# Todo

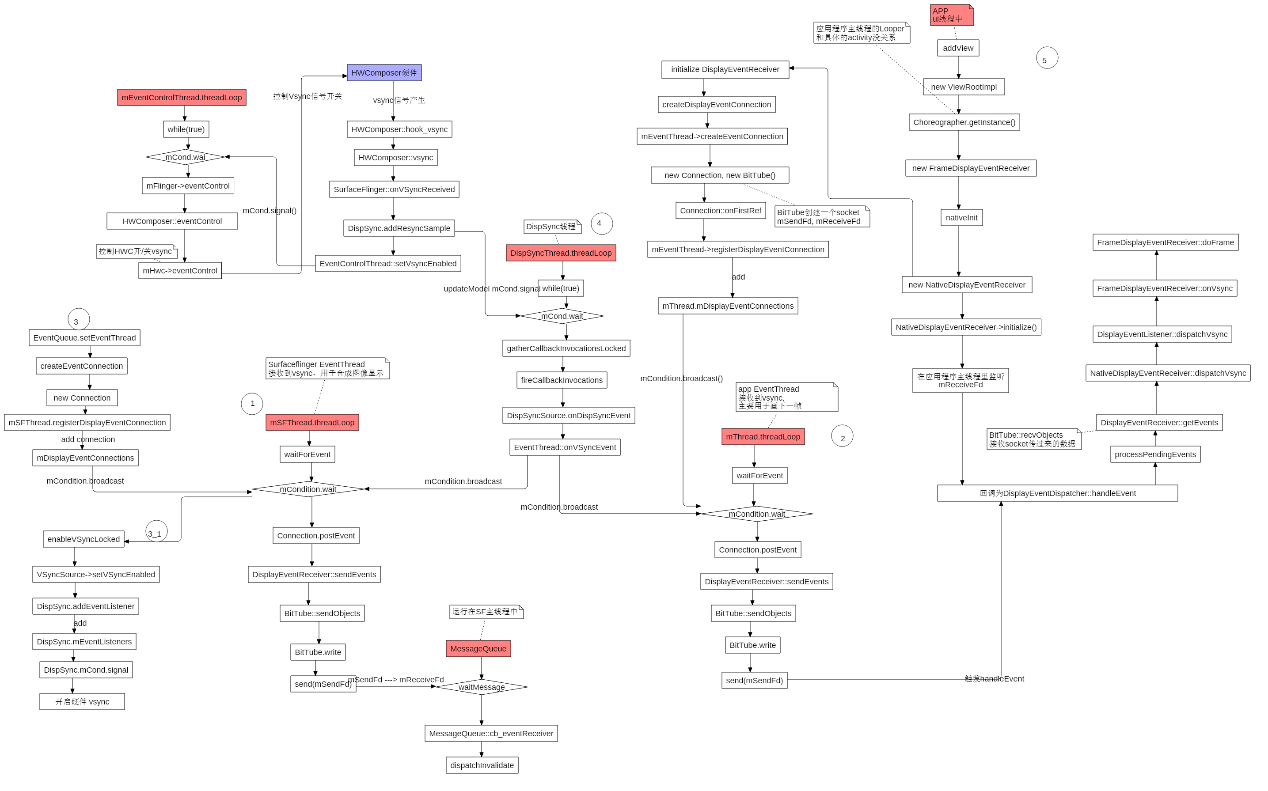
垂直同步（VSYNC）实现原理

测一下延时

# Vsync模型

Android 7.0

## Vsync的线程图



进程之前是怎么通信的呢？线程之间的通信呢

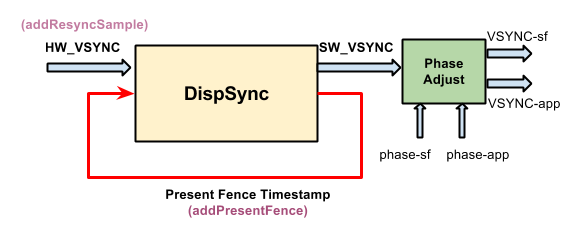
由图1可以看出与vsync相关的SurfaceFlinger线程主要有以下几个

1. **EventControlThread:** 控制硬件vsync的开关
2. **DispSyncThread:** 软件产生vsync的线程
3. **SF EventThread:** 该线程用于SurfaceFlinger接收vsync信号用于渲染
4. **App EventThread:** 该线程用于接收vsync信号并且上报给App进程，App开始画图

从这4个线程，其实我们可以将vsync分为4种不同的类型

* HW vsync, 真实由硬件产生的vsync信号
* SW vsync, 由DispSync产生的vsync信号
* SF vsync, SF接收到的vsync信号
* App vsync, App接收到的vsync信号

[DispSync](https://link.jianshu.com?t=http:/echuang54.blogspot.tw/2015/01/dispsync.html)这篇文章里用了一个非常非常准确的 PLL 图来表示上面4个vsync信号之间的关系。



因此可以看出 SW vsync/App vsync 并不是直接由HW vsync产生的，而是由SW vsync产生的，HW vsync作为SW vsync的参考，动态的更新SW vsync里的模型参数，这样让SW vsync能与HW vsync更加的精确吧。

那么为什么SurfaceFlinger要用SW vsync而不是直接用HW vsync呢？  
猜想可能是因为HW vsync每隔固定时间由显示屏产生中断，然后传给driver, driver再回调给SurfaceFlinger, 这样经过层层回调，会对performance有影响吧。而SW vsync直接由SurfaceFlinger产生，省略了很多步骤。

所以我个人觉得SurfaceFlinger最重要的是要搞明白 SW vsync是怎么运作的。

## 垂直同步（VSYNC）实现原理

VSYNC在显示周期内同步一些确定的事件,APP在VSYNC结束的时间点绘制画面，也是在这个时间点SurfaceFlinger进行画面合成。这种机制消除了卡顿，提高了图形的视觉表现。硬件合成器(HWC)引用了VSYNC的实现函数

int(waitForVsync\*)(int64\_t\*timestamp)

在VSYNC发生之前这个函数是阻塞的，函数会返回真实的VSYNC时间戳，每次VSYNC同步时总会有一个消息发出。客户端在指定的时间间隔会收到VSYNC时间戳，你必须用最大1毫秒的延迟来实现垂直同步（推荐0.5毫秒或更少）；

显式同步

显示同步是必须的，它提供了一个以同步方式持有和释放Gralloc buffers的方式。显示同步允许graphics buffers的生产者和消费者在处理完一个buffer的时候发送一个信号。这种机制允许安卓进入队列的buffers可以被异步读写，

此时这些buffers不会被其余的第三者消费者或生产者需要。

这个显示同步有很多优点比如:设备之间的行为变化更少,更好的调试支持，改进的测试指标。例如，同步框架更容易找出问题根源，并在系统正常流事件发生时集中表现surfaceflinger时间戳。

这种通信得益于synchronization fences的使用，当请求一个buffer用来消费或者生产的时候synchronization fences是会被使用的。同步框架有三个主要模块：sync\_timeline，sync\_pt，和sync\_fence。

sync\_timeline

一个sync\_timeline是一个单调递增的时间线，这个时间线会被每一个驱动实例使用，比如，一个GL context, display controller, or 2D blitter.本质上这是一个要提交到内核特定部位的一系列工作。它保证操作顺序和硬件可以实现。

这个sync\_timeline是提供一个CPU-only的参考实现，这个实现被称作sw\_sync(software sync)。如果可能的话，用它代替sync\_timeline以便节省资源和避免复杂性。如果你不使用硬件资源，sw\_sync应该足够了。

如果你要实现一个sync\_timeline，使用sw\_sync驱动。遵循这些准则：

为所有drivers, timelines, 和 fences提供有用的名称。这简化了调试。

在你的timelines实现timeline\_value\_str和pt\_value\_str操作让调试信息更加可读。

如果你想让你的用户空间库（如GL库）来访问你的timelines的私有数据，那么你可以填充driver\_data算子。这可以让你获得不会改变的sync\_fence和sync\_pts对象,建立基于命令行的命令。

当实现一个sync\_timeline请不要:

建立在任何真正的时间轴，比如墙上的挂钟。最好是创建一个你可以控制抽象的时间轴。

允许用户显式地创建或发送fence。这会导致一个用户管道产生拒绝服务攻击从而导致停止掉了所有的功能。这是因为用户不能对内核的稳定负责。

不要去访问sync\_timeline，sync\_pt或者sync\_fence元素，需要的话请使用相关API.

sync\_pt

一个sync\_pt是sync\_timeline的一个点，它有三个状态:active, signaled, and error. 它开始于active状态过度到signaled或error状态.例如，当一个buffer不再被一个image consumer需要，这个点会发出信号让image producers知道再一次写入这个buffer是可以的.

sync\_fence

一个sync\_fence是一个sync\_pts的集合,这个集合通常有不同的sync\_timeline父亲(比如display controller and GPU),这些是主要的驱动程序和用户空间通信原始的依赖关系。fence是内核保证能按时完成一个进入队列的一个工作的承诺。

它允许多个消费者或生产者发出信号说明他们正在使用一个buffer ，并允许这些信息与一个函数的参数进行通信。Fences由文件描述符支持，可以从内核空间传递到用户空间。比如说，一个fence可以包含两个sync\_points，这表示两个单独的image consumers正在读取同一个buffer。当fence发出信号时，image producers知道消费者都在消费。

Fences，像sync\_pts，开始的状态是active随后sync\_pts根据自己的点的状态来进行调整状态。如果所有的sync\_pts变成signaled状态，那么sync\_fence也会变成signaled状态。如果有一个sync\_pt变成error状态，整个sync\_fence也会变成error状态。

sync\_fence的成员在fence创建完成后是不可以改变的。一个sync\_pt只可在一个fence里面，作为一个副本。即使两个points有相同的值，那么也将有两份sync\_pt在fence里面。要在fence中获得一个以上的point，就要进行合并操作，从两个不同的fences把points添加到第三个fence。如果原始的fence里面有一个是在signaled状态，另一个则不是，被合成的第三个fence也不会处于signaled状态。

若要实现显式同步，请提供以下文件：

实现特定硬件的同步时间轴的内核空间驱动器。需要fence-aware的驱动程序通常是为了与Hardware Composer进行访问或通信。关键文件包括：

核心文件:

kernel/common/include/linux/sync.h

kernel/common/drivers/base/sync.c

sw\_sync:

kernel/common/include/linux/sw\_sync.h

kernel/common/drivers/base/sw\_sync.c

文档在kernel/common//Documentation/sync.txt.

和kernel-space通信的库在platform/system/core/libsync.

Hardware Composer HAL模块（v1.3或更高），支持新的同步功能。你必须提供synchronization fences作为HAL里面set()和prepare()函数的参数。

两个fence-related GL扩展（EGL\_ANDROID\_native\_fence\_sync 和 EGL\_ANDROID\_wait\_sync）并且你的图形驱动程序支持fence。

例如，使用支持同步功能的API，可以开发具有显示buffer功能的显示驱动程序。在同步框架存在之前，这个函数会接收dma-bufs，把这些buffers显示，如果buffer是可见的那么这个函数处于block状态。例如:

/\*

\* assumes buf is ready to be displayed.  returns when buffer is no longer on

\* screen.

\*/

voiddisplay\_buffer(structdma\_buf\*buf);

使用同步框架，API调用稍微复杂一些。在将buffer放在显示器上时，您可以将它与buffer准备就绪时的fence关联起来。在清除fence之后你可以把任务放入队列。

以这种方式，你不会阻塞任何东西。您立即返回您自己的fence，这是保证buffer从显示器消失。当buffers进入队列时，内核将列出同步框架的依赖项：

/\*

\* will display buf when fence is signaled.  returns immediately with a fence

\* that will signal when buf is no longer displayed.

\*/

structsync\_fence\*display\_buffer(structdma\_buf\*buf,structsync\_fence

\*fence);

Sync 集成

本节介绍如何将Android不同部分的低级别同步框架和必须相互通信的驱动程序集成在一起。

整合规范

对于图形的Android HAL接口遵循一致的约定，所以当文件描述符在HAL接口上传递时，文件描述符的所有权总是被转移的。这意味着：

如果从同步框架接收到一个fence文件描述符，则必须关闭它。

如果将一个fence文件描述符返回到同步框架，框架将关闭它。

要继续使用fence文件描述符，必须复制描述符。

每一次，一个fence传递给BufferQueue（如一个窗口通过传递fence给BufferQueue来说明自己新的内容已经准备好）fence对象会被重命名。内核支持fences用字符串表示名称，同步框架使用的窗口名称和正在进入队列的buffer index命名这个fence（比如,SurfaceView:0）。如果这个名字出现在/d/sync形式的log里或bugreports，那么对定位死锁问题的原因非常重要。

ANativeWindow集成

ANativeWindow是fence一体化，即dequeuebuffer，queuebuffer，和cancelbuffer 包含fence的引用。

OpenGL ES的集成

OpenGL ES同步整合依赖于两个EGL扩展：

EGL\_ANDROID\_native\_fence\_sync.提供了一种以EGLSyncKHR对象的方式来创建原生的Android fence文件描述符。

EGL\_ANDROID\_wait\_sync.允许GPU而不是CPU等待，让GPU等待一个EGLSyncKHR。本质上是和EGL\_KHR\_wait\_sync扩展是相同的（指规范细节）。

这些扩展可以独立使用，被在libgui里面的编译标志位控制。要使用它们，首先实现EGL\_ANDROID\_native\_fence\_sync扩展来获得内核的相关支持。下一步，添加一个对于fences支持的ANativeWindow驱动程序，然后打开libgui的编译标志位来确保使用EGL\_ANDROID\_native\_fence\_sync扩展。

第二步，打开EGL\_ANDROID\_wait\_sync扩展并把它分开。EGL\_ANDROID\_native\_fence\_sync扩展由不同的原生fence EGLSync对象类型组成，因此正在应用于EGLSync对象的类型不需要应用于EGL\_ANDROID\_native\_fence对象类型以避免不必要的相互影响。

EGL\_ANDROID\_native\_fence\_sync扩展采用相应的原生fence文件描述符属性。这种属性只能设置在创建时不能直接查询到一个现有的同步对象。此属性可以设置为两种模式之一：

有效的fence文件描述符。把原生Android fence 文件描述符封装为EGLSyncKHR 对象。

- 1.从EGLSyncKHR 对象创建一个原生的Android fence文件描述符.

DupNativeFenceFD函数的作用是从原生Android fence文件描述符提取EGLSyncKHR对象。这与查询被设置的属性有相同的结果，但遵守接收者关闭fence的约定（会重复操作）。最后，销毁EGLSync对象时应该关闭内部fence属性。

Hardware Composer集成

The Hardware Composer 处理下面三种类型的sync fences:

Acquire fence.每一个层，在调用 HWC::set 之前进行设置。当Hardware Composer读取buffer的时候它变为signals 模式。

Release fence.每一个层，在HWC::set调用后被填满。当Hardware Composer读取buffer完毕，它会变成signals 状态，以便framework可以给特定的层再次使用该buffer。

Retire fence. 对于每一个完整的层框架，HWC::set 每次被调用都会被填满。所有层的HWC::set 操作全部完成后它会变成signals状态并通知framework 。在下一次HWC::set操作的结果显示在屏幕上的时候，retire fence将变为signals状态。

垂直同步偏移（VSYNC offset）

APP和SurfaceFlinger循环渲染过程应该和hardware VSYNC同步。hardware VSYNC事件发生时顺序是display显示frame N，SurfaceFlinger合成frame N+1，app生成frame N+2。

Synchronizing with VSYNC 发出一致的延时。这样可以减少APP和SurfaceFlinger和display彼此的误差。假定APP和SurfaceFlinger的每帧时间损耗比较固定，那么即使这样，延时至少也是两帧。

为了解决这个问题，你可以采用VSYNC offsets减少input-to-display的延时通过让APP和composition signal与hardware VSYNC进行比较。这是可能的，因为app加composition的过程通常需要少于33ms。

VSYNC offset的结果是三个信号同一段时间偏移相同：

HW\_VSYNC\_0. Display开始显示下一个frame.

VSYNC. App读取输入开始展示下一个frame.

SF VSYNC. SurfaceFlinger开始合成下一个frame.

VSYNC offset，SurfaceFlinger接收buffer、合成frame，APP处理事件渲染frame，都发生在一个frame的时间周期内。

注意:VSYNC offsets减少了APP和composition的可利用时间从而减少了错误的发生。

DispSync

DispSync保持了一个定期的hardware-based VSYNC显示模型。这个模型保证了执行来自hardware VSYNC的定期回调函数。

DispSync是一个software phase lock loop (PLL) 。它生成Choreographer使用的VSYNC signals 和SurfaceFlinger使用的SF VSYNC信号也包括hardware VSYNC没有偏移（offset ）的情况。

图一.DispSync 流程

DispSync具有下面的特点:

Reference. HW\_VSYNC\_0.

Output. VSYNC and SF VSYNC.

Feedback. Retire fence signal timestamps from Hardware Composer.

VSYNC/Retire offset

retire fences signal 时间戳需要匹配HW VSYNC即使设备没有使用

offset phase。否则会出现严重错误。

三角模型里面Retire fence是direct memory access (DMA)显存显示的最后阶段，但实际display switch and HW VSYNC有时候会在后面。

PRESENT\_TIME\_OFFSET\_FROM\_VSYNC\_NS宏被设置在设备的BoardConfig.mk文件里。它是基于显示控制器和面板特性。这个值代表的是retire fence和HW VSYNC signal的纳秒级别的时间差。

VSYNC and SF\_VSYNC offsets

VSYNC\_EVENT\_PHASE\_OFFSET\_NS和SF\_VSYNC\_EVENT\_PHASE\_OFFSET\_NS高负荷的使用情况下的临界值，如部分GPU在窗口平移或者滚动页面的情況下。这些offsets允许更长的应用程序渲染时间和更长的GPU合成时间。

​

超过1ms或者两个延迟是明显的。我们建议减少延迟，而不是增加错误次数。

注意:这两个offsets也被配置在了设备的BoardConfig.mk文件里面。表示和

HW\_VSYNC\_0时间戳的差单位是纳秒，如果这两个值不被赋值那么默认是0，也可以被设置为负数。

作者：一只像汤圆的大崽  
链接：<https://www.jianshu.com/p/898af344bcd1>

## 源码路径

frameworks/native/services/surfaceflinger/EventThread.cpp

# EventThread

为什么要先说EventThread? 很奇怪是吧，图2 PLL图 明明是SW vsync将vsync信号传给 VSYNC-sf/VSYNC-app的，怎么还先讲结果了呢？而不先讲DispThread呢？

因为前面所说的4个线程互相影响，且是并行进行的，所以要想用一篇文章(单线程)来很顺利的写清楚(多线程的过程)，而还要交待清楚前因后果，非常考验这个作者的水平。所以第二节先说 EventThread 是为了写好 DispSync 作铺垫的。

由于SF EventThread和APP EventThread是同一套代码, 而SF EventThread先运作起来，所以下面以SF EventThread为例作介绍.

## 2.1 EventThread的初始化

frameworks/native/services/surfaceflinger/SurfaceFlinger\_hwc1.cpp

**void** SurfaceFlinger::init() {

*// start the EventThread*sp<VSyncSource> vsyncSrc = **new** DispSyncSource(&mPrimaryDispSync,  
 vsyncPhaseOffsetNs, **true**, **"app"**);  
mEventThread = **new** EventThread(vsyncSrc, \***this**);  
sp<VSyncSource> sfVsyncSrc = **new** DispSyncSource(&mPrimaryDispSync,  
 sfVsyncPhaseOffsetNs, **true**, **"sf"**);  
mSFEventThread = **new** EventThread(sfVsyncSrc, \***this**);  
mEventQueue.setEventThread(mSFEventThread);

}

如上面所示，生成两个EventThread，一个是APP EventThread, 一个是SF EventThread.它们的区别在于相移phase offset不同，

**static const** int64\_t vsyncPhaseOffsetNs = VSYNC\_EVENT\_PHASE\_OFFSET\_NS;  
*// This is the phase offset at which SurfaceFlinger's composition runs.***static const** int64\_t sfVsyncPhaseOffsetNs = SF\_VSYNC\_EVENT\_PHASE\_OFFSET\_NS;

| **EventThread** | **相移** |
| --- | --- |
| App | VSYNC\_EVENT\_PHASE\_OFFSET\_NS |
| SF | SF\_VSYNC\_EVENT\_PHASE\_OFFSET\_NS |

这两个值都可配，这两个一般用来调节performance. 具体可在 **BoardConfig.mk**里配置

frameworks/native/services/surfaceflinger/Android.mk也有

LOCAL\_CFLAGS += -DSF\_VSYNC\_EVENT\_PHASE\_OFFSET\_NS=1000000

**void** EventThread::onFirstRef() {  
 run(**"EventThread"**, PRIORITY\_URGENT\_DISPLAY + PRIORITY\_MORE\_FAVORABLE);  
}

**bool** EventThread::threadLoop() {  
 signalConnections = waitForEvent(&event); //阻塞式的等待事件发生  
 *// dispatch events to listeners...* **const** size\_t count = signalConnections.size();  
 **for** (size\_t i=0 ; i<count ; i++) {  
 **const** sp<Connection>& conn(signalConnections[i]);  
 *// now see if we still need to report this event* status\_t err = conn->postEvent(event);  
 }  
 **return true**;  
}

sp指针是生成对象结束后会调用onFirstRef.接着又调用Thread的run函数，线程就一直开始反复调用threadLoop.从threadLoop大致可以猜测出来，先等着事件发生(这里也就是vsync事件)，然后将vsync事件分发出去，不同的EventThread(SF/APP EventThread)作的事情就开始不同了。

### 2.2.1接着看 waitForEvent()

Vector< sp<EventThread::Connection> > EventThread::waitForEvent(  
 DisplayEventReceiver::Event\* event)  
{  
 Mutex::Autolock \_l(mLock);  
 Vector< sp<EventThread::Connection> > signalConnections;  
  
 **do** {  
 bool eventPending = **false**;  
 bool waitForVSync = **false**;  
  
 size\_t vsyncCount = 0;  
 nsecs\_t timestamp = 0;  
  
 **for** (int32\_t i=0 ; i<DisplayDevice::NUM\_BUILTIN\_DISPLAY\_TYPES ; i++) {  
 timestamp = mVSyncEvent[i].header.timestamp;  
 **if** (timestamp) {  
 *//如果这时从 mVSyncEvent里取得的timestamp大于0，表明这时已经有vsync事件待发送  
 // mVSyncEvent就是保存VSYNC信号的变量* \*event = mVSyncEvent[i];  
 mVSyncEvent[i].header.timestamp = 0; *//为什么要置为0呢？？* vsyncCount = mVSyncEvent[i].vsync.count;  
 **break**;  
 }  
 }  
  
 **if** (!timestamp) {  
 *//没有vsync事件， 来看下是否有其它pending的event, 这里主要是hotplug的事件* eventPending = !mPendingEvents.isEmpty();  
 **if** (eventPending) {  
 *// we have some other event to dispatch* \*event = mPendingEvents[0];  
 mPendingEvents.removeAt(0);  
 }  
 }  
  
 *// mDisplayEventConnections保存的是注册的Connection的,  
 // SF EventThread线程里只有一个Connection, 而这个Connection主要是用来渲染  
 // 而如果是APP EventThread, 这里会有多个connection* size\_t count = mDisplayEventConnections.size();  
 **for** (size\_t i=0 ; i<count ; i++) {  
 sp<Connection> connection(mDisplayEventConnections[i].promote());  
 **if** (connection != NULL) {  
 bool added = **false**;  
 *//这里的connection->count的值的大小有如下含义  
 // count >= 1 : continuous event. count is the vsync rate 如果在大于等于1，表示会持续接收vsync event  
 // count == 0 : one-shot event that has not fired 表示只接收一次  
 // count ==-1 : one-shot event that fired this round / disabled 等于-1，表示不能再接收vsync事件了* **if** (connection->count >= 0) { *//只能对还能接收的connection进行处理  
 // we need vsync events because at least  
 // one connection is waiting for it* waitForVSync = **true**; *//这个变量后面会用到* **if** (timestamp) {  
 *// we consume the event only if it's time  
 // (ie: we received a vsync event)* **if** (connection->count == 0) { *//如定义一样，如果是一次性的，那么在获得本次vsync后，将它的count置为-1了, 下次只能通过 requestNextVsync 来重置为0  
 // fired this time around* connection->count = -1;  
 signalConnections.add(connection); *//最外层的while判断条件会用到* added = **true**;  
 } **else if** (connection->count == 1 ||  
 (vsyncCount % connection->count) == 0) {  
 *// continuous event, and time to report it* signalConnections.add(connection);  
 added = **true**;  
 }  
 }  
 }  
  
 **if** (eventPending && !timestamp && !added) {  
 *// 此时没有vsync事件，但是有pending的事件，那不管connection是否能接收了  
 // messages.* signalConnections.add(connection);  
 }  
 } **else** {  
 *// we couldn't promote this reference, the connection has  
 // died, so clean-up!* mDisplayEventConnections.removeAt(i);  
 --i; --count;  
 }  
 }  
  
 *// Here we figure out if we need to enable or disable vsyncs* **if** (timestamp && !waitForVSync) {  
 *// vsync事件已经发生了，但是我都还没有client去监听，那么这时你再继续发vsync根本就是多余的  
 // 所以直接disable Vsync, 注意这里并不是真正的disable硬件的VSYNC信号，见下面的分析* disableVSyncLocked();  
 } **else if** (!timestamp && waitForVSync) {  
 *// 有client在监听了，但是还没有vsync事件，那么是否是之前vsync被disable了呢？  
 //就要打开vsync监听，* enableVSyncLocked();  
 }  
   
 **if** (!timestamp && !eventPending) {*//既没有vsync事件，也没有其它pending的事件(hotplug事件)  
 // wait for something to happen* **if** (waitForVSync) { *//但是有client在监听了，这时就等着上报vsync事件即可  
 // This is where we spend most of our time, waiting  
 // for vsync events and new client registrations.  
 //  
 // If the screen is off, we can't use h/w vsync, so we  
 // use a 16ms timeout instead. It doesn't need to be  
 // precise, we just need to keep feeding our clients.  
 //  
 // We don't want to stall if there's a driver bug, so we  
 // use a (long) timeout when waiting for h/w vsync, and  
 // generate fake events when necessary.* bool softwareSync = mUseSoftwareVSync; *//这里只考虑硬件vsync的情况,软件模拟的暂时不考虑* nsecs\_t timeout = softwareSync ? ms2ns(16) : ms2ns(1000);  
 *//如注释所说的，如果是driver的bug,如果硬件一直不上报vsync事件怎么办？？难道就一直等下去？？那client不就饿死了么？  
 //所以这里如果driver不报vsync，那么就软件模拟一个vsync事件，这里的timeout是1000ms，发一个* **if** (mCondition.waitRelative(mLock, timeout) == TIMED\_OUT) {  
 **if** (!softwareSync) {  
 ALOGW(**"Timed out waiting for hw vsync; faking it"**);  
 }  
 *//* ***FIXME: how do we decide which display id the fake*** *// vsync came from ?* mVSyncEvent[0].header.type = DisplayEventReceiver::DISPLAY\_EVENT\_VSYNC;  
 mVSyncEvent[0].header.id = DisplayDevice::DISPLAY\_PRIMARY;  
 mVSyncEvent[0].header.timestamp = systemTime(SYSTEM\_TIME\_MONOTONIC);  
 mVSyncEvent[0].vsync.count++;  
 }  
 } **else** {  
 *// Nobody is interested in vsync, so we just want to sleep.  
 // h/w vsync should be disabled, so this will wait until we  
 // get a new connection, or an existing connection becomes  
 // interested in receiving vsync again.  
 //既没有client, 又没有硬件vsync事件，那么就死等下去* mCondition.wait(mLock);  
 }  
  
 }  
 } **while** (signalConnections.isEmpty());  
  
 *// here we're guaranteed to have a timestamp and some connections to signal  
 // (The connections might have dropped out of mDisplayEventConnections  
 // while we were asleep, but we'll still have strong references to them.)* **return** signalConnections;  
}

#### 创建Connection

当初始化完SF EventThread后，就开始创建SF Connection了。**入口**

mEventQueue.setEventThread(mSFEventThread);

**创建Connection，加入回调函数**

**void** MessageQueue::setEventThread(**const** sp<EventThread>& eventThread)  
{  
 mEventThread = eventThread;  
 mEvents = eventThread->createEventConnection();  
 mEventTube = mEvents->getDataChannel();  
 mLooper->addFd(mEventTube->getFd(), 0, Looper::EVENT\_INPUT,  
 **MessageQueue**::cb\_eventReceiver, **this**);  
}  
sp<EventThread::Connection> EventThread::createEventConnection() **const** {  
 **return new** Connection(const\_cast<EventThread\*>(**this**));  
 *//初始化的Connection的count都为-1，即刚开始的时候，connection都不会接收vsync事件*}

#### **注册Connection**

**void** EventThread::Connection::onFirstRef() {  
 *// NOTE: mEventThread doesn't hold a strong reference on us* mEventThread->registerDisplayEventConnection(**this**);  
}  
status\_t EventThread::registerDisplayEventConnection(   
 **const** sp<**EventThread**::Connection>& connection) {  
 Mutex::Autolock \_l(mLock);  
 mDisplayEventConnections.add(connection);  
 *//加入要SF EventThread里的mDisplayEventConnections里* mCondition.broadcast(); *//并释放mCondition* **return** NO\_ERROR;  
}

mCondition.broadcast()会唤醒之前的mCondition.wait()，但是在**waitForEvent**的while循环为false,再做while一次循环

这时候 timestamp还是为0，还是没有pending的event, 但是这时有SF的connection了，只不过此时connection的count仍然为默认的-1,

最后还是进入 mCondition.wait死等.

**注意:** 实际在调试的时候 registerDisplayEventConnection会比SF EventThread的threadLoop先运行起来，不过最后的结果是一样的。

由第4步可知Connection的初始化count为-1，即表示该Connection不会接收vsync事件，那么这个值是在什么地方被修改的呢？

答案是在SurfaceFlinger初始化的最后initializeDisplays里

### requestNextVsync

initializeDisplays();

flinger->onInitializeDisplays();

setTransactionState(state, displays, 0);

setTransactionFlags(transactionFlags);

signalTransaction();

EventQueue.invalidate();

mEvents->requestNextVsync() //mEvents是Connection实例

EventThread->requestNextVsync(this);

requestNextVsync表示主动去请求获得vsync事件, 上面的意思是将Display初始化后，即显示屏可以工作后，那么SF EventThread就开始要监听vsync事件了。

**void** EventThread::requestNextVsync(  
 **const** sp<**EventThread**::Connection>& connection) {  
 Mutex::Autolock \_l(mLock);  
  
 mFlinger.resyncWithRateLimit();  
  
 **if** (connection->count < 0) {  
 connection->count = 0; *//这里将SurfaceFlinger的Count改为0，变成一次性接收的了* mCondition.broadcast(); *//释放EventThread里的mCondition* }  
}

a) requestNextVsync释放EventThread里的mCondition后,接着会唤醒 EventThread里的上面第5步的mCondition.wait, 这时会再走一遍while循环

b). 这时候timestamp还是为0，还是没有pending的event, 但是这时有SF的connection了, 且此时的connection的count已经被置为了0，表明此时有connection在监听了，即waitForVSync为true

c) 因此执行waitforEvent，满足执行 enableVSyncLocked

**else if** (!timestamp && waitForVSync) {enableVSyncLocked();  
}

d) 继续在waitforEvent中执行mCondition.waitRelative(), 其中超时时间为1000ms

**bool** softwareSync = mUseSoftwareVSync;  
 nsecs\_t timeout = softwareSync ? ms2ns(16) : ms2ns(1000);  
 **if** (mCondition.waitRelative(mLock, timeout) == TIMED\_OUT) {  
 **if** (!softwareSync) {  
 ALOGW(**"Timed out waiting for hw vsync; faking it"**);  
 }  
 *//* ***FIXME: how do we decide which display id the fake*** *// vsync came from ?* mVSyncEvent[0].header.type = DisplayEventReceiver::DISPLAY\_EVENT\_VSYNC;  
 mVSyncEvent[0].header.id = DisplayDevice::DISPLAY\_PRIMARY;  
 mVSyncEvent[0].header.timestamp = systemTime(SYSTEM\_TIME\_MONOTONIC);  
 mVSyncEvent[0].vsync.count++;  
 }

那么 enableVSyncLocked 这个函数又是干什么的呢？

### enableVSyncLocked

**void** EventThread::enableVSyncLocked() {  
 **if** (!mUseSoftwareVSync) {  
 *// never enable h/w VSYNC when screen is off* **if** (!mVsyncEnabled) { *//这里只考虑硬件vsync的情况，而不考虑软件模拟的情况* mVsyncEnabled = **true**;  
 mVSyncSource->setCallback(static\_cast<VSyncSource::Callback\*>(**this**));  
 mVSyncSource->setVSyncEnabled(**true**);  
 }  
 }  
 mDebugVsyncEnabled = **true**;  
 sendVsyncHintOnLocked();  
}

这里只考虑硬件vsync的情况，即mUseSoftwareVSync为false的情况，最后调用到 setVsyncEnabled, 且其值为true

frameworks/native/services/surfaceflinger/SurfaceFlinger\_hwc1.cpp

sf

virtual **void** setVSyncEnabled(bool enable) {  
 Mutex::Autolock lock(mVsyncMutex);  
 **if** (enable) {  
 *// 将EventListener最终加入到DispSyncThread的mEventListeners里* status\_t err = mDispSync->addEventListener(mName, mPhaseOffset,  
 static\_cast<DispSync::Callback\*>(**this**));  
 **if** (err != NO\_ERROR) {  
 ALOGE(**"error registering vsync callback: %s (%d)"**, strerror(-err), err);  
 }  
 *//ATRACE\_INT(mVsyncOnLabel.string(), 1);* } **else** {  
 *//相反如果 enable 为false时，那么就从EventListeners里删除掉* status\_t err = mDispSync->removeEventListener(static\_cast<DispSync::Callback\*>(**this**));  
 **if** (err != NO\_ERROR) {  
 ALOGE(**"error unregistering vsync callback: %s (%d)"**,strerror(-err), err);  
 }  
 *//ATRACE\_INT(mVsyncOnLabel.string(), 0);* }  
 mEnabled = enable;  
}

status\_t addEventListener(**const char**\* name, nsecs\_t phase,  
 **const** sp<DispSync::Callback>& callback) {  
 **if** (kTraceDetailedInfo) ATRACE\_CALL();  
 Mutex::Autolock lock(mMutex);  
  
 **for** (size\_t i = 0; i < mEventListeners.size(); i++) {  
 **if** (mEventListeners[i].mCallback == callback) {  
 **return** BAD\_VALUE;  
 }  
 }  
  
 EventListener listener;  
 listener.mName = name;  
 listener.mPhase = phase;  
 listener.mCallback = callback;  
  
 *// listener里的mLastEventTime这个在这里初始化的意义是防止之前的VSYNC事件被发送出去了  
 // We want to allow the firstmost future event to fire without  
 // allowing any past events to fire* listener.mLastEventTime = systemTime() - mPeriod / 2 + mPhase - mWakeupLatency;  
  
 mEventListeners.push(listener);  
 *//DispSyncThread的 mCond释放* mCond.signal();  
  
 **return** NO\_ERROR;  
}

主要是为DispSyncThread添加EventListener, 那下面这节就是为DispSyncThread设置Peroid. 这样DispSync模型就可以动作起来了。

# 开关硬件HWC

在SurfaceFlinger初始化Display后，会调用resyncToHardwareVsync跟硬件vsync进行同步

initializeDisplays();

flinger->onInitializeDisplays();

setPowerModeInternal()

resyncToHardwareVsync(true);

repaintEverything();

## resyncToHardwareVsync函数

**void** SurfaceFlinger::resyncToHardwareVsync(bool makeAvailable) {  
 Mutex::Autolock \_l(mHWVsyncLock);  
  
 **if** (makeAvailable) {  
 *// mHWVsyncAvailable表示HW vsync被enable了* mHWVsyncAvailable = **true**;  
 } **else if** (!mHWVsyncAvailable) {  
 *// Hardware vsync is not currently available, so abort the resync  
 // attempt for now* **return**;  
 }  
  
 *//获得显示设备的刷新率，比如60HZ, 那么period就是16.6667ms,即每隔16.6667就会产生一个硬件vsync信号***const** nsecs\_t period =  
 getHwComposer().getRefreshPeriod(HWC\_DISPLAY\_PRIMARY);  
 *//当前这个值跟具体的显示设备有关，并不一定是60HZ* mPrimaryDispSync.reset();  
 *//设置DispSync模型里period为显示设备的频率* mPrimaryDispSync.setPeriod(period);  
  
 *//mPrimaryHWVsyncEnabled表示当前的硬件vsync是否enable,* **if** (!mPrimaryHWVsyncEnabled) {  
 mPrimaryDispSync.beginResync();  
 *//如果硬件vsync没有enable,那么就通知EventControlThread去通知硬件enable VSYNC，这个和DispSync的setVsyncEnabled是不一样的* mEventControlThread->setVsyncEnabled(**true**);  
 mPrimaryHWVsyncEnabled = **true**;  
 }  
}

## setPeriod 更新mPeriod

mPrimaryDispSync.setPeriod(period);

void DispSync::setPeriod(nsecs\_t period) {

Mutex::Autolock lock(mMutex);

mPeriod = period;

mPhase = 0;

mReferenceTime = 0;

mThread->updateModel(mPeriod,mPhase,mReferenceTime);

}

**mPeriod**表示具体的硬件产生vsync的时间间隔  
**mThread**是DispSyncThread， DispSync在初始化的时候直接生成一个线程DispSyncThread并运行起来

**void** updateModel(nsecs\_t period, nsecs\_t phase, nsecs\_t referenceTime) {  
 **if** (kTraceDetailedInfo) ATRACE\_CALL();  
 Mutex::Autolock lock(mMutex);  
 mPeriod = period;  
 mPhase = phase;  
 mReferenceTime = referenceTime;  
 mCond.signal();  
}

updateModel里会再次唤醒 DispSyncThread的里的 mCond, 注意此时 mPeroid已经不为0了。

# 硬件Vsync的控制

## 4.1 默认开闭硬件vsync

SurfaceFlinger在初始化HWComposer时会默认关闭硬件Vsync信号，这里直接调用eventControl.  
具体代码如下

**HWComposer**::HWComposer() {  
 eventControl(HWC\_DISPLAY\_PRIMARY, HWC\_EVENT\_VSYNC, 0);  
}  
  
**void** HWComposer::eventControl(**int** disp, **int** event, **int** enabled) {  
 err = mHwc->eventControl(mHwc, disp, event, enabled);  
}

mHwc是hwc\_composer\_device\_1类型，它表示是对一个硬件设备的抽象吧，通过它就可以控制和使用硬件相关功能吧。

那么硬件的Vsync是在什么时候被打开的呢?

## 4.2 打开硬件vsync

具体是在**3.1 resyncToHardwareVsync** 函数最后的代码打开的。  
resyncToHardwareVsync函数从字面上看来就是和硬件的Vsync进行同步的意思。

**if** (!mPrimaryHWVsyncEnabled) {  
 mPrimaryDispSync.beginResync();  
 *//如果硬件vsync没有enable,那么就通知EventControlThread去通知硬件enable VSYNC，  
 //这个和DispSync的setVsyncEnabled是不一样的* mEventControlThread->setVsyncEnabled(**true**);  
 mPrimaryHWVsyncEnabled = **true**;  
}

resyncToHardwareVsync函数通过EventControlThread去控制硬件Vsync信号的开关

**void** EventControlThread::setVsyncEnabled(bool enabled) {  
 Mutex::Autolock lock(mMutex);  
 mVsyncEnabled = enabled; *// mVsyncEnabled一个控制开关* mCond.signal(); *//释放EventControlThread里的mCond信号*}

setVsyncEnabled会释放mCond信号，这样在EventControlThread的threadLoop里的mCond会被唤醒去操作硬件Vsync开关了

**bool** EventControlThread::threadLoop() {  
 Mutex::Autolock lock(mMutex);  
  
 **bool** vsyncEnabled = mVsyncEnabled;  
  
#ifdef USE\_HWC2  
 mFlinger->setVsyncEnabled(HWC\_DISPLAY\_PRIMARY, mVsyncEnabled);  
#else  
 mFlinger->eventControl(HWC\_DISPLAY\_PRIMARY, SurfaceFlinger::EVENT\_VSYNC,  
 mVsyncEnabled);  
#endif  
  
 **while** (**true**) {  
 status\_t err = mCond.wait(mMutex);  
 **if** (err != NO\_ERROR) {  
 ALOGE(**"error waiting for new events: %s (%d)"**,  
 strerror(-err), err);  
 **return false**;  
 }  
  
 **if** (vsyncEnabled != mVsyncEnabled) {  
#ifdef USE\_HWC2  
 mFlinger->setVsyncEnabled(HWC\_DISPLAY\_PRIMARY, mVsyncEnabled);  
#else  
 mFlinger->eventControl(HWC\_DISPLAY\_PRIMARY,  
 SurfaceFlinger::EVENT\_VSYNC, mVsyncEnabled);  
#endif  
 vsyncEnabled = mVsyncEnabled;  
 }  
 }  
  
 **return false**;  
}

好了，经过三节的铺垫终于可以说下DispSync.

# DispSync模型

DispSync 是定义在SurfaceFlinger类里的成员变量，因此在初始化 SurfaceFlinger时，就会初始化DispSync， 它在SurfaceFlinger里的具体定义是

**DispSync mPrimaryDispSync**

而DispSync在初始化的时候会生成 DispSyncThread 线程，紧接着将 DispSyncThread run起来，根据C++ Thread模型， DispSyncThread 会循环调用threadLoop() 函数。

下面来看下 DispSyncThread 里的 threadLoop()函数, 之所以把它的所有代码粘贴上来，是方便以后回顾之用

## 5.1 DispSync模型运作

### 5.1.1 等待可用的EventListener

frameworks/native/services/surfaceflinger/DispSync.cpp

virtual bool threadLoop() {  
 status\_t err;  
 nsecs\_t now = systemTime(SYSTEM\_TIME\_MONOTONIC);  
 *//获得当前的系统时间，这个是比较老的时间了* **while** (**true**) {  
 Vector<CallbackInvocation> callbackInvocations;  
  
 nsecs\_t targetTime = 0;  
  
 { *// Scope for lock* Mutex::Autolock lock(mMutex);  
  
 **if** (kTraceDetailedInfo) {  
 ATRACE\_INT64(**"DispSync:Frame"**, mFrameNumber);  
 }  
 ALOGV(**"[%s] Frame %"** PRId64, mName, mFrameNumber);  
 ++mFrameNumber;  
 *//mFrameNumber仅仅是一个计数而已，没有实际用处，它和vsync个数是不等同的* **if** (mStop) {  
 **return false**;  
 }  
  
 *//当threadLoop第一次进来后，由于mPeriod初始化为0，所以一直死等在这里* **if** (mPeriod == 0) {  
 err = mCond.wait(mMutex); *// \*\*blockingA\*\** **if** (err != NO\_ERROR) {  
 ALOGE(**"error waiting for new events: %s (%d)"**, strerror(-err), err);  
 **return false**;  
 }  
 **continue**;  
 }

targetTime = computeNextEventTimeLocked(now);  
 ...  
 }  
 }

当threadLoop第一次运行，mPeriod初始化为0，所以一直死等在**"blockingA"**处。

### 5.1.2 往DispSyncThread里加入EventListener

具体是在 setVSyncEnabled里，**参考 2.2.3 enableVSyncLocked**  
**setVSyncEnabled** 将 EventListener添加到 DispSync 里的mEventListeners里，然后释放mCond.signal(), 继而mCond会唤醒 5.1.1 中的 [**blockingA**](#dispsync_firstrun).

注意此时 mPeriod 依然为0，所以线程也一直死等在**"blockingA"** 处, 但是DispSyncThread的 mEventListeners 已经加入了listener了。

### 5.1.3 DispSyncThread收到mPeriod更新

由5.1.2可知，由于mPeriod为0，所以线程一直死等在**blockingA**处，  
而由**3.2 setPeriod**可知，此时mPeriod已经被更新成显示设备的刷新率了，且 mCond已经被释放了，因此 **blockingA** mCond.wait()被唤醒了。

这时进入threadLoop的第二阶段，计算下一个Vsync信号的时间戳，并且上报给EventListener. 就这样，DispSyncThread模型就运作起来了。

## 5.2 更新DispSync模型

由 4.2 小节可知，硬件Vsync已经在resyncToHardwareVsync被打开了，既然打开了，那么只要有硬件Vsync信号产生，就可回调 hook\_vsync函数(hook\_vsync函数在HWComposer的初始化的时候被注册的)

### 5.2.1 hook\_vsync的回调函数

**void** HWComposer::hook\_vsync(**const** struct hwc\_procs\* procs, **int** disp,  
 int64\_t timestamp) {  
 cb\_context\* ctx = reinterpret\_cast<cb\_context\*>(  
 const\_cast<hwc\_procs\_t\*>(procs));  
 ctx->hwc->vsync(disp, timestamp);  
}

**具体调用到HWComposer的vsync**

**void** HWComposer::vsync(**int** disp, int64\_t timestamp) {  
 **if** (uint32\_t(disp) < HWC\_NUM\_PHYSICAL\_DISPLAY\_TYPES) {  
 {  
 Mutex::Autolock \_l(mLock);  
  
 *// 防止重复上报相同的vsync  
 // There have been reports of HWCs that signal several vsync events  
 // with the same timestamp when turning the display off and on. This  
 // is a bug in the HWC implementation, but filter the extra events  
 // out here so they don't cause havoc downstream.* **if** (timestamp == mLastHwVSync[disp]) {  
 ALOGW(**"Ignoring duplicate VSYNC event from HWC (t=%"** PRId64 **")"**, timestamp);  
 **return**;  
 }  
  
 mLastHwVSync[disp] = timestamp;  
 }  
  
 **char** tag[16];  
 snprintf(tag, sizeof(tag), **"HW\_VSYNC\_%1u"**, disp);  
 ATRACE\_INT(tag, ++mVSyncCounts[disp] & 1);  
 *//回调 onVsyncReceived函数* mEventHandler.onVSyncReceived(disp, timestamp);  
 }  
}

### onVsyncReceived

**构建模型哦，实际上确实会关闭**

**void** SurfaceFlinger::onVSyncReceived(**int** type, nsecs\_t timestamp) {  
 bool needsHwVsync = **false**;  
  
 { *// Scope for the lock* Mutex::Autolock \_l(mHWVsyncLock);  
 *// 这里的type为0，表示的是primary display,   
 // 而 mPrimaryHWVsyncEnabled 在最初的resyncToHardwareVsync里已经被设置为true了，  
 // 所以这里会进入addResyncSample* **if** (type == 0 && mPrimaryHWVsyncEnabled) {  
 needsHwVsync = mPrimaryDispSync.addResyncSample(timestamp);  
 }  
 }  
 *//addResyncSample会根据现有的硬件Vsync样本计算SW Vsync模型，如果误差已经在可接受范围内  
 // 即认为不再需要硬件Vsync样本了，就得关闭硬件Vsync  
 // 反之，如果误差还比较大，这里还需要继续加入硬件Vsync样本继续计算SW Vsync模型   
 // enableHardwareVsync/disableHardwareVsync都是通过EventControlThread去控制硬件Vsync开关* **if** (needsHwVsync) {  
 enableHardwareVsync();  
 } **else** {  
 disableHardwareVsync(**false**);  
 }  
}

### 5.2.3 addResyncSample

addResyncSample函数从字面上来讲就是加入硬件vsync的样本，目的是为了计算更新SW Vsync里面的参数。 具体的解释全部以注释的方式写在代码里了。

bool DispSync::addResyncSample(nsecs\_t timestamp) {  
 Mutex::Autolock lock(mMutex);  
  
 *//这里MAX\_RESYNC\_SAMPLES为32，即最大只保存32次硬件vsync时间戳，用来计算SW vsync模型.  
 // mNumResyncSamples 表示已经有多少个硬件vsync 样本了  
 // 如果 mNumResyncSamples 等于32个了，那么下一次vsync来了，就用 mFirstResyncSample来记录是第几个  
 // 如果保存的vsync个数达到最大32个的时候， 这样 mNumResyncSamples 和  
 // mFirstResyncSample 两个变量就组成一个窗口(长度为32)向前滑动,  
 // 在滑动过程中丢掉最老的硬件vsync样本* size\_t idx = (mFirstResyncSample + mNumResyncSamples) % MAX\_RESYNC\_SAMPLES;  
 *// mResyncSamples 记录每个硬件vsync样本的时间戳，在计算sw vsync的模型时有用* mResyncSamples[idx] = timestamp;  
  
 *//如果是第一个硬件vsync样本，就直接更新模型 (注意，这里的第一个硬件vsync并不是指开机后的第一个vsync,  
 //而是指 mNumResyncSamples被清0后的第一个vsync信号)，具体在是beginResync里清0的  
 //这里提前说一下，当SW Vsync与硬件Vsync误差比较大后，要重新校准，这里就要 beginResync,  
 //它主要是重置一些值 ，比如 mNumResyncSamples, 既然有误差了，那么之前保存的硬件vsync样本就不能用了，就重新保存新的硬件vsync样本来调节精度了  
 //所这里也很好理解，首先让SW Vsync模型以第一个硬件vsync为基准(注意第一个硬件vsync的含义)，然后再慢慢调节它的精度* **if** (mNumResyncSamples == 0) {  
 mPhase = 0;  
 mReferenceTime = timestamp; *//参考时间设置为第一个硬件vsync的时间戳* mThread->updateModel(mPeriod, mPhase, mReferenceTime);  
 }  
  
 *//更新 mNumResyncSamples 或 mFirstResyncSample的值* **if** (mNumResyncSamples < MAX\_RESYNC\_SAMPLES) {  
 mNumResyncSamples++;  
 } **else** {  
 mFirstResyncSample = (mFirstResyncSample + 1) % MAX\_RESYNC\_SAMPLES;  
 }  
  
 *// 开始计算更新SW vsync 模型* updateModelLocked();  
  
 *// mNumResyncSamplesSincePresent 表示的是当目前的硬件 vsync samples个数大于4个时，就重置error信息。  
 // 注意，在硬件vsync被enable的条件下fence是无效的，所以在这里需要将error信息清空，  
 // 但是为什么要大于MAX\_RESYNC\_SAMPLES\_WITHOUT\_PRESENT(4)时才去reset error信息呢？  
 //注意: 当mNumResyncSamplesSincePresent大于4时，意味着已经保存有6个硬件Vsync样本了，自己好好算算，  
 //由于在硬件Vsync在enable时fence无效，那么应该是每来一个硬件Vsync就应该要reset error呀？为啥还要等到6个过后才reset呢？  
 //确实是这样的，但是在updateModelLocked中，要更新SW vsync模型，至少得有6个及以上的样本才行，所以至少要有6个硬件vsync样本，  
 //所以fense在前6个硬件vsync样本都是无效的，因此不必每次都reset，只要它大于6个过后再reset，真的是细思极恐啊。* **if** (mNumResyncSamplesSincePresent++ > MAX\_RESYNC\_SAMPLES\_WITHOUT\_PRESENT) {  
 resetErrorLocked();  
 }  
  
 **if** (kIgnorePresentFences) {  
 *// If we don't have the sync framework we will never have  
 // addPresentFence called. This means we have no way to know whether  
 // or not we're synchronized with the HW vsyncs, so we just request  
 // that the HW vsync events be turned on whenever we need to generate  
 // SW vsync events.* **return** mThread->hasAnyEventListeners();  
 }  
 *// 如果模型更新了，并且产生的错误小于 kErrorThreshold/2 这个值 (这个值是错误容忍度)，那么 modelLocked就被置为true, 即模型被锁定，模型被锁定的含义是  
 // 现在SW vsync工作的很好，暂时不需要硬件Vsync来进行校正了，最后会将硬件Vsync给disable掉* bool modelLocked = mModelUpdated && mError < (kErrorThreshold / 2);  
 ALOGV(**"[%s] addResyncSample returning %s"**, mName,  
 modelLocked ? **"locked"** : **"unlocked"**);  
 **return** !modelLocked;  
}

接下来继续看下是怎样更新模型里的参数的

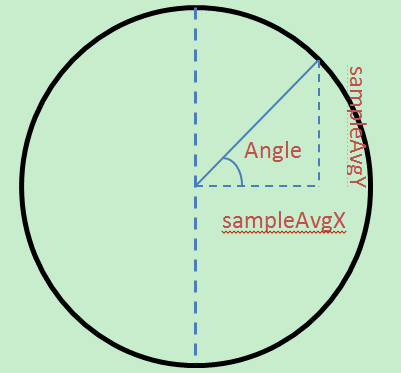
### 5.2.4 updateModelLocked更新模型参数

updateModelLocked函数是根据已经保存的硬件Vsync样本来计算模型的参数。

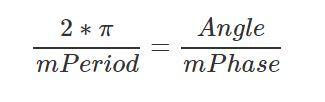
**void** DispSync::updateModelLocked() {  
 *// 如果已经保存了6个以上的 硬件 vsync 样本后，就要开始计算 sw vsync模型了* **if** (mNumResyncSamples >= MIN\_RESYNC\_SAMPLES\_FOR\_UPDATE) {  
 nsecs\_t durationSum = 0;  
 nsecs\_t minDuration = INT64\_MAX;  
 nsecs\_t maxDuration = 0;  
 *//还记得上面 如果 mNumResyncSamples=0,即第一个硬件vsync时，直接更新SW vsync模型了，所以这里把第一个给去除掉* **for** (size\_t i = 1; i < mNumResyncSamples; i++) {  
 size\_t idx = (mFirstResyncSample + i) % MAX\_RESYNC\_SAMPLES;  
 size\_t prev = (idx + MAX\_RESYNC\_SAMPLES - 1) % MAX\_RESYNC\_SAMPLES;  
  
 *// mResyncSamples[idx] - mResyncSamples[prev] 这个差值就是计算出两个硬件vsync样本之间的时间间隔* nsecs\_t duration = mResyncSamples[idx] - mResyncSamples[prev];  
 *// durationSum 表示保存的所有样本(除去第一个vsync)时间间隔之后，用于后面计算 平均 mPeriod* durationSum += duration;  
 minDuration = min(minDuration, duration);  
 maxDuration = max(maxDuration, duration);  
 }  
  
 *// 去掉一个最小，一个最大值再来计算平均值，这个平均值就是硬件vsync产生的时间间隔  
 // Exclude the min and max from the average* durationSum -= minDuration + maxDuration;  
 *// 这里减去3是 一个最大，一个最小，还有第一个硬件vsync* mPeriod = durationSum / (mNumResyncSamples - 3);  
  
 *//下面计算出模型需要的偏移, 因为现在 mPeriod 算出来的是平均值，所以并不是真的硬件vsync时间间隔就是 mPeriod, 存在着偏移与噪音(这个和样本个数有很大的关系)  
 // 即有些样本信号的时间间隔大于平均值，而有些样本时间间隔小于平均值，而这些与 mPriod的差值就是偏移  
 // 下面就是要算出这些平均的偏移值* **double** sampleAvgX = 0;  
 **double** sampleAvgY = 0;  
 *//将硬件vsync的时间间隔换算成对应的度数,即刻度，这里的刻度表示每ns代表多少度* **double** scale = 2.0 \* M\_PI / **double**(mPeriod);  
 *// Intentionally skip the first sample  
 //同样去掉第一个样本* **for** (size\_t i = 1; i < mNumResyncSamples; i++) {  
 size\_t idx = (mFirstResyncSample + i) % MAX\_RESYNC\_SAMPLES;  
 nsecs\_t sample = mResyncSamples[idx] - mReferenceTime;  
 *// 这里对mPeriod取余就是相对于mPeriod倍数的偏移值，然后将其转换成对应的度数* **double** samplePhase = **double**(sample % mPeriod) \* scale;  
 sampleAvgX += cos(samplePhase); *//依次累加成 sampleAvgX* sampleAvgY += sin(samplePhase); *//依次累加成 sampleAvgY* }  
  
 *//获得在x轴与y轴的偏移的平均值* sampleAvgX /= **double**(mNumResyncSamples - 1);  
 sampleAvgY /= **double**(mNumResyncSamples - 1);  
  
 *//最后再通过atan2获得最终的相移值* mPhase = nsecs\_t(atan2(sampleAvgY, sampleAvgX) / scale);  
  
 *//如果相移偏过了mPeriod的一半，那么重新调整一下* **if** (mPhase < -(mPeriod / 2)) {  
 mPhase += mPeriod;  
 ALOGV(**"[%s] Adjusting mPhase -> %"** PRId64, mName, ns2us(mPhase));  
 }  
  
 **if** (kTraceDetailedInfo) {  
 ATRACE\_INT64(**"DispSync:Period"**, mPeriod);  
 ATRACE\_INT64(**"DispSync:Phase"**, mPhase + mPeriod / 2);  
 }  
  
 *// 这个 mRefreshSkipCount 一般为0，它的意思是多少个vsync才进行刷新，即人为的降低显示设备的刷新率了  
 // mRefreshSkipCount 通过 setRefreshSkipCount来设置  
 // Artificially inflate the period if requested.* mPeriod += mPeriod \* mRefreshSkipCount;  
  
 *// 将最新的 偏移 mPhase和 vsync时间间隔mPeriod和mReferenceTime更新到SW vsync模型当中* mThread->updateModel(mPeriod, mPhase, mReferenceTime);  
  
 *// 模型更新了* mModelUpdated = **true**;  
 }  
}

下面来看下几个比较重要的变量

1. 硬件vsync样本个数 MIN\_RESYNC\_SAMPLES\_FOR\_UPDATE  
   要6个硬件vsync样本以上才计算，当然样本越多，模型越精确
2. mPeriod  
   即是显示屏的刷新率，这里mPeriod是根据样本个数去掉一个最大一个最小，算平均
3. mPhase  
   这个是偏移移时间，这个相称和具体的SF/APP Thread里固定的相称是不一样的，这个相移是针对 mPeroid的一个偏移。
4. mModelUpdated  
   这个bool变量表示是否模型已经更新了
5. mReferenceTime  
   这个是第一个硬件Vsync的时间，每次SW vsync计算下一个vsync时间时，都是以该时间作为基准，这样可以减少误差。  
   为什么不以上一个SW vsync时间为基准呢？  
   想像一下，如果SW vsync的每一个Vsync都以上一个vsync时间作为基准，那相当于误差就会不停的累加，而如果以第一个硬件vsync时间作基准，那每次vsync的误差是不会累加的。



*//将硬件vsync的时间间隔换算成对应的度数,即刻度，这里的刻度表示每ns代表多少度***double scale** = 2.0 \* M\_PI / **double**(mPeriod);  
*// Intentionally skip the first sample  
//同样去掉第一个样本* **for** (size\_t **i** = 1; i < mNumResyncSamples; **i**++) {  
 size\_t idx = (mFirstResyncSample + **i**) % MAX\_RESYNC\_SAMPLES;  
 nsecs\_t sample = mResyncSamples[idx] - mReferenceTime;  
 *// 这里对mPeriod取余就是相对于mPeriod倍数的偏移值，然后将其转换成对应的度数* **double** samplePhase = **double**(sample % mPeriod) \* **scale**;  
 sampleAvgX += cos(samplePhase); *//依次累加成 sampleAvgX* sampleAvgY += sin(samplePhase); *//依次累加成 sampleAvgY*}  
  
*//获得在x轴与y轴的偏移的平均值*sampleAvgX /= **double**(mNumResyncSamples - 1);  
sampleAvgY /= **double**(mNumResyncSamples - 1);  
  
*//最后再通过atan2获得最终的相移值*mPhase = nsecs\_t(atan2(sampleAvgY, sampleAvgX) / **scale**);



mPhase对应的角度Angle是通过atan2(sampleAvgY, sampleAvgX)计算出来的，  
最后将角度/scale即可得到相移，单位也是纳秒.

### 计算SW vsync下一个vsync时间点

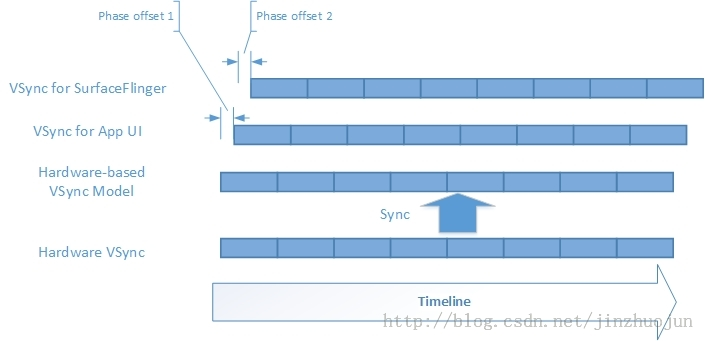
上面已经介绍了DispSync模型了，且模型已经更新好了，那就可以由SW vsync发出vsync信号了呀。  
那接着5.1.1 DispSyncThread的threadLoop的下半部分代码分析

virtual bool threadLoop() {  
 ...  
 *//计算下一次vsync事件的时间* targetTime = computeNextEventTimeLocked(now);  
 **bool** isWakeup = **false**;  
 *//如果计算出来的下一次vsync事件还没有到来，那就等着呗，等着时间到了，就发送SW VSYNC信号  
 //可以看出 DispSyncThread的发送的vsync信号和真正硬件发生的vsync信号没有直接的关系，  
 //发送给app/sf的vsync信号都是由 DispSyncThread发送出去的.* **if** (now < targetTime) {  
 **if** (kTraceDetailedInfo) ATRACE\_NAME(**"DispSync waiting"**);  
  
 **if** (targetTime == INT64\_MAX) {  
 err = mCond.wait(mMutex);  
 } **else** {  
 *//等着SW VSYNC时间到了，就唤醒，开始发送vsync信号* err = mCond.waitRelative(mMutex, targetTime - now);  
 }  
  
 **if** (err == TIMED\_OUT) {  
 *//mCond 是自己醒的，即在targetTime-now时间后醒来的，那就要计算wake up的时间* isWakeup = **true**;  
 } **else if** (err != NO\_ERROR) {  
 ALOGE(**"error waiting for next event: %s (%d)"**,  
 strerror(-err), err);  
 **return false**;  
 }  
 }  
  
 now = systemTime(SYSTEM\_TIME\_MONOTONIC);  
  
 *//计算wake up时间, 但是不能超过1.5 ms  
 // Don't correct by more than 1.5 ms* **static const** nsecs\_t kMaxWakeupLatency = us2ns(1500);  
  
 **if** (isWakeup) {  
 *// mWakeupLatency 醒来时间是累加的，这个在后面计算SW vsync的时间有用, 不过所有的wake up时间最大不能超过1.5 ms, 这点延迟就是代码上的延迟了，看来Google计算的很严谨呀* mWakeupLatency = ((mWakeupLatency \* 63) + (now - targetTime)) / 64;  
 mWakeupLatency = min(mWakeupLatency, kMaxWakeupLatency);  
 **if** (kTraceDetailedInfo) {  
 ATRACE\_INT64(**"DispSync:WakeupLat"**, now - targetTime);  
 ATRACE\_INT64(**"DispSync:AvgWakeupLat"**, mWakeupLatency);  
 }  
 }  
  
 *//收集回调的EventListener, 注意，前面已经加入了eventlistener，参见5.1.2 所以callbackInvocations.size()肯定大于0* callbackInvocations = gatherCallbackInvocationsLocked(now);  
  
 **if** (callbackInvocations.size() > 0) {  
 *//向SF/APP EventThread发送Vsync信号* fireCallbackInvocations(callbackInvocations);  
 }  
}

接着来看下SW vsync模型是怎样计算vsync时间的呢

nsecs\_t computeNextEventTimeLocked(nsecs\_t now) {  
 **if** (kTraceDetailedInfo) ATRACE\_CALL();  
 nsecs\_t nextEventTime = INT64\_MAX;  
 *//对所有的EventListener进行分别计算，里面的mLastEventTime值不同* **for** (size\_t i = 0; i < mEventListeners.size(); i++) {  
 nsecs\_t t = computeListenerNextEventTimeLocked(mEventListeners[i],now);  
 **if** (t < nextEventTime) {  
 nextEventTime = t;  
 }  
 }  
 **return** nextEventTime;  
}

这里其实就最多只有两种EventListener, 一个是SF EventThread,一个是App EventThread,它们都需要接收Vsync信号来分别做不同的事情。  
但是实际上两个线程都有一个偏移，见**2.1**，它们工作既保持一定的节拍，又可以相互错开，一前一后保持着咚次哒次, 还可以让CPU能错开工作高峰。  
见 [Android 5.1 SurfaceFlinger VSYNC详解](https://link.jianshu.com?t=http:/blog.csdn.net/newchenxf/article/details/49131167)



nsecs\_t computeListenerNextEventTimeLocked(**const** EventListener& listener,  
 nsecs\_t baseTime) {  
 **if** (kTraceDetailedInfo) ATRACE\_CALL();  
  
 *// lastEventTime 是求的是上一次vsync事件的时间，它等于上一次vsync事件加上wake up时间  
 // 一般来说baseTime应该不会小于 lastEventTime  
 // 也有小于的情况，比如第一次，threadLoop的now生成的时间比较早，而 addEventListener 发生的比较晚。  
 // 而listener的lastEventTime设为了当前的系统时间，这时baseTime 就会小于 lastEventTime* nsecs\_t lastEventTime = listener.mLastEventTime + mWakeupLatency;  
 **if** (baseTime < lastEventTime) {  
 *//重新修正 baseTime* baseTime = lastEventTime;  
 }  
 *// baseTime 减去参考的时间，这个 mReferenceTime就是第一个硬件Vsync样本的时间* baseTime -= mReferenceTime;  
 *// phase偏移, mPhase是通过硬件vsync的样本计算出来的，而listener.mPhase是固定的具体是在编译时设置的  
 // sf 使用的是 SF\_VSYNC\_EVENT\_PHASE\_OFFSET\_NS；  
 //而APP使用的VSYNC\_EVENT\_PHASE\_OFFSET\_NS* nsecs\_t phase = mPhase + listener.mPhase;  
  
 *// 减去偏移* baseTime -= phase;  
  
 *// If our previous time is before the reference (because the reference  
 // has since been updated), the division by mPeriod will truncate  
 // towards zero instead of computing the floor. Since in all cases  
 // before the reference we want the next time to be effectively now, we  
 // set baseTime to -mPeriod so that numPeriods will be -1.  
 // When we add 1 and the phase, we will be at the correct event time for  
 // this period.* **if** (baseTime < 0) {  
 baseTime = -mPeriod;  
 }  
  
 *//下面是求出下一时刻发送 sw vsync的时间，这个时间是以第一个硬件vsync作为参考来这样计算  
 //为什么不是以上一个sw vsync时间作为参考呢？为什么要以第一个硬件vsync时间作为参考呢？  
 //如果以一个sw vsync时间作为参考，因为sw vsync的时间本身就是一种根据模型模拟出来的，所以本身存在误差，所以如果每个sw vsync以上一个作为base的话，  
 //那么它的误差会慢慢积累。  
 //而每次以第一个硬件vsync时间作为基准，那么每个sw vsync的误差，并不会累加，这样就相对来说更加精确些* nsecs\_t numPeriods = baseTime / mPeriod;  
 *//算出距离第一个硬件Vsync时间的偏移，即得到下一个sw vsync的时间，numPeriods + 1,注意是下一个vsync的时间* nsecs\_t t = (numPeriods + 1) \* mPeriod + phase;  
 *// 这个时间t是相对于每一个硬件 vsync的时间* t += mReferenceTime;  
  
 *// 如果这个vsync距离上一个vsync时间小于3/5个mPeriod的话，为了避免连续的两个sw vsync, 那么这次sw vsync就放弃了，直接放到下一个周期里  
 // Check that it's been slightly more than half a period since the last  
 // event so that we don't accidentally fall into double-rate vsyncs* **if** (t - listener.mLastEventTime < (3 \* mPeriod / 5)) {  
 t += mPeriod;  
 }  
  
 *// 当然算出来的时间要减去wake up的时间了，这样才能精确的模拟硬件vsync的时间, 注意 mWakeupLatency 是所有wake up的时间累加,但是最大只能到1.5ms* t -= mWakeupLatency;  
  
 **return** t;  
}

继续看下 gatherCallbackInvocationsLocked

Vector<CallbackInvocation> gatherCallbackInvocationsLocked(nsecs\_t now) {  
 **if** (kTraceDetailedInfo) ATRACE\_CALL();  
 ALOGV(**"[%s] gatherCallbackInvocationsLocked @ now %"** PRId64, mName,  
 ns2us(now));  
  
 Vector<CallbackInvocation> callbackInvocations;  
 *//因为computeListenerNextEventTimeLocked计算的是下一个vsync时间，那么这一次的vsync就以上now - mPeriod作为基准时间* nsecs\_t onePeriodAgo = now - mPeriod;  
  
 **for** (size\_t i = 0; i < mEventListeners.size(); i++) {  
 nsecs\_t t = computeListenerNextEventTimeLocked(mEventListeners[i],  
 onePeriodAgo);  
  
 **if** (t < now) {  
 CallbackInvocation ci;  
 ci.mCallback = mEventListeners[i].mCallback;  
 ci.mEventTime = t;  
 callbackInvocations.push(ci);  
 *//记录SW vsync的时间* mEventListeners.editItemAt(i).mLastEventTime = t;  
 }  
 }  
  
 **return** callbackInvocations;  
}

到这里基本上说完了DispSync更新模型，以及计算SW Vsync时间。那到这里完了么？还没有呐，现在SW vsync已经按需要由DispSync发出了，但这就完全和硬件Vsync信号保持一致了么？还不一定，所以还需要看下SW vsync与硬件Vsync之间的误差是否还在可接收范围内。

### 5.2.4 更新SW Vsync的误差值

SurfaceFlinger在收到SW Vsync信号后就要去渲染，做图像的合成，在渲染完后会调用postComposition函数

#### 5.2.4.1 postComposition

**void** SurfaceFlinger::postComposition(nsecs\_t */\*refreshStartTime\*/*)  
{  
 mAnimFrameTracker.setPostCompositionTime(mPostCompositionTimestamp);  
**const** LayerVector& layers(mDrawingState.layersSortedByZ);  
**const** size\_t count = layers.size();  
 **for** (size\_t i=0 ; i<count ; i++) {  
 layers[i]->onPostComposition(mPostCompositionTimestamp);  
 }  
  
 *// 通过 HWComposer 获得 Fence***const** HWComposer& hwc = getHwComposer();  
 sp<Fence> presentFence = hwc.getDisplayFence(HWC\_DISPLAY\_PRIMARY);  
  
 *//注意，如果硬件vsync已经被打开了，那么fence是无效了，只有它在关闭的情况下，它才有效* **if** (presentFence->isValid()) {  
 **if** (mPrimaryDispSync.addPresentFence(presentFence)) {  
 ALOGD(**"in setPostCompositionTime will enableHardwareVsync"**);  
 enableHardwareVsync();  
 } **else** {  
 disableHardwareVsync(**false**);  
 }  
 }  
}

由 **5.2.4**的updateModelLocked函数可知，当更新SW Vsync模型后，就会关闭硬件Vsync信号，这时候Fence就有效了， 对于 Fence, 可以参考[Android中的GraphicBuffer同步机制-Fence](https://link.jianshu.com?t=http:/blog.csdn.net/jinzhuojun/article/details/39698317" \t "_blank), 这里简单的理解就是拿到真实硬件Vsync的状态，包含硬件Vsync发生的时间.

#### addPresentFence

bool DispSync::addPresentFence(**const** sp<Fence>& fence) {  
 Mutex::Autolock lock(mMutex);  
  
 *// 将当前硬件vsync的fence保存在 mPresentFences里, 目的是为了计算偏移  
 // mPresentFences 最多保存8个硬件 偏移* mPresentFences[mPresentSampleOffset] = fence;  
 mPresentTimes[mPresentSampleOffset] = 0;  
 mPresentSampleOffset = (mPresentSampleOffset + 1) % NUM\_PRESENT\_SAMPLES;  
 mNumResyncSamplesSincePresent = 0; *// 将 mNumResyncSamplesSincePresent 置为0，* **for** (size\_t i = 0; i < NUM\_PRESENT\_SAMPLES; i++) {  
 **const** sp<Fence>& f(mPresentFences[i]);  
 **if** (f != NULL) { *//这里 f 是有可能为NULL, 即只有一个 硬件 vsync 偏移时* nsecs\_t t = f->getSignalTime(); *//猜测这个就是硬件 vsync的时间* **if** (t < INT64\_MAX) {  
 mPresentFences[i].clear();  
 *//将每个vsync时间戳记录在 mPresentTimes 里，这里 kPresentTimeOffset是可以配置的，即可调的* mPresentTimes[i] = t + kPresentTimeOffset;  
 }  
 }  
 }  
 *//更新错误信息* updateErrorLocked();  
  
 *// 这里，一般的情况是 mModelUpdated 已经被更新了，然后硬件vsync被disable了，  
 // 所以这里只需要看SW vsync的真实的硬件vsync的误差是否在可  
 // 允许的范围内即可* **return** !mModelUpdated || mError > kErrorThreshold;  
}

addPresentFence最后的返回, mError是方差，见下面5.2.4.3分析，当方差大于 kErrorThreshold后就返回true

return !mModelUpdated || mError > kErrorThreshold;

#### 5.2.4.3 updateErrorLocked

void DispSync::updateErrorLocked() {

if (!mModelUpdated) {

return;

}

// Need to compare present fences against the un-adjusted refresh period,

// since they might arrive between two events.

//得到真实的 period, 具体见 5.2.4 updateModelLocked 里的分析

nsecs\_t period = mPeriod / (1 + mRefreshSkipCount);

int numErrSamples = 0;

nsecs\_t sqErrSum = 0;

//这里的 mReferenceTime 是第一个硬件vsync的时间戳 见 addResyncSample里的 mReferenceTime

for (size\_t i = 0; i < NUM\_PRESENT\_SAMPLES; i++) {

nsecs\_t sample = mPresentTimes[i] - mReferenceTime;

// 这里 sample 一般来说是大于偏移的

if (sample > mPhase) {

nsecs\_t sampleErr = (sample - mPhase) % period;

if (sampleErr > period / 2) {

sampleErr -= period;

}

//记录 偏移差的平方和

sqErrSum += sampleErr \* sampleErr;

numErrSamples++;

}

}

// 说到底mError就是方差

if (numErrSamples > 0) {

mError = sqErrSum / numErrSamples;

} else {

mError = 0;

}

if (kTraceDetailedInfo) {

ATRACE\_INT64("DispSync:Error", mError);

}

}

#### 5.2.4.4 硬件

接着返回 **5.2.4.1 postComposition**的最后,

if (mPrimaryDispSync.addPresentFence(presentFence)) {

ALOGD("in setPostCompositionTime will enableHardwareVsync");

enableHardwareVsync();

} else {

disableHardwareVsync(false);

}

如果 addPresentFence见**5.2.4.2** 返回true, 那么就说明SW vsync和硬件Vsync的误差已经无法接受了，那么这时就得重新打开硬件Vsync，来重新调节SW vsync模型了。

# Fw之java层

Java层提供了DisplayEventReceiver.java方法

# Trace

## HW\_VSYNC\_0

DisplayHardware/HWComposer\_hwc1.cpp

**void** HWComposer::vsync(**int** disp, int64\_t timestamp) {  
 **char** tag[16];  
 snprintf(tag, **sizeof**(tag), **"HW\_VSYNC\_%1u"**, disp);  
 ATRACE\_INT(tag, ++mVSyncCounts[disp] & 1);  
  
 mEventHandler.onVSyncReceived(disp, timestamp);  
 }  
}

## HW\_VSYNC\_ON\_0

**void** HWComposer::eventControl(**int** disp, **int** event, **int** enabled) {

**char** tag[16];  
snprintf(tag, **sizeof**(tag), **"HW\_VSYNC\_ON\_%1u"**, disp);  
ATRACE\_INT(tag, enabled);

**}**

## VSYNC-app 和VSYNC-sf

sp<VSyncSource> sfVsyncSrc = **new** DispSyncSource(&mPrimaryDispSync,  
 sfVsyncPhaseOffsetNs, **true**, **"sf"**);  
mSFEventThread = **new** EventThread(sfVsyncSrc, \***this**);  
mEventQueue.setEventThread(mSFEventThread);

DispSyncSource(DispSync\* dispSync, nsecs\_t phaseOffset, **bool** traceVsync,  
 **const char**\* name) :  
 mVsyncEventLabel(String8::format(**"VSYNC-%s"**, name)),

**virtual void** onDispSyncEvent(nsecs\_t when) {  
 sp<VSyncSource::Callback> callback;  
 {  
 Mutex::Autolock lock(mCallbackMutex);  
 callback = mCallback;  
  
 **if** (mTraceVsync) {  
 mValue = (mValue + 1) % 2;  
  **ATRACE\_INT(mVsyncEventLabel.string(), mValue);**  
 }  
 }

**if** (callback != NULL) {  
 callback->onVSyncEvent(when);  
 }  
}

这个的作用是啥？？：通知app层时钟信号

## ？？SurfaceView-com.ss.android.ugc.aweme/XXXtActivity

## UI 主线程

### 生产者消费者buffer个数统计

status\_t BufferQueueProducer::queueBuffer(**int** slot,  
 **const** QueueBufferInput &input, QueueBufferOutput \*output) {

ATRACE\_INT(mCore->mConsumerName.string(), mCore->mQueue.size());

所以这里有高有矮的

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SurfaceView - com.duowan.live/com.duowan.live.live.living.LivingLandActivity: 0 [🔍](https://cs.chromium.org/search/?sq=package:chromium&type=cs&q=SurfaceView%20-%20com.duowan.live%2Fcom.duowan.live.live.living.LivingLandActivity%3A%200) | 0.052 ms | 0.052 ms | 0.002 ms | 34 |

### onMessageReceived

**void** MessageQueue::Handler::handleMessage(**const** Message& message) {  
 **case** REFRESH:  
 mQueue.mFlinger->onMessageReceived(message.what);  
 **break**;  
 }  
}

**void** SurfaceFlinger::onMessageReceived(int32\_t what) {  
 ATRACE\_CALL();  
 handleMessageRefresh();  
}

**void** SurfaceFlinger::handleMessageRefresh() {  
 ATRACE\_CALL();  
 doComposition();

postComposition(refreshStartTime);  
}

### doComposition

**void** SurfaceFlinger::doComposition() {  
 ATRACE\_CALL();  
 postFramebuffer();  
}

#### postFramebuffer

**void** SurfaceFlinger::postFramebuffer()  
{  
 ATRACE\_CALL();

}

### postComposition

mFenceTracker.addFrame(refreshStartTime, presentFence,  
 hw->getVisibleLayersSortedByZ(), hw->getClientTargetAcquireFence());

frameworks/native/services/surfaceflinger/FenceTracker.cpp

#### mFenceTracker.addFrame

**void** FenceTracker::addFrame(nsecs\_t refreshStartTime, sp<Fence> retireFence,  
 **const** Vector<sp<Layer>>& layers, sp<Fence> glDoneFence) {  
 ATRACE\_CALL();



这个是啥事件，没有埋点啊

消费者就在UI线程里面哒哒哒

BufferQueueConsumer::acquireBuffer

ATRACE\_CALL();

frameworks/native/libs/gui/BufferQueueConsumer.cpp

### libs/ui/Fence.cpp

status\_t Fence::wait(**int** timeout) {  
 ATRACE\_CALL();

status\_t Fence::waitForever(**const char**\* logname) {  
 ATRACE\_CALL();

sp<Fence> Fence::merge(**const** String8& name, **const** sp<Fence>& f1,  
 **const** sp<Fence>& f2) {  
 ATRACE\_CALL();

### libs/gui/GLConsumer.cpp

status\_t GLConsumer::updateTexImage() {  
 ATRACE\_CALL();

status\_t GLConsumer::releaseTexImage() {  
 ATRACE\_CALL();

status\_t GLConsumer::detachFromContext() {  
 ATRACE\_CALL();

status\_t GLConsumer::attachToContext(uint32\_t tex) {  
 ATRACE\_CALL();

## 匿名线程-生产者

status\_t BufferQueueProducer::queueBuffer(**int** slot,  
 **const** QueueBufferInput &input, QueueBufferOutput \*output) {  
 ATRACE\_CALL();

## 匿名线程eventControl

eventControl

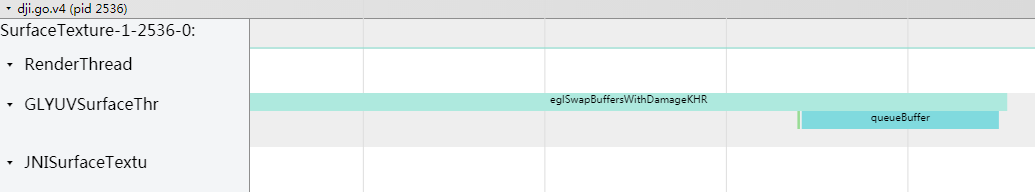
mEventControlThread = **new** EventControlThread(**this**);  
mEventControlThread->run(**"EventControl"**, PRIORITY\_URGENT\_DISPLAY);

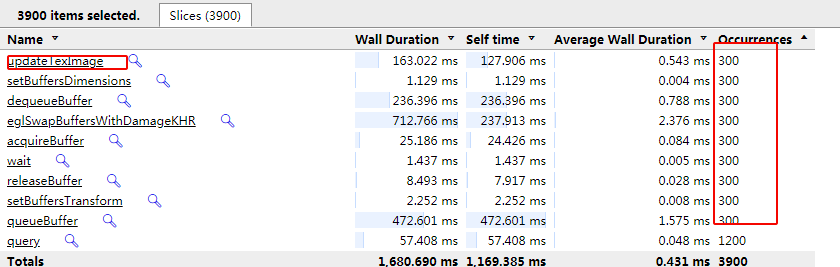
## TARGET\_USES\_HWC2

为何不用HWC2呢

都是用\_hwc1

# 实战





## 在c层实现自己的vysnc

**EventControlThread 和DispSyncThread 由sf实现**

**我们主要实现App EventThread:** 该线程用于接收vsync信号并且上报给App进程，App开始画图

# 参考

Android SurfaceFlinger SW Vsync模型

[http://www.jianshu.com/p/d3e4b1805c92](https://www.jianshu.com/p/d3e4b1805c92)

 [Android - SurfaceFlinger 之 VSync 概括](https://www.jianshu.com/p/6d02d8952a4e)  
这篇文章对 vsync 科普得还行, 没有涉及到一行代码。

 [Android 5.1 SurfaceFlinger VSYNC详解](https://link.jianshu.com?t=http:/blog.csdn.net/newchenxf/article/details/49131167)  
这篇文章对 vsync 的传递流程讲得还是挺不错了，对于理解Surface Vsync流程还是不错的。但是感觉仅仅是在分析代码调用流程而已。

 [Android中的GraphicBuffer同步机制-Fence](https://link.jianshu.com?t=http:/blog.csdn.net/jinzhuojun/article/details/39698317)  
话说第一次见到Fence,也没有仔细阅读

 [DispSync](https://link.jianshu.com?t=http:/echuang54.blogspot.tw/2015/01/dispsync.html)  
这篇文章真的是**五星推荐**，它将SurfaceFlinger的Vsync机制最重要的DispSync部分拿出来讲, 而且讲得非常好。BTW, 这篇文章是我在网上搜到，觉得好像是我们现公司一个大牛写的，于是跟他确认，结果真是他写的。真是大牛

## 其他

### libs/ui/GraphicBufferMapper.cpp

status\_t GraphicBufferMapper::registerBuffer(buffer\_handle\_t handle)

status\_t GraphicBufferMapper::registerBuffer(**const** GraphicBuffer\* buffer)

status\_t GraphicBufferMapper::unregisterBuffer(buffer\_handle\_t handle)

status\_t GraphicBufferMapper::lockAsync(buffer\_handle\_t handle,  
 uint32\_t usage, **const** Rect& bounds, **void**\*\* vaddr, **int** fenceFd)

status\_t GraphicBufferMapper::lockAsyncYCbCr(buffer\_handle\_t handle,  
 uint32\_t usage, **const** Rect& bounds, android\_ycbcr \*ycbcr, **int** fenceFd)

status\_t GraphicBufferMapper::unlockAsync(buffer\_handle\_t handle, **int** \*fenceFd)