关键字：android PhoneWindowManager组合键

# 模块分析

## mOrientationListener

发现在初始化的过程中，实例化一个方位监听对象MyOrientationListener，并且给它设置了初始值。该初始值从WMS中获取，默认为

mOrientationListener = new MyOrientationListener(mContext, mHandler);

try {

mOrientationListener.setCurrentRotation(windowManager.getRotation());

} catch (RemoteException ex) { }

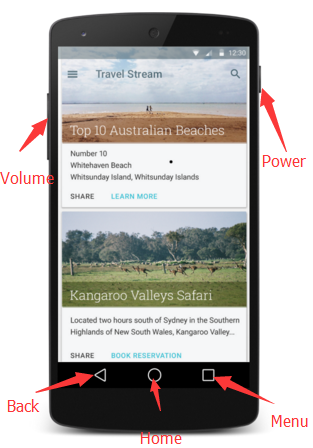
mWindowManager.updateRotation(alwaysSendConfiguration, false);

# 概述

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。

## 按键事件分类

在Android 中会有以下5个按键（Back、Home、Menu/recent、Power、Volume）与用户进行交互，Framework层中实现按键功能



按照时间

SHORT\_PRESS

LONG\_PRESS：LONG\_PRESS\_BACK\_GO\_TO\_VOICE\_ASSIST

MULTI\_PRESS：MULTI\_PRESS\_POWER\_BRIGHTNESS\_BOOST

DOUBLE\_TAP：

按照按键类型

## 配置

core/res/res/values/config.xml

mGoToSleepOnButtonPressTheaterMode = mContext.getResources().getBoolean(

com.android.internal.R.bool.config\_goToSleepOnButtonPressTheaterMode);

mSupportLongPressPowerWhenNonInteractive = mContext.getResources().getBoolean(

com.android.internal.R.bool.config\_supportLongPressPowerWhenNonInteractive);

mLongPressOnBackBehavior = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_longPressOnBackBehavior);

mShortPressOnPowerBehavior = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_shortPressOnPowerBehavior);

mLongPressOnPowerBehavior = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_longPressOnPowerBehavior);

mDoublePressOnPowerBehavior = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_doublePressOnPowerBehavior);

mTriplePressOnPowerBehavior = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_triplePressOnPowerBehavior);

mShortPressOnSleepBehavior = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_shortPressOnSleepBehavior);

## 外部接口

boolean isDeviceProvisioned() {

return Settings.Global.getInt(

mContext.getContentResolver(), Settings.Global.DEVICE\_PROVISIONED, 0) != 0;

}

boolean isUserSetupComplete() {

return Settings.Secure.getIntForUser(mContext.getContentResolver(),

Settings.Secure.USER\_SETUP\_COMPLETE, 0, UserHandle.USER\_CURRENT) != 0;

}

### SystemGesturesPointerEventListener

# 事件拦截原理

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,

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。

interceptKeyBeforeDispatching

# 如何启动

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。



**[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();

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,

# 类分析

## 变量

private static final String SYSUI\_PACKAGE = "com.android.systemui";

private static final String SYSUI\_SCREENSHOT\_SERVICE =

"com.android.systemui.screenshot.TakeScreenshotService";

private static final String SYSUI\_SCREENSHOT\_ERROR\_RECEIVER =

"com.android.systemui.screenshot.ScreenshotServiceErrorReceiver";

## interceptKeyBeforeDispatching

f (keyCode == KeyEvent.KEYCODE\_BRIGHTNESS\_UP

|| keyCode == KeyEvent.KEYCODE\_BRIGHTNESS\_DOWN

launchHomeFromHotKey

interceptKeyBeforeQueueing有组合键

## interceptKeyBeforeQueueing

@Override public void systemBooted() 把mSystemBooted = true，

Interactive:可拦截的？

Interactive 表示屏幕是否点亮，这个变量表示PowerManagerService关于屏幕的状态

Down 表示是否是ActionDown事件

Canceled 事件是否取消

interceptKeyBeforeQueueing(KeyEvent event, int policyFlags)

**[java]** [view plain](http://blog.csdn.net/liu149339750/article/details/49944271) [copy](http://blog.csdn.net/liu149339750/article/details/49944271)

1. if (!mSystemBooted) {
2. // If we have not yet booted, don't let key events do anything.
3. return 0;
4. }
5. //
6. final boolean interactive = (policyFlags & FLAG\_INTERACTIVE) != 0;
7. final boolean down = event.getAction() == KeyEvent.ACTION\_DOWN;
8. final boolean canceled = event.isCanceled();
9. final int keyCode = event.getKeyCode();
10. final boolean isInjected = (policyFlags & WindowManagerPolicy.FLAG\_INJECTED) != 0;
11. // If screen is off then we treat the case where the keyguard is open but hidden
12. // the same as if it were open and in front.
13. // This will prevent any keys other than the power button from waking the screen
14. // when the keyguard is hidden by another activity.
15. final boolean keyguardActive = (mKeyguardDelegate == null ? false :
16. (interactive ?
17. isKeyguardShowingAndNotOccluded() :
18. mKeyguardDelegate.isShowing()));
19. if (DEBUG\_INPUT) {
20. Log.d(TAG, "interceptKeyTq keycode=" + keyCode
21. + " interactive=" + interactive + " keyguardActive=" + keyguardActive
22. + " policyFlags=" + Integer.toHexString(policyFlags));
23. }
24. // Basic policy based on interactive state.
25. int result;
26. boolean isWakeKey = (policyFlags & WindowManagerPolicy.FLAG\_WAKE) != 0
27. || event.isWakeKey();
28. if (interactive || (isInjected && !isWakeKey)) {
29. // When the device is interactive or the key is injected pass the
30. // key 给APP
31. result = ACTION\_PASS\_TO\_USER;
32. isWakeKey = false;
33. if (interactive) {
34. // If the screen is awake, but the button pressed was the one that woke the device
35. // 不给APP，拦截掉了
36. if (keyCode == mPendingWakeKey && !down) {
37. result = 0;
38. }
39. // Reset the pending key
40. mPendingWakeKey = PENDING\_KEY\_NULL;
41. }
42. } else if (!interactive && shouldDispatchInputWhenNonInteractive(event)) {
43. // If we're currently dozing with the screen on and the keyguard showing, pass the key
44. // to the application but preserve its wake key status to make sure we still move
45. // from dozing to fully interactive if we would normally go from off to fully
46. // interactive.
47. result = ACTION\_PASS\_TO\_USER;
48. // Since we're dispatching the input, reset the pending key
49. mPendingWakeKey = PENDING\_KEY\_NULL;
50. } else {
51. // When the screen is off and the key is not injected, determine whether
52. // to wake the device but don't pass the key to the application.
53. result = 0;
54. if (isWakeKey && (!down || !isWakeKeyWhenScreenOff(keyCode))) {
55. isWakeKey = false;
56. }
57. // Cache the wake key on down event so we can also avoid sending the up event to the app
58. if (isWakeKey && down) {
59. mPendingWakeKey = keyCode;
60. }
61. }
62. // If the key would be handled globally, just return the result, don't worry about special
63. // key processing.
64. if (isValidGlobalKey(keyCode)
65. && mGlobalKeyManager.shouldHandleGlobalKey(keyCode, event)) {
66. if (isWakeKey) {
67. wakeUp(event.getEventTime(), mAllowTheaterModeWakeFromKey, "android.policy:KEY");
68. }
69. return result;
70. }
71. boolean useHapticFeedback = down
72. && (policyFlags & WindowManagerPolicy.FLAG\_VIRTUAL) != 0
73. && event.getRepeatCount() == 0;
74. // Handle special keys.

## **InputEventReceiver**

劫持输入事件的类！在其下面初始化了一个变量，

final InputEventReceiver.Factory mHideNavInputEventReceiverFactory =

new InputEventReceiver.Factory() {

@Override

public InputEventReceiver createInputEventReceiver(

InputChannel inputChannel, Looper looper) {

return new HideNavInputEventReceiver(inputChannel, looper);

}

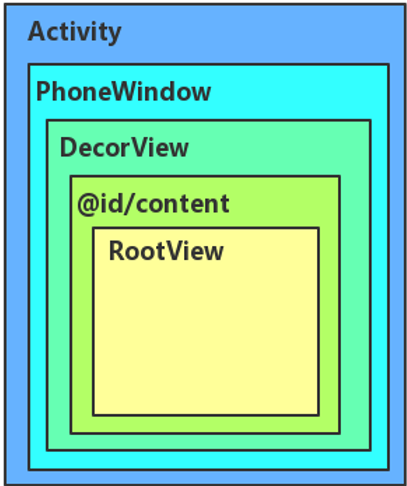
};

# 输入事件分发机制

## View层级

DecorView：整个Window界面的最顶层View，包括系统状态栏、底部虚拟键等和一个@id/content的FrameLayout。

RootView：根View，就是我们Activity里设置的布局View



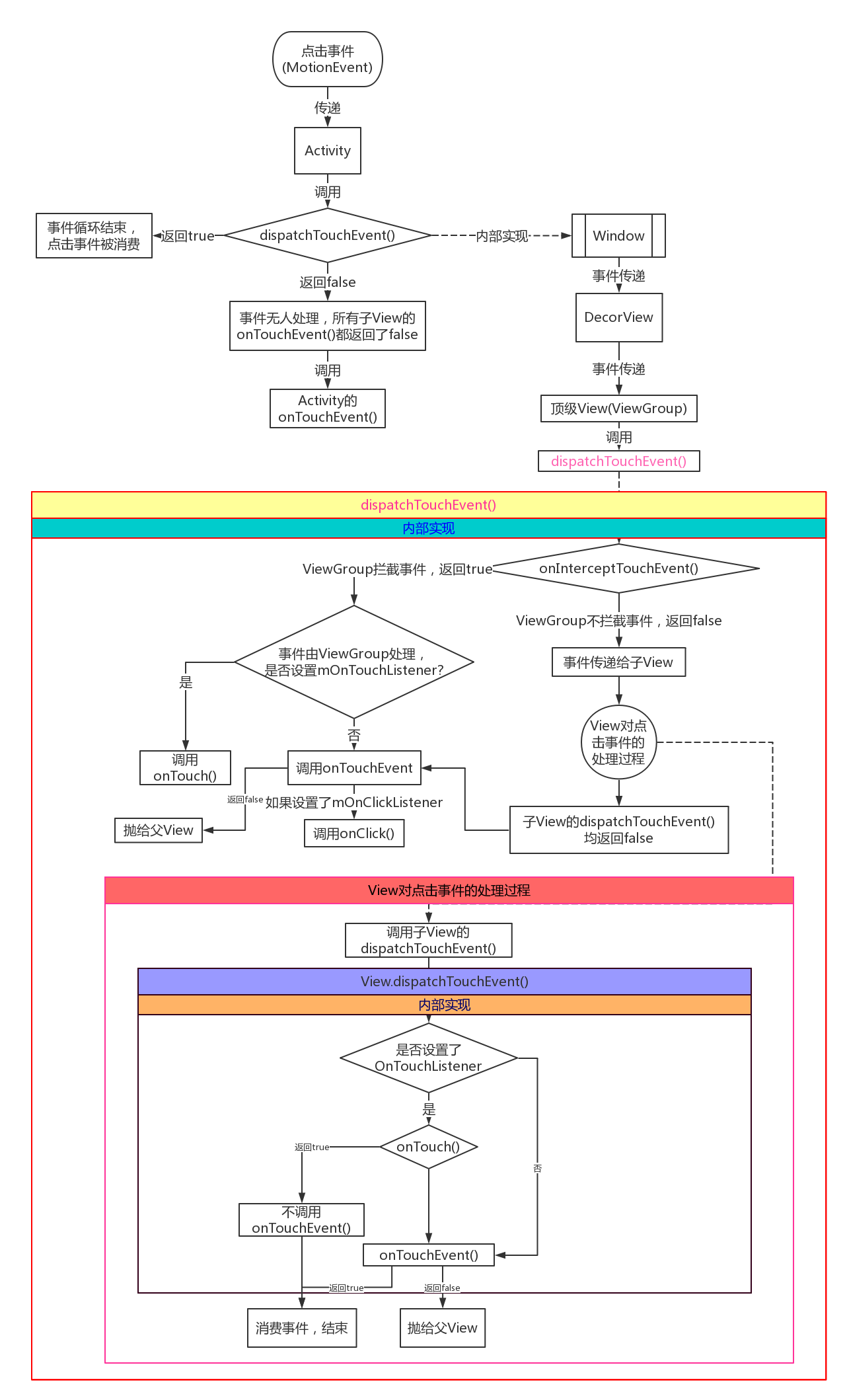
## 事件分发

自顶向下，一旦消费即停止

事件拦截：onInterceptTouchEvent()

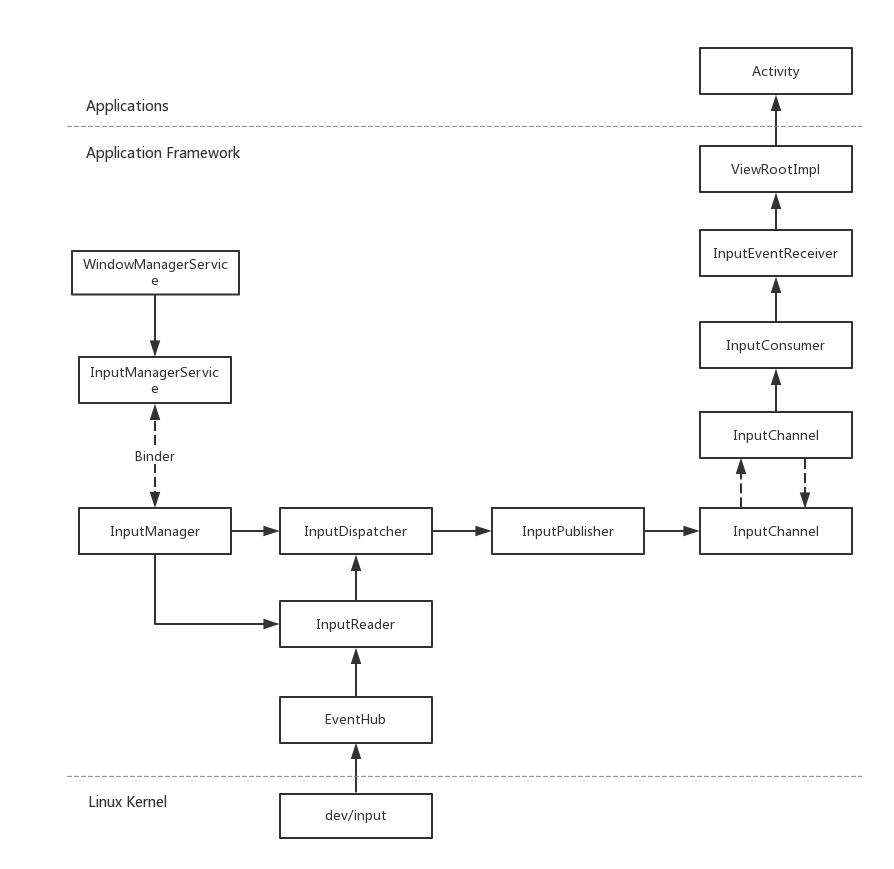
事件分发：dispatchTouchEvent()

事件响应：onTouchEvent()



# 系统层的事件传递

Android事件分发机制的本质是要解决：点击事件由哪个对象发出，经过哪些对象，最终达到哪个对象并最终得到处理。



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

## 服务启动

### InputManagerService

伴随着system\_server进程的启动而启动，整个调用过程：

### WindowManagerService

int menuState = mInputManager.getKeyCodeState(-1, InputDevice.SOURCE\_ANY,

KeyEvent.KEYCODE\_MENU);

int sState = mInputManager.getKeyCodeState(-1, InputDevice.SOURCE\_ANY, KeyEvent.KEYCODE\_S);

int dpadState = mInputManager.getKeyCodeState(-1, InputDevice.SOURCE\_DPAD,

KeyEvent.KEYCODE\_DPAD\_CENTER);

int trackballState = mInputManager.getScanCodeState(-1, InputDevice.SOURCE\_TRACKBALL,

InputManagerService.BTN\_MOUSE);

int volumeDownState = mInputManager.getKeyCodeState(-1, InputDevice.SOURCE\_ANY,

KeyEvent.KEYCODE\_VOLUME\_DOWN);

# 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。

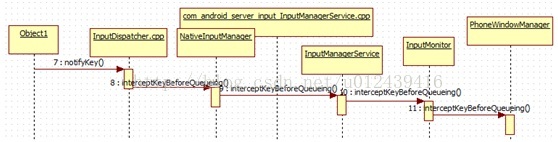
看下InputManagerService的setWindowManagerCallbacks方法,

**[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。

# 情景分析

[当界面全屏时，在顶部下拉时会显示statusbar的实现原理](http://aijiawang-126-com.iteye.com/blog/2393863)

## SystemGesturesPointerEventListenet接收事件原理

## 下来调起systemui原理

SystemGesturesPointerEventListener中的onSwipeFromTop

### requestTransientBars

requestTransientBars(mStatusBar);

{

synchronized (mWindowManagerFuncs.getWindowManagerLock()) {

if (!isUserSetupComplete()) {

// Swipe-up for navigation bar is disabled during setup

return;

}

**boolean sb = mStatusBarController.checkShowTransientBarLw();**

boolean nb = mNavigationBarController.checkShowTransientBarLw()

&& !isNavBarEmpty(mLastSystemUiFlags);

if (sb || nb) {

// Don't show status bar when swiping on already visible navigation bar

if (!nb && swipeTarget == mNavigationBar) {

if (DEBUG) Slog.d(TAG, "Not showing transient bar, wrong swipe target");

return;

}

if (sb) mStatusBarController.showTransient();

if (nb) mNavigationBarController.showTransient();

mImmersiveModeConfirmation.confirmCurrentPrompt();

updateSystemUiVisibilityLw();

}

## 长按原理

# 客制化

## 总体思路

找到PhoneWindowManager，会发现这里对于keyevent的各种intercept简直就是hook的绝佳地点：

interceptKeyBeforeDispatching(WindowState win, KeyEvent event, int policyFlags)

interceptKeyBeforeQueueing(KeyEvent event, int policyFlags)

等。。。

找到interceptKeyBeforeDispatching对volume down + power的处理（或者直接参考mHomeDoubleTapTimeoutRunnable），参考即可（可以顺着mVolumeDownKeyTriggered去看在哪里触发，在哪里判定tolerance的）

基本流程就是：

volume down先被点击的时候，记录下来时间；

power点击的时候，判断是否在tolerance里面；

同理，power先被点击的时候，一样记录下来时间；

在volume down被点击的时候，判断是否在tolerance里面；

这样就确定了是否是截屏“操作”，如果是，就consume这个事件（不再向上转发，直接return掉）；同时通过mScreenshotRunnable去截屏。

同样，双击事件也一样（甚至更简单），前一次点击纪录下来时间，下次点击计算是否在tolerance中即可（就不贴代码了，因为太懒（烂）了。。。）

其实L的代码，在window manager这里已经实现了volume down + power、volume up + power、 double tap home；参考对应的来实现一个即可。

还有一点就是如何在屏幕没点亮的时候监听按键；

在interceptKeyBeforeQueueing中找到interactive，（还没试，应该是用这个来决定是否screen on）

就这个啦，即使这个为false的时候依然把keyevent传递出去就好了~ （或者简单点就参考power键~ ）

## 打开 或者 关闭 Navigation Bar

解决方法：

### 1. 修改config.xml 文件中

搜索关键字config\_showNavigationBar， 查看 config\_showNavigationBar 值  
true 表示显示,false 表示不显示

<!-- Whether a software navigation bar should be shown. NOTE: in the future this may be

autodetected from the Configuration. -->

<bool name="config\_showNavigationBar">true</bool>

参考路径如下：  
alps\frameworks\base\core\res\res\values\config.xml

### 2. 修改 system.prop 文件

查询关键字 qemu.hw.mainkeys，并查看值，1表示关闭0.表示开启 。

# temporary enables NAV bar (soft keys)

qemu.hw.mainkeys=1

### 3.修改PhoneWindowManager代码

如果上面两个修改都不生效（搜索关键字config\_showNavigationBar、qemu.hw.mainkeys），请在PhoneWindowManager 查看setInitialDisplaySize方法中mHasNavigationBar 的值是否被写死，true表示会显示、false表示不显示导航栏。

### 其他

<https://blog.csdn.net/longtian635241/article/details/47981603>

输入事件接收器InputEventReceiver，这是我们发现HideNavInputEventReceiver这个类，它就是劫持输入事件的类！在其下面初始化了一个变量：

final InputEventReceiver.Factory mHideNavInputEventReceiverFactory =

new InputEventReceiver.Factory() {

@Override

public InputEventReceiver createInputEventReceiver(

InputChannel inputChannel, Looper looper) {

return new HideNavInputEventReceiver(inputChannel, looper);

}

};

## 长按Home 键启动Google Now

### 预制 Google Now APK

请自行安装APK

### 修改启动代码launchAssistLongPressAction

private void launchAssistLongPressAction() {

// launch the search activity

Intent intent = new Intent(Intent.ACTION\_SEARCH\_LONG\_PRESS);

intent.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);

try {

startActivityAsUser(intent, UserHandle.CURRENT);

} catch (ActivityNotFoundException e) {

Slog.w(TAG, "No activity to handle assist long press action.", e);

}

}

### 事件分发之前interceptKeyBeforeDispatching处理

在按键分发处理之前调用自定义长按Home键的方法

if (keyCode == KeyEvent.KEYCODE\_ASSIST) {

if (down) {

if (repeatCount == 0) {

mAssistKeyLongPressed = false;

} else if (repeatCount == 1) {

mAssistKeyLongPressed = true;

if (!keyguardOn) {

launchAssistLongPressAction();

}

}

}

## 双击Home 键调出最近任务列表请用以下方法

### interceptKeyBeforeQueueing 方法中修改

if (keyCode == KeyEvent.KEYCODE\_HOME) {

// Remember that home is pressed and handle special actions.

if (repeatCount == 0) {

mHomePressed = true;

if (mHomeDoubleTapPending) {

mHomeDoubleTapPending = false;

mHandler.removeCallbacks(mHomeDoubleTapTimeoutRunnable);

handleDoubleTapOnHome();

}

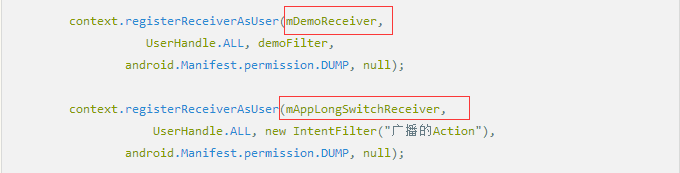
}

## 长按实体Menu键进入多窗口模式

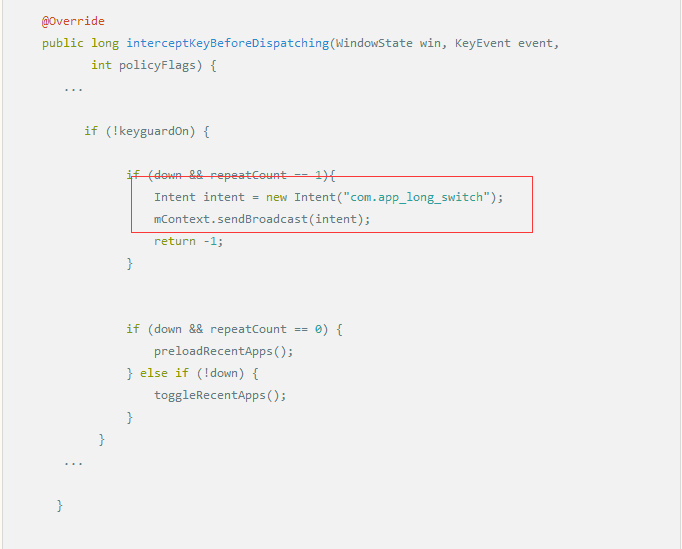
Android N上支持Multi-Window，通过recent key进入多窗口，对于没有打开虚拟导航栏，只有实体menu按键的手机，可以考虑向SystemUI发送广播的形式，进入Android 分屏多任务模式。  
解决方案如下：

1.PhoneStatusBar 里注册广播SystemUI模块的代码

动态注册广播方法如下，参考系统mDemoReceiver

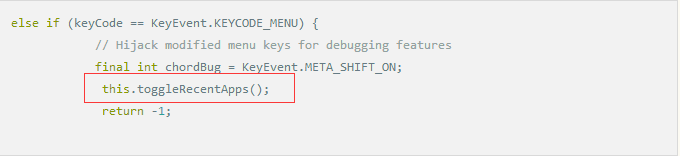


PhoneWindowManager 中发送广播

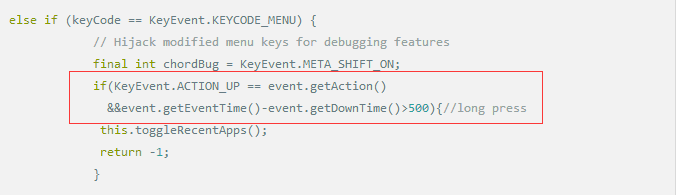


## 点击 Menu键进入调出最近任务列

需要拦截menu的事件，在PhoneWindowManager的interceptKeyBeforeDispatching 中处理即可



如果想长按Menu调出可以使用以下方法

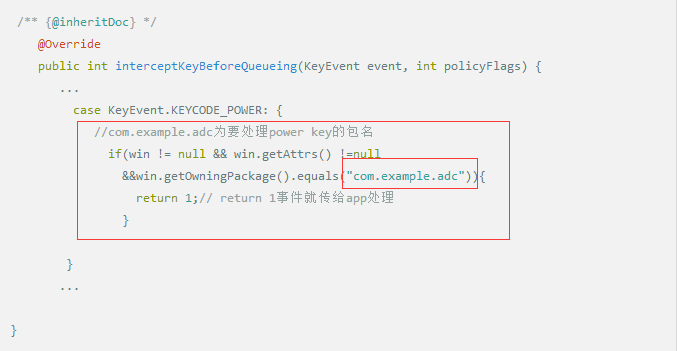


## 如何让 App 拿到Power key 值

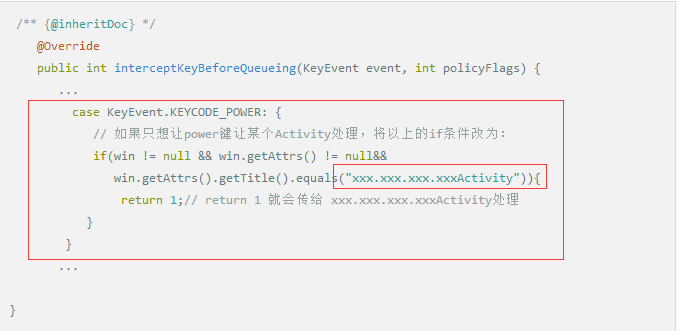
一般情况下App是拿不到Power的Key值，但通过以下方法可以实现

长按返回键处理流程

interceptKeyBeforeQueueing方法实现让特定的APP拿到Power key 值



只想让某个app的某个Activity 处理



## 修Activity启动是的窗口（app启动白屏，黑屏问题）

当用户从主菜单进入其他应用程序例如时钟、联系人、文件管理等时，可能会出现屏幕闪一下黑屏、白屏等问题，这种现象在当前手机主题(Theme)是浅色（例如白色）的情况下比较明显。

此所谓的闪"黑屏",其实是应用程序的启动窗口。  
启动窗口出现的条件如下：

1. 仅在要启动的Activity在新的Task或者新的Process时，才可能显示启动窗口
2. 启动窗口先于Activity窗口显示，当Activity窗口的内容准备好之后，启动窗口就会被移除掉，show出真正的activity 窗口
3. 启动窗口和普通的Activity window类似，只是没有画任何内容，默认是一个黑色背景的窗口

正是由于启动窗口默认是黑色背景的，所以在当前的手机主题为浅色调的时候，就比较容易因为颜色的深浅对比而产生一种视觉上的闪动感。

解决方法如下：

### 去掉启动窗口

在 ActivityStack.java中将SHOW\_APP\_STARTING\_PREVIEW设置为false既可

出现了这样的问题，说明，app本身设计得不行！！！优化项目，启动和stop时间，开线程去做

// How long between activity launches that we consider safe to not warn

// the user about an unexpected activity being launched on top.

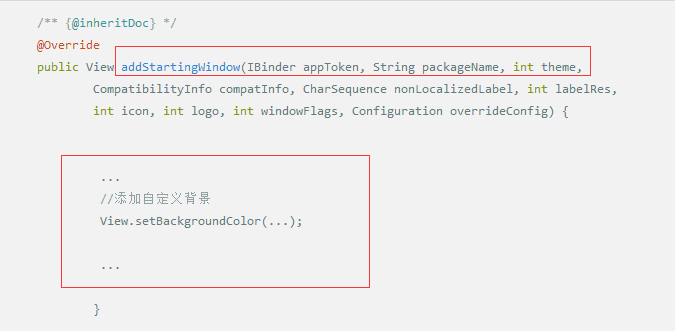
static final long START\_WARN\_TIME = 5 \* 1000;

// Set to false to disable the preview that is shown while a new activity

// is being started.

static final boolean SHOW\_APP\_STARTING\_PREVIEW = true;

## 修改启动窗口样式

在 PhoneWindowManager中的addStartingWindow方法中添加自定义样式或者背景等 

try {

Context context = mContext;

if (DEBUG\_STARTING\_WINDOW) Slog.d(TAG, "addStartingWindow " + packageName

+ ": nonLocalizedLabel=" + nonLocalizedLabel + " theme="

+ Integer.toHexString(theme));

if (theme != context.getThemeResId() || labelRes != 0) {

try {

context = context.createPackageContext(packageName, 0);

context.setTheme(theme);

} catch (PackageManager.NameNotFoundException e) {

// Ignore

}

}

if (overrideConfig != null && overrideConfig != EMPTY) {

if (DEBUG\_STARTING\_WINDOW) Slog.d(TAG, "addStartingWindow: creating context based"

+ " on overrideConfig" + overrideConfig + " for starting window");

final Context overrideContext = context.createConfigurationContext(overrideConfig);

overrideContext.setTheme(theme);

final TypedArray typedArray = overrideContext.obtainStyledAttributes(

com.android.internal.R.styleable.Window);

final int resId = typedArray.getResourceId(R.styleable.Window\_windowBackground, 0);

if (resId != 0 && overrideContext.getDrawable(resId) != null) {

// We want to use the windowBackground for the override context if it is

// available, otherwise we use the default one to make sure a themed starting

// window is displayed for the app.

if (DEBUG\_STARTING\_WINDOW) Slog.d(TAG, "addStartingWindow: apply overrideConfig"

+ overrideConfig + " to starting window resId=" + resId);

context = overrideContext;

}

}

final PhoneWindow win = new PhoneWindow(context);

修改启动窗口样式

# UI定制方案

Com.android.server.policy/globalActions.java

# REF

[Android 手机按键客制化详解](链接：https:/www.jianshu.com/p/3bda276cb68e)

[Android支持按键双击事件](http://nobodycare.me/2015/06/20/android-support-double-tap/)

# 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

[Power长按、组合键分析](https://blog.csdn.net/qq_30427341/article/details/77962749)

<https://blog.csdn.net/whut_fn/article/details/44218701>

<https://blog.csdn.net/s278777851/article/details/6956226>

https://blog.csdn.net/luoshengyang/article/details/6882903