前言

Window、Activity、View都经常用到，但三者关系还是没有系统的理清，今天咱们就开始整理整理这三者的关系：

* **Window**：顶级窗口外观和行为策略的抽象基类。唯一实现是 PhoneWindow类。
* **Activity**：四大组件之一，它提供一个界面让用户点击和各种滑动操作。
* **View**：代表用户界面组件的基本构建块，UI 组件。

源码：Android SDK 30

Activity.setContentView

不知道从哪入手，咱们就从setContentView开始，比较刚创建的Android项目Activity和setContentView是必不可少的。记住看Activity的setContentView而不是AppCompatActivity的

来瞅瞅setContentView 源码：

**public void setContentView(@LayoutRes int layoutResID) {**

**//重点来了**

**getWindow().setContentView(layoutResID);**

**//创建并设置ActionBar,**

**initWindowDecorActionBar();**

**}**

getWindow？这么快就遇到了，好简单，咱继续看。

Activity.getWindow

**public Window getWindow() {**

**return mWindow;**

**}**

本来以为发现大鱼了，显然 Activity 几乎什么都没做，将操作直接交给了一个 Window 来处理。getWindow 返回的是 Activity 中的全局变量 mWindow，它是 Window 窗口类型。那么它是什么时候赋值的呢？

Activity.attach()

找遍了整个Activity源码终于在 attach 方法找到了。咱们先看看代码。

**final void attach(Context context, ActivityThread aThread,**

**Instrumentation instr, IBinder token, int ident,**

**Application application, Intent intent, ActivityInfo info,**

**CharSequence title, Activity parent, String id,**

**NonConfigurationInstances lastNonConfigurationInstances,**

**Configuration config, String referrer, IVoiceInteractor voiceInteractor,**

**Window window, ActivityConfigCallback activityConfigCallback, IBinder assistToken) {**

**attachBaseContext(context);**

**mFragments.attachHost(null /\*parent\*/);**

**//重点来了对PhoneWindow进行实例化**

**mWindow = new PhoneWindow(this, window, activityConfigCallback);**

**mWindow.setWindowControllerCallback(mWindowControllerCallback);**

**mWindow.setCallback(this);**

**mWindow.setOnWindowDismissedCallback(this);**

**mWindow.getLayoutInflater().setPrivateFactory(this);**

**...**

**//调用setWindowManager**

**mWindow.setWindowManager(**

**(WindowManager)context.getSystemService(Context.WINDOW\_SERVICE),**

**mToken, mComponent.flattenToString(),**

**(info.flags & ActivityInfo.FLAG\_HARDWARE\_ACCELERATED) != 0);**

**...**

**}**

**Window**：实际上整个 Android 系统中 Window 只有一个实现类，就是 PhoneWindow。 **PhoneWindow**：Android-specific Window(特定窗口).

接下来调用 **setWindowManager** 方法，将系统 **WindowManager** 传给 PhoneWindow。

WindowManager对Window进行管理，说到管理那就离不开对Window的添加、更新和删除的操作，在这里我们把它们统称为Window的操作。对于Window的操作，最终都是交由WMS来进行处理。

Window.setWindowManager

**public void setWindowManager(WindowManager wm, IBinder appToken, String appName,**

**boolean hardwareAccelerated) {**

**mAppToken = appToken;**

**mAppName = appName;**

**mHardwareAccelerated = hardwareAccelerated;**

**if (wm == null) {**

**//通过Binder机制来获取WMS**

**wm = (WindowManager)mContext.getSystemService(Context.WINDOW\_SERVICE);**

**}**

**//注释**

**mWindowManager = ((WindowManagerImpl)wm).createLocalWindowManager(this);**

**}**

**public WindowManagerImpl createLocalWindowManager(Window parentWindow) {**

**return new WindowManagerImpl(mContext, parentWindow);**

**}**

**注释**：这个方法很简单就是就是又一次创建了 WindowManagerImpl 对象，这时 **WindowManager 真正和Window关联起来**。

PhoneWindow.setContentView

Activity 将 setContentView 的操作交给了 PhoneWindow，接下来看下其实现过程：

**@Override**

**public void setContentView(int layoutResID) {**

**//注意：当主题属性等结晶化时，可在安装窗口装饰的过程中设置功能内容转换。**

**//在这种情况发生之前，不要检查功能。**

**if (mContentParent == null) {**

**//注释1**

**installDecor();**

**} else if (!hasFeature(FEATURE\_CONTENT\_TRANSITIONS)) {**

**mContentParent.removeAllViews();**

**}**

**if (hasFeature(FEATURE\_CONTENT\_TRANSITIONS)) {**

**final Scene newScene = Scene.getSceneForLayout(mContentParent, layoutResID,**

**getContext());**

**transitionTo(newScene);**

**} else {**

**//注释2**

**mLayoutInflater.inflate(layoutResID, mContentParent);**

**}**

**mContentParent.requestApplyInsets();**

**final Callback cb = getCallback();**

**if (cb != null && !isDestroyed()) {**

**cb.onContentChanged();**

**}**

**mContentParentExplicitlySet = true;**

**}**

PhoneWindow.installDecor

**private void installDecor() {**

**mForceDecorInstall = false;**

**//初始化mDecor**

**if (mDecor == null) {**

**mDecor = generateDecor(-1);**

**mDecor.setDescendantFocusability(ViewGroup.FOCUS\_AFTER\_DESCENDANTS);**

**mDecor.setIsRootNamespace(true);**

**if (!mInvalidatePanelMenuPosted && mInvalidatePanelMenuFeatures != 0) {**

**mDecor.postOnAnimation(mInvalidatePanelMenuRunnable);**

**}**

**} else {**

**mDecor.setWindow(this);**

**}**

**//初始化mContentParent**

**if (mContentParent == null) {**

**mContentParent = generateLayout(mDecor);**

**...**

**}**

**}**

**注释1**：调用 **installDecor 初始化 DecorView 和 mContentParent。**

**注释2**：**mLayoutInflater.inflate(layoutResID, mContentParent);** 从指定的xml资源展开新的视图层次结构。调用 setContentView 传入的布局添加到 mContentParent 中。

从中可以看出PhoneWindow 中默认有一个 DecorView（实际上是一个 FrameLayout），在 DecorView 中默认自带一个 mContentParent（ViewGroup）。我们自己实现的布局是被添加到 mContentParent 中的，因此经过 setContentView 之后，PhoneWindow 内部的 View 关系如下所示：

目前为止 PhoneWindow 中只是创建出了一个 DecorView，并在 DecorView 中填充了我们在 Activity 中传入的 layoutId 布局，可是 DecorView 还没有跟 Activity 建立任何联系，也没有被绘制到界面上显示。那 DecorView 是何时被绘制到屏幕上的呢？

Activity 执行到 onCreate 时并不可见，只有执行完 onResume 之后 Activity 中的内容才是屏幕可见状态。造成这种现象的原因就是，onCreate 阶段只是初始化了 Activity 需要显示的内容，而在 onResume 阶段(当界面要与用户进行交互时，会调用ActivityThread的handleResumeActivity方法)才会将 PhoneWindow 中的 DecorView 真正的绘制到屏幕上。

ActivityThread.handleResumeActivity

在 ActivityThread 的 handleResumeActivity 中，会调用 WindowManager 的 addView 方法将 DecorView 添加到 WMS(WindowManagerService) 上，如下所示：

**@Override**

**public void handleResumeActivity(IBinder token, boolean finalStateRequest, boolean isForward,**

**String reason) {**

**...**

**ViewManager wm = a.getWindowManager();**

**...**

**if (a.mVisibleFromClient) {**

**if (!a.mWindowAdded) {**

**a.mWindowAdded = true;**

**//注释**

**wm.addView(decor, l);**

**} else {**

**a.onWindowAttributesChanged(l);**

**}**

**}**

**...**

**}**

**注释**：WindowManger 的 addView 结果有两个：

DecorView 被渲染绘制到屏幕上显示； DecorView 可以接收屏幕触摸事件。

WindowManager.addView

ViewManager是一个接口，WindowManager是个接口同时又实现了ViewManager接口，而WindowManagerImpl又实现了WindowManager，这是只捋清楚这个方法的层级，清楚后放到一边继续查看WindowManagerImpl中的addview方法

WindowManagerImpl.addView

**@Override**

**public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params) {**

**applyDefaultToken(params);**

**mGlobal.addView(view, params, mContext.getDisplayNoVerify(), mParentWindow,**

**mContext.getUserId());**

**}**

WindowManagerImpl.addView也是一个空壳，它调用了 **WindowManagerGlobal** 的 **addView** 方法。

WindowManagerGlobal.addView

**public void addView(View view, ViewGroup.LayoutParams params,**

**Display display, Window parentWindow, int userId) {**

**...**

**ViewRootImpl root;**

**View panelParentView = null;**

**synchronized (mLock) {**

**...**

**root = new ViewRootImpl(view.getContext(), display);**

**//注释1**

**view.setLayoutParams(wparams);**

**mViews.add(view);**

**mRoots.add(root);**

**mParams.add(wparams);**

**//最后执行此操作，因为它会发出消息开始执行操作**

**try {**

**//注释2**

**root.setView(view, wparams, panelParentView, userId);**

**} catch (RuntimeException e) {**

**...**

**}**

**}**

**}**

**注释1**：WindowMangerGlobal 是一个单例 在 addView 方法中，创建了一个最关键的 ViewRootImpl 对象。

**注释2**：然通过 root.setView 方法将 view 添加到 WMS 中。

ViewRootImpl.setView

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

**int userId) {**

**synchronized (this) {**

**if (mView == null) {**

**mView = view;**

**...**

**int res; /\* = WindowManagerImpl.ADD\_OKAY; \*/**

**//注释1**

**requestLayout();**

**InputChannel inputChannel = null;**

**if ((mWindowAttributes.inputFeatures**

**& WindowManager.LayoutParams.INPUT\_FEATURE\_NO\_INPUT\_CHANNEL) == 0) {**

**inputChannel = new InputChannel();**

**}**

**mForceDecorViewVisibility = (mWindowAttributes.privateFlags**

**& PRIVATE\_FLAG\_FORCE\_DECOR\_VIEW\_VISIBILITY) != 0;**

**try {**

**...**

**//注释2**

**res = mWindowSession.addToDisplayAsUser(mWindow, mSeq, mWindowAttributes,**

**getHostVisibility(), mDisplay.getDisplayId(), userId, mTmpFrame,**

**mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,**

**mAttachInfo.mDisplayCutout, inputChannel,**

**mTempInsets, mTempControls);**

**setFrame(mTmpFrame);**

**} catch (RemoteException e) {**

**...**

**} finally {**

**if (restore) {**

**attrs.restore();**

**}**

**}**

**}**

**}**

**注释1**： requestLayout 是刷新布局的操作，调用此方法后 ViewRootImpl 所关联的 View 也执行 measure -> layout -> draw 操作，确保在 View 被添加到 Window 上显示到屏幕之前，已经完成测量和绘制操作。

**注释2**：调用 mWindowSession 的 addToDisplay 方法将 View 添加到 WMS 中。

mWindowSession哪里来的？他是new RootViewlmpl对象是传入进来的，调用WindowManagerGlobal.getWindowSession()生成的。

new ViewRootImpl(context,display);

**public ViewRootImpl(Context context, Display display) {**

**this(context, display, WindowManagerGlobal.getWindowSession(),false);**

**}**

WindowManagerGlobal.getWindowSession()

WindowSession 是 WindowManagerGlobal 中的单例对象，初始化代码如下：

**public static IWindowSession getWindowSession() {**

**synchronized (WindowManagerGlobal.class) {**

**if (sWindowSession == null) {**

**try {**

**InputMethodManager.ensureDefaultInstanceForDefaultDisplayIfNecessary();**

**IWindowManager windowManager = getWindowManagerService();**

**//注释**

**sWindowSession = windowManager.openSession(**

**new IWindowSessionCallback.Stub() {**

**@Override**

**public void onAnimatorScaleChanged(float scale) {**

**ValueAnimator.setDurationScale(scale);**

**}**

**});**

**} catch (RemoteException e) {**

**throw e.rethrowFromSystemServer();**

**}**

**}**

**return sWindowSession;**

**}**

**}**

**注释1**：sWindowSession 实际上是 IWindowSession 类型，是一个 Binder 类型，真正的实现类是 System 进程中的 Session。用 AIDL 获取 System 进程中 Session 的对象。

Session.addToDisplay

**@Override**

**public int addToDisplay(IWindow window, int seq, WindowManager.LayoutParams attrs,**

**int viewVisibility, int displayId, Rect outFrame, Rect outContentInsets,**

**Rect outStableInsets,**

**DisplayCutout.ParcelableWrapper outDisplayCutout, InputChannel outInputChannel,**

**InsetsState outInsetsState, InsetsSourceControl[] outActiveControls) {**

**return mService.addWindow(this, window, seq, attrs, viewVisibility, displayId, outFrame,**

**outContentInsets, outStableInsets, outDisplayCutout, outInputChannel,**

**outInsetsState, outActiveControls, UserHandle.getUserId(mUid));**

**}**

**return mService.addWindow(…);**

其中的 mService 就是 WMS。至此，Window 已经成功的被传递给了 WMS。剩下的工作就全部转移到系统进程中的 WMS 来完成最终的添加操作。

又回到Activity

addView 成功有一个标志就是能够接收触屏事件，通过对 setContentView 流程的分析，可以看出添加 View 的操作实质上是 PhoneWindow 在全盘操作，背后负责人是 WMS，反之 Activity 自始至终没什么参与感。但是我们也知道当触屏事件发生之后，Touch 事件首先是被传入到 Activity，然后才被下发到布局中的 ViewGroup 或者 View（Touch事件分发了解一下）。那么 Touch 事件是如何传递到 Activity 上的呢？

ViewRootImpl 中的 setView 方法中，除了调用 IWindowSession 执行跨进程添加 View 之外，还有一项重要的操作就是设置输入事件的处理：

ViewRootImpl.setView

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

**int userId) {**

**..**

**res = mWindowSession.addToDisplayAsUser(mWindow, mSeq, mWindowAttributes,**

**getHostVisibility(), mDisplay.getDisplayId(), userId, mTmpFrame,**

**mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,**

**mAttachInfo.mDisplayCutout, inputChannel,**

**mTempInsets, mTempControls);**

**// 注释1：设置输入管道。**

**CharSequence counterSuffix = attrs.getTitle();**

**mSyntheticInputStage = new SyntheticInputStage();**

**InputStage viewPostImeStage = new ViewPostImeInputStage(mSyntheticInputStage);**

**InputStage nativePostImeStage = new NativePostImeInputStage(viewPostImeStage,**

**"aq:native-post-ime:" + counterSuffix);**

**InputStage earlyPostImeStage = new EarlyPostImeInputStage(nativePostImeStage);**

**InputStage imeStage = new ImeInputStage(earlyPostImeStage,**

**"aq:ime:" + counterSuffix);**

**InputStage viewPreImeStage = new ViewPreImeInputStage(imeStage);**

**InputStage nativePreImeStage = new NativePreImeInputStage(viewPreImeStage,**

**"aq:native-pre-ime:" + counterSuffix);**

**...**

**}**

**注释1**：设置了一系列的输入通道。一个触屏事件的发生是由屏幕发起，然后经过驱动层一系列的优化计算通过 Socket 跨进程通知 Android Framework 层（实际上就是 WMS），最终屏幕的触摸事件会被发送到代码中的输入管道中。

这些输入管道实际上是一个链表结构，当某一个屏幕触摸事件到达其中的 ViewPostImeInputState 时，会经过 onProcess 来处理，如下所示：

ViewRootImpl.ViewPostImeInputStage

**final class ViewPostImeInputStage extends InputStage {**

**public ViewPostImeInputStage(InputStage next) {**

**super(next);**

**}**

**@Override**

**protected int onProcess(QueuedInputEvent q) {**

**if (q.mEvent instanceof KeyEvent) {**

**return processKeyEvent(q);**

**} else {**

**final int source = q.mEvent.getSource();**

**if ((source & InputDevice.SOURCE\_CLASS\_POINTER) != 0) {**

**return processPointerEvent(q);**

**} else if ((source & InputDevice.SOURCE\_CLASS\_TRACKBALL) != 0) {**

**return processTrackballEvent(q);**

**} else {**

**return processGenericMotionEvent(q);**

**}**

**}**

**}**

**private int processPointerEvent(QueuedInputEvent q) {**

**final MotionEvent event = (MotionEvent)q.mEvent;**

**...**

**boolean handled = mView.dispatchPointerEvent(event);**

**maybeUpdatePointerIcon(event);**

**maybeUpdateTooltip(event);**

**mAttachInfo.mHandlingPointerEvent = false;**

**...**

**return handled ? FINISH\_HANDLED : FORWARD;**

**}**

**}**

processPointerEvent 在 ViewPostImeInputStage 中别找错了。可以看到在 onProcess 中最终调用了一个 mView的dispatchPointerEvent 方法，mView 实际上就是 PhoneWindow 中的 DecorView，而 dispatchPointerEvent 是被 View.java 实现的，如下所示：

View.dispatchPointerEvent

**public final boolean dispatchPointerEvent(MotionEvent event) {**

**if (event.isTouchEvent()) {**

**return dispatchTouchEvent(event);**

**} else {**

**return dispatchGenericMotionEvent(event);**

**}**

**}**

DecorView.dispatchTouchEvent

**@Override**

**public boolean dispatchTouchEvent(MotionEvent ev) {**

**final Window.Callback cb = mWindow.getCallback();**

**return cb != null && !mWindow.isDestroyed() && mFeatureId < 0**

**? cb.dispatchTouchEvent(ev) : super.dispatchTouchEvent(ev);**

**}**

最好调用了 Window.Callback 中 cb.dispatchTouchEvent(ev) 方法，那这个 Callback 是不是 Activity 呢？

Activity.attach

**final void attach(Context context, ActivityThread aThread,**

**Instrumentation instr, IBinder token, int ident,**

**Application application, Intent intent, ActivityInfo info,**

**CharSequence title, Activity parent, String id,**

**NonConfigurationInstances lastNonConfigurationInstances,**

**Configuration config, String referrer, IVoiceInteractor voiceInteractor,**

**Window window, ActivityConfigCallback activityConfigCallback, IBinder assistToken) {**

**...**

**mWindow = new PhoneWindow(this, window, activityConfigCallback);**

**mWindow.setWindowControllerCallback(mWindowControllerCallback);**

**//this:Activity**

**mWindow.setCallback(this);**

**...**

**}**

Activity 将自身传递给了 PhoneWindow，再接着看 Activity的dispatchTouchEvent。

Activity.dispatchTouchEvent

**public boolean dispatchTouchEvent(MotionEvent ev) {**

**if (ev.getAction() == MotionEvent.ACTION\_DOWN) {**

**//该方法是用户交互，每当向Activity分派按键、触摸或轨迹球事件时调用。**

**//在这里仅用于ACTION\_DOWN的判断**

**onUserInteraction();**

**}**

**//返回true**

**if (getWindow().superDispatchTouchEvent(ev)) {**

**//Activity.dispatchTouchEvent()就返回true，则方法结束。**

**//该点击事件停止往下传递&事件传递过程结束**

**return true;**

**}**

**return onTouchEvent(ev);**

**}**

PhoneWindow.superDispatchTouchEvent

**@Override**

**public boolean superDispatchTouchEvent(MotionEvent event) {**

**return mDecor.superDispatchTouchEvent(event);**

**}**

DecorView.superDispatchTouchEvent

**public boolean superDispatchTouchEvent(MotionEvent event) {**

**return super.dispatchTouchEvent(event);**

**}**

Touch 事件在 Activity 中只是绕了一圈最后还是回到了 PhoneWindow 中的 DecorView 来处理。 剩下的就是从 DecorView 开始将事件层层传递给内部的子 View 中了

ViewGroup. dispatchTouchEvent

**@Override**

**public boolean dispatchTouchEvent(MotionEvent ev) {**

**}**

到这里就不做多的重复了。感兴趣的可以看看事件分发机制。

小结

通过 setContentView 的流程，分析了 Activity、Window、View 之间的关系。整个过程 Activity 表面上参与度比较低，Activity持有Window的对象，View在Window上的增删等操作又是通过WindowManager来管理的，而WindowManager又是通过Binder机制获取到的WMS的映射，WMS把View真正显示到屏幕上。

三者关系：

1. 一个 Activity 中有一个 window，也就是 PhoneWindow 对象，在Activity中调用attach，创建了一个PhoneWindow。
2. 在 PhoneWindow 中有一个 DecorView，Activity在 调用setContentView 中会将 layout 填充到此 DecorView 中。
3. 一个应用进程中只有一个 WindowManagerGlobal 对象，因为在 ViewRootImpl 中它是 static 静态类型。
4. 每一个 PhoneWindow 对应一个 ViewRootImple 对象。
5. WindowMangerGlobal 通过调用 ViewRootImpl 的 setView 方法，完成 window 的添加过程。
6. 调用ViewGroup的removeAllView()，先将所有的view移除掉
7. ViewRootImpl 的 setView 方法中主要完成两件事情：View 渲染（requestLayout）以及接收触屏事件。