FPS是图像领域中的定义，是指画面每秒传输帧数，通俗来讲就是指动画或视频的画面数。每秒钟帧数愈多，所显示的动作就会愈流畅。通常，要避免动作不流畅的最低是30。FPS也可以理解为我们常说的刷新率（单位为Hz）。

#### 1.1 基本概念

显示系统分为：CPU、GPU、Display三个部分。

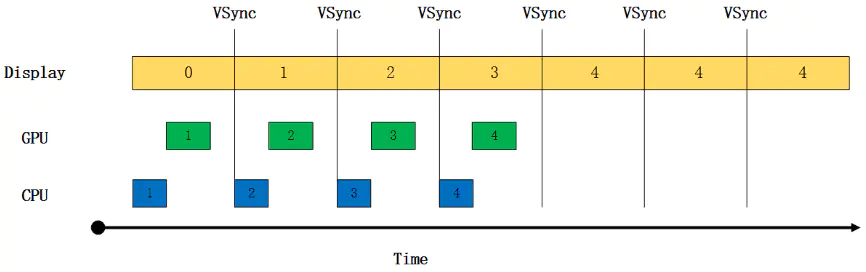
* CPU：计算屏幕数据，把计算好的数据交给GPU；
* GPU：对图形数据进行渲染（栅格化操作，将图像资源转化成像素图），渲染好后放到buffer里存起来；
* Display：负责把buffer里的数据呈现到屏幕上；

#### 1.2 绘制模型

① 软件绘制模型：  
这里由CPU主导绘图，该模型先让视图结构(view hierarchy)失效，再绘制整个视图结构。当应用程序需要更新它的部分UI时，都会调用内容发生改变的View对象的invalidate()方法。无效（invalidation）消息请求会在View对象层次结构中传递，以便计算出需要重绘的屏幕区域（脏区）。然后，Android系统会在View层次结构中绘制所有的跟脏区相交的区域。这种绘制的缺点在于绘制了不需要重绘的视图。

② 硬件加速绘制模型  
这里由GPU主导绘图，该模型先让视图结构失效，再记录和更新显示列表（Display List)，这两步是在CPU当中完成的。最后在GUP中绘制显示列表。这种模式下，Android系统依然会使用invalidate()方法和draw()方法来请求屏幕更新和展现View对象。但Android系统并不是立即执行绘制命令，而是首先把这些View的绘制函数作为绘制指令记录一个显示列表中，然后再读取显示列表中的绘制指令调用OpenGL相关函数完成实际绘制。另一个优化是，Android系统只需要针对由invalidate()方法调用所标记的View对象的脏区进行记录和更新显示列表。没有失效的View对象就简单重用先前显示列表记录的绘制指令来进行简单的重绘工作。使用显示列表的目的是，把视图的各种绘制函数翻译成绘制指令保存起来，对于没有发生改变的视图把原先保存的操作指令重新读取出来重放一次就可以了，提高了视图的显示速度。而对于需要重绘的View，则更新显示列表，然后再调用OpenGL完成绘制。这种模型提高了Android系统显示和刷新的速度，但是它的兼容性没有软件模型好，同时更消耗内存也更加耗电。

#### 1.3 Android显示方式



① 底层以固定频率发出VSync信号，这个信号周期是16.6ms；  
② VSync信号到来时，CPU对View进行计算（包含每个View的测量、布局、绘制），然后交给GPU进行渲染，Display则显示上次GPU渲染好的buffer数据；

* Display  
  黄色这一行可以理解成屏幕，底层以固定的频率发出VSync信号，其周期为16.6ms。数字：0,1,2,3,4，可以看到每次屏幕刷新信号到来时，数字就会变化，所以这些数字其实理解成每一帧屏幕显示的画面。
* CPU  
  蓝色这一行，表示App绘制当前view树的时间，而这段时间就跟我们自己写的代码有关系，如果布局很复杂、层次嵌套很多，每一帧内需要刷新的View又比较多时，那么每一帧的绘制耗时自然就会多一点。其中的数字代表在计算的画面数据，也叫屏幕数据，即在当前帧内，cpu计算下一帧的屏幕画面数据。

CPU跟Display是不同的硬件，它们是并行工作的。要理解的一点是，我们写的代码，只是控制让 CPU 在接收到屏幕刷新信号的时候开始去计算下一帧的画面工作。而底层在每一次屏幕刷新信号来的时候都会去切换这一帧的画面，这点我们是控制不了的，是底层的工作机制。之所以要讲这点，是因为，当我们的 app 界面没有必要再刷新时（比如用户不操作了，当前界面也没动画），这个时候，我们 app 是接收不到屏幕刷新信号的，所以也就不会让 CPU 去计算下一帧画面数据，但是底层仍然会以固定的频率来切换每一帧的画面，只是它后面切换的每一帧画面都一样，所以给我们的感觉就是屏幕没刷新。

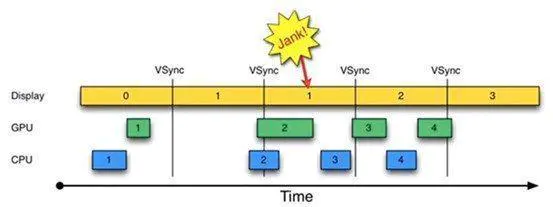
观察图形可以看出：前三帧跟原图一样，从第三帧之后，因为我们的 app 界面不需要刷新了（用户不操作了，界面也没有动画），那么这之后我们 app 就不会再接收到屏幕刷新信号了，所以也就不会再让 CPU 去绘制视图树来计算下一帧画面了。<strong>但是</strong>，底层还是会每隔 16.6ms 发出一个屏幕刷新信号，只是我们 app 不会接收到而已，Display 还是会在每一个屏幕刷新信号到的时候去显示下一帧画面，只是下一帧画面一直是第4帧的内容而已。

#### 1.4 VSYNC信号同步处理数据

在android 4.1以前UI不流畅问题较为严重，在4.1版本以后Android对显示系统进行了重构，引入了三个核心元素：VSYNC, Tripple Buffer和Choreographer。VSYNC是Vertical Synchronized的缩写，是一种定时中断；Tripple Buffer是显示数据的缓冲区；Choreographer起调度作用，将绘制工作统一到VSYNC的某个时间点上，使应用的绘制工作有序进行。

Android在绘制UI时，会采用一种称为“双缓冲”的技术，双缓冲即使用两个缓冲区（在SharedBufferStack中），其中一个称为Front Buffer，另外一个称为Back Buffer。UI总是先在Back Buffer中绘制，然后再和Front Buffer交换，渲染到显示设备中。

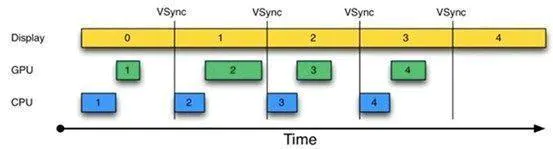
##### 1.4.1 没有VSYNC同步信号



理想情况下，一个刷新会在16ms内完成（60FPS），也就是两个VSync信号之间。这样并不会出现问题，如上图第一个16ms开始，此时Display显示第0帧，CPU处理完第1帧后，GPU紧接其后处理继续第1帧。在第二个16ms，第1帧能正常显示在Display中。但在第二个16ms阶段CPU和GPU却并未及时去绘制第2帧数据（前面的空白区表示CPU和GPU忙其它的事），直到在本周期快结束时，CPU/GPU才去处理第2帧数据。这时当进入第三个16ms，此时Display应该显示第2帧数据，但由于CPU和GPU还没有处理完第2帧数据，故Display只能继续显示第一帧的数据，结果使得第1帧多画了一次（对应时间段上标注了一个Jank），导致错过了显示第二帧。这种情况就会导致UI的卡顿感。

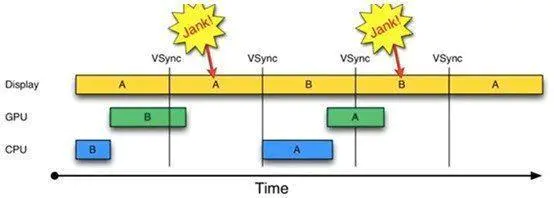
通过上述分析可知，此处发生Jank的关键问题在于，第1个16ms段内，CPU/GPU没有及时处理第2帧数据。 为解决这个问题，Android 4.1中引入了VSYNC，核心目的是解决刷新不同步的问题。

##### 1.4.2 加入 VSYNC同步信号



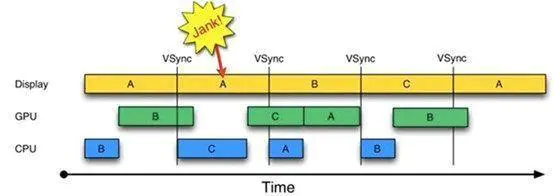
在加入VSYNC信号同步后，每收到VSYNC中断，CPU就立刻开始处理各帧数据。这样就解决了刷新不同步的问题。

但是上图中仍然存在一个问题：如果在两个同步信号之间的这段时间里也就是说16ms中CPU和GPU处理绘制不完这就又会遇到卡顿的现象，如下图：



在第二个16ms时间段，Display本应显示B帧，但却因为GPU还在处理B帧，导致A帧被重复显示。同理，在第二个16ms时间段内，CPU无所事事，因为A Buffer被Display在使用而B Buffer被GPU在使用。同时，一旦过了VSYNC时间点，CPU就不能再被触发以处理绘制工作了。要等下一个VSYNC信号。由于只有两个Buffer（Android 4.1之前）GPU和CPU不能同时进行处理。为了解决这一问题，于是在Android4.1以后，引出了第三个缓冲区：Tripple Buffer。

##### 1.4.3. 加入Tripple Buffer



第二个16ms时间段，CPU使用C Buffer绘图。虽然还是会多显示A帧一次，但后续显示就会流畅许多。虽然android4.1以后优化了绘制机制，但最好还是在16ms内能将展示界面绘制出来。

### 2 刷新机制

Android 设备呈现到界面上的大多数情况下都是一个 Activity，真正承载视图的是一个 Window，每个 Window 都有一个 DecorView，当调用 setContentView() 时其实是将自己写的布局文件添加到以 DecorView 为根布局的一个 ViewGroup 里，构成一颗 View 树。每个 Activity 对应一棵以 DecorView 为根布局的 View 树，DecorView 类里面有个 mParent属性，它的类型就是 ViewRootImpl，而且每个界面上的 View 的刷新，绘制，点击事件的分发其实都是由 ViewRootImpl 作为发起者的，由 ViewRootImpl 控制这些操作从 DecorView 开始遍历 View 树去分发处理。

#### 2.1 绘制起点

在View的绘制起点源于scheduleTraversals()方法，此处分析它是如何注册到VSync信号上，以及回调时做了哪些事情。首先看下scheduleTraversals的代码：

//=== ViewRootImpl.java ===

void scheduleTraversals() {

if (!mTraversalScheduled) {

mTraversalScheduled = true;

mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();

mChoreographer.postCallback(Choreographer.CALLBACK\_TRAVERSAL, mTraversalRunnable, null);

if (!mUnbufferedInputDispatch) {

scheduleConsumeBatchedInput();

}

notifyRendererOfFramePending();

pokeDrawLockIfNeeded();

}}

mTraversalScheduled这个变量是为了过滤一帧内重复的刷新请求，初始值是false。在开始这一帧的绘制流程时候也会重新置为false（doTraversal()中，一会儿分析），同时，在取消遍历绘制 View 操作 unscheduleTraversals() 里也会设置为false。也就是说一般情况下在开始这一帧的正式绘制前，在这期间重复调用scheduleTraversals()只有一次会生效。这么设计的原因是因为当刷新时候会对整个DecorView进行一次处理，所以不同view触发的scheduleTraversals()作用都是一样的，所以在这一帧里面只要有一次和多次刷新请求效果是一样的。

#### 2.2 注册监听VSync信号

##### 2.2.1 派送CALLBACK\_TRAVERSAL消息

**Choreographer.postCallback(...)**

public void postCallback(int callbackType, Runnable action, Object token) {

postCallbackDelayed(callbackType, action, token, 0);}

public void postCallbackDelayed(int callbackType, Runnable action, Object token, long delayMillis) {

if (action == null) {

throw new IllegalArgumentException("action must not be null");

}

if (callbackType < 0 || callbackType > CALLBACK\_LAST) {

throw new IllegalArgumentException("callbackType is invalid");

}

postCallbackDelayedInternal(callbackType, action, token, delayMillis);}

private void postCallbackDelayedInternal(int callbackType, Object action, Object token, long delayMillis) {

......

synchronized (mLock) {

final long now = SystemClock.uptimeMillis();

final long dueTime = now + delayMillis;

mCallbackQueues[callbackType].addCallbackLocked(dueTime, action, token);、

// 如果在Looper主线程上运行，则立即安排vsync；否则的话发送一个Message ，尽快从UI线程安排vsync。

if (dueTime <= now) {

scheduleFrameLocked(now);

} else {

Message msg = mHandler.obtainMessage(MSG\_DO\_SCHEDULE\_CALLBACK, action);

msg.arg1 = callbackType;

msg.setAsynchronous(true);

mHandler.sendMessageAtTime(msg, dueTime);

}

}}

在postCallbackDelayInternal()里面会先根据当前时间戳将这个Runnable保存到一个mCallbackQueues队列里，这个队列跟MessageQueue很相似，里面执行的任务都是根据一个时间戳来排序。最后会执行scheduleFrameLocked()方法。

##### 2.2.2 向底层注册刷新监听

private void scheduleFrameLocked(long now) {

if (!mFrameScheduled) {

mFrameScheduled = true;

if (USE\_VSYNC) {

......

//是否在主线程

if (isRunningOnLooperThreadLocked()) {

scheduleVsyncLocked();

} else {

Message msg = mHandler.obtainMessage(MSG\_DO\_SCHEDULE\_VSYNC);

msg.setAsynchronous(true);

mHandler.sendMessageAtFrontOfQueue(msg);

}

} else {

final long nextFrameTime = Math.max(mLastFrameTimeNanos / TimeUtils.NANOS\_PER\_MS + sFrameDelay, now);

Message msg = mHandler.obtainMessage(MSG\_DO\_FRAME);

msg.setAsynchronous(true);

mHandler.sendMessageAtTime(msg, nextFrameTime);

}

}}

private void scheduleVsyncLocked() {

mDisplayEventReceiver.scheduleVsync();}

最后会走到DisplayEventReceiver.scheduleVsync()方法中；

//=== DisplayEventReceiver.java ===public void scheduleVsync() {

if (mReceiverPtr == 0) {

Log.w(TAG, "Attempted to schedule a vertical sync pulse but the display event "

+ "receiver has already been disposed.");

} else {

nativeScheduleVsync(mReceiverPtr);

}}

nativeScheduleVsync(..)相当于向底层注册了个监听，订阅一个Vsync信号。android系统每过16.6ms会发送一个Vsync信号，但这个信号并不是所有app都能收到的，只有订阅了的才能收到。这样设计的合理之处在于，当UI没有变化的时候就不会去调用nativeScheduleVsync(mReceiverPtr)去订阅，从而也就不会收到Vsync信号，也就不会有不必要的绘制操作。

#### 2.3 处理回调事件

##### 2.3.1 底层回调

既然是mDisplayEventReceiver用scheduleVsync()来订阅Vsync信号的，也是由这个类来接收Vsync信号的。

private final class FrameDisplayEventReceiver extends DisplayEventReceiver implements Runnable {

public FrameDisplayEventReceiver(Looper looper, int vsyncSource){

super(looper, vsyncSource);

}

@Override

public void onVsync(long timestampNanos, int builtInDisplayId, int frame) {

......

mTimestampNanos = timestampNanos;

mFrame = frame;

Message msg = Message.obtain(mHandler, this);

msg.setAsynchronous(true);

mHandler.sendMessageAtTime(msg, timestampNanos / TimeUtils.NANOS\_PER\_MS);

}

@Override

public void run() {

mHavePendingVsync = false;

doFrame(mTimestampNanos, mFrame);

}}

FrameDisplayEventReceiver收到Vsync信号后，调用onVSync方法将自己的run()方法传给mHandler进行处理，同时将这个消息设为异步消息。消息处理时执行run方法里的doFrame()，这里mTimestampNanos是信号到来的时间参数。

onVSync是底层回调的，可以理解成，底层每隔16.6ms一个帧信号来的时候，底层就会回调这个方法，当然前提是我们得先注册，这样底层才能找到我们的app并回调。

##### 2.3.2 消息处理doFrame()

// Choreographer.class

void doFrame(long frameTimeNanos, int frame) {

final long startNanos;

synchronized (mLock) {

...

try {

...

// Choreographer.CALLBACK\_TRAVERSAL这个参数和 mChoreographer.postCallback(）里面传入的一致

doCallbacks(Choreographer.CALLBACK\_TRAVERSAL, frameTimeNanos);

...

}

...}

void doCallbacks(int callbackType, long frameTimeNanos) {

CallbackRecord callbacks;

synchronized (mLock) {

...

// 取出之前放入mCallbackQueues的mTraversalRunnable

callbacks = mCallbackQueues[callbackType].extractDueCallbacksLocked(now / TimeUtils.NANOS\_PER\_MS);

...

// 回调mTraversalRunnable的run函数

for (CallbackRecord c = callbacks; c != null; c = c.next) {

c.run(frameTimeNanos);

}

...

}}

doFrame()里面的doCallbacks(Choreographer.CALLBACK\_TRAVERSAL, frameTimeNanos);函数会回调之前 mChoreographer.postCallback(）里面传入的mTraversalRunnable，进而最终调用performTraversals()来计算下一阵的数据，等到下一帧将计算的数据显示到屏幕上。

##### 2.3.3 处理计算任务

mChoreographer.postCallback(...)方法传入Runnable参数，如代码所示：

//=== ViewRootImpl.java ===

final class TraversalRunnable implements Runnable {

@Override

public void run() {

doTraversal();

}}

//=== ViewRootImpl.java ===

void doTraversal() {

if (mTraversalScheduled) {

mTraversalScheduled = false;

mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);

......

//这里开始对view树进行测量、布局、绘制

performTraversals();

......

}}

// === ViewRootImpl.java ===

private void performTraversals() {

......

int childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width);

int childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height);

......

// Ask host how big it wants to be

performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);

......

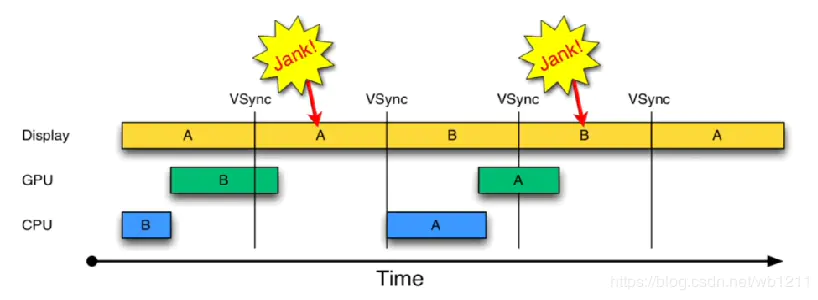
performLayout(lp, mWidth, mHeight);

......

performDraw();}

Runnable做的事很简单，就调用了一个doTraversal()方法，在performTraversals()方法中完成View的测量、布局、绘制三大流程。

#### 2.4 掉帧现象



当CPU+GPU处理的时间超过了16.6ms，在Vsync到来的时候，显示的数据还未准备好，只能显示之前的数据帧，导致出现丢帧现象。

##### 2.4.1 View布局过于复杂

遍历绘制View树计算屏幕数据的时间超过了 16.6ms，说明我们写的布局有问题，需要进行优化。

##### 2.4.2 主线程执行耗时任务

主线程一直在处理其他耗时的消息，导致遍历绘制 View 树的工作迟迟不能开始，从而超过了16.6 ms。这个问题告诉我们避免在主线程做耗时任务。

##### 2.4.3 同步屏障

我们清楚主线程其实是一直在处理 MessageQueue里的Message，主线程同一时间只能处理一个Message，那么如果不做任何操作，就有可能出现这样的情况： 底层回调我们app的onVsync()把遍历绘制View树的操作post到主线程的MessageQueue中去等待执行，主线程同一时间只能处理一个 Message，这些Message肯定有先后的问题，当我们接收到屏幕刷新信号时，就可能来不及第一时间去执行刷新屏幕的操作。这样一来，即使我们将布局优化得很彻底，保证绘制当前View树不会超过16ms，但如果不能第一时间优先处理绘制View的工作，那等16.6ms过了，底层需要去切换下一帧的画面了，我们app却还没处理完，这样也照样会出现丢帧了。而且这种场景是非常有可能出现的吧，毕竟主线程需要处理的事肯定不仅仅是刷新屏幕的事而已，那么这个问题是怎么处理的呢？

// scheduleTraversals()

mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();

// doTraversal()

mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);

在逻辑走进Choreographer前会先往队列里发送一个同步屏障，而当 doTraversal()被调用时才将同步屏障移除。同步屏障消息只能由内部来发送，这个接口并没有公开给我们使用。

同步屏障的作用可以理解成拦截同步消息的执行，主线程的Looper会一直循环调用MessageQueue的 next()来取出队头的Message执行，当Message执行完后再去取下一个。当next()方法在取Message时发现队头是一个同步屏障的消息时，就会去遍历整个队列，只寻找设置了异步标志的消息。如果有找到异步消息，那么就取出这个异步消息来执行，否则就让next()方法陷入阻塞状态。如果next()方法陷入阻塞状态，那么主线程此时就是处于空闲状态的，也就是没在干任何事。

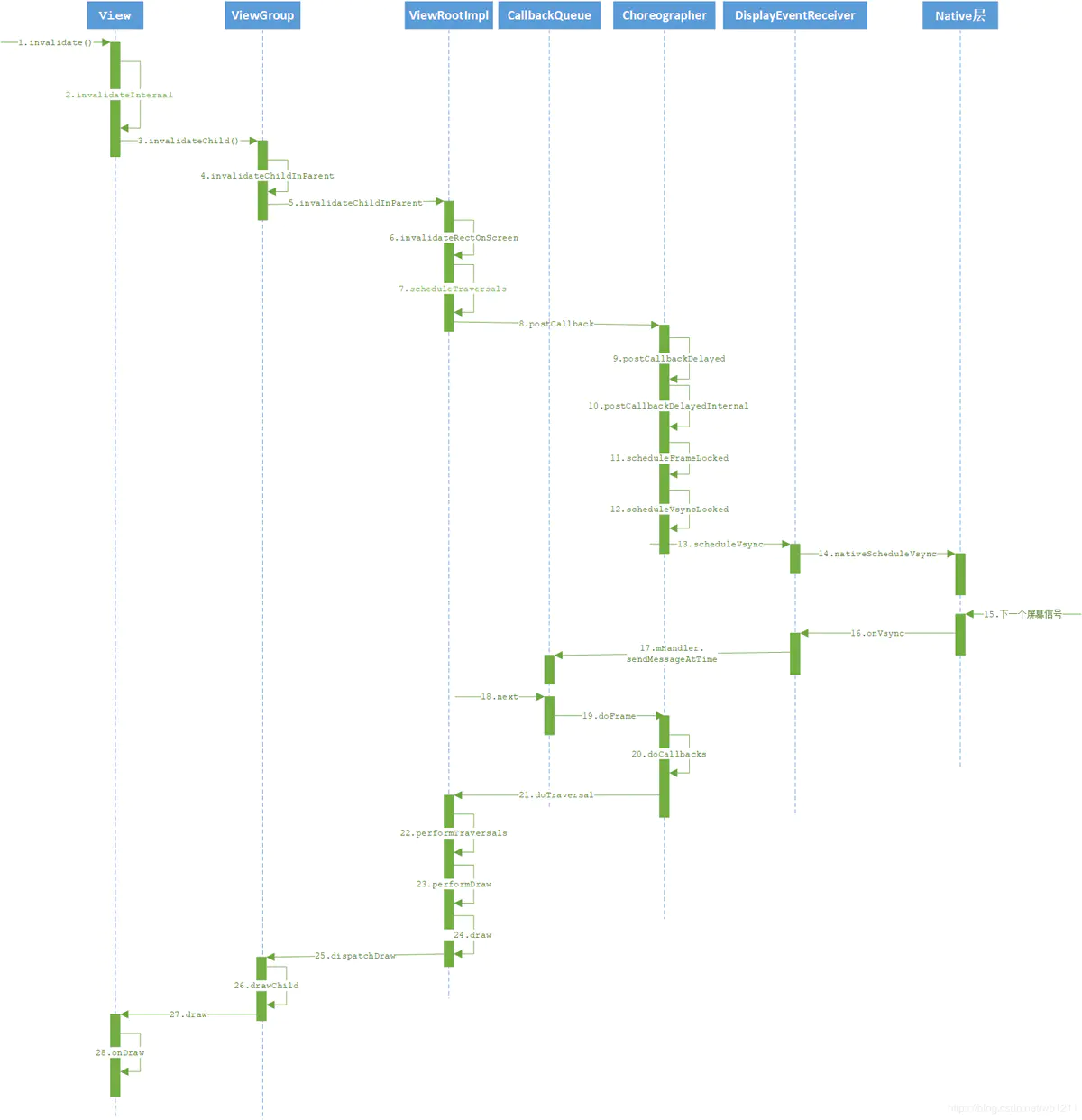
所以，如果队头是一个同步屏障的消息的话，那么在它后面的所有同步消息就都被拦截住了，直到这个同步屏障消息被移除出队列，否则主线程就一直不会去处理同步屏幕后面的同步消息。而所有消息默认都是同步消息，只有手动设置了异步标志，这个消息才会是异步消息。Choreographer里所有跟message有关的代码，都设置了异步消息的标志，所以这些操作是不受到同步屏障影响的。这样做的原因可能就是为了尽可能保证上层app在接收到屏幕刷新信号时，可以在第一时间执行遍历绘制View树的工作。

#### 2.5 小节总结

对整个过程的步骤进行梳理如下：

1. 界面上任何一个 View 的刷新请求最终都会走到 ViewRootImpl 中的 scheduleTraversals() 里来安排一次遍历绘制 View 树的任务；
2. scheduleTraversals() 会先过滤掉同一帧内的重复调用，确保同一帧内只需要安排一次遍历绘制 View 树的任务，遍历过程中会将所有需要刷新的 View 进行重绘。
3. scheduleTraversals() 会往主线程的消息队列中发送一个同步屏障，拦截这个时刻之后所有的同步消息的执行，但不会拦截异步消息，以此来尽可能的保证当接收到屏幕刷新信号时可以尽可能第一时间处理遍历绘制 View 树的工作。
4. 发完同步屏障后scheduleTraversals() 将 performTraversals() 封装到 Runnable 里面，然后调用 Choreographer 的 postCallback() 方法，将这个 Runnable 任务以当前时间戳放进一个待执行的队列里；
5. scheduleTraversals()方法中判断当前线程是否在主线程，如果是在主线程就会直接调用一个native 层方法，如果不是在主线程，会发一个最高优先级的 message 到主线程，让主线程第一时间调用这个 native 层的方法。
6. 调用一个native 层方法向底层订阅下一个屏幕刷新信号Vsync，当下一个屏幕刷新信号发出时，底层就会回调Choreographer 的onVsync()方法来通知上层 app；
7. onVsync() 方法被回调时，会往主线程的消息队列中发送一个执行 doFrame() 方法的异步消息；
8. doFrame()方法会去取出之前放进待执行队列里的任务来执行，取出来的这个任务实际上是 ViewRootImpl的doTraversal() 操作；
9. doTraversal() 中首先移除同步屏障，再调用performTraversals()根据当前状态判断是否需要执行performMeasure()测量、perfromLayout() 布局、performDraw()绘制流程，从而完成View树的刷新，在这几个流程中都会去遍历 View 树来刷新需要更新的View。

等到下一个Vsync信号到达，将上面计算好的数据渲染到屏幕上，同时如果有必要开始下一帧的数据处理。



#### 2.6 常见问题解答

Q1.我们都知道Android每隔16.6ms刷新一次屏幕，是指每隔16.6ms调用一次onDraw()?  
答：我们说的16.6ms刷新一次屏幕其实是指底层会以这个固定频率来切换每一帧画面。

Q2.如果当前界面没有任何变化，还会每隔16.6ms刷新一次屏幕吗？  
答：App并不是每隔16.6ms的屏幕刷新信号都可以接收到，只有当app向底层注册监听下一个屏幕刷新信号之后（DisplayEventReceiver#scheduleVsync()），才能接收到下一个屏幕刷新信号到来的通知。即当某个View发起了刷新请求时，才会去向底层注册监听下一个屏幕刷新信号。  
如果没有发出刷新请求，我们的app会接收不到每隔16.6ms的刷新信号，但底层仍然会每隔16.6ms切换每一帧画面，只不过这些画面都是一样的。

Q3.我们调用完了view的invalidate()方法或者对界面的一些操作后，屏幕会马上刷新吗？  
答：调用invalidate后需要等待下个刷新信号的到来，然后进行屏幕数据的计算，在执行完毕后等待那次的刷新信号到来才会将新的计算结果展示出来。

Q4. 我们说的主线程耗时会导致丢帧，这丢帧到底是怎么发生的？  
答：造成丢帧大体上有两类原因：  
① 遍历绘制View树计算屏幕数据的时间超过了 16.6ms；说明我们写的布局有问题，需要进行优化；  
② 主线程一直在处理其他耗时的消息，导致遍历绘制 View 树的工作迟迟不能开始，从而超过了16.6 ms；这个问题告诉我们避免在主线程做耗时任务；