Android 深入浅出之 Surface

一 目的

本节的目的就是为了讲清楚 Android 中的 Surface 系统，大家耳熟能详的 SurfaceFlinger 到底是个什么东西，它的工作流程又是怎样的。当然，鉴于 SurfaceFlinger 的复杂性，我们依然将采用情景分析的办法，找到合适的切入点。

一个 Activity 是怎么在屏幕上显示出来的呢？我将首先把这个说清楚。

接着我们把其中的关键调用抽象在 Native 层，以这些函数调用为切入点来研究 SurfaceFlinger。好了，开始我们的征途吧。

二 Activity 是如何显示的

最初的想法就是，Activity 获得一块显存，然后在上面绘图，最后交给设备去显示。这个道理是没错，但是 Android 的 SurfaceFlinger 是在 System Server 进程中创建的，Activity 一般另有线程，这之间是如何...如何挂上关系的呢？我可以先提前告诉大家，这个过程还比较复杂。呵呵。

好吧，我们从 Activity 最初的启动开始。代码在

framework/base/core/java/android/app/ActivityThread.java 中，这里有个函数叫 handleLaunchActivity

[---->ActivityThread:: handleLaunchActivity()]

```
private final void handleLaunchActivity(ActivityRecord r, Intent customIntent) {
     Activity a = performLaunchActivity(r, customIntent);

       if (a != null) {
           r.createdConfig = new Configuration(mConfiguration);
           Bundle oldState = r.state;
           handleResumeActivity(r.token, false, r.isForward);
       ---->调用 handleResumeActivity
    }
```

handleLaunchActivity 中会调用 handleResumeActivity。

[--->ActivityThread:: handleResumeActivity]

```
final void handleResumeActivity(IBinder token, boolean clearHide, boolean isForward) {
        boolean willBeVisible = !a.mStartedActivity;

        if (r.window == null && !a.mFinished && willBeVisible) {
               r.window = r.activity.getWindow();
               View decor = r.window.getDecorView();
```

```
                decor.setVisibility(View.INVISIBLE);
                ViewManager wm = a.getWindowManager();
                WindowManager.LayoutParams l = r.window.getAttributes();
                a.mDecor = decor;
                l.type = WindowManager.LayoutParams.TYPE_BASE_APPLICATION;
                if (a.mVisibleFromClient) {
                    a.mWindowAdded = true;
                    wm.addView(decor, l); //这个很关键。
                }
```

上面 addView 那几行非常关键，它关系到咱们在 Activity 中 setContentView 后，整个 Window 到底都包含了些什么。我先告诉大家。所有你创建的 View 之上，还有一个 DecorView，这是一个 FrameLayout，另外还有一个 PhoneWindow。上面这些东西的代码在

framework/Policies/Base/Phone/com/android/Internal/policy/impl。这些隐藏的 View 的创建都是由你在 Acitivty 的 onCreate 中调用 setContentView 导致的。

[---->PhoneWindow:: addContentView]

```
public void addContentView(View view, ViewGroup.LayoutParams params) {
    if (mContentParent == null) {  //刚创建的时候 mContentParent 为空
        installDecor();
    }
    mContentParent.addView(view, params);
    final Callback cb = getCallback();
    if (cb != null) {
        cb.onContentChanged();
    }
}
installDecor 将创建 mDecor 和 mContentParent。mDecor 是 DecorView 类型，
mContentParent 是 ViewGroup 类型
private void installDecor() {
        if (mDecor == null) {
            mDecor = generateDecor();
            mDecor.setDescendantFocusability(ViewGroup.FOCUS_AFTER_DESCENDANTS);
            mDecor.setIsRootNamespace(true);
        }
        if (mContentParent == null) {
            mContentParent = generateLayout(mDecor);
```

那么，ViewManager wm = a.getWindowManager()又返回什么呢？

PhoneWindow 从 Window 中派生，Acitivity 创建的时候会调用它的 setWindowManager。而

这个函数由 Window 类实现。

代码在 framework/base/core/java/android/view/Window.java 中

```
public void setWindowManager(WindowManager wm,IBinder appToken, String appName) {
        mAppToken = appToken;
        mAppName = appName;
        if (wm == null) {
            wm = WindowManagerImpl.getDefault();
        }
        mWindowManager = new LocalWindowManager(wm);
    }
```

你看见没，分析 JAVA 代码这个东西真的很复杂。mWindowManager 的实现是 LocalWindowManager，但由通过 Bridge 模式把功能交给 WindowManagerImpl 去实现了。

真的很复杂!

好了，罗里罗嗦的，我们回到 wm.addView(decor, l)。最终会由 WindowManagerImpl 来完成 addView 操作，我们直接看它的实现好了。

代码在 framework/base/core/java/android/view/WindowManagerImpl.java

[---->addView]

```
private void addView(View view, ViewGroup.LayoutParams params, boolean nest)
    {
          ViewRoot root; //ViewRoot，我们的主人公终于登场!
         synchronized (this) {
         root = new ViewRoot(view.getContext());
          root.mAddNesting = 1;
          view.setLayoutParams(wparams);

            if (mViews == null) {
                index = 1;
                mViews = new View[1];
                mRoots = new ViewRoot[1];
                mParams = new WindowManager.LayoutParams[1];
            } else {
           }
            index--;
            mViews[index] = view;
            mRoots[index] = root;
            mParams[index] = wparams;
        }
```

```
        root.setView(view, wparams, panelParentView);
    }
```

ViewRoot 是整个显示系统中最为关键的东西，看起来这个东西好像和 View 有那么点关系，其实它根本和 View 等 UI 关系不大，它不过是一个 Handler 罢了，唯一有关系的就是它其中有一个变量为 Surface 类型。我们看看它的定义。ViewRoot 代码在

framework/base/core/java/android/view/ViewRoot.java 中

```
public final class ViewRoot extends Handler implements ViewParent,
        View.AttachInfo.Callbacks
{
private final Surface mSurface = new Surface();
}
它竟然从 handler 派生，而 ViewParent 不过定义了一些接口函数罢了。
```

看到 Surface 直觉上感到它和 SurfaceFlinger 有点关系。要不先去看看？

Surface 代码在 framework/base/core/java/android/view/Surface.java 中，我们调用的是无参构造函数。

```
public Surface() {
        mCanvas = new CompatibleCanvas(); //就是创建一个 Canvas!
    }
```

如果你有兴趣的话，看看 Surface 其他构造函数，最终都会调用 native 的实现，而这些 native 的实现将和 SurfaceFlinger 建立关系，但我们这里 ViewRoot 中的 mSurface 显然还没有到这一步。那它到底是怎么和 SurfaceFlinger 搞上的呢？这一切待会就会水落石出的。

另外，为什么 ViewRoot 是主人公呢？因为 ViewRoot 建立了客户端和 SystemServer 的关系。我们看看它的构造函数。

```
public ViewRoot(Context context) {
        super();
       ....
         getWindowSession(context.getMainLooper());
}
getWindowsession 将建立和 WindowManagerService 的关系。
ublic static IWindowSession getWindowSession(Looper mainLooper) {
        synchronized (mStaticInit) {
            if (!mInitialized) {
                try {
                //sWindowSession 是通过 Binder 机制创建的。终于让我们看到点希望了
                    InputMethodManager imm = InputMethodManager.getInstance(mainLooper);
                    sWindowSession = IWindowManager.Stub.asInterface(
                            ServiceManager.getService("window"))
```

```
                            .openSession(imm.getClient(), imm.getInputContext());
                    mInitialized = true;
                } catch (RemoteException e) {
                }
            }
            return sWindowSession;
        }
    }
```

上面跨 Binder 的进程调用另一端是 WindowManagerService，代码在 framework/base/services/java/com/android/server/WindowManagerService.java 中。我们先不说这个。

回过头来看看 ViewRoot 接下来的调用。

[-->ViewRoot::setView()]，这个函数很复杂，我们看其中关键几句。

```
public void setView(View view, WindowManager.LayoutParams attrs,
            View panelParentView) {
        synchronized (this) {
            requestLayout();
                try {
                    res = sWindowSession.add(mWindow, mWindowAttributes,
                            getHostVisibility(), mAttachInfo.mContentInsets);
                }
}
```

requestLayout 实现很简单，就是往 handler 中发送了一个消息。

```
public void requestLayout() {
        checkThread();
        mLayoutRequested = true;
        scheduleTraversals(); //发送 DO_TRAVERSAL 消息
    }
    public void scheduleTraversals() {
        if (!mTraversalScheduled) {
            mTraversalScheduled = true;
            sendEmptyMessage(DO_TRAVERSAL);
        }
    }
```

我们看看跨进程的那个调用。sWindowSession.add。它的最终实现在 WindowManagerService 中。

[--->WindowSession::add()]

```
public int add(IWindow window, WindowManager.LayoutParams attrs,
                int viewVisibility, Rect outContentInsets) {
            return addWindow(this, window, attrs, viewVisibility, outContentInsets);
        }
```

WindowSession 是个内部类，会调用外部类的 addWindow

这个函数巨复杂无比，但是我们的核心目标是找到创建显示相关的部分。所以，最后精简的话就简单了。

[--->WindowManagerService:: addWindow]

```
public int addWindow(Session session, IWindow client,
            WindowManager.LayoutParams attrs, int viewVisibility,
            Rect outContentInsets) {
        //创建一个 WindowState，这个又是什么玩意儿呢?
              win = new WindowState(session, client, token,
                    attachedWindow, attrs, viewVisibility);
           win.attach();
           return res;
    }
```

WindowState 类中有一个和 Surface 相关的成员变量，叫 SurfaceSession。它会在

attach 函数中被创建。SurfaceSession 嘛，就和 SurfaceFlinger 有关系了。我们待会看。

好，我们知道 ViewRoot 创建及调用 add 后，我们客户端的 View 系统就和 WindowManagerService 建立了牢不可破的关系。

另外，我们知道 ViewRoot 是一个 handler，而且刚才我们调用了 requestLayout，所以接下来消息循环下一个将调用的就是 ViewRoot 的 handleMessage。

```
public void handleMessage(Message msg) {
        switch (msg.what) {
       case DO_TRAVERSAL:
            performTraversals();
```

performTraversals 更加复杂无比，经过我仔细挑选，目标锁定为下面几个函数。当然，后面我们还会回到 performTraversals，不过我们现在更感兴趣的是 Surface 是如何创建的。

```
private void performTraversals() {
        // cache mView since it is used so much below...
        final View host = mView;


         boolean initialized = false;
            boolean contentInsetsChanged = false;
            boolean visibleInsetsChanged;
            try {
```

```
//ViewRoot 也有一个 Surface 成员变量，叫 mSurface，这个就是代表 SurfaceFlinger 的客户端
//ViewRoot 在这个 Surface 上作画，最后将由 SurfaceFlinger 来合成显示。刚才说了 mSurface 还没有什么内容。
          relayoutResult = relayoutWindow(params, viewVisibility, insetsPending);
```

[---->ViewRoot:: relayoutWindow()]

```
 private int relayoutWindow(WindowManager.LayoutParams params, int viewVisibility,
            boolean insetsPending) throws RemoteException {


//relayOut 是跨进程调用，mSurface 做为参数传进去了，看来离真相越来越近了呀!
        int relayoutResult = sWindowSession.relayout(
                mWindow, params,
                (int) (mView.mMeasuredWidth * appScale + 0.5f),
                (int) (mView.mMeasuredHeight * appScale + 0.5f),
                viewVisibility, insetsPending, mWinFrame,
                mPendingContentInsets, mPendingVisibleInsets,
                mPendingConfiguration, mSurface); mSurface 做为参数传进去了。
      }
```

我们赶紧转到 WindowManagerService 去看看吧。、

```
public int relayoutWindow(Session session, IWindow client,
            WindowManager.LayoutParams attrs, int requestedWidth,
            int requestedHeight, int viewVisibility, boolean insetsPending,
            Rect outFrame, Rect outContentInsets, Rect outVisibleInsets,
            Configuration outConfig, Surface outSurface){
               .....
         try {
           //看到这里，我内心一阵狂喜，有戏，太有戏了!
         //其中 win 是我们最初创建的 WindowState!
                    Surface surface = win.createSurfaceLocked();
                    if (surface != null) {
                  //先创建一个本地 surface，然后把传入的参数 outSurface copyFrom 一下
                        outSurface.copyFrom(surface);
                        win.mReportDestroySurface = false;
                        win.mSurfacePendingDestroy = false;
                      } else {
                      outSurface.release();
                    }
```

```
                }
}
```

[--->WindowState::createSurfaceLocked]

```
Surface createSurfaceLocked() {


                try {
                    mSurface = new Surface(
                            mSession.mSurfaceSession, mSession.mPid,
                            mAttrs.getTitle().toString(),
                            0, w, h, mAttrs.format, flags);
                  }
                Surface.openTransaction();
```

这里使用了 Surface 的另外一个构造函数。

```
    public Surface(SurfaceSession s,
             int pid, String name, int display, int w, int h, int format, int flags)
        throws OutOfResourcesException {
        mCanvas = new CompatibleCanvas();
        init(s,pid,name,display,w,h,format,flags); ----->调用了 native 的 init 函数。
        mName = name;
    }
```

到这里，不进入 JNI 是不可能说清楚了。不过我们要先回顾下之前的关键步骤。

- add 中，new 了一个 SurfaceSession
- 创建 new 了一个 Surface
- 调用 copyFrom，把本地 Surface 信息传到 outSurface 中

JNI 层

上面两个类的 JNI 实现都在 framework/base/core/jni/android_view_Surface.cpp 中。

[---->SurfaceSession:: SurfaceSession()]

```
public class SurfaceSession {
    /** Create a new connection with the surface flinger. */
    public SurfaceSession() {
        init();
    }
```

它的 init 函数对应为：

[--->SurfaceSession_init]

```
static void SurfaceSession_init(JNIEnv* env, jobject clazz)
{
   //SurfaceSession 对应为 SurfaceComposerClient
```

```
    sp<SurfaceComposerClient> client = new SurfaceComposerClient;
    client->incStrong(clazz);
    //Google 常用做法，在 JAVA 对象中保存 C++对象的指针。
    env->SetIntField(clazz, sso.client, (int)client.get());
}
```

Surface 的 init 对应为：

[--->Surface_init]

```
static void Surface_init(
        JNIEnv* env, jobject clazz,
        jobject session,
        jint pid, jstring jname, jint dpy, jint w, jint h, jint format, jint flags)
{
   SurfaceComposerClient* client =
           (SurfaceComposerClient*)env->GetIntField(session, sso.client);

    sp<SurfaceControl> surface;
    if (jname == NULL) {
       //client 是 SurfaceComposerClient，返回的 surface 是一个 SurfaceControl
      //真得很复杂!
        surface = client->createSurface(pid, dpy, w, h, format, flags);
    } else {
        const jchar* str = env->GetStringCritical(jname, 0);
        const String8 name(str, env->GetStringLength(jname));
        env->ReleaseStringCritical(jname, str);
        surface = client->createSurface(pid, name, dpy, w, h, format, flags);
    }
      //把 surfaceControl 信息设置到 Surface 对象中
    setSurfaceControl(env, clazz, surface);
}
```

```
static void setSurfaceControl(JNIEnv* env, jobject clazz,
        const sp<SurfaceControl>& surface)
{
    SurfaceControl* const p =
        (SurfaceControl*)env->GetIntField(clazz, so.surfaceControl);
    if (surface.get()) {
```

```
        surface->incStrong(clazz);
    }
    if (p) {
        p->decStrong(clazz);
    }
    env->SetIntField(clazz, so.surfaceControl, (int)surface.get());
}
```

[--->Surface_copyFrom]

```
static void Surface_copyFrom(
        JNIEnv* env, jobject clazz, jobject other)
{
   const sp<SurfaceControl>& surface = getSurfaceControl(env, clazz);
    const sp<SurfaceControl>& rhs = getSurfaceControl(env, other);
    if (!SurfaceControl::isSameSurface(surface, rhs)) {
        setSurfaceControl(env, clazz, rhs);
//把本地那个 surface 的 surfaceControl 对象转移到 outSurface 上
    }
}
```

这里仅仅是 surfaceControl 的转移，但是并没有看到 Surface 相关的信息。

那么 Surface 在哪里创建的呢？为了解释这个问题，我使用了终极武器，aidl。

1 终极武器 AIDL

aidl 可以把 XXX.aidl 文件转换成对应的 java 文件。我们刚才调用的是 WindowSession 的 relayOut 函数。如下：

```
sWindowSession.relayout(
                mWindow, params,
                (int) (mView.mMeasuredWidth * appScale + 0.5f),
                (int) (mView.mMeasuredHeight * appScale + 0.5f),
                viewVisibility, insetsPending, mWinFrame,
                mPendingContentInsets, mPendingVisibleInsets,
                mPendingConfiguration, mSurface);
```

它的 aidl 文件在 framework/base/core/java/android/view/IWindowSession.aidl 中

```
interface IWindowSession {
    int add(IWindow window, in WindowManager.LayoutParams attrs,
            in int viewVisibility, out Rect outContentInsets);
    void remove(IWindow window);
   //注意喔，这个 outSurface 前面的是 out，表示输出参数，这个类似于 C++的引用。
```

```
   int relayout(IWindow window, in WindowManager.LayoutParams attrs,
            int requestedWidth, int requestedHeight, int viewVisibility,
            boolean insetsPending, out Rect outFrame, out Rect outContentInsets,
            out Rect outVisibleInsets, out Configuration outConfig,
            out Surface outSurface);
```

刚才说了，JNI 及其 JAVA 调用只是 copyFrom 了 SurfaceControl 对象到 outSurface 中，但是没看到哪里创建 Surface。这其中的奥秘就在 aidl 文件编译后生成的 java 文件中。

你在命令行下可以输入：

```
aidl -Id:\android-2.2-froyo-20100625-source\source\frameworks\base\core\java\ -Id:\android-2.2-froyo-20100625-source\source\frameworks\base\Graphics\java d:\android-2.2-froyo-20100625-source\source\frameworks\base\core\java\android\view\IWindowSession.aidl test.java
```

以生成 test.java 文件。-I 参数指定 include 目录，例如 aidl 有些参数是在别的 java 文件中指定的，那么这个-I 就需要把这些目录包含进来。

先看看 ViewRoot 这个客户端生成的代码是什么。

```
public int relayout(
 android.view.IWindow window,
 android.view.WindowManager.LayoutParams attrs,
  int requestedWidth, int requestedHeight,
 int viewVisibility, boolean insetsPending,
   android.graphics.Rect outFrame,
   android.graphics.Rect outContentInsets,
      android.graphics.Rect outVisibleInsets,
  android.content.res.Configuration outConfig,
  android.view.Surface outSurface) ---->outSurface 是第 11 个参数
  throws android.os.RemoteException
{
android.os.Parcel _data = android.os.Parcel.obtain();
android.os.Parcel _reply = android.os.Parcel.obtain();
int _result;
try {
_data.writeInterfaceToken(DESCRIPTOR);
_data.writeStrongBinder((((window!=null))?(window.asBinder()):(null)));
if ((attrs!=null)) {
_data.writeInt(1);
attrs.writeToParcel(_data, 0);
}
else {
```

```
_data.writeInt(0);
}
_data.writeInt(requestedWidth);
_data.writeInt(requestedHeight);
_data.writeInt(viewVisibility);
_data.writeInt(((insetsPending)?(1):(0)));
 //奇怪，outSurface 的信息没有写到_data 中。那.....
mRemote.transact(Stub.TRANSACTION_relayout, _data, _reply, 0);
_reply.readException();
_result = _reply.readInt();
if ((0!=_reply.readInt())) {
outFrame.readFromParcel(_reply);
}
....
if ((0!=_reply.readInt())) {
outSurface.readFromParcel(_reply); //从 Parcel 中读取信息来填充 outSurface
}
}
finally {
_reply.recycle();
_data.recycle();
}
return _result;
}
```

真奇怪啊，Binder 客户端这头竟然没有把 outSurface 的信息发过去。我们赶紧看看服务端。服务端这边处理是在 onTranscat 函数中。

```
@Override public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel
reply, int flags) throws android.os.RemoteException
{
switch (code)
{
case TRANSACTION_relayout:
{
data.enforceInterface(DESCRIPTOR);
android.view.IWindow _arg0;
android.view.Surface _arg10;
//刚才说了，Surface 信息并没有传过来，那么我们在 relayOut 中看到的 outSurface 是怎么
```

```
//出来的呢？看下面这句，原来在服务端这边竟然 new 了一个新的 Surface！！！
_arg10 = new android.view.Surface();
int _result = this.relayout(_arg0, _arg1, _arg2, _arg3, _arg4, _arg5, _arg6, _arg7, _arg8,
_arg9, _arg10);
reply.writeNoException();
reply.writeInt(_result);
//_arg10 是 copyFrom 了，那怎么传到客户端呢？
if ((_arg10!=null)) {
reply.writeInt(1);//调用 Surface 的 writeToParcel，把信息加入 reply
_arg10.writeToParcel(reply, android.os.Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
}
return true;
}
```

太诡异了！竟然有这么多花花肠子。我相信如果没有 aidl 的帮助，我无论如何也不会知道这其中的奥妙。

那好，我们的流程明白了。

- 客户端虽然传了一个 surface，但其实没传递给服务端
- 服务端调用 writeToParcel，把信息写到 Parcel 中，然后数据传回客户端
- 客户端调用 Surface 的 readFromParcel，获得 surface 信息。

那就去看看 writeToParcel 吧。

[---->Surface_writeToParcel]

```
static void Surface_writeToParcel(
        JNIEnv* env, jobject clazz, jobject argParcel, jint flags)
{
    Parcel* parcel = (Parcel*)env->GetIntField(
            argParcel, no.native_parcel);

    const sp<SurfaceControl>& control(getSurfaceControl(env, clazz));
    //还好，只是把数据序列化到 Parcel 中
    SurfaceControl::writeSurfaceToParcel(control, parcel);
    if (flags & PARCELABLE_WRITE_RETURN_VALUE) {
        setSurfaceControl(env, clazz, 0);
    }
}
```

那看看客户端的 Surface_readFromParcel 吧。

[----->Surface_readFromParcel]

```
static void Surface_readFromParcel(
        JNIEnv* env, jobject clazz, jobject argParcel)
```

```
{
    Parcel* parcel = (Parcel*)env->GetIntField( argParcel, no.native_parcel);

   //客户端这边还没有 surface 呢
    const sp<Surface>& control(getSurface(env, clazz));
    //不过我们看到希望了，根据服务端那边 Parcel 信息来构造一个新的 surface
    sp<Surface> rhs = new Surface(*parcel);
    if (!Surface::isSameSurface(control, rhs)) {
        setSurface(env, clazz, rhs); //把这个新 surface 赋给客户端。终于我们有了 surface!
    }
}
```

到此，我们终于七拐八绕的得到了 surface，这其中经历太多曲折了。下一节，我们将精简这其中复杂的操作，统一归到 Native 层，以这样为切入点来了解 Surface 的工作流程和原理。

好，反正你知道 ViewRoot 调用了 relayout 后，Surface 就真正从 WindowManagerService 那得到了。继续回到 ViewRoot，其中还有一个重要地方是我们知道却不了解的。

```
private void performTraversals() {
        // cache mView since it is used so much below...
        final View host = mView;

         boolean initialized = false;
            boolean contentInsetsChanged = false;
            boolean visibleInsetsChanged;
            try {
          relayoutResult = relayoutWindow(params, viewVisibility, insetsPending);
       // relayoutWindow 完后，我们得到了一个无比宝贵的 Surface
      //那我们画界面的地方在哪里？就在这个函数中，离 relayoutWindow 不远处。
        ....
  boolean cancelDraw = attachInfo.mTreeObserver.dispatchOnPreDraw();

        if (!cancelDraw && !newSurface) {
            mFullRedrawNeeded = false;
            draw(fullRedrawNeeded); //draw?draw 什么呀?
       }
```

[--->ViewRoot::draw()]

```
private void draw(boolean fullRedrawNeeded) {
        Surface surface = mSurface; //嘿嘿，不担心了，surface 资源都齐全了
```

```
        if (surface == null || !surface.isValid()) {
            return;
        }
        if (mAttachInfo.mViewScrollChanged) {
            mAttachInfo.mViewScrollChanged = false;
            mAttachInfo.mTreeObserver.dispatchOnScrollChanged();
        }

        int yoff;
        final boolean scrolling = mScroller != null && mScroller.computeScrollOffset();
        if (scrolling) {
            yoff = mScroller.getCurrY();
        } else {
            yoff = mScrollY;
        }
        if (mCurScrollY != yoff) {
            mCurScrollY = yoff;
            fullRedrawNeeded = true;
        }
        float appScale = mAttachInfo.mApplicationScale;
        boolean scalingRequired = mAttachInfo.mScalingRequired;

        Rect dirty = mDirty;
        if (mUseGL) { //我们不用 OPENGL
           ...
        }

         Canvas canvas;
        try {
            int left = dirty.left;
            int top = dirty.top;
            int right = dirty.right;
            int bottom = dirty.bottom;
          //从 Surface 中锁定一块区域，这块区域是我们认为的需要重绘的区域
            canvas = surface.lockCanvas(dirty);
            // TODO: Do this in native
```

```
            canvas.setDensity(mDensity);
        }


        try {
            if (!dirty.isEmpty() || mIsAnimating) {
                long startTime = 0L;
                try {
                    canvas.translate(0, -yoff);
                    if (mTranslator != null) {
                        mTranslator.translateCanvas(canvas);
                    }
                    canvas.setScreenDensity(scalingRequired
                            ? DisplayMetrics.DENSITY_DEVICE : 0);
                 //mView就是之前的decoreView,
                    mView.draw(canvas);
                }
            } finally {
             //我们的图画完了，告诉surface释放这块区域
            surface.unlockCanvasAndPost(canvas);
        }
        if (scrolling) {
            mFullRedrawNeeded = true;
            scheduleTraversals();
        }
    }
```

看起来，这个 surface 的用法很简单嘛：

- lockSurface，得到一个画布 Canvas
- 调用 View 的 draw，让他们在这个 Canvas 上尽情绘图才。另外，这个 View 会调用所有它的子 View 来画图，最终会进入到 View 的 onDraw 函数中，在这里我们可以做定制化的界面美化工作。当然，如果你想定制化整个系统画图的话，完全可以把 performTranvsal 看懂，然后再修改。
- unlockCanvasAndPost，告诉 Surface 释放这块画布

当然，这几个重要函数调用干了具体的活。这些重要函数，我们最终会精简到 Native 层的。

2 总结

到这里，你应该知道了一个 Activity 中，调用 setContentView 后它如何从系统中获取一块 Surface，以及它是如何使用这个 Surface 的了。不得不说，关于 UI 这块，Android 绝对是够复杂的。难怪 2.3 把 UI 这块代码基本重写一遍，希望能够简单精炼点。

三 简化的流程

上面很多代码都是在 JAVA 中转来转去，但重要的工作又是在 Native 层完成的，JAVA 就像雾一样遮住了真相。我们总结下上节的流程，把它精简于 Native 层中。

- 先创建一个 SurfaceSession
- 以 SurfaceSession 等为参数，构造一个 Surface 对象 A
- 通过 Surface 的 copyFrom 把 A 对象的信息传递到服务端的 outSurface
- AIDL 生成的 java 文件，我们发现服务端会把 outSurface 的信息通过 writeToParcel 函数写到 Parcel 中，跨进程传递信息
- 客户端通过 surface 的 readFromParcel 把信息读出来，传给 ViewRoot 的 mSurface 对象。
- 客户端画图前先调用 lockCanvas
- UI 画图
- 调用 unlockCanvasAndPost 完成这次绘图

这些 JNI 层代码都在 framework/base/core/jni/android_view_Surface.cpp 中，我们一步一步来看。

3.1 SurfaceSession

SurfaceSession 的构造函数会调用 Native 的 SurfaceSession_init 函数。

```
static void SurfaceSession_init(JNIEnv* env, jobject clazz)
{
    sp<SurfaceComposerClient> client = new SurfaceComposerClient;
    //sp 使用让人讨厌的地方，要自己主动调用 incStrong，否则函数调用完，实际对象就被
    //干掉了
    client->incStrong(clazz);

    env->SetIntField(clazz, sso.client, (int)client.get());
}
```

1 创建 SurfaceComposerClient

上面 new 了一个 SurfaceComposerClient，名字怪怪的，其实就是 SurfaceFlinger 的 client，因为 SurfaceFlinger 派生于 SurfaceComposer。

SurfaceComposerClient 的代码在 framework/base/libs/surfaceflinger_client.cpp 中

```
SurfaceComposerClient::SurfaceComposerClient()
{
   // getComposerService()返回 SurfaceFlinger 的代理 Binder
   sp<ISurfaceComposer> sm(getComposerService());
   //先调用 SF 的 createConnection，再调用_init
   _init(sm, sm->createConnection());

    if (mClient != 0) {
```

```
        Mutex::Autolock _l(gLock);
       //gActiveConnections 是全局变量，把刚才创建的 client 保存到这个 map 中去
        gActiveConnections.add(mClient->asBinder(), this);
    }
}
```

没办法，只能到 SF 中去看看 createConnection 函数了。SF 的代码是 framework/base/libs/surfaceFlinger.cpp 中。

[----->SurfaceFlinger::createConnection()]

```
sp<ISurfaceFlingerClient> SurfaceFlinger::createConnection()
{
    Mutex::Autolock _l(mStateLock);
    uint32_t token = mTokens.acquire();

    sp<Client> client = new Client(token, this);

    status_t err = mClientsMap.add(token, client);

    sp<BClient> bclient =
        new BClient(this, token, client->getControlBlockMemory());
    return bclient;
}
```

SF 和 AF 在有些地方上是出奇得相似，先搞一个内部 client，然后再搞一个支持跨进程的 BClient。回想下 AF，Video 的绘制也是需要内存的嘛，所以也会存在一个跨进程的共享内存以及一些同步用的锁。当然，SF 的 client 不是 SurfaceFlinger 的内部类。

我们看看 client 的构造。

```
Client::Client(ClientID clientID, const sp<SurfaceFlinger>& flinger)
    : ctrlblk(0), cid(clientID), mPid(0), mBitmap(0), mFlinger(flinger)
{
    const int pgsize = getpagesize();
    const int cblksize = ((sizeof(SharedClient)+(pgsize-1))&~(pgsize-1));

    mCblkHeap = new MemoryHeapBase(cblksize, 0,
            "SurfaceFlinger Client control-block");

    ctrlblk = static_cast<SharedClient *>(mCblkHeap->getBase());
    if (ctrlblk) {
        new(ctrlblk) SharedClient; //placement new。
```

```
    }
}
```

上面最有价值的是在共享内存中创建了一个 SharedClient。SF 比较难懂的主要原因并不在于它的流程，而是在于它诸多奇奇怪怪的名字和数据结构，个人感觉好像都是没有怎么认真思考的结果，这点和 MFC 的风格比起来相差甚远。AF 在这点上感觉比 SF 有进步。SharedClient，名字是有一个 Client，但实际上是一个由 SF 和 SF 客户端使用的关键数据结构。

2 SharedClient

SharedClient 定义了一些成员变量，我们可以看看。

```
class SharedClient
{
public:
    SharedClient();
    ~SharedClient();
    status_t validate(size_t token) const;
    uint32_t getIdentity(size_t token) const;

private:
    Mutex lock;
    Condition cv;
    // NUM_LAYERS_MAX 为 31
    SharedBufferStack surfaces[ NUM_LAYERS_MAX ];
};
SharedClient::SharedClient()
    : lock(Mutex::SHARED), cv(Condition::SHARED) //跨进程的同步对象
{
}
```

整体来说，SharedClient 封装下跨进程的同步对象，另外它还提供了 SharedBufferStack 数组，共 31 个元素。我们先不解释这些东西了，以后用到了我们再看。

3 重回 createConnection

```
sp<ISurfaceFlingerClient> SurfaceFlinger::createConnection() {

    sp<BClient> bclient =
        new BClient(this, token, client->getControlBlockMemory());
    return bclient;
}
```

BClient 就是封装下跨进程的数据之类的。我们以后碰到再看。反正你知道

SurfaceComposerClient 调用完 createConnection 后，将返回 BClient。

[--->SurfaceComposerClient::_init()]

```
void SurfaceComposerClient::_init(
        const sp<ISurfaceComposer>& sm, const sp<ISurfaceFlingerClient>& conn)
{
    mPrebuiltLayerState = 0;
    mTransactionOpen = 0;
    mStatus = NO_ERROR;
    mControl = 0;

    mClient = conn;//mClient 就是 BClient 的客户端
    mControlMemory = mClient->getControlBlock();
    mSignalServer = sm;// mSignalServer 是 SF 的 Binder 客户端
    //mControl 就是那个跨进程的 SharedClient
    mControl = static_cast<SharedClient *>(mControlMemory->getBase());
}
```

SurfaceComposerClient 创建完后，我们发现它在 SF 中获得了一个相应的 BClient 等东西。根据我们的精简流程，SurfaceComposerClient 对象会保存在 JAVA SurfaceSession 中。

下一步将是根据这个 SurfaceSession，构造一个新的 Surface。

3.2 Surface

```
static void Surface_init(
        JNIEnv* env, jobject clazz,
        jobject session,
        jint pid, jstring jname, jint dpy, jint w, jint h, jint format, jint flags)
{
    //从 JAVA SurfaceSession 中得到开始创建的 SurfaceComposerClient 对象
    SurfaceComposerClient* client =
            (SurfaceComposerClient*)env->GetIntField(session, sso.client);
    sp<SurfaceControl> surface;
    if (jname == NULL) {
        surface = client->createSurface(pid, dpy, w, h, format, flags);
    } else {
        const jchar* str = env->GetStringCritical(jname, 0);
        const String8 name(str, env->GetStringLength(jname));
        env->ReleaseStringCritical(jname, str);
        surface = client->createSurface(pid, name, dpy, w, h, format, flags);
```

```
    }
```

其中有个参数 dpy 是 displayid 之意，这个东西很重要。为什么呢？

现在手机经常有什么内屏外屏之分，而 Android 支持 4 个屏，这些屏就由这个 displayid 来区分。当然，目前的 Android 只支持一块屏，所以 dpy 为零。这是由 WindowMnagaerService 调用传进来的。看下面的调用代码。

[--->WinState:: createSurfaceLocked()]

```
mSurface = new Surface(
                            mSession.mSurfaceSession, mSession.mPid,
                            mAttrs.getTitle().toString(),
                            0, //displayId 为 0
                            w, h, mAttrs.format, flags);
```

```
    setSurfaceControl(env, clazz, surface);
}
```

好了。我们看看调用 client 的 createSurface 函数吧。这是一个跨进程的调用，具体实现在 SF 中。多希望此书也能有两个屏幕啊，这样可以在另外一个屏写 SF 的内容。

这里的 Client 是 SurfaceComposerClient。

```
sp<SurfaceControl> SurfaceComposerClient::createSurface(
        int pid,
        const String8& name,
        DisplayID display,
        uint32_t w,
        uint32_t h,
        PixelFormat format,
        uint32_t flags)
{
    sp<SurfaceControl> result;
    if (mStatus == NO_ERROR) {
        ISurfaceFlingerClient::surface_data_t data;
    //调用 BClient 的 createSurface
        sp<ISurface> surface = mClient->createSurface(&data, pid, name,
                display, w, h, format, flags);
        if (surface != 0) {
            if (uint32_t(data.token) < NUM_LAYERS_MAX) {
             //根据返回的 ISurface 接口，构造 SurfaceControl 对象
            //不知道为什么 Android 搞这么多对象干嘛，我感觉很多地方用不上
           //而且变量的名字也挺让人晕乎的，到处都是 Surface....
```

```
                result = new SurfaceControl(this, surface, data, w, h, format, flags);
            }
        }
    }
    return result;
}
```

BClient 一般都是把请求转给 SF。

```
//真正的处理在 SF 的 createSurface 中
sp<ISurface> SurfaceFlinger::createSurface(ClientID clientId, int pid,
        const String8& name, ISurfaceFlingerClient::surface_data_t* params,
        DisplayID d, uint32_t w, uint32_t h, PixelFormat format,
        uint32_t flags)
{
    sp<LayerBaseClient> layer;
    sp<LayerBaseClient::Surface> surfaceHandle;
   //关键无比的数据类型 LayerBaseClient 和 Surface，但同样也是无比晕乎的数据类型！！！
    Mutex::Autolock _l(mStateLock);
    sp<Client> client = mClientsMap.valueFor(clientId);
     int32_t id = client->generateId(pid);
    if (uint32_t(id) >= NUM_LAYERS_MAX) {
       一个 client 最多能创建 31 个 Layer。
        return surfaceHandle;
    }
//每次 id 都会递增，从 0 到 30。第一次进来，id 为 0。
 //我们的 flags 为 0，所以会创建
    switch (flags & eFXSurfaceMask) {
        case eFXSurfaceNormal:
            if (UNLIKELY(flags & ePushBuffers)) {
                layer = createPushBuffersSurfaceLocked(client, d, id,
                        w, h, flags);
            } else {
                layer = createNormalSurfaceLocked(client, d, id,
                        w, h, flags, format);
            }
            break;
        case eFXSurfaceBlur:
```

```
            layer = createBlurSurfaceLocked(client, d, id, w, h, flags);
            break;
        case eFXSurfaceDim:
            layer = createDimSurfaceLocked(client, d, id, w, h, flags);
            break;
    }
```

Android 定义了几种不同的 Surface。上面那个枚举的定义如下：

```
    enum { // (keep in sync with Surface.java)
        eHidden             = 0x00000004,
        eDestroyBackbuffer  = 0x00000020,
        eSecure             = 0x00000080,
        eNonPremultiplied   = 0x00000100,
        ePushBuffers        = 0x00000200,

        eFXSurfaceNormal    = 0x00000000,
        eFXSurfaceBlur      = 0x00010000,
        eFXSurfaceDim       = 0x00020000,
        eFXSurfaceMask      = 0x000F0000,
    };
```

其代码在 framework/base/include/surfaceFlinger/ISurfaceComposer.h 中。

一般接触较多的是 eFXSurfaceNormal 这个对应大多数的情况，另外一个就是 ePushBuffers，它主要用在 Camera 等由底层直接绘图的地方，也就是我们在 SDK 中碰到的 SurfaceView。虽然类型不同，但是不影响我们分析 SF。所以我们暂时以 eFXSurfaceNormal 为主。

接下来会调用 createNormalSurfaceLocked。

[---->SurfaceFlinger::createNormalSurfaceLocked()]

```
sp<LayerBaseClient> SurfaceFlinger::createNormalSurfaceLocked(
        const sp<Client>& client, DisplayID display,
        int32_t id, uint32_t w, uint32_t h, uint32_t flags,
        PixelFormat& format)
{
   //注意我们的参数，display 是 0，id 是生成的 layer 号，如果是第一次的话也是 0.
    // initialize the surfaces
    switch (format) { // TODO: take h/w into account
    case PIXEL_FORMAT_TRANSPARENT:
    case PIXEL_FORMAT_TRANSLUCENT:
        format = PIXEL_FORMAT_RGBA_8888;
        break;
```

```
    case PIXEL_FORMAT_OPAQUE:
        format = PIXEL_FORMAT_RGB_565;
        break;
    }
   //唉，又来一个类型，Layer。注意我们的参数，display=id=0
    sp<Layer> layer = new Layer(this, display, client, id);
    status_t err = layer->setBuffers(w, h, format, flags);
    if (LIKELY(err == NO_ERROR)) {
        layer->initStates(w, h, flags);
        addLayer_l(layer);
    } else {
         layer.clear();
    }
    return layer;
}
```

这里涉及到一个比较重要的数据结构 layer。我们看看它。

1 Layer

这里类之间的继承关系比较繁琐。

```
class Layer : public LayerBaseClient      ---->Layer 从 LayerBaseClient 派生
class LayerBaseClient : public LayerBase -----><LayerBaseClient 从 LayerBase 派生
名字让人巨愤怒，到处是 Client 的命名，Layer 也是让人摸不着头脑。
```

LayerBase 和它的子类们是 SF 中最重要的东西，名字确实让人很生气（不仅仅是晕，痛苦了），一个 LayerBase，你把它看成是按 z 序排列的一块屏幕好了，然后客户端程序就在这个 LayerBase 上作画，最后由 SF 按照 Z 序来对这些 LayerBase 上的内容进行混合。这么说的话，其实 SF 应该叫 LayerBaseFlinger。

刚才也说到，不同 Surface 类型其实也对应了不同的 LayerBase。这块的关系过于复杂。

Normal 的 Surface 对应 Layer 类，我们从这里进去，步步深入。

先看看 Layer 的构造函数吧。代码在 framework/base/libs/surfaceflinger/layer.cpp

[---->Layer::Layer]

```
Layer::Layer(SurfaceFlinger* flinger, DisplayID display,
        const sp<Client>& c, int32_t i)
    :   LayerBaseClient(flinger, display, c, i),
        mSecure(false),
        mNoEGLImageForSwBuffers(false),
        mNeedsBlending(true),
        mNeedsDithering(false)
{
```

```
    // lcblk 是基类 LayerBaseClient 的成员变量，不得以啊。
    mFrontBufferIndex = lcblk->getFrontBuffer();
}
```

[--->LayerBaseClient::LayerBaseClient()]

代码在 framework/base/libs/surfaceflinger/layerBase.cpp 中

```
LayerBaseClient::LayerBaseClient(SurfaceFlinger* flinger, DisplayID display,
        const sp<Client>& client, int32_t i)
    : LayerBase(flinger, display), lcblk(NULL), client(client), mIndex(i),
      mIdentity(uint32_t(android_atomic_inc(&sIdentity)))
{


    lcblk = new SharedBufferServer(
            client->ctrlblk, i, NUM_BUFFERS, //注意，i=0。NUM_BUFFERS=2
            mIdentity);
}
```

client->ctrlblk 实际就是先前创建的 SharedClient 结构，这里以这个结构为核心，构造了一个 SharedBufferServer。等下会看到一个 SharedBufferClient，这个是在客户端构建的，也是以这个跨进程的 SharedClient 为核心的结构。到这里，大家应该能猜想到大概的情况了：

- SF 以 SharedBufferServer 工作
- 客户端围绕 SharedBufferClient 工作
- 二者以同一个跨进程的共享 SharedClient 来工作

NUM_BUFFERS 在 framework/base/libs/surfaceflinger/layer.h 中定义，该值为 2。

我们看看 SharedBufferServer 又是什么玩意儿。它从 SharedBufferBase 派生而来。

[--->SharedBufferServer:: SharedBufferServer()]

```
SharedBufferServer::SharedBufferServer(SharedClient* sharedClient,
        int surface, int num, int32_t identity)
    : SharedBufferBase(sharedClient, surface, num, identity)
{
   //一个 SharedClient 有一个 31 元素的 SharedStack 数组
  //一个 SharedBufferServer 其实用得最重要的东西是这个 SharedStack 数组中的一个
  //SharedStack 到底干什么用的呢？从 Layer 这个类来说，它相当于一个缓存的读写控制
  //比如 SF 读到哪个缓存了，SF 客户端可以往哪个缓存写了，当前缓存还有多少 buffer 等
  //注意这里的参数，num 为 2，
    mSharedStack->init(identity);
    mSharedStack->head = num-1;  //head 为 1
    mSharedStack->available = num;  //available 为 2，表明有 2 块 buffer
    mSharedStack->queued = 0;
```

```
    mSharedStack->reallocMask = 0;
    memset(mSharedStack->dirtyRegion, 0, sizeof(mSharedStack->dirtyRegion));
}
```

没办法，只能跟进去了。

```
SharedBufferBase::SharedBufferBase(SharedClient* sharedClient,
        int surface, int num, int32_t identity)
    : mSharedClient(sharedClient),
      mSharedStack(sharedClient->surfaces + surface),
      mNumBuffers(num), mIdentity(identity)
{
  //sharedClient 中不是定义了一个 SharedBufferStack surfaces[ NUM_LAYERS_MAX ]吗。
 //sharedBufferBase 中的 mSharedStack 实际值就是上面那个 31 元数组中的一个。
//因为传进来的 surface 为 0，所以 mSharedStack 实际等于 surfaces[0]
}
```

回到 Layer 的构造函数。

```
Layer::Layer(SurfaceFlinger* flinger, DisplayID display,
        const sp<Client>& c, int32_t i)
    :   LayerBaseClient(flinger, display, c, i),
        mSecure(false),
        mNoEGLImageForSwBuffers(false),
        mNeedsBlending(true),
        mNeedsDithering(false)
{
   // lcblk 是一个 SharedBufferServer。
   mFrontBufferIndex = lcblk->getFrontBuffer();
   //OK，它会取出对应 stack 的 head 值，我们刚才计算过了，所以 mFrontBufferIndex=1
}
```

咱们回到

[---->SurfaceFlinger::createNormalSurfaceLocked()]

```
sp<LayerBaseClient> SurfaceFlinger::createNormalSurfaceLocked(
        const sp<Client>& client, DisplayID display,
        int32_t id, uint32_t w, uint32_t h, uint32_t flags,
        PixelFormat& format)
{
    // initialize the surfaces
    sp<Layer> layer = new Layer(this, display, client, id);
```

```
    //这里将 layer 强化了，导致会调用 layer 的 onFirstRef
```

由 LayerBaseClient 实现

[--->LayerBaseClient::onFirstRef]

```
void LayerBaseClient::onFirstRef()
{
    sp<Client> client(this->client.promote());
    if (client != 0) {
        client->bindLayer(this, mIndex);
    }
}
```

[--->client::bindLayer]

```
status_t Client::bindLayer(const sp<LayerBaseClient>& layer, int32_t id)
{
    ssize_t idx = mInUse.indexOf(id);
    return mLayers.insertAt(layer, idx); //把这个 layer 加到 client 的信息中。
}
```

上面那个 onFirstRef，将 layer 和 Client 绑定到一起。但是我们发现 Layer 并没有创建内存之类的东西。当然，这个操作在 setBuffers 完成。

[---->Layer::setBuffers()]

```
status_t Layer::setBuffers( uint32_t w, uint32_t h,
                            PixelFormat format, uint32_t flags)
{
    // this surfaces pixel format
    PixelFormatInfo info;
    status_t err = getPixelFormatInfo(format, &info);
    if (err) return err;

    // the display's pixel format
    const DisplayHardware& hw(graphicPlane(0).displayHardware());
    uint32_t const maxSurfaceDims = min(
            hw.getMaxTextureSize(), hw.getMaxViewportDims());

     if ((uint32_t(w)>maxSurfaceDims) || (uint32_t(h)>maxSurfaceDims)) {
        return BAD_VALUE;
    }
```

```
    PixelFormatInfo displayInfo;
    getPixelFormatInfo(hw.getFormat(), &displayInfo);
    const uint32_t hwFlags = hw.getFlags();

    mFormat = format;
    mWidth  = w;
    mHeight = h;
    mSecure = (flags & ISurfaceComposer::eSecure) ? true : false;
    mNeedsBlending = (info.h_alpha - info.l_alpha) > 0;
    mNoEGLImageForSwBuffers = !(hwFlags & DisplayHardware::CACHED_BUFFERS);

    // we use the red index
    int displayRedSize = displayInfo.getSize(PixelFormatInfo::INDEX_RED);
    int layerRedsize = info.getSize(PixelFormatInfo::INDEX_RED);
    mNeedsDithering = layerRedsize > displayRedSize;

    for (size_t i=0 ; i<NUM_BUFFERS ; i++) {
        mBuffers[i] = new GraphicBuffer(); //创建两个 GraphicBuffer
    }
     //唉，又来 SurfaceLayer。难道 Google 开发者自己不会搞乱吗?
    mSurface = new SurfaceLayer(mFlinger, clientIndex(), this);
    return NO_ERROR;
}
```

GraphicBuffer，这个名字还好理解，咱们就把它当做可以在里边画画的内存吧，然后 SF 就从这里把数据读出去写到驱动中就行了。我们不去深究 GraphicBuffer 的内容，它对我们的流程分析没有影响。

SurfaceLayer 是个什么玩意？我真的有点想抛刀子了...

2 Surface

前面我们讲到，不同类型的 Surface 类型对应不同的 LayerBase，大家有没有觉得奇怪呢，这个名字太不对应了，Surface 对应 LayerBase，有点不合理。你这么想就对了，每一个 LayerBaseClient（注意，不是 LayerBase）中还有一个 Surface 类型的对象，为何要整个这个东西呢？我也不知道为什么。我们只能分析它存在的意义了。

Surface 从 BnSurface 中派生，一看这个就是跨进程使用的。它的定义在 LayerBaseClient 类中，是一个内部类。

```
class Surface : public BnSurface
    {
    public:
        int32_t getToken() const { return mToken; }
```

```
        int32_t getIdentity() const { return mIdentity; }

    protected:
        Surface(const sp<SurfaceFlinger>& flinger,
                SurfaceID id, int identity,
                const sp<LayerBaseClient>& owner);
        virtual ~Surface();
        virtual status_t onTransact(uint32_t code, const Parcel& data,
                Parcel* reply, uint32_t flags);
        sp<LayerBaseClient> getOwner() const;

    private:
        virtual sp<GraphicBuffer> requestBuffer(int index, int usage);
        virtual status_t registerBuffers(const ISurface::BufferHeap& buffers);
        virtual void postBuffer(ssize_t offset);
        virtual void unregisterBuffers();
        virtual sp<OverlayRef> createOverlay(uint32_t w, uint32_t h,
                int32_t format, int32_t orientation);

    protected:
        friend class LayerBaseClient;
        sp<SurfaceFlinger>  mFlinger;
        int32_t             mToken;
        int32_t             mIdentity;
        wp<LayerBaseClient> mOwner; //mOwner, 给了你什么启发吗?
    };
```

你看看 Surface 的成员函数非常少，基本上都是和 Buffer 相关的（createOverlay 我们这里不说）。我个人觉得 Surface 存在的意思是：由于画图方面基本上是内存的读写，所以如果把 LayerBase 对象实现为跨进程的话开销巨大，而且有些功能没必要在客户端使用。所以 Surface 封装了基本的所有的这些基本操作。但是作为 SF 来说，一个简单的 Surface 是不足以支持系统工作的，所以 Surface 作为 LayerBaseClient 的内部类，通过 Surface 的 mOwner 变量来指向 LayerBaseClient。当然，不同的 LayerBase 类型会创建不同的 Surface。

这里需要 google 检讨下 LayerBaseClient 和 LayerBase 这两个东西的命名，难道没有代码审查嘛。

创建完 Layer 后，我们回到 createNormalSurfaceLocked。

[---->SurfaceFlinger::createNormalSurfaceLocked()]

```
sp<LayerBaseClient> SurfaceFlinger::createNormalSurfaceLocked(
        const sp<Client>& client, DisplayID display,
```

```
        int32_t id, uint32_t w, uint32_t h, uint32_t flags,
        PixelFormat& format)
{
    ..................
    sp<Layer> layer = new Layer(this, display, client, id);
    status_t err = layer->setBuffers(w, h, format, flags);
    if (LIKELY(err == NO_ERROR)) {
        layer->initStates(w, h, flags);
        addLayer_l(layer); //加到 SF 中，进行管理。
    }
    return layer;
}
```

[--->SurfaceFlinger::addLayer_l()]

```
status_t SurfaceFlinger::addLayer_l(const sp<LayerBase>& layer)
{
    // mCurrentState 是
    ssize_t i = mCurrentState.layersSortedByZ.add(
                //
                layer, &LayerBase::compareCurrentStateZ);
    sp<LayerBaseClient> lbc = LayerBase::dynamicCast< LayerBaseClient* >(layer.get());
    if (lbc != 0) {
        mLayerMap.add(lbc->serverIndex(), lbc);
    }
    return NO_ERROR;
}
mCurrentState 也是 SF 中一个比较有意思的数据类型。
struct State {
        State() {
            orientation = ISurfaceComposer::eOrientationDefault;
            freezeDisplay = 0;
        }
        LayerVector     layersSortedByZ; //按 Z 序排序的 LayerBase
        uint8_t         orientation;
        uint8_t         orientationType;
        uint8_t         freezeDisplay;
    };
```

SF 中有两个 state，一个是 mCurrentState，另外一个是 mDrawingState。一个 state 中有一个按 Z 序排列的 LayerBase 队列。这个应该好理解，SF 做混合的时候，会按 Z 顺序来混合显示内容。这样....在前边的图画就不会被后边的字给挡住了。

先回到 createSurface。

[---->SurfaceFlinger::createSurface]

```
sp<ISurface> SurfaceFlinger::createSurface(ClientID clientId, int pid,
        const String8& name, ISurfaceFlingerClient::surface_data_t* params,
        DisplayID d, uint32_t w, uint32_t h, PixelFormat format,
        uint32_t flags)
{
       switch (flags & eFXSurfaceMask) {
        case eFXSurfaceNormal:
            if (UNLIKELY(flags & ePushBuffers)) {
                layer = createPushBuffersSurfaceLocked(client, d, id,
                        w, h, flags);
            } else {
                layer = createNormalSurfaceLocked(client, d, id,
                        w, h, flags, format);
            }
            break;
        case eFXSurfaceBlur:
            layer = createBlurSurfaceLocked(client, d, id, w, h, flags);
            break;
        case eFXSurfaceDim:
            layer = createDimSurfaceLocked(client, d, id, w, h, flags);
            break;
    }

    if (layer != 0) {
        layer->setName(name);
        setTransactionFlags(eTransactionNeeded);
     //这种大量的封装导致我很痛苦—有什么好办法来解析这种代码吗?
     //layer 实际是 Layer 类，它返回的是 SurfaceLayer
        surfaceHandle = layer->getSurface();
        if (surfaceHandle != 0) {
            params->token = surfaceHandle->getToken();
            params->identity = surfaceHandle->getIdentity();
```

```
            params->width = w;
            params->height = h;
            params->format = format;
        }
    }

    return surfaceHandle;  //对于我们这种情况，它返回的 SurfaceLayer
}
```

好，回到我们本节最开始的地方。

```
sp<SurfaceControl> SurfaceComposerClient::createSurface(
        int pid,
        const String8& name,
        DisplayID display,
        uint32_t w,
        uint32_t h,
        PixelFormat format,
        uint32_t flags)
{
   //调用 BClient 的 createSurface
        sp<ISurface> surface = mClient->createSurface(&data, pid, name,
                display, w, h, format, flags);
        if (surface != 0) {
            if (uint32_t(data.token) < NUM_LAYERS_MAX) {
             //根据返回的 ISurface 接口，构造 SurfaceControl 对象
                result = new SurfaceControl(this, surface, data, w, h, format, flags);
            }
        }
    }
    return result;
}
```

mClient->createSurface 调用后返回一个 ISurface 对象，但是 SurfaceComposerClient 基于 ISurface 对象又构造了一个 SurfaceControl 对象。我个人觉得这个对象只不过提供了一些方便函数罢了。而且，在 Activity 那端，通过 readFormParcel 函数后实际 SurfaceControl 对象用不上，真正，真正保存在 java Surface 对象的还是 C++的 Surface 对象。待会我们可以看到。

回到 Surface_init。

```
static void Surface_init(
        JNIEnv* env, jobject clazz,
```

```
        jobject session,
        jint pid, jstring jname, jint dpy, jint w, jint h, jint format, jint flags)
{
    SurfaceComposerClient* client =
            (SurfaceComposerClient*)env->GetIntField(session, sso.client);

    sp<SurfaceControl> surface;
   surface = client->createSurface(pid, dpy, w, h, format, flags);
   setSurfaceControl(env, clazz, surface); //设置这个 SurfaceControl 到 java 对象中去。
```

//我头真得很晕了，有什么好东西可以生成这些类的关系图吗？？

3.3 copyFrom

copyFrom 的 native 实现是 Surface_copyFrom。

```
static void Surface_copyFrom(
        JNIEnv* env, jobject clazz, jobject other)
{
    const sp<SurfaceControl>& surface = getSurfaceControl(env, clazz);
    const sp<SurfaceControl>& rhs = getSurfaceControl(env, other);
    if (!SurfaceControl::isSameSurface(surface, rhs)) {
       setSurfaceControl(env, clazz, rhs); //就是把 other 对象的 surfaceControl 转移到
    //rhs 对象中
    }
}
```

3.5 writeToParcel 和 readFromParcel

writeToParcel 比较简单，就是把一些信息写到 Parcel 中去。我们主要看看 readFromParcel。
[---->Surface_readFromParcel()]

```
static void Surface_readFromParcel(
        JNIEnv* env, jobject clazz, jobject argParcel)
{
    Parcel* parcel = (Parcel*)env->GetIntField( argParcel, no.native_parcel);
    const sp<Surface>& control(getSurface(env, clazz));
    //根据服务端的 parcel 信息，来构造客户端的 Surface
    sp<Surface> rhs = new Surface(*parcel);
    if (!Surface::isSameSurface(control, rhs)) {
          setSurface(env, clazz, rhs);
    }
}
```

虽然在 WindowManagerService 那端存在 SurfaceControl 这个对象，但是到了 Activity 这端，就仅存在 Surface 对象了。我们看看 C++这个 Surface 到底以 Parcel 构造了什么东西。

```
Surface::Surface(const Parcel& parcel)
    :  mBufferMapper(GraphicBufferMapper::get()), mSharedBufferClient(NULL)
{
    sp<IBinder> clientBinder = parcel.readStrongBinder();
     //非常佩服 Android 的名字，mSurface 类型是 ISurface，实际是 Proxy 端
    //但是这里的类名也叫 Surface，能不让人头晕吗?
    mSurface    = interface_cast<ISurface>(parcel.readStrongBinder());
    mToken      = parcel.readInt32();
    mIdentity   = parcel.readInt32();
    mWidth      = parcel.readInt32();
    mHeight     = parcel.readInt32();
    mFormat     = parcel.readInt32();
    mFlags      = parcel.readInt32();

    // FIXME: what does that mean if clientBinder is NULL here?
    if (clientBinder != NULL) {
    //Activity 这端有一个 SFC 对象，但是没有 SurfaceControl 了。
        mClient = SurfaceComposerClient::clientForConnection(clientBinder);
    //哈哈，这个是和 SharedBufferServer 对应的东西
     //基于 SharedClient 跨进程共享对象而来。
        mSharedBufferClient = new SharedBufferClient(
                mClient->mControl, mToken, 2, mIdentity);
    }
    init();
}
```

3.6 小总结

磕磕碰碰，终于是把简化流程说了一遍，我们无不被 Google 无聊的命名而愤怒。但是我们是鱼肉，没办法了。咱们这里把简化流程中间的步骤和产出物总结下。

- SurfaceSession init 后会创建一个 SurfaceComposerClient，这个玩意会在 SF 中注册一个 Client 对象。当然秉着老规矩，Google 会搞一个跨进程的 BClient 来封装这个 Client 对象。我们可以不用考虑 BClient 了。Client 中呢，会构造一个跨进程的数据结构加 SharedClient。SFC 中有一个 SharedBufferClient 对象，它是以 SharedClient 为核心创建的。
- 接着在 AMS 中会以 SFC 来创建构造一个 Surface java 对象。在此过程中，会在 SF 中创建一个 Layer，这个 Layer 是从 LayerBaseClient 对象中派生而来，当然上面还有一个祖父是 LayerBase。LayerXXX 是 SF 中重要的数据结构。SF 的所有工

作都是围绕 Layer 来做的。另外，LayerXXX 会以 Client 的 SharedClient 来构造一个 SharedBufferServer。

- LayerXXX 太复杂，我们不想让他是一个跨进程的东西，所以每个 LayerBase 中还会有一个 ISurface 对象。对于 Layer 来说，这个 ISurface 对象是 SurfaceLayer 类型。
- SFC 中得到这个 ISurface 对象后，会以它为核心创建一个 SurfaceControl 对象。
- writeToParcel 会把整个 SurfaceControl 对象的信息写到 Parcel 中，比如 Isurface 对象，Client 对象和其他一些信息等。

- 然后我们会由 AIDL 生成的文件，Activity 进程会通过 readFromParcel 来创建一个 C++的 Surface 对象，这个 Surface 和 ISurface 没什么关系。Surface 读取 Parcel，得到 ISurface 对象，同时会根据 Client 信息创建一个 SFC 对象。因为 Client 对象包含一个跨进程的 SharedClient，所以这里新创建的 SFC 对象，会以这个 SharedClient 对象来创建一个 SharedBufferClient。

到此，我们就建立了 Activity 进程和 SF 的直接关系！

3.7 LockCanvas 和 UnlockCanvasandPost

到此，A 和 SF 进程有了直接的关系，体现在哪呢？

- A 的 Surface 对象中有 SF 的 client 信息，有 Layer 的 Surface 信息。

这样，A 就进入了画图的过程，这个过程的前后标志就是 Lock 和 Unlock 调用。

我们先看看 LockCanvas 干嘛了。

1 LockCanvas

实际对应的函数是

[---->Surface_lockCanvas()]

```
static jobject Surface_lockCanvas(JNIEnv* env, jobject clazz, jobject dirtyRect)
{
   //getSurface,从 JAVA 对象中取出对应的 C++ Surface 对象
  //此 Surface 对象不是 SF 中 LayerBaseClient 定义的 ISurface 对象
    const sp<Surface>& surface(getSurface(env, clazz));
    // dirtyRect 为 Region 类型，义如其名，表示一块需要重绘的区域、
    //从它的成员变量来说，区域应该是一块矩形。不知道圆形用区域如何表示,呵呵
    Region dirtyRegion;
    if (dirtyRect) {
        Rect dirty;
        dirty.left  = env->GetIntField(dirtyRect, ro.l);
        dirty.top   = env->GetIntField(dirtyRect, ro.t);
        dirty.right = env->GetIntField(dirtyRect, ro.r);
        dirty.bottom= env->GetIntField(dirtyRect, ro.b);
        if (!dirty.isEmpty()) {
            dirtyRegion.set(dirty);
        }
    } else {
```

```
        dirtyRegion.set(Rect(0x3FFF,0x3FFF));
    }


   //SurfaceInfo 结构，包含一个名为 bits 的变量，这个对应的值就是内存地址
  //
    Surface::SurfaceInfo info;
    status_t err = surface->lock(&info, &dirtyRegion);根据 dirtyRegion 区域，来 lock
```

[---->Surface::lock()]

```
status_t Surface::lock(SurfaceInfo* other, Region* dirtyIn, bool blocking)
{


    if (mApiLock.tryLock() != NO_ERROR) {
       }


    //mLockedBuffer 如果不为空，表示这块 buffer 还没有被释放，所以不能
    //lock 新的 buffer
    if (mLockedBuffer != 0) {
        LOGE("Surface::lock failed, already locked");
        mApiLock.unlock();
        return INVALID_OPERATION;
    }


   setUsage(GRALLOC_USAGE_SW_READ_OFTEN | GRALLOC_USAGE_SW_WRITE_OFTEN);


    sp<GraphicBuffer> backBuffer;
    status_t err = dequeueBuffer(&backBuffer);
```

[---->Surface::dequeueBuffer(sp<GraphicBuffer>* buffer)]

```
status_t Surface::dequeueBuffer(sp<GraphicBuffer>* buffer) {
    android_native_buffer_t* out;
    status_t err = dequeueBuffer(&out);
    if (err == NO_ERROR) {
        *buffer = GraphicBuffer::getSelf(out);
    }
    return err;
}
```

其中调用了

```
int Surface::dequeueBuffer(android_native_buffer_t** buffer)
{
    sp<SurfaceComposerClient> client(getClient());
    //使用 mSharedBufferClient，估计内部会使用跨进程的那个 SharedClient
    //就两个 buffer，这里获得 backbuffer 的 index，其实就是空闲 Buffer 的 index
    ssize_t bufIdx = mSharedBufferClient->dequeue();
    const uint32_t usage(getUsage());
    const sp<GraphicBuffer>& backBuffer(mBuffers[bufIdx]);
    //mBuffers 是 GraphicBuffer 的数组，个数为 2
    // backBuffer 为空，或者用法不一样的时候，都需要获取新的 buffer
    if (backBuffer == 0 ||
        ((uint32_t(backBuffer->usage) & usage) != usage) ||
        mSharedBufferClient->needNewBuffer(bufIdx))
    {
        err = getBufferLocked(bufIdx, usage);
        if (err == NO_ERROR) {
            // reset the width/height with the what we get from the buffer
            mWidth  = uint32_t(backBuffer->width);
            mHeight = uint32_t(backBuffer->height);
        }
    }
```

[--->Surface::getBufferLocked]

```
tatus_t Surface::getBufferLocked(int index, int usage)
{
    sp<ISurface> s(mSurface);
    status_t err = NO_MEMORY;
    // free the current buffer
    //currentBuffer 是一个引用
    sp<GraphicBuffer>& currentBuffer(mBuffers[index]);
    //终于用上了 ISurface 对象，这里的 ISurface 的对端是 SurfaceLayer
    sp<GraphicBuffer> buffer = s->requestBuffer(index, usage);
     if (buffer != 0) { // this should never happen by construction
         if (err == NO_ERROR) {
//ISurface request 得到的 Buffer 被赋值给了 currentBuffer
                currentBuffer = buffer;
                currentBuffer->setIndex(index);
```

```
                mNeedFullUpdate = true;
            }
     return err;
}
```

我们待会再去 ISurface 对端看看 requestBuffer。

我们回到

[--->Surface::dequeueBuffer(android_native_buffer_t** buffer)]

```
   ...到这里，通过 getBufferLocked，我们实际已经得到了 backBuffer
    if (err == NO_ERROR) {
        mDirtyRegion.set(backBuffer->width, backBuffer->height);
        *buffer = backBuffer.get(); //得到 sp 指向的指针
    }
    return err;
}
```

[---->Surface::dequeueBuffer(sp<GraphicBuffer>* buffer)]

```
status_t Surface::dequeueBuffer(sp<GraphicBuffer>* buffer) {
    android_native_buffer_t* out;
    status_t err = dequeueBuffer(&out);
    if (err == NO_ERROR) {
        *buffer = GraphicBuffer::getSelf(out); //好了，backBuffer 准备好了
    }
    return err;
}
```

[--->status_t Surface::lock]

```
    if (err == NO_ERROR) {
    //刚才是获取 Buffer，这里需要 lockBuffer
      err = lockBuffer(backBuffer.get());
```

[]

```
int Surface::lockBuffer(android_native_buffer_t* buffer)
{
    sp<SurfaceComposerClient> client(getClient());
    status_t err = validate();
    int32_t bufIdx = GraphicBuffer::getSelf(buffer)->getIndex();
    err = mSharedBufferClient->lock(bufIdx); //调用 SharedBufferClient 的 lock
     return err;
```

```
}
```

[---->SharedBufferClient::lock()]

```
status_t SharedBufferClient::lock(int buf)
{
    LockCondition condition(this, buf);
    status_t err = waitForCondition(condition);
    return err;
}
```

SharedBufferClient 的 lock 函数看起来很简单，但是用到了一个叫做函数对象的技巧。

我们先看看 waitForCondition 这个函数，注意它的参数是 LockCondition 对象。

[--->SharedBufferBase::waitForCondition()]

```
status_t SharedBufferBase::waitForCondition(T condition)
{
    const SharedBufferStack& stack( *mSharedStack );
    SharedClient& client( *mSharedClient );
    const nsecs_t TIMEOUT = s2ns(1);
    Mutex::Autolock _l(client.lock);
    while ((condition()==false) && //注意这个 condition()的用法
            (stack.identity == mIdentity) &&
            (stack.status == NO_ERROR))
    {
        status_t err = client.cv.waitRelative(client.lock, TIMEOUT);

        // handle errors and timeouts
        if (CC_UNLIKELY(err != NO_ERROR)) {
            if (err == TIMED_OUT) {
                if (condition()) {----->注意这个, condition()
                   break;
                } else {
               }
            } else {
               return err;
            }
        }
    }
```

```
    return (stack.identity != mIdentity) ? status_t(BAD_INDEX) : stack.status;
}
```

我们看到 waitForCondition 中有等待，这个好理解。那么等待的条件呢？没看到，不过看到了 conditon()这样的东西。我刚才说了，condition 是一个对象，对象()是什么呢？这就牵出了函数对象的概念。

函数对象，其实还是一个对象，不过重载了操作符()，这和重载操作符+,-等没什么区别。你可以把它当做是一个函数指针看待。或者你不重载()，而弄一个 public 函数叫 do()也行。

那为什么需要函数对象呢？因为对象可以保存信息嘛，而函数只是操作方面的。调用对象的函数就可以利用和更新对象内部的信息。

按上面的说法，其实重要的是 condition 对象的()函数了。

刚才传进来的是 LockCondition，它的()定义如下：

```
bool SharedBufferClient::LockCondition::operator()() {
    return (buf != stack.head ||
            (stack.queued > 0 && stack.inUse != buf));
} //就是根据读写位置判断是否这个 buffer 空闲了。buf 和 stack 都是 LockCondition 的成员
```

SF 中，尤其是 SharedBufferClient 中会大量出现这种用法。熟悉就好了，没什么特别之处。

[--->status_t Surface::lock]

好了，到此为止，假设我们先获取的那个 buffer 现在变成空闲了。

```
  if (err == NO_ERROR) {
            const Rect bounds(backBuffer->width, backBuffer->height);
            Region scratch(bounds);
            Region& newDirtyRegion(dirtyIn ? *dirtyIn : scratch);

            const sp<GraphicBuffer>& frontBuffer(mPostedBuffer);
  ...
            if (frontBuffer !=0 &&
                backBuffer->width  == frontBuffer->width &&
                backBuffer->height == frontBuffer->height &&
                !(mFlags & ISurfaceComposer::eDestroyBackbuffer))
            {
                const Region copyback(mOldDirtyRegion.subtract(newDirtyRegion));
                if (!copyback.isEmpty() && frontBuffer!=0) {
                  copyBlt(backBuffer, frontBuffer, copyback);
                }
            }
```

上面这段代码由有什么意思呢?为什么会出现 mPostedBuffer 的相关操作呢？

还记得 NUM_BUFFS 变量吗，那个值是 2.而且我们在 Layer 中确实也 new 了两个 GraphicBuffer。它们分别被称为 FrontBuffer 和 BackBuffer。这些东西的存在和 SF 混合的机制有

关系。大体的工作流程是这样的：

- 客户端在 BackBuffer 上画图，此时 SF 可能在使用 FrontBuffer 进行混合
- 客户端画完后，BackBuffer 变成新的 FrontBuffer，SF 使用完的 FrontBuffer 变成 BackBuffer。然后客户端和 SF 又可以使用这些 Buffer 进行工作了。这个转变的过程也叫 Flip，意为翻转。这个很形象！
- 这样周而复始，整个流程相比于只有一个 Buffer 的设计来说就显得很流畅。

当我们使用新的 BackBuffer 的时候，由于内存是整块更新的（按长*宽来计算），而实际可能我们只重绘了其中的一小块地方，这就需要在以前的信息和新的信息进行叠加以生成新的图像。而之前的信息保存在 mPostedBuffer 中。所以每次客户端在 unLockAndPost 中，会保存一下 BackBuffer 到 mPostedBuffer 变量中，然后下一次 Lock 的时候会比较上次 postedBuffer 和这次准备的 BackBuffer，看看是否需要上次 buffer 中的信息。

[---->Surface::lock()]

```
            mDirtyRegion = newDirtyRegion;
            mOldDirtyRegion = newDirtyRegion;

            void* vaddr;
            //GraphicBuffer 的 lock，lock 得到的地址在 vaddr 中保存。
            status_t res = backBuffer->lock(
                    GRALLOC_USAGE_SW_READ_OFTEN | GRALLOC_USAGE_SW_WRITE_OFTEN,
                    newDirtyRegion.bounds(), &vaddr);

            //other 就是 Surface_info 类型
            mLockedBuffer = backBuffer;
            other->w      = backBuffer->width;
            other->h      = backBuffer->height;
            other->s      = backBuffer->stride;
            other->usage  = backBuffer->usage;
            other->format = backBuffer->format;
            other->bits   = vaddr;  //把地址赋给 bits 变量
        }
    }
    mApiLock.unlock();
    return err;
}
```

OK，我们回到 JNI。

[--->Surface_lockCanvas()]

```
..到这里，我们已经有了内存了
```

```
// canvas是Surface java对象中的CompatiableCanvas对象
jobject canvas = env->GetObjectField(clazz, so.canvas);
env->SetIntField(canvas, co.surfaceFormat, info.format);

SkCanvas* nativeCanvas = (SkCanvas*)env->GetIntField(canvas, no.native_canvas);
SkBitmap bitmap;
ssize_t bpr = info.s * bytesPerPixel(info.format);
//把内存地址，宽，高等等信息设置到SKBitmap中
bitmap.setConfig(convertPixelFormat(info.format), info.w, info.h, bpr);
if (info.format == PIXEL_FORMAT_RGBX_8888) {
    bitmap.setIsOpaque(true);
}
if (info.w > 0 && info.h > 0) {
    bitmap.setPixels(info.bits);
}

nativeCanvas->setBitmapDevice(bitmap);

SkRegion clipReg;
if (dirtyRegion.isRect()) { // very common case
    const Rect& b(dirtyRegion.getBounds());
    clipReg.setRect(b.left, b.top, b.right, b.bottom);
} else {
    size_t count;
    Rect const* r = dirtyRegion.getArray(&count);
    while (count) {
        clipReg.op(r->left, r->top, r->right, r->bottom, SkRegion::kUnion_Op);
        r++, count--;
    }
}

nativeCanvas->clipRegion(clipReg);

int saveCount = nativeCanvas->save();
env->SetIntField(clazz, so.saveCount, saveCount);
```

```
    if (dirtyRect) {
        const Rect& bounds(dirtyRegion.getBounds());
        env->SetIntField(dirtyRect, ro.l, bounds.left);
        env->SetIntField(dirtyRect, ro.t, bounds.top);
        env->SetIntField(dirtyRect, ro.r, bounds.right);
        env->SetIntField(dirtyRect, ro.b, bounds.bottom);
    }

//此时的 canvas 已经有了对应的内存地址等重要信息
    return canvas;
}
```

从上面的函数可以看到。实际上 canvas 不过是封装了一块内存，JAVA 层在上面作图。而后我们在 Flip 下，把信息传到 SF，最后由 SF 混合这些东西输出到屏幕中去。

OK，让我们进入最后一步，那就是 unlockCanvasAndPost 了

```
static void Surface_unlockCanvasAndPost(
        JNIEnv* env, jobject clazz, jobject argCanvas)
{
    jobject canvas = env->GetObjectField(clazz, so.canvas);

    const sp<Surface>& surface(getSurface(env, clazz));
     SkCanvas* nativeCanvas = (SkCanvas*)env->GetIntField(canvas, no.native_canvas);
    int saveCount = env->GetIntField(clazz, so.saveCount);
    nativeCanvas->restoreToCount(saveCount);
    nativeCanvas->setBitmapDevice(SkBitmap());
    env->SetIntField(clazz, so.saveCount, 0);

    // unlock surface
    status_t err = surface->unlockAndPost();
    if (err < 0) {
        doThrow(env, "java/lang/IllegalArgumentException", NULL);
    }
}
```

调用 Surface C++对象的 unlockAndPost,

```
status_t Surface::unlockAndPost()
{
    if (mLockedBuffer == 0) {
```

```
        LOGE("Surface::unlockAndPost failed, no locked buffer");
        return INVALID_OPERATION;
    }

    status_t err = mLockedBuffer->unlock();
    err = queueBuffer(mLockedBuffer.get());
    mPostedBuffer = mLockedBuffer; //保存上次的 backBuffer
    mLockedBuffer = 0;
    return err;
}
int Surface::queueBuffer(android_native_buffer_t* buffer)
{
    sp<SurfaceComposerClient> client(getClient());

   int32_t bufIdx = GraphicBuffer::getSelf(buffer)->getIndex();
    mSharedBufferClient->setDirtyRegion(bufIdx, mDirtyRegion);
    err = mSharedBufferClient->queue(bufIdx);
     if (err == NO_ERROR) {
     //这个 Client 是 SFC
        client->signalServer();
    }
    return err;
}
```

最重要的是 queue 函数。使用了函数对象。

[---->SharedBufferClient::queue]

```
status_t SharedBufferClient::queue(int buf)
{
    QueueUpdate update(this);//有兴趣你可以去看看它的()函数，就是把写位置更新了下。
    status_t err = updateCondition( update );
     SharedBufferStack& stack( *mSharedStack );
    const nsecs_t now = systemTime(SYSTEM_TIME_THREAD);
    stack.stats.totalTime = ns2us(now - mDequeueTime[buf]);
    return err;
}
```

unlockAndPost 把写位置更新了下，然后通知 SF。我们下节在集中火力解决 SF 部分。

3.8 简化流程的总结

总体来说，SF 的流程相对简单，也不涉及到类似 Audio 上策略方面的问题。SF 难度主要是在它定义的那些形形色色的数据类型，以及随意而取的变量名，类型名。我个人觉得这块是 SF 的一个败笔。SF 显示这块是很重要，所以能通过我们这种分析代码就可以看出问题的几率是很小的。下一节我们集中在 SF 端，讲讲它的工作流程。