 http://blog.csdn.net/luoshengyang/article/details/8303098

在前文中，我们分析了应用程序窗口连接到WindowManagerService服务的过程。在这个过程中，WindowManagerService服务会为应用程序窗口创建过一个到SurfaceFlinger服务的连接。有了这个连接之后，WindowManagerService服务就可以为应用程序窗口创建绘图表面了，以便可以用来渲染窗口的UI。在本文中，我们就详细分析应用程序窗口的绘图表面的创建过程。

        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)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)这两个系列的文章可以知道，每一个在C++层实现的应用程序窗口都需要有一个绘图表面，然后才可以将自己的UI表现出来。这个绘图表面是需要由应用程序进程请求SurfaceFlinger服务来创建的，在SurfaceFlinger服务内部使用一个Layer对象来描述，同时，SurfaceFlinger服务会返回一个实现了ISurface接口的Binder本地对象给应用程序进程，于是，应用程序进程就可以获得一个实现了ISurface接口的Binder代理对象。有了这个实现了ISurface接口的Binder代理对象之后，在C++层实现的应用程序窗口就可以请求SurfaceFlinger服务分配图形缓冲区以及渲染已经填充好UI数据的图形缓冲区了。

        对于在Java层实现的Android应用程序窗口来说，它也需要请求SurfaceFlinger服务为它创建绘图表面，这个绘图表面使用一个Surface对象来描述。由于在Java层实现的Android应用程序窗口还要接受WindowManagerService服务管理，因此，它的绘图表面的创建流程就会比在C++层实现的应用程序窗口复杂一些。具体来说，就是在在Java层实现的Android应用程序窗口的绘图表面是通过两个Surface对象来描述，一个是在应用程序进程这一侧创建的，另一个是在WindowManagerService服务这一侧创建的，它们对应于SurfaceFlinger服务这一侧的同一个Layer对象，如图1所示：

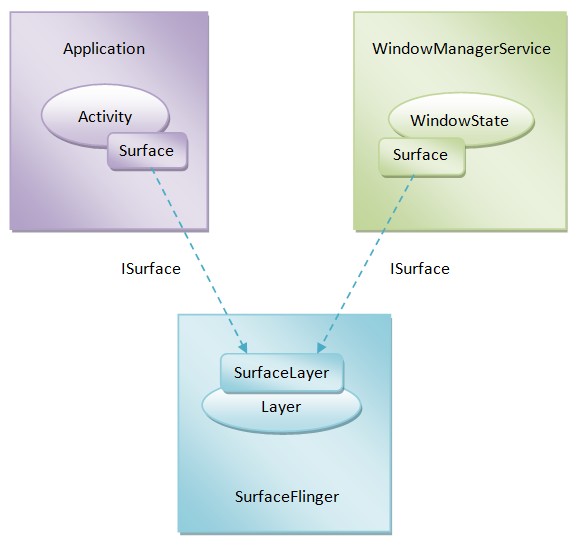


图1 应用程序窗口的绘图表面的模型图

        在应用程序进程这一侧，每一个应用程序窗口，即每一个Activity组件，都有一个关联的Surface对象，这个Surface对象是保在在一个关联的ViewRoot对象的成员变量mSurface中的，如图2所示：

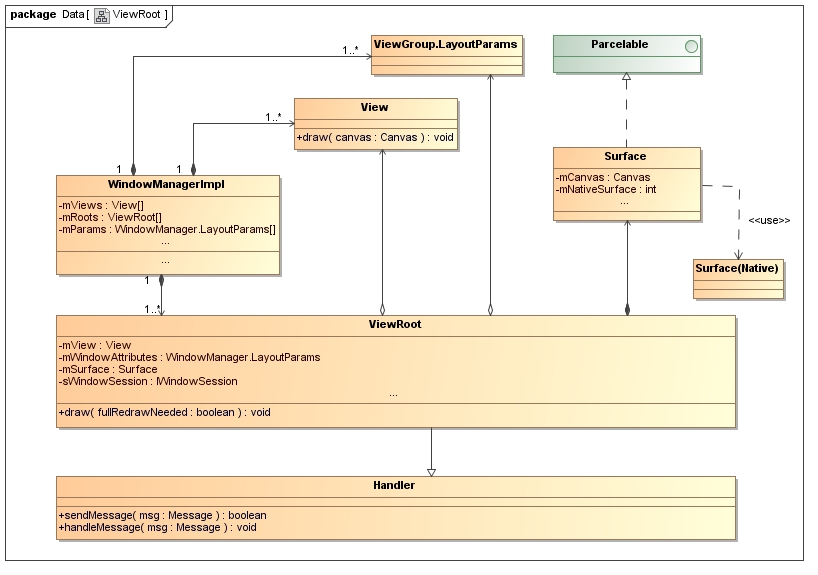


图2 应用程序窗口在应用程序进程这一侧的Surface的实现

        图2的类关系图的详细描述可以参考前面[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6，这里我们只关注Surface类的实现。在应用程序进程这一侧，每一个Java层的Surface对都对应有一个C++层的Surface对象，并且后者的地址值保存在前者的成员变量mNativeSurface中。C++层的Surface类的实现以及作用可以参考前面[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)这个系列的文章。

        在WindowManagerService服务这一侧，每一个应用程序窗口，即每一个Activity组件，都有一个对应的WindowState对象，这个WindowState对象的成员变量mSurface同样是指向了一个Surface对象，如图3所示：

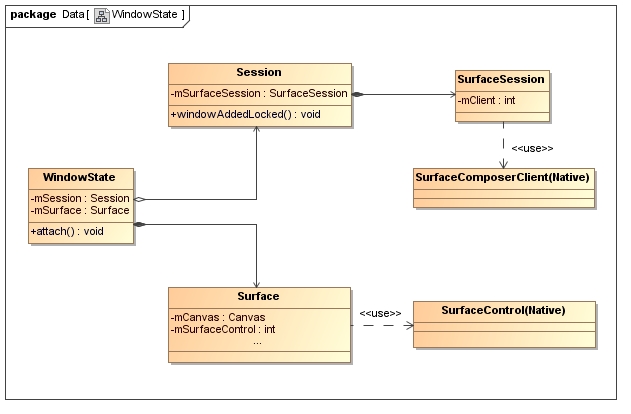


图3 应用程序窗口在WindowManagerService服务这一侧的Surface的实现

        图3的类关系图的详细描述可以参考前面[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，这里我们同样只关注Surface类的实现。在WindowManagerService服务这一侧，每一个Java层的Surface对都对应有一个C++层的SurfaceControl对象，并且后者的地址值保存在前者的成员变量mSurfaceControl中。C++层的SurfaceControl类的实现以及作用同样可以参考前面[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)这个系列的文章。

       一个应用程序窗口分别位于应用程序进程和WindowManagerService服务中的两个Surface对象有什么区别呢？虽然它们都是用来操作位于SurfaceFlinger服务中的同一个Layer对象的，不过，它们的操作方式却不一样。具体来说，就是位于应用程序进程这一侧的Surface对象负责绘制应用程序窗口的UI，即往应用程序窗口的图形缓冲区填充UI数据，而位于WindowManagerService服务这一侧的Surface对象负责设置应用程序窗口的属性，例如位置、大小等属性。这两种不同的操作方式分别是通过C++层的Surface对象和SurfaceControl对象来完成的，因此，位于应用程序进程和WindowManagerService服务中的两个Surface对象的用法是有区别的。之所以会有这样的区别，是因为绘制应用程序窗口是独立的，由应用程序进程来完即可，而设置应用程序窗口的属性却需要全局考虑，即需要由WindowManagerService服务来统筹安排，例如，一个应用程序窗口的Z轴坐标大小要考虑它到的窗口类型以及它与系统中的其它窗口的关系。

        说到这里，另外一个问题又来了，由于一个应用程序窗口对应有两个Surface对象，那么它们是如何创建出来的呢？简单地说，就是按照以下步骤来创建：

       1. 应用程序进程请求WindowManagerService服务为一个应用程序窗口创建一个Surface对象；

       2. WindowManagerService服务请求SurfaceFlinger服务创建一个Layer对象，并且获得一个ISurface接口；

       3. WindowManagerService服务将获得的ISurface接口保存在其内部的一个Surface对象中，并且将该ISurface接口返回给应用程序进程；

       4. 应用程序进程得到WindowManagerService服务返回的ISurface接口之后，再将其封装成其内部的另外一个Surface对象中。

       那么应用程序窗口的绘图表面又是什么时候创建的呢？一般是在不存在的时候就创建，因为应用程序窗口在运行的过程中，它的绘图表面会根据需要来销毁以及重新创建的，例如，应用程序窗口在第一次显示的时候，就会请求WindowManagerService服务为其创建绘制表面。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，当一个应用程序窗口被激活并且它的视图对象创建完成之后，应用程序进程就会调用与其所关联的一个ViewRoot对象的成员函数requestLayout来请求对其UI进行布局以及显示。由于这时候应用程序窗口的绘图表面尚未创建，因此，ViewRoot类的成员函数requestLayout就会请求WindowManagerService服务来创建绘图表面。接下来，我们就从ViewRoot类的成员函数requestLayout开始，分析应用程序窗口的绘图表面的创建过程，如图4所示：

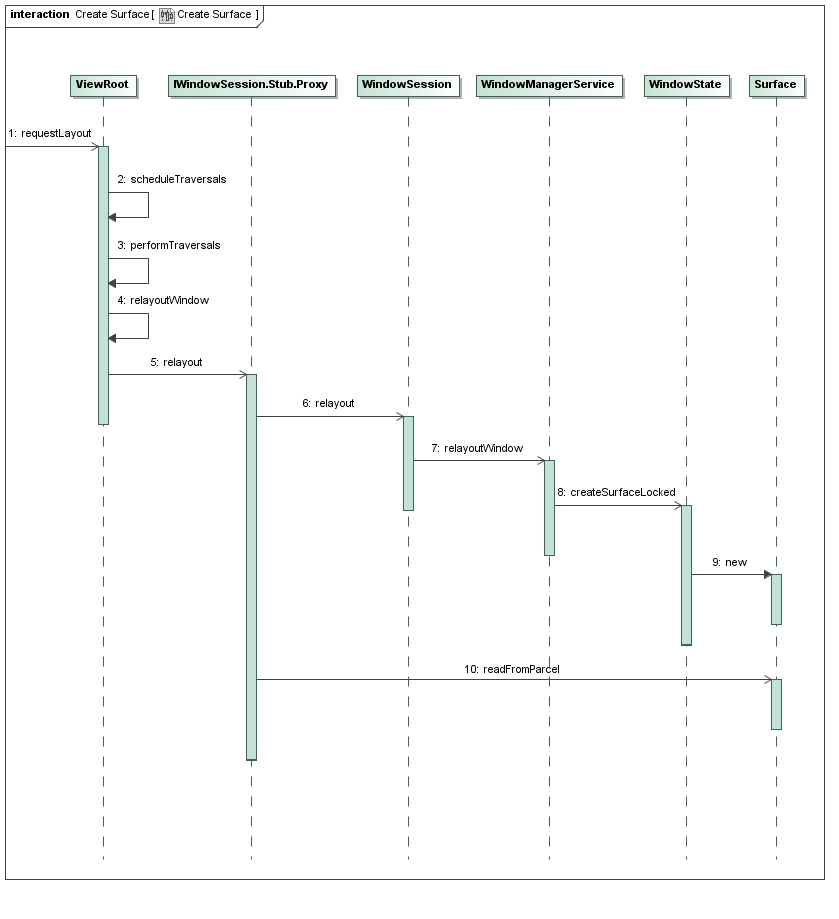


图4 应用程序窗口的绘图表面的创建过程

        这个过程可以分为10个步骤，接下来我们就详细分析每一个步骤。

        Step 1. ViewRoot.requestLayout

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. public final class ViewRoot extends Handler implements ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. boolean mLayoutRequested;
6. ......
8. public void requestLayout() {
9. checkThread();
10. mLayoutRequested = true;
11. scheduleTraversals();
12. }
14. ......
15. }

public final class ViewRoot extends Handler implements ViewParent,

View.AttachInfo.Callbacks {

......

boolean mLayoutRequested;

......

public void requestLayout() {

checkThread();

mLayoutRequested = true;

scheduleTraversals();

}

......

}

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

        ViewRoot类的成员函数requestLayout首先调用另外一个成员函数checkThread来检查当前线程是否就是创建当前正在处理的ViewRoot对象的线程。如果不是的话，那么ViewRoot类的成员函数checkThread就会抛出一个异常出来。ViewRoot类是从Handler类继承下来的，用来处理应用程序窗口的UI布局和渲染等消息。由于这些消息都是与Ui相关的，因此它们就需要在UI线程中处理，这样我们就可以推断出当前正在处理的ViewRoot对象是要应用程序进程的UI线程中创建的。进一步地，我们就可以推断出ViewRoot类的成员函数checkThread实际上就是用来检查当前线程是否是应用程序进程的UI线程，如果不是的话，它就会抛出一个异常出来。

        通过了上述检查之后，ViewRoot类的成员函数requestLayout首先将其成员变量mLayoutRequested的值设置为true，表示应用程序进程的UI线程正在被请求执行一个UI布局操作，接着再调用另外一个成员函数scheduleTraversals来继续执行UI布局的操作。

        Step 2. ViewRoot.scheduleTraversals

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. public final class ViewRoot extends Handler implements ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. boolean mTraversalScheduled;
6. ......
8. public void scheduleTraversals() {
9. if (!mTraversalScheduled) {
10. mTraversalScheduled = true;
11. sendEmptyMessage(DO\_TRAVERSAL);
12. }
13. }
15. ......
16. }

public final class ViewRoot extends Handler implements ViewParent,

View.AttachInfo.Callbacks {

......

boolean mTraversalScheduled;

......

public void scheduleTraversals() {

if (!mTraversalScheduled) {

mTraversalScheduled = true;

sendEmptyMessage(DO\_TRAVERSAL);

}

}

......

}

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

        ViewRoot类的成员变量mTraversalScheduled用来表示应用程序进程的UI线程是否已经调度了一个DO\_TRAVERSAL消息。如果已经调度了的话，它的值就会等于true。在这种情况下， ViewRoot类的成员函数scheduleTraversals就什么也不做，否则的话，它就会首先将成员变量mTraversalScheduled的值设置为true，然后再调用从父类Handler继承下来的成员函数sendEmptyMessage来往应用程序进程的UI线程发送一个DO\_TRAVERSAL消息。

        这个类型为DO\_TRAVERSAL的消息是由ViewRoot类的成员函数performTraversals来处理的，因此，接下来我们就继续分析ViewRoot类的成员函数performTraversals的实现。

        Step 3. ViewRoot.performTraversals

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. public final class ViewRoot extends Handler implements ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. View mView;
6. ......
8. boolean mLayoutRequested;
9. boolean mFirst;
10. ......
11. boolean mFullRedrawNeeded;
12. ......
14. private final Surface mSurface = new Surface();
15. ......
17. private void performTraversals() {
18. ......
20. final View host = mView;
21. ......
23. mTraversalScheduled = false;
24. ......
25. boolean fullRedrawNeeded = mFullRedrawNeeded;
26. boolean newSurface = false;
27. ......
29. if (mLayoutRequested) {
30. ......
32. host.measure(childWidthMeasureSpec, childHeightMeasureSpec);
34. .......
35. }
37. ......
39. int relayoutResult = 0;
40. if (mFirst || windowShouldResize || insetsChanged
41. || viewVisibilityChanged || params != null) {
42. ......
44. boolean hadSurface = mSurface.isValid();
45. try {
46. ......
48. relayoutResult = relayoutWindow(params, viewVisibility, insetsPending);
49. ......
51. if (!hadSurface) {
52. if (mSurface.isValid()) {
53. ......
54. newSurface = true;
55. fullRedrawNeeded = true;
56. ......
57. }
58. }
59. ......
60. } catch (RemoteException e) {
61. }
63. ......
64. }
66. final boolean didLayout = mLayoutRequested;
67. ......
69. if (didLayout) {
70. mLayoutRequested = false;
71. ......
73. host.layout(0, 0, host.mMeasuredWidth, host.mMeasuredHeight);
75. ......
76. }
78. ......
80. mFirst = false;
81. ......
83. boolean cancelDraw = attachInfo.mTreeObserver.dispatchOnPreDraw();
85. if (!cancelDraw && !newSurface) {
86. mFullRedrawNeeded = false;
87. draw(fullRedrawNeeded);
89. ......
90. } else {
91. ......
93. // Try again
94. scheduleTraversals();
95. }
96. }
98. ......
99. }

public final class ViewRoot extends Handler implements ViewParent,

View.AttachInfo.Callbacks {

......

View mView;

......

boolean mLayoutRequested;

boolean mFirst;

......

boolean mFullRedrawNeeded;

......

private final Surface mSurface = new Surface();

......

private void performTraversals() {

......

final View host = mView;

......

mTraversalScheduled = false;

......

boolean fullRedrawNeeded = mFullRedrawNeeded;

boolean newSurface = false;

......

if (mLayoutRequested) {

......

host.measure(childWidthMeasureSpec, childHeightMeasureSpec);

.......

}

......

int relayoutResult = 0;

if (mFirst || windowShouldResize || insetsChanged

|| viewVisibilityChanged || params != null) {

......

boolean hadSurface = mSurface.isValid();

try {

......

relayoutResult = relayoutWindow(params, viewVisibility, insetsPending);

......

if (!hadSurface) {

if (mSurface.isValid()) {

......

newSurface = true;

fullRedrawNeeded = true;

......

}

}

......

} catch (RemoteException e) {

}

......

}

final boolean didLayout = mLayoutRequested;

......

if (didLayout) {

mLayoutRequested = false;

......

host.layout(0, 0, host.mMeasuredWidth, host.mMeasuredHeight);

......

}

......

mFirst = false;

......

boolean cancelDraw = attachInfo.mTreeObserver.dispatchOnPreDraw();

if (!cancelDraw && !newSurface) {

mFullRedrawNeeded = false;

draw(fullRedrawNeeded);

......

} else {

......

// Try again

scheduleTraversals();

}

}

......

}

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

        ViewRoot类的成员函数performTraversals的实现是相当复杂的，这里我们分析它的实现框架，在以后的文章中，我们再详细分析它的实现细节。

        在分析ViewRoot类的成员函数performTraversals的实现框架之前，我们首先了解ViewRoot类的以下五个成员变量：

**--mView**：它的类型为View，但它实际上指向的是一个DecorView对象，用来描述应用程序窗口的顶级视图，这一点可以参考前面[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。

**--mLayoutRequested**：这是一个布尔变量，用来描述应用程序进程的UI线程是否需要正在被请求执行一个UI布局操作。

**--mFirst**：这是一个布尔变量，用来描述应用程序进程的UI线程是否第一次处理一个应用程序窗口的UI。

**--mFullRedrawNeeded**：这是一个布尔变量，用来描述应用程序进程的UI线程是否需要将一个应用程序窗口的全部区域都重新绘制。

**--mSurface**：它指向一个Java层的Surface对象，用来描述一个应用程序窗口的绘图表面。

        注意，成员变量mSurface所指向的Surface对象在创建的时候，还没有在C++层有一个关联的Surface对象，因此，这时候它描述的就是一个无效的绘图表面。另外，这个Surface对象在应用程序窗口运行的过程中，也可能被销毁，因此，这时候它描述的绘图表面也会变得无效。在上述两种情况中，我们都需要请求WindowManagerService服务来为当前正在处理的应用程序窗口创建有一个有效的绘图表面，以便可以在上面渲染UI。这个创建绘图表面的过程正是本文所要关心的。（重点）

        理解了上述五个成员变量之后，我们就可以分析ViewRoot类的成员函数performTraversals的实现框架了，如下所示：

        1. 将成员变量mView和mFullRedrawNeeded的值分别保存在本地变量host和fullRedrawNeeded中，并且将成员变量mTraversalScheduled的值设置为false，表示应用程序进程的UI线程中的DO\_TRAVERSAL消息已经被处理了。

        2. 本地变量newSurface用来描述当前正在处理的应用程序窗口在本轮的DO\_TRAVERSAL消息处理中是否新创建了一个绘图表面，它的初始值为false。

        3. 如果成员变量mLayoutRequested的值等于true，那么就表示应用程序进程的UI线程正在被请求对当前正在处理的应用程序窗口执行一个UI布局操作，因此，这时候就会调用本地变量host所描述的一个顶层视图对象的成员函数measure来测量位于各个层次的UI控件的大小。

        4. 如果当前正在处理的应用程序窗口的UI是第一次被处理，即成员变量mFirst的值等于true，或者当前正在处理的应用程序窗口的大小发生了变化，即本地变量windowShouldResize的值等于true，或者当前正在处理的应用程序窗口的边衬发生了变化，即本地变量insetsChanged的值等于true，或者正在处理的应用程序窗口的可见性发生了变化，即本地变量viewVisibilityChanged的值等于true，或者正在处理的应用程序窗口的UI布局参数发生了变化，即本地变量params指向了一个WindowManager.LayoutParams对象，那么应用程序进程的UI线程就会调用另外一个成员函数relayoutWindow来请求WindowManagerService服务重新布局系统中的所有窗口。WindowManagerService服务在重新布局系统中的所有窗口的过程中，如果发现当前正在处理的应用程序窗口尚未具有一个有效的绘图表面，那么就会为它创建一个有效的绘图表面，这一点是我们在本文中所要关注的。

        5. 应用程序进程的UI线程在调用ViewRoot类的成员函数relayoutWindow来请求WindowManagerService服务重新布局系统中的所有窗口之前，会调用成员变量mSurface所指向的一个Surface对象的成员函数isValid来判断它描述的是否是一个有效的绘图表面，并且将结果保存在本地变量hadSurface中。

        6. 应用程序进程的UI线程在调用ViewRoot类的成员函数relayoutWindow来请求WindowManagerService服务重新布局系统中的所有窗口之后，又会继续调用成员变量mSurface所指向的一个Surface对象的成员函数isValid来判断它描述的是否是一个有效的绘图表面。如果这时候成员变量mSurface所指向的一个Surface对象描述的是一个有效的绘图表面，并且本地变量hadSurface的值等于false，那么就说明WindowManagerService服务为当前正在处理的应用程序窗口新创建了一个有效的绘图表面，于是就会将本地变量newSurface和fullRedrawNeeded的值均修改为true。

        7. 应用程序进程的UI线程再次判断mLayoutRequested的值是否等于true。如果等于的话，那么就说明需要对当前正在处理的应用程序窗口的UI进行重新布局，这是通过调用本地变量host所描述的一个顶层视图对象的成员函数layout来实现的。在对当前正在处理的应用程序窗口的UI进行重新布局之前，应用程序进程的UI线程会将成员变量mLayoutRequested的值设置为false，表示之前所请求的一个UI布局操作已经得到处理了。

        8. 应用程序进程的UI线程接下来就要开始对当前正在处理的应用程序窗口的UI进行重新绘制了，不过在重绘之前，会先询问一下那些注册到当前正在处理的应用程序窗口中的Tree Observer，即调用它们的成员函数dispatchOnPreDraw，看看它们是否需要取消接下来的重绘操作，这个询问结果保存在本地变量cancelDraw中。

        9. 如果本地变量cancelDraw的值等于false，并且本地变量newSurface的值也等于false，那么就说明注册到当前正在处理的应用程序窗口中的Tree Observer不要求取消当前的这次重绘操作，并且当前正在处理的应用程序窗口也没有获得一个新的绘图表面。在这种情况下，应用程序进程的UI线程就会调用ViewRoot类的成员函数draw来对当前正在处理的应用程序窗口的UI进行重绘。在重绘之前，还会将ViewRoot类的成员变量mFullRedrawNeeded的值重置为false。

       10. 如果本地变量cancelDraw的值等于true，或者本地变量newSurface的值等于true，那么就说明注册到当前正在处理的应用程序窗口中的Tree Observer要求取消当前的这次重绘操作，或者当前正在处理的应用程序窗口获得了一个新的绘图表面。在这两种情况下，应用程序进程的UI线程就不能对当前正在处理的应用程序窗口的UI进行重绘了，而是要等到下一个DO\_TRAVERSAL消息到来的时候，再进行重绘，以便使得当前正在处理的应用程序窗口的各项参数可以得到重新设置。下一个DO\_TRAVERSAL消息需要马上被调度，因此，应用程序进程的UI线程就会重新执行ViewRoot类的成员函数scheduleTraversals。

       这样，我们就分析完成ViewRoot类的成员函数performTraversals的实现框架了，接下来我们就继续分析ViewRoot类的成员函数relayoutWindow的实现，以便可以看到当前正在处理的应用程序窗口的绘图表面是如何创建的。

       Step 4. ViewRoot.relayoutWindow

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. public final class ViewRoot extends Handler implements ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. static IWindowSession sWindowSession;
6. ......
8. final W mWindow;
9. ......
11. private final Surface mSurface = new Surface();
12. ......
14. private int relayoutWindow(WindowManager.LayoutParams params, int viewVisibility,
15. boolean insetsPending) throws RemoteException {
16. ......
18. int relayoutResult = sWindowSession.relayout(
19. mWindow, params,
20. (int) (mView.mMeasuredWidth \* appScale + 0.5f),
21. (int) (mView.mMeasuredHeight \* appScale + 0.5f),
22. viewVisibility, insetsPending, mWinFrame,
23. mPendingContentInsets, mPendingVisibleInsets,
24. mPendingConfiguration, mSurface);
25. ......
27. return relayoutResult;
28. }
30. ......
31. }

public final class ViewRoot extends Handler implements ViewParent,

View.AttachInfo.Callbacks {

......

static IWindowSession sWindowSession;

......

final W mWindow;

......

private final Surface mSurface = new Surface();

......

private int relayoutWindow(WindowManager.LayoutParams params, int viewVisibility,

boolean insetsPending) throws RemoteException {

......

int relayoutResult = sWindowSession.relayout(

mWindow, params,

(int) (mView.mMeasuredWidth \* appScale + 0.5f),

(int) (mView.mMeasuredHeight \* appScale + 0.5f),

viewVisibility, insetsPending, mWinFrame,

mPendingContentInsets, mPendingVisibleInsets,

mPendingConfiguration, mSurface);

......

return relayoutResult;

}

......

}

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

       ViewRoot类的成员函数relayoutWindow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relayout来请求WindowManagerService服务对成员变量mWindow所描述的一个应用程序窗口的UI进行重新布局，同时，还会将成员变量mSurface所描述的一个Surface对象传递给WindowManagerService服务，以便WindowManagerService服务可以根据需要来重新创建一个绘图表面给成员变量mWindow所描述的一个应用程序窗口使用。

       实现了IWindowSession接口的Binder代理对象是由IWindowSession.Stub.Proxy类来描述的，接下来我们就继续分析它的成员函数relayout的实现。

       Step 5. IWindowSession.Stub.Proxy.relayout

       IWindowSession接口是使用AIDL语言来描述的，如下所示：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. interface IWindowSession {
2. ......
4. int relayout(IWindow window, in WindowManager.LayoutParams attrs,
5. int requestedWidth, int requestedHeight, int viewVisibility,
6. boolean insetsPending, out Rect outFrame, out Rect outContentInsets,
7. out Rect outVisibleInsets, out Configuration outConfig,
8. out Surface outSurface);
10. ......
11. }

interface IWindowSession {

......

int relayout(IWindow window, in WindowManager.LayoutParams attrs,

int requestedWidth, int requestedHeight, int viewVisibility,

boolean insetsPending, out Rect outFrame, out Rect outContentInsets,

out Rect outVisibleInsets, out Configuration outConfig,

out Surface outSurface);

......

}

        这个接口定义在frameworks/base/core/java/android/view/IWindowSession.aidl文件中。

        使用AIDL语言来描述的IWindowSession接口被编译后，就会生成一个使用Java语言来描述的IWindowSession.Stub.Proxy类，它的成员函数relayout的实现如下所示：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. public interface IWindowSession extends android.os.IInterface
2. {
3. ......
5. public static abstract class Stub extends android.os.Binder implements android.view.IWindowSession
6. {
7. ......
9. private static class Proxy implements android.view.IWindowSession
10. {
11. private android.os.IBinder mRemote;
12. ......
14. public int relayout(android.view.IWindow window, android.view.WindowManager.LayoutParams attrs,
15. int requestedWidth, int requestedHeight, int viewVisibility, boolean insetsPending,
16. android.graphics.Rect outFrame,
17. android.graphics.Rect outContentInsets,
18. android.graphics.Rect outVisibleInsets,
19. android.content.res.Configuration outConfig,
20. android.view.Surface outSurface) throws android.os.RemoteException
21. {
22. android.os.Parcel \_data = android.os.Parcel.obtain();
23. android.os.Parcel \_reply = android.os.Parcel.obtain();
25. int \_result;
26. try {
27. \_data.writeInterfaceToken(DESCRIPTOR);
28. \_data.writeStrongBinder((((window!=null))?(window.asBinder()):(null)));
30. if ((attrs!=null)) {
31. \_data.writeInt(1);
32. attrs.writeToParcel(\_data, 0);
33. }
34. else {
35. \_data.writeInt(0);
36. }
38. \_data.writeInt(requestedWidth);
39. \_data.writeInt(requestedHeight);
40. \_data.writeInt(viewVisibility);
41. \_data.writeInt(((insetsPending)?(1):(0)));
43. mRemote.transact(Stub.TRANSACTION\_relayout, \_data, \_reply, 0);
45. \_reply.readException();
46. \_result = \_reply.readInt();
48. if ((0!=\_reply.readInt())) {
49. outFrame.readFromParcel(\_reply);
50. }
52. if ((0!=\_reply.readInt())) {
53. outContentInsets.readFromParcel(\_reply);
54. }
56. if ((0!=\_reply.readInt())) {
57. outVisibleInsets.readFromParcel(\_reply);
58. }
60. if ((0!=\_reply.readInt())) {
61. outConfig.readFromParcel(\_reply);
62. }
64. if ((0!=\_reply.readInt())) {
65. outSurface.readFromParcel(\_reply);
66. }
68. } finally {
69. \_reply.recycle();
70. \_data.recycle();
71. }
73. return \_result;
74. }
76. ......
77. }
79. ......
80. }
82. ......
83. }

public interface IWindowSession extends android.os.IInterface

{

......

public static abstract class Stub extends android.os.Binder implements android.view.IWindowSession

{

......

private static class Proxy implements android.view.IWindowSession

{

private android.os.IBinder mRemote;

......

public int relayout(android.view.IWindow window, android.view.WindowManager.LayoutParams attrs,

int requestedWidth, int requestedHeight, int viewVisibility, boolean insetsPending,

android.graphics.Rect outFrame,

android.graphics.Rect outContentInsets,

android.graphics.Rect outVisibleInsets,

android.content.res.Configuration outConfig,

android.view.Surface outSurface) throws android.os.RemoteException

{

android.os.Parcel \_data = android.os.Parcel.obtain();

android.os.Parcel \_reply = android.os.Parcel.obtain();

int \_result;

try {

\_data.writeInterfaceToken(DESCRIPTOR);

\_data.writeStrongBinder((((window!=null))?(window.asBinder()):(null)));

if ((attrs!=null)) {

\_data.writeInt(1);

attrs.writeToParcel(\_data, 0);

}

else {

\_data.writeInt(0);

}

\_data.writeInt(requestedWidth);

\_data.writeInt(requestedHeight);

\_data.writeInt(viewVisibility);

\_data.writeInt(((insetsPending)?(1):(0)));

mRemote.transact(Stub.TRANSACTION\_relayout, \_data, \_reply, 0);

\_reply.readException();

\_result = \_reply.readInt();

if ((0!=\_reply.readInt())) {

outFrame.readFromParcel(\_reply);

}

if ((0!=\_reply.readInt())) {

outContentInsets.readFromParcel(\_reply);

}

if ((0!=\_reply.readInt())) {

outVisibleInsets.readFromParcel(\_reply);

}

if ((0!=\_reply.readInt())) {

outConfig.readFromParcel(\_reply);

}

if ((0!=\_reply.readInt())) {

outSurface.readFromParcel(\_reply);

}

} finally {

\_reply.recycle();

\_data.recycle();

}

return \_result;

}

......

}

......

}

......

}

        这个函数定义在文件out/target/common/obj/JAVA\_LIBRARIES/framework\_intermediates/src/core/java/android/view/IWindowSession.java中。

        IWindowSession.Stub.Proxy类的成员函数relayout首先将从前面传进来的各个参数写入到Parcel对象\_data中，接着再通过其成员变量mRemote所描述的一个Binder代理对象向运行在WindowManagerService服务内部的一个Session对象发送一个类型为TRANSACTION\_relayout的进程间通信请求，其中，这个Session对象是用来描述从当前应用程序进程到WindowManagerService服务的一个连接的。

        当运行在WindowManagerService服务内部的Session对象处理完成当前应用程序进程发送过来的类型为TRANSACTION\_relayout的进程间通信请求之后，就会将处理结果写入到Parcel对象\_reply中，并且将这个Parcel对象\_reply返回给当前应用程序进程处理。返回结果包含了一系列与参数window所描述的应用程序窗口相关的参数，如下所示：

       1. 窗口的大小：最终保存在输出参数outFrame中。

       2. 窗口的内容区域边衬大小：最终保存在输出参数outContentInsets中。

       3. 窗口的可见区域边衬大小：最终保存在输出参数outVisibleInsets中。

       4. 窗口的配置信息：最终保存在输出参数outConfig中。

       5. 窗口的绘图表面：最终保存在输出参数outSurface中。

       这里我们只关注从WindowManagerService服务返回来的窗口绘图表面是如何保存到输出参数outSurface中的，即关注Surface类的成员函数readFromParcel的实现。从前面的调用过程可以知道，输出参数outSurface描述的便是当前正在处理的应用程序窗口的绘图表面，将WindowManagerService服务返回来的窗口绘图表面保存在它里面，就相当于是为当前正在处理的应用程序窗口创建了一个绘图表面。

       在分析Surface类的成员函数readFromParcel的实现之前，我们先分析Session类的成员函数relayout的实现，因为它是用来处理类型为TRANSACTION\_relayout的进程间通信请求的。

       Step 6. Session.relayout

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

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 relayout(IWindow window, WindowManager.LayoutParams attrs,
10. int requestedWidth, int requestedHeight, int viewFlags,
11. boolean insetsPending, Rect outFrame, Rect outContentInsets,
12. Rect outVisibleInsets, Configuration outConfig, Surface outSurface) {
13. //Log.d(TAG, ">>>>>> ENTERED relayout from " + Binder.getCallingPid());
14. int res = relayoutWindow(this, window, attrs,
15. requestedWidth, requestedHeight, viewFlags, insetsPending,
16. outFrame, outContentInsets, outVisibleInsets, outConfig, outSurface);
17. //Log.d(TAG, "<<<<<< EXITING relayout to " + Binder.getCallingPid());
18. return res;
19. }
21. ......
23. }
25. ......
26. }

public class WindowManagerService extends IWindowManager.Stub

implements Watchdog.Monitor {

......

private final class Session extends IWindowSession.Stub

implements IBinder.DeathRecipient {

......

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

int requestedWidth, int requestedHeight, int viewFlags,

boolean insetsPending, Rect outFrame, Rect outContentInsets,

Rect outVisibleInsets, Configuration outConfig, Surface outSurface) {

//Log.d(TAG, ">>>>>> ENTERED relayout from " + Binder.getCallingPid());

int res = relayoutWindow(this, window, attrs,

requestedWidth, requestedHeight, viewFlags, insetsPending,

outFrame, outContentInsets, outVisibleInsets, outConfig, outSurface);

//Log.d(TAG, "<<<<<< EXITING relayout to " + Binder.getCallingPid());

return res;

}

......

}

......

}

        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relayout调用了外部类WindowManagerService的成员函数relayoutWindow来对参数参数window所描述的一个应用程序窗口的UI进行布局，接下来我们就继续分析WindowManagerService类的成员函数relayoutWindow的实现。

        Step 7. WindowManagerService.relayoutWindow

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. public class WindowManagerService extends IWindowManager.Stub
2. implements Watchdog.Monitor {
3. ......
5. public int relayoutWindow(Session session, IWindow client,
6. WindowManager.LayoutParams attrs, int requestedWidth,
7. int requestedHeight, int viewVisibility, boolean insetsPending,
8. Rect outFrame, Rect outContentInsets, Rect outVisibleInsets,
9. Configuration outConfig, Surface outSurface) {
10. ......
12. synchronized(mWindowMap) {
13. WindowState win = windowForClientLocked(session, client, false);
14. if (win == null) {
15. return 0;
16. }
18. if (viewVisibility == View.VISIBLE &&
19. (win.mAppToken == null || !win.mAppToken.clientHidden)) {
20. ......
22. try {
23. Surface surface = win.createSurfaceLocked();
24. if (surface != null) {
25. outSurface.copyFrom(surface);
26. ......
27. } else {
28. // For some reason there isn't a surface.  Clear the
29. // caller's object so they see the same state.
30. outSurface.release();
31. }
32. } catch (Exception e) {
33. ......
34. return 0;
35. }
37. ......
38. }
40. ......
41. }
43. return (inTouchMode ? WindowManagerImpl.RELAYOUT\_IN\_TOUCH\_MODE : 0)
44. | (displayed ? WindowManagerImpl.RELAYOUT\_FIRST\_TIME : 0);
45. }
47. ......
48. }

public class WindowManagerService extends IWindowManager.Stub

implements Watchdog.Monitor {

......

public int relayoutWindow(Session session, IWindow client,

WindowManager.LayoutParams attrs, int requestedWidth,

int requestedHeight, int viewVisibility, boolean insetsPending,

Rect outFrame, Rect outContentInsets, Rect outVisibleInsets,

Configuration outConfig, Surface outSurface) {

......

synchronized(mWindowMap) {

WindowState win = windowForClientLocked(session, client, false);

if (win == null) {

return 0;

}

if (viewVisibility == View.VISIBLE &&

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

......

try {

Surface surface = win.createSurfaceLocked();

if (surface != null) {

outSurface.copyFrom(surface);

......

} else {

// For some reason there isn't a surface. Clear the

// caller's object so they see the same state.

outSurface.release();

}

} catch (Exception e) {

......

return 0;

}

......

}

......

}

return (inTouchMode ? WindowManagerImpl.RELAYOUT\_IN\_TOUCH\_MODE : 0)

| (displayed ? WindowManagerImpl.RELAYOUT\_FIRST\_TIME : 0);

}

......

}

        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relayoutWindow的实现是相当复杂的，这里我们只关注与创建应用程序窗口的绘图表面相关的代码，在后面的文章中，我们再详细分析它的实现。

        简单来说，WindowManagerService类的成员函数relayoutWindow根据应用程序进程传递过来的一系列数据来重新设置由参数client所描述的一个应用程序窗口的大小和可见性等信息，而当一个应用程序窗口的大小或者可见性发生变化之后，系统中当前获得焦点的窗口，以及输入法窗口和壁纸窗口等都可能会发生变化，而且也会对其它窗口产生影响，因此，这时候WindowManagerService类的成员函数relayoutWindow就会对系统中的窗口的布局进行重新调整。对系统中的窗口的布局进行重新调整的过程是整个WindowManagerService服务最为复杂和核心的内容，我们同样是在后面的文章中再详细分析。

        现在，我们就主要分析参数client所描述的一个应用程序窗口的绘图表面的创建过程。

        WindowManagerService类的成员函数relayoutWindow首先获得与参数client所对应的一个WindowState对象win，这是通过调用WindowManagerService类的成员函数windowForClientLocked来实现的。如果这个对应的WindowState对象win不存在，那么就说明应用程序进程所请求处理的应用程序窗口不存在，这时候WindowManagerService类的成员函数relayoutWindow就直接返回一个0值给应用程序进程。

        WindowManagerService类的成员函数relayoutWindow接下来判断参数client所描述的一个应用程序窗口是否是可见的。一个窗口只有在可见的情况下，WindowManagerService服务才会为它创建一个绘图表面。 一个窗口是否可见由以下两个条件决定：

        1. 参数viewVisibility的值等于View.VISIBLE，表示应用程序进程请求将它设置为可见的。

        2. WindowState对象win的成员变量mAppToken不等于null，并且它所描述的一个AppWindowToken对象的成员变量clientHidden的值等于false。这意味着参数client所描述的窗口是一个应用程序窗口，即一个Activity组件窗口，并且这个Activity组件当前是处于可见状态的。当一个Activity组件当前是处于不可见状态时，它的窗口就也必须是处于不可见状态。WindowState类的成员变量mAppToken的具体描述可以参考前面[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。

        注意，当WindowState对象win的成员变量mAppToken等于null时，只要满足条件1就可以了，因为这时候参数client所描述的窗口不是一个Activity组件窗口，它的可见性不像Activity组件窗口一样受到Activity组件的可见性影响。

        我们假设参数client所描述的是一个应用程序窗口，并且这个应用程序窗口是可见的，那么WindowManagerService类的成员函数relayoutWindow接下来就会调用WindowState对象win的成员函数createSurfaceLocked来为它创建一个绘图表面。如果这个绘图表面能创建成功，那么WindowManagerService类的成员函数relayoutWindow就会将它的内容拷贝到输出参数outSource所描述的一个Surface对象去，以便可以将它返回给应用程序进程处理。另一方面，如果这个绘图表面不能创建成功，那么WindowManagerService类的成员函数relayoutWindow就会将输出参数outSource所描述的一个Surface对象的内容释放掉，以便应用程序进程知道该Surface对象所描述的绘图表面已经失效了。

        接下来，我们就继续分析WindowState类的成员函数createSurfaceLocked的实现，以便可以了解一个应用程序窗口的绘图表面的创建过程。

        Step 8. WindowState.createSurfaceLocked

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. public class WindowManagerService extends IWindowManager.Stub
2. implements Watchdog.Monitor {
3. ......
5. private final class WindowState implements WindowManagerPolicy.WindowState {
6. ......
8. Surface mSurface;
9. ......
11. Surface createSurfaceLocked() {
12. if (mSurface == null) {
13. mReportDestroySurface = false;
14. mSurfacePendingDestroy = false;
15. mDrawPending = true;
16. mCommitDrawPending = false;
17. mReadyToShow = false;
18. if (mAppToken != null) {
19. mAppToken.allDrawn = false;
20. }
22. int flags = 0;
23. if (mAttrs.memoryType == MEMORY\_TYPE\_PUSH\_BUFFERS) {
24. flags |= Surface.PUSH\_BUFFERS;
25. }
27. if ((mAttrs.flags&WindowManager.LayoutParams.FLAG\_SECURE) != 0) {
28. flags |= Surface.SECURE;
29. }
31. ......
33. int w = mFrame.width();
34. int h = mFrame.height();
35. if ((mAttrs.flags & LayoutParams.FLAG\_SCALED) != 0) {
36. // for a scaled surface, we always want the requested
37. // size.
38. w = mRequestedWidth;
39. h = mRequestedHeight;
40. }
42. // Something is wrong and SurfaceFlinger will not like this,
43. // try to revert to sane values
44. if (w <= 0) w = 1;
45. if (h <= 0) h = 1;
47. mSurfaceShown = false;
48. mSurfaceLayer = 0;
49. mSurfaceAlpha = 1;
50. mSurfaceX = 0;
51. mSurfaceY = 0;
52. mSurfaceW = w;
53. mSurfaceH = h;
54. try {
55. mSurface = new Surface(
56. mSession.mSurfaceSession, mSession.mPid,
57. mAttrs.getTitle().toString(),
58. 0, w, h, mAttrs.format, flags);
59. ......
60. } catch (Surface.OutOfResourcesException e) {
61. ......
62. reclaimSomeSurfaceMemoryLocked(this, "create");
63. return null;
64. } catch (Exception e) {
65. ......
66. return null;
67. }
68. Surface.openTransaction();
69. try {
70. try {
71. mSurfaceX = mFrame.left + mXOffset;
72. mSurfaceY = mFrame.top + mYOffset;
73. mSurface.setPosition(mSurfaceX, mSurfaceY);
74. mSurfaceLayer = mAnimLayer;
75. mSurface.setLayer(mAnimLayer);
76. mSurfaceShown = false;
77. mSurface.hide();
78. if ((mAttrs.flags&WindowManager.LayoutParams.FLAG\_DITHER) != 0) {
79. ......
80. mSurface.setFlags(Surface.SURFACE\_DITHER,
81. Surface.SURFACE\_DITHER);
82. }
83. } catch (RuntimeException e) {
84. ......
85. reclaimSomeSurfaceMemoryLocked(this, "create-init");
86. }
87. mLastHidden = true;
88. } finally {
89. ......
90. Surface.closeTransaction();
91. }
92. ......
93. }
94. return mSurface;
95. }
97. ......
98. }
100. ......
101. }

public class WindowManagerService extends IWindowManager.Stub

implements Watchdog.Monitor {

......

private final class WindowState implements WindowManagerPolicy.WindowState {

......

Surface mSurface;

......

Surface createSurfaceLocked() {

if (mSurface == null) {

mReportDestroySurface = false;

mSurfacePendingDestroy = false;

mDrawPending = true;

mCommitDrawPending = false;

mReadyToShow = false;

if (mAppToken != null) {

mAppToken.allDrawn = false;

}

int flags = 0;

if (mAttrs.memoryType == MEMORY\_TYPE\_PUSH\_BUFFERS) {

flags |= Surface.PUSH\_BUFFERS;

}

if ((mAttrs.flags&WindowManager.LayoutParams.FLAG\_SECURE) != 0) {

flags |= Surface.SECURE;

}

......

int w = mFrame.width();

int h = mFrame.height();

if ((mAttrs.flags & LayoutParams.FLAG\_SCALED) != 0) {

// for a scaled surface, we always want the requested

// size.

w = mRequestedWidth;

h = mRequestedHeight;

}

// Something is wrong and SurfaceFlinger will not like this,

// try to revert to sane values

if (w <= 0) w = 1;

if (h <= 0) h = 1;

mSurfaceShown = false;

mSurfaceLayer = 0;

mSurfaceAlpha = 1;

mSurfaceX = 0;

mSurfaceY = 0;

mSurfaceW = w;

mSurfaceH = h;

try {

mSurface = new Surface(

mSession.mSurfaceSession, mSession.mPid,

mAttrs.getTitle().toString(),

0, w, h, mAttrs.format, flags);

......

} catch (Surface.OutOfResourcesException e) {

......

reclaimSomeSurfaceMemoryLocked(this, "create");

return null;

} catch (Exception e) {

......

return null;

}

Surface.openTransaction();

try {

try {

mSurfaceX = mFrame.left + mXOffset;

mSurfaceY = mFrame.top + mYOffset;

mSurface.setPosition(mSurfaceX, mSurfaceY);

mSurfaceLayer = mAnimLayer;

mSurface.setLayer(mAnimLayer);

mSurfaceShown = false;

mSurface.hide();

if ((mAttrs.flags&WindowManager.LayoutParams.FLAG\_DITHER) != 0) {

......

mSurface.setFlags(Surface.SURFACE\_DITHER,

Surface.SURFACE\_DITHER);

}

} catch (RuntimeException e) {

......

reclaimSomeSurfaceMemoryLocked(this, "create-init");

}

mLastHidden = true;

} finally {

......

Surface.closeTransaction();

}

......

}

return mSurface;

}

......

}

......

}

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

        在创建一个应用程序窗口的绘图表面之前，我们需要知道以下数据：

        1. 应用程序窗口它所运行的应用程序进程的PID。

        2. 与应用程序窗口它所运行的应用程序进程所关联的一个SurfaceSession对象。

        3. 应用程序窗口的标题。

        4. 应用程序窗口的像素格式。

        5. 应用程序窗口的宽度。

        6. 应用程序窗口的高度。

        7. 应用程序窗口的图形缓冲区属性标志。

        第1个和第2个数据可以通过当前正在处理的WindowState对象的成员变量mSession所描述的一个Session对象的成员变量mPid和mSurfaceSession来获得；第3个和第4个数据可以通过当前正在处理的WindowState对象的成员变量mAttr所描述的一个WindowManager.LayoutParams对象的成员函数getTitle以及成员变量format来获得；接下来我们就分析后面3个数据是如何获得的。

        WindowState类的成员变量mFrame的类型为Rect，它用来描述应用程序窗口的位置和大小，它们是由WindowManagerService服务根据屏幕大小以及其它属性计算出来的，因此，通过调用过它的成员函数width和height就可以得到要创建绘图表面的应用程序窗口的宽度和高度，并且保存在变量w和h中。但是，如果当前正在处理的WindowState对象的成员变量mAttr所描述的一个WindowManager.LayoutParams对象的成员变量flags的LayoutParams.FLAG\_SCALED位不等于0时，就说明应用程序进程指定了该应用程序窗口的大小，这时候指定的应用程序窗口的宽度和高度就保存在WindowState类的成员变量mRequestedWidth和mRequestedHeight中，因此，我们就需要将当前正在处理的WindowState对象的成员变量mRequestedWidth和mRequestedHeight的值分别保存在变量w和h中。经过上述两步计算之后，如果得到的变量w和h等于0，那么就说明当前正在处理的WindowState对象所描述的应用程序窗口的大小还没有经过计算，或者还没有被指定过，这时候就需要将它们的值设置为1，避免接下来创建一个大小为0的绘图表面。

        如果当前正在处理的WindowState对象的成员变量mAttr所描述的一个WindowManager.LayoutParams对象的成员变量memoryType的值等于MEMORY\_TYPE\_PUSH\_BUFFERS，那么就说明正在处理的应用程序窗口不拥有专属的图形缓冲区，这时候就需要将用来描述正在处理的应用程序窗口的图形缓冲区属性标志的变量flags的Surface.PUSH\_BUFFERS位设置为1。

       此外，如果当前正在处理的WindowState对象的成员变量mAttr所描述的一个WindowManager.LayoutParams对象的成员变量flags的WindowManager.LayoutParams.FLAG\_SECURE位不等于0，那么就说明正在处理的应用程序窗口的界面是安全的，即是受保护的，这时候就需要将用来描述正在处理的应用程序窗口的图形缓冲区属性标志的变量flags的Surface.SECURE位设置为1。当一个应用程序窗口的界面是受保护时，SurfaceFlinger服务在执行截屏功能时，就不能把它的界面截取下来。

       上述数据准备就绪之后，就可以创建当前正在处理的WindowState对象所描述的一个应用程序窗口的绘图表面了，不过在创建之前，还会初始化该WindowState对象的以下成员变量：

**--mReportDestroySurface的值被设置为false**。当一个应用程序窗口的绘图表面被销毁时，WindowManagerService服务就会将相应的WindowState对象的成员变量mReportDestroySurface的值设置为true，表示要向该应用程序窗口所运行在应用程序进程发送一个绘图表面销毁通知。

**--mSurfacePendingDestroy的值被设置为false**。当一个应用程序窗口的绘图表面正在等待销毁时，WindowManagerService服务就会将相应的WindowState对象的成员变量mReportDestroySurface的值设置为true。

**--mDrawPending的值被设置为true**。当一个应用程序窗口的绘图表面处于创建之后并且绘制之前时，WindowManagerService服务就会将相应的WindowState对象的成员变量mDrawPending的值设置为true，以表示该应用程序窗口的绘图表面正在等待绘制。

**--mCommitDrawPending的值被设置为false**。当一个应用程序窗口的绘图表面绘制完成之后并且可以显示出来之前时，WindowManagerService服务就会将相应的WindowState对象的成员变量mCommitDrawPending的值设置为true，以表示该应用程序窗口正在等待显示。

**--mReadyToShow的值被设置为fals**e。有时候当一个应用程序窗口的绘图表面绘制完成并且可以显示出来之后，由于与该应用程序窗口所关联的一个Activity组件的其它窗口还未准备好显示出来，这时候WindowManagerService服务就会将相应的WindowState对象的成员变量mReadyToShow的值设置为true，以表示该应用程序窗口需要延迟显示出来，即需要等到与该应用程序窗口所关联的一个Activity组件的其它窗口也可以显示出来之后再显示。

**--如果成员变量mAppToken的值不等于null，那么就需要将它所描述的一个AppWindowToken对象的成员变量allDrawn的值设置为false**。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AppWindowToken对象用来描述一个Activity组件的，当该AppWindowToken对象的成员变量allDrawn的值等于true时，就表示属于该Activity组件的所有窗口都已经绘制完成了。

**--mSurfaceShown的值被设置为false**，表示应用程序窗口还没有显示出来，它是用来显示调试信息的。

**--mSurfaceLayer的值被设置为0**，表示应用程序窗口的Z轴位置，它是用来显示调试信息的。

**--mSurfaceAlpha的值被设置为1**，表示应用程序窗口的透明值，它是用来显示调试信息的。

**--mSurfaceX的值被设置为0**，表示应用程序程序窗口的X轴位置，它是用来显示调试信息的。

**--mSurfaceY的值被设置为0**，表示应用程序程序窗口的Y轴位置，它是用来显示调试信息的。

**--mSurfaceW的值被设置为w**，表示应用程序程序窗口的宽度，它是用来显示调试信息的。

**--mSurfaceH的值被设置为h**，表示应用程序程序窗口的高度，它是用来显示调试信息的。

       一个应用程序窗口的绘图表面在创建完成之后，函数就会将得到的一个Surface对象保存在当前正在处理的WindowState对象的成员变量mSurface中。注意，如果创建绘图表面失败，并且从Surface类的构造函数抛出来的异常的类型为Surface.OutOfResourcesException，那么就说明系统当前的内存不足了，这时候函数就会调用WindowManagerService类的成员函数reclaimSomeSurfaceMemoryLocked来回收内存。

       如果一切正常，那么函数接下来还会设置当前正在处理的WindowState对象所描述应用程序窗口的以下属性：

      1. X轴和Y轴位置。前面提到，WindowState类的成员变量mFrame是用来描述应用程序窗口的位置和大小的，其中，位置就是通过它所描述的一个Rect对象的成员变量left和top来表示的，它们分别应用窗口在X轴和Y轴上的位置。此外，当一个WindowState对象所描述的应用程序窗口是一个壁纸窗口时，该WindowState对象的成员变量mXOffset和mYOffset用来描述壁纸窗口相对当前要显示的窗口在X轴和Y轴上的偏移量。因此，将WindowState类的成员变量mXOffset的值加上另外一个成员变量mFrame所描述的一个Rect对象的成员变量left的值，就可以得到一个应用程序窗口在X轴上的位置，同样，将WindowState类的成员变量mYOffset的值加上另外一个成员变量mFrame所描述的一个Rect对象的成员变量top的值，就可以得到一个应用程序窗口在Y轴上的位置。最终得到的位置值就分别保存在WindowState类的成员变量mSurfaceX和mSurfaceY，并且会调用WindowState类的成员变量mSurface所描述的一个Surface对象的成员函数setPosition来将它们设置到SurfaceFlinger服务中去。

      2. Z轴位置。在前面[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)一文中提到，WindowState类的成员变量mAnimLayer用来描述一个应用程序窗口的Z轴位置，因此，这里就会先将它保存在WindowState类的另外一个成员变量mSurfaceLayer中，然后再调用WindowState类的成员变量mSurface所描述的一个Surface对象的成员函数setLayer来将它设置到SurfaceFlinger服务中去。

      3. 抖动标志。如果WindowState类的成员变量mAttr所描述的一个WindowManager.LayoutParams对象的成员变量flags的WindowManager.LayoutParams.FLAG\_DITHER位不等于0，那么就说明一个应用程序窗口的图形缓冲区在渲染时，需要进行抖动处理，这时候就会调用WindowState类的成员变量mSurface所描述的一个Surface对象的成员函数setLayer来将对应的应用程序窗口的图形缓冲区的属性标志的Surface.SURFACE\_DITHER位设置为1。

      4. 显示状态。由于当前正在处理的WindowState对象所描述的一个应用程序窗口的绘图表面刚刚创建出来，因此，我们就需要通知SurfaceFlinger服务将它隐藏起来，这是通过调用当前正在处理的WindowState对象的成员变量mSurface所描述的一个Surface对象的成员变量hide来实现的。这时候还会将当前正在处理的WindowState对象的成员变量mSurfaceShown和mLastHidden的值分别设置为false和true，以表示对应的应用程序窗口是处于隐藏状态的。

      注意，为了避免SurfaceFlinger服务每设置一个应用程序窗口属性就重新渲染一次系统的UI，上述4个属性设置需要在一个事务中进行，这样就可以避免出现界面闪烁。我们通过调用Surface类的静态成员函数openTransaction和closeTransaction就可以分别在SurfaceFlinger服务中打开和关闭一个事务。

      还有另外一个地方需要注意的是，在设置应用程序窗口属性的过程中，如果抛出了一个RuntimeException异常，那么就说明系统当前的内存不足了，这时候函数也会调用WindowManagerService类的成员函数reclaimSomeSurfaceMemoryLocked来回收内存。

      接下来，我们就继续分析Surface类的构造函数的实现，以便可以了解一个应用程序窗口的绘图表面的详细创建过程。

       Step 9. new Surface

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. public class Surface implements Parcelable {
2. ......
4. private int mSurfaceControl;
5. ......
6. private Canvas mCanvas;
7. ......
8. private String mName;
9. ......
11. public Surface(SurfaceSession s,
12. int pid, String name, int display, int w, int h, int format, int flags)
13. throws OutOfResourcesException {
14. ......
16. mCanvas = new CompatibleCanvas();
17. init(s,pid,name,display,w,h,format,flags);
18. mName = name;
19. }
21. ......
23. private native void init(SurfaceSession s,
24. int pid, String name, int display, int w, int h, int format, int flags)
25. throws OutOfResourcesException;
27. ......
28. }

public class Surface implements Parcelable {

......

private int mSurfaceControl;

......

private Canvas mCanvas;

......

private String mName;

......

public Surface(SurfaceSession s,

int pid, String name, int display, int w, int h, int format, int flags)

throws OutOfResourcesException {

......

mCanvas = new CompatibleCanvas();

init(s,pid,name,display,w,h,format,flags);

mName = name;

}

......

private native void init(SurfaceSession s,

int pid, String name, int display, int w, int h, int format, int flags)

throws OutOfResourcesException;

......

}

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

        Surface类有三个成员变量mSurfaceControl、mCanvas和mName，它们的类型分别是int、Canvas和String，其中，mSurfaceControl保存的是在C++层的一个SurfaceControl对象的地址值，mCanvas用来描述一块类型为CompatibleCanvas的画布，mName用来描述当前正在创建的一个绘图表面的名称。画布是真正用来绘制UI的地方，不过由于现在正在创建的绘图表面是在WindowManagerService服务这一侧使用的，而WindowManagerService服务不会去绘制应用程序窗口的UI，它只会去设置应用程序窗口的属性，因此，这里创建的画布实际上没有什么作用，我们主要关注与成员变量mSurfaceControl所关联的C++层的SurfaceControl对象是如何创建的。

        Surface类的构造函数是通过调用另外一个成员函数init来创建与成员变量mSurfaceControl所关联的C++层的SurfaceControl对象的。Surface类的成员函数init是一个JNI方法，它是由C++层的函数Surface\_init来实现的，如下所示：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. static void Surface\_init(
2. JNIEnv\* env, jobject clazz,
3. jobject session,
4. jint pid, jstring jname, jint dpy, jint w, jint h, jint format, jint flags)
5. {
6. if (session == NULL) {
7. doThrow(env, "java/lang/NullPointerException");
8. return;
9. }
11. SurfaceComposerClient\* client =
12. (SurfaceComposerClient\*)env->GetIntField(session, sso.client);
14. sp<SurfaceControl> surface;
15. if (jname == NULL) {
16. surface = client->createSurface(pid, dpy, w, h, format, flags);
17. } else {
18. const jchar\* str = env->GetStringCritical(jname, 0);
19. const String8 name(str, env->GetStringLength(jname));
20. env->ReleaseStringCritical(jname, str);
21. surface = client->createSurface(pid, name, dpy, w, h, format, flags);
22. }
24. if (surface == 0) {
25. doThrow(env, OutOfResourcesException);
26. return;
27. }
28. setSurfaceControl(env, clazz, surface);
29. }

static void Surface\_init(

JNIEnv\* env, jobject clazz,

jobject session,

jint pid, jstring jname, jint dpy, jint w, jint h, jint format, jint flags)

{

if (session == NULL) {

doThrow(env, "java/lang/NullPointerException");

return;

}

SurfaceComposerClient\* client =

(SurfaceComposerClient\*)env->GetIntField(session, sso.client);

sp<SurfaceControl> surface;

if (jname == NULL) {

surface = client->createSurface(pid, dpy, w, h, format, flags);

} else {

const jchar\* str = env->GetStringCritical(jname, 0);

const String8 name(str, env->GetStringLength(jname));

env->ReleaseStringCritical(jname, str);

surface = client->createSurface(pid, name, dpy, w, h, format, flags);

}

if (surface == 0) {

doThrow(env, OutOfResourcesException);

return;

}

setSurfaceControl(env, clazz, surface);

}

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

        参数session指向了在Java层所创建的一个SurfaceSession对象。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，Java层的SurfaceSession对象有一个成员变量mClient，它指向了在C++层中的一个SurfaceComposerClient对象。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)这一系列的文章又可以知道，C++层的SurfaceComposerClient对象可以用来请求SurfaceFlinger服务为应用程序窗口创建绘图表面，即创建一个Layer对象。

        因此，函数首先将参数session所指向的一个Java层的SurfaceSession对象的成员变量mClient转换成一个SurfaceComposerClient对象，然后再调用这个SurfaceComposerClient对象的成员函数createSurface来请求SurfaceFlinger服务来为参数clazz所描述的一个Java层的Surface对象所关联的应用程序窗口创建一个Layer对象。SurfaceFlinger服务创建完成这个Layer对象之后，就会将该Layer对象内部的一个实现了ISurface接口的SurfaceLayer对象返回给函数，于是，函数就可以获得一个实现了ISurface接口的Binder代理对象。这个实现了ISurface接口的Binder代理对象被封装在C++层的一个SurfaceControl对象surface中。

       注意，sso是一个全局变量，它是一个类型为sso\_t的结构体，它的定义如下所示：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

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

struct sso\_t {

jfieldID client;

};

static sso\_t sso;

        它的成员函数client用来描述Java层中的SurfaceSession类的成员变量mClient在类中的偏移量，因此，函数Surface\_init通过这个偏移量就可以访问参数session所指向的一个SurfaceSession对象的成员变量mClient的值，从而获得一个SurfaceComposerClient对象。

       另外一个需要注意的地方是，当参数name的值等于null时，函数Surface\_init调用前面所获得一个SurfaceComposerClient对象的六个参数版本的成员函数createSurface来请求SurfaceFlinger服务创建一个Layer对象，否则的话，就调用七个参数版本的成员函数createSurface来请求SurfaceFlinger服务创建一个Layer对象。SurfaceComposerClient类的成员函数createSurface的实现可以参考前面[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。

       得到了SurfaceControl对象surface之后，函数Surface\_init接下来继续调用另外一个函数setSurfaceControl来它的地址值保存在参数clazz所指向的一个Java层的Surface对象的成员变量mSurfaceControl中，如下所示：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. static void setSurfaceControl(JNIEnv\* env, jobject clazz,
2. const sp<SurfaceControl>& surface)
3. {
4. SurfaceControl\* const p =
5. (SurfaceControl\*)env->GetIntField(clazz, so.surfaceControl);
6. if (surface.get()) {
7. surface->incStrong(clazz);
8. }
9. if (p) {
10. p->decStrong(clazz);
11. }
12. env->SetIntField(clazz, so.surfaceControl, (**int**)surface.get());
13. }

static void setSurfaceControl(JNIEnv\* env, jobject clazz,

const sp<SurfaceControl>& surface)

{

SurfaceControl\* const p =

(SurfaceControl\*)env->GetIntField(clazz, so.surfaceControl);

if (surface.get()) {

surface->incStrong(clazz);

}

if (p) {

p->decStrong(clazz);

}

env->SetIntField(clazz, so.surfaceControl, (int)surface.get());

}

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

        在分析函数setSurfaceControl的实现之前，我们先分析全局变量so的定义，如下所示：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. struct so\_t {
2. jfieldID surfaceControl;
3. jfieldID surface;
4. jfieldID saveCount;
5. jfieldID canvas;
6. };
7. static so\_t so;

struct so\_t {

jfieldID surfaceControl;

jfieldID surface;

jfieldID saveCount;

jfieldID canvas;

};

static so\_t so;

        它是一个类型为so\_t的结构体。结构体so\_t有四个成员变量surfaceControl、surface、saveCount和canvas，它们分别用来描述Java层的Surface类的四个成员变量mSurfaceControl、mNativeSurface、mSaveCount和mCanvas在类中的偏移量。

       回到函数setSurfaceControl中，它首先通过结构体so的成员变量surfaceControl来获得参数clazz所指向的一个Java层的Surface对象的成员变量mSurfaceControl所关联的一个C++层的SurfaceControl对象。如果这个SurfaceControl对象存在，那么变量p的值就不会等于null，在这种情况下，就需要调用它的成员函数decStrong来减少它的强引用计数，因为接下来参数clazz所指向的一个Java层的Surface对象不再通过成员变量mSurfaceControl来引用它了。

       另一方面，函数setSurfaceControl需要增加参数surface所指向的一个C++层的SurfaceControl对象的强引用计数，即调用参数surface所指向的一个C++层的SurfaceControl对象的成员函数incStrong，因为接下来参数clazz所指向的一个Java层的Surface对象要通过成员变量mSurfaceControl来引用它，即将它的地址值保存在参数clazz所指向的一个Java层的Surface对象的成员变量mSurfaceControl中。

       这一步执行完成之后，沿着调用路径层层返回，回到前面的Step 8中，即WindowState类的成员函数createSurfaceLocked中，这时候一个绘图表面就创建完成了。这个绘图表面最终会返回给请求创建它的应用程序进程，即前面的Step 5，也就是IWindowSession.Stub.Proxy类的成员函数relayout。

       IWindowSession.Stub.Proxy类的成员函数relayout获得了从WindowManagerService服务返回来的一个绘图表面，即一个Java层的Surface对象之后，就会将它的内容拷贝到参数outSurface所描述的另外一个Java层的Surface对象中，这里通过调用Surface类的成员函数readFromParcel来实现的。注意，参数outSurface所描述的这个Java层的Surface对象是在应用程序进程这一侧创建的，它的作用是用来绘制应用程序窗品的UI。

       接下来，我们就继续分析Surface类的成员函数readFromParcel的实现，以便可以了解在应用程序进程这一侧的Surface对象是如何创建的。

       Step 10. Surface.readFromParcel

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. public class Surface implements Parcelable {
2. ......
4. public native   void readFromParcel(Parcel source);
5. ......
7. }

public class Surface implements Parcelable {

......

public native void readFromParcel(Parcel source);

......

}

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

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

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. static void Surface\_readFromParcel(
2. JNIEnv\* env, jobject clazz, jobject argParcel)
3. {
4. Parcel\* parcel = (Parcel\*)env->GetIntField( argParcel, no.native\_parcel);
5. if (parcel == NULL) {
6. doThrow(env, "java/lang/NullPointerException", NULL);
7. return;
8. }
10. sp<Surface> sur(Surface::readFromParcel(\*parcel));
11. setSurface(env, clazz, sur);
12. }

static void Surface\_readFromParcel(

JNIEnv\* env, jobject clazz, jobject argParcel)

{

Parcel\* parcel = (Parcel\*)env->GetIntField( argParcel, no.native\_parcel);

if (parcel == NULL) {

doThrow(env, "java/lang/NullPointerException", NULL);

return;

}

sp<Surface> sur(Surface::readFromParcel(\*parcel));

setSurface(env, clazz, sur);

}

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

       参数argParcel所指向的一个Parcel对象的当前位置保存的是一个Java层的Surface对象的内容，函数readFromParcel首先调用C++层的Surface类的成员函数readFromParcel来将这些内容封装成一个C++层的Surface对象，如下所示：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. sp<Surface> Surface::readFromParcel(const Parcel& data) {
2. Mutex::Autolock \_l(sCachedSurfacesLock);
3. sp<IBinder> binder(data.readStrongBinder());
4. sp<Surface> surface = sCachedSurfaces.valueFor(binder).promote();
5. if (surface == 0) {
6. surface = new Surface(data, binder);
7. sCachedSurfaces.add(binder, surface);
8. }
9. if (surface->mSurface == 0) {
10. surface = 0;
11. }
12. cleanCachedSurfacesLocked();
13. return surface;
14. }

sp<Surface> Surface::readFromParcel(const Parcel& data) {

Mutex::Autolock \_l(sCachedSurfacesLock);

sp<IBinder> binder(data.readStrongBinder());

sp<Surface> surface = sCachedSurfaces.valueFor(binder).promote();

if (surface == 0) {

surface = new Surface(data, binder);

sCachedSurfaces.add(binder, surface);

}

if (surface->mSurface == 0) {

surface = 0;

}

cleanCachedSurfacesLocked();

return surface;

}

        这个函数定义在文件frameworks/base/libs/surfaceflinger\_client/Surface.cpp中。

        参数data所指向的一个Parcel对象的当前位置保存的是一个Binder代理对象，这个Binder代理对象实现了ISurface接口，它所引用的Binder本地对象就是在前面的Step 9中WindowManagerService服务请求SurfaceFlinger服务所创建的一个Layer对象的内部的一个SurfaceLayer对象。

        获得了一个实现了ISurface接口的Binder代理对象binder之后，C++层的Surface类的成员函数readFromParcel就可以将它封装在一个C++层的Surface对象中了，并且将这个C++层的Surface对象返回给调用者。注意，C++层的Surface类的成员函数readFromParcel在创建为Binder代理对象binder创建一个C++层的Surface对象之前，首先会在C++层的Surface类的静态成员变量sCachedSurfaces所描述的一个DefaultKeyedVectort向量中检查是否已经为存在一个对应的C++层的Surface对象了。如果已经存在，那么就会直接将这个C++层的Surface对象返回给调用者。

        回到函数Surface\_readFromParcel中，接下来它就会调用另外一个函数setSurface来将前面所获得的一个C++层的Surface对象sur保存在参数clazz所描述的一个Java层的Surface对象的成员变量mNativeSurface中，如下所示：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8303098)

1. static void setSurface(JNIEnv\* env, jobject clazz, const sp<Surface>& surface)
2. {
3. Surface\* const p = (Surface\*)env->GetIntField(clazz, so.surface);
4. if (surface.get()) {
5. surface->incStrong(clazz);
6. }
7. if (p) {
8. p->decStrong(clazz);
9. }
10. env->SetIntField(clazz, so.surface, (**int**)surface.get());
11. }

static void setSurface(JNIEnv\* env, jobject clazz, const sp<Surface>& surface)

{

Surface\* const p = (Surface\*)env->GetIntField(clazz, so.surface);

if (surface.get()) {

surface->incStrong(clazz);

}

if (p) {

p->decStrong(clazz);

}

env->SetIntField(clazz, so.surface, (int)surface.get());

}

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

       全局变量so是一个类型为so\_t的结构体，在前面的Step 9中，我们已经分析过它的定义了，函数setSurface首先通过它的成员变量surface来将参数clazz所描述的一个Java层的Surface对象的成员变量mNativeSurface转换成一个Surface对象p。如果这个Surface对象p存在，那么就需要调用它的成员函数decStrong来减少它的强引用计数，因为接下来参数clazz所描述的一个Java层的Surface对象不再通过成员变量mNativeSurface来引用它了。

       函数setSurface接下来就会将参数surface所指向的一个C++层的Surface对象的地址值保存在参数clazz所描述的一个Java层的Surface对象的成员变量mNativeSurface中。在执行这个操作之前，函数setSurface需要调用参数surface所指向的一个C++层的Surface对象的成员函数incStrong来增加它的强引用计数，这是因为接下来它要被参数clazz所描述的一个Java层的Surface对象通过成员变量mNativeSurface来引用了。

       至此，我们就分析完成Android应用程序窗口的绘图表面的创建过程了。通过这个过程我们就可以知道：

       1. 每一个应用程序窗口都对应有两个Java层的Surface对象，其中一个是在WindowManagerService服务这一侧创建的，而另外一个是在应用程序进程这一侧创建的。

       2. 在WindowManagerService服务这一侧创建的Java层的Surface对象在C++层关联有一个SurfaceControl对象，用来设置应用窗口的属性，例如，大小和位置等。

       3. 在应用程序进程这一侧创建的ava层的Surface对象在C++层关联有一个Surface对象，用来绘制应用程序窗品的UI。

       理解上述三个结论对理解Android应用程序窗口的实现框架以及WindowManagerService服务的实现都非常重要。 一个应用程序窗口的绘图表面在创建完成之后，接下来应用程序进程就可以在上面绘制它的UI了。在接下来的一篇文章中，我们就继续分析Android应用程序窗品的绘制过程，敬请关注！