**Android窗口管理服务WindowManagerService对输入法窗口（Input Method Window）的管理分析**

2014年06月16日 19:32:58 [xu\_song](https://me.csdn.net/xu_song) 阅读数：924

  在Android系统中，输入法窗口是一种特殊类型的窗口，它总是位于需要使用输入法的窗口的上面。也就是说，一旦WindowManagerService服务检测到焦点窗口需要使用输入法，那么它就会调整输入法窗口在窗口堆栈中的位置，使得输入法窗口位于在焦点窗口的上面，这样用户可以通过输入法窗口来录入字母或者文字。本文就将详细分析WindowManagerService服务是如何管理系统中的输入法窗口的。

        在Android系统中，除了输入法窗口之外，还有一种窗口称为输入法对话框，它们总是位于输入窗口的上面。Activity窗口、输入法窗口和输入法对话框的位置关系如图1所示：

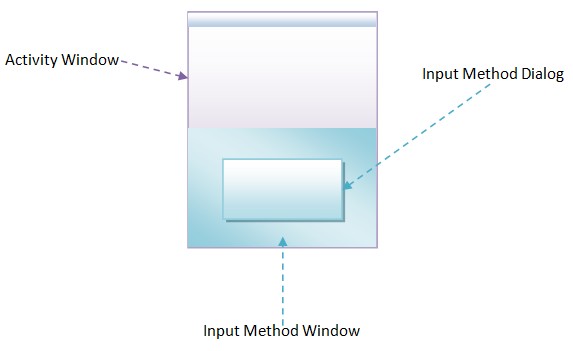


图1 Activity窗口、输入法窗口和输入法对话框的位置关系

        在前面[Android窗口管理服务WindowManagerService组织窗口的方式分析](http://blog.csdn.net/luoshengyang/article/details/8498908)一文中提到，WindowManagerService服务是使用堆栈来组织系统中的窗口的，因此，如果我们在窗口堆栈中观察Activity窗口、输入法窗口和输入法对话框，它们的位置关系就如图2所示：

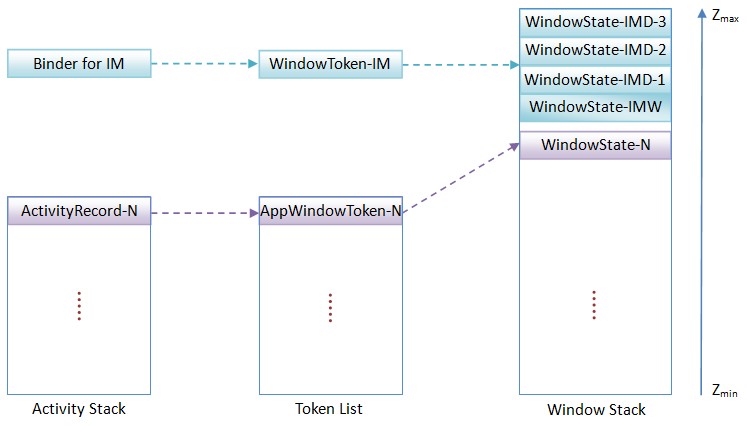


图2 Activity窗口、输入法窗口和输入法对话框在窗口堆栈中的位置关系

        图2中的对象的关系如下所示：

        1. 在ActivityManagerService服务内部的Activity组件堆栈顶端的ActivityRecord对象N描述的是系统当前激活的Activity组件。

        2. ActivityRecord对象N在WindowManagerService服务内部的窗口令牌列表顶端对应有一个AppWindowToken对象N。

        3. AppWindowToken对象N在WindowManagerService服务内部的窗口堆栈中对应有一个WindowState对象N，用来描述系统当前激活的Activity组件窗口。

        4. WindowState对象N上面有一个WindowState对象IMW，用来描述系统中的输入法窗口。

        5. WindowState对象IMW上面有三个WindowState对象IMD-1、IMD-2和IMD-3，它们用来描述系统中的输入法对话框。

        6. 系统中的输入法窗口以及输入法对话框在WindowManagerService服务内部中对应的窗口令牌是由WindowToken对象IM来描述的。

        7. WindowToken对象IM在InputMethodManagerService服务中对应有一个Binder对象。

        总的来说，就是图2描述了系统当前激活的Activity窗口上面显示输入法窗口，而输入法窗口上面又有一系列的输入法对话框的情景。WindowManagerService服务的职能之一就是要时刻关注系统中是否有窗口需要使用输入法。WindowManagerService服务一旦发现有窗口需要使用输入法，那么就会调整输入法窗口以及输入法对话框在窗口堆栈中的位置，使得它们放置在需要使用输入法的窗口的上面。

        接下来，我们就首先分析两个需要调整输入法窗口以及输入法对话框在窗口堆栈中的位置的情景，然后再分析它们是如何在窗口堆栈中进行调整的。

        第一个需要调整输入法窗口以及输入法对话框在窗口堆栈中的位置的情景是增加一个窗口到WindowManagerService服务去的时候。从前面[Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析](http://blog.csdn.net/luoshengyang/article/details/8275938)一文可以知道，增加一个窗口到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这个函数中与输入法窗口以及输入法对话框调整相关的逻辑，如下所示：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/8526644)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. WindowState mInputMethodWindow = **null**;
6. **final** ArrayList<WindowState> mInputMethodDialogs = **new** ArrayList<WindowState>();
7. ......
9. **public** **int** addWindow(Session session, IWindow client,
10. WindowManager.LayoutParams attrs, **int** viewVisibility,
11. Rect outContentInsets, InputChannel outInputChannel) {
12. ......
14. **synchronized**(mWindowMap) {
15. ......
17. WindowToken token = mTokenMap.get(attrs.token);
18. **if** (token == **null**) {
19. ......
20. **if** (attrs.type == TYPE\_INPUT\_METHOD) {
21. ......
22. **return** WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;
23. }
24. ......
25. }
26. ......
28. win = **new** WindowState(session, client, token,
29. attachedWindow, attrs, viewVisibility);
30. ......
32. **boolean** imMayMove = **true**;
34. **if** (attrs.type == TYPE\_INPUT\_METHOD) {
35. mInputMethodWindow = win;
36. addInputMethodWindowToListLocked(win);
37. imMayMove = **false**;
38. } **else** **if** (attrs.type == TYPE\_INPUT\_METHOD\_DIALOG) {
39. mInputMethodDialogs.add(win);
40. addWindowToListInOrderLocked(win, **true**);
41. adjustInputMethodDialogsLocked();
42. imMayMove = **false**;
43. }
44. ......
46. **boolean** focusChanged = **false**;
47. **if** (win.canReceiveKeys()) {
48. focusChanged = updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_ASSIGN\_LAYERS);
49. **if** (focusChanged) {
50. imMayMove = **false**;
51. }
52. }
54. **if** (imMayMove) {
55. moveInputMethodWindowsIfNeededLocked(**false**);
56. }
58. ......
59. }
61. ......
62. }
64. ......
65. }

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

        如果当前增加到WindowManagerService服务来的是一个输入法窗口，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，那么就要求与该输入法窗口所对应的类型为WindowToken的窗口令牌已经存在，否则的话，WindowManagerService类的成员函数addWindow就会直接返回一个错误码WindowManagerImpl.ADD\_BAD\_APP\_TOKEN给调用者。这个类型为WindowToken的窗口令牌是InputMethodManagerService服务请求WindowManagerService服务创建的，即调用WindowManagerService类的成员函数addWindowToken来创建的，具体可以参考前面[Android窗口管理服务WindowManagerService组织窗口的方式分析](http://blog.csdn.net/luoshengyang/article/details/8498908)一文。

        如果当前增加到WindowManagerService服务来的是一个输入法窗口，那么就会将前面为它所创建的一个WindowState对象win保存在WindowManagerService类的成员变量mInputMethodWindow中，接着还会调用WindowManagerService类的成员函数addInputMethodWindowToListLocked来将该WindowState对象插入到窗口堆栈的合适位置去。

        如果当前增加到WindowManagerService服务来的是一个输入法对话框，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，那么就会将前面为它所创建的一个WindowState对象win添加到WindowManagerService类的成员变量mInputMethodDialogs所描述的一个ArrayList中去，并且先后调用WindowManagerService类的成员函数addWindowToListInOrderLocked和adjustInputMethodDialogsLocked来将该WindowState对象插入到窗口堆栈的合适位置去。

        在上述两种情况中，由于用来描述输入法窗口或者输入法对话框的WindowState对象已经被插入到了窗口堆栈中的合适位置，因此，接下来就不再需要考虑移动该输入法窗口或者输入法对话框了，这时候变量imMayMove的值就会被设置为false。

       另一方面，如果当前增加到WindowManagerService服务来的既不是一个输入法窗口，也不是一个输入法对话框，并且该窗口需要接收键盘事件，即前面所创建的WindowState对象win的成员函数canReceiveKeys的返回值为true，那么就可能会导致系统当前获得焦点的窗口发生变化，这时候就需要调用WindowManagerService类的成员函数updateFocusedWindowLocked来重新计算系统当前获得焦点的窗口。如果系统当前获得焦点的窗口发生了变化，那么WindowManagerService类的成员函数updateFocusedWindowLocked的返回值focusChanged就会等于true，同时系统的输入法窗口和输入法对话框在窗口堆栈中的位置也会得到调整，即位它们会位于系统当前获得焦点的窗口的上面，因此，这时候变量imMayMove的值也会被设置为false，表示接下来不再需要考虑移动系统中的输入法窗口或者输入法对话框在窗口堆栈中的位置。

       最后，如果变量imMayMove的值保持为初始值，即保持为true，那么就说明当前增加的窗口可能会引发系统的输入法窗口和输入法对话框在窗口堆栈中的位置发生变化，因此，这时候就需要调用WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked来作检测，并且在发生变化的情况下，将系统的输入法窗口和输入法对话框移动到窗口堆栈的合适位置上去。

       从上面的分析就可以知道，在增加一个窗口的过程中，可能需要调用WindowManagerService类的成员函数addInputMethodWindowToListLocked、addWindowToListInOrderLocked、adjustInputMethodDialogsLocked和moveInputMethodWindowsIfNeededLocked来移动系统的输入法窗口和输入法对话框，其中，WindowManagerService类的成员函数addWindowToListInOrderLocked在前面[Android窗口管理服务WindowManagerService组织窗口的方式分析](http://blog.csdn.net/luoshengyang/article/details/8498908)一文已经分析过了，本文只要关注其余三个成员函数的实现。

       第二个需要调整输入法窗口以及输入法对话框在窗口堆栈中的位置的情景是一个应用程序进程请求WindowManagerService服务重新布局一个窗口的时候。从前面[Android窗口管理服务WindowManagerService计算Activity窗口大小的过程分析](http://blog.csdn.net/luoshengyang/article/details/8479101)一文可以知道，应用程序进程请求WindowManagerService服务重新布局一个窗口最终是通过调用WindowManagerService类的成员函数relayoutWindow来实现的。接下来我们就主要分析这个函数中与输入法窗口以及输入法对话框调整相关的逻辑，如下所示：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/8526644)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **public** **int** relayoutWindow(Session session, IWindow client,
6. WindowManager.LayoutParams attrs, **int** requestedWidth,
7. **int** requestedHeight, **int** viewVisibility, **boolean** insetsPending,
8. Rect outFrame, Rect outContentInsets, Rect outVisibleInsets,
9. Configuration outConfig, Surface outSurface) {
10. ......
12. **synchronized**(mWindowMap) {
13. WindowState win = windowForClientLocked(session, client, **false**);
14. ......
16. **int** attrChanges = 0;
17. **int** flagChanges = 0;
18. **if** (attrs != **null**) {
19. flagChanges = win.mAttrs.flags ^= attrs.flags;
20. attrChanges = win.mAttrs.copyFrom(attrs);
21. }
22. ......
24. **boolean** imMayMove = (flagChanges&(
25. WindowManager.LayoutParams.FLAG\_ALT\_FOCUSABLE\_IM |
26. WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE)) != 0;
28. **boolean** focusMayChange = win.mViewVisibility != viewVisibility
29. || ((flagChanges&WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE) != 0)
30. || (!win.mRelayoutCalled);
31. ......
33. **if** (viewVisibility == View.VISIBLE &&
34. (win.mAppToken == **null** || !win.mAppToken.clientHidden)) {
35. displayed = !win.isVisibleLw();
36. ......
38. **if** ((attrChanges&WindowManager.LayoutParams.FORMAT\_CHANGED) != 0) {
39. // To change the format, we need to re-build the surface.
40. win.destroySurfaceLocked();
41. displayed = **true**;
42. }
43. ......
45. **if** (win.mAttrs.type == TYPE\_INPUT\_METHOD
46. && mInputMethodWindow == **null**) {
47. mInputMethodWindow = win;
48. imMayMove = **true**;
49. }
51. **if** (displayed) {
52. focusMayChange = **true**;
53. }
55. ......
56. } **else** {
57. ......
59. **if** (win.mSurface != **null**) {
60. ......
61. // If we are not currently running the exit animation, we
62. // need to see about starting one.
63. **if** (!win.mExiting || win.mSurfacePendingDestroy) {
64. ......
66. **if** (!win.mSurfacePendingDestroy && win.isWinVisibleLw() &&
67. applyAnimationLocked(win, transit, **false**)) {
68. focusMayChange = **true**;
69. win.mExiting = **true**;
70. } **else** **if** (win.isAnimating()) {
71. // Currently in a hide animation... turn this into
72. // an exit.
73. win.mExiting = **true**;
74. } **else** **if** (win == mWallpaperTarget) {
75. // If the wallpaper is currently behind this
76. // window, we need to change both of them inside
77. // of a transaction to avoid artifacts.
78. win.mExiting = **true**;
79. win.mAnimating = **true**;
80. } **else** {
81. **if** (mInputMethodWindow == win) {
82. mInputMethodWindow = **null**;
83. }
84. win.destroySurfaceLocked();
85. }
86. }
87. }
89. ......
90. }
92. **if** (focusMayChange) {
93. ......
94. **if** (updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES)) {
95. imMayMove = **false**;
96. }
97. ......
98. }
100. // updateFocusedWindowLocked() already assigned layers so we only need to
101. // reassign them at this point if the IM window state gets shuffled
102. **boolean** assignLayers = **false**;
104. **if** (imMayMove) {
105. **if** (moveInputMethodWindowsIfNeededLocked(**false**) || displayed) {
106. // Little hack here -- we -should- be able to rely on the
107. // function to return true if the IME has moved and needs
108. // its layer recomputed.  However, if the IME was hidden
109. // and isn't actually moved in the list, its layer may be
110. // out of data so we make sure to recompute it.
111. assignLayers = **true**;
112. }
113. }
114. ......
116. **if** (assignLayers) {
117. assignLayersLocked();
118. }
120. ......
121. }
123. ......
125. **return** (inTouchMode ? WindowManagerImpl.RELAYOUT\_IN\_TOUCH\_MODE : 0)
126. | (displayed ? WindowManagerImpl.RELAYOUT\_FIRST\_TIME : 0);
127. }
129. ......
130. }

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

        应用程序进程在请求WindowManagerService服务重新布局一个窗口的时候，这个窗口的一些布局参数可能会发生变化，而这些变化可能会同时引发系统的输入法窗口以及输入法对话框在窗口堆栈中的位置发生变化。如果系统的输入法窗口以及输入法对话框在窗口堆栈中的位置发生了变化，那么就需要调整它们在窗口堆栈中的位置。

        WindowManagerService类的成员函数relayoutWindow首先调用根据参数session和client来调用另外一个成员函数windowForClientLocked，以便可以获得用来描述要重新布局的窗口的一个WindowState对象win。

        WindowState对象win的成员变量mAttrs指向的是一个WindowManager.LayoutParams对象，该WindowManager.LayoutParams对象的成员变量flags描述的是窗口上一次所设置的布局属性标志位，而参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量flags描述的是窗口当前被设置的布局属性标志位。WindowManagerService类的成员函数relayoutWindow通过对这两个标志位执行一个异或操作，就可以知道窗口的哪些布局属性标志位发生了变化，这些变化就记录在变量flagChanges中。

        WindowManagerService类的成员函数relayoutWindow在对WindowState对象win所描述的窗口进行布局之前，还要将参数attrs指的是一个WindowManager.LayoutParams对象的内容拷贝到 WindowState对象win的成员变量mAttrs指向的是一个WindowManager.LayoutParams对象中去。在拷贝的过程中，如果发现这两个WindowManager.LayoutParams对象所描述的窗口布局属性有发生变化，那么这些变化就会记录在变量attrChanges中。

        在窗口的布局属性标志中，位WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE表示窗口是否可以获得焦点，另外一个位WindowManager.LayoutParams.FLAG\_ALT\_FOCUSABLE\_IM是用来反转WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE位的作用的。一个窗口是否可以获得焦点意味着它是否需要与输入法窗口交互，即如果一个窗口是可以获得焦点的，那么就意味着它需要与输入法窗口交互，否则就不需要。当一个窗口的WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE位等于1，那么就表示窗口不可以获得焦点，即不需要与输入法窗口交互，但是如果该窗口的WindowManager.LayoutParams.FLAG\_ALT\_FOCUSABLE\_IM位也等于1，那么就表示窗口仍然是需要与输入法窗口交互的。另一方面，如果一个窗口的WindowManager.LayoutParams.FLAG\_ALT\_FOCUSABLE\_IM位等于1，但是该窗口的WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE位等于0，那么就表示窗口仍然是不可以与输入法窗口交互的。因此，当前面得到的变量flagChanges的WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE位或者WindowManager.LayoutParams.FLAG\_ALT\_FOCUSABLE\_IM位发生了变化时，都意味着对WindowState对象win所描述的窗口进行重新布局会影响系统中的输入法窗口以及输入法对话框，即该窗口可能会由需要显示输入法窗口以及输入法对话框，到不需要显示输入法窗口以及输入法对话框，反之亦然。最后得到的变量imMayMove的值等于true就表示要移动系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置。

        一个窗口由不可获得焦点到可以获得焦点，或者由可获得焦点到不可以获得焦点，即窗口布局属性标志中的WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE位发生了变化，那么就意味着要重新计算系统当前获得焦点的窗口。从前面分析增加窗口到WindowManagerService服务的情景可以知道，当系统当前获得焦点的窗口发生变化时，也意味着需要系统中的移动输入法窗口以及输入法对话框在窗口堆栈中的位置。除了窗口布局属性标志中的WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE位变化会引发系统当前获得焦点的窗口发生变化之外，还有另外两个因素会引发系统当前获得焦点的窗口发生变化。第一个因素是窗口的可见性发生变化。WindowState对象win的成员变量mViewVisibility描述的是窗口上一次布局时的可见性，而参数viewVisibility描述的是窗口当前的可见性，当它们的值不相等时，就意味着窗口的可见性发生了变化。第二个因素是窗口是第一次被应用程序进程请求WindowManagerService服务布局，这时候WindowState对象win的成员变量mRelayoutCalled的值就会等于false。最后得到的变量focusMayChange等于true，就表示需要重新计算系统当前获得焦点的窗口。

        WindowState对象win所描述的窗口在此次重新布局中是否会引起移动系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置，还取决于它在的可见性以及它的绘图表面属性等信息，接下来我们就按照 WindowState对象win所描述的窗口当前是可见还是不可见来分别分析。

        我们首先分析WindowState对象win所描述的窗口在此次重新布局中是可见的情景，即参数viewVisibility的值等于View.VISIBLE。注意，如果WindowState对象win所描述的是一个Activity窗口，而该Activity组件是不可见的，那么即使参数viewVisibility的值等于View.VISIBLE，那么WindowState对象win所描述的窗口在此次重新布局中也是认为不可见的。从前面[Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析](http://blog.csdn.net/luoshengyang/article/details/8275938)一文可以知道，当WindowState对象win的成员变量mAppToken的值不等于null时，那么该WindowState对象win描述的是一个Activity窗口，而当该成员变量所指向的一个AppWindowToken对象的成员变量clientHidden的值等于false时，就表示对应的Activity组件是可见的。

        WindowState对象win所描述的窗口在上一次布局时的可见性可以调用它的成员函数isVisibleLw来获得。如果WindowState对象win所描述的窗口在上一次布局时是不可见的，那么现在就需要将它设置为可见的，即要将它显示出来，这时候变量displayed的值就会等于true。另一方面，如果WindowState对象win所描述的窗口的绘图表面的像素格式发生了变化，即变量attrChanges的WindowManager.LayoutParams.FORMAT\_CHANGED位等于1，那么这时候就需要调用WindowState对象win的成员函数destroySurfaceLocked来销毁它所描述的窗口的绘图表面，以便接下来可以为它重新创建一个新的绘图表面，这时候也会将变量displayed的值设置为true，表示接下来是要显示WindowState对象win所描述的窗口的。如果最终得到的变量displayed的值设置为true，那么就相当于说明WindowState对象win所描述的窗口经历一个由不可见到可见的状态变化，因此就可能会导致系统当前获得焦点的窗口发生变化，这时候就会将变量focusMayChange的值设置为true。

        如果WindowState对象win描述的是一个输入法窗口，即它的成员变量mAttrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_INPUT\_METHOD，并且系统中的输入法窗口尚未设置，即WindowManagerService类的成员变量mInputMethodWindow的值等于null，那么就说明接下来要显示的其实是输入法窗口，这情况会导致需要移动系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置，因此，这时候除了需要将WindowState对象win保存在WindowManagerService类的成员变量mInputMethodWindow之外，还需要将变量imMayMove的值设置为true。

        我们接下来再分析WindowState对象win所描述的窗口在此次重新布局中是不可见的情景。一个窗口变得不可见了，就意味着可能要销毁它的绘图表面，取决于它的绘图表面是否存在，以及它的退出动画是否已经显示结束。WindowState对象win所描述的窗口的绘图表面保存在它的成员变量mSurface中，因此，当WindowState对象win的成员变量mSurface不等于null的时候，就意味着可能会销毁它所描述的绘图表面。

        如果WindowState对象win的成员变量mExiting等于false时，那么就说明该WindowState对象win所描述的窗口的退出动画可能尚未开始，也可能已经结束。另一方面，如果WindowState对象win的成员变量mSurfacePendingDestroy的值等于true，那么就说明该WindowState对象win所描述的窗口的绘图表面正在等待销毁。这两种情况都需要进一步确定接下来是要开始WindowState对象win所描述的窗口的退出动画，还是要销毁WindowState对象win所描述的窗口的绘图表面。

        如果WindowState对象win的成员变量mSurfacePendingDestroy的值等于false，那么同时也意味着它所描述的窗口还未开始显示退出动画，因而它的绘图表面就没有进入正在等待销毁的状态。在这种情况下，如果WindowState对象win所描述的窗口是可见的，即它的成员函数isWinVisibleLw的返回值等于true，那么就意味要开始该窗口的退出动画了，这是通过调用WindowManagerService类的成员函数applyAnimationLocked来实现的。WindowState对象win描述的窗口开始退出动画之后，就意味要重新计算系统当前获得焦点的窗口，因此，这时候就会将变量focusMayChange的值设置为true，同时还会将WindowState对象win的成员变量mExiting的值设置为true，表示它描述的窗口正在退出的过程中。

        如果WindowState对象win所描述的窗口正在处于退出动画的过程中，即它的成员函数isAnimating的返回值等于true，那么这时候需要确保WindowState对象win的成员变量mExiting的值为true。

        如果WindowState对象win所描述的窗口已经结束退出动画，但是它仍然是壁纸窗口的目标，即WindowManagerService类的成员变量mWallpaperTarget的值不等于null，并且它的值就等于WindowState对象win，那么这时候就需要等待壁纸窗口也退出之后，才销毁WindowState对象win所描述的窗口，因此，这时候就需要将WindowState对象win的成员变量mExiting和mAnimating的值设置为true，即假装它所描述的窗口还处于正在退出的过程，这样做是为了等待壁纸窗口退出完成。

        如果WindowState对象win所描述的窗口已经结束退出动画，并且它不是壁纸窗口的目标，那么这时候就需要调用它的成员函数destroySurfaceLocked来销毁它的绘图表面了。在销毁WindowState对象win所描述的窗口之前，还会判断它是否就是系统当前的输入法窗口，即WindowManagerService类的成员变量mInputMethodWindow的值是否等于win。如果等于的话，那么就说明系统当前的输入法窗口被销毁了，因此，就需要将WindowManagerService类的成员变量mInputMethodWindow的值设置为null。

        经过上面的一系列操作之后，如果最终得到的变量focusMayChange的值等于true，那么就说明需要重新计算系统当前获得焦点的窗口了，这是通过调用WindowManagerService类的成员函数updateFocusedWindowLocked来实现的。一旦WindowManagerService类的成员函数updateFocusedWindowLocked的返回值为true，那么就说明统当前获得焦点的窗口发生了变化，并且系统中的输入法窗口以及输入法对话框也移动到窗口堆栈中的正确位置了，因此，这时候就会将变量imMayMove的值设置为false。

        经过上面的一系列操作之后，如果最终得到的变量imMayMove的值等于true，那么就说明有可能需要移动系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置，这是通过调用WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked来实现的。一旦系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置发生了移动，那么WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked的返回值就等于true，这时候就需要将变量assignLayers的值设置为true，表示要重新计算系统中的窗口的Z轴位置，以便可以同步到SurfaceFlinger服务中去。注意，如果系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置没有发生变化，但是前面得到的变量displayed的值等于true，那么也是需要将变量assignLayers的值设置为true的，因为这个变量displayed的值等于true意味着WindowState对象win所描述的窗口经历了从不可见到可见的状态变化，因此也需要重新计算系统中的窗口的Z轴位置。

        经过上面的一系列操作之后，如果最终得到的变量assignLayers的值等于true，那么就需要调用WindowManagerService类的成员函数assignLayersLocked来执行重新计算统中的窗口的Z轴位置的操作了。在后面的文章中，我们再详细分析WindowManagerService服务是如何计算系统中的窗口的Z轴位置的。

        从上面的分析就可以知道，在布局一个窗口的过程中，可能需要调用WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked来移动系统的输入法窗口和输入法对话框。再结合前面增加窗口的情景，我们就可以知道，在WindowManagerService类中，与输入法窗口以及输入法对话框相关的成员函数有addInputMethodWindowToListLocked、adjustInputMethodDialogsLocked和moveInputMethodWindowsIfNeededLocked，它们的作用如下所示：

        A. 成员函数addInputMethodWindowToListLocked用来将输入法窗口插入到窗口堆栈的合适位置，即插入到需要显示输入法窗口的窗口的上面。

        B. 成员函数adjustInputMethodDialogsLocked用来移动输入法对话框到窗口堆栈的合适位置，即移动到输入法窗口的上面。

        C. 成员函数moveInputMethodWindowsIfNeededLocked用来检查是否需要移动输入法窗口以及输入法对话框。如果需要的话，那么就将它们移动到窗口堆栈的合适位置去，即将输入法窗口移动到需要显示输入法窗口的窗口的上面，而将输入法对话框移动到输入法窗口的上面。

        在分析这三个成员函数的实现之前，我们首先分析WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked和moveInputMethodDialogsLocked，它们是两个基本的操作，其中：

        D.  成员函数findDesiredInputMethodWindowIndexLocked用来查找输入法窗口在窗口堆栈的正确位置，这个位置刚好就是在需要显示输入法窗口的窗口在窗口堆栈中的上一个位置。

        E. 成员函数moveInputMethodDialogsLocked用来将移动输入法对话框移动到输入法窗口的上面去。

        接下来我们开始分析上述五个函数的实现。

        1. 计算输入法窗口在窗口堆栈中的位置

        输入法窗口在窗口堆栈中的位置是通过调用WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked来获得的，它首先找到需要显示输入法的窗口在窗口堆栈中的位置，然后再将这个位置加1，就可以得到输入法窗口在窗口堆栈中的位置。

        WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中，它的实现比较长，我们分段来阅读：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/8526644)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **int** findDesiredInputMethodWindowIndexLocked(**boolean** willMove) {
6. **final** ArrayList<WindowState> localmWindows = mWindows;
7. **final** **int** N = localmWindows.size();
8. WindowState w = **null**;
9. **int** i = N;
10. **while** (i > 0) {
11. i--;
12. w = localmWindows.get(i);
13. ......
15. **if** (canBeImeTarget(w)) {
16. ......
18. // Yet more tricksyness!  If this window is a "starting"
19. // window, we do actually want to be on top of it, but
20. // it is not -really- where input will go.  So if the caller
21. // is not actually looking to move the IME, look down below
22. // for a real window to target...
23. **if** (!willMove
24. && w.mAttrs.type == WindowManager.LayoutParams.TYPE\_APPLICATION\_STARTING
25. && i > 0) {
26. WindowState wb = localmWindows.get(i-1);
27. **while** (i > 1 && wb.mAppToken == w.mAppToken && !canBeImeTarget(wb)) {
28. i--;
29. wb = localmWindows.get(i-1);
30. }
31. **if** (wb.mAppToken == w.mAppToken && canBeImeTarget(wb)) {
32. i--;
33. w = wb;
34. }
35. }
36. **break**;
37. }
38. }
40. mUpcomingInputMethodTarget = w;

        这段代码从上到下遍历WindowManagerService服务内部的窗口堆栈，即WindowManagerService类的成员变量mWindows所描述的一个ArrayList。如果发现有一个窗口是可见的，并且需要显示输入法窗口，那么整个查找过程就会结束。检查一个窗口是否可见以及需要显示输入法窗口是通过调用WindowManagerService类的成员函数canBeImeTarget来实现的。最后得到的需要显示输入法的窗口就使用WindowState对象w中，这个WindowState对象w接下来还会保存在WindowManagerService类的成员变量mUpcomingInputMethodTarget中，表示它即将要成为输入法窗口的目标窗口。

        参数willMove表示调用者计算输入法窗口在窗口堆栈中的位置的目的。如果它的值等于true，那么就说明调用者获得了输入法窗口在窗口堆栈中的位置之后，接下来就会将输入法窗口移动到需要显示输入法窗口的窗口的上面去，否则的话，就说明调用者只是为了知道输入法窗口在窗口堆栈中的位置，而不打算移动输入法窗口。

        在从上到下查找需要显示输入法的窗口的过程中，如果找到一个WindowState对象w，它所描述的窗口需要显示输入法窗口，但是这个窗口其实是一个Activity窗口的启动窗口，即该WindowState对象w的成员变量mAttrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于WindowManager.LayoutParams.TYPE\_APPLICATION\_STARTING，那么由于调用WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked的目的不是用来移动输入法窗口，而是用来查找输入法窗口在窗口堆栈中的确切位置，因此就不能前面所找到的启动窗口看作是一个需要输入法的窗口，因为这个启动窗口只是Activity窗口在显示过程中出现的一个临时窗口。在这种情况下，这段代码就会继续沿着窗口堆栈往下查找另外一个窗口，该窗口一方面是需要显示输入法窗口的，另一方面要与前面所找到的启动窗口对应的是同一个窗口令牌的。如果能找到这样的一个窗口，那么就会将用来描述它的一个WindowState对象wb保存在变量w中。如果找不到这样的一个窗口，那么这段代码就会继续沿着窗口堆栈往下查找另外一个需要显示输入法的窗口。

        我们继续往下阅读代码：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/8526644)

1. **if** (willMove && w != **null**) {
2. **final** WindowState curTarget = mInputMethodTarget;
3. **if** (curTarget != **null** && curTarget.mAppToken != **null**) {
5. // Now some fun for dealing with window animations that
6. // modify the Z order.  We need to look at all windows below
7. // the current target that are in this app, finding the highest
8. // visible one in layering.
9. AppWindowToken token = curTarget.mAppToken;
10. WindowState highestTarget = **null**;
11. **int** highestPos = 0;
12. **if** (token.animating || token.animation != **null**) {
13. **int** pos = 0;
14. pos = localmWindows.indexOf(curTarget);
15. **while** (pos >= 0) {
16. WindowState win = localmWindows.get(pos);
17. **if** (win.mAppToken != token) {
18. **break**;
19. }
20. **if** (!win.mRemoved) {
21. **if** (highestTarget == **null** || win.mAnimLayer >
22. highestTarget.mAnimLayer) {
23. highestTarget = win;
24. highestPos = pos;
25. }
26. }
27. pos--;
28. }
29. }
31. **if** (highestTarget != **null**) {
32. ......
34. **if** (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {
35. // If we are currently setting up for an animation,
36. // hold everything until we can find out what will happen.
37. mInputMethodTargetWaitingAnim = **true**;
38. mInputMethodTarget = highestTarget;
39. **return** highestPos + 1;
40. } **else** **if** (highestTarget.isAnimating() &&
41. highestTarget.mAnimLayer > w.mAnimLayer) {
42. // If the window we are currently targeting is involved
43. // with an animation, and it is on top of the next target
44. // we will be over, then hold off on moving until
45. // that is done.
46. mInputMethodTarget = highestTarget;
47. **return** highestPos + 1;
48. }
49. }
50. }
51. }

        这段代码用来处理一种特殊情况，即参数willMove的值等于true，并且前面找到了一个需要显示输入法的窗口w，但是当前输入法窗口已经存在一个目标窗口，并且该目标窗口正在切换的过程中。在这种情况下，调用WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked的函数就需要等到当前输入法窗口的目标窗口的切换过程结束之后，再将输入法窗口移动到窗口w的上面去，换句话说，就是要保持输入法窗口在它当前的目标窗口的上面，直到它当前的目标窗口的切换过程结束为止。这样WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked就需要找到当前输入法窗口的目标窗口在窗口堆栈中的位置，然后再将该位置加1后返回给调用者。

        当WindowManagerService类的成员变量mInputMethodTarget的值不等于null，并且它描述的是一个Activity窗口时，即它的成员变量mAppToken的值不等于null时，那么就说明当前输入法窗口已经存在一个目标窗口，而这个目标窗口就是使用WindowManagerService类的成员变量mInputMethodTarget所指向的一个WindowState对象来描述的。接下来这段代码就检查该目标窗口是否正在切换的过程中，即是否正在显示切换动画。如果是的话，那么WindowState对象curTarget的成员变量animating的值就会等于true，或者另外一个成员变量animation的值不等于null，这时候就需要在与该目标窗口所对应的窗口令牌token所描述的一组窗口中，找到一个Z轴位置最大的并且不是已经被移除的窗口。WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked的调用者最后就是需要将输入法窗口移动到这个Z轴位置最大的并且不是已经被移除的窗口的上面的。

        一个窗口的Z轴位置是记录在用描述它的一个WindowState对象的成员变量mAnimLayer中的，而它是否是已经被移除是记录在这个WindowState对象的成员变量mRemoved中的，因此，如果在窗口令牌token所描述的一组WindowSate对象中，能找到一个WindowSate对象，它的成员变量mAnimLayer的值最大，并且它的成员变量mRemoved不等于true，那么这段代码就会将它保存在变量highestTarget中，并且将它描述的窗口在窗口堆栈中的位置保存在变量highestPos中。

        经过前面的一系列计算之后，如果变量highestTarget的值不等于null，那么就说明我们碰到前面所说的特殊的情况，这时候又要分为两种情况来讨论。

        第一种情况是当前输入法窗口的目标窗口即将要进入到切换过程，但是这个切换过程尚开始，即WindowManagerService类的成员变量mNextAppTransition的值不等于WindowManagerPolicy.TRANSIT\_UNSET。这时候就需要将WindowManagerService类的成员变量mInputMethodTargetWaitingAnim的值设置为true，表示当前输入法窗口的目标窗口正在等待进入切换动画中，并且需要将WindowManagerService类的成员变量mInputMethodTarget修正为变量highestTarget所描述的一个WindowState对象，因为这个WindowState对象才是真正用来描述当前输入法窗口的目标窗口的。

        第二种情况是当前输入法窗口的目标窗口已经处于切换的过程了，即变量highestTarget所描述的一个WindowState对象的成员函数isAnimating的返回值为true，并且该目标窗口的Z轴位置大于前面所找到的需要显示输入法窗口的窗口的Z轴，即变量highestTarget所描述的一个WindowState对象的成员变量mAnimLayer的值大于变量w所描述的一个WindowState对象的成员变量mAnimLayer的值。这时候就需要将WindowState对象highestTarget所描述的窗口维持为当前输入法窗口的目标窗口，即将WindowManagerService类的成员变量mInputMethodTarget设置为变量highestTarget，直到WindowState对象highestTarget所描述的窗口的切换过程结束为止。

       上述两种情况最后都需要将WindowState对象highestTarget所描述的窗口在窗口堆栈中的位置highestPos加1，然后再返回给调用者，以便调用者接下来可以输入法窗口移动在窗口堆栈的第（highestPos+1）个位置上。

        如果我们没有碰到前面所说的特殊的情况，那么WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked就会继续往下执行：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/8526644)

1. **if** (w != **null**) {
2. **if** (willMove) {
3. ......
4. mInputMethodTarget = w;
5. **if** (w.mAppToken != **null**) {
6. setInputMethodAnimLayerAdjustment(w.mAppToken.animLayerAdjustment);
7. } **else** {
8. setInputMethodAnimLayerAdjustment(0);
9. }
10. }
11. **return** i+1;
12. }

        如果变量w的值不等于null，那么就说明WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked在前面找到了一个需要显示输入法窗口的窗口。这个窗口是使用WindowState对象w来描述的，并且它在窗品堆栈中的位置记录在变量i中。这时候WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked就会执行以下三个操作：

        A. 将WindowState对象w保存在WindowManagerService类的成员变量mInputMethodTarget中，以便WindowManagerService服务可以知道当前输入法窗口的目标窗口是什么。

        B. 检查WindowState对象w描述的窗口是否是Activity窗口，即检查WindowState对象w的成员变量mAppToken的值是否不等于null。如果WindowState对象w描述的窗口是Activity窗口的话，那么就需要根据WindowState对象w的成员变量mAppToken所描述的一个AppWindowToken对象的成员变量animLayerAdjustment来调整系统中的输入法窗口以及输入法对话框的Z轴位置，即在系统中的输入法窗口以及输入法对话框的现有Z轴位置的基础上再增加一个调整量，这个调整量就保存在WindowState对象w的成员变量mAppToken所描述的一个AppWindowToken对象的成员变量animLayerAdjustment中。这个调整的过程是通过调用WindowManagerService类的成员函数setInputMethodAnimLayerAdjustment来实现的。如果WindowState对象w描述的窗口不是Activity窗口，那么就不需要调整系统中的输入法窗口以及输入法对话框的Z轴位置，但是仍然需要调用WindowManagerService类的成员函数setInputMethodAnimLayerAdjustment来将系统中的输入法窗口以及输入法对话框的Z轴位置调整量设置为0，即将WindowManagerService类的成员变量mInputMethodAnimLayerAdjustment的值设置为0。

       C. 将变量i的值加1之后返回给调用者，以便调用者可以将系统中的输入法窗口移动到窗口堆栈中的第（i+1）个位置上。

        如果变量w的值等于null，那么就说明WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked在前面没有找到一个需要显示输入法窗口的窗口，我们继续往下阅读它的代码，以便可以了解它是如何处理这种情况的：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/8526644)

1. **if** (willMove) {
2. ......
3. mInputMethodTarget = **null**;
4. setInputMethodAnimLayerAdjustment(0);
5. }
6. **return** -1;
7. }
9. ......
10. }

       WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked对在前面没有找到一个需要显示输入法窗口的窗口的情况的处理很简单。它判断参数willMove的值是否等于true。如果等于true的话，那么就会将WindowManagerService类的成员变量mInputMethodTarget的值设置为null，并且调用WindowManagerService类的成员函数setInputMethodAnimLayerAdjustment来将系统中的输入法窗口以及输入法对话框的Z轴位置调整量设置为0。这实际上是用来通知WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked的调用者，系统当前没有需要显示输入法窗口的窗口。

        最后，WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked返回一个-1值给调用者，也是表明系统当前没有需要显示输入法窗口的窗口。

        2. 移动输入法对话框移动到输入法窗口的上面

        系统中的输入法对话框是需要位于输入法窗口的上面的，因此，我们就需要有一个函数来将输入法对话框移动到输入法窗口的上面去。这个函数就是WindowManagerService类的成员函数moveInputMethodDialogsLocked，它的实现如下所示：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/8526644)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **void** moveInputMethodDialogsLocked(**int** pos) {
6. ArrayList<WindowState> dialogs = mInputMethodDialogs;
8. **final** **int** N = dialogs.size();
9. ......
10. **for** (**int** i=0; i<N; i++) {
11. pos = tmpRemoveWindowLocked(pos, dialogs.get(i));
12. }
13. ......
15. **if** (pos >= 0) {
16. **final** AppWindowToken targetAppToken = mInputMethodTarget.mAppToken;
17. **if** (pos < mWindows.size()) {
18. WindowState wp = mWindows.get(pos);
19. **if** (wp == mInputMethodWindow) {
20. pos++;
21. }
22. }
23. ......
24. **for** (**int** i=0; i<N; i++) {
25. WindowState win = dialogs.get(i);
26. win.mTargetAppToken = targetAppToken;
27. pos = reAddWindowLocked(pos, win);
28. }
29. ......
30. **return**;
31. }
32. **for** (**int** i=0; i<N; i++) {
33. WindowState win = dialogs.get(i);
34. win.mTargetAppToken = **null**;
35. reAddWindowToListInOrderLocked(win);
36. ......
37. }
38. }
40. ......
41. }

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

        在调用WindowManagerService类的成员函数moveInputMethodDialogsLocked之前，必须要保证系统中的输入法窗口已经被移动到窗口堆栈的正确位置，即已经被移动到需要显示输入法窗口的窗口的上面。这时候参数pos描述的或者是输入法窗口在窗口堆栈中的位置，或者是输入法窗口在窗口堆栈的位置的上一个位置，即输入法对话框在窗口堆栈中的起始位置。参数pos的值还可以小于0，这时候就表示系统当前没有需要显示输入法窗口的窗口。

        在移动输入法对话框到输入法窗口的上面之前，首先要将输入法对话框从窗口堆栈中移除，以便接下来可以重新将它们插入到窗口堆栈中。系统中的输入法对话框都保存在WindowManagerService类的成员变量mInputMethodDialogs所描述的一个ArrayList中，通过调用WindowManagerService类的成员函数来tmpRemoveWindowLocked来移除保存在这个ArrayList中的每一个WindowState对象，就可以将系统中的输入法对话框从窗口堆栈中移除中。注意，将一个WindowState对象从窗口堆栈中移除之后，可能会影响参数pos的值。例如，如果参数pos的值大于被移除的WindowState对象原来在窗口堆栈中的位置值，那么在该WindowState对象被移除之后，参数pos的值就要相应地减少1，这样它才能正确地反映输入法窗口在窗口堆栈中的位置，或者输入法对话框在窗口堆栈中的起始位置。WindowManagerService类的成员函数来tmpRemoveWindowLocked在将一个WindowState对象从窗口堆栈中移除的过程中，会正确处理好参数pos的值，这一点可以参考前面[Android窗口管理服务WindowManagerService组织窗口的方式分析](http://blog.csdn.net/luoshengyang/article/details/8498908)一文。

        接下来，我们就分为两种情况来分析输入法对话框在窗口是如何移动到输入法窗口的上面去的。

        第一种情况是参数pos的值大于等于0，这表明系统当前存在一个需要显示输入法窗口的窗口，这个窗口是通过WindowManagerService类的成员变量mInputMethodTarget所指向的一个WindowState对象来描述的。

        前面提到，参数pos描述的或者是输入法窗口在窗口堆栈中的位置，或者是输入法对话框在窗口堆栈中的起始位置，我们首先要将它统一描述为输入法对话框在窗口堆栈中的起始位置。这时候就需要检查保存在窗口堆栈的第pos个位置的WindowState对象wp，是否就是WindowManagerService类的成员变量mInputMethodWindow所指向的那个WindowState对象。如果是的话，那么就说明参数pos描述的或者是输入法窗口在窗口堆栈中的位置，这时候将它的值增加1，就可以让它表示为输入法对话框在窗口堆栈中的起始位置。

        得到了输入法对话框在窗口堆栈中的起始位置pos之后，接下来只需要调用WindowManagerService类的成员函数reAddWindowLocked来依次地将保存在WindowManagerService类的成员变量mInputMethodDialogs所描述的一个ArrayList中的第i个WindowState对象保存在窗口堆栈中的第（pos+i）个以位置上即可，这样就可以将输入法对话框都移动到输入法窗口的上面去了。

       注意，在移动的过程中，用来描述每一个输入法对话框的每一个WindowState对象的成员变量mTargetAppToken的值设置为WindowManagerService类的成员变量mInputMethodTarget所描述的一个WindowState对象的成员变量mAppToken的值，以便可以将输入法对话框和输入法窗口的目标窗口设置为同一个窗口。

        第二种情况是参数pos的值小于0，这表明系统当前不存在一个需要显示输入法窗口的窗口。这时候就需要根据输入法窗口自身的属性来将它们移动到窗口堆栈的合适的位置上去，这是通过调用WindowManagerService类的成员函数reAddWindowToListInOrderLocked来实现的。WindowManagerService类的成员函数reAddWindowToListInOrderLocked的实现可以参考前面[Android窗口管理服务WindowManagerService组织窗口的方式分析](http://blog.csdn.net/luoshengyang/article/details/8498908)一文，这里不再详细。

        注意，在移动的过程中，用来描述每一个输入法对话框的每一个WindowState对象的成员变量mTargetAppToken的值会被设置为null，这是因为系统当前不存在一个需要显示输入法窗口的窗口，即这时候每一个输入法对话框都没有目标窗口。

        理解了WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked和moveInputMethodDialogsLocked的实现之后，对WindowManagerService类的另外三个成员函数addInputMethodWindowToListLocked、adjustInputMethodDialogsLocked和moveInputMethodWindowsIfNeededLocked的实现就很有帮助，接下来我们就继续分析这三个成员函数的实现。

        3. 插入输入法窗口到需要显示输入法窗口的窗口上面

        插入输入法窗口到窗口堆栈的合适位置，使得它位于需要显示输入法窗口的窗口上面，这是通过调用WindowManagerService类的成员函数addInputMethodWindowToListLocked来实现的，它的实现如下所示：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/8526644)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **void** addInputMethodWindowToListLocked(WindowState win) {
6. **int** pos = findDesiredInputMethodWindowIndexLocked(**true**);
7. **if** (pos >= 0) {
8. win.mTargetAppToken = mInputMethodTarget.mAppToken;
9. ......
10. mWindows.add(pos, win);
11. mWindowsChanged = **true**;
12. moveInputMethodDialogsLocked(pos+1);
13. **return**;
14. }
15. win.mTargetAppToken = **null**;
16. addWindowToListInOrderLocked(win, **true**);
17. moveInputMethodDialogsLocked(pos);
18. }
20. ......
21. }

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

        参数win描述的是要添加到窗口堆栈中去的输入法窗口。

        WindowManagerService类的成员函数addInputMethodWindowToListLocked首先调用另外一个成员函数findDesiredInputMethodWindowIndexLocked来计算输入法窗口在窗口堆栈中的位置，并且保存在变量pos。

        如果变量pos的值大于等于0，那么就说明WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked在窗口堆栈中找到了一个合适的位置来放置输入法窗口，于是接下来就会参数win所描述的输入法窗口插入在WindowManagerService类的成员变量mWIndows所描述的窗口堆栈的第pos个位置上。由于系统中的输入法对话框要保持在输入法窗口的上面，因此，WindowManagerService类的成员函数addInputMethodWindowToListLocked还需要继续调用另外一个成员函数moveInputMethodDialogsLocked来将系统中的输入法对话框在窗口堆栈中的起始位置设置为（pos+1）。

        还有一个地方需要注意的是，前面在调用WindowManagerService类的成员函数addInputMethodWindowToListLocked来计算输入法窗口在窗口堆栈中的位置的时候，已经将用来描述需要显示输入法窗口的Activity窗口的一个WindowState对象保存了WindowManagerService类的成员变量mInputMethodTarget中，因此，这里就需要这个WindowState对象的成员变量mAppToken所指向的一个AppWindowToken对象保存在用来描述输入法窗口的WindowState对象的win的成员变量mTargetAppToken中，以便WindowManagerService服务可以知道当前输入法窗口的目标窗口是什么。

        如果变量pos的值小于0，那么就说明WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked没有找一个需要输入法窗口的窗口，因此，这时候就需要调用另外一个成员函数addWindowToListInOrderLocked来将参数win所描述的输入法窗口插入到窗口堆栈中去。WindowManagerService类的成员函数addWindowToListInOrderLocked会根据要目标窗口所对应的窗口令牌在窗口令牌列表中的位置以及是否在窗口堆栈中存在其它窗口等信息来在窗口堆栈中找到一个合适的前位置来放置目标窗口，它的具体实现可以参考前面[Android窗口管理服务WindowManagerService组织窗口的方式分析](http://blog.csdn.net/luoshengyang/article/details/8498908)一文。将参数win所描述的输入法窗口插入到窗口堆栈中去之后，WindowManagerService类的成员函数addInputMethodWindowToListLocked还需要继续调用另外一个成员函数moveInputMethodDialogsLocked来调整系统中的输入法对话框。

        注意，在调用WindowManagerService类的成员函数moveInputMethodDialogsLocked的时候，传递进去的参数pos的值等于-1，这时候WindowManagerService类的成员函数moveInputMethodDialogsLocked就不是直接调整输入法对话框在窗口堆栈中的位置的，而是调用另外一个成员函数reAddWindowToListInOrderLocked来调整的。

        还有另外一个地方需要注意的是，由于前面在调用WindowManagerService类的成员函数findDesiredInputMethodWindowIndexLocked的时候，没有找到一个需要输入法窗口的窗口，因此，这里就需要将参数win所描述的一个WindowState对象的成员变量mTargetAppToken的值设置为null，以便WindowManagerService服务可以知道当前输入法窗口的没有目标窗口。

        4. 调整输入法对话框在窗口堆栈的位置

        一旦系统中存在需要显示输入法窗口的窗口，那么就需要系统中的输入法对话框在窗口堆栈中的位置，使得它们放置在输入法窗品的上面，这是通过调用WindowManagerService类的成员函数adjustInputMethodDialogsLocked来实现的，如下所示：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/8526644)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **void** adjustInputMethodDialogsLocked() {
6. moveInputMethodDialogsLocked(findDesiredInputMethodWindowIndexLocked(**true**));
7. }
9. ......
10. }

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

        WindowManagerService类的成员函数adjustInputMethodDialogsLocked的实现很简单，它首先调用成员函数findDesiredInputMethodWindowIndexLocked来找到输入法窗口在窗口堆栈中的位置，然后再调用成员函数moveInputMethodDialogsLocked来将输入法对话框保存在这个位置之上。

        5. 调整输入法窗口在窗口堆栈的位置

        当系统中的窗口布局发生了变化之后，例如，当前获得焦点的窗口发生了变化，或者新增了一个窗口，那么都可能需要调整输入法窗口在窗口堆栈中的位置，以便它可以痊于需要显示输入法窗口的窗口的上面，这是通过调用WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked来实现的，如下所示：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/8526644)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **boolean** moveInputMethodWindowsIfNeededLocked(**boolean** needAssignLayers) {
6. **final** WindowState imWin = mInputMethodWindow;
7. **final** **int** DN = mInputMethodDialogs.size();
8. **if** (imWin == **null** && DN == 0) {
9. **return** **false**;
10. }
12. **int** imPos = findDesiredInputMethodWindowIndexLocked(**true**);
13. **if** (imPos >= 0) {
14. // In this case, the input method windows are to be placed
15. // immediately above the window they are targeting.
17. // First check to see if the input method windows are already
18. // located here, and contiguous.
19. **final** **int** N = mWindows.size();
20. WindowState firstImWin = imPos < N
21. ? mWindows.get(imPos) : **null**;
23. // Figure out the actual input method window that should be
24. // at the bottom of their stack.
25. WindowState baseImWin = imWin != **null**
26. ? imWin : mInputMethodDialogs.get(0);
27. **if** (baseImWin.mChildWindows.size() > 0) {
28. WindowState cw = baseImWin.mChildWindows.get(0);
29. **if** (cw.mSubLayer < 0) baseImWin = cw;
30. }
32. **if** (firstImWin == baseImWin) {
33. // The windows haven't moved...  but are they still contiguous?
34. // First find the top IM window.
35. **int** pos = imPos+1;
36. **while** (pos < N) {
37. **if** (!(mWindows.get(pos)).mIsImWindow) {
38. **break**;
39. }
40. pos++;
41. }
42. pos++;
43. // Now there should be no more input method windows above.
44. **while** (pos < N) {
45. **if** ((mWindows.get(pos)).mIsImWindow) {
46. **break**;
47. }
48. pos++;
49. }
50. **if** (pos >= N) {
51. // All is good!
52. **return** **false**;
53. }
54. }
56. **if** (imWin != **null**) {
57. ......
58. imPos = tmpRemoveWindowLocked(imPos, imWin);
59. ......
60. imWin.mTargetAppToken = mInputMethodTarget.mAppToken;
61. reAddWindowLocked(imPos, imWin);
62. ......
63. **if** (DN > 0) moveInputMethodDialogsLocked(imPos+1);
64. } **else** {
65. moveInputMethodDialogsLocked(imPos);
66. }
68. } **else** {
69. // In this case, the input method windows go in a fixed layer,
70. // because they aren't currently associated with a focus window.
72. **if** (imWin != **null**) {
73. ......
74. tmpRemoveWindowLocked(0, imWin);
75. imWin.mTargetAppToken = **null**;
76. reAddWindowToListInOrderLocked(imWin);
77. ......
78. **if** (DN > 0) moveInputMethodDialogsLocked(-1);;
79. } **else** {
80. moveInputMethodDialogsLocked(-1);;
81. }
83. }
85. **if** (needAssignLayers) {
86. assignLayersLocked();
87. }
89. **return** **true**;
90. }
92. ......
93. }

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

        WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked首先检查系统中是否存在输入法窗口和输入法对话框，即检查WindowManagerService类的成员变量mInputMethodWindow的值是否等于null，并且WindowManagerService类的成员变量mInputMethodDialogs所描述的一个ArrayList的大小是否等于0。如果输入法窗口和输入法对话框都不存在的话，那么就不用调整它们在窗口堆栈中的位置了，否则的话，WindowManagerService类的成员变量mInputMethodWindow所指向的一个WindowState对象就会保存在变量imWin中，以便接下来可以通过它来描述系统中的输入法窗口。

       在输入法窗口或者输入法对话框存在的情况下，WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked接下来就会继续调用另外一个成员函数findDesiredInputMethodWindowIndexLocked来找到输入法窗口在窗口堆栈中的位置，并且保存在变量imPos中。注意，变量imPos的值可能大于等于0，也可能等于-1。当变量imPos的值大于等于0的时候，就说明系统当前存在一个窗口需要显示输入法窗口，而当变量imPos的值等于-1的时候，就说明系统当前不存在一个窗口需要显示输入法窗口，或者系统中不存在输入法窗口。接下来我们分两种情况来分析WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked的实现。

        第一种情况是变量imPos的值可能大于等于0。这时候可能需要调整输入法窗口在窗口堆栈中的位置，也可能不需要调整输入法窗口在窗口堆栈中的位置，取决于输入法窗口的位置是否已经在窗口堆栈的第imPos个位置上，以及是否所有与输入法相关的窗口都连续在放置在窗口堆栈中。

        变量firstImWin描述的是当前位于窗口堆栈中Z轴位置最小的与输入法相关的窗口，它是通过变量imPos来获得的。另外一个变量baseImWin描述的是Z轴位置最小的与输入法相关的窗口。如果这两个变量描述的是同一个窗口，那么就说明输入法窗口的位置已经在窗口堆栈的第imPos个位置上，因此，就有可能不需要调整输入法窗品在窗口堆栈中的位置了。接下来我们就描述如何找到这个Z轴位置最小的与输入法相关的窗口。

        如果变量imWin的值不等于null，即WindowManagerService类的成员变量mInputMethodWindow的值不等于null，那么它所描述的窗口就是Z轴位置最小的与输入法相关的窗口，否则的话，Z轴位置最小的与输入法相关的窗口就是位于WindowManagerService类的成员变量mInputMethodDialogs所描述的一个ArrayList的第0个位置上的输入法对话框。这一步得到的Z轴位置最小的与输入法相关的窗口就保存在变量baseImWin中。

        如果变量baseImWin所描述的窗口有子窗口，即它所指向的一个WindowState对象的成员变量mChildWindows所描述的一个ArrayList的大小大于0。这时候如果用来描述第一个子窗口的WindowState对象的成员变量mSubLayer的值小于0，那么就说明变量baseImWin所描述的窗口在所有与输入法相关的窗口中的Z轴位置还不是最小的，因为在它的下面还存在着Z轴位置更小的子窗口。在这种情况下，变量baseImWin就会指向这个Z轴位置最小的子窗口。

        经过上面的一系列计算之后，如果变量firstImWin和变量baseImWin描述的是同一个窗口，那么还需要继续判断所有与输入法相关的窗口都连续在放置在窗口堆栈中。判断的方法如下所示：

       (1). 从窗口堆栈的第（imPos + 1）个位置开始往上查找一个非输入法相关的窗口。

       (2). 如果第(1)步能在窗口堆栈中大于等于（imPos+1）的位置pos上找到一个非输入法窗口，那么再继续从第pos个位置开始往上查找一个与输入法相关的窗口。

       (3). 如果第(2)步能在窗口堆栈中找到一个与输入法相关的窗口，那么就说明所有与输入法相关的窗口不是连续在放置在窗口堆栈中的，因为在它们中间有一个非输入法相关的窗口，否则的话，就说明所有与输入法相关的窗口都是连续在放置在窗口堆栈中的。

        在所有与输入法相关的窗口都是连续在放置在窗口堆栈中的情况下，WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked就会直接返回一个false值给调用者，表明不需要调整系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置。

        在所有与输入法相关的窗口不是连续在放置在窗口堆栈中的情况下，就需要重新调整系统中的输入法窗口以及输入法对话框在窗口堆栈中的位置。这里又需要分两个情景来讨论。

        第一个情景是变量imWin的值不等于null，这时候说明系统中存在一个输入法窗口，因此，就需要调整这个输入法窗口在窗口堆栈中的位置。调整的方法很简单：

        (1). 调用WindowManagerService类的成员函数tmpRemoveWindowLocked来从窗口堆栈中移除变量imWin所描述的输入法窗口。在移除的过程中，会同时计算输入法窗口在窗口堆栈中的新位置，这个位置还是保存在变量imPos中。

        (2). 调用WindowManagerService类的成员函数reAddWindowLocked重新将变量imWin所描述的输入法窗口插入到窗口堆栈的第imPos个位置中。在插入之前，还会将变量imWin所描述的一个WindowState对象的成员变量mTargetAppToken与WindowManagerService类的成员变量mInputMethodTarget所描述的一个WindowState对象的成员变量mAppToken指向同一个AppWindowToken对象，这样WindowManagerService服务就可以知道imWin所描述的输入法窗口的目标窗口是什么。

        (3). 如果系统中还存在输入法对话框，那么就调用WindowManagerService类的成员函数moveInputMethodDialogsLocked来将它们放置在第（imPos+1）个位置上，目的是将它们放置在输入法窗口的上面。

        第二个情景是变量imWin的值等于null，这时候说明系统中不存在输入法窗口。在这个情景下，系统中肯定会存在输入法对话框，否则的话，WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked在前面就会返回了。因此，WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked接下来就会直接调用成员函数moveInputMethodDialogsLocked来将系统中的输入法对话框放置在在第imPos个位置上。

        第二种情况是变量imPos的值等于-1。这时候说明系统中不存在需要显示输入法窗口的窗口。这里同样也需要分两个情景来分析。

        第一个情景是变量imWin的值不等于null，这时候说明系统中存在一个输入法窗口，因此，就需要调整这个输入法窗口在窗口堆栈中的位置。调整的方法与前面第一种情况的第一个情景是类似的。不过由于事先不知道输入法窗口在窗口堆栈中的位置，因此，这里就会调用WindowManagerService类的成员函数reAddWindowToListInOrderLocked和moveInputMethodDialogsLocked来间接地调整输入法窗口和输入法对话框在窗口堆栈中的位置。注意，在调用WindowManagerService类的成员函数moveInputMethodDialogsLocked的时候，传进去的参数为-1。另外一个地方需要注意的是，在WindowManagerService类的成员函数reAddWindowToListInOrderLocked来间接地调整输入法窗口在窗口堆栈中的位置之前，会将量imWin所描述的一个WindowState对象的成员变量mTargetAppToken的值设置为null，这样WindowManagerService服务就可以知道imWin所描述的输入法窗口没有目标窗口。

        第二情景是变量imWin的值等于null，这时候系统中不存在输入法窗口。这个情景与前面第一种情况的第二个情景也是类似的。由于系统中不存在输入法窗口，因此只需要调用WindowManagerService类的成员函数moveInputMethodDialogsLocked来间接地输入法对话框在窗口堆栈中的位置即可，即以参数-1来调用WindowManagerService类的成员函数moveInputMethodDialogsLocked。

        至此，我们就分析完成WindowManagerService服务对输入法窗口的基本操作了。从分析的过程中，我们可以得到以下两个结论：

        A. 系统中与输入法相关的窗口有两种，一种是输入法窗口，另一种是输入法对话框。

        B. 当Z轴位置最大的窗口需要使用输入法时，输入法窗口就会位于它的上面，而输入法对话框又会位于输入法窗口的上面。

        在WindowManagerService服务中，还有一种类型的窗口与输入法窗口类似，它总是与Activity窗口粘在一起。不过，这种类型的窗口是位于Activity窗口的下面，刚好与输入法窗口相反，它就是壁纸窗口（Wallpaper）。在接下来的一篇文章中，我们就将继续分析WindowManagerService服务是如何管理系统中的壁纸窗口的。敬请关注！

### WindowManager解析（二）Android悬浮框无法弹出输入法的原因和无需权限显示悬浮窗

* 来源:csdn
* 作者:iromkoear
* 2017-03-31 22:22:38
* 29
* 1

android  悬浮框

  来自睡不着起不来的万先生 的 Android悬浮窗使用TYPE\_TOAST的小结 最后 TYPE\_TOAST一直都可以显示, 但是用TYPE\_TOAST显示出来的在2.3上无法接收点击事件, 因此还是无法随意使用. 下面是我之前研究后台线程显示对话框的时候记得笔记, 大家可以看看我们项目中有需求需要在后台任务中显示Dialog, 项目最初的做法是用Activity模拟Dialog, 一个Ac

## ****Android悬浮框无法弹出输入法****

最近要研究悬浮窗方面的东西，遇到一个问题，我的悬浮窗里面有一个输入框，但是不弹出输入法，后来找到一个方法：

在WindowManager的实例获取方式不对，之前是这样获取的：

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

改这样：

mWindowManager = (WindowManager)

mContext.getApplicationContext().getSystemService(Context.WINDOW\_SERVICE);

一个是通过当前activity的上下文环境去获取窗口服务，一个是通过application去获取窗口服务

发现还是不行，后来我花了很长时间用谷歌翻译了WindowManager的源码英文注释才发现，原来我之前是用的：

bigWindowParams.flags = WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE

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

LayoutParams.FLAG\_NOT\_TOUCH\_MODAL的意思是：

/\*\*

窗口标志：即使该窗口是可对焦的（其#FLAG\_NOT\_FOCUSABLE未设置），

允许窗口外的任何指针事件发送到其后面的窗口。

否则它将消耗所有指针事件本身，而不管它们是否在窗口内。

\*/

public static final int FLAG\_NOT\_TOUCH\_MODAL = 0x00000020;

即不会阻挡后面的点击事件

LayoutParams.FLAG\_NOT\_FOCUSABLE的意思是：

/\*\*

窗口标志：此窗口不会获得关键输入焦点，因此用户无法向其发送键或其他按钮事件。

那些会改变它背后的任何可关注的窗口。

此标志还将启用#FLAG\_NOT\_TOUCH\_MODAL是否显式设置。

\*

\*

设置此标志也意味着该窗口不需要与软输入法进行交互，

因此它将被Z-排序并且与任何活动输入法无关（通常这意味着它在输入法之上得到Z-排序，

所以它可以使用全屏的内容，如果需要的话可以覆盖输入法。

可以使用{@link #FLAG\_ALT\_FOCUSABLE\_IM}来修改这个行为。

\* \*/

public static final int FLAG\_NOT\_FOCUSABLE = 0x00000008;

该窗口不需要与软输入法进行交互，说明是不会弹出输入框的，把这个去掉，就能正常显示出输入法了，不过去掉了也会造成另外一个问题，那就是系统的返回键用不了！

要想在WindowManager中直接添加EditText并且能够输入文字，而且还不影响其他操作的，需要保证以下两点：

type 为 LayoutParams.TYPE\_SYSTEM\_ALERT 而不是 LayoutParams.TYPE\_SYSTEM\_ERROR

防止没有三个按键的问题   
flag中不要有 WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE

防止无法弹出输入框的问题   
flag中要有 WindowManager.LayoutParams.FLAG\_NOT\_TOUCH\_MODAL

防止下方app无法获取焦点和事件的问题

看来花费了这么多时间翻译了一下源码的注释还是很有好处的，起码知道了用这个属性是为什么，也说明要想学得好还是得多看源码，翻译一下注释。

## ****无需权限显示悬浮窗****

#### ****悬浮窗原理****

做过悬浮窗功能的人都知道, 要想显示悬浮窗, 要有一个服务运行在后台, 通过getSystemService(Context.WINDOW\_SERVICE)拿到WindowManager, 然后向其中addView, addView第二个参数是一个WindowManager.LayoutParams, WindowManager.LayoutParams中有一个成员type, 有各种值, 一般设置成TYPE\_PHONE就可以悬浮在很多view的上方了, 但是调用这个方法需要申请android.permission.SYSTEM\_ALERT\_WINDOW权限, 在很多机型上, 这个权限的名字叫悬浮窗, 比如小米手机上默认是禁用这个权限的, 有些恶意app会用这个权限弹广告, 而且很难追查是哪个应用弹的. 如果这个权限被禁用, 那么结果就是悬浮窗无法展示, 比如有道词典的复制查词功能, 在小米手机上经常没用, 其实是用户没有授权, 而且应用也没有引导用户给它打开授权。

#### ****使用TYPE\_TOAST可以显示悬浮框****

使用 type 值为 WindowManager.LayoutParams.TYPE\_PHONE 和 WindowManager.LayoutParams.TYPE\_SYSTEM\_ALERT 需要申请 android.permission.SYSTEM\_ALERT\_WINDOW 权限，否则无法显示，报错：

E/AndroidRuntime: android.view.WindowManager$BadTokenException: Unable to add window android.view.ViewRoot$W@b64b5458 -- permission denied for this window type

但是，将type设置成TYPE\_TOAST果然有奇效, 不需要android.permission.SYSTEM\_ALERT\_WINDOW权限就能显示一个悬浮窗.

#### ****原因****

在Android源码中有这么一段：

public int checkAddPermission(WindowManager.LayoutParams attrs) {

int type = attrs.type;

if (type < WindowManager.LayoutParams.FIRST\_SYSTEM\_WINDOW

|| type > WindowManager.LayoutParams.LAST\_SYSTEM\_WINDOW) {

return WindowManagerImpl.ADD\_OKAY;

}

String permission = null;

switch (type) {

case TYPE\_TOAST:

// XXX right now the app process has complete control over

// this... should introduce a token to let the system

// monitor/control what they are doing.

break;

case TYPE\_INPUT\_METHOD:

case TYPE\_WALLPAPER:

// The window manager will check these.

break;

case TYPE\_PHONE:

case TYPE\_PRIORITY\_PHONE:

case TYPE\_SYSTEM\_ALERT:

case TYPE\_SYSTEM\_ERROR:

case TYPE\_SYSTEM\_OVERLAY:

permission = android.Manifest.permission.SYSTEM\_ALERT\_WINDOW;

break;

default:

permission = android.Manifest.permission.INTERNAL\_SYSTEM\_WINDOW;

}

if (permission != null) {

if (mContext.checkCallingOrSelfPermission(permission)

!= PackageManager.PERMISSION\_GRANTED) {

return WindowManagerImpl.ADD\_PERMISSION\_DENIED;

}

}

return WindowManagerImpl.ADD\_OKAY;

}

这个方法是往系统的WindowManager里addView的时候做权限检查用的, 那个type就是我们在构造WindowManager.LayoutParams时赋值的type, 可以看到, 除了TYPE\_TOAST, 其他都是要权限的, 而且非常喜感的是, 代码中的注释还说他们现在对这种type毫无限制, 应该引入标记来限制开发者.

#### ****兼容性****

1. type 值为 WindowManager.LayoutParams.TYPE\_TOAST 显示的 System overlay view 不需要权限，即可在任何平台显示。
2. 但仅在 API level >= 19 时可以达到目的。API level 19 以下因无法接收无法接收触摸（点击)和按键事件，故无法达到目的。
3. 对于 API level < 19 的机器（MIUI除外），想要达到目的，需要:

要有 android.permission.SYSTEM\_ALERT\_WINDOW 权限   
将 type 设置为 WindowManager.LayoutParams.TYPE\_PHONE 或者 WindowManager.LayoutParams.TYPE\_SYSTEM\_ALERT   
MIUI V5

在给悬浮窗权限之后，表现同 3 。但是，不给悬浮窗权限时，应用在前台时，却可以显示。这点非常不一样。

#### ****为什么采用 TYPE\_TOAST****

API level 19 之后，TYPE\_SYSTEM\_ALERT 就可以达到 TYPE\_SYSTEM\_ALERT 效果（即可以达到目的）。

但在 API level 23 (Android 6.0) 上，悬浮窗权限也单独弄出来了，需要到单独开启，这个处理和现在 Smartisan 还有小米类似。

故采用 TYPE\_TOAST 能让用不开启悬浮窗权限的情况下，也能显示。为的就是尽量少请求权限。

#### ****为什么 API level 19 之后 TYPE\_TOAST 可以接受事件****

PhoneWindowManager.adjustWindowParamsLw()，API level 19 后做了调整。

当我们使用 TYPE\_TOAST, Android 会偷偷给我们加上 FLAG\_NOT\_FOCUSABLE 和 FLAG\_NOT\_TOUCHABLE , 4.0.1 开始, 会额外再去掉 FLAG\_WATCH\_OUTSIDE\_TOUCH。 这样真的是什么事件都没了。

而4.4开始, TYPE\_TOAST被移除了。所以从 4.4 开始, 使用 TYPE\_TOAST 的同时还可以接收触摸事件和按键事件了, 而4.4以前只能显示出来, 不能交互。

[来自睡不着起不来的万先生 的 Android悬浮窗使用TYPE\_TOAST的小结](http://www.jianshu.com/p/634cd056b90c)

#### ****最后****

TYPE\_TOAST一直都可以显示, 但是用TYPE\_TOAST显示出来的在2.3上无法接收点击事件, 因此还是无法随意使用.

下面是我之前研究后台线程显示对话框的时候记得笔记, 大家可以看看我们项目中有需求需要在后台任务中显示Dialog, 项目最初的做法是用Activity模拟Dialog, 一个Activity已经承载了近20种Dialog, 代码混乱至极. 后来我发现Dialog可以通过改变Window Type实现不依赖Activity显示, 然后就很兴奋的要在使用这种方式来作为新的实现方式.

最初WindowType是WindowManager.LayoutParams.TYPE\_SYSTEM\_ALERT, 可是这是悬浮窗了, MIUI会默认禁止(真他妈操蛋，也没有任何提示)最终放弃. 后来试着换成了WindowManager.LayoutParams.TYPE\_TOAST, 起初效果很好，MIUI也不禁止了, 哪里都能显示, 这下开心了. 可是后来又发现在2.3上不能接收点击事件, 也就是说Dialog上的按钮不能点击, 这他妈就很操蛋了, 又放弃了. 又试了试其他的Type都不能满足需求, 结果如下：TYPE\_SEARCH\_BAR: 未知

TYPE\_ACCESSIBILITY\_OVERLAY: 拒绝使用

TYPE\_APPLICATION: 只能配合Activity在当前APP使用TYPE\_APPLICATION\_ATTACHED\_DIALOG: 只能配合Activity在当前APP使用

TYPE\_APPLICATION\_MEDIA: 无法使用(什么也不显示)

TYPE\_APPLICATION\_PANEL: 只能配合Activity在当前APP使用(PopupWindow默认就是这个Type)

TYPE\_APPLICATION\_STARTING: 无法使用(什么也不显示)

TYPE\_APPLICATION\_SUB\_PANEL: 只能配合Activity在当前APP使用TYPE\_BASE\_APPLICATION: 无法使用(什么也不显示)

TYPE\_CHANGED: 只能配合Activity在当前APP使用

TYPE\_INPUT\_METHOD: 无法使用(直接崩溃)

TYPE\_INPUT\_METHOD\_DIALOG: 无法使用(直接崩溃)

TYPE\_KEYGUARD\_DIALOG: 拒绝使用

TYPE\_PHONE: 属于悬浮窗(并且给一个Activity的话按下HOME键会出现看不到桌面上的图标异常情况)

TYPE\_TOAST: 不属于悬浮窗, 但有悬浮窗的功能, 缺点是在Android2.3上无法接收点击事件

TYPE\_SYSTEM\_ALERT: 属于悬浮窗, 但是会被禁止

## ****其他的实战文章****

#### [让你的app无法使用系统截图的探究](http://www.androidchina.net/5938.html)

想要像银联一样，在某Activity做到手机无法截屏，甚至是adb也拿不到，那么可以在Activity中加入:

getWindow().addFlags(WindowManager.LayoutParams. FLAG\_SECURE);

源码:

/\*\*

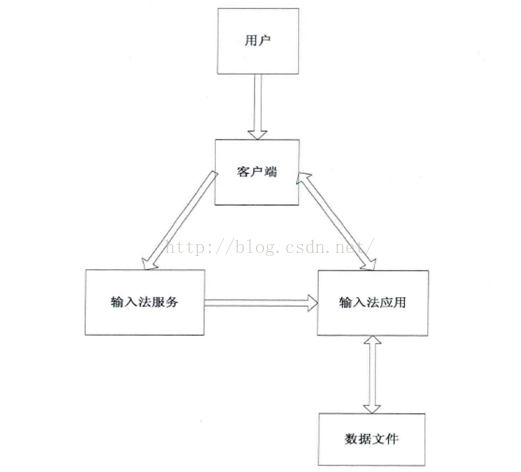
窗口标志：将窗口的内容视为安全的，防止窗口显示在屏幕截图中或在非安全显示器上查看。

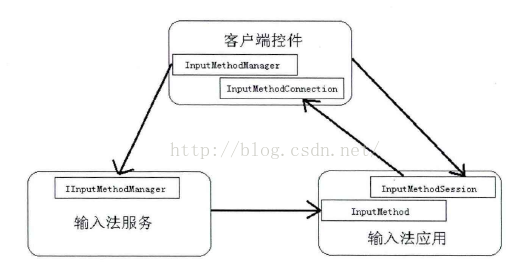
有关安全表面和安全显示的更多详细信息，请参阅android.view.Display＃FLAG\_SECURE。

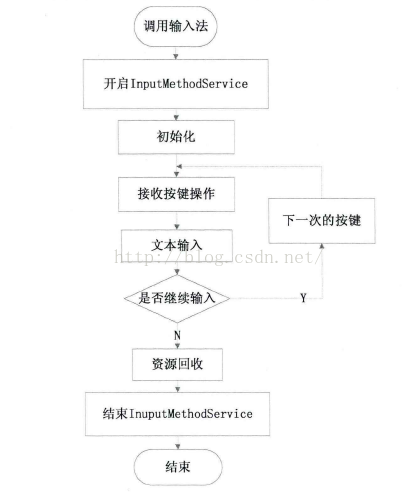
\*/

public static final int FLAG\_SECURE = 0x00002000;

## ****参考文章****

* [Android无需权限显示悬浮窗, 兼谈逆向分析app](http://www.jianshu.com/p/167fd5f47d5c)
* [Android悬浮窗TYPE\_TOAST小结: 源码分析](http://www.jianshu.com/p/634cd056b90c)
* [Android 悬浮窗的小结](https://www.liaohuqiu.net/cn/posts/android-windows-manager/)
* [Android项目:输入法软键盘显示/隐藏的监听和控制,InputMethodManager用法研究](http://glblong.blog.51cto.com/3058613/1543172)
* [在Activity，Service，Window中监听Home键和返回键的一些思考，如何把事件传递出来的做法！](http://blog.csdn.net/qq_26787115/article/details/52260393)
* [Android悬浮框中EditText无法获取焦点/获取焦点之后按键失灵的问题](http://www.aprilwei.com/article/19)
* [overlays-floating-window-like-facebook-messenger](https://github.com/atifmahmood29/overlays-floating-window-like-facebook-messenger)
* [使用WindowManager实现Android悬浮窗](http://www.jcodecraeer.com/a/anzhuokaifa/androidkaifa/2014/1204/2103.html)
* **Android输入法架构学习总结**
* 2016年08月30日 21:51:10 [温柔狠角色](https://me.csdn.net/qq_25827845) 阅读数：10348更多
* 所属专栏： [Android 开发设计](https://blog.csdn.net/column/details/12846.html)
* 版权声明：本文为博主原创文章，转载请注明出处，冷血之心的博客。 https://blog.csdn.net/qq\_25827845/article/details/52373345
* **此文为本人学习输入法之后所做的一个总结报告。与大家分享。**
* **安卓输入法框架（Input Method Framework）IMF**
* **一．输入法框架简介**
* 自Android平台1.5版本以后，Google开放了 Android平台输入法框架(InputMethod Framework, IMF) ， IMF是Android平台的特色设计。它的出现大大推动了不带实体键盘的设备的诞生。同时，Android平台输入法框架也给出了输入法的开发所需要的接口，为Android平台的输入法提供了可扩展性。
* **二．输入法框架组成**
* 一个IMF结构中包含三个主要的部分：  
  ●（输入法服务）input method manager(IMM)：管理各部分的交互，是一个客户端API，存在于各个应用程序的context中，用来沟通管理所有进程间交互的全局系统服务，可以通过Context.getSystemService()来获取一个InputMethodManager的实例。  
  ●（输入法应用）input method(IME)：实现一个允许用户生成文本的独立交互模块。系统绑定一个当前的输入法。使其创建和生成，决定输入法何时隐藏或者显示它的UI。同一时间只能有一个IME运行。  
  ●（客户端）client application：通过输入法管理器控制输入焦点和IME的状态。一次只能有一个客户端使用IME。
* 输入法整体框架如图1所示：
* 
* 图1.输入法的总体框架
* **2.1 IMM的实现**
* InputMethodManagerService.java是整个系统当中，一切与输入法有关的地方的总控制中心。它通过管理下面三个模块来实现系统的输入法框架。
* （1）WindowManagerService
* 负责显示输入法，接收用户事件。
* （2）InputMethodService
* 输入法内部逻辑，键盘布局，选词等，最终把选出的字符通过 commitText提交出来。
* （3）InputManager
* 由 UI 控件（View,TextView,EditText等）调用，用来操作输入法。比如，打开，关闭，切换输入法等。
* **2.2 输入法应用组成**
* 在IMF中，最主要的是输入法应用，他继承于Abstract InputMethodService。InputMethodService主要由以下几个组件构成,其中包括完成输入法的相关UI和文字的输出。
* （1）软键盘视图(SoftInput View)
* 这是软键盘的输入区域,主要完成在触摸屏下和用户的交互输入。onCreateInputView()被调用来进行软键盘视图的实例化;onEvaluateInputViewShown()决定是否显7K软键盘视图;当状态改变的时候,调用updateInputViewShownO来重新决策是否显示软键盘视图。
* （2）候选字视图(CandidatesView)
* Candidates View也是输入法中一个相当重要的组件。当用户输入字符的时候,显不相关的列表。停止输入的时候,就会自动消失。onCreateCandidatesView()来实例化自己的输入法。和软键盘视图不同的是,候选字视图对整个UI布局不会产生影响。setCandidatesViewShown(boolean)用来设置是否显示候选字视图。
* （3）输出字符
* 字符的输出是InputMethodService最核心的功能，输入法通过InputConnection从IMF来获得字符输出。并且通过不同的编辑器(editor)类型来获取相应的支持。通过 onFinishlnputO和onStartInput(EditorInfo, boolean)方法来进行输入目标的切换。
* 另外，
* onlnitializelnterfaceO用于InputMethodService在执行的过程中配置的改变;
* onBindlnputO切换一个新的输入通道;
* onStartInput(EditorInfo, boolean)处理一个新的输入。

* **三．输入法消息流转机制**
* 输入法框架包括客户端、输入法服务和输入法应用三部分组成，如图2所示。输入法框架中消息的流转机制为：当客户端获得焦点，启动输入法，创建一个连接类型对象，以实现输入法框架各个层次间信息的传输。该连接类型实现了InputConnection接口。
* 
* 图2.输入法消息流转

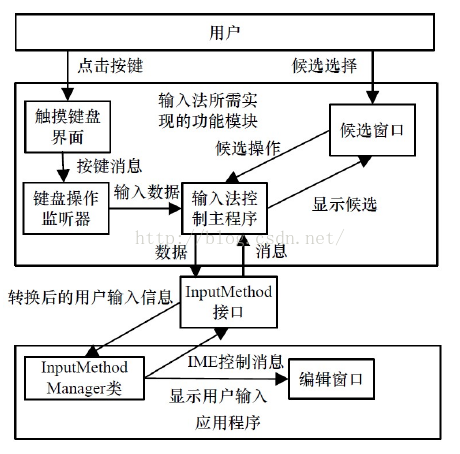
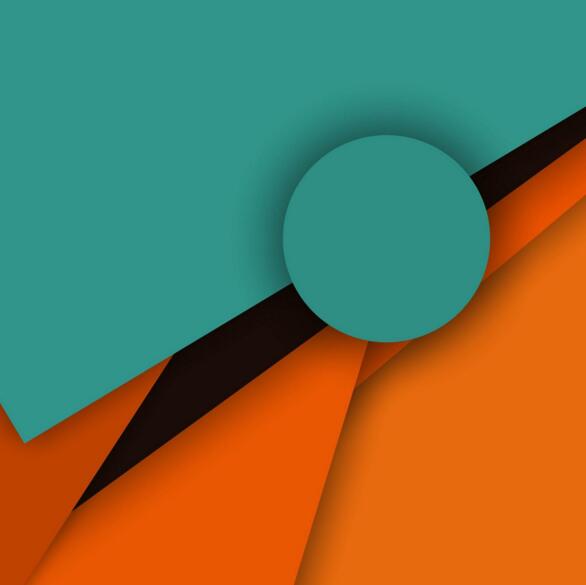
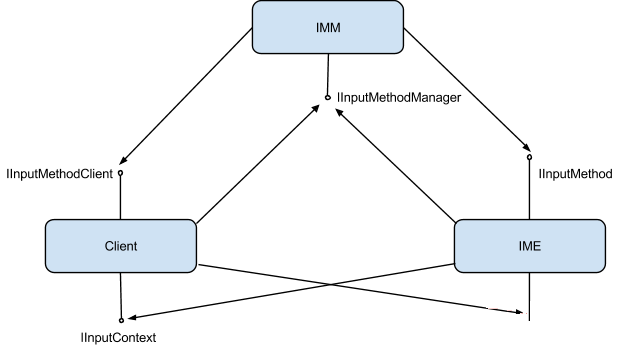
* **四．输入法生命周期**
* 输入法是用户、应用程序之间的交互的渠道，所有的输入法应用都需要继承特定的Android平台提供的服务。Android平台的输入法框架为输入法应用定义了一个基类InputMethodService，InputMethodService提供了一个输入法的标准实现流程，定义了输入法生命周期内的重要函数，以方便开发人员对Android输入法进行扩展。图3即为Android输入法生命周期。
* 
* 图3.Android输入法生命周期
* 具体的实现步骤为：
* 1.当用户点击客户端输入控件(如editor)，客户端控件获得焦点，InputMethodService启动，调用其onCreate()函数。该函数在输入法第一次启动的时候被调用，用来做初始化的设置；
* 2. 调用onCreatelnputViewO函数，在该函数中创建键盘视图(KeyboardView)并返
* 回；
* 3. 调用onCreateCandidatesView()函数，在该函数中创建候选字视图(Candidates
* View)实现并返回；
* 4. 调用onStartlnput()函数，始接收并处理输入内容；
* 5. 输入结束后调用onFinishlnputO函数来结束当前的输入；
* 6. 如果移动到下一个输入框，则重复调用onStartlnputView和onFinishlnput函数；
* 7. 在输入法关闭的时候调用onDestroy()函数。

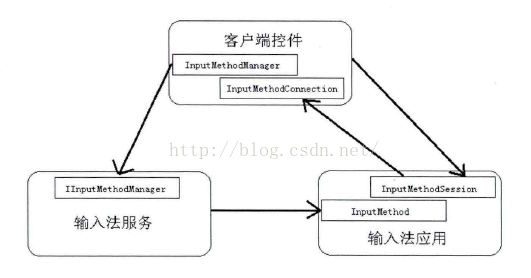
* **五．输入法主要的类和函数**
* 在Android输入法开发过程中,下面三个类的实现比较重要:
* InputMethodService类：提供了很多的输入法的基础实现，管理状态的条款、输入法的可见度、与当前可见的Activity沟通;
* CandidateView类：提供候选字选择视图,直接继承于View。负责显示软键盘上面的那个候选区域。
* LatinKeyboard类：软键盘类,直接继承与Keyboard类。负责解析并保存键盘布局，并提供选词算法，供程序运行当中使用。其中键盘布局是以XML文件存放在资源当中的。
* Android的输入法服务，通过一些接口函数与系统进行交互，在这些接口中，主要接口是InputMethodService。它提供了一个输入法的标准实现，开发输入法时可以参照和自定义该实现，表1显示了InputMethodService接口提供的函数。

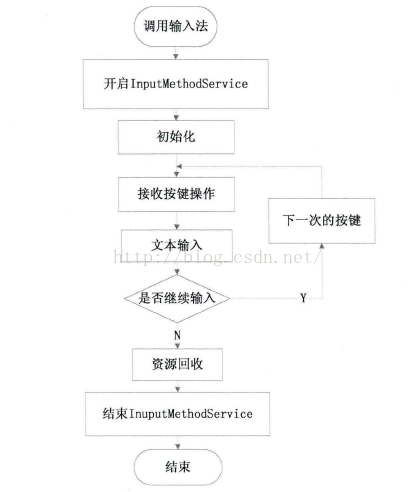
|  |  |
| --- | --- |
| 主要接口函数 | 说明 |
| onlnitializelnterface | 进行UI的初始化,创建和修改时调用此接口 |
| onBindlnput | 在另外的客户端和该输入法连接时调用 |
| onStartlnput | 初始化整个输入法,非常重要的一个回调,它在  编辑框中用户已经开始输入的时候调用 |
| onCreatelnputView | 创建输入视图,返回一个层次性的输入视图,而且只是在这个视图第一次显示的时候被调用 |
| onCreateCandidatesView | 创建候选框视图,当需要候选框显示时调用 |
| onCreateExtractTextView | 全屏模式下的视图 |
| onStartlnput  View | 在输入视图被显示并且在一个新的输入框中输入已经开始的时候调用 |
| getCurrentlnputConnection | 此函数呼叫应用程序,接收原始的按键事件 |
| onFinishlnput | 结束输入法 |

* 表1.InputMethodService接口提供的函数

* **六．输入法服务流程**
* **6.1客户端应用注册到输入法系统服务**
* **6.1.1创建LocationManger**
* 1）客户端应用创建时，调用ViewRoot(…)；
* 2）调用ViewRoot .getWindowSession(…)；
* 3）通过调用InputMethodManager.getInstance()创建LocationManager对象，一个客户端应用只会创建一个LocationManager对象；
* 4）LocationManager对象创建时，会创建一个IInputMethodClient对象，同时创建一个IInputContext对象；
* **6.1.2注册到输入法系统服务**
* 1）调用WindowMangerService. openSession(…)；
* 2）调用WindowMangerService. Session(…)；
* 3）调用InputMethodMangerService.addClient(IInputMethodClientclient, IInputContext inputContext, int uid, int pid)，将InputMethodManager中创建的IInputMethodClient对象以及InputMethodManager中创建的IInputContext对象传入进去，其中uid为客户端应用用户ID，pid为客户端应用进程ID；
* 4）加入到InputMethodManagerService维护的一个列表HashMap<IBinder, ClientState>中，其中IBinder对应IInputMethodClient，一个InputMethodManager只有一个IInputMethodCliend。
* **6.2客户端应用调用输入法**
* 1）TextView.setText();
* 2）调用InputMethodManager.restartInput();
* 3）调用InputMethodManager. startInputInner();
* 4）  调用InputMethodManagerService. startInput(IInputMethodClient,IInputContext…);
* **6.3输入法系统服务调用输入法**
* 输入法系统服务调用输入法包括输入法系统服务启动输入法，输入法创建，输入法系统服务建立与输入法的会话，输入法系统服务启动输入法以及输入法服务显示输入法。

* **七．Android手机输入法设计的主要内容**
* 一款手机输入法能否得到用户的认可，主要取决于输入法的易用性和高效性以及界面的美观度。因此，IME程序的设计主要是用户交互接口的设计和编码转换引擎的设计。
* 首先，对于软键盘输入法来说，软键盘的布局设计在很大程度上影响着用户的使用，同时，由于手机设备的屏幕大小限制，如何处理输入法中软键盘和候选框、组合框、状态窗口的相对位置和大小也是输入法界面设计的重要部分。
* 其次，由于智能终端设备的处理器和内存的限制，如何设计一个系统内存占用小，CPU使用率低的编码转换引擎和词库存储方案也是手机输入法设计的重要组成部分。
* 具体来说，IMF框架以及要编写 Android 输入法需要实现的模块如图4所示。
* 
* 图 4.Android 输入法模块
* 详细分析Android输入法框架的三大组件
* 作者：课课家教育 http://www.kokojia.com点击数：11945发布时间：2016-04-05 10:55:06
* 现在输入法在我们日常生活中是不可缺少的一部分了，因为现在科学技术的快速发展，很多电子产品都需要输入各种各样的字或者数字或者字母等输入，现在比较常见的是在手机端或者PC端上，都有使用各式各样的输入方式，比如搜狗、百度、五笔等。本文就详细介绍[Android](http://www.kokojia.com/list/260.html)输入法框架的三大组件。
* 
* Android输入法框架(Input Method Framework，IMF)是Android中非常重要的模块，它分布于三个部分(确切的说，是三个进程)，编辑框的客户、输入法服务和输入法管理者服务。编辑框的客户(Client)[app](http://www.kokojia.com/s531/)，表示普通的使用输入法的app进程。当点击编辑框时，会切换出当前选中的输入法;当用户在输入法输入字符，提交候选词，则会更新到编辑框中。为了完成这些行为，它需要跟下面的两个输入法相关服务进行交互。对于普通[app开发](http://www.kokojia.com/s2893/)者而言，他们一般使用系统提供的EditText，该类和其父类TextView已经很好的封装了跟输入法服务之间的交互;如果是自定义的编辑框，则需要自己处理这种交互。输入法(input method，IME)服务(service)，是具体的输入法进程，例如自带的拉丁输入法或者谷歌，搜狗等拼音输入法。它们一般提供一个输入窗口，可以根据用户的要求打开或者关闭;可以把用户输入的字符和提交的候选词更新给client等等。这是一个用户级别的Service。为了方便开发者编写新的输入法，IMF提供了抽象基类InputMethodService供输入法开发者扩展。输入法管理者(Input method manager，IMM)服务(Service)，这是一个Android系统级的服务，用于管理多个输入法以及同其他系统服务(例如[window](http://www.kokojia.com/s1444/) manager service)进行交互。这部分代码是app开发者和IME开发者都不需要关心的。
* **为了方便描述，后文分别称该三个组件为：client，IME和IMM。**
* 这三个部分需用共同合作才能完成输入法的工作。例如打开一个app，并且一个edit框获取了focus焦点。此时client会通知IMM打开输入法，然后IMM查看当前选中的IME，并调用该IME的start操作。这个简单的开始操作需要三个组件的配合。再比如用户提交了候选词，此时IME需要将候选词告诉client。这里须要IME和client的合作。
* 因为这三个部分是三个进程，所以它们之间必须通过IPC进行通讯。在Android中，IPC机制是通过binder机制和aidl接口进行通信的。
* 对于Client而言，它提供了两个接口IInputMethodClient.aidl和IInputContext.aidl。前者是供IMM调用的，后者是供IME调用的。IMM提供了接口IInputMethodManager.aidl供其他两个组件调用。IME提供了两个接口IInputMethod.aidl和IInputMethodSession.aidl，前者供IMM调用，后者供client直接调用。
* **这些调用关系可以参考下图：**
* 
* **这些接口定义都在**[**java**](http://www.kokojia.com/list/201.html)**/com/android/internal/view目录下。那这些接口是如何实现的呢?**
* 先看client提供的接口。IInputContext是由同一目录下的IInputConnectionWrapper实现的。正如名字所说，它只是一个wrapper，它把接收到的IPC消息委托给你InputConnection的一个实现。例如对于EditText而言，实现是EditableInputConneciton。
* 在调用方，IME也不是直接操作IInputContext接口。它会调用实现了InputConnection接口的InputConnectionWrapper(也在前面目录下)。该对象封装了从client传过来的IInputContext实例。
* 对于IME对client的调用操作，它会经历下面流程(以调用commitText为例，它表示提交候选词)：
* InputConnectionWrapper.commitText被IME进程中其他代码调用。委托给IInputContext stub对象。通过IPC跨进程传输IInputConnectionWrapper接受到该消息并调用其commitText处理。如果当前在主UI线程，则直接嗲用InputConnection的实现(例如EditableInputConnection)的commitText方法;否则通过handler进行线程间通信。
* 在IME看来，接口是InputConnection;在client上，实现的也是InputConnection。IInputContext完全被隐藏起来了。所以Android官方文档说IME通过InputConnection接口来操作client。
* 再看client提供的另外一个接口IInputMethodClient，IMM是直接调用的。IMM的代码就是InputMethodManagerService。在client端，InputMethodManager类中有一个对IInputMethodClient.stub的实现。
* 对IMM提供的IInputMethodManager接口而言，它是由InputMethodManagerService来实现的。在client端，InputMethodManager的getInstance(是个singleton)会调用ServiceManager.getService(Context.INPUT\_METHOD\_SERVICE)获取该接口，然后创建InputMethodManager。所以对于client而言，它跟IMM的交互都是通过InputMethodManager来封装完成的，并不需要关心IInputMethodManager接口。对于IME，如果它想操作IMM，也同样通过InputMethodManager。
* 下面是IME提供的接口。类似于使用InputConnection封装IInputContext，有两个接口InputMethod和InputMethodSession分别对应着了IInputMethod和IInputSession。
* 对于InputMethod，IInputMethodWrapper实现了IInputMethod.stub。对于收的的IPC请求，都转发给InputMethod实例。一般而言，这个实例是InputMethodService中定义的InputMethodImpl。该实例是InputMethodService的内部类，所以可以操作InputMethodService。对于其客户IMM，InputMethodManagerService会直接调用IInputMethod的方法发起IPC请求。
* 对于InputMethodSessoin，非常类似，IInputMethodSessionWrapper实现了IInputMethodSession.stub。同样在InputMethodService中有InputMethodSessionImpl实现了InputMethodSession接口，有一个该类型的对象在IInputMethodSessionWrapper中，负责具体处理过来的IPC消息。在client端，InputMethodManager有一个IInputMethodSession mCurMethod对象。开发者只需要调用InputMethodManager，而由InputMethodManager调用IInputMethodSession的IPC操作。
* 总结一下，无论是client还是IME的开发者，都不需要直接操作aidl接口。在client端，对于IMM和IME的操作都是通过InputMethodManager发起的，用户甚至不用关心这些IPC操作是发给谁的;在IME端，开发者通过InputConnection给client发IPC消息，通过InputMethodManager给IMM发。而在IMM端，虽然是直接操作aidl接口的stub对象，但因为一般开发者不需要改写它，所以也无关紧要。通过这种方式，简化了开发者的跨进程操作。
* **最后再总结下代码位置：**
* 所有的aidl接口都在java/com/android/internal/view目录下。它只是internal可见，所以普通开发者无法直接访问它。用户可以访问的接口或类，包括InputConnection，InputMethodManager，InputMethod，InputMethodSession都在java/android/view/inputmethod目录下。java/com/android/internal/view还包含IInputConnecitonWrapper，InputConnectionWrapper。java/android/view/inputmethod还包含BaseInputConnectionjava/android/inputmethodservice包含IME端相关代码，例如IInputMethodWrapper，IInputMethodSessionWrapper，InputMethodImpl，InputMethodSessionImpl，InputMethodService。IMM的InputMethodManagerService在services/java/com/android/server下。
* 代码主要就是这四个目录。
* 我们详细介绍了Android输入法框架的编辑框的客户、输入法服务和输入法管理者服务的三大部分，还详细介绍了三大组件操作接口的步骤，希望能帮助你理解Android输入法框架。
* Android输入法架构学习总结
* [几米](https://www.demoso.net/users/1)1年前 ⋅ 1076 阅读
* *标签：* [android](https://www.demoso.net/tag/android) [输入法](https://www.demoso.net/tag/%E8%BE%93%E5%85%A5%E6%B3%95) [架构](https://www.demoso.net/tag/%E6%9E%B6%E6%9E%84) [学习报告](https://www.demoso.net/tag/%E5%AD%A6%E4%B9%A0%E6%8A%A5%E5%91%8A)
* **此文为本人学习输入法之后所做的一个总结报告。与大家分享。**
* **安卓输入法框架（Input Method Framework）IMF**
* **一．输入法框架简介**
* 自Android平台1.5版本以后，Google开放了 Android平台输入法框架(InputMethod Framework, IMF) ， IMF是Android平台的特色设计。它的出现大大推动了不带实体键盘的设备的诞生。同时，Android平台输入法框架也给出了输入法的开发所需要的接口，为Android平台的输入法提供了可扩展性。
* **二．输入法框架组成**
* 一个IMF结构中包含三个主要的部分：  
  ●（输入法服务）input method manager(IMM)：管理各部分的交互，是一个客户端API，存在于各个应用程序的context中，用来沟通管理所有进程间交互的全局系统服务，可以通过Context.getSystemService()来获取一个InputMethodManager的实例。  
  ●（输入法应用）input method(IME)：实现一个允许用户生成文本的独立交互模块。系统绑定一个当前的输入法。使其创建和生成，决定输入法何时隐藏或者显示它的UI。同一时间只能有一个IME运行。  
  ●（客户端）client application：通过输入法管理器控制输入焦点和IME的状态。一次只能有一个客户端使用IME。
* 输入法整体框架如图1所示：
* 图1.输入法的总体框架
* **2.1 IMM的实现**
* InputMethodManagerService.java是整个系统当中，一切与输入法有关的地方的总控制中心。它通过管理下面三个模块来实现系统的输入法框架。
* （1）WindowManagerService
* 负责显示输入法，接收用户事件。
* （2）InputMethodService
* 输入法内部逻辑，键盘布局，选词等，最终把选出的字符通过 commitText提交出来。
* （3）InputManager
* 由 UI 控件（View,TextView,EditText等）调用，用来操作输入法。比如，打开，关闭，切换输入法等。
* **2.2 输入法应用组成**
* 在IMF中，最主要的是输入法应用，他继承于Abstract InputMethodService。InputMethodService主要由以下几个组件构成,其中包括完成输入法的相关UI和文字的输出。
* （1）软键盘视图(SoftInput View)
* 这是软键盘的输入区域,主要完成在触摸屏下和用户的交互输入。onCreateInputView()被调用来进行软键盘视图的实例化;onEvaluateInputViewShown()决定是否显7K软键盘视图;当状态改变的时候,调用updateInputViewShownO来重新决策是否显示软键盘视图。
* （2）候选字视图(CandidatesView)
* Candidates View也是输入法中一个相当重要的组件。当用户输入字符的时候,显不相关的列表。停止输入的时候,就会自动消失。onCreateCandidatesView()来实例化自己的输入法。和软键盘视图不同的是,候选字视图对整个UI布局不会产生影响。setCandidatesViewShown(boolean)用来设置是否显示候选字视图。
* （3）输出字符
* 字符的输出是InputMethodService最核心的功能，输入法通过InputConnection从IMF来获得字符输出。并且通过不同的编辑器(editor)类型来获取相应的支持。通过 onFinishlnputO和onStartInput(EditorInfo, boolean)方法来进行输入目标的切换。
* 另外，
* onlnitializelnterfaceO用于InputMethodService在执行的过程中配置的改变;
* onBindlnputO切换一个新的输入通道;
* onStartInput(EditorInfo, boolean)处理一个新的输入。

* **三．输入法消息流转机制**
* 输入法框架包括客户端、输入法服务和输入法应用三部分组成，如图2所示。输入法框架中消息的流转机制为：当客户端获得焦点，启动输入法，创建一个连接类型对象，以实现输入法框架各个层次间信息的传输。该连接类型实现了InputConnection接口。
* 
* 图2.输入法消息流转

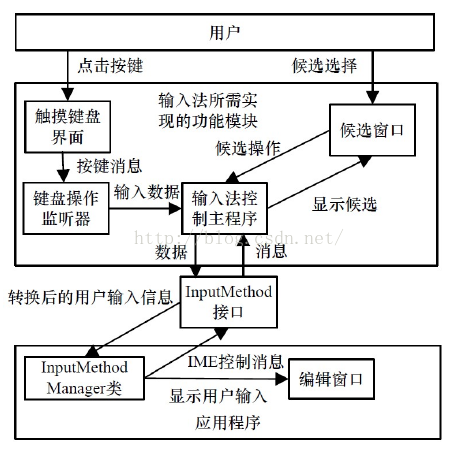
* **四．输入法生命周期**
* 输入法是用户、应用程序之间的交互的渠道，所有的输入法应用都需要继承特定的Android平台提供的服务。Android平台的输入法框架为输入法应用定义了一个基类InputMethodService，InputMethodService提供了一个输入法的标准实现流程，定义了输入法生命周期内的重要函数，以方便开发人员对Android输入法进行扩展。图3即为Android输入法生命周期。
* 
* 图3.Android输入法生命周期
* 具体的实现步骤为：
* 1.当用户点击客户端输入控件(如editor)，客户端控件获得焦点，InputMethodService启动，调用其onCreate()函数。该函数在输入法第一次启动的时候被调用，用来做初始化的设置；
* 2. 调用onCreatelnputViewO函数，在该函数中创建键盘视图(KeyboardView)并返
* 回；
* 3. 调用onCreateCandidatesView()函数，在该函数中创建候选字视图(Candidates
* View)实现并返回；
* 4. 调用onStartlnput()函数，始接收并处理输入内容；
* 5. 输入结束后调用onFinishlnputO函数来结束当前的输入；
* 6. 如果移动到下一个输入框，则重复调用onStartlnputView和onFinishlnput函数；
* 7. 在输入法关闭的时候调用onDestroy()函数。

* **五．输入法主要的类和函数**
* 在Android输入法开发过程中,下面三个类的实现比较重要:
* InputMethodService类：提供了很多的输入法的基础实现，管理状态的条款、输入法的可见度、与当前可见的Activity沟通;
* CandidateView类：提供候选字选择视图,直接继承于View。负责显示软键盘上面的那个候选区域。
* LatinKeyboard类：软键盘类,直接继承与Keyboard类。负责解析并保存键盘布局，并提供选词算法，供程序运行当中使用。其中键盘布局是以XML文件存放在资源当中的。
* Android的输入法服务，通过一些接口函数与系统进行交互，在这些接口中，主要接口是InputMethodService。它提供了一个输入法的标准实现，开发输入法时可以参照和自定义该实现，表1显示了InputMethodService接口提供的函数。

|  |  |
| --- | --- |
| 主要接口函数 | 说明 |
| onlnitializelnterface | 进行UI的初始化,创建和修改时调用此接口 |
| onBindlnput | 在另外的客户端和该输入法连接时调用 |
| onStartlnput | 初始化整个输入法,非常重要的一个回调,它在  编辑框中用户已经开始输入的时候调用 |
| onCreatelnputView | 创建输入视图,返回一个层次性的输入视图,而且只是在这个视图第一次显示的时候被调用 |
| onCreateCandidatesView | 创建候选框视图,当需要候选框显示时调用 |
| onCreateExtractTextView | 全屏模式下的视图 |
| onStartlnput  View | 在输入视图被显示并且在一个新的输入框中输入已经开始的时候调用 |
| getCurrentlnputConnection | 此函数呼叫应用程序,接收原始的按键事件 |
| onFinishlnput | 结束输入法 |

* 表1.InputMethodService接口提供的函数

* **六．输入法服务流程**
* **6.1客户端应用注册到输入法系统服务**
* **6.1.1创建LocationManger**
* 1）客户端应用创建时，调用ViewRoot(…)；
* 2）调用ViewRoot .getWindowSession(…)；
* 3）通过调用InputMethodManager.getInstance()创建LocationManager对象，一个客户端应用只会创建一个LocationManager对象；
* 4）LocationManager对象创建时，会创建一个IInputMethodClient对象，同时创建一个IInputContext对象；
* **6.1.2注册到输入法系统服务**
* 1）调用WindowMangerService. openSession(…)；
* 2）调用WindowMangerService. Session(…)；
* 3）调用InputMethodMangerService.addClient(IInputMethodClientclient, IInputContext inputContext, int uid, int pid)，将InputMethodManager中创建的IInputMethodClient对象以及InputMethodManager中创建的IInputContext对象传入进去，其中uid为客户端应用用户ID，pid为客户端应用进程ID；
* 4）加入到InputMethodManagerService维护的一个列表HashMap<IBinder, ClientState>中，其中IBinder对应IInputMethodClient，一个InputMethodManager只有一个IInputMethodCliend。
* **6.2客户端应用调用输入法**
* 1）TextView.setText();
* 2）调用InputMethodManager.restartInput();
* 3）调用InputMethodManager. startInputInner();
* 4）  调用InputMethodManagerService. startInput(IInputMethodClient,IInputContext…);
* **6.3输入法系统服务调用输入法**
* 输入法系统服务调用输入法包括输入法系统服务启动输入法，输入法创建，输入法系统服务建立与输入法的会话，输入法系统服务启动输入法以及输入法服务显示输入法。

* **七．Android手机输入法设计的主要内容**
* 一款手机输入法能否得到用户的认可，主要取决于输入法的易用性和高效性以及界面的美观度。因此，IME程序的设计主要是用户交互接口的设计和编码转换引擎的设计。
* 首先，对于软键盘输入法来说，软键盘的布局设计在很大程度上影响着用户的使用，同时，由于手机设备的屏幕大小限制，如何处理输入法中软键盘和候选框、组合框、状态窗口的相对位置和大小也是输入法界面设计的重要部分。
* 其次，由于智能终端设备的处理器和内存的限制，如何设计一个系统内存占用小，CPU使用率低的编码转换引擎和词库存储方案也是手机输入法设计的重要组成部分。
* 具体来说，IMF框架以及要编写 Android 输入法需要实现的模块如图4所示。
* 
* 图 4.Android 输入法模块

## [android 輸入法程式碼架構](http://fecbob.pixnet.net/blog/post/35579539)

facebook PLURK twitter

用戶端應用是一個包含有圖形介面的應用，如位址本。圖形介面上包含有能夠接收輸入的編輯方塊，如TextView。  
輸入法模組提供軟鍵盤，將使用者在軟鍵盤上的按鍵輸入根據某種演算法（如Zi, T9, 國筆等）轉換成單詞，然後傳遞給用戶端應用。目錄development/samples/SoftKeyboard下提供了一個輸入法模組實例。如果想 要實現一個中文輸入法，可參考這個實例。  
平臺部分實現一些管理功能，負責裝載某個輸入法模組，啟動，終止該模組等。  
相關代碼主要位於下麵幾個位置。其中，位於3,5,6,7目錄下的代碼最值得關注。  
1. frameworks/base/core/java/com/android/internal/view  
這個目錄下定義了幾個重要的idl 介面。  
IInputMethod.aidl 定義了IInputMethod idl 介面,用於用戶端跨進程操作InputMethod介面。  
IInputMethodSession.aidl 定義了IInputMethodSession介面，是IInputMethod的輔助介面。用於用戶端跨進程操作InputMethodSession介面。

IInputMethodCallback.aidl定義了一個helper 介面，由用戶端實現。IInputMethod.aidl和IInputMethodSession.aidl實例可以分別調用該介面中的不同方法

IInputMethodManager.aidl 定義了Input Method Manager的service介面。用戶端通過InputMethodManager interface調用這個service。  
InputMethodManagerServic  
e.java實現了IInputMethodManager.aidl介面

IInputMethodClient.aidl定義介面，標識一個Input Method Manager 的客戶。這個service在用戶端實現，提供給server端調用。

IInputContext.aidl定義了一個介面，由用戶端提供InputMethod使用。InputMethod可以與用戶端交互，調用用戶端提供的callback。  
IInputConnectionWrapper.java 實現了IInputContext介面。  
IInputContextCallback.aidl定義了一個介面，定義了一組callback函數給IInputContext.aidl實例調用，從用戶端返回資訊給InputMethod。  
InputConnectionWrapper.java實現了IInputContextCallback介面。

2. frameworks/base/services/java/com/android/server  
InputMethodManagerService.java實現了IInputMethodManager.aidl介面

3. frameworks/base/core/java/android/view/inputmethod  
這個目錄下定義了幾個重要的interface和類。  
InputMethodManager.java實現了InputMethodManager 類。此類調用IInputMethodManager.aidl介面功能，而IInputMethodManager.aidl介面功能由 InputMethodManagerService.java實現，並運行在不同于用戶端進程的server進程中。

InputConnection.java定義了InputConnection interface。InputConnection 介面在輸入法和用戶端之間建立了一個連接，輸入法可以使用該連接獲取或發送資訊給用戶端。InputConnection實例由用戶端創建之後傳遞給輸入 法使用。BaseInputConnection.java 實現了InputConnection介面的一個基類: BaseInputConnection。 EditableInputConnection.java實現了一個派生類

InputBinding.java 定義了類InputBinding，這個類實現了parcelable 介面。這個類的成員變數包含了用戶端傳向server的資訊。

InputMethod.java定義了InputMethod interface。檔InputMethodService.java中類InputMethodImpl實現了這個介面。這個介面定義了一套操縱一 個輸入法的方法。如，createSession，startInput等。要編寫一個具體輸入法的話，就需要派生這個介面。

InputMethodSession.java定義了InputMethodSession介面。檔InputMethodService.java 中類InputMethodSessionImpl實現了這個介面。InputMethodSession是InputMethod的輔助介面，用於具體 和某個輸入法用戶端交互。

CompletionInfo.java 類描述一個text completion.  
EditorInfo.java類描述一個接收輸入的view的屬性，如內容屬性(text, digit, etc)。  
ExtractedText.java類描述從view中提取的傳遞給輸入法的文字屬性。

4. frameworks/base/core/java/com/android/internal/widget  
EditableInputConnection.java實現了BaseInputConnection的一個派生類。

5. frameworks/base/core/java/android/inputmethodservice  
這個目錄下的代碼提供了實現一個具體輸入法的框架類。從這些類派生，就可以定制一個輸入法。  
SoftInputWindow.java中的SoftInputWindow類是一個Dialog子類。它代表一個輸入法的頂級視窗（由視窗管理器管理），這個視窗由上到下，包含extractArea, candidatesArea, 和 inputArea。

Keyboard.java 中的Keyboard類裝載並解析一個描述虛擬鍵盤（Soft Keyboard)的xml檔(如development/samples/SoftKeyboard/res/xml)，並存儲該鍵盤的屬性，如該虛 擬鍵盤包含多上行，每行有哪些鍵等。  
KeyboardView.java 中的KeyboardView類是一個View子類。它根據Keyboard資料結構真正的在screen上畫出一個虛擬鍵盤。這個虛擬鍵盤就是SoftInputWindow中的inputArea。

AbstractInputMethodService是Service的派生類，並實現了KeyEvent.Callback 介面。實現了InputMethod 和 InputMethodSession的基類。dispatchKeyEvent 函數將收到的key event傳給相應的key 處理函數（在派生類中實現）。當這個service被用戶端綁定時，其onBind()函數給用戶端返回了一個IInputMethodWrapper實 例，這個實例實現了IInputMethod idl介面。用戶端可以使用該介面的相關功能。

IInputMethodWrapper.java 實現了IInputMethod idl 介面。這個類收到用戶端的跨進程命令後，調用InputMethod完成相應功能。  
IInputMethodSessionWrapper.java 實現了IInputMethodSession idl介面。這個類收到用戶端的跨進程命令後，調用InputMethodSession完成相應功能。

6. frameworks/base/core/res/res/layout  
這個目錄下存放著一些系統資源。其中，  
input\_method.xml描述了一個輸入法的視窗（即SoftInputWindow)佈局，從上往下，依次排列extractArea, candidatesArea 和 inputArea。  
input\_method\_extract\_view.xml。

7. development/samples/SoftKeyboard  
這個目錄下代碼實現了一個的輸入法實例－－軟鍵盤英文／數位輸入法。這裡面實現的類大都是從frameworks/base/core/java/android/inputmethodservice 中的類派生而來。  
AndroidManifest.xml：描述這個.apk提供的service以及關於這個輸入法的一些資訊。  
res/xml/目錄下存儲著幾個描述不同虛擬鍵盤的xml檔。  
LatinKeyboard.java中的LatinKeyboard類是Keyboard的子類。  
LatinKeyboardView.java中的LatinKeyboardView類是KeyboardView的子類。

8. frameworks/base/core/java/android/widget  
在這裡TextView.java是使用Input Method Framework (IMF)的用戶端。TextView創建了一個InputMethodManager的實例並調用其restartInput 函數。  
InputMethodManager::restartInput函數創建了一個InputConnection 實例並調用IInputMethodManager::startInput。  
IInputMethodManager::startInput 函數使用mContext.bindService啟動一個InputMethod service, 如 Sample Soft Keyboard。

9. frameworks/base/core/java/com/android/internal/widget