http://blog.csdn.net/luoshengyang/article/details/8201936

在前文中，我们简要介绍了Android应用程序窗口的框架。Android应用程序窗口在运行的过程中，需要访问一些特定的资源或者类。这些特定的资源或者类构成了Android应用程序的运行上下文环境，Android应用程序窗口可以通过一个Context接口来访问它，这个Context接口也是我们在开发应用程序时经常碰到的。在本文中，我们就将详细分析Android应用程序窗口的运行上下文环境的创建过程。

        在前面[Android应用程序窗口（Activity）实现框架简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/8170307)一文中提到，Android应用程序窗口的运行上下文环境是通过ContextImpl类来描述的，即每一个Activity组件都关联有一个ContextImpl对象。ContextImpl类继承了Context类，它与Activity组件的关系如图1所示：

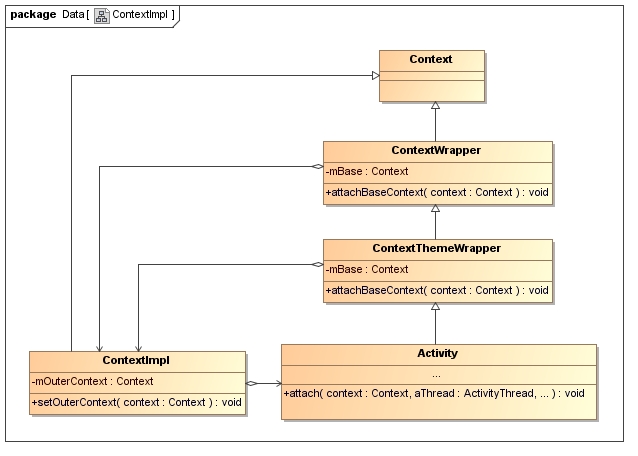


图1 ContextImpl类与Activity类的关系图

        这个类图在设计模式里面就可以称为**装饰模式**。Activity组件通过其父类ContextThemeWrapper和ContextWrapper的成员变量mBase来引用了一个ContextImpl对象，这样，Activity组件以后就可以通过这个ContextImpl对象来执行一些具体的操作，例如，[启动Service组件](http://blog.csdn.net/luoshengyang/article/details/6677029)、[注册广播接收者](http://blog.csdn.net/luoshengyang/article/details/6737352)和[启动Content Provider组件](http://blog.csdn.net/luoshengyang/article/details/6963418)等操作。同时，ContextImpl类又通过自己的成员变量mOuterContext来引用了与它关联的一个Activity组件，这样，ContextImpl类也可以将一些操作转发给Activity组件来处理。

        在前面[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)一文中，我们已经详细分析过一个Activity组件的启动过程了。在这个启动过程中，最后一步便是通过ActivityThread类的成员函数performLaunchActivity在应用程序进程中创建一个Activity实例，并且为它设置运行上下文环境，即为它创建一个ContextImpl对象。接下来，我们就从ActivityThread类的成员函数performLaunchActivity开始，分析一个Activity实例的创建过程，以便可以从中了解它的运行上下文环境的创建过程，如图2所示：

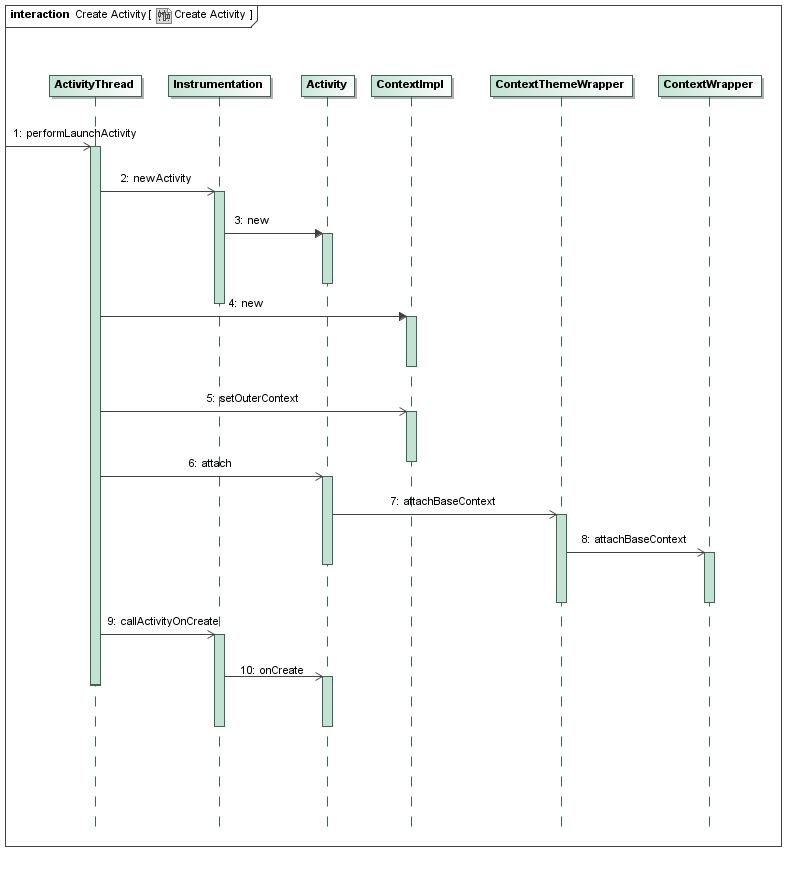


图2 Android应用程序窗口的运行上下文环境的创建过程

        这个过程一共分为10个步骤，接下来我们就详细分析每一个步骤。

        Step 1. ActivityThread.performLaunchActivity

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8201936)

1. public final class ActivityThread {
2. ......
4. Instrumentation mInstrumentation;
5. ......
7. private final Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
8. ......
10. ComponentName component = r.intent.getComponent();
11. ......
13. Activity activity = null;
14. try {
15. java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
16. activity = mInstrumentation.newActivity(
17. cl, component.getClassName(), r.intent);
18. ......
19. } catch (Exception e) {
20. ......
21. }
23. try {
24. Application app = r.packageInfo.makeApplication(false, mInstrumentation);
25. ......
27. if (activity != null) {
28. ContextImpl appContext = new ContextImpl();
29. ......
30. appContext.setOuterContext(activity);
31. ......
32. Configuration config = new Configuration(mConfiguration);
33. ......
35. activity.attach(appContext, this, getInstrumentation(), r.token,
36. r.ident, app, r.intent, r.activityInfo, title, r.parent,
37. r.embeddedID, r.lastNonConfigurationInstance,
38. r.lastNonConfigurationChildInstances, config);
39. ......
41. mInstrumentation.callActivityOnCreate(activity, r.state);
43. ......
44. }
46. ......
47. } catch (SuperNotCalledException e) {
48. ......
49. } catch (Exception e) {
50. ......
51. }
53. return activity;
54. }
55. }

public final class ActivityThread {

......

Instrumentation mInstrumentation;

......

private final Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {

......

ComponentName component = r.intent.getComponent();

......

Activity activity = null;

try {

java.lang.ClassLoader cl = r.packageInfo.getClassLoader();

activity = mInstrumentation.newActivity(

cl, component.getClassName(), r.intent);

......

} catch (Exception e) {

......

}

try {

Application app = r.packageInfo.makeApplication(false, mInstrumentation);

......

if (activity != null) {

ContextImpl appContext = new ContextImpl();

......

appContext.setOuterContext(activity);

......

Configuration config = new Configuration(mConfiguration);

......

activity.attach(appContext, this, getInstrumentation(), r.token,

r.ident, app, r.intent, r.activityInfo, title, r.parent,

r.embeddedID, r.lastNonConfigurationInstance,

r.lastNonConfigurationChildInstances, config);

......

mInstrumentation.callActivityOnCreate(activity, r.state);

......

}

......

} catch (SuperNotCalledException e) {

......

} catch (Exception e) {

......

}

return activity;

}

}

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

要启动的Activity组件的类名保存在变量component。有了这个类名之后，函数就可以调用ActivityThread类的成员变量mInstrumentation所描述一个Instrumentation对象的成员函数newActivity来创建一个Activity组件实例了，并且保存变量activity中。Instrumentation类是用来记录应用程序与系统的交互过程的，在接下来的Step 2中，我们再分析它的成员函数newActivity的实现。

       创建好了要启动的Activity组件实例之后，函数接下来就可以对它进行初始化了。初始化一个Activity组件实例需要一个Application对象app、一个ContextImpl对象appContext以及一个Configuration对象config，它们分别用来描述该Activity组件实例的应用程序信息、运行上下文环境以及配置信息。这里我们主要关心运行上下文环境的创建过程，即ContextImpl对象appContext的创建过程，这个过程我们在接下来的Step 4中再分析。

       ContextImpl对象appContext创建完成之后，函数就会调用它的成员函数setOuterContext来将与它所关联的Activity组件实例activity保存在它的内部。这样，ContextImpl对象appContext以后就可以访问与它所关联的Activity组件的属性或者方法。在接下来的Step 5中，我们再分析ContextImpl类的成员函数setOuterContext的实现。

       接着，函数就调用Activity组件实例activity的成员函数attach来将前面所创建的ContextImpl对象appContext以及Application对象app和Configuration对象config保存在它的内部。这样，Activity组件实例activity就可以访问它的运行上下文环境信息了。在接下来的Step 6中，我们再分析Activity类的成员函数attach的实现。

       最后，函数又通过调用ActivityThread类的成员变量mInstrumentation所描述一个Instrumentation对象的成员函数callActivityOnCreate来通知Activity组件实例activity，它已经被创建和启动起来了。在接下来的Step 9中，我们再分析它的成员函数callActivityOnCreate的实现。

       接下来，我们就分别分析Instrumentation类的成员函数newActivity、ContextImpl类的构造函数以及成员函数setOuterContext、Activity类的成员函数attach和Instrumentation类的成员函数callActivityOnCreate的实现。

       Step 2. Instrumentation.newActivity

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8201936)

1. public class Instrumentation {
2. ......
4. public Activity newActivity(ClassLoader cl, String className,
5. Intent intent)
6. throws InstantiationException, IllegalAccessException,
7. ClassNotFoundException {
8. return (Activity)cl.loadClass(className).newInstance();
9. }
11. ......
12. }

public class Instrumentation {

......

public Activity newActivity(ClassLoader cl, String className,

Intent intent)

throws InstantiationException, IllegalAccessException,

ClassNotFoundException {

return (Activity)cl.loadClass(className).newInstance();

}

......

}

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

        参数cl描述的是一个类加载器，而参数className描述的要加载的类。以className为参数来调用cl描述的是一个类加载器的成员函数loadClass，就可以得到一个Class对象。由于className描述的是一个Activity子类，因此，当函数调用前面得到的Class对象的成员函数newInstance的时候，就会创建一个Activity子类实例。这个Activity实例就是用来描述在前面Step 1中所要启动的Activity组件的。

       Activity子类实例在创建的过程，会调用父类Activity的默认构造函数，以便可以完成Activity组件的创建过程。

       Step 3.new Activity

       Activity类定义在文件frameworks/base/core/java/android/app/Activity.java中，它没有定义自己的构造函数，因此，系统就会为它提供一个默认的构造函数。一般来说，一个类的构造函数是用来初始化该类的实例的，但是，系统为Activity类提供的默认构造函数什么也不做，也就是说，Activity类实例在创建的时候，还没有执行实质的初始化工作。这个初始化工作要等到Activity类的成员函数attach被调用的时候才会执行。在后面的Step 6中，我们就会看到Activity类的成员函数attach是如何初始化一个Activity类实例的。

      这一步执行完成之后，回到前面的Step 1中，即ActivityThread类的成员函数performLaunchActivity中，接下来就会调用ContextImpl类的构造函数来创建一个ContextImpl对象，以便可以用来描述正在启动的Activity组件的运行上下文信息。

      Step 4.new ContextImpl

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8201936)

1. class ContextImpl extends Context {
2. ......
4. private Context mOuterContext;
5. ......
7. ContextImpl() {
8. // For debug only
9. //++sInstanceCount;
10. mOuterContext = this;
11. }
13. ......
14. }

class ContextImpl extends Context {

......

private Context mOuterContext;

......

ContextImpl() {

// For debug only

//++sInstanceCount;

mOuterContext = this;

}

......

}

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

       ContextImpl类的成员变量mOuterContext的类型为Context。当一个ContextImpl对象是用来描述一个Activity组件的运行上下文环境时，那么它的成员变量mOuterContext指向的就是该Activity组件。由于一个ContextImpl对象在创建的时候，并没有参数用来指明它是用来描述一个Activity组件的运行上下文环境，因此，这里就暂时将它的成员变量mOuterContext指向它自己。在接下来的Step 5中，我们就会看到，一个ContextImpl对象所关联的一个Activity组件是通过调用ContextImpl类的成员函数setOuterContext来设置的。

      这一步执行完成之后，回到前面的Step 1中，即ActivityThread类的成员函数performLaunchActivity中，接下来就会调用ContextImpl类的成员函数setOuterContext来设置前面所创建一个ContextImpl对象所关联的一个Activity组件，即正在启动的Activity组件。

      Step 5. ContextImpl.setOuterContext

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8201936)

1. class ContextImpl extends Context {
2. ......
4. private Context mOuterContext;
5. ......
7. final void setOuterContext(Context context) {
8. mOuterContext = context;
9. }
11. ......
12. }

class ContextImpl extends Context {

......

private Context mOuterContext;

......

final void setOuterContext(Context context) {

mOuterContext = context;

}

......

}

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

       参数context描述的是一个正在启动的Activity组件，ContextImpl类的成员函数setOuterContext只是简单地将它保存在成员变量mContext中，以表明当前正在处理的一个ContextImpl对象是用来描述一个Activity组件的运行上下文环境的。

       这一步执行完成之后，回到前面的Step 1中，即ActivityThread类的成员函数performLaunchActivity中，接下来就会调用Activity类的成员函数attach来初始化正在启动的Activity组件，其中，就包括设置正在启动的Activity组件的运行上下文环境。

       Step 6. Activity.attach

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8201936)

1. public class Activity extends ContextThemeWrapper
2. implements LayoutInflater.Factory,
3. Window.Callback, KeyEvent.Callback,
4. OnCreateContextMenuListener, ComponentCallbacks {
5. ......
7. private Application mApplication;
8. ......
10. /\*package\*/ Configuration mCurrentConfig;
11. ......
13. private Window mWindow;
15. private WindowManager mWindowManager;
16. ......
18. final void attach(Context context, ActivityThread aThread,
19. Instrumentation instr, IBinder token, int ident,
20. Application application, Intent intent, ActivityInfo info,
21. CharSequence title, Activity parent, String id,
22. Object lastNonConfigurationInstance,
23. HashMap<String,Object> lastNonConfigurationChildInstances,
24. Configuration config) {
25. attachBaseContext(context);
27. mWindow = PolicyManager.makeNewWindow(this);
28. mWindow.setCallback(this);
29. if (info.softInputMode != WindowManager.LayoutParams.SOFT\_INPUT\_STATE\_UNSPECIFIED) {
30. mWindow.setSoftInputMode(info.softInputMode);
31. }
32. ......
34. mApplication = application;
35. ......
37. mWindow.setWindowManager(null, mToken, mComponent.flattenToString());
38. ......
40. mWindowManager = mWindow.getWindowManager();
41. mCurrentConfig = config;
42. }
44. ......
45. }

public class Activity extends ContextThemeWrapper

implements LayoutInflater.Factory,

Window.Callback, KeyEvent.Callback,

OnCreateContextMenuListener, ComponentCallbacks {

......

private Application mApplication;

......

/\*package\*/ Configuration mCurrentConfig;

......

private Window mWindow;

private WindowManager mWindowManager;

......

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) {

attachBaseContext(context);

mWindow = PolicyManager.makeNewWindow(this);

mWindow.setCallback(this);

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

mWindow.setSoftInputMode(info.softInputMode);

}

......

mApplication = application;

......

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

......

mWindowManager = mWindow.getWindowManager();

mCurrentConfig = config;

}

......

}

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

        函数首先调用从父类ContextThemeWrapper继承下来的成员函数attachBaseConext来设置运行上下文环境，即将参数context所描述的一个ContextImpl对象保存在内部。在接下来的Step 7中，我们再分析ContextThemeWrapper类的成员函数attachBaseConext的实现。

        函数接下来调用PolicyManager类的静态成员函数makeNewWindow来创建了一个PhoneWindow，并且保存在Activity类的成员变量mWindow中。这个PhoneWindow是用来描述当前正在启动的应用程序窗口的。这个应用程序窗口在运行的过程中，会接收到一些事件，例如，键盘、触摸屏事件等，这些事件需要转发给与它所关联的Activity组件处理，这个转发操作是通过一个Window.Callback接口来实现的。由于Activity类实现了Window.Callback接口，因此，函数就可以将当前正在启动的Activity组件所实现的一个Window.Callback接口设置到前面创建的一个PhoneWindow里面去，这是通过调用Window类的成员函数setCallback来实现的。

       参数info指向的是一个ActivityInfo对象，用来描述当前正在启动的Activity组件的信息。其中，这个ActivityInfo对象的成员变量softInputMode用来描述当前正在启动的一个Activity组件是否接受软键盘输入。如果接受的话，那么它的值就不等于WindowManager.LayoutParams.SOFT\_INPUT\_STATE\_UNSPECIFIED，并且描述的是当前正在启动的Activity组件所接受的软键盘输入模式。这个软键盘输入模式设置到前面所创建的一个PhoneWindow对象内部去，这是通过调用Window类的成员函数setSoftInputMode来实现的。

       在Android系统中，每一个应用程序窗口都需要由一个窗口管理者来管理，因此，函数再接下来就会调用前面所创建的一个PhoneWindow对象从父类Window继承下来的成员函数setWindowManager来为它设置一个合适的窗口管理者。这个窗口管理者设置完成之后，就可以通过调用Window类的成员函数getWindowManager来获得。获得这个窗口管理者之后，函数就将它保存在Activity类的成员变量mWindowManager中。这样，当前正在启动的Activity组件以后就可以通过它的成员变量mWindowManager来管理与它所关联的窗口。

       除了创建和初始化一个PhoneWindow之外，函数还会分别把参数application和config所描述的一个Application对象和一个Configuration对象保存在Activity类的成员变量mApplication和mCurrentConfig中。这样，当前正在启动的Activity组件就可以访问它的应用程序信息以及配置信息。

      在接下来的一篇文章中，我们再详细分析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，以便可以了解应用程序窗口的创建过程。

      接下来，我们继续分析ContextThemeWrapper类的成员函数attachBaseConext的实现，以便可以继续了解一个应用程序窗口的运行上下文环境的设置过程。

      Step 7. ContextThemeWrapper.attachBaseConext

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8201936)

1. public class ContextThemeWrapper extends ContextWrapper {
2. private Context mBase;
3. ......
5. @Override protected void attachBaseContext(Context newBase) {
6. super.attachBaseContext(newBase);
7. mBase = newBase;
8. }
10. ......
11. }

public class ContextThemeWrapper extends ContextWrapper {

private Context mBase;

......

@Override protected void attachBaseContext(Context newBase) {

super.attachBaseContext(newBase);

mBase = newBase;

}

......

}

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

        ContextThemeWrapper类用来维护一个应用程序窗口的主题，而用来描述这个应用程序窗口的运行上下文环境的一个ContextImpl对象就保存在ContextThemeWrapper类的成员函数mBase中。

        ContextThemeWrapper类的成员函数attachBaseConext的实现很简单，它首先调用父类ContextWrapper的成员函数attachBaseConext来将参数newBase所描述的一个ContextImpl对象保存到父类ContextWrapper中去，接着再将这个ContextImpl对象保存在ContextThemeWrapper类的成员变量mBase中。

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

        Step 8. ContextWrapper.attachBaseConext

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8201936)

1. public class ContextWrapper extends Context {
2. Context mBase;
3. ......
5. protected void attachBaseContext(Context base) {
6. if (mBase != null) {
7. throw new IllegalStateException("Base context already set");
8. }
9. mBase = base;
10. }
12. ......
13. }

public class ContextWrapper extends Context {

Context mBase;

......

protected void attachBaseContext(Context base) {

if (mBase != null) {

throw new IllegalStateException("Base context already set");

}

mBase = base;

}

......

}

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

        ContextWrapper类只是一个代理类，它只是简单地封装了对其成员变量mBase所描述的一个Context对象的操作。ContextWrapper类的成员函数attachBaseConext的实现很简单，它只是将参数base所描述的一个ContextImpl对象保存在成员变量mBase中。这样，ContextWrapper类就可以将它的功能交给ContextImpl类来具体实现。

        这一步执行完成之后，当前正在启动的Activity组件的运行上下文环境就设置完成了，回到前面的Step 1中，即ActivityThread类的成员函数performLaunchActivity中，接下来就会调用Instrumentation类的成员函数callActivityOnCreate来通知当前正在启动的Activity组件，它已经创建和启动完成了。

        Step 9. Instrumentation.callActivityOnCreate

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8201936)

1. public class Instrumentation {
2. ......
4. public void callActivityOnCreate(Activity activity, Bundle icicle) {
5. ......
7. activity.onCreate(icicle);
9. ......
10. }
12. ......
13. }

public class Instrumentation {

......

public void callActivityOnCreate(Activity activity, Bundle icicle) {

......

activity.onCreate(icicle);

......

}

......

}

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

       函数主要就是调用当前正在启动的Activity组件的成员函数onCreate，用来通知它已经成功地创建和启动完成了。

      Step 10. Activity.onCreate

**[java]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8201936)

1. public class Activity extends ContextThemeWrapper
2. implements LayoutInflater.Factory,
3. Window.Callback, KeyEvent.Callback,
4. OnCreateContextMenuListener, ComponentCallbacks {
5. ......
7. boolean mCalled;
8. ......
10. /\*package\*/ boolean mVisibleFromClient = true;
11. ......
13. protected void onCreate(Bundle savedInstanceState) {
14. mVisibleFromClient = !mWindow.getWindowStyle().getBoolean(
15. com.android.internal.R.styleable.Window\_windowNoDisplay, false);
16. mCalled = true;
17. }
19. ......
20. }

public class Activity extends ContextThemeWrapper

implements LayoutInflater.Factory,

Window.Callback, KeyEvent.Callback,

OnCreateContextMenuListener, ComponentCallbacks {

......

boolean mCalled;

......

/\*package\*/ boolean mVisibleFromClient = true;

......

protected void onCreate(Bundle savedInstanceState) {

mVisibleFromClient = !mWindow.getWindowStyle().getBoolean(

com.android.internal.R.styleable.Window\_windowNoDisplay, false);

mCalled = true;

}

......

}

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

       一般来说，我们都是通过定义一个Activity子类来实现一个Activity组件的。重写父类Activity的某些成员函数的时候，必须要回调父类Activity的这些成员函数。例如，当Activity子类在重写父类Activity的成员函数onCreate时，就必须回调父类Activity的成员函数onCreate。这些成员函数被回调了之后，Activity类就会将其成员变量mCalled的值设置为true。这样，Activity类就可以通过其成员变量mCalled来检查其子类在重写它的某些成员函数时，是否正确地回调了父类的这些成员函数。

       Activity类的另外一个成员变量mVisibleFromClient用来描述一个应用程序窗口是否是可见的。如果是可见的，那么它的值就会等于true。当Activity类的成员函数onCreate被其子类回调时，它就会检查对应的应用程序窗口的主题属性android:windowNoDisplay的值是否等于true。如果等于true的话，那么就说明当前所启动的应用程序窗口是不可见的，这时候Activity类的成员变量mVisibleFromClient的值就会被设置为false，否则的话，就会被设置为true。

      Activity子类在重写成员函数onCreate的时候，一般都会调用父类Activity的成员函数setContentView来为为当前正启动的应用程序窗口创建视图（View）。在接下来的文章中，我们再详细描述应用程序窗口的视图的创建过程。

      至此，一个Activity组件的创建过程，以及它的运行上下文环境的创建过程，就分析完成了。这个过程比较简单，我们是从中获得以下三点信息：

      1. 一个Android应用窗口的运行上下文环境是使用一个ContextImpl对象来描述的，这个ContextImpl对象会分别保存在Activity类的父类ContextThemeWrapper和ContextWrapper的成员变量mBase中，即ContextThemeWrapper类和ContextWrapper类的成员变量mBase指向的是一个ContextImpl对象。

      2. Activity组件在创建过程中，即在它的成员函数attach被调用的时候，会创建一个PhoneWindow对象，并且保存在成员变量mWindow中，用来描述一个具体的Android应用程序窗口。

      3.  Activity组件在创建的最后，即在它的子类所重写的成员函数onCreate中，会调用父类Activity的成员函数setContentView来创建一个Android应用程序窗口的视图。