**Zygote进程解读**

**一、什么是Zygote进程**

Zygote这个词是受精卵的意思。在Android系统中，所有的应用程序进程以及系统服务进程SystemServer都是由Zygote进程孕育（fork）出来的。因此，Zygote进程在Android系统中有着非常重要的地位。

每一个Android应用程序进程都是由Zygote进程fork出来的。Zygote进程是由init进程启动起来的，也就是在系统启动的时候启动的。Zygote进程在启动的时候，会创建一个虚拟机实例，并且在这个虚拟机实例将所有的Java核心库都加载起来。每当Zygote 进程需要创建一个Android应用程序进程的时候，它就通过复制自身来实现，也就是通过fork系统调用来实现。这些被fork出来的Android应用程序进程，一方面是复制了Zygote进程中的虚拟机实例，另一方面是与Zygote进程共享了同一套Java核心库。这样不仅Android应用程序进程的创建过程很快，而且由于所有的Android应用程序进程都共享同一套Java核心库而节省了内存空间。

Zygote本身是一个应用层的程序，和驱动，内核模块之类的没点关系。通过ps命令可看到进程名为zygote，其最初的名字是app\_process，通过直接调用pctrl把名字给改成了“zygote”。下面结合源码具体看一下Zygote进程在启动过程中做了哪些事情。

**二、Zygote进程的启动过程**

Zygote进程的诞生对于整个Java世界可以说有着”开天辟地“的作用，它创建了Java虚拟机，并且繁殖了Java世界的核心服务system\_server进程，在完成Java世界的初创工作以后，Zygote并没有死去，它只是暂时的沉睡（socket事件堵塞）在那里，一旦有需要（有客户端请求的到来），它便马上起来工作。

Zygote进程启动过程可概括为以下三个过程：

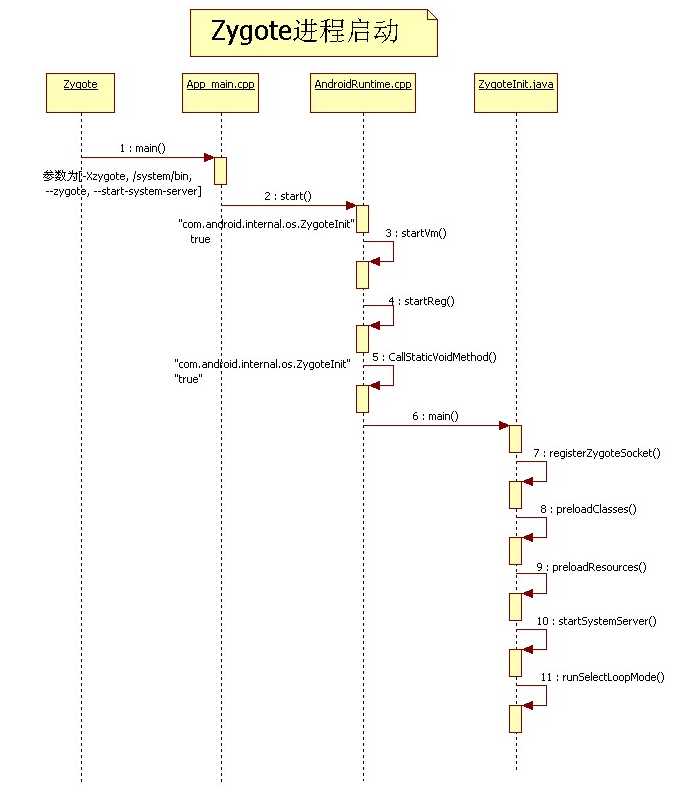
1. 系统启动时init进程会创建Zygote进程，Zygote进程负责后续Android应用程序框架层的其它进程的创建和启动工作。
2. Zygote进程会首先创建一个SystemServer进程，SystemServer进程负责启动系统的关键服务，如包管理服务PackageManagerService和应用程序组件管理服务ActivityManagerService。
3. Zygote进程启动完成后会陷入阻塞状态，等待ActivityManagerService通过Socket进程间通信发送创建进程请求，通知Zygote进程为这个应用程序创建一个新的进程。

在Linux系统中，所有的进程都是init进程的子孙进程，也就是说，所有的进程都是直接或者间接地由init进程fork出来的。Zygote进程也不例外，它是在系统启动的过程，由init进程创建的。在系统启动脚本system/core/rootdir/init.rc文件中，我们可以看到启动Zygote进程的脚本命令。

1. //关键字service告诉init进程创建一个名为"zygote"的进程，这个zygote进程要执行的程序是/system/bin/app\_process，后面是要传给app\_process的参数。
2. service zygote /system/bin/app\_process -Xzygote /system/bin --zygote --start-system-server
3. //socket关键字表示这个zygote进程需要一个名称为"zygote"的socket资源，这样，系统启动后，我们就可以在/dev/socket目录下看到有一个名为zygote的文件。这里定义的socket的类型为unix domain socket，它是用来作本地进程间通信用的
4. socket zygote stream 666
5. critical
6. onrestart write /sys/android\_power/request\_state wake
7. onrestart write /sys/power/state on
8. onrestart restart media
9. onrestart restart netd
10. onrestart restart mlistener

前面的关键字service告诉init进程创建一个名为"zygote"的进程，这个zygote进程要执行的程序是/system/bin/app\_process，后面是要传给app\_process的参数。接下来的socket关键字表示这个zygote进程需要一个名称为"zygote"的socket资源，这样，系统启动后，我们就可以在/dev/socket目录下看到有一个名为zygote的文件。这里定义的socket的类型为unix domain socket，它是用来作本地进程间通信用的，ActivityManagerService就是通这个socket来和zygote进程通信请求fork一个应用程序进程的了。最后的一系列onrestart关键字表示这个zygote进程重启时需要执行的命令。

从脚本命令得知Zygote进程要执行的程序便是system/bin/app\_process，入口函数是main，先看一下整个过程的时序图。



**Step 1. app\_process.main**

        这个函数定义在frameworks/base/cmds/app\_process/app\_main.cpp文件中：

1. **int** main(**int** argc, **const** **char**\* **const** argv[])
2. {
3. //zygote 是由init进程fork而来，init.rc文件中为zygote进程设置的启动参数如下
4. //argc = 4
5. //argv = [-Xzygote, /system/bin, --zygote, --start-system-server]
6. // These are global variables in ProcessState.cpp
7. mArgC = argc;
8. mArgV = argv;
10. mArgLen = 0;
11. **for** (**int** i=0; i<argc; i++) {
12. mArgLen += strlen(argv[i]) + 1;
13. }
14. mArgLen--;  // mArgLen = 48
16. AppRuntime runtime;
17. **const** **char** \*arg;
18. **const** **char** \*argv0;
20. argv0 = argv[0]; //-Xzygote
22. // Process command line arguments
23. // ignore argv[0]
24. argc--;
25. argv++;
27. //添加虚拟机参数选项
28. //addVmArguments(3, [/system/bin, --zygote, --start-system-server])结果返回 i=0
29. **int** i = runtime.addVmArguments(argc, argv);
31. // 设置虚拟机运行目录
32. **if** (i < argc) {
33. runtime.mParentDir = argv[i++]; //runtime.mParentDir =/system/bin
34. }
35. // i=1
36. // Next arg is startup classname or "--zygote"
37. **if** (i < argc) {
38. arg = argv[i++]; //arg = --zygote
39. //i=2
40. **if** (0 == strcmp("--zygote", arg)) {
41. //startSystemServer = true
42. bool startSystemServer = (i < argc) ? strcmp(argv[i], "--start-system-server") == 0 : **false**;
43. //设置argv0 的值为"zygote"
44. setArgv0(argv0, "zygote");
45. //设置进程名字
46. set\_process\_name("zygote");
47. // do last shutdown check
48. doLastShutDownCheck();
49. //调用Runtimed的start函数启动SystemServer进程，startSystemServer = true
50. runtime.start("com.android.internal.os.ZygoteInit",startSystemServer);
51. } **else** {
52. set\_process\_name(argv0);
54. runtime.mClassName = arg;
56. // Remainder of args get passed to startup class main()
57. runtime.mArgC = argc-i;
58. runtime.mArgV = argv+i;
60. LOGV("App process is starting with pid=%d, class=%s.\n",
61. getpid(), runtime.getClassName());
62. runtime.start();
63. }
64. } **else** {
65. LOG\_ALWAYS\_FATAL("app\_process: no class name or --zygote supplied.");
66. fprintf(stderr, "Error: no class name or --zygote supplied.\n");
67. app\_usage();
68. **return** 10;
69. }
70. }

 这个函数的主要作用就是创建一个AppRuntime变量，然后调用它的start成员函数。AppRuntime这个类同样是在frameworks/base/cmds/app\_process/app\_main.cpp文件中定义，它继承于AndroidRuntime类， AndroidRuntime类定义在frameworks/base/core/jni/AndroidRuntime.cpp文件中：

1. ......
3. **static** AndroidRuntime\* gCurRuntime = NULL;
5. ......
7. AndroidRuntime::AndroidRuntime()
8. {
9. ......
11. assert(gCurRuntime == NULL);        // one per process
12. gCurRuntime = **this**;
13. }

 当AppRuntime对象创建时，会调用其父类AndroidRuntime的构造函数，而在AndroidRuntime类的构造函数里面，会将this指针保存在静态全局变量gCurRuntime中，这样，当其它地方需要使用这个AppRuntime对象时，就可以通过同一个文件中的这个函数来获取这个对象的指针：

1. AndroidRuntime\* AndroidRuntime::getRuntime()
2. {
3. **return** gCurRuntime;
4. }

   回到上面的main函数中，由于我们在init.rc文件中，设置了app\_process启动参数--zygote和--start-system-server，因此，在main函数里面，最终会执行下面语句：

1. runtime.start("com.android.internal.os.ZygoteInit",
2. rtSystemServer);

这里的参数startSystemServer为true，表示要启动SystemServer组件。由于AppRuntime没有实现自己的start函数，它继承了父类AndroidRuntime的start函数，因此，下面会执行AndroidRuntime类的start函数。

**Step 2. AndroidRuntime.start**

        这个函数定义在frameworks/base/core/jni/AndroidRuntime.cpp文件中：

1. /\*
2. \* Start the Android runtime.  This involves starting the virtual machine
3. \* and calling the "static void main(String[] args)" method in the class
4. \* named by "className".
5. \*/
6. **void** AndroidRuntime::start(**const** **char**\* className, **const** bool startSystemServer)
7. {
8. //className = "com.android.internal.os.ZygoteInit"
9. //startSystemServer = true
10. **char**\* slashClassName = NULL;
11. **char**\* cp;
12. JNIEnv\* env;
14. blockSigpipe();
16. /\*
17. \* 'startSystemServer == true' means runtime is obslete and not run from
18. \* init.rc anymore, so we print out the boot start event here.
19. \*/
20. **if** (startSystemServer) {
21. /\* track our progress through the boot sequence \*/
22. **const** **int** LOG\_BOOT\_PROGRESS\_START = 3000;
23. LOG\_EVENT\_LONG(LOG\_BOOT\_PROGRESS\_START, ns2ms(systemTime(SYSTEM\_TIME\_MONOTONIC)));
24. }
25. //获取环境变量android\_root的根目录
26. **const** **char**\* rootDir = getenv("ANDROID\_ROOT");
27. **if** (rootDir == NULL) {
28. rootDir = "/system"; //如果获取结果为Null,则设置为"/system"
29. **if** (!hasDir("/system")) {
30. LOG\_FATAL("No root directory specified, and /android does not exist.");
31. **goto** bail;
32. }
33. setenv("ANDROID\_ROOT", rootDir, 1);//重新设置环境变量ANDROID\_ROOT
34. }
36. //const char\* kernelHack = getenv("LD\_ASSUME\_KERNEL");
37. //LOGD("Found LD\_ASSUME\_KERNEL='%s'\n", kernelHack);
39. /\* 启动虚拟机 \*/
40. **if** (startVm(&mJavaVM, &env) != 0)
41. **goto** bail;
43. /\*
44. \* 注册JNI函数
45. \*/
46. **if** (startReg(env) < 0) {
47. LOGE("Unable to register all android natives\n");
48. **goto** bail;
49. }
51. /\*
52. \* We want to call main() with a String array with arguments in it.
53. \* At present we only have one argument, the class name.  Create an
54. \* array to hold it.
55. \*/
56. jclass stringClass;
57. jobjectArray strArray;
58. jstring classNameStr;
59. jstring startSystemServerStr;
60. //通过JNI查找java的java.lang.String类
61. stringClass = env->FindClass("java/lang/String");
62. **assert**(stringClass != NULL);
63. //创建字符串数组String strArray[] = new String[2]；
64. strArray = env->NewObjectArray(2, stringClass, NULL);
65. **assert**(strArray != NULL);
66. //创建字符串classNameStr = new String("com.android.internal.os.ZygoteInit")
67. classNameStr = env->NewStringUTF(className);
68. **assert**(classNameStr != NULL);
69. //设置字符串数组元素strArray[0]="com.android.internal.os.ZygoteInit"
70. env->SetObjectArrayElement(strArray, 0, classNameStr);
71. //创建字符串startSystemServerStr= new String("true")
72. startSystemServerStr = env->NewStringUTF(startSystemServer ? "true" : "false");
73. //设置字符串数组元素strArray[1]="true"
74. env->SetObjectArrayElement(strArray, 1, startSystemServerStr);
76. /\*
77. \* Start VM.  This thread becomes the main thread of the VM, and will
78. \* not return until the VM exits.
79. \*/
80. jclass startClass;
81. jmethodID startMeth;
82. //将字符串className 复制到slashClassName
83. slashClassName = strdup(className);
84. //为符合JNI规范，将com.android.internal.os.ZygoteInit中的.换成/，变为slashClassName = com/android/internal/os/ZygoteInit
85. **for** (cp = slashClassName; \*cp != '\0'; cp++)
86. **if** (\*cp == '.')
87. \*cp = '/';
88. //查找Java类com/android/internal/os/ZygoteInit
89. startClass = env->FindClass(slashClassName);
90. **if** (startClass == NULL) {
91. LOGE("JavaVM unable to locate class '%s'\n", slashClassName);
92. /\* keep going \*/
93. } **else** {
94. //找到ZygoteInit类的静态main方法的jMethodID
95. startMeth = env->GetStaticMethodID(startClass, "main","([Ljava/lang/String;)V");
96. **if** (startMeth == NULL) {
97. LOGE("JavaVM unable to find main() in '%s'\n", className);
98. /\* keep going \*/
99. } **else** {
100. //通过JNI调用ZygoteInit类的main函数，传递的参数为：
101. //startClass = ZygoteInit
102. //startMeth = main
103. //strArray =["com.android.internal.os.ZygoteInit","true"]
104. //在调用ZygoteInit类的main方法后，zygote就进入了java世界
105. env->CallStaticVoidMethod(startClass, startMeth, strArray);
107. #**if** 0
108. **if** (env->ExceptionCheck())
109. threadExitUncaughtException(env);
110. #endif
111. }
112. }
113. LOGD("Shutting down VM\n");
114. **if** (mJavaVM->DetachCurrentThread() != JNI\_OK)
115. LOGW("Warning: unable to detach main thread\n");
116. **if** (mJavaVM->DestroyJavaVM() != 0)
117. LOGW("Warning: VM did not shut down cleanly\n");
118. bail:
119. free(slashClassName);
120. }

这个函数的作用是启动Android系统运行时库，它主要做了三件事情，一是调用函数startVM启动虚拟机，二是调用函数startReg注册JNI方法，三是调用了com.android.internal.os.ZygoteInit类的main函数。函数startVM、startReg函数代码不再贴，主要看下.ZygoteInit类的main函数代码。

**Step 3. ZygoteInit.main**

 这个函数定义在frameworks/base/core/java/com/android/internal/os/ZygoteInit.java文件中：

1. **public** **static** **void** main(String argv[]) {
2. //传入的参数argv = ["com.android.internal.os.ZygoteInit","true"]
3. **try** {
4. //设置虚拟机的最小堆栈大小
5. VMRuntime.getRuntime().setMinimumHeapSize(5 \* 1024 \* 1024);
7. // Start profiling the zygote initialization.启动性能统计
8. SamplingProfilerIntegration.start();
9. //注册zygote等待客户端连接的socket
10. registerZygoteSocket();
11. EventLog.writeEvent(LOG\_BOOT\_PROGRESS\_PRELOAD\_START, SystemClock.uptimeMillis());
12. //预加载java类和资源
13. preloadClasses();
14. preloadResources();
15. EventLog.writeEvent(LOG\_BOOT\_PROGRESS\_PRELOAD\_END,SystemClock.uptimeMillis());
17. // Finish profiling the zygote initialization.结束统计并生成结果文件
18. SamplingProfilerIntegration.writeZygoteSnapshot();
20. // Do an initial gc to clean up after startup，执行垃圾回收
21. gc();
23. // If requested, start system server directly from Zygote
24. **if** (argv.length != 2) {
25. **throw** **new** RuntimeException(argv[0] + USAGE\_STRING);
26. }
28. **if** (argv[1].equals("true")) {
29. //启动SystemServer进程
30. startSystemServer();
31. } **else** **if** (!argv[1].equals("false")) {
32. **throw** **new** RuntimeException(argv[0] + USAGE\_STRING);
33. }
35. Log.i(TAG, "Accepting command socket connections");
36. //boolean ZYGOTE\_FORK\_MODE = false； 因此调用runSelectLoopMode()函数
37. **if** (ZYGOTE\_FORK\_MODE) {
38. runForkMode();
39. } **else** {
40. runSelectLoopMode();
41. }
42. closeServerSocket(); //关闭socket
43. } **catch** (MethodAndArgsCaller caller) {
44. //捕获SytemServer进程调用RuntimeInit.java 中zygoteInit函数抛出的MethodAndArgsCaller异常
45. caller.run();
46. } **catch** (RuntimeException ex) {
47. Log.e(TAG, "Zygote died with exception", ex);
48. closeServerSocket();
49. **throw** ex;
50. }
51. }

         它主要作了四件事情，一是调用registerZygoteSocket函数创建了一个socket接口，用来和ActivityManagerService通讯，二是调用preload函数预加载类与资源，三是调用startSystemServer函数来启动SystemServer组件，四是调用runSelectLoopMode函数进入一个无限循环在前面创建的socket接口上等待ActivityManagerService请求创建新的应用程序进程。

**Step 4. ZygoteInit.registerZygoteSocket**

         这个函数定义frameworks/base/core/java/com/android/internal/os/ZygoteInit.java文件中：

1. **public** **class** ZygoteInit {
2. ......
4. /\*\*
5. \* Registers a server socket for zygote command connections
6. \*
7. \* @throws RuntimeException when open fails
8. \*/
9. **private** **static** **void** registerZygoteSocket() {
10. **if** (sServerSocket == **null**) {
11. **int** fileDesc;
12. **try** {
13. //从环境变量中获取文件句柄，这个socket接口是通过文件描述符来创建的，这个文件描符代表的就是我们前面说的/dev/socket/zygote文件了。这个文件描述符是通过环境变量ANDROID\_SOCKET\_ENV得到的，关于socket的创建及环境变量的设置请参考init进程源码
14. String env = System.getenv(ANDROID\_SOCKET\_ENV);
15. fileDesc = Integer.parseInt(env);
16. } **catch** (RuntimeException ex) {
17. **throw** **new** RuntimeException(ANDROID\_SOCKET\_ENV + " unset or invalid", ex);
18. }
20. **try** {
21. //创建服务端socket，该socket将监听并接受客户端请求
22. sServerSocket = **new** LocalServerSocket(createFileDescriptor(fileDesc));
23. } **catch** (IOException ex) {
24. **throw** **new** RuntimeException("Error binding to local socket '" + fileDesc + "'", ex);
25. }
26. }
27. }
29. ......
30. }

这个socket接口是通过文件描述符来创建的，这个文件描符代表的就是我们前面说的/dev/socket/zygote文件了。这个文件描述符是通过环境变量ANDROID\_SOCKET\_ENV得到的，它定义为：

1. **public** **class** ZygoteInit {
3. **private** **static** **final** String ANDROID\_SOCKET\_ENV = "ANDROID\_SOCKET\_zygote";
5. ......
6. }

ZygoteInit.registerZygoteSocket函数使用这个文件描述符来创建一个Java层的LocalServerSocket对象, 如果其它进程也需要打开这个/dev/socket/zygote文件来和Zygote进程进行通信，那就必须要通过文件名来连接这个LocalServerSocket了。ActivityManagerService是通过Process.start函数来创建一个新的进程的，而Process.start函数会首先通过Socket连接到Zygote进程中，最终由Zygote进程来完成创建新的应用程序进程，而Process类是通过openZygoteSocketIfNeeded函数来连接到Zygote进程中的Socket的，在第三章中将详细说明。

**Step 5. ZygoteInit.startSystemServe**

这个函数定义在frameworks/base/core/java/com/android/internal/os/ZygoteInit.java文件中：

1. **public** **class** ZygoteInit {
2. ......
4. **private** **static** **boolean** startSystemServer()
5. **throws** MethodAndArgsCaller, RuntimeException {
6. /\* Hardcoded command line to start the system server \*/
7. String args[] = {
8. "--setuid=1000",
9. "--setgid=1000",
10. "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,3001,3002,3003",
11. "--capabilities=130104352,130104352",
12. "--runtime-init",
13. "--nice-name=system\_server",
14. "com.android.server.SystemServer",
15. };
16. ZygoteConnection.Arguments parsedArgs = **null**;
18. **int** pid;
20. **try** {
21. parsedArgs = **new** ZygoteConnection.Arguments(args);
23. ......
25. /\* Request to fork the system server process \*/
26. pid = Zygote.forkSystemServer(
27. parsedArgs.uid, parsedArgs.gid,
28. parsedArgs.gids, debugFlags, **null**,
29. parsedArgs.permittedCapabilities,
30. parsedArgs.effectiveCapabilities);
31. } **catch** (IllegalArgumentException ex) {
32. ......
33. }
35. /\* For child process \*/
36. **if** (pid == 0) {
37. handleSystemServerProcess(parsedArgs);
38. }
40. **return** **true**;
41. }
43. ......
44. }

这里我们可以看到，Zygote进程通过Zygote.forkSystemServer函数来创建一个新的进程来启动SystemServer组件，返回值pid等于0的地方就是新的进程要执行的路径，即新创建的进程会执行handleSystemServerProcess函数。

由于由Zygote进程创建的子进程会继承Zygote进程在前面Step 4中创建的Socket文件描述符，而这里的子进程又不会用到它，因此，在handleSystemServerProcess函数中调用了closeServerSocket函数来关闭它，接着调用RuntimeInit.zygoteInit函数来进一步执行启动SystemServer组件的操作，一个是zygoteInitNative函数来执行一个Binder进程间通信机制的初始化工作，这个工作完成之后，这个进程中的Binder对象就可以方便地进行进程间通信了，另一个是调用上面Step 5传进来的com.android.server.SystemServer类的main函数。这整个过程涉及到的的都是SystemServer进程启动源码，在这里只做简单说明，不再贴代码。

**Step 6. ZygoteInit.runSelectLoopMode**

        这个函数定义在frameworks/base/core/java/com/android/internal/os/ZygoteInit.java文件中：

1. **private** **static** **void** runSelectLoopMode() **throws** MethodAndArgsCaller {
2. ArrayList<FileDescriptor> fds = **new** ArrayList(); //存储所有客户端连接的socket文件句柄
3. ArrayList<ZygoteConnection> peers = **new** ArrayList();//存储所有客户端连接
4. FileDescriptor[] fdArray = **new** FileDescriptor[4];
6. fds.add(sServerSocket.getFileDescriptor());//保存先前创建用于等待客户端连接的服务端socket
7. peers.add(**null**);
9. **int** loopCount = GC\_LOOP\_COUNT;
10. **while** (**true**) {
11. **int** index;
13. /\*
14. \* Call gc() before we block in select().
15. \* It's work that has to be done anyway, and it's better
16. \* to avoid making every child do it.  It will also
17. \* madvise() any free memory as a side-effect.
18. \*
19. \* Don't call it every time, because walking the entire
20. \* heap is a lot of overhead to free a few hundred bytes.
21. \*/
22. **if** (loopCount <= 0) {
23. gc();
24. loopCount = GC\_LOOP\_COUNT;
25. } **else** {
26. loopCount--;
27. }
29. **try** {
30. fdArray = fds.toArray(fdArray);
31. //selectReadable内部调用select，使用多路复用I/O模型
32. //当有客户端连接时，selectReadable返回
33. index = selectReadable(fdArray);
34. } **catch** (IOException ex) {
35. **throw** **new** RuntimeException("Error in select()", ex);
36. }
38. **if** (index < 0) {
39. **throw** **new** RuntimeException("Error in select()");
40. } **else** **if** (index == 0) {
41. //有一个客户端连接
42. ZygoteConnection newPeer = acceptCommandPeer();
43. peers.add(newPeer);
44. fds.add(newPeer.getFileDesciptor());
45. } **else** {
46. **boolean** done;
47. //客户端发送了请求，peers.get(index)获取当前客户端的ZygoteConnection，并调用当前连接的runOnce（）函数创建新的应用程序
48. done = peers.get(index).runOnce();
50. **if** (done) {
51. peers.remove(index);
52. fds.remove(index);
53. }
54. }
55. }
56. }

这个函数是在等待ActivityManagerService来连接这个Socket，然后调用ZygoteConnection.runOnce函数来创建新的应用程序，详细过程将在第三部分- Zygote进程启动Android应用程序过程详细说明。

**三、Zygote进程启动Android应用程序过程**

　 在Android系统中，应用程序是由Activity组成的，因此，应用程序的启动过程实际上就是应用程序中的默认Activity的启动过程。本章节以在手机屏幕中点击应用程序图标启动Android应用程序中的默认Activity为Android应用程序创建情景，启动一个新的应用程序。

在Android系统中，应用程序是由Launcher启动起来的，其实，Launcher本身也是一个应用程序，其它的应用程序安装后， Launcher的界面上就会出现一个相应的图标，点击这个图标时，Launcher就会对应的应用程序启动起来，Launcher进程将创建应用程序请求发送至ActivityManagerService的过程在这里不做详细说明，感兴趣可去<http://blog.csdn.net/luoshengyang/article/details/6689748>看一下分析过程。下面主要分析ActivityManagerService与Zygote进程通信的过程。

**Step 1. ActivityManagerService**

@/frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

1. **private** **final** **void** startProcessLocked(ProcessRecord app,
2. String hostingType, String hostingNameStr) {
3. ......
4. **try** {
5. ......
6. // Start the process.  It will either succeed and return a result containing
7. // the PID of the new process, or else throw a RuntimeException.
8. Process.ProcessStartResult startResult = Process.start("android.app.ActivityThread",
9. app.processName, uid, uid, gids, debugFlags, mountExternal,
10. app.info.targetSdkVersion, app.info.seinfo, **null**);
11. .......
12. }

  这里主要是调用Process.start接口来创建一个新的进程，新的进程会导入android.app.ActivityThread类，并且执行它的main函数，这就是为什么说每一个应用程序都有一个ActivityThread实例来对应的原因。

**Step 2. Process.start**

@/frameworks/base/core/java/android/os/Process.java

1. **public** **static** **final** ProcessStartResult start(**final** String processClass,
2. **final** String niceName,
3. **int** uid, **int** gid, **int**[] gids,
4. **int** debugFlags, **int** mountExternal,
5. **int** targetSdkVersion,
6. String seInfo,
7. String[] zygoteArgs) {
8. **try** {
9. **return** startViaZygote(processClass, niceName, uid, gid, gids,
10. debugFlags, mountExternal, targetSdkVersion, seInfo, zygoteArgs);
11. } **catch** (ZygoteStartFailedEx ex) {
12. Log.e(LOG\_TAG,
13. "Starting VM process through Zygote failed");
14. **throw** **new** RuntimeException(
15. "Starting VM process through Zygote failed", ex);
16. }
17. }

这里主要是调用startViaZygote来进一步按完成新进程的创建工作。

**Step 3. Process.startViaZygote**

@/frameworks/base/core/java/android/os/Process.java

1. **private** **static** ProcessStartResult startViaZygote(**final** String processClass,
2. **final** String niceName,
3. **final** **int** uid, **final** **int** gid,
4. **final** **int**[] gids,
5. **int** debugFlags, **int** mountExternal,
6. **int** targetSdkVersion,
7. String seInfo,
8. String[] extraArgs)
9. **throws** ZygoteStartFailedEx {
10. **synchronized**(Process.**class**) {
11. ArrayList<String> argsForZygote = **new** ArrayList<String>();
13. // --runtime-init, --setuid=, --setgid=,
14. // and --setgroups= must go first
15. argsForZygote.add("--runtime-init");
16. argsForZygote.add("--setuid=" + uid);
17. argsForZygote.add("--setgid=" + gid);
18. **if** ((debugFlags & Zygote.DEBUG\_ENABLE\_JNI\_LOGGING) != 0) {
19. argsForZygote.add("--enable-jni-logging");
20. }
21. **if** ((debugFlags & Zygote.DEBUG\_ENABLE\_SAFEMODE) != 0) {
22. argsForZygote.add("--enable-safemode");
23. }
24. **if** ((debugFlags & Zygote.DEBUG\_ENABLE\_DEBUGGER) != 0) {
25. argsForZygote.add("--enable-debugger");
26. }
27. **if** ((debugFlags & Zygote.DEBUG\_ENABLE\_CHECKJNI) != 0) {
28. argsForZygote.add("--enable-checkjni");
29. }
30. **if** ((debugFlags & Zygote.DEBUG\_ENABLE\_ASSERT) != 0) {
31. argsForZygote.add("--enable-assert");
32. }
33. **if** (mountExternal == Zygote.MOUNT\_EXTERNAL\_MULTIUSER) {
34. argsForZygote.add("--mount-external-multiuser");
35. } **else** **if** (mountExternal == Zygote.MOUNT\_EXTERNAL\_MULTIUSER\_ALL) {
36. argsForZygote.add("--mount-external-multiuser-all");
37. }
38. argsForZygote.add("--target-sdk-version=" + targetSdkVersion);
40. //TODO optionally enable debuger
41. //argsForZygote.add("--enable-debugger");
43. // --setgroups is a comma-separated list
44. **if** (gids != **null** && gids.length > 0) {
45. StringBuilder sb = **new** StringBuilder();
46. sb.append("--setgroups=");
48. **int** sz = gids.length;
49. **for** (**int** i = 0; i < sz; i++) {
50. **if** (i != 0) {
51. sb.append(',');
52. }
53. sb.append(gids[i]);
54. }
56. argsForZygote.add(sb.toString());
57. }
59. **if** (niceName != **null**) {
60. argsForZygote.add("--nice-name=" + niceName);
61. }
63. **if** (seInfo != **null**) {
64. argsForZygote.add("--seinfo=" + seInfo);
65. }
67. argsForZygote.add(processClass);
69. **if** (extraArgs != **null**) {
70. **for** (String arg : extraArgs) {
71. argsForZygote.add(arg);
72. }
73. }
75. **return** zygoteSendArgsAndGetResult(argsForZygote);
76. }
77. }

startViaZygote的主要工作就是处理传递到Zygote中的参数，将具体参数保存到内部成员变量argsForZygote 中，与Zygote通信通过zygoteSendArgsAndGetResult()方法完成。

**Step 4. Process.zygoteSendArgsAndGetResult**

1. **private** **static** ProcessStartResult zygoteSendArgsAndGetResult(ArrayList<String> args)
2. **throws** ZygoteStartFailedEx {
3. openZygoteSocketIfNeeded();//确保和Zygote通信的socket已被打开
5. **try** {
6. /\*\*
7. \* See com.android.internal.os.ZygoteInit.readArgumentList()
8. \* Presently the wire format to the zygote process is:
9. \* a) a count of arguments (argc, in essence)
10. \* b) a number of newline-separated argument strings equal to count
11. \*
12. \* After the zygote process reads these it will write the pid of
13. \* the child or -1 on failure, followed by boolean to
14. \* indicate whether a wrapper process was used.
15. \*/
17. sZygoteWriter.write(Integer.toString(args.size()));
18. sZygoteWriter.newLine();
20. **int** sz = args.size();
21. **for** (**int** i = 0; i < sz; i++) {//发送请求参数到Zygote
22. String arg = args.get(i);
23. **if** (arg.indexOf('\n') >= 0) {
24. **throw** **new** ZygoteStartFailedEx(
25. "embedded newlines not allowed");
26. }
27. sZygoteWriter.write(arg);
28. sZygoteWriter.newLine();
29. }
31. sZygoteWriter.flush();
33. // Should there be a timeout on this?
34. ProcessStartResult result = **new** ProcessStartResult();
35. result.pid = sZygoteInputStream.readInt();//Zygote处理完成会返回子进程的pid（即要创建的进程）
36. **if** (result.pid < 0) {
37. **throw** **new** ZygoteStartFailedEx("fork() failed");
38. }
39. result.usingWrapper = sZygoteInputStream.readBoolean();
40. **return** result;
41. } **catch** (IOException ex) {
42. **try** {
43. **if** (sZygoteSocket != **null**) {
44. sZygoteSocket.close();
45. }
46. } **catch** (IOException ex2) {
47. // we're going to fail anyway
48. Log.e(LOG\_TAG,"I/O exception on routine close", ex2);
49. }
51. sZygoteSocket = **null**;
53. **throw** **new** ZygoteStartFailedEx(ex);
54. }
55. }

到这里位置，客户端请求Zygote创建进程的请求就发送出去了，Zygote会返回进程的pid给客户端（ActivityMangerService）。由于ActivityMangerService在SystemServer进程中，所以这里即SystemServer进程通过socket向Zygote发送了信息。

接下来看一下Zygote处理客户端请求的过程。

**Step 5.ZygoteInit.runSelectLoopMode**

1. /\*\*
2. \* Runs the zygote process's select loop. Accepts new connections as
3. \* they happen, and reads commands from connections one spawn-request's
4. \* worth at a time.
5. \*
6. \* @throws MethodAndArgsCaller in a child process when a main() should
7. \* be executed.
8. \*/
9. **private** **static** **void** runSelectLoopMode() **throws** MethodAndArgsCaller {
10. ArrayList<FileDescriptor> fds = new ArrayList();
11. ArrayList <ZygoteConnection> peers = new ArrayList();
12. FileDescriptro[] fdArray = new FileDescriptor[4];
13. ......
14. **while** (**true**) {//死循环
15. ......
17. **if** (index < 0) {
18. **throw** **new** RuntimeException("Error in select()");
19. } **else** **if** (index == 0) {//index==0表示selcet接收到的是Zygote的socket的事件
20. ZygoteConnection newPeer = acceptCommandPeer();
21. peers.add(newPeer);
22. fds.add(newPeer.getFileDesciptor());
23. } **else** {//调用ZygoteConnection对象的runOnce方法，ZygoteConnection是在index == 0时被添加到peers的
24. **boolean** done;
25. done = peers.get(index).runOnce();
27. **if** (done) {
28. peers.remove(index);
29. fds.remove(index);
30. }
31. }
32. }
33. }

Zygote进程初始化完后会一直等待客户端的socket连接请求，阻塞在acceptCommandPeer方法处。

1. /\*\*
2. \* Waits for and accepts a single command connection. Throws
3. \* RuntimeException on failure.
4. \*/
5. **private static** ZygoteConnection acceptCommandPeer() {
6. **try {**
7. **return new** ZygoteConnection(sServerSocket.accept());
8. **} catch (**IOException ex**) {**
9. **throw new** RuntimeException("IOException during accept()", ex);
10. **}**
11. **}**

当客户端的socket连接请求到来时，将ZygoteConnection添加到内部成员变量peers，并运行runOnce方法。

**Step 6.****ZygoteConnection.runOnce**

@/frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java

1. /\*\*
2. \* Reads one start command from the command socket. If successful,
3. \* a child is forked and a {@link ZygoteInit.MethodAndArgsCaller}
4. \* exception is thrown in that child while in the parent process,
5. \*/
6. **boolean** runOnce() **throws** ZygoteInit.MethodAndArgsCaller {
8. String args[];
9. Arguments parsedArgs = **null**;
10. FileDescriptor[] descriptors;
12. **try** {
13. args = readArgumentList();//读取客户端发送过来的参数
14. descriptors = mSocket.getAncillaryFileDescriptors();
15. } **catch** (IOException ex) {
16. Log.w(TAG, "IOException on command socket " + ex.getMessage());
17. closeSocket();
18. **return** **true**;
19. }
20. ......
21. //fork一个新进程
22. pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid, parsedArgs.gids,parsedArgs.debugFlags, rlimits, parsedArgs.mountExternal, parsedArgs.seInfo,parsedArgs.niceName);
24. **try** {
25. **if** (pid == 0) {//子进程
26. // in child
27. IoUtils.closeQuietly(serverPipeFd);
28. serverPipeFd = **null**;
29. handleChildProc(parsedArgs, descriptors, childPipeFd, newStderr);
30. **return** **true**;
31. } **else** {//父进程
32. // in parent...pid of < 0 means failure
33. IoUtils.closeQuietly(childPipeFd);
34. childPipeFd = **null**;
35. **return** handleParentProc(pid, descriptors, serverPipeFd, parsedArgs);
36. }
37. ......
38. }

runOnce方法主要工作是读取客户端发送的参数，调用forkAndSpecialize方法创建子进程，此方法会调用Zygote类内部的nativeForkAndSpecialize完成新进程的创建工作。在创建进程时，需要返回两次，在子进程中返回0，在父进程中返回子进程的进程ID。

**Step 7. ZygoteConnection.handleChildProc**

@/frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java

1. /\*\*
2. \* Handles post-fork setup of child proc, closing sockets as appropriate,
3. \* reopen stdio as appropriate, and ultimately throwing MethodAndArgsCaller
4. \* if successful or returning if failed.
5. \*/
6. **private** **void** handleChildProc(Arguments parsedArgs,
7. FileDescriptor[] descriptors, FileDescriptor pipeFd, PrintStream newStderr)   **throws** ZygoteInit.MethodAndArgsCaller {
9. closeSocket();//关闭子进程中，从Zygote fork过来的服务端socket
10. ZygoteInit.closeServerSocket();
11. .....
12. **if** (parsedArgs.niceName != **null**) {
13. Process.setArgV0(parsedArgs.niceName);
14. }
15. **if** (parsedArgs.runtimeInit) {//从startViaZygote可知传入了--runtime-init参数，所以这里为true
16. **if** (parsedArgs.invokeWith != **null**) {//没有传入--invoke-with，所以这里走的是else的逻辑
17. WrapperInit.execApplication(parsedArgs.invokeWith,
18. parsedArgs.niceName, parsedArgs.targetSdkVersion,
19. pipeFd, parsedArgs.remainingArgs);
20. } **else** {
21. RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion,
22. parsedArgs.remainingArgs);
23. }
24. } **else** {
25. ......
26. }
27. }

handleChildProc方法主要是在子进程空间中，继续完成进程的创建工作。首先调用closeSocket方法关闭从zygote进程fork过来的server socket，通过解析从startViaZygote传递的参数，从而执行不同的创建过程。在这里，执行的是RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.remainingArgs)。

**Step 8. RuntimeInit.zygoteInit**

@/frameworks/base/core/java/com/android/internal/os/RuntimeInit.java

1. **public** **static** **final** **void** zygoteInit(**int** targetSdkVersion, String[] argv)
2. **throws** ZygoteInit.MethodAndArgsCaller {
3. **if** (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application from zygote");
4. redirectLogStreams();//将System.out 和 System.err 输出重定向到Android 的Log系统
5. /\* 初始化了一些系统属性，其中最重要的一点就是设置了一个未捕捉异常的handler， 当代码有任何未知异常，就会执行它 \*/
6. commonInit();
7. /\* 最终会调用app\_main的onZygoteInit函数 这里的作用是在新进程中引入Binder，也就说通过nativeZygoteInit以后，新的进程就可以使用Binder进程通信了 \*/
8. nativeZygoteInit();
10. applicationInit(targetSdkVersion, argv);//应用初始化
11. }

在zygoteInit方法中，主要工作是完成进程的初始化：将System.out与System.err输出重定向到Log系统，在Android应用程序代码调试过程中的作用非常显著；初始化系统属性与Binder通信的线程池，使得应用程序拥有Android运行时环境；调用applicationInit方法完成应用初始化。

1. **private** **static** **void** applicationInit(**int** targetSdkVersion, String[] argv)
2. **throws** ZygoteInit.MethodAndArgsCaller {
3. // If the application calls System.exit(), terminate the process
4. // immediately without running any shutdown hooks.  It is not possible to
5. // shutdown an Android application gracefully.  Among other things, the
6. // Android runtime shutdown hooks close the Binder driver, which can cause
7. // leftover running threads to crash before the process actually exits.
8. nativeSetExitWithoutCleanup(**true**);
10. // We want to be fairly aggressive about heap utilization, to avoid
11. // holding on to a lot of memory that isn't needed.
12. VMRuntime.getRuntime().setTargetHeapUtilization(0.75f);
13. VMRuntime.getRuntime().setTargetSdkVersion(targetSdkVersion);
15. **final** Arguments args;
16. **try** {
17. args = **new** Arguments(argv);
18. } **catch** (IllegalArgumentException ex) {
19. Slog.e(TAG, ex.getMessage());
20. // let the process exit
21. **return**;
22. }
24. // Remaining arguments are passed to the start class's static main
25. invokeStaticMain(args.startClass, args.startArgs);
26. }

在applicationInit方法中，同样完成进程的部分配置工作，具体工作在源码注释中都有解释，最重要的一点是调用invokeStaticMain方法，启动ActivityThread类的main方法。

1. **private** **static** **void** invokeStaticMain(String className, String[] argv)
2. **throws** ZygoteInit.MethodAndArgsCaller {
3. Class<?> cl;
5. **try** {
6. cl = Class.forName(className);
7. } **catch** (ClassNotFoundException ex) {
8. **throw** **new** RuntimeException(
9. "Missing class when invoking static main " + className,
10. ex);
11. }
13. Method m;
14. **try** {
15. m = cl.getMethod("main", **new** Class[] { String[].**class** });
16. } **catch** (NoSuchMethodException ex) {
17. **throw** **new** RuntimeException(
18. "Missing static main on " + className, ex);
19. } **catch** (SecurityException ex) {
20. **throw** **new** RuntimeException(
21. "Problem getting static main on " + className, ex);
22. }
24. **int** modifiers = m.getModifiers();
25. **if** (! (Modifier.isStatic(modifiers) && Modifier.isPublic(modifiers))) {
26. **throw** **new** RuntimeException(
27. "Main method is not public and static on " + className);
28. }
30. /\*
31. \* This throw gets caught in ZygoteInit.main(), which responds
32. \* by invoking the exception's run() method. This arrangement
33. \* clears up all the stack frames that were required in setting
34. \* up the process.
35. \*/
36. **throw** **new** ZygoteInit.MethodAndArgsCaller(m, argv);
37. }

invokeStaticMain方法本身就是调用ActivityThread类的main()，但与常规的反射有较大不同，而是通过抛出异常的方式调用ActivityThread的main()方法，如在源码中解释的一样，抛出的Zygote.MethodAndArgsCaller异常会在ZygoteInit.main()中捕获。这种方式可在主线程开启的时候，有效的清理进程在创建过程中的堆栈，也会让开发人员感觉到Android应用程序的入口就是ActivityThread.main()函数。

**Step 9 ZygoteInit.** **MethodAndArgsCaller**

@/frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

1. /\*\*
2. \* Helper exception class which holds a method and arguments and
3. \* can call them. This is used as part of a trampoline to get rid of
4. \* the initial process setup stack frames.
5. \*/
6. **public** **static** **class** MethodAndArgsCaller **extends** Exception
7. **implements** Runnable {
8. /\*\* method to call \*/
9. **private** **final** Method mMethod;
11. /\*\* argument array \*/
12. **private** **final** String[] mArgs;
14. **public** MethodAndArgsCaller(Method method, String[] args) {
15. mMethod = method;
16. mArgs = args;
17. }
19. **public** **void** run() {
20. **try** {
21. mMethod.invoke(**null**, **new** Object[] { mArgs });
22. } **catch** (IllegalAccessException ex) {
23. **throw** **new** RuntimeException(ex);
24. } **catch** (InvocationTargetException ex) {
25. Throwable cause = ex.getCause();
26. **if** (cause **instanceof** RuntimeException) {
27. **throw** (RuntimeException) cause;
28. } **else** **if** (cause **instanceof** Error) {
29. **throw** (Error) cause;
30. }
31. **throw** **new** RuntimeException(ex);
32. }
33. }
34. }

MethodAndArgsCaller方法的工作就是启动ActivityThread类的main方法，之后就进入消息循环了，在进程中启动Activity或者Service等。至此，Zygote进程启动应用程序的过程分析完成。

**四、zygote进程的注入**

由上述分析可知，zygote进程在启动完成之后，会进入一个死循环。在这个死循环中，zygote进程会在一个名称为“zygote”的的socket上等待ActivityManagerService向他发送启动新的应用程序进程的请求。代码可见第二节的Step 6，其中int型变量index表示此时与zygote通信的socket描述符的个数。index=0表示没有socket连接，此时zygote进程调用acceptCommandPeer()函数（源码见下），该函数的作用是等待一个连接命令并返回一个ZygoteConnection对象，也就是说，此时zygote进程进入了阻塞状态。当zygote进程接收到一个创建新的应用程序进程的请求后，就会调用ZygoteConnection类的成员函数runOnce来处理这个请求。

1. /\*
2. \* Waits for and accepts a single command connection. Throws
3. \* RuntimeException on failure.
4. \*/
5. **private static** ZygoteConnection acceptCommandPeer() {
6. **try {**
7. **return new** ZygoteConnection(sServerSocket.accept());
8. **} catch (**IOException ex**) {**
9. **throw new** RuntimeException("IOException during accept()", ex);
10. **}**
11. **}**

注入程序添加了一段对zygote进程的特殊处理connect\_to\_zygote，用于解除在注入过程中Zygote进程的阻塞状态，connect\_to\_zygote函数的作用是与zygote进程建立socket连接，然后关闭连接，其间没有传输任何实质性的内容。

1. //注入程序中对Zygote进程特殊处理的函数，用于解除注入过程阻塞问题
2. **static void\*** connect\_to\_zygote(**void\*** arg){
3. **int** s, len;
4. **struct** sockaddr\_un remote; //声明结构体
5. if ((s = socket(AF\_UNIX, SOCK\_STREAM, 0)) != -1) { //s为socket描述符
6. remote.sun\_family = AF\_UNIX,;
7. strcpy(remote.sun\_path, "/dev/socket/zygote");
8. len = strlen(remote.sun\_path) + sizeof(remote.sun\_family);
9. LOGI("[+] start to connect zygote socket");
10. connect(s, (struct sockaddr \*) &remote, len);//第二个参数为指向sockaddr结构体（即remote）的指针
11. LOGI("[+] close socket");
12. close(s);
13. }
15. **return** NULL ;
16. }

其中，sockaddr\_un是用于unix域进程通信的描述socket通信地址的结构体，其中包含sun\_family和 sun\_path两个成员变量。socket函数返回一个socket描述符s。connect函数的作用是建立socket连接，其中第一个参数为socket描述符，第二个参数为指向sockaddr结构体（即remote）的指针。close函数的作用为关闭socket连接。

表面上来看connect\_to\_zygote函数只是建立连接又关闭连接，没有做什么实质性的工作，可事实上它非常重要。由上一部分可知，zygote进程当没有收到创建新的应用程序进程的请求时，是处于阻塞状态的。此时如果我们对zygote进程进行注入，注入过程会完全阻塞住，当创建新进程时，整个注入过程才会完整的执行成功，这是因为在注入过程中，注入程序需要操作目标进程的运行，有很多waitpid的操作，如果zygote进程阻塞，则注入程序会阻塞在waitpid处。而connect\_to\_zygote函数通过与zygote进程建立socket连接，可以解除zygote进程的阻塞状态，注入程序中的waitpid操作就会有返回并顺利执行，因此，加入connect\_to\_zygote函数后，注入程序才能够跑通，最终完成注入。