彻底掌握 Android14 Vsync 原理

鸿洋 2025年02月13日 08:35 北京

以下文章来源于阿豪讲Framework ，作者阿豪

阿豪讲Framework
专注 Android Framework 系统源码分析

Android14 Vsync 部分变化较大，本文梳理了其主要要点。

核心要点：

- Vsync 是什么，有什么作用。
- 硬件 Vsync 计算模型的建立与校准。
- 软件 Vsync 信号的计算过程。
- 软件 Vsync 信号的申请与分发。

平台原因，配图可能不清晰，可以通过下面链接查看原图：
https://boardmix.cn/app/share/CAE.CMLx1wwgASoQxRAe7hQCW9dcM671yEwdozAGQAE/hxtUlY

1 Vsync 是什么，有什么作用

Vsync 是一个硬件上的周期性的电平信号，用于同步每一帧画面的渲染、合成和显示过程，防止出现因为渲染、合成和显示的不同步导致的画面撕裂情况出现。

Android 系统中有三个常用的 Vsync 信号：

- HW-Vsync，硬件产生的 Vsync 信号。
- Vsync-app，App 渲染使用的 Vsync 信号，和 HW-Vsync 同频率，但是有一定相位差。
- Vsync-sf，SurfaceFlinger 合成使用的 Vsync 信号，和 HW-Vsync 同频率，但是有一定相位差。

表现在代码层面，软件 Vsync 信号需要先申请后使用，Vsync 信号到来时，具体表示就是一个函数回调的调用：

- App 需要渲染时，向 SF 申请 Vsync-app 信号，SF 计算完成后，通过一个定时器在下一个 Vsync-app 信号到来时调用到 App 中对应的回调，进行图层渲染工作。

- SF 需要合成时，向内部成员申请 Vsync-sf 信号，计算完成后，通过一个定时器在下一个 Vsync-sf 信号到来时调用到 sf 中对应的回调，进行图层合成工作。

- 为什么不直接使用硬件 Vsync 信号，而使用软件 Vsync 信号?

  - VSync 起源于显示屏，如果每个 App 和 SurfaceFlinger 都去从硬件驱动中直接监听 VSync，耦合性太高。比较好的思路是有一个模块去专门跟驱动沟通，再由它将 VSync 信号广播给大家。但是 VSync 频率这么高，每次从 kernel 到 userspace 的消耗也不少，而且 VSync 是周期性的，很容易猜，所以没必要一直从 kernel 监听，通过软件计算模拟即可。

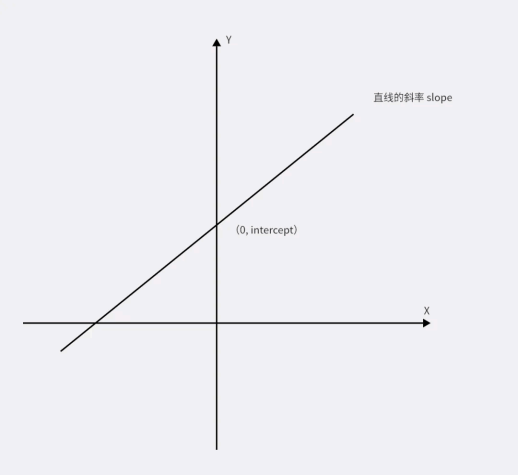
  - 渲染与合成的节奏保持一定的相位差，可以让图像显示更流程，cpu gpu 负载更均衡，这里的相位差，硬件 Vsync 不能直接提供，需要通过软件模拟。

2 HW_Vsync 计算模型的建立

HW_Vsync 计算模型的建立过程:

- SurfaceFlinger 启动时或者是用户修改屏幕刷新率后，会开启 HW-Vsync 信号，通过回调收到不少于 6 个的 HW-Vsync 信号到达时间数据。

- 以 HW_Vsync 信号到达的次序为 X 轴，实际计算中，次序是根据信号到达时间与首个信号到达时间的差值除以周期计算出的（实际代码还考虑了数据可能存在的误差做了一些额外处理），次序的计算结果还乘以了一个固定的倍数做了放大处理。以 HW_Vsync 信号的到达时间做 Y 轴。通过简单线性回归公式计算出 X 与 Y 的一个线性关系 y = ax + b。



- 有了这个模型，给出任意一个时间，将该时间套入公式中的 y = ax + b 中的 y，计算出 x，x 再加 1，就是该时间点的下一个 HW-Vsync 信号对应的次序，再将该次序代入 y = ax + b 的 x 中，就能计算出下一个 HW-Sync 信号的到达时间 c 。在实际代码中为了保证准确性，计算出的结果 c，减去半个周期即 c - slope/2，会代入上述流程再计算一次。

3 软件 Vsync 信号的计算过程

App 渲染使用的是一个软件周期信号 Vsync-app，SurfaceFlinger 合成图层使用的一个软件周期信号 Vsync-sf。这个两个信号都是基于 HW-Vsync 信号计算出的，与 HW-Vsync 同频率，但保持了一定的相位差。

接下来以 Vsync-app 为例，分析软件 Vsync 信号的计算过程。

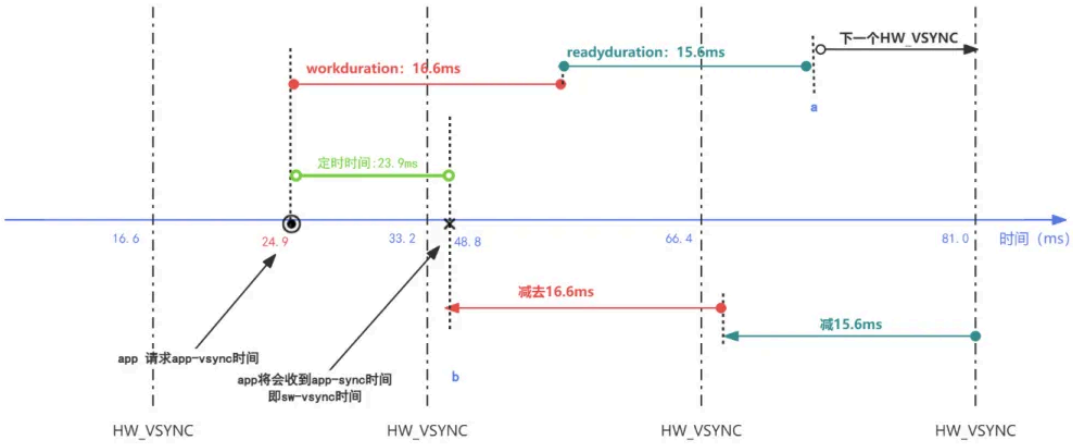
软件 Vsync 信号的几个重要参数：

- **app duration**：app 开始绘制一块 buffer 到 sf 开始消费这块 buffer 的时长（Vsync-app 与对应 Vsync-sf 的间隔，即相位差）；
- **sf duration**：sf 开始合成一块 buffer 到这块 buffer 开始上屏的时长（vsync-sf 到 HW-Vsync 的间隔，即相位差）；
- **app phase**: VSync-app 与 HW_Vsync 的相位差；
- **sf phase**：VSync-sf 与 HW_Vsync 的相位差。

这些参数都定义在手机的属性变量中，在开机 SurfaceFligner 初始化时，被加载进内存中。从定义就容易看出，duration 和 phase 是可以相互转换的。

举例：

app-sync：workduration：16.6ms readyduration: 15.6ms，那么 app-sync 的计算方式如下图所描述：



图片和示例来自：
https://blog.csdn.net/qq_41095045/article/details/136378829?spm=1001.2014.3001.5502

- 假如应用在 24.9ms 时向 surfaceflinger 申请 app-vsync 信号。
- 在 24.9ms 基础上加上 app-vsync 的 workduration 和 readyduration 计算出时间点 a。
- 找出 a 的下一个 HW_VSync 的时间点：81ms，这里就会使用到计算模型 y = ax + b。a 点时间与最早的 HW-Vsync 信号的时间差除以周期再加上 1，做放大处理后，就是下一个 HW-Vsync 信号的次序，代入计算模型即可计算出下一个 HW_Vsync 信号到达的时间。
- 在 a 的的下一个 HW_VSYNC 的时间点上减去 app-vsync 的 workduration 和 readyduration 得到时间点 b: 48.8ms，该时间点就是未来 app 收到 app-vsync 的时间点。

- 计算出surfaceflinger申请app-vsync信号的到时间点b的时间差：23.9ms，并设置定时器。

- 23.9m 后 app 申请的a pp-vsync 时间到，surfaceflinger向app发送app-vsync信号。

以上演示的是以 app-vsync 为例的 vsync 计算过程，appsf-vsync 和 sf-vsync 的计算过程类似，不同的是 appsf-vsync 和sf-vsync 的 workduration，readyduration 是不同的值，以使得信号之间保持一定的相位差。

4 软件 Vsync 模型的校准

校准的场景：

- App 每次请求时，如果距离上次校准超过 750ms，则进行校准。

- SurfaceFlinger 图层合成完成以后将 buffer 提交给 HWC HAL 显示时，会从 HWC HAL 返回一个 PresentFence，PresentFence 的 SignalTime 实际就是一个 HW-VSync 信号。SF 在合成完成以后，会去获取上一次的 PresentFence 的 SignalTime，并将其加入到模型中，如果偏差过大，则进行校准。

如何校准：

远程调用到 HWC HAL，打开硬件 Vsync 开关，接受新的硬件 Vsync 数据，重新计算模型。

5 App 申请 Vsync-app 信号的流程

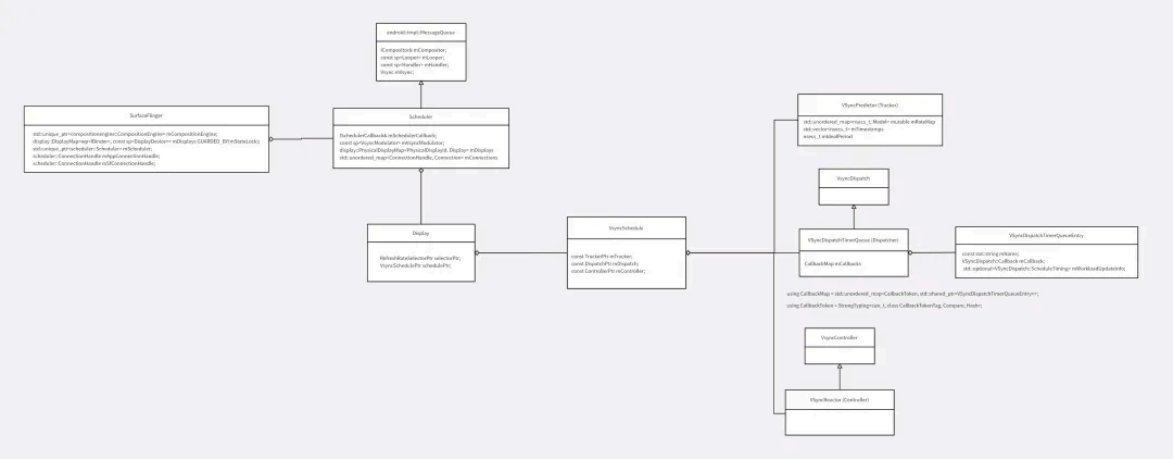
SurfaceFlinger 初始化时，会初始化一个负责处理 Vsync 的重要成员 VsyncSchedule，VsyncSchedule 有三个核心的帮手：

- **VSyncPredictor**，在 VsyncSchedule 中的名字是 Tracker，负责 HW-Vsync 信号的记录，软件 Vsync 模型的建立，模拟 Vsync 信号的计算等。

- **VSyncDispatchTimerQueue**，在 VsyncSchedule 中的名字是 Dispatcher，负责软件 Vsync 信号的分发。

- **VSyncReactor**，在 VsyncSchedule 中的名字是 Dispatcher，主要作用是负责传递 HWVsync，presentFence 信号。

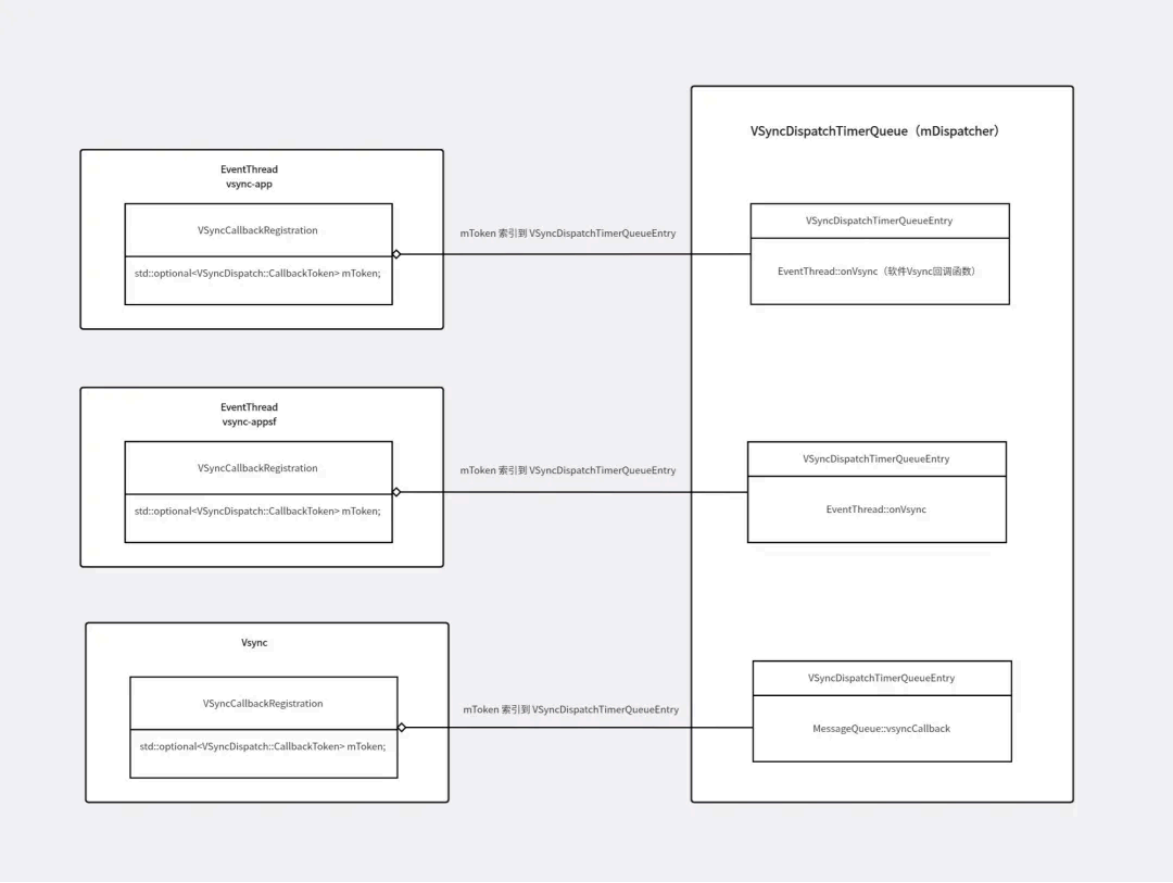
相关类图如下：



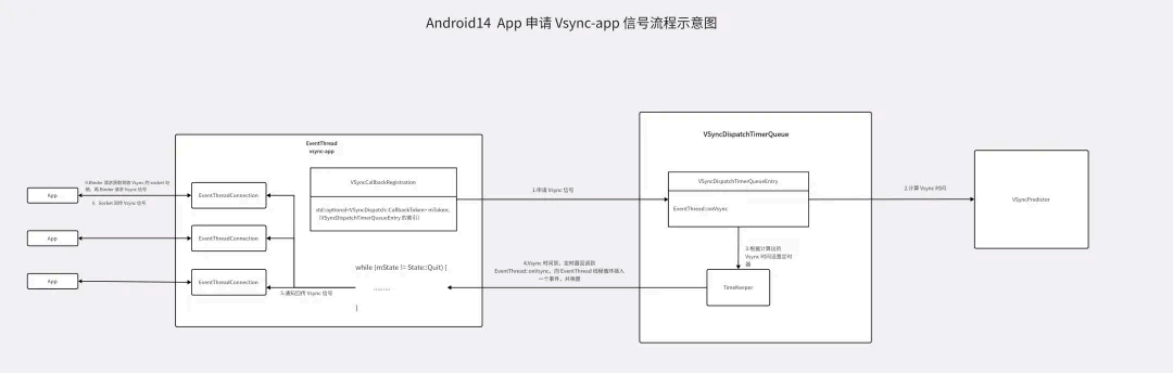
VSyncDispatchTimerQueue（dispatcher） 分发 Vsync 信号的目标有三个 EventThread（app）、EventThread(appsf) 和 Vsync 对象。

三个目标在 SurfaceFligner 初始化时完成初始化，初始化时，会构建一个 VSyncDispatchTimerQueueEntry 对象，插入 VSyncDispatchTimerQueue 的 CallbackMap 成员中。对应的 key 保存在 EventThread（app）、EventThread(appsf) 和 Vsync 对象的 VSyncCallbackRegistration 成员中的 mToken 成员中。

VSyncDispatchTimerQueueEntry 中会保存软件 Vsync 收到后对应的回调函数。



App 申请 Vsync-app 信号的流程



1. App 端建立与 SF 端 EventThreadConnection 匿名 Binder 通信通道。

2. App Binder 远程调用到 SF 端 EventThreadConnection::stealReceiveChannel 函数，获取到后续收 Vsync-app 信号的的 socket 句柄。

3. App 渲染前，远程调用到 SF 中的 EventThreadConnection::requestNextVsync 函数请求 Vsync-app 信号，该函数会修改 EventThreadConnection#vsyncRequest 的值为 VSyncRequest::Single，接着唤醒 wait 中的 EventThread 线程。

4. EventThread 唤醒后，调用到 VSyncCallbackRegistration::schedule 函数开始软件 Vsync 信号的计算分发。

   a. 通过 VSyncPredictor（Tracker）计算出下一次 Vsync-app 时间，

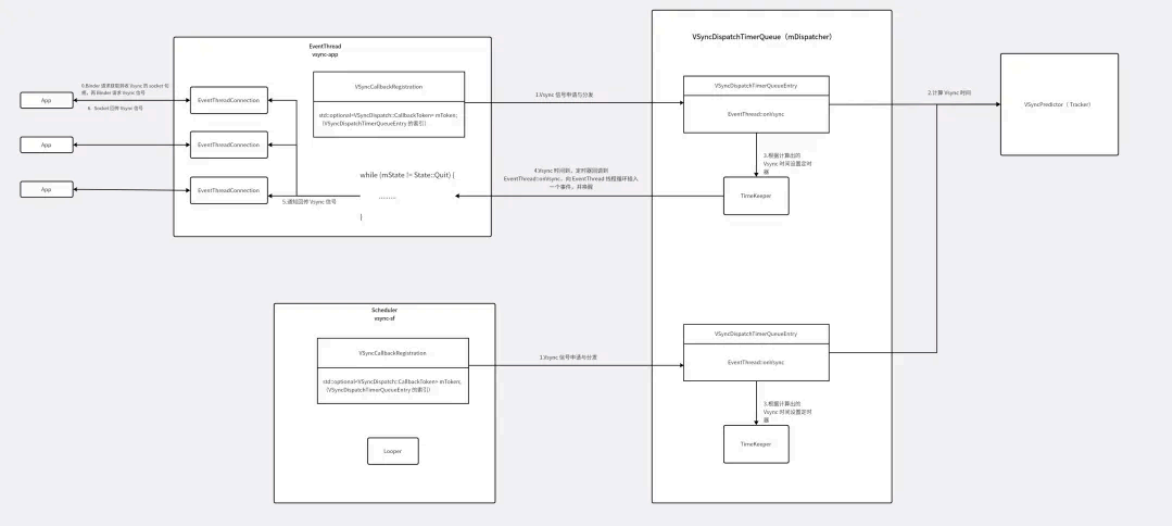
   b. 接着以该时间安排一个 TimeKeeper 定时器，时间到达后，回调到 EventThread 对应的回调函数 EventThread::onVsync。

5. EventThread::onVsync 会构建一个 DisplayEventReceiver::Event 对象，插入 EventThread 线程的事件队列中，接着唤醒 EventThread。

6. EventThread 唤醒后，读取到 Event，找到目标 EventThreadConnection，确定的主要标准是 EventThreadConnection#vsyncRequest 的值为 VSyncRequest::Single，然后通过 EventThreadConnection 的 socket 通信通道发送给 App，最后修改EventThreadConnection#vsyncRequest 的值为 SingleSuppressCallback。

7. App 中收到 Vsync 信号后，最终回调到 choreographer#doFrame 开始渲染。

6 SF 申请 Vsync-sf 信号的流程



1. 当 SF 需要请求刷新时，会调用到 Scheduler 的 scheduleFrame 函数，该函数定义在 Scheduler 的父类 MessageQueue 中。

2. 接着调用 Scheduler 的 mVsync 成员 VSyncCallbackRegistration 的 schedule 函数，开始软件 Vsync 信号的计算分发。

3. 通过 VSyncPredictor（Tracker）计算出下一次 Vsync-app 时间，

4. 接着以该时间安排一个 TimeKeeper 定时器，时间到达后，回调到对应的回调函数 MessageQueue::vsyncCallback。

5. MessageQueue::vsyncCallback 向 Looper 发送一个 Message，后续 Looper 回调到 Message 对应的回调函数 MessageQueue::Handler::handleMessage。

6. 在 MessageQueue::Handler::handleMessage 回调中会调用 Scheduler::onFrameSignal。

最后推荐一下我做的网站，玩Android: *wanandroid.com* ，包含详尽的知识体系、好用的工具，还有本公众号文章合集，欢迎体验和收藏！

扫一扫 关注我的公众号

如果你想要跟大家分享你的文章，欢迎投稿~

┏(^ 0 ^)┛明天见！