

# Android Surface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:

```
1 public void handleResumeActivity(IBinder token, boolean finalStateRequest,
2     boolean isForward,
3     String reason) {
4     ...
5     final ActivityClientRecord r = performResumeActivity(token,
6     finalStateRequest, reason);
7     final Activity a = r.activity;
8     ...
9     if (r.window == null && !a.mFinished && willBeVisible) {
10         //1、获取window对象
11         r.window = r.activity.getWindow();
12         //2、获取decorView
13         View decor = r.window.getDecorView();
14         ...
15         //3、获取windowManger
16         WindowManager wm = a.getWindowManager();
17         WindowManager.LayoutParams l = r.window.getAttributes();
18         a.mDecor = decor;
19         ...
20         if (a.mVisibleFromClient) {
21             if (!a.mWindowAdded) {
22                 a.mWindowAdded = true;
23                 //4、将decor加到viewManager当中
24                 wm.addView(decor, l);
```

```

23         } else {
24             ...
25         }
26     }
27     ...
28 }
29 }

```

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

```

1  @UnsupportedAppUsage
2  private Window mWindow;
3
4  public void setContentView(@LayoutRes int layoutResID) {
5      getWindow().setContentView(layoutResID);
6      initWindowDecorActionBar();
7  }
8
9  public Window getWindow() {
10     return mWindow;
11 }

```

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：

```

1  /**
2   * Abstract base class for a top-level window look and behavior policy. An
3   * instance of this class should be used as the top-level view added to the
4   * window manager. It provides standard UI policies such as a background, title
5   * area, default key processing, etc.
6
7   * 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。
8   *
9   * <p>The only existing implementation of this abstract class is
10  * android.view.PhoneWindow, which you should instantiate when needing a
11  * Window.
12  * 该抽象类的唯一实现是android.view.PhoneWindow， 当需要window时都使用的是PhoneWindow
13  */

```

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:

```
1  /** Core implementation of activity launch. */
2  private Activity performLaunchActivity(ActivityClientRecord r, Intent
   customIntent) {
3      ...
4      //创建activity的context
5      ContextImpl appContext = createBaseContextForActivity(r);
6      Activity activity = null;
7      try {
8          java.lang.ClassLoader cl = appContext.getClassLoader();
9          //创建activity对象
10         activity = mInstrumentation.newActivity(
11             cl, component.getClassName(), r.intent);
12         ...
13     } catch (Exception e) {
14         ...
15     }
16
17     try {
18         //创建application
19         Application app = r.packageInfo.makeApplication(false,
mInstrumentation);
20         ...
21         if (activity != null) {
22             ...
23             window window = null;
24             appContext.setOuterContext(activity);
25             //关键函数 绑定context token等各种信息
26             activity.attach(appContext, this, getInstrumentation(), r.token,
27                 r.ident, app, r.intent, r.activityInfo, title,
r.parent,
28                 r.embeddedID, r.lastNonConfigurationInstances,
config,
29                 r.referrer, r.voiceInteractor, window,
r.configCallback,
30                 r.assistToken);
31             ...
32             r.activity = activity;
33         }
34         r.setState(ON_CREATE);
35     } catch (SuperNotCalledException e) {
36         throw e;
37     } catch (Exception e) {
38         ...
39     }
40     return activity;
41 }
```

Activity.attach:

```
1  @UnsupportedAppUsage
2  final void attach(Context context, ActivityThread aThread,
3                    Instrumentation instr, IBinder token, int ident,
4                    Application application, Intent intent, ActivityInfo info,
5                    CharSequence title, Activity parent, String id,
6                    NonConfigurationInstances lastNonConfigurationInstances,
7                    Configuration config, String referrer, IVoiceInteractor
8                    voiceInteractor,
9                    Window window, ActivityConfigCallback activityConfigCallback,
10                   IBinder assistToken) {
11     attachBaseContext(context);
12     //window实例化, 类型为Phonewindow
13     mWindow = new Phonewindow(this, window, activityConfigCallback);
14     ...
15     //给window设置windowManger
16     mWindow.setWindowManager(
17         (WindowManager)context.getSystemService(Context.WINDOW_SERVICE),
18         mToken, mComponent.flattenToString(),
19         (info.flags & ActivityInfo.FLAG_HARDWARE_ACCELERATED) != 0);
20     if (mParent != null) {
21         mWindow.setContainer(mParent.getWindow());
22     }
23     ...
24 }
```

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:

```
1  public void setWindowManager(WindowManager wm, IBinder appToken, String appName,
2                               boolean hardwareAccelerated) {
3     mAppToken = appToken;
4     mAppName = appName;
5     mHardwareAccelerated = hardwareAccelerated;
6     if (wm == null) {
7         wm = (WindowManager)mContext.getSystemService(Context.WINDOW_SERVICE);
8     }
9     mWindowManager = ((WindowManagerImpl)wm).createLocalWindowManager(this);
10 }
11
12 //WindowManagerImpl.createLocalWindowManager
13 public WindowManagerImpl createLocalWindowManager(Window parentWindow) {
14     return new WindowManagerImpl(mContext, parentWindow);
15 }
16
17 //WindowManagerImpl 构造类
18 private WindowManagerImpl(Context context, Window parentWindow) {
19     mContext = context;
20     mParentWindow = parentWindow;
21 }
```

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：

```

1  @Override
2  public void handleResumeActivity(IBinder token, boolean finalStateRequest,
   boolean isForward,
3                                     String reason) {
4      ...
5      final Activity a = r.activity;
6      if (r.window == null && !a.mFinished && willBeVisible) {
7          //获取window, 上面分析过, 为phonewindow对象
8          r.window = r.activity.getWindow();
9          View decor = r.window.getDecorView();
10         decor.setVisibility(View.INVISIBLE);
11         //获取viewManger, viewManger的接口在windowManger当中实现了
12         WindowManager wm = a.getWindowManager();
13         WindowManager.LayoutParams l = r.window.getAttributes();
14         a.mDecor = decor;
15         l.type = WindowManager.LayoutParams.TYPE_BASE_APPLICATION;
16         l.softInputMode |= forwardBit;
17         ...
18         if (a.mVisibleFromClient) {
19             if (!a.mWindowAdded) {
20                 a.mWindowAdded = true;
21                 //将decor添加到windowManger当中。
22                 wm.addView(decor, l);
23             } else {
24                 ...
25             }
26         }
27     } else if (!willBeVisible) {
28         ...
29     }
30     ...
31     Looper.myQueue().addIdleHandler(new Idler());
32 }
33

```

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WindowManagerImpl.addView

```
1 //windowMangerImpl.addView
```

```

2  @Override
3  public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params)
4  {
5      applyDefaultToken(params);
6      mGlobal.addView(view, params, mContext.getDisplay(), mParentWindow);
7  }
8  //这里的mGlobal为WindowManagerGlobal对象，我们继续看WindowManagerGlobal.addView:
9  public void addView(View view, ViewGroup.LayoutParams params,
10                     Display display, Window parentWindow) {
11      ...
12      ViewRootImpl root;
13      View panelParentView = null;
14      synchronized (mLock) {
15          ...
16          //1、创建ViewRootImpl对象
17          root = new ViewRootImpl(view.getContext(), display);
18          try {
19              //2、调用setView 将view添加到ViewRootImpl当中
20              root.setView(view, wparams, panelParentView);
21          } catch (RuntimeException e) {
22              ...
23          }
24      }
25  }

```

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

```

1  @UnsupportedAppUsage
2  final IWindowSession mWindowSession;
3  final W mWindow;
4  public ViewRootImpl(Context context, Display display) {
5      ...
6      //mWindowSession创建
7      mWindowSession = WindowManagerGlobal.getWindowSession();
8      ...
9      //mWindow对象创建
10     mWindow = new W(this);
11     ...
12 }

```

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

```

1 //windowManagerGlobal.getWindowSession();
2 @UnsupportedAppUsage
3 public static IWindowSession getWindowSession() {
4     synchronized (WindowManagerGlobal.class) {
5         if (sWindowSession == null) {
6             try {
7                 ...
8                 //IWindowManager为WMS的AIDL服务
9                 IWindowManager windowManager = getWindowManagerService();
10                //调用WMS.opneSession获取一个IWindowSession
11                sWindowSession = windowManager.openSession(
12                    new IWindowSessionCallback.Stub() {
13                        @Override
14                        public void onAnimationScaleChanged(float scale) {
15                            valueAnimator.setDurationScale(scale);
16                        }
17                    });
18            } catch (RemoteException e) {
19                throw e.rethrowFromSystemServer();
20            }
21        }
22        return sWindowSession;
23    }
24 }
25
26 //getWindowManagerService实现
27 @UnsupportedAppUsage
28 public static IWindowManager getWindowManagerService() {
29     synchronized (WindowManagerGlobal.class) {
30         if (sWindowManagerService == null) {
31             sWindowManagerService = IWindowManager.Stub.asInterface(
32                 ServiceManager.getService("window"));
33             ...
34         }
35         return sWindowManagerService;
36     }
37 }

```

我们看看WMS提供的openSession方法

```

1 @Override
2 public IWindowSession openSession(IWindowSessionCallback callback) {
3     return new Session(this, callback);
4 }
5
6 //那么Session到底是什么，我们看看官方释义：
7 /**
8  * This class represents an active client session. There is generally one
9  * session object per process that is interacting with the window manager.
10
11 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。

```

```

12  */
13  class Session extends IWindowSession.Stub implements IBinder.DeathRecipient
14  {
15  }

```

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:

```

1  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

```

1  /**
2   * We have one child
3   */
4  public void setView(View view, WindowManager.LayoutParams attrs, View
panelParentView) {
5      synchronized (this) {
6          if (mView == null) {
7              mView = view;
8              ...
9              //requestLayout
10             requestLayout();
11             ...
12             try {
13                 ...
14                 //mwindowSession.addToDisplay
15                 res = mwindowSession.addToDisplay(mWindow, mSeq,
mWindowAttributes,
16                                                         getHostVisibility(),
mDisplay.getDisplayId(), mTmpFrame,
17                                                         mAttachInfo.mContentInsets,
mAttachInfo.mStableInsets,
18                                                         mAttachInfo.mOutsets,
mAttachInfo.mDisplayCutout, mInputChannel,
19                                                         mTempInsets);
20                 setFrame(mTmpFrame);
21             } catch (RemoteException e) {
22                 ...

```



```

23         } finally {
24             ...
25         }
26     }
27 }
28

```

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：

```

1  //Session.addToDisplay
2  @Override
3  public int addToDisplay(IWindow window, int seq, WindowManager.LayoutParams
    attrs,
4                          int viewVisibility, int displayId, Rect outFrame, Rect
    outContentInsets,
5                          Rect outStableInsets, Rect outOutsets,
6                          DisplayCutout.ParcelableWrapper outDisplayCutout,
    InputChannel outInputChannel,
7                          InsetsState outInsetsState) {
8      return mService.addWindow(this, window, seq, attrs, viewVisibility,
    displayId, outFrame,
9                          outContentInsets, outStableInsets, outOutsets,
    outDisplayCutout, outInputChannel,
10                          outInsetsState);
11 }

```

//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requestLayout，requestLayout会把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：

```

1  @UnsupportedAppUsage
2  public final Surface mSurface = new Surface();
3  private final SurfaceControl mSurfaceControl = new SurfaceControl();
4
5  private void performTraversals() {
6      // cache mView since it is used so much below...
7      final View host = mView;
8      ...
9      //这里有一个知识点，后续面试可以追踪下，八股之一
10     // Execute enqueued actions on every traversal in case a detached view
    enqueued an action
11     getRunQueue().executeActions(mAttachInfo.mHandler);
12     if (mFirst || windowShouldResize || insetsChanged ||
13         viewVisibilityChanged || params != null || mForceNextWindowRelayout) {
14         ...
15         try {

```

```

16         ...
17         //relayoutwindow 关键函数
18         relayoutResult = relayoutWindow(params, viewVisibility,
insetsPending);
19         ...
20     } catch (RemoteException e) {
21     }
22     //ThreadedRenderer构建
23     final ThreadedRenderer threadedRenderer = mAttachInfo.mThreadedRenderer;
24     ...
25 }
26 ...
27 mIsInTraversal = false;
28 }

```

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:

```

1     private int relayoutWindow(WindowManager.LayoutParams params, int
viewVisibility,
2         boolean insetsPending) throws RemoteException {
3         ...
4         //注意这里传入了mSurfaceControl对象，mSurfaceControl是初始化时创建的
5         int relayoutResult = mWindowSession.relayout(mWindow, mSeq, params,
6             (int) (mView.getMeasuredWidth() * appScale + 0.5f),
7             (int) (mView.getMeasuredHeight() * appScale + 0.5f),
viewVisibility,
8             insetsPending ? WindowManagerGlobal.RELAYOUT_INSETS_PENDING : 0,
frameNumber,
9             mTmpFrame, mPendingOverscanInsets, mPendingContentInsets,
mPendingVisibleInsets,
10            mPendingStableInsets, mPendingOutsets, mPendingBackDropFrame,
mPendingDisplayCutout,
11            mPendingMergedConfiguration, mSurfaceControl, mTempInsets);
12         if (mSurfaceControl.isValid()) {
13             //复制surface
14             mSurface.copyFrom(mSurfaceControl);
15         } else {
16             destroySurface();
17         }
18         ...
19         return relayoutResult;
20     }

```

这里调用了mWindowSession.relayout

```

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

```

```

3         int requestedWidth, int requestedHeight, int viewFlags, int flags,
    long frameNumber,
4         Rect outFrame, Rect outOverScanInsets, Rect outContentInsets, Rect
    outVisibleInsets,
5         Rect outStableInsets, Rect outsets, Rect outBackdropFrame,
6         DisplayCutout.ParcelableWrapper cutout, MergedConfiguration
    mergedConfiguration,
7         SurfaceControl outSurfaceControl, InsetsState outInsetsState) {
8     if (false) Slog.d(TAG_WM, ">>>>> ENTERED relayout from "
9         + Binder.getCallingPid());
10    Trace.traceBegin(TRACE_TAG_WINDOW_MANAGER, mRelayoutTag);
11    //传入outSurfaceControl对象
12    int res = mService.relaxoutwindow(this, window, seq, attrs,
13        requestedWidth, requestedHeight, viewFlags, flags, frameNumber,
14        outFrame, outOverScanInsets, outContentInsets, outVisibleInsets,
15        outStableInsets, outsets, outBackdropFrame, cutout,
16        mergedConfiguration, outSurfaceControl, outInsetsState);
17    Trace.traceEnd(TRACE_TAG_WINDOW_MANAGER);
18    if (false) Slog.d(TAG_WM, "<<<<< EXITING relayout to "
19        + Binder.getCallingPid());
20    return res;
21 }

```

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.relaxoutwindow，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

```
1  /**
2   * Handle onto a raw buffer that is being managed by the screen compositor.
3   *
4   * <p>A Surface is generally created by or from a consumer of image buffers
  (such as a
5   * {@link android.graphics.SurfaceTexture}, {@link android.media.MediaRecorder},
  or
6   * {@link android.renderscript.Allocation}), and is handed to some kind of
  producer (such as
7   * {@link
  android.opengl.EGL14#eglCreateWindowSurface(android.opengl.EGLDisplay,android.op
 engl.EGLConfig,java.lang.Object,int[],int) OpenGL},
8   * {@link android.media.MediaPlayer#setSurface MediaPlayer}, or
9   * {@link android.hardware.camera2.CameraDevice#createCaptureSession
  CameraDevice}) to draw
10  * into.</p>
11  *
12  * <p><strong>Note:</strong> A Surface acts like a
13  * {@link java.lang.ref.WeakReference weak reference} to the consumer it is
  associated with. By
14  * itself it will not keep its parent consumer from being reclaimed.</p>
15  */
16  //Surface用于操纵被屏幕合成器(一般是SurfaceFlinger) 元缓冲数据
17  public class Surface implements Parcelable {
18
19  }
20  /**
21   * Handle to an on-screen Surface managed by the system compositor. The
  SurfaceControl is
22   * a combination of a buffer source, and metadata about how to display the
  buffers.
23   * By constructing a {@link Surface} from this SurfaceControl you can submit
  buffers to be
24   * composited. Using {@link SurfaceControl.Transaction} you can manipulate
  various
25   * properties of how the buffer will be displayed on-screen. SurfaceControl's
  are
26   * arranged into a scene-graph like hierarchy, and as such any SurfaceControl
  may have
27   * a parent. Geometric properties like transform, crop, and Z-ordering will be
  inherited
28   * from the parent, as if the child were content in the parents buffer stream.
29   */
30  //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。
31  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

```
1 int res = mService.relayoutWindow(this, window, seq, attrs,  
2     requestedWidth, requestedHeight, viewFlags, flags, frameNumber,  
3     outFrame, outOverScanInsets, outContentInsets, outVisibleInsets,  
4     outStableInsets, outsets, outBackdropFrame, cutout,  
5     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：

```
1 try {  
2     result = createSurfaceControl(outSurfaceControl, result, win, winAnimator);  
3 } catch (Exception e) {  
4     displayContent.getInputMonitor().updateInputWindowsLw(true /*force*/);  
5  
6     slog.w(TAG_WM, "Exception thrown when creating surface for client "  
7         + client + " (" + win.mAttrs.getTitle() + ")",  
8         e);  
9     Binder.restoreCallingIdentity(origId);  
10    return 0;  
11 }
```

查看createSurfaceControl实现：

```
1 private int createSurfaceControl(SurfaceControl outSurfaceControl, int result,  
2     WindowState win,  
3     WindowStateAnimator winAnimator) {  
4     if (!win.mHasSurface) {  
5         result |= RELAYOUT_RES_SURFACE_CHANGED;  
6     }  
7  
8     WindowSurfaceController surfaceController;  
9     try {  
10        Trace.traceBegin(TRACE_TAG_WINDOW_MANAGER, "createSurfaceControl");  
11        //创建surfaceController  
12        surfaceController = winAnimator.createSurfaceLocked(win.mAttrs.type,  
13            win.mOwnerUid);  
14    } finally {
```

```

13     Trace.traceEnd(TRACE_TAG_WINDOW_MANAGER);
14 }
15 if (surfaceController != null) {
16     //将创建好的surfaceController给到outSurfaceControl
17     surfaceController.getSurfaceControl(outSurfaceControl);
18     if (SHOW_TRANSACTIONS) Slog.i(TAG_WM, " OUT SURFACE " +
outSurfaceControl + ": copied");
19 } else {
20     // For some reason there isn't a surface. Clear the
21     // caller's object so they see the same state.
22     Slog.w(TAG_WM, "Failed to create surface control for " + win);
23     outSurfaceControl.release();
24 }
25
26 return result;
27 }

```

winAnimator为WindowStateAnimator类, 查看该类的实现

WindowSurfaceController createSurfaceLocked(int windowType, int ownerId) {

```

1 WindowSurfaceController createSurfaceLocked(int windowType, int ownerId) {
2     final WindowState w = mwin;
3
4     if (mSurfaceController != null) {
5         //如果已经创建直接返回
6         return mSurfaceController;
7     }
8     //未创建surface reset相关状态
9     w.setHasSurface(false);
10    resetDrawState();
11
12    mService.makewindowFreezingScreenIfNeededLocked(w);
13
14    int flags = SurfaceControl.HIDDEN;
15    final WindowManager.LayoutParams attrs = w.mAttrs;
16
17    if (mService.isSecureLocked(w)) {
18        flags |= SurfaceControl.SECURE;
19    }
20    //计算surface区间
21    calculateSurfaceBounds(w, attrs, mTmpSize);
22
23    // Set up surface control with initial size.
24    try {
25        final boolean isHwAccelerated = (attrs.flags & FLAG_HARDWARE_ACCELERATED) !=
0;
26        final int format = isHwAccelerated ? PixelFormat.TRANSLUCENT : attrs.format;
27        if (!PixelFormat.formatHasAlpha(attrs.format)
28            // Don't make surface with surfaceInsets opaque as they display a
29            // translucent shadow.
30            && attrs.surfaceInsets.left == 0

```

```

31         && attrs.surfaceInsets.top == 0
32         && attrs.surfaceInsets.right == 0
33         && attrs.surfaceInsets.bottom == 0
34         // Don't make surface opaque when resizing to reduce the amount of
35         // artifacts shown in areas the app isn't drawing content to.
36         && !w.isDragResizing()) {
37             flags |= SurfaceControl.OPAQUE;
38         }
39         //创建surface关键逻辑
40         mSurfaceController = new WindowSurfaceController(mSession.mSurfaceSession,
41             attrs.getTitle().toString(), width, height, format, flags, this,
42             windowType, ownerId);
43         mSurfaceController.setColorSpaceAgnostic((attrs.privateFlags
44             & WindowManager.LayoutParams.PRIVATE_FLAG_COLOR_SPACE_AGNOSTIC) !=
45             0);
46         setOffsetPositionForStackResize(false);
47         mSurfaceFormat = format;
48
49         w.setHasSurface(true);
50     } catch (OutOfResourcesException e) {
51         Slog.w(TAG, "OutOfResourcesException creating surface");
52         mService.mRoot.reclaimSomeSurfaceMemory(this, "create", true);
53         mDrawState = NO_SURFACE;
54         return null;
55     } catch (Exception e) {
56         Slog.e(TAG, "Exception creating surface (parent dead?)", e);
57         mDrawState = NO_SURFACE;
58         return null;
59     }
60     return mSurfaceController;
61 }

```

查看该方法逻辑:

public WindowSurfaceController(SurfaceSession s, String name, int w, int h, int format,

```

1 public WindowSurfaceController(SurfaceSession s, String name, int w, int h, int
  format,
2     int flags, WindowStateAnimator animator, int windowType, int ownerId) {
3     mAnimator = animator;
4
5     mSurfaceW = w;
6     mSurfaceH = h;
7
8     title = name;
9
10    mService = animator.mService;
11    final WindowState win = animator.mWin;
12    mWindowType = windowType;
13    mWindowSession = win.mSession;
14

```

```

15     Trace.traceBegin(TRACE_TAG_WINDOW_MANAGER, "new SurfaceControl");
16     //可以看到WindowSurfaceController核心就是创建一个SurfaceControl对象,
17     final SurfaceControl.Builder b = win.makeSurface()
18         .setParent(win.getSurfaceControl())
19         .setName(name)
20         .setBufferSize(w, h)
21         .setFormat(format)
22         .setFlags(flags)
23         .setMetadata(METADATA_WINDOW_TYPE, windowType)
24         .setMetadata(METADATA_OWNER_UID, ownerId);
25     mSurfaceControl = b.build();
26     Trace.traceEnd(TRACE_TAG_WINDOW_MANAGER);
27 }

```

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() {

```

1  @NonNull
2  public SurfaceControl build() {
3      if (mWidth < 0 || mHeight < 0) {
4          throw new IllegalStateException(
5              "width and height must be positive or unset");
6      }
7      if ((mWidth > 0 || mHeight > 0) && (isColorLayerSet() ||
isContainerLayerSet())) {
8          throw new IllegalStateException(
9              "Only buffer layers can set a valid buffer size.");
10     }
11     //直接使用SurfaceControl的构造类使用
12     return new SurfaceControl(
13         mSession, mName, mWidth, mHeight, mFormat, mFlags, mParent,
mMetadata);
14 }

```

```

1  private SurfaceControl(SurfaceSession session, String name, int w, int h, int
format, int flags,
2      SurfaceControl parent, SparseIntArray metadata)
3      throws OutOfResourcesException, IllegalArgumentException {
4
5      mName = name;
6      mWidth = w;
7      mHeight = h;
8      Parcel metaParcel = Parcel.obtain();
9      try {
10         if (metadata != null && metadata.size() > 0) {
11             metaParcel.writeInt(metadata.size());
12             for (int i = 0; i < metadata.size(); ++i) {
13                 metaParcel.writeInt(metadata.keyAt(i));

```





```

21     }
22 }
23 //使用SurfaceComposerClient创建surface
24 status_t err = client->createSurfaceChecked(
25     String8(name.c_str()), w, h, format, &surface, flags, parent,
std::move(metadata));
26 if (err == NAME_NOT_FOUND) {
27     jniThrowException(env, "java/lang/IllegalArgumentException", NULL);
28     return 0;
29 } else if (err != NO_ERROR) {
30     jniThrowException(env, OutOfResourcesException, NULL);
31     return 0;
32 }
33
34 surface->incStrong((void *)nativeCreate);
35 return reinterpret_cast<jlong>(surface.get());
36 }

```

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

```

1  status_t SurfaceComposerClient::createSurfaceChecked(const String8& name,
    uint32_t w, uint32_t h,
2                                     PixelFormat format,
3                                     sp<SurfaceControl>*
    outSurface, uint32_t flags,
4                                     SurfaceControl* parent,
5                                     LayerMetadata metadata) {
6      sp<SurfaceControl> sur;
7      status_t err = mStatus;
8
9      if (mStatus == NO_ERROR) {
10         sp<IBinder> handle;
11         sp<IBinder> parentHandle;
12         sp<IGraphicBufferProducer> gbp;
13
14         if (parent != nullptr) {
15             parentHandle = parent->getHandle();
16         }
17         //SurfaceComposerClient中调用mClient->createSurface创建surface
18         err = mClient->createSurface(name, w, h, format, flags, parentHandle,
std::move(metadata),
19                                     &handle, &gbp);
20         ALOGE_IF(err, "SurfaceComposerClient::createSurface error %s",
strerror(-err));
21         if (err == NO_ERROR) {
22             *outSurface = new SurfaceControl(this, handle, gbp, true /* owned
*/);
23         }
24     }

```

```

25     return err;
26 }

```

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：

```

1  status_t Client::createSurface(const String8& name, uint32_t w, uint32_t h,
    PixelFormat format,
2                                  uint32_t flags, const sp<IBinder>& parentHandle,
3                                  LayerMetadata metadata, sp<IBinder>* handle,
4                                  sp<IGraphicBufferProducer>* gbp) {
5      // we rely on createLayer to check permissions.
6      //使用mFlinger.createLayer方法创建surface，mFlinger为SurfaceFlinger的智能指针
7      return mFlinger->createLayer(name, this, w, h, format, flags,
    std::move(metadata), handle, gbp,
8                                  parentHandle);
9  }

```

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

```

1  status_t SurfaceFlinger::createLayer(const String8& name, const sp<Client>&
    client, uint32_t w,
2                                  uint32_t h, PixelFormat format, uint32_t
    flags,
3                                  LayerMetadata metadata, sp<IBinder>*
    handle,
4                                  sp<IGraphicBufferProducer>* gbp,
5                                  const sp<IBinder>& parentHandle,
6                                  const sp<Layer>& parentLayer) {
7      //...省略非关键代码
8
9      sp<Layer> layer;
10
11     String8 uniqueName = getUniqueLayerName(name);
12
13     switch (flags & ISurfaceComposerClient::eFXSurfaceMask) {
14         case ISurfaceComposerClient::eFXSurfaceBufferQueue:
15             result = createBufferQueueLayer(client, uniqueName, w, h, flags,
    std::move(metadata),
16                                     format, handle, gbp, &layer);
17
18             break;
19         case ISurfaceComposerClient::eFXSurfaceBufferState:
20             result = createBufferStateLayer(client, uniqueName, w, h, flags,
    std::move(metadata),
21                                     handle, &layer);
22             break;
23         case ISurfaceComposerClient::eFXSurfaceColor:
24             // check if buffer size is set for color layer.
25             if (w > 0 || h > 0) {

```

```

26         ALOGE("createLayer() failed, w or h cannot be set for color
layer (w=%d, h=%d)",
27             int(w), int(h));
28         return BAD_VALUE;
29     }
30
31     result = createColorLayer(client, uniqueName, w, h, flags,
std::move(metadata), handle,
32         &layer);
33     break;
34     case ISurfaceComposerClient::eFXSurfaceContainer:
35         // check if buffer size is set for container layer.
36         if (w > 0 || h > 0) {
37             ALOGE("createLayer() failed, w or h cannot be set for container
layer (w=%d, h=%d)",
38                 int(w), int(h));
39             return BAD_VALUE;
40         }
41         result = createContainerLayer(client, uniqueName, w, h, flags,
std::move(metadata),
42             handle, &layer);
43         break;
44     default:
45         result = BAD_VALUE;
46         break;
47     }
48
49     if (result != NO_ERROR) {
50         return result;
51     }
52
53     if (primaryDisplayOnly) {
54         layer->setPrimaryDisplayOnly();
55     }
56
57     bool addToCurrentState = callingThreadHasUnscopedSurfaceFlingerAccess();
58     result = addClientLayer(client, *handle, *gbp, layer, parentHandle,
parentLayer,
59         addToCurrentState);
60     if (result != NO_ERROR) {
61         return result;
62     }
63     mInterceptor->saveSurfaceCreation(layer);
64
65     setTransactionFlags(eTransactionNeeded);
66     return result;
67 }

```

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`

```

1  status_t SurfaceFlinger::createBufferQueueLayer(const sp<Client>& client, const
    String8& name,
2
    uint32_t w, uint32_t h, uint32_t
    flags,
3
    LayerMetadata metadata,
    PixelFormat& format,
4
    sp<IBinder>* handle,
5
    sp<IGraphicBufferProducer>* gbp,
6
    sp<Layer>* outLayer) {
7
    // initialize the surfaces
8
    switch (format) {
9
    case PIXEL_FORMAT_TRANSPARENT:
10
    case PIXEL_FORMAT_TRANSLUCENT:
11
        format = PIXEL_FORMAT_RGBA_8888;
12
        break;
13
    case PIXEL_FORMAT_OPAQUE:
14
        format = PIXEL_FORMAT_RGBX_8888;
15
        break;
16
    }
17
18
    sp<BufferQueueLayer> layer = getFactory().createBufferQueueLayer(
19
        LayerCreationArgs(this, client, name, w, h, flags,
        std::move(metadata)));
20
    status_t err = layer->setDefaultBufferProperties(w, h, format);
21
    if (err == NO_ERROR) {
22
        *handle = layer->getHandle();
23
        //graphicBufferProducer从创建的layer中获取
24
        *gbp = layer->getProducer();
25
        //将创建的layer传递给传入的layer参数指针。
    }
}

```

```

26         *outLayer = layer;
27     }
28
29     ALOGE_IF(err, "createBufferQueueLayer() failed (%s)", strerror(-err));
30     return err;
31 }

```

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

```

1  sp<BufferQueueLayer> createBufferQueueLayer(const LayerCreationArgs& args)
   override {
2      return new BufferQueueLayer(args);
3  }

```

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

```

1  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

```

1  BufferLayer::BufferLayer(const LayerCreationArgs& args)
2      : Layer(args),
3        mTextureName(args.flinger->getNewTexture()),
4        mCompositionLayer{mFlinger->getCompositionEngine().createLayer(
5            compositionengine::LayerCreationArgs{this})} {
6      ALOGV("Creating Layer %s", args.name.string());
7
8      mPremultipliedAlpha = !(args.flags &
9          ISurfaceComposerClient::eNonPremultiplied);
10
11      mPotentialCursor = args.flags & ISurfaceComposerClient::eCursorWindow;
12      mProtectedByApp = args.flags & ISurfaceComposerClient::eProtectedByApp;
13  }

```

继续看起父类Layer的实现

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

```

1  Layer::Layer(const LayerCreationArgs& args)
2      : mFlinger(args.flinger),
3        mName(args.name),
4        mClientRef(args.client),
5        mWindowType(args.metadata.getInt32(METADATA_WINDOW_TYPE, 0)) {
6      mCurrentCrop.makeInvalid();
7
8      uint32_t layerFlags = 0;

```

```

9     if (args.flags & ISurfaceComposerClient::eHidden) layerFlags |=
layer_state_t::eLayerHidden;
10    if (args.flags & ISurfaceComposerClient::eOpaque) layerFlags |=
layer_state_t::eLayerOpaque;
11    if (args.flags & ISurfaceComposerClient::eSecure) layerFlags |=
layer_state_t::eLayerSecure;
12
13    mTransactionName = String8("TX - ") + mName;
14    //各种状态赋值
15    mCurrentState.active_legacy.w = args.w;
16    mCurrentState.active_legacy.h = args.h;
17    mCurrentState.flags = layerFlags;
18    mCurrentState.active_legacy.transform.set(0, 0);
19    mCurrentState.crop_legacy.makeInvalid();
20    mCurrentState.requestedCrop_legacy = mCurrentState.crop_legacy;
21    mCurrentState.z = 0;
22    mCurrentState.color.a = 1.0f;
23    mCurrentState.layerStack = 0;
24    mCurrentState.sequence = 0;
25    mCurrentState.requested_legacy = mCurrentState.active_legacy;
26    mCurrentState.active.w = UINT32_MAX;
27    mCurrentState.active.h = UINT32_MAX;
28    mCurrentState.active.transform.set(0, 0);
29    mCurrentState.transform = 0;
30    mCurrentState.transformToDisplayInverse = false;
31    mCurrentState.crop.makeInvalid();
32    mCurrentState.acquireFence = new Fence(-1);
33    mCurrentState.dataspace = ui::Dataspace::UNKNOWN;
34    mCurrentState.hdrMetadata.validTypes = 0;
35    mCurrentState.surfaceDamageRegion.clear();
36    mCurrentState.cornerRadius = 0.0f;
37    mCurrentState.api = -1;
38    mCurrentState.hasColorTransform = false;
39    mCurrentState.colorSpaceAgnostic = false;
40    mCurrentState.metadata = args.metadata;
41
42    // drawing state & current state are identical
43    mDrawingState = mCurrentState;
44
45    CompositorTiming compositorTiming;
46    args.flinger->getCompositorTiming(&compositorTiming);
47    mFrameEventHistory.initializeCompositorTiming(compositorTiming);
48    mFrameTracker.setDisplayRefreshPeriod(compositorTiming.interval);
49    //注册Layer
50    mSchedulerLayerHandle = mFlinger->mScheduler->registerLayer(mName.c_str(),
mWindowType);
51    //回调LayerCreated
52    mFlinger->onLayerCreated();
53 }

```

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

```

1  BufferLayer::BufferLayer(const LayerCreationArgs& args)
2      : Layer(args),
3        mTextureName(args.flinger->getNewTexture()),
4        //这里调用mFlinger->getCompositionEngine().createLayer
5        mCompositionLayer{mFlinger->getCompositionEngine().createLayer(
6            compositionengine::LayerCreationArgs{this})} {
7      ALOGV("Creating Layer %s", args.name.string());
8
9      mPremultipliedAlpha = !(args.flags &
10         ISurfaceComposerClient::eNonPremultiplied);
11
12      mPotentialCursor = args.flags & ISurfaceComposerClient::eCursorWindow;
13      mProtectedByApp = args.flags & ISurfaceComposerClient::eProtectedByApp;
14  }

```

mFlinger->getCompositionEngine()方法

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

```

1  const CompositionEngine& Output::getCompositionEngine() const {
2      return mCompositionEngine;
3  }

```

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

```

1  std::shared_ptr<compositionengine::Layer>
2  CompositionEngine::createLayer(LayerCreationArgs&& args) {
3      return compositionengine::impl::createLayer(*this, std::move(args));
4  }

```

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

```

1  std::shared_ptr<compositionengine::Layer> createLayer(
2      const compositionengine::CompositionEngine& compositionEngine,
3      compositionengine::LayerCreationArgs&& args) {
4      return std::make_shared<Layer>(compositionEngine, std::move(args));
5  }

```

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



```

1 void BufferQueueLayer::onFirstRef() {
2     BufferLayer::onFirstRef();
3
4     // Creates a custom BufferQueue for SurfaceFlingerConsumer to use
5     sp<IGraphicBufferProducer> producer;
6     sp<IGraphicBufferConsumer> consumer;
7     //创建bufferQueue,传入producer与consumer的引用
8     BufferQueue::createBufferQueue(&producer, &consumer, true);
9     mProducer = new MonitoredProducer(producer, mFlinger, this);
10    {
11        // Grab the SF state lock during this since it's the only safe way to
12        // access RenderEngine
13        Mutex::Autolock lock(mFlinger->mStateLock);
14        mConsumer =
15            new BufferLayerConsumer(consumer, mFlinger->getRenderEngine(),
16            mTextureName, this);
17    }
18    mConsumer->setConsumerUsageBits(getEffectiveUsage(0));
19    mConsumer->setContentsChangedListener(this);
20    mConsumer->setName(mName);
21
22    // BufferQueueCore::mMaxDequeuedBufferCount is default to 1
23    if (!mFlinger->isLayerTripleBufferingDisabled()) {
24        mProducer->setMaxDequeuedBufferCount(2);
25    }
26
27    if (const auto display = mFlinger->getDefaultDisplayDevice()) {
28        updateTransformHint(display);
29    }
30 }

```

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

```

1 void BufferQueue::createBufferQueue(sp<IGraphicBufferProducer>* outProducer,
2     sp<IGraphicBufferConsumer>* outConsumer,
3     bool consumerIsSurfaceFlinger) {
4     LOG_ALWAYS_FATAL_IF(outProducer == nullptr,
5         "BufferQueue: outProducer must not be NULL");
6     LOG_ALWAYS_FATAL_IF(outConsumer == nullptr,
7         "BufferQueue: outConsumer must not be NULL");
8     //创建BufferQueueCore 存储图形缓冲区的地方
9     sp<BufferQueueCore> core(new BufferQueueCore());
10    LOG_ALWAYS_FATAL_IF(core == nullptr,
11        "BufferQueue: failed to create BufferQueueCore");
12    //创建IGraphicBufferProducer 缓冲区数据生产者
13    sp<IGraphicBufferProducer> producer(new BufferQueueProducer(core,
14        consumerIsSurfaceFlinger));
15    LOG_ALWAYS_FATAL_IF(producer == nullptr,
16        "BufferQueue: failed to create BufferQueueProducer");
17    //创建IGraphicBufferConsumer 缓冲区数据消费者
18    sp<IGraphicBufferConsumer> consumer(new BufferQueueConsumer(core));
19 }

```

```

18     LOG_ALWAYS_FATAL_IF(consumer == nullptr,
19         "BufferQueue: failed to create BufferQueueConsumer");
20     //赋值给传入的参数
21     *outProducer = producer;
22     *outConsumer = consumer;
23 }

```

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

```

1  status_t SurfaceComposerClient::createSurfaceChecked(const String8& name,
    uint32_t w, uint32_t h,
2                                     PixelFormat format,
3                                     sp<SurfaceControl>*
    outSurface, uint32_t flags,
4                                     SurfaceControl* parent,
5                                     LayerMetadata metadata) {
6      sp<SurfaceControl> sur;
7      status_t err = mStatus;
8
9      if (mStatus == NO_ERROR) {
10         sp<IBinder> handle;
11         sp<IBinder> parentHandle;
12         sp<IGraphicBufferProducer> gbp;
13
14         if (parent != nullptr) {
15             parentHandle = parent->getHandle();
16         }
17
18         err = mClient->createSurface(name, w, h, format, flags, parentHandle,
    std::move(metadata),
19                                     &handle, &gbp);
20         ALOGE_IF(err, "SurfaceComposerClient::createSurface error %s",
    strerror(-err));
21         if (err == NO_ERROR) {
22             //创建surfaceControl，传入BufferLayer中生成的graphicBufferProducer
23             *outSurface = new SurfaceControl(this, handle, gbp, true /* owned
    */);
24         }
25     }
26     return err;
27 }

```

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

回到上层调用

E:\AOSP\frameworks\base\core\java\android\view\ViewRootImpl.java

```

1 private int relayoutWindow(WindowManager.LayoutParams params, int
viewVisibility,
2     boolean insetsPending) throws RemoteException {
3
4     int relayoutResult = mWindowSession.relayout(mWindow, mSeq, params,
5         (int) (mView.getMeasuredWidth() * appScale + 0.5f),
6         (int) (mView.getMeasuredHeight() * appScale + 0.5f), viewVisibility,
7         insetsPending ? WindowManager.Global.RELAYOUT_INSETS_PENDING : 0,
frameNumber,
8         mTmpFrame, mPendingOverscanInsets, mPendingContentInsets,
mPendingVisibleInsets,
9         mPendingStableInsets, mPendingOutsets, mPendingBackDropFrame,
mPendingDisplayCutout,
10        mPendingMergedConfiguration, mSurfaceControl, mTempInsets);
11     if (mSurfaceControl.isValid()) {
12         //mSurfaceControl创建成功
13         mSurface.copyFrom(mSurfaceControl);
14     } else {
15         destroySurface();
16     }
17     return relayoutResult;
18 }

```

E:\AOSP\frameworks\base\core\java\android\view\Surface.java

```

1 @UnsupportedAppUsage
2 public void copyFrom(SurfaceControl other) {
3     if (other == null) {
4         throw new IllegalArgumentException("other must not be null");
5     }
6
7     long surfaceControlPtr = other.mNativeObject;
8     if (surfaceControlPtr == 0) {
9         throw new NullPointerException(
10             "null surfaceControl native object. Are you using a released
SurfaceControl?");
11     }
12     //这里是关键
13     long newNativeObject = nativeGetFromSurfaceControl(mNativeObject,
surfaceControlPtr);
14
15     synchronized (mLock) {
16         if (newNativeObject == mNativeObject) {
17             return;
18         }
19         if (mNativeObject != 0) {
20             nativeRelease(mNativeObject);
21         }
22         setNativeObjectLocked(newNativeObject);
23     }
24 }

```

E:\AOSP\frameworks\base\core\jni\android\_view\_Surface.cpp

```
1 static jlong nativeGetFromSurfaceControl(JNIEnv* env, jclass clazz,
2     jlong nativeObject,
3     jlong surfaceControlNativeObj) {
4     Surface* self(reinterpret_cast<Surface*>(nativeObject));
5     sp<SurfaceControl> ctrl(reinterpret_cast<SurfaceControl*>
6         (surfaceControlNativeObj));
7
8     // If the underlying IGBP's are the same, we don't need to do anything.
9     if (self != nullptr &&
10         IInterface::asBinder(self->getIGraphicBufferProducer()) ==
11         IInterface::asBinder(ctrl->getIGraphicBufferProducer())) {
12         return nativeObject;
13     }
14     //调用SurfaceControl.getSurface获得surface
15     sp<Surface> surface(ctrl->getSurface());
16     if (surface != NULL) {
17         surface->incStrong(&sRefBaseOwner);
18     }
19     return reinterpret_cast<jlong>(surface.get());
20 }
```

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

```
1 sp<Surface> SurfaceControl::getSurface() const
2 {
3     Mutex::Autolock _l(mLock);
4     if (mSurfaceData == nullptr) {
5         //创建surface
6         return generateSurfaceLocked();
7     }
8     return mSurfaceData;
9 }
10
11 //最后的最后，是通过surfaceControl来生成surface，surface的创建需要传入生成
12 //BufferQueueLayer时创建的GraphicBufferProducer
13 sp<Surface> SurfaceControl::generateSurfaceLocked() const
14 {
15     // This surface is always consumed by SurfaceFlinger, so the
16     // producerControlledByApp value doesn't matter; using false.
17     mSurfaceData = new Surface(mGraphicBufferProducer, false);
18     return mSurfaceData;
19 }
20
```

至此整个surface的创建过程完成

### 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。