# TASK

<http://www.codexiu.cn/android/blog/36541/>

首页 / Android/ 旋屏事件上报流程分析

旋屏事件上报流程分析

rk屏幕旋转动画：com.wm/screenRotationAnimation.java

Toast太慢了，看DisplayContent.canAddRoastWindForUId

显示原理

两路显示窗口呢

# TASK

[Android 7.1.2 (Android N) Android WindowManager](http://zhoujinjian.cc/2018/03/01/Android-7-1-2-Android-N-Android-WindowManagerService-%E7%AA%97%E5%8F%A3%E7%AE%A1%E7%90%86%E6%9C%8D%E5%8A%A1%E5%88%86%E6%9E%90-i-wonder/index.html)

[《深入理解Android 卷III》第四章 深入理解WindowManagerService](https://blog.csdn.net/innost/article/details/47660193)

# 基本概念

Android系统中的窗口是屏幕上的一块用于绘制各种UI元素并可以响应应用户输入的一个矩形区域。从原理上来讲，窗口的概念是独自占有一个Surface实例的显示区域。例如Dialog、Activity的界面、壁纸、状态栏以及Toast等都是窗口。

WMS怎么来的？Surface：是一块画布，应用可以随心所欲地通过Canvas或者OpenGL在其上作画。然后通过SurfaceFlinger将多块Surface的内容按照特定的顺序（Z-order）进行混合并输出到FrameBuffer，从而将Android“漂亮的脸蛋”显示给用户。

既然每个窗口都有一块Surface供自己涂鸦，必然需要一个角色对所有窗口的Surface进行协调管理。于是，WMS便应运而生。WMS为所有窗口分配Surface，掌管Surface的显示顺序（Z-order）以及位置尺寸，控制窗口动画，并且还是输入系统的一重要的中转站。

本章将深入分析WMS的两个基础子系统的工作原理：

· 布局系统（Layout System），计算与管理窗口的位置、层次。

· 动画系统（Animation System），根据布局系统计算的窗口位置与层次渲染窗口动画。

## 源码清单

Ghj

**/frameworks/base/services/core/java/com/android/server/am/**

* ActivityStack.java
* ActivityManagerService.java
* ActivityStackSupervisor.java
* ActivityStarter.java
* ActivityRecord.java

**/frameworks/base/core/java/android/view/**

* WindowManagerImpl.java
* ViewManager.java
* WindowManagerGlobal.java
* ViewRootImpl.java
* Choreographer.java
* IWindowSession.aidl
* DisplayEventReceiver.java
* SurfaceControl.java
* Surface.java
* SurfaceSession.java

**/frameworks/base/services/core/java/com/android/server/wm/**

* WindowManagerService.java
* AppWindowAnimator.java
* AppTransition.java
* AppWindowToken.java
* Session.java
* WindowState.java
* WindowAnimator.java
* WindowStateAnimator.java
* WindowSurfacePlacer.java
* WindowSurfaceController.java

## Window 组织方式

ActivityManagerService（AMS），WindowManagerService（WMS），SurfaceFlinger（SF）等几个模块相互合作。App负责业务逻辑，绘制自己的视图；AMS管理组件、进程信息和Activity的堆栈及状态等等；WMS管理Activity对应的窗口及子窗口，还有系统窗口等；SF用于管理图形缓冲区，将App绘制的东西合成渲染在屏幕上。 窗口管理系统主要框架：

窗口管理系统WMS是Android中的主要子系统之一，它涉及到App中组件的管理，系统和应用窗口的管理和绘制等工作。由于其涉及模块众多，且与用户体验密切相关，所以它也是Android当中最为复杂的子系统之一。一个App从启动到主窗口显示出来，需要App，ActivityManagerService（AMS），WindowManagerService（WMS），SurfaceFlinger（SF）等几个模块相互合作。App负责业务逻辑，绘制自己的视图；AMS管理组件、进程信息和Activity的堆栈及状态等等；WMS管理Activity对应的窗口及子窗口，还有系统窗口等；SF用于管理图形缓冲区，将App绘制的东西合成渲染在屏幕上。



主要对象功能介绍：

WindowManagerService负责完成窗口的管理工作

WindowState和应用端窗口一一对应，应用调用WMS添加窗口时，最终会在WindowManagerService.addWindow()创建一个WindowState与之一一对应

WindowToken是一个句柄，保存了所有具有同一个token的WindowState。应用请求WindowManagerService添加窗口的时候，提供了一个token，该token标识了被添加窗口的归属，WindowManagerService为该token生成一个WindowToken对象，所有token相同的WindowState被关联到同一个WindowToken，如输入法添加窗口时，会传递一个IBinder mCurToken，墙纸服务添加窗口时，会传递一个WallpaperConnection::final Binder mToken。

AppWindowToken继承于WindowToken，专门用于标识一个Activity。AppWindowToken里的token实际上就是指向了一个Activity。ActivityManagerService通知应用启动的时候，在服务端生成一个token用于标识该Activity，并且把该token传递到应用客户端，客户端的Activity在申请添加窗口时，以该token作为标识传递到WindowManagerService。同一个Activity中的主窗口、对话框窗口、菜单窗口都关联到同一个AppWindowToken。

Session表示一个客户端和服务端的交互会话。一般来说不同的应用通过不同的会话来和WindowManagerService交互，但是处于同一个进程的不同应用通过同一个Session来交互。

## Android Token

Token是ActivityRecord的内部静态类，我们先来看下Token的继承关系，Token extends IApplicationToken.Stub，从IApplicationToken.Stub类进行继承，根据Binder的机制可以知道Token是一个匿名Binder实体类，这个匿名Binder实体会传递给其他进程，其他进程会拿到Token的代理端。 我们知道匿名Binder有两个比较重要的用途，一个是拿到Binder代理端后可跨Binder调用实体端的函数接口，另一个作用便是在多个进程中标识同一个对象。往往这两个作用是同时存在的，比如我们这里研究的Token就同时存在这两个作用，但最重要的便是后者，Token标识了一个ActivityRecord对象，即间接标识了一个Activity。

Token梳理： 分析源码，我们发现，大多数 token 的对象，都表示一个 IBinder 对象。提到 IBinder，大家一点也不陌生，就是 Android 的 IPC 通信机制。在创建窗口过程中，涉及到的 IPC 通信，无非包含两方面，一个是 WmS 用来跟应用所在的进程进行通信的 ViewRootImpl.W 类的对象，另一个是指向一个 ActivityRecord 的对象，自然应该是WMS用来跟 AMS进行通信的了。我们梳理了一下，token 以下几处的定义，分别来讲讲这里的 token 代表什么。



分析一下 View 的 AttachInfo 的赋值。ViewRootImpl 在构建方法里，会初始化一个 AttachInfo 实例，把它的 Session，以及 W类对象赋值给 AttachInfo。分析可以看到，AttachInfo 中的 mWindowToken，与mWindow 都是指向 ViewRootImpl 中的 mWindow(W类实例)。当一个 View attach 到窗口后，ViewRootImpl会执行performTraversals，如果发现是首次调用会，会把自己的 mAttachInfo 传递给根 View（通过dispatchAttachedToWindow），告诉 View 树现在已经 attch to Window 了，马上可以显示了。根 View（一般是 ViewGroup）会把这个信息，遍历地传递给 View 树中的每一个子 View，这样每个 View 的 mAttachInfo 都被赋值为 ViewRootImp 的 mAttachInfo了。

### Token对象的创建

下面这个图是Token的传递，首先会传递到WMS中，接着会传递到应用进程ActivityThread中，下面来具体分析这个传递流程。 总体流程图



## 源码（部分）：

## SampleWindow的实现

一个从命令行启动的动画窗口

<https://blog.csdn.net/innost/article/details/47660193>



## 类分析

final WindowManagerPolicy mPolicy = new PhoneWindowManager();

InputMonitor.java的主要目的还是解耦,防止PhoneWindowManager和InputManagerService太紧,

public int interceptKeyBeforeQueueing(KeyEvent event, int policyFlags) {

return mService.mPolicy.interceptKeyBeforeQueueing(event, policyFlags);

}

## WindowManagerPolicy

### Policy flags.事件策略

// Policy flags. These flags are also defined in frameworks/base/include/ui/Input.h.

public final static int FLAG\_WAKE = 0x00000001;

public final static int FLAG\_VIRTUAL = 0x00000002;

public final static int FLAG\_INJECTED = 0x01000000;

public final static int FLAG\_TRUSTED = 0x02000000;

public final static int FLAG\_FILTERED = 0x04000000;

public final static int FLAG\_DISABLE\_KEY\_REPEAT = 0x08000000;

public final static int FLAG\_INTERACTIVE = 0x20000000;

public final static int FLAG\_PASS\_TO\_USER = 0x40000000;

# WindowManagerService

本章涉及的源代码文件名及位置：

frameworks/base/services/java/com/android/server/SystemServer.java

frameworks/base/services/java/com/android/server/wm/WindowManagerService.java

frameworks/base/services/java/com/android/server/am/ActivityStack.java

frameworks/base/services/java/com/android/server/wm/WindowState.java

frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindowManager.java

frameworks/base/core/java/android/view/animation/AccelerateDecelerateInterpolator.java

frameworks/base/core/java/android/view/animation/Animation.java

frameworks/base/core/java/android/view/animation/AlphaAnimation.java

frameworks/base/services/java/com/android/server/wm/WindowAnimator.java

frameworks/base/services/java/com/android/server/wm/WindowStateAnimator.java

# 初识窗口的创建、绘制与销毁

# 理解Window和WindowManager

对于重要的系统服务在手机开机的时候进行初始化, 然后保存在了系统进程中. 当我们应用启动的时候会触发ContextImpl类的加载, 在类加载的时候通过静态代码块对各个系统服务进行进行了注册,并保存到一个静态map容器中, 之后就可以通过getSystemService(serviceName)的形式获取不同的系统服务.

Window表示的是一个窗口的概念，在日常生活中使用的并不是很多，但是某些特殊的需求还是需要的，比如悬浮窗之类的，他的具体实现是PhoneWindow,

创建一个Window很简单，只需要WindowManager去实现，WindowManager是外界访问Window的入口，Window的具体实现是在WindowManagerService中，他们两个的交互是一个IPC的过程，Android中的所有视图都是通过Windowl来实现的，无论是Activity,Dialog还是Toast,他们的视图都是直接附加在Window上的，因此Window是View的直接管理者，在之前的事件分发中我们说到，View的事件是通过WIndow传递给DecorView，然后DecorView传递给我们的View，就连Activity的setContentView,都是由Window传递的。

## Window和WindowManager



**Window与WindowManager基础关系**

ViewManager接口定义了一组规则，也就是add、update、remove的操作View接口。也就是说ViewManager是用来添加和移除activity中View的接口

WindowManager继承自ViewManager，然后自己还是一个接口，同时又定义了一个静态内部类LayoutParams（这个类比较重要，后面会分析。提前透漏下，如果你在APP做过类似360助手屏幕的那个悬浮窗或者做过那种类似IOS的小白圆点，点击展开菜单功能，你或多或少就能猜到这个类的重要性）。WindowManager用来在应用与Window之间的接口、窗口顺序、消息等的管理。

ViewManager的另一个实现子类ViewGroup

|  |
| --- |
| public abstract class ViewGroup extends View implements ViewParent, ViewManager {  //protected ViewParent mParent;  //这个成员是View定义的，ViewGroup继承自View，所以也可以拥有。  //这个变量就是前面我们一系列文章分析View向上传递的父节点，类似于一个链表Node的next一样  //最终指向了ViewRoot  ......  public void addView(View child, LayoutParams params) {  addView(child, -1, params);  }  ......  public void addView(View child, int index, LayoutParams params) {  ......  // addViewInner() will call child.requestLayout() when setting the new LayoutParams  // therefore, we call requestLayout() on ourselves before, so that the child's request  // will be blocked at our level  requestLayout();  invalidate(true);  addViewInner(child, index, params, false);  }  ......  } |

所以说View通过ViewGroup的addView方法添加到ViewGroup中，而ViewGroup层层嵌套到最顶级都会显示在在一个窗口Window中, 对于一个Activity只有一个DecorView（ViewRoot），也只有一个Window。

为了了解Window的工作机制，我们首先来看下如何通过WindowManager来创建一个Window.

|  |
| --- |
| Button btn = new Button(this);  btn.setText("我是窗口");  WindowManager wm = (WindowManager) getSystemService(WINDOW\_SERVICE);  WindowManager.LayoutParams layout = new WindowManager.LayoutParams(WindowManager.LayoutParams.WRAP\_CONTENT  , WindowManager.LayoutParams.WRAP\_CONTENT, 0, 0, PixelFormat.TRANSLUCENT);  layout.flags = WindowManager.LayoutParams.FLAG\_NOT\_TOUCH\_MODAL  | WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE  | WindowManager.LayoutParams.FLAG\_SHOW\_WHEN\_LOCKED;  layout.gravity = Gravity.CENTER;  layout.type = WindowManager.LayoutParams.TYPE\_PHONE;  layout.x = 300;  layout.y = 100;  wm.addView(btn, layout); |

述的代码，其中type和flag是比较重要的，我们来看下

**Flag参数表示window的属性**，他有很多选项，我们挑几个重点的

* FLAG\_NOT\_FOCUSABLE: 表示窗口不需要获取焦点，也不需要接收各种事件，这属性会同时启动FLAG\_NOT\_TOUCH\_MODAL，最终的事件会传递给下层的具体焦点的window.
* FLAG\_NOT\_TOUCH\_MODAL:在此模式下，系统会将当前window区域以外的单击事件传递给底层的Window，此前的Window区域以内的单机事件自己处理，这个标记很重要，一般来说都需要开启，否则其他window将无法获取单击事件
* FLAG\_SHOW\_WHEN\_LOCKED: 开启这个属性可以让window显示在锁屏上.

**Type参数表示window的类型**，window有三种类型，分别是应用，子，系统，应用window对应一个Activity,子Window不能单独存在，需要依赖一个父Window，比如常见的Dialog都是子Window,系统window需要声明权限，比如系统的状态栏.Window是分层的，每个Window对应着z-ordered,层级大的会覆盖在层级小的Window上面，这和HTML中的z-index的概念是一致的，在这三类中.

* 应用是层级范围是1-99.
* 子window的层级是1000-1999
* 系统的层级是2000-2999。

这些范围对应着type参数，如果想要window在最顶层，那么层级范围设置大一点就好了，很显然系统的值要大一些，系统的值很多，我们一般会选择TYPE\_SYSTEM\_OVERLAY和TYPE\_SYSTEM\_ERROR，记得要设置权限.

**<uses-permission android:name="android.permission.SYSTEM\_ALERT\_WINDOW"/>**

WindowManager所提供的功能很简单，常用的有三个方法，添加View,更新View,删除View,这三个方法定义在ViewManager中，而WindowManager继承自ViewManager:

|  |
| --- |
| public interface ViewManager {  public void addView(View view, ViewGroup.LayoutParams params);  public void updateViewLayout(View view, ViewGroup.LayoutParams params);  public void removeView(View view);  } |

虽然只有三个功能，但是这三个功能足够我们使用了，我们常见的可以推动的View，其实也很好实现，就是不断的更改他xy的位置.

|  |
| --- |
| button.setOnTouchListener(new View.OnTouchListener() {  @Override  public boolean onTouch(View v, MotionEvent event) {  int rawX = (int) event.getRawX();  int rawY = (int) event.getRawY();  switch (event.getAction()) {  case MotionEvent.ACTION\_MOVE:  layout.x = rawX;  layout.y = rawY;  wm.updateViewLayout(button, layout);  break;  }  return false;  }  }); |

## Window的内部机制

Window是一个抽象的概念，每一个Window对应着一个View和一个ViewRootImpl,Window和View通过ViewRootImpl建立关系，因此Window并不是实际存在的，这点从WindowManager定义的接口都是针对View，这说明View才是window的实体，在实际使用当中我们并不能直接访问Window的添加过程.

### Window的添加过程

Window的添加过程是通过WindowManager的addView去实现的，而真正实现的是一个接口，也就是WindowManagerImpl.

|  |
| --- |
| Activity.attach() {  mWindow = new PhoneWindow(this, window);  mWindow.setWindowManager(  (WindowManager)context.getSystemService(Context.WINDOW\_SERVICE));  **mWindowManager = mWindow.getWindowManager();**  }  Window.setWindowManager() {  mWindowManager = ((**WindowManagerImpl**)wm).createLocalWindowManager(this);  } |

|  |
| --- |
| @Override  public void addView(View view, ViewGroup.LayoutParams params) {  **mGlobal**.addView(view, params, mDisplay, mParentWindow);  }  @Override  public void updateViewLayout(View view, ViewGroup.LayoutParams params) {  mGlobal.updateViewLayout(view, params);  }  @Override  public void removeView(View view) {  mGlobal.removeView(view, false);  } |

可以发现，WindowManagerImpl并没有直接去实现一个Window的三大操作，而是全部交给了WindowManagerGlobal来处理，WindowManagerGlobal是一个工厂的性质提供自己的实现，在WindowManagerGlobal中有一段如下的代码：

private final WindowManagerGlobal mGlobal = WindowManagerGlobal.getInstance();

WindowManagerImpl这种工作模式就是典型的桥接模式，将所有的操作全部委托给WindowManagerGlobal（**@hide**这是系统隐藏类，普通app不可见！）去实现，WindowManagerGlobal的addView方法主要分如下几步：

|  |
| --- |
| public final class WindowManagerGlobal {  private final ArrayList<View> mViews = new ArrayList<View>();  private final ArrayList<ViewRootImpl> mRoots = new ArrayList<ViewRootImpl>();  private final ArrayList<WindowManager.LayoutParams> mParams =  new ArrayList<WindowManager.LayoutParams>();  public void addView(View view, ViewGroup.LayoutParams params,  Display display, Window parentWindow) {  ......  ViewRootImpl root;  View panelParentView = null;  synchronized (mLock) {  ......  **//1.创建ViewRootImpl**  root = new ViewRootImpl(view.getContext(), display);  view.setLayoutParams(wparams);  **//2.添加到列表中**  mViews.add(view);  mRoots.add(root);  mParams.add(wparams);  }  **//3. 最后执行这个，将View显示在手机屏幕上**  try {  root.setView(view, wparams, panelParentView);  } catch (RuntimeException e) {  ......  }  }  } |

#### 创建ViewRootImpl

**ViewRootImpl的作用**可以总结为: Android Framework与WMS之间的通信也是通过Binder机制进行的. 这里就建立了与WMS的通信. 最后通过openSession()函数来与WMS建立一个通信会话, 相当于建立了一个长期的处理中心, 双方有什么需要够可以通过这个Session来交换信息.

WMS是运行在SystemServer进程。 ViewRootImpl是通信的桥梁

上面的方法中先是new了一个ViewRootImpl，然后调用他的setView来显示布局。

Android中的View都是通过ViewRootImpl来完成绘制的。

|  |
| --- |
| ublic final class ViewRootImpl implements ViewParent,  View.AttachInfo.Callbacks, ThreadedRenderer.HardwareDrawCallbacks {  public ViewRootImpl(Context context, Display display) {  mContext = context;  //获取IWindowSession，与WindowManagerService建立连接  mWindowSession = **WindowManagerGlobal.getWindowSession();**  //这里保存当前线程  mThread = Thread.currentThread();  }  } |

继续追踪WindowManagerGlobal.getWindowSession()：

|  |
| --- |
| public static IWindowSession **getWindowSession**() {  synchronized (WindowManagerGlobal.class) {  if (sWindowSession == null) {  try {  InputMethodManager imm = InputMethodManager.getInstance();  //获取WindowManagerService  IWindowManager windowManager = **getWindowManagerService**();  //与系统的WindowManagerService建立一个IWindowSession  sWindowSession = windowManager.openSession(  new IWindowSessionCallback.Stub() {  @Override  public void onAnimatorScaleChanged(float scale) {  ValueAnimator.setDurationScale(scale);  }  },  imm.getClient(), imm.getInputContext());  } catch (RemoteException e) {  throw e.rethrowFromSystemServer();  }  }  return sWindowSession;  }  }  public static IWindowManager getWindowManagerService() {  synchronized (WindowManagerGlobal.class) {  if (sWindowManagerService == null) {  //这里返回的是IBinder对象，进行IPC通信  **sWindowManagerService = IWindowManager.Stub.asInterface(**  **ServiceManager.getService("window"));**  ......  }  return sWindowManagerService;  }  } |

继续看 sWindowManagerService = IWindowManager.Stub.asInterface(

ServiceManager.getService("window"));这行：

|  |
| --- |
| public final class ServiceManager {  public static IBinder getService(String name) {  try {  IBinder service = sCache.get(name);  if (service != null) {  return service;  } else {  return getIServiceManager().getService(name);  }  } catch (RemoteException e) {  Log.e(TAG, "error in getService", e);  }  return null;  }  } |

所以IWindowManager.Stub.asInterface(ServiceManager.getService("window"));得到的是一个IBinder对象。到这里，在ServiceManager的getService方法中通过getIServiceManager().getService("window")获取到一个IBinder，与WMS建立初步连接。然后通过IWindowManager.Stub.asInterface方法将IBinder转换成IWindowManager对象。通过这个对象调用openSession打开一个Session，实现通话。但是，WMS只负责管理View的z-order，也就是管理当前那个View在最上层显示，并不管理绘制。

#### ViewRootImpl.setView

绘制View的任务就在ViewRootImpl的setView方法中。

|  |
| --- |
| public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {  synchronized (this) {  ......  //请求绘制View  requestLayout();  ......  int res; /\* = WindowManagerImpl.ADD\_OKAY; \*/  try {  ......  //请求WindowManagerService，让WMS实现Window的添加。  res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,  getHostVisibility(), mDisplay.getDisplayId(),  mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,  mAttachInfo.mOutsets, mInputChannel);  }  ......  }  } |

setView主要做了两件事：

（1）requestLayout();

（2）向WMS请求添加Window；

#### requestLayout();

@Override

public void requestLayout() {

if (!mHandlingLayoutInLayoutRequest) {

checkThread();

mLayoutRequested = true;

scheduleTraversals();

}

}

**这里我另外注意一下checkThread()这个方法。**

void checkThread() {

if (mThread != Thread.currentThread()) {

throw new CalledFromWrongThreadException(

"Only the original thread that created a view hierarchy can touch its views.");

}

}

是不是看到了熟悉的异常提示？这个方法要判断当前更新UI 的线程是不是创建ViewRootImpl时的线程，只有在创建ViewRootImpl的线程中更新对应的UI才不会报错。所以不能在子线程中更新UI也是这个原因。

setView->requestLayout->scheduleTraversals—>mTraversalRunnable->doTraversal->performTraversals来通过一个handler把这个任务发送出去：完成异步刷新请求

看performTraversals();这个方法里将近一千行代码,大致是四个过程：

private void performTraversals(){

// 1 获取Surface对象，用于图形绘制

//2 测量整个视图树中各个View的大小，用performMeasure方法

//3 布局整个视图树，用performLayout方法

//4 绘制整个视图树，用performDraw方法

}

performDraw->ViewRootImpl.draw()方法->.ViewRootImpl.drawSoftware方法，调用GPU绘图。

draw()中获取到绘制表面Surface，里面最后调用ViewRootImpl的drawSoftware方法，调用GPU绘图。

private boolean drawSoftware(Surface surface, AttachInfo attachInfo, int xoff, int yoff,

boolean scalingRequired, Rect dirty) {

final Canvas canvas;

try {

//获取canvas

canvas = mSurface.lockCanvas(dirty);

// TODO: Do this in native

canvas.setDensity(mDensity);

} catch (Surface.OutOfResourcesException e) {

}

try {

if (!canvas.isOpaque() || yoff != 0 || xoff != 0) {

canvas.drawColor(0, PorterDuff.Mode.CLEAR);

}

try {

......

//从这里开始绘制整个视图树，从DecorView开始

mView.draw(canvas);

} finally {

......

}

} finally {

try {

//解锁canvas，并通知SurfaceFlinger更新这块区域。

surface.unlockCanvasAndPost(canvas);

} catch (IllegalArgumentException e) {

......

}

}

return true;

}

综上所述，视图树的绘制主要有以下步骤：

（1）判断使用CPU还是GPU绘制

（2）获取绘制表面Surface对象

（3）通过Surface对象获取并锁住绘图对象

（4）从DecorView开始绘制整个视图树

（5）解锁Canvas，并通知SurfaceFlinger更新这块区域。

**2.创建ViewRootImpl并将View添加到列表中**

在WindowManagerGlobal有如下几个列表是比较重要的.

|  |
| --- |
| private final ArrayList<View> mViews = new ArrayList<View>();  private final ArrayList<ViewRootImpl> mRoots = new ArrayList<ViewRootImpl>();  private final ArrayList<WindowManager.LayoutParams> mParams =  new ArrayList<WindowManager.LayoutParams>();  private final ArraySet<View> mDyingViews = new ArraySet<View>(); |

在上面的声明中，mViews存储所有window所对应的View，mRoots存储是所有window所对应的ViewRootImpl，mParams存储是所对应的布局参数，而mDyingViews则存储那些正在被删除的对象，在addView中通过如下方式将Window的一系列对象添加到列表中.

|  |
| --- |
| root = new ViewRootImpl(view.getContext(), display);  view.setLayoutParams(wparams);  mViews.add(view);  mRoots.add(root);  mParams.add(wparams); |

如此一来,Window的添加请求就交给WindowManagerService去处理了，在WindowManagerService内部为每一个应用添加了一个单独的Session,具体WIndow在WindowManagerService中怎么添加的，我们就不去分析了，读者可以自己去熟悉下.

### Window的删除过程

Window的删除过程和添加过程一样，都是通过Impl再通过Global来实现了，下面是Global的removeView.

|  |
| --- |
| public void removeView(View view) {  synchronized (this) {  mViews.remove(view);  for (int i = mViewRects.size(); i-- > 0; ) {  AttachInfo.InvalidateInfo info = mViewRects.get(i);  if (info.target == view) {  mViewRects.remove(i);  info.recycle();  }  }  if (mPosted && mViews.isEmpty() && mViewRects.isEmpty()) {  mChoreographer.removeCallbacks(Choreographer.CALLBACK\_ANIMATION, this, null);  mPosted = false;  }  } |

removeView的代码很清晰，首先通过findViewLocked来查找待删除的索引，这个超找过程就是建立的数组遍历，然后进一步的删除。

removeViewLocked是通过ViewRootImpl来完成删除操作的，在windowmanager中提供了两种接口removeView和removeViewImmediate，他们分别表示异步删除和同步删除，其中removeViewImmediate，使用起来要格外注意，一般来说不需要使用此方法来删除window以免发生意外的错误，这里主要是异步删除的问题，具体的删除操作是ViewImple的die方法来完成的，在异步删除的情况下，die只是发生一个删除的请求后就返回了，这个时候View并没有完成删除的操作，所有最后会将其添加到mDyingViews中，mDyingViews表示待删除的View列表，ViewRootImpe的die方法如下：

在die方法内部只是做了简单的判断，那么就发送了一个MSG\_DIE的消息，ViewRootImpl中的mHandler会处理此消息并且并调用doDie方法，如果是同步删除就会直接调用doDie方法，在doDie方法内部会操作dispatchDetachedFromWindow，真正删除window就是在这里面实现的，他主要做了四件事。

1.垃圾回收相关的工作，比如清除数据和消息，移除回调

2.通过Session的remove方法来删除window,这同样是一个IPC的过程，最终会调用wms的removeWindow方法

3.调用view的dispatchDetachedFromWindow方法，在内不会调用onDetachedFromWindow,他做了一些回收资源或者停止动画的一些操作

4.调用WindowManagerGlobal的doRemoveView方法刷新数据

### Window的更新过程

到这里，window的删除就接收完了，在说下更新，需要看WindowManagerGlobal的updateViewLayout方法。

|  |
| --- |
| public void updateViewLayout(View view, ViewGroup.LayoutParams params) {  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);  }  } |

updateViewLayout做的方法比较简单，首先他更新View的LayoutParams替换老的，接着再更新下ViewRootimpl中的LayoutParams，这一步是通过viewrootimpl的setLayoutParams来做的，在ViewRootImpl中会通过scheduleTraversals方法来对View，测量，布局，重绘等等，除了view本身的重绘之外，ViewRootImpl还会通过WindowSession来更新Window的视图，这个过程最终是WindowManagerService的relayoutWindow来实现的，具体也是一个IPC的过程

## Window的创建过程

通过上面的分析，我们知道，view是android中视图的呈现方式，但是view不能单独存在，他必须依附在window这个抽象类中，因此有视图的地方就有window，activity，toast都是，我们继续来分析window的创建过程。

### Activity的Window创建过程

要分析Activity的Window创建过程就需要去了解activity的启动过程，详细的会在后面说，这里简单概括，activity的启动过程很复杂，最终会由ActivityThread中的perfromLaunchActivity()来完成整个启动过程。

在Activity的attach方法中，系统会创建activity所属的window对象PhoneWindow并为其设置回调接口，window对象的创建过程是由PolicyManager的akeNewWindow方法实现的，由于activity实现了window的callback方法接口，因此当window接受到外界的状态改变的时候就会去调用activity的方法，callback接口中的方法很多，但是有几个确实我们非常熟悉的，你如onAttachedToWindow等。

下面分析activity的视图是怎么依附在window上的由于activity的视图是由setContentView开始的，所有我们先看下这个方法：

|  |
| --- |
| public void setContentView(@LayoutRes int layoutResID) {  getWindow().setContentView(layoutResID);  initWindowDecorActionBar();  } |

1.如果没有DecorView就去创建他

DecorView是一个FrameLayout,在之前就已经说过了，这里说一下。DecorView是activity中的顶级View,一般来说他的内部包含标题栏和内部栏，但是这个会随着主题的变化而发生改变的，不管怎么样，内容是一定要存在的，并且内容有固定的id，那就是content,完整的就是android.R.id.content，DecorView的创建是由installDecor方法来完成的，在方法内部会通过generateDecor方法来完成创建DecorView,这个时候他就是一个空白的FrameLayout.

|  |
| --- |
| protected DecorView generateDecor(){  return new DecorView(getContext(),-1);  } |

为了初始化DecorView的结构，PhoneWindow还需要通过generateLayout方法来加载具体的布局文件到DecorView中，这个跟主题有关.

|  |
| --- |
| View in = mLayoutInflater.inflate(layoutResource, null);  decor.addView(in, new ViewGroup.LayoutParams(MATCH\_PARENT, MATCH\_PARENT));  mContentRoot = (ViewGroup) in;  ViewGroup contentParent = (ViewGroup)findViewById(ID\_ANDROID\_CONTENT); |

其中ID\_ANDROID\_CONTENT的定义如下，这个id对应的就是ViewGroup的mContentParent

public static final int ID\_ANDROID\_CONTENT = com.android.internal.R.id.content

2.将View添加到DecorView的mContentParent中

这个过程比较简单，由于在第一步的时候已经初始化了DecorView，因此这一部就直接将activity的视图添加到DecorView的mContentParent中既可，mLayoutInflater.inflate(layoutResID, mContentParent);到此为止，由此可以理解activity的setcontentview的来历了，也许有读者会怀疑，为什么不叫setview来，他明明是给activity设置视图啊，从这里来看，他的确不适合叫做setview,因为activity的布局文件只是添加到了DecorView的mContentParent中，因此交setContentView更加准确。

3.回调Activity的onCreateChanged方法来通知Activity视图已经发生改变

这个过程很简单，由于window实现了Callback接口，这里表示布局已经被添加到DecorView的mContentParent中了，于是通知activity。使其可以做相应的处理Activity的onCreateChanged是一个空实现，我们可以在子activity处理这个回调

|  |
| --- |
| final callback cb = getCallback();  if(cb != null && !isDestroyed()){  cb.onContentChanged();  } |

经过了上面的三个步骤，到这里为止DecorView已经被创建并且初始化完毕了activity的布局文件也已经添加到了DecorView的内容中，但是这个时候DecorView还没有被windowmanager添加到window中，这里需要正确的理解window的概念，window更多的是表示一种抽象的功能集合，虽然说早在activity的attch中window就已经被创建了，但是这个时候由于DecorView还没有被windowmanager识别，所有还不能提供具体的功能，因为他还无法接收外界的输入，在activityThread的makeVisible中，才能被视图看到：

|  |
| --- |
| void makeVisible() {  if (!mWindowAdded) {  ViewManager wm = getWindowManager();  wm.addView(mDecor, getWindow().getAttributes());  mWindowAdded = true;  }  mDecor.setVisibility(View.VISIBLE);  } |

到这里，window的创建过程就已经分析完了

### Dialog的Window创建过程



#### 创建Window

dialog的window创建过程和activity的类似，我们来看下

|  |
| --- |
| Dialog(Context context, int theme, boolean createContextThemeWrapper) {  if (createContextThemeWrapper) {  if (theme == 0) {  TypedValue outValue = new TypedValue();  context.getTheme().resolveAttribute(com.android.internal.R.attr.dialogTheme,  outValue, true);  theme = outValue.resourceId;  }  mContext = new ContextThemeWrapper(context, theme);  } else {  mContext = context;  }  mWindowManager = (WindowManager)context.getSystemService(Context.WINDOW\_SERVICE);  Window w = PolicyManager.makeNewWindow(mContext);  mWindow = w;  w.setCallback(this);  w.setOnWindowDismissedCallback(this);  w.setWindowManager(mWindowManager, null, null);  w.setGravity(Gravity.CENTER);  mListenersHandler = new ListenersHandler(this);  } |

#### 初始化DecorView并将Dialog的师徒添加到DecorView

这个过程也和activity的类似，都是通过window去指定加载的布局文件:

|  |
| --- |
| public void setContentView(int layoutResID) {  mWindow.setContentView(layoutResID);  } |

#### 将DecorView添加到window并且显示

在Dialog的show方法中，会通过windowmanager将DecorView添加到window中

|  |
| --- |
| mWindowManager.addView(mDecor, l);  mShowing = true; |

从上述的三个步骤可以发现，dialog的创建过程和activity很类似

普通的dialog有一个特殊的地方，那就是必须用activity的context，否则会报错

Dialog dialog = new Dialog(this);

dialog.setTitle("Hello");

dialog.show();



上面的信息非常的明确，是没有应用token导致的，而应用token一般只有activity持有，所有这里需要activity的context.

另外系统比较特殊，他可以不需要token. 使用过程：在AndroidManifest.xml中加上权限android.permission.SYSTEM\_ALERT\_WINDOW; 调用dialog.show()之前加alertDialog.getWindow().setType(WindowManager.LayoutParams.TYPE\_SYSTEM\_ERROR); 系统的type有很多都可以用，TYPE\_SYSTEM\_ERROR，TYPE\_SYSTEM\_OVERLAY都可以。

但是Android6.0的则需要通过代码申请权限，使用要特别注意，实在嫌麻烦，最好自己new一个activity当dlg。

### Toast的Window创建过程

Toast和dialog不同，他的工作过稍微复杂一点，首先Toast也是基于Window来实现的，但是由于Toast具有定时取消的功能，所以系统采用了handler，在toast内部有两类IPC的过程，第一类Toast访问NotificationManagerService，第二类是NotificationManagerService回调toast的TN接口

Toast具有系统的window,他内部试下的师徒有两种方式制定，一个是系统设定，还可以setview指定

|  |
| --- |
| public void show() {  if (mNextView == null) {  throw new RuntimeException("setView must have been called");  }  INotificationManager service = getService();  String pkg = mContext.getOpPackageName();  TN tn = mTN;  tn.mNextView = mNextView;  try {  service.enqueueToast(pkg, tn, mDuration);  } catch (RemoteException e) {  // Empty  }  }  /\*\*  \* Close the view if it's showing, or don't show it if it isn't showing yet.  \* You do not normally have to call this. Normally view will disappear on its own  \* after the appropriate duration.  \*/  public void cancel() {  mTN.hide();  try {  getService().cancelToast(mContext.getPackageName(), mTN);  } catch (RemoteException e) {  // Empty  }  } |

从上面的代码可以看出，显示和隐藏是通过NMS来实现的，由于NMS运行在系统，所以只能通过远程调用，这里就跨进程实现了IPC，在这个时候NMS是运行在binder线程池中，所以需要handler切换到主线程中，所以这就意味着Toast无法**在没有Lopper的线程中弹出.**

我们首先来看下Toast的显示过程

|  |
| --- |
| INotificationManager service = getService();  String pkg = mContext.getOpPackageName();  TN tn = mTN;  tn.mNextView = mNextView;  try {  service.enqueueToast(pkg, tn, mDuration);  } catch (RemoteException e) {  // Empty  } |

NMS的enqueueToast方法的第一个参数就是当前应用的包名，第二个参数表示远程回调，第三个是时长，enqueueToast首先将Toast请求封装为一个ToastRecord对象将其添加到一个队列中，其实这本书就是一个arraylist，对于非系统应用来说，他只能存50个，这样是为了防止dos,如果不这样做，试想一下，其他应用还能弹出东西来吗？

代码丢失

正常情况下，一个应用不可能达到上限，当ToastRecord被添加到mToastQueue中后，NMS就会通过showNextToastLocked方法来显示Toast，下面的代码很好理解，需要注意的是，Toast的显示是由ToastRecord的callback来完成的，这个callback实际上就是TN对象的远程Binder，通过callback来访问TN中的方法是需要跨进程来完成的，最终被调用的对象是TN中方法发起在Binder线程池中。

|  |
| --- |
| void showNextToastLocked() {  ToastRecord record = mToastQueue.get(0);  while (record != null) {  if (DBG) Slog.d(TAG, "Show pkg=" + record.pkg + " callback=" + record.callback);  try {  record.callback.show(record.token);  scheduleTimeoutLocked(record);  return;  } catch (RemoteException e) {  Slog.w(TAG, "Object died trying to show notification " + record.callback  + " in package " + record.pkg);  // remove it from the list and let the process die  int index = mToastQueue.indexOf(record);  if (index >= 0) {  mToastQueue.remove(index);  }  keepProcessAliveIfNeededLocked(record.pid);  if (mToastQueue.size() > 0) {  record = mToastQueue.get(0);  } else {  record = null;  }  }  }  } |

Toast显示出来之后，NMS会通过scheduleTimeoutLocked来发送一个延时消息

|  |
| --- |
| private void scheduleTimeoutLocked(ToastRecord r)  {  mHandler.removeCallbacksAndMessages(r);  Message m = Message.obtain(mHandler, MESSAGE\_TIMEOUT, r);  long delay = r.duration == Toast.LENGTH\_LONG ? LONG\_DELAY : SHORT\_DELAY;  mHandler.sendMessageDelayed(m, delay);  } |

上面的代码，LONG\_DELAY是3.5s，SHORT\_DELAY是2s,NMS会通过cancelToastLocked来隐藏toast并且清楚队列。

|  |
| --- |
| ToastRecord record = mToastQueue.get(index);  try {  record.callback.hide();  } catch (RemoteException e) {  Slog.w(TAG, "Object died trying to hide notification " + record.callback  + " in package " + record.pkg);  // don't worry about this, we're about to remove it from  // the list anyway  } |

通过上面的分析，大家知道toast的显示隐藏实际上是toast的TN这个类来实现的，分别对应的show/hide，由于这两个方法都是NMS以跨进程的方式调用的，因此他运行在Binder池中，为了切换，在他们内部使用了handler

|  |
| --- |
| @Override  public void show(IBinder windowToken) {  if (localLOGV) Log.v(TAG, "SHOW: " + this);  mHandler.obtainMessage(0, windowToken).sendToTarget();  }  /\*\*  \* schedule handleHide into the right thread  \*/  @Override  public void hide() {  if (localLOGV) Log.v(TAG, "HIDE: " + this);  mHandler.post(mHide);  } |

上述的代码，mShow和mHide是两个Runnable，他们的内部实现分别调用了具体方法，由此可见，handlershow才是真正的方法，

|  |
| --- |
| public void handleShow(IBinder windowToken) {  if (localLOGV) Log.v(TAG, "HANDLE SHOW: " + this + " mView=" + mView  + " mNextView=" + mNextView);  if (mView != mNextView) {  // remove the old view if necessary  handleHide();  mView = mNextView;  Context context = mView.getContext().getApplicationContext();  String packageName = mView.getContext().getOpPackageName();  if (context == null) {  context = mView.getContext();  }  mWM = (WindowManager)context.getSystemService(Context.WINDOW\_SERVICE);  // We can resolve the Gravity here by using the Locale for getting  // the layout direction  final Configuration config = mView.getContext().getResources().getConfiguration();  final int gravity = Gravity.getAbsoluteGravity(mGravity, config.getLayoutDirection());  mParams.gravity = gravity;  if ((gravity & Gravity.HORIZONTAL\_GRAVITY\_MASK) == Gravity.FILL\_HORIZONTAL) {  mParams.horizontalWeight = 1.0f;  }  if ((gravity & Gravity.VERTICAL\_GRAVITY\_MASK) == Gravity.FILL\_VERTICAL) {  mParams.verticalWeight = 1.0f;  }  mParams.x = mX;  mParams.y = mY;  mParams.verticalMargin = mVerticalMargin;  mParams.horizontalMargin = mHorizontalMargin;  mParams.packageName = packageName;  mParams.hideTimeoutMilliseconds = mDuration ==  Toast.LENGTH\_LONG ? LONG\_DURATION\_TIMEOUT : SHORT\_DURATION\_TIMEOUT;  mParams.token = windowToken;  if (mView.getParent() != null) {  if (localLOGV) Log.v(TAG, "REMOVE! " + mView + " in " + this);  mWM.removeView(mView);  }  if (localLOGV) Log.v(TAG, "ADD! " + mView + " in " + this);  mWM.addView(mView, mParams);  trySendAccessibilityEvent();  }  } |

移除

|  |
| --- |
| public void handleHide() {  if (localLOGV) Log.v(TAG, "HANDLE HIDE: " + this + " mView=" + mView);  if (mView != null) {  // note: checking parent() just to make sure the view has  // been added... i have seen cases where we get here when  // the view isn't yet added, so let's try not to crash.  if (mView.getParent() != null) {  if (localLOGV) Log.v(TAG, "REMOVE! " + mView + " in " + this);  mWM.removeViewImmediate(mView);  }  mView = null;  }  }  } |

到这里，我们的window就全部分析完成了，相信大家对window有了一些新的见解了。

## 常见使用注意

1. 用Application作为dlg的context：
2. 在子线程更新ui
3. 实现系统转屏功能：

**ViewRootImpl**

**mWm** = WindowManagerGlobal.*getWindowManagerService*();

**mWm** = IWindowManager.Stub.*asInterface*(ServiceManager.*getService*(**"window"**));

**mWm**.freezeRotation(2);//180度

# WMS

frameworks/base/services/java/com/android/server/WindowManagerService.java

WMS只负责管理View的z-order，也就是管理当前那个View在最上层显示，并不管理绘制。

关键字：堆栈，

与Activity类似，Android系统中的窗口也是以**堆栈**的形式组织在WMS中的，其中，Z轴位置较低的窗口位于Z轴位置较高的窗口的下面（前台Z最大）。在本文中，我们就详细分析WMS是如何以堆栈的形式来组织窗口的。

应用程序进程中的每一个Activity组件在AMS中都对应有一个ActivityRecord对象。AMS的每一个ActivityRecord对象AMS中都对应有一个AppWindowToken对象。输入法窗口类似，在壁纸管理服务WallpaperManagerService中，每一个壁纸窗口都对应有一个Binder对象，这个Binder对象在WMS也对应有一个WindowToken对象。

在WMS中，无论是AppWindowToken对象，还是WindowToken对象，它们都是用来描述一组有着相同令牌的窗口的，每一个窗口都是通过一个WindowState对象来描述的。例如，一个Activity组件窗口可能有一个启动窗口（Starting Window），还有若干个子窗口，那么这些窗口就会组成一组，并且都是以Activity组件在AMS中所对应的AppWindowToken对象为令牌的。从抽象的角度来看，就是在AMS中，每一个令牌（AppWindowToken或者WindowToken）都是用来描述一组窗口（WindowState）的，并且每一个窗口的子窗口也是与它同属于一个组，即都有着相同的令牌。

上述的窗口组织方式如图1所示：



图1 窗口在WindowManagerService服务中的组织方式

其中，Activity Stack是在ActivityManagerService服务中创建的，Token List和Window Stack是在WindowManagerService中创建的，而Binder for IM和Binder for WP分别是在InputMethodManagerService服务和WallpaperManagerService服务中创建的，用来描述一个输入法窗口和一个壁纸窗口。

图1中的对象的对应关系如下所示：

1. ActivityRecord-J对应于AppWindowToken-J，后者描述的一组窗口是{WindowState-A, WindowState-B, WindowState-B-1}，其中， WindowState-B-1是WindowState-B的子窗口。

2. ActivityRecord-K对应于AppWindowToken-K，后者描述的一组窗口是{WindowState-C, WindowState-C-1, WindowState-D, WindowState-D-1}，其中， WindowState-C-1是WindowState-C的子窗口，WindowState-D-1是WindowState-D的子窗口。

3. ActivityRecord-N对应于AppWindowToken-N，后者描述的一组窗口是{WindowState-E}，其中， WindowState-E是系统当前激活的Activity窗口。

4. Binder for IM对应于WindowToken-I，后者描述的一组窗口是{WindowState-I}，其中， WindowState-I是WindowState-E的输入法窗口。

5. Binder for WP对应于WindowToken-W，后者描述的一组窗口是{WindowState-W}，其中， WindowState-W是WindowState-E的壁纸窗口。

从图1还可以知道，Window Stack中的WindowState是按照它们所描述的窗口的Z轴位置从低到高排列的。

以上就是WindowManagerService服务组织系统中的窗口的抽象模型，接下来我们将分析AppWindowToken、WindowToken和WindowState的一些增加、移动和删除等操作，以便可以对这个抽象模型有一个更深刻的认识。

## 增加AppWindowToken

从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，一个Activity组件在启动的过程中，ActivityManagerService服务会调用调用WindowManagerService类的成员函数addAppToken来为它增加一个AppWindowToken，如下所示：

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. /\*\*
6. \* Mapping from a token IBinder to a WindowToken object.
7. \*/
8. **final** HashMap<IBinder, WindowToken> mTokenMap =
9. **new** HashMap<IBinder, WindowToken>();
11. /\*\*
12. \* The same tokens as mTokenMap, stored in a list for efficient iteration
13. \* over them.
14. \*/
15. **final** ArrayList<WindowToken> mTokenList = **new** ArrayList<WindowToken>();
16. ......
18. /\*\*
19. \* Z-ordered (bottom-most first) list of all application tokens, for
20. \* controlling the ordering of windows in different applications.  This
21. \* contains WindowToken objects.
22. \*/
23. **final** ArrayList<AppWindowToken> mAppTokens = **new** ArrayList<AppWindowToken>();
24. ......
26. **public** **void** addAppToken(**int** addPos, IApplicationToken token,
27. **int** groupId, **int** requestedOrientation, **boolean** fullscreen) {
28. ......
30. **synchronized**(mWindowMap) {
31. AppWindowToken wtoken = findAppWindowToken(token.asBinder());
32. **if** (wtoken != ) {
33. ......
34. **return**;
35. }
36. wtoken = **new** AppWindowToken(token);
37. ......
38. mAppTokens.add(addPos, wtoken);
39. ......
40. mTokenMap.put(token.asBinder(), wtoken);
41. mTokenList.add(wtoken);
43. ......
44. }
45. }
47. ......
48. }

WMS类有三个成员变量mTokenMap、mTokenList和mAppTokens，它们都是用来描述系统中的窗口的。

成员变量mTokenMap指向的是一个HashMap，它里面保存的是一系列的WindowToken对象，每一个WindowToken对象都是用来描述一个窗口的，并且是以描述这些窗口的一个Binder对象的IBinder接口为键值的。例如，对于Activity组件类型的窗口来说，它们分别是以用来描述它们的一个ActivityRecord对象的IBinder接口保存在成员变量mTokenMap所指向的一个HashMap中的。

成员变量mTokenList指向的是一个ArrayList，它里面保存的也是一系列WindowToken对象，这些WindowToken对象与保存在成员变量mTokenMap所指向的一个HashMap中的WindowToken对象是一样的。成员变量mTokenMap和成员变量mTokenList的区别就在于，前者在给定一个IBinder接口的情况下，可以迅速指出是否存在一个对应的WindowToken对象，而后者可以迅速**遍**历WindowManagerService服务中的WindowToken对象。

成员变量mAppTokens指向的也是一个ArrayList，不过它里面保存的是一系列**AppWindowToken**对象，每一个AppWindowToken对象都是用来描述一个**Activity**组件窗口的，而这些AppWindowToken对象是以它们描述的窗口的Z轴坐标由小到大保存在这个ArrayList中的，这样我们就可以通过这个ArrayList来从上到下或者从下到上地遍历系统中的**所有Activity**组件窗口。由于这些AppWindowToken对象所描述的Activity组件窗口也是一个窗口，并且AppWindowToken类是从WindowToken继承下来的，因此，**这些AppWindowToken对象还会同时被保存在成员变量mTokenMap所指向的一个HashMap和成员变量mTokenList所指向的一个ArrayList中。**

理解了WMS类的这三个成员变量的含义之后，它的成员函数addAppToken的实现就好理解了，其中，参数token指向的便是用来描述正在启动的Activity组件所对应的一个ActivityRecord对象，而参数addPos用来描述该Activity组件在堆栈中的位置，这个位置同时也是接下来要创建的AppWindowToken对象在WMS类的mTokenList所描述的一个ArrayList中的位置。

WMS类的成员函数addAppToken首先调用另外一个成员函数findAppWindowToken来在成员变量mTokenMap所描述的一个HashMap检查是否已经存在一个AppWindowToken。如果已经存在的话，那么WindowManagerService类的成员函数addAppToken就什么也不做就返回了，否则的话，就会使用参数token来创建一个AppWindowToken对象，并且会将该AppWindowToken对象分别保存在WMS的成员变量mTokenMap、mTokenList和mAppTokens中。

## 删除AppWindowToken

删除AppWindowToken是通过调用WindowManagerService类的成员函数removeAppTokensLocked来实现的，如下所示：

1. **private** **void** removeAppTokensLocked(List<IBinder> tokens) {
2. // XXX This should be done more efficiently!
3. // (take advantage of the fact that both lists should be
4. // ordered in the same way.)
5. **int** N = tokens.size();
6. **for** (**int** i=0; i<N; i++) {
7. IBinder token = tokens.get(i);
8. **final** AppWindowToken wtoken = findAppWindowToken(token);
9. **if** (!mAppTokens.remove(wtoken)) {
10. ......
11. i--;
12. N--;
13. }
14. }
15. }

WindowManagerService类的成员函数removeAppTokensLocked可以同时删除一组AppWindowToken对象。

参数tokens所描述的是一个IBinder接口列表，与这些IBinder接口所对应的AppWindowToken对象就是接下来要删除的。WindowManagerService类的成员函数removeAppTokensLocked通过一个for循环来依次调用另外一个成员函数findAppWindowToken，以便可以找到保存在列表tokens中的每一个IBinder接口所对应的AppWindowToken对象，然后将该AppWindowToken对象从WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList中删除。

注意，WindowManagerService类的成员函数removeAppTokensLocked是在内部使用的，它只是把一个AppWindowToken对象从成员变量mAppTokens中删除，而没有从另外两个成员变量mTokenMap和mTokenList中删除。

## 移动AppWindowToken至指定位置

移动AppWindowToken至指定位置是通过调用WMS类的成员函数moveAppToken来实现的，如下所示：

1. **public** **void** moveAppToken(**int** index, IBinder token) {
2. **if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,
3. "moveAppToken()")) {
4. **throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");
5. }
7. **synchronized**(mWindowMap) {
8. ......
9. **final** AppWindowToken wtoken = findAppWindowToken(token);
10. **if** (wtoken ==  || !mAppTokens.remove(wtoken)) {
11. ......
12. **return**;
13. }
14. mAppTokens.add(index, wtoken);
15. ......
17. **final** **long** origId = Binder.clearCallingIdentity();
18. ......
19. **if** (tmpRemoveAppWindowsLocked(wtoken)) {
20. ......
21. reAddAppWindowsLocked(findWindowOffsetLocked(index), wtoken);
22. ......
23. updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES);
24. mLayoutNeeded = **true**;
25. performLayoutAndPlaceSurfacesLocked();
26. }
27. Binder.restoreCallingIdentity(origId);
28. }
29. }

参数token描述的是要移动的AppWindowToken对象所对应的一个IBinder接口，而参数index描述的是该AppWindowToken对象要移动到的位置。注意，移动一个AppWindowToken对象到指定的位置是需要android.Manifest.permission.MANAGE\_APP\_TOKENS权限的。

WMS类的成员函数moveAppToken首先找到与参数token所对应的AppWindowToken对象，并且将该AppWindowToken对象从WMS类的成员变量mAppTokens所描述的一个ArrayList中移除，这样做的目的是为了接下来可以将该AppWindowToken对象移动至该ArrayList中的指定位置上，即参数index所描述的位置上。

注意，上述操作只是将参数token所对应的AppWindowToken对象移动到了WMS类的成员变量mAppTokens所描述的一个ArrayList的指定位置上，接下来还需要同时将与该A**ppWindowToken对象所对应的WindowState**对象移动至WMS服务内部的一个WindowState堆栈合适位置上去。

**移动对应的WindowState**对象的操作同样也是分两步执行的：第一步先调用WMS类的成员函数tmpRemoveAppWindowsLocked来将这些WindowState对象从**原来的WindowState堆栈位置移除**；第二步再调用WMS类的成员函数reAddAppWindowsLocked来将这些WindowState对象**插入到WindowState堆栈的合适位置去**。

对应的WindowState对象被移动到的合适位置是通过调用WMS类的成员函数findWindowOffsetLocked来获得的，它的实现如下所示：

1. 参考/\*\*
2. \* Z-ordered (bottom-most first) list of all Window objects.
3. \*/
4. **final** ArrayList<WindowState> mWindows = **new** ArrayList<WindowState>();
5. ......
7. **private** **int** findWindowOffsetLocked(**int** tokenPos) {
8. **final** **int** NW = mWindows.size();
10. **if** (tokenPos >= mAppTokens.size()) {
11. **int** i = NW;
12. **while** (i > 0) {
13. i--;
14. WindowState win = mWindows.get(i);
15. **if** (win.getAppToken() != ) {
16. **return** i+1;
17. }
18. }
19. }
21. **while** (tokenPos > 0) {
22. // Find the first app token below the new position that has
23. // a window displayed.
24. **final** AppWindowToken wtoken = mAppTokens.get(tokenPos-1);
25. ......
26. **if** (wtoken.sendingToBottom) {
27. ......
28. tokenPos--;
29. **continue**;
30. }
31. **int** i = wtoken.windows.size();
32. **while** (i > 0) {
33. i--;
34. WindowState win = wtoken.windows.get(i);
35. **int** j = win.mChildWindows.size();
36. **while** (j > 0) {
37. j--;
38. WindowState cwin = win.mChildWindows.get(j);
39. **if** (cwin.mSubLayer >= 0) {
40. **for** (**int** pos=NW-1; pos>=0; pos--) {
41. **if** (mWindows.get(pos) == cwin) {
42. ......
43. **return** pos+1;
44. }
45. }
46. }
47. }
48. **for** (**int** pos=NW-1; pos>=0; pos--) {
49. **if** (mWindows.get(pos) == win) {
50. ......
51. **return** pos+1;
52. }
53. }
54. }
55. tokenPos--;
56. }
58. **return** 0;
59. }

参数tokenPos描述的是一个AppWindowToken对象在WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList的位置，WindowManagerService类的成员函数findWindowOffsetLocked的目标就要找到与该AppWindowToken对象所对应的WindowState对象在WindowManagerService服务内部的一个WindowState堆栈的起始偏移位置。有了这个起始偏移位置之后，我们就可以将对应的所有WindowState对象有序地插入到该WindowState堆栈中去。WindowManagerService服务内部的WindowState堆栈是通过WindowManagerService类的成员变量mWindows来描述的。接下来我们就分两种情况来分析这个起始偏移位置的计算过程。

第一种情况是参数tokenPos的值大于WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList的大小。这是一种**异常**情况，一般来说，参数tokenPos是指向mAppTokens列表的某一个位置的，不过这时候意味着它所描述的AppWindowToken对象的Z轴位置要大于mAppTokens列表的最上面的一个AppWindowToken对象的Z轴位置的。这也就是说，与参数tokenPos所描述的AppWindowToken对象所对应的WindowState对象的要位于与mAppTokens列表的最上面的一个AppWindowToken对象所对应的任一个WindoState对象的上面。因此，就需要找到与mAppTokens列表的最上面的一个AppWindowToken对象所对应的**Z轴位置最大的一个WindoState**对象在WindowState堆栈中的位置i，然后就可以知道与参数tokenPos所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈的起始偏移位置为i+1。

如何找到mAppTokens列表的最上面的一个AppWindowToken对象所对应的Z轴位置最大的一个WindoState对象在WindowState堆栈中的位置i呢？从图1可以可得到一个结论：WMS内部中的所有WindowState对象都是**按照Z轴从位置从小到大排列在WindowState堆栈中的**，并且在mAppTokens列表中，位于上面的一个AppWindowToken对象所对应的那些WindowState对象的Z轴位置是一定大于位于下面的一个AppWindowToken对象所对应的那些WindowState对象的Z轴位置的。**因此，我们只要从WindowState堆栈的顶端开始往下遍历，找到这样的一个WindowState对象，它是属于一个AppWindowToken对象的，即它的成员函数getAppToken的返回值不等于null**，那么它在WindowState堆栈中的位置就是我们要找到的位置i。有了这个位置i之后，将它的值加上1，就可以得到参数t所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈的起始偏移位置了。

第二种情况是参数tokenPos的值小于WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList的大小。根据前面得到的推论，我们只要在mAppTokens列表中找到一个AppWindowToken对象，它满足以下三个条件：

A. 它在mAppTokens列表中的位置小于tokenPos；

B. 它在WindowState堆栈中对应有WindowState对象；

C. 它不是将要置于WindowState堆栈的底部。

如果一个AppWindowToken对象在WindowState堆栈中对应有WindowState对象，那么这些WindowState对象也会同时按照Z轴从小到大的顺序保存它的成员变量windows所描述的一个ArrayList中，这意味着如果一个AppWindowToken对象满足条件B，那么它的成员变量windows所描述的一个ArrayList的大小就大于0。

如果一个AppWindowToken对象不是将要置于WindowState堆栈的底部，那么它的成员变量sendingToBottom的值就不等于true，这也意味这个AppWindowToken对象满足条件C。

如果能找到满足上述条件的一个AppWindowToken对象wtoken，那么我们只要找到与它所对应的Z轴位置最大的WindowState对象在WindowManagerService服务内部的WindowState堆栈中的位置i，那么将它的值加1，就可以得到与参数tokenPos所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈的起始偏移位置了。

那么如何找到与这个AppWindowToken对象wtoken对应的Z轴位置最大的WindowState对象在WMS服务内部的WindowState堆栈中的位置i呢？从前面的图1可以知道，一个AppWindowToken对象所对应的WindowState对象可以划分为两种类型：第一种类型是父窗口类型的；第二种是子窗口类型的。如果一个WindowState对象所描述的窗口是父窗口，那么它的子窗口就保存在它的成员变量mChildWindows所描述的一个ArrayList中，并且这些子窗口是按照Z轴位置从小到大的顺序排列的，同时，该WindowState对象也会保存在与它所对应的一个AppWindowToken对象的成员变量windows所描述的一个ArrayList中。

有了上述结论，并且假设存在一个能够满足上述三个条件的AppWindowToken对象wtoken，那么就可以从上到下遍历保存在它的成员变量windows所描述的一个ArrayList中的每一个WindowState对象win：

I. 如果WindowState对象win所描述的一个窗口具有子窗口，那么就继续从上到下遍历这些子窗口，即从上到下遍历WindowState对象win的成员变量mChildWindows所描述的一个ArrayList。如果能找到一个WindowState对象cwin，它的成员变量mSubLayer的值大于等于0，那么该WindowState对象cwin在WindowManagerService服务内部的WindowState堆栈中的位置就是我们要得到的位置i。注意，如果WindowState对象cwin的成员变量mSubLayer的值小于0，那么它虽然是一个子窗口，但是它却是位于父窗口的后面的，即它的Z轴位置是小于父窗口的Z轴位置的。

II. 如果WindowState对象win所描述的一个窗口不具有子窗口，即它的成员变量mChildWindows所描述的一个ArrayList的大小等于0，那么它在WindowManagerService服务内部的WindowState堆栈中的位置就是我们要得到的位置i。

得到了位置i之后，将它的值加1，那么就可以得到与参数tokenPos所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈的起始偏移位置了。

回到WindowManagerService类的成员函数moveAppToken中，调整好参数token所描述的AppWindowToken对象所对应的WindowState对象在WindowState堆栈中的位置之后，即调用了成员函数reAddAppWindowsLocked之后，这时候系统中的窗口的布局就会发生了变化，即系统中的窗口的Z轴位置关系发生了变化，那么接下来就需要调用成员函数updateFocusedWindowLocked来重新计算系统中的窗口的Z轴位置，并且调用成员函数performLayoutAndPlaceSurfacesLocked来重新布局系统中的窗口。

## 移动AppWindowToken至顶端

移动AppWindowToken至顶端是通过调用WindowManagerService类的成员函数moveAppTokensToTop来实现的，如下所示：

1. **public** **void** moveAppTokensToTop(List<IBinder> tokens) {
2. **if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,
3. "moveAppTokensToTop()")) {
4. **throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");
5. }
7. **final** **long** origId = Binder.clearCallingIdentity();
8. **synchronized**(mWindowMap) {
9. removeAppTokensLocked(tokens);
10. **final** **int** N = tokens.size();
11. **for** (**int** i=0; i<N; i++) {
12. AppWindowToken wt = findAppWindowToken(tokens.get(i));
13. **if** (wt != ) {
14. mAppTokens.add(wt);
15. **if** (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {
16. mToTopApps.remove(wt);
17. mToBottomApps.remove(wt);
18. mToTopApps.add(wt);
19. wt.sendingToBottom = **false**;
20. wt.sendingToTop = **true**;
21. }
22. }
23. }
25. **if** (mNextAppTransition == WindowManagerPolicy.TRANSIT\_UNSET) {
26. moveAppWindowsLocked(tokens, mAppTokens.size());
27. }
28. }
29. Binder.restoreCallingIdentity(origId);
30. }

WindowManagerService类的成员函数moveAppTokensToTop可以同时将一组AppWindowToken移至顶端，同时需要调用者具有android.Manifest.permission.MANAGE\_APP\_TOKENS权限。

参数tokens所描述的是一个IBinder接口列表，与这些IBinder接口所对应的AppWindowToken对象就是接下来要移至顶端的。在将保存在参数tokens中的IBinder接口所对应的AppWindowToken对象移至顶端之前，WindowManagerService类的成员函数首先会调用前面所描述的成员函数removeAppTokensLocked来删除这些AppWindowToken对象，然后再依次将它们添加到WindowManagerService类的成员变量mAppTokens所描述的一个ArrayList的末尾去。

注意，WindowManagerService类的成员变量mNextAppTransition用来描述系统当前是否正在切换Activity窗口。如果是的话，那么它的值就不等于WindowManagerPolicy.TRANSIT\_UNSET，这时候就需要：

A. 将所有要移至顶端的AppWindowToken对象都保存在WindowManagerService类的另外一个成员变量mToTopApps所描述的一个ArrayList中去，并且将这些AppWindowToken对象的成员变量sendingToTop的值设置为true。

B. 将所有要移至顶端的AppWindowToken对象所对应WindowState对象都移至WindowManagerService服务内部的一个WindowState堆栈的顶端去，这是通过调用另外一个成员函数moveAppWindowsLocked来实现的。

执行完成上述两个操作之后，与要移至顶端的AppWindowToken对象所对应的窗口就会位于窗口堆栈的最上面了。

## 移动AppWindowToken至底端

移动AppWindowToken至顶端是通过调用WindowManagerService类的成员函数moveAppTokensToBottom来实现的，如下所示：

1. **public** **void** moveAppTokensToBottom(List<IBinder> tokens) {
2. **if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,
3. "moveAppTokensToBottom()")) {
4. **throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");
5. }
7. **final** **long** origId = Binder.clearCallingIdentity();
8. **synchronized**(mWindowMap) {
9. removeAppTokensLocked(tokens);
10. **final** **int** N = tokens.size();
11. **int** pos = 0;
12. **for** (**int** i=0; i<N; i++) {
13. AppWindowToken wt = findAppWindowToken(tokens.get(i));
14. **if** (wt != ) {
15. mAppTokens.add(pos, wt);
16. **if** (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {
17. mToTopApps.remove(wt);
18. mToBottomApps.remove(wt);
19. mToBottomApps.add(i, wt);
20. wt.sendingToTop = **false**;
21. wt.sendingToBottom = **true**;
22. }
23. pos++;
24. }
25. }
27. **if** (mNextAppTransition == WindowManagerPolicy.TRANSIT\_UNSET) {
28. moveAppWindowsLocked(tokens, 0);
29. }
30. }
31. Binder.restoreCallingIdentity(origId);
32. }

WindowManagerService类的成员函数moveAppTokensToBottom可以同时将一组AppWindowToken移至底端。将一组AppWindowToken移至底端与将一组AppWindowToken移至顶端的实现是类似的，只不过是移动的方向相反而已。因此，WindowManagerService类的成员函数moveAppTokensToBottom的实现可以参考前面所分析的成员函数moveAppTokensToTop的实现，这里不再详述。

## 增加WindowToken

从图1可以知道，如果一个WindowState对象不是与一个AppWindowToken对象对应的，那么它就必须要与一个WindowToken对象对应。例如，用来描述输入法窗口和壁纸窗口的WindowState对象对应的就是WindowToken对象，而不是AppWindowToken对象，因为它们不是Activity类型的窗口。

输入法窗口和壁纸窗口分别是由输入法管理服务InputMethodManagerService和壁纸管理服务WallpaperManagerService调用WindowManagerService类的成员函数addWindowToken来增加对应的WindowToken对象的，如下所示：

1. **public** **void** addWindowToken(IBinder token, **int** type) {
2. **if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,
3. "addWindowToken()")) {
4. **throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");
5. }
7. **synchronized**(mWindowMap) {
8. WindowToken wtoken = mTokenMap.get(token);
9. **if** (wtoken != ) {
10. Slog.w(TAG, "Attempted to add existing input method token: " + token);
11. **return**;
12. }
13. wtoken = **new** WindowToken(token, type, **true**);
14. mTokenMap.put(token, wtoken);
15. mTokenList.add(wtoken);
16. **if** (type == TYPE\_WALLPAPER) {
17. mWallpaperTokens.add(wtoken);
18. }
19. }
20. }

调用WindowManagerService类的成员函数addWindowToken需要具有android.Manifest.permission.MANAGE\_APP\_TOKENS权限。

对于输入法窗口和壁纸窗口来说，参数token指向的是与它们所关联的一个Binder对象的IBinder接口，而参数type描述的是要在WindowManagerService服务内部增加WindowToken对象的窗口的类型。

WindowManagerService类的成员函数addWindowToken首先检查在成员变量mTokenMap所描述的一个HashMap检查是否已经存在一个WindowToken对象与参数token对应。如果已经存在的话，那么WindowManagerService类的成员函数addWindowToken就什么也不做就返回了，否则的话，就会使用参数token来创建一个WindowToken对象，并且会将该WindowToken对象分别保存在WindowManagerService类的成员变量mTokenMap和mTokenList中。

这里有两个地方需要注意：

A. 由于这里增加的是WindowToken对象，而不是AppWindowToken对象，因此，与增加AppWindowToken不同，这里不需要将新创建的WindowToken对象保存在WindowManagerService类的成员变量mAppTokens中。

B. 如果参数type的值等于TYPE\_WALLPAPER，那么就意味着新创建的WindowToken对象是用来描述壁纸窗口的，这时候还需要将新创建的WindowToken对象保存在WindowManagerService类的成员变量mWallpaperTokens所描述的一个ArrayList中，以方便管理壁纸窗口。

对于非输入法窗口、非壁纸窗口以及非Activity窗口来说，它们所对应的WindowToken对象是在它们增加到WindowManagerService服务的时候创建的。从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，增加一个窗口WindowManagerService服务最终是通过调用WindowManagerService类的成员函数addWindow来实现的，接下来我们就主要分析与创建WindowToken相关的逻辑，如下所示：

1. **public** **int** addWindow(Session session, IWindow client,
2. WindowManager.LayoutParams attrs, **int** viewVisibility,
3. Rect outContentInsets, InputChannel outInputChannel) {
4. ......
6. **synchronized**(mWindowMap) {
7. ......
9. **boolean** addToken = **false**;
10. WindowToken token = mTokenMap.get(attrs.token);
11. **if** (token == ) {
12. **if** (attrs.type >= FIRST\_APPLICATION\_WINDOW
13. && attrs.type <= LAST\_APPLICATION\_WINDOW) {
14. ......
15. **return** WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;
16. }
17. **if** (attrs.type == TYPE\_INPUT\_METHOD) {
18. ......
19. **return** WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;
20. }
21. **if** (attrs.type == TYPE\_WALLPAPER) {
22. ......
23. **return** WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;
24. }
25. token = **new** WindowToken(attrs.token, -1, **false**);
26. addToken = **true**;
27. }
29. ......
31. **if** (addToken) {
32. mTokenMap.put(attrs.token, token);
33. mTokenList.add(token);
34. }
36. ......
37. }
39. ......
40. }

如果参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量token所指向的一个IBinder接口在WindowManagerService类的成员变量mTokenMap所描述的一个HashMap中没有一个对应的WindowToken对象，并且该WindowManager.LayoutParams对象的成员变量type的值不等于TYPE\_INPUT\_METHOD、TYPE\_WALLPAPER，以及不在FIRST\_APPLICATION\_WINDOW和LAST\_APPLICATION\_WINDOW，那么就意味着这时候要增加的窗口就既不是输入法窗口，也不是壁纸窗口和Activity窗口，因此，就需要以参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量token所指向的一个IBinder接口为参数来创建一个WindowToken对象，并且将该WindowToken对象保存在WindowManagerService类的成员变量mTokenMap和mTokenList中。

## 删除WindowToken

删除WindowToken是通过调用WindowManagerService类的成员函数removeWindowToken来实现的，如下所示：

1. **public** **void** removeWindowToken(IBinder token) {
2. **if** (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,
3. "removeWindowToken()")) {
4. **throw** **new** SecurityException("Requires MANAGE\_APP\_TOKENS permission");
5. }
7. **final** **long** origId = Binder.clearCallingIdentity();
8. **synchronized**(mWindowMap) {
9. WindowToken wtoken = mTokenMap.remove(token);
10. mTokenList.remove(wtoken);
11. **if** (wtoken != ) {
12. **boolean** delayed = **false**;
13. **if** (!wtoken.hidden) {
14. wtoken.hidden = **true**;
16. **final** **int** N = wtoken.windows.size();
17. **boolean** changed = **false**;
19. **for** (**int** i=0; i<N; i++) {
20. WindowState win = wtoken.windows.get(i);
22. **if** (win.isAnimating()) {
23. delayed = **true**;
24. }
26. **if** (win.isVisibleNow()) {
27. applyAnimationLocked(win,
28. WindowManagerPolicy.TRANSIT\_EXIT, **false**);
29. changed = **true**;
30. }
31. }
33. **if** (changed) {
34. mLayoutNeeded = **true**;
35. performLayoutAndPlaceSurfacesLocked();
36. updateFocusedWindowLocked(UPDATE\_FOCUS\_NORMAL);
37. }
39. **if** (delayed) {
40. mExitingTokens.add(wtoken);
41. } **else** **if** (wtoken.windowType == TYPE\_WALLPAPER) {
42. mWallpaperTokens.remove(wtoken);
43. }
44. }
46. ......
47. } **else** {
48. Slog.w(TAG, "Attempted to remove non-existing token: " + token);
49. }
50. }
51. Binder.restoreCallingIdentity(origId);
52. }

调用WindowManagerService类的成员函数removeWindowToken需要具有android.Manifest.permission.MANAGE\_APP\_TOKENS权限。

WindowManagerService类的成员函数removeWindowToken首先找到与参数token所描述的Binder接口所对应的WindowToken对象，接着再将该WindowToken对象从WindowManagerService类的成员变量mTokenMap和mTokenList中删除。

删除了一个WindowToken对象之后，如果该WindowToken对象不是处于不可见的状态，即它的成员变量hidden的值不等于false，那么就意味着它所描述窗口口也有可能是可见的，那么WindowManagerService类的成员函数removeWindowToken就需要作以下两个检查：

A. 如果该WindowToken对象所描述的窗口的其中一个处于动画显示过程，即用来描述该窗口的一个WindowState对象的成员函数isAnimating的返回值等于true，那么就需要该WindowToken对象的状态设置为正在退出状态，即将它保存在WindowManagerService类的成员变量mExitingTokens所描述的一个ArrayList中。

B. 如果该WindowToken对象所描述的窗口是可见的，即用来描述该窗口的一个WindowState对象的成员函数isVisibleNow的返回值等于true，那么就需要调用WindowManagerService类的成员函数applyAnimationLocked来给它应用一个退出动画，该退出动画是通过调用WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked来实现的。当一个窗口退出了之后，系统当前获得焦点的窗口可能会发生变化，这时候就需要调用WindowManagerService类的成员函数updateFocusedWindowLocked来重新调整系统当前获得焦点的窗口。

注意，如果正在删除的WindowToken对象是用来描述壁纸窗口的，那么还需要将该WindowToken对象从WindowManagerService类的成员变量mWallpaperTokens所描述的一个ArrayList中删除。

* 1. 增加WindowState

从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，增加一个窗口WindowManagerService服务最终是通过调用WindowManagerService类的成员函数addWindow来实现的，如下所示：

1. /\*\*
2. \* Mapping from an IWindow IBinder to the server's Window object.
3. \* This is also used as the lock for all of our state.
4. \*/
5. **final** HashMap<IBinder, WindowState> mWindowMap = **new** HashMap<IBinder, WindowState>();
6. ......
8. /\*\*
9. \* Z-ordered (bottom-most first) list of all Window objects.
10. \*/
11. **final** ArrayList<WindowState> mWindows = **new** ArrayList<WindowState>();
12. ......
14. **public** **int** addWindow(Session session, IWindow client,
15. WindowManager.LayoutParams attrs, **int** viewVisibility,
16. Rect outContentInsets, InputChannel outInputChannel) {
17. ......
19. WindowState win = ;
21. **synchronized**(mWindowMap) {
22. ......
24. win = **new** WindowState(session, client, token,
25. attachedWindow, attrs, viewVisibility);
26. ......
28. mWindowMap.put(client.asBinder(), win);
29. ......
31. **if** (attrs.type == TYPE\_INPUT\_METHOD) {
32. mInputMethodWindow = win;
33. addInputMethodWindowToListLocked(win);
34. ......
35. } **else** **if** (attrs.type == TYPE\_INPUT\_METHOD\_DIALOG) {
36. mInputMethodDialogs.add(win);
37. addWindowToListInOrderLocked(win, **true**);
38. adjustInputMethodDialogsLocked();
39. ......
40. } **else** {
41. addWindowToListInOrderLocked(win, **true**);
42. **if** (attrs.type == TYPE\_WALLPAPER) {
43. .......
44. adjustWallpaperWindowsLocked();
45. } **else** **if** ((attrs.flags&FLAG\_SHOW\_WALLPAPER) != 0) {
46. adjustWallpaperWindowsLocked();
47. }
48. }
50. ......
51. }
53. ......
54. }

WindowManagerService类有两个成员变量mWindowMap和mWindows是用来保存系统中的WindowState对象。其中，成员变量mWindowMap指向的是一个HashMap，它的关键字是一个IBinder接口，一般这个IBinder接口指向的是一个Binder代理对象，引用了运行在应用程序进程这一侧的一个类型为W的Binder本地对象，用来描述一个窗口；成员变量mWindows指向的是一个ArrayList，保存在它里面的WindowState对象是按照其Z轴位置从小到大的顺序排列的。成员变量mWindowMap和mWindows的区别在于，前者给在定一个IBinder接口的情况下，可以快速找到与对应的WindowState对象，而后者用来从上到下或者下到上遍历系统的WindowState对象。由于系统中的WindowState对象是按照其Z轴位置从小到大的顺序排列在成员变量mWindows中的，因此，成员变量mWindows所指向的ArrayList就是我们在前面图1中所说的Window Stack。

理解了WindowManagerService类有两个成员变量mWindowMap和mWindows的作用之后，WindowManagerService类的成员函数addWindow增加一个WindowState对象的过程就容易理解了。

参数client是一个Binder代理对象，引用了运行在应用程序进程这一侧的一个类型为W的Binder本地对象，用来描述要增加到WindowManagerService服务中的一个窗口。WindowManagerService类的成员函数addWindow首先创建一个WindowState对象win，接着再以参数client所描述的一个Binder代理对象的IBinder接口为关键字，将WindowState对象win保存在WindowManagerService类的成员变量mWindowMap中，最后还会根据要增加到WindowManagerService服务中的窗口的类型来调用不同的成员函数将WindowState对象win增加到WindowManagerService类的成员变量mWindows中：

A. 如果要增加的是输入法窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_INPUT\_METHOD，那么就会调用成员函数addInputMethodWindowToListLocked来将WindowState对象win增加到WindowManagerService类的成员变量mWindows中去，并且会将WindowState对象win保存在WindowManagerService类的成员变量mInputMethodWindow中。

B. 如果要增加的是输入法对话框，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_INPUT\_METHOD\_DIALOG，那么就会调用成员函数addWindowToListInOrderLocked来将WindowState对象win增加到WindowManagerService类的成员变量mWindows中去，并且会将WindowState对象win保存在WindowManagerService类的成员变量mInputMethodDialogs中，以及调用成员函数adjustInputMethodDialogsLocked来调整刚才所添加的输入法窗口在窗口堆栈中的位置，使得它位于系统当前需要输入法窗口的窗口的上面。

C. 如果要增加的是壁纸窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_WALLPAPER，那么就会调用成员函数addWindowToListInOrderLocked来将WindowState对象win增加到WindowManagerService类的成员变量mWindows中去，并且会调用成员函数adjustWallpaperWindowsLocked来调整刚才所添加的壁纸窗口在窗口堆栈中的位置，使得它位于系统当前需要壁纸窗口的窗口的下面。

D . 如果要增加的既不是输入法窗口，也不是输入法对话框和壁纸窗口，那么就只会调用成员函数addWindowToListInOrderLocked来将WindowState对象win增加到WindowManagerService类的成员变量mWindows中去，但是如果要增加的窗口需要显示壁纸，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_SHOW\_WALLPAPER位等于1，那么还会继续调用成员函数adjustWallpaperWindowsLocked来调整系统中的壁纸窗口在窗口堆栈中的位置，使得它位于刚才所添加的窗口的下面。

在后面的两篇文章中，我们再详细分析WindowManagerService类的成员函数addInputMethodWindowToListLocked、adjustInputMethodDialogsLocked和adjustWallpaperWindowsLocked的实现，其中，前两者是与输入法窗口相关的，而后者是与壁纸窗口相关的。本文主要关注WindowManagerService类的成员函数addWindowToListInOrderLocked的实现，它会将一个指定的WindowState对象增加到窗口堆栈中的合适位置上去。

## 增加WindowState到窗口堆栈

从前面的分析可以知道，将一个WindowState对象增加到WMS服务内部中的窗口堆栈，即WMS类的成员变量mWindows，是通过调用WMS类的成员函数addWindowToListInOrderLocked来实现的。

WMS类的成员函数addWindowToListInOrderLocked的实现比较复杂，我们先列出它的框架，然后再详细分析它的实现，如下所示：

1. **private** **void** addWindowToListInOrderLocked(WindowState win, **boolean** addToToken) {
2. **final** IWindow client = win.mClient;
3. **final** WindowToken token = win.mToken;
4. **final** ArrayList<WindowState> localmWindows = mWindows;
6. **final** **int** N = localmWindows.size();
7. **final** WindowState attached = win.mAttachedWindow;
8. **int** i;
9. **if** (attached == ) {
10. //CASE 1：要增加的窗口win没有附加在其它窗口上
11. **int** tokenWindowsPos = token.windows.size();
12. **if** (token.appWindowToken != ) {
13. //CASE 1.1：要增加的窗口win是一个Activity窗口
14. **int** index = tokenWindowsPos-1;
15. **if** (index >= 0) {
16. //CASE 1.1.1：用来要增加的窗口win的令牌token已存在其它窗口
17. ......
18. } **else** {
19. //CASE 1.1.2：用来要增加的窗口win的令牌token尚未存在任何窗口
20. ......
21. }
22. } **else** {
23. //CASE 1.2：要增加的窗口win不是一个Activity窗口
24. ......
25. }
26. **if** (addToToken) {
27. token.windows.add(tokenWindowsPos, win);
28. }
29. } **else** {
30. //CASE 2：要增加的窗口win附加在窗口attached上
31. ......
32. }
34. **if** (win.mAppToken !=  && addToToken) {
35. win.mAppToken.allAppWindows.add(win);
36. }
37. }
39. ......

我们首先分析一下WMS类的成员函数addWindowToListInOrderLocked的几个本地变量的含义：

A. token。本地变量token指向的是参数win所描述的一个WindowState对象的成员变量mToken所指向一个WindowToken对象，这个WindowToken对象用来描述WindowState对象win所对应的窗口令牌。

B. localmWindows。本地变量localmWindows指向的是WMS类的成员变量mWindows所描述的一个ArrayList，即一个窗口堆栈，WMS类的成员函数addWindowToListInOrderLocked的目标就是要将参数win所描述的一个WindowState对象增加到该窗口堆栈的合适位置上去。

C. attached。本地变量attached指向的是参数win所描述的一个WindowState对象的成员变量mAttachedWindow 所指向的一个WindowState对象，如果它的值不等于null，那么就意味参数win所描述的窗口要附加在本地变量attached所描述的窗口上。

D. tokenWindowsPos。本地变量tokenWindowsPos用来描述与窗口令牌token所对应的窗口的数量。

E. token.appWindowToken。从前面Android应用程序窗口（Activity）与WMS服务的连接过程分析一文可以知道，如果一个WindowToken对象的成员变量appWindowToken的值不等于null，那么就意味着该WindowToken对象的实际类型为是AppWindowToken，即它所描述的是一个Activity窗口令牌，这种类型的令牌的特点是在ActivityManagerService服务的Activity组件堆栈中对应有一个ActivityRecord对象，如图1所示。

F. index。本地变量index的值等于tokenWindowsPos-1，如果它的值大于等于0，那么就意味着窗口令牌tokent已经存在其它窗口，否则的话，就意味着窗口令牌tokent尚未存在任何窗口。

从这些本地变量的含义，我们就可以分情况来将参数win所描述的一个WindowState对象增加到WMS服务内部的窗口堆栈的合适位置上去：

CASE 1：要增加的窗口win没有附加在其它窗口上

----CASE 1.1：要增加的窗口win是一个Activity窗口

----CASE 1.1.1：用来要增加的窗口win的令牌token已存在其它窗口。这时候意味着窗口win需要保存在其它已经存在的窗口的附近，因此，我们只要找到这些已经存在的窗口在窗口堆栈中的位置，那么再根据其它属性，就可以将窗口win保存在已经存在的窗口的上面或者下面。

----CASE 1.1.2：用来要增加的窗口win的令牌token尚未存在任何窗口。虽然这时候窗口win在窗口堆栈中没有位置可以参考，但是它毕竟是一个Activity窗口，我们可以通过与它所对应的AppWindowToken对象在App Token List（即WMS类的成员变量mAppTokens所描述的一个ArrayList）中的位置来获得它窗口堆栈中的位置。回忆我们在前面第3节分析移动AppWindowToken至指定位置的操作时得到的结论：WMS服务内部中的所有WindowState对象都是按照Z轴从位置从小到大排列在WindowState堆栈中的，并且在mAppTokens列表中，位于上面的一个AppWindowToken对象所对应的那些WindowState对象的Z轴位置是一定大于位于下面的一个AppWindowToken对象所对应的那些WindowState对象的Z轴位置的。因此，我们只要找到用来描述窗口win的一个AppWindowToken对象（token.appWindowToken）的上一个或者下一个AppWindowToken对象所对应的窗口在窗口堆栈中的位置，那么就可以这个位置为参考，得到窗口win在窗口堆栈中的位置。

----CASE 1.2：要增加的窗口win不是一个Activity窗口。这时候既然要增加的窗口也没有附加在其它窗口上，那么就意味着要增加的窗口win在窗口堆栈中没有位置可以参考，因此，我们就需要根据它的Z轴位置来决定它在窗口堆栈的位置。

CASE 2：要增加的窗口win附加在窗口attached上。这时候就意味着要增加的窗口win要保存在窗口attached的上面，即窗口在窗口堆栈的位置要以窗口attached在窗口堆栈的位置为参考。

从上面的分析就可以知道，CASE 1.1.1、CASE 1.1.2和CASE 2都有一个共同特点，即要增加的窗口win在窗口堆栈的位置有一个参考值，而在CASE 1.2中，要增加的窗口win在窗口堆栈的位置没有参考值，需要通过其Z轴位置来确定。

在分析上述四种情况之前， 我们还需要再说明一下WMS类的成员函数addWindowToListInOrderLocked的参数addToToken的含义。参数addToToken是一个布尔变量，如果它的值等于true，那么就说明需要将参数win所描述的一个WindowState对象添加用来描述它的窗口令牌token的成员变量windows所描述的一个ArrayList中去。注意，窗口令牌token的成员变量windows所描述的一个ArrayList里面所保存的WindowState对象是按照Z轴位置从小到大的顺序来排列的，因此，在将WindowState对象win保存到这个ArrayList之前，首先要按照它的Z轴位置计算得到它在这个ArrayList中的位置tokenWindowsPos。另一方面，在参数addToToken的值等于true，并且参数win所描述的是一个Activity窗口，即它的成员变量mAppToken不等于null的情况下，还需要将参数win所描述的一个WindowState对象保存在用来描述它的窗口令牌，即一个AppWindowToken对象成员变量allAppWindows所描述的一个ArrayList中去，以便可以知道一个AppWindowToken对象对应的Activity窗口都有哪些。

接下来，我们就分别分析这四种情况是如何将窗口win增加窗口堆栈中去的。

CASE 1.1.1对应的代码为：

1. **if** (win.mAttrs.type == TYPE\_BASE\_APPLICATION) {
2. // Base windows go behind everything else.
3. placeWindowBefore(token.windows.get(0), win);
4. tokenWindowsPos = 0;
5. } **else** {
6. AppWindowToken atoken = win.mAppToken;
7. **if** (atoken !=  &&
8. token.windows.get(index) == atoken.startingWindow) {
9. placeWindowBefore(token.windows.get(index), win);
10. tokenWindowsPos--;
11. } **else** {
12. **int** newIdx =  findIdxBasedOnAppTokens(win);
13. **if**(newIdx != -1) {
14. //there is a window above this one associated with the same
15. //apptoken note that the window could be a floating window
16. //that was created later or a window at the top of the list of
17. //windows associated with this token.
18. ......
19. localmWindows.add(newIdx+1, win);
20. mWindowsChanged = **true**;
21. }
22. }
23. }

这段代码又分为三种情况来将参数win所描述的一个WindowState对象添加到窗口堆栈中：

A. 参数win描述的窗口的类型为TYPE\_BASE\_APPLICATION。在一个令牌对应的所有窗口中，类型为TYPE\_BASE\_APPLICATION的窗口位于其它类型的窗口的下面。因此，这段代码就会调用WMS类的成员函数placeWindowBefore来将参数win所描述的一个WindowState对象保存窗口堆栈中，并且它是位于令牌token的窗口列表的第0个位置的WindowState对象的下面。这时候变量tokenWindowsPos的值会被设置为0，表示参数win所描述的一个WindowState对象要保存窗口令牌token的窗口列表的第0个位置上。

B. 参数win描述的一个WindowState对象的成员变量mAppToken的值不等于null，这意味着参数win描述的是一个Activity窗口，这时候如果窗口令牌atoken（与token描述的是同一个窗口令牌）的窗口列表的第index个位置（即最上面的一个位置） 的WindowState对象描述的是一个Activity启动窗口，即与窗口令牌atoken的成员变量startingWindow描述的是同一个窗口，那么就说明窗口令牌atoken的窗口列表的第index个位置的WindowState对象描述的是窗口win的启动窗口。由于一个窗口的启动窗口总是位于它的上面，因此，这段代码就会调用WMS类的成员函数placeWindowBefore来将参数win所描述的一个WindowState对象保存窗口堆栈中，并且它是位于令牌atoken的窗口列表的第index个位置的WindowState对象的下面。这时候变量tokenWindowsPos的值减少1，即相当于是等于index，表示参数win所描述的一个WindowState对象要插入在窗口令牌token的窗口列表的第index个位置上。

C. 参数win所描述的窗口的类型既不是TYPE\_BASE\_APPLICATION，而且它也没有启动窗口，那么这时候就需要将它保存在窗口令牌token的窗口列表的最上面一个窗口的上面。窗口令牌token的窗口列表的最上面一个窗口在窗口堆栈中的位置newIdx是通过调用WMS类的成员函数findIdxBaseOnAppTokens来获得的，这时候参数win所描述的一个WindowState对象就应该保存在窗口堆栈，即变量localmWindows所描述的一个ArrayList的第newIdx+1个位置上。

CASE 1.1.2对应的代码为：

1. // Figure out where the window should go, based on the
2. // order of applications.
3. **final** **int** NA = mAppTokens.size();
4. WindowState pos = ;
5. **for** (i=NA-1; i>=0; i--) {
6. AppWindowToken t = mAppTokens.get(i);
7. **if** (t == token) {
8. i--;
9. **break**;
10. }
12. // We haven't reached the token yet; if this token
13. // is not going to the bottom and has windows, we can
14. // use it as an anchor for when we do reach the token.
15. **if** (!t.sendingToBottom && t.windows.size() > 0) {
16. pos = t.windows.get(0);
17. }
18. }
19. // We now know the index into the apps.  If we found
20. // an app window above, that gives us the position; else
21. // we need to look some more.
22. **if** (pos != ) {
23. // Move behind any windows attached to this one.
24. WindowToken atoken = mTokenMap.get(pos.mClient.asBinder());
25. **if** (atoken != ) {
26. **final** **int** NC = atoken.windows.size();
27. **if** (NC > 0) {
28. WindowState bottom = atoken.windows.get(0);
29. **if** (bottom.mSubLayer < 0) {
30. pos = bottom;
31. }
32. }
33. }
34. placeWindowBefore(pos, win);
35. } **else** {
36. // Continue looking down until we find the first
37. // token that has windows.
38. **while** (i >= 0) {
39. AppWindowToken t = mAppTokens.get(i);
40. **final** **int** NW = t.windows.size();
41. **if** (NW > 0) {
42. pos = t.windows.get(NW-1);
43. **break**;
44. }
45. i--;
46. }
47. **if** (pos != ) {
48. // Move in front of any windows attached to this
49. // one.
50. WindowToken atoken = mTokenMap.get(pos.mClient.asBinder());
51. **if** (atoken != ) {
52. **final** **int** NC = atoken.windows.size();
53. **if** (NC > 0) {
54. WindowState top = atoken.windows.get(NC-1);
55. **if** (top.mSubLayer >= 0) {
56. pos = top;
57. }
58. }
59. }
60. placeWindowAfter(pos, win);
61. placeWindowAfter(pos, win);
62. } **else** {
63. // Just search for the start of this layer.
64. **final** **int** myLayer = win.mBaseLayer;
65. **for** (i=0; i<N; i++) {
66. WindowState w = localmWindows.get(i);
67. **if** (w.mBaseLayer > myLayer) {
68. **break**;
69. }
70. }
71. ......
72. localmWindows.add(i, win);
73. mWindowsChanged = **true**;
74. }
75. }

这段代码要能冠军WMS服务内部的一个AppWindowToken列表mAppTokens来在窗口堆栈中找到一个参数位置来保存参数win所描述的一个WindowState对象。

最上面的一个for循环执行完成之后，我们假设变量pos的值不等于null，这时候它与变量i以及变量token的关系如图2所示：



图2 窗口win位于窗口C的下面

这时候位于令牌token上面的令牌在窗口堆栈中对应有WindowState对象。注意，这时候第i+2个令牌在窗口堆栈中不对应有WindowState对象，而第i+3个令牌在窗口堆栈中对应有C和D两个WindowState对象，并且这两个WindowState对象所描述的窗口都不是即将要切换到窗口堆栈的底部的。由于第i+3个令牌位于令牌token的上面，并且这两个令牌之间的其它令牌在窗口堆栈中不对应有WindowState对象，因此，这时候参数win所描述的WindowState对象在窗口堆栈中的位置应该以第i+3个令牌所对应的Z轴位置最小的WindowState对象在窗口堆栈中的位置为参考，即以WindowState对象C在窗口堆栈中的位置为参考，而WindowState对象C也正好是变量pos所指向的WindowState对象。

接下来，上述代码会继续检查WindowState对象C是否附加有SubLayer值小于0的窗口。如果有的话，那么就会将变量pos指向SubLayer值最小的那个WindowState对象，这是因为该WindowState对象是在WindowState对象C的最下面的，并且它与WindowState对象C是同属一个令牌的。最后，上述代码就会调用WMS类的成员函数placeWindowBefore来将参数win所描述的一个WindowState对象保存窗口堆栈中由变量pos所指向的那个WindowState对象的下面。

假设最上面的一个for循环执行完成之后，变量pos的值等于null，那么就说明位于令牌token上面的令牌在窗口堆栈中都没有对应有WindowState对象，或者说它们所对应的WindowState对象都是即将要切换到窗口堆栈的底部去的，这时候就需要通过位于令牌token上面的令牌来在窗口堆栈中找到一个参考位置来保存参数win所描述的WindowState对象，这是通过中间的while循环来实现的。

中间的while循环执行完成之后，假设变量pos的值不等于null，这时候它与变量i以及变量token的关系如图3所示：



图3 窗口win位于窗口D的上面

这时候位于令牌token上面的令牌在窗口堆栈中没有对应有WindowState对象。注意，这时候第i-1个令牌在窗口堆栈中不对应有WindowState对象，而第i-2个令牌在窗口堆栈中对应有C和D两个WindowState对象。由于第i-2个令牌位于令牌token的下面，并且这两个令牌之间的其它令牌在窗口堆栈中不对应有WindowState对象，因此，这时候参数win所描述的WindowState对象在窗口堆栈中的位置应该以第i-2个令牌所对应的Z轴位置最大的WindowState对象在窗口堆栈中的位置为参考，即以WindowState对象D在窗口堆栈中的位置为参考，而WindowState对象D也正好是变量pos所指向的WindowState对象。

接下来，上述代码会继续检查WindowState对象D是否附加有SubLayer值大于等于0的窗口。如果有的话，那么就会将变量pos指向SubLayer值最大的那个WindowState对象，这是因为该WindowState对象是在WindowState对象D的最上面的，并且它与WindowState对象D是同属一个令牌的。最后，上述代码就会调用WMS类的成员函数placeWindowAfter来将参数win所描述的一个WindowState对象保存窗口堆栈中由变量pos所指向的那个WindowState对象的上面。

假设中间的while循环执行完成之后，变量pos的值等于null，这时候就说明在窗口堆栈中实在是找不到参考位置来保存参数win所描述的WindowState对象了，因此，就只能通过参数win所描述的WindowState对象的Z轴位置，即它的成员变量mBaseLayer的值来在窗口堆栈中找到一个合适的位置了，如最下面的for循环所示。由于窗口堆栈中的WindowState对象是按照它们的Z轴位置由小到大的顺序来排列的，因此，最下面的for循环只要从下到上找到一个Z轴位置比参数win所描述的WindowState对象的Z轴位置大的一个WindowState对象在窗口堆栈中的位置i，那么就可以将参数win所描述的WindowState对象插入在窗口堆栈的第i个位置上了。

CASE 1.2对应的代码为：

1. // Figure out where window should go, based on layer.
2. **final** **int** myLayer = win.mBaseLayer;
3. **for** (i=N-1; i>=0; i--) {
4. **if** (localmWindows.get(i).mBaseLayer <= myLayer) {
5. i++;
6. **break**;
7. }
8. }
9. **if** (i < 0) i = 0;
10. ......
11. localmWindows.add(i, win);
12. mWindowsChanged = **true**;

由于这时候在窗口堆栈中是没有参考位置来保存参数win所描述的WindowState对象的，因此，这段代码就只能通过参数win所描述的WindowState对象的Z轴位置，即它的成员变量mBaseLayer的值来在窗口堆栈中找到一个合适的位置了，如这段代码中的for循环所示。由于窗口堆栈中的WindowState对象是按照它们的Z轴位置由小到大的顺序来排列的，因此，这段代码中的for循环只要从上到下找到一个WindowState对象，它的Z轴位置小于或者等于参数win所描述的WindowState对象的Z轴位置，那么该WindowState对象在窗口堆栈中的位置i就可以用插入参数win所描述的WindowState对象了。

CASE 2对应的代码为：

1. // Figure out this window's ordering relative to the window
2. // it is attached to.
3. **final** **int** NA = token.windows.size();
4. **final** **int** sublayer = win.mSubLayer;
5. **int** largestSublayer = Integer.MIN\_VALUE;
6. WindowState windowWithLargestSublayer = ;
7. **for** (i=0; i<NA; i++) {
8. WindowState w = token.windows.get(i);
9. **final** **int** wSublayer = w.mSubLayer;
10. **if** (wSublayer >= largestSublayer) {
11. largestSublayer = wSublayer;
12. windowWithLargestSublayer = w;
13. }
14. **if** (sublayer < 0) {
15. // For negative sublayers, we go below all windows
16. // in the same sublayer.
17. **if** (wSublayer >= sublayer) {
18. **if** (addToToken) {
19. token.windows.add(i, win);
20. }
21. placeWindowBefore(
22. wSublayer >= 0 ? attached : w, win);
23. **break**;
24. }
25. } **else** {
26. // For positive sublayers, we go above all windows
27. // in the same sublayer.
28. **if** (wSublayer > sublayer) {
29. **if** (addToToken) {
30. token.windows.add(i, win);
31. }
32. placeWindowBefore(w, win);
33. **break**;
34. }
35. }
36. }
37. **if** (i >= NA) {
38. **if** (addToToken) {
39. token.windows.add(win);
40. }
41. **if** (sublayer < 0) {
42. placeWindowBefore(attached, win);
43. } **else** {
44. placeWindowAfter(largestSublayer >= 0
45. ? windowWithLargestSublayer
46. : attached,
47. win);
48. }
49. }

这段代码要将参数win所描述的WindowState对象附加在变量attached所描述的WindowState对象的上面或者下面，取决于它的成员变量mSubLayer的值是大于0还是小于0。我们分四种情况来考虑。

第一种情况是参数win所描述的WindowState对象的成员变量mSubLayer的值小于0，并且这时候在附加在窗口attached的WindowState对象中，存在一个WindowState对象，它的成员变量mSubLayer的值大于等于参数win所描述的WindowState对象的成员变量mSubLayer的值，如图4和图5所示：



图4 窗口win插入到窗口B的下面



图5 窗口win插入在窗口attached的下面

在图4和图5中，WindowState对象A和B均是附加在WindowState对象attached中。

在图4中，WindowState对象A和B的成员变量mSubLayer的值均小于0，而WindowState对象win的成员变量mSubLayer的值比WindowState对象A的大，但是比WindowState对象B的小，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象B的下面，这是通过调用WMS类的成员函数placeWindowBefore来实现的。

在图5中，WindowState对象A和B的成员变量mSubLayer的值均大于0，由于WindowState对象win的成员变量mSubLayer的值小于0，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象attached的下面，这是通过调用WMS类的成员函数placeWindowBefore来实现的。

第二种情况是参数win所描述的WindowState对象的成员变量mSubLayer的值大于0，并且这时候在附加在窗口attached的WindowState对象中，存在一个WindowState对象，它的成员变量mSubLayer的值大于参数win所描述的WindowState对象的成员变量mSubLayer的值，如图6所示：



图6 窗口win插入在窗口B的下面

在图6中，WindowState对象A和B均是附加在WindowState对象attached中。其中，WindowState对象A和B的成员变量mSubLayer的值均大于0，而WindowState对象win的成员变量mSubLayer的值比WindowState对象A的大，但是比WindowState对象B的小，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象B的下面，这是通过调用WMS类的成员函数placeWindowBefore来实现的。

第三种情况是参数win所描述的WindowState对象的成员变量mSubLayer的值小于0，但是在附加在窗口attached的WindowState对象中，找不到一个WindowState对象，它的成员变量mSubLayer的值比WindowState对象的成员变量mSubLayer的值大，如图7所示：



图7 窗口win插入在窗口attached的下面

        在图7中，WindowState对象A和B均是附加在WindowState对象attached中。其中，WindowState对象A和B以及win的成员变量mSubLayer的值均小于0，但是WindowState对象win的成员变量mSubLayer的值比WindowState对象A和B的都要大，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象attached的下面，这是通过调用WMS类的成员函数placeWindowBefore来实现的。

        第四种情况是参数win所描述的WindowState对象的成员变量mSubLayer的值大于等于0，但是在附加在窗口attached的WindowState对象中，找不到一个WindowState对象，它的成员变量mSubLayer的值比WindowState对象的成员变量mSubLayer的值大，如图8和图9所示：



图8 窗口win插入在窗口B的上面



图9 窗口win插入在窗口attached的上面

图9 窗口win插入在窗口attached的上面

        在图8和图9中，WindowState对象A和B均是附加在WindowState对象attached中。

        在图8中，WindowState对象A和B的成员变量mSubLayer的值均大于0，并且WindowState对象win的成员变量mSubLayer的值比WindowState对象A和B的都要大，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象B的上面，这是通过调用WMS类的成员函数placeWindowAfter来实现的。

        在图9中，WindowState对象A和B的成员变量mSubLayer的值均小于等于0，而WindowState对象win的成员变量mSubLayer的值大于0，这时候WindowState对象win在窗口堆栈中就应该位于WindowState对象attached的上面，这是通过调用WMS类的成员函数placeWindowAfter来实现的。

         注意，在这四种情况中，如果参数addToToken的值等于true，那么都需要将参数win所描述的WindowState对象增加到与它所对应的窗口令牌token的窗口列表windows中去。

         10. 删除WindowState

          删除WindowState是通过调用WMS类的成员函数tmpRemoveWindowLocked来实现的，如下所示：

**[java]** [view plain](http://blog.csdn.net/luoshengyang/article/details/8498908) [copy](http://blog.csdn.net/luoshengyang/article/details/8498908)

1. **public** **class** WMS **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **int** tmpRemoveWindowLocked(**int** interestingPos, WindowState win) {
6. **int** wpos = mWindows.indexOf(win);
7. **if** (wpos >= 0) {
8. **if** (wpos < interestingPos) interestingPos--;
9. ......
10. mWindows.remove(wpos);
11. mWindowsChanged = **true**;
12. **int** NC = win.mChildWindows.size();
13. **while** (NC > 0) {
14. NC--;
15. WindowState cw = win.mChildWindows.get(NC);
16. **int** cpos = mWindows.indexOf(cw);
17. **if** (cpos >= 0) {
18. **if** (cpos < interestingPos) interestingPos--;
19. ......
20. mWindows.remove(cpos);
21. }
22. }
23. }
24. **return** interestingPos;
25. }
27. ......
28. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WMS.java中。

        WMS类的成员函数tmpRemoveWindowLocked将参数win所描述的窗口及其子窗口从WMS服务内部的窗口堆栈中删除，即从 WMS类的成员变量mWindows所描述的一个ArrayList中删除。

        如果每一个被删除的窗口在窗口堆栈中的位置比参数interestingPos的值小，那么WMS类的成员函数tmpRemoveWindowLocked还会将参数interestingPos的值减少1，这相当于是计算当删除参数win所描述的窗口及其子窗口之后，原来位于窗口堆栈中第interestingPos个位置的窗口现在位于窗口堆栈的位置，这个位置最终会作为WMS类的成员函数tmpRemoveWindowLocked的返回值。

       11. 在指定位置增加WindowState

       在指定位置增加WindowState是通过调用WMS类的成员函数reAddWindowLocked来实现的，如下所示：

**[java]** [view plain](http://blog.csdn.net/luoshengyang/article/details/8498908) [copy](http://blog.csdn.net/luoshengyang/article/details/8498908)

1. **public** **class** WMS **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **final** **int** reAddWindowLocked(**int** index, WindowState win) {
6. **final** **int** NCW = win.mChildWindows.size();
7. **boolean** added = **false**;
8. **for** (**int** j=0; j<NCW; j++) {
9. WindowState cwin = win.mChildWindows.get(j);
10. **if** (!added && cwin.mSubLayer >= 0) {
11. ......
12. mWindows.add(index, win);
13. index++;
14. added = **true**;
15. }
16. ......
17. mWindows.add(index, cwin);
18. index++;
19. }
20. **if** (!added) {
21. ......
22. mWindows.add(index, win);
23. index++;
24. }
25. mWindowsChanged = **true**;
26. **return** index;
27. }
29. ......
30. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WMS.java中。

        参数win描述的即为要增加的WindowState对象，而参数index描述的即为要将参数win所描述的WindowState对象及其子WindowState对象要增加到窗口堆栈中的起始位置。

       由于参数win所描述的WindowState对象的子WindowState对象的成员变量mSubLayer的值可能会小于0，也可能大于0。大于0的子WindowState对象位于参数win所描述的WindowState对象的上面，而小于0的子WindowState对象位于参数win所描述的WindowState对象的下面。因此，WMS类的成员函数reAddWindowLocked先增加那些小于0的子WindowState对象，接着再增加参数win所描述的WindowState对象，最后增加那些大于0的子WindowState对象。

        假设WMS类的成员函数reAddWindowLocked一共在窗口堆栈中增加了N个WindowState对象，那么它的返回值就等于index + N，这样调用者就可以知道参数win所描述的WindowState对象及其子WindowState对象在窗口堆栈中的最高位置是多少。

        基于第9、第10和第11这三操作，可以组合成很多其它的WindowState操作，如接下来的第12、第13、第14和第15个操作所示。

        12. 将一个WindowState对象及其所有子WindowState对象增加到窗口堆栈中

         将一个WindowState对象及其所有子WindowState对象增加到窗口堆栈中是通过调用WMS类的成员函数reAddWindowToListInOrderLocked来实现的，如下所示：

**[java]** [view plain](http://blog.csdn.net/luoshengyang/article/details/8498908) [copy](http://blog.csdn.net/luoshengyang/article/details/8498908)

1. **public** **class** WMS **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **void** reAddWindowToListInOrderLocked(WindowState win) {
6. addWindowToListInOrderLocked(win, **false**);
7. // This is a hack to get all of the child windows added as well
8. // at the right position.  Child windows should be rare and
9. // this case should be rare, so it shouldn't be that big a deal.
10. **int** wpos = mWindows.indexOf(win);
11. **if** (wpos >= 0) {
12. ......
13. mWindows.remove(wpos);
14. mWindowsChanged = **true**;
15. reAddWindowLocked(wpos, win);
16. }
17. }
19. ......
20. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WMS.java中。

        为了得到参数win所描述的WindowState对象的子WindowState对象在窗口堆栈中的起始位置，WMS类的成员函数reAddWindowToListInOrderLocked首先将参数win所描述的WindowState对象增加到窗口堆栈中，这是通过调用前面所分析的成员函数addWindowToListInOrderLocked来实现的，目的是为了获得它在窗口堆栈的位置。有了这个位置之后，WMS类的成员函数reAddWindowToListInOrderLocked就可以调用前面所分析的成员函数reAddWindowLocked来将WindowState对象及其所有子WindowState对象增加到窗口堆栈中去了，不过在调用之前，要先将参数win所描述的WindowState对象从窗口中堆栈删除。

        13. 将一个WindowToken对象对应的所有WindowState对象及其子WindowState对象增加到窗口堆栈的指定位置上

         将一个WindowToken对象对应的所有WindowState对象都增加到窗口堆栈中是通过调用WMS类的成员函数reAddAppWindowsLocked来实现的，如下所示：

**[java]** [view plain](http://blog.csdn.net/luoshengyang/article/details/8498908) [copy](http://blog.csdn.net/luoshengyang/article/details/8498908)

1. **public** **class** WMS **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **final** **int** reAddAppWindowsLocked(**int** index, WindowToken token) {
6. **final** **int** NW = token.windows.size();
7. **for** (**int** i=0; i<NW; i++) {
8. index = reAddWindowLocked(index, token.windows.get(i));
9. }
10. **return** index;
11. }
13. ......
14. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WMS.java中。

        与参数token所描述的WindowToken对象所对应的WindowState对象保存在它的成员变量windows所描述的一个ArrayList中。通过遍历这个ArrayList，就可以将与参数token所描述的WindowToken对象所对应的WindowState对象及其子WindowState对象都增加到窗口堆栈的指定的起始位置上去，这是通过调用前面所分析的成员函数reAddWindowLocked来实现的。

        参数index描述的便是最初指定的起始位置，每一次调用WMS类的成员函数reAddWindowLocked之后，它的值都便会被更新为下一个WindowState对象及其子WindowState对象要增加到窗口堆栈中的位置。

        最后，WMS类的成员函数reAddAppWindowsLocked将与参数token所描述的WindowToken对象所对应的WindowState对象在窗口堆栈中的最高位置加1后的得到结果返回给调用者。

       14. 将一个AppWindowToken对象所对应的WindowState对象及其子 WindowState对象移动到窗口堆栈的指定位置上

        将一个AppWindowToken对象所对应的WindowState对象及其子 WindowState对象移动到窗口堆栈的指定位置上是通过调用WMS类的成员函数moveAppWindowsLocked来实现的，如下所示：

**[java]** [view plain](http://blog.csdn.net/luoshengyang/article/details/8498908) [copy](http://blog.csdn.net/luoshengyang/article/details/8498908)

1. **public** **class** WMS **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **void** moveAppWindowsLocked(AppWindowToken wtoken, **int** tokenPos,
6. **boolean** updateFocusAndLayout) {
7. // First remove all of the windows from the list.
8. tmpRemoveAppWindowsLocked(wtoken);
10. // Where to start adding?
11. **int** pos = findWindowOffsetLocked(tokenPos);
13. // And now add them back at the correct place.
14. pos = reAddAppWindowsLocked(pos, wtoken);
16. **if** (updateFocusAndLayout) {
17. **if** (!updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES)) {
18. assignLayersLocked();
19. }
20. mLayoutNeeded = **true**;
21. performLayoutAndPlaceSurfacesLocked();
22. }
23. }
25. ......
26. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WMS.java中。

        参数wtoken描述的是要移动其所对应的WindowState对象的一个AppWindowToken对象，而参数tokenPos描述的是该AppWindowToken对象在WMS服务内部的AppWindowToken列表中的新位置。

        WMS类的成员函数moveAppWindowsLocked首先调用前面所分析的成员函数tmpRemoveAppWindowsLocked来移除所有与参数wtoken所描述的AppWindowToken对象所对应的WindowState对象，接着再调用也是前面所分析的成员函数findWindowOffsetLocked来获得与参数wtoken所描述的AppWindowToken对象所对应的WindowState对象在窗口堆栈中的起始位置。有了这个起始位置之后，就可以也是前面所分析的成员函数reAddAppWindowsLocked来将与参数wtoken所描述的AppWindowToken对象所对应的WindowState对象及其子WindowState对象移动到窗口堆栈上去了。

        最后，如果参数updateFocusAndLayout的值等于true，那么WMS类的成员函数moveAppWindowsLocked还会更新系统当前获得焦点的窗口，以及重新计算系统中的所有窗口的Z轴位置以及重新布局系统中的所有窗口，这三个操作分别是通过调用WMS类的成员函数updateFocusedWindowLocked、assignLayersLocked和performLayoutAndPlaceSurfacesLocked来实现的。

        15. 将一组AppWindowToken对象所对应的WindowState对象及其子 WindowState对象移动到窗口堆栈的指定位置上

         将一组AppWindowToken对象所对应的WindowState对象及其子WindowState对象移动到窗口堆栈的指定位置上是通过调用WMS类的另外一个版本的成员函数moveAppWindowsLocked来实现的，如下所示：

**[java]** [view plain](http://blog.csdn.net/luoshengyang/article/details/8498908) [copy](http://blog.csdn.net/luoshengyang/article/details/8498908)

1. **public** **class** WMS **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **void** moveAppWindowsLocked(List<IBinder> tokens, **int** tokenPos) {
6. // First remove all of the windows from the list.
7. **final** **int** N = tokens.size();
8. **int** i;
9. **for** (i=0; i<N; i++) {
10. WindowToken token = mTokenMap.get(tokens.get(i));
11. **if** (token != ) {
12. tmpRemoveAppWindowsLocked(token);
13. }
14. }
16. // Where to start adding?
17. **int** pos = findWindowOffsetLocked(tokenPos);
19. // And now add them back at the correct place.
20. **for** (i=0; i<N; i++) {
21. WindowToken token = mTokenMap.get(tokens.get(i));
22. **if** (token != ) {
23. pos = reAddAppWindowsLocked(pos, token);
24. }
25. }
27. **if** (!updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES)) {
28. assignLayersLocked();
29. }
30. mLayoutNeeded = **true**;
31. performLayoutAndPlaceSurfacesLocked();
33. //dump();
34. }
36. ......
37. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WMS.java中。

        这个操作与前面分析的第14个操作是类似，区别只在于前者是批量地移动一组AppWindowToken对象所对应的WindowState对象及其子 WindowState对象，而后者是只移动一个AppWindowToken对象所对应的WindowState对象及其子WindowState对象，此外，前者总是会调用WMS类的成员函数updateFocusedWindowLocked、assignLayersLocked和performLayoutAndPlaceSurfacesLocked来更新系统当前获得焦点的窗口、以及重新计算每一个窗口的Z轴位置，并且对这些窗口进行重新布局。

        至此，我们就分析完成WMS服务组织系统中的窗口的方式了。从分析的过程中，可以得到以下结论：

        1. WMS服务维护有一个AppWindowToken堆栈和一个WindowState堆栈，它们与ActivityManagerService服务维护的Actvity堆栈是有关相同的Z轴位置关系的。

        2. ActivityManagerService服务中的每一个ActivityRecord对象在WMS服务中都对应有一个AppWindowToken对象，而WMS服务中的每一个AppWindowToken对象都对应有一组WindowState对象。

        3. 在WindowState堆栈中，AppWindowToken堆栈中的第i+1个AppWindowToken对象所对应的WindowState对象都位于第i个AppWindowToken对象所对应的WindowState对象的上面。

        4. 一个WindowState对象可以附加在另外一个WindowState对象上面，此外，一个WindowState对象还可以有子WindowState对象，它们都是与同一个AppWindowToken对象或者WindowToken对象所对应的。

        5. WMS服务有两个特殊的WindowToken，它们分别用来描述系统中的输入法窗口令牌和壁纸窗口令牌，其中，输入法窗口位于需要输入法的窗口的上面，而壁纸窗口位于需要壁纸的窗口的下面。

        最后，我们可以将WMS服务中的AppWindowToken理解成一个Activity组件令牌，而将它所对应的WindowState对象理解成一个Activity窗口。有了这些概念之后，就为学习WMS服务的各种实现打下坚实的基础。在接下来的两篇文章中，我们就会在本文的基础上，继续分析WMS服务是如何管理系统中的输入法窗口和壁纸窗口的，敬请关注！

## 实例分析：显示一个系统dlg流

new WindowState()

## REF

[Android窗口管理服务WMS对窗口的组织方式分析](http://blog.csdn.net/luoshengyang/article/details/8498908)

<http://www.jianshu.com/p/40776c123adb>

<http://gityuan.com/2017/01/08/windowmanger/>

## REF

[Android设计模式（二） 续：WindowManager](http://www.jianshu.com/p/3e9f068ed82d)

Dlg:

<http://blog.csdn.net/yanbober/article/details/46361191>

<http://blog.csdn.net/hwliu51/article/details/75040297>

http://www.jianshu.com/p/672e6486a72a