1. Activity组件在启动的过程中，系统会为它创建一个ContextImpl对象，用来描述它的运行上下文环境。也就是说上下文环境是系统为应用程序创建的，而且上下文环境也是一个对象。如果想区分context之间的不同，只要区分context对象的不同即可。既然系统是在每个Activity启动的时候为其创建一个ContextImpl对象，所以没有销毁的Activities之间的ContextImpl对象不是一个，互相之间的上下文环境并不完全相同，或者说不指向同一个上下文环境。当Activity销毁的时候，ContextImpl对象由于失去了Activity的引用，也就被回收了。
2. Activity类还有另外一个类型为WindowManager的成员变量 mWindowManager，它实际上指向的一个LocalWindowManager对象。LocalWindowManager类是用来管理应用程 序窗口的，例如，用来维护应用程序窗口内部的视图（View）。LocalWindowManager类有一个类型为WindowManager的成员变 量mWindowManager，它实际上指向的是一个WindowManagerImpl对象。系统通过调用WindowManagerImpl类的静态成员函数getDefault来获得一个WindowManagerImpl对象，然后保存在LocalWindowManager类的成员变量 mWindowManager中。这样，LocalWindowManager类就可以通过WindowManagerImpl类来真正实现管理应用程序窗口的功能。三者之间的关系见下图：

这里必须先明确一个观念，Activity不是视图，Activity包含视图，并且通过自己的成员变量mWindowManager来管理视图。

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

每一个Activity对应一个window，那么同时存在多个Activities对象，就会有多个window对象存在。那这么来说，不是一个View对应一个window，而是一个Activity对应一个window，也就是一个完整的View Hierarchy对应一个window，这个完整的View Hierarchy中的所有View均绘制在这一个window上。

Window类中有三个成员变量需要说明：

1. mContext：存储的是window宿主Activity对象的引用
2. mCallBack：由宿主Activity实现。这样window接收到的事件就可以通过该回调函数传递给Activity来处理。
3. mWindowManager：存储的是宿主Activity的成员变量mWindowManager的引用。也就是说Activity和window使用同一个WindowManager。但是为什么两者都要拿到这个WindowManager？到底是window自己来管理UI视图，还是Activity来管理，还是两者同时来管理？
4. Activity包含的两个成员变量window manager和window均是系统为其创建的。
5. IO事件的起点：系统输入管理器InputManager。InputManager用来获取键盘、触摸屏等的输入事件。然后ViewRoot将事件发送给主线程的消息队列，等待下一步处理。

键盘、触摸屏等输入事件的处理和绘制UI的操作在应用程序进程的主线程中进行。

1. ViewRoot也是每一个Activity对应一个，不是所有Activity共用一个。
2. 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、注册广播接收者和启动Content Provider组件等操作。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。

Activity、Service、Provider、Broadcast互相之间是独立的，Context是连接各个组件的纽带。所以Context中保存了自己对应的当前Activity的引用。

创建的ContextImpl类

Instrumentation类是用来记录应用程序与系统的交互过程的。

1. 所有Activity绑定的WindowManagerImpl是一个。Window和ViewRoot是每个Activity一套，自己用自己的。
2. Activity初始化不是通过构造方法，而是通过自己的attach方法。
   1. 调用父类方法attachBaseContext()，给ContextThemeWrapper和ContextWrapper绑定上下文对象ContextImpl。ContextImpl对象由系统在ActivityThread类的performLaunchActivity()方法中创建。
   2. 创建一个window，绑定到成员变量mWindow，给window对象设置callback
   3. 成员变量mApplication绑定Application对象
   4. 给成员变量mWindow保存的window对象设置WindowManager对象
   5. 成员变量mWindowManager绑定WindowManager对象

Activity对象初始化时候不需要上下文，需要上下文的是activity的父类ContextThemeWrapper和ContextWrapper。

1. Activity、ContextThemeWrapper、ContextWrapper、context本身就是上下文对象，为什么还要设置一个成员变量来保存一个上下文对象？什么时候用到这个上下文对象的成员变量，因为在Activity中使用上下文对象的时候，都是使用this。
2. ContextThemeWrapper类用来维护一个应用程序窗口的主题。
3. 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。
4. Activity和ContextImpl虽然都是上下文，但是不是一回事儿。ContextImpl也没有保存到Activity中，而是直接保存到了Activity的父类中。Window中保存的是Activity对象的引用。Window.CallBack是由Activity实现的，而不是ContextImpl，所以Window.setCallBack(this)不能把this换成ContextImpl。
5. Activity的生命周期方法只不过是Activity生命周期的一些片段，实际的声明周期是从ActivityThread创建Activity对象开始，到ActivityThread销毁Activity结束。Activity的每个生命周期方法执行完成之后，都会返回到ActivityThread的方法中，本质上来说Activity运行在ActivityThread中，生命周期方法只是一个回调方法，供用户定义一些自己的操作。比如onCreate运行完成之后，就会回到ActivityThread中继续执行其他一些没有暴露给用户的方法（这些方法远比onCreate方法代码要多），然后当ActivityThread执行到回调onStart方法时，又会回到Activity中执行用户在onStart中定义的代码。Activity就相当于一个ActivityThread暴露给用户的一个接口，里面定义了很多回调方法，用户可以通过实现这些回调方法来执行一些自定义操作。
6. PhoneWindow中成员变量DecorView对象的ActionBar和ContentView的均是在Activity的setContentView()方法中创建的。只不过两个创建过程是执行了两个方法：
   1. ContentView：getWindow. setContentView()
   2. ActionBar：initActionBar()
7. Activity自己的window对象，Activity调用自己的初始化方法activity.attach()时由自己创建的。谁的东西谁负责，DecorView是在Activity.setContentView()方法中创建的，表面上看是在Activity中创建的，实际setContentView方法底层调用了window的setContentView方法，所以DecorView由window自己创建的。只不过因为DecorView这个视图对象肯定是根据用户的要求定制的，所以Activity对window的setContentView方法进行了封装，使用户可以直接在onCreate方法中直接调用。DecorView创建的时候，ContextImpl、Activity、window、windowManger均已创建完成，所以说系统也是按照由大到小、由外到里、由先到后进行对象的创建的。因为这时还用不到ViewRoot，所以在创建DecorView的时候还没有创建ViewRoot。而且由于ViewRoot中的成员变量mView需要绑定DecorView，所以首先创建DecorView，然后在创建ViewRoot，顺序也是比较合理的。用到谁创建谁，被用到的对象先创建。
8. DecorView包含Title和Content两部分，两者的测量、布局均是在ViewRootImpl的performTraversal()方法中进行的。

## Activity视图结构的创建流程

ActivityThread ->Activity -> ContextImpl -> activity.attach -> new Window -> window.setWindowManager –> activity.onCreate -> activity.setContentView -> phoneWindow.setContentView -> decorView -> actionBar -> onCreate执行完成 -> ViewRoot –> onResume

1. onCreate()方法没有执行完成之前，ViewRoot对象还没有创建，所以在onCreate中取view的宽高肯定是拿不到的。

|  |
| --- |
| 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;  }  } |

Activity

---------------------------------------------------------

-mWindowManager：LocalWindowManager

-mWindow：PhoneWindow

Window

---------------------------------------------------------

-mContext: Activity

-mCallBack: Activity.CallBack(Activity中实现)

-mWindowManager: LocalWindowManager(与Activity引用的是一个对象)

-mDefaultDisplay: 针对当前Window所对应的Activity特定的经过调整的Display对象

PhoneWindow

-----------------------------------------------------------------

-mDecor: DecorView描述自己的窗口视图

-mContentParent: ViewGroup描述视图内容的父窗口

Content

Title

ViewRoot

--------------------------------------------------------------------------------

-mView：DecorView (PhoneWindow中的mDecor=DecorView)

-mWindowAttributes：WindowManager.LayoutParams

(描述与该ViewRoot对象对应的一个Activity组件的UI布局信息)

LocalWindowManager

---------------------------------------------------------

-mWindowManager：WindowManagerImpl

WindowManagerImpl

真正管理应用程序窗口的类---------------------------------------------------------

-mViews: View[] DecorView

-mRoots: ViewRoot[] ViewRoot

-mParams: WindowManager.LayoutParams[]

Window

-------------------------

+Interface:CallBack

{

-dispatchTouchEvent()

-dispatchKeyEvent()

……

}

-------------------------

-mCallBack:Activity返回的CallBack实现

Activity

Implements Window.CallBack

-----------------------------------

-dispatchTouchEvent()

-dispatchKeyEvent()

. . . . . . . . . .

-------------------------------------

Activity并不负责执行事件分发，Activity只是实现Window.CallBack接口，定义事件分发的流程。

事件由InputManager获取后，并没有经过Activity，是Window调用的事件分发方法。

Window

接收InputManager发来的事件(是否直接发到window这，待确认)

执行Window.CallBack接口中的事件分发方法（方法由Activity实现）

图 事件分发所涉及对象相互之间的关系及分发执行流程图