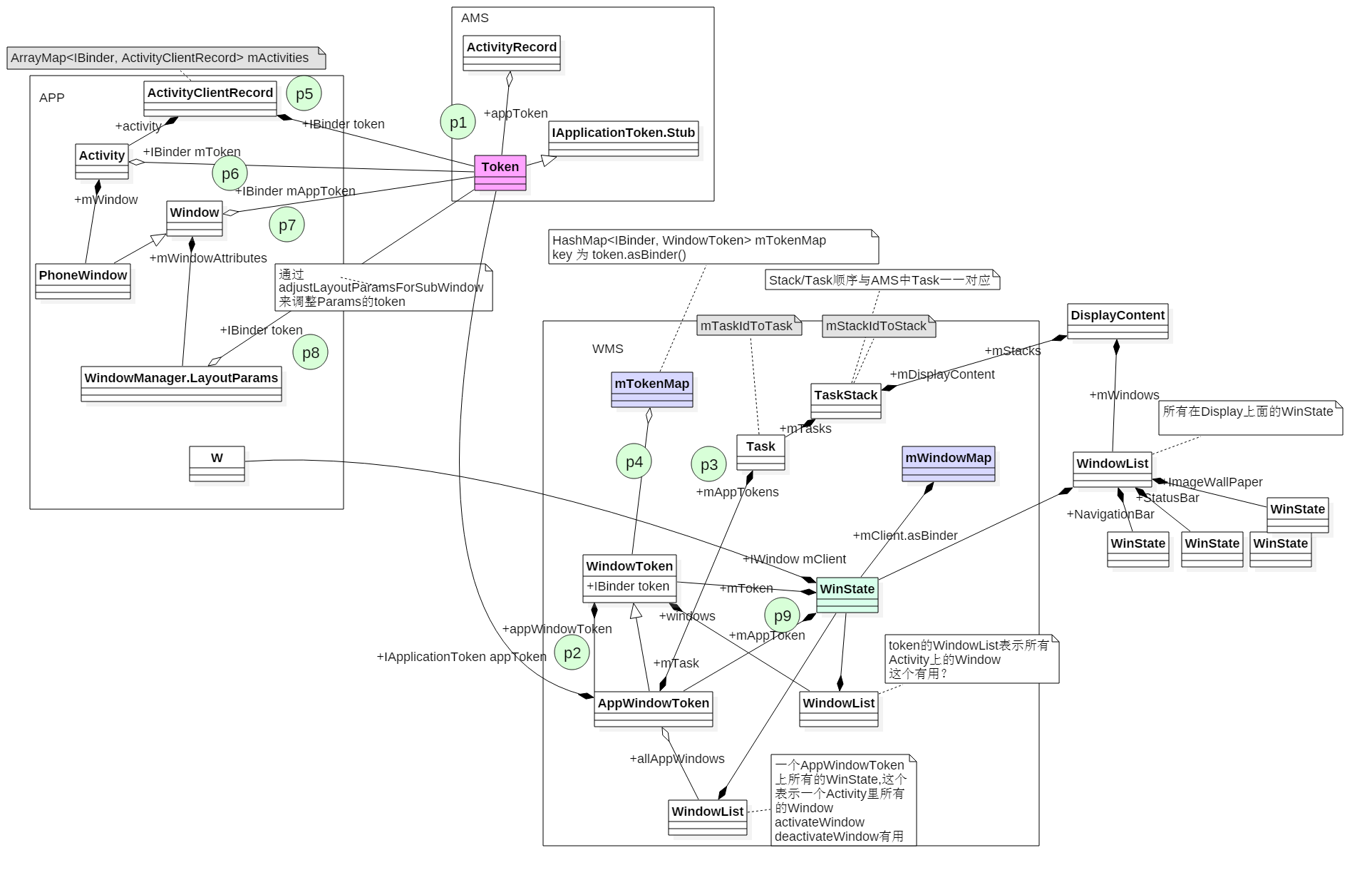
**WMS中的Z order**

2017年07月13日 16:01:00 [weixin\_33889245](https://me.csdn.net/weixin_33889245) 阅读数：5

转载请标注: [http://www.jianshu.com/p/5e802482caa4](https://www.jianshu.com/p/5e802482caa4)  
始终觉得要理解一个庞大的东西，需要将它分解，从部分理解开始，慢慢理解了，那么这个庞大的东西也就理解。

在看这篇文章之前，非常建议参考[AMS/WMS/APP 中Token惟一性](https://www.jianshu.com/p/5e2efbaa2949), 因为这里面会涉及到各种Token,稍不注意就不知道具体的token是指什么了。

再把这张图贴在这里， 这样看代码时，对照着看就更容易理解了。



Token的惟一性

**注意:** 本篇以Launcher启动一个App (MainActivity)时，来学习Z order. 另外由于在启动一个App时会有一个starting Window, 略过这部分。

**一、加入Window到WMS的窗口堆栈**

1. public int addWindow(Session session, IWindow client, int seq,
2. WindowManager.LayoutParams attrs, int viewVisibility, int displayId,
3. Rect outContentInsets, Rect outStableInsets, Rect outOutsets,
4. InputChannel outInputChannel) {
6. final int type = attrs.type;
7. *//attrs即Activity的Window中的LayoutParams, 这里值是FIRST\_APPLICATION\_WINDOW*
9. synchronized(mWindowMap) {
10. *//displayId为具体的显示屏，一般没有外接都是默认的显示屏*
11. *//DisplayContent为显示屏上显示的内容，主要是一些Window对应的WindowState*
12. final DisplayContent displayContent = getDisplayContentLocked(displayId);
14. *//如果WindowMap已经加入过了，就直接返回，不用再生成对应的WindowState了*
15. if (mWindowMap.containsKey(client.asBinder())) {
16. return WindowManagerGlobal.ADD\_DUPLICATE\_ADD;
17. }
19. boolean addToken = false;
20. WindowToken token = mTokenMap.get(attrs.token);
21. *//获得Activity对应在WMS中的WindowToken, 在这里已经不为空了，具体见[2. WMS中的token]*
22. AppWindowToken atoken = null;
23. if (token == null) {
24. } else if (type >= FIRST\_APPLICATION\_WINDOW && type <= LAST\_APPLICATION\_WINDOW) {
25. atoken = token.appWindowToken; *//获得AppWindowToken*
26. } else (XXXX){
27. }
29. *//生成对应的WindowState用于保存Window信息*
30. WindowState win = new WindowState(this, session, client, token,
31. attachedWindow, appOp[0], seq, attrs, viewVisibility, displayContent);
33. res = WindowManagerGlobal.ADD\_OKAY;
35. if (addToken) {
36. mTokenMap.put(attrs.token, token); *//这个在addAppToken中已经加入过了，所以不用加入了*
37. }
38. win.attach(); *//Session里 mNumWindow加1，表示现在总共有多少window了*
39. mWindowMap.put(client.asBinder(), win); *//储存WindowState*
41. if (type == TYPE\_INPUT\_METHOD) {*//输入法窗口，这里不讨论*
42. } else if (type == TYPE\_INPUT\_METHOD\_DIALOG) {
43. } else {
44. addWindowToListInOrderLocked(win, true); *//将WindowState加入到窗口堆栈中*
45. }
46. }
47. return res;
48. }

**1.1 Window的类型**

addWindow最后会根据不同的Window类型做不同的处理，那这些Window类型是什么呢？

**attrs.type** 这个指明 Window的类型, 它的值在如下代码中初始化

1. private final WindowManager.LayoutParams mWindowAttributes = new WindowManager.LayoutParams();
3. public LayoutParams() {
4. super(LayoutParams.MATCH\_PARENT, LayoutParams.MATCH\_PARENT);
5. type = TYPE\_APPLICATION; *//默认为TYPE\_APPLICATION类型*
6. format = PixelFormat.OPAQUE;
7. }

由代码可知，LayoutParams默认的Type是TYPE\_APPLICATION = 2; 即普通的应用窗口

但是会在**handleResumeActivity**被修改掉，如下

1. --- in handleResumeActivity
3. WindowManager.LayoutParams l = r.window.getAttributes();
4. a.mDecor = decor;
5. l.type = WindowManager.LayoutParams.TYPE\_BASE\_APPLICATION; *//修改窗口类型*
6. wm.addView(decor, l);

但是在addView之前修改了窗口类型为TYPE\_BASE\_APPLICATION = 1类型，这个类型是基类，是Activity中最底层的窗口，换句话说，其它的窗口都会叠加在该窗口之上，比如Dialog窗口(它使用默认的TYPE\_APPLICATION)。 这也说明type值越大，它就越在上面。

Android定义的窗口值如下:

* **应用程序窗口**

| **定义** | **值** | **意义** |
| --- | --- | --- |
| FIRST\_APPLICATION\_WINDOW | 1 | 第一个窗口，也就是TYPE\_BASE\_APPLICATION窗口 |
| TYPE\_BASE\_APPLICATION | 1 | Activity的基窗口 |
| TYPE\_BASE\_APPLICATION | 2 | 默认窗口类型 |
| TYPE\_APPLICATION\_STARTING | 3 | 应用程序启动过程中的中间窗口，当真下窗口适配完会关闭该窗口 |
| LAST\_APPLICATION\_WINDOW | 99 | 应用程序最后一个窗口 |

* **子窗口**
* **系统窗口**  
  其它两种窗口请参考 [Android入门之窗口类型](https://link.jianshu.com/?t=http://blog.csdn.net/geloin/article/details/42779025)([http://blog.csdn.net/geloin/article/details/42779025](https://link.jianshu.com/?t=http://blog.csdn.net/geloin/article/details/42779025))

**Question:** 如果将type设为大于99会怎么呢？

在WMS的addWindow会对type进行权限检查

int res = mPolicy.checkAddPermission(attrs, appOp);

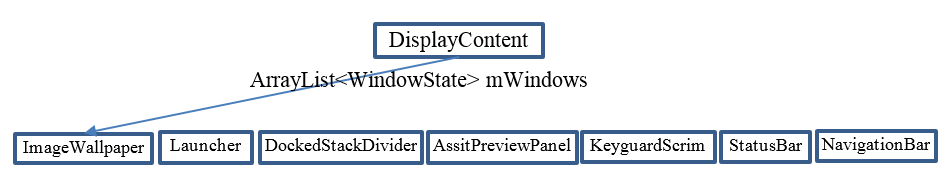
即如果type不属于 应用窗口/子窗口/系统窗口，直接会报异常。

而真正将Window加入到窗口堆栈的函数是 **addWindowToListInOrderLocked**

**1.2 addWindowToListInOrderLocked**

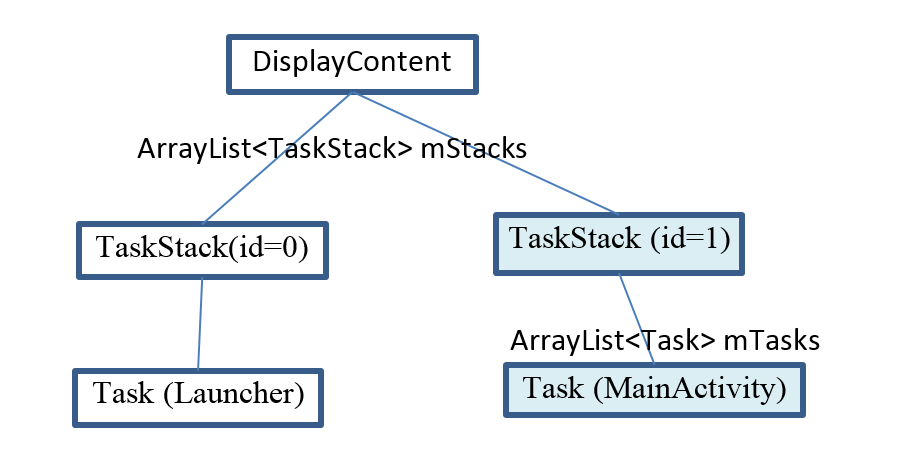
1. private void addWindowToListInOrderLocked(final WindowState win, boolean addToToken) {
2. if (win.mAttachedWindow == null) { *//新启动的Activity, 没有attach的Window*
3. final WindowToken token = win.mToken; *//WindowState中的mToken*
4. int tokenWindowsPos = 0;
5. if (token.appWindowToken != null) { *//这里显然不为null,在生成AppWindowToken时指向 见图p2*
6. tokenWindowsPos = addAppWindowToListLocked(win); *//进入这个分支*
7. }
8. }
9. private int addAppWindowToListLocked(final WindowState win) {
10. final DisplayContent displayContent = win.getDisplayContent();
11. final IWindow client = win.mClient;
12. final WindowToken token = win.mToken;
14. final WindowList windows = displayContent.getWindowList(); *//得到DisplayContent中所有的WindowState,*

此时窗口堆栈顺序如下:

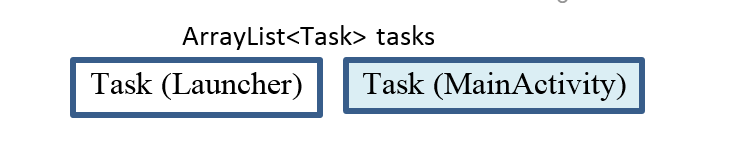


窗口堆栈顺序

1. WindowList tokenWindowList = getTokenWindowsOnDisplay(token, displayContent);
2. *//当前Activity MainActivity是第一次启动，所以它并没有其它Window*
4. final ArrayList<Task> tasks = displayContent.getTasks(); *//这里会得到所有的task*



此时的Stack\_Task的关系

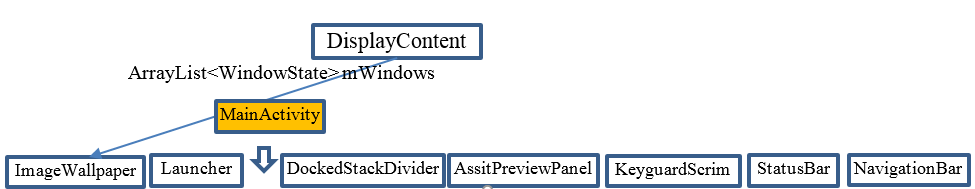


tasks的值

接着看addAppWindowToListLocked后面的代码

1. int taskNdx;
2. int tokenNdx = -1;
3. for (taskNdx = tasks.size() - 1; taskNdx >= 0; --taskNdx) { *//当taskNdx = 1时*
4. AppTokenList tokens = tasks.get(taskNdx).mAppTokens; *//此时的task里仅有一个MainActivity中的AppWindowToken*
5. for (tokenNdx = tokens.size() - 1; tokenNdx >= 0; --tokenNdx) { *//tokenNdx = 0*
6. final AppWindowToken t = tokens.get(tokenNdx); *//此时t 为 MainActivity中的AppWindowToken*
7. if (t == token) { *//相等*
8. --tokenNdx; *//此时tokenNdx = -1*
9. if (tokenNdx < 0) { *//进入此分支*
10. --taskNdx; *//寻找前一个task, 此时taskNdx = 0*
11. if (taskNdx >= 0) { *//进入该分支*
12. *//Launcher Task中的也仅有一个App token, 这里减1的目的是当 add 的时候，以tokenNdx为索引，加在后面*
13. tokenNdx = tasks.get(taskNdx).mAppTokens.size() - 1;
14. }
15. }
16. break;
17. }
18. }
19. if (tokenNdx >= 0) { *//此时tokenNdx = 0, 直接break掉*
20. break;
21. }
22. }

25. *// Continue looking down until we find the first*
26. *// token that has windows on this display.*
27. *//此时taskNdx = 0, tokenNdx = 0*
28. for ( ; taskNdx >= 0; --taskNdx) {
29. AppTokenList tokens = tasks.get(taskNdx).mAppTokens; *//获得Launcher Task中所有的tokens*
30. for ( ; tokenNdx >= 0; --tokenNdx) {
31. final AppWindowToken t = tokens.get(tokenNdx); *//这里获得的是Launcher的token*
32. tokenWindowList = getTokenWindowsOnDisplay(t, displayContent); *//获得Launcher token中的WinState*
33. final int NW = tokenWindowList.size(); *//这里NW为1，因为没有其它子窗口*
34. if (NW > 0) {
35. pos = tokenWindowList.get(NW-1); *//直接获得最后的win state (这里也是Launcher的WinState)*
36. break;
37. }
38. }
39. }
41. if (pos != null) { *//pos已经不为空了*
42. WindowToken atoken = mTokenMap.get(pos.mClient.asBinder()); *//*
43. if (atoken != null) { *//获得Launcher的 WindowToken*
44. final int NC = atoken.windows.size();
45. if (NC > 0) {
46. WindowState top = atoken.windows.get(NC-1);
47. if (top.mSubLayer >= 0) {
48. pos = top;
49. }
50. }
51. }
52. placeWindowAfter(pos, win); *//在Launcher后插入Win (MainActivity)*
53. return tokenWindowsPos; *//返回*
54. }
55. }



此时窗口堆栈顺序

**二、Z -order**

前面已经将Window加入到了窗口堆栈，并不是Z order,下面来看下什么是Z order。

由于前面窗口堆栈已经新加入了一个window, 即窗口堆栈变化了，那么此时就要计算 Z order了  
具体是在 addWindow的最后了

1. --- in addWindow()
2. mLayersController = new WindowLayersController(this);

WMS初始化时初始化WindowLayersController, 该类主要是为DisplayContent里的Windows也就是Window堆栈指定layer, 也就是计算它们的Z order。 计算Z order时，从窗口堆栈的底部到顶部，层级越高，那么它的 layer 越大，即它的Z order越大。

再继续讲计算 Z -order之前先简单看下与Z order有关的几个变量, 具体在WindowState中

1. --- in WindowState
3. final WindowStateAnimator mWinAnimator;
4. int mLayer;
6. final int mBaseLayer;
7. final int mSubLayer;

WindowStateAnimator 是一个Window的动画以及对Surface的操作类，它类中有个Animator的 Z order值，

* mLayer  
  mLayer是Base的Z -order值，它的值是动态算出来的。
* mAnimLayer  
  它的值是由mLayer算出来的, SurfaceFlinger中的Layer的Z order也就是mAnimLayer。

可以从定义看出 mBaseLayer, mSubLayer是final, 是不可再重新赋值的，所以它们的值是固定的，很显然是在WindowState初始化时赋值的。

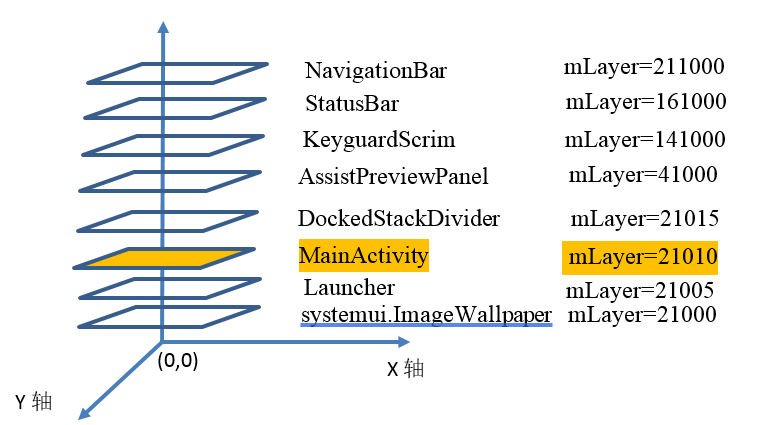
1. --- in WindowState构造函数中
3. if ((mAttrs.type >= FIRST\_SUB\_WINDOW && mAttrs.type <= LAST\_SUB\_WINDOW)) {
4. *//子窗口，不讨论*
5. } else {
6. mBaseLayer = mPolicy.windowTypeToLayerLw(a.type)
7. \* WindowManagerService.TYPE\_LAYER\_MULTIPLIER *//该值是10000*
8. + WindowManagerService.TYPE\_LAYER\_OFFSET; *//该值为1000*
9. mSubLayer = 0;
10. }
11. public int windowTypeToLayerLw(int type) {
12. if (type >= FIRST\_APPLICATION\_WINDOW && type <= LAST\_APPLICATION\_WINDOW) {
13. return 2;
14. }
15. }

**以非子窗口为例**  
此时mSubLayer=0, 而mBaseLayer与Window类型相关，以本文例子为例, 应用程序的窗口映射的z-order为2, 所以经过计算, mBaseLayer就为21000

**注意:并不代表 Window type值越大，它的Z -order值越大，type值会被windowTypeToLayerLw重新映射。**

下面来看下是怎么计算z order值的

1. mLayersController.assignLayersLocked(displayContent.getWindowList());
3. final void assignLayersLocked(WindowList windows) {
4. int curBaseLayer = 0; *//临时的基值*
5. int curLayer = 0; *//当前的值*
6. boolean anyLayerChanged = false; *//只要一个Z order值变了，这里就会为true*
7. for (int i = 0, windowCount = windows.size(); i < windowCount; i++) {
8. final WindowState w = windows.get(i);
9. boolean layerChanged = false; *//针对每一个window*
11. int oldLayer = w.mLayer;
12. if (w.mBaseLayer == curBaseLayer || w.mIsImWindow || (i > 0 && w.mIsWallpaper)) {
13. curLayer += WINDOW\_LAYER\_MULTIPLIER; *//WINDOW\_LAYER\_MULTIPLIER值为5,*
14. } else {
15. curBaseLayer = curLayer = w.mBaseLayer;
16. }
17. assignAnimLayer(w, curLayer); *//设置Z order值，以及动画的Z order*
19. if (w.mLayer != oldLayer || w.mWinAnimator.mAnimLayer != oldLayer) {
20. layerChanged = true;
21. anyLayerChanged = true;
22. }
24. if (w.mAppToken != null) {
25. mHighestApplicationLayer = Math.max(mHighestApplicationLayer,
26. w.mWinAnimator.mAnimLayer); *//获得应用窗口中最大值，注意这里只是针对应用窗口。*
27. }
28. collectSpecialWindows(w); *//收集特殊的窗口*
30. if (layerChanged) {
31. w.scheduleAnimationIfDimming();
32. }
33. }
35. adjustSpecialWindows(); *//针对特殊窗口再做调整, 不讨论*
37. if (mService.mAccessibilityController != null && anyLayerChanged
38. && windows.get(windows.size() - 1).getDisplayId() == Display.DEFAULT\_DISPLAY) {
39. mService.mAccessibilityController.onWindowLayersChangedLocked();
40. }
41. }



Z order值

**三、将 Z order更新到Native层**

# Z order值是Surface里的概念(SurfaceFlinger中对应的是Layer类), 由于addWindow发生的很早，此时并没有Surface. 所以并不是在这里设备Z order值的，Android窗口管理分析（3）：窗口分组及Z-order的确定

[](https://www.jianshu.com/u/3b1099674c2c)

[看书的小蜗牛](https://www.jianshu.com/u/3b1099674c2c)  关注

 0.1 2017.09.08 13:35\* 字数 4364 阅读 1840评论 1喜欢 9

在Android系统中，窗口是有分组概念的，例如，Activity中弹出的所有PopupWindow会随着Activity的隐藏而隐藏，可以说这些都附属于Actvity的子窗口分组，对于Dialog也同样如此，只不过Dialog与Activity属于同一个分组。之间已经简单介绍了窗口类型划分：应用窗口、子窗口、系统窗口，Activity与Dialog都属于应用窗口，而PopupWindow属于子窗口，Toast、输入法等属于系统窗口。只有应用窗口与系统窗口可以作为父窗口，子窗口不能作为子窗口的父窗口，也就说Activity与Dialog或者系统窗口中可以弹出PopupWindow，但是PopupWindow不能在自己内部弹出PopupWindow子窗口。日常开发中，一些常见的问题都同窗口的分组有关系，比如为什么新建Dialog的时候必须要用Activity的Context，而不能用Application的；为什么不能以PopupWindow的View为锚点弹出子PopupWindow？其实这里面就牵扯都Android的窗口组织管理形式，本文主要包含以下几点内容：

* 窗口的分组管理 ：应用窗口组、子窗口组、系统窗口组
* Activity、Dialg应用窗口及PopWindow子窗口的添加原理跟注意事项
* 窗口的Z次序管理：窗口的分配序号、次序调整等
* WMS中窗口次序分配如何影响SurfaceFlinger服务

在[WMS窗口添加一文](https://www.jianshu.com/p/e4b19fc36a0e)中分析过,窗口的添加是通过WindowManagerGlobal.addView()来完成 函数原型如下

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

Display display, Window parentWindow)

前三个参数是必不可少的，view、params、display，其中display表示要输出的显示设备，先不考虑。view 就是APP要添加到WindowManagerGlobal管理的View，而 params是WindowManager.LayoutParams，主要用来描述窗口属性，WindowManager.LayoutParams有两个很重要的参数type与token，

public static class LayoutParams extends ViewGroup.LayoutParams

implements Parcelable {

...

public int type;

...

public IBinder token = null;

}

type用来描述窗口的类型，而token其实是标志窗口的分组，token相同的窗口属于同一分组，后面会知道这个token其实是WMS在APP端对应的一个WindowToken的键值。这里先看一下type参数，之前曾添加过Toast窗口，它的type值是TYPE\_TOAST，标识是一个系统提示窗口，下面先简单看下三种窗口类型的Type对应的值，首先看一下应用窗口

| **窗口TYPE值** | **窗口类型** |
| --- | --- |
| FIRST\_APPLICATION\_WINDOW = 1 | 开始应用程序窗口 |
| TYPE\_BASE\_APPLICATION=1 | 所有程序窗口的base窗口，其他应用程序窗口都显示在它上面 |
| TYPE\_APPLICATION =2 | 普通应用程序窗口，token必须设置为Activity的token |
| TYPE\_APPLICATION\_STARTING =3 | 应用程序启动时所显示的窗口 |
| LAST\_APPLICATION\_WINDOW = 99 | 结束应用程序窗口 |

一般Activity都是TYPE\_BASE\_APPLICATION类型的，而TYPE\_APPLICATION主要是用于Dialog，再看下子窗口类型

| **窗口TYPE值** | **窗口类型** |
| --- | --- |
| FIRST\_SUB\_WINDOW = 1000 | SubWindows子窗口，子窗口的Z序和坐标空间都依赖于他们的宿主窗口 |
| TYPE\_APPLICATION\_PANEL =1000 | 面板窗口，显示于宿主窗口的上层 |
| TYPE\_APPLICATION\_MEDIA =1001 | 媒体窗口（例如视频），显示于宿主窗口下层 |
| TYPE\_APPLICATION\_SUB\_PANEL =1002 | 应用程序窗口的子面板，显示于所有面板窗口的上层 |
| TYPE\_APPLICATION\_ATTACHED\_DIALOG = 1003 | 对话框，类似于面板窗口，绘制类似于顶层窗口，而不是宿主的子窗口 |
| TYPE\_APPLICATION\_MEDIA\_OVERLAY =1004 | 媒体信息，显示在媒体层和程序窗口之间，需要实现半透明效果 |
| LAST\_SUB\_WINDOW=1999 | 结束子窗口 |

最后看几个系统窗口类型，

| **窗口TYPE值** | **窗口类型** |
| --- | --- |
| FIRST\_SYSTEM\_WINDOW = 2000 | 系统窗口 |
| TYPE\_STATUS\_BAR = FIRST\_SYSTEM\_WINDOW | 状态栏 |
| TYPE\_SYSTEM\_ALERT = FIRST\_SYSTEM\_WINDOW+3 | 系统提示，出现在应用程序窗口之上 |
| TYPE\_TOAST = FIRST\_SYSTEM\_WINDOW+5 | 显示Toast |

了解窗口类型后，我们需要面对的首要问题是：**窗口如何根据类型进行分组归类的？Dialog是如何确定附属Activity，PopupWindow如何确定附属父窗口？**。

**窗口的分组原理**

如果用一句话概括窗口分组的话：**Android窗口是以token来进行分组的**，同一组窗口握着相同的token，什么是token呢？在 Android WMS管理框架中，token一个IBinder对象，IBinder在实体端与代理端会相互转换，这里只看实体端，它的取值只有两种:ViewRootImpl中ViewRootImpl.W，或者是ActivityRecord中的IApplicationToken.Stub对象，其中ViewRootImpl.W的实体对象在ViewRootImpl中实例化，而IApplicationToken.Stub在ActivityManagerService端实例化，之后被AMS添加到WMS服务中去，作为Activity应用窗口的键值标识。之前说过Activity跟Dialog属于同一分组，现在就来看一下Activity跟Dialog的token是如何复用的，这里的复用分为APP端及WMS服务端，关于窗口的添加流程之前已经分析过，这里只跟随窗口token来分析窗口的分组，我们知道在WMS端，WindowState与窗口的一一对应，而WindowToken与窗口分组，这可以从两者的定义看出如下：

class WindowToken {

final WindowManagerService service;

final IBinder token;

final int windowType;

final boolean explicit;

<!--当前窗口对应appWindowToken，是不是同Activity存在依附关系-->

AppWindowToken appWindowToken;

<!--关键点1 当前WindowToken对应的窗口列表-->

final WindowList windows = new WindowList();

...

}

final class WindowState implements WindowManagerPolicy.WindowState {

static final String TAG = "WindowState";

final WindowManagerService mService;

final WindowManagerPolicy mPolicy;

final Context mContext;

final Session mSession;

<!--当前WindowState对应IWindow窗口代理-->

final IWindow mClient;

<!--当前WindowState对应的父窗口-->

final WindowState mAttachedWindow;

...

<!--当前WindowState隶属的token-->

WindowToken mToken;

WindowToken mRootToken;

AppWindowToken mAppToken;

AppWindowToken mTargetAppToken;

...

}

可以看到WindowToken包含一个 WindowList windows = new WindowList()，其实就是WindowState列表；而WindowState有一个WindowToken mToken，也就是WindowToken包含一个WindowState列表，而每个WindowState附属一个WindowToken窗口组，示意图如下：

WindowToken与WindowState关系.jpg

**Activity对应token及WindowToken（AppWindowToken）的添加**

AMS在为Activity创建ActivityRecord的时候，会新建IApplicationToken.Stub appToken对象，在startActivity之前会首先向WMS服务登记当前Activity的Token，随后，通过Binder通信将IApplicationToken传递给APP端，在通知ActivityThread新建Activity对象之后，利用Activity的attach方法添加到Activity中，先看第一步AMS将Activity的token加入到WMS中，并且为Activity创建APPWindowToken。

<!--AMS ActivityStack.java中代码 -->

final void startActivityLocked(ActivityRecord r, boolean newTask,

boolean doResume, boolean keepCurTransition, Bundle options) {

...<!--关键点1 添加Activity token到WMS-->

mWindowManager.addAppToken(task.mActivities.indexOf(r), r.appToken,XXX);

}

@Override

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

int requestedOrientation, boolean fullscreen, boolean showForAllUsers, int userId,

int configChanges, boolean voiceInteraction, boolean launchTaskBehind) {

synchronized(mWindowMap) {

<!--新建AppWindowToken-->

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

atoken = new AppWindowToken(this, token, voiceInteraction);

...

<!--将AppWindowToken以IApplicationToken.Stub为键值放如WMS的mTokenMap中-->

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

<!--开始肯定是隐藏状态，因为还没有resume-->

atoken.hidden = true;

atoken.hiddenRequested = true;

}

}

也就是说Activity分组的Token其实是早在Activity显示之前就被AMS添加到WMS中去的，之后AMS才会通知App端去新建Activity，并将Activity的Window添加到WMS中去，接着看下APP端的流程：

private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {

<!--关键点1 新建Activity-->

Activity activity = null;

try {

java.lang.ClassLoader cl = r.packageInfo.getClassLoader();

activity = mInstrumentation.newActivity(

cl, component.getClassName(), r.intent);

...

try {

Application app = r.packageInfo.makeApplication(false, mInstrumentation);

if (activity != null) {

<!--关键点2 新建appContext-->

Context appContext = createBaseContextForActivity(r, activity);

CharSequence title = r.activityInfo.loadLabel(appContext.getPackageManager());

Configuration config = new Configuration(mCompatConfiguration);

<!--关键点3 attach到WMS-->

activity.attach(appContext, this, getInstrumentation(), r.token，XXX);

...

}

关键点1，新建一个Activity，之后会为Activiyt创建一个appContext，这个Context主要是为了activity.attach使用的，其实就是单纯new一个ContextImpl，之后Activity会利用attach函数将ContextImpl绑定到自己身上。

static ContextImpl createActivityContext(ActivityThread mainThread,

LoadedApk packageInfo, int displayId, Configuration overrideConfiguration) {

return new ContextImpl(null, mainThread, packageInfo, null, null, false,

null, overrideConfiguration, displayId);

}

final void attach(Context context, ActivityThread aThread,

Instrumentation instr, IBinder token, int ident,

Application application, Intent intent, ActivityInfo info,

CharSequence title, Activity parent, String id,

NonConfigurationInstances lastNonConfigurationInstances,

Configuration config, String referrer, IVoiceInteractor voiceInteractor) {

<!--关键点1 为Activity绑定ContextImpl 因为Activity只是一个ContextWraper-->

attachBaseContext(context);

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

<!--关键点2 new一个PhoneWindow 并设置回调-->

mWindow = new PhoneWindow(this);

mWindow.setCallback(this);

mWindow.setOnWindowDismissedCallback(this);

mWindow.getLayoutInflater().setPrivateFactory(this);

...

<!--关键点3 Token的传递-->

mToken = token;

mIdent = ident;

mApplication = application;

...

mWindow.setWindowManager(

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

mToken, mComponent.flattenToString(),

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

<!--将Window的WindowManager赋值给Activity-->

mWindowManager = mWindow.getWindowManager();

mCurrentConfig = config;

}

mWindow.setWindowManager并不是直接为Window设置WindowManagerImpl，而是利用当前的WindowManagerImpl重新为Window创建了一个WindowManagerImpl，并将自己设置此WindowManagerImpl的parentWindow：

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

boolean hardwareAccelerated) {

mAppToken = appToken;

mAppName = appName;

mHardwareAccelerated = hardwareAccelerated

|| SystemProperties.getBoolean(PROPERTY\_HARDWARE\_UI, false);

if (wm == null) {

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

}

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

}

public WindowManagerImpl createLocalWindowManager(Window parentWindow) {

return new WindowManagerImpl(mDisplay, parentWindow);

}

之后将Window的WindowManagerImpl传递给Activity，作为Activity的WindowManager将来Activity通过getSystemService获取WindowManager服务的时候，其实是直接返回了Window的WindowManagerImpl，

@Override

public Object getSystemService(String name) {

if (WINDOW\_SERVICE.equals(name)) {

return mWindowManager;

} else if (SEARCH\_SERVICE.equals(name)) {

ensureSearchManager();

return mSearchManager;

}

return super.getSystemService(name);

}

之后看一下关键点3，这里传递的token其实就是AMS端传递过来的IApplicationToken代理，一个IBinder对象。之后利用ContextImpl的getSystemService()函数得到一个一个WindowManagerImpl对象，再通过setWindowManager为Activity创建自己的WindowManagerImpl。到这一步，Activity已经准备完毕，剩下的就是在resume中通过addView将窗口添加到到WMS，具体实现在ActivityThread的handleResumeActivity函数中：

final void handleResumeActivity(IBinder token,

boolean clearHide, boolean isForward, boolean reallyResume) {

ActivityClientRecord r = performResumeActivity(token, clearHide);

if (r != null) {

final Activity a = r.activity;

...

if (r.window == null && !a.mFinished && willBeVisible) {

<!--关键点1-->

r.window = r.activity.getWindow();

View decor = r.window.getDecorView();

decor.setVisibility(View.INVISIBLE);

<!--关键点2 获取WindowManager-->

ViewManager wm = a.getWindowManager();

WindowManager.LayoutParams l = r.window.getAttributes();

a.mDecor = decor;

l.type = WindowManager.LayoutParams.TYPE\_BASE\_APPLICATION;

l.softInputMode |= forwardBit;

if (a.mVisibleFromClient) {

a.mWindowAdded = true;

<!--关键点3 添加到WMS管理-->

wm.addView(decor, l);

}

...

}

关键点1是为了获取Activit的Window及DecorView对象，如果用户没有通过setContentView方式新建DecorView，这里会利用PhoneWindow的getDecorView()新建DecorView，

@Override

public final View getDecorView() {

if (mDecor == null) {

installDecor();

}

return mDecor;

}

之后通过Activity的getWindowManager()获取WindowManagerImpl对象，这里获取的WindowManagerImpl其实是Activity自己的WindowManagerImpl，

private WindowManagerImpl(Display display, Window parentWindow) {

mDisplay = display;

mParentWindow = parentWindow;

}

它的mParentWindow 是非空的，获取WindowManagerImpl之后，便利用 addView(decor, l)将DecorView对应的窗口添加到WMS中去，最后调用的是

@Override

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

applyDefaultToken(params);

mGlobal.addView(view, params, mDisplay, mParentWindow);

}

可以看到这里会传递mParentWindow给WindowManagerGlobal对象，作为调整WindowMangaer.LayoutParams 中token的依据：

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

Display display, Window parentWindow) {

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

<!--调整wparams的token参数-->

if (parentWindow != null) {

parentWindow.adjustLayoutParamsForSubWindow(wparams);

}

ViewRootImpl root;

View panelParentView = null;

..

<!--新建ViewRootImpl ,并利用wparams参数添加窗口-->

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

view.setLayoutParams(wparams);

..

<!--新建ViewRootImpl -->

root.setView(view, wparams, panelParentView);

}

parentWindow.adjustLayoutParamsForSubWindow是一个很关键的函数，从名字就能看出，这是为了他调整子窗口的参数：

void adjustLayoutParamsForSubWindow(WindowManager.LayoutParams wp) {

CharSequence curTitle = wp.getTitle();

<!--如果是子窗口如何处理-->

if (wp.type >= WindowManager.LayoutParams.FIRST\_SUB\_WINDOW &&

wp.type <= WindowManager.LayoutParams.LAST\_SUB\_WINDOW) {

<!--后面会看到，其实PopupWindow类的子窗口的wp.token是在上层显示赋值的-->

if (wp.token == null) {

View decor = peekDecorView();

if (decor != null) {

// 这里其实是父窗口的IWindow对象 Window只有Dialog跟Activity才有

wp.token = decor.getWindowToken();

}

}

} else {

<!--这里其实只对应用窗口有用 Activity与Dialog都一样-->

if (wp.token == null) {

wp.token = mContainer == null ? mAppToken : mContainer.mAppToken;

}

}

}

对于Activity来说，wp.token = mContainer == null ? mAppToken : mContainer.mAppToken，其实就是AMS端传过来的IApplicationToken，之后在ViewRootImpl中setView的时候，会利用IWindowSession代理与WMS端的Session通信，将窗口以及token信息传递到WMS端，其中IApplicationToken就是该Activity所处于的分组，在WMS端，会根据IApplicationToken IBinder键值，从全局的mTokenMap中找到对应的AppWindowToken。既然说分组，就应该有其他的子元素，下面看一下Activity上弹出Dialog的流程，进一步了解为什么Activity与它弹出的Dialog是统一分组（复用同一套token）。

**Dialg分组及显示原理：为什么Activity与Dialog算同一组？**

在添加到WMS的时候，Dialog的窗口属性是WindowManager.LayoutParams.TYPE\_APPLICATION，同样属于应用窗口，因此，必须使用Activity的AppToken才行，换句话说，必须使用Activity内部的WindowManagerImpl进行addView才可以。Dialog和Activity共享同一个WindowManager（也就是WindowManagerImpl），而WindowManagerImpl里面有个Window类型的mParentWindow变量，这个变量在Activity的attach中创建WindowManagerImpl时传入的为当前Activity的Window，而Activity的Window里面的mAppToken值又为当前Activity的token，所以Activity与Dialog共享了同一个mAppToken值，只是Dialog和Activity的Window对象不同,下面用代码确认一下：

Dialog(@NonNull Context context, @StyleRes int themeResId, boolean createContextThemeWrapper) {

<!--关键点 1 根据theme封装context-->

if (createContextThemeWrapper) {

...

mContext = new ContextThemeWrapper(context, themeResId);

} else {

mContext = context;

}

<!--获取mWindowManager-->

mWindowManager = (WindowManager) context.getSystemService(Context.WINDOW\_SERVICE);

<!--创建PhoneWindow-->

final Window w = new PhoneWindow(mContext);

mWindow = w;

w.setCallback(this);

w.setOnWindowDismissedCallback(this);

w.setWindowManager(mWindowManager, null, null);

w.setGravity(Gravity.CENTER);

mListenersHandler = new ListenersHandler(this);

}

以上代码先根据Theme调整context，之后利用context.getSystemService(Context.WINDOW\_SERVICE)，这里Dialog是从Activity弹出来的，所以context是Activity，如果你设置Application，会有如下error，至于为什么，后面分析会看到。

android.view.WindowManager$BadTokenException: Unable to add window -- token null is not for an application

at android.view.ViewRootImpl.setView(ViewRootImpl.java:563)

at android.view.WindowManagerGlobal.addView(WindowManagerGlobal.java:269)

at android.view.WindowManagerImpl.addView(WindowManagerImpl.java:69)

接着看Activity的getSystemService，上文分析过这种方法获取的其实是Activity中PhoneWindow的WindowManagerImpl，所以后面利用WindowManagerImpl addView的时候，走的流程与Activity一样。看一下show的代码：

public void show() {

...

onStart();

mDecor = mWindow.getDecorView();

...

<!--关键点 WindowManager.LayoutParams的获取-->

WindowManager.LayoutParams l = mWindow.getAttributes();

...

try {

mWindowManager.addView(mDecor, l);

mShowing = true;

sendShowMessage();

} finally {

}

}

Window在创建的时候，默认新建WindowManager.LayoutParams mWindowAttributes

private final WindowManager.LayoutParams mWindowAttributes =

new WindowManager.LayoutParams();

采用的是无参构造方法，

public LayoutParams() {

super(LayoutParams.MATCH\_PARENT, LayoutParams.MATCH\_PARENT);

type = TYPE\_APPLICATION;

format = PixelFormat.OPAQUE;

}

因此这里的type = TYPE\_APPLICATION，也就是说Dialog的窗口类型其实是应用窗口。因此在addView走到上文的adjustLayoutParamsForSubWindow的时候，仍然按照Activity的WindowManagerImpl addView的方式处理，并利用Activity的PhoneWindow的 adjustLayoutParamsForSubWindow调整参数，赋值给WindowManager.LayoutParams token的值仍然是Activity的IApplicationToken，同样在WMS端，对应就是APPWindowToken，也就是Activity与Dialog属于同一分组。

void adjustLayoutParamsForSubWindow(WindowManager.LayoutParams wp) {

CharSequence curTitle = wp.getTitle();

<!--这里其实只对应用窗口有用 Activity与Dialog都一样-->

if (wp.token == null) {

wp.token = mContainer == null ? mAppToken : mContainer.mAppToken;

}

}

回到之前遗留的一个问题，为什么Dialog用Application作为context不行呢？Dialog的窗口类型属于应用窗口，如果采用Application作为context，那么通过context.getSystemService(Context.WINDOW\_SERVICE)获取的WindowManagerImpl就不是Activity的WindowManagerImpl，而是Application，它同Activity的WindowManagerImpl的区别是没有parentWindow，所以adjustLayoutParamsForSubWindow函数不会被调用，WindowManager.LayoutParams的token就不会被赋值，因此ViewRootImpl在通过setView向WMS在添加窗口的时候会失败：

public int addWindow(Session session, IWindow client, XXX )

...

<!--对于应用窗口 token不可以为null-->

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;

}

WMS会返回WindowManagerGlobal.ADD\_BAD\_APP\_TOKEN的错误给APP端，APP端ViewRootImpl端收到后会抛出如下异常

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

synchronized (this) {

....

case WindowManagerGlobal.ADD\_NOT\_APP\_TOKEN:

throw new WindowManager.BadTokenException(

"Unable to add window -- token " + attrs.token

+ " is not for an application");

以上就为什么不能用Application作为Dialog的context的理由（**不能为Dialog提供正确的token**），接下来看一下PopupWindow是如何处理分组的。

**PopupWindow类子窗口的添加流程及WindowToken分组**

PopupWindow是最典型的子窗口，必须依附父窗口才能存在，先看下PopupWindow一般用法：

View root = LayoutInflater.from(AppProfile.getAppContext()).inflate(R.layout.pop\_window, null);

PopupWindow popupWindow = new PopupWindow(root, ViewGroup.LayoutParams.MATCH\_PARENT, ViewGroup.LayoutParams.WRAP\_CONTENT, true);

popupWindow.setBackgroundDrawable(new BitmapDrawable());

popupWindow.showAsDropDown(archorView);

PopupWindow的构造函数很普通，主要是一些默认入场、出厂动画的设置，如果在新建PopupWindow的时候已经将根View传递到构造函数中去，PopupWindow的构造函数会调用setContentView，如果在show之前，没有调用setContentView，则抛出异常。

public PopupWindow(View contentView, int width, int height, boolean focusable) {

if (contentView != null) {

mContext = contentView.getContext();

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

}

setContentView(contentView);

setWidth(width);

setHeight(height);

setFocusable(focusable);

}

下面主要看PopupWindow的showAsDropDown函数

public void showAsDropDown(View anchor, int xoff, int yoff, int gravity) {

<!--关键点1 利用通过View锚点所在窗口显性构建PopupWindow的token-->

final WindowManager.LayoutParams p = createPopupLayoutParams(anchor.getWindowToken());

<!--关键点2-->

preparePopup(p);

...

<!--关键点3-->

invokePopup(p);

}

showAsDropDown有3个关键点，关键点1是生成WindowManager.LayoutParams参数，WindowManager.LayoutParams参数里面的type、token是非常重要参数，**PopupWindow的type是TYPE\_APPLICATION\_PANEL = FIRST\_SUB\_WINDOW**，是一个子窗口。关键点2是PopupDecorView的生成，这个View是PopupWindow的根ViewGroup，类似于Activity的DecorView，关键3利用WindowManagerService的代理，将View添加到WMS窗口管理中去显示，先看关键点1：

private WindowManager.LayoutParams createPopupLayoutParams(IBinder token) {

final WindowManager.LayoutParams p = new WindowManager.LayoutParams();

p.gravity = computeGravity();

p.flags = computeFlags(p.flags);

p.type = mWindowLayoutType;

<!--显性赋值token-->

p.token = token;

p.softInputMode = mSoftInputMode;

p.windowAnimations = computeAnimationResource();

if (mBackground != null) {

p.format = mBackground.getOpacity();

} else {

p.format = PixelFormat.TRANSLUCENT;

}

..

p.privateFlags = PRIVATE\_FLAG\_WILL\_NOT\_REPLACE\_ON\_RELAUNCH

| PRIVATE\_FLAG\_LAYOUT\_CHILD\_WINDOW\_IN\_PARENT\_FRAME;

return p;

}

上面的Token其实用的是anchor.getWindowToken()，**如果是Activity中的View，其实用的Token就是Activity的ViewRootImpl中的IWindow对象**，如果这个View是一个系统窗口中的View，比如是Toast窗口中弹出来的，用的就是Toast ViewRootImpl的IWindow对象，归根到底，PopupWindow自窗口中的Token是ViewRootImpl的IWindow对象，同Activity跟Dialog的token（IApplicationToken）不同，该Token标识着PopupWindow在WMS所处的分组，最后来看一下PopupWindow的显示:

private void invokePopup(WindowManager.LayoutParams p) {

if (mContext != null) {

p.packageName = mContext.getPackageName();

}

final PopupDecorView decorView = mDecorView;

decorView.setFitsSystemWindows(mLayoutInsetDecor);

setLayoutDirectionFromAnchor();

<!--关键点1-->

mWindowManager.addView(decorView, p);

if (mEnterTransition != null) {

decorView.requestEnterTransition(mEnterTransition);

}

}

主要是调用了WindowManager的addView添加视图并显示，这里首先需要关心一下mWindowManager，

mWindowManager = (WindowManager) context.getSystemService(Context.WINDOW\_SERVICE);

这的context 可以是Activity，也可以是Application，因此WindowManagerImpl也可能不同，不过这里并没有多大关系，因为PopupWindow的token是显性赋值的，就是是就算用Application，也不会有什么问题，对于PopupWindow子窗口，关键点是View锚点决定其token，而不是WindowManagerImpl对象：

@Override

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

applyDefaultToken(params);

mGlobal.addView(view, params, mContext.getDisplay(), mParentWindow);

}

之后利用ViewRootImpl的setView函数的时候，WindowManager.LayoutParams里的token其实就是view锚点获取的IWindow对象，WindowManagerService在处理该请求的时候，

public int addWindow(Session session, IWindow client, XXX ) {

<!--关键点1，必须找到子窗口的父窗口，否则添加失败-->

WindowState attachedWindow = null;

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

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

if (attachedWindow == null) {

return WindowManagerGlobal.ADD\_BAD\_SUBWINDOW\_TOKEN;

}

}

<!--关键点2 如果Activity第一次添加子窗口 ，子窗口分组对应的WindowToken一定是null-->

boolean addToken = false;

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

AppWindowToken atoken = null;

if (token == null) {

...

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

addToken = true;

}

<!--关键点2 新建窗口WindowState对象 注意这里的attachedWindow非空-->

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

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

...

<!--关键点4 添加更新全部map，-->

if (addToken) {

mTokenMap.put(attrs.token, token);

}

mWindowMap.put(client.asBinder(), win);

}

从上面的分析可以看出，WMS会为PopupWindow窗口创建一个子窗口分组WindowToken，每个子窗口都会有一个指向父窗口的引用，因为是利用父窗口的IWindow作为键值，父窗口可以很方便的利用自己的IWindow获取WindowToken，进而得到全部的子窗口，

关于系统窗口，前文层分析过Toast系统窗口，Toast类系统窗口在WMS端只有一个WindowToken，键值是null，这个比较奇葩，不过还没验证过。

**窗口的Z次序管理：窗口的分配序号、次序调整等**

虽然我们看到的手机屏幕只是一个二维平面X\*Y，但其实Android系统是有隐形的Z坐标轴的，其方向与手机屏幕垂直，与我们的实现平行，所以并不能感知到。

Z order.jpg

前面分析了窗口分组的时候涉及了两个对象WindowState与Windtoken，但仅限分组，分组无法决定窗口的显示的Z-order，那么再WMS是怎么管理所有窗口的Z-order的？ 在WMS中窗口被抽象成WindowState，因此WindowState内部一定有属性来标志这个窗口的Z-order，实现也确实如此，WindowState采用三个个int值mBaseLayer+ mSubLayer + mLayer 来标志窗口所处的位置，前两个主要是根据窗口类型确定窗口位置，mLayer才是真正的值，定义如下：

final class WindowState implements WindowManagerPolicy.WindowState {

final WindowList mChildWindows = new WindowList();

final int mBaseLayer;

final int mSubLayer;

<!--最终Z次序的赋值-->

int mLayer;

}

从名字很容知道mBaseLayer是标志窗口的主次序，面向的是一个窗口组，而mSubLayer主要面向单独窗口，要来标志一个窗口在一组窗口中的位置，对两者来说值越大，窗口越靠前，从此final属性知道，两者的值是不能修改的，而mLayer可以修改，对于系统窗口，一般不会同时显示两个，因此，可以用主序决定，比较特殊的就是Activity与子窗口，首先子窗口的主序肯定是父窗口决定的，子窗口只关心次序就行。而父窗口的主序却相对麻烦，比如对于应用窗口来说，他们的主序都是一样的，因此还要有一个其他的维度来作为参考，比如**对于Activity，主序都是一样的，怎么定他们真正的Z-order呢？其实Activity的顺序是由AMS保证的，这个顺序定了，WMS端Activity窗口的顺序也是定了，这样下来次序也方便定了**。

WindowState(WindowManagerService service, Session s, IWindow c, WindowToken token,

WindowState attachedWindow, int appOp, int seq, WindowManager.LayoutParams a,

int viewVisibility, final DisplayContent displayContent) {

...

<!--关键点1 子窗口类型的Z order-->

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

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

mBaseLayer = mPolicy.windowTypeToLayerLw(

attachedWindow.mAttrs.type) \* WindowManagerService.TYPE\_LAYER\_MULTIPLIER

+ WindowManagerService.TYPE\_LAYER\_OFFSET;

mSubLayer = mPolicy.subWindowTypeToLayerLw(a.type);

mAttachedWindow = attachedWindow; final WindowList childWindows = mAttachedWindow.mChildWindows;

final int numChildWindows = childWindows.size();

if (numChildWindows == 0) {

childWindows.add(this);

} else {

...

} else {

<!--关键点2 普通窗口类型的Z order-->

mBaseLayer = mPolicy.windowTypeToLayerLw(a.type)

\* WindowManagerService.TYPE\_LAYER\_MULTIPLIER

+ WindowManagerService.TYPE\_LAYER\_OFFSET;

mSubLayer = 0;

mAttachedWindow = null;

mLayoutAttached = false;

}

...

}

由于窗口所能选择的类型是确定的，因此mBaseLayer与mSubLayer所能选择的值只有固定几个，很明显这两个参数不能精确的确定Z-order，还会有其他微调的手段，也仅限微调，**在系统层面，决定了不同类型窗口所处的位置，比如系统Toast类型的窗口一定处于所有应用窗口之上**，不过我们最关心的是Activity类的窗口如何确定Z-order的，在new WindowState之后，只是粗略的确定了Activity窗口的次序，看一下添加窗口的示意代码：

addWindow(){

<!--1-->

new WindowState

<!--2-->

addWindowToListInOrderLocked(win, true);

<!--3-->

assignLayersLocked(displayContent.getWindowList());

}

新建state对象之后，Z-order还要通过addWindowToListInOrderLocked及assignLayersLocked才能确定，addWindowToListInOrderLocked主要是根据窗口的Token找到归属，插入到对应Token的WindowState列表，如果是子窗口还要插入到父窗口的对应位置中：

次序确定.jpg

插入到特定位置后其实Z-order就确定了，接下来就是通过assignLayersLocked为WindowState分配真正的Z-order mLayer,

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)) {

<!--通过偏移量-->

curLayer += WINDOW\_LAYER\_MULTIPLIER;

w.mLayer = curLayer;

} else {

curBaseLayer = curLayer = w.mBaseLayer;

w.mLayer = curLayer;

}

if (w.mLayer != oldLayer) {

layerChanged = true;

anyLayerChanged = true;

}

...

}

mLayer最终确定后，窗口的次序也就确定了，这个顺序要最终通过后续的relayout更新到SurfaceFlinger服务，之后，SurfaceFlinger在图层混排的时候才知道如何处理。

**WMS中窗口次序分配如何影响SurfaceFlinger服务**

SurfaceFlinger在图层混排的时候应该不会混排所有的窗口，只会混排可见的窗口，比如有多个全屏Activity的时候，SurfaceFlinger只会处理最上面的，那么SurfaceFlinger如何知道哪些窗口可见哪些不可见呢？前文分析了WMS分配Z-order之后，要通过setLayer更新到SurfaceFlinger，接下来看具体流程，创建SurfaceControl之后，会创建一次事务，确定Surface的次序：

SurfaceControl.openTransaction();

try {

mSurfaceX = left;

mSurfaceY = top;

try {

mSurfaceControl.setPosition(left, top);

mSurfaceLayer = mAnimLayer;

final DisplayContent displayContent = w.getDisplayContent();

if (displayContent != null) {

mSurfaceControl.setLayerStack(displayContent.getDisplay().getLayerStack());

}

<!--设置次序-->

mSurfaceControl.setLayer(mAnimLayer);

mSurfaceControl.setAlpha(0);

mSurfaceShown = false;

} catch (RuntimeException e) {

mService.reclaimSomeSurfaceMemoryLocked(this, "create-init", true);

}

mLastHidden = true;

} finally {

SurfaceControl.closeTransaction();

}

}

这里通过openTransaction与closeTransaction保证一次事务的完整性，中间就Surface次序的调整，closeTransaction会与SurfaceFlinger通信，通知SurfaceFlinger更新Surface信息，这其中就包括Z-order。

**总结**

本文简要分析了Android窗口的分组，以及WMS窗口次序的确定，最后简单提及了一下窗口次序如何更新到SurfaceFlinger服务的，也方便将来理解图层合成。

作者：看书的小蜗牛  
原文链接:[Android窗口管理分析（3）：窗口分组及Z-order的确定](https://www.jianshu.com/p/90ede7b2a64a)  
**仅供参考，欢迎指正**

准备的说是在创建Surface过后，

1. WindowSurfaceController createSurfaceLocked() {
2. ...
3. *//layer stack表示是该Surface显示在哪个显示设备上，*
4. final int layerStack = w.getDisplayContent().getDisplay().getLayerStack();
5. *//将 layerstack与 z -order值(mAnimlayer)设置到Surface对应的SurfaceFlinger中的Layer中*
6. mSurfaceController.setPositionAndLayer(mTmpSize.left, mTmpSize.top, layerStack, mAnimLayer);
7. mLastHidden = true;
8. }

最后通过Layer.cpp中的setLayer函数将Z order值保存到Layer的mCurrentState中的z变量中。

1. bool Layer::setLayer(uint32\_t z) {
2. if (mCurrentState.z == z)
3. return false;
4. mCurrentState.sequence++;
5. mCurrentState.z = z;
6. mCurrentState.modified = true;
7. setTransactionFlags(eTransactionNeeded);
8. return true;
9. }

**【Android】WMS和AMS初探**

[](https://www.jianshu.com/u/4cb5b23aa64a)

[bersaty](https://www.jianshu.com/u/4cb5b23aa64a) 关注

 0.1 2018.12.06 15:13 字数 655 阅读 375评论 2喜欢 2

代码基于Android 9.0

**1、wms的启动**

SystemServer的main函数，启动了SystemServer线程，同时启动了各种服务，其中的startOtherServices启动了对应的AMS，IMS，WMS等系统服务。

SystemServer.java

traceBeginAndSlog("StartInputManagerService");

//新建IMS

inputManager = new InputManagerService(context);

traceEnd();

traceBeginAndSlog("StartWindowManagerService");

// WMS needs sensor service ready

ConcurrentUtils.waitForFutureNoInterrupt(mSensorServiceStart, START\_SENSOR\_SERVICE);

mSensorServiceStart = null;

//初始化WMS，绑定IMS

wm = WindowManagerService.main(context, inputManager,

mFactoryTestMode != FactoryTest.FACTORY\_TEST\_LOW\_LEVEL,

!mFirstBoot, mOnlyCore, new PhoneWindowManager());

ServiceManager.addService(Context.WINDOW\_SERVICE, wm, /\* allowIsolated= \*/ false,

DUMP\_FLAG\_PRIORITY\_CRITICAL | DUMP\_FLAG\_PROTO);

ServiceManager.addService(Context.INPUT\_SERVICE, inputManager,

/\* allowIsolated= \*/ false, DUMP\_FLAG\_PRIORITY\_CRITICAL);

traceEnd();

traceBeginAndSlog("SetWindowManagerService");

//关联AMS和WMS

mActivityManagerService.setWindowManager(wm);

traceEnd();

traceBeginAndSlog("WindowManagerServiceOnInitReady");

//初始化WMP

wm.onInitReady();

traceEnd();

WMS的main函数和InitPolicy函数都执行了handler的runWithScissors，该方法会执行新开的线程，然后阻塞当前线程，直到新开线程执行完毕。WMS单例的创建是在DisplayThread进行，WindowManagerPolicy（具体实现是PhoneWindowManager）的初始化是在UiThread进行。

WindowManagerService.java

public static WindowManagerService main(final Context context, final InputManagerService im,

final boolean haveInputMethods, final boolean showBootMsgs, final boolean onlyCore,

WindowManagerPolicy policy) {

//运行在DisplayThread

DisplayThread.getHandler().runWithScissors(() ->

sInstance = new WindowManagerService(context, im, haveInputMethods, showBootMsgs,

onlyCore, policy), 0);

return sInstance;

}

private void initPolicy() {

//运行在UiThread

UiThread.getHandler().runWithScissors(new Runnable() {

@Override

public void run() {

WindowManagerPolicyThread.set(Thread.currentThread(), Looper.myLooper());

mPolicy.init(mContext, WindowManagerService.this, WindowManagerService.this);

}

}, 0);

}

**2、AMS和WMS关联**

Activity启动的时候，ActivityStarter.StartActivity会创建一个ActivityRecord。

ActivityStarter.java

private int startActivity(IApplicationThread caller, Intent intent, Intent ephemeralIntent,

String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,

IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,

IBinder resultTo, String resultWho, int requestCode, int callingPid, int callingUid,

String callingPackage, int realCallingPid, int realCallingUid, int startFlags,

SafeActivityOptions options,

boolean ignoreTargetSecurity, boolean componentSpecified, ActivityRecord[] outActivity,

TaskRecord inTask, boolean allowPendingRemoteAnimationRegistryLookup) {

...

ActivityRecord r = new ActivityRecord(mService, callerApp, callingPid, callingUid,

callingPackage, intent, resolvedType, aInfo, mService.getGlobalConfiguration(),

resultRecord, resultWho, requestCode, componentSpecified, voiceSession != null,

mSupervisor, checkedOptions, sourceRecord);

...

}

ActivityRecord创建的时候会创建一个appToken（IApplicationToken），通过该token创建AppWindowContainerController，创建时传入AppWindowContainerListener回调，WMS的单例，初始化mRoot。把ActivityRecord传过来的appToken转化成AppWindowToken，AppWindowToken继承WindowToken，会调用onDisplayChanged(dc)，通过DisplayContent最终会调用WMS的addWindowToken方法。通过win.attach()保存Seesion和创建SurfaceSession。

Session.java

public SurfaceSession() {

mNativeClient = nativeCreate();

}

//这里创建的是SurfaceComposerClient，

static jlong nativeCreate(JNIEnv\* env, jclass clazz) {

SurfaceComposerClient\* client = new SurfaceComposerClient();

client->incStrong((void\*)nativeCreate);

return reinterpret\_cast<jlong>(client);

}

DisplayContent的tokenMap保存了ActivityRecord的token，key为token，value为WindowToken 。  
AppWindowToken持有WMS，所以可以直接调用WMS的方法。  
总结:ActivityRecord通过AppWindowContainerController持有的WMS直接执行方法（AppWindowContainerController和AppWindowToken都继承WindowContainer，持有WMS），实现AMS和WMS通讯。

ActivityRecord.java

public AppWindowContainerController(TaskWindowContainerController taskController,

IApplicationToken token, AppWindowContainerListener listener, int index,

int requestedOrientation, boolean fullscreen, boolean showForAllUsers, int configChanges,

boolean voiceInteraction, boolean launchTaskBehind, boolean alwaysFocusable,

int targetSdkVersion, int rotationAnimationHint, long inputDispatchingTimeoutNanos,

WindowManagerService service) {

//初始化mRoot

super(listener, service);

mHandler = new H(service.mH.getLooper());

mToken = token;

synchronized(mWindowMap) {

AppWindowToken atoken = mRoot.getAppWindowToken(mToken.asBinder());

if (atoken != null) {

// TODO: Should this throw an exception instead?

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

return;

}

final Task task = taskController.mContainer;

if (task == null) {

throw new IllegalArgumentException("AppWindowContainerController: invalid "

+ " controller=" + taskController);

}

//把IApplicationToken转化成AppWindowToken

atoken = createAppWindow(mService, token, voiceInteraction, task.getDisplayContent(),

inputDispatchingTimeoutNanos, fullscreen, showForAllUsers, targetSdkVersion,

requestedOrientation, rotationAnimationHint, configChanges, launchTaskBehind,

alwaysFocusable, this);

if (DEBUG\_TOKEN\_MOVEMENT || DEBUG\_ADD\_REMOVE) Slog.v(TAG\_WM, "addAppToken: " + atoken

+ " controller=" + taskController + " at " + index);

//添加token到task里面，task也是WindwContainer

task.addChild(atoken, index);

}

}

**3、Activity与WMS关联**

启动Activity的时候，ActivityThread执行到performLaunchActivity时，创建activity对象，调用Activity.attach方法，这里创建了一个PhoneWindow和关联了WindowManager。

ActivityThread.java

/\*\* Core implementation of activity launch. \*/

private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {

...

activity.attach(appContext, this, getInstrumentation(), r.token,

r.ident, app, r.intent, r.activityInfo, title, r.parent,

r.embeddedID, r.lastNonConfigurationInstances, config,

r.referrer, r.voiceInteractor, window, r.configCallback);

...

}

handleResumeActivity时把PhoneWindow里面的DecorView设置成不可见，最后调用Activity的makevisible。

ActivityThread.java

@Override

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

String reason) {

...

if (r.window == null && !a.mFinished && willBeVisible) {

r.window = r.activity.getWindow();

View decor = r.window.getDecorView();

//设置DecorView不可见

decor.setVisibility(View.INVISIBLE);

ViewManager wm = a.getWindowManager();

WindowManager.LayoutParams l = r.window.getAttributes();

a.mDecor = decor;

l.type = WindowManager.LayoutParams.TYPE\_BASE\_APPLICATION;

l.softInputMode |= forwardBit;

if (r.mPreserveWindow) {

a.mWindowAdded = true;

r.mPreserveWindow = false;

// Normally the ViewRoot sets up callbacks with the Activity

// in addView->ViewRootImpl#setView. If we are instead reusing

// the decor view we have to notify the view root that the

// callbacks may have changed.

ViewRootImpl impl = decor.getViewRootImpl();

if (impl != null) {

impl.notifyChildRebuilt();

}

}

if (a.mVisibleFromClient) {

if (!a.mWindowAdded) {

a.mWindowAdded = true;

//把decorview添加到wm,具体实现是在WindowManagerImpl里

wm.addView(decor, l);

} else {

// The activity will get a callback for this {@link LayoutParams} change

// earlier. However, at that time the decor will not be set (this is set

// in this method), so no action will be taken. This call ensures the

// callback occurs with the decor set.

a.onWindowAttributesChanged(l);

}

}

...

if (r.activity.mVisibleFromClient) {

//activity设置成可见的

r.activity.makeVisible();

}

...

}

WindowManager（WindowManagerImpl）添加DecorView，实际是调用了WindowManagerGolbal的addView。

WindowManagerGlobal.java

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

Display display, Window parentWindow) {

...

ViewRootImpl root;

View panelParentView = null;

...

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

view.setLayoutParams(wparams);

//所有的view对象

mViews.add(view);

//viewrootimpl

mRoots.add(root);

//对应view的参数

mParams.add(wparams);

// do this last because it fires off messages to start doing things

try {

//viewRootImpl关联DecorView

root.setView(view, wparams, panelParentView);

} catch (RuntimeException e) {

// BadTokenException or InvalidDisplayException, clean up.

if (index >= 0) {

removeViewLocked(index, true);

}

throw e;

}

}

}

这里创建了ViewRootImpl，其初始化创建Session，mWindow，attachInfo等，然后通过ViewRootImpl的setView把DecorView添加上去，setView会调用addToDisplay，传递了Session和Window（IWindow），InputChanel。

ViewRootImpl.java

public ViewRootImpl(Context context, Display display) {

mContext = context;

//进程单例

mWindowSession = WindowManagerGlobal.getWindowSession();

mDisplay = display;

...

//IWindow的binder对象，属于ViewRootImpl

mWindow = new W(this);

...

//AttachInfo比较重要的

mAttachInfo = new View.AttachInfo(mWindowSession, mWindow, display, this, mHandler, this,

context);

...

}

/\*\*

\* We have one child

\*/

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

....

try {

mOrigWindowType = mWindowAttributes.type;

mAttachInfo.mRecomputeGlobalAttributes = true;

collectViewAttributes();

//此处最终调用了WindowManagerService的addWindow方法

res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,

getHostVisibility(), mDisplay.getDisplayId(), mWinFrame,

mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,

mAttachInfo.mOutsets, mAttachInfo.mDisplayCutout, mInputChannel);

} catch (RemoteException e) {

mAdded = false;

mView = null;

mAttachInfo.mRootView = null;

mInputChannel = null;

mFallbackEventHandler.setView(null);

unscheduleTraversals();

setAccessibilityFocus(null, null);

throw new RuntimeException("Adding window failed", e);

} finally {

if (restore) {

attrs.restore();

}

}

...

}

这里最后是调用了WindowManagerService的addWindow（通过Session的IPC），同时传递了Session和mWindow。这里的Session由于是WindowManagerGolbal单例来的，所以对应的是app的，mWindow对应的是ViewRootImpl的。这里还创建了相应的token。

WindowManagerService.java

public int addWindow(Session session, IWindow client, int seq,

LayoutParams attrs, int viewVisibility, int displayId, Rect outFrame,

Rect outContentInsets, Rect outStableInsets, Rect outOutsets,

DisplayCutout.ParcelableWrapper outDisplayCutout, InputChannel outInputChannel) {

...

//省略部分通过各种逻辑创建对应的token

//创建对应的WindowState

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

appOp[0], seq, attrs, viewVisibility, session.mUid,

session.mCanAddInternalSystemWindow);

...

//这里调用了mSession.put(session)

win.attach();

mWindowMap.put(client.asBinder(), win);

....

}

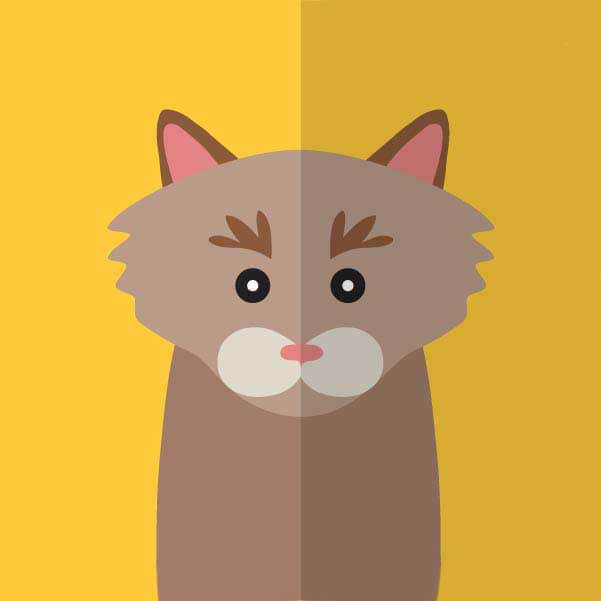
WMS里面的windowMap保存的是key为mWindow，value为生成的WindowState，也就是间接的ViewRootImpl。WMS同时也保存了Session在数组里。

app和WMS的关系图

**想到几个问题，后续再研究解决**

1、status bar，wallpaper，dialog，popupwindow，Navigation等窗口的显示流程逻辑。  
2、Activity切换动画过程。  
3、输入事件和window的交互，位置大小变化。  
4、多任务上滑的是否为多窗口。  
5、窗口的移除。

**WMS中的Z order**

[](https://www.jianshu.com/u/38c34e595b1c)

[wbo4958](https://www.jianshu.com/u/38c34e595b1c) 关注

2017.07.13 16:01\* 字数 1024 阅读 406评论 0喜欢 1

转载请标注: [http://www.jianshu.com/p/5e802482caa4](https://www.jianshu.com/p/5e802482caa4)  
始终觉得要理解一个庞大的东西，需要将它分解，从部分理解开始，慢慢理解了，那么这个庞大的东西也就理解。

在看这篇文章之前，非常建议参考[AMS/WMS/APP 中Token惟一性](https://www.jianshu.com/p/5e2efbaa2949), 因为这里面会涉及到各种Token,稍不注意就不知道具体的token是指什么了。

再把这张图贴在这里， 这样看代码时，对照着看就更容易理解了。

Token的惟一性

**注意:** 本篇以Launcher启动一个App (MainActivity)时，来学习Z order. 另外由于在启动一个App时会有一个starting Window, 略过这部分。

**一、加入Window到WMS的窗口堆栈**

public int addWindow(Session session, IWindow client, int seq,

WindowManager.LayoutParams attrs, int viewVisibility, int displayId,

Rect outContentInsets, Rect outStableInsets, Rect outOutsets,

InputChannel outInputChannel) {

final int type = attrs.type;

//attrs即Activity的Window中的LayoutParams, 这里值是FIRST\_APPLICATION\_WINDOW

synchronized(mWindowMap) {

//displayId为具体的显示屏，一般没有外接都是默认的显示屏

//DisplayContent为显示屏上显示的内容，主要是一些Window对应的WindowState

final DisplayContent displayContent = getDisplayContentLocked(displayId);

//如果WindowMap已经加入过了，就直接返回，不用再生成对应的WindowState了

if (mWindowMap.containsKey(client.asBinder())) {

return WindowManagerGlobal.ADD\_DUPLICATE\_ADD;

}

boolean addToken = false;

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

//获得Activity对应在WMS中的WindowToken, 在这里已经不为空了，具体见[2. WMS中的token]

AppWindowToken atoken = null;

if (token == null) {

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

atoken = token.appWindowToken; //获得AppWindowToken

} else (XXXX){

}

//生成对应的WindowState用于保存Window信息

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

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

res = WindowManagerGlobal.ADD\_OKAY;

if (addToken) {

mTokenMap.put(attrs.token, token); //这个在addAppToken中已经加入过了，所以不用加入了

}

win.attach(); //Session里 mNumWindow加1，表示现在总共有多少window了

mWindowMap.put(client.asBinder(), win); //储存WindowState

if (type == TYPE\_INPUT\_METHOD) {//输入法窗口，这里不讨论

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

} else {

addWindowToListInOrderLocked(win, true); //将WindowState加入到窗口堆栈中

}

}

return res;

}

**1.1 Window的类型**

addWindow最后会根据不同的Window类型做不同的处理，那这些Window类型是什么呢？

**attrs.type** 这个指明 Window的类型, 它的值在如下代码中初始化

private final WindowManager.LayoutParams mWindowAttributes = new WindowManager.LayoutParams();

public LayoutParams() {

super(LayoutParams.MATCH\_PARENT, LayoutParams.MATCH\_PARENT);

type = TYPE\_APPLICATION; //默认为TYPE\_APPLICATION类型

format = PixelFormat.OPAQUE;

}

由代码可知，LayoutParams默认的Type是TYPE\_APPLICATION = 2; 即普通的应用窗口

但是会在**handleResumeActivity**被修改掉，如下

--- in handleResumeActivity

WindowManager.LayoutParams l = r.window.getAttributes();

a.mDecor = decor;

l.type = WindowManager.LayoutParams.TYPE\_BASE\_APPLICATION; //修改窗口类型

wm.addView(decor, l);

但是在addView之前修改了窗口类型为TYPE\_BASE\_APPLICATION = 1类型，这个类型是基类，是Activity中最底层的窗口，换句话说，其它的窗口都会叠加在该窗口之上，比如Dialog窗口(它使用默认的TYPE\_APPLICATION)。 这也说明type值越大，它就越在上面。

Android定义的窗口值如下:

* **应用程序窗口**

| **定义** | **值** | **意义** |
| --- | --- | --- |
| FIRST\_APPLICATION\_WINDOW | 1 | 第一个窗口，也就是TYPE\_BASE\_APPLICATION窗口 |
| TYPE\_BASE\_APPLICATION | 1 | Activity的基窗口 |
| TYPE\_BASE\_APPLICATION | 2 | 默认窗口类型 |
| TYPE\_APPLICATION\_STARTING | 3 | 应用程序启动过程中的中间窗口，当真下窗口适配完会关闭该窗口 |
| LAST\_APPLICATION\_WINDOW | 99 | 应用程序最后一个窗口 |

* **子窗口**
* **系统窗口**  
  其它两种窗口请参考 [Android入门之窗口类型](https://link.jianshu.com/?t=http://blog.csdn.net/geloin/article/details/42779025)([http://blog.csdn.net/geloin/article/details/42779025](https://link.jianshu.com/?t=http://blog.csdn.net/geloin/article/details/42779025))

**Question:** 如果将type设为大于99会怎么呢？

在WMS的addWindow会对type进行权限检查

int res = mPolicy.checkAddPermission(attrs, appOp);

即如果type不属于 应用窗口/子窗口/系统窗口，直接会报异常。

而真正将Window加入到窗口堆栈的函数是 **addWindowToListInOrderLocked**

**1.2 addWindowToListInOrderLocked**

private void addWindowToListInOrderLocked(final WindowState win, boolean addToToken) {

if (win.mAttachedWindow == null) { //新启动的Activity, 没有attach的Window

final WindowToken token = win.mToken; //WindowState中的mToken

int tokenWindowsPos = 0;

if (token.appWindowToken != null) { //这里显然不为null,在生成AppWindowToken时指向 见图p2

tokenWindowsPos = addAppWindowToListLocked(win); //进入这个分支

}

}

private int addAppWindowToListLocked(final WindowState win) {

final DisplayContent displayContent = win.getDisplayContent();

final IWindow client = win.mClient;

final WindowToken token = win.mToken;

final WindowList windows = displayContent.getWindowList(); //得到DisplayContent中所有的WindowState,

此时窗口堆栈顺序如下:

窗口堆栈顺序

WindowList tokenWindowList = getTokenWindowsOnDisplay(token, displayContent);

//当前Activity MainActivity是第一次启动，所以它并没有其它Window

final ArrayList<Task> tasks = displayContent.getTasks(); //这里会得到所有的task

此时的Stack\_Task的关系

tasks的值

接着看addAppWindowToListLocked后面的代码

int taskNdx;

int tokenNdx = -1;

for (taskNdx = tasks.size() - 1; taskNdx >= 0; --taskNdx) { //当taskNdx = 1时

AppTokenList tokens = tasks.get(taskNdx).mAppTokens; //此时的task里仅有一个MainActivity中的AppWindowToken

for (tokenNdx = tokens.size() - 1; tokenNdx >= 0; --tokenNdx) { //tokenNdx = 0

final AppWindowToken t = tokens.get(tokenNdx); //此时t 为 MainActivity中的AppWindowToken

if (t == token) { //相等

--tokenNdx; //此时tokenNdx = -1

if (tokenNdx < 0) { //进入此分支

--taskNdx; //寻找前一个task, 此时taskNdx = 0

if (taskNdx >= 0) { //进入该分支

//Launcher Task中的也仅有一个App token, 这里减1的目的是当 add 的时候，以tokenNdx为索引，加在后面

tokenNdx = tasks.get(taskNdx).mAppTokens.size() - 1;

}

}

break;

}

}

if (tokenNdx >= 0) { //此时tokenNdx = 0, 直接break掉

break;

}

}

// Continue looking down until we find the first

// token that has windows on this display.

//此时taskNdx = 0, tokenNdx = 0

for ( ; taskNdx >= 0; --taskNdx) {

AppTokenList tokens = tasks.get(taskNdx).mAppTokens; //获得Launcher Task中所有的tokens

for ( ; tokenNdx >= 0; --tokenNdx) {

final AppWindowToken t = tokens.get(tokenNdx); //这里获得的是Launcher的token

tokenWindowList = getTokenWindowsOnDisplay(t, displayContent); //获得Launcher token中的WinState

final int NW = tokenWindowList.size(); //这里NW为1，因为没有其它子窗口

if (NW > 0) {

pos = tokenWindowList.get(NW-1); //直接获得最后的win state (这里也是Launcher的WinState)

break;

}

}

}

if (pos != null) { //pos已经不为空了

WindowToken atoken = mTokenMap.get(pos.mClient.asBinder()); //

if (atoken != null) { //获得Launcher的 WindowToken

final int NC = atoken.windows.size();

if (NC > 0) {

WindowState top = atoken.windows.get(NC-1);

if (top.mSubLayer >= 0) {

pos = top;

}

}

}

placeWindowAfter(pos, win); //在Launcher后插入Win (MainActivity)

return tokenWindowsPos; //返回

}

}

此时窗口堆栈顺序

**二、Z -order**

前面已经将Window加入到了窗口堆栈，并不是Z order,下面来看下什么是Z order。

由于前面窗口堆栈已经新加入了一个window, 即窗口堆栈变化了，那么此时就要计算 Z order了  
具体是在 addWindow的最后了

--- in addWindow()

mLayersController = new WindowLayersController(this);

WMS初始化时初始化WindowLayersController, 该类主要是为DisplayContent里的Windows也就是Window堆栈指定layer, 也就是计算它们的Z order。 计算Z order时，从窗口堆栈的底部到顶部，层级越高，那么它的 layer 越大，即它的Z order越大。

再继续讲计算 Z -order之前先简单看下与Z order有关的几个变量, 具体在WindowState中

--- in WindowState

final WindowStateAnimator mWinAnimator;

int mLayer;

final int mBaseLayer;

final int mSubLayer;

WindowStateAnimator 是一个Window的动画以及对Surface的操作类，它类中有个Animator的 Z order值，

* mLayer  
  mLayer是Base的Z -order值，它的值是动态算出来的。
* mAnimLayer  
  它的值是由mLayer算出来的, SurfaceFlinger中的Layer的Z order也就是mAnimLayer。

可以从定义看出 mBaseLayer, mSubLayer是final, 是不可再重新赋值的，所以它们的值是固定的，很显然是在WindowState初始化时赋值的。

--- in WindowState构造函数中

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

//子窗口，不讨论

} else {

mBaseLayer = mPolicy.windowTypeToLayerLw(a.type)

\* WindowManagerService.TYPE\_LAYER\_MULTIPLIER //该值是10000

+ WindowManagerService.TYPE\_LAYER\_OFFSET; //该值为1000

mSubLayer = 0;

}

public int windowTypeToLayerLw(int type) {

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

return 2;

}

}

**以非子窗口为例**  
此时mSubLayer=0, 而mBaseLayer与Window类型相关，以本文例子为例, 应用程序的窗口映射的z-order为2, 所以经过计算, mBaseLayer就为21000

**注意:并不代表 Window type值越大，它的Z -order值越大，type值会被windowTypeToLayerLw重新映射。**

下面来看下是怎么计算z order值的

mLayersController.assignLayersLocked(displayContent.getWindowList());

final void assignLayersLocked(WindowList windows) {

int curBaseLayer = 0; //临时的基值

int curLayer = 0; //当前的值

boolean anyLayerChanged = false; //只要一个Z order值变了，这里就会为true

for (int i = 0, windowCount = windows.size(); i < windowCount; i++) {

final WindowState w = windows.get(i);

boolean layerChanged = false; //针对每一个window

int oldLayer = w.mLayer;

if (w.mBaseLayer == curBaseLayer || w.mIsImWindow || (i > 0 && w.mIsWallpaper)) {

curLayer += WINDOW\_LAYER\_MULTIPLIER; //WINDOW\_LAYER\_MULTIPLIER值为5,

} else {

curBaseLayer = curLayer = w.mBaseLayer;

}

assignAnimLayer(w, curLayer); //设置Z order值，以及动画的Z order

if (w.mLayer != oldLayer || w.mWinAnimator.mAnimLayer != oldLayer) {

layerChanged = true;

anyLayerChanged = true;

}

if (w.mAppToken != null) {

mHighestApplicationLayer = Math.max(mHighestApplicationLayer,

w.mWinAnimator.mAnimLayer); //获得应用窗口中最大值，注意这里只是针对应用窗口。

}

collectSpecialWindows(w); //收集特殊的窗口

if (layerChanged) {

w.scheduleAnimationIfDimming();

}

}

adjustSpecialWindows(); //针对特殊窗口再做调整, 不讨论

if (mService.mAccessibilityController != null && anyLayerChanged

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

mService.mAccessibilityController.onWindowLayersChangedLocked();

}

}

Z order值

**三、将 Z order更新到Native层**

Z order值是Surface里的概念(SurfaceFlinger中对应的是Layer类), 由于addWindow发生的很早，此时并没有Surface. 所以并不是在这里设备Z order值的，

准备的说是在创建Surface过后，

WindowSurfaceController createSurfaceLocked() {

...

//layer stack表示是该Surface显示在哪个显示设备上，

final int layerStack = w.getDisplayContent().getDisplay().getLayerStack();

//将 layerstack与 z -order值(mAnimlayer)设置到Surface对应的SurfaceFlinger中的Layer中

mSurfaceController.setPositionAndLayer(mTmpSize.left, mTmpSize.top, layerStack, mAnimLayer);

mLastHidden = true;

}

最后通过Layer.cpp中的setLayer函数将Z order值保存到Layer的mCurrentState中的z变量中。

bool Layer::setLayer(uint32\_t z) {

if (mCurrentState.z == z)

return false;

mCurrentState.sequence++;

mCurrentState.z = z;

mCurrentState.modified = true;

setTransactionFlags(eTransactionNeeded);

return true;

}

**小礼物走一走，来简书关注我**

赞赏支持