZygoteStartFailedEx("Error connecting to secondary zygote", ioe);**

**}**

**}**

**if (secondaryZygoteState.matches(abi)) {**

**return secondaryZygoteState;**

**}**

**throw new ZygoteStartFailedEx("Unsupported zygote ABI: " + abi);**

**}**

好了，ActivityManagerService终于向Zygote发送请求了。请求的参数中有一个字符串，它的值是“android.app.ActivityThread”。现在该回到Zygote处理请求那块去看看了。

注意：由于ActivityManagerService驻留于SystemServer进程中，所以正是SystemServer向Zygote发送了消息。

4.4.2 有求必应之响应请求

前面有一节，题目叫“有求必应之等待请求”，那么这一节“有求必应之响应请求”会回到ZygoteInit。下面就看看它是如何处理请求的。

**[--->ZygoteInit.java]**

**private static void runSelectLoop(String abiList) throws MethodAndArgsCaller {**

**ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();**

**ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>();**

**FileDescriptor[] fdArray = new FileDescriptor[4];**

**fds.add(sServerSocket.getFileDescriptor());**

**peers.add(null);**

**int loopCount = GC\_LOOP\_COUNT;**

**while (true) {**

**int index;**

**/\***

**\* Call gc() before we block in select().**

**\* It's work that has to be done anyway, and it's better**

**\* to avoid making every child do it. It will also**

**\* madvise() any free memory as a side-effect.**

**\***

**\* Don't call it every time, because walking the entire**

**\* heap is a lot of overhead to free a few hundred bytes.**

**\*/**

**if (loopCount <= 0) {**

**gc();**

**loopCount = GC\_LOOP\_COUNT;**

**} else {**

**loopCount--;**

**}**

**try {**

**fdArray = fds.toArray(fdArray);**

**index = selectReadable(fdArray);**

**} catch (IOException ex) {**

**throw new RuntimeException("Error in select()", ex);**

**}**

**if (index < 0) {**

**throw new RuntimeException("Error in select()");**

**} else if (index == 0) {**

**ZygoteConnection newPeer = acceptCommandPeer(abiList);**

**peers.add(newPeer);**

**fds.add(newPeer.getFileDescriptor());**

**} else {**

**boolean done;**

**done = peers.get(index).runOnce();//调用ZygoteConnection的runOnce**

**if (done) {**

**peers.remove(index);**

**fds.remove(index);**

**}**

**}**

**}**

**}**

每当有请求数据发来时，Zygote都会调用ZygoteConnection的runOnce函数。ZygoteConnection代码在ZygoteConnection.java文件中，来看看它的runOnce函数：

**-->ZygoteConnection.java]**

**boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {**

**......**

**try {**

**args = readArgumentList(); //读取SystemServer发送过来的参数**

**descriptors = mSocket.getAncillaryFileDescriptors();**

**} catch (IOException ex) {**

**Log.w(TAG, "IOException on command socket " + ex.getMessage());**

**closeSocket();**

**return true;**

**}**

**......**

**try {**

**parsedArgs = new Arguments(args);**

**......**

**applyUidSecurityPolicy(parsedArgs, peer, peerSecurityContext);**

**......**

**////根据函数名，可知Zygote又分裂出了一个子进程。**

**pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid, parsedArgs.gids,**

**parsedArgs.debugFlags, rlimits, parsedArgs.mountExternal, parsedArgs.seInfo,**

**parsedArgs.niceName, fdsToClose, parsedArgs.instructionSet,**

**parsedArgs.appDataDir);**

**......**

**try {**

**if (pid == 0) {**

**// in child**

**IoUtils.closeQuietly(serverPipeFd);**

**serverPipeFd = null;**

**//子进程处理，这个子进程是不是我们要创建的Activity对应的子进程呢？**

**handleChildProc(parsedArgs, descriptors, childPipeFd, newStderr);**

**// should never get here, the child is expected to either**

**// throw ZygoteInit.MethodAndArgsCaller or exec().**

**return true;**

**} else {**

**// in parent...pid of < 0 means failure**

**IoUtils.closeQuietly(childPipeFd);**

**childPipeFd = null;**

**//zygote进程**

**return handleParentProc(pid, descriptors, serverPipeFd, parsedArgs);**

**}**

**} finally {**

**IoUtils.closeQuietly(childPipeFd);**

**IoUtils.closeQuietly(serverPipeFd);**

**}**

**......**

**}**

接下来，看看新创建的子进程在handleChildProc中做了些什么。

[-->ZygoteConnection.java]

**private void handleChildProc(Arguments parsedArgs,**

**FileDescriptor[] descriptors, FileDescriptor pipeFd, PrintStream newStderr)**

**throws ZygoteInit.MethodAndArgsCaller {**

**......**

**//根据传入的参数设置新进程的一些属性**

**//SS发来的参数中有“--runtime-init“，所以parsedArgs.runtimeInit为true。**

**if(parsedArgs.runtimeInit) {**

**if (parsedArgs.invokeWith != null) {**

**WrapperInit.execApplication(parsedArgs.invokeWith,**

**parsedArgs.niceName, parsedArgs.targetSdkVersion,**

**pipeFd, parsedArgs.remainingArgs);**

**} else {**

**RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion,**

**parsedArgs.remainingArgs, null /\* classLoader \*/);**

**}**

**}else {**

**......**

**}**

**}**

**public static final void zygoteInit(int targetSdkVersion, String[] argv, ClassLoader classLoader)**

**throws ZygoteInit.MethodAndArgsCaller {**

**if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application from zygote");**

**redirectLogStreams();**

**commonInit();**

**nativeZygoteInit();**

**applicationInit(targetSdkVersion, argv, classLoader);**

**}**

**private static void applicationInit(int targetSdkVersion, String[] argv,**

**ClassLoader classLoader)**

**throws ZygoteInit.MethodAndArgsCaller {**

**nativeSetExitWithoutCleanup(true);**

**VMRuntime.getRuntime().setTargetHeapUtilization(0.75f);**

**VMRuntime.getRuntime().setTargetSdkVersion(targetSdkVersion);**

**final Arguments args;**

**try {**

**args = new Arguments(argv);**

**} catch (IllegalArgumentException ex) {**

**Slog.e(TAG, ex.getMessage());**

**// let the process exit**

**return;**

**}**

**// Remaining arguments are passed to the start class's static main**

**//最终还是调用invokeStaticMain函数，这个函数我们已经见识过了。**

**invokeStaticMain(args.startClass, args.startArgs, classLoader);**

**}**

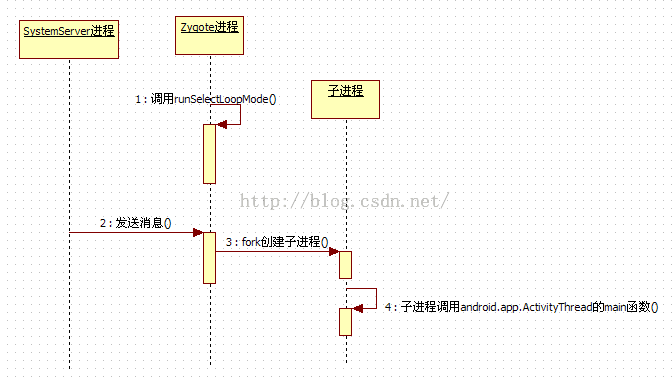
Zygote分裂子进程后，自己将在handleParentProc中做一些扫尾工作，然后继续等待请求进行下一次分裂。

这个android.app.ActivityThread类，实际上是Android中apk程序所对应的进程，它的main函数就是apk程序的main函数。从这个类的命名（android.app）中也可以看出些端倪。

通过这一节的分析，读者可以想到，Android系统运行的那些apk程序，其父都是zygote。这一点，可以通过adb shell登录后，用ps命令查看进程和父进程号来确认。

4.4.3 关于 Zygote分裂的总结

Zygote的分裂由SystemServer控制，这个过程我们用图4-3来表示：



4.5 拓展思考

4.5.1 虚拟机heapsize的限制

在分析Zygote创建虚拟机的时候，我们说过系统默认设置的Java虚拟机堆栈最大为16MB，这个值对于需要使用较大内存的程序（例如图片处理程序）来说还远远不够。当然，可以修改这个默认值，例如我的HTC G7就将其修改为32MB了，但是这个改动是全局性的，也就是所有的Java程序都会是这个32MB。我们能动态配置这个值吗？例如：

· 设置一个配置文件，每个进程启动的时候根据配置文件的参数来设置堆大小。

不过正如前面所说，我的这一美好愿望最终破灭了，原因只有一个：

· Zygote是通过fork来创建子进程的，Zygote本身设置的信息会被子进程全部继承，例如Zygote设置的堆栈为16MB，那么它的子进程也是用这个16MB。

关于这个问题，我目前想到了两个解决方案：

· 为Dalivk增加一个函数，这个函数允许动态调整最大堆的大小。

· Zygote通过fork子进程后，调用exec家族的函数来加载另外一个映像，该映像对应的程序会重新创建虚拟机，重新注册JNI函数，也就是模拟Zygote创世界中前两天的工作，最后调用android.app.ActivityThread的main函数。这种方式应该是可行的，但难度较大，而且会影响运行速度。

关于本节所提出的问题，欢迎广大读者踊跃讨论。

4.5.2 开机速度优化

Android开机速度慢这一现象一直受人诟病，Google好像也没有要做这方面优化的意向，那么，在实际工作中又在哪些地方可以做一些优化呢？根据我目前所掌握的资料分析，有三个地方耗时比较长：

· ZygoteInit的main函数中preloadClasses加载的那一千多个类。

· 开机启动时，会对系统内所有的apk文件扫描并收集信息，这个动作耗费的时间非常长。

· SystemServer创建的那些Service，会占用不少时间。

我们这里讨论第一个问题，如何减少preloadClasses的时间呢？其实，这个函数是可以去掉的，因为系统最终还是会在使用这些类时去加载，但这样就破坏了Android采用fork机制来创建Java进程的本意，而fork机制的好处则是显而易见的：

· Zygote预加载的这些class，在fork子进程时，仅需做一个复制即可。这样就节约了子进程的启动时间。

· 根据fork的copy-on-write机制，有些类如果不做改变，甚至连复制都不用，它们会直接和父进程共享数据。这样就会省去不少内存的占用。

开机速度优化是一项比较复杂的研究，目前有人使用Berkeley Lab Checkpoint/Restart（BLCR）技术来提升开机速度。这一技术的构想其实挺简单，就是对当前系统做一个快照，保存到一个文件中，当系统重启时，直接根据文件的快照信息来恢复重启之前的状态。当然想法很简单，实现却是很复杂的，这里，我们对此不做进一步的讨论了，读者可自行展开深入的思考和研究。

我在VMWare虚拟机上使用过类似的技术，它叫Snapshort。开机速度的问题我更希望Google自己能加以重视并推动它的解决。

4.6 本章小结

本章对Zygote进程做了较为深入的分析，Zygote的主要工作是开创Java世界，本章介绍了它创世纪的七大步骤。另外，本章还分析了Zygote的“嫡长子”——System\_server进程，这个进程是Java世界中的系统Service的驻留地，所以它非常重要。对于System\_server进程，本章重点关注的是它的创建和初始化过程。此外，我们还分析了一个Activity所属进程的创建过程，原来这个进程是由ActivityManagerService发送请求给Zygote，最后由Zygote通过fork的方式创建的。

在本章拓展部分，我们讨论了Dalvik虚拟机对heap大小的设置及其可能的修改方法，另外还探讨了Android系统开机速度的问题。最后，本章还分析了System\_server中Watchdog的工作流程。