Android Surfacer创建流程

1、概述

1.1、核心理解点：

- 应用程序与Surface的关系
- Surface与SurfaceFlinger的关系

1.2、源码依据

android10

两个Android源码地址：
platform_frameworks_native

platform_frameworks_base

大家自行下载两个库的Android源码，并切换到android10分支，跟着下面的流程一起看。

2、Activity的显示

2.1 Activity对象的创建

ActivityThread.handleResumeActivity：

```
public void handleResumeActivity(IBinder token, boolean finalStateRequest,
boolean isForward,
            String reason) {
        ...
        final ActivityClientRecord r = performResumeActivity(token,
finalStateRequest, reason);
        final Activity a = r.activity;
        ...
        if (r.window == null && !a.mFinished && willBeVisible) {
            //1、获取window对象
            r.window = r.activity.getWindow();
            //2、获取decorView
            View decor = r.window.getDecorView();
            ...
            //3、获取windowManger
            ViewManager wm = a.getWindowManager();
            WindowManager.LayoutParams l = r.window.getAttributes();
            a.mDecor = decor;
            ...
            if (a.mVisibleFromClient) {
                if (!a.mWindowAdded) {
                    a.mWindowAdded = true;
                    //4、将decor加到viewManager当中
                    wm.addView(decor, l);
```

```
                } else {
                    ...
                }
            }
            ...
        }
    }
```

以上有四步流程，在来了解四步流程以前，先看看Activity的setContentView或获取，上面四步中某些数据的来源。

```
@UnsupportedAppUsage
private Window mWindow;

public void setContentView(@LayoutRes int layoutResID) {
    getWindow().setContentView(layoutResID);
    initWindowDecorActionBar();
}

public Window getWindow() {
    return mWindow;
}
```

我们需要探究mWindow对象到底从何而来，Window的定义我们先看看官方的说明：

```
/**
 * Abstract base class for a top-level window look and behavior policy.  An
 * instance of this class should be used as the top-level view added to the
 * window manager. It provides standard UI policies such as a background, title
 * area, default key processing, etc.

 一个对于顶层窗口外观和行为策略的抽象基类。 该类的实例应作为加入到window manager中的顶层view来使用。他提供标准的ui策略，比如背景、标题栏、默认按键处理等。
 *
 * <p>The only existing implementation of this abstract class is
 * android.view.PhoneWindow, which you should instantiate when needing a
 * Window.
 该抽象类的唯一实现是android.view.PhoneWindow，当需要window时都使用的是PhoneWindow
 */
```

ok，其实认真读官方对于window的释义，还是挺清晰明了的，而且对于其用途也是清晰明了的。 通过这个介绍，我们知道在android中，window会承载着view，并添加到window Manger中。

那么接下来我们有两个小目标

- 了解window的实现类PhoneWindow，以及PhoneWindow的使用时机。
- 了解window manager是什么。

2.2、Window抽象类的唯一实现PhoneWindow

AcitivityThread.performLaunchActivity:

```
/**  Core implementation of activity launch. */
private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    ...
    //创建activity的context
    ContextImpl appContext = createBaseContextForActivity(r);
    Activity activity = null;
    try {
        java.lang.ClassLoader cl = appContext.getClassLoader();
        //创建activity对象
        activity = mInstrumentation.newActivity(
            cl, component.getClassName(), r.intent);
        ...
    } catch (Exception e) {
        ...
    }

    try {
        //创建application
        Application app = r.packageInfo.makeApplication(false, mInstrumentation);
        ...
        if (activity != null) {
            ...
            Window window = null;
            appContext.setOuterContext(activity);
            //关键函数 绑定context token等各种信息
            activity.attach(appContext, this, getInstrumentation(), r.token,
                            r.ident, app, r.intent, r.activityInfo, title, r.parent,
                            r.embeddedID, r.lastNonConfigurationInstances, config,
                            r.referrer, r.voiceInteractor, window, r.configCallback,
                            r.assistToken);
            ...
            r.activity = activity;
        }
        r.setState(ON_CREATE);
    } catch (SuperNotCalledException e) {
        throw e;
    } catch (Exception e) {
        ...
    }
    return activity;
}
```

Activity.attach:

```
@UnsupportedAppUsage
final void attach(Context context, ActivityThread aThread,
                  Instrumentation instr, IBinder token, int ident,
                  Application application, Intent intent, ActivityInfo info,
                  CharSequence title, Activity parent, String id,
                  NonConfigurationInstances lastNonConfigurationInstances,
                  Configuration config, String referrer, IVoiceInteractor
voiceInteractor,
                  Window window, ActivityConfigCallback activityConfigCallback,
IBinder assistToken) {
    attachBaseContext(context);
    //window实例化，类型为PhoneWindow
    mWindow = new PhoneWindow(this, window, activityConfigCallback);
    ...
    //给window设置windowManger
    mWindow.setWindowManager(
        (WindowManager)context.getSystemService(Context.WINDOW_SERVICE),
        mToken, mComponent.flattenToString(),
        (info.flags & ActivityInfo.FLAG_HARDWARE_ACCELERATED) != 0);
    if (mParent != null) {
        mWindow.setContainer(mParent.getWindow());
    }
    ...
}
```

到这里我们知道了window对象实例化的过程。 这里设置给window设置windowManger的过程也关注下：
mWindow.setWindowManager:

```
public void setWindowManager(WindowManager wm, IBinder appToken, String appName,
                             boolean hardwareAccelerated) {
    mAppToken = appToken;
    mAppName = appName;
    mHardwareAccelerated = hardwareAccelerated;
    if (wm == null) {
        wm = (WindowManager)mContext.getSystemService(Context.WINDOW_SERVICE);
    }
    mWindowManager = ((WindowManagerImpl)wm).createLocalWindowManager(this);
}

//WindowManagerImple.createLocalWindowManager
public WindowManagerImpl createLocalWindowManager(Window parentWindow) {
    return new WindowManagerImpl(mContext, parentWindow);
}

//WindowManagerImpl 构造类
private WindowManagerImpl(Context context, Window parentWindow) {
    mContext = context;
    mParentWindow = parentWindow;
}
```

```
22
```

这里我们可以看到mWindow中的windowManger实际为WindowManagerImpl对象。 这里需要先记一下，便于后面流程的解析。

2.4、HandleResumeActivity-将decorView添加到windowManger当中

从上面的解析中我们知道了Window的具体实现类，我们重回handleResumeActivity看一看：

```
@Override
public void handleResumeActivity(IBinder token, boolean finalStateRequest,
boolean isForward,
                                 String reason) {
    ...
    final Activity a = r.activity;
    if (r.window == null && !a.mFinished && willBeVisible) {
        //获取window，上面分析过，为phoneWindow对象
        r.window = r.activity.getWindow();
        View decor = r.window.getDecorView();
        decor.setVisibility(View.INVISIBLE);
        //获取viewManager，viewManger的接口在windowManger当中实现了
        ViewManager wm = a.getWindowManager();
        WindowManager.LayoutParams l = r.window.getAttributes();
        a.mDecor = decor;
        l.type = WindowManager.LayoutParams.TYPE_BASE_APPLICATION;
        l.softInputMode |= forwardBit;
        ...
        if (a.mVisibleFromClient) {
            if (!a.mWindowAdded) {
                a.mWindowAdded = true;
                //将decor添加到windowManager当中。
                wm.addView(decor, l);
            } else {
                ...
            }
        }
    } else if (!willBeVisible) {
        ...
    }
    ...
    Looper.myQueue().addIdleHandler(new Idler());
}

```

这里的的关键为wm.addView(decor, l); wm对象为activity当中的windowManger成员变量，该成员变量的具体类为WindowManagerImpl，这个在上一节分析过，我们继续追踪wm.addView(decor, l)，实际为WindowMangerImpl.addView

```
//WindowMangerImpl.addView
```

```
@Override
public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params)
{
    applyDefaultToken(params);
    mGlobal.addView(view, params, mContext.getDisplay(), mParentWindow);
}

//这里的mGlobal为WindowManagerGlobal对象，我们继续看WindowManagerGlobal.addView:
public void addView(View view, ViewGroup.LayoutParams params,
                    Display display, Window parentWindow) {
    ...
    ViewRootImpl root;
    View panelParentView = null;
    synchronized (mLock) {
        ...
            //1、创建ViewRootImpl对象
            root = new ViewRootImpl(view.getContext(), display);
        try {
            //2、调用setView 将view添加到ViewRootImpl当中
            root.setView(view, wparams, panelParentView);
        } catch (RuntimeException e) {
            ...
        }
    }
}
```

以上过程我们主要关注两个，一个是创建ViewRootImpl， 一个是ViewRootImpl.setView， 下一节我们先分析new ViewRootImpl(view.getContext(), display)， 里面大有乾坤。

2.5、newViewRootImpl

构造函数

```
@UnsupportedAppUsage
final IWindowSession mWindowSession;
final W mWindow;
public ViewRootImpl(Context context, Display display) {
    ...
    //mWindowSession创建
    mWindowSession = WindowManagerGlobal.getWindowSession();
    ...
    //mWindow对象创建
    mWindow = new W(this);
    ...
}
```

创建ViewRootImpl的构造函数设计很多成员变量的初始化，我们先只关注mWindowSession， 跟mWindow对象的创建，这两个对象创建与WMS有关。

- IWindowSession的创建

```
//WindowManagerGlobal.getWindowSession();
@UnsupportedAppUsage
public static IWindowSession getWindowSession() {
    synchronized (WindowManagerGlobal.class) {
        if (sWindowSession == null) {
            try {
                ...
                //IWindowManager为WMS的AIDL服务
                IWindowManager windowManager = getWindowManagerService();
                //调用WMS.opneSession获取一个IWindowSession
                sWindowSession = windowManager.openSession(
                    new IWindowSessionCallback.Stub() {
                        @Override
                        public void onAnimatorScaleChanged(float scale) {
                            ValueAnimator.setDurationScale(scale);
                        }
                    });
            } catch (RemoteException e) {
                throw e.rethrowFromSystemServer();
            }
        }
        return sWindowSession;
    }
}

//getWindowManagerService实现
@UnsupportedAppUsage
public static IWindowManager getWindowManagerService() {
    synchronized (WindowManagerGlobal.class) {
        if (sWindowManagerService == null) {
            sWindowManagerService = IWindowManager.Stub.asInterface(
                ServiceManager.getService("window"));
            ...
        }
        return sWindowManagerService;
    }
}
```

我们看看WMS提供的openSession方法

```
@Override
public IWindowSession openSession(IWindowSessionCallback callback) {
  return new Session(this, callback);
}

//那么Session到底是什么，我们看看官方释义：
/**
 * This class represents an active client session.  There is generally one
 * Session object per process that is interacting with the window manager.

该类代表了一个活跃的客户端会话，通常每个应用继承会持有一个Session来用于与WMS交互。
```

```
 */
class Session extends IWindowSession.Stub implements IBinder.DeathRecipient
{

}
```

ok，通过上面的官方注释，我们已经很清晰知道Session的作用了，可以简单理解为应用程序进 程与WMS交互的工具。

所以在创建ViewRootImpl时，会在WMS中申请一个Session，该Session可被 ViewRootImpl用于与WMS进行交互。

- mWindow对象的创建

  mWindow对象实际为W类型：

  W为ViewRootImpl的静态内部类：

```
static class W extends IWindow.Stub
```

这里我先给到结论，W是ViewRootImpl创建时传递给WMS的AIDL服务，WMS持有该对象对Window的行为进行控制。具体什么时候传递给WMS我们后续再ViewRootImpl.setView中分析

2.6、ViewRootImpl.setView

```
/**
     * We have one child
     */
public void setView(View view, WindowManager.LayoutParams attrs, View
panelParentView) {
    synchronized (this) {
        if (mView == null) {
            mView = view;
            ...
            //requestLayout
            requestLayout();
            ...
            try {
                ...
                //mWindowSession.addToDisplay
                res = mWindowSession.addToDisplay(mWindow, mSeq,
mWindowAttributes,
                                                  getHostVisibility(),
mDisplay.getDisplayId(), mTmpFrame,
                                                  mAttachInfo.mContentInsets,
mAttachInfo.mStableInsets,
                                                  mAttachInfo.mOutsets,
mAttachInfo.mDisplayCutout, mInputChannel,
                                                  mTempInsets);
                setFrame(mTmpFrame);
            } catch (RemoteException e) {
                ...
```

```
            } finally {
                ...
            }
    }
}

```

我们先关注mWindowSession.addToDisplay， 这里mWindowSession是ViewRootImpl在WMS中申请的Session, 可以调用WMS的服务，我们看看Session中addToDisplay的实现：

```
//Session.addToDisplay
@Override
public int addToDisplay(IWindow window, int seq, WindowManager.LayoutParams attrs,
                        int viewVisibility, int displayId, Rect outFrame, Rect outContentInsets,
                        Rect outStableInsets, Rect outOutsets,
                        DisplayCutout.ParcelableWrapper outDisplayCutout, InputChannel outInputChannel,
                        InsetsState outInsetsState) {
    return mService.addWindow(this, window, seq, attrs, viewVisibility, displayId, outFrame,
                              outContentInsets, outStableInsets, outOutsets, outDisplayCutout, outInputChannel,
                              outInsetsState);
}
```

//mService为WindowManagerService, WindowMangerService.addWindow里面的方法太繁琐了，我们就不看了，但可以关注到在addToDisplay时ViewRootImpl的mWindow对象，W类型，被传入到WMS当中，后续WMS可以通过该AIDL服务区操作客户端的window。

再看requstLayout，requestLayout会吧performTraversal任务发送给Choreographer，并请求在Vsync信号，在接收到Vsync信号时，performTraversal会被执行(这里面的过程这里就不分析了，外面很多文章有对应分析)。 我们关注下performTraversal里面的关键函数：

```
@UnsupportedAppUsage
public final Surface mSurface = new Surface();
private final SurfaceControl mSurfaceControl = new SurfaceControl();

private void performTraversals() {
    // cache mView since it is used so much below...
    final View host = mView;
    ...
    //这里有一个知识点，后续面试可以追踪下，八股之一
    // Execute enqueued actions on every traversal in case a detached view enqueued an action
    getRunQueue().executeActions(mAttachInfo.mHandler);
    if (mFirst || windowShouldResize || insetsChanged ||
        viewVisibilityChanged || params != null || mForceNextWindowRelayout) {
        ...
        try {
```

```
            ...
            //relayoutWindow 关键函数
            relayoutResult = relayoutWindow(params, viewVisibility,
insetsPending);
            ...
        } catch (RemoteException e) {
        }
        //ThreadedRenderer构建
        final ThreadedRenderer threadedRenderer = mAttachInfo.mThreadedRenderer;
        ...
    }
    ...
    mIsInTraversal = false;
}
```

performTraversals中有绘制三联操作，老生常谈了，这里我们只关注与WMS的有关的逻辑，

relayoutWindow：

```
    private int relayoutWindow(WindowManager.LayoutParams params, int
viewVisibility,
            boolean insetsPending) throws RemoteException {
        ...
        //注意这里传入了mSurfaceControl对象，mSurfaceControl是初始化时创建的
        int relayoutResult = mWindowSession.relayout(mWindow, mSeq, params,
                (int) (mView.getMeasuredWidth() * appScale + 0.5f),
                (int) (mView.getMeasuredHeight() * appScale + 0.5f),
viewVisibility,
                insetsPending ? WindowManagerGlobal.RELAYOUT_INSETS_PENDING : 0,
frameNumber,
                mTmpFrame, mPendingOverscanInsets, mPendingContentInsets,
mPendingVisibleInsets,
                mPendingStableInsets, mPendingOutsets, mPendingBackDropFrame,
mPendingDisplayCutout,
                mPendingMergedConfiguration, mSurfaceControl, mTempInsets);
        if (mSurfaceControl.isValid()) {
            //复制Surface
            mSurface.copyFrom(mSurfaceControl);
        } else {
            destroySurface();
        }
        ...
        return relayoutResult;
    }
```

这里调用了mWindowSession.relayout

```
    @Override
    public int relayout(IWindow window, int seq, WindowManager.LayoutParams
attrs,
```

```
            int requestedWidth, int requestedHeight, int viewFlags, int flags,
long frameNumber,
            Rect outFrame, Rect outOverscanInsets, Rect outContentInsets, Rect
outVisibleInsets,
            Rect outStableInsets, Rect outsets, Rect outBackdropFrame,
            DisplayCutout.ParcelableWrapper cutout, MergedConfiguration
mergedConfiguration,
            SurfaceControl outSurfaceControl, InsetsState outInsetsState) {
        if (false) Slog.d(TAG_WM, ">>>>>> ENTERED relayout from "
                + Binder.getCallingPid());
        Trace.traceBegin(TRACE_TAG_WINDOW_MANAGER, mRelayoutTag);
        //传入outSurfaceControl对象
        int res = mService.relayoutWindow(this, window, seq, attrs,
                requestedWidth, requestedHeight, viewFlags, flags, frameNumber,
                outFrame, outOverscanInsets, outContentInsets, outVisibleInsets,
                outStableInsets, outsets, outBackdropFrame, cutout,
                mergedConfiguration, outSurfaceControl, outInsetsState);
        Trace.traceEnd(TRACE_TAG_WINDOW_MANAGER);
        if (false) Slog.d(TAG_WM, "<<<<<< EXITING relayout to "
                + Binder.getCallingPid());
        return res;
    }
```

好了，前面铺垫这么久，终于到了与Surface相关的逻辑了，我们可以看到mSurface与mSurfaceControl都是在创建ViewRootImpl时创建的对象，但是最终传给了WMS, 后续使用WMS返回的mSurfaceControl复制到当前的Surface当中。所以实际上看来，window创建的Surface真正的操作是在WMS当中。

2.7、阶段总结

经历了上面一长串逻辑，我们先理清一下我们得到的结论。

- 在ActivityThread.handleResumeActivity当中，会将decorView加入到windowManager当中
- 该windowManager的实现为WindowMangerImpl，WindoMangerImpl.addView由WindowMangerGlobal代理实现
- WindowManagerGlobal.addView方法中会创建ViewRootImpl
- ViewRootImpl创建时会新建surface, 与surfaceControl，但实质为空实现。 并且会通过AIDL调用WMS请求一个Session, 持有该Session可以调用WMS的一些服务。
- ViewRootImpl.setView方法中会调用addToDisplay，将自身暴露给WMS的AIDL服务(W类型)传递给WMS。
- ViewRootImpl.setView方法中会调用requestLayout，最终执行的performTravelsal当中会调用mSession.relayoutWindow, 调用时会传入surfaceControl，最终服务结束时，将返回的surfaceControl复制到Surface当中。

接下来，我们就针对Surface怎么创建的主题进行进一步的分析。

3、Surface与SurfaceControl的创建。

3.1、Surface & SurfaceControl是什么

```
/**
 * Handle onto a raw buffer that is being managed by the screen compositor.
 *
 * <p>A Surface is generally created by or from a consumer of image buffers (such as a
 * {@link android.graphics.SurfaceTexture}, {@link android.media.MediaRecorder}, or
 * {@link android.renderscript.Allocation}), and is handed to some kind of producer (such as
 * {@link android.opengl.EGL14#eglCreateWindowSurface(android.opengl.EGLDisplay,android.opengl.EGLConfig,java.lang.Object,int[],int) OpenGL},
 * {@link android.media.MediaPlayer#setSurface MediaPlayer}, or
 * {@link android.hardware.camera2.CameraDevice#createCaptureSession CameraDevice}) to draw
 * into.</p>
 *
 * <p><strong>Note:</strong> A Surface acts like a
 * {@link java.lang.ref.WeakReference weak reference} to the consumer it is associated with. By
 * itself it will not keep its parent consumer from being reclaimed.</p>
 */
//Surface用于操纵被屏幕合成器(一般是SurfaceFlinger) 元缓冲数据
public class Surface implements Parcelable {

}
/**
 * Handle to an on-screen Surface managed by the system compositor. The SurfaceControl is
 * a combination of a buffer source, and metadata about how to display the buffers.
 * By constructing a {@link Surface} from this SurfaceControl you can submit buffers to be
 * composited. Using {@link SurfaceControl.Transaction} you can manipulate various
 * properties of how the buffer will be displayed on-screen. SurfaceControl's are
 * arranged into a scene-graph like hierarchy, and as such any SurfaceControl may have
 * a parent. Geometric properties like transform, crop, and Z-ordering will be inherited
 * from the parent, as if the child were content in the parents buffer stream.
 */
//处理被系统合成器管理的surface， SurfaceControl是缓冲数据源、和如何显示缓冲数据的的元数据组成。 通过构造SurfaceControl，你可以提交被合成的buffer。 使用Transaction事务你可以操纵有关buffer如何显示的各种属性。SurfaceControl 在图形场景中被以层级的关系组织，所以SurfaceControl是可能存在父类的，并且继承父类想Z-order、transform、crop等属性，就像子类是父类缓冲数据流中的内容。
public final class SurfaceControl implements Parcelable {
```

```
32
33  }
34
```

翻译的可能不是很恰当，理解大概的要点就行。

3.2、SurfaceControl创建 java流程

回到之前的

```
int res = mService.relayoutWindow(this, window, seq, attrs,
                requestedWidth, requestedHeight, viewFlags, flags, frameNumber,
                outFrame, outOverscanInsets, outContentInsets, outVisibleInsets,
                outStableInsets, outsets, outBackdropFrame, cutout,
                mergedConfiguration, outSurfaceControl, outInsetsState);
```

这里的mService对象为的实现类为WindowManagerService.java

找到对应实现类的relayoutWindow方法，寻找对outSurfaceControl操作的逻辑
我们找到这样的一个关键逻辑：

```
try {
    result = createSurfaceControl(outSurfaceControl, result, win, winAnimator);
} catch (Exception e) {
    displayContent.getInputMonitor().updateInputWindowsLw(true /*force*/);

    Slog.w(TAG_WM, "Exception thrown when creating surface for client "
             + client + " (" + win.mAttrs.getTitle() + ")",
             e);
    Binder.restoreCallingIdentity(origId);
    return 0;
}
```

查看createSurfaceControl实现：

```
private int createSurfaceControl(SurfaceControl outSurfaceControl, int result,
WindowState win,
        WindowStateAnimator winAnimator) {
    if (!win.mHasSurface) {
        result |= RELAYOUT_RES_SURFACE_CHANGED;
    }

    WindowSurfaceController surfaceController;
    try {
        Trace.traceBegin(TRACE_TAG_WINDOW_MANAGER, "createSurfaceControl");
        //创建surfaceController
        surfaceController = winAnimator.createSurfaceLocked(win.mAttrs.type,
win.mOwnerUid);
    } finally {
```

```
        Trace.traceEnd(TRACE_TAG_WINDOW_MANAGER);
    }
    if (surfaceController != null) {
        //将创建好的surfaceController给到outSurfaceControl
        surfaceController.getSurfaceControl(outSurfaceControl);
        if (SHOW_TRANSACTIONS) Slog.i(TAG_WM, "  OUT SURFACE " +
outSurfaceControl + ": copied");
    } else {
        // For some reason there isn't a surface.  Clear the
        // caller's object so they see the same state.
        Slog.w(TAG_WM, "Failed to create surface control for " + win);
        outSurfaceControl.release();
    }

    return result;
}
```

winAnimator为WindowStateAnimator类， 查看该类的实现

WindowSurfaceController createSurfaceLocked(int windowType, int ownerUid) {

```
WindowSurfaceController createSurfaceLocked(int windowType, int ownerUid) {
final WindowState w = mWin;

if (mSurfaceController != null) {
    //如果已经创建直接返回
    return mSurfaceController;
}
//未创建surface reset相关状态
w.setHasSurface(false);
resetDrawState();

mService.makeWindowFreezingScreenIfNeededLocked(w);

int flags = SurfaceControl.HIDDEN;
final WindowManager.LayoutParams attrs = w.mAttrs;

if (mService.isSecureLocked(w)) {
    flags |= SurfaceControl.SECURE;
}
//计算surface区间
calculateSurfaceBounds(w, attrs, mTmpSize);

// Set up surface control with initial size.
try {
    final boolean isHwAccelerated = (attrs.flags & FLAG_HARDWARE_ACCELERATED) !=
0;
    final int format = isHwAccelerated ? PixelFormat.TRANSLUCENT : attrs.format;
    if (!PixelFormat.formatHasAlpha(attrs.format)
            // Don't make surface with surfaceInsets opaque as they display a
            // translucent shadow.
            && attrs.surfaceInsets.left == 0
```

```
            && attrs.surfaceInsets.top == 0
            && attrs.surfaceInsets.right == 0
            && attrs.surfaceInsets.bottom == 0
            // Don't make surface opaque when resizing to reduce the amount of
            // artifacts shown in areas the app isn't drawing content to.
            && !w.isDragResizing()) {
        flags |= SurfaceControl.OPAQUE;
    }
    //创建surface关键逻辑
    mSurfaceController = new WindowSurfaceController(mSession.mSurfaceSession,
            attrs.getTitle().toString(), width, height, format, flags, this,
            windowType, ownerUid);
    mSurfaceController.setColorSpaceAgnostic((attrs.privateFlags
            & WindowManager.LayoutParams.PRIVATE_FLAG_COLOR_SPACE_AGNOSTIC) !=
0);

    setOffsetPositionForStackResize(false);
    mSurfaceFormat = format;

    w.setHasSurface(true);
} catch (OutOfResourcesException e) {
    Slog.w(TAG, "OutOfResourcesException creating surface");
    mService.mRoot.reclaimSomeSurfaceMemory(this, "create", true);
    mDrawState = NO_SURFACE;
    return null;
} catch (Exception e) {
    Slog.e(TAG, "Exception creating surface (parent dead?)", e);
    mDrawState = NO_SURFACE;
    return null;
}
return mSurfaceController;
}
```

查看该方法逻辑：
public WindowSurfaceController(SurfaceSession s, String name, int w, int h, int format,

```
public WindowSurfaceController(SurfaceSession s, String name, int w, int h, int
format,
        int flags, WindowStateAnimator animator, int windowType, int ownerUid) {
    mAnimator = animator;

    mSurfaceW = w;
    mSurfaceH = h;

    title = name;

    mService = animator.mService;
    final WindowState win = animator.mWin;
    mWindowType = windowType;
    mWindowSession = win.mSession;

```

```
    Trace.traceBegin(TRACE_TAG_WINDOW_MANAGER, "new SurfaceControl");
    //可以看到WindowSurfaceController核心就是创建一个SurfaceControl对象，
    final SurfaceControl.Builder b = win.makeSurface()
            .setParent(win.getSurfaceControl())
            .setName(name)
            .setBufferSize(w, h)
            .setFormat(format)
            .setFlags(flags)
            .setMetadata(METADATA_WINDOW_TYPE, windowType)
            .setMetadata(METADATA_OWNER_UID, ownerUid);
    mSurfaceControl = b.build();
    Trace.traceEnd(TRACE_TAG_WINDOW_MANAGER);
}
```

WindowSurfaceController会出一个SurfaceControl对象，并复制到他的mSurfaceControl成员变量中。

继续看看的创建：

public SurfaceControl build() {

```
@NonNull
public SurfaceControl build() {
    if (mWidth < 0 || mHeight < 0) {
        throw new IllegalStateException(
                "width and height must be positive or unset");
    }
    if ((mWidth > 0 || mHeight > 0) && (isColorLayerSet() ||
isContainerLayerSet())) {
        throw new IllegalStateException(
                "Only buffer layers can set a valid buffer size.");
    }
    //直接使用SurfaceControl的构造类使用
    return new SurfaceControl(
            mSession, mName, mWidth, mHeight, mFormat, mFlags, mParent,
mMetadata);
}
```

```
private SurfaceControl(SurfaceSession session, String name, int w, int h, int
format, int flags,
        SurfaceControl parent, SparseIntArray metadata)
                throws OutOfResourcesException, IllegalArgumentException {

    mName = name;
    mWidth = w;
    mHeight = h;
    Parcel metaParcel = Parcel.obtain();
    try {
        if (metadata != null && metadata.size() > 0) {
            metaParcel.writeInt(metadata.size());
            for (int i = 0; i < metadata.size(); ++i) {
                metaParcel.writeInt(metadata.keyAt(i));
```

```
                metaParcel.writeByteArray(
                        ByteBuffer.allocate(4).order(ByteOrder.nativeOrder())
                                .putInt(metadata.valueAt(i)).array());
            }
            metaParcel.setDataPosition(0);
        }
        //nativeCreate方法构建对应的native对象，这里可以看到surface对象的实际创建都是在native进行创建的。
        mNativeObject = nativeCreate(session, name, w, h, format, flags,
                parent != null ? parent.mNativeObject : 0, metaParcel);
    } finally {
        metaParcel.recycle();
    }
    if (mNativeObject == 0) {
        throw new OutOfResourcesException(
                "Couldn't allocate SurfaceControl native object");
    }

    mCloseGuard.open("release");
}
```

3.3、SurfaceControl创建native层流程

通过上面java层的分析知道，surface的实际创建是在native层完成的

private static native long nativeCreate(SurfaceSession session, String name,...) 该方法的实现在E:\AOSP\frameworks\base\core\jni\android_view_SurfaceControl.cpp当中

```
static jlong nativeCreate(JNIEnv* env, jclass clazz, jobject sessionObj,
        jstring nameStr, jint w, jint h, jint format, jint flags, jlong parentObject,
        jobject metadataParcel) {
    ScopedUtfChars name(env, nameStr);
    sp<SurfaceComposerClient> client;
    //创建SurfaceComposerClient对象，SurfaceComposerClient对象是创建surface的核心，其内部会通过binder调用surfaceFlinger的服务创建surface。
    if (sessionObj != NULL) {
        client = android_view_SurfaceSession_getClient(env, sessionObj);
    } else {
        client = SurfaceComposerClient::getDefault();
    }
    SurfaceControl *parent = reinterpret_cast<SurfaceControl*>(parentObject);
    sp<SurfaceControl> surface;
    LayerMetadata metadata;
    Parcel* parcel = parcelForJavaObject(env, metadataParcel);
    if (parcel && !parcel->objectsCount()) {
        status_t err = metadata.readFromParcel(parcel);
        if (err != NO_ERROR) {
          jniThrowException(env, "java/lang/IllegalArgumentException",
                            "Metadata parcel has wrong format");
```

```
        }
    }
    //使用SurfaceComposerClient创建surface
    status_t err = client->createSurfaceChecked(
            String8(name.c_str()), w, h, format, &surface, flags, parent,
std::move(metadata));
    if (err == NAME_NOT_FOUND) {
        jniThrowException(env, "java/lang/IllegalArgumentException", NULL);
        return 0;
    } else if (err != NO_ERROR) {
        jniThrowException(env, OutOfResourcesException, NULL);
        return 0;
    }

    surface->incStrong((void *)nativeCreate);
    return reinterpret_cast<jlong>(surface.get());
}
```

surface是由client->createSurfaceChecked创建的，这里的client对象为SurfaceComposerClient。

我们切换到E:\AOSP\frameworks\native\libs\gui\SurfaceComposerClient.cpp中去查看createSurfaceChecked方法的实现

```
status_t SurfaceComposerClient::createSurfaceChecked(const String8& name,
uint32_t w, uint32_t h,
                                                      PixelFormat format,
                                                      sp<SurfaceControl>*
outSurface, uint32_t flags,
                                                      SurfaceControl* parent,
                                                      LayerMetadata metadata) {
    sp<SurfaceControl> sur;
    status_t err = mStatus;

    if (mStatus == NO_ERROR) {
        sp<IBinder> handle;
        sp<IBinder> parentHandle;
        sp<IGraphicBufferProducer> gbp;

        if (parent != nullptr) {
            parentHandle = parent->getHandle();
        }
        //SurfaceComposerClient中调用mClient->createSurface创建surface
        err = mClient->createSurface(name, w, h, format, flags, parentHandle,
std::move(metadata),
                                     &handle, &gbp);
        ALOGE_IF(err, "SurfaceComposerClient::createSurface error %s",
strerror(-err));
        if (err == NO_ERROR) {
            *outSurface = new SurfaceControl(this, handle, gbp, true /* owned
*/);
        }
    }
```

```
    return err;
}
```

mClient->createSurface中， mClient对象为E:\AOSP\frameworks\native\services\surfaceflinger\Client.cpp对象，切换到对应的类当中查看其实现：

```
status_t Client::createSurface(const String8& name, uint32_t w, uint32_t h, PixelFormat format,
                               uint32_t flags, const sp<IBinder>& parentHandle,
                               LayerMetadata metadata, sp<IBinder>* handle,
                               sp<IGraphicBufferProducer>* gbp) {
    // We rely on createLayer to check permissions.
    //使用mFlinger.createLayer方法创建surface， mFlinger为SurfaceFlinger的智能指针
    return mFlinger->createLayer(name, this, w, h, format, flags, std::move(metadata), handle, gbp,
                                 parentHandle);
}
```

切换到E:\AOSP\frameworks\native\services\surfaceflinger\SurfaceFlinger.cpp当中

```
status_t SurfaceFlinger::createLayer(const String8& name, const sp<Client>& client, uint32_t w,
                                     uint32_t h, PixelFormat format, uint32_t flags,
                                     LayerMetadata metadata, sp<IBinder>* handle,
                                     sp<IGraphicBufferProducer>* gbp,
                                     const sp<IBinder>& parentHandle,
                                     const sp<Layer>& parentLayer) {
    //...省略非关键代码

    sp<Layer> layer;

    String8 uniqueName = getUniqueLayerName(name);

    switch (flags & ISurfaceComposerClient::eFXSurfaceMask) {
        case ISurfaceComposerClient::eFXSurfaceBufferQueue:
            result = createBufferQueueLayer(client, uniqueName, w, h, flags, std::move(metadata),
                                            format, handle, gbp, &layer);

            break;
        case ISurfaceComposerClient::eFXSurfaceBufferState:
            result = createBufferStateLayer(client, uniqueName, w, h, flags, std::move(metadata),
                                            handle, &layer);
            break;
        case ISurfaceComposerClient::eFXSurfaceColor:
            // check if buffer size is set for color layer.
            if (w > 0 || h > 0) {
```

```
                ALOGE("createLayer() failed, w or h cannot be set for color layer (w=%d, h=%d)",
                        int(w), int(h));
                return BAD_VALUE;
            }

            result = createColorLayer(client, uniqueName, w, h, flags, std::move(metadata), handle,
                                      &layer);
            break;
        case ISurfaceComposerClient::eFXSurfaceContainer:
            // check if buffer size is set for container layer.
            if (w > 0 || h > 0) {
                ALOGE("createLayer() failed, w or h cannot be set for container layer (w=%d, h=%d)",
                        int(w), int(h));
                return BAD_VALUE;
            }
            result = createContainerLayer(client, uniqueName, w, h, flags, std::move(metadata),
                                          handle, &layer);
            break;
        default:
            result = BAD_VALUE;
            break;
    }

    if (result != NO_ERROR) {
        return result;
    }

    if (primaryDisplayOnly) {
        layer->setPrimaryDisplayOnly();
    }

    bool addToCurrentState = callingThreadHasUnscopedSurfaceFlingerAccess();
    result = addClientLayer(client, *handle, *gbp, layer, parentHandle, parentLayer,
                            addToCurrentState);
    if (result != NO_ERROR) {
        return result;
    }
    mInterceptor->saveSurfaceCreation(layer);

    setTransactionFlags(eTransactionNeeded);
    return result;
}
```

switch (flags & ISurfaceComposerClient::eFXSurfaceMask)里面会有四种case

- ISurfaceComposerClient::eFXSurfaceBufferQueue
  当 `flags` 中包含 `ISurfaceComposerClient::eFXSurfaceBufferQueue` 标志时，会创建一个 `BufferQueue` 对象。BufferQueue 是 SurfaceFlinger 用来进行缓冲区管理的类，它可以保证 SurfaceFlinger 能够正确地显示和更新屏幕内容。这种情况通常用于显示视频或动画等内容。
- ISurfaceComposerClient::eFXSurfaceBufferState
  当 `flags` 中包含 `ISurfaceComposerClient::eFXSurfaceBufferState` 标志时，会创建一个 Surface 作为存储缓冲区的目标，这种情况通常用于存储帧缓冲区，例如屏幕截图或录屏。
- ISurfaceComposerClient::eFXSurfaceColor
  当 `flags` 中包含 `ISurfaceComposerClient::eFXSurfaceColor` 标志时，会创建一个 Surface 作为纯色的显示目标。这种情况通常用于创建底色或者背景。
- ISurfaceComposerClient::eFXSurfaceContainer
  当 `flags` 中包含 `ISurfaceComposerClient::eFXSurfaceContainer` 标志时，会创建一个 Surface，作为一个容器来包含其他 Surface，这种情况通常用于创建一个 Surface 的集合。例如，Android 中的活动（Activity）就是一个 Surface，它可以包含其他 Surface，例如窗口和视图等。

可以记住一个简单的场景，我们平时使用的activity是ISurfaceComposerClient::eFXSurfaceContainer类型，使用surfaceView的话对应ISurfaceComposerClient::eFXSurfaceBufferQueue类型。

我们关注下第一种case:ISurfaceComposerClient::eFXSurfaceBufferQueue

```
status_t SurfaceFlinger::createBufferQueueLayer(const sp<Client>& client, const String8& name,
                                                uint32_t w, uint32_t h, uint32_t flags,
                                                LayerMetadata metadata, PixelFormat& format,
                                                sp<IBinder>* handle,
                                                sp<IGraphicBufferProducer>* gbp,
                                                sp<Layer>* outLayer) {
    // initialize the surfaces
    switch (format) {
    case PIXEL_FORMAT_TRANSPARENT:
    case PIXEL_FORMAT_TRANSLUCENT:
        format = PIXEL_FORMAT_RGBA_8888;
        break;
    case PIXEL_FORMAT_OPAQUE:
        format = PIXEL_FORMAT_RGBX_8888;
        break;
    }

    sp<BufferQueueLayer> layer = getFactory().createBufferQueueLayer(
            LayerCreationArgs(this, client, name, w, h, flags, std::move(metadata)));
    status_t err = layer->setDefaultBufferProperties(w, h, format);
    if (err == NO_ERROR) {
        *handle = layer->getHandle();
        //graphicBufferProducer从创建的layer中获取
        *gbp = layer->getProducer();
        //将创建的layer传递给传入的layer参数指针。
```

```
        *outLayer = layer;
    }

    ALOGE_IF(err, "createBufferQueueLayer() failed (%s)", strerror(-err));
    return err;
}
```

切换到：E:\AOSP\frameworks\native\services\surfaceflinger\SurfaceFlingerFactory.cpp 中查看

```
sp<BufferQueueLayer> createBufferQueueLayer(const LayerCreationArgs& args)
override {
    return new BufferQueueLayer(args);
}
```

E:\AOSP\frameworks\native\services\surfaceflinger\BufferQueueLayer.cpp

```
BufferQueueLayer::BufferQueueLayer(const LayerCreationArgs& args) :
BufferLayer(args) {}
```

里面的创建逻辑在父类BufferLayer当中：

E:\AOSP\frameworks\native\services\surfaceflinger\BufferLayer.cpp

```
BufferLayer::BufferLayer(const LayerCreationArgs& args)
      : Layer(args),
        mTextureName(args.flinger->getNewTexture()),
        mCompositionLayer{mFlinger->getCompositionEngine().createLayer(
                compositionengine::LayerCreationArgs{this})} {
    ALOGV("Creating Layer %s", args.name.string());

    mPremultipliedAlpha = !(args.flags &
ISurfaceComposerClient::eNonPremultiplied);

    mPotentialCursor = args.flags & ISurfaceComposerClient::eCursorWindow;
    mProtectedByApp = args.flags & ISurfaceComposerClient::eProtectedByApp;
}

```

继续看起父类Layer的实现
E:\AOSP\frameworks\native\services\surfaceflinger\Layer.cpp

```
Layer::Layer(const LayerCreationArgs& args)
      : mFlinger(args.flinger),
        mName(args.name),
        mClientRef(args.client),
        mWindowType(args.metadata.getInt32(METADATA_WINDOW_TYPE, 0)) {
    mCurrentCrop.makeInvalid();

    uint32_t layerFlags = 0;
```

```
    if (args.flags & ISurfaceComposerClient::eHidden) layerFlags |=
layer_state_t::eLayerHidden;
    if (args.flags & ISurfaceComposerClient::eOpaque) layerFlags |=
layer_state_t::eLayerOpaque;
    if (args.flags & ISurfaceComposerClient::eSecure) layerFlags |=
layer_state_t::eLayerSecure;

    mTransactionName = String8("TX - ") + mName;
    //各种状态赋值
    mCurrentState.active_legacy.w = args.w;
    mCurrentState.active_legacy.h = args.h;
    mCurrentState.flags = layerFlags;
    mCurrentState.active_legacy.transform.set(0, 0);
    mCurrentState.crop_legacy.makeInvalid();
    mCurrentState.requestedCrop_legacy = mCurrentState.crop_legacy;
    mCurrentState.z = 0;
    mCurrentState.color.a = 1.0f;
    mCurrentState.layerStack = 0;
    mCurrentState.sequence = 0;
    mCurrentState.requested_legacy = mCurrentState.active_legacy;
    mCurrentState.active.w = UINT32_MAX;
    mCurrentState.active.h = UINT32_MAX;
    mCurrentState.active.transform.set(0, 0);
    mCurrentState.transform = 0;
    mCurrentState.transformToDisplayInverse = false;
    mCurrentState.crop.makeInvalid();
    mCurrentState.acquireFence = new Fence(-1);
    mCurrentState.dataspace = ui::Dataspace::UNKNOWN;
    mCurrentState.hdrMetadata.validTypes = 0;
    mCurrentState.surfaceDamageRegion.clear();
    mCurrentState.cornerRadius = 0.0f;
    mCurrentState.api = -1;
    mCurrentState.hasColorTransform = false;
    mCurrentState.colorSpaceAgnostic = false;
    mCurrentState.metadata = args.metadata;

    // drawing state & current state are identical
    mDrawingState = mCurrentState;

    CompositorTiming compositorTiming;
    args.flinger->getCompositorTiming(&compositorTiming);
    mFrameEventHistory.initializeCompositorTiming(compositorTiming);
    mFrameTracker.setDisplayRefreshPeriod(compositorTiming.interval);
    //注册Layer
    mSchedulerLayerHandle = mFlinger->mScheduler->registerLayer(mName.c_str(),
mWindowType);
    //回调LayerCreated
    mFlinger->onLayerCreated();
}
```

这里细看了下没看到surface创建的相关逻辑，我们回过头看
E:\AOSP\frameworks\native\services\surfaceflinger\BufferLayer.cpp

```
BufferLayer::BufferLayer(const LayerCreationArgs& args)
      : Layer(args),
        mTextureName(args.flinger->getNewTexture()),
       //这里调用mFlinger->getCompositionEngine().createLayer
        mCompositionLayer{mFlinger->getCompositionEngine().createLayer(
                compositionengine::LayerCreationArgs{this})} {
    ALOGV("Creating Layer %s", args.name.string());

    mPremultipliedAlpha = !(args.flags &
ISurfaceComposerClient::eNonPremultiplied);

    mPotentialCursor = args.flags & ISurfaceComposerClient::eCursorWindow;
    mProtectedByApp = args.flags & ISurfaceComposerClient::eProtectedByApp;
}

```

mFlinger->getCompositionEngine()方法

E:\AOSP\frameworks\native\services\surfaceflinger\CompositionEngine\src\Output.cpp

```
const CompositionEngine& Output::getCompositionEngine() const {
    return mCompositionEngine;
}
```

E:\AOSP\frameworks\native\services\surfaceflinger\CompositionEngine\src\CompositionEngine.cpp

```
std::shared_ptr<compositionengine::Layer>
CompositionEngine::createLayer(LayerCreationArgs&& args) {
    return compositionengine::impl::createLayer(*this, std::move(args));
}
```

E:\AOSP\frameworks\native\services\surfaceflinger\CompositionEngine\src\Layer.cpp

```
std::shared_ptr<compositionengine::Layer> createLayer(
        const compositionengine::CompositionEngine& compositionEngine,
        compositionengine::LayerCreationArgs&& args) {
    return std::make_shared<Layer>(compositionEngine, std::move(args));
}
```

该方法使用C++11的变长模板参数和完美转发来创建一个任意类型的Layer对象。这里我们传递的是`BufferQueueLayer`类型。`createLayer`方法会调用`std::make_shared`来创建一个`shared_ptr`类型的Layer对象。

到这里看都没有看到具体的Surface实例的创建，我们忘记了一个很重要的一点`onFirstRef()`是在`sp<T>`或者`wp<T>`第一次被引用的时候被调用的。我们回看到BufferQueueLayer中，里面有对
onFirstRef的实现：

E:\AOSP\frameworks\native\services\surfaceflinger\BufferQueueLayer.cpp

```
void BufferQueueLayer::onFirstRef() {
    BufferLayer::onFirstRef();

    // Creates a custom BufferQueue for SurfaceFlingerConsumer to use
    sp<IGraphicBufferProducer> producer;
    sp<IGraphicBufferConsumer> consumer;
    //创建bufferQueue,传入producer与consumer的引用
    BufferQueue::createBufferQueue(&producer, &consumer, true);
    mProducer = new MonitoredProducer(producer, mFlinger, this);
    {
        // Grab the SF state lock during this since it's the only safe way to access RenderEngine
        Mutex::Autolock lock(mFlinger->mStateLock);
        mConsumer =
                new BufferLayerConsumer(consumer, mFlinger->getRenderEngine(), mTextureName, this);
    }
    mConsumer->setConsumerUsageBits(getEffectiveUsage(0));
    mConsumer->setContentsChangedListener(this);
    mConsumer->setName(mName);

    // BufferQueueCore::mMaxDequeuedBufferCount is default to 1
    if (!mFlinger->isLayerTripleBufferingDisabled()) {
        mProducer->setMaxDequeuedBufferCount(2);
    }

    if (const auto display = mFlinger->getDefaultDisplayDevice()) {
        updateTransformHint(display);
    }
}
```

E:\AOSP\frameworks\native\libs\gui\BufferQueue.cpp

```
void BufferQueue::createBufferQueue(sp<IGraphicBufferProducer>* outProducer,
        sp<IGraphicBufferConsumer>* outConsumer,
        bool consumerIsSurfaceFlinger) {
    LOG_ALWAYS_FATAL_IF(outProducer == nullptr,
            "BufferQueue: outProducer must not be NULL");
    LOG_ALWAYS_FATAL_IF(outConsumer == nullptr,
            "BufferQueue: outConsumer must not be NULL");
    //创建BufferQueueCore 存储图形缓冲区的地方
    sp<BufferQueueCore> core(new BufferQueueCore());
    LOG_ALWAYS_FATAL_IF(core == nullptr,
            "BufferQueue: failed to create BufferQueueCore");
    //创建IGraphicBufferProducer 缓冲区数据生产者
    sp<IGraphicBufferProducer> producer(new BufferQueueProducer(core, consumerIsSurfaceFlinger));
    LOG_ALWAYS_FATAL_IF(producer == nullptr,
            "BufferQueue: failed to create BufferQueueProducer");
    //创建IGraphicBufferConsumer 缓冲区数据消费者
    sp<IGraphicBufferConsumer> consumer(new BufferQueueConsumer(core));
```

```
    LOG_ALWAYS_FATAL_IF(consumer == nullptr,
            "BufferQueue: failed to create BufferQueueConsumer");
    //赋值给传入的参数
    *outProducer = producer;
    *outConsumer = consumer;
}
```

到这里可以看到BufferQueue中的BufferQueueCore，生产者，消费者全部被创建出来了，Layer已经全部创建成功，可以Surface的创建我们回到
E:\AOSP\frameworks\native\libs\gui\SurfaceComposerClient.cpp

```
status_t SurfaceComposerClient::createSurfaceChecked(const String8& name, uint32_t w, uint32_t h,
                                                      PixelFormat format,
                                                      sp<SurfaceControl>* outSurface, uint32_t flags,
                                                      SurfaceControl* parent,
                                                      LayerMetadata metadata) {
    sp<SurfaceControl> sur;
    status_t err = mStatus;

    if (mStatus == NO_ERROR) {
        sp<IBinder> handle;
        sp<IBinder> parentHandle;
        sp<IGraphicBufferProducer> gbp;

        if (parent != nullptr) {
            parentHandle = parent->getHandle();
        }

        err = mClient->createSurface(name, w, h, format, flags, parentHandle, std::move(metadata),
                                     &handle, &gbp);
        ALOGE_IF(err, "SurfaceComposerClient::createSurface error %s", strerror(-err));
        if (err == NO_ERROR) {
            //创建surfaceControl，传入BufferLayer中生成的graphicBufferProducer
            *outSurface = new SurfaceControl(this, handle, gbp, true /* owned */);
        }
    }
    return err;
}
```

可以到这里可以看到SurfaceControl被创建生成出来了，具体的surface内呢？ 看到

E:\AOSP\frameworks\native\libs\gui\SurfaceControl.cpp 内部的结构：

```
sp<Surface> SurfaceControl::getSurface() const
{
    Mutex::Autolock _l(mLock);
```

```
    if (mSurfaceData == nullptr) {
        return generateSurfaceLocked();
    }
    return mSurfaceData;
}

sp<Surface> SurfaceControl::createSurface() const
{
    Mutex::Autolock _l(mLock);
    return generateSurfaceLocked();
}
//最后的最后，是通过surfaceControl来生成surface，surface的创建需要传入生成BufferQueueLayer时创建的GraphicBufferProducer
sp<Surface> SurfaceControl::generateSurfaceLocked() const
{
    // This surface is always consumed by SurfaceFlinger, so the
    // producerControlledByApp value doesn't matter; using false.
    mSurfaceData = new Surface(mGraphicBufferProducer, false);

    return mSurfaceData;
}

```

3.4 小结

- 调用windowManagerService.relayoutWindow方法，传入outSurfaceControl参数让windowManagerService内部构建后进行赋值
- WMS中顺着逻辑执行createSurfaceControl，调用WindowStateAnimator.createSurfaceLocked创建WindowSurfaceController 对象
- WindowSurfaceController 构造函数中通过 SurfaceControl.Builder创建SurfaceControl对象，并且WindowSurfaceController 会内部持有该对象
- SurfaceControl的构造函数中是通过nativeCreate在native层进行构建的，至此java层调用结束
- android_view_SurfaceControl.nativeCreate通过SurfaceComposerClient->createSurfaceChecked创建surfaceControl对象。
- SurfaceComposerClient->createSurfaceChecked最会通过Client对象远程binder调用SurfaceFlinger.createLayer方法创建BufferQueueLayer， 创建BufferQueueLayer过程中会生成BufferQueue、GraphicBufferProducer与GraphicBufferConsumer对象。
- SurfaceComposerClient会使用生成的GraphicBufferProducer以及自身对象创建SurfaceControl
- SurfaceControl对象生成后，会通过createSurface创建Surface，创建surface过程中会传入之前生成的mGraphicBufferProducer对象，至此整个Surface的创建流程完成。