**3.Window之relayout**

1. **SurfaceControl的创建：**

在WMS中relayoutwindow中通过WindowStateAnimator来创建SurfaceControl对象。

在正常窗口创建的过程中使用的flag为0x00000404（OPAQUE&&HIDDEN）

通过WindowManager.LayoutParams中的format来决定是否为不透明窗口。

通过WindowManager.LayoutParams.FLAG\_SECURE或者窗口所有者userID存在于mScreenCaptureDisabled中设置flag为SECURE = 0x00000080。

通过是否支持硬件加速来设置format，TRANSLUCENT、TRANSPARENT、OPAQUE。

针对Dim surface 创建的时候设置flag为FX\_SURFACE\_DIM = 0x00020000。

static jlong nativeCreate(JNIEnv\* env, jclass clazz, jobject sessionObj,

jstring nameStr, jint w, jint h, jint format, jint flags) {

ScopedUtfChars name(env, nameStr);

sp<SurfaceComposerClient> client(android\_view\_SurfaceSession\_getClient(env, sessionObj));

sp<SurfaceControl> surface = client->createSurface(

String8(name.c\_str()), w, h, format, flags);

surface->incStrong((void \*)nativeCreate);

return reinterpret\_cast<jlong>(surface.get());

}

通过之前创建的SurfaceSession来获取SurfaceComposerClient，通过该对象的createSurface函数来创建SurfaceControl。SurfaceComposerClient一个进程只有一个，SurfaceControl每个窗口对应一个SurfaceControl。

sp<SurfaceControl> SurfaceComposerClient::createSurface(

const String8& name,

uint32\_t w,

uint32\_t h,

PixelFormat format,

uint32\_t flags)

{

sp<SurfaceControl> sur;

if (mStatus == NO\_ERROR) {

**sp<IBinder> handle;**

**sp<IGraphicBufferProducer> gbp;**

status\_t err = mClient->createSurface(name, w, h, format, flags,

&handle, &gbp);

ALOGE\_IF(err, "SurfaceComposerClient::createSurface error %s", strerror(-err));

if (err == NO\_ERROR) {

**sur = new SurfaceControl(this, handle, gbp);**

}

}

return sur;

}

status\_t Client::createSurface(

const String8& name,

uint32\_t w, uint32\_t h, PixelFormat format, uint32\_t flags,

sp<IBinder>\* handle,

sp<IGraphicBufferProducer>\* gbp)

{

class MessageCreateLayer : public MessageBase {

SurfaceFlinger\* flinger;

Client\* client;

sp<IBinder>\* handle;

sp<IGraphicBufferProducer>\* gbp;

status\_t result;

const String8& name;

uint32\_t w, h;

PixelFormat format;

uint32\_t flags;

public:

MessageCreateLayer(SurfaceFlinger\* flinger,

const String8& name, Client\* client,

uint32\_t w, uint32\_t h, PixelFormat format, uint32\_t flags,

sp<IBinder>\* handle,

sp<IGraphicBufferProducer>\* gbp)

: flinger(flinger), client(client),

handle(handle), gbp(gbp),

name(name), w(w), h(h), format(format), flags(flags) {

}

status\_t getResult() const { return result; }

virtual bool handler() {

result = flinger->createLayer(name, client, w, h, format, flags,

handle, gbp);

return true;

}

};

sp<MessageBase> msg = new MessageCreateLayer(mFlinger.get(),

name, this, w, h, format, flags, handle, gbp);

mFlinger->postMessageSync(msg);

return static\_cast<MessageCreateLayer\*>( msg.get() )->getResult();

}

创建一个MessageCreateLayer 消息同步发送到surfaceflinger的主线程中进行处理。

在surfaceflinger主线程中通过调用createLayer函数来创建相应的layer。

Layer的创建：

根据上层传过来的flag值来决定创建normal layer还是dim layer。

status\_t SurfaceFlinger::createLayer(

const String8& name,

const sp<Client>& client,

uint32\_t w, uint32\_t h, PixelFormat format, uint32\_t flags,

sp<IBinder>\* handle, sp<IGraphicBufferProducer>\* gbp)

{

status\_t result = NO\_ERROR;

sp<Layer> layer;

switch (flags & ISurfaceComposerClient::eFXSurfaceMask) {

case ISurfaceComposerClient::eFXSurfaceNormal:

result = createNormalLayer(client,

name, w, h, flags, format,

handle, gbp, &layer);

break;

case ISurfaceComposerClient::eFXSurfaceDim:

result = createDimLayer(client,

name, w, h, flags,

handle, gbp, &layer);

break;

default:

result = BAD\_VALUE;

break;

}

result = addClientLayer(client, \*handle, \*gbp, layer);

if (result != NO\_ERROR) {

return result;

}

setTransactionFlags(eTransactionNeeded);

return result;

}

@1：创建layer

@2：将创建的layer加入到client等一些容器中。

status\_t SurfaceFlinger::createNormalLayer(const sp<Client>& client,

const String8& name, uint32\_t w, uint32\_t h, uint32\_t flags, PixelFormat& format,

sp<IBinder>\* handle, sp<IGraphicBufferProducer>\* gbp, sp<Layer>\* outLayer)

{

\*outLayer = new Layer(this, client, name, w, h, flags);

status\_t err = (\*outLayer)->setBuffers(w, h, format, flags);

if (err == NO\_ERROR) {

**\*handle = (\*outLayer)->getHandle();**

**\*gbp = (\*outLayer)->getProducer();**

}

ALOGE\_IF(err, "createNormalLayer() failed (%s)", strerror(-err));

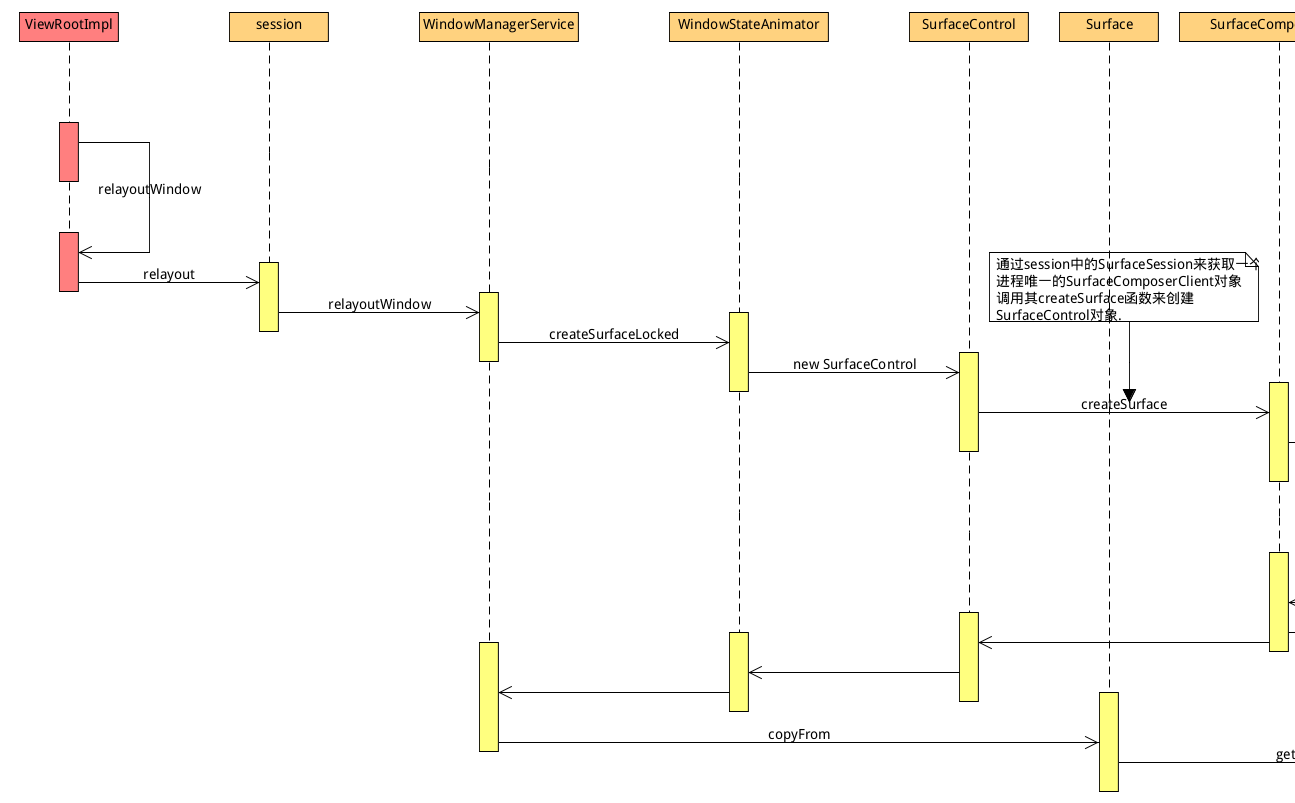
return err;

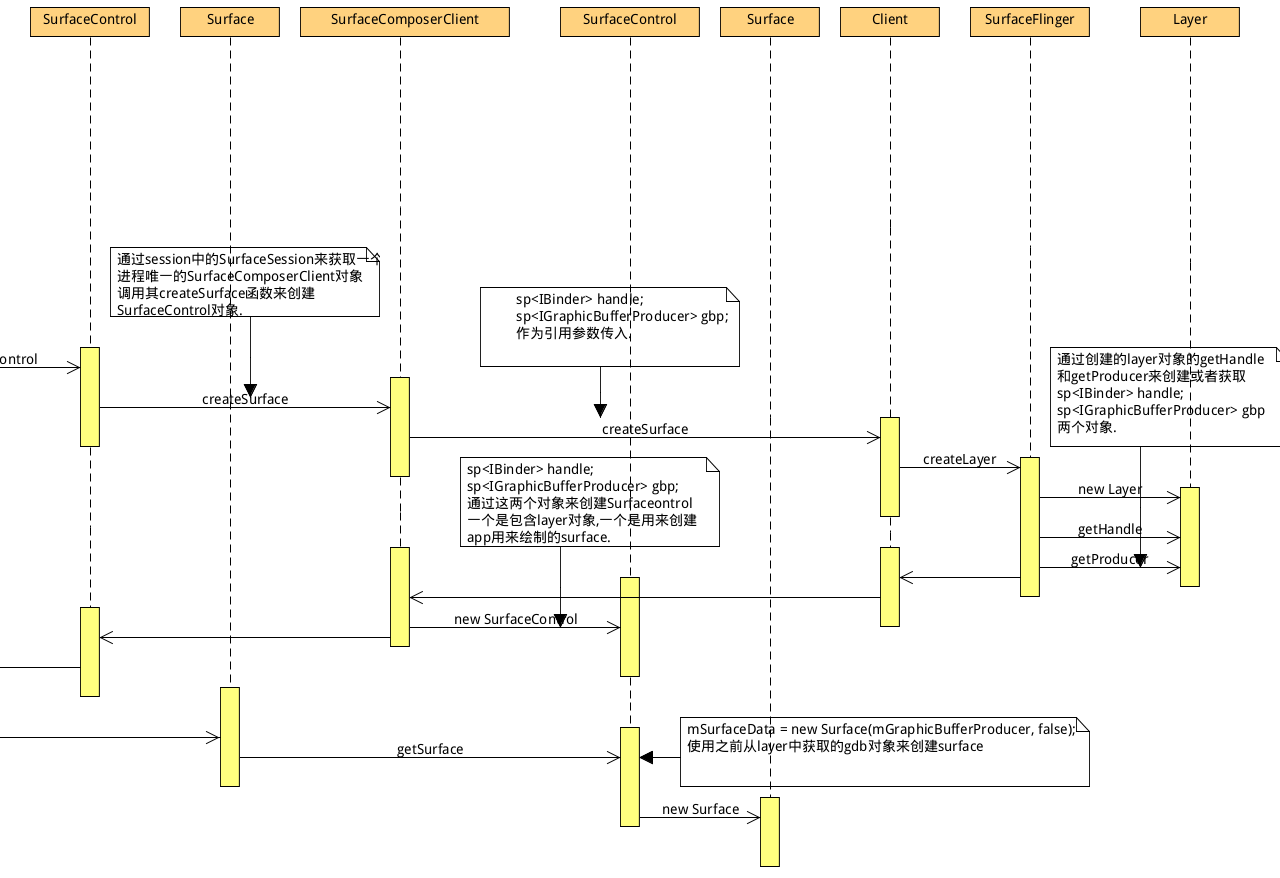
}

通过创建的layer对象获取handle和gbp对象，handle是一个包了surfaceflinger和layer对象的binder对象。gbp是一个IGraphicBufferProducer对象用于GraphicBuffer的生产者部分。

最终通过这两个对象来创建SurfaceControl对象。

handle是用来通过surfaceControl对象对layer的属性进行设置，IGraphicBufferProducer对象用来创建surface给app端进行绘制。





SurfaceControl 的作用：

1.通过其保存的gbp对象创建surface 供app进行绘制使用。

2.通过其对ComposerState来设置layer的state通过surfaceflinger改变layer的属性。

2.通过其对DisplayState来设置display的state通过surfaceflinger改变display的属性。

App端在什么时候进行relayout window操作：

"main@4083" prio=5 runnable

java.lang.Thread.State: RUNNABLE

at android.view.ViewRootImpl.scheduleTraversals(ViewRootImpl.java:1155)

at android.view.ViewRootImpl.requestLayout(ViewRootImpl.java:987)

at android.view.ViewRootImpl.setView(ViewRootImpl.java:554)

- locked <0x10c1> (a android.view.ViewRootImpl)

at android.view.WindowManagerGlobal.addView(WindowManagerGlobal.java:310)

at android.view.WindowManagerImpl.addView(WindowManagerImpl.java:85)

at android.app.ActivityThread.handleResumeActivity(ActivityThread.java:3419)

at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2574)

at android.app.ActivityThread.access$900(ActivityThread.java:150)

at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1399)

在add window之后，请求vsync信号进行relayout window的操作。

**2.LayoutWindow（计算窗口的位置）:**

performLayoutLockedInner:

performLayoutAndPlaceSurfacesLockedInner:

performLayoutAndPlaceSurfacesLockedLoop:

performLayoutAndPlaceSurfacesLocked:

relayoutWindow:

relayout:

@1:layer compute:

主要是通过PhoneWindowManager来计算windowstate的值，为后面WindowAnimator对WindowStateAnimator进行操作作准备。

beginLayoutLw:

**public void beginLayoutLw(boolean isDefaultDisplay, int displayWidth, int displayHeight, int displayRotation) {**

mDockLeft = mContentLeft = mVoiceContentLeft = mStableLeft = mStableFullscreenLeft

= mCurLeft = mUnrestrictedScreenLeft;

mDockTop = mContentTop = mVoiceContentTop = mStableTop = mStableFullscreenTop

= mCurTop = mUnrestrictedScreenTop;

mDockRight = mContentRight = mVoiceContentRight = mStableRight = mStableFullscreenRight

= mCurRight = displayWidth - overscanRight;

mDockBottom = mContentBottom = mVoiceContentBottom = mStableBottom = mStableFullscreenBottom

= mCurBottom = displayHeight - overscanBottom;

// start with the current dock rect, which will be (0,0,displayWidth,displayHeight)

**final Rect pf = mTmpParentFrame;**

**final Rect df = mTmpDisplayFrame;**

**final Rect of = mTmpOverscanFrame;**

**final Rect vf = mTmpVisibleFrame;**

**final Rect dcf = mTmpDecorFrame;**

**final Rect osf = mTmpOutsetFrame;**

**pf.left = df.left = of.left = vf.left = mDockLeft;**

**pf.top = df.top = of.top = vf.top = mDockTop;**

**pf.right = df.right = of.right = vf.right = mDockRight;**

**pf.bottom = df.bottom = of.bottom = vf.bottom = mDockBottom;**

dcf.setEmpty(); // Decor frame N/A for system bars.

通过display size和系统是否有status bar 和Navigation bar来计算pf、df、vf、dcf、osf等变量，为后面对其他窗口计算做准备。

Navigation bar：

1.当Navigation bar 位置在底部的时候，计算的Navigation bar top位置为displayHeight减去overscan的bottom减去Navigation bar的高度，为mTmpNavigationFrame、mStableBottom赋值为Navigation bar top，当navVisible 可见的时候为mDockBottom赋值为Navigation bar top。

mNavigationBarOnBottom = (!mNavigationBarCanMove || displayWidth < displayHeight);

if (mNavigationBarOnBottom) {

// It's a system nav bar or a portrait screen; nav bar goes on bottom.

int top = displayHeight - overscanBottom

- mNavigationBarHeightForRotation[displayRotation];

mTmpNavigationFrame.set(0, top, displayWidth, displayHeight - overscanBottom);

**mStableBottom = mStableFullscreenBottom = mTmpNavigationFrame.top;**

if (navVisible) {

mNavigationBarController.setBarShowingLw(true);

**mDockBottom = mTmpNavigationFrame.top;**

mRestrictedScreenHeight = mDockBottom - mRestrictedScreenTop;

mRestrictedOverscanScreenHeight = mDockBottom - mRestrictedOverscanScreenTop;

}

}

使用新计算的dock值为content区域更新计算结果：

// Make sure the content and current rectangles are updated to

// account for the restrictions from the navigation bar.

mContentTop = mVoiceContentTop = mCurTop = mDockTop;

mContentBottom = mVoiceContentBottom = mCurBottom = mDockBottom;

mContentLeft = mVoiceContentLeft = mCurLeft = mDockLeft;

mContentRight = mVoiceContentRight = mCurRight = mDockRight;

使用mTmpNavigationFrame为navigation bar 的windowstate 进行计算。

StatusBar：

使用Unrestricted和Stable为Status Bar的windowstate 进行计算，当status bar可见的时候使用mStatusBarHeight 高度为mStableTop赋值。

**mStableTop = mUnrestrictedScreenTop + mStatusBarHeight;**

if (mStatusBar.isVisibleLw() && !statusBarTransient) {

**mDockTop = mUnrestrictedScreenTop + mStatusBarHeight;**

mContentTop = mVoiceContentTop = mCurTop = mDockTop;

mContentBottom = mVoiceContentBottom = mCurBottom = mDockBottom;

mContentLeft = mVoiceContentLeft = mCurLeft = mDockLeft;

mContentRight = mVoiceContentRight = mCurRight = mDockRight;

}

layoutWindowLw:

if (!gone || !win.mHaveFrame || win.mLayoutNeeded

|| ((win.isConfigChanged() || win.setInsetsChanged()) &&

((win.mAttrs.privateFlags & PRIVATE\_FLAG\_KEYGUARD) != 0 ||

(win.mHasSurface && win.mAppToken != null &&

win.mAppToken.layoutConfigChanges)))) {

if (!win.mLayoutAttached) {

if (win.mAttrs.type == TYPE\_DREAM) {

// Don't layout windows behind a dream, so that if it

// does stuff like hide the status bar we won't get a

// bad transition when it goes away.

behindDream = true;

}

win.mLayoutNeeded = false;

win.prelayout();

mPolicy.layoutWindowLw(win, null);

win.mLayoutSeq = seq;

} else {

if (topAttached < 0) topAttached = i;

}

}

根据当前窗口是否可见来决定是否为当前窗口做layoutWindowLw。

final Rect pf = mTmpParentFrame;

final Rect df = mTmpDisplayFrame;

final Rect of = mTmpOverscanFrame;

final Rect cf = mTmpContentFrame;

final Rect vf = mTmpVisibleFrame;

final Rect dcf = mTmpDecorFrame;

final Rect sf = mTmpStableFrame;

// Default policy decor for the default display

dcf.left = mSystemLeft;

dcf.top = mSystemTop;

dcf.right = mSystemRight;

dcf.bottom = mSystemBottom;

win.computeFrameLw(pf, df, of, cf, vf, dcf, sf, osf);

关于layoutwindow过程中几个值得计算和使用：

1.mTmpParentFrame

在beginLayoutLw函数中默认情况下使用mDockLeft、mDockTop、mDockRight、mDockBottom为其赋值。

而dock区域由overscan区域和display宽高获得，就是整个屏幕的显示区域。

mUnrestrictedScreenLeft = overscanLeft;

mUnrestrictedScreenTop = overscanTop;

mDockLeft = mUnrestrictedScreenLeft;

mDockTop = mUnrestrictedScreenTop;

mDockRight = mCurRight = displayWidth - overscanRight;

mDockBottom = mCurBottom = displayHeight - overscanBottom;

Overscan是通过DisplayInfo中的overscan区域来赋值，overscan就是屏幕中四周黑色的区域，这4个变量的初始化在配置显示设备是完成的，默认为0。

finishLayoutLw:

static final int NO\_SURFACE = 0;

static final int DRAW\_PENDING = 1;

static final int COMMIT\_DRAW\_PENDING = 2;

static final int READY\_TO\_SHOW = 3;

static final int HAS\_DRAWN = 4;