Start\_SurfaceFlinger

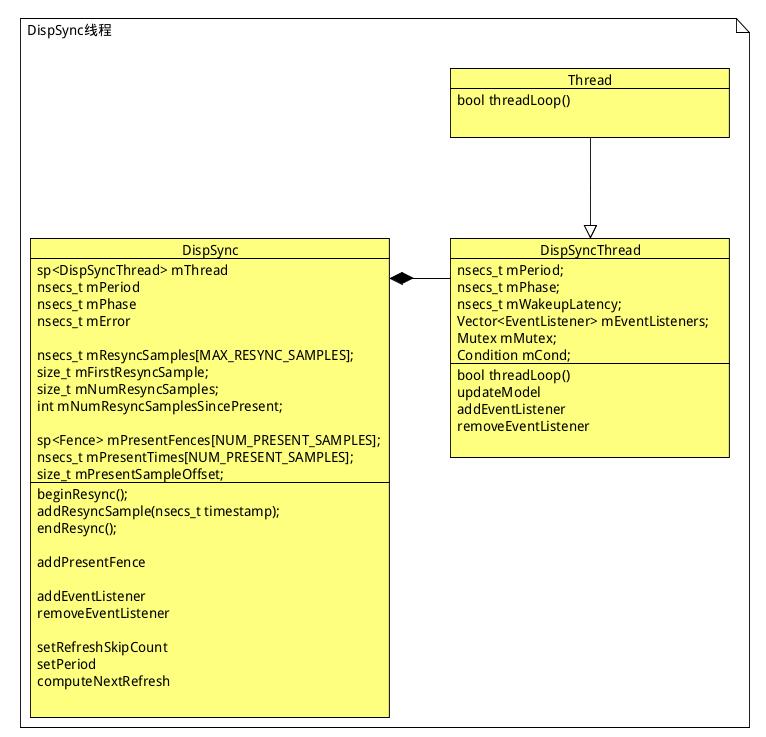
1. DispSync线程（采样分发vsync信号）的创建

DispSync mPrimaryDispSync;

在DispSync对象的构造函数中创建和运行DispSync线程。

检查mPeriod是否为0，当为0线程进入睡眠。

mPeriod只有通过updateModel函数来更新。



1. 接收hwcomposer层的vsync信号和采样。

void DispSync::updateModelLocked() {

if (mNumResyncSamples >= MIN\_RESYNC\_SAMPLES\_FOR\_UPDATE) {

nsecs\_t durationSum = 0;

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;

durationSum += mResyncSamples[idx] - mResyncSamples[prev];

}

mPeriod = durationSum / (mNumResyncSamples - 1);

double sampleAvgX = 0;

double sampleAvgY = 0;

double scale = 2.0 \* M\_PI / double(mPeriod);

for (size\_t i = 0; i < mNumResyncSamples; i++) {

size\_t idx = (mFirstResyncSample + i) % MAX\_RESYNC\_SAMPLES;

nsecs\_t sample = mResyncSamples[idx];

double samplePhase = double(sample % mPeriod) \* scale;

sampleAvgX += cos(samplePhase);

sampleAvgY += sin(samplePhase);

}

sampleAvgX /= double(mNumResyncSamples);

sampleAvgY /= double(mNumResyncSamples);

mPhase = nsecs\_t(atan2(sampleAvgY, sampleAvgX) / scale);

if (mPhase < 0) {

mPhase += mPeriod;

}

if (kTraceDetailedInfo) {

ATRACE\_INT64("DispSync:Period", mPeriod);

ATRACE\_INT64("DispSync:Phase", mPhase);

}

// Artificially inflate the period if requested.

mPeriod += mPeriod \* mRefreshSkipCount;

mThread->updateModel(mPeriod, mPhase);

}

}

1. eventThread线程添加/移除其DispSync::Callback（DispSyncSource）。

status\_t DispSync::addEventListener(nsecs\_t phase,

const sp<Callback>& callback) {

Mutex::Autolock lock(mMutex);

return mThread->addEventListener(phase, callback);

}

status\_t DispSync::removeEventListener(const sp<Callback>& callback) {

Mutex::Autolock lock(mMutex);

return mThread->removeEventListener(callback);

}

1. 通知eventThread线程Vsync信号到达。

virtual bool threadLoop() {

status\_t err;

nsecs\_t now = systemTime(SYSTEM\_TIME\_MONOTONIC);

nsecs\_t nextEventTime = 0;

while (true) {

Vector<CallbackInvocation> callbackInvocations;

nsecs\_t targetTime = 0;

{ // Scope for lock

Mutex::Autolock lock(mMutex);

if (mStop) {

return false;

}

if (mPeriod == 0) {

err = mCond.wait(mMutex);

continue;

}

nextEventTime = computeNextEventTimeLocked(now);

targetTime = nextEventTime;

bool isWakeup = false;

if (now < targetTime) {

err = mCond.waitRelative(mMutex, targetTime - now);

ALOGD("DispSyncThread::threadLoop leave mCond.waitRelative");

if (err == TIMED\_OUT) {

isWakeup = true;

} else if (err != NO\_ERROR) {

return false;

}

}

now = systemTime(SYSTEM\_TIME\_MONOTONIC);

if (isWakeup) {

mWakeupLatency = ((mWakeupLatency \* 63) +

(now - targetTime)) / 64;

if (mWakeupLatency > 500000) {

mWakeupLatency = 500000;

}

}

callbackInvocations = gatherCallbackInvocationsLocked(now);

}

if (callbackInvocations.size() > 0) {

fireCallbackInvocations(callbackInvocations);

}

}

return false;

}

1. EventThread（app）和EventThread（sf）线程的创建

EventThread 线程详解：

1@：获取vsync中的time stamp

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++) {

**//获取mVSyncEvent容器中对应vsync的时间戳**

**//当时间戳不为0的时候将容器中vsync时间戳更新为0**

timestamp = mVSyncEvent[i].header.timestamp;

if (timestamp) {

// we have a vsync event to dispatch

\*event = mVSyncEvent[i];

mVSyncEvent[i].header.timestamp = 0;

vsyncCount = mVSyncEvent[i].vsync.count;

break;

}

}

**//当获取出来的vsync时间戳为0的时候检查Hotplug事件**

if (!timestamp) {

// no vsync event, see if there are some other event

eventPending = !mPendingEvents.isEmpty();

if (eventPending) {

// we have some other event to dispatch

\*event = mPendingEvents[0];

mPendingEvents.removeAt(0);

}

}

2@：遍历mDisplayEventConnections容器

**//遍历当前连接的connect容器**

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;

**//当connection的count大于等于0的时候说明有client在请求了 //vsync信号waitForVSync 需要置为true。**

if (connection->count >= 0) {

waitForVSync = true;

**//timestamp 不为0需要将当前的connect加入到**

**//signalConnections容器中为发送vsync信号callback做准备**

**//timestamp 不为0的情况：**

**//1.H/W vsync打开通过DispSync线程在onVSyncEvent函数中赋值**

**//2. H/Wvsync关闭，fake vsync信号产生（灭屏情况下）。**

**if (timestamp) {**

if (connection->count == 0) {

connection->count = -1;

signalConnections.add(connection);

added = true;

} else if (connection->count == 1 ||

(vsyncCount % connection->count) == 0) {

signalConnections.add(connection);

added = true;

}

}

}

if (eventPending && !timestamp && !added) {

signalConnections.add(connection);

}

} else {

mDisplayEventConnections.removeAt(i);

--i; --count;

}

}

3@：H/W vsync信号的关闭和打开

**//当有产生vsync信号的时间戳，但是没有client端等待vsync信号**

if (timestamp && !waitForVSync) {

**//将eventThread中的DispSyncSource从DispSync线程的Event Listener队 //列中移除**

disableVSyncLocked();

**//当有client端等待vsync信号，但是没有vsync的时间戳**

} else if (!timestamp && waitForVSync) {

**//将eventThread中的DispSyncSource添加到DispSync线程的Event Listener //队列中**

enableVSyncLocked();

}

4@：产生fake vsync信号

**//当vsync信号的时间戳为0（h/w vsync被关闭）而且没有要处理的事件**

**//而且有client已经请求了vsync信号等待刷新界面**

if (!timestamp && !eventPending) {

if (waitForVSync) {

bool softwareSync = mUseSoftwareVSync;

nsecs\_t timeout = softwareSync ? ms2ns(16) : ms2ns(1000);

**//模拟一个每秒一次的vsync信号，时间戳为当前时间。**

if (mCondition.waitRelative(mLock, timeout) == TIMED\_OUT) {

if (!softwareSync) {

ALOGW("Timed out waiting for hw vsync; faking it");

}

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 {

mCondition.wait(mLock);

}

}

} while (signalConnections.isEmpty());

灭屏：

没有vsync信号事件timestamp = 0；

当灭屏的时候EventThread进入mCondition.wait(mLock)的等待，当有client端调用requestNextVsync函数来请求vsync信号的时候，唤醒EventThread线程。由于signalConnections容器为空所以线程重新进入do{}while()的循环。在1@中取出的vsync的时间戳为0.

由于在client端在请求vsync信号的函数requestNextVsync将connection->count置为0，所以在2@步骤中将waitForVSync置为true，说明有client端请求vsync。同时由于vsync的时间戳为0，所以不会将connection加入到signalConnections容器中。

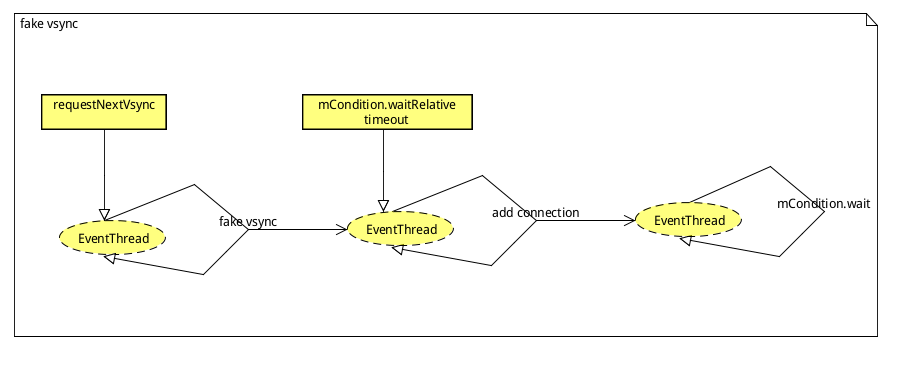
当到达4@中时条件满足，没有vsync信号的时间戳，同时有等待vsync的client，开始进行一次fake vsync的动作，产生一个timeout为16ms的waite事件，当超时的时候设置mVSyncEvent容器中记录vsync信号的Event对象的值，使用当前的时间为vsync时间戳赋值。

同时由于signalConnections容器为空重新进入do{}while()的循环。

在2@中因为timestamp不为0和connection->count置为0将当前connection加入到signalConnections容器中，connection->count置为-1；

由于signalConnections容器不为空所以返回到threadLoop函数中，在该函数中当将vsync信号事件发送给client端之后重新进入waitForEvent函数中。

在该函数中由于mVSyncEvent容器中的vsync的时间戳在上一次waite的过程中被置为了0，connection->count置为了-1，所以直接进入mCondition.wait(mLock)的等待状态。



不灭屏：

没有vsync信号事件timestamp = 0；

当灭屏的时候EventThread进入mCondition.wait(mLock)的等待，当有client端调用requestNextVsync函数来请求vsync信号的时候，唤醒EventThread线程。由于signalConnections容器为空所以线程重新进入do{}while()的循环。在1@中取出的vsync的时间戳为0.

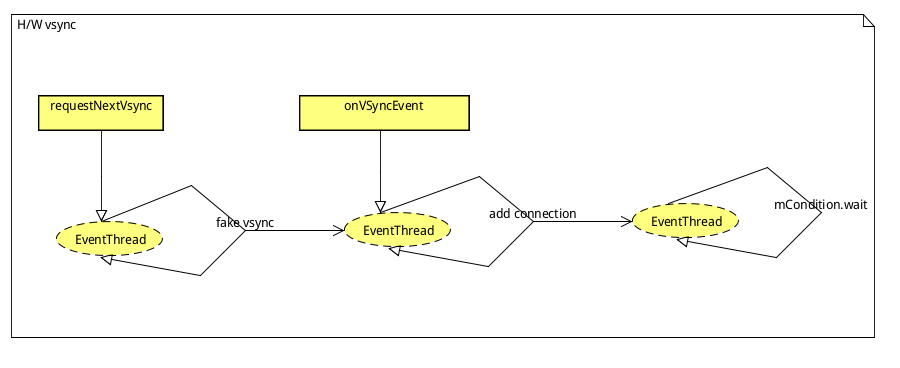
由于在client端在请求vsync信号的函数requestNextVsync将connection->count置为0，所以在2@步骤中将waitForVSync置为true，说明有client端请求vsync。同时由于vsync的时间戳为0，所以不会将connection加入到signalConnections容器中。

当到达4@中时条件满足，没有vsync信号的时间戳，同时有等待vsync的client，开始进行一次fake vsync的动作，产生一个timeout为16ms的waite事件，如果在这个时候onVSyncEvent函数被DispSync线程调用，唤醒等待的线程导致不会产生timeout事件，同时由于signalConnections容器为空重新进入do{}while()的循环。

在2@中因为timestamp不为0（onVSyncEvent函数为其赋值）和connection->count置为0将当前connection加入到signalConnections容器中，connection->count置为-1；

由于signalConnections容器不为空所以返回到threadLoop函数中，在该函数中当将vsync信号事件发送给client端之后重新进入waitForEvent函数中。

在该函数中由于mVSyncEvent容器中的vsync的时间戳在上一次waite的过程中被置为了0，connection->count置为了-1，所以直接进入mCondition.wait(mLock)的等待状态。



App-EventThread:

App通过IDisplayEventConnection进行binder调用requestNextVsync。

通过IDisplayEventConnection将vsync请求发送到app的主线程中。

Sf-EventThread:

WMS或者BufferQueueProducer::queueBuffer通过binder调用在MessageQueue创建的IDisplayEventConnection进行调用requestNextVsync

通过IDisplayEventConnection将vsync请求发送到surfaceflinger的主线程中。

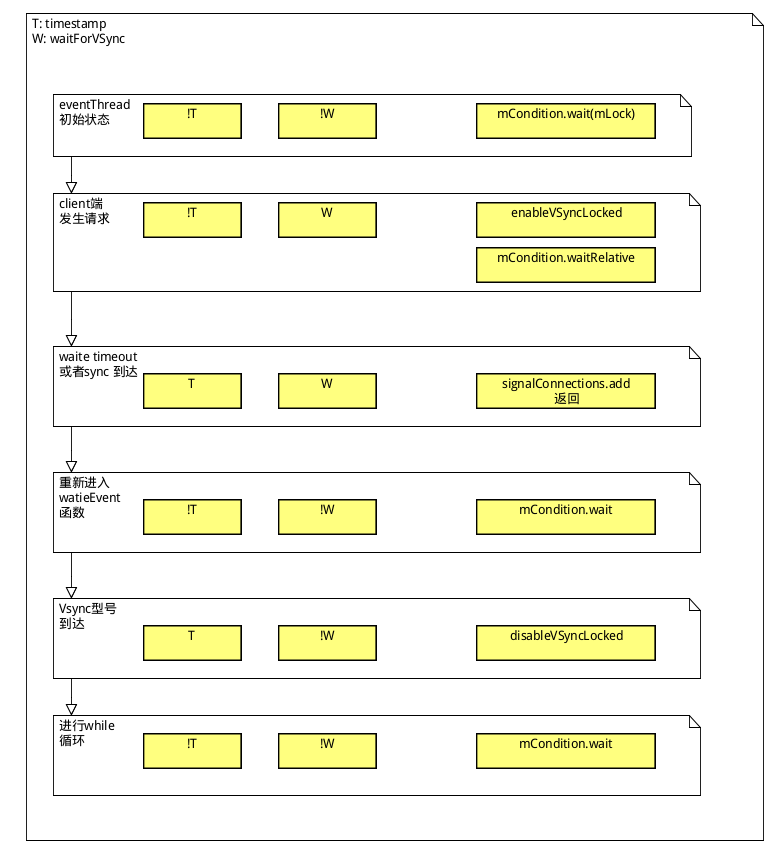
总结：

1.检查是否有waitForVSync的client端。

2.检查是否有vsync的时间戳。

3.控制当前的eventThread中的VSyncSource是否添加到DispSync线程的event listener队列中（决定onVSyncEvent函数是否会被调用）。

当没有client端请求requestNextVsync但是有vsync的时间戳不为0，那么移除VSyncSource。



1. hwcUeventThread（hook\_hotplug）和hwcVsyncThread（hook\_vsync）线程的创建

4.EventControl线程的创建（setVsyncEnabled）