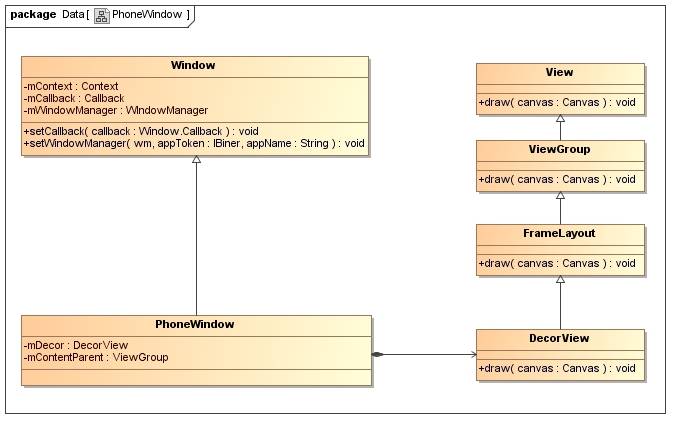
Android之旅<https://www.kancloud.cn/alex_wsc/androids/473772>

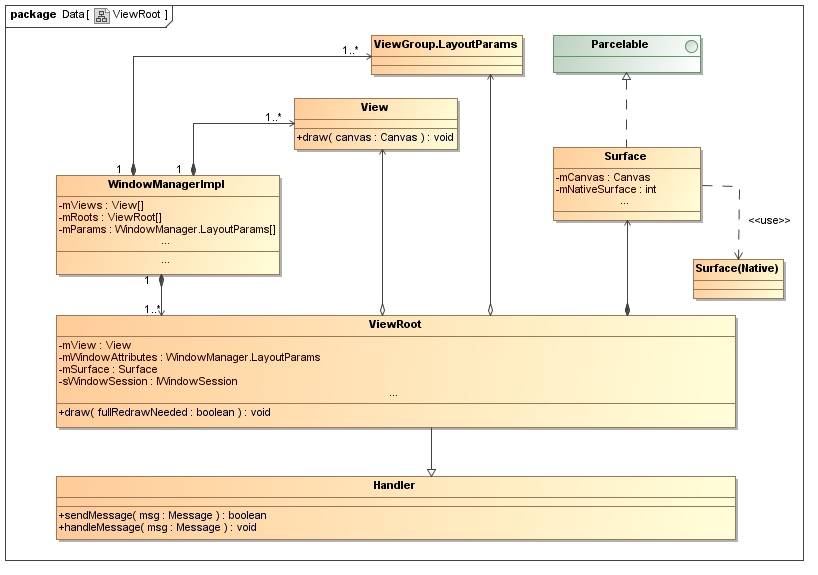
视图对象（View）的创建过程分析

原文出处——>[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)

从前文可知道，每一个Activity组件都有一个关联的Window对象，用来描述一个应用程序窗口。每一个应用程序窗口内部又包含有一个View对象，用来描述应用程序窗口的视图。应用程序窗口视图是真正用来实现UI内容和布局的，也就是说，每一个Activity组件的UI内容和布局都是通过与其所关联的一个Window对象的内部的一个View对象来实现的。在本文中，我们就详细分析应用程序窗口视图的创建过程。

在前面Android应用程序窗口（Activity）实现框架简要介绍和学习计划一文中提到，应用程序窗口内部所包含的视图对象的实际类型为DecorView。DecorView类继承了View类，是作为容器（ViewGroup）来使用的，它的实现如图1所示：  
:-:   
图1 DecorView类的实现

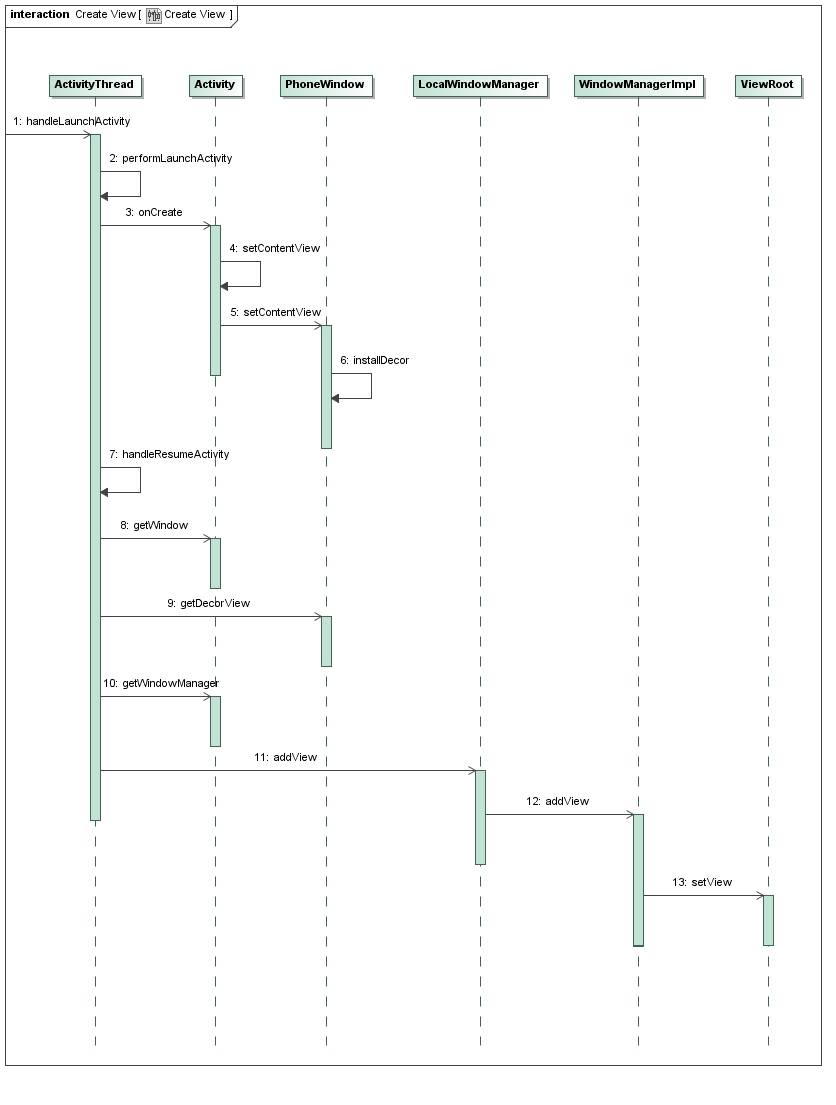
这个图的具体描述可以参考Android应用程序窗口（Activity）实现框架简要介绍和学习计划一文中的图5，这里不再详述。

从前面Android应用程序窗口（Activity）实现框架简要介绍和学习计划一文还可以知道，每一个应用程序窗口的视图对象都有一个关联的ViewRoot对象，这些关联关系是由窗口管理器来维护的，如图2所示：  
:-:   
图2 应用程序窗口视图与ViewRoot的关系图

这个图的具体描述可以参考Android应用程序窗口（Activity）实现框架简要介绍和学习计划一文中的图6，这里不再详述。

简单来说，ViewRoot相当于是MVC模型中的Controller，它有以下职责：  
1. 负责为应用程序窗口视图创建Surface。  
2. 配合WindowManagerService来管理系统的应用程序窗口。  
3. 负责管理、布局和渲染应用程序窗口视图的UI。

那么，应用程序窗口的视图对象及其所关联的ViewRoot对象是什么时候开始创建的呢？ 从前面Android应用程序窗口（Activity）的窗口对象（Window）的创建过程分析一文可以知道，Activity组件在启动的时候，系统会为它创建窗口对象（Window），同时，系统也会为这个窗口对象创建视图对象。另一方面，当Activity组件被激活的时候，系统如果发现与它的应用程序窗口视图对象所关联的ViewRoot对象还没有创建，那么就会先创建这个ViewRoot对象，以便接下来可以将它的UI渲染出来。

从前面Android应用程序启动过程源代码分析一文可以知道，Activity组件在启动的过程中，会调用ActivityThread类的成员函数handleLaunchActivity，用来创建以及首次激活Activity组件，因此，接下来我们就从这个函数开始，具体分析应用程序窗口的视图对象及其所关联的ViewRoot对象的创建过程，如图3所示：  
  
图3 应用程序窗口视图的创建过程

这个过程一共可以分为13个步骤，接下来我们就详细分析每一个步骤。

**Step 1. ActivityThread.handleLaunchActivity**

public final class ActivityThread {

......

private final void handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {

......

Activity a = performLaunchActivity(r, customIntent);

if (a != null) {

......

handleResumeActivity(r.token, false, r.isForward);

......

}

......

}

......

}

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

函数首先调用ActivityThread类的成员函数performLaunchActivity来创建要启动的Activity组件。在创建Activity组件的过程中，还会为该Activity组件创建窗口对象和视图对象。Activity组件创建完成之后，就可以将它激活起来了，这是通过调用ActivityThread类的成员函数handleResumeActivity来执行的。

接下来，我们首先分析ActivityThread类的成员函数performLaunchActivity的实现，以便可以了解应用程序窗口视图对象的创建过程，接着再回过头来继续分析ActivityThread类的成员函数handleResumeActivity的实现，以便可以了解与应用程序窗口视图对象所关联的ViewRoot对象的创建过程。

**Step 2. ActivityThread.performLaunchActivity**

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

这一步可以参考Android应用程序窗口（Activity）的运行上下文环境（Context）的创建过程分析一文的Step 1，它主要就是创建一个Activity组件实例，并且调用这个Activity组件实例的成员函数onCreate来让其执行一些自定义的初始化工作。

**Step 3. Activity.onCreate**

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

这一步可以参考Android应用程序窗口（Activity）的运行上下文环境（Context）的创建过程分析一文的Step 10。我们在实现一个Activity组件的时候，也就是在实现一个Activity子类的时候，一般都会重写成员函数onCreate，以便可以执行一些自定义的初始化工作，其中就包含初始化UI的工作。例如，在前面在Ubuntu上为Android系统内置Java应用程序测试Application Frameworks层的硬件服务一文中，我们实现了一个名称为Hello的Activity组件，用来测试硬件服务，它的成员函数onCreate的样子长得大概如下所示：

public class Hello extends Activity implements OnClickListener {

......

/\*\* Called when the activity is first created. \*/

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

......

}

......

}

其中，调用从父类Activity继承下来的成员函数setContentView就是用来创建应用程序窗口视图对象的。

接下来，我们就继续分析Activity类的成员函数setContentView的实现。

**Step 4. Activity.setContentView**

public class Activity extends ContextThemeWrapper

implements LayoutInflater.Factory,

Window.Callback, KeyEvent.Callback,

OnCreateContextMenuListener, ComponentCallbacks {

......

private Window mWindow;

......

public Window getWindow() {

return mWindow;

}

......

public void setContentView(int layoutResID) {

getWindow().setContentView(layoutResID);

}

......

}

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

Activity类的成员函数setContentView首先调用另外一个成员函数getWindow来获得成员变量mWindow所描述的一个窗口对象，接着再调用这个窗口对象的成员函数setContentView来执行创建应用程序窗口视图对象的工作。

从前面Android应用程序窗口（Activity）的窗口对象（Window）的创建过程分析一文可以知道，Activity类的成员变量mWindow指向的是一个PhoneWindow对象，因此，接下来我们就继续分析PhoneWindow类的成员函数setContentView的实现。

**Step 5. PhoneWindow.setContentView**

public class PhoneWindow extends Window implements MenuBuilder.Callback {

......

// This is the view in which the window contents are placed. It is either

// mDecor itself, or a child of mDecor where the contents go.

private ViewGroup mContentParent;

......

@Override

public void setContentView(int layoutResID) {

if (mContentParent == null) {

installDecor();

} else {

mContentParent.removeAllViews();

}

mLayoutInflater.inflate(layoutResID, mContentParent);

final Callback cb = getCallback();

if (cb != null) {

cb.onContentChanged();

}

}

......

}

这个函数定义在文件frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindow.java中。

PhoneWindow类的成员变量mContentParent用来描述一个类型为DecorView的视图对象，或者这个类型为DecorView的视图对象的一个子视图对象，用作UI容器。当它的值等于null的时候，就说明正在处理的应用程序窗口的视图对象还没有创建。在这种情况下，就会调用成员函数installDecor来创建应用程序窗口视图对象。否则的话，就说明是要重新设置应用程序窗口的视图。在重新设置之前，首先调用成员变量mContentParent所描述的一个ViewGroup对象来移除原来的UI内空。

由于我们是在Activity组件启动的过程中创建应用程序窗口视图的，因此，我们就假设此时PhoneWindow类的成员变量mContentParent的值等于null。接下来，函数就会调用成员函数installDecor来创建应用程序窗口视图对象，接着再通过调用PhoneWindow类的成员变量mLayoutInflater所描述的一个LayoutInflater对象的成员函数inflate来将参数layoutResID所描述的一个UI布局设置到前面所创建的应用程序窗口视图中去，最后还会调用一个Callback接口的成员函数onContentChanged来通知对应的Activity组件，它的视图内容发生改变了。从前面Android应用程序窗口（Activity）的窗口对象（Window）的创建过程分析一文可以知道，Activity组件自己实现了这个Callback接口，并且将这个Callback接口设置到了与它所关联的应用程序窗口对象的内部去，因此，前面实际调用的是Activity类的成员函数onContentChanged来发出一个视图内容变化通知。

接下来，我们就继续分析PhoneWindow类的成员函数installDecor的实现，以便可以继续了解应用程序窗口视图对象的创建过程。

**Step 6. PhoneWindow.installDecor**

public class PhoneWindow extends Window implements MenuBuilder.Callback {

......

// This is the top-level view of the window, containing the window decor.

private DecorView mDecor;

......

// This is the view in which the window contents are placed. It is either

// mDecor itself, or a child of mDecor where the contents go.

private ViewGroup mContentParent;

......

private TextView mTitleView;

......

private CharSequence mTitle = null;

......

private void installDecor() {

if (mDecor == null) {

mDecor = generateDecor();

......

}

if (mContentParent == null) {

mContentParent = generateLayout(mDecor);

mTitleView = (TextView)findViewById(com.android.internal.R.id.title);

if (mTitleView != null) {

if ((getLocalFeatures() & (1 << FEATURE\_NO\_TITLE)) != 0) {

View titleContainer = findViewById(com.android.internal.R.id.title\_container);

if (titleContainer != null) {

titleContainer.setVisibility(View.GONE);

} else {

mTitleView.setVisibility(View.GONE);

}

if (mContentParent instanceof FrameLayout) {

((FrameLayout)mContentParent).setForeground(null);

}

} else {

mTitleView.setText(mTitle);

}

}

}

}

......

}

这个函数定义在文件frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindow.java中。

由于我们是在Activity组件启动的过程中创建应用程序窗口视图的，因此，我们同时假设此时PhoneWindow类的成员变量mDecor的值等于null。这时候PhoneWindow类的成员函数installDecor就会调用另外一个成员函数generateDecor来创建一个DecorView对象，并且保存在PhoneWindow类的成员变量mDecor中。

PhoneWindow类的成员函数installDecor接着再调用另外一个成员函数generateLayout来根据当前应用程序窗口的Feature来加载对应的窗口布局文件。这些布局文件保存在frameworks/base/core/res/res/layout目录下，它们必须包含有一个id值为“content”的布局控件。这个布局控件必须要从ViewGroup类继承下来，用来作为窗口的UI容器。PhoneWindow类的成员函数generateLayout执行完成之后，就会这个id值为“content”的ViewGroup控件来给PhoneWindow类的成员函数installDecor，后者再将其保存在成员变量mContentParent中。

PhoneWindow类的成员函数installDecor还会检查前面加载的窗口布局文件是否包含有一个id值为“title”的TextView控件。如果包含有的话，就会将它保存在PhoneWindow类的成员变量mTitleView中，用来描述当前应用程序窗口的标题栏。但是，如果当前应用程序窗口是没有标题栏的，即它的Feature位FEATURE\_NO\_TITLE的值等于1，那么PhoneWindow类的成员函数installDecor就需要将前面得到的标题栏隐藏起来。注意，PhoneWindow类的成员变量mTitleView所描述的标题栏有可能是包含在一个id值为“title\_container”的容器里面的，在这种情况下，就需要隐藏该标题栏容器。另一方面，如果当前应用程序窗口是设置有标题栏的，那么PhoneWindow类的成员函数installDecor就会设置它的标题栏文字。应用程序窗口的标题栏文字保存在PhoneWindow类的成员变量mTitle中，我们可以调用PhoneWindow类的成员函数setTitle来设置。

这一步执行完成之后，应用程序窗口视图就创建完成了，回到前面的Step 1中，即ActivityThread类的成员函数handleLaunchActivity中，接下来就会调用ActivityThread类的另外一个成员函数handleResumeActivity来激活正在启动的Activity组件。由于在是第一次激活该Activity组件，因此，在激活之前，还会为该Activity组件创建一个ViewRoot对象，并且与前面所创建的应用程序窗口视图关联起来，以便后面可以通过该ViewRoot对象来控制应用程序窗口视图的UI展现。

接下来，我们就继续分析ActivityThread类的成员函数handleResumeActivity的实现。

**Step 7. ActivityThread.handleResumeActivity**

public final class ActivityThread {

......

final void handleResumeActivity(IBinder token, boolean clearHide, boolean isForward) {

......

ActivityClientRecord r = performResumeActivity(token, clearHide);

if (r != null) {

final Activity a = r.activity;

......

// If the window hasn't yet been added to the window manager,

// and this guy didn't finish itself or start another activity,

// then go ahead and add the window.

boolean willBeVisible = !a.mStartedActivity;

if (!willBeVisible) {

try {

willBeVisible = ActivityManagerNative.getDefault().willActivityBeVisible(

a.getActivityToken());

} catch (RemoteException e) {

}

}

if (r.window == null && !a.mFinished && willBeVisible) {

r.window = r.activity.getWindow();

View decor = r.window.getDecorView();

decor.setVisibility(View.INVISIBLE);

ViewManager wm = a.getWindowManager();

WindowManager.LayoutParams l = r.window.getAttributes();

a.mDecor = decor;

l.type = WindowManager.LayoutParams.TYPE\_BASE\_APPLICATION;

......

if (a.mVisibleFromClient) {

a.mWindowAdded = true;

wm.addView(decor, l);

}

}

......

}

......

}

......

}

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

ActivityThread类的成员函数handleResumeActivity首先调用另外一个成员函数performResumeActivity来通知Activity组件，它要被激活了，即会导致Activity组件的成员函数onResume被调用。ActivityThread类的成员函数performResumeActivity的返回值是一个ActivityClientRecord对象r，这个ActivityClientRecord对象的成员变量activity描述的就是正在激活的Activity组件a。

ActivityThread类的成员函数handleResumeActivity接下来判断正在激活的Activity组件接下来是否是可见的。如果是可见的，那么变量willBeVisible的值就会等于true。Activity类的成员变量mStartedActivity用来描述一个Activity组件是否正在启动一个新的Activity组件，并且等待这个新的Activity组件的执行结果。如果是的话，那么这个Activity组件的成员变量mStartedActivity的值就会等于true，表示在新的Activity组件的执行结果返回来之前，当前Activity组件要保持不可见的状态。因此，当Activity组件a的成员变量mStartedActivity的值等于true的时候，它接下来就是不可见的，否则的话，就是可见的。

虽然说在Activity组件a的成员变量mStartedActivity的值等于true的情况下，它接下来的状态要保持不可见的，但是有可能它所启动的Activity组件的UI不是全屏的。在这种情况下，Activity组件a的UI仍然是有部分可见的，这时候也要将变量willBeVisible的值设置为true。因此，如果前面得到变量willBeVisible的值等于false，那么ActivityThread类的成员函数handleResumeActivity接下来就会通过Binder进程间通信机制来调用ActivityManagerService服务的成员函数willActivityBeVisible来检查位于Activity组件a上面的其它Activity组件（包含了Activity组件a正在等待其执行结果的Activity组件）是否是全屏的。如果不是，那么ActivityManagerService服务的成员函数willActivityBeVisible的返回值就会等于true，表示接下来需要显示Activity组件a。

前面得到的ActivityClientRecord对象r的成员变量window用来描述当前正在激活的Activity组件a所关联的应用程序窗口对象。当它的值等于null的时候，就表示当前正在激活的Activity组件a所关联的应用程序窗口对象还没有关联一个ViewRoot对象。进一步地，如果这个正在激活的Activity组件a还活着，并且接下来是可见的，即ActivityClientRecord对象r的成员变量mFinished的值等于false，并且前面得到的变量willBeVisible的值等于true，那么这时候就说明需要为与Activity组件a所关联的一个应用程序窗口视图对象关联的一个ViewRoot对象。

将一个Activity组件的应用程序窗口视图对象与一个ViewRoot对象关联是通过该Activity组件所使用的窗口管理器来执行的。从前面Android应用程序窗口（Activity）的窗口对象（Window）的创建过程分析一文可以知道，一个Activity组件所使用的本地窗口管理器保存它的成员变量mWindowManager中，这可以通过Activity类的成员函数getWindowManager来获得。在接下来的Step 10中，我们再分析Activity类的成员函数getWindowManager的实现。

由于我们现在要给Activity组件a的应用程序窗口视图对象关联一个ViewRoot对象，因此，我们就需要首先获得这个应用程序窗口视图对象。从前面的Step 6可以知道，一个Activity组件的应用程序窗口视图对象保存在与其所关联的一个应用程序窗口对象的内部，因此，我们又要首先获得这个应用程序窗口对象。与一个Activity组件所关联的应用程序窗口对象可以通过调用该Activity组件的成员函数getWindow来获得。一旦获得了这个应用程序窗口对象（类型为PhoneWindow）之后，我们就可以调用它的成员函数getDecorView来获得它内部的视图对象。在接下来的Step 8和Step 9中，我们再分别分析Activity类的成员函数Activity类的成员函数getWindow和PhoneWindow类的成员函数getDecorView的实现。

在关联应用程序窗口视图对象和ViewRoot对象的时候，还需要第三个参数，即应用程序窗口的布局参数，这是一个类型为WindowManager.LayoutParams的对象，可以通过调用应用程序窗口的成员函数getAttributes来获得。一切准备就绪之后，还要判断最后一个条件是否成立，即当前正在激活的Activity组件a在本地进程中是否是可见的，即它的成员变量mVisibleFromClient的值是否等于true。如果是可见的，那么最后就可以调用前面所获得的一个本地窗口管理器wm（类型为LocalWindowManager）的成员函数addView来执行关联应用程序窗口视图对象和ViewRoot对象的操作。

接下来，我们就分别分析Activity类的成员函数getWindow、PhoneWindow类的成员函数getDecorView、ctivity类的成员函数getWindowManager以及LocalWindowManager类的成员函数addView的实现。

**Step 8. Activity.getWindow**

public class Activity extends ContextThemeWrapper

implements LayoutInflater.Factory,

Window.Callback, KeyEvent.Callback,

OnCreateContextMenuListener, ComponentCallbacks {

......

private Window mWindow;

......

public Window getWindow() {

return mWindow;

}

......

}

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

从前面Android应用程序窗口（Activity）的窗口对象（Window）的创建过程分析一文可以知道，Activity类的成员变量mWindow指向的是一个类型为PhoneWindow的窗口对象，因此，Activity类的成员函数getWindow返回给调用者的是一个PhoneWindow对象。

这一步执完成之后，返回到前面的Step 7中，即ActivityThread类的成员函数handleResumeActivity中，接下来就会继续调用前面所获得的一个PhoneWindow对象的成员函数getDecorView来获得当前正在激活的Activity组件所关联的一个应用程序窗口视图对象。

**Step 9. PhoneWindow.getDecorView**

public class PhoneWindow extends Window implements MenuBuilder.Callback {

......

private DecorView mDecor;

......

@Override

public final View getDecorView() {

if (mDecor == null) {

installDecor();

}

return mDecor;

}

......

}

这个函数定义在文件frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindow.java中。

PhoneWindow类的成员函数getDecorView首先判断成员变量mDecor的值是否等于null。如果是的话，那么就说明当前正在处理的应用程序窗口还没有创建视图对象。这时候就会调用另外一个成员函数installDecor来创建这个视图对象。从前面的调用过程可以知道，当前正在处理的应用程序窗口已经创建过视图对象，因此，这里的成员变量mDecor的值不等于null，PhoneWindow类的成员函数getDecorView直接将它返回给调用者。

这一步执完成之后，返回到前面的Step 7中，即ActivityThread类的成员函数handleResumeActivity中，接下来就会继续调用当前正在激活的Activity组件的成员函数getWindowManager来获得一个本地窗口管理器。

**Step 10. Activity.getWindowManager**

public class Activity extends ContextThemeWrapper

implements LayoutInflater.Factory,

Window.Callback, KeyEvent.Callback,

OnCreateContextMenuListener, ComponentCallbacks {

......

private WindowManager mWindowManager;

......

public WindowManager getWindowManager() {

return mWindowManager;

}

......

}

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

从前面Android应用程序窗口（Activity）的运行上下文环境（Context）的创建过程分析一文可以知道，Activity类的成员变量mWindowManager指向的一是类型为LocalWindowManager的本地窗口管理器，Activity类的成员函数getWindowManager直接将它返回给调用者。

这一步执完成之后，返回到前面的Step 7中，即ActivityThread类的成员函数handleResumeActivity中，接下来就会继续调用前面所获得的一个LocalWindowManager对象的成员函数addView来为当前正在激活的Activity组件的应用程序窗口视图对象关联一个ViewRoot对象。

**Step 11. LocalWindowManager.addView**

public abstract class Window {

......

private class LocalWindowManager implements WindowManager {

......

public final void addView(View view, ViewGroup.LayoutParams params) {

......

mWindowManager.addView(view, params);

}

......

private final WindowManager mWindowManager;

......

}

......

}

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

从前面Android应用程序窗口（Activity）的窗口对象（Window）的创建过程分析一文可以知道，LocalWindowManager类的成员变量mWindowManager指向的是一个WindowManagerImpl对象，因此，LocalWindowManager类的成员函数addView接下来调用WindowManagerImpl类的成员函数addView来给参数view所描述的一个应用程序窗口视图对象关联一个ViewRoot对象。

**Step 12. WindowManagerImpl.addView**

public class WindowManagerImpl implements WindowManager {

......

public void addView(View view, ViewGroup.LayoutParams params)

{

addView(view, params, false);

}

......

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

{

......

final WindowManager.LayoutParams wparams

= (WindowManager.LayoutParams)params;

ViewRoot root;

View panelParentView = null;

synchronized (this) {

// Here's an odd/questionable case: if someone tries to add a

// view multiple times, then we simply bump up a nesting count

// and they need to remove the view the corresponding number of

// times to have it actually removed from the window manager.

// This is useful specifically for the notification manager,

// which can continually add/remove the same view as a

// notification gets updated.

int index = findViewLocked(view, false);

if (index >= 0) {

if (!nest) {

throw new IllegalStateException("View " + view

+ " has already been added to the window manager.");

}

root = mRoots[index];

root.mAddNesting++;

// Update layout parameters.

view.setLayoutParams(wparams);

root.setLayoutParams(wparams, true);

return;

}

// If this is a panel window, then find the window it is being

// attached to for future reference.

if (wparams.type >= WindowManager.LayoutParams.FIRST\_SUB\_WINDOW &&

wparams.type <= WindowManager.LayoutParams.LAST\_SUB\_WINDOW) {

final int count = mViews != null ? mViews.length : 0;

for (int i=0; i<count; i++) {

if (mRoots[i].mWindow.asBinder() == wparams.token) {

panelParentView = mViews[i];

}

}

}

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

root.mAddNesting = 1;

view.setLayoutParams(wparams);

if (mViews == null) {

index = 1;

mViews = new View[1];

mRoots = new ViewRoot[1];

mParams = new WindowManager.LayoutParams[1];

} else {

index = mViews.length + 1;

Object[] old = mViews;

mViews = new View[index];

System.arraycopy(old, 0, mViews, 0, index-1);

old = mRoots;

mRoots = new ViewRoot[index];

System.arraycopy(old, 0, mRoots, 0, index-1);

old = mParams;

mParams = new WindowManager.LayoutParams[index];

System.arraycopy(old, 0, mParams, 0, index-1);

}

index--;

mViews[index] = view;

mRoots[index] = root;

mParams[index] = wparams;

}

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

root.setView(view, wparams, panelParentView);

}

......

private View[] mViews;

private ViewRoot[] mRoots;

private WindowManager.LayoutParams[] mParams;

......

}

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

在WindowManagerImpl类中，两个参数版本的成员函数addView是通过调用三个参数版本的成同函数addView来实现的，因此，我们接下来就主要分析三个参数版本的成员函数addView的实现。

在分析WindowManagerImpl类的三个参数版本的成员函数addView的实现之前，我们首先了解一下WindowManagerImpl类是如何关联一个应用程序窗口视图对象（View对象）和一个ViewRoot对象的。一个View对象在与一个ViewRoot对象关联的同时，还会关联一个WindowManager.LayoutParams对象，这个WindowManager.LayoutParams对象是用来描述应用程序窗口视图的布局属性的。

WindowManagerImpl类有三个成员变量mViews、mRoots和mParams，它们分别是类型为View、ViewRoot和WindowManager.LayoutParams的数组。这三个数组的大小是始终保持相等的。这样， 有关联关系的View对象、ViewRoot对象和WindowManager.LayoutParams对象就会分别保存在数组mViews、mRoots和mParams的相同位置上，也就是说，mViews[i]、mRoots[i]和mParams[i]所描述的View对象、ViewRoot对象和WindowManager.LayoutParams对象是具有关联关系的。因此，WindowManagerImpl类的三个参数版本的成员函数addView在关联一个View对象、一个ViewRoot对象和一个WindowManager.LayoutParams对象的时候，只要分别将它们放在数组mViews、mRoots和mParams的相同位置上就可以了。

理解了一个View对象、一个ViewRoot对象和一个WindowManager.LayoutParams对象是如何关联之后，WindowManagerImpl类的三个参数版本的成员函数addView的实现就容易理解了。

参数view和参数params描述的就是要关联的View对象和WindowManager.LayoutParams对象。成员函数addView首先调用另外一个成员函数findViewLocked来检查参数view所描述的一个View对象是否已经存在于数组中mViews中了。如果已经存在的话，那么就说明该View对象已经关联过ViewRoot对象以及WindowManager.LayoutParams对象了。在这种情况下，如果参数nest的值等于false，那么成员函数addView是不允许重复对参数view所描述的一个View对象进行重新关联的。另一方面，如果参数nest的值等于true，那么成员函数addView只是重新修改参数view所描述的一个View对象及其所关联的一个ViewRoot对象内部使用的一个WindowManager.LayoutParams对象，即更新为参数params所描述的一个WindowManager.LayoutParams对象，这是通过调用它们的成员函数setLayoutParams来实现的。

如果参数view所描述的一个View对象还没有被关联过一个ViewRoot对象，那么成员函数addView就会创建一个ViewRoot对象，并且将它与参数view和params分别描述的一个View对象和一个WindowManager.LayoutParams对象保存在数组mViews、mRoots和mParams的相同位置上。注意，如果数组mViews、mRoots和mParams尚未创建，那么成员函数addView就会首先分别为它们创建一个大小为1的数组，以便可以用来分别保存所要关联的View对象、ViewRoot对象和WindowManager.LayoutParams对象。另一方面，如果数组mViews、mRoots和mParams已经创建，那么成员函数addView就需要分别将它们的大小增加1，以便可以在它们的末尾位置上分别保存所要关联的View对象、ViewRoot对象和WindowManager.LayoutParams对象。

还有另外一个需要注意的地方是当参数view描述的是一个子应用程序窗口的视图对象时，即WindowManager.LayoutParams对象wparams的成员变量type的值大于等于WindowManager.LayoutParams.FIRST\_SUB\_WINDOW并且小于等于WindowManager.LayoutParams.LAST\_SUB\_WINDOW时，那么成员函数addView还需要找到这个子视图对象的父视图对象panelParentView，这是通过遍历数组mRoots来查找的。首先，WindowManager.LayoutParams对象wparams的成员变量token指向了一个类型为W的Binder本地对象的一个IBinder接口，用来描述参数view所描述的一个子应用程序窗口视图对象所属的父应用程序窗口视图对象。其次，每一个ViewRoot对象都通过其成员变量mWindow来保存一个类型为W的Binder本地对象，因此，如果在数组mRoots中，存在一个ViewRoot对象，它的成员变量mWindow所描述的一个W对象的一个IBinder接口等于WindowManager.LayoutParams对象wparams的成员变量token所描述的一个IBinder接口时，那么就说明与该ViewRoot对象所关联的View对象即为参数view的父应用程序窗口视图对象。

成员函数addView为参数view所描述的一个View对象和参数params所描述的一个WindowManager.LayoutParams对象关联好一个ViewRoot对象root之后，最后还会将这个View对view象和这个WindowManager.LayoutParams对象，以及变量panelParentView所描述的一个父应用程序窗视图对象，保存在这个ViewRoot对象root的内部去，这是通过调用这个ViewRoot对象root的成员函数setView来实现的，因此，接下来我们就继续分析ViewRoot类的成员函数setView的实现。

**Step 13. ViewRoot.setView**

public final class ViewRoot extends Handler implements ViewParent,

View.AttachInfo.Callbacks {

......

final WindowManager.LayoutParams mWindowAttributes = new WindowManager.LayoutParams();

......

View mView;

......

final View.AttachInfo mAttachInfo;

......

boolean mAdded;

......

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

View panelParentView) {

synchronized (this) {

if (mView == null) {

mView = view;

mWindowAttributes.copyFrom(attrs);

......

mAttachInfo.mRootView = view;

.......

if (panelParentView != null) {

mAttachInfo.mPanelParentWindowToken

= panelParentView.getApplicationWindowToken();

}

mAdded = true;

......

requestLayout();

......

try {

res = sWindowSession.add(mWindow, mWindowAttributes,

getHostVisibility(), mAttachInfo.mContentInsets,

mInputChannel);

} catch (RemoteException e) {

mAdded = false;

mView = null;

......

throw new RuntimeException("Adding window failed", e);

} finally {

if (restore) {

attrs.restore();

}

}

......

}

......

}

}

......

}

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

参数view所描述的一个View对象会分别被保存在ViewRoot类的成员变量mView以及成员变量mAttachInfo所描述的一个AttachInfo的成员变量mRootView中，而参数attrs所描述的一个WindowManager.LayoutParams对象的内容会被拷贝到ViewRoot类的成员变量mWindowAttributes中去。

当参数panelParentView的值不等于null的时候，就表示参数view描述的是一个子应用程序窗口视图对象。在这种情况下，参数panelParentView描述的就是一个父应用程序窗口视图对象。这时候我们就需要获得用来描述这个父应用程序窗口视图对象的一个类型为W的Binder本地对象的IBinder接口，以便可以保存在ViewRoot类的成员变量mAttachInfo所描述的一个AttachInfo的成员变量mPanelParentWindowToken中去。这样以后就可以知道ViewRoot类的成员变量mView所描述的一个子应用程序窗口视图所属的父应用程序窗口视图是什么了。注意，通过调用参数panelParentView的所描述的一个View对象的成员函数getApplicationWindowToken即可以获得一个对应的W对象的IBinder接口。

上述操作执行完成之后，ViewRoot类的成员函数setView就可以将成员变量mAdded的值设置为true了，表示当前正在处理的一个ViewRoot对象已经关联好一个View对象了。接下来，ViewRoot类的成员函数setView还需要执行两个操作：

1. 调用ViewRoot类的另外一个成员函数requestLayout来请求对应用程序窗口视图的UI作第一次布局。
2. 调用ViewRoot类的静态成员变量sWindowSession所描述的一个类型为Session的Binder代理对象的成员函数add来请求WindowManagerService增加一个WindowState对象，以便可以用来描述当前正在处理的一个ViewRoot所关联的一个应用程序窗口。

至此，我们就分析完成Android应用程序窗口视图对象的创建过程了。在接下来的一篇文章中，我们将会继续分析Android应用程序窗口与WindowManagerService服务的连接过程，即Android应用程序窗口请求WindowManagerService为其增加一个WindowState对象的过程，而在接下来的两篇文章中，我们还会分析用来渲染Android应用程序窗口的Surface的创建过程，以及Android应用程序窗口的渲染过程。通过这三个过程的分析，我们就可以看到上述第1点和第2点提到的两个函数的执行过程，敬请期待！

上一篇：[窗口对象（Window）的创建过程分析](https://www.kancloud.cn/alex_wsc/androids/473771)下一篇：[与WindowManagerService服务的连接过程分析](https://www.kancloud.cn/alex_wsc/androids/473773)