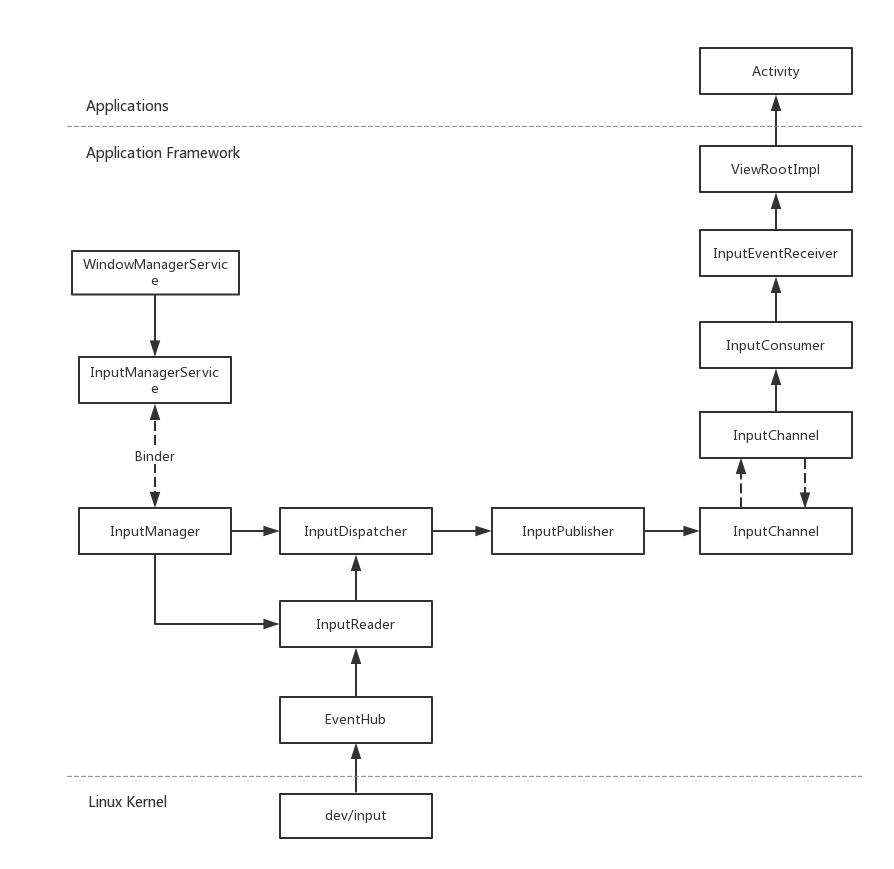
运行于systemserver线程中,在Event事件分发之前处理,比如电源键。Event事件分发后,仅有包含Activity的apk线程才可以处理,如果apk中没有activity但是想处理Event事件怎么办呢?可以在PhoneWindowManager做做文章了。\

# 概述

# 系统层的事件传递



当用户触摸屏幕或者按键操作，首次触发的是硬件驱动，驱动收到事件后，将该相应事件写入到输入设备节点， 这便产生了最原生态的内核事件。接着，输入系统取出原生态的事件，经过层层封装后成为KeyEvent或者MotionEvent ；最后，交付给相应的目标窗口(Window)来消费该输入事件。可见，输入系统在整个过程起到承上启下的衔接作用。

## 事件模型原理

事件模型原理

* WindowManagerService主要为窗口系统提供服务，把事件消息分发给最上层的窗口，WindowManagerService通过InputManager提供的native接口开启了两个线程驱动做Event读取和分发。
* EventHub是系统所有事件的中央处理站，从驱动文件读取RawEvents。
* InputReader负责从EventHub取出事件并转化处理，再交给InputDispatcher；
* InputDispatcher接收来自InputReader的输入事件，并记录WMS的窗口信息，用于派发事件到合适的窗口；
* InputManagerService跟WMS交互，WMS记录所有窗口信息，并同步更新到IMS，为InputDispatcher正确派发事件到ViewRootImpl提供保障；

Ghjk

# 启动过程

在SystemServer中,

**[html]** [view plain](https://blog.csdn.net/u012439416/article/details/54605998) [copy](https://blog.csdn.net/u012439416/article/details/54605998)

1. inputManager = new InputManagerService(context);
2. wm = WindowManagerService.main(context, inputManager,
3. mFactoryTestMode != FactoryTest.FACTORY\_TEST\_LOW\_LEVEL,
4. !mFirstBoot, mOnlyCore);
5. ServiceManager.addService(Context.WINDOW\_SERVICE, wm);
6. ServiceManager.addService(Context.INPUT\_SERVICE, inputManager);
8. mActivityManagerService.setWindowManager(wm);
9. inputManager.setWindowManagerCallbacks(wm.getInputMonitor()); // 消息分发之前的回调
10. inputManager.start();



InputManagerService的创建以及启动已经论述了,在此就不多说了。在WindowManagerService中新建了PhoneWindowManager和InputMonitor,并且将设置为InputManagerService对象的回调对象。其实这2个类都和Event的处理有关,在Event分发之前进行处理,相当于截取,具体的细节就一层一层抽丝剥茧了。

PhoneWindowManager是没有对应的C/C++层代码了。

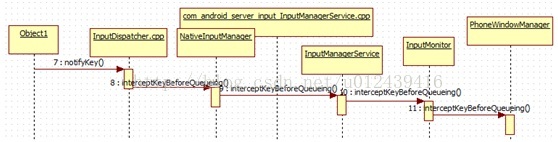
看下InputManagerService的setWindowManagerCallbacks方法,

**[html]** [view plain](https://blog.csdn.net/u012439416/article/details/54605998) [copy](https://blog.csdn.net/u012439416/article/details/54605998)

1. public void setWindowManagerCallbacks(WindowManagerCallbacks callbacks) {
2. mWindowManagerCallbacks = callbacks;
3. }

因此, InputManagerService的变量mWindowManagerCallbacks指向InputMonitor对象。

## 处理流程



InputDispatcher的notifyKey方法中,Event进队列之前,会调用interceptKeyBeforeQueueing方法,

**[html]** [view plain](https://blog.csdn.net/u012439416/article/details/54605998) [copy](https://blog.csdn.net/u012439416/article/details/54605998)

1. mPolicy-**>**interceptKeyBeforeQueueing(&event, /\*byref\*/ policyFlags);

mPolicy变量是什么呢?通过查找,是NativeInputManager对象,

**[html]** [view plain](https://blog.csdn.net/u012439416/article/details/54605998) [copy](https://blog.csdn.net/u012439416/article/details/54605998)

1. void NativeInputManager::interceptKeyBeforeQueueing(const KeyEvent\* keyEvent,
2. uint32\_t& policyFlags) {
3. bool interactive = mInteractive.load();
4. if (interactive) {
5. policyFlags |= POLICY\_FLAG\_INTERACTIVE;
6. }
7. if ((policyFlags & POLICY\_FLAG\_TRUSTED)) {
8. nsecs\_t when = keyEvent-**>**getEventTime();
9. JNIEnv\* env = jniEnv();
10. jobject keyEventObj = android\_view\_KeyEvent\_fromNative(env, keyEvent);
11. jint wmActions;
12. if (keyEventObj) {
13. wmActions = env-**>**CallIntMethod(mServiceObj,
14. gServiceClassInfo.interceptKeyBeforeQueueing,
15. keyEventObj, policyFlags);
16. if (checkAndClearExceptionFromCallback(env, "interceptKeyBeforeQueueing")) {
17. wmActions = 0;
18. }
19. android\_view\_KeyEvent\_recycle(env, keyEventObj);
20. env-**>**DeleteLocalRef(keyEventObj);
21. }
22. •••
23. }

在register\_android\_server\_InputManager方法中,

**[html]** [view plain](https://blog.csdn.net/u012439416/article/details/54605998) [copy](https://blog.csdn.net/u012439416/article/details/54605998)

1. int res = jniRegisterNativeMethods(env, "com/android/server/input/InputManagerService",
2. gInputManagerMethods, NELEM(gInputManagerMethods));
3. •••
4. GET\_METHOD\_ID(gServiceClassInfo.interceptKeyBeforeQueueing, clazz,
5. "interceptKeyBeforeQueueing", "(Landroid/view/KeyEvent;I)I");

很明显了,调用Java层InputManagerService的interceptKeyBeforeQueueing方法,后面就没有什么困难了,最后调用PhoneWindowManager的interceptKeyBeforeQueueing方法。

看下InputMonitor的方法,有一些也是同样的方法调用,关键是C/C++层的代码什么时候调用,觉得InputMonitor的主要目的还是解耦,防止PhoneWindowManager和InputManagerService太紧,

**[html]** [view plain](https://blog.csdn.net/u012439416/article/details/54605998) [copy](https://blog.csdn.net/u012439416/article/details/54605998)

1. public void notifyLidSwitchChanged(long whenNanos, boolean lidOpen) {
2. mService.mPolicy.notifyLidSwitchChanged(whenNanos, lidOpen);
3. }
4. public void notifyCameraLensCoverSwitchChanged(long whenNanos, boolean lensCovered) {
5. mService.mPolicy.notifyCameraLensCoverSwitchChanged(whenNanos, lensCovered);
6. }
7. public int interceptKeyBeforeQueueing(KeyEvent event, int policyFlags) {
8. return mService.mPolicy.interceptKeyBeforeQueueing(event, policyFlags);
9. }
10. public int interceptMotionBeforeQueueingNonInteractive(long whenNanos, int policyFlags) {
11. return mService.mPolicy.interceptMotionBeforeQueueingNonInteractive(
12. whenNanos, policyFlags);
13. }
14. public long interceptKeyBeforeDispatching(
15. InputWindowHandle focus, KeyEvent event, int policyFlags) {
16. WindowState windowState = focus != null ? (WindowState) focus.windowState : null;
17. return mService.mPolicy.interceptKeyBeforeDispatching(windowState, event, policyFlags);
18. }
19. public KeyEvent dispatchUnhandledKey(
20. InputWindowHandle focus, KeyEvent event, int policyFlags) {
21. WindowState windowState = focus != null ? (WindowState) focus.windowState : null;
22. return mService.mPolicy.dispatchUnhandledKey(windowState, event, policyFlags);
23. }
24. public int getPointerLayer() {
25. return mService.mPolicy.windowTypeToLayerLw(WindowManager.LayoutParams.TYPE\_POINTER)
26. \* WindowManagerService.TYPE\_LAYER\_MULTIPLIER
27. + WindowManagerService.TYPE\_LAYER\_OFFSET;
28. }

在InputDispatcher.cpp的中的Event入队列之后,dispatchKeyLocked方法中会调用doInterceptKeyBeforeDispatchingLockedInterruptible方法,然后调用NativeInputManager的interceptKeyBeforeDispatching方法,

**[html]** [view plain](https://blog.csdn.net/u012439416/article/details/54605998) [copy](https://blog.csdn.net/u012439416/article/details/54605998)

1. nsecs\_t delay = mPolicy-**>**interceptKeyBeforeDispatching(commandEntry-**>**inputWindowHandle,
2. &event, entry-**>**policyFlags);

最后会调用PhoneWindowManager的interceptKeyBeforeDispatching方法。

所以,如果apk中没有acitivity但是想监听Event事件怎么办呢?

在PhoneWindowManager的interceptKeyBeforeQueueing或者interceptKeyBeforeDispatching方法中稍加处理就可以了。

# 理解Window和WindowManager

对于重要的系统服务在手机开机的时候进行初始化, 然后保存在了系统进程中. 当我们应用启动的时候会触发ContextImpl类的加载, 在类加载的时候通过静态代码块对各个系统服务进行进行了注册,并保存到一个静态map容器中, 之后就可以通过getSystemService(serviceName)的形式获取不同的系统服务.

Window表示的是一个窗口的概念，在日常生活中使用的并不是很多，但是某些特殊的需求还是需要的，比如悬浮窗之类的，他的具体实现是PhoneWindow,创建一个Window很简单，只需要WindowManager去实现，WindowManager是外界访问Window的入口，Window的具体实现是在WindowManagerService中，他们两个的交互是一个IPC的过程，Android中的所有视图都是通过Windowl来实现的，无论是Activity,Dialog还是Toast,他们的视图都是直接附加在Window上的，因此Window是View的直接管理者，在之前的事件分发中我们说到，View的事件是通过WIndow传递给DecorView，然后DecorView传递给我们的View，就连Activity的setContentView,都是由Window传递的。

## Window和WindowManager



**Window与WindowManager基础关系**

ViewManager接口定义了一组规则，也就是add、update、remove的操作View接口。也就是说ViewManager是用来添加和移除activity中View的接口

WindowManager继承自ViewManager，然后自己还是一个接口，同时又定义了一个静态内部类LayoutParams（这个类比较重要，后面会分析。提前透漏下，如果你在APP做过类似360助手屏幕的那个悬浮窗或者做过那种类似IOS的小白圆点，点击展开菜单功能，你或多或少就能猜到这个类的重要性。）。WindowManager用来在应用与Window之间的接口、窗口顺序、消息等的管理。

ViewManager的另一个实现子类ViewGroup

|  |
| --- |
| public abstract class ViewGroup extends View implements ViewParent, ViewManager {  //protected ViewParent mParent;  //这个成员是View定义的，ViewGroup继承自View，所以也可以拥有。  //这个变量就是前面我们一系列文章分析View向上传递的父节点，类似于一个链表Node的next一样  //最终指向了ViewRoot  ......  public void addView(View child, LayoutParams params) {  addView(child, -1, params);  }  ......  public void addView(View child, int index, LayoutParams params) {  ......  // addViewInner() will call child.requestLayout() when setting the new LayoutParams  // therefore, we call requestLayout() on ourselves before, so that the child's request  // will be blocked at our level  requestLayout();  invalidate(true);  addViewInner(child, index, params, false);  }  ......  } |

所以说View通过ViewGroup的addView方法添加到ViewGroup中，而ViewGroup层层嵌套到最顶级都会显示在在一个窗口Window中, 对于一个Activity只有一个DecorView（ViewRoot），也只有一个Window。

为了了解Window的工作机制，我们首先来看下如何通过WindowManager来创建一个Window.

|  |
| --- |
| Button btn = new Button(this);  btn.setText("我是窗口");  WindowManager wm = (WindowManager) getSystemService(WINDOW\_SERVICE);  WindowManager.LayoutParams layout = new WindowManager.LayoutParams(WindowManager.LayoutParams.WRAP\_CONTENT  , WindowManager.LayoutParams.WRAP\_CONTENT, 0, 0, PixelFormat.TRANSLUCENT);  layout.flags = WindowManager.LayoutParams.FLAG\_NOT\_TOUCH\_MODAL  | WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE  | WindowManager.LayoutParams.FLAG\_SHOW\_WHEN\_LOCKED;  layout.gravity = Gravity.CENTER;  layout.type = WindowManager.LayoutParams.TYPE\_PHONE;  layout.x = 300;  layout.y = 100;  wm.addView(btn, layout); |

上述的代码，其中type和flag是比较重要的，我们来看下

**Flag参数表示window的属性**，他有很多选项，我们挑几个重点的

* FLAG\_NOT\_FOCUSABLE: 表示窗口不需要获取焦点，也不需要接收各种事件，这属性会同时启动FLAG\_NOT\_TOUCH\_MODAL，最终的事件会传递给下层的具体焦点的window.
* FLAG\_NOT\_TOUCH\_MODAL:在此模式下，系统会将当前window区域以外的单击事件传递给底层的Window，此前的Window区域以内的单机事件自己处理，这个标记很重要，一般来说都需要开启，否则其他window将无法获取单击事件
* FLAG\_SHOW\_WHEN\_LOCKED: 开启这个属性可以让window显示在锁屏上.

**Type参数表示window的类型**，window有三种类型，分别是应用，子，系统，应用window对应一个Activity,子Window不能单独存在，需要依赖一个父Window，比如常见的Dialog都是子Window,系统window需要声明权限，比如系统的状态栏.Window是分层的，每个Window对应着z-ordered,层级大的会覆盖在层级小的Window上面，这和HTML中的z-index的概念是一致的，在这三类中.

* 应用是层级范围是1-99.
* 子window的层级是1000-1999
* 系统的层级是2000-2999。

这些范围对应着type参数，如果想要window在最顶层，那么层级范围设置大一点就好了，很显然系统的值要大一些，系统的值很多，我们一般会选择TYPE\_SYSTEM\_OVERLAY和TYPE\_SYSTEM\_ERROR，记得要设置权限.

**<uses-permission android:name="android.permission.SYSTEM\_ALERT\_WINDOW"/>**

WindowManager所提供的功能很简单，常用的有三个方法，添加View,更新View,删除View,这三个方法定义在ViewManager中，而WindowManager继承自ViewManager:

|  |
| --- |
| public interface ViewManager {  public void addView(View view, ViewGroup.LayoutParams params);  public void updateViewLayout(View view, ViewGroup.LayoutParams params);  public void removeView(View view);  } |

虽然只有三个功能，但是这三个功能足够我们使用了，我们常见的可以推动的View，其实也很好实现，就是不断的更改他xy的位置.

|  |
| --- |
| button.setOnTouchListener(new View.OnTouchListener() {  @Override  public boolean onTouch(View v, MotionEvent event) {  int rawX = (int) event.getRawX();  int rawY = (int) event.getRawY();  switch (event.getAction()) {  case MotionEvent.ACTION\_MOVE:  layout.x = rawX;  layout.y = rawY;  wm.updateViewLayout(button, layout);  break;  }  return false;  }  }); |

## Window的内部机制

Window是一个抽象的概念，每一个Window对应着一个View和一个ViewRootImpl,Window和View通过ViewRootImpl建立关系，因此Window并不是实际存在的，这点从WindowManager定义的接口都是针对View，这说明View才是window的实体，在实际使用当中我们并不能直接访问Window的添加过程.

### Window的添加过程

Window的添加过程是通过WindowManager的addView去实现的，而真正实现的是一个接口，也就是WindowManagerImpl.

|  |
| --- |
| Activity.attach() {  mWindow = new PhoneWindow(this, window);  mWindow.setWindowManager(  (WindowManager)context.getSystemService(Context.WINDOW\_SERVICE));  **mWindowManager = mWindow.getWindowManager();**  }  Window.setWindowManager() {  mWindowManager = ((**WindowManagerImpl**)wm).createLocalWindowManager(this);  } |

|  |
| --- |
| @Override  public void addView(View view, ViewGroup.LayoutParams params) {  **mGlobal**.addView(view, params, mDisplay, mParentWindow);  }  @Override  public void updateViewLayout(View view, ViewGroup.LayoutParams params) {  mGlobal.updateViewLayout(view, params);  }  @Override  public void removeView(View view) {  mGlobal.removeView(view, false);  } |

可以发现，WindowManagerImpl并没有直接去实现一个Window的三大操作，而是全部交给了WindowManagerGlobal来处理，WindowManagerGlobal是一个工厂的性质提供自己的实现，在WindowManagerGlobal中有一段如下的代码：

private final WindowManagerGlobal mGlobal = WindowManagerGlobal.getInstance();

WindowManagerImpl这种工作模式就是典型的桥接模式，将所有的操作全部委托给WindowManagerGlobal（**@hide**这是系统隐藏类，普通app不可见！）去实现，WindowManagerGlobal的addView方法主要分如下几步：

|  |
| --- |
| public final class WindowManagerGlobal {  private final ArrayList<View> mViews = new ArrayList<View>();  private final ArrayList<ViewRootImpl> mRoots = new ArrayList<ViewRootImpl>();  private final ArrayList<WindowManager.LayoutParams> mParams =  new ArrayList<WindowManager.LayoutParams>();  public void addView(View view, ViewGroup.LayoutParams params,  Display display, Window parentWindow) {  ......  ViewRootImpl root;  View panelParentView = null;  synchronized (mLock) {  ......  **//1.创建ViewRootImpl**  root = new ViewRootImpl(view.getContext(), display);  view.setLayoutParams(wparams);  **//2.添加到列表中**  mViews.add(view);  mRoots.add(root);  mParams.add(wparams);  }  **//3. 最后执行这个，将View显示在手机屏幕上**  try {  root.setView(view, wparams, panelParentView);  } catch (RuntimeException e) {  ......  }  }  } |

#### 创建ViewRootImpl

**ViewRootImpl的作用**可以总结为: Android Framework与WMS之间的通信也是通过Binder机制进行的. 这里就建立了与WMS的通信. 最后通过openSession()函数来与WMS建立一个通信会话, 相当于建立了一个长期的处理中心, 双方有什么需要够可以通过这个Session来交换信息.

WMS是运行在Native层的, 而现在只是在Framewrok层. ViewRootImpl是通信的桥梁

上面的方法中先是new了一个ViewRootImpl，然后调用他的setView来显示布局。

Android中的View都是通过ViewRootImpl来完成绘制的。

|  |
| --- |
| ublic final class ViewRootImpl implements ViewParent,  View.AttachInfo.Callbacks, ThreadedRenderer.HardwareDrawCallbacks {  public ViewRootImpl(Context context, Display display) {  mContext = context;  //获取IWindowSession，与WindowManagerService建立连接  mWindowSession = **WindowManagerGlobal.getWindowSession();**  //这里保存当前线程  mThread = Thread.currentThread();  }  } |

继续追踪WindowManagerGlobal.getWindowSession()：

|  |
| --- |
| public static IWindowSession **getWindowSession**() {  synchronized (WindowManagerGlobal.class) {  if (sWindowSession == null) {  try {  InputMethodManager imm = InputMethodManager.getInstance();  //获取WindowManagerService  IWindowManager windowManager = **getWindowManagerService**();  //与系统的WindowManagerService建立一个IWindowSession  sWindowSession = windowManager.openSession(  new IWindowSessionCallback.Stub() {  @Override  public void onAnimatorScaleChanged(float scale) {  ValueAnimator.setDurationScale(scale);  }  },  imm.getClient(), imm.getInputContext());  } catch (RemoteException e) {  throw e.rethrowFromSystemServer();  }  }  return sWindowSession;  }  }  public static IWindowManager getWindowManagerService() {  synchronized (WindowManagerGlobal.class) {  if (sWindowManagerService == null) {  //这里返回的是IBinder对象，进行IPC通信  **sWindowManagerService = IWindowManager.Stub.asInterface(**  **ServiceManager.getService("window"));**  ......  }  return sWindowManagerService;  }  } |

继续看 sWindowManagerService = IWindowManager.Stub.asInterface(

ServiceManager.getService("window"));这行：

|  |
| --- |
| public final class ServiceManager {  public static IBinder getService(String name) {  try {  IBinder service = sCache.get(name);  if (service != null) {  return service;  } else {  return getIServiceManager().getService(name);  }  } catch (RemoteException e) {  Log.e(TAG, "error in getService", e);  }  return null;  }  } |

所以IWindowManager.Stub.asInterface(ServiceManager.getService("window"));得到的是一个IBinder对象。到这里，在ServiceManager的getService方法中通过getIServiceManager().getService("window")获取到一个IBinder，与WMS建立初步连接。然后通过IWindowManager.Stub.asInterface方法将IBinder转换成IWindowManager对象。通过这个对象调用openSession打开一个Session，实现通话。但是，WMS只负责管理View的z-order，也就是管理当前那个View在最上层显示，并不管理绘制。

#### ViewRootImpl.setView

绘制View的任务就在ViewRootImpl的setView方法中。

|  |
| --- |
| public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {  synchronized (this) {  ......  //请求绘制View  requestLayout();  ......  int res; /\* = WindowManagerImpl.ADD\_OKAY; \*/  try {  ......  //请求WindowManagerService，让WMS实现Window的添加。  res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,  getHostVisibility(), mDisplay.getDisplayId(),  mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,  mAttachInfo.mOutsets, mInputChannel);  }  ......  }  } |

setView主要做了两件事：

（1）requestLayout();

（2）向WMS请求添加Window；

#### requestLayout();

@Override

public void requestLayout() {

if (!mHandlingLayoutInLayoutRequest) {

checkThread();

mLayoutRequested = true;

scheduleTraversals();

}

}

**这里我另外注意一下checkThread()这个方法。**

void checkThread() {

if (mThread != Thread.currentThread()) {

throw new CalledFromWrongThreadException(

"Only the original thread that created a view hierarchy can touch its views.");

}

}

是不是看到了熟悉的异常提示？这个方法要判断当前更新UI 的线程是不是创建ViewRootImpl时的线程，只有在创建ViewRootImpl的线程中更新对应的UI才不会报错。所以不能在子线程中更新UI也是这个原因。

setView->requestLayout->scheduleTraversals—>mTraversalRunnable->doTraversal->performTraversals来通过一个handler把这个任务发送出去：完成异步刷新请求

看performTraversals();这个方法里将近一千行代码,大致是四个过程：

private void performTraversals(){

// 1 获取Surface对象，用于图形绘制

//2 测量整个视图树中各个View的大小，用performMeasure方法

//3 布局整个视图树，用performLayout方法

//4 绘制整个视图树，用performDraw方法

}

performDraw->ViewRootImpl.draw()方法->.ViewRootImpl.drawSoftware方法，调用GPU绘图。

draw()中获取到绘制表面Surface，里面最后调用ViewRootImpl的drawSoftware方法，调用GPU绘图。

private boolean drawSoftware(Surface surface, AttachInfo attachInfo, int xoff, int yoff,

boolean scalingRequired, Rect dirty) {

final Canvas canvas;

try {

//获取canvas

canvas = mSurface.lockCanvas(dirty);

// TODO: Do this in native

canvas.setDensity(mDensity);

} catch (Surface.OutOfResourcesException e) {

}

try {

if (!canvas.isOpaque() || yoff != 0 || xoff != 0) {

canvas.drawColor(0, PorterDuff.Mode.CLEAR);

}

try {

......

//从这里开始绘制整个视图树，从DecorView开始

mView.draw(canvas);

} finally {

......

}

} finally {

try {

//解锁canvas，并通知SurfaceFlinger更新这块区域。

surface.unlockCanvasAndPost(canvas);

} catch (IllegalArgumentException e) {

......

}

}

return true;

}

综上所述，视图树的绘制主要有以下步骤：

（1）判断使用CPU还是GPU绘制

（2）获取绘制表面Surface对象

（3）通过Surface对象获取并锁住绘图对象

（4）从DecorView开始绘制整个视图树

（5）解锁Canvas，并通知SurfaceFlinger更新这块区域。

**2.创建ViewRootImpl并将View添加到列表中**

在WindowManagerGlobal有如下几个列表是比较重要的.

|  |
| --- |
| private final ArrayList<View> mViews = new ArrayList<View>();  private final ArrayList<ViewRootImpl> mRoots = new ArrayList<ViewRootImpl>();  private final ArrayList<WindowManager.LayoutParams> mParams =  new ArrayList<WindowManager.LayoutParams>();  private final ArraySet<View> mDyingViews = new ArraySet<View>(); |

在上面的声明中，mViews存储所有window所对应的View，mRoots存储是所有window所对应的ViewRootImpl，mParams存储是所对应的布局参数，而mDyingViews则存储那些正在被删除的对象，在addView中通过如下方式将Window的一系列对象添加到列表中.

|  |
| --- |
| root = new ViewRootImpl(view.getContext(), display);  view.setLayoutParams(wparams);  mViews.add(view);  mRoots.add(root);  mParams.add(wparams); |

如此一来,Window的添加请求就交给WindowManagerService去处理了，在WindowManagerService内部为每一个应用添加了一个单独的Session,具体WIndow在WindowManagerService中怎么添加的，我们就不去分析了，读者可以自己去熟悉下.

### Window的删除过程

Window的删除过程和添加过程一样，都是通过Impl再通过Global来实现了，下面是Global的removeView.

|  |
| --- |
| public void removeView(View view) {  synchronized (this) {  mViews.remove(view);  for (int i = mViewRects.size(); i-- > 0; ) {  AttachInfo.InvalidateInfo info = mViewRects.get(i);  if (info.target == view) {  mViewRects.remove(i);  info.recycle();  }  }  if (mPosted && mViews.isEmpty() && mViewRects.isEmpty()) {  mChoreographer.removeCallbacks(Choreographer.CALLBACK\_ANIMATION, this, null);  mPosted = false;  }  } |

removeView的代码很清晰，首先通过findViewLocked来查找待删除的索引，这个超找过程就是建立的数组遍历，然后进一步的删除。

removeViewLocked是通过ViewRootImpl来完成删除操作的，在windowmanager中提供了两种接口removeView和removeViewImmediate，他们分别表示异步删除和同步删除，其中removeViewImmediate，使用起来要格外注意，一般来说不需要使用此方法来删除window以免发生意外的错误，这里主要是异步删除的问题，具体的删除操作是ViewImple的die方法来完成的，在异步删除的情况下，die只是发生一个删除的请求后就返回了，这个时候View并没有完成删除的操作，所有最后会将其添加到mDyingViews中，mDyingViews表示待删除的View列表，ViewRootImpe的die方法如下：

在die方法内部只是做了简单的判断，那么就发送了一个MSG\_DIE的消息，ViewRootImpl中的mHandler会处理此消息并且并调用doDie方法，如果是同步删除就会直接调用doDie方法，在doDie方法内部会操作dispatchDetachedFromWindow，真正删除window就是在这里面实现的，他主要做了四件事。

1.垃圾回收相关的工作，比如清除数据和消息，移除回调

2.通过Session的remove方法来删除window,这同样是一个IPC的过程，最终会调用wms的removeWindow方法

3.调用view的dispatchDetachedFromWindow方法，在内不会调用onDetachedFromWindow,他做了一些回收资源或者停止动画的一些操作

4.调用WindowManagerGlobal的doRemoveView方法刷新数据

### Window的更新过程

到这里，window的删除就接收完了，在说下更新，需要看WindowManagerGlobal的updateViewLayout方法。

|  |
| --- |
| public void updateViewLayout(View view, ViewGroup.LayoutParams params) {  final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams)params;  view.setLayoutParams(wparams);  synchronized (mLock) {  int index = findViewLocked(view, true);  ViewRootImpl root = mRoots.get(index);  **mParams.remove(index);**  **mParams.add(index, wparams);**  root.setLayoutParams(wparams, false);  }  } |

updateViewLayout做的方法比较简单，首先他更新View的LayoutParams替换老的，接着再更新下ViewRootimpl中的LayoutParams，这一步是通过viewrootimpl的setLayoutParams来做的，在ViewRootImpl中会通过scheduleTraversals方法来对View，测量，布局，重绘等等，除了view本身的重绘之外，ViewRootImpl还会通过WindowSession来更新Window的视图，这个过程最终是WindowManagerService的relayoutWindow来实现的，具体也是一个IPC的过程

## Window的创建过程

通过上面的分析，我们知道，view是android中视图的呈现方式，但是view不能单独存在，他必须依附在window这个抽象类中，因此有视图的地方就有window，activity，toast都是，我们继续来分析window的创建过程。

### Activity的Window创建过程

要分析Activity的Window创建过程就需要去了解activity的启动过程，详细的会在后面说，这里简单概括，activity的启动过程很复杂，最终会由ActivityThread中的perfromLaunchActivity()来完成整个启动过程。

在Activity的attach方法中，系统会创建activity所属的window对象PhoneWindow并为其设置回调接口，window对象的创建过程是由PolicyManager的akeNewWindow方法实现的，由于activity实现了window的callback方法接口，因此当window接受到外界的状态改变的时候就会去调用activity的方法，callback接口中的方法很多，但是有几个确实我们非常熟悉的，你如onAttachedToWindow等。

下面分析activity的视图是怎么依附在window上的由于activity的视图是由setContentView开始的，所有我们先看下这个方法：

|  |
| --- |
| public void setContentView(@LayoutRes int layoutResID) {  getWindow().setContentView(layoutResID);  initWindowDecorActionBar();  } |

1.如果没有DecorView就去创建他

DecorView是一个FrameLayout,在之前就已经说过了，这里说一下。DecorView是activity中的顶级View,一般来说他的内部包含标题栏和内部栏，但是这个会随着主题的变化而发生改变的，不管怎么样，内容是一定要存在的，并且内容有固定的id，那就是content,完整的就是android.R.id.content，DecorView的创建是由installDecor方法来完成的，在方法内部会通过generateDecor方法来完成创建DecorView,这个时候他就是一个空白的FrameLayout.

|  |
| --- |
| protected DecorView generateDecor(){  return new DecorView(getContext(),-1);  } |

为了初始化DecorView的结构，PhoneWindow还需要通过generateLayout方法来加载具体的布局文件到DecorView中，这个跟主题有关.

|  |
| --- |
| View in = mLayoutInflater.inflate(layoutResource, null);  decor.addView(in, new ViewGroup.LayoutParams(MATCH\_PARENT, MATCH\_PARENT));  mContentRoot = (ViewGroup) in;  ViewGroup contentParent = (ViewGroup)findViewById(ID\_ANDROID\_CONTENT); |

其中ID\_ANDROID\_CONTENT的定义如下，这个id对应的就是ViewGroup的mContentParent

public static final int ID\_ANDROID\_CONTENT = com.android.internal.R.id.content

2.将View添加到DecorView的mContentParent中

这个过程比较简单，由于在第一步的时候已经初始化了DecorView，因此这一部就直接将activity的视图添加到DecorView的mContentParent中既可，mLayoutInflater.inflate(layoutResID, mContentParent);到此为止，由此可以理解activity的setcontentview的来历了，也许有读者会怀疑，为什么不叫setview来，他明明是给activity设置视图啊，从这里来看，他的确不适合叫做setview,因为activity的布局文件只是添加到了DecorView的mContentParent中，因此交setContentView更加准确。

3.回调Activity的onCreateChanged方法来通知Activity视图已经发生改变

这个过程很简单，由于window实现了Callback接口，这里表示布局已经被添加到DecorView的mContentParent中了，于是通知activity。使其可以做相应的处理Activity的onCreateChanged是一个空实现，我们可以在子activity处理这个回调

|  |
| --- |
| final callback cb = getCallback();  if(cb != null && !isDestroyed()){  cb.onContentChanged();  } |

经过了上面的三个步骤，到这里为止DecorView已经被创建并且初始化完毕了activity的布局文件也已经添加到了DecorView的内容中，但是这个时候DecorView还没有被windowmanager添加到window中，这里需要正确的理解window的概念，window更多的是表示一种抽象的功能集合，虽然说早在activity的attch中window就已经被创建了，但是这个时候由于DecorView还没有被windowmanager识别，所有还不能提供具体的功能，因为他还无法接收外界的输入，在activityThread的makeVisible中，才能被视图看到：

|  |
| --- |
| void makeVisible() {  if (!mWindowAdded) {  ViewManager wm = getWindowManager();  wm.addView(mDecor, getWindow().getAttributes());  mWindowAdded = true;  }  mDecor.setVisibility(View.VISIBLE);  } |

到这里，window的创建过程就已经分析完了

### Dialog的Window创建过程



#### 创建Window

dialog的window创建过程和activity的类似，我们来看下

|  |
| --- |
| Dialog(Context context, int theme, boolean createContextThemeWrapper) {  if (createContextThemeWrapper) {  if (theme == 0) {  TypedValue outValue = new TypedValue();  context.getTheme().resolveAttribute(com.android.internal.R.attr.dialogTheme,  outValue, true);  theme = outValue.resourceId;  }  mContext = new ContextThemeWrapper(context, theme);  } else {  mContext = context;  }  mWindowManager = (WindowManager)context.getSystemService(Context.WINDOW\_SERVICE);  Window w = PolicyManager.makeNewWindow(mContext);  mWindow = w;  w.setCallback(this);  w.setOnWindowDismissedCallback(this);  w.setWindowManager(mWindowManager, null, null);  w.setGravity(Gravity.CENTER);  mListenersHandler = new ListenersHandler(this);  } |

#### 初始化DecorView并将Dialog的师徒添加到DecorView

这个过程也和activity的类似，都是通过window去指定加载的布局文件:

|  |
| --- |
| public void setContentView(int layoutResID) {  mWindow.setContentView(layoutResID);  } |

#### 将DecorView添加到window并且显示

在Dialog的show方法中，会通过windowmanager将DecorView添加到window中

|  |
| --- |
| mWindowManager.addView(mDecor, l);  mShowing = true; |

从上述的三个步骤可以发现，dialog的创建过程和activity很类似

普通的dialog有一个特殊的地方，那就是必须用activity的context，否则会报错

Dialog dialog = new Dialog(this);

dialog.setTitle("Hello");

dialog.show();



上面的信息非常的明确，是没有应用token导致的，而应用token一般只有activity持有，所有这里需要activity的context.

另外系统比较特殊，他可以不需要token. 使用过程：在AndroidManifest.xml中加上权限android.permission.SYSTEM\_ALERT\_WINDOW; 调用dialog.show()之前加alertDialog.getWindow().setType(WindowManager.LayoutParams.TYPE\_SYSTEM\_ERROR); 系统的type有很多都可以用，TYPE\_SYSTEM\_ERROR，TYPE\_SYSTEM\_OVERLAY都可以。

但是Android6.0的则需要通过代码申请权限，使用要特别注意，实在嫌麻烦，最好自己new一个activity当dlg。

### Toast的Window创建过程

Toast和dialog不同，他的工作过稍微复杂一点，首先Toast也是基于Window来实现的，但是由于Toast具有定时取消的功能，所以系统采用了handler，在toast内部有两类IPC的过程，第一类Toast访问NotificationManagerService，第二类是NotificationManagerService回调toast的TN接口

Toast具有系统的window,他内部试下的师徒有两种方式制定，一个是系统设定，还可以setview指定

|  |
| --- |
| public void show() {  if (mNextView == null) {  throw new RuntimeException("setView must have been called");  }  INotificationManager service = getService();  String pkg = mContext.getOpPackageName();  TN tn = mTN;  tn.mNextView = mNextView;  try {  service.enqueueToast(pkg, tn, mDuration);  } catch (RemoteException e) {  // Empty  }  }  /\*\*  \* Close the view if it's showing, or don't show it if it isn't showing yet.  \* You do not normally have to call this. Normally view will disappear on its own  \* after the appropriate duration.  \*/  public void cancel() {  mTN.hide();  try {  getService().cancelToast(mContext.getPackageName(), mTN);  } catch (RemoteException e) {  // Empty  }  } |

从上面的代码可以看出，显示和隐藏是通过NMS来实现的，由于NMS运行在系统，所以只能通过远程调用，这里就跨进程实现了IPC，在这个时候NMS是运行在binder线程池中，所以需要handler切换到主线程中，所以这就意味着Toast无法**在没有Lopper的线程中弹出.**

我们首先来看下Toast的显示过程

|  |
| --- |
| INotificationManager service = getService();  String pkg = mContext.getOpPackageName();  TN tn = mTN;  tn.mNextView = mNextView;  try {  service.enqueueToast(pkg, tn, mDuration);  } catch (RemoteException e) {  // Empty  } |

NMS的enqueueToast方法的第一个参数就是当前应用的包名，第二个参数表示远程回调，第三个是时长，enqueueToast首先将Toast请求封装为一个ToastRecord对象将其添加到一个队列中，其实这本书就是一个arraylist，对于非系统应用来说，他只能存50个，这样是为了防止dos,如果不这样做，试想一下，其他应用还能弹出东西来吗？

代码丢失

正常情况下，一个应用不可能达到上限，当ToastRecord被添加到mToastQueue中后，NMS就会通过showNextToastLocked方法来显示Toast，下面的代码很好理解，需要注意的是，Toast的显示是由ToastRecord的callback来完成的，这个callback实际上就是TN对象的远程Binder，通过callback来访问TN中的方法是需要跨进程来完成的，最终被调用的对象是TN中方法发起在Binder线程池中。

|  |
| --- |
| void showNextToastLocked() {  ToastRecord record = mToastQueue.get(0);  while (record != null) {  if (DBG) Slog.d(TAG, "Show pkg=" + record.pkg + " callback=" + record.callback);  try {  record.callback.show(record.token);  scheduleTimeoutLocked(record);  return;  } catch (RemoteException e) {  Slog.w(TAG, "Object died trying to show notification " + record.callback  + " in package " + record.pkg);  // remove it from the list and let the process die  int index = mToastQueue.indexOf(record);  if (index >= 0) {  mToastQueue.remove(index);  }  keepProcessAliveIfNeededLocked(record.pid);  if (mToastQueue.size() > 0) {  record = mToastQueue.get(0);  } else {  record = null;  }  }  }  } |

Toast显示出来之后，NMS会通过scheduleTimeoutLocked来发送一个延时消息

|  |
| --- |
| private void scheduleTimeoutLocked(ToastRecord r)  {  mHandler.removeCallbacksAndMessages(r);  Message m = Message.obtain(mHandler, MESSAGE\_TIMEOUT, r);  long delay = r.duration == Toast.LENGTH\_LONG ? LONG\_DELAY : SHORT\_DELAY;  mHandler.sendMessageDelayed(m, delay);  } |

上面的代码，LONG\_DELAY是3.5s，SHORT\_DELAY是2s,NMS会通过cancelToastLocked来隐藏toast并且清楚队列。

|  |
| --- |
| ToastRecord record = mToastQueue.get(index);  try {  record.callback.hide();  } catch (RemoteException e) {  Slog.w(TAG, "Object died trying to hide notification " + record.callback  + " in package " + record.pkg);  // don't worry about this, we're about to remove it from  // the list anyway  } |

通过上面的分析，大家知道toast的显示隐藏实际上是toast的TN这个类来实现的，分别对应的show/hide，由于这两个方法都是NMS以跨进程的方式调用的，因此他运行在Binder池中，为了切换，在他们内部使用了handler

|  |
| --- |
| @Override  public void show(IBinder windowToken) {  if (localLOGV) Log.v(TAG, "SHOW: " + this);  mHandler.obtainMessage(0, windowToken).sendToTarget();  }  /\*\*  \* schedule handleHide into the right thread  \*/  @Override  public void hide() {  if (localLOGV) Log.v(TAG, "HIDE: " + this);  mHandler.post(mHide);  } |

上述的代码，mShow和mHide是两个Runnable，他们的内部实现分别调用了具体方法，由此可见，handlershow才是真正的方法，

|  |
| --- |
| public void handleShow(IBinder windowToken) {  if (localLOGV) Log.v(TAG, "HANDLE SHOW: " + this + " mView=" + mView  + " mNextView=" + mNextView);  if (mView != mNextView) {  // remove the old view if necessary  handleHide();  mView = mNextView;  Context context = mView.getContext().getApplicationContext();  String packageName = mView.getContext().getOpPackageName();  if (context == null) {  context = mView.getContext();  }  mWM = (WindowManager)context.getSystemService(Context.WINDOW\_SERVICE);  // We can resolve the Gravity here by using the Locale for getting  // the layout direction  final Configuration config = mView.getContext().getResources().getConfiguration();  final int gravity = Gravity.getAbsoluteGravity(mGravity, config.getLayoutDirection());  mParams.gravity = gravity;  if ((gravity & Gravity.HORIZONTAL\_GRAVITY\_MASK) == Gravity.FILL\_HORIZONTAL) {  mParams.horizontalWeight = 1.0f;  }  if ((gravity & Gravity.VERTICAL\_GRAVITY\_MASK) == Gravity.FILL\_VERTICAL) {  mParams.verticalWeight = 1.0f;  }  mParams.x = mX;  mParams.y = mY;  mParams.verticalMargin = mVerticalMargin;  mParams.horizontalMargin = mHorizontalMargin;  mParams.packageName = packageName;  mParams.hideTimeoutMilliseconds = mDuration ==  Toast.LENGTH\_LONG ? LONG\_DURATION\_TIMEOUT : SHORT\_DURATION\_TIMEOUT;  mParams.token = windowToken;  if (mView.getParent() != null) {  if (localLOGV) Log.v(TAG, "REMOVE! " + mView + " in " + this);  mWM.removeView(mView);  }  if (localLOGV) Log.v(TAG, "ADD! " + mView + " in " + this);  mWM.addView(mView, mParams);  trySendAccessibilityEvent();  }  } |

移除

|  |
| --- |
| public void handleHide() {  if (localLOGV) Log.v(TAG, "HANDLE HIDE: " + this + " mView=" + mView);  if (mView != null) {  // note: checking parent() just to make sure the view has  // been added... i have seen cases where we get here when  // the view isn't yet added, so let's try not to crash.  if (mView.getParent() != null) {  if (localLOGV) Log.v(TAG, "REMOVE! " + mView + " in " + this);  mWM.removeViewImmediate(mView);  }  mView = null;  }  }  } |

到这里，我们的window就全部分析完成了，相信大家对window有了一些新的见解了。

## 常见使用注意

1. 用Application作为dlg的context：
2. 在子线程更新ui
3. 实现系统转屏功能：

**ViewRootImpl**

**mWm** = WindowManagerGlobal.*getWindowManagerService*();

**mWm** = IWindowManager.Stub.*asInterface*(ServiceManager.*getService*(**"window"**));

**mWm**.freezeRotation(2);//180度

## REF

[Android设计模式（二） 续：WindowManager](http://www.jianshu.com/p/3e9f068ed82d)

Dlg:

<http://blog.csdn.net/yanbober/article/details/46361191>

<http://blog.csdn.net/hwliu51/article/details/75040297>

http://www.jianshu.com/p/672e6486a72a

# TASK

[《深入理解Android 卷III》第四章 深入理解WindowManagerService](https://blog.csdn.net/Innost/article/details/47660193)

Android事件分发机制.ppt