 http://blog.csdn.net/luoshengyang/article/details/8170307

前面我们学习了SurfaceFlinger服务的实现原理。有了这个基础之后，从本文开始，我们就可以分析Android系统在Java层的UI实现了。我们知道，在Android应用程序的四大组件中，只有Activity组件与UI相关，它描述的是应用程序窗口，因此，我们就通过它的UI实现来分析Android系统在Java层的UI实现。本文主要是对Activity组件的UI实现作简要介绍以及制定学习计划。

        Activity组件的UI实现需要与WindowManagerService服务和SurfaceFlinger服务进行交互。从前面[Android应用程序键盘（Keyboard）消息处理机制分析](http://blog.csdn.net/luoshengyang/article/details/6882903)一文可以知道，Activity组件在启动完成后，会通过一个类型为Session的Binder对象来请求WindowManagerService为它创建一个类型为WindowState的对象，用来描述它的窗口状态。此外，[从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)这一系列的文章又可以知道，Android应用程序会通过一个类型为Client的Binder对象来请求SurfaceFlinger服务为它创建一个类型为Layer的对象，用来描述它的窗口数据。

        从[Android应用程序请求SurfaceFlinger服务创建Surface的过程分析](http://blog.csdn.net/luoshengyang/article/details/7884628)一文又可以知道，SurfaceFlinger服务为Android应用程序创建一个类型为Layer的对象之后，会返回一个类型为SurfaceLayer的Binder对象给Android应用程序，这样Android应用程序就可以通过这个Binder对象来请求SurfaceFlinger服务来分配图形缓冲区。

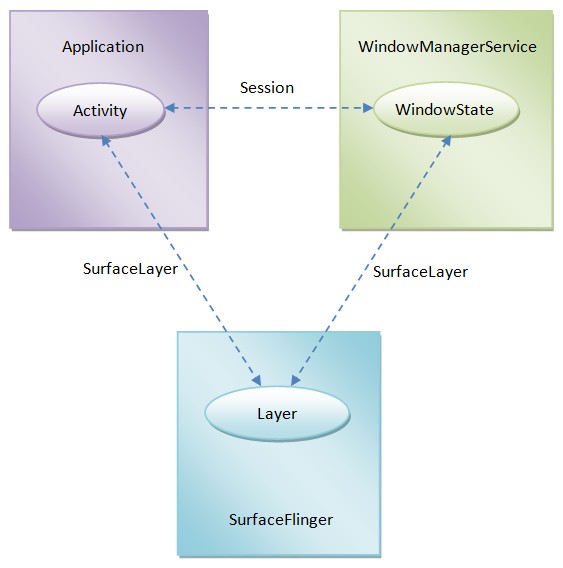
        综合上述信息，我们就可以得到Activity组件与WindowManagerService服务和SurfaceFlinger服务的交互模型，如图1所示：

图1 Activity组件与WindowManagerService服务和SurfaceFlinger服务的交互模型

        事实上，用来关联Activity组件和Layer对象的SurfaceLayer对象并不是由Android应用程序请求SurfaceFlinger服务来创建的，而是由WindowManagerService服务请求SurfaceFlinger服务来创建的。WindowManagerService服务得到这个SurfaceLayer对象之后，再将它的一个代理对象返回给在Android应用程序这一侧的Activity组件。这样，Activity组件和WindowManagerService服务就可以通过同一个SurfaceLayer对象来操作在SurfaceFlinger服务这一侧的Layer对象，而操作Layer对象的目的就是为了修改Activity组件的UI。

        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)和[Android系统Surface机制的SurfaceFlinger服务简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/8010977)这两个系列的文章中，我们已经分析在SurfaceFlinger服务这一侧的Layer类和SurfaceLayer类的实现了。在现在的这一系列文章中，我们主要分析在Android应用程序这一侧的Activity组件与UI相关的类的实现，以及在WindowManagerService服务这一侧的WindowState类的实现。

        我们首先看Activity组件的实现，如图2所示：

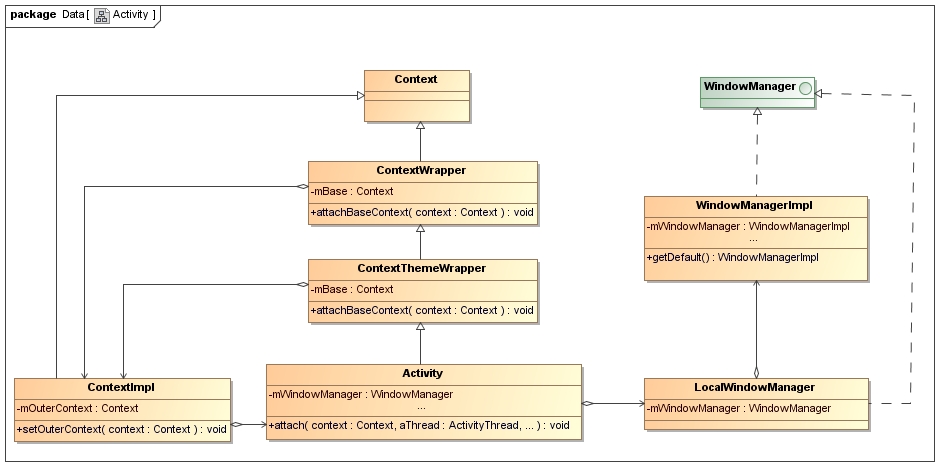


图2 Activity组件的类关系图

        Activity类是从ContextThemeWrapper类继承下来的，而ContextThemeWrapper类又是从ContextWrapper类继承下来的，最后ContextWrapper类又继承了Context类。

        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，Activity组件在启动的过程中，系统会为它创建一个ContextImpl对象，用来描述它的运行上下文环境。这个ContextImpl对象首先是通过调用Acitivity类的成员函数attach传递到Acticity组件内部，接着再依次通过调用父类ContextThemeWrapper和ContextWrapper的成员函数attachBaseContext来分别保存在它们的成员变量mBase中。因此，ContextThemeWrapper和ContextWrapper类的成员变量mBase指向的实际上是一个ContextImpl对象。

       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)一文还可以知道，系统为一个正在启动的Activity组件创建了一个ContextImpl对象之后，还会调用这个ContextImpl对象的成员函数setOuterContext来将正在启动的Activity组件保存在其成员变量mOuterContext中。这样，一个Activity组件就可以通过其父类ContextThemeWrapper或者ContextWrapper的成员变量mBase来访问用来描述它的运行上下文环境的一个ContextImpl对象，同时，一个ContextImpl对象也可以通过它的成员变量mOuterContext来访问它的宿主Activity组件。

      Activity类还有另外一个类型为WindowManager的成员变量mWindowManager，它实际上指向的一个LocalWindowManager对象。LocalWindowManager类是用来管理应用程序窗口的，例如，用来维护应用程序窗口内部的视图（View）。LocalWindowManager类有一个类型为WindowManager的成员变量mWindowManager，它实际上指向的是一个WindowManagerImpl对象。系统通过调用WindowManagerImpl类的静态成员函数getDefault来获得一个WindowManagerImpl对象，然后保存在LocalWindowManager类的成员变量mWindowManager中。这样，LocalWindowManager类就可以通过WindowManagerImpl类来真正实现管理应用程序窗口的功能。

      从上面的分析中，我们还看不出的一个Activity组件的窗口是如何描述的。为了弄清楚这个问题，我们继续分析Activity类的另外一个成员变量mWindow，如图3所示：

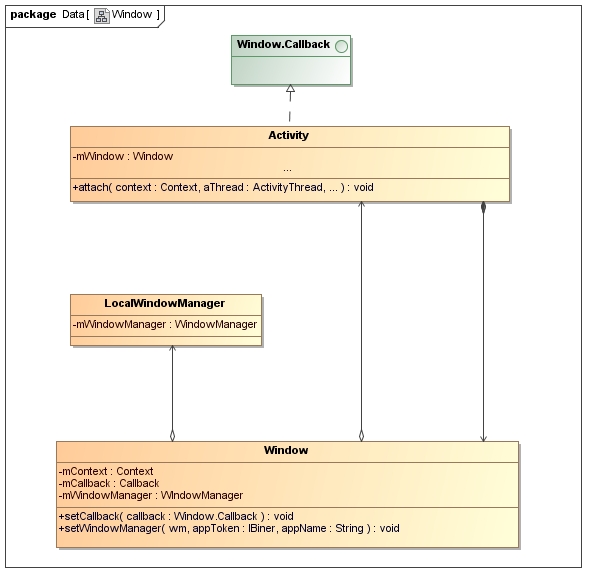


图3 Window类的实现

        Activity类的成员变量mWindow的类型为Window，它用来描述一个应用程序窗口。这样，通过这个成员变量，每一个Activity组件就都会有一个对应的Window对象，即一个对应的应用程序窗口。

        Window类有一个类型为Context的成员变量mContext。这个成员变量指向的是一个Activity对象。当系统为一个Activity组件创建一个对应的Window对象时，就会将这个Activity组件的Context接口保存在这个对应的Window对象的成员变量mContext中。这样，一个Window对象就可以通过它的成员变量mContext来访问它所描述的Activity组件的资源。

        Window类还有一个类型为Window.Callback的成员变量mCallback。这个成员变量和成员变量mContext一样，都是指向同一个Activity对象，因为Activity类是实现了Window.Callback接口的。当系统为一个Activity组件创建一个对应的Window对象时，就会将这个Activity组件所实现的Window.Callback接口通过Window类的成员函数setCallback保存在对应的Window对象的成员变量mCallback。这样，一个Window对象就可以通过它的成员变量mCallback来将一些事件交给与它所对应的Activity组件来处理，例如，将接收的键盘事件交给对应的Activity组件来处理。

        最后，Window类还有一个类型为WindowManager的成员变量mWindowManager。这个成员变量指向的是一个LocalWindowManager对象。前面提到，Activity组件的成员变量mWindowManager指向的也是一个LocalWindowManager对象。系统在启动一个Activity组件的过程中，会通过Window类的成员函数setWindowManager来将保存在它的成员变量mWindowManager中的一个LocalWindowManager对象也保存在对应的Window对象的成员变量mWindowManager。这样，一个Activity组件以及它所对应的Window对象就可以使用同一个LocalWindowManager对象来管理它们所描述的UI了。

       事实上，Activity类的成员变量mWindow指向的并不是一个Window对象，而是一个PhoneWindow对象。也就是说，一个Activity组件的UI是使用一个PhoneWindow对象来描述的。

       Activity类的成员变量mWindow所指向的一个PhoneWindow对象是通过调用PolicyManager类的静态成员函数makeNewWindow来创建的。PolicyManager类的实现如图4所示：

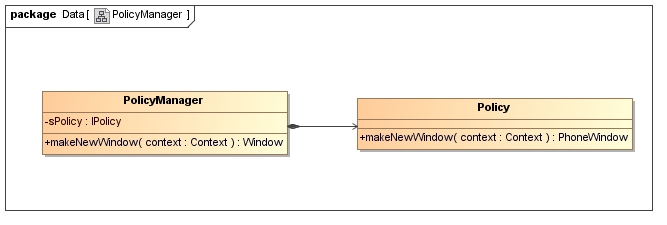


图4 PolicyManager类的实现

        PolicyManager类有一个类型为IPolicy的静态成员变量sPolicy，它实际指向的是一个Policy对象。Policy类实现了IPolicy接口的成员函数makeNewWindow，而PolicyManager类就是通过这个成员函数来为一个Activity组件创建一个PhoneWindow对象的。

       PhoneWindow类继承了Window类，因此，它的对象可以保存Activity类的成员变量mWindow中。PhoneWindow类的实现如图5所示：

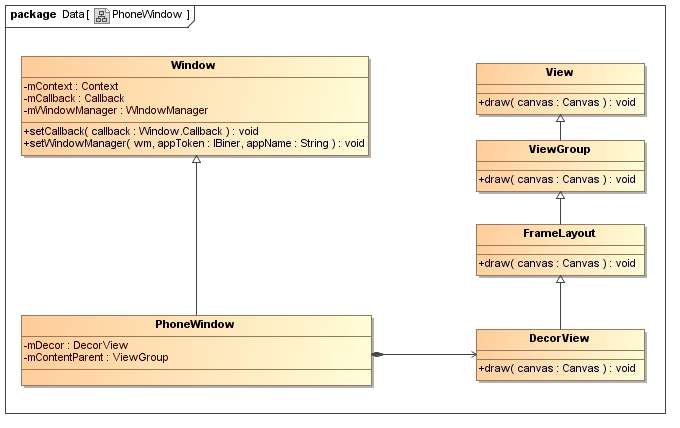


图5 PhoneWindow类的实现

        PhoneWindow类有两个重要的成员变量mDecor和mContentParent，它们的类型分别DecorView和ViewGroup。其中，成员变量mDecor是用描述自己的窗口视图，而成员变量mContentParent用来描述视图内容的父窗口。

        DecorView类继承了FrameLayout类，而FrameLayout类又继承了ViewGroup类，最后ViewGroup类又继承了View类。View类有一个成员函数draw，它是用来绘制应用程序窗口的UI的。DecorView类、FrameLayout类和ViewGroup类都重写了父类的成员函数draw，这样，它们就都可以定制自己的UI。

        DecorView类所描述的应用程序窗口视图是否需要重新绘制是由另外一个类ViewRoot来控制的。系统在启动一个Activity组件的过程中，会为这个Activity组件创建一个ViewRoot对象，同时还会将前面为这个Activity组件所创建的一个PhoneWindow对象的成员变量mDecor所描述的一个视图（DecorView）保存在这个ViewRoot对象的成员变量mView中。这样，这个ViewRoot对象就可以通过调用它的成员变量mView的所描述的一个DecorView的成员函数draw来绘制一个Acitivity组件的UI了。ViewRoot类的作用是非常大的，它除了用来控制一个Acitivity组件的UI绘制之外，还负责接收Acitivity组件的IO输入事件，例如，键盘事件，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。

        ViewRoot类的实现如图6所示：

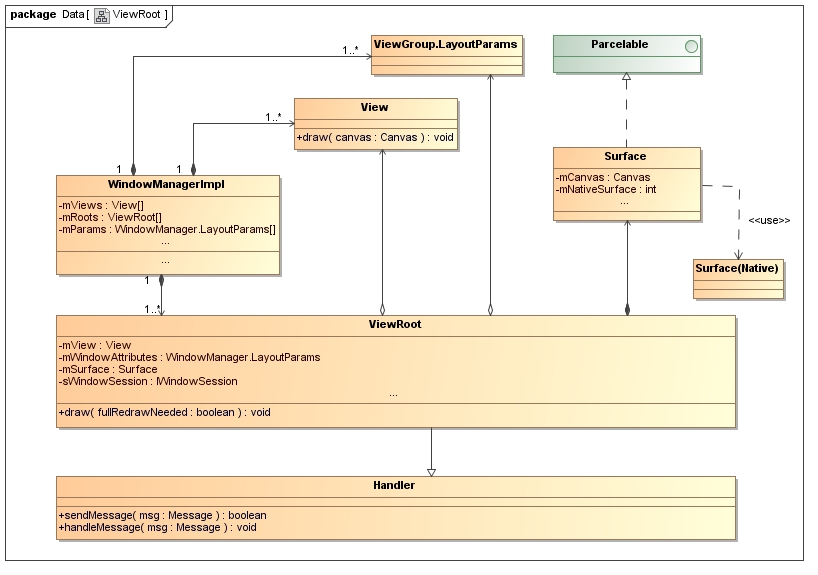


图6 ViewRoot类的实现

        ViewRoot类是从Handler类继承下来的。从前面[Android应用程序消息处理机制（Looper、Handler）分析](http://blog.csdn.net/luoshengyang/article/details/6817933)一文可以知道，从Handler类继承下来的子类可以调用父类的成员函数sendMessage来向指定的线程的消息队列发送消息，以及在自己重写的成员函数handleMessage中处理该消息。 ViewRoot类在两种情况需要经常应用程序进程的主线程的消息队列发送消息。

        第一种情况是当ViewRoot类从系统输入管理器InputManager接收到键盘、触摸屏等输入事件时，它就会把这些输入事件封装成一个消息，并且发送到应用程序进程的主线程的消息队列中去进一步处理，这样就可以保证键盘、触摸屏等输入事件可以在应用程序进程的主线程中进行处理。这一点可以参考前面A[ndroid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。

        第二种情况是当ViewRoot类需要重新绘制与它所关联的一个Activity组件的UI时，它就会将这个绘制UI的操作封装成一个消息，并且发送到应用程序进程的主线程的消息队列中去进一步处理，这样同样可以保证绘制UI的操作可以在应用程序进程的主线程中执行。

        每一个ViewRoot对象都有一个类型为View的成员变量mView，它指向了一个DecorView对象。这个DecorView对象是从哪里来的呢？前面提到，每一个Activity组件都有一个对应的ViewRoot对象以及一个对应的PhoneWindow对象，这个DecorView对象就是来自于这个对应的PhoneWindow对象的成员变量mDecor。也就是说，与同一个Activity组件对应的ViewRoot对象和PhoneWindow对象分别通过各自的成员变量mView和mDecor引用了共一个DecorView对象。

        每一个ViewRoot对象都有一个类型为WindowManager.LayoutParams的成员变量mWindowAttributes，它指向了一个ViewGroup.LayoutParams对象，用来描述与该ViewRoot对象对应的一个Activity组件的UI布局信息。

        从上面的描述就可以知道，每一个Activity组件都有一个对应的ViewRoot对象、View对象以及WindowManager.LayoutParams对象。这三个对象的对应关系是由WindowManagerImpl类来维护的。具体来说，就是由WindowManagerImpl类的成员变量mRoots、mViews和mParams所描述的三个数组来维护的。例如，假设一个应用程序进程运行有两个Activity组件，那么WindowManagerImpl类的成员变量mRoots、mViews和mParams所描述的三个数组的大小就等于2，其中，mRoots[0]、mViews[0]和mParams[0]对应于第一个启动的Activity组件，而mRoots[1]、mViews[1]和mParams[1]对应于第二个启动的Activity组件。

        每一个ViewRoot对象都有一个类型为Surface的成员变量mSurface，它指向了一个Java层的Surface对象。这个Java层的Surface对象通过它的成员变量mNativeSurface与一个C++层的Surface对象。这个C++层的Surface对象就是我们在[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)这一系列文章中所分析的Surface类的实例了。这个Surface类是用来在Android应用程序进程这一侧描述应用程序窗口的。从前面[Android应用程序请求SurfaceFlinger服务创建Surface的过程分析](http://blog.csdn.net/luoshengyang/article/details/7884628)一文可以知道，在C++层中，每一个Surface对象都有一个对应的SurfaceControl对象。这个对应的SurfaceControl对象是用来设置应用程序窗口的属性，例如，设置大小、位置等属性。

        但是，与ViewRoot类的成员变量mSurface所对应的在C++层的Surface对象并没有一个对应的SurfaceControl对象，这是因为ViewRoot类并不需要设置应用程序窗口的属性，它需要做的只是往应用程序窗口的图形缓冲区填充UI数据，即它需要设置的只是应用程序窗口的纹理。应用程序窗口的纹理保存在Java层的Surface类的成员变量mCanvas所描述一个画布（Canvas）中，即通过这个画布可以访问到应用程序窗口的图形缓冲区。当ViewRoot类需要重新绘制与它对应的Activity组件的UI时，它就会调用它的成员函数draw来执行这个绘制的操作。ViewRoot类的成员函数draw首先通过获得保存它的成员变量mSurface内部的一块画布，然后再将这个画布传递给它的成员变量mView所描述的一个View对象的成员函数draw。View类的成员函数draw得到了这块画布之后，就可以随心所欲地上面绘制应用程序窗口的纹理了。这些纹理的绘制工作是通过Skia图形库API来进行的。

        那么，应用程序窗口的属性是由谁来管理的呢？这是由WindowManagerService服务来管理的。前面提到，在Android应用程序这一侧的Activity组件是由WindowManagerService服务来为它请求SurfaceFlinger服务创建一个Layer对象以及一个SurfaceLayer对象的。这个SurfaceLayer对象创建完成之后，WindowManagerService服务就会将它封装在一个Java层的Surface对象中，以后就可以通过这个Java层的Surface对象来请求SurfaceFlinger服务设置一个对应的应用程序窗口的属性。

        由于Java层的Surface对象实现了Parcelable接口，因此，WindowManagerService服务在为一个Activity组件请求SurfaceFlinger服务创建一个Layer对象以及一个SurfaceLayer对象之后，就可以将得到的Java层的Surface对象跨进程地返回给该Activity组件。Activity组件得到这个Surface对象之后，再使用保存在里面的SurfaceLayer对象来初始化与它所对应的一个ViewRoot对象的成员变量mSurface所描述的一个Java层的Surface对象。

        那么，WindowManagerService服务又是什么时候会为一个Activity组件请求SurfaceFlinger服务创建一个Layer对象以及一个SurfaceLayer对象呢？ViewRoot类有一个类型为IWindowSession的静态成员变量sWindowSession，它指向的实际上是一个实现了IWindowSession接口的Binder对象。这个Binder对象的类型为Session，运行在WindowManagerService服务这一侧。当一个Activity组件的UI第一次要被绘制之前，它所运行在的应用程序进程就会通过ViewRoot类的静态成员变量sWindowSession来向WindowManagerService服务发送一个请求。WindowManagerService服务接收到这个请求之后，再请求SurfaceFlinger服务为这个Activity组件创建一个Layer对象以及一个SurfaceLayer对象。这样，这个Activity组件的UI才能真正地绘制在屏幕中。

        至此，我们就简要分析完成了在Android应用程序这一侧的Activity组件与UI相关的类的实现，接下来我们继续分析在WindowManagerService服务这一侧的WindowState类的实现。

        WindowState类的实现如图7所示：

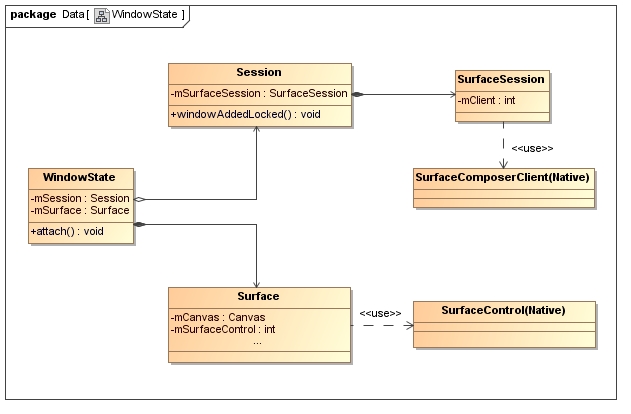


图7 WindowState类的实现

        在Android应用程序这一侧，每一个Activity组件在WindowManagerService服务这一侧都有一个对应的WindowState对象，用来描述Activity组件的窗口状态。WindowState类有两个重要的成员变量mSession和mSurface，它们的类型分别为SurfaceSession和Surface。接下来，我们就描述这两个成员变量的作用。

        前面提到，Activity组件是在启动完成之后，请求WindowManagerService服务为它创建一个WindowState对象的。创建完成这个WindowState对象之后，WindowManagerService服务再调用它的成员函数attach来为它附加一个SurfaceSession对象。WindowState类的成员函数attach又是通过调用它的成员变量mSession所描述的一个Session对象的成员函数windowAddedLocked来附加一个SurfaceSession对象的。

        Session类有一个类型为SurfaceSession的成员变量mSurfaceSession。当WindowState类的成员函数attach调用Session类的成员函数windowAddedLocked来为一个WindowState对象附加一个SurfaceSession对象的时候，后者首先会检查它的成员变量mSurfaceSession是否已经指向了一个SurfaceSession对象。如果如果指向了的话，那么Session类的成员函数windowAddedLocked就什么也不用做，否则的话，Session类的成员函数windowAddedLocked就会创建一个SurfaceSession对象，并且保存在它的成员变量mSurfaceSession中。

        SurfaceSession类有一个类型为int的成员变量mClient，它保存的是一个C++层的SurfaceComposerClient对象的地址，即每一个Java层的SurfaceSession对象在C++层都有一个对应的SurfaceComposerClient对象。当一个SurfaceSession对象创建的时候，与它所关联的SurfaceComposerClient对象也会同时被创建。从前面[Android应用程序与SurfaceFlinger服务的连接过程分析](http://blog.csdn.net/luoshengyang/article/details/7857163)一文可以知道，SurfaceComposerClient类用来描述Android应用程序进程与SurfaceFlinger服务之间的一个连接，即每一个与UI相关的Android应用程序进程都有一个SurfaceComposerClient对象。

        读者可能会觉得奇怪，既然SurfaceComposerClient是用来描述Android应用程序进程与SurfaceFlinger服务的连接的，那么为什么WindowManagerService服务会在内部创建SurfaceComposerClient对象呢？由于WindowManagerService需要请求SurfaceFlinger服务来设置Android应用程序窗口的属性，例如，设置应用程序窗口的位置、大小等，因此，它就需要为每一个Android应用程序进程创建一个SurfaceComposerClient对象连接到SurfaceFlinger服务中去，以便可以和SurfaceFlinger服务进行通信。

        从上面的描述我们就可以知道，在WindowManagerService服务中，每一个Android应用程序进程都对应有一个SurfaceComposerClient对象。由于每一个SurfaceComposerClient对象都关联有一个SurfaceSession对象，因此，我们又可以推断出每一个Android应用程序进程在WindowManagerService服务中都对应有一个SurfaceSession对象。由于每一个SurfaceSession对象所属的Session对象是一个Binder本地对象，并且它的Binder代理对象是保存在Android应用程序进程这一侧的ViewRoot类的静态成员变量sWindowSession中，因此，我们又可以推断出每一个Android应用程序进程在WindowManagerService服务都有一个对应的Session对象。综合起来就是，每一个Android应用程序进程在WindowManagerService服务这一侧对应有一个Session对象、一个SurfaceSession对象以及一个SurfaceComposerClient对象。由于每一个Android应用程序进程都可以运行若干个Activity组件，因此，我们又可以说，Activity组件与WindowServiceManager服务这一侧的Session对象、SurfaceSession对象以及SurfaceComposerClient对象是多对一的关系。

        介绍了WindowState类的成员变量mSession之后，我们接着介绍另外一个成员变量mSurface，它的类型为Surface，前面我们已经介绍过Surface类在Android应用程序进程这一侧的作用了，接下来我们就介绍它在WindowManagerService这一侧的作用。

       前面提到，WindowManagerService服务会在内部为每一个应用程序窗口，即每一个Activity组件，创建一个SurfaceLayer对象，这个SurfaceLayer对象是封装成一个Java层的Surface对象中的。在Java层的Surface类中，有一个类型为int的成员变量mSurfaceControl，它保存的是在C++层的一个SurfaceControl对象的地址值，即在WindowManagerService服务这一侧，每一个Java层的Surface对象在 C++层都有一个对应的SurfaceControl对象。这里我们强调是在WindowManagerService服务这一侧，是因为在前面提到，在Android应用程序这一侧，每一个Activity组件所对应的Java层的Surface对象在C++层是没有一个对应的SurfaceControl对象，而只是对应有一个C++层的Surface对象。通过C++层的SurfaceControl对象可以设置应用程序窗口的属性，而通过C++层的Surface对象则可以设置应用程序窗口的图形缓冲区，即设置应用程序窗口的纹理，因此，我们就可以知道应用程序窗口的属性是由WindowManagerService服务来设置的，而应用程序窗口的纹理是由它所在的进程负责设置的。

       至此，我们就简要地介绍了Android应用程序窗口的实现框架了。上面所介绍的类及其交互关系可能会比较模糊，不易理解。不要紧，接下来我们会通过一系列的文章来弄清楚它们的来龙去脉：

       1. [Android应用程序窗口的运行上下文的创建过程，即创建ContextImpl的创建过程](http://blog.csdn.net/luoshengyang/article/details/8201936)；

       2. [Android应用程序窗口的创建过程，即PhoneWindow的创建过程](http://blog.csdn.net/luoshengyang/article/details/8223770)；

       3. [Android应用程序窗口的视图的创建过程，即DecorView的创建过程](http://blog.csdn.net/luoshengyang/article/details/8245546)；

       4. [Android应用程序窗口与WindowManagerService服务的连接过程，即WindowState的创建过程](http://blog.csdn.net/luoshengyang/article/details/8275938)；

       5. [Android应用程序窗口与SurfaceFlinger服务的连接过程，即Surface的创建过程；](http://blog.csdn.net/luoshengyang/article/details/8303098)

       6. [Android应用程序窗口的绘制过程，即Surface的渲染过程；](http://blog.csdn.net/luoshengyang/article/details/8372924)

       学习了这些文章，我们就可以掌握Android应用程序窗口的实现框架了。掌握了Android应用程序窗口的实现框架之后，我们就可以再进一步去详细地学习Android应用程序窗口的渲染过程。敬请关注！