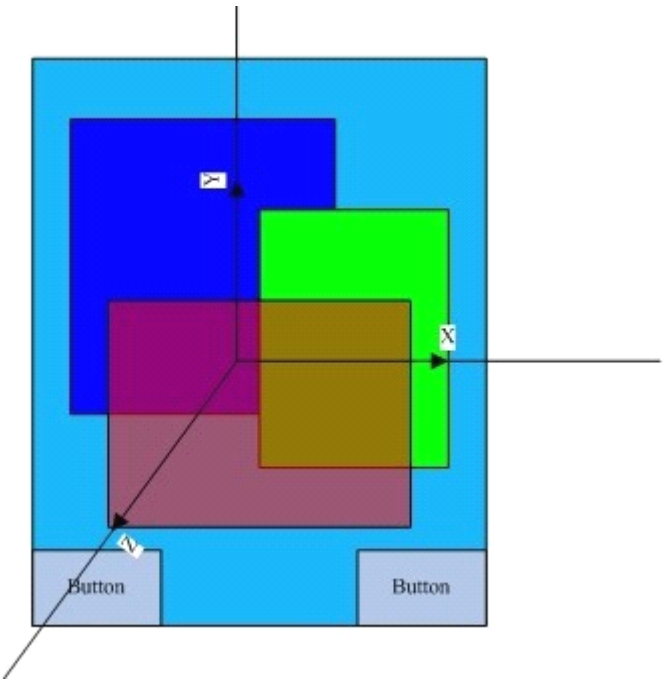
Android Display System --- Surface Flinger

Android Display System --- Surface Flinger

SurfaceFlinger 是 Android multimedia 的一个部分，在 Android 的实现中它是一个 service ，提供系统 范围内的 surface composer 功能，它能够将各种应用 程序的2D 、3D surface 进行组合。在具体讲 SurfaceFlinger 之前，我们先来看一下有关显示方面的一些基础 知识 。

1 、原理 分析

让我们首先看一下下面的屏幕简略图：



每个应用程序可能对应着一个或者多个图形界面，而每个界面我们就称之为一个 surface ，或者说是 window ，在上面的图中我们能看到4 个 surface ，一个是 home 界面，还有就是红、绿、蓝分别代表的3 个 surface ，而两个 button 实际是 home surface 里面的内容。在这里我们能看到我们进行图形显示所需要解决 的问题：

a 、首先每个 surface 在屏幕上有它的位置，以及大小，然后每个 surface 里面还有要显示的内容，内容，大小，位置 这些元素 在我们改变应用程序的时候都可能会改变，改变时应该如何处理 ？

b 、然后就各个 surface 之间可能有重叠，比如说在上面的简略图中，绿色覆盖了蓝色，而红色又覆盖了绿色和蓝色以及下面的 home ，而且还具有一定透明度。这种层之间的关系应该如何描述？

我们首先来看第二个问题，我们可以想象在屏幕平面的垂直方向还有一个 Z 轴，所有的 surface 根据在 Z 轴上的坐标来确定前后，这样就可以描述各个 surface 之间的上下覆盖关系了，而这个在 Z 轴上的顺序，图形上有个专业术语叫 Z-order 。

对于第一个问题，我们需要一个结构来记录应用程序界面的位置，大小，以及一个 buffer 来记录需要显示的内容，所以这就是我们 surface 的概念，surface 实际我们可以把它理解成一个容器，这个容器记录着应用程序界面的控制信息，比如说大小啊，位置啊，而它还有 buffer 来专门存储需要显示的内容。

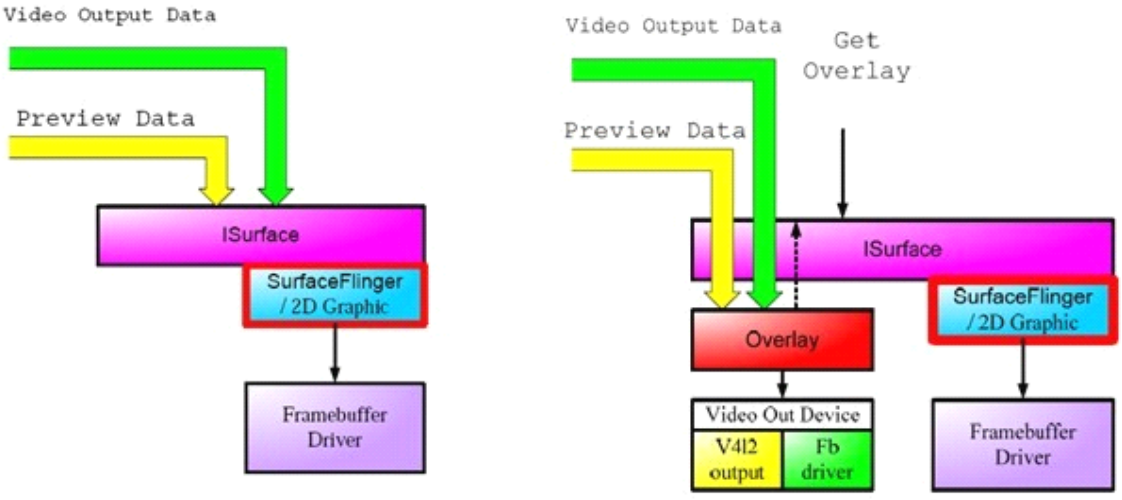
在这里还存在一个问题，那就是当存在图形重合的时候应该如何处理呢，而且可能有些 surface 还带有透明信息，这里就是我们 SurfaceFlinger 需要解决问题，它要把各个 surface 组合(compose/merge) 成一个 main Surface ，最后将 Main Surface 的内容发送给 FB/V4l2 Output ，这样屏幕上就能看到我们想要的效果。

在实际中对这些 Surface 进行 merge 可以采用两种方式，一种就是采用软件的形式来 merge ，还一种就是采用硬件的方式，软件的方式就是我们的 SurfaceFlinger ，而硬件的方式就是 Overlay 。

2 、OverLay

因为硬件 merge 内容相对简单，我们首先来看 overlay 。 Overlay 实现的方式有很多，但都需要硬件的支持。以 IMX51 为例子，当 IPU 向内核申请 FB 的时候它会申请3 个 FB ，一个是主屏的，还一个是副屏的，还一个就是 Overlay 的。 简单地来说，Overlay 就是我们将硬件所能接受的格式数据 和控制信息送到这个 Overlay FrameBuffer，由硬件驱动来负责 merge Overlay buffer 和主屏 buffer 中的内容。

一般来说现在的硬件都只支持一个 Overlay，主要用在视频播放以及 camera preview 上，因为视频内容的不断变化用硬件 Merge 比用软件 Merge 要有效率得多，下面就是使用 Overlay 和不使用 Overlay 的过程：



SurfaceFlinger 中加入了 Overlay hal，只要实现这个 Overlay hal 可以使用 overlay 的功能，这个头文件 在：/hardware/libhardware/include/harware/Overlay.h，可以使用 FB 或者 V4L2 output 来实现，这个可能是我们将来工作的内容。实现 Overlay hal 以后,使用 Overlay 接口的 sequence 就在：/frameworks/base/libs/surfaceflinger/tests/overlays/Overlays.cpp,这个 sequnce 是很重要的，后面我们会讲到。

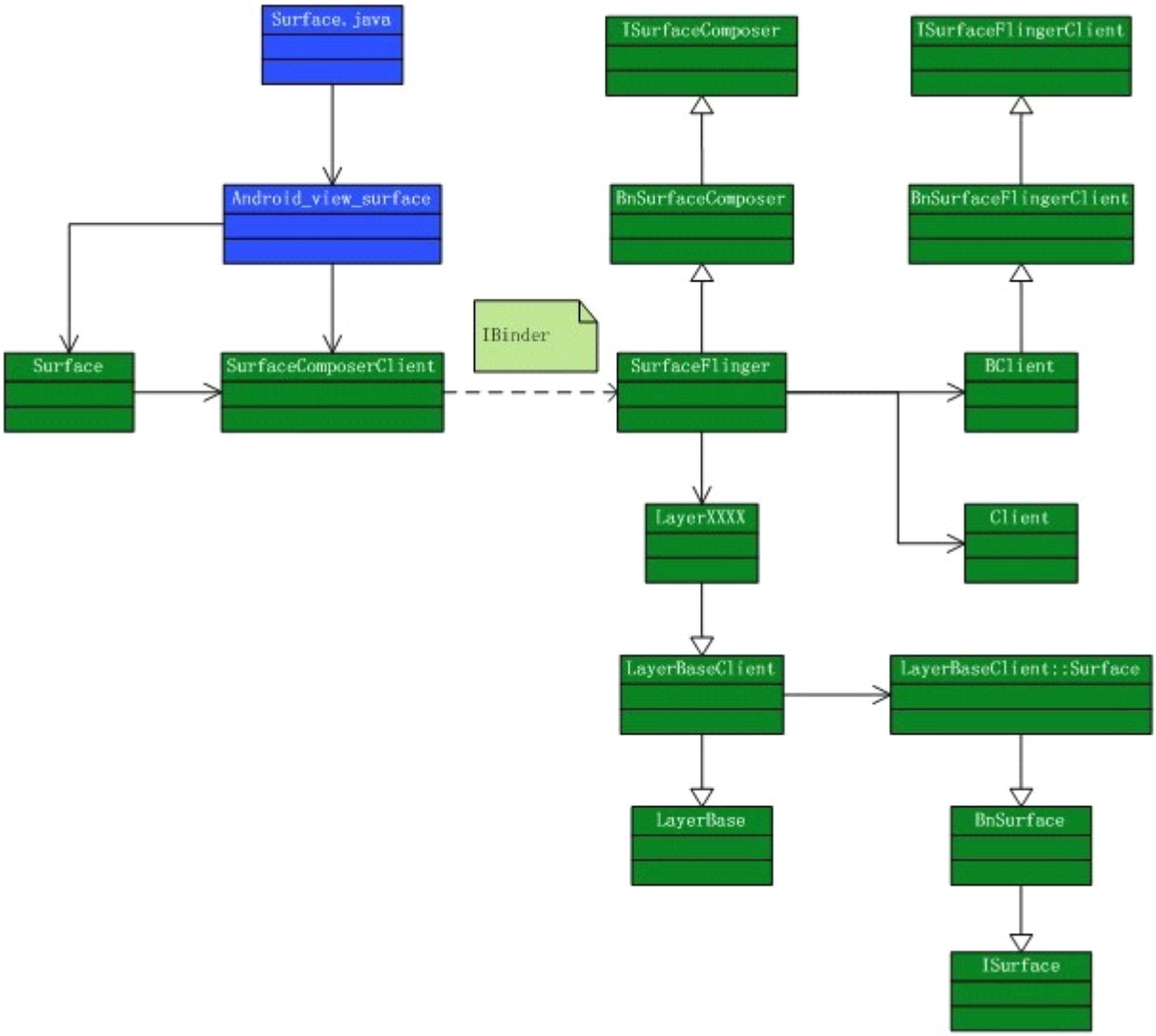
不 过在实际中我们不一定需要实现 Overlay hal,如果了解硬件的话，可以在驱动中直接把这些信息送到 Overlay Buffer,而不需要走上层的 Android。Fsl 现在的 Camera preview 就是采用的这种方式，而且我粗略看了 r3补丁的内容，应该在 opencore 的视频播放这块也实现了 Overlay。

3、SurfaceFlinger

现 在就来看看最复杂的 SurfaceFlinger，首先要明确的是 SurfaceFlinger 只是负责 merge Surface 的控制，比如说计算出两个 Surface 重叠的区域，至于 Surface 需要显示的内容，则通过 skia，opengl 和 pixflinger 来计算。 所以我们在介绍 SurfaceFlinger 之前先忽略里面存储的内容究竟是什么，先弄清楚它对 merge 的一系列控制的过程，然后再结合2D ，3D 引擎来看它的处理过程。

3.1 、Surface 的创建过程

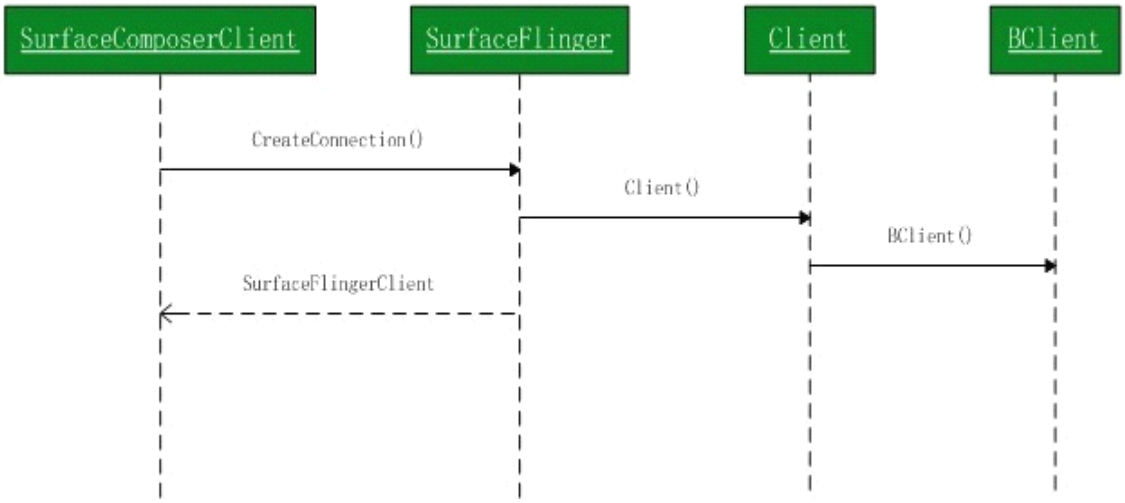
前面提到了每个应用程序可能有一个或者多个 Surface ， 我们需要一些数据结构来存储我们的窗口信息，我们还需要 buffer 来存储我们的窗口内容， 而且最主要的是我们应该确定一个方案 来和 SurfaceFlinger 来交互这些信息，让我们首先看看下面的 Surface 创建过程的类图 ：



在 IBinder 左边的就是客户端部分，也就是需要窗口显示的应用程序，而右边就是我

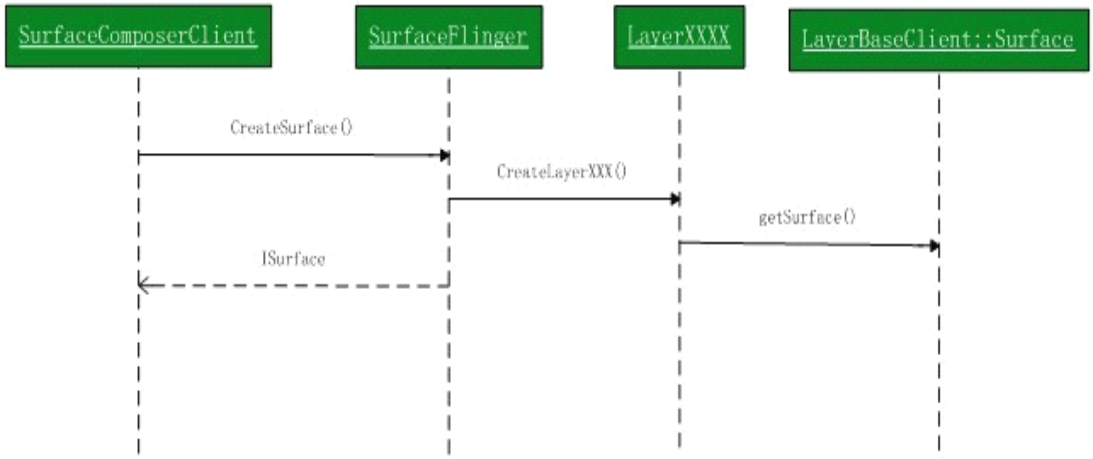
们的 Surface Flinger service 。 创建一个 surface 分为两个过程，一个是在 SurfaceFlinger 这边为每个应用程序(Client) 创建一个管理 结构，另一个就是创建存储内容的 buffer ，以及在这个 buffer 上的一系列画图之类的操作。

因为 SurfaceFlinger 要管理多个应用程序的多个窗口界面，为了进行管理它提供了一个 Client 类，每个来请求服务的应用程序就对应了一个 Client 。因为 surface 是在 SurfaceFlinger 创建的，必须返回一个结构让应用程序知道自己申请的 surface 信息，因此SurfaceFlinger 将Client 创建的控制结构per_client_cblk_t 经过BClient 的封装以后返回给 SurfaceComposerClient ，并向应用程序提供了一组创建和销毁 surface 的操作：



为应用程序创建一个 Client 以后，下面需要做的就是为这个 Client 分配 Surface ， Flinger 为每个 Client 提供了 8M 的空间 ，包括控制信息和存储内容的 buffer 。在说创建 surface 之前首先要理解 layer 这个概念，回到我们前面看的屏幕简略图，实际上每个窗口就是 z 轴上的一个 layer ， layer 提供了对窗口控制信息的操作，以及内容的处理 （ 调用 opengl 或者 skia），也就是说 SurfaceFlinger 只是控制什么时候应该进行这些信息的处理以及处理的过程，所有实际的处理都是在 layer 中进行的，可以理解为创建一个 Surface 就是创建一个 Layer 。不得不说 Android 这些乱七八糟的名字，让我绕了很久……

创建 Layer 的过程，首先是由这个应用程序的 Client 根据应用程序的 pid 生成一个唯一的 layer ID ，然后根据大小，位置，格式啊之类的信息创建出 Layer 。在 Layer 里面有一个嵌套的 Surface 类，它主要包含一个 ISurfaceFlingerClient::Surface_data_t ，包含了这个 Surace 的统一标识符以及 buffer 信息等，提供给应用程序使用。最后应用程序会根据返回来的 ISurface 信息等创建自己的一个 Surface 。



Android 提供了 4 种类型的 layer 供选择，每个 layer 对应一种类型的窗口，并对应这种窗口相应的操作： Layer ， LayerBlur ， LayerBuffer ， LayerDim 。不得不说再说 Android 起的乱七八糟的名字， LayerBuffer 很容易让人理解成是 Layer 的 Buffer ，它实际上是一种 Layer 类型。各个 Layer 的效果大家可以参考 Surface.java 里面的描述： /frameworks/base/core/java/android/view/surface.java 。这里要重点说一下两种 Layer ，一个是 Layer (norm layer) ，另一个是 LayerBuffer 。

Norm Layer 是 Android 种使用最多的一种 Layer ，一般的应用程序在创建 surface 的时候都是采用的这样的 layer ，了解 Normal Layer 可以让我们知道 Android 进行 display 过程中的一些基础原理。 Normal Layer 为每个 Surface 分配两个 buffer ： front buffer 和 back buffer ，这个前后是相对的概念，他们是可以进行 Flip 的。Front buffer 用于 SurfaceFlinger 进行显示，而 Back buffer 用于应用程序进行画图，当 Back buffer 填满数据 (dirty) 以后，就会 flip ， back buffer 就变成了 front buffer 用于显示，而 front buffer 就变成了 back buffer 用来画图，这两个 buffer 的大小是根据 surface 的大小格式动态变化的。这个动态变化的实现我没仔细看，可以参照 ： /frameworks/base/lib/surfaceflinger/layer.cpp 中的 setbuffers() 。

两个 buffer flip 的方式是 Android display 中的一个重要实现方式，不只是每个 Surface 这么实现，最后写入 FB 的 main surface 也是采用的这种方式。

LayerBuffer 也是将来必定会用到的一个 Layer ，个人觉得也是最复杂的一个 layer ，它不具备 render buffer ，主要用在 camera preview / video playback 上。它提供了两种实现方式，一种就是 post buffer ，另外一种就是我们前面提到的 overlay ， Overlay 的接口实际上就是在这个 layer 上实现的。不管是 overlay 还是 post buffer 都是指这个 layer 的数据来源自其他地方，只是 post buffer 是通过软件的方式最后还是将这个 layer merge 主的 FB ，而 overlay 则是通过硬件 merge 的方式来实现。与这个 layer 紧密联系在一起的是 ISurface 这个接口，通过它来注册数据来源，下面我举个例子来说明这两种方式的使用方法：

前面几个步骤是通用的：

// 要使用 Surfaceflinger 的服务必须先创建一个 client

```
sp<SurfaceComposerClient> client = new SurfaceComposerClient();
// 然后向 Surfaceflinger 申请一个 Surface ， surface 类型为 PushBuffers
sp<Surface> surface = client->createSurface(getpid(), 0, 320, 240,
            PIXEL_FORMAT_UNKNOWN, ISurfaceComposer::ePushBuffers);
// 然后取得 ISurface 这个接口， getISurface() 这个函数的调用时具有权限限制的，必须在 Surface.h 中打开： /framewoks/base/include/ui/Surface.h
sp<ISurface> isurface = Test::getISurface(surface);

//overlay 方式下就创建 overlay ，然后就可以使用 overlay 的接口了
sp<OverlayRef> ref = isurface->createOverlay(320, 240, PIXEL_FORMAT_RGB_565);
sp<Overlay> verlay = new Overlay(ref);

//post buffer 方式下，首先要创建一个 buffer ，然后将 buffer 注册到 ISurface 上
ISurface::BufferHeap buffers(w, h, w, h,
                                        PIXEL_FORMAT_YCbCr_420_SP,
                                        transform,
                                        0,
                                        mHardware->getPreviewHeap());
mSurface->registerBuffers(buffers);
```

3.2 、应用 程序对窗口的控制和画图

Surface 创建以后，应用程序就可以在 buffer 中画图了，这里就面对着两个问题了，一个是怎么知道在哪个 buffer 上来画图，还一个就是画图以后如何通知 SurfaceFlinger 来进行 flip 。除了画图之外，如果我们移动窗口以及改变窗口大小的时候，如何告诉 SurfaceFlinger 来进行处理呢 ？在明白这些问题之前，首先我们要了解 SurfaceFlinger 这个服务 是如何运作的：

从类图中可以看到 SurfaceFlinger 是一个线程类，它继承了 Thread 类。当创建 SurfaceFlinger 这个服务的时候会启动一个 SurfaceFlinger 监听线程，这个线程会一直等待事件的发生，比如说需要进行 sruface flip ，或者说窗口位置大小发生了变化等等，一旦产生这些事件， SurfaceComposerClient 就会通过 IBinder 发出信号，这个线程就会结束等待处理这些事件，处理完成以后会继续等待，如此循环。

SurfaceComposerClient 和 SurfaceFlinger 是通过 SurfaceFlingerSynchro 这个类来同步信号的，其实说穿了就是一个条件变量。监听线程等待条件的值变成 OPEN ，一旦变成 OPEN 就结束等待并将条件置成 CLOSE 然后进行事件处理，处理完成以后再继续等待条件的值变成 OPEN ，而 Client 的 Surface 一旦改变就通过 IBinder 通知 SurfaceFlinger 将条件变量的值变成 OPEN ，并唤醒等待的线程，这样就通过线程类和条件变量实现了一个动态处理机制。

了解了 SurfaceFlinger 的事件机制我们再回头看看前面提到的问题了。首先在对 Surface 进行画图之前必须锁定 Surface 的 layer ，实际上就是锁定了 Layer_cblk_t 里的 swapstate 这个变量。SurfaceComposerClient 通过 swapsate 的值来确定要使用哪个 buffer 画图，如果 swapstate 是下面的值就会阻塞 Client ，就不翻译了直接 copy 过来：

```
// We block the client if:
 // eNextFlipPending:  we've used both buffers already, so we need to
 //                    wait for one to become availlable.
 // eResizeRequested:  the buffer we're going to acquire is being
 //                    resized. Block until it is done.
 // eFlipRequested && eBusy: the buffer we're going to acquire is
 //                    currently in use by the server.
 // eInvalidSurface:   this is a special case, we don't block in this
 //                    case, we just return an error.
```

所以应用程序先调用 lockSurface() 锁定 layer 的 swapstate ，并获得画图的 buffer 然后就可以在上面进行画图了，完成以后就会调用 unlockSurfaceAndPost() 来通知 SurfaceFlinger 进行 Flip 。或者仅仅调用 unlockSurface() 而不通知 SurfaceFlinger 。

一般来说画图的过程需要重绘 Surface 上的所有像素，因为一般情况下显示过后的像素是不做保存的，不过也可以通过设定来保存一些像素，而只绘制部分像素，这里就涉及到像素的拷贝了，需要将 Front buffer 的内容拷贝到 Back buffer 。在 SurfaceFlinger 服务实现中像素的拷贝是经常需要进行的操作，而且还可能涉及拷贝过程的转换，比如说屏幕的旋转，翻转等一系列操作。因此 Android 提供了拷贝像素的 hal ，这个也可能是我们将来需要实现的，因为用硬件完成像素的拷贝，以及拷贝过程中可能的矩阵变换等操作，比用 memcpy 要有效率而且节省资源。这个 HAL 头文件在：/hardware/libhardware/hardware/include/copybit.h

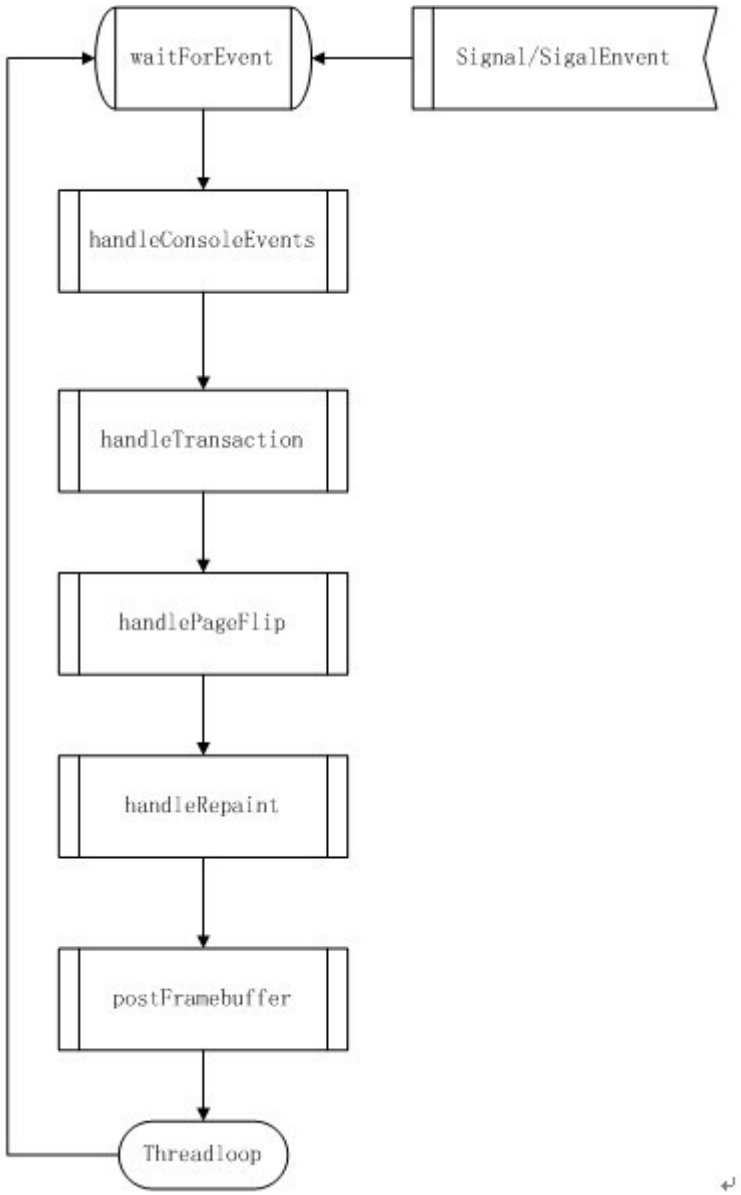
窗口状态变化的处理是一个很复杂的过程，首先要说明一下， SurfaceFlinger 只是执行 Windows manager 的指令，由 Windows manager 来决定什么是偶改变大小，位置，设置透明度,以及如何调整 layer 之间的顺序，SurfaceFlinger 仅仅只是执行它的指令。PS ：Windows Manager 是 java 层的一个服务，提供对所有窗口的管理 功能，这部分的内容我没细看过，觉得是将来需要了解的内容。

窗口状态的变化包括位置的移动，窗口大小，透明度， z-order 等等，首先我们来了解一下 SurfaceComposerClient 是如何和 SurfaceFlinger 来交互这些信息的。当应用程序需要改变窗口状态的时候它将所有的状态改变信息打包，然后一起发送给 SurfaceFlinger ， SurfaceFlinger 改变这些状态信息以后，就会唤醒等待的监听线程，并设置一个标志位告诉监听线程窗口的状态已经改变了，必须要进行处理，在 Android 的实现中，这个打包的过程就是一个 Transaction ，所有对窗口状态（layer_state_t）的改变都必须在一个 Transaction 中。

到这里应用程序客户端的处理过程已经说完了，基本分为两个部分，一个就是在窗口画图，还一个就是窗口状态改变的处理。

4 、 SurfaceFlinger 的处理过程

了解了 Flinger 和客户端的交互，我们再来仔细看看 SurfaceFlinger 的处理过程，前面已经说过了 SurfaceFlinger 这个服务在创建的时候会启动一个监听的线程,这个线程负责每次窗口更新时候的处理，下面我们来仔细看看这个线程的事件的处理，大致就是下面的这个图：



先大致讲一下 Android 组合各个窗口的原理 ： Android 实际上是通过计算每一个窗口的可见区域，就是我们在屏幕上可见的窗口区域 （ 用 Android 的词汇来说就是 visibleRegionScreen ），然后将各个窗口的可见区域画到一个主 layer 的相应部分，最后就拼接成了一个完整的屏幕，然后将主 layer 输送到 FB 显示。在将各个窗口可见区域画到主 layer 过程中涉及到一个硬件实现和一个软件实现的问题，如果是软件实现则通过 Opengl 重新画图，其中还包括存在透明度的 alpha 计算；如果实现了 copybit hal 的话，可以直接将窗口的这部分数据 直接拷贝过来，并完成可能的旋转，翻转，以及 alhpa 计算等。

下面来看看 Android 组合各个 layer 并送到 FB 显示的具体过程：

4.1 、 handleConsoleEvent

当接收到 signal 或者 singalEvent 事件以后，线程就停止等待开始对 Client 的请求进行处理，第一个步骤是 handleConsoleEvent ，这个步骤我看了下和 /dev/console 这个设备有关，它会取得屏幕或者释放屏幕，只有取得屏幕的时候才能够在屏幕上画图。

4.2 、 handleTransaction

前面提到过，窗口状态的改变只能在一个 Transaction 中进行。因为窗口状态的改变可能造成本窗口和其他窗口的可见区域变化，所以就必须重新来计算窗口的可见区域。在这个处理子过程中 Android 会根据标志位来对所有 layer 进行遍历，一旦发现哪个窗口的状态发生了变化就设置标志位以在将来重新计算这个窗口的可见区域。在完成所有子 layer 的遍历以后， Android 还会根据标志位来处理主 layer ，举个例子，比如说传感器感应到手机横过来了，会将窗口横向显示，此时就要重新设置主 layer 的方向。

4.3 、 handlePageFlip

这里会处理每个窗口 surface buffer 之间的翻转，根据 layer_state_t 的 swapsate 来决定是否要翻转，当 swapsate 的值是 eNextFlipPending 是就会翻转。处理完翻转以后它会重新计算每个 layer 的可见区域，这个重新计算的过程我还没看太明白，但大致是一个这么的过程：

从 Z 值最大的 layer 开始计算，也就是说从最上层的 layer 计算，去掉本身的透明区域和覆盖在它上面的不透明区域，得到的就是这个 layer 的可见区域。然后这个 layer 的不透明区域就会累加到不透明覆盖区域，这个 layer 的可见区域会放入到主 layer 的可见区域，然后计算下一个 layer ，直到计算完所有的 layer 的可见区域。这中间的计算是通过定义在 skia 中的一种与或非的图形逻辑运算实现的，类似我们数学中的与或非逻辑图。

4.4 、 handleRepaint

计算出每个 layer 的可见区域以后，这一步就是将所有可见区域的内容画到主 layer 的相应部分了，也就是说将各个 surface buffer 里面相应的内容拷贝到主 layer 相应的 buffer ，其中可能还涉及到 alpha 运算，像素的翻转，旋转等等操作，这里就像我前面说的可以用硬件来实现也可以用软件来实现。在使用软件的 opengl 做计算的过程中还会用到 PixFlinger 来做像素的合成，这部分内容我还没时间来细看。

4.5 、 postFrameBuffer

最后的任务就是翻转主 layer 的两个 buffer ，将刚刚写入的内容放入 FB 内显示了。