## 引言

 Android系统的流畅性一直被拿来与iOS比较，并且认为不如后者。这一方面与Android设备硬件质量参差不齐有关，另一方面也与Android系统的实现有关。例如在3.0前，Android应用程序UI绘制不支持硬件加速。不过从4.0开始，Android系统一直以“run fast, smooth, and responsively”为目标对UI进行优化。Android系统的UI从绘制到显示到屏幕是分两步进行的：第一步是在Android应用程序进程这一侧进行的；第二步是在SurfaceFlinger进程这一侧进行的。前一步将UI绘制一个图形缓冲区中，并且将该图形缓冲区交给后一步进行合成以及显示在屏幕中。其中，后一步的UI合成一直都是以硬件加速方式完成的。

## 渲染技术

在支持Android应用程序UI硬件加速渲染之前，Android应用程序UI的绘制是以软件方式进行的，软件渲染技术，如图1所示：

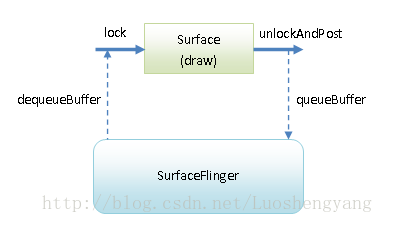


图1 Android应用程序UI软件渲染过程

在Android应用程序进程这一侧，每一个窗口都关联有一个Surface。每当窗口需要绘制UI时，就会调用其关联的Surface的成员函数lock获得一个Canvas，其本质上是向SurfaceFlinger服务Dequeue一个Graphic Buffer。Canvas封装了由Skia提供的2D UI绘制接口，并且都是在前面获得的Graphic Buffer上面进行绘制的。绘制完成之后，Android应用程序进程再调用前面获得的Canvas的成员函数unlockAndPost请求显示在屏幕中，其本质上是向SurfaceFlinger服务Queue一个Graphic Buffer，以便SurfaceFlinger服务可以对Graphic Buffer的内容进行合成，以及显示到屏幕上去。

Android应用程序UI硬件加速渲染技术，如图2所示：

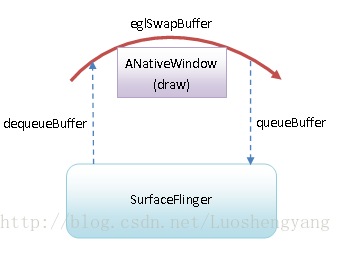


图2 Android应用程序UI硬件加速渲染过程

硬件加速渲染，其实就是通过GPU来进行渲染。GPU作为一个硬件，用户空间是不可以直接使用的，它是由GPU厂商按照Open GL规范实现的驱动间接进行使用的。也就是说，如果一个设备支持GPU硬件加速渲染，那么当Android应用程序调用Open GL接口来绘制UI时，Android应用程序的UI就是通过硬件加速技术进行渲染的。

从图2可以看到，硬件加速渲染和软件渲染一样，在开始渲染之前，都是要先向SurfaceFlinger服务Dequeue一个Graphic Buffer。不过对硬件加速渲染来说，这个Graphic Buffer会被封装成一个ANativeWindow，并且传递给Open GL进行硬件加速渲染环境初始化。在Android系统中，ANativeWindow和Surface可以是认为等价的，只不过是ANativeWindow常用于Native层中，而Surface常用于Java层中。另外，我们还可以将ANativeWindow和Surface看作是像Skia和Open GL这样图形渲染库与操作系统底层的图形系统建立连接的一个桥梁。

 在Android应用程序窗口中，每一个View都抽象为一个Render Node，而且如果一个View设置有Background，这个Background也被抽象为一个Render Node。这是由于在OpenGLRenderer库中，并没有View的概念，所有的一切可绘制的元素都抽象为一个Render Node。

 每一个Render Node都关联有一个Display List Renderer。这里又涉及到另外一个概念——Display List。注意，这个Display List不是Open GL里面的Display List，不过它们在概念上是差不多的。Display List是一个绘制命令缓冲区。也就是说，当View的成员函数onDraw被调用时，我们调用通过参数传递进来的Canvas的drawXXX成员函数绘制图形时，我们实际上只是将对应的绘制命令以及参数保存在一个Display List中。接下来再通过Display List Renderer执行这个Display List的命令，这个过程称为Display List Replay。

注意，只有使用硬件加速渲染的View，才会关联有Render Node，也就才会使用到Display List。我们知道，目前并不是所有的2D UI绘制命令都是GPU可以支持的。

对于使用了GPU不支持的2D UI绘制命令的View，只能通过软件方式来渲染。具体的做法是将创建一个新的Canvas，这个Canvas的底层是一个Bitmap，也就是说，绘制都发生在这个Bitmap上。绘制完成之后，这个Bitmap再被记录在其Parent View的Display List中。而当Parent View的Display List的命令被执行时，记录在里面的Bitmap再通过Open GL命令来绘制。

另一方面，对于前面提到的在Android 4.0引进的TextureView，它也不是通过Display List来绘制。由于它的底层实现直接就是一个Open GL纹理，因此就可以跳过Display List这一中间层，从而提高效率。这个Open GL纹理的绘制通过一个Layer Renderer来封装。Layer Renderer和Display List Renderer可以看作是同一级别的概念，它们都是通过Open GL命令来绘制UI元素的。只不过前者操作的是Open GL纹理，而后者操作的是Display List。

 我们知道，Android应用程序窗口的View是通过树形结构来组织的。这些View不管是通过硬件加速渲染还是软件渲染，或者是一个特殊的TextureView，在它们的成员函数onDraw被调用期间，它们都是将自己的UI绘制在Parent View的Display List中。其中，最顶层的Parent View是一个Root View，它关联的Root Node称为Root Render Node。也就是说，最终Root Render Node的Display List将会包含有一个窗口的所有绘制命令。在绘制窗口的下一帧时，Root Render Node的Display List都会通过一个Open GL Renderer真正地通过Open GL命令绘制在一个Graphic Buffer中。最后这个Graphic Buffer被交给SurfaceFlinger服务进行合成和显示。Render Thread是在Android 5.0中引进的，它用来分担Android应用程序的Main Thread的工作。在Android 5.0之前，Android应用程序的Main Thread不仅负责渲染UI，还负责处理用户输入。通过引进Render Thread，我们就可以将UI渲染工作从Main Thread释放出来，交由Render Thread来处理，从而也使得Main Thread可以更专注高效地处理用户输入，这样使得在提高UI绘制效率的同时，也使得UI具有更高的响应性。Main Thread与Render Thread的交互模型如图3所示：

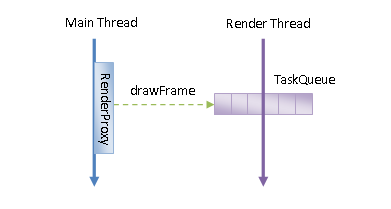


图4 Android应用程序Main Thread与Render Thread的交互模型

Main Thread主要是负责调用View的成员函数onDraw来构造它们的Display List，然后在下一个Vsync信号到来时，再通过一个Render Proxy对象向Render Thread发出一个drawFrame命令。Render Thread内部有一个Task Queue，从Main Thread发送过来的drawFrame命令就会保存在Render Thread的Task Queue，等待Render Thread处理。

当一个View需要以动画的形式显示时，我们可以通过调用这个View的成员函数animate获得一个ViewPropertyAnimator。ViewPropertyAnimator可以通过两种方式来显示动画。第一种方式是旧的方式，动画的每一帧由Main Thread进行计算，然后再由Render Thread进行渲染。第二种方式是Android 5.0引进的，Main Thread将动画注册到Render Thread中去，然后由Render Thread计算和显示动画的每一帧。第二种方式在动画的显示期间，完全不需要Main Thread参与，不过前提是Main Thread不想参与。如果Main Thread需要参与，例如，Main Thread想在动画开始之前，将View的Layer Type设置为LAYER\_TYPE\_HARDWARE，以便它可以以Open GL的Frame Buffer Object（FBO）的形式进行渲染，或者Main Thread想侦听动画的每一帧显示，那么就不能将动画的计算和显示完全交给Render Thread来做了。将动画的计算和显示完全交给Render Thread来做的好处就是使得动画的显示不影响Main Thread响应用户的其它输入。

从View获得一个用来显示动画的ViewPropertyAnimator之后，调用它的成员函数withLayer，就会导致Main Thread向Render Thread发送一个buildLayer的命令。Render Thread执行这个buildLayer的命令的时候，就会为与View关联的Render Node设置一个Layer。以后Render Thread就以Layer的方式，即FBO的方式，来渲染View的动画。

另一方面，如果一个View的动画显示不需要Main Thread的参与，那么从View获得一个ViewPropertyAnimator会将动画注册到Render Thread里面的一个AnimatorManager中。Render Thread通过AnitmatorManager检测注册到它里面的动画是否还没有结束。如果还没有结束，那么Render Thread就自动地计算和显示动画的下一帧，直到动画显示结束为止。

## UI硬件加速渲染环境初始化

 在Android应用程序中，我们是通过Canvas API来绘制UI元素的。在硬件加速渲染环境中，这些Canvas API调用最终会转化为Open GL API调用（转化过程对应用程序来说是透明的）。由于Open GL API调用要求发生在Open GL环境中，因此在每当有新的Activity窗口启动时，系统都会为其初始化好Open GL环境。

在Android 5.0之前，Android应用程序的主线程同时也是一个Open GL线程。但是从Android 5.0之后，Android应用程序的Open GL线程就独立出来了，称为Render Thread。Render Thread有一个Task Queue，Main Thread通过一个代理对象Render Proxy向这个Task Queue发送一个drawFrame命令，从而驱使Render Thread执行一次渲染操作。因此，Android应用程序UI硬件加速渲染环境的初始化过程任务之一就是要创建一个Render Thread。

一个Android应用程序可能存在多个Activity组件。在Android系统中，每一个Activity组件都是一个独立渲染的窗口。由于一个Android应用程序只有一个Render Thread，因此当Main Thread向Render Thread发出渲染命令时，Render Thread要知道当前要渲染的窗口是什么。从这个角度看，Android应用程序UI硬件加速渲染环境的初始化过程任务之二就是要告诉Render Thread当前要渲染的窗口是什么。 一旦Render Thread知道了当前要渲染的窗口，它就将可以将该窗口绑定到Open GL渲染上下文中去，从而使得后面的渲染操作都是针对被绑定的窗口的，

Java层的Activity窗口到了Open GL这一层，被抽象为一个ANativeWindow。将它绑定到Open GL渲染上下文之后，就可以通过eglSwapBuffer函数向SurfaceFlinger服务Dequeue和Queue Graphic Buffer。其中，Dequeue Graphic Buffer是为了在上面进行绘制UI，而Queue Graphic Buffer是为了将绘制好的UI交给Surface Flinger合成和显示。

Activity组件在创建的过程中，也就是在其生命周期函数onCreate的调用过程中，一般会通过调用另外一个成员函数setContentView创建和初始化关联的窗口视图，最后通过调用ViewRoot类的成员函数setView完成这一过程。Android应用程序UI硬件加速渲染环境的初始化过程是在ViewRootImpl类的成员函数setView开始，如下所示：

public final class ViewRootImpl implements ViewParent,

View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {

......

public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {

synchronized (this) {

if (mView == null) {

mView = view;

......

if (view instanceof RootViewSurfaceTaker) {

mSurfaceHolderCallback =

((RootViewSurfaceTaker)view).willYouTakeTheSurface();

if (mSurfaceHolderCallback != null) {

mSurfaceHolder = new TakenSurfaceHolder();

mSurfaceHolder.setFormat(PixelFormat.UNKNOWN);

}

}

......

// If the application owns the surface, don't enable hardware acceleration

if (mSurfaceHolder == null) {

enableHardwareAcceleration(attrs);

}

......

}

}

}

......

}

参数view描述的是当前正在创建的Activity窗口的顶级视图。如果它实现了RootViewSurfaceTaker接口，并且通过该接口的成员函数willYouTakeTheSurface提供了一个SurfaceHolder.Callback2接口，那么就表明应用程序想自己接管对窗口的一切渲染操作。这样创建出来的Activity窗口就类似于一个SurfaceView一样，完全由应用程序自己来控制它的渲染。

如果当前创建的窗口支持硬件加速渲染，那么就会调用HardwareRenderer类的静态成员函数create创建一个HardwareRenderer对象，并且保存在与该窗口关联的一个AttachInfo对象的成员变量的成员变量mHardwareRenderer对象。这个HardwareRenderer对象以后将负责执行窗口硬件加速渲染的相关操作。

 HardwareRenderer类的静态成员函数create的实现如下所示：

public abstract class HardwareRenderer {

......

static HardwareRenderer create(Context context, boolean translucent) {

HardwareRenderer renderer = null;

if (GLES20Canvas.isAvailable()) {

renderer = new ThreadedRenderer(context, translucent);

}

return renderer;

}

......

}

 从这里就可以看到，在设备支持Open GL ES 2.0的情况下，HardwareRenderer类的静态成员函数create创建的实际上是一个ThreadedRenderer对象。该ThreadedRenderer对象是从HardwareRenderer类继承下来的。

 接下来我们就继续分析ThreadedRenderer对象的创建过程，如下所示：

public class ThreadedRenderer extends HardwareRenderer {

......

private long mNativeProxy;

......

private RenderNode mRootNode;

......

ThreadedRenderer(Context context, boolean translucent) {

......

long rootNodePtr = nCreateRootRenderNode();

mRootNode = RenderNode.adopt(rootNodePtr);

......

mNativeProxy = nCreateProxy(translucent, rootNodePtr);

AtlasInitializer.sInstance.init(context, mNativeProxy);

......

}

......

}

在创建ThreadedRenderer对象的过程中，最主要的是做了三件事情：

1. 调用ThreadedRenderer类的成员函数nCreateRootRenderNode在Native层创建了一个Render Node，并且通过Java层的RenderNode类的静态成员函数adopt将其封装在一个Java层的Render Node中。这个Render Node即为窗口的Root Render Node。

2. 调用ThreadedRenderer类的成员函数nCreateProxy在Native层创建了一个Render Proxy对象。该Render Proxy对象以后将负责从Main Thread向Render Thread发送命令。

3. 调用AtlasInitializer类的成员函数init初始化一个系统预加载资源的地图集。通过这个地图集，可以优化资源的内存使用。

  窗口的Root Render Node是通过调用ThreadedRenderer类的成员函数nCreateRootRenderNode创建的。这是一个JNI函数，由Native层的函数android\_view\_ThreadedRenderer\_createRootRenderNode实现，如下所示：

static jlong android\_view\_ThreadedRenderer\_createRootRenderNode(JNIEnv\* env, jobject clazz) {

RootRenderNode\* node = new RootRenderNode(env);

node->incStrong(0);

node->setName("RootRenderNode");

return reinterpret\_cast<jlong>(node);

}

从这里就可以看出，窗口在Native层的Root Render Node实际上是一个RootRenderNode对象。

窗口在Main Thread线程中使用的Render Proxy对象是通过调用ThreadedRenderer类的成员函数nCreateProxy创建的。这是一个JNI函数，由Native层的函数android\_view\_ThreadedRenderer\_createProxy实现，如下所示：

static jlong android\_view\_ThreadedRenderer\_createProxy(JNIEnv\* env, jobject clazz,

jboolean translucent, jlong rootRenderNodePtr) {

RootRenderNode\* rootRenderNode = reinterpret\_cast<RootRenderNode\*>(rootRenderNodePtr);

ContextFactoryImpl factory(rootRenderNode);

return (jlong) new RenderProxy(translucent, rootRenderNode, &factory);

}

参数rootRenderNodePtr指向前面创建的RootRenderNode对象。有了这个RootRenderNode对象之后，函数android\_view\_ThreadedRenderer\_createProxy就创建了一个RenderProxy对象。RenderProxy对象的创建过程如下所示：

RenderProxy::RenderProxy(bool translucent, RenderNode\* rootRenderNode, IContextFactory\* contextFactory)

: mRenderThread(RenderThread::getInstance())

, mContext(0) {

SETUP\_TASK(createContext);

args->translucent = translucent;

args->rootRenderNode = rootRenderNode;

args->thread = &mRenderThread;

args->contextFactory = contextFactory;

mContext = (CanvasContext\*) postAndWait(task);

mDrawFrameTask.setContext(&mRenderThread, mContext);

}

参数rootRenderNodePtr指向前面创建的RootRenderNode对象。有了这个RootRenderNode对象之后，函数android\_view\_ThreadedRenderer\_createProxy就创建了一个RenderProxy对象。

RenderProxy::RenderProxy(bool translucent, RenderNode\* rootRenderNode, IContextFactory\* contextFactory)

: mRenderThread(RenderThread::getInstance())

, mContext(0) {

SETUP\_TASK(createContext);

args->translucent = translucent;

args->rootRenderNode = rootRenderNode;

args->thread = &mRenderThread;

args->contextFactory = contextFactory;

mContext = (CanvasContext\*) postAndWait(task);

mDrawFrameTask.setContext(&mRenderThread, mContext);

}

   RenderProxy类有三个重要的成员变量mRenderThread、mContext和mDrawFrameTask，它们的类型分别为RenderThread、CanvasContext和DrawFrameTask。其中，mRenderThread描述的就是Render Thread，mContext描述的是一个画布上下文，mDrawFrameTask描述的是一个用来执行渲染任务的Task。接下来我们就重点分析这三个成员变量的初始化过程。

 RenderProxy类的成员变量mRenderThread指向的Render Thread是通过调用RenderThread类的静态成员函数getInstance获得的。从名字我们就可以看出，RenderThread类的静态成员函数getInstance返回的是一个RenderThread单例。也就是说，在一个Android应用程序进程中，只有一个Render Thread存在。Render Thread的创建过程，如下所示：

RenderThread::RenderThread() : Thread(true), Singleton<RenderThread>()

...... {

mFrameCallbackTask = new DispatchFrameCallbacks(this);

mLooper = new Looper(false);

run("RenderThread");

}

RenderThread类的成员变量mFrameCallbackTask指向一个DispatchFrameCallbacks对象，用来描述一个帧绘制任务。

RenderThread类是从Thread类继承下来的，当我们调用它的成员函数run的时候，就会创建一个新的线程。这个新的线程的入口点函数为RenderThread类的成员函数threadLoop，它的实现如下所示：

bool RenderThread::threadLoop() {

.......

initThreadLocals();

int timeoutMillis = -1;

for (;;) {

int result = mLooper->pollOnce(timeoutMillis);

......

nsecs\_t nextWakeup;

// Process our queue, if we have anything

while (RenderTask\* task = nextTask(&nextWakeup)) {

task->run();

// task may have deleted itself, do not reference it again

}

if (nextWakeup == LLONG\_MAX) {

timeoutMillis = -1;

} else {

nsecs\_t timeoutNanos = nextWakeup - systemTime(SYSTEM\_TIME\_MONOTONIC);

timeoutMillis = nanoseconds\_to\_milliseconds(timeoutNanos);

if (timeoutMillis < 0) {

timeoutMillis = 0;

}

}

if (mPendingRegistrationFrameCallbacks.size() && !mFrameCallbackTaskPending) {

drainDisplayEventQueue(true);

mFrameCallbacks.insert(

mPendingRegistrationFrameCallbacks.begin(), mPendingRegistrationFrameCallbacks.end());

mPendingRegistrationFrameCallbacks.clear();

requestVsync();

}

}

return false;

}

这里我们就可以看到Render Thread的运行模型：

1. 空闲的时候，Render Thread就睡眠在成员变量mLooper指向的一个Looper对象的成员函数pollOnce中。

2. 当其它线程需要调度Render Thread，就会向它的任务队列增加一个任务，然后唤醒Render Thread进行处理。Render Thread通过成员函数nextTask获得需要处理的任务，并且调用它的成员函数run进行处理。

RenderThread类的成员函数nextTask的实现如下所示：

RenderTask\* RenderThread::nextTask(nsecs\_t\* nextWakeup) {

AutoMutex \_lock(mLock);

RenderTask\* next = mQueue.peek();

if (!next) {

mNextWakeup = LLONG\_MAX;

} else {

mNextWakeup = next->mRunAt;

// Most tasks won't be delayed, so avoid unnecessary systemTime() calls

if (next->mRunAt <= 0 || next->mRunAt <= systemTime(SYSTEM\_TIME\_MONOTONIC)) {

next = mQueue.next();

} else {

next = 0;

}

}

if (nextWakeup) {

\*nextWakeup = mNextWakeup;

}

return next;

}

RenderThread类的成员变量mQueue描述的是一个Task Queue。每一个Task都是用一个RenderTask对象来描述。同时，RenderTask类有一个成员变量mRunAt，用来表明Task的执行时间。这样，保存在Task Queue的Task就可以按照执行时间从先到后的顺序排序。于是，RenderThread类的成员函数nextTask通过判断排在队列头的Task的执行时间是否小于等于当前时间，就可以知道当前是否有Task需要执行。如果有Task需要执行的话，就将它返回给调用者。

 RenderThread类的成员函数nextTask除了返回下一个要执行的Task之外，还会通过参数nextWakeup返回下一个要执行的Task的执行时间。这个时间同时也会记录在RenderThread类的成员变量mNextWakeup中。注意，下一个要执行的Task可能是马上就要执行的，也有可能是由于执行时间还未到而不能执行的Task。返回这个时间的意义是使得Render Thread可以准确计算下一次需要进入睡眠状态的时间。这个计算可以回过头去看前面分析的RenderThread类的成员函数threadLoop。 注意，如果没有下一个任务可以执行，那么RenderThread类的成员函数nextTask通过参数nextWakeup返回的值为LLONG\_MAX，表示Render Thread接下来无限期进入睡眠状态，直到被其它线程唤醒为止。

RenderThread类提供了queue、queueAtFront和queueDelayed三个成员函数向Task Queue增加一个Task，它们的实现如下所示：

void RenderThread::queue(RenderTask\* task) {

AutoMutex \_lock(mLock);

mQueue.queue(task);

if (mNextWakeup && task->mRunAt < mNextWakeup) {

mNextWakeup = 0;

mLooper->wake();

}

}

void RenderThread::queueAtFront(RenderTask\* task) {

AutoMutex \_lock(mLock);

mQueue.queueAtFront(task);

mLooper->wake();

}

void RenderThread::queueDelayed(RenderTask\* task, int delayMs) {

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

task->mRunAt = now + milliseconds\_to\_nanoseconds(delayMs);

queue(task);

}

 其中， RenderThread类的成员函数queue按照执行时间将参数task描述的Task排列在Task Queue中，并且如果该Task的执行时间小于之前记录的下一个要执行任务的执行时间，就会马上唤醒Render Thread来处理；RenderThread类的成员函数queue将参数task描述的Task排列在Task Queue的头部，并且马上唤醒Render Thread来处理；RenderThread类的成员函数queueDelayed指定参数task描述的Task的执行时间为当前时间之后的delayMs毫秒。

Render Thread在进入无限循环之前调用的RenderThread类的成员函数initThreadLocals，它的实现如下所示：

void RenderThread::initThreadLocals() {

initializeDisplayEventReceiver();

mEglManager = new EglManager(\*this);

mRenderState = new RenderState();

}

 RenderThread类的成员函数initThreadLocals首先调用另外一个成员函数initializeDisplayEventReceiver创建和初始化一个DisplayEventReceiver对象，用来接收Vsync信号。接着又会分别创建一个EglManager对象和一个RenderState对象，并且保存在成员变量mEglManager和mRenderState中。前者用在初始化Open GL渲染上下文需要用到，而后者用来记录Render Thread当前的一些渲染状态。

DisplayEventReceiver对象的创建和初始化过程，即RenderThread类的成员函数initializeDisplayEventReceiver的实现，如下所示：

void RenderThread::initializeDisplayEventReceiver() {

......

mDisplayEventReceiver = new DisplayEventReceiver();

......

// Register the FD

mLooper->addFd(mDisplayEventReceiver->getFd(), 0,

Looper::EVENT\_INPUT, RenderThread::displayEventReceiverCallback, this);

}

 创建的DisplayEventReceiver对象关联的文件描述符被注册到了Render Thread的消息循环中。这意味着屏幕产生Vsync信号时，SurfaceFlinger服务（Vsync信号由SurfaceFlinger服务进行管理和分发）会通过上述文件描述符号唤醒Render Thread。这时候Render Thread就会调用RenderThread类的静态成员函数displayEventReceiverCallback。

RenderThread类的静态成员函数displayEventReceiverCallback的实现如下所示：

int RenderThread::displayEventReceiverCallback(int fd, int events, void\* data) {

......

reinterpret\_cast<RenderThread\*>(data)->drainDisplayEventQueue();

return 1; // keep the callback

}

RenderThread类的静态成员函数displayEventReceiverCallback的实现如下所示：

int RenderThread::displayEventReceiverCallback(int fd, int events, void\* data) {

......

reinterpret\_cast<RenderThread\*>(data)->drainDisplayEventQueue();

return 1; // keep the callback

}

RenderThread类的成员变量mFrameCallbackTask描述的Task是用来显示动画的。当Java层注册一个动画类型的Render Node到Render Thread时，一个类型为IFrameCallback的回调接口就会通过RenderThread类的成员函数postFrameCallback注册到Render Thread的一个Pending Registration Frame Callbacks列表中，如下所示：

void RenderThread::postFrameCallback(IFrameCallback\* callback) {

mPendingRegistrationFrameCallbacks.insert(callback);

}

当Pending Registration Frame Callbacks列表不为空时，每次Vsync信号到来时，Render Thread都会通过RenderThread类的成员变量mFrameCallbackTask描述的一个Task来执行它，这样就相当于是将动画的每一帧都同步到Vsync信号来显示。这也是为什么RenderThread类的成员函数drainDisplayEventQueue每次被调用要检查是否需要将成员变量mFrameCallbackTask描述的一个Task添加到Render Thread的Task Queue的原因。当RenderThread类的成员变量mFrameCallbackTask描述的Task的类型为DispatchFrameCallbacks，当它被调度执行时，它的成员函数run就会被调用，如下所示：

class DispatchFrameCallbacks : public RenderTask {

private:

RenderThread\* mRenderThread;

public:

DispatchFrameCallbacks(RenderThread\* rt) : mRenderThread(rt) {}

virtual void run() {

mRenderThread->dispatchFrameCallbacks();

}

};

 DispatchFrameCallbacks类的成员函数run调用了RenderThread类的成员函数dispatchFrameCallbacks来执行注册到Pending Registration Frame Callbacks列表中的IFrameCallback回调接口，如下所示：

void RenderThread::dispatchFrameCallbacks() {

ATRACE\_CALL();

mFrameCallbackTaskPending = false;

std::set<IFrameCallback\*> callbacks;

mFrameCallbacks.swap(callbacks);

for (std::set<IFrameCallback\*>::iterator it = callbacks.begin(); it != callbacks.end(); it++) {

(\*it)->doFrame();

}

}

我们回过头来看前面分析的RenderThread类的成员函数threadLoop，每当Render Thread被唤醒时，它都会检查Pending Registration Frame Callbacks列表是否不为空。如果不为空，那么就会将保存在里面的IFrameCallback回调接口转移至由RenderThread类的成员变量mFrameCallbacks描述的另外一个IFrameCallback回调接口列表中，并且调用RenderThread类的另外一个成员函数requestVsync请求SurfaceFlinger服务在下一个Vsync信号到来时通知Render Thread，以便Render Thread可以执行刚才被转移的IFrameCallback回调接口。

现在既然下一个Vsync信号已经到来，因此RenderThread类的成员函数dispatchFrameCallbacks就执行所有转移至保存在成员变量mFrameCallbacks描述的IFrameCallback回调接口列表中的IFrameCallback接口，即调用它们的成员函数doFrame。

Render Thread在运行时主要是做以下两件事情：

 1. 执行Task Queue的任务，这些Task一般就是由Main Thread发送过来的，例如，Main Thread通过发送一个Draw Frame Task给Render Thread的Task Queue中，请求Render Thread渲染窗口的下一帧。

 2. 执行Pending Registration Frame Callbacks列表的IFrameCallback回调接口。每一个IFrameCallback回调接口代表的是一个动画帧，这些动画帧被同步到Vsync信号到来由Render Thread自动执行。具体来说，就是每当Vsync信号到来时，就将一个类型为DispatchFrameCallbacks的Task添加到Render Thread的Task Queue去等待调度。一旦该Task被调度，就可以在Render Thread中执行注册在Pending Registration Frame Callbacks列表中的IFrameCallback回调接口了。