这片文章将详细分析WindowState中的各个关键成员变量及成员函数。

1、Session类对象mSession

每一个WindowState都有一个Session类成员变量，那么每一个窗口保存的都是同一个mSession吗？还是每一个窗口的mSession都不同呢？dump一下便知道了，dump日志告诉我上述两种情况都不是，事实上是每一个UI进程的所有窗口保存的是同一个Session，而每一个UI进程之间的Session又是不同的。Session顾名思义就是会话的意思，是ViewRootImpl用来调用WMS服务功能的东东，这个可以从Session类中的函数可以看出来。每一个UI进程在何时创建一个Session的呢？是在add第一个View时创建的，这点大家可以去研究下源码。Session类继承IWindowSession.Stub，可以知道WindowState中保存的是本地对象，而ViewRootImpl中 mWindowSession保存的是代理对象。

2、IWindow类对象mClient

mClient是一个代理对象，本地对象保存在ViewRootImpl中的mWindow中。WMS利用mClient来通知ViewRootImpl一些状态的改变等。mClient代表的是UI进程侧的一个窗口。

3、mOwnerUid

该变量保存的是UID，UID在Linux中是为多用户设计的，而在Android中赋予了新的使命--数据共享，android为每个应用几乎都分配了不同的UID，如果要实现两个程序的互访，可以定义相同的android:sharedUserId，并且签名相同便可互访。

4、mShowToOwnerOnly

这个变量是为多用户准备的，具体作用是标示一个窗口是否仅仅在当前UID下才能显示。可以查看DisplayContent.switchUserStacks()关于多用户转换逻辑。

boolean isHiddenFromUserLocked() {

return win.mShowToOwnerOnly

&& UserHandle.getUserId(win.mOwnerUid) != mService.mCurrentUserId;

}

5、mAppOp

该变量保存着该窗口特有的一些权限，比如如果该窗口是一个SYSTEM\_ALERT\_WINDOW窗口，那么mAppOp中会保存这个特殊权限，在很多地方权限管理AppOpsManager会检查这个窗口是否能申请到对应的资源。

6、mAttrs

这个变量中保存了几乎所有的关于该窗口的属性信息。比如窗口类型、flags、高度、宽度等等。computeFrameLw()中就是根据mAttrs来确定窗口Frame的。

7、mRequestedWidth、mRequestedHeight

\* The window size that was requested by the application.  These are in the application's coordinate space\*，保存着应用要求的窗口高和宽，可以查看WMS.relayoutWindow()函数。

8、mLayoutSeq

WMS.performLayoutLockedInner被调用一次，WMS.mLayoutSeq就加1，在一次Layout过程中，如果对某一个窗口调用了mPolicy.layoutWindowLw()函数进行布局，win.mLayoutSeq 就更新为WMS.mLayoutSeq值。所以通过查看WindowState.mLayoutSeq值就可以知道该窗口是在哪一次重新布局过的。

9、mAttachedWindow

如果该变量不为null，表示当前窗口需要附加在mAttachedWindow之上，当前窗口是mAttachedWindow的子窗口。mAttachedWindow的赋值流程如下：当UI进程需要添加一个Window时，最终会调用WMS.addWindow()，addWindow()函数中有这段代码：

if (type >= FIRST\_SUB\_WINDOW && type <= LAST\_SUB\_WINDOW) {

attachedWindow = windowForClientLocked(null, attrs.token, false);

if (attachedWindow == null) {

Slog.w(TAG, "Attempted to add window with token that is not a window: "

+ attrs.token + ". Aborting.");

return WindowManagerGlobal.ADD\_BAD\_SUBWINDOW\_TOKEN;

}

if (attachedWindow.mAttrs.type >= FIRST\_SUB\_WINDOW

&& attachedWindow.mAttrs.type <= LAST\_SUB\_WINDOW) {

Slog.w(TAG, "Attempted to add window with token that is a sub-window: "

+ attrs.token + ". Aborting.");

return WindowManagerGlobal.ADD\_BAD\_SUBWINDOW\_TOKEN;

}

}

如果添加的Window是一个子窗口，即(type >= FIRST\_SUB\_WINDOW && type <= LAST\_SUB\_WINDOW)，那么会根据attrs.token调用windowForClientLocked()找到父窗口，可以猜测并得出父窗口跟子窗口的attrs中保存的是同一个W对象的结论，结论是否正确待验证。windowForClientLocked()找到的父窗口WindowState会保存在mAttachedWindow中。

10、mLayoutAttached

对于(type >= FIRST\_SUB\_WINDOW && type <= LAST\_SUB\_WINDOW)的窗口，该属性在mAttrs.type !=WindowManager.LayoutParams.TYPE\_APPLICATION\_ATTACHED\_DIALOG才为true，也就是说一般的子窗口这个属性为true，除非type==TYPE\_APPLICATION\_ATTACHED\_DIALOG才为false；而对于非子窗口，该属性必须为false。mLayoutAttached的值会影响mPolicy.layoutWindowLw()，具体影响请研究performLayoutLockedInner()函数。

11、mIsImWindow、mIsWallpaper、mIsFloatingLayer

前两个变量很好理解，就是指示该窗口是否是输入法窗口、壁纸窗口，mIsFloatingLayer是什么？该变量在WindowState构造函数中赋值：

mIsFloatingLayer = mIsImWindow || mIsWallpaper;

如果当前窗口是输入法窗口或壁纸窗口，就为true，否则为false；全部搜索了一下这个变量的使用，没有地方使用这个变量，也就是说这个变量是打酱油的。

12、mWallpaperVisible

该变量是壁纸窗口WindowState的可见属性，其他窗口没有用到该属性。壁纸窗口本身不复杂，复杂的是壁纸窗口必须附在某些窗口之下，并且参与一些窗口动画之类的逻辑，导致WMS中到处都有wallpaper逻辑，看起来相当蛋疼。

13、mBaseLayer

mBaseLayer = mPolicy.windowTypeToLayerLw(a.type)

\* WindowManagerService.TYPE\_LAYER\_MULTIPLIER

+ WindowManagerService.TYPE\_LAYER\_OFFSET;

android中所有的WindowState都是依据窗口Z轴值来排列的，高度越高，越可见。这个Z轴高度的计算是基于mBaseLayer、可见性等等因素。上述代码是任何窗口mBaseLayer的计算公式，可以看出mBaseLayer=窗口类型值\*10000+1000，普通窗口类型值为2，所以普通窗口mBaseLayer=21000，这点可以通过dump日志验证。

14、mSubLayer

对于非子窗口，该变量值为0；对于子窗口mSubLayer的值为

mSubLayer = mPolicy.subWindowTypeToLayerLw(a.type);

mPolicy指向PhoneWindowManager，获得的mSubLayer可正可负，表示该子窗口在父窗口的上面或下面。

15、mLayer

mLayer保存的是该窗口的Layer高度值，但不是最终Z轴值，最终Z轴值是在mLayer的基础上再增加一个调整值。mLayer、最终Z轴值的计算是在WMS.assignLayersLocked()中完成的，并且Z轴值保存在WindowStateAnimator.mAnimLayer中。看看assignLayersLocked（）函数：

private final void assignLayersLocked(WindowList windows) {

int N = windows.size();

int curBaseLayer = 0;

int curLayer = 0;

int i;

boolean anyLayerChanged = false;

for (i=0; i<N; i++) {

final WindowState w = windows.get(i);

final WindowStateAnimator winAnimator = w.mWinAnimator;

boolean layerChanged = false;

int oldLayer = w.mLayer;

if (w.mBaseLayer == curBaseLayer || w.mIsImWindow

|| (i > 0 && w.mIsWallpaper)) {

curLayer += WINDOW\_LAYER\_MULTIPLIER;

w.mLayer = curLayer;

} else {

curBaseLayer = curLayer = w.mBaseLayer;

w.mLayer = curLayer;

}

if (w.mLayer != oldLayer) {

layerChanged = true;

anyLayerChanged = true;

}

final AppWindowToken wtoken = w.mAppToken;

oldLayer = winAnimator.mAnimLayer;

if (w.mTargetAppToken != null) {

winAnimator.mAnimLayer =

w.mLayer + w.mTargetAppToken.mAppAnimator.animLayerAdjustment;

} else if (wtoken != null) {

winAnimator.mAnimLayer =

w.mLayer + wtoken.mAppAnimator.animLayerAdjustment;

} else {

winAnimator.mAnimLayer = w.mLayer;

}

if (w.mIsImWindow) {

winAnimator.mAnimLayer += mInputMethodAnimLayerAdjustment;

} else if (w.mIsWallpaper) {

winAnimator.mAnimLayer += mWallpaperAnimLayerAdjustment;

}

if (winAnimator.mAnimLayer != oldLayer) {

layerChanged = true;

anyLayerChanged = true;

}

if (layerChanged && w.getStack().isDimming(winAnimator)) {

scheduleAnimationLocked();

}

}

//TODO (multidisplay): Magnification is supported only for the default display.

if (mDisplayMagnifier != null && anyLayerChanged

&& windows.get(windows.size() - 1).getDisplayId() == Display.DEFAULT\_DISPLAY) {

mDisplayMagnifier.onWindowLayersChangedLocked();

}

}

传进来的参数windows是所有窗口列表，大致排好序的。对于同一类型窗口的第一个窗口w.mLayer=w.mBaseLayer，第二个、第三个依次增加WINDOW\_LAYER\_MULTIPLIER（5），这个可以通过dump日志进行验证。

16、WindowToken类对象mToken

WindowToken是窗口令牌，跟WindowState之间的关系参考老罗的博客，引用一下“

在Window管理服务WindowManagerService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Window管理服务WindowManagerService中所对应的AppWindowToken对象为令牌的。从抽象的角度来看，就是在Window管理服务WindowManagerService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。”

还是自己分析一下，mToken的赋值在构造WindowState时传进来的，具体还的分析WMS.addWindow()函数，关键代码如下：

WindowToken token = mTokenMap.get(attrs.token);

if (token == null) {

if (type >= FIRST\_APPLICATION\_WINDOW && type <= LAST\_APPLICATION\_WINDOW) {

Slog.w(TAG, "Attempted to add application window with unknown token "

+ attrs.token + ". Aborting.");

return WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN;

}

if (type == TYPE\_INPUT\_METHOD) {

Slog.w(TAG, "Attempted to add input method window with unknown token "

+ attrs.token + ". Aborting.");

return WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN;

}

if (type == TYPE\_WALLPAPER) {

Slog.w(TAG, "Attempted to add wallpaper window with unknown token "

+ attrs.token + ". Aborting.");

return WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN;

}

if (type == TYPE\_DREAM) {

Slog.w(TAG, "Attempted to add Dream window with unknown token "

+ attrs.token + ". Aborting.");

return WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN;

}

token = new WindowToken(this, attrs.token, -1, false);

addToken = true;

} else if (type >= FIRST\_APPLICATION\_WINDOW && type <= LAST\_APPLICATION\_WINDOW) {

AppWindowToken atoken = token.appWindowToken;

if (atoken == null) {

Slog.w(TAG, "Attempted to add window with non-application token "

+ token + ". Aborting.");

return WindowManagerGlobal.ADD\_NOT\_APP\_TOKEN;

} else if (atoken.removed) {

Slog.w(TAG, "Attempted to add window with exiting application token "

+ token + ". Aborting.");

return WindowManagerGlobal.ADD\_APP\_EXITING;

}

if (type == TYPE\_APPLICATION\_STARTING && atoken.firstWindowDrawn) {

// No need for this guy!

if (localLOGV) Slog.v(

TAG, "\*\*\*\* NO NEED TO START: " + attrs.getTitle());

return WindowManagerGlobal.ADD\_STARTING\_NOT\_NEEDED;

}

} else if (type == TYPE\_INPUT\_METHOD) {

if (token.windowType != TYPE\_INPUT\_METHOD) {

Slog.w(TAG, "Attempted to add input method window with bad token "

+ attrs.token + ". Aborting.");

return WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN;

}

} else if (type == TYPE\_WALLPAPER) {

if (token.windowType != TYPE\_WALLPAPER) {

Slog.w(TAG, "Attempted to add wallpaper window with bad token "

+ attrs.token + ". Aborting.");

return WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN;

}

} else if (type == TYPE\_DREAM) {

if (token.windowType != TYPE\_DREAM) {

Slog.w(TAG, "Attempted to add Dream window with bad token "

+ attrs.token + ". Aborting.");

return WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN;

}

}

win = new WindowState(this, session, client, token,

attachedWindow, appOp[0], seq, attrs, viewVisibility, displayContent);

mTokenMap是一个HashMap，里面成对的保存着attrs.token和WindowToken

if (addToken) {

mTokenMap.put(attrs.token, token);

}

attrs.token比较关键，new一个新的WindowToken必须是attrs.token第一次被添加到WMS中时，如果之间已经添加过一个与attrs.token对应的窗口，那么就使用之前的WindowToken，就是说attrs.token相同的窗口，对应的WindowToken是相同的，那么attrs.token是什么东西呢？特别说明下，对应壁纸和输入法窗口比较特殊，这两个窗口是通过WMS.addWindowToken()来添加WindowToken到WMS中的，而不是像普通窗口那样通过addWindow()，这个大家搜一下源码就知道了。再来研究下attrs.token，还是算了吧，完全不知道从哪里赋值的，不过老罗已经给出关系图了，这个attrs.token应该就是跟ActivityRecord对应的东西。但是比较有趣的是对于StatusBar、Keyguard、KeyguardScrim及自己调用mWindowManager.addView()添加的View的WindowToken都是同一个WindowToken，这点通过dump日志可以验证。

17、mRootToken

mRootToken就是当前窗口的根窗口令牌，这个很好理解，比如某一个窗口的mAttachedWindow不为null，那么mRootToken就指向mAttachedWindow根部窗口的WindowToken，这个值可以为WindowToken，也可以为AppWindowToken，不可能为null。

WindowState appWin = this;

while (appWin.mAttachedWindow != null) {

appWin = appWin.mAttachedWindow;

}

WindowToken appToken = appWin.mToken;

while (appToken.appWindowToken == null) {

WindowToken parent = mService.mTokenMap.get(appToken.token);

if (parent == null || appToken == parent) {

break;

}

appToken = parent;

}

mRootToken = appToken;

mAppToken = appToken.appWindowToken;

18、mAppToken

mAppToken指向mAttachedWindow根部窗口的appWindowToken，如果mAttachedWindow根部窗口的appWindowToken不是一个Activity窗口，那么就为null，否则就为null。对于Activity的窗口难道不走WMS.addWindow()？因为addWindow()中没有new AppWindowToken逻辑，Activity窗口令牌是用AppWindowToken来描述的，好诡异。

19、AppWindowToken

该类用来描述一个Activity窗口令牌，对于输入法，壁纸，自己addView的窗口，及PopupWindow等都是用WindowToken来描述窗口令牌的。上面第18条说了既然Activity窗口令牌是用AppWindowToken来描述的，但addWindow()的逻辑中并没有针对Activity窗口new AppWindowToken，还是研究下其他地方看看是怎么回事。源码里只有addAppToken()函数中有new AppWindowToken的操作，想必对于Activity来说，会调用这个函数来创建一个AppWindowToken对象。在第16条中提到WindowToken与attrs.token一一对应，并且attrs.token会保存在WindowToken.token中，而对于AppWindowToken来说，与IApplicationToken一一对应，并且保存在AppWindowToken.token中，IApplicationToken是啥稍后再说。

public void addAppToken(int addPos, IApplicationToken token, int taskId, int stackId,

int requestedOrientation, boolean fullscreen, boolean showWhenLocked, int userId,

int configChanges) {

long inputDispatchingTimeoutNanos;

synchronized(mWindowMap) {

AppWindowToken atoken = findAppWindowToken(token.asBinder());

if (atoken != null) {

Slog.w(TAG, "Attempted to add existing app token: " + token);

return;

}

atoken = new AppWindowToken(this, token);

atoken.inputDispatchingTimeoutNanos = inputDispatchingTimeoutNanos;

atoken.groupId = taskId;

atoken.appFullscreen = fullscreen;

atoken.showWhenLocked = showWhenLocked;

atoken.requestedOrientation = requestedOrientation;

atoken.layoutConfigChanges = (configChanges &

(ActivityInfo.CONFIG\_SCREEN\_SIZE | ActivityInfo.CONFIG\_ORIENTATION)) != 0;

Task task = mTaskIdToTask.get(taskId);

if (task == null) {

task = createTask(taskId, stackId, userId, atoken);

} else {

task.addAppToken(addPos, atoken);

}

mTokenMap.put(token.asBinder(), atoken);

atoken.hidden = true;

atoken.hiddenRequested = true;

}

}

这段代码逻辑很容易懂，就是如果IApplicationToken已经添加到WMS中了就直接返回，否则new AppWindowToken，并把IApplicationToken作为参数保存在

AppWindowToken.appToken中，然后设置一下AppWindowToken的一些属性，包括groupId、appFullscreen、showWhenLocked、requestedOrientation、layoutConfigChanges等。mTaskIdToTask维护了TaskId和Task对，如果调用addAppToken时指定一个新的TaskId，那么就需要调用createTask()创建一个新的Task，否则就找到对应的Task，并把AppWindowToken添加到Task对应的位置，位置由addAppToken()的参数指定。还有新new出来的AppWindowToken属性hidden和hiddenRequested都为true，表示为隐藏。

特别说一下AppWindowToken的构造函数：

AppWindowToken(WindowManagerService \_service, IApplicationToken \_token) {

super(\_service, \_token.asBinder(),

WindowManager.LayoutParams.TYPE\_APPLICATION, true);

appWindowToken = this;

appToken = \_token;

mInputApplicationHandle = new InputApplicationHandle(this);

mAnimator = service.mAnimator;

mAppAnimator = new AppWindowAnimator(this);

}

AppWindowToken是继承WindowToken的，对于AppWindowToken来说appWindowToken变量不为null，指向本身，对于WindowToken来说appWindowToken变量为null，因此可以根据WindowToken.appWindowToken来判断该窗口是否是Activity窗口令牌。从构造函数中还可以看出每个AppWindowToken.mAnimator均指向WMS.mAnimator对象，每个AppWindowToken.mAppAnimator都不同。

addAppToken()函数在ActivityStack.startActivityLocked()中被调用，而startActivityLocked()被调用的流程如下：ActivityStackSupervisor.startActivityLocked()-->ActivityStackSupervisor.startActivityUncheckedLocked()-->ActivityStack.startActivityLocked()，这个以后再研究。

20、mTargetAppToken

对与输入法窗口，mTargetAppToken就是指向待输入的窗口，对于壁纸窗口，mTargetAppToken为null。搜了下源码，貌似只有输入法窗口mTargetAppToken才不为空，其他所有窗口该变量为null，这个变量应该是专门用来处理输入法窗口相关的逻辑。

21、mViewVisibility

这个是保存的是当前窗口的可见性状态。这个变量只有在两个地方可能会改变其值，一个是构造WindowState时，会初始化该值，另一个在WMS.relayoutWindow()函数中。部分代码如下：

boolean focusMayChange = isDefaultDisplay && (win.mViewVisibility != viewVisibility

|| ((flagChanges & FLAG\_NOT\_FOCUSABLE) != 0)

|| (!win.mRelayoutCalled));

boolean wallpaperMayMove = win.mViewVisibility != viewVisibility

&& (win.mAttrs.flags & FLAG\_SHOW\_WALLPAPER) != 0;

wallpaperMayMove |= (flagChanges & FLAG\_SHOW\_WALLPAPER) != 0;

win.mRelayoutCalled = true;

final int oldVisibility = win.mViewVisibility;

win.<span style="color:#ff0000;">mViewVisibility </span>= viewVisibility;

viewVisibility是relayoutWindow()函数中的参数。可以知道一个WindowState的可见性是通过WMS.relayoutWindow()来更新的。

22、mHaveFrame

mHaveFrame标示该窗口是否有Frame，其实准确的说应该是该窗口是否已经准备好了各种Frame尺寸，换句话就是说是否调用过WindowState.computeFrameLw()函数，只要调用过，mHaveFrame就为true。

23、mObscured

标识该窗口是否做模糊化处理？具体是什么，没搞清。正常情况下所有窗口mObscured为false。

24、mSeq

貌似跟“ StatusBar on multiple screens”有关，没研究清楚。正常情况下所有窗口mSeq为0.

25、mSystemUiVisibility

这个变量标识当前窗口下STATUSBAR、导航栏的可见属性值，这个值可以通过setSystemUiVisibility()函数来更新，mSystemUiVisibility一般是跟窗口的attrs.systemUiVisibility属性值保持一致的。一般情况下mSystemUiVisibility值为0，

systemUiVisibility = (attrs.systemUiVisibility|attrs.subtreeSystemUiVisibility);

.....

if (attrs != null && seq == win.mSeq) {

win.mSystemUiVisibility = systemUiVisibility;

}

26、mPolicyVisibility

mPolicyVisibility是窗口显示策略变量，与第21条的mViewVisibility不同，mViewVisibility用来保存当前窗口是否可见的状态，而mPolicyVisibility指导下一步该窗口是否可见，进而做一些逻辑处理。

27、mPolicyVisibilityAfterAnim

由名字可以知道窗口动画播完后mPolicyVisibility的值需更新为mPolicyVisibilityAfterAnim的值

boolean stepAnimationLocked(long currentTime) {

.......

if (mWin.mPolicyVisibility != mWin.mPolicyVisibilityAfterAnim) {

if (DEBUG\_VISIBILITY) {

Slog.v(TAG, "Policy visibility changing after anim in " + this + ": "

+ mWin.mPolicyVisibilityAfterAnim);

}

mWin.mPolicyVisibility = mWin.mPolicyVisibilityAfterAnim;

mWin.mDisplayContent.layoutNeeded = true;

if (!mWin.mPolicyVisibility) {

if (mService.mCurrentFocus == mWin) {

if (WindowManagerService.DEBUG\_FOCUS\_LIGHT) Slog.i(TAG,

"setAnimationLocked: setting mFocusMayChange true");

mService.mFocusMayChange = true;

}

mService.enableScreenIfNeededLocked();

}

}

28、mAppOpVisibility

mAppOpVisibility也是一个显示控制变量，默认为true，在setAppOpVisibilityLw()函数中设置其值。

public void setAppOpVisibilityLw(boolean state) {

if (mAppOpVisibility != state) {

mAppOpVisibility = state;

if (state) {

// If the policy visibility had last been to hide, then this

// will incorrectly show at this point since we lost that

// information. Not a big deal -- for the windows that have app

// ops modifies they should only be hidden by policy due to the

// lock screen, and the user won't be changing this if locked.

// Plus it will quickly be fixed the next time we do a layout.

showLw(true, true);

} else {

hideLw(true, true);

}

}

}

setAppOpVisibilityLw()函数在addWindow()和updateAppOpsState()中被调用，跟AppOpsManager权限检查紧密结合在一起。mAppOpVisibility就是跟权限管理器相关的显示属性。

29、mAttachedHidden

mAttachedHidden属性标识父窗口的可见状态。对于子窗口才设置mAttachedHidden属性。

30、mRelayoutCalled

mRelayoutCalled标识该窗口是否调用过relayoutWindow()函数，只有调用过mRelayoutCalled一次便一直为true。

31、mLayoutNeeded

mLayoutNeeded用来在UI进程在调用WMS.relayoutWindow()时记录窗口是否需要Layout，如果窗口大小发生改变、WindowManager.LayoutParams发生改变，那么mLayoutNeeded就设为true，等到调用performLayoutAndPlaceSurfacesLocked()-->performLayoutAndPlaceSurfacesLockedLoop()-->performLayoutAndPlaceSurfacesLockedInner()-->performLayoutLockedInner()时，mLayoutNeeded为true时才调用mPolicy.layoutWindowLw()。

32、mXOffset、mYOffset

这个值表示窗口的相对于屏幕左上角坐标原点的偏移量。对于一般窗口及自己addView的窗口，这两个值为0，壁纸窗口mXOffset一般为负。不过我在源码里也只有看到壁纸窗口的mXOffset才可能不为0，其他窗口不会动这个值。

boolean updateWallpaperOffsetLocked(WindowState wallpaperWin, int dw, int dh,

boolean sync) {

boolean changed = false;

boolean rawChanged = false;

float wpx = mLastWallpaperX >= 0 ? mLastWallpaperX : 0.5f;

float wpxs = mLastWallpaperXStep >= 0 ? mLastWallpaperXStep : -1.0f;

int availw = wallpaperWin.mFrame.right-wallpaperWin.mFrame.left-dw;

int offset = availw > 0 ? -(int)(availw\*wpx+.5f) : 0;

changed = wallpaperWin.mXOffset != offset;

if (changed) {

if (DEBUG\_WALLPAPER) Slog.v(TAG, "Update wallpaper "

+ wallpaperWin + " x: " + offset);

wallpaperWin.mXOffset = offset;

}

33、getTouchableRegion（）

public void getTouchableRegion(Region outRegion) {

final Rect frame = mFrame;

switch (mTouchableInsets) {

default:

case ViewTreeObserver.InternalInsetsInfo.TOUCHABLE\_INSETS\_FRAME:

outRegion.set(frame);

break;

case ViewTreeObserver.InternalInsetsInfo.TOUCHABLE\_INSETS\_CONTENT:

applyInsets(outRegion, frame, mGivenContentInsets);

break;

case ViewTreeObserver.InternalInsetsInfo.TOUCHABLE\_INSETS\_VISIBLE:

applyInsets(outRegion, frame, mGivenVisibleInsets);

break;

case ViewTreeObserver.InternalInsetsInfo.TOUCHABLE\_INSETS\_REGION: {

final Region givenTouchableRegion = mGivenTouchableRegion;

outRegion.set(givenTouchableRegion);

outRegion.translate(frame.left, frame.top);

break;

}

}

}

获取一个窗口的触摸区域，触摸区域的计算跟mTouchableInsets有关，mTouchableInsets指示当前窗口的触摸区域如何计算，“Flag indicating whether the touchable region should be adjusted by the visible insets”。

事实上真正的触摸区域计算并不是完全调用getTouchableRegion()来获取，这点可以通过InputMonitor.addInputWindowHandleLw()看出来

final boolean modal = (flags & (WindowManager.LayoutParams.FLAG\_NOT\_TOUCH\_MODAL

| WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE)) == 0;

if (modal && child.mAppToken != null) {

// Limit the outer touch to the activity stack region.

flags |= WindowManager.LayoutParams.FLAG\_NOT\_TOUCH\_MODAL;

inputWindowHandle.touchableRegion.set(child.getStackBounds());

} else {

// Not modal or full screen modal

child.getTouchableRegion(inputWindowHandle.touchableRegion);

}

updateInputWindowsLw()-->addInputWindowHandleLw()会将WMS中WindowList中的所有窗口更新到InputDispatcher中去，方便InputDispatcher根据窗口关系和触摸区域找到相应的窗口，然后将触摸事件分发给找到的窗口，而每个窗口的触摸区域便是上面这段代码设定的。可以看出对于无WindowManager.LayoutParams.FLAG\_NOT\_TOUCH\_MODAL和WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE属性的Activity窗口，触摸区域等于WindowState.getStackBounds()，其他类型窗口的触摸区域则通过getTouchableRegion()计算得来。

34、mConfiguration

这个变量保存值跟WMS.mCurConfiguration是完全相同的，可以通过日志进行验证。

35、mHasSurface

该变量指示是否为该窗口new 一个SurfaceControl对象，即调用WindowState.mWinAnimator.createSurfaceLocked()来创建。SurfaceControl是WMS与SurfaceFlinger通信的接口之一，每一个WindowState都对应有一个SurfaceControl，还对应在SurfaceFlinger中一个Layer对象。

36、mShownFrame

非全屏应用：mShownFrame=[0.0,0.0][720.0,1280.0]

全屏应用：    mShownFrame=[0.0,0.0][720.0,1280.0]

壁纸：            mShownFrame=[-310.0,0.0][1030.0,1280.0]

mShownFrame的计算在WindowStateAnimator.computeShownFrameLocked()中完成。mShownFrame设置到SurfaceFlinger中是在WindowStateAnimator.setSurfaceBoundariesLocked()中完成，保存在Layer的mCurrentState.transform.set(x, y)和mCurrentState.requested中。

37、isReadyForDisplay()

boolean isReadyForDisplay() {

if (mRootToken.waitingToShow &&

mService.mAppTransition.isTransitionSet()) {

return false;

}

return mHasSurface && mPolicyVisibility && !mDestroying

&& ((!mAttachedHidden && mViewVisibility == View.VISIBLE

&& !mRootToken.hidden)

|| mWinAnimator.mAnimation != null

|| ((mAppToken != null) && (mAppToken.mAppAnimator.animation != null)));

}

函数虽短，理解起来很费劲。。。通过日志来看，只有当前显示的窗口isReadyForDisplay()才返回true。

38、mFrame

非全屏应用：mFrame=[0,0][720,1280]

全屏应用：    mFrame=[0,0][720,1280]

壁纸：           mFrame=[0,0][1340,1280]

mFrame的计算在WindowState.computeFrameLw()中完成：

Gravity.apply(mAttrs.gravity, w, h, mContainingFrame,

(int) (x + mAttrs.horizontalMargin \* pw),

(int) (y + mAttrs.verticalMargin \* ph), mFrame);

//System.out.println("Out: " + mFrame);

// Now make sure the window fits in the overall display.

Gravity.applyDisplay(mAttrs.gravity, df, mFrame);

具体如何计算可自行仔细研究。

39、mSystemDecorRect