内容目录

# 第2章 Android系统启动

## 2.2 Zygote进程启动过程

### 2.2.2启动脚本

Zygote启动脚本的位置，system/core/rootdir/

build/make/target/product/core\_64\_bit.mk

PRODUCT\_DEFAULT\_PROPERTY\_OVERRIDES += ro.zygote=zygote64\_32

M01\_AE:/ # getprop ro.zygote

zygote64\_32

通过ro.zygote来判断启动哪个脚本

init.zygote32\_64.rc

init.zygote32.rc

init.zygote64\_32.rc

init.zygote64.rc

init.zygote64\_32.rc

//主模式

service zygote /system/bin/app\_process64 -Xzygote /system/bin --zygote --start-system-server --socket-name=zygote

class main

priority -20

user root

group root readproc

socket zygote stream 660 root system

onrestart write /sys/android\_power/request\_state wake

onrestart write /sys/power/state on

onrestart restart audioserver

onrestart restart cameraserver

onrestart restart media

onrestart restart netd

onrestart restart wificond

writepid /dev/cpuset/foreground/tasks

//辅模式

service zygote\_secondary /system/bin/app\_process32 -Xzygote /system/bin --zygote --socket-name=zygote\_secondary --enable-lazy-preload

class main

priority -20

user root

group root readproc

socket zygote\_secondary stream 660 root system

onrestart restart zygote

writepid /dev/cpuset/foreground/tasks

### 2.2.3 启动过程

frameworks/base/cmds/app\_process/app\_main.cpp

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

...

while (i < argc) {

const char\* arg = argv[i++];

if (strcmp(arg, "--zygote") == 0) {

zygote = true;//如果当前运行在Zygote进程中，则将zygote设置为true

niceName = ZYGOTE\_NICE\_NAME;

} else if (strcmp(arg, "--start-system-server") == 0) {//如果当前运行在SystemServer进程中，则将 startSystemServer设置为true

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) {//如果运行在Zygote进程中

runtime.start("com.android.internal.os.ZygoteInit", args, zygote);

...

}

frameworks/base/core/jni/AndroidRuntime.cpp

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

{

...

env->CallStaticVoidMethod(startClass, startMeth, strArray);//通过JNI调用ZygoteInit的main方法，Zygote开创了Java框架层。

...

}

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

public static void main(String argv[]) {

…

try {

…

//创建一个Server端的Socket，socketName的值为"zygote"

zygoteServer.registerServerSocket(socketName);//1

//预加载类和资源

preload(bootTimingsTraceLog);//2，本书未分析

//启动SystemServer进程，书中是startSystemServer进程

Runnable r = forkSystemServer(abiList, socketName, zygoteServer);//3

r.run();//mMethod.invoke(null, new Object[] { mArgs });

//等待AMS请求

caller = zygoteServer.runSelectLoop(abiList);//4

}

}

**1. registerServerSocket**

在Zygote进程将SystemServer进程启动后，就会在这个服务器端的Socket上等待AMS请求Zygote进程来创建新的应用程序进程。

void registerServerSocket(String socketName) {

final String fullSocketName = ANDROID\_SOCKET\_PREFIX + socketName;//1 "ANDROID\_SOCKET\_" + "zygote" = ANDROID\_SOCKET\_zygote

String env = System.getenv(fullSocketName);//2 得到Socket的环境变量的值 -- /dev/socket/zygote

fileDesc = Integer.parseInt(env);//3 将Socket环境变量的值转换为文件描述符的参数

FileDescriptor fd = new FileDescriptor();//4 创建文件描述符

fd.setInt$(fileDesc);//5 传入fileDesc

mServerSocket = new LocalServerSocket(fd);//6 创建服务端Socket

}

**2. 启动SystemServer进程**

private static Runnable forkSystemServer(String abiList, String socketName, ZygoteServer zygoteServer) {

...

String args[] = { //1 创建args数组，这个数组用来保存启动SystemServer的启动参数

"--setuid=1000", //用户id被设置为1000

"--setgid=1000", //用户组id被设置为1000

"--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,1021,1023,1032,3001,3002,3003,3006,3007,3009,3010", //拥有这些用户组的权限

"--capabilities=" + capabilities + "," + capabilities, //设置 capabilities

"--nice-name=system\_server", //进程名为system\_server

"--runtime-args",

"com.android.server.SystemServer",//启动的类名

};

parsedArgs = new ZygoteConnection.Arguments(args);//2 将args数组封装成Arguments对象并供注释3处的forkSystemServer的函数调用

int pid;

pid = Zygote.forkSystemServer(//3 通过fork函数在当前进程创建一个子进程，也就是SystemServer进程

parsedArgs.uid, parsedArgs.gid,

parsedArgs.gids,

parsedArgs.debugFlags,

null,

parsedArgs.permittedCapabilities,

parsedArgs.effectiveCapabilities);

//当前代码逻辑运行在子进程中

if (pid == 0) {

if (hasSecondZygote(abiList)) {

waitForSecondaryZygote(socketName);

}

zygoteServer.closeServerSocket();

//处理SystemServer进程，2.3节会介绍

return handleSystemServerProcess(parsedArgs);//4

}

}

**3. runSelectLoop**

启动SystemServer进程后，会执行ZygoteServer的runSelectLoop方法

Runnable runSelectLoop(String abiList) {

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

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

注释1处的mServerSocket就是我们在registerServerSocket函数中创建的服务端Socket，调用mServerSocket.getFileDescriptor()函数获得该Socket的fd字段的值并添加到fd列表fds中，在fds中索引为0

fds.add(mServerSocket.getFileDescriptor());//1

//接下来无限循环等待AMS的请求

while (true) {

for (int i = 0; i < pollFds.length; ++i) {//2 通过遍历将fds存储的信息转移到pollFds数组中

...

}

try {

Os.poll(pollFds, -1); //处理轮询状态，当pollFds有事件到来时则往下执行，否则阻塞在这里

} catch (ErrnoException ex) {

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

}

//采用IO多路复用机制，当接收到客户端发出的连接请求时或者数据处理请求到来时则往下执行，否则进入continue跳出本次循环。

for (int i = pollFds.length - 1; i >= 0; --i) {//3

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

continue;

}

//索引为0，即为sServerSocket，表示接收到客户端发来的连接请求。

if (i == 0) {

ZygoteConnection newPeer = acceptCommandPeer(abiList);//4

peers.add(newPeer);

fds.add(newPeer.getFileDesciptor());

} else {//索引不为0，表示通过Socket接收来自对端的数据，并执行相应的操作。说明AMS向Zygote进程发送了一个创建应用进程的请求

try {

ZygoteConnection connection = peers.get(i);

final Runnable command = connection.processOneCommand(this);//5 创建一个新的应用进程

...

if (connection.isClosedByPeer()) {//创建成功后将这个连接从Socket连接列表peers和fd列表fds中清除

connection.closeSocket();

peers.remove(i);

fds.remove(i);

}

} catch (Exception e) {

...

}

}//end 3

}//while(true)

}

## 2.3 SystemServer处理过程

### 2.3.1 Zygote处理SystemServer进程

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

private static Runnable forkSystemServer(String abiList, String socketName, ZygoteServer zygoteServer) {

…

//当前运行在SystemServer进程中

if (pid == 0) {

if (hasSecondZygote(abiList)) {

waitForSecondaryZygote(socketName);

}

//关闭zygote进程创建的Socket

zygoteServer.closeServerSocket();//1 Zygote创建的Socket对SystemServer没用处，故关闭

return handleSystemServerProcess(parsedArgs);//2

}

}

private static Runnable handleSystemServerProcess(ZygoteConnection.Arguments parsedArgs) {

if (parsedArgs.invokeWith != null) {

…

} else {

ClassLoader cl = null;

cl = createPathClassLoader(systemServerClasspath, parsedArgs.targetSdkVersion);//1 创建了PathClassLoader，第12章会介绍

return ZygoteInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.remainingArgs, cl);//2 ZygoteInit的 zygoteInit方法

}

}

public static final Runnable zygoteInit(int targetSdkVersion, String[] argv, ClassLoader classLoader) {

if (RuntimeInit.DEBUG) {

Slog.d(RuntimeInit.TAG, "RuntimeInit: Starting application from zygote");

}

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "ZygoteInit");

RuntimeInit.redirectLogStreams();

RuntimeInit.commonInit();

//启动Binder线程池，这样SystemServer进程就可以使用Binder与其他进程进行通信了

ZygoteInit.nativeZygoteInit();//1

//进入SystemServer的main方法

return RuntimeInit.applicationInit(targetSdkVersion, argv, classLoader);//2

}

**1. 启动binder线程池**

frameworks/base/core/jni/AndroidRuntime.cpp

static void com\_android\_internal\_os\_ZygoteInit\_nativeZygoteInit(JNIEnv\* env, jobject clazz)

{

gCurRuntime->onZygoteInit();

}

frameworks/base/cmds/app\_process/app\_main.cpp

virtual void onZygoteInit()

{

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

ALOGV("App process: starting thread pool.\n");

proc->startThreadPool();//1 启动Binder线程池

}

**2. 进入SystemServer的main方法**

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

protected static Runnable applicationInit(int targetSdkVersion, String[] argv, ClassLoader classLoader)

→ return findStaticMain(args.startClass, args.startArgs, classLoader)

→ {

cl = Class.forName(className, true, classLoader);//1 通过反射找到SystemServer类，com.android.server.SystemServer，c1得到是SystemServer类

Method m;

m = cl.getMethod("main", new Class[] { String[].class });//2 找到SystemServer的main方法

return new MethodAndArgsCaller(m, argv);//3 返回一个Runnable但是书中的8.0是抛出异常，然后由ZygoteInit.java的main方法捕获，但是8.1.0里是直接调用run了

}

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

r.run();//2.2.3节中，forkSystemServer之后，执行run

→ mMethod.invoke(null, new Object[] { mArgs });//反射动态调用SystemServer的main方法

### 2.3.2 解析SystemServer进程

frameworks/base/services/java/com/android/server/SystemServer.java

public static void main(String[] args) {

new SystemServer().run();

}

private void run() {

try {

Looper.prepareMainLooper();//创建消息Looper

System.loadLibrary("android\_servers");//1 加载了动态库libandroid\_servers.so

performPendingShutdown();

createSystemContext();//创建系统的Context

mSystemServiceManager = new SystemServiceManager(mSystemContext);//2

mSystemServiceManager.setRuntimeRestarted(mRuntimeRestart);

LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);

SystemServerInitThreadPool.get();

} finally {

traceEnd();

}

try {

traceBeginAndSlog("StartServices");

startBootstrapServices();//3 启动引导服务 ActivityManagerService、PowerManagerService、PackageManagerService ...

startCoreServices();//4 启动核心服务 DropBoxManagerService、BatteryService、UsageStatsService、WebViewUpdateService ...

startOtherServices();//5 启动其他服务 CameraService、AlarmManagerService、VrManagerService ...

SystemServerInitThreadPool.shutdown();

}

}

以启动PowerManagerService来进行举例

mPowerManagerService = mSystemServiceManager.startService(PowerManagerService.class);

frameworks/base/services/core/java/com/android/server/SystemServiceManager.java

mServices.add(service);//1 注册Service

service.onStart();//2 启动Service

或者是直接调用main方法，比如 PackageManagerService的main

mPackageManagerService = PackageManagerService.main(mSystemContext, installer, mFactoryTestMode != FactoryTest.FACTORY\_TEST\_OFF, mOnlyCore);

## 2.4 Launcher

Launcher作为Android系统的启动器用于启动应用程序，也用于管理桌面上的快捷图标等资源。

SystemServer.java

private void startOtherServices() {

mActivityManagerService.systemReady(() -> {

Slog.i(TAG, "Making services ready");

traceBeginAndSlog("StartActivityManagerReadyPhase");

mSystemServiceManager.startBootPhase(

SystemService.PHASE\_ACTIVITY\_MANAGER\_READY);

...

}

通过层层调用。。。，ActivityManagerService.java

boolean startHomeActivityLocked(int userId, String reason) {

if (mFactoryTest == FactoryTest.FACTORY\_TEST\_LOW\_LEVEL && mTopAction == null) {//1 判断工厂模式和mTopAction的值，符合要求就继续执行下去

return false;

}

Intent intent = getHomeIntent();//2创建Launcher启动所需的Intent

}

## 11.3 DVM和ART的诞生

M01\_AE:/ # getprop | grep dalvik.vm.lib

[persist.sys.dalvik.vm.lib.2]: [libart.so]

app\_main.cpp → AndroidRuntime.cpp

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

jni\_invocation.Init(NULL);//jni\_invocation的Init函数

if (startVm(&mJavaVM, &env, zygote) != 0) {//启动Java虚拟机

if (startReg(env) < 0) {//为Java虚拟机注册JNI方法

}

libnativehelper/JniInvocation.cpp

library = GetLibrary(library, buffer);

handle\_ = dlopen(library, kDlopenFlags);

## 12.1 JAVA中的ClassLoader

**12.1.1 ClassLoader的类型**

1. Bootstrap ClassLoader，加载JDK的核心类库，java.lang.、java.uti.等

sop(System.getProperty("sun.boot.class.path"));

程序打印

/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/resources.jar:/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/rt.jar:/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/sunrsasign.jar:/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/jsse.jar:/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/jce.jar:/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/charsets.jar:/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/jfr.jar:/usr/lib/jvm/java-8-openjdk-amd64/jre/classes

2. Extensions ClassLoader，加载Java的拓展类

sop(System.getProperty("java.ext.dirs"));//Extensions ClassLoader

程序打印

/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/ext:/usr/java/packages/lib/ext

3. Application ClassLoader，应用程序类加载器，当前程序的Classpath目录，系统属性java.class.path指定的目录

sop(System.getProperty("java.class.path"));//Application ClassLoader

程序打印

.

4. Custom ClassLoader，自定义类加载器，通过继承java.lang.ClassLoader方式来实现自定义的类加载器，2和3也是这么做的

**12.1.2 ClassLoader的继承关系**

sop("How many ClassLoader needs to run a java elf?");

ClassLoader loader = ClassLoaderInJavaDemo.class.getClassLoader();

while(loader != null) {

sop(loader);

loader = loader.getParent();

}

程序打印

How many ClassLoader needs to run a java elf?

sun.misc.Launcher$AppClassLoader@3d4eac69

sun.misc.Launcher$ExtClassLoader@6bc7c054

没有 Bootstrap ClassLoader，因为它是C/C++编写的，我们无法获取该引用， AppClassLoader和 ExtClassLoader也不是继承的关系，他们继承自URLClassLoader

**12.1.3 双亲委托模式**

先自底向上委托父加载器查找，顶层先执行查找，查不到再向下执行，直到最后回交给自身去查找

**12.1.4 自定义ClassLoader--略**