http://blog.csdn.net/luoshengyang/article/details/8223770

在前文中，我们分析了Android应用程序窗口的运行上下文环境的创建过程。由此可知，每一个Activity组件都有一个关联的ContextImpl对象，同时，它还关联有一个Window对象，用来描述一个具体的应用程序窗口。由此又可知，Activity只不过是一个高度抽象的UI组件，它的具体UI实现其实是由其它的一系列对象来实现的。在本文中，我们就将详细分析Android应用程序窗口对象的创建过程。

       从前面[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)一文可以知道，在PHONE平台上，与Activity组件所关联的窗口对象的实际类型为PhoneWindow，后者是从Window类继承下来的。Activity、Window和PhoneWindow三个类的关系可以参考[Android应用程序窗口（Activity）实现框架简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/8170307)一文中的图3和图5。为了方便接下来描述类型为PhoneWindow的应用程序窗口的创建过程，我们将这两个图拿过来，如以下的图1和图2所示：

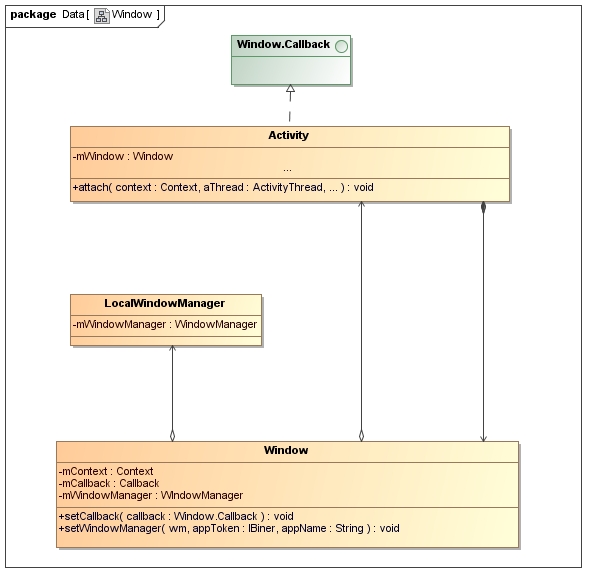


图1 Activity和Window的类关系图

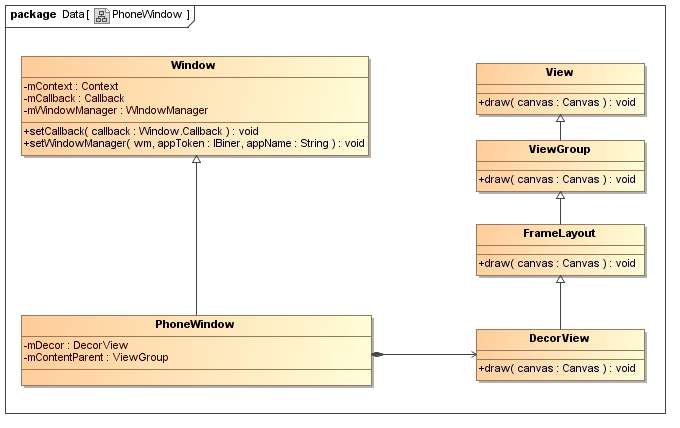


图2 Window和PhoneWindow的类关系图

       上述两个图中所涉及到的类的描述可以参考[Android应用程序窗口（Activity）实现框架简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/8170307)一文，本文主要从Android应用程序窗口的创建过程来理解Activity、Window和PhoneWindow三个类的关系。

       从[Android应用程序窗口（Activity）的运行上下文环境（Context）的创建过程分析](http://blog.csdn.net/luoshengyang/article/details/8201936)一文又可以知道，与Activity组件所关联的一个PhoneWindow对象是从Activity类的成员函数attach中创建的，如图3所示：

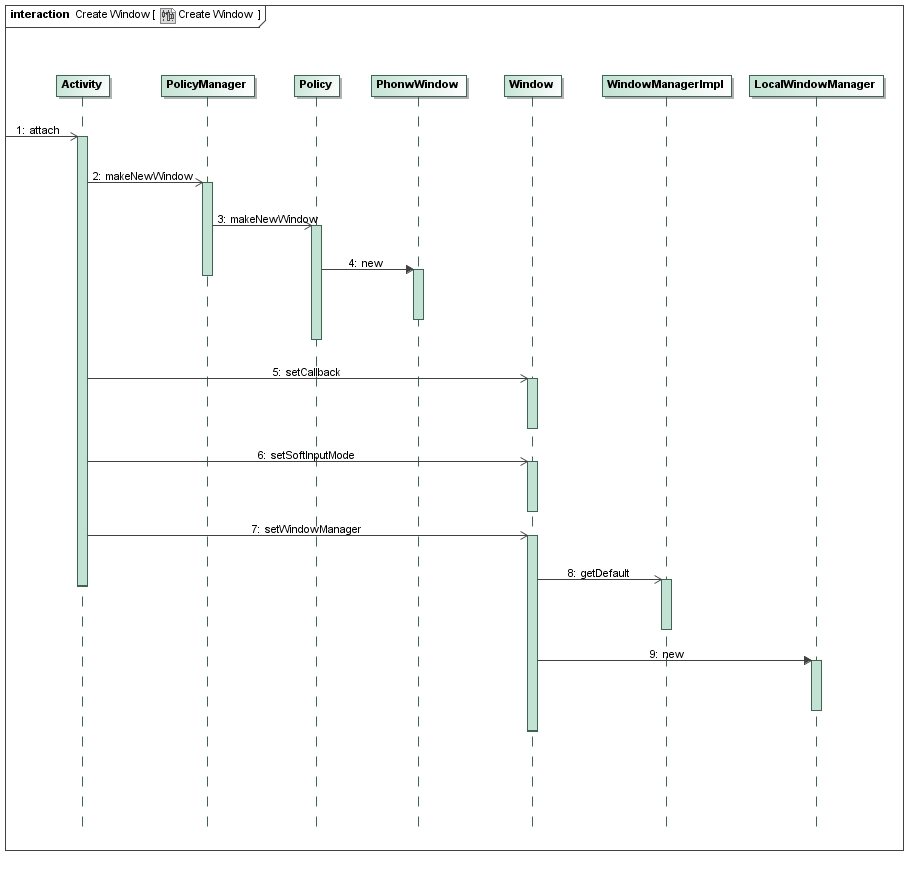


图3 Android应用程序窗口的创建过程

       这个过程可以分为9个步骤，接下来我们就详细分析每一个步骤。

       Step 1. Activity.attach

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8223770)

1. public class Activity extends ContextThemeWrapper
2. implements LayoutInflater.Factory,
3. Window.Callback, KeyEvent.Callback,
4. OnCreateContextMenuListener, ComponentCallbacks {
5. ......
7. private Window mWindow;
8. ......
10. final void attach(Context context, ActivityThread aThread,
11. Instrumentation instr, IBinder token, int ident,
12. Application application, Intent intent, ActivityInfo info,
13. CharSequence title, Activity parent, String id,
14. Object lastNonConfigurationInstance,
15. HashMap<String,Object> lastNonConfigurationChildInstances,
16. Configuration config) {
17. ......
19. mWindow = PolicyManager.makeNewWindow(this);
20. mWindow.setCallback(this);
21. if (info.softInputMode != WindowManager.LayoutParams.SOFT\_INPUT\_STATE\_UNSPECIFIED) {
22. mWindow.setSoftInputMode(info.softInputMode);
23. }
24. ......
26. mWindow.setWindowManager(null, mToken, mComponent.flattenToString());
27. ......
29. }
31. ......
32. }

public class Activity extends ContextThemeWrapper

implements LayoutInflater.Factory,

Window.Callback, KeyEvent.Callback,

OnCreateContextMenuListener, ComponentCallbacks {

......

private Window mWindow;

......

final void attach(Context context, ActivityThread aThread,

Instrumentation instr, IBinder token, int ident,

Application application, Intent intent, ActivityInfo info,

CharSequence title, Activity parent, String id,

Object lastNonConfigurationInstance,

HashMap<String,Object> lastNonConfigurationChildInstances,

Configuration config) {

......

mWindow = PolicyManager.makeNewWindow(this);

mWindow.setCallback(this);

if (info.softInputMode != WindowManager.LayoutParams.SOFT\_INPUT\_STATE\_UNSPECIFIED) {

mWindow.setSoftInputMode(info.softInputMode);

}

......

mWindow.setWindowManager(null, mToken, mComponent.flattenToString());

......

}

......

}

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

        在前面[Android应用程序窗口（Activity）的运行上下文环境（Context）的创建过程分析](http://blog.csdn.net/luoshengyang/article/details/8201936)一文中，我们已经分析过这个函数的实现了，这里我们只关注与应用程序窗口创建相关的代码。

        函数首先调用PolicyManager类的静态成员函数makeNewWindow来创建一个类型为PhoneWindow的应用程序窗口，并且保存在Activity类的成员变量mWindow中。有了这个类型为PhoneWindow的应用程序窗口，函数接下来还会调用它的成员函数setCallback、setSoftInputMode和setWindowManager来设置窗口回调接口、软键盘输入区域的显示模式和本地窗口管理器。

        PhoneWindow类的成员函数setCallback、setSoftInputMode和setWindowManager都是从父类Window继承下来的，因此，接下来我们就继续分析PolicyManager类的静态成员函数makeNewWindow，以及Window类的成员函数setCallback、setSoftInputMode和setWindowManager的实现。

        Step 2. PolicyManager.makeNewWindow

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8223770)

1. public final class PolicyManager {
2. private static final String POLICY\_IMPL\_CLASS\_NAME =
3. "com.android.internal.policy.impl.Policy";
5. private static final IPolicy sPolicy;
7. static {
8. // Pull in the actual implementation of the policy at run-time
9. try {
10. Class policyClass = Class.forName(POLICY\_IMPL\_CLASS\_NAME);
11. sPolicy = (IPolicy)policyClass.newInstance();
12. } catch (ClassNotFoundException ex) {
13. throw new RuntimeException(
14. POLICY\_IMPL\_CLASS\_NAME + " could not be loaded", ex);
15. } catch (InstantiationException ex) {
16. throw new RuntimeException(
17. POLICY\_IMPL\_CLASS\_NAME + " could not be instantiated", ex);
18. } catch (IllegalAccessException ex) {
19. throw new RuntimeException(
20. POLICY\_IMPL\_CLASS\_NAME + " could not be instantiated", ex);
21. }
22. }
24. ......
26. // The static methods to spawn new policy-specific objects
27. public static Window makeNewWindow(Context context) {
28. return sPolicy.makeNewWindow(context);
29. }
31. ......
32. }

public final class PolicyManager {

private static final String POLICY\_IMPL\_CLASS\_NAME =

"com.android.internal.policy.impl.Policy";

private static final IPolicy sPolicy;

static {

// Pull in the actual implementation of the policy at run-time

try {

Class policyClass = Class.forName(POLICY\_IMPL\_CLASS\_NAME);

sPolicy = (IPolicy)policyClass.newInstance();

} catch (ClassNotFoundException ex) {

throw new RuntimeException(

POLICY\_IMPL\_CLASS\_NAME + " could not be loaded", ex);

} catch (InstantiationException ex) {

throw new RuntimeException(

POLICY\_IMPL\_CLASS\_NAME + " could not be instantiated", ex);

} catch (IllegalAccessException ex) {

throw new RuntimeException(

POLICY\_IMPL\_CLASS\_NAME + " could not be instantiated", ex);

}

}

......

// The static methods to spawn new policy-specific objects

public static Window makeNewWindow(Context context) {

return sPolicy.makeNewWindow(context);

}

......

}

       这个函数定义在文件frameworks/base/core/java/com/android/internal/policy/PolicyManager.java中。

       PolicyManager是一个窗口管理策略类，它在第一次被使用的时候，就会创建一个Policy类实例，并且保存在静态成员变量sPolicy中，以后PolicyManager类的窗口管理策略就是通过这个Policy类实例来实现的，例如，PolicyManager类的静态成员函数makeNewWindow就是通过调用这个Policy类实例的成员函数makeNewWindow来创建一个具体的应用程序窗口的。

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

       Step 3. Policy.makeNewWindow

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8223770)

1. public class Policy implements IPolicy {
2. ......
4. public PhoneWindow makeNewWindow(Context context) {
5. return new PhoneWindow(context);
6. }
8. ......
9. }

public class Policy implements IPolicy {

......

public PhoneWindow makeNewWindow(Context context) {

return new PhoneWindow(context);

}

......

}

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

        Policy类的成员函数makeNewWindow的实现很简单，它只是创建了一个PhoneWindow对象，然后返回给调用者。

        接下来，我们就继续分析PhoneWindow类的构造函数的实现，以便可以了解一个类型为PhoneWindow的应用程序窗口的创建过程。

        Step 4.new PhoneWindow

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8223770)

1. public class PhoneWindow extends Window implements MenuBuilder.Callback {
2. ......
4. // This is the top-level view of the window, containing the window decor.
5. private DecorView mDecor;
7. // This is the view in which the window contents are placed. It is either
8. // mDecor itself, or a child of mDecor where the contents go.
9. private ViewGroup mContentParent;
10. ......
12. private LayoutInflater mLayoutInflater;
13. ......
15. public PhoneWindow(Context context) {
16. super(context);
17. mLayoutInflater = LayoutInflater.from(context);
18. }
20. ......
21. }

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 LayoutInflater mLayoutInflater;

......

public PhoneWindow(Context context) {

super(context);

mLayoutInflater = LayoutInflater.from(context);

}

......

}

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

       PhoneWindow类的构造函数很简单，它首先调用父类Window的构造函数来执行一些初始化操作，接着再调用LayoutInflater的静态成员函数from创建一个LayoutInflater实例，并且保存在成员变量mLayoutInflater中。这样，PhoneWindow类以后就可以通过成员变量mLayoutInflater来创建应用程序窗口的视图，这个视图使用类型为DecorView的成员变量mDecor来描述。PhoneWindow类还有另外一个类型为ViewGroup的成员变量mContentParent，用来描述一个视图容器，这个容器存放的就是成员变量mDecor所描述的视图的内容，不过这个容器也有可能指向的是mDecor本身。在后面的文章中，我们再详细分析类型为PhoneWindow的应用程序窗口的视图的创建过程。

      Window的构造函数定义在文件frameworks/base/core/java/android/view/Window.java中，它的实现很简单，只是初始化了其成员变量mContext，如下所示：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8223770)

1. public abstract class Window {
2. ......
4. private final Context mContext;
5. ......
7. public Window(Context context) {
8. mContext = context;
9. }
11. ......
12. }

public abstract class Window {

......

private final Context mContext;

......

public Window(Context context) {

mContext = context;

}

......

}

       从前面的调用过程可以知道，参数context描述的是正在启动的Activity组件，将它保存在Window类的成员变量mContext之后，Window类就可以通过它来访问与Activity组件相关的资源了。

       这一步执行完成之后，回到前面的Step 1中，即Activity类的成员函数attach中，接下来就会继续调用前面所创建的PhoneWindow对象从父类Window继承下来的成员函数setCallback来设置窗口回调接口，因此，接下来我们就继续分析Window类的成员函数setCallback的实现。

       Step 5. Window.setCallback

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8223770)

1. public abstract class Window {
2. ......
4. private Callback mCallback;
5. ......
7. /\*\*
8. \* Set the Callback interface for this window, used to intercept key
9. \* events and other dynamic operations in the window.
10. \*
11. \* @param callback The desired Callback interface.
12. \*/
13. public void setCallback(Callback callback) {
14. mCallback = callback;
15. }
17. ......
18. }

public abstract class Window {

......

private Callback mCallback;

......

/\*\*

\* Set the Callback interface for this window, used to intercept key

\* events and other dynamic operations in the window.

\*

\* @param callback The desired Callback interface.

\*/

public void setCallback(Callback callback) {

mCallback = callback;

}

......

}

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

        正在启动的Activity组件会将它所实现的一个Callback接口设置到与它所关联的一个PhoneWindow对象的父类Window的成员变量mCallback中去，这样当这个PhoneWindow对象接收到系统给它分发的IO输入事件，例如，键盘和触摸屏事件，转发给与它所关联的Activity组件处理，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。

        这一步执行完成之后，回到前面的Step 1中，即Activity类的成员函数attach中，接下来就会继续调用前面所创建的PhoneWindow对象从父类Window继承下来的成员函数setSoftInputMode来设置应用程序窗口的软键盘输入区域的显示模式，因此，接下来我们就继续分析Window类的成员函数setSoftInputMode的实现。

        Step 6. Window.setSoftInputMode

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8223770)

1. public abstract class Window {
2. ......
4. private boolean mHasSoftInputMode = false;
5. ......
7. public void setSoftInputMode(int mode) {
8. final WindowManager.LayoutParams attrs = getAttributes();
9. if (mode != WindowManager.LayoutParams.SOFT\_INPUT\_STATE\_UNSPECIFIED) {
10. attrs.softInputMode = mode;
11. mHasSoftInputMode = true;
12. } else {
13. mHasSoftInputMode = false;
14. }
15. if (mCallback != null) {
16. mCallback.onWindowAttributesChanged(attrs);
17. }
18. }
20. ......
21. }

public abstract class Window {

......

private boolean mHasSoftInputMode = false;

......

public void setSoftInputMode(int mode) {

final WindowManager.LayoutParams attrs = getAttributes();

if (mode != WindowManager.LayoutParams.SOFT\_INPUT\_STATE\_UNSPECIFIED) {

attrs.softInputMode = mode;

mHasSoftInputMode = true;

} else {

mHasSoftInputMode = false;

}

if (mCallback != null) {

mCallback.onWindowAttributesChanged(attrs);

}

}

......

}

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

       参数mode有SOFT\_INPUT\_STATE\_UNSPECIFIED、SOFT\_INPUT\_STATE\_UNCHANGED、SOFT\_INPUT\_STATE\_HIDDEN、SOFT\_INPUT\_STATE\_ALWAYS\_HIDDEN、SOFT\_INPUT\_STATE\_VISIBLE和SOFT\_INPUT\_STATE\_ALWAYS\_VISIBLE一共六个取值，用来描述窗口的软键盘输入区域的显示模式，它们的含义如下所示：

      1. **SOFT\_INPUT\_STATE\_UNSPECIFIED**：没有指定软键盘输入区域的显示状态。

      2. **SOFT\_INPUT\_STATE\_UNCHANGED**：不要改变软键盘输入区域的显示状态。

      3. **SOFT\_INPUT\_STATE\_HIDDEN**：在合适的时候隐藏软键盘输入区域，例如，当用户导航到当前窗口时。

      4. **SOFT\_INPUT\_STATE\_ALWAYS\_HIDDEN**：当窗口获得焦点时，总是隐藏软键盘输入区域。

      5. **SOFT\_INPUT\_STATE\_VISIBLE**：在合适的时候显示软键盘输入区域，例如，当用户导航到当前窗口时。

      6. **SOFT\_INPUT\_STATE\_ALWAYS\_VISIBLE**：当窗口获得焦点时，总是显示软键盘输入区域。

      当参数mode的值不等于SOFT\_INPUT\_STATE\_UNSPECIFIED时，就表示当前窗口被指定软键盘输入区域的显示模式，这时候Window类的成员函数setSoftInputMode就会将成员变量mHasSoftInputMode的值设置为true，并且将这个显示模式保存在用来描述窗口布局属性的一个WindowManager.LayoutParams对象的成员变量softInputMode中，否则的话，就会将成员变量mHasSoftInputMode的值设置为false。

      设置完成窗口的软键盘输入区域的显示模式之后，如果Window类的成员变量mCallback指向了一个窗口回调接口，那么Window类的成员函数setSoftInputMode还会调用它的成员函数onWindowAttributesChanged来通知与窗口所关联的Activity组件，它的窗口布局属性发生了变化。

       这一步执行完成之后，回到前面的Step 1中，即Activity类的成员函数attach中，接下来就会继续调用前面所创建的PhoneWindow对象从父类Window继承下来的成员函数setWindowManager来设置应用程序窗口的本地窗口管理器，因此，接下来我们就继续分析Window类的成员函数setWindowManager的实现。

       Step 7. Window.setWindowManager

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8223770)

1. public abstract class Window {
2. ......
4. private WindowManager mWindowManager;
5. private IBinder mAppToken;
6. private String mAppName;
7. ......
9. public void setWindowManager(WindowManager wm,
10. IBinder appToken, String appName) {
11. mAppToken = appToken;
12. mAppName = appName;
13. if (wm == null) {
14. wm = WindowManagerImpl.getDefault();
15. }
16. mWindowManager = new LocalWindowManager(wm);
17. }
19. ......
20. }

public abstract class Window {

......

private WindowManager mWindowManager;

private IBinder mAppToken;

private String mAppName;

......

public void setWindowManager(WindowManager wm,

IBinder appToken, String appName) {

mAppToken = appToken;

mAppName = appName;

if (wm == null) {

wm = WindowManagerImpl.getDefault();

}

mWindowManager = new LocalWindowManager(wm);

}

......

}

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

       参数appToken用来描述当前正在处理的窗口是与哪一个Activity组件关联的，它是一个Binder代理对象，引用了在ActivityManagerService这一侧所创建的一个类型为ActivityRecord的Binder本地对象。从前面[Android应用程序的Activity启动过程简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/6685853)一系列文章可以知道，每一个启动起来了的Activity组件在ActivityManagerService这一侧，都有一个对应的ActivityRecord对象，用来描述该Activity组件的运行状态。这个Binder代理对象会被保存在Window类的成员变量mAppToken中，这样当前正在处理的窗口就可以知道与它所关联的Activity组件是什么。

       参数appName用来描述当前正在处理的窗口所关联的Activity组件的名称，这个名称会被保存在Window类的成员变量mAppName中。

       参数wm用来描述一个窗口管理器。从前面的调用过程可以知道，这里传进来的参数wm的值等于null，因此，函数首先会调用WindowManagerImpl类的静态成员函数getDefault来获得一个默认的窗口管理器。有了这个窗口管理器之后，函数接着再使用它来创建一个本地窗口管理器，即一个LocalWindowManager对象，用来维护当前正在处理的应用程序窗口。

       接下来，我们首先分析WindowManagerImpl类的静态成员函数getDefault的实现，接着再分析本地窗口管理器的创建过程，即LocalWindowManager类的构造函数的实现。

       Step 8. WindowManagerImpl.getDefault

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8223770)

1. public class WindowManagerImpl implements WindowManager {
2. ......
4. public static WindowManagerImpl getDefault()
5. {
6. return mWindowManager;
7. }
9. ......
11. private static WindowManagerImpl mWindowManager = new WindowManagerImpl();
12. }

public class WindowManagerImpl implements WindowManager {

......

public static WindowManagerImpl getDefault()

{

return mWindowManager;

}

......

private static WindowManagerImpl mWindowManager = new WindowManagerImpl();

}

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

       WindowManagerImpl类的静态成员函数getDefault的实现很简单，它只是将静态成员变量mWindowManager所指向的一个WindowManagerImpl对象返回给调用者，这个WindowManagerImpl对象实现了WindowManager接口，因此，它就可以用来管理应用程序窗口。

      这一步执行完成之后，回到前面的Step 7中，即Window类的成员函数setWindowManager中，接下来就会使用前面所获得一个WindowManagerImpl对象来创建一个本地窗口管理器，即一个LocalWindowManager对象。

      Step 9.new LocalWindowManager

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8223770)

1. public abstract class Window {
2. ......
4. private final Context mContext;
5. ......
7. private class LocalWindowManager implements WindowManager {
8. LocalWindowManager(WindowManager wm) {
9. mWindowManager = wm;
10. mDefaultDisplay = mContext.getResources().getDefaultDisplay(
11. mWindowManager.getDefaultDisplay());
12. }
14. ......
16. private final WindowManager mWindowManager;
18. private final Display mDefaultDisplay;
19. }
21. ......
22. }

public abstract class Window {

......

private final Context mContext;

......

private class LocalWindowManager implements WindowManager {

LocalWindowManager(WindowManager wm) {

mWindowManager = wm;

mDefaultDisplay = mContext.getResources().getDefaultDisplay(

mWindowManager.getDefaultDisplay());

}

......

private final WindowManager mWindowManager;

private final Display mDefaultDisplay;

}

......

}

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

       LocalWindowManager类的构造函数首先将参数wm所描述的一个WindowManagerImpl对象保存它的成员变量mWindowManager中，这样以后就将窗口管理工作交给它来处理。

       LocalWindowManager类的构造函数接着又通过成员变量mWindowManager所描述的一个WindowManagerImpl对象的成员函数getDefaultDisplay来获得一个Display对象，用来描述系统屏幕属性。

       由于前面所获得的Display对象描述的是全局的屏幕属性，而当前正在处理的窗口可能配置了一些可自定义的屏幕属性，因此，LocalWindowManager类的构造函数需要进一步地调整前面所获得的Display对象所描述的屏幕属性，以便可以适合当前正在处理的窗口使用。LocalWindowManager类的构造函数首先通过外部类Window的成员变量mContext的成员函数getResources来获得一个Resources对象，接着再调用这个Resources对象的成员函数getDefaultDisplay来调整前面所获得的Display对象所描述的屏幕属性。最终调整完成的Display对象就保存在LocalWindowManager类的成员变量mDefaultDisplay中。

       从前面的Step 4可以知道，类Window的成员变量mContext描述的是与当前窗口所关联的一个Activity组件。Activity类的成员函数getResources是从父类ContextWrapper继续下来的，它实现在文件frameworks/base/core/java/android/content/ContextWrapper.java中，如下所示：

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8223770)

1. public class ContextWrapper extends Context {
2. Context mBase;
3. ......
5. @Override
6. public Resources getResources()
7. {
8. return mBase.getResources();
9. }
11. ......
12. }

public class ContextWrapper extends Context {

Context mBase;

......

@Override

public Resources getResources()

{

return mBase.getResources();

}

......

}

       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](http://blog.csdn.net/luoshengyang/article/details/8201936)一文可以知道，ContextWrapper类的成员变量mBase指向的是一个ContextImpl对象，用来描述一个Activity组件的运行上下文环境。通过调用这个ContextImpl对象的成员函数getResources，就可以获得与一个Resources对象，而通过这个Resources对象，就可以访问一个Activity组件的资源信息，从而可以获得它所配置的屏幕属性。

       至此，我们就分析完成一个Activity组件所关联的应用程序窗口对象的创建过程了。从分析的过程可以知道：

      1. 一个Activity组件所关联的应用程序窗口对象的类型为PhoneWindow。最终调用的是PhoneWindow的构造函数

      2. 这个类型为PhoneWindow的应用程序窗口是通过一个类型为LocalWindowManager的本地窗口管理器来维护的。LocalWindowManager变量维护着PhoneWindow变量，体现在：上面生成的mWidow调用的SetWindowManager方法，最终生成了一个LocalWindowManager的对象。意思就是由LocalWindowManager来做PhoneWindow的窗口管理器。

      3. 这个类型为LocalWindowManager的本地窗口管理器又是通过一个类型为WindowManagerImpl的窗口管理器来维护应用程序窗口的。

      4. 这个类型为PhoneWindow的应用程序窗口内部有一个类型为DecorView的视图对象，这个视图对象才是真正用来描述一个Activity组件的UI的。