目录

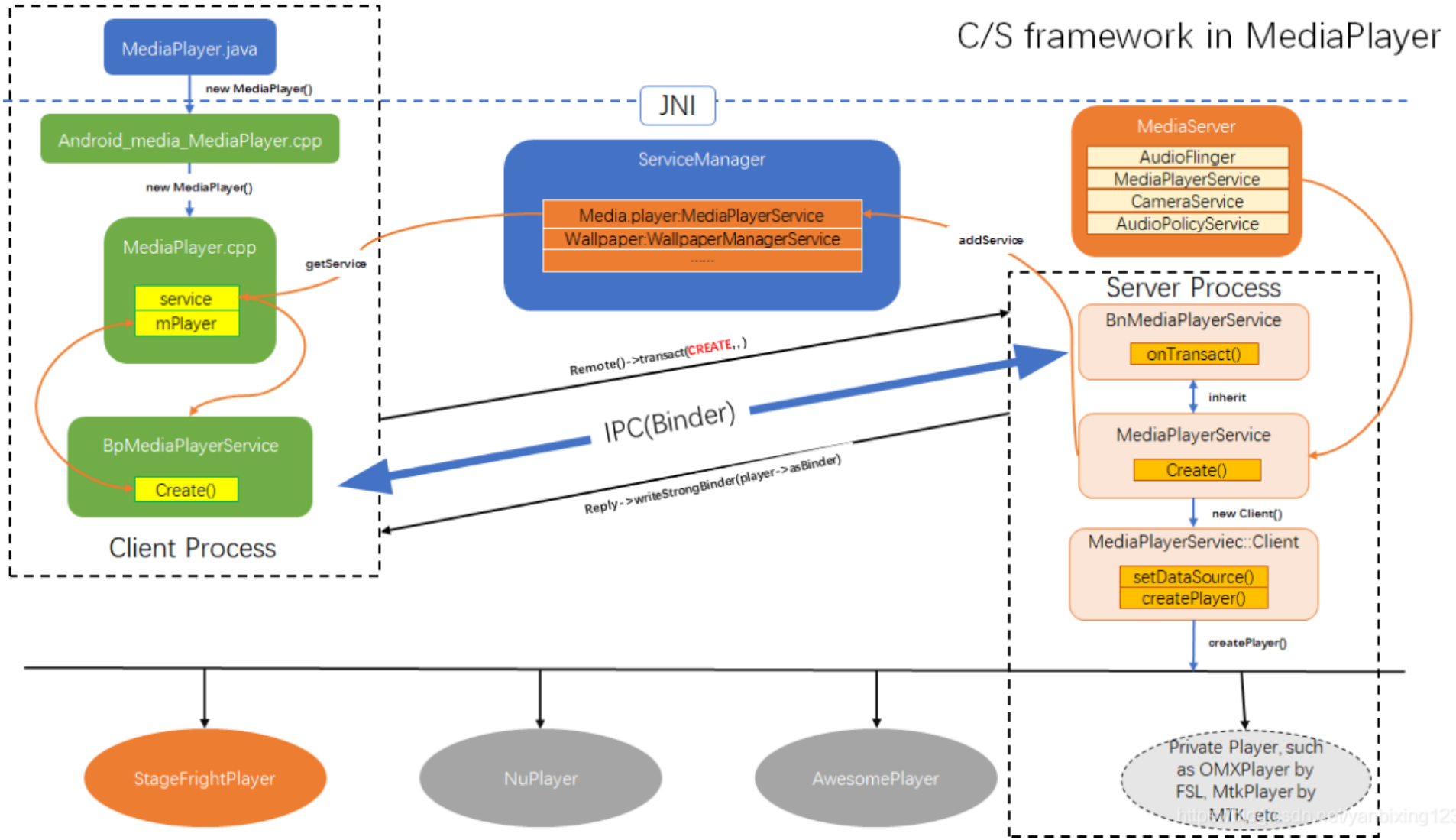
概述

Android 多媒体框架支持播放各种常见媒体类型，以便轻松地将音频、视频和图片集成到应用中。

我们可以通过 MediaPlayer API 播放存储在应用资源中的媒体文件（原始资源）、文件系统中的独立文件或者通过网络连接收到的数据流中的音频或视频。

MediaPlayer 可以分为 Client/Server 两个部分，他们分别在不同的进程中，不同进程间的通信使用 Binder 机制，整体架构图如下：

概述


C/S framework in MediaPlayer
MediaPlayer.java
new MediaPlayer()
Android_media_MediaPlayer.cpp
new MediaPlayer()
MediaPlayer.cpp
service
mPlayer
getService
BpMediaPlayerService
Create()
Client Process
JNI
ServiceManager
Media.player:MediaPlayerService
Wallpaper:WallpaperManagerService
.....
addService
MediaServer
AudioFlinger
MediaPlayerService
CameraService
AudioPolicyService
Server Process
BnMediaPlayerService
onTransact()
inherit
MediaPlayerService
Create()
new Client()
MediaPlayerServiec::Client
setDataSource()
createPlayer()
createPlayer()
Remote()->transact(CREATE,,)
IPC(Binder)
Reply->writeStrongBinder(player->asBinder)
StageFrightPlayer
NuPlayer
AwesomePlayer
Private Player, such as OMXPlayer by FSL, MtkPlayer by MTK, etc.

mediaserver 启动流程

mediaserver 通过 Android.bp 文件配置相关编译选项。

在 Android.bp 中利用 init_rc 属性指定 mediaserver 的 mediaserver.rc 文件，让 mediaserver 由 init 进程启动执行。

```
// frameworks/av/media/mediaserver/Android.bp

cc_binary {
    name: "mediaserver",
    srcs: ["main_mediaserver.cpp"],
    init_rc: ["mediaserver.rc"],
    // ......
}
```

mediaserver 会被编译成一个可执行文件，位于设备的 /system/bin/ 目录下

```
console:/ # ls -l /system/bin/mediaserver
-rwxr-xr-x 1 root shell 8892 2009-01-01 08:00 /system/bin/mediaserver
```

mediaserver 启动流程

mediaserver.rc 设置了 mediaserver 相关启动参数，并给予其高优先级，以确保媒体框架和相关应用的流畅运行。

```
// frameworks/av/media/mediaserver/mediaserver.rc

on property:init.svc.media=*
    setprop init.svc.mediadrm ${init.svc.media}

service media /system/bin/mediaserver
    class main
    user media
    group audio camera inet net_bt net_bt_admin net_bw_acct drmrpc mediadrm
    ioprio rt 4
    task_profiles ProcessCapacityHigh HighPerformance
```

mediaserver 启动流程

在 main_mediaserver.cpp 的 main 函数中通过调用 MediaPlayerService 的 instantiate 函数，完成了 MeadiaPlayerService 的实例化。

```
int main(int argc __unused, char **argv __unused)
{
    signal(SIGPIPE, SIG_IGN);

    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm(defaultServiceManager());
    ALOGI("ServiceManager: %p", sm.get());
    AIcu_initializeIcuOrDie();
    MediaPlayerService::instantiate(); // 实例化 MediaPlayerService
    ResourceManagerService::instantiate(); // 实例化 ResourceManagerService
    registerExtensions();
    ::android::hardware::configureRpcThreadpool(16, false);
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool(); // 将当前线程加入到线程池中
    ::android::hardware::joinRpcThreadpool();
}
```

mediaserver 启动流程

MediaPlayerService 的 instantiate 函数中 new 了一个 MediaPlayerService 对象，并将其注册到了 ServiceManager 中。

```
void MediaPlayerService::instantiate() {
    defaultServiceManager()->addService(
            String16("media.player"), new MediaPlayerService());
}
```

```
class MediaPlayerService : public BnMediaPlayerService;

class BnMediaPlayerService: public BnInterface<IMediaPlayerService>;

class IMediaPlayerService: public IInterface;
```

mediaserver 启动流程

再来看 MediaPlayerService 的构造函数，在其中完成了 MediaPlayer 内置工厂的注册。MediaPlayer 工厂实例最终会保存在 sFactoryMap 中。

```
MediaPlayerService::MediaPlayerService()
{
    // ......
    MediaPlayerFactory::registerBuiltinFactories(); // 注册 MediaPlayer 内置工厂
}
```

```
void MediaPlayerFactory::registerBuiltinFactories() {
    // ......
    IFactory* factory = new NuPlayerFactory();
    if (registerFactory_l(factory, NU_PLAYER) != OK)
        delete factory;
    factory = new TestPlayerFactory();
    if (registerFactory_l(factory, TEST_PLAYER) != OK)
        delete factory;
    // ......
}
```

```
MediaPlayerFactory::tFactoryMap MediaPlayerFactory::sFactoryMap;
```

setDataSource()

接下来我们通过 setDataSource 函数来分析 MediaPlayer 的 Client 端是如何与 Server 端建立连接与设置数据源。

首先是 Java 层调用 MediaPlayer.java 的 setDataSource() 方法。支持多种不同的媒体来源，例如：本地资源、内部 URI、外部网址（流式传输）。

我们这里以播放本地文件资源为例，Java层最终会调用到 JNI 层的 setDataSourceFD 函数，如下：

setDataSource()

```
// frameworks/base/media/jni/android_media_MediaPlayer.cpp

static void
android_media_MediaPlayer_setDataSourceFD(JNIEnv *env, jobject thiz, jobject
fileDescriptor, jlong offset, jlong length)
{
    sp<MediaPlayer> mp = getMediaPlayer(env, thiz); // 获取 MediaPlayer 对象
    if (mp == NULL ) {
        jniThrowException(env, "java/lang/IllegalStateException", NULL);
        return;
    }

    if (fileDescriptor == NULL) {
        jniThrowException(env, "java/lang/IllegalArgumentException", NULL);
        return;
    }
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    ALOGV("setDataSourceFD: fd %d", fd);

    // 调用 MediaPlayer 的 setDataSource 函数
    process_media_player_call( env, thiz, mp->setDataSource(fd, offset, length),
"java/io/IOException", "setDataSourceFD failed." );
}
```

setDataSource()

```
status_t MediaPlayer::setDataSource(int fd, int64_t offset, int64_t length)
{
    ALOGV("setDataSource(%d, %" PRId64 ", %" PRId64 ")", fd, offset, length);
    status_t err = UNKNOWN_ERROR;
    // 获取 MediaPlayerService
    const sp<IMediaPlayerService> service(getMediaPlayerService());
    if (service != 0) {
       // 调用 MediaPlayerService 的 create 函数创建 MediaPlayerService::Client 对象
       // MediaPlayerService 会为每一个 Client 端创建一个 MediaPlayerService::Client 对象
       // 这里创建的 player 对象才是真正干活的
        sp<IMediaPlayer> player(service->create(this, mAudioSessionId));
        if ((NO_ERROR != doSetRetransmitEndpoint(player)) ||
          // 继续往下设置数据源
            (NO_ERROR != player->setDataSource(fd, offset, length))) {
            player.clear();
        }
        // 保存 player，便于后续使用
        err = attachNewPlayer(player);
    }
    return err;
}
```

Client 与 Server

在这里解释一下MediaPlayer, MediaPlayerService和MediaPlayerService::Client的三角关系。

MediaPlayer 是客户端

```
class MediaPlayer : public BnMediaPlayerClient,
                    public virtual IMediaDeathNotifier;
```

MediaPlayerService 和 MediaPlayerService::Client 是服务端

```
class MediaPlayerService : public BnMediaPlayerService;
```

```
class MediaPlayerService::Client : public BnMediaPlayer;
```

真正负责播放工作的是 MediaPlayerService::Client，MediaPlayer 与 MediaPlayerService::Client 之间通过 Binder 机制通信，它们互相持有对方的 IBinder 对象。

我们通过 MediaPlayerService 的 create() 函数执行过程可以看到它们是如何建立联系。

Client 与 Server

```
class BpMediaPlayerService: public BpInterface<IMediaPlayerService>
{
public:
    virtual sp<IMediaPlayer> create(
            const sp<IMediaPlayerClient>& client, audio_session_t
audioSessionId) {
        Parcel data, reply;
        data.writeInterfaceToken(IMediaPlayerService::getInterfaceDescriptor());
        // 将 MediaPlayer 对象转换为 IBinder 对象
        data.writeStrongBinder(IInterface::asBinder(client));
        data.writeInt32(audioSessionId);

        remote()->transact(CREATE, data, &reply);
        // 返回 MediaPlayerService::Client 的 IBinder 对象
        return interface_cast<IMediaPlayer>(reply.readStrongBinder());
    }
}
```

Client 与 Server

```
status_t BnMediaPlayerService::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch (code) {
        case CREATE: {
            CHECK_INTERFACE(IMediaPlayerService, data, reply);
            // 获取 MediaPlayer 的 IBinder 对象
            sp<IMediaPlayerClient> client =
                interface_cast<IMediaPlayerClient>(data.readStrongBinder());
            audio_session_t audioSessionId = (audio_session_t) data.readInt32();
// 调用 MediaPlayerService 的 create() 函数，创建 MediaPlayerService::Client 对象
            sp<IMediaPlayer> player = create(client, audioSessionId);
// 将 MediaPlayerService::Client 对象转换为 IBinder 对象并返回给调用的客户端
            reply->writeStrongBinder(IInterface::asBinder(player));
            return NO_ERROR;
        } break;
    }
}
```

Client 与 Server

```
sp<IMediaPlayer> MediaPlayerService::create(const sp<IMediaPlayerClient>&
client,
        audio_session_t audioSessionId)
{
    pid_t pid = IPCThreadState::self()->getCallingPid();
    int32_t connId = android_atomic_inc(&mNextConnId);
    // 创建 MediaPlayerService::Client 对象
    sp<Client> c = new Client(
            this, pid, connId, client, audioSessionId,
            IPCThreadState::self()->getCallingUid());

    ALOGV("Create new client(%d) from pid %d, uid %d, ", connId, pid,
         IPCThreadState::self()->getCallingUid());

    wp<Client> w = c;
    {
        Mutex::Autolock lock(mLock);
        mClients.add(w);
    }
    return c;
}
```

setDataSource()

我们继续往下看，MediaPlayerService::Client 是如何设置数据源

```
status_t MediaPlayerService::Client::setDataSource(int fd, int64_t offset,
int64_t length)
{
    // ......
    // 获取 playerType
    player_type playerType = MediaPlayerFactory::getPlayerType(this,
                                                               fd,
                                                               offset,
                                                               length);
    // 根据 playerType 创建 player，实际是创建了一个 NuPlayerDriver 对象
    sp<MediaPlayerBase> p = setDataSource_pre(playerType);
    if (p == NULL) {
        return NO_INIT;
    }

    // now set data source
    // 继续往下设置 NuPlayerDriver 数据源
    return mStatus = setDataSource_post(p, p->setDataSource(fd, offset,
length));
}
```

playerType 如何获取

playerType 的确定最终是由 GET_PLAYER_TYPE_IMPL 这个宏定义实现的

```
#define GET_PLAYER_TYPE_IMPL(a...)                          \
    Mutex::Autolock lock_(&sLock);                          \
                                                            \
    player_type ret = STAGEFRIGHT_PLAYER;                   \
    float bestScore = 0.0;                                  \
                                                            \
    for (size_t i = 0; i < sFactoryMap.size(); ++i) {       \
                                                            \
        IFactory* v = sFactoryMap.valueAt(i);               \
        float thisScore;                                    \
        CHECK(v != NULL);                                   \
        thisScore = v->scoreFactory(a, bestScore);          \
        if (thisScore > bestScore) {                        \
            ret = sFactoryMap.keyAt(i);                     \
            bestScore = thisScore;                          \
        }                                                   \
    }                                                       \
                                                            \
    if (0.0 == bestScore) {                                 \
        ret = getDefaultPlayerType();                       \
    }                                                       \
                                                            \
    return ret;
```

NuPlayerDriver 与 NuPlayer

NuPlayerDriver 与 NuPlayer 的关系

NuPlayerDriver 是 NuPlayer 的接口类，它负责与应用程序进行交互。NuPlayerDriver 提供了一系列方法，用于控制播放、设置播放参数、获取播放状态等。

NuPlayer 是 NuPlayerDriver 的实现类，它负责具体的播放逻辑。NuPlayer 使用 MediaCodec 进行音视频解码，并使用 SurfaceFlinger 进行视频渲染。

NuPlayerDriver 在初始化时会 new 一个 NuPlayer 对象存储在 mPlayer 字段中。

```
NuPlayerDriver::NuPlayerDriver(pid_t pid)
    : mState(STATE_IDLE),
      // ......
      mPlayer(new NuPlayer(pid, mMediaClock)),
      // ......
      mAutoLoop(false) {
    // ......
    mPlayer->init(this);
}
```

NuPlayerDriver 创建过程

NuPlayerDriver 创建是采用的工厂模式

```
sp<MediaPlayerBase> MediaPlayerFactory::createPlayer(
        player_type playerType,
        const sp<MediaPlayerBase::Listener> &listener,
        pid_t pid) {
    sp<MediaPlayerBase> p;
    IFactory* factory;
    // ......
    factory = sFactoryMap.valueFor(playerType);
    CHECK(NULL != factory);
    p = factory->createPlayer(pid);
    // ......
    return p;
}
```

这里的 sFactoryMap 由 mediaserver 启动时初始化

setDataSource()

我们继续往下看，NuPlayerDriver 是如何设置数据源

```
status_t NuPlayerDriver::setDataSource(int fd, int64_t offset, int64_t length)
{
    ALOGV("setDataSource(%p) file(%d)", this, fd);
    Mutex::Autolock autoLock(mLock);

    if (mState != STATE_IDLE) {
        return INVALID_OPERATION;
    }

    mState = STATE_SET_DATASOURCE_PENDING;

    mPlayer->setDataSourceAsync(fd, offset, length);

    while (mState == STATE_SET_DATASOURCE_PENDING) {
        mCondition.wait(mLock);
    }

    return mAsyncResult;
}
```

NuPlayerDriver 调用 NuPlayer 进行异步设置数据源

setDataSource()

我们继续往下看，NuPlayerDriver 是如何设置数据源

```
status_t NuPlayerDriver::setDataSource(int fd, int64_t offset, int64_t length)
{
    ALOGV("setDataSource(%p) file(%d)", this, fd);
    Mutex::Autolock autoLock(mLock);

    if (mState != STATE_IDLE) {
        return INVALID_OPERATION;
    }

    mState = STATE_SET_DATASOURCE_PENDING;

    mPlayer->setDataSourceAsync(fd, offset, length);

    while (mState == STATE_SET_DATASOURCE_PENDING) {
        mCondition.wait(mLock);
    }

    return mAsyncResult;
}
```

NuPlayerDriver 调用 NuPlayer 进行异步设置数据源

setDataSource()

最终数据源信息会被存储在 NuPlayer 的 mSource 字段中

```
void NuPlayer::setDataSourceAsync(int fd, int64_t offset, int64_t length) {
    sp<AMessage> msg = new AMessage(kWhatSetDataSource, this);
    sp<AMessage> notify = new AMessage(kWhatSourceNotify, this);
    sp<GenericSource> source =
            new GenericSource(notify, mUIDValid, mUID, mMediaClock);
    status_t err = source->setDataSource(fd, offset, length);
    // ......
    msg->setObject("source", source);
    msg->post();
    mDataSourceType = DATA_SOURCE_TYPE_GENERIC_FD;
}
```

```
case kWhatSetDataSource:
{
    // ......
    CHECK(msg->findObject("source", &obj));
    if (obj != NULL) {
        Mutex::Autolock autoLock(mSourceLock);
        // 保存数据源到 mSource 中
        mSource = static_cast<Source *>(obj.get());
    } else {
        err = UNKNOWN_ERROR;
    }
    // ......
    break;
}
```

prepare()

对于播放非本地原始资源（未保存在应用的 res/raw/ 目录中），需要调用 prepare 函数完成准备后才可以开始播放。调用 MediaPlayer.java 的 prepare() 或 prepareAsync() 最终会走到 NuPlayer::GenericSource::onPrepareAsync() 完成数据源的准备工作。

```
if (!mUri.empty()) {
    // 如果有 URI,则根据 URI 的前缀是 http/https 还是其他,选择创建 Http 数据源还是文件数据源
} else {
    // 对于文件数据源,会优先尝试通过 media extractor 服务创建
    // 如果失败则直接通过 PlayerServiceFileSource 创建
    if (property_get_bool("media.stagefright.extractremote", true) &&
            !PlayerServiceFileSource::requiresDrm(
                    mFd.get(), mOffset, mLength, nullptr /* mime */)) {
        sp<IBinder> binder =
                defaultServiceManager()->getService(String16("media.extractor"));
        if (binder != nullptr) {
            ALOGD("FileSource remote");
            // 获取 media extractor 服务
            sp<IMediaExtractorService> mediaExService(
                    interface_cast<IMediaExtractorService>(binder));
```

prepare()

```
            sp<IDataSource> source;
            // 使用 media extractor 服务创建数据源
            mediaExService->makeIDataSource(base::unique_fd(dup(mFd.get())), mOffset, mLength,
&source);
            ALOGV("IDataSource(FileSource): %p %d %lld %lld",
                    source.get(), mFd.get(), (long long)mOffset, (long long)mLength);
            if (source.get() != nullptr) {
                // 创建成功，保存至 mDataSource 中
                mDataSource = CreateDataSourceFromIDataSource(source);
            } else {
                ALOGW("extractor service cannot make data source");
            }
        } else {
            ALOGW("extractor service not running");
        }
    }
    // 如果 media extractor 服务创建数据源失败，则使用 PlayerServiceFileSource
    if (mDataSource == nullptr) {
        ALOGD("FileSource local");
        mDataSource = new PlayerServiceFileSource(dup(mFd.get()), mOffset, mLength);
    }
}
```

完成数据源的创建后还会调用 initFromDataSource()，在其中会完成解析媒体文件的格式和轨道信息的相关工作。

start()

调用 MediaPlayer.java 的 start() 便可以开始播放了，会走到 NuPlayer::GenericSource::start() 中，调用 postReadBuffer 获取缓冲的音频与视频数据，数据由前面提到的 initFromDataSource() 所准备。

```
void NuPlayer::GenericSource::start() {
    Mutex::Autolock _l(mLock);
    ALOGI("start");
    if (mAudioTrack.mSource != NULL) {
        postReadBuffer(MEDIA_TRACK_TYPE_AUDIO);
    }
    if (mVideoTrack.mSource != NULL) {
        postReadBuffer(MEDIA_TRACK_TYPE_VIDEO);
    }
    mStarted = true;
}
```

THANK YOU !

感谢聆听