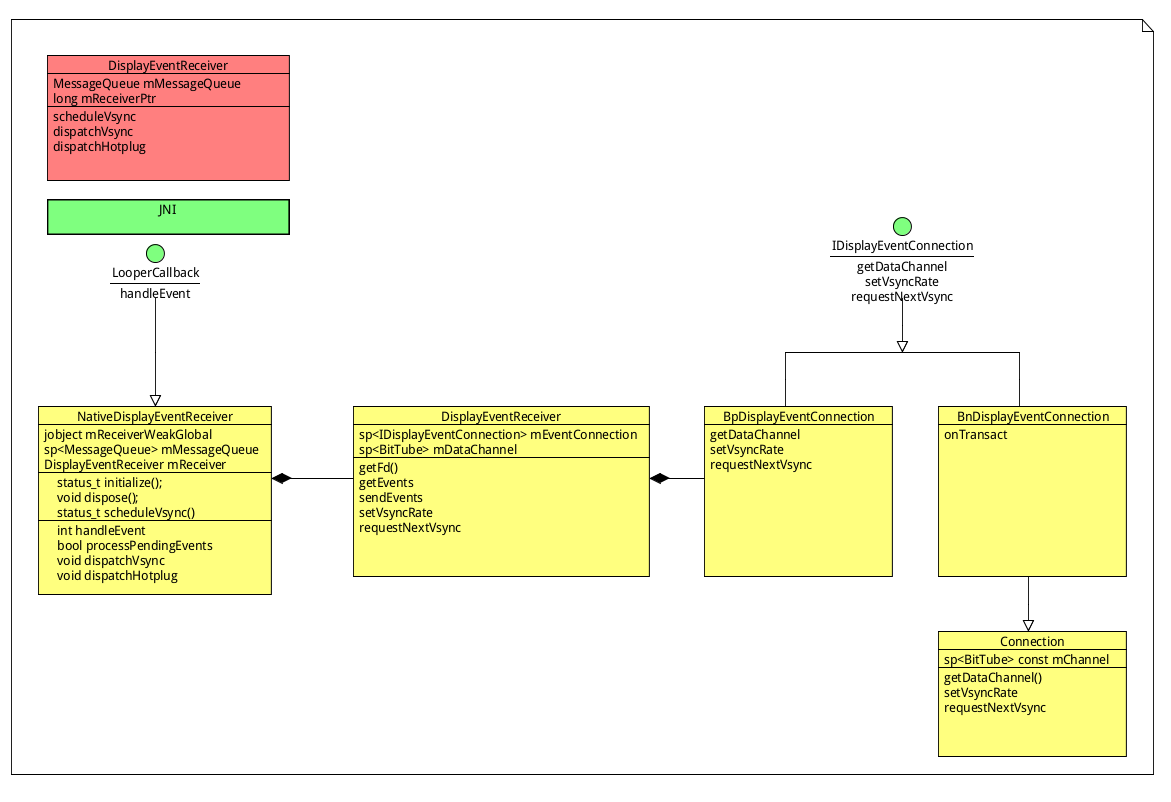
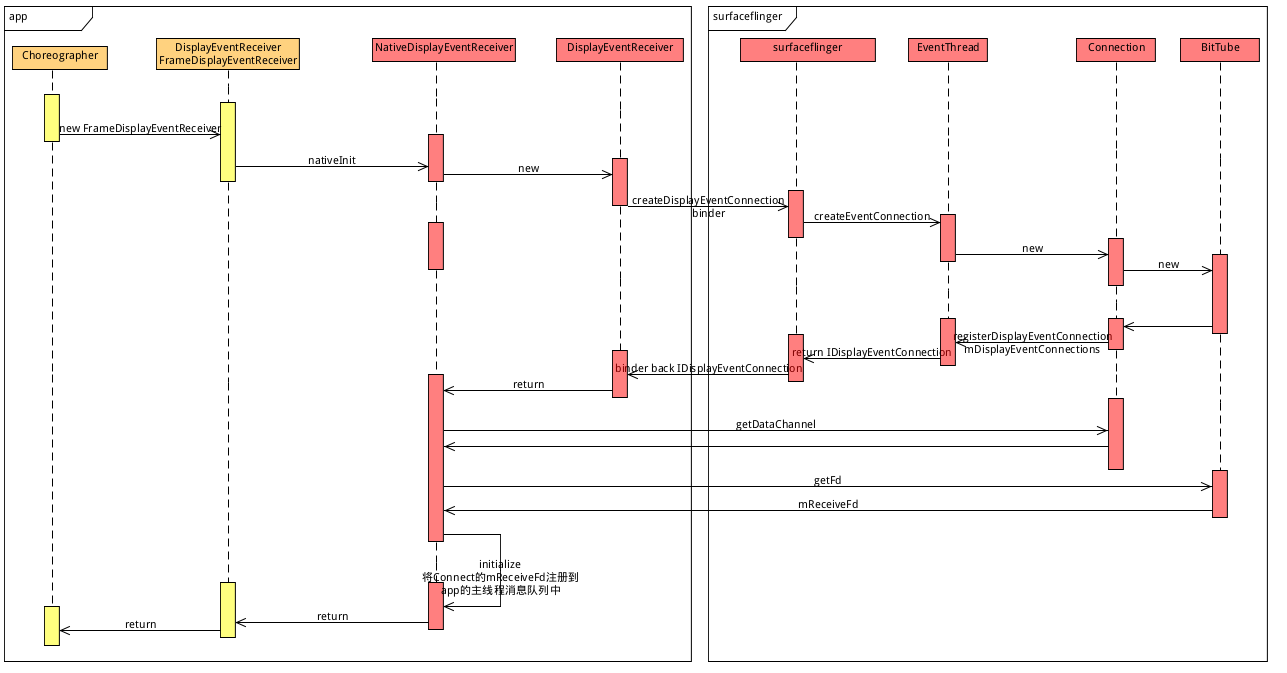
app 端和surfaceflinger之间的通信

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);

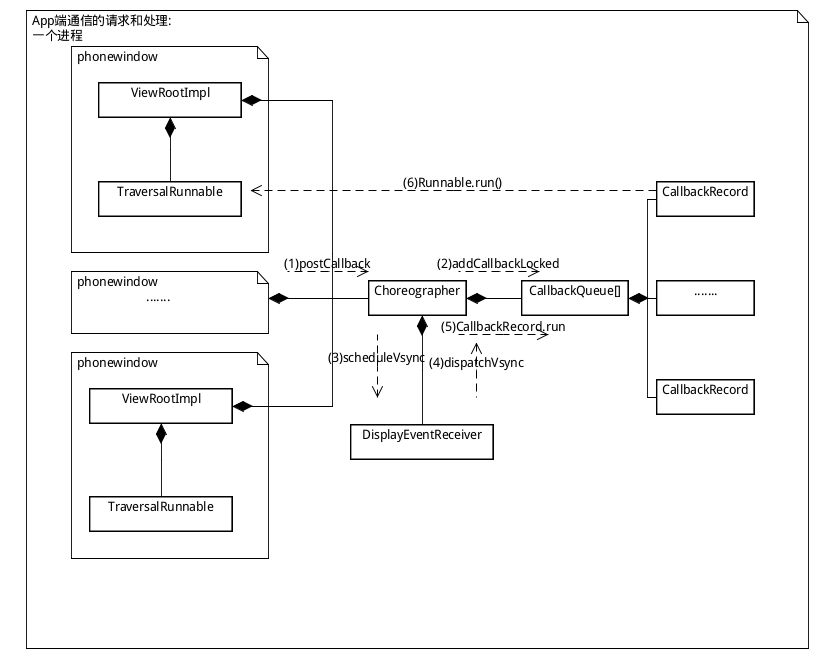
close(mReceiveFd);

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

获取需要请求vsync的connection，向其发送vsync event。

bool EventThread::threadLoop() {

DisplayEventReceiver::Event event;

Vector< sp<EventThread::Connection> > signalConnections;

**//通过waitForEvent函数来生成需要发送vsync信号的app connect进行发送event**

signalConnections = waitForEvent(&event);

// dispatch events to listeners...

const size\_t count = signalConnections.size();

for (size\_t i=0 ; i<count ; i++) {

const sp<Connection>& conn(signalConnections[i]);

status\_t err = conn->postEvent(event);

}

**//说明该线程循环做上面的事情**

return true;

}

waitForEvent函数详解：

**do{**

**1：检查是否有vsync信号过来，如果有的话取vsync信号的timestamp，**

**2：遍历所有的connection，如果connection->count大于等于0，说明该**

**Connection请求vsync信号，将waitForVSync = true，**

**2.1如果timestamp不为0，那么将该connection加入到可以发送vsync**

**信号的connection集合中。**

**3：检查timestamp和waitForVSync条件：**

**3.1如果timestamp && !waitForVSync那么将该eventThread从**

**DispSyncThread的callback集合中移除。**

**3.2如果!timestamp && waitForVSync那么将eventThread加入到**

**DispSyncThread的callback集合中。**

**4：当timestamp=0，**

**4.1 waitForVSync为true说明有app 请求vsync信号，进行waitRelative**

**Timeout等待，当有vsync信号到达直接返回，当timeout发生fake**

**一个vsync信号。**

**4.2waitForVSync不为true，线程进入mCondition.wait。**

**}while(是否有可以发送vsync信号的connection)**

app-eventThread 线程的几个状态：

1.没有vsync和requstvsync，进入4.2步状态等待broadcast事件。

2.当有client请求requestNextVsync，进入2步将waitForVSync = true，同时进入3.2步开

始接收vsync信号，进入4.1步进行timeout等待。

3.当有vsync信号到达，onVSyncEvent唤醒timeout等待，重新开始do while循环进入1

步获取vsync信号的timestamp后进入2.1步将当前connection加入到signalConnections，

退 出do while循环。

4.connection 的vsync信号处理完之后重新进入4.2步等待broadcast事件。

5.当有vsync信号到达唤醒等待，重新循环进入1步，获取vsync信号的timestamp，同时

进入3.1步将该eventThread从DispSyncThread的callback集合中移除，同时进入4.2步等

待broadcast事件

