# Android系统启动

## 概述

Android系统底层基于Linux Kernel, 当Kernel启动过程会创建init进程, 该进程是用户空间的鼻祖, init进程会启动servicemanager(binder服务管家), Zygote进程(Java进程的鼻祖). Zygote进程会创建 system\_server进程以及各种app进程.



## init

[init](http://gityuan.com/2016/02/05/android-init/)是Linux系统中用户空间的第一个进程(pid=1), Kernel启动后会调用/system/core/init/Init.cpp的main()方法.

root@zs600b:/etc/wifi # ps | grep init

root 1 0 748 392 c01191c8 00035888 S /init

### Init.main

|  |
| --- |
| int main(int argc, char\*\* argv) {  ...  klog\_init(); //初始化kernel log  property\_init(); //创建一块共享的内存空间，用于属性服务  signal\_handler\_init(); //初始化子进程退出的信号处理过程  property\_load\_boot\_defaults(); //加载/default.prop文件  start\_property\_service(); //启动属性服务器(通过socket通信)  init\_parse\_config\_file("/init.rc"); //解析init.rc文件  //执行rc文件中触发器为 on early-init的语句  action\_for\_each\_trigger("early-init", action\_add\_queue\_tail);  //执行rc文件中触发器为 on init的语句  action\_for\_each\_trigger("init", action\_add\_queue\_tail);  //执行rc文件中触发器为 on late-init的语句  action\_for\_each\_trigger("late-init", action\_add\_queue\_tail);  while (true) {  if (!waiting\_for\_exec) {  execute\_one\_command();  restart\_processes();  }  int timeout = -1;  if (process\_needs\_restart) {  timeout = (process\_needs\_restart - gettime()) \* 1000;  if (timeout < 0)  timeout = 0;  }  if (!action\_queue\_empty() || cur\_action) {  timeout = 0;  }  epoll\_event ev;  //循环 等待事件发生  int nr = TEMP\_FAILURE\_RETRY(epoll\_wait(epoll\_fd, &ev, 1, timeout));  if (nr == -1) {  ERROR("epoll\_wait failed: %s\n", strerror(errno));  } else if (nr == 1) {  ((void (\*)()) ev.data.ptr)();  }  }  return 0;  } |

init进程的主要功能点:

* 分析和运行所有的init.rc文件;
* 生成设备驱动节点; （通过rc文件创建）
* 处理子进程的终止(signal方式);
* 提供属性服务property service。

### 启动Zygote

当init解析到下面这条语句,便会启动Zygote进程

|  |
| --- |
| service zygote /system/bin/app\_process -Xzygote /system/bin --zygote --start-system-server  class main //伴随着main class的启动而启动  socket zygote stream 660 root system //创建socket  onrestart write /sys/android\_power/request\_state wake  onrestart write /sys/power/state on  onrestart restart media //当zygote重启时,则会重启media  onrestart restart netd // 当zygote重启时,则会重启netd  } |

当init子进程(Zygote)退出时，会产生SIGCHLD信号，并发送给init进程，通过socket套接字传递数据，调用到wait\_for\_one\_process()方法，根据是否是oneshot，来决定是重启子进程，还是放弃启动。由于缺省模式oneshot=false,因此**Zygote一旦被杀便会再次由init进程拉起**.



接下来,便是进入了Zygote进程.

## Zygote

当[Zygote](http://gityuan.com/2016/02/13/android-zygote/)进程启动后, 便会执行到frameworks/base/cmds/app\_process/App\_main.cpp文件的main()方法. 整个调用流程:

|  |
| --- |
| App\_main.main  AR.start  AR.startVm  AR.startReg  ZygoteInit.main (首次进入Java世界)  registerZygoteSocket  preload  startSystemServer  runSelectLoop |

### App\_main.main

|  |
| --- |
| int main(int argc, char\* const argv[])  {  AppRuntime runtime(argv[0], computeArgBlockSize(argc, argv));  while (i < argc) {  ...//参数解析  }  //设置进程名  if (!niceName.isEmpty()) {  runtime.setArgv0(niceName.string());  set\_process\_name(niceName.string());  }  if (zygote) {  // 启动AppRuntime  runtime.start("com.android.internal.os.ZygoteInit", args, zygote);  }  ...  } |

### AndroidRuntime::start

|  |
| --- |
| void AndroidRuntime::start(const char\* className, const Vector<String8>& options)  {  ALOGD("\n>>>>>> AndroidRuntime START %s <<<<<<\n",  className != NULL ? className : "(unknown)");  ...  // 虚拟机创建  if (startVm(&mJavaVM, &env, zygote) != 0) {  return;  }  onVmCreated(env);  // JNI方法注册  if (startReg(env) < 0) {  return;  }  ...  // 调用ZygoteInit.main()方法[见小节3.3]  env->CallStaticVoidMethod(startClass, startMeth, strArray); |

### ZygoteInit.main

|  |
| --- |
| public static void main(String argv[]) {  try {  ...  registerZygoteSocket(socketName); //为Zygote注册socket  preload(); // 预加载类和资源[见小节3.4]  ...  if (startSystemServer) {  startSystemServer(abiList, socketName);//启动system\_server【见小节3.6】  }  Log.i(TAG, "Accepting command socket connections");  runSelectLoop(abiList); //进入循环模式[见小节3.5]  ...  } catch (MethodAndArgsCaller caller) {  caller.run(); //启动system\_server中会讲到。  }  ...  } |

### ZygoteInit.preload

|  |
| --- |
| static void preload() {  Log.d(TAG, "begin preload");  preloadClasses();  preloadResources();  preloadOpenGL();  preloadSharedLibraries();  WebViewFactory.prepareWebViewInZygote();  Log.d(TAG, "end preload");  } |

### ZygoteInit.runSelectLoop

|  |
| --- |
| private static void runSelectLoop(String abiList) throws MethodAndArgsCaller {  ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();  ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>();    //sServerSocket是socket通信中的服务端，即zygote进程  fds.add(sServerSocket.getFileDescriptor());  peers.add(null);  while (true) {  StructPollfd[] pollFds = new StructPollfd[fds.size()];  for (int i = 0; i < pollFds.length; ++i) {  pollFds[i] = new StructPollfd();  pollFds[i].fd = fds.get(i);  pollFds[i].events = (short) POLLIN;  }  ...  Os.poll(pollFds, -1);  for (int i = pollFds.length - 1; i >= 0; --i) {  //采用I/O多路复用机制，当客户端发出连接请求或者数据处理请求时，跳过continue，执行后面的代码  if ((pollFds[i].revents & POLLIN) == 0) {  continue;  }  if (i == 0) {  //创建客户端连接  ZygoteConnection newPeer = acceptCommandPeer(abiList);  peers.add(newPeer);  fds.add(newPeer.getFileDesciptor());  } else {  //处理客户端数据事务  boolean done = peers.get(i).runOnce();  if (done) {  peers.remove(i);  fds.remove(i);  }  }  }  }  } |

### ZygoteInit.startSystemServer

|  |
| --- |
| private static boolean startSystemServer(String abiList, String socketName)  throws MethodAndArgsCaller, RuntimeException {  ...  // fork子进程system\_server  pid = Zygote.forkSystemServer(  parsedArgs.uid, parsedArgs.gid,  parsedArgs.gids,  parsedArgs.debugFlags,  null,  parsedArgs.permittedCapabilities,  parsedArgs.effectiveCapabilities);  ...  if (pid == 0) {  if (hasSecondZygote(abiList)) {  waitForSecondaryZygote(socketName);  }  //进入system\_server进程[见小节4.1]  handleSystemServerProcess(parsedArgs);  }  return true;  } |

Zygote进程创建Java虚拟机,并注册JNI方法, 真正成为Java进程的母体,用于孵化Java进程. 在创建完[小节4.1]system\_server进程后,zygote功成身退，调用runSelectLoop()，随时待命，当接收到请求创建新进程请求时立即唤醒并执行相应工作。

<http://gityuan.com/2016/02/01/android-booting/>

## system\_server

Zygote通过fork后创建system\_server进程。

### handleSystemServerProcess

|  |
| --- |
| private static void handleSystemServerProcess(  ZygoteConnection.Arguments parsedArgs)  throws ZygoteInit.MethodAndArgsCaller {  ...  if (parsedArgs.niceName != null) {  //设置当前进程名为"system\_server"  Process.setArgV0(parsedArgs.niceName);  }  final String systemServerClasspath = Os.getenv("SYSTEMSERVERCLASSPATH");  if (systemServerClasspath != null) {  //执行dex优化操作,比如services.jar  performSystemServerDexOpt(systemServerClasspath);  }  if (parsedArgs.invokeWith != null) {  ...  } else {  ClassLoader cl = null;  if (systemServerClasspath != null) {  cl = new PathClassLoader(systemServerClasspath, ClassLoader.getSystemClassLoader());  Thread.currentThread().setContextClassLoader(cl);  }  //[见小节4.2]  RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.remainingArgs, cl);  }  } |

system\_server进程创建PathClassLoader类加载器.

### RuntimeInit.zygoteInit

|  |
| --- |
| public static final void zygoteInit(int targetSdkVersion, String[] argv, ClassLoader classLoader)  throws ZygoteInit.MethodAndArgsCaller {  Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "RuntimeInit");  redirectLogStreams(); //重定向log输出  commonInit(); // 通用的一些初始化  nativeZygoteInit(); // zygote初始化  applicationInit(targetSdkVersion, argv, classLoader); // [见小节3.4]  } |

nativeZygoteInit()方法经过层层调用,会进入app\_main.cpp中的onZygoteInit()方法, Binder线程池的创建也是在这个过程,如下:

|  |
| --- |
| virtual void onZygoteInit()  {  sp<ProcessState> proc = ProcessState::self();  proc->startThreadPool(); //启动新binder线程  } |

applicationInit()方法经过层层调用,会抛出异常ZygoteInit.MethodAndArgsCaller(m, argv), ZygoteInit.main() 会捕捉该异常, 见下文.

### ZygoteInit.main

ZygoteInit.java

|  |
| --- |
| public static void main(String argv[]) {  try {  startSystemServer(abiList, socketName); //抛出MethodAndArgsCaller异常  ....  } catch (MethodAndArgsCaller caller) {  caller.run(); //此处通过反射,会调用SystemServer.main()方法 [见小节4.4]  } catch (RuntimeException ex) {  ...  }  } |

采用抛出异常的方式,用于栈帧清空,提供利用率, 以至于现在大家看到的每个Java进程的调用栈如下:

|  |
| --- |
| ...  at com.android.server.SystemServer.main(SystemServer.java:175)  at java.lang.reflect.Method.invoke!(Native method)  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:738)  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:628) |

### SystemServer.main

SystemServer.java

|  |
| --- |
| public final class SystemServer {  ...  public static void main(String[] args) {  //先初始化SystemServer对象，再调用对象的run()方法  new SystemServer().run();  }  } |

### SystemServer.run

SystemServer.java

|  |
| --- |
| private void run() {  if (System.currentTimeMillis() < EARLIEST\_SUPPORTED\_TIME) {  Slog.w(TAG, "System clock is before 1970; setting to 1970.");  SystemClock.setCurrentTimeMillis(EARLIEST\_SUPPORTED\_TIME);  }  ...  Slog.i(TAG, "Entered the Android system server!");  EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_SYSTEM\_RUN, SystemClock.uptimeMillis());  Looper.prepareMainLooper();// 准备主线程looper  //加载android\_servers.so库，该库包含的源码在frameworks/base/services/目录下  System.loadLibrary("android\_servers");  //检测上次关机过程是否失败，该方法可能不会返回[见小节3.6.1]  performPendingShutdown();  createSystemContext(); //初始化系统上下文  //创建系统服务管理  mSystemServiceManager = new SystemServiceManager(mSystemContext);  LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);  //启动各种系统服务[见小节3.7]  try {  startBootstrapServices(); // 启动引导服务  startCoreServices(); // 启动核心服务  startOtherServices(); // 启动其他服务[见小节4.6]  } catch (Throwable ex) {  Slog.e("System", "\*\*\*\*\*\*\*\*\*\*\*\* Failure starting system services", ex);  throw ex;  }  //一直循环执行  Looper.loop();  throw new RuntimeException("Main thread loop unexpectedly exited");  } |

### 服务启动

|  |
| --- |
| public final class SystemServer {  private void startBootstrapServices() {  ...  //phase100  mSystemServiceManager.startBootPhase(SystemService.PHASE\_WAIT\_FOR\_DEFAULT\_DISPLAY);  ...  }  private void startOtherServices() {  ...  //phase480 和phase500  mSystemServiceManager.startBootPhase(SystemService.PHASE\_LOCK\_SETTINGS\_READY);  mSystemServiceManager.startBootPhase(SystemService.PHASE\_SYSTEM\_SERVICES\_READY);  ...  //[见小节4.7]  mActivityManagerService.systemReady(new Runnable() {  @Override  public void run() {  //phase550  mSystemServiceManager.startBootPhase(  SystemService.PHASE\_ACTIVITY\_MANAGER\_READY);  ...  //phase600  mSystemServiceManager.startBootPhase(  SystemService.PHASE\_THIRD\_PARTY\_APPS\_CAN\_START);  }  }  }  } |

* start: 创建AMS, PMS, LightsService, DMS.
* phase100: 进入Phase100, 创建PKMS, WMS, IMS, DBMS, LockSettingsService, JobSchedulerService, MmsService等服务;
* phase480 && 500: 进入Phase480, 调用WMS, PMS, PKMS, DisplayManagerService这4个服务的systemReady();
* Phase550: 进入phase550, 执行AMS.systemReady(), 启动SystemUI, WebViewFactory, Watchdog.
* Phase600: 进入phase600, 执行AMS.systemReady(), 执行各种服务的systemRunning().
* Phase1000: 进入1000, 执行finishBooting, 启动启动on-hold进程.

### AMS.systemReady

|  |
| --- |
| public final class ActivityManagerService extends ActivityManagerNative  implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {    public void systemReady(final Runnable goingCallback) {  ... //update相关  mSystemReady = true;    //杀掉所有非persistent进程  removeProcessLocked(proc, true, false, "system update done");  mProcessesReady = true;  goingCallback.run(); //[见小节1.6.2]    addAppLocked(info, false, null); //启动所有的persistent进程  mBooting = true;    //启动home  startHomeActivityLocked(mCurrentUserId, "systemReady");  //恢复栈顶的Activity  mStackSupervisor.resumeTopActivitiesLocked();  }  } |

System\_server主线程的启动工作,总算完成, 进入Looper.loop()状态,等待其他线程通过handler发送消息再处理.

## App

对于普通的app进程,跟system\_server进程的启动过来有些类似.不同的是app进程是向发消息给system\_server进程, 由system\_server向zygote发出创建进程的请求.

[理解Android进程创建流程](http://gityuan.com/2016/03/26/app-process-create/), 可知进程创建后 接下来会进入ActivityThread.main()过程。

### ActivityThread.main

|  |
| --- |
| public static void main(String[] args) {  ...  Environment.initForCurrentUser();  ...  Process.setArgV0("<pre-initialized>");  //创建主线程looper  Looper.prepareMainLooper();  ActivityThread thread = new ActivityThread();  thread.attach(false); //attach到系统进程  if (sMainThreadHandler == null) {  sMainThreadHandler = thread.getHandler();  }    //主线程进入循环状态  Looper.loop();  throw new RuntimeException("Main thread loop unexpectedly exited");  } |

app进程的主线程调用栈的栈底如下:

|  |
| --- |
| ...  at android.app.ActivityThread.main(ActivityThread.java:5442)  at java.lang.reflect.Method.invoke!(Native method)  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:738)  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:628) |

跟前面介绍的system\_server进程调用栈对比:

|  |
| --- |
| at com.android.server.SystemServer.main(SystemServer.java:175)  at java.lang.reflect.Method.invoke!(Native method)  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:738)  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:628) |

**六. 实战分析**

以下列举启动部分重要进程以及关键节点会打印出的log

/system/bin/vold: 383

/system/bin/lmkd: 432

/system/bin/surfaceflinger: 434

/system/bin/debuggerd64: 537

/system/bin/mediaserver: 540

/system/bin/installd: 541

/system/vendor/bin/thermal-engine: 552

zygote64: 557

zygote: 558

system\_server: 1274

**1. before zygote**

//启动vold, 再列举当前系统所支持的文件系统. 执行到system/vold/main.cpp的main()

11-23 14:36:47.474 383 383 I vold : Vold 3.0 (the awakening) firing up

11-23 14:36:47.475 383 383 V vold : Detected support for: ext4 vfat

//使用内核的lmk策略

11-23 14:36:47.927 432 432 I lowmemorykiller: Using in-kernel low memory killer interface

//启动SurfaceFlinger

11-23 14:36:48.041 434 434 I SurfaceFlinger: SurfaceFlinger is starting

11-23 14:36:48.042 434 434 I SurfaceFlinger: SurfaceFlinger's main thread ready to run. Initializing graphics H/W...

// 开机动画

11-23 14:36:48.583 508 508 I BootAnimation: bootanimation launching ...

// debuggerd

11-23 14:36:50.306 537 537 I : debuggerd: starting

// installd启动

11-23 14:36:50.311 541 541 I installd: installd firing up

// thermal守护进程

11-23 14:36:50.369 552 552 I ThermalEngine: Thermal daemon started

**2. zygote**

// Zygote64进程(Zygote): AndroidRuntime::start

11-23 14:36:51.260 557 557 D AndroidRuntime: >>>>>> START com.android.internal.os.ZygoteInit uid 0 <<<<<<

// Zygote64进程: AndroidRuntime::startVm

11-23 14:36:51.304 557 557 D AndroidRuntime: CheckJNI is OFF

// 执行ZygoteInit.preload()

11-23 14:36:52.134 557 557 D Zygote : begin preload

// 执行ZygoteInit.preloadClasses(), 预加载3860个classes, 花费时长746ms

11-23 14:36:52.134 557 557 I Zygote : Preloading classes...

11-23 14:36:52.881 557 557 I Zygote : ...preloaded 3860 classes in 746ms.

// 执行ZygoteInit.preloadClasses(), 预加载86组资源, 花费时长179ms

11-23 14:36:53.114 557 557 I Zygote : Preloading resources...

11-23 14:36:53.293 557 557 I Zygote : ...preloaded 86 resources in 179ms.

// 执行ZygoteInit.preloadSharedLibraries()

11-23 14:36:53.494 557 557 I Zygote : Preloading shared libraries...

11-23 14:36:53.503 557 557 D Zygote : end preload

// 执行com\_android\_internal\_os\_Zygote\_nativeForkSystemServer(),成功fork出system\_server进程

11-23 14:36:53.544 557 557 I Zygote : System server process 1274 has been created

// Zygote开始进入runSelectLoop()

11-23 14:36:53.546 557 557 I Zygote : Accepting command socket connections

**3. system\_server**

//进入system\_server, 建立跟Zygote进程的socket通道

11-23 14:36:53.586 1274 1274 I Zygote : Process: zygote socket opened, supported ABIS: armeabi-v7a,armeabi

// 执行SystemServer.run()

11-23 14:36:53.618 1274 1274 I SystemServer: Entered the Android system server! <===> boot\_progress\_system\_run

// 等待installd准备就绪

11-23 14:36:53.707 1274 1274 I Installer: Waiting for installd to be ready.

//服务启动

11-23 14:36:53.732 1274 1274 I ActivityManager: Memory class: 192

//phase100

11-23 14:36:53.883 1274 1274 I SystemServiceManager: Starting phase 100

11-23 14:36:53.902 1274 1274 I SystemServer: Package Manager

11-23 14:37:03.816 1274 1274 I SystemServer: User Service

...

11-23 14:37:03.940 1274 1274 I SystemServer: Init Watchdog

11-23 14:37:03.941 1274 1274 I SystemServer: Input Manager

11-23 14:37:03.946 1274 1274 I SystemServer: Window Manager

...

11-23 14:37:04.081 1274 1274 I SystemServiceManager: Starting com.android.server.MountService$Lifecycle

11-23 14:37:04.088 1274 2717 D MountService: Thinking about reset, mSystemReady=false, mDaemonConnected=true

11-23 14:37:04.088 1274 1274 I SystemServiceManager: Starting com.android.server.UiModeManagerService

11-23 14:37:04.520 1274 1274 I SystemServer: NetworkTimeUpdateService

//phase480 && 500

11-23 14:37:05.056 1274 1274 I SystemServiceManager: Starting phase 480

11-23 14:37:05.061 1274 1274 I SystemServiceManager: Starting phase 500

11-23 14:37:05.231 1274 1274 I ActivityManager: System now ready <==> boot\_progress\_ams\_ready

11-23 14:37:05.234 1274 1274 I SystemServer: Making services ready

11-23 14:37:05.243 1274 1274 I SystemServer: WebViewFactory preparation

//phase550

11-23 14:37:05.234 1274 1274 I SystemServiceManager: Starting phase 550

11-23 14:37:05.237 1274 1288 I ActivityManager: Force stopping com.android.providers.media appid=10010 user=-1: vold reset

//Phase600

11-23 14:37:06.066 1274 1274 I SystemServiceManager: Starting phase 600

11-23 14:37:06.236 1274 1274 D MountService: onStartUser 0

**4. logcat小技巧**

通过adb bugreport抓取log信息.先看zygote是否起来, 再看system\_server主线程的运行情况,再看ActivityManager情况

adb logcat -s Zygote

adb logcat -s SystemServer

adb logcat -s SystemServiceManager

adb logcat | grep "1359 1359" //system\_server情况

adb logcat -s ActivityManager

现场调试命令

1. cat proc/[pid]/stack ==> 查看kernel调用栈
2. debuggerd -b [pid] ==> 也不可以不带参数-b, 则直接输出到/data/tombstones/目录
3. kill -3 [pid] ==> 生成/data/anr/traces.txt文件
4. lsof [pid] ==> 查看进程所打开的文件

**七. 总结**

各大核心进程启动后，都会进入各种对象所相应的main()方法，如下

| **进程** | **主方法** |
| --- | --- |
| init进程 | Init.main() |
| zygote进程 | ZygoteInit.main() |
| app\_process进程 | RuntimeInit.main |
| system\_server进程 | SystemServer.main() |
| app进程 | ActivityThread.main() |

注意其中app\_process进程是指通过/system/bin/app\_process来启动的进程，且后面跟的参数不带–zygote，即并非启动zygote进程。 比如常见的有通过adb shell方式来执行am,pm等命令，便是这种方式。

关于重要进程重启的过程，会触发哪些关联进程重启名单：

* zygote：触发media、netd以及子进程(包括system\_server进程)重启；
* system\_server: 触发zygote重启;
* surfaceflinger：触发zygote重启;
* servicemanager: 触发zygote、healthd、media、surfaceflinger、drm重启

所以，surfaceflinger,servicemanager,zygote自身以及system\_server进程被杀都会触发Zygote重启。

## REF

[Android系统启动-概述](http://gityuan.com/2016/02/01/android-booting/)