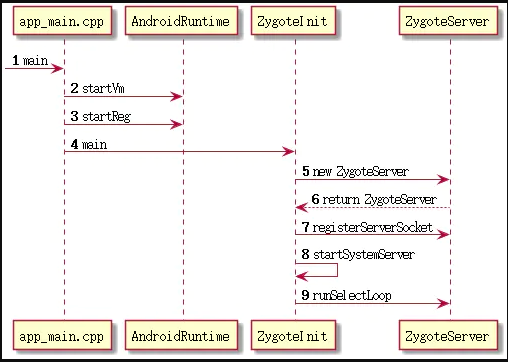
# 一. 简介

zygote是受精卵的意思，它是Android中的一个非常重要的守护进程服务(Daem Service),所有的其他Dalvik虚拟机进程都是通过zygote孵化（fork）出来的。Android应用程序是由Java语言编写的，运行在各自独立的Dalvik虚拟机中。如果每个应用程序在启动之时都需要单独运行和初始化一个虚拟机，会大大降低系统性能，**因此Android首先创建一个zygote虚拟机，然后通过它孵化出其他的虚拟机进程，进而共享虚拟机内存和框架层资源，这样大幅度提高应用程序的启动和运行速度。**

Zygote是Android中最重要的一个进程，和Init进程，SystemServer进程是支撑Android世界的三极。Zygote进程在Init进程中以service的方式启动的。

# 二. 启动流程



ZygoteInit类负责Zygote进程Java的初始化工作，首先来看下ZygoteInit类的入口main()方法：

```csharp

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

public static void main(String argv[]) {

//1、创建ZygoteServer

ZygoteServer zygoteServer = new ZygoteServer();

.......

try {

.......

boolean startSystemServer = false;

String socketName = "zygote";

String abiList = null;

boolean enableLazyPreload = false;

// 2、解析app\_main.cpp传来的参数

for (int i = 1; i < argv.length; i++) {

if ("start-system-server".equals(argv[i])) {

startSystemServer = true;

} else if ("--enable-lazy-preload".equals(argv[i])) {

enableLazyPreload = true;

} else if (argv[i].startsWith(ABI\_LIST\_ARG)) {

abiList = argv[i].substring(ABI\_LIST\_ARG.length());

} else if (argv[i].startsWith(SOCKET\_NAME\_ARG)) {

socketName = argv[i].substring(SOCKET\_NAME\_ARG.length());

} else {

throw new RuntimeException("Unknown command line argument: " + argv[i]);

}

}

if (abiList == null) {

throw new RuntimeException("No ABI list supplied.");

}

//3、创建一个Server端的Socket

zygoteServer.registerServerSocket(socketName);

// In some configurations, we avoid preloading resources and classes eagerly.

// In such cases, we will preload things prior to our first fork.

if (!enableLazyPreload) {

bootTimingsTraceLog.traceBegin("ZygotePreload");

EventLog.writeEvent(LOG\_BOOT\_PROGRESS\_PRELOAD\_START,

SystemClock.uptimeMillis());

//4、加载进程的资源和类

preload(bootTimingsTraceLog);

EventLog.writeEvent(LOG\_BOOT\_PROGRESS\_PRELOAD\_END,

SystemClock.uptimeMillis());

bootTimingsTraceLog.traceEnd(); // ZygotePreload

} else {

Zygote.resetNicePriority();

}

........

if (startSystemServer) {

//5、开启SystemServer进程，这是受精卵进程的第一次分裂

startSystemServer(abiList, socketName, zygoteServer);

}

Log.i(TAG, "Accepting command socket connections");

//6、启动一个死循环监听来自Client端的消息

zygoteServer.runSelectLoop(abiList);

//7、关闭SystemServer的Socket

zygoteServer.closeServerSocket();

} catch (Zygote.MethodAndArgsCaller caller) {

//8、这里捕获这个异常调用MethodAndArgsCaller的run方法。

caller.run();

} catch (Throwable ex) {

Log.e(TAG, "System zygote died with exception", ex);

zygoteServer.closeServerSocket();

throw ex;

}

}

```

做了一下几件事情:

一，创建ZygoteServer，在Android O上把与Socket的操作都封装到了ZygoteServer类中；

二，解析app\_main.cpp传来的参数。

三，创建一个Server端的Socket，作用是当Zygote进程将SystemServer进程启动后，就会在这个Socket上来等待ActivityManagerService请求，即请求创建我们自己APP应用程序进程；

四，预加载类和资源，包括颜色啊，R文件，drawable、类等；

五，启动system\_server进程，这是上层framework的运行载体，ActivityManagerService就是运行在这个进程里面的；

六，开启一个循环，等待着接收ActivityManagerService的请求，随时待命，当接收到创建新进程的请求时立即唤醒并执行相应工作；

主要做了以下几件事：1）注册zygote的socket端口监听；2）预加载系统资源；3）启动SystemServer进程；4）进入监听和接受消息的循环；依次看下这四个方法：

我觉得**这段代码的主线**是，\*\*ZygoteInit进程启动后，会注册一个Socket，在runSelectLoop方法中开启一个while死循环等待ActivityManagerService创建新进程的请求，其次，ZygoteInit启动了SystemServer进程，执行SystemServer的main方法。\*\*

这种模式其实可以理解成一个模板格式的代码，不信你在看看WebViewZygoteInit中的写法和ZygoteInit的写法是不是如出一辙呢？

```csharp

frameworks/base/core/java/com/android/internal/os/WebViewZygoteInit.java

public static void main(String argv[]) {

sServer = new WebViewZygoteServer();

// Zygote goes into its own process group.

try {

Os.setpgid(0, 0);

} catch (ErrnoException ex) {

throw new RuntimeException("Failed to setpgid(0,0)", ex);

}

try {

sServer.registerServerSocket("webview\_zygote");

sServer.runSelectLoop(TextUtils.join(",", Build.SUPPORTED\_ABIS));

sServer.closeServerSocket();

} catch (Zygote.MethodAndArgsCaller caller) {

caller.run();

} catch (RuntimeException e) {

Log.e(TAG, "Fatal exception:", e);

}

System.exit(0);

}

```

以下是runtime的start方法

```php

/\*

\* Start the Android runtime. This involves starting the virtual machine

\* and calling the "static void main(String[] args)" method in the class

\* named by "className".

\*

\* Passes the main function two arguments, the class name and the specified

\* options string.

\*/

void AndroidRuntime::start(const char\* className, const Vector<String8>& options, bool zygote)

{

ALOGD(">>>>>> START %s uid %d <<<<<<\n",

className != NULL ? className : "(unknown)", getuid());

/\* start the virtual machine \*/

JniInvocation jni\_invocation;

jni\_invocation.Init(NULL);

JNIEnv\* env;

if (startVm(&mJavaVM, &env, zygote) != 0) {

return;

}

onVmCreated(env);

/\*

\* Register android functions.

\*/

if (startReg(env) < 0) {

ALOGE("Unable to register all android natives\n");

return;

}

/\*

\* We want to call main() with a String array with arguments in it.

\* At present we have two arguments, the class name and an option string.

\* Create an array to hold them.

\*/

jclass stringClass;

jobjectArray strArray;

jstring classNameStr;

stringClass = env->FindClass("java/lang/String");

assert(stringClass != NULL);

strArray = env->NewObjectArray(options.size() + 1, stringClass, NULL);

assert(strArray != NULL);

classNameStr = env->NewStringUTF(className);

assert(classNameStr != NULL);

env->SetObjectArrayElement(strArray, 0, classNameStr);

for (size\_t i = 0; i < options.size(); ++i) {

jstring optionsStr = env->NewStringUTF(options.itemAt(i).string());

assert(optionsStr != NULL);

env->SetObjectArrayElement(strArray, i + 1, optionsStr);

}

/\*

\* Start VM. This thread becomes the main thread of the VM, and will

\* not return until the VM exits.

\*/

char\* slashClassName = toSlashClassName(className);

jclass startClass = env->FindClass(slashClassName);

if (startClass == NULL) {

ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);

/\* keep going \*/

} else {

jmethodID startMeth = env->GetStaticMethodID(startClass, "main",

"([Ljava/lang/String;)V");

if (startMeth == NULL) {

ALOGE("JavaVM unable to find main() in '%s'\n", className);

/\* keep going \*/

} else {

env->CallStaticVoidMethod(startClass, startMeth, strArray);

#if 0

if (env->ExceptionCheck())

threadExitUncaughtException(env);

#endif

}

}

free(slashClassName);

//这行Log比较常见，因为其他应用进程也是由zygote 进程fork 出来的，所有其他进程也包含这段代码，如果其他进程在java 层crash，那么也会走到这里

ALOGD("Shutting down VM\n");

if (mJavaVM->DetachCurrentThread() != JNI\_OK)

ALOGW("Warning: unable to detach main thread\n");

if (mJavaVM->DestroyJavaVM() != 0)

ALOGW("Warning: VM did not shut down cleanly\n");

}

```

代码很简单，主要做了三件事情，一调用startVm开启虚拟机，二调用startReg注册JNI方法，三就是使用JNI把Zygote进程启动起来。

```cpp

996 ALOGD(">>>>>> START %s uid %d <<<<<<\n",

997 className != NULL ? className : "(unknown)", getuid());

```

这个是进入Zygote进程的重要依据，开机的时候一般都会打印这一行Log。如

```css

07-09 14:40:37.788 16504 16504 D AndroidRuntime: >>>>>> START com.android.internal.os.ZygoteInit uid 0 <<<<<<

```

如果遇到不能开机的情况，这行Log没有打开，极有可能不是上层的问题。

## 2.1 registerZygoteSocket()方法

```java

private static void registerZygoteSocket(String socketName) {

if (sServerSocket == null) {

int fileDesc;

final String fullSocketName = ANDROID\_SOCKET\_PREFIX + socketName;

try {

String env = System.getenv(fullSocketName);

fileDesc = Integer.parseInt(env);

}

try {

FileDescriptor fd = new FileDescriptor();

fd.setInt$(fileDesc);

sServerSocket = new LocalServerSocket(fd);

}

}

}

12345678910111213141516

```

创建一个本地的socket,然后等待调用runSelectLoop()来进入等待socket等待连接的循环中；

## 2.2 预加载系统资源，preload()方法；

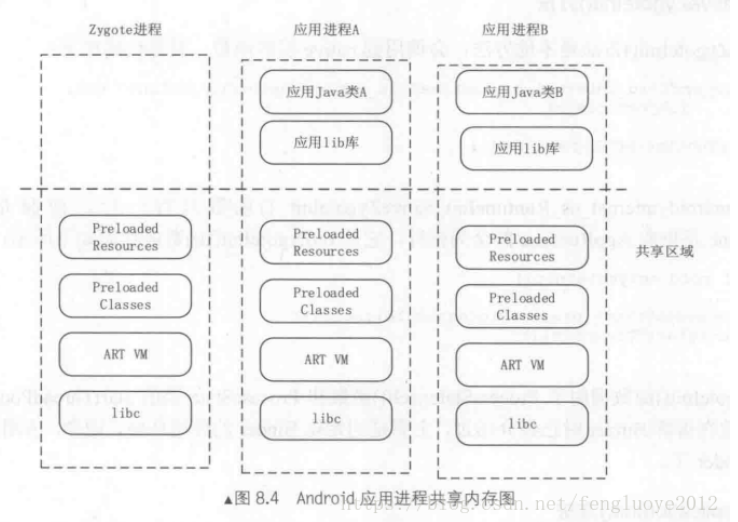
- 何为预加载？

android系统资源加载分两种方式，预加载和使用进程中加载。 预加载是指在zygote进程启动的时候就加载，这样系统只在zygote执行一次加载操作，所有APP用到该资源不需要再重新加载，减少资源加载时间，加快了应用启动速度，一般情况下，系统中App共享的资源会被列为预加载资源。

- 预加载是什么原理？

预加载的原理很简单，就是在zygote进程启动后将资源读取出来，保存到Resources一个全局静态变量中，下次读取系统资源的时候优先从静态变量中查找。主要代码在zygoteInit.java类中方法preloadResources()

先来看下Android应用进程共享内存图



通过上图可以很容易理解在Zygote进程预加载系统资源后，然后通过它孵化出其他的虚拟机进程，进而共享虚拟机内存和框架层资源，这样大幅度提高应用程序的启动和运行速度。 现在来看下preload()方法;

```java

static void preload() {

Log.d(TAG, "begin preload");

Trace.traceBegin(Trace.TRACE\_TAG\_DALVIK, "BeginIcuCachePinning");

beginIcuCachePinning();

Trace.traceEnd(Trace.TRACE\_TAG\_DALVIK);

Trace.traceBegin(Trace.TRACE\_TAG\_DALVIK, "PreloadClasses");

//2.1)预加载系统类

preloadClasses();

Trace.traceEnd(Trace.TRACE\_TAG\_DALVIK);

Trace.traceBegin(Trace.TRACE\_TAG\_DALVIK, "PreloadResources");

//2.2）预加载系统资源

preloadResources();

Trace.traceEnd(Trace.TRACE\_TAG\_DALVIK);

Trace.traceBegin(Trace.TRACE\_TAG\_DALVIK, "PreloadOpenGL");

//预加载OpenGL资源

preloadOpenGL();

Trace.traceEnd(Trace.TRACE\_TAG\_DALVIK);

//2.3）预加载共享的so库

preloadSharedLibraries();

preloadTextResources();

// Ask the WebViewFactory to do any initialization that must run in the zygote process,

// for memory sharing purposes.

//预加载WebView资源库

WebViewFactory.prepareWebViewInZygote();

endIcuCachePinning();

warmUpJcaProviders();

Log.d(TAG, "end preload");

}

```

主要是预加载各种系统资源，主要看下2.1）预加载系统类；2.2）预加载系统资源；2.3）预加载so库

### 2.2.1 preloadClasses();加载系统类

frameworks/base/preloaded-classes 文本文件里。大概有四千多个

```java

private static final String PRELOADED\_CLASSES = "/system/etc/preloaded-classes";

private static void preloadClasses() {

final VMRuntime runtime = VMRuntime.getRuntime();

InputStream is;

try {

is = new FileInputStream(PRELOADED\_CLASSES);

} catch (FileNotFoundException e) {

Log.e(TAG, "Couldn't find " + PRELOADED\_CLASSES + ".");

return;

}

....

float defaultUtilization = runtime.getTargetHeapUtilization();

runtime.setTargetHeapUtilization(0.8f);

try {

BufferedReader br= new BufferedReader(new InputStreamReader(is), 256);

int count = 0;

String line;

while ((line = br.readLine()) != null) {

// Skip comments and blank lines.

line = line.trim();

if (line.startsWith("#") || line.equals("")) {

continue;

}

......

try {

//装载Java类信息

Class.forName(line, true, null);

count++;

} catch (ClassNotFoundException e) {}

.....

} finally {

Trace.traceBegin(Trace.TRACE\_TAG\_DALVIK, "PreloadDexCaches");

runtime.preloadDexCaches();

Trace.traceEnd(Trace.TRACE\_TAG\_DALVIK);

}

}

```

去读PRELOADED\_CLASSES文件下的文件，得到InputStream对象，在转换为BufferedReader，逐行读取文件的内容，每行通过trim(),过滤掉空行，然后调用 Class.forName()方法，加载Java类信息，而不是创建一个对象；

### 2.2.2 preloadResources();预加载系统资源

系统中有大量的资源可以直接被App所使用，比如一个颜色，一个drawble，这些都是通过preloadResources加载的。

```java

private static void preloadResources() {

final VMRuntime runtime = VMRuntime.getRuntime();

try {

mResources = Resources.getSystem();

mResources.startPreloading();

if (PRELOAD\_RESOURCES) {

//加载系统Drawable资源

TypedArray ar = mResources.obtainTypedArray(com.android.internal.R.array.preloaded\_drawables);

int N = preloadDrawables(ar);

ar.recycle();

//加载系统颜色资源

ar = mResources.obtainTypedArray(com.android.internal.R.array.preloaded\_color\_state\_lists);

N = preloadColorStateLists(ar);

ar.recycle();

}

mResources.finishPreloading();

} catch (RuntimeException e) {

}

}

```

加载资源的位置:

```xml

/frameworks/base/core/res/res/values/arrays.xml

<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">

<!-- Do not translate. These are all of the drawable resources that should be preloaded by

the zygote process before it starts forking application processes. -->

<array name="preloaded\_drawables">

<item>@drawable/ab\_share\_pack\_material</item>

<item>@drawable/ab\_solid\_shadow\_material</item>

<item>@drawable/action\_bar\_item\_background\_material</item>

<item>@drawable/activated\_background\_material</item>

......

<item>@drawable/toast\_frame</item>

</array>

<!-- Do not translate. These are all of the color state list resources that should be

preloaded by the zygote process before it starts forking application processes. -->

<array name="preloaded\_color\_state\_lists">

<item>@color/primary\_text\_dark</item>

<item>@color/primary\_text\_dark\_disable\_only</item>

<item>@color/primary\_text\_dark\_nodisable</item>

.......

<item>@color/search\_url\_text\_material\_light</item>

</array>

<array name="preloaded\_freeform\_multi\_window\_drawables">

<item>@drawable/decor\_maximize\_button\_dark</item>

<item>@drawable/decor\_maximize\_button\_light</item>

</array>

```

### 2.2.3 preloadSharedLibraries();加载系统共享so库

```java

private static void preloadSharedLibraries() {

Log.i(TAG, "Preloading shared libraries...");

System.loadLibrary("android");

System.loadLibrary("compiler\_rt");

System.loadLibrary("jnigraphics");

}

```

对于preloadOpenGL、preloadSharedLibraries和preloadTextResources在此不一一分析了，原来ZygoteInit中的preload方法加载了这么多资源，这个也就是为什么开机慢而打开一个应用快的原因之一。我们能不能把加载资源的这些耗时操作放到子线程中做呢？

```cpp

public static void main(String argv[]) {

ZygoteServer zygoteServer = new ZygoteServer();

// Mark zygote start. This ensures that thread creation will throw

// an error.

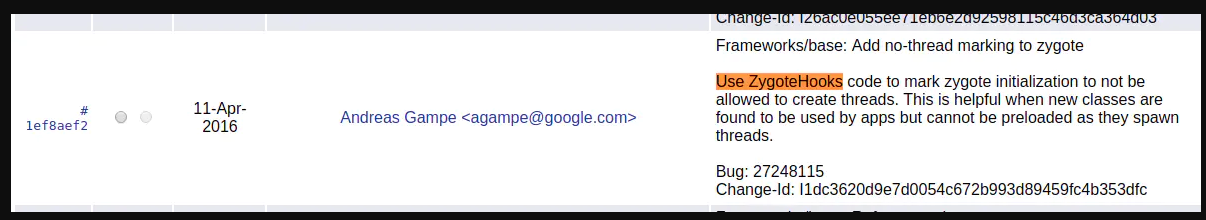
ZygoteHooks.startZygoteNoThreadCreation();

.......

ZygoteHooks.stopZygoteNoThreadCreation();

}

```



看到为防止多线程带来的同步问题，google这个地方禁止开启多线程。故网上有很多说法，尝试把这些代码放到子线程中去做应该是不对的(Android O)。

## 3 startSystemServer() 启动SystemServer进程

```java

private static boolean startSystemServer(String abiList, String socketName)

throws MethodAndArgsCaller, RuntimeException {

....

//3.1 为启动SystemServer进程准备参数

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 {

parsedArgs = new ZygoteConnection.Arguments(args);

ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);

ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);

//3.2 fork出SystemServer进程

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

pid = Zygote.forkSystemServer(

parsedArgs.uid, parsedArgs.gid,

parsedArgs.gids,

parsedArgs.debugFlags,

null,

parsedArgs.permittedCapabilities,

parsedArgs.effectiveCapabilities);

} catch (IllegalArgumentException ex) {

throw new RuntimeException(ex);

}

/\* For child process \*/

if (pid == 0) {

if (hasSecondZygote(abiList)) {

waitForSecondaryZygote(socketName);

}

//3.3 fork出SystemServer进程之后，初始化SystemServer进程

handleSystemServerProcess(parsedArgs);

}

return true;

}

```

主要做了三件事，3.1）为启动SystemServer进程准备参数，可以看到SystemServer的进程Id和组Id均为1000，SystemServer的执行类是com.android.server.SystemServer；3.2）fork出SystemServer进程；3.3）fork出SystemServer进程之后，初始化SystemServer进程；

看下handleSystemServerProcess()方法

```java

private static void handleSystemServerProcess( ZygoteConnection.Arguments parsedArgs)

throws ZygoteInit.MethodAndArgsCaller {

//关闭zygote的socket

closeServerSocket();

//设置umask为0077；只有SystemServer进程可以访问；

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

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

//由3.1可以看出nice-name=system\_server，设置进程的名称为system\_server；

if (parsedArgs.niceName != null) {

Process.setArgV0(parsedArgs.niceName);

}

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

if (systemServerClasspath != null) {

performSystemServerDexOpt(systemServerClasspath);

}

//由3.1可以看出invokeWith为null;

if (parsedArgs.invokeWith != null) {

} else {

ClassLoader cl = null;

if (systemServerClasspath != null) {

cl = createSystemServerClassLoader(systemServerClasspath,

parsedArgs.targetSdkVersion);

Thread.currentThread().setContextClassLoader(cl);

}

RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.remainingArgs, cl);

}

}

```

## 4 runSelectLoop()方法

```java

private static void runSelectLoop(String abiList) throws MethodAndArgsCaller {

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

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

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;

}

try {

Os.poll(pollFds, -1);

} catch (ErrnoException ex) {

throw new RuntimeException("poll failed", ex);

}

for (int i = pollFds.length - 1; i >= 0; --i) {

if ((pollFds[i].revents & POLLIN) == 0) {

continue;

}

//4.1）接受连接请求

if (i == 0) {

ZygoteConnection newPeer = acceptCommandPeer(abiList);

peers.add(newPeer);

fds.add(newPeer.getFileDesciptor());

//4.2) 接受消息

} else {

boolean done = peers.get(i).runOnce();

if (done) {

peers.remove(i);

fds.remove(i);

}

}

}

}

}

```

主要做了两件事4.1）接受连接请求；i=0,说明请求连接的事件过来了，调用acceptCommandPeer()和客户端建立socket连接，然后加入监听数组，等待这个socket上命令的到来；4.2）接受消息；i>0 说明已经连接上的socket已经有数据到了，调用ZygoteConnection类的runOnce()方法处理完成后，会断开和客户端的连接，并且从监听数组中移除；

以上就是Zygote进程的启动流程和在main()方法中主要做的四件事的解析，如有问题，请多指教，谢谢！

Android FrameWork的文章现在有很多，相关的书籍也有不少，都写的很通俗易懂，我写相关的文章主要是为了记录在学习FrameWork过程中的点滴。