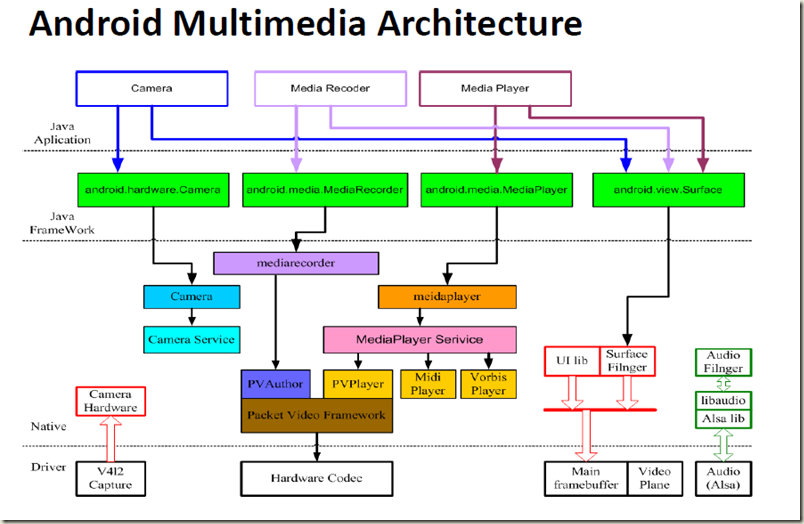
Smooth Streaming

# Introduce



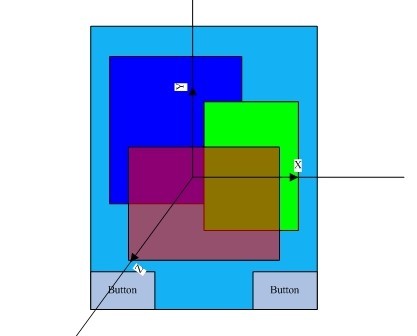
<http://blog.chinaunix.net/uid-20564848-id-96788.html>

## 1.1 Surface Filnger

  SurfaceFlinger服务负责绘制Android应用程序的UI，接着再请求SurfaceFlinger服务将这个已经绘制好了UI的Surface渲染到设备显示屏上去。SurfaceFlinger服务运行在Android系统的System进程中，它负责管理Android系统的帧缓冲区（Frame Buffer）。Android应用程序为了能够将自己的UI绘制在系统的帧缓冲区上，它们就必须要与SurfaceFlinger服务进行通信。

  Android应用程序在请求SurfaceFlinger服务渲染一个Surface之前，首先要将该Surface作为当前活动的绘图上下文，以便可以使用OpengGL库或者其它库的API来在上面绘制UI。

SurfaceFlinger是Android multimedia的一个部分，在Android 的实现中它是一个service，提供[系统](http://blog.chinaunix.net/%2E:;)范围内的surface composer功能，它能够将各种[应用](http://blog.chinaunix.net/%2E:;)程序的2D、3D surface进行组合。



每个应用程序可能对应着一个或者多个图形界面，而每个界面我们就称之为一个surface，或者说是window，在上面的图中我们能看到4个surface，一个是home界面，还有就是红、绿、蓝分别代表的3个surface，而两个button实际是home surface里面的内容。surface实际我们可以把它理解成一个容器，这个容器记录着应用程序界面的控制信息。

SurfaceFlinger 要把各surface组合(compose/merge)成一个main Surface，最后将Main Surface的内容发送给FB/V4l2 Output，这样屏幕上就能看到我们想要的效果。在实际中对这些Surface 进行merge可以采用两种方式，一种就是采用软件的形式来merge，还一种就是采用硬件的方式，软件的方式就是我们的SurfaceFlinger，而硬件的方式就是Overlay。

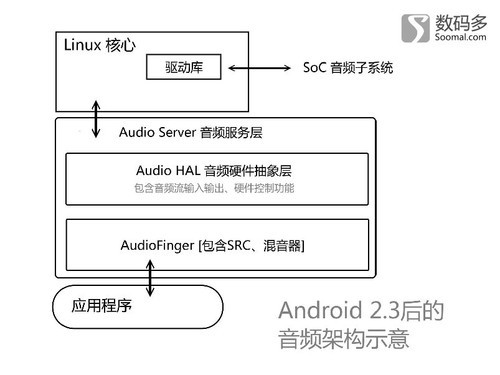
首先要明确的是SurfaceFlinger只是负责merge Surface的控制，比如说计算出两个Surface重叠的区域，至于Surface需要显示的内容，则通过skia，opengl和 pixflinger来计算。 所以我们在介绍SurfaceFlinger 之前先忽略里面存储的内容究竟是什么，先弄清楚它对merge 的一系列控制的过程，然后再结合2D ，3D 引擎来看它的处理过程。

## 1.2 Audio Filnger

AudioFlinger负责管理每个音轨AudioTrack及RecordTrack，主音量控制，每种声音流的属性设置，设备控制，音效控制。AudioFlinger是android中的一个service，在android启动时就已经被加载。

播放线程实际上是MixerThread的一个实例，会把该线程中的各个Track进行混合，必要时还要进行ReSample(重采样)的动作，转换为统一的采样率（44.1K），然后通过音频系统的AudioHardware层输出音频数据。

SRC[Sample Rate Converter，采样频率转换] SRC的作用就是改变信号的采样率，低采样率往高采样率转换时就是一个重采样的过程，重采样对象不再是原始信号，而是这个低采样率的信号，因为采样率不够需要插入更多的采样点以达到需要的采样率和采样大小。



## 1.3 ALSA

ALSA（AdvancedLinux Sound Architecture，高级Linux声音架构）是Linux社区为了取代OSS而提出的一种框架，是一个源代码完全开放的系统(遵循GNU GPL和GNU LGPL)。ALSA在Kernel 2.5版本中被正式引入后，OSS就逐步被排除在内核之外。当然，OSS本身还是在不断维护的，只是不再为Kernel所采用而已。

## 1.4 PVAuthor

提供媒体流记录的功能，完成各种音频(Audio)、视频(Video)流的以及静态图像捕获功能。在媒体流记录的方面，PVAuthor的输入的(Source)是照相机、麦克风等设备，输出(Sink)是各种文件，包含了流的同步、音频视频流的编码(Encode)以及文件的写入等功能。

# 2. Binder机制

<http://blog.csdn.net/luoshengyang/article/details/6618363>

## 2.1什么是Binder？

简单地说，Binder是Android平台上的一种跨进程交互技术。该技术最早并不是由Google公司提出的，它的前身是Be Inc公司开发的OpenBinder，而且在Palm中也有应用。后来OpenBinder的作者Dianne Hackborn加入了Google公司，并负责Android平台的开发工作，所以把这项技术也带进了Android。

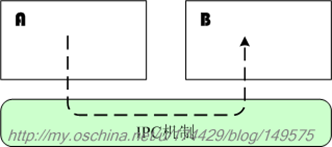
       我们知道，在Android的应用层次上，基本上已经没有过去的进程概念了。然而在实现层次，它毕竟还是要建构在一个个进程之上的。实际上，在Android内部，那些支撑应用的组件往往会身处于不同的进程，那么应用的底层必然会牵涉大量的跨进程通信。为了保证通信的高效性，Android提供了Binder机制。

       Binder机制具有两层含义：   
1） 是一种跨进程通信手段（IPC，Inter-Process Communication）。   
2） 是一种远程过程调用手段（RPC，Remote Procedure Call）。

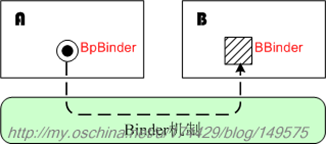
       从实现的角度来说，Binder核心被实现成一个Linux驱动程序，并运行于内核态。这样它才能具有强大的跨进程访问能力。

### 2.1.1 简述Binder的跨进程机制

       为了理解Binder，我们可以先画一张最简单的跨进程通信示意图：

[](http://static.oschina.net/uploads/img/201308/02213509_gW7z.png)

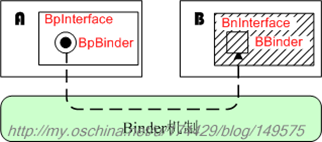
这个很容易理解，不需赘言。到了Android平台上，IPC机制就变成了Binder机制，情况类似，只不过为了便于说明问题，我们需要稍微调整一下示意图：

[](http://static.oschina.net/uploads/img/201308/02213510_BFaY.png)

图中A侧的圆形块[http://static.oschina.net/uploads/space/2013/0731/225838_gb0n_174429.png](http://static.oschina.net/uploads/space/2013/0731/225838_gb0n_174429.png)，表示“Binder代理方”，主要用于向远方发送语义，而B侧的方形块[http://static.oschina.net/uploads/space/2013/0731/230005_alRu_174429.png](http://static.oschina.net/uploads/space/2013/0731/230005_alRu_174429.png)则表示“Binder响应方”，主要用于响应语义。需要说明的是，这种图形表示方法是我自己杜撰的，并没有正规的出处。我个人觉得这种图形非常简便，所以在分析Android架构时，会经常使用这种表示法。

        在后文中，我们可以看到，Binder代理方大概对应于C++层次的BpBinder对象，而Binder响应方则对应于BBinder对象。这两个对象在后文会详细阐述，此处不必太细究。

        然而，上图的Binder代理方主要只负责了“传递信息”的工作，并没有起到“远程过程调用”的作用，如果要支持远程过程调用，我们还必须提供“接口代理方”和“接口实现体”。这样，我们的示意图就需要再调整一下，如下：

[](http://static.oschina.net/uploads/img/201308/02213510_9LvE.png)

        从图中可以看到，A进程并不直接和BpBinder（Binder代理）打交道，而是通过调用BpInterface（接口代理）的成员函数来完成远程调用的。此时，BpBinder已经被聚合进BpInterface了，它在BpInterface内部完成了一切跨进程的机制。另一方面，与BpInterface相对的响应端实体就是BnInterface（接口实现）了。需要注意的是，BnInterface是继承于BBinder的，它并没有采用聚合的方式来包含一个BBinder对象，所以上图中B侧的BnInterface块和BBinder块的背景图案是相同的。

        这样看来，对于远程调用的客户端而言，主要搞的就是两个东西，一个是“Binder代理”，一个是“接口代理”。而服务端主要搞的则是“接口实现体”。因为binder是一种跨进程通信机制，所以还需要一个专门的管理器来为通信两端牵线搭桥，这个管理器就是Service Manager Service。不过目前我们可以先放下Service Manager Service，以后再详细研究。

## 2.2 ServiceBinder相关接口和类

      Android的整个跨进程通信机制都是基于Binder的，这种机制不但会在底层使用，也会在上层使用，所以必须提供Java和C++两个层次的支持。

### 2.2.1 Java层次的binder元素

      Java层次里并没有我们前文图中所表示的BpBinder、BpInterface、BBinder等较低层次的概念，取而代之的是IBinder接口、IInterface等接口。Android要求所有的Binder实体都必须实现IBinder接口，该接口的定义截选如下：

【frameworks/base/core/java/android/os/IBinder.java】

public interface IBinder

{

. . . . . .

public String getInterfaceDescriptor() throws RemoteException;

public boolean pingBinder();

public boolean isBinderAlive();

public IInterface queryLocalInterface(String descriptor);

public void dump(FileDescriptor fd, String[] args) throws RemoteException;

public void dumpAsync(FileDescriptor fd, String[] args) throws RemoteException;

public boolean transact(int code, Parcel data, Parcel reply, int flags)

throws RemoteException;

public interface DeathRecipient

{

public void binderDied();

}

public void linkToDeath(DeathRecipient recipient, int flags)throws RemoteException;

public boolean unlinkToDeath(DeathRecipient recipient, int flags);

}

另外，不管是代理方还是实体方，都必须实现IInterface接口：

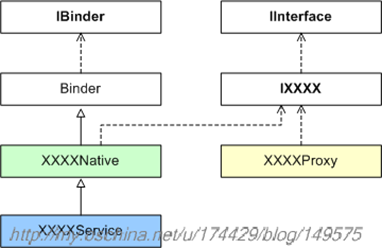
public interface IInterface

{

public IBinder asBinder();

}

   Java层次中，与Binder相关的接口或类的继承关系如下：

[](http://static.oschina.net/uploads/img/201308/02213511_C4Rg.png)

       在实际使用中，我们并不需要编写上图的XXXXNative、XXXXProxy，它们会由ADT根据我们编写的aidl脚本自动生成。用户只需继承XXXXNative编写一个具体的XXXXService即可，这个XXXXService就是远程通信的服务实体类，而XXXXProxy则是其对应的代理类。

        关于Java层次的binder组件，我们就先说这么多，主要是先介绍一个大概。就研究跨进程通信而言，其实质内容基本上都在C++层次，Java层次只是一个壳而已。以后我会写专文来打通Java层次和C++层次，看看它们是如何通过JNI技术关联起来的。现在我们还是把注意力集中在C++层次吧。

### 2.2.2 C++层次的binder元素

        在C++层次，就能看到我们前文所说的BpBinder类和BBinder类了。这两个类都继承于IBinder，IBinder的定义截选如下：

【frameworks/native/include/binder/IBinder.h】

class IBinder : public virtual RefBase

{

public:

. . . . . .

IBinder();

virtual sp<IInterface> queryLocalInterface(const String16& descriptor);

virtual const String16& getInterfaceDescriptor() const = 0;

virtual bool isBinderAlive() const = 0;

virtual status\_t pingBinder() = 0;

virtual status\_t dump(int fd, const Vector<String16>& args) = 0;

virtual status\_t transact(uint32\_t code, const Parcel& data,

Parcel\* reply, uint32\_t flags = 0) = 0;

class DeathRecipient : public virtual RefBase

{

public:

virtual void binderDied(const wp<IBinder>& who) = 0;

};

virtual status\_t linkToDeath(const sp<DeathRecipient>& recipient,

void\* cookie = NULL, uint32\_t flags = 0) = 0;

virtual status\_t unlinkToDeath(const wp<DeathRecipient>& recipient,

void\* cookie = NULL, uint32\_t flags = 0,

wp<DeathRecipient>\* outRecipient = NULL) = 0;

virtual bool checkSubclass(const void\* subclassID) const;

typedef void (\*object\_cleanup\_func)(const void\* id, void\* obj, void\* cleanupCookie);

virtual void attachObject(const void\* objectID, void\* object,

void\* cleanupCookie, object\_cleanup\_func func) = 0;

virtual void\* findObject(const void\* objectID) const = 0;

virtual void detachObject(const void\* objectID) = 0;

virtual BBinder\* localBinder();

virtual BpBinder\* remoteBinder();

protected:

virtual ~IBinder();

private:

};

         C++层次的继承关系图如下：

[](http://static.oschina.net/uploads/img/201308/02213511_MJtb.png)

其中有以下几个很关键的类：

* BpBinder
* BpInterface
* BBinder
* BnInterface

它们扮演着很重要的角色。

#### 2.2.2.1 BpBinder

        BpBinder的定义截选如下：

class BpBinder : public IBinder

{

public:

BpBinder(int32\_t handle);

inline int32\_t handle() const { return mHandle; }

virtual const String16& getInterfaceDescriptor() const;

virtual bool isBinderAlive() const;

virtual status\_t pingBinder();

virtual status\_t dump(int fd, const Vector<String16>& args);

virtual status\_t transact(uint32\_t code, const Parcel& data,

Parcel\* reply, uint32\_t flags = 0);

virtual status\_t linkToDeath(const sp<DeathRecipient>& recipient,

void\* cookie = NULL, uint32\_t flags = 0);

virtual status\_t unlinkToDeath(const wp<DeathRecipient>& recipient,

void\* cookie = NULL, uint32\_t flags = 0,

wp<DeathRecipient>\* outRecipient = NULL);

. . . . . .

. . . . . .

        作为代理端的核心，BpBinder最重要的职责就是实现跨进程传输的传输机制，至于具体传输的是什么语义，它并不关心。我们观察它的transact()函数的参数，可以看到所有的语义都被打包成Parcel了。其他的成员函数，我们先不深究，待我们储备了足够的基础知识后，再回过头研究它们不迟。

#### 2.2.2.2 BpInterface

        另一个重要的类是BpInterface，它的定义如下：

template<typename INTERFACE>

class BpInterface : public INTERFACE, public BpRefBase

{

public:

BpInterface(const sp<IBinder>& remote);

protected:

virtual IBinder\* onAsBinder();

};

其基类BpRefBase的定义如下：

class BpRefBase : public virtual RefBase

{

protected:

BpRefBase(const sp<IBinder>& o);

virtual ~BpRefBase();

virtual void onFirstRef();

virtual void onLastStrongRef(const void\* id);

virtual bool onIncStrongAttempted(uint32\_t flags, const void\* id);

inline IBinder\* remote() { return mRemote; }

inline IBinder\* remote() const { return mRemote; }

private:

BpRefBase(const BpRefBase& o);

BpRefBase& operator=(const BpRefBase& o);

IBinder\* const mRemote;

RefBase::weakref\_type\* mRefs;

volatile int32\_t mState;

};

BpInterface使用了模板技术，而且因为它继承了BpRefBase，所以先天上就聚合了一个mRemote成员，这个成员记录的就是前面所说的BpBinder对象啦。以后，我们还需要继承BpInterface<>实现我们自己的代理类。

       在实际的代码中，我们完全可以创建多个聚合同一BpBinder对象的代理对象，这些代理对象就本质而言，对应着同一个远端binder实体。在Android框架中，常常把指向同一binder实体的多个代理称为token，这样即便这些代理分别处于不同的进程中，它们也具有了某种内在联系。这个知识点需要大家关注。

#### 2.2.2.3 BBinder

       Binder远程通信的目标端实体必须继承于BBinder类，该类和BpBinder相对，主要关心的只是传输方面的东西，不太关心所传输的语义。

class BBinder : public IBinder

{

public:

BBinder();

virtual const String16& getInterfaceDescriptor() const;

virtual bool isBinderAlive() const;

virtual status\_t pingBinder();

virtual status\_t dump(int fd, const Vector<String16>& args);

virtual status\_t transact(uint32\_t code, const Parcel& data,

Parcel\* reply, uint32\_t flags = 0);

virtual status\_t linkToDeath(const sp<DeathRecipient>& recipient,

void\* cookie = NULL, uint32\_t flags = 0);

virtual status\_t unlinkToDeath(const wp<DeathRecipient>& recipient,

void\* cookie = NULL, uint32\_t flags = 0,

wp<DeathRecipient>\* outRecipient = NULL);

virtual void attachObject(const void\* objectID, void\* object,

void\* cleanupCookie, object\_cleanup\_func func);

virtual void\* findObject(const void\* objectID) const;

virtual void detachObject(const void\* objectID);

virtual BBinder\* localBinder();

protected:

virtual ~BBinder();

virtual status\_t onTransact(uint32\_t code, const Parcel& data,

Parcel\* reply, uint32\_t flags = 0);

private:

BBinder(const BBinder& o);

BBinder& operator=(const BBinder& o);

class Extras;

Extras\* mExtras;

void\* mReserved0;

};

       我们目前只需关心上面的transact()成员函数，其他函数留待以后再分析。transact函数的代码如下：

【frameworks/native/libs/binder/Binder.cpp】

status\_t BBinder::transact(uint32\_t code, const Parcel& data,

Parcel\* reply, uint32\_t flags)

{

data.setDataPosition(0);

status\_t err = NO\_ERROR;

switch (code)

{

case PING\_TRANSACTION:

reply->writeInt32(pingBinder());

break;

default:

err = onTransact(code, data, reply, flags);

break;

}

if (reply != NULL)

{

reply->setDataPosition(0);

}

return err;

}

看到了吗，transact()内部会调用onTransact()，从而走到用户所定义的子类的onTransact()里。这个onTransact()的一大作用就是解析经由Binder机制传过来的语义了。

#### 2.2.2.4 BnInterface

       远程通信目标端的另一个重要类是BnInterface<>，它是与BpInterface<>相对应的模板类，比较关心传输的语义。一般情况下，服务端并不直接使用BnInterface<>，而是使用它的某个子类。为此，我们需要编写一个新的BnXXX子类，并重载它的onTransact()成员函数。

         BnInterface<>的定义如下：

template<typename INTERFACE>

class BnInterface : public INTERFACE, public BBinder

{

public:

virtual sp<IInterface> queryLocalInterface(const String16& \_descriptor);

virtual const String16& getInterfaceDescriptor() const;

protected:

virtual IBinder\* onAsBinder();

};

如上所示，BnInterface<>继承于BBinder，但它并没有实现一个默认的onTransact()成员函数，所以在远程通信时，前文所说的BBinder::transact()调用的onTransact()应该就是BnInterface<>的某个子类的onTransact()成员函数。

## 2.3 几个重要的C++宏或模板

       为了便于编写新的接口和类，Android在C++层次提供了几个重要的宏和模板，比如我们在IInterface.h文件中，可以看到DECLARE\_META\_INTERFACE、IMPLEMENT\_META\_INTERFACE的定义。

### 2.3.1 DECLARE\_META\_INTERFACE()

        DECLARE\_META\_INTERFACE()的定义如下：

#define DECLARE\_META\_INTERFACE(INTERFACE) \

static const android::String16 descriptor; \

static android::sp<I##INTERFACE> asInterface( \

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

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

I##INTERFACE(); \

virtual ~I##INTERFACE(); \

我们举个实际的例子，来说明如何使用这个宏：

[](http://static.oschina.net/uploads/img/201308/02212349_yUPC.png)

上例中ICamera内部使用了DECLARE\_META\_INTERFACE(Camera)，我们把宏展开后，可以看到ICamera类的定义相当于：

class ICamera: public IInterface

{

public:

static const android::String16 descriptor;

static android::sp<ICamera> asInterface( const android::sp<android::IBinder>& obj);

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

ICamera();

virtual ~ICamera();

virtual void disconnect() = 0;

. . . . . .

宏展开的部分就是中间那5行代码，其中最关键的就是asInterface()函数了，这个函数将承担把BpBinder打包成BpInterface的职责。

### 2.3.2 IMPLEMENT\_META\_INTERFACE()

       与DECLARE\_META\_INTERFACE相对的就是IMPLEMENT\_META\_INTERFACE宏。它的定义如下：

#define IMPLEMENT\_META\_INTERFACE(INTERFACE, NAME) \

const android::String16 I##INTERFACE::descriptor(NAME); \

const android::String16& \

I##INTERFACE::getInterfaceDescriptor() const { \

return I##INTERFACE::descriptor; \

} \

android::sp<I##INTERFACE> I##INTERFACE::asInterface( \

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

{ \

android::sp<I##INTERFACE> intr; \

if (obj != NULL) { \

intr = static\_cast<I##INTERFACE\*>( \

obj->queryLocalInterface( \

I##INTERFACE::descriptor).get()); \

if (intr == NULL) { \

intr = new Bp##INTERFACE(obj); \

} \

} \

return intr; \

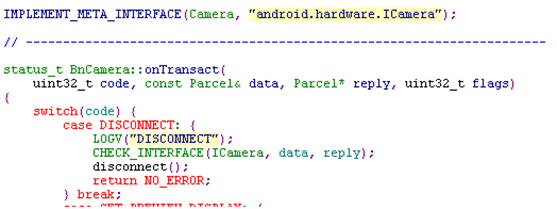
} \

I##INTERFACE::I##INTERFACE() { } \

I##INTERFACE::~I##INTERFACE() { } \

其中，实现了关键的asInterface()函数。

       实际使用IMPLEMENT\_META\_INTERFACE时，我们只需把它简单地写在binder实体所处的cpp文件中即可，举例如下：

[](http://static.oschina.net/uploads/img/201308/02212349_xaHC.png)

其中的IMPLEMENT\_META\_INTERFACE(Camera, “android.hardware.ICamera”);一句相当于以下这段代码：

const android::String16 ICamera::descriptor(“android.hardware.ICamera”);

const android::String16& ICamera::getInterfaceDescriptor() const

{

return ICamera::descriptor;

}

android::sp<ICamera> ICamera::asInterface(const android::sp<android::IBinder>& obj)

{

android::sp<ICamera > intr;

if (obj != NULL)

{

intr = static\_cast<ICamera\*>(obj->queryLocalInterface(

ICamera::descriptor).get());

if (intr == NULL)

{

intr = new BpCamera(obj);

}

}

return intr;

}

ICamera::ICamera() { }

ICamera::~ICamera () { }

看来，其中重点实现了asInterface()成员函数。请注意，asInterface()函数中会先尝试调用queryLocalInterface()来获取intr。此时，如果asInterface()的obj参数是个代理对象（BpBinder），那么intr = static\_cast<ICamera\*>(obj->queryLocalInterface(...)一句得到的intr基本上就是NULL啦。这是因为除非用户编写的代理类重载queryLocalInterface()函数，否则只会以默认函数为准。而IBinder类中的默认queryLocalInterface()函数如下：

【frameworks/native/libs/binder/Binder.cpp】

sp<IInterface> IBinder::queryLocalInterface(const String16& descriptor)

{

return NULL;

}

另一方面，如果obj参数是个实现体对象（BnInterface对象）的话，那么queryLocalInterface()函数的默认返回值就是实体对象的this指针了，代码如下：

【frameworks/native/include/binder/IInterface.h】

template<typename INTERFACE>

inline sp<IInterface> BnInterface<INTERFACE>::queryLocalInterface(const String16& \_descriptor)

{

if (\_descriptor == INTERFACE::descriptor)

return this;

return NULL;

}

 在我们所举的Camera例子中，我们要研究的是如何将BpBinder转成BpInterface，所以现在我们只阐述obj参数为BpBinder的情况。此时asInterface()函数中obj->queryLocalInterface()的返回值为NULL，于是asInterface()会走到new BpCamera(obj)一句，这一句是最关键的一句。我们知道，BpCamera继承于BpInterface<ICamera>，所以此时所创建的BpCamera对象正是可被App使用的BpInterface代理对象。

 BpCamera的定义如下：

class BpCamera: public BpInterface<ICamera>

{

public:

BpCamera(const sp<IBinder>& impl)

: BpInterface<ICamera>(impl)

{

}

// disconnect from camera service

void disconnect()

{

LOGV("disconnect");

Parcel data, reply;

data.writeInterfaceToken(ICamera::getInterfaceDescriptor());

remote()->transact(DISCONNECT, data, &reply);

}

. . . . . .

至此，IMPLEMENT\_META\_INTERFACE宏和asInterface()函数的关系就分析完毕了。

### 2.3.3 interface\_cast

       不过，我们经常使用的其实并不是asInterface()函数，而是interface\_cast()，它简单包装了asInterface()：

template<typename INTERFACE>

inline sp<INTERFACE> interface\_cast(const sp<IBinder>& obj)

{

return INTERFACE::asInterface(obj);

}

        以上就是关于C++层次中一些binder元素的介绍，下面我们再进一步分析其他细节。

## 2.4 ProcessState

       前文我们已经提到过，在Android的上层架构中，已经大幅度地弱化了进程的概念。应用程序员能看到的主要是activity、service、content provider等概念，再也找不到以前熟悉的main()函数了。然而，底层程序（C++层次）毕竟还是得跑在一个个进程之上，现在我们就来看底层进程是如何运用Binder机制来完成跨进程通信的。

       在每个进程中，会有一个全局的ProcessState对象。这个很容易理解，ProcessState的字面意思不就是“进程状态”吗，当然应该是每个进程一个ProcessState。ProcessState的定义位于frameworks/native/include/binder/ProcessState.h中，我们只截选其中的一部分：

class ProcessState : public virtual RefBase

{

public:

static sp<ProcessState> self();

. . . . . .

void startThreadPool();

. . . . . .

void spawnPooledThread(bool isMain);

status\_t setThreadPoolMaxThreadCount(size\_t maxThreads);

private:

friend class IPCThreadState;

. . . . . .

struct handle\_entry

{

IBinder\* binder;

RefBase::weakref\_type\* refs;

};

handle\_entry\* lookupHandleLocked(int32\_t handle);

int mDriverFD;

void\* mVMStart;

mutable Mutex mLock; // protects everything below.

Vector<handle\_entry> mHandleToObject;

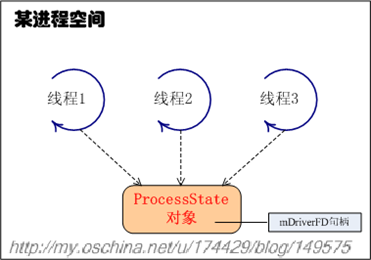
. . . . . .

KeyedVector<String16, sp<IBinder> > mContexts;

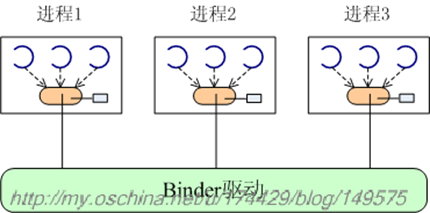
. . . . . .

};

我们知道，Binder内核被设计成一个驱动程序，所以ProcessState里专门搞了个mDriverFD域，来记录binder驱动对应的句柄值，以便随时和binder驱动通信。ProcessState对象采用了典型的单例模式，在一个应用进程中，只会有唯一的一个ProcessState对象，它将被进程中的多个线程共用，因此每个进程里的线程其实是共用所打开的那个驱动句柄（mDriverFD）的，示意图如下：

[](http://static.oschina.net/uploads/img/201308/02213512_87cy.png)

每个进程基本上都是这样的结构，组合起来的示意图就是：

[](http://static.oschina.net/uploads/img/201308/02213512_gaQv.png)

我们常见的使用ProcessState的代码如下：

int main(int argc, char\*\* argv)

{

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

. . . . . .

. . . . . .

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

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

}

因为ProcessState采用的是单例模式，所以它的构造函数是private的，我们只能通过调用ProcessState::self()来获取进程中唯一的一个ProcessState对象。self()函数的代码如下：

sp<ProcessState> ProcessState::self()

{

Mutex::Autolock \_l(gProcessMutex);

if (gProcess != NULL) {

return gProcess;

}

gProcess = new ProcessState;

return gProcess;

}

  ProcessState对象构造之时，就会打开binder驱动：

ProcessState::ProcessState()

: mDriverFD(open\_driver()) // 打开binder驱动。

, mVMStart(MAP\_FAILED)

, mManagesContexts(false)

, mBinderContextCheckFunc(NULL)

, mBinderContextUserData(NULL)

, mThreadPoolStarted(false)

, mThreadPoolSeq(1)

{

. . . . . .

mVMStart = mmap(0, BINDER\_VM\_SIZE, PROT\_READ, MAP\_PRIVATE | MAP\_NORESERVE, mDriverFD, 0);

. . . . . .

}

注意上面那句mDriverFD(open\_driver())，其中的open\_driver()就负责打开“/dev/binder”驱动：

static int open\_driver()

{

int fd = open("/dev/binder", O\_RDWR);

. . . . . .

status\_t result = ioctl(fd, BINDER\_VERSION, &vers);

. . . . . .

size\_t maxThreads = 15;

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

. . . . . .

return fd;

}

ProcessState中另一个比较有意思的域是mHandleToObject：

Vector<handle\_entry> mHandleToObject;

它是本进程中记录所有BpBinder的向量表噢，非常重要。我们前文已经说过，BpBinder是代理端的核心，现在终于看到它的藏身之处了。在Binder架构中，应用进程是通过“binder句柄”来找到对应的BpBinder的。从这张向量表中我们可以看到，那个句柄值其实对应着这个向量表的下标。这张表的子项类型为handle\_entry，定义如下：

struct handle\_entry

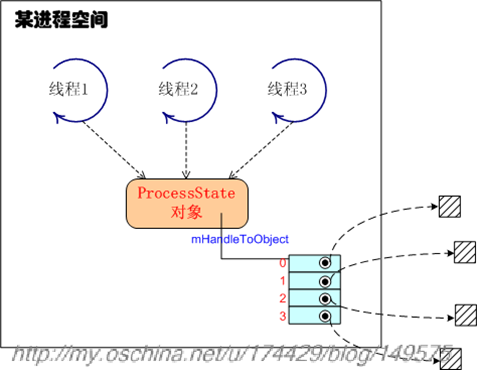
{

IBinder\* binder;

RefBase::weakref\_type\* refs;

};

其中的binder域，记录的就是BpBinder对象。

[](http://static.oschina.net/uploads/img/201308/02213513_8Bqe.png)

         以上是对Binder的初步知识。

## 2.5 Binder（ServiceManager篇）

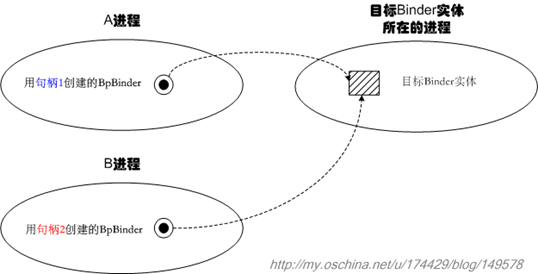
  Android平台的一个基本设计理念是构造一个相对平坦的功能集合，这些功能可能会身处于不同的进程中，然而却可以高效地整合到一起，实现不同的用户需求。这就必须打破过去各个孤立App所形成的天然藩篱。为此，Android提供了Binder机制。

       在Android中，系统提供的服务被包装成一个个系统级service，这些service往往会在设备启动之时添加进Android系统。在上一篇文档中，我们已经了解了BpBinder和BBinder的概念，而service实体的底层说到底就是一个BBinder实体。

  我们知道，如果某个程序希望享受系统提供的服务，它就必须调用系统提供的外部接口，向系统发出相应的请求。因此，Android中的程序必须先拿到和某个系统service对应的代理接口，然后才能通过这个接口，享受系统提供的服务。说白了就是我们得先拿到一个和目标service对应的合法BpBinder。

       然而，该怎么获取和系统service对应的代理接口呢？Android是这样设计的：先启动一个特殊的系统服务，叫作Service Manager Service（简称SMS），它的基本任务就是管理其他系统服务。其他系统服务在系统启动之时，就会向SMS注册自己，于是SMS先记录下与那个service对应的名字和句柄值。有了句柄值就可以用来创建合法的BpBinder了。只不过在实际的代码中，SMS并没有用句柄值创建出BpBinder，这个其实没什么，反正指代目标service实体的目的已经达到了。后续当某程序需要享受某系统服务时，它必须先以“特定手法”获取SMS代理接口，并经由这个接口查询出目标service对应的合法Binder句柄，然后再创建出合法的BpBinder对象。

在此，我们有必要交代一下“Binder句柄”的作用。句柄说穿了是个简单的整数值，用来告诉Binder驱动我们想找的目标Binder实体是哪个。但是请注意，句柄只对发起端进程和Binder驱动有意义，A进程的句柄直接拿到B进程，是没什么意义的。也就是说，不同进程中指代相同Binder实体的句柄值可能是不同的。示意图如下：

[](http://static.oschina.net/uploads/img/201308/02220830_donF.png)

       SMS记录了所有系统service所对应的Binder句柄，它的核心功能就是维护好这些句柄值。后续，当用户进程需要获取某个系统service的代理时，SMS就会在内部按service名查找到合适的句柄值，并“逻辑上”传递给用户进程，于是用户进程会得到一个新的合法句柄值，这个新句柄值可能在数值上和SMS所记录的句柄值不同，然而，它们指代的却是同一个Service实体。句柄的合法性是由Binder驱动保证的，这一点我们不必担心。

        前文我们提到要以“特定手法”获取SMS代理接口，这是什么意思呢？在IServiceManager.cpp文件中，我们可以看到一个defaultServiceManager()函数，代码如下：

【frameworks/native/libs/binder/IServiceManager.cpp】

sp<IServiceManager> defaultServiceManager()

{

if (gDefaultServiceManager != NULL)

return gDefaultServiceManager;

{

AutoMutex \_l(gDefaultServiceManagerLock);

if (gDefaultServiceManager == NULL)

{

gDefaultServiceManager = interface\_cast<IServiceManager>(

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

}

}

return gDefaultServiceManager;

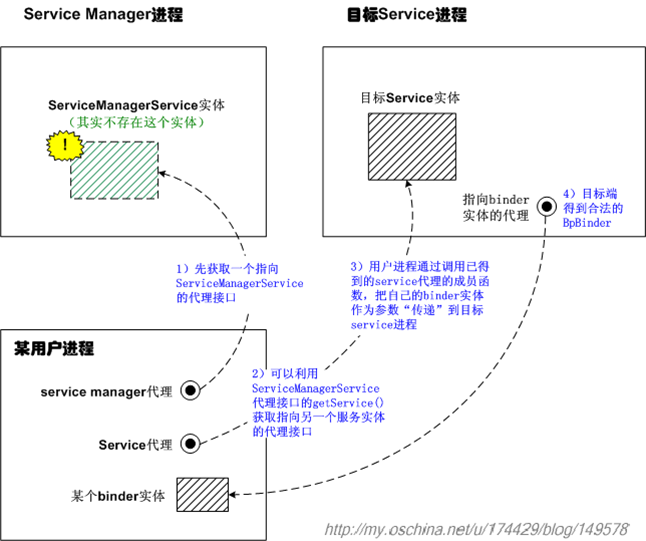
}

这个函数里调用interface\_cast的地方是用一句getContextObject(NULL)来获取BpBinder对象的。我们先不深入讲解这个函数，只需要知道这一句里的getContextObject(NULL)实际上相当于new BpBinder(0)就可以了。噢，看来要得到BpBinder对象并不复杂嘛，直接new就好了。然而，我之所以使用“特定手法”一词，是因为这种直接new BpBinder(xxx)的做法，只能用于获取SMS的代理接口。大家可不要想当然地随便用这种方法去创建其他服务的代理接口噢。

        在Android里，对于Service Manager Service这个特殊的服务而言，其对应的代理端的句柄值已经预先定死为0了，所以我们直接new BpBinder(0)拿到的就是个合法的BpBinder，其对端为“Service Manager Service实体”（至少目前可以先这么理解）。那么对于其他“服务实体”对应的代理，句柄值又是多少呢？使用方又该如何得到这个句柄值呢？我们总不能随便蒙一个句柄值吧。正如我们前文所述，要得到某个服务对应的BpBinder，主要得借助Service Manager Service系统服务，查询出一个合法的Binder句柄，并进而创建出合法的BpBinder。

        这里有必要澄清一下，利用SMS获取合法BpBinder的方法，并不是Android中得到BpBinder的唯一方法。另一种方法是，“起始端”经由一个已有的合法BpBinder，将某个binder实体或代理对象作为跨进程调用的参数，“传递”给“目标端”，这样目标端也可以拿到一个合法的BpBinder。

        我们把以上介绍的知识绘制成示意图，如下：

[](http://static.oschina.net/uploads/img/201308/02220831_LSOf.png)

请顺着图中标出的1）、2）、3）、4）序号，读一下图中的说明。

        在跨进程通信方面，所谓的“传递”一般指的都是逻辑上的传递，所以应该打上引号。事实上，binder实体对象是不可能完全打包并传递到另一个进程的，而且也没有必要这么做。目前我们只需理解，binder架构会保证“传递”动作的目标端可以拿到一个和binder实体对象对应的代理对象即可。详细情况，要到分析binder驱动的部分再阐述。

        既然SMS承担着让客户端获取合法BpBinder的责任，那么它的重要性就不言而喻了。现在我们就来详细看看具体如何使用它。

### 2.5.1具体使用ServiceManager

#### 2.5.1.1 必须先得到IServiceManager代理接口

要获取某系统service的代理接口，必须先得到IServiceManager代理接口。还记得前文C++代码中获取IServiceManager代理接口的句子吗？

gDefaultServiceManager = interface\_cast<IServiceManager>(

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

我们在前一篇文档中已经介绍过interface\_cast了，现在再贴一下这个函数的代码：

template<typename INTERFACE>

inline sp<INTERFACE> interface\_cast(const sp<IBinder>& obj)

{

return INTERFACE::asInterface(obj);

}

也就是说，其实调用的是IServiceManager::asInterface(obj)，而这个obj参数就是new BpBinder(0)得到的对象。当然，这些都是C++层次的概念，Java层次把这些概念都包装起来了。

      在Java层次，是这样获取IServiceManager接口的：

【frameworks/base/core/java/android/os/ServiceManager.java】

private static IServiceManager getIServiceManager()

{

if (sServiceManager != null) {

return sServiceManager;

}

// Find the service manager

sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject());

return sServiceManager;

}

噢，又出现了一个asInterface，看来Java层次和C++层的代码在本质上是一致的。

        ServiceManagerNative的asInterface()代码如下：

static public IServiceManager asInterface(IBinder obj)

{

if (obj == null)

{

return null;

}

IServiceManager in = (IServiceManager)obj.queryLocalInterface(descriptor);

if (in != null)

{

return in;

}

return new ServiceManagerProxy(obj);

}

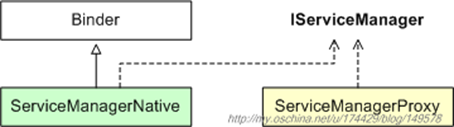
目前我们只需了解，用户进程在调用到getIServiceManager()时，最终会走到return new ServiceManagerProxy(obj)即可。

     又出现了两个名字：ServiceManagerProxy和ServiceManagerNative。简单地说：

1） ServiceManagerProxy就是IServiceManager代理接口；

2） ServiceManagerNative显得很鸡肋；

它们的继承关系图如下：

[](http://static.oschina.net/uploads/img/201308/02221437_nv01.png)

下面我们分别来说明。

##### 2.5.1.1.1 ServiceManagerProxy就是IServiceManager代理接口

用户要访问Service Manager Service服务，必须先拿到IServiceManager代理接口，而ServiceManagerProxy就是代理接口的实现。这个从前文代码中的new ServiceManagerProxy(obj)一句就可以看出来了。ServiceManagerProxy的构造函数内部会把obj参数记录到mRemote域中：

public ServiceManagerProxy(IBinder remote)

{

mRemote = remote;

}

mRemote的定义是：

private IBinder mRemote;

其实说白了，mRemote的核心包装的就是句柄为0的BpBinder对象，这个应该很容易理解。

      日后，当我们通过IServiceManager代理接口访问SMS时，其实调用的就是ServiceManagerProxy的成员函数。比如getService()、checkService()等等。

##### 2.5.1.1.2 ServiceManagerNative显得很鸡肋

另一方面，ServiceManagerNative就显得很鸡肋了。

        ServiceManagerNative是个抽象类：

public abstract class ServiceManagerNative extends Binder implements IServiceManager

它继承了Binder，实现了IServiceManager，然而却是个虚有其表的class。它唯一有用的大概就是前文列出的那个静态成员函数asInterface()了，而其他成员函数（像onTransact()）就基本上没什么用。

       如果我们花点儿时间在工程里搜索一下ServiceManagerNative，会发现根本找不到它的子类。一个没有子类的抽象类不就是虚有其表吗。到头来我们发现，关于ServiceManagerNative的用法只有一种，就是：

ServiceManagerNative.asInterface(BinderInternal.getContextObject());

用一下它的asInterface()静态函数而已。

      为什么会这样呢？我想这可能是某种历史的遗迹吧。同理，我们看它的onTransact()函数，也会发现里面调用的类似addService()那样的函数，也都是找不到对应的实现体的。当然，因为ServiceManagerNative本身是个抽象类，所以即便它没有实现IServiceManager的addService()等成员函数，也是可以编译通过的。

       这里透出一个信息，既然Java层的ServiceManagerNative没什么大用处，是不是表示C++层也缺少对应的SMS服务实体呢？在后文我们可以看到，的确是这样的，Service Manager Service在C++层被实现成一个独立的进程，而不是常见的Binder实体。

#### 2.5.1.2通过addService()来注册系统服务

我们还是回过头接着说对于IServiceManager接口的使用吧。最重要的当然是注册系统服务。比如在System Server进程中，是这样注册PowerManagerService系统服务的：

public void run()

{

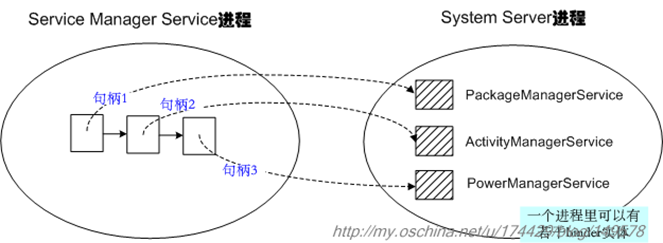
. . . . . .

power = new PowerManagerService();

ServiceManager.addService(Context.POWER\_SERVICE, power);

. . . . . .

addService()的第一个参数就是所注册service的名字，比如上面的POWER\_SERVICE对应的字符串就是"power"。第二个参数传入的是service Binder实体。Service实体在Service Manager Service一侧会被记录成相应的句柄值，如图：

[](http://static.oschina.net/uploads/img/201308/02223321_kczy.png)

        有关addService()内部机理，我们会在后文讲述，这里先不细说。

#### 2.5.1.3通过getService()来获取某系统服务的代理接口

除了注册系统服务，Service Manager Service的另一个主要工作就是让用户进程可以获取系统service的代理接口，所以其getService()函数就非常重要了。

       其实，ServiceManagerProxy中的getService()等成员函数，仅仅是把语义整理进parcel，并通过mRemote将parcel传递到目标端而已。所以我们只看看getService()就行了，其他的函数都大同小异。

public IBinder getService(String name) throws RemoteException

{

Parcel data = Parcel.obtain();

Parcel reply = Parcel.obtain();

data.writeInterfaceToken(IServiceManager.descriptor);

data.writeString(name);

mRemote.transact(GET\_SERVICE\_TRANSACTION, data, reply, 0);

IBinder binder = reply.readStrongBinder();

reply.recycle();

data.recycle();

return binder;

}

传递的语义就是GET\_SERVICE\_TRANSACTION，非常简单。mRemote从本质上看就是句柄为0的BpBinder，所以binder驱动很清楚这些语义将去向何方。

       关于Service Manager Service的使用，我们就先说这么多。下面我们要开始探索SMS内部的运作机制了。

### 2.5.2 Service Manager Service的运作机制

#### 2.5.2.1 Service Manager Service服务的启动

既然前文说ServiceManagerNative虚有其表，而且没有子类，那么Service Manager Service服务的真正实现代码位于何处呢？答案就在init.rc脚本里。关于init.rc的详细情况，可参考其他阐述Android启动流程的文档，此处不再赘述。

        init.rc脚本中，在描述zygote service之前就已经写明service manager service的信息了：

service servicemanager /system/bin/servicemanager

user system

critical

onrestart restart zygote

onrestart restart media

可以看到，servicemanager是一种native service。这种native service都是需要用C/C++编写的。Service Manager Service对应的实现代码位于frameworks/base/cmds/servicemanager/Service\_manager.c文件中。这个文件中有每个C程序员都熟悉的main()函数，其编译出的可执行程序就是/system/bin/servicemanager。

        另外，还有一个干扰我们视线的cpp文件，名为IServiceManager.cpp，位于frameworks/base/libs/binder/目录中，这个文件里的BnServiceManager应该和前文的ServiceManagerNative类似，它的onTransact()也不起什么作用。

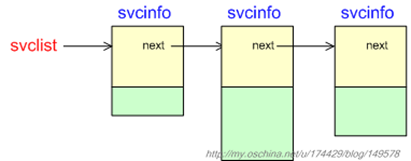
#### 2.5.2.2 Service Manager Service是如何管理service句柄的？

在C语言层次，简单地说并不存在一个单独的ServiceManager结构。整个service管理机制都被放在一个独立的进程里了，该进程对应的实现文件就是Service\_manager.c。

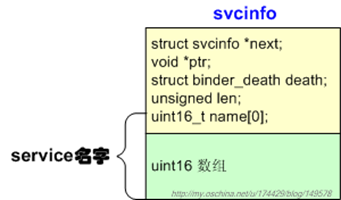
        进程里有一个全局性的svclist变量：

struct svcinfo \*svclist = 0;

它记录着所有添加进系统的“service代理”信息，这些信息被组织成一条单向链表，我们不妨称这条链表为“服务向量表”。示意图如下：

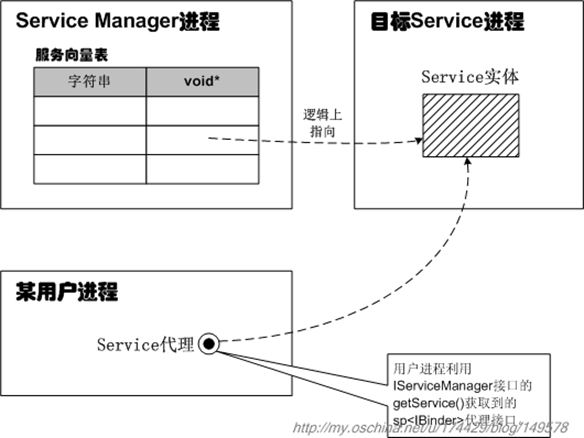
[](http://static.oschina.net/uploads/img/201308/02223321_cnDq.png)

链表节点类型为svcinfo。

[](http://static.oschina.net/uploads/img/201308/02223322_6ftO.png)

因为svcinfo里要记录下service的名字字符串，所以它需要的buffer长度是(len + 1) \* sizeof(uint16\_t)，记得要留一个’\0’的结束位置。另外，svcinfo的ptr域，实际上记录的就是系统service对应的binder句柄值。

        日后，当应用调用getService()获取系统服务的代理接口时，SMS就会搜索这张“服务向量表”，查找是否有节点能和用户传来的服务名匹配，如果能查到，就返回对应的sp<IBinder>，这个接口在远端对应的实体就是“目标Service实体”。如此一来，系统中就会出现如下关系：

[](http://static.oschina.net/uploads/img/201308/02223322_DFKs.png)

#### 2.5.2.3 Service Manager Service的主程序（C++层）

要更加深入地了解Service Manager进程的运作，我们必须研究其主程序。参考代码是frameworks\base\cmds\servicemanager\Service\_manager.c。

        Service\_manager.c中的main()函数如下：

int main(int argc, char \*\*argv)

{

struct binder\_state \*bs;

void \*svcmgr = BINDER\_SERVICE\_MANAGER;

bs = binder\_open(128\*1024);

if (binder\_become\_context\_manager(bs))

{

ALOGE("cannot become context manager (%s)\n", strerror(errno));

return -1;

}

svcmgr\_handle = svcmgr;

binder\_loop(bs, svcmgr\_handler);

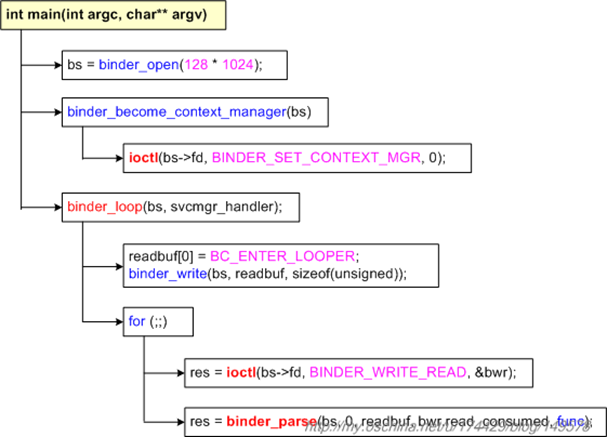
return 0;

}

       main()函数一开始就打开了binder驱动，然后调用binder\_become\_context\_manager()让自己成为整个系统中唯一的上下文管理器，其实也就是service管理器啦。接着main()函数调用binder\_loop()进入无限循环，不断监听并解析binder驱动发来的命令。

        binder\_loop()中解析驱动命令的函数是binder\_parse()，其最后一个参数func来自于binder\_loop()的最后一个参数——svcmgr\_handler函数指针。这个svcmgr\_handler()应该算是Service Manager Service的核心回调函数了。

        为了方便查看，我把main()函数以及其间接调用的ioctl()语句绘制成如下的调用关系图：

[](http://static.oschina.net/uploads/img/201308/02223322_FGFJ.png)

下面我们逐个分析其中调用的函数。

##### 2.5.2.3.1 binder\_open()

Service Manager Service必须先调用binder\_open()来打开binder驱动，驱动文件为“/dev/binder”。binder\_open()的代码截选如下：

struct binder\_state \* binder\_open(unsigned mapsize)

{

struct binder\_state \*bs;

bs = malloc(sizeof(\*bs));

. . . . . .

bs->fd = open("/dev/binder", O\_RDWR);

. . . . . .

bs->mapsize = mapsize;

bs->mapped = mmap(NULL, mapsize, PROT\_READ, MAP\_PRIVATE, bs->fd, 0);

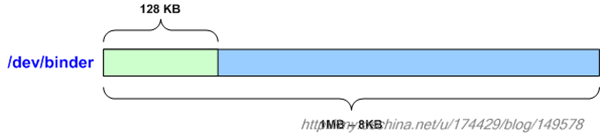
. . . . . .

return bs;

. . . . . .

}

       binder\_open()的参数mapsize表示它希望把binder驱动文件的多少字节映射到本地空间。可以看到，Service Manager Service和普通进程所映射的binder大小并不相同。它把binder驱动文件的128K字节映射到内存空间，而普通进程则会映射binder文件里的BINDER\_VM\_SIZE（即1M减去8K）字节。

[](http://static.oschina.net/uploads/img/201308/03001252_5zk3.png)

具体的映射动作由mmap()一句完成，该函数将binder驱动文件的一部分映射到进程空间。mmap()的函数原型如下：

void\* mmap ( void \* addr , size\_t len , int prot , int flags , int fd , off\_t offset );

该函数会把“参数fd所指代的文件”中的一部分映射到进程空间去。这部分文件内容以offset为起始位置，以len为字节长度。其中，参数offset表明从文件起始处开始算起的偏移量。参数prot表明对这段映射空间的访问权限，可以是PROT\_READ（可读）、PROT\_WRITE （可写）、PROT\_EXEC （可执行）、PROT\_NONE（不可访问）。参数addr用于指出文件应被映射到进程空间的起始地址，一般指定为空指针，此时会由内核来决定起始地址。

        binder\_open()的返回值类型为binder\_state\*，里面记录着刚刚打开的binder驱动文件句柄以及mmap()映射到的最终目标地址。

struct binder\_state

{

int fd;

void \*mapped;

unsigned mapsize;

};

以后，SMS会不断读取这段映射空间，并做出相应的动作。

##### 2.5.2.3.2 binder\_become\_context\_manager()

我们前面已经说过，binder\_become\_context\_manager()的作用是让当前进程成为整个系统中唯一的上下文管理器，即service管理器。其代码非常简单：

int binder\_become\_context\_manager(struct binder\_state \*bs)

{

return ioctl(bs->fd, BINDER\_SET\_CONTEXT\_MGR, 0);

}

仅仅是把BINDER\_SET\_CONTEXT\_MGR发送到binder驱动而已。驱动中与ioctl()对应的binder\_ioctl()是这样处理的：

static long binder\_ioctl(struct file \*filp, unsigned int cmd, unsigned long arg)

{

int ret;

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

struct binder\_thread \*thread;

unsigned int size = \_IOC\_SIZE(cmd);

void \_\_user \*ubuf = (void \_\_user \*)arg;

. . . . . .

. . . . . .

case BINDER\_SET\_CONTEXT\_MGR:

. . . . . .

. . . . . .

binder\_context\_mgr\_uid = current->cred->euid;

binder\_context\_mgr\_node = binder\_new\_node(proc, NULL, NULL);

if (binder\_context\_mgr\_node == NULL)

{

ret = -ENOMEM;

goto err;

}

binder\_context\_mgr\_node->local\_weak\_refs++;

binder\_context\_mgr\_node->local\_strong\_refs++;

binder\_context\_mgr\_node->has\_strong\_ref = 1;

binder\_context\_mgr\_node->has\_weak\_ref = 1;

break;

. . . . . .

. . . . . .

}

       代码的意思很明确，要为整个系统的上下文管理器专门生成一个binder\_node节点，并记入静态变量binder\_context\_mgr\_node。

       我们在这里多说两句，一般情况下，应用层的每个binder实体都会在binder驱动层对应一个binder\_node节点，然而binder\_context\_mgr\_node比较特殊，它没有对应的应用层binder实体。在整个系统里，它是如此特殊，以至于系统规定，任何应用都必须使用句柄0来跨进程地访问它。现在大家可以回想一下前文在获取SMS接口时说到的那句new BpBinder(0)，是不是能加深一点儿理解。

##### 2.5.2.3.3 binder\_loop()

我们再回到SMS的main()函数。

        接下来的binder\_loop()会先向binder驱动发出了BC\_ENTER\_LOOPER命令，接着进入一个for循环不断调用ioctl()读取发来的数据，接着解析这些数据。参考代码在：

【frameworks/base/cmds/servicemanager/Binder.c】（注意！这个Binder.c文件不是binder驱动层那个Binder.c文件噢。）

void binder\_loop(struct binder\_state \*bs, binder\_handler func)

{

int res;

struct binder\_write\_read bwr;

unsigned readbuf[32];

bwr.write\_size = 0;

bwr.write\_consumed = 0;

bwr.write\_buffer = 0;

readbuf[0] = BC\_ENTER\_LOOPER;

binder\_write(bs, readbuf, sizeof(unsigned));

for (;;)

{

bwr.read\_size = sizeof(readbuf);

bwr.read\_consumed = 0;

bwr.read\_buffer = (unsigned) readbuf;

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

if (res < 0) {

LOGE("binder\_loop: ioctl failed (%s)\n", strerror(errno));

break;

}

res = binder\_parse(bs, 0, readbuf, bwr.read\_consumed, func);

if (res == 0) {

LOGE("binder\_loop: unexpected reply?!\n");

break;

}

if (res < 0) {

LOGE("binder\_loop: io error %d %s\n", res, strerror(errno));

break;

}

}

}

注意binder\_loop()的参数func，它的值是svcmgr\_handler()函数指针。而且这个参数会进一步传递给binder\_parse()。

###### 2.5.2.3.3.1 BC\_ENTER\_LOOPER

# 3.智能指针

Android系统的运行时库层代码是用C++来编写的，用C++来写代码最容易出错的地方就是指针了，一旦使用不当，轻则造成内存泄漏，重则造成系统崩溃。不过系统为我们提供了智能指针，避免出现上述问题，本文将系统地分析Android系统智能指针（轻量级指针、强指针和弱指针）的实现原理。

  在使用C++来编写代码的过程中，指针使用不当造成内存泄漏一般就是因为new了一个对象并且使用完之后，忘记了delete这个对象，而造成系统崩溃一般就是因为一个地方delete了这个对象之后，其它地方还在继续使原来指向这个对象的指针。为了避免出现上述问题，一般的做法就是使用引用计数的方法，每当有一个指针指向了一个new出来的对象时，就对这个对象的引用计数增加1，每当有一个指针不再使用这个对象时，就对这个对象的引用计数减少1，每次减1之后，如果发现引用计数值为0时，那么，就要delete这个对象了，这样就避免了忘记delete对象或者这个对象被delete之后其它地方还在使用的问题了。但是，如何实现这个对象的引用计数呢？肯定不是由开发人员来手动地维护了，要开发人员时刻记住什么时候该对这个对象的引用计数加1，什么时候该对这个对象的引用计数减1，一来是不方便开发，二来是不可靠，一不小心哪里多加了一个1或者多减了一个1，就会造成灾难性的后果。这时候，智能指针就粉墨登场了。首先，智能指针是一个对象，不过这个对象代表的是另外一个真实使用的对象，当智能指针指向实际对象的时候，就是智能指针对象创建的时候，当智能指针不再指向实际对象的时候，就是智能指针对象销毁的时候，我们知道，在C++中，对象的创建和销毁时会分别自动地调用对象的构造函数和析构函数，这样，负责对真实对象的引用计数加1和减1的工作就落实到智能指针对象的构造函数和析构函数的身上了，这也是为什么称这个指针对象为智能指针的原因。

      在计算机科学领域中，提供垃圾收集（Garbage Collection）功能的系统框架，即提供对象托管功能的系统框架，例如Java应用程序框架，也是采用上述的引用计数技术方案来实现的，然而，简单的引用计数技术不能处理系统中对象间循环引用的情况。考虑这样的一个场景，系统中有两个对象A和B，在对象A的内部引用了对象B，而在对象B的内部也引用了对象A。当两个对象A和B都不再使用时，垃圾收集系统会发现无法回收这两个对象的所占据的内存的，因为系统一次只能收集一个对象，而无论系统决定要收回对象A还是要收回对象B时，都会发现这个对象被其它的对象所引用，因而就都回收不了，这样就造成了内存泄漏。这样，就要采取另外的一种引用计数技术了，即对象的引用计数同时存在强引用和弱引用两种计数，例如，Apple公司提出的[Cocoa](http://en.wikipedia.org/wiki/Cocoa_(API))框架，当父对象要引用子对象时，就对子对象使用强引用计数技术，而当子对象要引用父对象时，就对父对象使用弱引用计数技术，而当垃圾收集系统执行对象回收工作时，只要发现对象的强引用计数为0，而不管它的弱引用计数是否为0，都可以回收这个对象，但是，如果我们只对一个对象持有弱引用计数，当我们要使用这个对象时，就不直接使用了，必须要把这个弱引用升级成为强引用时，才能使用这个对象，在转换的过程中，如果对象已经不存在，那么转换就失败了，这时候就说明这个对象已经被销毁了，不能再使用了。

      了解了这些背景知识后，我们就可以进一步学习Android系统的智能指针的实现原理了。Android系统提供了强大的智能指针技术供我们使用，这些智能指针实现方案既包括简单的引用计数技术，也包括了复杂的引用计数技术，即对象既有强引用计数，也有弱引用计数，对应地，这三种智能指针分别就称为轻量级指针（Light Pointer）、强指针（Strong Pointer）和弱指针（Weak Pointer）。无论是轻量级指针，还是强指针和弱指针，它们的实现框架都是一致的，即由对象本身来提供引用计数器，但是它不会去维护这个引用计数器的值，而是由智能指针来维护，就好比是对象提供素材，但是具体怎么去使用这些素材，就交给智能指针来处理了。由于不管是什么类型的对象，它都需要提供引用计数器这个素材，在C++中，我们就可以把这个引用计数器素材定义为一个公共类，这个类只有一个成员变量，那就是引用计数成员变量，其它提供智能指针引用的对象，都必须从这个公共类继承下来，这样，这些不同的对象就天然地提供了引用计数器给智能指针使用了。总的来说就是我们在实现智能指会的过程中，第一是要定义一个负责提供引用计数器的公共类，第二是我们要实现相应的智能指针对象类，后面我们会看到这种方案是怎么样实现的。

      接下来，我们就先介绍轻量级指针的实现原理，然后再接着介绍强指针和弱指针的实现原理。

## 3.1 轻量级指针

        先来看一下实现引用计数的类LightRefBase，它定义在frameworks/base/include/utils/RefBase.h文件中：

[cpp] [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. template <class T>
2. class LightRefBase
3. {
4. public:
5. inline LightRefBase() : mCount(0) { }
6. inline void incStrong(const void\* id) const {
7. android\_atomic\_inc(&mCount);
8. }
9. inline void decStrong(const void\* id) const {
10. if (android\_atomic\_dec(&mCount) == 1) {
11. delete static\_cast<const T\*>(this);
12. }
13. }
14. //! DEBUGGING ONLY: Get current strong ref count.
15. inline int32\_t getStrongCount() const {
16. return mCount;
17. }
19. protected:
20. inline ~LightRefBase() { }
22. private:
23. mutable volatile int32\_t mCount;
24. };

      这个类很简单，它只一个成员变量mCount，这就是引用计数器了，它的初始化值为0，另外，这个类还提供两个成员函数incStrong和decStrong来维护引用计数器的值，这两个函数就是提供给智能指针来调用的了，这里要注意的是，在decStrong函数中，如果当前引用计数值为1，那么当减1后就会变成0，于是就会delete这个对象。

      前面说过，要实现自动引用计数，除了要有提供引用计数器的基类外，还需要有智能指针类。在Android系统中，配合LightRefBase引用计数使用的智能指针类便是sp了，它也是定义在frameworks/base/include/utils/RefBase.h文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **template** <**typename** T>
2. **class** sp
3. {
4. **public**:
5. **typedef** **typename** RefBase::weakref\_type weakref\_type;
7. **inline** sp() : m\_ptr(0) { }
9. sp(T\* other);
10. sp(**const** sp<T>& other);
11. **template**<**typename** U> sp(U\* other);
12. **template**<**typename** U> sp(**const** sp<U>& other);
14. ~sp();
16. // Assignment
18. sp& operator = (T\* other);
19. sp& operator = (**const** sp<T>& other);
21. **template**<**typename** U> sp& operator = (**const** sp<U>& other);
22. **template**<**typename** U> sp& operator = (U\* other);
24. //! Special optimization for use by ProcessState (and nobody else).
25. **void** force\_set(T\* other);
27. // Reset
29. **void** clear();
31. // Accessors
33. **inline**  T&      operator\* () **const**  { **return** \*m\_ptr; }
34. **inline**  T\*      operator-> () **const** { **return** m\_ptr;  }
35. **inline**  T\*      get() **const**         { **return** m\_ptr; }
37. // Operators
39. COMPARE(==)
40. COMPARE(!=)
41. COMPARE(>)
42. COMPARE(<)
43. COMPARE(<=)
44. COMPARE(>=)
46. **private**:
47. **template**<**typename** Y> **friend** **class** sp;
48. **template**<**typename** Y> **friend** **class** wp;
50. // Optimization for wp::promote().
51. sp(T\* p, weakref\_type\* refs);
53. T\*              m\_ptr;
54. };

      这个类的内容比较多，但是这里我们只关注它的成员变量m\_ptr、构造函数和析构函数。不难看出，成员变量m\_ptr就是指向真正的对象了，它是在构造函数里面初始化的。接下来我们就再看一下它的两个构造函数，一个是普通构造函数，一个拷贝构造函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **template**<**typename** T>
2. sp<T>::sp(T\* other)
3. : m\_ptr(other)
4. {
5. **if** (other) other->incStrong(**this**);
6. }
8. **template**<**typename** T>
9. sp<T>::sp(**const** sp<T>& other)
10. : m\_ptr(other.m\_ptr)
11. {
12. **if** (m\_ptr) m\_ptr->incStrong(**this**);
13. }

      这两个构造函数都会首先初始化成员变量m\_ptr，然后再调用m\_ptr的incStrong函数来增加对象的引用计数，在我们这个场景中，就是调用LightRefBase类的incStrong函数了。

      最后，看一下析构函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **template**<**typename** T>
2. sp<T>::~sp()
3. {
4. **if** (m\_ptr) m\_ptr->decStrong(**this**);
5. }

      析构函数也很简单，只是调用m\_ptr的成员函数decStrong来减少对象的引用计数值，这里就是调用LightRefBase类的decStrong函数了，前面我们看到，当这个引用计数减1后变成0时，就会自动delete这个对象了。

      轻量级智能指针的实现原理大概就是这样了，比较简单，下面我们再用一个例子来说明它的用法。

## 3.2 轻量级指针的用法

      参考[在Ubuntu上为Android系统内置C可执行程序测试Linux内核驱动程序](http://blog.csdn.net/luoshengyang/article/details/6571210)一文，我们在external目录下建立一个C++工程目录lightpointer，它里面有两个文件，一个lightpointer.cpp文件，另外一个是Android.mk文件。

      源文件lightpointer.cpp的内容如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. #include <stdio.h>
2. #include <utils/RefBase.h>
4. **using** **namespace** android;
6. **class** LightClass : **public** LightRefBase<LightClass>
7. {
8. **public**:
9. LightClass()
10. {
11. printf("Construct LightClass Object.");
12. }
14. **virtual** ~LightClass()
15. {
16. printf("Destory LightClass Object.");
17. }
18. };
20. **int** main(**int** argc, **char**\*\* argv)
21. {
22. LightClass\* pLightClass = **new** LightClass();
23. sp<LightClass> lpOut = pLightClass;
25. printf("Light Ref Count: %d.\n", pLightClass->getStrongCount());
27. {
28. sp<LightClass> lpInner = lpOut;
30. printf("Light Ref Count: %d.\n", pLightClass->getStrongCount());
31. }
33. printf("Light Ref Count: %d.\n", pLightClass->getStrongCount());
35. **return** 0;
36. }

      我们创建一个自己的类LightClass，继承了LightRefBase模板类，这样类LightClass就具有引用计数的功能了。在main函数里面，我们首先new一个LightClass对象，然后把这个对象赋值给智能指针lpOut，这时候通过一个printf语句来将当前对象的引用计数值打印出来，从前面的分析可以看出，如果一切正常的话，这里打印出来的引用计数值为1。接着，我们又在两个大括号里面定义了另外一个智能指针lpInner，它通过lpOut间接地指向了前面我们所创建的对象，这时候再次将当前对象的引用计数值打印出来，从前面 的分析也可以看出，如果一切正常的话，这里打印出来的引用计数值应该为2。程序继承往下执行，当出了大括号的范围的时候，智能指针对象lpInner就被析构了，从前面的分析可以知道，智能指针在析构的时候，会减少当前对象的引用计数值，因此，最后一个printf语句打印出来的引用计数器值应该为1。当main函数执行完毕后，智能指针lpOut也会被析构，被析构时，它会再次减少当前对象的引用计数，这时候，对象的引用计数值就为0了，于是，它就会被delete了。

      编译脚本文件Android.mk的内容如下：

**[plain]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. LOCAL\_PATH := $(call my-dir)
2. include $(CLEAR\_VARS)
3. LOCAL\_MODULE\_TAGS := optional
4. LOCAL\_MODULE := lightpointer
5. LOCAL\_SRC\_FILES := lightpointer.cpp
6. LOCAL\_SHARED\_LIBRARIES := \
7. libcutils \
8. libutils
9. include $(BUILD\_EXECUTABLE)

     最后，我们参照[如何单独编译Android源代码中的模块](http://blog.csdn.net/luoshengyang/article/details/6566662)一文，使用mmm命令对工程进行编译：

**[plain]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. USER-NAME@MACHINE-NAME:~/Android$ mmm ./external/lightpointer

      编译之后，就可以打包了：

**[plain]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. USER-NAME@MACHINE-NAME:~/Android$ make snod

      最后得到可执行程序lightpointer就位于设备上的/system/bin/目录下。启动模拟器，通过adb shell命令进入到模拟器终端，进入到/system/bin/目录，执行lightpointer可执行程序，验证程序是否按照我们设计的逻辑运行：

**[plain]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. USER-NAME@MACHINE-NAME:~/Android$ adb shell
2. root@android:/ # cd system/bin/
3. root@android:/system/bin # ./lightpointer
4. Construct LightClass Object.
5. Light Ref Count: 1.
6. Light Ref Count: 2.
7. Light Ref Count: 1.
8. Destory LightClass Object.

     这里可以看出，程序一切都是按照我们的设计来运行，这也验证了我们上面分析的轻量级智能指针的实现原理。

## 3.3 强指针

      强指针所使用的引用计数类为RefBase，它LightRefBase类要复杂多了，所以才称后者为轻量级的引用计数基类吧。我们先来看看RefBase类的实现，它定义在frameworks/base/include/utils/RefBase.h文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **class** RefBase
2. {
3. **public**:
4. **void**            incStrong(**const** **void**\* id) **const**;
5. **void**            decStrong(**const** **void**\* id) **const**;
7. **void**            forceIncStrong(**const** **void**\* id) **const**;
9. //! DEBUGGING ONLY: Get current strong ref count.
10. int32\_t         getStrongCount() **const**;
12. **class** weakref\_type
13. {
14. **public**:
15. RefBase\*            refBase() **const**;
17. **void**                incWeak(**const** **void**\* id);
18. **void**                decWeak(**const** **void**\* id);
20. **bool**                attemptIncStrong(**const** **void**\* id);
22. //! This is only safe if you have set OBJECT\_LIFETIME\_FOREVER.
23. **bool**                attemptIncWeak(**const** **void**\* id);
25. //! DEBUGGING ONLY: Get current weak ref count.
26. int32\_t             getWeakCount() **const**;
28. //! DEBUGGING ONLY: Print references held on object.
29. **void**                printRefs() **const**;
31. //! DEBUGGING ONLY: Enable tracking for this object.
32. // enable -- enable/disable tracking
33. // retain -- when tracking is enable, if true, then we save a stack trace
34. //           for each reference and dereference; when retain == false, we
35. //           match up references and dereferences and keep only the
36. //           outstanding ones.
38. **void**                trackMe(**bool** enable, **bool** retain);
39. };
41. weakref\_type\*   createWeak(**const** **void**\* id) **const**;
43. weakref\_type\*   getWeakRefs() **const**;
45. //! DEBUGGING ONLY: Print references held on object.
46. **inline**  **void**            printRefs() **const** { getWeakRefs()->printRefs(); }
48. //! DEBUGGING ONLY: Enable tracking of object.
49. **inline**  **void**            trackMe(**bool** enable, **bool** retain)
50. {
51. getWeakRefs()->trackMe(enable, retain);
52. }
54. **protected**:
55. RefBase();
56. **virtual**                 ~RefBase();
58. //! Flags for extendObjectLifetime()
59. **enum** {
60. OBJECT\_LIFETIME\_WEAK    = 0x0001,
61. OBJECT\_LIFETIME\_FOREVER = 0x0003
62. };
64. **void**            extendObjectLifetime(int32\_t mode);
66. //! Flags for onIncStrongAttempted()
67. **enum** {
68. FIRST\_INC\_STRONG = 0x0001
69. };
71. **virtual** **void**            onFirstRef();
72. **virtual** **void**            onLastStrongRef(**const** **void**\* id);
73. **virtual** **bool**            onIncStrongAttempted(uint32\_t flags, **const** **void**\* id);
74. **virtual** **void**            onLastWeakRef(**const** **void**\* id);
76. **private**:
77. **friend** **class** weakref\_type;
78. **class** weakref\_impl;
80. RefBase(**const** RefBase& o);
81. RefBase&        operator=(**const** RefBase& o);
83. weakref\_impl\* **const** mRefs;
84. };

      RefBase类和LightRefBase类一样，提供了incStrong和decStrong成员函数来操作它的引用计数器；而RefBase类与LightRefBase类最大的区别是，它不像LightRefBase类一样直接提供一个整型值（mutable volatile int32\_t mCount）来维护对象的引用计数，前面我们说过，复杂的引用计数技术同时支持强引用计数和弱引用计数，在RefBase类中，这两种计数功能是通过其成员变量mRefs来提供的。

      RefBase类的成员变量mRefs的类型为weakref\_impl指针，它实现在frameworks/base/libs/utils/RefBase.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **class** RefBase::weakref\_impl : **public** RefBase::weakref\_type
2. {
3. **public**:
4. **volatile** int32\_t    mStrong;
5. **volatile** int32\_t    mWeak;
6. RefBase\* **const**      mBase;
7. **volatile** int32\_t    mFlags;

10. #if !DEBUG\_REFS
12. weakref\_impl(RefBase\* base)
13. : mStrong(INITIAL\_STRONG\_VALUE)
14. , mWeak(0)
15. , mBase(base)
16. , mFlags(0)
17. {
18. }
20. **void** addStrongRef(**const** **void**\* /\*id\*/) { }
21. **void** removeStrongRef(**const** **void**\* /\*id\*/) { }
22. **void** addWeakRef(**const** **void**\* /\*id\*/) { }
23. **void** removeWeakRef(**const** **void**\* /\*id\*/) { }
24. **void** printRefs() **const** { }
25. **void** trackMe(**bool**, **bool**) { }
27. #else
28. weakref\_impl(RefBase\* base)
29. : mStrong(INITIAL\_STRONG\_VALUE)
30. , mWeak(0)
31. , mBase(base)
32. , mFlags(0)
33. , mStrongRefs(NULL)
34. , mWeakRefs(NULL)
35. , mTrackEnabled(!!DEBUG\_REFS\_ENABLED\_BY\_DEFAULT)
36. , mRetain(**false**)
37. {
38. //LOGI("NEW weakref\_impl %p for RefBase %p", this, base);
39. }
41. ~weakref\_impl()
42. {
43. LOG\_ALWAYS\_FATAL\_IF(!mRetain && mStrongRefs != NULL, "Strong references remain!");
44. LOG\_ALWAYS\_FATAL\_IF(!mRetain && mWeakRefs != NULL, "Weak references remain!");
45. }
47. **void** addStrongRef(**const** **void**\* id)
48. {
49. addRef(&mStrongRefs, id, mStrong);
50. }
52. **void** removeStrongRef(**const** **void**\* id)
53. {
54. **if** (!mRetain)
55. removeRef(&mStrongRefs, id);
56. **else**
57. addRef(&mStrongRefs, id, -mStrong);
58. }
60. **void** addWeakRef(**const** **void**\* id)
61. {
62. addRef(&mWeakRefs, id, mWeak);
63. }
64. **void** removeWeakRef(**const** **void**\* id)
65. {
66. **if** (!mRetain)
67. removeRef(&mWeakRefs, id);
68. **else**
69. addRef(&mWeakRefs, id, -mWeak);
70. }
72. **void** trackMe(**bool** track, **bool** retain)
73. {
74. mTrackEnabled = track;
75. mRetain = retain;
76. }
78. ......
80. **private**:
81. **struct** ref\_entry
82. {
83. ref\_entry\* next;
84. **const** **void**\* id;
85. #if DEBUG\_REFS\_CALLSTACK\_ENABLED
86. CallStack stack;
87. #endif
88. int32\_t ref;
89. };
91. **void** addRef(ref\_entry\*\* refs, **const** **void**\* id, int32\_t mRef)
92. {
93. **if** (mTrackEnabled) {
94. AutoMutex \_l(mMutex);
95. ref\_entry\* ref = **new** ref\_entry;
96. // Reference count at the time of the snapshot, but before the
97. // update.  Positive value means we increment, negative--we
98. // decrement the reference count.
99. ref->ref = mRef;
100. ref->id = id;
101. #if DEBUG\_REFS\_CALLSTACK\_ENABLED
102. ref->stack.update(2);
103. #endif
105. ref->next = \*refs;
106. \*refs = ref;
107. }
108. }
110. **void** removeRef(ref\_entry\*\* refs, **const** **void**\* id)
111. {
112. **if** (mTrackEnabled) {
113. AutoMutex \_l(mMutex);
115. ref\_entry\* ref = \*refs;
116. **while** (ref != NULL) {
117. **if** (ref->id == id) {
118. \*refs = ref->next;
119. **delete** ref;
120. **return**;
121. }
123. refs = &ref->next;
124. ref = \*refs;
125. }
127. LOG\_ALWAYS\_FATAL("RefBase: removing id %p on RefBase %p (weakref\_type %p) that doesn't exist!",
128. id, mBase, **this**);
129. }
130. }
132. ......
134. Mutex mMutex;
135. ref\_entry\* mStrongRefs;
136. ref\_entry\* mWeakRefs;
138. **bool** mTrackEnabled;
139. // Collect stack traces on addref and removeref, instead of deleting the stack references
140. // on removeref that match the address ones.
141. **bool** mRetain;
143. ......
144. #endif
145. };

     这个类看起来实现得很复杂，其实不然，这个类的实现可以分成两部分：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. #if !DEBUG\_REFS
3. ......
5. #else

     编译指令之间的这部分源代码是Release版本的源代码，它的成员函数都是空实现；

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. #else
3. ......
5. #endif

     编译指令之间的部分源代码是Debug版本的源代码，它的成员函数都是有实现的，实现这些函数的目的都是为了方便开发人员调试引用计数用的，除此之外，还在内部实现了一个结构体ref\_entry：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **struct** ref\_entry
2. {
3. ref\_entry\* next;
4. **const** **void**\* id;
5. #if DEBUG\_REFS\_CALLSTACK\_ENABLED
6. CallStack stack;
7. #endif
8. int32\_t ref;
9. };

      这个结构体也是为了方便调试而使用的，我们可以不关注这部分用于调试的代码。

总的来说，weakref\_impl类只要提供了以下四个成员变量来维护对象的引用计数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **volatile** int32\_t    mStrong;
2. **volatile** int32\_t    mWeak;
3. RefBase\* **const**      mBase;
4. **volatile** int32\_t    mFlags;

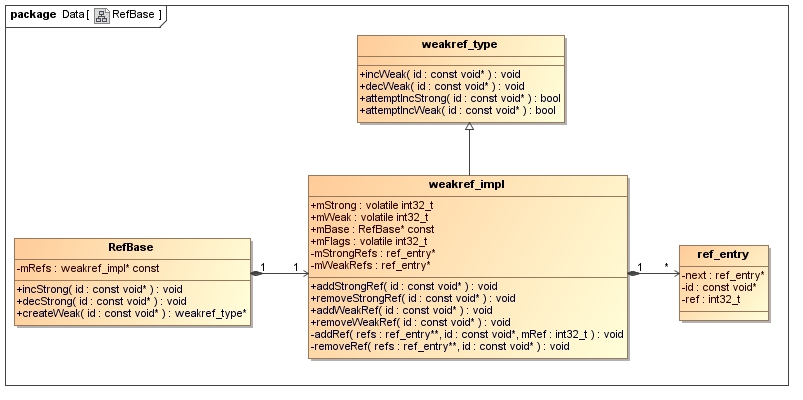
      其中mStrong和mWeak分别表示对象的强引用计数和弱引用计数；RefBase类包含了一个weakref\_impl类指针mRefs，而这里的weakref\_impl类也有一个成员变量mBase来指向它的宿主类RefBase；mFlags是一个标志位，它指示了维护对象引用计数所使用的策略，后面我们将会分析到，它的取值为0，或者以下的枚举值：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. //! Flags for extendObjectLifetime()
2. **enum** {
3. OBJECT\_LIFETIME\_WEAK    = 0x0001,
4. OBJECT\_LIFETIME\_FOREVER = 0x0003
5. };

      这里我们还需要注意的一点的是，从weakref\_impl的类名来看，它应该是一个实现类，那么，就必然有一个对应的接口类，这个对应的接口类的就是RefBase类内部定义的weakref\_type类了，这是一种把类的实现与接口定义分离的设计方法。学习过设计模式的读者应该知道，在设计模式里面，非常强调类的接口定义和类的实现分离，以便利于后续扩展和维护，这里就是用到了这种设计思想。

      说了这多，RefBase类给人的感觉还是挺复杂的，不要紧，我们一步步来，先通过下面这个图来梳理一下这些类之间的关系：



      从这个类图可以看出，每一个RefBase对象包含了一个weakref\_impl对象，而weakref\_impl对象实现了weakref\_type接口，同时它可以包含多个ref\_entry对象，前面说过，ref\_entry是调试用的一个结构体，实际使用中可以不关注。

      提供引用计数器的类RefBase我们就暂时介绍到这里，后面我们再结合智能指针类一起分析，现在先来看看强指针类和弱指针类的定义。强指针类的定义我们在前面介绍轻量级指针的时候已经见到了，就是sp类了，这里就不再把它的代码列出来了。我们来看看它的构造函数的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **template**<**typename** T>
2. sp<T>::sp(T\* other)
3. : m\_ptr(other)
4. {
5. **if** (other) other->incStrong(**this**);
6. }

       这里传进来的参数other一定是继承于RefBase类的，因此，在函数的内部，它调用的是RefBase类的incStrong函数，它定义在frameworks/base/libs/utils/RefBase.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **void** RefBase::incStrong(**const** **void**\* id) **const**
2. {
3. weakref\_impl\* **const** refs = mRefs;
4. refs->addWeakRef(id);
5. refs->incWeak(id);
6. refs->addStrongRef(id);
8. **const** int32\_t c = android\_atomic\_inc(&refs->mStrong);
9. LOG\_ASSERT(c > 0, "incStrong() called on %p after last strong ref", refs);
11. #if PRINT\_REFS
12. LOGD("incStrong of %p from %p: cnt=%d\n", **this**, id, c);
13. #endif
15. **if** (c != INITIAL\_STRONG\_VALUE) {
16. **return**;
17. }
19. android\_atomic\_add(-INITIAL\_STRONG\_VALUE, &refs->mStrong);
20. **const\_cast**<RefBase\*>(**this**)->onFirstRef();
21. }

       成员变量mRefs是在RefBase类的构造函数中创建的：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. RefBase::RefBase()
2. : mRefs(**new** weakref\_impl(**this**))
3. {
4. //    LOGV("Creating refs %p with RefBase %p\n", mRefs, this);
5. }

       在这个incStrong函数中，主要做了三件事情：

       一是增加弱引用计数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. refs->addWeakRef(id);
2. refs->incWeak(id);

       二是增加强引用计数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. refs->addStrongRef(id);
2. **const** int32\_t c = android\_atomic\_inc(&refs->mStrong);

      三是如果发现是首次调用这个对象的incStrong函数，就会调用一个这个对象的onFirstRef函数，让对象有机会在对象被首次引用时做一些处理逻辑：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **if** (c != INITIAL\_STRONG\_VALUE)  {
2. **return**;
3. }
5. android\_atomic\_add(-INITIAL\_STRONG\_VALUE, &refs->mStrong);
6. **const\_cast**<RefBase\*>(**this**)->onFirstRef();

      这里的c返回的是refs->mStrong加1前的值，如果发现等于INITIAL\_STRONG\_VALUE，就说明这个对象的强引用计数是第一次被增加，因此，refs->mStrong就是初始化为INITIAL\_STRONG\_VALUE的，它的值为：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. #define INITIAL\_STRONG\_VALUE (1<<28)

      这个值加1后等于1<<28 + 1，不等于1，因此，后面要再减去-INITIAL\_STRONG\_VALUE，于是，refs->mStrong就等于1了，就表示当前对象的强引用计数值为1了，这与这个对象是第一次被增加强引用计数值的逻辑是一致的。

     回过头来看弱引用计数是如何增加的，首先是调用weakref\_impl类的addWeakRef函数，我们知道，在Release版本中，这个函数也不做，而在Debug版本中，这个函数增加了一个ref\_entry对象到了weakref\_impl对象的mWeakRefs列表中，表示此weakref\_impl对象的弱引用计数被增加了一次。接着又调用了weakref\_impl类的incWeak函数，真正增加弱引用计数值就是在这个函数实现的了，weakref\_impl类的incWeak函数继承于其父类weakref\_type的incWeak函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **void** RefBase::weakref\_type::incWeak(**const** **void**\* id)
2. {
3. weakref\_impl\* **const** impl = **static\_cast**<weakref\_impl\*>(**this**);
4. impl->addWeakRef(id);
5. **const** int32\_t c = android\_atomic\_inc(&impl->mWeak);
6. LOG\_ASSERT(c >= 0, "incWeak called on %p after last weak ref", **this**);
7. }

       增加弱引用计数是下面语句执行的：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **const** int32\_t c = android\_atomic\_inc(&impl->mWeak);

     但是前面为什么又调用了一次addWeakRef函数呢？前面不是已经调用过了吗？在Release版本中，因为weakref\_impl类的addWeakRef函数是空实现，这里再调用一次没有什么害处，但是如果在Debug版本，岂不是冗余了吗？搞不清，有人问过负责开发[Android系统Binder通信机制](http://blog.csdn.net/luoshengyang/article/details/6618363)模块的作者[Dianne Hackborn](http://www.angryredplanet.com/~hackbod/)这个问题，他是这样回答的：

<http://groups.google.com/group/android-platform/browse_thread/thread/cc641db8487dd83>

**Ah I see.  Well the debug code may be broken, though I wouldn't leap to that         conclusion without actually testing it; I know it has been used in the         past.  Anyway, these things get compiled out in non-debug builds, so there         is no reason to change them unless you are actually trying to use this debug         code and it isn't working and need to do this to fix it.**

      既然他也不知道怎么回事，我们也不必深究了，知道有这么回事就行。

      这里总结一下强指针类sp在其构造函数里面所做的事情就是分别为目标对象的强引用计数和弱引和计数增加了1。

      再来看看强指针类的析构函数的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **template**<**typename** T>
2. sp<T>::~sp()
3. {
4. **if** (m\_ptr) m\_ptr->decStrong(**this**);
5. }

      同样，这里的m\_ptr指向的目标对象一定是继承了RefBase类的，因此，这里调用的是RefBase类的decStrong函数，这也是定义在frameworks/base/libs/utils/RefBase.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **void** RefBase::decStrong(**const** **void**\* id) **const**
2. {
3. weakref\_impl\* **const** refs = mRefs;
4. refs->removeStrongRef(id);
5. **const** int32\_t c = android\_atomic\_dec(&refs->mStrong);
6. #if PRINT\_REFS
7. LOGD("decStrong of %p from %p: cnt=%d\n", **this**, id, c);
8. #endif
9. LOG\_ASSERT(c >= 1, "decStrong() called on %p too many times", refs);
10. **if** (c == 1) {
11. **const\_cast**<RefBase\*>(**this**)->onLastStrongRef(id);
12. **if** ((refs->mFlags&OBJECT\_LIFETIME\_WEAK) != OBJECT\_LIFETIME\_WEAK) {
13. **delete** **this**;
14. }
15. }
16. refs->removeWeakRef(id);
17. refs->decWeak(id);
18. }

      这里的refs->removeStrongRef函数调用语句是对应前面在RefBase::incStrong函数里的refs->addStrongRef函数调用语句的，在Release版本中，这也是一个空实现函数，真正实现强引用计数减1的操作是下面语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **const** int32\_t c = android\_atomic\_dec(&refs->mStrong);

      如果发现减1前，此对象的强引用计数为1，就说明从此以后，就再没有地方引用这个目标对象了，这时候，就要看看是否要delete这个目标对象了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **if** (c == 1) {
2. **const\_cast**<RefBase\*>(**this**)->onLastStrongRef(id);
3. **if** ((refs->mFlags&OBJECT\_LIFETIME\_WEAK) != OBJECT\_LIFETIME\_WEAK) {
4. **delete** **this**;
5. }
6. }

      在强引用计数为0的情况下，如果对象的标志位OBJECT\_LIFETIME\_WEAK被设置了，就说明这个对象的生命周期是受弱引用计数所控制的，因此，这时候就不能delete对象，要等到弱引用计数也为0的情况下，才能delete这个对象。

      接下来的ref->removeWeakRef函数调用语句是对应前面在RefBase::incStrong函数里的refs->addWeakRef函数调用语句的，在Release版本中，这也是一个空实现函数，真正实现强引用计数减1的操作下面的refs->decWeak函数，weakref\_impl类没有实现自己的decWeak函数，它继承了weakref\_type类的decWeak函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **void** RefBase::weakref\_type::decWeak(**const** **void**\* id)
2. {
3. weakref\_impl\* **const** impl = **static\_cast**<weakref\_impl\*>(**this**);
4. impl->removeWeakRef(id);
5. **const** int32\_t c = android\_atomic\_dec(&impl->mWeak);
6. LOG\_ASSERT(c >= 1, "decWeak called on %p too many times", **this**);
7. **if** (c != 1) **return**;
9. **if** ((impl->mFlags&OBJECT\_LIFETIME\_WEAK) != OBJECT\_LIFETIME\_WEAK) {
10. **if** (impl->mStrong == INITIAL\_STRONG\_VALUE)
11. **delete** impl->mBase;
12. **else** {
13. //            LOGV("Freeing refs %p of old RefBase %p\n", this, impl->mBase);
14. **delete** impl;
15. }
16. } **else** {
17. impl->mBase->onLastWeakRef(id);
18. **if** ((impl->mFlags&OBJECT\_LIFETIME\_FOREVER) != OBJECT\_LIFETIME\_FOREVER) {
19. **delete** impl->mBase;
20. }
21. }
22. }

    这里又一次调用了weakref\_impl对象的removeWeakRef函数，这也是和RefBase::weakref\_type::incWeak函数里面的impl->addWeakRef语句所对应的，实现弱引用计数减1的操作是下面语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **const** int32\_t c = android\_atomic\_dec(&impl->mWeak);

      减1前如果发现不等于1，那么就什么也不用做就返回了，如果发现等于1，就说明当前对象的弱引用计数值为0了，这时候，就要看看是否要delete这个对象了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **if** ((impl->mFlags&OBJECT\_LIFETIME\_WEAK) != OBJECT\_LIFETIME\_WEAK) {
2. **if** (impl->mStrong == INITIAL\_STRONG\_VALUE)
3. **delete** impl->mBase;
4. **else** {
5. //      LOGV("Freeing refs %p of old RefBase %p\n", this, impl->mBase);
6. **delete** impl;
7. }
8. } **else** {
9. impl->mBase->onLastWeakRef(id);
10. **if** ((impl->mFlags&OBJECT\_LIFETIME\_FOREVER) != OBJECT\_LIFETIME\_FOREVER) {
11. **delete** impl->mBase;
12. }
13. }

      如果目标对象的生命周期是不受弱引用计数控制的，就执行下面语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **if** (impl->mStrong == INITIAL\_STRONG\_VALUE)
2. **delete** impl->mBase;
3. **else** {
4. //  LOGV("Freeing refs %p of old RefBase %p\n", this, impl->mBase);
5. **delete** impl;
6. }

      这个代码段是什么意思呢？这里是减少对象的弱引用计数的地方，如果调用到这里，那么就说明前面一定有增加过此对象的弱引用计数，而增加对象的弱引用计数有两种场景的，一种场景是增加对象的强引用计数的时候，会同时增加对象的弱引用计数，另一种场景是当我们使用一个弱指针来指向对象时，在弱指针对象的构造函数里面，也会增加对象的弱引用计数，不过这时候，就只是增加对象的弱引用计数了，并没有同时增加对象的强引用计数。因此，这里在减少对象的弱引用计数时，就要分两种情况来考虑。

      如果是前一种场景，这里的impl->mStrong就必然等于0，而不会等于INITIAL\_STRONG\_VALUE值，因此，这里就不需要delete目标对象了（impl->mBase），因为前面的RefBase::decStrong函数会负责delete这个对象。这里唯一需要做的就是把weakref\_impl对象delete掉，但是，为什么要在这里delete这个weakref\_impl对象呢？这里的weakref\_impl对象是在RefBase的构造函数里面new出来的，理论上说应该在在RefBase的析构函数里delete掉这个weakref\_impl对象的。在RefBase的析构函数里面，的确是会做这件事情：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. RefBase::~RefBase()
2. {
3. //    LOGV("Destroying RefBase %p (refs %p)\n", this, mRefs);
4. **if** (mRefs->mWeak == 0) {
5. //        LOGV("Freeing refs %p of old RefBase %p\n", mRefs, this);
6. **delete** mRefs;
7. }
8. }

      但是不要忘记，在这个场景下，目标对象是前面的RefBase::decStrong函数delete掉的，这时候目标对象就会被析构，但是它的弱引用计数值尚未执行减1操作，因此，这里的mRefs->mWeak == 0条件就不成立，于是就不会delete这个weakref\_impl对象，因此，就延迟到执行这里decWeak函数时再执行。

      如果是后一种情景，这里的impl->mStrong值就等于INITIAL\_STRONG\_VALUE了，这时候由于没有地方会负责delete目标对象，因此，就需要把目标对象（imp->mBase）delete掉了，否则就会造成内存泄漏。在delete这个目标对象的时候，就会执行RefBase类的析构函数，这时候目标对象的弱引用计数等于0，于是，就会把weakref\_impl对象也一起delete掉了。

      回到外层的if语句中，如果目标对象的生命周期是受弱引用计数控制的，就执行下面语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. impl->mBase->onLastWeakRef(id);
2. **if** ((impl->mFlags&OBJECT\_LIFETIME\_FOREVER) != OBJECT\_LIFETIME\_FOREVER) {
3. **delete** impl->mBase;
4. }

      理论上说，如果目标对象的生命周期是受弱引用计数控制的，那么当强引用计数和弱引用计数都为0的时候，这时候就应该delete目标对象了，但是这里还有另外一层控制，我们可以设置目标对象的标志值为OBJECT\_LIFETIME\_FOREVER，即目标对象的生命周期完全不受强引用计数和弱引用计数控制，在这种情况下，即使目标对象的强引用计数和弱引用计数都同时为0，这里也不能delete这个目标对象，那么，由谁来delete掉呢？当然是谁new出来的，就谁来delete掉了，这时候智能指针就完全退化为普通指针了，这里的智能指针设计的非常强大。

      分析到这里，有必要小结一下：

      A. 如果对象的标志位被设置为0，那么只要发现对象的强引用计数值为0，那就会自动delete掉这个对象；

      B. 如果对象的标志位被设置为OBJECT\_LIFETIME\_WEAK，那么只有当对象的强引用计数和弱引用计数都为0的时候，才会自动delete掉这个对象；

      C. 如果对象的标志位被设置为OBJECT\_LIFETIME\_FOREVER，那么对象就永远不会自动被delete掉，谁new出来的对象谁来delete掉。

      到了这里，强指针就分析完成了，最后来分析弱指针。

## 3.4. 弱指针

      弱指针所使用的引用计数类与强指针一样，都是RefBase类，因此，这里就不再重复介绍了，我们直接来弱指针的实现，它定义在frameworks/base/include/utils/RefBase.h文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **template** <**typename** T>
2. **class** wp
3. {
4. **public**:
5. **typedef** **typename** RefBase::weakref\_type weakref\_type;
7. **inline** wp() : m\_ptr(0) { }
9. wp(T\* other);
10. wp(**const** wp<T>& other);
11. wp(**const** sp<T>& other);
12. **template**<**typename** U> wp(U\* other);
13. **template**<**typename** U> wp(**const** sp<U>& other);
14. **template**<**typename** U> wp(**const** wp<U>& other);
16. ~wp();
18. // Assignment
20. wp& operator = (T\* other);
21. wp& operator = (**const** wp<T>& other);
22. wp& operator = (**const** sp<T>& other);
24. **template**<**typename** U> wp& operator = (U\* other);
25. **template**<**typename** U> wp& operator = (**const** wp<U>& other);
26. **template**<**typename** U> wp& operator = (**const** sp<U>& other);
28. **void** set\_object\_and\_refs(T\* other, weakref\_type\* refs);
30. // promotion to sp
32. sp<T> promote() **const**;
34. // Reset
36. **void** clear();
38. // Accessors
40. **inline**  weakref\_type\* get\_refs() **const** { **return** m\_refs; }
42. **inline**  T\* unsafe\_get() **const** { **return** m\_ptr; }
44. // Operators
46. COMPARE\_WEAK(==)
47. COMPARE\_WEAK(!=)
48. COMPARE\_WEAK(>)
49. COMPARE\_WEAK(<)
50. COMPARE\_WEAK(<=)
51. COMPARE\_WEAK(>=)
53. **inline** **bool** operator == (**const** wp<T>& o) **const** {
54. **return** (m\_ptr == o.m\_ptr) && (m\_refs == o.m\_refs);
55. }
56. **template**<**typename** U>
57. **inline** **bool** operator == (**const** wp<U>& o) **const** {
58. **return** m\_ptr == o.m\_ptr;
59. }
61. **inline** **bool** operator > (**const** wp<T>& o) **const** {
62. **return** (m\_ptr == o.m\_ptr) ? (m\_refs > o.m\_refs) : (m\_ptr > o.m\_ptr);
63. }
64. **template**<**typename** U>
65. **inline** **bool** operator > (**const** wp<U>& o) **const** {
66. **return** (m\_ptr == o.m\_ptr) ? (m\_refs > o.m\_refs) : (m\_ptr > o.m\_ptr);
67. }
69. **inline** **bool** operator < (**const** wp<T>& o) **const** {
70. **return** (m\_ptr == o.m\_ptr) ? (m\_refs < o.m\_refs) : (m\_ptr < o.m\_ptr);
71. }
72. **template**<**typename** U>
73. **inline** **bool** operator < (**const** wp<U>& o) **const** {
74. **return** (m\_ptr == o.m\_ptr) ? (m\_refs < o.m\_refs) : (m\_ptr < o.m\_ptr);
75. }
76. **inline** **bool** operator != (**const** wp<T>& o) **const** { **return** m\_refs != o.m\_refs; }
77. **template**<**typename** U> **inline** **bool** operator != (**const** wp<U>& o) **const** { **return** !operator == (o); }
78. **inline** **bool** operator <= (**const** wp<T>& o) **const** { **return** !operator > (o); }
79. **template**<**typename** U> **inline** **bool** operator <= (**const** wp<U>& o) **const** { **return** !operator > (o); }
80. **inline** **bool** operator >= (**const** wp<T>& o) **const** { **return** !operator < (o); }
81. **template**<**typename** U> **inline** **bool** operator >= (**const** wp<U>& o) **const** { **return** !operator < (o); }
83. **private**:
84. **template**<**typename** Y> **friend** **class** sp;
85. **template**<**typename** Y> **friend** **class** wp;
87. T\*              m\_ptr;
88. weakref\_type\*   m\_refs;
89. };

      与强指针类相比，它们都有一个成员变量m\_ptr指向目标对象，但是弱指针还有一个额外的成员变量m\_refs，它的类型是weakref\_type指针，下面我们分析弱指针的构造函数时再看看它是如果初始化的。这里我们需要关注的仍然是弱指针的构造函数和析构函数。

        先来看构造函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **template**<**typename** T>
2. wp<T>::wp(T\* other)
3. : m\_ptr(other)
4. {
5. **if** (other) m\_refs = other->createWeak(**this**);
6. }

      这里的参数other一定是继承了RefBase类，因此，这里调用了RefBase类的createWeak函数，它定义在frameworks/base/libs/utils/RefBase.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. RefBase::weakref\_type\* RefBase::createWeak(**const** **void**\* id) **const**
2. {
3. mRefs->incWeak(id);
4. **return** mRefs;
5. }

      这里的成员变量mRefs的类型为weakref\_impl指针，weakref\_impl类的incWeak函数我们在前面已经看过了，它的作用就是增加对象的弱引用计数。函数最后返回mRefs，于是，弱指针对象的成员变量m\_refs就指向目标对象的weakref\_impl对象了。

       再来看析构函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **template**<**typename** T>
2. wp<T>::~wp()
3. {
4. **if** (m\_ptr) m\_refs->decWeak(**this**);
5. }

     这里，弱指针在析构的时候，与强指针析构不一样，它直接就调用目标对象的weakref\_impl对象的decWeak函数来减少弱引用计数了，当弱引用计数为0的时候，就会根据在目标对象的标志位（0、OBJECT\_LIFETIME\_WEAK或者OBJECT\_LIFETIME\_FOREVER）来决定是否要delete目标对象，前面我们已经介绍过了，这里就不再介绍了。

     分析到这里，弱指针还没介绍完，它最重要的特性我们还没有分析到。前面我们说过，弱指针的最大特点是它不能直接操作目标对象，这是怎么样做到的呢？秘密就在于弱指针类没有重载\*和->操作符号，而强指针重载了这两个操作符号。但是，如果我们要操作目标对象，应该怎么办呢，这就要把弱指针升级为强指针了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **template**<**typename** T>
2. sp<T> wp<T>::promote() **const**
3. {
4. **return** sp<T>(m\_ptr, m\_refs);
5. }

      升级的方式就使用成员变量m\_ptr和m\_refs来构造一个强指针sp，这里的m\_ptr为指目标对象的一个指针，而m\_refs则是指向目标对象里面的weakref\_impl对象。

      我们再来看看这个强指针的构造过程：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **template**<**typename** T>
2. sp<T>::sp(T\* p, weakref\_type\* refs)
3. : m\_ptr((p && refs->attemptIncStrong(**this**)) ? p : 0)
4. {
5. }

      主要就是初始化指向目标对象的成员变量m\_ptr了，如果目标对象还存在，这个m\_ptr就指向目标对象，如果目标对象已经不存在，m\_ptr就为NULL，升级成功与否就要看refs->attemptIncStrong函数的返回结果了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **bool** RefBase::weakref\_type::attemptIncStrong(**const** **void**\* id)
2. {
3. incWeak(id);
5. weakref\_impl\* **const** impl = **static\_cast**<weakref\_impl\*>(**this**);
7. int32\_t curCount = impl->mStrong;
8. LOG\_ASSERT(curCount >= 0, "attemptIncStrong called on %p after underflow",
9. **this**);
10. **while** (curCount > 0 && curCount != INITIAL\_STRONG\_VALUE) {
11. **if** (android\_atomic\_cmpxchg(curCount, curCount+1, &impl->mStrong) == 0) {
12. **break**;
13. }
14. curCount = impl->mStrong;
15. }
17. **if** (curCount <= 0 || curCount == INITIAL\_STRONG\_VALUE) {
18. **bool** allow;
19. **if** (curCount == INITIAL\_STRONG\_VALUE) {
20. // Attempting to acquire first strong reference...  this is allowed
21. // if the object does NOT have a longer lifetime (meaning the
22. // implementation doesn't need to see this), or if the implementation
23. // allows it to happen.
24. allow = (impl->mFlags&OBJECT\_LIFETIME\_WEAK) != OBJECT\_LIFETIME\_WEAK
25. || impl->mBase->onIncStrongAttempted(FIRST\_INC\_STRONG, id);
26. } **else** {
27. // Attempting to revive the object...  this is allowed
28. // if the object DOES have a longer lifetime (so we can safely
29. // call the object with only a weak ref) and the implementation
30. // allows it to happen.
31. allow = (impl->mFlags&OBJECT\_LIFETIME\_WEAK) == OBJECT\_LIFETIME\_WEAK
32. && impl->mBase->onIncStrongAttempted(FIRST\_INC\_STRONG, id);
33. }
34. **if** (!allow) {
35. decWeak(id);
36. **return** **false**;
37. }
38. curCount = android\_atomic\_inc(&impl->mStrong);
40. // If the strong reference count has already been incremented by
41. // someone else, the implementor of onIncStrongAttempted() is holding
42. // an unneeded reference.  So call onLastStrongRef() here to remove it.
43. // (No, this is not pretty.)  Note that we MUST NOT do this if we
44. // are in fact acquiring the first reference.
45. **if** (curCount > 0 && curCount < INITIAL\_STRONG\_VALUE) {
46. impl->mBase->onLastStrongRef(id);
47. }
48. }
50. impl->addWeakRef(id);
51. impl->addStrongRef(id);
53. #if PRINT\_REFS
54. LOGD("attemptIncStrong of %p from %p: cnt=%d\n", **this**, id, curCount);
55. #endif
57. **if** (curCount == INITIAL\_STRONG\_VALUE) {
58. android\_atomic\_add(-INITIAL\_STRONG\_VALUE, &impl->mStrong);
59. impl->mBase->onFirstRef();
60. }
62. **return** **true**;
63. }

      这个函数的作用是试图增加目标对象的强引用计数，但是有可能会失败，失败的原因可能是因为目标对象已经被delete掉了，或者是其它的原因，下面会分析到。前面我们在讨论强指针的时候说到，增加目标对象的强引用计数的同时，也会增加目标对象的弱引用计数，因此，函数在开始的地方首先就是调用incWeak函数来先增加目标对象的引用计数，如果后面试图增加目标对象的强引用计数失败时，会调用decWeak函数来回滚前面的incWeak操作。

      这里试图增加目标对象的强引用计数时，分两种情况讨论，一种情况是此时目标对象正在被其它强指针引用，即它的强引用计数大于0，并且不等于INITIAL\_STRONG\_VALUE，另一种情况是此时目标对象没有被任何强指针引用，即它的强引用计数小于等于0，或者等于INITIAL\_STRONG\_VALUE。

      第一种情况比较简单，因为这时候说明目标对象一定存在，因此，是可以将这个弱指针提升为强指针的，在这种情况下，只要简单地增加目标对象的强引用计数值就行了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **while** (curCount > 0 && curCount != INITIAL\_STRONG\_VALUE) {
2. **if** (android\_atomic\_cmpxchg(curCount, curCount+1, &impl->mStrong) == 0) {
3. **break**;
4. }
5. curCount = impl->mStrong;
6. }

       当我们在这里对目标对象的强引用计数执行加1操作时，要保证原子性，因为其它地方也有可能正在对这个目标对象的强引用计数执行加1的操作，前面我们一般是调用android\_atomic\_inc函数来完成，但是这里是通过调用android\_atomic\_cmpxchg函数来完成，android\_atomic\_cmpxchg函数是体系结构相关的函数，在提供了一些特殊的指令的体系结构上，调用android\_atomic\_cmpxchg函数来执行加1操作的效率会比调用android\_atomic\_inc函数更高一些。函数android\_atomic\_cmpxchg是在system/core/include/cutils/atomic.h文件中定义的一个宏：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **int** android\_atomic\_release\_cas(int32\_t oldvalue, int32\_t newvalue,
2. **volatile** int32\_t\* addr);
4. #define android\_atomic\_cmpxchg android\_atomic\_release\_cas

      它实际执行的函数是android\_atomic\_release\_cas，这个函数的工作原理大概是这样的：如果它发现\*addr == oldvalue，就会执行\*addr = newvalue的操作，然后返回0，否则什么也不做，返回1。在我们讨论的这个场景中，oldvalue等于curCount，而newvalue等于curCount + 1，于是，在\*addr == oldvalue的条件下，就相当于是对目标对象的强引用计数值增加了1。什么情况下\*addr != oldvalue呢？在调用android\_atomic\_release\_cas函数之前，oldvalue和值就是从地址addr读出来的，如果在执行android\_atomic\_release\_cas函数的时候，有其它地方也对地址addr进行操作，那么就会有可能出现\*addr != oldvalue的情况，这时候就说明其它地方也在操作目标对象的强引用计数了，因此，这里就不能执行增加目标对象的强引用计数的操作了，它必须要等到其它地方操作完目标对象的强引用计数之后再重新执行，这就是为什么要通过一个while循环来执行了。

      第二种情况比较复杂一点，因为这时候目标对象可能还存在，也可能不存了，这要根据实际情况来判断。如果此时目标对象的强引用计数值等于INITIAL\_STRONG\_VALUE，说明此目标对象还从未被强指针引用过，这时候弱指针能够被提升为强指针的条件就为：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. allow = (impl->mFlags&OBJECT\_LIFETIME\_WEAK) != OBJECT\_LIFETIME\_WEAK
2. || impl->mBase->onIncStrongAttempted(FIRST\_INC\_STRONG, id);

      即如果目标对象的生命周期只受到强引用计数控制或者在目标对象的具体实现中总是允许这种情况发生。怎么理解呢？如果目标对象的生命周期只受强引用计数控制（它的标志位mFlags为0），而这时目标对象又还未被强指针引用过，它自然就不会被delete掉，因此，这时候可以判断出目标对象是存在的；如果目标对象的生命周期受弱引用计数控制（OBJECT\_LIFETIME\_WEAK），这时候由于目标对象正在被弱指针引用，因此，弱引用计数一定不为0，目标对象一定存在；如果目标对象的生命周期不受引用计数控制（BJECT\_LIFETIME\_FOREVER），这时候目标对象也是下在被弱指针引用，因此，目标对象的所有者必须保证这个目标对象还没有被delete掉，否则就会出问题了。在后面两种场景下，因为目标对象的生命周期都是不受强引用计数控制的，而现在又要把弱指针提升为强指针，就需要进一步调用目标对象的onIncStrongAttempted来看看是否允许这种情况发生，这又该怎么理解呢？可以这样理解，目标对象的设计者可能本身就不希望这个对象被强指针引用，只能通过弱指针来引用它，因此，这里它就可以重载其父类的onIncStrongAttempted函数，然后返回false，这样就可以阻止弱指针都被提升为强指针。在RefBase类中，其成员函数onIncStrongAttempted默认是返回true的：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **bool** RefBase::onIncStrongAttempted(uint32\_t flags, **const** **void**\* id)
2. {
3. **return** (flags&FIRST\_INC\_STRONG) ? **true** : **false**;
4. }

      如果此时目标对象的强引用计数值小于等于0，那就说明该对象之前一定被强指针引用过，这时候就必须保证目标对象是被弱引用计数控制的（BJECT\_LIFETIME\_WEAK），否则的话，目标对象就已经被delete了。同样，这里也要调用一下目标对象的onIncStrongAttempted成员函数，来询问一下目标对象在强引用计数值小于等于0的时候，是否允计将弱指针提升为强指针。下面这个代码段就是执行上面所说的逻辑：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. allow = (impl->mFlags&OBJECT\_LIFETIME\_WEAK) == OBJECT\_LIFETIME\_WEAK
2. && impl->mBase->onIncStrongAttempted(FIRST\_INC\_STRONG, id);

     继续往下看：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **if** (!allow) {
2. decWeak(id);
3. **return** **false**;
4. }
5. curCount = android\_atomic\_inc(&impl->mStrong);

      如果allow值为false，那么就说明不允计把这个弱指针提升为强指针，因此就返回false了，在返回之前，要先调用decWeak函数来减少目标对象的弱引用计数，因为函数的开头不管三七二十一，首先就调用了incWeak来增加目标对象的弱引用计数值。

      函数attemptIncStrong的主体逻辑大概就是这样了，比较复杂，读者要细细体会一下。函数的最后，如果此弱指针是允计提升为强指针的，并且此目标对象是第一次被强指针引用，还需要调整一下目标对象的强引用计数值：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. **if** (curCount == INITIAL\_STRONG\_VALUE) {
2. android\_atomic\_add(-INITIAL\_STRONG\_VALUE, &impl->mStrong);
3. impl->mBase->onFirstRef();
4. }

      这个逻辑我们在前面分析强指针时已经分析过了，这里不再详述。

     分析到这里，弱指针就介绍完了。强指针和弱指针的关系比较密切，同时它们也比较复杂，下面我们再举一个例子来说明强指针和弱指针的用法，同时也验证一下它们的实现原理。

## 3.5 强指针和弱指针的用法

      参考[在Ubuntu上为Android系统内置C可执行程序测试Linux内核驱动程序](http://blog.csdn.net/luoshengyang/article/details/6571210)一文，我们在external目录下建立一个C++工程目录weightpointer，它里面有两个文件，一个weightpointer.cpp文件，另外一个是Android.mk文件。

      源文件weightpointer.cpp的内容如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. #include <stdio.h>
2. #include <utils/RefBase.h>
4. #define INITIAL\_STRONG\_VALUE (1<<28)
6. **using** **namespace** android;
8. **class** WeightClass : **public** RefBase
9. {
10. **public**:
11. **void** printRefCount()
12. {
13. int32\_t strong = getStrongCount();
14. weakref\_type\* ref = getWeakRefs();
16. printf("-----------------------\n");
17. printf("Strong Ref Count: %d.\n", (strong  == INITIAL\_STRONG\_VALUE ? 0 : strong));
18. printf("Weak Ref Count: %d.\n", ref->getWeakCount());
19. printf("-----------------------\n");
20. }
21. };
23. **class** StrongClass : **public** WeightClass
24. {
25. **public**:
26. StrongClass()
27. {
28. printf("Construct StrongClass Object.\n");
29. }
31. **virtual** ~StrongClass()
32. {
33. printf("Destory StrongClass Object.\n");
34. }
35. };

38. **class** WeakClass : **public** WeightClass
39. {
40. **public**:
41. WeakClass()
42. {
43. extendObjectLifetime(OBJECT\_LIFETIME\_WEAK);
45. printf("Construct WeakClass Object.\n");
46. }
48. **virtual** ~WeakClass()
49. {
50. printf("Destory WeakClass Object.\n");
51. }
52. };
54. **class** ForeverClass : **public** WeightClass
55. {
56. **public**:
57. ForeverClass()
58. {
59. extendObjectLifetime(OBJECT\_LIFETIME\_FOREVER);
61. printf("Construct ForeverClass Object.\n");
62. }
64. **virtual** ~ForeverClass()
65. {
66. printf("Destory ForeverClass Object.\n");
67. }
68. };

71. **void** TestStrongClass(StrongClass\* pStrongClass)
72. {
73. wp<StrongClass> wpOut = pStrongClass;
74. pStrongClass->printRefCount();
76. {
77. sp<StrongClass> spInner = pStrongClass;
78. pStrongClass->printRefCount();
79. }
81. sp<StrongClass> spOut = wpOut.promote();
82. printf("spOut: %p.\n", spOut.get());
83. }
85. **void** TestWeakClass(WeakClass\* pWeakClass)
86. {
87. wp<WeakClass> wpOut = pWeakClass;
88. pWeakClass->printRefCount();
90. {
91. sp<WeakClass> spInner = pWeakClass;
92. pWeakClass->printRefCount();
93. }
95. pWeakClass->printRefCount();
96. sp<WeakClass> spOut = wpOut.promote();
97. printf("spOut: %p.\n", spOut.get());
98. }

101. **void** TestForeverClass(ForeverClass\* pForeverClass)
102. {
103. wp<ForeverClass> wpOut = pForeverClass;
104. pForeverClass->printRefCount();
106. {
107. sp<ForeverClass> spInner = pForeverClass;
108. pForeverClass->printRefCount();
109. }
110. }
112. **int** main(**int** argc, **char**\*\* argv)
113. {
114. printf("Test Strong Class: \n");
115. StrongClass\* pStrongClass = **new** StrongClass();
116. TestStrongClass(pStrongClass);
118. printf("\nTest Weak Class: \n");
119. WeakClass\* pWeakClass = **new** WeakClass();
120. TestWeakClass(pWeakClass);
122. printf("\nTest Froever Class: \n");
123. ForeverClass\* pForeverClass = **new** ForeverClass();
124. TestForeverClass(pForeverClass);
125. pForeverClass->printRefCount();
126. **delete** pForeverClass;
128. **return** 0;
129. }

     首先定义了一个基类WeightClass，继承于RefBase类，它只有一个成员函数printRefCount，作用是用来输出引用计数。接着分别定义了三个类StrongClass、WeakClass和ForeverClass，其中实例化StrongClass类的得到的对象的标志位为默认值0，实例化WeakClass类的得到的对象的标志位为OBJECT\_LIFETIME\_WEAK，实例化ForeverClass类的得到的对象的标志位为OBJECT\_LIFETIME\_FOREVER，后两者都是通过调用RefBase类的extendObjectLifetime成员函数来设置的。

     在main函数里面，分别实例化了这三个类的对象出来，然后分别传给TestStrongClass函数、TestWeakClass函数和TestForeverClass函数来说明智能指针的用法，我们主要是通过考察它们的强引用计数和弱引用计数来验证智能指针的实现原理。

      编译脚本文件Android.mk的内容如下：

**[plain]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. LOCAL\_PATH := $(call my-dir)
2. include $(CLEAR\_VARS)
3. LOCAL\_MODULE\_TAGS := optional
4. LOCAL\_MODULE := weightpointer
5. LOCAL\_SRC\_FILES := weightpointer.cpp
6. LOCAL\_SHARED\_LIBRARIES := \
7. libcutils \
8. libutils
9. include $(BUILD\_EXECUTABLE)

      最后，我们参照[如何单独编译Android源代码中的模块](http://blog.csdn.net/luoshengyang/article/details/6566662)一文，使用mmm命令对工程进行编译：

**[plain]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. USER-NAME@MACHINE-NAME:~/Android$ mmm ./external/weightpointer

      编译之后，就可以打包了：

**[plain]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. USER-NAME@MACHINE-NAME:~/Android$ make snod

      最后得到可执行程序weightpointer就位于设备上的/system/bin/目录下。启动模拟器，通过adb shell命令进入到模拟器终端，进入到/system/bin/目录，执行weightpointer可执行程序，验证程序是否按照我们设计的逻辑运行：

**[plain]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. USER-NAME@MACHINE-NAME:~/Android$ adb shell
2. root@android:/ # cd system/bin/
3. root@android:/system/bin # ./weightpointer

      执行TestStrongClass函数的输出为：

**[plain]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. Test Strong Class:
2. Construct StrongClass Object.
3. -----------------------
4. Strong Ref Count: 0.
5. Weak Ref Count: 1.
6. -----------------------
7. -----------------------
8. Strong Ref Count: 1.
9. Weak Ref Count: 2.
10. -----------------------
11. Destory StrongClass Object.
12. spOut: 0x0.

      在TestStrongClass函数里面，首先定义一个弱批针wpOut指向从main函数传进来的StrongClass对象，这时候我们可以看到StrongClass对象的强引用计数和弱引用计数值分别为0和1；接着在一个大括号里面定义一个强指针spInner指向这个StrongClass对象，这时候我们可以看到StrongClass对象的强引用计数和弱引用计数值分别为1和2；当程序跳出了大括号之后，强指针spInner就被析构了，从上面的分析我们知道，强指针spInner析构时，会减少目标对象的强引用计数值，因为前面得到的强引用计数值为1，这里减1后，就变为0了，又由于这个StrongClass对象的生命周期只受强引用计数控制，因此，这个StrongClass对象就被delete了，这一点可以从后面的输出（“Destory StrongClass Object.”）以及试图把弱指针wpOut提升为强指针时得到的对象指针为0x0得到验证。

      执行TestWeakClass函数的输出为：

**[plain]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. Test Weak Class:
2. Construct WeakClass Object.
3. -----------------------
4. Strong Ref Count: 0.
5. Weak Ref Count: 1.
6. -----------------------
7. -----------------------
8. Strong Ref Count: 1.
9. Weak Ref Count: 2.
10. -----------------------
11. -----------------------
12. Strong Ref Count: 0.
13. Weak Ref Count: 1.
14. -----------------------
15. spOut: 0xa528.
16. Destory WeakClass Object.

      TestWeakClass函数和TestStrongClass函数的执行过程基本一样，所不同的是当程序跳出大括号之后，虽然这个WeakClass对象的强引用计数值已经为0，但是由于它的生命周期同时受强引用计数和弱引用计数控制，而这时它的弱引用计数值大于0，因此，这个WeakClass对象不会被delete掉，这一点可以从后面试图把弱批针wpOut提升为强指针时得到的对象指针不为0得到验证。

      执行TestForeverClass函数的输出来：

**[plain]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. Test Froever Class:
2. Construct ForeverClass Object.
3. -----------------------
4. Strong Ref Count: 0.
5. Weak Ref Count: 1.
6. -----------------------
7. -----------------------
8. Strong Ref Count: 1.
9. Weak Ref Count: 2.
10. -----------------------

     当执行完TestForeverClass函数返回到main函数的输出来：

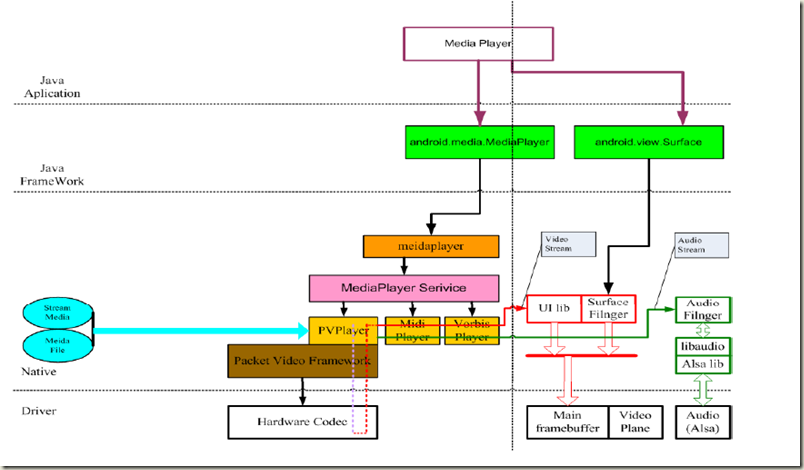
**[plain]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6786239)[在CODE上查看代码片](https://code.csdn.net/snippets/130109)

1. -----------------------
2. Strong Ref Count: 0.
3. Weak Ref Count: 0.
4. -----------------------
5. Destory ForeverClass Object.

      这里我们可以看出，虽然这个ForeverClass对象的强引用计数和弱引用计数值均为0了，但是它不自动被delete掉，虽然由我们手动地delete这个对象，它才会被析构，这是因为这个ForeverClass对象的生命周期是既不受强引用计数值控制，也不会弱引用计数值控制。

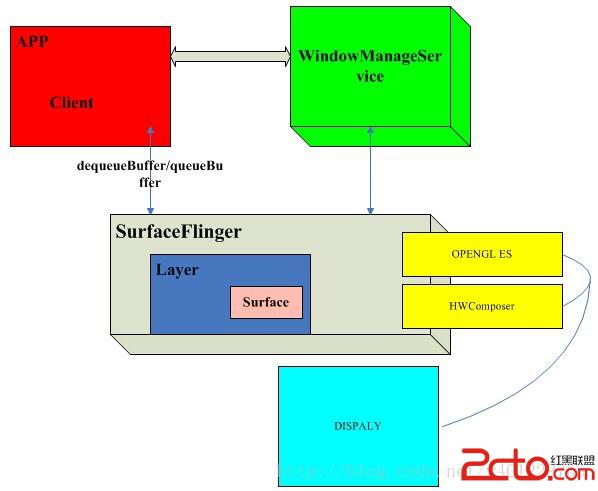
      这样，从TestStrongClass、TestWeakClass和TestForeverClass这三个函数的输出就可以验证了我们上面对Android系统的强指针和弱指针的实现原理的分析。

# 4 MediaPlayer框架



## 4.1 MediaPlayer Service

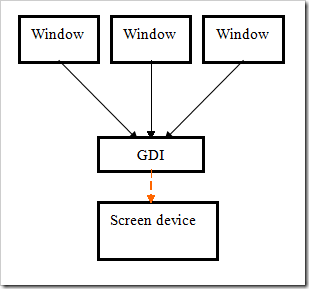
## 4.2 MediaPlayer Surface



### 4.2.1 Android GDI基本框架

在Android中所涉及的概念和代码最多，最繁杂的就是GDI相关的代码了。但是本质从抽象上来讲，这么多的代码和框架就干了一件事情：对显示缓冲区的操作和管理。

    GDI主要管理图形图像的输出,从整体方向上来看,GDI可以被认为是一个物理屏幕使用的管理器。因为在实际的产品中，我们需要在物理屏幕上输出不同的窗口，而每个窗口认为自己独占屏幕的使用，对所有窗口输出，应用程序不会关心物理屏幕是否被别的窗口占用，而只是关心自己在本窗口的输出，至于输出是否能在屏幕上看见，则需要GDI来管理。

[](http://hi.csdn.net/attachment/201006/13/0_127644050310jf.gif)

    从最上层到最底层的数据流的分析可以看到实际上GDI在上层为GUI提供一个抽象的概念,就好像操作系统中的文件系统所提供文件，目录等抽象概念一样，GDI输出抽象成了文本,画笔,位图操作等设备无关的操作,让应用程序员只需要面对逻辑的设备上下文进行输出操作,而不要涉及到具体输出设备,以及输出边界的管理。GDI负责将文本、线条、位图等概念对象映射到具体的物理设备，所以GDI的在大体方向上可以分为以下几大要素：

* 画布
* 字体
* 文本输出
* 绘画对象
* 位图输出

Android的GDI系统

     Android的GDI系统所涉及到概念太多，加之使用了OpenGL使得Android的层次和代码很繁杂。但是我们对于Android的GDI系统需要了解的方面不是他的静态的代码关系，而是动态的对象关系，在逻辑运行的架构上理解GDI。我们首先还是需要从代码结构开始我们的理解。

Frameworks/Libs/Surfaceflinger

Frameworks/base/core/jni/android\_view\_Surface.cpp

Frameworks/base/core/java/android/view/surface.java

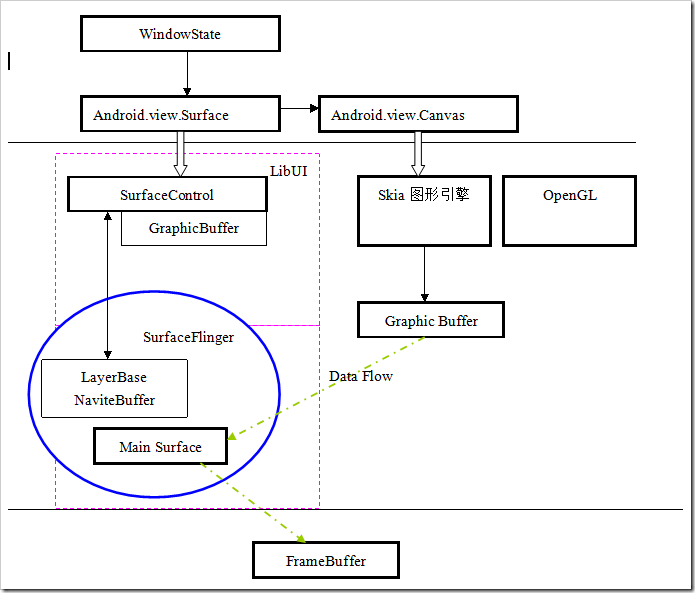
Frameworks/base/Graphics：绘图接口

Frameworks/Libs/Ui

External/Skia

其中External/Skia是一个C++的2D图形引擎库，Android的2D绘制系统都是建立在该基础之上.Skia完成了：文本输出，位图，点，线，图像解码等功能。

我在这里给出Android GDI的基本框架示意图。

[](http://hi.csdn.net/attachment/201006/13/0_1276440506aSNE.gif)

  对于上面的GDI架构图我们只是一个大概的了解，我们有太多的问题需要解决，有太多的疑问需要得到答案，我就一直在想，为什么设计者有提出如此众多的概念，这个概念的背景是什么？他要管理什么，他要抽象什么？从前面知道，Android的整个设计理念就是无边界化，他是如何穿透Linux进程这个鸿沟来达到无边界的？Surface，Canvas， Layer，LayerBase, NativeBuffer，SurfaceFlinger，SurfaceFlingerClient这些到底是一个什么东西？如何管理，传递的是什么？创建的是什么？这些都是抽象的概念，绘画的终极的缓冲区到底是如何管理的？缓冲区到底在哪里？

     我们还是看看做终极的，最本质的设计概念，在从这些概念出发，来探讨这些概念的形成过程，是否有必要去生成写概念。SurfaceFlinger本质上干什么的？SurfaceFlinger的确就是这个意义：应用程序通过SurfaceFlinger将自己的“Surface”投掷到屏幕缓冲区。至于如何投掷的，我们将会在后面详细描述。

### 4.2.2 Android GDI之显示缓冲管理

万丈高楼从地起，从最根源的硬件帧缓冲区开始。我们知道显示FrameBuffer在系统中就是一段内存,GDI的工作就是把需要输出的内容放入到该段内存的某个位置。我们从基本的点（像素点）和基本的缓冲区操作开始。

#### 4.2.2.1 基本知识

##### 4.2.2.1.1点的格式

对于不同的LCD来讲，FrameBuffer的二进制格式不一样，并且可以分为两部分：

   1)点的格式：通常将Depth,即表示多少位表示一个点。

1位表示一个点

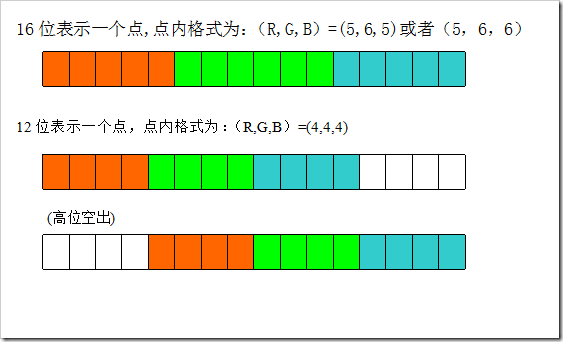
2位表示一个点

16位表示一个点

32位表示一个点（Alpha通道）

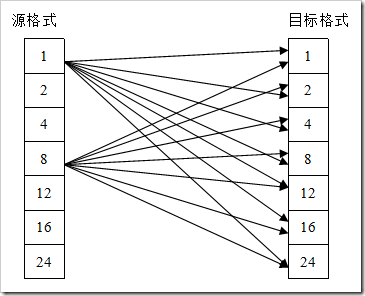
      2) 点内格式：RGB分量分布表示。

例如对于我们常见的16位表示一个点

[](http://hi.csdn.net/attachment/201006/14/0_127649370468hY.gif)

##### 4.2.2.1.2格式之间的转换

所以屏幕输出实际上是一个值映射的关系。我们可以有如下的点格式转换，

[](http://hi.csdn.net/attachment/201006/14/0_1276493706k5xC.gif)

源格式可能来自单色位图和彩色位图，对于具体的目标机来讲，我们的目标格式可能就是一种，例如16位（5/6/5）格式。其实就只存在一种格式的转换，即从目标格式都是16位格式。

[](http://hi.csdn.net/attachment/201006/14/0_1276493707JNOB.gif)

但是，在设计GDI时，基本要求有一个可移植性好，所以我们还是必须考虑对于不同点格式LCD之间的转换操作。所以在GDI的驱动程序中涉及到如下几类主要操作:

区域操作(Blit)：我们在显示缓冲区上做的最多的操作就是区块搬运。由此，很多的应用处理器使用了硬件图形加速器来完成区域搬运:blit.从我们的主要操作的对象来看,可以分为两个方向:

1)内存区域到屏幕区域

2)屏幕区域到屏幕区域

3)屏幕区域到内存区域

4)内存区域到内存区域

在这里我们需要特别提出的是，由于在Linux不同进程之间的内存不能自由的访问，使得我们的每个Android应用对于内存区域和屏幕缓冲区的使用变得很复杂。在Android的设计中，在屏幕缓冲区和显示内存缓冲区的管理分类很多的层次，最上层的对象是可以在进程间自由传递，但是对于缓冲区内容则使用共享内存的机制。

基于以上的基础知识，我们可以知道：

（1）代码中Config及其Format的意义所在了。也就理解了兼容性的意义：采用同硬件相同的点的描述对象

（2）所有屏幕上图形的移动都是显示缓冲区搬运的结果。

##### 4.2.2.1.3图形加速器

应用处理器都可能带有图形加速器，对于不同的应用处理器对其图形加速器可能有不同的处理方式，对于2D加速来讲，都可归结为Blit。多为数据的搬运，放大缩小，旋转等。

#### 4.2.2.2 Android的缓冲区抽象定义

  不同的硬件有不同的硬件图形加速设备和缓冲内存实现方法。Android Gralloc动态库抽象的任务就是消除不同的设备之间的差别，在上层看来都是同样的方法和对象。在Moudle层隐藏缓冲区操作细节。Android使用了动态链接库gralloc.xxx.so，来完成底层细节的封装。

##### 4.2.2.2.1 本地定义@hardware/libhandware/modules/gralloc

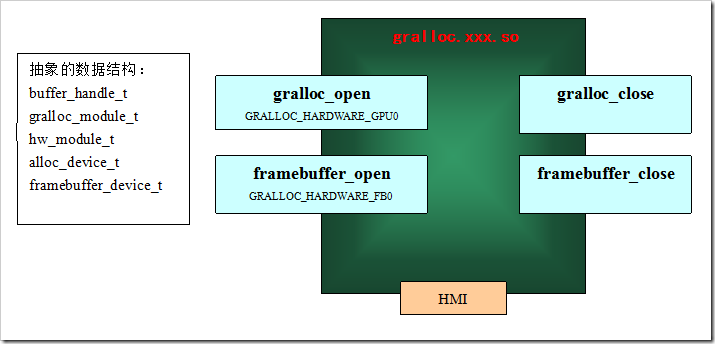
每个动态链接库都是用相同名称的调用接口：

1)硬件图形加速器的抽象：BlitEngine，CopyBit的加速操作。

2)硬件FrameBuffer内存管理

3)共享缓存管理

从数据关系上我们来考察..动态链接库的抽象行为：在层次：[Hardware.c@hardware/libhardware](mailto:Hardware.c@hardware/libhardware) 中对动态链接库中的内容作了全新的包装。/system/lib/hw/gralloc.xxx.so动态库文件。从文件Gralloc.h(handware/libhardware/include/hardware)是抽象的结果：hw\_get\_module从gralloc.xxx.so提取了HAL\_MODULE\_INFO\_SYM（SYM变量）

[](http://hi.csdn.net/attachment/201006/14/0_1276493710BnSi.gif)

从展露在外部的数据结构，[我们在@Gralloc.cpp](mailto:%E6%88%91%E4%BB%AC%E5%9C%A8@Gralloc.cpp)看到到了这样的布局：

static struct hw\_module\_methods\_t gralloc\_module\_methods = {

open: gralloc\_device\_open

};

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

    base: {

        common: {

            tag: HARDWARE\_MODULE\_TAG,

             …

            id: GRALLOC\_HARDWARE\_MODULE\_ID,

            name: "Graphics Memory Allocator Module",

            author: "The Android Open Source Project",

            methods: &gralloc\_module\_methods

        },

registerBuffer: gralloc\_register\_buffer,

unregisterBuffer: gralloc\_unregister\_buffer,

lock:  gralloc\_lock,

unlock:  gralloc\_unlock,

    },

    framebuffer: 0,

    flags: 0,

    numBuffers: 0,

    bufferMask: 0,

…

};

我们建立了什么对象来支撑缓冲区的操作？

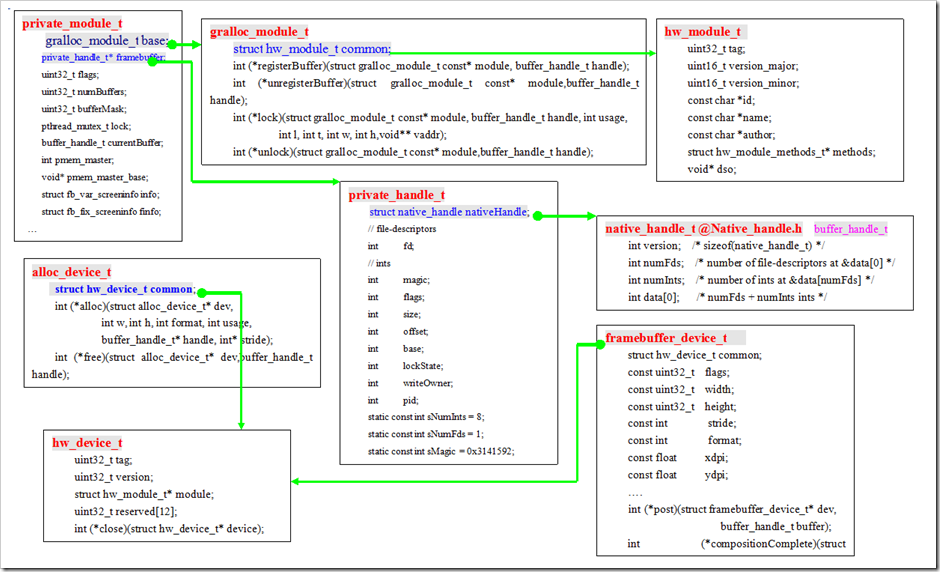
buffer\_handle\_t：外部接口。

methods.open，registerBuffer，unregisterBuffer，lock，unlock

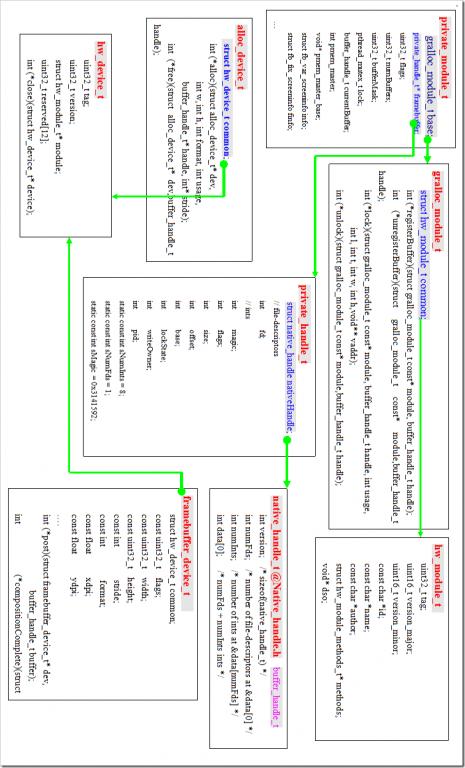
下面是外部接口和内部对象的结构关系，该类型的结构充分利用C Struct的数据排列特性：基本结构体放置在最前面，本地私有放置在后面，满足了抽象的需要。

typedef const native\_handle\* buffer\_handle\_t;

private\_module\_t  HAL\_MODULE\_INFO\_SYM 向往暴露的动态链接库接口，通过该接口，我们直接可以使用该对象。

[](http://hi.csdn.net/attachment/201006/14/0_1276493716565n.gif)

看不清楚上面图，可以偏一下头横着看：

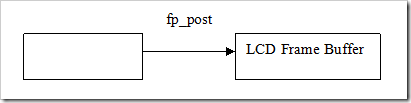
[](http://hi.csdn.net/attachment/201006/14/0_1276493727LN9N.gif)

几个接口函数的解释：

(1)fb\_post

对于帧缓冲区实际地址并不需要向上层报告，所有的操作都是通过fb\_post了完成。

fp\_post的任务就是将一个Buffer的内容传递到硬件缓冲区。其实现方式有两种：



（方式1）无需拷贝动作，是把Framebuffer的后buffer切为前buffer，然后通过IOCTRL机制告诉FB驱动切换DMA源地地址。这个实现方式的前提是Linux内核必须分配至少两个缓冲区大小的物理内存和实现切换的ioctrol，这个实现快速切换。

（方式2）利用Copy的方式。不修改内核，则在适配层利用从拷贝的方式进行，但是这个是费时了。

(2)gralloc的主要功能是要完成：

     1）打开屏幕设备 "/dev/fb0",，并映射硬件显示缓冲区。

     2）提供分配共享显示缓存的接口

     3）提供BiltEngine接口（完成硬件加速器的包装）

（3）gralloc\_alloc输出buffer\_handle\_t句柄。

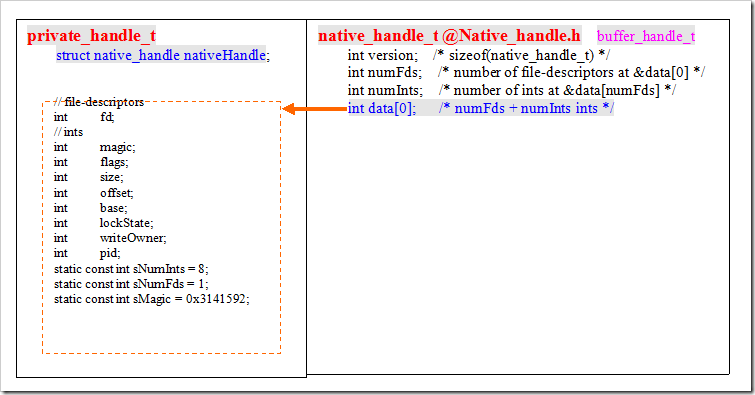
     这个句柄是共享的基本依据，其基本原理在后面的章节有详细描述。

总结一下，/system/lib/hw/gralloc.xxx.so是跟硬件体系相关的一个动态链接库，也可以叫做Android的硬件抽象层。他实现了Android的硬件抽象接口标准，提供显示内存的分配机制和CopyBit等的加速实现。而如何具体实现这些功能，则跟硬件平台的配备有关系，所以我们看到了对于与不同的硬件架构，有不同的配置关系。

### 4.2.3 Android GDI之共享缓冲区机制

#### 4.2.3.1 native\_handle\_t对private\_handle\_t 的包裹

private\_handle\_t是gralloc.so使用的本地缓冲区私有的数据结构，而Native\_handle\_t是上层抽象的可以在进程间传递的数据结构。在客户端是如何还原所传递的数据结构呢？首先看看native\_handle\_t对private\_handle\_t的抽象包装。

[](http://hi.csdn.net/attachment/201006/14/0_1276504107ekdp.gif)

numFds= sNumFds=1;

numInts= sNumInts=8;

这个是Parcel中描述句柄的抽象模式。实际上是指的Native\_handle所指向句柄对象的具体内容：

numFds=1表示有一个文件句柄：fd/

numInts= 8表示后面跟了8个INT型的数据：magic,flags,size,offset,base,lockState,writeOwner,pid;

由于在上层系统不要关心buffer\_handle\_t中data的具体内容。在进程间传递buffer\_handle\_t(native\_handle\_t)句柄是其实是将这个句柄内容传递到Client端。在客户端通过Binder读取readNativeHandle @Parcel.cpp新生成一个native\_handle。

native\_handle\* Parcel::readNativeHandle() const

{

…

native\_handle\* h = native\_handle\_create(numFds, numInts);

    for (int i=0 ; err==NO\_ERROR && i<="" font="">

        h->data[i] = dup(readFileDescriptor());

        if (h->data[i] < 0) err = BAD\_VALUE;

    }

    err = read(h->data + numFds, sizeof(int)\*numInts);

    ….

return h;

}

这里需要提到的是为在构造客户端的native\_handle时，对于对方传递过来的文件句柄的处理。由于不是在同一个进程中，所以需要dup(…)一下为客户端使用。这样就将Native\_handle句柄中的，客户端感兴趣的从服务端复制过来。这样就将Private\_native\_t的数据：magic,flags,size,offset,base,lockState,writeOwner,pid;复制到了客户端。

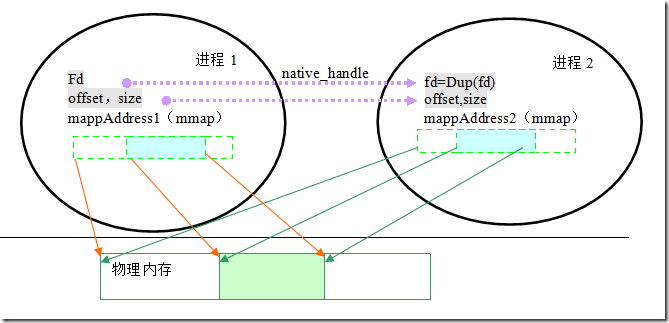
客户端利用这个新的Native\_buffer被Mapper传回到gralloc.xxx.so中，获取到native\_handle关联的共享缓冲区映射地址，从而获取到了该缓冲区的控制权，达到了客服端和Server间的内存共享。从SurfaceFlinger来看就是作图区域的共享。

#### 4.2.3.2 Graphic Mapper是干什么的？

服务端（SurfaceFlinger）分配了一段内存作为Surface的作图缓冲，客户端怎样在这个作图缓冲区上工作呢？这个就是Mapper(GraphicBufferMapper)y要干的事情。两个进程间如何共享内存，如何获取到共享内存？Mapper就是干这个得。需要利用到两个信息：共享缓冲区设备句柄，分配时的偏移量。Mapper利用这样的原理：

    客户端只有lock,unlock,实质上就是mmap和ummap的操作。对于同样一个共享缓冲区，偏移量才是总要的，起始地址不重要。实际上他们操作了同一物理地址的内存块。我们在上面讨论了native\_handle\_t对private\_handle\_t 的包裹过程，从中知道服务端给客户端传递了什么东西。

        进程1在共享内存设备上预分配了8M的内存。以后所有的分配都是在这个8M的空间进行。对以该文件设备来讲，8M物理内存提交后，就实实在在的占用了8M内存。每个每个进程都可以同个该内存设备共享该8M的内存，他们使用的工具就会mmap。由于在mmap都是用0开始获取映射地址，所以所有的客户端进程都是有了同一个物理其实地址，所以此时偏移量和size就可以标识一段内存。而这个偏移量和size是个数值，从服务进程传递到客户端直接就可以使用。

[](http://hi.csdn.net/attachment/201006/14/0_1276504121th7n.gif)

#### 4.2.3.3 Graphic Buffer（缓冲区代理对象）

typedef struct android\_native\_buffer\_t

{

    struct android\_native\_base\_t common;

    int width;

    int height;

    int stride;

    int format;

    int usage;

     …

**buffer\_handle\_t handle;**

     …

} android\_native\_buffer\_t;

关系图表：

GraphicBuffer :EGLNativeBase :android\_native\_buffer\_t

GraphicBuffer(parcel &)建立本地的GraphicBuffer的数据native\_buffer\_t。在通过接收对方的传递的native\_buffer\_t 构建GraphicBuffer。我们来看看在客户端Surface：：lock获取操作缓冲区的函数调用：

Surface::lock(SurfaceInfo\* other, Region\* dirtyIn, bool blocking)

     {int Surface::dequeueBuffer(android\_native\_buffer\_t\*\* buffer)（Surface）

        {status\_t Surface::getBufferLocked(int index, int usage)

          {

               sp buffer = s->requestBuffer(index, usage);

           {

virtual sp requestBuffer(int bufferIdx, int usage)

{   remote()->transact(REQUEST\_BUFFER, data, &reply);

   sp buffer = new GraphicBuffer(reply);

Surface：：Lock建立一个在Client端建立了一个新的GraphicBuffer 对象，该对象通过（1）描述的原理将SurfaceFlinger的buffer\_handle\_t相关数据构成新的客户端buffer\_handle\_t数据。在客户端的Surface对象就可以使用GraphicMapper对客户端buffer\_handle\_t进行mmap从而获取到共享缓冲区的开始地址了。

  Android在该节使用了共享内存的方式来管理与显示相关的缓冲区，他设计成了两层，上层是缓冲区管理的代理机构GraphicBuffer,及其相关的native\_buffer\_t,下层是具体的缓冲区的分配管理及其缓冲区本身。上层的对象是可以在经常间通过Binder传递的，而在进程间并不是传递缓冲区本身，而是使用mmap来获取指向共同物理内存的映射地址。

### 4.2.4 Android GDI[之SurfaceFlinger](http://blog.csdn.net/maxleng/article/details/5671271)

SurfaceFinger按英文翻译过来就是Surface投递者。SufaceFlinger的构成并不是太复杂，复杂的是他的客户端建构。SufaceFlinger主要功能是：

1） 将Layers （Surfaces） 内容的刷新到屏幕上

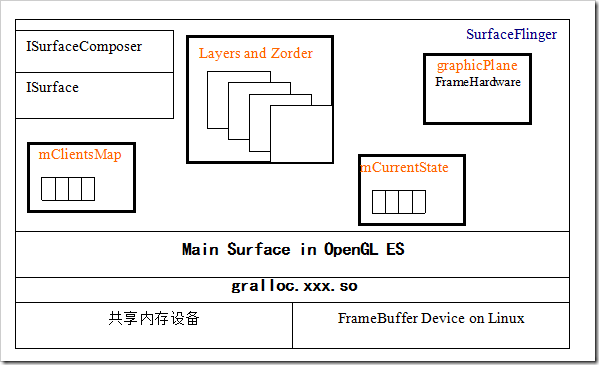
2） 维持Layer的Zorder序列，并对Layer最终输出做出裁剪计算。

3） 响应Client要求，创建Layer与客户端的Surface建立连接

4） 接收Client要求，修改Layer属性（输出大小，Alpha等设定）

但是作为投递者的实际意义，我们首先需要知道的是如何投递，投掷物，投递路线，投递目的地。

#### 4.2.4.1 SurfaceFlinger的基本组成框架

[[](http://hi.csdn.net/attachment/201006/14/0_1276518584xvSo.gif)](http://hi.csdn.net/attachment/201006/14/0_1276518584xvSo.gif) SurfaceFlinger管理对象为：

mClientsMap：管理客户端与服务端的连接。

ISurface，IsurfaceComposer：AIDL调用接口实例

mLayerMap：服务端的Surface的管理对象。

mCurrentState.layersSortedByZ ：以Surface的Z-order序列排列的Layer数组。

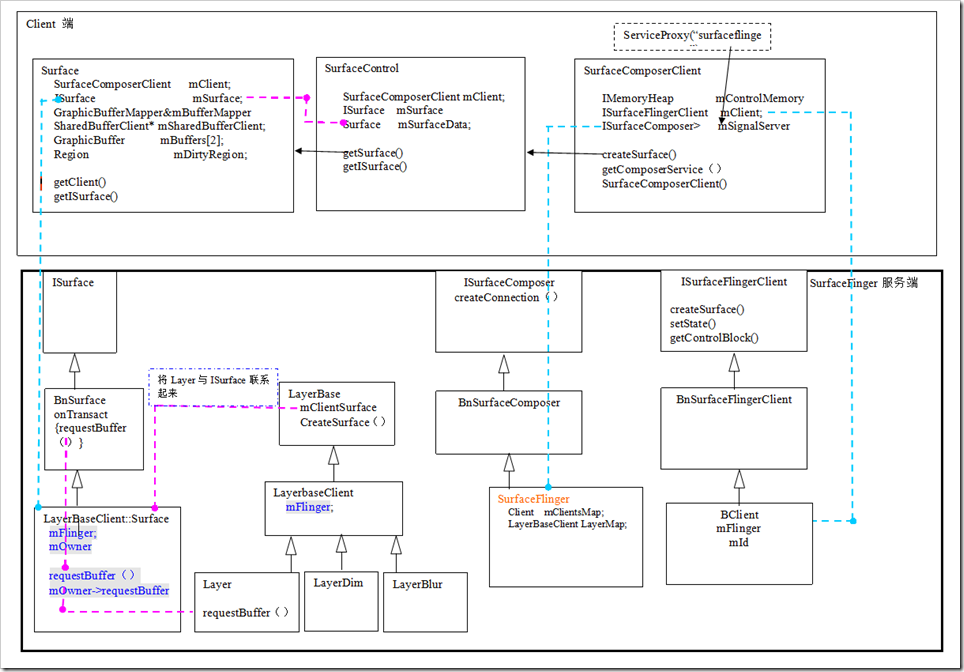
graphicPlane 缓冲区输出管理

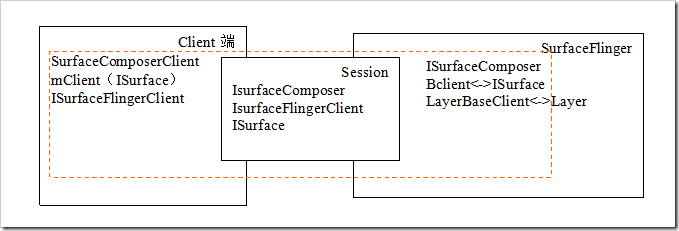
OpenGL ES：图形计算，图像合成等图形库。

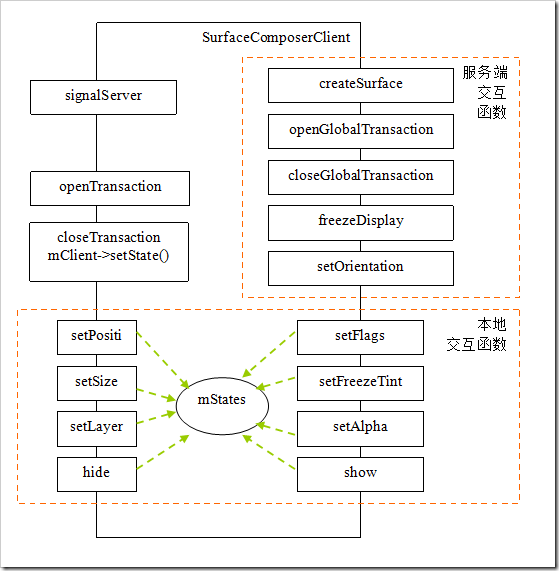
gralloc.xxx.so这是个跟平台相关的图形缓冲区管理器。

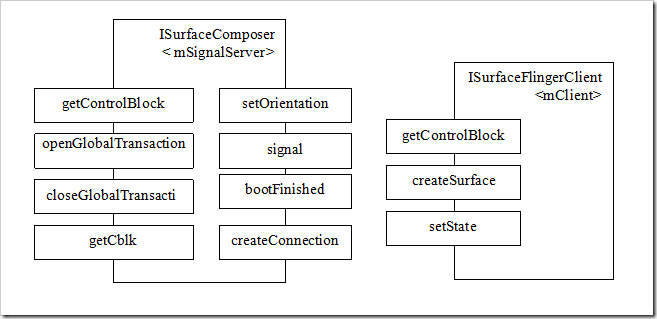
pmem Device：提供共享内存，在这里只是在gralloc.xxx.so可见，在上层被gralloc.xxx.so抽象了。

#### 4.2.4.2 Surface Flinger Client和服务端对象关系图

[[](http://hi.csdn.net/attachment/201006/14/0_12765192776tBE.gif)](http://hi.csdn.net/attachment/201006/14/0_12765192776tBE.gif) Client端与Surface Flinger连接图：

[](http://hi.csdn.net/attachment/201006/14/0_1276518598A6OZ.gif) Client对象：一般的在客户端都是通过SurfaceComposerClient来跟SurfaceFlinger打交道。

[](http://hi.csdn.net/attachment/201006/14/0_1276518601fTft.gif)

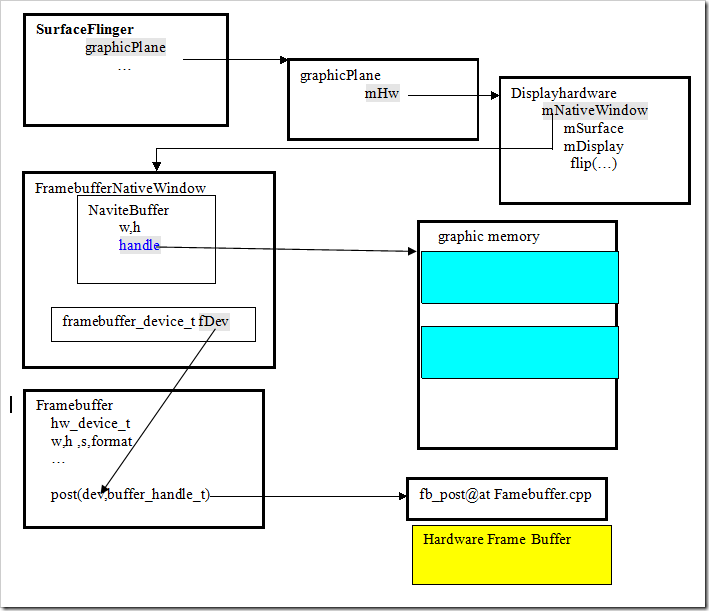
[](http://hi.csdn.net/attachment/201006/14/0_1276518603bAHB.gif)

#### 4.2.4.3 主要对象说明

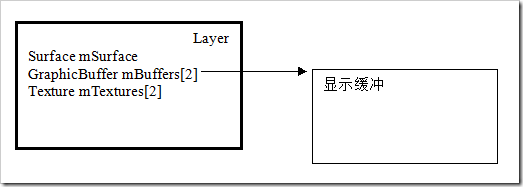
##### 4.2.4.3.1 DisplayHardware &FrameBuffer

首先SurfaceFlinger需要操作到屏幕，需要建立一个屏幕硬件缓冲区管理框架。Android在设计支持时，考虑多个屏幕的情况，引入了graphicPlane的概念。在SurfaceFlinger上有一个graphicPlane数组，每一个graphicPlane对象都对应一个DisplayHardware.在当前的Android（2.1）版本的设计中，系统支持一个graphicPlane，所以也就支持一个DisplayHardware。

SurfaceFlinger，Hardware硬件缓冲区的数据结构关系图。

[](http://hi.csdn.net/attachment/201006/14/0_1276518606owJM.gif)

##### 4.2.4.3.2 Layer

[](http://hi.csdn.net/attachment/201006/14/0_1276518608eft1.gif)

method:setBuffer  在SurfaceFlinger端建立显示缓冲区。这里的缓冲区是指的HW性质的，PMEM设备文件映射的内存。

1) layer的绘制

void Layer::onDraw(const Region& clip) const

{

    int index = mFrontBufferIndex;

    GLuint textureName = mTextures[index].name;

…

  drawWithOpenGL(clip, mTextures[index]);

}

##### 4.2.4.3.3 mCurrentState.layersSortedByZ

以Surface的Z-order序列排列的LayerBase数组，该数组是层显示遮挡的依据。在每个层计算自己的可见区域时，从Z-order 顶层开始计算，是考虑到遮挡区域的裁减，自己之前层的可见区域就是自己的不可见区域。而绘制Layer时，则从Z-order底层开始绘制，这个考虑到透明层的叠加。

#### 4.2.4.4 Surface Flinger的运行框架

我们从前面的章节<Android Service>的基本原理可以知道，SurfaceFlinger的运行框架存在于：threadLoop,他是SurfaceFlinger的主循环体。SurfaceFlinger在进入主体循环之前会首先运行：SurfaceFlinger::readyToRun()。

##### 4.2.4.4.1 SurfaceFlinger::readyToRun()

（1）建立GraphicPanle

（2）建立FrameBufferHardware(确定输出目标)

      初始化：OpenGL ES

       建立兼容的mainSurface.利用eglCreateWindowSurface。

        建立OpenGL ES进程上下文。

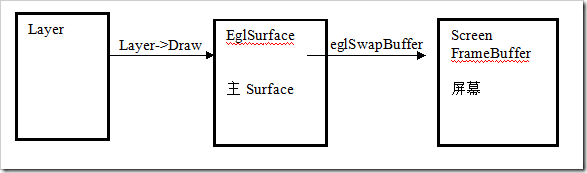
   建立主Surface（OpenGL ES）。 DisplayHardware的Init()@DisplayHardware.cpp函数对OpenGL做了初始化，并创建立主Surface。为什么叫主Surface，因为所有的Layer在绘制时，都需要先绘制在这个主Surface上，最后系统才将主Surface的内容”投掷”到真正的屏幕上。

（3） 主Surface的绑定

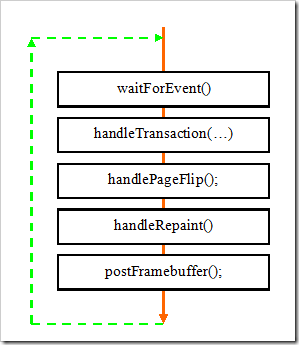
1）在DisplayHandware初始完毕后，hw.makeCurrent()将主Surface，OpenGL ES进程上下文绑定到SurfaceFlinger的上下文中，

2）之后所有的SurfaceFlinger进程中使用EGL的所有的操作目的地都是[mSurface@DisplayHardware](mailto:mSurface@DisplayHardware)。

这样，在OpenGL绘制图形时，主Surface被记录在进程的上下文中，所以看不到显示的主Surfce相关参数的传递。下面是Layer-Draw，Hardware.flip的动作示意图：

[](http://hi.csdn.net/attachment/201006/14/0_1276518609tWF6.gif)

##### 4.2.4.4.2 ThreadLoop

[[](http://hi.csdn.net/attachment/201006/14/0_12765186106SH3.gif)](http://hi.csdn.net/attachment/201006/14/0_12765186106SH3.gif)

(1)handleTransaction(…):主要计算每个Layer有无属性修改，如果有修改着内用需要重画。

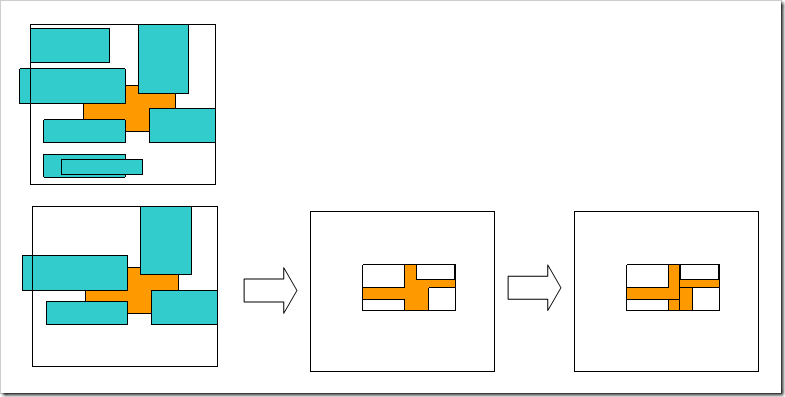
(2)handlePageFlip()

    computeVisibleRegions：根据Z-Order序列计算每个Layer的可见区域和被覆盖区域。裁剪输出范围计算-

在生成裁剪区域的时候，根据Z\_order依次，每个Layer在计算自己在屏幕的可显示区域时，需要经历如下步骤：

  1）以自己的W,H给出自己初始的可见区域

  2）减去自己上面窗口所覆盖的区域

[](http://hi.csdn.net/attachment/201006/14/0_1276518612eyG3.gif)

在绘制时，Layer将根据自己的可将区域做相应的区域数据Copy。

（3）handleRepaint()

composeSurfaces（需要刷新区域）：

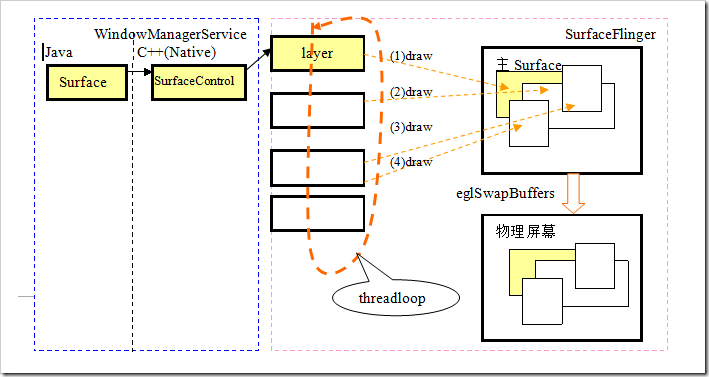
根据每个Layer的可见区域与需要刷新区域的交集区域从Z-Order序列从底部开始绘制到主Surface上。

（4）postFramebuffer()

（DisplayHardware）hw.flip(mInvalidRegion);

eglSwapBuffers(display,mSurface) :将mSruface投递到屏幕。

现在SurfaceFlinger干的事情利用下面的示意图表示出来：

[](http://hi.csdn.net/attachment/201006/14/0_1276518615Bgfw.gif)

### 4.2.5 Android GDI[之SurfaceFlinger](http://blog.csdn.net/maxleng/article/details/5671271)动态结构

#### 4.2.5.1 SurfaceSession的建立

客户端请求建立Surface时，首先在要与SurfaceFlinger建立一个Session,然后再Session上建立一个Connection通过概念返回Bclient对象。WindowManagerService在添加第一个窗口前会检查SurfaceSession是否建立，如何没有建立，将会新建立一个实例来代表与SurfaceFlinger的一个连接。

new SurfaceSession()@windowAddedLocked() @WindowManagerService.java。

SurfaceSession的建立过程大部分是在C++ Native空间中完成的，表现在SurfaceSession的初始化函数：init()本地函数上。从下面的初始化函数可以看到：

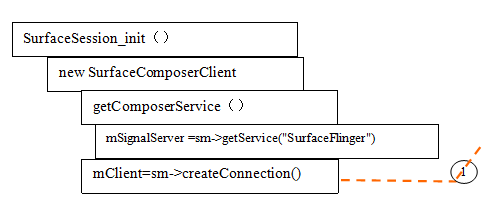
Init()<->[SurfaceSession\_init@android\_view\_Surface.cpp](mailto:SurfaceSession_init@android_view_Surface.cpp)

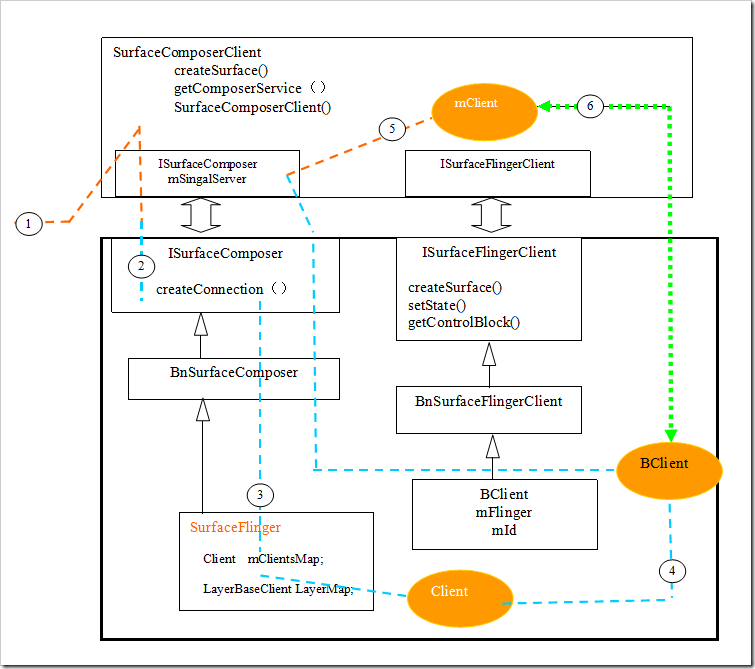
     new SurfaceComposerClient

SurfaceSession在C++Native空间建立一个SurfaceComposerClient实例。而该实例的建立实现了如下的与SurfaceFlinger通讯基础：

（1）建立了代理SurfaceFlinger服务的代理服务端

（2）建立了IsurfaceFlingerClient连接，在SurfaceFlinger端建立了对应的Client，并将BClient返回给WindowManagerService。

[](http://hi.csdn.net/attachment/201006/14/0_1276524101W877.gif)

[](http://hi.csdn.net/attachment/201006/14/0_1276524104Q1xk.gif)

#### 4.2.5.2 Surface的建立

在WindowManagerService中WindowState类中，我们知道每个主窗口子啊需要是都需要建立一个Surface与之对应。win.createSurfaceLocked()@relayoutWindow

Surface.java

Init()< -- >Surface\_init(….,session,pid,dpy,w,h,format)@android\_view\_Surface.cpp

     SurfaceControl surface(client->createSurface

      在mClient的连接上：建立ISurface接口：

      M\_Client->greateSurface（...）@

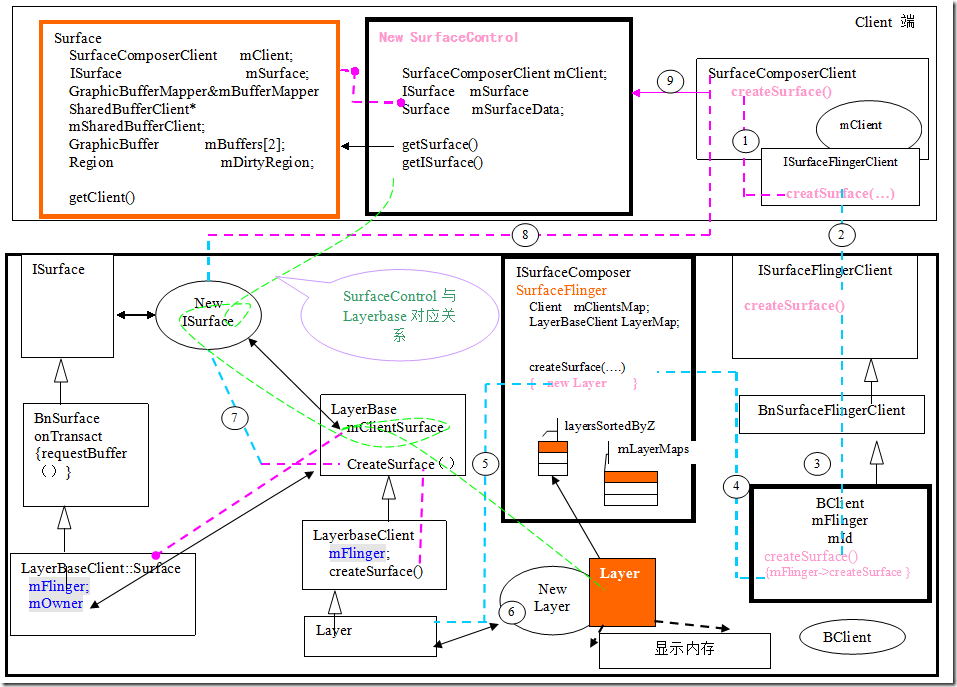
        Bclient ::createSurface(mId...)@SurfaceFlinger.cpp

            mFlinger->createSurface(clientid....)

               createNormalSurfaceLocked

\*createNormalSurfaceLocked:建立一个Layer分配显示内存

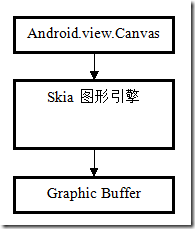
\*createPushBuffersSurfaceLocked:建立一个LayBuffer但是不分配显示内存。

[](http://hi.csdn.net/attachment/201006/14/0_1276524110bgH4.gif)

### 4.2.6 Android GDI之Surface&Canvas

#### 4.2.6.1 Surface&Canvas

Canvas为在画布的意思。Android上层的作图几乎都通过Canvas实例来完成，其实Canvas更多是一种接口的包装。drawPaints ，drawPoints，drawRect，drawBitmap ...

[](http://hi.csdn.net/attachment/201006/14/0_1276524262qcXC.gif)

对于本节，我们不去研究Skia图形引擎本身，我们需要了解的我们的所做的图形到底放置到了那个地方，并且这个Canvas如何与Surface连接在一起的。

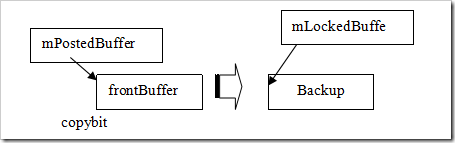
Canvas（Java）在C++Native层有一个Native Canvas的C++对象所对应。

lockCanvas()@java

[Surface\_lockCanvas@android\_view\_Surface.cpp](mailto:Surface_lockCanvas@android_view_Surface.cpp)

      SurfaceControl->new Surface(control) @Surface.cpp

      Surface: lock操作：

[](http://hi.csdn.net/attachment/201006/14/0_1276524264rrrr.gif)

      GraphicBuffer :lock

        getBufferMapper().lock<-> GraphicBufferMapper ::lock

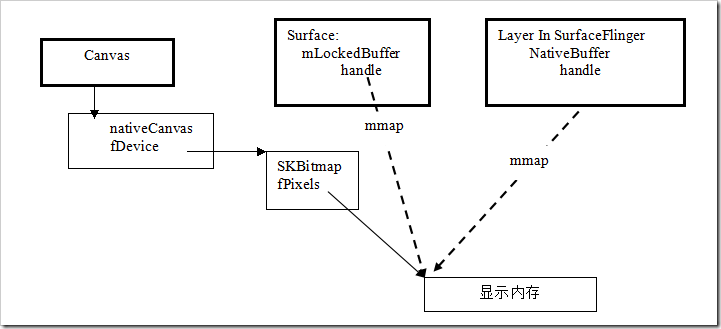
                  mAllocMod->lock<->gralloc\_module\_t::lock

通过SurfaceLock可取得Surface（mLockedBuffe）所对应的图形缓冲区地址。

（1） 建立与SkCanvas连接的位图设备，而该位图使用上面取得的图形缓冲区地址做自己的位图内存。

（2） 设置SkCanvas的作图目标设备为该位图。

通过该过程就建立起了SurfaceControl与Canvas之间的联系。

[](http://hi.csdn.net/attachment/201006/14/0_1276524267jPRF.gif)

#### 4.2.6.2 View:OnDraw的本源

不是使用OpenGL绘制时，Android在View属性发生变化，新建View时，或者Z-order发生变化时，需要对系统屏幕上的View重新绘制，此时我们的View会执行OnDraw(canvas），这个根源在哪里呢？

ViewRoot.Java

  performTraversals(..)

     …

     draw()

        canvas = surface.lockCanvas(dirty);

         …

      mView.draw(canvas);

         draw(cavas)@view.java

             background.draw(canvas);

**onDraw(cavas)**

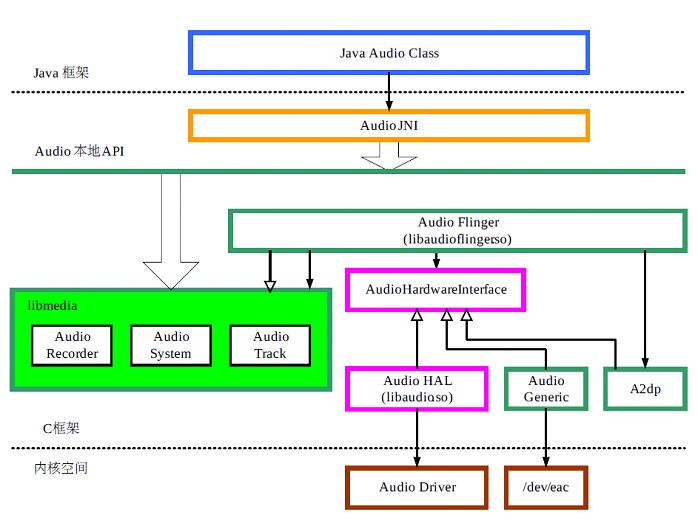
             dispatchDraw(cavas)

              onDrawScrolbars(cavas)

    surface.unlockCanvasAndPost(canvas);

## 4.3 MediaPlayer Audio系统

Android的Audio系统结构:



在Stagefright中，AwesomePlayer只负责完成Video的处理，而Audio部分则交由AudioPlayer完成。

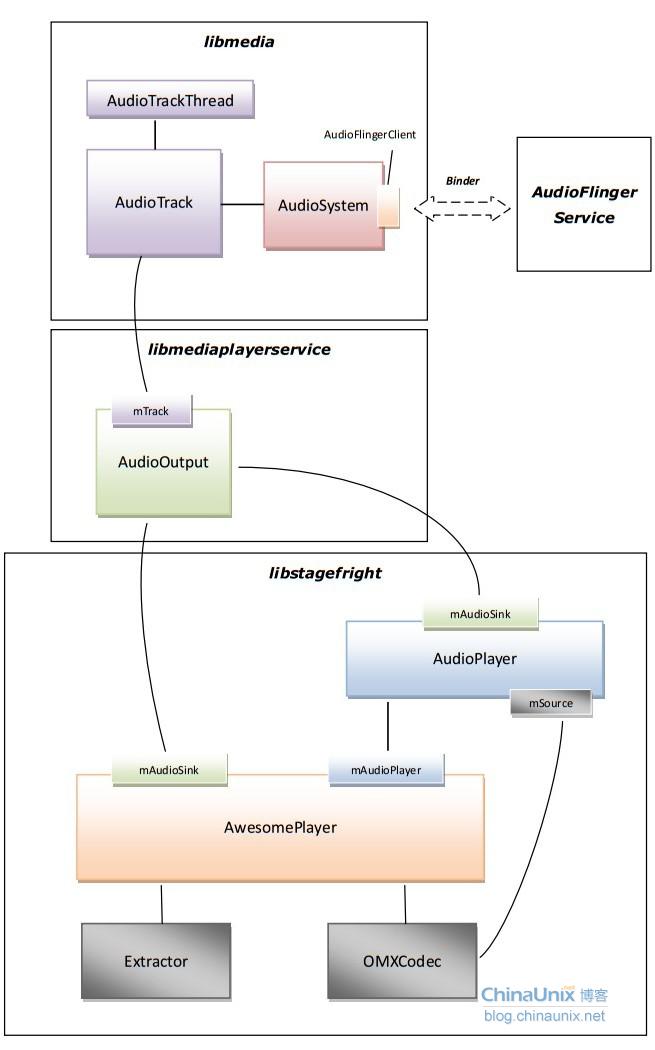
整个Android Mediaframework 的运作部分可分为三层，即libmedia, libmediaplayerservice, libstagefright三大模块。AudioPlayer与AwesomePlayer同在stagefright模块，但AudioPlayer运作却联系到了mediaplayerservice层以及media层，如图所示。其中，mediaplayerservice模块中的AudioOutput作为AudioSink与AudioPlayer相接，而其本身又与media模块中的AudioTrack交互；AudioTrack是AudioThread线程触发的对象，主要完成对Audiobuffer的处理。AudioSystem则作为工具对象被AudioTrack所使用，例如获取audio的samplingrate, framecount等，另一个重要的作用就是AudioSystem通过Binder机制直接与AudioFingerService进行交互，并将AudioFingerClient交给AudioTrack使用。

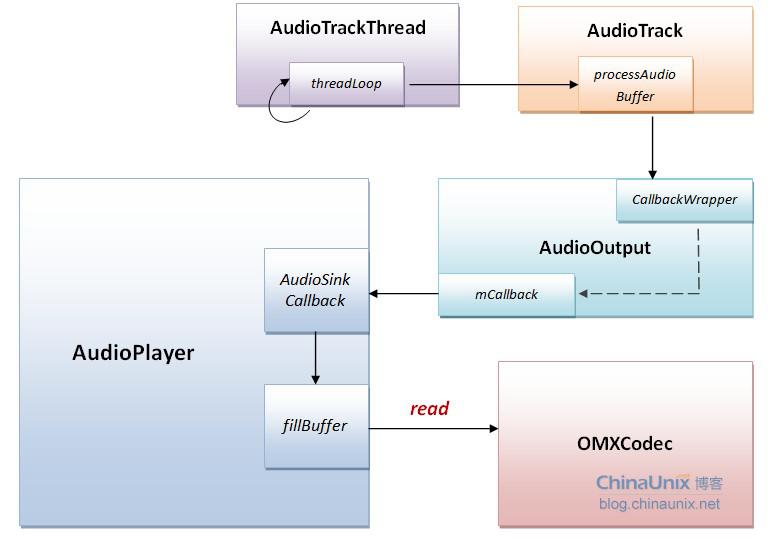
<http://www.codeceo.com/article/android-audiopolicyservice-start.html>

<http://blog.csdn.net/droidphone/article/details/5941344>

<http://blog.chinaunix.net/uid-24227137-id-2948838.html>

<http://www.it165.net/pro/html/201410/23660.html>





### 4.3.1 AudioTrack分析（JAVA）

先看看Audio里边有哪些东西？通过Android的SDK文档，发现主要有三个：

* AudioManager：这个主要是用来管理Audio系统的
* AudioTrack：这个主要是用来播放声音的
* AudioRecord：这个主要是用来录音的

其中AudioManager的理解需要考虑整个系统上声音的策略问题，例如来电话铃声，短信铃声等，主要是策略上的问题。一般看来，最简单的就是播放声音了。所以我们打算从AudioTrack开始分析。

JAVA的AudioTrack类的代码在：

framework/base/media/java/android/media/AudioTrack.java中。

#### 4.3.1.1 AudioTrack API的使用例子

先看看使用例子，然后跟进去分析。至于AudioTrack的其他使用方法和说明，需要大家自己去看API文档了。

//根据采样率，采样精度，单双声道来得到frame的大小。

int bufsize = AudioTrack.getMinBufferSize(8000,//每秒8K个点

　　AudioFormat.CHANNEL\_CONFIGURATION\_STEREO,//双声道

AudioFormat.ENCODING\_PCM\_16BIT);//一个采样点16比特-2个字节

//注意，按照数字音频的知识，这个算出来的是一秒钟buffer的大小。

//创建AudioTrack

AudioTrack trackplayer = new AudioTrack(AudioManager.STREAM\_MUSIC, 8000,

　　AudioFormat.CHANNEL\_CONFIGURATION\_ STEREO,

　　AudioFormat.ENCODING\_PCM\_16BIT,

　　bufsize,

AudioTrack.MODE\_STREAM);//

 trackplayer.play() ;//开始

trackplayer.write(bytes\_pkg, 0, bytes\_pkg.length) ;//往track中写数据

….

trackplayer.stop();//停止播放

trackplayer.release();//释放底层资源。

这里需要解释下两个东西：

1 AudioTrack.MODE\_STREAM的意思：

AudioTrack中有MODE\_STATIC和MODE\_STREAM两种分类。STREAM的意思是由用户在应用程序通过write方式把数据一次一次得写到audiotrack中。这个和我们在socket中发送数据一样，应用层从某个地方获取数据，例如通过编解码得到PCM数据，然后write到audiotrack。

这种方式的坏处就是总是在JAVA层和Native层交互，效率损失较大。

而STATIC的意思是一开始创建的时候，就把音频数据放到一个固定的buffer，然后直接传给audiotrack，后续就不用一次次得write了。AudioTrack会自己播放这个buffer中的数据。

这种方法对于铃声等内存占用较小，延时要求较高的声音来说很适用。

2 StreamType

这个在构造AudioTrack的第一个参数中使用。这个参数和Android中的AudioManager有关系，涉及到手机上的音频管理策略。

Android将系统的声音分为以下几类常见的（未写全）：

         STREAM\_ALARM：警告声

         STREAM\_MUSCI：音乐声，例如music等

         STREAM\_RING：铃声

         STREAM\_SYSTEM：系统声音

         STREAM\_VOCIE\_CALL：电话声音

为什么要分这么多呢？以前在台式机上开发的时候很少知道有这么多的声音类型，不过仔细思考下，发现这样做是有道理的。例如你在听music的时候接到电话，这个时候music播放肯定会停止，此时你只能听到电话，如果你调节音量的话，这个调节肯定只对电话起作用。当电话打完了，再回到music，你肯定不用再调节音量了。

其实系统将这几种声音的数据分开管理，所以，这个参数对AudioTrack来说，它的含义就是告诉系统，我现在想使用的是哪种类型的声音，这样系统就可以对应管理他们了。

#### 4.3.1.2分析之getMinBufferSize

AudioTrack的例子就几个函数。先看看第一个函数：

AudioTrack.getMinBufferSize(8000,//每秒8K个点

　　AudioFormat.CHANNEL\_CONFIGURATION\_STEREO,//双声道

AudioFormat.ENCODING\_PCM\_16BIT);

----->AudioTrack.JAVA

//注意，这是个static函数

static public int getMinBufferSize(int sampleRateInHz, int channelConfig, int audioFormat) {

        int channelCount = 0;

        switch(channelConfig) {

        case AudioFormat.CHANNEL\_OUT\_MONO:

        case AudioFormat.CHANNEL\_CONFIGURATION\_MONO:

            channelCount = 1;

            break;

        case AudioFormat.CHANNEL\_OUT\_STEREO:

        case AudioFormat.CHANNEL\_CONFIGURATION\_STEREO:

            channelCount = 2;--->看到了吧，外面名字搞得这么酷，其实就是指声道数

            break;

        default:

            loge("getMinBufferSize(): Invalid channel configuration.");

            return AudioTrack.ERROR\_BAD\_VALUE;

        }

    //目前只支持PCM8和PCM16精度的音频

        if ((audioFormat != AudioFormat.ENCODING\_PCM\_16BIT)

            && (audioFormat != AudioFormat.ENCODING\_PCM\_8BIT)) {

            loge("getMinBufferSize(): Invalid audio format.");

            return AudioTrack.ERROR\_BAD\_VALUE;

        }

      //ft，对采样频率也有要求，太低或太高都不行，人耳分辨率在20HZ到40KHZ之间

        if ( (sampleRateInHz < 4000) || (sampleRateInHz > 48000) ) {

            loge("getMinBufferSize(): " + sampleRateInHz +"Hz is not a supported sample rate.");

            return AudioTrack.ERROR\_BAD\_VALUE;

        }

       //调用native函数，够烦的，什么事情都搞到JNI层去。

        int size = native\_get\_min\_buff\_size(sampleRateInHz, channelCount, audioFormat);

        if ((size == -1) || (size == 0)) {

            loge("getMinBufferSize(): error querying hardware");

            return AudioTrack.ERROR;

        }

        else {

            return size;

        }

native\_get\_min\_buff\_size--->在framework/base/core/jni/android\_media\_track.cpp中实现。（不了解JNI的一定要学习下，否则只能在JAVA层搞，太狭隘了。）最终对应到函数

static jint android\_media\_AudioTrack\_get\_min\_buff\_size(JNIEnv \*env,  jobject thiz,

jint sampleRateInHertz, jint nbChannels, jint audioFormat)

{//注意我们传入的参数是：

//sampleRateInHertz = 8000

//nbChannels = 2；

//audioFormat = AudioFormat.ENCODING\_PCM\_16BIT

    int afSamplingRate;

    int afFrameCount;

    uint32\_t afLatency;

//下面涉及到AudioSystem，这里先不解释了，

//反正知道从AudioSystem那查询了一些信息

    if (AudioSystem::getOutputSamplingRate(&afSamplingRate) != NO\_ERROR) {

        return -1;

    }

    if (AudioSystem::getOutputFrameCount(&afFrameCount) != NO\_ERROR) {

        return -1;

    }

    if (AudioSystem::getOutputLatency(&afLatency) != NO\_ERROR) {

        return -1;

    }

//音频中最常见的是frame这个单位，什么意思？经过多方查找，最后还是在ALSA的wiki中

//找到解释了。一个frame就是1个采样点的字节数\*声道。为啥搞个frame出来？因为对于多//声道的话，用1个采样点的字节数表示不全，因为播放的时候肯定是多个声道的数据都要播出来//才行。所以为了方便，就说1秒钟有多少个frame，这样就能抛开声道数，把意思表示全了。

    // Ensure that buffer depth covers at least audio hardware latency

    uint32\_t minBufCount = afLatency / ((1000 \* afFrameCount)/afSamplingRate);

    if (minBufCount < 2) minBufCount = 2;

uint32\_t minFrameCount =

 (afFrameCount\*sampleRateInHertz\*minBufCount)/afSamplingRate;

//下面根据最小的framecount计算最小的buffersize

int minBuffSize = minFrameCount

            \* (audioFormat == javaAudioTrackFields.PCM16 ? 2 : 1)

            \* nbChannels;

    return minBuffSize;

}

getMinBufSize函数完了后，我们得到一个满足最小要求的缓冲区大小。这样用户分配缓冲区就有了依据。下面就需要创建AudioTrack对象了

#### 4.3.1.3分析之new AudioTrack

先看看调用函数：

AudioTrack trackplayer = new AudioTrack(

AudioManager.STREAM\_MUSIC,

8000,

　　AudioFormat.CHANNEL\_CONFIGURATION\_ STEREO,

　　AudioFormat.ENCODING\_PCM\_16BIT,

　　bufsize,

AudioTrack.MODE\_STREAM);//

其实现代码在AudioTrack.java中。

public AudioTrack(int streamType, int sampleRateInHz, int channelConfig, int audioFormat,

            int bufferSizeInBytes, int mode)

    throws IllegalArgumentException {

        mState = STATE\_UNINITIALIZED;

        // 获得主线程的Looper，这个在MediaScanner分析中已经讲过了

        if ((mInitializationLooper = Looper.myLooper()) == null) {

            mInitializationLooper = Looper.getMainLooper();

        }

    //检查参数是否合法之类的，可以不管它

        audioParamCheck(streamType, sampleRateInHz, channelConfig, audioFormat, mode);

   //我是用getMinBufsize得到的大小，总不会出错吧？

        audioBuffSizeCheck(bufferSizeInBytes);

        // 调用native层的native\_setup，把自己的WeakReference传进去了

     //不了解JAVA WeakReference的可以上网自己查一下，很简单的

        int initResult = native\_setup(new WeakReference<AudioTrack>(this),

                mStreamType, 这个值是AudioManager.STREAM\_MUSIC

 mSampleRate, 这个值是8000

mChannels, 这个值是2

mAudioFormat,这个值是AudioFormat.ENCODING\_PCM\_16BIT

                mNativeBufferSizeInBytes, //这个是刚才getMinBufSize得到的

mDataLoadMode);DataLoadMode是MODE\_STREAM

         ....

}

上面函数调用最终进入了JNI层android\_media\_AudioTrack.cpp下面的函数

static int

android\_media\_AudioTrack\_native\_setup(JNIEnv \*env, jobject thiz, jobject weak\_this,

        jint streamType, jint sampleRateInHertz, jint channels,

        jint audioFormat, jint buffSizeInBytes, jint memoryMode)

{

    int afSampleRate;

    int afFrameCount;

   下面又要调用一堆东西，烦不烦呐？具体干什么用的，以后分析到AudioSystem再说。

    AudioSystem::getOutputFrameCount(&afFrameCount, streamType)；

   AudioSystem::getOutputSamplingRate(&afSampleRate, streamType)；

   AudioSystem::isOutputChannel(channels)；

    popCount是统计一个整数中有多少位为1的算法

int nbChannels = AudioSystem::popCount(channels);

    if (streamType == javaAudioTrackFields.STREAM\_MUSIC) {

        atStreamType = AudioSystem::MUSIC;

    }

   int bytesPerSample = audioFormat == javaAudioTrackFields.PCM16 ? 2 : 1;

    int format = audioFormat == javaAudioTrackFields.PCM16 ?

            AudioSystem::PCM\_16\_BIT : AudioSystem::PCM\_8\_BIT;

    int frameCount = buffSizeInBytes / (nbChannels \* bytesPerSample);

//上面是根据Buffer大小和一个Frame大小来计算帧数的。

// AudioTrackJniStorage，就是一个保存一些数据的地方，这

//里边有一些有用的知识，下面再详细解释

    AudioTrackJniStorage\* lpJniStorage = new AudioTrackJniStorage();

     jclass clazz = env->GetObjectClass(thiz);

      lpJniStorage->mCallbackData.audioTrack\_class = (jclass)env->NewGlobalRef(clazz);

     lpJniStorage->mCallbackData.audioTrack\_ref = env->NewGlobalRef(weak\_this);

     lpJniStorage->mStreamType = atStreamType;

//创建真正的AudioTrack对象

    AudioTrack\* lpTrack = new AudioTrack();

       if (memoryMode == javaAudioTrackFields.MODE\_STREAM) {

  //如果是STREAM流方式的话，把刚才那些参数设进去

       lpTrack->set(

            atStreamType,// stream type

            sampleRateInHertz,

            format,// word length, PCM

            channels,

            frameCount,

            0,// flags

            audioCallback,

&(lpJniStorage->mCallbackData),//callback, callback data (user)

            0,// notificationFrames == 0 since not using EVENT\_MORE\_DATA to feed the AudioTrack

            0,// 共享内存，STREAM模式需要用户一次次写，所以就不用共享内存了

            true);// thread can call Java

    } else if (memoryMode == javaAudioTrackFields.MODE\_STATIC) {

         //如果是static模式，需要用户一次性把数据写进去，然后

       //再由audioTrack自己去把数据读出来，所以需要一个共享内存

//这里的共享内存是指C++AudioTrack和AudioFlinger之间共享的内容

 //因为真正播放的工作是由AudioFlinger来完成的。

          lpJniStorage->allocSharedMem(buffSizeInBytes);

          lpTrack->set(

            atStreamType,// stream type

            sampleRateInHertz,

            format,// word length, PCM

            channels,

            frameCount,

            0,// flags

            audioCallback,

&(lpJniStorage->mCallbackData),//callback, callback data (user));

            0,// notificationFrames == 0 since not using EVENT\_MORE\_DATA to feed the AudioTrack

            lpJniStorage->mMemBase,// shared mem

            true);// thread can call Java

    }

    if (lpTrack->initCheck() != NO\_ERROR) {

        LOGE("Error initializing AudioTrack");

        goto native\_init\_failure;

    }

//又来这一招，把C++AudioTrack对象指针保存到JAVA对象的一个变量中

//这样，Native层的AudioTrack对象就和JAVA层的AudioTrack对象关联起来了。

    env->SetIntField(thiz, javaAudioTrackFields.nativeTrackInJavaObj, (int)lpTrack);

    env->SetIntField(thiz, javaAudioTrackFields.jniData, (int)lpJniStorage);

  }

1 AudioTrackJniStorage详解

这个类其实就是一个辅助类，但是里边有一些知识很重要，尤其是Android封装的一套共享内存的机制。这里一并讲解，把这块搞清楚了，我们就能轻松得在两个进程间进行内存的拷贝。

AudioTrackJniStorage的代码很简单。

struct audiotrack\_callback\_cookie {

    jclass      audioTrack\_class;

    jobject     audioTrack\_ref;

 };  cookie其实就是把JAVA中的一些东西保存了下，没什么特别的意义

class AudioTrackJniStorage {

    public:

        sp<MemoryHeapBase>         mMemHeap;//这两个Memory很重要

        sp<MemoryBase>             mMemBase;

        audiotrack\_callback\_cookie mCallbackData;

        int                        mStreamType;

      bool allocSharedMem(int sizeInBytes) {

        mMemHeap = new MemoryHeapBase(sizeInBytes, 0, "AudioTrack Heap Base");

        mMemBase = new MemoryBase(mMemHeap, 0, sizeInBytes);

//注意用法，先弄一个HeapBase，再把HeapBase传入到MemoryBase中去。

        return true;

    }

};

2 MemoryHeapBase

MemroyHeapBase也是Android搞的一套基于Binder机制的对内存操作的类。既然是Binder机制，那么肯定有一个服务端（Bnxxx），一个代理端Bpxxx。看看MemoryHeapBase定义：

class MemoryHeapBase : public virtual BnMemoryHeap

{

  果然，从BnMemoryHeap派生，那就是Bn端。这样就和Binder挂上钩了

//Bp端调用的函数最终都会调到Bn这来

对Binder机制不了解的，可以参考：

http://blog.csdn.net/Innost/archive/2011/01/08/6124685.aspx

  有好几个构造函数，我们看看我们使用的：

MemoryHeapBase::MemoryHeapBase(size\_t size, uint32\_t flags, char const \* name)

    : mFD(-1), mSize(0), mBase(MAP\_FAILED), mFlags(flags),

      mDevice(0), mNeedUnmap(false)

{

    const size\_t pagesize = getpagesize();

size = ((size + pagesize-1) & ~(pagesize-1));

//创建共享内存，ashmem\_create\_region这个是系统提供的,可以不管它

//设备上打开的是/dev/ashmem设备，而Host上打开的是一个tmp文件

int fd = ashmem\_create\_region(name == NULL ? "MemoryHeapBase" : name, size);

mapfd(fd, size);//把刚才那个fd通过mmap方式得到一块内存

//不明白得去man mmap看看

mapfd完了后，mBase变量指向内存的起始位置, mSize是分配的内存大小，mFd是

ashmem\_create\_region返回的文件描述符

}

MemoryHeapBase提供了一下几个函数，可以获取共享内存的大小和位置。

getBaseID()--->返回mFd，如果为负数，表明刚才创建共享内存失败了

getBase()->返回mBase，内存位置

  getSize()->返回mSize，内存大小

有了MemoryHeapBase，又搞了一个MemoryBase，这又是一个和Binder机制挂钩的类。

唉，这个估计是一个在MemoryHeapBase上的方便类吧？因为我看见了offset

那么估计这个类就是一个能返回当前Buffer中写位置（就是offset）的方便类

这样就不用用户到处去计算读写位置了。

class MemoryBase : public BnMemory

{

public:

    MemoryBase(const sp<IMemoryHeap>& heap, ssize\_t offset, size\_t size);

    virtual sp<IMemoryHeap> getMemory(ssize\_t\* offset, size\_t\* size) const;

protected:

    size\_t getSize() const { return mSize; }

    ssize\_t getOffset() const { return mOffset; }

    const sp<IMemoryHeap>& getHeap() const { return mHeap; }

};

好了，明白上面两个MemoryXXX，我们可以猜测下大概的使用方法了。

         BnXXX端先分配BnMemoryHeapBase和BnMemoryBase，

         然后把BnMemoryBase传递到BpXXX

         BpXXX就可以使用BpMemoryBase得到BnXXX端分配的共享内存了。

注意，既然是进程间共享内存，那么Bp端肯定使用memcpy之类的函数来操作内存，这些函数是没有同步保护的，而且Android也不可能在系统内部为这种共享内存去做增加同步保护。所以看来后续在操作这些共享内存的时候，肯定存在一个跨进程的同步保护机制。我们在后面讲实际播放的时候会碰到。

另外，这里的SharedBuffer最终会在Bp端也就是AudioFlinger那用到。

#### 4.3.1.4分析之play和write

JAVA层到这一步后就是调用play和write了。JAVA层这两个函数没什么内容，都是直接转到native层干活了。

先看看play函数对应的JNI函数

static void

android\_media\_AudioTrack\_start(JNIEnv \*env, jobject thiz)

{

//看见没，从JAVA那个AudioTrack对象获取保存的C++层的AudioTrack对象指针

//从int类型直接转换成指针。要是以后ARM变成64位平台了，看google怎么改！

    AudioTrack \*lpTrack = (AudioTrack \*)env->GetIntField(

        thiz, javaAudioTrackFields.nativeTrackInJavaObj);

    lpTrack->start(); //这个以后再说

}

下面是write。我们写的是short数组，

static jint

android\_media\_AudioTrack\_native\_write\_short(JNIEnv \*env,  jobject thiz,

                                                  jshortArray javaAudioData,

                                                  jint offsetInShorts,

jint sizeInShorts,

                                                  jint javaAudioFormat) {

    return (android\_media\_AudioTrack\_native\_write(env, thiz,

                                                 (jbyteArray) javaAudioData,

                                                 offsetInShorts\*2, sizeInShorts\*2,

                                                 javaAudioFormat)

            / 2);

}

烦人，又根据Byte还是Short封装了下，最终会调到重要函数writeToTrack去

jint writeToTrack(AudioTrack\* pTrack, jint audioFormat, jbyte\* data,

                  jint offsetInBytes, jint sizeInBytes) {

      ssize\_t written = 0;

    // regular write() or copy the data to the AudioTrack's shared memory?

if (pTrack->sharedBuffer() == 0) {

//创建的是流的方式，所以没有共享内存在track中

//还记得我们在native\_setup中调用的set吗？流模式下AudioTrackJniStorage可没创建

//共享内存

        written = pTrack->write(data + offsetInBytes, sizeInBytes);

    } else {

        if (audioFormat == javaAudioTrackFields.PCM16) {

            // writing to shared memory, check for capacity

            if ((size\_t)sizeInBytes > pTrack->sharedBuffer()->size()) {

                sizeInBytes = pTrack->sharedBuffer()->size();

            }

           //看见没？STATIC模式的，就直接把数据拷贝到共享内存里

          //当然，这个共享内存是pTrack的，是我们在set时候把AudioTrackJniStorage的

//共享设进去的

            memcpy(pTrack->sharedBuffer()->pointer(),

data + offsetInBytes, sizeInBytes);

            written = sizeInBytes;

        } else if (audioFormat == javaAudioTrackFields.PCM8) {

           PCM8格式的要先转换成PCM16

    }

    return written;

}

到这里，似乎很简单啊，JAVA层的AudioTrack，无非就是调用write函数，而实际由JNI层的C++ AudioTrack write数据。反正JNI这层是再看不出什么有意思的东西了。

### 4.3.2 AudioTrack分析（C++）

接上面的内容，我们知道在JNI层，有以下几个步骤：

         new了一个AudioTrack

         调用set函数，把AudioTrackJniStorage等信息传进去

         调用了AudioTrack的start函数

         调用AudioTrack的write函数

那么，我们就看看真正干活的的C++AudioTrack吧。

AudioTrack.cpp位于framework/base/libmedia/AudioTrack.cpp

#### 4.3.2.1 new AudioTrack()和set调用

JNI层调用的是最简单的构造函数：

AudioTrack::AudioTrack()

    : mStatus(NO\_INIT) //把状态初始化成NO\_INIT。Android大量使用了设计模式中的state。

{

}

接下来调用set。我们看看JNI那set了什么

  lpTrack->set(

            atStreamType, //应该是Music吧

            sampleRateInHertz,//8000

            format,// 应该是PCM\_16吧

            channels,//立体声=2

            frameCount,//

            0,// flags

            audioCallback, //JNI中的一个回调函数

&(lpJniStorage->mCallbackData),//回调函数的参数

            0,// 通知回调函数，表示AudioTrack需要数据，不过暂时没用上

            0,//共享buffer地址，stream模式没有

            true);//回调线程可以调JAVA的东西

那我们看看set函数把。

status\_t AudioTrack::set(

        int streamType,

        uint32\_t sampleRate,

        int format,

        int channels,

        int frameCount,

        uint32\_t flags,

        callback\_t cbf,

        void\* user,

        int notificationFrames,

        const sp<IMemory>& sharedBuffer,

        bool threadCanCallJava)

{

   ...前面一堆的判断，等以后讲AudioSystem再说

audio\_io\_handle\_t output =

AudioSystem::getOutput((AudioSystem::stream\_type)streamType,

            sampleRate, format, channels, (AudioSystem::output\_flags)flags);

   //createTrack？看来这是真正干活的

    status\_t status = createTrack(streamType, sampleRate, format, channelCount,

                                  frameCount, flags, sharedBuffer, output);

  //cbf是JNI传入的回调函数audioCallback

     if (cbf != 0) { //看来，怎么着也要创建这个线程了！

        mAudioTrackThread = new AudioTrackThread(\*this, threadCanCallJava);

       }

   return NO\_ERROR;

}

看看真正干活的createTrack

status\_t AudioTrack::createTrack(

        int streamType,

        uint32\_t sampleRate,

        int format,

        int channelCount,

        int frameCount,

        uint32\_t flags,

        const sp<IMemory>& sharedBuffer,

        audio\_io\_handle\_t output)

{

status\_t status;

//啊，看来和audioFlinger挂上关系了呀。

    const sp<IAudioFlinger>& audioFlinger = AudioSystem::get\_audio\_flinger();

  //下面这个调用最终会在AudioFlinger中出现。暂时不管它。

    sp<IAudioTrack> track = audioFlinger->createTrack(getpid(),

                                                      streamType,

                                                      sampleRate,

                                                      format,

                                                      channelCount,

                                                      frameCount,

                                                      ((uint16\_t)flags) << 16,

                                                      sharedBuffer,

                                                      output,

                                                      &status);

   //看见没，从track也就是AudioFlinger那边得到一个IMemory接口

//这个看来就是最终write写入的地方

    sp<IMemory> cblk = track->getCblk();

    mAudioTrack.clear();

    mAudioTrack = track;

    mCblkMemory.clear();//sp<XXX>的clear，就看着做是delete XXX吧

    mCblkMemory = cblk;

    mCblk = static\_cast<audio\_track\_cblk\_t\*>(cblk->pointer());

    mCblk->out = 1;

    mFrameCount = mCblk->frameCount;

if (sharedBuffer == 0) {

//终于看到buffer相关的了。注意我们这里的情况

//STREAM模式没有传入共享buffer，但是数据确实又需要buffer承载。

//反正AudioTrack是没有创建buffer，那只能是刚才从AudioFlinger中得到

//的buffer了。

        mCblk->buffers = (char\*)mCblk + sizeof(audio\_track\_cblk\_t);

    }

    return NO\_ERROR;

}

还记得我们说MemoryXXX没有同步机制，所以这里应该有一个东西能体现同步的，

那么我告诉大家，就在audio\_track\_cblk\_t结构中。它的头文件在

framework/base/include/private/media/AudioTrackShared.h

实现文件就在AudioTrack.cpp中

audio\_track\_cblk\_t::audio\_track\_cblk\_t()

//看见下面的SHARED没？都是表示跨进程共享的意思。这个我就不跟进去说了

//等以后介绍同步方面的知识时，再细说

    : lock(Mutex::SHARED), cv(Condition::SHARED), user(0), server(0),

    userBase(0), serverBase(0), buffers(0), frameCount(0),

    loopStart(UINT\_MAX), loopEnd(UINT\_MAX), loopCount(0), volumeLR(0),

    flowControlFlag(1), forceReady(0)

{

}

到这里，大家应该都有个大概的全景了。

         AudioTrack得到AudioFlinger中的一个IAudioTrack对象，这里边有一个很重要的数据结构audio\_track\_cblk\_t，它包括一块缓冲区地址，包括一些进程间同步的内容，可能还有数据位置等内容

         AudioTrack启动了一个线程，叫AudioTrackThread，这个线程干嘛的呢？还不知道

         AudioTrack调用write函数，肯定是把数据写到那块共享缓冲了，然后IAudioTrack在另外一个进程AudioFlinger中（其实AudioFlinger是一个服务，在mediaservice中运行）接收数据，并最终写到音频设备中。

那我们先看看AudioTrackThread干什么了。

调用的语句是：

mAudioTrackThread = new AudioTrackThread(\*this, threadCanCallJava);

AudioTrackThread从Thread中派生，这个内容在深入浅出Binder机制讲过了。

反正最终会调用AudioTrackAThread的threadLoop函数。

先看看构造函数

AudioTrack::AudioTrackThread::AudioTrackThread(AudioTrack& receiver, bool bCanCallJava)

    : Thread(bCanCallJava), mReceiver(receiver)

{  //mReceiver就是AudioTrack对象

  // bCanCallJava为TRUE

}

这个线程的启动由AudioTrack的start函数触发。

void AudioTrack::start()

{

  //start函数调用AudioTrackThread函数触发产生一个新的线程，执行mAudioTrackThread的

threadLoop

    sp<AudioTrackThread> t = mAudioTrackThread;

t->run("AudioTrackThread", THREAD\_PRIORITY\_AUDIO\_CLIENT);

//让AudioFlinger中的track也start

    status\_t status = mAudioTrack->start();

}

bool AudioTrack::AudioTrackThread::threadLoop()

{

  //太恶心了，又调用AudioTrack的processAudioBuffer函数

return mReceiver.processAudioBuffer(this);

}

bool AudioTrack::processAudioBuffer(const sp<AudioTrackThread>& thread)

{

Buffer audioBuffer;

    uint32\_t frames;

    size\_t writtenSize;

      ...回调1

         mCbf(EVENT\_UNDERRUN, mUserData, 0);

...回调2 都是传递一些信息到JNI里边

         mCbf(EVENT\_BUFFER\_END, mUserData, 0);

         // Manage loop end callback

    while (mLoopCount > mCblk->loopCount) {

        mCbf(EVENT\_LOOP\_END, mUserData, (void \*)&loopCount);

    }

  //下面好像有写数据的东西

      do {

       audioBuffer.frameCount = frames;

//获得buffer，

       status\_t err = obtainBuffer(&audioBuffer, 1);

        size\_t reqSize = audioBuffer.size;

//把buffer回调到JNI那去，这是单独一个线程，而我们还有上层用户在那不停

//地write呢，怎么会这样？

        mCbf(EVENT\_MORE\_DATA, mUserData, &audioBuffer);

         audioBuffer.size = writtenSize;

         frames -= audioBuffer.frameCount;

       releaseBuffer(&audioBuffer); //释放buffer，和obtain相对应，看来是LOCK和UNLOCK

操作了

    }

    while (frames);

   return true;

}

难道真的有两处在write数据？看来必须得到mCbf去看看了，传的是EVENT\_MORE\_DATA标志。

mCbf由set的时候传入C++的AudioTrack，实际函数是：

static void audioCallback(int event, void\* user, void \*info) {

    if (event == AudioTrack::EVENT\_MORE\_DATA) {

         //哈哈，太好了，这个函数没往里边写数据

        AudioTrack::Buffer\* pBuff = (AudioTrack::Buffer\*)info;

        pBuff->size = 0;

     }

从代码上看，本来google考虑是异步的回调方式来写数据，可惜发现这种方式会比较复杂，尤其是对用户开放的JAVA AudioTrack会很不好处理，所以嘛，偷偷摸摸得给绕过去了。

太好了，看来就只有用户的write会真正的写数据了，这个AudioTrackThread除了通知一下，也没什么实际有意义的操作了。

让我们看看write吧。

#### 4.3.2.2 write

ssize\_t AudioTrack::write(const void\* buffer, size\_t userSize)

{

  够简单，就是obtainBuffer，memcpy数据，然后releasBuffer

眯着眼睛都能想到，obtainBuffer一定是Lock住内存了，releaseBuffer一定是unlock内存了

     do {

        audioBuffer.frameCount = userSize/frameSize();

        status\_t err = obtainBuffer(&audioBuffer, -1);

         size\_t toWrite;

         toWrite = audioBuffer.size;

         memcpy(audioBuffer.i8, src, toWrite);

         src += toWrite;

        }

        userSize -= toWrite;

        written += toWrite;

        releaseBuffer(&audioBuffer);

    } while (userSize);

    return written;

}

obtainBuffer太复杂了，不过大家知道其大概工作方式就可以了

status\_t AudioTrack::obtainBuffer(Buffer\* audioBuffer, int32\_t waitCount)

{

   //恕我中间省略太多，大部分都是和当前数据位置相关，

 uint32\_t framesAvail = cblk->framesAvailable();

     cblk->lock.lock();//看见没，lock了

     result = cblk->cv.waitRelative(cblk->lock, milliseconds(waitTimeMs));

//我发现很多地方都要判断远端的AudioFlinger的状态，比如是否退出了之类的，难道

//没有一个好的方法来集中处理这种事情吗？

      if (result == DEAD\_OBJECT) {

        result = createTrack(mStreamType, cblk->sampleRate, mFormat, mChannelCount,

          mFrameCount, mFlags, mSharedBuffer,getOutput());

        }

//得到buffer

    audioBuffer->raw = (int8\_t \*)cblk->buffer(u);

  return active ? status\_t(NO\_ERROR) : status\_t(STOPPED);

}

在看看releaseBuffer

void AudioTrack::releaseBuffer(Buffer\* audioBuffer)

{

    audio\_track\_cblk\_t\* cblk = mCblk;

cblk->stepUser(audioBuffer->frameCount);

}

uint32\_t audio\_track\_cblk\_t::stepUser(uint32\_t frameCount)

{

    uint32\_t u = this->user;

    u += frameCount;

     if (out) {

          if (bufferTimeoutMs == MAX\_STARTUP\_TIMEOUT\_MS-1) {

            bufferTimeoutMs = MAX\_RUN\_TIMEOUT\_MS;

        }

    } else if (u > this->server) {

         u = this->server;

    }

    if (u >= userBase + this->frameCount) {

        userBase += this->frameCount;

    }

   this->user = u;

  flowControlFlag = 0;

  return u;

}

奇怪了，releaseBuffer没有unlock操作啊？难道我失误了？

再去看看obtainBuffer?为何写得这么晦涩难懂？

原来在obtainBuffer中会某一次进去lock，再某一次进去可能就是unlock了。没看到obtainBuffer中到处有lock,unlock,wait等同步操作吗。一定是这个道理。难怪写这么复杂。还使用了少用的goto语句。

唉，有必要这样吗！

通过这一次的分析，我自己觉得有以下几个点：

         AudioTrack的工作原理，尤其是数据的传递这一块，做了比较细致的分析，包括共享内存，跨进程的同步等，也能解释不少疑惑了。

         看起来，最重要的工作是在AudioFlinger中做的。通过AudioTrack的介绍，我们给后续深入分析AudioFlinger提供了一个切入点

  工作原理和流程嘛，再说一次好了，JAVA层就看最前面那个例子吧，实在没什么说的。

         AudioTrack被new出来，然后set了一堆信息，同时会通过Binder机制调用另外一端的AudioFlinger，得到IAudioTrack对象，通过它和AudioFlinger交互。

         调用start函数后，会启动一个线程专门做回调处理，代码里边也会有那种数据拷贝的回调，但是JNI层的回调函数实际并没有往里边写数据，大家只要看write就可以了

         用户一次次得write，那AudioTrack无非就是把数据memcpy到共享buffer中咯

         可想而知，AudioFlinger那一定有一个线程在memcpy数据到音频设备中去。我们拭目以待。

### 4.3.3 AudioFlinger分析

本文承接Audio第一部分的AudioTrack，通过AudioTrack作为AF（AudioFlinger）的客户端，来看看AF是如何完成工作的。

在AT（AudioTrack）中，我们涉及到的都是流程方面的事务，而不是系统Audio策略上的内容。WHY？因为AT是AF的客户端，而AF是Android系统中Audio管理的中枢。AT我们分析的是按流程方法，那么以AT为切入点的话，AF的分析也应该是流程分析了。

对于分析AT来说，只要能把它的调用顺序（也就是流程说清楚就可以了），但是对于AF的话，简单的分析调用流程 我自己感觉是不够的。因为我发现手机上的声音交互和管理是一件比较复杂的事情。举个简单例子，当听music的时候来电话了，声音处理会怎样？

虽然在Android中，还有一个叫AudioPolicyService的（APS）东西，但是它最终都会调用到AF中去，因为AF实际创建并管理了硬件设备。所以，针对Android声音策略上的分析，我会单独在以后来分析。

直接从头看代码是没法掌握AF的主干的，必须要有一个切入点，也就是用一个正常的调用流程来分析AF的处理流程。先看看AF的产生吧，这个C/S架构的服务者是如何产生的呢？

#### 4.3.3.1 AudioFlinger的诞生

AF是一个服务，这个就不用我多说了吧？代码在

framework/base/media/mediaserver/Main\_mediaServer.cpp中。

int main(int argc, char\*\* argv)

{

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

sp<IServiceManager> sm = defaultServiceManager();

....

    AudioFlinger::instantiate();--->AF的实例化

AudioPolicyService::instantiate();--->APS的实例化

....

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

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

}

哇塞，看来这个程序的负担很重啊。没想到。为何AF，APS要和MediaService和CameraService都放到一个篮子里？

看看AF的实例化静态函数，在framework/base/libs/audioFlinger/audioFlinger.cpp中

void AudioFlinger::instantiate() {

    defaultServiceManager()->addService( //把AF实例加入系统服务

            String16("media.audio\_flinger"), new AudioFlinger());

}

再来看看它的构造函数是什么做的。

AudioFlinger::AudioFlinger()

    : BnAudioFlinger(),//初始化基类

        mAudioHardware(0), //audio硬件的HAL对象

mMasterVolume(1.0f), mMasterMute(false), mNextThreadId(0)

{

mHardwareStatus = AUDIO\_HW\_IDLE;

//创建代表Audio硬件的HAL对象

    mAudioHardware = AudioHardwareInterface::create();

    mHardwareStatus = AUDIO\_HW\_INIT;

    if (mAudioHardware->initCheck() == NO\_ERROR) {

        setMode(AudioSystem::MODE\_NORMAL);

//设置系统的声音模式等，其实就是设置硬件的模式

        setMasterVolume(1.0f);

        setMasterMute(false);

    }

}

AF中经常有setXXX的函数，到底是干什么的呢？我们看看setMode函数。

status\_t AudioFlinger::setMode(int mode)

{

     mHardwareStatus = AUDIO\_HW\_SET\_MODE;

    status\_t ret = mAudioHardware->setMode(mode);//设置硬件的模式

    mHardwareStatus = AUDIO\_HW\_IDLE;

    return ret;

}

当然，setXXX还有些别的东西，但基本上都会涉及到硬件对象。我们暂且不管它。等分析到Audio策略再说。

好了，Android系统启动的时候，看来AF也准备好硬件了。不过，创建硬件对象就代表我们可以播放了吗？

#### 4.3.3.2 AT调用AF的流程

我这里简单的把AT调用AF的流程列一下，待会按这个顺序分析AF的工作方式。

--参加AudioTrack分析的4.1节

1. 创建

AudioTrack\* lpTrack = new AudioTrack();

lpTrack->set(...);

这个就进入到C++的AT了。下面是AT的set函数

audio\_io\_handle\_t output =

AudioSystem::getOutput((AudioSystem::stream\_type)streamType,

            sampleRate, format, channels, (AudioSystem::output\_flags)flags);

    status\_t status = createTrack(streamType, sampleRate, format, channelCount,

                                  frameCount, flags, sharedBuffer, output);

----->creatTrack会和AF打交道。我们看看createTrack重要语句

const sp<IAudioFlinger>& audioFlinger = AudioSystem::get\_audio\_flinger();

   //下面很重要，调用AF的createTrack获得一个IAudioTrack对象

    sp<IAudioTrack> track = audioFlinger->createTrack();

    sp<IMemory> cblk = track->getCblk();//获取共享内存的管理结构

总结一下创建的流程，AT调用AF的createTrack获得一个IAudioTrack对象，然后从这个对象中获得共享内存的对象。

2. start和write

看看AT的start，估计就是调用IAudioTrack的start吧？

void AudioTrack::start()

{

//果然啊...

   status\_t status = mAudioTrack->start();

}

那write呢?我们之前讲了，AT就是从共享buffer中:

         Lock缓存

         写缓存

         Unlock缓存

注意，这里的Lock和Unlock是有问题的，什么问题呢?待会我们再说

按这种方式的话，那么AF一定是有一个线程在那也是：

         Lock，

         读缓存，写硬件

         Unlock

总之，我们知道了AT的调用AF的流程了。下面一个一个看。

#### 4.3.3.3 AF流程

1 createTrack

sp<IAudioTrack> AudioFlinger::createTrack(

        pid\_t pid,//AT的pid号

        int streamType,//MUSIC，流类型

        uint32\_t sampleRate,//8000 采样率

        int format,//PCM\_16类型

        int channelCount,//2，双声道

        int frameCount,//需要创建的buffer可包含的帧数

        uint32\_t flags,

        const sp<IMemory>& sharedBuffer,//AT传入的共享buffer，这里为空

        int output,//这个是从AuidoSystem获得的对应MUSIC流类型的索引

        status\_t \*status)

{

    sp<PlaybackThread::Track> track;

    sp<TrackHandle> trackHandle;

    sp<Client> client;

    wp<Client> wclient;

    status\_t lStatus;

       {

        Mutex::Autolock \_l(mLock);

//根据output句柄，获得线程？

        PlaybackThread \*thread = checkPlaybackThread\_l(output);

//看看这个进程是不是已经是AF的客户了

//这里说明一下，由于是C/S架构，那么作为服务端的AF肯定有地方保存作为C的AT的信息

//那么，AF是根据pid作为客户端的唯一标示的

//mClients是一个类似map的数据组织结构

         wclient = mClients.valueFor(pid);

        if (wclient != NULL) {

       } else {

         //如果还没有这个客户信息，就创建一个，并加入到map中去

            client = new Client(this, pid);

            mClients.add(pid, client);

        }

//从刚才找到的那个线程对象中创建一个track

        track = thread->createTrack\_l(client, streamType, sampleRate, format,

                channelCount, frameCount, sharedBuffer, &lStatus);

    }

//喔，还有一个trackHandle，而且返回到AF端的是这个trackHandle对象

     trackHandle = new TrackHandle(track);

   return trackHandle;

}

这个AF函数中，突然冒出来了很多新类型的数据结构。说实话，我刚开始接触的时候，大脑因为常接触到这些眼生的东西而死机！大家先不要拘泥于这些东西，我会一一分析到的。

先进入到checkPlaybackThread\_l看看。

AudioFlinger::PlaybackThread \*AudioFlinger::checkPlaybackThread\_l(int output) const

{

PlaybackThread \*thread = NULL;

//看到这种indexOfKey的东西，应该立即能想到：

//喔，这可能是一个map之类的东西，根据key能找到实际的value

    if (mPlaybackThreads.indexOfKey(output) >= 0) {

        thread = (PlaybackThread \*)mPlaybackThreads.valueFor(output).get();

}

//这个函数的意思是根据output值，从一堆线程中找到对应的那个线程

    return thread;

}

看到这里很疑惑啊：

         AF的构造函数中没有创建线程，只创建了一个audio的HAL对象

         如果AT是AF的第一个客户的话，我们刚才的调用流程里边，也没看到哪有创建线程的地方呀。

         output是个什么玩意儿？为什么会根据它作为key来找线程呢？

看来，我们得去Output的来源那看看了。

我们知道，output的来源是由AT的set函数得到的：如下：

audio\_io\_handle\_t output = AudioSystem::getOutput(

(AudioSystem::stream\_type)streamType, //MUSIC类型

            sampleRate, //8000

format, //PCM\_16

channels, //2两个声道

(AudioSystem::output\_flags)flags//0

);

上面这几个参数后续不再提示了，大家知道这些值都是由AT做为切入点传进去的

然后它在调用AT自己的createTrack，最终把这个output值传递到AF了。其中audio\_io\_handle\_t类型就是一个int类型。

//叫handle啊？好像linux下这种叫法的很少，难道又是受MS的影响吗？

我们进到AudioSystem::getOutput看看。注意，大家想想这是系统的第一次调用,而且发生在AudioTrack那个进程里边。AudioSystem的位置在framework/base/media/libmedia/AudioSystem.cpp中

audio\_io\_handle\_t AudioSystem::getOutput(stream\_type stream,

                                    uint32\_t samplingRate,

                                    uint32\_t format,

                                    uint32\_t channels,

                                    output\_flags flags)

{

    audio\_io\_handle\_t output = 0;

    if ((flags & AudioSystem::OUTPUT\_FLAG\_DIRECT) == 0 &&

        ((stream != AudioSystem::VOICE\_CALL && stream != AudioSystem::BLUETOOTH\_SCO) ||

         channels != AudioSystem::CHANNEL\_OUT\_MONO ||

         (samplingRate != 8000 && samplingRate != 16000))) {

        Mutex::Autolock \_l(gLock);

//根据我们的参数，我们会走到这个里边来

//喔，又是从map中找到stream=music的output。可惜啊，我们是第一次进来

//output一定是0

        output = AudioSystem::gStreamOutputMap.valueFor(stream);

       }

if (output == 0) {

//我晕，又到AudioPolicyService(APS)

//由它去getOutput

        const sp<IAudioPolicyService>& aps = AudioSystem::get\_audio\_policy\_service();

        output = aps->getOutput(stream, samplingRate, format, channels, flags);

        if ((flags & AudioSystem::OUTPUT\_FLAG\_DIRECT) == 0) {

            Mutex::Autolock \_l(gLock);

//如果取到output了，再把output加入到AudioSystem维护的这个map中去

//说白了，就是保存一些信息吗。免得下次又这么麻烦去骚扰APS！

            AudioSystem::gStreamOutputMap.add(stream, output);

        }

    }

    return output;

}

怎么办？需要到APS中才能找到output的信息？

没办法，硬着头皮进去吧。那先得看看APS是如何创建的。不过这个刚才已经说了，是和AF一块在那个Main\_mediaService.cpp中实例化的。

位置在framework/base/lib/libaudioflinger/ AudioPolicyService.cpp中

AudioPolicyService::AudioPolicyService()

    : BnAudioPolicyService() , mpPolicyManager(NULL)

{

    // 下面两个线程以后再说

mTonePlaybackThread = new AudioCommandThread(String8(""));

mAudioCommandThread = new AudioCommandThread(String8("ApmCommandThread"));

#if (defined GENERIC\_AUDIO) || (defined AUDIO\_POLICY\_TEST)

//喔，使用普适的AudioPolicyManager，把自己this做为参数

//我们这里先使用普适的看看吧

mpPolicyManager = new AudioPolicyManagerBase(this);

//使用硬件厂商提供的特殊的AudioPolicyManager

    //mpPolicyManager = createAudioPolicyManager(this);

    }

}

我们看看AudioManagerBase的构造函数吧，在framework/base/lib/audioFlinger/

AudioPolicyManagerBase.cpp中。

AudioPolicyManagerBase::AudioPolicyManagerBase(AudioPolicyClientInterface \*clientInterface)

    : mPhoneState(AudioSystem::MODE\_NORMAL), mRingerMode(0), mMusicStopTime(0), mLimitRingtoneVolume(false)

{

mpClientInterface = clientInterface;这个client就是APS，刚才通过this传进来了

AudioOutputDescriptor \*outputDesc = new AudioOutputDescriptor();

outputDesc->mDevice = (uint32\_t)AudioSystem::DEVICE\_OUT\_SPEAKER;

    mHardwareOutput = mpClientInterface->openOutput(&outputDesc->mDevice,

                                    &outputDesc->mSamplingRate,

                                    &outputDesc->mFormat,

                                    &outputDesc->mChannels,

                                    &outputDesc->mLatency,

                                    outputDesc->mFlags);

  openOutput又交给APS的openOutput来完成了，真绕....

}

唉，看来我们还是得回到APS，

audio\_io\_handle\_t AudioPolicyService::openOutput(uint32\_t \*pDevices,

                                uint32\_t \*pSamplingRate,

                                uint32\_t \*pFormat,

                                uint32\_t \*pChannels,

                                uint32\_t \*pLatencyMs,

                                AudioSystem::output\_flags flags)

{

    sp<IAudioFlinger> af = AudioSystem::get\_audio\_flinger();

//FT,FT,FT,FT,FT,FT,FT

//绕了这么一个大圈子，竟然回到AudioFlinger中了啊？？

return af->openOutput(pDevices, pSamplingRate, (uint32\_t \*)pFormat, pChannels,

 pLatencyMs, flags);

}

在我们再次被绕晕之后，我们回眸看看足迹吧：

         在AudioTrack中，调用set函数

         这个函数会通过AudioSystem::getOutput来得到一个output的句柄

         AS的getOutput会调用AudioPolicyService的getOutput

         然后我们就没继续讲APS的getOutPut了，而是去看看APS创建的东西

         发现APS创建的时候会创建一个AudioManagerBase，这个AMB的创建又会调用APS的openOutput。

         APS的openOutput又会调用AudioFlinger的openOutput

有一个疑问，AT中set参数会和APS构造时候最终传入到AF的openOutput一样吗？如果不一样，那么构造时候openOutput的又是什么参数呢？

先放下这个悬念，我们继续从APS的getOutPut看看。

audio\_io\_handle\_t AudioPolicyService::getOutput(AudioSystem::stream\_type stream,

                                    uint32\_t samplingRate,

                                    uint32\_t format,

                                    uint32\_t channels,

                                    AudioSystem::output\_flags flags)

{

     Mutex::Autolock \_l(mLock);

//自己又不干活，由AudioManagerBase干活

    return mpPolicyManager->getOutput(stream, samplingRate, format, channels, flags);

}

进去看看吧

audio\_io\_handle\_t AudioPolicyManagerBase::getOutput(AudioSystem::stream\_type stream,

                                    uint32\_t samplingRate,

                                    uint32\_t format,

                                    uint32\_t channels,

                                    AudioSystem::output\_flags flags)

{

    audio\_io\_handle\_t output = 0;

    uint32\_t latency = 0;

    // open a non direct output

    output = mHardwareOutput; //这个是在哪里创建的？在AMB构造的时候..

    return output;

}

具体AMB的分析待以后Audio系统策略的时候我们再说吧。反正，到这里，我们知道了，在APS构造的时候会open一个Output，而这个Output又会调用AF的openOutput。

int AudioFlinger::openOutput(uint32\_t \*pDevices,

                                uint32\_t \*pSamplingRate,

                                uint32\_t \*pFormat,

                                uint32\_t \*pChannels,

                                uint32\_t \*pLatencyMs,

                                uint32\_t flags)

{

    status\_t status;

    PlaybackThread \*thread = NULL;

    mHardwareStatus = AUDIO\_HW\_OUTPUT\_OPEN;

    uint32\_t samplingRate = pSamplingRate ? \*pSamplingRate : 0;

    uint32\_t format = pFormat ? \*pFormat : 0;

    uint32\_t channels = pChannels ? \*pChannels : 0;

    uint32\_t latency = pLatencyMs ? \*pLatencyMs : 0;

     Mutex::Autolock \_l(mLock);

   //由Audio硬件HAL对象创建一个AudioStreamOut对象

    AudioStreamOut \*output = mAudioHardware->openOutputStream(\*pDevices,

                                                             (int \*)&format,

                                                             &channels,

                                                             &samplingRate,

                                                             &status);

   mHardwareStatus = AUDIO\_HW\_IDLE;

if (output != 0) {

//创建一个Mixer线程

        thread = new MixerThread(this, output, ++mNextThreadId);

        }

//终于找到了，把这个线程加入线程管理组织中

        mPlaybackThreads.add(mNextThreadId, thread);

       return mNextThreadId;

    }

}

明白了，看来AT在调用AF的createTrack的之前，AF已经在某个时候把线程创建好了，而且是一个Mixer类型的线程，看来和混音有关系呀。这个似乎和我们开始设想的AF工作有点联系喔。Lock，读缓存，写Audio硬件，Unlock。可能都是在这个线程里边做的。

2 继续createTrack

AudioFlinger::createTrack(

        pid\_t pid,

        int streamType,

        uint32\_t sampleRate,

        int format,

        int channelCount,

        int frameCount,

        uint32\_t flags,

        const sp<IMemory>& sharedBuffer,

        int output,

        status\_t \*status)

{

    sp<PlaybackThread::Track> track;

    sp<TrackHandle> trackHandle;

    sp<Client> client;

    wp<Client> wclient;

    status\_t lStatus;

    {

//假设我们找到了对应的线程

        Mutex::Autolock \_l(mLock);

        PlaybackThread \*thread = checkPlaybackThread\_l(output);

       //晕，调用这个线程对象的createTrack\_l

track = thread->createTrack\_l(client, streamType, sampleRate, format,

                channelCount, frameCount, sharedBuffer, &lStatus);

    }

        trackHandle = new TrackHandle(track);

return trackHandle；----》注意，这个对象是最终返回到AT进程中的。

   实在是....太绕了。再进去看看thread->createTrack\_l吧。\_l的意思是这个函数进入之前已经获得同步锁了。

跟着sourceinsight ctrl+鼠标左键就进入到下面这个函数。

下面这个函数的签名好长啊。这是为何？

原来Android的C++类中大量定义了内部类。说实话，我之前几年的C++的经验中基本没接触过这么频繁使用内部类的东东。--->当然，你可以说STL也大量使用了呀。

我们就把C++的内部类当做普通的类一样看待吧，其实我感觉也没什么特殊的含义，和外部类是一样的，包括函数调用，public/private之类的东西。这个和JAVA的内部类是大不一样的。

sp<AudioFlinger::PlaybackThread::Track>  AudioFlinger::PlaybackThread::createTrack\_l(

        const sp<AudioFlinger::Client>& client,

        int streamType,

        uint32\_t sampleRate,

        int format,

        int channelCount,

        int frameCount,

        const sp<IMemory>& sharedBuffer,

        status\_t \*status)

{

    sp<Track> track;

    status\_t lStatus;

    { // scope for mLock

        Mutex::Autolock \_l(mLock);

//new 一个track对象

//我有点愤怒了，Android真是层层封装啊，名字取得也非常相似。

//看看这个参数吧，注意sharedBuffer这个，此时的值应是0

        track = new Track(this, client, streamType, sampleRate, format,

                channelCount, frameCount, sharedBuffer);

       mTracks.add(track); //把这个track加入到数组中，是为了管理用的。

}

lStatus = NO\_ERROR;

   return track;

}

看到这个数组的存在，我们应该能想到什么吗？这时已经有：

         一个MixerThread，内部有一个数组保存track的

看来，不管有多少个AudioTrack，最终在AF端都有一个track对象对应，而且这些所有的track对象都会由一个线程对象来处理。----难怪是Mixer啊

再去看看new Track，我们一直还没找到共享内存在哪里创建的！！！

AudioFlinger::PlaybackThread::Track::Track(

            const wp<ThreadBase>& thread,

            const sp<Client>& client,

            int streamType,

            uint32\_t sampleRate,

            int format,

            int channelCount,

            int frameCount,

            const sp<IMemory>& sharedBuffer)

    :   TrackBase(thread, client, sampleRate, format, channelCount, frameCount, 0, sharedBuffer),

    mMute(false), mSharedBuffer(sharedBuffer), mName(-1)

{

// mCblk !=NULL?什么时候创建的？？

//只能看基类TrackBase，还是很愤怒，太多继承了。

    if (mCblk != NULL) {

       mVolume[0] = 1.0f;

        mVolume[1] = 1.0f;

        mStreamType = streamType;

         mCblk->frameSize = AudioSystem::isLinearPCM(format) ? channelCount \*

 sizeof(int16\_t) : sizeof(int8\_t);

    }

}

看看基类TrackBase干嘛了

AudioFlinger::ThreadBase::TrackBase::TrackBase(

            const wp<ThreadBase>& thread,

            const sp<Client>& client,

            uint32\_t sampleRate,

            int format,

            int channelCount,

            int frameCount,

            uint32\_t flags,

            const sp<IMemory>& sharedBuffer)

    :   RefBase(),

        mThread(thread),

        mClient(client),

        mCblk(0),

        mFrameCount(0),

        mState(IDLE),

        mClientTid(-1),

        mFormat(format),

        mFlags(flags & ~SYSTEM\_FLAGS\_MASK)

{

    size\_t size = sizeof(audio\_track\_cblk\_t);

   size\_t bufferSize = frameCount\*channelCount\*sizeof(int16\_t);

   if (sharedBuffer == 0) {

       size += bufferSize;

   }

//调用client的allocate函数。这个client是什么？就是我们在CreateTrack中创建的

那个Client，我不想再说了。反正这里会创建一块共享内存

    mCblkMemory = client->heap()->allocate(size);

  有了共享内存，但是还没有里边有同步锁的那个对象audio\_track\_cblk\_t

     mCblk = static\_cast<audio\_track\_cblk\_t \*>(mCblkMemory->pointer());

     下面这个语法好怪啊。什么意思？？？

new(mCblk) audio\_track\_cblk\_t();

  //各位，这就是C++语法中的placement new。干啥用的啊?new后面的括号中是一块buffer，再

后面是一个类的构造函数。对了，这个placement new的意思就是在这块buffer中构造一个对象。

我们之前的普通new是没法让一个对象在某块指定的内存中创建的。而placement new却可以。

这样不就达到我们的目的了吗？搞一块共享内存，再在这块内存上创建一个对象。这样，这个对象不也就能在两个内存中共享了吗？太牛了。怎么想到的？

       // clear all buffers

       mCblk->frameCount = frameCount;

       mCblk->sampleRate = sampleRate;

       mCblk->channels = (uint8\_t)channelCount;

}

好了，解决一个重大疑惑，跨进程数据共享的重要数据结构audio\_track\_cblk\_t是通过placement new在一块共享内存上来创建的。

回到AF的CreateTrack，有这么一句话：

trackHandle = new TrackHandle(track);

return trackHandle；----》注意，这个对象是最终返回到AT进程中的。

trackHandle的构造使用了thread->createTrack\_l的返回值。

#### 4.3.3.4到底有多少种对象

读到这里的人，一定会被异常多的class类型，内部类，继承关系搞疯掉。说实话，这里废点心血整个或者paste一个大的UML图未尝不可。但是我是不太习惯用图说话，因为图我实在是记不住。那好吧。我们就用最简单的话语争取把目前出现的对象说清楚。

1 AudioFlinger

class AudioFlinger : public BnAudioFlinger, public IBinder::DeathRecipient

AudioFlinger类是代表整个AudioFlinger服务的类，其余所有的工作类都是通过内部类的方式在其中定义的。你把它当做一个壳子也行吧。

2 Client

Client是描述C/S结构的C端的代表，也就算是一个AT在AF端的对等物吧。不过可不是Binder机制中的BpXXX喔。因为AF是用不到AT的功能的。

class Client : public RefBase {

    public:

        sp<AudioFlinger>    mAudioFlinger;//代表S端的AudioFlinger

        sp<MemoryDealer>    mMemoryDealer;//每个C端使用的共享内存，通过它分配

        pid\_t               mPid;//C端的进程id

    };

3 TrackHandle

Trackhandle是AT端调用AF的CreateTrack得到的一个基于Binder机制的Track。

这个TrackHandle实际上是对真正干活的PlaybackThread::Track的一个跨进程支持的封装。

什么意思？本来PlaybackThread::Track是真正在AF中干活的东西，不过为了支持跨进程的话，我们用TrackHandle对其进行了一下包转。这样在AudioTrack调用TrackHandle的功能，实际都由TrackHandle调用PlaybackThread::Track来完成了。可以认为是一种Proxy模式吧。

这个就是AudioFlinger异常复杂的一个原因！！！

class TrackHandle : public android::BnAudioTrack {

    public:

                            TrackHandle(const sp<PlaybackThread::Track>& track);

        virtual             ~TrackHandle();

        virtual status\_t    start();

        virtual void        stop();

        virtual void        flush();

        virtual void        mute(bool);

        virtual void        pause();

        virtual void        setVolume(float left, float right);

        virtual sp<IMemory> getCblk() const;

        sp<PlaybackThread::Track> mTrack;

};

4 线程类

AF中有好几种不同类型的线程，分别有对应的线程类型：

         RecordThread：

RecordThread : public ThreadBase, public AudioBufferProvider

用于录音的线程。

         PlaybackThread:

class PlaybackThread : public ThreadBase

用于播放的线程

         MixerThread

MixerThread : public PlaybackThread

用于混音的线程，注意他是从PlaybackThread派生下来的。

         DirectoutputThread

DirectOutputThread : public PlaybackThread

直接输出线程，我们之前在代码里老看到DIRECT\_OUTPUT之类的判断，看来最终和这个线程有关。

         DuplicatingThread：

DuplicatingThread : public MixerThread

复制线程？而且从混音线程中派生？暂时不知道有什么用

这么多线程，都有一个共同的父类ThreadBase，这个是AF对Audio系统单独定义的一个以Thread为基类的类。------》FT，真的很麻烦。

ThreadBase我们不说了，反正里边封装了一些有用的函数。

我们看看PlayingThread吧，里边由定义了内部类：

5  PlayingThread的内部类Track

我们知道，TrackHandle构造用的那个Track是PlayingThread的createTrack\_l得到的。

class Track : public TrackBase

晕喔，又来一个TrackBase。

TrackBase是ThreadBase定义的内部类

class TrackBase : public AudioBufferProvider, public RefBase

基类AudioBufferProvider是一个对Buffer的封装，以后在AF读共享缓冲，写数据到硬件HAL中用得到。

个人感觉：上面这些东西，其实完完全全可以独立到不同的文件中，然后加一些注释说明。

写这样的代码，要是我是BOSS的话，一定会很不爽。有什么意义吗？有什么好处吗？

#### 4.3.3.5 AF流程继续

好了，这里终于在AF中的createTrack返回了TrackHandle。这个时候系统处于什么状态？

         AF中的几个Thread我们之前说了，在AF启动的某个时间就已经起来了。我们就假设AT调用AF服务前，这个线程就已经启动了。

这个可以看代码就知道了：

void AudioFlinger::PlaybackThread::onFirstRef()

{

    const size\_t SIZE = 256;

    char buffer[SIZE];

    snprintf(buffer, SIZE, "Playback Thread %p", this);

//onFirstRef，实际是RefBase的一个方法，在构造sp的时候就会被调用

//下面的run就真正创建了线程并开始执行threadLoop了

    run(buffer, ANDROID\_PRIORITY\_URGENT\_AUDIO);

}

到底执行哪个线程的threadLoop？我记得我们是根据output句柄来查找线程的。

看看openOutput的实行，真正的线程对象创建是在那儿。

int AudioFlinger::openOutput(uint32\_t \*pDevices,

                                uint32\_t \*pSamplingRate,

                                uint32\_t \*pFormat,

                                uint32\_t \*pChannels,

                                uint32\_t \*pLatencyMs,

                                uint32\_t flags)

{

        if ((flags & AudioSystem::OUTPUT\_FLAG\_DIRECT) ||

            (format != AudioSystem::PCM\_16\_BIT) ||

            (channels != AudioSystem::CHANNEL\_OUT\_STEREO)) {

            thread = new DirectOutputThread(this, output, ++mNextThreadId);

//如果flags没有设置直接输出标准，或者format不是16bit，或者声道数不是2立体声

//则创建DirectOutputThread。

} else {

    //可惜啊，我们创建的是最复杂的MixerThread

 thread = new MixerThread(this, output, ++mNextThreadId);

1. MixerThread

非常重要的工作线程，我们看看它的构造函数。

AudioFlinger::MixerThread::MixerThread(const sp<AudioFlinger>& audioFlinger, AudioStreamOut\* output, int id)

    :   PlaybackThread(audioFlinger, output, id),

        mAudioMixer(0)

{

mType = PlaybackThread::MIXER;

//混音器对象，传进去的两个参数时基类ThreadBase的，都为0

//这个对象巨复杂，最终混音的数据都由它生成，以后再说...

    mAudioMixer = new AudioMixer(mFrameCount, mSampleRate);

   }

2. AT调用start

此时，AT得到IAudioTrack对象后，调用start函数。

status\_t AudioFlinger::TrackHandle::start() {

    return mTrack->start();

} //果然，自己又不干活，交给mTrack了，这个是PlayintThread createTrack\_l得到的Track对象

status\_t AudioFlinger::PlaybackThread::Track::start()

{

    status\_t status = NO\_ERROR;

sp<ThreadBase> thread = mThread.promote();

//这个Thread就是调用createTrack\_l的那个thread对象，这里是MixerThread

    if (thread != 0) {

        Mutex::Autolock \_l(thread->mLock);

        int state = mState;

         if (mState == PAUSED) {

            mState = TrackBase::RESUMING;

           } else {

            mState = TrackBase::ACTIVE;

        }

  //把自己由加到addTrack\_l了

//奇怪，我们之前在看createTrack\_l的时候，不是已经有个map保存创建的track了

//这里怎么又出现了一个类似的操作？

        PlaybackThread \*playbackThread = (PlaybackThread \*)thread.get();

        playbackThread->addTrack\_l(this);

    return status;

}

看看这个addTrack\_l函数

status\_t AudioFlinger::PlaybackThread::addTrack\_l(const sp<Track>& track)

{

    status\_t status = ALREADY\_EXISTS;

    // set retry count for buffer fill

    track->mRetryCount = kMaxTrackStartupRetries;

    if (mActiveTracks.indexOf(track) < 0) {

        mActiveTracks.add(track);//啊，原来是加入到活跃Track的数组啊

        status = NO\_ERROR;

}

//我靠，有戏啊！看到这个broadcast，一定要想到：恩，在不远处有那么一个线程正

//等着这个CV呢。

    mWaitWorkCV.broadcast();

   return status;

}

让我们想想吧。start是把某个track加入到PlayingThread的活跃Track队列，然后触发一个信号事件。由于这个事件是PlayingThread的内部成员变量，而PlayingThread又创建了一个线程，那么难道是那个线程在等待这个事件吗？这时候有一个活跃track，那个线程应该可以干活了吧？

这个线程是MixerThread。我们去看看它的线程函数threadLoop吧。

bool AudioFlinger::MixerThread::threadLoop()

{

    int16\_t\* curBuf = mMixBuffer;

    Vector< sp<Track> > tracksToRemove;

 while (!exitPending())

    {

        processConfigEvents();

//Mixer进到这个循环中来

        mixerStatus = MIXER\_IDLE;

        { // scope for mLock

           Mutex::Autolock \_l(mLock);

            const SortedVector< wp<Track> >& activeTracks = mActiveTracks;

//每次都取当前最新的活跃Track数组

//下面是预备操作，返回状态看看是否有数据需要获取

mixerStatus = prepareTracks\_l(activeTracks, &tracksToRemove);

       }

//LIKELY，是GCC的一个东西，可以优化编译后的代码

//就当做是TRUE吧

if (LIKELY(mixerStatus == MIXER\_TRACKS\_READY)) {

            // mix buffers...

//调用混音器，把buf传进去，估计得到了混音后的数据了

//curBuf是mMixBuffer，PlayingThread的内部buffer，在某个地方已经创建好了，

//缓存足够大

            mAudioMixer->process(curBuf);

            sleepTime = 0;

            standbyTime = systemTime() + kStandbyTimeInNsecs;

        }

有数据要写到硬件中，肯定不能sleep了呀

if (sleepTime == 0) {

           //把缓存的数据写到outPut中。这个mOutput是AudioStreamOut

//由Audio HAL的那个对象创建得到。等我们以后分析再说

           int bytesWritten = (int)mOutput->write(curBuf, mixBufferSize);

            mStandby = false;

        } else {

            usleep(sleepTime);//如果没有数据，那就休息吧..

        }

3. MixerThread核心

到这里，大家是不是有种焕然一新的感觉？恩，对了，AF的工作就是如此的精密，每个部分都配合得丝丝入扣。不过对于我们看代码的人来说，实在搞不懂这么做的好处----哈哈 有点扯远了。

MixerThread的线程循环中，最重要的两个函数：

prepare\_l和mAudioMixer->process，我们一一来看看。

uint32\_t AudioFlinger::MixerThread::prepareTracks\_l(const SortedVector< wp<Track> >& activeTracks, Vector< sp<Track> > \*tracksToRemove)

{

    uint32\_t mixerStatus = MIXER\_IDLE;

    //得到活跃track个数，这里假设就是我们创建的那个AT吧，那么count=1

    size\_t count = activeTracks.size();

    float masterVolume = mMasterVolume;

    bool  masterMute = mMasterMute;

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

        sp<Track> t = activeTracks[i].promote();

      Track\* const track = t.get()；

   //得到placement new分配的那个跨进程共享的对象

        audio\_track\_cblk\_t\* cblk = track->cblk();

//设置混音器，当前活跃的track。

        mAudioMixer->setActiveTrack(track->name());

        if (cblk->framesReady() && (track->isReady() || track->isStopped()) &&

                !track->isPaused() && !track->isTerminated())

        {

            // compute volume for this track

//AT已经write数据了。所以肯定会进到这来。

            int16\_t left, right;

            if (track->isMuted() || masterMute || track->isPausing() ||

                mStreamTypes[track->type()].mute) {

                left = right = 0;

                if (track->isPausing()) {

                    track->setPaused();

                }

//AT设置的音量假设不为零，我们需要聆听声音！

//所以走else流程

            } else {

                // read original volumes with volume control

                float typeVolume = mStreamTypes[track->type()].volume;

                float v = masterVolume \* typeVolume;

                float v\_clamped = v \* cblk->volume[0];

                if (v\_clamped > MAX\_GAIN) v\_clamped = MAX\_GAIN;

                left = int16\_t(v\_clamped);

                v\_clamped = v \* cblk->volume[1];

                if (v\_clamped > MAX\_GAIN) v\_clamped = MAX\_GAIN;

                right = int16\_t(v\_clamped);

//计算音量

            }

//注意，这里对混音器设置了数据提供来源，是一个track，还记得我们前面说的吗？Track从

AudioBufferProvider派生

          mAudioMixer->setBufferProvider(track);

            mAudioMixer->enable(AudioMixer::MIXING);

            int param = AudioMixer::VOLUME;

           //为这个track设置左右音量等

          mAudioMixer->setParameter(param, AudioMixer::VOLUME0, left);

            mAudioMixer->setParameter(param, AudioMixer::VOLUME1, right);

            mAudioMixer->setParameter(

                AudioMixer::TRACK,

                AudioMixer::FORMAT, track->format());

            mAudioMixer->setParameter(

                AudioMixer::TRACK,

                AudioMixer::CHANNEL\_COUNT, track->channelCount());

            mAudioMixer->setParameter(

                AudioMixer::RESAMPLE,

                AudioMixer::SAMPLE\_RATE,

                int(cblk->sampleRate));

        } else {

           if (track->isStopped()) {

                track->reset();

            }

  //如果这个track已经停止了，那么把它加到需要移除的track队列tracksToRemove中去

//同时停止它在AudioMixer中的混音

            if (track->isTerminated() || track->isStopped() || track->isPaused()) {

                tracksToRemove->add(track);

                mAudioMixer->disable(AudioMixer::MIXING);

            } else {

                mAudioMixer->disable(AudioMixer::MIXING);

            }

        }

    }

    // remove all the tracks that need to be...

    count = tracksToRemove->size();

    return mixerStatus;

}

看明白了吗？prepare\_l的功能是什么？根据当前活跃的track队列，来为混音器设置信息。可想而知，一个track必然在混音器中有一个对应的东西。我们待会分析AudioMixer的时候再详述。

为混音器准备好后，下面调用它的process函数:

void AudioMixer::process(void\* output)

{

    mState.hook(&mState, output);//hook？难道是钩子函数？

}

晕乎，就这么简单的函数？？？

CTRL+左键，hook是一个函数指针啊，在哪里赋值的？具体实现函数又是哪个？

没办法了，只能分析AudioMixer类了。

4. AudioMixer

AudioMixer实现在framework/base/libs/audioflinger/AudioMixer.cpp中

AudioMixer::AudioMixer(size\_t frameCount, uint32\_t sampleRate)

    :   mActiveTrack(0), mTrackNames(0), mSampleRate(sampleRate)

{

    mState.enabledTracks= 0;

    mState.needsChanged = 0;

    mState.frameCount   = frameCount;

    mState.outputTemp   = 0;

    mState.resampleTemp = 0;

    mState.hook         = process\_\_nop;//process\_\_nop，是该类的静态函数

track\_t\* t = mState.tracks;

//支持32路混音。牛死了

    for (int i=0 ; i<32 ; i++) {

        t->needs = 0;

        t->volume[0] = UNITY\_GAIN;

        t->volume[1] = UNITY\_GAIN;

        t->volumeInc[0] = 0;

        t->volumeInc[1] = 0;

        t->channelCount = 2;

        t->enabled = 0;

        t->format = 16;

        t->buffer.raw = 0;

        t->bufferProvider = 0;

        t->hook = 0;

        t->resampler = 0;

        t->sampleRate = mSampleRate;

        t->in = 0;

        t++;

    }

}

//其中，mState是在AudioMixer.h中定义的一个数据结构

//注意，source insight没办法解析这个mState，因为....见下面的注释。

struct state\_t {

        uint32\_t        enabledTracks;

        uint32\_t        needsChanged;

        size\_t          frameCount;

        mix\_t           hook;

        int32\_t         \*outputTemp;

        int32\_t         \*resampleTemp;

        int32\_t         reserved[2];

        track\_t         tracks[32];// \_\_attribute\_\_((aligned(32)));《--把这里注释掉

//否则source insight会解析不了这个state\_t类型

    };

    int             mActiveTrack;

    uint32\_t        mTrackNames;//names？搞得像字符串，实际是一个int

    const uint32\_t  mSampleRate;

state\_t         mState

好了，没什么吗。hook对应的可选函数实现有：

process\_\_validate

process\_\_nop

process\_\_genericNoResampling

process\_\_genericResampling

process\_\_OneTrack16BitsStereoNoResampling

process\_\_TwoTracks16BitsStereoNoResampling

AudioMixer构造的时候，hook是process\_\_nop，有几个地方会改变这个函数指针的指向。

这部分涉及到数字音频技术，我就无力讲解了。我们看看最接近的函数

process\_\_OneTrack16BitsStereoNoResampling

void AudioMixer::process\_\_OneTrack16BitsStereoNoResampling(state\_t\* state, void\* output)

{

单track，16bit双声道，不需要重采样,大部分是这种情况了

    const int i = 31 - \_\_builtin\_clz(state->enabledTracks);

    const track\_t& t = state->tracks[i];

    AudioBufferProvider::Buffer& b(t.buffer);

    int32\_t\* out = static\_cast<int32\_t\*>(output);

    size\_t numFrames = state->frameCount;

    const int16\_t vl = t.volume[0];

    const int16\_t vr = t.volume[1];

    const uint32\_t vrl = t.volumeRL;

    while (numFrames) {

        b.frameCount = numFrames;

//获得buffer

        t.bufferProvider->getNextBuffer(&b);

        int16\_t const \*in = b.i16;

       size\_t outFrames = b.frameCount;

       if  UNLIKELY--->不走这.

        else {

            do {

          //计算音量等数据，和数字音频技术有关。这里不说了

                uint32\_t rl = \*reinterpret\_cast<uint32\_t const \*>(in);

                in += 2;

                int32\_t l = mulRL(1, rl, vrl) >> 12;

                int32\_t r = mulRL(0, rl, vrl) >> 12;

                \*out++ = (r<<16) | (l & 0xFFFF);

            } while (--outFrames);

        }

        numFrames -= b.frameCount;

//释放buffer。

        t.bufferProvider->releaseBuffer(&b);

    }

}

好像挺简单的啊，不就是把数据处理下嘛。这里注意下buffer。到现在，我们还没看到取共享内存里AT端write的数据呐。

那只能到bufferProvider去看了。

注意，这里用的是AudioBufferProvider基类，实际的对象是Track。它从AudioBufferProvider派生。

我们用得是PlaybackThread的这个Track

status\_t AudioFlinger::PlaybackThread::Track::getNextBuffer(AudioBufferProvider::Buffer\* buffer)

{

//一阵暗喜吧。千呼万唤始出来，终于见到cblk了

     audio\_track\_cblk\_t\* cblk = this->cblk();

     uint32\_t framesReady;

     uint32\_t framesReq = buffer->frameCount;

 //哈哈，看看数据准备好了没，

      framesReady = cblk->framesReady();

     if (LIKELY(framesReady)) {

        uint32\_t s = cblk->server;

        uint32\_t bufferEnd = cblk->serverBase + cblk->frameCount;

        bufferEnd = (cblk->loopEnd < bufferEnd) ? cblk->loopEnd : bufferEnd;

        if (framesReq > framesReady) {

            framesReq = framesReady;

        }

        if (s + framesReq > bufferEnd) {

            framesReq = bufferEnd - s;

        }

获得真实的数据地址

         buffer->raw = getBuffer(s, framesReq);

         if (buffer->raw == 0) goto getNextBuffer\_exit;

         buffer->frameCount = framesReq;

        return NO\_ERROR;

     }

getNextBuffer\_exit:

     buffer->raw = 0;

     buffer->frameCount = 0;

    return NOT\_ENOUGH\_DATA;

}

再看看释放缓冲的地方：releaseBuffer，这个直接在ThreadBase中实现了

void AudioFlinger::ThreadBase::TrackBase::releaseBuffer(AudioBufferProvider::Buffer\* buffer)

{

    buffer->raw = 0;

    mFrameCount = buffer->frameCount;

    step();

    buffer->frameCount = 0;

}

看看step吧。mFrameCount表示我已经用完了这么多帧。

bool AudioFlinger::ThreadBase::TrackBase::step() {

    bool result;

    audio\_track\_cblk\_t\* cblk = this->cblk();

result = cblk->stepServer(mFrameCount);//哼哼，调用cblk的stepServer，更新

服务端的使用位置

    return result;

}

到这里，大伙应该都明白了吧。原来AudioTrack中write的数据，最终是这么被使用的呀！！！

恩，看一个process\_\_OneTrack16BitsStereoNoResampling不过瘾，再看看

process\_\_TwoTracks16BitsStereoNoResampling。

void AudioMixer::process\_\_TwoTracks16BitsStereoNoResampling(state\_t\* state, void\*

output)

int i;

    uint32\_t en = state->enabledTracks;

    i = 31 - \_\_builtin\_clz(en);

    const track\_t& t0 = state->tracks[i];

    AudioBufferProvider::Buffer& b0(t0.buffer);

    en &= ~(1<<i);

    i = 31 - \_\_builtin\_clz(en);

    const track\_t& t1 = state->tracks[i];

    AudioBufferProvider::Buffer& b1(t1.buffer);

    int16\_t const \*in0;

    const int16\_t vl0 = t0.volume[0];

    const int16\_t vr0 = t0.volume[1];

    size\_t frameCount0 = 0;

    int16\_t const \*in1;

    const int16\_t vl1 = t1.volume[0];

    const int16\_t vr1 = t1.volume[1];

    size\_t frameCount1 = 0;

    int32\_t\* out = static\_cast<int32\_t\*>(output);

    size\_t numFrames = state->frameCount;

    int16\_t const \*buff = NULL;

    while (numFrames) {

        if (frameCount0 == 0) {

            b0.frameCount = numFrames;

            t0.bufferProvider->getNextBuffer(&b0);

            if (b0.i16 == NULL) {

                if (buff == NULL) {

                    buff = new int16\_t[MAX\_NUM\_CHANNELS \* state->frameCount];

                }

                in0 = buff;

                b0.frameCount = numFrames;

            } else {

                in0 = b0.i16;

            }

            frameCount0 = b0.frameCount;

        }

        if (frameCount1 == 0) {

            b1.frameCount = numFrames;

            t1.bufferProvider->getNextBuffer(&b1);

            if (b1.i16 == NULL) {

                if (buff == NULL) {

                    buff = new int16\_t[MAX\_NUM\_CHANNELS \* state->frameCount];

                }

                in1 = buff;

                b1.frameCount = numFrames;

               } else {

                in1 = b1.i16;

            }

            frameCount1 = b1.frameCount;

        }

        size\_t outFrames = frameCount0 < frameCount1?frameCount0:frameCount1;

        numFrames -= outFrames;

        frameCount0 -= outFrames;

        frameCount1 -= outFrames;

        do {

            int32\_t l0 = \*in0++;

            int32\_t r0 = \*in0++;

            l0 = mul(l0, vl0);

            r0 = mul(r0, vr0);

            int32\_t l = \*in1++;

            int32\_t r = \*in1++;

            l = mulAdd(l, vl1, l0) >> 12;

            r = mulAdd(r, vr1, r0) >> 12;

            // clamping...

            l = clamp16(l);

            r = clamp16(r);

            \*out++ = (r<<16) | (l & 0xFFFF);

        } while (--outFrames);

        if (frameCount0 == 0) {

            t0.bufferProvider->releaseBuffer(&b0);

        }

        if (frameCount1 == 0) {

            t1.bufferProvider->releaseBuffer(&b1);

        }

    }

    if (buff != NULL) {

        delete [] buff;

    }

}

看不懂了吧？？哈哈，知道有这回事就行了，专门搞数字音频的需要好好研究下了！

#### 4.3.3.6 再论共享audio\_track\_cblk\_t

为什么要再论这个？因为我在网上找了下，有人说audio\_track\_cblk\_t是一个环形buffer，环形buffer是什么意思？自己查查！

这个吗，和我之前的工作经历有关系，某BOSS费尽心机想搞一个牛掰掰的环形buffer，搞得我累死了。现在audio\_track\_cblk\_t是环形buffer？我倒是想看看它是怎么实现的。

顺便我们要解释下，audio\_track\_cblk\_t的使用和我之前说的Lock,读/写，Unlock不太一样。为何？

         第一因为我们没在AF代码中看到有缓冲buffer方面的wait，MixThread只有当没有数据的时候会usleep一下。

         第二，如果有多个track，多个audio\_track\_cblk\_t的话，假如又是采用wait信号的办法，那么由于pthread库缺乏WaitForMultiObjects的机制，那么到底该等哪一个？这个问题是我们之前在做跨平台同步库的一个重要难题。

1. 写者的使用

我们集中到audio\_track\_cblk\_t这个类，来看看写者是如何使用的。写者就是AudioTrack端，在这个类中，叫user

         framesAvailable，看看是否有空余空间

         buffer，获得写空间起始地址

         stepUser，更新user的位置。

2. 读者的使用

读者是AF端，在这个类中加server。

         framesReady，获得可读的位置

         stepServer，更新读者的位置

看看这个类的定义：

struct audio\_track\_cblk\_t

{

               Mutex       lock; //同步锁

                Condition   cv;//CV

volatile    uint32\_t    user;//写者

    volatile    uint32\_t    server;//读者

                uint32\_t    userBase;//写者起始位置

                uint32\_t    serverBase;//读者起始位置

    void\*       buffers;

    uint32\_t    frameCount;

    // Cache line boundary

    uint32\_t    loopStart; //循环起始

    uint32\_t    loopEnd; //循环结束

    int         loopCount;

uint8\_t     out;   //如果是Track的话，out就是1，表示输出。

}

注意这是volatile，跨进程的对象，看来这个volatile也是可以跨进程的嘛。

         唉，又要发挥下了。volatile只是告诉编译器，这个单元的地址不要cache到CPU的缓冲中。也就是每次取值的时候都要到实际内存中去读，而且可能读内存的时候先要锁一下总线。防止其他CPU核执行的时候同时去修改。由于是跨进程共享的内存，这块内存在两个进程都是能见到的，又锁总线了，又是同一块内存，volatile当然保证了同步一致性。

         loopStart和loopEnd这两个值是表示循环播放的起点和终点的，下面还有一个loopCount吗，表示循环播放次数的

那就分析下吧。

先看写者的那几个函数

4 写者分析

先用frameavail看看当前剩余多少空间，我们可以假设是第一次进来嘛。读者还在那sleep呢。

uint32\_t audio\_track\_cblk\_t::framesAvailable()

{

    Mutex::Autolock \_l(lock);

    return framesAvailable\_l();

}

int32\_t audio\_track\_cblk\_t::framesAvailable\_l()

{

    uint32\_t u = this->user; 当前写者位置，此时也为0

    uint32\_t s = this->server; //当前读者位置，此时为0

    if (out) { out为1

        uint32\_t limit = (s < loopStart) ? s : loopStart;

我们不设循环播放时间吗。所以loopStart是初始值INT\_MAX，所以limit=0

        return limit + frameCount - u;

//返回0+frameCount-0，也就是全缓冲最大的空间。假设frameCount=1024帧

    }

}

然后调用buffer获得其实位置，buffer就是得到一个地址位置。

void\* audio\_track\_cblk\_t::buffer(uint32\_t offset) const

{

    return (int8\_t \*)this->buffers + (offset - userBase) \* this->frameSize;

}

完了，我们更新写者，调用stepUser

uint32\_t audio\_track\_cblk\_t::stepUser(uint32\_t frameCount)

{

//framecount，表示我写了多少，假设这一次写了512帧

    uint32\_t u = this->user;//user位置还没更新呢，此时u=0；

    u += frameCount;//u更新了，u=512

    // Ensure that user is never ahead of server for AudioRecord

    if (out) {

       //没甚，计算下等待时间

}

//userBase还是初始值为0，可惜啊，我们只写了1024的一半

//所以userBase加不了

   if (u >= userBase + this->frameCount) {

        userBase += this->frameCount;

//但是这句话很重要，userBase也更新了。根据buffer函数的实现来看，似乎把这个

//环形缓冲铺直了....连绵不绝。

    }

    this->user = u;//喔，user位置也更新为512了，但是useBase还是0

    return u;

}

好了，假设写者这个时候sleep了，而读者起来了。

5 读者分析

uint32\_t audio\_track\_cblk\_t::framesReady()

{

    uint32\_t u = this->user; //u为512

    uint32\_t s = this->server;//还没读呢，s为零

    if (out) {

        if (u < loopEnd) {

            return u - s;//loopEnd也是INT\_MAX，所以这里返回512，表示有512帧可读了

        } else {

            Mutex::Autolock \_l(lock);

            if (loopCount >= 0) {

                return (loopEnd - loopStart)\*loopCount + u - s;

            } else {

                return UINT\_MAX;

            }

        }

    } else {

        return s - u;

    }

}

使用完了，然后stepServer

bool audio\_track\_cblk\_t::stepServer(uint32\_t frameCount)

{

    status\_t err;

   err = lock.tryLock();

    uint32\_t s = this->server;

    s += frameCount; //读了512帧了，所以s=512

    if (out) {

    }

   没有设置循环播放嘛，所以不走这个

    if (s >= loopEnd) {

       s = loopStart;

        if (--loopCount == 0) {

            loopEnd = UINT\_MAX;

            loopStart = UINT\_MAX;

        }

}

//一样啊，把环形缓冲铺直了

    if (s >= serverBase + this->frameCount) {

        serverBase += this->frameCount;

    }

    this->server = s; //server为512了

    cv.signal(); //读者读完了。触发下写者吧。

    lock.unlock();

    return true;

}

6 真的是环形缓冲吗？

环形缓冲是这样一个场景，现在buffer共1024帧。

假设：

         写者先写到1024帧

         读者读到512帧

         那么，写者还可以从头写512帧。

所以，我们得回头看看frameavail是不是把这512帧算进来了。

uint32\_t audio\_track\_cblk\_t::framesAvailable\_l()

{

    uint32\_t u = this->user;  //1024

    uint32\_t s = this->server;//512

    if (out) {

        uint32\_t limit = (s < loopStart) ? s : loopStart;

        return limit + frameCount - u;返回512，用上了！

    }

}

再看看stepUser这句话

if (u >= userBase + this->frameCount) {u为1024，userBase为0，frameCount为1024

        userBase += this->frameCount;//好，userBase也为1024了

}

看看buffer

return (int8\_t \*)this->buffers + (offset - userBase) \* this->frameSize;

//offset是外界传入的基于user的一个偏移量。offset-userBase，得到的正式从头开始的那段数据空间。太牛了！

### 4.3.4 Audio Policy

上回我们说了AudioFlinger(AF)，总感觉代码里边有好多东西没说清楚，心里发毛。就看了看AF的流程，我们敢说自己深入了解了Android系统吗？AudioPolicyService（APS）是个什么东西？为什么要有它的存在？下层的Audio HAL层又是怎么结合到Android中来的？更有甚者，问个实在问题：插入耳机后，声音又怎么从最开始的外放变成从耳机输出了？调节音量的时候到底是调节Music的还是调节来电音量呢？这些东西，我们在AF的流程中统统都没讲到。但是这些他们又是至关重要的。从我个人理解来看，策略（Policy）比流程更复杂和难懂。

当然，遵循我们的传统分析习惯，得有一个切入点，否则我们都不知道从何入手了。

这里的切入点将是：

         AF和APS系统第一次起来后，到底干了什么。

         检测到耳机插入事件后，AF和APS的处理。

大家跟着我一步步来看，很快就发现，啊哈，APS也不是那么难嘛。

另外，这次代码分析的格式将参考《Linux内核情景分析》的样子，函数调用的解析将采用深度优先的办法，即先解释所调用的函数，然后再出来继续讲。

#### 4.3.4.1 AF和APS的诞生

这个东西，已经说得太多了。在framework/base/media/MediaServer/Main\_MediaServer中。

我们看看。

int main(int argc, char\*\* argv)

{

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

    sp<IServiceManager> sm = defaultServiceManager();

    //先创建AF

AudioFlinger::instantiate();

    //再创建APS

AudioPolicyService::instantiate();

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

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

}

##### 4.3.4.1.1 new AudioFringer

前面说过，instantiate内部会实例化一个对象，那直接看AF的构造函数。

AudioFlinger::AudioFlinger()

    : BnAudioFlinger(),//基类构造函数

        mAudioHardware(0), mMasterVolume(1.0f), mMasterMute(false), mNextThreadId(0)

{

  注意mAudioHardware和mNextThreadId

 mHardwareStatus = AUDIO\_HW\_IDLE;

//创建audio的HAL代表

    mAudioHardware = AudioHardwareInterface::create();

mHardwareStatus = AUDIO\_HW\_INIT;

//下面这些不至于会使用APS吧？APS还没创建呢！

    if (mAudioHardware->initCheck() == NO\_ERROR) {

        setMode(AudioSystem::MODE\_NORMAL);

        setMasterVolume(1.0f);

        setMasterMute(false);

    }

}

感觉上，AF的构造函数就是创建了一个最重要的AudioHardWare的HAL代表。

其他好像是没干什么策略上的事情。

不过：AF创建了一个AudioHardware的HAL对象。注意整个系统就这一个AudioHardware了。也就是说，不管是线控耳机，蓝牙耳机，麦克，外放等等，最后都会由这一个HAL统一管理。

再看APS吧。

##### 4.3.4.1.2 new AudioPolicyService

AudioPolicyService::AudioPolicyService()

    : BnAudioPolicyService() , mpPolicyManager(NULL)

{

  //  mpPolicyManager?策略管理器?可能很重要

char value[PROPERTY\_VALUE\_MAX];

    // TonePlayback？播放铃声的？为什么放在这里？以后来看看

    mTonePlaybackThread = new AudioCommandThread(String8(""));

// Audio Command？音频命令？看到Command，我就想到设计模式中的Command模式了

//Android尤其是MediaPlayerService中大量使用了这种模式。

    mAudioCommandThread = new AudioCommandThread(String8("ApmCommandThread"));

#if (defined GENERIC\_AUDIO) || (defined AUDIO\_POLICY\_TEST)

 //注意AudioPolicyManagerBase的构造函数，把this传进去了。

    mpPolicyManager = new AudioPolicyManagerBase(this);

    //先假设我们使用Generic的Audio设备吧。

#else

    ...

#endif

// 根据系统属性来判断摄像机是否强制使用声音。这个...为什么会放在这里？

//手机带摄像机好像刚出来的时候，为了防止偷拍，强制按快门的时候必须发出声音

//就是这个目的吧？

    property\_get("ro.camera.sound.forced", value, "0");

mpPolicyManager->setSystemProperty("ro.camera.sound.forced", value);

}

so easy!，不至于吧？我们不应该放过任何一个疑问！这么多疑问，先看哪个呢？这里分析的是Audio Policy，而构造函数中又创建了一个AudioPolicyManagerBase，而且不同厂商还可以实现自己的AudioPolicyManager，看来这个对于音频策略有至关重要的作用了。

不得不说的是，Android代码中的这些命名在关键地方上还是比较慎重和准确的。

另外，AudioPolicyManagerBase的构造函数可是把APS传进去了，看来又会有一些回调靠APS了。真绕。

##### 4.3.4.1.3 AudioPolicyManagerBase

代码位置在framework/base/libs/audioflinger/AudioPolicyManagerBase.cpp中

AudioPolicyManagerBase::AudioPolicyManagerBase(AudioPolicyClientInterface \*clientInterface)

    :

 mPhoneState(AudioSystem::MODE\_NORMAL), ---->这里有电话的状态？

mRingerMode(0),

mMusicStopTime(0),

 mLimitRingtoneVolume(false)

{

[--->mPhoneState(AudioSystem::MODE\_NORMAL)]

   AudioSystem其实是窥视Android如何管理音频系统的好地方。位置在

framework/base/include/media/AudioSystem.h中，定义了大量的枚举之类的东西来表达Google对音频系统的看法。我们只能见招拆招了。

下面是audio\_mode的定义。这里要注意一个地方：

这些定义都和SDK中的JAVA层定义类似。实际上应该说先有C++层的定义，然后再反映到JAVA层中。但是C++层的定义一般没有解释说明，而SDK中有。所以我们不能不面对的一个痛苦现实就是：常常需要参考SDK的说明才能搞明白到底是什么。

关于C++的AudioSystem这块，SDK的说明在AudioManager中。

enum audio\_mode {

//解释参考SDK说明，以下不再说明

        MODE\_INVALID = -2, //无效mode

        MODE\_CURRENT = -1,//当前mode，和音频设备的切换（路由）有关

        MODE\_NORMAL = 0,//正常mode，没有电话和铃声

        MODE\_RINGTONE,//收到来电信号了，此时会有铃声

        MODE\_IN\_CALL,//电话mode，这里表示已经建立通话了

        NUM\_MODES  // Android大量采用这种技巧来表示枚举结束了。

    };

...

mPhoneState(AudioSystem::MODE\_NORMAL), ---->这里有电话的状态？

mRingerMode(0),

mMusicStopTime(0),

 mLimitRingtoneVolume(false)

{

mpClientInterface = clientInterface;//BT，保存APS对象。

//forceUse？这是个什么玩意儿？

    for (int i = 0; i < AudioSystem::NUM\_FORCE\_USE; i++) {

        mForceUse[i] = AudioSystem::FORCE\_NONE;

    }

[---->AudioSystem::FORCE\_NONE和AudioSystem::NUM\_FORCE\_USE]

注意，这里有两个枚举，太无耻了。先看看FORCE\_NONE这个

enum forced\_config {强制\_配置，看名字好像是强制使用设备吧，比如外放，耳机，蓝牙等

        FORCE\_NONE,

        FORCE\_SPEAKER,

        FORCE\_HEADPHONES,

        FORCE\_BT\_SCO,

        FORCE\_BT\_A2DP,

        FORCE\_WIRED\_ACCESSORY,

        FORCE\_BT\_CAR\_DOCK,

        FORCE\_BT\_DESK\_DOCK,

        NUM\_FORCE\_CONFIG,

        FORCE\_DEFAULT = FORCE\_NONE //这个，太无聊了。

};

再看看AudioSystem::NUM\_FORCE\_USE这个

enum force\_use {

        FOR\_COMMUNICATION,//这里是for\_xxx，不是force\_xxx。

        FOR\_MEDIA,

        FOR\_RECORD,

        FOR\_DOCK,

        NUM\_FORCE\_USE

    };

不懂，两个都不懂。为何？能猜出来什么吗？也不行。因为我们没找到合适的场景！那好吧，我们去SDK找找。恩

我看到AudioManager这个函数setSpeakerphoneOn (boolean on)。好吧，我

这么调用

setSpeakerphoneOn(true)，看看实现。

这次我没再浪费时间了，我用一个新的工具coolfind，把搜索framework目录，寻找\*.java文件，匹配字符串setSpeakerphone。终于，我在

framework/base/media/java/android/media/AudioService.java中找到了。

public void setSpeakerphoneOn(boolean on){

        if (!checkAudioSettingsPermission("setSpeakerphoneOn()")) {

            return;

        }

        if (on) {

//看到这里，是不是明白十之八九了?下面这个调用是：

//强制通话使用speaker！原来是这么个意思！

            AudioSystem.setForceUse(AudioSystem.FOR\_COMMUNICATION,

AudioSystem.FORCE\_SPEAKER);

            mForcedUseForComm = AudioSystem.FORCE\_SPEAKER;

        } else {

            AudioSystem.setForceUse(AudioSystem.FOR\_COMMUNICATION,

AudioSystem.FORCE\_NONE);

            mForcedUseForComm = AudioSystem.FORCE\_NONE;

        }

    }

好了，说点题外话，既然Android源码都放开给我们了，有什么理由我们不去多搜搜呢？上网google也是搜，查源代码也是一样吗。不过我们要有目的：就是找到一个合适的使用场景。

force\_use和force\_config就不用我再解释了吧？

[--->AudioPolicyManagerBase::AudioPolicyManagerBase]

...

//下面这个意思就是把几种for\_use的情况使用的设备全部置为NONE。

//比如设置FOR\_MEDIA的场景，使用的设备就是FORCE\_NONE

for (int i = 0; i < AudioSystem::NUM\_FORCE\_USE; i++) {

        mForceUse[i] = AudioSystem::FORCE\_NONE;

    }

  // 目前可以的输出设备，耳机和外放

    mAvailableOutputDevices = AudioSystem::DEVICE\_OUT\_EARPIECE |

                        AudioSystem::DEVICE\_OUT\_SPEAKER;

//目前可用的输入设备，内置MIC

    mAvailableInputDevices = AudioSystem::DEVICE\_IN\_BUILTIN\_MIC;

又得来看看AudioSystem是怎么定义输入输出设备的了。

[--->mAvailableOutputDevices = AudioSystem::DEVICE\_OUT\_EARPIECE]

enum audio\_devices {

        // output devices

        DEVICE\_OUT\_EARPIECE = 0x1,

        DEVICE\_OUT\_SPEAKER = 0x2,

        DEVICE\_OUT\_WIRED\_HEADSET = 0x4,

        DEVICE\_OUT\_WIRED\_HEADPHONE = 0x8,

        DEVICE\_OUT\_BLUETOOTH\_SCO = 0x10,

        DEVICE\_OUT\_BLUETOOTH\_SCO\_HEADSET = 0x20,

        DEVICE\_OUT\_BLUETOOTH\_SCO\_CARKIT = 0x40,

        DEVICE\_OUT\_BLUETOOTH\_A2DP = 0x80,

        DEVICE\_OUT\_BLUETOOTH\_A2DP\_HEADPHONES = 0x100,

        DEVICE\_OUT\_BLUETOOTH\_A2DP\_SPEAKER = 0x200,

        DEVICE\_OUT\_AUX\_DIGITAL = 0x400,

        DEVICE\_OUT\_DEFAULT = 0x8000,

        DEVICE\_OUT\_ALL = (DEVICE\_OUT\_EARPIECE | DEVICE\_OUT\_SPEAKER |

 DEVICE\_OUT\_WIRED\_HEADSET | DEVICE\_OUT\_WIRED\_HEADPHONE | DEVICE\_OUT\_BLUETOOTH\_SCO | DEVICE\_OUT\_BLUETOOTH\_SCO\_HEADSET |DEVICE\_OUT\_BLUETOOTH\_SCO\_CARKIT |

 DEVICE\_OUT\_BLUETOOTH\_A2DP | DEVICE\_OUT\_BLUETOOTH\_A2DP\_HEADPHONES |

 DEVICE\_OUT\_BLUETOOTH\_A2DP\_SPEAKER | DEVICE\_OUT\_AUX\_DIGITAL | DEVICE\_OUT\_DEFAULT),

   DEVICE\_OUT\_ALL\_A2DP = (DEVICE\_OUT\_BLUETOOTH\_A2DP |

DEVICE\_OUT\_BLUETOOTH\_A2DP\_HEADPHONES |DEVICE\_OUT\_BLUETOOTH\_A2DP\_SPEAKER),

        // input devices

        DEVICE\_IN\_COMMUNICATION = 0x10000,

        DEVICE\_IN\_AMBIENT = 0x20000,

        DEVICE\_IN\_BUILTIN\_MIC = 0x40000,

        DEVICE\_IN\_BLUETOOTH\_SCO\_HEADSET = 0x80000,

        DEVICE\_IN\_WIRED\_HEADSET = 0x100000,

        DEVICE\_IN\_AUX\_DIGITAL = 0x200000,

        DEVICE\_IN\_VOICE\_CALL = 0x400000,

        DEVICE\_IN\_BACK\_MIC = 0x800000,

        DEVICE\_IN\_DEFAULT = 0x80000000,

        DEVICE\_IN\_ALL = (DEVICE\_IN\_COMMUNICATION | DEVICE\_IN\_AMBIENT |

DEVICE\_IN\_BUILTIN\_MIC |DEVICE\_IN\_BLUETOOTH\_SCO\_HEADSET | DEVICE\_IN\_WIRED\_HEADSET |

  DEVICE\_IN\_AUX\_DIGITAL | DEVICE\_IN\_VOICE\_CALL | DEVICE\_IN\_BACK\_MIC |

DEVICE\_IN\_DEFAULT)

    };

一些比较容易眼花的东西我标成红色的了。这么多东西，不过没什么我们不明白的了。

得嘞，继续走。

[--->AudioPolicyManagerBase::AudioPolicyManagerBase]

// 目前可以的输出设备，又有耳机又有外放，配置很强悍啊。

//注意这里是OR操作符，最终mAvailableOutputDevices = 0X3

    mAvailableOutputDevices = AudioSystem::DEVICE\_OUT\_EARPIECE |

                        AudioSystem::DEVICE\_OUT\_SPEAKER;

//目前可用的输入设备，内置MIC，mAvailableInputDevices为0x4000，不过我们不关注input

 mAvailableInputDevices = AudioSystem::DEVICE\_IN\_BUILTIN\_MIC;

 ...

  下面东西就很少了，我们一气呵成。

//创建一个AudioOutputDescriptor，并设置它的device为外设0x2

AudioOutputDescriptor \*outputDesc = new AudioOutputDescriptor();

outputDesc->mDevice = (uint32\_t)AudioSystem::DEVICE\_OUT\_SPEAKER;

//调用APS的openOutput，得到一个mHardwareOutput东东。这是个int型

//不过保不准是一个指针也不一定喔。

//而且，下面的参数都是指针类型（flags除外），难道？有人会改value吗？

    mHardwareOutput = mpClientInterface->openOutput(&outputDesc->mDevice,

                                    &outputDesc->mSamplingRate,

                                    &outputDesc->mFormat,

                                    &outputDesc->mChannels,

                                    &outputDesc->mLatency,

                                    outputDesc->mFlags);

  //这个...估计是把int和指针加入到一个map了，方便管理。

addOutput(mHardwareOutput, outputDesc);

//不知道干嘛，待会看。

setOutputDevice(mHardwareOutput, (uint32\_t)AudioSystem::DEVICE\_OUT\_SPEAKER, true);

//不知道干嘛，待会看。

    updateDeviceForStrategy();

好了，上面还有一系列函数，等着我们调用呢。我们一个一个看。

提前说一下，这块可是AudioManagerBase的核心喔。

[---->AudioOutputDescriptor \*outputDesc = new AudioOutputDescriptor()]

AudioOutputDescriptor是个什么？我不是神，我也得看注释。

**// descriptor for audio outputs. Used to maintain current configuration of each opened audio output**

**// and keep track of the usage of this output by each audio stream type.**

明白了么？大概意思就是它，是这么一个东西：

         描述audio输出的，可以用来保存一些配置信息。

         跟踪音频stream类型使用这个output的一些情况。

没明白吧？以后碰到场景就明白了。

它的构造函数干了如下勾当：

AudioPolicyManagerBase::AudioOutputDescriptor::AudioOutputDescriptor()

    : mId(0), mSamplingRate(0), mFormat(0), mChannels(0), mLatency(0),

    mFlags((AudioSystem::output\_flags)0), mDevice(0), mOutput1(0), mOutput2(0)

{}

//很好，统统都置零了。上面这些东西不用我解释了吧？命名规则也可以看出来。

[--->mHardwareOutput = mpClientInterface->openOutput()]:

这里调用的是APS的openOutput，看看去：

  [--->AudioPolicyService::openOutput]

audio\_io\_handle\_t AudioPolicyService::openOutput(uint32\_t \*pDevices,

                                uint32\_t \*pSamplingRate,

                                uint32\_t \*pFormat,

                                uint32\_t \*pChannels,

                                uint32\_t \*pLatencyMs,

                                AudioSystem::output\_flags flags)

{

sp<IAudioFlinger> af = AudioSystem::get\_audio\_flinger();

//娘希匹，搞到AF去了

return af->openOutput(pDevices, pSamplingRate, (uint32\_t \*)pFormat, pChannels,

 pLatencyMs, flags);

}

[----->AudioFlinger::openOutput]

int AudioFlinger::openOutput(uint32\_t \*pDevices,

                                uint32\_t \*pSamplingRate,

                                uint32\_t \*pFormat,

                                uint32\_t \*pChannels,

                                uint32\_t \*pLatencyMs,

                                uint32\_t flags)

{

//我们思考下传进来的值吧

//\*pDevices=0x2,代表外放

//其他都是0。 嘿嘿，有了值，这不就知道下面该怎么走了吗？

    status\_t status;

    PlaybackThread \*thread = NULL;

    mHardwareStatus = AUDIO\_HW\_OUTPUT\_OPEN;

    uint32\_t samplingRate = pSamplingRate ? \*pSamplingRate : 0;

    uint32\_t format = pFormat ? \*pFormat : 0;

    uint32\_t channels = pChannels ? \*pChannels : 0;

    uint32\_t latency = pLatencyMs ? \*pLatencyMs : 0;

     Mutex::Autolock \_l(mLock);

//HAL对象得到一个AudioStreamOut，传进去的值会改吗？

    AudioStreamOut \*output = mAudioHardware->openOutputStream(\*pDevices,

                                                             (int \*)&format,

                                                             &channels,

                                                             &samplingRate,

                                                             &status);

      mHardwareStatus = AUDIO\_HW\_IDLE;

if (output != 0) {

//走哪个分支？我把答案告诉大家吧。

//刚才那个mAudioHardware->openOutputStream确实会更改指针对应的value。

//当然，我们说了，AF使用的是GENERIC的Audio硬件。大家有兴趣可以去看看它的实现。

//我待会再贴出它的内容。反正到这里。

//那几个值变成：format为PCM\_16\_BIT，channels为2，samplingRate为44100

//这样的话，那只能走else分支了。

        if ((flags & AudioSystem::OUTPUT\_FLAG\_DIRECT) ||

            (format != AudioSystem::PCM\_16\_BIT) ||

            (channels != AudioSystem::CHANNEL\_OUT\_STEREO)) {

            thread = new DirectOutputThread(this, output, ++mNextThreadId);

           } else {

//还记得前两节分析的同学，看到这里是不是明白了？恩，原来

//open一个Output，就会在AF中创建一个混音线程。设计得真好。

//想象下，所有设置为外放的程序，它的输出都是这个外放stream混音线程来工作

//所有设置为耳机的程序，它的输出都是这个耳机stream混音线程来完成。

//为什么对stream特加强调呢，没看见

//我们调用的是mAudioHardware->openOutputStream(0x2,,,)嘛。返回的

//是一个AudioStreamOut，可不是设备喔。Android把这些个东西都交给HAL层去实现了。

//不用自己来管理系统上有什么耳机，外设，蓝牙真实设备之类的东东，它反正用AudioStreamOut来表示它想要的就可以了。例如Generic的Audio Hal只支持一个OutputStream。--> only my opinion

           thread = new MixerThread(this, output, ++mNextThreadId);

  }

//好了，又多得了一个线程，

        mPlaybackThreads.add(mNextThreadId, thread);

        if (pSamplingRate) \*pSamplingRate = samplingRate;

        if (pFormat) \*pFormat = format;

        if (pChannels) \*pChannels = channels;

        if (pLatencyMs) \*pLatencyMs = thread->latency();

//从这里返回的是混音线程的索引。

        return mNextThreadId;

    }

    return 0;//如果没创建成功线程，则返回零。

}

好，我们回到AudioManagerBase中。

[--->AudioPolicyManagerBase::AudioPolicyManagerBase]

mHardwareOutput = mpClientInterface->openOutput(&outputDesc->mDevice,

                                    &outputDesc->mSamplingRate,

                                    &outputDesc->mFormat,

                                    &outputDesc->mChannels,

                                    &outputDesc->mLatency,

                                    outputDesc->mFlags);

//上面实际就返回一个线程index。我有点疑惑，难道APS就只这么一个实际是线程index的东西就就行了吗？虽然它把这个index当成hardware的标识了。

  //这个...估计是把int和指针加入到一个map了，方便管理。不看了。

addOutput(mHardwareOutput, outputDesc);

//不知道干嘛，待会看。

setOutputDevice(mHardwareOutput, (uint32\_t)AudioSystem::DEVICE\_OUT\_SPEAKER, true);

[--->setOutputDevice(mHardwareOutput,...)]

这个函数，很重要！另外，再传点技巧。不要老在source insight中后退后退了，直接找到window菜单，里边列出了最近打开的文件，找到我们的AudioManagerBase.cpp，不就行了吗？

void AudioPolicyManagerBase::setOutputDevice(audio\_io\_handle\_t output, uint32\_t device, bool force, int delayMs)

{

//注意我们的参数：

// output = 1，

//device为AudioSystem::DEVICE\_OUT\_SPEAKER

// force为true，delayMs用默认值0

//map吧？刚才通过addOutput已经加进去了

AudioOutputDescriptor \*outputDesc = mOutputs.valueFor(output);

    if (outputDesc->isDuplicated()) {

        setOutputDevice(outputDesc->mOutput1->mId, device, force, delayMs);

        setOutputDevice(outputDesc->mOutput2->mId, device, force, delayMs);

        return;

}

//还记得addOutput前设置的device吗？对了，为0X3，外放|耳机

uint32\_t prevDevice = (uint32\_t)outputDesc->device();

现在设置的是外设，

    if ((device == 0 || device == prevDevice) && !force) {

        return;

    }

//喔，设置这个outputDesc为外放

    outputDesc->mDevice = device;

popCount为2，因为device=0x2=0010

//另外，我对下面这个output== mHardwareOutput尤其感兴趣。还记得我们刚才的疑问吗？

// mHardwareOutput实际上是AF返回的一个线程索引，那AMB怎么根据这样一个东西来

//管理所有的线程呢？果然，这里就比较了output是不是等于最初创建的线程索引

//这就表明。虽然只有这么一个mHardwareOutput，但实际上还是能够操作其他output的！

    if (output == mHardwareOutput && AudioSystem::popCount(device) == 2) {

        setStrategyMute(STRATEGY\_MEDIA, true, output);

  usleep(outputDesc->mLatency\*2\*1000);

    }

// 晕，又冒出来一个AudioParameter，不过意思却很明白

//说我们要设置路由，新的输出设备为外放

//等我们以后讲由外放切换到耳机，再来看这个问题。

    AudioParameter param = AudioParameter();

    param.addInt(String8(AudioParameter::keyRouting), (int)device);

    mpClientInterface->setParameters(mHardwareOutput, param.toString(), delayMs);

    // update stream volumes according to new device

    applyStreamVolumes(output, device, delayMs);

    // if changing from a combined headset + speaker route, unmute media streams

    if (output == mHardwareOutput && AudioSystem::popCount(prevDevice) == 2) {

      //这里说，把media的音量置为0。以后再说。

setStrategyMute(STRATEGY\_MEDIA, false, output, delayMs);

    }

}

好了，返回了。

setOutputDevice(mHardwareOutput, (uint32\_t)AudioSystem::DEVICE\_OUT\_SPEAKER, true);

这个调研，更新了mHardwareOutput对应的输出路由设备，而且还发了一个命令给APS，说你给我更新对应混音线程的输出路由设备。

[--->AudioPolicyManagerBase::AudioPolicyManagerBase]

    .....

addOutput(mHardwareOutput, outputDesc);

        setOutputDevice(mHardwareOutput, (uint32\_t)AudioSystem::DEVICE\_OUT\_SPEAKER,

true);

  //只剩下最后一个函数了

updateDeviceForStrategy();

[----->updateDeviceForStrategy()]

  void AudioPolicyManagerBase::updateDeviceForStrategy()

{

    for (int i = 0; i < NUM\_STRATEGIES; i++) {

        mDeviceForStrategy[i] = getDeviceForStrategy((routing\_strategy)i, false);

    }

}

晕，又出来一个枚举。我们看看

[---->for (int i = 0; i < NUM\_STRATEGIES; i++)]

NUM\_STRATEGIES在hardware/libhardware\_legacy/include/hardware\_legacy/

AudioPolicyManagerBase.h中定义。

enum routing\_strategy {

//好像很好理解

            STRATEGY\_MEDIA,

            STRATEGY\_PHONE,//通话音吗？

            STRATEGY\_SONIFICATION,//除了其他三个外的，可以是铃声，提醒声等。

            STRATEGY\_DTMF,//好像是拨号音

            NUM\_STRATEGIES

        };

这个，反正我在SDK上没找到对应说明，我们待到以后看看会不会柳暗花明呢？

[----->getDeviceForStrategy((routing\_strategy)i, false)]

看这个函数名的意思是，为各种策略找到它对应的设备。

uint32\_t AudioPolicyManagerBase::getDeviceForStrategy(routing\_strategy strategy, bool fromCache)

{

  //  fromCache为false

//放眼望去，这个函数好像涉及到很对策略方面的事情。

//我们大概讲解下，至于系统为什么要这么做，问Google吧。

uint32\_t device = 0;

    switch (strategy) {

    case STRATEGY\_DTMF:

        if (mPhoneState != AudioSystem::MODE\_IN\_CALL) {

            //如果在打电话过程中，你再按按键，则和MEDIA走一个设备

            device = getDeviceForStrategy(STRATEGY\_MEDIA, false);

            break;

        }

        //注意这里没有break，所以在其他mode下，DTMF和PHONE用一个策略

    case STRATEGY\_PHONE:

       //还得判断用户是不是强制使用了输出设备。

        switch (mForceUse[AudioSystem::FOR\_COMMUNICATION]) {

        case AudioSystem::FORCE\_BT\_SCO:

            if (mPhoneState != AudioSystem::MODE\_IN\_CALL || strategy != STRATEGY\_DTMF) {

                device = mAvailableOutputDevices &

 AudioSystem::DEVICE\_OUT\_BLUETOOTH\_SCO\_CARKIT;

                if (device) break;

            }

            device = mAvailableOutputDevices &

AudioSystem::DEVICE\_OUT\_BLUETOOTH\_SCO\_HEADSET;

            if (device) break;

            device = mAvailableOutputDevices & AudioSystem::DEVICE\_OUT\_BLUETOOTH\_SCO;

            if (device) break;

            // if SCO device is requested but no SCO device is available, fall back to default

// case

            // FALL THROUGH

       //我们还记得强制设置那里吗？对了，此时都是FORCE\_NONE

      //而且，mAvailableOutputDevices是0X3 (外放|耳机)

        default:    // FORCE\_NONE

            device = mAvailableOutputDevices & AudioSystem::DEVICE\_OUT\_WIRED\_HEADPHONE;

            if (device) break;

            device = mAvailableOutputDevices & AudioSystem::DEVICE\_OUT\_WIRED\_HEADSET;

            if (device) break;

   //看，下面这句会成立。啥意思？如果有耳机的话，那么输出设备就是耳机

//太正确了。实际手机是不是就是这样的呢？

            device = mAvailableOutputDevices & AudioSystem::DEVICE\_OUT\_EARPIECE;

            break;

     //再验证下我们刚才说的，如果强制使用外放的话，

        case AudioSystem::FORCE\_SPEAKER:

            if (mPhoneState != AudioSystem::MODE\_IN\_CALL || strategy != STRATEGY\_DTMF) {

                device = mAvailableOutputDevices &

 AudioSystem::DEVICE\_OUT\_BLUETOOTH\_SCO\_CARKIT;

                if (device) break;

            }

//果然，会强制使用外放。

           device = mAvailableOutputDevices & AudioSystem::DEVICE\_OUT\_SPEAKER;

            break;

        }

    break;

    case STRATEGY\_SONIFICATION://分析方法同上，我不说了。

        if (mPhoneState == AudioSystem::MODE\_IN\_CALL) {

            device = getDeviceForStrategy(STRATEGY\_PHONE, false);

            break;

        }

        device = mAvailableOutputDevices & AudioSystem::DEVICE\_OUT\_SPEAKER;

// 同样没有break，说明SONIFICATION受MEDIA策略影响。

    case STRATEGY\_MEDIA: {

        uint32\_t device2 = mAvailableOutputDevices & AudioSystem::DEVICE\_OUT\_AUX\_DIGITAL;

        if (device2 == 0) {

            device2 = mAvailableOutputDevices & AudioSystem::DEVICE\_OUT\_WIRED\_HEADPHONE;

        }

        if (device2 == 0) {

            device2 = mAvailableOutputDevices & AudioSystem::DEVICE\_OUT\_WIRED\_HEADSET;

        }

   //可惜，上面那些高级设备我们都没有

        if (device2 == 0) {

            device2 = mAvailableOutputDevices & AudioSystem::DEVICE\_OUT\_SPEAKER;

        }

  //假设我们没有从SONIFICATION下来，那么device最终会= DEVICE\_OUT\_SPEAKER。

//假设我们从SONIFICATION下来，那么device还是等于DEVICE\_OUT\_SPEAKER

//奇怪，如果有耳机的话为何会走外放呢？普通耳机和线控耳机还能区分？

        device |= device2;

        } break;

    default:

        break;

    }

    return device;

}

好了，回到

[---->AudioPolicyManagerBase::updateDeviceForStrategy()]

void AudioPolicyManagerBase::updateDeviceForStrategy()

{

    for (int i = 0; i < NUM\_STRATEGIES; i++) {

        mDeviceForStrategy[i] = getDeviceForStrategy((routing\_strategy)i, false);

    }

}

这个函数完了，表明各种策略下使用的对应设备也准备好了。

真爽，一路回去，APS的构造就完了。

留个纪念：

AudioPolicyManagerBase::AudioPolicyManagerBase(AudioPolicyClientInterface \*clientInterface)

{

  ....

updateDeviceForStrategy();

}

AudioPolicyService::AudioPolicyService()

: BnAudioPolicyService() , mpPolicyManager(NULL)

{

  #if (defined GENERIC\_AUDIO) || (defined AUDIO\_POLICY\_TEST)

    mpPolicyManager = new AudioPolicyManagerBase(this);

LOGV("build for GENERIC\_AUDIO - using generic audio policy");

 ...

#endif

property\_get("ro.camera.sound.forced", value, "0");

mpPolicyManager->setSystemProperty("ro.camera.sound.forced", value);

}

总结下吧，AF,APS都创建完了，得到什么了吗？下面按先后顺序说说。

         AF创建了一个代表HAL对象的东西

         APS创建了两个AudioCommandThread，一个用来处理命令，一个用来播放tone。我们还没看。

         APS同时会创建AudioManagerBase，做为系统默认的音频管理

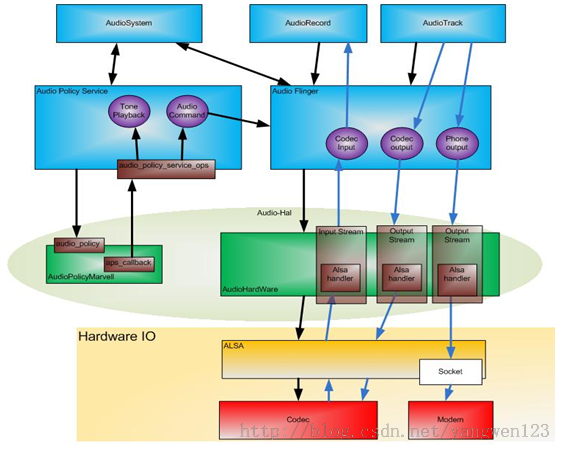
         AMB集中管理了策略上面的事情，同时会在AF的openOutput中创建一个混音线程。同时，AMB会更新一些策略上的安排。

另外，我们分析的AMB是Generic的，但不同厂商可以实现自己的策略。例如我可以设置只要有耳机，所有类型声音都从耳机出。

上面关于AMB方面，我们还只是看了看它的代码，还没有一个实际例子来体会。

#### 4.3.4.2 AudioPolicyService服务启动过程详解

AudioPolicyService是策略的制定者，比如什么时候打开音频接口设备、某种Stream类型的音频对应什么设备等等。而AudioFlinger则是策略的执行者，例如具体如何与音频设备通信，如何维护现有系统中的音频设备，以及多个音频流的混音如何处理等等都得由它来完成。AudioPolicyService根据用户配置来指导AudioFlinger加载设备接口，起到路由功能。



AudioPolicyService服务运行在mediaserver进程中，随着mediaserver进程启动而启动。

frameworks\av\media\mediaserver\ Main\_mediaserver.cpp

int main(int argc, char\*\* argv)

{

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

sp<IServiceManager> sm = defaultServiceManager();

ALOGI("ServiceManager: %p", sm.get());

VolumeManager::instantiate(); // volumemanager have to be started before audioflinger

AudioFlinger::instantiate();

MediaPlayerService::instantiate();

CameraService::instantiate();

AudioPolicyService::instantiate();

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

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

}

AudioPolicyService继承了模板类BinderService，该类用于注册native service。

frameworks\native\include\binder\ BinderService.h

template<typename SERVICE>

class BinderService

{

public:

static status\_t publish(bool allowIsolated = false) {

sp<IServiceManager> sm(defaultServiceManager());

return sm->addService(String16(SERVICE::getServiceName()), new SERVICE(), allowIsolated);

}

static void instantiate() { publish(); }

};

BinderService是一个模板类，该类的publish函数就是完成向ServiceManager注册服务。

static const char \*getServiceName() { return "media.audio\_policy"; }

AudioPolicyService注册名为media.audio\_policy的服务。

AudioPolicyService::AudioPolicyService()

: BnAudioPolicyService() , mpAudioPolicyDev(NULL) , mpAudioPolicy(NULL)

{

char value[PROPERTY\_VALUE\_MAX];

const struct hw\_module\_t \*module;

int forced\_val;

int rc;

Mutex::Autolock \_l(mLock);

// start tone playback thread

mTonePlaybackThread = new AudioCommandThread(String8("ApmTone"), this);

// start audio commands thread

mAudioCommandThread = new AudioCommandThread(String8("ApmAudio"), this);

// start output activity command thread

mOutputCommandThread = new AudioCommandThread(String8("ApmOutput"), this);

/\* instantiate the audio policy manager \*/

/\* 加载audio\_policy.default.so库得到audio\_policy\_module模块 \*/

rc = hw\_get\_module(AUDIO\_POLICY\_HARDWARE\_MODULE\_ID, &module);

if (rc)

return;

/\* 通过audio\_policy\_module模块打开audio\_policy\_device设备 \*/

rc = audio\_policy\_dev\_open(module, &mpAudioPolicyDev);

ALOGE\_IF(rc, "couldn't open audio policy device (%s)", strerror(-rc));

if (rc)

return;

//通过audio\_policy\_device设备创建audio\_policy

rc = mpAudioPolicyDev->create\_audio\_policy(mpAudioPolicyDev, &aps\_ops, this,

&mpAudioPolicy);

ALOGE\_IF(rc, "couldn't create audio policy (%s)", strerror(-rc));

if (rc)

return;

rc = mpAudioPolicy->init\_check(mpAudioPolicy);

ALOGE\_IF(rc, "couldn't init\_check the audio policy (%s)", strerror(-rc));

if (rc)

return;

/\* SPRD: maybe set this property better, but here just change the default value @{ \*/

property\_get("ro.camera.sound.forced", value, "1");

forced\_val = strtol(value, NULL, 0);

ALOGV("setForceUse() !forced\_val=%d ",!forced\_val);

mpAudioPolicy->set\_can\_mute\_enforced\_audible(mpAudioPolicy, !forced\_val);

ALOGI("Loaded audio policy from %s (%s)", module->name, module->id);

// 读取audio\_effects.conf文件

if (access(AUDIO\_EFFECT\_VENDOR\_CONFIG\_FILE, R\_OK) == 0) {

loadPreProcessorConfig(AUDIO\_EFFECT\_VENDOR\_CONFIG\_FILE);

} else if (access(AUDIO\_EFFECT\_DEFAULT\_CONFIG\_FILE, R\_OK) == 0) {

loadPreProcessorConfig(AUDIO\_EFFECT\_DEFAULT\_CONFIG\_FILE);

}

}

1. 创建AudioCommandThread (ApmTone、ApmAudio、ApmOutput)
2. 加载legacy\_ap\_module
3. 打开legacy\_ap\_device
4. 创建legacy\_audio\_policy
5. 读取audio\_effects.conf



创建AudioCommandThread线程

在AudioPolicyService对象构造过程中，分别创建了ApmTone、ApmAudio、ApmOutput三个AudioCommandThread线程：

1、 ApmTone用于播放tone音；

2、 ApmAudio用于执行audio命令；

3、ApmOutput用于执行输出命令；

在第一次强引用AudioCommandThread线程对象时，AudioCommandThread的onFirstRef函数被回调，在此启动线程

void AudioPolicyService::AudioCommandThread::onFirstRef()

{

run(mName.string(), ANDROID\_PRIORITY\_AUDIO);

}

这里采用异步方式来执行audio command，当需要执行上表中的命令时，首先将命令投递到AudioCommandThread的mAudioCommands命令向量表中，然后通过mWaitWorkCV.signal()唤醒AudioCommandThread线程，被唤醒的AudioCommandThread线程执行完command后，又通过mWaitWorkCV.waitRelative(mLock, waitTime)睡眠等待命令到来。

加载audio\_policy\_module模块

audio\_policy硬件抽象层动态库位于/system/lib/hw/目录下，命名为：audio\_policy.$(TARGET\_BOARD\_PLATFORM).so。audiopolicy的硬件抽象层定义在hardware\libhardware\_legacy\audio\audio\_policy\_hal.cpp中，AUDIO\_POLICY\_HARDWARE\_MODULE\_ID硬件抽象模块定义如下：

hardware\libhardware\_legacy\audio\ audio\_policy\_hal.cpp【audio\_policy.scx15.so】

struct legacy\_ap\_module HAL\_MODULE\_INFO\_SYM = {

module: {

common: {

tag: HARDWARE\_MODULE\_TAG,

version\_major: 1,

version\_minor: 0,

id: AUDIO\_POLICY\_HARDWARE\_MODULE\_ID,

name: "LEGACY Audio Policy HAL",

author: "The Android Open Source Project",

methods: &legacy\_ap\_module\_methods,

dso : NULL,

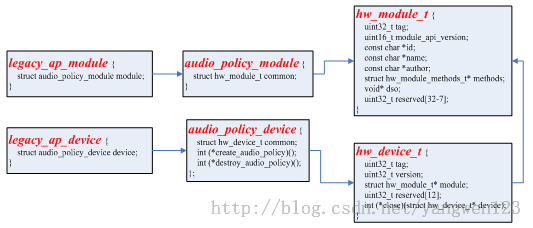
reserved : {0},

},

},

};

legacy\_ap\_module继承于audio\_policy\_module。



关于hw\_get\_module函数加载硬件抽象层模块的过程请参考[Android硬件抽象Hardware库加载过程源码分析](http://blog.csdn.net/yangwen123/article/details/12040909)。

打开audio\_policy\_device设备

hardware\libhardware\include\hardware\ audio\_policy.h

static inline int audio\_policy\_dev\_open(const hw\_module\_t\* module,

struct audio\_policy\_device\*\* device)

{

return module->methods->open(module, AUDIO\_POLICY\_INTERFACE,

(hw\_device\_t\*\*)device);

}

通过legacy\_ap\_module模块的open方法来打开一个legacy\_ap\_device设备。

hardware\libhardware\_legacy\audio\ audio\_policy\_hal.cpp

static int legacy\_ap\_dev\_open(const hw\_module\_t\* module, const char\* name,

hw\_device\_t\*\* device)

{

struct legacy\_ap\_device \*dev;

if (strcmp(name, AUDIO\_POLICY\_INTERFACE) != 0)

return -EINVAL;

dev = (struct legacy\_ap\_device \*)calloc(1, sizeof(\*dev));

if (!dev)

return -ENOMEM;

dev->device.common.tag = HARDWARE\_DEVICE\_TAG;

dev->device.common.version = 0;

dev->device.common.module = const\_cast<hw\_module\_t\*>(module);

dev->device.common.close = legacy\_ap\_dev\_close;

dev->device.create\_audio\_policy = create\_legacy\_ap;

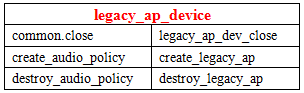
dev->device.destroy\_audio\_policy = destroy\_legacy\_ap;

\*device = &dev->device.common;

return 0;

}

打开得到一个legacy\_ap\_device设备，通过该抽象设备可以创建一个audio\_policy对象。



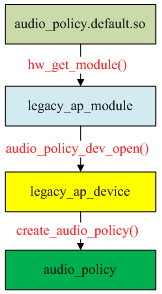
创建audio\_policy对象

在打开legacy\_ap\_device设备时，该设备的create\_audio\_policy成员初始化为create\_legacy\_ap函数指针，我们通过legacy\_ap\_device设备可以创建一个legacy\_audio\_policy对象。

rc = mpAudioPolicyDev->create\_audio\_policy(mpAudioPolicyDev, &aps\_ops, this,

&mpAudioPolicy);

这里通过audio\_policy\_device设备创建audio策略对象



hardware\libhardware\_legacy\audio\ audio\_policy\_hal.cpp

static int create\_legacy\_ap(const struct audio\_policy\_device \*device,

struct audio\_policy\_service\_ops \*aps\_ops,

void \*service,

struct audio\_policy \*\*ap)

{

struct legacy\_audio\_policy \*lap;

int ret;

if (!service || !aps\_ops)

return -EINVAL;

lap = (struct legacy\_audio\_policy \*)calloc(1, sizeof(\*lap));

if (!lap)

return -ENOMEM;

lap->policy.set\_device\_connection\_state = ap\_set\_device\_connection\_state;

…

lap->policy.dump = ap\_dump;

lap->policy.is\_offload\_supported = ap\_is\_offload\_supported;

lap->service = service;

lap->aps\_ops = aps\_ops;

lap->service\_client = new AudioPolicyCompatClient(aps\_ops, service);

if (!lap->service\_client) {

ret = -ENOMEM;

goto err\_new\_compat\_client;

}

lap->apm = createAudioPolicyManager(lap->service\_client);

if (!lap->apm) {

ret = -ENOMEM;

goto err\_create\_apm;

}

\*ap = &lap->policy;

return 0;

err\_create\_apm:

delete lap->service\_client;

err\_new\_compat\_client:

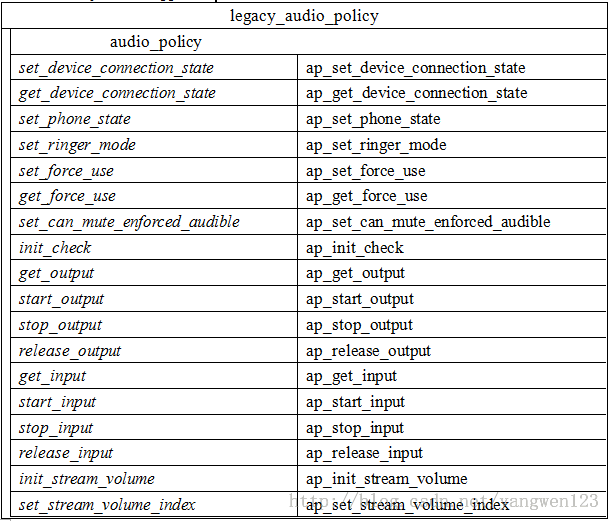
free(lap);

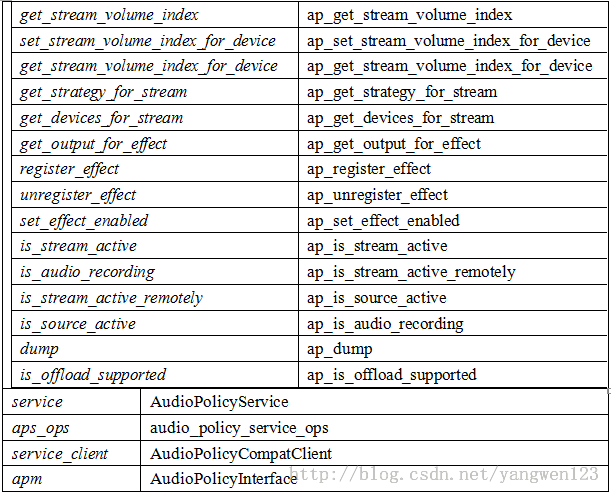
\*ap = NULL;

return ret;

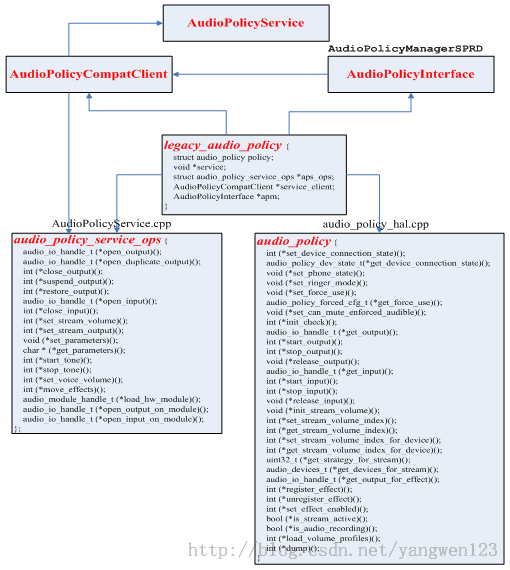
}

audio\_policy实现在audio\_policy\_hal.cpp中，audio\_policy\_service\_ops实现在AudioPolicyService.cpp中。create\_audio\_policy()函数就是创建并初始化一个legacy\_audio\_policy对象。





audio\_policy与AudioPolicyService、AudioPolicyCompatClient之间的关系如下：



AudioPolicyClient创建

hardware\libhardware\_legacy\audio\ AudioPolicyCompatClient.h

AudioPolicyCompatClient(struct audio\_policy\_service\_ops \*serviceOps,void \*service) :

mServiceOps(serviceOps) , mService(service) {}

AudioPolicyCompatClient是对audio\_policy\_service\_ops的封装类，对外提供audio\_policy\_service\_ops数据结构中定义的接口。

AudioPolicyManager创建

extern "C" AudioPolicyInterface\* createAudioPolicyManager(AudioPolicyClientInterface \*clientInterface)

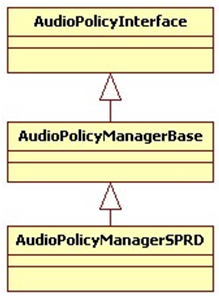
{

ALOGI("SPRD policy manager created.");

return new AudioPolicyManagerSPRD(clientInterface);

}

使用AudioPolicyClientInterface对象来构造AudioPolicyManagerSPRD对象，AudioPolicyManagerSPRD继承于AudioPolicyManagerBase，而AudioPolicyManagerBase又继承于AudioPolicyInterface。



hardware\libhardware\_legacy\audio\ AudioPolicyManagerBase.cpp

AudioPolicyManagerBase::AudioPolicyManagerBase(AudioPolicyClientInterface \*clientInterface)

:

#ifdef AUDIO\_POLICY\_TEST

Thread(false),

#endif //AUDIO\_POLICY\_TEST

//变量初始化

mPrimaryOutput((audio\_io\_handle\_t)0),

mAvailableOutputDevices(AUDIO\_DEVICE\_NONE),

mPhoneState(AudioSystem::MODE\_NORMAL),

mLimitRingtoneVolume(false), mLastVoiceVolume(-1.0f),

mTotalEffectsCpuLoad(0), mTotalEffectsMemory(0),

mA2dpSuspended(false), mHasA2dp(false), mHasUsb(false), mHasRemoteSubmix(false),

mSpeakerDrcEnabled(false), mFmOffGoing(false)

{

//引用AudioPolicyCompatClient对象，这样音频管理器AudioPolicyManager就可以使用audio\_policy\_service\_ops中的接口

mpClientInterface = clientInterface;

for (int i = 0; i < AudioSystem::NUM\_FORCE\_USE; i++) {

mForceUse[i] = AudioSystem::FORCE\_NONE;

}

mA2dpDeviceAddress = String8("");

mScoDeviceAddress = String8("");

mUsbCardAndDevice = String8("");

/\*\*

\* 优先加载/vendor/etc/audio\_policy.conf配置文件，如果该配置文件不存在，则

\* 加载/system/etc/audio\_policy.conf配置文件，如果该文件还是不存在，则通过

\* 函数defaultAudioPolicyConfig()来设置默认音频接口

\*/

if (loadAudioPolicyConfig(AUDIO\_POLICY\_VENDOR\_CONFIG\_FILE) != NO\_ERROR) {

if (loadAudioPolicyConfig(AUDIO\_POLICY\_CONFIG\_FILE) != NO\_ERROR) {

ALOGE("could not load audio policy configuration file, setting defaults");

defaultAudioPolicyConfig();

}

}

//设置各种音频流对应的音量调节点,must be done after reading the policy

initializeVolumeCurves();

// open all output streams needed to access attached devices

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

//通过名称打开对应的音频接口硬件抽象库

mHwModules[i]->mHandle = mpClientInterface->loadHwModule(mHwModules[i]->mName);

if (mHwModules[i]->mHandle == 0) {

ALOGW("could not open HW module %s", mHwModules[i]->mName);

continue;

}

// open all output streams needed to access attached devices

// except for direct output streams that are only opened when they are actually

// required by an app.

for (size\_t j = 0; j < mHwModules[i]->mOutputProfiles.size(); j++)

{

const IOProfile \*outProfile = mHwModules[i]->mOutputProfiles[j];

//打开mAttachedOutputDevices对应的输出

if ((outProfile->mSupportedDevices & mAttachedOutputDevices) &&

((outProfile->mFlags & AUDIO\_OUTPUT\_FLAG\_DIRECT) == 0)) {

//将输出IOProfile封装为AudioOutputDescriptor对象

AudioOutputDescriptor \*outputDesc = new AudioOutputDescriptor(outProfile);

//设置当前音频接口的默认输出设备

outputDesc->mDevice = (audio\_devices\_t)(mDefaultOutputDevice & outProfile->mSupportedDevices);

//打开输出，在AudioFlinger中创建PlaybackThread线程，并返回该线程的id

audio\_io\_handle\_t output = mpClientInterface->openOutput(

outProfile->mModule->mHandle,

&outputDesc->mDevice,

&outputDesc->mSamplingRate,

&outputDesc->mFormat,

&outputDesc->mChannelMask,

&outputDesc->mLatency,

outputDesc->mFlags);

if (output == 0) {

delete outputDesc;

} else {

//设置可以使用的输出设备为mAttachedOutputDevices

mAvailableOutputDevices =(audio\_devices\_t)(mAvailableOutputDevices | (outProfile->mSupportedDevices & mAttachedOutputDevices));

if (mPrimaryOutput == 0 && outProfile->mFlags & AUDIO\_OUTPUT\_FLAG\_PRIMARY) {

mPrimaryOutput = output;

}

//将输出描述符对象AudioOutputDescriptor及创建的PlaybackThread线程id以键值对形式保存

addOutput(output, outputDesc);

//设置默认输出设备

setOutputDevice(output,(audio\_devices\_t)(mDefaultOutputDevice & outProfile->mSupportedDevices),true);

}

}

}

}

ALOGE\_IF((mAttachedOutputDevices & ~mAvailableOutputDevices),

"Not output found for attached devices %08x",

(mAttachedOutputDevices & ~mAvailableOutputDevices));

ALOGE\_IF((mPrimaryOutput == 0), "Failed to open primary output");

updateDevicesAndOutputs();

// add for bug158794 start

char bootvalue[PROPERTY\_VALUE\_MAX];

// prop sys.boot\_completed will set 1 when system ready (ActivityManagerService.java)...

property\_get("sys.boot\_completed", bootvalue, "");

if (strncmp("1", bootvalue, 1) != 0) {

startReadingThread();

}

// add for bug158794 end

#ifdef AUDIO\_POLICY\_TEST

...

#endif //AUDIO\_POLICY\_TEST

}

AudioPolicyManagerBase对象构造过程中主要完成以下几个步骤：

1、  loadAudioPolicyConfig(AUDIO\_POLICY\_CONFIG\_FILE)加载audio\_policy.conf配置文件；

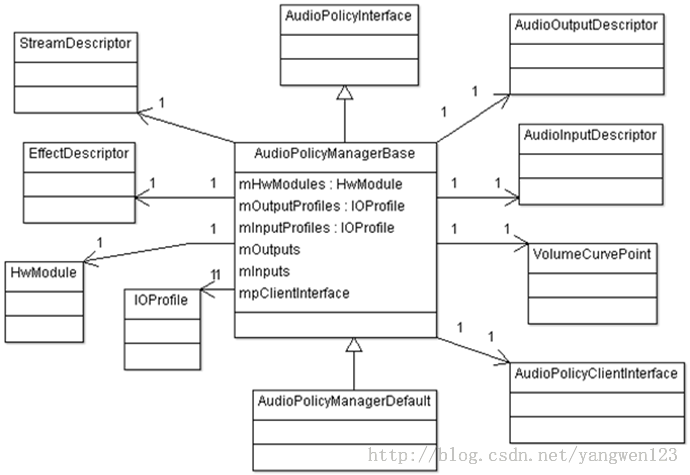
2、  initializeVolumeCurves()初始化各种音频流对应的音量调节点；

3、  加载audio policy硬件抽象库：mpClientInterface->loadHwModule(mHwModules[i]->mName)

4、  打开attached\_output\_devices输出：

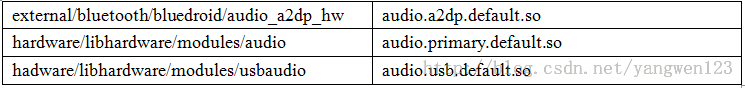
mpClientInterface->openOutput()；

5、  保存输出设备描述符对象：addOutput(output, outputDesc);



读取audio\_policy.conf文件

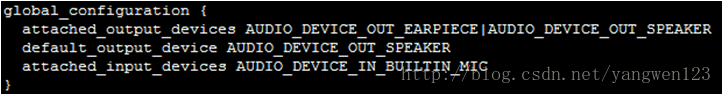
Android为每种音频接口定义了对应的硬件抽象层，且编译为单独的so库。



每种音频接口定义了不同的输入输出，一个接口可以具有多个输入或者输出，每个输入输出有可以支持不同的音频设备。通过读取audio\_policy.conf文件可以获取系统支持的音频接口参数。

audio\_policy.conf文件定义了两种音频配置信息：

1、  当前系统支持的音频输入输出设备及默认输入输出设备；



这些信息时通过global\_configuration配置项来设置，在global\_configuration中定义了三种音频设备信息：

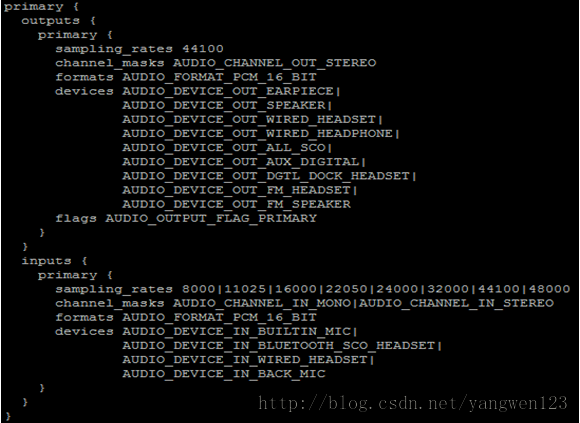
attached\_output\_devices：已连接的输出设备；

default\_output\_device：默认输出设备；

attached\_input\_devices：已连接的输入设备；

1、  系统支持的音频接口信息；

audio\_policy.conf定义了系统支持的所有音频接口参数信息，比如primary、a2dp、usb等，对于primary定义如下：



a2dp定义：



usb定义：



每种音频接口包含输入输出，每种输入输出又包含多种输入输出配置，每种输入输出配置又支持多种音频设备。AudioPolicyManagerBase首先加载/vendor/etc/audio\_policy.conf，如果该文件不存在，则加/system/etc/audio\_policy.conf。

status\_t AudioPolicyManagerBase::loadAudioPolicyConfig(const char \*path)

{

cnode \*root;

char \*data;

data = (char \*)load\_file(path, NULL);

if (data == NULL) {

return -ENODEV;

}

root = config\_node("", "");

//读取配置文件

config\_load(root, data);

//解析global\_configuration

loadGlobalConfig(root);

//解析audio\_hw\_modules

loadHwModules(root);

config\_free(root);

free(root);

free(data);

ALOGI("loadAudioPolicyConfig() loaded %s\n", path);

return NO\_ERROR;

}

通过loadGlobalConfig(root)函数来读取这些全局配置信息。

void AudioPolicyManagerBase::loadGlobalConfig(cnode \*root)

{

cnode \*node = config\_find(root, GLOBAL\_CONFIG\_TAG);

if (node == NULL) {

return;

}

node = node->first\_child;

while (node) {

//attached\_output\_devices AUDIO\_DEVICE\_OUT\_EARPIECE

if (strcmp(ATTACHED\_OUTPUT\_DEVICES\_TAG, node->name) == 0) {

mAttachedOutputDevices = parseDeviceNames((char \*)node->value);

ALOGW\_IF(mAttachedOutputDevices == AUDIO\_DEVICE\_NONE,

"loadGlobalConfig() no attached output devices");

ALOGV("loadGlobalConfig()mAttachedOutputDevices%04x", mAttachedOutputDevices);

//default\_output\_device AUDIO\_DEVICE\_OUT\_SPEAKER

} else if (strcmp(DEFAULT\_OUTPUT\_DEVICE\_TAG, node->name) == 0) {

mDefaultOutputDevice= (audio\_devices\_t)stringToEnum(sDeviceNameToEnumTable,ARRAY\_SIZE(sDeviceNameToEnumTable),(char \*)node->value);

ALOGW\_IF(mDefaultOutputDevice == AUDIO\_DEVICE\_NONE,

"loadGlobalConfig() default device not specified");

ALOGV("loadGlobalConfig() mDefaultOutputDevice %04x", mDefaultOutputDevice);

//attached\_input\_devices AUDIO\_DEVICE\_IN\_BUILTIN\_MIC

} else if (strcmp(ATTACHED\_INPUT\_DEVICES\_TAG, node->name) == 0) {

mAvailableInputDevices = parseDeviceNames((char \*)node->value) & ~AUDIO\_DEVICE\_BIT\_IN;

ALOGV("loadGlobalConfig() mAvailableInputDevices %04x", mAvailableInputDevices);

//speaker\_drc\_enabled

} else if (strcmp(SPEAKER\_DRC\_ENABLED\_TAG, node->name) == 0) {

mSpeakerDrcEnabled = stringToBool((char \*)node->value);

ALOGV("loadGlobalConfig() mSpeakerDrcEnabled = %d", mSpeakerDrcEnabled);

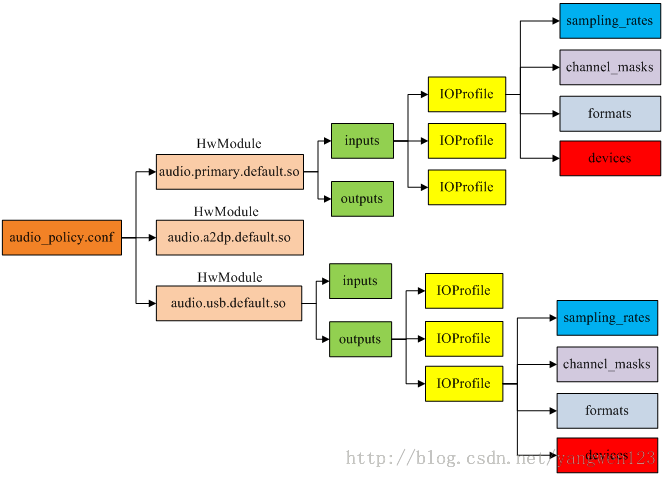
}

node = node->next;

}

}

audio\_policy.conf同时定义了多个audio 接口，每一个audio 接口包含若干output和input，而每个output和input又同时支持多种输入输出模式，每种输入输出模式又支持若干种设备。



通过loadHwModules ()函数来加载系统配置的所有audio 接口：

void AudioPolicyManagerBase::loadHwModules(cnode \*root)

{

//audio\_hw\_modules

cnode \*node = config\_find(root, AUDIO\_HW\_MODULE\_TAG);

if (node == NULL) {

return;

}

node = node->first\_child;

while (node) {

ALOGV("loadHwModules() loading module %s", node->name);

//加载音频接口

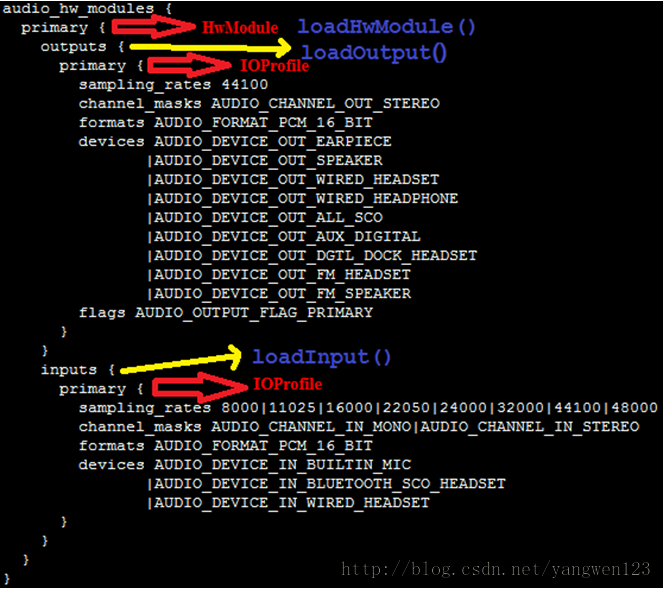
loadHwModule(node);

node = node->next;

}

}

由于audio\_policy.conf可以定义多个音频接口，因此该函数循环调用loadHwModule()来解析每个音频接口参数信息。Android定义HwModule类来描述每一个audio 接口参数，定义IOProfile类来描述输入输出模式配置。





到此就将audio\_policy.conf文件中音频接口配置信息解析到了AudioPolicyManagerBase的成员变量mHwModules、mAttachedOutputDevices、mDefaultOutputDevice、mAvailableInputDevices中。

初始化音量调节点

音量调节点设置在Android4.1与Android4.4中的实现完全不同，在Android4.1中是通过VolumeManager服务来管理，通过devicevolume.xml文件来配置，但Android4.4取消了VolumeManager服务，将音量控制放到AudioPolicyManagerBase中。在AudioPolicyManagerBase中定义了音量调节对应的音频流描述符数组：

StreamDescriptor mStreams[AudioSystem::NUM\_STREAM\_TYPES];

initializeVolumeCurves()函数就是初始化该数组元素：

void AudioPolicyManagerBase::initializeVolumeCurves()

{

for (int i = 0; i < AUDIO\_STREAM\_CNT; i++) {

for (int j = 0; j < DEVICE\_CATEGORY\_CNT; j++) {

mStreams[i].mVolumeCurve[j] =

sVolumeProfiles[i][j];

}

}

// Check availability of DRC on speaker path: if available, override some of the speaker curves

if (mSpeakerDrcEnabled) {

mStreams[AUDIO\_STREAM\_SYSTEM].mVolumeCurve[DEVICE\_CATEGORY\_SPEAKER] =

sDefaultSystemVolumeCurveDrc;

mStreams[AUDIO\_STREAM\_RING].mVolumeCurve[DEVICE\_CATEGORY\_SPEAKER] =

sSpeakerSonificationVolumeCurveDrc;

mStreams[AUDIO\_STREAM\_ALARM].mVolumeCurve[DEVICE\_CATEGORY\_SPEAKER] =

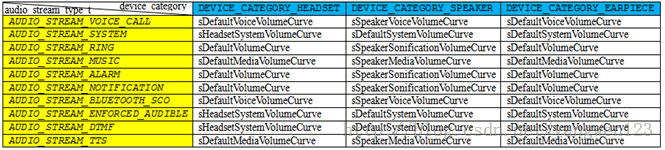
sSpeakerSonificationVolumeCurveDrc;

mStreams[AUDIO\_STREAM\_NOTIFICATION].mVolumeCurve[DEVICE\_CATEGORY\_SPEAKER] =sSpeakerSonificationVolumeCurveDrc;

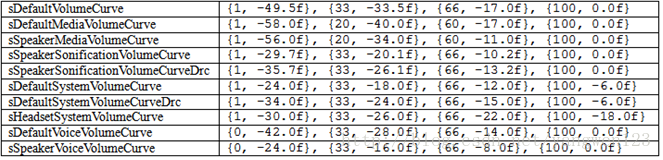
}

}

sVolumeProfiles数组定义了不同音频设备下不同音频流对应的音量调节档位，定义如下：

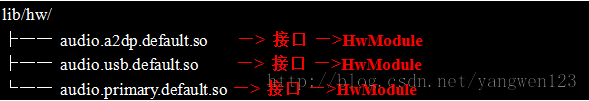


数组元素为音量调节档位，每种模式下的音量调节都包含4个档位，定义如下：



加载audio\_module模块

AudioPolicyManager通过读取audio\_policy.conf配置文件，可以知道系统当前支持那些音频接口以及attached的输入输出设备、默认输出设备。接下来就需要加载这些音频接口的硬件抽象库。



这三中音频接口硬件抽象定义如下：

/vendor/sprd/open-source/libs/audio/audio\_hw.c 【audio.primary.scx15.so】

struct audio\_module HAL\_MODULE\_INFO\_SYM = {

.common = {

.tag = HARDWARE\_MODULE\_TAG,

.module\_api\_version = AUDIO\_MODULE\_API\_VERSION\_0\_1,

.hal\_api\_version = HARDWARE\_HAL\_API\_VERSION,

.id = AUDIO\_HARDWARE\_MODULE\_ID,

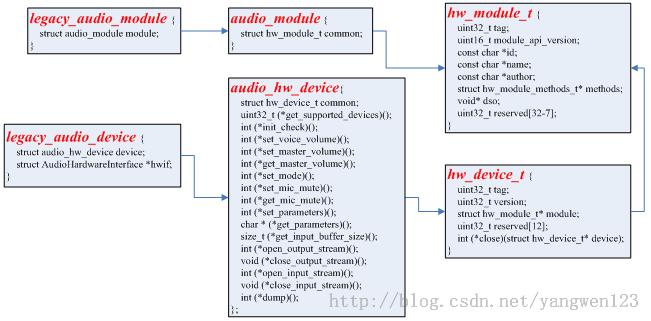
.name = "Spreadtrum Audio HW HAL",

.author = "The Android Open Source Project",

.methods = &hal\_module\_methods,

},

};



external/bluetooth/bluedroid/audio\_a2dp\_hw/audio\_a2dp\_hw.c【audio.a2dp.default.so】

struct audio\_module HAL\_MODULE\_INFO\_SYM = {

.common = {

.tag = HARDWARE\_MODULE\_TAG,

.version\_major = 1,

.version\_minor = 0,

.id = AUDIO\_HARDWARE\_MODULE\_ID,

.name = "A2DP Audio HW HAL",

.author = "The Android Open Source Project",

.methods = &hal\_module\_methods,

},

};

hardware/libhardware/modules/usbaudio/audio\_hw.c【audio. usb.default.so】

struct audio\_module HAL\_MODULE\_INFO\_SYM = {

.common = {

.tag = HARDWARE\_MODULE\_TAG,

.module\_api\_version = AUDIO\_MODULE\_API\_VERSION\_0\_1,

.hal\_api\_version = HARDWARE\_HAL\_API\_VERSION,

.id = AUDIO\_HARDWARE\_MODULE\_ID,

.name = "USB audio HW HAL",

.author = "The Android Open Source Project",

.methods = &hal\_module\_methods,

},

};

AudioPolicyClientInterface提供了加载音频接口硬件抽象库的接口函数，通过前面的介绍，我们知道，AudioPolicyCompatClient通过代理audio\_policy\_service\_ops实现AudioPolicyClientInterface接口。

hardware\libhardware\_legacy\audio\ AudioPolicyCompatClient.cpp

audio\_module\_handle\_t AudioPolicyCompatClient::loadHwModule(const char \*moduleName)

{

return mServiceOps->load\_hw\_module(mService, moduleName);

}

AudioPolicyCompatClient将音频模块加载工作交给audio\_policy\_service\_ops

frameworks\av\services\audioflinger\ AudioPolicyService.cpp

static audio\_module\_handle\_t aps\_load\_hw\_module(void \*service,const char \*name)

{

sp<IAudioFlinger> af = AudioSystem::get\_audio\_flinger();

if (af == 0) {

ALOGW("%s: could not get AudioFlinger", \_\_func\_\_);

return 0;

}

return af->loadHwModule(name);

}

AudioPolicyService又将其转交给AudioFlinger

frameworks\av\services\audioflinger\ AudioFlinger.cpp

audio\_module\_handle\_t AudioFlinger::loadHwModule(const char \*name)

{

if (!settingsAllowed()) {

return 0;

}

Mutex::Autolock \_l(mLock);

return loadHwModule\_l(name);

}



audio\_module\_handle\_t AudioFlinger::loadHwModule\_l(const char \*name)

{

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

if (strncmp(mAudioHwDevs.valueAt(i)->moduleName(), name, strlen(name)) == 0) {

ALOGW("loadHwModule() module %s already loaded", name);

return mAudioHwDevs.keyAt(i);

}

}

audio\_hw\_device\_t \*dev;

//加载音频接口对应的so库，得到对应的音频接口设备audio\_hw\_device\_t

int rc = load\_audio\_interface(name, &dev);

if (rc) {

ALOGI("loadHwModule() error %d loading module %s ", rc, name);

return 0;

}

mHardwareStatus = AUDIO\_HW\_INIT;

rc = dev->init\_check(dev);

mHardwareStatus = AUDIO\_HW\_IDLE;

if (rc) {

ALOGI("loadHwModule() init check error %d for module %s ", rc, name);

return 0;

}

if ((mMasterVolumeSupportLvl != MVS\_NONE) &&

(NULL != dev->set\_master\_volume)) {

AutoMutex lock(mHardwareLock);

mHardwareStatus = AUDIO\_HW\_SET\_MASTER\_VOLUME;

dev->set\_master\_volume(dev, mMasterVolume);

mHardwareStatus = AUDIO\_HW\_IDLE;

}

audio\_module\_handle\_t handle = nextUniqueId();

mAudioHwDevs.add(handle, new AudioHwDevice(name, dev));

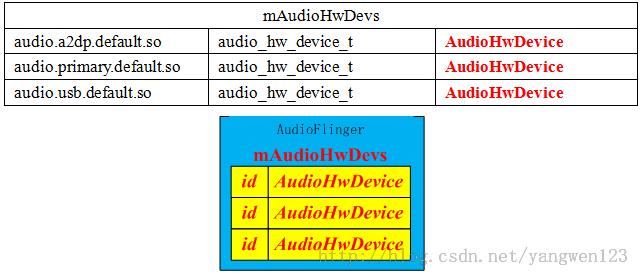
ALOGI("loadHwModule() Loaded %s audio interface from %s (%s) handle %d",

name, dev->common.module->name, dev->common.module->id, handle);

return handle;

}

函数首先加载系统定义的音频接口对应的so库，并打开该音频接口的抽象硬件设备audio\_hw\_device\_t，为每个音频接口设备生成独一无二的ID号，同时将打开的音频接口设备封装为AudioHwDevice对象，将系统中所有的音频接口设备保存到AudioFlinger的成员变量mAudioHwDevs中。



函数load\_audio\_interface根据音频接口名称来打开抽象的音频接口设备audio\_hw\_device\_t。

static int load\_audio\_interface(const char \*if\_name, audio\_hw\_device\_t \*\*dev)

{

const hw\_module\_t \*mod;

int rc;

//根据名字加载audio\_module模块

rc = hw\_get\_module\_by\_class(AUDIO\_HARDWARE\_MODULE\_ID, if\_name, &mod);

ALOGE\_IF(rc, "%s couldn't load audio hw module %s.%s (%s)", \_\_func\_\_,

AUDIO\_HARDWARE\_MODULE\_ID, if\_name, strerror(-rc));

if (rc) {

goto out;

}

//打开audio\_device设备

rc = audio\_hw\_device\_open(mod, dev);

ALOGE\_IF(rc, "%s couldn't open audio hw device in %s.%s (%s)", \_\_func\_\_,

AUDIO\_HARDWARE\_MODULE\_ID, if\_name, strerror(-rc));

if (rc) {

goto out;

}

if ((\*dev)->common.version != AUDIO\_DEVICE\_API\_VERSION\_CURRENT) {

ALOGE("%s wrong audio hw device version %04x", \_\_func\_\_, (\*dev)->common.version);

rc = BAD\_VALUE;

goto out;

}

return 0;

out:

\*dev = NULL;

return rc;

}

hardware\libhardware\include\hardware\ Audio.h

static inline int audio\_hw\_device\_open(const struct hw\_module\_t\* module,

struct audio\_hw\_device\*\* device)

{

return module->methods->open(module, AUDIO\_HARDWARE\_INTERFACE,

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

}

hardware\libhardware\_legacy\audio\ audio\_hw\_hal.cpp

static int legacy\_adev\_open(const hw\_module\_t\* module, const char\* name,

hw\_device\_t\*\* device)

{

struct legacy\_audio\_device \*ladev;

int ret;

if (strcmp(name, AUDIO\_HARDWARE\_INTERFACE) != 0)

return -EINVAL;

ladev = (struct legacy\_audio\_device \*)calloc(1, sizeof(\*ladev));

if (!ladev)

return -ENOMEM;

ladev->device.common.tag = HARDWARE\_DEVICE\_TAG;

ladev->device.common.version = AUDIO\_DEVICE\_API\_VERSION\_1\_0;

ladev->device.common.module = const\_cast<hw\_module\_t\*>(module);

ladev->device.common.close = legacy\_adev\_close;

ladev->device.get\_supported\_devices = adev\_get\_supported\_devices;

…

ladev->device.dump = adev\_dump;

ladev->hwif = createAudioHardware();

if (!ladev->hwif) {

ret = -EIO;

goto err\_create\_audio\_hw;

}

\*device = &ladev->device.common;

return 0;

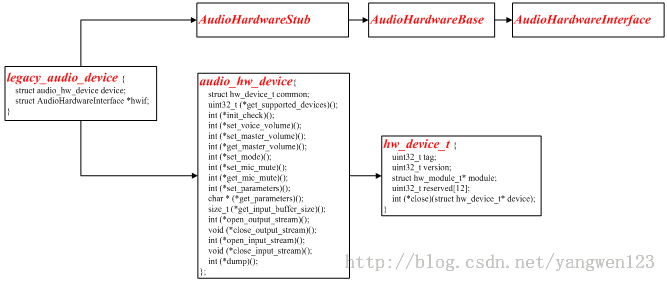
err\_create\_audio\_hw:

free(ladev);

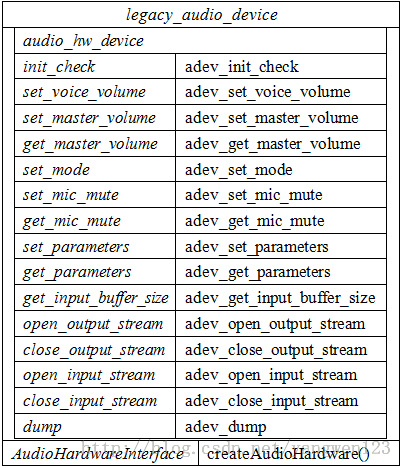
return ret;

}

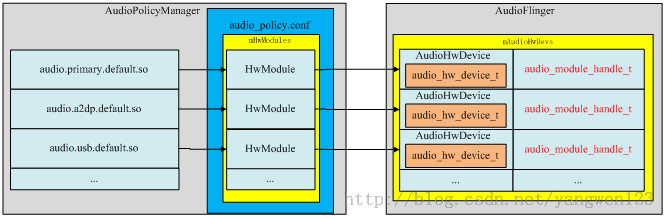
打开音频接口设备过程其实就是构造并初始化legacy\_audio\_device对象过程，legacy\_audio\_device数据结构关系如下：



legacy\_adev\_open函数就是创建并初始化一个legacy\_audio\_device对象：

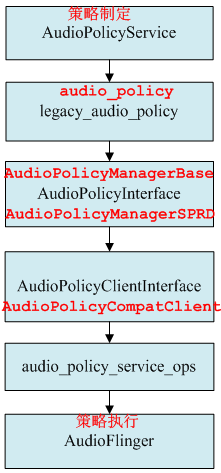


到此就加载完系统定义的所有音频接口，并生成相应的数据对象，如下图所示：



打开音频输出

AudioPolicyService加载完所有音频接口后，就知道了系统支持的所有音频接口参数，可以为音频输出提供决策。



为了能正常播放音频数据，需要创建抽象的音频输出接口对象，打开音频输出过程如下：

audio\_io\_handle\_t AudioPolicyCompatClient::openOutput(audio\_module\_handle\_t module,

audio\_devices\_t \*pDevices,

uint32\_t \*pSamplingRate,

audio\_format\_t \*pFormat,

audio\_channel\_mask\_t \*pChannelMask,

uint32\_t \*pLatencyMs,

audio\_output\_flags\_t flags,

const audio\_offload\_info\_t \*offloadInfo)

{

return mServiceOps->open\_output\_on\_module(mService,module, pDevices, pSamplingRate,

pFormat, pChannelMask, pLatencyMs,

flags, offloadInfo);

}

static audio\_io\_handle\_t aps\_open\_output\_on\_module(void \*service,

audio\_module\_handle\_t module,

audio\_devices\_t \*pDevices,

uint32\_t \*pSamplingRate,

audio\_format\_t \*pFormat,

audio\_channel\_mask\_t \*pChannelMask,

uint32\_t \*pLatencyMs,

audio\_output\_flags\_t flags,

const audio\_offload\_info\_t \*offloadInfo)

{

sp<IAudioFlinger> af = AudioSystem::get\_audio\_flinger();

if (af == 0) {

ALOGW("%s: could not get AudioFlinger", \_\_func\_\_);

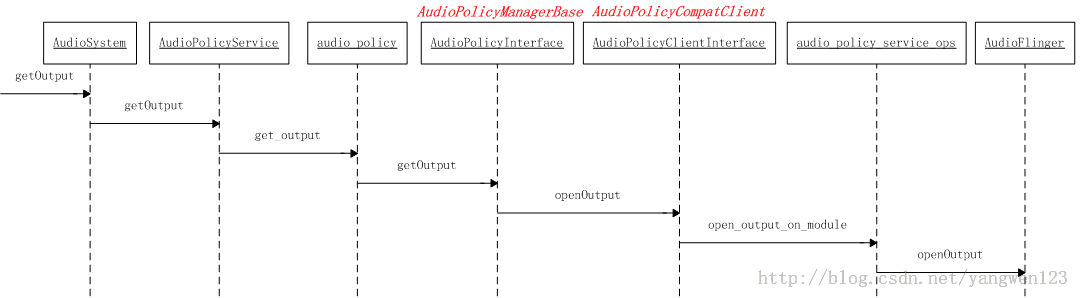
return 0;

}

return af->openOutput(module, pDevices, pSamplingRate, pFormat, pChannelMask,

pLatencyMs, flags, offloadInfo);

}



audio\_io\_handle\_t AudioFlinger::openOutput(audio\_module\_handle\_t module,

audio\_devices\_t \*pDevices,

uint32\_t \*pSamplingRate,

audio\_format\_t \*pFormat,

audio\_channel\_mask\_t \*pChannelMask,

uint32\_t \*pLatencyMs,

audio\_output\_flags\_t flags,

const audio\_offload\_info\_t \*offloadInfo)

{

PlaybackThread \*thread = NULL;

struct audio\_config config;

config.sample\_rate = (pSamplingRate != NULL) ? \*pSamplingRate : 0;

config.channel\_mask = (pChannelMask != NULL) ? \*pChannelMask : 0;

config.format = (pFormat != NULL) ? \*pFormat : AUDIO\_FORMAT\_DEFAULT;

if (offloadInfo) {

config.offload\_info = \*offloadInfo;

}

//创建一个音频输出流对象audio\_stream\_out\_t

audio\_stream\_out\_t \*outStream = NULL;

AudioHwDevice \*outHwDev;

ALOGV("openOutput(), module %d Device %x, SamplingRate %d, Format %#08x, Channels %x, flags %x",

module,

(pDevices != NULL) ? \*pDevices : 0,

config.sample\_rate,

config.format,

config.channel\_mask,

flags);

ALOGV("openOutput(), offloadInfo %p version 0x%04x",

offloadInfo, offloadInfo == NULL ? -1 : offloadInfo->version );

if (pDevices == NULL || \*pDevices == 0) {

return 0;

}

Mutex::Autolock \_l(mLock);

//从音频接口列表mAudioHwDevs中查找出对应的音频接口，如果找不到，则重新加载音频接口动态库

outHwDev = findSuitableHwDev\_l(module, \*pDevices);

if (outHwDev == NULL)

return 0;

//取出module对应的audio\_hw\_device\_t设备

audio\_hw\_device\_t \*hwDevHal = outHwDev->hwDevice();

//为音频输出流生成一个独一无二的id号

audio\_io\_handle\_t id = nextUniqueId();

mHardwareStatus = AUDIO\_HW\_OUTPUT\_OPEN;

//打开音频输出流

status\_t status = hwDevHal->open\_output\_stream(hwDevHal,

id,

\*pDevices,

(audio\_output\_flags\_t)flags,

&config,

&outStream);

mHardwareStatus = AUDIO\_HW\_IDLE;

ALOGV("openOutput() openOutputStream returned output %p, SamplingRate %d, Format %#08x, "

"Channels %x, status %d",

outStream,

config.sample\_rate,

config.format,

config.channel\_mask,

status);

if (status == NO\_ERROR && outStream != NULL) {

//使用AudioStreamOut来封装音频输出流audio\_stream\_out\_t

AudioStreamOut \*output = new AudioStreamOut(outHwDev, outStream, flags);

//根据flag标志位，创建不同类型的线程

if (flags & AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD) {

thread = new OffloadThread(this, output, id, \*pDevices);

ALOGV("openOutput() created offload output: ID %d thread %p", id, thread);

} else if ((flags & AUDIO\_OUTPUT\_FLAG\_DIRECT) ||

(config.format != AUDIO\_FORMAT\_PCM\_16\_BIT) ||

(config.channel\_mask != AUDIO\_CHANNEL\_OUT\_STEREO)) {

thread = new DirectOutputThread(this, output, id, \*pDevices);

ALOGV("openOutput() created direct output: ID %d thread %p", id, thread);

} else {

thread = new MixerThread(this, output, id, \*pDevices);

ALOGV("openOutput() created mixer output: ID %d thread %p", id, thread);

}

//将创建的线程及id以键值对的形式保存在mPlaybackThreads中

mPlaybackThreads.add(id, thread);

if (pSamplingRate != NULL) {

\*pSamplingRate = config.sample\_rate;

}

if (pFormat != NULL) {

\*pFormat = config.format;

}

if (pChannelMask != NULL) {

\*pChannelMask = config.channel\_mask;

}

if (pLatencyMs != NULL) {

\*pLatencyMs = thread->latency();

}

// notify client processes of the new output creation

thread->audioConfigChanged\_l(AudioSystem::OUTPUT\_OPENED);

// the first primary output opened designates the primary hw device

if ((mPrimaryHardwareDev == NULL) && (flags & AUDIO\_OUTPUT\_FLAG\_PRIMARY)) {

ALOGI("Using module %d has the primary audio interface", module);

mPrimaryHardwareDev = outHwDev;

AutoMutex lock(mHardwareLock);

mHardwareStatus = AUDIO\_HW\_SET\_MODE;

hwDevHal->set\_mode(hwDevHal, mMode);

mHardwareStatus = AUDIO\_HW\_IDLE;

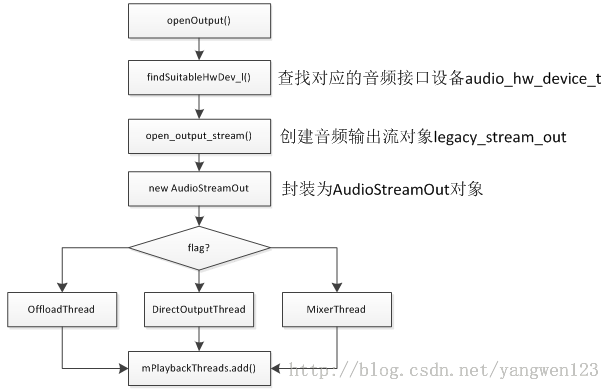
}

return id;

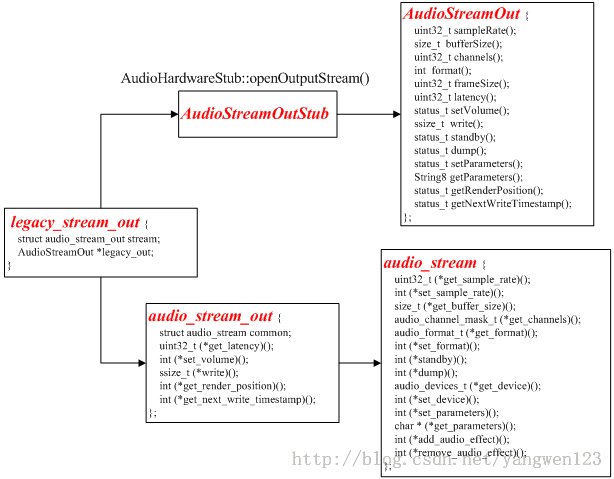
}

return 0;

}



打开音频输出流过程其实就是创建AudioStreamOut对象及PlaybackThread线程过程。首先通过抽象的音频接口设备audio\_hw\_device\_t来创建输出流对象legacy\_stream\_out。



static int adev\_open\_output\_stream(struct audio\_hw\_device \*dev,

audio\_io\_handle\_t handle,

audio\_devices\_t devices,

audio\_output\_flags\_t flags,

struct audio\_config \*config,

struct audio\_stream\_out \*\*stream\_out)

{

struct legacy\_audio\_device \*ladev = to\_ladev(dev);

status\_t status;

struct legacy\_stream\_out \*out;

int ret;

//分配一个legacy\_stream\_out对象

out = (struct legacy\_stream\_out \*)calloc(1, sizeof(\*out));

if (!out)

return -ENOMEM;

devices = convert\_audio\_device(devices, HAL\_API\_REV\_2\_0, HAL\_API\_REV\_1\_0);

//创建AudioStreamOut对象

out->legacy\_out = ladev->hwif->openOutputStream(devices, (int \*) &config->format,

&config->channel\_mask,

&config->sample\_rate, &status);

if (!out->legacy\_out) {

ret = status;

goto err\_open;

}

//初始化成员变量audio\_stream

out->stream.common.get\_sample\_rate = out\_get\_sample\_rate;

…

\*stream\_out = &out->stream;

return 0;

err\_open:

free(out);

\*stream\_out = NULL;

return ret;

}

由于legacy\_audio\_device的成员变量hwif的类型为AudioHardwareInterface，因此通过调用AudioHardwareInterface的接口openOutputStream()来创建AudioStreamOut对象。

AudioStreamOut\* AudioHardwareStub::openOutputStream(

uint32\_t devices, int \*format, uint32\_t \*channels, uint32\_t \*sampleRate, status\_t \*status)

{

AudioStreamOutStub\* out = new AudioStreamOutStub();

status\_t lStatus = out->set(format, channels, sampleRate);

if (status) {

\*status = lStatus;

}

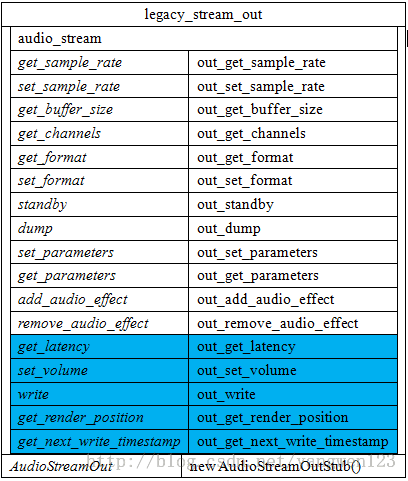
if (lStatus == NO\_ERROR)

return out;

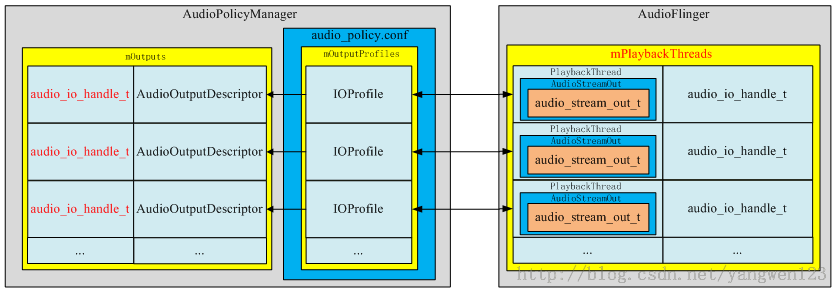
delete out;

return 0;

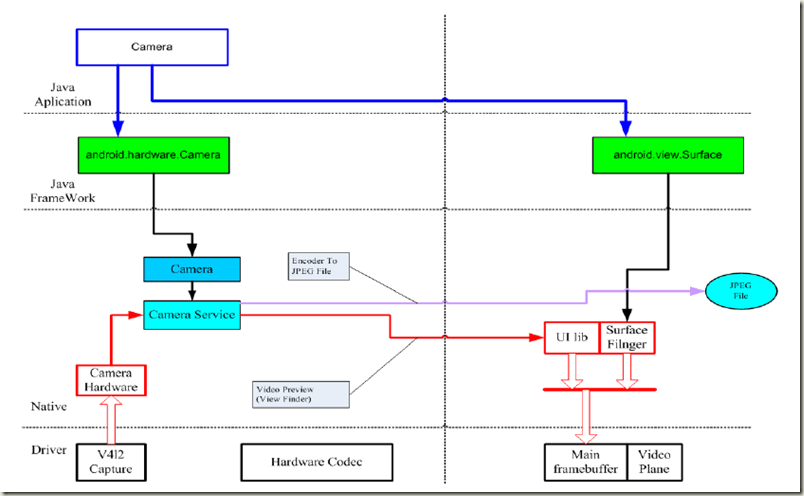
}



打开音频输出后，在AudioFlinger与AudioPolicyService中的表现形式如下：



# Camera框架



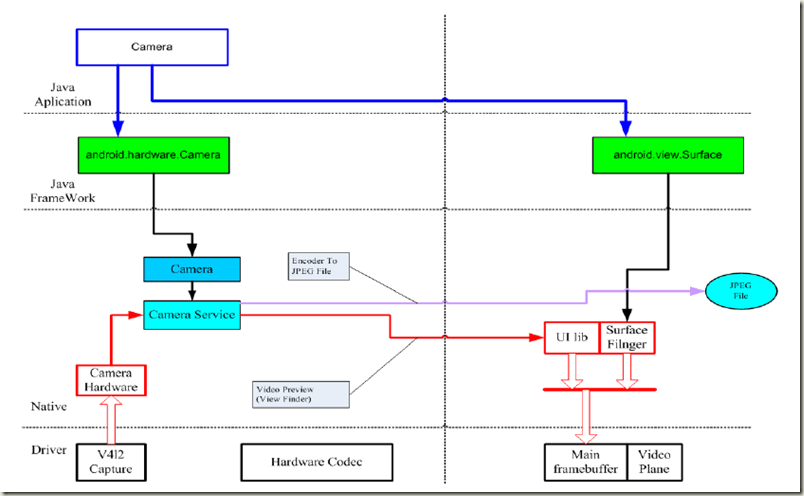
<http://blog.chinaunix.net/uid-26941022-id-3916492.html>

<http://rnmichelle.blog.51cto.com/2729211/491743/>

<http://blog.csdn.net/liuxu0703/article/details/6127258>

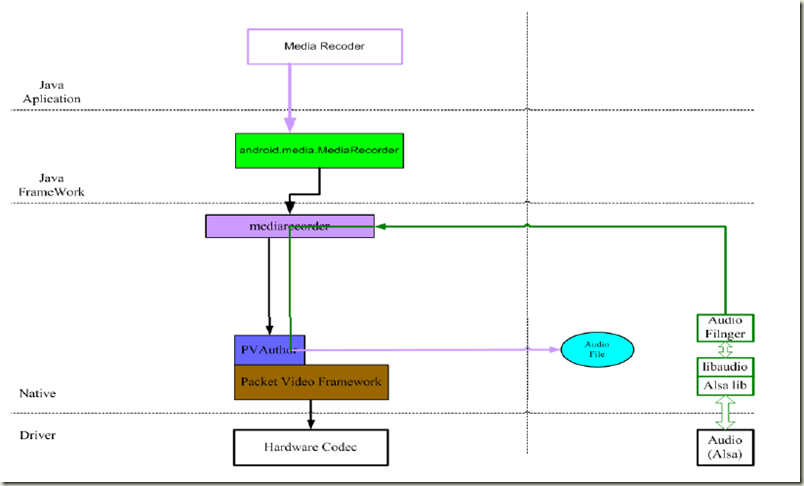
<http://blog.csdn.net/bm7623/article/details/7073941>

<http://blog.csdn.net/michaelcao1980/article/details/43233637>



# Media Recoder

**SoundRecorder框架图**



**VideoCamera框架图**

