在前两文中，我们分析了Activity组件的窗口对象和视图对象的创建过程。Activity组件在其窗口对象和视图对象创建完成之后，就会请求与WindowManagerService建立一个连接，即请求WindowManagerService为其增加一个WindowState对象，用来描述它的窗口状态。在本文中，我们就详细分析Activity组件与WindowManagerService的连接过程。

        我们从两方面来看Activity组件与WindowManagerService服务之间的连接。一方面是从Activity组件到WindowManagerService服务的连接，另一方面是从WindowManagerService服务到Activity组件的连接。

1. 从Activity组件到WindowManagerService服务的连接是以Activity组件所在的应用程序进程为单位来进行的。当一个应用程序进程在启动第一个Activity组件的时候，它便会打开一个到WindowManagerService服务的连接，这个连接以应用程序进程从WindowManagerService服务处获得一个实现了IWindowSession接口的Session代理对象来标志。
2. 从WindowManagerService服务到Activity组件的连接是以Activity组件为单位来进行的。在应用程序进程这一侧，每一个Activity组件都关联一个实现了IWindow接口的W对象，这个W对象在Activity组件的视图对象创建完成之后，就会通过前面所获得一个Session代理对象来传递给WindowManagerService服务，而WindowManagerService服务接收到这个W对象之后，就会在内部创建一个WindowState对象来描述与该W对象所关联的Activity组件的窗口状态，并且以后就通过这个W对象来控制对应的Activity组件的窗口状态。

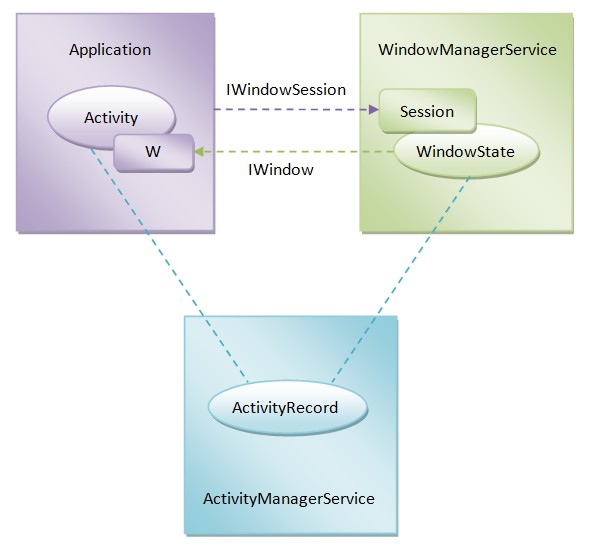
         上述Activity组件与WindowManagerService服务之间的连接模型如图1所示：

图1 Activity组件与WindowManagerService服务之间的连接模型

        从图1还可以看出，每一个Activity组件在ActivityManagerService服务内部，都对应有一个ActivityRecord对象，这个ActivityRecord对象是Activity组件启动的过程中创建的，用来描述Activity组件的运行状态，这一点可以参考前面[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)一文。这样，每一个Activity组件在应用程序进程、WindowManagerService服务和ActivityManagerService服务三者之间就分别一一地建立了连接。在本文中，我们主要关注Activity组件在应用程序进程和WindowManagerService服务之间以及在WindowManagerService服务和ActivityManagerService服务之间的连接。

        接下来我们就通过Session类、W类和WindowState类的实现来简要描述Activity组件与WindowManagerService服务之间的连接，如图2和图3所示：

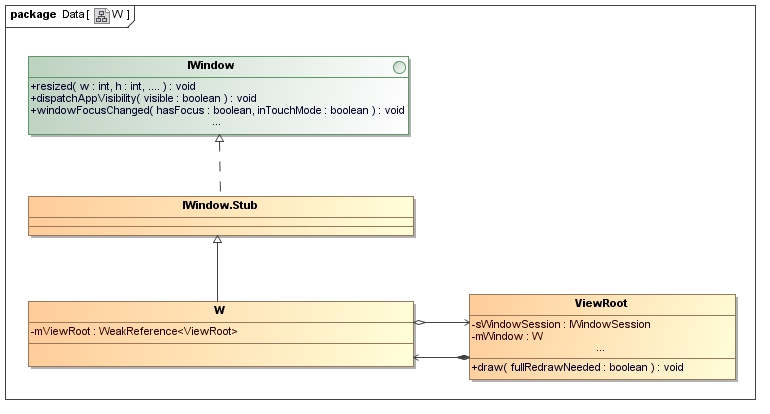


图2 W类的实现

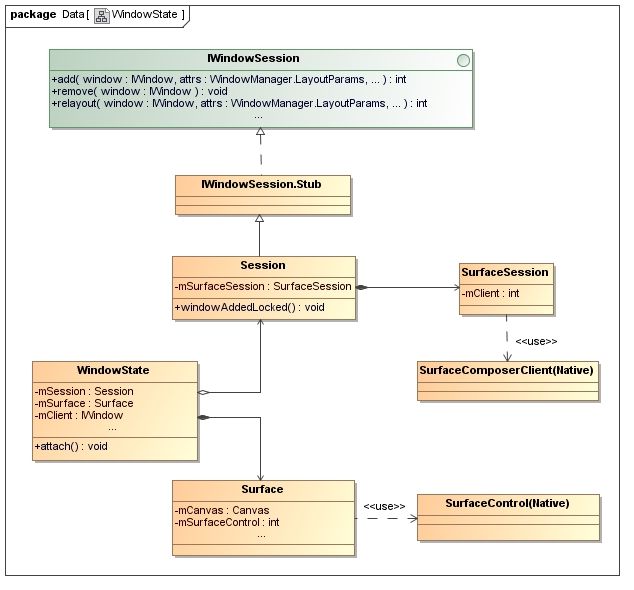


图3 Session类和WindowState类的实现

        W类实现了IWindow接口，它的类实例是一个Binder本地对象。从前面[Android应用程序窗口（Activity）的视图对象（View）的创建过程分析](http://blog.csdn.net/luoshengyang/article/details/8245546)一文可以知道，一个Activity组件在启动的过程中，会创建一个关联的ViewRoot对象，用来配合WindowManagerService服务来管理该Activity组件的窗口状态。在这个ViewRoot对象内部，有一个类型为W的成员变量mWindow，它是在ViewRoot对象的创建过程中创建的。

        ViewRoot类有一个静态成员变量sWindowSession，它指向了一个实现了IWindowSession接口的Session代理对象。当应用程序进程启动第一个Activity组件的时候，它就会请求WindowManagerService服务发送一个建立连接的Binder进程间通信请求。WindowManagerService服务接收到这个请求之后，就会在内部创建一个类型为Session的Binder本地对象，并且将这个Binder本地对象返回给应用程序进程，后者于是就会得到一个Session代理对象，并且保存在ViewRoot类的静态成员变量sWindowSession中。

        有了这个Session代理对象之后，应用程序进程就可以在启动Activity组件的时候，调用它的成员函数add来将与该Activity组件所关联的一个W对象传递给WindowManagerService服务，后者于是就会得到一个W代理对象，并且会以这个W代理对象来创建一个WindowState对象，即将这个W代理对象保存在新创建的WindowState对象的成员变量mClient中。这个WindowState对象的其余成员变量的描述可以参考前面[Android应用程序窗口（Activity）实现框架简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/8170307)一文的图7，这里不再详述。

        Session类的描述同样可以参考前面[Android应用程序窗口（Activity）实现框架简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/8170307)一文，这里我们主要描述一下它的作用。从图3可以看出，Session类实现了IWindowSession接口，因此，应用程序进程就可以通过保存在ViewRoot类的静态成员变量sWindowSession所描述的一个Session代理对象所实现的IWindowSession接口来与WindowManagerService服务通信，例如：

        1. 在Activity组件的启动过程中，调用这个IWindowSession接口的成员函数add可以将一个关联的W对象传递到WindowManagerService服务，以便WindowManagerService服务可以为该Activity组件创建一个WindowState对象。

        2. 在Activity组件的销毁过程中，调用这个这个IWindowSession接口的成员函数remove来请求WindowManagerService服务之前为该Activity组件所创建的一个WindowState对象，这一点可以参考前面[Android应用程序键盘（Keyboard）消息处理机制分析](http://blog.csdn.net/luoshengyang/article/details/6882903)一文的键盘消息接收通道注销过程分析。

        3. 在Activity组件的运行过程中，调用这个这个IWindowSession接口的成员函数relayout来请求WindowManagerService服务来对该Activity组件的UI进行布局，以便该Activity组件的UI可以正确地显示在屏幕中。

        我们再来看W类的作用。从图2可以看出，W类实现了IWindow接口，因此，WindowManagerService服务就可以通过它在内部所创建的WindowState对象的成员变量mClient来要求运行在应用程序进程这一侧的Activity组件来配合管理窗口的状态，例如：

        1. 当一个Activity组件的窗口的大小发生改变后，WindowManagerService服务就会调用这个IWindow接口的成员函数resized来通知该Activity组件，它的大小发生改变了。

        2. 当一个Activity组件的窗口的可见性之后，WindowManagerService服务就会调用这个IWindow接口的成员函数dispatchAppVisibility来通知该Activity组件，它的可见性发生改变了。

        3. 当一个Activity组件的窗口获得或者失去焦点之后，WindowManagerService服务就会调用这个IWindow接口的成员函数windowFoucusChanged来通知该Activity组件，它的焦点发生改变了。

        理解了Activity组件在应用程序进程和WindowManagerService服务之间的连接模型之后，接下来我们再通过简要分析Activity组件在WindowManagerService服务和ActivityManagerService服务之间的连接。

        Activity组件在WindowManagerService服务和ActivityManagerService服务之间的连接是通过一个AppWindowToken对象来描述的。AppWindowToken类的实现如图4所示：

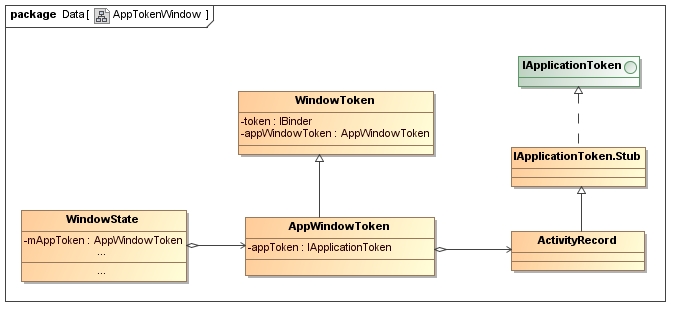


图4 AppWindowToken类的实现

        每一个Activity组件在启动的时候，ActivityManagerService服务都会内部为该Activity组件创建一个ActivityRecord对象，并且会以这个ActivityRecord对象所实现的一个IApplicationToken接口为参数，请求WindowManagerService服务为该Activity组件创建一个AppWindowToken对象，即将这个IApplicationToken接口保存在新创建的AppWindowToken对象的成员变量appToken中。同时，这个ActivityRecord对象还会传递给它所描述的Activity组件所运行在应用程序进程，于是，应用程序进程就可以在启动完成该Activity组件之后，将这个ActivityRecord对象以及一个对应的W对象传递给WindowManagerService服务，后者接着就会做两件事情：

       1. 根据获得的ActivityRecord对象的IApplicationToken接口来找到与之对应的一个AppWindowToken对象；

       2. 根据获得的AppWindowToken对象以及前面传递过来的W代理对象来为正在启动的Activity组件创建一个WindowState对象，并且将该AppWindowToken对象保存在新创建的WindowState对象的成员变量mAppToken中。

       顺便提一下，AppWindowToken类是从WindowToken类继续下来的。WindowToken类也是用来标志一个窗口的，不过这个窗口类型除了是应用程序窗口，即Activity组件窗口之外，还可以是其它的，例如，输入法窗口或者壁纸窗口类型等，而AppWindowToken类只是用来描述Activity组件窗口。当WindowToken类是用来描述Activity组件窗口的时候，它的成员变量token指向的就是用来描述该Activity组件的一个ActivityRecord对象所实现的一个IBinder接口，而成员变量appWindowToken指向的就是其子类AppWindowToken对象。当另一方面，当WindowToken类是用来描述非Activity组件窗口的时候，它的成员变量appWindowToken的值就会等于null。这样，我们就可以通过WindowToken类的成员变量appWindowToken的值来判断一个WindowToken对象是否是用来描述一个Activity组件窗口的，即是否是用来描述一个应用程序窗口的。

       上面所描述的Activity组件在ActivityManagerService服务和WindowManagerService服务之间以及应用程序进程和WindowManagerService服务之间的连接模型比较抽象，接下来，我们再通过三个过程来分析它们彼此之间的连接模型，如下所示：

       1. ActivityManagerService服务请求WindowManagerService服务为一个Activity组件创建一个AppWindowToken对象的过程；

       2. 应用程序进程请求WindowManagerService服务创建一个Session对象的过程；

       3. 应用程序进程请求WindowManagerService服务为一个Activity组件创建一个WindowState对象的过程。

       通过这三个过程的分析，我们就可以对应用程序进程、ActivityManagerService服务和WindowManagerService服务的关系有一个深刻的认识了。

       一. AppWindowToken对象的创建过程

      从前面[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)一文的Step 9可以知道，Activity组件在启动的过程中，会调用到ActivityStack类的成员函数startActivityLocked，该函数会请求WindowManagerService服务为当前正在启动的Activity组件创建一个AppWindowToken对象。接下来，我们就从ActivityStack类的成员函数startActivityLocked开始分析一个AppWindowToken对象的创建过程，如图5所示：

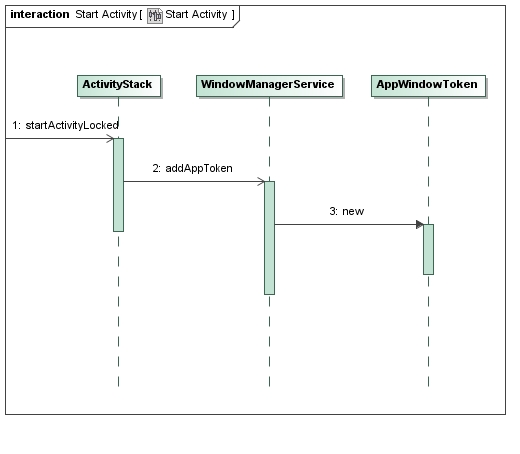


图4 AppWindowToken对象的创建过程

        这个过程可以分为3步，接下来我们就详细分析每一个步骤。

        Step 1. ActivityStack.startActivityLocked

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public class ActivityStack {
2. ......
4. final ActivityManagerService mService;
5. ......
7. final ArrayList mHistory = new ArrayList();
8. ......
10. private final void startActivityLocked(ActivityRecord r, boolean newTask,
11. boolean doResume) {
12. final int NH = mHistory.size();
14. int addPos = -1;
16. if (!newTask) {
17. // If starting in an existing task, find where that is...
18. boolean startIt = true;
19. for (int i = NH-1; i >= 0; i--) {
20. ActivityRecord p = (ActivityRecord)mHistory.get(i);
21. if (p.finishing) {
22. continue;
23. }
24. if (p.task == r.task) {
25. // Here it is!  Now, if this is not yet visible to the
26. // user, then just add it without starting; it will
27. // get started when the user navigates back to it.
28. addPos = i+1;
29. if (!startIt) {
30. mHistory.add(addPos, r);
31. ......
33. mService.mWindowManager.addAppToken(addPos, r, r.task.taskId,
34. r.info.screenOrientation, r.fullscreen);
35. ......
36. return;
37. }
38. break;
39. }
40. if (p.fullscreen) {
41. startIt = false;
42. }
43. }
44. }
46. // Place a new activity at top of stack, so it is next to interact
47. // with the user.
48. if (addPos < 0) {
49. addPos = NH;
50. }
51. ......
53. // Slot the activity into the history stack and proceed
54. mHistory.add(addPos, r);
55. ......
57. if (NH > 0) {
58. ......
60. mService.mWindowManager.addAppToken(
61. addPos, r, r.task.taskId, r.info.screenOrientation, r.fullscreen);
62. .....
63. } else {
64. // If this is the first activity, don't do any fancy animations,
65. // because there is nothing for it to animate on top of.
66. mService.mWindowManager.addAppToken(addPos, r, r.task.taskId,
67. r.info.screenOrientation, r.fullscreen);
68. }
69. ......
71. if (doResume) {
72. resumeTopActivityLocked(null);
73. }
74. }
76. ......
77. }

public class ActivityStack {

......

final ActivityManagerService mService;

......

final ArrayList mHistory = new ArrayList();

......

private final void startActivityLocked(ActivityRecord r, boolean newTask,

boolean doResume) {

final int NH = mHistory.size();

int addPos = -1;

if (!newTask) {

// If starting in an existing task, find where that is...

boolean startIt = true;

for (int i = NH-1; i >= 0; i--) {

ActivityRecord p = (ActivityRecord)mHistory.get(i);

if (p.finishing) {

continue;

}

if (p.task == r.task) {

// Here it is! Now, if this is not yet visible to the

// user, then just add it without starting; it will

// get started when the user navigates back to it.

addPos = i+1;

if (!startIt) {

mHistory.add(addPos, r);

......

mService.mWindowManager.addAppToken(addPos, r, r.task.taskId,

r.info.screenOrientation, r.fullscreen);

......

return;

}

break;

}

if (p.fullscreen) {

startIt = false;

}

}

}

// Place a new activity at top of stack, so it is next to interact

// with the user.

if (addPos < 0) {

addPos = NH;

}

......

// Slot the activity into the history stack and proceed

mHistory.add(addPos, r);

......

if (NH > 0) {

......

mService.mWindowManager.addAppToken(

addPos, r, r.task.taskId, r.info.screenOrientation, r.fullscreen);

.....

} else {

// If this is the first activity, don't do any fancy animations,

// because there is nothing for it to animate on top of.

mService.mWindowManager.addAppToken(addPos, r, r.task.taskId,

r.info.screenOrientation, r.fullscreen);

}

......

if (doResume) {

resumeTopActivityLocked(null);

}

}

......

}

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

        参数r是一个ActivityRecord对象，用来描述的是要启动的Activity组件，参数newTask是一个布尔变量，用来描述要启动的Activity组件是否是在新任务中启动的，参数doResume也是一个布尔变量，用来描述是否需要马上将Activity组件启动起来。

        ActivityStack类的成员变量mService指向的是系统的ActivityManagerService，另外一个成员变量mHistory是一个数组列表，用来描述系统的Activity组件堆栈。

        当参数newTask的值等于false时，就说明参数r所描述的Activity组件是在已有的一个任务中启动的，因此，这时候ActivityStack类的成员函数startActivityLocked就会从上到下遍历保存成员变量mHistory，找到一个已有的Activity组件，它与参数r所描述的Activity组件是属于同一个任务的，即它们的成员变量task的值相等。找到这样的一个Activity组件之后，如果位于它上面的其它Activity组件的窗口至少有一个全屏的，即变量startIt的值等于true，那么ActivityStack类的成员函数startActivityLocked就只是将参数r所描述的Activity组件加入到成员变量mHistory所描述的一个Activity组件堆栈中，以及调用成员变量mService所描述的ActivityManagerService服务的成员变量mWindowManager所描述的WindowManagerService服务的成员函数addAppToken来为它创建一个AppWindowToken对象，然后就返回了，即不会执行下面的代码来启动参数r所描述的Activity组件。这相当于是延迟到参数r所描述的Activity组件可见时，才将它启动起来。

       当参数newTask的值等于true时，就说明参数r所描述的Activity组件是在一个任务中启动的，这时候ActivityStack类的成员函数startActivityLocked就会首先将它添加到成员变量mHistory所描述的一个Activity组件堆栈，接着再判断它是否是系统中第一个启动的Activity组件。如果是系统中第一个启动的Activity组件，那么ActivityStack类的成员函数startActivityLocked就只是简单地调用WindowManagerService服务的成员函数addAppToken来为它创建一个AppWindowToken对象就完事了。如果不是系统系统中第一个启动的Activity组件，那么ActivityStack类的成员函数startActivityLocked除了会调用WindowManagerService服务的成员函数addAppToken来为它创建一个AppWindowToken对象之外，还会为它创建一些启动动画等，我们忽略这些代码。

       从上面的分析就可以看出，无论如何，ActivityStack类的成员函数startActivityLocked都会调用WindowManagerService服务的成员函数addAppToken为正在启动的Activity组件创建一个AppWindowToken对象。创建完成这个AppWindowToken对象之后，如果参数doResume的值等于true，那么ActivityStack类的成员函数startActivityLocked就会继续调用另外一个成员函数resumeTopActivityLocked来继续执行启动参数r所描述的一个Activity组件，这一步可以参考前面[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)一文的Step 10。

      接下来，我们就继续分析WindowManagerService类的成员函数addAppToken的实现，以便可以了解WindowManagerService服务是如何为一个Activity组件创建一个AppWindowToken对象的。

      Step 2. WindowManagerService.addAppToken

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public class WindowManagerService extends IWindowManager.Stub
2. implements Watchdog.Monitor {
3. ......
5. /\*\*
6. \* Mapping from a token IBinder to a WindowToken object.
7. \*/
8. final HashMap<IBinder, WindowToken> mTokenMap =
9. new HashMap<IBinder, WindowToken>();
11. /\*\*
12. \* The same tokens as mTokenMap, stored in a list for efficient iteration
13. \* over them.
14. \*/
15. final ArrayList<WindowToken> mTokenList = new ArrayList<WindowToken>();
16. ......
18. /\*\*
19. \* Z-ordered (bottom-most first) list of all application tokens, for
20. \* controlling the ordering of windows in different applications.  This
21. \* contains WindowToken objects.
22. \*/
23. final ArrayList<AppWindowToken> mAppTokens = new ArrayList<AppWindowToken>();
24. ......
26. public void addAppToken(int addPos, IApplicationToken token,
27. int groupId, int requestedOrientation, boolean fullscreen) {
28. ......
30. long inputDispatchingTimeoutNanos;
31. try {
32. inputDispatchingTimeoutNanos = token.getKeyDispatchingTimeout() \* 1000000L;
33. } catch (RemoteException ex) {
34. ......
35. inputDispatchingTimeoutNanos = DEFAULT\_INPUT\_DISPATCHING\_TIMEOUT\_NANOS;
36. }
38. synchronized(mWindowMap) {
39. AppWindowToken wtoken = findAppWindowToken(token.asBinder());
40. if (wtoken != null) {
41. ......
42. return;
43. }
44. wtoken = new AppWindowToken(token);
45. wtoken.inputDispatchingTimeoutNanos = inputDispatchingTimeoutNanos;
46. wtoken.groupId = groupId;
47. wtoken.appFullscreen = fullscreen;
48. wtoken.requestedOrientation = requestedOrientation;
49. mAppTokens.add(addPos, wtoken);
50. ......
51. mTokenMap.put(token.asBinder(), wtoken);
52. mTokenList.add(wtoken);
54. // Application tokens start out hidden.
55. wtoken.hidden = true;
56. wtoken.hiddenRequested = true;
58. //dump();
59. }
60. }
62. ......
63. }

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) {

......

long inputDispatchingTimeoutNanos;

try {

inputDispatchingTimeoutNanos = token.getKeyDispatchingTimeout() \* 1000000L;

} catch (RemoteException ex) {

......

inputDispatchingTimeoutNanos = DEFAULT\_INPUT\_DISPATCHING\_TIMEOUT\_NANOS;

}

synchronized(mWindowMap) {

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

if (wtoken != null) {

......

return;

}

wtoken = new AppWindowToken(token);

wtoken.inputDispatchingTimeoutNanos = inputDispatchingTimeoutNanos;

wtoken.groupId = groupId;

wtoken.appFullscreen = fullscreen;

wtoken.requestedOrientation = requestedOrientation;

mAppTokens.add(addPos, wtoken);

......

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

mTokenList.add(wtoken);

// Application tokens start out hidden.

wtoken.hidden = true;

wtoken.hiddenRequested = true;

//dump();

}

}

......

}

       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窗口，而后者可以迅速遍历系统中的窗口。

       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组件的窗口的，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类的成员变量mTokenMap、mTokenList和mAppTokens的作用之后，WindowManagerService类的成员函数addAppToken的实现就容易理解了。由于参数token描述的是一个Activity组件窗口，因此，函数就会为它创建一个AppWindowToken对象，并且将这个AppWindowToken对象分别保存在 WindowManagerService类的三个成员变量mTokenMap、mTokenList和mAppTokens中。不过在创建对象，首先会检查与参数token所对应的AppWindowToken对象已经存在。如果已经存在，就什么也不做就返回了。注意，参数addPos用来指定参数token描述的是一个Activity组件在系统Activity组件堆栈中的位置，这个位置同时也指定了为该Activity组件在成员变量成员变量mAppTokens所指向的一个ArrayList中的位置。由于保存在系统Activity组件堆栈的Activity组件本来就是按照它们的Z轴坐标从小到大的顺序来排列的，因此，保存在成员变量mAppTokens所指向的一个ArrayList中的AppWindowToken对象也是按照它们的Z轴坐标从小到大的顺序来排列的。

       函数在为参数token所描述一个Activity组件窗口创建了一个AppWindowToken对象之后，还会初始化它的一系列成员变量，这些成员变量的含义如下所示：

      1. inputDispatchingTimeoutNanos，表示Activity组件窗口收到了一个IO输入事件之后，如果没有在规定的时间内处理完成该事件，那么系统就认为超时。这个超时值可以由Activity组件本身来指定，即可以通过调用一个对应的ActivityRecord对象的成员函数getKeyDispatchingTimeout来获得。假如Activity组件没有指定的话，那么就会使用默认值DEFAULT\_INPUT\_DISPATCHING\_TIMEOUT\_NANOS，即5000 \* 1000000纳秒。

      2. groupId，表示Activity组件所属的任务的ID。从前面[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)一文可以知道，每一个Activity组件都是属于某一个任务的，而每一个任务都用来描述一组相关的Activity组件的，这些Activity组件用来完成用户的某一个操作。

      3. appFullscreen，表示Activity组件的窗口是否是全屏的。如果一个Activity组件的窗口是全屏的，那么它就会将位于它下面的所有窗口都挡住，这样就可以在渲染系统UI时进行优化，即不用渲染位于全屏窗口以下的其它窗口。

      4. requestedOrientation，表示Activity组件所请求的方向。这个方向可以是横的（LANDSCAPE），也可以是竖的（PORTRAIT）。

      5. hidden，表示Activity组件是否是处于不可见状态。

      6. hiddenRequested，与hidden差不多，也是表示Activity组件是否是处于不可见状态。两者的区别在于，在设置目标Activity组件的可见状态时，如果系统等待执行Activity组件切换操作，那么目标Activity组件的可见状态不会马上被设置，即它的hidden值不会马上被设置，而只是设置它的hiddenRequested值，表示它的可见性状态正在等待被设置。等到系统执行完成Activity组件切换操作之后，两者的值就会一致了。

      接下来，我们继续分析一个AppWindowToken对象的创建过程，它AppWindowToken类的构造函数的实现。

      Step 3.new AppWindowToken

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public class WindowManagerService extends IWindowManager.Stub
2. implements Watchdog.Monitor {
3. ......
5. class WindowToken {
6. // The actual token.
7. final IBinder token;
9. // The type of window this token is for, as per WindowManager.LayoutParams.
10. final int windowType;
12. // Set if this token was explicitly added by a client, so should
13. // not be removed when all windows are removed.
14. final boolean explicit;
15. ......
17. // If this is an AppWindowToken, this is non-null.
18. AppWindowToken appWindowToken;
19. ......
21. WindowToken(IBinder \_token, int type, boolean \_explicit) {
22. token = \_token;
23. windowType = type;
24. explicit = \_explicit;
25. }
27. ......
28. }

31. class AppWindowToken extends WindowToken {
32. // Non-null only for application tokens.
33. final IApplicationToken appToken;
34. ......
36. AppWindowToken(IApplicationToken \_token) {
37. super(\_token.asBinder(),
38. WindowManager.LayoutParams.TYPE\_APPLICATION, true);
39. appWindowToken = this;
40. appToken = \_token;
41. }
43. ......
44. }
46. ......
47. }

public class WindowManagerService extends IWindowManager.Stub

implements Watchdog.Monitor {

......

class WindowToken {

// The actual token.

final IBinder token;

// The type of window this token is for, as per WindowManager.LayoutParams.

final int windowType;

// Set if this token was explicitly added by a client, so should

// not be removed when all windows are removed.

final boolean explicit;

......

// If this is an AppWindowToken, this is non-null.

AppWindowToken appWindowToken;

......

WindowToken(IBinder \_token, int type, boolean \_explicit) {

token = \_token;

windowType = type;

explicit = \_explicit;

}

......

}

class AppWindowToken extends WindowToken {

// Non-null only for application tokens.

final IApplicationToken appToken;

......

AppWindowToken(IApplicationToken \_token) {

super(\_token.asBinder(),

WindowManager.LayoutParams.TYPE\_APPLICATION, true);

appWindowToken = this;

appToken = \_token;

}

......

}

......

}

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

       AppWindowToken类的构造函数首先调用父类WindowToken的构造函数来执行父类的初始化工作，然后从父类WindowToken继承下来的成员变量appWindowToken以及自己的成员变量appToken的值。参数\_token指向的是一个ActivityRecord对象的IBinder接口，因此，AppWindowToken类的成员变量appToken描述的就是一个ActivityRecord对象。

       WindowToken类的构造函数用来初始化token、windowType和explicit这三个成员变量。在我们这个场景中，成员变量token指向的也是一个ActivityRecord对象的IBinder接口，用来标志一个Activity组件的窗口，成员变量windowType用来描述窗口的类型，它的值等于WindowManager.LayoutParams.TYPE\_APPLICATION，表示这是一个Activity组件窗口，成员变量explicit用来表示窗口是否是由应用程序进程请求添加的。

       注意，当一个WindowToken对象的成员变量appWindowToken的值不等于null时，就表示它实际描述的是Activity组件窗口，并且这个成员变量指向的就是与该Activity组件所关联的一个AppWindowToken对象。

       至此，我们就分析完成一个AppWindowToken对象的创建过程了，通过这个过程我们就可以知道，每一个Activity组件，在ActivityManagerService服务内部都有一个对应的ActivityRecord对象，并且在WindowManagerService服务内部关联有一个AppWindowToken对象。

       二. Session对象的创建过程

       应用程序进程在启动第一个Activity组件的时候，就会请求与WindowManagerService服务建立一个连接，以便可以配合WindowManagerService服务来管理系统中的所有窗口。具体来说，就是应用程序进程在为它里面启动的第一个Activity组件的视图对象创建一个关联的ViewRoot对象的时候，就会向WindowManagerService服务请求返回一个类型为Session的Binder本地对象，这样应用程序进程就可以获得一个类型为Session的Binder代理对象，以后就可以通过这个Binder代理对象来和WindowManagerService服务进行通信了。

       从前面[Android应用程序窗口（Activity）的视图对象（View）的创建过程分析](http://blog.csdn.net/luoshengyang/article/details/8245546)一文可以知道，应用程序进程在为它里面启动的Activity组件的视图对象创建关联ViewRoot对象是通过调用WindowManagerImpl类的成员函数addView来实现的，因此，接下来我们就从WindowManagerImpl类的成员函数addView开始分析应用程序进程与WindowManagerService服务的连接过程，即一个Session对象的创建过程，如图5所示：

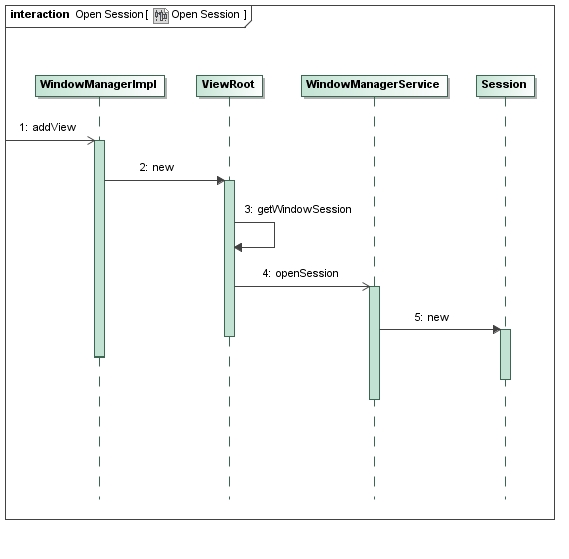


图5 Session对象的创建过程

        这个过程可以分为5个步骤，接下来我们就详细分析每一个步骤。

        Step 1. WindowManagerImpl.addView

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public class WindowManagerImpl implements WindowManager {
2. ......
4. private void addView(View view, ViewGroup.LayoutParams params, boolean nest)
5. {
6. ......
8. ViewRoot root;
9. ......
11. synchronized (this) {
12. ......
14. root = new ViewRoot(view.getContext());
16. ......
17. }
18. // do this last because it fires off messages to start doing things
19. root.setView(view, wparams, panelParentView);
20. }
22. ......
23. }

public class WindowManagerImpl implements WindowManager {

......

private void addView(View view, ViewGroup.LayoutParams params, boolean nest)

{

......

ViewRoot root;

......

synchronized (this) {

......

root = new ViewRoot(view.getContext());

......

}

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

root.setView(view, wparams, panelParentView);

}

......

}

        这个函数定义在文件frameworks/base/core/java/android/view/WindowManagerImpl.java中。

        这里的参数view即为正在启动的Activity组件的视图对象，WindowManagerImpl类的成员函数addView的目标就是为它创建一个ViewRoot对象。这里我们只关注这个ViewRoot对象的创建过程，即ViewRoot类的构造函数的实现，WindowManagerImpl类的成员函数addView的详细实现可以参考前面[Android应用程序窗口（Activity）的视图对象（View）的创建过程分析](http://blog.csdn.net/luoshengyang/article/details/8245546)一文的Step 12。

        Step 2.new ViewRoot

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public final class ViewRoot extends Handler implements ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. public ViewRoot(Context context) {
6. super();
8. ......
10. // Initialize the statics when this class is first instantiated. This is
11. // done here instead of in the static block because Zygote does not
12. // allow the spawning of threads.
13. getWindowSession(context.getMainLooper());
15. ......
16. }
18. ......
19. }

public final class ViewRoot extends Handler implements ViewParent,

View.AttachInfo.Callbacks {

......

public ViewRoot(Context context) {

super();

......

// Initialize the statics when this class is first instantiated. This is

// done here instead of in the static block because Zygote does not

// allow the spawning of threads.

getWindowSession(context.getMainLooper());

......

}

......

}

        这个函数定义在文件frameworks/base/core/java/android/view/ViewRoot.java文件中。

        ViewRoot类的构造函数是通过调用静态成员函数getWindowSession来请求WindowManagerService服务为应用程序进程创建一个返回一个类型为Session的Binder本地对象的，因此，接下来我们就继续分析ViewRoot类的静态成员函数getWindowSession的实现。

        Step 3. ViewRoot.getWindowSession

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public final class ViewRoot extends Handler implements ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. static IWindowSession sWindowSession;
7. static final Object mStaticInit = new Object();
8. static boolean mInitialized = false;
9. ......
11. public static IWindowSession getWindowSession(Looper mainLooper) {
12. synchronized (mStaticInit) {
13. if (!mInitialized) {
14. try {
15. InputMethodManager imm = InputMethodManager.getInstance(mainLooper);
16. sWindowSession = IWindowManager.Stub.asInterface(
17. ServiceManager.getService("window"))
18. .openSession(imm.getClient(), imm.getInputContext());
19. mInitialized = true;
20. } catch (RemoteException e) {
21. }
22. }
23. return sWindowSession;
24. }
25. }
27. ......
28. }

public final class ViewRoot extends Handler implements ViewParent,

View.AttachInfo.Callbacks {

......

static IWindowSession sWindowSession;

static final Object mStaticInit = new Object();

static boolean mInitialized = false;

......

public static IWindowSession getWindowSession(Looper mainLooper) {

synchronized (mStaticInit) {

if (!mInitialized) {

try {

InputMethodManager imm = InputMethodManager.getInstance(mainLooper);

sWindowSession = IWindowManager.Stub.asInterface(

ServiceManager.getService("window"))

.openSession(imm.getClient(), imm.getInputContext());

mInitialized = true;

} catch (RemoteException e) {

}

}

return sWindowSession;

}

}

......

}

       这个函数定义在文件frameworks/base/core/java/android/view/ViewRoot.java文件中。

       ViewRoot类的静态成员函数getWindowSession首先获得获得应用程序所使用的输入法管理器，接着再获得系统中的WindowManagerService服务的一个Binder代理对象。有了这个Binder代理对象之后，就可以调用它的成员函数openSession来请求WindowManagerService服务返回一个类型为Session的Binder本地对象。这个Binder本地对象返回来之后，就变成了一个类型为Session的Binder代理代象，即一个实现了IWindowSession接口的Binder代理代象，并且保存在ViewRoot类的静态成员变量sWindowSession中。在请求WindowManagerService服务返回一个类型为Session的Binder本地对象的时候，应用程序进程传递给WindowManagerService服务的参数有两个，一个是实现IInputMethodClient接口的输入法客户端对象，另外一个是实现了IInputContext接口的一个输入法上下文对象，它们分别是通过调用前面所获得的一个输入法管理器的成员函数getClient和getInputContext来获得的。

       从这里就可以看出，只有当ViewRoot类的静态成员函数getWindowSession第一次被调用的时候，应用程序进程才会请求WindowManagerService服务返回一个类型为Session的Binder本地对象。又由于ViewRoot类的静态成员函数getWindowSession第一次被调用的时候，正好是处于应用程序进程中的第一个Activity组件启动的过程中，因此，应用程序进程是在启动第一个Activity组件的时候，才会请求与WindowManagerService服务建立连接。一旦连接建立起来之后，ViewRoot类的静态成员变量mInitialized的值就会等于true，并且另外一个静态成员变量sWindowSession的值不等于null。同时，ViewRoot类的静态成员函数getWindowSession是线程安全的，这样就可以避免多个线程同时调用它来重复请求WindowManagerService服务为当前应用程序进程创建连接。

       接下来，我们就继续分析求WindowManagerService类的成员函数openSession的实现，以便可以了解一个Session对象的创建过程。

       Step 4. indowManagerService.openSession

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public class WindowManagerService extends IWindowManager.Stub
2. implements Watchdog.Monitor {
3. ......
5. public IWindowSession openSession(IInputMethodClient client,
6. IInputContext inputContext) {
7. if (client == null) throw new IllegalArgumentException("null client");
8. if (inputContext == null) throw new IllegalArgumentException("null inputContext");
9. Session session = new Session(client, inputContext);
10. return session;
11. }
13. ......
14. }

public class WindowManagerService extends IWindowManager.Stub

implements Watchdog.Monitor {

......

public IWindowSession openSession(IInputMethodClient client,

IInputContext inputContext) {

if (client == null) throw new IllegalArgumentException("null client");

if (inputContext == null) throw new IllegalArgumentException("null inputContext");

Session session = new Session(client, inputContext);

return session;

}

......

}

       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openSession的实现很简单，它以应用程序进程传递过来的一个输入法客户端对象和一个输入法上下文对象来参数，来创建一个类型为Session的Binder本地对象，并且将这个类型为Session的Binder本地对象返回给应用程序进程。

       接下来我们就继续分析Session类的构造函数的实现，以便可以了解一个Session对象的创建过程。

       Step 5.new Session

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public class WindowManagerService extends IWindowManager.Stub
2. implements Watchdog.Monitor {
3. ......
5. final boolean mHaveInputMethods;
6. ......
8. IInputMethodManager mInputMethodManager;
9. ......
11. private final class Session extends IWindowSession.Stub
12. implements IBinder.DeathRecipient {
13. final IInputMethodClient mClient;
14. final IInputContext mInputContext;
15. ......
17. public Session(IInputMethodClient client, IInputContext inputContext) {
18. mClient = client;
19. mInputContext = inputContext;
20. ......
22. synchronized (mWindowMap) {
23. if (mInputMethodManager == null && mHaveInputMethods) {
24. IBinder b = ServiceManager.getService(
25. Context.INPUT\_METHOD\_SERVICE);
26. mInputMethodManager = IInputMethodManager.Stub.asInterface(b);
27. }
28. }
29. long ident = Binder.clearCallingIdentity();
30. try {
31. // Note: it is safe to call in to the input method manager
32. // here because we are not holding our lock.
33. if (mInputMethodManager != null) {
34. mInputMethodManager.addClient(client, inputContext,
35. mUid, mPid);
36. } else {
37. client.setUsingInputMethod(false);
38. }
39. client.asBinder().linkToDeath(this, 0);
40. } catch (RemoteException e) {
41. ......
42. } finally {
43. Binder.restoreCallingIdentity(ident);
44. }
45. }
47. ......
48. }
50. ......
51. }

public class WindowManagerService extends IWindowManager.Stub

implements Watchdog.Monitor {

......

final boolean mHaveInputMethods;

......

IInputMethodManager mInputMethodManager;

......

private final class Session extends IWindowSession.Stub

implements IBinder.DeathRecipient {

final IInputMethodClient mClient;

final IInputContext mInputContext;

......

public Session(IInputMethodClient client, IInputContext inputContext) {

mClient = client;

mInputContext = inputContext;

......

synchronized (mWindowMap) {

if (mInputMethodManager == null && mHaveInputMethods) {

IBinder b = ServiceManager.getService(

Context.INPUT\_METHOD\_SERVICE);

mInputMethodManager = IInputMethodManager.Stub.asInterface(b);

}

}

long ident = Binder.clearCallingIdentity();

try {

// Note: it is safe to call in to the input method manager

// here because we are not holding our lock.

if (mInputMethodManager != null) {

mInputMethodManager.addClient(client, inputContext,

mUid, mPid);

} else {

client.setUsingInputMethod(false);

}

client.asBinder().linkToDeath(this, 0);

} catch (RemoteException e) {

......

} finally {

Binder.restoreCallingIdentity(ident);

}

}

......

}

......

}

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

        Session类有两个成员变量mClient和mInputContext，分别用来保存在从应用程序进程传递过来的一个输入法客户端对象和一个输入法上下文对象，它们是在Session类的构造函数中初始化的。

        Session类的构造函数还会检查WindowManagerService服务是否需要获得系统中的输入法管理器服务，即检查WindowManagerService类的成员变量mHaveInputMethods的值是否等于true。如果这个值等于true，并且WindowManagerService服务还没有获得系统中的输入法管理器服务，即WindowManagerService类的成员变量mInputMethodManager的值等于null，那么Session类的构造函数就会首先获得这个输入法管理器服务，并且保存在WindowManagerService类的成员变量mInputMethodManager中。

       获得了系统中的输入法管理器服务之后，Session类的构造函数就可以调用它的成员函数addClient来为正在请求与WindowManagerService服务建立连接的应用程序进程增加它所使用的输入法客户端对象和输入法上下文对象了。

       至此，我们就分析完成一个Session对象的创建过程了，通过这个过程我们就可以知道，每一个应用程序进程在WindowManagerService服务内部都有一个类型为Session的Binder本地对象，用来它与WindowManagerService服务之间的连接，而有了这个连接之后，WindowManagerService服务就可以请求应用进程配合管理系统中的应用程序窗口了。

      三. WindowState对象的创建过程

      在Android系统中，WindowManagerService服务负责统一管理系统中的所有窗口，因此，当运行在应用程序进程这一侧的Activity组件在启动完成之后，需要与WindowManagerService服务建立一个连接，以便WindowManagerService服务可以管理它的窗口。具体来说，就是应用程序进程将一个Activitty组件的视图对象设置到与它所关联的一个ViewRoot对象的内部的时候，就会将一个实现了IWindow接口的Binder本地对象传递WindowManagerService服务。这个实现了IWindow接口的Binder本地对象唯一地标识了一个Activity组件，WindowManagerService服务接收到了这个Binder本地对象之后，就会将它保存在一个新创建的WindowState对象的内部，这样WindowManagerService服务以后就可以通过它来和Activity组件通信，以便可以要求Activity组件配合来管理系系统中的所有窗口。

       从前面[Android应用程序窗口（Activity）的视图对象（View）的创建过程分析](http://blog.csdn.net/luoshengyang/article/details/8245546)一文可以知道，应用程序进程将一个Activity组件的视图对象设置到与它所关联的一个ViewRoot对象的内部是通过调用ViewRoot类的成员函数setView来实现的，因此，接下来我们就从ViewRoot类的成员函数setView开始分析Activity组件与WindowManagerService服务的连接过程，即一个WindowState对象的创建过程，如图6所示：

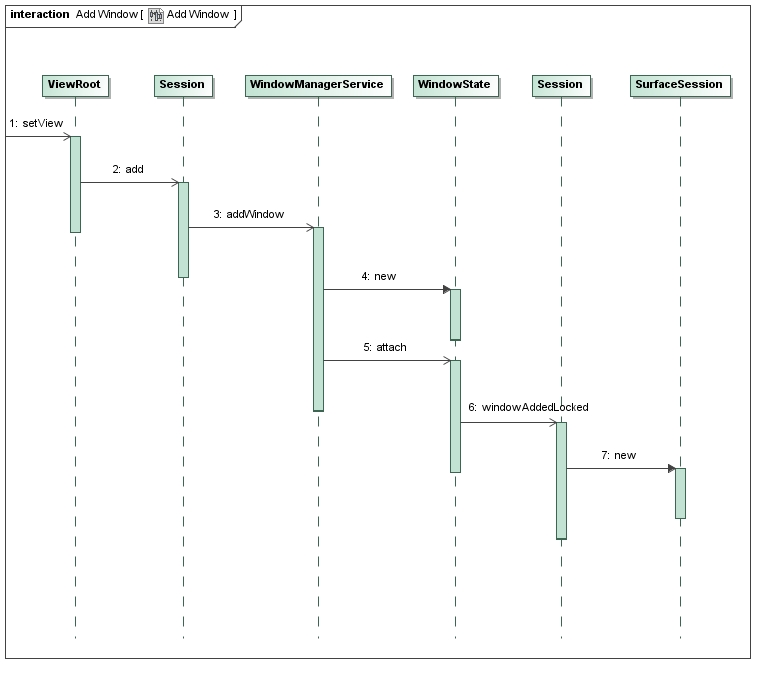


图6 WindowState对象的创建过程

        这个过程可以分为7个步骤，接下来我们就详细分析每一个步骤。

        Step 1. ViewRoot.setView

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public final class ViewRoot extends Handler implements ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. static IWindowSession sWindowSession;
6. ......
8. final W mWindow;
10. View mView;
11. ......
13. public ViewRoot(Context context) {
14. super();
15. ......
17. mWindow = new W(this, context);
18. ......
20. }
22. ......
24. public void setView(View view, WindowManager.LayoutParams attrs,
25. View panelParentView) {
26. synchronized (this) {
27. if (mView == null) {
28. mView = view;
29. ......
31. try {
32. res = sWindowSession.add(mWindow, mWindowAttributes,
33. getHostVisibility(), mAttachInfo.mContentInsets,
34. mInputChannel);
35. } catch (RemoteException e) {
36. ......
37. } finally {
38. ......
39. }
41. ......
42. }
44. ......
45. }
46. }
48. ......
49. }

public final class ViewRoot extends Handler implements ViewParent,

View.AttachInfo.Callbacks {

......

static IWindowSession sWindowSession;

......

final W mWindow;

View mView;

......

public ViewRoot(Context context) {

super();

......

mWindow = new W(this, context);

......

}

......

public void setView(View view, WindowManager.LayoutParams attrs,

View panelParentView) {

synchronized (this) {

if (mView == null) {

mView = view;

......

try {

res = sWindowSession.add(mWindow, mWindowAttributes,

getHostVisibility(), mAttachInfo.mContentInsets,

mInputChannel);

} catch (RemoteException e) {

......

} finally {

......

}

......

}

......

}

}

......

}

        这个函数定义在文件frameworks/base/core/java/android/view/ViewRoot.java文件中。

        这里的参数view即为正在启动的Activity组件的视图对象，ViewRoot类的成员函数setView会将它保存成员变量mView中，这样就可以将一个Activity组件的视图对象和一个ViewRoot对象关联起来。ViewRoot类的成员函数setView接下来还会调用静态成员变量sWindowSession所描述的一个实现了IWindowSession接口的Binder代理对象的成员函数add来请求WindowManagerService服务为正在启动的Activity组件创建一个WindowState对象。接下来我们就主要关注WindowState对象的创建过程，ViewRoot类的成员函数setView的详细实现可以参考前面[Android应用程序窗口（Activity）的视图对象（View）的创建过程分析](http://blog.csdn.net/luoshengyang/article/details/8245546)一文的Step 13。

        注意，ViewRoot类的成员函数setView在请求WindowManagerService服务为正在启动的Activity组件创建一个WindowState对象的时候，会传递一个类型为W的Binder本地对象给WindowManagerService服务。这个类型为W的Binder本地对象实现了IWindow接口，保存在ViewRoot类的成员变量mWindow中，它是在ViewRoot类的构造函数中创建的，以后WindowManagerService服务就会通过它来和Activity组件通信。

       从前面第二部的内容可以知道，ViewRoot类的静态成员变量sWindowSession所指向的一个Binder代理对象引用的是运行在WindowManagerService服务这一侧的一个Session对象，因此，接下来我们就继续分析Session类的成员函数add的实现，以便可以了解一个WindowState对象的创建过程。

       Step 2. Session.add

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public class WindowManagerService extends IWindowManager.Stub
2. implements Watchdog.Monitor {
3. ......
5. private final class Session extends IWindowSession.Stub
6. implements IBinder.DeathRecipient {
7. ......
9. public int add(IWindow window, WindowManager.LayoutParams attrs,
10. int viewVisibility, Rect outContentInsets, InputChannel outInputChannel) {
11. return addWindow(this, window, attrs, viewVisibility, outContentInsets,
12. outInputChannel);
13. }
15. ......
16. }
18. ......
19. }

public class WindowManagerService extends IWindowManager.Stub

implements Watchdog.Monitor {

......

private final class Session extends IWindowSession.Stub

implements IBinder.DeathRecipient {

......

public int add(IWindow window, WindowManager.LayoutParams attrs,

int viewVisibility, Rect outContentInsets, InputChannel outInputChannel) {

return addWindow(this, window, attrs, viewVisibility, outContentInsets,

outInputChannel);

}

......

}

......

}

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

        Session类的成员函数add的实现很简单，它只是调用了外部类WindowManagerService的成员函数addWindow来进一步为正在启动的Activity组件创建一个WindowState对象。

        Step 3. WindowManagerService.addWindow

        这个函数定义在文件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 plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public class WindowManagerService extends IWindowManager.Stub
2. implements Watchdog.Monitor {
3. ......
5. public int addWindow(Session session, IWindow client,
6. WindowManager.LayoutParams attrs, int viewVisibility,
7. Rect outContentInsets, InputChannel outInputChannel) {
8. ......
10. WindowState attachedWindow = null;
11. WindowState win = null;
13. synchronized(mWindowMap) {
14. ......
16. if (mWindowMap.containsKey(client.asBinder())) {
17. ......
18. return WindowManagerImpl.ADD\_DUPLICATE\_ADD;
19. }

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) {

......

WindowState attachedWindow = null;

WindowState win = null;

synchronized(mWindowMap) {

......

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

......

return WindowManagerImpl.ADD\_DUPLICATE\_ADD;

}

        这段代码首先在WindowManagerService类的成员变量mWindowMap所描述的一个HashMap中检查是否存在一个与参数client所对应的WindowState对象，如果已经存在，那么就说明WindowManagerService服务已经为它创建过一个WindowState对象了，因此，这里就不会继续往前执行，而是直接返回一个错误码WindowManagerImpl.ADD\_DUPLICATE\_ADD。

        我们继续往前看代码：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. if (attrs.type >= FIRST\_SUB\_WINDOW && attrs.type <= LAST\_SUB\_WINDOW) {
2. attachedWindow = windowForClientLocked(null, attrs.token, false);
3. if (attachedWindow == null) {
4. ......
5. return WindowManagerImpl.ADD\_BAD\_SUBWINDOW\_TOKEN;
6. }
7. if (attachedWindow.mAttrs.type >= FIRST\_SUB\_WINDOW
8. && attachedWindow.mAttrs.type <= LAST\_SUB\_WINDOW) {
9. ......
10. return WindowManagerImpl.ADD\_BAD\_SUBWINDOW\_TOKEN;
11. }
12. }

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

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

if (attachedWindow == null) {

......

return WindowManagerImpl.ADD\_BAD\_SUBWINDOW\_TOKEN;

}

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

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

......

return WindowManagerImpl.ADD\_BAD\_SUBWINDOW\_TOKEN;

}

}

        参数attrs指向的是一个WindowManager.LayoutParams对象，用来描述正在启动的Activity组件的UI布局，当它的成员变量type的值大于等于FIRST\_SUB\_WINDOW并且小于等于LAST\_SUB\_WINDOW的时候，就说明现在要增加的是一个子窗口。在这种情况下，就必须要指定一个父窗口，而这个父窗口是通过数attrs指向的是一个WindowManager.LayoutParams对象的成员变量token来指定的，因此，这段代码就会调用WindowManagerService类的另外一个成员函数windowForClientLocked来获得用来描述这个父窗口的一个WindowState对象，并且保存在变量attachedWindow。

        如果得到变量attachedWindow的值等于null，那么就说明父窗口不存在，这是不允许的，因此，函数就不会继续向前执行，而是直接返回一个错误码WindowManagerImpl.ADD\_BAD\_SUBWINDOW\_TOKEN。另一方面，如果变量attachedWindow的值不等于null，但是它的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量type的值也是大于等于FIRST\_SUB\_WINDOW并且小于等于LAST\_SUB\_WINDOW，那么也说明找到的父窗口也是一个子窗口，这种情况也是不允许的，因此，函数就不会继续向前执行，而是直接返回一个错误码WindowManagerImpl.ADD\_BAD\_SUBWINDOW\_TOKEN。

       我们继续往前看代码：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. boolean addToken = false;
2. WindowToken token = mTokenMap.get(attrs.token);
3. if (token == null) {
4. if (attrs.type >= FIRST\_APPLICATION\_WINDOW
5. && attrs.type <= LAST\_APPLICATION\_WINDOW) {
6. ......
7. return WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;
8. }
9. if (attrs.type == TYPE\_INPUT\_METHOD) {
10. ......
11. return WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;
12. }
13. if (attrs.type == TYPE\_WALLPAPER) {
14. ......
15. return WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;
16. }
17. token = new WindowToken(attrs.token, -1, false);
18. addToken = true;
19. } else if (attrs.type >= FIRST\_APPLICATION\_WINDOW
20. && attrs.type <= LAST\_APPLICATION\_WINDOW) {
21. AppWindowToken atoken = token.appWindowToken;
22. if (atoken == null) {
23. ......
24. return WindowManagerImpl.ADD\_NOT\_APP\_TOKEN;
25. } else if (atoken.removed) {
26. ......
27. return WindowManagerImpl.ADD\_APP\_EXITING;
28. }
29. if (attrs.type == TYPE\_APPLICATION\_STARTING && atoken.firstWindowDrawn) {
30. // No need for this guy!
31. ......
32. return WindowManagerImpl.ADD\_STARTING\_NOT\_NEEDED;
33. }
34. } else if (attrs.type == TYPE\_INPUT\_METHOD) {
35. if (token.windowType != TYPE\_INPUT\_METHOD) {
36. ......
37. return WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;
38. }
39. } else if (attrs.type == TYPE\_WALLPAPER) {
40. if (token.windowType != TYPE\_WALLPAPER) {
41. ......
42. return WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;
43. }
44. }

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;

} else if (attrs.type >= FIRST\_APPLICATION\_WINDOW

&& attrs.type <= LAST\_APPLICATION\_WINDOW) {

AppWindowToken atoken = token.appWindowToken;

if (atoken == null) {

......

return WindowManagerImpl.ADD\_NOT\_APP\_TOKEN;

} else if (atoken.removed) {

......

return WindowManagerImpl.ADD\_APP\_EXITING;

}

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

// No need for this guy!

......

return WindowManagerImpl.ADD\_STARTING\_NOT\_NEEDED;

}

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

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

......

return WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;

}

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

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

......

return WindowManagerImpl.ADD\_BAD\_APP\_TOKEN;

}

}

         这段代码首先在WindowManagerService类的成员变量mTokenMap所描述的一个HashMap中检查WindowManagerService服务是否已经为正在增加的窗口创建过一个WindowToken对象了。

         如果还没有创建过WindowToken对，即变量token的值等于null，那么这段代码就会进一步检查正在增加的窗口的类型。如果正在增加的窗口是属于应用程序窗口（FIRST\_APPLICATION\_WINDOW~LAST\_APPLICATION\_WINDOW，即Activity组件窗口），或者输入法窗口（TYPE\_INPUT\_METHOD），或者壁纸窗口（TYPE\_WALLPAPER），那么这三种情况都是不允许的，因此，函数就不会继续向前执行，而是直接返回一个错误码WindowManagerImpl.ADD\_BAD\_APP\_TOKEN。例如，对于应用程序窗口来说，与它所对应的Activity组件是在之前就已经由ActivityManagerService服务请求WindowManagerService服务创建过一个AppWindowToken对象了的（AppWindowToken对象也是一个WindowToken对象，因为AppWindowToken类继承了WindowToken类），这个过程可以参考前面第一部分的内容。如果不是上述三种情况，那么这段代码就会为正在增加的窗口创建一个WindowToken对象，并且保存在变量token中。

        如果已经创建过WindowToken对象，即变量token的值不等于null，那么就说明正在增加的窗口或者是应用程序窗口，或者是输入法窗口，或者是壁纸窗口。下面我们就为三种情况来讨论。

        如果是应用程序窗口，从前面第一部分的内容可以知道，WindowToken对象token的成员变量appWindowToken的值必须不能等于null，并且指向的是一个AppWindowToken对象。因此，当WindowToken对象token的成员变量appWindowToken的值等于null的时候，函数就不会继续向前执行，而是直接返回一个错误码WindowManagerImpl.ADD\_NOT\_APP\_TOKEN。另一方面，虽然WindowToken对象token的成员变量appWindowToken的值不等于null，但是它所指向的一个AppWindowToken对象的成员变量removed的值等于true时，那么就表示对应的Activity组件已经被移除，在这种情况下，函数也不会继续向前执行，而是直接返回一个错误码WindowManagerImpl.ADD\_APP\_EXITING。还有一种特殊的应用程序窗口，它的类型为TYPE\_APPLICATION\_STARTING。这种类型的窗口称为起始窗口，它是在一个Activity组件的窗口显示出来之前就显示的。因此，如果当前正在增加的是一个超始窗口，并且它所附属的应用程序窗口，即变量atoken所描述的应用程序窗口，已经显示出来了，即变量atoken所指向的一个AppWindowToken对象的成员变量firstWindowDrawn的值等于true，那么函数也不会继续向前执行，而是直接返回一个错误码WindowManagerImpl.ADD\_STARTING\_NOT\_NEEDED。

        如果是输入法窗口，但是参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量windowType的值不等于TYPE\_INPUT\_METHOD，那么指定的窗口类型就是不匹配的。在这种情况下，函数就不会继续向前执行，而是直接返回一个错误码WindowManagerImpl.ADD\_BAD\_APP\_TOKEN。

        如果是壁纸窗口，但是参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量windowType的值不等于TYPE\_WALLPAPER，那么指定的窗口类型就是不匹配的。在这种情况下，函数也不会继续向前执行，而是直接返回一个错误码WindowManagerImpl.ADD\_BAD\_APP\_TOKEN。

        我们继续往前看代码：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. win = new WindowState(session, client, token,
2. attachedWindow, attrs, viewVisibility);
3. ......
5. mPolicy.adjustWindowParamsLw(win.mAttrs);
7. res = mPolicy.prepareAddWindowLw(win, attrs);
8. if (res != WindowManagerImpl.ADD\_OKAY) {
9. return res;
10. }

win = new WindowState(session, client, token,

attachedWindow, attrs, viewVisibility);

......

mPolicy.adjustWindowParamsLw(win.mAttrs);

res = mPolicy.prepareAddWindowLw(win, attrs);

if (res != WindowManagerImpl.ADD\_OKAY) {

return res;

}

        通过上面的合法性检查之后，这里就可以为正在增加的窗口创建一个WindowState对象了。

        WindowManagerService类的成员变量mPolicy指向的是一个实现了WindowManagerPolicy接口的窗口管理策略器。在Phone平台中，这个窗口管理策略器是由com.android.internal.policy.impl.PhoneWindowManager来实现的，它负责对系统中的窗口实现制定一些规则。这里主要是调用窗口管理策略器的成员函数adjustWindowParamsLw来调整当前正在增加的窗口的布局参数，以及调用成员函数prepareAddWindowLw来检查当前应用程序进程请求增加的窗口是否是合法的。如果不是合法的，即变量res的值不等于WindowManagerImpl.ADD\_OKAY，那么函数就不会继续向前执行，而直接返回错误码res。

        WindowState对象的创建过程我们在接下来的Step 4中再分析，现在我们继续向前看代码：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. if (outInputChannel != null) {
2. String name = win.makeInputChannelName();
3. InputChannel[] inputChannels = InputChannel.openInputChannelPair(name);
4. win.mInputChannel = inputChannels[0];
5. inputChannels[1].transferToBinderOutParameter(outInputChannel);
6. mInputManager.registerInputChannel(win.mInputChannel);
7. }

if (outInputChannel != null) {

String name = win.makeInputChannelName();

InputChannel[] inputChannels = InputChannel.openInputChannelPair(name);

win.mInputChannel = inputChannels[0];

inputChannels[1].transferToBinderOutParameter(outInputChannel);

mInputManager.registerInputChannel(win.mInputChannel);

}

       这段代码是用创建IO输入事件连接通道的，以便正在增加的窗口可以接收到系统所发生的键盘和触摸屏事件，可以参考前面[Android应用程序键盘（Keyboard）消息处理机制分析](http://blog.csdn.net/luoshengyang/article/details/6882903)一文，这里不再详述。

       我们继续向前看代码：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. if (addToken) {
2. mTokenMap.put(attrs.token, token);
3. mTokenList.add(token);
4. }
5. win.attach();
6. mWindowMap.put(client.asBinder(), win);
8. if (attrs.type == TYPE\_APPLICATION\_STARTING &&
9. token.appWindowToken != null) {
10. token.appWindowToken.startingWindow = win;
11. }

if (addToken) {

mTokenMap.put(attrs.token, token);

mTokenList.add(token);

}

win.attach();

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

if (attrs.type == TYPE\_APPLICATION\_STARTING &&

token.appWindowToken != null) {

token.appWindowToken.startingWindow = win;

}

        这段代码首先检查变量addToken的值是否等于true。如果等于true的话，那么就说明变量token所指向的一个WindowToken对象是在前面新创建的。在这种情况下，就需要将这个新创建的WindowToken对象分别添加到WindowManagerService类的成员变量mTokeMap和mTokenList分别描述的一个HashMap和一个ArrayList中去。

        这段代码接下来再调用前面所创建的一个WindowState对象win的成员函数attach来为当前正在增加的窗口创建一个用来连接到SurfaceFlinger服务的SurfaceSession对象。有了这个SurfaceSession对象之后，当前正在增加的窗口就可以和SurfaceFlinger服务通信了。在接下来的Step 5中，我们再详细分析WindowState类的成员函数attach的实现。

        这段代码最后还做了两件事情。第一件事情是将前面所创建的一个WindowState对象win添加到WindowManagerService类的成员变量mWindowMap所描述的一个HashMap中，这是以参数所描述的一个类型为W的Binder代理对象的IBinder接口来键值来保存的。第二件事情是检查当前正在增加的是否是一个起始窗口，如果是的话，那么就会将前面所创建的一个WindowState对象win设置到用来描述它所属的Activity组件的一个AppWindowToken对象的成员变量startingWindow中去，这样系统就可以在显示这个Activity组件的窗口之前，先显示该起始窗口。

        我们继续向前看代码：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. boolean imMayMove = true;
3. if (attrs.type == TYPE\_INPUT\_METHOD) {
4. mInputMethodWindow = win;
5. addInputMethodWindowToListLocked(win);
6. imMayMove = false;
7. } else if (attrs.type == TYPE\_INPUT\_METHOD\_DIALOG) {
8. mInputMethodDialogs.add(win);
9. addWindowToListInOrderLocked(win, true);
10. adjustInputMethodDialogsLocked();
11. imMayMove = false;
12. } else {
13. addWindowToListInOrderLocked(win, true);
14. if (attrs.type == TYPE\_WALLPAPER) {
15. ......
16. adjustWallpaperWindowsLocked();
17. } else if ((attrs.flags&FLAG\_SHOW\_WALLPAPER) != 0) {
18. adjustWallpaperWindowsLocked();
19. }
20. }

boolean imMayMove = true;

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

mInputMethodWindow = win;

addInputMethodWindowToListLocked(win);

imMayMove = false;

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

mInputMethodDialogs.add(win);

addWindowToListInOrderLocked(win, true);

adjustInputMethodDialogsLocked();

imMayMove = false;

} else {

addWindowToListInOrderLocked(win, true);

if (attrs.type == TYPE\_WALLPAPER) {

......

adjustWallpaperWindowsLocked();

} else if ((attrs.flags&FLAG\_SHOW\_WALLPAPER) != 0) {

adjustWallpaperWindowsLocked();

}

}

        WindowManagerService类有一个成员变量mWindows，它是一个类型为WindowState的ArrayList，保存在它里面的WindowState对象所描述的窗口是按照它们的Z轴坐标从小到大的顺序来排列的。现在我们需要考虑将用来描述当前正在增加的窗口的WindowState对象win添加到这个ArrayList的合适位置中去。我们分三种情况考虑。

        如果当前正在增加的窗口是一个输入法窗口，那么WindowManagerService服务就需要按照Z轴坐标从大到小的顺序来检查当前是哪一个窗口是需要输入法窗口的。找到了这个位于最上面的需要输入法窗口的窗口之后，就可以将输入法窗口放置在它的上面了，即可以将WindowState对象win添加到WindowManagerService类的成员变量mWindows所描述的一个ArrayList的合适位置了。这两个操作是通过调用WindowManagerService类的成员函数addInputMethodWindowToListLocked来实现的。同时，WindowManagerService类的成员变量mInputMethodWindow指向的是系统当前正在使用的输入法窗口，因此，在这种情况下，我们还需要将WindowState对象win保存在它里面。

        如果当前正在添加的窗口是一个输入法对话框，那么WindowManagerService服务就需要做两件事情。第一件事情是将WindowState对象win添加到WindowManagerService类的成员变量mWindows所描述的一个ArrayList中去，这是通过调用WindowManagerService类的成员函数addWindowToListInOrderLocked来实现的。第二件事情是调用WindowManagerService类的成员函数adjustInputMethodDialogsLocked来调整正在增加的输入法对话框在WindowManagerService类的成员变量mWindows所描述的一个ArrayList中的位置，使得它位于输入法窗口的上面。

       在上述两种情况下，系统的输入法窗口和输入法对话框的位置都是得到合适的调整的了，因此，这段代码就会将变量imMayMove的值设置为false，表示后面不需要再调整输入法窗口和输入法对话框的位置了。

       如果当前正在添加的窗口是一个应用程序窗口或者壁纸窗口，那么WindowManagerService服务都需要将WindowState对象win添加到WindowManagerService类的成员变量mWindows所描述的一个ArrayList中去。添加完成之后，如果发现当前添加的窗口是一个壁纸窗口或者当前添加的是一个需要显示壁纸的应用程序窗口，那么WindowManagerService服务还需要进一步调整系统中已经添加了的壁纸窗口的Z轴位置，使得它们位于最上面的需要显示壁纸的窗口的下面，这是通过调用WindowManagerService类的成员函数adjustWallpaperWindowsLocked来实现的。

       我们继续向前看代码：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. win.mEnterAnimationPending = true;
3. mPolicy.getContentInsetHintLw(attrs, outContentInsets);
5. if (mInTouchMode) {
6. res |= WindowManagerImpl.ADD\_FLAG\_IN\_TOUCH\_MODE;
7. }
8. if (win == null || win.mAppToken == null || !win.mAppToken.clientHidden) {
9. res |= WindowManagerImpl.ADD\_FLAG\_APP\_VISIBLE;
10. }

win.mEnterAnimationPending = true;

mPolicy.getContentInsetHintLw(attrs, outContentInsets);

if (mInTouchMode) {

res |= WindowManagerImpl.ADD\_FLAG\_IN\_TOUCH\_MODE;

}

if (win == null || win.mAppToken == null || !win.mAppToken.clientHidden) {

res |= WindowManagerImpl.ADD\_FLAG\_APP\_VISIBLE;

}

        这段代码做了四件事情。

        第一件事情是将前面所创建的一个WindowState对象win的成员变量mEnterAnimationPending的值设置为true，表示当前正在增加的窗口需要显示一个进入动画。

        第二件事情是调用WindowManagerService类的成员变量mPolicy所描述的一个窗口管理策略器的成员函数getContentInsetHintLw来获得当前正在增加的窗口的UI内容边衬大小，即当前正在增加的窗口可以在屏幕中所获得的用来显示UI内容的区域的大小，这通常是要排除屏幕边框和状态栏等所占据的屏幕区域。

       第三件事情是检查WindowManagerService类的成员变量mInTouchMode的值是否等于true。如果等于true的话，那么就说明系统运行在触摸屏模式中，这时候这段代码就会将返回值res的WindowManagerImpl.ADD\_FLAG\_IN\_TOUCH\_MODE位设置为1。

       第四件事情是检查当前正在增加的窗口是否是处于可见的状态。从第二个if语句可以看出，由于WindowState对象win的值在这里不可以等于null，因此，这里只有两种情况下，前正在增加的窗口是处于可见状态的。第一种情况是WindowState对象的成员变量mAppToken的值等于null，这表明当前正在增加的窗口不是一个应用程序窗口，即不是一个Activity组件窗口，那么它就有可能是一个子窗口。由于子窗口通常是在其父窗口处于可见的状态下才会创建的，因此，这个子窗口就需要马上显示出来的，即需要将它的状态设置为可见的。第二种情况是WindowState对象的成员变量mAppToken的值不等于null，这表明当前正在增加的窗口是一个应用程序窗口。在这种情况下，WindowState对象的成员变量mAppToken指向的就是一个AppWindowToken对象。当这个AppWindowToken对象的成员变量clientHidden的值等于false的时候，就表明它所描述的一个Activity组件是处于可见状态的，因此，这时候就需要将该Activity组件的窗口（即当前正在增加的窗口）的状态设置为可见的。在当前正在增加的窗口是处于可见状态的情况下，这段代码就会将返回值res的WindowManagerImpl.ADD\_FLAG\_APP\_VISIBLE位设置为1。

        我们继续向前看最后一段代码：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. boolean focusChanged = false;
2. if (win.canReceiveKeys()) {
3. focusChanged = updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_ASSIGN\_LAYERS);
4. if (focusChanged) {
5. imMayMove = false;
6. }
7. }
9. if (imMayMove) {
10. moveInputMethodWindowsIfNeededLocked(false);
11. }
13. assignLayersLocked();
14. ......
16. if (focusChanged) {
17. finishUpdateFocusedWindowAfterAssignLayersLocked();
18. }
20. ......
21. }
23. ...
25. return res;
26. }
28. ......

boolean focusChanged = false;

if (win.canReceiveKeys()) {

focusChanged = updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_ASSIGN\_LAYERS);

if (focusChanged) {

imMayMove = false;

}

}

if (imMayMove) {

moveInputMethodWindowsIfNeededLocked(false);

}

assignLayersLocked();

......

if (focusChanged) {

finishUpdateFocusedWindowAfterAssignLayersLocked();

}

......

}

...

return res;

}

......

}

        最后一段代码做了以下五件事情。

        第一件事情是检查当前正在增加的窗口是否是能够接收IO输入事件的，即键盘输入事件或者触摸屏输入事件。如果可以的话，那么就需要调用WindowManagerService类的成员函数updateFocusedWindowLocked来调整系统当前获得输入焦点的窗口，因为当前正在增加的窗口可能会成为新的可以获得输入焦点的窗口。如果WindowManagerService类的成员函数updateFocusedWindowLocked的返回值等于true，那么就表明系统当前获得输入焦点的窗口发生了变化。在这种情况下，WindowManagerService类的成员函数updateFocusedWindowLocked也会顺便调整输入法窗口的位置，使得它位于系统当前获得输入焦点的窗口的上面，因此，这时候这段代码也会将变量imMayMove的值设置为false，表示接下来不用调整输入法窗口的位置了。

       第二件事情是检查变量imMayMove的值是否等于true。如果等于true的话，那么就说明当前正在增加的窗口可能已经影响到系统的输入法窗口的Z轴位置了，因此，这段代码就需要调用WindowManagerService类的成员函数moveInputMethodWindowsIfNeededLocked来重新调整调整输入法窗口的Z轴位置，使得它可以位于最上面的那个需要输入法窗口的窗口的上面。

       第三件事情是调用WindowManagerService类的成员函数assignLayersLocked来调整系统中所有窗口的Z轴位置，这也是因为当前正在增加的窗口可能已经影响到系统中所有窗口的Z轴位置了。例如，假如当前增加的是一个正在启动的Activity组件的窗口，那么这个窗口的Z轴位置就应该是最大的，以便可以在最上面显示。又如，假如当前增加的是一个子窗口，那么这个子窗口就应该位于它的父窗口的上面。这些都要求重新调整系统中所有窗口的Z轴位置，以便每一个窗口都可以在一个正确的位置上来显示。

       第四件事情检查变量focusChanged的值是否等于true。如果等于true的话，那么就说明系统中获得输入焦点的窗口已经发生了变化。在这种情况下，这段代码就会调用WindowManagerService类的成员函数finishUpdateFocusedWindowAfterAssignLayersLocked来通知系统IO输入管理器，新的获得焦点的窗口是哪一个，以便系统IO输入管理器接下来可以将键盘输入事件或者触摸屏输入事件分发给新的获得焦点的窗口来处理。WindowManagerService类的成员函数finishUpdateFocusedWindowAfterAssignLayersLocked的实现可以参考前面[Android应用程序键盘（Keyboard）消息处理机制分析](http://blog.csdn.net/luoshengyang/article/details/6882903)一文，这里不再详述。

       第五件事情是变量res的值返回给应用程序进程，以便它可以知道请求WindowManagerService服务增加一个窗口的执行情况。

       至此，WindowManagerService类的成员函数addWindow的实现就分析完成了，接下来我们就继续分析WindowState类的构造函数和成员函数attach的实现，以便可以了解一个WindowSate对象及其相关联的一个SurfaceSession对象的创建过程。

       Step 4.new WindowState

       这个函数定义在文件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 plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public class WindowManagerService extends IWindowManager.Stub
2. implements Watchdog.Monitor {
3. ......
5. private final class WindowState implements WindowManagerPolicy.WindowState {
6. ......
8. WindowState(Session s, IWindow c, WindowToken token,
9. WindowState attachedWindow, WindowManager.LayoutParams a,
10. int viewVisibility) {
11. mSession = s;
12. mClient = c;
13. mToken = token;
14. mAttrs.copyFrom(a);
15. mViewVisibility = viewVisibility;
16. DeathRecipient deathRecipient = new DeathRecipient();
17. mAlpha = a.alpha;

public class WindowManagerService extends IWindowManager.Stub

implements Watchdog.Monitor {

......

private final class WindowState implements WindowManagerPolicy.WindowState {

......

WindowState(Session s, IWindow c, WindowToken token,

WindowState attachedWindow, WindowManager.LayoutParams a,

int viewVisibility) {

mSession = s;

mClient = c;

mToken = token;

mAttrs.copyFrom(a);

mViewVisibility = viewVisibility;

DeathRecipient deathRecipient = new DeathRecipient();

mAlpha = a.alpha;

        这段代码初始化WindowState类的以下六个成员变量：

**-mSession**：指向一个类型为Session的Binder本地对象，使用参数s来初始化，表示当前所创建的WindowState对象是属于哪一个应用程序进程的。

**-mClient**：指向一个实现了IWindow接口的Binder代理对象，它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，使用参数c来初始化，通过它可以与运行在应用程序进程这一侧的Activity组件进行通信。

**-mToken**：指向一个WindowToken对象，使用参数token来初始化，通过它就可以知道唯一地标识一个窗口。

**-mAttrs**：指向一个WindowManager.LayoutParams对象，使用参数a来初始化，通过它就可以知道当前当前所创建的WindowState对象所描述的窗口的布局参数。

**-mViewVisibility**：这是一个整型变量，使用参数viewVisibility来初始化，表示当前所创建的WindowState对象所描述的窗口视图的可见性。

**-mAlpha**：这是一个浮点数，使用参数a所描述的一WindowManager.LayoutParams对象的成员变量alpha来初始化，表示当前所创建的WindowState对象所描述的窗口的Alpha通道。

        此外，这段代码还创建了一个类型为DeathRecipient的死亡通知接收者deathRecipient，它是用来监控参数c所引用的一个类型为W的Binder本地对象的生命周期的。当这个Binder本地对象死亡的时候，就意味着当前所创建的WindowState对象所描述的窗口所在的应用程序进程已经退出了。接下来的这段代码就是用来注册死亡通知接收者deathRecipient的，如下所示：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. try {
2. c.asBinder().linkToDeath(deathRecipient, 0);
3. } catch (RemoteException e) {
4. mDeathRecipient = null;
5. mAttachedWindow = null;
6. mLayoutAttached = false;
7. mIsImWindow = false;
8. mIsWallpaper = false;
9. mIsFloatingLayer = false;
10. mBaseLayer = 0;
11. mSubLayer = 0;
12. return;
13. }
14. mDeathRecipient = deathRecipient;

try {

c.asBinder().linkToDeath(deathRecipient, 0);

} catch (RemoteException e) {

mDeathRecipient = null;

mAttachedWindow = null;

mLayoutAttached = false;

mIsImWindow = false;

mIsWallpaper = false;

mIsFloatingLayer = false;

mBaseLayer = 0;

mSubLayer = 0;

return;

}

mDeathRecipient = deathRecipient;

        注册完成之后，前面所创建的死亡通知接收者deathRecipient就会保存在WindowState类的成员变量mDeathRecipientk 。

        我们继续向前看代码：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. if ((mAttrs.type >= FIRST\_SUB\_WINDOW &&
2. mAttrs.type <= LAST\_SUB\_WINDOW)) {
3. // The multiplier here is to reserve space for multiple
4. // windows in the same type layer.
5. mBaseLayer = mPolicy.windowTypeToLayerLw(
6. attachedWindow.mAttrs.type) \* TYPE\_LAYER\_MULTIPLIER
7. + TYPE\_LAYER\_OFFSET;
8. mSubLayer = mPolicy.subWindowTypeToLayerLw(a.type);
9. mAttachedWindow = attachedWindow;
10. mAttachedWindow.mChildWindows.add(this);
11. mLayoutAttached = mAttrs.type !=
12. WindowManager.LayoutParams.TYPE\_APPLICATION\_ATTACHED\_DIALOG;
13. mIsImWindow = attachedWindow.mAttrs.type == TYPE\_INPUT\_METHOD
14. || attachedWindow.mAttrs.type == TYPE\_INPUT\_METHOD\_DIALOG;
15. mIsWallpaper = attachedWindow.mAttrs.type == TYPE\_WALLPAPER;
16. mIsFloatingLayer = mIsImWindow || mIsWallpaper;
17. } else {
18. // The multiplier here is to reserve space for multiple
19. // windows in the same type layer.
20. mBaseLayer = mPolicy.windowTypeToLayerLw(a.type)
21. \* TYPE\_LAYER\_MULTIPLIER
22. + TYPE\_LAYER\_OFFSET;
23. mSubLayer = 0;
24. mAttachedWindow = null;
25. mLayoutAttached = false;
26. mIsImWindow = mAttrs.type == TYPE\_INPUT\_METHOD
27. || mAttrs.type == TYPE\_INPUT\_METHOD\_DIALOG;
28. mIsWallpaper = mAttrs.type == TYPE\_WALLPAPER;
29. mIsFloatingLayer = mIsImWindow || mIsWallpaper;
30. }

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

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

// The multiplier here is to reserve space for multiple

// windows in the same type layer.

mBaseLayer = mPolicy.windowTypeToLayerLw(

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

+ TYPE\_LAYER\_OFFSET;

mSubLayer = mPolicy.subWindowTypeToLayerLw(a.type);

mAttachedWindow = attachedWindow;

mAttachedWindow.mChildWindows.add(this);

mLayoutAttached = mAttrs.type !=

WindowManager.LayoutParams.TYPE\_APPLICATION\_ATTACHED\_DIALOG;

mIsImWindow = attachedWindow.mAttrs.type == TYPE\_INPUT\_METHOD

|| attachedWindow.mAttrs.type == TYPE\_INPUT\_METHOD\_DIALOG;

mIsWallpaper = attachedWindow.mAttrs.type == TYPE\_WALLPAPER;

mIsFloatingLayer = mIsImWindow || mIsWallpaper;

} else {

// The multiplier here is to reserve space for multiple

// windows in the same type layer.

mBaseLayer = mPolicy.windowTypeToLayerLw(a.type)

\* TYPE\_LAYER\_MULTIPLIER

+ TYPE\_LAYER\_OFFSET;

mSubLayer = 0;

mAttachedWindow = null;

mLayoutAttached = false;

mIsImWindow = mAttrs.type == TYPE\_INPUT\_METHOD

|| mAttrs.type == TYPE\_INPUT\_METHOD\_DIALOG;

mIsWallpaper = mAttrs.type == TYPE\_WALLPAPER;

mIsFloatingLayer = mIsImWindow || mIsWallpaper;

}

        这段代码初始化WindowState类的以下七个成员变量：

**-mBaseLayer**：这是一个整型变量，用来描述一个窗口的基础Z轴位置值，这个值是与窗口类型相关的。对于子窗口来说，它的值由父窗口的基础Z轴位置值乘以常量TYPE\_LAYER\_MULTIPLIER再加固定偏移量TYPE\_LAYER\_OFFSET得到；对于非子窗口来说，它的值就是由窗口的类型来决定的。一个窗口的基础Z轴位置值是通过调用WindowManagerService类的成员变量mPolicy所描述的一个窗口管理策略器的成员函数windowTypeToLayerLw来获得的，而窗口管理策略器的成员函数windowTypeToLayerLw主要是根据窗口的类型来决定它的基础Z轴位置值的。

**-mSubLayer**：这是一个整型变量，用来描述一个子窗口相对其父窗口的Z轴偏移值。对于非子窗口来说，这个值固定为0；对于子窗口来说，这个值是由WindowManagerService类的成员变量mPolicy所描述的一个窗口管理策略器的成员函数subWindowTypeToLayerLw来获得的，而窗口管理策略器的成员函数subWindowTypeToLayerLw主要是根据子窗口的类型来决定它相对其父窗口的Z轴偏移值的。

**-mAttachedWindow**：指向一个WindowState对象，用来描述一个子窗口的父窗口。对于非子窗口来说，这个值固定为null；对于子窗口来说，这个值使用参数attachedWindow来初始化。如果当前所创建的WindowState对象所描述的窗口是一个子窗口，那么这个子窗口还会被添加用来描述它的父窗口的一WindowState对象的成员变量mChildWindows所描述的一个子窗口列表中去。

**-mLayoutAttached**：这是一个布尔变量，用来描述一个子窗口的视图是否是嵌入在父窗口的视图里面的。对于非子窗口来说，这个值固定为false；对于子窗口来说，这个值只有子窗口的类型是非对话框时，它的值才会等于true，否则都等于false。

**-mIsImWindow**：这是一个布尔变量，表示当前所创建的WindowState对象所描述的窗口是否是一个输入法窗口或者一个输入法对话框。

**-mIsWallpaper**：这是一个布尔变量，表示当前所创建的WindowState对象所描述的窗口是否是一个壁纸窗口。

**-mIsFloatingLayer**：这是一个布尔变量，表示当前所创建的WindowState对象所描述的窗口是否是一个浮动窗口。当一个窗口是一个输入法窗口、输入法对话框口或者壁纸窗口时，它才是一个浮动窗口。

        我们继续向前看代码：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. WindowState appWin = this;
2. while (appWin.mAttachedWindow != null) {
3. appWin = mAttachedWindow;
4. }
5. WindowToken appToken = appWin.mToken;
6. while (appToken.appWindowToken == null) {
7. WindowToken parent = mTokenMap.get(appToken.token);
8. if (parent == null || appToken == parent) {
9. break;
10. }
11. appToken = parent;
12. }
13. mRootToken = appToken;
14. mAppToken = appToken.appWindowToken;

WindowState appWin = this;

while (appWin.mAttachedWindow != null) {

appWin = mAttachedWindow;

}

WindowToken appToken = appWin.mToken;

while (appToken.appWindowToken == null) {

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

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

break;

}

appToken = parent;

}

mRootToken = appToken;

mAppToken = appToken.appWindowToken;

        这段代码主要用来初始化成员变量mRootToken和mAppToken。

        成员变量mRootToken的类型为WindowToken，用来描述当前所创建的WindowState对象所描述的窗口的根窗口。如果当前所创建的WindowState对象所描述的窗口是一个子窗口，那么就先找到它的父窗口，然后再找到它的父窗口所属的应用程序窗口，即Activity组件窗口，这时候找到的Activity组件窗口就是一个根窗口。如果当前所创建的WindowState对象所描述的窗口是一个子窗口，但是它不属于任何一个应用程序窗口的，那么它的父窗口就是一个根窗口。如果当前所创建的WindowState对象所描述的窗口不是一个子窗口，并且它也不属于一个应用程序窗口的，那么它本身就是一个根窗口。

        成员变量mAppToken的类型为AppWindowToken，只有当成员变量mRootToken所描述的一个根窗口是一个应用程序窗口时，它的值才不等于null。

        我们继续向前看最后一段代码：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. mSurface = null;
2. mRequestedWidth = 0;
3. mRequestedHeight = 0;
4. mLastRequestedWidth = 0;
5. mLastRequestedHeight = 0;
6. mXOffset = 0;
7. mYOffset = 0;
8. mLayer = 0;
9. mAnimLayer = 0;
10. mLastLayer = 0;
11. }
13. ......
14. }
16. ......
17. }

mSurface = null;

mRequestedWidth = 0;

mRequestedHeight = 0;

mLastRequestedWidth = 0;

mLastRequestedHeight = 0;

mXOffset = 0;

mYOffset = 0;

mLayer = 0;

mAnimLayer = 0;

mLastLayer = 0;

}

......

}

......

}

       这段代码将以下十个成员变量的值设置为null或者0：

**-mSurface**：指向一个mSurface对象，用来描述窗口的绘图表面。

**-mRequestedWidth**：这是一个整型变量，用来描述应用程序进程所请求的窗口宽度。

**-mRequestedHeight**：这是一个整型变量，用来描述应用程序进程所请求的窗口高度。

**-mLastRequestedWidth**：这是一个整型变量，用来描述应用程序进程上一次所请求的窗口宽度。

**-mLastRequestedHeight**：这是一个整型变量，用来描述应用程序进程上一次所请求的窗口高度。

**-mXOffset**：这是一个整型变量，用来描述壁纸窗口相对屏幕在X轴上的偏移量，对其它类型的窗口为说，这个值等于0。

**-mYOffset**：这是一个整型变量，用来描述壁纸窗口相对屏幕在Y轴上的偏移量，对其它类型的窗口为说，这个值等于0。

**-mLayer**：这是一个整型变量，用来描述窗口的Z轴位置值。

**-mAnimLayer**：这是一个整型变量，用来描述窗口的Z轴位置值，它的值可能等于mLayer的值，但是在以下四种情况中不相等。当一个窗口关联有一个目标窗口的时候，那么它的值就等于mLayer的值加上目标窗口指定的一个动画层调整值；当一个窗口的根窗口是一个应用程序窗口时，那么它的值就等于mLayer的值加上根窗口指定的一个动画层调整值；当一个窗口是一个输入法窗口时，那么它的值就等于mLayer的值加上系统设置的输入法动画层调整值；当一个窗口是壁纸窗口时，那么它的值就等于mLayer的值加上系统设置的壁纸动画层调整值。

**-mLastLayer**：这是一个整型变量，用描述窗口上一次所使用的mAnimLayer的值。

       至此，我们就分析完成WindowState的构造函数的实现了，返回到前面的Step 3中，即WindowManagerService类的成员函数addWindow中，接下来就会继续调用前面所创建的一个WindowState对象的成员函数attach来创建一个关联的SurfaceSession对象，以便可以用来和SurfaceFlinger服务通信。

       Step 5. WindowState.attach

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public class WindowManagerService extends IWindowManager.Stub
2. implements Watchdog.Monitor {
3. ......
5. private final class WindowState implements WindowManagerPolicy.WindowState {
6. final Session mSession;
7. ......
9. void attach() {
10. ......
11. mSession.windowAddedLocked();
12. }
14. ......
15. }
17. ......
18. }

public class WindowManagerService extends IWindowManager.Stub

implements Watchdog.Monitor {

......

private final class WindowState implements WindowManagerPolicy.WindowState {

final Session mSession;

......

void attach() {

......

mSession.windowAddedLocked();

}

......

}

......

}

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

        WindowState类的成员变量mSession指向的是一个Session对象，这个Session对象就是用来连接应用程序进程和WindowManagerService服务，WindowState类的成员函数attach调用它的成员函数windowAddedLocked来检查是否需要为当前正在请求增加窗口的应用程序进程创建一个SurfaceSession对象。

        接下来，我们继续分析Session类的成员函数windowAddedLocked的实现。

        Step 6. Session.windowAddedLocked

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public class WindowManagerService extends IWindowManager.Stub
2. implements Watchdog.Monitor {
3. ......
5. /\*\*
6. \* All currently active sessions with clients.
7. \*/
8. final HashSet<Session> mSessions = new HashSet<Session>();
9. ......
11. private final class Session extends IWindowSession.Stub
12. implements IBinder.DeathRecipient {
13. ......
15. SurfaceSession mSurfaceSession;
16. int mNumWindow = 0;
17. ......
19. void windowAddedLocked() {
20. if (mSurfaceSession == null) {
21. ......
22. mSurfaceSession = new SurfaceSession();
23. ......
24. mSessions.add(this);
25. }
26. mNumWindow++;
27. }
29. ......
30. }
32. ......
33. }

public class WindowManagerService extends IWindowManager.Stub

implements Watchdog.Monitor {

......

/\*\*

\* All currently active sessions with clients.

\*/

final HashSet<Session> mSessions = new HashSet<Session>();

......

private final class Session extends IWindowSession.Stub

implements IBinder.DeathRecipient {

......

SurfaceSession mSurfaceSession;

int mNumWindow = 0;

......

void windowAddedLocked() {

if (mSurfaceSession == null) {

......

mSurfaceSession = new SurfaceSession();

......

mSessions.add(this);

}

mNumWindow++;

}

......

}

......

}

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

        Session类的成员变量mSurfaceSession指向的是一个SurfaceSession对象，这个SurfaceSession对象是WindowManagerService服务用来与SurfaceSession服务建立连接的。Session类的成员函数windowAddedLocked首先检查这个成员变量的值是否等于null。如果等于null的话，那么就说明WindowManagerService服务尚未为当前正在请求增加窗口的应用程序进程创建一个用来连接SurfaceSession服务的SurfaceSession对象，因此，Session类的成员函数windowAddedLocked就会创建一个SurfaceSession对象，并且保存在成员变量mSurfaceSession中，并且将正在处理的Session对象添加WindowManagerService类的成员变量mSession所描述的一个HashSet中去，表示WindowManagerService服务又多了一个激活的应用程序进程连接。

        Session类的另外一个成员变量mNumWindow是一个整型变量，用来描述当前正在处理的Session对象内部包含有多少个窗口，即运行在引用了当前正在处理的Session对象的应用程序进程的内部的窗口的数量。每当运行在应用程序进程中的窗口销毁时，该应用程序进程就会通知WindowManagerService服务移除用来描述该窗口状态的一个WindowState对象，并且通知它所引用的Session对象减少其成员变量mNumWindow的值。当一个Session对象的成员变量mNumWindow的值减少为0时，就说明该Session对象所描述的应用程序进程连接已经不需要了，因此，该Session对象就可以杀掉其成员变量mSurfaceSession所描述的一个SurfaceSession对象，以便可以断开和SurfaceSession服务的连接。

        接下来，我们就继续分析SurfaceSession类的构造函数的实现，以便可以了解一个SurfaceSession对象是如何与SurfaceSession服务建立连接的。

        Step 7.new SurfaceSession

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. public class SurfaceSession {
2. /\*\* Create a new connection with the surface flinger. \*/
3. public SurfaceSession() {
4. init();
5. }
7. ......
9. private native void init();
10. ......
12. private int mClient;
13. }

public class SurfaceSession {

/\*\* Create a new connection with the surface flinger. \*/

public SurfaceSession() {

init();

}

......

private native void init();

......

private int mClient;

}

       这个函数定义在文件frameworks/base/core/java/android/view/SurfaceSession.java中。

       SurfaceSession类的构造函数的实现很简单，它只是调用了另外一个成员函数init来执行一些初始化操作，其实就是用来初始化SurfaceSession类的成员变量mClient。

       SurfaceSession类的成员函数init是一个JNI方法，它是由C++层的函数SurfaceSession\_init来实现的，如下所示：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. static void SurfaceSession\_init(JNIEnv\* env, jobject clazz)
2. {
3. sp<SurfaceComposerClient> client = new SurfaceComposerClient;
4. client->incStrong(clazz);
5. env->SetIntField(clazz, sso.client, (**int**)client.get());
6. }

static void SurfaceSession\_init(JNIEnv\* env, jobject clazz)

{

sp<SurfaceComposerClient> client = new SurfaceComposerClient;

client->incStrong(clazz);

env->SetIntField(clazz, sso.client, (int)client.get());

}

       这个函数定义在文件frameworks/base/core/jni/android\_view\_Surface.cpp中。

       在分析函数SurfaceSession\_init的实现之前，我们首先看看全局变量sso的定义，如下所示：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. struct sso\_t {
2. jfieldID client;
3. };
4. static sso\_t sso;

struct sso\_t {

jfieldID client;

};

static sso\_t sso;

       它是一个类型为sso\_t的结构体变量，它的成员变量client描述的是SurfaceSession类的成员变量mClient在SurfaceSession类中的偏移量：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8275938)

1. void nativeClassInit(JNIEnv\* env, jclass clazz)
2. {
3. ......
5. jclass surfaceSession = env->FindClass("android/view/SurfaceSession");
6. sso.client = env->GetFieldID(surfaceSession, "mClient", "I");
8. ......
9. }

void nativeClassInit(JNIEnv\* env, jclass clazz)

{

......

jclass surfaceSession = env->FindClass("android/view/SurfaceSession");

sso.client = env->GetFieldID(surfaceSession, "mClient", "I");

......

}

       回到函数SurfaceSession\_init中，它首先创建一个SurfaceComposerClient对象client，接着再增加这个SurfaceComposerClient对象的强引用计数，因为再接下来会将这个SurfaceComposerClient对象的地址值保存在参数clazz所描述的一个SurfaceSession对象的成员变量mClient中，这相当于是参数clazz所描述的一个SurfaceSession对象引用了刚才所创建的SurfaceComposerClient对象client。

       在前面[Android应用程序与SurfaceFlinger服务的关系概述和学习计划](http://blog.csdn.net/luoshengyang/article/details/7846923)这一系列的文章中，我们已经分析过SurfaceComposerClient类的作用了，这主要就是用来在应用程序进程和SurfaceFlinger服务之间建立连接的，以便应用程序进程可以为运行在它里面的应用程序窗口请求SurfaceComposerClient创建绘制表面（Surface）的操作等。

       这样，每一个Java层的SurfaceSession对象在C++层就都有一个对应的SurfaceComposerClient对象，因此，Java层的应用程序就可以通过SurfaceSession类来和SurfaceFlinger服务建立连接。

       至此，我们就分析完成一个WindowState对象的创建过程了，通过这个过程我们就可以知道，每一个Activity组件窗口在WindowManagerService服务内部都有一个对应的WindowState对象，用来描述它的窗口状态。

       至此，我们也分析完成Android应用程序窗口与WindowManagerService服务的连接过程了。从这个连接过程以及前面[Android应用程序窗口（Activity）的窗口对象（Window）的创建过程分析](http://blog.csdn.net/luoshengyang/article/details/8223770)和[Android应用程序窗口（Activity）的视图对象（View）的创建过程分析](http://blog.csdn.net/luoshengyang/article/details/8245546)这两篇文章，我们就可以知道，为了实现一个Activity组件的UI，无论是应用程序进程，还是WindowManagerService，都做了大量的工作，例如，应用程序进程为它创建一个窗口（Window）对象、一个视图（View）对象、一个ViewRoot对象、一个W对象，WindowManagerService服务为它创建一个AppWindowToken对象和一个WindowState对象。此外，WindowManagerService服务还为一个Activity组件所运行在的应用程序进程创建了一个Session对象。理解这些对象的实现以及作用对我们了解Android应用程序窗口的实现框架以及WindowManagerService服务的实现原理都是非常重要的。

       虽然到目前为止，我们已经为Android应用程序窗口创建了很多对象，但是我们仍然还有一个最重要的对象还没有创建，那就是Android应用程序窗口的绘图表面，即用来渲染UI的Surface还没有创建。从前面[Android应用程序与SurfaceFlinger服务的关系概述和学习计划](http://blog.csdn.net/luoshengyang/article/details/7846923)这一系列的文章可以知道，这个Surface是要请求SurfaceFlinger服务来创建的，因此，在接下来的一篇文章中，我们就将继续分析Android应用程序窗口的绘图表面（Surface）的创建过程，敬请关注！