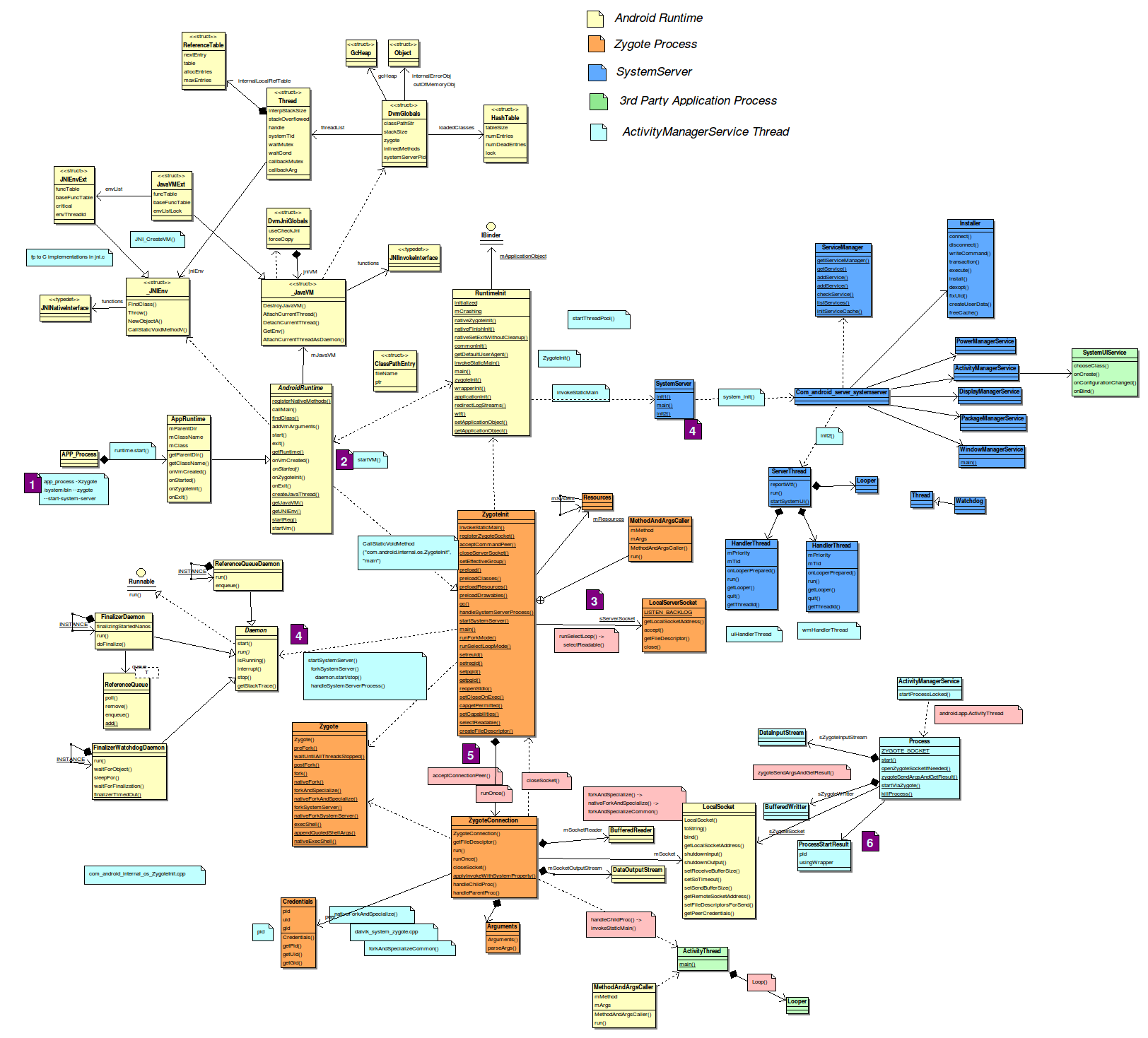
Init 是所有Linux程序的起点，而Zygote于Android，正如它的英文意思，是所有java程序的'孵化池'（玩过星际虫族的兄弟都晓得的）。用ps 输出可以看到

init 是 zygote的父进程, 而system\_server和其他所有的com.xxx结尾的应用程序都是从zygote fork 而来。本文将图过图表（辅予少量的代码）的方式来描述Zygote，system server 以及android application的启动过程。

废话少说，奉上两张大图开启我们的Zygote之旅。 第一张图是Zygote相关的所有类的结构图，另一张是Zygote启动的流程图。

# 概述

## 类图



按图索骥，我们按照图一中的序号一一分解Zygote的启动过程。

# App\_Process

APP\_Process: 启动zygote和其他Java程序的应用程序, 代码位于frameworks/base/cmds/app\_process/app\_main.cpp， 在init.rc 里面指定。

#init.rc

service zygote /system/bin/app\_process -Xzygote /system/bin --zygote --start-system-server

...

**int** main(**int** argc, **char**\* **const** argv[])  
{

++i; *// Skip unused "parent dir" argument.***while** (i < argc) {  
 **const char**\* arg = argv[i++];  
 **if** (strcmp(arg, **"--zygote"**) == 0) {  
 zygote = **true**;  
 niceName = ZYGOTE\_NICE\_NAME;  
 } **else if** (strcmp(arg, **"--start-system-server"**) == 0) {  
 startSystemServer = **true**;  
 } **else if** (strcmp(arg, **"--application"**) == 0) {  
 application = **true**;  
 } **else if** (strncmp(arg, **"--nice-name="**, 12) == 0) {  
 niceName.setTo(arg + 12);  
 } **else if** (strncmp(arg, **"--"**, 2) != 0) {  
 className.setTo(arg);  
 **break**;  
 } **else** {  
 --i;  
 **break**;  
 }  
}

**if** (zygote) {  
 runtime.start(**"com.android.internal.os.ZygoteInit"**, args, zygote);  
} **else if** (className) {  
 runtime.start(**"com.android.internal.os.RuntimeInit"**, args, zygote);  
} **else** {  
 fprintf(stderr, **"Error: no class name or --zygote supplied.\n"**);  
 app\_usage();  
 LOG\_ALWAYS\_FATAL(**"app\_process: no class name or --zygote supplied."**);  
}

可以看到，app\_process 里面定义了三种应用程序类型：

       1.  Zygote:  com.android.internal.os.ZygoteInit

       2.  System Server, 不单独启动，而是由Zygote启动

       3.  其他指定类名的Java 程序，比如说常用的 am. /system/bin/am 其实是一个shell程序，它的真正实现是

#!/system/bin/sh

if [ "$1" != "instrument" ] ; then

cmd activity "$@"

else

base=/system

export CLASSPATH=$base/framework/am.jar

exec app\_process $base/bin com.android.commands.am.Am "$@"

fi

这些Java的应用都是通过 AppRuntime.start（className)开始的。从第一张大图可以看出，其实AppRuntime是AndroidRuntime的子类，它主要实现了几个回调函数，而start()方法是实现在AndroidRuntime这个方法类里。什么是AnroidRuntime? 我们接下来马上开始。

需要注意的是Zygote并不是Init启动的第一个程序，从PID可以看出来，在它之前，一下Native实现的重要System Daemon （后台进程）可能先起来，比如 ServiceManager (service的DNS服务）.

# AndroidRuntime

首先，什么是Runtime ？看看Wiki给的几种解释：

* [*Run time (program lifecycle phase)*](http://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase))*, the period during which a computer program is executing*
* [*Runtime library*](http://en.wikipedia.org/wiki/Runtime_library)*, a program library designed to implement functions built into a programming language*

     我倾向这里指的是后者，看看更进一步的解释：

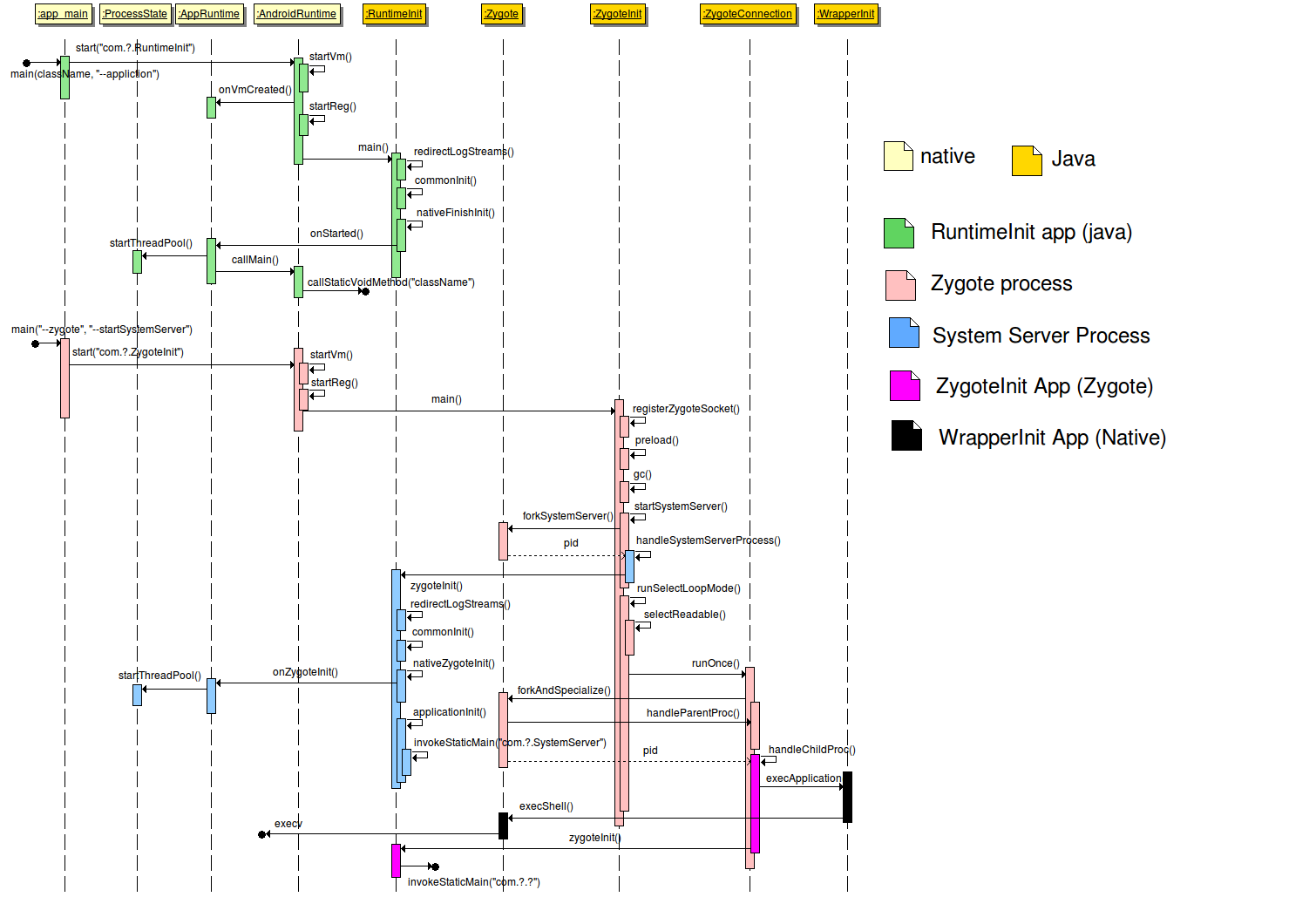
*In*[*computer programming*](http://en.wikipedia.org/wiki/Computer_programming)*, a****runtime library****is the*[*API*](http://en.wikipedia.org/wiki/API)*used by a*[*compiler*](http://en.wikipedia.org/wiki/Compiler)*to invoke some of the behaviors of a*[*runtime system*](http://en.wikipedia.org/wiki/Runtime_system)*. The runtime system implements the execution model and other fundamental behaviors of a*[*programming language*](http://en.wikipedia.org/wiki/Programming_language)*. The compiler inserts calls to the runtime library into the executable binary. During execution (*[*run time*](http://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase))*) of that*[*computer program*](http://en.wikipedia.org/wiki/Computer_program)*, execution of those calls to the runtime library cause communication between the application and the*[*runtime system*](http://en.wikipedia.org/wiki/Runtime_system)*. This often includes functions for input and output, or for memory management.*

     归纳起来的意思就是，Runtime 是支撑程序运行的基础库，它是与语言绑定在一起的。比如：

* C Runtime：就是C standard lib, 也就是我们常说的libc。（有意思的是， Wiki会自动将“C runtime" 重定向到 "C Standard Library").
* Java Runtime: 同样，Wiki将其重定向到” Java Virtual Machine", 这里当然包括Java 的支撑类库（.jar).
* AndroidRuntime:  显而易见，就是为Android应用运行所需的运行时环境。这个环境包括以下东东：
  + Dalvik VM: Android的Java VM, 解释运行Dex格式Java程序。每个进程运行一个虚拟机（什么叫运行虚拟机？说白了，就是一些C代码，不停的去解释Dex格式的二进制码(Bytecode)，把它们转成机器码(Machine code)，然后执行，当然，现在大多数的Java 虚拟机都支持JIT，也就是说，bytecode可能在运行前就已经被转换成机器码，从而大大提高了性能。过去一个普遍的认识是Java 程序比C，C++等静态编译的语言慢，但随着JIT的介入和发展，这个已经完全是过去时了，JIT的动态性运行允许虚拟机根据运行时环境，优化机器码的生成，在某些情况下，Java甚至可以比C/C++跑得更快，同时又兼具平台无关的特性，这也是为什么Java如今如此流行的原因之一吧）。
  + Android的Java 类库, 大部分来自于 Apache Hamony, 开源的Java API 实现，如 java.lang, java.util, java.net. 但去除了AWT, Swing 等部件。
  + JNI: C和Java互调的接口。
  + Libc: Android也有很多C代码，自然少不了libc，注意的是，Android的libc叫 bionic C.

       OK, 那就首先看看AndroidRuntime是怎么搭建起来的吧

## 时序图



上图给出了Zygote启动的大概流程，入口是AndroidRuntime.start(), 根据传入参数的不同可以有两种启动方式，一个是 "com.android.internal.os.RuntimeInit", 另一个是 ”com.android.internal.os.ZygoteInit", 对应RuntimeInit 和 ZygoteInit 两个类， 图中用绿色和粉红色分别表示。这两个类的主要区别在于Java端，可以明显看出，ZygoteInit 相比 RuntimeInit 多做了很多事情，比如说 “preload", "gc" 等等。但是在Native端，他们都做了相同的事， startVM() 和 startReg(), 让我们先从这里开始吧。

## JavaVM 和 JNIEnv

      从类图中看出，JavaVM 和 JNIEnv 是连结 AndroidRuntime 和 Dalvik VM 之间的唯一两个关卡，它隐藏了Dalvik 里面的实现细节，事实上，他就是两个函数指针结构体，给本地代码提供访问Java资源的接口。JNIEnv则相对于线程，通过JNIEnv的指针最终可以对应到Dalvik VM 内部的Thread 结构体，所有的调用就在这个结构体上下文完成。而JavaVM 对应的是DVMGlobal, 一个进程唯一的结构体，他内部维护了一个线程队列threadList，存放每个Thread 结构体对象， 同时还有各类状态的对象列表，及存放GC的结构体

### JavaVM

[复制代码](javascript:void(0);)

struct \_JavaVM {

const struct JNIInvokeInterface\* functions; //C的函数指针

#if defined(\_\_cplusplus) ...

jint GetEnv(void\*\* env, jint version)

{ return functions->GetEnv(this, env, version); }

#endif /\*\_\_cplusplus\*/

};

struct JNIInvokeInterface {

void\* reserved0;

...

jint (\*DestroyJavaVM)(JavaVM\*);

jint (\*AttachCurrentThread)(JavaVM\*, JNIEnv\*\*, void\*);

jint (\*DetachCurrentThread)(JavaVM\*);

jint (\*GetEnv)(JavaVM\*, void\*\*, jint);

jint (\*AttachCurrentThreadAsDaemon)(JavaVM\*, JNIEnv\*\*, void\*);

};

[复制代码](javascript:void(0);)

里面最常见的接口就是GetEnv()， 它返回一个JNIEnv对象，对应于每个DVM线程。JNIEnv的定义很长，有兴趣的同学可以到Jni.h 里面找，这里我们只看看这个对象是如何获static jint GetEnv(JavaVM\* vm, void\*\* env, jint version)

### GetEnv

调用位置：

[复制代码](javascript:void(0);)

JNIEnv\* AndroidRuntime::getJNIEnv()

{

JNIEnv\* env;

JavaVM\* vm = AndroidRuntime::getJavaVM();

assert(vm != NULL);

if (vm->GetEnv((void\*\*) &env, JNI\_VERSION\_1\_4) != JNI\_OK)

return NULL;

return env;

}

[复制代码](javascript:void(0);)

art/runtime/java\_vm\_ext.cc

**static** jint GetEnv(JavaVM\* vm, **void**\*\* env, jint version) {  
 **if** (vm == **nullptr** || env == **nullptr**) {  
 **return** JNI\_ERR;  
 }  
 Thread\* thread = Thread::Current();  
 **if** (thread == **nullptr**) {  
 \*env = **nullptr**;  
 **return** JNI\_EDETACHED;  
 }  
 JavaVMExt\* raw\_vm = **reinterpret\_cast**<JavaVMExt\*>(vm);  
 **return** raw\_vm->HandleGetEnv(env, version);  
}

#### HandleGetEnv

jint JavaVMExt::HandleGetEnv(*/\*out\*/***void**\*\* env, jint version) {  
 **for** (GetEnvHook hook : env\_hooks\_) {  
 jint res = hook(**this**, env, version);  
 **if** (res == JNI\_OK) {  
 **return** JNI\_OK;  
 } **else if** (res != JNI\_EVERSION) {  
 LOG(ERROR) << **"Error returned from a plugin GetEnv handler! "** << res;  
 **return** res;  
 }  
 }  
 LOG(ERROR) << **"Bad JNI version passed to GetEnv: "** << version;  
 **return** JNI\_EVERSION;  
}

到这里，我们知道JavaVM 和 JNIEnv 是本地（C/C++)代码用来与Java代码进行互调的，那在Java那端一定就是Java虚拟机以及对应的Java应用了。Java虚拟机到底是什么东东，它是如何创建的？答案从AndroidRuntime::startVM() 函数开始。startVM

## AndroidRuntime::startVm

AndroidRuntime::startVm(JavaVM\*\* pJavaVM, JNIEnv\*\* pEnv, **bool** zygote)

**bool** checkJni = **false**;  
property\_get(**"dalvik.vm.checkjni"**, propBuf, **""**);  
**if** (strcmp(propBuf, **"true"**) == 0) {  
 checkJni = **true**;  
} **else if** (strcmp(propBuf, **"false"**) != 0) {  
 */\* property is neither true nor false; fall back on kernel parameter \*/* property\_get(**"ro.kernel.android.checkjni"**, propBuf, **""**);  
 **if** (propBuf[0] == **'1'**) {  
 checkJni = **true**;  
 }  
}  
ALOGV(**"CheckJNI is %s\n"**, checkJni ? **"ON"** : **"OFF"**);  
**if** (checkJni) {  
 */\* extended JNI checking \*/* addOption(**"-Xcheck:jni"**);  
  
 */\* with -Xcheck:jni, this provides a JNI function call trace \*/  
 //addOption("-verbose:jni");*}

initArgs.version = JNI\_VERSION\_1\_4;

*/\*  
 \* Initialize the VM.  
 \*  
 \* The JavaVM\* is essentially per-process, and the JNIEnv\* is per-thread.  
 \* If this call succeeds, the VM is ready, and we can start issuing  
 \* JNI calls.  
 \*/*

//创建VM并返回JavaVM 和 JniEnv，pEnv对应于当前线程**if** (JNI\_CreateJavaVM(pJavaVM, pEnv, &initArgs) < 0) {  
 ALOGE(**"JNI\_CreateJavaVM failed\n"**);  
 **return** -1;  
 }  
  
 **return** 0;  
}

### java\_vm\_ext.JNI\_CreateJavaVM

art/runtime/java\_vm\_ext.cc

*// JNI Invocation interface.***extern "C"** jint JNI\_CreateJavaVM(JavaVM\*\* p\_vm, JNIEnv\*\* p\_env, **void**\* vm\_args) {  
 ScopedTrace trace(\_\_FUNCTION\_\_);  
 **const** JavaVMInitArgs\* args = **static\_cast**<JavaVMInitArgs\*>(vm\_args);  
 **if** (JavaVMExt::IsBadJniVersion(args->version)) {  
 LOG(ERROR) << **"Bad JNI version passed to CreateJavaVM: "** << args->version;  
 **return** JNI\_EVERSION;  
 }  
 RuntimeOptions options;  
 **for** (**int** i = 0; i < args->nOptions; ++i) {  
 JavaVMOption\* option = &args->options[i];  
 options.push\_back(std::make\_pair(std::string(option->optionString), option->extraInfo));  
 }  
 **bool** ignore\_unrecognized = args->ignoreUnrecognized;  
 **if** (!Runtime::Create(options, ignore\_unrecognized)) {  
 **return** JNI\_ERR;  
 }  
  
 *// Initialize native loader. This step makes sure we have  
 // everything set up before we start using JNI.* android::InitializeNativeLoader();  
  
 Runtime\* runtime = Runtime::Current();  
 **bool** started = runtime->Start();  
 **if** (!started) {  
 **delete** Thread::Current()->GetJniEnv();  
 **delete** runtime->GetJavaVM();  
 LOG(WARNING) << **"CreateJavaVM failed"**;  
 **return** JNI\_ERR;  
 }  
  
 \*p\_env = Thread::Current()->GetJniEnv();  
 \*p\_vm = runtime->GetJavaVM();  
 **return** JNI\_OK;  
}

Java虚拟机的启动有太多的细节在这里无法展开，这里我们只需要知道它做了以下一些事情：

ava虚拟机的启动有太多的细节在这里无法展开，这里我们只需要知道它做了以下一些事情：

1.  从property读取一系列启动参数。  
2.  创建和初始化结构体全局对象（每个进程）gDVM，及对应与JavaVM和JNIEnv的内部结构体 JavaVMExt, JNIEnvExt.  
3.  初始化java虚拟机，并创建虚拟机线程。"ps -t"，你可以发现每个Android应用都有以下几个线程

[复制代码](javascript:void(0);)

u0\_a46 1284 1281 714900 57896 20 0 0 0 fg ffffffff 00000000 S GC //垃圾回收

u0\_a46 1285 1281 714900 57896 20 0 0 0 fg ffffffff 00000000 S Signal Catcher

u0\_a46 1286 1281 714900 57896 20 0 0 0 fg ffffffff 00000000 S JDWP //Java 调试

u0\_a46 1287 1281 714900 57896 20 0 0 0 fg ffffffff 00000000 S Compiler //JIT

u0\_a46 1288 1281 714900 57896 20 0 0 0 fg ffffffff 00000000 S ReferenceQueueD

u0\_a46 1289 1281 714900 57896 20 0 0 0 fg ffffffff 00000000 S FinalizerDaemon //Finalizer监护

u0\_a46 1290 1281 714900 57896 20 0 0 0 fg ffffffff 00000000 S FinalizerWatchd //

[复制代码](javascript:void(0);)

4. 注册系统的JNI，Java程序通过这些JNI接口来访问底层的资源。

loadJniLibrary("javacore");

loadJniLibrary("nativehelper");

5. 为Zygote的启动做最后的准备，包括设置SID/UID, 以及mount 文件系统。  
6. 返回JavaVM 给Native代码，这样它就可以向上访问Java的接口。  
  
除了系统的JNI接口（”javacore", "nativehelper"), android framework 还有大量的Native实现，Android将所有这些接口一次性的通过start\_reg()来完成

## startReg

int AndroidRuntime::startReg(JNIEnv\* env){

androidSetCreateThreadFunc((android\_create\_thread\_fn) javaCreateThreadEtc); //创建JVM能访问的线程必须通过特定的接口。

env->PushLocalFrame(200);

if (**register\_jni\_procs**(gRegJNI, NELEM(gRegJNI), env) < 0) {

env->PopLocalFrame(NULL);

return -1;

}

env->PopLocalFrame(NULL);

return 0;

}

### androidSetCreateThreadFunc

system/core/libutils/Threads.cpp

**void** androidSetCreateThreadFunc(android\_create\_thread\_fn func)  
{  
 gCreateThreadFn = func;  
}

Android native层有两种Thread的创建方式

threads.cpp

status\_t Thread::run(const char\* name, int32\_t priority, size\_t stack)

{

...

if (mCanCallJava) {

res = createThreadEtc(\_threadLoop, this, name, priority, stack, &mThread);

} else {

res = androidCreateRawThreadEtc(\_threadLoop,this, name, priority, stack, &mThread);

}

...

}

 它们的区别在是是否能够调用Java端函数，普通的thread就是对pthread\_create的简单封装。

### androidCreateRawThreadEtc

**int** androidCreateRawThreadEtc(android\_thread\_func\_t entryFunction,  
 **void** \*userData,  
 **const char**\* threadName \_\_android\_unused,  
 int32\_t threadPriority,  
 size\_t threadStackSize,  
 android\_thread\_id\_t \*threadId)  
{  
 pthread\_attr\_t attr;  
 pthread\_attr\_init(&attr);  
 pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED);  
  
#if defined(\_\_ANDROID\_\_) */\* valgrind is rejecting RT-priority create reqs \*/* **if** (threadPriority != PRIORITY\_DEFAULT || threadName != NULL) {  
 *// Now that the pthread\_t has a method to find the associated  
 // android\_thread\_id\_t (pid) from pthread\_t, it would be possible to avoid  
 // this trampoline in some cases as the parent could set the properties  
 // for the child. However, there would be a race condition because the  
 // child becomes ready immediately, and it doesn't work for the name.  
 // prctl(PR\_SET\_NAME) only works for self; prctl(PR\_SET\_THREAD\_NAME) was  
 // proposed but not yet accepted.* thread\_data\_t\* t = **new** thread\_data\_t;  
 t->priority = threadPriority;  
 t->threadName = threadName ? strdup(threadName) : NULL;  
 t->entryFunction = entryFunction;  
 t->userData = userData;  
 entryFunction = (android\_thread\_func\_t)&thread\_data\_t::trampoline;  
 userData = t;  
 }  
#endif  
  
 **if** (threadStackSize) {  
 pthread\_attr\_setstacksize(&attr, threadStackSize);  
 }  
  
 errno = 0;  
 pthread\_t thread;  
 **int** result = pthread\_create(&thread, &attr,  
 (android\_pthread\_entry)entryFunction, userData);  
 pthread\_attr\_destroy(&attr);  
 **if** (result != 0) {  
 ALOGE(**"androidCreateRawThreadEtc failed (entry=%p, res=%d, %s)\n"  
 "(android threadPriority=%d)"**,  
 entryFunction, result, strerror(errno), threadPriority);  
 **return** 0;  
 }  
  
 *// Note that \*threadID is directly available to the parent only, as it is  
 // assigned after the child starts. Use memory barrier / lock if the child  
 // or other threads also need access.* **if** (threadId != NULL) {  
 \*threadId = (android\_thread\_id\_t)thread; *// XXX: this is not portable* }  
 **return** 1;  
}

### javaCreateThreadEtc

而能够访问Java端的thread需要跟JVM进行绑定，下面是具体的实现函数

*/\*static\*/* **int** AndroidRuntime::javaCreateThreadEtc(  
 android\_thread\_func\_t entryFunction,  
 **void**\* userData,  
 **const char**\* threadName,  
 int32\_t threadPriority,  
 size\_t threadStackSize,  
 android\_thread\_id\_t\* threadId)  
{  
 **void**\*\* args = (**void**\*\*) malloc(3 \* **sizeof**(**void**\*)); *// javaThreadShell must free* **int** result;  
  
 LOG\_ALWAYS\_FATAL\_IF(threadName == **nullptr**, **"threadName not provided to javaCreateThreadEtc"**);  
  
 args[0] = (**void**\*) entryFunction;  
 args[1] = userData;  
 args[2] = (**void**\*) strdup(threadName); *// javaThreadShell must free* result = androidCreateRawThreadEtc(AndroidRuntime::javaThreadShell, args,  
 threadName, threadPriority, threadStackSize, threadId);//entryFunc变成javaThreadShell.

**return** result;  
}

#### javaThreadShell

*/\*  
 \* When starting a native thread that will be visible from the VM, we  
 \* bounce through this to get the right attach/detach action.  
 \* Note that this function calls free(args)  
 \*/  
/\*static\*/* **int** AndroidRuntime::javaThreadShell(**void**\* args) {  
 **void**\* start = ((**void**\*\*)args)[0];  
 **void**\* userData = ((**void** \*\*)args)[1];  
 **char**\* name = (**char**\*) ((**void** \*\*)args)[2]; *// we own this storage* free(args);  
 JNIEnv\* env;  
 **int** result;  
  
 */\* hook us into the VM \*/*

/\* 跟 VM 绑定 \*/**if** (javaAttachThread(name, &env) != JNI\_OK)  
 **return** -1;  
/\* 运行真正的'entryFunc' \*/  
 */\* start the thread running \*/* result = (\*(android\_thread\_func\_t)start)(userData);  
  
 */\* unhook us \*/* javaDetachThread();  
 free(name);  
  
 **return** result;  
}

attachVM() 到底做什么事情？ 篇幅有限无法展开，这里只需要知道这么几点：

* + 一个进程里有一个Java 虚拟机，Java 虚拟机内部有很多线程，如上面列到的 GC, FinalizeDaemon, 以及用户创建的线程等等.
  + 每个Java线程都维护一个JNIEnvExt对象，里面存放一个指向DVM 内部Thread对象的指针，也就是说，所有从native到Java端的调用，都会引用到这个对象。
  + 所有通过JVM创建的线程都会在VM内部记录在案，但是当前，我们还没有进入Java世界，本地创建的线程VM自然就不知道，因此我们需要通过attach来通知VM来创建相应的内部数据结构

#### javaAttachThread

**static int** javaAttachThread(**const char**\* threadName, JNIEnv\*\* pEnv)  
{  
 JavaVMAttachArgs args;  
 JavaVM\* vm;  
 jint result;  
  
 vm = AndroidRuntime::getJavaVM();  
 assert(vm != NULL);  
  
 args.version = JNI\_VERSION\_1\_4;  
 args.name = (**char**\*) threadName;  
 args.group = NULL;  
  
 result = vm->AttachCurrentThread(pEnv, (**void**\*) &args);  
 **if** (result != JNI\_OK)  
 ALOGI(**"NOTE: attach of thread '%s' failed\n"**, threadName);  
  
 **return** result;  
}

#### AttachCurrentThread

**static** jint AttachCurrentThread(JavaVM\* vm, JNIEnv\*\* p\_env, **void**\* thr\_args) {  
 **return** AttachCurrentThreadInternal(vm, p\_env, thr\_args, **false**);  
}

看看下面代码，你就知道，其实Attach()做的一件重要的事情就是 创建thread和JNIEnvExt

**private**:  
 **static** jint AttachCurrentThreadInternal(JavaVM\* vm, JNIEnv\*\* p\_env, **void**\* raw\_args, **bool** as\_daemon) {  
 **if** (vm == **nullptr** || p\_env == **nullptr**) {  
 **return** JNI\_ERR;  
 }  
  
 *// Return immediately if we're already attached.* Thread\* self = Thread::Current();  
 **if** (self != **nullptr**) {  
 \*p\_env = self->GetJniEnv();  
 **return** JNI\_OK;  
 }  
  
 Runtime\* runtime = **reinterpret\_cast**<JavaVMExt\*>(vm)->GetRuntime();  
  
 *// No threads allowed in zygote mode.* **if** (runtime->IsZygote()) {  
 LOG(ERROR) << **"Attempt to attach a thread in the zygote"**;  
 **return** JNI\_ERR;  
 }  
  
 JavaVMAttachArgs\* args = **static\_cast**<JavaVMAttachArgs\*>(raw\_args);  
 **const char**\* thread\_name = **nullptr**;  
 jobject thread\_group = **nullptr**;  
 **if** (args != **nullptr**) {  
 **if** (JavaVMExt::IsBadJniVersion(args->version)) {  
 LOG(ERROR) << **"Bad JNI version passed to "** << (as\_daemon ? **"AttachCurrentThreadAsDaemon"** : **"AttachCurrentThread"**) << **": "** << args->version;  
 **return** JNI\_EVERSION;  
 }  
 thread\_name = args->name;  
 thread\_group = args->group;  
 }  
  
 **if** (!runtime->AttachCurrentThread(thread\_name, as\_daemon, thread\_group,  
 !runtime->IsAotCompiler())) {  
 \*p\_env = **nullptr**;  
 **return** JNI\_ERR;  
 } **else** {  
 \*p\_env = Thread::Current()->GetJniEnv();  
 **return** JNI\_OK;  
 }  
 }  
};

#### Runtime::AttachCurrentThread

**bool** Runtime::AttachCurrentThread(**const char**\* thread\_name, **bool** as\_daemon, jobject thread\_group,  
 **bool** create\_peer) {  
 ScopedTrace trace(\_\_FUNCTION\_\_);  
 **return** Thread::Attach(thread\_name, as\_daemon, thread\_group, create\_peer) != **nullptr**;  
}

### register\_jni\_procs

开始注册本地的JNI接口函数了- register\_jni\_procs(), 这个函数其实就是对一个全局数组gRegJni[] 进行遍历调用，这个数组展开可以得到以下的结果

**static int** register\_jni\_procs(**const** RegJNIRec array[], size\_t count, JNIEnv\* env)  
{  
 **for** (size\_t i = 0; i < count; i++) {  
 **if** (array[i].mProc(env) < 0) {  
#ifndef NDEBUG  
 ALOGD(**"----------!!! %s failed to load\n"**, array[i].mName);  
#endif  
 **return** -1;  
 }  
 }  
 **return** 0;  
}

#### gRegJN

**static const RegJNIRec** gRegJNI[] = {  
 REG\_JNI(register\_com\_android\_internal\_os\_RuntimeInit),  
 REG\_JNI(register\_com\_android\_internal\_os\_ZygoteInit),  
 REG\_JNI(register\_android\_os\_SystemClock),  
 REG\_JNI(register\_android\_util\_EventLog),  
 REG\_JNI(register\_android\_util\_Log),  
 REG\_JNI(register\_android\_util\_PathParser),  
 REG\_JNI(register\_android\_app\_admin\_SecurityLog),  
 REG\_JNI(register\_android\_content\_AssetManager),,  
 REG\_JNI(register\_android\_content\_XmlBlock),  
 REG\_JNI(register\_android\_text\_AndroidCharacter),  
,  
 REG\_JNI(register\_android\_os\_Process),  
 REG\_JNI(register\_android\_os\_SystemProperties),  
 REG\_JNI(register\_android\_os\_Binder),  
 REG\_JNI(register\_android\_os\_Parcel),  
   
 REG\_JNI(register\_android\_view\_Surface),  
 REG\_JNI(register\_android\_view\_SurfaceControl),  
  
 REG\_JNI(register\_android\_os\_Debug),  
 REG\_JNI(register\_android\_os\_FileObserver),  
 REG\_JNI(register\_android\_os\_GraphicsEnvironment),  
 REG\_JNI(register\_android\_os\_MessageQueue),  
 ,  
 REG\_JNI(register\_android\_os\_Trace),  
 REG\_JNI(register\_android\_os\_UEventObserver),  
 REG\_JNI(register\_android\_net\_LocalSocketImpl),  
 REG\_JNI(register\_android\_net\_NetworkUtils),  
 REG\_JNI(register\_android\_net\_TrafficStats),  
 REG\_JNI(register\_android\_os\_MemoryFile),  
 REG\_JNI(register\_com\_android\_internal\_os\_PathClassLoaderFactory),  
 REG\_JNI(register\_com\_android\_internal\_os\_Zygote),  
 ,  
 REG\_JNI(register\_android\_app\_Activity),  
 REG\_JNI(register\_android\_app\_ActivityThread),  
 REG\_JNI(register\_android\_app\_ApplicationLoaders),  
 REG\_JNI(register\_android\_app\_NativeActivity),  
 REG\_JNI(register\_android\_view\_KeyEvent),  
 REG\_JNI(register\_android\_view\_MotionEvent),  
 REG\_JNI(register\_com\_android\_internal\_content\_NativeLibraryHelper),  
 REG\_JNI(register\_com\_android\_internal\_net\_NetworkStatsFactory),  
 REG\_JNI(register\_com\_android\_internal\_os\_FuseAppLoop),  
};

##### REG\_JNI

采用的是宏定义，展开后可以得到：

#ifdef NDEBUG  
 #define REG\_JNI(name) { name }  
 **struct RegJNIRec** {  
 **int** (\*mProc)(JNIEnv\*);  
 };  
#else  
 #define REG\_JNI(name) { name, #name }  
 **struct** RegJNIRec {  
 **int** (\*mProc)(JNIEnv\*);  
 **const char**\* mName;  
 };  
#endif

##### register\_XXX\_RuntimeInit

**【】p**

**int** register\_com\_android\_internal\_os\_RuntimeInit(JNIEnv\* env)  
{  
 **const** JNINativeMethod methods[] = {  
 { **"nativeFinishInit"**, **"()V"**,  
 (**void**\*) com\_android\_internal\_os\_RuntimeInit\_nativeFinishInit },  
 { **"nativeSetExitWithoutCleanup"**, **"(Z)V"**,  
 (**void**\*) com\_android\_internal\_os\_RuntimeInit\_nativeSetExitWithoutCleanup },  
 };  
 **return** jniRegisterNativeMethods(env, **"com/android/internal/os/RuntimeInit"**,  
 methods, NELEM(methods));  
}

##### register\_XXX\_ZygoteInit

**int** register\_com\_android\_internal\_os\_ZygoteInit(JNIEnv\* env)  
{  
 **const** JNINativeMethod methods[] = {  
 { **"nativeZygoteInit"**, **"()V"**,  
 (**void**\*) com\_android\_internal\_os\_ZygoteInit\_nativeZygoteInit },  
 };  
 **return** jniRegisterNativeMethods(env, **"com/android/internal/os/ZygoteInit"**,  
 methods, NELEM(methods));  
}

#### register\_xxx

每个 register\_xxx是一个函数指针

int jniRegisterNativeMethods(

C\_JNIEnv\* env,

const char\* className,

const JNINativeMethod\* gMethods,

int numMethods);

#### jniRegisterNativeMethods

RegisterNativeMethods 在VM内部到底发生了什么？

## 小结

好了，经过了千辛万苦，Android的运行时环境都已经准备就绪了，让我们再回顾一下AndroidRuntime的初始化都做了哪些工作，

1. 创建了Dalvik VM.
2. 获取Native 访问Java的两个接口对象，JavaVM 和 JNIENV。
3. 注册了一批 (见gRegJni[]) native接口给VM。

这些操作都是相对耗时的工作，如果每个进程都做同样的工作势必会影响到启动速度，这也是为什么我们需要通过Zygote来创建Android 应用，因为通过Linux fork 的 copy\_on\_write的机制，子进程可以将这些初始化好的内存空间直接映射到自己的进程空间里，不在需要做重复的工作，从而提高了应用启动的速度。

可以是，Android系统只需要基本的运行时环境就够了吗？ 答案显然是No。AndriodRuntime 只是提供了语言层面的基础支持，在一个多任务，多用户的图形操作系统上快速的孵化和运行应用程序，我们需要更多。这就是Zygote，这就是为什么在图2中，ZygoteInit会比RuntimeInit做更多的事情。那接下来，让我们真正进入Zygote的世界

# ZygoteInit

当VM准备就绪，就可以运行Java代码了，系统也将在此第一次进入Java世界，还记得app\_main.cpp里面调到的 Runtime.start()的参数吗, 那就是我们要运行的Java类。Android支持两个类做为起点，一个是‘com.android.internal.os.ZygoteInit', 另一个是'com.android.internal.os.RuntimeInit'。

     此外Runtime\_Init 类里还定义了一个ZygoteInit() 静态方法。它在Zygote 创建一个新的应用进程的时候被创建，它和RuntimeInit 类的main() 函数做了以下相同的事情：

* redirectLogStreams(): 将System.out 和 System.err 输出重定向到Android 的Log系统（定义在 android.util.Log).
* commonInit(): 初始化了一下系统属性，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，调试过Android代码的同学经常看到的"\*\*\* FATAL EXCEPTION IN SYSTEM PROCESS" 打印就出自这里：

## main

**public static void** main(String argv[]) {  
 ZygoteServer zygoteServer = **new** ZygoteServer();  
  
 *// Mark zygote start. This ensures that thread creation will throw  
 // an error.* ZygoteHooks.startZygoteNoThreadCreation();  
  
 *// 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** {  
 *// Report Zygote start time to tron unless it is a runtime restart* **if** (!**"1"**.equals(SystemProperties.get(**"sys.boot\_completed"**))) {  
 MetricsLogger.histogram(**null**, **"boot\_zygote\_init"**,  
 (**int**) SystemClock.elapsedRealtime());  
 }  
  
 String bootTimeTag = Process.is64Bit() ? **"Zygote64Timing"** : **"Zygote32Timing"**;  
 BootTimingsTraceLog bootTimingsTraceLog = **new** BootTimingsTraceLog(bootTimeTag,  
 Trace.TRACE\_TAG\_DALVIK);  
 bootTimingsTraceLog.traceBegin(**"ZygoteInit"**);  
 RuntimeInit.enableDdms();  
 *// Start profiling the zygote initialization.* SamplingProfilerIntegration.start();  
  
 **boolean** startSystemServer = **false**;  
 String socketName = **"zygote"**;  
 String abiList = **null**;  
 **boolean** enableLazyPreload = **false**;  
 **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."**);  
 }  
  
 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());  
 preload(bootTimingsTraceLog);  
 EventLog.writeEvent(LOG\_BOOT\_PROGRESS\_PRELOAD\_END,  
 SystemClock.uptimeMillis());  
 bootTimingsTraceLog.traceEnd(); *// ZygotePreload* } **else** {  
 Zygote.resetNicePriority();  
 }  
  
 *// Finish profiling the zygote initialization.* SamplingProfilerIntegration.writeZygoteSnapshot();  
  
 *// Do an initial gc to clean up after startup* bootTimingsTraceLog.traceBegin(**"PostZygoteInitGC"**);  
 gcAndFinalize();  
 bootTimingsTraceLog.traceEnd(); *// PostZygoteInitGC* bootTimingsTraceLog.traceEnd(); *// ZygoteInit  
 // Disable tracing so that forked processes do not inherit stale tracing tags from  
 // Zygote.* Trace.setTracingEnabled(**false**);  
  
 *// Zygote process unmounts root storage spaces.* Zygote.nativeUnmountStorageOnInit();  
  
 *// Set seccomp policy* Seccomp.setPolicy();  
  
 ZygoteHooks.stopZygoteNoThreadCreation();  
  
 **if** (startSystemServer) {  
 startSystemServer(abiList, socketName, zygoteServer);  
 }  
  
 Log.i(TAG, **"Accepting command socket connections"**);  
 zygoteServer.runSelectLoop(abiList);  
  
 zygoteServer.closeServerSocket();  
 } **catch** (Zygote.MethodAndArgsCaller caller) {  
 caller.run();  
 } **catch** (Throwable ex) {  
 Log.e(TAG, **"System zygote died with exception"**, ex);  
 zygoteServer.closeServerSocket();  
 **throw** ex;  
 }  
}

## Runtime\_init. commonInit

**protected static final void** commonInit() {  
 **if** (DEBUG) Slog.d(TAG, **"Entered RuntimeInit!"**);  
  
 */\*  
 \* set handlers; these apply to all threads in the VM. Apps can replace  
 \* the default handler, but not the pre handler.  
 \*/* Thread.setUncaughtExceptionPreHandler(**new** LoggingHandler());  
 Thread.setDefaultUncaughtExceptionHandler(**new** KillApplicationHandler());

Runtime\_init.java

...

Thread.setDefaultUncaughtExceptionHandler(new UncaughtHandler());

...

**private static class** LoggingHandler **implements** Thread.UncaughtExceptionHandler {  
 @Override  
 **public void** uncaughtException(Thread t, Throwable e) {  
 *// Don't re-enter if KillApplicationHandler has already run* **if** (mCrashing) **return**;  
 **if** (mApplicationObject == **null**) {  
 *// The "FATAL EXCEPTION" string is still used on Android even though  
 // apps can set a custom UncaughtExceptionHandler that renders uncaught  
 // exceptions non-fatal.* Clog\_e(TAG, **"\*\*\* FATAL EXCEPTION IN SYSTEM PROCESS: "** + t.getName(), e);  
 } **else** {  
 StringBuilder message = **new** StringBuilder();  
 *// The "FATAL EXCEPTION" string is still used on Android even though  
 // apps can set a custom UncaughtExceptionHandler that renders uncaught  
 // exceptions non-fatal.* message.append(**"FATAL EXCEPTION: "**).append(t.getName()).append(**"\n"**);  
 **final** String processName = ActivityThread.currentProcessName();  
 **if** (processName != **null**) {  
 message.append(**"Process: "**).append(processName).append(**", "**);  
 }  
 message.append(**"PID: "**).append(Process.myPid());  
 Clog\_e(TAG, message.toString(), e);  
 }  
 }  
}

接下来，RuntimeInit::main() 和 RuntimeInit::ZygoteInit() 分别调用里nativeFinishInit() 和 nativeZygoteInit(), 由此开始分道扬镳.

RuntimeInit 的nativeFinishInit（） 最终会调用到 app\_main.cpp 里的 onStarted() 函数，里面调用Java类的main() 函数，然后结束进程退出。

## app\_main.onStarted()

**virtual void** onStarted()  
{  
 sp<ProcessState> proc = ProcessState::self();  
 ALOGV(**"App process: starting thread pool.\n"**);  
 proc->startThreadPool();  
  
 AndroidRuntime\* ar = AndroidRuntime::getRuntime();  
 ar->callMain(mClassName, mClass, mArgs);  
  
 IPCThreadState::self()->stopProcess();  
 hardware::IPCThreadState::self()->stopProcess();  
}

而 RuntimeInit::ZygoteInit() 则会调到 app\_main.cpp 的 onZygoteInit()

## app\_main.onZygoteInit

**virtual void** onZygoteInit()  
{  
 sp<ProcessState> proc = ProcessState::self();  
 ALOGV(**"App process: starting thread pool.\n"**);  
 proc->startThreadPool();  
}

它仅仅是启动了一个ThreadPool, 剩下的工作则回到Java端由RuntimeInit::applicationInit()完成。

    所以，我们不妨这样理解RuntimeInit::main(), RuntimeInit::ZygoteInit(), ZygoteInit::main()三者关系， RuntimeInit的main() 方法提供标准的Java程序运行方式，而RuntimeInit的ZygoteInit() 则是关门为Android应用启动的方法，它是在Zygote 创建一个新的应用进程的时候调用的，这部分代码实现在ZygoteInit 类里。除了上面描述的差别，ZygoteInit 类里还多做了如下几件事情，让我们一一详细解析。

1. registerZygoteSocket();
2. startSystemServer();
3. runSelectLoopMode();

## ZygoteInit其他附加流程

ZygoteInit 类里还多做了如下几件事情，让我们一一详细解析。

1. registerZygoteSocket();
2. startSystemServer();
3. runSelectLoopMode();

### RegisterZygoteSocket

其实做的事情很简单，就是初始化Server端（也就是Zygote)的socket。值得一提的是，这里用到的socket类型是LocalSocket, 它是Android对Linux 的 Local Socket的一个封装。Local Socket 是Linux提供的一种基于Socket的进程间通信方式，对Server端来讲，唯一的区别就是bind到一个本地的文件描述符(fd)而不是某个IP地址和端口号。Android里很多地方用到了Local Socket做进程间的通信，搜索一下init.rc， 你会看到很多这样的语句

[复制代码](javascript:void(0);)

socket adbd stream 660 system system

socket vold stream 0660 root mount

socket netd stream 0660 root system

socket dnsproxyd stream 0660 root inet

socket mdns stream 0660 root system

socket rild stream 660 root radio

socket rild-debug stream 660 radio system

socket zygote stream 660 root system

socket installd stream 600 system system

socket racoon stream 600 system system

socket mtpd stream 600 system system

socket dumpstate stream 0660 shell log

socket mdnsd stream 0660 mdnsr inet

 当init 解析到这样一条语句，它将做这么几件事：

            1. 调用 create\_socket() (system/core/init/util.c)， 创建一个Socket fd, 将这个fd 与某个文件(/dev/socket/xxx, xxx 就是上面列到的名字，比如，zygote) 绑定(bind), 根据init.rc 里面定义来设定相关的用户，组和权限。最后返回这个fd。  
            2. 将socket 名字（带‘ANDROID\_SOCKET\_'前缀)（比如 zygote) 和 fd 注册到init 进程的环境变量里，这样所有的其他进程（所有进程都是init的子进程）都可以通过 getenv(name)获取到这个fd.

       ZygoteInit 通过以下代码来完成Socket Server端的配置

#### zygoteServer.registerServerSocket;

private static final String ANDROID\_SOCKET\_ENV = "ANDROID\_SOCKET\_zygote";

**void** registerServerSocket(String socketName) {  
 **if** (mServerSocket == **null**) {  
 **int** fileDesc;  
 **final** String fullSocketName = ANDROID\_SOCKET\_PREFIX + socketName;  
 **try** {  
 String env = System.getenv(fullSocketName);  
 fileDesc = Integer.parseInt(env);  
 } **catch** (RuntimeException ex) {  
 **throw new** RuntimeException(fullSocketName + **" unset or invalid"**, ex);  
 }  
  
 **try** {  
 FileDescriptor fd = **new** FileDescriptor();  
 fd.setInt$(fileDesc);  
 mServerSocket = **new** LocalServerSocket(fd);  
 } **catch** (IOException ex) {  
 **throw new** RuntimeException(  
 **"Error binding to local socket '"** + fileDesc + **"'"**, ex);  
 }  
 }  
}

Server端创建完毕，接下来就可以相应客户端连接请求了。我们前面讲过，AndroidRuntime 一系列复杂的初始化工作可以通过fork来帮助子进程来简化这个过程，对了，Zygote创建Socket server 端就是用来响应这个fork的请求。那发起请求的是谁？Zygote fork的子进程又是谁？答案是ActivityManagerService 和 Android Application. 这个过程是怎样的？ 答案就在Andriod System Server的启动过程

### preload

这是Android启动过程中最耗时间的两件事情。preloadClassess 将framework.jar里的preloaded-classes 定义的所有class load到内存里，preloaded-classes 编译Android后可以在framework/base下找到。而preloadResources 将系统的Resource(不是在用户apk里定义的resource）load到内存。

    资源preload到Zygoted的进程地址空间，所有fork的子进程将共享这份空间而无需重新load, 这大大减少了应用程序的启动时间，但反过来增加了系统的启动时间。通过对preload 类和资源数目进行调整可以加快系统启动。

**static void** preload(BootTimingsTraceLog bootTimingsTraceLog) {  
 Log.d(TAG, **"begin preload"**);  
 bootTimingsTraceLog.traceBegin(**"BeginIcuCachePinning"**);  
 beginIcuCachePinning();  
 bootTimingsTraceLog.traceEnd(); *// BeginIcuCachePinning* bootTimingsTraceLog.traceBegin(**"PreloadClasses"**);  
 preloadClasses();  
 bootTimingsTraceLog.traceEnd(); *// PreloadClasses* bootTimingsTraceLog.traceBegin(**"PreloadResources"**);  
 preloadResources();  
 bootTimingsTraceLog.traceEnd(); *// PreloadResources* Trace.traceBegin(Trace.TRACE\_TAG\_DALVIK, **"PreloadOpenGL"**);  
 preloadOpenGL();  
 Trace.traceEnd(Trace.TRACE\_TAG\_DALVIK);  
 preloadSharedLibraries();  
 preloadTextResources();  
 *// Ask the WebViewFactory to do any initialization that must run in the zygote process,  
 // for memory sharing purposes.* WebViewFactory.prepareWebViewInZygote();  
 endIcuCachePinning();  
 warmUpJcaProviders();  
 Log.d(TAG, **"end preload"**);  
  
 sPreloadComplete = **true**;  
}

### gcAndFinalize

为什么调了2次System.gc()和runFinalizationSync()? 这是因为gc()调用只是通知VM进行垃圾回收，是否回收，什么时候回收全又VM内部算法决定。GC的回收有一个复杂的状态机控制，通过多次调用，可以使得尽可能多的资源得到回收。gc()必须在fork之前完成（接下来的StartSystemServer就会有fork操作），这样将来被复制出来的子进程才能有尽可能少的垃圾内存没有释放。

*/\*package\*/* **static void** gcAndFinalize() {  
 **final** VMRuntime runtime = VMRuntime.getRuntime();  
  
 */\* runFinalizationSync() lets finalizers be called in Zygote,  
 \* which doesn't have a HeapWorker thread.  
 \*/* System.gc();  
 runtime.runFinalizationSync();  
 System.gc();  
}

### Start SystemServer

想起init.rc 里面启动zygote的参数了吗， "--start-system-server", System Server 是Zygote fork 的第一个Java 进程， 这个进程非常重要

**private static boolean** startSystemServer(String abiList, String socketName, ZygoteServer zygoteServer)  
 **throws** Zygote.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\_PTRACE,  
 OsConstants.CAP\_SYS\_TIME,  
 OsConstants.CAP\_SYS\_TTY\_CONFIG,  
 OsConstants.CAP\_WAKE\_ALARM  
 );  
 */\* 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=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,1021,1023,1032,3001,3002,3003,3006,3007,3009,3010"**,  
 **"--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);  
  
 */\* 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);  
 }  
  
 zygoteServer.closeServerSocket();  
 handleSystemServerProcess(parsedArgs);  
 }  
  
 **return true**;  
}

因为他们有很多的系统线程，提供所有核心的系统服务，我们可以用 'ps -t |grep <system server pid>'来看看都有哪些线程，排除前面列出的几个Java 虚拟机线程，还有

#### ps -T -A| grep 844

他们有很多的系统线程，提供所有核心的系统服务，我们可以用 'ps -t |grep <system server pid>'来看看都有哪些线程，排除前面列出的几个Java 虚拟机线程，还有

原来这些线程都是binder线程！！！

看到大名鼎鼎的WindowManager, ActivityManager了吗？对了，它们都是运行在system\_server的进程里。还有很多“Binder-x"的线程，它们是各个Service为了响应应用程序远程调用请求而创建的。除此之外，还有很多内部的线程，比如 ”UI thread", "InputReader", "InputDispatch" 等等，我们将在后续的文章里将这些模块具体分析。本文，我们只关心System Server是如何创建起来的。

bullhead:/ # ps -T -A| grep 844

system 844 844 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S system\_server

system 844 849 542 2473672 242472 do\_sigtimedwait 7f2774c3b4 S Signal Catcher

system 844 850 542 2473672 242472 \_\_skb\_recv\_datagram 7f2774cd8c S JDWP

system 844 858 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S android.bg

system 844 859 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S ActivityManager

system 844 860 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S android.ui

system 844 861 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S ActivityManager

system 844 863 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S batterystats-sy

system 844 866 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S system\_server

system 844 867 542 2473672 242472 inotify\_read 7f2774ccfc S FileObserver

system 844 868 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S android.fg

system 844 869 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S android.io

system 844 870 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S android.display

system 844 871 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S CpuTracker

system 844 872 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S PowerManagerSer

system 844 873 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S system\_server

system 844 874 542 2473672 242472 pm\_get\_wakeup\_count 7f2774ccfc S system\_server

system 844 875 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S BatteryStats\_wa

system 844 876 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S android.display

system 844 877 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S PackageManager

system 844 898 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S PackageInstalle

system 844 904 542 2473672 242472 binder\_thread\_read 7f2774c2c4 S Binder:844\_3

system 844 905 542 2473672 242472 poll\_schedule\_timeout 7f2774c2f4 S HubConnection

system 844 906 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S system-server-i

system 844 908 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S AccountManagerS

system 844 909 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S SettingsProvide

system 844 910 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S system\_server

system 844 911 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S AlarmManager

system 844 922 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S InputDispatcher

system 844 923 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S InputReader

system 844 924 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S StorageManagerS

system 844 929 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S NetworkStats

system 844 930 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S NetworkPolicy

system 844 945 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S AudioService

system 844 947 542 2473672 242472 poll\_schedule\_timeout 7f2774c2f4 S UEventObserver

system 844 949 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S system\_server

system 844 950 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S ConnectivityThr

system 844 951 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S system\_server

system 844 952 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S system\_server

system 844 953 542 2473672 242472 poll\_schedule\_timeout 7f2774c2f4 S system\_server

system 844 1027 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S SoundPool

system 844 1028 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S SoundPoolThread

system 844 1029 542 2473672 242472 \_\_skb\_recv\_datagram 7f2774c174 S Thread-4

system 844 1074 542 2473672 242472 binder\_thread\_read 7f2774c2c4 S Binder:844\_4

system 844 1078 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S NetworkStatsObs

system 844 1087 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S watchdog

system 844 1112 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S system\_server

system 844 1115 542 2473672 242472 binder\_thread\_read 7f2774c2c4 S Binder:844\_5

system 844 1134 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S Thread-5

system 844 1137 542 2473672 242472 pipe\_wait 7f2774ccfc S system\_server

system 844 1140 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S android.bg

system 844 1141 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S NetworkTimeUpda

system 844 1206 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S RenderThread

system 844 1449 542 2473672 242472 binder\_thread\_read 7f2774c2c4 S Binder:844\_7

system 844 1454 542 2473672 242472 binder\_thread\_read 7f2774c2c4 S Binder:844\_8

system 844 1495 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S RenderThread

system 844 2362 542 2473672 242472 binder\_thread\_read 7f2774c2c4 S Binder:844\_9

system 844 2378 542 2473672 242472 binder\_thread\_read 7f2774c2c4 S Binder:844\_A

system 844 2379 542 2473672 242472 binder\_thread\_read 7f2774c2c4 S Binder:844\_B

system 844 3416 542 2473672 242472 SyS\_epoll\_wait 7f2774c1d4 S NetworkMonitorN

system 844 3421 542 2473672 242472 \_\_skb\_recv\_datagram 7f2774ccfc S Thread-9

system 844 3424 542 2473672 242472 \_\_skb\_recv\_datagram 7f2774ccfc S Thread-10

system 844 3429 542 2473672 242472 \_\_skb\_recv\_datagram 7f2774ccfc S Thread-11

system 844 3433 542 2473672 242472 \_\_skb\_recv\_datagram 7f2774ccfc S Thread-12

system 844 3447 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S OkHttp Connecti

system 844 3456 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S Okio Watchdog

system 844 4232 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S AudioTrack

system 844 5150 542 2473672 242472 futex\_wait\_queue\_me 7f27700630 S OkHttp Connecti

#### System Server 启动流程

这个过程代码很多，涉及到很多类，我们用一张时序图来描述这个过程。图中不同的颜色代表运行在不同的线程中。



1. ZygoteInit fork 出一个新的进程，这个进程就是SystemServer进程。

**2**.  fork出来的子进程在**handleSystemServerProcess** 里开始初始化工作，初始化分两步，一部在native 完成，另外一部分（大部分）在Java端完成。 native端的工作在AppRuntime(AndroidRuntime的子类）::onZygoteInit()完成，做的一件事情就是启动了一个Thread, 这个Thread是SystemServer的主线程(最左边的粉色方块), 负责接收来至其他进程的Binder调用请求。代码如下

#### handleSystemServerProcess

fork出来的子进程在**handleSystemServerProcess** 里开始初始化工作，初始化分两步，一部在native 完成，另外一部分（大部分）在Java端完成。 native端的工作在AppRuntime(AndroidRuntime的子类）::onZygoteInit()完成，做的一件事情就是启动了一个Thread, 这个Thread是SystemServer的主线程(最左边的粉色方块), 负责接收来至其他进程的Binder调用请求

**private static void** handleSystemServerProcess(  
 ZygoteConnection.Arguments parsedArgs)  
 **throws** Zygote.MethodAndArgsCaller {  
  
 *// set umask to 0077 so new files and directories will default to owner-only permissions.* Os.umask(S\_IRWXG | S\_IRWXO);  
  
 **if** (parsedArgs.niceName != **null**) {  
 Process.setArgV0(parsedArgs.niceName);  
 }  
  
 **final** String systemServerClasspath = Os.getenv(**"SYSTEMSERVERCLASSPATH"**);  
 **if** (systemServerClasspath != **null**) {  
 performSystemServerDexOpt(systemServerClasspath);  
 *// Capturing profiles is only supported for debug or eng builds since selinux normally  
 // prevents it.* **boolean** profileSystemServer = SystemProperties.getBoolean(  
 **"dalvik.vm.profilesystemserver"**, **false**);  
 **if** (profileSystemServer && (Build.IS\_USERDEBUG || Build.IS\_ENG)) {  
 **try** {  
 File profileDir = Environment.getDataProfilesDePackageDirectory(  
 Process.SYSTEM\_UID, **"system\_server"**);  
 File profile = **new** File(profileDir, **"primary.prof"**);  
 profile.getParentFile().mkdirs();  
 profile.createNewFile();  
 String[] codePaths = systemServerClasspath.split(**":"**);  
 VMRuntime.registerAppInfo(profile.getPath(), codePaths);  
 } **catch** (Exception e) {  
 Log.wtf(TAG, **"Failed to set up system server profile"**, e);  
 }  
 }  
 }  
  
 **if** (parsedArgs.invokeWith != **null**) {  
 String[] args = parsedArgs.remainingArgs;  
 *// If we have a non-null system server class path, we'll have to duplicate the  
 // existing arguments and append the classpath to it. ART will handle the classpath  
 // correctly when we exec a new process.* **if** (systemServerClasspath != **null**) {  
 String[] amendedArgs = **new** String[args.length + 2];  
 amendedArgs[0] = **"-cp"**;  
 amendedArgs[1] = systemServerClasspath;  
 System.arraycopy(args, 0, amendedArgs, 2, args.length);  
 args = amendedArgs;  
 }  
  
 WrapperInit.execApplication(parsedArgs.invokeWith,  
 parsedArgs.niceName, parsedArgs.targetSdkVersion,  
 VMRuntime.getCurrentInstructionSet(), **null**, args);  
 } **else** {  
 ClassLoader cl = **null**;  
 **if** (systemServerClasspath != **null**) {  
 cl = createPathClassLoader(systemServerClasspath, parsedArgs.targetSdkVersion);  
  
 Thread.currentThread().setContextClassLoader(cl);  
 }  
  
 */\*  
 \* Pass the remaining arguments to SystemServer.  
 \*/* ZygoteInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.remainingArgs, cl);  
 }  
  
 */\* should never reach here \*/*}

#### ZygoteInit.zygoteInit

**public static final void** zygoteInit(**int** targetSdkVersion, String[] argv,  
 ClassLoader classLoader) **throws** Zygote.MethodAndArgsCaller {  
   
 RuntimeInit.redirectLogStreams();  
 RuntimeInit.commonInit();  
 ZygoteInit.nativeZygoteInit();  
 RuntimeInit.applicationInit(targetSdkVersion, argv, classLoader);  
}

##### nativeZygoteInit

**static void** com\_android\_internal\_os\_ZygoteInit\_nativeZygoteInit(JNIEnv\* env, jobject clazz)  
{  
 gCurRuntime->onZygoteInit();  
}

###### onZygoteInit

**virtual void** onZygoteInit()  
{  
 sp<ProcessState> proc = ProcessState::self();  
 ALOGV(**"App process: starting thread pool.\n"**);  
 proc->startThreadPool();  
}

##### RuntimeInit.applicationInit

nativeZygoteInit() 完成后， 接下来开始Java层的初始化，这个流程比较长，也比较复杂，我们分成很多步进行讲解。初始化的入口是SystemServer的main() 函数，这里又调用了Native的 Init1(). Init1实现在com\_android\_server\_SystemServer.cpp,   最终调用到的函数是system\_init(). system\_init（）的实现如下：

**protected static void** applicationInit(**int** targetSdkVersion, String[] argv, ClassLoader classLoader)  
 **throws** Zygote.MethodAndArgsCaller {  
 *// If the application calls System.exit(), terminate the process  
 // immediately without running any shutdown hooks. It is not possible to  
 // shutdown an Android application gracefully. Among other things, the  
 // Android runtime shutdown hooks close the Binder driver, which can cause  
 // leftover running threads to crash before the process actually exits.* nativeSetExitWithoutCleanup(**true**);  
  
 *// We want to be fairly aggressive about heap utilization, to avoid  
 // holding on to a lot of memory that isn't needed.* 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**;  
 }  
  
 *// The end of of the RuntimeInit event (see #zygoteInit).* Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);  
  
 *// Remaining arguments are passed to the start class's static main* invokeStaticMain(args.startClass, args.startArgs, classLoader);  
}

###### invokeStaticMain

fghdf

**private static void** invokeStaticMain(String className, String[] argv, ClassLoader classLoader)  
 **throws** Zygote.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** Zygote.MethodAndArgsCaller(m, argv);  
}

细心的读者可能会问这么几个问题：

QA：

1. 为什么不直接Call 相应的Java函数，而是通过一个exception?  MethodAndArgsCaller. 这里Android 巧妙的运用到了Java Exception的一些设计特性。Execption一大特性就是当发生或抛出异常的时候，可以从发生错误的地方顺着调用栈回溯直到找到捕捉该异常的代码段。捕捉该异常的代码如下

[复制代码](javascript:void(0);)

public static void main(String argv[]) {

...

try {

runSelectLoop();

closeServerSocket();

} catch (MethodAndArgsCaller caller) {

caller.run(); //真正的入口在这里。

} catch (RuntimeException ex) {

...

}

}

[复制代码](javascript:void(0);)

这下你明白为什么总在dumpstate 文件里看到以下的调用栈了吧

...

at java.lang.reflect.Method.invokeNative(Native Method)

at java.lang.reflect.Method.invoke(Method.java:511)

at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:793)

at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:560)

at dalvik.system.NativeStart.main(Native Method)

## Init1()

nativeZygoteInit() 完成后， 接下来开始Java层的初始化，这个流程比较长，也比较复杂，我们分成很多步进行讲解。初始化的入口是SystemServer的main() 函数，这里又调用了Native的 Init1(). Init1实现在com\_android\_server\_SystemServer.cpp,   最终调用到的函数是system\_init(). system\_init（）的实现如下：

[复制代码](javascript:void(0);)

extern "C" status\_t system\_init()

{

sp<ProcessState> proc(ProcessState::self());

sp<IServiceManager> sm = defaultServiceManager();

sm->asBinder()->linkToDeath(grim, grim.get(), 0);

property\_get("system\_init.startsurfaceflinger", propBuf, "1");

if (strcmp(propBuf, "1") == 0) {

// Start the SurfaceFlinger

SurfaceFlinger::instantiate(); //初始化 SurfaceFlinger

android\_vt = 7;

}

property\_get("system\_init.startsensorservice", propBuf, "1");

if (strcmp(propBuf, "1") == 0) {

// Start the sensor service

SensorService::instantiate(); // 初始化SensorService.

}

ALOGI("System server: starting Android runtime.\n");

AndroidRuntime\* runtime = AndroidRuntime::getRuntime();

JNIEnv\* env = runtime->getJNIEnv();  
 ...

jclass clazz = env->FindClass("com/android/server/SystemServer");

jmethodID methodId = env->GetStaticMethodID(clazz, "init2", "()V");

...

env->CallStaticVoidMethod(clazz, methodId);

ProcessState::self()->startThreadPool();

IPCThreadState::self()->joinThreadPool();

return NO\_ERROR;

}

[复制代码](javascript:void(0);)

 几点需要注意：

 A. SurfaceFlinger Service可以运行在System\_Server 进程里，也可以运行在独立的进程里，如果是后者的话，需要在init.rc 里加上一句 "setprop system\_init.startsurfaceflinger=1" 并且确保 service surfaceflinger 没有被 “disable”

 B. init2 实现在System\_Server.java，我们后面会详细介绍。

## join\_threadpool()

system\_init() 最后，join\_threadpool() 将当前线程挂起，等待binder的请求。这个thread的名字就是"Binder\_1". 关于service 和 binder的内部机制，请参考文章<http://www.cnblogs.com/samchen2009/p/3316001.html>

## init2

至此，system server的native初始化工作完成，又重新回到了Java端，在这里，很多非常重要的系统服务将被启动。 这些工作将在一个新的线程内开始，线程名"android.server.ServerThread", 见下图的绿色条块。在ServerThread里，SystemServer 首先创建了两个线程，UI thread 和 WindowManager thread, 见图中的橙色和桃色条块，这两个thread的handle将被传给某些Service的构造函数，部分的启动工作会分发到这两个Thread内进行。

    每个Thread都最终进入等待循环，这里用到了Android的Looper机制，Looper，Handler是Android的进程内的消息传递和处理机制，我们将会在文章 <http://www.cnblogs.com/samchen2009/p/3316004.html> 里详细介绍，这里，我们只需要知道，Looper在某个线程里睡眠等待消息队列里的消息，然后在某个特定的Handler里来处理这个消息。换句话说，指定某件事情在某个线程里进行处理

## 启动一系列的Service

Df

接下来，System Server会启动一系列的Service， 其中最重要的就是Acitivity Manager 和Window Manager.

从图中可以看出，Activity Manager 有一个Looper Thread, AThread。这里请注意Binder Thread和 Looper的区别，我们在后面会有专门的文章介绍它们。Android里大量用到了Binder 与 Looper的组合，其中很重要的一个原因是为了解决多线程中复杂的同步问题，通过一个Looper和对应的Message 队列，可以将来着不同进程的Binder 调用序列化，而不需要维护复杂的且容易出问题的锁。

WindowManager 类似，他的Handler Thread 是我们刚才提到的System server 启动初期创建的两个Handler线程之一，WMThread. 他的Binder线程会由Kernel的Binder Driver来指定。

除了ActivityManager Service 和 WindowManager Service, 还有很多其他的服务相继启动，这里不再详述，只需要知道一个Service启动需要的几个步骤：

       1. 初始化Service 对象，获得IBinder对象。

       2. 启动后台线程，并进入Loop等待。

       3. 将自己注册到Service Manager, 让其他进程通过名字可以获得远程调用必须的IBinder的对象。

## 依赖关系

Dfg

这么多服务之间是有依赖关系的，比如说，ActivityManager Service 在WindowManager Service 初始化完成之前是不能启动应用的。那如何控制这些先后顺序的？这里由System server的启动线程（下图中的绿色条快）通过SystemReady()接口来完成。每个系统服务必须实现一个SystemReady() 接口，当被调用，表明系统已经OK， 该服务可以访问（直接或间接）其他服务的资源。最后一个被调到的服务就是AcitivyManager Service. AM的SystemReady() 是通过Runnable 在另外一个线程里完成，参加下图中注释8下方的那个箭头。在这个Runnable里面要做的事情，就是将当前排在最顶层的一个应用程序启动 - resumeTopActivityLocked()，通常来讲，这就是我们常说的‘HOME’， 当然，这里可以指定其他的应用程序作为Startup应用，比如说GoogleTV 里面可以将电视应用作为启动程序，这样用户启动后直接可以看到节目，类似现在家里的机顶盒。此外，ActivityManager Service 还会广播 BOOT\_COMPLETED 事件给整个系统，一般来说，很多应用的后台Service可以通过注册这个Event的 Receiver 来监听并启动

## Android应用的启动

Android应用的启动比较复杂，我们会在专门的章节里面去研究ActivityManager的工作细节，此处，我们只需要知道ActivityStack 是存在当前运行Activity的栈，resumeTopActivityLocked() 从其中找到要启动的那一个（在最开始，该栈是空的，因为需要通过moveTaskFromFrontLocked() 将‘Home' 推到该栈中），如果该应用从来没有启动过，我们需要通过AcitivyManagerService 为其创建一个进程。注意！进程 并不是由ActivityManager创建的，别忘了，我们前面提到Zygote是所有Android 应用的孵化器，对，ActivityManager 只是通知Zygote创建而已。这个通信是通过Process.java里面实现的，具体代码如下：

[复制代码](javascript:void(0);)

static LocalSocket sZygoteSocket;

private static ProcessStartResult zygoteSendArgsAndGetResult(ArrayList<String> args)

throws ZygoteStartFailedEx {

openZygoteSocketIfNeeded();

try {

...

sZygoteWriter.write(arg);

}

...

sZygoteWriter.flush(); //发送后等待...

...

result.pid = sZygoteInputStream.readInt();

...

return result;

}

sZygoteSocket = null;

}

}

[复制代码](javascript:void(0);)

到此为止，System Server的启动已经完成，Zygote的启动也已经完成，接下来我们介绍Zygote进程生命里做的唯一一件事，克隆自己。

# Fork

在Process.java 发送fork 请求之前，Zygote已经准备好了服务器端，这个我们已经在前面的Zygote Init 章节里介绍过了。此处我们简要分析一下Zygote Server端收到请求的处理。代码在ZygoteServer.java 的runSelectLoop()里，

## runSelectLoop

**void** runSelectLoop(String abiList) **throws** Zygote.MethodAndArgsCaller {  
 ArrayList<FileDescriptor> fds = **new** ArrayList<FileDescriptor>();  
 ArrayList<ZygoteConnection> peers = **new** ArrayList<ZygoteConnection>();  
  
 fds.add(mServerSocket.getFileDescriptor());  
 peers.add(**null**);  
  
 **while** (**true**) {

ds  
 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**;  
 }

/\* 接收新的连接 \*/  
 **if** (i == 0) {  
 ZygoteConnection newPeer = acceptCommandPeer(abiList);  
 peers.add(newPeer);  
 fds.add(newPeer.getFileDesciptor());  
 } **else** {

/\* 在这里完成fork操作 \*/  
 **boolean** done = **peers.get(i).runOnce**(**this**);  
 **if** (done) {  
 peers.remove(i);  
 fds.remove(i);  
 }  
 }  
 }  
 }  
}

## ZygoteConnection.runOnce

Sdf

**boolean** runOnce(ZygoteServer zygoteServer) **throws** Zygote.MethodAndArgsCaller {  
  
 String args[];  
 Arguments parsedArgs = **null**;  
 FileDescriptor[] descriptors;  
  
 **try** {  
 args = readArgumentList();  
 descriptors = mSocket.getAncillaryFileDescriptors();  
 } **catch** (IOException ex) {  
 Log.w(TAG, **"IOException on command socket "** + ex.getMessage());  
 closeSocket();  
 **return true**;  
 }  
  
 **if** (args == **null**) {  
 *// EOF reached.* closeSocket();  
 **return true**;  
 }  
  
 */\*\* the stderr of the most recent request, if avail \*/* PrintStream newStderr = **null**;  
  
 **if** (descriptors != **null** && descriptors.length >= 3) {  
 newStderr = **new** PrintStream(  
 **new** FileOutputStream(descriptors[2]));  
 }  
  
 **int** pid = -1;  
 FileDescriptor childPipeFd = **null**;  
 FileDescriptor serverPipeFd = **null**;  
  
 **try** {  
 parsedArgs = **new** Arguments(args);  
  
 **if** (parsedArgs.abiListQuery) {  
 **return** handleAbiListQuery();  
 }  
  
 **if** (parsedArgs.preloadDefault) {  
 **return** handlePreload();  
 }  
  
 **if** (parsedArgs.preloadPackage != **null**) {  
 **return** handlePreloadPackage(parsedArgs.preloadPackage,  
 parsedArgs.preloadPackageLibs, parsedArgs.preloadPackageCacheKey);  
 }  
  
 **if** (parsedArgs.permittedCapabilities != 0 || parsedArgs.effectiveCapabilities != 0) {  
 **throw new** ZygoteSecurityException(**"Client may not specify capabilities: "** +  
 **"permitted=0x"** + Long.toHexString(parsedArgs.permittedCapabilities) +  
 **", effective=0x"** + Long.toHexString(parsedArgs.effectiveCapabilities));  
 }  
  
 applyUidSecurityPolicy(parsedArgs, peer);  
 applyInvokeWithSecurityPolicy(parsedArgs, peer);  
  
 applyDebuggerSystemProperty(parsedArgs);  
 applyInvokeWithSystemProperty(parsedArgs);  
  
 **int**[][] rlimits = **null**;  
  
 **if** (parsedArgs.rlimits != **null**) {  
 rlimits = parsedArgs.rlimits.toArray(intArray2d);  
 }  
  
 **int**[] fdsToIgnore = **null**;  
  
 **if** (parsedArgs.invokeWith != **null**) {  
 FileDescriptor[] pipeFds = Os.pipe2(O\_CLOEXEC);  
 childPipeFd = pipeFds[1];  
 serverPipeFd = pipeFds[0];  
 Os.fcntlInt(childPipeFd, F\_SETFD, 0);  
 fdsToIgnore = **new int**[] { childPipeFd.getInt$(), serverPipeFd.getInt$() };  
 }  
  
 */\*\*  
 \* In order to avoid leaking descriptors to the Zygote child,  
 \* the native code must close the two Zygote socket descriptors  
 \* in the child process before it switches from Zygote-root to  
 \* the UID and privileges of the application being launched.  
 \*  
 \* In order to avoid "bad file descriptor" errors when the  
 \* two LocalSocket objects are closed, the Posix file  
 \* descriptors are released via a dup2() call which closes  
 \* the socket and substitutes an open descriptor to /dev/null.  
 \*/* **int** [] fdsToClose = { -1, -1 };  
  
 FileDescriptor fd = mSocket.getFileDescriptor();  
  
 **if** (fd != **null**) {  
 fdsToClose[0] = fd.getInt$();  
 }  
  
 fd = zygoteServer.getServerSocketFileDescriptor();  
  
 **if** (fd != **null**) {  
 fdsToClose[1] = fd.getInt$();  
 }  
  
 fd = **null**;  
  
 pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid, parsedArgs.gids,  
 parsedArgs.debugFlags, rlimits, parsedArgs.mountExternal, parsedArgs.seInfo,  
 parsedArgs.niceName, fdsToClose, fdsToIgnore, parsedArgs.instructionSet,  
 parsedArgs.appDataDir);  
 } **catch** (ErrnoException ex) {  
 logAndPrintError(newStderr, **"Exception creating pipe"**, ex);  
 } **catch** (IllegalArgumentException ex) {  
 logAndPrintError(newStderr, **"Invalid zygote arguments"**, ex);  
 } **catch** (ZygoteSecurityException ex) {  
 logAndPrintError(newStderr,  
 **"Zygote security policy prevents request: "**, ex);  
 }  
  
 **try** {  
 **if** (pid == 0) {  
 *// in child* // 子进程，将serverFd释放zygoteServer.closeServerSocket();  
 IoUtils.closeQuietly(serverPipeFd);  
 serverPipeFd = **null**;  
 handleChildProc(parsedArgs, descriptors, childPipeFd, newStderr);  
  
 *// should never get here, the child is expected to either  
 // throw Zygote.MethodAndArgsCaller or exec().* **return true**;  
 } **else** {  
 *// in parent...pid of < 0 means failure*

// 父进程，将不用的子进程的PipeFd释放IoUtils.closeQuietly(childPipeFd);  
 childPipeFd = **null**;  
 **return** handleParentProc(pid, descriptors, serverPipeFd, parsedArgs);  
 }  
 } **finally** {  
 IoUtils.closeQuietly(childPipeFd);  
 IoUtils.closeQuietly(serverPipeFd);  
 }  
}

## handleChildProc

Android 应用的启动在handleChildProc里完成

**private void** handleChildProc(Arguments parsedArgs,  
 FileDescriptor[] descriptors, FileDescriptor pipeFd, PrintStream newStderr)  
 **throws** Zygote.MethodAndArgsCaller {  
 */\*\*  
 \* By the time we get here, the native code has closed the two actual Zygote  
 \* socket connections, and substituted /dev/null in their place. The LocalSocket  
 \* objects still need to be closed properly.  
 \*/*// 不需要服务端的socketcloseSocket();  
 **if** (parsedArgs.niceName != **null**) {  
 Process.setArgV0(parsedArgs.niceName);  
 }  
  
 *// End of the postFork event.* Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

//从Process.java 来的都为true  
 **if** (parsedArgs.invokeWith != **null**) {

// 启动命令行程序  
 WrapperInit.execApplication(parsedArgs.invokeWith,  
 parsedArgs.niceName, parsedArgs.targetSdkVersion,  
 VMRuntime.getCurrentInstructionSet(),  
 pipeFd, parsedArgs.remainingArgs);  
 } **else** {

// 几乎所有的应用启动都走这条路  
 ZygoteInit.zygoteInit(parsedArgs.targetSdkVersion,  
 parsedArgs.remainingArgs, **null** */\* classLoader \*/*);  
 }  
}

这里走的是RuntimeInit.ZygoteInit()， 和startSystemServer 一样，最后invokeStaticMain(,""android.app.ActivityThread",); invokeStatickMain() 函数实现是

static void invokeStaticMain(ClassLoader loader,String className, String[] argv) throws zygoteInit.MethodAndArgsCaller {

...

throw new ZygoteInit.MethodAndArgsCaller(m, argv);

}

1.

这里Android 巧妙的运用到了Java Exception的一些设计特性。Execption一大特性就是当发生或抛出异常的时候，可以从发生错误的地方顺着调用栈回溯直到找到捕捉该异常的代码段

2：为什么所有的应用都从“android.app.ActivityThread")开始？ 在这里留个伏笔，我们会在http://www.cnblogs.com/samchen2009/p/3315993.html 系统的介绍Android Activity 从启动到显示的整个过程。  
  
应用程序启动完毕，Zygote有重新睡去，等待新的应用程序启动请求。

# 善后工作

是不是到此之后，Zygote的工作变得很轻松了，可以宜养天年了？可惜现代社会，哪个父母把孩子养大就可以撒手不管了？尤其是像Sytem Server 这样肩负社会重任的大儿子，出问题了父母还是要帮一把的。这里，Zygote会默默的在后台凝视这自己的大儿子，一旦发现System Server 挂掉了，将其回收，然后将自己杀掉，重新开始新的一生， 可怜天下父母心啊。这段实现在代码 ，frameworks/base/core/jni/com\_android\_internal\_os\_Zygote.cpp

**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) {  
 pid\_t pid = ForkAndSpecializeCommon(env, uid, gid, gids,  
 debug\_flags, rlimits,  
 permittedCapabilities, effectiveCapabilities,  
 MOUNT\_EXTERNAL\_DEFAULT, NULL, NULL, **true**, NULL,  
 NULL, NULL, NULL);  
 **if** (pid > 0) {  
 *// The zygote process checks whether the child process has died or not.* ALOGI(**"System server process %d has been created"**, pid);  
 gSystemServerPid = pid;  
 *// There is a slight window that the system server process has crashed  
 // but it went unnoticed because we haven't published its pid yet. So  
 // we recheck here just to make sure that all is well.* **int** status;

/\* WNOHANG 会让waitpid 立即返回，这里只是为了预防上面的赋值语句没有完成之前SystemServer就crash 了\*/  
 **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;  
}

## sigchldHandler

/\* 真正的处理在这里 \*/

static void sigchldHandler(int s)

{

...

pid\_t pid;

int status;

...

while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {

...

if (pid == gDvm.systemServerPid) {

...

kill(getpid(), SIGKILL);

}

}

...

}

在Unix-like系统，父进程必须用 waitpid 等待子进程的退出，否则子进程将变成"Zombie" (僵尸）进程，不仅系统资源泄漏，而且系统将崩溃（没有system server，所有Android应用程序都无法运行）。但是waitpid() 是一个阻塞函数（WNOHANG参数除外)，所以通常做法是在signal 处理函数里进行无阻塞的处理，因为每个子进程退出的时候，系统会发出 SIGCHID 信号。Zygote会把自己杀掉， 那父亲死了，所有的应用程序不就成为孤儿了？ 不会，因为父进程被杀掉后系统会自动给所有的子进程发生SIGHUP信号，该信号的默认处理就是将杀掉自己退出当前进程。但是一些后台进程（Daemon)可以通过设置SIG\_IGN参数来忽略这个信号，从而得以在后台继续运行。

# 总结

Zygote和System Server的启动过程终于介绍完了，让我们对着上面这张完整的类图再来重温一下这个过程吧。

1.  init 根据init.rc 运行 app\_process, 并携带‘--zygote' 和 ’--startSystemServer' 参数。

2.  AndroidRuntime.cpp::start() 里将启动JavaVM，并且注册所有framework相关的系统JNI接口。

3.  第一次进入Java世界，运行ZygoteInit.java::main() 函数初始化Zygote. Zygote 并创建Socket的server 端。

4.  然后fork一个新的进程并在新进程里初始化SystemServer. Fork之前，Zygote是preload常用的Java类库，以及系统的resources，同时GC（）清理内存空间，为子进程省去重复的工作。

5.  SystemServer 里将所有的系统Service初始化，包括ActivityManager 和 WindowManager, 他们是应用程序运行起来的前提。

6.  依次同时，Zygote监听服务端Socket，等待新的应用启动请求。

7.  ActivityManager ready 之后寻找系统的“Startup” Application, 将请求发给Zygote。

8.  Zygote收到请求后，fork出一个新的进程。

9.  Zygote监听并处理SystemServer 的 SIGCHID 信号，一旦System Server崩溃，立即将自己杀死。init会重启Zygote.

# REF

图解Android - Zygote, System Server 启动分析

https://www.cnblogs.com/samchen2009/p/3294713.html