我们知道，在Android系统中，Activity是以堆栈的形式组织在ActivityManagerService服务中的。与Activity类似，Android系统中的窗口也是以堆栈的形式组织在WindowManagerService服务中的，其中，Z轴位置较低的窗口位于Z轴位置较高的窗口的下面。在本文中，我们就详细分析WindowManagerService服务是如何以堆栈的形式来组织窗口的。

       从前面[Android应用程序启动过程源代码分析](http://shyluo.blog.51cto.com/5725845/965994)一文可以知道，应用程序进程中的每一个Activity组件在Activity管理服务ActivityManagerService中都对应有一个ActivityRecord对象。从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文又可以知道，Activity管理服务ActivityManagerService中每一个ActivityRecord对象在Window管理服务WindowManagerService中都对应有一个AppWindowToken对象。

       此外，在输入法管理服务InputMethodManagerService中，每一个输入法窗口都对应有一个Binder对象，这个Binder对象在Window管理服务WindowManagerService又对应有一个WindowToken对象。

       与输入法窗口类似，在壁纸管理服务WallpaperManagerService中，每一个壁纸窗口都对应有一个Binder对象，这个Binder对象在Window管理服务WindowManagerService也对应有一个WindowToken对象。

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

       上述的窗口组织方式如图1所示：



图1 窗口在WindowManagerService服务中的组织方式

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

       图1中的对象的对应关系如下所示：

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

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

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

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

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

      从图1还可以知道，Window Stack中的WindowState是按照它们所描述的窗口的Z轴位置从低到高排列的。从上面的分析可以看出对于App，使用的是AppWindowToken，对于输入法和壁纸使用的WindowToken

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

      1.  增加AppWindowToken

      从前面[Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析](http://blog.csdn.net/luoshengyang/article/details/8275938)一文可以知道，一个Activity组件在启动的过程中，ActivityManagerService服务会调用调用WindowManagerService类的成员函数addAppToken来为它增加一个AppWindowToken，如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        /\*\*       \* Mapping from a token IBinder to a WindowToken object.       \*/      final HashMap<IBinder, WindowToken> **mTokenMap** =              new HashMap<IBinder, WindowToken>();        /\*\*       \* The same tokens as mTokenMap, stored in a list for efficient iteration       \* over them.       \*/      final ArrayList<WindowToken> **mTokenList** = new ArrayList<WindowToken>();      ......        /\*\*       \* Z-ordered (bottom-most first) list of all application tokens, for       \* controlling the ordering of windows in different applications.  This       \* contains WindowToken objects.       \*/      final ArrayList<AppWindowToken> **mAppTokens** = new ArrayList<AppWindowToken>();      ......        public void addAppToken(int addPos, IApplicationToken token,              int groupId, int requestedOrientation, boolean fullscreen) {          ......            synchronized(mWindowMap) {              AppWindowToken wtoken = findAppWindowToken(token.asBinder());              if (wtoken != null) {                  ......                  return;              }              wtoken = new AppWindowToken(token);              ......              mAppTokens.add(addPos, wtoken);              ......              mTokenMap.put(token.asBinder(), wtoken);              mTokenList.add(wtoken);                ......          }      }        ......  } |

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

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

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

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

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

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

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

      2. 删除AppWindowToken

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        private void removeAppTokensLocked(List<IBinder> tokens) {          // XXX This should be done more efficiently!          // (take advantage of the fact that both lists should be          // ordered in the same way.)          int N = tokens.size();          for (int i=0; i<N; i++) {              IBinder token = tokens.get(i);              final AppWindowToken wtoken = findAppWindowToken(token);              if (!mAppTokens.remove(wtoken)) {                  ......                  i--;                  N--;              }          }      }        ......  } |

       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removeAppTokensLocked可以同时删除一组AppWindowToken对象。

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

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

       3. 移动AppWindowToken至指定位置

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        public void moveAppToken(int index, IBinder token) {          if (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,                  "moveAppToken()")) {              throw new SecurityException("Requires MANAGE\_APP\_TOKENS permission");          }            synchronized(mWindowMap) {              ......              final AppWindowToken wtoken = findAppWindowToken(token);              if (wtoken == null || !mAppTokens.remove(wtoken)) {                  ......                  return;              }              mAppTokens.add(index, wtoken);              ......                final long origId = Binder.clearCallingIdentity();              ......              if (tmpRemoveAppWindowsLocked(wtoken)) {                  ......                  reAddAppWindowsLocked(findWindowOffsetLocked(index), wtoken);                  ......                  updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES);                  mLayoutNeeded = true;                  performLayoutAndPlaceSurfacesLocked();              }              Binder.restoreCallingIdentity(origId);          }      }        ......  } |

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

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

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

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

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        /\*\*       \* Z-ordered (bottom-most first) list of all Window objects.       \*/      final ArrayList<WindowState> **mWindows** = new ArrayList<WindowState>();      ......        private int findWindowOffsetLocked(int tokenPos) {          final int NW = mWindows.size();            if (tokenPos >= mAppTokens.size()) {              int i = NW;              while (i > 0) {                  i--;                  WindowState win = mWindows.get(i);                  if (win.getAppToken() != null) {                      return i+1;                  }              }          }            while (tokenPos > 0) {              // Find the first app token below the new position that has              // a window displayed.              final AppWindowToken wtoken = mAppTokens.get(tokenPos-1);              ......              if (wtoken.sendingToBottom) {                  ......                  tokenPos--;                  continue;              }              int i = wtoken.windows.size();              while (i > 0) {                  i--;                  WindowState win = wtoken.windows.get(i);                  int j = win.mChildWindows.size();                  while (j > 0) {                      j--;                      WindowState cwin = win.mChildWindows.get(j);                      if (cwin.mSubLayer >= 0) {                          for (int pos=NW-1; pos>=0; pos--) {                              if (mWindows.get(pos) == cwin) {                                  ......                                  return pos+1;                              }                          }                      }                  }                  for (int pos=NW-1; pos>=0; pos--) {                      if (mWindows.get(pos) == win) {                          ......                          return pos+1;                      }                  }              }              tokenPos--;          }            return 0;      }        ......  } |

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

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

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

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

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

       A. 它在mAppTokens列表中的位置小于tokenPos；

       B. 它在WindowState堆栈中对应有WindowState对象；

       C. 它不是将要置于WindowState堆栈的底部。

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

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

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

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

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

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

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

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

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

       4. 移动AppWindowToken至顶端

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        public void moveAppTokensToTop(List<IBinder> tokens) {          if (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,                  "moveAppTokensToTop()")) {              throw new SecurityException("Requires MANAGE\_APP\_TOKENS permission");          }          final long origId = Binder.clearCallingIdentity();          synchronized(mWindowMap) {              removeAppTokensLocked(tokens);              final int N = tokens.size();              for (int i=0; i<N; i++) {                  AppWindowToken wt = findAppWindowToken(tokens.get(i));                  if (wt != null) {                      mAppTokens.add(wt);                      if (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {                          mToTopApps.remove(wt);                          mToBottomApps.remove(wt);                          mToTopApps.add(wt);                          wt.sendingToBottom = false;                          wt.sendingToTop = true;                      }                  }              }              if (mNextAppTransition == WindowManagerPolicy.TRANSIT\_UNSET) {                  moveAppWindowsLocked(tokens, mAppTokens.size());              }          }          Binder.restoreCallingIdentity(origId);      }        ......  } |

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

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

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

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

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

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

      5. 移动AppWindowToken至底端

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        public void moveAppTokensToBottom(List<IBinder> tokens) {          if (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,                  "moveAppTokensToBottom()")) {              throw new SecurityException("Requires MANAGE\_APP\_TOKENS permission");          }          final long origId = Binder.clearCallingIdentity();          synchronized(mWindowMap) {              removeAppTokensLocked(tokens);              final int N = tokens.size();              int pos = 0;              for (int i=0; i<N; i++) {                  AppWindowToken wt = findAppWindowToken(tokens.get(i));                  if (wt != null) {                      mAppTokens.add(pos, wt);                      if (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {                          mToTopApps.remove(wt);                          mToBottomApps.remove(wt);                          mToBottomApps.add(i, wt);                          wt.sendingToTop = false;                          wt.sendingToBottom = true;                      }                      pos++;                  }              }              if (mNextAppTransition == WindowManagerPolicy.TRANSIT\_UNSET) {                  moveAppWindowsLocked(tokens, 0);              }          }          Binder.restoreCallingIdentity(origId);      }        ......  } |

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

       6. 增加WindowToken

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        public void addWindowToken(IBinder token, int type) {          if (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,                  "addWindowToken()")) {              throw new SecurityException("Requires MANAGE\_APP\_TOKENS permission");          }          synchronized(mWindowMap) {              WindowToken wtoken = mTokenMap.get(token);              if (wtoken != null) {                  Slog.w(TAG, "Attempted to add existing input method token: " + token);                  return;              }              wtoken = new WindowToken(token, type, true);              mTokenMap.put(token, wtoken);              mTokenList.add(wtoken);              if (type == TYPE\_WALLPAPER) {                  mWallpaperTokens.add(wtoken);              }          }      }        ......  } |

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

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

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

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

       这里有两个地方需要注意：

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

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        public int addWindow(Session session, IWindow client,              WindowManager.LayoutParams attrs, int viewVisibility,              Rect outContentInsets, InputChannel outInputChannel) {          ......            synchronized(mWindowMap) {              ......                boolean addToken = false;              WindowToken token = mTokenMap.get(attrs.token);              if (token == null) {                  if (attrs.type >= FIRST\_APPLICATION\_WINDOW                          && attrs.type <= LAST\_APPLICATION\_WINDOW) {                      ......                      return WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;                  }                  if (attrs.type == TYPE\_INPUT\_METHOD) {                      ......                      return WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;                  }                  if (attrs.type == TYPE\_WALLPAPER) {                      ......                      return WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;                  }                  token = new WindowToken(attrs.token, -1, false);                  addToken = true;              }                ......                if (addToken) {                  mTokenMap.put(attrs.token, token);                  mTokenList.add(token);              }                ......          }            ......      }        ......  } |

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

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

       7. 删除WindowToken

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        public void removeWindowToken(IBinder token) {          if (!checkCallingPermission(android.Manifest.permission.MANAGE\_APP\_TOKENS,                  "removeWindowToken()")) {              throw new SecurityException("Requires MANAGE\_APP\_TOKENS permission");          }          final long origId = Binder.clearCallingIdentity();          synchronized(mWindowMap) {              WindowToken wtoken = mTokenMap.remove(token);              mTokenList.remove(wtoken);              if (wtoken != null) {                  boolean delayed = false;                  if (!wtoken.hidden) {                      wtoken.hidden = true;                      final int N = wtoken.windows.size();                      boolean changed = false;                      for (int i=0; i<N; i++) {                          WindowState win = wtoken.windows.get(i);                          if (win.isAnimating()) {                              delayed = true;                          }                          if (win.isVisibleNow()) {                              applyAnimationLocked(win,                                      WindowManagerPolicy.TRANSIT\_EXIT, false);                              changed = true;                          }                      }                      if (changed) {                          mLayoutNeeded = true;                          performLayoutAndPlaceSurfacesLocked();                          updateFocusedWindowLocked(UPDATE\_FOCUS\_NORMAL);                      }                      if (delayed) {                          mExitingTokens.add(wtoken);                      } else if (wtoken.windowType == TYPE\_WALLPAPER) {                          mWallpaperTokens.remove(wtoken);                      }                  }                  ......              } else {                  Slog.w(TAG, "Attempted to remove non-existing token: " + token);              }          }          Binder.restoreCallingIdentity(origId);      }        ......  } |

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

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

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

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

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

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

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

       8. 增加WindowState

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......      /\*\*       \* Mapping from an IWindow IBinder to the server's Window object.       \* This is also used as the lock for all of our state.       \*/      final HashMap<IBinder, WindowState> mWindowMap = new HashMap<IBinder, WindowState>();      ......      /\*\*       \* Z-ordered (bottom-most first) list of all Window objects.       \*/      final ArrayList<WindowState> mWindows = new ArrayList<WindowState>();      ......        public int addWindow(Session session, IWindow client,              WindowManager.LayoutParams attrs, int viewVisibility,              Rect outContentInsets, InputChannel outInputChannel) {          ......            WindowState win = null;            synchronized(mWindowMap) {              ......              win = new WindowState(session, client, token,                      attachedWindow, attrs, viewVisibility);              ......                mWindowMap.put(client.asBinder(), win);              ......              if (attrs.type == TYPE\_INPUT\_METHOD) {                  mInputMethodWindow = win;                  addInputMethodWindowToListLocked(win);                  ......              } else if (attrs.type == TYPE\_INPUT\_METHOD\_DIALOG) {                  mInputMethodDialogs.add(win);                  addWindowToListInOrderLocked(win, true);                  adjustInputMethodDialogsLocked();                  ......              } else {                  addWindowToListInOrderLocked(win, true);                  if (attrs.type == TYPE\_WALLPAPER) {                      .......                      adjustWallpaperWindowsLocked();                  } else if ((attrs.flags&FLAG\_SHOW\_WALLPAPER) != 0) {                      adjustWallpaperWindowsLocked();                  }              }              ......          }          ......      }      ......  } |

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

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

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

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

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

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

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

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

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

       9. 增加WindowState到窗口堆栈

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......      private void addWindowToListInOrderLocked(WindowState win, boolean addToToken) {          final IWindow client = win.mClient;          final WindowToken token = win.mToken;          final ArrayList<WindowState> localmWindows = mWindows;          final int N = localmWindows.size();          final WindowState attached = win.mAttachedWindow;          int i;          if (attached == null) {              //CASE 1：要增加的窗口win没有附加在其它窗口上              int tokenWindowsPos = token.windows.size();              if (token.appWindowToken != null) {                  //CASE 1.1：要增加的窗口win是一个Activity窗口                  int index = tokenWindowsPos-1;                  if (index >= 0) {                      //CASE 1.1.1：用来要增加的窗口win的令牌token已存在其它窗口                      ......                  } else {                      //CASE 1.1.2：用来要增加的窗口win的令牌token尚未存在任何窗口                      ......                  }              } else {                  //CASE 1.2：要增加的窗口win不是一个Activity窗口                  ......              }              if (addToToken) {                  token.windows.add(tokenWindowsPos, win);              }          } else {              //CASE 2：要增加的窗口win附加在窗口attached上              ......          }          if (win.mAppToken != null && addToToken) {              win.mAppToken.allAppWindows.add(win);          }      }      ......  } |

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

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

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

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

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

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

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

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

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

**CASE 1**：要增加的窗口win没有附加在其它窗口上

**----CASE 1.1**：要增加的窗口win是一个Activity窗口

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

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

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

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

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

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

      接下来，我们就分别分析这四种情况是如何将窗口win增加窗口堆栈中去的。

**CASE 1.1.1**对应的代码为：

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

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

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

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

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

**CASE 1.1.2**对应的代码为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74 | // Figure out where the window should go, based on the  // order of applications.  final int NA = mAppTokens.size();  WindowState pos = null;  for (i=NA-1; i>=0; i--) {      AppWindowToken t = mAppTokens.get(i);      if (t == token) {          i--;          break;      }      // We haven't reached the token yet; if this token      // is not going to the bottom and has windows, we can      // use it as an anchor for when we do reach the token.      if (!t.sendingToBottom && t.windows.size() > 0) {          pos = t.windows.get(0);      }  }  // We now know the index into the apps.  If we found  // an app window above, that gives us the position; else  // we need to look some more.  if (pos != null) {      // Move behind any windows attached to this one.      WindowToken atoken = mTokenMap.get(pos.mClient.asBinder());      if (atoken != null) {          final int NC = atoken.windows.size();          if (NC > 0) {              WindowState bottom = atoken.windows.get(0);              if (bottom.mSubLayer < 0) {                  pos = bottom;              }          }      }      placeWindowBefore(pos, win);  } else {      // Continue looking down until we find the first      // token that has windows.      while (i >= 0) {          AppWindowToken t = mAppTokens.get(i);          final int NW = t.windows.size();          if (NW > 0) {              pos = t.windows.get(NW-1);              break;          }          i--;      }      if (pos != null) {          // Move in front of any windows attached to this          // one.          WindowToken atoken = mTokenMap.get(pos.mClient.asBinder());          if (atoken != null) {              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);          placeWindowAfter(pos, win);      } else {          // Just search for the start of this layer.          final int myLayer = win.mBaseLayer;          for (i=0; i<N; i++) {              WindowState w = localmWindows.get(i);              if (w.mBaseLayer > myLayer) {                  break;              }          }          ......          localmWindows.add(i, win);          mWindowsChanged = true;      }  } |

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

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



图2 窗口win位于窗口C的下面

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

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

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

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



图3 窗口win位于窗口D的上面

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

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

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

**CASE 1.2**对应的代码为：

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

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

**CASE 2**对应的代码为：

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

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

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



图4 窗口win插入到窗口B的下面



图5 窗口win插入在窗口attached的下面

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

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

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

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



图6 窗口win插入在窗口B的下面

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

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



图7 窗口win插入在窗口attached的下面

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

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



图8 窗口win插入在窗口B的上面



图9 窗口win插入在窗口attached的上面

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

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

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

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

        10. 删除WindowState

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        private int tmpRemoveWindowLocked(int interestingPos, WindowState win) {          int wpos = mWindows.indexOf(win);          if (wpos >= 0) {              if (wpos < interestingPos) interestingPos--;              ......              mWindows.remove(wpos);              mWindowsChanged = true;              int NC = win.mChildWindows.size();              while (NC > 0) {                  NC--;                  WindowState cw = win.mChildWindows.get(NC);                  int cpos = mWindows.indexOf(cw);                  if (cpos >= 0) {                      if (cpos < interestingPos) interestingPos--;                      ......                      mWindows.remove(cpos);                  }              }          }          return interestingPos;      }        ......  } |

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

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

      11. 在指定位置增加WindowState

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......      private final int reAddWindowLocked(int index, WindowState win) {          final int NCW = win.mChildWindows.size();          boolean added = false;          for (int j=0; j<NCW; j++) {              WindowState cwin = win.mChildWindows.get(j);              if (!added && cwin.mSubLayer >= 0) {                  ......                  mWindows.add(index, win);                  index++;                  added = true;              }              ......              mWindows.add(index, cwin);              index++;          }          if (!added) {              ......              mWindows.add(index, win);              index++;          }          mWindowsChanged = true;          return index;      }      ......  } |

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

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

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

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

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

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......      private void reAddWindowToListInOrderLocked(WindowState win) {          addWindowToListInOrderLocked(win, false);          // This is a hack to get all of the child windows added as well          // at the right position.  Child windows should be rare and          // this case should be rare, so it shouldn't be that big a deal.          int wpos = mWindows.indexOf(win);          if (wpos >= 0) {              ......              mWindows.remove(wpos);              mWindowsChanged = true;              reAddWindowLocked(wpos, win);          }      }      ......  } |

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

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

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......      private final int reAddAppWindowsLocked(int index, WindowToken token) {          final int NW = token.windows.size();          for (int i=0; i<NW; i++) {              index = reAddWindowLocked(index, token.windows.get(i));          }          return index;      }      ......  } |

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

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

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

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

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......      private void moveAppWindowsLocked(AppWindowToken wtoken, int tokenPos,              boolean updateFocusAndLayout) {          // First remove all of the windows from the list.          tmpRemoveAppWindowsLocked(wtoken);          // Where to start adding?          int pos = findWindowOffsetLocked(tokenPos);          // And now add them back at the correct place.          pos = reAddAppWindowsLocked(pos, wtoken);          if (updateFocusAndLayout) {              if (!updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES)) {                  assignLayersLocked();              }              mLayoutNeeded = true;              performLayoutAndPlaceSurfacesLocked();          }      }      ......  } |

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

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

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

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

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......      private void moveAppWindowsLocked(List<IBinder> tokens, int tokenPos) {          // First remove all of the windows from the list.          final int N = tokens.size();          int i;          for (i=0; i<N; i++) {              WindowToken token = mTokenMap.get(tokens.get(i));              if (token != null) {                  tmpRemoveAppWindowsLocked(token);              }          }          // Where to start adding?          int pos = findWindowOffsetLocked(tokenPos);          // And now add them back at the correct place.          for (i=0; i<N; i++) {              WindowToken token = mTokenMap.get(tokens.get(i));              if (token != null) {                  pos = reAddAppWindowsLocked(pos, token);              }          }          if (!updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_PLACE\_SURFACES)) {              assignLayersLocked();          }          mLayoutNeeded = true;          performLayoutAndPlaceSurfacesLocked();          //dump();      }      ......  } |

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

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

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

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

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

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

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

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

       最后，我们可以将WindowManagerService服务中的AppWindowToken理解成一个Activity组件令牌，而将它所对应的WindowState对象理解成一个Activity窗口。有了这些概念之后，就为学习WindowManagerService服务的各种实现打下坚实的基础