内容目录

[1 窗口类图结构 1](#__RefHeading___Toc240_2075698673)

[2 各窗口对象的新建过程 4](#__RefHeading___Toc242_2075698673)

[2.1 PhonwWindow 以及WindowManagerImpl新建流程 4](#__RefHeading___Toc244_2075698673)

[2.2新建DecorView对象 5](#__RefHeading___Toc246_2075698673)

[2.3 新建ViewRootImpl 连接wms 以及通知WMS新建WindowState 5](#__RefHeading___Toc248_2075698673)

[2.4 Activity的appToken以及WMS的WindowToken新建 6](#__RefHeading___Toc1075_998579151)

[3 WMS为每个app通知SurfaceFlinger新建Client对象的流程 6](#__RefHeading___Toc1077_998579151)

[3.1 新建SurfaceSession对象 6](#__RefHeading___Toc1079_998579151)

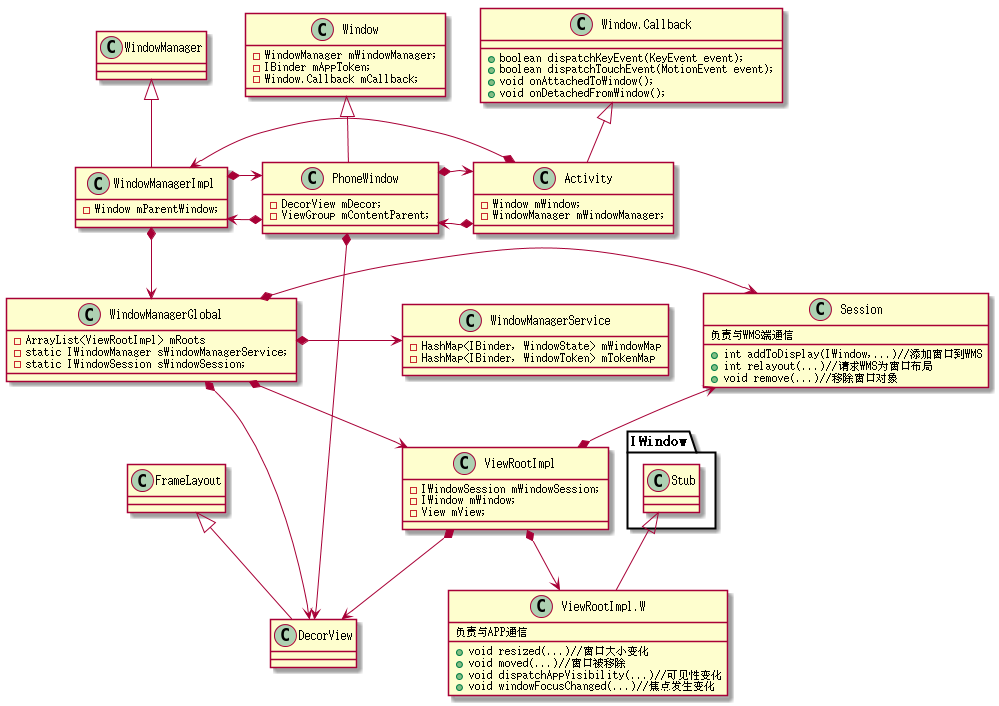
[3.2 native层的surfaceFlinger连接过程 7](#__RefHeading___Toc1081_998579151)

[3.3 app端如何获得SurfaceFlinger分配的绘图缓存 8](#__RefHeading___Toc1083_998579151)

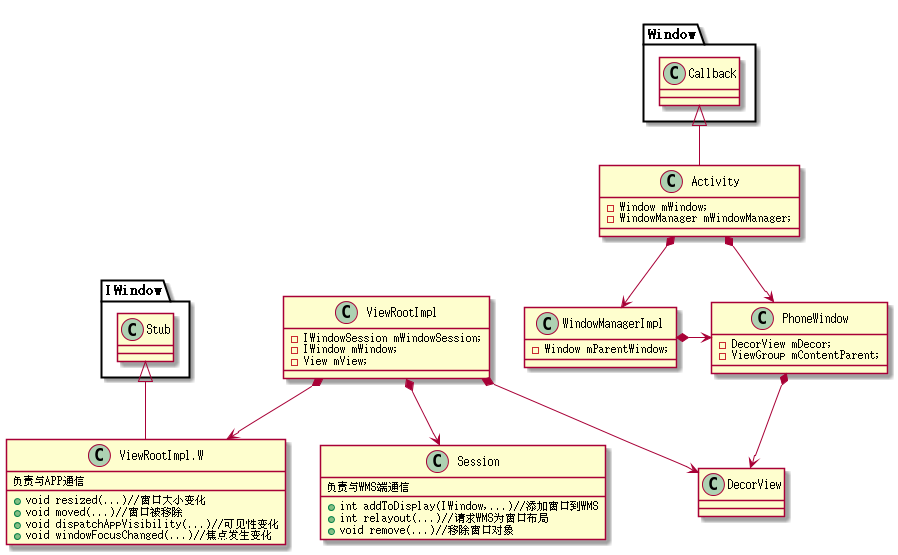
[3.4 Surface绘制过程 13](#__RefHeading___Toc1095_1857248225)

[3.5 lockCanvas与unlockCanvas的底层操作 15](#__RefHeading___Toc1093_1857248225)

**1 窗口类图结构**



核心类图如下



1 PhoneWindow是Activity中维护窗口属性的类，其中有一个DecorView成员变量是Activity最顶层的一个控件，其他控件都是这个控件的子控件

2 WindowManagerImpl是窗口的管理类，负责管理窗口的状态，这个类非常简单，仅仅是提供了添加View以及移除View还有更新View布局的操作（对于Activity这里的view就是指的DecorView）,

而这些接口的实现也并不是由WindowManagerImpl完成，WindowManagerImpl仅仅是一个代理，他将这些操作全部转交给了WindowMangerGlobal

3 WindowMangerGlobal是一个单例类，这个类维护了APP窗口的列表，每个窗口会对应在WindowMangerGlobal有一个ViewRootImpl与窗口对应，同时，这个WindowMangerGlobal也是获得与WMS会话连接的地方

4 ViewRootImpl才是真正维护窗口状态的地方，每当一个新的窗口添加到WindowMangerGlobal管理的时候，会生成一个ViewRootImpl来维护窗口状态，同时如果还没有与WMS连接的话，则会在生成第一个ViewRootImpl对象后会立马建立与WMS的会话连接，

通过WMS的openSession函数获得Session代理对象，然后通过这个对象通知WMS创建一个WindowState对象维护这个窗口的状态，并且新建一个ViewRootImpl.W对象并添加到WMS,这个对象是WMS用来与app窗口进行对话的对象，当窗口状态发生变化，例如窗口大小变化，窗口焦点变化等等，都是通过这个对象通知app.

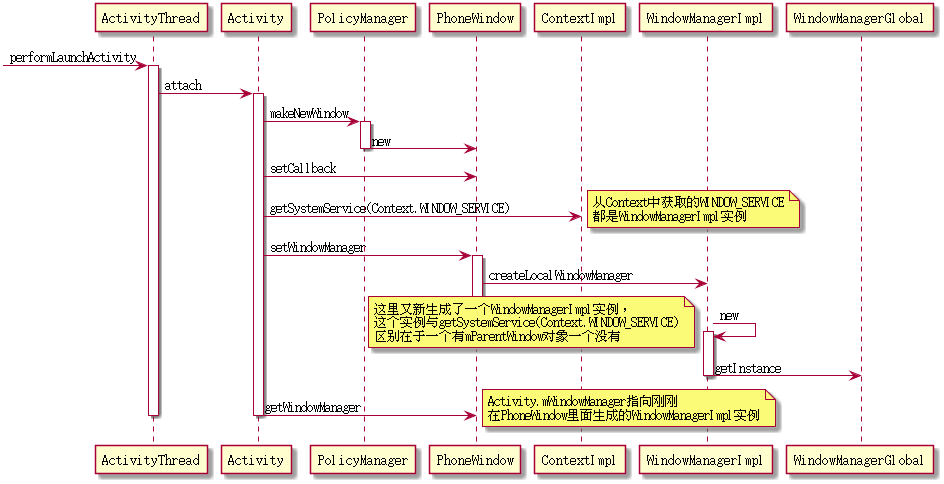
WindowState新建的同时，WMS会为APP建立一个与SurfaceFlinger的链接，SurfaceFlinger对应生成一个Client对象来维护app窗口的绘制

5 当app收到wms通知的窗口状态变化通知需要重绘或者自身申请重绘的时候，ViewRootImpl会调用performTraversals函数进行绘制，绘制事件的分发由此而始。

每次绘制的时候，app会向WMS申请重新布局，WMS会为app对应的窗口向SurfaceFlinger申请一段绘图缓存，然后写回给ViewRootImpl的Surface对象。

**2 各窗口对象的新建过程**

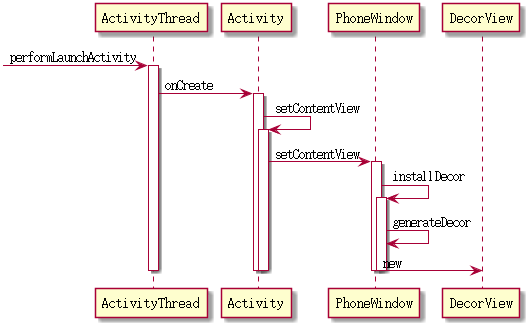
**2.1 PhonwWindow 以及WindowManagerImpl新建流程**

1 在ActivityThread的performLaunchActivity函数中，也就是启动activity的时候，先利用Instrumatation新建了一个activity对象，然后调用activity的attach函数为activity附加ContextImpl等属性。

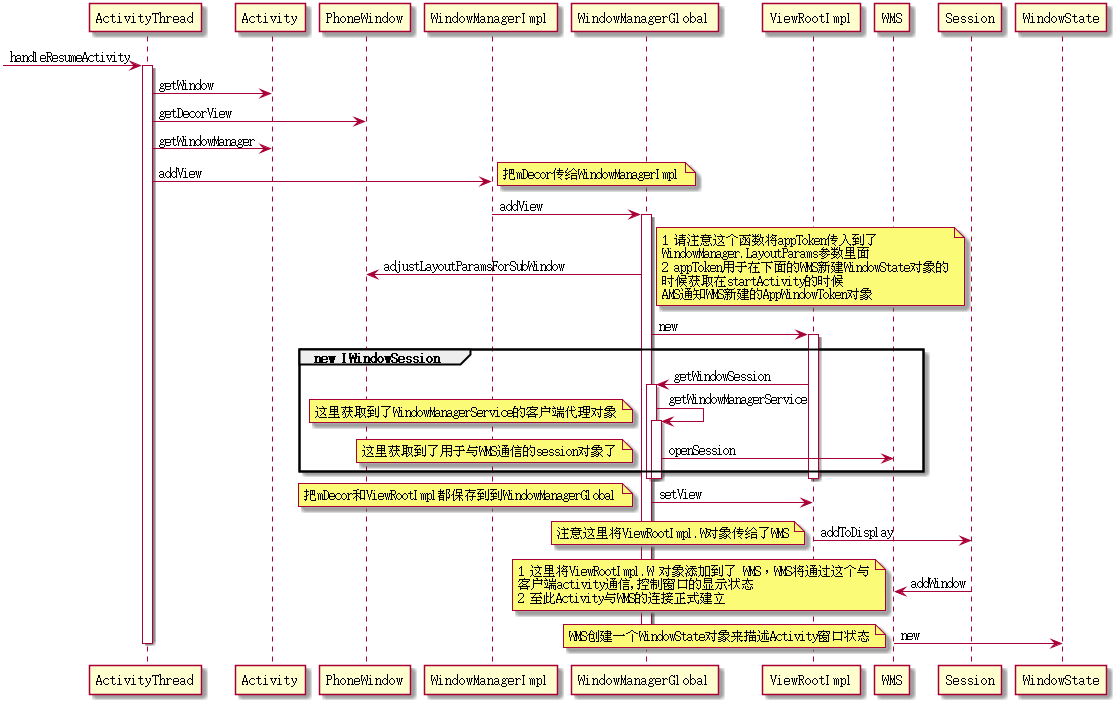
2 先调用PolicyManger生成了Window对象，这个PolicyManger直接新建了一个PhoneWindow对象返回

3 新建WindowManagerImpl的时候，先调用了getSystemService接口获得WindowManagerImpl对象，然后将获得的对象设置给了PhoneWindow对象，这个时候PhoneWindow又调用了传入的WindowManagerImpl对象的createLocalWindowManger新建了一个WindowManagerImpl对象。这两个WindowManagerImpl对象稍微有些区别，第一个对象是缓存在ContextImpl的SYSTEM\_SERVICE\_MAP静态列表里面的，因此整个app进程只有一个，这个对象包含了Display信息（从DisplayMangerService获得），而第二个对象则比第一个对象相比多了一个PhoneWindow成员变量。因此第二个对象才是真正用于维护窗口属性的，每个窗口都回对应新建一个这样的WindowManagerImpl对象。

**2.2新建DecorView对象**



**2.3 新建ViewRootImpl 连接wms 以及通知WMS新建WindowState**



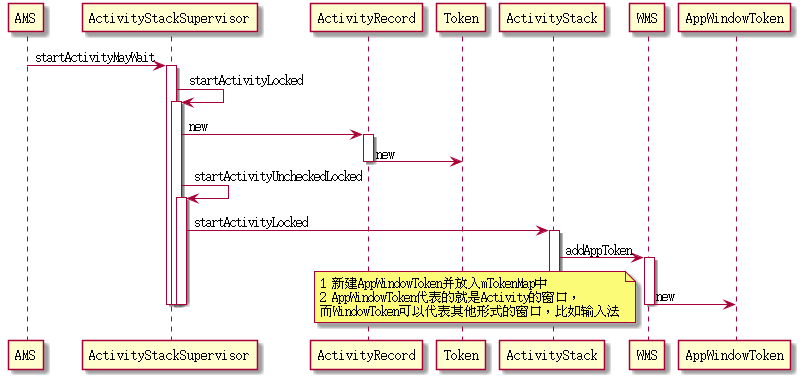
1 这里主要做了几件事

（1）新建ViewRootImpl对象并且添加到WindowMangerGlobal管理

（2）如果当前还没有连接到WMS则获取WMS服务并且获得Session对象，这个对象用于app与WMS通信，各种窗口操作请求都是通过这个session通知WMS

（3）调用session的addToDisplay通知WMS新建了一个窗口对象，WMS会新建一个WindowState对象来维护这个窗口的状态，需要注意 的是，在这一步app将ViewRootImpl.W对象传给了WMS,这个对象是WMS用于跟app通信用的，每当窗口状态变化，如大小变化，焦点变化的时候，WMS就会通过这个对象进行回调

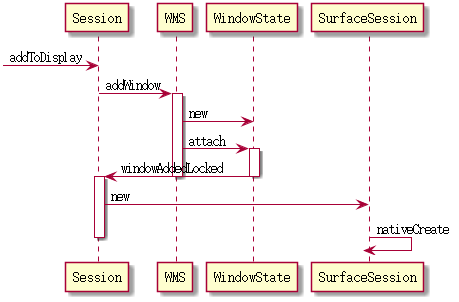
**2.4 Activity的appToken以及WMS的WindowToken新建**



也就是说，appToken的新建在AMS启动一个Activity的时候，为这个Activitity生成一个ActivityRecord对象的时候就会新建一个Token对象，首先吧这个对象传给WMS，WMS就会使用这个对象新建一个AppWindowToken对象并保存，然后AMS会在启动activity的最后一步将Token传递给APP，APP新建了Activity对象之后会传递给Activity。同时在ActvitityThread调用performResumeActivity的时候会调用对应WindowMangerImpl对象的addView将窗口添加到WindowMangerGlobal管理，WindowMangerGlobal则对应这个窗口生成一个ViewRoorImpl对象，然后通过ViewRoorImpl通知WMS新建了一个窗口，WMS会根据APP传过来的appToken找到之前AMS通知WMS新建的WindowToken对象，如果找不到的话窗口就不能添加到WMS管理啦！

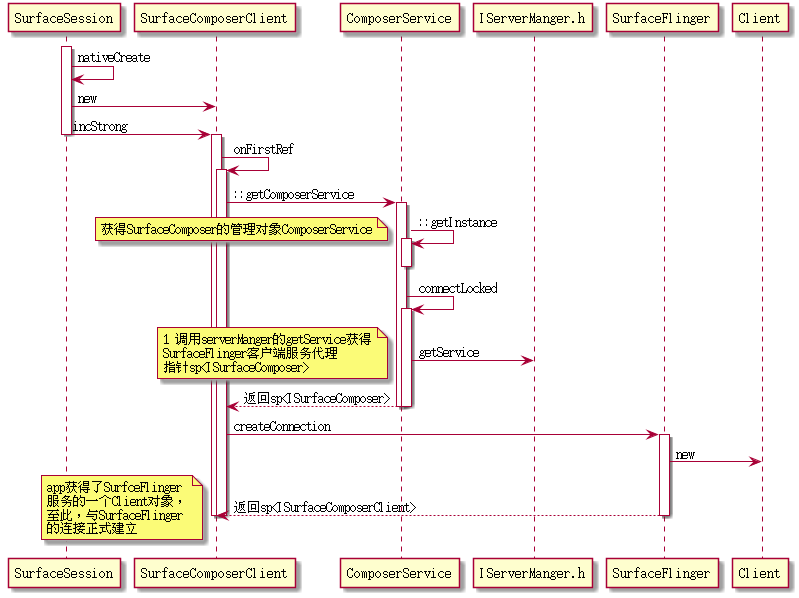
**3 WMS为每个app通知SurfaceFlinger新建Client对象的流程**

**3.1 新建SurfaceSession对象**



每个WindowState新建的时候会调用Session的windowAddedLocked方法，这个时候Session对象发现还没有连接SurfaceFlinger就会去新建一个SurfaceSession对象，新建的过程中就会区连接SurfaceFlinger，然后通知SurfaceFlinger新建一个Client对象维护app的绘制。主要的连接过程再native层

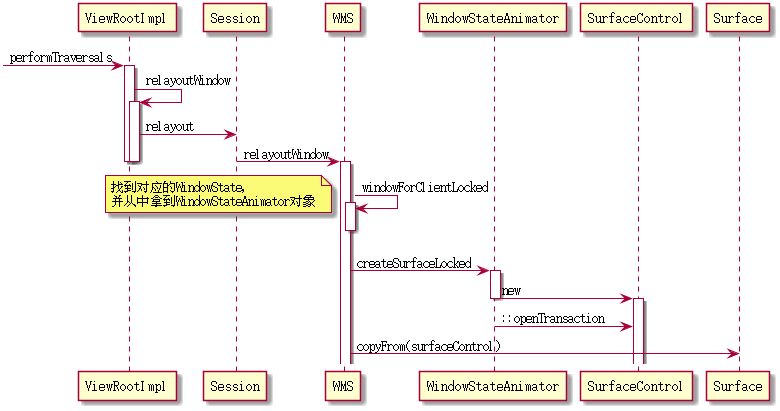
**3.2 native层的surfaceFlinger连接过程**



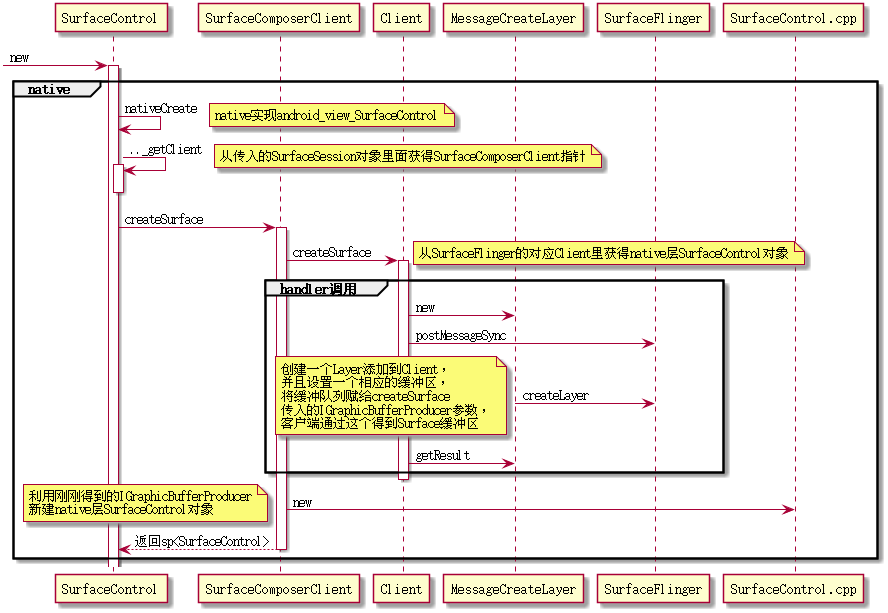
nativeCreate会new一个SurfaceComposerClient对象，然后主动进行引用数+1，这是智能指针sp的用法，可参考相关文档，每当要保存在某个地方的时候为了防止这个对象销毁应该引用数+1不用的时候-1就好了，然后会调用ComposerService的静态函数获得SurfaceFlinger的客户端代理指针sp<ISurfaceComposer>，然后调用SurfaceFlinger的createConnection函数连接SurfaceFlinger，SurfaceFlinger会新建一个Client对象返回，客户端就得到这个对象的代理指针sp<ISurfaceComposerClient>

其中SurfaceSession的底层实现在android\_view\_SurfaceSession.cpp

**3.3 app端如何获得SurfaceFlinger分配的绘图缓存**

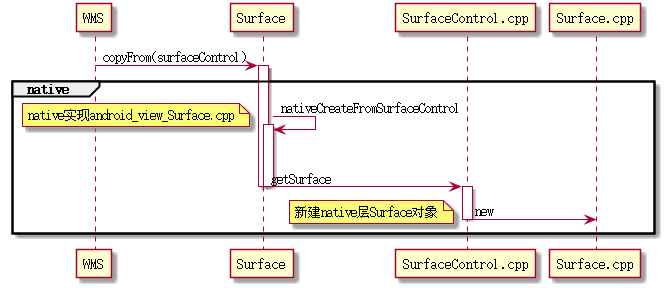
如图，当需要窗口进行绘制的时候，最终都会出发ViewRootImpl的performTraversals函数，这个函数会先向WMS调用relayoutWindow函数，WMS这时候就会先拿到窗口对应的WindowState对象，然后从中拿到其WindowStateAnimator对象，然后通过这个对象获得SurfaceControl对象，这个SurfaceControl对象就是用来为窗口分配Surface缓冲的，当Surface调用copyFrom函数的时候，就会使用这个对象来新建一个对应的native层的sp<Surface>对象。

SurfaceControl的native对象sp<SurfaceControl>新建过程如下



nativeCreate首先从传入的SurfaceSession参数（java对象）中得到SurfaceComposerClient对象指针，这个对象指针是在WMS与SurfaceFlinger连接那一步保存在SurfaceSession对象里面的，可详细查看上文。然后通过这个对象调用Client的createSurface函数请求SurfaceFlinger分配一个指定宽高，像素类型的缓冲区。SurfaceFlinger会将分配好的缓冲区防止到传入的IGraphicBufferProducer参数里面，然后再使用这个IGraphicBufferProducer对象分配一个SurfaceControl 的native对象

SurfaceControl对象新建好之后接下来就是通过Surface的copyFrom函数来新建一个Surface的native层对象了，过程如下：



调用copyFrom函数的时候会调用native函数nativeCreateFromSurfaceControl,这个函数会从传入的SurfaceControl对象（java对象）获得sp<SurfaceControl>native层对象,然后调用其getSurface函数新建一个sp<Surface>对象

注意，relayoutWindow这些操作都是再WMS进程完成的，那么传递到app端是怎么做到的呢，答案就是，relayoutWindow传入的outSurface参数是一个parcelable类型的参数，是一个out类型的参数哦，可以查看IWindowSession.aidl的定义

以下就是 IWindowSession.aidl转换成.java文件并分析这个转换过程的流程

IWindowSession.aidl

'/home/lss/Android/Sdk/build-tools/19.1.0/aidl' -I'/media/lss/else/androidSrc/mydroid/frameworks/base/core/java' -I'/media/lss/else/androidSrc/mydroid/frameworks/base/graphics/java' '/media/lss/else/androidSrc/mydroid/frameworks/base/core/java/android/view/IWindowSession.aidl' ./ttt.java

-I表示添加依赖路径，这个命令可以生成aidl文件对应的java文件

int relayout(IWindow window, int seq, in WindowManager.LayoutParams attrs,

int requestedWidth, int requestedHeight, int viewVisibility,

int flags, out Rect outFrame, out Rect outOverscanInsets,

out Rect outContentInsets, out Rect outVisibleInsets,

out Configuration outConfig, out Surface outSurface);

注意声明，outFrame，outOverscanInsets，outContentInsets，outVisibleInsets，outConfig，outSurface都是out参数

proxy的relayout函数实现（参数序列化）

@Override public int relayout(android.view.IWindow window, int seq, android.view.WindowManager.LayoutParams attrs, int requestedWidth, int requestedHeight, int viewVisibility, int flags, android.graphics.Rect outFrame, android.graphics.Rect outOverscanInsets, android.graphics.Rect outContentInsets, android.graphics.Rect outVisibleInsets, android.content.res.Configuration outConfig, android.view.Surface outSurface) throws android.os.RemoteException

{

android.os.Parcel \_data = android.os.Parcel.obtain();

android.os.Parcel \_reply = android.os.Parcel.obtain();

int \_result;

try {

\_data.writeInterfaceToken(DESCRIPTOR);

\_data.writeStrongBinder((((window!=null))?(window.asBinder()):(null)));

\_data.writeInt(seq);

…………….

省略

…………...

\_data.writeInt(flags);//序列化完毕

mRemote.transact(Stub.TRANSACTION\_relayout, \_data, \_reply, 0);//远程调用

\_reply.readException();

\_result = \_reply.readInt();

if ((0!=\_reply.readInt())) {

outFrame.readFromParcel(\_reply);//注意，这里在反序列化

}

…………….

省略

…………...

if ((0!=\_reply.readInt())) {

outSurface.readFromParcel(\_reply);//注意，这里得到了服务端的 Surface数据

}

}

finally {

\_reply.recycle();

\_data.recycle();

}

return \_result;

}

service的onTransact的relayout实现（参数序列化，已经实现relayout返回out参数）

case TRANSACTION\_relayout:

{

data.enforceInterface(DESCRIPTOR);

android.view.IWindow \_arg0;

\_arg0 = android.view.IWindow.Stub.asInterface(data.readStrongBinder());

int \_arg1;

\_arg1 = data.readInt();

…………….

省略

…………...

android.graphics.Rect \_arg7;//注意，这里开始new了一个用于out数据的对象

\_arg7 = new android.graphics.Rect();

…………….

省略

…………...

android.view.Surface \_arg12;

\_arg12 = new android.view.Surface();//注意，这里生成了一个surface对象

int \_result = this.relayout(\_arg0, \_arg1, \_arg2, \_arg3, \_arg4, \_arg5, \_arg6, \_arg7, \_arg8, \_arg9, \_arg10, \_arg11, \_arg12);

reply.writeNoException();

reply.writeInt(\_result);

if ((\_arg7!=null)) {

reply.writeInt(1);

\_arg7.writeToParcel(reply, android.os.Parcelable.PARCELABLE\_WRITE\_RETURN\_VALUE);

}

else {

reply.writeInt(0);

}

………..

if ((\_arg12!=null)) {//注意，这里吧surface数据都写回到了parcel包里面

reply.writeInt(1);

\_arg12.writeToParcel(reply, android.os.Parcelable.PARCELABLE\_WRITE\_RETURN\_VALUE);

}

else {

reply.writeInt(0);

}

return true;

}

static jint nativeReadFromParcel(JNIEnv\* env, jclass clazz,

jint nativeObject, jobject parcelObj) {

Parcel\* parcel = parcelForJavaObject(env, parcelObj);

if (parcel == NULL) {

doThrowNPE(env);

return 0;

}

sp<Surface> self(reinterpret\_cast<Surface \*>(nativeObject));

sp<IBinder> binder(parcel->readStrongBinder());

// update the Surface only if the underlying IGraphicBufferProducer

// has changed.

if (self != NULL

&& (self->getIGraphicBufferProducer()->asBinder() == binder)) {

// same IGraphicBufferProducer, return ourselves

return int(self.get());

}//从WMS转换到APP第一次的时候这里是NULL，不会进

sp<Surface> sur;

sp<IGraphicBufferProducer> gbp(interface\_cast<IGraphicBufferProducer>(binder));

if (gbp != NULL) {//获取 IGraphicBufferProducer对象指针

// we have a new IGraphicBufferProducer, create a new Surface for it

sur = new Surface(gbp, true);//新建Surface 的native对象

// and keep a reference before passing to java

sur->incStrong(&sRefBaseOwner);

}

if (self != NULL) {

// and loose the java reference to ourselves

self->decStrong(&sRefBaseOwner);

}

return int(sur.get());

}

static void nativeWriteToParcel(JNIEnv\* env, jclass clazz,

jint nativeObject, jobject parcelObj) {

Parcel\* parcel = parcelForJavaObject(env, parcelObj);

if (parcel == NULL) {

doThrowNPE(env);

return;

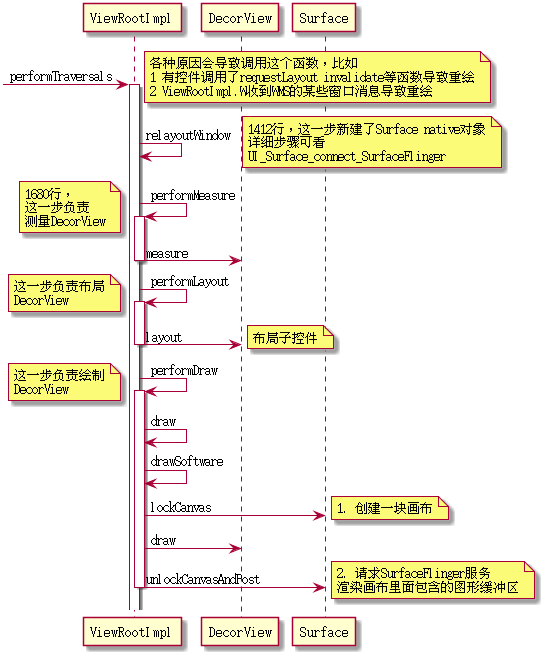
}

sp<Surface> self(reinterpret\_cast<Surface \*>(nativeObject));

parcel->writeStrongBinder( self != 0 ? self->getIGraphicBufferProducer()->asBinder() : NULL);

}

**3.4 Surface绘制过程**



1 当app请求，例如发送postInvalidate postLayout等命令的时候，还有app收到WMS的重绘信号的时候，最终都会调用performTraversals这个函数进行窗口的绘制。

2 绘制的流程总共分为4步

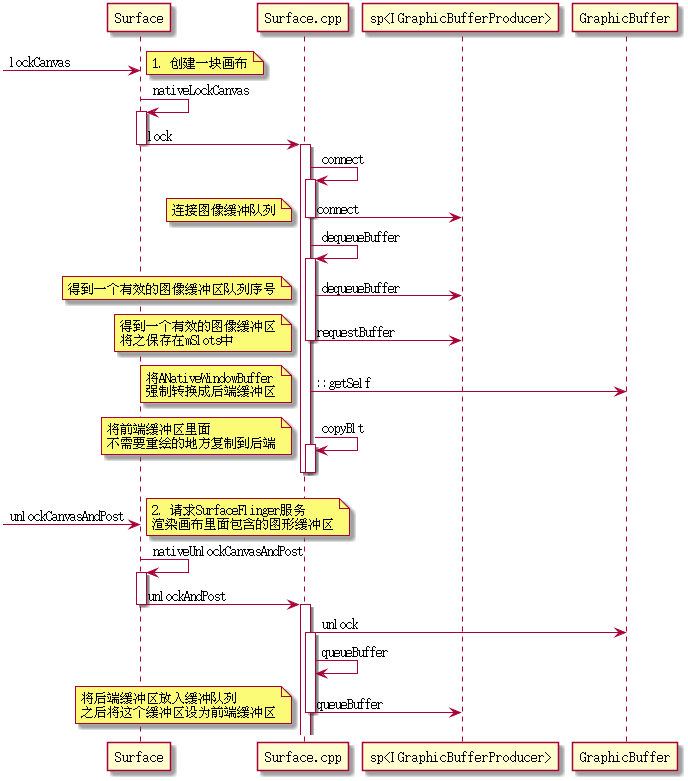
（1）relayoutWindow，这一步是WMS完成

（2）performMeasure，调用Decorview的measure函数进行控件的测量

（3）performLayout，调用Decorview的layout函数进行控件的布局

（4）performDraw，这一步完成具体的绘制，首先lockCanvas向SurfaceFlinger申请一块绘制缓冲区，然后调用Decorview的draw函数进行控件的绘制，吧控件绘制到了缓冲区上面，然后调用Surface的unlockCanvasAndPost函数通知SurfaeFlinger吧这个缓冲区显示出来。

**3.5 lockCanvas与unlockCanvas的底层操作**



1 lockCanvas就是申请绘图缓冲的过程，而unlockCanvas就是吧缓冲区放入SurfaceFlinger缓冲队列进行显示的过程

2 这里面需要注意的就是，dequeueBuffer函数从IGraphicBufferProducer里面获取是绘图缓冲队列的序号，requestBuffer函数是通过这个序号得到具体的缓冲区，不知道当时写这个代码的人是什么想法，从IGraphicBufferProducer里面获取到的缓冲区明明已经是GraphicBuffer了，偏偏要先转换成ANativeWindowBuffer指针类型，后面又调用GraphicBuffer::get()方法强制转换回GraphicBuffer类型指针，完全没有道理。

前面的Surface如何获取绘图缓冲区的过程我们可以知道，每个窗口在SurfaceFlinger端都会调用createLayer函数生成一个Layer对象(每个窗口都会新建一个SurfaceControl对象，这个对象创建的时候会通知SurfaceFlinger创建一个Layer)，这个Layer对象新建的时候就会新建一个BufferQueue也就是客户端的IGraphicBufferProducer，BufferQueue的实现类是SurfaceTextureLayer（路径/framework/native/services/SurfaceFlinger/SurfaceTextureLayer.h）这个类的主体实现都是BufferQueue这个类实现的（/framework/native/include/gui/BufferQueue.h）这里才是真正的IGraphicBufferProducer服务端绘图缓冲队列定义.