Android之SystemServer启动(上)

发表于 2016-11-28   |

SystemServer是Android Framework的核心，大部分的Android系统核心服务都运行在SystemServer进程当中。此篇博客主要分析SystemServer的进程创建流程，下篇分析SystemServer的初始化流程，相关的源代码在以下文件中：

* frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
* frameworks/base/core/java/com/android/internal/os/Zygote.java
* frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java
* frameworks/base/core/jni/com\_android\_internal\_os\_Zygote.cpp
* frameworks/base/core/java/com/android/internal/os/RuntimeInit.java
* frameworks/base/core/jni/AndroidRuntime.cpp
* frameworks/base/cmds/app\_process/app\_main.cpp

**启动流程**

SystemServer(system\_server进程)是通过Zygote fork生成的，如下init.zygote32.rc/init.zygote64\_32.rc中的定义，参数--start-system-server指定了Zygote启动SystemServer。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | service zygote /system/bin/app\_process64 -Xzygote /system/bin --zygote --start-system-server --socket-name=zygote  class main  socket zygote stream 660 root system  onrestart write /sys/android\_power/request\_state wake  onrestart write /sys/power/state on  onrestart restart audioserver  onrestart restart cameraserver  onrestart restart media  onrestart restart netd  writepid /dev/cpuset/foreground/tasks /sys/fs/cgroup/stune/foreground/tasks  service zygote\_secondary /system/bin/app\_process32 -Xzygote /system/bin --zygote --socket-name=zygote\_secondary  class main  socket zygote\_secondary stream 660 root system  onrestart restart zygote  writepid /dev/cpuset/foreground/tasks /dev/stune/foreground/tasks   USER PID PPID VSIZE RSS WCHAN PC NAME root 1 0 16980 2636 SyS\_epoll\_ 0000500ab0 S /init root 929 1 2166716 82876 poll\_sched 7f8efff61c S zygote64 system 1695 929 2488940 152772 SyS\_epoll\_ 7f8efff4fc S system\_server |

SystemServer的启动流程大概可以分为二个阶段，第一阶段Zygote fork system\_server进程，第二阶段执行SystemServer类，进行系统服务的启动及初始化，此篇博客分析SystemServer进程创建过程，下篇分析系统服务启动初始化过程。Zygote fork SystemServer的流程如下图示：

**ZygoteInit.main()**

ZygoteInit.main()通过解析argv[]参数列表来确定是否启动SystemServer，如果有定义参数--start-system-server则启动调用startSystemServer启动SystemServer

|  |  |
| --- | --- |
| 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 | <<< frameworks/base/core/java/com/android/internal/os/ZygoteInit.java  public static void main(String argv[]) {  try {  boolean startSystemServer = false;  ......  for (int i = 1; i < argv.length; i++) {  if ("start-system-server".equals(argv[i])) {  startSystemServer = true;  }  ......  }  ......  if (startSystemServer) {  startSystemServer(abiList, socketName);  }  ......  } catch (MethodAndArgsCaller caller) {  caller.run();  } catch (RuntimeException ex) {  Log.e(TAG, "Zygote died with exception", ex);  closeServerSocket();  throw ex;  } } |

**ZygoteInit.startSystemServer()**

|  |  |
| --- | --- |
| 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 52 53 54 55 56 57 58 59 60 61 62 63 64 | <<< frameworks/base/core/java/com/android/internal/os/ZygoteInit.java  private static boolean startSystemServer(String abiList, String socketName)  throws MethodAndArgsCaller, RuntimeException {  long capabilities = posixCapabilitiesAsBits(  OsConstants.CAP\_IPC\_LOCK,  OsConstants.CAP\_KILL,  OsConstants.CAP\_NET\_ADMIN,  OsConstants.CAP\_NET\_BIND\_SERVICE,  OsConstants.CAP\_NET\_BROADCAST,  OsConstants.CAP\_NET\_RAW,  OsConstants.CAP\_SYS\_MODULE,  OsConstants.CAP\_SYS\_NICE,  OsConstants.CAP\_SYS\_RESOURCE,  OsConstants.CAP\_SYS\_TIME,  OsConstants.CAP\_SYS\_TTY\_CONFIG  );  /\* Containers run without this capability, so avoid setting it in that case \*/  if (!SystemProperties.getBoolean(PROPERTY\_RUNNING\_IN\_CONTAINER, false)) {  capabilities |= posixCapabilitiesAsBits(OsConstants.CAP\_BLOCK\_SUSPEND);  }  /\* Hardcoded command line to start the system server \*/  String args[] = {  "--setuid=1000",  "--setgid=1000",  "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,1021,1032,3001,3002,3003,3006,3007,3009,3010",  "--capabilities=" + capabilities + "," + capabilities,  "--nice-name=system\_server",  "--runtime-args",  "com.android.server.SystemServer",  };  ZygoteConnection.Arguments parsedArgs = null;   int pid;   try {  // 准备执行参数  parsedArgs = new ZygoteConnection.Arguments(args);  ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);  ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);   // fork SystemServer并返回进程id  pid = Zygote.forkSystemServer(  parsedArgs.uid, parsedArgs.gid,  parsedArgs.gids,  parsedArgs.debugFlags,  null,  parsedArgs.permittedCapabilities,  parsedArgs.effectiveCapabilities);  } catch (IllegalArgumentException ex) {  throw new RuntimeException(ex);  }   // 子进程即system\_server进程中执行handleSystemServerProcess  if (pid == 0) {  if (hasSecondZygote(abiList)) {  waitForSecondaryZygote(socketName);  }   handleSystemServerProcess(parsedArgs);  }   return true; } |

startSystemServer中，首先为SystemServer准备启动参数，指定uid，gid为1000，进程名为“system\_server”，并指定执行类为“com.android.server.SystemServer”。然后执行Zygote.forkSystemServer() fork出进程，Zygote fork完成后将在子进程system\_server中继续执行handleSystemServerProcess()。

**Zygote.forkSystemServer()**

forkSystemServer()继续调用native方法nativeForkSystemServer()

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | <<< frameworks/base/core/java/com/android/internal/os/Zygote.java  public static int forkSystemServer(int uid, int gid, int[] gids, int debugFlags,  int[][] rlimits, long permittedCapabilities, long effectiveCapabilities) {  VM\_HOOKS.preFork();  int pid = nativeForkSystemServer(  uid, gid, gids, debugFlags, rlimits, permittedCapabilities, effectiveCapabilities);  // Enable tracing as soon as we enter the system\_server.  if (pid == 0) {  Trace.setTracingEnabled(true);  }  VM\_HOOKS.postForkCommon();  return pid; } |

与fork普通应用进程实现一样，nativeForkSystemServer在frameworks/base/core/jni/com\_android\_internal\_os\_Zygote.cpp中实现，映射关联到com\_android\_internal\_os\_Zygote\_nativeForkSystemServer()

**nativeForkSystemServer()**

|  |  |
| --- | --- |
| 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 | <<< frameworks/base/core/jni/com\_android\_internal\_os\_Zygote.cpp  { "nativeForkSystemServer", "(II[II[[IJJ)I",  (void \*) com\_android\_internal\_os\_Zygote\_nativeForkSystemServer },  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) {  // 与fork普通应用进程一样调用ForkAndSpecializeCommon创建出子进程  pid\_t pid = ForkAndSpecializeCommon(env, uid, gid, gids,  debug\_flags, rlimits,  permittedCapabilities, effectiveCapabilities,  MOUNT\_EXTERNAL\_DEFAULT, NULL, NULL, true, NULL,  NULL, NULL);  if (pid > 0) {  // 保存system\_server进程pid  gSystemServerPid = pid;   int status;  // 判断system\_server是否启动成功，不成功则重启zygote  if (waitpid(pid, &status, WNOHANG) == pid) {  ALOGE("System server process %d has died. Restarting Zygote!", pid);  RuntimeAbort(env, \_\_LINE\_\_, "System server process has died. Restarting Zygote!");  }  }  return pid; } |

与创建普通进程的执行流程一样，com\_android\_internal\_os\_Zygote\_nativeForkSystemServer()最终也是通过调用ForkAndSpecializeCommon()来创建system\_server进程，然后执行waitpid来判断system\_server是否退出(WNOHANG-若pid指定的子进程没有结束，则waitpid()函数返回0，不予以等待。若结束，则返回该子进程的ID)，如果返回了system\_server的pid说明system\_server启动失败，Zygote执行RuntimeAbort自杀，然后通过Init重启Zygote与system\_server。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | <<< frameworks/base/core/jni/com\_android\_internal\_os\_Zygote.cpp  // Utility routine to fork zygote and specialize the child process. 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) {  　SetSigChldHandler();　//设置SIGCHLD处理函数SigChldHandler  ......   pid\_t pid = fork();   if (pid == 0) {  // The child process.  ......  } else if (pid > 0) {  // the parent process  ......  }  return pid; } |

ForkAndSpecializeCommon中通过设置SetSigChldHandler接收子进程退出的消息，如果接收到gSystemServerPid退出，那么Zygote将自杀，然后通过Init重启Zygote与system\_server。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | <<< frameworks/base/core/jni/com\_android\_internal\_os\_Zygote.cpp  static void SigChldHandler(int /\*signal\_number\*/) {  pid\_t pid; 　......  while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {  ......  // 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", pid);  kill(getpid(), SIGKILL);  }  }  ...... } |

在ForkAndSpecializeCommon中fork出子进程systemServer后，父子进程返回继续执行，子进程(pid == 0)继续执行handleSystemServerProcess。

**Zygote.handleSystemServerProcess()**

在Zygote成功fork出子进程后，将在子进程system\_server中执行handleSystemServerProcess()

|  |  |
| --- | --- |
| 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 | <<< frameworks/base/core/java/com/android/internal/os/ZygoteInit.java  private static void handleSystemServerProcess(  ZygoteConnection.Arguments parsedArgs)  throws ZygoteInit.MethodAndArgsCaller {  //关闭继承自父进程zygote的Socket  closeServerSocket();   //umask 0077后system\_server创建的文件属性为0700  Os.umask(S\_IRWXG | S\_IRWXO);   //设置进程名"system\_server",ps可以看到这个进程名  if (parsedArgs.niceName != null) {  Process.setArgV0(parsedArgs.niceName);　  }   // 获取环境变量SYSTEMSERVERCLASSPATH  //SYSTEMSERVERCLASSPATH=/system/framework/services.jar:/system/framework/ethernet-service.jar:/system/framework/wifi-service.jar:/system/framework/container-service.jar  final String systemServerClasspath = Os.getenv("SYSTEMSERVERCLASSPATH");  if (systemServerClasspath != null) {  // 创建与installd的socket连接，对systemServerClasspath执行dex优化操作  performSystemServerDexOpt(systemServerClasspath);  }   if (parsedArgs.invokeWith != null) {  ......  } else {  ClassLoader cl = null;  // 创建systemServer ClassLoader  if (systemServerClasspath != null) {  cl = createSystemServerClassLoader(systemServerClasspath,  parsedArgs.targetSdkVersion);   Thread.currentThread().setContextClassLoader(cl);  }   /\*  \* Pass the remaining arguments to SystemServer.  \*/  RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.remainingArgs, cl);  }   /\* should never reach here \*/ } |

handleSystemServerProcess()中在关闭Zygote中复制而来的socket，设置进程名，并继续执行RuntimeInit.zygoteInit()。

**RuntimeInit.zygoteInit()**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | <<< frameworks/base/core/java/com/android/internal/os/RuntimeInit.java  public static final void zygoteInit(int targetSdkVersion, String[] argv, ClassLoader classLoader)  throws ZygoteInit.MethodAndArgsCaller {  if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application from zygote");   Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "RuntimeInit");  redirectLogStreams();   commonInit();  nativeZygoteInit();  applicationInit(targetSdkVersion, argv, classLoader); } |

zygoteInit()中主要执行初始化相关的工作，最后将调到startClas(SystemServer)的main方法。commonInit中主要设置了默认的未捕获异常处理方法，timezone等。

|  |  |
| --- | --- |
| 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 | <<< frameworks/base/core/java/com/android/internal/os/RuntimeInit.java  private static final void commonInit() {  // 设置Default未捕捉异常处理方法  /\* set default handler; this applies to all threads in the VM \*/  Thread.setDefaultUncaughtExceptionHandler(new UncaughtHandler());   /\*  \* Install a TimezoneGetter subclass for ZoneInfo.db  \*/  TimezoneGetter.setInstance(new TimezoneGetter() {  @Override  public String getId() {  return SystemProperties.get("persist.sys.timezone");  }  });  TimeZone.setDefault(null);   /\*  \* Sets handler for java.util.logging to use Android log facilities.  \* The odd "new instance-and-then-throw-away" is a mirror of how  \* the "java.util.logging.config.class" system property works. We  \* can't use the system property here since the logger has almost  \* certainly already been initialized.  \*/  LogManager.getLogManager().reset();  new AndroidConfig();   /\*  \* Sets the default HTTP User-Agent used by HttpURLConnection.  \*/  String userAgent = getDefaultUserAgent();  System.setProperty("http.agent", userAgent);   ......   initialized = true; } |

nativeZygoteInit最终将调用到jni/AndroidRuntime.cpp中的onZygoteInit方法。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | <<<　frameworks/base/core/jni/AndroidRuntime.cpp  { "nativeZygoteInit", "()V",  (void\*) com\_android\_internal\_os\_RuntimeInit\_nativeZygoteInit },  static void com\_android\_internal\_os\_RuntimeInit\_nativeZygoteInit(JNIEnv\* env, jobject clazz) {  gCurRuntime->onZygoteInit(); }  virtual void onZygoteInit() {  sp<ProcessState> proc = ProcessState::self();  ALOGV("App process: starting thread pool.\n");  proc->startThreadPool(); } |

onZygoteInit中创建SystemServer进程的ProcessState对象，打开并映射binder，并开启binder线程池。

|  |  |
| --- | --- |
| 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 52 53 54 55 56 57 58 59 | >>> frameworks/base/core/java/com/android/internal/os/RuntimeInit.java  private static void applicationInit(int targetSdkVersion, String[] argv, ClassLoader classLoader)  throws ZygoteInit.MethodAndArgsCaller {  ......  设置虚拟机的内存利用率及targetSdkVersion  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  invokeStaticMain(args.startClass, args.startArgs, classLoader); }  private static void invokeStaticMain(String className, String[] argv, ClassLoader classLoader)  throws ZygoteInit.MethodAndArgsCaller {  Class<?> cl;   try {  cl = Class.forName(className, true, classLoader);  } catch (ClassNotFoundException ex) {  throw new RuntimeException(  "Missing class when invoking static main " + className,  ex);  }   Method m;  try {  m = cl.getMethod("main", new Class[] { String[].class });  } catch (NoSuchMethodException ex) {  throw new RuntimeException(  "Missing static main on " + className, ex);  } catch (SecurityException ex) {  throw new RuntimeException(  "Problem getting static main on " + className, ex);  }   int modifiers = m.getModifiers();  if (! (Modifier.isStatic(modifiers) && Modifier.isPublic(modifiers))) {  throw new RuntimeException(  "Main method is not public and static on " + className);  }   /\*  \* This throw gets caught in ZygoteInit.main(), which responds  \* by invoking the exception's run() method. This arrangement  \* clears up all the stack frames that were required in setting  \* up the process.  \*/  throw new ZygoteInit.MethodAndArgsCaller(m, argv); } |

最终在invokeStaticMain中抛出MethodAndArgsCaller异常，并传递参数”com.android.server.SystemServer.main”,ZygoteInit.main()中catch该异常，并调用MethodAndArgsCaller.run反射执行SystemServer.main方法。

**MethodAndArgsCaller.run()**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | public void run() {  try {  mMethod.invoke(null, new Object[] { mArgs });  } catch (IllegalAccessException ex) {  throw new RuntimeException(ex);  } catch (InvocationTargetException ex) {  Throwable cause = ex.getCause();  if (cause instanceof RuntimeException) {  throw (RuntimeException) cause;  } else if (cause instanceof Error) {  throw (Error) cause;  }  throw new RuntimeException(ex);  } } |

[#Android](http://www.robinheztto.com/tags/Android/) [#Framework](http://www.robinheztto.com/tags/Framework/)

# Android之SystemServer启动(下)

发表于 2016-11-28   |

上篇已经分析了SystemServer进程的创建过程，在ZygoteInit.main() catch MethodAndArgsCaller异常后，将执行MethodAndArgsCaller.run并反射SystemServer.main()方法，下面将从SystemServer.main()开始分析。  
相关的源代码在以下文件中：

* frameworks/base/services/java/com/android/server/SystemServer.java
* frameworks/base/core/java/android/app/ActivityThread.java
* frameworks/base/core/java/android/app/ContextImpl.java

# SystemServer.main()

SystemServer.main()的代码如下：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | <<< frameworks/base/services/java/com/android/server/SystemServer.java  /\*\*  \* The main entry point from zygote.  \*/ public static void main(String[] args) {  new SystemServer().run(); }  public SystemServer() {  // Check for factory test mode.  mFactoryTestMode = FactoryTest.getMode(); } |

main()中创建SystemServer对象并调用其run()方法，run()的实现如下：

|  |  |
| --- | --- |
| 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 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 | <<< frameworks/base/services/java/com/android/server/SystemServer.java  // system\_server进程binder线程池最大线程数 private static final int sMaxBinderThreads = 31;  private void run() {  try {  Trace.traceBegin(Trace.TRACE\_TAG\_SYSTEM\_SERVER, "InitBeforeStartServices");  // 如果系统时间早于1970年，调整为1970年  if (System.currentTimeMillis() < EARLIEST\_SUPPORTED\_TIME) {  Slog.w(TAG, "System clock is before 1970; setting to 1970.");  SystemClock.setCurrentTimeMillis(EARLIEST\_SUPPORTED\_TIME);  }   // 设置language相关属性  if (!SystemProperties.get("persist.sys.language").isEmpty()) {  final String languageTag = Locale.getDefault().toLanguageTag();   SystemProperties.set("persist.sys.locale", languageTag);  SystemProperties.set("persist.sys.language", "");  SystemProperties.set("persist.sys.country", "");  SystemProperties.set("persist.sys.localevar", "");  }   // Here we go!  Slog.i(TAG, "Entered the Android system server!");  EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_SYSTEM\_RUN, SystemClock.uptimeMillis());   // 设置persist.sys.dalvik.vm.lib.2属性值为当前虚拟机运行库，默认libart.so  SystemProperties.set("persist.sys.dalvik.vm.lib.2", VMRuntime.getRuntime().vmLibrary());   // 开启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);  }   // 清除VM内存增长上限  VMRuntime.getRuntime().clearGrowthLimit();   // 设置VM堆内存利用率为0.8  VMRuntime.getRuntime().setTargetHeapUtilization(0.8f);   // Some devices rely on runtime fingerprint generation, so make sure  // we've defined it before booting further.  Build.ensureFingerprintProperty();   // 设置需指定用户访问环境变量  Environment.setUserRequired(true);   // Within the system server, any incoming Bundles should be defused  // to avoid throwing BadParcelableException.  BaseBundle.setShouldDefuse(true);   // 确保binder调用运行在前台优先级(foreground priority)  BinderInternal.disableBackgroundScheduling(true);   // 设置system\_server binder线程池最大线程数为sMaxBinderThreads  BinderInternal.setMaxThreads(sMaxBinderThreads);   // 主线程的Looper在当前线程中循环  android.os.Process.setThreadPriority(  android.os.Process.THREAD\_PRIORITY\_FOREGROUND);  android.os.Process.setCanSelfBackground(false);  Looper.prepareMainLooper();   // 加载libandroid\_servers.so(对应源码frameworks/base/services/core/jni/)，初始化本地服务  System.loadLibrary("android\_servers");   // Check whether we failed to shut down last time we tried.  // This call may not return.  performPendingShutdown();   // 获取系统Context，后面具体分析  createSystemContext();   // 创建SystemServiceManager，用于启动系统服务  mSystemServiceManager = new SystemServiceManager(mSystemContext);  // 将SystemServiceManager也添加到LocalServices中管理  LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);  } finally {  Trace.traceEnd(Trace.TRACE\_TAG\_SYSTEM\_SERVER);  }   // 启动服务  try {  Trace.traceBegin(Trace.TRACE\_TAG\_SYSTEM\_SERVER, "StartServices");  startBootstrapServices();  startCoreServices();  startOtherServices();  } catch (Throwable ex) {  Slog.e("System", "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  Slog.e("System", "\*\*\*\*\*\*\*\*\*\*\*\* Failure starting system services", ex);  throw ex;  } finally {  Trace.traceEnd(Trace.TRACE\_TAG\_SYSTEM\_SERVER);  }   // debug版本log输出到dropbox  if (StrictMode.conditionallyEnableDebugLogging()) {  Slog.i(TAG, "Enabled StrictMode for system server main thread.");  }   // 消息队列循环.  Looper.loop();  throw new RuntimeException("Main thread loop unexpectedly exited"); } |

run()中主要完成相关的初始化工作，启动系统服务，随后进入到消息队列的循环中。下面将具体分析一下createSystemContext()与启动服务的过程。

# createSystemContext()

createSystemContext()的实现如下：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | <<< frameworks/base/services/java/com/android/server/SystemServer.java  private void createSystemContext() {  ActivityThread activityThread = ActivityThread.systemMain();  mSystemContext = activityThread.getSystemContext();  mSystemContext.setTheme(DEFAULT\_SYSTEM\_THEME); } |

createSystemContext()中创建了ActivityThread对象，并通过其getSystemContext获取系统Context，然后通过系统Context设置主题，接下来一步步分析此过程。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | <<< frameworks/base/core/java/android/app/ActivityThread.java  public static ActivityThread systemMain() {  // The system process on low-memory devices do not get to use hardware  // accelerated drawing, since this can add too much overhead to the  // process.  if (!ActivityManager.isHighEndGfx()) {  HardwareRenderer.disable(true);  } else {  HardwareRenderer.enableForegroundTrimming();  }  ActivityThread thread = new ActivityThread();  thread.attach(true);  return thread; } |

ActivityThread.systemMain()中创建ActivityThread对象并调用其attch方法，具体实现如下：

|  |  |
| --- | --- |
| 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 | <<< frameworks/base/core/java/android/app/ActivityThread.java  ActivityThread() {  mResourcesManager = ResourcesManager.getInstance(); }  private void attach(boolean system) {  sCurrentActivityThread = this;  mSystemThread = system;  if (!system) {  ......   } else {  // 设置DDMS中名称  android.ddm.DdmHandleAppName.setAppName("system\_process",  UserHandle.myUserId());  try {  // 创建Instrumentation对象   mInstrumentation = new Instrumentation();  // 通过ContextImpl创建context对象  ContextImpl context = ContextImpl.createAppContext(  this, getSystemContext().mPackageInfo);  // 创建Application对象并调用onCreate()方法   mInitialApplication = context.mPackageInfo.makeApplication(true, null);  mInitialApplication.onCreate();  } catch (Exception e) {  throw new RuntimeException(  "Unable to instantiate Application():" + e.toString(), e);  }  }   // add dropbox logging to libcore  DropBox.setReporter(new DropBoxReporter());   ViewRootImpl.addConfigCallback(new ComponentCallbacks2() {  public void onConfigurationChanged(Configuration newConfig) {...}  public void onLowMemory() {...}  public void onTrimMemory(int level) {...}  }); } |

ActivityThread的attach方法中通过getSystemContext获取系统Context对象，getSystemContext的实现如下：

|  |  |
| --- | --- |
| 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 | <<< frameworks/base/core/java/android/app/ActivityThread.java  public ContextImpl getSystemContext() {  synchronized (this) {  if (mSystemContext == null) {  mSystemContext = ContextImpl.createSystemContext(this);  }  return mSystemContext;  } }  <<< frameworks/base/core/java/android/app/ContextImpl.java  static ContextImpl createSystemContext(ActivityThread mainThread) {  LoadedApk packageInfo = new LoadedApk(mainThread);  ContextImpl context = new ContextImpl(null, mainThread,  packageInfo, null, null, false, null, null, Display.INVALID\_DISPLAY);  context.mResources.updateConfiguration(context.mResourcesManager.getConfiguration(),  context.mResourcesManager.getDisplayMetricsLocked());  return context; }  <<< frameworks/base/core/java/android/app/LoadedApk.java  LoadedApk(ActivityThread activityThread) {  mActivityThread = activityThread;  mApplicationInfo = new ApplicationInfo();  // packageName设置成android  mApplicationInfo.packageName = "android";  mPackageName = "android";  mAppDir = null;  mResDir = null;  mSplitAppDirs = null;  mSplitResDirs = null;  mOverlayDirs = null;  mSharedLibraries = null;  mDataDir = null;  mDataDirFile = null;  mLibDir = null;  mBaseClassLoader = null;  mSecurityViolation = false;  mIncludeCode = true;  mRegisterPackage = false;  mClassLoader = ClassLoader.getSystemClassLoader();  mResources = Resources.getSystem(); } |

getSystemContext()中调用ContextImpl.createSystemContext()创建了ContextImpl对象，以获取类似apk的Context上下文环境，在创建ContextImpl同时也需要LoadedApk对象，对应framework-res.apk，PackageName命名为android。

# StartServices

SystemServer分成三类启动服务，分别时Bootstrap，Core，Other，首先分析startBootstrapServices启动Bootstrap服务流程。

|  |  |
| --- | --- |
| 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 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 | private void startBootstrapServices() {  // 等待与installd建立socket通信  Installer installer = mSystemServiceManager.startService(Installer.class);   // 启动ActivityManagerService  mActivityManagerService = mSystemServiceManager.startService(  ActivityManagerService.Lifecycle.class).getService();  mActivityManagerService.setSystemServiceManager(mSystemServiceManager);  mActivityManagerService.setInstaller(installer);   // 启动PowerManagerService  mPowerManagerService = mSystemServiceManager.startService(PowerManagerService.class);   // 初始化PowerManagement  mActivityManagerService.initPowerManagement();   // 启动LightsService  mSystemServiceManager.startService(LightsService.class);   // 启动DisplayManagerService  mDisplayManagerService = mSystemServiceManager.startService(DisplayManagerService.class);   // Phase100阶段  mSystemServiceManager.startBootPhase(SystemService.PHASE\_WAIT\_FOR\_DEFAULT\_DISPLAY);   // 当设备加密时仅仅运行core  String cryptState = SystemProperties.get("vold.decrypt");  if (ENCRYPTING\_STATE.equals(cryptState)) {  Slog.w(TAG, "Detected encryption in progress - only parsing core apps");  mOnlyCore = true;  } else if (ENCRYPTED\_STATE.equals(cryptState)) {  Slog.w(TAG, "Device encrypted - only parsing core apps");  mOnlyCore = true;  }   // 启动PackageManagerService  mPackageManagerService = PackageManagerService.main(mSystemContext, installer,  mFactoryTestMode != FactoryTest.FACTORY\_TEST\_OFF, mOnlyCore);  mFirstBoot = mPackageManagerService.isFirstBoot();  mPackageManager = mSystemContext.getPackageManager();   // Manages A/B OTA dexopting. This is a bootstrap service as we need it to rename  // A/B artifacts after boot, before anything else might touch/need them.  // Note: this isn't needed during decryption (we don't have /data anyways).  if (!mOnlyCore) {  boolean disableOtaDexopt = SystemProperties.getBoolean("config.disable\_otadexopt",  false);  if (!disableOtaDexopt) {  traceBeginAndSlog("StartOtaDexOptService");  try {  OtaDexoptService.main(mSystemContext, mPackageManagerService);  } catch (Throwable e) {  reportWtf("starting OtaDexOptService", e);  } finally {  Trace.traceEnd(Trace.TRACE\_TAG\_SYSTEM\_SERVER);  }  }  }   // 启动UserManagerService  mSystemServiceManager.startService(UserManagerService.LifeCycle.class);   // Initialize attribute cache used to cache resources from packages.  AttributeCache.init(mSystemContext);   // Set up the Application instance for the system process and get started.  mActivityManagerService.setSystemProcess();   // 启动SensorService  startSensorService(); } |

startBootstrapServices()中主要启动了ActivityManagerService, PowerManagerService, LightsService, DisplayManagerService， PackageManagerService， UserManagerService， sensor服务。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | private void startCoreServices() {  // 启动BatteryService(Tracks the battery level. Requires LightService)  mSystemServiceManager.startService(BatteryService.class);   // 启动UsageStatsService  mSystemServiceManager.startService(UsageStatsService.class);  mActivityManagerService.setUsageStatsManager(  LocalServices.getService(UsageStatsManagerInternal.class));  // Update after UsageStatsService is available, needed before performBootDexOpt.  mPackageManagerService.getUsageStatsIfNoPackageUsageInfo();   // 启动WebViewUpdateService  mSystemServiceManager.startService(WebViewUpdateService.class); } |

startCoreServices()中主要启动了BatteryService，UsageStatsService，WebViewUpdateService。

Android之SystemService

发表于 2016-12-20   |

Android系统中，大部分的系统服务都运行在SystemServer中，且基本上都继承于SystemService(除AMS,PMS等特例外)，本篇具体分析SystemService。相关代码在以下文件中：  
frameworks/base/services/core/java/com/android/server/SystemService.java  
frameworks/base/services/core/java/com/android/server/SystemServiceManager.java  
frameworks/base/core/java/com/android/server/LocalServices.java  
frameworks/base/services/java/com/android/server/SystemServer.java  
frameworks/base/core/java/android/app/SystemServiceRegistry.java  
frameworks/base/core/java/android/content/Context.java  
frameworks/base/core/java/android/app/ContextImpl.java

**SystemService**

SystemService是一个抽象类，具体的系统服务继承并实现其抽象方法，在SystemService中定义了Boot Phases启动阶段及系统服务启动与事件的回调。下面首先看一下Boot Phases的定义：

|  |  |
| --- | --- |
| 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 | >>> frameworks/base/services/core/java/com/android/server/SystemService.java  public abstract class SystemService {  /\*\*  \* 创建ActivityManagerService，PowerManagerService，LightsService，  \* DisplayManagerService之后，PackageManagerService之前.  \*/   public static final int PHASE\_WAIT\_FOR\_DEFAULT\_DISPLAY = 100;   /\*\*  \* 创建PackageManagerService，WindowManagerService等等诸多服务.  \*/  public static final int PHASE\_LOCK\_SETTINGS\_READY = 480;   /\*\*  \* PHASE\_LOCK\_SETTINGS\_READY到PHASE\_SYSTEM\_SERVICES\_READY之间无任何操作，PHASE\_SYSTEM\_SERVICES\_READY之后，  \*　系统服务（例如PowerManager，PackageManager等等）能够被调用.  \*/  public static final int PHASE\_SYSTEM\_SERVICES\_READY = 500;   /\*\*  \* ActivityManagerService ready.  \*/  public static final int PHASE\_ACTIVITY\_MANAGER\_READY = 550;   /\*\*  \* 启动SystemUI，Watchdog，services能够start/bind三方应用，App能够通过Binder调用系统服务.  \*/  public static final int PHASE\_THIRD\_PARTY\_APPS\_CAN\_START = 600;   /\*\*  \* 启动完成，系统服务执行PHASE\_BOOT\_COMPLETED回调比注册ACTION\_BOOT\_COMPLETED广播能减低延迟.  \*/  public static final int PHASE\_BOOT\_COMPLETED = 1000;  } |

下面继续看SystemService中定义的方法，主要是系统服务生命周期/系统事件的回调。

|  |  |
| --- | --- |
| 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 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 | /\*\*  \* 构造方法，初始化system server context.  \*  \* @param context The system server context.  \*/ public SystemService(Context context) {  mContext = context; }  /\*\*  \* 获取system context.  \*/ public final Context getContext() {  return mContext; }  /\*\*  \* 是否是安全模式.  \*/ public final boolean isSafeMode() {  return getManager().isSafeMode(); }  /\*\*  \* 在构造方法后调用.  \*/ public abstract void onStart();  /\*\*  \* 不同BootPhase阶段的回调.  \*/ public void onBootPhase(int phase) {}  ......  /\*\*  \* Publish the service到ServiceManager，对其他进程提供binder服务.  \*/ protected final void publishBinderService(String name, IBinder service) {  publishBinderService(name, service, false); }  /\*\*  \* Publish the service到ServiceManager，对其他进程提供binder服务.  \*/ protected final void publishBinderService(String name, IBinder service,  boolean allowIsolated) {  ServiceManager.addService(name, service, allowIsolated); }  /\*\*  \* 通过服务名获取binder服务.  \*/ protected final IBinder getBinderService(String name) {  return ServiceManager.getService(name); }  /\*\*  \* Publish service到LocalServices，以供SystemServer进程中的其他服务调用.  \*/ protected final <T> void publishLocalService(Class<T> type, T service) {  LocalServices.addService(type, service); }  /\*\*  \* 通过Class Type从LocalServices中获取服务.  \*/ protected final <T> T getLocalService(Class<T> type) {  return LocalServices.getService(type); }  /\*\*  \* 从LocalServices中获取SystemServiceManager（SystemServiceManager也被注册到了从LocalServices）.  \*/ private SystemServiceManager getManager() {  return LocalServices.getService(SystemServiceManager.class); } |

**SystemServiceManager**

SystemServiceManager用来管理系统服务的生命周期，事件回调等。在SystemServer启动过程中创建SystemServiceManager，并通过SystemServiceManager启动系统服务并根据事件回调其方法。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | >>> frameworks/base/services/java/com/android/server/SystemServer.java  private void run() {  try {  ......  // 在启动系统服务之前，先创建SystemServiceManager并注册到LocalServices中.  mSystemServiceManager = new SystemServiceManager(mSystemContext);  LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);  } finally {  ......  }   // Start services.  try {  startBootstrapServices();  startCoreServices();  startOtherServices();  } catch (Throwable ex) {  ......  } finally {  ......  }  ...... } |

SystemServiceManag它的实现如下：

|  |  |
| --- | --- |
| 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 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 | >>> frameworks/base/services/core/java/com/android/server/SystemServiceManager.java  public class SystemServiceManager {  private final Context mContext;  private boolean mSafeMode;   // 系统服务列表.  private final ArrayList<SystemService> mServices = new ArrayList<SystemService>();   private int mCurrentPhase = -1;   public SystemServiceManager(Context context) {  mContext = context;  }   /\*\*  \* 通过className启动系统服务.  \*  \* @return The service instance.  \*/  @SuppressWarnings("unchecked")  public SystemService startService(String className) {  final Class<SystemService> serviceClass;  try {  serviceClass = (Class<SystemService>)Class.forName(className);  } catch (ClassNotFoundException ex) {  ......  }  return startService(serviceClass);  }   /\*\*  \* 创建系统服务实例并返回，同时调用其onStart()方法.  \*/  @SuppressWarnings("unchecked")  public <T extends SystemService> T startService(Class<T> serviceClass) {  try {  final String name = serviceClass.getName();  // Create the service.  if (!SystemService.class.isAssignableFrom(serviceClass)) {  throw new RuntimeException("Failed to create " + name  + ": service must extend " + SystemService.class.getName());  }  final T service;  try {  Constructor<T> constructor = serviceClass.getConstructor(Context.class);  service = constructor.newInstance(mContext);  } catch (......) {  ......  }   // Register it.  mServices.add(service);   // Start it.  try {  service.onStart();  } catch (RuntimeException ex) {  ......  }  return service;  } finally {  ......  }  }   /\*\*  \*　根据boot phase不同阶段，回调系统服务中的onBootPhase()方法．  \* @param phase The boot phase to start.  \*/  public void startBootPhase(final int phase) {  if (phase <= mCurrentPhase) {  throw new IllegalArgumentException("Next phase must be larger than previous");  }  mCurrentPhase = phase;   try {  final int serviceLen = mServices.size();  for (int i = 0; i < serviceLen; i++) {  final SystemService service = mServices.get(i);  try {  service.onBootPhase(mCurrentPhase);  } catch (Exception ex) {  ......  }  }  } finally {  ......  }  }   ......   /\*\* 设置安全模式供系统服务查询. \*/  public void setSafeMode(boolean safeMode) {  mSafeMode = safeMode;  }   /\*\*  \* 返回当前是否是安全模式启动.  \* @return safe mode flag  \*/  public boolean isSafeMode() {  return mSafeMode;  } } |

**LocalServices**

LocalServices的作用与SystemServiceManager类似，不过LocalService的主要目的是提供系统服务的查询，注册，以供SystemServer进程内部的其他服务调用。

|  |  |
| --- | --- |
| 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 | >>> frameworks/base/core/java/com/android/server/LocalServices.java  public final class LocalServices {  private LocalServices() {}   // 保存所有系统服务，包过SystemServiceManager   private static final ArrayMap<Class<?>, Object> sLocalServiceObjects =  new ArrayMap<Class<?>, Object>();   /\*\*  \* 获取服务  \*  \* @param type The type of service.  \* @return The service object.  \*/  @SuppressWarnings("unchecked")  public static <T> T getService(Class<T> type) {  synchronized (sLocalServiceObjects) {  return (T) sLocalServiceObjects.get(type);  }  }   /\*\*  \* 注册服务  \*/  public static <T> void addService(Class<T> type, T service) {  synchronized (sLocalServiceObjects) {  if (sLocalServiceObjects.containsKey(type)) {  throw new IllegalStateException("Overriding service registration");  }  sLocalServiceObjects.put(type, service);  }  }  ...... } |

**Context.getSystemService**

系统服务运行在SystemServer进程中，App使用系统服务首先得通过Context.getSystemService()获取系统服务的代理，下面继续看Context.getSystemService()的过程。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | >>> frameworks/base/core/java/android/content/Context.java  public abstract class Context {  ......  public abstract Object getSystemService(@ServiceName @NonNull String name);   public final <T> T getSystemService(Class<T> serviceClass) {  String serviceName = getSystemServiceName(serviceClass);  return serviceName != null ? (T)getSystemService(serviceName) : null;  }  ......  } |

Context.getSystemService()在ContextImpl中具体实现。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | >>> frameworks/base/core/java/android/app/ContextImpl.java  class ContextImpl extends Context {  ......  @Override  public Object getSystemService(String name) {  return SystemServiceRegistry.getSystemService(this, name);  }   @Override  public String getSystemServiceName(Class<?> serviceClass) {  return SystemServiceRegistry.getSystemServiceName(serviceClass);  }  ...... ｝ |

ContextImpl中是直接调用SystemServiceRegistry.getSystemService()来获取系统服务。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | >>> frameworks/base/core/java/android/app/SystemServiceRegistry.java  private static final HashMap<String, ServiceFetcher<?>> SYSTEM\_SERVICE\_FETCHERS =  new HashMap<String, ServiceFetcher<?>>();  public static Object getSystemService(ContextImpl ctx, String name) {  ServiceFetcher<?> fetcher = SYSTEM\_SERVICE\_FETCHERS.get(name);  return fetcher != null ? fetcher.getService(ctx) : null; } |

根据服务名从SYSTEM\_SERVICE\_FETCHERS找到服务对应的ServiceFetcher，并通过ServiceFetcher.getService()获取到对应的服务。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | >>> frameworks/base/core/java/android/app/SystemServiceRegistry.java  static {  ......   registerService(Context.ACTIVITY\_SERVICE, ActivityManager.class,  new CachedServiceFetcher<ActivityManager>() {  @Override  public ActivityManager createService(ContextImpl ctx) {  return new ActivityManager(ctx.getOuterContext(), ctx.mMainThread.getHandler());  }});  ...... }  private static <T> void registerService(String serviceName, Class<T> serviceClass,  ServiceFetcher<T> serviceFetcher) {  SYSTEM\_SERVICE\_NAMES.put(serviceClass, serviceName);  SYSTEM\_SERVICE\_FETCHERS.put(serviceName, serviceFetcher); } |

在SystemServiceRegistry中的静态区域中完成了系统服务的注册，serviceName，serviceClass，serviceFetcher保存到SYSTEM\_SERVICE\_NAMES与SYSTEM\_SERVICE\_FETCHERS中。下面接着看fetcher.getService(ctx)的过程。

|  |  |
| --- | --- |
| 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 | >>> frameworks/base/core/java/android/app/SystemServiceRegistry.java  static abstract interface ServiceFetcher<T> {  T getService(ContextImpl ctx); }  static abstract class CachedServiceFetcher<T> implements ServiceFetcher<T> {  private final int mCacheIndex;   public CachedServiceFetcher() {  mCacheIndex = sServiceCacheSize++;  }   @Override  @SuppressWarnings("unchecked")  public final T getService(ContextImpl ctx) {  final Object[] cache = ctx.mServiceCache;  synchronized (cache) {  // Fetch or create the service.  Object service = cache[mCacheIndex];  if (service == null) {  service = createService(ctx);  cache[mCacheIndex] = service;  }  return (T)service;  }  }   public abstract T createService(ContextImpl ctx); } |

根据前面静态区域注册服务代码可知，注册服务时创建CachedServiceFetcher并实现了其createService方法，当通过fetcher.getService(ctx)获取服务时，首先判断cache中是否存在缓存的对象，如果没有，则调用createService()创建服务代理对象并返回，至此，APP就获取到了系统服务代理对象，通过该代理就可以调用各种系统服务了。