Android面试题之刷新机制，包括View是如何在16.7ms内完成画面刷新、注册监听ASYNC信号、同步屏障等。

> 本文是我一点点归纳总结的干货，但是难免有疏忽和遗漏，希望不吝赐教。
> 转载请注明链接：https://blog.csdn.net/feather_wch/article/details/81437689

有帮助的话请点个赞！万分感谢！

Android面试题-刷新机制详解(9题)

版本:2018/8/5-1(23:23)

1、界面的刷新为什么需要16.6ms?

> 1. 系统每16.6ms会发出一个 VSYNC信号 ，发出信号后，才会开始进行测量、布局和绘制。
> 2. 发出 VSYNC信号 时，还会将此时 显示器的buffer缓冲区 的数据取出，并显示在屏幕上。

2、画面的显示需要哪些步骤?

> 1. CPU计算数据(View树遍历并执行三大流程：测量、布局和绘制)，然后将数据交给GPU``
> 2. GPU渲染处理，然后将数据放到 Buffer 中。
> 3. 显示屏(display)从 buffer 中取出数据，并进行显示。

3、界面保持不变时，还会16.6ms刷新一次屏幕吗?

> 1. 对于底层显示器，每间隔16.6ms接收到 VSYNC信号 时，就会用 buffer 中数据进行一次显示。所以一定会刷新。

4、界面刷新的本质流程

> 1. 通过 ViewRootImpl 的 scheduleTraversals() 进行界面的三大流程。
> 2. 调用到 scheduleTraversals() 时不会立即执行，而是将该操作保存到 待执行队列 中。并给底层的刷新信号注册监听。
> 3. 当 VSYNC 信号到来时，会从 待执行队列 中取出对应的 scheduleTraversals() 操作，并将其加入到 主线程 的 消息队列中 。
> 4. 主线程 从 消息队列中 取出并执行 三大流程: onMeasure()-onLayout()-onDraw()

5、View的界面刷新方法最终都会执行到 ViewRootImpl的scheduleTraversals()

> 1. invalidate(请求重绘)

2. requestLayout(重新布局)
3. requestFocus(请求焦点)等
   界面刷新操作会从View树向上层层找到最顶层的 `DecorView` ， 然后通过 `DecorView` 的 `mParent` 也就是 `ViewRootImpl` 执行 `scheduleTraversals()` 方法。

6、ViewRootImpl如何和DecorView绑定起来?

1. Activity的启动在 `ActivityThread` 中完成， `handleLaunchActivity()` 会依次间接执行到 `onCreate()-onStart()-onResume()`
2. 之后会调用 `WindowManager` 的 `addView()` 将 `View` 和 `Window` 关联起来。
3. `addView()` 会创建 `ViewRootImpl` 并调用其 `setView(decorView)` ， 内部调用 `decorView.assignParent(this)` ， 将 `ViewRootImpl` 设置为 `DecorView`的`mParent` 。

7、ViewRootImpl的scheduleTraversals()源码解析

主要分为两部分：

1. 将界面刷新操作打包后加入到待执行队列中，并监听下一次VSYNC信号。
2. 接收到VSYNC信号后，进行界面刷新-测量、布局、绘制 。

```
/**============================================================*
 * 上层app请求界面刷新的主要思路：
 *    1. 不会立即执行performTraversals()-测量、布局、绘制三大流程
 *    2. 将performTraversals()方法封装到Runnable中，保存到“待执行队列中”
 *    3. 在DisplayEventReceiver中注册监听底层的VSYNC信号
 * //ViewRootImpl.java
 *============================================================*/
final TraversalRunnable mTraversalRunnable = new TraversalRunnable();
//ViewRootImpl.java
void scheduleTraversals() {
    //1. mTraversalScheduled避免一帧数据内重复提交刷新请求(仅仅会在VYSNC信号后调用的doTraversa
    if (!mTraversalScheduled) {
        mTraversalScheduled = true;
        //2. 发送同步屏障
        mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();
        //3. 本质将TraversalRunnable存放到“待执行队列中”，等待接收到VSYNC信号后取出并执行
        mChoreographer.postCallback(Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, n
        ......
    }
}
//Choreographer.java
public void postCallback(int callbackType, Runnable action, Object token) {
    //层层执行
    postCallbackDelayed(callbackType, action, token, 0);
}
public void postCallbackDelayed(int callbackType, Runnable action, Object token, long delay
    //层层执行
    postCallbackDelayedInternal(callbackType, action, token, delayMillis);
}

//Choreographer.java
private void postCallbackDelayedInternal(int callbackType, Object action, Object token, lon
    synchronized (mLock) {
        //1. 时间戳
        final long now = SystemClock.uptimeMillis();
        final long dueTime = now + delayMillis;
        //2. 放置到“待执行队列”中，以“时间戳”进行排序的队列
        mCallbackQueues[callbackType].addCallbackLocked(dueTime, action, token);
        //3. 层层执行到DisplayEventReceiver.java的native方法
        scheduleFrameLocked(now);
    }

}

//Choreographer.java
private void scheduleFrameLocked(long now) {
    //1. 最终都会在主线程中执行该方法(会受到“同步屏障”的保护而优先执行)
    scheduleVsyncLocked();
    ...
}
//Choreographer.java
private void scheduleVsyncLocked() {
    //2. 层层执行
    mDisplayEventReceiver.scheduleVsync();
```

```
}
//DisplayEventReceiver.java
public void scheduleVsync() {
    //3. 最终会执行到native方法
    nativeScheduleVsync(mReceiverPtr);
}

/**============================================================*
 * 底层VSYNC信号触发上层app进行三大流程的主要思路：
 *    1. FrameDisplayEventReceiver继承自DisplayEventReceiver
 *    2. VSYNC信号由“SurfaceFlinger”实现并定时发送，最终回调onVsync()方法
 *    3. 通过异步Message切换到UI线程中，然后从“待执行队列”中取出Runnable
 *    4. 执行TraversalRunnable的run()->doTraversal()->performTraversals()
 * //Choreographer.java
 *============================================================*/
private final class FrameDisplayEventReceiver extends DisplayEventReceiver implements Runna
    //1. 底层会回调App的onVsync()方法
    public void onVsync(long timestampNanos, int builtInDisplayId, int frame) {
        //2. 通过Handler切换到主线程，去执行run()方法中的doFrame()
        Message msg = Message.obtain(mHandler, this);
        msg.setAsynchronous(true); //异步Message以防止同步屏障的拦截，具有最高优先级
        mHandler.sendMessageAtTime(msg, timestampNanos / TimeUtils.NANOS_PER_MS);
    }
    //3. 需要在UI线程执行
    public void run() {
        doFrame(mTimestampNanos, mFrame);
    }
}
//Choreographer.java
void doFrame(long frameTimeNanos, int frame) {
    doCallbacks(Choreographer.CALLBACK_TRAVERSAL, frameTimeNanos);
}
//Choreographer.java
void doCallbacks(int callbackType, long frameTimeNanos) {
    //1. 取出“待执行队列”中的TraversalRunnable(CallbackQueue的extractDueCallbacksLocked方法)
    callbacks = mCallbackQueues[callbackType].extractDueCallbacksLocked(
            now / TimeUtils.NANOS_PER_MS);
    //2. 执行TraverslaRunnable的run方法
    for (CallbackRecord c = callbacks; c != null; c = c.next) {
        c.run(frameTimeNanos);
    }
}
//ViewRootImpl.java
final class TraversalRunnable implements Runnable {
    @Override
    public void run() {
        doTraversal();
    }
}
//ViewRootImpl.java
void doTraversal() {
    if (mTraversalScheduled) {
        //1. 避免一帧内重复进行刷新
        mTraversalScheduled = false;
        //2. 移除主线程消息队列中的同步屏障
```

```
            mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);
            //3. 执行三大流程
            performTraversals();
        }
    }
    //ViewRootImpl.java
    private void performTraversals() {
        //1. View树的测量-可以不测量直接走布局和绘制
        if (mFirst || windowShouldResize || insetsChanged || viewVisibilityChanged || ...) {
            performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);
            layoutRequested = true;  //需要进行布局
        }
        //2. View树的布局-可以不布局直接走绘制
        final boolean didLayout = layoutRequested && (!mStopped || mReportNextDraw);
        if (didLayout) {
            performLayout(lp, mWidth, mHeight);
        }
        //3. View树的绘制-可以不进行绘制
        boolean cancelDraw = mAttachInfo.mTreeObserver.dispatchOnPreDraw() || !isViewVisible;
        if (!cancelDraw && !newSurface) {
            performDraw();
        }
    }
```

8、测量、布局、绘制三大流程是否一定要都执行?

> 不是。
> 会根据条件执行部分流程。如：申请重新布局时就不会重新测量(requestLayout)
> 走 测量 流程就一定需要走 布局 流程，但不一定走 绘制 流程，因为可能View处于不可见状态。

9、同步屏障的作用和原理?

> 1. 同步屏障 用于 阻塞 住所有的 同步消息 (底层VSYNC的回调onVsync方法提交的消息是 异步消息)
> 2. 用于保证 界面刷新功能的performTraversals() 的优先执行。
> 3. 消息默认为 同步消息 ， 异步消息 只能由内部发送
> 4. 同步屏障 原理是：主线程的 Looper 会一直循环调用 MessageQueue 的 next 方法取出 队列头部的Message 执行，遇到 同步屏障(一种特殊消息) 后会去寻找 异步消息 执行。如果没有找到 异步消息 就会一直阻塞下去，除非将 同步屏障 取出，否则永远不会执行 同步消息 。

参考资料

1. Android 屏幕刷新机制
2. 破译Android性能优化中的16ms问题
3. android屏幕刷新显示机制
4. Android Choreographer 源码分析