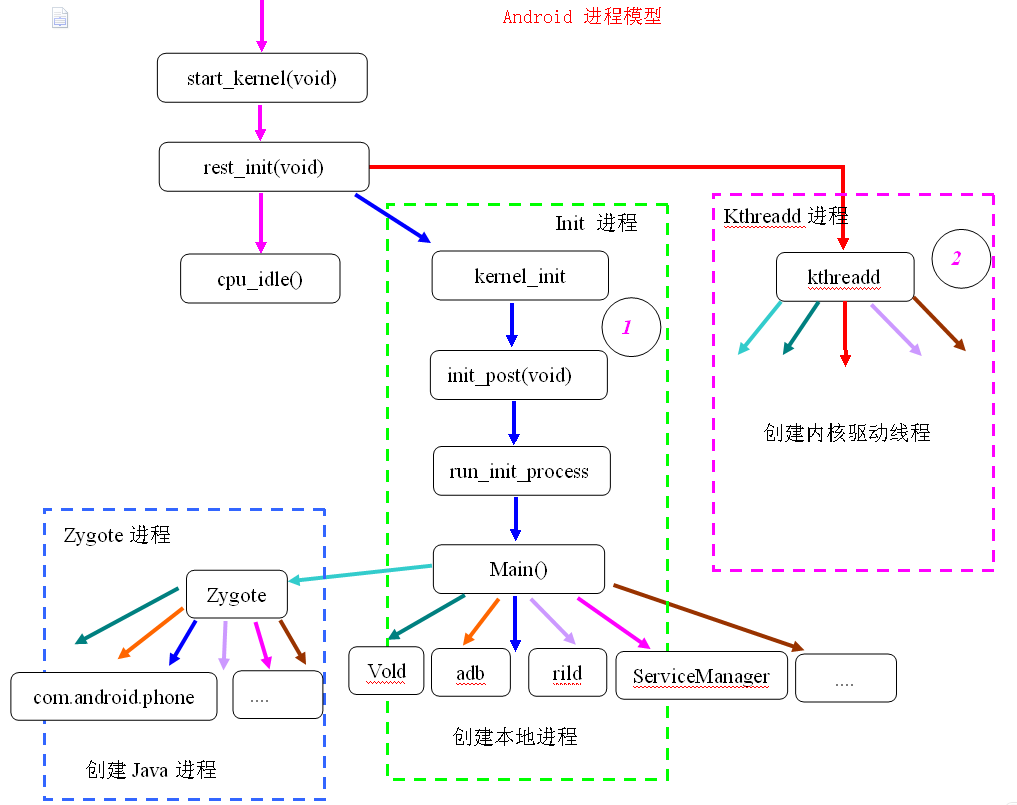
# Init

基于Linux内核的android系统，在内核启动完成后将创建一个Init用户进程，实现了内核空间到用户空间的转变。在[Android 启动过程介绍](http://blog.csdn.net/yangwen123/article/details/8023654)一文中介绍了Android系统的各个启动阶段，init进程启动后会读取init.rc配置文件，通过fork系统调用启动init.rc文件中配置的各个Service进程。init进程首先启动启动android的服务大管家ServiceManager服务，然后启动Zygote进程。Zygote进程的启动开创了Java世界，无论是SystemServer进程还是android的应用进程都是Zygote的子进程，[Zygote进程启动过程的源代码分析](http://blog.csdn.net/yangwen123/article/details/8059995)一文中详细介绍了Zygote进程的启动过程，[System Server进程启动过程源码分析](http://blog.csdn.net/yangwen123/article/details/8060156)则详细介绍了在Zygote进程启动完成后创建的第一个进程SystemServer进程的启动过程，SystemServer进程的启动包括两个阶段，在第一阶段主要是启动C++相关的本地服务，如SurfaceFlinger等，在第二阶段通过在ServerThread线程中启动android的各大关键Java服务。[Zygote孵化应用进程过程的源码分析](http://blog.csdn.net/yangwen123/article/details/8080924)一文中详细介绍了Zygote进程创建android应用进程的过程，当用户点击Luncher上的应用图标时，Luncher进程通过socket向Zygote进程发送进程创建请求，Zygote进程接受客户端的请求后，通过fork系统调用为应用程序创建相应的进程。本文则介绍android用户进程的始祖Init进程，Init进程是Linux系统中用户空间的第一个进程，负责创建系统中的关键进程，同时提供属性服务来管理系统属性。

## Android进程模型

Linux通过调用start\_kernel函数来启动内核，当内核启动模块启动完成后，将启动用户空间的第一个进程——Init进程，下图为Android系统的进程模型图：



从上图可以看出，Linux内核在启动过程中，创建一个名为kthreadd的内核进程，PID=2，用于创建内核空间的其他进程；同时创建第一个用户空间Init进程，该进程PID = 1，用于启动一些本地进程，比如Zygote进程，而Zygote进程也是一个专门用于孵化Java进程的本地进程，上图清晰地描述了整个Android系统的进程模型，为了证明以上进程模型的正确性，可以通过ps命令来查看进程的PID级PPID，下图显示了Init进程的PID为1，其他的本地进程的PPID都是1，说明它们的父进程都是Init进程，都是由Init进程启动的。

root@gl300e:/dev # ps

USER PID PPID VSIZE RSS WCHAN PC NAME

**root 1 0 744 404 c0118a30 0003503c S /init**

**root 2 0 0 0 c0053f24 00000000 S kthreadd**

root 3 2 0 0 c005acc4 00000000 S ksoftirqd/0

root 5 2 0 0 c004e538 00000000 S kworker/0:0H

logd 145 1 8704 2116 ffffffff b6f97408 S /system/bin/logd

system 148 1 1216 352 c0495dc4 b6f71490 S /system/bin/servicemanager

root 150 1 6936 1892 ffffffff b6ef0a68 S /system/bin/vold

system 151 1 258108 15032 ffffffff b6f837dc S /system/bin/surfaceflinger

**root 164 1 1362960 42992 ffffffff b6e54300 S zygote**

u0\_a40 1392 164 734392 31916 ffffffff b6e547dc S dji.system.upgrade

u0\_a15 1607 164 1461176 62568 ffffffff b6e547dc S com.twitter.android

## 源码分析

<http://blog.csdn.net/yangwen123/article/details/9029959>

<http://blog.csdn.net/gaugamela/article/details/52133186>?

# 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。

# 属性系统build.prop

## 概述

### build.prop是什么？

在Android中.prop类型的文件是一个属性文件，记录一些系统设置。.prop文件类似/etc中的文件。更简单的来说，就相当于Windows中的系统注册表。所以，build.prop中记录了一些系统的设置属性。这些属性包括系统初始或固定的一些参数属性、功能的开放等。通过调整/增加这些参数可以达到较调系统性能偏重点和附加功能开启的作用。

### build.prop是怎样生成的？

首先需要确定build.prop在编译过程中是由哪个Makefile、.mk或编译脚本控制的。熟悉Android编译过程的可能很容易定位到./build/core/Makefile这个文件。如果不熟悉Android编译过程可以通过查看Android整编的log。在./build/core/Makefile中，可以看到：

# build.prop

INSTALLED\_BUILD\_PROP\_TARGET := $(TARGET\_OUT)/build.prop

ALL\_DEFAULT\_INSTALLED\_MODULES += $(INSTALLED\_BUILD\_PROP\_TARGET)

ADDITIONAL\_BUILD\_PROPERTIES := \

$(call collapse-pairs, $(ADDITIONAL\_BUILD\_PROPERTIES))

ADDITIONAL\_BUILD\_PROPERTIES := $(call uniq-pairs-by-first-component, \

$(ADDITIONAL\_BUILD\_PROPERTIES),=)

ifdef TARGET\_SYSTEM\_PROP

system\_prop\_file := $(TARGET\_SYSTEM\_PROP)

else

system\_prop\_file := $(wildcard $(TARGET\_DEVICE\_DIR)/system.prop)

endif

通过Makefile文件，可以得到，build.prop的来源有：

* ./build/tools/buildinfo.sh：将buildinfo.sh的内容，直接echo到build.prop中
* system.prop：把默认的system.prop内容追加到build.prop中。如，./device/jrdcom/Alto5TF/system.prop
* ADDITIONAL\_BUILD\_PROPERTIES：把ADDITIONAL\_BUILD\_PROPERTIES中的属性追加到build.prop
* ODM/OED厂商的定制：一些厂商一般都会有自己的一些额外定制

### build.prop读/写

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| Code-java | 主要通过SystemProperties中提供的get相关的方法；者直接用类**Build**.java。 | get(String key, String def)  **Build.XXX** | frameworks\base\core\[**Java**](http://lib.csdn.net/base/java)\[**android**](http://lib.csdn.net/base/android)\os\ SystemProperties.java调用jni  \frameworks\base\core\jni\android\_os\_SystemProperties.cpp |  |
| C/C++层 |  | property\_get("xxxx", value, "0"); |  |  |
| Adb命令 |  | adb shell getprop xxxx | adb shell setprop xxxx  临时修改，不重启系统 |  |
| 文件修改 | /system/build.prop | Pull/push要修改build.prop权限。可以改为644，重启生效。 |  |  |
|  |  |  |  |  |

### key前缀分类

系统属性根据不同的应用类型，分为不可变型，持久型，网络型，启动和停止服务等。

#### ro

属性名称以“ro.”开头，read-only那么这个属性被视为只读属性。一旦设置，属性值不能改变。

root@gl300e:/data/property # setprop ro.test 1

root@gl300e:/data/property # getprop | grep ro.test

[ro.test]: [1]

root@gl300e:/data/property # setprop ro.test 2

root@gl300e:/data/property # getprop | grep ro.test

[ro.test]: [1]

生命周期：存在于内存上，手机一旦关机这些属性值，也不存在了，

#### persist.

属性名称以“persist.”开头，当设置这个属性时，其值也将写入/data/property。

root@gl300e:/data/property # ls

persist.country.code

persist.dji.install.go

persist.hdmi.enable

其他类型的属性除了位于属性文件中的属性值之外，手机一旦关机这些属性值都会丢失，不能持久存在

#### net.

属性名称以“net.”开头，当设置这个属性时，“net.change”属性将会自动设置，以加入到最后修改的属性名。（这是很巧妙的。 netresolve模块的使用这个属性来追踪在net.\*属性上的任何变化。）

1|root@gl300e:/ # setprop net.test 0

root@gl300e:/ # getprop | grep net.test

[net.change]: [net.test]

[net.test]: [0]

root@gl300e:/ # setprop net.test2 2

root@gl300e:/ # getprop | grep net.test

[net.change]: [net.test2]

[net.test2]: [2]

[net.test]: [0]

#### ctrl.start/stop

　　属性“ ctrl.start ”和“ ctrl.stop ”是用来启动和停止服务。每一项服务必须在/init.rc中定义.系统启动时，与init守护进程将解析init.rc和启动属性服务。一旦收到设置“ ctrl.start ”属性的请求，属性服务将使用该属性值作为服务名找到该服务，启动该服务。这项服务的启动结果将会放入“ init.svc.<服务名>“属性中 。客户端应用程序可以轮询那个属性值，以确定结果。

#### 定制

1. 设置了一个属性值，广播出来？.brocast.
2. 设置meta.

## 参数详解

build.prop中参数包括

1. Dalvik虚拟机相关参数
2. 系统版本、定义相关参数、
3. 基本性能相关参数、
4. 基本耗电相关参数、
5. 扩展性能较调及附加功能开启

### Dalvik虚拟机相关参数

**dalvik.vm.** **heapstartsize**

本参数控制Dalvik虚拟机在启动一个应用程序之后为其分配的初始堆栈大小，可填写的值为**2m~48m**。例如：dalvik.vm.heapstartsize=8m，就表示应用程序启动后为其分配的初始堆栈大小为8兆字节。这里分配的内存容量会影响到整个系统对RAM的使用程度，和第一次使用应用程序时的流畅程序。这个值越大，系统消耗RAM则越快，但是应用程序打开后的反应也越快。值越小，系统的RAM剩余则越多，但是程序在启动后会很卡。建议值是8m，既可以保持140M左右的RAM，程序的反应速度也会大幅度提高。

**dalvik.vm.heapsize**

本参数控制Dalvik虚拟机给一个应用程序分配的最大堆栈量，可填写的值为12m~48m。例如：dalvik.vm.heapsize=48m，就表示应用程序在任意时刻内可以使用的最大堆栈大小为48兆字节。这里分配的内存容量会影响到整个系统对RAM的使用程序，和程序在运行一段时间后的反应速度。**这个值越大，系统消耗RAM则越快，但是程序会运行的非常稳定**，尤其是游戏和视频程序的内容加载速度可以大幅度提升。值越小，系统的RAM剩余则越多，但是程序会很卡，尤其是游戏在切换场景Loading的时候会花费很多的时间。若应用程序需要使用超过这个值的内存时，将会触发系统的垃圾收集器，系统和程序就会卡顿。建议值是40~40m。

**dalvik.vm.lockprof.threshold**

本参数控制Dalvik虚拟机调试记录程序内部锁资源争夺的阈值，默认值是500。多用于程序的数据统计，对性能较调意义不大。

**dalvik.vm.stack-trace-file**

本参数控制Dalvik虚拟机的堆栈记录调试文件。用于系统调试，一般用户对其调整无意义,默认为/data/anr/traces.txt，这是开发者分析anr的重要文件。

**dalvik.vm.execution-mode**

本参数控制Dalvik虚拟机的程序执行机制。可填写的值有"int:portable"、"int:fast"和"int:jit"。int:portable表示以兼容模式运行(脚本翻译模式)，此模式下程序的兼容性最高，但其执行效率最低(程序优化度依赖于dalvik虚拟机版本)。官方默认此模式。int:fast表示以快速自优化模式运行(脚本翻译和预优化混合)，此模式下程序的兼容性很高，执行效率也比较高。因为此时dalvik虚拟机允许程序使用自己的预定义优化模式和代码(包括C/C  /汇编代码)。推荐使用。**int:jit表示以Just-In-Time**模式运行(JIT模式)，此模式下程序的兼容性最差，**但程序一旦加载后其运行效率最高(与C/C  直接编写的程序效率无异)**，因为在此模式下dalvik虚拟机会预先将Java程序翻译成针对机器平台的本地语言(Native)，同时完全允许代码中的所有预优化和代码，**允许所有不安全的非托管代码，同时不严谨的程序如果运行在JIT模式可能会造成内存泄露**。但要注意，**很多Dalvik虚拟机并不支持此模式**(如官方2.2)。

**dalvik.vm.dexopt-flags**

本参数控制Dalvik虚拟机的程序代码校验和优化。可填写的值有m、v和o。m为标准选项，可以是m=y或m=n，若m=y则启用不安全代码的校验和托管代码的优化，兼容性和安全性最高，推荐使用；

v为校验选项，可与o并存，可以是v=a或v=n。若v=a则表示校验所有代码，v=n则关闭代码的校验

o为优化选项，可与v并存。可以是o=v或o=a。若o=v则表示优化以校验过的代码，o=a则表示优化所有代码。例如：dalvik.vm.dexopt-flags=m=ydalvik.vm.dexopt-flags=v=n,o=v注意，这个参数只会影响到安装APK之后或初次使用APK时生成dex文件时有效。若整个系统(包括应用程序)为odex化，则无意义。

**dalvik.vm.verify-bytecode**

本参数控制Dalvik虚拟机是否验证应用程序的可执行代码。可以与上一个参数配合使用。可填写的值为true和false。其具体意义与dalvik.vm.dexopt-flags的v=n一模一样。但可以与dalvik.vm.dexopt-flags配合使用以取得更好的效果。例如：dalvik.vm.dexopt-flags=v=n,o=vdalvik.vm.verify-bytecode=false这样可以令后来安装的apk文件可以被优化而不被检验。dalvik.vm.checkjni，本参数控制Dalvik虚拟机在调用外部jni链接库的时候是否对其做安全性检验。可填写的值为true和false。此参数会覆盖ro.kernel.android.checkjni。若值为true，会增加程序的兼容性和稳定性，但也会增加其加载和执行的时间。推荐为false。

**dalvik.vm.deadlock-predict**

本参数控制Dalvik虚拟机对程序死锁预测处理。可填写的值有off、warn和err。off表示关闭死锁预测功能(默认设置)。warn表示在继续程序运行的同时只记录该死锁预测(如果为真死锁就会出现程序假死现象，然后等N久出现关闭)。err表示预测到死锁时马上弹出FC。注意：有些Dalvik虚拟机版本并不支持此参数。

超级急速流畅型

dalvik.vm.startheapsize=16m

dalvik.vm.heapsize=48m

dalvik.vm.execution-mode=int:jit

dalvik.vm.dexopt-flags=v=n,o=v

dalvik.vm.verify-bytecode=false

dalvik.vm.checkjni=false

常用稳定加流畅型：

dalvik.vm.startheapsize=8m

dalvik.vm.heapsize=40m

dalvik.vm.execution-mode=int:fast

dalvik.vm.dexopt-flags=m=y

dalvik.vm.verify-bytecode=false

dalvik.vm.checkjni=false

超级稳定大内存型：

dalvik.vm.startheapsize=4m

dalvik.vm.heapsize=30m

dalvik.vm.execution-mode=int:portable

dalvik.vm.dexopt-flags=v=a,o=v

dalvik.vm.verify-bytecode=true

dalvik.vm.checkjni=true

**(以荣耀一代为例，RAM 512M，以下是像我一样追求速度的童鞋的推荐参数)**  
**dalvik.vm.startheapsize=16m  
dalvik.vm.heapsize=64m（RAM大的童鞋这个参数可以再大一些）  
dalvik.vm.execution-mode=int:jit  
dalvik.vm.dexopt-flags=v=n,o=v  
dalvik.vm.verify-bytecode=false  
dalvik.vm.checkjni=false**

### 系统版本、定义等参数

**ro.build.id**

本参数定义了系统的版本ID。为系统内部使用，OTA时作为粗略版本比较。更改后可避免OTA提示，但可能会引起预装程序(如Blur)的稳定性。

ro.build.type

*user:性能好,Log/Debug信息少,相当于是正式版*

*//eng:Log/Debug信息相对多*

*//userdebug:debug强*

**ro.build.display.id**

本参数定义了设置中显示的系统版本号。主要用于设置中显式出现可读版本，一般用于个性化定制和第三方应用程序对系统版本的判断(如魔趣设置)。更改后可自定义版本显示，但某些第三方应用程序会出现错误(如魔趣设置无法实现机器保修查询)。

**ro.build.version.incremental**

本参数定义了系统的升级字。主要用于系统OTA精确版本比对，同时与ro.build.description和ro.build.fingerprint相匹配。更改后可以免OTA提示(如避免Miui的升级提示和Blur的升级提示)。

#### ro.product.model

本参数定义了机器的型号字符串。主要用于机器型号**显式定义(如系统设置中的手机型号和Blur、Google设置向导中的机型等)**。更改后可自定义手机型号名称，供用户界面显示。

*/\*\* The end-user-visible name for the end product. \*/***public static final** String ***MODEL*** = *getString*(**"ro.product.model"**);

如小米的

C:\Users\key.guan>adb shell

sagit:/ $ getprop | grep product

[ro.boot.product.region]: [cn]

[ro.build.product]: [sagit]

[ro.product.board]: [msm8998]

[ro.product.brand]: [Xiaomi]

[ro.product.cert]: [MCE16]

[ro.product.cpu.abi]: [arm64-v8a]

[ro.product.cpu.abilist]: [arm64-v8a,armeabi-v7a,armeabi]

[ro.product.cpu.abilist32]: [armeabi-v7a,armeabi]

[ro.product.cpu.abilist64]: [arm64-v8a]

[ro.product.cuptsm]: [XIAOMI|ESE|02|01]

[ro.product.device]: [sagit]

[ro.product.first\_api\_level]: [25]

[ro.product.locale]: [zh-CN]

[ro.product.manufacturer]: [Xiaomi]

[**ro.product.model**]: [MI 6]

[ro.product.name]: [sagit]

*/\*\* The name of the overall product. \*/***public static final** String ***PRODUCT*** = *getString*(**"ro.product.name"**);  
  
*/\*\* The name of the industrial design. \*/***public static final** String ***DEVICE*** = *getString*(**"ro.product.device"**);  
  
*/\*\* The name of the underlying board, like "goldfish". \*/***public static final** String ***BOARD*** = *getString*(**"ro.product.board"**);

# ro.build.product is obsolete; use ro.product.device （旧代码ro.build.product，使用代码ro.product.device）

**ro.product.locale.language**

本参数定义了系统的初始(默认)语言。此处注意是语言，如中文是zh，英文是en。更改后改变系统**初次启动**时的语言设置。

**ro.product.locale.region**

本参数定义了系统的初始(默认)区域。此处注意是区域，如中国大陆为CN，台湾为TW，美国为US。更改后改变系统初次启动时的区域设置。

**ro.build.description和ro.build.fingerprint**

ROM的编译综合说明。其中包含了平台硬件、Android版本、源代码分支和标签、OTA详细版本等。其中的OTA部分，例如：umts\_jordan\_china-user 2.3.6 4.5.3-109\_DPP-141323416413release-keys将此数字与ro.build.version.incremental一同更改可避免OTA升级提醒(如Miui和Blur等)。

### 基本性能相关参数

**windowsmgr.max\_events\_per\_sec**

本参数定义了Android系统的窗体事件管理器在单位时间内可以处理的最大事件数量。通过更改本参数可以获得非常明显的丝滑流畅体验。可填写的值范围为“大于0的正整数”，官方默认为60。建议150、200、260、300这几个值。当此值变大时，系统触控平滑度明显提高，但对应的CPU使用率也会升高，最终的结果就是电池续航能力下降。以我个人的经验来说，此值取到240左右时在系统设置中滑动可以得到接近WP7的流畅和平滑度。

**ro.min\_pointer\_dur**

本参数定义了两次触摸之间的最短时间间隔，单位是毫秒。默认值为25，推荐值是10。通过调整此参数可以提高系统触控的灵敏度或稳定度。当此值越大时，触控越稳定。此值越小，触控越灵敏。

**mot.proximity.delay**

本参数定义了手机光纤感应器的抖动消除时间，单位是毫秒。默认值是500，推荐值是250。通过调整此参数可以提高在通话结束后屏幕点亮的速度。当此值越大时，通话结束后屏幕点亮所需要的时间越长，但在通话过程中如果手机意外瞬间离开脸部也不会点亮屏幕，可防止通话过程中的误操作(比方说通话时不小心手机移动了一下，屏幕就会点亮，此时如果脸部触碰到了屏幕就会对通话造成影响)。此值越小，则当手机离开脸部或装入口袋后会立即点亮或关闭屏幕。

**mot.proximity.distance**

本参数定义了手机屏幕上的两个触摸点之间的最短距离，若距离小于此值则认为是一个触摸点，单位是像素。默认值是60，推荐值是100。为什么推荐100呢？因为Defy的屏幕分辨率为480x854，也就是说横向有480个像素点，对应上去也就相当于是横向并排允许4个触摸点，平均一个手指一个点，这样在类似于杀西瓜等游戏中可以提升游戏操作。

**ro.kernel.android.checkjni**

本参数定义了Dalvik虚拟机在执行程序的时候是否要做Jni链接库的检查工作。详细见Dalvik参数属性期。若考虑稳定性可使用true，若需要性能可使用false。注意：此参数会被Dalvik参数覆盖。

**ro.media.enc.jpeg.quality**

本参数定义了JPEG图像编码器所使用的质量因子，可填写的值为1~100，默认为80，推荐为100。想照出更好的照片吗？想让照片的大小轻松上M吗？那就使用100吧。

**debug.sf.hw**

本参数定义了系统是否启用GPU来渲染程序的UI，默认为0，推荐为1。但要注意，如果此值为1，在某些应用程序中可能会出现显示错乱的现象(极少见)。

**persist.sys.use\_dithering**

本参数定义了系统渲染器对图像的缩放是否启用抖动技术。可填写的值为0或1。当开启抖动后，图像的显示(指背景、解锁等的图像，并非图库、相机那些的)会很柔和，但会增加CPU负载，最终导致ROM卡顿。

**persist.sys.purgeable\_assets**

本参数定义了系统是否可以清除暂时不用的数据以释放更多的RAM。可填写的值为0或1。当值为1时，系统会定期清理不用的数据以释放更多的RAM，同时作为代价就是下次启动程序或游戏加载数据会变慢。

**video.accelerate.hw**

本参数定义了系统是否对视频启用**硬件加速**功能。这里的视频指代屏幕上显示的东西，不仅仅是“电影视频”。可填写的值为0或1。需要注意的是：摩托官方的2.2和2.3系统对此功能支持的不是很好，开启后有时反而会降低系统流畅度。但CM系统绝对建议开启。

**debug.performance.tuning**

本参数定义了系统是否针对性能做较调。可填写的值为0或1。需要注意的是：摩托官方的2.2和2.3系统对此功能支持的不是很好，开启后有时反而会降低系统流畅度。但CM系统绝对建议开启。

ro.HOME\_APP\_ADJ

ro.FOREGROUND\_APP\_ADJ

ro.VISIBLE\_APP\_ADJ

ro.PERCEPTIBLE\_APP\_ADJ

ro.HEAVY\_WEIGHT\_APP\_ADJ

ro.SECONDARY\_SERVER\_ADJ

ro.BACKUP\_APP\_ADJ

ro.HIDDEN\_APP\_MIN\_ADJ

ro.EMPTY\_APP\_ADJ

以上参数可填写的值为整数。这里只给出值的规律

0代表降低进程的优先级且驻留内存，1代表驻留内存，4代表缓存较多的内存，15代表尽量缓存内存。也就是说内存缓存器是按照ADJ从大到小来进行缓存的。大家可根据自系统中自己对各种应用程序的要求进行更改。

以下给出一个经典用例：

ro.FOREGROUND\_APP\_ADJ=0 前台程序驻留内存(不缓存)

ro.VISIBLE\_APP\_ADJ=1         可见的程序驻留内存(不缓存)

ro.PERCEPTIBLE\_APP\_ADJ=2    缓存的RAM多一些

ro.HOME\_APP\_ADJ=3         桌面程序，缓存的RAM稍多一些

ro.HEAVY\_WEIGHT\_APP\_ADJ=4        缓存的RAM再多一些

ro.SECONDARY\_SERVER\_ADJ=5        缓存的RAM再再多一些

ro.BACKUP\_APP\_ADJ=6       缓存的RAM再再再多一些

ro.HIDDEN\_APP\_MIN\_ADJ=7         隐藏的程序，根据程序的类型进行内存管理，最低为缓存的RAM再再再再多一些，最高就是直接缓存内存。

ro.EMPTY\_APP\_ADJ=15         已经退出的程序，直接缓存内存

ro.FOREGROUND\_APP\_MEM

ro.VISIBLE\_APP\_MEM

ro.PERCEPTIBLE\_APP\_MEM

ro.HEAVY\_WEIGHT\_APP\_MEM

ro.SECONDARY\_SERVER\_MEM

ro.BACKUP\_APP\_MEM

ro.HOME\_APP\_MEM

ro.HIDDEN\_APP\_MEM

ro.CONTENT\_PROVIDER\_MEM

ro.EMPTY\_APP\_MEM

以上参数定义了各种类型的应用程序在内存缓冲的大小，单位是页下面给出一个经典用例：

ro.FOREGROUND\_APP\_MEM=1280

ro.VISIBLE\_APP\_MEM=2560

ro.PERCEPTIBLE\_APP\_MEM=3840

ro.HEAVY\_WEIGHT\_APP\_MEM=6400

ro.SECONDARY\_SERVER\_MEM=7680

ro.BACKUP\_APP\_MEM=8960

ro.HOME\_APP\_MEM=5120

ro.HIDDEN\_APP\_MEM=12800

ro.CONTENT\_PROVIDER\_MEM=15360

ro.EMPTY\_APP\_MEM=20480

### 基本耗电相关参数

**wifi.supplicant\_scan\_interval**

本参数定义了Wifi扫描已保存节电的时间间隔。当点亮屏幕或打开Wifi时，系统会不停的扫描环境中是否存在已经保存的Wifi节点，当发现后则进行连接，而这个参数控制了每次扫描的时间间隔。单位是秒。取值范围是正整数。**官方默认为45，推荐180**。

**ro.mot.battmanager.wifictrl**

本参数定义了电源管理模块对Wifi的控制。默认为0。当此值为1时可以明显节电，但有时Wifi会出现不稳定的情况(不是所有ROM都如此)。

**ro.mot.deep.sleep.supported**

本参数定义了是否开启摩托的“休眠”模式。取值为true或false。当值为true时，在电源菜单中会出现“休眠”模式。此模式类似于电脑的睡眠，即将CPU等部件的电源全部关闭，只为**RAM供电**以保存休眠前的系统状态。耗电量比完全关机多一些，但可以做到**瞬间开机**。仅在官方ROM有效。

**pm.sleep\_mode**

本参数定义了系统待机时的睡眠深度，在所有Android系统上有效。取值范围是0~4，对应解释如下。0：强制关闭除RAM之外的所有部件，此状态下最省电。Defy几乎可以纯待机3~4个礼拜。但是此模式与“休眠”类似，一旦进入之后射频也会关闭，手机的2G/3G信号也就断了(语音和数据)。

1：让ARM进入中断触发的待机(超低功耗)模式。与模式0相比，本模式下射频不会关闭，而ARM可以通过软件(闹铃)和硬件(来电)中断来唤醒，因此耗电方面远大于模式0，Defy可以纯待机7天(不安装任何软件)。非常建议使用。  
2：将所有应用程序挂起到后台。与模式1相比，本模式下硬件几乎不参与多少节电，耗电自然比模式1多很多。当应用程序被挂起后，CPU的负载会大幅度降低，从而节电。此模式下Defy纯待机5天。  
3：将CPU的频率和电压降至最低，低到主频只有几十MHz的水平，而此时CPU接受外部中断(通过中断来恢复频率和电压)。与模式2相比，本模式下CPU通过降频和降压参与了节电，因此本模式的耗电比模式2多了一点。Defy纯待机约4~5天。本模式也是官方ROM和官方CM系统的默认值。  
4：CPU接受外部中断。与上述4个模式相比，此模式下几乎不做任何节电，只是关闭了屏幕和按键背光而已。Defy纯待机约2天。将上述5个模式的节电按照星级来分就是，模式0和1为5颗星，模式2和3为3颗星，模式4为1颗星。综上所属就是，模式0和模式1基本一样，是靠完全关闭几乎所有硬件部件来进行节电，省电效果最佳。模式2和模式3是靠调节CPU频率来进行节电。个人强烈推荐采用pm.sleep\_mode=1，即省电又稳定。如果想用模式0但又担心基带射频的同学可以继续往下看，解决办法在下面。

**ro.ril.disable.power.collapse**

本参数定义了是否禁止射频参与电源休眠。取值是0或1。这个参数的使用需要与上一个参数相匹配(我看到很多ROM中的这两个参数都是不匹配的，**最终造成的效果就是点亮屏幕后信号存在问题**)。当本参数为1的时候即射频永远打开，为0的时候根据上一个参数pm.sleep\_mode来判断是否关闭射频。永远打开射频必然费电，但是如果射频关闭，那手机就没信号了。那么当pm.sleep\_mode=0的时候，上面说过，此时待机会关闭几乎所有硬件部件，包括射频。而此时如果ro.ril.disable.power.collapse=1，就会保持射频的开启(即使进入休眠模式也一样)。这样即使待机，手机也有信号。但是又存在这样一个现象，在有些ROM中pm.sleep\_mode=0会带来更多的问题， **如睡死、亮屏后Wifi打不开、蓝牙打不开等**。因此建议同学们可以先尝试一下pm.sleep\_mode=0和ro.ril.disable.power.collapse=1组合使用，看看是否有bug，如果没有那自然使用此种模式，毕竟最省电了(极端省电)。对于稳定与省电兼得，可使用如下组合：pm.sleep\_mode=1ro.ril.disable.power.collapse=0这样射频在pm.sleep\_mode=1下不会被关闭，而进入休眠模式后射频会关闭。

### 参数定制建议

|  |  |  |  |
| --- | --- | --- | --- |
|  | 原生定义 | 现状 | 修改意见 |
| 系统设置-》关于设备显示-》  产品名 | **ro.product.model：**The end-user-visible name for the end product. | 硬编码或者，没有 | 类似公司华为手机，实现属性ro.product.model |
| 打印在产品硬件印刷名：同上 | 同上 | 直接是ma；GL300A | 同上 |
| 上传到服务器产品名字：同上 | 同上 | 同上 | 同上 |
| 对外宣传的名字 | **无** |  | * 旧产品保留，新产品改为，加一个ro. \*\*.product. [**media**](http://dict.cn/media)   在系统设置-》关于设备显示-》增加一个显示 |
| App与系统之间代码标志名字 | **ro.product.device ：***The name of the industrial design.* | 使用**ro.product.name** | 旧产品保留，新产品改为原生定义**ro.product.device** |
|  |  |  |  |

ro.build.product，使用代码ro.product.device

/\*\* \*/

public static final String MODEL = getString("ro.product.model");

*/\*\* The name of the overall product. \*/***public static final** String ***PRODUCT*** = *getString*(**"ro.product.name"**);  
  
*/\*\* The name of the industrial design. \*/***public static final** String ***DEVICE*** = *getString*(**"ro.product.device"**);  
  
*/\*\* The name of the underlying board, like "goldfish". \*/***public static final** String ***BOARD*** = *getString*(**"ro.product.board"**);

## 源码分析

### frameworks

**JAVA**：\frameworks\base\core\[**Java**](http://lib.csdn.net/base/java)\[**android**](http://lib.csdn.net/base/android)\os\ SystemProperties.java：get()—》native\_get

**JIN**:\frameworks\base\core\jni\android\_os\_SystemProperties.cpp: SystemProperties\_getSS-> property\_get

### **bionic：设置属性**异步socket通信

bionic是一种 [C 标准库](https://zh.wikipedia.org/wiki/C%E6%A8%99%E6%BA%96%E5%87%BD%E5%BC%8F%E5%BA%AB)，由 [Google](https://zh.wikipedia.org/wiki/Google) 所开发的[自由软件](https://zh.wikipedia.org/wiki/%E8%87%AA%E7%94%B1%E8%BB%9F%E9%AB%94)，用于 [Android](https://zh.wikipedia.org/wiki/Android) [嵌入式](https://zh.wikipedia.org/wiki/%E5%B5%8C%E5%85%A5%E5%BC%8F)系统上，采用 [BSD 授权条款](https://zh.wikipedia.org/wiki/BSD%E6%8E%88%E6%AC%8A%E6%A2%9D%E6%AC%BE)。Google 希望用它来取代 [glibc](https://zh.wikipedia.org/wiki/Glibc)，它的发展目标是达到轻量化以及高运行速度。

**操作在\bionic\libc\bionic\system\_properties.c中：**

|  |
| --- |
| int \_\_system\_property\_get(const char \*name, char \*value)  {    //数据已经存储在内存中\_\_system\_property\_area\_\_ 等待读取完返回  const prop\_info \*pi = \_\_system\_property\_find(name);  return \_\_system\_property\_read(pi, 0, value);  } |

进程启动后数据已经将系统属性数据读取到相应的**共享内存**中，保存在全局变量\_\_system\_property\_area\_\_，在

|  |
| --- |
| int \_\_system\_property\_set(const char \*key, const char \*value)  {     msg.cmd = PROP\_MSG\_SETPROP;     strlcpy(msg.name, key, sizeof msg.name);     strlcpy(msg.value, value, sizeof msg.value);     err = send\_prop\_msg(&msg);  }  static int send\_prop\_msg(prop\_msg \*msg)  {    //sokcet 通信 /dev/socket/property\_service     s = socket(AF\_LOCAL, SOCK\_STREAM, 0);     connect(s, (struct sockaddr \*) &addr, alen)     send(s, msg, sizeof(prop\_msg), 0)     close(s);  } |

通过socket向property\_service发送消息，property\_service运行在哪里呢？

### \system\core\init\：

#### Property Service创建服务端socket

root@gl300e:/dev/socket # ls -al | grep property\_service

srw-rw-rw- root root 2013-01-21 17:47 property\_service

**init进程启动监听过程中：**，**Property Service 是运行在init守护进程中。**

|  |
| --- |
| \system\core\init\Init.c  int main(int argc, char \*\*argv)  {     //加入到action queue队列  queue\_builtin\_action(property\_service\_init\_action, **"property\_service\_init");**   for(;;)  //执行action queue队列  //接收通过socket向property service 发送的数据;         nr = poll(ufds, fd\_count, timeout);         ……         handle\_property\_set\_fd();  }  static int **property\_service\_init\_action**(int nargs, char \*\*args)  {     start\_property\_service();  }  \system\core\init\property\_service.c：  void **start\_property\_service(void)**  {    //加载属性配置文件      load\_properties\_from\_file(PROP\_PATH\_SYSTEM\_BUILD);     load\_properties\_from\_file(PROP\_PATH\_SYSTEM\_DEFAULT);     load\_properties\_from\_file(PROP\_PATH\_LOCAL\_OVERRIDE);     load\_persistent\_properties();     //创建socket资源 并绑定     fd **= create\_socket(PROP\_SERVICE\_NAME, SOCK\_STREAM**, 0666, 0, 0);    //监听     listen(fd, 8);  } |



#define PROP\_PATH\_SYSTEM\_BUILD     "/system/build.prop"  
#define PROP\_PATH\_SYSTEM\_DEFAULT   "/system/default.prop"  
#define PROP\_PATH\_LOCAL\_OVERRIDE   "/data/local.prop"

Persist 顾名思义是从持久化文件读写的

#### Property Service 监听socket处理

|  |
| --- |
| //\system\core\init\property\_service.c：  void handle\_property\_set\_fd()  {    //等待建立通信     s = accept(property\_set\_fd, (struct sockaddr \*) &addr, &addr\_size)    //获取套接字相关信息 uid gid     getsockopt(s, SOL\_SOCKET, SO\_PEERCRED, &cr, &cr\_size);    //接收属性设置请求消息     recv(s, &msg, sizeof(msg), 0);    //处理消息     switch(msg.cmd) {    case PROP\_MSG\_SETPROP:        //通过设置系统属性  处理ctl.开头消息         if(memcmp(msg.name,"ctl.",4) == 0)         {            //权限检测             if (check\_control\_perms(msg.value, cr.uid, cr.gid))             {                 handle\_control\_message((char\*) msg.name + 4, (char\*) msg.value);             }         } else         {            //更改系统属性值             if (check\_perms(msg.name, cr.uid, cr.gid))             {                 property\_set((char\*) msg.name, (char\*) msg.value);             }         }    break;     }     close(s);  } |

#### 设置属性（根据key类型）

|  |
| --- |
| **int** property\_set(**const char** \*name, **const char** \*value)  {  prop\_info \*pi;  **int** ret;   size\_t namelen = strlen(name);  size\_t valuelen = strlen(value);   **if** (!is\_legal\_property\_name(name, namelen)) **return** -1;   **if (valuelen >= PROP\_VALUE\_MAX) return -1;**   pi = (prop\_info\*) \_\_system\_property\_find(name);   **if**(pi != 0) {  */\* ro.\* properties may NEVER be modified once set \*/* **if**(!strncmp(name, **"ro."**, 3)) **return** -1;   \_\_system\_property\_update(pi, value, valuelen);  } **else** {  ret = \_\_system\_property\_add(name, namelen, value, valuelen);  **if** (ret < 0) {  ERROR(**"Failed to set '%s'='%s'\n"**, name, value);  **return** ret;  }  }  */\* If name starts with "net." treat as a DNS property. \*/* **if** (strncmp(**"net."**, name, strlen(**"net."**)) == 0) {  **if** (strcmp(**"net.change"**, name) == 0) {  **return** 0;  }  */\*  \* The 'net.change' property is a special property used track when any  \* 'net.\*' property name is updated. It is \_ONLY\_ updated here. Its value  \* contains the last updated 'net.\*' property.  \*/* property\_set(**"net.change"**, name);  } **else if** (persistent\_properties\_loaded &&  strncmp(**"persist."**, name, strlen(**"persist."**)) == 0) {  */\*  \* Don't write properties to disk until after we have read all default properties  \* to prevent them from being overwritten by default values.  \*/* write\_persistent\_property(name, value);  } **else if** (strcmp(**"selinux.reload\_policy"**, name) == 0 &&  strcmp(**"1"**, value) == 0) {  selinux\_reload\_policy();  }  property\_changed(name, value);  **return** 0;  }   **void** handle\_property\_set\_fd()  {  prop\_msg msg;  **int** s;  **int** r;  **int** res;  struct ucred cr;  struct sockaddr\_un addr;  socklen\_t addr\_size = sizeof(addr);  socklen\_t cr\_size = sizeof(cr);  **char** \* source\_ctx = NULL;  struct pollfd ufds[1];  **const int** timeout\_ms = 2 \* 1000; */\* Default 2 sec timeout for caller to send property. \*/* **int** nr;   **if** ((s = accept(property\_set\_fd, (struct sockaddr \*) &addr, &addr\_size)) < 0) {  **return**;  }   */\* Check socket options here \*/* **if** (getsockopt(s, SOL\_SOCKET, SO\_PEERCRED, &cr, &cr\_size) < 0) {  close(s);  ERROR(**"Unable to receive socket options\n"**);  **return**;  }   ufds[0].fd = s;  ufds[0].events = POLLIN;  ufds[0].revents = 0;  nr = TEMP\_FAILURE\_RETRY(poll(ufds, 1, timeout\_ms));  **if** (nr == 0) {  ERROR(**"sys\_prop: timeout waiting for uid=%d to send property message.\n"**, cr.uid);  close(s);  **return**;  } **else if** (nr < 0) {  ERROR(**"sys\_prop: error waiting for uid=%d to send property message. err=%d %s\n"**, cr.uid, errno, strerror(errno));  close(s);  **return**;  }   r = TEMP\_FAILURE\_RETRY(recv(s, &msg, sizeof(msg), MSG\_DONTWAIT));  **if**(r != sizeof(prop\_msg)) {  ERROR(**"sys\_prop: mis-match msg size received: %d expected: %zu errno: %d\n"**,  r, sizeof(prop\_msg), errno);  close(s);  **return**;  }   **switch**(msg.cmd) {  **case** PROP\_MSG\_SETPROP:  msg.name[PROP\_NAME\_MAX-1] = 0;  msg.value[PROP\_VALUE\_MAX-1] = 0;   **if** (!is\_legal\_property\_name(msg.name, strlen(msg.name))) {  ERROR(**"sys\_prop: illegal property name. Got: \"%s\"\n"**, msg.name);  close(s);  **return**;  }   getpeercon(s, &source\_ctx);   **if**(memcmp(msg.name,**"ctl."**,4) == 0) {  *// Keep the old close-socket-early behavior when handling  // ctl.\* properties.* close(s);  **if** (check\_control\_mac\_perms(msg.value, source\_ctx)) {  handle\_control\_message((**char**\*) msg.name + 4, (**char**\*) msg.value);  } **else** {  ERROR(**"sys\_prop: Unable to %s service ctl [%s] uid:%d gid:%d pid:%d\n"**,  msg.name + 4, msg.value, cr.uid, cr.gid, cr.pid);  }  } **else** {  **if** (check\_perms(msg.name, source\_ctx)) {  property\_set((**char**\*) msg.name, (**char**\*) msg.value);  } **else** {  ERROR(**"sys\_prop: permission denied uid:%d name:%s\n"**,  cr.uid, msg.name);  }   *// Note: bionic's property client code assumes that the  // property server will not close the socket until \*AFTER\*  // the property is written to memory.* close(s);  }  freecon(source\_ctx);  **break**;  **default**:  close(s);  **break**;  }  } |

**修改系统属性权限表：**

property\_perms[] = {

   { "net.dns",          AID\_RADIO,    0 },

   { "net.",             AID\_SYSTEM,   0 },

   { "dev.",             AID\_SYSTEM,   0 },

   { "runtime.",         AID\_SYSTEM,   0 },

   { "sys.",             AID\_SYSTEM,   0 },

   { "service.",         AID\_SYSTEM,   0 },

   { "persist.sys.",     AID\_SYSTEM,   0 },

   { "persist.service.", AID\_SYSTEM,   0 },

   ……

    { NULL, 0, 0 }

};

一般property启动应该加在init.<your hardware>.rc而不是直接init.rc里。下面是一个init.rc里的例子：

### 开关服务

可以通过设置系统属性 改变服务的执行状态 start/stop：如果想要应用有权限启动/关闭某Native Service：需要具有system/root权限，找到对应应用uid gid，将应用名称加入到control\_perms列表中。

连着前面就是ctr.start和ctr.stop系统属性：用来启动和停止服务的。

|  |
| --- |
| void handle\_control\_message(const char \*msg, const char \*arg)  {    if (!strcmp(msg,"start")) {         msg\_start(arg);     } else if (!strcmp(msg,"stop")) {         msg\_stop(arg);     } else if (!strcmp(msg,"restart")) {         msg\_stop(arg);         msg\_start(arg);     }  }static void msg\_start(const char \*name)  {     service\_start(svc, args);  }void service\_start(struct service \*svc, const char \*dynamic\_args){    //创建进程启动服务     pid = fork();     execve(svc->args[0], (char\*\*) svc->args, (char\*\*) ENV);     //修改服务的系统属性 执行状态     notify\_service\_state(svc->name, "running");  } |

例如：　// start boot animation

# setprop ctl.start bootanim

　　　　property\_set("ctl.start", "bootanim");

定义开机启动的服务:**启动服务的时候会判断：**

static void service\_start\_if\_not\_disabled(struct service \*svc)

{        //判断是否启动

       if (!(svc->flags & SVC\_DISABLED)) {

           service\_start(svc, NULL);

       }

}

在init.rc中表明服务是否在开机时启动：

service adbd /sbin/adbd

　　　　class core

　　　　disabled //不自动启动

### 开机动画情景分析

Android系统的开机动画是由应用程序bootanimation来实现的，它位于/system/bin目录下，bootanimation在init.rc中的定义如下：

1. service bootanim /system/bin/bootanimation
2. class core
3. user graphics
4. group graphics audio
5. disabled
6. oneshot

可见，由于设置为"disable"，该应用在init启动过程中是不会启动的，需要其他地方显示的调用才能启动。那是什么时候启动的呢？当SurfaceFlinger服务启动时，会修改系统属性值ctl.start，通知init进程启动bootanimation。

SurfaceFlinger调用了startBootAnim()启动了开机动画。该函数代码如下：

1. void SurfaceFlinger::startBootAnim() {
2. // start boot animation
3. property\_set("service.bootanim.exit", "0");
4. property\_set("ctl.start", "bootanim");
5. }

可见，将系统属性ctl.start的值设置为"bootanim"。

回到init进程的init.c的main函数中：

1. **int** main(**int** argc, **char** \*\*argv) {
2. **for**(;;) {
3. nr = poll(ufds, fd\_count, timeout);
4. **if** (nr <= 0)
5. **continue**;
6. **for** (i = 0; i < fd\_count; i++) {
7. **if** (ufds[i].revents & POLLIN) {
8. **if** (ufds[i].fd == get\_property\_set\_fd())
9. handle\_property\_set\_fd();
10. **else** **if** (ufds[i].fd == get\_keychord\_fd())
11. handle\_keychord();
12. **else** **if** (ufds[i].fd == get\_signal\_fd())
13. handle\_signal();
14. }
15. }
16. }
17. }

可以看到，init进程会使用poll机制来轮询事件，其中一个事件是系统属性值被修改。得到该事件后，会执行handle\_property\_set\_fd()，代码如下：

1. if(memcmp(msg.name,"ctl.",4) == 0) {
2. // Keep the old close-socket-early behavior when handling
3. // ctl.\* properties.
4. close(s);
5. if (check\_control\_mac\_perms(msg.value, source\_ctx)) {
6. handle\_control\_message((char\*) msg.name + 4, (char\*) msg.value);
7. } else {
8. ERROR("sys\_prop: Unable to %s service ctl [%s] uid:%d gid:%d pid:%d\n",
9. msg.name + 4, msg.value, cr.uid, cr.gid, cr.pid);
10. }
11. }

该函数会进一步执行handle\_control\_message()，传入的参数msg.name=ctl.start，msg.value=bootanim。

1. void handle\_control\_message(const char \*msg, const char \*arg)
2. {
3. if (!strcmp(msg,"start")) {
4. msg\_start(arg);
5. } else if (!strcmp(msg,"stop")) {
6. msg\_stop(arg);
7. } else if (!strcmp(msg,"restart")) {
8. msg\_restart(arg);
9. } else {
10. ERROR("unknown control msg '%s'\n", msg);
11. }
12. }

由于msg == "start"，handle\_control\_message进一步执行msg\_start（），且传入的arg参数等于bootanim。msg\_start代码如下：

1. static void msg\_start(const char \*name)
2. {
3. struct service \*svc = NULL;
4. char \*tmp = NULL;
5. char \*args = NULL;
7. if (!strchr(name, ':'))
8. svc = service\_find\_by\_name(name);
9. else {
10. tmp = strdup(name);
11. if (tmp) {
12. args = strchr(tmp, ':');
13. \*args = '\0';
14. args++;
16. svc = service\_find\_by\_name(tmp);
17. }
18. }
20. if (svc) {
21. service\_start(svc, args);
22. } else {
23. ERROR("no such service '%s'\n", name);
24. }
25. if (tmp)
26. free(tmp);
27. }

该函数首先调用service\_find\_by\_name()，从service\_list中查询要启动的服务是否有存在，若存在，返回服务的相关信息。因为init.rc中有bootanimation的定义，因此在init进程执行parse\_config()时，会将该服务添加到service\_list中，所以bootanimation应用是存在的。然后，如果找到了该服务，就调用service\_start启动服务。

到此，bootanimation应用就启动了。

### 共享属性原理

初始化共享内存：属性是存储在共享内存中的,而要在使用共享内存之前呢,又必须要先初始化共享内存.初始化之后,肯定要加载那些存储属性的文件等等.这些都是init进程中完成的.

init.c 的main函数中调用了property\_init函数

init. property\_init（）

init\_property\_area

|  |  |
| --- | --- |
| 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | static int init\_property\_area(void) {  if (property\_area\_inited)  return -1;   if(\_\_system\_property\_area\_init())//会以读写方式打开/dev/\_\_properties\_\_  return -1;   if(init\_workspace(&pa\_workspace, 0))//这里面会以只读方式再次打开/dev/\_\_properties\_\_  return -1;   fcntl(pa\_workspace.fd, F\_SETFD, FD\_CLOEXEC);   property\_area\_inited = 1;//表明共享内存已经被初始化了  return 0; } |

system\_property\_area\_init用来初始化共享内存.

|  |  |
| --- | --- |
| 1 2 3 4 | int \_\_system\_property\_area\_init() {  return map\_prop\_area\_rw(); } |

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 | static int map\_prop\_area\_rw() {  /\* dev is a tmpfs that we can use to carve a shared workspace  \* out of, so let's do that...  \*/  //proerty\_file是 /dev/\_\_properties\_\_,要注意,这里是以可读可写方式打开的,这个文件描述符是给property service使用的  const int fd = open(property\_filename,  O\_RDWR | O\_CREAT | O\_NOFOLLOW | O\_CLOEXEC | O\_EXCL, 0444);   if (fd < 0) {  if (errno == EACCES) {  /\* for consistency with the case where the process has already  \* mapped the page in and segfaults when trying to write to it  \*/  abort();  }  return -1;  }   // TODO: Is this really required ? Does android run on any kernels that  // don't support O\_CLOEXEC ?  const int ret = fcntl(fd, F\_SETFD, FD\_CLOEXEC);  if (ret < 0) {  close(fd);  return -1;  }   if (ftruncate(fd, PA\_SIZE) < 0) {  close(fd);  return -1;  }   pa\_size = PA\_SIZE;  pa\_data\_size = pa\_size - sizeof(prop\_area);  compat\_mode = false;   //----------------pa\_size大小为128K  void \*const memory\_area = mmap(NULL, pa\_size, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);  if (memory\_area == MAP\_FAILED) {  close(fd);  return -1;  }   prop\_area \*pa = new(memory\_area) prop\_area(PROP\_AREA\_MAGIC, PROP\_AREA\_VERSION);   /\* plug into the lib property services \*/  \_\_system\_property\_area\_\_ = pa;//共享内存的起始地址存储在这个全局变量上   close(fd);  return 0; } |

我们可以看到，在init进程的main()函数里，打开了一个设备文件“/dev/\_\_**properties**”，并把它设定为128KB大小，接着调用mmap()将这块内存映射到init进程空间了。这个内存的首地址被记录在**system\_property\_area**全局变量里，以后每添加或修改一个属性，都会基于这个**system\_property\_area**变量来计算位置。

在来看下面的函数:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | static int init\_workspace(workspace \*w, size\_t size)//size传入的参数值是0 {  void \*data;  int fd = open(PROP\_FILENAME, O\_RDONLY | O\_NOFOLLOW);//以只读的方式再次打开/dev/\_\_properties\_\_,  if (fd < 0)  return -1;   w->size = size;  w->fd = fd;  return 0; } |

打开的句柄记录在pa\_workspace.fd处，以后每当init进程调用service\_start()时，会执行下面的代码

|  |  |
| --- | --- |
| 1 2 3 4 5 | if (properties\_inited()) {  get\_property\_workspace(&fd, &sz);  sprintf(tmp, "%d,%d", dup(fd), sz);  add\_environment("ANDROID\_PROPERTY\_WORKSPACE", tmp);  } |

说白了就是把 pa\_workspace.fd 的句柄记入一个名叫“ ANDROID\_PROPERTY\_WORKSPACE ”的环境变量去,另外size似乎没什么用,一直是0.

|  |  |
| --- | --- |
| 1 2 | root@generic\_x86\_64:/ # echo $ANDROID\_PROPERTY\_WORKSPACE  8,0 |

存入环境变量的作用是其他进程可以很方便拿到文件描述符fd,利用这个fd就可以读取属性值了.

为什么要两次open那个/dev/**properties**文件呢？是这样的：第一次open的句柄，最终是给属性服务自己用的，所以需要有读写权限；而第二次open的句柄，会被记入pa\_workspace.fd，并在合适时机添加进环境变量，**供其他进程使用，因此只能具有读取权限。**

## 小结

Property Service运行在init进程中，开机从属性文件中加载到共享内存中；设置系统属性通过socket与Property Service通信。

　　Property Consumer进程将存储系统属性值的共享内存，加载到当前进程虚拟空间中，实现对系统属性值的读取。

Property Setter进程修改系统属性，通过socket向Property Service发送消息，更改系统属性值。

属性系统设计的关键就是：跨进程共享内存的实现。

[详情看](http://blog.csdn.net/hecong_kit/article/details/46008895)

属性初始化的入口点是property\_init ，在system/core/init/property\_service.c中定义。它的主要工作是申请32k共享内存，其中前1k是属性区的头，后面31k可以存247个属性（受前1k头的限制）。property\_init初始化完property以后，加载/default.prop的属性定义。   
  
其它的系统属性（build.prop, local.prop,…）在start\_property\_service中加载。加载完属性服务创建一个socket和其他进程通信（设置或读取属性）。   
  
Init进程poll属性的socket，等待和处理属性请求。如果有请求到来，则调用handle\_property\_set\_fd来处理这个请求。在这个函数里，首先检查请求者的uid/gid看看是否有权限，如果有权限则调property\_service.c中的property\_set函数。   
  
在property\_set函数中，它先查找就没有这个属性，如果找到，更改属性。如果找不到，则添加新属性。更改时还会判断是不是“ro”属性，如果是，则不能更改。如果是persist的话还会写到/data/property/<name>中。   
  
**最后它会调property\_changed，把事件挂到队列里**，如果有人注册这个属性的话（比如init.rc中on property:ro.kernel.qemu=1），最终会调它的会调函数。   
  
property名字长度限制是32字节，值的限制是92字节。不知道是google怎么想的 — 一般都是名字比值长得多！比如[dalvik.vm.heapgrowthlimit]: [48m]

## Ref

[Android 5.1 property属性系统分析上篇](http://www.iloveandroid.net/2015/09/26/Android_property_1/)

## REF

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