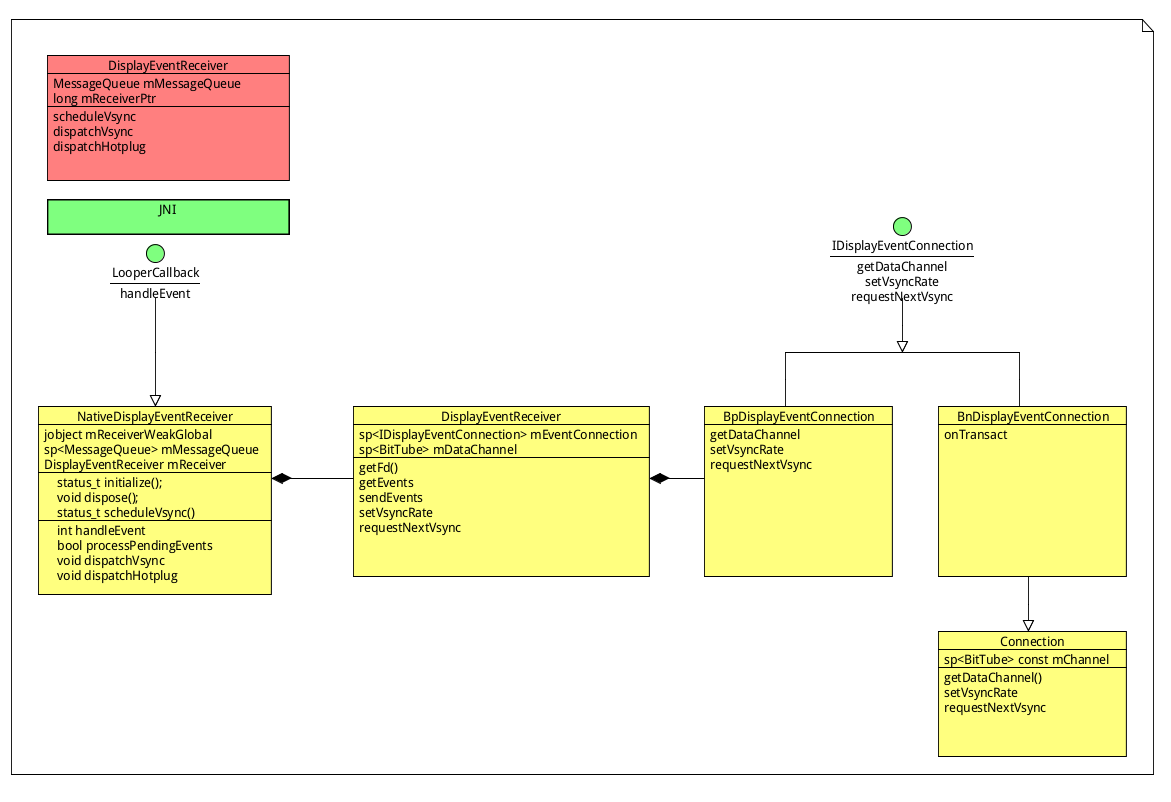
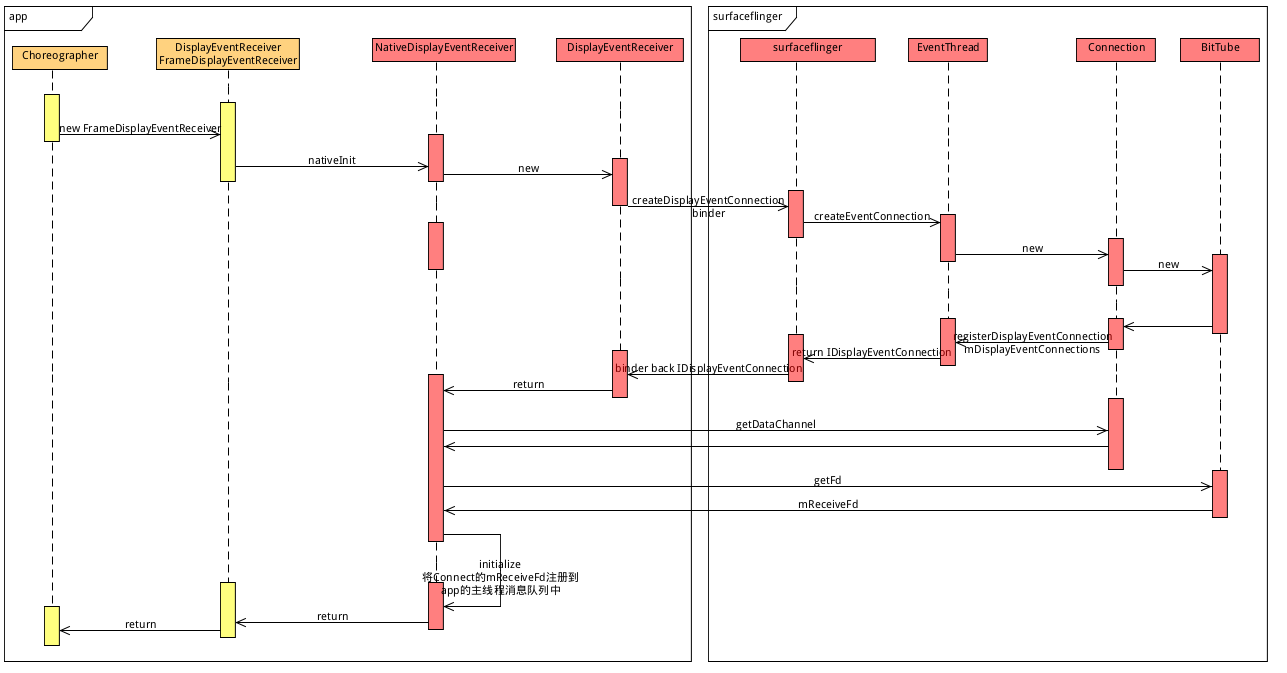
app 端和surfaceflinger之间的通信

1. 通信的建立





在java层的DisplayEventReceiver构造函数中调用nativeInit函数将this weak引用和messageQueue传送给JNI层。

在JNI层new NativeDisplayEventReceiver对象，同时创建DisplayEventReceiver对象，在DisplayEventReceiver的构造函数中通过surfaceflinger来创建IDisplayEventConnection的代理端，同时调用getDataChannel函数来创建对应于IDisplayEventConnection中的BitTube对象。

使用创建的BitTube对象获取mReceiveFd 文件描述符和服务端进行通信。

Bp通过Parcel来创建BitTube对象。

class BpDisplayEventConnection : public BpInterface<IDisplayEventConnection>

{ virtual sp<BitTube> getDataChannel() const

{

Parcel data, reply;

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

remote()->transact(GET\_DATA\_CHANNEL, data, &reply);

return new BitTube(reply);

}

}

BitTube::BitTube(const Parcel& data)

: mSendFd(-1), mReceiveFd(-1)

{

**//读取从binder中传输过来的Parcel对象中读取fd，同时dup该fd给mReceiveFd**

int data\_fd = data.readFileDescriptor();

mReceiveFd = dup(data\_fd);

if (mReceiveFd < 0) {

mReceiveFd = -errno;

}

}

status\_t BnDisplayEventConnection::onTransact(

uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t flags)

{

switch(code) {

case GET\_DATA\_CHANNEL: {

CHECK\_INTERFACE(IDisplayEventConnection, data, reply);

**//调用EventThread::Connection中的getDataChannel函数来返回BitTube对象**

sp<BitTube> channel(getDataChannel());

channel->writeToParcel(reply);

return NO\_ERROR;

}

}

**//这个是服务端的BitTube对象**

sp<BitTube> const mChannel

sp<BitTube> EventThread::Connection::getDataChannel() const {

return mChannel;

}

**//服务端使用BitTube的writeToParcel函数将mReceiveFd 描述符写入到Parcel中传送给bp端**

**//同时关闭服务端域套接字的mReceiveFd 文件描述符**

status\_t BitTube::writeToParcel(Parcel\* reply) const

{

if (mReceiveFd < 0)

return -EINVAL;

status\_t result = reply->writeDupFileDescriptor(mReceiveFd);

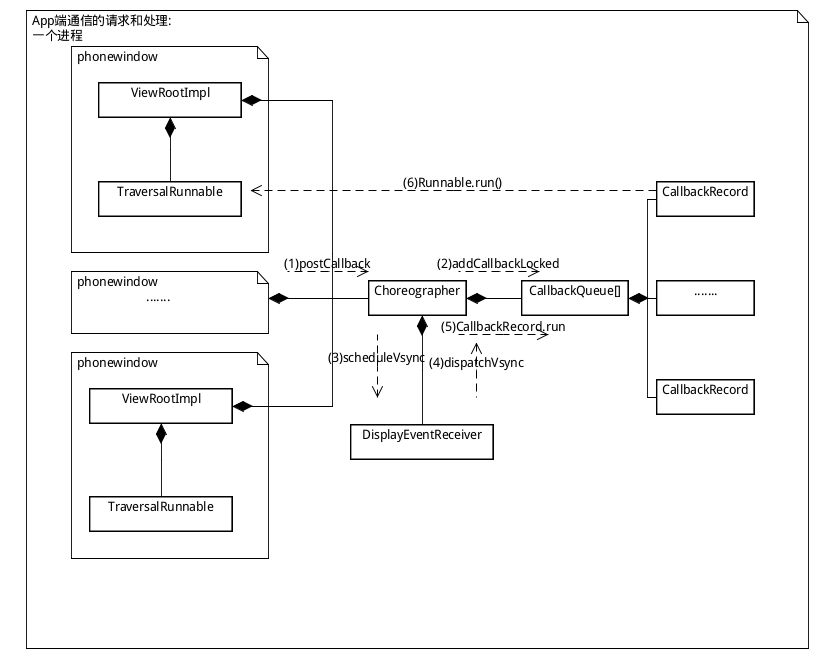
close(mReceiveFd);

mReceiveFd = -1;

return result;

}

1. App端通信的请求和处理



Native 层对app请求vsync信号的处理：

status\_t NativeDisplayEventReceiver::scheduleVsync() {

**//首先检查上一个请求vsync信号的动作是否完成接收**

**//没有的情况不发送请求**

if (!mWaitingForVsync) {

nsecs\_t vsyncTimestamp;

int32\_t vsyncDisplayId;

uint32\_t vsyncCount;

**//请求之前将域套接字中的数据全部处理完成**

**//由于socket都为O\_NONBLOCK非阻塞所以没有数据不会阻塞线程**

processPendingEvents(&vsyncTimestamp, &vsyncDisplayId, &vsyncCount);

**//发送vsync请求**

status\_t status = mReceiver.requestNextVsync();

if (status) {

ALOGW("Failed to request next vsync, status=%d", status);

return status;

}

mWaitingForVsync = true;

}

return OK;

}

最终vsync信号请求到达surfaceflinger中。

status\_t DisplayEventReceiver::requestNextVsync() {

if (mEventConnection != NULL) {

mEventConnection->requestNextVsync();

return NO\_ERROR;

}

return NO\_INIT;

}

void EventThread::Connection::requestNextVsync() {

mEventThread->requestNextVsync(this);

}

void EventThread::requestNextVsync(

const sp<EventThread::Connection>& connection) {

Mutex::Autolock \_l(mLock);

if (connection->count < 0) {

connection->count = 0;

**//唤醒eventthread线程**

mCondition.broadcast();

}

}

1. Surfaceflinger端vsync 信号请求的处理