ActivityThread类:performLaunchActivity函数：

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

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

r.embeddedID, r.lastNonConfigurationInstance,

r.lastNonConfigurationChildInstances, config);

Activity类中的attach方法：

//创建新的window，与Activity相关联,在android2.2以下以及2.2版本返回的是MidWindow,但是sdk类库中包含PhoneWindow，2.2版本以上删除了MidWindow类，使用的是PhoneWindow。这两个类都是继承自Window，因此可以认为两个类是相同的。

mWindow = PolicyManager.makeNewWindow(this);

\*\*\*省略操作

//创建一个与window相关的WindowManager,由于Activity类中使用的setManager方法传递的WindowManager参数为null，因此在Window类中使用WindowManager变量是同一个对象。

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

if (mParent != null) {

mWindow.setContainer(mParent.getWindow());

}

//从Window类的setWindowManager方法可以知道Activity的WindowManager是一个//LocalWindowManager

mWindowManager = mWindow.getWindowManager();

Window类的setWindowManager方法：

public void setWindowManager(WindowManager wm, IBinder appToken,

String appName) {

mAppToken = appToken;

mAppName = appName;

if (wm == null) {

wm = WindowManagerImpl.getDefault();

}

//使用WindowManager的变量来创建LocalWindowManager

mWindowManager = new LocalWindowManager(wm);

}

private class LocalWindowManager implements WindowManager {

LocalWindowManager(WindowManager wm) {

mWindowManager = wm;

mDefaultDisplay = mContext.getResources().getDefaultDisplay(

mWindowManager.getDefaultDisplay());

}

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

\*\*\*\*

View decor = peekDecorView();

\*\*\*\*

mWindowManager.addView(\*\*\*);

}

\*\*\*\*\*\*\*\*\*

}

Window类的peekDecorView()方法，获取当前的Decor View

public abstract View peekDecorView();

获取顶层窗口decor view（包含了标准窗口）可以作为window添加到window manager。

public abstract View getDecorView();

这两个函数在PhoneWindow中被实现。

在WindowImpl类中：

public static WindowManagerImpl getDefault() {

return mWindowManager;

}

private static WindowManagerImpl mWindowManager = new WindowManagerImpl();

因此通过getDefault()方法返回的WindowManagerImpl是同一个对象。

addView方法,将View保存到数组中，并设置View的root信息：

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

\*\*\*\*

ViewRoot root;

View panelParentView = null;

\*\*\*\*

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

\*\*\*\*

mViews[index] = view;

mRoots[index] = root;

mParams[index] = wparams;

\*\*\*\*

root.setView(view, wparams, panelParentView);

}

PhoneWindow继承了Window，实现了Window中的所有虚函数。

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

// 窗口的顶层View

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.

// 放置窗口内容的View，或者是mDecor或者mDecor的子View用来放置内容，对应于ID为com.android.internal.R.id.content的FramLayout。

private ViewGroup mContentParent;

//设置view

@Override

public void setContentView(int layoutResID) {

//判断contentParent是否为null，如果为null则需要installDecor,否则需要清空原来已经填充的view

if (mContentParent == null) {

installDecor();

} else {

mContentParent.removeAllViews();

}

//inflate xml并设置父view为mContentParent，初次设置view

mLayoutInflater.inflate(layoutResID, mContentParent);

final Callback cb = getCallback();

if (cb != null) {

cb.onContentChanged();

}

}

@Override

public void setContentView(View view) {

setContentView(view, new ViewGroup.LayoutParams(MATCH\_PARENT, MATCH\_PARENT));

}

@Override

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

//判断contentParent是否为null，如果为null则需要installDecor,否则需要清空原来已经填充的view

if (mContentParent == null) {

installDecor();

} else {

mContentParent.removeAllViews();

}

//给mContentParent增加view

mContentParent.addView(view, params);

final Callback cb = getCallback();

if (cb != null) {

cb.onContentChanged();

}

}

private void installDecor() {

//生成mDecor

if (mDecor == null) {

mDecor = generateDecor();

mDecor.setDescendantFocusability(ViewGroup.FOCUS\_AFTER\_DESCENDANTS);

mDecor.setIsRootNamespace(true);

}

if (mContentParent == null) {

//生成mContentParent

mContentParent = generateLayout(mDecor);

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

\*\*\*\*设置标题栏信息

}

}

//使用DecorView作为参数创建mContentParent，并设置mDecor将mContentParent添加为子view

//mContentParent对应于id为content的framlayout可以参考代码中提供的window布局文件，以及使用的findViewById函数的参数。

protected ViewGroup generateLayout(DecorView decor) {

设置flags

WindowManager.LayoutParams params = getAttributes();

设置params

// Inflate the window decor.

int layoutResource;

int features = getLocalFeatures();

获取整个屏幕对应的layout资源id

mDecor.startChanging();

//根据使用的资源不同，显示出来的布局也不同，可以参考sdk下的platforms/android-\*/data/res/layout/screen\*.xml或dialog\*.xml

View in = mLayoutInflater.inflate(layoutResource, null);

//将整个屏幕对应的view作为decor子view，并设置布局参数为铺满整个屏幕(match\_parent)

decor.addView(in, new ViewGroup.LayoutParams(MATCH\_PARENT, MATCH\_PARENT));

//找到id为content的framlayout，由于使用的资源不同，所以content不一定为mDecor的直接子View。使用的findViewById是mDecor的方法，

//由于已经将屏幕View加为mDecor的子View，因此mContentParent也是mDecor的子View

ViewGroup contentParent = (ViewGroup)findViewById(ID\_ANDROID\_CONTENT);

设置mDecor的背景和标题

mDecor.finishChanging();

return contentParent;

}

//从本质上来讲Decor为一个FramLayout

private final class DecorView extends FrameLayout {

\*\*\*省略各种操作

}

Activity在onCreate之前调用attach方法，在attach方法中会创建window对象。window对象创建时并没有创建 Decor对象对象。用户在Activity中调用setContentView,然后调用window的setContentView，这时会检查 DecorView是否存在，如果不存在则创建DecorView对象，然后把用户自己的View 添加到DecorView中。

在ActivityThread函数中:

ActivityThread.java中调用wm.addView(decor, l);把它加入到window manager proxy的mViews中，同时为这个decor view创建一个ViewRoot，ViewRoot负责协调decor view与window manager直接绘图、事件处理。

ViewRoot中有IWindowSession和IWindow用来和window manger打交道和接收window manager传过来的消息，消息传过来后ViewRoot分发给decor view，再由decor view进行分发

windowManager创建和使用流程:

创建：

1、Activity的attach方法中：

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

2、Window的setWindowManager方法，会返回一个LocalWindowManager对象：

if (wm == null) {

//调用WindowManagerImpl的getDefault方法

wm = WindowManagerImpl.getDefault();

}

//使用WindowManager的变量来创建LocalWindowManager

mWindowManager = new LocalWindowManager(wm);

3、WindowManagerImpl的getDefault方法：

public static WindowManagerImpl getDefault() {

return mWindowManager;

}

mWindowManager生成方法，在WIndowManageImpl类中：

private static WindowManagerImpl mWindowManager = new WindowManagerImpl();

使用：

1. 在Activity中生成
2. ActivityThread类的handleResumeActivity方法中：

ActivityRecord r = performResumeActivity(token, clearHide);

if (r != null) {

final Activity a = r.activity;

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

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

View decor = r.window.getDecorView();

decor.setVisibility(View.INVISIBLE);

//只有在此时才创建一个新的WindowManager

ViewManager wm = a.getWindowManager();

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

a.mDecor = decor;

if (a.mVisibleFromClient) {

a.mWindowAdded = true;

//将decor添加到WindowManager中,具体的wm为WindowManagerImpl //在wm的addView方法中将ViewRoot和WindowManager关联起来

wm.addView(decor, l);

}

}

}

1. WindowManagerImpl类的addView方法，windowManager通过数组来建立View,ViewRoot之间的关联：

ViewRoot root;

//使用view来创建ViewRoot

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

root.mAddNesting = 1;

//使用索引将view和ViewRoot关联起来

mViews[index] = view;

mRoots[index] = root;

mParams[index] = wparams;

//设置root的view,并设置view的parent为root

root.setView(view, wparams, panelParentView);

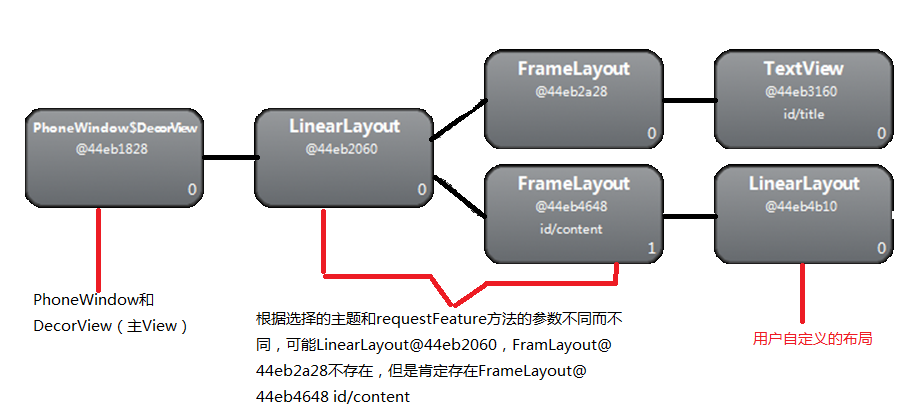
总结Activity，View，window，DecorView的关系如下:

android中真正展示给用户的是window和view，activity在android中所的作用主要是处理一些逻辑问题，比如生命周期的管理、建立窗口等。在android中，窗口的管理还是比较重要的一块，因为他直接负责把内容展示给用户，并和用户进行交互。响应用户的输入等。

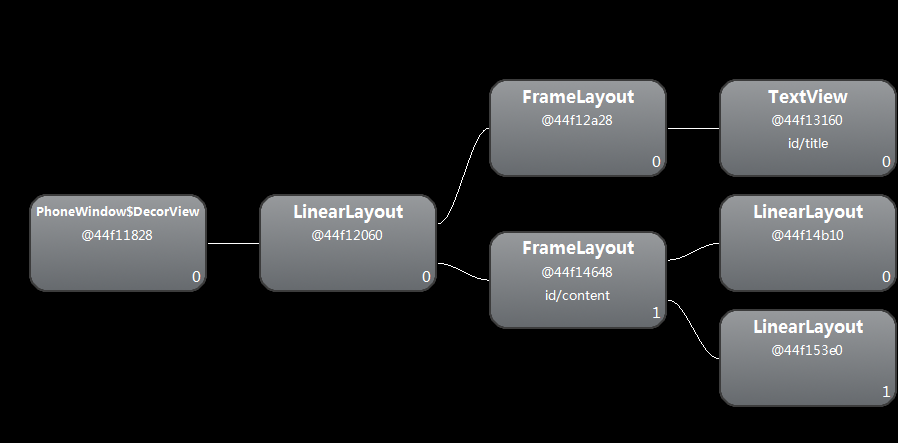
View是真正显示的矩形区域，DecorView是顶层View，也就是主View。

相互之间的关系可以理解为一个Activity包含了一个Window，这个Window其实是一个PhoneWindow，在PhoneWindow中包含了DecorView，变量名称为mDecor，mDecor有一个子View，这个子View的布局方式根据设定的主题来确定，在这个子View的xml布局中包含了一个FrameLayout元素，这个FrameLayout元素的id为content，这个content对应于PhoneWindow中的mContentParent变量，用户自定义的布局作为mContentParent的子View存在,一般情况下mContentParnet只有一个子View，如果在Activity调用addView方式实际上是给PhoneWindow中的mContentParent添加子View，由于mContentParent是一个FrameLayout,因此新的子view会覆盖通过setContentView添加的子view。

仅通过setContentView添加子View类View层次与下图类似：



通过addView方式给Activity添加子view层次图如下所示，右下角两个LinearLayout为用户自定义的布局。



ViewRoot作用：是View和WindowManager之间的桥梁，用于两者之间的协议通讯。

Android 的窗口管理是基于 C/S 模式的，客户端就是应用程序，服务端 就是 Window Manager服务。如下图：

 public static IWindowSession getWindowSession(Looper mainLooper) {

InputMethodManager imm = InputMethodManager.getInstance(mainLooper);

//获取回话Session，ViewRoot和Window manager通信，

sWindowSession = IWindowManager.Stub.asInterface(

ServiceManager.getService("window"))

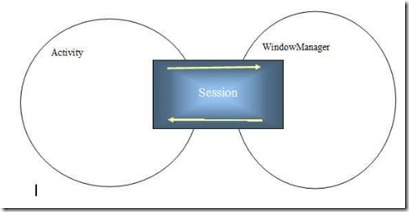
.openSession(imm.getClient(), imm.getInputContext());

return sWindowSession;

}

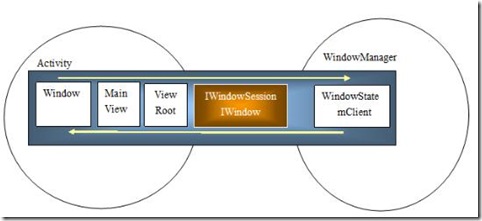
}

Activity和Window manager之间的通讯：

[](http://photo.blog.sina.com.cn/list/blogpic.php?pid=4b3c1f9549fd7abd57a5e&bid=4b3c1f950100qe9m&uid=1262231445)

    Activity 建立一个主窗口之后，在将主窗口添加到 Window Manager 时，首先要建立 Window Manager的代理对象，并打开一个Session（session：会话；该session由上面函数中的第二行代码实现，实现 IWindowSession AIDL 接口），并维持该会话（Activity 将通过该会话与 Window Manager 建立联系，这个Session 是C/S 体系的基础）。Client 通过这个Session 将 window 加入到 Window Manager 中。

    一个完整的窗口概念包含了 View，ViewRoot，Window Manager Service，Window，Decor View，IWindow，ISession，WindowState。他们之间的关系如下：

[](http://photo.blog.sina.com.cn/list/blogpic.php?pid=4b3c1f9549fd7abdeb119&bid=4b3c1f950100qe9m&uid=1262231445)

    Client 端的 Activity 通过 IWindowSession 会话与Window Manager Service 建立对话，而 Window Manager Service 通过 IWindow 接口访问 Client，将消息传递到Client 端，在通过消息分发渠道，将消息传递到具体的消息处理函数。（用户输入等操作最先是到 窗口管理服务，由窗口管理服务发给活动窗口，再一步步传递到焦点）。

public ViewRoot(Context context) {

super();

//获取IWindowSession,

getWindowSession(context.getMainLooper());

mWindow = new W(this, context);

}

W类，用于window manager向Activity传递各种消息。

static class W extends IWindow.Stub {

private final WeakReference<ViewRoot> mViewRoot;

private final Looper mMainLooper;

public W(ViewRoot viewRoot, Context context) {

mViewRoot = new WeakReference<ViewRoot>(viewRoot);

mMainLooper = context.getMainLooper();

}

\*\*\*\*\*\*从Window mananger来的消息通过W出递给ViewRoot，由ViewRoot分发

public void dispatchKey(KeyEvent event) {

final ViewRoot viewRoot = mViewRoot.get();

if (viewRoot != null) {

viewRoot.dispatchKey(event);

} else {

new EventCompletion(mMainLooper, this, event, false, null);

}

}

}

通过IWindowSession机制就可以保证Activity和Window manager保持联系，通过IWindow可以将window manager收集到的消息分发出去。