# Android Java

## ART

JNI\_CreateJavaVM(JavaVM\*\* p\_vm, JNIEnv\*\* p\_env)

Runtime::Create()

instance\_ = new Runtime

instance\_->Init()

MemMap::Init()

QuasiAtomic::Startup()

Monitor::Init()

java\_vm\_ = JavaVMExt::Create

Thread::Startup()

Runtime\* runtime = Runtime::Current()

return instance\_

bool started = runtime->Start()

\*p\_env = Thread::Current()->GetJniEnv()

\*p\_vm = runtime->GetJavaVM()

return java\_vm\_.get()

## JNI

### 注册JNI函数

静态方法：根据函数名来建立Java函数和JNI函数之间的关联关系

动态方法：

//native层数据结构方法表

JNINativeMethod {

const char \*name //Java中native函数的名字，不用携带包

路径

const char \*signature //Java函数的签名信息，是参数类型和返

回值类型的组合

void \*fnPtr //JNI层对应函数的函数指针

}

//JNI注册函数

int AndroidRuntime::registerNativeMethods(JNIEnv\* env,

const char\* className, const JNINativeMethod\* gMethods, int numMethods){

jniRegisterNativeMethods(env, className, gMethods, numMethods)

}

int jniRegisterNativeMethods(JNIEnv\* env, const char\* className,

const JNINativeMethod\* gMethods, int numMethods){

(\*env)->FindClass(env, className)

(\*env)->RegisterNatives(env, clazz, gMethods, numMethods)

}

Java层注册函数

System.loadLibrary加载JNI动态库

jint JNI\_OnLoad(JavaVM\* vm, void\* reserved){

register\_android\_media\_MediaPlayer(env)

}

### 数据类型转换

1. 基本数据类型转换

Java类型 native类型 Java类型 native类型

int jint byte jbyte

1. 引用数据类型转换

Java引用类型 native类型 Java引用类型 native类型

all objects jobject int[] jintArray

java.lang.Class实例 jclass

java.lang.String实例 jstring

java.lang.Throwable实例 jthrowable

### JNIEnv和JavaVM

jfieldID //表示Java类的成员变量类型

jmethodID //表示Java类的成员函数类型

JNIEnv{ //与线程相关的代表JNI环境的结构体

//获取Java类的成员变量，

jfieldID (\*GetFieldID)(JNIEnv\*, jclass, const char\*, const char\*)

//获取Java类的成员函数

jmethodID (\*GetMethodID)(JNIEnv\*, jclass, const char\*, const char\*)

//调用Java对象的函数

void (\*CallVoidMethod)(JNIEnv\*, jobject, jmethodID, ...)

//获取java object对象变量值

jobject (\*GetObjectField)(JNIEnv\*, jobject, jfieldID)

//设置java object对象变量值

void (\*SetObjectField)(JNIEnv\*, jobject, jfieldID, jobject)

}

JavaVM{ //虚拟机在JNI层的代表

}

//android\_media\_MediaScanner.cpp

### JNI类型签名

格式：

(参数1类型表示……参数n类型表示)返回值类型标示

Java函数：void processFile(String path, String mimeType)

对应JNI函数签名：

(Ljava/lang/String;Ljava/lang/String;Landroid/media/

MediaScannerClient;)V

当参数是引用类型时，格式是“L包名”，“.”用“/”代替

类型标识示意表

类型标识 Java类型 类型标识 Java类型

B byte I int

L/java/langauageString String

L/java/lang/object Object[]

# init和zygote

## init分析

init是一个进程，是Linux系统中用户空间的第一个进程，入口函数

是system/core/init/init.c中的main

main()

init\_parse\_config\_file("/init.rc") //解析init.rc配置文件

snprintf(tmp, sizeof(tmp), "/init.%s.rc", hardware)

init\_parse\_config\_file(tmp) //解析机器相关的配置文件

### 配置文件init.rc

init.rc文件基本组成单位是section, section分为三种类型，分别由三个关键字(所谓关键字即每一行的第一列)来区分，这三个关键字是

on、service、import；

1. on类型的section表示一系列命令的组合
2. service类型的section表示一个可执行程序
3. import类型的section表示引入另外一个.rc文件

有四种类型的语句:

1. 动作(Actions)

动作（Actions）是一个有名字的指令（commands）序列。每个动作（Actions）都定义一个触发条件（trigger），用于指示什么时候执行这个动作。当与动作的触发器匹配的事件发生时，该动作将被添加到一个即将被执行的队列的队尾（除非它已经在队列中）。

队列中的每一个动作被依次取出执行，动作中的每一个指令也将依次执行。初始化程序（Init）在执行一个动作的各项指令的期间，还需要处理其它操作（比如，设备创建/销毁，属性设置，进程重启）。

动作（Actions）和服务（Services）语句隐含表示一个新的段落（section）的开始。 所有的指令（commands）和选项（options）归属于上方最近的一个段落

一个动作定义的形式如下:

on <trigger>

<command>

<command>

<command>

trigger主要包括：

boot 当/init.conf加载完毕时

<name>=<value> 当<name>被设置为<value>时

device-added-<path> 设备<path>被添加时

device-removed-<path> 设备<path>被移除时

service-exited-<name> 服务<name>退出时

1. 指令（Commands）

命令主要包括：

exec <path> [ <argument> ]\*执行一个<path>指定的程序

export <name> <value>设置一个全局变量

ifup <interface>使网络接口<interface>连接

import <filename>引入其他的配置文件

hostname <name>设置主机名

chdir <directory>切换工作目录

chmod <octal-mode> <path>设置访问权限

chown <owner> <group> <path>设置用户和组

chroot <directory>设置根目录

class\_start <serviceclass>启动类中的service

class\_stop <serviceclass>停止类中的service

domainname <name>设置域名

insmod <path>安装模块

mkdir <path> [mode] [owner] [group]创建一个目录，并可以指定权限，

用户和组

mount <type> <device> <dir> [ <mountoption> ]\*加载指定设备到目录

下

<mountoption> 包括"ro", "rw", "remount", "noatime"

setprop <name> <value>设置系统属性

setrlimit <resource> <cur> <max>设置资源访问权限

start <service>开启服务

stop <service>停止服务

symlink <target> <path>创建一个动态链接

sysclktz <mins\_west\_of\_gmt>设置系统时钟

trigger <event>触发事件

write <path> <string> [ <string> ]\*向<path>路径的文件写入多个<string>

1. 服务（Services）

服务是初始化程序需要启动的一些程序，初始化程序还有可能会在这些程序退出之后重启它们。Services take 一个服务定义的形式如下：

service <name> <pathname> [ <argument> ]\*

<option>

<option>

1. 选项（Options）

option是service的修饰词，主要包括:

critical: 表示如果服务在4分钟内存在多于4次，则系统重启到recovery mode

disabled: 表示服务不会自动启动，需要手动调用名字启动

setEnv <name> <value> 设置启动环境变量

socket <name> <type> <permission> [<user> [<group>]]开启一个unix

域的socket，名字为/dev/socket/<name> , <type>只能是dgram

或者stream,<user>和<group>默认为0

user <username>表示将用户切换为<username>,用户名已经定义好

了，只能是system/root

group <groupname> 表示将组切换为<groupname>

oneshot 表示这个service只启动一次

class <name> 指定一个要启动的类，这个类中如果有多个service，

将会被同时启动。默认的class将会是“default”

onrestart 在重启时执行一条命令

### 解析配置文件

//解析配置文件system/core/rootdir/init.rc

struct {

const char \*name;

int (\*func)(int nargs, char \*\*args);

unsigned char nargs;

unsigned char flags;

} keyword\_info[KEYWORD\_COUNT]

### init main

main()

Parser& parser = Parser::GetInstance()

return static Parser instance

parser.AddSectionParser("service",std::make\_unique<ServiceParser>(

))

section\_parsers\_["service"] = std::move(ServiceParser)

parser.ParseConfig("/init.rc")

ActionManager& am = ActionManager::GetInstance()

return static ActionManager instance

am.QueueEventTrigger("early-init")

trigger\_queue\_.push(std::make\_unique<EventTrigger>("early-

init"))

am.QueueBuiltinAction(console\_init\_action, "console\_init")

auto action = std::make\_unique<Action>(true)

std::vector<std::string> name\_vector{"console\_init"}

action->AddCommand(console\_init\_action, name\_vector)

trigger\_queue\_.push(std::make\_unique<BuiltinTrigger>

(action.get()))

actions\_.emplace\_back(std::move(action))

ServiceManager::GetInstance().IsWaitingForExec()

am.ExecuteOneCommand()

trigger\_queue\_.front()->CheckTriggers(\*action)

return action\_ == &action

action->ExecuteOneCommand(current\_command\_)

ExecuteCommand(cmd)

command.InvokeFunc()

### 解析zygote

//以解析zygote服务为例

//system/core/init/init.c

main()

init\_parse\_config\_file("/init.rc")

parse\_config()

for(;;){

//解析init.rc文件中的一个新的section

parse\_new\_section()

//如果section是service

parse\_service() //搭建了一个service的架子

parse\_line\_service() //根据配置文件内容填充service

结构体

//如果是action

parse\_action()

parse\_line\_action()

}

// system\core\init\init.h

service{

struct listnode slist

const char \*name //service名字

const char \*classname //service所属类的名字

unsigned flags //service属性

pid\_t pid //进程号

struct socketinfo \*sockets //用来描述socket相关信息

struct svcenvinfo \*envvars //创建service运行进程所需的环境变

量信息

struct action onrestart //存储command信息

int nargs //参数个数

char \*args[1] //用来存储参数

}

action{

struct listnode alist //存储所有的action

struct listnode qlist //链接等待执行执行的action

struct listnode tlist //链接等待条件触发的action

unsigned hash;

const char \*name;

struct listnode commands;

struct command \*current;

}

//启动zygote服务

parse\_line\_action()

//根据配置文件的command在keyword\_info[]结构中查找相应的

函数执行，比如do\_class\_start()函数

cmd->func = kw\_func(kw)

//system/core/init/Builtins.c

do\_class\_start()

service\_for\_each\_class(args[1], service\_start\_if\_not\_disabled)

service\_start\_if\_not\_disabled()

service\_start()

fork()

execve()

//解析配置文件init.brava.rc(与硬件平台相关)

get\_hardware\_name(hardware, &revision);

snprintf(tmp, sizeof(tmp), "/init.%s.rc", hardware);

init\_parse\_config\_file(tmp)

parse\_config()

## zygote

//app\_main.cpp

//此时zygote service已经在init.c的main函数中创建好

main()

AppRuntime runtime(argv[0], computeArgBlockSize(argc, argv))

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

startVm(&mJavaVM, &env, zygote) //创建虚拟机

JNI\_CreateJavaVM(&mJavaVM, &env) //art/runtime

Runtime::Create()

Runtime\* runtime = Runtime::Current()

bool started = runtime->Start()

\*p\_env = Thread::Current()->GetJniEnv()

\*p\_vm = runtime->GetJavaVM()

startReg(env) //注册JNI函数

// JNIEnv\* env

//通过JNI调用Java函数，进入Java世界，（通过调用ZygoteInit）

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

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

env->CallStaticVoidMethod(startClass, "main",

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

ZygoteInit::main()

//Java类，在ZygoteInit.java文件中

ZygoteInit {

main()

ZygoteServer zygoteServer = new ZygoteServer()

registerZygoteSocket() //注册zygote用的socket

preloadClasses() //预加载类和资源

startSystemServer() //启动system\_service进程

runSelectLoopMode() //zygote调用这个函数

caller.run()

}

## SystemServer分析

// ZygoteInit.java

startSystemServer()

pid = Zygote.forkSystemServer() //native函数

//native实现在dalvik.system.Zygote.c中

Dalvik\_dalvik\_system\_Zygote\_fork()

setSignalHandler()

pid = fork()

handleSystemServerProcess() //

RuntimeInit.zygoteInit()

// native函数，AndroidRuntime.cpp中

zygoteInitNative() //native层的初始化，

com\_android\_internal\_os\_RuntimeInit\_zygoteInit()

invokeStaticMain(startClass, startArgs) //调用startClass

throw new ZygoteInit.MethodAndArgsCaller() //抛出异常

catch (MethodAndArgsCaller caller) {

caller.run()}

//这个mMethod为com.android.server.SystemServer的main函数

mMethod.invoke()

SystemServer.main()

init1(args) //native函数

android\_server\_SystemServer\_init1()

system\_init()

runtime->callStatic(

"com/android/server

SystemServer", "init2")

init2() //SystemServer.java

//SystemServer.java

SystemServer::main()

new SystemServer().run()

mSystemServiceManager = new

SystemServiceManager(mSystemContext)

startOtherServices()

//com.android.server.wifi.WifiService

mSystemServiceManager.startService(

WIFI\_SERVICE\_CLASS)

# C++智能指针

## 轻量级指针

轻量级指针使用简单的引用计数

//RefBase.h

template <class T>

class LightRefBase{

public:

inline void incStrong(const void\* id) const //增加引用计数值

inline void decStrong(const void\* id) const //减少引用计数值

private:

mutable volatile int32\_t mCount //描述一个对象的引用计数值

}

//轻量级指针的实现类为sp，同时也是强指针的实现类

//RefBase.h

template <typename T>

class sp{

private:

T\* m\_ptr //指向实际引用的对象

}

## 强指针

强指针使用强引用计数

对象的生命周期只受强引用计数控制，弱指针不可以直接操作它所引用的对象

//RefBase.h

class RefBase{

public:

class weakref\_type{}

private:

// class RefBase::weakref\_impl : public RefBase::weakref\_type

class weakref\_impl{

public:

volatile int32\_t mStrong //描述对象的强引用计数

volatile int32\_t mWeak //描述对象的弱引用计数

RefBase\* const mBase //指向所引用对象的地址

volatile int32\_t mFlags //描述对象的生命周期控制方式

}

weakref\_impl\* const mRefs //描述对象的引用计数

}

//强指针的实现类为sp，同时也是轻量级指针的实现类

//RefBase.h

template <typename T>

class sp{

private:

T\* m\_ptr //指向实际引用的对象

}

## 弱指针

弱指针使用弱引用计数

//同样使用了RefBase类

//RefBase.h 弱指针类的实现

template <typename T>

class wp{

public:

sp<T> promote() const //将弱指针升级为强指针后操作所引用的对

象

private:

T\* m\_ptr //指向所引用的对象

weakref\_type\* m\_refs //维护对象的弱引用计数

}

## 常见类

//线程操作

class Thread : virtual public RefBase{

public:

virtual status\_t run()

}

//互斥类，线程同步

class Mutex{

}

//条件类

class Condition{

}

# Logger

## logd

log daemon

## main log

//log\_main.h

#define ALOGD(...) ((void)ALOG(LOG\_DEBUG, LOG\_TAG,

\_\_VA\_ARGS\_\_))

LOG\_PRI(ANDROID\_##priority, tag, \_\_VA\_ARGS\_\_)

android\_printLog(priority, tag, \_\_VA\_ARGS\_\_)

\_\_android\_log\_print(prio, tag, \_\_VA\_ARGS\_\_)

\_\_android\_log\_write()

\_\_android\_log\_buf\_write()

write\_to\_log()

\_\_write\_to\_log\_init() //first call

\_\_write\_to\_log\_initialize()

\_\_write\_to\_log\_daemon()

\_\_write\_to\_log\_daemon() //later call

struct android\_log\_transport\_write logdLoggerWrite {

.node = { &logdLoggerWrite.node, &logdLoggerWrite.node },

.context.sock = -EBADF,

.name = "logd",

.available = logdAvailable,

.open = logdOpen,

.close = logdClose,

.write = logdWrite,

}

## system log

#define SLOGD(...)

((void)\_\_android\_log\_buf\_print(LOG\_ID\_SYSTEM, ANDROID\_LOG\_DEBUG, LOG\_TAG, \_\_VA\_ARGS\_\_))

\_\_android\_log\_buf\_write()

## kernel log

#define LOG(severity) LOG\_TO(DEFAULT, severity)

LOG\_STREAM\_TO(dest, severity)

::android::base::LogMessage().stream()

LogMessageData::GetBuffer()

LogMessage::LogMessage()

: data\_(new LogMessageData(file, line, id, severity, error)){}

android::base::InitLogging(argv, &android::base::KernelLogger)

# Binder

## 工作原理

工作原理: 工作进程会被分别分配一个用户空间的内存线性区和内

核空间的内存线性区。且这两个线性区都被映射在相同的物理内存上。进程对用户空间的内存线性区只有读的权限，用于接收数据。而

binder驱动对进程的内核线性区具有写的权限。当其他进程向工作进程发送数据数据时，发送的数据会被从其他进程的用户空间拷贝到工作进程的内核空间的内存线性区，然后binder驱动会通知工作进程接收数据。由于工作进程的两种内存线性区映射的是相同的物理内存，工作进程只需从自己的用户空间的内存线性区拷贝数据即可。

## Binder驱动程序

### 基础数据结构

//Android/kernel/…/Binder.c

struct binder\_work{ //待处理的工作项

}

struct binder\_node{ //描述一个Binder实体对象

binder\_uintptr\_t ptr //指向service组件内部引用计数的地址

binder\_uintptr\_t cookie //指向service组件的地址

}

struct binder\_ref\_death{ //描述service组件的死亡接收通知

}

// IBinder.h

class IBinder : public virtual RefBase{

}

struct binder\_ref{ //描述一个Binder引用对象

}

struct binder\_buffer{ //描述一个内核缓冲区

}

struct binder\_proc{ //描述一个正在使用Binder的进程

}

struct binder\_thread{ //描述Binder线程池中的一个线程

}

struct binder\_transaction{ //描述进程间通信过程

}

### Binder设备初始化

//Android/kernel/…/binder.c

static int \_\_init binder\_init(void){

}

//打开

static int binder\_open(struct inode \*nodp, struct file \*filp){

}

//内存映射

static int binder\_mmap(struct file \*filp, struct vm\_area\_struct \*vma){

}

//分配内核缓冲区

binder\_alloc\_buf()

//释放内核缓冲区

binder\_free\_buf()

//查询内核缓冲区

binder\_buffer\_lookup()

### binder\_mmap

binder\_mmap(struct file \*filp, struct vm\_area\_struct \*vma)

struct binder\_proc \*proc = filp->private\_data

//指向线性区的操作方法，一个很重要的就是缺页的处理

vma->vm\_ops = &binder\_vm\_ops

//binder\_alloc\_mmap\_handler(struct binder\_alloc \*alloc,

struct vm\_area\_struct \*vma)

struct vm\_struct \*area;

//内核进入这个函数时，就已经预先为此次映射分配好了调用进程

//在用户空间的虚拟地址范围（vma->vm\_start，vma->vm\_end）

struct binder\_buffer \*buffer

//为进程所在的内核空间申请与用户空间同样长度的虚拟地址空

//间，这段空间用于内核来访问和管理binder内存区域

area = get\_vm\_area(vma->vm\_end - vma->vm\_start, VM\_IOREMAP)

//对应内核虚拟地址的开始，即为binder内存的开始地址

proc->buffer = area->addr

//同一块物理内存，内核的虚拟地址同应用空间的虚拟地址的差

proc->user\_buffer\_offset = vma->vm\_start - (uintptr\_t)proc->buffer

//用于存放内核分配的物理页的页描述指针：struct page，每个物

//理页对应这样一个struct page结构

alloc->pages = kzalloc(sizeof(alloc->pages[0]) \*

((vma->vm\_end - vma->vm\_start) / PAGE\_SIZE),

GFP\_KERNEL)

//映射的长度即为binder内存的大小

proc->buffer\_size = vma->vm\_end - vma->vm\_start

//binder\_update\_page\_range(alloc, 1, alloc->buffer,

alloc->buffer + PAGE\_SIZE, vma)

//分配一个物理页，并将该物理页的struct page指针值存放在

proc->pages二维数组中

\*page = alloc\_page(GFP\_KERNEL | \_\_GFP\_HIGHMEM |

\_\_GFP\_ZERO)

//为内核的这段虚拟地址建立虚拟到物理页的映射

//map\_new\_virtual()

map\_kernel\_range\_noflush((unsigned long)page\_addr,

PAGE\_SIZE, PAGE\_KERNEL, page)

//由内核的虚拟地址得到用户空间的虚拟地址

user\_page\_addr =

(uintptr\_t)page\_addr + alloc->user\_buffer\_offset

//为用户空间的这段虚拟地址建立虚拟到物理的映射

//至此用户空间和内核空间都映射到了同一块物理页内存

vm\_insert\_page(vma, user\_page\_addr, page[0])

### binder\_alloc\_new\_buf

binder\_alloc\_new\_buf(struct binder\_alloc \*alloc,

size\_t data\_size,size\_t offsets\_size, size\_t extra\_buffers\_size,

int is\_async)

binder\_alloc\_new\_buf\_locked(alloc, data\_size, offsets\_size,

extra\_buffers\_size, is\_async)

## Binder通信库

//Service组件

//IInterface.h

// Binder本地对象

template<typename INTERFACE>

class BnInterface : public INTERFACE, public BBinder{

}

//Binder代理对象

template<typename INTERFACE>

class BpInterface : public INTERFACE, public BpRefBase{

}

class IInterface : public virtual RefBase{

}

// Binder.h

class BBinder : public IBinder{ //Binder本地对象

}

// BpBinder.h

class BpBinder : public IBinder{ //Binder代理对象

transact(uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t

flags)

IPCThreadState::self()->transact(

mHandle, code, data, reply, flags)

waitForResponse()

talkWithDriver()

bwr.write\_size = outAvail

bwr.write\_buffer = (uintptr\_t)mOut.data()

ioctl(mProcess->mDriverFD,

BINDER\_WRITE\_READ, &bwr)

int32\_t mHandle //client组件的句柄值

}

class BpRefBase : public virtual RefBase{

IBinder\* const mRemote //指向BpBinder对象

}

//IBinder.h

class IBinder : public virtual RefBase{

}

//BBinder和BpBinder通过IPCThreadState类和Binder驱动程序交互

// IPCThreadState.h

class IPCThreadState{

}

## Server Manager

service\_manager.c main() //服务管理的入口函数

main()

binder\_open("/dev/binder",128\*1024) //打开Binder设备/dev/binder

open("/dev/binder", O\_RDWR | O\_CLOEXEC)

ioctl(bs->fd, BINDER\_VERSION, &vers)

mmap(NULL, 128\*1024, PROT\_READ, MAP\_PRIVATE, bs->fd,

0)

binder\_become\_context\_manager() //成为manager

ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr)

binder\_loop(bs, svcmgr\_handler) //处理客户端发过来的请求

readbuf[0] = BC\_ENTER\_LOOPER

binder\_write(bs, readbuf, sizeof(uint32\_t))

ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr)

//for (;;) start

ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr)

binder\_parse(svcmgr\_handler)

// switch(cmd)

case BR\_TRANSACTION

svcmgr\_handler()

binder\_send\_reply //

case BR\_REPLY

case BR\_DEAD\_BINDER

svcmgr\_handler()

// switch(txn->code)

case SVC\_MGR\_GET\_SERVICE

do\_find\_service()

find\_svc()

svclist …

svc\_can\_find()

case SVC\_MGR\_ADD\_SERVICE

do\_add\_service()

//服务注册

int do\_add\_service(struct binder\_state \*bs,

uint16\_t \*s, unsigned len,

void \*ptr, unsigned uid)

## Binder的Java接口

// ServiceManager.java

public final class ServiceManager {

public static IBinder getService(String name) { //获取服务

}

}

//IServiceManager.java

public interface IServiceManager extends IInterface{ //服务代理对象

}

//java接口的定义

//Java服务的启动过程，ServerThread.java

class ServerThread extends Thread {

}

## binder通信实例

class IInterface : public virtual RefBase {

}

class IFregService : IInterface {

//DECLARE\_META\_INTERFACE(FregService) start

static const ::android::String16 descriptor;

static ::android::sp<IFregService> asInterface(

const ::android::sp<::android::IBinder>& obj);

virtual const ::android::String16& getInterfaceDescriptor() const;

IFregService();

virtual ~IFregService();

//DECLARE\_META\_INTERFACE(FregService) end

virtual getVal() = 0

virtual setVal() = 0

}

class BpInterface : public IFregService, public BpRefBase {

}

class BpFregService : BpInterface<IFregService> {

remote() -> transact()

}

### client

sp<IBinder> binder = defaultServiceManager()->getService()

//sp<IServiceManager> defaultServiceManager()

//sp<IServiceManager> gDefaultServiceManager

gDefaultServiceManager = interface\_cast<IServiceManager>(

ProcessState::self()->getContextObject(NULL))

getStrongProxyForHandle(0)

IBinder\* b

b = new BpBinder(handle)

return result

BpServiceManager::getService()

sp<IFregService> service = IFregService::asInterface(binder)

intr = static\_cast<IFregService\*> obj->queryLocalInterface(IFregService::descriptor).get()

intr = new BpFregService(obj) //if intr == NULL

return intr

### server

main()

FregService::instantiate()

defaultServiceManager()->addService(String16(FREG\_SERVICE),

new FregService())

interface\_cast<IServiceManager>ProcessState::self()->getContext

Object(NULL)->addService()

interface\_cast<IServiceManager>(getStrongProxyForHandle(

0))->addService()

interface\_cast<IServiceManager>((b = new

BpBinder))->addService()

(IServiceManager::asInterface(b = new

BpBinder))->addService()

BpServiceManager::addService()

remote()->transact()

BpBinder::transact()

ProcessState::self()->startThreadPool()

IPCThreadState::self()->joinThreadPool()

defaultServiceManager()

interface\_cast<IServiceManager>(

ProcessState::self()->getContextObject(NULL))

IServiceManager::asInterface(ProcessState ::

getStrongProxyForHandle(0))

ServiceManager::asInterface(new BpBinder)

ProcessState::self()

gProcess = new ProcessState("/dev/binder")

ProcessState::ProcessState(const char \*driver)

: mDriverFD(open\_driver(driver)

mVMStart = mmap(0, BINDER\_VM\_SIZE, PROT\_READ,

MAP\_PRIVATE | MAP\_NORESERVE, mDriverFD, 0)

# HwBinder和HIDL

## hwservicemanager

//service.cpp

main()

configureRpcThreadpool(1, true /\* callerWillJoin \*/)

configureBinderRpcThreadpool(1, true)

ProcessState::self()->setThreadPoolConfiguration(1, true)

ProcessState:: ProcessState()

: mDriverFD(open\_driver())

open("/dev/hwbinder", O\_RDWR | O\_CLOEXEC)

mmap(0, BINDER\_VM\_SIZE, PROT\_READ,

MAP\_PRIVATE | MAP\_NORESERVE, mDriverFD, 0)

ioctl(mDriverFD, BINDER\_SET\_MAX\_THREADS, &)

ServiceManager \*manager = new ServiceManager()

manager->add("default", manager)

TokenManager \*tokenManager = new TokenManager()

manager->add(serviceName, tokenManager)

sp<Looper> looper(Looper::prepare(0 /\* opts \*/))

IPCThreadState::self()->setupPolling(&binder\_fd)

IPCThreadState::self()->flushCommands()

IPCThreadState::talkWithDriver()

ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ,

&bwr)

sp<BinderCallback> cb(new BinderCallback)

looper->addFd(binder\_fd, Looper::POLL\_CALLBACK,

Looper::EVENT\_INPUT, cb, nullptr)

Request request

request.callback = cb

mRequests.add(fd, request)

sp<BnHwServiceManager> service = new

BnHwServiceManager(manager)

IPCThreadState::self()->setTheContextObject(service)

IPCThreadState:: mContextObject = service

ioctl(binder\_fd, BINDER\_SET\_CONTEXT\_MGR, 0)

ioctl(binder\_fd, BINDER\_SET\_INHERIT\_FIFO\_PRIO)

//while (true)

looper->pollAll(-1 /\* timeoutMillis \*/)

ProcessState:: ProcessState()

: mDriverFD(open\_driver())

fd = open("/dev/hwbinder", O\_RDWR | O\_CLOEXEC)

ioctl(fd, BINDER\_SET\_MAX\_THREADS, &maxThreads)

mVMStart = mmap(0, BINDER\_VM\_SIZE, PROT\_READ,

MAP\_PRIVATE | MAP\_NORESERVE, mDriverFD, 0)

ServiceManager::add(name, const sp<IBase>& service)

pid = IPCThreadState::self()->getCallingPid()

BnHwServiceManager : BnHwBase {

onTransact() override //被重新实现

}

BnHwBase : BHwBinder, HidlInstrumentor {

onTransact() override

}

BHwBinder {

transact()

onTransact()

onTransact() //空函数，需要被继承者实现

}

## NFC

### data structure

struct INfc : public ::android::hidl::base::V1\_0::IBase {}

struct BsNfc : INfc, ::android::hardware::details::HidlInstrumentor {}

struct BnHwNfc : public ::android::hidl::base::V1\_0::BnHwBase {}

struct BpHwNfc : public ::android::hardware::BpInterface<INfc>,

public ::android::hardware::details::HidlInstrumentor {}

\_\_attribute\_\_((constructor))static void static\_constructor() {

gBnConstructorMap.set(INfc::descriptor, [](void \*iIntf) ->< IBinder>{

return new BnHwNfc(static\_cast<INfc \*>(iIntf));

});

gBsConstructorMap.set(INfc::descriptor, [](void \*iIntf) ->< IBase>{

return new BsNfc(static\_cast<INfc \*>(iIntf));

});

}

### client

mHal = INfc::getService()

INfc::getService("default",false)

//获取BpHwServiceManager

sp<IServiceManager> sm = defaultServiceManager()

//从vendor/manifest.xml获取transport

transportRet = sm->getTransport()

//获取BpHwNfc

base = sm->get(INfc::descriptor, "default")

castRet = INfc::castFrom(base, true )

return castRet

INfc::castFrom(<IBase>& parent, bool emitError = true)

castInterface<INfc, IBase, BpHwNfc, BpHwBase>(parent, “”, true)

defaultServiceManager()

gDefaultServiceManager = fromBinder<IServiceManager,

BpHwServiceManager,

BnHwServiceManager>(ProcessState::self()->getContextObject(

NULL))

ProcessState::getContextObject(NULL)

getStrongProxyForHandle(0)

b = new BpHwBinder(handle)

return b

fromBinder(sp<IBinder>& binderIface = BpHwBinder, , )

binderIface.get()

binderIface->localBinder() // IBinder::localBinder()

return new BpHwServiceManager (IServiceManager)

base = binderIface.get())->getImpl()

details::canCastInterface(base.get(), IType::descriptor)

StubType\* stub = static\_cast<StubType\*>(binderIface.get());

return stub->getImpl();

getPassthroughServiceManager()

sp<PassthroughServiceManager>manager(new

PassthroughServiceManager()

return manager

struct PassthroughServiceManager : IServiceManager {

Return<sp<IBase>> get(const hidl\_string& fqName, const

hidl\_string& name) override {

std::string sym = "HIDL\_FETCH\_" + ifaceName

}

}

### server

// service.cpp NFC

int main() {

return defaultPassthroughServiceImplementation<INfc>();

return defaultPassthroughServiceImplementation<INfc>("default",

maxThreads = 1);

configureRpcThreadpool(maxThreads = 1, true)

registerPassthroughServiceImplementation<INfc>(name =

"default")

sp<INfc> service = INfc::getService(name = "default", true

/\* getStub \*/)

sp<IServiceManager> sm = defaultServiceManager()

//INfc::descriptor = "android.hardware.nfc@1.0::INfc"

transportRet = sm->getTransport(INfc::descriptor,

"default" )

//PassthroughServiceManager

sp<IServiceManager> pm =

getPassthroughServiceManager()

Return<sp<:IBase>> ret = pm->get(INfc::descriptor,

"default")

PassthroughServiceManager::get(fqName =

android.hardware.nfc@1.0::INfc, name = "default")

//android.hardware.nfc@1.0

const std::string prefix = packageAndVersion + "-

impl";

//INfc

const std::string sym = "HIDL\_FETCH\_" +

ifaceName;

//查找android.hardware.nfc@1.0-impl.so库

std::vector<std::string> libs = search(path, prefix,

".so")

//调用HIDL\_FETCH\_INfc加载hal

\*(void \*\*)(&generator) = dlsym(handle,

sym.c\_str())

iface = new BsNfc(INfc::castFrom(ret)

service->registerAsService(name = "default")

joinRpcThreadpool()

}

INfc::castFrom(<IBase>& parent>, bool emitError = false)

castInterface<INfc, IBase, BpHwNfc, BpHwBase>(parent, emitError)

# Audio系统

## AudioTrack

//AudioTrack.java 音频的采集和输出

public class AudioTrack{

static public int getMinBufferSize(

int sampleRateInHz, //采样率

int channelConfig, //声道数

int audioFormat) //采样精度

native\_get\_min\_buff\_size()

}

static jint android\_media\_AudioTrack\_get\_min\_buff\_size(

JNIEnv \*env, jobject thiz,

jint sampleRateInHertz, jint nbChannels, jint audioFormat){

AudioTrack::getMinFrameCount()

AudioSystem::getOutputSamplingRate() //查询采样率

AudioSystem::getOutputFrameCount() //查询硬件内部缓存大小

AudioSystem::getOutputLatency() //查询硬件的延时时间

}

AudioTrack数据加载模式：

MODE\_STREAM：通过write一次次把把音频数据写到AudioTrac中

MODE\_STATIC：把所有数据一次性写到AudioTrack的内部缓存中

音频流的类型：

STREAM\_ALARM：警告声

STREAM\_MUSIC：音乐声

STREAM\_RING：铃声

STREAM\_SYSTEM：系统声

STREAM\_VOICE\_CALL：通话声

//AudioTrack.h native空间

class AudioTrack{

}

## AudioFlinger

// AudioFlinger.h

class AudioFlinger :

public BinderService<AudioFlinger>,

public BnAudioFlinger{

}

## AudioPolicyService

// AudioPolicyService.h 声音设备的选择、切换和音量控制

class AudioPolicyService: public BnAudioPolicyService, public

AudioPolicyClientInterface, public IBinder::DeathRecipient{

}

# Surface系统

## Activity

//ActivityThread.java

public final class ActivityThread{

private final void handleLaunchActivity(){

performLaunchActivity()

handleResumeActivity()

}

private final Activity performLaunchActivity(){

//调用Activity的onCreate()函数

mInstrumentation.callActivityOnPostCreate()

}

final void handleResumeActivity(){

r.window.getDecorView() //获得View对象

a.getWindowManager()() //获得ViewManager对象

wm.addView(decor, l) //加入带ViewManager中

}

}

// Activity.java

public class Activity extends ContextThemeWrapper

implements LayoutInflater.Factory,

Window.Callback, KeyEvent.Callback,

OnCreateContextMenuListener, ComponentCallbacks{

private Window mWindow

private WindowManager mWindowManager

View mDecor

public void setContentView(){

}

final void attach(){

PolicyManager.makeNewWindow() //创建Window对象

mWindow.setWindowManager() //创建WindowManager对象

}

}

//Window.java

public abstract class Window{

public View findViewById(int id){

return getDecorView().findViewById(id)

}

public final void addView(){

mWindowManager.addView()

}

}

// ViewRoot.java

public final class ViewRoot extends Handler implements ViewParent,

View.AttachInfo.Callbacks {

View mView

private final Surface mSurface

public void setView(){

}

private void draw(){

}

}

## Surface

Bitmap：用于存储像素，也就是画布

Canvas：用于记载画图的动作

Drawing primitive：绘图基元

Paint：描述绘画时使用的颜色，风格等

public class Surface implements Parcelable{

}

//SurfaceFlinger.h

class SurfaceFlinger :

public BinderService<SurfaceFlinger>,

public BnSurfaceComposer,

protected Thread{

}

# Vold和Rild

## Vold

//Volume Daemon，用于管理和控制外部存储设备的后台管理进程

//system/vold/main.cpp

main()

//class NetlinkManager.h

class NetlinkManager{

int start(){

socket()

bind()

NetlinkHandler:: start()

}

}

//NetlinkHandler.h

class NetlinkHandler: public NetlinkListener{

}

//VolumeManager.h

class VolumeManager{

}

// CommandListener.h

class CommandListener : public FrameworkListener {

}

## Rild

//Radio Interface Layer Daemon，用于通信管理和控制的后台进程

//Rild.c

main()

RIL\_startEventLoop() //启动EventLoop，事件处理

pthread\_create(eventLoop) //创建工作线程eventLoop

eventLoop()

ril\_event\_init()

pipe()

ril\_event\_set()

rilEventAddWakeup()

ril\_event\_loop()

funcs = rilInit() //调用RefRil库输出的RIL\_Init()函数，

RIL\_register(funcs) //注册，位于Ril.cpp中

ril\_event\_set(listenCallback)

listenCallback()

# MediaScanner

// MediaScanner.java

public class MediaScanner{

}

// MediaScanner.h

struct MediaScanner{

}

# PackageManagerService

// PackageManagerService.java

class PackageManagerService extends IPackageManager.Stub{

}

# PowerManagerService

// PowerManagerService.java

class PowerManagerService extends IPowerManager.Stub

implements LocalPowerManager, Watchdog.Monitor{

}

## 关机

//frameworks/base/services/core/java/com/android/server/power/ShutdownThread.java

ShutdownThread {

shutdown()

shutdownInner()

shutdownInner()

beginShutdownSequence()

run()

Intent intent = new Intent(Intent.ACTION\_SHUTDOWN)

mContext.sendOrderedBroadcastAsUser()

//Shutting down activity manager

IActivityManager am = IActivityManager.Stub.asInterface()

am.shutdown()

//Shutting down package manager

PackageManagerService pm = getService("package")

pm.shutdown()

// Shutdown radios

shutdownRadios()

//Shutdown StorageManagerService

storageManager.shutdown()

rebootOrShutdown()

shutdownRadios()

nfc.disable(false)

bluetooth.disable()

phone.shutdownMobileRadios()

}

# ActivityManagerService

// ActivityManagerService

public final class ActivityManagerService extends ActivityManagerNative

implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback{

}

# Android驱动

## 硬件抽象层HAL

### 硬件抽象层模块编写规范

硬件抽象层模块文件命名规范：

<MODULE\_ID>.variant.so

MODULE\_ID表示模块ID

variant表示四个系统属性:

ro.hardware, ro.product.board, ro.board.platform, ro.arch

硬件抽象层模块结构体定义规范：

struct hw\_module\_t{ //

uint32\_t tag

uint16\_t version\_major

uint16\_t version\_minor

const char \*id

const char \*name

const char \*author

void\* dso

struct hw\_module\_methods\_t{

int (\*open)(const struct hw\_module\_t\* module, const char\* id,

struct hw\_device\_t\*\* device)

}\* methods //打开硬件抽象层模块中的硬件设备

uint32\_t reserved[32-7]

}

struct hw\_device\_t{ //硬件抽象层中的硬件设备

uint32\_t tag

uint32\_t version

struct hw\_module\_t\* module

uint32\_t reserved[12]

int (\*close)(struct hw\_device\_t\* device)

}

### freg内核程序驱动实现

//Android/kernel/…/freg.c

1. 传统的设备文件系统接口

struct file\_operations freg\_fops{

.owner = THIS\_MODULE,

.open = freg\_open,

.release = freg\_release,

.read = freg\_read,

.write = freg\_write,

}

freg\_open()

container\_of()

freg\_release()

freg\_read()

copy\_to\_user()

freg\_write()

copy\_from\_user()

2.devfs文件系统接口

freg\_val\_show()

freg\_val\_store()

3.proc文件系统接口

freg\_proc\_read()

freg\_proc\_write()

freg\_create\_proc()

create\_proc\_entry()

freg\_remove\_proc()

remove\_proc\_entry()

4.驱动程序模块的加载与卸载函数

freg\_init()

alloc\_chrdev\_region(&dev) //分配主、从设备号

cdev\_add(freg\_fops) //注册字符设备

class\_create() //在/sys/class/下创建设备类目录freg

device\_create() //在/sys/class/freg下创建设备文件freg

device\_create\_file() //在/sys/class/freg下创建属性文件val

freg\_create\_proc() //创建/proc/freg文件

freg\_exit()

remove\_proc\_entry() //删除/proc/freg文件

device\_destroy() //销毁设备

class\_destroy() //销毁设备类别

cdev\_del(dev) //删除字符设备

unregister\_chrdev\_region() //释放设备号资源

module\_init(freg\_init)

module\_exit(freg\_exit)

### 硬件抽象层的模块接口

1. 基本结构

//Android/hardware/…/freg.cpp

freg\_device\_open()

open() //用户空间下open系统调用

freg\_device\_close()

close()

freg\_get\_val()

read()

freg\_set\_val()

write()

struct freg\_module\_t{

struct hw\_module\_t common

}

struct freg\_device\_t{

struct hw\_device\_t common

int fd

int (\*set\_val)()

int (\*get\_val)()

}

static struct hw\_module\_methods\_t freg\_module\_methods{

open: freg\_device\_open

}

struct freg\_module\_t HAL\_MODULE\_INFO\_SYM = {

common: {

id: FREG\_HARDWARE\_MODULE\_ID,

}

}

打包编译后会得到freg.default.so文件

2. 加载过程:

//统一加载函数，通过ID区别不同的模块

int hw\_get\_module(id,hw\_module\_t \*\*module)

load()

dlopen() //将动态链接库形式的HAL层文件加载到内存

dlsym() //获取hw\_module\_t结构体指针

### 硬件访问服务

1. 硬件访问服务接口定义

//frameworks\base\core\java\android\os/IFregService.aidl

interface IFregService{

void set\_val()

int get\_val()

}

2. 硬件访问服务接口实现

//frameworks/base/core/java/android/os/FregService.java

public class FregService extends IFregService.Stub{

private static native init init\_native()

private static native void setVal\_native()

private static native init getVal\_native()

}

native函数的实现

//frameworks/base/services/jni/com\_android\_server\_FregService.cpp

namespace android{

static void freg\_setVal()

static jint freg\_getVal()

static jint freg\_init(){

//加载硬件抽象层freg

hw\_get\_module(FREG\_HARDWARE\_MODULE\_ID,

hw\_module\_t \*module)

module->method->open()

}

3. 启动硬件访问服务

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

//ServerThread类

class ServerThread extends Thread {

public void run() {

try{

……

Slog.i(TAG, "Freg Service")

//创建FregService实例并注册服务

ServiceManager.addService("freg", new FregService())

}

}

### Java上层代码测试

//Freg.java

public class Freg extends Activity implements OnClickListener{

private IFregService fregService = NULL;

fregService = IFregService.Stub.asInterface(

ServiceManager.getService(“freg”))

……

fregService.getVal()

fregService.setVal()

}

### Java本地接口

java本地接口方法表

static const JNINativeMethod method\_table[] = {

{“init\_native”,”()I”,(void\*)freg\_init},

{“setVal\_native”,”(II)V”,(void\*)freg\_setVal}.

{“getVal\_native”,”(I)I”,(void\*)freag\_getVal},

}

//注册java本地接口方法

int register\_android\_server\_FregServerce(){

jniRegisterNtiveMethods(“com/android/server/FregService”,

method\_table)

}

}

//添加到注册函数的注册列表里面去

//frameworks/base/services/jni/onload.cpp

namespace android {

……

int register\_android\_server\_FregServerce(JNIEna \*env);

}

extern "C" jint JNI\_OnLoad(JavaVM\* vm, void\* reserved)

{

……

register\_android\_server\_FregServerce(env)

}

# Ashmem匿名共享内存

## Ashmem驱动

## cutil访问接口

//ashmem-dev.c

//用来访问Ashmem驱动程序的接口

int ashmem\_create\_region(const char \*name, size\_t size)

int ashmem\_pin\_region(int fd, size\_t offset, size\_t len)

int ashmem\_unpin\_region(int fd, size\_t offset, size\_t len)

int ashmem\_set\_prot\_region(int fd, int prot)

int ashmem\_get\_size\_region(int fd)

## C++访问接口

// MemoryHeapBase.h

//描述一个匿名共享内存服务

class MemoryHeapBase : public virtual BnMemoryHeap{

}

MemoryBase.h

class MemoryBase : public BnMemory{

}

## Java访问接口

// MemoryFile.java

public class MemoryFile{

}

# 消息处理

## MessageQueue

// MessageQueue.java

public class MessageQueue{消息队列描述

MessageQueue() {

nativeInit() //创建MessageQueue

}

private native void nativeInit()

android\_os\_MessageQueue\_nativeInit() //native函数

final Message next() //提取消息

final boolean enqueueMessage() //投递消息到消息队列

}

//android\_os\_MessageQueue.cpp

class NativeMessageQueue{

}

## Looper

//Looper.java

public class Looper{

}

## Hand**l**er

Handler.java

public class Handler {

}

# 应用程序框架

## Activity

根Activity组件的启动过程：

//step1 Launcher.java

Launcher:: startActivitySafely(Intent intent, Object tag){

try{

startActivity(intent)

}

}

//step2 Activity.java

Activity:: startActivity(Intent intent){

//step3

startActivityForResult(intent, -1)

Instrumentation.ActivityResult ar =

mInstrumentation.execStartActivity(this,

mMainThread.getApplicationThread(), mToken, this,

intent, requestCode)

}

//step4 Instrumentation.java

Instrumentation:: execStartActivity(Context who, IBinder contextThread,

IBinder token, Activity

target,Intent intent, int requestCode){

try{

int result = ActivityManagerNative.getDefault().startActivity(

whoThread, intent, intent.resolveTypeIfNeeded(

who.getContentResolver()),null,0,token,target != null ?

target.mEmbeddedID : null, requestCode, false, false)

}

}

}

//step5

public abstract class ActivityManagerNative extends Binder implements

IActivityManager{

}

## Service

## Broadcast Receiver

## Content Provider

## 应用程序启动

### 应用程序进程的创建：

//step1 ActivityManagerService.java

ActivityManagerService:: startProcessLocked(

ProcessRecord app,String hostingType, String hostingNameStr){

int pid = Process.start("android.app.ActivityThread",

mSimpleProcessManagement ? app.processName : null, uid,

uid, gids, debugFlags, null)

}

//step2 Process.java

Process:: start(final String processClass, final String niceName, int uid, int

gid, int[] gids, int debugFlags, String[] zygoteArgs){

try {

//step3

return startViaZygote(processClass, niceName, uid, gid, gids,

debugFlags, zygoteArgs)

//step4

pid = zygoteSendArgsAndGetPid(argsForZygote)

openZygoteSocketIfNeeded()

}

}

//step5 ZygoteInit.java

ZygoteInit:: runSelectLoopMode() throws

MethodAndArgsCaller{

done = peers.get(index).runOnce()

}

//step6 ZygoteConnection.java

ZygoteConnection:: runOnce() throws ZygoteInit.MethodAndArgsCaller{

//step7

handleChildProc(parsedArgs, descriptors, newStderr)

RuntimeInit.zygoteInit(parsedArgs.remainingArgs)

}

//step8 RuntimeInit.java

RuntimeInit:: zygoteInit(String[] argv)

throws ZygoteInit.MethodAndArgsCaller{

}

### Binder线程池的启动

//step1 RuntimeInit.java

RuntimeInit:: zygoteInitNative(){

//AndroidRuntime.cpp

com\_android\_internal\_os\_RuntimeInit\_zygoteInit(JNIEnv\* env,

jobject clazz)

gCurRuntime->onZygoteInit()

}

//step2 app\_main.cpp

AppRuntime:: onZygoteInit(){

proc->startThreadPool()

}

//step3 ProcessState.cpp

ProcessState::startThreadPool(){

}

### 消息循环的创建

//step1 RuntimeInit.java

RuntimeInit:: invokeStaticMain(String className, String[] argv)

hrows ZygoteInit.MethodAndArgsCaller{

throw new ZygoteInit.MethodAndArgsCaller(m, argv)

}

//step2 ZygoteInit.java

ZygoteInit:: main(String argv[]){

//step3

catch (MethodAndArgsCaller caller) {

caller.run()

try {

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

}

}

}

//step4 ActivityThread.java

ActivityThread::main{

}

## 应用程序的安装和显示

### 应用程序的安装

### 应用程序的显示

# OTA

## A/B系统

1. A/B系统和传统方式下镜像内容的比较

传统OTA方式下：

1. boot.img内有一个boot ramdisk，用于系统启动时加载system.img；

2. recovery.img内有一个recovery ramdisk，作为recovery系统运行的

ramdisk；

3. system.img只包含android系统的应用程序和库文件；

A/B系统下：

1. system.img除了包含android系统的应用程序和库文件还，另外含

有boot ramdisk，相当于传统OTA下boot.img内的ramdisk存放到

system.img内了；

2. boot.img内包含的是recovery ramdisk，而不是boot ramdisk。Android

系统启动时不再加载boot.img内的ramdisk，而是通过device mapper

机制选择system.img内的ramdisk进行加载；

3. 没有recovery.img文件