**从源码角度看WMS**

[RickAi](https://xiaozhuanlan.com/u/1253528474)+ 关注

预计阅读时间55分钟7 月前

**简介**

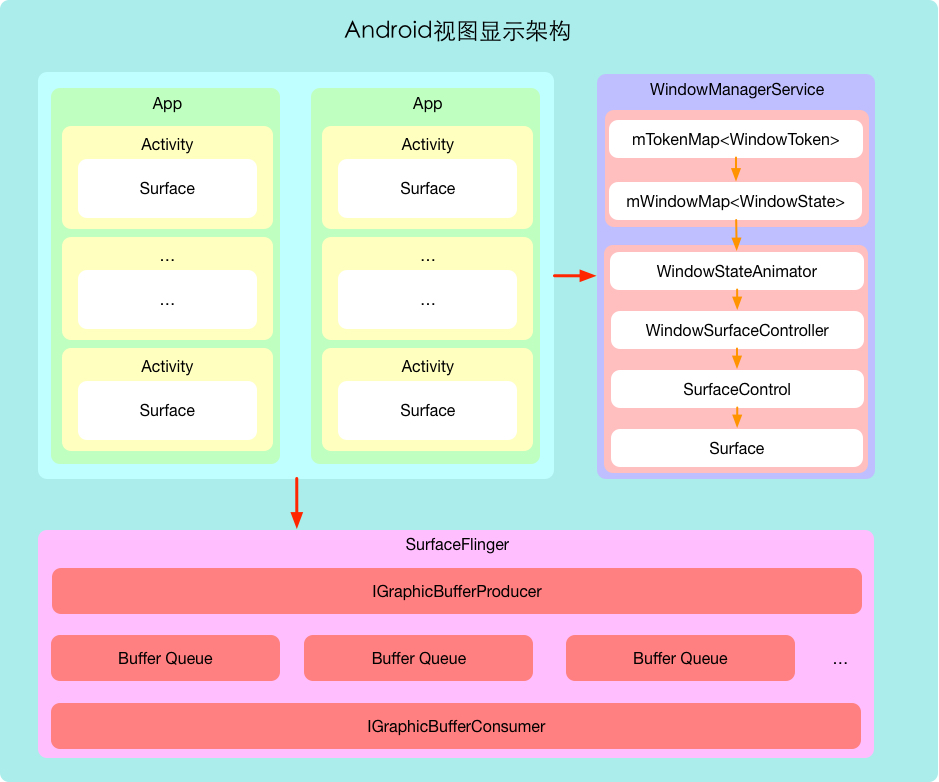
因为我以前接触APP层的视图绘制相关业务比较多，对View的显示相关源码相对来说更感兴趣，所以想写WMS相关的源码解析很久了，但是一直迟迟没有开始写。主要是因为WMS的确是属于Android系统中最复杂的组件之一，知识点涉及从Activity生命周期、Binder、JNI、Native再到SurfaceFlinger等方方面面，分析起来容易陷入无形的大网中迷失方向

之前我有分析过View视图的测量、布局、绘制与Activity的联系，也分析过ViewRootImpl在App客户端的地位，其实就是为了这篇做准备，现在感觉基础的知识点已经具备了。但是一篇博客的内容实在是太难覆盖WMS相关的知识点了, 所以我对其中的代码进行了一定的删减，避免陷入阅读源码细节的囧境

我的目的本身是为了理解WMS在视图绘制的角色，本文中尽量多画图，少列举代码，方便快速的理解App, WMS与SurfaceFlinger的交互模式与流程

**设计图**

**架构图**



视图绘制流程：

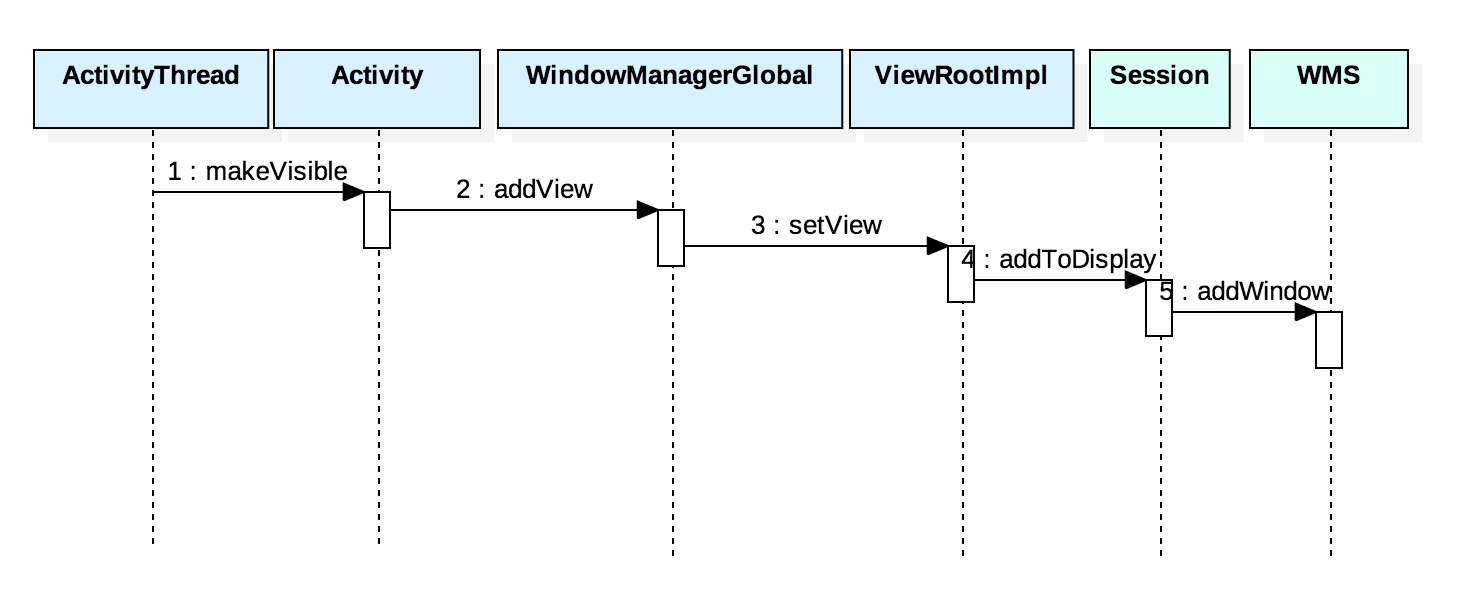
1. AMS binder call到应用，Activity到resume的生命周期
2. Activity调用WMSGlobal，委托ViewRootImpl进行addWindow操作
3. ViewRootImpl binder call到system\_server到Session服务
4. Session将请求委托给WMS，创建WindowToken, WindowState
5. ViewRootImpl首次调用performTraverals进行窗口刷新
6. ViewRootImpl调用到WMS，创建SurfaceControl
7. WMS binder call到SurfaceFlinger进行Layer, producer, consumer, bufferqueue的初始化
8. SurfaceControl返回Surface给App客户端
9. App客户端调用View.onDraw方法进行绘制
10. ViewRootImpl通过Surface的producer句柄binder call到SurfaceFlinger进行绘制

**类图**



* ViewRootImpl: 应用进程中视图相关管理类
* WindowManagerGlobal: 应用进程中WMS相关管理类
* Surface: 视图的封装，本质上是和SurfaceFlinger通信的"钥匙"
* WMS: 用于组织安卓系统中不同视图的层级、动画、大小等，并不参与绘制
* SurfaceControl: 用于创建、获取、设置Surface
* SurfaceFlinger: 安卓系统中真正进行绘制的进程服务
* BufferQueueProducer: 绘制生产者
* BufferQueueConsumer: 绘制消费者
* BufferQueue: 绘制队列
* Layer: 在SurfaceFlinger服务中的窗口表示类

**新建窗口: WMS.addWindow**



frameworks/base/core/java/android/app/ActivityThread.java

1. **final** **void** **handleResumeActivity**(IBinder token,
2. **boolean** clearHide, **boolean** isForward, **boolean** reallyResume, **int** seq, String reason) {
3. ...
4. **if** (r.activity.mVisibleFromClient) {
5. r.activity.makeVisible();
6. }
7. ...
8. }

关于窗口的新建与添加，我直接跳过了Activity的启动流程，直接从AMS调用scheduleResumeActivity到App端后讲起，当Activity可见时，回调用对应Activity的makeVisible方法进行可视化

frameworks/base/core/java/android/app/Activity.java

1. **void** **makeVisible**() {
2. *// 如果窗口之前并没有被添加，那么先获取WindowManager服务进行添加*
3. **if** (!mWindowAdded) {
4. ViewManager wm = getWindowManager();
5. wm.addView(mDecor, getWindow().getAttributes());
6. mWindowAdded = **true**;
7. }
8. mDecor.setVisibility(View.VISIBLE);
9. }

frameworks/base/core/java/android/view/WindowManagerGlobal.java

1. **public** **void** **addView**(View view, ViewGroup.LayoutParams params,
2. Display display, Window parentWindow) {
3. ...
4. *// App端视图相关的大管家是ViewRootImpl，一个窗口会对应一个，这里进行新建*
5. root = **new** ViewRootImpl(view.getContext(), display);
6. view.setLayoutParams(wparams);
7. *// 保存在列表中*
8. mViews.add(view);
9. mRoots.add(root);
10. mParams.add(wparams);
11. *// 这里进行视图设置相关操作*
12. **try** {
13. root.setView(view, wparams, panelParentView);
14. } **catch** (RuntimeException e) {
15. ...
16. }
17. }

和ViewRootImpl的控制器角色类似，WindowManagerGlobal是窗口相关的控制器，专门用来协调窗口相关的操作

1. **public** **void** **setView**(View view, WindowManager.LayoutParams attrs, View panelParentView) {
2. ...
3. res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
4. getHostVisibility(), mDisplay.getDisplayId(),
5. mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
6. mAttachInfo.mOutsets, mInputChannel);
7. ...
8. }

ViewRootImpl并没有和WindowManagerService直接进行通信，而是通过运行在system\_server的WindowSession服务进行了中转

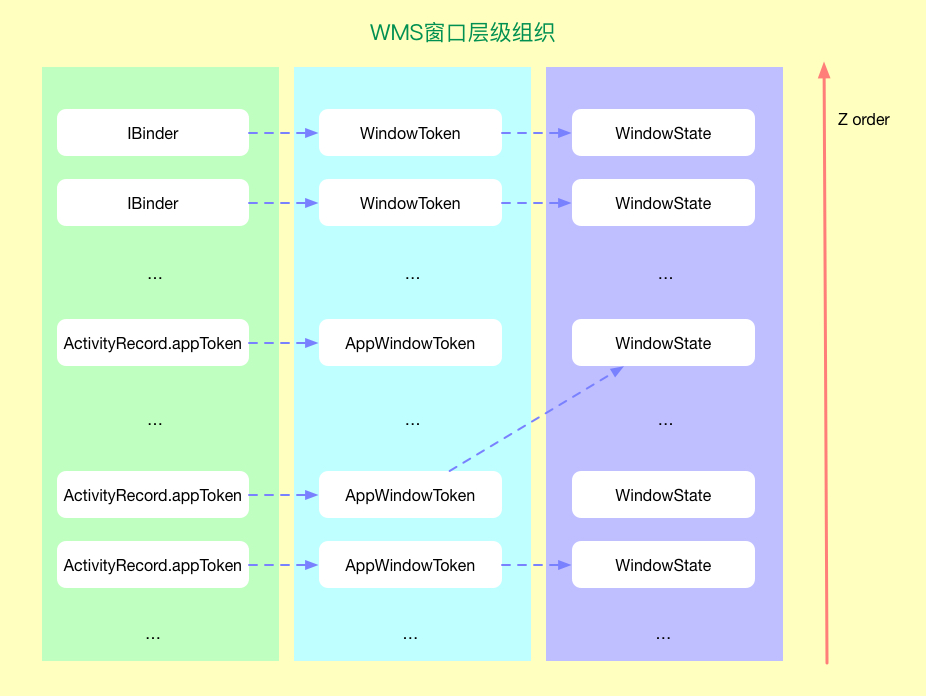
frameworks/base/services/core/java/com/android/server/wm/Session.java

1. **@Override**
2. **public** **int** **addToDisplay**(IWindow window, **int** seq, WindowManager.LayoutParams attrs,
3. **int** viewVisibility, **int** displayId, Rect outContentInsets, Rect outStableInsets,
4. Rect outOutsets, InputChannel outInputChannel) {
5. **return** mService.addWindow(**this**, window, seq, attrs, viewVisibility, displayId,
6. outContentInsets, outStableInsets, outOutsets, outInputChannel);
7. }

这里并没有进行多余的操作，直接中转到WMS进行处理

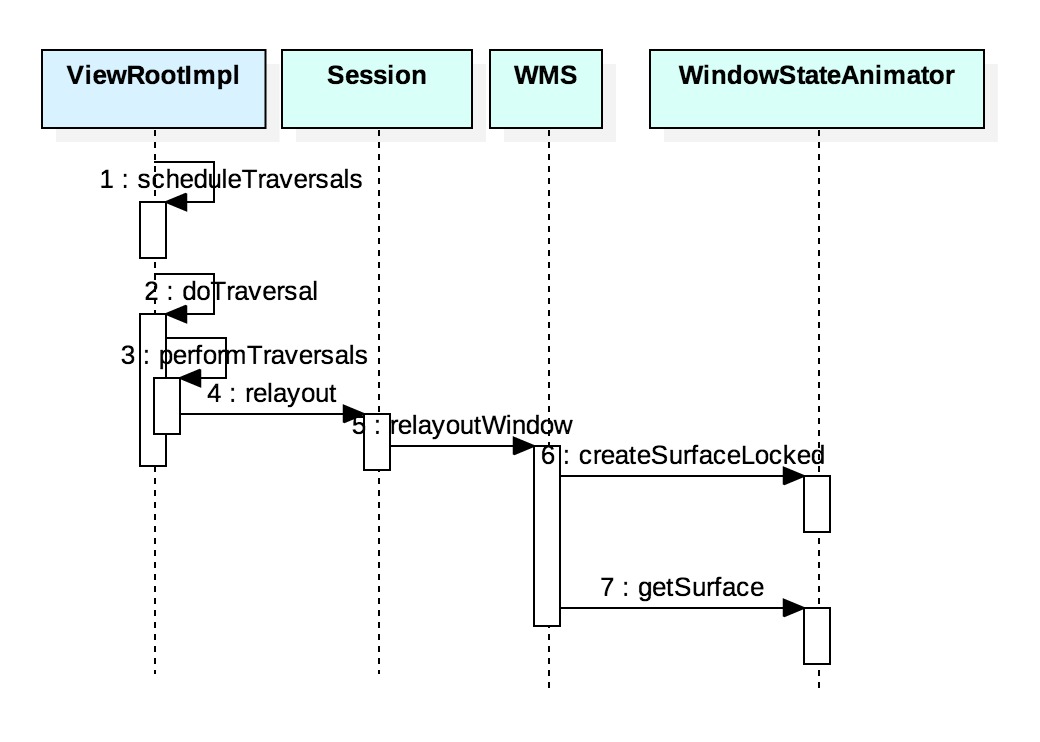
frameworks/base/services/core/java/com/android/server/wm/WindowManagerService.java

1. **public** **int** **addWindow**(Session session, IWindow client, **int** seq,
2. WindowManager.LayoutParams attrs, **int** viewVisibility, **int** displayId,
3. Rect outContentInsets, Rect outStableInsets, Rect outOutsets,
4. InputChannel outInputChannel) {
5. ...
6. *// 尝试获取token*
7. WindowToken token = mTokenMap.get(attrs.token);
8. **if** (token == **null**) {
9. *// 如果为空，那么会新建一个WindowToken*
10. token = **new** WindowToken(**this**, attrs.token, -1, **false**);
11. }
12. ...
13. *// 新创建一个WindowState*
14. WindowState win = **new** WindowState(**this**, session, client, token,
15. attachedWindow, appOp[0], seq, attrs, viewVisibility, displayContent);
16. ...
17. *// 进行保存*
18. mWindowMap.put(client.asBinder(), win);
19. ...
20. }



至此，创建视图的大概操作已经完成，addWindow的操作主要在于初始化WindowToken, WindowState这两个数据结构，WMS正是主要凭借着它们来组织视图的系统层级结构的

**刷新窗口: WMS.relayoutWindow**



frameworks/base/core/java/android/view/ViewRootImpl.java

1. **void** **scheduleTraversals**() {
2. **if** (!mTraversalScheduled) {
3. ...
4. mChoreographer.postCallback(
5. Choreographer.CALLBACK\_TRAVERSAL, mTraversalRunnable, **null**);
6. ...
7. }
8. }

窗口的刷新是由VSYNC进行控制的，主要是通过调用scheduleTraversals方法进行实现，这里可以看到，主要是通过mChoreographer发送了一个Message进行实现的

1. **final** **class** **TraversalRunnable** **implements** **Runnable** {
2. **@Override**
3. **public** **void** **run**() {
4. doTraversal();
5. }
6. }
7. **void** **doTraversal**() {
8. **if** (mTraversalScheduled) {
9. ...
10. performTraversals();
11. ...
12. }
13. }
14. **private** **void** **performTraversals**() {
15. ...
16. relayoutResult = relayoutWindow(params, viewVisibility, insetsPending);
17. ...
18. }
19. **private** **int** **relayoutWindow**(WindowManager.LayoutParams params, **int** viewVisibility, **boolean** insetsPending) **throws** RemoteException {
20. ...
21. **int** relayoutResult = mWindowSession.relayout(
22. mWindow, mSeq, params,
23. (**int**) (mView.getMeasuredWidth() \* appScale + 0.5f),
24. (**int**) (mView.getMeasuredHeight() \* appScale + 0.5f),
25. viewVisibility, insetsPending ? WindowManagerGlobal.RELAYOUT\_INSETS\_PENDING : 0,
26. mWinFrame, mPendingOverscanInsets, mPendingContentInsets, mPendingVisibleInsets,
27. mPendingStableInsets, mPendingOutsets, mPendingBackDropFrame, mPendingConfiguration,
28. mSurface);
29. ...
30. }

TraversalRunnable这个类中，主要是进行doTraversal的调用，最终同样是通过mWindowSession binder call到system\_server进行处理

frameworks/base/services/core/java/com/android/server/wm/Session.java

1. **public** **int** **relayout**(IWindow window, **int** seq...) {
2. **int** res = mService.relayoutWindow(**this**, window, seq, attrs,
3. requestedWidth, requestedHeight, viewFlags, flags,
4. outFrame, outOverscanInsets, outContentInsets, outVisibleInsets,
5. outStableInsets, outsets, outBackdropFrame, outConfig, outSurface);
6. **return** res;
7. }

Session的调用仅仅是起到了中转的作用

frameworks/base/services/core/java/com/android/server/wm/WindowManagerService.java

1. **public** **int** **relayoutWindow**(Session session, IWindow client...) {
2. ...
3. *// 首先获取到对应的WindowState*
4. WindowState win = windowForClientLocked(session, client, **false**);
5. ...
6. WindowStateAnimator winAnimator = win.mWinAnimator;
7. *// 随后创建Surface*
8. result = createSurfaceControl(outSurface, result, win, winAnimator);
9. }
10. **private** **int** **createSurfaceControl**(Surface outSurface, **int** result, WindowState win, WindowStateAnimator winAnimator) {
11. ...
12. *// 创建Surface*
13. WindowSurfaceController surfaceController = winAnimator.createSurfaceLocked();
14. **if** (surfaceController != **null**) {
15. *// 进行拷贝*
16. surfaceController.getSurface(outSurface);
17. } **else** {
18. *// For some reason there isn't a surface. Clear the*
19. *// caller's object so they see the same state.*
20. outSurface.release();
21. }
22. **return** result;
23. }

Surface的创建其实是层层委托，最终交给了SurfaceControl去创建Surface

frameworks/base/services/core/java/com/android/server/wm/WindowSurfaceController.java

1. **public** **WindowSurfaceController**(SurfaceSession s,
2. String name, **int** w, **int** h, **int** format, **int** flags, WindowStateAnimator animator) {
3. ...
4. **if** (animator.mWin.isChildWindow() &&
5. animator.mWin.mSubLayer < 0 &&
6. animator.mWin.mAppToken != **null**) {
7. ...
8. } **else** **if** (DEBUG\_SURFACE\_TRACE) {
9. ...
10. } **else** {
11. mSurfaceControl = **new** SurfaceControl(
12. s, name, w, h, format, flags);
13. }
14. }

frameworks/base/core/java/android/view/SurfaceControl.java

1. **public** **SurfaceControl**(SurfaceSession session,
2. String name, **int** w, **int** h, **int** format, **int** flags)
3. **throws** OutOfResourcesException {
4. ...
5. mNativeObject = nativeCreate(session, name, w, h, format, flags);
6. ...
7. }

在调用nativeCreate后，Surface的创建就完成了，之后通过SurfaceControl可以将它的成员Surface进行拷贝，随后会返回给App客户端

接下来深入到native中就看看Surface的初始化

frameworks/base/core/jni/android\_view\_SurfaceControl.cpp

1. **static** jlong **nativeCreate**(JNIEnv\* env, jclass clazz, jobject sessionObj,
2. jstring nameStr, jint w, jint h, jint format, jint flags) {
3. ScopedUtfChars **name**(env, nameStr);
4. sp<SurfaceComposerClient> client(android\_view\_SurfaceSession\_getClient(env, sessionObj));
5. *// 发送binder call到运行在surface flinger的进程SurfaceComposerClient服务进行创建*
6. sp<SurfaceControl> surface = client->createSurface(
7. String8(name.c\_str()), w, h, format, flags);
8. ...
9. surface->incStrong((**void** \*)nativeCreate);
10. **return** **reinterpret\_cast**<jlong>(surface.get());
11. }

frameworks/native/libs/gui/SurfaceComposerClient.cpp

1. sp<SurfaceControl> SurfaceComposerClient::createSurface(
2. **const** String8& name,
3. **uint32\_t** w,
4. **uint32\_t** h,
5. PixelFormat format,
6. **uint32\_t** flags)
7. {
8. sp<SurfaceControl> sur;
9. **if** (mStatus == NO\_ERROR) {
10. sp<IBinder> handle;
11. sp<IGraphicBufferProducer> gbp;
12. *// 委托给Client进行创建*
13. **status\_t** err = mClient->createSurface(name, w, h, format, flags,
14. &handle, &gbp);
15. ALOGE\_IF(err, "SurfaceComposerClient::createSurface error %s", strerror(-err));
16. **if** (err == NO\_ERROR) {
17. *// 将返回的binder句柄与producer引用封装成SurfaceControl返回*
18. sur = **new** SurfaceControl(**this**, handle, gbp);
19. }
20. }
21. **return** sur;
22. }

到这里，获取到了producer的引用其实Surface就已经创建成功了，我们直接看SurfaceControl是如何获取到Surface的：

frameworks/native/libs/gui/SurfaceControl.cpp

1. sp<Surface> SurfaceControl::getSurface() **const**
2. {
3. Mutex::Autolock \_l(mLock);
4. **if** (mSurfaceData == 0) {
5. mSurfaceData = **new** Surface(mGraphicBufferProducer, false);
6. }
7. **return** mSurfaceData;
8. }

可以看到，Surface的创建不过是将producer引用封装到Surface中，本质上客户端是通过producer与surface flinger进程的服务进行通信，真正的绘制相关的操作都是需要跨进程调用到该进程才能够完成

这里稍微深究一下Layer的创建

frameworks/native/services/surfaceflinger/Client.cpp

1. **status\_t** Client::createSurface(
2. **const** String8& name,
3. **uint32\_t** w, **uint32\_t** h, PixelFormat format, **uint32\_t** flags,
4. sp<IBinder>\* handle,
5. sp<IGraphicBufferProducer>\* gbp)
6. {
7. ...
8. result = flinger->createLayer(name, client, w, h, format, flags,
9. handle, gbp);
10. }

主要创建操作委托给SurfaceFlinger进行

frameworks/native/services/surfaceflinger/SurfaceFlinger.cpp

1. **status\_t** SurfaceFlinger::createLayer(
2. **const** String8& name,
3. **const** sp<Client>& client,
4. **uint32\_t** w, **uint32\_t** h, PixelFormat format, **uint32\_t** flags,
5. sp<IBinder>\* handle, sp<IGraphicBufferProducer>\* gbp)
6. {
7. ...
8. *// 可以看到handle与producer都是在该方法进行创建*
9. result = createNormalLayer(client,
10. name, w, h, flags, format,
11. handle, gbp, &layer);
12. }
13. **status\_t** SurfaceFlinger::createNormalLayer(**const** sp<Client>& client,
14. **const** String8& name, **uint32\_t** w, **uint32\_t** h, **uint32\_t** flags, PixelFormat& format,
15. sp<IBinder>\* handle, sp<IGraphicBufferProducer>\* gbp, sp<Layer>\* outLayer)
16. {
17. *// 进行初始化*
18. \*outLayer = **new** Layer(**this**, client, name, w, h, flags);
19. **status\_t** err = (\*outLayer)->setBuffers(w, h, format, flags);
20. **if** (err == NO\_ERROR) {
21. *// 获取引用*
22. \*handle = (\*outLayer)->getHandle();
23. \*gbp = (\*outLayer)->getProducer();
24. }
25. }

frameworks/native/services/surfaceflinger/Layer.cpp

1. **void** Layer::onFirstRef() {
2. *// Creates a custom BufferQueue for SurfaceFlingerConsumer to use*
3. sp<IGraphicBufferProducer> producer;
4. sp<IGraphicBufferConsumer> consumer;
5. *// 创建BufferQueue，同时创建对应的生产者与消费者，每个Layer单独拥有这一套模型*
6. BufferQueue::createBufferQueue(&producer, &consumer);
7. mProducer = **new** MonitoredProducer(producer, mFlinger);
8. mSurfaceFlingerConsumer = **new** SurfaceFlingerConsumer(consumer, mTextureName);
9. ...
10. }

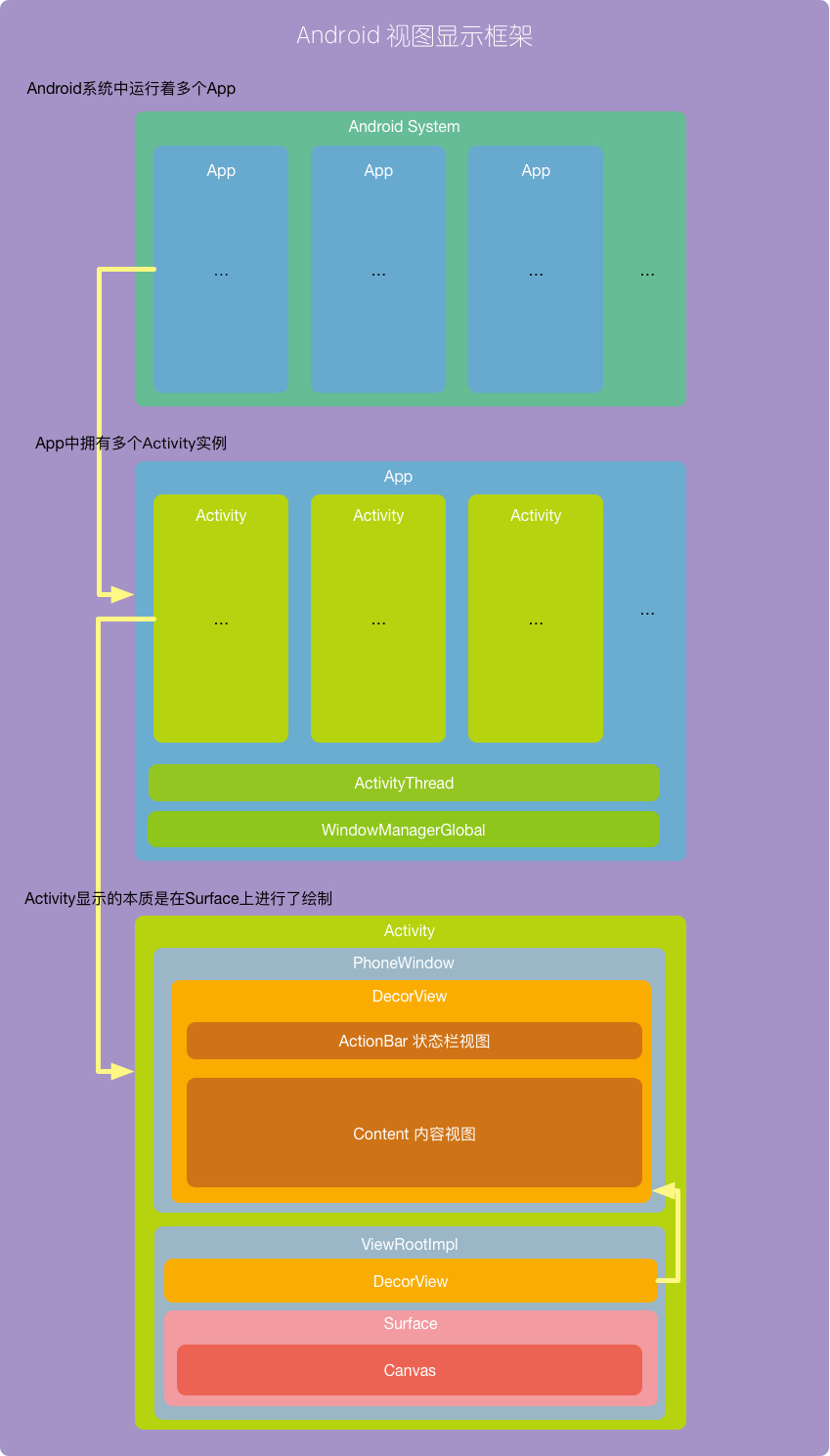
frameworks/native/libs/gui/BufferQueue.cpp

1. **void** BufferQueue::createBufferQueue(sp<IGraphicBufferProducer>\* outProducer,
2. sp<IGraphicBufferConsumer>\* outConsumer,
3. **const** sp<IGraphicBufferAlloc>& allocator) {
4. sp<BufferQueueCore> core(**new** BufferQueueCore(allocator));
5. sp<IGraphicBufferProducer> producer(**new** BufferQueueProducer(core));
6. sp<IGraphicBufferConsumer> consumer(**new** BufferQueueConsumer(core));
7. \*outProducer = producer;
8. \*outConsumer = consumer;
9. }

最终到BufferQueue，整个SurfaceControl的创建操作就完成了。可以看到，这个创建过程为之后的绘制刷新提供了操作环境。其中的生产者引用会通过层层的返回，最终通过WMS返回给App客户端，客户但就可以通过Surface直接与surface flinger进程的服务进行通信，完成视图刷新操作

**绘制窗口: ViewRootImpl.drawSoftware**

窗口绘制的本质，其实就在于将从Surface中获取到的Canvas进行绘制，如下图：



安卓系统中运行着多个应用，每个应用又可以运行着多个Activity，每个Activity其实本质上拥有着一个父View，即DecorView，它绘制的其实是从ViewRootImpl的Surface中拿到的Canvas

frameworks/base/core/java/android/view/ViewRootImpl.java

1. **private** **boolean** **drawSoftware**(Surface surface, AttachInfo attachInfo, **int** xoff, **int** yoff,
2. **boolean** scalingRequired, Rect dirty) {
3. ...
4. *// 获取一个Canvas*
5. canvas = mSurface.lockCanvas(dirty);
6. ...
7. *// 进行绘制*
8. mView.draw(canvas);
9. ...
10. *// 请求SurfaceFlinger进行绘制*
11. surface.unlockCanvasAndPost(canvas);
12. }

mView对象是控件树中的root节点，也就是所APP控件中常常获取到的Canvas是从ViewRootImpl中传递过来的

frameworks/base/core/jni/android\_view\_Surface.cpp

1. **static** jlong **nativeLockCanvas**(JNIEnv\* env, jclass clazz,
2. jlong nativeObject, jobject canvasObj, jobject dirtyRectObj) {
3. sp<Surface> surface(**reinterpret\_cast**<Surface \*>(nativeObject));
4. ...
5. ANativeWindow\_Buffer outBuffer;
6. **status\_t** err = surface->lock(&outBuffer, dirtyRectPtr);
7. ...
8. *// 之后APP会在该Canvas上进行2D图像的绘制*
9. Canvas\* nativeCanvas = GraphicsJNI::getNativeCanvas(env, canvasObj);
10. nativeCanvas->setBitmap(bitmap);
11. ...
12. sp<Surface> lockedSurface(surface);
13. lockedSurface->incStrong(&sRefBaseOwner);
14. **return** (jlong) lockedSurface.get();
15. }

当APP绘制完成后，ViewRootImpl继续调用unlockCanvasAndPost, 之后又会调用回native

frameworks/base/core/jni/android\_view\_Surface.cpp

1. **static** **void** **nativeUnlockCanvasAndPost**(JNIEnv\* env, jclass clazz,
2. jlong nativeObject, jobject canvasObj) {
3. sp<Surface> surface(**reinterpret\_cast**<Surface \*>(nativeObject));
4. ...
5. *// detach the canvas from the surface*
6. Canvas\* nativeCanvas = GraphicsJNI::getNativeCanvas(env, canvasObj);
7. nativeCanvas->setBitmap(SkBitmap());
8. *// unlock surface*
9. **status\_t** err = surface->unlockAndPost();
10. ...
11. }

frameworks/native/libs/gui/Surface.cpp

1. **status\_t** Surface::unlockAndPost()
2. {
3. ...
4. **int** fd = -1;
5. *// 将GraphicBuffer解锁*
6. **status\_t** err = mLockedBuffer->unlockAsync(&fd);
7. *// 将完成绘制的图像buffer入队*
8. err = queueBuffer(mLockedBuffer.get(), fd);
9. mPostedBuffer = mLockedBuffer;
10. mLockedBuffer = 0;
11. **return** err;
12. }
13. **int** Surface::queueBuffer(**android\_native\_buffer\_t**\* buffer, **int** fenceFd) {
14. ...
15. **int** i = getSlotFromBufferLocked(buffer);
16. ...
17. *// binder call到运行在surfaceflinger进程中的GraphicBufferProducer*
18. **status\_t** err = mGraphicBufferProducer->queueBuffer(i, input, &output);
19. ...
20. }

mGraphicBufferProducer的是在Surface初始化时被创建的，是一个binder句柄，通过它可以调用到surface flinger进程中进行图像buffer入列

frameworks/native/libs/gui/BufferQueueProducer.cpp

1. **status\_t** BufferQueueProducer::queueBuffer(**int** slot,
2. **const** QueueBufferInput &input, QueueBufferOutput \*output) {
3. ...
4. **if** (mCore->mQueue.empty()) {
5. *// 如果队列为空，直接将该项插入到队列中*
6. mCore->mQueue.push\_back(item);
7. frameAvailableListener = mCore->mConsumerListener;
8. } **else** {
9. *// 如果不为空，就需要检查队列中是否有子项需要被替代*
10. **const** BufferItem& last = mCore->mQueue.itemAt(
11. mCore->mQueue.size() - 1);
12. **if** (last.mIsDroppable) {
13. ...
14. *// 进行替换*
15. mCore->mQueue.editItemAt(mCore->mQueue.size() - 1) = item;
16. frameReplacedListener = mCore->mConsumerListener;
17. } **else** {
18. mCore->mQueue.push\_back(item);
19. frameAvailableListener = mCore->mConsumerListener;
20. }
21. }
22. ...
23. }

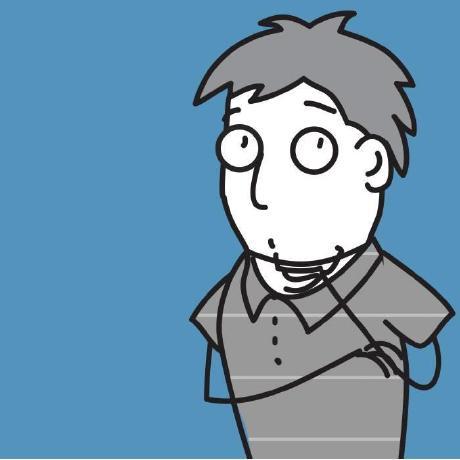
这个是典型的生产者消费者模式，应用绘制完成后，将会把经过绘制的buffer通过binder call的方式传给运行在surfaceflinger的producer，代替客户端进行消费队列的插入操作。插入完成后，图形缓存去就可以等待consumer来进行消费了。

frameworks/native/libs/gui/BufferQueueConsumer.cpp

1. **status\_t** BufferQueueConsumer::acquireBuffer(BufferItem\* outBuffer,
2. **nsecs\_t** expectedPresent, **uint64\_t** maxFrameNumber) {
3. ...
4. BufferQueueCore::Fifo::iterator **front**(mCore->mQueue.begin());
5. *// 对buffer进行处理*
6. ...
7. *// 处理完成后进行移除*
8. mCore->mQueue.erase(front);
9. }

© 著作权归作者所有

**从源码角度看 InputManagerService**

[](https://xiaozhuanlan.com/u/1253528474)

[RickAi](https://xiaozhuanlan.com/u/1253528474)+ 关注

预计阅读时间82分钟11 月前

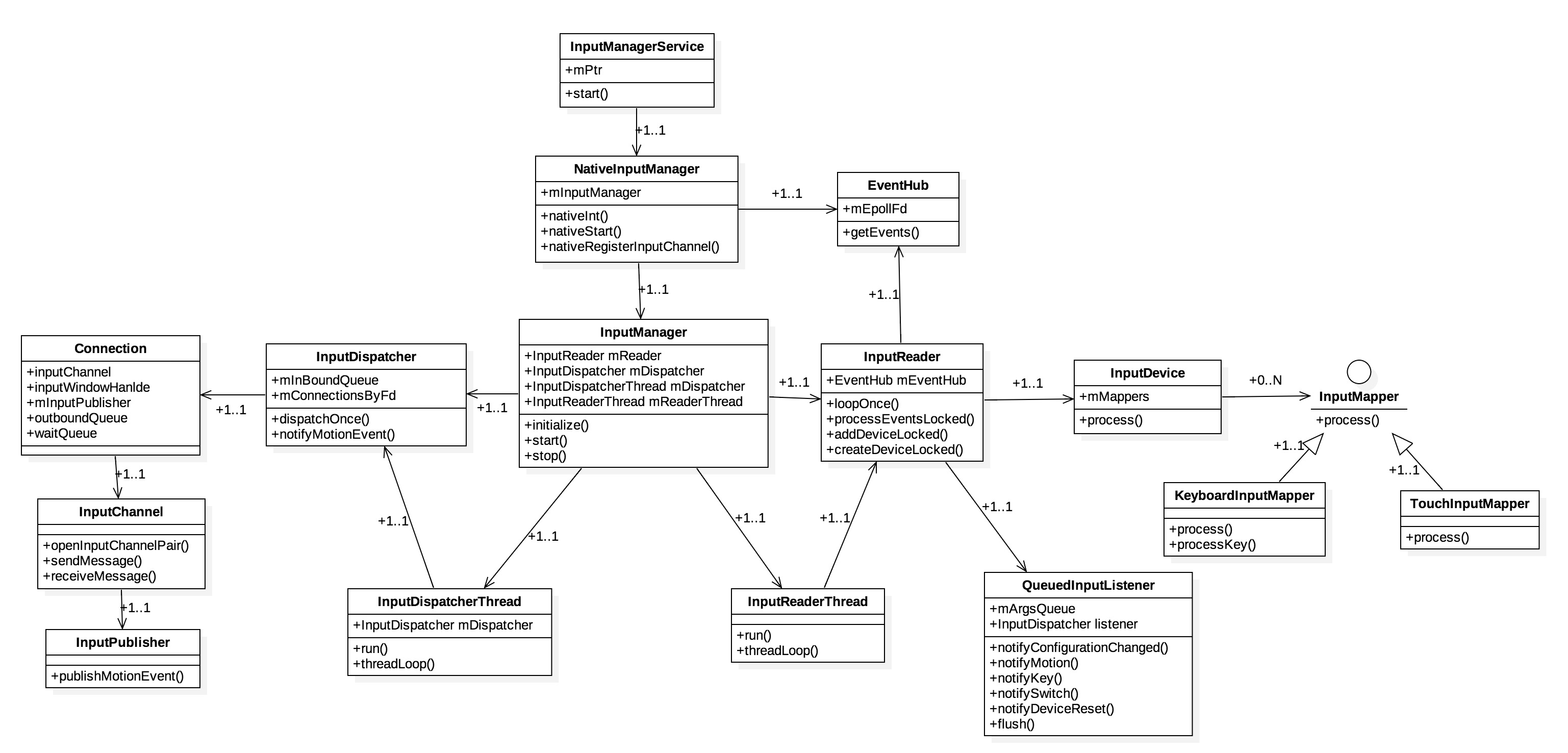
## 简介

上篇文章《从源码角度看Activity显示视图流程》中，我从Activity被成功创建开始分析，到 ViewRootImpl 进行控件树的绘制，最后再到 ViewRootImpl 与 WMS 的通信。这期间其实涉及到了 InputManagerService 的一些初始化工作，里面 InputChannel 初始化的代码我都没有继续深入分析

这篇文章里，我将系统的分析 IMS 的源码，照惯例先放出一张整体的类图与流程图，方便把握整体。最后分了三个小节分别去介绍其中的细节

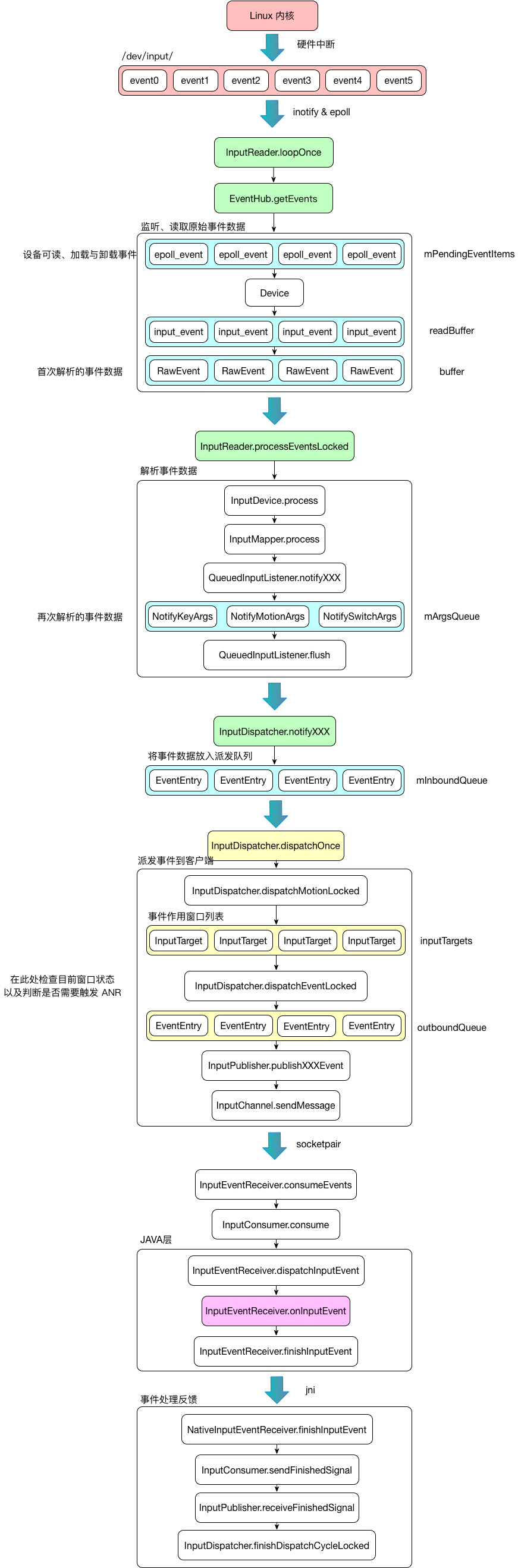
文章将会分析到 InputEventReceive.onInputEvent 方法，这个方法是客户端事件分发机制的入口，下片文章里我将从 onInputEvent 为入口开始分析

[查看大图](https://raw.githubusercontent.com/RickAi/Images/master/blog/IMS_class_diagram_overall.jpg)



## 整体流程图

[查看大图](https://raw.githubusercontent.com/RickAi/Images/master/blog/InputManagerService.jpg)



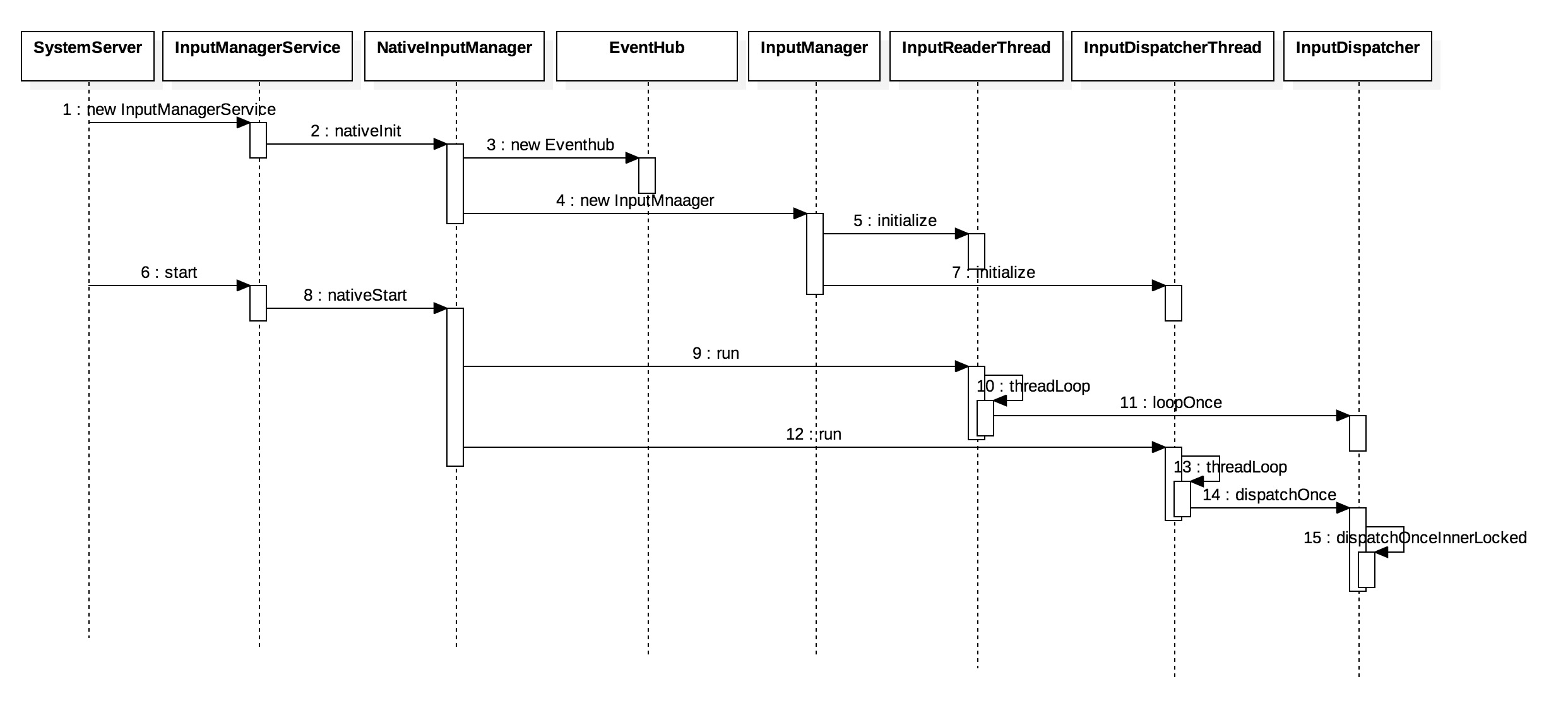
1. 用户轻点屏幕，linux 内核产生中断，向 /dev/input/ 目录下的设备文件 eventxx 下入数据
2. native 层，EventHub 通过 epoll 监测到文件被写入，使用 inotify 读取文件中的数据
3. InputReader 将事件数据解析成装满 RawEvent 的缓冲区中，随后批量使用 InputMapper 进行处理
4. 用户的触摸事件最终被加工为 NotifyMotionArgs，随后被批量插入到 InputDispatcher 的队列中
5. InputDispacher 从队列中取出 EventEntry 数据进行派发
6. 获取触摸事件目标窗口列表，使用 socketpair 向客户端发送输入事件
7. 客户端在将事件分发到各个窗口，处理完毕后会调用 finish 告诉服务端事件已经处理完成
8. InputDispatcher 收到事件处理完成通知，重新初始化 ANR 相关变量

## IMS 初始化与启动

InputManagerService 读取事件数据、解析并派发事件到客户端的逻辑都是在 native 实现的，java 层所做的工作不过是接收并反馈事件处理的结果

这一节对应着整体流程的初始化步骤，如epoll 与 inotify 对于输入事件设备文件的初始化，有了它们 android 才得以以高效的方式获取用户输入事件的数据；如InputReaderThread 和 InputDispatcherThread 两个重要线程，才使得整个 ims 服务的工作流程犹如工厂生产线一般前后贯通

[查看大图](https://raw.githubusercontent.com/RickAi/Images/master/blog/IMS_init_start.jpg)



### 初始化 epoll 与 inotify

frameworks/base/services/java/com/android/server/SystemServer.java

1. **private** **void** **startOtherServices**() {
2. ...
3. Slog.i(TAG, "Input Manager");
4. inputManager = **new** InputManagerService(context);
5. ServiceManager.addService(Context.INPUT\_SERVICE, inputManager);
6. inputManager.start();
7. ...
8. }

和WMS, AMS 这些重要系统服务一样，IMS也是在SystemServer.startOtherServices中进行初始化，并调用 start 方法启动

frameworks/base/services/core/java/com/android/server/input/InputManagerService.java

1. **public** **InputManagerService**(Context context) {
2. mPtr = nativeInit(**this**, mContext, mHandler.getLooper().getQueue());
3. }

InputManagerService 中，直接调用 nativeInit 在 native 中进行初始化

frameworks/base/services/core/jni/com\_android\_server\_input\_InputManagerService.cpp

1. **static** jlong **nativeInit**(JNIEnv\* env, jclass */\* clazz \*/*,
2. jobject serviceObj, jobject contextObj, jobject messageQueueObj) {
3. NativeInputManager\* im = **new** NativeInputManager(contextObj, serviceObj,
4. messageQueue->getLooper());
5. im->incStrong(0);
6. **return** **reinterpret\_cast**<jlong>(im);
7. }
8. NativeInputManager::NativeInputManager(jobject contextObj,
9. jobject serviceObj, **const** sp<Looper>& looper) :
10. mLooper(looper), mInteractive(true) {
11. ...
12. sp<EventHub> eventHub = **new** EventHub();
13. mInputManager = **new** InputManager(eventHub, **this**, **this**);
14. }

初始化 EventHub 与 InputManager，其中 EventHub 的本质要封装了 epoll 与 inotify，监听设备写入的事件并读取到缓冲中进行简单解析

frameworks/native/services/inputflinger/EventHub.cpp

1. EventHub::EventHub(**void**) :
2. mBuiltInKeyboardId(NO\_BUILT\_IN\_KEYBOARD), mNextDeviceId(1), mControllerNumbers(),
3. mOpeningDevices(0), mClosingDevices(0),
4. mNeedToSendFinishedDeviceScan(false),
5. mNeedToReopenDevices(false), mNeedToScanDevices(true),
6. mPendingEventCount(0), mPendingEventIndex(0), mPendingINotify(false) {
7. acquire\_wake\_lock(PARTIAL\_WAKE\_LOCK, WAKE\_LOCK\_ID);
8. *// 新建一个 epoll*
9. mEpollFd = epoll\_create(EPOLL\_SIZE\_HINT);
10. LOG\_ALWAYS\_FATAL\_IF(mEpollFd < 0, "Could not create epoll instance. errno=%d", errno);
11. *// 初始化 inotify*
12. mINotifyFd = inotify\_init();
13. *// 监听设备的创建与删除*
14. **int** result = inotify\_add\_watch(mINotifyFd, DEVICE\_PATH, IN\_DELETE | IN\_CREATE);
15. LOG\_ALWAYS\_FATAL\_IF(result < 0, "Could not register INotify for %s. errno=%d",
16. DEVICE\_PATH, errno);
17. **struct** **epoll\_event** **eventItem**;
18. memset(&eventItem, 0, **sizeof**(eventItem));
19. eventItem.events = EPOLLIN;
20. eventItem.data.u32 = EPOLL\_ID\_INOTIFY;
21. *// 增加需要监听的设备 fd*
22. result = epoll\_ctl(mEpollFd, EPOLL\_CTL\_ADD, mINotifyFd, &eventItem);
23. ...
24. }

epoll 初始化完成后，当输入设备被写入数据后，epoll\_wait 就会返回数据，并可以对其进行解析

### 启动 IMS

frameworks/base/services/core/java/com/android/server/input/InputManagerService.java

1. **public** **void** **start**() {
2. Slog.i(TAG, "Starting input manager");
3. nativeStart(mPtr);
4. ...
5. }

调用 native 方法，启动 InputDispatcherThread 与 InputReaderThread 两个重要线程

frameworks/base/services/core/jni/com\_android\_server\_input\_InputManagerService.cpp

1. **static** **void** **nativeStart**(JNIEnv\* env, jclass */\* clazz \*/*, jlong ptr) {
2. NativeInputManager\* im = **reinterpret\_cast**<NativeInputManager\*>(ptr);
3. **status\_t** result = im->getInputManager()->start();
4. **if** (result) {
5. jniThrowRuntimeException(env, "Input manager could not be started.");
6. }
7. }

通过 JNI 调用到 native 后，调用 InputManager 的 start 方法启动 IMS 服务

frameworks/native/services/inputflinger/InputManager.cpp

1. **status\_t** InputManager::start() {
2. **status\_t** result = mDispatcherThread->run("InputDispatcher", PRIORITY\_URGENT\_DISPLAY);
3. result = mReaderThread->run("InputReader", PRIORITY\_URGENT\_DISPLAY);
4. **return** OK;
5. }

可以看到，InputManager 只是启动了两个 native 线程，循环来处理输入事件的数据

frameworks/native/services/inputflinger/InputReader.cpp

1. **bool** InputReaderThread::threadLoop() {
2. mReader->loopOnce();
3. **return** true;
4. }

frameworks/native/services/inputflinger/InputDispatcher.cpp

1. **bool** InputDispatcherThread::threadLoop() {
2. mDispatcher->dispatchOnce();
3. **return** true;
4. }

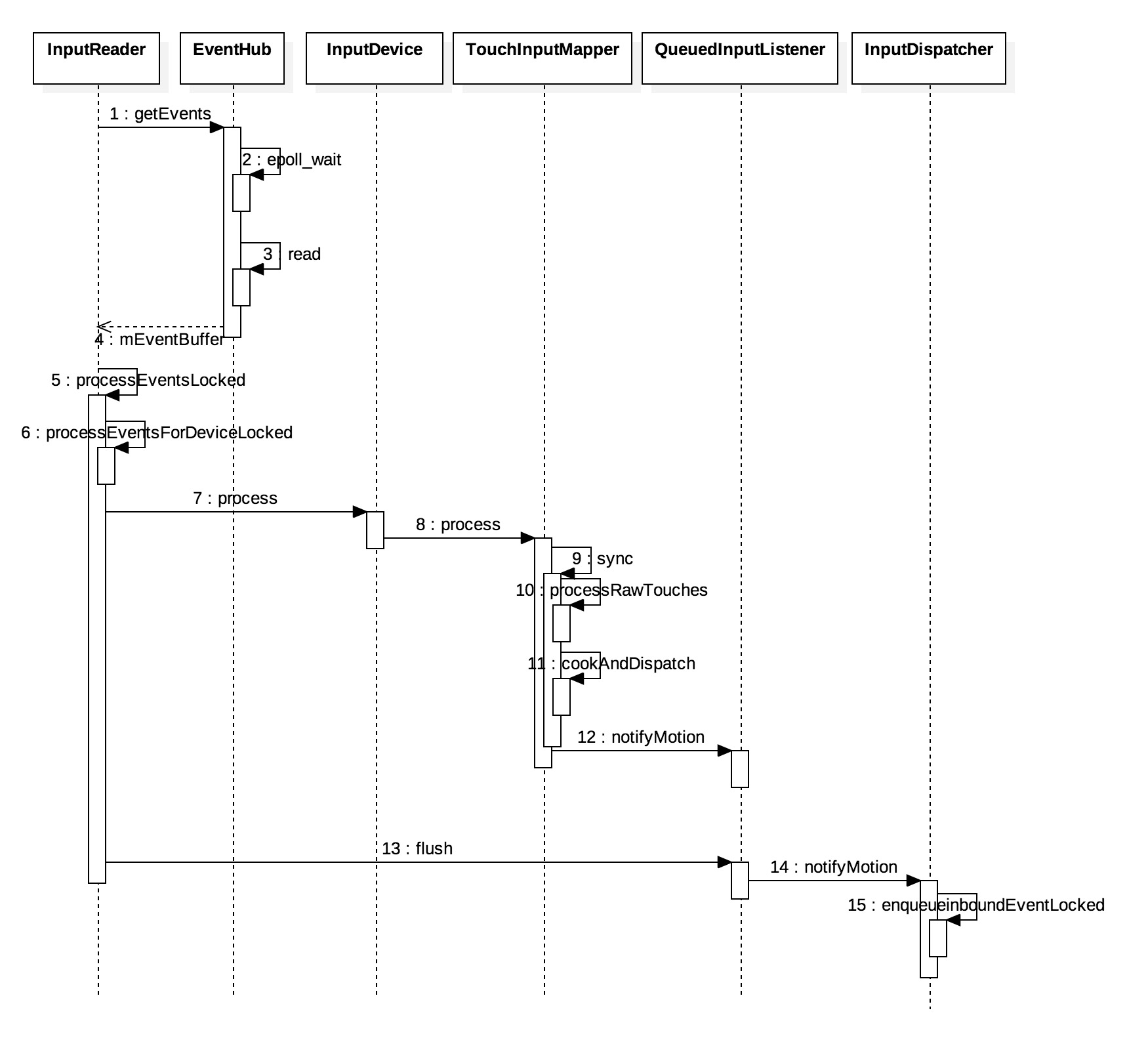
以上，通过两个线程不断调用 loopOnce 与 dispatchOnce，IMS 的基本服务已经初始化完毕，接下来进入正题，从 InputReader 读取输入设备事件开始分析 IMS 是如何一步一步的将设备中的数据分发到用户能看到的视图中的

### 一些背景知识

* 内核中断: 当触屏驱动被挂载到对应的 /dev/input/event0 后，触摸屏幕会导致触摸屏的引脚电平变低，随后CPU引脚监测到电压的变化就会产生中断，中断处理程序就会读取触屏的数据
* inotify: 实现 /dev/input 目录文件状态的监听
* epoll: I/O 多路复用机制，使用 epoll\_ctl 函数进行注册监听 inotify 的文件句柄，当事件到来时会采用类似回调的方式执行回调方法

## 输入事件的读取与解析

[查看大图](https://raw.githubusercontent.com/RickAi/Images/master/blog/IMS_process.jpg)



当 epoll, inotify 与两个线程被初始化，当事件到来时，InputReader 便会率先开始工作。这一节中，IuputReader 会从 EventHub 中读取数据，随后会交给对应的 InputMapper 将事件的类型进行加工。加工完毕后会将事件数据批量的插入到 InputDispatcher 的队列中以供派发

### InputReader.loopOnce：InputReaderThread 的一次读取

frameworks/native/services/inputflinger/InputReader.cpp

1. **void** InputReader::loopOnce() {
2. ...
3. *// 从 EventHub 中读取数据*
4. **size\_t** count = mEventHub->getEvents(timeoutMillis, mEventBuffer, EVENT\_BUFFER\_SIZE);
5. { *// acquire lock*
6. AutoMutex \_l(mLock);
7. mReaderIsAliveCondition.broadcast();
8. **if** (count) {
9. *// 如果有数据则进行处理*
10. processEventsLocked(mEventBuffer, count);
11. }
12. ...
13. **if** (oldGeneration != mGeneration) {
14. inputDevicesChanged = true;
15. getInputDevicesLocked(inputDevices);
16. }
17. } *// release lock*
18. ...
19. *// 讲添加进缓存队列中的事件一齐插入到 InputDispatcher 的队列中*
20. mQueuedListener->flush();
21. }

这段代码基本可以概括 InputReader 的主要职责，逻辑很清晰。其中 EventHub.getEvents 如果获取到了数据才会进行到下一个状态中，大部分时间里，如果没有输入事件 getEvents 将会产生阻塞，不会消耗 CPU 资源

### EventHub.getEvents：IMS 的心跳

frameworks/native/services/inputflinger/EventHub.cpp

1. **size\_t** EventHub::getEvents(**int** timeoutMillis, RawEvent\* buffer, **size\_t** bufferSize) {
2. ALOG\_ASSERT(bufferSize >= 1);
3. AutoMutex \_l(mLock);
4. **struct** **input\_event** **readBuffer**[**bufferSize**];
5. *// 事件的获取不是一个一个的获取，而是采用缓冲队列的形式，分批去获取的*
6. RawEvent\* event = buffer;
7. **for** (;;) {
8. **while** (mPendingEventIndex < mPendingEventCount) {
9. **const** **struct** **epoll\_event**& **eventItem** = **mPendingEventItems**[**mPendingEventIndex**++];
10. **ssize\_t** deviceIndex = mDevices.indexOfKey(eventItem.data.u32);
11. Device\* device = mDevices.valueAt(deviceIndex);
12. **if** (eventItem.events & EPOLLIN) {
13. *// 读取 mPendingEventItems 中的设备数据*
14. **int32\_t** readSize = read(device->fd, readBuffer,
15. **sizeof**(struct input\_event) \* capacity);
16. **for** (**size\_t** i = 0; i < count; i++) {
17. *// 封装成 input\_event*
18. **struct** **input\_event**& **iev** = **readBuffer**[**i**];
19. ...
20. event->deviceId = deviceId;
21. event->type = iev.type;
22. event->code = iev.code;
23. event->value = iev.value;
24. event += 1;
25. capacity -= 1;
26. }
27. }
28. ...
29. *// epoll 监测到设备可读*
30. **int** pollResult = epoll\_wait(mEpollFd, mPendingEventItems, EPOLL\_MAX\_EVENTS, timeoutMillis);
31. }
32. **return** event - buffer;

### InputReader.processEventsLocked: 预处理输入事件

以上的代码是个大致的逻辑，当通过 epoll 获取到可读设备的 fd 后，下一个循环将会读取设备中的数据

frameworks/native/services/inputflinger/InputReader.cpp

1. **void** InputReader::processEventsLocked(**const** RawEvent\* rawEvents, **size\_t** count) {
2. **for** (**const** RawEvent\* rawEvent = rawEvents; count;) {
3. **int32\_t** type = rawEvent->type;
4. **size\_t** batchSize = 1;
5. **if** (type < EventHubInterface::FIRST\_SYNTHETIC\_EVENT) {
6. **int32\_t** deviceId = rawEvent->deviceId;
7. ...
8. processEventsForDeviceLocked(deviceId, rawEvent, batchSize);
9. }
10. count -= batchSize;
11. rawEvent += batchSize;
12. }
13. }
14. **void** InputReader::processEventsForDeviceLocked(**int32\_t** deviceId,
15. **const** RawEvent\* rawEvents, **size\_t** count) {
16. **ssize\_t** deviceIndex = mDevices.indexOfKey(deviceId);
17. ...
18. device->process(rawEvents, count);
19. }

可以看到获取到的事件数据是委托给 InputDevice 调用 process 方法来进行处理的

frameworks/native/services/inputflinger/InputReader.cpp#InputDevice

1. **void** InputDevice::process(**const** RawEvent\* rawEvents, **size\_t** count) {
2. ...
3. **for** (**size\_t** i = 0; i < numMappers; i++) {
4. InputMapper\* mapper = mMappers[i];
5. mapper->process(rawEvent);
6. }
7. ...
8. }
9. **void** TouchInputMapper::process(**const** RawEvent\* rawEvent) {
10. ...
11. **if** (rawEvent->type == EV\_SYN && rawEvent->code == SYN\_REPORT) {
12. sync(rawEvent->when);
13. }
14. }
15. **void** TouchInputMapper::sync(**nsecs\_t** when) {
16. **const** RawState\* last = mRawStatesPending.isEmpty() ?
17. &mCurrentRawState : &mRawStatesPending.top();
18. ...
19. processRawTouches(false */\*timeout\*/*);
20. }
21. **void** TouchInputMapper::cookAndDispatch(**nsecs\_t** when) {
22. ...
23. dispatchPointerUsage(when, policyFlags, pointerUsage);
24. ...
25. }
26. **void** TouchInputMapper::dispatchPointerUsage(**nsecs\_t** when, **uint32\_t** policyFlags,
27. PointerUsage pointerUsage) {
28. ...
29. **switch** (mPointerUsage) {
30. **case** POINTER\_USAGE\_STYLUS:
31. dispatchPointerStylus(when, policyFlags);
32. **break**;
33. **default**:
34. **break**;
35. }
36. }
37. **void** TouchInputMapper::dispatchPointerStylus(**nsecs\_t** when, **uint32\_t** policyFlags) {
38. ...
39. dispatchPointerSimple(when, policyFlags, down, hovering);
40. }
41. **void** TouchInputMapper::dispatchPointerSimple(**nsecs\_t** when, **uint32\_t** policyFlags,
42. **bool** down, **bool** hovering) {
43. ...
44. **if** (mPointerSimple.down && !down) {
45. mPointerSimple.down = false;
46. *// Send up.*
47. *// 将事件封装成 NotifyMotionArgs，随后调用 QueuedInputListener 接口插入到缓存队列中*
48. NotifyMotionArgs **args**(when, getDeviceId(), mSource, policyFlags,
49. AMOTION\_EVENT\_ACTION\_UP, 0, 0, metaState, mLastRawState.buttonState, 0,
50. mViewport.displayId,
51. 1, &mPointerSimple.lastProperties, &mPointerSimple.lastCoords,
52. mOrientedXPrecision, mOrientedYPrecision,
53. mPointerSimple.downTime);
54. getListener()->notifyMotion(&args);
55. }
56. ...
57. }

这一步事件的处理逻辑很复杂，我不准备详细分析了。我这边列出的是精简的关于触摸类型的事件处理，只需记得事件处理完成后，最后的输出结果是将封装好的 NotifyMotionArgs 插入到 QueuedInputListener 队列中即可

frameworks/native/services/inputflinger/InputListener.cpp#QueuedInputListener

1. **void** QueuedInputListener::notifyMotion(**const** NotifyMotionArgs\* args) {
2. mArgsQueue.push(**new** NotifyMotionArgs(\*args));
3. }
4. **void** QueuedInputListener::flush() {
5. **size\_t** count = mArgsQueue.size();
6. **for** (**size\_t** i = 0; i < count; i++) {
7. NotifyArgs\* args = mArgsQueue[i];
8. args->notify(mInnerListener);
9. **delete** args;
10. }
11. mArgsQueue.clear();
12. }

可以看到, 调用notifyXXX系列方法最终都是插入到名为 mArgsQueue 的队列中。当缓冲队列中的所有事件都处理完毕之后，InputReader 会调用 flush 批量将队列中的事件进行 notify

### QueuedInputListener.notify：插入到 InputDispatcher 队列

frameworks/native/services/inputflinger/InputListener.cpp#QueuedInputListener

1. **void** NotifyMotionArgs::notify(**const** sp<InputListenerInterface>& listener) **const** {
2. listener->notifyMotion(**this**);
3. }

以触摸事件为例，notify实际上调用的是 listener 的notifyMotion，而 listener 对象实际是 InputDispatcher 的一个引用，所以最终调用的还是 InputDispatcher 的notifyMotion方法

1. **void** InputDispatcher::notifyMotion(**const** NotifyMotionArgs\* args) {
2. ...
3. **uint32\_t** policyFlags = args->policyFlags;
4. policyFlags |= POLICY\_FLAG\_TRUSTED;
5. mPolicy->interceptMotionBeforeQueueing(args->eventTime, */\*byref\*/* policyFlags);
6. **bool** needWake;
7. { *// acquire lock*
8. mLock.lock();
9. ...
10. *// Just enqueue a new motion event.*
11. MotionEntry\* newEntry = **new** MotionEntry(args->eventTime,
12. args->deviceId, args->source, policyFlags,
13. args->action, args->actionButton, args->flags,
14. args->metaState, args->buttonState,
15. args->edgeFlags, args->xPrecision, args->yPrecision, args->downTime,
16. args->displayId,
17. args->pointerCount, args->pointerProperties, args->pointerCoords, 0, 0);
18. needWake = enqueueInboundEventLocked(newEntry);
19. mLock.unlock();
20. } *// release lock*
21. **if** (needWake) {
22. mLooper->wake();
23. }
24. }
25. **bool** InputDispatcher::enqueueInboundEventLocked(EventEntry\* entry) {
26. **bool** needWake = mInboundQueue.isEmpty();
27. mInboundQueue.enqueueAtTail(entry);
28. ...
29. }

最终会将 NotifyMotionArgs 事件插入到 InputDispatcher 的 mInboundQueue 队列中。到这里 InputReader 的 loopOnce 就结束了，正如方法名阐述的，进行一个输入设备的数据读取，至于后面的事件派发工作就交给 InputDispatcher 来处理了

## 输入事件的派发

### InputDispatcher.dispatchOnce

frameworks/native/services/inputflinger/InputDispatcher.cpp

1. **void** InputDispatcher::dispatchOnce() {
2. **nsecs\_t** nextWakeupTime = LONG\_LONG\_MAX;
3. { *// acquire lock*
4. AutoMutex \_l(mLock);
5. mDispatcherIsAliveCondition.broadcast();
6. ...
7. **if** (!haveCommandsLocked()) {
8. dispatchOnceInnerLocked(&nextWakeupTime);
9. }
10. ...
11. } *// release lock*
12. *// Wait for callback or timeout or wake. (make sure we round up, not down)*
13. **nsecs\_t** currentTime = now();
14. **int** timeoutMillis = toMillisecondTimeoutDelay(currentTime, nextWakeupTime);
15. mLooper->pollOnce(timeoutMillis);
16. }
17. **void** InputDispatcher::dispatchOnceInnerLocked(**nsecs\_t**\* nextWakeupTime) {
18. **if** (! mPendingEvent) {
19. **if** (mInboundQueue.isEmpty()) {
20. ...
21. } **else** {
22. mPendingEvent = mInboundQueue.dequeueAtHead();
23. }
24. }
25. ...
26. **case** EventEntry::TYPE\_MOTION: {
27. MotionEntry\* typedEntry = **static\_cast**<MotionEntry\*>(mPendingEvent);
28. ...
29. done = dispatchMotionLocked(currentTime, typedEntry,
30. &dropReason, nextWakeupTime);
31. **break**;
32. }
33. }
34. **bool** InputDispatcher::dispatchMotionLocked(
35. **nsecs\_t** currentTime, MotionEntry\* entry, DropReason\* dropReason, **nsecs\_t**\* nextWakeupTime) {
36. Vector<InputTarget> inputTargets;
37. ...
38. **if** (isPointerEvent) {
39. *// Pointer event. (eg. touchscreen)*
40. injectionResult = findTouchedWindowTargetsLocked(currentTime,
41. entry, inputTargets, nextWakeupTime, &conflictingPointerActions);
42. } **else** {
43. *// Non touch event. (eg. trackball)*
44. injectionResult = findFocusedWindowTargetsLocked(currentTime,
45. entry, inputTargets, nextWakeupTime);
46. }
47. dispatchEventLocked(currentTime, entry, inputTargets);
48. }

这段代码的逻辑是讲触屏的事件数据进行分发处理。期间省略掉了许多操作，其中包括在 findTouchedWindowTargetsLocked 对 ANR 的判断 ，如果监测到派发的时间超过了 5s 还没有处理完成的话，native 就会通知 java 层需要触发 ANR

findTouchedWindowTargetsLocked 同时会找到事件作用的客户端并保存在 inputTargets 列表中

frameworks/native/services/inputflinger/InputDispatcher.cpp

1. **void** InputDispatcher::dispatchEventLocked(**nsecs\_t** currentTime,
2. EventEntry\* eventEntry, **const** Vector<InputTarget>& inputTargets) {
3. ...
4. **for** (**size\_t** i = 0; i < inputTargets.size(); i++) {
5. **const** InputTarget& inputTarget = inputTargets.itemAt(i);
6. **ssize\_t** connectionIndex = getConnectionIndexLocked(inputTarget.inputChannel);
7. **if** (connectionIndex >= 0) {
8. sp<Connection> connection = mConnectionsByFd.valueAt(connectionIndex);
9. prepareDispatchCycleLocked(currentTime, connection, eventEntry, &inputTarget);
10. ...
11. }
12. }
13. **void** InputDispatcher::prepareDispatchCycleLocked(**nsecs\_t** currentTime,
14. **const** sp<Connection>& connection, EventEntry\* eventEntry, **const** InputTarget\* inputTarget) {
15. ...
16. enqueueDispatchEntriesLocked(currentTime, connection, eventEntry, inputTarget);
17. }
18. **void** InputDispatcher::startDispatchCycleLocked(**nsecs\_t** currentTime,
19. **const** sp<Connection>& connection) {
20. ...
21. **case** EventEntry::TYPE\_MOTION: {
22. MotionEntry\* motionEntry = **static\_cast**<MotionEntry\*>(eventEntry);
23. ...
24. *// Publish the motion event.*
25. status = connection->inputPublisher.publishMotionEvent(dispatchEntry->seq,
26. motionEntry->deviceId, motionEntry->source,
27. dispatchEntry->resolvedAction, motionEntry->actionButton,
28. dispatchEntry->resolvedFlags, motionEntry->edgeFlags,
29. motionEntry->metaState, motionEntry->buttonState,
30. xOffset, yOffset, motionEntry->xPrecision, motionEntry->yPrecision,
31. motionEntry->downTime, motionEntry->eventTime,
32. motionEntry->pointerCount, motionEntry->pointerProperties,
33. usingCoords);
34. **break**;
35. }
36. ...
37. }

最终通过获取到的 Connection，调用 InputPublishser 的publishMotionEvent 方法将事件传输到客户端，这里看一下 Connection 的实现，其实就是封装了 InputChannel 成员变量与 window 句柄的类

1. InputDispatcher::Connection::Connection(**const** sp<InputChannel>& inputChannel,
2. **const** sp<InputWindowHandle>& inputWindowHandle, **bool** monitor) :
3. status(STATUS\_NORMAL), inputChannel(inputChannel), inputWindowHandle(inputWindowHandle),
4. monitor(monitor),
5. inputPublisher(inputChannel), inputPublisherBlocked(false) {
6. }

在分析 InputChannel 发送消息到客户端，传输数据事件之前，先看看 InputChannel 的初始化

### InputChannel 服务端的初始化

上一篇文章《从源码角度看Activity显示视图流程》分析视图的显示层看到过 WindowManager.addWindow 的方法，其实在视图进行显示时，也正是发送 Input 事件的 InputChannel 进行初始化的时机

frameworks/base/services/core/java/com/android/server/wm/WindowManagerService.java

1. **public** **int** **addWindow**(Session session, IWindow client, **int** seq,
2. WindowManager.LayoutParams attrs, **int** viewVisibility, **int** displayId,
3. Rect outContentInsets, Rect outStableInsets, Rect outOutsets,
4. InputChannel outInputChannel) {
5. ...
6. **if** (outInputChannel != **null** && (attrs.inputFeatures
7. & WindowManager.LayoutParams.INPUT\_FEATURE\_NO\_INPUT\_CHANNEL) == 0) {
8. String name = win.makeInputChannelName();
9. InputChannel[] inputChannels = InputChannel.openInputChannelPair(name);
10. win.setInputChannel(inputChannels[0]);
11. inputChannels[1].transferTo(outInputChannel);
12. mInputManager.registerInputChannel(win.mInputChannel, win.mInputWindowHandle);
13. }
14. ...
15. }

openInputChannelPair 会调用 native 方法初始化 socketpair，registerInputChannel 则是将 inputChannel 句柄传入 native 给 InputDispatcher

frameworks/base/services/core/jni/com\_android\_server\_input\_InputManagerService.cpp

1. **static** **void** **nativeRegisterInputChannel**(JNIEnv\* env, jclass */\* clazz \*/*,
2. jlong ptr, jobject inputChannelObj, jobject inputWindowHandleObj, jboolean monitor) {
3. ...
4. sp<InputWindowHandle> inputWindowHandle =
5. android\_server\_InputWindowHandle\_getHandle(env, inputWindowHandleObj);
6. **status\_t** status = im->registerInputChannel(
7. env, inputChannel, inputWindowHandle, monitor);
8. ...
9. }
10. **status\_t** NativeInputManager::registerInputChannel(JNIEnv\* */\* env \*/*,
11. **const** sp<InputChannel>& inputChannel,
12. **const** sp<InputWindowHandle>& inputWindowHandle, **bool** monitor) {
13. **return** mInputManager->getDispatcher()->registerInputChannel(
14. inputChannel, inputWindowHandle, monitor);
15. }

frameworks/native/services/inputflinger/InputDispatcher.cpp

1. **status\_t** InputDispatcher::registerInputChannel(**const** sp<InputChannel>& inputChannel,
2. **const** sp<InputWindowHandle>& inputWindowHandle, **bool** monitor) {
3. { *// acquire lock*
4. AutoMutex \_l(mLock);
5. ...
6. sp<Connection> connection = **new** Connection(inputChannel, inputWindowHandle, monitor);
7. **int** fd = inputChannel->getFd();
8. mConnectionsByFd.add(fd, connection);
9. ...
10. mLooper->addFd(fd, 0, ALOOPER\_EVENT\_INPUT, handleReceiveCallback, **this**);
11. } *// release lock*
12. *// Wake the looper because some connections have changed.*
13. mLooper->wake();
14. **return** OK;
15. }

regsiterInputChannel 最终的操作是初始化一个 Connection，将引用传给它并添加到 mConnectionByFd 队列中，可以根据 socketpair 的fd来查询

### InputPublisher.publishMotionEvent

frameworks/native/libs/input/InputTransport.cpp

1. **status\_t** InputPublisher::publishMotionEvent(
2. **uint32\_t** seq,
3. **int32\_t** deviceId,
4. **int32\_t** source,
5. **int32\_t** action,
6. **int32\_t** actionButton,
7. **int32\_t** flags,
8. **int32\_t** edgeFlags,
9. **int32\_t** metaState,
10. **int32\_t** buttonState,
11. **float** xOffset,
12. **float** yOffset,
13. **float** xPrecision,
14. **float** yPrecision,
15. **nsecs\_t** downTime,
16. **nsecs\_t** eventTime,
17. **uint32\_t** pointerCount,
18. **const** PointerProperties\* pointerProperties,
19. **const** PointerCoords\* pointerCoords) {
20. InputMessage msg;
21. msg.header.type = InputMessage::TYPE\_MOTION;
22. msg.body.motion.seq = seq;
23. msg.body.motion.deviceId = deviceId;
24. msg.body.motion.source = source;
25. msg.body.motion.action = action;
26. msg.body.motion.actionButton = actionButton;
27. msg.body.motion.flags = flags;
28. msg.body.motion.edgeFlags = edgeFlags;
29. msg.body.motion.metaState = metaState;
30. msg.body.motion.buttonState = buttonState;
31. msg.body.motion.xOffset = xOffset;
32. msg.body.motion.yOffset = yOffset;
33. msg.body.motion.xPrecision = xPrecision;
34. msg.body.motion.yPrecision = yPrecision;
35. msg.body.motion.downTime = downTime;
36. msg.body.motion.eventTime = eventTime;
37. msg.body.motion.pointerCount = pointerCount;
38. **for** (**uint32\_t** i = 0; i < pointerCount; i++) {
39. msg.body.motion.pointers[i].properties.copyFrom(pointerProperties[i]);
40. msg.body.motion.pointers[i].coords.copyFrom(pointerCoords[i]);
41. }
42. **return** mChannel->sendMessage(&msg);
43. }
44. **status\_t** InputChannel::sendMessage(**const** InputMessage\* msg) {
45. **size\_t** msgLength = msg->size();
46. **ssize\_t** nWrite;
47. **do** {
48. nWrite = ::send(mFd, msg, msgLength, MSG\_DONTWAIT | MSG\_NOSIGNAL);
49. } **while** (nWrite == -1 && errno == EINTR);
50. ...
51. }

通过将信息封装成 InputMessage，随后使用 InputChannel.sendMessage 将信息发送到客户端

### InputChannel 客户端的初始化

frameworks/base/core/java/android/view/ViewRootImpl.java

1. **public** **void** **setView**(View view, WindowManager.LayoutParams attrs, View panelParentView) {
2. res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
3. getHostVisibility(), mDisplay.getDisplayId(),
4. mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
5. mAttachInfo.mOutsets, mInputChannel);
6. **if** (mInputChannel != **null**) {
7. **if** (mInputQueueCallback != **null**) {
8. mInputQueue = **new** InputQueue();
9. mInputQueueCallback.onInputQueueCreated(mInputQueue);
10. }
11. mInputEventReceiver = **new** WindowInputEventReceiver(mInputChannel,
12. Looper.myLooper());
13. }
14. }
15. **final** **class** **WindowInputEventReceiver** **extends** **InputEventReceiver** {
16. **public** **WindowInputEventReceiver**(InputChannel inputChannel, Looper looper) {
17. **super**(inputChannel, looper);
18. }
19. }

ViewRootImpl.setView 时，与WMS 通信完成之后会去初始化 InputChannel 的客户端

frameworks/base/core/java/android/view/InputEventReceiver.java

1. **public** **InputEventReceiver**(InputChannel inputChannel, Looper looper) {
2. ...
3. mInputChannel = inputChannel;
4. mMessageQueue = looper.getQueue();
5. mReceiverPtr = nativeInit(**new** WeakReference<InputEventReceiver>(**this**),
6. inputChannel, mMessageQueue);
7. mCloseGuard.open("dispose");
8. }

通过调用 nativeInit 在 native 层进行初始化

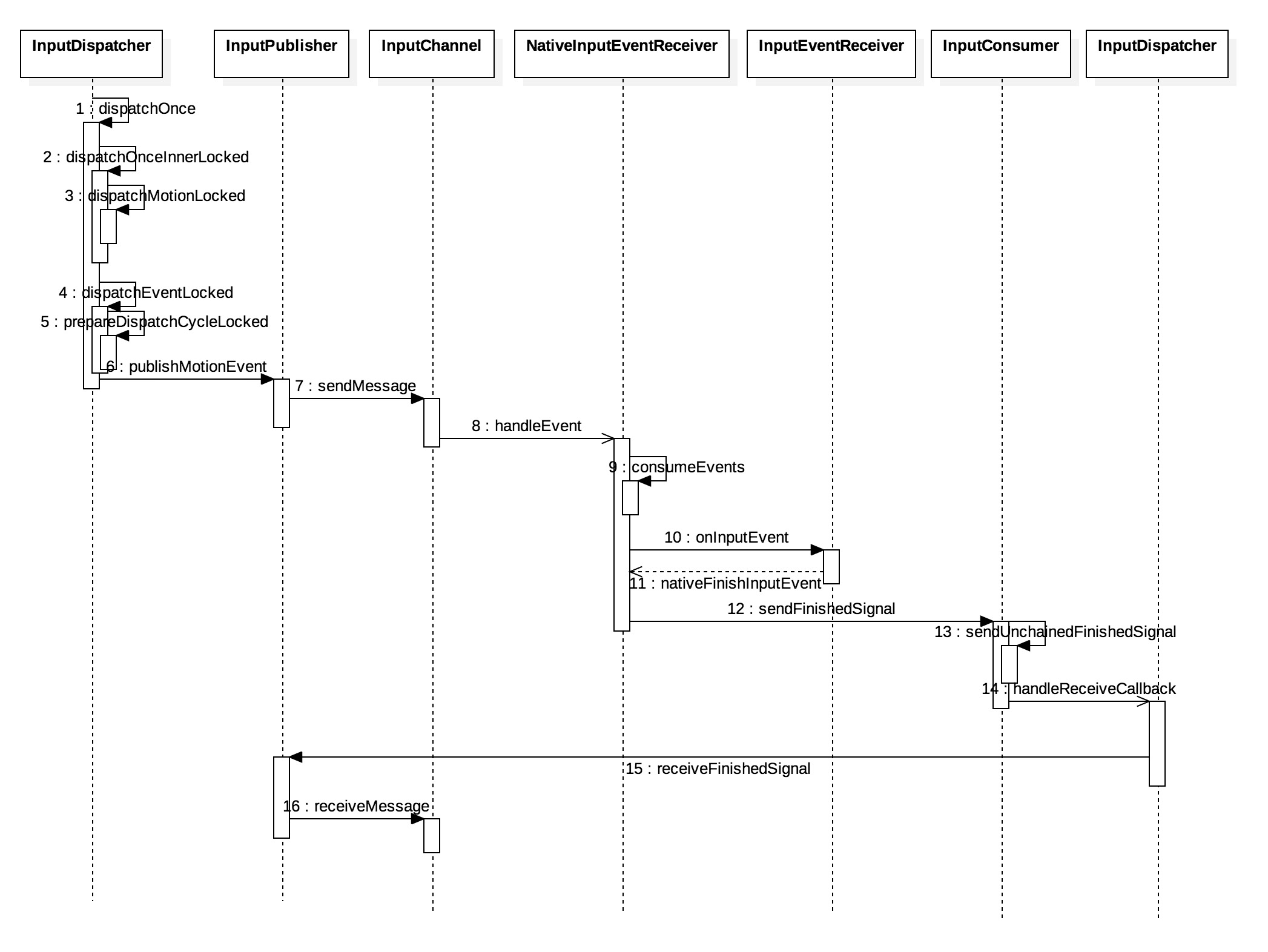
frameworks/base/core/jni/android\_view\_InputEventReceiver.cpp

1. **static** jlong **nativeInit**(JNIEnv\* env, jclass clazz, jobject receiverWeak,
2. jobject inputChannelObj, jobject messageQueueObj) {
3. sp<InputChannel> inputChannel = android\_view\_InputChannel\_getInputChannel(env,
4. inputChannelObj);
5. ...
6. sp<NativeInputEventReceiver> receiver = **new** NativeInputEventReceiver(env,
7. receiverWeak, inputChannel, messageQueue);
8. **status\_t** status = receiver->initialize();
9. ...
10. **return** **reinterpret\_cast**<jlong>(receiver.get());
11. }
12. **status\_t** NativeInputEventReceiver::initialize() {
13. setFdEvents(ALOOPER\_EVENT\_INPUT);
14. **return** OK;
15. }
16. **void** NativeInputEventReceiver::setFdEvents(**int** events) {
17. **if** (mFdEvents != events) {
18. mFdEvents = events;
19. **int** fd = mInputConsumer.getChannel()->getFd();
20. **if** (events) {
21. mMessageQueue->getLooper()->addFd(fd, 0, events, **this**, NULL);
22. } **else** {
23. mMessageQueue->getLooper()->removeFd(fd);
24. }
25. }
26. }

真正的实现是在 NativeInputEventReceiver.initialize 方法进行的，最终使用的是传入的 MessageQueue 中的 Looper 来监听 InputChannel 客户端的 fd 实现的，本质上还是使用了 epoll 机制来监听 fd 的可读状态。当事件到来时，会调用回调方法 handleEvent 来处理事件

## 输入事件的处理与反馈

[查看大图](https://raw.githubusercontent.com/RickAi/Images/master/blog/IMS_send_recev.jpg)



frameworks/base/core/jni/android\_view\_InputEventReceiver.cpp

1. **int** NativeInputEventReceiver::handleEvent(**int** receiveFd, **int** events, **void**\* data) {
2. ...
3. **if** (events & ALOOPER\_EVENT\_INPUT) {
4. JNIEnv\* env = AndroidRuntime::getJNIEnv();
5. **status\_t** status = consumeEvents(env, false */\*consumeBatches\*/*, -1, NULL);
6. mMessageQueue->raiseAndClearException(env, "handleReceiveCallback");
7. **return** status == OK || status == NO\_MEMORY ? 1 : 0;
8. }
9. ...
10. }
11. **status\_t** NativeInputEventReceiver::consumeEvents(JNIEnv\* env,
12. **bool** consumeBatches, **nsecs\_t** frameTime, **bool**\* outConsumedBatch) {
13. **for** (;;) {
14. env->CallVoidMethod(receiverObj.get(), gInputEventReceiverClassInfo.dispatchBatchedInputEventPending);
15. }
16. ...
17. mInputConsumer.sendFinishedSignal(seq, false);
18. }

当事件到来时，handleEvent 会调用 java 方法，通知上层继续处理输入事件，当事件处理完成之后会调用 sendFinishedSignal 通过服务端事件处理已经完成

frameworks/base/core/java/android/view/InputEventReceiver.java

1. **public** **void** **onInputEvent**(InputEvent event) {
2. finishInputEvent(event, **false**);
3. }

此处是 java 层收到输入事件的入口，后续的文章会分析 View 的事件派发机制，将会接着这条线继续分析

frameworks/native/libs/input/InputTransport.cpp

1. **status\_t** InputConsumer::sendFinishedSignal(**uint32\_t** seq, **bool** handled) {
2. ...
3. **return** sendUnchainedFinishedSignal(seq, handled);
4. }
5. **status\_t** InputConsumer::sendUnchainedFinishedSignal(**uint32\_t** seq, **bool** handled) {
6. InputMessage msg;
7. msg.header.type = InputMessage::TYPE\_FINISHED;
8. msg.body.finished.seq = seq;
9. msg.body.finished.handled = handled;
10. **return** mChannel->sendMessage(&msg);
11. }

将 java 层处理完毕输入事件的信息由客户端发向服务端

frameworks/native/services/inputflinger/InputDispatcher.cpp

1. **int** InputDispatcher::handleReceiveCallback(**int** fd, **int** events, **void**\* data) {
2. ...
3. **ssize\_t** connectionIndex = d->mConnectionsByFd.indexOfKey(fd);
4. **for** (;;) {
5. **uint32\_t** seq;
6. **bool** handled;
7. status = connection->inputPublisher.receiveFinishedSignal(&seq, &handled);
8. **if** (status) {
9. **break**;
10. }
11. d->finishDispatchCycleLocked(currentTime, connection, seq, handled);
12. gotOne = true;
13. }
14. }
15. **void** InputDispatcher::finishDispatchCycleLocked(**nsecs\_t** currentTime,
16. **const** sp<Connection>& connection, **uint32\_t** seq, **bool** handled) {
17. ...
18. onDispatchCycleFinishedLocked(currentTime, connection, seq, handled);
19. }

handleReceiveCallback 也是使用 Looper 中的 epoll 机制监听的回调方式，注册的方式和客户端一样，这里就不累述了

## 总结

InputManagerService 相对于 Android 其它几个大服务来说代码量虽然较少，看起来也没有那么复杂，整个流程从 Input 事件的读取、派发、处理和反馈都井然有序，如同一个生产工厂，虽然庞大但是有秩序，看起来很有条有理。

因为平时我接触 Android 源码部分最多的还是 AMS，看 Android 源码给我的印象就如同阅读一本史诗级的厚书，虽然有分有量但是及其难啃。读 IMS 的过程中，我明显感觉到代码的设计之优雅，虽然整个对输入事件的处理流程很长，但是各个重要模块的封装都十分精致，对缓冲区、设计模式、epoll和socketpair 的运行恰如其分，读的不仅能够学得 Android 事件的流程，其中的代码设计范式也很值得我去学习

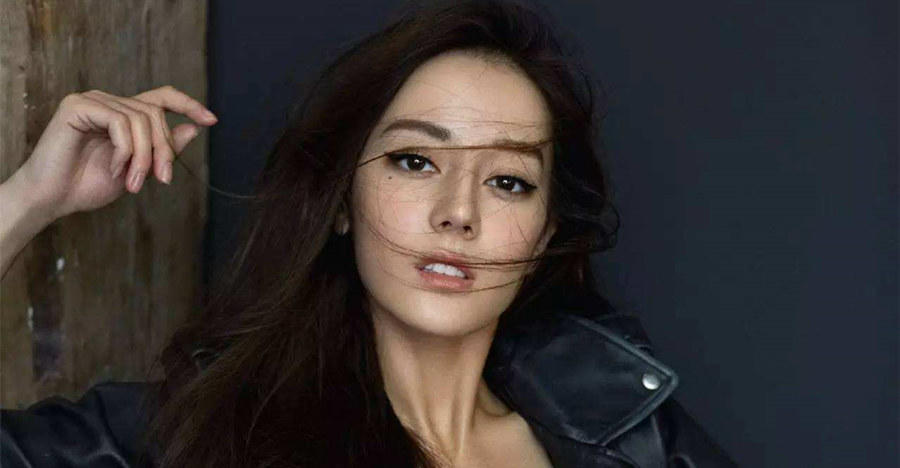
**Android解析WindowManagerService（二）WMS的重要成员和Window的添加过程**

[2017-10-23](http://liuwangshu.cn/framework/wms/2-wms-member.html)

[ANDROID框架层](http://liuwangshu.cn/categories/Android%E6%A1%86%E6%9E%B6%E5%B1%82/)

[ANDROID框架层](http://liuwangshu.cn/tags/Android%E6%A1%86%E6%9E%B6%E5%B1%82/), [ANDROID系统服务](http://liuwangshu.cn/tags/Android%E7%B3%BB%E7%BB%9F%E6%9C%8D%E5%8A%A1/), [WINDOWMANAGERSERVICE](http://liuwangshu.cn/tags/WindowManagerService/)

阅读 1097

[](http://upload-images.jianshu.io/upload_images/1417629-64b3742c779702c0.jpg?imageMogr2/auto-orient/strip|imageView2/2/w/1240)

关联系列  
[Android系统启动系列](http://liuwangshu.cn/tags/Android%E7%B3%BB%E7%BB%9F%E5%90%AF%E5%8A%A8/)  
[Android深入四大组件系列](http://liuwangshu.cn/tags/Android%E6%B7%B1%E5%85%A5%E5%9B%9B%E5%A4%A7%E7%BB%84%E4%BB%B6/)  
[Android应用进程启动过程系列](http://liuwangshu.cn/tags/Android%E5%BA%94%E7%94%A8%E8%BF%9B%E7%A8%8B/)  
[Android解析WindowManager系列](http://liuwangshu.cn/tags/WindowManager/)

**前言**

在本系列的上一篇文章中，我们学习了WMS的诞生，WMS被创建后，它的重要的成员有哪些？Window添加过程的WMS部分做了什么呢？这篇文章会给你解答。

**1.WMS的重要成员**

所谓WMS的重要成员是指WMS中的重要的成员变量，如下所示。  
**frameworks/base/services/core/java/com/android/server/wm/WindowManagerService.java**

|  |
| --- |
| final WindowManagerPolicy mPolicy;  final IActivityManager mActivityManager;  final ActivityManagerInternal mAmInternal;  final AppOpsManager mAppOps;  final DisplaySettings mDisplaySettings;  ...  final ArraySet<Session> mSessions = new ArraySet<>();  final WindowHashMap mWindowMap = new WindowHashMap();  final ArrayList<AppWindowToken> mFinishedStarting = new ArrayList<>();  final ArrayList<AppWindowToken> mFinishedEarlyAnim = new ArrayList<>();  final ArrayList<AppWindowToken> mWindowReplacementTimeouts = new ArrayList<>();  final ArrayList<WindowState> mResizingWindows = new ArrayList<>();  final ArrayList<WindowState> mPendingRemove = new ArrayList<>();  WindowState[] mPendingRemoveTmp = new WindowState[20];  final ArrayList<WindowState> mDestroySurface = new ArrayList<>();  final ArrayList<WindowState> mDestroyPreservedSurface = new ArrayList<>();  ...  final H mH = new H();  ...  final WindowAnimator mAnimator;  ...  final InputManagerService mInputManager |

这里列出了WMS的部分成员变量，下面分别对它们进行简单的介绍。

**mPolicy：WindowManagerPolicy**  
WindowManagerPolicy（WMP）类型的变量。WindowManagerPolicy是窗口管理策略的接口类，用来定义一个窗口策略所要遵循的通用规范，并提供了WindowManager所有的特定的UI行为。它的具体实现类为PhoneWindowManager，这个实现类在WMS创建时被创建。WMP允许定制窗口层级和特殊窗口类型以及关键的调度和布局。

**mSessions：ArraySet**  
ArraySet类型的变量，元素类型为Session。在[Android解析WindowManager（三）Window的添加过程](http://liuwangshu.cn/framework/wm/3-add-window.html)这篇文章中我提到过Session，它主要用于进程间通信，其他的应用程序进程想要和WMS进程进行通信就需要经过Session，并且每个应用程序进程都会对应一个Session，WMS保存这些Session用来记录所有向WMS提出窗口管理服务的客户端。  
**mWindowMap：WindowHashMap**  
WindowHashMap类型的变量，WindowHashMap继承了HashMap，它限制了HashMap的key值的类型为IBinder，value值的类型为WindowState。WindowState用于保存窗口的信息，在WMS中它用来描述一个窗口。综上得出结论，mWindowMap就是用来保存WMS中各种窗口的集合。

**mFinishedStarting：ArrayList**  
ArrayList类型的变量，元素类型为AppWindowToken，它是WindowToken的子类。要想理解mFinishedStarting的含义，需要先了解WindowToken是什么。WindowToken主要有两个作用：

* 可以理解为窗口令牌，当应用程序想要向WMS申请新创建一个窗口，则需要向WMS出示有效的WindowToken。AppWindowToken作为WindowToken的子类，主要用来描述应用程序的WindowToken结构，  
  应用程序中每个Activity都对应一个AppWindowToken。
* WindowToken会将相同组件（比如Acitivity）的窗口（WindowState）集合在一起，方便管理。

mFinishedStarting就是用于存储已经完成启动的应用程序窗口（比如Acitivity）的AppWindowToken的列表。  
除了mFinishedStarting，还有类似的mFinishedEarlyAnim和mWindowReplacementTimeouts，其中mFinishedEarlyAnim存储了已经完成窗口绘制并且不需要展示任何已保存surface的应用程序窗口的AppWindowToken。mWindowReplacementTimeout存储了等待更换的应用程序窗口的AppWindowToken，如果更换不及时，旧窗口就需要被处理。

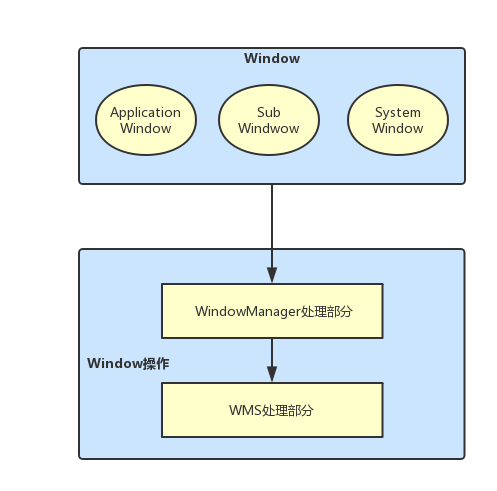
**mResizingWindows：ArrayList**  
ArrayList类型的变量，元素类型为WindowState。  
mResizingWindows是用来存储正在调整大小的窗口的列表。与mResizingWindows类似的还有mPendingRemove、mDestroySurface和mDestroyPreservedSurface等等。其中mPendingRemove是在内存耗尽时设置的，里面存有需要强制删除的窗口。mDestroySurface里面存有需要被Destroy的Surface。mDestroyPreservedSurface里面存有窗口需要保存的等待销毁的Surface，为什么窗口要保存这些Surface？这是因为当窗口经历Surface变化时，窗口需要一直保持旧Surface，直到新Surface的第一帧绘制完成。

**mAnimator：WindowAnimator**  
WindowAnimator类型的变量，用于管理窗口的动画以及特效动画。

**mH：H**  
H类型的变量，系统的Handler类，用于将任务加入到主线程的消息队列中，这样代码逻辑就会在主线程中执行。

**mInputManager：InputManagerService**  
InputManagerService类型的变量，输入系统的管理者。InputManagerService（IMS）会对触摸事件进行处理，它会寻找一个最合适的窗口来处理触摸反馈信息，WMS是窗口的管理者，因此，WMS“理所应当”的成为了输入系统的中转站，WMS包含了IMS的引用不足为怪。

**2.Window的添加过程（WMS部分）**

我们知道Window的操作分为两大部分，一部分是WindowManager处理部分，另一部分是WMS处理部分，如下所示。[](http://upload-images.jianshu.io/upload_images/1417629-a2307e2c73db270d.png?imageMogr2/auto-orient/strip|imageView2/2/w/1240)  
在[Android解析WindowManager（三）Window的添加过程](http://liuwangshu.cn/framework/wm/3-add-window.html)这篇文章中，我讲解了Window的添加过程的WindowManager处理部分，这一篇文章我们接着来学习Window的添加过程的WMS部分。  
无论是系统窗口还是Activity，它们的Window的添加过程都会调用WMS的addWindow方法，由于这个方法代码逻辑比较多，这里分为3个部分来阅读。  
**frameworks/base/services/core/java/com/android/server/wm/WindowManagerService.java**

**addWindow方法part1**

|  |
| --- |
| public int addWindow(Session session, IWindow client, int seq,  WindowManager.LayoutParams attrs, int viewVisibility, int displayId,  Rect outContentInsets, Rect outStableInsets, Rect outOutsets,  InputChannel outInputChannel) {  int[] appOp = new int[1];  int res = mPolicy.checkAddPermission(attrs, appOp);//1  if (res != WindowManagerGlobal.ADD\_OKAY) {  return res;  }  ...  synchronized(mWindowMap) {  if (!mDisplayReady) {  throw new IllegalStateException("Display has not been initialialized");  }  final DisplayContent displayContent = mRoot.getDisplayContentOrCreate(displayId);//2  if (displayContent == null) {  Slog.w(TAG\_WM, "Attempted to add window to a display that does not exist: "  + displayId + ". Aborting.");  return WindowManagerGlobal.ADD\_INVALID\_DISPLAY;  }  ...  if (type >= FIRST\_SUB\_WINDOW && type <= LAST\_SUB\_WINDOW) {//3  parentWindow = windowForClientLocked(null, attrs.token, false);//4  if (parentWindow == null) {  Slog.w(TAG\_WM, "Attempted to add window with token that is not a window: "  + attrs.token + ". Aborting.");  return WindowManagerGlobal.ADD\_BAD\_SUBWINDOW\_TOKEN;  }  if (parentWindow.mAttrs.type >= FIRST\_SUB\_WINDOW  && parentWindow.mAttrs.type <= LAST\_SUB\_WINDOW) {  Slog.w(TAG\_WM, "Attempted to add window with token that is a sub-window: "  + attrs.token + ". Aborting.");  return WindowManagerGlobal.ADD\_BAD\_SUBWINDOW\_TOKEN;  }  }  ...  }  ...  } |

WMS的addWindow返回的是addWindow的各种状态，比如添加Window成功，无效的display等等，这些状态被定义在WindowManagerGlobal中。  
注释1处根据Window的属性，调用WMP的checkAddPermission方法来检查权限，具体的实现在PhoneWindowManager的checkAddPermission方法中，如果没有权限则不会执行后续的代码逻辑。注释2处通过displayId来获得窗口要添加到哪个DisplayContent上，如果没有找到DisplayContent，则返回WindowManagerGlobal.ADD\_INVALID\_DISPLAY这一状态，其中DisplayContent用来描述一块屏幕。注释3处，type代表一个窗口的类型，它的数值介于FIRST\_SUB\_WINDOW和LAST\_SUB\_WINDOW之间（1000~1999），这个数值定义在WindowManager中，说明这个窗口是一个子窗口，不了解窗口类型取值范围的请阅读[Android解析WindowManager（二）Window的属性](http://liuwangshu.cn/framework/wm/2-window-property.html)这篇文章。注释4处，attrs.token是IBinder类型的对象，windowForClientLocked方法内部会根据attrs.token作为key值从mWindowMap中得到该子窗口的父窗口。接着对父窗口进行判断，如果父窗口为null或者type的取值范围不正确则会返回错误的状态。

**addWindow方法part2**

|  |
| --- |
| ...  AppWindowToken atoken = null;  final boolean hasParent = parentWindow != null;  WindowToken token = displayContent.getWindowToken(  hasParent ? parentWindow.mAttrs.token : attrs.token);//1  final int rootType = hasParent ? parentWindow.mAttrs.type : type;//2  boolean addToastWindowRequiresToken = false;  if (token == null) {  if (rootType >= FIRST\_APPLICATION\_WINDOW && rootType <= LAST\_APPLICATION\_WINDOW) {  Slog.w(TAG\_WM, "Attempted to add application window with unknown token "  + attrs.token + ". Aborting.");  return WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN;  }  if (rootType == TYPE\_INPUT\_METHOD) {  Slog.w(TAG\_WM, "Attempted to add input method window with unknown token "  + attrs.token + ". Aborting.");  return WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN;  }  if (rootType == TYPE\_VOICE\_INTERACTION) {  Slog.w(TAG\_WM, "Attempted to add voice interaction window with unknown token "  + attrs.token + ". Aborting.");  return WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN;  }  if (rootType == TYPE\_WALLPAPER) {  Slog.w(TAG\_WM, "Attempted to add wallpaper window with unknown token "  + attrs.token + ". Aborting.");  return WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN;  }  ...  if (type == TYPE\_TOAST) {  // Apps targeting SDK above N MR1 cannot arbitrary add toast windows.  if (doesAddToastWindowRequireToken(attrs.packageName, callingUid,  parentWindow)) {  Slog.w(TAG\_WM, "Attempted to add a toast window with unknown token "  + attrs.token + ". Aborting.");  return WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN;  }  }  final IBinder binder = attrs.token != null ? attrs.token : client.asBinder();  token = new WindowToken(this, binder, type, false, displayContent,  session.mCanAddInternalSystemWindow);//3  } else if (rootType >= FIRST\_APPLICATION\_WINDOW && rootType <= LAST\_APPLICATION\_WINDOW) {//4  atoken = token.asAppWindowToken();//5  if (atoken == null) {  Slog.w(TAG\_WM, "Attempted to add window with non-application token "  + token + ". Aborting.");  return WindowManagerGlobal.ADD\_NOT\_APP\_TOKEN;  } else if (atoken.removed) {  Slog.w(TAG\_WM, "Attempted to add window with exiting application token "  + token + ". Aborting.");  return WindowManagerGlobal.ADD\_APP\_EXITING;  }  } else if (rootType == TYPE\_INPUT\_METHOD) {  if (token.windowType != TYPE\_INPUT\_METHOD) {  Slog.w(TAG\_WM, "Attempted to add input method window with bad token "  + attrs.token + ". Aborting.");  return WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN;  }  }  ... |

注释1处通过displayContent的getWindowToken方法来得到WindowToken。注释2处，如果有父窗口就将父窗口的type值赋值给rootType，如果没有将当前窗口的type值赋值给rootType。接下来如果WindowToken为null，则根据rootType或者type的值进行区分判断，如果rootType值等于TYPE\_INPUT\_METHOD、TYPE\_WALLPAPER等值时，则返回状态值WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN，说明rootType值等于TYPE\_INPUT\_METHOD、TYPE\_WALLPAPER等值时是不允许WindowToken为null的。通过多次的条件判断筛选，最后会在注释3处隐式创建WindowToken，这说明当我们添加窗口时是可以不向WMS提供WindowToken的，前提是rootType和type的值不为前面条件判断筛选的值。WindowToken隐式和显式的创建肯定是要加以区分的，注释3处的第4个参数为false就代表这个WindowToken是隐式创建的。接下来的代码逻辑就是WindowToken不为null的情况，根据rootType和type的值进行判断，比如在注释4处判断如果窗口为应用程序窗口，在注释5处会将WindowToken转换为专门针对应用程序窗口的AppWindowToken，然后根据AppWindowToken的值进行后续的判断。

**addWindow方法part3**

|  |
| --- |
| ...  final WindowState win = new WindowState(this, session, client, token, parentWindow,  appOp[0], seq, attrs, viewVisibility, session.mUid,  session.mCanAddInternalSystemWindow);//1  if (win.mDeathRecipient == null) {//2  // Client has apparently died, so there is no reason to  // continue.  Slog.w(TAG\_WM, "Adding window client " + client.asBinder()  + " that is dead, aborting.");  return WindowManagerGlobal.ADD\_APP\_EXITING;  }  if (win.getDisplayContent() == null) {//3  Slog.w(TAG\_WM, "Adding window to Display that has been removed.");  return WindowManagerGlobal.ADD\_INVALID\_DISPLAY;  }  mPolicy.adjustWindowParamsLw(win.mAttrs);//4  win.setShowToOwnerOnlyLocked(mPolicy.checkShowToOwnerOnly(attrs));  res = mPolicy.prepareAddWindowLw(win, attrs);//5  ...  win.attach();  mWindowMap.put(client.asBinder(), win);//6  if (win.mAppOp != AppOpsManager.OP\_NONE) {  int startOpResult = mAppOps.startOpNoThrow(win.mAppOp, win.getOwningUid(),  win.getOwningPackage());  if ((startOpResult != AppOpsManager.MODE\_ALLOWED) &&  (startOpResult != AppOpsManager.MODE\_DEFAULT)) {  win.setAppOpVisibilityLw(false);  }  }  final AppWindowToken aToken = token.asAppWindowToken();  if (type == TYPE\_APPLICATION\_STARTING && aToken != null) {  aToken.startingWindow = win;  if (DEBUG\_STARTING\_WINDOW) Slog.v (TAG\_WM, "addWindow: " + aToken  + " startingWindow=" + win);  }  boolean imMayMove = true;  win.mToken.addWindow(win);//7  if (type == TYPE\_INPUT\_METHOD) {  win.mGivenInsetsPending = true;  setInputMethodWindowLocked(win);  imMayMove = false;  } else if (type == TYPE\_INPUT\_METHOD\_DIALOG) {  displayContent.computeImeTarget(true /\* updateImeTarget \*/);  imMayMove = false;  } else {  if (type == TYPE\_WALLPAPER) {  displayContent.mWallpaperController.clearLastWallpaperTimeoutTime();  displayContent.pendingLayoutChanges |= FINISH\_LAYOUT\_REDO\_WALLPAPER;  } else if ((attrs.flags&FLAG\_SHOW\_WALLPAPER) != 0) {  displayContent.pendingLayoutChanges |= FINISH\_LAYOUT\_REDO\_WALLPAPER;  } else if (displayContent.mWallpaperController.isBelowWallpaperTarget(win)) {  displayContent.pendingLayoutChanges |= FINISH\_LAYOUT\_REDO\_WALLPAPER;  }  }  ... |

在注释1处创建了WindowState，它存有窗口的所有的状态信息，在WMS中它代表一个窗口。从WindowState传入的参数，可以发现WindowState中包含了WMS、Session、WindowToken、父类的WindowState、LayoutParams等信息。紧接着在注释2和3处分别判断请求添加窗口的客户端是否已经死亡、窗口的DisplayContent是否为null，如果是则不会再执行下面的代码逻辑。注释4处调用了WMP的adjustWindowParamsLw方法，该方法的实现在PhoneWindowManager中，会根据窗口的type对窗口的LayoutParams的一些成员变量进行修改。注释5处调用WMP的prepareAddWindowLw方法，用于准备将窗口添加到系统中。  
注释6处将WindowState添加到mWindowMap中。注释7处将WindowState添加到该WindowState对应的WindowToken中(实际是保存在WindowToken的父类WindowContainer中)，这样WindowToken就包含了相同组件的WindowState。

**addWindow方法总结**

addWindow方法分了3个部分来进行讲解，主要就是做了下面4件事：

1. 对所要添加的窗口进行检查，如果窗口不满足一些条件，就不会再执行下面的代码逻辑。
2. WindowToken相关的处理，比如有的窗口类型需要提供WindowToken，没有提供的话就不会执行下面的代码逻辑，有的窗口类型则需要由WMS隐式创建WindowToken。
3. WindowState的创建和相关处理，将WindowToken和WindowState相关联。
4. 创建和配置DisplayContent，完成窗口添加到系统前的准备工作。

**结语**

在本篇文章中我们首先学习了WMS的重要成员，了解这些成员有利于对WMS的进一步分析。接下来我们又学习了Window的添加过程的WMS部分，将addWindow方法分为了3个部分来进行讲解，从addWindow方法我们得知WMS有3个重要的类分别是WindowToken、WindowState和DisplayContent，关于它们会在本系列后续的文章中进行介绍。

参考资料  
《深入理解Android卷III》

2016-04-17 发布

[Android视图架构详解](https://segmentfault.com/a/1190000004955256)

* [view-architecture](https://segmentfault.com/t/view-architecture/blogs)

* [[https://avatar-static.segmentfault.com/287/281/287281599-5a263edae47e9_small](https://segmentfault.com/t/android/blogs) android](https://segmentfault.com/t/android/blogs)

 1.9k 次阅读  ·  读完需要 23 分钟

0

作者： [ztelur](http://www.jianshu.com/users/481d9f540fb9/latest_articles)  
联系方式：[segmentfault](https://segmentfault.com/u/remcarpediem)，[csdn](http://blog.csdn.net/u012422440)，[github](http://ztelur.github.io/)

本文仅供个人学习，不用于任何形式商业目的，转载请注明原作者、文章来源，链接，版权归原文作者所有。

最近一直在研究View的绘制相关的机制，发现需要补充一下Android View Architecture的相关知识，所以就特地研究了一下这方面的代码，写成本篇文章

为了节约你的时间，本篇文章内容大致如下：

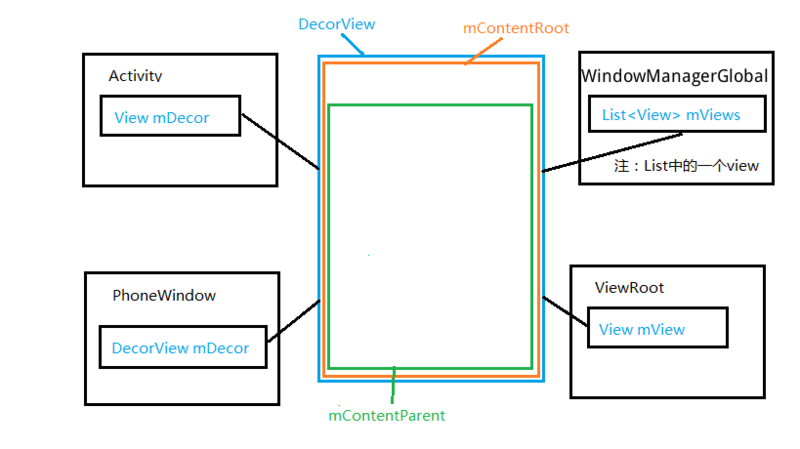
* Activity，DecorView，PhoneWindow和ViewRoot的作用和相关关系

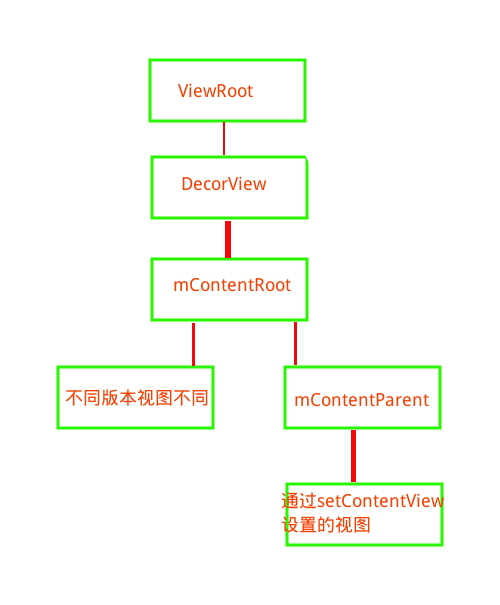
Android View Architecture

先来几张图，大致展现一下Android 视图架构的大概。

感谢网友提醒，泛化和实现这两种关系的箭头画反啦。以后要仔细学一遍UML了，平时经常画，如果有错误可真是闹笑话啊。







Activity和Window

总所周知，Activity并不负责视图控制，它只是控制生命周期和处理事件，真正控制视图的是Window。一个Activity包含了一个Window，Window才是真正代表一个窗口，也就是说Activity可以没有Window，那就相当于是Service了。在ActivityThread中也有控制Service的相关函数或许正好印证了这一点。

Activity和Window的第一次邂逅是在ActivityThread调用Activity的attach()函数时。

//[window]:通过PolicyManager创建window,实现callback函数,所以,当window接收到

//外界状态改变时,会调用activity的方法,

**final** **void** **attach**(Context context, ActivityThread aThread,

Instrumentation instr, IBinder token, **int** ident,

Application application, Intent intent, ActivityInfo info,

CharSequence title, Activity parent, String id,

NonConfigurationInstances lastNonConfigurationInstances,

Configuration config, String referrer, IVoiceInteractor voiceInteractor) {

....

mWindow = PolicyManager.makeNewWindow(**this**);

//当window接收系统发送给它的IO输入事件时,例如键盘和触摸屏事件,就可以转发给相应的Activity

mWindow.setCallback(**this**);

.....

//设置本地窗口管理器

mWindow.setWindowManager(

(WindowManager)context.getSystemService(Context.WINDOW\_SERVICE),

mToken, mComponent.flattenToString(),

(info.flags & ActivityInfo.FLAG\_HARDWARE\_ACCELERATED) != 0);

.....

}

在attach()中，新建一个Window实例作为自己的成员变量，它的类型为PhoneWindow,这是抽象类Window的一个子类。然后设置mWindow的WindowManager。

Window,Activity和DecorView

DecorView是FrameLayout的子类，它可以被认为是Android视图树的根节点视图。DecorView作为顶级View，一般情况下它内部包含一个竖直方向的LinearLayout，在这个LinearLayout里面有上下两个部分（具体情况和Android版本及主体有关），上面的是标题栏，下面的是内容栏。在Activity中通过setContentView所设置的布局文件其实就是被加到内容栏之中的，而内容栏的id是content，在代码中可以通过ViewGroup content = （ViewGroup)findViewById(R.android.id.content)来得到content对应的layout。

Window中有几个视图相关的比较重要的成员变量如下所示:

* mDecor:DecorView的实例，标示Window内部的顶级视图
* mContentParent:setContetView所设置的布局文件就加到这个视图中
* mContentRoot:是DecorView的唯一子视图，内部包含mContentParent,标题栏和状态栏。

Activity中不仅持有一个Window实例，还有一个类型为View的mDecor实例。这个实例和Window中的mDecor实例有什么关系呢？它又是什么时候被创建的呢？

二者其实指向同一个对象，这个对象是在Activity调用setContentView时创建的。我们都知道Activity的setContentView实际上是调用了Window的setContentView方法。

**@Override**

**public** **void** **setContentView**(**int** layoutResID) {

**if** (mContentParent == **null**) { //[window]如何没有DecorView,那么就新建一个

installDecor(); //[window]

} **else** **if** (!hasFeature(FEATURE\_CONTENT\_TRANSITIONS)) {

mContentParent.removeAllViews();

}

....

//[window]第二步,将layout添加到mContentParent

mLayoutInflater.inflate(layoutResID, mContentParent);

.....

}

代码很清楚的显示了布局文件的视图是添加到mContentParent中，而且Window通过installDecor来新建DecorView。

//[window]创建一个decorView

**private** **void** **installDecor**() {

**if** (mDecor == **null**) {

mDecor = generateDecor(); //直接new出一个DecorView返回

....

}

**if** (mContentParent == **null**) {

//[window] 这一步也是很重要的.

mContentParent = generateLayout(mDecor); //mContentParent是setContentVIew的关键啊

.....

}

....

}

**protected** ViewGroup **generateLayout**(DecorView decor) {

// Apply data from current theme.

.......

//[window] 根据不同的style生成不同的decorview啊

View in = mLayoutInflater.inflate(layoutResource, **null**);

// 加入到deco中,所以应该是其第一个child

decor.addView(in, **new** ViewGroup.LayoutParams(MATCH\_PARENT, MATCH\_PARENT));

mContentRoot = (ViewGroup) in; //给DecorView的第一个child是mContentView

// 这是获得所谓的content

ViewGroup contentParent = (ViewGroup)findViewById(ID\_ANDROID\_CONTENT);

}

.....

**return** contentParent;

}

从上述的代码中，我们可以清楚的看到mDecor和mContentParent和mContentRoot的关系。

那么，Activity中的mDecor是何时被赋值的？我们如何确定它和Widnow中的mDecor指向同一个对象呢？我们可以查看ActivityThread的handleResumeActivity函数，它负责处理Activity的resume阶段。在这个函数中，Android直接将Window中的DecorView实例赋值给Activity。

**final** Activity a = r.activity;

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;

Window，DecorView 和 ViewRoot

ViewRoot对应ViewRootImpl类，它是连接WindowManagerService和DecorView的纽带，View的三大流程(测量（measure），布局（layout），绘制（draw）)均通过ViewRoot来完成。ViewRoot并不属于View树的一份子。从源码实现上来看，它既非View的子类，也非View的父类，但是，它实现了ViewParent接口，这让它可以作为View的名义上的父视图。RootView继承了Handler类，可以接收事件并分发，Android的所有触屏事件、按键事件、界面刷新等事件都是通过ViewRoot进行分发的。**ViewRoot可以被理解为“View树的管理者”——它有一个mView成员变量，它指向的对象和上文中Window和Activity的mDecor指向的对象是同一个对象**。

我们来先看一下ViewRoot的创建过程。由于ViewRoot作为WindowMangerService和DecorView的纽带，只有在WindowManager将持有DecorView的Window添加进窗口管理器才创建。我们可以查看WindowMangerGlobal中的addView函数。对WindowManager不太熟悉的同学可以参考[《Window和WindowManager解析》](http://ztelur.github.io/2015/10/28/Window%E5%92%8CWindowManager%E8%A7%A3%E6%9E%90/)

**public** **void** **addView**(View view, ViewGroup.LayoutParams params,

Display display, Window parentWindow) {

// 创建ViewRootImpl,然后将下述对象添加到列表中

....

root = **new** ViewRootImpl(view.getContext(), display);

view.setLayoutParams(wparams);

mViews.add(view);

mRoots.add(root);

mParams.add(wparams);

....

**try** {

// 添加啦!!!!!!!!这是通过ViewRootImpl的setView来完成，这个View就是DecorView实例

root.setView(view, wparams, panelParentView);

} **catch** (RuntimeException e) {

....

}

....

}

那么，Window是什么时候被添加到WindowManager中的呢？我们回到ActivityThread的handleResumeActivity函数。我们都知道Activity的resume阶段就是要显示到屏幕上的阶段，在Activity也就是DecorView将要显示到屏幕时，系统才会调用addView方法。

我们在handleResumeActivity函数中找到了下面一段代码,它调用了Activity的makeVisible()函数。

// ActivityThread

r.activity.makeVisible();

//Activity

//[windows] DecorView正式添加并显示

**void** **makeVisible**() {

**if** (!mWindowAdded) {

ViewManager wm = getWindowManager();

wm.addView(mDecor, getWindow().getAttributes());

mWindowAdded = **true**;

}

mDecor.setVisibility(View.VISIBLE);

}

我们通过源代码发现，正式在makeVisible函数中，系统进行了Window的添加。

**从Android图形架构上理解什么是SurfaceView**

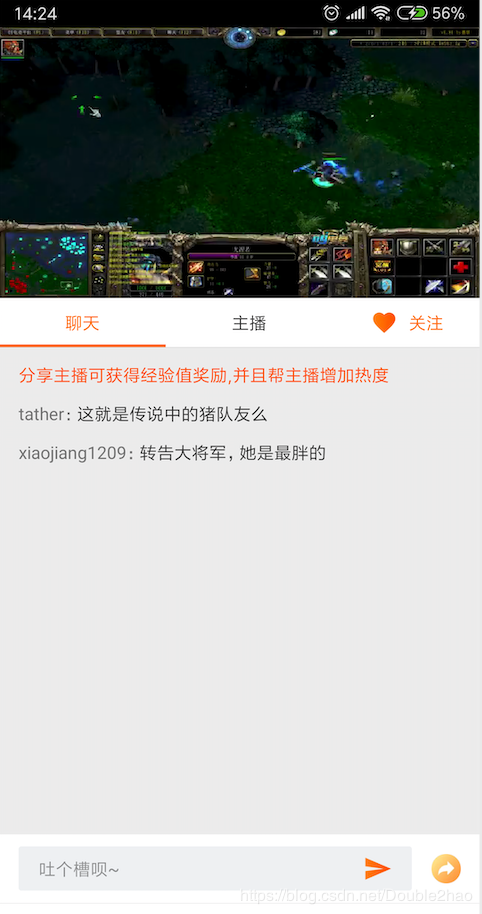
2018年11月24日 16:46:50 [许佳佳233](https://me.csdn.net/Double2hao) 阅读数：104

 版权声明：本文为博主许佳佳原创文章，转载请务必注明出处。 https://blog.csdn.net/Double2hao/article/details/84447935

**概念**

SurfaceView的内容可以在单独的线程和单独的进程中渲染。

**使用场景**

拿斗鱼APP举个例子，如下图：  
  
视频肯定是用的SurfaceView，这样就可以单独线程或者进程渲染。

如果不用单独渲染会有什么问题？  
可以看到下面会有一个聊天框，在用户特别多的时候，下面的聊天框肯定渲染非常频繁，而视频也需要及时渲染，如果两者在同一个线程，那么频繁渲染的聊天框很可能会影响到视频的渲染。  
这样用户只要一多，视频就会显示卡顿，显然这样是不符合预期的。

**图形架构上什么是SurfaceView**

下面是Android图形架构，如果不理解的读者可以看笔者上一篇文章：  
<https://blog.csdn.net/Double2hao/article/details/83928961>

Android图形架构中，有多个Surface，每个Surface之间的渲染不会互相影响，常见的Surface有主屏幕的Surface，状态栏的Surface等等。  
而使用SurfaceView则会有一个单独的Surface，因此SurfaceView的渲染可以和其他Surface不会互相影响，也就是和主屏幕以及状态栏等的渲染不会互相影响。

**注意点：SurfaceView的生命周期**

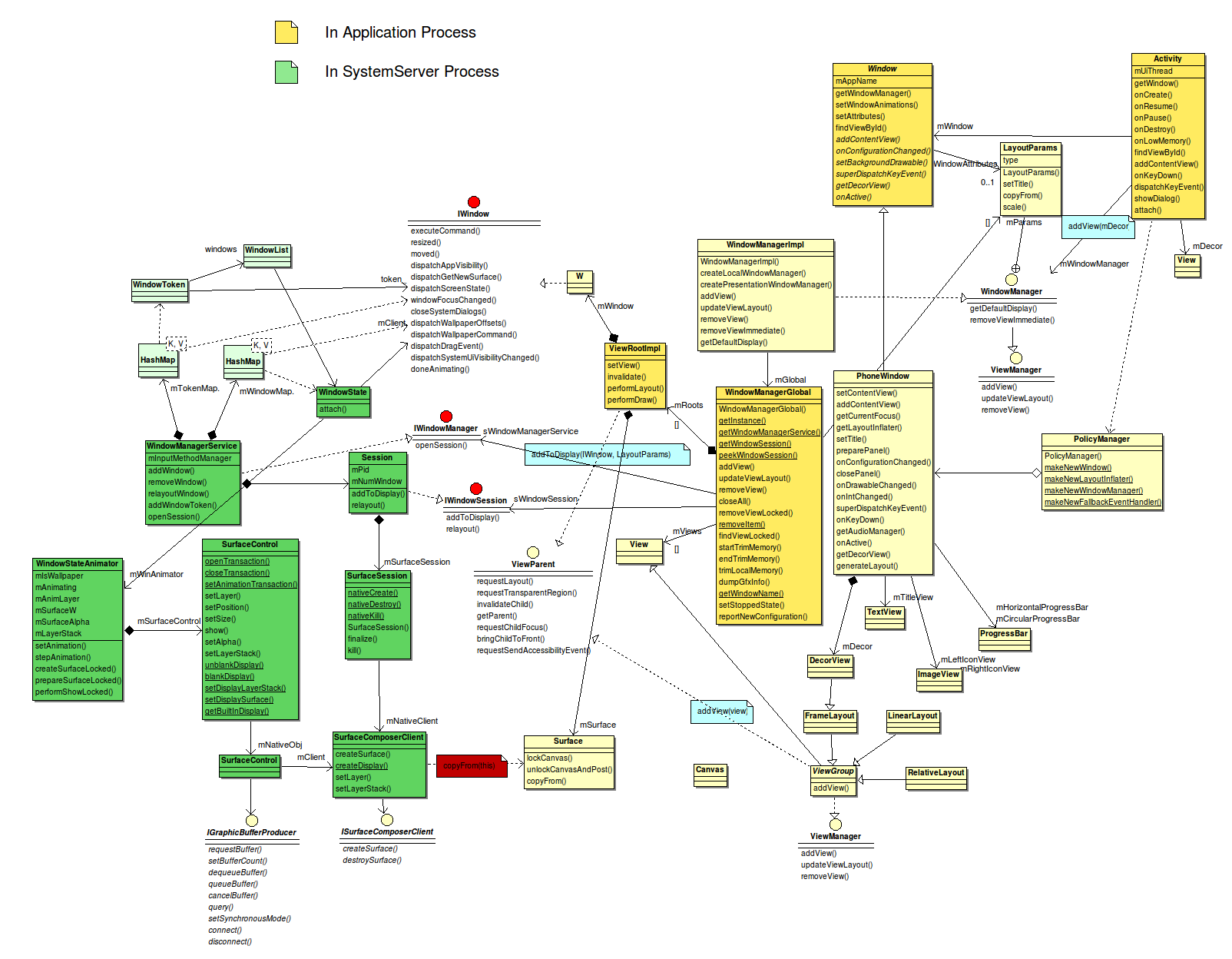
由于SurfaceView有自己单独的Surface，和App主线的View不会在一起渲染，因此SurfaceView的生命周期也是独立于其他View之外的。  
打个比方：  
在Activity中有一个SurfaceView，在Activity onPause()，onDestroy()的时候，SurfaceView不会和其他View一样与Activity的生命周期同步。就拿在Activity onPause()的时候来说，SurfaceView不会管Activity的onPause()，继续执行自己的渲染逻辑。

因此在使用SurfaceView的时候一定要注意同步SurfaceView和主线程的View的生命周期。

[图解Android - Android GUI 系统 (2) - 窗口管理 (View, Canvas, Window Manager)](https://www.cnblogs.com/samchen2009/p/3367496.html)

**Android 的窗口管理系统 (View, Canvas, WindowManager)**

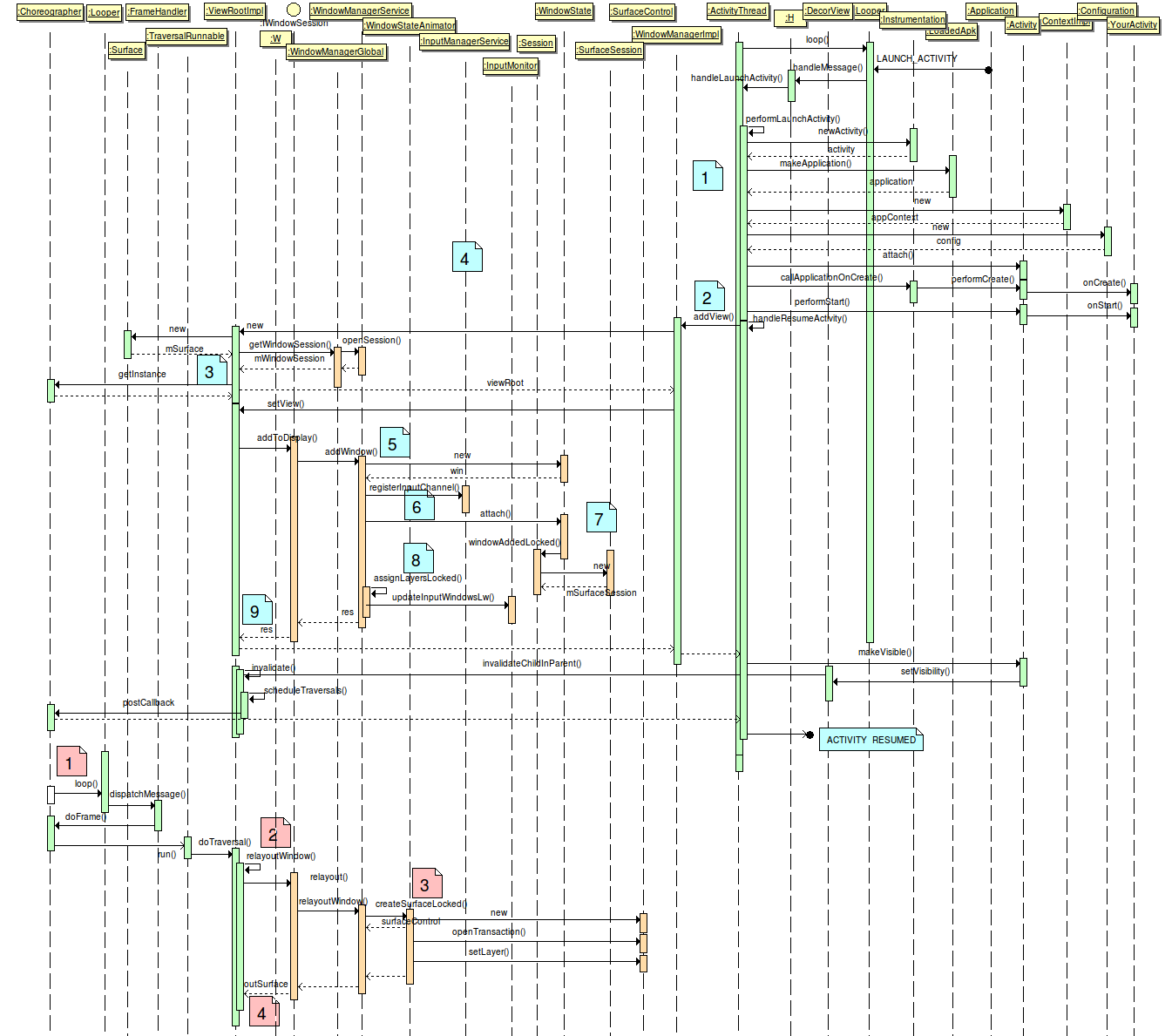
在[图解Android - Zygote 和 System Server 启动分析](http://www.cnblogs.com/samchen2009/p/3294713.html)一 文里，我们已经知道Android 应用程序是怎么创建出来的，大概的流程是 ActivityManagerService -> Zygote -> Fork App, 然后应用程序在ActivityThread 中的进入loop循环等待处理来自AcitivyManagerService的消息。如果一个Android的应用有Acitivity， 那它起来后的第一件事情就是将自己显示出来，这个过程是怎样的？ 这就是本章节要讨论的话题。

[](https://images0.cnblogs.com/blog/563439/201310/15210950-af68b156feff4de5a5526d30d67a64b6.png)

Android 中跟窗口管理相关（不包括显示和按键处理）主要有两个进程，Acitivty所在进程 和 WndowManagerService 所在进程（SystemServer).  上图中用不同颜色区分这两个进程，黄色的模块运行在Activity的进程里，绿色的模块则在System Server内部，本文主要讨论的是WindowManager Service。它们的分工是，Activity进程负责窗口内View的管理，而WindowManager Service 管理来自与不同Acitivity以及系统的的窗口。

**1. Acitivty显示前的准备工作**

在[图解Android - Zygote, System Server 启动分析](http://www.cnblogs.com/samchen2009/p/3294713.html)中我们已经知道，一个新的应用被fork完后，第一个调用的方法就是 ActivityThread的main()，这个函数主要做的事情就是创建一个ActivityThread线程，然后调用loop()开始等待。当收到来自 ActivityManager 的 LAUNCH\_ACTIVITY 消息后，Activity开始了他的显示之旅。下图描绘的是Activity在显示前的准备流程。

[[](https://images0.cnblogs.com/blog/563439/201310/15205324-5f7d9df5d9ed43b990990e086dcdbeba.png)](https://images0.cnblogs.com/blog/563439/201310/15205324-5f7d9df5d9ed43b990990e086dcdbeba.png)

图分为三部分， 右上角是Acitivity应用的初始化。中间部分是Acitivity 与WindowManager Service的交互准备工作，左下角是window显示的开始。本文主要描述后两部分，而Activity的启动会放在[图解Android - Android GUI 系统 (4) - Activity的生命周期](http://www.cnblogs.com/samchen2009/p/3368156.html)里讲解。

1. Activity内部的准备过程，这里面有一个重要对象，ContextImpl 生成，它是Context类的具体实现，它里面封装了应用程序访问系统资源的一些基本API, 比如说，连接某一个服务并获取其IBinder，发送Intent， 获取应用程序的信息，访问数据库等等，在应用看来，它就是整个AndroidSDK的入口。ContextImpl 除了实现函数，里面还维护成员变量，其中有一个mDisplay，代表当前应用输出的显示设备，如果应用没有特别指定，一般指向系统的默认显示输出，比如手机的液晶屏。
2. 在[图解Android - Android GUI 系统 (1) - 概论](http://www.cnblogs.com/samchen2009/p/3364327.html)中我们已经介绍过ViewRootImpl 的地位相当与MVC架构中的C，Controller是连接View和Modal的关键，所以需要首先创建它。当addView(view, param)被调用的时候，一个ViewRoot就被创建出来，addView()的实现如下：
3. public void addView(View view, ViewGroup.LayoutParams params) {
4. mGlobal.addView(view, params, mDisplay, mParentWindow);

}

public void addView(View view, ViewGroup.LayoutParams params,Display display, Window parentWindow){

...

root = new ViewRootImpl(view.getContext(), display);

...

}

这里的参数View是想要添加到WindowManagerService 的“window", 一般一个Activity只需要一个’Window', 所以，Acitivy的默认实现是将DecorView作为”Window" 交给Window Manager Service 进行管理。Params是Layout相关的参数，里面包含有长，宽，边缘尺寸(Margin)等信息，mDisplay就是这个窗口想要输出的Display设备编号，由ContextImpl传递过来。mParentWindow 就是Activity的成员变量mWindow，从最上面的类图可以很容易看出来，对于手机而言，就是一个PhoneWindow对象，对于GoogleTV，就是TVWindow对象。

1. ViewRootImpl 在构造过程成初始化一些重要的成员变量，包括一个Surface对象（注意这是一个空的Surface对象，没有赋给任何有效的值，后面会通过CopyFromParcel来填充），还有mChoreophaer定时器（Singleton对象，每个进程只有一个），与此同时，ViewRootImp通过WindowManagerGlobal创建了一个和WindowManagerService 的通话通道，接下来会利用这条通道做进一步的初始化工作。
2. 还是在addView()里，WindowManagerImpl拿到ViewRoot对象后调用它的setView方法，将view, layout参数交给ViewRootImpl开始接管。在setView()里ViewRootImpl做进一步的初始化工作，包括创建一个InputChannel接收用户按键输入，enable图形硬件加速，请求第一次的Layout等等，这里只介绍跟WindowManagerService 有关系的一件事，就是向WindowManager service 报道，加入到WindowManager的窗口管理队列中。这个函数是 addToDisplay(),
3. int addToDisplay(in IWindow window, //提供给WMS的回调接口
4. in int seq,
5. in windowManager.LayoutParams attrs, // layout参数
6. in int viewVisibility, in int layerStackId, // display ID
7. out Rect outContentInsets, // WMS计算后返回这个View在显示屏上的位置

out InputChannel outInputChannel); // 用户输入通道Handle

addToDisplay() 最终会调到WindowManager Service的addWindow() 接口。

1. addWindow() 里首先生成了一个WindowState对象，它是ViewRootImpl 在WindowManager Service端的代表。在它的构造函数里，WindowState 会生成IWindowId.Stub 对象和DeathRecipient对象来分别监听Focus和窗口死亡的信息，根据用户传进来的Window Type计算出窗口的mBaseLayer，mSubLayer和mLastLayer值，分别对应于主窗口，主窗口上弹出的子窗口（如输入法），以及动画时分别对应的ZOrder值，（在本文后面会具体介绍），生成一个WindowStateAnimation 负责整个Window的动画，并在内部将windowToken, appWindowToken等关联起来。
2. WindowManager Service 调用openInputChannelPair() and RegisterInputChannel(), 创建用于通信的SocketPair , 将其传给InputManagerService, 用于接下来的用户输入事件对应的响应窗口（参考[Android的用户输入处理](http://www.cnblogs.com/samchen2009/p/3368158.html)），
3. 最后，WindowManagerService 调用WindowState的attach()，创建了一个Surface Session 并将Surface Session，WindowSession 还有WindowState 三者关联起来.
4. WindowManager Service 调用 assignLayersLocked()计算所有Window的Z-Order。
5. addToDisplay() 返回，ViewRootImpl 和 WindowManager Service 内部的准备工作就绪。ActivityThread会发送ACTIVITY\_RESUMED消息告诉Activity显示开始。可以是图还没有画，不是吗？对的，此刻Surface还没有真正初始化（我们前面说过ViewRootImpl只是New了一个空的对象，需要有人往里面填东西）。底层存放绘制结果的Buffer也没有创建，但是最多16ms以后这一切就会开始。

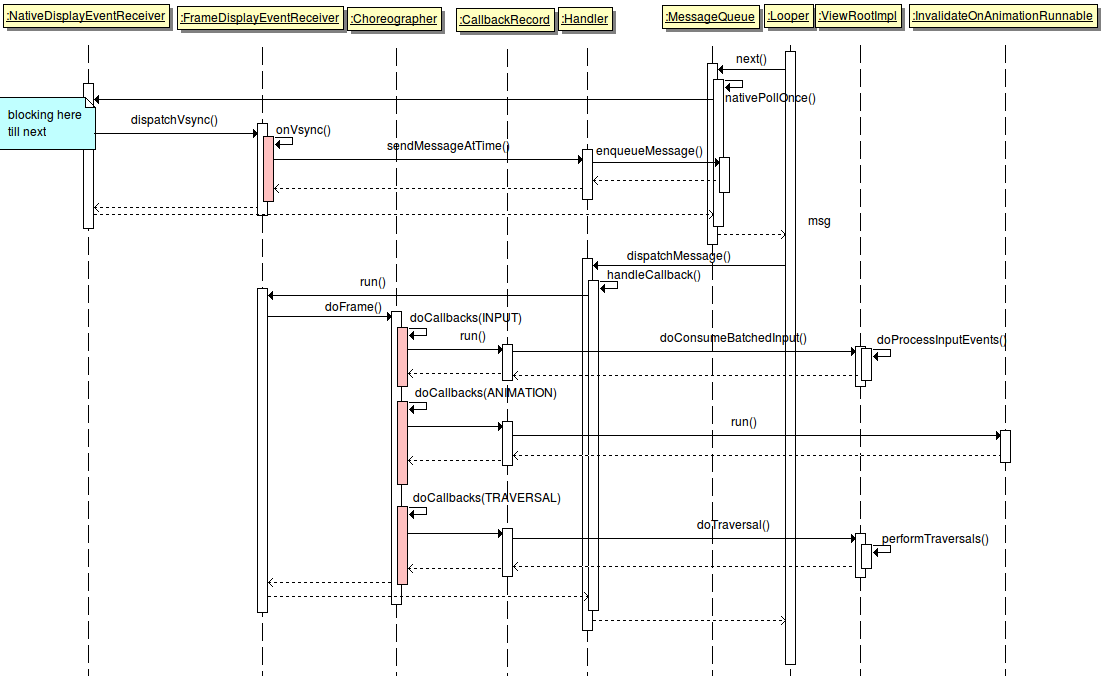
**2. Choreographer 和 Surface的创建**

所有的图像显示输出都是由时钟驱动的，这个驱动信号称为VSYNC。这个名词来源于模拟电视时代，在那个年代，因为带宽的限制，每一帧图像都有分成两次传输，先扫描偶数行（也称偶场）传输，再回到头部扫描奇数行（奇场），扫描之前，发送一个VSYNC同步信号，用于标识这个这是一场的开始。场频，也就是VSYNC 频率决定了帧率（场频/2). 在现在的数字传输中，已经没有了场的概念，但VSYNC这一概念得于保持下来，代表了图像的刷新频率，意味着收到VSYNC信号后，我们必须将新的一帧进行显示。

VSYNC一般由硬件产生，也可以由软件产生（如果够准确的话），Android 中VSYNC来着于HWComposer，接收者没错，就是Choreographer。Choreographer英文意思是编舞者，跳舞很讲究节奏不是吗，必须要踩准点。Choreographer 就是用来帮助Android的动画，输入，还是显示刷新按照固定节奏来完成工作的。看看Chroreographer 和周边的类结构。



从图中我们可以看到， Choreographer 是ViewRootImpl 创建的（Choreographer是一个sigleton类，第一个访问它的ViewRootImpl创建它），它拥有一个Receiver, 用来接收外部传入的Event，它还有一个Callback Queue, 里面存放着若干个CallbackRecord, 还有一个FrameHandler，用来handleMessage, 最后，它还跟Looper有引用关系。再看看下面这张时序图，一切就清楚了，



首先Looper调用loop() 后，线程进入进入睡眠，直到收到一个消息。Looper也支持addFd()方法，这样如果某个fd上发生了IO操作(read/write), 它也会从睡眠中醒来。Choreographer的实现用到了这两种方式，首先他通过某种方式获取到SurfaceFlinger 进程提供的fd，然后将其交给Looper进行监听，只要SurfaceFlinger往这个fd写入VSync事件，looper便会唤醒。Lopper唤醒后，会执行onVsync()时间，这里面没有做太多事情，而是调用Handler接口 sendMessageAtTime() 往消息队列里又送了一个消息。这个消息最终调到了Handler （实际是FrameHandler)的handleCallback来完成上层安排的工作。为什么要绕这么大个圈？为什么不在onVSync里直接handleCallback()? 毕竟onVSync 和 handleCallback() 都在一个线程里。这是因为MessageQueue 不光接收来自SurfaceFlinger 的VSync 事件，还有来自上层的控制消息。VSync的处理是相当频繁的，如果不将VSync信号送人MessageQueue进行排队，MessageQueue里的事件就有可能得不到及时处理，严重的话会导致溢出。当然了，如果因为VSync信号排队而导致处理延迟，这就是设计的问题了，这也是为什么Android文档里反复强调在Activity的onXXX()里不要做太耗时的工作，因为这些回调函数和Choreographer运行在同一个线程里，这个线程就是所谓的UI线程。

言归正传，继续往前，VSync事件最终在doFrame()里调了三次doCallbacks()来完成不同的功能, 分别处理用户输入事件，动画刷新（动画就是定时更新的图片), 最后执行performTraversals()，这个函数里面主要是检查当前窗口当前状态，比如说是否依然可见，尺寸，方向，布局是否发生改变（可能是由前面的用户输入触发的），分别调用performMeasure()， performLayout, performDraw()完成测量，布局和绘制工作。我们会在后面详细学习这三个函数，这里我们主要看一下第一次进入performTraversals的情况，因为第一次会做些初始化的工作，最重要的一件就是如本节标题，创建Surface对象。

回看图2，我们可以看到Surface的创建不是在Activity进程里，而是在WindowManagerService完成的（粉颜色）。当一个Activity第一次显示的时候，Android显示切换动画，因此Surface是在动画的准备过程中创建的，具体发生在类WindowStateAnimator的createSurfaced()函数。它最终创建了一个SurfaceControl 对象。SurfaceControl是Android 4.3 里新引进的类,Google从之前的Surface类里拆出部分接口，变成SurfaceControl，为什么要这样？ 为了让结构更清晰，WindowManagerService 只能对Surface进行控制，但并不更新Surface里的内容，分拆之后，WindowManagerService 只能访问SurfaceControl，它主要控制Surface的创建，销毁，Z-order，透明度，显示或隐藏，等等。而真正的更新者，View会通过Canvas的接口将内容画到Surface上。那View怎么拿到WMService创建的Surface，答案是下面的代码里，surfaceControl 被转换成一个Surface对象，然后传回给ViewRoot, 前面创建的空的Surface现在有了实质内容。Surface通过这种方式被创建出来，Surface对应的Buffer 也相应的在SurfaceFlinger内部通过HAL层模块（GRAlloc)分配并维护在SurfaceFlinger 内部，Canvas() 通过dequeueBuffer（）接口拿到Surface的一个Buffer，绘制完成后通过queueBuffer（）还给SurfaceFlinger进行绘制。

[复制代码](javascript:void(0);)

SurfaceControl surfaceControl = winAnimator.createSurfaceLocked();

if (surfaceControl != null) {

outSurface.copyFrom(surfaceControl);

if (SHOW\_TRANSACTIONS) Slog.i(TAG,

" OUT SURFACE " + outSurface + ": copied");

} else {

outSurface.release();

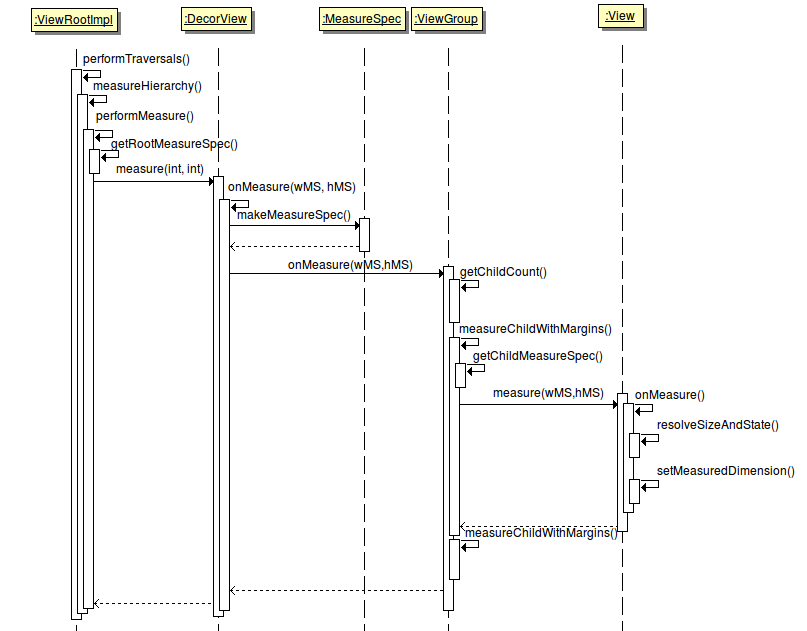
}

[复制代码](javascript:void(0);)

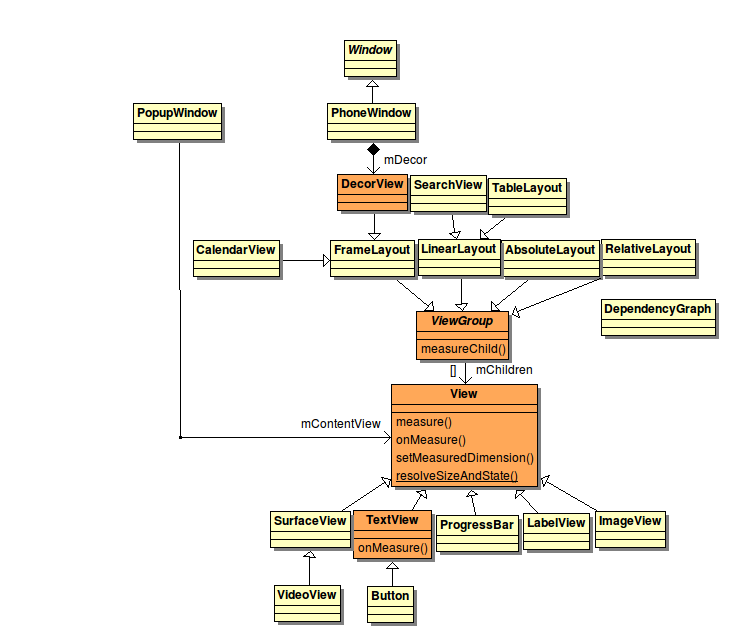
到这里，我们知道了Activity的三大工作，用户输入响应，动画，和绘制都是由一个定时器驱动的，Surface在Activity第一次启动时由WindowManager Service创建。接下来我们具体看一下View是如何画在Surface Buffer上的，而Surface Buffer的显示则交由[图解Android - Android GUI 系统 (3) - Surface Flinger](http://www.cnblogs.com/samchen2009/p/3367499.html) 来讨论。

**3. View的Measure, Layout 和 Draw**

直接从前面提到的performMeasure()函数开始.



 因为递归调用，实际的函数调用栈比这里显示的深得很多，这个函数会从view的结构树顶（DecorView), 一直遍历到叶节点。中间会经过三个基类，DecorView， ViewGroup 和 View， 它们的类结构如下图所示：



所有可见的View(不包括DecorView 和 ViewGroup)都是一个矩形，Measure的目的就是算出这个矩形的尺寸, mMeasuredWidth 和 mMeasuredHeight (注意，这不是最终在屏幕上显示的尺寸），这两个尺寸的计算受其父View的尺寸和类型限制，这些信息存放在 MeasureSpec里。MeasureSpec 里定义了三种constraints,

* [复制代码](javascript:void(0);)
* /\*父View对子View尺寸没有任何要求，其可以设任意尺寸\*/
* public static final int UNSPECIFIED = 0 << MODE\_SHIFT;
* /\* 父View为子View已经指定了大小\*/
* public static final int EXACTLY = 1 << MODE\_SHIFT;
* /\*父View没有指定子View大小，但其不能超过父View的边界 \*/

public static final int AT\_MOST = 2 << MODE\_SHIFT;

[复制代码](javascript:void(0);)

widthMeasureSpec 和 heightMeasureSpec 作为 onMeasure的参数出入，子View根据这两个值计算出自己的尺寸，最终调用 setMeasuredDimension() 更新mMeasuredWidth 和 mMeasuredHeight.

performMeasure() 结束后，所有的View都更新了自己的尺寸，接下来进入performLayout().

performLayout() 的流程和performMeasure基本上一样，可以将上面图中的measure() 和 onMeasure 简单的换成 layout() 和 onLayout()， 也是遍历整课View树，根据之前算出的大小将每个View的位置信息计算出来。这里不做太多描述，我们把重心放到performDraw(), 因为这块最复杂，也是最为重要的一块。

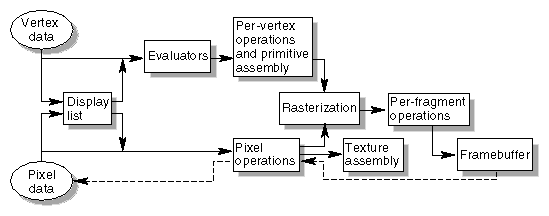
很早就玩过Android手机的同学应该能体会到Android2.3 到 Android 4.0 (其实Android3.0就有了，只是这个版本只在平板上有）的性能的巨大提升，UI界面的滑动效果一下变得顺滑很多，到底是framework的什么改动带来的？我们马上揭晓。。。（这块非本人工作领域，网上相关的资料也很少，所以纯凭个人砖研，欢迎拍砖指正）

**Android Graphics Hardware Acceleration**

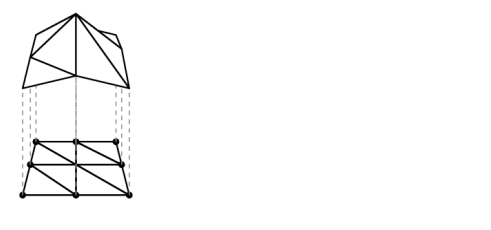
OK, 如标题所述，最根本的原因就是引入了硬件加速, GPU是专门优化图形绘制的硬件单元，很多GPU（至少手机上的）都支持OpenGL，一种开放的跨平台的3D绘图API。Android3.0以前，几乎所有的图形绘制都是由Skia完成，Skia是一个向量绘图库，使用CPU来进行运算, 所以它的performance是一个问题（当然，Skia也可以用GPU进行加速，有人在研究，但好像GPU对向量绘图的提升不像对Opengl那么明显），所以从Android3.0 开始，Google用hwui取代了Skia，准确的说，是推荐取代，因为Opengl的支持不完全，有少量图形api仍由Skia完成，另外还要考虑到兼容性，硬件加速的功能并不是默认打开，需要程序在AndroidManifests.xml 或代码里控制开关。当然，大部分Canvas的基本操作都通过hwui重写了，hwui下面就是Opengl和后面的GPU，这也是为什么Android 4.0的launcher变得异常流畅的缘故。OK，那我们接下来的重点就是要分析HWUI的实现了。

在此之前，简单的介绍一下OpenGL的一些概念，否则很难理解。要想深入理解Opengl，请必读经典的红包书：http://www.glprogramming.com/red/

Opengl说白了，就是一组图形绘制的API。 这些API都是一些非常基本的命令，通过它，你可以构造出非常复杂的图形和动画，同时，它又是跟硬件细节无关的，所以无需改动就可以运行在不同的硬件平台上（前提是硬件支持所需特性）。OpenGL的输入是最基本几何元素(*geometric primitives),*点(points), 线(lines), 多边形(polygons), 以及bitmap和pixle data, 他的输出是一个或两个Framebuffer(真3D立体). 输入到输出的流程(rendering pipeline)如下图所示：

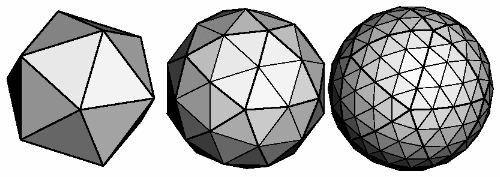


 这里有太多的概念，我们只描述跟本文相关的几个:

*vertex data*所有的几何元素（点线面）都可以用点(vertics)来描述, 每个点都对应三维空间中的一个坐标(x,y,z), 如下图所示，改变若干点的位置，我们便可以构造出一个立体的图形。       


*Triangles*

OpenGL只能画非凹（nonconvex)的多边形，可是现实世界中存在太多的凹性的物体，怎么办呢？通过连线可以将凹的物体分成若干个三角形，三角形永远都是凸（convex)的。同时三角形还有一个特性，三个点可以唯一确定一个平面，所以用尽可能多的三角形就可以逼近现实世界中复杂的曲线表面，比如下图的例子，三角形的数目越多，球体的表示就越逼真。这也是为什么我们经常看到显卡的性能评测都以三角形的生成和处理作为一个非常重要的指标。



*Display List*

所有的Vertex和Pixel信息均可以存在Display List 里面，用于后续处理，换句话说，Display List 就是OpenGL命令的缓存。Display List的使用对OpenGL的性能提升有很大帮助。这个很容易理解，想象一个复杂的物体，需要大量的OpenGL命令来描绘，如果画一次都需要重新调用OpenGL API，并把它转换成Vertex data，显然是很低效的，如果把他们缓存在Display List里，需要重绘的时候，发一个个命令通知OpenGL直接从Display List 读取缓存的Vertex Data，那势必会快很多，如果考虑到Opengl是基于C/S架构，可以支持远程Client，这个提升就更大了。Display也可以缓存BitMap 或 Image, 举个例子，假设要显示一篇文章，里面有很多重复的字符，如果每个字符都去字库读取它的位图，然后告诉Opengl去画，那显然是很慢的。但如果将整个字库放到Display List里，显示字符时候只需要告诉Opengl这个字符的偏移量，OpenGL直接访问Display List，那就高效多了。

*Pixel Data*

Pixle data 包括位图(bitmap), Image, 和任何用于绘制的Pixel数据(比如Fonts)。通常是以矩阵的形式存放在内存当中。通过Pxiel data， 我们避免大量的图形绘制命令。同时通过现实世界中获取的纹理图片，可以将最终的物体渲染得更逼真。比如说画一堵墙，如果没有pixel data，我们需要将每块砖头都画出来，也就是说需要大量的Vertex。可是如果通过一张现实生活中拍摄的砖墙的图片，只需要4个点画出一个大矩形，然后上面贴上纹理，显然，速度和效果都要好得多。

*FrameBuffer*

Framebuffer就是Opengl用来存储结果的buffer。Opengl的frameBuffer类型有几种。Front Buffer 和 Back Buffer, 分别用于显示和绘制，两者通过swapBuffer 进行交换。Left Buffer 和 Right buffer, 用于真立体（需要带眼镜的那种) 图像的左右眼Buffer，Stencil buffer, 用于禁止在某些区域上进行绘制，想像一下如果在一件T恤上印上图案，你是不是需要一个镂空的纸板？这个纸板就是stencil buffer.

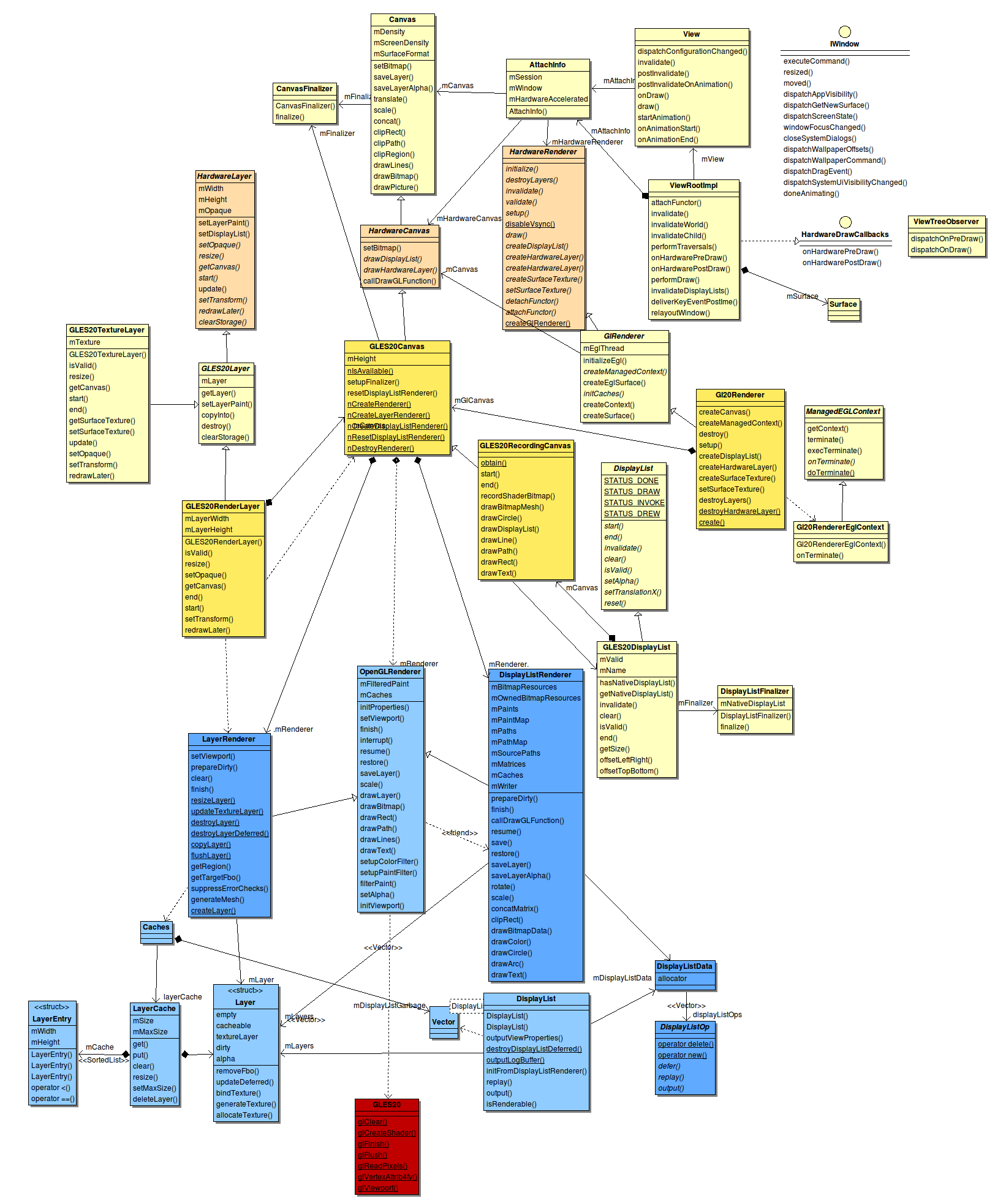
OK， 对Opengl rendering pipeline简单介绍到此，有兴趣的同学可以阅读opengl的红包书或运行一些简单的例子来深入理解Opengl。回到主题，仅仅使用Opengl 和 GPU 取代Skia 就能够大幅提升性能？答案当然不是，性能的优化很大程度上取决于应用，应用必须正确的使用Opengl命令才能发挥其最大效能。Android从pipeline 角度提供了两种机制来提升性能，一个就是我们刚才说到的Display List，另一个叫 Hardware Layer, 其实就是缓存的FrameBuffer, 比如说Android的墙纸，一般来说，他是不会发生变化的，因此我们可以将它缓存在Hardware Layer里，这张就不需要每次进行拷贝和重绘，从而大幅提升性能。

说白了，优化图形性能的核心在于 1）用硬件来减少CPU的参与，加速图形计算。 2）从软件角度，通过Display List 和 Hardware Layer, 将已经完成的工作尽可能的缓存起来，只做必须要做的事情，尽可能的减少运算量。

接下来看实现吧。

**Canvas, Renderer, DisplayList, HardwareLayer 实现**

这块代码相当的复杂，花了两天时间才把下面的图整理出来，但还是没有把细节完全吃透，简单的介绍一下框架和流程吧，如果有需要大家可以下来细看代码。

[](https://images0.cnblogs.com/blog/563439/201310/24000733-366142f7596e4d63b407e3d59caffdf2.png)

图中上半部为Java 代码，下半部为Native层。先介绍里面出现的一些概念：

***Canvas***

*C*anvas是Java层独有的概念，它为View提供了大部分图形绘制的接口。这个类主要用于纯软件的绘制，硬件加速的图形绘制则由HardwareCanvas取代。

***HardwareCanvas，GLES20Canvas, GLES20RecordingCanvas***

hardwareCanvas是一个抽象类，如果系统属性和应用程序指定使用硬件加速（现已成为默认），它将会被View（通过AttachInfo,如 下图所示) 引用来完成所有的图形绘制工作。GLES20Canvas 则是hardwareCanvas的实现，但它也只是一层封装而已，真正的实现在Native 层，通过jni (andriod\_view\_gles20Canvas.cpp)接口来访问底层的Renderer, 进而执行OpenGL的命令。 此外，GLES20Canvas还提供了一些静态接口，用于创建各类Renderer对象。

GLES20RecordingCanvas 继承GLES20Canvas, 通过它调用的OpenGL命令将会存储在DisplayList里面，而不会立即执行。

***HardwareRenderer, GLRender, GL20Renderer***

这三个类都是Java的Wrapper类，通过访问各种Canvas来控制绘制流程。详见下面两张时序图。

***OpenGLRenderer, DisplayListRenderer, HardwareLayerRenderer***

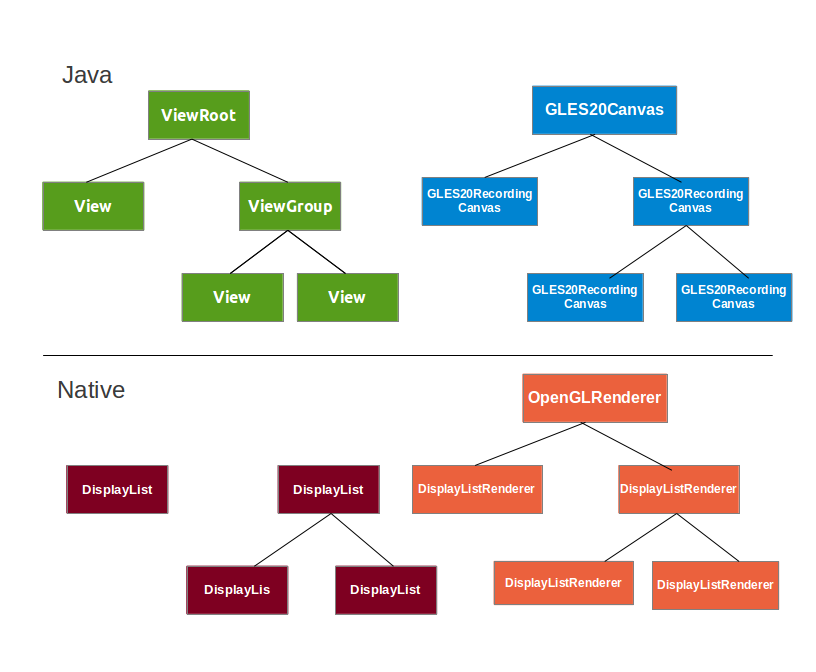
Java的HardwareCanvas 和 HardwareRenderer在底层的对应实现。OpenGLRenderer是基类，只有它直接访问底层OpenGL库。DisplayListRenderer 将View通过GLES20Canvas传过来的OpenGL 命令存在OpenGL的DisplayList中。而HardwareLayerRenderer 管理HardwareLayer的资源。

***GLES20***就是OpenGL ES 2.0 的API。它的实现一般由GPU的设计厂家提供，可以在设备的/system/lib/egl/ 找到它的so，名字为 libGLES\_xxx.so, xxx 就是特定设备的代号，比如说，libGLES\_gc.so 就是Vivante公司的GPU实现，libGLESv2\_mali.so 就是ARM公司提供的Mali GPU的实现。它也可以由软件实现，比如说 libGLES\_android.so, 是Google提供的软件实现。

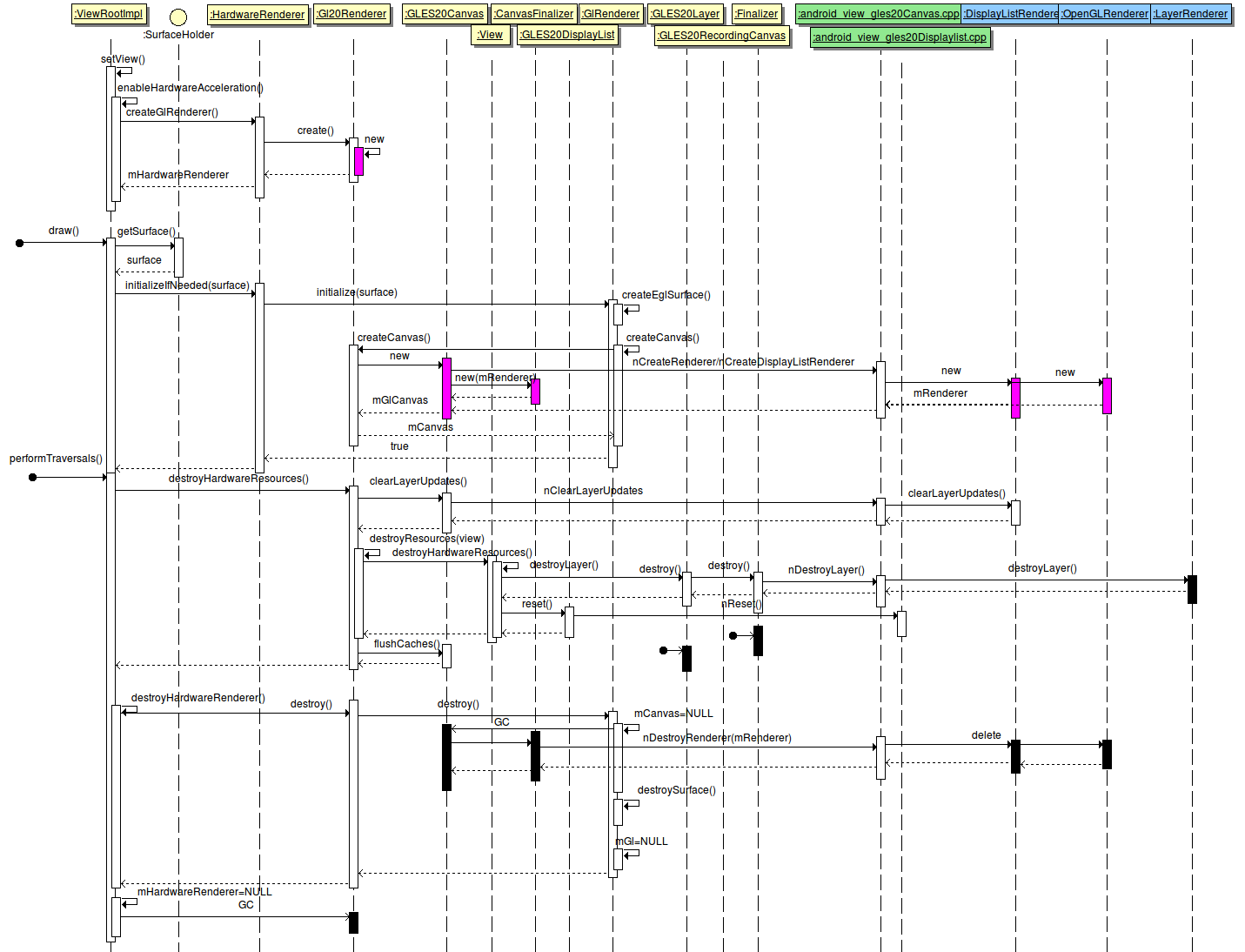
***EGL***虽然对于上层应用来说OpenGL接口是跨平台的，但是它的底层(GPU)实现和平台(SoC)是紧密相关的，于是OpenGL组织定义一套接口用来访问平台本地的窗口系统（native platform window system)，这套接口就是EGL，比如说 eglCreateDisplay(), eglCreateSurface(), eglSwapBuffer()等等。EGL的实现一般明白libEGL.so, 放在/system/lib/egl/ 下面。

View, Canvas, Renderer, DisplayList, HardwareLayer 的关系如下图所示：

每个**View**都对应一个**DisplayList**, 在Native层代码里管理。每个**View**通过**GLESRecordingCanvas** 以及Native层对应的**DisplayRenderer** 将OpenGL命令存入DisplayList.最后**View** 通过**GLES20Canvas** 通知**OpenGLRenderer** 执行这些DisplayList 里面的OpenGL 命令。



 他们的生命周期如下图所示 （粉红代表 New， 黑色代表 Delete, 黄色代表Java类，蓝色代表C++, 绿色代表JNI).

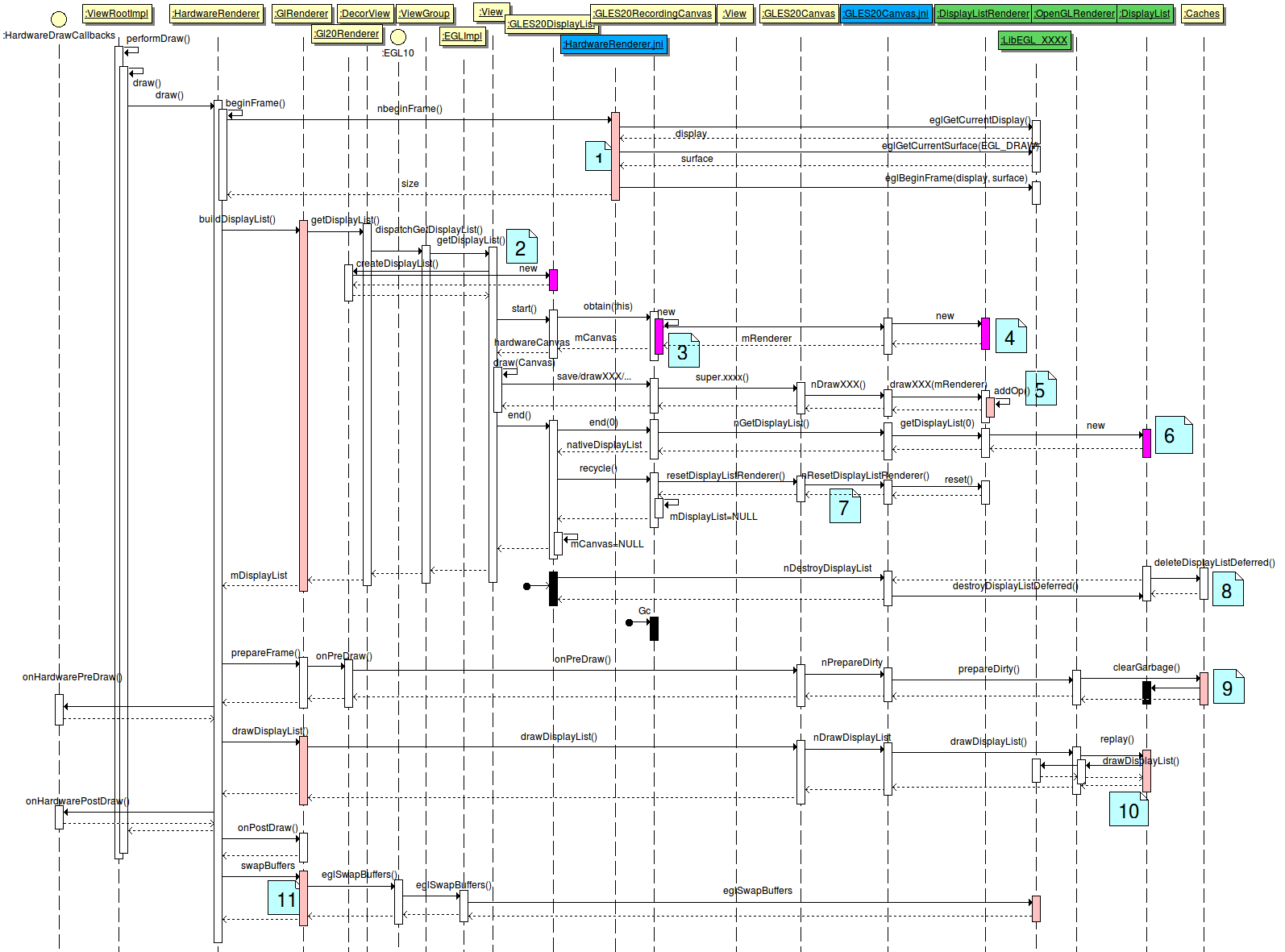
[](https://images0.cnblogs.com/blog/563439/201310/24004821-7d00f74a681646ceb87b858e06409376.png)

1. 如果系统支持硬件加速，ViewRootImpl首先创建一个GL20Renderer, 存在成员变量 mHardwareRenderer 里。
2. Surafce是绘画的基础，如果不存在，HardwareRenderer会调用GLRenderer->createEglSurface() 创建一个新的Surface。
3. Surface创建好后，接着生成Canvas，因为大部分应用程序不会直接操作Surface。
4. 在Canvas的构造函数里，会依次创建Native层对应的OpenGLRenderer对象，它会直接访问库 libGLES\_xxx提供的OpenGL ES API，来负责整个View树的绘制工作。
5. 与此同时，一个CanvasFinalizer 成员对象也会被创建，它保存刚刚创建出来的OpenGLRenderer 指针，当它的finalizer（）函数被调用是，负责将其销毁，回收资源。
6. 在运行过程中，如果窗口死掉或者不在可见，ViewRootImpl 会调用DestroyHardwareResource() 来释放资源。这里会最终将底层创建的Hardware Layer回收。
7. 同时Java端的GLES20Layer 对象也会因为被赋值 NULL 被GC在将来回收。
8. 接下来，ViewRootImpl调用 destroyHardwareRenderer() 将之前创建的Native Renderer（DisplayListRenderer，OpenGLRenderer)依次回收。
9. 最后将Java 层的mHardwareRenderer 赋空，GC将会回收最开始创建的GL20Renderer对象。支持，一个View树的生命周期完成，所有资源清楚干净。

等等！好像少了点什么，怎么没有DisplayList? 前面不是说它是性能优化的帮手之一吗？对了，上面只介绍了绘制的开始和结尾，在View的生命周期中，还有最重要的一步，Draw 还没有被介绍，DisplayList 相关的操作就是在Draw()里面完成的。

**Draw 流程**

  绕了好大一圈，终于回到最初的话题，Android是怎样将View画出来的？ 让我们按照图中的序号一一进行讲解。（黄色：Java, 绿色：C++，蓝色：JNI，粉色：New， 黑色：Delete).

[](https://images0.cnblogs.com/blog/563439/201310/24004912-eec51492576e4b5aa2f39c6a7f5f91a7.png)

1. 首先，ViewRootImpl直接访问的HardwareRenderer 对象，首先在BeginFrame() 里获取EGLDisplay(用于显示) 和 初始化一个EGLSurface，OpenGL将在这个Surface上进行绘图。关于EGLDisplay 和 EGLSurface 将在 [图解Android - Android GUI 系统 (3) - Surface Flinger](http://www.cnblogs.com/samchen2009/p/3367499.html) 里详细描述。
2. 我们前面说过，Android 4.0 带来的图形性能的提升，有很大程度是DisplayList 带来的，所以，HardwareRenderer接下来就通过buildDisplayList() 创建整个View树的DisplayList. 最开始，GL20Renderer为View生成了一个GLES20DisplayList, 这是一个影子类，没有实际用途，主要用来管理Native层的DisplayList的销毁。
3. View 调用刚刚生成的GLES20DisplayList的 start() 方法，真正开始构建DisplayList的上下文。obtain() 函数里会判断是否mCanvas 已经存在，如果没有则New 一个新的GLES20RecordingCanvas对象。
4. 创建与GLES20RecordingCanvas 一一对应的Native层的 DisplayListRenderer。
5. 3，4是一个递归的过程，直到所有的View的DisplayList上下文生成，View才真正开始Draw(), 这里，OpenGL命令(Op)不会被立即执行，而是被存储到刚刚生成的DisplayListRenderer里。
6. 接着View调用GLESDisplayList的 end() 方法，这里Native层的DisplayList 对象才真正被创建出来，Java 和 Native 的 DisplayList 对象一一对应起来。
7. end() 方法最后，调用GLES20RecordingCanvas.recycle() 方法，首先将Native的DisplayListRender 进行重新初始化，然后将刚才创建出来的临时对象赋NULL值（GLES20RecordingCanvas 和 GLES20DisplayList), 因为它们的使命已经完成，将View的OpenGL命令存储在Native层的DisplayList对象里。
8. GC() 被调用后，GLES20RecordingCanvas 和 GLES20DisplayList 被释放，相应的Finalizer 对象方法会被调用，进而调用Natice层的deleteDisplayListDefered()， 将之前不用的DisplayList 送入 Caches的Gabage 队列中，等待回收。（这是在Native层实现的类似Java GC的机制）
9. 到此，所有的DisplayList 上下文准备就绪。进入preDraw 状态，在GL20Renderer的 onPreDraw() 方法里，最终调用到底层的clearGarbage将上一次绘图操作的DisplayList在底层释放。
10. HardwareRenderer调用 GLES20Canvas 的drawDisplayList(), 通知 Native层的 OpenGLRenderer，其最终调用每个DisplayList的Replay() 方法执行OpenGL命令。
11. 所有OpenGL命令执行完后, 图形被绘制到第一步生成的EGLSurface, hardwareRender 调用 GLES20Canvas 的 eglSwapBuffers() 交换Buffer，交由SurfaceFlinger 在下一个VSync到来时进行显示。

**Hardware Layer**

即便是使用了DisplayList, 对于复杂的图形，仍然需要执行大量的OpenGL命令，如果需要对这一部分进行优化，就需要使用到 HardwareLayer对绘制的图形进行缓存，如果图形不发生任何变化，就不需要执行任何OpenGL命令，而是将之前缓存在GPU内存的Buffer 直接与其他View进行合成，从而大大的提高性能。这些存储在GPU内部的Buffer就称为 Hardware Layer。除了Hardware Layer, Android 还支持Software Layer，和Hardware Layer 不同之处在于，它不存在于GPU内部，而是存在CPU的内存里，因此它不经过前面所说的 Hardware Render Pipeline， 而是走Android最初的软件Render pipeline。不管是Hardware Layer 还是 Software Layer， 在Draw() 内部均称为Cache，只要有Cache的存在，相对应的View将不用重绘，而是使用已有的Cache。Hardware Layer， Software Layer 和 Display List 是互斥的，同时只能有一种方法生效（当然，Hardware Layer的第一次绘制还是通过Display List 完成），下表总结了它们的差别, 从中可以看到，Hardware Layer 对性能的提升是最大的，唯一的问题是占用GPU的内存（这也是为什么显卡的内存变得越来越大的原因之一），所以一般来说，Hardware Layer使用在那些图片较为复杂，但不经常改变，有动画操作或与其他窗口有合成的场景，比如说WallPaper, Animation 等等。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Enabled if** | **GPU accelerated?** | **Cached in** | **Performance(from Google I/O 2001)** | **Usage** |
| **Display List** | Hardware Accelerated = True | Y | GPU DisplayList | 2.1 | Complex View |
| **Hardware Layer** | LayerType = HARDWARE | Y | GPU Memory | 0.009 | Complex View, Color Filter(颜色过滤）, Alpha blending （透明度设置）, etc. |
| **Software Layer** | LayerType = SOFTWARE | N | CPU Memory | 10.3 | No Hardware Accelerated, Color filter, Alpha blending, etc. |

**重绘 - Invaliate**

前面介绍了View的第一次绘制的过程。但是一个View在运行中终究是要发生变化的，比如说，用户在TextView的文字发生了改变，或者动画导致View的尺寸发生变化，再或者说一个对话框弹出然后又消失，被遮挡的部分重新露了出来，这些都需要对View进行重绘。在介绍重绘之前，我们先了解一下View内部的一些Flag 定义。

|  |  |  |  |
| --- | --- | --- | --- |
| **Flags** | **== 1** | **Set at** | **Clear at** |
| PFFLAG\_HAS\_BOUNDS | 1: View has size and Position set. 0: setFrame() was not called yet | View::setFrame() |  |
| PFFLAG\_DRAWN | 1: Has been drawn, only after which, invalidate() is valid. 0: not drawn, need to draw() later to show it. | ViewGroup::addViewInLayout ViewGroup::attachViewToParent View::draw() View::buildDrawingCache() View::getHardwareLayer() View::invalidate() View::setLeft/Right/Top/Bottom() | View::invalidate() |
| PFFLAG\_DRAWING\_CACHE\_VALID | 1: Has DisplayList / Hardware Layer / Software Layer　　 0: No Cache, using Software rendering path. | View::GetHardwareLayer() View::GetDisplayList() | View::invalidate(true) View::invalidate(rect) View::invalidateChild() |
| PFFLAG\_INVALIDATED | View is specifically invalidated, not just dirty(child for instance).  DisplayList will be recreated if set. | ViewGroup::addViewInLayout ViewGroup::attachViewToParent View::force/requestLayout() View::invalidate() | View::draw() |
| PFLAG\_DIRTY | Set to indicate that the view need to be redrawn.  (use displaylist cache if PFFLAG\_INVALIDATED flag is false) | View::invalidate() | ViewGroup::addViewInLayout ViewGroup::attachViewToParent View::draw() View::buildDrawingCache() View::getHardwareLayer() View::invalidate() View::setLeft/Right/Top/Bottom() View::getDisplayList() |
| PFLAG\_DIRTY\_OPAQUE | DIRTY because of OPAQUE (hidden by others). | View::invalidateChild() | ？ |

从上表可以看出，View通过内部这些Flag来控制重绘。基本上重绘分两种情况，一种是需要重新生成DisplayList， 另外一种是使用之前已有的Cache，包括DisplayList或者是Hardware Layer。使用哪种重绘方式由当前View的Flags，以及应用程序传入的参数决定。控制它的就是一组Invalidate() 函数。

[复制代码](javascript:void(0);)

void invalidate(boolean invalidateCache) {

if (skipInvalidate()) { //如果View不可见，并且不再动画退出过程中（fade out),将不执行Invalidate().

return;

}

if ((mPrivateFlags & (PFLAG\_DRAWN | PFLAG\_HAS\_BOUNDS)) == (PFLAG\_DRAWN | PFLAG\_HAS\_BOUNDS) || //DRAWN -> 已经被Draw()过

(invalidateCache && (mPrivateFlags & PFLAG\_DRAWING\_CACHE\_VALID) == PFLAG\_DRAWING\_CACHE\_VALID) || //有Cache，且被要求重新刷新Cache

(mPrivateFlags & PFLAG\_INVALIDATED) != PFLAG\_INVALIDATED || isOpaque() != mLastIsOpaque) //没有正在Invalidate()中

{

mLastIsOpaque = isOpaque();

mPrivateFlags &= ~PFLAG\_DRAWN;

mPrivateFlags |= PFLAG\_DIRTY;

if (invalidateCache) {

mPrivateFlags |= PFLAG\_INVALIDATED;

mPrivateFlags &= ~PFLAG\_DRAWING\_CACHE\_VALID; //标记将来清除Cache，如果为false，则有系统根据Dirty Region决定是否需要重新生成DisplayList。

}

final AttachInfo ai = mAttachInfo;

final ViewParent p = mParent;

if (!HardwareRenderer.RENDER\_DIRTY\_REGIONS) { //系统不支持Dirty Region，必须重绘整个区域, 基本不会进去

}

if (p != null && ai != null) {

final Rect r = ai.mTmpInvalRect;

r.set(0, 0, mRight - mLeft, mBottom - mTop);

p.invalidateChild(this, r); //通知兄弟view（有共同的ViewParent(ViewGroup 或者 ViewRoot)进行 Invalidate.

}

}

}

[复制代码](javascript:void(0);)

假如所有的条件都支持重绘，便会调用到ViewParent的invalidateChild()方法。（ViewParent是一个接口类，它的实现类是ViewGroup 和 ViewRootImpl。）这个方法会从当前View开始，向上遍历到ViewRoot 或者 到某个ViewGroup的区域与当前View的Dirty区域没有重叠为止。途中的每个ViewGroup都会被标记上Dirty。在接下来VSYNC的performDraw（）里，ViewRootImpl 会遍历所有标记Dirty的ViewGroup，然后找到里面标记Dirty的View，只有这些View的DisplayList 被重建，而其他实际上没有变化的View（虽然它们在同一个ViewGroup里面），如果没有Hardware Layer, 只需重新执行对应Display List 里面的OpenGL 命令。通过这种方式，Android只重绘需要重绘的View，从软件层面将GPU的输入最小化，从而优化图形性能。

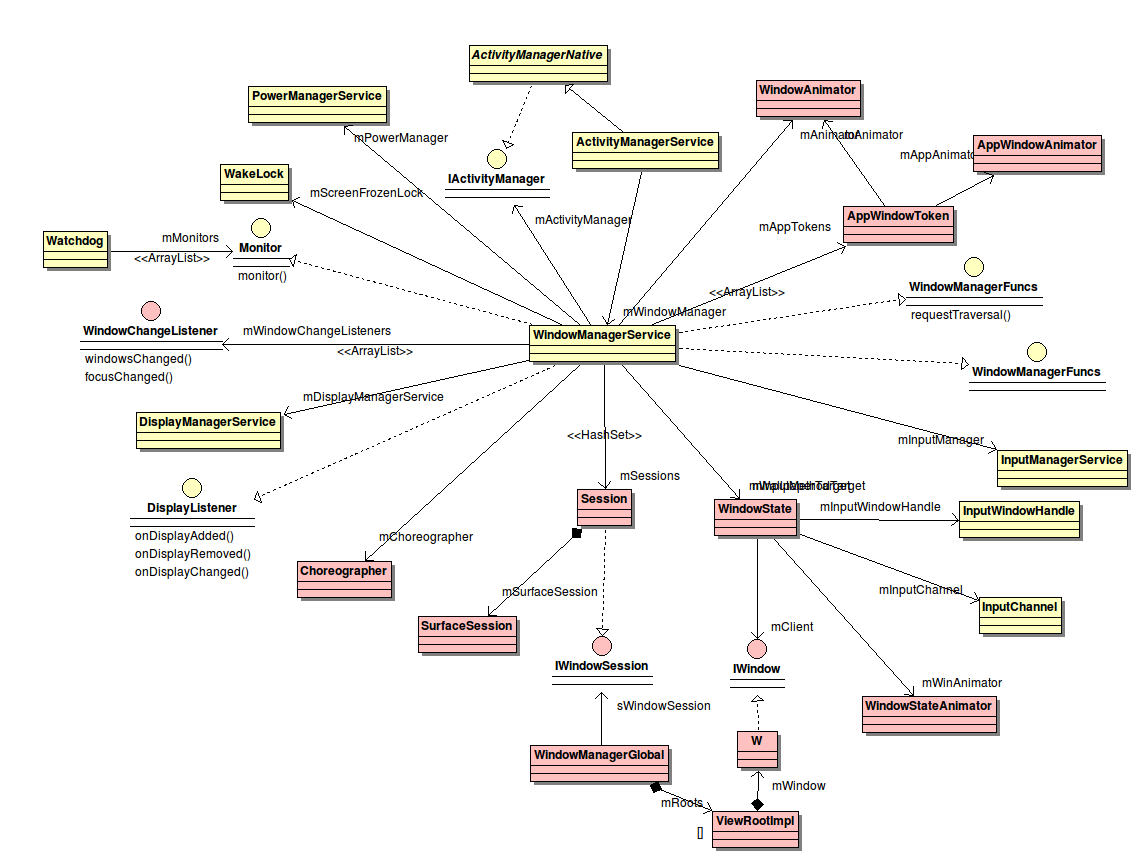
**4. Windows 的管理**

到此，我们已经了解了一个Acitivty(Window)是如何画出来的，让我们在简要重温一下这个过程：

1. Acitivity创建， ViewRootImpl将窗口注册到WindowManager Service，WindowManager Service 通过SurfaceFlinger 的接口创建了一个Surface Session用于接下来的Surface 管理工作。
2. 由Surface Flinger 传上来的VSYNC事件到来，Choreographer 会运行ViewRootImpl 注册的Callback函数， 这个函数会最终调用 performTraversal 遍历View树里的每个View， 在第一个VSYNC里，WindowManager Service 会创建一个SurafceControll 对象，ViewRootImpl 根据Parcel返回的该对象生成了Window对应的Surface对象，通过这个对象，Canvas 可以要求Sruface Flinger 分配OpenGL绘图用的Buffer。
3. View树里的每个View 会根据需要依次执行 measure()，layout() 和 draw() 操作。Android 在3.0之后引入了硬件加速机制，为每个View生成DisplayList，并根据需要在GPU内部生成Hardware Layer，从而充分利用GPU的功能提升图形绘制速度。
4. 当某个View发生变化，它会调用invalidate() 请求重绘，这个函数从当前View 出发，向上遍历找到View Tree中所有Dirty的 View 和 ViewGroup， 根据需要重新生成DisplayList, 并在drawDisplayList() 函数里执行OpenGL命令将其绘制在某个Surface Buffer上。
5. 最后，ViewRootImpl 调用 eglSwapBuffer 通知OpenGL 将绘制的Buffer 在下一个VSync点进行显示。

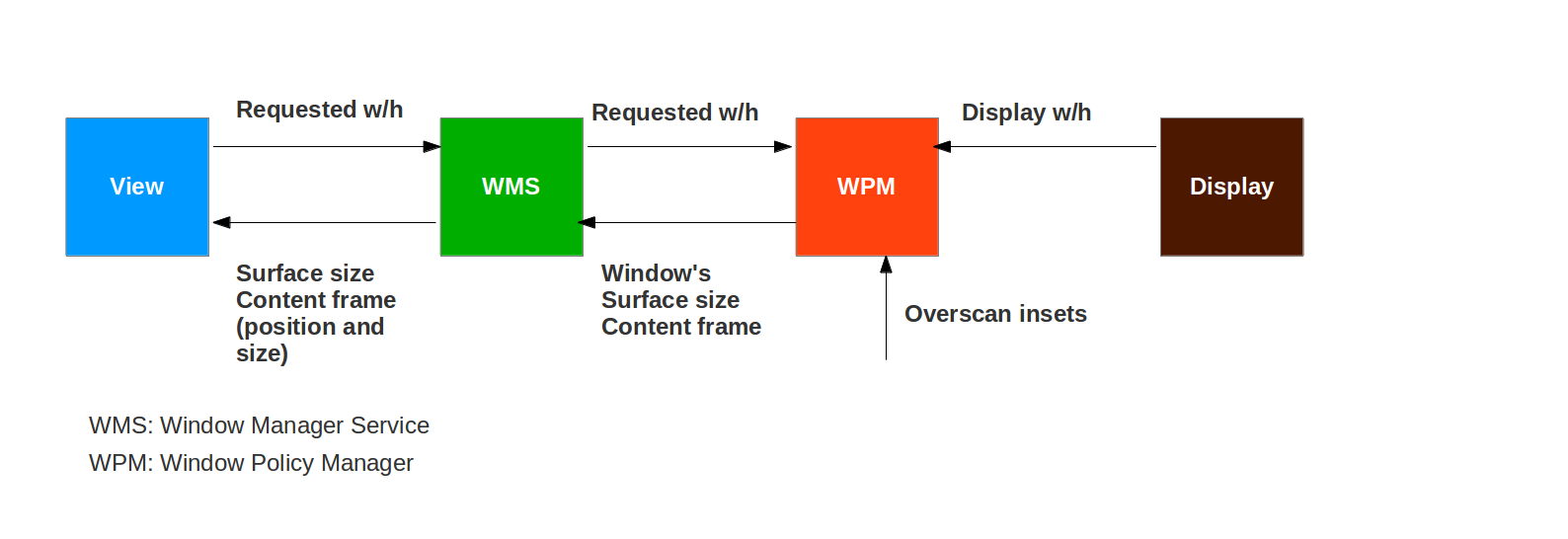
注意的是，上面讨论的只是一个窗口的流程，而Android是个多窗口的系统，窗口之间可能会有重叠，窗口切换会有动画产生，窗口的显示和隐藏都有可能会导致资源的分配和释放，这一切需要有一个全局的服务进行统一的管理，这个服务就是我们大名鼎鼎的Window Manager Service (简写 WMS).

其实Window Manager Service 的工作不仅仅是管理窗口，还会跟很多其他服务打交道，如 InputManager Service， AcitivityManager Service 等等，但本章只讨论它在Window Manager 方面的工作，下图中红色标记部分。



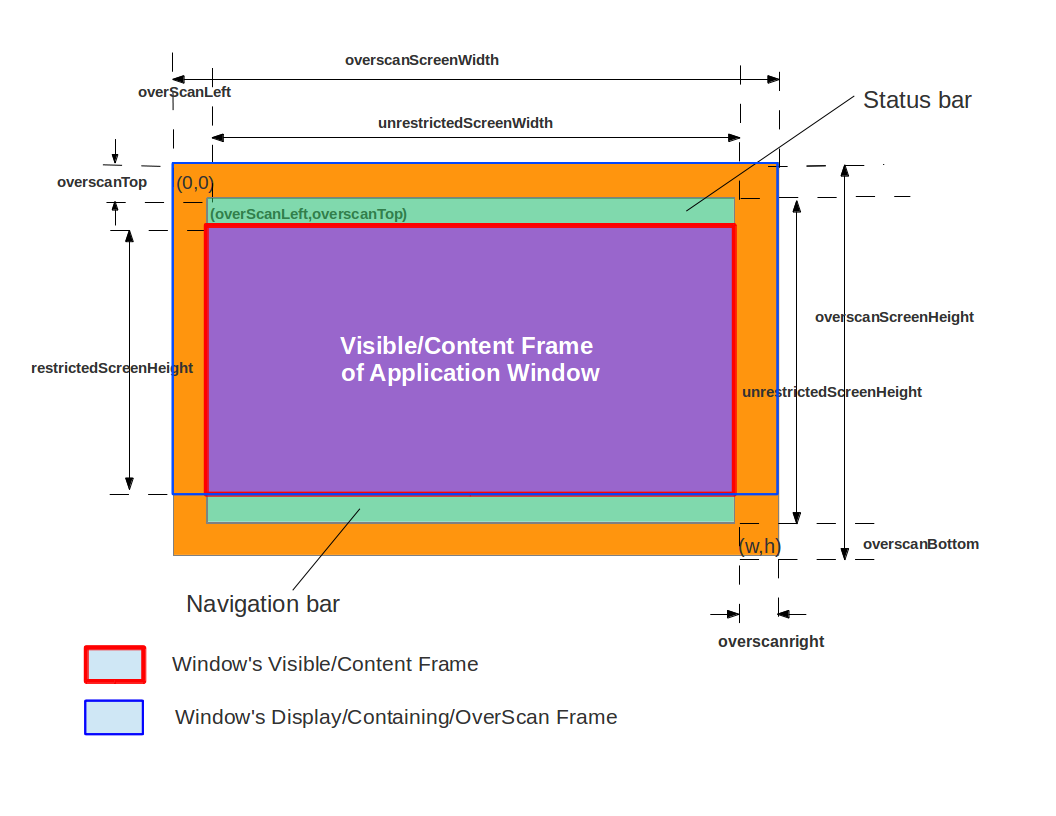
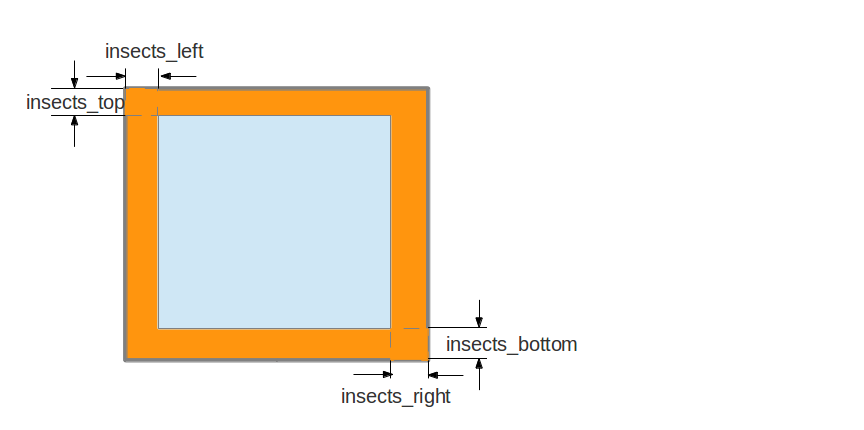
**Layout**

 首先来看Layout。Layout 是Window Manager Service 重要工作之一，它的流程如下图所示：



* 每个View将期望窗口尺寸交给WMS（WindowManager Service).
* WMS 将所有的窗口大小以及当前的Overscan区域传给WPM （WindowPolicy Manager).
* WPM根据用户配置确定每个Window在最终Display输出上的位置以及需要分配的Surface大小。
* 返回这些信息给每个View，他们将在给会的区域空间里绘图。

Android里定义了很多区域,如下图所示

*Overscan*:   
    Overscan 是电视特有的概念，上图中黄色部分就是Overscan区域，指的是电视机屏幕四周某些不可见的区域（因为电视特性，这部分区域的buffer内容显示时被丢弃），也意味着如果窗口的某些内容画在这个区域里，它在某些电视上就会看不到。为了避免这种情况发生，通常要求UI不要画在屏幕的边角上，而是预留一定的空间。因为Overscan的区域大小随着电视不 同而不同，它一般由终端用户通过UI指定，（比如说GoogleTV里就有确定Overscan大小的应用）。

*OverscanScreen, Screen:*OverscanScreen 是包含Overscan区域的屏幕大小,而Screen则为去除Overscan区域后的屏幕区域, OverscanScreen > Screen.

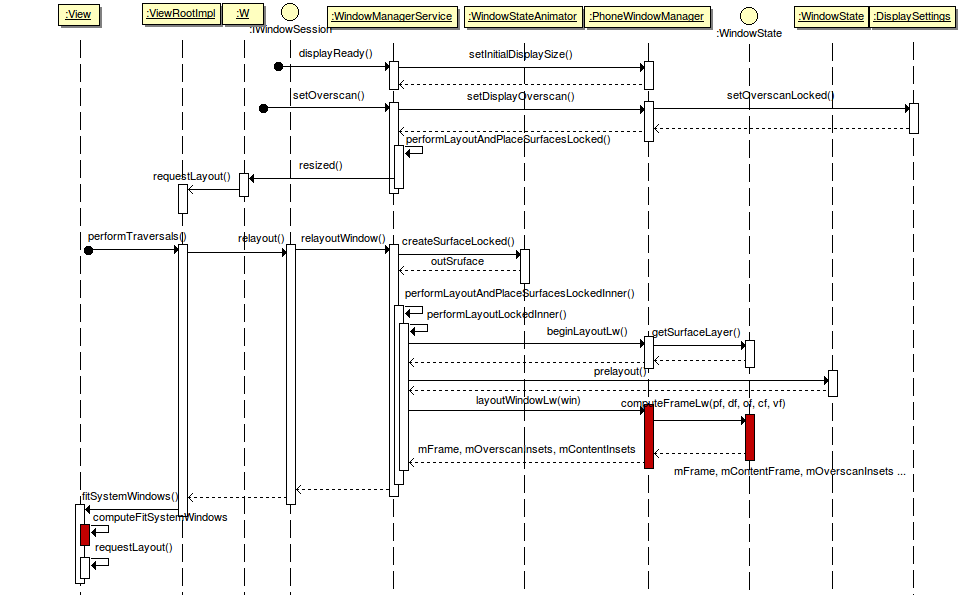
*Restricted and Unrestricted:*    某些区域是被系统保留的，比如说手机屏幕上方的状态栏(如图纸绿色区域）和下方的导航栏，根据是否包括这些预留的区域，Android把区域分为Unrestricted Area 和 Resctrited Aread, 前者包括这部分预留区域，后者则不包含, Unrestricted area > Rectricted area。

*mFrame, mDisplayFrame, mContainingFrame*Frame指的是一片内存区域, 对应于屏幕上的一块矩形区域. mFrame的大小就是Surface的大小, 如上上图中的蓝色区域. mDisplayFrame 和 mContainingFrame 一般和mFrame 大小一致. mXXX 是Window(ViewRootImpl, Windowstate) 里面定义的成员变量.

*mContentFrame, mVisibleFrame*一个Surface的所有内容不一定在屏幕上都得到显示, 与Overscan重叠的部分会被截掉, 系统的其他窗口也会遮挡掉部分区域 (比如短信窗口，ContentFrame是800x600(没有Status Bar), 但当输入法窗口弹出是，变成了800x352), 剩下的区域称为Visible Frame, UI内容只有画在这个区域里才能确保可见. 所以也称为Content Frame. mXXX也是Window(ViewRootImpl, WindowState) 里面定义的成员变量.

*Insects*insets的定义如上图所示*,*用了表示某个Frame的边缘大小.

Layout 在WMS 内部的时序如下图所示，外部调整Overscan参数或View内部主动调用requestLayout() 都会触发WMS的重新layout，layout完成后，WMS会通过IWindow的resized()接口通知ViewRoot, 最终会调用requestLayout(), 并在下一个VSYNC 事件到来时更新。

。

 计算Layout主要有图中三个红色的函数完成，它们代码很多，涉及到很多计算，但只要对着我们上面给的三个图来看，不难看出它的意思，本文将不详细深入。

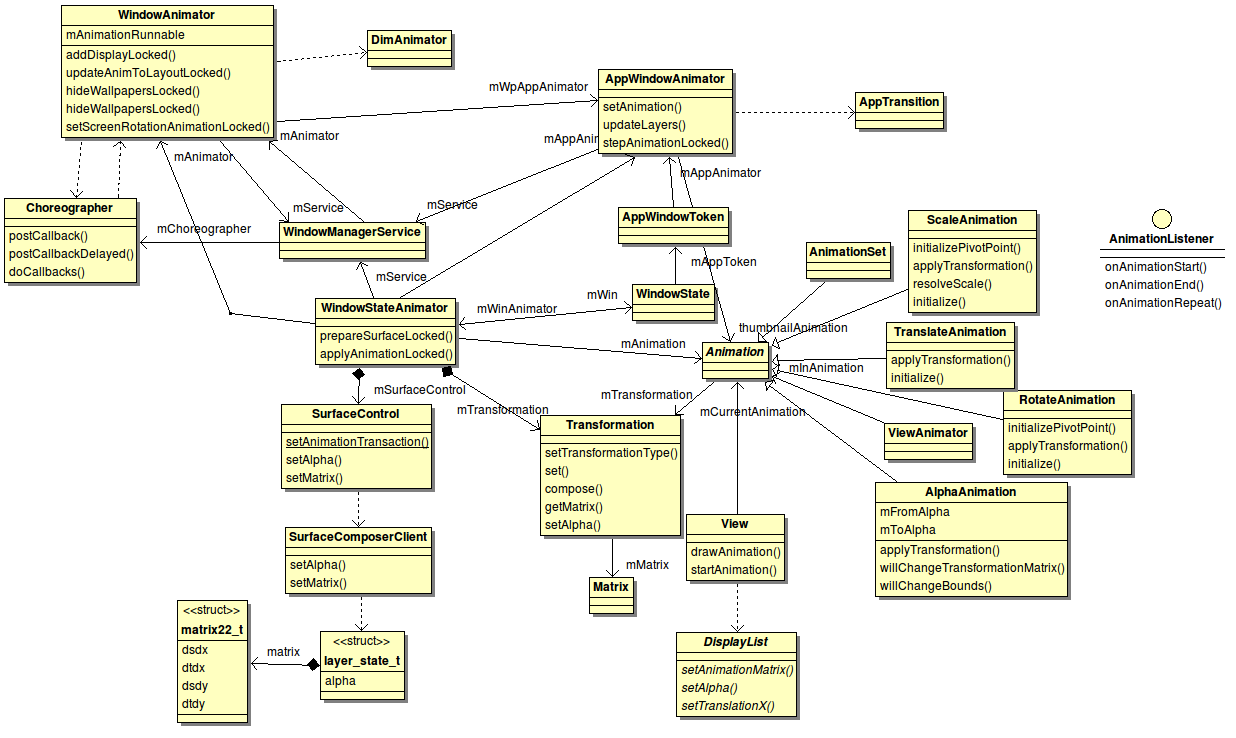
**Animation**

Animation的原理很简单，就是定时重绘图形。下面的类图中给出了Android跟Animation相关的类。

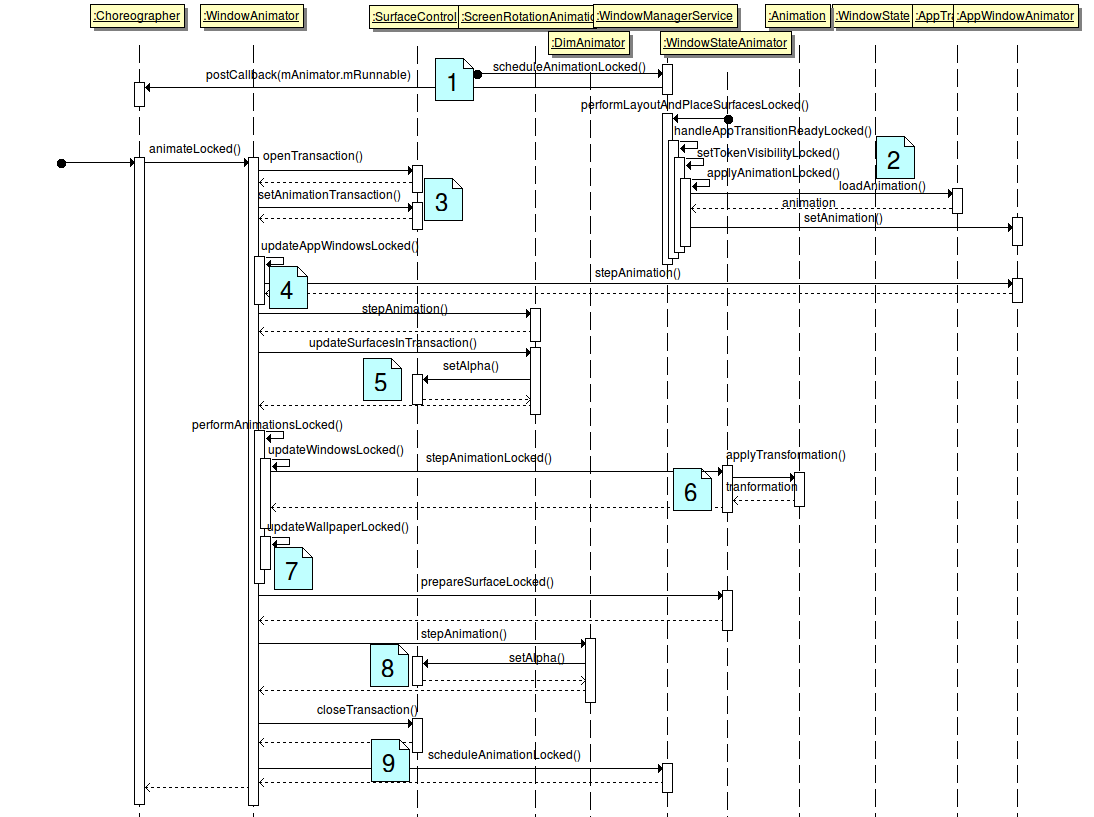
*Animation*:  
    Animation抽象类，里面最重要的一个接口就是applyTranformation, 它的输入是当前的一个描述进度的浮点数(0.0 ~ 1.0)， 输出是一个Transformation类对象，这个对象里有两个重要的成员变量，mAlpha 和 mMatrix, 前者表示下一个动画点的透明度(用于灰度渐变效果），后者则是一个变形矩阵，通过它可以生成各种各样的变形效果。Android提供了很多Animation的具体实现，比如RotationAnimation, AlphaAnimation 等等，用户也可以实现自己的Animation类，只需要重载applyTransform 这个接口。注意，Animation类只生成绘制动画所需的参数（alpha 或 matrix)，不负责完成绘制工作。完成这个工作的是Animator.

*Animator：*控制动画的‘人’， 它通常通过向定时器Choreographer 注册一个Runnable对象来实现定时触发，在回调函数里它要做两件事情：1. 从Animation那里获取新的Transform， 2. 将Transform里的值更新底层参数，为接下来的重绘做准备。动画可以发生在Window上，也可以发生在某个具体的View。前者的动画会通过SurfaceControl直接在某个Surface上进行操作(会在SurfaceFlinger里详细描述），比如设置Alpha值。后者则通过OpenGL完成（生成我们前面提过的DisplayList).

*WindowStateAnimator, WindowAnimator,  AppWindowAnimator:*针对不同对象的Animator. WindowAnimator, 负责整个屏幕的动画，比如说转屏，它提供Runnable实现。WindowStateAnimator, 负责ViewRoot，即某一个窗口的动画。AppWindowAnimator, 负责应用启动和退出时候的动画。这几个Animator都会提供一个函数，stepAnimationLocked(), 它会完成一个动画动作的一系列工作，从计算Transformation到更新Surface的Matrix.



具体来看一下Window的Animation和View的Animation



1. WindowManagerService 的 scheduleAnimationLocked() 将windowAnimator的mAnimationRunnable 注册到定时器 Choreographer.
2. 如果应用程序的res/anim/下有xml文件定义animation，在layout过程中，会通过appTransition类的loadAnimation() 函数将XML转换成 Animation\_Set 对象，它里面可以包含多个Animation。
3. 当下一个VSYNC事件到来，刚才注册的Callback函数被调用，即WindowAnimator的mAnimationRunnable，里面调用 animateLocked(), 首先，打开一个SurfaceControl的动画会话，animationSession。
4. 首先执行的动画是 appWindowAnimator, 如果刚才loadAnimation() 返回的animation不为空，便会走到Animation的getTransform() 获取动画的参数，这里可能会同时有多个动画存在，通过Transform的compose()函数将它们最终合为一个。
5. 接下来上场的是DisplayContentsAnimator, 它主要用来实现灰度渐变和转屏动画。同样，首先通过stepAnimation() 获取动画变形参数，然后通过SurfaceControl将其更新到SrufaceFlinger内部对应的Layer. 这里首先完成的是转屏的动画
6. 然后就是每个窗口的动画。后面跟着的 perpareSurfaceLocked() 则会更新参数。
7. Wallpaper的动画。
8. 接下来，就是上面提到的DisplayContentsAnimator的第二部分，通过DimLayer实现渐变效果。
9. Surface的控制完成后，关闭对话。然后scheduleAnimationLocked() 规划下一步动画。
10. 接下来的performDraw()会把所有更新参数的View，或Surface交给OpenGL或HWcomposer进行处理，于是我们就看到了动画效果。

 View 的动画实现步骤与Windows 类似，有兴趣的同学可以去看View.java 的 drawAnimation() 函数。

**管理窗口**

WMS 里面管理着各式各样的窗口, 如下表所示(在WindowManagerService.java 中定义）

|  |  |  |
| --- | --- | --- |
|  | 类型 | 用途 |
| mAnimatingAppToken | ArrayList<AppWindowToken> | 正在动画中的应用 |
| mExistingAppToken | ArrayList<AppWindowToken> | 退出但退出动画还没有完成的应用。 |
| mResizingWindows | ArrayList<WindowState> | 尺寸正在改变的窗口，当改变完成后，需要通知应用。 |
| mFinishedStarting | ArrayList<AppWindowToken> | 已经完成启动的应用。 |
| mPendingRemove | ArrayList<WindowState> | 动画结束的窗口。 |
| mLosingFocus | ArrayList<WindowState> | 失去焦点的窗口，等待获得焦点的窗口进行显示。 |
| mDestorySurface | ArrayList<WindowState> | 需要释放Surface的窗口。 |
| mForceRemoves | ArrayList<WindowState> | 需要强行关闭的窗口，以释放内存。 |
| mWaitingForDrawn | ArrayList<Pair<WindowState, IRemoteCallback>> | 等待绘制的窗口 |
| mRelayoutWhileAnimating | ArrayList<WindowState> | 请求relayout但此时仍然在动画中的窗口。 |
| mStrictModeFlash | StrictModeFlash | 一个红色的背景窗口，用于提示可能存在的内存泄露。 |
| mCurrentFocus | WindowState | 当前焦点窗口 |
| mLastFocus | WindowState | 上一焦点窗口 |
| mInputMethodTarget | WindowState | 输入法窗口下面的窗口。 |
| mInputMethodWindow | WindowState | 输入法窗口 |
| mWallpaperTarget | WindowState | 墙纸窗口 |
| mLowerWallpaperTarget | WindowState | 墙纸切换动画过程中Z-Order 在下面的窗口 |
| mHigherWallpaperTarget | WindowState | 墙纸切换动画过程中Z-Order 在上面的窗口 |
|  |  |  |
|  |  |  |

可以看到这里大量的用到了队列，不同的窗口，或同一窗口在不同的阶段，可能会出现在不同的队列里。另外因为WindowManager Service 的服务可能被很多个线程同时调用，在这种复杂的多线程环境里，通过锁来实现线程安全非常难以实现，一不小心就可能导致死锁，所以在 WindowManager 内专门有一个执行线程(WM Thread)来将所有的服务请求通过消息进行异步处理，实现调用的序列化。队列是实现异步处理的常用手段。队列加Looper线程是Android 应用常用的设计模型。

此外，WindowManager还根据Window的类型进行了分类(在WindowManager.java)，如下表，

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 类型 | 常量范围 | 子类 | 常量值 | 说明 | 例子 |
| APPLICATION\_WINDOW | 1～99 | TYPE\_BASE\_APPLICATION | 1 |  |  |
|  |  | TYPE\_APPLICATION | 2 | 应用窗口 | 大部分的应用程序窗口 |
|  |  | TYPE\_APPLICATION\_STARTING | 3 | 应用程序的Activity显示之前由系统显示的窗口 |  |
|  |  | LAST\_APPLICATION\_WINDOW | 99 |  |  |
| SUB\_WINDOW | 1000～1999 | FIRST\_SUB\_WINDOW | 1000 |  |  |
|  |  | TYPE\_APPLICATION\_PANEL | 1000 | 显示在母窗口之上，遮挡其下面的应用窗口。 |  |
|  |  | TYPE\_APPLICATION\_MEDIA | 1001 | 显示在母窗口之下，如果应用窗口不挖洞，即不可见。 | SurfaceView，在小窗口显示时设为MEDIA, 全屏显示时设为PANEL |
|  |  | TYPE\_APPLICATION\_SUB\_PANEL | 1002 |  |  |
|  |  | TYPE\_APPLICATION\_ATTACHED\_DIALOG | 1003 |  |  |
|  |  | TYPE\_APPLICATION\_MEIDA\_OVERLAY | 1004 | 用于两个SurfaceView的合成，如果设为MEDIA,  则上面的SurfaceView 挡住下面的SurfaceView |  |
| SYSTEM\_WINDOW | 2000~2999 | TYPE\_STATUS\_BAR | 2000 | 顶部的状态栏 |  |
|  |  | TYPE\_SEARCH\_BAR | 2001 | 搜索窗口，系统中只能有一个搜索窗口 |  |
|  |  | TYPE\_PHONE | 2002 | 电话窗口 |  |
|  |  | TYPE\_SYSTEM\_ALERT | 2003 | 警告窗口，在所有其他窗口之上显示 | 电量不足提醒窗口 |
|  |  | TYPE\_KEYGUARD | 2004 | 锁屏界面 |  |
|  |  | TYPE\_TOAST | 2005 | 短时的文字提醒小窗口 |  |
|  |  | TYPE\_SYSTEM\_OVERLAY | 2006 | 没有焦点的浮动窗口 |  |
|  |  | TYPE\_PRIORITY\_PHONE | 2007 | 紧急电话窗口，可以显示在屏保之上 |  |
|  |  | TYPE\_SYSTEM\_DIALOG | 2008 | 系统信息弹出窗口 | 比如SIM插上后弹出的运营商信息窗口 |
|  |  | TYPE\_KEYGUARD\_DIALOG | 2009 | 跟KeyGuard绑定的弹出对话框 | 锁屏时的滑动解锁窗口 |
|  |  | TYPE\_SYSTEM\_ERROR | 2010 | 系统错误提示窗口 | ANR 窗口 |
|  |  | TYPE\_INPUT\_METHOD | 2011 | 输入法窗口，会挤占当前应用的空间 |  |
|  |  | TYPE\_INPUT\_METHOD\_DIALOG | 2012 | 弹出的输入法窗口，不会挤占当前应用窗口空间，在其之上显示 |  |
|  |  | TYPE\_WALLPAPER | 2013 | 墙纸 |  |
|  |  | TYPE\_STATUS\_BAR\_PANEL | 2014 | 从状态条下拉的窗口 |  |
|  |  | TYPE\_SECURE\_SYSTEM\_OVERLAY | 2015 | 只有系统用户可以创建的OVERLAY窗口 |  |
|  |  | TYPE\_DRAG | 2016 | 浮动的可拖动窗口 | 360安全卫士的浮动精灵 |
|  |  | TYPE\_STATUS\_BAR\_PANEL | 2017 |  |  |
|  |  | TYPE\_POINTER | 2018 | 光标 |  |
|  |  | TYPE\_NAVIGATION\_BAR | 2019 |  |  |
|  |  | TYPE\_VOLUME\_OVERLAY | 2020 | 音量调节窗口 |  |
|  |  | TYPE\_BOOT\_PROGRESS | 2021 | 启动进度，在所有窗口之上 |  |
|  |  | TYPE\_HIDDEN\_NAV\_CONSUMER | 2022 | 隐藏的导航栏 |  |
|  |  | TYPE\_DREAM | 2023 | 屏保动画 |  |
|  |  | TYPE\_NAVIGATION\_BAR\_PANEL | 2024 | Navigation bar 弹出的窗口 | 比如说应用收集栏 |
|  |  | TYPE\_UNIVERSAL\_BACKGROUND | 2025 |  |  |
|  |  | TYPE\_DISPLAY\_OVERLAY | 2026 | 用于模拟第二显示设备 |  |
|  |  | TYPE\_MAGNIFICATION | 2027 | 用于放大局部 |  |
|  |  | TYPE\_RECENTS\_OVERLAY | 2028 | 当前应用窗口，多用户情况下只显示在用户节目 |  |

windowManager Service 会根据窗口的类型值来决定Z-Order (于常量值无关，值大说明是后面Android版本添加的，比如说2025～2028就是4.3 新加的)。比如说SurfaceView.java 里的一个函数，

[复制代码](javascript:void(0);)

public void setZOrderOnTop(boolean onTop) {

if (onTop) {

mWindowType = WindowManager.LayoutParams.TYPE\_APPLICATION\_PANEL; //PANEL在上面

// ensures the surface is placed below the IME

mLayout.flags |= WindowManager.LayoutParams.FLAG\_ALT\_FOCUSABLE\_IM;

} else {

mWindowType = WindowManager.LayoutParams.TYPE\_APPLICATION\_MEDIA; //MEDIA类型窗口在应用窗口之下，应用必需挖洞(设Alpha值)才能露出它。

mLayout.flags &= ~WindowManager.LayoutParams.FLAG\_ALT\_FOCUSABLE\_IM;

}

}

[复制代码](javascript:void(0);)

这些类型最终在WindowManager 内部转换成几个Z-Order 值，mBaseLayer, mSubLayer, mAnimationLayer, 分别表明主窗口，子窗口（附加在主窗口之上），和动画窗口的Z-Order值（越大越在上边）。不同的窗口类型在不同的硬件产品上有不同的定义，因此它是实现在WindowManagerPolicy里的windowTypeToLayerLw(), 举PhoneWindowManager 为例，它的ZOrder 顺序是:

Univese background < Wallpaper < Phone < Search Bar < System Dialog < Input Method Window < Keyguard < Volume < System　Overlay　< Navigation < System Error <  < Display Overlay< Drag < Pointer < Hidden NAV consumer,

所以，我们如果要在手机锁屏时显示歌曲播放进度，就必须给这个窗口分配一个大于Keyguard的type，如 system overlay 等。

一个Window可以有若干个Sub Window, 他们和主窗口的ZOrder关系是

Media Sublayer(-2) < Media Overlay sublayer (-1) < Main Layer(0) < Attached Dialog (1) < Sub panel Sublayer (2)

通过 "adb shell dumpsys window" 可以查看系统当前运行的窗口的ZOrder 和 Visibility, 比如下面就是在短信输入界面下运行“dumpsys" 获得的结果，

[复制代码](javascript:void(0);)

1 Window #0 Window{4ea4e178 u0 Keyguard}:

2 mBaseLayer=121000 mSubLayer=0 mAnimLayer=121000+0=121000 mLastLayer=121000

3 mViewVisibility=0x8 mHaveFrame=true mObscured=false

4 Window #1 Window{4ea4aa7c u0 InputMethod}:

5 mBaseLayer=101000 mSubLayer=0 mAnimLayer=21020+0=21020 mLastLayer=21020

6 mViewVisibility=0x0 mHaveFrame=true mObscured=false

7 Window #2 Window{4ec1a150 u0 com.android.mms/com.android.mms.ui.ComposeMessageActivity}:

8 mBaseLayer=21000 mSubLayer=0 mAnimLayer=21015+0=21015 mLastLayer=21015

9 mViewVisibility=0x0 mHaveFrame=true mObscured=false

10 Window #3 Window{4ea7c714 u0 com.android.mms/com.android.mms.ui.ConversationList}:

11 mBaseLayer=21000 mSubLayer=0 mAnimLayer=21010+0=21010 mLastLayer=21015

12 mViewVisibility=0x8 mHaveFrame=true mObscured=true

13 Window #4 Window{4eaedefc u0 com.android.launcher/com.android.launcher2.Launcher}:

14 mBaseLayer=21000 mSubLayer=0 mAnimLayer=21005+0=21005 mLastLayer=21010

15 mViewVisibility=0x8 mHaveFrame=true mObscured=true

16 Window #5 Window{4ea17064 u0 jackpal.androidterm/jackpal.androidterm.Term}:

17 mBaseLayer=21000 mSubLayer=0 mAnimLayer=21000+0=21000 mLastLayer=22000

18 mViewVisibility=0x8 mHaveFrame=true mObscured=true

[复制代码](javascript:void(0);)

可以看到：

1. 当前只有两个窗口可见， InputMethod 和 com.android.mms/com.android.mms.ui.ComposeMessageActivity, mViewVisibility = 0 (0在View.java定义是可见）, 而其他 mViewVisibility=0x8 (定义在View.java里, 意思是”GONE").
2. InputMethod(mLastLayer=21020) 在 ComposeMessageActivity(mLastLayer=21015) 之上. 细心的同学可能会发现，InputMethod的mBaseLayer = 101000，为什么mLastLayer小那么多？因为mLastLayer才是真正的z-order, 它经过了WidowManager的调整。当用户点击输入框，View会通过InputMethodManager 发送一个showSoftInput命令，经过InputManagerService的处理，输入法窗口(KeyboardView)会被加入到WndowManager Service里，WindowManager Service 会寻找它的目标窗口, 即需要输入的窗口，（遍历WindowList 然后根据窗口的Flags判断)，然后将输入法窗口的mLayer值改为 目标窗口的mLayer + 5，这样，输入法窗口就显示在了目标窗口之上。在这里，输入法窗口存在于InputMethodManagerService 的上下文里，而不是某个Activity，所以他可以跟任何需要输入法的Activity绑定。其他一些应用，比如说PiP（三星的Galaxy S3可以在一个浮动的小窗口里显示视频）也是运用了类似的方法来实现的。Android的输入法是一个非常值得研究的模块，留到后面探讨。

所以，WindowManager Service 是通过调整窗口的mViewVisibility 和 mLayer 值来实现窗口重叠。最后给出跟Z-order相关的类图。



 图中序号表示输入法窗口找到它的目标窗口的过程：

1. WindowManagerService 找到默认输出（Default Display) 的DisplayContents成员变量。
2. 里面有一个数组WindowList-mWindows, 按照Z-Order顺序保存了当前在这个Display上输出的所有窗口WindowState。
3. 遍历所有的WindowState，判断它的mAppToken是否和输入法窗口的mAppToken一致。呼起输入法窗口的窗口会将自己的mAppToken拷贝给它。
4. 相同的Token下，可能有多个窗口，通过WindowToken.windows 或者 AppWindowToken.allAppWindows, 可以找到他们。

WindowManager Service的介绍暂告一段落，它与其他重要的Service，SurfaceFlinger, ActivityManager， InputManager, PowerManager, WatchDog 之间的关系将在其他文章介绍。

* 评论
* 收藏

Android窗口管理服务WindowManagerService的简要介绍和学习计划

[审核此文](javascript:showCheckDoc('jyUjim','http://www.itboth.com/d/jyUjim/android-surfaceflinger-window-phonewindowmanager-windowmanagerservice');)

itRec 分享于 2017-04-24 阅读 57365 收藏 0

主题 [Android](http://www.itboth.com/tag/Android) [WindowManagerService](http://www.itboth.com/tag/WindowManagerService) [SurfaceFlinger](http://www.itboth.com/tag/SurfaceFlinger) [Window](http://www.itboth.com/tag/Window) [PhoneWindowManager](http://www.itboth.com/tag/PhoneWindowManager)

**领取地址**：

推荐：[Android窗口管理服务WindowManagerService对窗口的组织方式分析](http://www.itboth.com/d/ZFJFzu/android-windowmanagerservice)

[我们知道，在Android系统中，Activity是以堆栈的形式组织在ActivityManagerService服务中的。与Activity类似，Android系统中的窗口也是以堆栈的形式组织在WindowManagerSe

        在前一个系列文章中，我们从个体的角度来分析了Android应用程序窗口的实现框架。事实上，如果我们从整体的角度来看，Android应用程序窗口的实现要更复杂，因为它们的类型和作用不同，且会相互影响。在Android系统中，对系统中的所有窗口进行管理是窗口管理服务WindowManagerService的职责。在本文中，我们就将简要介绍WindowManagerService的职能以及制定学习计划。

《Android系统源代码情景分析》一书正在进击的程序员网（[http://0xcc0xcd.com](http://0xcc0xcd.com/)）中连载，点击进入！

        我们知道，在Android系统中，同一时刻，只有一个Activity窗口是激活的，但是，对于WindowManagerService服务来说，这并不意味着它每次只需要管理一个Activity窗口，例如，在两个Activity窗口的切换过程中，前后两个Activity窗口实际上都是可见的。即使在只有一个Activity窗口是可见的时候，WindowManagerService服务仍然需要同时管理着多个窗口，这是因为可见的Activity窗口可能还会被设置了壁纸窗口（Wallpaper Winodw）或者弹出了子窗口（Sub Window），以及可能会出现状态栏（Status Bar）以及输入法窗口（Input Method Window），如图1所示。

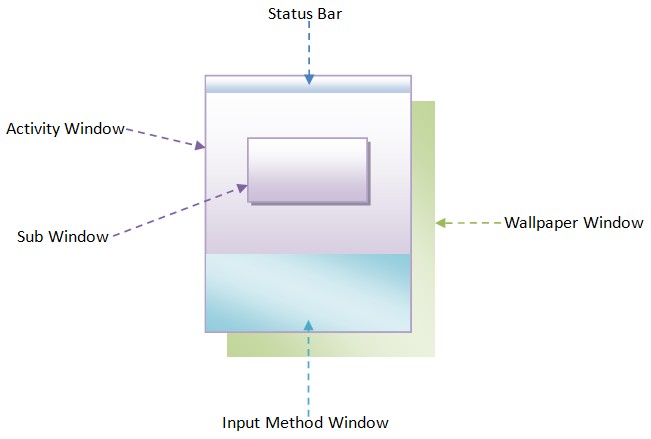


图1 Activity窗口及其子窗口、壁纸窗口、输入法窗口和状态栏的位置结构

        因此，WindowManagerService服务是不可以假设同一时刻它只需要管理一个窗口的，它需要通过各个窗口在屏幕上的位置以及大小来决定哪些窗口需要显示的以及要显在哪里，这实际上就是要计算出各个窗口的可见区域。

        从前面[Android系统Surface机制的SurfaceFlinger服务渲染应用程序UI的过程分析](http://blog.csdn.net/luoshengyang/article/details/8079456)一文可以知道，SurfaceFlinger服务在渲染整个屏幕的UI的时候，会对各个窗品的可见性进行计算，因此，WindowManagerService服务只要将它所管理的各个窗品的位置以及大小告诉SurfaceFlinger服务，后者可以帮帮它计算出各个窗口的可见区域了。注意，这里，这里所说的窗口位置包括窗口在X、Y和Z轴的位置。

       WindowManagerService服务大致按照以下方式来控制哪些窗口需要显示的以及要显在哪里：

       1. 每一个Activity窗口的大小都等于屏幕的大小，因此，只要对每一个Activity窗口设置一个不同的Z轴位置，然后就可以使得位于最上面的，即当前被激活的Activity窗口，才是可见的。

       2. 每一个子窗口的Z轴位置都比它的父窗口大，但是大小要比父窗口小，这时候Activity窗口及其所弹出的子窗口都可以同时显示出来。

       3. 对于非全屏Activity窗口来说，它会在屏幕的上方留出一块区域，用来显示状态栏。这块留出来的区域称对于屏幕来说，称为装饰区（decoration），而对于Activity窗口来说，称为内容边衬区（Content Inset）。

       4. 输入法窗口只有在需要的时候才会出现，它同样是出现在屏幕的装饰区或者说Activity窗口的内容边衬区的。

       5. 对于壁纸窗口，它出现需要壁纸的Activity窗口的下方，这时候要求Activity窗口是半透明的，这样就可以将它后面的壁纸窗口一同显示出来。

       6. 两个Activity窗口在切换过程，实际上就是前一个窗口显示退出动画而后一个窗口显示开始动画的过程，而在动画的显示过程，窗口的大小会有一个变化的过程，这样就导致前后两个Activity窗口的大小不再都等于屏幕的大小，因而它们就有可能同时都处于可见的状态。事实上，Activity窗口的切换过程是相当复杂的，因为即将要显示的Activity窗口可能还会被设置一个启动窗口（Starting Window）。一个被设置了启动窗口的Activity窗口要等到它的启动窗口显示了之后才可以显示出来。

       从以上六点就可以看出，窗口在X、Y和Z轴的位置及其大小的计算非常重要，它们共同决定了一个窗口是否是整体可见的，还是部分可见的，或者整体不可见的。在Android系统中，WindowManagerService服务是通过一个实现了WindowManagerPolicy接口的策略类来计算一个窗口的位置和大小的。例如，在Phone平台上，这个策略类就是PhoneWindowManager。这样做的好处就是对于不同的平台实现不同的策略类来达到不同的窗口控制模式。

      从上面的描述就可以看出，WindowManagerService服务除了要与Activity窗口所运行在的应用程序进程打交道之外，还需要与SurfaceFlinger服务以及窗口管理策略类PhoneWindowManager交互，如图2所示。

推荐：[Android窗口管理服务WindowManagerService对壁纸窗口（Wallpaper Window）的管理分析](http://www.itboth.com/d/EJJJvy/android-window-windowmanagerservice-wallpaper)

[    在Android系统中，壁纸窗口和输入法窗口一样，都是一种特殊类型的窗口，而且它们都是喜欢和一个普通的Activity窗口缠绵在一起。大家可以充分地想象这样的一个3W场景：



图2 WindowManagerService服务与Activity窗口、SurfaceFlinger服务、PhoneWindowManager策略的关系图

        在前面[Android应用程序窗口（Activity）实现框架简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/8170307)的一系列文章中，我们已经分析过应用程序进程与WindowManagerService服务之间的交互过程了，因此，在这一系列文章中，我们就将主要分析WindowManagerService服务的实现，以及它与SurfaceFlinger服务、PhoneWindowManager策略类的交互过程。

        从总体上来看，WindowManagerService服务的实现是相当复杂的，例如，WindowManagerService类的核心成员函数performLayoutAndPlaceSurfacesLockedInner的代码有1200+行，比600-行代码的ViewRoot类的核心成员函数performTraversals还要恐怖。不过，WindowManagerService服务实现的复杂性是在预料之中的，毕竟它要管理的整个系统所有窗口的UI，而在任何一个系统中，窗口管理子系统都是极其复杂的。基于上述理由，采用硬碰硬的方式来分析WindowManagerService服务的实现是以卵击石，因此，这个系列的文章将对WindowManagerService服务进行分拆，然后再逐个击破，这是算法中的分而治之思想是一致的。

        具体来说，我们将按照以下几个情景来分析WindowManagerService服务的实现：

        1. [窗口大小和位置（X轴和Y轴）的计算过程](http://blog.csdn.net/luoshengyang/article/details/8479101)。

        2. [窗口的组织方式](http://blog.csdn.net/luoshengyang/article/details/8498908)。

        3. [输入法窗口的调整过程](http://blog.csdn.net/luoshengyang/article/details/8526644)。

        4. [壁纸窗口的调整过程](http://blog.csdn.net/luoshengyang/article/details/8550820)。

        5. [窗口Z轴位置的计算和调整过程](http://blog.csdn.net/luoshengyang/article/details/8570428)。

        6. [Activity窗口的启动窗口的显示过程](http://blog.csdn.net/luoshengyang/article/details/8577789)。

        7. [Activity窗口的切换过程](http://blog.csdn.net/luoshengyang/article/details/8596449)。

        8.[Activity窗口的动画显示过程](http://blog.csdn.net/luoshengyang/article/details/8611754)。

        再次地，由于WindowManagerService服务的实现实在是太复杂，因此上述八个情景可能还不足于说明WindowManagerService服务的实现。如果出现这种情况，我们在分析的过程中会进行相应的调整。相信对WindowManagerService服务的实现进行分而治之的分析后，我们就可以对Android系统的UI架构有一个深刻的理解！敬请关注接下来的文章！

**老罗的新浪微博：**[**http://weibo.com/shengyangluo**](http://weibo.com/shengyangluo)**，欢迎关注！**

推荐：[Android窗口管理服务WindowManagerService对输入法窗口（Input Method Window）的管理分析](http://www.itboth.com/d/RzQ7BjNvIBjy/android-windowmanagerservice)

[    在Android系统中，输入法窗口是一种特殊类型的窗口，它总是位于需要使用输入法的窗口的上面。也就是说，一旦WindowManagerService服务检测到焦点窗口需要使用输入法，

Android窗口管理服务WindowManagerService对窗口的组织方式分析

[审核此文](javascript:showCheckDoc('ZFJFzu','http://www.itboth.com/d/ZFJFzu/android-windowmanagerservice');)

itRec 分享于 2017-04-14 阅读 45033 收藏 0

主题 [WindowManagerService](http://www.itboth.com/tag/WindowManagerService) [android](http://www.itboth.com/tag/android) [窗口组织](http://www.itboth.com/tag/%E7%AA%97%E5%8F%A3%E7%BB%84%E7%BB%87)

**领取地址**：

推荐：[Android窗口管理服务WindowManagerService对输入法窗口（Input Method Window）的管理分析](http://www.itboth.com/d/RzQ7BjNvIBjy/android-windowmanagerservice)

[    在Android系统中，输入法窗口是一种特殊类型的窗口，它总是位于需要使用输入法的窗口的上面。也就是说，一旦WindowManagerService服务检测到焦点窗口需要使用输入法，

        我们知道，在Android系统中，Activity是以堆栈的形式组织在ActivityManagerService服务中的。与Activity类似，Android系统中的窗口也是以堆栈的形式组织在WindowManagerService服务中的，其中，Z轴位置较低的窗口位于Z轴位置较高的窗口的下面。在本文中，我们就详细分析WindowManagerService服务是如何以堆栈的形式来组织窗口的。

《Android系统源代码情景分析》一书正在进击的程序员网（[http://0xcc0xcd.com](http://0xcc0xcd.com/)）中连载，点击进入！

        从前面[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)一文可以知道，应用程序进程中的每一个Activity组件在Activity管理服务ActivityManagerService中都对应有一个ActivityRecord对象。从前面[Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析](http://blog.csdn.net/luoshengyang/article/details/8275938)一文又可以知道，Activity管理服务ActivityManagerService中每一个ActivityRecord对象在Window管理服务WindowManagerService中都对应有一个AppWindowToken对象。

        此外，在输入法管理服务InputMethodManagerService中，每一个输入法窗口都对应有一个Binder对象，这个Binder对象在Window管理服务WindowManagerService又对应有一个WindowToken对象。

        与输入法窗口类似，在壁纸管理服务WallpaperManagerService中，每一个壁纸窗口都对应有一个Binder对象，这个Binder对象在Window管理服务WindowManagerService也对应有一个WindowToken对象。

        在Window管理服务WindowManagerService中，无论是AppWindowToken对象，还是WindowToken对象，它们都是用来描述一组有着相同令牌的窗口的，每一个窗口都是通过一个WindowState对象来描述的。例如，一个Activity组件窗口可能有一个启动窗口（Starting Window），还有若干个子窗口，那么这些窗口就会组成一组，并且都是以Activity组件在Window管理服务WindowManagerService中所对应的AppWindowToken对象为令牌的。从抽象的角度来看，就是在Window管理服务WindowManagerService中，每一个令牌（AppWindowToken或者WindowToken）都是用来描述一组窗口（WindowState）的，并且每一个窗口的子窗口也是与它同属于一个组，即都有着相同的令牌。

        上述的窗口组织方式如图1所示：

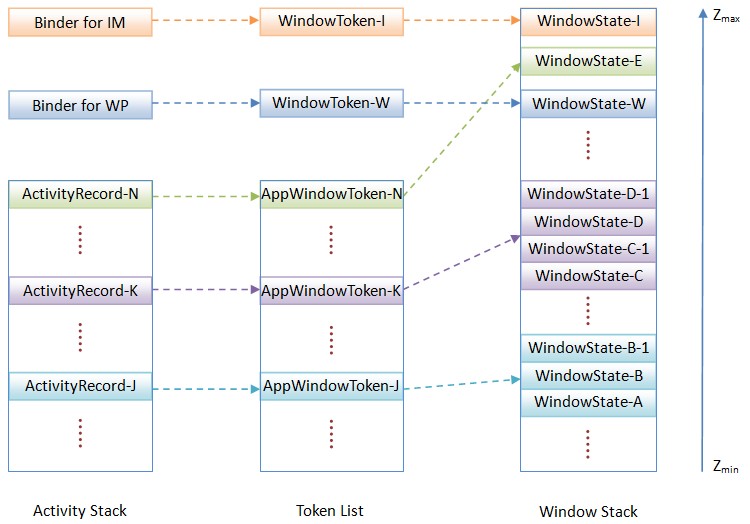


图1 窗口在WindowManagerService服务中的组织方式

        其中，Activity Stack是在ActivityManagerService服务中创建的，Token List和Window Stack是在WindowManagerService中创建的，而Binder for IM和Binder for WP分别是在InputMethodManagerService服务和WallpaperManagerService服务中创建的，用来描述一个输入法窗口和一个壁纸窗口。

        图1中的对象的对应关系如下所示：

       1. ActivityRecord-J对应于AppWindowToken-J，后者描述的一组窗口是{WindowState-A, WindowState-B, WindowState-B-1}，其中， WindowState-B-1是WindowState-B的子窗口。

       2. ActivityRecord-K对应于AppWindowToken-K，后者描述的一组窗口是{WindowState-C, WindowState-C-1, WindowState-D, WindowState-D-1}，其中， WindowState-C-1是WindowState-C的子窗口，WindowState-D-1是WindowState-D的子窗口。

       3. ActivityRecord-N对应于AppWindowToken-N，后者描述的一组窗口是{WindowState-E}，其中， WindowState-E是系统当前激活的Activity窗口。

       4. Binder for IM对应于WindowToken-I，后者描述的一组窗口是{WindowState-I}，其中， WindowState-I是WindowState-E的输入法窗口。

       5. Binder for WP对应于WindowToken-W，后者描述的一组窗口是{WindowState-W}，其中， WindowState-W是WindowState-E的壁纸窗口。

       从图1还可以知道，Window Stack中的WindowState是按照它们所描述的窗口的Z轴位置从低到高排列的。

       以上就是WindowManagerService服务组织系统中的窗口的抽象模型，接下来我们将分析AppWindowToken、WindowToken和WindowState的一些增加、移动和删除等操作，以便可以对这个抽象模型有一个更深刻的认识。

       1.  增加AppWindowToken

       从前面[Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析](http://blog.csdn.net/luoshengyang/article/details/8275938)一文可以知道，一个Activity组件在启动的过程中，ActivityManagerService服务会调用调用WindowManagerService类的成员函数addAppToken来为它增加一个AppWindowToken，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

/\*\*

\* Mapping from a token IBinder to a WindowToken object.

\*/

**final** HashMap<IBinder, WindowToken> mTokenMap =

**new** HashMap<IBinder, WindowToken>();

/\*\*

\* The same tokens as mTokenMap, stored in a list for efficient iteration

\* over them.

\*/

**final** ArrayList<WindowToken> mTokenList = **new** ArrayList<WindowToken>();

......

/\*\*

\* Z-ordered (bottom-most first) list of all application tokens, for

\* controlling the ordering of windows in different applications. This

\* contains WindowToken objects.

\*/

**final** ArrayList<AppWindowToken> mAppTokens = **new** ArrayList<AppWindowToken>();

......

**public** **void** **addAppToken**(**int** addPos, IApplicationToken token,

**int** groupId, **int** requestedOrientation, **boolean** fullscreen) {

......

**synchronized**(mWindowMap) {

AppWindowToken wtoken = findAppWindowToken(token.asBinder());

**if** (wtoken != **null**) {

......

**return**;

}

wtoken = **new** AppWindowToken(token);

......

mAppTokens.add(addPos, wtoken);

......

mTokenMap.put(token.asBinder(), wtoken);

mTokenList.add(wtoken);

......

}

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       WindowManagerService类有三个成员变量mTokenMap、mTokenList和mAppTokens，它们都是用来描述系统中的窗口的。

       成员变量mTokenMap指向的是一个HashMap，它里面保存的是一系列的WindowToken对象，每一个WindowToken对象都是用来描述一个窗口的，并且是以描述这些窗口的一个Binder对象的IBinder接口为键值的。例如，对于Activity组件类型的窗口来说，它们分别是以用来描述它们的一个ActivityRecord对象的IBinder接口保存在成员变量mTokenMap所指向的一个HashMap中的。

       成员变量mTokenList指向的是一个ArrayList，它里面保存的也是一系列WindowToken对象，这些WindowToken对象与保存在成员变量mTokenMap所指向的一个HashMap中的WindowToken对象是一样的。成员变量mTokenMap和成员变量mTokenList的区别就在于，前者在给定一个IBinder接口的情况下，可以迅速指出是否存在一个对应的WindowToken对象，而后者可以迅速遍历WindowManagerService服务中的WindowToken对象。

       成员变量mAppTokens指向的也是一个ArrayList，不过它里面保存的是一系列AppWindowToken对象，每一个AppWindowToken对象都是用来描述一个Activity组件窗口的，而这些AppWindowToken对象是以它们描述的窗口的Z轴坐标由小到大保存在这个ArrayList中的，这样我们就可以通过这个ArrayList来从上到下或者从下到上地遍历系统中的所有Activity组件窗口。由于这些AppWindowToken对象所描述的Activity组件窗口也是一个窗口，并且AppWindowToken类是从WindowToken继承下来的，因此，这些AppWindowToken对象还会同时被保存在成员变量mTokenMap所指向的一个HashMap和成员变量mTokenList所指向的一个ArrayList中。

       理解了WindowManagerService类的这三个成员变量的含义之后，它的成员函数addAppToken的实现就好理解了，其中，参数token指向的便是用来描述正在启动的Activity组件所对应的一个ActivityRecord对象，而参数addPos用来描述该Activity组件在堆栈中的位置，这个位置同时也是接下来要创建的AppWindowToken对象在WindowManagerService类的mTokenList所描述的一个ArrayList中的位置。

       WindowManagerService类的成员函数addAppToken首先调用另外一个成员函数findAppWindowToken来在成员变量mTokenMap所描述的一个HashMap检查是否已经存在一个AppWindowToken。如果已经存在的话，那么WindowManagerService类的成员函数addAppToken就什么也不做就返回了，否则的话，就会使用参数token来创建一个AppWindowToken对象，并且会将该AppWindowToken对象分别保存在WindowManagerService类的成员变量mTokenMap、mTokenList和mAppTokens中。

       2. 删除AppWindowToken

       删除AppWindowToken是通过调用WindowManagerService类的成员函数removeAppTokensLocked来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **void** **removeAppTokensLocked**(List<IBinder> tokens) {

// XXX This should be done more efficiently!

// (take advantage of the fact that both lists should be

// ordered in the same way.)

**int** N = tokens.size();

**for** (**int** i=0; i<N; i++) {

IBinder token = tokens.get(i);

**final** AppWindowToken wtoken = findAppWindowToken(token);

**if** (!mAppTokens.remove(wtoken)) {

......

i--;

N--;

}

}

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        WindowManagerService类的成员函数removeAppTokensLocked可以同时删除一组AppWindowToken对象。

        参数tokens所描述的是一个IBinder接口列表，与这些IBinder接口所对应的AppWindowToken对象就是接下来要删除的。WindowManagerService类的成员函数removeAppTokensLocked通过一个for循环来依次调用另外一个成员函数findAppWindowToken，以便可以找到保存在列表tokens中的每一个IBinder接口所对应的AppWindowToken对象，然后将该AppWindowToken对象从WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList中删除。

        注意，WindowManagerService类的成员函数removeAppTokensLocked是在内部使用的，它只是把一个AppWindowToken对象从成员变量mAppTokens中删除，而没有从另外两个成员变量mTokenMap和mTokenList中删除。

        3. 移动AppWindowToken至指定位置

        移动AppWindowToken至指定位置是通过调用WindowManagerService类的成员函数moveAppToken来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**public** **void** **moveAppToken**(**int** index, IBinder token) {

**if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,

"moveAppToken()")) {

**throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");

}

**synchronized**(mWindowMap) {

......

**final** AppWindowToken wtoken = findAppWindowToken(token);

**if** (wtoken == **null** || !mAppTokens.remove(wtoken)) {

......

**return**;

}

mAppTokens.add(index, wtoken);

......

**final** **long** origId = Binder.clearCallingIdentity();

......

**if** (tmpRemoveAppWindowsLocked(wtoken)) {

......

reAddAppWindowsLocked(findWindowOffsetLocked(index), wtoken);

......

updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES);

mLayoutNeeded = **true**;

performLayoutAndPlaceSurfacesLocked();

}

Binder.restoreCallingIdentity(origId);

}

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。 

        参数token描述的是要移动的AppWindowToken对象所对应的一个IBinder接口，而参数index描述的是该AppWindowToken对象要移动到的位置。注意，移动一个AppWindowToken对象到指定的位置是需要android.Manifest.permission.MANAGE\_APP\_TOKENS权限的。

        WindowManagerService类的成员函数moveAppToken首先找到与参数token所对应的AppWindowToken对象，并且将该AppWindowToken对象从WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList中移除，这样做的目的是为了接下来可以将该AppWindowToken对象移动至该ArrayList中的指定位置上，即参数index所描述的位置上。

        注意，上述操作只是将参数token所对应的AppWindowToken对象移动到了WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList的指定位置上，接下来还需要同时将与该AppWindowToken对象所对应的WindowState对象移动至WindowManagerService服务内部的一个WindowState堆栈合适位置上去。

        移动对应的WindowState对象的操作同样也是分两步执行的：第一步先调用WindowManagerService类的成员函数tmpRemoveAppWindowsLocked来将这些WindowState对象从原来的WindowState堆栈位置移除；第二步再调用WindowManagerService类的成员函数reAddAppWindowsLocked来将这些WindowState对象插入到WindowState堆栈的合适位置去。

        对应的WindowState对象被移动到的合适位置是通过调用WindowManagerService类的成员函数findWindowOffsetLocked来获得的，它的实现如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

/\*\*

\* Z-ordered (bottom-most first) list of all Window objects.

\*/

**final** ArrayList<WindowState> mWindows = **new** ArrayList<WindowState>();

......

**private** **int** **findWindowOffsetLocked**(**int** tokenPos) {

**final** **int** NW = mWindows.size();

**if** (tokenPos >= mAppTokens.size()) {

**int** i = NW;

**while** (i > 0) {

i--;

WindowState win = mWindows.get(i);

**if** (win.getAppToken() != **null**) {

**return** i+1;

}

}

}

**while** (tokenPos > 0) {

// Find the first app token below the new position that has

// a window displayed.

**final** AppWindowToken wtoken = mAppTokens.get(tokenPos-1);

......

**if** (wtoken.sendingToBottom) {

......

tokenPos--;

**continue**;

}

**int** i = wtoken.windows.size();

**while** (i > 0) {

i--;

WindowState win = wtoken.windows.get(i);

**int** j = win.mChildWindows.size();

**while** (j > 0) {

j--;

WindowState cwin = win.mChildWindows.get(j);

**if** (cwin.mSubLayer >= 0) {

**for** (**int** pos=NW-1; pos>=0; pos--) {

**if** (mWindows.get(pos) == cwin) {

......

**return** pos+1;

}

}

}

}

**for** (**int** pos=NW-1; pos>=0; pos--) {

**if** (mWindows.get(pos) == win) {

......

**return** pos+1;

}

}

}

tokenPos--;

}

**return** 0;

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。 

        参数tokenPos描述的是一个AppWindowToken对象在WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList的位置，WindowManagerService类的成员函数findWindowOffsetLocked的目标就要找到与该AppWindowToken对象所对应的WindowState对象在WindowManagerService服务内部的一个WindowState堆栈的起始偏移位置。有了这个起始偏移位置之后，我们就可以将对应的所有WindowState对象有序地插入到该WindowState堆栈中去。WindowManagerService服务内部的WindowState堆栈是通过WindowManagerService类的成员变量mWindows来描述的。接下来我们就分两种情况来分析这个起始偏移位置的计算过程。

        第一种情况是参数tokenPos的值大于WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList的大小。这是一种异常情况，一般来说，参数tokenPos是指向mAppTokens列表的某一个位置的，不过这时候意味着它所描述的AppWindowToken对象的Z轴位置要大于mAppTokens列表的最上面的一个AppWindowToken对象的Z轴位置的。这也就是说，与参数tokenPos所描述的AppWindowToken对象所对应的WindowState对象的要位于与mAppTokens列表的最上面的一个AppWindowToken对象所对应的任一个WindoState对象的上面。因此，就需要找到与mAppTokens列表的最上面的一个AppWindowToken对象所对应的Z轴位置最大的一个WindoState对象在WindowState堆栈中的位置i，然后就可以知道与参数tokenPos所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈的起始偏移位置为i+1。

        如何找到mAppTokens列表的最上面的一个AppWindowToken对象所对应的Z轴位置最大的一个WindoState对象在WindowState堆栈中的位置i呢？从图1可以可得到一个结论：WindowManagerService服务内部中的所有WindowState对象都是按照Z轴从位置从小到大排列在WindowState堆栈中的，并且在mAppTokens列表中，位于上面的一个AppWindowToken对象所对应的那些WindowState对象的Z轴位置是一定大于位于下面的一个AppWindowToken对象所对应的那些WindowState对象的Z轴位置的。因此，我们只要从WindowState堆栈的顶端开始往下遍历，找到这样的一个WindowState对象，它是属于一个AppWindowToken对象的，即它的成员函数getAppToken的返回值不等于null，那么它在WindowState堆栈中的位置就是我们要找到的位置i。有了这个位置i之后，将它的值加上1，就可以得到参数t所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈的起始偏移位置了。

        第二种情况是参数tokenPos的值小于WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList的大小。根据前面得到的推论，我们只要在mAppTokens列表中找到一个AppWindowToken对象，它满足以下三个条件：

        A. 它在mAppTokens列表中的位置小于tokenPos；

        B. 它在WindowState堆栈中对应有WindowState对象；

        C. 它不是将要置于WindowState堆栈的底部。

        如果一个AppWindowToken对象在WindowState堆栈中对应有WindowState对象，那么这些WindowState对象也会同时按照Z轴从小到大的顺序保存它的成员变量windows所描述的一个ArrayList中，这意味着如果一个AppWindowToken对象满足条件B，那么它的成员变量windows所描述的一个ArrayList的大小就大于0。

        如果一个AppWindowToken对象不是将要置于WindowState堆栈的底部，那么它的成员变量sendingToBottom的值就不等于true，这也意味这个AppWindowToken对象满足条件C。

        如果能找到满足上述条件的一个AppWindowToken对象wtoken，那么我们只要找到与它所对应的Z轴位置最大的WindowState对象在WindowManagerService服务内部的WindowState堆栈中的位置i，那么将它的值加1，就可以得到与参数tokenPos所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈的起始偏移位置了。

        那么如何找到与这个AppWindowToken对象wtoken对应的Z轴位置最大的WindowState对象在WindowManagerService服务内部的WindowState堆栈中的位置i呢？从前面的图1可以知道，一个AppWindowToken对象所对应的WindowState对象可以划分为两种类型：第一种类型是父窗口类型的；第二种是子窗口类型的。如果一个WindowState对象所描述的窗口是父窗口，那么它的子窗口就保存在它的成员变量mChildWindows所描述的一个ArrayList中，并且这些子窗口是按照Z轴位置从小到大的顺序排列的，同时，该WindowState对象也会保存在与它所对应的一个AppWindowToken对象的成员变量windows所描述的一个ArrayList中。

        有了上述结论，并且假设存在一个能够满足上述三个条件的AppWindowToken对象wtoken，那么就可以从上到下遍历保存在它的成员变量windows所描述的一个ArrayList中的每一个WindowState对象win：

        I. 如果WindowState对象win所描述的一个窗口具有子窗口，那么就继续从上到下遍历这些子窗口，即从上到下遍历WindowState对象win的成员变量mChildWindows所描述的一个ArrayList。如果能找到一个WindowState对象cwin，它的成员变量mSubLayer的值大于等于0，那么该WindowState对象cwin在WindowManagerService服务内部的WindowState堆栈中的位置就是我们要得到的位置i。注意，如果WindowState对象cwin的成员变量mSubLayer的值小于0，那么它虽然是一个子窗口，但是它却是位于父窗口的后面的，即它的Z轴位置是小于父窗口的Z轴位置的。

        II. 如果WindowState对象win所描述的一个窗口不具有子窗口，即它的成员变量mChildWindows所描述的一个ArrayList的大小等于0，那么它在WindowManagerService服务内部的WindowState堆栈中的位置就是我们要得到的位置i。

        得到了位置i之后，将它的值加1，那么就可以得到与参数tokenPos所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈的起始偏移位置了。

        回到WindowManagerService类的成员函数moveAppToken中，调整好参数token所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈中的位置之后，即调用了成员函数reAddAppWindowsLocked之后，这时候系统中的窗口的布局就会发生了变化，即系统中的窗口的Z轴位置关系发生了变化，那么接下来就需要调用成员函数updateFocusedWindowLocked来重新计算系统中的窗口的Z轴位置，并且调用成员函数performLayoutAndPlaceSurfacesLocked来重新布局系统中的窗口。

        4. 移动AppWindowToken至顶端

        移动AppWindowToken至顶端是通过调用WindowManagerService类的成员函数moveAppTokensToTop来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**public** **void** **moveAppTokensToTop**(List<IBinder> tokens) {

**if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,

"moveAppTokensToTop()")) {

**throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");

}

**final** **long** origId = Binder.clearCallingIdentity();

**synchronized**(mWindowMap) {

removeAppTokensLocked(tokens);

**final** **int** N = tokens.size();

**for** (**int** i=0; i<N; i++) {

AppWindowToken wt = findAppWindowToken(tokens.get(i));

**if** (wt != **null**) {

mAppTokens.add(wt);

**if** (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {

mToTopApps.remove(wt);

mToBottomApps.remove(wt);

mToTopApps.add(wt);

wt.sendingToBottom = **false**;

wt.sendingToTop = **true**;

}

}

}

**if** (mNextAppTransition == WindowManagerPolicy.TRANSIT\_UNSET) {

moveAppWindowsLocked(tokens, mAppTokens.size());

}

}

Binder.restoreCallingIdentity(origId);

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        WindowManagerService类的成员函数moveAppTokensToTop可以同时将一组AppWindowToken移至顶端，同时需要调用者具有android.Manifest.permission.MANAGE\_APP\_TOKENS权限。

        参数tokens所描述的是一个IBinder接口列表，与这些IBinder接口所对应的AppWindowToken对象就是接下来要移至顶端的。在将保存在参数tokens中的IBinder接口所对应的AppWindowToken对象移至顶端之前，WindowManagerService类的成员函数首先会调用前面所描述的成员函数removeAppTokensLocked来删除这些AppWindowToken对象，然后再依次将它们添加到WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList的末尾去。

        注意，WindowManagerService类的成员变量mNextAppTransition用来描述系统当前是否正在切换Activity窗口。如果是的话，那么它的值就不等于WindowManagerPolicy.TRANSIT\_UNSET，这时候就需要：

       A. 将所有要移至顶端的AppWindowToken对象都保存在WindowManagerService类的另外一个成员变量mToTopApps所描述的一个ArrayList中去，并且将这些AppWindowToken对象的成员变量sendingToTop的值设置为true。

       B. 将所有要移至顶端的AppWindowToken对象所对应WindowState对象都移至WindowManagerService服务内部的一个WindowState堆栈的顶端去，这是通过调用另外一个成员函数moveAppWindowsLocked来实现的。

       执行完成上述两个操作之后，与要移至顶端的AppWindowToken对象所对应的窗口就会位于窗口堆栈的最上面了。

       5. 移动AppWindowToken至底端

       移动AppWindowToken至顶端是通过调用WindowManagerService类的成员函数moveAppTokensToBottom来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**public** **void** **moveAppTokensToBottom**(List<IBinder> tokens) {

**if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,

"moveAppTokensToBottom()")) {

**throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");

}

**final** **long** origId = Binder.clearCallingIdentity();

**synchronized**(mWindowMap) {

removeAppTokensLocked(tokens);

**final** **int** N = tokens.size();

**int** pos = 0;

**for** (**int** i=0; i<N; i++) {

AppWindowToken wt = findAppWindowToken(tokens.get(i));

**if** (wt != **null**) {

mAppTokens.add(pos, wt);

**if** (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {

mToTopApps.remove(wt);

mToBottomApps.remove(wt);

mToBottomApps.add(i, wt);

wt.sendingToTop = **false**;

wt.sendingToBottom = **true**;

}

pos++;

}

}

**if** (mNextAppTransition == WindowManagerPolicy.TRANSIT\_UNSET) {

moveAppWindowsLocked(tokens, 0);

}

}

Binder.restoreCallingIdentity(origId);

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。 

        WindowManagerService类的成员函数moveAppTokensToBottom可以同时将一组AppWindowToken移至底端。将一组AppWindowToken移至底端与将一组AppWindowToken移至顶端的实现是类似的，只不过是移动的方向相反而已。因此，WindowManagerService类的成员函数moveAppTokensToBottom的实现可以参考前面所分析的成员函数moveAppTokensToTop的实现，这里不再详述。

        6. 增加WindowToken

        从图1可以知道，如果一个WindowState对象不是与一个AppWindowToken对象对应的，那么它就必须要与一个WindowToken对象对应。例如，用来描述输入法窗口和壁纸窗口的WindowState对象对应的就是WindowToken对象，而不是AppWindowToken对象，因为它们不是Activity类型的窗口。

        输入法窗口和壁纸窗口分别是由输入法管理服务InputMethodManagerService和壁纸管理服务WallpaperManagerService调用WindowManagerService类的成员函数addWindowToken来增加对应的WindowToken对象的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**public** **void** **addWindowToken**(IBinder token, **int** type) {

**if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,

"addWindowToken()")) {

**throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");

}

**synchronized**(mWindowMap) {

WindowToken wtoken = mTokenMap.get(token);

**if** (wtoken != **null**) {

Slog.w(TAG, "Attempted to add existing input method token: " + token);

**return**;

}

wtoken = **new** WindowToken(token, type, **true**);

mTokenMap.put(token, wtoken);

mTokenList.add(wtoken);

**if** (type == TYPE\_WALLPAPER) {

mWallpaperTokens.add(wtoken);

}

}

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        调用WindowManagerService类的成员函数addWindowToken需要具有android.Manifest.permission.MANAGE\_APP\_TOKENS权限。

        对于输入法窗口和壁纸窗口来说，参数token指向的是与它们所关联的一个Binder对象的IBinder接口，而参数type描述的是要在WindowManagerService服务内部增加WindowToken对象的窗口的类型。

        WindowManagerService类的成员函数addWindowToken首先检查在成员变量mTokenMap所描述的一个HashMap检查是否已经存在一个WindowToken对象与参数token对应。如果已经存在的话，那么WindowManagerService类的成员函数addWindowToken就什么也不做就返回了，否则的话，就会使用参数token来创建一个WindowToken对象，并且会将该WindowToken对象分别保存在WindowManagerService类的成员变量mTokenMap和mTokenList中。

        这里有两个地方需要注意：

        A. 由于这里增加的是WindowToken对象，而不是AppWindowToken对象，因此，与增加AppWindowToken不同，这里不需要将新创建的WindowToken对象保存在WindowManagerService类的成员变量mAppTokens中。

        B. 如果参数type的值等于TYPE\_WALLPAPER，那么就意味着新创建的WindowToken对象是用来描述壁纸窗口的，这时候还需要将新创建的WindowToken对象保存在WindowManagerService类的成员变量mWallpaperTokens所描述的一个ArrayList中，以方便管理壁纸窗口。

        对于非输入法窗口、非壁纸窗口以及非Activity窗口来说，它们所对应的WindowToken对象是在它们增加到WindowManagerService服务的时候创建的。从前面[Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析](http://blog.csdn.net/luoshengyang/article/details/8275938)一文可以知道，增加一个窗口WindowManagerService服务最终是通过调用WindowManagerService类的成员函数addWindow来实现的，接下来我们就主要分析与创建WindowToken相关的逻辑，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**public** **int** **addWindow**(Session session, IWindow client,

WindowManager.LayoutParams attrs, **int** viewVisibility,

Rect outContentInsets, InputChannel outInputChannel) {

......

**synchronized**(mWindowMap) {

......

**boolean** addToken = **false**;

WindowToken token = mTokenMap.get(attrs.token);

**if** (token == **null**) {

**if** (attrs.type >= FIRST\_APPLICATION\_WINDOW

&& attrs.type <= LAST\_APPLICATION\_WINDOW) {

......

**return** WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;

}

**if** (attrs.type == TYPE\_INPUT\_METHOD) {

......

**return** WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;

}

**if** (attrs.type == TYPE\_WALLPAPER) {

......

**return** WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;

}

token = **new** WindowToken(attrs.token, -1, **false**);

addToken = **true**;

}

......

**if** (addToken) {

mTokenMap.put(attrs.token, token);

mTokenList.add(token);

}

......

}

......

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        如果参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量token所指向的一个IBinder接口在WindowManagerService类的成员变量mTokenMap所描述的一个HashMap中没有一个对应的WindowToken对象，并且该WindowManager.LayoutParams对象的成员变量type的值不等于TYPE\_INPUT\_METHOD、TYPE\_WALLPAPER，以及不在FIRST\_APPLICATION\_WINDOW和LAST\_APPLICATION\_WINDOW，那么就意味着这时候要增加的窗口就既不是输入法窗口，也不是壁纸窗口和Activity窗口，因此，就需要以参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量token所指向的一个IBinder接口为参数来创建一个WindowToken对象，并且将该WindowToken对象保存在WindowManagerService类的成员变量mTokenMap和mTokenList中。

        7. 删除WindowToken

        删除WindowToken是通过调用WindowManagerService类的成员函数removeWindowToken来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**public** **void** **removeWindowToken**(IBinder token) {

**if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,

"removeWindowToken()")) {

**throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");

}

**final** **long** origId = Binder.clearCallingIdentity();

**synchronized**(mWindowMap) {

WindowToken wtoken = mTokenMap.remove(token);

mTokenList.remove(wtoken);

**if** (wtoken != **null**) {

**boolean** delayed = **false**;

**if** (!wtoken.hidden) {

wtoken.hidden = **true**;

**final** **int** N = wtoken.windows.size();

**boolean** changed = **false**;

**for** (**int** i=0; i<N; i++) {

WindowState win = wtoken.windows.get(i);

**if** (win.isAnimating()) {

delayed = **true**;

}

**if** (win.isVisibleNow()) {

applyAnimationLocked(win,

WindowManagerPolicy.TRANSIT\_EXIT, **false**);

changed = **true**;

}

}

**if** (changed) {

mLayoutNeeded = **true**;

performLayoutAndPlaceSurfacesLocked();

updateFocusedWindowLocked(UPDATE\_FOCUS\_NORMAL);

}

**if** (delayed) {

mExitingTokens.add(wtoken);

} **else** **if** (wtoken.windowType == TYPE\_WALLPAPER) {

mWallpaperTokens.remove(wtoken);

}

}

......

} **else** {

Slog.w(TAG, "Attempted to remove non-existing token: " + token);

}

}

Binder.restoreCallingIdentity(origId);

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        调用WindowManagerService类的成员函数removeWindowToken需要具有android.Manifest.permission.MANAGE\_APP\_TOKENS权限。

        WindowManagerService类的成员函数removeWindowToken首先找到与参数token所描述的Binder接口所对应的WindowToken对象，接着再将该WindowToken对象从WindowManagerService类的成员变量mTokenMap和mTokenList中删除。

        删除了一个WindowToken对象之后，如果该WindowToken对象不是处于不可见的状态，即它的成员变量hidden的值不等于false，那么就意味着它所描述窗口口也有可能是可见的，那么WindowManagerService类的成员函数removeWindowToken就需要作以下两个检查：

        A. 如果该WindowToken对象所描述的窗口的其中一个处于动画显示过程，即用来描述该窗口的一个WindowState对象的成员函数isAnimating的返回值等于true，那么就需要该WindowToken对象的状态设置为正在退出状态，即将它保存在WindowManagerService类的成员变量mExitingTokens所描述的一个ArrayList中。

        B. 如果该WindowToken对象所描述的窗口是可见的，即用来描述该窗口的一个WindowState对象的成员函数isVisibleNow的返回值等于true，那么就需要调用WindowManagerService类的成员函数applyAnimationLocked来给它应用一个退出动画，该退出动画是通过调用WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked来实现的。当一个窗口退出了之后，系统当前获得焦点的窗口可能会发生变化，这时候就需要调用WindowManagerService类的成员函数updateFocusedWindowLocked来重新调整系统当前获得焦点的窗口。

        注意，如果正在删除的WindowToken对象是用来描述壁纸窗口的，那么还需要将该WindowToken对象从WindowManagerService类的成员变量mWallpaperTokens所描述的一个ArrayList中删除。

        8. 增加WindowState

        从前面[Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析](http://blog.csdn.net/luoshengyang/article/details/8275938)一文可以知道，增加一个窗口WindowManagerService服务最终是通过调用WindowManagerService类的成员函数addWindow来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

/\*\*

\* Mapping from an IWindow IBinder to the server's Window object.

\* This is also used as the lock for all of our state.

\*/

**final** HashMap<IBinder, WindowState> mWindowMap = **new** HashMap<IBinder, WindowState>();

......

/\*\*

\* Z-ordered (bottom-most first) list of all Window objects.

\*/

**final** ArrayList<WindowState> mWindows = **new** ArrayList<WindowState>();

......

**public** **int** **addWindow**(Session session, IWindow client,

WindowManager.LayoutParams attrs, **int** viewVisibility,

Rect outContentInsets, InputChannel outInputChannel) {

......

WindowState win = **null**;

**synchronized**(mWindowMap) {

......

win = **new** WindowState(session, client, token,

attachedWindow, attrs, viewVisibility);

......

mWindowMap.put(client.asBinder(), win);

......

**if** (attrs.type == TYPE\_INPUT\_METHOD) {

mInputMethodWindow = win;

addInputMethodWindowToListLocked(win);

......

} **else** **if** (attrs.type == TYPE\_INPUT\_METHOD\_DIALOG) {

mInputMethodDialogs.add(win);

addWindowToListInOrderLocked(win, **true**);

adjustInputMethodDialogsLocked();

......

} **else** {

addWindowToListInOrderLocked(win, **true**);

**if** (attrs.type == TYPE\_WALLPAPER) {

.......

adjustWallpaperWindowsLocked();

} **else** **if** ((attrs.flags&FLAG\_SHOW\_WALLPAPER) != 0) {

adjustWallpaperWindowsLocked();

}

}

......

}

......

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        WindowManagerService类有两个成员变量mWindowMap和mWindows是用来保存系统中的WindowState对象。其中，成员变量mWindowMap指向的是一个HashMap，它的关键字是一个IBinder接口，一般这个IBinder接口指向的是一个Binder代理对象，引用了运行在应用程序进程这一侧的一个类型为W的Binder本地对象，用来描述一个窗口；成员变量mWindows指向的是一个ArrayList，保存在它里面的WindowState对象是按照其Z轴位置从小到大的顺序排列的。成员变量mWindowMap和mWindows的区别在于，前者给在定一个IBinder接口的情况下，可以快速找到与对应的WindowState对象，而后者用来从上到下或者下到上遍历系统的WindowState对象。由于系统中的WindowState对象是按照其Z轴位置从小到大的顺序排列在成员变量mWindows中的，因此，成员变量mWindows所指向的ArrayList就是我们在前面图1中所说的Window Stack。

        理解了WindowManagerService类有两个成员变量mWindowMap和mWindows的作用之后，WindowManagerService类的成员函数addWindow增加一个WindowState对象的过程就容易理解了。

       参数client是一个Binder代理对象，引用了运行在应用程序进程这一侧的一个类型为W的Binder本地对象，用来描述要增加到WindowManagerService服务中的一个窗口。WindowManagerService类的成员函数addWindow首先创建一个WindowState对象win，接着再以参数client所描述的一个Binder代理对象的IBinder接口为关键字，将WindowState对象win保存在WindowManagerService类的成员变量mWindowMap中，最后还会根据要增加到WindowManagerService服务中的窗口的类型来调用不同的成员函数将WindowState对象win增加到WindowManagerService类的成员变量mWindows中：

        A. 如果要增加的是输入法窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_INPUT\_METHOD，那么就会调用成员函数addInputMethodWindowToListLocked来将WindowState对象win增加到WindowManagerService类的成员变量mWindows中去，并且会将WindowState对象win保存在WindowManagerService类的成员变量mInputMethodWindow中。

        B. 如果要增加的是输入法对话框，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_INPUT\_METHOD\_DIALOG，那么就会调用成员函数addWindowToListInOrderLocked来将WindowState对象win增加到WindowManagerService类的成员变量mWindows中去，并且会将WindowState对象win保存在WindowManagerService类的成员变量mInputMethodDialogs中，以及调用成员函数adjustInputMethodDialogsLocked来调整刚才所添加的输入法窗口在窗口堆栈中的位置，使得它位于系统当前需要输入法窗口的窗口的上面。

        C.  如果要增加的是壁纸窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_WALLPAPER，那么就会调用成员函数addWindowToListInOrderLocked来将WindowState对象win增加到WindowManagerService类的成员变量mWindows中去，并且会调用成员函数adjustWallpaperWindowsLocked来调整刚才所添加的壁纸窗口在窗口堆栈中的位置，使得它位于系统当前需要壁纸窗口的窗口的下面。

        D . 如果要增加的既不是输入法窗口，也不是输入法对话框和壁纸窗口，那么就只会调用成员函数addWindowToListInOrderLocked来将WindowState对象win增加到WindowManagerService类的成员变量mWindows中去，但是如果要增加的窗口需要显示壁纸，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_SHOW\_WALLPAPER位等于1，那么还会继续调用成员函数adjustWallpaperWindowsLocked来调整系统中的壁纸窗口在窗口堆栈中的位置，使得它位于刚才所添加的窗口的下面。

        在后面的两篇文章中，我们再详细分析WindowManagerService类的成员函数addInputMethodWindowToListLocked、adjustInputMethodDialogsLocked和adjustWallpaperWindowsLocked的实现，其中，前两者是与输入法窗口相关的，而后者是与壁纸窗口相关的。本文主要关注WindowManagerService类的成员函数addWindowToListInOrderLocked的实现，它会将一个指定的WindowState对象增加到窗口堆栈中的合适位置上去。

推荐：[Android窗口管理服务WindowManagerService显示窗口动画的原理分析](http://www.itboth.com/d/Q7BzM3/android)

[ 在前一文中，我们分析了Activity组件的切换过程。从这个过程可以知道，所有参与切换操作的窗口都会被设置切换动画。事实上，一个窗口在打开（关闭）的过程中，除了可

        9. 增加WindowState到窗口堆栈

        从前面的分析可以知道，将一个WindowState对象增加到WindowManagerService服务内部中的窗口堆栈，即WindowManagerService类的成员变量mWindows，是通过调用WindowManagerService类的成员函数addWindowToListInOrderLocked来实现的。

       WindowManagerService类的成员函数addWindowToListInOrderLocked的实现比较复杂，我们先列出它的框架，然后再详细分析它的实现，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **void** **addWindowToListInOrderLocked**(WindowState win, **boolean** addToToken) {

**final** IWindow client = win.mClient;

**final** WindowToken token = win.mToken;

**final** ArrayList<WindowState> localmWindows = mWindows;

**final** **int** N = localmWindows.size();

**final** WindowState attached = win.mAttachedWindow;

**int** i;

**if** (attached == **null**) {

//CASE 1：要增加的窗口win没有附加在其它窗口上

**int** tokenWindowsPos = token.windows.size();

**if** (token.appWindowToken != **null**) {

//CASE 1.1：要增加的窗口win是一个Activity窗口

**int** index = tokenWindowsPos-1;

**if** (index >= 0) {

//CASE 1.1.1：用来要增加的窗口win的令牌token已存在其它窗口

......

} **else** {

//CASE 1.1.2：用来要增加的窗口win的令牌token尚未存在任何窗口

......

}

} **else** {

//CASE 1.2：要增加的窗口win不是一个Activity窗口

......

}

**if** (addToToken) {

token.windows.add(tokenWindowsPos, win);

}

} **else** {

//CASE 2：要增加的窗口win附加在窗口attached上

......

}

**if** (win.mAppToken != **null** && addToToken) {

win.mAppToken.allAppWindows.add(win);

}

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        我们首先分析一下WindowManagerService类的成员函数addWindowToListInOrderLocked的几个本地变量的含义：

        A. **token**。本地变量token指向的是参数win所描述的一个WindowState对象的成员变量mToken所指向一个WindowToken对象，这个WindowToken对象用来描述WindowState对象win所对应的窗口令牌。

        B.**localmWindows**。本地变量localmWindows指向的是WindowManagerService类的成员变量mWindows所描述的一个ArrayList，即一个窗口堆栈，WindowManagerService类的成员函数addWindowToListInOrderLocked的目标就是要将参数win所描述的一个WindowState对象增加到该窗口堆栈的合适位置上去。

        C. **attached**。本地变量attached指向的是参数win所描述的一个WindowState对象的成员变量mAttachedWindow 所指向的一个WindowState对象，如果它的值不等于null，那么就意味参数win所描述的窗口要附加在本地变量attached所描述的窗口上。

        D. **tokenWindowsPos**。本地变量tokenWindowsPos用来描述与窗口令牌token所对应的窗口的数量。

        E. **token.appWindowToken**。从前面[Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析](http://blog.csdn.net/luoshengyang/article/details/8275938)一文可以知道，如果一个WindowToken对象的成员变量appWindowToken的值不等于null，那么就意味着该WindowToken对象的实际类型为是AppWindowToken，即它所描述的是一个Activity窗口令牌，这种类型的令牌的特点是在ActivityManagerService服务的Activity组件堆栈中对应有一个ActivityRecord对象，如图1所示。

        F. **index**。本地变量index的值等于tokenWindowsPos-1，如果它的值大于等于0，那么就意味着窗口令牌tokent已经存在其它窗口，否则的话，就意味着窗口令牌tokent尚未存在任何窗口。

        从这些本地变量的含义，我们就可以分情况来将参数win所描述的一个WindowState对象增加到WindowManagerService服务内部的窗口堆栈的合适位置上去：

**CASE 1**：要增加的窗口win没有附加在其它窗口上

**----CASE 1.1**：要增加的窗口win是一个Activity窗口

**----CASE 1.1.1**：用来要增加的窗口win的令牌token已存在其它窗口。这时候意味着窗口win需要保存在其它已经存在的窗口的附近，因此，我们只要找到这些已经存在的窗口在窗口堆栈中的位置，那么再根据其它属性，就可以将窗口win保存在已经存在的窗口的上面或者下面。

**----CASE 1.1.2**：用来要增加的窗口win的令牌token尚未存在任何窗口。虽然这时候窗口win在窗口堆栈中没有位置可以参考，但是它毕竟是一个Activity窗口，我们可以通过与它所对应的AppWindowToken对象在App Token List（即WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList）中的位置来获得它窗口堆栈中的位置。回忆我们在前面第3节分析移动AppWindowToken至指定位置的操作时得到的结论：WindowManagerService服务内部中的所有WindowState对象都是按照Z轴从位置从小到大排列在WindowState堆栈中的，并且在mAppTokens列表中，位于上面的一个AppWindowToken对象所对应的那些WindowState对象的Z轴位置是一定大于位于下面的一个AppWindowToken对象所对应的那些WindowState对象的Z轴位置的。因此，我们只要找到用来描述窗口win的一个AppWindowToken对象（token.appWindowToken）的上一个或者下一个AppWindowToken对象所对应的窗口在窗口堆栈中的位置，那么就可以这个位置为参考，得到窗口win在窗口堆栈中的位置。

**----CASE 1.2**：要增加的窗口win不是一个Activity窗口。这时候既然要增加的窗口也没有附加在其它窗口上，那么就意味着要增加的窗口win在窗口堆栈中没有位置可以参考，因此，我们就需要根据它的Z轴位置来决定它在窗口堆栈的位置。

**CASE 2**：要增加的窗口win附加在窗口attached上。这时候就意味着要增加的窗口win要保存在窗口attached的上面，即窗口在窗口堆栈的位置要以窗口attached在窗口堆栈的位置为参考。

        从上面的分析就可以知道，**CASE 1.1.1**、**CASE 1.1.2**和**CASE 2**都有一个共同特点，即要增加的窗口win在窗口堆栈的位置有一个参考值，而在**CASE 1.2**中，要增加的窗口win在窗口堆栈的位置没有参考值，需要通过其Z轴位置来确定。

       在分析上述四种情况之前， 我们还需要再说明一下WindowManagerService类的成员函数addWindowToListInOrderLocked的参数addToToken的含义。参数addToToken是一个布尔变量，如果它的值等于true，那么就说明需要将参数win所描述的一个WindowState对象添加用来描述它的窗口令牌token的成员变量windows所描述的一个ArrayList中去。注意，窗口令牌token的成员变量windows所描述的一个ArrayList里面所保存的WindowState对象是按照Z轴位置从小到大的顺序来排列的，因此，在将WindowState对象win保存到这个ArrayList之前，首先要按照它的Z轴位置计算得到它在这个ArrayList中的位置tokenWindowsPos。另一方面，在参数addToToken的值等于true，并且参数win所描述的是一个Activity窗口，即它的成员变量mAppToken不等于null的情况下，还需要将参数win所描述的一个WindowState对象保存在用来描述它的窗口令牌，即一个AppWindowToken对象成员变量allAppWindows所描述的一个ArrayList中去，以便可以知道一个AppWindowToken对象对应的Activity窗口都有哪些。

       接下来，我们就分别分析这四种情况是如何将窗口win增加窗口堆栈中去的。

**CASE 1.1.1**对应的代码为：

**if** (win.mAttrs.type == TYPE\_BASE\_APPLICATION) {

// Base windows go behind everything else.

placeWindowBefore(token.windows.**get**(0), win);

tokenWindowsPos = 0;

} **else** {

AppWindowToken atoken = win.mAppToken;

**if** (atoken != **null** &&

token.windows.**get**(index) == atoken.startingWindow) {

placeWindowBefore(token.windows.**get**(index), win);

tokenWindowsPos--;

} **else** {

**int** newIdx = findIdxBasedOnAppTokens(win);

**if**(newIdx != -1) {

//there is a window above this one associated with the same

//apptoken note that the window could be a floating window

//that was created later or a window at the top of the list of

//windows associated with this token.

......

localmWindows.add(newIdx+1, win);

mWindowsChanged = **true**;

}

}

}

        这段代码又分为三种情况来将参数win所描述的一个WindowState对象添加到窗口堆栈中：

        A. 参数win描述的窗口的类型为TYPE\_BASE\_APPLICATION。在一个令牌对应的所有窗口中，类型为TYPE\_BASE\_APPLICATION的窗口位于其它类型的窗口的下面。因此，这段代码就会调用WindowManagerService类的成员函数placeWindowBefore来将参数win所描述的一个WindowState对象保存窗口堆栈中，并且它是位于令牌token的窗口列表的第0个位置的WindowState对象的下面。这时候变量tokenWindowsPos的值会被设置为0，表示参数win所描述的一个WindowState对象要保存窗口令牌token的窗口列表的第0个位置上。

        B. 参数win描述的一个WindowState对象的成员变量mAppToken的值不等于null，这意味着参数win描述的是一个Activity窗口，这时候如果窗口令牌atoken（与token描述的是同一个窗口令牌）的窗口列表的第index个位置（即最上面的一个位置） 的WindowState对象描述的是一个Activity启动窗口，即与窗口令牌atoken的成员变量startingWindow描述的是同一个窗口，那么就说明窗口令牌atoken的窗口列表的第index个位置的WindowState对象描述的是窗口win的启动窗口。由于一个窗口的启动窗口总是位于它的上面，因此，这段代码就会调用WindowManagerService类的成员函数placeWindowBefore来将参数win所描述的一个WindowState对象保存窗口堆栈中，并且它是位于令牌atoken的窗口列表的第index个位置的WindowState对象的下面。这时候变量tokenWindowsPos的值减少1，即相当于是等于index，表示参数win所描述的一个WindowState对象要插入在窗口令牌token的窗口列表的第index个位置上。

        C. 参数win所描述的窗口的类型既不是TYPE\_BASE\_APPLICATION，而且它也没有启动窗口，那么这时候就需要将它保存在窗口令牌token的窗口列表的最上面一个窗口的上面。窗口令牌token的窗口列表的最上面一个窗口在窗口堆栈中的位置newIdx是通过调用WindowManagerService类的成员函数findIdxBaseOnAppTokens来获得的，这时候参数win所描述的一个WindowState对象就应该保存在窗口堆栈，即变量localmWindows所描述的一个ArrayList的第newIdx+1个位置上。

**CASE 1.1.2**对应的代码为：

// Figure out where the window should go, based on the

// order of applications.

final **int** NA = mAppTokens.size();

WindowState pos = **null**;

**for** (i=NA-1; i>=0; i--) {

AppWindowToken t = mAppTokens.**get**(i);

**if** (t == token) {

i--;

**break**;

}

// We haven't reached the token yet; if this token

// is not going to the bottom and has windows, we can

// use it as an anchor for when we do reach the token.

**if** (!t.sendingToBottom && t.windows.size() > 0) {

pos = t.windows.**get**(0);

}

}

// We now know the index into the apps. If we found

// an app window above, that gives us the position; else

// we need to look some more.

**if** (pos != **null**) {

// Move behind any windows attached to this one.

WindowToken atoken = mTokenMap.**get**(pos.mClient.asBinder());

**if** (atoken != **null**) {

final **int** NC = atoken.windows.size();

**if** (NC > 0) {

WindowState bottom = atoken.windows.**get**(0);

**if** (bottom.mSubLayer < 0) {

pos = bottom;

}

}

}

placeWindowBefore(pos, win);

} **else** {

// Continue looking down until we find the first

// token that has windows.

**while** (i >= 0) {

AppWindowToken t = mAppTokens.**get**(i);

final **int** NW = t.windows.size();

**if** (NW > 0) {

pos = t.windows.**get**(NW-1);

**break**;

}

i--;

}

**if** (pos != **null**) {

// Move in front of any windows attached to this

// one.

WindowToken atoken = mTokenMap.**get**(pos.mClient.asBinder());

**if** (atoken != **null**) {

final **int** NC = atoken.windows.size();

**if** (NC > 0) {

WindowState top = atoken.windows.**get**(NC-1);

**if** (top.mSubLayer >= 0) {

pos = top;

}

}

}

placeWindowAfter(pos, win);

placeWindowAfter(pos, win);

} **else** {

// Just search for the start of this layer.

final **int** myLayer = win.mBaseLayer;

**for** (i=0; i<N; i++) {

WindowState w = localmWindows.**get**(i);

**if** (w.mBaseLayer > myLayer) {

**break**;

}

}

......

localmWindows.add(i, win);

mWindowsChanged = **true**;

}

}

        这段代码要能冠军WindowManagerService服务内部的一个AppWindowToken列表mAppTokens来在窗口堆栈中找到一个参数位置来保存参数win所描述的一个WindowState对象。

        最上面的一个for循环执行完成之后，我们假设变量pos的值不等于null，这时候它与变量i以及变量token的关系如图2所示：

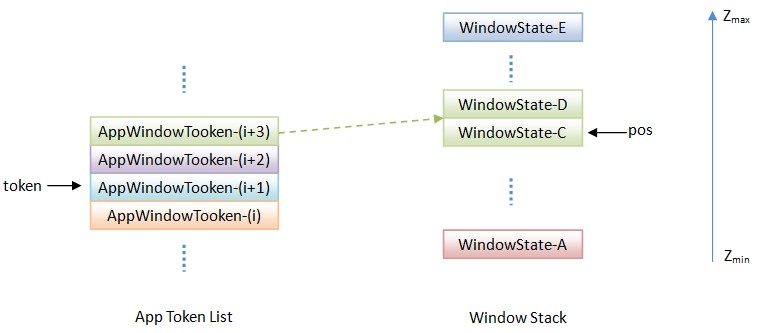


图2 窗口win位于窗口C的下面

        这时候位于令牌token上面的令牌在窗口堆栈中对应有WindowState对象。注意，这时候第i+2个令牌在窗口堆栈中不对应有WindowState对象，而第i+3个令牌在窗口堆栈中对应有C和D两个WindowState对象，并且这两个WindowState对象所描述的窗口都不是即将要切换到窗口堆栈的底部的。由于第i+3个令牌位于令牌token的上面，并且这两个令牌之间的其它令牌在窗口堆栈中不对应有WindowState对象，因此，这时候参数win所描述的WindowState对象在窗口堆栈中的位置应该以第i+3个令牌所对应的Z轴位置最小的WindowState对象在窗口堆栈中的位置为参考，即以WindowState对象C在窗口堆栈中的位置为参考，而WindowState对象C也正好是变量pos所指向的WindowState对象。

       接下来，上述代码会继续检查WindowState对象C是否附加有SubLayer值小于0的窗口。如果有的话，那么就会将变量pos指向SubLayer值最小的那个WindowState对象，这是因为该WindowState对象是在WindowState对象C的最下面的，并且它与WindowState对象C是同属一个令牌的。最后，上述代码就会调用WindowManagerService类的成员函数placeWindowBefore来将参数win所描述的一个WindowState对象保存窗口堆栈中由变量pos所指向的那个WindowState对象的下面。

        假设最上面的一个for循环执行完成之后，变量pos的值等于null，那么就说明位于令牌token上面的令牌在窗口堆栈中都没有对应有WindowState对象，或者说它们所对应的WindowState对象都是即将要切换到窗口堆栈的底部去的，这时候就需要通过位于令牌token上面的令牌来在窗口堆栈中找到一个参考位置来保存参数win所描述的WindowState对象，这是通过中间的while循环来实现的。

       中间的while循环执行完成之后，假设变量pos的值不等于null，这时候它与变量i以及变量token的关系如图3所示：

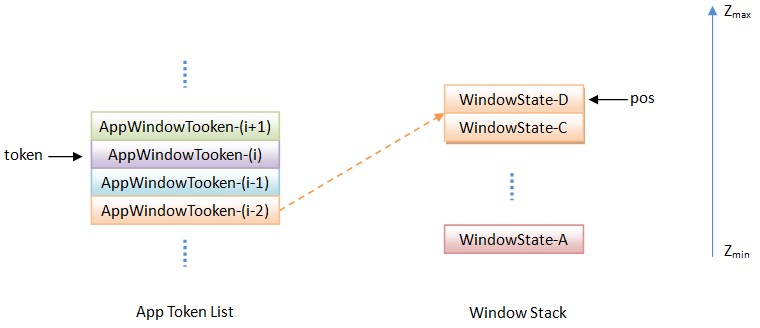


图3 窗口win位于窗口D的上面

        这时候位于令牌token上面的令牌在窗口堆栈中没有对应有WindowState对象。注意，这时候第i-1个令牌在窗口堆栈中不对应有WindowState对象，而第i-2个令牌在窗口堆栈中对应有C和D两个WindowState对象。由于第i-2个令牌位于令牌token的下面，并且这两个令牌之间的其它令牌在窗口堆栈中不对应有WindowState对象，因此，这时候参数win所描述的WindowState对象在窗口堆栈中的位置应该以第i-2个令牌所对应的Z轴位置最大的WindowState对象在窗口堆栈中的位置为参考，即以WindowState对象D在窗口堆栈中的位置为参考，而WindowState对象D也正好是变量pos所指向的WindowState对象。

        接下来，上述代码会继续检查WindowState对象D是否附加有SubLayer值大于等于0的窗口。如果有的话，那么就会将变量pos指向SubLayer值最大的那个WindowState对象，这是因为该WindowState对象是在WindowState对象D的最上面的，并且它与WindowState对象D是同属一个令牌的。最后，上述代码就会调用WindowManagerService类的成员函数placeWindowAfter来将参数win所描述的一个WindowState对象保存窗口堆栈中由变量pos所指向的那个WindowState对象的上面。

         假设中间的while循环执行完成之后，变量pos的值等于null，这时候就说明在窗口堆栈中实在是找不到参考位置来保存参数win所描述的WindowState对象了，因此，就只能通过参数win所描述的WindowState对象的Z轴位置，即它的成员变量mBaseLayer的值来在窗口堆栈中找到一个合适的位置了，如最下面的for循环所示。由于窗口堆栈中的WindowState对象是按照它们的Z轴位置由小到大的顺序来排列的，因此，最下面的for循环只要从下到上找到一个Z轴位置比参数win所描述的WindowState对象的Z轴位置大的一个WindowState对象在窗口堆栈中的位置i，那么就可以将参数win所描述的WindowState对象插入在窗口堆栈的第i个位置上了。

**CASE 1.2**对应的代码为：

// Figure out where window should go, based on layer.

final **int** myLayer = win.mBaseLayer;

**for** (i=N-1; i>=0; i--) {

**if** (localmWindows.**get**(i).mBaseLayer <= myLayer) {

i++;

**break**;

}

}

**if** (i < 0) i = 0;

......

localmWindows.add(i, win);

mWindowsChanged = **true**;

        由于这时候在窗口堆栈中是没有参考位置来保存参数win所描述的WindowState对象的，因此，这段代码就只能通过参数win所描述的WindowState对象的Z轴位置，即它的成员变量mBaseLayer的值来在窗口堆栈中找到一个合适的位置了，如这段代码中的for循环所示。由于窗口堆栈中的WindowState对象是按照它们的Z轴位置由小到大的顺序来排列的，因此，这段代码中的for循环只要从上到下找到一个WindowState对象，它的Z轴位置小于或者等于参数win所描述的WindowState对象的Z轴位置，那么该WindowState对象在窗口堆栈中的位置i就可以用插入参数win所描述的WindowState对象了。 

**CASE 2**对应的代码为：

// Figure out this window's ordering relative to the window

// it is attached to.

**final** **int** NA = token.windows.size();

**final** **int** sublayer = win.mSubLayer;

**int** largestSublayer = Integer.MIN\_VALUE;

WindowState windowWithLargestSublayer = **null**;

**for** (i=0; i<NA; i++) {

WindowState w = token.windows.get(i);

**final** **int** wSublayer = w.mSubLayer;

**if** (wSublayer >= largestSublayer) {

largestSublayer = wSublayer;

windowWithLargestSublayer = w;

}

**if** (sublayer < 0) {

// For negative sublayers, we go below all windows

// in the same sublayer.

**if** (wSublayer >= sublayer) {

**if** (addToToken) {

token.windows.add(i, win);

}

placeWindowBefore(

wSublayer >= 0 ? attached : w, win);

**break**;

}

} **else** {

// For positive sublayers, we go above all windows

// in the same sublayer.

**if** (wSublayer > sublayer) {

**if** (addToToken) {

token.windows.add(i, win);

}

placeWindowBefore(w, win);

**break**;

}

}

}

**if** (i >= NA) {

**if** (addToToken) {

token.windows.add(win);

}

**if** (sublayer < 0) {

placeWindowBefore(attached, win);

} **else** {

placeWindowAfter(largestSublayer >= 0

? windowWithLargestSublayer

: attached,

win);

}

}

        这段代码要将参数win所描述的WindowState对象附加在变量attached所描述的WindowState对象的上面或者下面，取决于它的成员变量mSubLayer的值是大于0还是小于0。我们分四种情况来考虑。

        第一种情况是参数win所描述的WindowState对象的成员变量mSubLayer的值小于0，并且这时候在附加在窗口attached的WindowState对象中，存在一个WindowState对象，它的成员变量mSubLayer的值大于等于参数win所描述的WindowState对象的成员变量mSubLayer的值，如图4和图5所示：

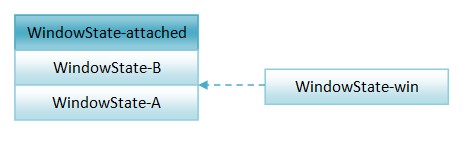


图4 窗口win插入到窗口B的下面

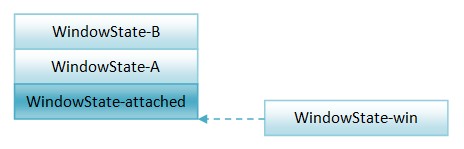


图5 窗口win插入在窗口attached的下面

        在图4和图5中，WindowState对象A和B均是附加在WindowState对象attached中。

        在图4中，WindowState对象A和B的成员变量mSubLayer的值均小于0，而WindowState对象win的成员变量mSubLayer的值比WindowState对象A的大，但是比WindowState对象B的小，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象B的下面，这是通过调用WindowManagerService类的成员函数placeWindowBefore来实现的。

        在图5中，WindowState对象A和B的成员变量mSubLayer的值均大于0，由于WindowState对象win的成员变量mSubLayer的值小于0，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象attached的下面，这是通过调用WindowManagerService类的成员函数placeWindowBefore来实现的。

        第二种情况是参数win所描述的WindowState对象的成员变量mSubLayer的值大于0，并且这时候在附加在窗口attached的WindowState对象中，存在一个WindowState对象，它的成员变量mSubLayer的值大于参数win所描述的WindowState对象的成员变量mSubLayer的值，如图6所示：

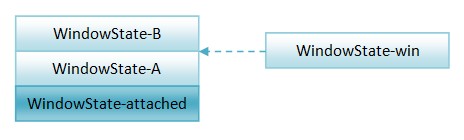


图6 窗口win插入在窗口B的下面

        在图6中，WindowState对象A和B均是附加在WindowState对象attached中。其中，WindowState对象A和B的成员变量mSubLayer的值均大于0，而WindowState对象win的成员变量mSubLayer的值比WindowState对象A的大，但是比WindowState对象B的小，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象B的下面，这是通过调用WindowManagerService类的成员函数placeWindowBefore来实现的。

        第三种情况是参数win所描述的WindowState对象的成员变量mSubLayer的值小于0，但是在附加在窗口attached的WindowState对象中，找不到一个WindowState对象，它的成员变量mSubLayer的值比WindowState对象的成员变量mSubLayer的值大，如图7所示：

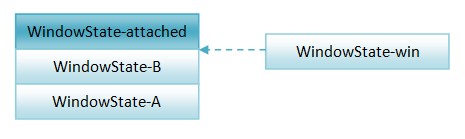


图7 窗口win插入在窗口attached的下面

        在图7中，WindowState对象A和B均是附加在WindowState对象attached中。其中，WindowState对象A和B以及win的成员变量mSubLayer的值均小于0，但是WindowState对象win的成员变量mSubLayer的值比WindowState对象A和B的都要大，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象attached的下面，这是通过调用WindowManagerService类的成员函数placeWindowBefore来实现的。

        第四种情况是参数win所描述的WindowState对象的成员变量mSubLayer的值大于等于0，但是在附加在窗口attached的WindowState对象中，找不到一个WindowState对象，它的成员变量mSubLayer的值比WindowState对象的成员变量mSubLayer的值大，如图8和图9所示：

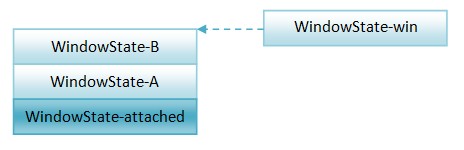


图8 窗口win插入在窗口B的上面

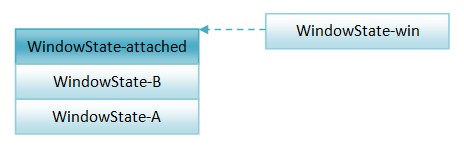


图9 窗口win插入在窗口attached的上面

        在图8和图9中，WindowState对象A和B均是附加在WindowState对象attached中。

        在图8中，WindowState对象A和B的成员变量mSubLayer的值均大于0，并且WindowState对象win的成员变量mSubLayer的值比WindowState对象A和B的都要大，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象B的上面，这是通过调用WindowManagerService类的成员函数placeWindowAfter来实现的。

        在图9中，WindowState对象A和B的成员变量mSubLayer的值均小于等于0，而WindowState对象win的成员变量mSubLayer的值大于0，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象attached的上面，这是通过调用WindowManagerService类的成员函数placeWindowAfter来实现的。

         注意，在这四种情况中，如果参数addToToken的值等于true，那么都需要将参数win所描述的WindowState对象增加到与它所对应的窗口令牌token的窗口列表windows中去。

         10. 删除WindowState

          删除WindowState是通过调用WindowManagerService类的成员函数tmpRemoveWindowLocked来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **int** **tmpRemoveWindowLocked**(**int** interestingPos, WindowState win) {

**int** wpos = mWindows.indexOf(win);

**if** (wpos >= 0) {

**if** (wpos < interestingPos) interestingPos--;

......

mWindows.remove(wpos);

mWindowsChanged = **true**;

**int** NC = win.mChildWindows.size();

**while** (NC > 0) {

NC--;

WindowState cw = win.mChildWindows.get(NC);

**int** cpos = mWindows.indexOf(cw);

**if** (cpos >= 0) {

**if** (cpos < interestingPos) interestingPos--;

......

mWindows.remove(cpos);

}

}

}

**return** interestingPos;

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        WindowManagerService类的成员函数tmpRemoveWindowLocked将参数win所描述的窗口及其子窗口从WindowManagerService服务内部的窗口堆栈中删除，即从 WindowManagerService类的成员变量mWindows所描述的一个ArrayList中删除。

        如果每一个被删除的窗口在窗口堆栈中的位置比参数interestingPos的值小，那么WindowManagerService类的成员函数tmpRemoveWindowLocked还会将参数interestingPos的值减少1，这相当于是计算当删除参数win所描述的窗口及其子窗口之后，原来位于窗口堆栈中第interestingPos个位置的窗口现在位于窗口堆栈的位置，这个位置最终会作为WindowManagerService类的成员函数tmpRemoveWindowLocked的返回值。

       11. 在指定位置增加WindowState

       在指定位置增加WindowState是通过调用WindowManagerService类的成员函数reAddWindowLocked来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **final** **int** **reAddWindowLocked**(**int** index, WindowState win) {

**final** **int** NCW = win.mChildWindows.size();

**boolean** added = **false**;

**for** (**int** j=0; j<NCW; j++) {

WindowState cwin = win.mChildWindows.get(j);

**if** (!added && cwin.mSubLayer >= 0) {

......

mWindows.add(index, win);

index++;

added = **true**;

}

......

mWindows.add(index, cwin);

index++;

}

**if** (!added) {

......

mWindows.add(index, win);

index++;

}

mWindowsChanged = **true**;

**return** index;

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        参数win描述的即为要增加的WindowState对象，而参数index描述的即为要将参数win所描述的WindowState对象及其子WindowState对象要增加到窗口堆栈中的起始位置。

       由于参数win所描述的WindowState对象的子WindowState对象的成员变量mSubLayer的值可能会小于0，也可能大于0。大于0的子WindowState对象位于参数win所描述的WindowState对象的上面，而小于0的子WindowState对象位于参数win所描述的WindowState对象的下面。因此，WindowManagerService类的成员函数reAddWindowLocked先增加那些小于0的子WindowState对象，接着再增加参数win所描述的WindowState对象，最后增加那些大于0的子WindowState对象。

        假设WindowManagerService类的成员函数reAddWindowLocked一共在窗口堆栈中增加了N个WindowState对象，那么它的返回值就等于index + N，这样调用者就可以知道参数win所描述的WindowState对象及其子WindowState对象在窗口堆栈中的最高位置是多少。

        基于第9、第10和第11这三操作，可以组合成很多其它的WindowState操作，如接下来的第12、第13、第14和第15个操作所示。

        12. 将一个WindowState对象及其所有子WindowState对象增加到窗口堆栈中

         将一个WindowState对象及其所有子WindowState对象增加到窗口堆栈中是通过调用WindowManagerService类的成员函数reAddWindowToListInOrderLocked来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **void** **reAddWindowToListInOrderLocked**(WindowState win) {

addWindowToListInOrderLocked(win, **false**);

// This is a hack to get all of the child windows added as well

// at the right position. Child windows should be rare and

// this case should be rare, so it shouldn't be that big a deal.

**int** wpos = mWindows.indexOf(win);

**if** (wpos >= 0) {

......

mWindows.remove(wpos);

mWindowsChanged = **true**;

reAddWindowLocked(wpos, win);

}

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        为了得到参数win所描述的WindowState对象的子WindowState对象在窗口堆栈中的起始位置，WindowManagerService类的成员函数reAddWindowToListInOrderLocked首先将参数win所描述的WindowState对象增加到窗口堆栈中，这是通过调用前面所分析的成员函数addWindowToListInOrderLocked来实现的，目的是为了获得它在窗口堆栈的位置。有了这个位置之后，WindowManagerService类的成员函数reAddWindowToListInOrderLocked就可以调用前面所分析的成员函数reAddWindowLocked来将WindowState对象及其所有子WindowState对象增加到窗口堆栈中去了，不过在调用之前，要先将参数win所描述的WindowState对象从窗口中堆栈删除。

        13. 将一个WindowToken对象对应的所有WindowState对象及其子WindowState对象增加到窗口堆栈的指定位置上

         将一个WindowToken对象对应的所有WindowState对象都增加到窗口堆栈中是通过调用WindowManagerService类的成员函数reAddAppWindowsLocked来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **final** **int** **reAddAppWindowsLocked**(**int** index, WindowToken token) {

**final** **int** NW = token.windows.size();

**for** (**int** i=0; i<NW; i++) {

index = reAddWindowLocked(index, token.windows.get(i));

}

**return** index;

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        与参数token所描述的WindowToken对象所对应的WindowState对象保存在它的成员变量windows所描述的一个ArrayList中。通过遍历这个ArrayList，就可以将与参数token所描述的WindowToken对象所对应的WindowState对象及其子WindowState对象都增加到窗口堆栈的指定的起始位置上去，这是通过调用前面所分析的成员函数reAddWindowLocked来实现的。

        参数index描述的便是最初指定的起始位置，每一次调用WindowManagerService类的成员函数reAddWindowLocked之后，它的值都便会被更新为下一个WindowState对象及其子WindowState对象要增加到窗口堆栈中的位置。

        最后，WindowManagerService类的成员函数reAddAppWindowsLocked将与参数token所描述的WindowToken对象所对应的WindowState对象在窗口堆栈中的最高位置加1后的得到结果返回给调用者。

       14. 将一个AppWindowToken对象所对应的WindowState对象及其子 WindowState对象移动到窗口堆栈的指定位置上

        将一个AppWindowToken对象所对应的WindowState对象及其子 WindowState对象移动到窗口堆栈的指定位置上是通过调用WindowManagerService类的成员函数moveAppWindowsLocked来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **void** **moveAppWindowsLocked**(AppWindowToken wtoken, **int** tokenPos,

**boolean** updateFocusAndLayout) {

// First remove all of the windows from the list.

tmpRemoveAppWindowsLocked(wtoken);

// Where to start adding?

**int** pos = findWindowOffsetLocked(tokenPos);

// And now add them back at the correct place.

pos = reAddAppWindowsLocked(pos, wtoken);

**if** (updateFocusAndLayout) {

**if** (!updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES)) {

assignLayersLocked();

}

mLayoutNeeded = **true**;

performLayoutAndPlaceSurfacesLocked();

}

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        参数wtoken描述的是要移动其所对应的WindowState对象的一个AppWindowToken对象，而参数tokenPos描述的是该AppWindowToken对象在WindowManagerService服务内部的AppWindowToken列表中的新位置。

        WindowManagerService类的成员函数moveAppWindowsLocked首先调用前面所分析的成员函数tmpRemoveAppWindowsLocked来移除所有与参数wtoken所描述的AppWindowToken对象所对应的WindowState对象，接着再调用也是前面所分析的成员函数findWindowOffsetLocked来获得与参数wtoken所描述的AppWindowToken对象所对应的WindowState对象在窗口堆栈中的起始位置。有了这个起始位置之后，就可以也是前面所分析的成员函数reAddAppWindowsLocked来将与参数wtoken所描述的AppWindowToken对象所对应的WindowState对象及其子WindowState对象移动到窗口堆栈上去了。

        最后，如果参数updateFocusAndLayout的值等于true，那么WindowManagerService类的成员函数moveAppWindowsLocked还会更新系统当前获得焦点的窗口，以及重新计算系统中的所有窗口的Z轴位置以及重新布局系统中的所有窗口，这三个操作分别是通过调用WindowManagerService类的成员函数updateFocusedWindowLocked、assignLayersLocked和performLayoutAndPlaceSurfacesLocked来实现的。

        15. 将一组AppWindowToken对象所对应的WindowState对象及其子 WindowState对象移动到窗口堆栈的指定位置上

         将一组AppWindowToken对象所对应的WindowState对象及其子WindowState对象移动到窗口堆栈的指定位置上是通过调用WindowManagerService类的另外一个版本的成员函数moveAppWindowsLocked来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **void** **moveAppWindowsLocked**(List<IBinder> tokens, **int** tokenPos) {

// First remove all of the windows from the list.

**final** **int** N = tokens.size();

**int** i;

**for** (i=0; i<N; i++) {

WindowToken token = mTokenMap.get(tokens.get(i));

**if** (token != **null**) {

tmpRemoveAppWindowsLocked(token);

}

}

// Where to start adding?

**int** pos = findWindowOffsetLocked(tokenPos);

// And now add them back at the correct place.

**for** (i=0; i<N; i++) {

WindowToken token = mTokenMap.get(tokens.get(i));

**if** (token != **null**) {

pos = reAddAppWindowsLocked(pos, token);

}

}

**if** (!updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES)) {

assignLayersLocked();

}

mLayoutNeeded = **true**;

performLayoutAndPlaceSurfacesLocked();

//dump();

}

......

}

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        这个操作与前面分析的第14个操作是类似，区别只在于前者是批量地移动一组AppWindowToken对象所对应的WindowState对象及其子 WindowState对象，而后者是只移动一个AppWindowToken对象所对应的WindowState对象及其子WindowState对象，此外，前者总是会调用WindowManagerService类的成员函数updateFocusedWindowLocked、assignLayersLocked和performLayoutAndPlaceSurfacesLocked来更新系统当前获得焦点的窗口、以及重新计算每一个窗口的Z轴位置，并且对这些窗口进行重新布局。

        至此，我们就分析完成WindowManagerService服务组织系统中的窗口的方式了。从分析的过程中，可以得到以下结论：

        1. WindowManagerService服务维护有一个AppWindowToken堆栈和一个WindowState堆栈，它们与ActivityManagerService服务维护的Actvity堆栈是有关相同的Z轴位置关系的。

        2. ActivityManagerService服务中的每一个ActivityRecord对象在WindowManagerService服务中都对应有一个AppWindowToken对象，而WindowManagerService服务中的每一个AppWindowToken对象都对应有一组WindowState对象。

        3. 在WindowState堆栈中，AppWindowToken堆栈中的第i+1个AppWindowToken对象所对应的WindowState对象都位于第i个AppWindowToken对象所对应的WindowState对象的上面。

        4. 一个WindowState对象可以附加在另外一个WindowState对象上面，此外，一个WindowState对象还可以有子WindowState对象，它们都是与同一个AppWindowToken对象或者WindowToken对象所对应的。

        5. WindowManagerService服务有两个特殊的WindowToken，它们分别用来描述系统中的输入法窗口令牌和壁纸窗口令牌，其中，输入法窗口位于需要输入法的窗口的上面，而壁纸窗口位于需要壁纸的窗口的下面。

        最后，我们可以将WindowManagerService服务中的AppWindowToken理解成一个Activity组件令牌，而将它所对应的WindowState对象理解成一个Activity窗口。有了这些概念之后，就为学习WindowManagerService服务的各种实现打下坚实的基础。在接下来的两篇文章中，我们就会在本文的基础上，继续分析WindowManagerService服务是如何管理系统中的输入法窗口和壁纸窗口的，敬请关注！

**老罗的新浪微博：**[**http://weibo.com/shengyangluo**](http://weibo.com/shengyangluo)**，欢迎关注！**

推荐：[Android窗口管理服务WindowManagerService对壁纸窗口（Wallpaper Window）的管理分析](http://www.itboth.com/d/EJJJvy/android-window-windowmanagerservice-wallpaper)

[    在Android系统中，壁纸窗口和输入法窗口一样，都是一种特殊类型的窗口，而且它们都是喜欢和一个普通的Activity窗口缠绵在一起。大家可以充分地想象这样的一个3W场景：

# Android窗口管理服务WindowManagerService对输入法窗口（Input Method Window）的管理分析

[审核此文](javascript:showCheckDoc('RzQ7BjNvIBjy','http://www.itboth.com/d/RzQ7BjNvIBjy/android-windowmanagerservice');)

yousheng34 分享于 2017-10-13 阅读 60 收藏 0

主题 [android](http://www.itboth.com/tag/android) [android窗口管理](http://www.itboth.com/tag/android%E7%AA%97%E5%8F%A3%E7%AE%A1%E7%90%86) [窗口管理](http://www.itboth.com/tag/%E7%AA%97%E5%8F%A3%E7%AE%A1%E7%90%86) [管理](http://www.itboth.com/tag/%E7%AE%A1%E7%90%86) [windowmanagerservice](http://www.itboth.com/tag/windowmanagerservice)

**领取地址**：

推荐：[Android窗口管理服务WindowManagerService显示窗口动画的原理分析](http://www.itboth.com/d/Q7BzM3/android)

[ 在前一文中，我们分析了Activity组件的切换过程。从这个过程可以知道，所有参与切换操作的窗口都会被设置切换动画。事实上，一个窗口在打开（关闭）的过程中，除了可

       在Android系统中，输入法窗口是一种特殊类型的窗口，它总是位于需要使用输入法的窗口的上面。也就是说，一旦WindowManagerService服务检测到焦点窗口需要使用输入法，那么它就会调整输入法窗口在窗口堆栈中的位置，使得输入法窗口位于在焦点窗口的上面，这样用户可以通过输入法窗口来录入字母或者文字。本文就将详细分析WindowManagerService服务是如何管理系统中的输入法窗口的。

       在Android系统中，除了输入法窗口之外，还有一种窗口称为输入法对话框，它们总是位于输入窗口的上面。Activity窗口、输入法窗口和输入法对话框的位置关系如图1所示：



图1 Activity窗口、输入法窗口和输入法对话框的位置关系

       在前面Android窗口管理服务WindowManagerService组织窗口的方式分析一文中提到，WindowManagerService服务是使用堆栈来组织系统中的窗口的，因此，如果我们在窗口堆栈中观察Activity窗口、输入法窗口和输入法对话框，它们的位置关系就如图2所示：

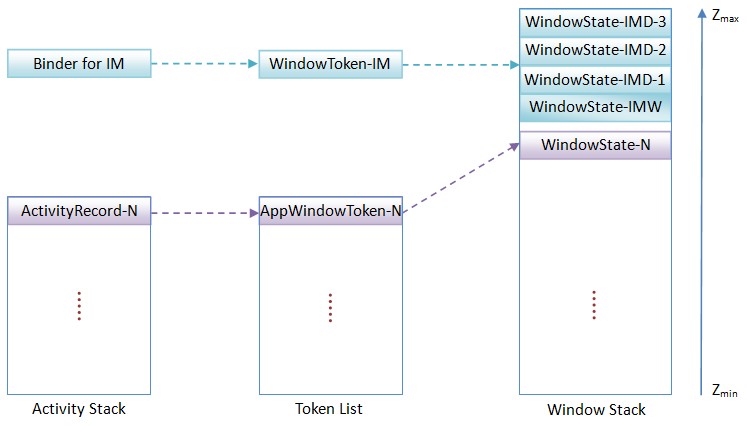


图2 Activity窗口、输入法窗口和输入法对话框在窗口堆栈中的位置关系

       图2中的对象的关系如下所示：

       1. 在ActivityManagerService服务内部的Activity组件堆栈顶端的ActivityRecord对象N描述的是系统当前激活的Activity组件。

       2. ActivityRecord对象N在WindowManagerService服务内部的窗口令牌列表顶端对应有一个AppWindowToken对象N。

       3. AppWindowToken对象N在WindowManagerService服务内部的窗口堆栈中对应有一个WindowState对象N，用来描述系统当前激活的Activity组件窗口。

       4. WindowState对象N上面有一个WindowState对象IMW，用来描述系统中的输入法窗口。

       5. WindowState对象IMW上面有三个WindowState对象IMD-1、IMD-2和IMD-3，它们用来描述系统中的输入法对话框。

       6. 系统中的输入法窗口以及输入法对话框在WindowManagerService服务内部中对应的窗口令牌是由WindowToken对象IM来描述的。

       7. WindowToken对象IM在InputMethodManagerService服务中对应有一个Binder对象。

       总的来说，就是图2描述了系统当前激活的Activity窗口上面显示输入法窗口，而输入法窗口上面又有一系列的输入法对话框的情景。WindowManagerService服务的职能之一就是要时刻关注系统中是否有窗口需要使用输入法。WindowManagerService服务一旦发现有窗口需要使用输入法，那么就会调整输入法窗口以及输入法对话框在窗口堆栈中的位置，使得它们放置在需要使用输入法的窗口的上面。

       接下来，我们就首先分析两个需要调整输入法窗口以及输入法对话框在窗口堆栈中的位置的情景，然后再分析它们是如何在窗口堆栈中进行调整的。

       第一个需要调整输入法窗口以及输入法对话框在窗口堆栈中的位置的情景是增加一个窗口到WindowManagerService服务去的时候。从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，增加一个窗口到WindowManagerService服务最终是通过调用WindowManagerService类的成员函数addWindow来实现的。接下来我们就主要分析这个函数中与输入法窗口以及输入法对话框调整相关的逻辑，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

WindowState mInputMethodWindow = **null**;

**final** ArrayList<WindowState> mInputMethodDialogs = **new** ArrayList<WindowState>();

......

**public** **int** **addWindow**(Session session, IWindow client,

WindowManager.LayoutParams attrs, **int** viewVisibility,

Rect outContentInsets, InputChannel outInputChannel) {

......

**synchronized**(mWindowMap) {

......

WindowToken token = mTokenMap.get(attrs.token);

**if** (token == **null**) {

......

**if** (attrs.type == TYPE\_INPUT\_METHOD) {

......

**return** WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;

}

......

}

......

win = **new** WindowState(session, client, token,

attachedWindow, attrs, viewVisibility);

......

**boolean** imMayMove = **true**;

**if** (attrs.type == TYPE\_INPUT\_METHOD) {

mInputMethodWindow = win;

addInputMethodWindowToListLocked(win);

imMayMove = **false**;

} **else** **if** (attrs.type == TYPE\_INPUT\_METHOD\_DIALOG) {

mInputMethodDialogs.add(win);

addWindowToListInOrderLocked(win, **true**);

adjustInputMethodDialogsLocked();

imMayMove = **false**;

}

......

**boolean** focusChanged = **false**;

**if** (win.canReceiveKeys()) {

focusChanged = updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_ASSIGN\_LAYERS);

**if** (focusChanged) {

imMayMove = **false**;

}

}

**if** (imMayMove) {

moveInputMethodWindowsIfNeededLocked(**false**);

}

......

}

......

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       如果当前增加到WindowManagerService服务来的是一个输入法窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_INPUT\_METHOD，那么就要求与该输入法窗口所对应的类型为WindowToken的窗口令牌已经存在，否则的话，WindowManagerService类的成员函数addWindow就会直接返回一个错误码WindowManagerImpl.ADD\_BAD\_APP\_TOKEN给调用者。这个类型为WindowToken的窗口令牌是InputMethodManagerService服务请求WindowManagerService服务创建的，即调用WindowManagerService类的成员函数addWindowToken来创建的，具体可以参考前面Android窗口管理服务WindowManagerService组织窗口的方式分析一文。

       如果当前增加到WindowManagerService服务来的是一个输入法窗口，那么就会将前面为它所创建的一个WindowState对象win保存在WindowManagerService类的成员变量mInputMethodWindow中，接着还会调用WindowManagerService类的成员函数addInputMethodWindowToListLocked来将该WindowState对象插入到窗口堆栈的合适位置去。

       如果当前增加到WindowManagerService服务来的是一个输入法对话框，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_INPUT\_METHOD\_DIALOG，那么就会将前面为它所创建的一个WindowState对象win添加到WindowManagerService类的成员变量mInputMethodDialogs所描述的一个ArrayList中去，并且先后调用WindowManagerService类的成员函数addWindowToListInOrderLocked和adjustInputMethodDialogsLocked来将该WindowState对象插入到窗口堆栈的合适位置去。

       在上述两种情况中，由于用来描述输入法窗口或者输入法对话框的WindowState对象已经被插入到了窗口堆栈中的合适位置，因此，接下来就不再需要考虑移动该输入法窗口或者输入法对话框了，这时候变量imMayMove的值就会被设置为false。

      另一方面，如果当前增加到WindowManagerService服务来的既不是一个输入法窗口，也不是一个输入法对话框，并且该窗口需要接收键盘事件，即前面所创建的WindowState对象win的成员函数canReceiveKeys的返回值为true，那么就可能会导致系统当前获得焦点的窗口发生变化，这时候就需要调用WindowManagerService类的成员函数updateFocusedWindowLocked来重新计算系统当前获得焦点的窗口。如果系统当前获得焦点的窗口发生了变化，那么WindowManagerService类的成员函数updateFocusedWindowLocked的返回值focusChanged就会等于true，同时系统的输入法窗口和输入法对话框在窗口堆栈中的位置也会得到调整，即位它们会位于系统当前获得焦点的窗口的上面，因此，这时候变量imMayMove的值也会被设置为false，表示接下来不再需要考虑移动系统中的输入法窗口或者输入法对话框在窗口堆栈中的位置。

      最后，如果变量imMayMove的值保持为初始值，即保持为true，那么就说明当前增加的窗口可能会引发系统的输入法窗口和输入法对话框在窗口堆栈中的位置发生变化，因此，这时候就需要调用WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked来作检测，并且在发生变化的情况下，将系统的输入法窗口和输入法对话框移动到窗口堆栈的合适位置上去。

      从上面的分析就可以知道，在增加一个窗口的过程中，可能需要调用WindowManagerService类的成员函数addInputMethodWindowToListLocked、addWindowToListInOrderLocked、adjustInputMethodDialogsLocked和moveInputMethodWindowsIfNeededLocked来移动系统的输入法窗口和输入法对话框，其中，WindowManagerService类的成员函数addWindowToListInOrderLocked在前面Android窗口管理服务WindowManagerService组织窗口的方式分析一文已经分析过了，本文只要关注其余三个成员函数的实现。

      第二个需要调整输入法窗口以及输入法对话框在窗口堆栈中的位置的情景是一个应用程序进程请求WindowManagerService服务重新布局一个窗口的时候。从前面Android窗口管理服务WindowManagerService计算Activity窗口大小的过程分析一文可以知道，应用程序进程请求WindowManagerService服务重新布局一个窗口最终是通过调用WindowManagerService类的成员函数relayoutWindow来实现的。接下来我们就主要分析这个函数中与输入法窗口以及输入法对话框调整相关的逻辑，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**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) {

......

**synchronized**(mWindowMap) {

WindowState win = windowForClientLocked(session, client, **false**);

......

**int** attrChanges = 0;

**int** flagChanges = 0;

**if** (attrs != **null**) {

flagChanges = win.mAttrs.flags ^= attrs.flags;

attrChanges = win.mAttrs.copyFrom(attrs);

}

......

**boolean** imMayMove = (flagChanges&(

WindowManager.LayoutParams.FLAG\_ALT\_FOCUSABLE\_IM |

WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE)) != 0;

**boolean** focusMayChange = win.mViewVisibility != viewVisibility

|| ((flagChanges&WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE) != 0)

|| (!win.mRelayoutCalled);

......

**if** (viewVisibility == View.VISIBLE &&

(win.mAppToken == **null** || !win.mAppToken.clientHidden)) {

displayed = !win.isVisibleLw();

......

**if** ((attrChanges&WindowManager.LayoutParams.FORMAT\_CHANGED) != 0) {

// To change the format, we need to re-build the surface.

win.destroySurfaceLocked();

displayed = **true**;

}

......

**if** (win.mAttrs.type == TYPE\_INPUT\_METHOD

&& mInputMethodWindow == **null**) {

mInputMethodWindow = win;

imMayMove = **true**;

}

**if** (displayed) {

focusMayChange = **true**;

}

......

} **else** {

......

**if** (win.mSurface != **null**) {

......

// If we are not currently running the exit animation, we

// need to see about starting one.

**if** (!win.mExiting || win.mSurfacePendingDestroy) {

......

**if** (!win.mSurfacePendingDestroy && win.isWinVisibleLw() &&

applyAnimationLocked(win, transit, **false**)) {

focusMayChange = **true**;

win.mExiting = **true**;

} **else** **if** (win.isAnimating()) {

// Currently in a hide animation... turn this into

// an exit.

win.mExiting = **true**;

} **else** **if** (win == mWallpaperTarget) {

// If the wallpaper is currently behind this

// window, we need to change both of them inside

// of a transaction to avoid artifacts.

win.mExiting = **true**;

win.mAnimating = **true**;

} **else** {

**if** (mInputMethodWindow == win) {

mInputMethodWindow = **null**;

}

win.destroySurfaceLocked();

}

}

}

......

}

**if** (focusMayChange) {

......

**if** (updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES)) {

imMayMove = **false**;

}

......

}

// updateFocusedWindowLocked() already assigned layers so we only need to

// reassign them at this point if the IM window state gets shuffled

**boolean** assignLayers = **false**;

**if** (imMayMove) {

**if** (moveInputMethodWindowsIfNeededLocked(**false**) || displayed) {

// Little hack here -- we -should- be able to rely on the

// function to return true if the IME has moved and needs

// its layer recomputed. However, if the IME was hidden

// and isn't actually moved in the list, its layer may be

// out of data so we make sure to recompute it.

assignLayers = **true**;

}

}

......

**if** (assignLayers) {

assignLayersLocked();

}

......

}

......

**return** (inTouchMode ? WindowManagerImpl.RELAYOUT\_IN\_TOUCH\_MODE : 0)

| (displayed ? WindowManagerImpl.RELAYOUT\_FIRST\_TIME : 0);

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       应用程序进程在请求WindowManagerService服务重新布局一个窗口的时候，这个窗口的一些布局参数可能会发生变化，而这些变化可能会同时引发系统的输入法窗口以及输入法对话框在窗口堆栈中的位置发生变化。如果系统的输入法窗口以及输入法对话框在窗口堆栈中的位置发生了变化，那么就需要调整它们在窗口堆栈中的位置。

       WindowManagerService类的成员函数relayoutWindow首先调用根据参数session和client来调用另外一个成员函数windowForClientLocked，以便可以获得用来描述要重新布局的窗口的一个WindowState对象win。

       WindowState对象win的成员变量mAttrs指向的是一个WindowManager.LayoutParams对象，该WindowManager.LayoutParams对象的成员变量flags描述的是窗口上一次所设置的布局属性标志位，而参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量flags描述的是窗口当前被设置的布局属性标志位。WindowManagerService类的成员函数relayoutWindow通过对这两个标志位执行一个异或操作，就可以知道窗口的哪些布局属性标志位发生了变化，这些变化就记录在变量flagChanges中。

       WindowManagerService类的成员函数relayoutWindow在对WindowState对象win所描述的窗口进行布局之前，还要将参数attrs指的是一个WindowManager.LayoutParams对象的内容拷贝到 WindowState对象win的成员变量mAttrs指向的是一个WindowManager.LayoutParams对象中去。在拷贝的过程中，如果发现这两个WindowManager.LayoutParams对象所描述的窗口布局属性有发生变化，那么这些变化就会记录在变量attrChanges中。

       在窗口的布局属性标志中，位WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE表示窗口是否可以获得焦点，另外一个位WindowManager.LayoutParams.FLAG\_ALT\_FOCUSABLE\_IM是用来反转WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE位的作用的。一个窗口是否可以获得焦点意味着它是否需要与输入法窗口交互，即如果一个窗口是可以获得焦点的，那么就意味着它需要与输入法窗口交互，否则就不需要。当一个窗口的WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE位等于1，那么就表示窗口不可以获得焦点，即不需要与输入法窗口交互，但是如果该窗口的WindowManager.LayoutParams.FLAG\_ALT\_FOCUSABLE\_IM位也等于1，那么就表示窗口仍然是需要与输入法窗口交互的。另一方面，如果一个窗口的WindowManager.LayoutParams.FLAG\_ALT\_FOCUSABLE\_IM位等于1，但是该窗口的WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE位等于0，那么就表示窗口仍然是不可以与输入法窗口交互的。因此，当前面得到的变量flagChanges的WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE位或者WindowManager.LayoutParams.FLAG\_ALT\_FOCUSABLE\_IM位发生了变化时，都意味着对WindowState对象win所描述的窗口进行重新布局会影响系统中的输入法窗口以及输入法对话框，即该窗口可能会由需要显示输入法窗口以及输入法对话框，到不需要显示输入法窗口以及输入法对话框，反之亦然。最后得到的变量imMayMove的值等于true就表示要移动系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置。

       一个窗口由不可获得焦点到可以获得焦点，或者由可获得焦点到不可以获得焦点，即窗口布局属性标志中的WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE位发生了变化，那么就意味着要重新计算系统当前获得焦点的窗口。从前面分析增加窗口到WindowManagerService服务的情景可以知道，当系统当前获得焦点的窗口发生变化时，也意味着需要系统中的移动输入法窗口以及输入法对话框在窗口堆栈中的位置。除了窗口布局属性标志中的WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE位变化会引发系统当前获得焦点的窗口发生变化之外，还有另外两个因素会引发系统当前获得焦点的窗口发生变化。第一个因素是窗口的可见性发生变化。WindowState对象win的成员变量mViewVisibility描述的是窗口上一次布局时的可见性，而参数viewVisibility描述的是窗口当前的可见性，当它们的值不相等时，就意味着窗口的可见性发生了变化。第二个因素是窗口是第一次被应用程序进程请求WindowManagerService服务布局，这时候WindowState对象win的成员变量mRelayoutCalled的值就会等于false。最后得到的变量focusMayChange等于true，就表示需要重新计算系统当前获得焦点的窗口。

       WindowState对象win所描述的窗口在此次重新布局中是否会引起移动系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置，还取决于它在的可见性以及它的绘图表面属性等信息，接下来我们就按照 WindowState对象win所描述的窗口当前是可见还是不可见来分别分析。

       我们首先分析WindowState对象win所描述的窗口在此次重新布局中是可见的情景，即参数viewVisibility的值等于View.VISIBLE。注意，如果WindowState对象win所描述的是一个Activity窗口，而该Activity组件是不可见的，那么即使参数viewVisibility的值等于View.VISIBLE，那么WindowState对象win所描述的窗口在此次重新布局中也是认为不可见的。从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，当WindowState对象win的成员变量mAppToken的值不等于null时，那么该WindowState对象win描述的是一个Activity窗口，而当该成员变量所指向的一个AppWindowToken对象的成员变量clientHidden的值等于false时，就表示对应的Activity组件是可见的。

       WindowState对象win所描述的窗口在上一次布局时的可见性可以调用它的成员函数isVisibleLw来获得。如果WindowState对象win所描述的窗口在上一次布局时是不可见的，那么现在就需要将它设置为可见的，即要将它显示出来，这时候变量displayed的值就会等于true。另一方面，如果WindowState对象win所描述的窗口的绘图表面的像素格式发生了变化，即变量attrChanges的WindowManager.LayoutParams.FORMAT\_CHANGED位等于1，那么这时候就需要调用WindowState对象win的成员函数destroySurfaceLocked来销毁它所描述的窗口的绘图表面，以便接下来可以为它重新创建一个新的绘图表面，这时候也会将变量displayed的值设置为true，表示接下来是要显示WindowState对象win所描述的窗口的。如果最终得到的变量displayed的值设置为true，那么就相当于说明WindowState对象win所描述的窗口经历一个由不可见到可见的状态变化，因此就可能会导致系统当前获得焦点的窗口发生变化，这时候就会将变量focusMayChange的值设置为true。

       如果WindowState对象win描述的是一个输入法窗口，即它的成员变量mAttrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_INPUT\_METHOD，并且系统中的输入法窗口尚未设置，即WindowManagerService类的成员变量mInputMethodWindow的值等于null，那么就说明接下来要显示的其实是输入法窗口，这情况会导致需要移动系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置，因此，这时候除了需要将WindowState对象win保存在WindowManagerService类的成员变量mInputMethodWindow之外，还需要将变量imMayMove的值设置为true。

       我们接下来再分析WindowState对象win所描述的窗口在此次重新布局中是不可见的情景。一个窗口变得不可见了，就意味着可能要销毁它的绘图表面，取决于它的绘图表面是否存在，以及它的退出动画是否已经显示结束。WindowState对象win所描述的窗口的绘图表面保存在它的成员变量mSurface中，因此，当WindowState对象win的成员变量mSurface不等于null的时候，就意味着可能会销毁它所描述的绘图表面。

       如果WindowState对象win的成员变量mExiting等于false时，那么就说明该WindowState对象win所描述的窗口的退出动画可能尚未开始，也可能已经结束。另一方面，如果WindowState对象win的成员变量mSurfacePendingDestroy的值等于true，那么就说明该WindowState对象win所描述的窗口的绘图表面正在等待销毁。这两种情况都需要进一步确定接下来是要开始WindowState对象win所描述的窗口的退出动画，还是要销毁WindowState对象win所描述的窗口的绘图表面。

       如果WindowState对象win的成员变量mSurfacePendingDestroy的值等于false，那么同时也意味着它所描述的窗口还未开始显示退出动画，因而它的绘图表面就没有进入正在等待销毁的状态。在这种情况下，如果WindowState对象win所描述的窗口是可见的，即它的成员函数isWinVisibleLw的返回值等于true，那么就意味要开始该窗口的退出动画了，这是通过调用WindowManagerService类的成员函数applyAnimationLocked来实现的。WindowState对象win描述的窗口开始退出动画之后，就意味要重新计算系统当前获得焦点的窗口，因此，这时候就会将变量focusMayChange的值设置为true，同时还会将WindowState对象win的成员变量mExiting的值设置为true，表示它描述的窗口正在退出的过程中。

       如果WindowState对象win所描述的窗口正在处于退出动画的过程中，即它的成员函数isAnimating的返回值等于true，那么这时候需要确保WindowState对象win的成员变量mExiting的值为true。

       如果WindowState对象win所描述的窗口已经结束退出动画，但是它仍然是壁纸窗口的目标，即WindowManagerService类的成员变量mWallpaperTarget的值不等于null，并且它的值就等于WindowState对象win，那么这时候就需要等待壁纸窗口也退出之后，才销毁WindowState对象win所描述的窗口，因此，这时候就需要将WindowState对象win的成员变量mExiting和mAnimating的值设置为true，即假装它所描述的窗口还处于正在退出的过程，这样做是为了等待壁纸窗口退出完成。

       如果WindowState对象win所描述的窗口已经结束退出动画，并且它不是壁纸窗口的目标，那么这时候就需要调用它的成员函数destroySurfaceLocked来销毁它的绘图表面了。在销毁WindowState对象win所描述的窗口之前，还会判断它是否就是系统当前的输入法窗口，即WindowManagerService类的成员变量mInputMethodWindow的值是否等于win。如果等于的话，那么就说明系统当前的输入法窗口被销毁了，因此，就需要将WindowManagerService类的成员变量mInputMethodWindow的值设置为null。

       经过上面的一系列操作之后，如果最终得到的变量focusMayChange的值等于true，那么就说明需要重新计算系统当前获得焦点的窗口了，这是通过调用WindowManagerService类的成员函数updateFocusedWindowLocked来实现的。一旦WindowManagerService类的成员函数updateFocusedWindowLocked的返回值为true，那么就说明统当前获得焦点的窗口发生了变化，并且系统中的输入法窗口以及输入法对话框也移动到窗口堆栈中的正确位置了，因此，这时候就会将变量imMayMove的值设置为false。

       经过上面的一系列操作之后，如果最终得到的变量imMayMove的值等于true，那么就说明有可能需要移动系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置，这是通过调用WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked来实现的。一旦系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置发生了移动，那么WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked的返回值就等于true，这时候就需要将变量assignLayers的值设置为true，表示要重新计算系统中的窗口的Z轴位置，以便可以同步到SurfaceFlinger服务中去。注意，如果系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置没有发生变化，但是前面得到的变量displayed的值等于true，那么也是需要将变量assignLayers的值设置为true的，因为这个变量displayed的值等于true意味着WindowState对象win所描述的窗口经历了从不可见到可见的状态变化，因此也需要重新计算系统中的窗口的Z轴位置。

       经过上面的一系列操作之后，如果最终得到的变量assignLayers的值等于true，那么就需要调用WindowManagerService类的成员函数assignLayersLocked来执行重新计算统中的窗口的Z轴位置的操作了。在后面的文章中，我们再详细分析WindowManagerService服务是如何计算系统中的窗口的Z轴位置的。

       从上面的分析就可以知道，在布局一个窗口的过程中，可能需要调用WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked来移动系统的输入法窗口和输入法对话框。再结合前面增加窗口的情景，我们就可以知道，在WindowManagerService类中，与输入法窗口以及输入法对话框相关的成员函数有addInputMethodWindowToListLocked、adjustInputMethodDialogsLocked和moveInputMethodWindowsIfNeededLocked，它们的作用如下所示：

       A. 成员函数addInputMethodWindowToListLocked用来将输入法窗口插入到窗口堆栈的合适位置，即插入到需要显示输入法窗口的窗口的上面。

       B. 成员函数adjustInputMethodDialogsLocked用来移动输入法对话框到窗口堆栈的合适位置，即移动到输入法窗口的上面。

       C. 成员函数moveInputMethodWindowsIfNeededLocked用来检查是否需要移动输入法窗口以及输入法对话框。如果需要的话，那么就将它们移动到窗口堆栈的合适位置去，即将输入法窗口移动到需要显示输入法窗口的窗口的上面，而将输入法对话框移动到输入法窗口的上面。

       在分析这三个成员函数的实现之前，我们首先分析WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked和moveInputMethodDialogsLocked，它们是两个基本的操作，其中：

       D.  成员函数findDesiredInputMethodWindowIndexLocked用来查找输入法窗口在窗口堆栈的正确位置，这个位置刚好就是在需要显示输入法窗口的窗口在窗口堆栈中的上一个位置。

       E. 成员函数moveInputMethodDialogsLocked用来将移动输入法对话框移动到输入法窗口的上面去。

       接下来我们开始分析上述五个函数的实现。

       1. 计算输入法窗口在窗口堆栈中的位置

       输入法窗口在窗口堆栈中的位置是通过调用WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked来获得的，它首先找到需要显示输入法的窗口在窗口堆栈中的位置，然后再将这个位置加1，就可以得到输入法窗口在窗口堆栈中的位置。

       WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中，它的实现比较长，我们分段来阅读：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**int** **findDesiredInputMethodWindowIndexLocked**(**boolean** willMove) {

**final** ArrayList<WindowState> localmWindows = mWindows;

**final** **int** N = localmWindows.size();

WindowState w = **null**;

**int** i = N;

**while** (i > 0) {

i--;

w = localmWindows.get(i);

......

**if** (canBeImeTarget(w)) {

......

// Yet more tricksyness! If this window is a "starting"

// window, we do actually want to be on top of it, but

// it is not -really- where input will go. So if the caller

// is not actually looking to move the IME, look down below

// for a real window to target...

**if** (!willMove

&& w.mAttrs.type == WindowManager.LayoutParams.TYPE\_APPLICATION\_STARTING

&& i > 0) {

WindowState wb = localmWindows.get(i-1);

**while** (i > 1 && wb.mAppToken == w.mAppToken && !canBeImeTarget(wb)) {

i--;

wb = localmWindows.get(i-1);

}

**if** (wb.mAppToken == w.mAppToken && canBeImeTarget(wb)) {

i--;

w = wb;

}

}

**break**;

}

}

mUpcomingInputMethodTarget = w;

       这段代码从上到下遍历WindowManagerService服务内部的窗口堆栈，即WindowManagerService类的成员变量mWindows所描述的一个ArrayList。如果发现有一个窗口是可见的，并且需要显示输入法窗口，那么整个查找过程就会结束。检查一个窗口是否可见以及需要显示输入法窗口是通过调用WindowManagerService类的成员函数canBeImeTarget来实现的。最后得到的需要显示输入法的窗口就使用WindowState对象w中，这个WindowState对象w接下来还会保存在WindowManagerService类的成员变量mUpcomingInputMethodTarget中，表示它即将要成为输入法窗口的目标窗口。

       参数willMove表示调用者计算输入法窗口在窗口堆栈中的位置的目的。如果它的值等于true，那么就说明调用者获得了输入法窗口在窗口堆栈中的位置之后，接下来就会将输入法窗口移动到需要显示输入法窗口的窗口的上面去，否则的话，就说明调用者只是为了知道输入法窗口在窗口堆栈中的位置，而不打算移动输入法窗口。

       在从上到下查找需要显示输入法的窗口的过程中，如果找到一个WindowState对象w，它所描述的窗口需要显示输入法窗口，但是这个窗口其实是一个Activity窗口的启动窗口，即该WindowState对象w的成员变量mAttrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于WindowManager.LayoutParams.TYPE\_APPLICATION\_STARTING，那么由于调用WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked的目的不是用来移动输入法窗口，而是用来查找输入法窗口在窗口堆栈中的确切位置，因此就不能前面所找到的启动窗口看作是一个需要输入法的窗口，因为这个启动窗口只是Activity窗口在显示过程中出现的一个临时窗口。在这种情况下，这段代码就会继续沿着窗口堆栈往下查找另外一个窗口，该窗口一方面是需要显示输入法窗口的，另一方面要与前面所找到的启动窗口对应的是同一个窗口令牌的。如果能找到这样的一个窗口，那么就会将用来描述它的一个WindowState对象wb保存在变量w中。如果找不到这样的一个窗口，那么这段代码就会继续沿着窗口堆栈往下查找另外一个需要显示输入法的窗口。

       我们继续往下阅读代码：

**if** (willMove && w != **null**) {

final WindowState curTarget = mInputMethodTarget;

**if** (curTarget != **null** && curTarget.mAppToken != **null**) {

// Now some fun for dealing with window animations that

// modify the Z order. We need to look at all windows below

// the current target that are in this app, finding the highest

// visible one in layering.

AppWindowToken token = curTarget.mAppToken;

WindowState highestTarget = **null**;

**int** highestPos = 0;

**if** (token.animating || token.animation != **null**) {

**int** pos = 0;

pos = localmWindows.indexOf(curTarget);

**while** (pos >= 0) {

WindowState win = localmWindows.**get**(pos);

**if** (win.mAppToken != token) {

**break**;

}

**if** (!win.mRemoved) {

**if** (highestTarget == **null** || win.mAnimLayer >

highestTarget.mAnimLayer) {

highestTarget = win;

highestPos = pos;

}

}

pos--;

}

}

**if** (highestTarget != **null**) {

......

**if** (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {

// If we are currently setting up for an animation,

// hold everything until we can find out what will happen.

mInputMethodTargetWaitingAnim = **true**;

mInputMethodTarget = highestTarget;

**return** highestPos + 1;

} **else** **if** (highestTarget.isAnimating() &&

highestTarget.mAnimLayer > w.mAnimLayer) {

// If the window we are currently targeting is involved

// with an animation, and it is on top of the next target

// we will be over, then hold off on moving until

// that is done.

mInputMethodTarget = highestTarget;

**return** highestPos + 1;

}

}

}

}

       这段代码用来处理一种特殊情况，即参数willMove的值等于true，并且前面找到了一个需要显示输入法的窗口w，但是当前输入法窗口已经存在一个目标窗口，并且该目标窗口正在切换的过程中。在这种情况下，调用WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked的函数就需要等到当前输入法窗口的目标窗口的切换过程结束之后，再将输入法窗口移动到窗口w的上面去，换句话说，就是要保持输入法窗口在它当前的目标窗口的上面，直到它当前的目标窗口的切换过程结束为止。这样WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked就需要找到当前输入法窗口的目标窗口在窗口堆栈中的位置，然后再将该位置加1后返回给调用者。

       当WindowManagerService类的成员变量mInputMethodTarget的值不等于null，并且它描述的是一个Activity窗口时，即它的成员变量mAppToken的值不等于null时，那么就说明当前输入法窗口已经存在一个目标窗口，而这个目标窗口就是使用WindowManagerService类的成员变量mInputMethodTarget所指向的一个WindowState对象来描述的。接下来这段代码就检查该目标窗口是否正在切换的过程中，即是否正在显示切换动画。如果是的话，那么WindowState对象curTarget的成员变量animating的值就会等于true，或者另外一个成员变量animation的值不等于null，这时候就需要在与该目标窗口所对应的窗口令牌token所描述的一组窗口中，找到一个Z轴位置最大的并且不是已经被移除的窗口。WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked的调用者最后就是需要将输入法窗口移动到这个Z轴位置最大的并且不是已经被移除的窗口的上面的。

       一个窗口的Z轴位置是记录在用描述它的一个WindowState对象的成员变量mAnimLayer中的，而它是否是已经被移除是记录在这个WindowState对象的成员变量mRemoved中的，因此，如果在窗口令牌token所描述的一组WindowSate对象中，能找到一个WindowSate对象，它的成员变量mAnimLayer的值最大，并且它的成员变量mRemoved不等于true，那么这段代码就会将它保存在变量highestTarget中，并且将它描述的窗口在窗口堆栈中的位置保存在变量highestPos中。

       经过前面的一系列计算之后，如果变量highestTarget的值不等于null，那么就说明我们碰到前面所说的特殊的情况，这时候又要分为两种情况来讨论。

       第一种情况是当前输入法窗口的目标窗口即将要进入到切换过程，但是这个切换过程尚开始，即WindowManagerService类的成员变量mNextAppTransition的值不等于WindowManagerPolicy.TRANSIT\_UNSET。这时候就需要将WindowManagerService类的成员变量mInputMethodTargetWaitingAnim的值设置为true，表示当前输入法窗口的目标窗口正在等待进入切换动画中，并且需要将WindowManagerService类的成员变量mInputMethodTarget修正为变量highestTarget所描述的一个WindowState对象，因为这个WindowState对象才是真正用来描述当前输入法窗口的目标窗口的。

       第二种情况是当前输入法窗口的目标窗口已经处于切换的过程了，即变量highestTarget所描述的一个WindowState对象的成员函数isAnimating的返回值为true，并且该目标窗口的Z轴位置大于前面所找到的需要显示输入法窗口的窗口的Z轴，即变量highestTarget所描述的一个WindowState对象的成员变量mAnimLayer的值大于变量w所描述的一个WindowState对象的成员变量mAnimLayer的值。这时候就需要将WindowState对象highestTarget所描述的窗口维持为当前输入法窗口的目标窗口，即将WindowManagerService类的成员变量mInputMethodTarget设置为变量highestTarget，直到WindowState对象highestTarget所描述的窗口的切换过程结束为止。

      上述两种情况最后都需要将WindowState对象highestTarget所描述的窗口在窗口堆栈中的位置highestPos加1，然后再返回给调用者，以便调用者接下来可以输入法窗口移动在窗口堆栈的第（highestPos+1）个位置上。

推荐：[Android对输入法窗口的管理分析,Android窗口管理服务WindowManagerService对输入法窗口（Input Method Window）的管理分析](http://www.itboth.com/d/UFnyiu/android)

[原文：Android窗口管理服务WindowManagerService对输入法窗口（Input Method Window）的管理分析 链接：http://blog.csdn.net/luoshengyang/article/details/8526644在And

       如果我们没有碰到前面所说的特殊的情况，那么WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked就会继续往下执行：

**if** (w != null) {

**if** (willMove) {

......

mInputMethodTarget = w;

**if** (w.mAppToken != null) {

setInputMethodAnimLayerAdjustment(w.mAppToken.animLayerAdjustment);

} **else** {

setInputMethodAnimLayerAdjustment(0);

}

}

return i+1;

}

       如果变量w的值不等于null，那么就说明WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked在前面找到了一个需要显示输入法窗口的窗口。这个窗口是使用WindowState对象w来描述的，并且它在窗品堆栈中的位置记录在变量i中。这时候WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked就会执行以下三个操作：

       A. 将WindowState对象w保存在WindowManagerService类的成员变量mInputMethodTarget中，以便WindowManagerService服务可以知道当前输入法窗口的目标窗口是什么。

       B. 检查WindowState对象w描述的窗口是否是Activity窗口，即检查WindowState对象w的成员变量mAppToken的值是否不等于null。如果WindowState对象w描述的窗口是Activity窗口的话，那么就需要根据WindowState对象w的成员变量mAppToken所描述的一个AppWindowToken对象的成员变量animLayerAdjustment来调整系统中的输入法窗口以及输入法对话框的Z轴位置，即在系统中的输入法窗口以及输入法对话框的现有Z轴位置的基础上再增加一个调整量，这个调整量就保存在WindowState对象w的成员变量mAppToken所描述的一个AppWindowToken对象的成员变量animLayerAdjustment中。这个调整的过程是通过调用WindowManagerService类的成员函数setInputMethodAnimLayerAdjustment来实现的。如果WindowState对象w描述的窗口不是Activity窗口，那么就不需要调整系统中的输入法窗口以及输入法对话框的Z轴位置，但是仍然需要调用WindowManagerService类的成员函数setInputMethodAnimLayerAdjustment来将系统中的输入法窗口以及输入法对话框的Z轴位置调整量设置为0，即将WindowManagerService类的成员变量mInputMethodAnimLayerAdjustment的值设置为0。

      C. 将变量i的值加1之后返回给调用者，以便调用者可以将系统中的输入法窗口移动到窗口堆栈中的第（i+1）个位置上。

       如果变量w的值等于null，那么就说明WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked在前面没有找到一个需要显示输入法窗口的窗口，我们继续往下阅读它的代码，以便可以了解它是如何处理这种情况的：

**if** (willMove) {

......

mInputMethodTarget = null;

setInputMethodAnimLayerAdjustment(0);

}

return -1;

}

......

}

      WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked对在前面没有找到一个需要显示输入法窗口的窗口的情况的处理很简单。它判断参数willMove的值是否等于true。如果等于true的话，那么就会将WindowManagerService类的成员变量mInputMethodTarget的值设置为null，并且调用WindowManagerService类的成员函数setInputMethodAnimLayerAdjustment来将系统中的输入法窗口以及输入法对话框的Z轴位置调整量设置为0。这实际上是用来通知WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked的调用者，系统当前没有需要显示输入法窗口的窗口。

       最后，WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked返回一个-1值给调用者，也是表明系统当前没有需要显示输入法窗口的窗口。

       2. 移动输入法对话框移动到输入法窗口的上面

       系统中的输入法对话框是需要位于输入法窗口的上面的，因此，我们就需要有一个函数来将输入法对话框移动到输入法窗口的上面去。这个函数就是WindowManagerService类的成员函数moveInputMethodDialogsLocked，它的实现如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**void** **moveInputMethodDialogsLocked**(**int** pos) {

ArrayList<WindowState> dialogs = mInputMethodDialogs;

**final** **int** N = dialogs.size();

......

**for** (**int** i=0; i<N; i++) {

pos = tmpRemoveWindowLocked(pos, dialogs.get(i));

}

......

**if** (pos >= 0) {

**final** AppWindowToken targetAppToken = mInputMethodTarget.mAppToken;

**if** (pos < mWindows.size()) {

WindowState wp = mWindows.get(pos);

**if** (wp == mInputMethodWindow) {

pos++;

}

}

......

**for** (**int** i=0; i<N; i++) {

WindowState win = dialogs.get(i);

win.mTargetAppToken = targetAppToken;

pos = reAddWindowLocked(pos, win);

}

......

**return**;

}

**for** (**int** i=0; i<N; i++) {

WindowState win = dialogs.get(i);

win.mTargetAppToken = **null**;

reAddWindowToListInOrderLocked(win);

......

}

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       在调用WindowManagerService类的成员函数moveInputMethodDialogsLocked之前，必须要保证系统中的输入法窗口已经被移动到窗口堆栈的正确位置，即已经被移动到需要显示输入法窗口的窗口的上面。这时候参数pos描述的或者是输入法窗口在窗口堆栈中的位置，或者是输入法窗口在窗口堆栈的位置的上一个位置，即输入法对话框在窗口堆栈中的起始位置。参数pos的值还可以小于0，这时候就表示系统当前没有需要显示输入法窗口的窗口。

       在移动输入法对话框到输入法窗口的上面之前，首先要将输入法对话框从窗口堆栈中移除，以便接下来可以重新将它们插入到窗口堆栈中。系统中的输入法对话框都保存在WindowManagerService类的成员变量mInputMethodDialogs所描述的一个ArrayList中，通过调用WindowManagerService类的成员函数来tmpRemoveWindowLocked来移除保存在这个ArrayList中的每一个WindowState对象，就可以将系统中的输入法对话框从窗口堆栈中移除中。注意，将一个WindowState对象从窗口堆栈中移除之后，可能会影响参数pos的值。例如，如果参数pos的值大于被移除的WindowState对象原来在窗口堆栈中的位置值，那么在该WindowState对象被移除之后，参数pos的值就要相应地减少1，这样它才能正确地反映输入法窗口在窗口堆栈中的位置，或者输入法对话框在窗口堆栈中的起始位置。WindowManagerService类的成员函数来tmpRemoveWindowLocked在将一个WindowState对象从窗口堆栈中移除的过程中，会正确处理好参数pos的值，这一点可以参考前面Android窗口管理服务WindowManagerService组织窗口的方式分析一文。

       接下来，我们就分为两种情况来分析输入法对话框在窗口是如何移动到输入法窗口的上面去的。

       第一种情况是参数pos的值大于等于0，这表明系统当前存在一个需要显示输入法窗口的窗口，这个窗口是通过WindowManagerService类的成员变量mInputMethodTarget所指向的一个WindowState对象来描述的。

       前面提到，参数pos描述的或者是输入法窗口在窗口堆栈中的位置，或者是输入法对话框在窗口堆栈中的起始位置，我们首先要将它统一描述为输入法对话框在窗口堆栈中的起始位置。这时候就需要检查保存在窗口堆栈的第pos个位置的WindowState对象wp，是否就是WindowManagerService类的成员变量mInputMethodWindow所指向的那个WindowState对象。如果是的话，那么就说明参数pos描述的或者是输入法窗口在窗口堆栈中的位置，这时候将它的值增加1，就可以让它表示为输入法对话框在窗口堆栈中的起始位置。

       得到了输入法对话框在窗口堆栈中的起始位置pos之后，接下来只需要调用WindowManagerService类的成员函数reAddWindowLocked来依次地将保存在WindowManagerService类的成员变量mInputMethodDialogs所描述的一个ArrayList中的第i个WindowState对象保存在窗口堆栈中的第（pos+i）个以位置上即可，这样就可以将输入法对话框都移动到输入法窗口的上面去了。

      注意，在移动的过程中，用来描述每一个输入法对话框的每一个WindowState对象的成员变量mTargetAppToken的值设置为WindowManagerService类的成员变量mInputMethodTarget所描述的一个WindowState对象的成员变量mAppToken的值，以便可以将输入法对话框和输入法窗口的目标窗口设置为同一个窗口。

       第二种情况是参数pos的值小于0，这表明系统当前不存在一个需要显示输入法窗口的窗口。这时候就需要根据输入法窗口自身的属性来将它们移动到窗口堆栈的合适的位置上去，这是通过调用WindowManagerService类的成员函数reAddWindowToListInOrderLocked来实现的。WindowManagerService类的成员函数reAddWindowToListInOrderLocked的实现可以参考前面Android窗口管理服务WindowManagerService组织窗口的方式分析一文，这里不再详细。

       注意，在移动的过程中，用来描述每一个输入法对话框的每一个WindowState对象的成员变量mTargetAppToken的值会被设置为null，这是因为系统当前不存在一个需要显示输入法窗口的窗口，即这时候每一个输入法对话框都没有目标窗口。

       理解了WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked和moveInputMethodDialogsLocked的实现之后，对WindowManagerService类的另外三个成员函数addInputMethodWindowToListLocked、adjustInputMethodDialogsLocked和moveInputMethodWindowsIfNeededLocked的实现就很有帮助，接下来我们就继续分析这三个成员函数的实现。

       3. 插入输入法窗口到需要显示输入法窗口的窗口上面

       插入输入法窗口到窗口堆栈的合适位置，使得它位于需要显示输入法窗口的窗口上面，这是通过调用WindowManagerService类的成员函数addInputMethodWindowToListLocked来实现的，它的实现如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**void** **addInputMethodWindowToListLocked**(WindowState win) {

**int** pos = findDesiredInputMethodWindowIndexLocked(**true**);

**if** (pos >= 0) {

win.mTargetAppToken = mInputMethodTarget.mAppToken;

......

mWindows.add(pos, win);

mWindowsChanged = **true**;

moveInputMethodDialogsLocked(pos+1);

**return**;

}

win.mTargetAppToken = **null**;

addWindowToListInOrderLocked(win, **true**);

moveInputMethodDialogsLocked(pos);

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       参数win描述的是要添加到窗口堆栈中去的输入法窗口。

       WindowManagerService类的成员函数addInputMethodWindowToListLocked首先调用另外一个成员函数findDesiredInputMethodWindowIndexLocked来计算输入法窗口在窗口堆栈中的位置，并且保存在变量pos。

       如果变量pos的值大于等于0，那么就说明WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked在窗口堆栈中找到了一个合适的位置来放置输入法窗口，于是接下来就会参数win所描述的输入法窗口插入在WindowManagerService类的成员变量mWIndows所描述的窗口堆栈的第pos个位置上。由于系统中的输入法对话框要保持在输入法窗口的上面，因此，WindowManagerService类的成员函数addInputMethodWindowToListLocked还需要继续调用另外一个成员函数moveInputMethodDialogsLocked来将系统中的输入法对话框在窗口堆栈中的起始位置设置为（pos+1）。

       还有一个地方需要注意的是，前面在调用WindowManagerService类的成员函数addInputMethodWindowToListLocked来计算输入法窗口在窗口堆栈中的位置的时候，已经将用来描述需要显示输入法窗口的Activity窗口的一个WindowState对象保存了WindowManagerService类的成员变量mInputMethodTarget中，因此，这里就需要这个WindowState对象的成员变量mAppToken所指向的一个AppWindowToken对象保存在用来描述输入法窗口的WindowState对象的win的成员变量mTargetAppToken中，以便WindowManagerService服务可以知道当前输入法窗口的目标窗口是什么。

       如果变量pos的值小于0，那么就说明WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked没有找一个需要输入法窗口的窗口，因此，这时候就需要调用另外一个成员函数addWindowToListInOrderLocked来将参数win所描述的输入法窗口插入到窗口堆栈中去。WindowManagerService类的成员函数addWindowToListInOrderLocked会根据要目标窗口所对应的窗口令牌在窗口令牌列表中的位置以及是否在窗口堆栈中存在其它窗口等信息来在窗口堆栈中找到一个合适的前位置来放置目标窗口，它的具体实现可以参考前面[Android窗口管理服务WindowManagerService组织窗口的方式分析](http://blog.csdn.net/luoshengyang/article/details/8498908)一文。将参数win所描述的输入法窗口插入到窗口堆栈中去之后，WindowManagerService类的成员函数addInputMethodWindowToListLocked还需要继续调用另外一个成员函数moveInputMethodDialogsLocked来调整系统中的输入法对话框。

       注意，在调用WindowManagerService类的成员函数moveInputMethodDialogsLocked的时候，传递进去的参数pos的值等于-1，这时候WindowManagerService类的成员函数moveInputMethodDialogsLocked就不是直接调整输入法对话框在窗口堆栈中的位置的，而是调用另外一个成员函数reAddWindowToListInOrderLocked来调整的。

       还有另外一个地方需要注意的是，由于前面在调用WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked的时候，没有找到一个需要输入法窗口的窗口，因此，这里就需要将参数win所描述的一个WindowState对象的成员变量mTargetAppToken的值设置为null，以便WindowManagerService服务可以知道当前输入法窗口的没有目标窗口。

       4. 调整输入法对话框在窗口堆栈的位置

       一旦系统中存在需要显示输入法窗口的窗口，那么就需要系统中的输入法对话框在窗口堆栈中的位置，使得它们放置在输入法窗品的上面，这是通过调用WindowManagerService类的成员函数adjustInputMethodDialogsLocked来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**void** **adjustInputMethodDialogsLocked**() {

moveInputMethodDialogsLocked(findDesiredInputMethodWindowIndexLocked(**true**));

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       WindowManagerService类的成员函数adjustInputMethodDialogsLocked的实现很简单，它首先调用成员函数findDesiredInputMethodWindowIndexLocked来找到输入法窗口在窗口堆栈中的位置，然后再调用成员函数moveInputMethodDialogsLocked来将输入法对话框保存在这个位置之上。

       5. 调整输入法窗口在窗口堆栈的位置

       当系统中的窗口布局发生了变化之后，例如，当前获得焦点的窗口发生了变化，或者新增了一个窗口，那么都可能需要调整输入法窗口在窗口堆栈中的位置，以便它可以痊于需要显示输入法窗口的窗口的上面，这是通过调用WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**boolean** **moveInputMethodWindowsIfNeededLocked**(**boolean** needAssignLayers) {

**final** WindowState imWin = mInputMethodWindow;

**final** **int** DN = mInputMethodDialogs.size();

**if** (imWin == **null** && DN == 0) {

**return** **false**;

}

**int** imPos = findDesiredInputMethodWindowIndexLocked(**true**);

**if** (imPos >= 0) {

// In this case, the input method windows are to be placed

// immediately above the window they are targeting.

// First check to see if the input method windows are already

// located here, and contiguous.

**final** **int** N = mWindows.size();

WindowState firstImWin = imPos < N

? mWindows.get(imPos) : **null**;

// Figure out the actual input method window that should be

// at the bottom of their stack.

WindowState baseImWin = imWin != **null**

? imWin : mInputMethodDialogs.get(0);

**if** (baseImWin.mChildWindows.size() > 0) {

WindowState cw = baseImWin.mChildWindows.get(0);

**if** (cw.mSubLayer < 0) baseImWin = cw;

}

**if** (firstImWin == baseImWin) {

// The windows haven't moved... but are they still contiguous?

// First find the top IM window.

**int** pos = imPos+1;

**while** (pos < N) {

**if** (!(mWindows.get(pos)).mIsImWindow) {

**break**;

}

pos++;

}

pos++;

// Now there should be no more input method windows above.

**while** (pos < N) {

**if** ((mWindows.get(pos)).mIsImWindow) {

**break**;

}

pos++;

}

**if** (pos >= N) {

// All is good!

**return** **false**;

}

}

**if** (imWin != **null**) {

......

imPos = tmpRemoveWindowLocked(imPos, imWin);

......

imWin.mTargetAppToken = mInputMethodTarget.mAppToken;

reAddWindowLocked(imPos, imWin);

......

**if** (DN > 0) moveInputMethodDialogsLocked(imPos+1);

} **else** {

moveInputMethodDialogsLocked(imPos);

}

} **else** {

// In this case, the input method windows go in a fixed layer,

// because they aren't currently associated with a focus window.

**if** (imWin != **null**) {

......

tmpRemoveWindowLocked(0, imWin);

imWin.mTargetAppToken = **null**;

reAddWindowToListInOrderLocked(imWin);

......

**if** (DN > 0) moveInputMethodDialogsLocked(-1);;

} **else** {

moveInputMethodDialogsLocked(-1);;

}

}

**if** (needAssignLayers) {

assignLayersLocked();

}

**return** **true**;

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked首先检查系统中是否存在输入法窗口和输入法对话框，即检查WindowManagerService类的成员变量mInputMethodWindow的值是否等于null，并且WindowManagerService类的成员变量mInputMethodDialogs所描述的一个ArrayList的大小是否等于0。如果输入法窗口和输入法对话框都不存在的话，那么就不用调整它们在窗口堆栈中的位置了，否则的话，WindowManagerService类的成员变量mInputMethodWindow所指向的一个WindowState对象就会保存在变量imWin中，以便接下来可以通过它来描述系统中的输入法窗口。

      在输入法窗口或者输入法对话框存在的情况下，WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked接下来就会继续调用另外一个成员函数findDesiredInputMethodWindowIndexLocked来找到输入法窗口在窗口堆栈中的位置，并且保存在变量imPos中。注意，变量imPos的值可能大于等于0，也可能等于-1。当变量imPos的值大于等于0的时候，就说明系统当前存在一个窗口需要显示输入法窗口，而当变量imPos的值等于-1的时候，就说明系统当前不存在一个窗口需要显示输入法窗口，或者系统中不存在输入法窗口。接下来我们分两种情况来分析WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked的实现。

       第一种情况是变量imPos的值可能大于等于0。这时候可能需要调整输入法窗口在窗口堆栈中的位置，也可能不需要调整输入法窗口在窗口堆栈中的位置，取决于输入法窗口的位置是否已经在窗口堆栈的第imPos个位置上，以及是否所有与输入法相关的窗口都连续在放置在窗口堆栈中。

       变量firstImWin描述的是当前位于窗口堆栈中Z轴位置最小的与输入法相关的窗口，它是通过变量imPos来获得的。另外一个变量baseImWin描述的是Z轴位置最小的与输入法相关的窗口。如果这两个变量描述的是同一个窗口，那么就说明输入法窗口的位置已经在窗口堆栈的第imPos个位置上，因此，就有可能不需要调整输入法窗品在窗口堆栈中的位置了。接下来我们就描述如何找到这个Z轴位置最小的与输入法相关的窗口。

       如果变量imWin的值不等于null，即WindowManagerService类的成员变量mInputMethodWindow的值不等于null，那么它所描述的窗口就是Z轴位置最小的与输入法相关的窗口，否则的话，Z轴位置最小的与输入法相关的窗口就是位于WindowManagerService类的成员变量mInputMethodDialogs所描述的一个ArrayList的第0个位置上的输入法对话框。这一步得到的Z轴位置最小的与输入法相关的窗口就保存在变量baseImWin中。

       如果变量baseImWin所描述的窗口有子窗口，即它所指向的一个WindowState对象的成员变量mChildWindows所描述的一个ArrayList的大小大于0。这时候如果用来描述第一个子窗口的WindowState对象的成员变量mSubLayer的值小于0，那么就说明变量baseImWin所描述的窗口在所有与输入法相关的窗口中的Z轴位置还不是最小的，因为在它的下面还存在着Z轴位置更小的子窗口。在这种情况下，变量baseImWin就会指向这个Z轴位置最小的子窗口。

       经过上面的一系列计算之后，如果变量firstImWin和变量baseImWin描述的是同一个窗口，那么还需要继续判断所有与输入法相关的窗口都连续在放置在窗口堆栈中。判断的方法如下所示：

      (1). 从窗口堆栈的第（imPos + 1）个位置开始往上查找一个非输入法相关的窗口。

      (2). 如果第(1)步能在窗口堆栈中大于等于（imPos+1）的位置pos上找到一个非输入法窗口，那么再继续从第pos个位置开始往上查找一个与输入法相关的窗口。

      (3). 如果第(2)步能在窗口堆栈中找到一个与输入法相关的窗口，那么就说明所有与输入法相关的窗口不是连续在放置在窗口堆栈中的，因为在它们中间有一个非输入法相关的窗口，否则的话，就说明所有与输入法相关的窗口都是连续在放置在窗口堆栈中的。

       在所有与输入法相关的窗口都是连续在放置在窗口堆栈中的情况下，WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked就会直接返回一个false值给调用者，表明不需要调整系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置。

       在所有与输入法相关的窗口不是连续在放置在窗口堆栈中的情况下，就需要重新调整系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置。这里又需要分两个情景来讨论。

       第一个情景是变量imWin的值不等于null，这时候说明系统中存在一个输入法窗口，因此，就需要调整这个输入法窗口在窗口堆栈中的位置。调整的方法很简单：

       (1). 调用WindowManagerService类的成员函数tmpRemoveWindowLocked来从窗口堆栈中移除变量imWin所描述的输入法窗口。在移除的过程中，会同时计算输入法窗口在窗口堆栈中的新位置，这个位置还是保存在变量imPos中。

       (2). 调用WindowManagerService类的成员函数reAddWindowLocked重新将变量imWin所描述的输入法窗口插入到窗口堆栈的第imPos个位置中。在插入之前，还会将变量imWin所描述的一个WindowState对象的成员变量mTargetAppToken与WindowManagerService类的成员变量mInputMethodTarget所描述的一个WindowState对象的成员变量mAppToken指向同一个AppWindowToken对象，这样WindowManagerService服务就可以知道imWin所描述的输入法窗口的目标窗口是什么。

       (3). 如果系统中还存在输入法对话框，那么就调用WindowManagerService类的成员函数moveInputMethodDialogsLocked来将它们放置在第（imPos+1）个位置上，目的是将它们放置在输入法窗口的上面。

       第二个情景是变量imWin的值等于null，这时候说明系统中不存在输入法窗口。在这个情景下，系统中肯定会存在输入法对话框，否则的话，WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked在前面就会返回了。因此，WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked接下来就会直接调用成员函数moveInputMethodDialogsLocked来将系统中的输入法对话框放置在在第imPos个位置上。

       第二种情况是变量imPos的值等于-1。这时候说明系统中不存在需要显示输入法窗口的窗口。这里同样也需要分两个情景来分析。

       第一个情景是变量imWin的值不等于null，这时候说明系统中存在一个输入法窗口，因此，就需要调整这个输入法窗口在窗口堆栈中的位置。调整的方法与前面第一种情况的第一个情景是类似的。不过由于事先不知道输入法窗口在窗口堆栈中的位置，因此，这里就会调用WindowManagerService类的成员函数reAddWindowToListInOrderLocked和moveInputMethodDialogsLocked来间接地调整输入法窗口和输入法对话框在窗口堆栈中的位置。注意，在调用WindowManagerService类的成员函数moveInputMethodDialogsLocked的时候，传进去的参数为-1。另外一个地方需要注意的是，在WindowManagerService类的成员函数reAddWindowToListInOrderLocked来间接地调整输入法窗口在窗口堆栈中的位置之前，会将量imWin所描述的一个WindowState对象的成员变量mTargetAppToken的值设置为null，这样WindowManagerService服务就可以知道imWin所描述的输入法窗口没有目标窗口。

       第二情景是变量imWin的值等于null，这时候系统中不存在输入法窗口。这个情景与前面第一种情况的第二个情景也是类似的。由于系统中不存在输入法窗口，因此只需要调用WindowManagerService类的成员函数moveInputMethodDialogsLocked来间接地输入法对话框在窗口堆栈中的位置即可，即以参数-1来调用WindowManagerService类的成员函数moveInputMethodDialogsLocked。

       至此，我们就分析完成WindowManagerService服务对输入法窗口的基本操作了。从分析的过程中，我们可以得到以下两个结论：

       A. 系统中与输入法相关的窗口有两种，一种是输入法窗口，另一种是输入法对话框。

       B. 当Z轴位置最大的窗口需要使用输入法时，输入法窗口就会位于它的上面，而输入法对话框又会位于输入法窗口的上面。

       在WindowManagerService服务中，还有一种类型的窗口与输入法窗口类似，它总是与Activity窗口粘在一起。不过，这种类型的窗口是位于Activity窗口的下面，刚好与输入法窗口相反，它就是壁纸窗口（Wallpaper）。在接下来的一篇文章中，我们就将继续分析WindowManagerService服务是如何管理系统中的壁纸窗口的。敬请关注！

**老罗的新浪微博：**[**http://weibo.com/shengyangluo**](http://weibo.com/shengyangluo)**，欢迎关注！**

本文出自 “[老罗的Android之旅](http://shyluo.blog.51cto.com/)” 博客，请务必保留此出处<http://shyluo.blog.51cto.com/5725845/1229284>

推荐：[Android窗口管理服务WindowManagerService对窗口的组织方式分析](http://www.itboth.com/d/ZFJFzu/android-windowmanagerservice)

[我们知道，在Android系统中，Activity是以堆栈的形式组织在ActivityManagerService服务中的。与Activity类似，Android系统中的窗口也是以堆栈的形式组织在WindowManagerSe

# Android窗口管理服务WindowManagerService计算Activity窗口大小的过程分析

[审核此文](javascript:showCheckDoc('IRfQfaBNNFBz','http://www.itboth.com/d/IRfQfaBNNFBz/android-windowmanagerservice-activity');)

jianghujiujiyihao 分享于 2017-10-16 阅读 70 收藏 0

主题 [android](http://www.itboth.com/tag/android) [窗口管理](http://www.itboth.com/tag/%E7%AA%97%E5%8F%A3%E7%AE%A1%E7%90%86) [管理](http://www.itboth.com/tag/%E7%AE%A1%E7%90%86) [activity](http://www.itboth.com/tag/activity) [windowmanagerservice](http://www.itboth.com/tag/windowmanagerservice)

**领取地址**：

推荐：[Android窗口管理服务WindowManagerService显示Activity组件的启动窗口（Starting Window）的过程分析](http://www.itboth.com/d/JBBvqaiMRNVr/window-android-starting-windowmanagerservice-activity)

[在Android系统中，Activity组件在启动之后，并且在它的窗口显示出来之前，可以显示一个启动窗口。这个启动窗口可以看作是Activity组件的预览窗口，是由WindowManagerServi

       在Android系统中，Activity窗口的大小是由WindowManagerService服务来计算的。WindowManagerService服务会根据屏幕及其装饰区的大小来决定Activity窗口的大小。一个Activity窗口只有知道自己的大小之后，才能对它里面的UI元素进行测量、布局以及绘制。本文将详细分析WindowManagerService服务计算Activity窗口大小的过程。

       一般来说，Activity窗口的大小等于整个屏幕的大小，但是它并不占据着整块屏幕。为了理解这一点，我们首先分析一下Activity窗口的区域是如何划分的。

       我们知道，Activity窗口的上方一般会有一个状态栏，用来显示3G信号、电量使用等图标，如图1所示。

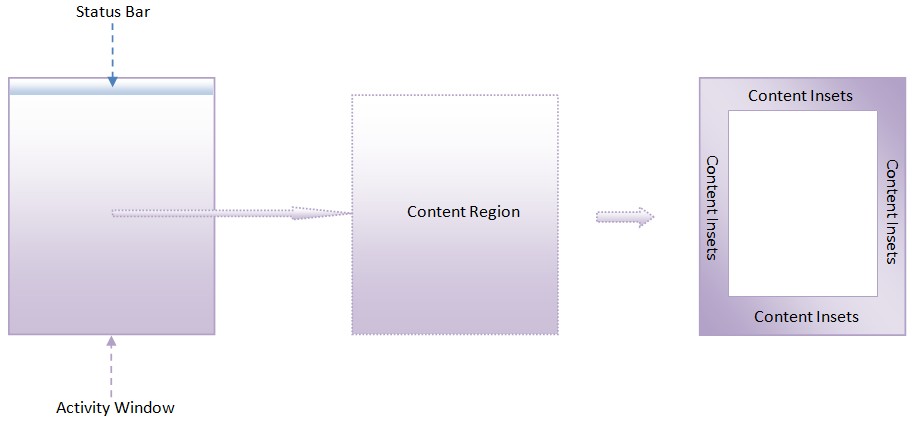


图1 Activity窗口的Content区域示意图

      从Activity窗口剔除掉状态栏所占用的区域之后，所得到的区域就称为内容区域（Content Region）。顾名思义，内容区域就是用来显示Activity窗口的内容的。我们再抽象一下，假设Activity窗口的四周都有一块类似状态栏的区域，那么将这些区域剔除之后，得到中间的那一块区域就称为内容区域，而被剔除出来的区域所组成的区域就称为内容边衬区域（Content Insets）。Activity窗口的内容边衬区域可以用一个四元组（content-left, content-top, content-right, content-bottom）来描述，其中，content-left、content-right、content-top、content-bottom分别用来描述内容区域与窗口区域的左右上下边界距离。

      我们还知道，Activity窗口有时候需要显示输入法窗口，如图2所示。

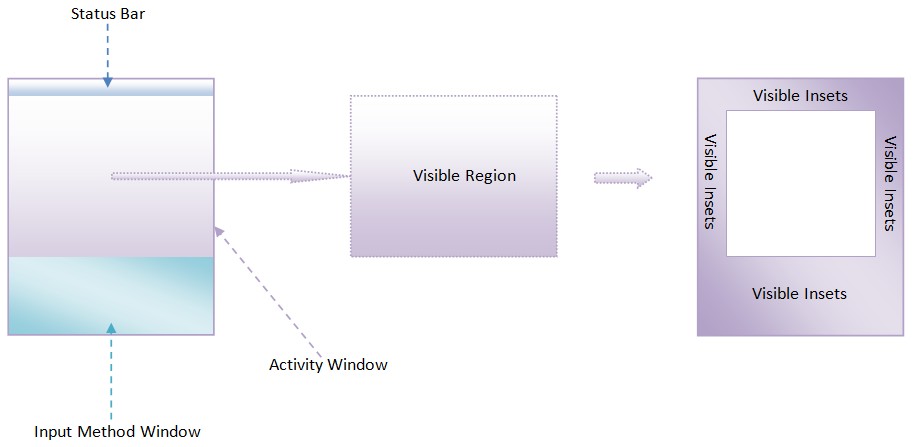


图2 Activity窗口的Visible区域示意图

       这时候Activity窗口的内容区域的大小有可能没有发生变化，这取决于它的Soft Input Mode。我们假设Activity窗口的内容区域没有发生变化，但是它在底部的一些区域被输入法窗口遮挡了，即它在底部的一些内容是不可见的。从Activity窗口剔除掉状态栏和输入法窗口所占用的区域之后，所得到的区域就称为可见区域（Visible Region）。同样，我们再抽象一下，假设Activity窗口的四周都有一块类似状态栏和输入法窗口的区域，那么将这些区域剔除之后，得到中间的那一块区域就称为可见区域，而被剔除出来的区域所组成的区域就称为可见边衬区域（Visible Insets）。Activity窗口的可见边衬区域可以用一个四元组（visible-left, visible-top, visible-right, visible-bottom）来描述，其中，visible-left、visible-right、visible-top、visible-bottom分别用来描述可见区域与窗口区域的左右上下边界距离。

       在大多数情况下，Activity窗口的内容区域和可见区域的大小是一致的，而状态栏和输入法窗口所占用的区域又称为屏幕装饰区。理解了这些概念之后，我们就可以推断，WindowManagerService服务实际上就是需要根据屏幕以及可能出现的状态栏和输入法窗口的大小来计算出Activity窗口的整体大小及其内容区域边衬和可见区域边衬的大小。有了这三个数据之后，Activity窗口就可以对它里面的UI元素进行测量、布局以及绘制等操作了。

       从前面Android应用程序窗口（Activity）的绘图表面（Surface）的创建过程分析一文可以知道，应用程序进程是从ViewRoot类的成员函数performTraversals开始，向WindowManagerService服务请求计算一个Activity窗口的大小的，因此，接下来我们就从ViewRoot类的成员函数performTraversals开始分析一个Activity窗口大小的计算过程，如图3所示。

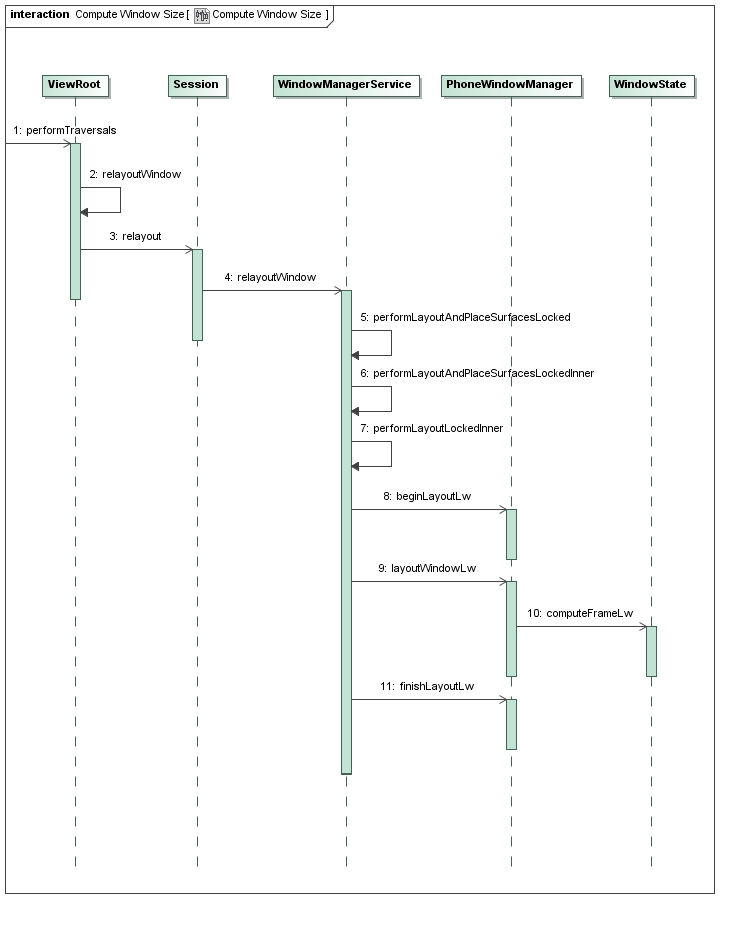


图3 Activity窗口大小的计算过程

        这个过程可以分为11个步骤，接下来我们就详细分析每一个步骤。

        Step 1. ViewRoot.performTraversals

        这个函数定义在文件frameworks/base/core/java/android/view/ViewRoot.java中，它的实现很复杂，一共有600-行，不过大部分代码都是用来计算Activity窗口的大小的，我们分段来阅读：

**public** **final** **class** **ViewRoot** **extends** **Handler** **implements** **ViewParent**,

**View**.**AttachInfo**.**Callbacks** {

......

**private** **void** **performTraversals**() {

......

**final** View host = mView;

......

**int** desiredWindowWidth;

**int** desiredWindowHeight;

**int** childWidthMeasureSpec;

**int** childHeightMeasureSpec;

......

Rect frame = mWinFrame;

**if** (mFirst) {

......

DisplayMetrics packageMetrics =

mView.getContext().getResources().getDisplayMetrics();

desiredWindowWidth = packageMetrics.widthPixels;

desiredWindowHeight = packageMetrics.heightPixels;

} **else** {

desiredWindowWidth = frame.width();

desiredWindowHeight = frame.height();

**if** (desiredWindowWidth != mWidth || desiredWindowHeight != mHeight) {

......

windowResizesToFitContent = **true**;

}

}

       这段代码用来获得Activity窗口的当前宽度desiredWindowWidth和当前高度desiredWindowHeight。

       注意，Activity窗口当前的宽度和高度是保存ViewRoot类的成员变量mWinFrame中的。ViewRoot类的另外两个成员变量mWidth和mHeight也是用来描述Activity窗口当前的宽度和高度的，但是它们的值是由应用程序进程上一次主动请求WindowManagerService服务计算得到的，并且会一直保持不变到应用程序进程下一次再请求WindowManagerService服务来重新计算为止。Activity窗口的当前宽度和高度有时候是被WindowManagerService服务主动请求应用程序进程修改的，修改后的值就会保存在ViewRoot类的成员变量mWinFrame中，它们可能会与ViewRoot类的成员变量mWidth和mHeight的值不同。

       如果Activity窗口是第一次被请求执行测量、布局和绘制操作，即ViewRoot类的成员变量mFirst的值等于true，那么它的当前宽度desiredWindowWidth和当前高度desiredWindowHeight就等于屏幕的宽度和高度，否则的话，它的当前宽度desiredWindowWidth和当前高度desiredWindowHeight就等于保存在ViewRoot类的成员变量mWinFrame中的宽度和高度值。

       如果Activity窗口不是第一次被请求执行测量、布局和绘制操作，并且Activity窗口主动上一次请求WindowManagerService服务计算得到的宽度mWidth和高度mHeight不等于Activity窗口的当前宽度desiredWindowWidth和当前高度desiredWindowHeight，那么就说明Activity窗口的大小发生了变化，这时候变量windowResizesToFitContent的值就会被标记为true，以便接下来可以对Activity窗口的大小变化进行处理。

       我们继续往下阅读代码：

boolean insetsChanged = false;

**if** (mLayoutRequested) {

......

**if** (mFirst) {

host.fitSystemWindows(mAttachInfo.mContentInsets);

......

} **else** {

**if** (!mAttachInfo.mContentInsets.equals(mPendingContentInsets)) {

mAttachInfo.mContentInsets.set(mPendingContentInsets);

host.fitSystemWindows(mAttachInfo.mContentInsets);

insetsChanged = true;

......

}

**if** (!mAttachInfo.mVisibleInsets.equals(mPendingVisibleInsets)) {

mAttachInfo.mVisibleInsets.set(mPendingVisibleInsets);

......

}

**if** (lp.width == ViewGroup.LayoutParams.WRAP\_CONTENT

|| lp.height == ViewGroup.LayoutParams.WRAP\_CONTENT) {

windowResizesToFitContent = true;

DisplayMetrics packageMetrics =

mView.getContext().getResources().getDisplayMetrics();

desiredWindowWidth = packageMetrics.widthPixels;

desiredWindowHeight = packageMetrics.heightPixels;

}

}

childWidthMeasureSpec = getRootMeasureSpec(desiredWindowWidth, lp.width);

childHeightMeasureSpec = getRootMeasureSpec(desiredWindowHeight, lp.height);

......

host.measure(childWidthMeasureSpec, childHeightMeasureSpec);

......

}

       这段代码用来在Activity窗口主动请求WindowManagerService服务计算大小之前，对它的顶层视图进行一次测量操作。

       在分析这段代码之前，我们首先解释一下ViewRoot类的成员变量mAttachInfo和mPendingContentInsets、mPendingVisibleInsets。ViewRoot类的成员变量mAttachInfo指向的一个AttachInfo对象，这个AttachInfo对象用来描述Activity窗口的属性，例如，这个AttachInfo对象的成员变量mContentInsets和mVisibleInsets分别用来描述Activity窗口上一次主动请求WindowManagerService服务计算得到的内容边衬大小和可见边衬大小，即Activity窗口的当前内容边衬大小和可见边衬大小。ViewRoot类的成员变量mPendingContentInsets和mPendingVisibleInsets也是用来描述Activity窗口的内容边衬大小和可见边衬大小的，不过它们是由WindowManagerService服务主动请求Activity窗口设置的，但是尚未生效。

       我们分两种情况来分析这段代码。

       第一种情况是Activity窗口是第一次被请求执行测量、布局和绘制操作，即ViewRoot类的成员变量mFirst的值等于true，那么这段代码在测量Activity窗口的顶层视图host的大小之前，首先会调用这个顶层视图host的成员函数fitSystemWindows来设置它的四个内边距（mPaddingLeft，mPaddingTop，mPaddingRight，mPaddingBottom）的大小设置为Activity窗口的初始化内容边衬大小。这样做的目的是可以在Activity窗口的四周留下足够的区域来放置可能会出现的系统窗口，也就是状态栏和输入法窗口。

       第二种情况是Activity窗口不是第一次被请求执行测量、布局和绘制操作，即ViewRoot类的成员变量mFirst的值等于false，那么这段代码就会检查Activity窗口是否被WindowManagerService服务主动请求设置了一个新的内容边衬大小mPendingContentInsets和一个新的可见边衬大小mPendingVisibleInsets。如果是的话，那么就会分别将它们保存在ViewRoot类的成员变量mAttachInfo所指向的一个AttachInfo对象的成员变量mContentInsets和成员变量mVisibleInsets中。注意，如果Activity窗口被WindowManagerService服务主动请求设置了一个新的内容边衬大小mPendingContentInsets，那么这段代码同时还需要同步调用Activity窗口的顶层视图host的成员函数fitSystemWindows来将它的四个内边距（mPaddingLeft，mPaddingTop，mPaddingRight，mPaddingBottom）的大小设置为新的内容边衬大小，并且将变量insetsChanged的值设置为true，表明Activity窗口的内容边衬大小发生了变化。

       在第二种情况下，如果Activity窗口的宽度被设置为ViewGroup.LayoutParams.WRAP\_CONTENT或者高度被设置为ViewGroup.LayoutParams.WRAP\_CONTENT，那么就意味着Activity窗口的大小要等于内容区域的大小。但是由于Activity窗口的大小是需要覆盖整个屏幕的，因此，这时候就会Activity窗口的当前宽度desiredWindowWidth和当前高度desiredWindowHeight设置为屏幕的宽度和高度。也就是说，如果我们将Activity窗口的宽度和高度设置为ViewGroup.LayoutParams.WRAP\_CONTENT，实际上就意味着它的宽度和高度等于屏幕的宽度和高度。这种情况也意味着Acitivity窗口的大小发生了变化，因此，就将变量windowResizesToFitContent的值设置为true。

       经过上面的一系列处理之后，这段代码就会调用ViewRoot类的成员函数getRootMeasureSpec来根据Activity窗口的当前宽度和宽度测量规范以及高度和高度测量规范来计算得到它的顶层视图host的宽度测量规范childWidthMeasureSpec和高度测量规范childHeightMeasureSpec。有了这两个规范之后，就可以调用Activity窗口的顶层视图host的成员函数measure来执行大小测量的工作了。这个大小测量的过程可以参考前面Android应用程序窗口（Activity）的测量（Measure）、布局（Layout）和绘制（Draw）过程分析一文。

      我们继续往下阅读代码：

**boolean** windowShouldResize = mLayoutRequested && windowResizesToFitContent

&& ((mWidth != host.mMeasuredWidth || mHeight != host.mMeasuredHeight)

|| (lp.width == ViewGroup.LayoutParams.WRAP\_CONTENT &&

frame.width() < desiredWindowWidth && frame.width() != mWidth)

|| (lp.height == ViewGroup.LayoutParams.WRAP\_CONTENT &&

frame.height() < desiredWindowHeight && frame.height() != mHeight));

**final** **boolean** computesInternalInsets =

attachInfo.mTreeObserver.hasComputeInternalInsetsListeners();

      这段代码主要是做两件事情。

      第一件事情是检查是否需要处理Activity窗口的大小变化事件。如果满足以下条件，那么就需要处理，即将变量windowShouldResize的值设置为true：

      1. ViewRoot类的成员变量mLayoutRequest的值等于true，这说明应用程序进程正在请求对Activity窗口执行一次测量、布局和绘制操作；

      2. 变量windowResizesToFitContent的值等于true，这说明前面检测到了Activity窗口的大小发生了变化；

      3. 前面我们已经Activity窗口的顶层视图host的大小重新进行了测量。如果测量出来的宽度host.mMeasuredWidth和高度host.mMeasuredHeight和Activity窗口的当前宽度mWidth和高度mHeight一样，那么即使条件1和条件2能满足，那么也是可以认为是Activity窗口的大小是没有发生变化的。换句话说，只有当测量出来的大小和当前大小不一致时，才认为Activity窗口大小发生了变化。另一方面，如果测量出来的大小和当前大小一致，但是Activity窗口的大小被要求设置成WRAP\_CONTENT，即设置成和屏幕的宽度desiredWindowWidth和高度desiredWindowHeight一致，但是WindowManagerService服务请求Activity窗口设置的宽度frame.width()和高度frame.height()与它们不一致，而且与Activity窗口上一次请求WindowManagerService服务计算的宽度mWidth和高度mHeight也不一致，那么也是认为Activity窗口大小发生了变化的。

       第二件事情是检查Activity窗口是否需要指定有额外的内容边衬区域和可见边衬区域。如果有的话，那么变量attachInfo所指向的一个AttachInfo对象的成员变量mTreeObserver所描述的一个TreeObserver对象的成员函数hasComputeInternalInsetsListerner的返回值ComputeInternalInsets就会等于true。Activity窗口指定额外的内容边衬区域和可见边衬区域是为了放置一些额外的东西。

       我们继续往下阅读代码：

**if** (mFirst || windowShouldResize || insetsChanged

|| viewVisibilityChanged || params != null) {

**if** (viewVisibility == View.VISIBLE) {

// If **this** window **is** giving internal insets to the window

// manager, **and** it **is** being added **or** changing its visibility,

// **then** we want to first give the window manager "fake"

// insets to cause it to effectively ignore the content **of**

// the window during layout. This avoids it briefly causing

// other windows to resize/move based on the raw frame **of** the

// window, waiting **until** we can finish laying out **this** window

// **and** get back to the window manager with the ultimately

// computed insets.

insetsPending = computesInternalInsets

&& (mFirst || viewVisibilityChanged);

......

}

       这段代码以及接下来的两段代码都是在满足下面的条件之一的情况下执行的：

       1. Activity窗口是第一次执行测量、布局和绘制操作，即ViewRoot类的成员变量mFirst的值等于true。

       2. 前面得到的变量windowShouldResize的值等于true，即Activity窗口的大小的确是发生了变化。

       3. 前面得到的变量insetsChanged的值等于true，即Activity窗口的内容区域边衬发生了变化。

       4. Activity窗口的可见性发生了变化，即变量viewVisibilityChanged的值等于true。

       5. Activity窗口的属性发生了变化，即变量params指向了一个WindowManager.LayoutParams对象。

       在满足上述条件之一，并且Activity窗口处于可见状态，即变量viewVisibility的值等于View.VISIBLE，那么就需要检查接下来请求WindowManagerService服务计算大小时，是否要告诉WindowManagerService服务它指定了额外的内容区域边衬和可见区域边衬，但是这些额外的内容区域边衬和可见区域边衬又还有确定。这种情况发生在Activity窗口第一次执行测量、布局和绘制操作或者由不可见变化可见时。因此，当前面得到的变量computesInternalInsets等于true时，即Activity窗口指定了额外的内容区域边衬和可见区域边衬，那么就需要检查ViewRoot类的成员变量mFirst或者变量viewVisibilityChanged的值是否等于true。如果这些条件能满足，那么变量insetsPending的值就会等于true，表示Activity窗口有额外的内容区域边衬和可见区域边衬等待指定。

       我们继续往下阅读代码：

boolean contentInsetsChanged = **false**;

boolean visibleInsetsChanged;

......

**try** {

......

relayoutResult = relayoutWindow(**params**, viewVisibility, insetsPending);

contentInsetsChanged = !mPendingContentInsets.equals(

mAttachInfo.mContentInsets);

visibleInsetsChanged = !mPendingVisibleInsets.equals(

mAttachInfo.mVisibleInsets);

**if** (contentInsetsChanged) {

mAttachInfo.mContentInsets.**set**(mPendingContentInsets);

host.fitSystemWindows(mAttachInfo.mContentInsets);

......

}

**if** (visibleInsetsChanged) {

mAttachInfo.mVisibleInsets.**set**(mPendingVisibleInsets);

......

}

......

} **catch** (RemoteException e) {

}

......

attachInfo.mWindowLeft = frame.left;

attachInfo.mWindowTop = frame.top;

// !!FIXME!! This next section handles the case where we did not get the

// window size we asked for. We should avoid this by getting a maximum size from

// the window session beforehand.

mWidth = frame.width();

mHeight = frame.height();

       这段代码主要就是调用ViewRoot类的另外一个成员函数relayoutWindow来请求WindowManagerService服务计算Activity窗口的大小以及内容区域边衬大小和可见区域边衬大小。计算完毕之后，Activity窗口的大小就会保存在ViewRoot类的成员变量mWinFrame中，而Activity窗口的内容区域边衬大小和可见区域边衬大小分别保存在ViewRoot类的成员变量mPendingContentInsets和mPendingVisibleInsets中。

       如果这次计算得到的Activity窗口的内容区域边衬大小mPendingContentInsets和可见区域边衬大小mPendingVisibleInsets与上一次计算得到的不一致，即与ViewRoot类的成员变量mAttachInfo所指向的一个AttachInfo对象的成员变量mContentInsets和mVisibleInsets所描述的大小不一致，那么变量contentInsetsChanged和visibleInsetsChanged的值就会等于true，表示Activity窗口的内容区域边衬大小和可见区域边衬大小发生了变化。

       由于变量frame和ViewRoot类的成员变量mWinFrame引用的是同一个Rect对象，因此，这时候变量frame描述的也是Activity窗口请求WindowManagerService服务计算之后得到的大小。这段代码分别将计算得到的Activity窗口的左上角坐标保存在变量attachInfo所指向的一个AttachInfo对象的成员变量mWindowLeft和mWindowTop中，并且将计算得到的Activity窗口的宽度和高度保存在ViewRoot类的成员变量mWidth和mHeight中。

       我们继续往下阅读代码：

**boolean** focusChangedDueToTouchMode = ensureTouchModeLocally(

(relayoutResult&WindowManagerImpl.RELAYOUT\_IN\_TOUCH\_MODE) != 0);

**if** (focusChangedDueToTouchMode || mWidth != host.mMeasuredWidth

|| mHeight != host.mMeasuredHeight || contentInsetsChanged) {

childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width);

childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height);

......

// Ask host how big it wants to be

host.measure(childWidthMeasureSpec, childHeightMeasureSpec);

// Implementation of weights from WindowManager.LayoutParams

// We just grow the dimensions as needed and re-measure if

// needs be

**int** width = host.mMeasuredWidth;

**int** height = host.mMeasuredHeight;

**boolean** measureAgain = **false**;

**if** (lp.horizontalWeight > 0.0f) {

width += (**int**) ((mWidth - width) \* lp.horizontalWeight);

childWidthMeasureSpec = MeasureSpec.makeMeasureSpec(width,

MeasureSpec.EXACTLY);

measureAgain = **true**;

}

**if** (lp.verticalWeight > 0.0f) {

height += (**int**) ((mHeight - height) \* lp.verticalWeight);

childHeightMeasureSpec = MeasureSpec.makeMeasureSpec(height,

MeasureSpec.EXACTLY);

measureAgain = **true**;

}

**if** (measureAgain) {

......

host.measure(childWidthMeasureSpec, childHeightMeasureSpec);

}

mLayoutRequested = **true**;

}

       这段代码用来检查是否需要重新测量Activity窗口的大小。如果满足以下条件之一，那么就需要重新测量：

       1. Activity窗口的触摸模式发生了变化，并且由此引发了Activity窗口当前获得焦点的控件发生了变化，即变量focusChangedDueToTouchMode的值等于true。这个检查是通过调用ViewRoot类的成员函数ensureTouchModeLocally来实现的。

       2. Activity窗口前面测量出来的宽度host.mMeasuredWidth和高度host.mMeasuredHeight不等于WindowManagerService服务计算出来的宽度mWidth和高度mHeight。

       3. Activity窗口的内容区域边衬大小和可见区域边衬大小发生了变化，即前面得到的变量contentInsetsChanged的值等于true。

       重新计算了一次之后，如果Activity窗口的属性lp表明需要对测量出来的宽度width和高度height进行扩展，即变量lp所指向的一个WindowManager.LayoutParams对象的成员变量horizontalWeight和verticalWeight的值大于0.0，那么就需要对Activity窗口的顶层视图host的最大可用空间进行扩展后再进行一次测量工作。

       我们继续往下阅读最后一段代码：

**final** **boolean** didLayout = mLayoutRequested;

......

**if** (didLayout) {

......

host.layout(0, 0, host.mMeasuredWidth, host.mMeasuredHeight);

......

}

**if** (computesInternalInsets) {

ViewTreeObserver.InternalInsetsInfo insets = attachInfo.mGivenInternalInsets;

**final** Rect givenContent = attachInfo.mGivenInternalInsets.contentInsets;

**final** Rect givenVisible = attachInfo.mGivenInternalInsets.visibleInsets;

givenContent.left = givenContent.top = givenContent.right

= givenContent.bottom = givenVisible.left = givenVisible.top

= givenVisible.right = givenVisible.bottom = 0;

attachInfo.mTreeObserver.dispatchOnComputeInternalInsets(insets);

Rect contentInsets = insets.contentInsets;

Rect visibleInsets = insets.visibleInsets;

**if** (mTranslator != **null**) {

contentInsets = mTranslator.getTranslatedContentInsets(contentInsets);

visibleInsets = mTranslator.getTranslatedVisbileInsets(visibleInsets);

}

**if** (insetsPending || !mLastGivenInsets.equals(insets)) {

mLastGivenInsets.set(insets);

**try** {

sWindowSession.setInsets(mWindow, insets.mTouchableInsets,

contentInsets, visibleInsets);

} **catch** (RemoteException e) {

}

}

}

......

}

......

       经过前面漫长的操作后，Activity窗口的大小测量工作终于尘埃落定，这时候就可以对Activity窗口的内容进行布局了，前提是ViewRoot类的成员变量mLayoutRequest的值等于true。对Activity窗口的内容进行布局是通过调用它的顶层视图host的成员函数layout来实现的，这个过程可以参考前面Android应用程序窗口（Activity）的测量（Measure）、布局（Layout）和绘制（Draw）过程分析一文。

       从前面的描述可以知道，当变量computesInternalInsets的值等于true时，就表示Activity窗口指定有额外的内容区域边衬和可见区域边衬，这时候就是时候把它们告诉给WindowManagerService服务了，以便WindowManagerService服务下次可以知道Activity窗口的真实布局。Activity窗口额外指定的内容区域边衬大小和可见区域边衬大小是通过调用变量attachInfo所指向的一个AttachInfo对象的成员变量mTreeObserver所描述的一个TreeObserver对象的成员函数dispatchOnComputeInternalInsets来计算的。计算完成之后，就会保存在变量attachInfo所指向的一个AttachInfo对象的成员变量mGivenInternalInsets中，并且会通过ViewRoot类的静态成员变量sWindowSession所指向一个Binder代理对象来设置到WindowManagerService服务中去。

       注意，如果ViewRoot类的成员变量mTranslator指向了一个Translator对象，那么就说明Activity窗口是运行兼容模式中，这时候就需要将前面计算得到的内容区域边衬大小和可见区域边衬大小转化到兼容模式下，然后才可以保存在变量attachInfo所指向的一个AttachInfo对象的成员变量mGivenInternalInsets中，以及设置到WindowManagerService服务中去。

       另外，只有前面得到的变量insetsPending的值等于true，即Activity窗口正在等待告诉WindowManagerService服务它有额外指定的内容区域边衬和可见区域边衬，或者Activty窗口额外指定的内容区域边衬和可见区域边衬发生了变化，即Activty窗口上一次额外指定的内容区域边衬和可见区域边衬mLastGivenInsets不等于当前这次指定的内容区域边衬和可见区域边衬insets，Activity窗口额外指定的内容区域边衬和可见区域边衬才会被设置到WindowManagerService服务中去。

       ViewRoot类的成员函数再接下来的工作就是绘制Activity窗口的UI了，这个过程同样可以参考前面Android应用程序窗口（Activity）的测量（Measure）、布局（Layout）和绘制（Draw）过程分析一文。

       接下来，我们继续分析ViewRoot类的成员函数relayoutWindow的实现，以便可以了解它是如何请求WindowManagerService服务计算Activity窗口的大小的。

       Step 2. ViewRoot.relayoutWindow

**public** **final** **class** **ViewRoot** **extends** **Handler** **implements** **ViewParent**,

**View**.**AttachInfo**.**Callbacks** {

......

**private** **int** **relayoutWindow**(WindowManager.LayoutParams params, **int** viewVisibility,

**boolean** insetsPending) **throws** RemoteException {

**float** appScale = mAttachInfo.mApplicationScale;

......

**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);

......

**if** (mTranslator != **null**) {

mTranslator.translateRectInScreenToAppWinFrame(mWinFrame);

mTranslator.translateRectInScreenToAppWindow(mPendingContentInsets);

mTranslator.translateRectInScreenToAppWindow(mPendingVisibleInsets);

}

**return** relayoutResult;

}

......

}

      这个函数定义在文件frameworks/base/core/java/android/view/ViewRoot.java中。

      从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，ViewRoot类的静态成员变量sWindowSession是一个Binder代理对象，它引用了运行在WindowManagerService服务这一侧的一个Session对象，ViewRoot类的成员函数relayoutWindow通过调用这个Session对象的成员函数relayout来请求WindowManagerService服务计算Activity窗口的大小，其中，传递给WindowManagerService服务的参数包括：

      1. ViewRoot类的成员变量mWindow，用来标志要计算的是哪一个Activity窗口的大小。

      2. Activity窗口的顶层视图经过测量后得到的宽度和高度。注意，传递给WindowManagerService服务的宽度和高度是已经考虑了Activity窗口所设置的缩放因子了的。

      3. Activity窗口的可见状态，即参数viewVisibility。

      4. Activity窗口是否有额外的内容区域边衬和可见区域边衬等待告诉给WindowManagerService服务，即参数insetsPending。

      5. ViewRoot类的成员变量mWinFrame，这是一个输出参数，用来保存WindowManagerService服务计算后得到的Activity窗口的大小。

      6. ViewRoot类的成员变量mPendingContentInsets，这是一个输出参数，用来保存WindowManagerService服务计算后得到的Activity窗口的内容区域边衬大小。

      7. ViewRoot类的成员变量mPendingVisibleInsets，这是一个输出参数，用来保存WindowManagerService服务计算后得到的Activity窗口的可见区域边衬大小。

      8. ViewRoot类的成员变量mPendingConfiguration，这是一个输出参数，用来保存WindowManagerService服务返回来的Activity窗口的配置信息。

      9. ViewRoot类的成员变量mSurface，这是一个输出参数，用来保存WindowManagerService服务返回来的Activity窗口的绘图表面。

      得到了Activity窗口的大小以及内容区域边衬大小和可见区域边衬大小之后，如果Activity窗口是运行在兼容模式中，即ViewRoot类的成员变量mTranslator指向了一个Translator对象，那么就需要调用它的成员函数translateRectInScreenToAppWindow来对它们进行转换。

      接下来，我们继续分析Session类的成员函数relayout，以便可以了解WindowManagerService服务是如何计算一个Activity窗口的大小的。

      Step 3. Session.relayout

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **final** **class** **Session** **extends** **IWindowSession**.**Stub**

**implements** **IBinder**.**DeathRecipient** {

......

**public** **int** **relayout**(IWindow window, WindowManager.LayoutParams attrs,

**int** requestedWidth, **int** requestedHeight, **int** viewFlags,

**boolean** insetsPending, Rect outFrame, Rect outContentInsets,

Rect outVisibleInsets, Configuration outConfig, Surface outSurface) {

//Log.d(TAG, ">>>>>> ENTERED relayout from " + Binder.getCallingPid());

**int** res = relayoutWindow(**this**, window, attrs,

requestedWidth, requestedHeight, viewFlags, insetsPending,

outFrame, outContentInsets, outVisibleInsets, outConfig, outSurface);

//Log.d(TAG, "<<<<<< EXITING relayout to " + Binder.getCallingPid());

**return** res;

}

......

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       Session类的成员函数relayout的实现很简单，它只是调用了WindowManagerService类的成员函数relayoutWindow来进一步计算参数window所描述的一个Activity窗品的大小，接下来我们就继续分析WindowManagerService类的成员函数relayoutWindow的实现。

       Step 4. WindowManagerService.relayoutWindow

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**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) {

......

**synchronized**(mWindowMap) {

WindowState win = windowForClientLocked(session, client, **false**);

......

win.mRequestedWidth = requestedWidth;

win.mRequestedHeight = requestedHeight;

......

**final** **boolean** scaledWindow =

((win.mAttrs.flags & WindowManager.LayoutParams.FLAG\_SCALED) != 0);

**if** (scaledWindow) {

// requested{Width|Height} Surface's physical size

// attrs.{width|height} Size on screen

win.mHScale = (attrs.width != requestedWidth) ?

(attrs.width / (**float**)requestedWidth) : 1.0f;

win.mVScale = (attrs.height != requestedHeight) ?

(attrs.height / (**float**)requestedHeight) : 1.0f;

} **else** {

win.mHScale = win.mVScale = 1;

}

......

win.mGivenInsetsPending = insetsPending;

......

performLayoutAndPlaceSurfacesLocked();

......

outFrame.set(win.mFrame);

outContentInsets.set(win.mContentInsets);

outVisibleInsets.set(win.mVisibleInsets);

......

}

**return** (inTouchMode ? WindowManagerImpl.RELAYOUT\_IN\_TOUCH\_MODE : 0)

| (displayed ? WindowManagerImpl.RELAYOUT\_FIRST\_TIME : 0);

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       参数client是一个Binder代理对象，它引用了运行在应用程序进程这一侧中的一个W对象，用来标志一个Activity窗口。从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，在应用程序进程这一侧的每一个W对象，在WindowManagerService服务这一侧都有一个对应的WindowState对象，用来描述一个Activity窗口的状态。因此，WindowManagerService类的成员函数relayoutWindow首先通过调用另外一个成员函数windowForClientLocked来获得与参数client所对应的一个WindowState对象win，以便接下来可以对它进行操作。

       本文我们只关注WindowManagerService类的成员函数relayoutWindow中与窗口大小计算有关的逻辑，计算过程如下所示：

       1. 参数requestedWidth和requestedHeight描述的是应用程序进程请求设置Activity窗口中的宽度和高度，它们会被记录在WindowState对象win的成员变量mRequestedWidth和mRequestedHeight中。

       2. WindowState对象win的成员变量mAttr，它指向的是一个WindowManager.LayoutParams对象，用来描述Activity窗口的布局参数。其中，这个WindowManager.LayoutParams对象的成员变量width和height是用来描述Activity窗口的宽度和高度的。当这个WindowManager.LayoutParams对象的成员变量flags的WindowManager.LayoutParams.FLAG\_SCALED位不等于0的时候，就说明需要给Activity窗口的大小设置缩放因子。缩放因子分为两个维度，分别是宽度缩放因子和高度缩放因子，保存在WindowState对象win的成员变量HScale和VScale中，计算方法分别是用应用程序进程请求设置Activity窗口中的宽度和高度除以Activity窗口在布局参数中所设置的宽度和高度。

       3. 参数insetsPending用来描述Activity窗口是否有额外的内容区域边衬和可见区域边衬未设置，它被记录在WindowState对象win的成员变量mGivenInsetsPending中。

       4. 调用WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked来计算Activity窗口的大小。计算完成之后，参数client所描述的Activity窗口的大小、内容区域边衬大小和可见区域边边衬大小就会分别保存在WindowState对象win的成员变量mFrame、mContentInsets和mVisibleInsets中。

       5. 将WindowState对象win的成员变量mFrame、mContentInsets和mVisibleInsets的值分别拷贝到参数出数outFrame、outContentInsets和outVisibleInsets中，以便可以返回给应用程序进程。

       经过上述五个操作后，Activity窗口的大小计算过程就完成了，接下来我们继续分析WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked的实现，以便可以详细了解Activity窗口的大小计算过程。

       Step 5. WindowManagerService.performLayoutAndPlaceSurfacesLocked

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **final** **void** **performLayoutAndPlaceSurfacesLocked**() {

**if** (mInLayout) {

......

**return**;

}

......

**boolean** recoveringMemory = **false**;

**if** (mForceRemoves != **null**) {

recoveringMemory = **true**;

// Wait a little it for things to settle down, and off we go.

**for** (**int** i=0; i<mForceRemoves.size(); i++) {

WindowState ws = mForceRemoves.get(i);

Slog.i(TAG, "Force removing: " + ws);

removeWindowInnerLocked(ws.mSession, ws);

}

mForceRemoves = **null**;

......

}

mInLayout = **true**;

**try** {

performLayoutAndPlaceSurfacesLockedInner(recoveringMemory);

**int** i = mPendingRemove.size()-1;

**if** (i >= 0) {

**while** (i >= 0) {

WindowState w = mPendingRemove.get(i);

removeWindowInnerLocked(w.mSession, w);

i--;

}

mPendingRemove.clear();

mInLayout = **false**;

assignLayersLocked();

mLayoutNeeded = **true**;

performLayoutAndPlaceSurfacesLocked();

} **else** {

mInLayout = **false**;

......

}

......

} **catch** (RuntimeException e) {

mInLayout = **false**;

......

}

}

......

}

      这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

      从WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked的名称可以推断出，它执行的操作绝非是计算窗口大小这么简单。计算窗口大小只是其中的一个小小功能点，它主要的功能是用来刷新系统的UI。在我们这个情景中，为什么需要刷新系统的UI呢？Activity窗口在其属性发生了变化，例如，可见性、大小发生了变化，又或者它新增、删除了子视图，都需要重新计算大小，而这些变化都是要求WindowManagerService服务重新刷新系统的UI的。事实上，刷新系统的UI是WindowManagerService服务的主要任务，在新增和删除了窗口、窗口动画显示过程、窗口切换过程中，WindowManagerService服务都需要不断地刷新系统的UI。

      WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked主要是通过调用另外一个成员函数performLayoutAndPlaceSurfacesLockedInner来刷新系统的UI的，而在刷新的过程中，就会对系统中的各个窗口的大小进行计算。

      在调用成员函数performLayoutAndPlaceSurfacesLockedInner来刷新系统UI的前后，WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked还会执行以下两个操作：

      1. 调用前，检查系统中是否存在强制删除的窗口。有内存不足的情况下，有一些窗口就会被回收，即要从系统中删除，这些窗口会保存在WindowManagerService类的成员变量mForceRemoves所描述的一个ArrayList中。如果存在这些窗口，那么WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked就会调用另外一个成员函数removeWindowInnerLocked来删除它们，以便可以回收它们所占用的内存。

      2. 调用后，检查系统中是否有窗口需要移除。如果有的话，那么WindowManagerService类的成员变量mPendingRemove所描述的一个ArrayList的大小就会大于0。这种情况下，WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked就会调用另外一个成员函数removeWindowInnerLocked来移除这些窗口。注意，WindowManagerService类的成员函数removeWindowInnerLocked只是用来移除窗口，但是并没有回收这些窗口所占用的内存。等到合适的时候，例如，内存不足时，才会考虑回收这些窗口所占用的内存。移除一个窗口的操作也是很复杂的，除了要将窗口从WindowManagerService类的相关成员变量中移除之外，还要考虑重新调整输入法窗口和壁纸窗口，因为被移除的窗口可能要求显示壁纸和输入法窗口，当它被移除之后，就要将壁纸窗口和输入法窗口调整到合适的Z轴位置上去，以便可以交给下一个需要显示壁纸和输入法窗口的窗口使用。此外，在移除了窗口之后，WindowManagerService服务还需要重新计算现存的其它窗口的Z轴位置，以便可以正确地反映系统当前的UI状态，这是通过调用WindowManagerService类的成员函数assignLayersLocked来实现的。重新计算了现存的其它窗口的Z轴位置之后，又需要再次刷新系统的UI，即要对WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked进行递归调用，并且在调用前，将WindowManagerService类的成员变量mLayoutNeeded的值设置为true。由此就可见，系统UI的刷新过程是非常复杂的。

      注意，为了防止在刷新系统UI的过程中被重复调用，WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked在刷新系统UI之前，即调用成员函数performLayoutAndPlaceSurfacesLockedInner之前，会将WindowManagerService类的成员变量mInLayout的值设置为true，并且在调用之后，重新将这个成员变量的值设置为false。这样，WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked就可以在一开始的时候检查成员变量mInLayout的值是否等于true，如果等于的话，那么就说明WindowManagerService服务正在刷新系统UI的过程中，于是就不用往下执行了。

      接下来，我们就继续分析WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLockedInner的实现，以便可以了解Activity窗口的大小计算过程。

      Step 6. WindowManagerService.performLayoutAndPlaceSurfacesLockedInner

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **final** **void** **performLayoutAndPlaceSurfacesLockedInner**(

**boolean** recoveringMemory) {

......

Surface.openTransaction();

......

**try** {

......

**int** repeats = 0;

**int** changes = 0;

do {

repeats++;

**if** (repeats > 6) {

......

**break**;

}

// FIRST LOOP: Perform a layout, if needed.

**if** (repeats < 4) {

changes = performLayoutLockedInner();

**if** (changes != 0) {

**continue**;

}

} **else** {

Slog.w(TAG, "Layout repeat skipped after too many iterations");

changes = 0;

}

// SECOND LOOP: Execute animations and update visibility of windows.

......

} **while** (changes != 0);

// THIRD LOOP: Update the surfaces of all windows.

......

} **catch** (RuntimeException e) {

......

}

......

Surface.closeTransaction();

......

// Destroy the surface of any windows that are no longer visible.

......

// Time to remove any exiting tokens?

......

// Time to remove any exiting applications?

......

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLockedInner是一个巨无霸的函数，它一共有1200+行代码，承载了WindowManagerService服务的核心功能。对于这样一个巨无霸函数，要逐行地分析它的实现是很困难的，因为要理解各种上下文信息，才可以清楚地知道它的执行过程。这里我们就大概地分析它的实现框架，以后再逐步地分析它的具体实现：

       1. 在一个最多执行7次的while循环中，做两件事情：第一件事情是计算各个窗品的大小，这是通过调用另外一个成员函数performLayoutLockedInner来实现的；第二件事情是执行窗口的动画，主要是处理窗口的启动窗口显示动画和窗口切换过程中的动画，以及更新各个窗口的可见性。注意，每一次while循环执行之后，如果发现系统中的各个窗口的相应布局属性不再发生变化，那么就不行执行下一次的while循环了，即该while循环可能不用执行7次就结束了。窗口的动画显示过程和窗口的可见性更新过程是相当复杂的，它们也是WindowManagerService服务最为核的地方，在后面的文章中，我们再详细分析。

       2. 经过第1点的操作之后，接下来就可以将各个窗口的属性，例如，大小、位置等属性，通知SurfaceFlinger服务了，也就是让SurfaceFlinger服务更新它里面的各个Layer的属性值，以便可以对这些Layer执行可见性计算、合成等操作，最后渲染到硬件帧缓冲区中去。SurfaceFlinger服务计算系统中各个窗口，即各个Layer的可见性，以便将它们合成、渲染到硬件帧缓冲区的过程可以参考前面Android系统Surface机制的SurfaceFlinger服务渲染应用程序UI的过程分析一文。注意，各个窗口的属性更新操作是被包含在SurfaceFlinger服务的一个事务中的，即一个Transaction中，这样做是为了避免每更新一个窗口的一个属性就触发SurfaceFlinger服务重新计算各个Layer的可见性，以及对各个Layer进行合并和渲染的操作。启动SurfaceFlinger服务的一个事务可以通过调用Surface类的静态成员函数openTransaction来实现，而关闭SurfaceFlinger服务的一个事务可以通过调用Surface类的静态成员函数closeTransaction来实现。

      3. 经过第1点和第2点的操作之后，一次系统UI的刷新过程就完成了，这时候就会将系统中的那些不会再显示的窗口的绘图表面销毁掉，并且将那些已经完成退出了的窗口令牌，即将我们在前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文中所提到的WindowToken移除掉，以及将那些已经退出了的Activity窗口令牌，即将我们在前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文中所提到的AppWindowToken也移除掉。这一步实际执行的是窗口清理操作。

      上述三个操作是WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLockedInner的实现关键所在，理解了这三个操作，基本也就可以理解WindowManagerService服务刷新系统UI的过程了。

      接下来，我们继续分析WindowManagerService类的成员函数performLayoutLockedInner的实现，以便可以继续了解Activity窗口的大小计算过程。

      Step 7. WindowManagerService.performLayoutLockedInner

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**final** WindowManagerPolicy mPolicy = PolicyManager.makeNewWindowManager();

......

/\*\*

\* Z-ordered (bottom-most first) list of all Window objects.

\*/

**final** ArrayList<WindowState> mWindows = **new** ArrayList<WindowState>();

......

**private** **final** **int** **performLayoutLockedInner**() {

......

**final** **int** dw = mDisplay.getWidth();

**final** **int** dh = mDisplay.getHeight();

**final** **int** N = mWindows.size();

**int** i;

......

mPolicy.beginLayoutLw(dw, dh);

**int** seq = mLayoutSeq+1;

**if** (seq < 0) seq = 0;

mLayoutSeq = seq;

// First perform layout of any root windows (not attached

// to another window).

**int** topAttached = -1;

**for** (i = N-1; i >= 0; i--) {

WindowState win = mWindows.get(i);

......

**final** AppWindowToken atoken = win.mAppToken;

**final** **boolean** gone = win.mViewVisibility == View.GONE

|| !win.mRelayoutCalled

|| win.mRootToken.hidden

|| (atoken != **null** && atoken.hiddenRequested)

|| win.mAttachedHidden

|| win.mExiting || win.mDestroying;

......

**if** (!gone || !win.mHaveFrame) {

**if** (!win.mLayoutAttached) {

mPolicy.layoutWindowLw(win, win.mAttrs, **null**);

win.mLayoutSeq = seq;

......

} **else** {

**if** (topAttached < 0) topAttached = i;

}

}

}

......

**for** (i = topAttached; i >= 0; i--) {

WindowState win = mWindows.get(i);

// If this view is GONE, then skip it -- keep the current

// frame, and let the caller know so they can ignore it

// if they want. (We do the normal layout for INVISIBLE

// windows, since that means "perform layout as normal,

// just don't display").

**if** (win.mLayoutAttached) {

......

**if** ((win.mViewVisibility != View.GONE && win.mRelayoutCalled)

|| !win.mHaveFrame) {

mPolicy.layoutWindowLw(win, win.mAttrs, win.mAttachedWindow);

win.mLayoutSeq = seq;

......

}

}

}

......

**return** mPolicy.finishLayoutLw();

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       在分析WindowManagerService类的成员函数performLayoutLockedInner的实现之前，我们首先介绍WindowManagerService类的两个成员变量mPolicy和mWindows：

       1. mPolicy指向的是一个窗口管理策略类，它是通过调用PolicyManager类的静态成员函数makeNewWindowManager来初始化的，在Phone平台中，它指向的是便是一个PhoneWindowManager对象，主要是用来制定窗口的大小计算策略。

       2. mWindows指向的是一个类型为WindowState的ArrayList，它里面保存的就是系统中的所有窗口，这些窗口是按照Z轴位置从小到大的顺序保存在这个ArrayList中的，也就是说，第i个窗口位于第i-1个窗口的上面，其中，i > 0。

       理解了这两个成员变量的含义之后，我们就分析WindowManagerService类的成员函数performLayoutLockedInner的执行过程，主要是分三个阶段：

       1. 准备阶段：调用PhoneWindowManager类的成员函数beginLayoutLw来设置屏幕的大小。屏幕的大小可以通过调用WindowManagerService类的成员变量mDisplay所描述的一个Display对象的成员函数getWidth和getHeight来获得。

       2. 计算阶段：调用PhoneWindowManager类的成员函数layoutWindowLw来计算各个窗口的大小、内容区域边衬大小以及可见区域边衬大小。

       3. 结束阶段：调用PhoneWindowManager类的成员函数finishLayoutLw来执行一些清理工作。

       按照父子关系来划分，系统中的窗口可以分为父窗口和子窗口两种。如果一个WindowState对象的成员变量mLayoutAttached的值等于false，那么它所描述的窗口就可以作为一个父窗口，否则的话，它所描述的窗口就是一个子窗口。由于子窗口的大小计算是依赖于其父窗口的，因此，在计算各个窗口的大小的过程中，即在上述的第2阶段中，按照以下方式来进行：

       1.  先计算父窗口的大小。一般来说，能够作为父窗口的，是那些Activity窗口。从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，如果一个窗口是Activity窗口，那么用来描述它的一个WindowState对象的成员变量mAppToken就不等于null，并且指向的是一个AppWindowToken对象。这个AppWindowToken对象主要是用来描述一个Activity，即与ActivityManagerService服务中的一个ActivityRecord对象对应。一个Activity窗口只有在两种情况下才会被计算大小：第一种情况是窗口不是处于不可见状态的；第二种情况是窗口从来还没有被计算过大小，即用来描述该Activity窗口的WindowState对象的成员变量mHaveFrame的值等于false，这种情况一般发生在窗口刚刚被添加到WindowManagerService的过程中。一个Activity窗口的不可见状态由它本身的状态、它所在的窗口结构树状态以及它所属的Activity的状态有关，也就是说，如果一个Activity窗口本身是可见的，但是由于它的父窗口、它所在的窗口组的根窗口或者它所属的Activity是不可见的，那么该Activity窗口也是不可见的。一个Activity窗口的不可见状态由以下因素决定：

       1). 它本身处于不可见状态，即对应的WindowState对象的成员变量mViewVisibility的值等于View.GONE；

       2). 它本身处于正在退出的状态，即对应的WindowState对象的成员变量mExiting的值等于true；

       3). 它本身处于正在销毁的状态，即对应的WindowState对象的成员变量mDestroying的值等于true；

       4). 它的父窗口处于不可见状态，即对应的WindowState对象的成员变量mAttachedHidden的值等于true；

       5). 它所在窗口结构树中的根窗口处于不可见状态，即对应的WindowState对象的成员变量mRootToken所描述的一个WindowToken对象的成员变量hidden的值等于true；

       6). 它所属的Activity处于不可见状态，即对应的WindowState对象的成员变量mAppToken所描述的一个AppWindowToken对象的成员变量hiddenRequested的值等于true。

       除了上述六个因素之外，如果一个Activity窗口没有被它所运行在的应用程序进程主动请求WindowManagerService服务对它进行布局，即对应的WindowState对象的成员变量mRelayoutCalled的值等于false，那么此时也是不需要计算Activity窗口的大小的。

      一个Activity窗口的大小一旦确定是需要计算大小之后，PhoneWindowManager类的成员函数layoutWindowLw就被调用来计算它的大小。

      2. 接着计算子窗口的大小。前面在计算父窗口的大小过程中，会记录位于系统最上面的一个子窗口在mWindows所描述的一个ArrayList的位置topAttached，接下来就可以从这个位置开始向下计算每一个子窗口的大小。一个子窗口在以下两种情况下，才会被计算大小：

      1). 它本身处于可见状态，即对应的WindowState对象的成员变量mViewVisibility的值不等于View.GONE，并且它所运行在的应用程序进程主动请求WindowManagerService服务对它进行布局，即对应的WindowState对象的成员变量mRelayoutCalled的值等于true。

      2). 它从来还没有被计算过大小，即用来描述该子窗口的WindowState对象的成员变量mHaveFrame的值等于false，这种情况一般发生在子窗口刚刚被添加到WindowManagerService的过程中。

      接下来，我们就分别分析PhoneWindowManager类的成员函数beginLayoutLw、layoutWindowLw和finishLayoutLw的实现，以便可以了解Activity窗口的大小计算过程。

      Step 8. PhoneWindowManager.beginLayoutLw

**public** **class** **PhoneWindowManager** **implements** **WindowManagerPolicy** {

......

WindowState mStatusBar = **null**;

......

// The current size of the screen.

**int** mW, mH;

// During layout, the current screen borders with all outer decoration

// (status bar, input method dock) accounted for.

**int** mCurLeft, mCurTop, mCurRight, mCurBottom;

// During layout, the frame in which content should be displayed

// to the user, accounting for all screen decoration except for any

// space they deem as available for other content. This is usually

// the same as mCur\*, but may be larger if the screen decor has supplied

// content insets.

**int** mContentLeft, mContentTop, mContentRight, mContentBottom;

// During layout, the current screen borders along with input method

// windows are placed.

**int** mDockLeft, mDockTop, mDockRight, mDockBottom;

// During layout, the layer at which the doc window is placed.

**int** mDockLayer;

**static** **final** Rect mTmpParentFrame = **new** Rect();

**static** **final** Rect mTmpDisplayFrame = **new** Rect();

**static** **final** Rect mTmpContentFrame = **new** Rect();

**static** **final** Rect mTmpVisibleFrame = **new** Rect();

......

**public** **void** **beginLayoutLw**(**int** displayWidth, **int** displayHeight) {

mW = displayWidth;

mH = displayHeight;

mDockLeft = mContentLeft = mCurLeft = 0;

mDockTop = mContentTop = mCurTop = 0;

mDockRight = mContentRight = mCurRight = displayWidth;

mDockBottom = mContentBottom = mCurBottom = displayHeight;

mDockLayer = 0x10000000;

// decide where the status bar goes ahead of time

**if** (mStatusBar != **null**) {

**final** Rect pf = mTmpParentFrame;

**final** Rect df = mTmpDisplayFrame;

**final** Rect vf = mTmpVisibleFrame;

pf.left = df.left = vf.left = 0;

pf.top = df.top = vf.top = 0;

pf.right = df.right = vf.right = displayWidth;

pf.bottom = df.bottom = vf.bottom = displayHeight;

mStatusBar.computeFrameLw(pf, df, vf, vf);

**if** (mStatusBar.isVisibleLw()) {

// If the status bar is hidden, we don't want to cause

// windows behind it to scroll.

mDockTop = mContentTop = mCurTop = mStatusBar.getFrameLw().bottom;

......

}

}

}

......

}

       这个函数定义在文件frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindowManager.java中。

       在分析PhoneWindowManager类的成员函数beginLayoutLw的实现之前，我们首先介绍PhoneWindowManager类的五组成员变量。

       第一组成员变量是mW和mH，它们分别用来描述当前这轮窗口大小计算过程的屏幕宽度和高度。

       第二组成员变量是mCurLeft、mCurTop、mCurRight和mCurBottom，它们组成一个四元组（mCurLeft, mCurTop, mCurRight, mCurBottom），用来描述当前这轮窗口大小计算过程的屏幕装饰区，它对应于前面所提到的Activity窗口的可见区域边衬。

       第三组成员变量是mContentLeft、mContentTop、mContentRight和mContentBottom，它们组成一个四元组（mContentLeft, mContentTop, mContentRight, mContentBottom），也是用来描述当前这轮窗口大小计算过程的屏幕装饰区，不过它对应的是前面所提到的Activity窗口的内容区域边衬。

       第四组成员变量是mDockLeft、mDockTop、mDockRight、mDockBottom和mDockLayer，其中，前四个成员变量组成一个四元组（mDockLeft, mDockTop, mDockRight, mDockBottom），用来描述当前这轮窗口大小计算过程中的输入法窗口所占据的位置，后一个成员变量mDockLayer用来描述输入法窗品的Z轴位置。

       第五组成员变量是mTmpParentFrame、mTmpDisplayFrame、mTmpContentFrame和mTmpVisibleFrame，它们是一组临时Rect区域，用来作为参数传递给具体的窗口计算大小的，避免每次都创建一组新的Rect区域来作来参数传递窗口。

       除了这五组成员变量之外，PhoneWindowManager类还有一个成员变量mStatusBar，它的类型为WindowState，用来描述系统的状态栏。

       理解了这些成员变量的含义之后，PhoneWindowManager类的成员函数beginLayoutLw的实现就容易理解了，它主要做了以下两件事情：

       1. 初始化前面所提到的四组成员变量，其中，mW和mH设置为参数displayWidth和displayHeight所指定的屏幕宽度和高度，并且使得（mCurLeft, mCurTop, mCurRight, mCurBottom）、（mContentLeft, mContentTop, mContentRight, mContentBottom）和（mDockLeft, mDockTop, mDockRight, mDockBottom）这三个区域的大小等于屏幕的大小。

       2. 计算状态栏的大小。状态栏的大小一经确定，并且它是可见的，那么就会修改成员变量mCurLeft、mContentLeft和mDockLeft的值为状态栏的所占据的区域的下边界位置，这样就可以将（mCurLeft, mCurTop, mCurRight, mCurBottom）、（mContentLeft, mContentTop, mContentRight, mContentBottom）和（mDockLeft, mDockTop, mDockRight, mDockBottom）这三个区域限制为剔除状态栏区域之后所得到的屏幕区域。

       还有另外一个地方需要注意的是，输入法窗口的Z轴被初始化为0x10000000，这个值是相当大的了，可以保证输入法窗口作为顶层窗口出现。

       这一步执行完成之后，返回到前面的Step 7中，即WindowManagerService类的成员函数performLayoutLockedInner，接下来就会调用PhoneWindowManager类的成员函数layoutWindowLw来计算系统中各个可见窗口的大小。

       Step 9. PhoneWindowManager.layoutWindowLw

**public** **class** **PhoneWindowManager** **implements** **WindowManagerPolicy** {

......

**public** **void** **layoutWindowLw**(WindowState win, WindowManager.LayoutParams attrs,

WindowState attached) {

// we've already done the status bar

**if** (win == mStatusBar) {

**return**;

}

......

**final** **int** fl = attrs.flags;

**final** **int** sim = attrs.softInputMode;

**final** Rect pf = mTmpParentFrame;

**final** Rect df = mTmpDisplayFrame;

**final** Rect cf = mTmpContentFrame;

**final** Rect vf = mTmpVisibleFrame;

**if** (attrs.type == TYPE\_INPUT\_METHOD) {

pf.left = df.left = cf.left = vf.left = mDockLeft;

pf.top = df.top = cf.top = vf.top = mDockTop;

pf.right = df.right = cf.right = vf.right = mDockRight;

pf.bottom = df.bottom = cf.bottom = vf.bottom = mDockBottom;

// IM dock windows always go to the bottom of the screen.

attrs.gravity = Gravity.BOTTOM;

mDockLayer = win.getSurfaceLayer();

} **else** {

**if** ((fl &

(FLAG\_LAYOUT\_IN\_SCREEN | FLAG\_FULLSCREEN | FLAG\_LAYOUT\_INSET\_DECOR))

== (FLAG\_LAYOUT\_IN\_SCREEN | FLAG\_LAYOUT\_INSET\_DECOR)) {

// This is the case for a normal activity window: we want it

// to cover all of the screen space, and it can take care of

// moving its contents to account for screen decorations that

// intrude into that space.

**if** (attached != **null**) {

// If this window is attached to another, our display

// frame is the same as the one we are attached to.

setAttachedWindowFrames(win, fl, sim, attached, **true**, pf, df, cf, vf);

} **else** {

pf.left = df.left = 0;

pf.top = df.top = 0;

pf.right = df.right = mW;

pf.bottom = df.bottom = mH;

**if** ((sim & SOFT\_INPUT\_MASK\_ADJUST) != SOFT\_INPUT\_ADJUST\_RESIZE) {

cf.left = mDockLeft;

cf.top = mDockTop;

cf.right = mDockRight;

cf.bottom = mDockBottom;

} **else** {

cf.left = mContentLeft;

cf.top = mContentTop;

cf.right = mContentRight;

cf.bottom = mContentBottom;

}

vf.left = mCurLeft;

vf.top = mCurTop;

vf.right = mCurRight;

vf.bottom = mCurBottom;

}

}

......

}

win.computeFrameLw(pf, df, cf, vf);

// Dock windows carve out the bottom of the screen, so normal windows

// can't appear underneath them.

**if** (attrs.type == TYPE\_INPUT\_METHOD && !win.getGivenInsetsPendingLw()) {

**int** top = win.getContentFrameLw().top;

top += win.getGivenContentInsetsLw().top;

**if** (mContentBottom > top) {

mContentBottom = top;

}

top = win.getVisibleFrameLw().top;

top += win.getGivenVisibleInsetsLw().top;

**if** (mCurBottom > top) {

mCurBottom = top;

}

......

}

}

......

}

       这个函数定义在文件frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindowManager.java中。

       第一个参数win描述的是当前要计算大小的窗口，第二个参数attrs描述的是窗口win的布局参数，第三个参数attached描述的是窗口win的父窗口，如果它的值等于null，就表示窗口win没有父窗口。

       PhoneWindowManager类的成员函数layoutWindowLw会根据窗口win的是子窗口还是全屏窗口及其软键盘显示模式来决定它的大小如何计算。这里我们只关注输入法窗口和非全屏的Activity窗口的大小计算方式，其它类型的窗口大小计算方式是差不多的。

       从前面的Step 8可以知道，系统的状态栏大小已经计算过了，因此，PhoneWindowManager类的成员函数layoutWindowLw如果发现参数win描述的正好是状态栏的话，它就什么也不做就返回了。

       在计算一个窗口的大小的时候，我们需要四个参数。第一个参数是父窗口的大小pf，第二个参数是屏幕的大小df，第三个参数是内容区域边衬大小cf，第四个参数是可见区域边衬大小vf。

       如果参数win描述的是输入法窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_INPUT\_METHOD，那么上述四个用来计算窗口大小的区域pf、df、cf和vf就等于PhoneWindowManager类的成员变量mDockLeft、mDockTop、mDockRight和mDockBottom所组成的区域的大小。

       如果参数win描述的是一个非全屏的Activity窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_LAYOUT\_IN\_SCREEN位和FLAG\_LAYOUT\_INSET\_DECOR位等于1，那么PhoneWindowManager类的成员函数layoutWindowLw就会继续检查参数attached的值是否不等于null。如果不等于null的话，那么就说明参数win所描述的一个非全屏的Activity窗口附加在其它窗口上，即它具有一个父窗口，这时候就会调用另外一个成员函数setAttachedWindowFrames来计算它的大小。

       接下来我们就只关注参数win描述的是一个非全屏的、并且没有附加到其它窗口的Activity窗口的大小计算过程。

       首先，父窗口大小pf和屏幕大小df都会被设置为整个屏幕区域的大小。

       其次，可见区域边衬大小vf被设置为PhoneWindowManager类的成员变量mCurLeft、mCurTop、mCurRight和mCurBottom所组成的区域的大小。

       第三，内容区域边衬大小cf的计算相对复杂一些，需要考虑窗口win的软键盘显示模式sim的值。如果变量sim的SOFT\_INPUT\_ADJUST\_RESIZE位等于1，那么就意味着窗口win在出向输入法窗口的时候，它的内容要重新进行排布，避免被输入法窗口挡住，因此，这时候窗口win的内容区域大小就会等于PhoneWindowManager类的成员变量mContentLeft、mContentTop、mContentRight和mContentBottom所组成的区域的大小。另一方面，如果变量sim的SOFT\_INPUT\_ADJUST\_RESIZE位等于0，那么就意味着窗口win在出向输入法窗口的时候，它的内容不需要重新进行排布，这时候它的内容区域大小就会等于PhoneWindowManager类的成员变量mDockLeft、mDockTop、mDockRight和mDockBottom所组成的区域的大小。注意，PhoneWindowManager类的成员变量mDockLeft、mDockTop、mDockRight和mDockBottom所组成的区域的大小并不是等于输入法窗口的大小的，而是包含了输入法窗口所占据的区域的大小，这就意味着输入法窗口与窗口win会有重叠的部分，或者说输入法窗口覆盖了窗口win的一部分。

       得到了用来计算窗口win四个参数pf、 df、cf和vf之后，就可以调用参数win所描述的一个WindowState对象的成员函数computeFrameLw来计算窗口win的具体大小了。计算的结果便得到了窗口win的大小，以及它的内容区域边衬大小和可见区域边衬大小。注意，窗口经过计算后得到的内容区域边衬大小和可见区域边衬大小并不一定是等于参数cf和vf所指定的大小的。

       计算完成窗口win的大小之后，PhoneWindowManager类的成员函数layoutWindowLw还会检查窗口win是否是一个输入法窗口，并且它是否指定了额外的内容区域边衬和可见区域边衬。如果这两个条件都成立的话，那么就需要相应地调整PhoneWindowManager类的成员变量mContentBottom和mCurBottom的值，以便使得PhoneWindowManager类的成员变量是mContentLeft、mContentTop、mContentRight和mContentBottom所围成的内容区域和成员变量mCurLeft、mCurTop、mCurRight和mCurBottom所围成的可见区域不会覆盖到输入法窗口额外指定的内容区域边衬和可见区域边衬。

       接下来，我们就继续分析WindowState类的成员函数computeFrameLw的实现，以便可以了解Activity窗口的大小计算的具体过程。

       Step 10. WindowState.computeFrameLw

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **final** **class** **WindowState** **implements** **WindowManagerPolicy**.**WindowState** {

......

**boolean** mHaveFrame;

......

// "Real" frame that the application sees.

**final** Rect mFrame = **new** Rect();

......

**final** Rect mContainingFrame = **new** Rect();

**final** Rect mDisplayFrame = **new** Rect();

**final** Rect mContentFrame = **new** Rect();

**final** Rect mVisibleFrame = **new** Rect();

**public** **void** **computeFrameLw**(Rect pf, Rect df, Rect cf, Rect vf) {

mHaveFrame = **true**;

**final** Rect container = mContainingFrame;

container.set(pf);

**final** Rect display = mDisplayFrame;

display.set(df);

**if** ((mAttrs.flags & FLAG\_COMPATIBLE\_WINDOW) != 0) {

container.intersect(mCompatibleScreenFrame);

**if** ((mAttrs.flags & FLAG\_LAYOUT\_NO\_LIMITS) == 0) {

display.intersect(mCompatibleScreenFrame);

}

}

**final** **int** pw = container.right - container.left;

**final** **int** ph = container.bottom - container.top;

**int** w,h;

**if** ((mAttrs.flags & mAttrs.FLAG\_SCALED) != 0) {

w = mAttrs.width < 0 ? pw : mAttrs.width;

h = mAttrs.height< 0 ? ph : mAttrs.height;

} **else** {

w = mAttrs.width == mAttrs.MATCH\_PARENT ? pw : mRequestedWidth;

h = mAttrs.height== mAttrs.MATCH\_PARENT ? ph : mRequestedHeight;

}

**final** Rect content = mContentFrame;

content.set(cf);

**final** Rect visible = mVisibleFrame;

visible.set(vf);

**final** Rect frame = mFrame;

**final** **int** fw = frame.width();

**final** **int** fh = frame.height();

......

Gravity.apply(mAttrs.gravity, w, h, container,

(**int**) (mAttrs.x + mAttrs.horizontalMargin \* pw),

(**int**) (mAttrs.y + mAttrs.verticalMargin \* ph), frame);

......

// Now make sure the window fits in the overall display.

Gravity.applyDisplay(mAttrs.gravity, df, frame);

// Make sure the content and visible frames are inside of the

// final window frame.

**if** (content.left < frame.left) content.left = frame.left;

**if** (content.top < frame.top) content.top = frame.top;

**if** (content.right > frame.right) content.right = frame.right;

**if** (content.bottom > frame.bottom) content.bottom = frame.bottom;

**if** (visible.left < frame.left) visible.left = frame.left;

**if** (visible.top < frame.top) visible.top = frame.top;

**if** (visible.right > frame.right) visible.right = frame.right;

**if** (visible.bottom > frame.bottom) visible.bottom = frame.bottom;

**final** Rect contentInsets = mContentInsets;

contentInsets.left = content.left-frame.left;

contentInsets.top = content.top-frame.top;

contentInsets.right = frame.right-content.right;

contentInsets.bottom = frame.bottom-content.bottom;

**final** Rect visibleInsets = mVisibleInsets;

visibleInsets.left = visible.left-frame.left;

visibleInsets.top = visible.top-frame.top;

visibleInsets.right = frame.right-visible.right;

visibleInsets.bottom = frame.bottom-visible.bottom;

**if** (mIsWallpaper && (fw != frame.width() || fh != frame.height())) {

updateWallpaperOffsetLocked(**this**, mDisplay.getWidth(),

mDisplay.getHeight(), **false**);

}

......

}

......

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       WindowState类的成员变量mHaveFrame用来描述一个窗口的大小是否计算过了。当WindowState类的成员函数computeFrameLw被调用的时候，就说明一个相应的窗口的大小得到计算了，因此，WindowState类的成员函数computeFrameLw一开始就会将成员变量mHaveFrame的值设置为true。

       回忆一下，在前面的Step 9中提到，参数pf描述的是父窗口的大小，参数df描述的是屏幕的大小，参数cf描述的内容区域大小，参数vf描述的是可见区域大小，接下来我们就分析WindowState类的成员函数computeFrameLw是如何利用这些参数来计算一个窗口的大小的。

       WindowState类的成员变量mContainingFrame和mDisplayFrame描述的是当前正在处理的窗口的父窗口和屏幕的大小，它们刚好就分别等于参数pf和df的大小，因此，函数就直接将参数pf和df的值分别保存在WindowState类的成员变量mContainingFrame和mDisplayFrame中。如果当前正在处理的窗口运行在兼容模式，即WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_COMPATIBLE\_WINDOW位等于1，那么就需要将其父窗口的大小限制mContainingFrame在兼容模式下的屏幕区域中。兼容模式下的屏幕区域保存在WindowManagerService类的成员变量mCompatibleScreenFrame中，将父窗口的大小mContainingFrame与它执行一个相交操作，就可以将父窗品的大小限制兼容模式下的屏幕区域中。在当前正在处理的窗口运行在兼容模式的情况下，如果它的大小被限制在了兼容模式下的屏幕区域之中，即WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_LAYOUT\_NO\_LIMITS位等于0，那么同样需要将屏幕大小mDisplayFrame限制在兼容模式下的屏幕区域mCompatibleScreenFrame，这也是通过执行一个相交操作来完成的。

       WindowState类的成员变量mContentFrame和mVisibleFrame描述的是当前正在处理的窗口的内容区域和可见区域大小，它们刚好就分别等于参数cf和vf的大小，因此，函数就直接将参数cf和vf的值分别保存在WindowState类的成员变量mContainingFrame和mDisplayFrame中。现在，就剩下窗口的大小还没有计算。一旦窗口大小确定下来之后，就可以继续计算窗口的内容区域边衬和可见区域边衬大小了。接下来我们就继续分析窗口大小的计算过程。

        WindowState类的成员变量mFrame描述的就是当前正在处理的窗品的大小，我们的目标就是计算它的值。一个窗口的大小是受以下因素影响的：

        1. 是否指定了缩放因子。如果一个窗口的大小被指定了缩放因子，即WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_SCALED位等于1，那么该窗口的大小就是在它的布局参数中指定的，即是由WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量width和height所指定的。但是，如果在布局参数中指定的窗口宽度或者高度小于0，那么就会使用其父窗口的大小来作为当前窗口的大小。当前窗口的父窗口的宽度和高度分别保存在变量pw和ph中。

        2. 是否指定了等于父窗口的大小。如果一个窗口的大小被指定为其父窗口的大小，即WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量width和height的值等于mAttrs.MATCH\_PARENT，那么该窗口的大小就会等于其父窗口的大小，即等于变量pw和ph所描述的宽度和高度。另一方面，如果一个窗口的大小没有指定为其父窗口的大小的话，那么它的大小就会等于应用程序进程请求WindowManagerService所设置的大小，即等于WindowState类的成员变量mRequestedWidth和mRequestedHeight所描述的宽度和高度。

        经过上述2个操作之后，我们就初步地得到了窗口的宽度w和高度h，但是，它们还不是最终的窗口大小，还要进一步地根据窗口的Gravity属性来作调整，这个调整分两步进行：

       1. 根据窗口的Gravity值，以及位置、初始大小和父窗口大小，来计算窗口的大小，并且保存在变量frame中，即保存在WindowState类的成员变量mFrame中，这是通过调用Gravity类的静态成员函数apply来实现的。其中，窗口的初始大小保存在变量w和h中，父窗口大小保存在变量container中，即WindowState类的成员变量mContainingFrame中，位置保存在WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量x和y中。注意，如果窗口指定了相对父窗口的margin值，那么还需要相应的调整其位置值，即要在指定的位置值的基础上，再加上相对父窗口的margin值。一个窗口相对父窗口的margion是通过一个百分比来表示的，用这个百分比乘以父窗口的大小就可以得到绝对值。这个百分比又分为在水平方向和垂直方向两个值，分别保存在WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量horizontalMargin和verticalMargin中。

       2. 前面计算得到的窗口大小没有考虑在屏幕的大小，因此，接下来还需要继续调用Gravity类的静态成员函数applyDisplay来将前面计算得到的窗口大小限制在屏幕区域df中，即限制在WindowState类的成员变量mDisplayFrame所描述的区域中。

       经过上述2个操作之后，窗口的最终大小就保存在变量frame中了，即WindowState类的成员变量mFrame中，接下来就可以计算窗品的内容区域边衬和可见区域边衬大小了。

       内容区域边衬和可见区域边衬大小的计算很简单的，只要将窗口的大小frame，即WindowState类的成员变量mFrame所描述的区域，分别减去变量content和visible，即WindowState类的成员变量mContentFrame和mVisibleFrame所描述的区域，就可以得到窗口的内容区域边衬和可见区域边衬大小，它们分别保存在WindowState类的成员变量mContentInsets和mVisibleInsets中。注意，在计算窗口的内容区域边衬和可见区域边衬大小之前，首先要保证窗口的内容区域和可见区域包含在整个窗口区域中，这一点是由中间的8个if语句来保证的。

       窗口上一次的大小保存在变量fw和fh中。如果当前正在处理的窗口是一个壁纸窗口，即WindowState类的成员变量mIsWallpaper的值等于true，并且该窗口的大小发生了变化，即变量fw和fh的所描述的窗口大小不等于变量frame描述的窗口大小，那么就需要调用WindowManagerService类的成员函数updateWallpaperOffsetLocked来更新壁纸的位置。在后面的文章中，我们再详细描述系统的壁纸窗口的位置是如何计算的。

       这一步执行完成之后，一个窗口的大小就计算完成了。从计算的过程可以知道，整个窗口大小保存在WindowState类的成员变量mFrame中，而窗品的内容区域边衬大小和可见区域边衬大小分别保在WindowState类的成员变量mContentInsets和mVisibleInsets中。这些值最终会通过前面的Step 4返回给应用程序进程。

       返回到前面的Step 7中，即WindowManagerService类的成员函数performLayoutLockedInner，接下来就会调用PhoneWindowManager类的成员函数finishLayoutLw来结束当前这轮窗口大小的计算工作。

       Step 11. PhoneWindowManager.finishLayoutLw

**public** **class** **PhoneWindowManager** **implements** **WindowManagerPolicy** {

......

**public** **int** **finishLayoutLw**() {

**return** 0;

}

......

}

       这个函数定义在文件frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindowManager.java中。

       PhoneWindowManager类的成员函数finishLayoutLw是设计来结束一轮窗口大小的计算过程中，不过目前它什么也不做，只是一个空实现。

       至此，我们就分析完成Activity窗口的大小计算过程了。从这个计算过程中，我们就可以知道一个Activity窗口除了有一个整体大小之外，还有一个内容区域边衬大小和一个可见区域边衬大小。此外，我们还知道，一个Activity窗口的内容区域边衬大小和可见区域边衬大小是可能会受到与其所关联的输入法窗口的影响的，因为输入法窗口会叠加在该Activity窗口上面，这就涉及到了系统中的窗口的组织方式。在接下来的一篇文章中，我们就将继续分析WindowManagerService服务是如何组织系统中的窗口的。敬请关注！

**老罗的新浪微博：**[**http://weibo.com/shengyangluo**](http://weibo.com/shengyangluo)**，欢迎关注！**

本文出自 “[老罗的Android之旅](http://shyluo.blog.51cto.com/)” 博客，请务必保留此出处<http://shyluo.blog.51cto.com/5725845/1229286>

推荐：[Android窗口管理服务WindowManagerService切换Activity窗口（App Transition）的过程分析](http://www.itboth.com/d/6va63i/app-android-app-transition-transition-activity)

[在Android系统中，同一时刻只有一个Activity组件是处于激活状态的，因此，当ActivityManagerService服务激活了一个新的Activity组件时，它就需要通知WindowManagerService

# Android窗口管理服务WindowManagerService对壁纸窗口（Wallpaper Window）的管理分析

[审核此文](javascript:showCheckDoc('EJJJvy','http://www.itboth.com/d/EJJJvy/android-window-windowmanagerservice-wallpaper');)

itRec 分享于 2017-04-15 阅读 66 收藏 0

主题 [Android](http://www.itboth.com/tag/Android) [WindowManagerService](http://www.itboth.com/tag/WindowManagerService) [窗口管理](http://www.itboth.com/tag/%E7%AA%97%E5%8F%A3%E7%AE%A1%E7%90%86) [Android窗口管理](http://www.itboth.com/tag/Android%E7%AA%97%E5%8F%A3%E7%AE%A1%E7%90%86)

**领取地址**：

推荐：[Android窗口管理服务WindowManagerService显示窗口动画的原理分析](http://www.itboth.com/d/Q7BzM3/android)

[ 在前一文中，我们分析了Activity组件的切换过程。从这个过程可以知道，所有参与切换操作的窗口都会被设置切换动画。事实上，一个窗口在打开（关闭）的过程中，除了可

       在Android系统中，壁纸窗口和输入法窗口一样，都是一种特殊类型的窗口，而且它们都是喜欢和一个普通的Activity窗口缠绵在一起。大家可以充分地想象这样的一个3W场景：输入法窗口在上面，壁纸窗口在下面，Activity窗口夹在它们的中间。在前面一篇文章中，我们已经分析过输入法窗口是如何压在Activity窗口上面的了。在这篇文章中，我们就将继续分析壁纸窗口是如何贴在Activity窗口下面的。

       一个Activity窗口如果需要显示壁纸，那么它必须满足以下两个条件：

      1. 背景是半透明的，例如，它在AndroidManifest.xml文件中的android:theme属性设置为Theme.Translucent：

<**activity** android:name=".WallpaperActivity"

android:theme="@android:style/Theme.Translucent">

......

</**activity**>

      2. 窗口属性中的WindowManager.LayoutParams.FLAG\_SHOW\_WALLPAPER位设置为1：

<**activity** android:name=".WallpaperActivity"

android:theme="@android:style/Theme.Translucent">

......

</**activity**>

      满足了以上两个条件之后，Activity窗口和壁纸窗口的位置关系就如图1所示：

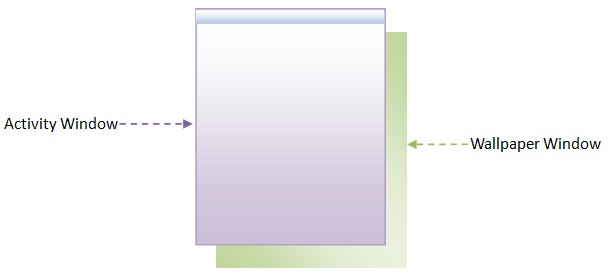


图1 Activity窗口和Wallpaper窗口的位置关系

       在前面Android窗口管理服务WindowManagerService组织窗口的方式分析一文中提到，WindowManagerService服务是使用堆栈来组织系统中的窗口的，因此，如果我们在窗口堆栈中观察Activity窗口和壁纸窗口，它们的位置关系就如图2所示：

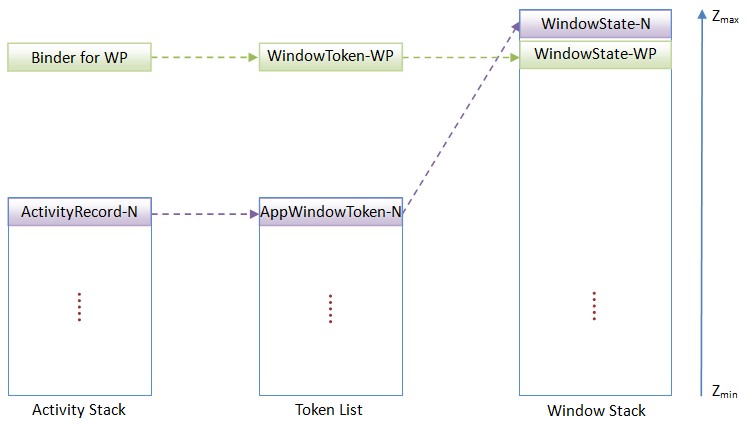


图2 Activity窗口和Wallpaper窗口在窗口堆栈中的位置关系

        图2中的对象的关系如下所示：

       1. 在ActivityManagerService服务内部的Activity组件堆栈顶端的ActivityRecord对象N描述的是系统当前激活的Activity组件。

       2. ActivityRecord对象N在WindowManagerService服务内部的窗口令牌列表顶端对应有一个AppWindowToken对象N。

       3. AppWindowToken对象N在WindowManagerService服务内部的窗口堆栈中对应有一个WindowState对象N，用来描述系统当前激活的Activity组件窗口。

       4. WindowState对象N下面有一个WindowState对象WP，用来描述系统中的壁纸窗口。

       5. 系统中的壁纸窗口在WindowManagerService服务内部中对应的窗口令牌是由WindowToken对象WP来描述的。

       6. WindowToken对象WP在WallpaperManagerService服务中对应有一个Binder对象。

       总的来说，就是图2描述了系统当前激活的Activity窗口需要显示壁纸的情景。WindowManagerService服务的职能之一就是要时刻关注系统中是否有窗口需要显示壁纸。WindowManagerService服务一旦发现有窗口需要显示壁纸，那么就会调整壁纸窗口在窗口堆栈中的位置，使得它放置在需要显示壁纸的窗口的下面。此外，需要显示壁纸的窗口还可以设置壁纸窗口在X轴和Y轴上的偏移位置，以便可以将壁纸窗口的某一部分指定为它的背景。

       接下来，我们就首先分析两个需要调整壁纸窗口在窗口堆栈中的位置的情景，然后再分析壁纸窗口在X轴和Y轴上的偏移位置的调整过程，最后分析壁纸窗口在窗口堆栈中的位置调整过程。

       一. 调整壁纸窗口在窗口堆栈中的位置的情景

       第一个需要调整壁纸窗口在窗口堆栈中的位置的情景是增加一个窗口到WindowManagerService服务去的时候。从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，增加一个窗口到WindowManagerService服务最终是通过调用WindowManagerService类的成员函数addWindow来实现的。接下来我们就主要分析这个函数中与壁纸窗口调整相关的逻辑，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**public** **int** **addWindow**(Session session, IWindow client,

WindowManager.LayoutParams attrs, **int** viewVisibility,

Rect outContentInsets, InputChannel outInputChannel) {

......

**synchronized**(mWindowMap) {

......

WindowToken token = mTokenMap.get(attrs.token);

**if** (token == **null**) {

......

**if** (attrs.type == TYPE\_WALLPAPER) {

......

**return** WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;

}

......

}

......

win = **new** WindowState(session, client, token,

attachedWindow, attrs, viewVisibility);

......

**if** (attrs.type == TYPE\_INPUT\_METHOD) {

......

} **else** **if** (attrs.type == TYPE\_INPUT\_METHOD\_DIALOG) {

......

} **else** {

addWindowToListInOrderLocked(win, **true**);

**if** (attrs.type == TYPE\_WALLPAPER) {

......

adjustWallpaperWindowsLocked();

} **else** **if** ((attrs.flags&FLAG\_SHOW\_WALLPAPER) != 0) {

adjustWallpaperWindowsLocked();

}

}

......

assignLayersLocked();

......

}

......

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       如果当前增加到WindowManagerService服务来的是一个壁纸窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_WALLPAPER，那么就要求与该壁纸窗口所对应的类型为WindowToken的窗口令牌已经存在，否则的话，WindowManagerService类的成员函数addWindow就会直接返回一个错误码WindowManagerImpl.ADD\_BAD\_APP\_TOKEN给调用者。这个类型为WindowToken的窗口令牌是WallpaperManagerService服务请求WindowManagerService服务创建的，即调用WindowManagerService类的成员函数addWindowToken来创建的，具体可以参考前面Android窗口管理服务WindowManagerService组织窗口的方式分析一文。

       如果当前增加到WindowManagerService服务来的既不是一个输入法窗口，也不是一个输入法对话框，那么WindowManagerService类的成员函数addWindow就会调用另外一个成员函数addWindowToListInOrderLocked来将前面为它所创建的一个WindowState对象win增加到窗口堆栈的合适位置上去。

       如果前面增加到窗口堆栈中的窗口是一个壁纸窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_WALLPAPER，或者是一个需要显示壁纸的窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量flags的值的FLAG\_SHOW\_WALLPAPER位等于1，那么就说明需要调整壁纸窗口在窗口堆栈中的位置，使得它位于需要显示壁纸的窗口的下面，这是通过调用WindowManagerService类的成员函数adjustWallpaperWindowsLocked来实现的。

       最后，由于增加了一个窗口到窗口堆栈中，以及窗口堆栈的窗口位置发生了变化，因此，就需要重新各个窗口的Z轴位置，这是通过调用WindowManagerService类的成员函数assignLayersLocked来实现的。

       在这个情景中，主要涉及到了WindowManagerService类的三个成员函数addWindowToListInOrderLocked、adjustWallpaperWindowsLocked和assignLayersLocked，其中，成员函数addWindowToListInOrderLocked的实现可以参考前面前面Android窗口管理服务WindowManagerService组织窗口的方式分析一文，成员函数assignLayersLocked的实现在接下来的一篇文章中再分析，本文主要是关注成员函数adjustWallpaperWindowsLocked的实现。

       第二个需要调整壁纸窗口在窗口堆栈中的位置的情景是一个应用程序进程请求WindowManagerService服务重新布局一个窗口的时候。从前面Android窗口管理服务WindowManagerService计算Activity窗口大小的过程分析一文可以知道，应用程序进程请求WindowManagerService服务重新布局一个窗口最终是通过调用WindowManagerService类的成员函数relayoutWindow来实现的。接下来我们就主要分析这个函数中与壁纸窗口调整相关的逻辑，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**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) {

**boolean** displayed = **false**;

......

**synchronized**(mWindowMap) {

WindowState win = windowForClientLocked(session, client, **false**);

......

**int** attrChanges = 0;

......

**if** (attrs != **null**) {

......

attrChanges = win.mAttrs.copyFrom(attrs);

}

......

**boolean** wallpaperMayMove = win.mViewVisibility != viewVisibility

&& (win.mAttrs.flags & FLAG\_SHOW\_WALLPAPER) != 0;

......

**if** (viewVisibility == View.VISIBLE &&

(win.mAppToken == **null** || !win.mAppToken.clientHidden)) {

displayed = !win.isVisibleLw();

......

**if** ((attrChanges&WindowManager.LayoutParams.FORMAT\_CHANGED) != 0) {

// To change the format, we need to re-build the surface.

win.destroySurfaceLocked();

displayed = **true**;

}

......

}

......

**boolean** assignLayers = **false**;

......

**if** (wallpaperMayMove) {

**if** ((adjustWallpaperWindowsLocked()&ADJUST\_WALLPAPER\_LAYERS\_CHANGED) != 0) {

assignLayers = **true**;

}

}

......

**if** (assignLayers) {

assignLayersLocked();

}

......

performLayoutAndPlaceSurfacesLocked();

**if** (displayed && win.mIsWallpaper) {

updateWallpaperOffsetLocked(win, mDisplay.getWidth(),

mDisplay.getHeight(), **false**);

}

......

}

......

**return** (inTouchMode ? WindowManagerImpl.RELAYOUT\_IN\_TOUCH\_MODE : 0)

| (displayed ? WindowManagerImpl.RELAYOUT\_FIRST\_TIME : 0);

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       应用程序进程在请求WindowManagerService服务重新布局一个窗口的时候，这个窗口的一些布局参数可能会发生变化，而这些变化可能会引发系统的壁纸窗口在窗口堆栈中的位置发生变化。如果系统的壁纸窗口在窗口堆栈中的位置发生了变化，那么就需要调整它们在窗口堆栈中的位置。

       WindowManagerService类的成员函数relayoutWindow首先调用根据参数session和client来调用另外一个成员函数windowForClientLocked，以便可以获得用来描述要重新布局的窗口的一个WindowState对象win。

       WindowState对象win的成员变量mViewVisibility描述的是窗口上一次布局时的可见性，而参数viewVisibility描述的是窗口当前的可见性，当它们的值不相等时，就意味着窗口的可见性发生了变化。在窗口的可见性发生了变化的情况下，如果正在请求重新布局的是一个需要显示壁纸的窗口，即WindowState对象win的成员变量mAttrs所指向的是一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_SHOW\_WALLPAPER位等于1，那么就说明可能需要调整壁纸窗口在窗口堆栈中的位置，以便它可以位于WindowState对象win所描述的窗口的下面，这时候变量wallpaperMayMove的值就会等于true。

       WindowManagerService类的成员函数relayoutWindow执行了一系列的其它操作之后，接下来就会判断变量wallpaperMayMove的值是否等于true。如果等于true的话，那么就会调用另外一个成员函数adjustWallpaperWindowsLocked来调整壁纸窗口在窗口堆栈中的位置，以便它可以位于需要显示壁纸的窗口的下面。WindowManagerService类的成员函数adjustWallpaperWindowsLocked的返回值是一个整数，当它的ADJUST\_WALLPAPER\_LAYERS\_CHANGED位等于1的时候，就说明壁纸窗口在窗口堆栈的位置发生了变化，于是就会将变量assignLayers的值设置为true，以便接下来可以调用WindowManagerService类的成员函数assignLayersLocked来重新计算系统中各个窗品的Z轴位置。

       变量displayed用来描述WindowState对象win所描述的窗口在当前布局中是由不可见变为可见的。在满足以下的条件之下，WindowState对象win所描述的窗口是由不可见变为可见的：

       1. 参数viewVisibility的值等于View.VISIBLE，即应用程序进程请求显示WindowState对象win所描述的窗口。

       2. WindowState对象win描述的是一个Activity窗口，即它的成员变量mAppToken不等于null，并且它所指向的AppWindowToken对象的成员变量clientHidden的值等于false，即WindowState对象win的窗口所对应的Activity组件当前是可见的。注意，如果WindowState对象win描述的不是一个Activity窗口，即它的成员变量mAppToken等于null，那么就可以忽略条件2。

       3. WindowState对象win所描述的窗口上一次是不可见的，即调用WindowState对象win的成员函数isVisibleLw的返回值等于false。

       此外，在满足条件1和条件2的情况下，如果WindowState对象win所描述的窗口的像素格式发生了变化，那么就需要将该窗口的绘图表面销毁掉，然后再重新创建一个，这时候也会认为该窗口由不可见变为了可见。

       参数attrs所指向的一个WindowManager.LayoutParams对象是用来保存WindowState对象win所描述的窗口在当前布局中所使用的布局参数的，而WindowState对象win的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象是用来保存WindowState对象win所描述的窗口在上一次布局所使用的布局参数的。在将参数attrs所指向的一个WindowManager.LayoutParams对象的内容拷贝到WindowState对象win的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的过程中，如果某些布局参数发生了变化，那么就会记录在变量attrChanges中。当变量attrChanges的WindowManager.LayoutParams.FORMAT\_CHANGED位等于1时，就说明WindowState对象win所描述的窗口的像素格式发生了变化，因此，WindowManagerService类的成员函数relayoutWindow就会调用WindowState对象win的成员函数destroySurfaceLocked来销毁该窗口的绘图表面，并且将变量displayed的值设置为true。

       WindowManagerService类的成员函数relayoutWindow调用另外一个成员函数performLayoutAndPlaceSurfacesLocked来对WindowState对象win所描述的窗口进行了布局之后，如果发现变量displayed的值等于true，并且WindowState对象win描述的是一个壁纸窗口，即它的成员变量mIsWallpaper的值等于true，那么还需要调用另外一个成员函数updateWallpaperOffsetLocked来重新计算该壁纸窗口在X轴和Y轴上的偏移位置，以便可以将它的某一部分区域指定在需要显示壁纸的窗口的背景。

       在这个情景中，主要涉及到了WindowManagerService类的四个成员函数adjustWallpaperWindowsLocked、updateWallpaperOffsetLocked、performLayoutAndPlaceSurfacesLocked和assignLayersLocked，其中，成员函数performLayoutAndPlaceSurfacesLocked的实现框架可以参考前面Android窗口管理服务WindowManagerService计算Activity窗口大小的过程分析一文，成员函数assignLayersLocked的实现如上所述在接下来的一篇文章中再分析，本文主要是关注成员函数adjustWallpaperWindowsLocked和updateWallpaperOffsetLocked的实现。

       从上面的分析就可以知道，在布局一个窗口的过程中，可能需要调用WindowManagerService类的成员函数updateWallpaperOffsetLocked和adjustWallpaperWindowsLocked来调整壁纸窗口在X轴和Y轴上的偏移位置和在窗口堆栈中的位置。接下来我们就分别分析壁纸窗口在X轴和Y轴上的偏移位置和在窗口堆栈中的位置的调整过程。

       二. 调整壁纸窗口在X轴和Y轴上的偏移位置

       壁纸窗口的大小是可以大于屏幕大小的。在这种情况下，需要显示壁纸的Activity窗口就需要指定壁纸在X轴和Y轴上的偏移位置，以便可以将壁纸的某一部分作为窗口的背景。

       假设壁纸窗口的大小为（WallpaperWidth, WallpaperHeight），屏幕的大小为（DisplayWidth, DisplayHeight），并且壁纸在X轴和Y轴上的偏移位置为WallpaperX和WallpaperY，其中，WallpaperWidth > DisplayWidth，WallpaperHeight > DisplayHeight，0.0 <= WallpaperX <= 1.0，0.0 <= WallpaperY <= 1.0，如图3所示：

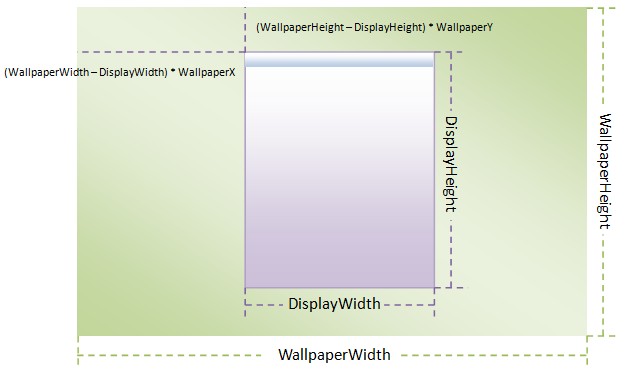


图3 指定壁纸窗口在X轴和Y轴上的偏移位置

        这时候壁纸窗口在X轴和Y轴上的偏移位置的绝对值XOffset和YOffset就分别等于（WallpaperWidth - DisplayWidth）\* WallpaperX和（WallpaperHeight - DisplayHeight）\* WallpaperY。这意味道着：

       1. 当WallpaperX = WallpaperY = 0.0时，取壁纸窗口的左上角区域作为窗口背景。

       2. 当WallpaperX = WallpaperY = 0.5时，取壁纸窗口的中间区域作为窗口背景。

       3. 当WallpaperX = WallpaperY = 1.0时，取壁纸窗口的右下角区域作为窗口背景。

       除了使用WallpaperX和WallpaperY来描述壁纸窗口在X轴和Y轴上的偏移位置之外，WindowManagerService服务还使用WallpaperXStep和WallpaperYStep来描述壁纸窗口跨越了多少个虚拟屏幕。例如，假设一个Activity窗口在X轴上有3个虚拟屏幕，即它的实际宽度是屏幕宽度的3倍，而在Y轴上有一个屏幕，即它的实际高度刚好等于屏幕高度，并且壁纸窗口的宽度也刚好是屏幕宽度的3倍，而高度也刚好是等于屏幕高度，那么WallpaperXStep和WallpaperYStep的值就可以分别指定为0.5和0，这意味着：

       1. 第1个虚拟屏幕取壁纸窗口的左边三分之一的区域作为窗口背景，相当于是将壁纸窗口在X轴和Y轴上的偏移位置WallpaperX和WallpaperY的值分别设置为0.0和0.0。

       2. 第2个虚拟屏幕取壁纸窗口的中间三分之一的区域作为窗口背景，相当于是将壁纸窗口在X轴和Y轴上的偏移位置WallpaperX和WallpaperY的值分别设置为0.5和0.0。

       3. 第3个虚拟屏幕取壁纸窗口的右边三分之一的区域作为窗口背景，相当于是将壁纸窗口在X轴和Y轴上的偏移位置WallpaperX和WallpaperY的值分别设置为1.0和0.0。

       一般地，如果一个Activity窗口在X轴上有N个虚拟屏幕，而在Y轴上有M个虚拟屏幕，那么它就会将壁纸窗口的WallpaperXStep和WallpaperYStep值分别设置为1.0 / (N - 1)和1.0 / (M - 1)。对于WindowManagerService服务来说，它并不关心壁纸窗口的WallpaperXStep和WallpaperYStep值，而只关心壁纸窗口的WallpaperX和WallpaperY值，因为通过后两者，它就可以知道怎么显示壁纸窗口了。壁纸窗口的WallpaperXStep和WallpaperYStep值是用来传递给提供壁纸的服务的。提供壁纸的服务一旦知道壁纸窗口的WallpaperXStep和WallpaperYStep值是多少，就可以知道当前需要显示避纸的窗口有多少个虚拟屏幕。

       上面提到的与壁纸窗口在X轴和Y轴上的偏移位置相关的六个状态WallpaperX、WallpaperY、WallpaperXStep、WallpaperYStep、XOffset和YOffset由WindowManagerService服务来统一维护，它们分别对应于WindowState类的六个成员变量mWallpaperX、mWallpaperY、mWallpaperXStep、mWallpaperYStep、mXOffset和mYOffset，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**float** mLastWallpaperX = -1;

**float** mLastWallpaperY = -1;

**float** mLastWallpaperXStep = -1;

**float** mLastWallpaperYStep = -1;

......

**private** **final** **class** **WindowState** **implements** **WindowManagerPolicy**.**WindowState** {

......

// If a window showing a wallpaper: the requested offset for the

// wallpaper; if a wallpaper window: the currently applied offset.

**float** mWallpaperX = -1;

**float** mWallpaperY = -1;

// If a window showing a wallpaper: what fraction of the offset

// range corresponds to a full virtual screen.

**float** mWallpaperXStep = -1;

**float** mWallpaperYStep = -1;

// Wallpaper windows: pixels offset based on above variables.

**int** mXOffset;

**int** mYOffset;

......

}

......

}

       这段代码定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       此外，WindowManagerService类还使用四个成员变量mLastWallpaperX、mLastWallpaperY、mLastWallpaperXStep和mLastWallpaperYStep来记录壁纸窗口上一次所使用的WallpaperX、WallpaperY、WallpaperXStep和WallpaperYStep值。

       在Android系统中，提供壁纸功能的组件叫做WallpaperService，它是一个Service组件，是由壁纸管理服务WallpaperManagerService负责启动的。WallpaperService有两个内部类BaseIWindow和Engine，其中，BaseIWindow是一个实现了IWindow接口的Binder本地对象类，用来和WindowManagerService服务通信，而Engine是一个真正用来实现壁纸功能的类。当一个Activity窗口需要指定壁纸窗口的某一部分区域作为它的背景时，它就会通过WallpaperManager类来通知WallpaperService设置壁纸窗口在X轴和Y轴上的偏移位置，这个过程如图4所示：

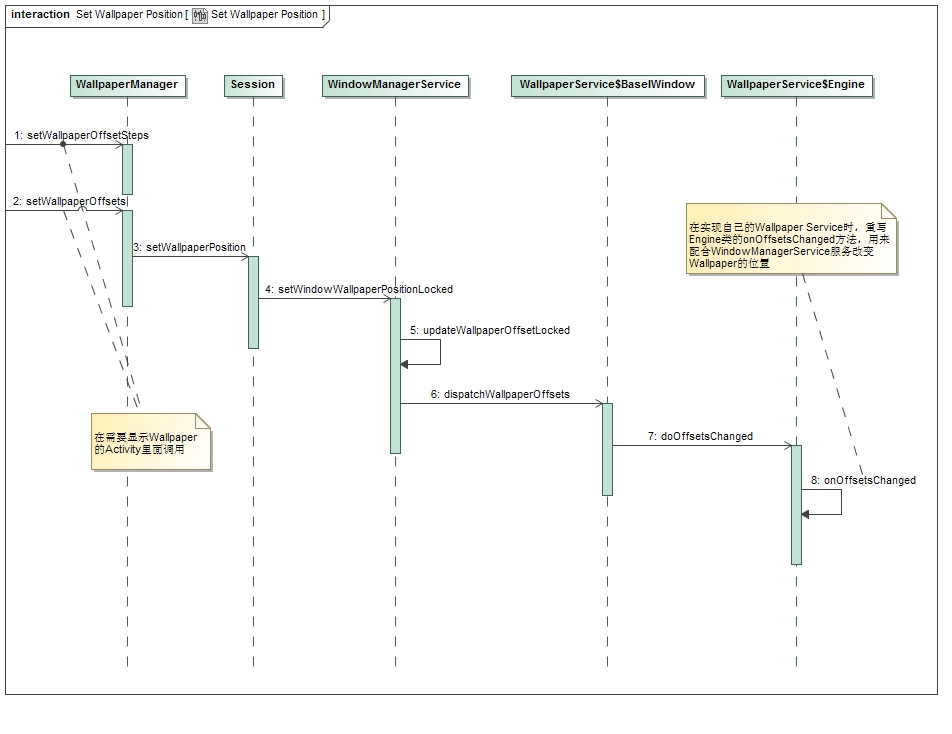


图4 壁纸窗口在X轴和Y轴上的偏移位置的设置过程

       这个过程大概如下所示：

       Step 1. 需要显示壁纸的Activity组件调用WallpaperManager类的成员函数setWallpaperOffsetSteps来设置壁纸窗口的WallpaperXStep和WallpaperYStep值。

       Step 2. 需要显示壁纸的Activity组件调用WallpaperManager类的成员函数setWallpaperOffsets来设置壁纸窗口的WallpaperX和WallpaperY值。

       Step 3. 一个类型为Session的Binder代理对象的成员函数setWallpaperPosition会被调用来通知WindowManagerService服务来重新计算壁纸窗口在X轴和Y轴上的偏移位置，传递的参数包括在Step 1和Step 2中所设置的WallpaperXStep、WallpaperYStep、WallpaperX和WallpaperY四个值。

       Step 4. WindowManagerService类的成员函数setWindowWallpaperPositionLocked会被调用来保存从前面Step 3传递过来的WallpaperXStep、WallpaperYStep、WallpaperX和WallpaperY值。

       Step 5. WindowManagerService类的成员函数updateWallpaperOffsetLocked会被调用来计算壁纸窗口在X轴和Y轴上的偏移位置的绝对值XOffset和YOffset，是根据壁纸窗口的大小（WallpapperWidth, WallpaperHeight）、屏幕的大小（DisplayWidth, DisplayHeight），以及保存在前面Step 4中的WallpaperX和WallpaperY来计算的。

       Step 6. 在WallpaperService类内部的一个BaseIWindow对象的成员函数dispatchWallpaperOffsets会被调用来通知WallpaperService服务，壁纸窗口在X轴和Y轴上的偏移位置发生改变了，传递过来的参数包括壁纸窗口的XOffset、YOffset、WallpaperXStep和WallpaperYStep值。

       Step 7. 在WallpaperService类内部的一个Engine对象的成员函数doOffsetsChanged会被调用来处理壁纸窗口在X轴和Y轴上的偏移位置变化事件。

       Step 8. Engine类的成员函数doOffsetsChanged会调用另外一个成员函数onOffsetsChanged来分发壁纸窗口在X轴和Y轴上的偏移位置变化事件。Engine类的成员函数onOffsetsChanged一般是由其子类来重写的，以便子类可以实现自己的壁纸效果。

       本文不打算详细这八个步骤，而主要关注Step 3、Step 4和Step 5这三步是如何计算壁纸窗口在X轴和Y轴上的偏移位置的，即主要关注Session类的成员函数setWallpaperPosition，以及WindowManagerService类的成员函数setWindowWallpaperPositionLocked和updateWallpaperOffsetLocked的实现。

        Session类的成员函数setWallpaperPosition的实现如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **final** **class** **Session** **extends** **IWindowSession**.**Stub**

**implements** **IBinder**.**DeathRecipient** {

......

**public** **void** **setWallpaperPosition**(IBinder window, **float** x, **float** y, **float** xStep, **float** yStep) {

**synchronized**(mWindowMap) {

**long** ident = Binder.clearCallingIdentity();

**try** {

setWindowWallpaperPositionLocked(

windowForClientLocked(**this**, window, **true**),

x, y, xStep, yStep);

} **finally** {

Binder.restoreCallingIdentity(ident);

}

}

}

......

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       Session类的成员函数setWallpaperPosition首先调用WindowManagerService类的成员函数windowForClientLocked来找到与参数window所对应的一个WindowState对象，这个WindowState对象描述的是要改变壁纸窗口位置的窗口，接着再调用WindowManagerService类的另外一个成员函数setWindowWallpaperPositionLocked来执行设置壁纸窗口在X轴和Y轴的偏移位置的操作。

       WindowManagerService类的成员函数setWindowWallpaperPositionLocked的实现如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**public** **void** **setWindowWallpaperPositionLocked**(WindowState window, **float** x, **float** y,

**float** xStep, **float** yStep) {

**if** (window.mWallpaperX != x || window.mWallpaperY != y) {

window.mWallpaperX = x;

window.mWallpaperY = y;

window.mWallpaperXStep = xStep;

window.mWallpaperYStep = yStep;

**if** (updateWallpaperOffsetLocked(window, **true**)) {

performLayoutAndPlaceSurfacesLocked();

}

}

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       WindowManagerService类的成员函数setWindowWallpaperPositionLocked首先检查参数window所描述的WindowState对象上一次所设置的壁纸窗口的偏移位置与参数x和y所描述的偏移位置是否不一样。如果不一样的话，那么就会分别将参数x、y、xStep和yStep分别保存在参数window所描述的WindowState对象的成员变量mWallpaperX、mWallpaperY、mWallpaperXStep和mWallpaperYStep中，并且调用WindowManagerService类的成员函数updateWallpaperOffsetLocked来更新系统中的壁纸窗口的偏移位置。

       如果WindowManagerService类的成员函数updateWallpaperOffsetLocked的返回值等于true，那么就说明它更新了系统中的壁纸窗口的偏移位置，因此，就需要调用WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked来刷新系统的UI。

       接下来我们继续分析WindowManagerService类的成员函数updateWallpaperOffsetLocked的实现，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**final** ArrayList<WindowToken> mWallpaperTokens = **new** ArrayList<WindowToken>();

// If non-null, this is the currently visible window that is associated

// with the wallpaper.

WindowState mWallpaperTarget = **null**;

......

**boolean** **updateWallpaperOffsetLocked**(WindowState changingTarget, **boolean** sync) {

**final** **int** dw = mDisplay.getWidth();

**final** **int** dh = mDisplay.getHeight();

**boolean** changed = **false**;

WindowState target = mWallpaperTarget;

**if** (target != **null**) {

**if** (target.mWallpaperX >= 0) {

mLastWallpaperX = target.mWallpaperX;

} **else** **if** (changingTarget.mWallpaperX >= 0) {

mLastWallpaperX = changingTarget.mWallpaperX;

}

**if** (target.mWallpaperY >= 0) {

mLastWallpaperY = target.mWallpaperY;

} **else** **if** (changingTarget.mWallpaperY >= 0) {

mLastWallpaperY = changingTarget.mWallpaperY;

}

}

**int** curTokenIndex = mWallpaperTokens.size();

**while** (curTokenIndex > 0) {

curTokenIndex--;

WindowToken token = mWallpaperTokens.get(curTokenIndex);

**int** curWallpaperIndex = token.windows.size();

**while** (curWallpaperIndex > 0) {

curWallpaperIndex--;

WindowState wallpaper = token.windows.get(curWallpaperIndex);

**if** (updateWallpaperOffsetLocked(wallpaper, dw, dh, sync)) {

wallpaper.computeShownFrameLocked();

changed = **true**;

// We only want to be synchronous with one wallpaper.

sync = **false**;

}

}

}

**return** changed;

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       当WindowManagerService类的成员变量mWallpaperTarget的值不等于null时，它所指向的一个WindowState对象描述的是系统当前可见的并且需要显示壁纸的窗口。在这种情况下，要将这个WindowState对象当前正在使用的壁纸窗口在X轴和Y轴上的偏移位置分别保存在WindowManagerService类的成员变量mLastWallpaperX和mLastWallpaperY，以便接下来可以用来计算壁纸窗口在X轴和Y轴上的偏移位置的绝对值。

       注意，如果WindowManagerService类的成员变量mWallpaperTarget所指向的一个WindowState对象的成员变量mWallpaperX（mWallpaperY）的值小于0，那么就说明这个WindowState对象所描述的窗口还没有设置过壁纸窗口在X轴上（Y轴上）的偏移位置，这时候就需要将参数changingTarget所指向的一个WindowState对象的成员变量mWallpaperX（mWallpaperY）的值保存在WindowManagerService类的成员变量mLastWallpaperX（mLastWallpaperY）中，前提也是它的值大于等于0，即它描述的是一个有效的偏移值。

       WindowManagerService类的成员变量mWallpaperTokens保存的是一系列与壁纸相关的窗口令牌，与这些窗口令牌所对应的窗口就是系统当前所设置的壁纸窗口。WindowManagerService类的成员函数updateWallpaperOffsetLocked依次调用另外一个四参数版本的成员函数updateWallpaperOffsetLocked来更新系统当前所设置的每一个壁纸窗口在X轴和Y轴上的偏移位置。

       注意， WindowManagerService类的四个参数版本的成员函数updateWallpaperOffsetLocked的最后一个参数sync是一个布尔值，用来表示在更新壁纸窗口在X轴和Y轴上的偏移位置的时候，是否需要同步等待提供壁纸窗口的服务处理完成壁纸窗口在X轴和Y轴上的偏移位置变化事件。参数sync本身也是由两个参数版本的成员函数updateWallpaperOffsetLocked的调用者传进来的，它的值即使等于true，两个参数版本的成员函数updateWallpaperOffsetLocked也只会同步等待提供第一个壁纸窗口的服务处理完成壁纸窗口在X轴和Y轴上的偏移位置变化事件。

       WindowManagerService类的四个参数版本的成员函数updateWallpaperOffsetLocked的实现如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**boolean** **updateWallpaperOffsetLocked**(WindowState wallpaperWin, **int** dw, **int** dh,

**boolean** sync) {

**boolean** changed = **false**;

**boolean** rawChanged = **false**;

**float** wpx = mLastWallpaperX >= 0 ? mLastWallpaperX : 0.5f;

**float** wpxs = mLastWallpaperXStep >= 0 ? mLastWallpaperXStep : -1.0f;

**int** availw = wallpaperWin.mFrame.right-wallpaperWin.mFrame.left-dw;

**int** offset = availw > 0 ? -(**int**)(availw\*wpx+.5f) : 0;

changed = wallpaperWin.mXOffset != offset;

**if** (changed) {

......

wallpaperWin.mXOffset = offset;

}

**if** (wallpaperWin.mWallpaperX != wpx || wallpaperWin.mWallpaperXStep != wpxs) {

wallpaperWin.mWallpaperX = wpx;

wallpaperWin.mWallpaperXStep = wpxs;

rawChanged = **true**;

}

**float** wpy = mLastWallpaperY >= 0 ? mLastWallpaperY : 0.5f;

**float** wpys = mLastWallpaperYStep >= 0 ? mLastWallpaperYStep : -1.0f;

**int** availh = wallpaperWin.mFrame.bottom-wallpaperWin.mFrame.top-dh;

offset = availh > 0 ? -(**int**)(availh\*wpy+.5f) : 0;

**if** (wallpaperWin.mYOffset != offset) {

......

changed = **true**;

wallpaperWin.mYOffset = offset;

}

**if** (wallpaperWin.mWallpaperY != wpy || wallpaperWin.mWallpaperYStep != wpys) {

wallpaperWin.mWallpaperY = wpy;

wallpaperWin.mWallpaperYStep = wpys;

rawChanged = **true**;

}

**if** (rawChanged) {

**try** {

......

**if** (sync) {

mWaitingOnWallpaper = wallpaperWin;

}

wallpaperWin.mClient.dispatchWallpaperOffsets(

wallpaperWin.mWallpaperX, wallpaperWin.mWallpaperY,

wallpaperWin.mWallpaperXStep, wallpaperWin.mWallpaperYStep, sync);

**if** (sync) {

**if** (mWaitingOnWallpaper != **null**) {

**long** start = SystemClock.uptimeMillis();

**if** ((mLastWallpaperTimeoutTime+WALLPAPER\_TIMEOUT\_RECOVERY)

< start) {

**try** {

......

mWindowMap.wait(WALLPAPER\_TIMEOUT);

} **catch** (InterruptedException e) {

}

......

**if** ((start+WALLPAPER\_TIMEOUT)

< SystemClock.uptimeMillis()) {

......

mLastWallpaperTimeoutTime = start;

}

}

mWaitingOnWallpaper = **null**;

}

}

} **catch** (RemoteException e) {

}

}

**return** changed;

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       WindowManagerService类的四个参数版本的成员函数updateWallpaperOffsetLocked首先计算参数wallpaper所描述的壁纸窗口在X轴和Y轴上的偏移位置，接着再向提供该壁纸窗口的服务发向一个壁纸窗口在X轴和Y轴上的偏移位置变化事件通知。

       参数wallpaper所描述的壁纸窗口在X轴和Y轴上的偏移位置的计算过程是一样的，这里我们只要结合前面的图3来分析壁纸窗口在X轴上的偏移位置的计算过程。

       在图3中，壁纸窗口的WallpaperX、WallpaperXStep、WallpaperWidth和DisplayWidth值分别等于这里的mLastWallpaperX、mLastWallpaperXStep、wallpaperWin.mFrame.right - wallpaperWin.mFrame.left和dw。有了这些值之后，就可以计算得到参数wallpaper所描述的壁纸窗口在X轴上的偏移位置的绝对值XOffset了。

       如果计算得到的XOffset、WallpaperX、WallpaperXStep的值与原来保存在参数wallpaper所指向的一个WindowState对象的成员变量mXOffset、mWallpaperX、mWallpaperXStep的值不相等，那么就会将计算得到的XOffset、WallpaperX、WallpaperXStep的值分别保存在这个WindowState对象的成员变量mXOffset、mWallpaperX、mWallpaperXStep，并且相应地将变量changed和rawChanged的值设置为true，表示参数wallpaper所描述的壁纸窗口在X轴上的偏移位置发生了变化。

       有四个地方需要注意：

       1. 当mLastWallpaperX的值小于0的时候，那么就说明系统中的壁纸窗口还没有被设置一个有效的X轴偏移位置，这时候计算壁纸窗口在X轴上的偏移位置所采用的WallpaperX值就会取为0.5，即默认将壁纸窗口的中间区域指定为需要显示壁纸的窗口的背景。

       2. 当mLastWallpaperXStep的值小于0的时候，那么就说明需要显示壁纸的窗口还没有告诉WindowManagerService服务它有多少个虚拟屏幕，这时候就会将壁纸窗口的WallpaperXStep值设置为-1.0，用来告诉提供壁纸窗口的服务，需要显示壁纸的窗口没有指定虚拟屏幕的个数。

       3. 当壁纸窗口的宽度小于等于屏幕宽度的时候，即变量availw的值小于等于0的时候，那么就说明不需要设置壁纸窗口在X轴上的偏移位置，也就是说，这时候壁纸窗口在X轴上的偏移位置始终保持为0。

       4. 当壁纸窗口的宽度大于屏幕宽度的时候，即变量availw的值大于0的时候，壁纸窗口在X轴上的偏移值等于availw \* wps，加上0.5是为了向上取整，向上取整后需要取反，因为负数才能正确表达出壁纸窗口相对屏幕的偏移。

       计算完成参数wallpaper所描述的壁纸窗口在X轴和Y轴上的偏移位置之后，如果变量rawChanged的值等于true，那么就说明参数wallpaper所描述的壁纸窗口在X轴和Y轴上的偏移位置发生了变化，这时候就需要向提供该壁纸窗口的服务发送一个事件通知，这是通过调用参数wallpaperWin所指向的一个WindowState对象的成员变量mClient所描述的一个实现了IWindow接口的Binder代理对象的成员函数dispatchWallpaperOffsets来实现的，同时传递给壁纸窗口的服务的参数有壁纸窗口当前所使用的WallpaperX、WallpaperY、WallpaperXStep和WallpaperYStep值，以及另外一个同步参数sync。

       当参数sync的值等于true的时候，就表示WindowManagerService服务需要等待提供壁纸窗口wallpaperWin的服务处理完成前面所发送的偏移位置变化事件通知，等待的最长时间为WALLPAPER\_TIMEOUT。如果提供壁纸窗口wallpaperWin的服务不能在WALLPAPER\_TIMEOUT时间内向WindowManagerService服务发送一个事件处理完成通知，那么WindowManagerService服务就会将这次事件通知发送时间start保存在WindowManagerService类的成员变量mLastWallpaperTimeoutTime中。

       如果上一次发送的壁纸窗口偏移位置变化事件通知发生了超时，那么在上次发送这个事件通知起的WALLPAPER\_TIMEOUT\_RECOVERY时间内，是不允许再次发送壁纸窗口偏移位置变化事件通知的。这是因为在上一次事件通知超时的情况下，在短时间内再次发送相同的事件通知也是非常有可能是超时的，因此，就不允许短时间内重复发送相同的事件通知，避免出现雪崩现象。

       关于互联网的雪崩现象，可以举一个常见的例子来说明。假设现在有一个Web页面正在现场直播一项非常热门的体育赛事，这时候就会有海量的用户访问这个页面。一旦访问量快要达到Web服务器的承受能力的时候，Web页面的打开速度就会越来越慢。Web页面打开速度变慢的时候，用户就会下意识地不断按F5刷新。越是不断地按F5刷新，Web页面的请求量就越大，而当请求量大于Web服务器的承受能力的时候，Web服务器就会宕机了，这个就是雪崩现象。为了避免雪崩现象，就需要在请求量快要达到Web服务器的承受能力的时候，避免用户发送更多的访问请求，以使得Web服务器有喘息的机会。

       废话少说了，当WindowManagerService服务在等待壁纸窗口wallpaper所属的服务处理它的偏移位置变化事件通知时，会将该壁纸窗口wallpaper保存在WindowManagerService类的成员变量mWaitingOnWallpaper中，用来表示WindowManagerService服务正在处于等待壁纸服务处理完成一个壁纸窗口偏移位置变化事件通知。一旦壁纸服务处理完成该事件通知，WindowManagerService类的成员变量mWaitingOnWallpaper的值就会被设置为null。

       壁纸服务处理壁纸窗口在X轴和Y轴上的偏移位置变化事件通知的过程就如图4的Step 6至Step 8所示。

       至此，我们就分析完成壁纸窗口在X轴和Y轴上的偏移位置的调整过程了，接下来我们就继续分析壁纸窗口在窗口堆栈中的位置调整过程。

       三. 调整壁纸窗口在窗口堆栈中的位置

       调整壁纸窗口在窗口堆栈中的位置实际上就是将壁纸窗口放置在需要显示壁纸的窗口的下面，这是是通过调用WindowManagerService类的成员函数adjustWallpaperWindowsLocked来实现的。

       WindowManagerService类的成员函数adjustWallpaperWindowsLocked的实现框架如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**int** **adjustWallpaperWindowsLocked**() {

**int** changed = 0;

**final** **int** dw = mDisplay.getWidth();

**final** **int** dh = mDisplay.getHeight();

// First find top-most window that has asked to be on top of the

// wallpaper; all wallpapers go behind it.

**final** ArrayList<WindowState> localmWindows = mWindows;

**int** N = localmWindows.size();

WindowState w = **null**;

WindowState foundW = **null**;

**int** foundI = 0;

WindowState topCurW = **null**;

**int** topCurI = 0;

**int** i = N;

//Label #1:

**while** (i > 0) {

//从上到下遍历窗口堆栈，查找需要显示壁纸的窗口foundW，foundI为窗口foundW在窗口堆栈中

//的位置如果没有找到需要显示壁纸的窗口，并且系统中存在壁纸窗口，那么topCurW就指向Z轴

//位置最大的壁纸窗口，topCurI为窗口topCurW在窗口堆栈中的位置，这时候foundW一定等于

//null。

......

}

//Label #2:

**if** (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {

//如果系统当前正在窗口切换的过程中，并且系统当前存在一个需要显示壁纸的Activity窗口，

//那么就认为当前正在执行的窗口切换涉及到了这个需要显示壁纸的Activity窗口，

//因此，就暂时不要调整壁纸窗口的位置了，等到窗口切换过程完成了再说。

//系统当前存在一个需要显示壁纸的Activity窗口，意味着mWallpaperTarget不等于null，

//或者foundW不等于null。

......

}

**if** (mWallpaperTarget != foundW) {

//上一次显示壁纸的窗口和接下来要显示壁纸的窗口发生了变化

mLowerWallpaperTarget = **null**;

mUpperWallpaperTarget = **null**;

WindowState oldW = mWallpaperTarget;

mWallpaperTarget = foundW;

// Now what is happening... if the current and new targets are

// animating, then we are in our super special mode!

**if** (foundW != **null** && oldW != **null**) {

**boolean** oldAnim = oldW.mAnimation != **null**

|| (oldW.mAppToken != **null** && oldW.mAppToken.animation != **null**);

**boolean** foundAnim = foundW.mAnimation != **null**

|| (foundW.mAppToken != **null** && foundW.mAppToken.animation != **null**);

......

//Label #3:

**if** (foundAnim && oldAnim) {

//上一次显示壁纸的窗口oldW和接下来要显示壁纸的窗口foundW正在显示动画的

//过程中，那么就将Z轴位置较高的窗口保存在mUpperWallpaperTarget中，而将

//Z轴位置较低的窗口保存在mLowerWallpaperTarget中，并且将变量foundW指向

//Z轴位置较高的窗口，这样就可以在这两个窗口的动画显示过程中都能看到壁

//纸窗口.

......

}

}

｝**else** **if** (mLowerWallpaperTarget != **null**) {

//Label #4:

//检查mUpperWallpaperTarget和mLowerWallpaperTarget所指向的窗口的动画显示过程

//是否已经结束，如果已经结束，那么就将mUpperWallpaperTarget和

//mLowerWallpaperTarget的值置null。

// Is it time to stop animating?

......

}

**boolean** visible = foundW != **null**;

//Label #5:

**if** (visible) {

//前面找到了一个需要显示壁纸的窗口foundW，并且存在其它窗口与它关联，这些关联的

//窗口包括：

//1. 在与该窗口所对应的窗口令牌的其它窗口

//2. 该窗口所设置的启动窗口

//3. 附加在该窗口的其它窗口

//在上述这些关联的窗口中，如果存在一些Z轴位置比窗口foundW小，那么就将需要将壁纸

//窗口放在Z轴位置最小的那个窗口下面，即将变量foundW指向Z轴位置最小的那个窗口。

......

}

//让变量foundW指向前面找到的需要显示壁纸的窗口的下一个窗口，

//这时候变量foundI记录的仍是需要显示壁纸的窗口在窗口堆栈中的位置，

//接下来会根据这两个变量来调整壁纸窗口在窗口堆栈中的位置

**if** (foundW == **null** && topCurW != **null**) {

//前面提到，如果没有找到需要显示壁纸的窗口，并且系统中存在壁纸窗口，那么foundW一

//定等于null，并且topCurW一定不等于null，这时候就不需要调整壁纸窗口在窗口堆栈中的

//位置。为了与其它情况统一处理，这时候假设位于壁纸窗口上面的那个窗口就是需要显示

//壁纸的窗口。因此，就会将foundI的值设置为(topCurI+1)，而将foundW的值设置为

//topCurW。

// There is no wallpaper target, so it goes at the bottom.

// We will assume it is the same place as last time, if known.

foundW = topCurW;

foundI = topCurI+1;

} **else** {

//前面找到了需要显示壁纸的窗口，因此，就将它的下一个窗口保存在foundW中，变量foundI

//的值不需要修改。

// Okay i is the position immediately above the wallpaper. Look at

// what is below it for later.

foundW = foundI > 0 ? localmWindows.get(foundI-1) : **null**;

}

//如果前面找到的需要显示壁纸的窗口是可见的，并且当前正在显示壁纸的窗口设置了壁纸窗口

//在X轴和Y轴上的偏移位置，那么就将用来描述壁纸窗口在X轴和Y轴上的偏移位置的WallpaperX、

//WallpaperY、WallpaperXStep和WallpaperYStep值记录在mLastWallpaperX、

//mLastWallpaperXStep、mLastWallpaperY和mLastWallpaperYStep中。

**if** (visible) {

**if** (mWallpaperTarget.mWallpaperX >= 0) {

mLastWallpaperX = mWallpaperTarget.mWallpaperX;

mLastWallpaperXStep = mWallpaperTarget.mWallpaperXStep;

}

**if** (mWallpaperTarget.mWallpaperY >= 0) {

mLastWallpaperY = mWallpaperTarget.mWallpaperY;

mLastWallpaperYStep = mWallpaperTarget.mWallpaperYStep;

}

}

//Label #6:

// Start stepping backwards from here, ensuring that our wallpaper windows

// are correctly placed.

**int** curTokenIndex = mWallpaperTokens.size();

**while** (curTokenIndex > 0) {

//一切准备就绪，开始调整系统中的壁纸窗口在窗口堆栈的位置，算法如下所示。

//对于从Z轴位置从高到低的每一个壁纸窗口wallpaper：

//1. 如果它与变量foundW指向的不是同一个壁纸窗口，那么就说明它在窗口堆栈中

//的位置不对，这时候就需要将它调整到窗口堆栈中的第foundI个位置上。

//2. 如果它与变量foundW指向的是同一个壁纸窗口，那么就说明它在窗口堆栈中的

//位置是正确，这时候就不需要对它进行调整，不过要让变量foundI的值减1，并且将

//在窗口堆栈第(foundI - 1)个位置的窗口记录在变量foundW中。

//注意，变量foundW一开始就指向Z轴位置最高的壁纸窗口，而变量foundI记录的是

//位于Z轴位置最高的壁纸窗口上面的那个窗口在窗口堆栈中的位置。

//上述算法实际上是用状态机的方法将系统中的所有壁纸窗口（假设数量为N）按照Z轴

//位置从高到底的顺序放置在窗口堆栈中的第（foundI - 1）、（foundI - 2）、

//（foundI - 3）、......、（foundI - N）个位置上。

......

}

**return** changed;

}

......

}

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       WindowManagerService类的成员函数adjustWallpaperWindowsLocked是按照以下流程来调整壁纸窗口在窗口堆栈中的位置的：

       1. 通过一个while循环来从上到下地遍历窗口堆栈，找到需要显示壁纸的窗口foundW，其中，foundI为窗口foundW在窗口堆栈中的位置。如果没有找到需要显示壁纸的窗口，并且系统中存在壁纸窗口，那么topCurW就指向Z轴位置最大的壁纸窗口，其中，topCurI为窗口topCurW在窗口堆栈中的位置。在这种情况下，变量foundW的值一定等于null的。

       2. 如果WindowManagerService类的成员变量mNextAppTransition的值不等于WindowManagerPolicy.TRANSIT\_UNSET，那么就说明系统当前正在窗口切换的过程中。在这种情况下，如果系统当前存在一个需要显示壁纸的Activity窗口，即WindowManagerService类的成员变量mWallpaperTarget的值不等于null，或者前面得到的变量foundW的值不等于null，那么就认为当前正在执行的窗口切换操作涉及到了这个需要显示壁纸的Activity窗口。这时候就不需要调整壁纸窗口的位置，要等到窗口切换过程完成了之后再调整。

       3. 如果WindowManagerService类的成员变量mWallpaperTarget和前面得到的变量foundW指向的不是同一个WindowState对象，那么就说明上一次显示壁纸的窗口和接下来要显示壁纸的窗口发生了变化。在这种情况下，就会使用变量oldW来描述上一次显示壁纸的窗口，而接下来要显示壁纸的窗口通过WindowManagerService类的成员变量mWallpaperTarget以及变量foundW来描述。这时候如果检查发现上一次显示壁纸的窗口和接下来要显示壁纸的窗口都处于显示动画的过程中，那么就会将Z轴位置较高的窗口保存在WindowManagerService类的成员变量mUpperWallpaperTarget中，而将Z轴位置较低的窗口保存在WindowManagerService类的成员变量mLowerWallpaperTarget中，并且将变量foundW指向Z轴位置较高的窗口。这样就能保证在这两个窗口的动画显示过程中都能看到壁纸窗口，实际上就是保证在两个窗口的切换过程中看到壁纸窗口。

       4. 如果WindowManagerService类的成员变量mWallpaperTarget和前面得到的变量foundW指向的是同一个WindowState对象，并且WindowManagerService类的成员变量mLowerWallpaperTarget的值不等于null，那么就说明需要检查系统的窗口切换过程完成了没有。如果已经完成，那么就需要将WindowManagerService类的成员变量mUpperWallpaperTarget和mLowerWallpaperTarget的值设置为null。由此可以，WindowManagerService类的成员变量mUpperWallpaperTarget和mLowerWallpaperTarget的作用就是用来记录两个处于切换状态的需要显示壁纸的窗口。

       5. 如果变量foundW的值不等于null，那么就说明前面找到了一个接下来要显示壁纸的窗口。在这种情况下，需要做两件事情。第一件事情是判断接下来要显示壁纸的窗口是否是可见的。如果是的话，那么就会将变量visible的值设置为true。第二件事情是在与接下来要显示壁纸的窗口相关联的窗口中，即与变量foundW所描述的窗口相关联的窗口中，找到一个Z轴位置最小的窗口，因为壁纸窗口最终是要放置在这个Z轴位置最小的窗口的下面，而不是最初找到的那个窗口的下面。与变量foundW所描述的窗口相关联的窗口包括：A.  与变量foundW所描述的窗口具有相同窗口令牌的其它窗口；B. 与变量foundW所描述的窗口附加在同一个窗口的其它窗口；C. 为变量foundW所描述的窗口所设置的启动窗口；D. 附加在变量foundW所描述的窗口上的其它窗口。一旦找到这样的一个窗口，那么就会让重新让变量foundW指向它。

       6. 再次重新调整变量foundW的值，让它指向位于前面所找到的需要显示壁纸的窗口的下面的一个窗口。注意，这个窗口有可能就是壁纸窗口。这时候变量foundI记录的然是前面所找到的需要显示壁纸的窗口在窗口堆栈中的位置。这样做的目的是为了接下来可以方便地调整壁纸窗口在窗口堆栈中的位置。但是如果变量foundW的值等于null，那么就说明前面根本没有找到需要显示壁纸的窗口。在这种情况下，如果变量topCurW的值不等于null，那么就说明系统中存在壁纸窗口。这种情况其实就不需要调整壁纸窗口在窗口堆栈中的位置了，但是为了接下来的逻辑可以统一处理，就假定位于壁纸窗口上面的那个窗口是需要显示壁纸的窗口。因此，就会将变量foundI的值设置为(topCurI+1)，而将变量foundW的值设置为topCurW。

       7. 如果前面所找到的需要显示壁纸的窗口是可见的，即变量visible的值等于true，并且当前正在显示壁纸的窗口设置了壁纸窗口在X轴和Y轴上的有效偏移位置，即WindowManagerService类的成员变量mWallpaperTarget所指向的一个WindowState对象的成员变量mWallpaperX和mWallpaperY的值大于等于0，那么就将用来描述壁纸窗口在X轴和Y轴上的偏移位置的WallpaperX、WallpaperY、WallpaperXStep和WallpaperYStep值记录在WindowManagerService类的成员变量mLastWallpaperX、mLastWallpaperY、mLastWallpaperXStep和mLastWallpaperYStep中。

       8. 经过上面的一系列操作之后，现在一切准备就绪，因此就可以按照以下的算法来调整系统中的壁纸窗口在窗口堆栈的位置。对于从Z轴位置从高到低的每一个壁纸窗口wallpaper：(1). 如果它与变量foundW指向的不是同一个壁纸窗口，那么就说明它在窗口堆栈中的位置不对，这时候就需要将它调整到窗口堆栈中的第foundI个位置上；(2). 如果它与变量foundW指向的是同一个壁纸窗口，那么就说明它在窗口堆栈中的位置是正确，这时候就不需要对它进行调整，不过要让变量foundI的值减1，并且将在窗口堆栈第(foundI - 1)个位置的窗口记录在变量foundW中；(3). 重复执行第(1)和第(2)步的操作，直到系统所有的壁纸窗口都检查完成为止。注意，在上述算法中，变量foundW一开始就指向Z轴位置最高的壁纸窗口，而变量foundI记录的是位于Z轴位置最高的壁纸窗口上面的那个窗口在窗口堆栈中的位置。每当Z轴位置最高的壁纸窗口在窗口堆栈中的位置调整完成之后，变量foundW就会指向Z轴位置次高的壁纸窗口，而变量foundI的值也会相应的地减少1。这个算法其实就是用状态机的方法来将系统中的所有壁纸窗口（假设数量为N）按照Z轴位置从高到底的顺序放置在窗口堆栈中的第（foundI - 1）、（foundI - 2）、（foundI - 3）、......、（foundI - N）个位置上。

       上述流程可能还是比较抽象，接下来我们就通过在标号为Label #1、Label #2、Label #3、Label #4、Label #5和Label #6处所忽略的代码来详细分析壁纸窗口在窗口堆栈中的位置的调整过程。

       标号为Label #1的代码如下所示：

**while** (i > 0) {

i--;

w = localmWindows.**get**(i);

**if** ((w.mAttrs.type == WindowManager.LayoutParams.TYPE\_WALLPAPER)) {

**if** (topCurW == **null**) {

topCurW = w;

topCurI = i;

}

**continue**;

}

topCurW = **null**;

**if** (w.mAppToken != **null**) {

// If this window's app token is hidden and not animating,

// it is of no interest to us.

**if** (w.mAppToken.hidden && w.mAppToken.animation == **null**) {

......

topCurW = **null**;

**continue**;

}

}

......

**if** ((w.mAttrs.flags&FLAG\_SHOW\_WALLPAPER) != 0 && w.isReadyForDisplay()

&& (mWallpaperTarget == w

|| (!w.mDrawPending && !w.mCommitDrawPending))) {

......

foundW = w;

foundI = i;

**if** (w == mWallpaperTarget && ((w.mAppToken != **null**

&& w.mAppToken.animation != **null**)

|| w.mAnimation != **null**)) {

// The current wallpaper target is animating, so we'll

// look behind it for another possible target and figure

// out what is going on below.

......

**continue**;

}

**break**;

}

}

       这段代码从上到下遍历保存在窗口堆栈中的窗口，目的是要找到一个Z轴位置最大的并且需要显示壁纸的窗口。一个窗口如果需要显示壁纸，那么用来描述它的一个WindowState对象w的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量flags的值的FLAG\_SHOW\_WALLPAPER位就不等于0。

       一个需要显示壁纸的窗口只有准备就绪显示并且UI也已经绘制完成之后，WindowManagerService服务才会将壁纸窗口放置在它的下面。 一个需要显示壁纸的窗口如果已经准备就绪显示，那么用来描述它的一个WindowState对象w的成员函数isReadyForDisplay的返回值等于true。另一方面，如果一个窗口的UI还没有绘制，那么用来描述它的一个WindowState对象w的成员变量mDrawPending的值就会等于true。一个窗口的UI虽然绘制好了，但是还没有提交给SurfaceFlinger服务处理，即用来描述它的一个WindowState对象w的成员变量mCommitDrawPending的值等于true，那么它的UI也是认为还没有绘制完成的。

       在遍历的过程中，如果发现一个窗口w刚好就是当前正在显示壁纸的窗口mWallpaperTarget，那么就会继续检查该窗口是否正处于显示动画的过程中。如果是的话，那么就需要跳过该窗口，因为我们的目标是要找到另外一个接下来要显示壁纸的窗口。对于Activity窗口和非Activity窗口来说，判断它们是否是正处于显示动画的过程中的方法是不一样的。对于一个处于显示动画过程的Activity窗口来说，用来描述它的一个WindowState对象w的成员变量mAppToken的值不等于null，并且指向了一个AppWindowToken对象，并且这个AppWindowToken对象的成员变量animation的值不等于null。对于一个处于显示动画过程的非Activity窗口来说，用来描述它的一个WindowState对象w的成员变量mAnimation的值不等于null。这就是说，AppWindowToken类的成员变量animation和WindowState类的成员变量mAnimation都是用来描述一个动画对象的。

      在遍历的过程中，有两种类型的窗口是需要跳过的。第一种类型的窗口是壁纸窗口，即用来描述它的一个WindowState对象w的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量type的值等于WindowManager.LayoutParams.TYPE\_WALLPAPER。第二种类型的窗口是Activity窗口，但是与之所对应的Activity组件处于不可见状态，这意味着这种类型的窗口也是不可见的。前面提到，对于Activity窗口来说，用来描述它的一个WindowState对象w的成员变量mAppToken的值是不等于null的，并且指向了一个AppWindowToken对象。当这个AppWindowToken对象的成员变量hidden的值等于true的时候，就意味着对应的Activity组件是不可见的。有时候一个AppWindowToken对象的成员变量hidden的值虽然等于true，但是如果这个AppWindowToken对象的成员变量animation的值不等于null，那么隐含着对应的Activity组件其实还是可见的，因为它还处于显示动画的过程中。

       遍历完成之后，有可能找到了接下来要显示壁纸的窗口，也有可能找不到接下来要显示壁纸的窗口。

       如果找到了接下来要显示壁纸的窗口，那么变量foundW的值就不等于null，并且指向了这个接下来要显示壁纸的窗口，另外一个变量foundI记录的是该窗口在窗口堆栈中位置。这时候变量topCurW的值一定等于null，但是变量topCurI的值却不一定等于0，它有可能指向了Z轴位置最大的那个壁纸窗口。

       假设foundW的值不等于null，并且变量topCurI的值等于0.，那么窗口堆栈的状态就如图5所示：

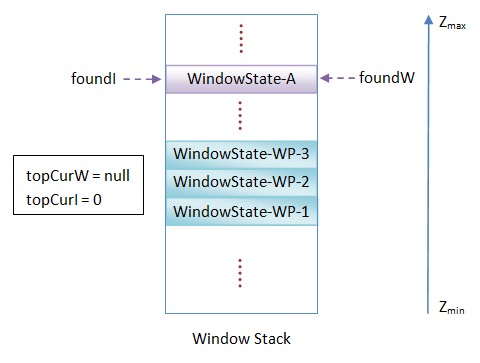


图5 foundw != null & topCurI == 0

       假设foundW的值不等于null，并且变量topCurI的值大于0.，那么窗口堆栈的状态就如图6所示：

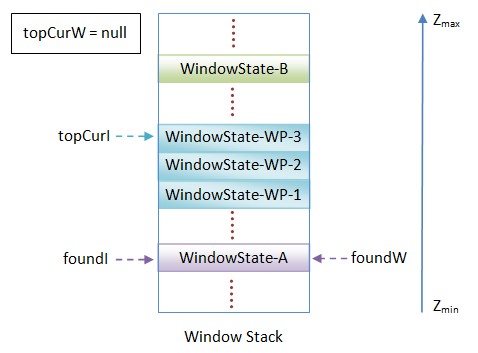


图6 foundW != null & topCurI != 0

       如果没有找到接下来要显示壁纸的窗口，那么变量foundW的值就等于null，并且另外一个变量foundI的值等于0。这时候变量topCurW的值始终等于null，而变量topCurI的值可能不等于0，取决于系统中是否存在壁纸窗口。

       为了方便描述，我们假设系统中是存在壁纸窗口，那么这时候topCurI的值就不等于0，并且它记录的是Z轴位置最大的那个壁纸窗口在窗口堆栈中的位置，如图7所示：

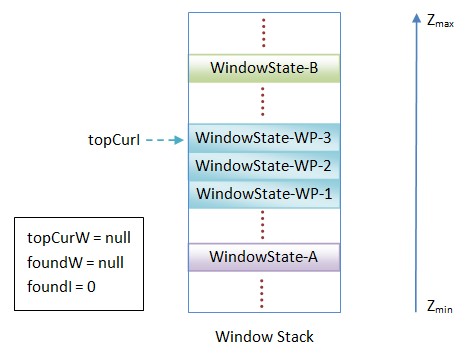


图7 foundW == null && topCurI != 0

       标号为Label #2的代码如下所示：

**if** (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {

// If we are currently waiting for an app transition, and either

// the current target or the next target are involved with it,

// then hold off on doing anything with the wallpaper.

// Note that we are checking here for just whether the target

// is part of an app token... which is potentially overly aggressive

// (the app token may not be involved in the transition), but good

// enough (we'll just wait until whatever transition is pending

// executes).

**if** (mWallpaperTarget != **null** && mWallpaperTarget.mAppToken != **null**) {

......

**return** 0;

}

**if** (foundW != **null** && foundW.mAppToken != **null**) {

......

**return** 0;

}

}

       WindowManagerService类的成员变量mNextAppTransition的值不等于WindowManagerPolicy.TRANSIT\_UNSET意味着系统当前正在窗口切换的过程中。这里说的窗口切换其实就是由Activity组件切换引起来的，即切换的是Activity窗口。如果正在切换的Activity窗口是是需要显示壁纸的，那么WindowManagerService类的成员函数adjustWallpaperWindowsLocked就要等到切换过程结束后，才能调整重新调整壁纸窗口在窗口堆栈中的位置。

       这里本来是要判断正在发生切换的Activity窗口是否是当前壁纸窗口的目标窗口或者前面所找到的接下来要显示壁纸的窗口的，但是却没有这样做。这段代码采取了一种比较激进的方法，即主要发现当前壁纸窗口的目标窗口是一个Activity窗口，或者前面所找到的接下来要显示壁纸的窗口是一个Activity窗口，那么就认为当前正在执行的窗口切换过程涉及到了壁纸窗口，因此，就要等到切换过程结束后，再来重新调整壁纸窗口在窗口堆栈中的位置。

       WindowManagerService类的成员变量mWallpaperTarget描述的就是当前壁纸窗口的目标窗口，当它的值不等于null时，并且它所指向的一个WindowState对象的成员变量mAppToken的值不等于null，那么就说明当前壁纸窗口的目标窗口是一个Activity窗口。同样，如果前面得到的变量foundW的值不等于null，并且它所指向的一个WindowState对象的成员变量mAppToken的值不等于null，那么就说明前面所找到的接下来要显示壁纸的窗口是一个Activity窗口。

       标号为Label #3的代码如下所示：

**if** (foundAnim && oldAnim) {

**int** oldI = localmWindows.indexOf(oldW);

......

**if** (oldI >= 0) {

......

// Set the new target correctly.

**if** (foundW.mAppToken != **null** && foundW.mAppToken.hiddenRequested) {

......

mWallpaperTarget = oldW;

}

// Now set the upper and lower wallpaper targets

// correctly, and make sure that we are positioning

// the wallpaper below the lower.

**if** (foundI > oldI) {

// The new target is on top of the old one.

......

mUpperWallpaperTarget = foundW;

mLowerWallpaperTarget = oldW;

foundW = oldW;

foundI = oldI;

} **else** {

// The new target is below the old one.

......

mUpperWallpaperTarget = oldW;

mLowerWallpaperTarget = foundW;

}

}

}

       当变量foundAnim和oldAnim的值均等于true的时候，就说明当前正在显示壁纸的窗口oldW和接下来要显示壁纸的窗口foundW均处于显示动画的过程中，那么就分别将它们记录在WindowManagerService类的成员变量mLowerWallpaperTarget和mUpperWallpaperTarget中，其中，前者用来描述Z轴位置较低的窗口，而后者用来描述Z轴位置较高的的窗口。

        变量foundI和oldI记录的分别是窗口foundW和oldW在窗口堆栈中的位置。因此，当变量foundI的值大于变量oldI的值的时候，窗口foundW就是Z轴位置较高的的窗口，而窗口oldW就是Z轴位置较低的的窗口。相反，当变量foundI的值小于等于变量oldI的值的时候，窗口oldW就是Z轴位置较高的的窗口，而窗口foundW就是Z轴位置较低的的窗口。

        这里有三个地方是需要注意的：

        1. 当前正在显示壁纸的窗口oldW其实就是WindowManagerService类的成员变量mWallpaperTarget所描述的那个窗口。

        2. 变量foundW和foundI记录的始终都是Z轴位置较低的那个窗口及其在窗口堆栈的位置，因此，当变量foundI的值大于变量oldI的值的时候，要将变量foundW和foundI的值分别设置为oldW和oldI，这样做的目的是为了接下来可以将壁纸窗口放置在Z轴位置较低的窗口的下面，以便可以在两个窗口的动画显示过程中看到壁纸。

        3. 如果前面找到的接下来要显示壁纸的窗口是一个Activity窗口，即变量foundW所描述的一个WindowState对象的成员变量mAppToken的值不等于null，并且它所指向的一个AppWindowToken对象的成员变量hiddenRequested的值等于true，那么就说明与窗口foundW所对应的一个Activity组件已经被请求隐藏起来了。在这种情况下，当前正在显示壁纸的窗口就会仍然被当作是接下来壁纸窗口的目标窗口。由于此前我们已经将WindowManagerService类的成员变量mWallpaperTarget的值设置了为foundW，因此，这时候就需要将它的值修改为oldW。

        这段代码执行完成之后，窗口堆栈的状态就如图8所示：

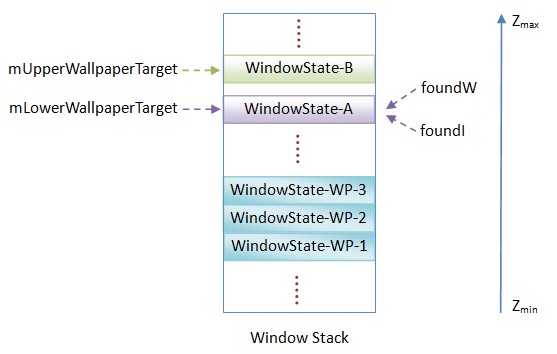


图8 mUpperWallpaperTarget、mLowerWallpaperTarget、foundW和foundI的关系

       标号为Label #4的代码如下所示：

// Is it time to stop animating?

**boolean** lowerAnimating = mLowerWallpaperTarget.mAnimation != **null**

|| (mLowerWallpaperTarget.mAppToken != **null**

&& mLowerWallpaperTarget.mAppToken.animation != **null**);

**boolean** upperAnimating = mUpperWallpaperTarget.mAnimation != **null**

|| (mUpperWallpaperTarget.mAppToken != **null**

&& mUpperWallpaperTarget.mAppToken.animation != **null**);

**if** (!lowerAnimating || !upperAnimating) {

......

mLowerWallpaperTarget = **null**;

mUpperWallpaperTarget = **null**;

}

       这段代码检查WindowManagerService类的成员变量mLowerWallpaperTarget和mUpperWallpaperTarget所描述的两个窗口的动画是否已经显示结束。如果已经显示结束，那么就会将这两个成员变量的值设置为null。

       注意，如果一个窗口的动画已经显示结束，那么用来描述它的一个WindowState对象的成员变量mAnimation的值就会等于null。另外，如果一个Activity窗口的动画已经显示结束，那么用来描述它的WindowState对象的成员变量mAppWindowToken所指向的一个AppWindowToken对象的成员变量animation的值也会等于null。

       标号为Label #5的代码如下所示：

**boolean** visible = foundW != **null**;

**if** (visible) {

// The window is visible to the compositor... but is it visible

// to the user? That is what the wallpaper cares about.

visible = isWallpaperVisible(foundW);

......

// If the wallpaper target is animating, we may need to copy

// its layer adjustment. Only do this if we are not transfering

// between two wallpaper targets.

mWallpaperAnimLayerAdjustment =

(mLowerWallpaperTarget == **null** && foundW.mAppToken != **null**)

? foundW.mAppToken.animLayerAdjustment : 0;

**final** **int** maxLayer = mPolicy.getMaxWallpaperLayer()

\* TYPE\_LAYER\_MULTIPLIER

+ TYPE\_LAYER\_OFFSET;

// Now w is the window we are supposed to be behind... but we

// need to be sure to also be behind any of its attached windows,

// AND any starting window associated with it, AND below the

// maximum layer the policy allows for wallpapers.

**while** (foundI > 0) {

WindowState wb = localmWindows.get(foundI-1);

**if** (wb.mBaseLayer < maxLayer &&

wb.mAttachedWindow != foundW &&

wb.mAttachedWindow != foundW.mAttachedWindow &&

(wb.mAttrs.type != TYPE\_APPLICATION\_STARTING ||

wb.mToken != foundW.mToken)) {

// This window is not related to the previous one in any

// interesting way, so stop here.

**break**;

}

foundW = wb;

foundI--;

}

}

       当变量foundW的值不等于null时，就说明前面找到了一个接下来要显示壁纸的窗口。在这种情况下，需要做三件事件：

       1. 判断窗口foundW是否是可见的，这是通过调用WindowManagerService类的成员函数isWallpaperVisible来实现的。如果可见，那么变量visible的值就会等于true，否则就会等于false。后面在调整壁纸窗口在窗口堆栈中的位置时，会根据变量visible的值来决定要显示壁纸窗口还是隐藏壁纸窗口。

       2. 检查窗口foundW是否是一个Activity窗口。如果是的话，那么就会将用来描述它的一个WindowState对象的成员变量mAppToken所指向的一个AppWindowToken对象的成员变量animLayerAdjustment的值保存在WindowManagerService类的成员变量mWallpaperAnimLayerAdjustment中。在计算壁纸窗品的Z轴位置的时候，需要使用到WindowManagerService类的成员变量mWallpaperAnimLayerAdjustment，用来调整壁纸窗品的Z轴位置。在后面一篇文章分析窗口的Z轴位置的计算方法时，我们再详细分析壁纸窗口的Z轴位置是如何计算的。注意，如果这时候系统的壁纸窗口有两个目标窗口，即WindowManagerService类的成员变量mLowerWallpaperTarget的值不等于null，那么就说明壁纸窗口的目标窗口正在显示动画的过程中。在这种情况下，就不需要调整壁纸窗品的Z轴位置，即会将WindowManagerService类的成员变量mLowerWallpaperTarget的值设置为0。等到壁纸窗口的目标窗口结束动画显示过程之后，再来调整它的Z轴位置。

       3. 检查窗口foundW的下面是否存在一些关联的窗口。如果存在的话，就需要将壁纸窗口放置在这些关联的窗口中Z轴位置最低的窗口的下面。这段代码通过一个while循环从窗口foundW的下面一个窗口开始往下检查，直到找到一个没有关联的窗口为止。在检查的过程中，每碰到一个关联的窗口，那么就让变量foundW指向它，并且将变量foundI的值减少1。这样最终得到的变量foundW和foundI就是用来描述与窗口foundW有联的、Z轴位置最低的窗口及其在窗口堆栈中的位置。

       前面提到，窗口foundW所关联的窗口四种，即对于一个窗口wb来，如果它满足以下四个条件，那么它就与窗口foundW有关联：

       A.  窗口wb与窗口foundW对应的是同一个窗品令牌，即分别用来描述窗口wb和窗口foundW的两个WindowState对象的成员变量mToken指向的是同一个WindowToken对象。

       B.  窗口wb附加在窗口foundW上，即用来描述窗口wb的一个WindowState对象的成员变量mAttachedWindow与变量foundW指向的是同一个WindowState对象。

       C.  窗口wb与窗口foundW附加在同一个窗口上，即分别用来描述窗口wb和窗口foundW的两个WindowState对象的成员变量mAttachedWindow指向的是同一个WindowState对象。

       D. 窗口wb是窗口foundW的启动窗口，即用来描述窗口wb的一个WindowState对象的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_APPLICATION\_STARTING。

       此外，WindowManagerService类的成员变量mPolicy所指向的一个PhoneWindowManager对象会规定系统中的壁纸窗口的Z轴位置不能大于某一个值，也就是说，壁纸窗口的Z轴位置有一个最大值限制。这个限制值可以通过调用WindowManagerService类的成员变量mPolicy所指向的一个PhoneWindowManager对象的成员函数getMaxWallpaperLayer来获得。获得了这个限制值之后，还需要乘以一个窗口类型因子TYPE\_LAYER\_MULTIPLIER，最后再加一个窗口类型偏移值TYPE\_LAYER\_OFFSET，就可以得到壁纸窗口的最大Z轴位置限制值maxLayer。这时候如果在窗口foundW的下面找到一个窗口wb，它的Z轴位置大于等于maxLayer，即用来描述它的一个WindowState对象的成员变量mBaseLayer的值大于maxLayer，那么也会认为窗口wb是与窗口foundW有关联的。

      我们通过图9和图10来说明查找与窗口foundW关联的、Z轴位置最小的窗口的过程：

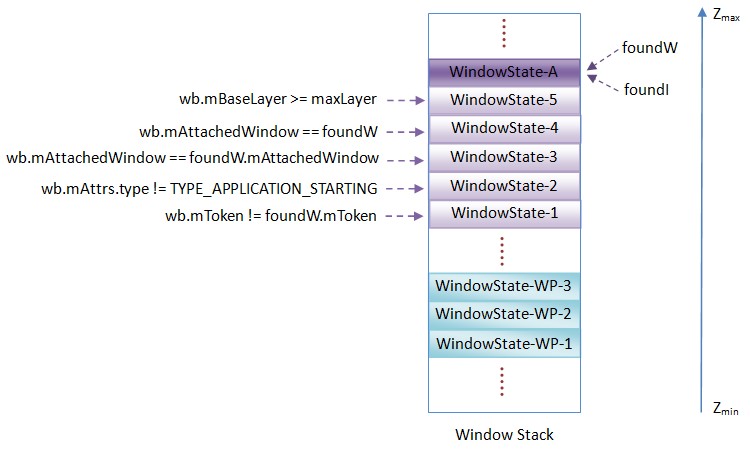


图9 查找与窗口foundW关联的窗口之前

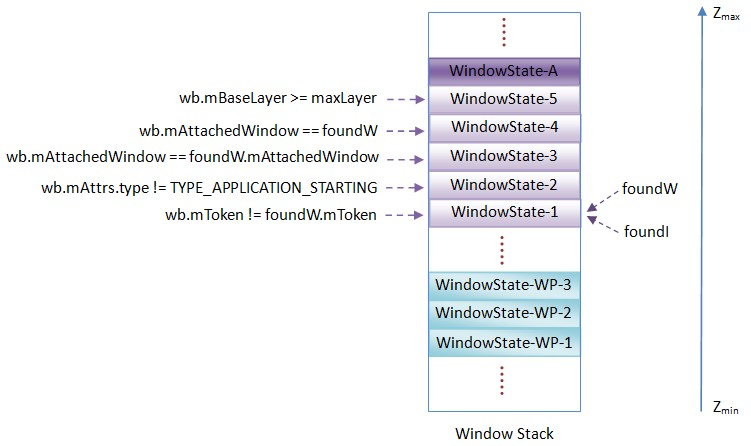


图10 查找与窗口foundW关联的窗口之后

       标号为Label #6的代码如下所示：

// Start stepping backwards from here, ensuring that our wallpaper windows

// are correctly placed.

**int** curTokenIndex = mWallpaperTokens.size();

**while** (curTokenIndex > 0) {

curTokenIndex--;

WindowToken token = mWallpaperTokens.**get**(curTokenIndex);

**if** (token.hidden == visible) {

changed |= ADJUST\_WALLPAPER\_VISIBILITY\_CHANGED;

token.hidden = !visible;

// Need to do a layout to ensure the wallpaper now has the

// correct size.

mLayoutNeeded = **true**;

}

**int** curWallpaperIndex = token.windows.size();

**while** (curWallpaperIndex > 0) {

curWallpaperIndex--;

WindowState wallpaper = token.windows.**get**(curWallpaperIndex);

**if** (visible) {

updateWallpaperOffsetLocked(wallpaper, dw, dh, **false**);

}

// First, make sure the client has the current visibility

// state.

**if** (wallpaper.mWallpaperVisible != visible) {

wallpaper.mWallpaperVisible = visible;

**try** {

......

wallpaper.mClient.dispatchAppVisibility(visible);

} **catch** (RemoteException e) {

}

}

wallpaper.mAnimLayer = wallpaper.mLayer + mWallpaperAnimLayerAdjustment;

......

// First, if this window is at the current index, then all

// is well.

**if** (wallpaper == foundW) {

foundI--;

foundW = foundI > 0

? localmWindows.**get**(foundI-1) : **null**;

**continue**;

}

// The window didn't match... the current wallpaper window,

// wherever it is, is in the wrong place, so make sure it is

// not in the list.

**int** oldIndex = localmWindows.indexOf(wallpaper);

**if** (oldIndex >= 0) {

i......

localmWindows.remove(oldIndex);

mWindowsChanged = **true**;

**if** (oldIndex < foundI) {

foundI--;

}

}

// Now stick it in.

......

localmWindows.add(foundI, wallpaper);

mWindowsChanged = **true**;

changed |= ADJUST\_WALLPAPER\_LAYERS\_CHANGED;

}

}

       这段代码就是用来调整系统中的壁纸窗口在窗口堆栈中的位置的，目标就是要将它们放置在前面所找到的接下来要显示壁纸的窗口的下面。

       WindowManagerService类的成员变量mWallpaperTokens保存的是一系列WindowToken对象，它们描述的是系统中的壁纸窗口令牌。这些WindowToken对象都有一个成员变量windows，里面保存的是一系列WindowState对象，它们描述的是系统中的壁纸窗口。这段代码就目标就要通过两个嵌套的while循环来将这些WindowState对象调整到前面所找到的接下来要显示壁纸的窗口的下面去。

       在调整壁纸窗口在窗口堆栈中的位置的过程中，还会做以下四件事情：

       1. 设置壁纸窗口令牌的可见性。也就是说，如果一个用来描述壁纸窗口令牌的WindowToken对象token的成员变量hidden的值不等于前面得到的变量visible的值，那么就说明该壁纸窗口令牌的可见性发生了变化。由于WindowToken类的成员变量hidden是用来表示壁纸窗口令牌的不可见状态的，而变量visible是用来表示接下来要显示壁纸的窗口是可见的，因此，当一个壁纸窗口令牌的可见性发生变化时，就要将用来描述它的WindowToken对象token的成员变量hidden的值设置为!visbile。壁纸窗口令牌的可见性发生了变化之后，需要重新刷新系统的UI，因此，就需要将WindowManagerService类的成员变量mLayoutNeeded 的值设置为true，并且将函数返回值changed的ADJUST\_WALLPAPER\_VISIBILITY\_CHANGED位设置为1。

       2. 在前面所找到的接下来要显示壁纸的窗口是可见的情况下，即在变量visible的值等于true的情况下，重新计算每一个壁纸窗口wallpaper在X轴和Y轴上的偏移位置，这是通过调用WindowManagerService类的成员函数updateWallpaperOffsetLocked来实现的。

       3. 如果一个壁纸窗口之前是不可见的，现在变得可见了，或者之前是可见的，现在变得不可见了，具体就表现在用来描述该壁纸窗口的一个WindowState对象的成员变量mWallpaperVisible的值不等于变量visible的值，那么就需要该WindowState对象的成员变量mWallpaperVisible的值设置为visible，并且向提供该壁纸窗口的服务发送一个可见性变化事件通知。

       4. 调整每一个壁纸窗口的Z轴位置。一个壁纸窗口的Z轴位置保存在用来描述它的一个WindowState对象的成员变量mLayer中，用这个成员变量的值加上前面已经计算好的壁纸窗口的Z轴位置调整值，即保存在WindowManagerService类的成员变量mWallpaperAnimLayerAdjustment中的值，就可以得到一个壁纸窗口的最终Z轴位置值，并且保存WindowState对象的成员变量mAnimLayer中。

       前面在分析WindowManagerService类的成员函数adjustWallpaperWindowsLocked的实现框架时提到，在调整系统中的壁纸窗口在窗口堆栈中的位置之前，变量foundW描述的应该是Z轴位置最大的壁纸窗口，而变量foundI记录的是需要显示壁纸的窗口在窗口堆栈中的位置，如图11所示：

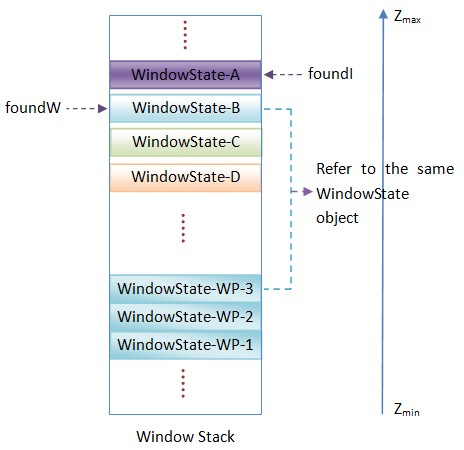


图11 调整壁纸窗口前的窗口堆栈状态

      在图11中，接下来需要显示壁纸的是窗口A，在它下面依次是窗口B、C和D，并且系统中存在着三个壁纸窗口，它们的编号分别为1、2和3。假设窗口B和编号为3的壁纸窗口是同一个窗口，那么就说明编号为3的壁纸窗口已经在窗口堆栈中的正确位置了，因此，就不需要调整它在窗口堆栈中的位置了。这时候窗口堆栈中的状态如图12所示：

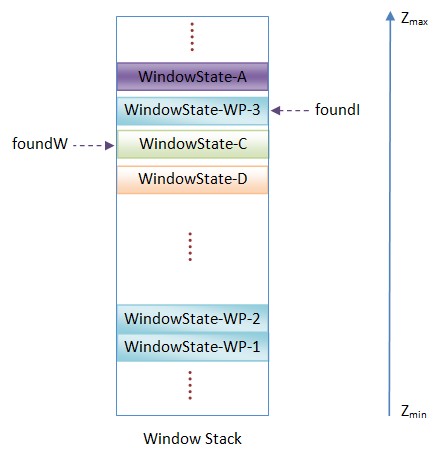


图12 处理完成编号为3的壁纸窗口后的窗口堆栈状态

      在图12中，假设窗口C和编号为2的壁纸窗口不是同一个窗口，那么就需要将编号为2的壁纸窗口放置在窗口C的位置上，如图13所示：

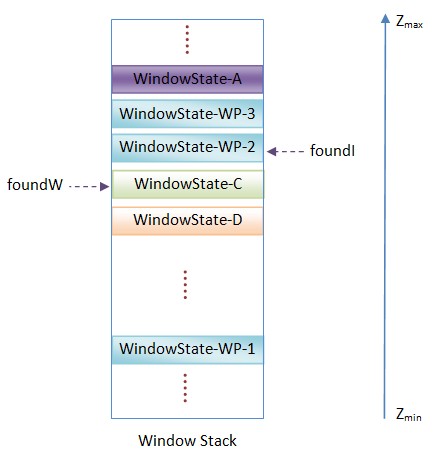


图13 处理完成编号为2的壁纸窗口后的窗口堆栈状态

       在图13中，假设窗口C和编号为1的壁纸窗口也不是同一个窗口，那么就需要将编号为1的壁纸窗口放置在窗口C的位置上，如图14所示：

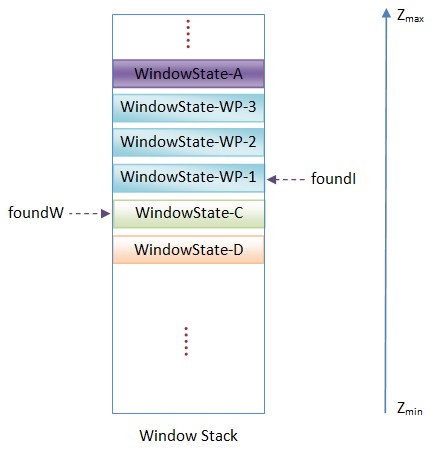


图14 处理完成编号为1的壁纸窗口的窗口堆栈状态

      处理完成编号为1的壁纸窗口之后，系统中所有的壁纸窗口都调整到窗口A的下面去了，这样在下一次在刷新系统UI时，就可以将系统中的壁纸窗口作为窗口A的背景了。

      至此，我们就分析完成壁纸窗口在窗口堆栈中的位置调整过程了，WindowManagerService服务对壁纸窗口的管理也分析完成了。结合前面Android窗口管理服务WindowManagerService对窗口的组织方式分析和Android窗口管理服务WindowManagerService对输入法窗口（Input Method Window）的管理分析这两篇文章，我们就可以对WindowManagerService服务在内部所维护的窗口堆栈有一个清晰的认识了。

      当系统中的所有窗口都在窗口堆栈排列好之后，WindowManagerService服务就可以计算每一个窗口的Z轴坐标了，以便可以传递给SurfaceFlinger服务做可见性计算，从而正确地将系统的UI渲染出来。在接下来的一篇文章中，我们就将继续分析WindowManagerService服务计算窗口的Z轴坐标的过程，敬请关注！

**老罗的新浪微博：**[**http://weibo.com/shengyangluo**](http://weibo.com/shengyangluo)**，欢迎关注！**

本文出自 “[老罗的Android之旅](http://shyluo.blog.51cto.com/)” 博客，请务必保留此出处<http://shyluo.blog.51cto.com/5725845/1229283>

推荐：[Android窗口管理服务WindowManagerService对窗口的组织方式分析](http://www.itboth.com/d/ZFJFzu/android-windowmanagerservice)

[我们知道，在Android系统中，Activity是以堆栈的形式组织在ActivityManagerService服务中的。与Activity类似，Android系统中的窗口也是以堆栈的形式组织在WindowManagerSe

# Android窗口管理服务WindowManagerService显示窗口动画的原理分析

[审核此文](javascript:showCheckDoc('Q7BzM3','http://www.itboth.com/d/Q7BzM3/android');)

dragon\_feeling 分享于 2017-07-08 阅读 76 收藏 0

主题 [android](http://www.itboth.com/tag/android) [动画](http://www.itboth.com/tag/%E5%8A%A8%E7%94%BB) [原理](http://www.itboth.com/tag/%E5%8E%9F%E7%90%86) [管理](http://www.itboth.com/tag/%E7%AE%A1%E7%90%86)

**领取地址**：

推荐：[Android窗口管理服务WindowManagerService显示Activity组件的启动窗口（Starting Window）的过程分析](http://www.itboth.com/d/JBBvqaiMRNVr/window-android-starting-windowmanagerservice-activity)

[在Android系统中，Activity组件在启动之后，并且在它的窗口显示出来之前，可以显示一个启动窗口。这个启动窗口可以看作是Activity组件的预览窗口，是由WindowManagerServi

在前一文中，我们分析了Activity组件的切换过程。从这个过程可以知道，所有参与切换操作的窗口都会被设置切换动画。事实上，一个窗口在打开（关闭）的过程中，除了可能会设置切换动画之外，它本身也可能会设置有进入（退出）动画。再进一步地，如果一个窗口是附加在另外一个窗口之上的，那么被附加窗口所设置的动画也会同时传递给该窗口。本文就详细分析WindowManagerService服务显示窗口动画的原理。

在Android系统中，窗口动画的本质就是对原始窗口施加一个变换（Transformation）。在线性数学中，对物体的形状进行变换是通过乘以一个矩阵（Matrix）来实现，目的就是对物体进行偏移、旋转、缩放、切变、反射和投影等。因此，给窗口设置动画实际上就给窗口设置一个变换矩阵（Transformation Matrix）。

如前所述，一个窗口在打开（关闭）的过程，可能会被设置三个动画，它们分别是窗口本身所设置的进入（退出）动画（SelfTransformation）、从被附加窗口传递过来的动画（Attached Transformation），以及宿主Activity组件传递过来的切换动画（App Transformation）。这三个Transformation组合在一起形成一个变换矩阵，以60fps的速度应用在窗口的原始形状之上，完成窗口的动画过程，如图1所示。

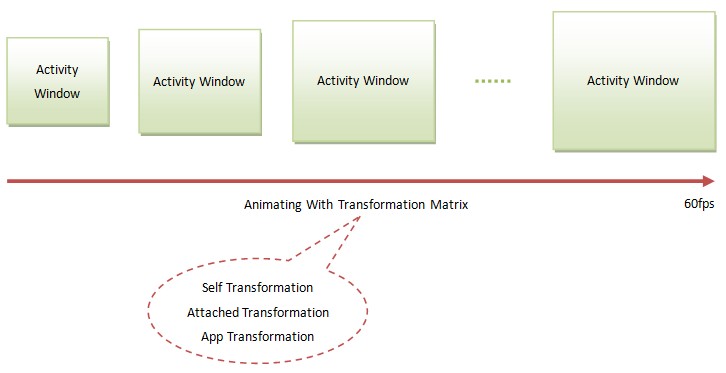


图1 窗口的动画显示过程

从上面的分析可以知道，窗口的变换矩阵是应用在窗口的原始位置和大小之上的，因此，在显示窗口的动画之前，除了要给窗口设置变换矩阵之外，还要计算好窗口的原始位置和大小，以及布局和绘制好窗口的UI。在前面[Android窗口管理服务WindowManagerService计算Activity窗口大小的过程分析](http://blog.csdn.net/luoshengyang/article/details/8479101)和[Android应用程序窗口（Activity）的测量（Measure）、布局（Layout）和绘制（Draw）过程分析](http://blog.csdn.net/luoshengyang/article/details/8372924)这两篇文章中，我们已经分析过窗口的位置和大小计算过程以及窗口UI的布局和绘制过程了，本文主要关注窗口动画的设置、合成和显示过程。这三个过程通过以下四个部分的内容来描述：

1.窗口动画的设置过程

2. 窗口动画的显示框架

3. 窗口动画的推进过程

4. 窗口动画的合成过程

其中，窗口动画的设置过程包括上述三个动画的设置过程，窗口动画的推进过程是指定动画的一步一步地迁移的过程，窗口动画的合成过程是指上述三个动画组合成一个变换矩阵的过程，后两个过程包含在了窗口动画的显示框架中。

一. 窗口动画的设置过程

窗口被设置的动画虽然可以达到三个，但是这三个动画可以归结为两类，一类是普通动画，例如，窗口在打开过程中被设置的进入动画和在关闭过程中被设置的退出动画，另一类是切换动画。其中，SelfTransformation和Attached Transformation都是属于普通动画，而App Transformation属于切换动画。接下来我们就分别分析这两种类型的动画的设置过程。

1. 普通动画的设置过程

从前面[Android窗口管理服务WindowManagerService显示Activity组件的启动窗口（Starting Window）的过程分析](http://blog.csdn.net/luoshengyang/article/details/8577789)一文可以知道，窗口在打开的过程中，是通过调用WindowState类的成员函数performShowLocked来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **final** **class** **WindowState** **implements** **WindowManagerPolicy**.**WindowState** {

......

**boolean** **performShowLocked**() {

......

**if** (mReadyToShow && isReadyForDisplay()) {

......

**if** (!showSurfaceRobustlyLocked(**this**)) {

**return** **false**;

}

......

applyEnterAnimationLocked(**this**);

......

}

**return** **true**;

}

......

}

......

}

这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

WindowState类的成员函数performShowLocked首先是调用WindowManagerService类的成员函数showSurfaceRobustlyLocked来通知SurfaceFlinger服务将当前正在处理的窗口设置为可见，接着再调用WindowManagerService类的成员函数applyEnterAnimationLocked来给当前正在处理的窗口设置一个进入动画，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **void** **applyEnterAnimationLocked**(WindowState win) {

**int** transit = WindowManagerPolicy.TRANSIT\_SHOW;

**if** (win.mEnterAnimationPending) {

win.mEnterAnimationPending = **false**;

transit = WindowManagerPolicy.TRANSIT\_ENTER;

}

applyAnimationLocked(win, transit, **true**);

}

......

}

这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。 

如果参数win所指向的一个WindowState对象的成员变量mEnterAnimationPending的值等于true，那么就说明它所描述的窗口正在等待显示，也就是正处于不可见到可见状态的过程中，那么WindowManagerService类的成员函数applyEnterAnimationLocked就会对该窗口设置一个类型为WindowManagerPolicy.TRANSIT\_ENTER的动画，否则的话，就会对该窗口设置一个类型为WindowManagerPolicy.TRANSIT\_SHOW的动画。

确定好窗口的动画类型之后，WindowManagerService类的成员函数applyEnterAnimationLocked就调用另外一个成员函数applyAnimationLocked来为窗口创建一个动画了。接下来我们先分析窗口在关闭的过程中所设置的动画类型，然后再来分析WindowManagerService类的成员函数applyAnimationLocked的实现。

从前面[Android窗口管理服务WindowManagerService计算Activity窗口大小的过程分析](http://blog.csdn.net/luoshengyang/article/details/8479101)一文可以知道，当应用程序进程请求WindowManagerService服务刷新一个窗口的时候，会调用到WindowManagerService类的成员函数relayoutWindow。WindowManagerService类的成员函数relayoutWindow在执行的过程中，如果发现需要将一个窗口从可见状态设置为不可见状态时，也就是发现需要关闭一个窗口时，就会对该窗口设置一个退出动出，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**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) {

......

**synchronized**(mWindowMap) {

WindowState win = windowForClientLocked(session, client, **false**);

......

**if** (viewVisibility == View.VISIBLE &&

(win.mAppToken == **null** || !win.mAppToken.clientHidden)) {

......

} **else** {

......

**if** (win.mSurface != **null**) {

......

// If we are not currently running the exit animation, we

// need to see about starting one.

**if** (!win.mExiting || win.mSurfacePendingDestroy) {

// Try starting an animation; if there isn't one, we

// can destroy the surface right away.

**int** transit = WindowManagerPolicy.TRANSIT\_EXIT;

......

**if** (!win.mSurfacePendingDestroy && win.isWinVisibleLw() &&

applyAnimationLocked(win, transit, **false**)) {

......

win.mExiting = **true**;

}

......

}

......

}

......

}

......

}

......

}

......

}

这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

WindowState对象win描述的便是要刷新的窗口。当参数viewVisibility的值不等于View.VISIBLE时，就说明要将WindowState对象win所描述的窗口设置为不可见。另一方面，如果WindowState对象win的成员变量mAppToken的值不等于null，并且这个成员变量所指向的一个AppWindowToken对象的成员变量clientHidden的值等于true，那么就说明WindowState对象win所描述的窗口是一个与Activity组件相关的窗口，并且该Activity组件是处于不可见状态的。在这种情况下，也需要将WindowState对象win所描述的窗口设置为不可见。

一旦WindowState对象win所描述的窗口要设置为不可见，就需要考虑给它设置一个退出动画，不过有四个前提条件：

1. 该窗口有一个绘图表面，即WindowState对象win的成员变量mSurface的值不等于null；

2. 该窗口的绘图表面不是处于等待销毁的状态，即WindowState对象win的成员变量mSurfacePendingDestroy的值不等于true；

3.该窗口不是处于正在关闭的状态，即WindowState对象win的成员变量mExiting的值不等于true；

4.该窗口当前正在处于可见的状态，即WindowState对象win的成员isWinVisibleLw的返回值等于true。

在满足上述四个条件的情况下，就说明WindowState对象win所描述的窗口的状态要由可见变为不可见，因此，就需要给它设置一个退出动画，即一个类型为WindowManagerPolicy.TRANSIT\_EXIT的动画，这同样是通过调用WindowManagerService类的成员函数applyAnimationLocked来实现的。

从上面的分析就可以知道，无论是窗口在打开时所需要的进入动画，还是窗口在关闭时所需要的退出动画，都是通过调用WindowManagerService类的成员函数applyAnimationLocked来设置的，它的实现如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **boolean** **applyAnimationLocked**(WindowState win,

**int** transit, **boolean** isEntrance) {

**if** (win.mLocalAnimating && win.mAnimationIsEntrance == isEntrance) {

// If we are trying to apply an animation, but already running

// an animation of the same type, then just leave that one alone.

**return** **true**;

}

// Only apply an animation if the display isn't frozen. If it is

// frozen, there is no reason to animate and it can cause strange

// artifacts when we unfreeze the display if some different animation

// is running.

**if** (!mDisplayFrozen && mPolicy.isScreenOn()) {

**int** anim = mPolicy.selectAnimationLw(win, transit);

**int** attr = -1;

Animation a = **null**;

**if** (anim != 0) {

a = AnimationUtils.loadAnimation(mContext, anim);

} **else** {

**switch** (transit) {

**case** WindowManagerPolicy.TRANSIT\_ENTER:

attr = com.android.internal.R.styleable.WindowAnimation\_windowEnterAnimation;

**break**;

**case** WindowManagerPolicy.TRANSIT\_EXIT:

attr = com.android.internal.R.styleable.WindowAnimation\_windowExitAnimation;

**break**;

**case** WindowManagerPolicy.TRANSIT\_SHOW:

attr = com.android.internal.R.styleable.WindowAnimation\_windowShowAnimation;

**break**;

**case** WindowManagerPolicy.TRANSIT\_HIDE:

attr = com.android.internal.R.styleable.WindowAnimation\_windowHideAnimation;

**break**;

}

**if** (attr >= 0) {

a = loadAnimation(win.mAttrs, attr);

}

}

......

**if** (a != **null**) {

......

win.setAnimation(a);

win.mAnimationIsEntrance = isEntrance;

}

}

......

**return** win.mAnimation != **null**;

}

......

}

这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

参数win描述的是要设置动画的窗口，参数transit描述的是要设置的动画的类型，而参数isEntrance描述的是要设置的动画是进入类型还是退出类型的。

如果参数win所指向的一个WindowState对象的成员变量mLocalAnimating的值等于true，那么就说明它所描述的窗口已经被设置过动画了，并且这个动画正在显示的过程中。在这种情况下，如果这个WindowState对象的成员变量mAnimationIsEntrance的值等于参数isEntrance的值，那么就说明该窗口正在显示的动画就是所要求设置的动画，这时候就不需要给窗口重新设置一个动画了，因此，WindowManagerService类的成员函数applyAnimationLocked就直接返回了。

我们假设需要给参数win所描述的窗口设置一个新的动画，这时候还需要继续判断屏幕当前是否是处于非冻结和点亮的状态的。只有在屏幕不是被冻结并且是点亮的情况下，WindowManagerService类的成员函数applyAnimationLocked才真正需要给参数win所描述的窗口设置一个动画，否则的话，设置了也是无法显示的。当WindowManagerService类的成员变量mDisplayFrozen的时候，就说明屏幕不是被冻结的，而当WindowManagerService类的成员变量mPolicy所指向的一个PhoneWindowManager对象的成员函数isScreenOn的返回值等于true的时候，就说明屏幕是点亮的。在满足上述两个条件的情况下，WindowManagerService类的成员函数applyAnimationLocked就开始给参数win所描述的窗口创建动画了。

WindowManagerService类的成员函数applyAnimationLocked首先是检查WindowManagerService类的成员变量mPolicy所指向的一个PhoneWindowManager对象是否可以为参数win所描述的窗口提供一个类型为transit的动画。如果可以的话，那么调用这个PhoneWindowManager对象的成员函数selectAnimationLw的返回值anim就不等于0，这时候WindowManagerService类的成员函数applyAnimationLocked就可以调用AnimationUtils类的静态成员函数loadAnimation来根据该返回值anim来创建一个动画，并且保存在变量a中。

如果WindowManagerService类的成员变量mPolicy所指向的一个PhoneWindowManager对象不可以为参数win所描述的窗口提供一个类型为transit的动画的话，那么WindowManagerService类的成员函数applyAnimationLocked就需要根据该窗口的布局参数来创建这个动画了。这个创建过程分为两步执行：

1. 将参数transit的值转化为一个对应的动画样式名称；

2. 调用WindowManagerService类的成员函数loadAnimation来在指定的窗口布局参数中创建前面第1步所指定样式名称的动画，并且保存在变量a中，其中，指定的窗口布局参数是由WindowState对象win的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象来描述的。

最后，如果变量a的值不等于null，即前面成功地为参数win所描述的窗口创建了一个动画，那么接下来就会将该动画设置给参数win所描述的窗口。这是通过参数win所指向的一个WindowState对象的成员函数setAnimation来实现的，实际上就是将变量所指向的一个Animation对象保存在参数win所指向的一个WindowState对象的成员变量mAnimation中。同时，WindowManagerService类的成员函数applyAnimationLocked还会将参数isEntrance的值保存在参数win所指向的一个WindowState对象的成员变量mAnimationIsEntrance，以表明前面给它所设置的动画是属于进入类型还是退出类型的。

2. 切换动画的设置过程

从前面[Android窗口管理服务WindowManagerService切换Activity窗口（App Transition）的过程分析](http://blog.csdn.net/luoshengyang/article/details/8596449)一文可以知道，如果一个窗口属于一个Activity组件窗口，那么当该Activity组件被切换的时候，就会被设置一个切换动画，这是通过调用WindowManagerService类的成员函数setTokenVisibilityLocked来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**boolean** **setTokenVisibilityLocked**(AppWindowToken wtoken, WindowManager.LayoutParams lp,

**boolean** visible, **int** transit, **boolean** performLayout) {

**boolean** delayed = **false**;

......

**if** (wtoken.hidden == visible) {

**final** **int** N = wtoken.allAppWindows.size();

......

**boolean** runningAppAnimation = **false**;

**if** (transit != WindowManagerPolicy.TRANSIT\_UNSET) {

**if** (wtoken.animation == sDummyAnimation) {

wtoken.animation = **null**;

}

applyAnimationLocked(wtoken, lp, transit, visible);

......

**if** (wtoken.animation != **null**) {

delayed = runningAppAnimation = **true**;

}

}

**for** (**int** i=0; i<N; i++) {

WindowState win = wtoken.allAppWindows.get(i);

......

**if** (visible) {

**if** (!win.isVisibleNow()) {

**if** (!runningAppAnimation) {

applyAnimationLocked(win,

WindowManagerPolicy.TRANSIT\_ENTER, **true**);

}

......

}

} **else** **if** (win.isVisibleNow()) {

**if** (!runningAppAnimation) {

applyAnimationLocked(win,

WindowManagerPolicy.TRANSIT\_EXIT, **false**);

}

......

}

}

......

}

**if** (wtoken.animation != **null**) {

delayed = **true**;

}

**return** delayed;

}

......

}

这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

参数wtoken描述的是要切换的Activity组件，参数lp描述的是要用来创建切换动画的布局参数，参数transit描述的是要创建的切换动画的类型，而参数visible描述的是要切换的Activity组件接下来是否是可见的。

WindowManagerService类的成员函数setTokenVisibilityLocked首先判断要切换的Activity组件当前的可见性是否已经就是要设置的可见性，即参数wtoken所指向的一个AppWindowToken对象的成员变量hidden的值是否不等于参数visible的值。如果不等于的话，就说明要切换的Activity组件当前的可见性已经就是要设置的可见性了，这时候WindowManagerService类的成员函数setTokenVisibilityLocked就不用再为它设置切换动画了。

我们假设要切换的Activity组件当前的可见性不是要求设置的可见性，即参数wtoken所指向的一个AppWindowToken对象的成员变量hidden的值等于参数visible的值，那么WindowManagerService类的成员函数setTokenVisibilityLocked还会继续检查参数transit描述的是否是一个有效的动画类型，即它的值是否不等于WindowManagerPolicy.TRANSIT\_UNSET。如果参数transit描述的是一个有效的动画类型的话，那么WindowManagerService类的成员函数setTokenVisibilityLocked接下来就会执行以下三个操作：

1. 判断要切换的Activity组件当前是否被设置了一个哑动画，即参数wtoken所指向的一个AppWindowToken对象的成员变量animation是否与WindowManagerService类的成员变量sDummyAnimation指向了同一个Animation对象。如果是的话，那么就会将wtoken所指向的一个AppWindowToken对象的成员变量animation的值设置为null，因为接下来要重新为它设置一个新的Animation对象。从前面[Android窗口管理服务WindowManagerService切换Activity窗口（App Transition）的过程分析](http://blog.csdn.net/luoshengyang/article/details/8596449)一文可以知道，一个需要参与切换的Activity组件会设置可见性的时候，是会被设置一个哑动画的。

2. 调用WindowManagerService类的四个参数版本的成员函数applyAnimationLocked根据参数lp、transit和visible的值来为要切换的Activity组件创建一个动画。

3.如果第2步可以成功地为要切换的Activity组件创建一个动画的话，那么这个动画就会保存在wtoken所指向的一个AppWindowToken对象的成员变量animation中，这时候就会将变量delayed和runningAppAnimation的值均设置为true。

变量runningAppAnimation的值等于true意味着与参数wtoken所描述的Activity组件所对应的窗口接下来要执行一个切换动画。在这种情况下，WindowManagerService类的成员函数setTokenVisibilityLocked就不需要为这些窗口单独设置一个进入或者退出类型的动画，否则的话，WindowManagerService类的成员函数setTokenVisibilityLocked就会根据这些窗口的当前可见性状态以及参数wtoken所描述的Activity组件被要求设置的可见性来单独设置一个进入或者退出类型的动画：

1. 如果一个窗口当前是不可见的，即用来描述它的一个WindowState对象的成员函数isVisibleNow的返回值等于false，但是参数wtoken所描述的Activity组件被要求设置成可见的，即参数visible的值等于true，那么就需要给该窗口设置一个类型为WindowManagerPolicy.TRANSIT\_ENTER的动画；

2.如果一个窗口当前是可见的，即用来描述它的一个WindowState对象的成员函数isVisibleNow的返回值等于true，但是参数wtoken所描述的Activity组件被要求设置成不可见的，即参数visible的值等于false，那么就需要给该窗口设置一个类型为WindowManagerPolicy.TRANSIT\_EXIT的动画。

给与参数wtoken所描述的Activity组件所对应的窗口设置动画是通过调用WindowManagerService类的三个参数版本的成员函数applyAnimationLocked来实现的，这个成员函数在前面已经分析过了。另外，与参数wtoken所描述的Activity组件所对应的窗口是保存在参数wtoken所指向的一个AppWindowToken对象的成员变量allAppWindows所描述的一个ArrayList中的，因此，通过遍历这个ArrayList，就可以为与参数wtoken所描述的Activity组件所对应的每一个窗口设置一个动画。

最后，如果前面成功地为参数wtoken所描述的Activity组件创建了一个切换动画，即该参数所描述的一个AppWindowToken对象的成员变量animation的值不等于null，那么WindowManagerService类的成员函数setTokenVisibilityLocked的返回值delayed就会等于true，表示参数wtoken所描述的Activity组件要执行一个动换动画，同时也表明该Activity组件的窗口要延迟到切换动画显示结束后，才真正显示出来。

接下来，我们继续分析WindowManagerService类的四个参数版本的成员函数applyAnimationLocked的实现，以便可以了解Activity组件的切换动画的创建过程，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **boolean** **applyAnimationLocked**(AppWindowToken wtoken,

WindowManager.LayoutParams lp, **int** transit, **boolean** enter) {

// Only apply an animation if the display isn't frozen. If it is

// frozen, there is no reason to animate and it can cause strange

// artifacts when we unfreeze the display if some different animation

// is running.

**if** (!mDisplayFrozen && mPolicy.isScreenOn()) {

Animation a;

**if** (lp != **null** && (lp.flags & FLAG\_COMPATIBLE\_WINDOW) != 0) {

a = **new** FadeInOutAnimation(enter);

......

} **else** **if** (mNextAppTransitionPackage != **null**) {

a = loadAnimation(mNextAppTransitionPackage, enter ?

mNextAppTransitionEnter : mNextAppTransitionExit);

} **else** {

**int** animAttr = 0;

**switch** (transit) {

**case** WindowManagerPolicy.TRANSIT\_ACTIVITY\_OPEN:

animAttr = enter

? com.android.internal.R.styleable.WindowAnimation\_activityOpenEnterAnimation

: com.android.internal.R.styleable.WindowAnimation\_activityOpenExitAnimation;

**break**;

**case** WindowManagerPolicy.TRANSIT\_ACTIVITY\_CLOSE:

animAttr = enter

? com.android.internal.R.styleable.WindowAnimation\_activityCloseEnterAnimation

: com.android.internal.R.styleable.WindowAnimation\_activityCloseExitAnimation;

**break**;

**case** WindowManagerPolicy.TRANSIT\_TASK\_OPEN:

animAttr = enter

? com.android.internal.R.styleable.WindowAnimation\_taskOpenEnterAnimation

: com.android.internal.R.styleable.WindowAnimation\_taskOpenExitAnimation;

**break**;

**case** WindowManagerPolicy.TRANSIT\_TASK\_CLOSE:

animAttr = enter

? com.android.internal.R.styleable.WindowAnimation\_taskCloseEnterAnimation

: com.android.internal.R.styleable.WindowAnimation\_taskCloseExitAnimation;

**break**;

**case** WindowManagerPolicy.TRANSIT\_TASK\_TO\_FRONT:

animAttr = enter

? com.android.internal.R.styleable.WindowAnimation\_taskToFrontEnterAnimation

: com.android.internal.R.styleable.WindowAnimation\_taskToFrontExitAnimation;

**break**;

**case** WindowManagerPolicy.TRANSIT\_TASK\_TO\_BACK:

animAttr = enter

? com.android.internal.R.styleable.WindowAnimation\_taskToBackEnterAnimation

: com.android.internal.R.styleable.WindowAnimation\_taskToBackExitAnimation;

**break**;

**case** WindowManagerPolicy.TRANSIT\_WALLPAPER\_OPEN:

animAttr = enter

? com.android.internal.R.styleable.WindowAnimation\_wallpaperOpenEnterAnimation

: com.android.internal.R.styleable.WindowAnimation\_wallpaperOpenExitAnimation;

**break**;

**case** WindowManagerPolicy.TRANSIT\_WALLPAPER\_CLOSE:

animAttr = enter

? com.android.internal.R.styleable.WindowAnimation\_wallpaperCloseEnterAnimation

: com.android.internal.R.styleable.WindowAnimation\_wallpaperCloseExitAnimation;

**break**;

**case** WindowManagerPolicy.TRANSIT\_WALLPAPER\_INTRA\_OPEN:

animAttr = enter

? com.android.internal.R.styleable.WindowAnimation\_wallpaperIntraOpenEnterAnimation

: com.android.internal.R.styleable.WindowAnimation\_wallpaperIntraOpenExitAnimation;

**break**;

**case** WindowManagerPolicy.TRANSIT\_WALLPAPER\_INTRA\_CLOSE:

animAttr = enter

? com.android.internal.R.styleable.WindowAnimation\_wallpaperIntraCloseEnterAnimation

: com.android.internal.R.styleable.WindowAnimation\_wallpaperIntraCloseExitAnimation;

**break**;

}

a = animAttr != 0 ? loadAnimation(lp, animAttr) : **null**;

......

}

**if** (a != **null**) {

......

wtoken.setAnimation(a);

}

}

......

**return** wtoken.animation != **null**;

}

......

}

这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

WindowManagerService类的四个参数版本的成员函数applyAnimationLocked和三个参数版本的成员函数applyAnimationLocked的实现是类似的，不过它是为Activity组件创建动画，并且：

1. 如果参数lp所描述的布局参数表明它是用来描述一个兼容窗口的，即它所指向的一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_COMPATIBLE\_WINDOW位不等于0，那么创建的切换动画就固定为FadeInOutAnimation。

2. 如果WindowManagerService类的成员变量mNextAppTransitionPackage的值不等于null，那么就说明Package名称为mNextAppTransitionPackage的Activity组件指定了一个自定义的切换动画，其中，指定的进入动画的类型由WindowManagerService类的成员变量mNextAppTransitionEnter来描述，指定的退出动画的类型由WindowManagerService类的成员变量mNextAppTransitionExit来描述。在这种情况下，WindowManagerService服务就会使用这个指定的切换动画，而不是使用默认的切换动画。一般来说，一个个Activity组件在调用成员函数startActivity来通知ActivityManagerService服务启动另外一个Activity之外，可以马上调用另外一个成员函数overridePendingTransition来指定自定久的切换动画。

3. 切换动画的类型主要分三类：第一类是和Activity相关的；第二类是和Task相关的；第三类是和Wallpaper相关的。这一点可以参考前面[Android窗口管理服务WindowManagerService切换Activity窗口（App Transition）的过程分析](http://blog.csdn.net/luoshengyang/article/details/8596449)一文。

切换动画创建成功之后，就会调用参数wtoken所指向的一个AppWindowToken对象的成员函数setAnimation来保存在其成员变量animation中，这样就表示它所描述的Activity组件被设置了一个切换动画。

二. 窗口动画的显示框架

窗口动画是在WindowManagerService服务刷新系统UI的时候显示的。从前面前面[Android窗口管理服务WindowManagerService切换Activity窗口（App Transition）的过程分析](http://blog.csdn.net/luoshengyang/article/details/8596449)一文可以知道，WindowManagerService服务刷新系统UI是通过调用WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLockedInner来实现的，接下来我们就主要分析与窗口动画的显示框架相关的逻辑，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **final** **void** **performLayoutAndPlaceSurfacesLockedInner**(

**boolean** recoveringMemory) {

......

Surface.openTransaction();

......

**try** {

......

**int** repeats = 0;

**int** changes = 0;

do {

repeats++;

**if** (repeats > 6) {

......

**break**;

}

// 1. 计算各个窗口的大小

// FIRST LOOP: Perform a layout, if needed.

**if** (repeats < 4) {

changes = performLayoutLockedInner();

**if** (changes != 0) {

**continue**;

}

} **else** {

Slog.w(TAG, "Layout repeat skipped after too many iterations");

changes = 0;

}

// 2. 推进各个Activity组件的切换动画

// Update animations of all applications, including those

// associated with exiting/removed apps

**boolean** tokensAnimating = **false**;

**final** **int** NAT = mAppTokens.size();

**for** (i=0; i<NAT; i++) {

**if** (mAppTokens.get(i).stepAnimationLocked(currentTime, dw, dh)) {

tokensAnimating = **true**;

}

}

**final** **int** NEAT = mExitingAppTokens.size();

**for** (i=0; i<NEAT; i++) {

**if** (mExitingAppTokens.get(i).stepAnimationLocked(currentTime, dw, dh)) {

tokensAnimating = **true**;

}

}

// 3. 推进各个窗口的动画

// SECOND LOOP: Execute animations and update visibility of windows.

......

animating = tokensAnimating;

......

mPolicy.beginAnimationLw(dw, dh);

**final** **int** N = mWindows.size();

**for** (i=N-1; i>=0; i--) {

WindowState w = mWindows.get(i);

......

**if** (w.mSurface != **null**) {

// Execute animation.

......

**if** (w.stepAnimationLocked(currentTime, dw, dh)) {

animating = **true**;

......

}

......

}

......

mPolicy.animatingWindowLw(w, attrs);

}

......

changes |= mPolicy.finishAnimationLw();

......

} **while** (changes != 0);

// 4. 更新各个窗口的绘图表面

// THIRD LOOP: Update the surfaces of all windows.

......

**final** **int** N = mWindows.size();

**for** (i=N-1; i>=0; i--) {

WindowState w = mWindows.get(i);

......

**if** (w.mSurface != **null**) {

......

//计算实际要显示的大小和位置

w.computeShownFrameLocked();

......

//设置大小和位置等

......

**if** (w.mAttachedHidden || !w.isReadyForDisplay()) {

......

} **else** **if** (w.mLastLayer != w.mAnimLayer

|| w.mLastAlpha != w.mShownAlpha

|| w.mLastDsDx != w.mDsDx

|| w.mLastDtDx != w.mDtDx

|| w.mLastDsDy != w.mDsDy

|| w.mLastDtDy != w.mDtDy

|| w.mLastHScale != w.mHScale

|| w.mLastVScale != w.mVScale

|| w.mLastHidden) {

......

//设置Z轴位置、Alpha通道和变换矩阵

......

**if** (w.mLastHidden && !w.mDrawPending

&& !w.mCommitDrawPending

&& !w.mReadyToShow) {

......

**if** (showSurfaceRobustlyLocked(w)) {

w.mHasDrawn = **true**;

w.mLastHidden = **false**;

}

......

}

......

}

......

}

......

}

......

} **catch** (RuntimeException e) {

......

}

......

Surface.closeTransaction();

......

// 5. 检查是否需要再一次刷新系统UI

**if** (needRelayout) {

requestAnimationLocked(0);

} **else** **if** (animating) {

requestAnimationLocked(currentTime+(1000/60)-SystemClock.uptimeMillis());

}

......

}

......

}

这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLockedInner按照以下步骤来显示窗口动画：

第一步是调用WindowManagerService类的成员函数performLayoutLockedInner来计算各个窗口的大小以及位置。只有知道了窗口的大小以及位置之后，我们才能它应用一个变换矩阵，然后得到窗口下一步要显示的大小以及位置。这一步可以参考前面[Android窗口管理服务WindowManagerService计算Activity窗口大小的过程分析](http://blog.csdn.net/luoshengyang/article/details/8479101)一文。

第二步是推进各个Activity组件的切换动画，也就是计算每一个Activity组件下一步所要执行的动画。注意，只有那些正在参与切换操作的Activity组件才设置有动画，而只有设置有切换动画的Activity组件才需要计算它们下一步所要执行的动画。从前面第一部分的内容可以知道，如果一个Activity组件参与了切换操作，那么用来描述它的一个AppWindowToken对象的成员变量animation就会指向一个Animation对象，用来描述一个切换动画。

用来描述系统当前的Activity组件的AppWindowToken对象有一部分保存在WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList中，另外一部分保存在WindowManagerService类的成员变量mExitingAppTokens所描述的一个ArrayList中。其中，保存在成员变量mAppTokens中的AppWindowToken对象有可能描述的就是正在打开的Activity组件，而保存在成员变量mExitingAppTokens中的AppWindowToken对象描述的就是正在关闭的Activity组件，这两类Activity组件都是参与了切换操作的，因此，我们需要计算它们下一步所要执行的动画，这是通过调用用来描述它们的AppWindowToken对象的成员函数stepAnimationLocked来实现的。注意，对于那些不是正在打开或者正在关闭的Activity组件，调用用来描述它们的AppWindowToken对象的成员函数stepAnimationLocked不会有任何效果，因为这些AppWindowToken对象的成员变量animation的值等于null。

在这一步中，如果有任何一个正在打开或者正在关闭的Activity组件的动画还没有执行完成，那么调用用来描述它的AppWindowToken对象的成员函数stepAnimationLocked的返回值就会等于true，这时候变量tokensAnimating的值也会等于true，表示Activity组件的切换动画还在进行中。后面我们再详细分析AppWindowToken类的成员函数stepAnimationLocked的实现。

第三步是推进各个Activity组件的动画，也就是计算每一个窗口下一步所要执行的动画。注意，只有那些被设置有动画的窗口才需要计算它们下一下所要执行的动画。从前面第一部分的内容可以知道，如果一个窗口设置有动画，那么用来描述它的一个WindowState对象的成员变量mAnimation就会指向一个Animation对象，用来描述一个窗口动画。

用来描述系统当前的窗口的WindowState对象保存在WindowManagerService类的成员变量mWindows所描述的一个ArrayList中，只要调用这些WindowState对象的成员函数stepAnimationLocked，就可以计算它们所描述的窗口下一步所要执行的动画。同样，对于那些没有设置动画的窗口来说，调用用来描述它们的WindowState对象的成员函数stepAnimationLocked是不会有任何效果的，因为这些WindowState对象的成员变量mAnimation的值等于null。

注意，对于那些还没有创建绘图表面的窗口来说，即使它们设置有动画，在这一步里面，也是不需要计算它们下一步所要执行的动画的，这是因为一个没有绘图表面的窗口是无法对它应用动画的。

此外，在计算所有窗口下一步要执行的动画之前以及之后，会通知系统的窗口管理策略类窗口下一次要执行的动画就要开始计算了以及已经计算结束了，同时，每计算完成一个窗口下一次要执行的动画之后，也会通知系统的窗口管理策略类该窗口下一次要执行的动画已经计算完成了，这分别是通过调用WindowManagerService类的成员变量mPolicy所指向的一个PhoneWindowManager对象的成员函数beginAnimationLw、finishAnimationLw和animatingWindowLw来实现的。

我们知道，在Android系统中，有两个特殊的系统窗口，即状态栏窗口和锁屏窗口。如果当前激活的窗口是一个全屏窗口的时候，那么系统的状态栏窗口就会被隐藏。同时，在显示锁屏窗口的时候，一般系统中的所有其它窗口都会被它挡在后面。但是有些被设置了FLAG\_SHOW\_WHEN\_LOCKED标志位的窗口，能够显示在锁屏窗口的上面。还有些被设置了FLAG\_DISMISS\_KEYGUARD位的窗口，它们能够解散系统当前正在显示的锁屏窗口。这些全屏窗口、能够显示在锁屏窗口上面的窗口以及能够解散锁屏窗口的窗口，会影响到系统的状态栏窗口和锁屏窗口是否需要显示的问题，而这是需要由窗口管理策略类来决定的。因此，在计算所有窗口下一步要执行的动画的前后，以及计算每一个窗口下一次要执行的动画的过程中，都要通知一下窗口管理策略类，以便它可以决定是否需要显示系统的状态栏窗口和锁屏窗口。

窗口管理策略类按照以下方法来决定是否需要显示系统的状态栏窗口和锁屏窗口：

1.在计算所有窗口下一步要执行的动画之前，假设状态栏窗口是不可见的，而锁屏窗口是可见的、不可解散的；

2.在计算一个窗口下一步要执行的动画的过程中，检查该窗口是否是可见的以及全屏显示的。如果该窗口是可见的，但是不是全屏的，那么就意味着状态栏窗口是可见的。同时，如果该窗口是可见的，并且也是全屏的，那么就会继续检查该窗口是否被设置了FLAG\_SHOW\_WHEN\_LOCKED和FLAG\_DISMISS\_KEYGUARD标志位。如果这两个标志位分别被设置了，那么就意味着锁屏窗口是不可见的和需要解散的。

3.在计算完成所有窗口下一步要执行的动画之后，根据第2步收集到的信息来决定是否要显示状态栏窗口以及锁屏窗口。如果系统当前存在状态栏窗口，并且第2步收集到的信息表明它是需要显示的，那么就会将它的状态设置为可见的；另一方面，如果系统当前存在状态栏窗口，并且第2步收集到的信息表明系统当前有一个全屏的窗口，那么状态栏窗口无论如何都是需要隐藏的。如果系统当前存在锁屏窗口，并且第2步收集到的信息表明它是不需要显示的或者需要解散的，那么就会将它的状态设置不可见，否则的话，就会将它的状态设置为可见。

在计算完成所有窗口下一步要执行的动画之后，如果状态栏窗口或者锁屏窗口的状态由可见变为不可见，或者由不可见变为可见，那么PhoneWindowManager类的成员函数finishAnimationLw的返回值就会不等于0，这意味着WindowManagerService服务需要重新布局系统中各个窗口，即重新计算各个窗口的大小、位置以及下一步要执行的动画等操作，因为状态栏窗口或者锁屏窗口的可见性变化引发窗口堆栈发生变化。这就是上述的第一步、第二步和第三步要放在一个while循环来执行的原因。但是，这个while循环不能无限地执行下去，否则的话，系统的UI就永远刷新不出来。这个while循环最多允许执行7次，并且各个窗口的大小和位置的重复计算次数最多为4次。

第四步是更新各个窗口的绘图表面。既然是更新各个窗口的绘图表面，那么就意味着只有具有绘图表面的窗口才需要更新。一个窗口如果具有绘图表面，那么用来描述它的一个WindowState对象的成员变量mSurface的值就不等于null。对于具有绘图表面的窗口，这一步主要是执行以下操作：

1. 调用用来描述它的WindowState对象的成员函数computeShownFrameLocked来计算它实际要显示的大小和位置，这是需要考虑它的原始大小和位置，以及它所被设置的变换矩阵。这个变换矩阵是通过组合它的动画来得到的，其中包括窗口本身所设置的及其所附加在的窗口所设置的动画，还有窗口所属的Activity组件所设置的切换动画。后面我们再分析WindowState类的成员函数computeShownFrameLocked的实现。

2. 将第1步计算得到的窗口实际要显示的大小以及位置设置到SurfaceFlinger服务中去。这一点可以参考前面[Android窗口管理服务WindowManagerService计算窗口Z轴位置的过程分析](http://blog.csdn.net/luoshengyang/article/details/8570428)一文。

3. 如果窗口当前已经就准备就绪显示，即用来描述它的WindowState对象的成员函数isReadyForDisplay的返回值等于true，并且它所附加在的窗口是可见的，即用来描述它的WindowState对象的成员变量mAttachedHidden的值等于false，那么只要满足以下条件之一，就需要考虑通知SurfaceFlinger服务将该窗口的状态设置为可见的：

A. 该窗口的Z轴位置发生了变化，即用来描述该窗口的WindowState对象的成员变量mLastLayer和mAnimLayer的值不相等；

B. 该窗口的Alpha通道值发生了变化，即用来描述该窗口的WindowState对象的成员变量mLastAlpha和mShownAlpha的值不相等；

C. 该窗口的变换矩阵发生了变化，即用来描述该窗口的WindowState对象的成员变量mLastDsDx、mLastDtDx、mLastDsDy和mLastDtDy的值不等于mDsDx、mDtDx、mDsDy和mDtDy的值；

D.该窗口在宽度和高度上所设置的缩放因子发生了变化，即用来描述该窗口的WindowState对象的成员变量mLastHScale和mLastVScale的值不等于mHScale和mVScale的值；

E.该窗口在上一次系统UI刷新时是不可见的，即用来描述该窗口的WindowState对象的成员变量mLastHidden的值等于true。

如果一个具有绘图表面的窗口满足上述条件，那么这一步会将该窗口的最新Z轴位置、Alpha通道值和变换矩阵设置到SurfaceFlinger服务中去，这一点可以参考前面[Android窗口管理服务WindowManagerService计算窗口Z轴位置的过程分析](http://blog.csdn.net/luoshengyang/article/details/8570428)一文。

此外，如果一个具有绘图表面的窗口满足的是上述的条件E，并且还满足以下条件：

F. 它的UI已经绘制完成，即用来描述它的WindowState对象的成员变量mDrawPending和mCommitDrawPending值等于false；

G. 它不是处于等待同一个窗口令牌的其它窗口的完成UI绘制的状态，即用来描述它的WindowState对象的成员变量mReadyToShow的值等于false。

那么这一步还会调用WindowManagerService类的成员函数showSurfaceRobustlyLocked来通知SurfaceFlinger服务将该窗口的状态设置为可见的。如果能够成功地通知SurfaceFlinger服务将该窗口的状态设置为可见，那么还会分别将用来描述该窗口的WindowState对象的成员变量mHasDrawn和mLastHidden的值设置为true和false，表示该窗口的UI已经绘制完成，并且状态已经设置为可见。

注意，上述四步对窗口的操作，例如设置窗口的大小、位置、Alpha通道值和变换矩阵等，都是在一个事务中执行的。这个事务从调用Surface类的静态成员函数openTransaction时开始，一直到调用Surface类的静态成员函数closeTransaction时结束。当事务结束的时候，前面所设置的窗口的状态才会批量地同步到SurfaceFlinger服务中去，这样就可以避免每修改一个窗口的一个状态，就触发SurfaceFlinger服务刷新一次系统的UI，造成屏幕闪烁。

第五步是检查是否需要再一次刷新系统UI。在三种情况下，是需要再一次刷新系统UI的。第一种情况是发现此时Activity组件的切换动画已经显示完成了；第二种情况是发现前面的操作会导致壁纸窗口的目标窗口被销毁了；第三种情况是发现此时还有窗口的动画未结束。

由于在Activity组件的切换过程中，很多操作都会被延迟执行，例如，窗口的Z轴位置的计算，因此，当出现第一种情况下，就需要重新执行这些延迟操作，主要就是重建窗口堆栈，以及计算每一个窗口的Z轴位置。在第二种情况下，由于当壁纸窗口的目标窗品被销毁之后，因此，就需要重新调整壁纸窗口在窗口堆栈中的位置。这两种情况都会导致变量needRelayout的值就等于true，表示需要马上重新刷新系统UI，这是通过以0为参数来调用WindowManagerService类的成员函数requestAnimationLocked来实现的。

在第三种情况中，变量animating的值会等于true，表示有窗口的动画还未结束，有可能是窗口本身的动画尚未结束，也有可能是Activity组件的切换动画尚未结束。在WindowManagerService服务中，窗口动画是以60帧每秒（fps）的速度来显示的，因此，这一步就会以约等于1000/60的参数来调用WindowManagerService类的成员函数requestAnimationLocked，表示要在1/60秒后再次刷新系统的UI，这样就可以把动画效果展现出来。

三.窗口动画的推进过程

从前面第一部分的内容可以知道，窗口动画有可能是是来自窗口本身所设置的动画，也有可能是来自于其宿主Activity组件所设置的切换动画，接下来我们就分别分析这两种类型的动画的推进过程。

1. 窗口动画的推进过程

从前面第二部分的内容可以知道，窗口动画的推进过程是由WindowState类的成员函数stepAnimationLocked的实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **final** **class** **WindowState** **implements** **WindowManagerPolicy**.**WindowState** {

......

// Currently running animation.

**boolean** mAnimating;

**boolean** mLocalAnimating;

Animation mAnimation;

......

**boolean** mHasLocalTransformation;

**final** Transformation mTransformation = **new** Transformation();

......

// This must be called while inside a transaction. Returns true if

// there is more animation to run.

**boolean** **stepAnimationLocked**(**long** currentTime, **int** dw, **int** dh) {

**if** (!mDisplayFrozen && mPolicy.isScreenOn()) {

// We will run animations as long as the display isn't frozen.

**if** (!mDrawPending && !mCommitDrawPending && mAnimation != **null**) {

......

mHasLocalTransformation = **true**;

**if** (!mLocalAnimating) {

......

mAnimation.initialize(mFrame.width(), mFrame.height(), dw, dh);

mAnimation.setStartTime(currentTime);

mLocalAnimating = **true**;

mAnimating = **true**;

}

mTransformation.clear();

**final** **boolean** more = mAnimation.getTransformation(

currentTime, mTransformation);

......

**if** (more) {

// we're not done!

**return** **true**;

}

......

}

......

}

......

mAnimating = **false**;

mLocalAnimating = **false**;

mAnimation = **null**;

......

mHasLocalTransformation = **false**;

......

mTransformation.clear();

......

**return** **false**;

}

......

}

......

}

这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

WindowState类有几个成员变量是用来描述窗口的动画状态的，其中：

**--mAnimating**，表示窗口是否处于正在显示动画的过程中。

**--mLocalAnimating**，表示窗口的动画是否已经初始化过了。一个动画只有经过初始化之后，才能开始执行。

**--mAnimation**，表示窗口的动画对象。

**--mHasLocalTransformation**，表示窗口的动画是否是一个本地动画，即这个动画是否是来自窗口本身的。有时候一个窗口虽然正在显示动画，但是这个动画有可能是其宿主Activity组件的切换动画。在这种情况下，mHasLocalTransformation的值就会等于false。

**--mTransformation**，表示一个变换矩阵，是根据窗口动画的当前推进状态来计算得到的，用来改变窗口的大小以及位置。

窗口动画的推进只有以下四个条件均满足的情况下才会执行：

(1).屏幕没有被冻结，即WindowManagerService类的成员变量mDisplayFrozen的值等于false；

(2). 屏幕是点亮的，即WindowManagerService类的成员变量mPolicy所指向的一个PhoneWindowManager对象的成员函数isScreenOn的返回值等于true；

(3). 窗口的UI已经绘制完成并且已经提交，即WindowState类的成员变量mDrawPending和mCommitDrawPending的值均等于false；

(4). 窗口已经被设置过动画，即WindowState类的成员变量mAnimation的值不等于null。

一旦满足上述四个条件，那么WindowState类的成员函数stepAnimationLocked就会执行以下操作：

(1). 将成员变量mHasLocalTransformation的值设置为true，表明窗口当前具有一个本地动画。

(2). 检查成员变量mLocalAnimating的值是否等于false。如果等于false的话，那么就说明窗口的动画还没有经过初始化，这时候就会对该动画进行初始化，这是通过调用成员变量mAnimation所指向的一个Animation对象的成员函数initialize和setStartTime来实现的。窗口动画初始化完成之后，还需要将成员变量mLocalAnimating和mAnimating的值均设置为true，表明窗口动画已经初始化过了，并且窗口当前正在执行动画的过程中。

(3). 将成员变量mTransformation所描述的变换矩阵的数据清空。

(4). 调用成员变量mAnimation所指向的一个Animation对象的成员函数getTransformation来计算窗口动画下一步所对应的变换矩阵，并且将这个变换矩阵的数据保存在成员变量mTransformation。

(5). 如果窗口的动画尚未结束显示，那么调用成员变量mAnimation所指向的一个Animation对象的成员函数getTransformation得到的返回值more就会等于true，这时候窗口动画的向前推进操作就完成了。

(6). 如果窗口的动画已经结束显示，那么调用成员变量mAnimation所指向的一个Animation对象的成员函数getTransformation得到的返回值more就会等于false，这时候就需要执行一些清理工作。这些清理工作包括将成员变量mAnimating、mLocalAnimating和mHasLocalTransformation的值设置为false，以及将成员变量mAnimation的值设置为null，还有将成员变量mTransformation所描述的变换矩阵的数据清空。

最后，如果窗口的动画尚未结束显示，那么WindowState类的成员函数stepAnimationLocked会返回一个true值给调用者，否则的话，就会返回一个false值给调用者。

2. Activity组件切换动画的推进过程

从前面第二部分的内容可以知道，Activity组件切换动画的推进过程是由AppWindowToken类的成员函数stepAnimationLocked来实现的，如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**class** **AppWindowToken** **extends** **WindowToken** {

......

**boolean** animating;

Animation animation;

**boolean** hasTransformation;

**final** Transformation transformation = **new** Transformation();

......

// This must be called while inside a transaction.

**boolean** **stepAnimationLocked**(**long** currentTime, **int** dw, **int** dh) {

**if** (!mDisplayFrozen && mPolicy.isScreenOn()) {

// We will run animations as long as the display isn't frozen.

**if** (animation == sDummyAnimation) {

// This guy is going to animate, but not yet. For now count

// it as not animating for purposes of scheduling transactions;

// when it is really time to animate, this will be set to

// a real animation and the next call will execute normally.

**return** **false**;

}

**if** ((allDrawn || animating || startingDisplayed) && animation != **null**) {

**if** (!animating) {

......

animation.initialize(dw, dh, dw, dh);

animation.setStartTime(currentTime);

animating = **true**;

}

transformation.clear();

**final** **boolean** more = animation.getTransformation(

currentTime, transformation);

......

**if** (more) {

// we're done!

hasTransformation = **true**;

**return** **true**;

}

......

animation = **null**;

}

}

......

hasTransformation = **false**;

......

animating = **false**;

......

transformation.clear();

......

**return** **false**;

}

......

}

......

}

这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

AppWindowToken类有几个成员变量是用来描述Activity组件的切换动画状态的，其中：

**--animating**，表示Activity组件的切换动画是否已经初始化过了。

**--animation**，表示Activity组件的切换动画对象。

**--mHasTransformation**，表示Activity组件是否具有切换动画。

**--transformation**，表示一个变换矩阵，是根据Activity组件切换动画的当前推进状态来计算得到的，用来改变Activity组件窗口的大小以及位置。

Activity组件切换动画的推进只有以下五个条件均满足的情况下才会执行：

(1).屏幕没有被冻结，即WindowManagerService类的成员变量mDisplayFrozen的值等于false。

(2). 屏幕是点亮的，即WindowManagerService类的成员变量mPolicy所指向的一个PhoneWindowManager对象的成员函数isScreenOn的返回值等于true。

(3).Activity组件所设置的切换动画不是一个哑动画，即AppWindowToken类的成员变量animation与WindowManagerService类的静态成员变量sDummyAnimation不是指向同一个Animation对象。从前面[Android窗口管理服务WindowManagerService切换Activity窗口（App Transition）的过程分析](http://blog.csdn.net/luoshengyang/article/details/8596449)一文可以知道，一个Activity组件在进行切换之前，用来描述它的一个AppWindowToken对象的成员变量animation的值会被设置为sDummyAnimation，用来表示它准备要执行一个切换动画，但是这个切换动画还没有设置。

(4). Activity组件的所有窗口的UI都已经绘制完成，或者Activity组件的切换动画已经初始化过了，或者Activity组件的启动窗口已经结束显示，即AppWindowToken类的成员变量allDrawn、animating和startingDisplayed中的其中一个的值等于true。

(5). Activity组件已经被设置过切换动画，即AppWindowToken类的成员变量animation的值不等于null。

一旦满足上述五个条件，那么AppWindowToken类的成员函数stepAnimationLocked就会执行以下操作：

(1). 检查成员变量animating的值是否等于false。如果等于false的话，那么就说明Activity组件的切换动画还没有经过初始化，这时候就会对该动画进行初始化，这是通过调用成员变量animation所指向的一个Animation对象的成员函数initialize和setStartTime来实现的。窗口动画初始化完成之后，还需要将成员变量animating的值设置为true，表明窗口动画已经初始化过了。

(2). 将成员变量transformation所描述的变换矩阵的数据清空。

(3). 调用成员变量animation所指向的一个Animation对象的成员函数getTransformation来计算Activity组件切换动画下一步所对应的变换矩阵，并且将这个变换矩阵的数据保存在成员变量transformation。

(4). 如果Activity组件切换动画尚未结束显示，那么调用成员变量animation所指向的一个Animation对象的成员函数getTransformation得到的返回值more就会等于true，这时候Activity组件切换动画的向前推进操作就完成了。

(6). 如果Activity组件切换动画已经结束显示，那么调用成员变量animation所指向的一个Animation对象的成员函数getTransformation得到的返回值more就会等于false，这时候就需要执行一些清理工作。这些清理工作包括将成员变量animating和hasTransformation的值设置为false，以及将成员变量animation的值设置为null，还有将成员变量transformation所描述的变换矩阵的数据清空。

最后，如果窗口的动画尚未结束显示，那么AppWindowToken类的成员函数stepAnimationLocked就会将成员变量hasTransformation的值设置为true，并且返回一个true值给调用者，否则的话，就会返回一个false值给调用者。

四.窗口动画的合成过程

从前面第二部分的内容可以知道，WindowState类的成员函数computeShownFrameLocked负责合成窗口的动画，包括窗口本身所设置的进入（退出）动画、从被附加窗口传递过来的动画，以及宿主Activity组件传递过来的切换动画。窗口的这三个动画合成之后，就可以得到一个变换矩阵。将这个变换矩阵应用到窗口的原始大小和位置上去，就可以得到窗口经过动画变换后所得到的位置和大小。

WindowState类的成员函数computeShownFrameLocked的实现如下所示：

**public** **class** **WindowManagerService** **extends** **IWindowManager**.**Stub**

**implements** **Watchdog**.**Monitor** {

......

**private** **final** **class** **WindowState** **implements** **WindowManagerPolicy**.**WindowState** {

......

// Actual frame shown on-screen (may be modified by animation)

**final** Rect mShownFrame = **new** Rect();

......

// Current transformation being applied.

**float** mDsDx=1, mDtDx=0, mDsDy=0, mDtDy=1;

......

// "Real" frame that the application sees.

**final** Rect mFrame = **new** Rect();

......

**void** **computeShownFrameLocked**() {

**final** **boolean** selfTransformation = mHasLocalTransformation;

Transformation attachedTransformation =

(mAttachedWindow != **null** && mAttachedWindow.mHasLocalTransformation)

? mAttachedWindow.mTransformation : **null**;

Transformation appTransformation =

(mAppToken != **null** && mAppToken.hasTransformation)

? mAppToken.transformation : **null**;

// Wallpapers are animated based on the "real" window they

// are currently targeting.

**if** (mAttrs.type == TYPE\_WALLPAPER && mLowerWallpaperTarget == **null**

&& mWallpaperTarget != **null**) {

**if** (mWallpaperTarget.mHasLocalTransformation &&

mWallpaperTarget.mAnimation != **null** &&

!mWallpaperTarget.mAnimation.getDetachWallpaper()) {

attachedTransformation = mWallpaperTarget.mTransformation;

......

}

**if** (mWallpaperTarget.mAppToken != **null** &&

mWallpaperTarget.mAppToken.hasTransformation &&

mWallpaperTarget.mAppToken.animation != **null** &&

!mWallpaperTarget.mAppToken.animation.getDetachWallpaper()) {

appTransformation = mWallpaperTarget.mAppToken.transformation;

......

}

}

**if** (selfTransformation || attachedTransformation != **null**

|| appTransformation != **null**) {

// cache often used attributes locally

**final** Rect frame = mFrame;

**final** **float** tmpFloats[] = mTmpFloats;

**final** Matrix tmpMatrix = mTmpMatrix;

// Compute the desired transformation.

tmpMatrix.setTranslate(0, 0);

**if** (selfTransformation) {

tmpMatrix.postConcat(mTransformation.getMatrix());

}

tmpMatrix.postTranslate(frame.left, frame.top);

**if** (attachedTransformation != **null**) {

tmpMatrix.postConcat(attachedTransformation.getMatrix());

}

**if** (appTransformation != **null**) {

tmpMatrix.postConcat(appTransformation.getMatrix());

}

// "convert" it into SurfaceFlinger's format

// (a 2x2 matrix + an offset)

// Here we must not transform the position of the surface

// since it is already included in the transformation.

//Slog.i(TAG, "Transform: " + matrix);

tmpMatrix.getValues(tmpFloats);

mDsDx = tmpFloats[Matrix.MSCALE\_X];

mDtDx = tmpFloats[Matrix.MSKEW\_X];

mDsDy = tmpFloats[Matrix.MSKEW\_Y];

mDtDy = tmpFloats[Matrix.MSCALE\_Y];

**int** x = (**int**)tmpFloats[Matrix.MTRANS\_X] + mXOffset;

**int** y = (**int**)tmpFloats[Matrix.MTRANS\_Y] + mYOffset;

**int** w = frame.width();

**int** h = frame.height();

mShownFrame.set(x, y, x+w, y+h);

......

**return**;

}

mShownFrame.set(mFrame);

**if** (mXOffset != 0 || mYOffset != 0) {

mShownFrame.offset(mXOffset, mYOffset);

}

......

mDsDx = 1;

mDtDx = 0;

mDsDy = 0;

mDtDy = 1;

}

......

}

......

}

这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

在分析WindowState类的成员函数computeShownFrameLocked的实现之前，我们先来看几个相关的成员变量：

**--mFrame**，用来描述窗口的实际位置以及大小。它们的计算过程可以参考前面[Android窗口管理服务WindowManagerService计算Activity窗口大小的过程分析](http://blog.csdn.net/luoshengyang/article/details/8479101)一文。

**--mShownFrame**，用来描述窗口当前所要显示的位置以及大小。

**--mDsDx、mDtDx、mDsDy、mDtDy**，用来描述窗口的变换矩阵（二维）。

WindowState类的成员函数computeShownFrameLocked的目标就根据窗口的实际位置、大小，以及窗口的动画，来计算得到窗口当前所要显示的位置以及大小。

注意，在调用WindowState类的成员函数computeShownFrameLocked之前，窗口的实际位置和大小是已经计算好了的，并且窗口的动画也是已经向前推进好了的。窗口的实际位置和大小的计算过程可以参考前面[Android窗口管理服务WindowManagerService计算Activity窗口大小的过程分析](http://blog.csdn.net/luoshengyang/article/details/8479101)一文，而窗口动画向前推进的过程可以参考前面第三部分的内容。

WindowState类的成员函数computeShownFrameLocked首先检查当前正在处理的窗口有多少个动画是需要合成的，即：

1. 检查成员变量mHasLocalTransformation的值是否等于true。如果等于true的话，那么就会将变量selfTransformation的值也设置为true，表示窗口本身设置有一个动画。这个动画要么是一个打开窗口类型的动画，要么是一个关闭窗口类型的动画。

2. 检查成员变量mAttachedWindow的值是否等于null。如果不等于null的话，那么就说明当前正在处理的窗口是附加在另外一个窗口之上。如果这个被附加的窗口也设置有动画，那么成员变量mAttachedWindow所指向的一个WindowState对象的成员变量mHasLocalTransformation的值就会等于true，这时候用来描述被附加窗口当前所要执行的动画的一个变换矩阵就由该WindowState对象的成员变量mTransformation来描述。这个变换矩阵最终会被保存在变量attachedTransformation中。

3.检查成员变量mAppToken的值是否等于null。如果不等于null的话，那么就说明当前正在处理的窗口是一个Activity组件的窗口。如果这个Activity组件设置有切换动画，那么成员变量mAppToken所指向的一个AppWindowToken对象的成员变量hasTransformation的值就会等于true，这时候用来描述该Activity组件当前有所要执行的切换动画的一个变换矩阵就由该AppWindowToken对象的成员变量transformation来描述。这个变换矩阵最终会被保存在变量appTransformation中。

WindowState类的成员函数computeShownFrameLocked接着检查当前正在处理的窗口是否是一个壁纸窗口，即检查成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量type的值的TYPE\_WALLPAPER位是否等于1。如果当前正在处理的窗口是一个壁纸窗口，并且它当前有且仅有一个目标窗口，即WindowManagerService类的成员变量mWallpaperTarget和mLowerWallpaperTarget的值分别不等于null和等于null。在这种情况下，需要对壁纸窗口的动画进行特殊处理，即：

1. 要把壁纸窗口所附加在的窗口的动画设置为壁纸窗口的目标窗口所附加在的窗口的动画，即将变量attachedTransformation指向用来描述壁纸窗口的目标窗口所附加在的窗口当前所要执行的动画的一个变换矩阵，前提是壁纸窗口的目标窗口设置有动画，并且这个目标窗口在结束动画过程后不会与壁纸窗口分离。

2. 要把壁纸窗口的切换动画设置为壁纸窗口的目标窗口的切换动画，即将变量appTransformation指向用来描述壁纸窗口的目标窗口当前所要执行的切换动画的一个变换矩阵，前提是壁纸窗口的目标窗口设置有切换动画，并且这个目标窗口结束动画过程后不会与壁纸窗口分离。

通过上面的处理之后，如果变量selfTransformation的值等于true，或者变量attachedTransformation和appTransformation的值不等于null，那么就说明当前正在处理的窗口有动画需要显示，因此，接下来就要将这些动画组合成一个总的变换矩阵。这个总的变换矩阵就是通过WindowState类的成员变量mTmpMatrix来描述的，它是通过下面的步骤来获得的：

1. 将该矩阵的偏移位置初始化为(0, 0)，这是通过调用变量tmpMatrix所描述的一个Matrix对象的成员函数setTranslate来实现的。

2. 如果变量selfTransformation的值等于true，那么就说明当前正在处理的窗口本身设置有动画。这个动画是通过WindowState类的成员变量mTransformation来描述的，调用这个成员变量所指向的一个Transformation对象的成员函数getMatrix就可以获得一个对应的变换矩阵。将这个变换矩阵与变量tmpMatrix所描述的变换矩阵相乘，就可以得到一个中间变换矩阵，这是通过调用变量tmpMatrix所指向的一个Matrix对象的成员函数postConcat来实现的。

3. 将上面第2步得到的中间变换矩阵的偏移位置设置为当前正在处理的窗口的实际位置，这是通过调用变量tmpMatrix所描述的一个Matrix对象的成员函数postTranslate来实现的，而当前正在处理的窗口的实际位置由变量frame所指向的一个Frame对象的成员变量left和top来描述。注意，变量frame和WindowState类的成员变量mFrame指向的是同一个Frame对象，因此它的成员变量left和top描述的是正在处理的窗口的实际位置。

4. 如果变量attachedTransformation的值不等于null，那么就说明当前正在处理的窗口所附加在的窗口设置有动画。这个动画就是通过变量attachedTransformation所指向的一个Transformation对象来描述的，调用这个Transformation对象的成员函数getMatrix就可以获得一个对应的变换矩阵。将这个变换矩阵与变量tmpMatrix所描述的变换矩阵相乘，就可以得到一个中间变换矩阵，这是通过调用变量tmpMatrix所指向的一个Matrix对象的成员函数postConcat来实现的。

5.如果变量attachedTransformation的值不等于null，那么就说明当前正处理的窗口的宿主Activity组件设置有切换动画。这个切换动画就是通过变量appTransformation所指向的一个Transformation对象来描述的，调用这个Transformation对象的成员函数getMatrix就可以获得一个对应的变换矩阵。将这个变换矩阵与变量tmpMatrix所描述的变换矩阵相乘，就可以得到一个中间变换矩阵，这是通过调用变量tmpMatrix所指向的一个Matrix对象的成员函数postConcat来实现的。

经过上面的5个步骤之后，最终得到的变换矩阵就保存在变量tmpMatrix中。由于这个变换矩阵是要设置到SurfaceFlinger服务中去的，因此就需要将这个变换矩阵转换为SurfaceFlinger服务所要求的格式。SurfaceFlinger服务所要求的变换矩阵的格式是由窗口在X轴和Y轴上的切变值以及缩放值来表示的，它们可以按照以下的步骤来获得：

1. 调用变量tmpMatrix所描述的一个Matrix对象的成员函数getValues来获得一个数组，并且保存在变量tmpFloats中。

2. 数组tmpFloats的第Matrix.MSKEW\_X和Matrix.MSKEW\_Y个位置的值就分别表示窗口在X轴和Y轴上的切变值，分别保存在WindowState类的成员变量mDtDx和mDsDy中。

3.数组tmpFloats的第Matrix.MSCALE\_X和Matrix.MSCALE\_Y个位置的值就分别表示窗口在X轴和Y轴上的缩放值，分别保存在WindowState类的成员变量mDsDx和mDtDy中。

获得了当前正在处理的窗口的变换矩阵之后，接下来还要计算当前正在处理的窗口接下来要显示的位置以及大小。

前面得到的数组tmpFloats中的第Matrix.MTRANS\_X和Matrix.MTRANS\_Y个位置的值分别表示窗口在X轴和Y轴上的偏移值，它们实际就是窗口接下来要显示的位置。从前面[Android窗口管理服务WindowManagerService对壁纸窗口（Wallpaper Window）的管理分析](http://blog.csdn.net/luoshengyang/article/details/8550820)一文可以知道，如果当前正在处理的是一个壁纸窗口，那么WindowState类的成员变量mXOffset和mYOffset描述的就是壁纸窗口相对其目标窗口的偏移值。对于普通的窗口来说，WindowState类的成员变量mXOffset和mYOffset的值等于0。因此，需要将保在在数组tmpFloats中的第Matrix.MTRANS\_X和Matrix.MTRANS\_Y个位置的值与WindowState类的成员变量mXOffset和mYOffset的值分别相加，才能得到当前正在处理接下来要显示在的位置。

由于前面获得的变换矩阵已经包含了当前正在处理的窗口的大小缩放因子，因此，我们就将当前正在处理的窗口的大小设置为它的实际大小即可。通过调用变量frame所指向的一个Frame对象的成员函数width和height可以获得当前正在处理的窗口的实际大小。

经过上面两步之后，当前正在处理的窗口接下来要显示的位置以及大小就计算完成了，其中，位置值保存在变量x和y中，而大小值保存在变量w和h，最终就可以将它们保存在WindowState类的成员变量mShownFrame所指向的一个Frame对象中。

如果当前正在处理的窗口没有动画可以显示，即变量selfTransformation的值等于false，并且变量attachedTransformation和appTransformation的值均等于null，那么WindowState类的成员函数computeShownFrameLocked的实现就简单了，它只要简单地将成员变量mFrame的内容设置到成员变量mShownFrame中，并且将成员变量mDsDx、mDtDx、mDsDy和mDtDy分别设置为1、0、0和1即可，表示当前正在处理的窗口既没有切变变换，也没有缩放变换。另外，如果WindowState类的成员变量mXOffset或者mYOffset的值不等于0，那么就需要将它们作来偏移值设置到成员变量mShownFrame所描述的一个Frame对象去，以便可以正确地计算出当前正在处理的窗口的位置。

至此，我们就分析完成窗口动画的显示过程了，整个WindowManagerService服务的分析也到此结束了。WindowManagerService服务可以说是整个Android应用程序框架层最为复杂的模块了，它与SurfaceFlinger服务一起为整个Android系统提供了UI服务，理解它对理解Android系统有着重要的意义。不过，要理解WindowManagerService服务的实现，是必须下些功夫的，同时也希望这个系列的文章能够帮助到大家。重新学习WindowManagerService服务请参考[Android窗口管理服务WindowManagerService的简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/8462738)一文。

**老罗的新浪微博：**[**http://weibo.com/shengyangluo**](http://weibo.com/shengyangluo)**，欢迎关注！**

推荐：[Android窗口管理服务WindowManagerService对窗口的组织方式分析](http://www.itboth.com/d/ZFJFzu/android-windowmanagerservice)

[我们知道，在Android系统中，Activity是以堆栈的形式组织在ActivityManagerService服务中的。与Activity类似，Android系统中的窗口也是以堆栈的形式组织在WindowManagerSe

# Android視圖SurfaceView的實現原理分析

 04-17

在Android系統中，有一種特殊的視圖，稱為SurfaceView，它擁有獨立的繪圖表面，即它不與其宿主窗口共享同一個繪圖表面。由於擁有獨立的繪圖表面，因此SurfaceView的UI就可以在一個獨立的線程中進行繪製。又由於不會佔用主線程資源，SurfaceView一方面可以實現複雜而高效的UI，另一方面又不會導致用戶輸入得不到及時響應。在本文中，我們就詳細分析SurfaceView的實現原理。

為了接下來可以方便地描述SurfaceView的實現原理分析，我們假設在一個Activity窗口的視圖結構中，除了有一個DecorView頂層視圖之外，還有兩個TextView控制項，以及一個SurfaceView視圖，這樣該Activity窗口在SurfaceFlinger服務中就對應有兩個Layer或者一個Layer的一個LayerBuffer，如圖1所示：

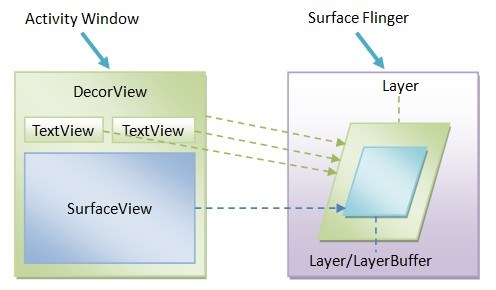


圖1 SurfaceView及其宿主Activity窗口的繪圖表面示意圖

在圖1中，Activity窗口的頂層視圖DecorView及其兩個TextView控制項的UI都是繪製在SurfaceFlinger服務中的同一個Layer上面的，而SurfaceView的UI是繪製在SurfaceFlinger服務中的另外一個Layer或者LayerBuffer上的。

注意，用來描述SurfaceView的Layer或者LayerBuffer的Z軸位置是小於用來其宿主Activity窗口的Layer的Z軸位置的，但是前者會在後者的上面挖一個「洞」出來，以便它的UI可以對用戶可見。實際上，SurfaceView在其宿主Activity窗口上所挖的「洞」只不過是在其宿主Activity窗口上設置了一塊透明區域。

從總體上描述了SurfaceView的大致實現原理之後，接下來我們就詳細分析它的具體實現過程，包括它的繪圖表面的創建過程、在宿主窗口上面進行挖洞的過程，以及繪製過程。

1. SurfaceView的繪圖表面的創建過程

由於SurfaceView具有獨立的繪圖表面，因此，在它的UI內容可以繪製之前，我們首先要將它的繪圖表面創建出來。儘管SurfaceView不與它的宿主窗口共享同一個繪圖表面，但是它仍然是屬於宿主窗口的視圖結構的一個結點的，也就是說，SurfaceView仍然是會參與到宿主窗口的某些執行流程中去。

每當一個窗口需要刷新UI時，就會調用ViewRoot類的成員函數performTraversals。ViewRoot類的成員函數performTraversals在執行的過程中，如果發現當前窗口的繪圖表面還沒有創建，或者發現當前窗口的繪圖表面已經失效了，那麼就會請求WindowManagerService服務創建一個新的繪圖表面，同時，它還會通過一系列的回調函數來讓嵌入在窗口裡面的SurfaceView有機會創建自己的繪圖表面。

接下來，我們就從ViewRoot類的成員函數performTraversals開始，分析SurfaceView的繪圖表面的創建過程，如圖2所示：

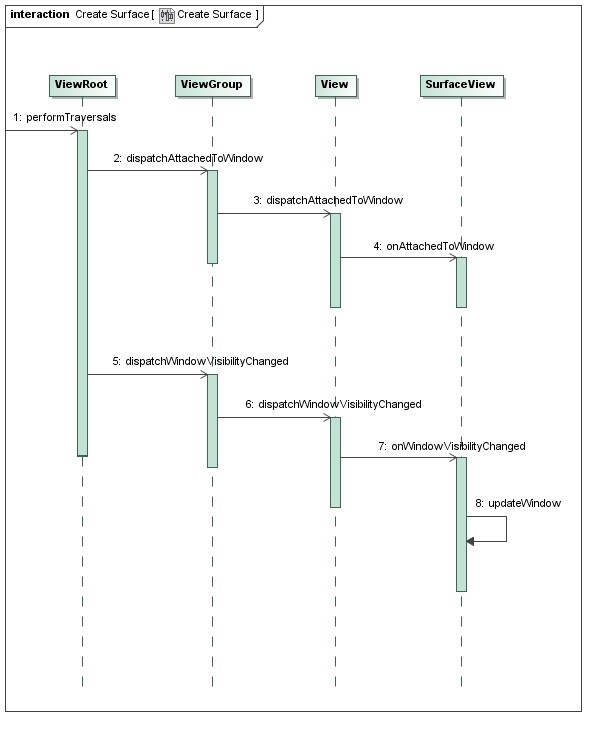
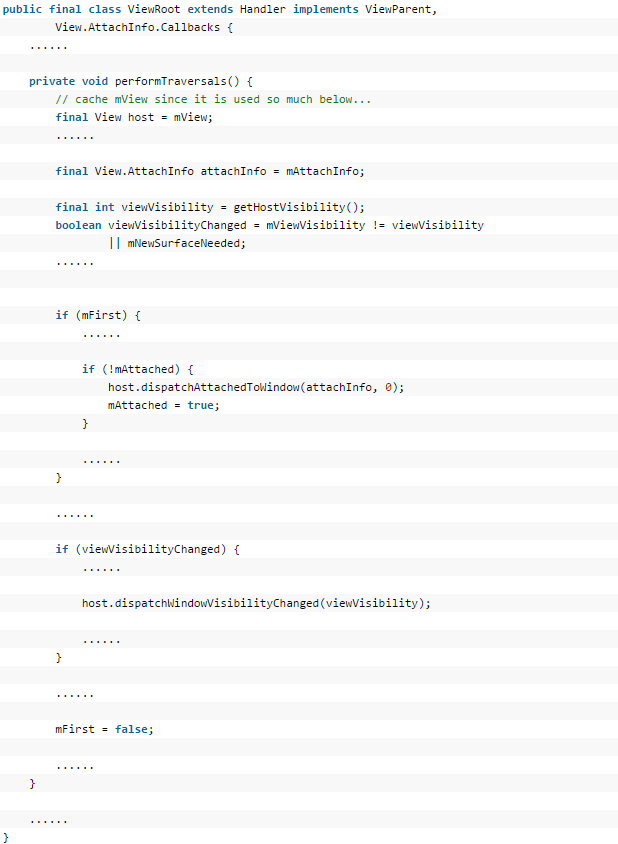


圖2 SurfaceView的繪圖表面的創建過程

這個過程可以分為8個步驟，接下來我們就詳細分析每一個步驟。

Step 1. ViewRoot.performTraversals



這個函數定義在文件frameworks/base/core/Java/android/view/ViewRoot.java中。

我們首先分析在ViewRoot類的成員函數performTraversals中四個相關的變數host、attachInfo、viewVisibility和viewVisibilityChanged。

變數host與ViewRoot類的成員變數mView指向的是同一個DecorView對象，這個DecorView對象描述的當前窗口的頂層視圖。

變數attachInfo與ViewRoot類的成員變數mAttachInfo指向的是同一個AttachInfo對象。在Android系統中，每一個視圖附加到它的宿主窗口的時候，都會獲得一個AttachInfo對象，用來描述被附加的窗口的信息。

變數viewVisibility描述的是當前窗口的可見性。

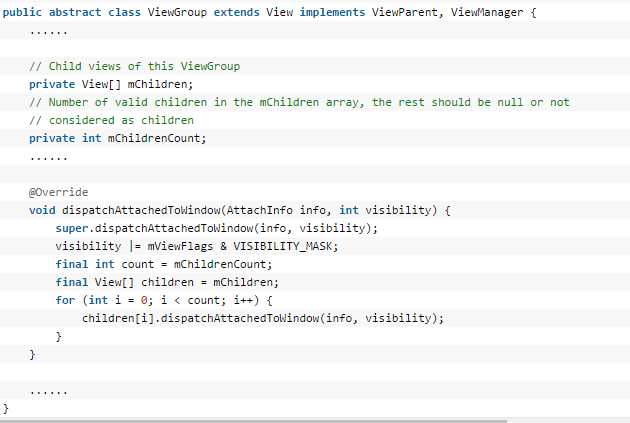
變數viewVisibilityChanged描述的是當前窗口的可見性是否發生了變化。

ViewRoot類的成員變數mFirst表示當前窗口是否是第一次被刷新UI。如果是的話，那麼它的值就會等於true，說明當前窗口的繪圖表面還未創建。在這種情況下，如果ViewRoot類的另外一個成員變數mAttached的值也等於true，那麼就表示當前窗口還沒有將它的各個子視圖附加到它的上面來。這時候ViewRoot類的成員函數performTraversals就會從當前窗口的頂層視圖開始，通知每一個子視圖它要被附加到宿主窗口上去了，這是通過調用變數host所指向的一個DecorView對象的成員函數dispatchAttachedToWindow來實現的。DecorView類的成員函數dispatchAttachedToWindow是從父類ViewGroup繼承下來的，在後面的Step 2中，我們再詳細分析ViewGroup類的成員數dispatchAttachedToWindow的實現。

接下來， ViewRoot類的成員函數performTraversals判斷當前窗口的可見性是否發生了變化，即檢查變數viewVisibilityChanged的值是否等於true。如果發生了變化，那麼就會從當前窗口的頂層視圖開始，通知每一個子視圖它的宿主窗口的可見發生變化了，這是通過調用變數host所指向的一個DecorView對象的成員函數dispatchWindowVisibilityChanged來實現的。DecorView類的成員函數dispatchWindowVisibilityChanged是從父類ViewGroup繼承下來的，在後面的Step 5中，我們再詳細分析ViewGroup類的成員數dispatchWindowVisibilityChanged的實現。

我們假設當前窗口有一個SurfaceView，那麼當該SurfaceView接收到它被附加到宿主窗口以及它的宿主窗口的可見性發生變化的通知時，就會相應地將自己的繪圖表面創建出來。接下來，我們就分別分析ViewGroup類的成員數dispatchAttachedToWindow和dispatchWindowVisibilityChanged的實現，以便可以了解SurfaceView的繪圖表面的創建過程。

Step 2. ViewGroup.dispatchAttachedToWindow



這個函數定義在文件frameworks/base/core/java/android/view/ViewGroup.java中。

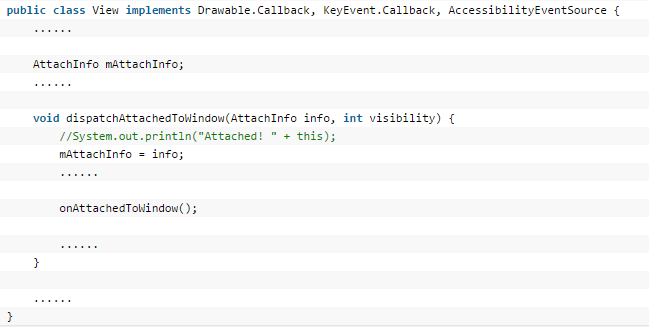
ViewGroup類的成員變數mChildren保存的是當前正在處理的視圖容器的子視圖，而另外一個成員變數mChildrenCount保存的是這些子視圖的數量。

ViewGroup類的成員函數dispatchAttachedToWindow的實現很簡單，它只是簡單地調用當前正在處理的視圖容器的每一個子視圖的成員函數dispatchAttachedToWindow，以便可以通知這些子視圖，它們被附加到宿主窗口上去了。

當前正在處理的視圖容器即為當前正在處理的窗口的頂層視圖，由於前面我們當前正在處理的窗口有一個SurfaceView，因此這一步就會調用到該SurfaceView的成員函數dispatchAttachedToWindow。

由於SurfaceView類的成員函數dispatchAttachedToWindow是從父類View繼承下來的，因此，接下來我們就繼續分析View類的成員函數dispatchAttachedToWindow的實現。

Step 3. View.dispatchAttachedToWindow

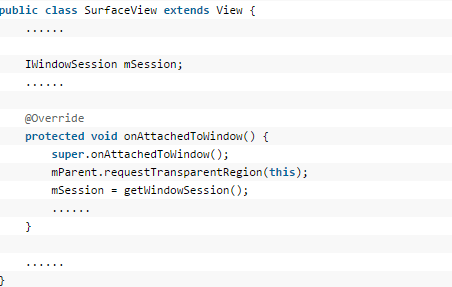


這個函數定義在文件frameworks/base/core/java/android/view/View.java中。

View類的成員函數dispatchAttachedToWindow首先將參數info所指向的一個AttachInfo對象保存在自己的成員變數mAttachInfo中，以便當前視圖可以獲得其所附加在的窗口的相關信息，接下來再調用另外一個成員函數onAttachedToWindow來讓子類有機會處理它被附加到宿主窗口的事件。

前面我們已經假設了當前處理的是一個SurfaceView。SurfaceView類重寫了父類View的成員函數onAttachedToWindow，接下來我們就繼續分析SurfaceView的成員函數onAttachedToWindow的實現，以便可以了解SurfaceView的繪圖表面的創建過程。

Step 4. SurfaceView.onAttachedToWindow



這個函數定義在文件frameworks/base/core/java/android/view/SurfaceView.java中。

SurfaceView類的成員函數onAttachedToWindow做了兩件重要的事。

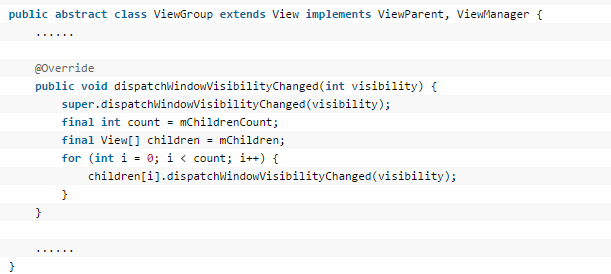
第一件事情是通知父視圖，當前正在處理的SurfaceView需要在宿主窗口的繪圖表面上挖一個洞，即需要在宿主窗口的繪圖表面上設置一塊透明區域。當前正在處理的SurfaceView的父視圖保存在父類View的成員變數mParent中，通過調用這個成員變數mParent所指向的一個ViewGroup對象的成員函數requestTransparentRegion，就可以通知到當前正在處理的SurfaceView的父視圖，當前正在處理的SurfaceView需要在宿主窗口的繪圖表面上設置一塊透明區域。在後面第2部分的內容中，我們再詳細分析SurfaceView在宿主窗口的繪圖表面的挖洞過程。

第二件事情是調用從父類View繼承下來的成員函數getWindowSession來獲得一個實現了IWindowSession介面的Binder代理對象，並且將該Binder代理對象保存在SurfaceView類的成員變數mSession中。在Android系統中，每一個應用程序進程都有一個實現了IWindowSession介面的Binder代理對象，這個Binder代理對象是用來與WindowManagerService服務進行通信的，View類的成員函數getWindowSession返回的就是該Binder代理對象。在接下來的Step 8中，我們就可以看到，SurfaceView就可以通過這個實現了IWindowSession介面的Binder代理對象來請求WindowManagerService服務為自己創建繪圖表面的。

這一步執行完成之後，返回到前面的Step 1中，即ViewRoot類的成員函數performTraversals中，我們假設當前窗口的可見性發生了變化，那麼接下來就會調用頂層視圖的成員函數dispatchWindowVisibilityChanged，以便可以通知各個子視圖，它的宿主窗口的可見性發生化了。

窗口的頂層視圖是使用DecorView類來描述的，而DecroView類的成員函數dispatchWindowVisibilityChanged是從父類ViewGroup類繼承下來的，因此，接下來我們就繼續分析GroupView類的成員函數dispatchWindowVisibilityChanged的實現，以便可以了解包含在當前窗口裡面的一個SurfaceView的繪圖表面的創建過程。

Step 5. ViewGroup.dispatchWindowVisibilityChanged



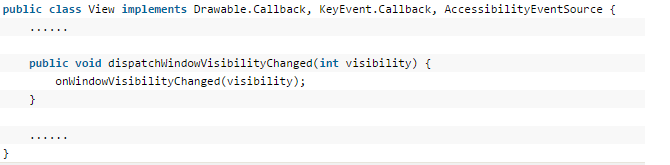
這個函數定義在文件frameworks/base/core/java/android/view/ViewGroup.java中。

ViewGroup類的成員函數dispatchWindowVisibilityChanged的實現很簡單，它只是簡單地調用當前正在處理的視圖容器的每一個子視圖的成員函數dispatchWindowVisibilityChanged，以便可以通知這些子視圖，它們所附加在的宿主窗口的可見性發生變化了。

當前正在處理的視圖容器即為當前正在處理的窗口的頂層視圖，由於前面我們當前正在處理的窗口有一個SurfaceView，因此這一步就會調用到該SurfaceView的成員函數dispatchWindowVisibilityChanged。

由於SurfaceView類的成員函數dispatchWindowVisibilityChanged是從父類View繼承下來的，因此，接下來我們就繼續分析View類的成員函數dispatchWindowVisibilityChanged的實現。

Step 6. View.dispatchWindowVisibilityChanged

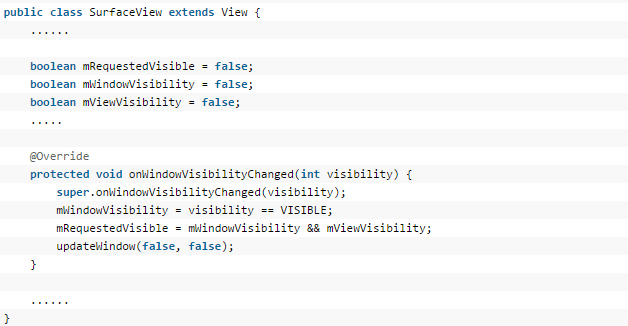


這個函數定義在文件frameworks/base/core/java/android/view/View.java中。

View類的成員函數dispatchWindowVisibilityChanged的實現很簡單，它只是調用另外一個成員函數onWindowVisibilityChanged來讓子類有機會處理它所附加在的宿主窗口的可見性變化事件。

前面我們已經假設了當前處理的是一個SurfaceView。SurfaceView類重寫了父類View的成員函數onWindowVisibilityChanged，接下來我們就繼續分析SurfaceView的成員函數onWindowVisibilityChanged的實現，以便可以了解SurfaceView的繪圖表面的創建過程。

Step 7. SurfaceView.onWindowVisibilityChanged



這個函數定義在文件frameworks/base/core/java/android/view/SurfaceView.java中。

SurfaceView類有三個用來描述可見性的成員變數mRequestedVisible、mWindowVisibility和mViewVisibility。其中，mWindowVisibility表示SurfaceView的宿主窗口的可見性，mViewVisibility表示SurfaceView自身的可見性。只有當mWindowVisibility和mViewVisibility的值均等於true的時候，mRequestedVisible的值才為true，表示SurfaceView是可見的。

參數visibility描述的便是當前正在處理的SurfaceView的宿主窗口的可見性，因此，SurfaceView類的成員函數onWindowVisibilityChanged首先將它記錄在成員變數mWindowVisibility，接著再綜合另外一個成員變數mViewVisibility來判斷當前正在處理的SurfaceView是否是可見的，並且記錄在成員變數mRequestedVisible中。

最後，SurfaceView類的成員函數onWindowVisibilityChanged就會調用另外一個成員函數updateWindow來更新當前正在處理的SurfaceView。在更新的過程中，如果發現當前正在處理的SurfaceView還沒有創建繪圖表面，那麼就地請求WindowManagerService服務為它創建一個。

接下來，我們就繼續分析SurfaceView類的成員函數updateWindow的實現，以便可以了解SurfaceView的繪圖表面的創建過程。

Step 8. SurfaceView.updateWindow

publicclassSurfaceViewextendsView {

......

finalSurface mSurface =newSurface();

......

MyWindow mWindow;

.....

intmWindowType = WindowManager.LayoutParams.TYPE\_APPLICATION\_MEDIA;

......

intmRequestedType = -1;

......

privatevoidupdateWindow(booleanforce,booleanredrawNeeded) {

if(!mHaveFrame) {

return;

}

......

intmyWidth = mRequestedWidth;

if(myWidth <=0) myWidth = getWidth();

intmyHeight = mRequestedHeight;

if(myHeight <=0) myHeight = getHeight();

getLocationInWindow(mLocation);

finalbooleancreating = mWindow ==null;

finalbooleanformatChanged = mFormat != mRequestedFormat;

finalbooleansizeChanged = mWidth != myWidth || mHeight != myHeight;

finalbooleanvisibleChanged = mVisible != mRequestedVisible

|| mNewSurfaceNeeded;

finalbooleantypeChanged = mType != mRequestedType;

if(force || creating || formatChanged || sizeChanged || visibleChanged

|| typeChanged || mLeft != mLocation[0] || mTop != mLocation[1]

|| mUpdateWindowNeeded || mReportDrawNeeded || redrawNeeded) {

......

try{

finalbooleanvisible = mVisible = mRequestedVisible;

mLeft = mLocation[0];

mTop = mLocation[1];

mWidth = myWidth;

mHeight = myHeight;

mFormat = mRequestedFormat;

mType = mRequestedType;

......

// Places the window relative

mLayout.x = mLeft;

mLayout.y = mTop;

mLayout.width = getWidth();

mLayout.height = getHeight();

......

mLayout.memoryType = mRequestedType;

if(mWindow ==null) {

mWindow =newMyWindow(this);

mLayout.type = mWindowType;

......

mSession.addWithoutInputChannel(mWindow, mLayout,

mVisible ? VISIBLE : GONE, mContentInsets);

}

......

mSurfaceLock.lock();

try{

......

finalintrelayoutResult = mSession.relayout(

mWindow, mLayout, mWidth, mHeight,

visible ? VISIBLE : GONE,false, mWinFrame, mContentInsets,

mVisibleInsets, mConfiguration, mSurface);

......

}finally{

mSurfaceLock.unlock();

}

......

}catch(RemoteException ex) {

}

.....

}

}

......

}

這個函數定義在文件frameworks/base/core/java/android/view/SurfaceView.java中。

在分析SurfaceView類的成員函數updateWindow的實現之前，我們首先介紹一些相關的成員變數的含義，其中，mSurface、mWindow、mWindowType和mRequestedType這四個成員變數是最重要的。

SurfaceView類的成員變數mSurface指向的是一個Surface對象，這個Surface對象描述的便是SurfaceView專有的繪圖表面。對於一般的視圖來說，例如，TextView或者Button，它們是沒有專有的繪圖表面的，而是與專宿主窗口共享同一個繪圖表面，因此，它們就不會像SurfaceView一樣，有一個專門的類型為Surface的成員變數來描述自己的繪圖表面。每一個Activity窗口都關聯有一個W對象。這個W對象是一個實現了IWindow介面的Binder本地對象，它是用來傳遞給WindowManagerService服務的，以便WindowManagerService服務可以通過它來和它所關聯的Activity窗口通信。例如，WindowManagerService服務通過這個W對象來通知它所關聯的Activity窗口的大小或者可見性發生變化了。同時，這個W對象還用來在WindowManagerService服務這一側唯一地標誌一個窗口，也就是說，WindowManagerService服務會為這個W對象創建一個WindowState對象。

SurfaceView類的成員變數mWindow指向的是一個MyWindow對象。MyWindow類是從BaseIWindow類繼承下來的，後者與W類一樣，實現了IWindow介面。也就是說，每一個SurfaceView都關聯有一個實現了IWindow介面的Binder本地對象，就如第一個Activity窗口都關聯有一個實現了IWindow介面的W對象一樣。從這裡我們就可以推斷出，每一個SurfaceView在WindowManagerService服務這一側都對應有一個WindowState對象。從這一點來看，WindowManagerService服務認為Activity窗口和SurfaceView的地位是一樣的，即認為它們都是一個窗口，並且具有繪圖表面。接下來我們就會通過SurfaceView類的成員函數updateWindow的實現來證實這個推斷。

SurfaceView類的成員變數mWindowType描述的是SurfaceView的窗口類型，它的默認值等於TYPE\_APPLICATION\_MEDIA。也就是說，我們在創建一個SurfaceView的時候，默認是用來顯示多媒體的，例如，用來顯示視頻。SurfaceView還有另外一個窗口類型TYPE\_APPLICATION\_MEDIA\_OVERLAY，它是用來在視頻上面顯示一個Overlay的，這個Overlay可以用來顯示視字幕等信息。

我們假設一個Activity窗口嵌入有兩個SurfaceView，其中一個SurfaceView的窗口類型為TYPE\_APPLICATION\_MEDIA，另外一個SurfaceView的窗口類型為TYPE\_APPLICATION\_MEDIA\_OVERLAY，那麼在WindowManagerService服務這一側就會對應有三個WindowState對象，其中，用來描述SurfaceView的WindowState對象是附加在用來描述Activity窗口的WindowState對象上的。如果一個WindowState對象所描述的窗口的類型為TYPE\_APPLICATION\_MEDIA或者TYPE\_APPLICATION\_MEDIA\_OVERLAY，那麼它就會位於它所附加在的窗口的下面。也就是說，類型為TYPE\_APPLICATION\_MEDIA或者TYPE\_APPLICATION\_MEDIA\_OVERLAY的窗口的Z軸位置是小於它所附加在的窗口的Z軸位置的。同時，如果一個窗口同時附加有類型為TYPE\_APPLICATION\_MEDIA和TYPE\_APPLICATION\_MEDIA\_OVERLAY的兩個窗口，那麼類型為TYPE\_APPLICATION\_MEDIA\_OVERLAY的窗口的Z軸大於類型為TYPE\_APPLICATION\_MEDIA的窗口的Z軸位置。

從上面的描述就可以得出一個結論：如果一個Activity窗口嵌入有兩個類型分別為TYPE\_APPLICATION\_MEDIA和TYPE\_APPLICATION\_MEDIA\_OVERLAY的SurfaceView，那麼該Activity窗口的Z軸位置大於類型為TYPE\_APPLICATION\_MEDIA\_OVERLAY的SurfaceView的Z軸位置，而類型為TYPE\_APPLICATION\_MEDIA\_OVERLAY的SurfaceView的Z軸位置又大於類型為TYPE\_APPLICATION\_MEDIA的窗口的Z軸位置。

注意，我們在創建了一個SurfaceView之後，可以調用它的成員函數setZOrderMediaOverlay、setZOrderOnTop或者setWindowType來修改該SurfaceView的窗口類型，也就是修改該SurfaceView的成員變數mWindowType的值。

SurfaceView類的成員變數mRequestedType描述的是SurfaceView的繪圖表面的類型，一般來說，它的值可能等於SURFACE\_TYPE\_NORMAL，也可能等於SURFACE\_TYPE\_PUSH\_BUFFERS。

當一個SurfaceView的繪圖表面的類型等於SURFACE\_TYPE\_NORMAL的時候，就表示該SurfaceView的繪圖表面所使用的內存是一塊普通的內存。一般來說，這塊內存是由SurfaceFlinger服務來分配的，我們可以在應用程序內部自由地訪問它，即可以在它上面填充任意的UI數據，然後交給SurfaceFlinger服務來合成，並且顯示在屏幕上。在這種情況下，SurfaceFlinger服務使用一個Layer對象來描述該SurfaceView的繪圖表面。

當一個SurfaceView的繪圖表面的類型等於SURFACE\_TYPE\_PUSH\_BUFFERS的時候，就表示該SurfaceView的繪圖表面所使用的內存不是由SurfaceFlinger服務分配的，因而我們不能夠在應用程序內部對它進行操作。例如，當一個SurfaceView是用來顯示攝像頭預覽或者視頻播放的時候，我們就會將它的繪圖表面的類型設置為SURFACE\_TYPE\_PUSH\_BUFFERS，這樣攝像頭服務或者視頻播放服務就會為該SurfaceView繪圖表面創建一塊內存，並且將採集的預覽圖像數據或者視頻幀數據源源不斷地填充到該內存中去。注意，這塊內存有可能是來自專用的硬體的，例如，它可能是來自視頻卡的。在這種情況下，SurfaceFlinger服務使用一個LayerBuffer對象來描述該SurfaceView的繪圖表面。

從上面的描述就得到一個重要的結論：繪圖表面類型為SURFACE\_TYPE\_PUSH\_BUFFERS的SurfaceView的UI是不能由應用程序來控制的，而是由專門的服務來控制的，例如，攝像頭服務或者視頻播放服務，同時，SurfaceFlinger服務會使用一種特殊的LayerBuffer來描述這種繪圖表面。使用LayerBuffer來描述的繪圖表面在進行渲染的時候，可以使用硬體加速，例如，使用copybit或者overlay來加快渲染速度，從而可以獲得更流暢的攝像頭預覽或者視頻播放。

注意，我們在創建了一個SurfaceView之後，可以調用它的成員函數getHolder獲得一個SurfaceHolder對象，然後再調用該SurfaceHolder對象的成員函數setType來修改該SurfaceView的繪圖表面的類型，即修改該SurfaceView的成員變數mRequestedType的值。

介紹完成SurfaceView類的成員變數mSurface、mWindow、mWindowType和mRequestedType的含義之後，我們再介紹其它幾個接下來要用到的其它成員變數的含義：

**--mHaveFrame**，用來描述SurfaceView的宿主窗口的大小是否已經計算好了。只有當宿主窗口的大小計算之後，SurfaceView才可以更新自己的窗口。

**--mRequestedWidth**，用來描述SurfaceView最後一次被請求的寬度。

**--mRequestedHeight**，用來描述SurfaceView最後一次被請求的高度。

**--mRequestedFormat**，用來描述SurfaceView最後一次被請求的繪圖表面的像素格式。

**--mNewSurfaceNeeded**，用來描述SurfaceView是否需要新創建一個繪圖表面。

**--mLeft、mTop、mWidth、mHeight**，用來描述SurfaceView上一次所在的位置以及大小。

**--mFormat**，用來描述SurfaceView的繪圖表面上一次所設置的格式。

**--mVisible**，用來描述SurfaceView上一次被設置的可見性。

**--mType**，用來描述SurfaceView的繪圖表面上一次所設置的類型。

**--mUpdateWindowNeeded**，用來描述SurfaceView是否被WindowManagerService服務通知執行一次UI更新操作。

**--mReportDrawNeeded**，用來描述SurfaceView是否被WindowManagerService服務通知執行一次UI繪製操作。

**--mLayout**，指向的是一個WindowManager.LayoutParams對象，用來傳遞SurfaceView的布局參數以及屬性值給WindowManagerService服務，以便WindowManagerService服務可以正確地維護它的狀態。

理解了上述成員變數的含義的之後，接下來我們就可以分析SurfaceView類的成員函數updateWindow創建繪圖表面的過程了，如下所示：

(1). 判斷成員變數mHaveFrame的值是否等於false。如果是的話，那麼就說明現在還不是時候為SurfaceView創建繪圖表面，因為它的宿主窗口還沒有準備就緒。

(2). 獲得SurfaceView當前要使用的寬度和高度，並且保存在變數myWidth和myHeight中。注意，如果SurfaceView沒有被請求設置寬度或者高度，那麼就通過調用父類View的成員函數getWidth和getHeight來獲得它默認所使用的寬度和高度。

(3). 調用父類View的成員函數getLocationInWindow來獲得SurfaceView的左上角位置，並且保存在成員變數mLocation所描述的一個數組中。

(4). 判斷以下條件之一是否成立：

--SurfaceView的繪圖表面是否還未創建，即成員變數mWindow的值是否等於null；

--SurfaceView的繪圖表面的像素格式是否發生了變化，即成員變數mFormat和mRequestedFormat的值是否不相等；

--SurfaceView的大小是否發生了變化，即變數myWidth和myHeight是否與成員變數mWidth和mHeight的值不相等；

--SurfaceView的可見性是否發生了變化，即成員變數mVisible和mRequestedVisible的值是否不相等，或者成員變數NewSurfaceNeeded的值是否等於true；

--SurfaceView的繪圖表面的類型是否發生了變化，即成員變數mType和mRequestedType的值是否不相等；

--SurfaceView的位置是否發生了變化，即成員變數mLeft和mTop的值是否不等於前面計算得到的mLocation[0]和mLocation[1]的值；

--SurfaceView是否被WindowManagerService服務通知執行一次UI更新操作，即成員變數mUpdateWindowNeeded的值是否等於true；

--SurfaceView是否被WindowManagerService服務通知執行一次UI繪製操作，即成員變數mReportDrawNeeded的值是否等於true；

--SurfaceView類的成員函數updateWindow是否被調用者強制要求刷新或者繪製SurfaceView，即參數force或者redrawNeeded的值是否等於true。

只要上述條件之一成立，那麼SurfaceView類的成員函數updateWindow就需要對SurfaceView的各種信息進行更新，即執行以下第5步至第7步操作。

(5). 將SurfaceView接下來要設置的可見性、位置、大小、繪圖表面像素格式和類型分別記錄在成員變數mVisible、mLeft、mTop、mWidth、mHeight、mFormat和mType，同時還會將這些信息整合到成員變數mLayout所指向的一個WindowManager.LayoutParams對象中去，以便接下來可以傳遞給WindowManagerService服務。

(6). 檢查成員變數mWindow的值是否等於null。如果等於null的話，那麼就說明該SurfaceView還沒有增加到WindowManagerService服務中去。在這種情況下，就會創建一個MyWindow對象保存在該成員變數中，並且調用成員變數mSession所描述的一個Binder代理對象的成員函數addWithoutInputChannel來將該MyWindow對象傳遞給WindowManagerService服務。在前面的Step 4中提到，SurfaceView類的成員變數mSession指向的是一個實現了IWindowSession介面的Binder代理對象，該Binder代理對象引用的是運行在WindowManagerService服務這一側的一個Session對象。Session類的成員函數addWithoutInputChannel與另外一個成員函數add的實現是類似的，它們都是用來在WindowManagerService服務內部為指定的窗口增加一個WindowState對象，不過，Session類的成員函數addWithoutInputChannel只是在WindowManagerService服務內部為指定的窗口增加一個WindowState對象，而Session類的成員函數add除了會在WindowManagerService服務內部為指定的窗口增加一個WindowState對象之外，還會為該窗口創建一個用來接收用戶輸入的通道。

(7). 調用成員變數mSession所描述的一個Binder代理對象的成員函數relayout來請求WindowManagerService服務對SurfaceView的UI進行布局。WindowManagerService服務在對一個窗口進行布局的時候，如果發現該窗口的繪製表面還未創建，或者需要需要重新創建，那麼就會為請求SurfaceFlinger服務為該窗口創建一個新的繪圖表面，並且將該繪圖表面返回來給調用者。在我們這個情景中，WindowManagerService服務返回來的繪圖表面就會保存在成員變數mSurface。注意，這一步由於可能會修改SurfaceView的繪圖表面，即修改成員變數mSurface的指向的一個Surface對象的內容，因此，就需要在獲得成員變數mSurfaceLock所描述的一個鎖的情況下執行，避免其它線程同時修改該繪圖表面的內容，這是因為我們可能會使用一個獨立的線程來來繪製SurfaceView的UI。

執行完成上述步驟之後，SurfaceView的繪圖表面的創建操作就執行完成了，而當SurfaceView有了繪圖表面之後，我們就可以使用獨立的線程來繪製它的UI了，不過，在繪製之前，我們還需要在SurfaceView的宿主窗口上挖一個洞，以便繪製出來的UI不會被擋住。

2. SurfaceView的挖洞過程

SurfaceView的窗口類型一般都是TYPE\_APPLICATION\_MEDIA或者TYPE\_APPLICATION\_MEDIA\_OVERLAY，也就是說，它的Z軸位置是小於其宿主窗口的Z位置的。為了保證SurfaceView的UI是可見的，SurfaceView就需要在其宿主窗口的上面挖一個洞出來，實際上就是在其宿主窗口的繪圖表面上設置一塊透明區域，以便可以將自己顯示出來。

從SurfaceView的繪圖表面的創建過程可以知道，SurfaceView在被附加到宿主窗口之上的時候，會請求在宿主窗口上設置透明區域，而每當其宿主窗口刷新自己的UI的時候，就會將所有嵌入在它裡面的SurfaceView所設置的透明區域收集起來，然後再通知WindowManagerService服務為其設置一個總的透明區域。

從SurfaceView的繪圖表面的創建過程可以知道，SurfaceView在被附加到宿主窗口之上的時候，SurfaceView類的成員函數onAttachedToWindow就會被調用。SurfaceView類的成員函數onAttachedToWindow在被調用的期間，就會請求在宿主窗口上設置透明區域。接下來，我們就從SurfaceView類的成員函數onAttachedToWindow開始，分析SurfaceView的挖洞過程，如圖3所示：

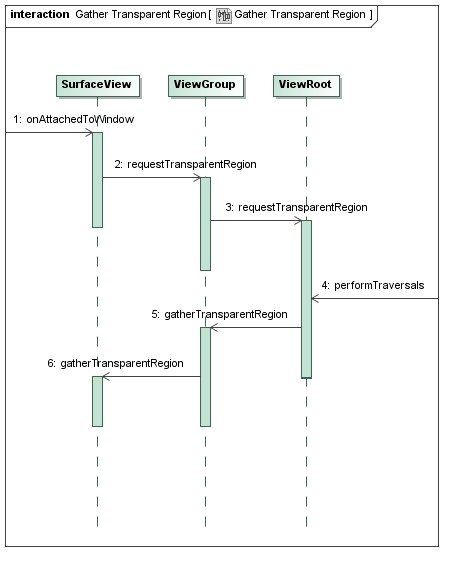
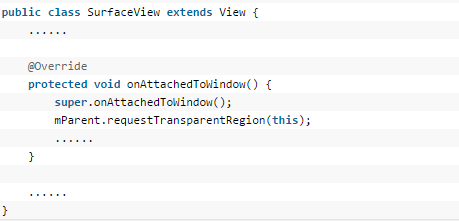


圖3 SurfaceView的挖洞過程

這個過程可以分為6個步驟，接下來我們就詳細分析每一個步驟。

Step 1. SurfaceView.onAttachedToWindow

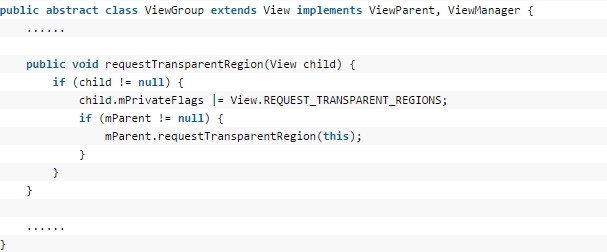


這個函數定義在文件frameworks/base/core/java/android/view/SurfaceView.java中。

SurfaceView類的成員變數mParent是從父類View繼承下來的，用來描述當前正在處理的SurfaceView的父視圖。我們假設當前正在處理的SurfaceView的父視圖就為其宿主窗口的頂層視圖，因此，接下來SurfaceView類的成員函數onAttachedToWindow就會調用DecorView類的成員函數requestTransparentRegion來請求在宿主窗口之上挖一個洞。

DecorView類的成員函數requestTransparentRegion是從父類ViewGroup繼承下來的，因此，接下來我們就繼續分析ViewGroup類的成員函數requestTransparentRegion的實現。

Step 2. ViewGroup.requestTransparentRegion

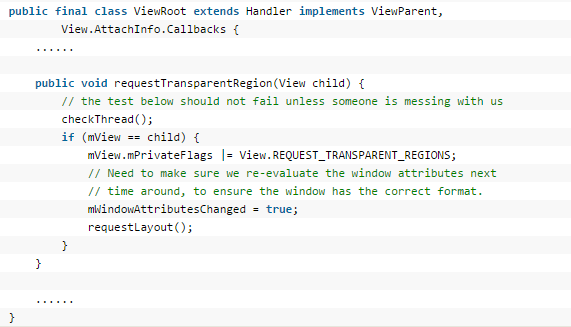


這個函數定義在文件frameworks/base/core/java/android/view/ViewGroup.java中。

參數child描述的便是要在宿主窗口設置透明區域的SurfaceView，ViewGroup類的成員函數requestTransparentRegion首先將它的成員變數mPrivateFlags的值的View.REQUEST\_TRANSPARENT\_REGIONS位設置為1，表示它要在宿主窗口上設置透明區域，接著再調用從父類View繼承下來的成員變數mParent所指向的一個視圖容器的成員函數requestTransparentRegion來繼續向上請求設置透明區域，這個過程會一直持續到當前正在處理的視圖容器為窗口的頂層視圖為止。

前面我們已經假設了參數child所描述的SurfaceView是直接嵌入在宿主窗口的頂層視圖中的，而窗口的頂層視圖的父視圖是使用一個ViewRoot對象來描述的，也就是說，當前正在處理的視圖容器的成員變數mParent指向的是一個ViewRoot對象，因此，接下來我們就繼續分析ViewRoot類的成員函數requestTransparentRegion的實現，以便可以繼續了解SurfaceView的挖洞過程。

Step 3. ViewRoot.requestTransparentRegion



這個函數定義在文件frameworks/base/core/java/android/view/ViewRoot.java中。

ViewRoot類的成員函數requestTransparentRegion首先調用另外一個成員函數checkThread來檢查當前執行的線程是否是應用程序的主線程，如果不是的話，那麼就會拋出一個類型為CalledFromWrongThreadException的異常。

通過了上面的檢查之後，ViewRoot類的成員函數requestTransparentRegion再檢查參數child所描述的視圖是否就是當前正在處理的ViewRoot對象所關聯的窗口的頂層視圖，即檢查它與ViewRoot類的成員變數mView是否是指向同一個View對象。由於一個ViewRoot對象有且僅有一個子視圖，因此，如果上述檢查不通過的話，那麼就說明調用者正在非法調用ViewRoot類的成員函數requestTransparentRegion來設置透明區域。

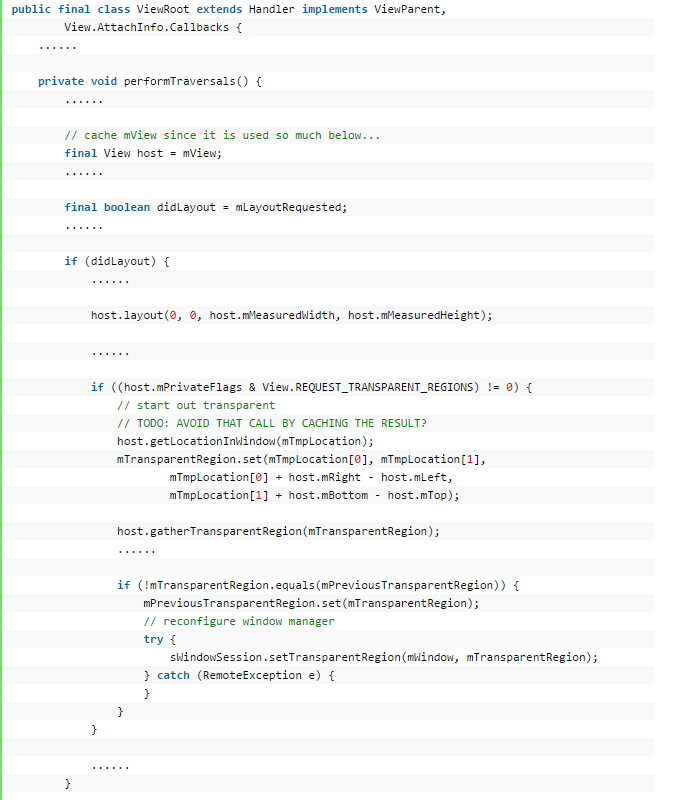
通過了上述兩個檢查之後，ViewRoot類的成員函數requestTransparentRegion就將成員變數mView所描述的一個窗口的頂層視圖的成員變數mPrivateFlags的值的View.REQUEST\_TRANSPARENT\_REGIONS位設置為1，表示該窗口被設置了一塊透明區域。

當一個窗口被請求設置了一塊透明區域之後，它的窗口屬性就發生變化了，因此，這時候除了要將與它所關聯的一個ViewRoot對象的成員變數mWindowAttributesChanged的值設置為true之外，還要調用該ViewRoot對象的成員函數requestLayout來請求刷新一下窗口的UI，即請求對窗口的UI進行重新布局和繪製。

ViewRoot類的成員函數requestLayout最終會調用到另外一個成員函數performTraversals來實際執行刷新窗口UI的操作。ViewRoot類的成員函數performTraversals在刷新窗口UI的過程中，就會將嵌入在它裡面的SurfaceView所要設置的透明區域收集起來，以便可以請求WindowManagerService將這塊透明區域設置到它的繪圖表面上去。

接下來，我們就繼續分析ViewRoot類的成員函數performTraversals的實現，以便可以繼續了解SurfaceView的挖洞過程。

Step 4. ViewRoot.performTraversals



這個函數定義在文件frameworks/base/core/java/android/view/ViewRoot.java中。

ViewRoot類的成員函數performTraversals是在窗口的UI布局完成之後，並且在窗口的UI繪製之前，收集嵌入在它裡面的SurfaceView所設置的透明區域的，這是因為窗口的UI布局完成之後，各個子視圖的大小和位置才能確定下來，這樣SurfaceView才知道自己要設置的透明區域的位置和大小。

變數host與ViewRoot類的成員變數mView指向的是同一個DecorView對象，這個DecorView對象描述的便是當前正在處理的窗口的頂層視圖。從前面的Step 3可以知道，如果當前正在處理的窗口的頂層視圖內嵌有SurfaceView，那麼用來描述它的一個DecorView對象的成員變數mPrivateFlags的值的View.REQUEST\_TRANSPARENT\_REGIONS位就會等於1。在這種情況下，ViewRoot類的成員函數performTraversals就知道需要在當前正在處理的窗口的上面設置一塊透明區域了。這塊透明區域的收集過程如下所示：

(1). 計算頂層視圖的位置和大小，即計算頂層視圖所佔據的區域。

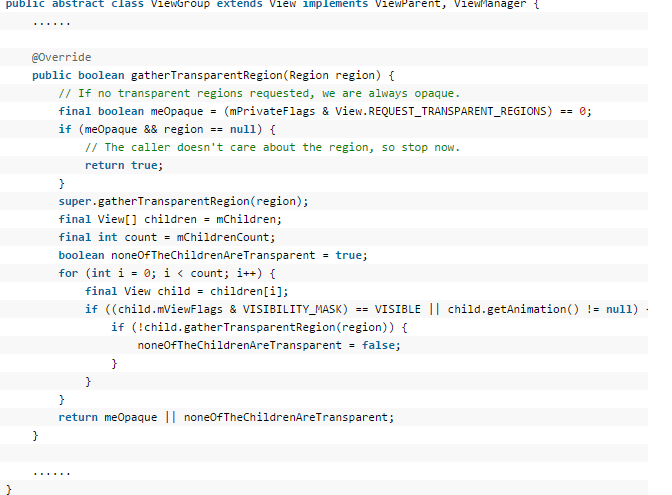
(2). 將頂層視圖所佔據的區域作為窗口的初始化透明區域，保存在ViewRoot類的成員變數mTransparentRegion中。

(3). 從頂層視圖開始，從上到下收集每一個子視圖所要設置的區域，最終收集到的總透明區域也是保存在ViewRoot類的成員變數mTransparentRegion中。

(4). 檢查ViewRoot類的成員變數mTransparentRegion和mPreviousTransparentRegion所描述的區域是否相等。如果不相等的話，那麼就說明窗口的透明區域發生了變化，這時候就需要調用ViewRoot類的的靜態成員變數sWindowSession所描述的一個Binder代理對象的成員函數setTransparentRegion通知WindowManagerService為窗口設置由成員變數mTransparentRegion所指定的透明區域。

其中，第(3)步是通過調用變數host所描述的一個DecorView對象的成員函數gatherTransparentRegion來實現的。 DecorView類的成員函數gatherTransparentRegion是從父類ViewGroup繼承下來的，因此，接下來我們就繼續分析ViewGroup類的成員函數gatherTransparentRegion的實現，以便可以了解SurfaceView的挖洞過程。

Step 5. ViewGroup.gatherTransparentRegion



這個函數定義在文件frameworks/base/core/java/android/view/ViewGroup.java中。

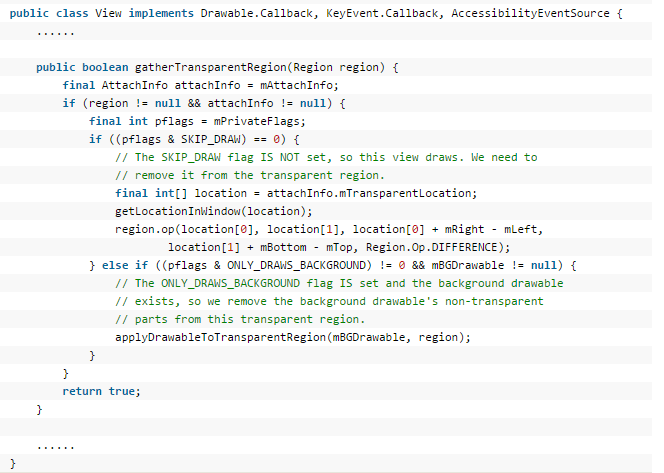
ViewGroup類的成員函數gatherTransparentRegion首先是檢查當前正在處理的視圖容器是否被請求設置透明區域，即檢查成員變數mPrivateFlags的值的 View.REQUEST\_TRANSPARENT\_REGIONS位是否等於1。如果不等於1，那麼就說明不用往下繼續收集窗口的透明區域了，因為在這種情況下，當前正在處理的視圖容器及其子視圖都不可能設置有透明區域。另一方面，如果參數region的值等於null，那麼就說明調用者不關心當前正在處理的視圖容器的透明區域，而是關心它是透明的，還是不透明的。在上述兩種情況下，ViewGroup類的成員函數gatherTransparentRegion都不用進一步處理了。

假設當前正在處理的視圖容器被請求設置有透明區域，並且參數region的值不等於null，那麼接下來ViewGroup類的成員函數gatherTransparentRegion就執行以下兩個操作：

(1). 調用父類View的成員函數gatherTransparentRegion來檢查當前正在處理的視圖容器是否需要繪製。如果需要繪製的話，那麼就會將它所佔據的區域從參數region所佔據的區域移除，這是因為參數region所描述的區域開始的時候是等於窗口的頂層視圖的大小的，也就是等於窗口的整個大小的。

(2). 調用當前正在處理的視圖容器的每一個子視圖的成員函數gatherTransparentRegion來繼續往下收集透明區域。

在接下來的Step 6中，我們再詳細分析當前正在處理的視圖容器的每一個子視圖的透明區域的收集過程，現在我們主要分析View類的成員函數gatherTransparentRegion的實現，如下所示：



這個函數定義在文件frameworks/base/core/java/android/view/View.java中。

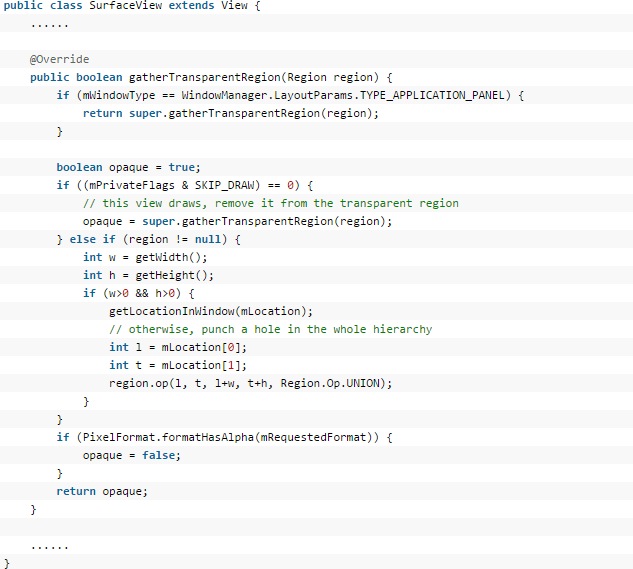
View類的成員函數gatherTransparentRegion首先是檢查當前正在處理的視圖的前景是否需要繪製，即檢查成員變數mPrivateFlags的值的SKIP\_DRAW位是否等於0。如果等於0的話，那麼就說明當前正在處理的視圖的前景是需要繪製的。在這種情況下，View類的成員函數gatherTransparentRegion就會將當前正在處理的視圖所佔據的區域從參數region所描述的區域中移除，以便當前正在處理的視圖的前景可以顯示出來。

另一方面，如果當前正在處理的視圖的前景不需要繪製，但是該視圖的背景需要繪製，並且該視圖是設置有的，即成員變數mPrivateFlags的值的SKIP\_DRAW位不等於0，並且成員變數mBGDrawable的值不等於null，這時候View類的成員函數gatherTransparentRegion就會調用另外一個成員函數applyDrawableToTransparentRegion來將該背景中的不透明區域從參數region所描述的區域中移除，以便當前正在處理的視圖的背景可以顯示出來。

回到ViewGroup類的成員函數gatherTransparentRegion中，當前正在處理的視圖容器即為當前正在處理的窗口的頂層視圖，前面我們已經假設它裡面嵌入有一個SurfaceView子視圖，因此，接下來就會收集該SurfaceView子視圖所設置的透明區域，這是通過調用SurfaceView類的成員函數gatherTransparentRegion來實現的。

接下來，我們就繼續分析SurfaceView類的成員函數gatherTransparentRegion的實現，以便可以繼續了解SurfaceView的挖洞過程。

Step 6. SurfaceView.gatherTransparentRegion



這個函數定義在文件frameworks/base/core/java/android/view/SurfaceView.java中。

SurfaceView類的成員函數gatherTransparentRegion首先是檢查當前正在處理的SurfaceView是否是用作窗口面板的，即它的成員變數mWindowType的值是否等於WindowManager.LayoutParams.TYPE\_APPLICATION\_PANEL。如果等於的話，那麼就會調用父類View的成員函數gatherTransparentRegion來檢查該面板是否需要繪製。如果需要繪製，那麼就會將它所佔據的區域從參數region所描述的區域移除。

假設當前正在處理的SurfaceView不是用作窗口面板的，那麼SurfaceView類的成員函數gatherTransparentRegion接下來就會直接檢查當前正在處理的SurfaceView是否是需要在宿主窗口的繪圖表面上進行繪製，即檢查成員變數mPrivateFlags的值的SKIP\_DRAW位是否等於1。如果需要的話，那麼也會調用父類View的成員函數gatherTransparentRegion來將它所佔據的區域從參數region所描述的區域移除。

假設當前正在處理的SurfaceView不是用作窗口面板，並且也是不需要在宿主窗口的繪圖表面上進行繪製的，而參數region的值又不等於null，那麼SurfaceView類的成員函數gatherTransparentRegion就會先計算好當前正在處理的SurfaceView所佔據的區域，然後再將該區域添加到參數region所描述的區域中去，這樣就可以得到窗口的一個新的透明區域。

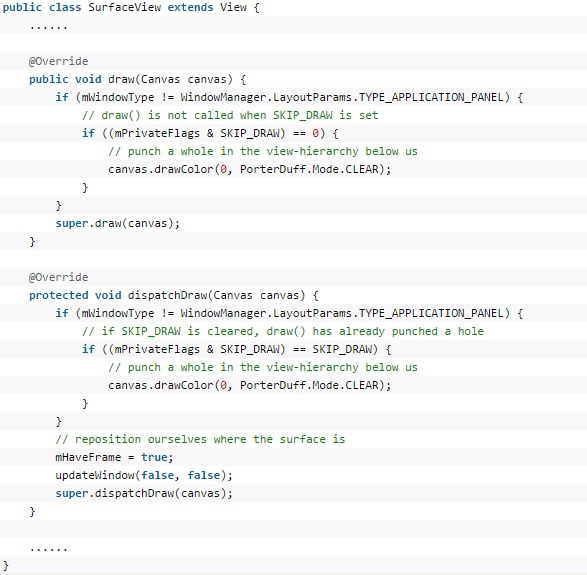
最後，SurfaceView類的成員函數gatherTransparentRegion判斷當前正在處理的SurfaceView的繪圖表面的像素格式是否設置有透明值。如果有的話，那麼就會將變數opaque的值設置為false，否則的話，變數opaque的值就保持為true。變數opaque的值最終會返回給調用者，這樣調用者就可以知道當前正在處理的SurfaceView的繪圖表面是否是半透明的了。

至此，我們就分析完成SurfaceView的挖洞過程了，接下來我們繼續分析SurfaceView的繪製過程。

3. SurfaceView的繪製過程

SurfaceView雖然具有獨立的繪圖表面，不過它仍然是宿主窗口的視圖結構中的一個結點，因此，它仍然是可以參與到宿主窗口的繪製流程中去的。從前面Android應用程序窗口（Activity）的測量（Measure）、布局（Layout）和繪製（Draw）過程分析一文可以知道，窗口在繪製的過程中，每一個子視圖的成員函數draw或者dispatchDraw都會被調用到，以便它們可以繪製自己的UI。

SurfaceView類的成員函數draw和dispatchDraw的實現如下所示：



這兩個函數定義在文件frameworks/base/core/java/android/view/SurfaceView.java中。

SurfaceView類的成員函數draw和dispatchDraw的參數canvas所描述的都是建立在宿主窗口的繪圖表面上的畫布，因此，在這塊畫布上繪製的任何UI都是出現在宿主窗口的繪圖表面上的。

本來SurfaceView類的成員函數draw是用來將自己的UI繪製在宿主窗口的繪圖表面上的，但是這裡我們可以看到，如果當前正在處理的SurfaceView不是用作宿主窗口面板的時候，即其成員變數mWindowType的值不等於WindowManager.LayoutParams.TYPE\_APPLICATION\_PANEL的時候，SurfaceView類的成員函數draw只是簡單地將它所佔據的區域繪製為黑色。

本來SurfaceView類的成員函數dispatchDraw是用來繪製SurfaceView的子視圖的，但是這裡我們同樣看到，如果當前正在處理的SurfaceView不是用作宿主窗口面板的時候，那麼SurfaceView類的成員函數dispatchDraw只是簡單地將它所佔據的區域繪製為黑色，同時，它還會通過調用另外一個成員函數updateWindow更新自己的UI，實際上就是請求WindowManagerService服務對自己的UI進行布局，以及創建繪圖表面，具體可以參考前面第1部分的內容。

從SurfaceView類的成員函數draw和dispatchDraw的實現就可以看出，SurfaceView在其宿主窗口的繪圖表面上面所做的操作就是將自己所佔據的區域繪為黑色，除此之外，就沒有其它更多的操作了，這是因為SurfaceView的UI是要展現在它自己的繪圖表面上面的。接下來我們就分析如何在SurfaceView的繪圖表面上面進行UI繪製。

從前面Android應用程序窗口（Activity）的測量（Measure）、布局（Layout）和繪製（Draw）過程分析一文可以知道，如果要在一個繪圖表面進行UI繪製，那麼就順序執行以下的操作：

(1). 在繪圖表面的基礎上建立一塊畫布，即獲得一個Canvas對象。

(2). 利用Canvas類提供的繪圖介面在前面獲得的畫布上繪製任意的UI。

(3). 將已經填充好了UI數據的畫布緩衝區提交給SurfaceFlinger服務，以便SurfaceFlinger服務可以將它合成到屏幕上去。

SurfaceView提供了一個SurfaceHolder介面，通過這個SurfaceHolder介面就可以執行上述的第(1)和引(3)個操作，示例代碼如下所示：

SurfaceView sv = (SurfaceView )findViewById(R.id.surface\_view);

SurfaceHolder sh = sv.getHolder();

Cavas canvas = sh.lockCanvas()

//Draw something on canvas

sh.unlockCanvasAndPost(canvas);

注意，只有在一個SurfaceView的繪圖表面的類型不是SURFACE\_TYPE\_PUSH\_BUFFERS的時候，我們才可以自由地在上面繪製UI。我們使用SurfaceView來顯示攝像頭預覽或者播放視頻時，一般就是會將它的繪圖表面的類型設置為SURFACE\_TYPE\_PUSH\_BUFFERS。在這種情況下，SurfaceView的繪圖表面所使用的圖形緩衝區是完全由攝像頭服務或者視頻播放服務來提供的，因此，我們就不可以隨意地去訪問該圖形緩衝區，而是要由攝像頭服務或者視頻播放服務來訪問，因為該圖形緩衝區有可能是在專門的硬體裡面分配的。

另外還有一個地方需要注意的是，上述代碼既可以在應用程序的主線程中執行，也可以是在一個獨立的線程中執行。如果上述代碼是在應用程序的主線程中執行，那麼就需要保證它們不會佔用過多的時間，否則的話，就會導致應用程序的主線程不能及時地響應用戶輸入，從而導致ANR問題。

為了方便起見，我們假設一個SurfaceView的繪圖表面的類型不是SURFACE\_TYPE\_PUSH\_BUFFERS，接下來，我們就從SurfaceView的成員函數getHolder開始，分析這個SurfaceView的繪製過程，如下所示：

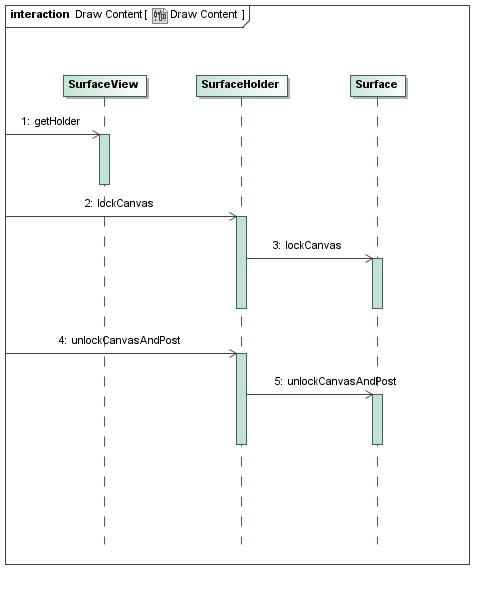
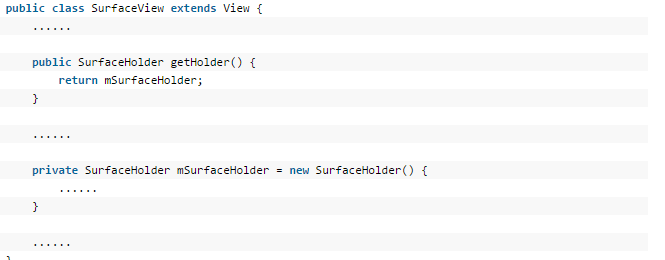


圖4 SurfaceView的繪製過程

這個過程可以分為5個步驟，接下來我們就詳細分析每一個步驟。

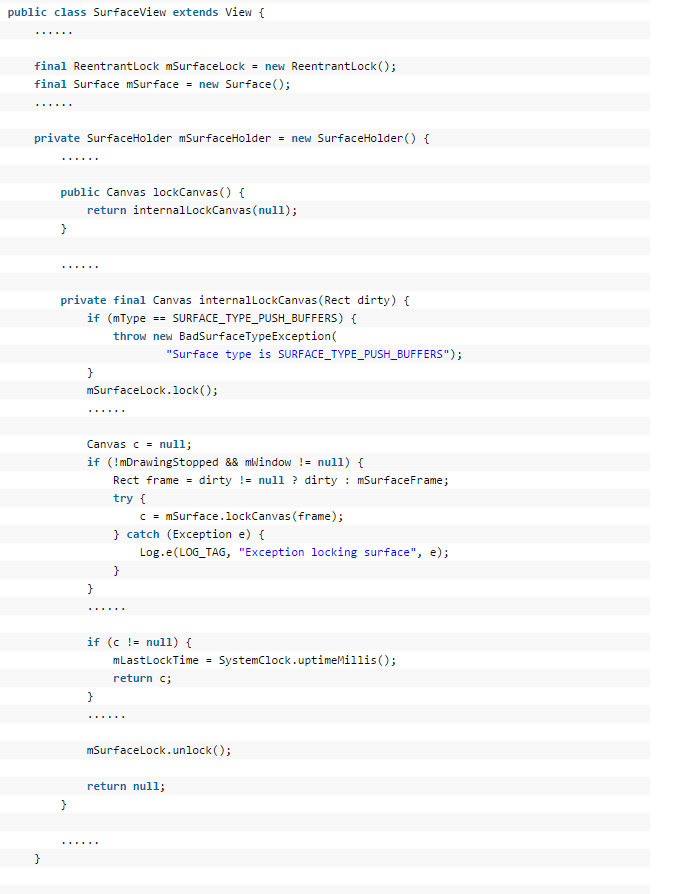
Step 1. SurfaceView.getHolder



這個函數定義在文件frameworks/base/core/java/android/view/SurfaceView.java中。

SurfaceView類的成員函數getHolder的實現很簡單，它只是將成員變數mSurfaceHolder所指向的一個SurfaceHolder對象返回給調用者。

Step 2. SurfaceHolder.lockCanvas



這個函數定義在文件frameworks/base/core/java/android/view/SurfaceView.java中。

SurfaceHolder類的成員函數lockCanvas通過調用另外一個成員函數internalLockCanvas來在當前正在處理的SurfaceView的繪圖表面上建立一塊畫布返回給調用者。

SurfaceHolder類的成員函數internalLockCanvas首先是判斷當前正在處理的SurfaceView的繪圖表面的類型是否是SURFACE\_TYPE\_PUSH\_BUFFERS，如果是的話，那麼就會拋出一個類型為BadSurfaceTypeException的異常，原因如前面所述。

由於接下來SurfaceHolder類的成員函數internalLockCanvas要在當前正在處理的SurfaceView的繪圖表面上建立一塊畫布，並且返回給調用者訪問，而這塊畫布不是線程安全的，也就是說它不能同時被多個線程訪問，因此，就需要對當前正在處理的SurfaceView的繪圖表面進行鎖保護，這是通過它的鎖定它的成員變數mSurfaceLock所指向的一個ReentrantLock對象來實現的。

注意，如果當前正在處理的SurfaceView的成員變數mWindow的值等於null，那麼就說明它的繪圖表面還沒有創建好，這時候就無法創建一塊畫布返回給調用者。同時，如果當前正在處理的SurfaceView的繪圖表面已經創建好，但是該SurfaceView當前是處於停止繪製的狀態，即它的成員變數mDrawingStopped的值等於true，那麼也是無法創建一塊畫布返回給調用者的。

假設當前正在處理的SurfaceView的繪製表面已經創建好，並且它不是處於停止繪製的狀態，那麼SurfaceHolder類的成員函數internalLockCanvas就會通過調用該SurfaceView的成員變數mSurface所指向的一個Surface對象的成員函數lockCanvas來創建一塊畫布，並且返回給調用者。注意，在這種情況下，當前正在處理的SurfaceView的繪製表面還是處於鎖定狀態的。

另一方面，如果SurfaceHolder類的成員函數internalLockCanvas不能成功地在當前正在處理的SurfaceView的繪製表面上創建一塊畫布，即變數c的值等於null，那麼SurfaceHolder類的成員函數internalLockCanvas在返回一個null值調用者之前，還會將該SurfaceView的繪製表面就會解鎖。

從前面第1部分的內容可以知道，SurfaceView類的成員變數mSurface描述的是就是SurfaceView的專有繪圖表面，接下來我們就繼續分析它所指向的一個Surface對象的成員函數lockCanvas的實現，以便可以了解SurfaceView的畫布的創建過程。

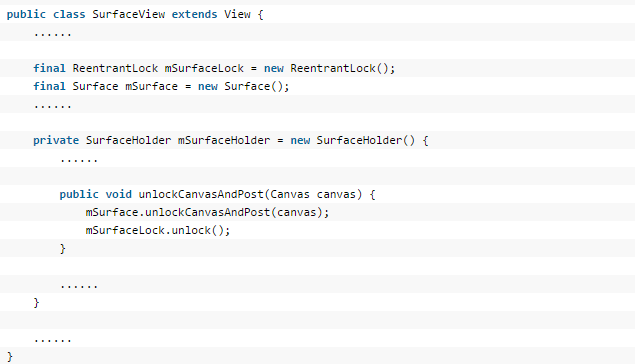
Step 3. Surface.lockCanvas

Surface類的成員函數lockCanvas的具體實現可以參考前面Android應用程序窗口（Activity）的測量（Measure）、布局（Layout）和繪製（Draw）過程分析一文，它大致就是通過JNI方法來在當前正在處理的繪圖表面上獲得一個圖形緩衝區，並且將這個圖形繪沖區封裝在一塊類型為Canvas的畫布中返回給調用者使用。

調用者獲得了一塊類型為Canvas的畫布之後，就可以調用Canvas類所提供的繪圖函數來繪製任意的UI了，例如，調用Canvas類的成員函數drawLine、drawRect和drawCircle可以分別用來畫直線、矩形和圓。

調用者在畫布上繪製完成所需要的UI之後，就可以將這塊畫布的圖形繪沖區的UI數據提交給SurfaceFlinger服務來處理了，這是通過調用SurfaceHolder類的成員函數unlockCanvasAndPost來實現的。

Step 4. SurfaceHolder.unlockCanvasAndPost



這個函數定義在文件frameworks/base/core/java/android/view/SurfaceView.java中。

SurfaceHolder類的成員函數unlockCanvasAndPost是通過調用當前正在處理的SurfaceView的成員變數mSurface所指向的一個Surface對象的成員函數unlockCanvasAndPost來將參數canvas所描述的一塊畫布的圖形緩衝區提交給SurfaceFlinger服務處理的。

提交完成參數canvas所描述的一塊畫布的圖形緩衝區給SurfaceFlinger服務之後，SurfaceHolder類的成員函數unlockCanvasAndPost再調用當前正在處理的SurfaceView的成員變數mSurfaceLock所指向的一個ReentrantLock對象的成員函數unlock來解鎖當前正在處理的SurfaceView的繪圖表面，因為在前面的Step 2中，我們曾經將該繪圖表面鎖住了。

接下來，我們就繼續分析Surface類的成員函數unlockCanvasAndPost的實現，以便可以了解SurfaceView的繪製過程。

Step 5. Surface.unlockCanvasAndPost

Surface類的成員函數unlockCanvasAndPost的具體實現同樣是可以參考前面Android應用程序窗口（Activity）的測量（Measure）、布局（Layout）和繪製（Draw）過程分析一文，它大致就是將在前面的Step 3中所獲得的一個圖形緩衝區提交給SurfaceFlinger服務，以便SurfaceFlinger服務可以在合適的時候將該圖形緩衝區合成到屏幕上去顯示，這樣就可以將對應的SurfaceView的UI展現出來了。

至此，我們就分析完成SurfaceView的繪製過程了，整個SurfaceView的實現原理也就分析完了。總結來說，就是SurfaceView有以下三個特點：

A. 具有獨立的繪圖表面；

B. 需要在宿主窗口上挖一個洞來顯示自己；

C. 它的UI繪製可以在獨立的線程中進行，這樣就可以進行複雜的UI繪製，並且不會影響應用程序的主線程響應用戶輸入。

TAG: |