Android Window 机制

Chunyu J 郭霖 2022-10-31 08:00 发表于江苏

/ 今日科技快讯 /

近日，马斯克已完成对社交平台公司推特的收购。舆论担忧马斯克对推特的接管意味着“仇恨言论时代”的到来，更有分析显示推特上的“污言秽语和种族主义言论”已如潮水般涌现。

据报道，马斯克自称“言论自由绝对主义者”，而推特此前一直试图“公平地”管理其平台上的内容。随着马斯克接管推特，一些保守派政客、极端主义者和“阴谋论推销者”看到了契机。

/ 作者简介 /

大家周一好，新的一周继续努力！

本篇文章来自八千里路山与海的投稿，文章主要分享了Window 机制的相关内容，相信会对大家有所帮助！同时也感谢作者贡献的精彩文章。

/ 基础概念 /

Window

在 Android 系统中，屏幕的抽象是 DisplayContent ，在屏幕的抽象上，通过不同的窗口，展示不同的应用程序页面和一些其他UI 组件（例如 Dialog、状态栏等）。Window 通过在 Z 轴叠放，从而实现了复杂的交互逻辑。

Window 是一个处理顶级窗口外观和行为策略的抽象基类，它的具体实现是 PhoneWindow 类，PhoneWindow 对 View 进行管理。

如用户所见，Android 应用程序中的页面以 Activity 组件的形式呈现，而在 Activity 初始化时，就会创建 PhoneWindow ，这样 Activity 组件就具备了窗口作为容器来展示各种各样的 View 与用户进行交互。

Window 是顶级窗口外观和行为策略的抽象基类。它的实例应用作添加到 Window Manager 的顶级视图。它提供了标准的 UI 策略，例如背景、标题区域、默认按键处理等。此抽象类的唯一现有实现是 android.view.PhoneWindow，您应该在需要 Window 时对其进行实例化。

提前需要了解的是 Window 的管理需要与系统的服务进行通信 ，这是一套 IPC 机制，CS 架构，服务端是 WindowManagerService，客户端是 WindowManager 。

PhoneWindow 的创建

在 Activity 启动阶段，最终调用的 ActivityThread 的 performLaunchActivity 方法中，调用到了 Activity 的 attach 方法，PhoneWindow 就是在此时创建的：

```
@UnsupportedAppUsage(maxTargetSdk = Build.VERSION_CODES.R, trackingBug = 170729553)
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,
        IBinder shareableActivityToken) {
    attachBaseContext(context);

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

    mWindow = new PhoneWindow(this, window, activityConfigCallback);
    mWindow.setWindowControllerCallback(mWindowControllerCallback);
    mWindow.setCallback(this);
    mWindow.setOnWindowDismissedCallback(this);
    mWindow.getLayoutInflater().setPrivateFactory(this);
    if (info.softInputMode != WindowManager.LayoutParams.SOFT_INPUT_STATE_UNSPECIFIED) {
        mWindow.setSoftInputMode(info.softInputMode);
    }
    if (info.uiOptions != 0) {
        mWindow.setUiOptions(info.uiOptions);
    }
    mUiThread = Thread.currentThread();
```


并且它还设置了一个 WindowManager：

```
mWindow.setWindowManager(
        (WindowManager)context.getSystemService(Context.WINDOW_SERVICE),
        mToken, mComponent.flattenToString(),
        (info.flags & ActivityInfo.FLAG_HARDWARE_ACCELERATED) != 0);
if (mParent != null) {
    mWindow.setContainer(mParent.getWindow());
}
mWindowManager = mWindow.getWindowManager();
mCurrentConfig = config;

mWindow.setColorMode(info.colorMode);
mWindow.setPreferMinimalPostProcessing(
        (info.flags & ActivityInfo.FLAG_PREFER_MINIMAL_POST_PROCESSING) != 0);
```


这个方法在 Window 中实现，如果传入的 WindowManager 为 null，内部获取了一个 WindowManagerImpl 来作为 WindowManager。WindowManager 是系统提供的窗口管理工具，与 Android 的窗口和 UI 展示息息相关，后续会详细解读。

WindowType 和显示次序

Android 中的关于窗口的概念有很多，比如应用程序页面、系统错误弹窗、输入法窗口、PopupWindow 、 Toast 、 Dialog 等等。总的来说分为三大类：

1. Application Window：应用程序窗口，Activity 就是一个典型的应用程序窗口，它的 Type 枚举范围在 1 - 99 。这个数值会涉及窗口的层级。
2. System Window：系统窗口，Toast 、输入法窗口、系统音量窗口、系统错误窗口都属于系统窗口。范围在 2000 - 2999 。
3. Sub Window：子窗口，不能独立存在，需要依附于其他窗口，典型的例子是 PopupWindow 。它的 Type 范围是 1000 -1999 。

窗口在代码中通过 Window 接口表示，而窗口类型，就是通过 Window 的 type 属性表示，它被定义在 WindowManager 的内部类 LayoutParams 中：

```
@WindowType
public int type;

@IntDef(prefix = "TYPE_", value = {
    TYPE_BASE_APPLICATION,
    TYPE_APPLICATION,
    TYPE_APPLICATION_STARTING,
   //....
})
@Retention(RetentionPolicy.SOURCE)
public @interface WindowType {}
```

窗口的显示次序

当一个进程向 WMS 申请一个窗口时，WMS 会为窗口确定显示次序。为了方便次序的管理，手机可以虚拟地用 X 、 Y 、 Z 轴来表示， Z 轴垂直于屏幕，由 Z 轴来确定窗口的显示次序。一般情况下，WindowType 值越大，显示层级越高，即越接近用户，不容易被遮挡。当然也有比较复杂的规则，这里只是说明基础层级规则。

Window Flag

同样在 WindowManager 的内部类 LayoutParams 中定义。用来控制 Window 的显示和行为。

```
/** Window flag: as long as this window is visible to the user, allow
     * the lock screen to activate while the screen is on.
     * This can be used independently, or in combination with
     * {@link #FLAG_KEEP_SCREEN_ON} and/or {@link #FLAG_SHOW_WHEN_LOCKED} */
    public static final int FLAG_ALLOW_LOCK_WHILE_SCREEN_ON    = 0x00000001;

    /** Window flag: everything behind this window will be dimmed.
     * Use {@link #dimAmount} to control the amount of dim. */
    public static final int FLAG_DIM_BEHIND     = 0x00000002;

    /** Window flag: enable blur behind for this window. */
    public static final int FLAG_BLUR_BEHIND     = 0x00000004;

 // ...
```

例如常见的例如 FLAG_BOT_TOUCHABLLE ，表示窗口不在接收任何触摸事件。

为 Window 添加 Flag 可以通过 addFlags 方法和 setFlags 方法：

```
window.addFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN);
// or
window.setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN);
```

也可以通过 WindowManager.LayoutParams 去设置：

```
val lp = WindowManager.LayoutParams()
lp.flags = WindowManager.LayoutParams.FLAG_FULLSCREEN
val tv = TextView(context)
windowManager.addView(tv, lp)
```

Window SoftInputMode

SoftInputMode 是用来控制软键盘相关的。窗口叠加是很常见的场景，尤其是在输入的时候，默认情况下输入发会顶在输入框下方，输入框下方的内容遮盖住，这种场景在例如登录界面是非常不好的体验。为了能够自由的控制软键盘窗口，WindowManager 的 LayoutParams 提供了软键盘相关的模式：

int	SOFT_INPUT_ADJUST_NOTHING Adjustment option for softInputMode: set to have a window not adjust for a shown input method.
int	SOFT_INPUT_ADJUST_PAN Adjustment option for softInputMode: set to have a window pan when an input method is shown, so it doesn't need to deal with resizing but just panned by the framework to ensure the current input focus is visible.
int	SOFT_INPUT_ADJUST_RESIZE *This constant was deprecated in API level 30. Call* Window#setDecorFitsSystemWindows(boolean) *with* false *and install an* OnApplyWindowInsetsListener *on your root content view that fits insets of type* Type#ime().
int	SOFT_INPUT_ADJUST_UNSPECIFIED Adjustment option for softInputMode: nothing specified.
int	SOFT_INPUT_IS_FORWARD_NAVIGATION Bit for softInputMode: set when the user has navigated forward to the window.
int	SOFT_INPUT_MASK_ADJUST Mask for softInputMode of the bits that determine how the window should be adjusted to accommodate the soft input window.

截图中仅为部分属性，这些内容可以在 AndroidManifest.xml 中的 Activity 的属性：android:windowSoftInputMode 设置。当然也可以通过代码去进行设置：

```
window.softInputMode = WindowManager.LayoutParams.SOFT_INPUT_MASK_ADJUST
```

WindowManager

Window 由窗口管理器 WindowManager 来进行管理，它通过与 WindowManagerService 进行通信，作为客户端的工具类来管理 Window ：

```
@SystemService(Context.WINDOW_SERVICE)
public interface WindowManager extends ViewManager
```

ViewManager 中定义了对 View 管理，他有三个方法，分别代表了添加、更新和删除：

- 添加：addView
- 更新：updateViewLayout
- 删除：removeView

WindowManager 因为继承关系，所以具备了管理 View 的能力，事实上，WindowManager 实际管理的是 Window 中的 View ，并通过 Binder 机制来与 WindowManagerService （WMS）进行 IPC 通信，通过 WMS 来管理 Window 。

WindowManager 的实现是 WindowManagerImpl ，但它的内部并没有实现逻辑，而是通过桥接模式，将功能委托给 WindowManagerGlobal 来实现。

WindowManagerGlobal 是一个单例，一个进程只有一个 WindowManagerGlobal 实例。负责 WindowManager 的具体功能实现。WindowManagerGlobal 中包含了 View 的添加、删除和更新的核心代码。WindowManagerGlobal 中处理 View 的添加、更新和删除，与 ViewRootImpl 密切相关。

ViewRootImpl

ViewRootImpl 代表的是 Android 视图层次结构的顶部，是 View 和 WindowManager 之间的桥梁。Android 的视图层次结构是树结构，ViewRootImpl 实际上就是树的根节点的抽象。WindowManager 通过根节点对象，来更新整个树结构上的 View 节点的内容。

一个 Window 会有一个根节点，这一点我们可以通过 Android 的视图结构亦或是无障碍服务节点的树结构来证实。所以，在客户端 WindowManager 这一侧，实际上是在更新树结构；而在服务端 WMS 中的逻辑，则是管理窗口与屏幕之间的联系。

WindowManagerService

WindowManagerService 是系统中重要的服务，WindowManagerService 负责管理窗口的启动、添加、删除、大小和层级等；它还持有 InputManagerService 的引用来处理窗口的触摸事件；亦或是处理窗口之间切换动画。

注意：本文只讲解 WMS 处理窗口相关的内容。

WMS 创建流程

简述创建流程，有兴趣可自行参考源码。

WMS 是在 SystemServer 进行中创建的， SystemServer 执行到 startOtherServices 方法中启动 WMS 。在 startOtherServices 方法中创建流程是：

- 初始化 Watchdog ，这是一个监控系统关键服务的运行状态的监控器。
- 创建 InputManagerService 。
- 开始创建 WMS， 运行 WMS 的 main 方法，将 IMS 传递给 WMS （WMS 持有 IMS 引用）。
  - 通过 DisplayManager 获取 Display 数组。
  - 将 Display 封装成 DisplayContent ，用来作为屏幕的抽象。
  - 获取 AMS 的实例（WMS 持有 AMS 的引用）。
  - 创建 WindowAnimator 。
  - 初始化 WindowManagerPolicy 。
  - 将 WMS 添加到 Watchdog 中。
- WMS、IMS 注册到 ServiceManager 中。
- 初始化屏幕信息，调用 WMS 的 displayReady 方法。
- 通过调研 WMS 的 systemReady 方法通知 WMS 初始化完成。

Session

Session 是用于其他应用程序和 WMS 通信之间的通道，每个应用程序都会有一个 Session ，在 WMS 通过 mSessions 属性保存，它的类型是 ArraySet<Session> 。

WindowContainer

WindowContainer 定义了一组可以直接或通过其子类以层次结构形式保存 Window 的类的常用功能。

WindowState

WindowState 代表 WMS 管理的一个窗口，继承自 WindowContainer<WindowState> ，并实现了一些其他接口。

WindowToken

WindowToken 表示 WMS 中一组 Window 的容器。通常这是表示一个应用程序中用于显示一组窗口 Activity ； 对于嵌套的窗口组合来说，则需要为其父窗口创建一个 WindowToken 来管理子窗口。

/ Window 添加过程 /

WindowManager 添加过程

窗口的添加过程入口是 WindowManager 的 addView 方法，实际逻辑在 WindowManagerGlobal 中：

```
public void addView(View view, ViewGroup.LayoutParams params, Display display, Window pare
    // ...
    ViewRootImpl root;
    View panelParentView = null;

    synchronized (mLock) {
      // ...
      if (windowlessSession == null) {
        root = new ViewRootImpl(view.getContext(), display);
      } else {
        root = new ViewRootImpl(view.getContext(), display,
            windowlessSession);
      }
      view.setLayoutParams(wparams);
      mViews.add(view);
      mRoots.add(root);
      mParams.add(wparams);
```

```
        try {
            root.setView(view, wparams, panelParentView, userId);
        } catch (RuntimeException e) {
            // BadTokenException or InvalidDisplayException, clean up.
            if (index >= 0) {
                removeViewLocked(index, true);
            }
            throw e;
        }
    }
}
```

这里只提取关键的逻辑：

1. 根据 windowlessSession （IWindowSession）是否可空创建 ViewRootImpl
2. 给 view 设置布局参数
3. 将 View 、ViewRootImpl 、LayoutParams 分别保存到各自的数组中
4. 最终调用 ViewRootImpl 的 setView 方法

处理逻辑从 WindowManagerGlobal 来到了 ViewRootImpl 中，继续跟进 setView 方法：

```
public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView, int
    synchronized (this) {
        if (mView == null) {
            mView = view;
            // ...
            int res; /* = WindowManagerImpl.ADD_OKAY; */

            // 在添加到窗口管理器之前安排第一次布局，以确保我们在从系统接收任何其他事件之前进行布局。
            requestLayout();
            // ...
            try {
                // ...
                res = mWindowSession.addToDisplayAsUser(mWindow, mWindowAttributes,
                        getHostVisibility(), mDisplay.getDisplayId(), userId,
                        mInsetsController.getRequestedVisibilities(), inputChannel, mTempInset
                        mTempControls);
                // ... 处理窗口转换
            } catch (RemoteException e) {
                // ...
            } finally {
                // ...
            }
```

```
            // WindowLayout 用于计算窗口框架尺寸
            mWindowLayout.computeFrames(mWindowAttributes, state,
                 displayCutoutSafe, winConfig.getBounds(), winConfig.getWindowingMode(),
                 UNSPECIFIED_LENGTH, UNSPECIFIED_LENGTH,
                 mInsetsController.getRequestedVisibilities(),
                 getAttachedWindowFrame(), 1f /* compactScale */, mTmpFrames);
            setFrame(mTmpFrames.frame);
            // ...
        }
    }
}
```

这部分代码只截取关键逻辑：

1. 在真正添加到 WMS 之前，安排了一次布局，以确保我们在从系统接收任何其他事件之前已经进行布局
2. 通过 mWindowSession.addToDisplayAsUser 调用到 WMS 中的逻辑
3. 最后拿到 WMS 添加窗口的处理结果后，重新计算并更新窗口的 Frame

在第二步中， mWindowSession 的类型是 IWindowSession.aidl ，它的实现是 Session ：

```
class Session extends IWindowSession.Stub implements IBinder.DeathRecipient {
  final WindowManagerService mService;
  // ...
  @Override
   public int addToDisplayAsUser(IWindow window, WindowManager.LayoutParams attrs,
        int viewVisibility, int displayId, int userId, InsetsVisibilities requestedVisibil
        InputChannel outInputChannel, InsetsState outInsetsState,
        InsetsSourceControl[] outActiveControls) {
     return mService.addWindow(this, window, attrs, viewVisibility, displayId, userId,
         requestedVisibilities, outInputChannel, outInsetsState, outActiveControls);
   }
  // ...
}
```

Session 本身就是 Binder ，通过 Session ，跨进程调用到了 WMS 的 addWindow 方法，此时流程进入到系统服务端。这个流程在后续 WMS 的流程中分析，这里可以看出 addToDisplayAsUser 返回了 WMS 的处理结果，通过 int 形式返回。

根据 WMS 的处理结果 res ，WindowManager 中，继续计算了 Frame ，经过计算后通过 ViewRootImpl 的 setFrame(Rect) 方法进行设置：

```
private void setFrame(Rect frame) {
   mWinFrame.set(frame); // mWinFrame: Rect
   // ...
   mInsetsController.onFrameChanged(mOverrideInsetsFrame != null ?
        mOverrideInsetsFrame : frame);
}
```

mInsetsController 的类型是 InsetsController ，它是 WindowInsetsController 在客户端的实现，用来控制生产嵌入窗口的接口。也就是说这里会处理嵌入窗口结构的尺寸更新：

```
  // in InsetsController.java
  @VisibleForTesting
   public void onFrameChanged(Rect frame) {
      if (mFrame.equals(frame)) {
         return;
      }
      mHost.notifyInsetsChanged();
      mFrame.set(frame);
   }
```

这里调用 mHost 通知更新，mHost 是 InsetsController.Host 接口对象，这个接口有两个实现，在 ViewRootImpl 的创建是：

```
mInsetsController = new InsetsController(new ViewRootInsetsControllerHost(this));
```

ViewRootInsetsControllerHost 中的 notifyInsetsChanged 方法：

```
  @Override
   public void notifyInsetsChanged() {
      mViewRoot.notifyInsetsChanged();
   }
```

调回到了 ViewRootImpl 自己的方法：

```
   void notifyInsetsChanged() {
```

```
        mApplyInsetsRequested = true;
        requestLayout();

        // See comment for View.sForceLayoutWhenInsetsChanged
        if (View.sForceLayoutWhenInsetsChanged && mView != null && (mWindowAttributes.softIn
            forceLayout(mView);
        }

        // If this changes during traversal, no need to schedule another one as it will dispatch
        if (!mIsInTraversal) {
            scheduleTraversals();
        }
    }
```

在这里请求进行布局：

```
    @Override
    public void requestLayout() {
        if (!mHandlingLayoutInLayoutRequest) {
            checkThread();
            mLayoutRequested = true;
            scheduleTraversals();
        }
    }
```

检查线程，然后执行 scheduleTraversals ：

```
    void scheduleTraversals() {
        if (!mTraversalScheduled) {
            mTraversalScheduled = true;
            mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();
            mChoreographer.postCallback(
                    Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null);
            notifyRendererOfFramePending();
            pokeDrawLockIfNeeded();
        }
    }
```

scheduleTraversals 方法中通过 mChoreographer 对象发布一个回调来运行下一帧。Choreographer 是用来协调动画，输入和绘图的时间的类。执行的 Runnable 类型是 TraversalRunnable ：

```
final class TraversalRunnable implements Runnable {
    @Override
    public void run() {
        doTraversal();
    }
}
final TraversalRunnable mTraversalRunnable = new TraversalRunnable();
```

doTraversal 方法中，触发了 View 的绘制流程的核心方法 performTraversals ：

```
void doTraversal() {
    if (mTraversalScheduled) {
        mTraversalScheduled = false;
        mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);
        performTraversals();
    }
}
```

performTraversals 方法使得窗口中的视图树开始 View 的工作流程：1. relayoutWindow，内部调用 IWindowSession 的 relayout 方法来更新 Window视图。2. performMeasure，内部调用 View 的 measure 。3. performLayout，内部调用 View 的 layout 。4. performDraw，内部调用 View 的 draw 。

这样就完成了对 Window 中 UI 的刷新。

WindowManagerService 添加过程

在 ViewRootImpl 的 setView 方法中，通过 Session 调用到了 WMS 的 addWindow 方法，addWindow 中逻辑十分复杂：

1. 通过 WindowManagerPolicy 的 checkAddPermission 检查添加权限
2. 通过 displayId 获取所添加到的屏幕抽象 DisplayContent 对象
3. 根据 Window 是否是子窗口，创建 WindowToken
4. 如果上一步创建 token 失败，重新根据是否存在父窗口创建 WindowToken
    a. 子窗口直接取父窗口的 WindowToken
    b. 新的窗口直接构造一个新的 WindowToken

5. 根据根窗口的 type 来处理不同的情况（Toast、键盘、无障碍浮层等不同类型返回不同的结果）
6. 创建 WindowState
7. 检查客户端状态，判断是否可以将 Window 添加到系统中
8. 以 session 为 key ， windowState 为 value ， 存入到 mWindowMap 中：

```
/** Mapping from an IWindow IBinder to the server's Window object. */
  final HashMap<IBinder, WindowState> mWindowMap = new HashMap<>();
```

9.将 WindowState 添加到相应的 WindowToken 中

a. 若 WindowState 代表一个子窗口，直接 return

b. 若仍没有 SurfaceControl ， 为该 token 创建 Surface

c. 更新 Layer

d. 根据 Z 轴排序顺序将 WindowState 所代表的 Window 添加到合适的位置，此数据结构保存在 WindowToken 的父类 WindowContainer 的 mChildren 中：

```
protected final WindowList<E> mChildren = new WindowList<E>();

class WindowList<E> extends ArrayList<E> {
   void addFirst(E e) {
      add(0, e);
   }

   E peekLast() {
      return size() > 0 ? get(size() - 1) : null;
   }

   E peekFirst() {
      return size() > 0 ? get(0) : null;
   }
}
```

10.检查是否需要转换动画

11.更新窗口焦点

12.最后返回执行结果

在 ViewRootImpl 中的 setView 方法拿到结果后，开始重新计算尺寸，并重新绘制。

/ Window 更新过程 /

Window 的更新过程与 Window 的添加过程类似，通过 updateViewLayout 方法，它的实现也是在 WindowManagerImpl 中，同样的，实际调用的是 WindowManagerGlobal 对应的方法 updateViewLayout：

```
public void updateViewLayout(View view, ViewGroup.LayoutParams params) {
    if (view == null) {
        throw new IllegalArgumentException("view must not be null");
    }
    if (!(params instanceof WindowManager.LayoutParams)) {
        throw new IllegalArgumentException("Params must be WindowManager.LayoutParams");
    }

    final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams)params;

    view.setLayoutParams(wparams);

    synchronized (mLock) {
        int index = findViewLocked(view, true);
        ViewRootImpl root = mRoots.get(index);
        mParams.remove(index);
        mParams.add(index, wparams);
        root.setLayoutParams(wparams, false);
    }
}
```

更新过程没有涉及 WMS，直接给参数 view 更新 LayoutParams，然后同步更新了 ViewRootImpl 维护的 root / view / params 数组中的值。

在 ViewRootImpl 的 setLayoutParams 中，会调用 requestLayout 重新绘制 UI 。

/ Window 删除过程 /

Window 的删除过程通过 WindowManager 的 removeView 方法进行的，实际逻辑在 WindowManagerGlobal 的实现中：

```
public void removeView(View view, boolean immediate) {
    if (view == null) {
        throw new IllegalArgumentException("view must not be null");
```

```
        }

        synchronized (mLock) {
            int index = findViewLocked(view, true); // 1
            View curView = mRoots.get(index).getView(); // 2
            removeViewLocked(index, immediate); // 3
            if (curView == view) {
                return;
            }

            throw new IllegalStateException("Calling with view " + view
                    + " but the ViewAncestor is attached to " + curView);
        }
    }
```

注释 1 处首先通过 view 取查找 View 所在的索引。注释 2 从 mRoots 中根据索引找出 ViewRootImpl，通过 ViewRootImpl 找到它的 View，最后在注释 3 处调用 removeViewLocked 方法进行删除：

```
// WindowManagerGlobal.removeViewLocked
  private void removeViewLocked(int index, boolean immediate) {
        ViewRootImpl root = mRoots.get(index);
        View view = root.getView();

        if (root != null) {
            root.getImeFocusController().onWindowDismissed(); // 1
        }
        boolean deferred = root.die(immediate); // 2
        if (view != null) {
            view.assignParent(null);
            if (deferred) {
                mDyingViews.add(view);
            }
        }
    }
```

在这个方法中，首先也是获取 ViewRootImpl 和 View 对象，如果 ViewRootImpl 存在，通过注释 1 处的调用去清除输入焦点，结束处理输入法相关的逻辑。然后在注释 2 处调用 ViewRootImpl 对象的 die 方法：

```
// ViewRootImpl.die
boolean die(boolean immediate) {
    if (immediate && !mIsInTraversal) {
        doDie(); // 1
```

```
            return false;
        }

        if (!mIsDrawing) {
            destroyHardwareRenderer();
        } else {
            Log.e(mTag, "Attempting to destroy the window while drawing!\n  window=" + this + ", ti
        }
        mHandler.sendEmptyMessage(MSG_DIE);
        return true;
    }
```

die 方法中，在需要立即执行且当前没有进行 Traversal 的情况下执行 doDie 方法：

```
void doDie() {
    checkThread(); // 1
    synchronized (this) {
        // 防止多次执行 doDie
        if (mRemoved) {
            return;
        }
        mRemoved = true;

        if (mAdded) {
            dispatchDetachedFromWindow(); // 2
        }
        // 子 View 且不是首次添加的情况下，执行这些逻辑
        if (mAdded && !mFirst) {
            destroyHardwareRenderer();
            if (mView != null) {
                int viewVisibility = mView.getVisibility();
                boolean viewVisibilityChanged = mViewVisibility != viewVisibility;
                if (mWindowAttributesChanged || viewVisibilityChanged) {
                    // If layout params have been changed, first give them to the window manager to
                    try {
                        if ((relayoutWindow(mWindowAttributes, viewVisibility, false)
                                & WindowManagerGlobal.RELAYOUT_RES_FIRST_TIME) != 0) {
                            mWindowSession.finishDrawing(mWindow, null /* postDrawTransaction */);
                        }
                    } catch (RemoteException e) {
                    }
                }
                destroySurface();
            }
        }
        mAdded = false;
    }
```

```
    WindowManagerGlobal.getInstance().doRemoveView(this); // 3
}
```

在 doDie 方法中，首先进行的就是线程检查，然后确保同步的情况下，如果存在子 View ，调用注释 2 处的 dispatchDetachedFromWindow 方法；然后处理子 View 且不是首次添加的情况，最后调用注释 3 处的 WindowManagerGlobal 的 doRemoveView 方法。

首先来看 dispatchDetachedFromWindow ：

```
void dispatchDetachedFromWindow() {
    // ...

    // 视图树分发 dispatchDetachedFromWindow
    if (mView != null && mView.mAttachInfo != null) {
        mAttachInfo.mTreeObserver.dispatchOnWindowAttachedChange(false);
        mView.dispatchDetachedFromWindow();
    }
    // 销毁硬件渲染
    destroyHardwareRenderer();
    // 清除无障碍焦点
    setAccessibilityFocus(null, null);
    // 取消动画
    mInsetsController.cancelExistingAnimations();
    // 释放资源
    mView.assignParent(null);
    mView = null;
    mAttachInfo.mRootView = null;
    // 销毁 Surface
    destroySurface(); // 1

    // ...

    try {
        mWindowSession.remove(mWindow); // 2
    } catch (RemoteException e) {
    }

    // ...

    // 删除待执行的 Traversals
    unscheduleTraversals();
}
```

dispatchDetachedFromWindow 方法中，开始向视图树中的 View 分发 DetachedFromWindow，然后释放一些资源，在注释 1 处释放 Surface，注释 2 处通过 Session IPC 调用了 WMS 移除 Window：

```
// Session.remove
@Override
 public void remove(IWindow window) {
    mService.removeWindow(this, window);
 }
```

此时来到了服务端 WMS 的 removeWindow 方法中：

```
// WMS .removeWindow
 void removeWindow(Session session, IWindow client) {
     synchronized (mGlobalLock) {
        WindowState win = windowForClientLocked(session, client, false);
        if (win != null) {
           win.removeIfPossible();
           return;
        }

        // Remove embedded window map if the token belongs to an embedded window
        mEmbeddedWindowController.remove(client);
     }
  }
```

通过 windowForClientLocked 方法获取到了待移除 Window 的描述对象 WindowState，调用它的 removeIfPossible 方法：

```
// WindowState.removeIfPossible
@Override
 void removeIfPossible() {
    super.removeIfPossible();
    removeIfPossible(false /*keepVisibleDeadWindow*/);
 }
```

内部调用了同名不同参的方法：

```
// WindowState.removeIfPossible
private void removeIfPossible(boolean keepVisibleDeadWindow) {
   try {
      // ... 条件判断过滤，满足条件会直接 return 延迟删除操作
```

```
            removeImmediately(); // 1
            // 删除一个可见窗口将影响计算方向，如果需要，更新方向，
            if (wasVisible) {
                final DisplayContent displayContent = getDisplayContent();
                if (displayContent.updateOrientation()) {
                    displayContent.sendNewConfiguration();
                }
            }
            // 2
            mWmService.updateFocusedWindowLocked(isFocused()
                        ? UPDATE_FOCUS_REMOVING_FOCUS
                        : UPDATE_FOCUS_NORMAL,
                    true /*updateInputWindows*/);
        } finally {
            Binder.restoreCallingIdentity(origId);
        }
    }
```

在这个方法中，首先进行一些条件过滤，然后调用注释 1 处的 removeImmediately 执行删除操作，然后计算方向，注释 2 处更新 WMS 的 Window 焦点。跟进 removeImmediately 方法：

```
@Override
void removeImmediately() {
    // 确保同一时刻只执行一次 removeImmediately
    if (mRemoved) {
        return;
    }
    mRemoved = true;

    // ...

    mSession.windowRemovedLocked(); // 1
    try {
        mClient.asBinder().unlinkToDeath(mDeathRecipient, 0); // 2 解除与客户端的连接
    } catch (RuntimeException e) {
        // Ignore if it has already been removed (usually because
        // we are doing this as part of processing a death note.)
    }

    mWmService.postWindowRemoveCleanupLocked(this); // 3 执行一些集中清理的工作
}
```

在 removeImmediately 方法中：

- 注释 1 处处理 Session 相关的逻辑，从 mSessions 中删除了当前 mSession ，并清除 mSession 对应的 SurfaceSession 资源，（SurfaceSession 是 SurfaceFlinger 的一个连接，通过这个连接可以创建多个 Surface 并渲染到屏幕上）；
- 注释 2 处解除与客户端的连接；
- 注释 3 处集中执行了一些清理工作。

/ 总结 /

Window 的管理是基于 C/S 架构，依赖系统服务。服务端是 WMS ，客户端是 WindowManager 。

Window 的更新过程不涉及 WMS ，而添加和删除需要服务端与客户端一同合作，WindowManager 来请求执行，并处理 UI 渲染刷新，WMS 则是负责管理 Window 与系统其他能力结合。

Window 的概念不同于 View ，尽管看起来十分相似，但它们是两种概念。

- Window 代表一套视图树的容器，而 View 是视图树中的节点。
- ViewRootImpl 是视图树对象。
- WindowState 用来描述一个 Window。
- WindowToken 用来表示一组 Window 的容器，可以代表 Activity 和嵌套窗口的父容器。

另外，不同类型的 Window 添加、删除的逻辑略有不同，本文并没有介绍它们的差异性，感兴趣的读者可以自行查看源码。

推荐阅读：

我的新书，《第一行代码 第3版》已出版！

Kotlin Flow响应式编程，基础知识入门

Android View动画主流程全解析

欢迎关注我的公众号

学习技术或投稿

▼

长按上图，识别图中二维码即可关注

阅读原文

喜欢此内容的人还喜欢

这些不知道，别说你熟悉 Spring

CodeFox

某局VoLTE异常掉话问题分析

中兴文档

MAUI发布APK初体验

鹏祥