图解Android - Zygote, System Server 启动分析

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

```
>adb shell ps | grep -E 'init|926'
 root      1     0     656    372   00000000 0805d546 S /init
 root      926   1     685724 43832 ffffffff b76801e0 S zygote
 system    1018  926   795924 62720 ffffffff b767fff6 S system_server
 u0_a6     1241  926   717704 39252 ffffffff b76819eb S com.android.systemui
 u0_a37    1325  926   698280 29024 ffffffff b76819eb S com.android.inputmethod.latin
 radio     1349  926   711284 30116 ffffffff b76819eb S com.android.phone
 u0_a7     1357  926   720792 41444 ffffffff b76819eb S com.android.launcher
 u0_a5     1523  926   703576 26416 ffffffff b76819eb S com.android.providers.calendar
 u0_a25    1672  926   693716 21328 ffffffff b76819eb S com.android.musicfx
 u0_a17    2040  926   716888 33992 ffffffff b76819eb S android.process.acore
 u0_a21    2436  926   716060 23904 ffffffff b76819eb S com.android.calendar
```

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。

公告

昵称：漫天尘沙
园龄：2年4个月
粉丝：66
关注：0
+加关注

< 2016年1月 >

日	一	二	三	四	五	六
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

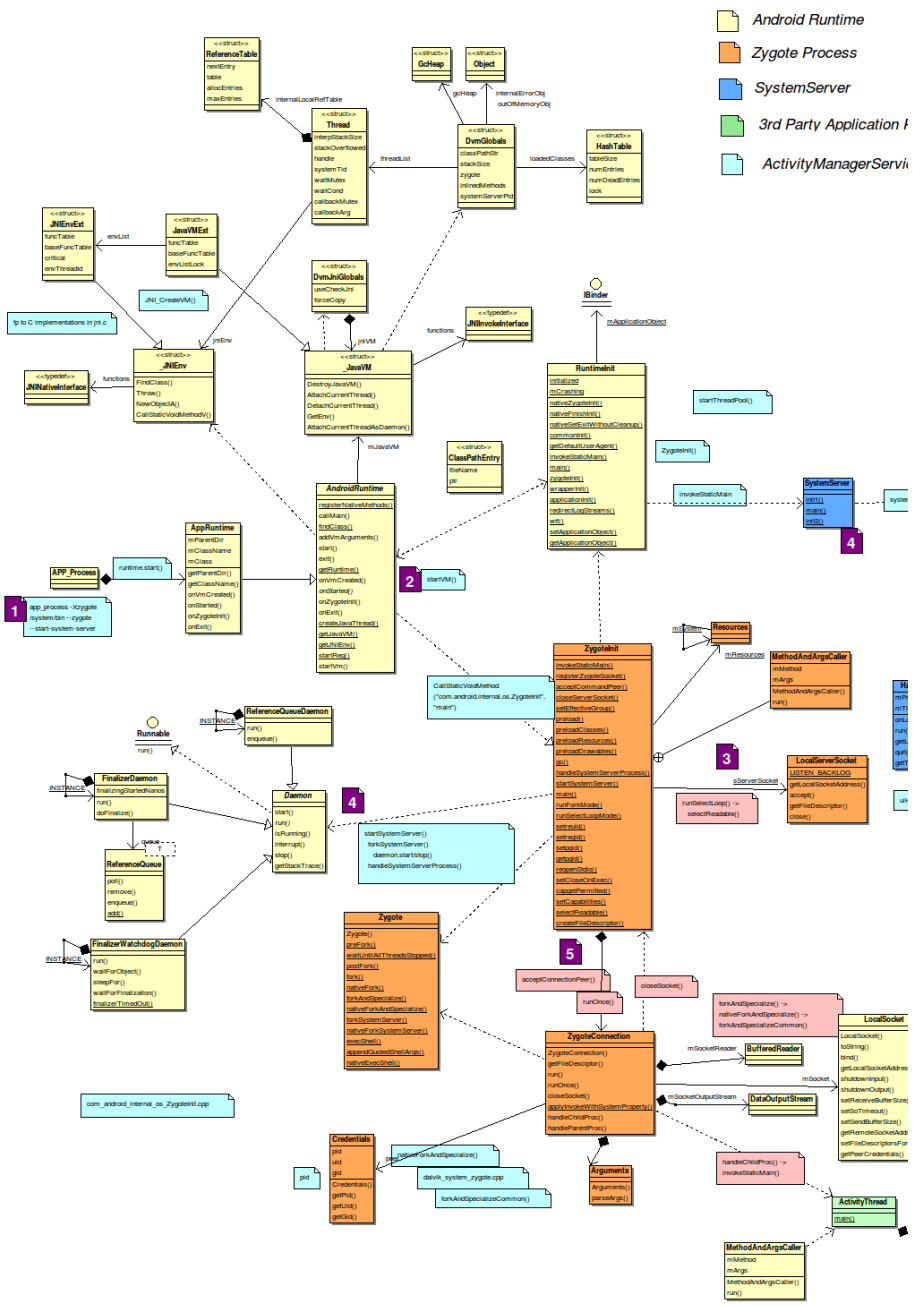
我的标签

Android (8)
Framework (4) Handler (1)
Init (1)
Input Dispatcher (1)
Input Manager Service (1)
Input reader (1)
InputManager (1)
Key processing (1)
Looper (1) 更多

随笔分类(9)

Ruby 学算法
读书笔记(1)
软件开发那点事
图解Android 系列(8)
虚拟化

随笔档案(9)



按图索骥，我们按照图一中的序号一一分解Zygote的启动过程。

1. 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
```

代码如下

```
...
else if (strcmp(arg, "--zygote") == 0) {
    zygote = true;
    niceName = "zygote";
} else if (strcmp(arg, "--start-system-server") == 0) {
    startSystemServer = true;
} else if (strcmp(arg, "--application") == 0) {
    application = true;
```


2013年11月 (2)
2013年10月 (5)
2013年9月 (2)

积分与排名

积分 - 22763
排名 - 9244

最新评论

1. Re:图解Android - Syst...
跪倒 膜拜
--Dark_Flame_Master
2. Re:图解Android - Bind...
System Server这个大儿子，笑死了，先评论再看内容。
--Bourneer
3. Re:图解Android - Andr...
nice share, thk very.
--哦哈哟嘿
4. Re:图解Android - Andr...
请问是用的什么软件画的流程图？
--juude
5. Re:图解Android - Andr...
写的已经很详细， 不知是对于opengl硬件加速部分是否有了解！
--shuangquanjun

阅读排行榜

1. 图解Android - Zygote, ...
2. 图解Android - Android ...
3. 图解Android - Android ...
4. 图解Android - Binder ...
5. 图解Android - Android ...
6. 图解Android - 如何看An...
7. 图解Android - System ...
8. 图解Android - Looper, ...
9. 推荐书单(727)

评论排行榜

1. 图解Android - Android ...
2. 图解Android - Android ...
3. 图解Android - Binder ...
4. 图解Android - Zygote, ...
5. 图解Android - Android ...
6. 图解Android - System ...
7. 图解Android - 如何看An...
8. 图解Android - Looper, ...
9. 推荐书单(1)

推荐排行榜

1. 图解Android - Binder ...
2. 图解Android - Android ...
3. 图解Android - 如何看An...
4. 图解Android - Android ...
5. 图解Android - Zygote, ...
6. 图解Android - Android ...

```
}
...

if (zygote) {
   runtime.start("com.android.internal.os.ZygoteInit",
        startSystemServer ? "start-system-server" : "");
} else if (className) {
   // Remainder of args get passed to startup class main()
   runtime.mClassName = className;
   ...
   runtime.start("com.android.internal.os.RuntimeInit",
         application ? "application" : "tool");
} else {
}
```

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

```
exec app_process $base/bin com.android.commands.am.Am "$@"
```

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）.

2. AndroidRuntime

首先，什么是Runtime ？看看Wiki给的几种解释：

- *Run time (program lifecycle phase), the period during which a computer program is executing*
- *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：

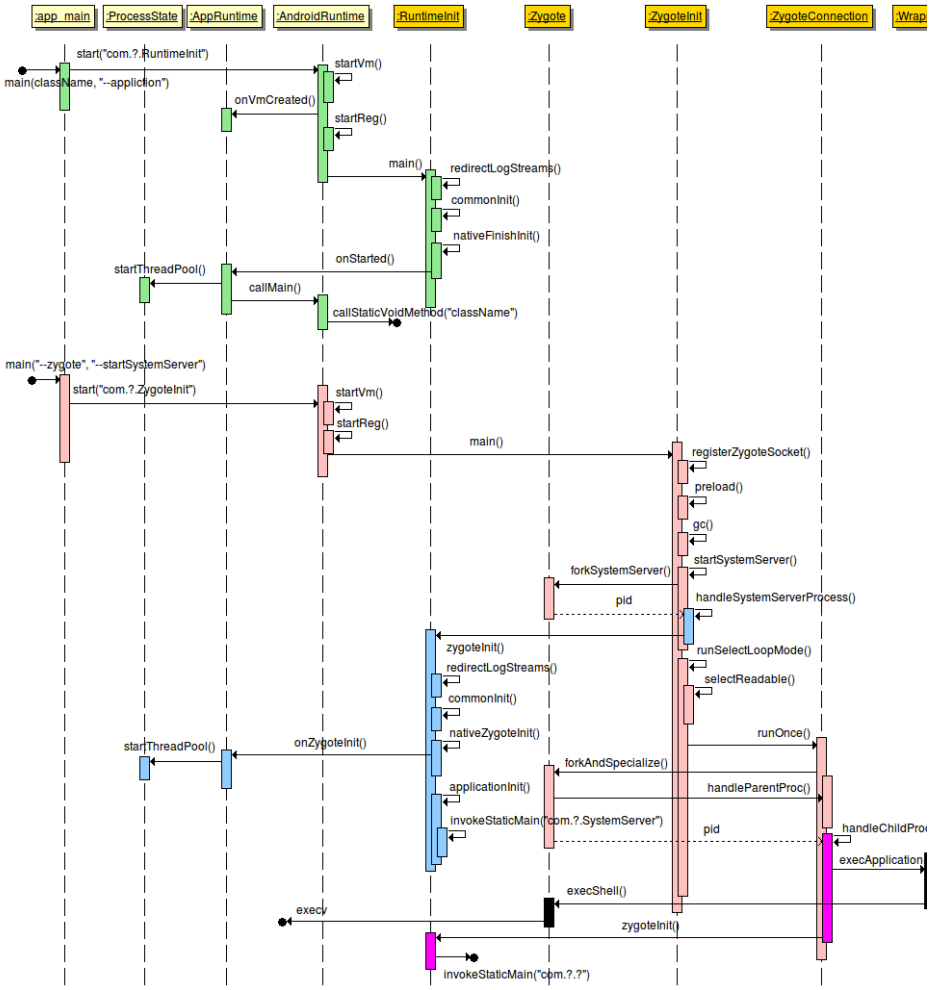
*In computer programming, a **runtime library** is the API used by a compiler to invoke some of the behaviors of a runtime system. The runtime system implements the execution model and other fundamental behaviors of a programming language. The compiler inserts calls to the runtime library into the executable binary. During execution (run time) of that computer program, execution of those calls to the runtime library cause communication between the application and the 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



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。

从类图中看出，JavaVM 和 JNIEnv 是连结 AndroidRuntim 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，等等。本文无法深入，只作简单介绍。

- ## JavaVM 和 JNIENV

```
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*);
};
```

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) {

```
    Thread* self = dvmThreadSelf(); //获取当前线程对象。
    if (version < JNI_VERSION_1_1 || version > JNI_VERSION_1_6) {
        return JNI_EVERSION;
    } //检查版本号，Android 4.3对应 1.6

    ...
        *env = (void*) dvmGetThreadJNIEnv(self);  //很简单，见最下面一行
        dvmChangeStatus(self, THREAD_NATIVE);
    return (*env != NULL) ? JNI_OK : JNI_EDETACHED;
}

INLINE JNIEnv* dvmGetThreadJNIEnv(Thread* self) { return self->jniEnv; }
```

很简单嘛，原来就是从当前所在线程的结构体对象里读取即可，这里面好像没JavaVM什么事吗，为什么当参数传入？不知道，也许Google留作将来扩展？但不管怎么，要想调用GetEnv,还是需要JavaVM。将来要写JNI代码的同学可以参考以下的代码看如何获取JavaVM和JniENV.

```
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;
}
```

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

- **startVM()**

```
int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv)
{
    property_get("dalvik.vm.checkjni", propBuf, "");
    ...
    initArgs.version = JNI_VERSION_1_4;
    ...
    //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");
        goto bail;
    }
    ...
}

jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args) {
    memset(&gDvm, 0, sizeof(gDvm)); /* 这里才是真正的VM结构体*/
    JavaVMExt* pVM = (JavaVMExt*) calloc(1, sizeof(JavaVMExt));
    pVM->funcTable = &gInvokeInterface; //初始化函数指针
    pVM->envList = NULL;
    ...
    gDvmJni.jniVm = (JavaVM*) pVM; //native代码接触的JavaVM原来只是JniVm而已
    JNIEnvExt* pEnv = (JNIEnvExt*) dvmCreateJNIEnv(NULL); //创建JNIEnv,因为接下来的虚拟机初始化需要访问C/C++实现
    /* 开始初始化. */
    gDvm.initializing = true;
    std::string status =
            dvmStartup(argc, argv.get(), args->ignoreUnrecognized, (JNIEnv*)pEnv);
```

```
    gDvm.initializing = false;

    dvmChangeStatus(NULL, THREAD_NATIVE);
    *p_env = (JNIEnv*) pEnv;
    *p_vm = (JavaVM*) pVM;
    return JNI_OK;
```

```
std::string dvmStartup(int argc, const char* const argv[],
        bool ignoreUnrecognized, JNIEnv* pEnv)
{
    /*
     * 检查输入并准备初始化参数
     */
    int cc = processOptions(argc, argv, ignoreUnrecognized);
    ...


    /* 真正初始化开始，初始化各个内部模块，并创建一系列线程*/
    if (!dvmAllocTrackerStartup()) {
        return "dvmAllocTrackerStartup failed";
    }

    if (!dvmGcStartup()) {
        return "dvmGcStartup failed";
    }

    if (!dvmThreadStartup()) {
        return "dvmThreadStartup failed";
    }

    if (!dvmInlineNativeStartup()) {
        return "dvmInlineNativeStartup";
    }

    if (!dvmRegisterMapStartup()) {
        return "dvmRegisterMapStartup failed";
    }

    if (!dvmInstanceofStartup()) {
        return "dvmInstanceofStartup failed";
    }

    if (!dvmClassStartup()) {
        return "dvmClassStartup failed";
    }

    if (!dvmNativeStartup()) {
        return "dvmNativeStartup failed";
    }

    if (!dvmInternalNativeStartup()) {
        return "dvmInternalNativeStartup failed";
    }

    if (!dvmJniStartup()) {
        return "dvmJniStartup failed";
    }

    if (!dvmProfilingStartup()) {
        return "dvmProfilingStartup failed";
    }

    if (!dvmInitClass(gDvm.classJavaLangClass)) {
        return "couldn't initialized java.lang.Class";
    }

    if (!registerSystemNatives(pEnv)) {
        return "couldn't register system natives";
    }

    if (!dvmCreateStockExceptions()) {
        return "dvmCreateStockExceptions failed";
    }

    if (!dvmPrepMainThread()) {
        return "dvmPrepMainThread failed";
```

```
    }

    if (dvmReferenceTableEntries(&dvmThreadSelf()->internalLocalRefTable) != 0)
    {
        ALOGW("Warning: tracked references remain post-initialization");
        dvmDumpReferenceTable(&dvmThreadSelf()->internalLocalRefTable, "MAIN");
    }

    if (!dvmDebuggerStartup()) {
        return "dvmDebuggerStartup failed";
    }

    if (!dvmGcStartupClasses()) {
        return "dvmGcStartupClasses failed";
    }

    if (gDvm.zygote) {
        if (!initZygote()) {
            return "initZygote failed";
        }
    } else {
        if (!dvmInitAfterZygote()) {
            return "dvmInitAfterZygote failed";
        }
    }
    return "";
}
```

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：

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

```
 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    //
```

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;
}
```

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。

```
int androidCreateRawThreadEtc(android_thread_func_t entryFunction,
                              void *userData,
                              const char* threadName,
                              int32_t threadPriority,
                              size_t threadStackSize,
                              android_thread_id_t *threadId)
{
    ...
    int result = pthread_create(&thread, &attr,android_pthread_entry)entryFunction,
userData);
    ...
}
```

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

```
#AndroidRuntime.cpp
int AndroidRuntime::javaCreateThreadEtc(
                                android_thread_func_t entryFunction,
                                void* userData,
                                const char* threadName,
                                int32_t threadPriority,
                                size_t threadStackSize,
                                android_thread_id_t* threadId)
{
     args[0] = (void*) entryFunction; //将entryFunc 暂存在args[0]
     args[1] = userData;
     args[2] = (void*) strdup(threadName);
     result =AndroidCreateRawThreadEtc(AndroidRuntime::javaThreadShell, args,
threadName, threadPriority, threadStackSize, threadId); //entryFunc变成
javaThreadShell.
    return result;
}
```

```
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

    JNIEnv* env;
    /* 跟 VM 绑定 */
```

```
    if (javaAttachThread(name, &env) != JNI_OK)
        return -1;

    /* 运行真正的'entryFunc' */
    result = (*(android_thread_func_t)start)(userData);


    /* unhook us */
    javaDetachThread();
    ...
    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。

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.

```
bool dvmAttachCurrentThread(const JavaVMAttachArgs* pArgs, bool isDaemon)

{
    Thread* self = NULL;
    ...
    self = allocThread(gDvm.stackSize);
    ...
    self->jniEnv = dvmCreateJNIEnv(self);
    ...
    gDvm.threadList->next = self;
    ...
    threadObj = dvmAllocObject(gDvm.classJavaLangThread, ALLOC_DEFAULT);
    vmThreadObj = dvmAllocObject(gDvm.classJavaLangVMThread, ALLOC_DEFAULT);
    ...
    self->threadObj = threadObj;
    ...
}
```

完了，就开始注册本地的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 const RegJNIRec gRegJNI[] = {
   {register_android_debug_JNITest},
   {register_com_android_internal_os_RuntimeInit}.
    ...
}
```

每个 register_xxx是一个函数指针

```
int jniRegisterNativeMethods(
      C_JNIEnv* env,
      const char* className,
      const JNINativeMethod* gMethods,
      int numMethods);
```

RegisterNativeMethods 在VM内部到底发生了什么？ 同样，这里只需要知道以下几点即可：

gRegJni[]

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。

3. 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：

```
Runtime_init.java
...
Thread.setDefaultUncaughtExceptionHandler(new UncaughtHandler());
...

private static class UncaughtHandler implements Thread.UncaughtExceptionHandler {
        public void uncaughtException(Thread t, Throwable e) {
            try {
                // Don't re-enter -- avoid infinite loops if crash-reporting crashes.
                if (mCrashing) return;
                mCrashing = true;
                if (mApplicationObject == null) {
                    Slog.e(TAG, "*** FATAL EXCEPTION IN SYSTEM PROCESS: " +
t.getName(), e);
                } else {
                    Slog.e(TAG, "FATAL EXCEPTION: " + t.getName(), e);
                }
                ActivityManagerNative.getDefault().handleApplicationCrash(
                        mApplicationObject, new ApplicationErrorReport.CrashInfo(e));
            } catch (Throwable t2) {
              ...
            } finally {
                Process.killProcess(Process.myPid());
                System.exit(10);
            }
        }
    }
```

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。

```
    virtual void onStarted()
    {
        sp<ProcessState> proc = ProcessState::self();
        proc->startThreadPool();

        AndroidRuntime* ar = AndroidRuntime::getRuntime();
```

```
        ar->callMain(mClassName, mClass, mArgC, mArgV);

        IPCThreadState::self()->stopProcess();
    }
```

而 RuntimeInit::ZygoteInit() 则会调到 app_main.cpp 的 onZygoteInit()

```
virtual void onZygoteInit()
    {
        // Re-enable tracing now that we're no longer in Zygote.
        atrace_set_tracing_enabled(true);

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

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:

```
    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：

```
private static final String ANDROID_SOCKET_ENV = "ANDROID_SOCKET_zygote";
private static void registerZygoteSocket() {
```

```
            String env = System.getenv(ANDROID_SOCKET_ENV);
             fileDesc = Integer.parseInt(env);
             ...
             sServerSocket = new LocalServerSocket(
                       createFileDescriptor(fileDesc));
             ...
}
```

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的启动过程中。

Preload

preload() 做了两件事情：

```
   static void preload() {
       preloadClasses();
       preloadResources();
   }
```

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。

GC

```
  static void gc() {
        final VMRuntime runtime = VMRuntime.getRuntime();
        System.gc();
        runtime.runFinalizationSync();
        System.gc();
        runtime.runFinalizationSync();
        System.gc();
        runtime.runFinalizationSync();
    }
```

为什么调了3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。

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，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

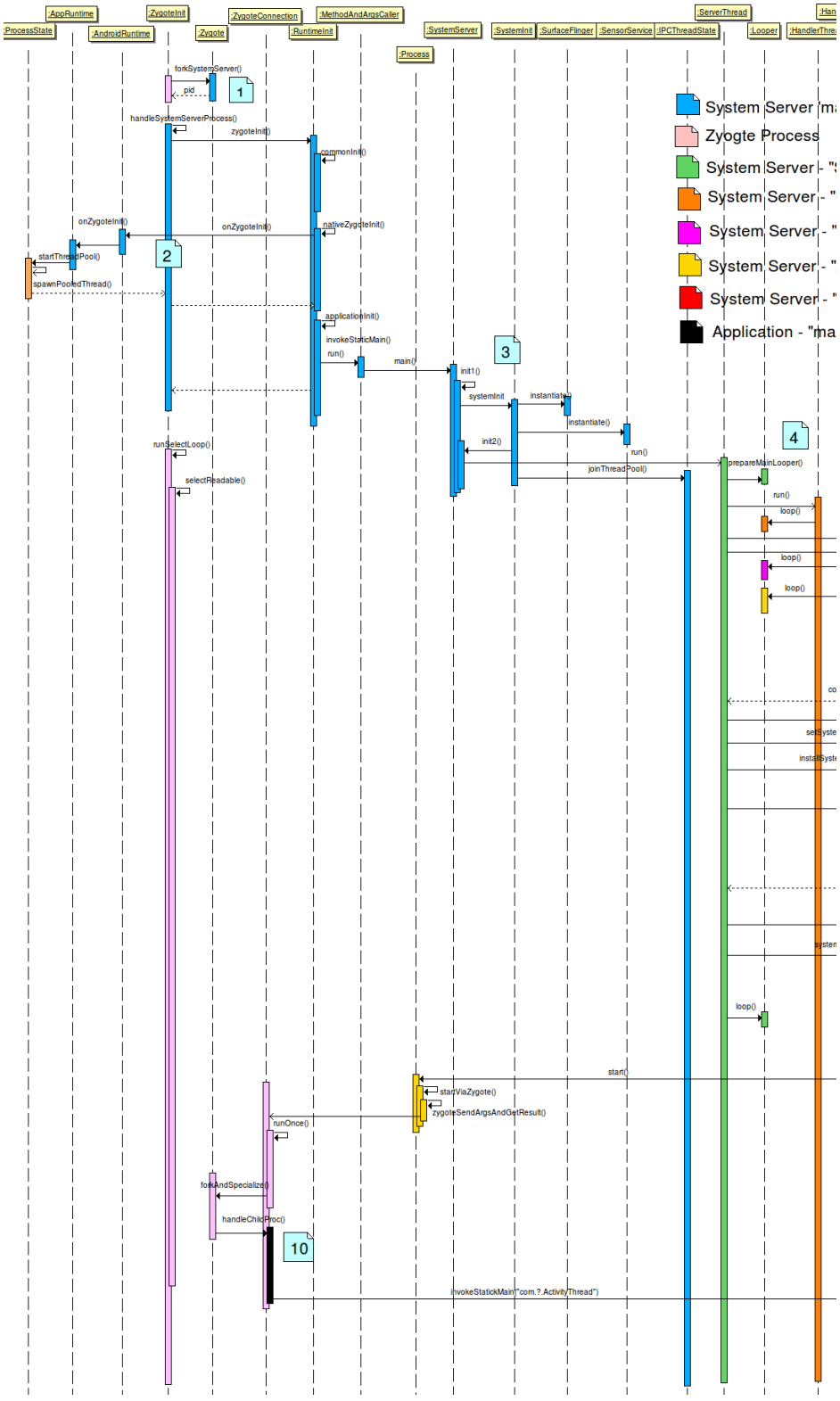
```
system    1176  1163  774376 51144 00000000 b76c4ab6 S SensorService
system    1177  1163  774376 51144 00000000 b76c49eb S er.ServerThread
system    1178  1163  774376 51144 00000000 b76c49eb S UI
system    1179  1163  774376 51144 00000000 b76c49eb S WindowManager
system    1180  1163  774376 51144 00000000 b76c49eb S ActivityManager
system    1182  1163  774376 51144 00000000 b76c4d69 S ProcessStats
system    1183  1163  774376 51144 00000000 b76c2bb6 S FileObserver
system    1184  1163  774376 51144 00000000 b76c49eb S PackageManager
system    1185  1163  774376 51144 00000000 b76c49eb S AccountManagerS
system    1187  1163  774376 51144 00000000 b76c49eb S PackageMonitor
system    1188  1163  774376 51144 00000000 b76c4ab6 S UEventObserver
system    1189  1163  774376 51144 00000000 b76c4d69 S BatteryUpdateTi
system    1190  1163  774376 51144 00000000 b76c49eb S PowerManagerSer
```

```
system    1191  1163  774376 51144 00000000 b76c2ff6 S AlarmManager
system    1192  1163  774376 51144 00000000 b76c4d69 S SoundPool
system    1193  1163  774376 51144 00000000 b76c4d69 S SoundPoolThread
system    1194  1163  774376 51144 00000000 b76c49eb S InputDispatcher
system    1195  1163  774376 51144 00000000 b76c49eb S InputReader
system    1196  1163  774376 51144 00000000 b76c49eb S BluetoothManage
system    1197  1163  774376 51144 00000000 b76c49eb S MountService
system    1198  1163  774376 51144 00000000 b76c4483 S VoldConnector
system    1199  1163  774376 51144 00000000 b76c49eb S CallbackHandler
system    1201  1163  774376 51144 00000000 b76c4483 S NetdConnector
system    1202  1163  774376 51144 00000000 b76c49eb S CallbackHandler
system    1203  1163  774376 51144 00000000 b76c49eb S NetworkStats
system    1204  1163  774376 51144 00000000 b76c49eb S NetworkPolicy
system    1205  1163  774376 51144 00000000 b76c49eb S WifiP2pService
system    1206  1163  774376 51144 00000000 b76c49eb S WifiStateMachin
system    1207  1163  774376 51144 00000000 b76c49eb S WifiService
system    1208  1163  774376 51144 00000000 b76c49eb S ConnectivitySer
system    1214  1163  774376 51144 00000000 b76c49eb S WifiManager
system    1215  1163  774376 51144 00000000 b76c49eb S Tethering
system    1216  1163  774376 51144 00000000 b76c49eb S CaptivePortalTr
system    1217  1163  774376 51144 00000000 b76c49eb S WifiWatchdogSta
system    1218  1163  774376 51144 00000000 b76c49eb S NsdService
system    1219  1163  774376 51144 00000000 b76c4483 S mDnsConnector
system    1220  1163  774376 51144 00000000 b76c49eb S CallbackHandler
system    1227  1163  774376 51144 00000000 b76c49eb S SyncHandlerThre
system    1228  1163  774376 51144 00000000 b76c49eb S AudioService
system    1229  1163  774376 51144 00000000 b76c49eb S backup
system    1233  1163  774376 51144 00000000 b76c49eb S AppWidgetServic
system    1240  1163  774376 51144 00000000 b76c4d69 S AsyncTask #1
system    1244  1163  774376 51144 00000000 b76c42a3 S Thread-64
system    1284  1163  774376 51144 00000000 b76c4d69 S AsyncTask #2
system    1316  1163  774376 51144 00000000 b76c2bb6 S UsbService host
system    1319  1163  774376 51144 00000000 b76c4d69 S watchdog
system    1330  1163  774376 51144 00000000 b76c49eb S LocationManager
system    1336  1163  774376 51144 00000000 b76c2ff6 S Binder_3
system    1348  1163  774376 51144 00000000 b76c49eb S CountryDetector
system    1354  1163  774376 51144 00000000 b76c49eb S NetworkTimeUpda
system    1360  1163  774376 51144 00000000 b76c2ff6 S Binder_4
system    1391  1163  774376 51144 00000000 b76c2ff6 S Binder_5
system    1395  1163  774376 51144 00000000 b76c2ff6 S Binder_6
system    1397  1163  774376 51144 00000000 b76c2ff6 S Binder_7
system    1516  1163  774376 51144 00000000 b76c4d69 S SoundPool
system    1517  1163  774376 51144 00000000 b76c4d69 S SoundPoolThread
system    1692  1163  774376 51144 00000000 b76c4d69 S AsyncTask #3
system    1694  1163  774376 51144 00000000 b76c4d69 S AsyncTask #4
system    1695  1163  774376 51144 00000000 b76c4d69 S AsyncTask #5
system    1791  1163  774376 51144 00000000 b76c4d69 S pool-1-thread-1
system    2758  1163  774376 51144 00000000 b76c4d69 S AudioTrack
system    2829  1163  774376 51144 00000000 b76c49eb S KeyguardWidgetP
```

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。

4. 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

```
void ProcessState::spawnPooledThread(bool isMain)
{
    if (mThreadPoolStarted) {
        String8 name = makeBinderThreadName(); //"Binder_1"
        sp<Thread> t = new PoolThread(isMain);
        t->run(name.string());
    }
```

```
}
virtual bool threadLoop() {

     IPCThreadState::self()->joinThreadPool(mIsMain); //阻塞知道被Binder driver唤醒
     return false;
  }
```

3. 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：

```
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;
}
```

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。

4. 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

5. 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。

6. 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。

7. 毫无疑问，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。启动Home的代码如下

```
boolean startHomeActivityLocked(int userId) {
        ...
        Intent intent = new Intent(...);
        ... intent.addCategory(Intent.CATEGORY_HOME);
        ...
        mMainStack.startActivityLocked(null, intent, null, aInfo,null, null, 0, 0, 0,
null, 0, null, false, null);
        ...
}
```

8. 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：

```
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;
    }
}
```

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。

5. 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ZygoteInit.java 的runSelectLoop()里，

```
    private static void runSelectLoop() throws MethodAndArgsCaller {

        ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();
        ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>();
        FileDescriptor[] fdArray = new FileDescriptor[4];
        fds.add(sServerSocket.getFileDescriptor());
        peers.add(null);

        while (true) {
            /* 在fork 子进程之前做把GC而不是在每个子进程里自己做，可以提高效率，但也不能每次都做因为GC还是很耗时的。*/
            if (loopCount <= 0) {
                gc();
                loopCount = GC_LOOP_COUNT;
            } else {
                loopCount--;
            }

            try {
                fdArray = fds.toArray(fdArray);
                index = selectReadable(fdArray); //select 阻塞等待
            } catch (IOException ex) {
               ...
            }

            /* 接收新的连接 */
            else if (index == 0) {
                ZygoteConnection newPeer = acceptCommandPeer();
                peers.add(newPeer);
                fds.add(newPeer.getFileDesciptor());
            } else {
                boolean done;
                /* 在这里完成fork操作 */
                done = peers.get(index).runOnce();
                if (done) {
                    peers.remove(index);
                    fds.remove(index);
                }
            }
        }
    }
```

```
    boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {
        ...
        try {
            args = readArgumentList();
            descriptors = mSocket.getAncillaryFileDescriptors();
        } catch (IOException ex) {
            ...
        }

        FileDescriptor childPipeFd = null;
        FileDescriptor serverPipeFd = null;

        /*
        if (parsedArgs.runtimeInit && parsedArgs.invokeWith != null) {

             FileDescriptor[] pipeFds = Libcore.os.pipe();
             childPipeFd = pipeFds[1];
             serverPipeFd = pipeFds[0];
             ZygoteInit.setCloseOnExec(serverPipeFd, true);
         }

try {
            parsedArgs = new Arguments(args);
            applyUidSecurityPolicy(parsedArgs, peer, peerSecurityContext);
            ...
```

```
            pid = Zygote.forkAndSpecialize(parsedArgs.uid,parsedArgs.gid,
parsedArgs.gids,parsedArgs.debugFlags, rlimits, parsedArgs.mountExternal,
parsedArgs.seInfo,parsedArgs.niceName);
        } catch (IOException ex) {
            ...
        }

        try {
            if (pid == 0) {
                // 子进程，将serverFd释放
                serverPipeFd = null;
                handleChildProc(parsedArgs, descriptors, childPipeFd, newStderr);
            } else {
                // 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);
        }
    }
```

Android 应用的启动在handleChildProc里完成：

```
    private void handleChildProc(Arguments parsedArgs,
            FileDescriptor[] descriptors, FileDescriptor pipeFd, PrintStream newStderr)
            throws ZygoteInit.MethodAndArgsCaller {

        closeSocket(); // 不需要服务端的socket
        ZygoteInit.closeServerSocket();
        ...
        if (parsedArgs.runtimeInit) { //从Process.java 来的都为true
            if (parsedArgs.invokeWith != null) {
                WrapperInit.execApplication(parsedArgs.invokeWith,
                        parsedArgs.niceName, parsedArgs.targetSdkVersion,
                        pipeFd, parsedArgs.remainingArgs); // 启动命令行程序
            } else {
                RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion,
                        parsedArgs.remainingArgs); // 几乎所有的应用启动都走这条路
            }
        } else {
            ...
            if (parsedArgs.invokeWith != null) {
                WrapperInit.execStandalone(parsedArgs.invokeWith,
                        parsedArgs.classpath, className, mainArgs);
            } else {
                ...
                try {
                    ZygoteInit.invokeStaticMain(cloader, className, mainArgs);
                } catch (RuntimeException ex) {
                    ...
                }
            }
        }
    }
```

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);
    }
```

细心的读者可能会问这么几个问题：

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

```
    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) {
            ...
        }
    }
```

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)
```

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。

6. 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：dalvik/vm/native/dalvik_system_zygote.cpp 中，

```
static void Dalvik_dalvik_system_Zygote_forkSystemServer(
        const u4* args, JValue* pResult){
    ...
    pid_t pid;
    pid = forkAndSpecializeCommon(args, true);
    ...
    if (pid > 0) {
        int status;
        gDvm.systemServerPid = pid;
        /* 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);
            kill(getpid(), SIGKILL);
        }
    }
    RETURN_INT(pid);
}


/* 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);
        }
    }
```

```
    ...
}
```

```
static void Dalvik_dalvik_system_Zygote_fork(const u4* args, JValue* pResult)
{
    pid_t pid;
    ...
    setSignalHandler(); //signalHandler 在这里注册
    ...
    pid = fork();
    ...
    RETURN_INT(pid);
}
```

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.

« 上一篇： 图解Android - Binder 和 Service
» 下一篇： 推荐书单

posted @ 2013-10-25 00:09 漫天尘沙 阅读(11514) 评论(6) 编辑 收藏

评论列表

#1楼 2013-11-19 23:10 qingyuanxingsi

博主好,能否把该文章中的两张图的高清版本发到我邮箱,qingyuanxingsi@163.com,谢啦

支持(0) 反对(0)

#2楼 [楼主] 2013-11-20 22:31 漫天尘沙

已经是高清图了，你点击他们就看到了，另外，你也可以到https://github.com/samchen2009/android_uml 去下其他图，或者按照上面的说明自己画UML图。

支持(0) 反对(0)

#3楼 2013-11-23 18:05 wangmuy

博主好强大~
另，在 3. ZygoteInit 里面关于 ZygoteInit 和 RuntimeInit 的说法和图片里的有点出入吧：这里的ZygoteInit是指JAVA类，不是RuntimeInit类里面的zygoteInit()方法。ZygoteInit#main方法里没有调用到 redirectLogStreams() 和 commonInit()，zygote进程本身没有redirectLogStream()和commonInit(), systemserver进程才是调了RuntimeInit#zygoteInit()

支持(0) 反对(0)

#4楼 [楼主] 2013-11-24 15:54 漫天尘沙

谢谢wangmuy的纠正，的确，原文的描述混淆了ZygoteInit::main() RuntimeInit::main() 和 RuntimeInit::ZygoteInit()三者的关系，已更新该节，如有错误请再指正，谢谢。

支持(0) 反对(0)

#5楼 2015-01-19 20:06 隐于市

博主你好，请问你这图是用啥工具画的？

支持(0) 反对(0)

#6楼 2015-01-29 09:30 dannycai

结构相当清晰，很强大。。。

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

· 谈谈Model S的设计失误与Model X的车门及Autopilot
· 快速将C#类型转成TypeScript介面定义
· 你的代码活着吗？
» 更多新闻...

最新知识库文章：
· Docker简介
· Docker简明教程
· Git协作流程
· 企业计算的终结
· 软件开发的核心
» 更多知识库文章...