CameraDeviceImpl中CameraDeviceCallbacks回调的Binder过程

CameraDeivceImple对ICameraDeviceCallbacks回调接口的Binder创建

以CameraDeviceCallbacks为例, 在CameraDeviceImpl中:

```
// frameworks/base/core/java/android/hardware/camera2/impl/CameraDeviceImpl.java
public class CameraDeviceImpl extends CameraDevice
        implements IBinder.DeathRecipient {
    ...
    private final CameraDeviceCallbacks mCallbacks = new CameraDeviceCallbacks();
    ...
    public class CameraDeviceCallbacks extends ICameraDeviceCallbacks.Stub {
        ...
    }
    ...
}
```

CameraDeviceCallbacks是ICameraDeviceCallbacks.Stub的子类, 对于ICameraDeviceCallbacks.Stub:

```
// out/soong/.intermediates/frameworks/base/framework/android_common/gen/aidl/frameworks/av/camera/aidl/android/hardware/camera2/ICameraDeviceCallbacks.java
public interface ICameraDeviceCallbacks extends android.os.IInterface
{
    ...
    public static abstract class Stub extends android.os.Binder implements android.hardware.camera2.ICameraDeviceCallbacks
    {
        ...
    }
    ...
}
```

因此创建CameraDeviceImpl对导致android.os.Binder的构造:

```
// frameworks/base/core/java/android/os/Binder.java
public class Binder implements IBinder {
    ...
    public Binder() {
        this(null);
    }
    public Binder(@Nullable String descriptor)  {
        mObject = getNativeBBinderHolder();
        NoImagePreloadHolder.sRegistry.registerNativeAllocation(this, mObject);
        ...
    }
    ...
    private static native long getNativeBBinderHolder();
    ...
}
```

特别的, 在getNativeBBinderHolder()中:

```
// frameworks/base/core/jni/android_util_Binder.cpp
class JavaBBinderHolder
{
public:
    ...
};
static jlong android_os_Binder_getNativeBBinderHolder(JNIEnv* env, jobject clazz)
{
    JavaBBinderHolder* jbh = new JavaBBinderHolder();
    return (jlong) jbh;
}
...
static const JNINativeMethod gBinderMethods[] = {
    ...
    { "getNativeBBinderHolder", "()J", (void*)android_os_Binder_getNativeBBinderHolder },
    ...
};
```

JavaBBinderHolder::get()方法被调用时会创建JavaBBinder对象, 该对象集成了BBinder, 可以用于接受binder的消息, 这意味着只要将JavaBBinder的地址最为cookie传递给client即可告知对端自己的binder引用是什么.

CameraDeviceImple对CameraDeviceCallbacks对应的IBinder的转换与发送

以CameraDeviceCallbacks为例, 该类的Binder对象需要传递给cameraservice, 此时CameraDeviceImpl对于该Binder而言是service, 而cameraservice持有的IBinder实际上是client端的, 本文中所提及的service端和client端均是以binder为参照的.

检查打开相机部分的代码:

```
// frameworks/base/core/java/android/hardware/camera2/CameraManager.java
@SystemService(Context.CAMERA_SERVICE)
public final class CameraManager {
    ...
    private CameraDevice openCameraDeviceUserAsync(String cameraId,
            CameraDevice.StateCallback callback, Executor executor, final int uid)
            throws CameraAccessException {
        ...
        synchronized (mLock) {
            ...
            try {
                ...
                if (supportsCamera2ApiLocked(cameraId)) {
                    ...
                    cameraUser = cameraService.connectDevice(callbacks, cameraId,
                            mContext.getOpPackageName(), uid);
                } else {
                    ...
                    cameraUser = CameraDeviceUserShim.connectBinderShim(callbacks, id,
                            getDisplaySize());
                    ...
                }
            }
            ...
        }
        ...
    }
    ...
}
```

只关注cameraService.connectDevice()这个方法:

```
// out/soong/.intermediates/frameworks/base/framework-android-common/gen/aidl/frameworks/av/camera/aidl/android/hardware/ICameraService.java
```

```
// out/soong/.intermediates/frameworks/base/framework/android_common/gen/aidl/frameworks/av/camera/aidl/android/hardware/ICameraService.java
public interface ICameraService extends android.os.IInterface

{
    ...
    public static abstract class Stub extends android.os.Binder implements android.hardware.ICameraService
    {
        ...
        private static class Proxy implements android.hardware.ICameraService
        {
            ...
            @Override public android.hardware.camera2.ICameraDeviceUser connectDevice(android.hardware.camera2.ICameraDeviceCallbacks callbacks, java.lang.String cameraId, java.lang.String opPackageName, int clientUid) throws andr
            {
                ...
                try {
                    ...
                    _data.writeStrongBinder((((client!=null))?(client.asBinder()):(null)));
                    ...
                    boolean _status = mRemote.transact(Stub.TRANSACTION_connectDevice, _data, _reply, 0);
                }
                ...
            }
            ...
        }
        ...
    }
    ...
}
```

上层的cameraService.connectDevice()所对应的ICameraService实际上是ICameraService.Stub.Proxy, 在调用mRemote.transact()前, 需要构造Parcel, 并将CameraDeviceCallbacks的父类ICameraDeviceCallbacks.Stub所对应的Binder写入到Parcel中才能用于后面的transact(), 此时会会调用Java层Parcel的writeStrongBinder()方法, 将ICameraDeviceCallbacks先转换为IBinder的native指针, 然后再写入到Parcel

先看获得CameraDeviceCallbacks的IBinder的过程:

```
// out/soong/.intermediates/frameworks/base/framework/android_common/gen/aidl/frameworks/av/camera/aidl/android/hardware/ICameraService.java
public interface ICameraDeviceCallbacks extends android.os.IInterface
{
    ...
    public static abstract class Stub extends android.os.Binder implements android.hardware.camera2.ICameraDeviceCallbacks
    {
        ...
        @Override
        public android.os.IBinder asBinder() {
            return this;
        }
        ...
    }
    ...
}
```

可以看到对CameraDeviceCallbacks执行asBinder()的到的是:android.os.IBinder, 然后执行Parce.writeStrongBinder():

```
public final class Parcel {
    ...
    public final void writeStrongBinder(IBinder val) {
        nativeWriteStrongBinder(mNativePtr, val);
    }
    ...
}
```

可以看到IBinder最为一个值传到了底层, 检查底层的执行:

```
// frameworks/base/core/jni/android_os_Parcel.cpp
static void android_os_Parcel_writeStrongBinder(JNIEnv* env, jclass clazz, jlong nativePtr, jobject object)
{
    Parcel* parcel = reinterpret_cast<Parcel*>(nativePtr);
    if (parcel != NULL) {
        const status_t err = parcel->writeStrongBinder(ibinderForJavaObject(env, object));
        if (err != NO_ERROR) {
            signalExceptionForError(env, clazz, err);
        }
    }
}
...
static const JNINativeMethod gParcelMethods[] = {
    ...
    {"nativeWriteStrongBinder",   "(JLandroid/os/IBinder;)V", (void*)android_os_Parcel_writeStrongBinder},
    ...
};
```

Java层的Parcel对应着一个Native层的Parcel, 由mNativePtr引用, 查看ibinderForJavaObject对Java层IBinder的转换:

```
// frameworks/base/core/jni/android_util_Binder.cpp
sp<IBinder> ibinderForJavaObject(JNIEnv* env, jobject obj)
{
    if (obj == NULL) return NULL;

    // Instance of Binder?
    if (env->IsInstanceOf(obj, gBinderOffsets.mClass)) {
        JavaBBinderHolder* jbh = (JavaBBinderHolder*)
            env->GetLongField(obj, gBinderOffsets.mObject);
        return jbh->get(env, obj);
    }

    // Instance of BinderProxy?
    if (env->IsInstanceOf(obj, gBinderProxyOffsets.mClass)) {
        return getBPNativeData(env, obj)->mObject;
    }

    ALOGW("ibinderForJavaObject: %p is not a Binder object", obj);
    return NULL;
}
```

分两种情况, 因为传下来的object可能是service端的也可能是服务端的, 如果是service端, 那就是android.os.Binder, 如果是client端呢? 是android.os.BinderProxy, 对于该场景, 我们需要将service段的Binder传递给cameraservice, 因此执行地一个if流程, 因此会执行到JavaBBinderHolder::get():

```
// frameworks/base/core/jni/android_util_Binder.cpp
class JavaBBinderHolder
{
public:
    sp<JavaBBinder> get(JNIEnv* env, jobject obj)
    {
        AutoMutex _l(mLock);
        sp<JavaBBinder> b = mBinder.promote();
        if (b == NULL) {
            b = new JavaBBinder(env, obj);
            mBinder = b;
            ALOGV("Creating JavaBinder %p (refs %p) for Object %p, weakCount=%" PRId32 "\n",
                 b.get(), b->getWeakRefs(), obj, b->getWeakRefs()->getWeakCount());
        }

        return b;
    }
```

```
    ...
};
```

对于每个service端的进程而言, 每个BBinder都应该**当且仅当**只有一个实例, 因此此处用了单例模式, 并且在第一次执行JavaBBinderHolder::get()才会创建个BBinder的子类JavaBBinder;

而

```
// frameworks/base/core/jni/android_util_Binder.cpp
class JavaBBinder : public BBinder
{
public:
    JavaBBinder(JNIEnv* env, jobject /* Java Binder */ object)
        : mVM(jnienv_to_javavm(env)), mObject(env->NewGlobalRef(object))
    {
        ...
    }
protected:
    status_t onTransact(
        uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0) override
    {
        ...
    }
    ...
}
...
}
```

JavaBBinder要通过mObject和mVM持有对Java层对象和JavaVM的引用, 一边与, 在IPCThreadState收到binder消息时调用JavaBBinder::onTransact()方法时能正常调用到Java层的ICameraDeviceCallbacks.Stub.onTransact()

cameraserver对binder消息的接收

对于cameraservice, 其对binder消息的监听是在其启动时完成的:

```
// frameworks/av/camera/cameraserver/main_cameraserver.cpp
int main(int argc __unused, char** argv __unused)
{
    signal(SIGPIPE, SIG_IGN);

    // Set 5 threads for HIDL calls. Now cameraserver will serve HIDL calls in
    // addition to consuming them from the Camera HAL as well.
    hardware::configureRpcThreadpool(5, /*willjoin*/ false);

    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
    ALOGI("ServiceManager: %p", sm.get());
    CameraService::instantiate();
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

首先IPCThreadState::self()的执行, 单例创建了IPCThreadState类, 检查其构造函数:

```
// frameworks/native/libs/binder/IPCThreadState.cpp
IPCThreadState::IPCThreadState()
    : mProcess(ProcessState::self()),
      mWorkSource(kUnsetWorkSource),
      mPropagateWorkSource(false),
      mStrictModePolicy(0),
      mLastTransactionBinderFlags(0),
      mCallRestriction(mProcess->mCallRestriction)
{
```

```
    pthread_setspecific(gTLS, this);
    clearCaller();
    mIn.setDataCapacity(256);
    mOut.setDataCapacity(256);
    mIPCThreadStateBase = IPCThreadStateBase::self();
}
```

间接执行了ProcessState::self(), 而该方法单例模式构造了ProcessState:

```
// frameworks/native/libs/binder/ProcessState.cpp
ProcessState::ProcessState(const char *driver)
    : mDriverName(String8(driver))
    , mDriverFD(open_driver(driver))
    ...
    , mCallRestriction(CallRestriction::NONE)
{
    if (mDriverFD >= 0) {
        // mmap the binder, providing a chunk of virtual address space to receive transactions.
        mVMStart = mmap(nullptr, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE | MAP_NORESERVE, mDriverFD, 0);
        if (mVMStart == MAP_FAILED) {
            // *sigh*
            ALOGE("Using %s failed: unable to mmap transaction memory.\n", mDriverName.c_str());
            close(mDriverFD);
            mDriverFD = -1;
            mDriverName.clear();
        }
    }

    LOG_ALWAYS_FATAL_IF(mDriverFD < 0, "Binder driver could not be opened.  Terminating.");
}
```

构造函数调用了open_driver(driver), 该方法会打开**"/dev/binder"**设备:

```
// frameworks/native/libs/binder/ProcessState.cpp
static int open_driver(const char *driver)
{
    int fd = open(driver, O_RDWR | O_CLOEXEC);
    if (fd >= 0) {
        int vers = 0;
        status_t result = ioctl(fd, BINDER_VERSION, &vers);
        // 忽略检查与设置
        ...
    }
    ...
    return fd;
}
```

打开的binder驱动文件句柄被ProcessState.mDriverFD保存, 此时IPCThreadState和其mProcess所引用的ProcessState都已经建立, 该调用IPCThreadState::joinThreadPool()方法了:

```
// frameworks/native/libs/binder/IPCThreadState.cpp
void IPCThreadState::joinThreadPool(bool isMain)
{
    ...
    do {
        ...
        result = getAndExecuteCommand();
    } while (result != -ECONNREFUSED && result != -EBADF);
    ...
}
```

此时已经可以开始从binder接受消息, 那它接受到的binder是如何处理的呢?

cameraserver对binder消息中的binder引用的接收

如下步骤中, IPCThreadState负责完成binder消息的获取和处理

```
// frameworks/native/libs/binder/IPCThreadState.cpp
status_t IPCThreadState::getAndExecuteCommand()
{
    status_t result;
    int32_t cmd;

    result = talkWithDriver();
    if (result >= NO_ERROR) {
        ...
        result = executeCommand(cmd);
        ...
    }
    ...
}
```

对于获取, 通过"/dev/binder"的ioctl操作完成:

```
// frameworks/native/libs/binder/IPCThreadState.cpp
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    ...
    do {
        ...
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
            err = NO_ERROR;
        else
            err = -errno;
        ...
    } while (err == -EINTR);
    ...
}
```

对于处理, 在获取到binder消息后会调用executeCommand()进行处理, 只考虑BR_TRANSACTION的情况

```
// frameworks/native/libs/binder/IPCThreadState.cpp
status_t IPCThreadState::executeCommand(int32_t cmd)
{
    BBinder* obj;
    RefBase::weakref_type* refs;
    status_t result = NO_ERROR;

    switch ((uint32_t)cmd) {
        ...
        case BR_TRANSACTION:
        {
            ...
            if (tr.target.ptr) {
                // We only have a weak reference on the target object, so we must first try to
                // safely acquire a strong reference before doing anything else with it.
                if (reinterpret_cast<RefBase::weakref_type*>(
                        tr.target.ptr)->attemptIncStrong(this)) {
                    error = reinterpret_cast<BBinder*>(tr.cookie)->transact(tr.code, buffer,
                            &reply, tr.flags);
                    reinterpret_cast<BBinder*>(tr.cookie)->decStrong(this);
                } else {
                    error = UNKNOWN_TRANSACTION;
```

```
                }

            } else {
                error = the_context_object->transact(tr.code, buffer, &reply, tr.flags);
            }
            ...
        }
        ...
    }
    ...
}
```

tr.cookie是什么呢? 就是client端要请求的BBinder的地址, 它是当时service返回给client端的IBinder. 这个地址只能在service侧使用, 所以客户端是不会理会这个值的.

```
// 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:
            // 如果是PING_TRANSACTION就直接写reply返回给对端了
            reply->writeInt32(pingBinder());
            break;
        default:
            // 处理client的请求, 此时会调用父类的onTransact方法
            err = onTransact(code, data, reply, flags);
            break;
    }

    if (reply != nullptr) {
        reply->setDataPosition(0);
    }

    return err;
}
```

cameraserver转换CameraDeviceImpl传递过来的ICameraDeviceCallback接口对应的binder引用为接口

对于cameraserver, 此处的tr.cookie是CameraService, 直接查看CameraService的定义:

```
// frameworks/av/services/camera/libcameraservice/CameraService.h
class CameraService :
    public BinderService<CameraService>,
    public virtual ::android::hardware::BnCameraService,
    public virtual IBinder::DeathRecipient,
    public virtual CameraProviderManager::StatusListener
{
    ...
};
```

其继承自::android::hardware::BnCameraService:

```
// out/soong/.intermediates/frameworks/av/camera/libcamera_client/android_arm64_armv8-a_kryo_core_shared/gen/aidl/android/hardware/BnCameraService.h
class BnCameraService : public ::android::BnInterface<ICameraService> {
public:
  ::android::status_t onTransact(uint32_t _aidl_code, const ::android::Parcel& _aidl_data, ::android::Parcel* _aidl_reply, uint32_t _aidl_flags) override;
};
```

而BnCameraService所继承的BnInterface正是BBinder的子类:

```
// frameworks/native/libs/binder/include/binder/IInterface.h
template<typename INTERFACE>
class BnInterface : public INTERFACE, public BBinder
{
public:
    virtual sp<IInterface>      queryLocalInterface(const String16& _descriptor);
    virtual const String16&     getInterfaceDescriptor() const;

protected:
    typedef INTERFACE           BaseInterface;
    virtual IBinder*            onAsBinder();
}
```

收到binder消息按上文的流程, 先用cameraservice的CameraService::onTransact()

```
// frameworks/av/services/camera/libcameraservice/CameraService.cpp
status_t CameraService::onTransact(uint32_t code, const Parcel& data, Parcel* reply,
        uint32_t flags) {
    switch (code) {
        case SHELL_COMMAND_TRANSACTION: {
            ...
        }
    }
    return BnCameraService::onTransact(code, data, reply, flags);
};
```

而BnCameraService::onTransact()是生成的:

```
// out/soong/.intermediates/frameworks/av/camera/libcamera_client/android_arm64_armv8-a_kryo_core_shared/gen/aidl/frameworks/av/camera/aidl/android/hardware/ICameraService.cpp
::android::status_t BnCameraService::onTransact(uint32_t _aidl_code, const ::android::Parcel& _aidl_data, ::android::Parcel* _aidl_reply, uint32_t _aidl_flags) {
    ...
    switch (_aidl_code) {
        ...
        case ::android::IBinder::FIRST_CALL_TRANSACTION + 3 /* connectDevice */:
        {
            ...
            _aidl_ret_status = _aidl_data.readStrongBinder(&in_callbacks);
            ...
        }
        break;
        ...
    }
    ...
}
```

此处分两个步骤, 首先要解决从binde中创建IBinder的过程, 这个过程由Parcel::readStrongBinder()完成:

```
// frameworks/native/libs/binder/include/binder/Parcel.h
template<typename T>
status_t Parcel::readStrongBinder(sp<T>* val) const {
    sp<IBinder> tmp;
    status_t ret = readStrongBinder(&tmp);

    if (ret == OK) {
        *val = interface_cast<T>(tmp);

        if (val->get() == nullptr) {
            return UNKNOWN_ERROR;
```

```
        }
    }

    return ret;
}
```

分两部完成:

1. 读取Binder
2. 将Binder转换为接口

先看Binder的提取与创建:

```
// frameworks/native/libs/binder/Parcel.cpp
status_t Parcel::readStrongBinder(sp<IBinder>* val) const
{
    status_t status = readNullableStrongBinder(val);
    if (status == OK && !val->get()) {
        status = UNEXPECTED_NULL;
    }
    return status;
}

status_t Parcel::readNullableStrongBinder(sp<IBinder>* val) const
{
    return unflatten_binder(ProcessState::self(), *this, val);
}

status_t unflatten_binder(const sp<ProcessState>& proc,
    const Parcel& in, wp<IBinder>* out)
{
    const flat_binder_object* flat = in.readObject(false);
    if (flat) {
        switch (flat->hdr.type) {
            case BINDER_TYPE_BINDER:
                *out = reinterpret_cast<IBinder*>(flat->cookie);
                return finish_unflatten_binder(nullptr, *flat, in);
            ...
            case BINDER_TYPE_WEAK_HANDLE:
                *out = proc->getWeakProxyForHandle(flat->handle);
                return finish_unflatten_binder(
                    static_cast<BpBinder*>(out->unsafe_get()), *flat, in);
        }
    }
    return BAD_TYPE;
}
```

为什么不是`BINDER_TYPE_BINDER`? 因为在对端执行`flatten_binder()`时, 写入binder句柄时设置的是`BINDER_TYPE_WEAK_HANDLE`, 此时的binder需要转化为`BinderProxy`才能使用:

```
// frameworks/native/libs/binder/ProcessState.cpp
wp<IBinder> ProcessState::getWeakProxyForHandle(int32_t handle)
{
    ...
    handle_entry* e = lookupHandleLocked(handle);
    if (e != nullptr) {
        IBinder* b = e->binder;
        // 首次接受一个binder引用是, 一般这里都是空的, 否则会缓存住
        if (b == nullptr || !e->refs->attemptIncWeak(this)) {
            // 创建了一个BpBinder, 稍后这个BpBinder会被用于创建BpInterface
            b = BpBinder::create(handle);
            result = b;
```

```
            e->binder = b;
            if (b) e->refs = b->getWeakRefs();
        } else {
            result = b;
            e->refs->decWeak(this);
        }
    }
    return result;
};
```

此时通过unflatten_binder()的到了一个sp<IBinder>>, 也就是新创建的BpBinder, 然后看看在调用CameraService::connectDevice()钱, BpBinder是如何转化为BpCameraDeviceCallbacks并作为参数向下传递的, 继续上文的interface_cast<T>(tmp):

```
// frameworks/native/libs/binder/include/binder/IInterface.h
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}
```

对接口ICameraDeviceCallbacks:

```
inline sp<ICameraDeviceCallbacks> interface_cast(const sp<IBinder>& obj)
{
    return ICameraDeviceCallbacks::asInterface(obj);
}
```

查看ICameraDeviceCallbacks::asInterface()的声明:

```
class ICameraDeviceCallbacks : public ::android::IInterface {
public:
    DECLARE_META_INTERFACE(CameraDeviceCallbacks)
    ...
}
```

从DECLARE_META_INTERFACE(CameraDeviceCallbacks)展开:

```
class ICameraDeviceCallbacks : public ::android::IInterface {
public:
    static const ::android::String16 descriptor;
    static ::android::sp<ICameraDeviceCallbacks> asInterface(
            const ::android::sp<::android::IBinder>& obj);
    virtual const ::android::String16& getInterfaceDescriptor() const;  \
    ICameraDeviceCallbacks();
    virtual ~ICameraDeviceCallbacks();
    static bool setDefaultImpl(std::unique_ptr<ICameraDeviceCallbacks> impl);     \
    static const std::unique_ptr<ICameraDeviceCallbacks>& getDefaultImpl();       \
private:
    static std::unique_ptr<ICameraDeviceCallbacks> default_impl;
public:
...
}
```

查看ICameraDeviceCallbacks::asInterface()的实现:

```
// out/soong/.intermediates/frameworks/av/camera/libcamera_client/android_arm64_armv8-a_kryo_core_shared/gen/aidl/android/hardware/camera2/ICameraDeviceCallbacks.h
class ICameraDeviceCallbacks : public ::android::IInterface {
public:
    DECLARE_META_INTERFACE(CameraDeviceCallbacks)
```

```
    ...
}
```

从DECLARE_META_INTERFACE(CameraDeviceCallbacks)展开:

```
// out/soong/.intermediates/frameworks/av/camera/libcamera_client/android_arm64_armv8-a_kryo_core_shared/gen/aidl/frameworks/av/camera/aidl/android/hardware/camera2/ICameraDeviceCallbacks.cpp
IMPLEMENT_META_INTERFACE(CameraDeviceCallbacks, "android.hardware.camera2.ICameraDeviceCallbacks")
```

从IMPLEMENT_META_INTERFACE(CameraDeviceCallbacks)展开:

```
#define IMPLEMENT_META_INTERFACE(CameraDeviceCallbacks, NAME)
    const ::android::String16 ICameraDeviceCallbacks::descriptor(NAME);
    const ::android::String16&
            ICameraDeviceCallbacks::getInterfaceDescriptor() const {
        return ICameraDeviceCallbacks::descriptor;
    }
    ::android::sp<ICameraDeviceCallbacks> ICameraDeviceCallbacks::asInterface(
            const ::android::sp<::android::IBinder>& obj)
    {
        ::android::sp<ICameraDeviceCallbacks> intr;
        if (obj != nullptr) {
            intr = static_cast<ICameraDeviceCallbacks*>(
                obj->queryLocalInterface(
                        ICameraDeviceCallbacks::descriptor).get());
            if (intr == nullptr) {
                intr = new BpCameraDeviceCallbacks(obj);
            }
        }
        return intr;
    }
    std::unique_ptr<ICameraDeviceCallbacks> ICameraDeviceCallbacks::default_impl;
    bool ICameraDeviceCallbacks::setDefaultImpl(std::unique_ptr<ICameraDeviceCallbacks> impl)\
    {
        if (!ICameraDeviceCallbacks::default_impl && impl) {
            ICameraDeviceCallbacks::default_impl = std::move(impl);
            return true;
        }
        return false;
    }
    const std::unique_ptr<ICameraDeviceCallbacks>& ICameraDeviceCallbacks::getDefaultImpl() \
    {
        return ICameraDeviceCallbacks::default_impl;
    }
    ICameraDeviceCallbacks::ICameraDeviceCallbacks() { }
    ICameraDeviceCallbacks::~ICameraDeviceCallbacks() { }
```

对于远程的BpBinder, obj->queryLocalInterface()返回false, 因此ICameraDeviceCallbacks::asInterface()的到是:BpCameraDeviceCallbacks

此时继续执行BnCameraService::onTransact(), 将转换的BpCameraDeviceCallbacks传递给CameraService:

```
// out/soong/.intermediates/frameworks/av/camera/libcamera_client/android_arm64_armv8-a_kryo_core_shared/gen/aidl/frameworks/av/camera/aidl/android/hardware/ICameraService.cpp
::android::status_t BnCameraService::onTransact(uint32_t _aidl_code, const ::android::Parcel& _aidl_data, ::android::Parcel* _aidl_reply, uint32_t _aidl_flags) {
    ...
    switch (_aidl_code) {
        ...
        case ::android::IBinder::FIRST_CALL_TRANSACTION + 3 /* connectDevice */:
        {
            ...
            _aidl_ret_status = _aidl_data.readStrongBinder(&in_callbacks);
            ...
            ::android::binder::Status _aidl_status(connectDevice(in_callbacks, in_cameraId, in_opPackageName, in_clientUid, &_aidl_return));
```

```
            ...
        }
        break;
        ...
    }
    ...
}
```

然后才是熟悉的CameraService::connectDevice()方法:

```
// frameworks/av/services/camera/libcameraservice/CameraService.cpp
Status CameraService::connectDevice(
        const sp<hardware::camera2::ICameraDeviceCallbacks>& cameraCb,
        const String16& cameraId,
        const String16& clientPackageName,
        int clientUid,
        /*out*/
        sp<hardware::camera2::ICameraDeviceUser>* device) {
    ...
    ret = connectHelper<hardware::camera2::ICameraDeviceCallbacks,CameraDeviceClient>(cameraCb, id,
            /*api1CameraId*/-1,
            CAMERA_HAL_API_VERSION_UNSPECIFIED, clientPackageName,
            clientUid, USE_CALLING_PID, API_2, /*shimUpdateOnly*/ false, /*out*/client);
    ...
}

template<class CALLBACK, class CLIENT>
Status CameraService::connectHelper(const sp<CALLBACK>& cameraCb, const String8& cameraId,
        int api1CameraId, int halVersion, const String16& clientPackageName, int clientUid,
        int clientPid, apiLevel effectiveApiLevel, bool shimUpdateOnly,
        /*out*/sp<CLIENT>& device) {
    ...
    {
        ...
        if(!(ret = makeClient(this, cameraCb, clientPackageName,
                cameraId, api1CameraId, facing,
                clientPid, clientUid, getpid(),
                halVersion, deviceVersion, effectiveApiLevel,
                /*out*/&tmp)).isOk()) {
            return ret;
        }
        ...
    }
    ...
}
...
Status CameraService::makeClient(const sp<CameraService>& cameraService,
        const sp<IInterface>& cameraCb, const String16& packageName, const String8& cameraId,
        int api1CameraId, int facing, int clientPid, uid_t clientUid, int servicePid,
        int halVersion, int deviceVersion, apiLevel effectiveApiLevel,
        /*out*/sp<BasicClient>* client) {
    ...
    if (halVersion < 0 || halVersion == deviceVersion) {
        switch(deviceVersion) {
            ...
            case CAMERA_DEVICE_API_VERSION_3_0:
            case CAMERA_DEVICE_API_VERSION_3_1:
            case CAMERA_DEVICE_API_VERSION_3_2:
            case CAMERA_DEVICE_API_VERSION_3_3:
            case CAMERA_DEVICE_API_VERSION_3_4:
            case CAMERA_DEVICE_API_VERSION_3_5:
                if (effectiveApiLevel == API_1) { // Camera1 API route
                    sp<ICameraClient> tmp = static_cast<ICameraClient*>(cameraCb.get());
```

```
                    *client = new Camera2Client(cameraService, tmp, packageName,
                            cameraId, api1CameraId,
                            facing, clientPid, clientUid,
                            servicePid);
                } else { // Camera2 API route
                    sp<hardware::camera2::ICameraDeviceCallbacks> tmp =
                            static_cast<hardware::camera2::ICameraDeviceCallbacks*>(cameraCb.get());
                    *client = new CameraDeviceClient(cameraService, tmp, packageName, cameraId,
                            facing, clientPid, clientUid, servicePid);
                }
                break;
    } else {
        ...
    }
    ...
}
```

由于本文的假设是API2 -> Camera HAL3, 因此, 关注CameraDeviceClient的构造:

```
// frameworks/av/services/camera/libcameraservice/api2/CameraDeviceClient.cpp
CameraDeviceClient::CameraDeviceClient(const sp<CameraService>& cameraService,
        const sp<hardware::camera2::ICameraDeviceCallbacks>& remoteCallback,
        const String16& clientPackageName,
        const String8& cameraId,
        int cameraFacing,
        int clientPid,
        uid_t clientUid,
        int servicePid) :
    Camera2ClientBase(cameraService, remoteCallback, clientPackageName,
                cameraId, /*API1 camera ID*/ -1,
                cameraFacing, clientPid, clientUid, servicePid),
        mInputStream(),
        mStreamingRequestId(REQUEST_ID_NONE),
        mRequestIdCounter(0) {

    ATRACE_CALL();
    ALOGI("CameraDeviceClient %s: Opened", cameraId.string());
}
```

ICameraDeviceCallbacks作为参数传递给了Camera2ClientBase的构造函数:

```
// frameworks/av/services/camera/libcameraservice/common/Camera2ClientBase.cpp
template <typename TClientBase>
Camera2ClientBase<TClientBase>::Camera2ClientBase(
        const sp<CameraService>& cameraService,
        const sp<TCamCallbacks>& remoteCallback,
        const String16& clientPackageName,
        const String8& cameraId,
        int api1CameraId,
        int cameraFacing,
        int clientPid,
        uid_t clientUid,
        int servicePid):
        TClientBase(cameraService, remoteCallback, clientPackageName,
                cameraId, api1CameraId, cameraFacing, clientPid, clientUid, servicePid),
        mSharedCameraCallbacks(remoteCallback),
        mDeviceVersion(cameraService->getDeviceVersion(TClientBase::mCameraIdStr)),
        mDevice(new Camera3Device(cameraId)),
        mDeviceActive(false), mApi1CameraId(api1CameraId)
{
    ALOGI("Camera %s: Opened. Client: %s (PID %d, UID %d)", cameraId.string(),
            String8(clientPackageName).string(), clientPid, clientUid);
```

```
    mInitialClientPid = clientPid;
    LOG_ALWAYS_FATAL_IF(mDevice == 0, "Device should never be NULL here.");
}
```

ICameraDeviceCallbacks设置给了Camera2ClientBase的mSharedCameraCallbacks成员的mRemoteCallback中, 然后继续传递ICameraDeviceCallbacks到TClientBase中, 这个TClientBase对于本文而言是CameraDeviceClientBase:

```
// frameworks/av/services/camera/libcameraservice/api2/CameraDeviceClient.cpp
CameraDeviceClientBase::CameraDeviceClientBase(
        const sp<CameraService>& cameraService,
        const sp<hardware::camera2::ICameraDeviceCallbacks>& remoteCallback,
        const String16& clientPackageName,
        const String8& cameraId,
        int api1CameraId,
        int cameraFacing,
        int clientPid,
        uid_t clientUid,
        int servicePid) :
    BasicClient(cameraService,
            IInterface::asBinder(remoteCallback),
            clientPackageName,
            cameraId,
            cameraFacing,
            clientPid,
            clientUid,
            servicePid),
    mRemoteCallback(remoteCallback) {
    // We don't need it for API2 clients, but Camera2ClientBase requires it.
    (void) api1CameraId;
}
```

此处的mRemoteCallback方法保存了sp<ICameraDeviceCallback>也就是BpCameraDeviceCallback.

cameraservice对CameraDeviceImpl的通知

在设备准备完成后, cameraservice负责通过CameraDeviceClient::notifyPrepared()通知应用:

```
// frameworks/av/services/camera/libcameraservice/api2/CameraDeviceClient.cpp
void CameraDeviceClient::notifyPrepared(int streamId) {
    // Thread safe. Don't bother locking.
    sp<hardware::camera2::ICameraDeviceCallbacks> remoteCb = getRemoteCallback();
    if (remoteCb != 0) {
        remoteCb->onPrepared(streamId);
    }
}
```

此时执行到BpCameraDeviceCallbacks::onPrepared():

```
// out/soong/.intermediates/frameworks/av/camera/libcamera_client/android_arm64_armv8-a_kryo_core_shared/gen/aidl/frameworks/av/camera/aidl/android/hardware/camera2/ICameraDeviceCallbacks.cpp
::android::binder::Status BpCameraDeviceCallbacks::onPrepared(int32_t streamId) {
    ...
    _aidl_ret_status = remote()->transact(::android::IBinder::FIRST_CALL_TRANSACTION + 4 /* onPrepared */, _aidl_data, &_aidl_reply, ::android::IBinder::FLAG_ONEWAY);
    ...
}
```

此处的remote()返回的是BpBinder, 因此继续调用BpBinder::transact()通过binder发送数据

CameraDeviceImpl所在App对binder消息的接收

CameraDeviceImpl运行在Camera 2中, 属于App的上下文, 而App在启动完成后, 其RuntimeInit类的main()方法被调用, 继而RuntimeInit.nativeFinishInit()方法也会被调用:

```
// frameworks/base/core/java/com/android/internal/os/RuntimeInit.java
public class RuntimeInit {
    ...
    public static final void main(String[] argv) {
        ...
        commonInit();
        nativeFinishInit();
        ...
    }
    ...
}
```

此时App可以处理Binder消息.

查看实现:

```
// frameworks/base/core/jni/AndroidRuntime.cpp
static AndroidRuntime* gCurRuntime = NULL;
static void com_android_internal_os_RuntimeInit_nativeFinishInit(JNIEnv* env, jobject clazz)
{
    gCurRuntime->onStarted();
}
```

由此AndroidRuntime::onStarted()被执行, 而AndroidRuntime的子类是存活与当前App中的AppRuntime:

```
// frameworks/base/cmds/app_process/app_main.cpp
class AppRuntime : public AndroidRuntime
{
public:
    ...
    virtual void onStarted()
    {
        sp<ProcessState> proc = ProcessState::self();
        ALOGV("App process: starting thread pool.\n");
        proc->startThreadPool();
        // 正常不应当执行到此, 略去.
        ...
    }
    ...
};
```

接下来与上文camearservice接收binder消息的流程相同, 略去

CameraDeviceImpl处理cameraserver的binder回调

而对于CameraDeviceImpl而言, 是此处的BBinder是JavaBBinder, 这是在传递ICameraDeviceCallbacks.Stub时从底层的JavaBBinderHolder创建的, 因此查看代码:

```
// frameworks/base/core/jni/android_util_Binder.cpp
class JavaBBinder : public BBinder
{
public:
    ...
protected:
    ...
    status_t onTransact(
        uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0) override
    {
        ...
        jboolean res = env->CallBooleanMethod(mObject, gBinderOffsets.mExecTransact,
            code, reinterpret_cast<jlong>(&data), reinterpret_cast<jlong>(reply), flags);
```

```
        ...
    }
    ...
};
```

CallBooleanMethod()直接调用到了Java层的Binder的execTransact()方法, 这点从一下代码即可看出:

```
// frameworks/base/core/jni/android_util_Binder.cpp
static int int_register_android_os_Binder(JNIEnv* env)
{
    jclass clazz = FindClassOrDie(env, kBinderPathName);

    gBinderOffsets.mClass = MakeGlobalRefOrDie(env, clazz);
    gBinderOffsets.mExecTransact = GetMethodIDOrDie(env, clazz, "execTransact", "(IJJI)Z");
    ...
}
```

重新回到Binder的Java层:

```
// frameworks/base/core/java/android/os/Binder.java
public class Binder implements IBinder {
    ...
    private boolean execTransact(int code, long dataObj, long replyObj,
            int flags) {
        ...
        try {
            return execTransactInternal(code, dataObj, replyObj, flags, callingUid);
        } finally {
            ThreadLocalWorkSource.restore(origWorkSource);
        }
    }
    private boolean execTransactInternal(int code, long dataObj, long replyObj, int flags,
            int callingUid) {
        ...
        try {
            if (tracingEnabled) {
                final String transactionName = getTransactionName(code);
                Trace.traceBegin(Trace.TRACE_TAG_ALWAYS, getClass().getName() + ":"
                        + (transactionName != null ? transactionName : code));
            }
            res = onTransact(code, data, reply, flags);
        } catch (RemoteException|RuntimeException e) {
            ...
        }
        ...
    }
    ...
};
```

onTransact()是谁的子类呢? 一般谁继承了android.os.Binder, 谁就是子类, 以CameraDeviceCallback为例:

```
// out/soong/.intermediates/frameworks/base/framework/android_common/gen/aidl/frameworks/av/camera/aidl/android/hardware/camera2/ICameraDeviceCallbacks.java
public interface ICameraDeviceCallbacks extends android.os.IInterface {
    ...
    public static abstract class Stub extends android.os.Binder implements android.hardware.camera2.ICameraDeviceCallbacks
    {
        @Override public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags) throws android.os.RemoteException
        {
            java.lang.String descriptor = DESCRIPTOR;
            switch (code)
```

```
            {
                ...
                case TRANSACTION_onPrepared:
                {
                    ...
                    long _arg1;
                    _arg1 = data.readLong();
                    this.onPrepared(_arg0, _arg1);
                    return true;
                }
                ...
            }
        }
    }
    ...
}
```

最后检查ICameraDeviceCallbacks.Stub的实际拥有者:

```
// frameworks/base/core/java/android/hardware/camera2/impl/CameraDeviceImpl.java
public class CameraDeviceImpl extends CameraDevice
        implements IBinder.DeathRecipient {
    public class CameraDeviceCallbacks extends ICameraDeviceCallbacks.Stub {
        ...
        @Override
        public void onPrepared(final CaptureResultExtras resultExtras, final long timestamp) {
            ...
        }
        ...
    }
    ...
}
```