# Todo

Android Media Projection服务漏洞深度剖析

<https://4hou.win/wordpress/?cat=3133>

https://blog.csdn.net/jiangwei0910410003/article/details/52135654

com.android.systemui.media.MediaProjectionPermissionActivity

http://www.codexiu.cn/android/blog/20180/

06-30 18:27:24.983 2381-2421/com.android.systemui W/Binder: Binder call failed.

java.lang.SecurityException: Permission Denial: not allowed to send broadcast android.intent.action.USER\_PRESENT from pid=2381, uid=10043

at android.os.Parcel.readException(Parcel.java:1684)

at android.os.Parcel.readException(Parcel.java:1637)

at android.app.ActivityManagerProxy.broadcastIntent(ActivityManagerNative.java:3550)

at android.app.ContextImpl.sendBroadcastAsUser(ContextImpl.java:1042)

at android.content.ContextWrapper.sendBroadcastAsUser(ContextWrapper.java:489)

at com.android.systemui.keyguard.KeyguardViewMediator.sendUserPresentBroadcast(KeyguardViewMediator.java:1591)

at com.android.systemui.keyguard.KeyguardViewMediator.maybeSendUserPresentBroadcast(KeyguardViewMediator.java:983)

at com.android.systemui.keyguard.KeyguardViewMediator.onSystemReady(KeyguardViewMediator.java:750)

at com.android.systemui.keyguard.KeyguardService$1.onSystemReady(KeyguardService.java:170)

at com.android.internal.policy.IKeyguardService$Stub.onTransact(IKeyguardService.java:156)

# SystemUI

系统版本：Android 7.0 SystemUI

尽管SystemUI的表现形式与普通的Android应用程序大相径庭，但它却是以一个APK的形式存在于系统之中，即它与普通的Android应用程序并没有本质上的区别。无非是通过Android四大组件中的Activity、Service、BroadcastReceiver接受外界的请求并执行相关的操作，只不过它们所接受到的请求主要来自各个系统服务而已。

## 功能描述

SystemUI是为用户提供系统级别的信息显示与交互的一套UI组件，因此它所实现的功能包罗万象，主要功能如下：

|  |  |
| --- | --- |
| 大类 | 功能 |
| 快捷开关QSPanel | 亮度WiFi蓝牙数据飞行模式投射屏幕  手电筒位置信息设置 |
| 通知面板 | 系统消息，第三方应用消息，都是在通知面板显示 |
| 状态栏(Icon) | 时间电量信号蓝牙应用IconWiFiGPS |
| 导航栏NavigationBar | 返回Back、主键Home、最近应用 Recent |
| 最近应用RecentPanel | 按需启动，完成其既定任务后退出，与普通的Activity以及Service别无二致。 |
| 其他 | 部分锁屏，壁纸, 屏保设置，USB，截屏和音量  TakeScreenshotService的服务，用于在用户按下音量下键与电源键时进行截屏操作。  图片壁纸 |
| PowerUI | 负责监控系统的剩余电量并在必要时为用户显示低电警告 |
| RingtonePlayer | 依托AudioService为向其他应用程序提供播放铃声的功能。 |

尽管SystemUI的表现形式与普通的Android应用程序大相径庭，但它却是以一个APK的形式存在于系统之中，即它与普通的Android应用程序并没有本质上的区别。无非是通过Android四大组件中的Activity、Service、BroadcastReceiver接受外界的请求并执行相关的操作，只不过它们所接受到的请求主要来自各个系统服务而已。

大部分功能之间相互独立，比较特殊的是状态栏、导航栏等组件的启动方式。它们运行于一个称之为SystemUIService的一个Service之中。因此讨论状态栏与导航栏的启动过程其实就是SystemUIService的启动过程。

## 代码配置

### 主要目录

源码路径:/framework/base/packages/SystemUI

apk路径:/system/priv-app

### Android.mk

1)应用编译后的名称：SysemUI

LOCAL\_PACKAGE\_NAME := SystemUI

2)应用编译后系统路径：/system/pri-app/SystemUI

LOCAL\_PRIVILEGED\_MODULE := true

说明：不添加此属性，编译后的路径为/system/app/SystemUI，/system/pri-app下的apk权限比/system/app下大

3)在Android.mk文件里面添加LOCAL\_CERTIFICATE := platform（使用系统签名）

### AndroidManifest.xml

1. SystemUI属于系统apk：android:sharedUserId="android.uid.systemui"
2. systemUI属于核心apk：coreApp="true"
3. SystemUI不会被KILL，即被KILL后会自动重启： android:persistent="true"，这里的KILL主要指不会被low memory killer杀掉
4. 进程名是com.android.systemui： android:process="com.android.systemui"
5. 对外开放的Service主要有: SystemUIService/ LoadAverageService/ ImageWallpaper，截图服务居然没有对外开放，看来只有系统进程才可以启动并使用这个服务
6. 常见Activity :

|  |  |  |
| --- | --- | --- |
|  | Activity | 备注 |
| 最近应用程序 | recents.RecentsActivity | android:launchMode="singleInstance"全局单例 |
| usb | .usb.UsbConfirmActivity，  .usb.UsbPermissionActivity  .usb.UsbResolverActivity，  .usb.UsbAccessoryUriActivity" UsbDebuggingActivity，  UsbDebuggingSecondaryUserActivity | 比如手机在电脑上插入，手机上的提示框；adb connect到手机，手机授权提示框等。 |
| 亮度调节 | .settings.BrightnessDialog |  |

## 架构分析

根目录：frameworks/base/packages/SystemUI

SystemUI/src/com/android/systemui/statusbar/phone/PhoneStatusBar.java

base/packages/SystemUI/src/com/android/systemui/SystemUIService.java

frameworks/base/services/java/com/android/server/SystemServer.java

packages/SystemUI/src/com/android/systemui/statusbar/SystemBars.java

packages/SystemUI/src/com/android/systemui/statusbar/BaseStatusBar.java

packages/SystemUI/src/com/android/systemui/SystemUIApplication.java

base/packages/SystemUI/src/com/android/systemui/qs/QSPanel.java

src/com/android/systemui/statusbar/policy/BrightnessMirrorController.java

packages/SystemUI/src/com/android/systemui/recent/RecentsPanelView.java

SystemUI/src/com/android/systemui/statusbar/phone/NavigationBarView.java

packages/SystemUI/src/com/android/systemui/volume/VolumePanel.java

frameworks/base/packages/SystemUI/res/layout/super\_status\_bar.xml

frameworks/base/packages/SystemUI/res/layout/navigation\_bar.xml

frameworks/base/packages/SystemUI/res/layout/system\_icons.xml

frameworks/base/packages/SystemUI/res/layout/status\_bar\_expanded.xml

frameworks/base/packages/SystemUI/res/layout/status\_bar\_expanded\_header.xml

frameworks/base/packages/SystemUI/res/layout/qs\_panel.xml

### SystemUI类图

SystemUI是一个抽象类，定义了模板和相关抽象方法：成员变量mComponents为何是一个Map？成员变量mContext和mcompenent都是对SystemUIApplication的引用

（1）状态的核心类是BaseStatusBar，这个类是一个抽象类。它的start()方法(继承自SystemUI)定义了状态栏启动时的具体步骤。

（2）BaseStatusBar继承自SystemUI，SystemUI被SystemUIService调用，SystemUIService继承Service，所以StatuBar也是一个Service。SystemUIService实现非常简单，只是在onCreat的时候调用SystemUIApplication.startServiceIfNeeded()

（3）BaseStatusBar实现了CommandQueue.Callbacks接口，同时可以发现CommandQueue继承自IStatusBar.Stub远程接口，而IStatusBar.Stub接口的方法则通过CommandQueue的Callbacks接口实现，所以说BaseStatusBar又是IStatusBar.stub远程接口的实现类。

（4）我们说BaseStatusBar是抽象类，那么IStatusBar.stub接口中方法的实现该如何实现呢？很简单，我们可以通过StatuBar的两个子类：PhoneStatusBar和TabletStatusBar来实现。

（5）另外KeyguardViewMediator, RingtonePlayer, VolumeUI, SystemBars, PowerUI, StorageNotification, Recents 这几个UI组件也继承自 SystemUI，上面没有画出来。

### SystemUI 启动时机

#### SystemServer启动SystemUIService服务

SystemServer.startOtherServices()->startSystemUi:启动SystemUIService

1. **static** **final** **void** startSystemUi(Context context) {
2. Intent intent = **new** Intent();
3. intent.setComponent(**new** ComponentName("com.android.systemui",
4. "com.android.systemui.SystemUIService"));
5. context.startServiceAsUser(intent, UserHandle.OWNER);
6. }

Context = mSystemContext = ActivityThread.systemMain().getSystemContext（）

== ContextImpl.createContext(ActivityThread.this)

== new ContextImpl()

可见，当核心的系统服务启动完毕后，ServerThread通过Context.startServiceAsUser()方法完成了SystemUIService的启动。

#### AMS启动SystemUIApplication

**SystemUIApplication**是SystemUI的Application，在AndroidMenifest.xml的Application注册了，启动SystemUIService的时候，AMS就会先启动这个应用SystemUIApplication。

这个类在onCreate()监听回调（通过高优先级的广播监听开机是否完成），在onConfigurationChanged（）回调Component的onConfigurationChanged方法。

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **private** **final** Class<?>[] SERVICES = **new** Class[] {
2. com.android.systemui.keyguard.KeyguardViewMediator.**class**,
3. **com.android.systemui.recent.Recents**.**class**,
4. com.android.systemui.volume.VolumeUI.**class**,
5. com.android.systemui.statusbar.SystemBars.**class**,
6. com.android.systemui.usb.StorageNotification.**class**,
7. com.android.systemui.power.PowerUI.**class**,
8. com.android.systemui.media.RingtonePlayer.**class**
9. };
10. **private** **final** SystemUI[] mServices = **new** SystemUI[SERVICES.length];
11. @Override
12. **public** **void** **onCreate**() {
13. **super**.onCreate();
14. IntentFilter filter = **new** IntentFilter(Intent.ACTION\_BOOT\_COMPLETED);
15. filter.setPriority(IntentFilter.SYSTEM\_HIGH\_PRIORITY);
16. registerReceiver(**new** BroadcastReceiver() {
17. @Override
18. **public** **void** onReceive(Context context, Intent intent) {
19. unregisterReceiver(**this**);
20. **for** (**int** i = 0; i < N; i++) {
21. mServices[i].onBootCompleted();
22. } }}, filter);
23. }
24. @Override
25. **public** **void** **onConfigurationChanged**(Configuration newConfig) {
26. **for** (**int** i = 0; i < len; i++) {
27. mServices[i].onConfigurationChanged(newConfig);
28. }
29. }

// 0号元素存储的其实是一个字符串资源号，这个字符串资源存储了实现了状态栏/导航栏的类名

#### AMS启动SystemUIService

启动完成Application后，再启动SystemUIService，其onCreate会被调用

1. **public** **void** onCreate() {
2. **super**.onCreate();
3. ((SystemUIApplication) getApplication()).startServicesIfNeeded();
4. }

除了onCreate()方法之外，SystemUIService没有其他有意义的代码了。显而易见，SystemUIService是一个容器。在调用了startServicesIfNeeded，进而触发子服务的start()方法之后，SystemUIService便不再做任何其他的事情，任由各个子服务自行运行。而状态栏导航栏则是这些子服务中的一个。

#### SystemUIApplication.startServicesIfNeeded：

接着SystemUIApplication.startServicesIfNeeded判断是否需要启动，先判断sys.boot\_completed属性值

**[java]** [view plain](http://blog.csdn.net/kehyuanyu/article/details/46710591) [copy](http://blog.csdn.net/kehyuanyu/article/details/46710591)

1. **public** **void** startServicesIfNeeded() {
2. **if** (mServicesStarted) {
3. **return**;
4. }
5. **if** (!mBootCompleted) {
6. // check to see if maybe it was already completed long before we began
7. // see ActivityManagerService.finishBooting()
8. **if** ("1".equals(SystemProperties.get("sys.boot\_completed"))) {
9. mBootCompleted = **true**;
10. **if** (DEBUG) Log.v(TAG, "BOOT\_COMPLETED was already sent");
11. }
12. }
13. **for** (**int** i=0; i<N; i++) {
14. Class<?> cl = SERVICES[i]; // 所有子服务
15. **if** (DEBUG) Log.d(TAG, "loading: " + cl);
16. **try** {
17. mServices[i] = (SystemUI)cl.newInstance();
18. }
20. mServices[i].mContext = **this**;
21. mServices[i].mComponents = mComponents;
22. mServices[i].start(); // 启动子服务
23. **if** (mBootCompleted) {
24. mServices[i].onBootCompleted(); //
25. }
26. }
27. mServicesStarted = **true**;
28. }

sys.boot\_completed属性值，在系统的boot启动完成时，ActivityManagerService中会进行设置。

下面看一下所有的子服务，这里启动了各种Service，但是**它们并不是真正的Service ，是继承了SystemUI.java这个抽象类**，复写了start()方法。

1. **private** **final** Class<?>[] SERVICES = **new** Class[] {
2. com.android.systemui.keyguard.KeyguardViewMediator.**class**,
3. com.android.systemui.recent.Recents.**class**,
4. com.android.systemui.volume.VolumeUI.**class**,
5. com.android.systemui.statusbar.SystemBars.**class**,
6. com.android.systemui.usb.StorageNotification.**class**,
7. com.android.systemui.power.PowerUI.**class**,
8. com.android.systemui.media.RingtonePlayer.**class**
9. };

### SERVICES 设计思想

mAm，mIAm；mPm，mIPm区别在哪里？

#### SystemUI

是一个抽象类，定义了模板和相关抽象方法：成员变量mComponents

生命周期监听方法onConfigurationChanged和onBootCompleted，组件map表mComponents

管理

#### SystemUIApplication.mComponents

在SystemUIApplication new 出这个map，在SystemUIApplication.startServicesIfNeeded：

* + 该map地址赋给了mServices[i]对象的mComponents
  + mServices[i]在start方法调用putComponent(Recents.class, this);

通过上述方法，SystemUIApplication.mComponents == mServices[i].mComponents

Q为何putComponent是放在mServices[i]在start调用，而不是直接在SystemUIApplication，

A：mComponents可能不同，比如VolumeUI里面的，实现服务和组件分离



取出单例的办法：

((RecorderUI)((SystemUIApplication)getApplicationContext()).getComponent(RecorderUI.**class**)).setView(**mRecorderView**);

#### RecentsDebugFlags

集中管理所有调试开关

定义黑名单

*<!-- Fully qualified activity class names to be blacklisted in Recents, add package names into overlay as needed -->*<**string-array name="recents\_blacklist\_array"**>  
</**string-array**>

#### SystemServicesProxy

## SystemBars服务启动分析

讨论状态栏与导航栏的启动过程其实就是SystemUIService的启动过程。

相关代码位置

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **# 应用接口相关代码**
2. frameworks/base/core/java/android/app/StatusBarManager.java
3. **# 系统内部接口相关代码**
4. frameworks/base/core/java/com/android/internal/statusbar/IStatusBarService.aidl
5. frameworks/base/core/java/com/android/internal/statusbar/IStatusBar.aidl
6. frameworks/base/core/java/com/android/internal/statusbar/StatusBarIcon.java
7. **# System Server 相关代码**
8. frameworks/base/services/java/com/android/server/StatusBarManagerService.java
9. **# SystemUI 相关代码**
10. frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/StatusBarIconView.java
11. frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/BaseStatusBar.java
12. frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/CommandQueue.java
13. frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/phone/PhoneStatusBar.java

SystemUI 中状态栏的设计架构：就是说 IStatusBar.aidl 定义一系列接口给 SS 用，然后 BaseStatusBar 完成一些状态栏公用的工作（例如生成 notification row 的模板），其他的接口交由子类实现。这里我们以 PhoneStatusBar 来说明一下 addNotification 的流程。

### SystemBars.start方法

以**SystemBars**为例，start()方法又启动了**ServiceMonitor**的start()方法；在此方法里面，通过Handler发mHandler.sendEmptyMessage消息，分别调用了 StartService() 和 continueStartService()两个方法

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **class** SystemBars **extends** SystemUI **implements** ServiceMonitor.Callbacks {
2. **public** **void** start() {
3. //  ServiceMonitor是服务监听者
4. mServiceMonitor = **new** ServiceMonitor(TAG, DEBUG, mContext, Settings.Secure.BAR\_SERVICE\_COMPONENT, **this**);
5. mServiceMonitor.start();  // will call onNoService if no remote service is found
6. }
7. }
8. **public** **class** ServiceMonitor {
9. **public** **void** start() {
10. mHandler.sendEmptyMessage(MSG\_START\_SERVICE);
11. }
12. **private** **final** Handler mHandler = **new** Handler() {
13. **public** **void** handleMessage(Message msg) {
14. **switch**(msg.what) {
15. **case** MSG\_START\_SERVICE:
16. //启动服务
17. startService();
18. **break**;
19. **case** MSG\_CONTINUE\_START\_SERVICE:
20. continueStartService();
21. **break**;
22. **case** MSG\_STOP\_SERVICE:
23. stopService();
24. **break**;
25. }
26. }
27. };
28. }
29. //这里对mServiceName是否为空进行判断，总之无论如何它最终都会启动这个服务
30. **private** **void** startService() {
31. //获取服务组件名称
32. mServiceName = getComponentNameFromSetting();
33. //如果为空，回调服务的onNoService方法
34. **if** (mServiceName == **null**) {
35. mBound = **false**;
36. mCallbacks.onNoService();//回调SystemBars的onNoService里创建StatusBar
37. } **else** {
38. //不为空,回调服务的的onServiceStartAttempt方法
39. **long** delay = mCallbacks.onServiceStartAttempt();
40. mHandler.sendEmptyMessageDelayed(MSG\_CONTINUE\_START\_SERVICE, delay);
41. }
42. }
43. }

### 创建StatusBar

调用：mCallbacks.onNoService()；回到SystemBars 的对象中；这里调用关键方法：createStatusBarFromConfig()；这里面的取值：R.string.config\_statusBarComponent实际就是PhoneStatusBar；

配置文件：base/packages/SystemUI/res/values/config.xml；这样就调到了**PhoneStatusBar**的start()方法。同时PhoneStatusBar在start()方法里面也调用了父类**BaseStatusBar**的start()；

所以这里的mStatusBar是PhoneStatusBar实例,启动了PhoneStatusBar。

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **class** SystemBars **extends** SystemUI **implements** ServiceMonitor.Callbacks {
2. @Override
3. **public** **void** onNoService() {
4. //创建StatusBar
5. createStatusBarFromConfig();  // fallback to using an in-process implementation
6. }
8. **private** **void** createStatusBarFromConfig() {
9. //config\_statusBarComponent就是PhoneStatusBar
10. **final** String clsName = mContext.getString(R.string.config\_statusBarComponent);
11. Class<?> cls = mContext.getClassLoader().loadClass(clsName);
12. mStatusBar = (BaseStatusBar) cls.newInstance();//创建BaseStatusBar实例
13. mStatusBar.mContext = mContext;
14. mStatusBar.mComponents = mComponents;
15. //启动
16. mStatusBar.start();
17. }
18. }

### PhoneStatusBar.start方法

从这里一些列，将NavigationBar和QuickSettingPanel初始化好，添加到UI中。

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **class** PhoneStatusBar **extends** BaseStatusBar **implements** DemoMode, DragDownHelper.DragDownCallback, ActivityStarter, OnUnlockMethodChangedListener,HeadsUpManager.OnHeadsUpChangedListener {
2. **public** **void** start() {
3. //调用父类的start方法,在父类BaseStatusBar里调用createAndAddWindows方法
4. // 3.1
5. **super**.start(); // calls createAndAddWindows()
6. //添加导航栏
7. // 3.2
8. **addNavigationBar();**
9. **//** PhoneStatusBarPolicy定义了系统通知图标的设置策略
10. **mIconPolicy = new PhoneStatusBarPolicy(mContext, mIconController, mCastController,**
11. **mHotspotController, mUserInfoController, mBluetoothController,**
12. **mRotationLockController, mNetworkController.getDataSaverController());**
13. **mIconPolicy.setCurrentUserSetup(mUserSetup);**
14. }
15. }

启动时序图：



### BaseStatusBar.start方法

BaseStatusBar进行一些设置,获取了IStatusBarService实例并注册一些信息到IStatusBarService中,IStatusBarService是一个系统服务,BaseStatusBar将自己注册到IStatusBarService之中,IStatusBarService会把操作状态栏和导航栏的请求转发给BaseStatusBar

为了保证SystemUI意外退出后不会发生信息丢失，IStatusBarService保存了所有需要状态栏与导航栏进行显示或处理的信息副本。在注册时将一个继承自IStatusBar.Stub的CommandQueue的实例注册到IStatusBarService以建立通信，并将信息副本取回。

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **abstract** **class** BaseStatusBar **extends** SystemUI **implements**
2. CommandQueue.Callbacks, ActivatableNotificationView.OnActivatedListener,  ExpandableNotificationRow.ExpansionLogger, NotificationData.Environment,     ExpandableNotificationRow.OnExpandClickListener,
3. OnGutsClosedListener {
4. **public** **void** start() {
5. /\* 由于状态栏的窗口不属于任何一个Activity，所以需要使用第6章所介绍的WindowManager 进行窗口的创建 \*/
6. mDisplay = ((WindowManager)mContext.getSystemService(Context.WINDOW\_SERVICE))
7. .getDefaultDisplay();
8. /\*mProvisioningOberver是一个ContentObserver。
9. 它负责监听Settings.Global.DEVICE\_PROVISIONED设置的变化 \*/
10. mProvisioningObserver.onChange(**false**); // set up
11. mContext.getContentResolver().registerContentObserver(
12. Settings.Global.getUriFor(Settings.Global.DEVICE\_PROVISIONED), **true**,
13. mProvisioningObserver);
14. /\* // 向 SM 获取 SBMS：实例化IStatusBarService,    随后BaseStatusBar将自己注册到IStatusBarService之中。以此声明本实例才是状态栏的真正实现者，IStatusBarService会将其所接受到的请求转发给本实例。 IStatusBarService会保存SystemUi的状态信息,避免SystemUi崩溃而造成信息的丢失
15. \*/
16. mBarService = IStatusBarService.Stub.asInterface(
17. ServiceManager.getService(Context.STATUS\_BAR\_SERVICE));
18. //IStatusBarService与BaseStatusBar进行通信的桥梁。
19. mCommandQueue = **new** CommandQueue(**this**);
21. /\*switches则存储了一些杂项：禁用功能列表，SystemUIVisiblity，是否在导航栏中显示虚拟的菜单键，输入法窗口是否可见、输入法窗口是否消费BACK键、是否接入了实体键盘、实体键盘是否被启用。
22. \*/
23. **int**[] switches = **new** **int**[9];
24. ArrayList<IBinder> binders = **new** ArrayList<IBinder>();
25. /\*它保存了用于显示在状态栏的系统状态区中的状态图标列表。在完成注册之后，IStatusBarService将会在其中填充两个数组，一个字符串 数组用于表示状态的名称，一个StatusBarIcon类型的数组用于存储需要显示的图标资源。 \*/
26. ArrayList<String> iconSlots = **new** ArrayList<>();
27. ArrayList<StatusBarIcon> icons = **new** ArrayList<>();
29. Rect fullscreenStackBounds = **new** Rect();
30. Rect dockedStackBounds = **new** Rect();
32. //调用 SBMS 的注册接口注册：IStatusBarService注册一些信息
33. mBarService.registerStatusBar(mCommandQueue, iconSlots, icons, switches, binders,  fullscreenStackBounds, dockedStackBounds);
34. //创建状态栏窗口
35. createAndAddWindows();
36. }
37. }

/\* 由于状态栏的窗口不属于任何一个Activity，所以需要使用WindowManager进行窗口的创建 \*/

获取IStatusBarService的实例。\*\*IStatusBarService是一个系统服务，由ServerThread

启动并常驻system\_server进程中。IStatusBarService为那些对状态栏感兴趣的其他系统服务定义了一系列API，然而对SystemUI而言，它更像是一个客户端。因为IStatusBarService会将操作状态栏的请求发送给SystemUI，并由后者完成请求

随后BaseStatusBar将自己注册到IStatusBarService之中。以此声明本实例才是状态栏的真正实现者，IStatusBarService会将其所接受到的请求转发给本实例。

“天有不测风云”，SystemUI难免会因为某些原因使得其意外终止。而状态栏中所显示的信息并不属于状态栏自己，而是属于其他的应用程序或是其他的系统服务。因此当SystemUI重新启动时，便需要恢复其终止前所显示的信息以避免信息的丢失。为此，IStatusBarService中保存了所有的需要状态栏进行显示的信息的副本，并在新的状态栏实例启动后，这些副本将会伴随着注册的过程传递给状态栏并进行显示，从而避免了信息的丢失。 从代码分析的角度来看，这一从IstatusBarService中取回信息副本的过程正好完整地体现了状态栏 所能显示的信息的类型

iconList是向IStatusBarService进行注册的参数之一。它保存了用于显示在状态栏的系统状态区中的状态图标列表。在完成注册之后，IStatusBarService将会在其中填充两个数组，一个字符串数组用于表示状态的名称，一个StatusBarIcon类型的数组用于存储需要显示的图标资源。

mCommandQueue是CommandQueue类的一个实例。CommandQueue继承自IStatusBar.Stub。因此它是IStatusBar的Bn端。在完成注册后，这一Binder对象的Bp端将会保存IStatusBarService之中。因此它是IStatusBarService与BaseStatusBar进行通信的桥梁。

public class CommandQueue extends IStatusBar.Stub {

IStatusBarService的真身是StatusBarManagerService

可见，状态栏与导航栏的启动分为如下几个过程：

* 获取IStatusBarService，IStatusBarService是运行于system\_server的一个系统服务，它接受操作状态栏/导航栏的请求并将其转发给BaseStatusBar。为了保证SystemUI意外退出后不会发生信息丢失，IStatusBarService保存了所有需要状态栏与导航栏进行显示或处理的信息副本。
* 将一个继承自IStatusBar.Stub的CommandQueue的实例注册到IStatusBarService以建立通信，并将信息副本取回。
* 通过调用子类的createAndAddWindows()方法完成状态栏与导航栏的控件树及窗口的创建与显示。
* 使用从IStatusBarService取回的信息副本。

回到PhoneStatusBar中, 父类BaseStatusBar中的createAndAddWindows为抽象方法,由子类实现,看下PhoneStatusBar的 createAndAddWindows

### PhoneStatusBar. createAndAddWindows

makeStatusBarView的方法里调用 inflateStatusBarWindow(context)加载布局

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. @Override
2. **public** **void** createAndAddWindows() {
3. //添加状态栏窗口
4. addStatusBarWindow();
5. }
6. **private** **void** addStatusBarWindow() {
7. //创建控件
8. makeStatusBarView();
9. //创建StatusBarWindowManager实例
10. mStatusBarWindowManager = **new** StatusBarWindowManager(mContext);
11. //创建远程输入控制实例
12. mRemoteInputController = **new** RemoteInputController(mStatusBarWindowManager,
13. mHeadsUpManager);
14. //添加状态栏窗口
15. mStatusBarWindowManager.add(mStatusBarWindow, getStatusBarHeight());

/ ② 为状态栏创建WindowManager.LayoutParams

finalWindowManager.LayoutParams lp = new WindowManager.LayoutParams(

ViewGroup.LayoutParams.MATCH\_PARENT, // 状态栏的宽度为充满整个屏幕宽度

height, // 高度来自于getStatusBarHeight()方法

WindowManager.LayoutParams.TYPE\_STATUS\_BAR, // 窗口类型

WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE // 状态栏不接受按键事件

/\* FLAG\_TOUCHABLE\_WHEN\_WAKING这一标记将使得状态栏接受导致设备唤醒的触摸

事件。通常这一事件会在interceptMotionBeforeQueueing()的过程中被用于

唤醒设备（或从变暗状态下恢复），而InputDispatcher会阻止这一事件发送给

窗口。\*/

| WindowManager.LayoutParams.FLAG\_TOUCHABLE\_WHEN\_WAKING

// FLAG\_SPLIT\_TOUCH允许状态栏支持触摸事件序列的拆分

| WindowManager.LayoutParams.FLAG\_SPLIT\_TOUCH,

PixelFormat.TRANSLUCENT); // 状态栏的Surface像素格式为支持透明度

// 启用硬件加速

lp.flags|= WindowManager.LayoutParams.FLAG\_HARDWARE\_ACCELERATED;

//StatusBar的gravity是LEFT和FILL\_HORIZONTAL

lp.gravity = getStatusBarGravity();

lp.setTitle("StatusBar");

lp.packageName = mContext.getPackageName();

1. }
2. **protected** **void** inflateStatusBarWindow(Context context) {
3. mStatusBarWindow = (StatusBarWindowView) View.inflate(context, R.layout.super\_status\_bar, **null**);
4. }

此方法提供了很多重要的信息。

首先是状态栏的高度，由getStatusBarHeight()从资源com.android.internal.R.dimen.status\_bar\_height中获得。这一资源定义在frameworks\base\core\res\res\values\dimens.xml中，默认为25dip。此资源同样在PhoneWindowManager中被用来计算作为布局准绳的八个矩形。

然后是状态栏窗口的LayoutParams的创建。LayoutParams描述了状态栏是怎样的一个窗口。TYPE\_STATUS\_BAR使得PhoneWindowManager为状态栏的窗口分配了较大的layer值，使其可以显示在其他应用窗口之上。FLAG\_NOT\_FOCUSABLE、FLAG\_TOUCHABLE\_WHEN\_WAKING和FLAG\_SPLIT\_TOUCH则定义了状态栏对输入事件的响应行为。

注意 通过创建窗口所使用的LayoutParams来推断一个窗口的行为十分重要。在分析一个需要创建窗口的模块的工作原理时，从窗口创建过程往往是一个不错的切入点。

另外需要知道的是，窗口创建之后，其LayoutParams是会发生变化的。以状态栏为例，创建窗口时其高度为25dip，flags描述其不可接收按键事件。不过当用户按下状态栏导致卷帘下拉时，PhoneStatusBar会通过WindowManager.updateViewLayout()方法修改窗口的LayoutParams的高度为MATCH\_PARENT，即充满整个屏幕以使得卷帘可以满屏显示，并且移除FLAG\_NOT\_FOCUSABLE，使得PhoneStatusBar可以通过监听BACK键以收回卷帘。

在makeStatusBarView()完成控件树的创建之后，WindowManager.addView()将根据控件树创建出状态栏的窗口。显而易见，状态栏控件树的根控件被保存在mStatusBarWindow成员中。

createStatusBarView()负责从R.layout.super\_status\_bar所描述的布局中实例化出一棵控件树。并从这个控件树中取出一些比较重要的控件并保存在对应的成员变量中。因此从R.layout.super\_status\_bar入手可以很容易地得知状态栏的控件树的结构：

### 网络信号栏显示刷新

PhoneStatusBar.java的makeStatusBarView()方法开始，https://www.2cto.com/kf/201702/603116\_2.html

### 导航栏添加

#### PhoneStatusBar.addNavigationBar

这个方法是添加底部的导航栏的,就是那些home键,back键所在的区域.

PhoneStatusBar类的start()方法里面调用 addNavigationBar() ; 流程如上图，第 9 步；然后此方法内调用 prepareNavigationBarView()；最后 WindowManager调addView将NavigationBarView添加到 UI 界面。

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **class** PhoneStatusBar **extends** BaseStatusBar  {
2. **protected** **void** addNavigationBar() {
3. **if** (mNavigationBarView == **null**) **return**;
4. //初始化导航栏
5. prepareNavigationBarView();
6. //添加到WindowManager中
7. mWindowManager.addView(mNavigationBarView, getNavigationBarLayoutParams());
8. }
10. **private** **void** prepareNavigationBarView() {
11. //重新初始化
12. mNavigationBarView.reorient();
13. //最近应用键
14. ButtonDispatcher recentsButton = mNavigationBarView.getRecentsButton();
15. recentsButton.setOnClickListener(mRecentsClickListener);
16. recentsButton.setOnTouchListener(mRecentsPreloadOnTouchListener);
17. recentsButton.setLongClickable(**true**);
18. recentsButton.setOnLongClickListener(mRecentsLongClickListener);
20. //后退键
21. ButtonDispatcher backButton = mNavigationBarView.getBackButton();
22. backButton.setLongClickable(**true**);
23. backButton.setOnLongClickListener(mLongPressBackListener);
25. //home键
26. ButtonDispatcher homeButton = mNavigationBarView.getHomeButton();
27. homeButton.setOnTouchListener(mHomeActionListener);
28. homeButton.setOnLongClickListener(mLongPressHomeListener);
30. //监听配置改变
31. mAssistManager.onConfigurationChanged();
32. }
33. }

NavigationBar主要包含三个按钮：Back，Home和Recent。

**[java]** [view plain](http://blog.csdn.net/a2758963/article/details/42675773) [copy](http://blog.csdn.net/a2758963/article/details/42675773)

1. **try** {
2. **boolean** showNav = mWindowManagerService.hasNavigationBar();
3. **if** (DEBUG) Slog.v(TAG, "hasNavigationBar=" + showNav);
4. **if** (showNav) {
5. mNavigationBarView =
6. (NavigationBarView) View.inflate(context, R.layout.navigation\_bar, **null**);
8. mNavigationBarView.setDisabledFlags(mDisabled);
9. mNavigationBarView.setBar(**this**);
10. }
11. } **catch** (RemoteException ex) {
12. // no window manager? good luck with that
13. }</span>

WindowManagerService类实现了WindowManagerPolicy的接口，所以WindowManagerService会回调WindowManagerPolicy 的hasNavigationBar()接口

**[java]** [view plain](http://blog.csdn.net/a2758963/article/details/42675773) [copy](http://blog.csdn.net/a2758963/article/details/42675773)

1. @Override
2. **public** **boolean** hasNavigationBar() {
3. **return** mPolicy.hasNavigationBar();
4. }

Policy向下调用实际上调用的是PhoneWindowManager实现的hasNavigationBar方法，下面代码是PhoneWindowManager中的hasNavigationBar()方法。

**[java]** [view plain](http://blog.csdn.net/a2758963/article/details/42675773) [copy](http://blog.csdn.net/a2758963/article/details/42675773)

1. // Use this instead of checking config\_showNavigationBar so that it can be consistently
2. // overridden by qemu.hw.mainkeys in the emulator.
3. **public** **boolean** hasNavigationBar() {
4. **return** mHasNavigationBar;
5. }

而mHasNavigationBar的赋值可以在PhoneWindowManager中的setInitialDisplaySize(Display display, int width, int height, int density)方法中找到，

**[java]** [view plain](http://blog.csdn.net/a2758963/article/details/42675773) [copy](http://blog.csdn.net/a2758963/article/details/42675773)

1. **if** (!mHasSystemNavBar) {
2. mHasNavigationBar = mContext.getResources().getBoolean(
3. com.android.internal.R.bool.config\_showNavigationBar);
4. // Allow a system property to override this. Used by the emulator.
5. // See also hasNavigationBar().
6. String navBarOverride = SystemProperties.get("qemu.hw.mainkeys");
7. **if** (! "".equals(navBarOverride)) {
8. **if**      (navBarOverride.equals("1")) mHasNavigationBar = **false**;
9. **else** **if** (navBarOverride.equals("0")) mHasNavigationBar = **true**;
10. }
11. } **else** {
12. mHasNavigationBar = **false**;
13. }

#### NavigationBar 的显示与隐藏

从上面代码可以看到mHasNavigationBar的值的设定是由两处决定的：

1.首先从系统的资源文件中取设定值config\_showNavigationBar, 这个值的设定的文件路径是frameworks/base/core/res/res/values/config.xml

1. <!-- Whether a software navigation bar should be shown. NOTE: in the future **this** may be
2. autodetected from the Configuration. -->
3. <bool name="config\_showNavigationBar">**false**</bool>

2.然后系统要获取“qemu.hw.mainkeys”的值，这个值可能会覆盖上面获取到的mHasNavigationBar的值。如果“qemu.hw.mainkeys”获取的值不为空的话，不管值是true还是false,都要依据后面的情况来设定。

所以上面的两处设定共同决定了NavigationBar的显示与隐藏。

值得注意的是，onCreate()方法根据IWindowManager.hasSystemNavBar()方法的返回值为状态栏/导航栏选择了不同的实现。进行这一选择的原因为了能够在大尺寸的设备中更有效地利用屏幕空间。在小屏幕设备如手机中，由于屏幕宽度有限，Android采取了状态栏与导航栏分离的布局方案，也就是说导航栏与状态栏占用了更多的垂直空间，使得导航栏的虚拟按键尺寸足够大以及状态栏的信息量足够多。而在大屏幕设备如平板电脑中，由于屏幕宽度比较大，足以在一个屏幕宽度中同时显示足够大的虚拟按键以及足够多的状态栏信息量，此时可以选择将状态栏与导航栏功能集成在一起成为系统栏作为大屏幕下的布局方案，以节省对垂直空间的占用。

#### N~~avigationBar 的显示与隐藏XXXXX~~

public void setDisabledFlags(int disabledFlags, boolean force) {

if (!force && mDisabledFlags == disabledFlags) return;

mDisabledFlags = disabledFlags;

final boolean disableHome = ((disabledFlags & View.STATUS\_BAR\_DISABLE\_HOME) != 0);

final boolean disableRecent = ((disabledFlags & View.STATUS\_BAR\_DISABLE\_RECENT) != 0);

final boolean disableBack = ((disabledFlags & View.STATUS\_BAR\_DISABLE\_BACK) != 0)

&& ((mNavigationIconHints & StatusBarManager.NAVIGATION\_HINT\_BACK\_ALT) == 0);

### 其他

#### 理解IStatusBarService

<https://wizardforcel.gitbooks.io/deepin-android-vol3/content/7.html>

## SystemUI主要视图

StatusBarWindowView

SystemUI的根视图是在PhoneStatusBar的makeStatusBarView()方法中加载的。

protected PhoneStatusBarView makeStatusBarView() {

mStatusBarWindow = (StatusBarWindowView) View.inflate(context,

R.layout.super\_status\_bar, null);

}

通过对super\_status\_bar.xml文件的认识，就能了解SystemUI的大体视图构成

主布局是R.layout.super\_status\_bar,对应的是StatusBarWindowView这个自定义布局.

StatusBarWindowView里有几个主要的布局:

* layout/status\_bar
* layout/brightness\_mirror
* layout/status\_bar\_expanded

如下图



**[html]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **<com.android.systemui.statusbar.phone.StatusBarWindowView**
2. xmlns:android="http://schemas.android.com/apk/res/android"
3. xmlns:sysui="http://schemas.android.com/apk/res-auto"
4. android:layout\_width="match\_parent"
5. android:layout\_height="match\_parent"
6. android:fitsSystemWindows="true"**>**
8. ......
10. <!--正常状态栏下的布局 -->
11. **<include** layout="@layout/status\_bar"
12. android:layout\_width="match\_parent"
13. android:layout\_height="@dimen/status\_bar\_height" **/>**
15. <!--状态栏图标下的SeekBar布局 -->
16. **<include** layout="@layout/brightness\_mirror" **/>**
18. <!--车载模式的布局  -->
19. **<ViewStub** android:id="@+id/fullscreen\_user\_switcher\_stub"
20. android:layout="@layout/car\_fullscreen\_user\_switcher"
21. android:layout\_width="match\_parent"
22. android:layout\_height="match\_parent"**/>**
24. <!--状态栏下拉的布局  -->
25. **<include** layout="@layout/status\_bar\_expanded"
26. android:layout\_width="match\_parent"
27. android:layout\_height="match\_parent"
28. android:visibility="gone" **/>**
30. **</com.android.systemui.statusbar.phone.StatusBarWindowView>**

### layout/status\_bar：PhoneStatusBarView



这个是正常状态下(未下拉的状态栏图标区域),这个布局对应的是PhoneStatusBarView

**[html]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **<com.android.systemui.statusbar.phone.PhoneStatusBarView**
2. xmlns:android="http://schemas.android.com/apk/res/android"
3. xmlns:systemui="http://schemas.android.com/apk/res/com.android.systemui"
4. android:id="@+id/status\_bar"
5. android:background="@drawable/system\_bar\_background"
6. android:orientation="vertical"
7. android:focusable="false"
8. android:descendantFocusability="afterDescendants"
9. **>**
10. ......
11. <!--状态栏  -->
12. **<LinearLayout** android:id="@+id/status\_bar\_contents"
13. android:layout\_width="match\_parent"
14. android:layout\_height="match\_parent"
15. android:paddingStart="6dp"
16. android:paddingEnd="8dp"
17. android:orientation="horizontal"
18. **>**
20. <!-- The alpha of this area is controlled from both PhoneStatusBarTransitions and
21. PhoneStatusBar (DISABLE\_NOTIFICATION\_ICONS). --**>**
22. <!-- 通知图标区域-->
23. **<com.android.systemui.statusbar.AlphaOptimizedFrameLayout**
24. android:id="@+id/notification\_icon\_area"
25. android:layout\_width="0dip"
26. android:layout\_height="match\_parent"
27. android:layout\_weight="1"
28. android:orientation="horizontal" **/>**
30. <!-- 系统图标区域 -->
31. **<com.android.keyguard.AlphaOptimizedLinearLayout** android:id="@+id/system\_icon\_area"
32. android:layout\_width="wrap\_content"
33. android:layout\_height="match\_parent"
34. android:orientation="horizontal"
35. **>**
37. <!-- 系统图标 -->
38. **<include** layout="@layout/system\_icons" **/>**
40. <!-- 时钟信息 -->
41. **<com.android.systemui.statusbar.policy.Clock**
42. android:id="@+id/clock"
43. android:textAppearance="@style/TextAppearance.StatusBar.Clock"
44. android:layout\_width="wrap\_content"
45. android:layout\_height="match\_parent"
46. android:singleLine="true"
47. android:paddingStart="@dimen/status\_bar\_clock\_starting\_padding"
48. android:paddingEnd="@dimen/status\_bar\_clock\_end\_padding"
49. android:gravity="center\_vertical|start"
50. **/>**
51. **</com.android.keyguard.AlphaOptimizedLinearLayout>**
52. **</LinearLayout>**
54. **</com.android.systemui.statusbar.phone.PhoneStatusBarView>**

### layout/brightness\_mirror

这个布局就是中间那个调整亮度的seekBar.没啥好介绍的.



### layout/status\_bar\_expanded: NotificationPanelView

这个布局是下拉时的状态栏的布局





**[html]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **<com.android.systemui.statusbar.phone.NotificationPanelView**
2. xmlns:android="http://schemas.android.com/apk/res/android"
3. xmlns:systemui="http://schemas.android.com/apk/res/com.android.systemui"
4. android:id="@+id/notification\_panel"
5. android:layout\_width="match\_parent"
6. android:layout\_height="match\_parent"
7. android:background="@android:color/transparent"
8. **>**
9. <!--锁屏时的时钟布局  -->
10. **<include**
11. layout="@layout/keyguard\_status\_view"
12. android:layout\_height="wrap\_content"
13. android:visibility="gone" **/>**
15. **<com.android.systemui.statusbar.phone.NotificationsQuickSettingsContainer**
16. android:layout\_width="match\_parent"
17. android:layout\_height="match\_parent"
18. android:layout\_gravity="@integer/notification\_panel\_layout\_gravity"
19. android:id="@+id/notification\_container\_parent"
20. android:clipToPadding="false"
21. android:clipChildren="false"**>**
23. <!--quciksetting区域  -->
24. **<com.android.systemui.AutoReinflateContainer**
25. android:id="@+id/qs\_auto\_reinflate\_container"
26. android:layout="@layout/qs\_panel"
27. android:layout\_width="@dimen/notification\_panel\_width"
28. android:layout\_height="match\_parent"
29. android:layout\_gravity="@integer/notification\_panel\_layout\_gravity"
30. android:clipToPadding="false"
31. android:clipChildren="false" **/>**
33. <!-- 通知栏区域 -->
34. **<com.android.systemui.statusbar.stack.NotificationStackScrollLayout**
35. android:id="@+id/notification\_stack\_scroller"
36. android:layout\_width="@dimen/notification\_panel\_width"
37. android:layout\_height="match\_parent"
38. android:layout\_gravity="@integer/notification\_panel\_layout\_gravity"
39. android:layout\_marginBottom="@dimen/close\_handle\_underlap" **/>**
41. <!--锁屏切换 -->
42. **<ViewStub**
43. android:id="@+id/keyguard\_user\_switcher"
44. android:layout="@layout/keyguard\_user\_switcher"
45. android:layout\_height="match\_parent"
46. android:layout\_width="match\_parent" **/>**
48. <!--锁屏下的状态栏 -->
49. **<include**
50. layout="@layout/keyguard\_status\_bar"
51. android:visibility="invisible" **/>**
53. **</com.android.systemui.statusbar.phone.NotificationsQuickSettingsContainer>**
55. <!--锁屏界面底部的图标 -->
56. **<include**
57. layout="@layout/keyguard\_bottom\_area"
58. android:visibility="gone" **/>**

61. **</com.android.systemui.statusbar.phone.NotificationPanelView>**<!-- end of sliding panel -->

细节图

创建完布局后，就会添加窗口到WindowManager里,这样状态栏就创建完成了.接下来会回到 PhoneStatusBar的start方法调用addNavigationBar()处

## SystemUI小结

Sd

完成了对SystemUI的启动过程的分析之后便可以对其体系结构做出总结，如图7-1所示。

· SystemUIService，一个普通的Android服务，它以一个容器的角色运行于SystemUI进程中。在它内部运行着多个子服务，其中之一便是状态栏与导航栏的实现者——BaseStatusBar的子类之一。

· IStatusBarService，即系统服务StatusBarManagerService是状态栏导航栏向外界提供服务的前端接口，运行于system\_server进程中。

· BaseStatusBar及其子类是状态栏与导航栏的实际实现者，运行于SystemUIService中。

· IStatusBar，即SystemUI中的CommandQueue是联系StatusBarManagerService与BaseStatusBar的桥梁。

· SystemUI中还包含了ImageWallpaper、RecentPanel以及TakeScreenshotService等功能的实现。它们是Service、Activity等标准的Android应用程序组件，并且互相独立。对这些功能感兴趣的使用者可以通过startService()/startActivity()等方式方便地启动相应的功能。



图 7 - 1 SystemUI的体系结构

在本章将主要介绍SystemUI中最常用的状态栏、导航栏以及RecentPanel的实现。ImageWallpaper将在第8章中进行详细地介绍。而SystemUI其他的功能读者可以自行研究。

# QS调起逻辑

在phoneManager管理

Ui显示

*//http://www.voidcn.com/article/p-aldyshja-sy.html  
 mStatusBarManager = (StatusBarManager)getSystemService(Context.STATUS\_BAR\_SERVICE);  
 mStatusBarManager.disable(StatusBarManager.DISABLE\_EXPAND);*

packagecom.android.systemui.statusbar.phone;

public class PhoneStatusBarView extends PanelBar {

@Override

public boolean panelsEnabled() { *//可以在此处修改*

return ((mBar.mDisabled & StatusBarManager.DISABLE\_EXPAND) == 0);

}

}

有先时候有需求需要屏蔽状态栏下拉，例如com.android.phone中的紧急呼叫时（sim卡锁住），此时源码设计是状态栏下拉不了的。

## Back键收起逻辑

Sdf

### StatusBarWindowView.dispatchKeyEvent

**case** KeyEvent.KEYCODE\_BACK:  
 **if** (!down) {  
 mService.onBackPressed();  
 }

### PhoneStatusBar. onBackPressed

**public boolean** onBackPressed() {  
 **if** (**mStatusBarKeyguardViewManager**.onBackPressed()) {  
 **return true**;  
 }  
 **if** (**mNotificationPanel**.isQsExpanded()) {  
 **if** (**mNotificationPanel**.isQsDetailShowing()) {  
 **mNotificationPanel**.closeQsDetail();  
 } **else** {  
 **mNotificationPanel**.animateCloseQs();  
 }  
 **return true**;  
 }  
 **if** (**mState** != StatusBarState.***KEYGUARD*** && **mState** != StatusBarState.***SHADE\_LOCKED***) {  
 animateCollapsePanels();  
 **return true**;  
 }  
 **return false**;  
}

# 深入理解PhoneStatusBarView

## Phone状态分类描述

其中,状态图的区域分为以下几种:

*  通知栏图标,在状态栏的最左侧显示通知信息,比如来了一个短信,那么就会弹出一个短信图标
*  时间信息,显示一个时间,比如上午9:58
*  信号图标,显示手机信号,wifi信号等
*  电量图标,显示当前电量状态
*  状态图标,wifi,蓝牙等开关状态

### 通知信息

它可以在状态栏左侧显示一个图标以引起用户的主意，并在下拉卷帘中为用户显示更加详细的信息。这是状态栏所能提供的信息显示服务之中最灵活的一种功能。它对信息种类以及来源没有做任何限制。使用者可以通过StatusBarManagerService所提供的接口向状态栏中添加或移除一条通知信息。

### 时间信息

显示在状态栏最右侧的一个小型数字时钟，是一个名为Clock的继承自TextView的控件。它监听了几个和时间相关的广播：ACTION\_TIME\_TICK、ACTION\_TIME\_CHANGED、ACTION\_TIMEZONE\_CHANGED以及ACTION\_CONFIGURATION\_CHANGED。当其中一个广播到来时从Calendar类中获取当前的系统时间，然后进行字符串格式化后显示出来。时间信息的维护工作在状态栏内部完成，因此外界无法通过API修改时间信息的显示或行为。

### 电量信息

显示在数字时钟左侧的一个电池图标，用于提示设备当前的电量情况。它是一个被BatteryController类所管理的ImageView。BatteryController通过监听android.intent.action.BATTERY\_CHANGED广播以从BetteryService中获取电量信息，并根据电量信息选择一个合适的电池图标显示在ImageView上。同时间信息一样，这也是在状态栏内部维护的，外界无法干预状态栏对电量信息的显示行为。

### 信号信息

显示在电量信息的左侧的一系列ImageView，用于显示系统当前的Wifi、移动网络的信号状态。用户所看到的Wifi图标、手机信号图标、飞行模式图标都属于信号信息的范畴。它们被NetworkController类维护着。NetworkController监听了一系列与信号相关的广播如WIFI\_STATE\_CHANGED\_ACTION、ACTION\_SIM\_STATE\_CHANGED、ACTION\_AIRPLANE\_MODE\_CHANGED等，并在这些广播到来时显示、更改或移除相关的ImageView。同样，外界无法干预状态栏对信号信息的显示行为。

### 系统状态图标区

这个区域用一系列图标标识系统当前的状态，位于信号信息的左侧，与状态栏左侧通知信息隔岸相望。通知信息类似，StatusBarManagerService通过setIcon()接口为外界提供了修改系统状态图标区的图标的途径，而然它对信息的内容有很强的限制。首先，系统状态图标区无法显示图标以外的信息，另外，系统状态图标区的对其所显示的图标数量以及图标所表示的意图有着严格的限制。

## PhoneStatusBarView树形图

见status\_bar

## 本文内容

由于时间信息、电量信息以及信号信息的实现原理比较简单而且与状态栏外界相对隔离，因此读者可以通过分析上文所介绍的相关组件自行研究。本节将主要介绍状态栏的一下几个方面的内容：

·  状态栏窗口的创建与控件树结构。

·  通知的管理与显示。

·  系统状态图标区的管理与显示。

## 状态栏窗口的创建

BaseStatusBar.start()方法的代码中调用了createAndAddWindows()方法进行状态栏窗口的创建。很显然，createAndAddWindow()由PhoneStatusBar实现

<https://blog.csdn.net/innost/article/details/47660591>

## StatusBar加图标AddIcons

### SignalClusterView

Fg

@Override

public void setNoSims(boolean show) {

mNoSimsVisible = show && !mBlockMobile && DjiSystemUIHelper.getIns().isNosimVisible();

apply();

}

这里的流程，需要参考到前面的SystemUI的类图，PhoneStatusBar继承自 BaseStatusBar，而BaseStatusBar实现了CommandQueue里的Callbacks接口；而这个接口里面有个关键的addIcon() 方法。

PhoneStatusBar的start() 方法里面调用PhoneStatusBarPolicy的构造器；

**注意：这里现在所有的Icon初始，都是不可见的。**

此构造器代码片段：



然后，后面依次调用StatusBarManager，StatusBarManagerService和 CommandQueue的setIcon()方法；

再然后CommandQueue对象回调PhoneStatusBar的AddIcon()方法；最后通过mStatusIconsKeyguard.addView()方法，将Icon显示到界面上。

最后，那些图标真的需要显示了，是通过接收系统广播，然后动态显示，如下图的StatusBarManager对象的setIcon()调用关系：



此部分时序图如下：

Start

#### Icons排列规则

状态栏的布局文件：

frameworks/base/packages/SystemUI/res/layout/**status\_bar.xml**

此文件 又依次包含下面两个文件：

frameworks/base/packages/SystemUI/res/layout/**system\_icons.xml**

frameworks/base/packages/SystemUI/res/layout/**signal\_cluster\_view.xml**



总体来说，Icons的排列分四个区域，容器是 **PhoneStatusBarView**；com.android.systemui.statusbar.phone.PhoneStatusBarView；它是继承自 **PanelBar extends FrameLayout** ；

四个区域中，

三个常态从左往右，分别是：**应用通知区**(gone**过量**)**、系统图标、时钟区；**

另外一个动态浮动显示区域，**通知提示区**(ticker\_stub)；

**系统图标**又包含三部分：**状态图标、信号区、电量区**(默认gone**电量百分比**)；

**信号区**域包括：**VPN、WiFi、手机信号、飞行模式、no\_sims；**

以上各区域，关系结构图如下：

通知栏

(浮动)通知提示区(ticker\_stub)

(常态)状态栏

应用通知区

系统图标区

应用通知过量(gone)

时钟

状态

电量百分比(gone)

电量icon

信号区

VPN

手机信号

WiFi

飞行模式

no\_sims

**各图标和区域，宽度与互挤原则如下：**

其中，只有最左边的 **应用通知区** 设置了权重，因而它可以得到更多的区域(在其他区动态不显示的时候)，如下图XML文件。



**其他区域宽度基本都是** (wrap\_content);

**另外有三个固定宽度，和一个占满宽度**(match\_parent) **ticker** ：



电量图标宽度9.5dp，如下图：



wifi\_signal\_spacer和wifi\_airplane\_spacer宽高4dp，可见性gone：





### APP通知到PhoneStatusBar

然后是对Notification的组装，然后对Notification的icon进行判断，若其不为零，便为有效通知，之后调用关键方法**notifyPostedLocked()**：



到这里做了可见性的一些判断，就开始准备回调了，因为前面通知已经组装完毕准备显示到状态栏了，之后就需要将相关的通知消息告诉所有监听者。继续看到**notifyPostedLocked()**中调到的**notifyPosted()**方法：



这里INotificationListener的实现类是**NotificationListenerService**.Java；到这里就调到Listener的**onNotificationPosted(**)方法；此方法是一个**空的抽象方法**；是由它的APP实例类来实现的。源码截图如下：



此方法，在**BaseStatusBar**类中实现，new NotificationListenerService() { ... }；如下图：



时序图如下：



### APP清除(Cancel)通知

与"新增通知"类似的流程是"删除通知"，发起点在NotificationManager，之后经由NotificationManagerService处理和NotificationListenerService传递，最后到达各个继承自NotificationListenerService的子类中，只不过最后的处理方法变成了onNotificationRemoved。调用时序图下：



**NotificationListenerService**调用流程小结

简单来看，NotificationListenerService在系统通知的消息传递过程中，起到了代理的作用。继承自NotificationListenerService的类作为client端，真正的server端则是**NotificationManagerService**，由它负责整个Notification的控制与管理。NotificationManagerService将处理之后的结果通过NotificationListenerService返回给client端，最终各个client端(**BaseStatusBar**)通过**onNotificationPosted()**和**onNotificationRemoved()**方法拿到系统通知状态变更的相关信息。

一. 锁屏界面下拉通知如何实现

1. 线索点

A. 下拉、上拉时会触发NotificationPanelView.setNotifyStackExpansion

### QuickSettingPanel快捷开关

QSTitleHost.Java中定义快捷开关的各个标题createTiles()：代码如下：



此方法在recreateTiles() 中调用，而recreateTiles() 又是在QSTitleHost的构造器中被调用，

QSTitleHost被构造好之后，为QSPanel对象配置QSTitleHost和标题；然后给顶部状态View设置QSPanel；最后为QSTitleHost设置CallBack回调方法。

PhoneStatusBar.Java中，相关代码如下：



QuickSettingPanel时序图：

# 深入理解导航栏

深入理解ANDROID 卷3 PDF 下载

# 如何禁用状态栏和导航栏的功能

深入理解ANDROID 卷3 PDF 下载

# SystemUiVisibiliy理解

尽管StatusBarManager以及StatusBarManagerService为应用程序以及系统服务提供了操作状态栏与导航栏的所有接口，但是这些接口并不适用于那些没有系统签名的普通应用程序。

倘若一个普通的应用程序希望对状态栏以及导航栏进行操作，就必须使用SystemUIVisibility机制、

SystemUIVisibility与禁用标记一样，是一个int型的变量，其中可以按位容纳多个定义在View类中用于调整状态栏与导航栏行为的标记。其实SystemUIVisibility可容纳的标记的丰富性远远不止控制SystemUI的可见性而已。按照这些标记所产生的影响，它们可以分为以下三类。

影响状态栏与导航栏可见性的标记:

0 SYSTEM UI FLAG LOW PROFILE，使得状态栏与导航栏进人低辨识度模式。

口SYSTEM UI FLAG HIDE NAVIGATION,隐藏导航栏。窗口在布局时的ParentFrame

ContentFrame以及DisplayFrame都会延展到整个屏幕。

口SYSTEM UI FLAG FULLSCREEN，**隐藏状态栏**。窗口在布局时的ParentFrame ,

ContentFrame以及DisplayFrame都会延展到屏幕的顶部。并不隐藏导航栏哦！

影响窗口布局结果的标记:

0 SYSTEM UI FLAG LAYOUT STABLE，声明这个标记的窗口的ContentFrame不会 随着导航栏或状态栏的显示或隐藏而发生变化。

0 SYSTEMes UI\_ FLAG\_ LAYOUTes HIDE一AVIGATION, PhoneWindowManager在对声 明了这个标记的窗口进行布局时会认为导航栏已经隐藏，无论导航栏是否真的被 隐藏。

0 SYSTEM\_ UI\_ FLAG\_ LAYOUT\_ FULLSCREEN, PhoneWindowManager在对声明了这个标记的窗口进行布局时会认为导航栏和状态栏都已经隐藏，无论状态栏和导航栏是 否真的被隐藏。

用于禁用状态栏与导航栏的某些功能的标记:

口STATUS BAR DISABLE EXPAND

日STATUS BAR DISABLE NOTIFICATION ICONS

口STATUS BAR DISABLE NOTIFICATION ALERTS

口STATUS BAR DISABLE NOTIFICATION TICKER

口STATUS BAR DISABLE SYSTEM INFO

口STATUS BAR DISABLE HOME

口STATUS BAR DISABLE BACK

口STATUS BAR DISABLE CLOCK

口STATUS BAR DISABLE RECENT

口STATUS BAR DISABLE SEARCH

这些以STATUS\_ BAR DISABLE\_为前缀的标记的功能与在StatusBarManager中所定义的禁用标记一一对应并且功能相同，并且可以通过systemUIVisibility & StatusBar-

Manager.DISABLE\_ MASK这种方式将它们从SystemUIVisibility中提取出来，然后作

为StatusBarManager.disable()方法的参数设置给状态栏与导航栏。不过目前Android

中并没有做出这种行为的代码。因此，通过SystemUIVisibility的方式进行状态栏与导

航栏功能的禁用目前是行不通的。

设置SystemUIVisibility的方式有两种。一是在任意一个已经显示在窗口上的控件调用View.setSystemUiVisibility()，二是直接在窗口的LayoutParams.systemUiVisibility上进行设置并通过WindowManager.updateViewLayout()方法使其生效。

## View.setSystemUiVisibility()过程

从可用的标记列表可以看出SystemUIVisibility主要涉及状态栏和导航栏的行为以及窗口布局两个方面，因此SystemUIVisibility的消费者有SystemUI中的BaseStatusBar，以及负责窗口布局的PhoneWindowManager。本节将从View.setSystemUiVisibility()开始追寻

SystemUIVisibility从控件到BaseStatusBar以及PhoneWindowManager的过程。

### setSystemUiVisibility

public void setSystemUiVisibility(int visibility) {

if (visibility != mSystemUiVisibility) {

mSystemUiVisibility = visibility;// SyetemUIViaibility会保存在View自己的成员变蚤mSyatemUiViaibility中

if (mParent != null && mAttachInfo != null && !mAttachInfo.mRecomputeGlobalAttributes) {//通知父控件这一变化

mParent.recomputeViewAttributes(this);

}

}

}

recomputeViewAttributes()会沿着控件树向根部回溯，最终ViewRootImpl的recompute-

ViewAttributes()会对此做出如下处理:

### ViewRootImpl. recomputeViewAttributes

public void recomputeViewAttributes(View child) {

checkThread();//必须在窗口的主线程中调用

if (mView == child) {

//标记随后需要处理SystemU工Visibility的变化

mAttachInfo.mRecomputeGlobalAttributes = true;

if (!mWillDrawSoon) {

scheduleTraversals();//触发一次“遮历”动作

}

}

}

在ViewRootImpl的“遍历”过程中会执行ViewRootImpl.collectViewAttributes()方法收集

控件树中每个View所保存的SystemUIVisibility。其实现如下:

### ViewRootImpl.collectViewAttributes

private boolean collectViewAttributes() {

if (mAttachInfo.mRecomputeGlobalAttributes) {

//Log.i(mTag, "Computing view hierarchy attributes!");

mAttachInfo.mRecomputeGlobalAttributes = false;

boolean oldScreenOn = mAttachInfo.mKeepScreenOn;

mAttachInfo.mKeepScreenOn = false;

////①清空Attachlnfo中所保存的SyatemUIVisibility

mAttachInfo.mSystemUiVisibility = 0;

mAttachInfo.mHasSystemUiListeners = false;

///\*②View.dispatchCollectViewAttributes()会追历整个控件树。

·并在这个过程中以按位或的方式将每个控件所存储的SystemU工Visibility累加到Attach工nfo.mSystemUiVisibility之上\*/

mView.dispatchCollectViewAttributes(mAttachInfo, 0);

//从收集到的System工Visibility中移除被禁用的System工Visib主工ity标记

mAttachInfo.mSystemUiVisibility &= ~mAttachInfo.mDisabledSystemUiVisibility;

WindowManager.LayoutParams params = mWindowAttributes;

mAttachInfo.mSystemUiVisibility |= getImpliedSystemUiVisibility(params);

if (mAttachInfo.mKeepScreenOn != oldScreenOn

|| mAttachInfo.mSystemUiVisibility != params.subtreeSystemUiVisibility

|| mAttachInfo.mHasSystemUiListeners != params.hasSystemUiListeners) {

applyKeepScreenOnFlag(params);

////**③将SystemU工Visibility保存到窗口的LayoutParams中**

params.subtreeSystemUiVisibility = mAttachInfo.mSystemUiVisibility;

params.hasSystemUiListeners = mAttachInfo.mHasSystemUiListeners;

mView.dispatchWindowSystemUiVisiblityChanged(mAttachInfo.mSystemUiVisibility);

return true;

}

}

return false;

}

可见，在控件树中，每一个控件都在View.mSystemUiVisibility中保存了本控件所期望的SystemUIVisibility。当ViewRootImpl进行“遍历”操作时，会通过View.dispatch-CollectViewAttributes()方法将每个控件所期望的SystemUIVisibility收集到AttachInfo.mSystemUiVisibility成员中，并保存到当前窗口的LayoutParams.sutreeSystemUiVisibility作为本窗口所期望的SystemUIVisibility。当ViewRootImpl通过WMS.relayoutWindow()方法对

窗口进行重新布局时，本窗口所期望的SystemUIVisibility会伴随着LayoutParams.subtree-

SystemUiVisibility以及LayoutParams.systemUiVisibility两个字段进人WMS}

### WMS中的SystemUIVisibility

既然一个窗口所期望的SystemUIVisibility会通过WMS.relayoutWindow()方法进人

WMS.干是可以参考WMS.relavoutWindow(1中与SvstemUIVsibilitv相关的代码:

## setSystemUiVisibili原理

设置状态栏和导航栏都会显示成半透明的状态：

* Xml方式

res/values/styles.xml写<item name="android:windowTranslucentStatus">true</item>

<item name="android: windowTranslucentNavigation">true</item>

* Java方式

setContentView(R.layout.activity\_main);

getWindow().setFlags(WindowManager.LayoutParams.FLAG\_TRANSLUCENT\_STATUS, WindowManager.LayoutParams.FLAG\_TRANSLUCENT\_STATUS);

getWindow().setFlags(WindowManager.LayoutParams.FLAG\_TRANSLUCENT\_NAVIGATION, WindowManager.LayoutParams.FLAG\_TRANSLUCENT\_NAVIGATION)



我们只要理解了windowTranslucentStatus实现流程，自然而然也就知道了windowTranslucentNavigation的实现原理

### PhoneWindow.generateLayout

protected ViewGroup generateLayout(DecorView decor) {

    // Apply data from current theme.

    TypedArray a = getWindowStyle();

    if (a.getBoolean(com.android.internal.R.styleable.Window\_windowTranslucentStatus, false)) {

        setFlags(FLAG\_TRANSLUCENT\_STATUS, FLAG\_TRANSLUCENT\_STATUS

                & (~getForcedWindowFlags()));

    }

}

#### Window. setFlags

public void setFlags(int flags, int mask) {

final WindowManager.LayoutParams attrs = getAttributes();

attrs.flags = (attrs.flags&~mask) | (flags&mask);

mForcedWindowFlags |= mask;

dispatchWindowAttributesChanged(attrs);

}

上面代码可以看出我们在styles.xml中设置的属性，会通过setFlags设置下去。而setFlags其实就是将属性设置给了WindowManager.LayoutParams。也就是说，我们设置的windowTranslucentStatus属性，最终变成WindowManager.LayoutParamsflags属性中FLAG\_TRANSLUCENT\_STATUS.

### PhoneWindowManager.updateSystemUiVisibilityLw

更新状态栏的状态变化。在PhoneWindowManager.java中updateSystemUiVisibilityLw方法是处理系统状态栏变化的地方。在其内部它会调用updateSystemBarsLw方法，然后，根据其返回值设置给SystemUI。而updateSystemBarsLw中mStatusBarController才是真正管理状态栏的实例类。在updateSystemBarsLw中，它会调用mStatusBarController的applyTranslucentFlagLw和updateVisibilityLw的方法。这两个方法就会返回相应的显示状态。SystemUI就是根据这个显示状态值，做相应的变化。

private final BarController mStatusBarController = new BarController("StatusBar",

View.STATUS\_BAR\_TRANSIENT,

View.STATUS\_BAR\_UNHIDE,

View.STATUS\_BAR\_TRANSLUCENT,

StatusBarManager.WINDOW\_STATUS\_BAR,

WindowManager.LayoutParams.FLAG\_TRANSLUCENT\_STATUS);

#### updateSystemUiVisibilityLw

private int updateSystemUiVisibilityLw() {

...

final int visibility = updateSystemBarsLw(win, mLastSystemUiFlags, tmpVisibility);

mFocusedApp = win.getAppToken();

mHandler.post(new Runnable() {

@Override

public void run() {

try {

IStatusBarService statusbar = getStatusBarService();

if (statusbar != null) {

//改变SystemUI的地方

statusbar.setSystemUiVisibility(visibility, 0xffffffff);

statusbar.topAppWindowChanged(needsMenu);

}

} catch (RemoteException e) {

// re-acquire status bar service next time it is needed.

mStatusBarService = null;

}

}

});

}

#### updateSystemBarsLw

private int updateSystemBarsLw(WindowState win, int oldVis, int vis) {

// apply translucent bar vis flags

WindowState transWin = mKeyguard != null && mKeyguard.isVisibleLw() && !mHideLockScreen ? mKeyguard : mTopFullscreenOpaqueWindowState;

vis = mStatusBarController.applyTranslucentFlagLw(transWin, vis, oldVis);

...

vis = mStatusBarController.updateVisibilityLw(transientStatusBarAllowed, oldVis, vis);

...

return vis;

}

上面的代码主要是通过mStatusBarController来处理状态栏的显示状态。现在我们来看看它对应的实现方法。在applyTranslucentFlagLw方法中

### StatusBarController.applyTranslucentFlagLw

在applyTranslucentFlagLw方法中，就真正的联系上了，我们之前设置的windowTranslucentStatus属性。为什么这么说？因为applyTranslucentFlagLw方法体有一个if ((win.getAttrs().flags & mTranslucentWmFlag) != 0)判断。这个判断其实就是判断我们有没有把windowTranslucentStatus属性设置为true。win.getAttrs().flags对应的我们设置的WindowManager.LayoutParams的flags，而mTranslucentWmFlag就等于WindowManager.LayoutParams.FLAG\_TRANSLUCENT\_STATUS。

如果我们设置了windowTranslucentStatus为true。这个if判断就为true,vis就增加一个mTranslucentFlag标志，其值为View.STATUS\_BAR\_TRANSLUCENT。这个标志很重要，我们现在先放到这，后面会用这个标志。这个新vis就会返回回去。updateVisibilityLw方法没有做什么很重要的事情就忽略不分析了

#### BarController

public BarController(String tag, int transientFlag, int unhideFlag, int translucentFlag, int statusBarManagerId, int translucentWmFlag) {

mTag = "BarController." + tag; //tag="StatusBar"

mTransientFlag = transientFlag; //transientFlag=View.STATUS\_BAR\_TRANSIENT

mUnhideFlag = unhideFlag; //unhideFlag=View.STATUS\_BAR\_UNHIDE

mTranslucentFlag = translucentFlag;//translucentFlag=View.STATUS\_BAR\_TRANSLUCENT,

mStatusBarManagerId = statusBarManagerId; //statusBarManagerId=StatusBarManager.WINDOW\_STATUS\_BAR

mTranslucentWmFlag = translucentWmFlag; //translucentWmFlag=WindowManager.LayoutParams.FLAG\_TRANSLUCENT\_STATUS

mHandler = new Handler();

}

#### applyTranslucentFlagLw

public int applyTranslucentFlagLw(WindowState win, int vis, int oldVis) {

if (mWin != null) {

if (win != null && (win.getAttrs().privateFlags

& WindowManager.LayoutParams.PRIVATE\_FLAG\_INHERIT\_TRANSLUCENT\_DECOR) == 0) {

//mTranslucentWmFlag=WindowManager.LayoutParams.FLAG\_TRANSLUCENT\_STATUS

//如果我们设置了FLAG\_TRANSLUCENT\_STATUS属性，vis就会添加一个mTranslucentFlag标记，即View.STATUS\_BAR\_TRANSLUCENT

if ((win.getAttrs().flags & mTranslucentWmFlag) != 0) {

vis |= mTranslucentFlag;

} else {

vis &= ~mTranslucentFlag;

}

} else {

vis = (vis & ~mTranslucentFlag) | (oldVis & mTranslucentFlag);

}

}

return vis;

}

#### updateVisibilityLw

public int updateVisibilityLw(boolean transientAllowed, int oldVis, int vis) {

if (mWin == null) return vis;

if (isTransientShowing() || isTransientShowRequested()) { // transient bar requested

if (transientAllowed) {

vis |= mTransientFlag;

if ((oldVis & mTransientFlag) == 0) {

vis |= mUnhideFlag; // tell sysui we're ready to unhide

}

setTransientBarState(TRANSIENT\_BAR\_SHOWING); // request accepted

} else {

setTransientBarState(TRANSIENT\_BAR\_NONE); // request denied

}

}

if (mTransientBarState != TRANSIENT\_BAR\_NONE) {

vis |= mTransientFlag; // ignore clear requests until transition completes

vis &= ~View.SYSTEM\_UI\_FLAG\_LOW\_PROFILE; // never show transient bars in low profile

}

if ((vis & mTranslucentFlag) != 0 || (oldVis & mTranslucentFlag) != 0) {

mLastTranslucent = SystemClock.uptimeMillis();

}

return vis;

}

### StatusBarManagerService

接下来就是调用statusbar.setSystemUiVisibility(visibility, 0xffffffff);，这才是真正改变SystemUI状态的接口。visibility就是updateSystemBarsLw的返回值，其实也就是mStatusBarController.applyTranslucentFlagLw和mStatusBarController.updateVisibilityLw的返回值

#### registerStatusBar

public void registerStatusBar(IStatusBar bar, StatusBarIconList iconList, List<ibinder> notificationKeys, List<statusbarnotification> notifications, int switches[], List<ibinder> binders) {

enforceStatusBarService();

Slog.i(TAG, "registerStatusBar bar=" + bar);

mBar = bar;

}

#### setSystemUiVisibility

public void setSystemUiVisibility(int vis, int mask) {

// also allows calls from window manager which is in this process.

enforceStatusBarService();

if (SPEW) Slog.d(TAG, "setSystemUiVisibility(0x" + Integer.toHexString(vis) + ")");

synchronized (mLock) {

updateUiVisibilityLocked(vis, mask);

disableLocked(mCurrentUserId, vis & StatusBarManager.DISABLE\_MASK, mSysUiVisToken, "WindowManager.LayoutParams");

}

}

#### updateUiVisibilityLocked

private void updateUiVisibilityLocked(final int vis, final int mask) {

if (mSystemUiVisibility != vis) {

mSystemUiVisibility = vis;

mHandler.post(new Runnable() {

public void run() {

if (mBar != null) {

try {

mBar.setSystemUiVisibility(vis, mask);

} catch (RemoteException ex) {

}

}

}

});

}

}

### SystemUI

它在BaseStatusBar.java中调用了StatusBarManagerService.java的registerStatusBar将CommandQueue注册到StatusBarManagerService与其产生关联。所以StatusBarManagerService中setSystemUiVisibility最终调用的是CommandQueue.java中方法。而CommandQueue.java和BaseStatusBar.java是通过接口回调实现对应关系的，PhoneStatusBar.java继承自BaseStatusBar.java。所以CommandQueue调用setSystemUiVisibility方法，最终调用的是PhoneStatusBar中的setSystemUiVisibility方法。

#### BaseStatusBar.start

public void start() {

...

// Connect in to the status bar manager service

StatusBarIconList iconList = new StatusBarIconList();

ArrayList<ibinder> notificationKeys = new ArrayList<ibinder>();

ArrayList<statusbarnotification> notifications = new ArrayList<statusbarnotification>();

mCommandQueue = new CommandQueue(this, iconList);

try {

mBarService.registerStatusBar(mCommandQueue, iconList, notificationKeys, notifications, switches, binders);

} catch (RemoteException ex) {

// If the system process isn't there we're doomed anyway.

}

...

}

#### CommandQueue

Adf

public CommandQueue(Callbacks callbacks, StatusBarIconList list) {

mCallbacks = callbacks;

mList = list;

}

public void setSystemUiVisibility(int vis, int mask) {

synchronized (mList) {

mHandler.removeMessages(MSG\_SET\_SYSTEMUI\_VISIBILITY);

mHandler.obtainMessage(MSG\_SET\_SYSTEMUI\_VISIBILITY, vis, mask, null).sendToTarget();

}

}

private final class H extends Handler {

public void handleMessage(Message msg) {

final int what = msg.what & MSG\_MASK;

switch (what) {

case MSG\_SET\_SYSTEMUI\_VISIBILITY:

mCallbacks.setSystemUiVisibility(msg.arg1, msg.arg2);

break;

...

}

}

}

### PhoneStatusBar.setSystemUiVisibility

PhoneStatusBar.java中setSystemUiVisibility的实现。这个方法真正关键地方是在调用barMode这个方法中获取最新的mode状态。它会用vis比较各个flag，vis就是我们在PhoneWindowManager.java中传入的参数。我们知道在第二步后面增加了View.STATUS\_BAR\_TRANSLUCENT这个标志，说了很重要。在这里才真正的体现出来了。barMode方法通过比较得到mode为MODE\_TRANSLUCENT。然后，通过checkBarModes应用相应的mode。

@Override // CommandQueue

public void setSystemUiVisibility(int vis, int mask) {

final int oldVal = mSystemUiVisibility;

final int newVal = (oldVal&~mask) | (vis&mask);

final int diff = newVal ^ oldVal;

...

// update status bar mode

final int sbMode = computeBarMode(oldVal, newVal, mStatusBarView.getBarTransitions(), View.STATUS\_BAR\_TRANSIENT, View.STATUS\_BAR\_TRANSLUCENT);

final boolean sbModeChanged = sbMode != -1;

boolean checkBarModes = false;

if (sbModeChanged && sbMode != mStatusBarMode) {

mStatusBarMode = sbMode;

checkBarModes = true;

}

if (checkBarModes) {

checkBarModes();

}

}

private int computeBarMode(int oldVis, int newVis, BarTransitions transitions, int transientFlag, int translucentFlag) {

final int oldMode = barMode(oldVis, transientFlag, translucentFlag);

final int newMode = barMode(newVis, transientFlag, translucentFlag);

if (oldMode == newMode) {

return -1; // no mode change

}

return newMode;

}

private int barMode(int vis, int transientFlag, int translucentFlag) {

return (vis & transientFlag) != 0 ? MODE\_SEMI\_TRANSPARENT

: (vis & translucentFlag) != 0 ? MODE\_TRANSLUCENT

: (vis & View.SYSTEM\_UI\_FLAG\_LOW\_PROFILE) != 0 ? MODE\_LIGHTS\_OUT

: MODE\_OPAQUE;

}

private void checkBarModes() {

if (mDemoMode) return;

int sbMode = mStatusBarMode;

if (panelsEnabled() && (mInteractingWindows & StatusBarManager.WINDOW\_STATUS\_BAR) != 0) {

// if panels are expandable, force the status bar opaque on any interaction

sbMode = MODE\_OPAQUE;

}

checkBarMode(sbMode, mStatusBarWindowState, mStatusBarView.getBarTransitions());

if (mNavigationBarView != null) {

checkBarMode(mNavigationBarMode,

mNavigationBarWindowState, mNavigationBarView.getBarTransitions());

}

}

private void checkBarMode(int mode, int windowState, BarTransitions transitions) {

final boolean anim = (mScreenOn == null || mScreenOn) && windowState != WINDOW\_STATE\_HIDDEN;

transitions.transitionTo(mode, anim);

}

在checkBarMode中transitions是从PhoneStatusBarView中创建的PhoneStatusBarTransitions。

public PhoneStatusBarView(Context context, AttributeSet attrs) {

super(context, attrs);

mBarTransitions = new PhoneStatusBarTransitions(this);

}

public BarTransitions getBarTransitions() {

return mBarTransitions;

}

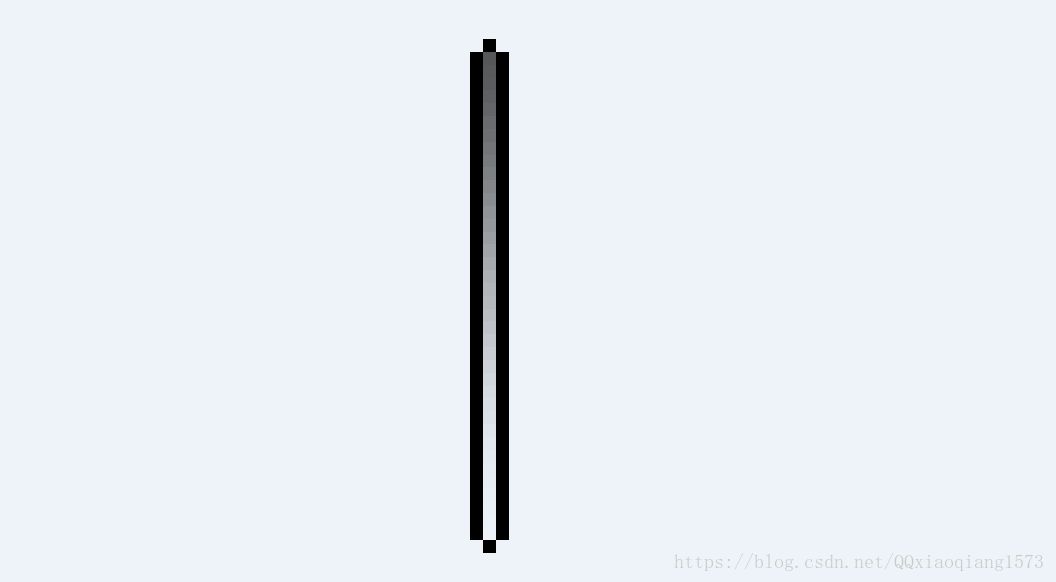
public PhoneStatusBarTransitions(PhoneStatusBarView view) {

super(view, R.drawable.status\_background);

mView = view;

}

PhoneStatusBarTransitions继承自BarTransitions。通过PhoneStatusBarTransitions的构造函数，我们看到其调用了父类的构造方法super(view, R.drawable.status\_background);。在其父类的构造方法中会将R.drawable.status\_background这个资源图片设置到view的背景中去。那R.drawable.status\_background是个什么样的图片



原来我们看到的状态渐变效果就是因为设置这张点9的背景图片。那问题来，我们的状态栏不应该一直都是这种渐变效果吗？哈哈，是不是被忽悠了，客官别急，还有最后一步

### BarTransitions

public BarTransitions(View view, int gradientResourceId) {

mTag = "BarTransitions." + view.getClass().getSimpleName();

mView = view;

mBarBackground = new BarBackgroundDrawable(mView.getContext(), radientResourceId);

if (HIGH\_END) {

mView.setBackground(mBarBackground);

}

}

public void transitionTo(int mode, boolean animate) {

// low-end devices do not support translucent modes, fallback to opaque

if (!HIGH\_END && (mode == MODE\_SEMI\_TRANSPARENT || mode == MODE\_TRANSLUCENT)) {

mode = MODE\_OPAQUE;

}

if (mMode == mode) return;

int oldMode = mMode;

mMode = mode;

if (DEBUG) Log.d(mTag, String.format("%s -> %s animate=%s",

modeToString(oldMode), modeToString(mode), animate));

onTransition(oldMode, mMode, animate);

}

protected void onTransition(int oldMode, int newMode, boolean animate) {

if (HIGH\_END) {

applyModeBackground(oldMode, newMode, animate);

}

}

protected void applyModeBackground(int oldMode, int newMode, boolean animate) {

if (DEBUG) Log.d(mTag, String.format("applyModeBackground oldMode=%s newMode=%s animate=%s",

modeToString(oldMode), modeToString(newMode), animate));

mBarBackground.applyModeBackground(oldMode, newMode, animate);

}

private static class BarBackgroundDrawable extends Drawable {

public BarBackgroundDrawable(Context context, int gradientResourceId) {

final Resources res = context.getResources();

mGradient = res.getDrawable(gradientResourceId);

mInterpolator = new LinearInterpolator();

}

public void applyModeBackground(int oldMode, int newMode, boolean animate) {

if (mMode == newMode) return;

mMode = newMode;

mAnimating = animate;

if (animate) {

long now = SystemClock.elapsedRealtime();

mStartTime = now;

mEndTime = now + BACKGROUND\_DURATION;

mGradientAlphaStart = mGradientAlpha;

mColorStart = mColor;

}

invalidateSelf();

}

@Override

public void draw(Canvas canvas) {

int targetGradientAlpha = 0, targetColor = 0;

if (mMode == MODE\_TRANSLUCENT) {

targetGradientAlpha = 0xff;

} else if (mMode == MODE\_SEMI\_TRANSPARENT) {

targetColor = mSemiTransparent;

} else {

targetColor = mOpaque;

}

if (!mAnimating) {

mColor = targetColor;

mGradientAlpha = targetGradientAlpha;

} else {

...

}

if (mGradientAlpha > 0) {

mGradient.setAlpha(mGradientAlpha);

mGradient.draw(canvas);

}

if (Color.alpha(mColor) > 0) {

canvas.drawColor(mColor);

}

if (mAnimating) {

invalidateSelf(); // keep going

}

}

}

最后，我们再来看下PhoneStatusBar.java中checkBarMode方法的transitions.transitionTo调用。这个transitionTo最终调用的是BarTransitions.java中的transitionTo方法，通过上面的源码可以知道它其实最后调用就是BarBackgroundDrawable中的applyModeBackground方法。会传入mode，·并调用invalidateSelf方法。最终在draw中，根据mode显示对应的背景效果。这里因为我们的mode为MODE\_TRANSLUCENT，所以，最终会调用mGradient.draw(canvas);。而mGradient就是这个R.drawable.status\_background图片的drawable，所以我们在mode为MODE\_TRANSLUCENT就会显示为一个渐变的效果。而在其它mode时候显示一个rgb颜色背景。

### 布局也会拓展到状态栏和导航栏后面去

ViewRootImpl.java

private int getImpliedSystemUiVisibility(WindowManager.LayoutParams params) {

int vis = 0;

// Translucent decor window flags imply stable system ui visibility.

if ((params.flags & WindowManager.LayoutParams.FLAG\_TRANSLUCENT\_STATUS) != 0) {

vis |= View.SYSTEM\_UI\_FLAG\_LAYOUT\_STABLE | View.SYSTEM\_UI\_FLAG\_LAYOUT\_FULLSCREEN;

}

if ((params.flags & WindowManager.LayoutParams.FLAG\_TRANSLUCENT\_NAVIGATION) != 0) {

vis |= View.SYSTEM\_UI\_FLAG\_LAYOUT\_STABLE | View.SYSTEM\_UI\_FLAG\_LAYOUT\_HIDE\_NAVIGATION;

}

return vis;

}

系统在我们添加windowTranslucentStatus和windowTranslucentNavigation属性时候，会自动为我们增加View.SYSTEM\_UI\_FLAG\_LAYOUT\_STABLE | View.SYSTEM\_UI\_FLAG\_LAYOUT\_FULLSCREEN | View.SYSTEM\_UI\_FLAG\_LAYOUT\_HIDE\_NAVIGATION属性。

那要怎么解决我们的布局会被拓展到系统栏后面的效果。在layout.xml增加android:fitsSystemWindows="true"即可



### setSystemUiVisibility(vis, mask)

真实精彩啊

https://cloud.tencent.com/developer/article/1154092

<http://www.dayexie.com/detail1885435.html>

### Theme.NoTitleBar.Fullscreen

<**item name="windowFullscreen"**>true</**item**>

<**item name="windowNoTitle"**>true</**item**>

if (a.getBoolean(R.styleable.Window\_windowFullscreen, false)) {

setFlags(FLAG\_FULLSCREEN, FLAG\_FULLSCREEN & (~getForcedWindowFlags()));

}

**Android setSystemUiVisibility详解**

[**https://blog.csdn.net/QQxiaoqiang1573/article/details/79867127**](https://blog.csdn.net/QQxiaoqiang1573/article/details/79867127)

[**https://www.jianshu.com/p/08ff70c15667**](https://www.jianshu.com/p/08ff70c15667)

**https://www.cnblogs.com/8dull/p/5383135.html**

# powerUI

## powerUI

### start

dfg

start函数主要实现：

1)实例化PowerManager，因为是处理和Power相关的业务，必须实例化PowerManager.

2)初始化关屏时间，关屏时间会持续刷新。

3)实例化PowerNotificationWarnings,Power相关的事件会通过Notification方式进行处理。

4)监控电池电量，在电量不足是给出警告和提示。

5)更新电池电量等级。

**public void** start() {  
 **mPowerManager** = (PowerManager) **mContext**.getSystemService(Context.***POWER\_SERVICE***);  
 **mHardwarePropertiesManager** = (HardwarePropertiesManager)  
 **mContext**.getSystemService(Context.***HARDWARE\_PROPERTIES\_SERVICE***);  
 **mScreenOffTime** = **mPowerManager**.isScreenOn() ? -1 : SystemClock.*elapsedRealtime*();  
 **mWarnings** = **new** PowerNotificationWarnings(**mContext**, getComponent(PhoneStatusBar.**class**));  
  
 ContentObserver obs = **new** ContentObserver(**mHandler**) {  
 @Override  
 **public void** onChange(**boolean** selfChange) {  
 updateBatteryWarningLevels();  
 }  
 };  
 **final** ContentResolver resolver = **mContext**.getContentResolver();  
 resolver.registerContentObserver(Settings.Global.*getUriFor*(  
 Settings.Global.LOW\_POWER\_MODE\_TRIGGER\_LEVEL),  
 **false**, obs, UserHandle.USER\_ALL);  
 updateBatteryWarningLevels();  
 **mReceiver**.init();  
  
 initTemperatureWarning();  
}

得到powerManager。主要用于在62行得到屏幕关闭的时间，以及后面判断是否进入powerSaveMode，setSaverMode(mPowerManager.isPowerSaveMode());

得到一个WarningsUI 类型的mWanings，只要用于在状态栏显示当前的power状态

注册一个contentObserver 用于检测 Settings.Global.LOW\_POWER\_MODE\_TRIGGER\_LEVEL的变化，如果变化则调用updateBatteryWarningLevels 更新waninglevels

### updateBatteryWarningLevels

我们来看看updateBatteryWarningLevels的实现

**void** updateBatteryWarningLevels() {  
 **int** critLevel = **mContext**.getResources().getInteger(  
 com.android.internal.R.integer.***config\_criticalBatteryWarningLevel***);  
  
 **final** ContentResolver resolver = **mContext**.getContentResolver();  
 **int** defWarnLevel = **mContext**.getResources().getInteger(  
 com.android.internal.R.integer.***config\_lowBatteryWarningLevel***);  
 **int** warnLevel = Settings.Global.*getInt*(resolver,  
 Settings.Global.LOW\_POWER\_MODE\_TRIGGER\_LEVEL, defWarnLevel);  
 **if** (warnLevel == 0) {  
 warnLevel = defWarnLevel;  
 }  
 **if** (warnLevel < critLevel) {  
 warnLevel = critLevel;  
 }  
  
 **mLowBatteryReminderLevels**[0] = warnLevel;  
 **mLowBatteryReminderLevels**[1] = critLevel;  
 **mLowBatteryAlertCloseLevel** = **mLowBatteryReminderLevels**[0]  
 + **mContext**.getResources().getInteger(  
 com.android.internal.R.integer.***config\_lowBatteryCloseWarningBump***);  
}

#### 电池电量有三个关键值定义

Fdg

 <!-- Display low battery warning when battery level dips to this value.  
         Also, the battery stats are flushed to disk when we hit this level.  -->  
    <integer name="config\_criticalBatteryWarningLevel">5</integer>

<!-- Display low battery warning when battery level dips to this value -->  
    <integer name="config\_lowBatteryWarningLevel">15</integer>

 <!-- Close low battery warning when battery level reaches the lowBatteryWarningLevel  
         plus this -->  
    <integer name="config\_lowBatteryCloseWarningBump">5</integer>

将warnLevel,critLevel,放到mLowBatteryReminderLevels 数组，并更新mLowBatteryAlertCloseLevel

### mReceiver.init()

而mReceiver  是在下面赋值的 private final Receiver mReceiver = new Receiver();  
那我们再来看看Receiver 的实现

**private final class** Receiver **extends** BroadcastReceiver {  
  
 **public void** init() {  
 *// Register for Intent broadcasts for...* IntentFilter filter = **new** IntentFilter();  
 filter.addAction(Intent.***ACTION\_BATTERY\_CHANGED***);  
 filter.addAction(Intent.***ACTION\_SCREEN\_OFF***);  
 filter.addAction(Intent.***ACTION\_SCREEN\_ON***);  
 filter.addAction(Intent.ACTION\_USER\_SWITCHED);  
 filter.addAction(PowerManager.ACTION\_POWER\_SAVE\_MODE\_CHANGING);  
 filter.addAction(PowerManager.***ACTION\_POWER\_SAVE\_MODE\_CHANGED***);  
 **mContext**.registerReceiver(**this**, filter, **null**, **mHandler**);  
 }  
  
 @Override  
 **public void** onReceive(Context context, Intent intent) {  
 String action = intent.getAction();  
 **if** (action.equals(Intent.***ACTION\_BATTERY\_CHANGED***)) {  
 **final int** oldBatteryLevel = **mBatteryLevel**;  
 **mBatteryLevel** = intent.getIntExtra(BatteryManager.***EXTRA\_LEVEL***, 100);  
 **final int** oldBatteryStatus = **mBatteryStatus**;  
 **mBatteryStatus** = intent.getIntExtra(BatteryManager.***EXTRA\_STATUS***,  
 BatteryManager.***BATTERY\_STATUS\_UNKNOWN***);  
 **final int** oldPlugType = **mPlugType**;  
 **mPlugType** = intent.getIntExtra(BatteryManager.***EXTRA\_PLUGGED***, 1);  
 **final int** oldInvalidCharger = **mInvalidCharger**;  
 **mInvalidCharger** = intent.getIntExtra(BatteryManager.EXTRA\_INVALID\_CHARGER, 0);  
  
 **final boolean** plugged = **mPlugType** != 0;  
 **final boolean** oldPlugged = oldPlugType != 0;  
  
 **int** oldBucket = findBatteryLevelBucket(oldBatteryLevel);  
 **int** bucket = findBatteryLevelBucket(**mBatteryLevel**);  
  
 **if** (***DEBUG***) {  
 Slog.*d*(***TAG***, **"buckets ....."** + **mLowBatteryAlertCloseLevel** + **" .. "** + **mLowBatteryReminderLevels**[0]  
 + **" .. "** + **mLowBatteryReminderLevels**[1]);  
 Slog.*d*(***TAG***, **"level "** + oldBatteryLevel + **" --> "** + **mBatteryLevel**);  
 Slog.*d*(***TAG***, **"status "** + oldBatteryStatus + **" --> "** + **mBatteryStatus**);  
 Slog.*d*(***TAG***, **"plugType "** + oldPlugType + **" --> "** + **mPlugType**);  
 Slog.*d*(***TAG***, **"invalidCharger "** + oldInvalidCharger + **" --> "** + **mInvalidCharger**);  
 Slog.*d*(***TAG***, **"bucket "** + oldBucket + **" --> "** + bucket);  
 Slog.*d*(***TAG***, **"plugged "** + oldPlugged + **" --> "** + plugged);  
 }  
  
 **mWarnings**.update(**mBatteryLevel**, bucket, **mScreenOffTime**);  
 **if** (oldInvalidCharger == 0 && **mInvalidCharger** != 0) {  
 Slog.*d*(***TAG***, **"showing invalid charger warning"**);  
 **mWarnings**.showInvalidChargerWarning();  
 **return**;  
 } **else if** (oldInvalidCharger != 0 && **mInvalidCharger** == 0) {  
 **mWarnings**.dismissInvalidChargerWarning();  
 } **else if** (**mWarnings**.isInvalidChargerWarningShowing()) {  
 *// if invalid charger is showing, don't show low battery* **return**;  
 }  
  
 **boolean** isPowerSaver = **mPowerManager**.isPowerSaveMode();  
 **if** ( DjiSystemUIHelper.*getIns*().showBatteryWarning()  
 && !plugged  
 && !isPowerSaver  
 && (bucket < oldBucket || oldPlugged)  
 && **mBatteryStatus** != BatteryManager.***BATTERY\_STATUS\_UNKNOWN*** && bucket < 0) {  
 *// only play SFX when the dialog comes up or the bucket changes* **final boolean** playSound = bucket != oldBucket || oldPlugged;  
 **mWarnings**.showLowBatteryWarning(playSound);  
 } **else if** (isPowerSaver || plugged || (bucket > oldBucket && bucket > 0)) {  
 **mWarnings**.dismissLowBatteryWarning();  
 } **else** {  
 **mWarnings**.updateLowBatteryWarning();  
 }  
 } **else if** (Intent.***ACTION\_SCREEN\_OFF***.equals(action)) {  
 **mScreenOffTime** = SystemClock.*elapsedRealtime*();  
 } **else if** (Intent.***ACTION\_SCREEN\_ON***.equals(action)) {  
 **mScreenOffTime** = -1;  
 } **else if** (Intent.ACTION\_USER\_SWITCHED.equals(action)) {  
 **mWarnings**.userSwitched();  
 } **else** {  
 Slog.*w*(***TAG***, **"unknown intent: "** + intent);  
 }  
 }  
};

Receiver 的init函数主要是注册一个监听事件的IntentFilter，当有我们监听的事件到来时会调动onReceive函数  
我们以Intent.ACTION\_USER\_SWITCHED为例，看看mWarnings是如何在systemUI中更新通知的  
mWarnings.userSwitched();

## PowerNotificationWarnings

我们以Intent.ACTION\_USER\_SWITCHED为例，看看mWarnings是如何在systemUI中更新通知的  
mWarnings.userSwitched()

### userSwitched

f

**public void** userSwitched() {  
 updateNotification();  
}

### updateNotification

f

**private void** updateNotification() {  
 **if** (***DEBUG***) Slog.*d*(***TAG***, **"updateNotification mWarning="** + **mWarning** + **" mPlaySound="** + **mPlaySound** + **" mInvalidCharger="** + **mInvalidCharger**);  
 **if** (**mInvalidCharger**) {  
 showInvalidChargerNotification();  
 **mShowing** = ***SHOWING\_INVALID\_CHARGER***;  
 } **else if** (**mWarning**) {  
 showWarningNotification();  
 **mShowing** = ***SHOWING\_WARNING***;  
 } **else** {  
 **mNoMan**.cancelAsUser(***TAG\_BATTERY***, SystemMessage.***NOTE\_BAD\_CHARGER***, UserHandle.ALL);  
 **mNoMan**.cancelAsUser(***TAG\_BATTERY***, SystemMessage.***NOTE\_POWER\_LOW***, UserHandle.ALL);  
 **mShowing** = ***SHOWING\_NOTHING***;  
 }  
}

会看当前的电源的状态，我们mInvalidCharger=true为例

### showInvalidChargerNotification

**private void** showInvalidChargerNotification() {  
 **final** Notification.Builder nb = **new** Notification.Builder(**mContext**)  
 .setSmallIcon(R.drawable.***ic\_power\_low***)  
 .setWhen(0)  
 .setShowWhen(**false**)  
 .setOngoing(**true**)  
 .setContentTitle(**mContext**.getString(R.string.***invalid\_charger\_title***))  
 .setContentText(**mContext**.getString(R.string.***invalid\_charger\_text***))  
 .setPriority(Notification.***PRIORITY\_MAX***)  
 .setVisibility(Notification.***VISIBILITY\_PUBLIC***)  
 .setColor(**mContext**.getColor(  
 com.android.internal.R.color.***system\_notification\_accent\_color***));  
 SystemUI.*overrideNotificationAppName*(**mContext**, nb);  
 **final** Notification n = nb.build();  
 **mNoMan**.cancelAsUser(***TAG\_BATTERY***, SystemMessage.***NOTE\_POWER\_LOW***, UserHandle.ALL);  
 **mNoMan**.notifyAsUser(***TAG\_BATTERY***, SystemMessage.***NOTE\_BAD\_CHARGER***, n, UserHandle.ALL);  
}

原来是通过Notification 来更新通知的

# QS

## Inversion

Quick settings tile: Invert colors

ColorInversionTile

root@gl300k:/ # settings get secure accessibility\_display\_inversion\_enabled

1

settings put secure accessibility\_display\_inversion\_enabled 0

反色逻辑

# Notification

APP与SystemUI的UI交互的一种方式

### 代码架构

#### 代码位置

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. # 应用接口相关代码
2. frameworks/base/core/java/android/app/Notification.java
3. frameworks/base/core/java/android/app/NotificationManager.java
4. # 系统内部接口相关代码
5. frameworks/base/core/java/com/android/internal/statusbar/StatusBarNotification.java
6. frameworks/base/core/java/com/android/internal/statusbar/StatusBarIcon.java
7. # System Server 相关代码
8. frameworks/base/services/java/com/android/server/NotificationManagerService.java
9. frameworks/base/services/java/com/android/server/StatusBarManagerService.java
10. # SystemUI 相关代码
11. frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/NotificationData.java
12. frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/StatusBarIconView.java
13. frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/BaseStatusBar.java
14. frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/phone/PhoneStatusBar.java
15. frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/phone/Ticker.java

### UI 表现

目前 android 系统中的通知 UI 表现有3种：

* Ticker：就是刚发送一条通知的时候，通知栏上会有 一个小图标 + 一条文本 伴随一个滚动动画滚动出来，过一小段时间（3s）后会消失。作用是用一个动态效果告诉用户有一条通知来了。
* StatusBar Icon：当 Ticker 消失后，状态栏的左上角就会多一个小图标（和刚刚 Ticker 的小图标一样），告诉用户已经有一条通知了。
* Notification Row：发送一条通知后，下拉通知面板中的通知列表中会多一条通知行。上面那2个相当于是预览信息，这里的就是通知的完整信息了（不过话说本来通知就是应用程序的预览信息）。

所以说应用要对系统发送一条通知，显示上可以设置的就是这3大块（效果图见后面的系统分析那里），还有一个可以设置的是不可见的部分——PendingIntent：点击通知要完成的动作，一般是启动某个程序的 Activity 界面（还可以设置一个通知被删除时候的动作）。

通知分为2种：

* 可删除的：这种可以在通知面板上由用户用手动删除，删除后，状态栏上对应的这条通知的小图标也会被删除掉（一般的通知类通知可以用这种）。
* 不可删除的：这种通知用户无法手动删除，只能由发送通知的程序用代码取消掉（一些工具类的挂在通知面板上的快捷方式可以用这种，例如音乐播放器）。当然这种方式可能会被一些流氓应用利用，长按这条通知，然后可以查看到发送这条通知的应用（系统设置中），强制终止掉发送通知的 apk（在系统设置中强制终止 apk 或是进程自己挂掉了，发送的通知会被清楚掉）。当然还可以更狠点，直接禁止这个 apk 发送通知。

上面那2种类型可以通过设置 Notification 的 flags FLAG\_NO\_CLEAR 来表示。同时还有一个容易和这个搞混的标志叫： FLAG\_ONGOING\_EVENT 这个表示发送这条通知的应用正在进行某些工作。设置了这个正在进行的标志，效果和设置不可删除的是一样的，用户同样无法手动去删除这条通知。在老的 android 版本（好像应该是 4.0 以前），通知面板上显示通知是分2组显示的，一组叫“正在进行的”，一组叫“通知”。设置了 FLAG\_ONGOING\_EVENT 会被分到“正在进行的”那组，其他的在“通知”组。不过从 4.0 开始，android 改了设计了，全都归到一组去显示了。所以 4.0 以后，FLAG\_ONGOING\_EVENT 从视觉上看不出和没设置的有啥区别的（当然如果你自己定制的 framework 你可以继续改回分组的显示方式）

#### 优先级

Notification 支持5个级别，分别是：Max，High，Default，Low，Min，级别也是依次降低。

Max：提示一些重要并且紧急的通知，主要适用于，在开始某项任务前，必须要解决和处理的提示信息。

High：提示一些比较重要的信息，比如短信等聊天内容。

Default：系统默认的级别，在同是Default级别的其他提醒，不会覆盖掉当前的提醒。

Low：提示用户，一些紧迫性较低的提醒。

Min：最低级别，只有用户在滑下通知栏时，才会显示提示信息，否则是不会出现在状态栏中。比如一些天气信息的提醒。



#### Notification的堆栈处理

同一应用的两个相同的提示信息，进行了堆栈处理，比如途中两条相同的短信提醒，并不是显示两个提醒。而是提示有两条信息。



当然，也可以使用一些扩展性的布局，通过堆栈的方式，获取更好的用户体验



#### Heads-up Notification

在Android5.0 API Level21 如下图中，在屏幕顶端的浮动的窗口，又叫Heads-up Notification，当应用全屏显示的时候，这时使用heads-up notification就比较合适了，这个Notification具有很高的优先级，可以使用系统的铃声和振动效果。



在Android5.0中，Notification也可以出现在锁屏的情况中，这样应用就可以进行一些播放器的操作，可以通过[setVisibility()](http://developer.android.com/reference/android/support/v4/app/NotificationCompat.Builder.html#setVisibility(int))  来控制Notification在锁屏情况下的显示和隐藏。Google给我们提供了三个备选参数，分别是：

* [VISIBILITY\_PUBLIC](http://developer.android.com/reference/android/support/v4/app/NotificationCompat.html%23VISIBILITY_PUBLIC) 显示Notification完整的内容
* [VISIBILITY\_SECRET](http://developer.android.com/reference/android/support/v4/app/NotificationCompat.html%23VISIBILITY_SECRET) 不显示任何内容
* [VISIBILITY\_PRIVATE](http://developer.android.com/reference/android/support/v4/app/NotificationCompat.html%23VISIBILITY_PRIVATE) 只显示基本的信息，比如图标和标题，隐藏了内
* 显示有三个等级
* ***VISIBILITY\_PUBLIC：***在任何地方都可看得到
* ***VISIBILITY\_PRIVATE：***如果你手机上了 pin，解锁 pin 过后才能看到通知
* ***VISIBILITY\_SECRET：***在任何地方都看不到

#### 使用上的建议

官方文档上，建议我们的app添加设置选项，让用户来控制是否显示提示信息，是否有声音，震动的提示，推荐使用Android提供的图标和显示样式，尤其



，这是我们在设计app时很容易出现的问题，总想与众不同，结果用户体验很差。

官方文档中，推荐我们调用一些LED提醒，这样在屏幕关闭的情况下，比如通过呼吸灯的颜色，来提醒用户。在设计app时，提示信息要友好，一些不必要的，

不重要的提醒，就不要骚扰用户啦，小心用户烦了，分分钟干掉你……

### 通知标示

应用 new 一个 Notification 之后，获取 NotificationManager(NM) 之后，调用相应的接口就可以发送通知：

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. // ================== NotificationManager.java =====================
2. /\*
3. \* Post a notification to be shown in the status bar. If a notification with
4. \* the same id has already been posted by your application and has not yet been canceled, it
5. \* will be replaced by the updated information.
6. \*
7. \* @param id An identifier for this notification unique within your
8. \*        application.
9. \* @param notification A {@link Notification} object describing what to show the user. Must not
10. \*        be null.
11. \*/
12. **public** **void** notify(**int** id, Notification notification) {
13. notify(**null**, id, notification);
14. }
16. /\*
17. \* Post a notification to be shown in the status bar. If a notification with
18. \* the same tag and id has already been posted by your application and has not yet been
19. \* canceled, it will be replaced by the updated information.
20. \*
21. \* @param tag A string identifier for this notification.  May be {@code null}.
22. \* @param id An identifier for this notification.  The pair (tag, id) must be unique
23. \*        within your application.
24. \* @param notification A {@link Notification} object describing what to
25. \*        show the user. Must not be null.
26. \*/
27. **public** **void** notify(String tag, **int** id, Notification notification) {
28. **int**[] idOut = **new** **int**[1];
29. INotificationManager service = getService();
30. String pkg = mContext.getPackageName();
31. **if** (notification.sound != **null**) {
32. notification.sound = notification.sound.getCanonicalUri();
33. }
34. **if** (localLOGV) Slog.v(TAG, pkg + ": notify(" + id + ", " + notification + ")");
35. **try** {
36. service.enqueueNotificationWithTag(pkg, tag, id, notification, idOut,
37. user.getIdentifier());
38. **if** (id != idOut[0]) {
39. Slog.w(TAG, "notify: id corrupted: sent " + id + ", got back " + idOut[0]);
40. }
41. } **catch** (RemoteException e) {
42. }
43. }
44. /\*
45. \* Cancel a previously shown notification.  If it's transient, the view
46. \* will be hidden.  If it's persistent, it will be removed from the status
47. \* bar.
48. \*/
49. **public** **void** cancel(**int** id) {
50. cancel(**null**, id);
51. }
52. /\*
53. \* Cancel a previously shown notification.  If it's transient, the view
54. \* will be hidden.  If it's persistent, it will be removed from the status
55. \* bar.
56. \*/
57. **public** **void** cancel(String tag, **int** id) {
58. INotificationManager service = getService();
59. String pkg = mContext.getPackageName();
60. **if** (localLOGV) Slog.v(TAG, pkg + ": cancel(" + id + ")");
61. **try** {
62. service.cancelNotificationWithTag(pkg, tag, id, UserHandle.myUserId());
63. } **catch** (RemoteException e) {
64. }
65. }
66. /\*
67. \* Cancel all previously shown notifications. See {@link #cancel} for the
68. \* detailed behavior.
69. \*/
70. **public** **void** cancelAll() {
71. INotificationManager service = getService();
72. String pkg = mContext.getPackageName();
73. **if** (localLOGV) Slog.v(TAG, pkg + ": cancelAll()");
74. **try** {
75. service.cancelAllNotifications(pkg, UserHandle.myUserId());
76. } **catch** (RemoteException e) {
77. }
78. }

NMS 中有几种方式可以用来标示一条通知。首先说说为什么要标示一条通知。因为发送一条通知，之后可以更新它的状态（显示的信息），取消它（删除），所以要有一个东西能唯一表示一条通知。首先有一个原则，那就是除了系统以外，**一个应用只能操作自己发送的通知**。所以对于普通应用，虽然说有几种标示，但是都在 pkg（包名）这个大的标示下（其实还有一个大的标示，就是用户，不过我们这里先不讨论多用户）。所以看 NM 接口的实现，都自动把调用接口的应用的 pkg 发给 NMS，不让应用自己设置（免得有些应用故意写别人的包名，干坏事）。

然后我们看到接口，有 id（int） 和 tag（String） 这2种标示。一般应用开发中 id 用得比较多，然后我们看下 NMS 中识别的代码：

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. // lock on mNotificationList
2. **private** **int** indexOfNotificationLocked(String pkg, String tag, **int** id, **int** userId)
3. {
4. ArrayList<NotificationRecord> list = mNotificationList;
5. **final** **int** len = list.size();
6. **for** (**int** i=0; i<len; i++) {
7. NotificationRecord r = list.get(i);
8. **if** (!notificationMatchesUserId(r, userId) || r.id != id) {
9. **continue**;
10. }
11. **if** (tag == **null**) {
12. **if** (r.tag != **null**) {
13. **continue**;
14. }
15. } **else** {
16. **if** (!tag.equals(r.tag)) {
17. **continue**;
18. }
19. }
20. **if (r.pkg.equals(pkg)) {**
21. **return** i;
22. }
23. }
24. **return** -1;
25. }

发现优先匹配 id，然后才是 tag（最后包名那来个保险，确保你只能操作自己发的通知）。所以说要自己设计通知的 id 话，可以随意点，因为系统帮你确保你的 id 只在你的 pkg 中有效了，android 系统里面是直接拿图标的 R 资源的 id 来做 id 用的。也有一些用 tag，不过那个用得少。

### Content

接下来介绍下通知的显示内容。通知的 Content 一般来说是这样的：



从 4.0 开始多了一个 big view 的模式，是这样的：



前面看了通知的接口。应用需要构造出一个 Notification 对象出来。在 2.x 的时候是直接设置 Notification 的字段的（里面的字段都是 public 的），后面 android 本着封装的设计感觉原来的不太好，所以搞了一个 Notification.Builder 出来，之后建议开发者用这个 Builder 来构造 Notification。其实2个原理上基本上是一样的，但是某些地方有点点不一样。就算是新的 sdk，你仍然可以坚持直接设置 Notification。所以我们来分开说一下：

#### 直接设置 Notification

* icon(int)：通知图标的资源 id。是上面 normal 模式的 2。
* iconLevel(int)：上面那个 icon 可以设成 LevelListDrawable 的，然后可以通过设置 iconLevel 来改变 icon 的图标 level，如果定时改变的话，可以形成一些下载通知图标的动态效果。
* setLatestEventInfo：这是个@Deprecated接口，参数如下：

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. @Deprecated
2. **public** **void** setLatestEventInfo(Context context,
3. CharSequence contentTitle, CharSequence contentText, PendingIntent contentIntent) {
4. // TODO: rewrite this to use Builder
5. RemoteViews contentView = **new** RemoteViews(context.getPackageName(),
6. R.layout.notification\_template\_base);
7. **if** (**this**.icon != 0) {
8. contentView.setImageViewResource(R.id.icon, **this**.icon);
9. }
10. **if** (priority < PRIORITY\_LOW) {
11. contentView.setInt(R.id.icon,
12. "setBackgroundResource", R.drawable.notification\_template\_icon\_low\_bg);
13. contentView.setInt(R.id.status\_bar\_latest\_event\_content,
14. "setBackgroundResource", R.drawable.notification\_bg\_low);
15. }
16. **if** (contentTitle != **null**) {
17. contentView.setTextViewText(R.id.title, contentTitle);
18. }
19. **if** (contentText != **null**) {
20. contentView.setTextViewText(R.id.text, contentText);
21. }
22. **if** (**this**.when != 0) {
23. contentView.setViewVisibility(R.id.time, View.VISIBLE);
24. contentView.setLong(R.id.time, "setTime", when);
25. }
26. **if** (**this**.number != 0) {
27. NumberFormat f = NumberFormat.getIntegerInstance();
28. contentView.setTextViewText(R.id.info, f.format(**this**.number));
29. }
30. **this**.contentView = contentView;
31. **this**.contentIntent = contentIntent;
32. }

这个可以说以前（4.0 之前）通知的最主要的部分了。看这个函数里面的实现知道，系统帮我们提供了一个通知的模板（xml 在 frameworks/base/core/res/res 下面），帮我们 new 了一个 RemoteViews 来当作通知的 ContentView。一个通知必须要有的2个元素，一个是 id，另外一个就是 contentView。这个 RemoteViews 是跨进程的 UI 组件，这个东西以后再分析，这里先不管它。系统的模板弄出来的样子就和上面的 normal view 那样。这里参数可以让你设置通知的标题（title）和内容（text）。分别对应 normal view 的 1 和 3。

这里除了 UI 元素设置，最后还有一个 PendingIntent 的参数（contentIntent），算是功能性设置。注释中说是用户点击通知的时候对应完成的功能。可以是 activity（点击时 launch），也可以是 broadcast（点击时发送）。可以说还是挺灵活的，这里就不展开分析了。

**number(int)**

标准模板中的右下角显示一个小数字（一些消息、邮件类的通知很有用），对应 normal view 的 4。注意如果不想显示把这个设置 0 就可以了（默认是 0，上面 setLatestEventInfo 处理那里代码能看得出的）。

**when(long)**

标志模板中右上角显示发送通知的时间，对应 normal view 的 6。看到类型是 long 的就知道这个时间是自从 1970-01-01 00:00:00.0 的 UTC，可以通过 System.currentTimeMillis() 取当前的时间，当然你也可以故意设置成别的时间。顺带一提，如果你不想显示时间，把这个设置成 0 就行了（和 number 一样的）。

**contentView(RemoteViews)**

其实上面 setLatestEventInfo 是让系统帮你生成一个标准的通知模板。但是你会发现一些音乐播放器和一些手机助手，在通知栏上有一些比较高级的通知，这些是通过自定义的 content view 来实现的。不要用系统的模板，自己写 content view 就行了，这个是 RemoteViews，和写 widget 差不多的。

**bigContentView(RemoteViews)**

这个是 Notification 新增的 big view 模式，和 contentView 一样，可以自定义，详见下面 Builder 的说明。

**deleteIntent**

上面那个 contentIntent 是点击通知时候的动作。这个顾名思义，是通知被删除的时候的动作，例如说通知被删掉了，可以做一些清除发送通知相关的数据的操作。

**fullScreenIntent**

这个看名字不是太直白（但是也有点沾边），这个是指发送通知的时候做的动作，例如说闹钟来了，对通知栏发送一条通知，顺带 launch 闹钟界面的 activity（系统闹钟的做法）；或者是插入 usb，发送 usb 连接通知，顺带 launch usb storage 操作界面。

**tickerText(Char CharSequence)**

设置前面说的那个 ticker 显示的文本信息。这个也是系统有一个标准模板的，左边一个 icon，右边是文本。模板中 ticker 的 icon 是取 Notification 的 icon（源码中本着封装的原则，不同模块搞了一堆数据结构，又绕了半天，这篇侧重应用，代码这里不分析了）。所以 ticker 图标是和状态栏上的显示的图标是一样的，独立的只能设置文本。如果你没设 tickerText（为 null），那你发通知的时间就没 ticker 显示了（就是没那段通知滚动的动画了）。

**tickerView(RemoteViews)**

看到前面 tickerText 那里提到模板就能猜到 ticker 也是可以自定义的。但是注意一点原生的 android 系统只有 Tablet UI 才允许你自定义 ticker view，因为只有 Tablet UI 才会使用 Notification 中设置的 tickerView。这里顺带说下 android SystemUI 的一些策略问题：

SystemUI 的状态栏（包括虚拟按键、通知面板）以分为好几种 UI 风格，目前已有的是 phone、table、tv。它共同继续只同一个父类 BaseStatusBar。然后实现不同的 UI 风格。在 frameworks/base/package/SystemUI/src/com/android/systemui/statusbar 下面有对应的包。framework 中可以在 xml 中指定要运行的 UI 风格。要想知道自己的设备状态用的是哪一种风格，拿 hierarchyviewer 看下状态栏的包名和类名就行了。

当然这是原生定的，你也可以改成 phone 也可以使用自定义的 tickerView，或者你自己都可以完全搞一套状态栏 UI 风格（只要实现了父类抽象的功能即可）。不过 android 内部也是矛盾不断，4.0 ~ 4.2 的时候原生还是有 phone 和 table 2种风格的（tv 的现在基本上是摆设），到了 4.4 之后 android 就把 table 给干掉了，然后全部只有 phone 的（也许是想统一手机和平板的风格吧）。所以说现在这个接口在原生系统上就是摆设，老老实实用系统标准的模板。

经过上面的介绍，你会发现前面 normal view 下的通知栏，好像还有一个元素（6）没哪个接口可以设置。对的，在直接使用 Notification 设置是没办法设置右边那个小图标的（我不知道这个小图标是不是后面新加入的新元素）。因为这种方法已经被系统标记为过时的（deprecated），新的 sdk 推荐你使用 Notification.Builder 来设置通知。当然系统为了兼容性还保留着以前的接口，你要坚持使用老接口也可以，就是功能没那么全而已。

#### 使用 Notification.Builder

自从 4.0（应该是吧，我就不具体去考查版本了）通知新加入了一种模式，就出现了这个 Builder 的辅助类。我们先来说下新加入的这种通知模式。这模式官方叫 big view。一般通知的表现是在通知面板上一条一条的，虽然可以自定义 content view，但是系统规定死了 normal view 下每一条通知的高度的，所以所有 normal view 的通知的大小都是一样的（这样才好看）。但是你会发现某些音乐播放器的通知栏上的控制工具会比普通的通知要大，这个就是 big view 模式：

big view 高度没有限制（我看代码是，具体没试过），可以比 normal view 高很多（宽是不可能了，屏幕就那么宽），效果见上面的 big view 图。

一个 Notification 可以有 normal view 和 big view。

通知普通情况下显示 normal view，可以通过双指滑动展开 big view（隐藏 normal view），双指缩放关闭 big view（显示 normal view），其实好像不是双指，但是我实在没明白官方怎么个操作法的，但是双指很容易能搞出来。

big view 系统也是提供了若干模板，通过 new 不同的 Notiifcation.Style 可以设置不同风格的 big view 模板，当然也可以自定义（自己设置 bigContentView）。这就不细说 Builder、Style 有哪些接口了，自己去看下官方文档一目了然（其中包括上面说的设置右边的那个小图标的接口）。这里稍微看下系统是在 Builder 和 Style 中怎么帮我们创建 Notification 的 content view 的。

##### 不使用风格

如果直接 new Notification.Builder 然后设置一堆东西，然后调用 builder 的话，会是这样的：

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** Notification build() {
2. **if** (mStyle != **null**) {
3. **return** mStyle.build();
4. } **else** {
5. **return** buildUnstyled();
6. }
7. }

你如果没有调用 setStyle 设置任何风格的话，使用的是 buildUnstyled：

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **private** Notification buildUnstyled() {
2. Notification n = **new** Notification();
3. n.when = mWhen;
4. n.icon = mSmallIcon;
5. n.iconLevel = mSmallIconLevel;
6. n.number = mNumber;
7. // 创建 normal view 的 content view
8. n.contentView = makeContentView();
9. n.contentIntent = mContentIntent;
10. n.deleteIntent = mDeleteIntent;
11. n.fullScreenIntent = mFullScreenIntent;
12. n.tickerText = mTickerText;
13. // 创建 ticker view
14. n.tickerView = makeTickerView();
15. n.largeIcon = mLargeIcon;
16. n.sound = mSound;
17. n.audioStreamType = mAudioStreamType;
18. n.vibrate = mVibrate;
19. n.ledARGB = mLedArgb;
20. n.ledOnMS = mLedOnMs;
21. n.ledOffMS = mLedOffMs;
22. n.defaults = mDefaults;
23. n.flags = mFlags;
24. // 创建 big view 的 content view
25. n.bigContentView = makeBigContentView();
26. **if** (mLedOnMs != 0 && mLedOffMs != 0) {
27. n.flags |= FLAG\_SHOW\_LIGHTS;
28. }
29. **if** ((mDefaults & DEFAULT\_LIGHTS) != 0) {
30. n.flags |= FLAG\_SHOW\_LIGHTS;
31. }
32. **if** (mKindList.size() > 0) {
33. n.kind = **new** String[mKindList.size()];
34. mKindList.toArray(n.kind);
35. } **else** {
36. n.kind = **null**;
37. }
38. n.priority = mPriority;
39. n.extras = mExtras != **null** ? **new** Bundle(mExtras) : **null**;
40. **if** (mActions.size() > 0) {
41. n.actions = **new** Action[mActions.size()];
42. mActions.toArray(n.actions);
43. }
44. **return** n;
45. }

我们稍微看下 normal view 和 big view 的创建：

1. **private** RemoteViews makeContentView() {
2. **if** (mContentView != **null**) {
3. **return** mContentView;
4. } **else** {
5. **return** applyStandardTemplate(R.layout.notification\_template\_base, **true**); // no more special large\_icon flavor
6. }
7. }
8. **private** RemoteViews applyStandardTemplate(**int** resId, **boolean** fitIn1U) {
9. RemoteViews contentView = **new** RemoteViews(mContext.getPackageName(), resId);
10. **boolean** showLine3 = **false**;
11. **boolean** showLine2 = **false**;
12. **int** smallIconImageViewId = R.id.icon;
13. **if** (mLargeIcon != **null**) {
14. contentView.setImageViewBitmap(R.id.icon, mLargeIcon);
15. smallIconImageViewId = R.id.right\_icon;
16. }
17. **if** (mPriority < PRIORITY\_LOW) {
18. contentView.setInt(R.id.icon,
19. "setBackgroundResource", R.drawable.notification\_template\_icon\_low\_bg);
20. contentView.setInt(R.id.status\_bar\_latest\_event\_content,
21. "setBackgroundResource", R.drawable.notification\_bg\_low);
22. }
23. **if** (mSmallIcon != 0) {
24. contentView.setImageViewResource(smallIconImageViewId, mSmallIcon);
25. contentView.setViewVisibility(smallIconImageViewId, View.VISIBLE);
26. } **else** {
27. contentView.setViewVisibility(smallIconImageViewId, View.GONE);
28. }
29. **if** (mContentTitle != **null**) {
30. contentView.setTextViewText(R.id.title, mContentTitle);
31. }
32. **if** (mContentText != **null**) {
33. contentView.setTextViewText(R.id.text, mContentText);
34. showLine3 = **true**;
35. }
36. **if** (mContentInfo != **null**) {
37. contentView.setTextViewText(R.id.info, mContentInfo);
38. contentView.setViewVisibility(R.id.info, View.VISIBLE);
39. showLine3 = **true**;
40. } **else** **if** (mNumber > 0) {
41. **final** **int** tooBig = mContext.getResources().getInteger(
42. R.integer.status\_bar\_notification\_info\_maxnum);
43. **if** (mNumber > tooBig) {
44. contentView.setTextViewText(R.id.info, mContext.getResources().getString(
45. R.string.status\_bar\_notification\_info\_overflow));
46. } **else** {
47. NumberFormat f = NumberFormat.getIntegerInstance();
48. contentView.setTextViewText(R.id.info, f.format(mNumber));
49. }
50. contentView.setViewVisibility(R.id.info, View.VISIBLE);
51. showLine3 = **true**;
52. } **else** {
53. contentView.setViewVisibility(R.id.info, View.GONE);
54. }
55. // Need to show three lines?
56. **if** (mSubText != **null**) {
57. contentView.setTextViewText(R.id.text, mSubText);
58. **if** (mContentText != **null**) {
59. contentView.setTextViewText(R.id.text2, mContentText);
60. contentView.setViewVisibility(R.id.text2, View.VISIBLE);
61. showLine2 = **true**;
62. } **else** {
63. contentView.setViewVisibility(R.id.text2, View.GONE);
64. }
65. } **else** {
66. contentView.setViewVisibility(R.id.text2, View.GONE);
67. **if** (mProgressMax != 0 || mProgressIndeterminate) {
68. contentView.setProgressBar(
69. R.id.progress, mProgressMax, mProgress, mProgressIndeterminate);
70. contentView.setViewVisibility(R.id.progress, View.VISIBLE);
71. showLine2 = **true**;
72. } **else** {
73. contentView.setViewVisibility(R.id.progress, View.GONE);
74. }
75. }
76. **if** (showLine2) {
77. **if** (fitIn1U) {
78. // need to shrink all the type to make sure everything fits
79. **final** Resources res = mContext.getResources();
80. **final** **float** subTextSize = res.getDimensionPixelSize(
81. R.dimen.notification\_subtext\_size);
82. contentView.setTextViewTextSize(R.id.text, TypedValue.COMPLEX\_UNIT\_PX, subTextSize);
83. }
84. // vertical centering
85. contentView.setViewPadding(R.id.line1, 0, 0, 0, 0);
86. }
88. **if** (mWhen != 0 && mShowWhen) {
89. **if** (mUseChronometer) {
90. contentView.setViewVisibility(R.id.chronometer, View.VISIBLE);
91. contentView.setLong(R.id.chronometer, "setBase",
92. mWhen + (SystemClock.elapsedRealtime() - System.currentTimeMillis()));
93. contentView.setBoolean(R.id.chronometer, "setStarted", **true**);
94. } **else** {
95. contentView.setViewVisibility(R.id.time, View.VISIBLE);
96. contentView.setLong(R.id.time, "setTime", mWhen);
97. }
98. } **else** {
99. contentView.setViewVisibility(R.id.time, View.GONE);
100. }
101. contentView.setViewVisibility(R.id.line3, showLine3 ? View.VISIBLE : View.GONE);
102. contentView.setViewVisibility(R.id.overflow\_divider, showLine3 ? View.VISIBLE : View.GONE);
103. **return** contentView;
104. }

具体细节不看了，就是根据 Builder 设置的参数设置 R.layout.notification\_template\_base 中的界面元素。从这里能看出使用 Builder 比上面使用 Notification.setLatestEventInfo 创建的模板多很多东西。所以要想更多的控制自己程序发送的通知样式还是使用 sdk 推荐的接口比较好。

然后看下 big view 的：

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **private** RemoteViews makeBigContentView() {
2. **if** (mActions.size() == 0) **return** **null**;
3. **return** applyStandardTemplateWithActions(R.layout.notification\_template\_big\_base);
4. }
5. **private** RemoteViews applyStandardTemplateWithActions(**int** layoutId) {
6. RemoteViews big = applyStandardTemplate(layoutId, **false**);
7. **int** N = mActions.size();
8. **if** (N > 0) {
9. // Log.d("Notification", "has actions: " + mContentText);
10. big.setViewVisibility(R.id.actions, View.VISIBLE);
11. big.setViewVisibility(R.id.action\_divider, View.VISIBLE);
12. **if** (N>MAX\_ACTION\_BUTTONS) N=MAX\_ACTION\_BUTTONS;
13. big.removeAllViews(R.id.actions);
14. **for** (**int** i=0; i<N; i++) {
15. **final** RemoteViews button = generateActionButton(mActions.get(i));
16. //Log.d("Notification", "adding action " + i + ": " + mActions.get(i).title);
17. big.addView(R.id.actions, button);
18. }
19. }
20. **return** big;
21. }

不设置任何风格的 Builder，并且又没设置任何 Actions，是没有 big view 的。

##### 使用风格

Builder 有一个 setStyle 的接口，可以设置一个风格，系统 Notification 目前提供了下面几种风格：

BigPictureStyle: 就是能设置很大一张的

BigTextStyle: 有很大的空间（高度）显示一大串文本

InboxStyle： 就是上面贴图的那个 big view

我这里偷下懒就不一个一个上效果图了（自己试一下就能看到效果了）。然后这里的风格的区别主要是在 big view 上，这几个风格分别提供了3种 big view 的风格，设置也是设置相应的 big view 的模板元素。我们来稍微看下源码，还记得上面那个 Builder 的 build 函数么，如果 mStyle 不是 null 的话，那么就调用相应 Style 的 build 函数，我们这里稍微看下 BigPictureStyle 的：。。。

从代码中看，这些 Style 就是先调用 Builder 的无风格 build 函数，然后再重新把 bigContentView 自己生成一个，覆盖以前的而已。其他2个我就不贴代码了，套路基本上是一样的。通知栏智能机时代是 android 发明的（IOS 的在后面，android 在界面设计方面有领先 IOS 的地方了哟），加入了 big view 之后 UI 更加灵活了，好像 5.0 又多了一个新功能，以后有时间再看看。

### 系统处理流程

上面说了一些 Notification 的基本用法，也稍微分析下源代码的一些实现。这些稍微说下，应用对 NM 发送一条通知，framework 中会对应生成什么数据，然后通知的 content view 是怎么加到 SystemUI 的状态栏和通知面板中的。先是上一张图先：



从图中可以看到一个通知从 app 自己构造通知然后调用 NotificationManager（NM）的接口发送，经过了 NotificationManagerService（NMS）到 StatusBarManagerService(SBMS) 最后到添加 Notification 的 UI 元素到 SystemUI 中。好像也有点麻烦的样子，然后图中蓝色的部分是每个模块（NMS、SBMS、SystemUI）保存通知的相关数据（前面说了每个模块都不一样，还真是不一样）。然后我们再贴下通知相关的 UI 元素最后在 SystemUI 中的表现：



(PS: 这不是 android 原生的 SystemUI，是被我定制过的，通知没分组了)

上面我们稍微注意下每个模块对应的数据结构，和最后扔到 SystemUI 中的 view 就差不多能摸清系统处理通知的流程了。下面开始跟流程：

#### App

首先是构造 Notification，前面分析过了，然后获取 NM 调用 notify 的接口，把 id 和 Notificaiton 传给 NM。这里代码前面贴过了，回去看看就行。

即，APP通过调用android.app.**NotificationManager**的Notify方法，调用到 com.android.server.notification.**NotificationManagerService**的此**SystemService**的 enqueueNotificationWithTag()方法，再进入到enqueueNotificationInternal()方法。

#### frameworks -NotificationManagerService

然后 NM 会调用 NMS 的 enqueueNotificationInternal（Notification 是 Parcelable 所以可以当作 Binder 的参数传给 NMS）

##### enqueueNotificationInternal

g

1. **public** **void** enqueueNotificationInternal(String pkg, **int** callingUid, **int** callingPid,
2. String tag, **int** id, Notification notification, **int**[] idOut, **int** userId)
3. {
4. **if** (DBG) {
5. Slog.v(TAG, "enqueueNotificationInternal: pkg=" + pkg + " id=" + id + " notification=" + notification);
6. }
7. // 判断是不是当前用户应用发出来的通知
8. checkCallerIsSystemOrSameApp(pkg);
9. // 判断是不是系统应用发出来的通知
10. **final** **boolean** isSystemNotification = isUidSystem(callingUid) || ("android".equals(pkg));
11. **final** **int** userId = ActivityManager.handleIncomingUser(callingPid,
12. callingUid, incomingUserId, **true**, **false**, "enqueueNotification", pkg);
13. **final** UserHandle user = **new** UserHandle(userId);
14. // 如果不是系统应用的话，限制一下一个包发送总共应用的条数，原生限制 50 条
15. // 通过统计 NotificationRecord 的 list 包名一样的通知记录
16. // Limit the number of notifications that any given package except the android
17. // package can enqueue.  Prevents DOS attacks and deals with leaks.
18. **if** (!isSystemNotification) {
19. **synchronized** (mNotificationList) {
20. **int** count = 0;
21. **final** **int** N = mNotificationList.size();
22. **for** (**int** i=0; i<N; i++) {
23. **final** NotificationRecord r = mNotificationList.get(i);
24. **if** (r.sbn.getPackageName().equals(pkg) && r.sbn.getUserId() == userId) {
25. count++;
26. **if** (count >= MAX\_PACKAGE\_NOTIFICATIONS) {
27. Slog.e(TAG, "Package has already posted " + count
28. + " notifications.  Not showing more.  package=" + pkg);
29. **return**;
30. }
31. }
32. }
33. }
34. }
35. // This conditional is a dirty hack to limit the logging done on
36. //     behalf of the download manager without affecting other apps.
37. **if** (!pkg.equals("com.android.providers.downloads")
38. || Log.isLoggable("DownloadManager", Log.VERBOSE)) {
39. EventLog.writeEvent(EventLogTags.NOTIFICATION\_ENQUEUE, pkg, id, tag, userId,
40. notification.toString());
41. }
42. **if** (pkg == **null** || notification == **null**) {
43. **throw** **new** IllegalArgumentException("null not allowed: pkg=" + pkg
44. + " id=" + id + " notification=" + notification);
45. }
46. **if** (notification.icon != 0) {
47. // content view 必须要有
48. **if** (notification.contentView == **null**) {
49. **throw** **new** IllegalArgumentException("contentView required: pkg=" + pkg
50. + " id=" + id + " notification=" + notification);
51. }
52. }
53. // === Scoring ===
54. // 通知有个分数排序的，分数高的排在列表前面，在 SystemUI 的显示通知列表中也靠前，我们以后再分析这个分数怎么算的
55. // 0. Sanitize inputs
56. notification.priority = clamp(notification.priority, Notification.PRIORITY\_MIN, Notification.PRIORITY\_MAX);
57. // Migrate notification flags to scores
58. **if** (0 != (notification.flags & Notification.FLAG\_HIGH\_PRIORITY)) {
59. **if** (notification.priority < Notification.PRIORITY\_MAX) notification.priority = Notification.PRIORITY\_MAX;
60. } **else** **if** (SCORE\_ONGOING\_HIGHER && 0 != (notification.flags & Notification.FLAG\_ONGOING\_EVENT)) {
61. **if** (notification.priority < Notification.PRIORITY\_HIGH) notification.priority = Notification.PRIORITY\_HIGH;
62. }
63. // 1. initial score: buckets of 10, around the app
64. **int** score = notification.priority \* NOTIFICATION\_PRIORITY\_MULTIPLIER; //[-20..20]
65. // 2. Consult external heuristics (TBD)
66. // 3. Apply local rules
67. // 在系统设置中，可以屏蔽掉某一个应用的通知
68. // blocked apps
69. **if** (ENABLE\_BLOCKED\_NOTIFICATIONS && !isSystemNotification && !areNotificationsEnabledForPackageInt(pkg)) {
70. score = JUNK\_SCORE;
71. Slog.e(TAG, "Suppressing notification from package " + pkg + " by user request.");
72. }
74. **if** (DBG) {
75. Slog.v(TAG, "Assigned score=" + score + " to " + notification);
76. }
77. **if** (score < SCORE\_DISPLAY\_THRESHOLD) {
78. // Notification will be blocked because the score is too low.
79. Slog.i(TAG, "Notification=" + notification + " score is low than display threshold, we don't show it.");
80. **return**;
81. }
82. // Should this notification make noise, vibe, or use the LED?
83. **final** **boolean** canInterrupt = (score >= SCORE\_INTERRUPTION\_THRESHOLD);
84. // 喜闻乐见的 SS 业务函数多线程同步锁
85. **synchronized** (mNotificationList) {
86. // 这里用发通知 app的包名、通知的 tag、id、app 的 uid、app 的 pid、
87. // app 的用户 id 和 Notiifcaiton 构造出一个 NotiifcationRecord（nr）
88. NotificationRecord r = **new** NotificationRecord(pkg, tag, id,
89. callingUid, callingPid, userId,
90. score,
91. notification);
92. NotificationRecord old = **null**;
93. // 这里去看之前应用有没有发送过相同的通知（通过前面的标示区分）
94. **int** index = indexOfNotificationLocked(pkg, tag, id, userId);
95. **if** (index < 0) {
96. // 如果是新通知就加入到 NR 列表中
97. mNotificationList.add(r);
98. } **else** {
99. // 如果原来有，取出来原来的，顺带把原来的从 list 删掉，再把新的加进去 -\_-||
100. old = mNotificationList.remove(index);
101. mNotificationList.add(index, r);
102. // Make sure we don't lose the foreground service state.
103. **if** (old != **null**) {
104. notification.flags |=
105. old.notification.flags&Notification.FLAG\_FOREGROUND\_SERVICE;
106. }
107. }
109. // Ensure if this is a foreground service that the proper additional
110. // flags are set.
111. **if** ((notification.flags&Notification.FLAG\_FOREGROUND\_SERVICE) != 0) {
112. notification.flags |= Notification.FLAG\_ONGOING\_EVENT
113. | Notification.FLAG\_NO\_CLEAR;
114. }
115. **final** **int** currentUser;
116. **final** **long** token = Binder.clearCallingIdentity();
117. **try** {
118. currentUser = ActivityManager.getCurrentUser();
119. } **finally** {
120. Binder.restoreCallingIdentity(token);
121. }
122. **if** (notification.icon != 0) {
123. // 通过和构造 NR 差不多的东西构造出一个 StatusBarNotification(sbn)
124. **final** StatusBarNotification n = **new** StatusBarNotification(
125. pkg, id, tag, r.uid, r.initialPid, score, notification, user);
126. // 判断下原来的和新的是不是同一样，通过以前通知保存的 SBMS 返回的 IBinder 来判断
127. // 如果是一样的话那么调用 SBMS 的 updateNotification
128. **if** (old != **null** && old.statusBarKey != **null**) {
129. r.statusBarKey = old.statusBarKey;
130. **long** identity = Binder.clearCallingIdentity();
131. **try** {
132. mStatusBar.updateNotification(r.statusBarKey, n);
133. }
134. **finally** {
135. Binder.restoreCallingIdentity(identity);
136. }
137. } **else** {
138. // 如果是新发送的通知，那么调用 SBMS 的 addNotification 添加通知
139. **long** identity = Binder.clearCallingIdentity();
140. **try** {
141. // NR 保存一下 SBMS 返回的 IBinder(Bp)
142. r.statusBarKey = mStatusBar.addNotification(n);
143. **if** ((n.notification.flags & Notification.FLAG\_SHOW\_LIGHTS) != 0
144. && canInterrupt) {
145. mAttentionLight.pulse();
146. }
147. }
148. **finally** {
149. Binder.restoreCallingIdentity(identity);
150. }
151. }
152. // Send accessibility events only for the current user.
153. **if** (currentUser == userId) {
154. sendAccessibilityEvent(notification, pkg);
155. }
156. } **else** {
157. // 对于没有设置 icon 的通知，当作 cancel 来处理（删掉通知）
158. Slog.e(TAG, "Ignoring notification with icon==0: " + notification);
159. **if** (old != **null** && old.statusBarKey != **null**) {
160. **long** identity = Binder.clearCallingIdentity();
161. **try** {
162. mStatusBar.removeNotification(old.statusBarKey);
163. }
164. **finally** {
165. Binder.restoreCallingIdentity(identity);
166. }
167. }
168. }
169. }
170. idOut[0] = id;
171. }

NMS 处理通知的函数一开始是一些检测性的工作（其实很多 SS 的业务函数一开始都是一些检测工作）。限制每个应用最多只能提交**50个**通知；这是防止恶意软件注册大量通知。

是根据通知重要性，对其进行打分，分数在 **-20到20** 之间；



我们稍微看下 checkCallerIsSystemOrSameApp：

##### checkCallerIsSystemOrSameApp

这里可以看得出系统在多用户方面做的一些限制操作。

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **void** checkCallerIsSystemOrSameApp(String pkg) {
2. // 获取 IPC 调用者（发送通知的应用）的 uid
3. **int** uid = Binder.getCallingUid();
4. // 如果是系统应用（uid 0 是 root 组），那么可以向所有用户发送通知
5. **if** (UserHandle.getAppId(uid) == Process.SYSTEM\_UID || uid == 0) {
6. **return**;
7. }
8. **try** {
9. // 如果发送通知的应用（普通应用）的 uid 不属于当前用户，那么不允许发送
10. ApplicationInfo ai = AppGlobals.getPackageManager().getApplicationInfo(
11. pkg, 0, UserHandle.getCallingUserId());
12. **if** (!UserHandle.isSameApp(ai.uid, uid)) {
13. **throw** **new** SecurityException("Calling uid " + uid + " gave package"
14. + pkg + " which is owned by uid " + ai.uid);
15. }
16. } **catch** (RemoteException re) {
17. **throw** **new** SecurityException("Unknown package " + pkg + "\n" + re);
18. }
19. }

##### NotificationRecrod

NMS 这里使用的数据结构是 NotificationRecrod，然后 NMS 是用一个 ArrayList 来保存的:

private final ArrayList<NotificationRecord> mNotificationList =

new ArrayList<NotificationRecord>();

private static final class NotificationRecord

{

final String pkg;

final String tag;

final int id;

final int uid;

final int initialPid;

final int userId;

final Notification notification;

final int score;

IBinder statusBarKey;

... ...

这里会判断一下之前应用有没有发送相同标志的通知（indexOfNotificationLocked 前面有贴代码），如果有的话从 NR list 中删掉原来的，再插入新的，如果没有的话直接插入新的。然后再构造出一个 StatusBarNotifcation。注意一下这个数据结构虽然在 NMS 中 new 了出来，但是是在 SBMS 中使用的，这里只是为了传递给 SBMS 而已（SBMS 提供的接口的参数是 sbn），所以 NMS 中只是持有 NR 的数据而已，sbn 在这里这是临时数据。然后根据 indexOfNotificationLocked 的结果是否已经存在 NR 记录会调用 SBMS 不同的接口，如果有会调用 updateNotification，新通知的话会调用 addNotification。这里我们先以新通知来分析，所以到这里就到 SBMS 中了去。细心的会发现在我流程图中 NMS 到 SBMS 那里没有标 IPC 而是标了 Bn（表示本地）。这是因为 NMS 和 SBMS 都是在 SS（SystemServer）进程中的（忘记了的去 Binder 篇复习下），所以它之间可以直接持有对方的对象直接调用相关的接口，无需跨进程。同时 SBMS 提供的 IPC 接口只是占本身接口的一小部分的（aidl 中的），这里调用的接口是没在 aidl 中申明的，所以别的进程只能使用 SMBS 很有限的一部分功能。可以说这里 NMS 转到 SBMS 属于 SS 内部的功能。

#### StatusBarManagerService

现在从 NMS 转到 SBMS 中的 addNotification 中了：

这个函数非常简单，因为这里的 SBMS 其实只是起到一个桥接作用，大部分工作在提供了 IStatusBar 接口的 SystemUI 中。不过虽然简单这里 SBMS 还是持有了一个数据结构：StatusBarNotification：

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **class** StatusBarManagerService **extends** IStatusBarService.Stub
2. **implements** WindowManagerService.OnHardKeyboardStatusChangeListener
3. {
4. **volatile** IStatusBar mBar;
5. // 保存数据结构的是一个 HashMap
6. HashMap<IBinder,StatusBarNotification> mNotifications
7. = **new** HashMap<IBinder,StatusBarNotification>();
8. **public** IBinder addNotification(StatusBarNotification notification) {
9. **synchronized** (mNotifications) {
10. // 生成 NotificationRecord 的 Binder 对象
11. IBinder key = **new** Binder();
12. mNotifications.put(key, notification);
13. **if** (mBar != **null**) {
14. **try** {
15. mBar.addNotification(key, notification);
16. } **catch** (RemoteException ex) {
17. }
18. }
19. **return** key;
20. }
21. }
22. }
23. /\*
24. \* Class encapsulating a Notification. Sent by the NotificationManagerService to the IStatusBar (in System UI).
25. \*/
26. **public** **class** StatusBarNotification **implements** Parcelable {
27. **public** **final** String pkg;
28. **public** **final** **int** id;
29. **public** **final** String tag;
30. **public** **final** **int** uid;
31. **public** **final** **int** initialPid;
32. // TODO: make this field private and move callers to an accessor that
33. // ensures sourceUser is applied.
34. **public** **final** Notification notification;
35. **public** **final** **int** score;
36. **public** **final** UserHandle user;
37. }

可以看到这里 StatusBarNotification 也是支持 Parcelable，不过构造的地方是在 NMS 中（前面 NMS 那里传过来的）。然后 SBMS 保存 sbn 的是一个 HashMap，以这条新通知的 IBinder 对象作为 key。这里的 IBinder 在 SBMS 这里本地生成，所以是 Bn，所以通知的 IBinder 对象在 SS 中都是 Bn 来的。这个 key 被返回给 NMS 同时保存在这条记录对应的 NotificationRecord 的 statusBarKey 中。到后面 SystemUI 中标示通知 view 相关对象的时候也是拿一个 IBinder 对象区分的。

然后直接调用 SBMS 的 IstatusBar 对象的 addNotification 函数去 SystemUI 中去处理通知 UI 表现相关的东西去了。

#### **SystemUI**

上面说到 SBMS 有一个 IStatusBar 对象。前面说了 SystemUI 的多 UI 风格，这里就是抽象地方表现之一。SystemUI 抽象了一个状态栏的抽象基类： BaseStatusBar 然后定义了一系列状态栏的功能接口，只要实现了这些接口，那么可以表现出不同的 UI 风格（”小屏”手机，大屏平板，超大屏电视等）。其中抽象出一个 IStatusBar 的 IBinder 接口提供 SS 中的服务调用 SystemUI 相关的接口，让 SystemUI 在 UI 上展现一条通知。然后 BaseStatusBar 的具体子类只要能满足 SS 中（具体是 SBMS）的接口需求就行。所以 BaseStatusBar 中在状态栏初始化的时候会向 SBMS 注册当前实现 IStatusBar 接口的对象（当前使用的 UI 风格）：具体分析见

### 设计思想

mAm，mIAm；mPm，mIPm区别在哪里？

#### SystemUI

是一个抽象类，定义了模板和相关抽象方法：成员变量mComponents

生命周期监听方法onConfigurationChanged和onBootCompleted，组件map表mComponents

管理

#### SystemUIApplication.mComponents

在SystemUIApplication new 出这个map，在SystemUIApplication.startServicesIfNeeded：

* + 该map地址赋给了mServices[i]对象的mComponents
  + mServices[i]在start方法调用putComponent(Recents.class, this);

通过上述方法，SystemUIApplication.mComponents == mServices[i].mComponents

Q为何putComponent是放在mServices[i]在start调用，而不是直接在SystemUIApplication，

A：mComponents可能不同，比如VolumeUI里面的，实现服务和组件分离



#### RecentsDebugFlags

集中管理所有调试开关

定义黑名单

*<!-- Fully qualified activity class names to be blacklisted in Recents, add package names into overlay as needed -->*<**string-array name="recents\_blacklist\_array"**>  
</**string-array**>

#### SystemServicesProxy

SystemBars服务启动分析

SBMS IPC 调用 addNotification 之后把通知的 IBinder key 和 sbn 传了过来，我们来看下 PhoneStatusBar 中的具体实现

##### PhoneStatusBar.addNotification

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **void** addNotification(IBinder key, StatusBarNotification notification) {
2. // 大部分工作在这里
3. StatusBarIconView iconView = addNotificationViews(key, notification);
4. **if** (iconView == **null**) **return**;
5. **boolean** immersive = **false**;
6. **try** {
7. immersive = ActivityManagerNative.getDefault().isTopActivityImmersive();
8. **if** (DEBUG) {
9. Slog.d(TAG, "Top activity is " + (immersive?"immersive":"not immersive"));
10. }
11. } **catch** (RemoteException ex) {}
12. // 如果通知设置了 fullScreenIntent，执行 fullScreenIntent 中设置的动作
13. //（启动相应的 activity 或是发送相应的广播）
14. **if** (notification.notification.fullScreenIntent != **null**) {
15. // Stop screensaver if the notification has a full-screen intent.
16. // (like an incoming phone call)
17. awakenDreams();
18. // not immersive & a full-screen alert should be shown
19. **if** (DEBUG) Slog.d(TAG, "Notification has fullScreenIntent; sending fullScreenIntent");
20. **try** {
21. notification.notification.fullScreenIntent.send();
22. } **catch** (PendingIntent.CanceledException e) {
23. }
24. } **else** {
25. // usual case: status bar visible & not immersive
26. // 如果没指定 fullScreenIntent 那么在状态栏上展现一下 ticker 动画
27. // show the ticker if there isn't an intruder too
28. **if** (mCurrentlyIntrudingNotification == **null**) {
29. tick(**null**, notification, **true**);
30. }
31. }
32. // Recalculate the position of the sliding windows and the titles.
33. setAreThereNotifications();
34. updateExpandedViewPos(EXPANDED\_LEAVE\_ALONE);
35. }

##### BaseStatusBar.addNotificationViews

这里一开始就调用了基类 BaseStatusBar 中的 addNotificationViews 函数，这个函数就属于状态栏的共用函数之一，通知在 SystemUI 上的 UI 元素基本上都由这个函数生成：

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **protected** StatusBarIconView addNotificationViews(IBinder key,
2. StatusBarNotification notification) {
3. // 构造状态栏小图标
4. **final** StatusBarIconView iconView = **new** StatusBarIconView(mContext,
5. notification.pkg + "/0x" + Integer.toHexString(notification.id),
6. notification.notification);
7. iconView.setScaleType(ImageView.ScaleType.CENTER\_INSIDE);
9. // 构造状态栏小图标对应数据 -\_-||
10. **final** StatusBarIcon ic = **new** StatusBarIcon(notification.pkg,
11. notification.user,
12. notification.notification.icon,
13. notification.notification.iconLevel,
14. notification.notification.number,
15. notification.notification.tickerText);
16. // 把数据设置给状态栏小图标
17. **if** (!iconView.set(ic)) {
18. handleNotificationError(key, notification, "Couldn't create icon: " + ic);
19. **return** **null**;
20. }
21. // 构造 SystemUI 通知数据结构
22. // Construct the expanded view.
23. NotificationData.Entry entry = **new** NotificationData.Entry(key, notification, iconView);
24. // 构造通知面板上的通知 view，并将其加入通知面板上
25. **if** (!**inflateViews**(entry, mPile)) {
26. handleNotificationError(key, notification, "Couldn't expand RemoteViews for: "
27. + notification);
28. **return** **null**;
29. }
31. // 保存通知数据
32. // Add the expanded view and icon.
33. **int** pos = mNotificationData.add(entry);
34. updateExpansionStates();
35. // 更新状态栏上的通知小图标（新的加入到状态栏上去）
36. updateNotificationIcons();
37. **return** iconView;
38. }

##### StatusBarIconView

这里 StatusBarIconView 别看是继承了 AnimatedImageView，其实最后 AnimatedImageView 最后是继承了 ImageView，也就是说状态栏上左上角那一排通知的小图标就是一堆 ImageView（右边那一排也是一样的，不过那一排叫 Status Icon，系统状态图标，和通知小图标是不一样的东西，刚开始容易搞混，这个东西后面我单独开一篇来说）。然后后面这里就是 framework 中最后一个模块 SystemUI 中持有的通知的数据结构了 NotificationData：

##### NotificationData

G

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. /\* The list of currently displaying notifications.
2. \*/
3. **public** **class** NotificationData {
4. **public** **static** **final** **class** Entry {
5. **public** IBinder key;
6. **public** StatusBarNotification notification;
7. **public** StatusBarIconView icon;
8. **public** View row; // the outer expanded view
9. **public** View content; // takes the click events and sends the PendingIntent
10. **public** View expanded; // the inflated RemoteViews
11. **public** ImageView largeIcon;
12. **protected** View expandedLarge;
14. **public** Entry() {
15. }
17. **public** Entry(IBinder key, StatusBarNotification n, StatusBarIconView ic) {
18. **this**.key = key;
19. **this**.notification = n;
20. **this**.icon = ic;
21. }
23. // 保存的数据的最后是一个 ArrayList
24. **private** **final** ArrayList<Entry> mEntries = **new** ArrayList<Entry>();
25. **private** **final** Comparator<Entry> mEntryCmp = **new** Comparator<Entry>() {
26. // sort first by score, then by when
27. **public** **int** compare(Entry a, Entry b) {
28. **final** StatusBarNotification na = a.notification;
29. **final** StatusBarNotification nb = b.notification;
30. **int** d = na.score - nb.score;
31. **return** (d != 0)
32. ? d
33. : (**int**) (na.notification.when - nb.notification.when);
34. }
35. };
37. // 它的大多数接口都是封装对上面那个 ArrayList 的操作
38. **public** **int** add(Entry entry) {
39. **int** i;
40. **int** N = mEntries.size();
41. **for** (i = 0; i < N; i++) {
42. **if** (mEntryCmp.compare(mEntries.get(i), entry) > 0) {
43. **break**;
44. }
45. }
46. mEntries.add(i, entry);
47. **return** i;
48. }
49. }
50. }

这个数据结构内部还有一个 Entry 类，可以看到 Entry 中不光有 sbn，还有好几个 view，这几个 view 就是 SystemUI 中一个通知的 UI 元素了。然后它的插入啊，删除啊都通过一个 Entry 的 ArrayList 来实现的。回到 addNotificationViews 中，后面有一个 inflateViews

##### BaseStatusBar.inflateViews

**D**

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **protected**  **boolean** inflateViews(NotificationData.Entry entry, ViewGroup parent) {
2. **int** minHeight =
3. mContext.getResources().getDimensionPixelSize(R.dimen.notification\_min\_height);
4. **int** maxHeight =
5. mContext.getResources().getDimensionPixelSize(R.dimen.notification\_max\_height);
6. StatusBarNotification sbn = entry.notification;
7. // oneU 是通知 normal view
8. // large 是通知 big view
9. RemoteViews oneU = sbn.notification.contentView;
10. RemoteViews large = sbn.notification.bigContentView;
11. **if** (oneU == **null**) {
12. **return** **false**;
13. }
14. // create the row view
15. LayoutInflater inflater = (LayoutInflater)mContext.getSystemService(
16. Context.LAYOUT\_INFLATER\_SERVICE);
17. // new 通知的容器（notiifcation row），这个容器是模板来的。
18. // 这个是用来装通知的 content view 和 big content view 的。
19. // 这里注意一点：LayoutInflater 的第二参数 ViewGroup 如果不是 null 的话，那么 new 出来的 view
20. // 会自动 add 到传递的 ViewGroup 中，所以这里在 new 出 notification row 后，
21. // 就自动添加到前面传递过来的 mPile 中去了（这个是通知面板中显示通知的容器）。
22. View row = inflater.inflate(R.layout.status\_bar\_notification\_row, parent, **false**);
23. // for blaming (see SwipeHelper.setLongPressListener)
24. row.setTag(sbn.pkg);
25. workAroundBadLayerDrawableOpacity(row);
26. View vetoButton = updateNotificationVetoButton(row, sbn);
27. vetoButton.setContentDescription(mContext.getString(
28. R.string.accessibility\_remove\_notification));
29. // NB: the large icon is now handled entirely by the template
30. // bind the click event to the content area
31. ViewGroup content = (ViewGroup)row.findViewById(R.id.content);
32. ViewGroup adaptive = (ViewGroup)row.findViewById(R.id.adaptive);
33. content.setDescendantFocusability(ViewGroup.FOCUS\_BLOCK\_DESCENDANTS);
34. // 如果通知设置了点击动作，那么设置到通知的 view OnClick 时间中去
35. PendingIntent contentIntent = sbn.notification.contentIntent;
36. **if** (contentIntent != **null**) {
37. **final** View.OnClickListener listener = **new** NotificationClicker(contentIntent,
38. sbn.pkg, sbn.tag, sbn.id);
39. content.setOnClickListener(listener);
40. } **else** {
41. content.setOnClickListener(**null**);
42. }
43. // TODO(cwren) normalize variable names with those in updateNotification
44. View expandedOneU = **null**;
45. View expandedLarge = **null**;
46. **try** {
47. // 把通知的 content view（RemoteViews）加入到 notification row
48. expandedOneU = oneU.apply(mContext, adaptive, mOnClickHandler);
49. // 如果通知有 big content view（RemoteViews） 也一起加入 notification row
50. **if** (large != **null**) {
51. expandedLarge = large.apply(mContext, adaptive, mOnClickHandler);
52. }
53. }
54. **catch** (RuntimeException e) {
55. **final** String ident = sbn.pkg + "/0x" + Integer.toHexString(sbn.id);
56. Slog.e(TAG, "couldn't inflate view for notification " + ident, e);
57. **return** **false**;
58. }
59. **if** (expandedOneU != **null**) {
60. SizeAdaptiveLayout.LayoutParams params =
61. **new** SizeAdaptiveLayout.LayoutParams(expandedOneU.getLayoutParams());
62. params.minHeight = minHeight;
63. params.maxHeight = minHeight;
64. adaptive.addView(expandedOneU, params);
65. }
66. **if** (expandedLarge != **null**) {
67. SizeAdaptiveLayout.LayoutParams params =
68. **new** SizeAdaptiveLayout.LayoutParams(expandedLarge.getLayoutParams());
69. params.minHeight = minHeight+1;
70. params.maxHeight = maxHeight;
71. adaptive.addView(expandedLarge, params);
72. }
73. row.setDrawingCacheEnabled(**true**);
74. applyLegacyRowBackground(sbn, content);
75. row.setTag(R.id.expandable\_tag, Boolean.valueOf(large != **null**));
76. **if** (MULTIUSER\_DEBUG) {
77. TextView debug = (TextView) row.findViewById(R.id.debug\_info);
78. **if** (debug != **null**) {
79. debug.setVisibility(View.VISIBLE);
80. debug.setText("U " + entry.notification.getUserId());
81. }
82. }
83. // 设置一下 SystemUI 的通知数据
84. entry.row = row;
85. entry.content = content;
86. entry.expanded = expandedOneU;
87. entry.setLargeView(expandedLarge);
88. **return** **true**;
89. }

要讲解的我都注释在代码里面了，这里就是稍微注意下 LayoutInflater.inflate 那里会自动把生成的 view 添加到通知面板上的通知容器中，如果第一次不注意会觉得奇怪，不知道在哪里把 notification row 加到通知面板中去的。可以看到其实 BaseStatusBar 又给通知做了一个模板套套，然后才是把应用设置（其实大多时候也是用 Notification 生成的模板）的通知的 content view 加到这个模板套套中，因为这个可以控制每一个 notification row 的大小。这样可以由系统控制最终通知显示的 UI 效果（notification row 的最后大小是由 SystemUI 决定的）。

这样通知面板上的 notification row 就弄好了，然后回到 addNotificationViews 中最后 updateNotificationIcons ，这个虽然不是 IStatusBar 的接口，但是 BaseStatusBar 中的抽象接口，留给子类实现的，BaseStatusBar 有好几这样的抽象几个，基本都是 UI 相关的（毕竟子类要实现不同的 UI 风格么）：

##### PhoneStatusBar .updateNotificationIcons

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. @Override
2. **protected** **void** updateNotificationIcons() {
3. **if** (mNotificationIcons == **null**) **return**;
4. loadNotificationShade();
5. **final** LinearLayout.LayoutParams params
6. = **new** LinearLayout.LayoutParams(mIconSize + 2\*mIconHPadding, mNaturalBarHeight);
7. **int** N = mNotificationData.size();
8. ArrayList<View> toShow = **new** ArrayList<View>();
9. **final** **boolean** provisioned = isDeviceProvisioned();
10. // If the device hasn't been through Setup, we only show system notifications
11. **for** (**int** i=0; i<N; i++) {
12. Entry ent = mNotificationData.get(N-i-1);
13. **if** (!((provisioned && ent.notification.score >= HIDE\_ICONS\_BELOW\_SCORE)
14. || showNotificationEvenIfUnprovisioned(ent.notification))) **continue**;
15. **if** (!notificationIsForCurrentUser(ent.notification)) **continue**;
16. toShow.add(ent.icon);
17. }
18. ArrayList<View> toRemove = **new** ArrayList<View>();
19. **for** (**int** i=0; i<mNotificationIcons.getChildCount(); i++) {
20. View child = mNotificationIcons.getChildAt(i);
21. **if** (!toShow.contains(child)) {
22. toRemove.add(child);
23. }
24. }
25. **for** (View remove : toRemove) {
26. mNotificationIcons.removeView(remove);
27. }
28. **for** (**int** i=0; i<toShow.size(); i++) {
29. View v = toShow.get(i);
30. **if** (v.getParent() == **null**) {
31. mNotificationIcons.addView(v, i, params);
32. }
33. }
34. }

PhoneStatusBar 中的 mNotificationIcons 是一个叫 IconMerger 的东西，这个是继承自 LinearLayout 的一个自定义的布局。叫 IconMerger 是因为它有一个功能，当通知很多（状态栏的小图标很多）的情况下，它会把显示不下的图标合并显示成一个类似 “+” 号的图标表示显示不下了。这里我们就不去细看了。然后这里也不用细说什么就是玩 ViewGroup（mNotificationIcons）添加（删除）子 view（StatusBarIconView） 而已。然后到这里 BaseStatusBar 中的 addNotificationViews 就处理完了。然后最后回到 PhoneStatusBar 的 addNotification 最后那里，如果通知设置了 fullScreenIntent 就执行相应的操作，否则展现 ticker 动画效果（效果见前面的效果的那个 ticker view）

#### 小结

到这里一条通知的发送流程就走完了。然后这里是以发送一条新通知来说的，前面看到系统有判断如果发送的这条通知前面已经在系统中存在了，那么就会更新对应的数据（NMS，SBMS中）和对应的 SystemUI 中的 UI 元素（会去通知容器中去找），鉴于这篇已经很长了，后面单独开一篇来说更新的事吧。

然后最后总结一下：这里涉及到系统里面的3个模块：NMS，SBMS 和 SystemUI。其中 NMS 直接是管理通知服务的，SBMS 是界面（SystemUI）系统功能（通知等）桥接，应用通过系统功能的接口（例如 NMS）使用系统提供的一系列 UI 接口。然后这些系统接口再通过系统界面的桥接（SBMS）让界面系统（SystemUI）展现相关 UI 元素（视图和控制分工明确，可以学习一下 android 的设计）。最后我们来列下相关模块的对应的数据结构（第一次看还是有点晕的）：

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. App                        --> Notification
2. NotificationManagerService --> mNotificationList (ArrayList)
3. StatusBarManagerService    --> mNotifications (HashMap)
4. SystemUI                   --> mNotificationData(NotificationData.Entry[ArrayList])
5. |--> StatusBarIconView(StatusBarIcon)
6. |--> notification row
7. |--> Ticker

##### 小技巧

看完上面流程分析，大家应该会发现默认状态栏的小图标和通知 content view 那个显示的图标都是用 Notification 的 icon 的，就是默认是一样的。但是有些时候想让它们不一样，没有没办法咧，仔细看上面的代码会发现方法是用的。因为 content view 在 Notification build 那里创建，而且 SystemUI 状态栏上的 StatusBarIconView 是在 SystemUI 这边创建的（NM 发送 notify 后），那么我们就有曲线救国的方法了：那就是在构造 Notification 的 content view 前把 icon 设置为 content view 想要的图标（如果不使用 Builder 就是在调用 setLatestEventInfo 前，如果使用 Builder 的话在 Builder 的接口中设），然后再改为状态栏想要的图标，最后再调用 notify 就行咯（大家看到我贴的效果图的这2个图标是不一样的了没，这里我可是没改系统实现的哦）。哎，还是上下代码比较直接：

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. NotificationManager nm = (NotificationManager)getSystemService(
2. Context.NOTIFICATION\_SERVICE);
3. Notification n = **new** Notification(
4. // 这里先设置通知面板那通知要显示的图标
5. R.drawable.stat\_sys\_data\_usb,
6. label, System.currentTimeMillis());
7. // 然后构造 content view 设置 notification row 那的图标
8. n.setLatestEventInfo(**this**, label,
9. "No. " + ID + ": info This is just for test notification bar info",
10. intent);
11. // 然后马上把 icon 改成状态栏想要显示的小图标
12. n.icon = R.drawable.stat\_sys\_data\_usb\_small;
13. // 最后用 NM 发送通知，这样在 SystemUI 那生成的 StatusBarIcon 就和通知面板上的图标不一样了
14. nm.notify(ID, n);

### Headup消息

Notification开始支持Heads Up模式。该模式下，不需要下拉通知栏就直接显示出来 ，悬挂在屏幕上方，并且焦点不变，仍在用户操作的界面。因此，不会打断用户的操作，通知过几秒就会自动消失。最典型的应用是，微信的聊天消息通知。网络上对于这种模式的叫法很多，类似横幅通知，悬挂式通知等。

从Android 5.0开始，如果notification priority设置为HIGH, MAX, 或者fullscreenIntent不为空，在非锁屏界面收到notification时屏幕上方会显示一个小悬浮窗口提醒用户，方便用户在不退出当前浏览界面的前提下快速响应该notification，即Heads-Up Notification(简称HUN)。如下图



#### 第三方app

网上对于实现效果众说纷纭，大多倾向于调用NotificationCompat.Builder的方法

public Builder setFullScreenIntent(PendingIntent intent, boolean highPriority)

但在不同机器上，似乎实现效果也不尽相同。   
在华为Mate8（Android 7.0）上，调用setFullScreenIntent反而无法实现Heads Up。   
最终注掉setFullScreenIntent。核心代码如下。

Notification.Builder builder = new Notification.Builder(this);

Intent mIntent = new Intent();

PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, mIntent, 0);

builder.setContentIntent(pendingIntent);//设置pendingIntent

builder.setSmallIcon(R.drawable.foldleft);

builder.setLargeIcon(BitmapFactory.decodeResource(getResources(), R.drawable.lanucher));

builder.setAutoCancel(true);

builder.setContentTitle("悬挂式通知");

builder.setPriority(Notification.PRIORITY\_MAX);//设置最高权限

builder.setDefaults(Notification.DEFAULT\_ALL);//设置声音和震动

notificationManager.notify(2, builder.build());

使用Heads Up Notification也有***必要***前提条件。   
（1）Android API >=21，即5.0以上   
（2）需要在设置中开启横幅通知权限（在设置通知管理中，不同机型可能存在差异）   
（3）***需要设置震动或声音。（必须！）***   
（4）***需要设置Notification为最高权限。（必须！）***

Android对于Notification的设计，个人感觉比较乱。在3.0前后的实现方式不同，且存在很多隐藏的必要条件未说明，可能是需要向前兼容的需求吧。

比如，普通的Notification有大小图标的设置。若不设置大图标，系统会自动使用APP图标，但如果不设置小图标，Notificatin将无法显示。

显示一个消息

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. PendingIntent pi = PendingIntent.getActivity(getContext(), 0, **new** Intent(getContext(), Qs.**class**),
2. PendingIntent.FLAG\_CANCEL\_CURRENT);
4. Notification notification=**new** NotificationCompat.Builder(getContext())
5. .setVisibility(Notification.VISIBILITY\_PUBLIC)
6. .setSmallIcon(R.drawable.ic\_launcher)
7. .setFullScreenIntent(pi, **false**)
8. .setContentTitle("这是标题")
9. .setContentText("这是内容")
10. .build();
11. NotificationManager mNM = (NotificationManager) getContext()
12. .getSystemService(Context.NOTIFICATION\_SERVICE);// NotificationManager 是一个系统Service，必须通过 getSystemService()方法来获取。
13. mNM .notify(1, notification);

其中 Builder.setFullScreenIntent(pendingIntent, false) 是关键; setUsesChronometer (boolean ) 设置这个为 true 以后 heads-up会将action 也显示出来.

#### Systemui处理

D

addNotification

#### 禁止某个应用notification以HUN方式显示

如果需要禁止某个应用notification以HUN方式显示，又不想降低notification或者拿掉fullscreenIntent,毕竟这两种方式会改变notification其他行为

如需禁止某个应用notification以HUN方式显示，可在baseStatusbar.java中根据需求客制化interrupt值。

例如，要禁止来电以Heads-Up Notification方式显示，可修改如下

# Recent

从Android 5.0开始，Recents就有了诸多变化，比如：Android4.4 在SystemUI中并没有Recents文件结构。

Recents界面是个Activity，并且属于SystemUI,既然是属于SystemUI,那就是说Recents界面正常情况下都是在手机开机进入系统时就开始初始化。事实上也确实如此，在手机开机进入锁屏或者Launcher界面以后，触发Recents界面并不会执行onCreate()方法，只是执行onStart()方法。可见通常情况下RecentsActivity被隐藏起来了，这点，我刚开始时候也有点颠覆了我对activity的认知，不过经过后面看代码也就释然了，RecentsActivity是一个比较特殊的activity,framework层会对它做特殊处理：RecentsActivity在Manifest文件中是这么声明的：

**[html]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. <!-- Alternate Recents -->
2. **<activity** android:name=".recents.RecentsActivity"
3. android:label="@string/accessibility\_desc\_recent\_apps"
4. android:launchMode="singleInstance"
5. android:excludeFromRecents="true"
6. android:stateNotNeeded="true"
7. android:resumeWhilePausing="true"
8. android:theme="@style/config\_recents\_activity\_theme"**>**
9. **<intent-filter>**
10. **<action** android:name="com.android.systemui.recents.TOGGLE\_RECENTS" **/>**
11. **</intent-filter>**
12. **</activity>**

### 代码架构说明

根目录：RecentsView

一个任务视图(TaskView)包含缩略图（TaskViewThumbnail@task\_view\_thumbnail）和头部（TaskViewHeader@layout/recents\_task\_view\_header）

数据

一个缩略图数据(ThumbnailData)

### 开机启动Recents服务

同**SystemBars**类似，**Recents的服务是**com.android.systemui.recent.Recents

#### android:excludeFromRecents="true"

 RecentsActivity是如何被隐藏的？隐藏的时候会调用PhoneStatusBar.java中的onVisibilityChanged()方法

**[java]** [view plain](http://blog.csdn.net/u013656135/article/details/49686425) [copy](http://blog.csdn.net/u013656135/article/details/49686425)

1. @Override
2. **public** **void** onVisibilityChanged(**boolean** visible) {
3. // Update the recents visibility flag
4. **if** (visible) {
5. mSystemUiVisibility |= View.RECENT\_APPS\_VISIBLE;
6. } **else** {
7. mSystemUiVisibility &= ~View.RECENT\_APPS\_VISIBLE;
8. }
9. notifyUiVisibilityChanged(mSystemUiVisibility);
10. }

通过View中的一个FLAG来控制它的visibility。不光如此，ActivityManagerService也会对它做特殊处理，管理Recents，大部分工作都是ActivityManagerService管理activity的TASK。ActivityRecord会对它做判断,与excludeFromRecents 属性相对应。

**[java]** [view plain](http://blog.csdn.net/u013656135/article/details/49686425) [copy](http://blog.csdn.net/u013656135/article/details/49686425)

1. **boolean** isRecentsActivity() {
2. **return** mActivityType == RECENTS\_ACTIVITY\_TYPE;
3. }

### 物理键触发Recents流程



上面的流程图可能显得比较粗糙，但是大致可以看出触发Recents时先PreLoadRecents,完成以还要执行toggleRecentApps()，这个的流程跟上面图中基本一样。

都是通过AIDL远程通信，在SystemUI这边通过接口，实现对应的回调来完成。toggleRecentApps()最终会走到AlternateRecentsComponent.java中的toggleRecentsActivity()->startRecentsActivity()->startAlternateRecentsActivity()方法。

关于RecentsActivity先就了解这些，下面跟一下触发Recents的流程，流程用一张简约的流程图来描述可能更加直观一些。既然是长按HOME键来触发Recents，那么肯定是从framework层按键消息分发开始-PhoneWindowManager.java->interceptKeyBeforeDispatching();

##### startAlternateRecentsActivity

**[java]** [view plain](http://blog.csdn.net/u013656135/article/details/49686425) [copy](http://blog.csdn.net/u013656135/article/details/49686425)

1. ...
2. **final** **static** String sToggleRecentsAction = "com.android.systemui.recents.SHOW\_RECENTS";
3. **public** **final** **static** String sRecentsPackage = "com.android.systemui";
4. **public** **final** **static** String sRecentsActivity = "com.android.systemui.recents.RecentsActivity";
5. ...
6. /\*\* Starts the recents activity \*/
7. **void** startAlternateRecentsActivity(ActivityManager.RunningTaskInfo topTask,
8. ActivityOptions opts, **boolean** fromHome, **boolean** fromSearchHome, **boolean** fromThumbnail,
9. TaskStackViewLayoutAlgorithm.VisibilityReport vr) {
10. // Update the configuration based on the launch options
11. mConfig.launchedFromHome = fromSearchHome || fromHome;
12. mConfig.launchedFromSearchHome = fromSearchHome;
13. mConfig.launchedFromAppWithThumbnail = fromThumbnail;
14. mConfig.launchedToTaskId = (topTask != **null**) ? topTask.id : -1;
15. mConfig.launchedWithAltTab = mTriggeredFromAltTab;
16. mConfig.launchedReuseTaskStackViews = mCanReuseTaskStackViews;
17. mConfig.launchedNumVisibleTasks = vr.numVisibleTasks;
18. mConfig.launchedNumVisibleThumbnails = vr.numVisibleThumbnails;
19. mConfig.launchedHasConfigurationChanged = **false**;
20. Intent intent = **new** Intent(sToggleRecentsAction);
21. intent.setClassName(sRecentsPackage, sRecentsActivity);
22. intent.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK
23. | Intent.FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS
24. | Intent.FLAG\_ACTIVITY\_TASK\_ON\_HOME);
25. **if** (opts != **null**) {
26. mContext.startActivityAsUser(intent, opts.toBundle(), UserHandle.CURRENT);
27. } **else** {
28. mContext.startActivityAsUser(intent, UserHandle.CURRENT);
29. }
30. mCanReuseTaskStackViews = **true**;
31. }

启动RecentsActivity,执行的是RecentsActivity的onStart()方法，其中比较重点是update Recents task:

##### updateRecentsTasks

asdfa s

sadf

1. /\*\* Updates the set of recent tasks \*/
2. **void** updateRecentsTasks(Intent launchIntent) {
3. // If AlternateRecentsComponent has preloaded a load plan, then use that to prevent
4. // reconstructing the task stack
5. RecentsTaskLoader loader = RecentsTaskLoader.getInstance();
6. RecentsTaskLoadPlan plan = AlternateRecentsComponent.consumeInstanceLoadPlan();
7. **if** (plan == **null**) {
8. plan = loader.createLoadPlan(**this**);
9. }
10. // Start loading tasks according to the load plan
11. **if** (plan.getTaskStack() == **null**) {
12. loader.preloadTasks(plan, mConfig.launchedFromHome);
13. }
14. RecentsTaskLoadPlan.Options loadOpts = **new** RecentsTaskLoadPlan.Options();
15. loadOpts.runningTaskId = mConfig.launchedToTaskId;
16. loadOpts.numVisibleTasks = mConfig.launchedNumVisibleTasks;
17. loadOpts.numVisibleTaskThumbnails = mConfig.launchedNumVisibleThumbnails;
18. loader.loadTasks(**this**, plan, loadOpts);
19. SpaceNode root = plan.getSpaceNode();
20. ArrayList<TaskStack> stacks = root.getStacks();
21. **boolean** hasTasks = root.hasTasks();
22. **if** (hasTasks) {
23. mRecentsView.setTaskStacks(stacks);
24. }
25. mConfig.launchedWithNoRecentTasks = !hasTasks;
26. // Create the home intent runnable
27. Intent homeIntent = **new** Intent(Intent.ACTION\_MAIN, **null**);
28. homeIntent.addCategory(Intent.CATEGORY\_HOME);
29. homeIntent.addFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK |
30. Intent.FLAG\_ACTIVITY\_RESET\_TASK\_IF\_NEEDED);
31. mFinishLaunchHomeRunnable = **new** FinishRecentsRunnable(homeIntent,
32. ActivityOptions.makeCustomAnimation(**this**,
33. mConfig.launchedFromSearchHome ? R.anim.recents\_to\_search\_launcher\_enter :
34. R.anim.recents\_to\_launcher\_enter,
35. mConfig.launchedFromSearchHome ? R.anim.recents\_to\_search\_launcher\_exit :
36. R.anim.recents\_to\_launcher\_exit));
37. // Mark the task that is the launch target
38. **int** taskStackCount = stacks.size();
39. **if** (mConfig.launchedToTaskId != -1) {
40. **for** (**int** i = 0; i < taskStackCount; i++) {
41. TaskStack stack = stacks.get(i);
42. ArrayList<Task> tasks = stack.getTasks();
43. **int** taskCount = tasks.size();
44. **for** (**int** j = 0; j < taskCount; j++) {
45. Task t = tasks.get(j);
46. **if** (t.key.id == mConfig.launchedToTaskId) {
47. t.isLaunchTarget = **true**;
48. **break**;
49. }
50. }
51. }
52. }
53. // Update the top level view's visibilities
54. **if** (mConfig.launchedWithNoRecentTasks) {
55. **if** (mEmptyView == **null**) {
56. mEmptyView = mEmptyViewStub.inflate();
57. }
58. mEmptyView.setVisibility(View.VISIBLE);
59. mRecentsView.setSearchBarVisibility(View.GONE);
60. } **else** {
61. **if** (mEmptyView != **null**) {
62. mEmptyView.setVisibility(View.GONE);
63. }
64. **if** (mRecentsView.hasSearchBar()) {
65. mRecentsView.setSearchBarVisibility(View.VISIBLE);
66. } **else** {
67. addSearchBarAppWidgetView();
68. }
69. }
70. // Animate the SystemUI scrims into view
71. mScrimViews.prepareEnterRecentsAnimation();
72. }

sfg

##### 应用展示界面的View是RecentsView

**[html]** [view plain](http://blog.csdn.net/u013656135/article/details/49686425) [copy](http://blog.csdn.net/u013656135/article/details/49686425)

1. ...
2. <!-- Recents View -->
3. **<com.android.systemui.recents.views.RecentsView**
4. android:id="@+id/recents\_view"
5. android:layout\_width="match\_parent"
6. android:layout\_height="match\_parent"
7. android:focusable="true" **/>**
9. <!-- Empty View -->
10. **<ViewStub** android:id="@+id/empty\_view\_stub"
11. android:layout="@layout/recents\_empty"
12. android:layout\_width="match\_parent"
13. android:layout\_height="match\_parent" **/>**

RecentsView里面又包含哪些东西：

**[java]** [view plain](http://blog.csdn.net/u013656135/article/details/49686425) [copy](http://blog.csdn.net/u013656135/article/details/49686425)

1. /\*\*
2. \* This is called with the full size of the window since we are handling our own insets.
3. \*/
4. @Override
5. **protected** **void** onLayout(**boolean** changed, **int** left, **int** top, **int** right, **int** bottom) {
6. // Get the search bar bounds so that we lay it out
7. **if** (mSearchBar != **null**) {
8. Rect searchBarSpaceBounds = **new** Rect();
9. mConfig.getSearchBarBounds(getMeasuredWidth(), getMeasuredHeight(),
10. mConfig.systemInsets.top, searchBarSpaceBounds);
11. mSearchBar.layout(searchBarSpaceBounds.left, searchBarSpaceBounds.top,
12. searchBarSpaceBounds.right, searchBarSpaceBounds.bottom);
13. }
14. // Layout each TaskStackView with the full width and height of the window since the
15. // transition view is a child of that stack view
16. **int** childCount = getChildCount();
17. **for** (**int** i = 0; i < childCount; i++) {
18. View child = getChildAt(i);
19. **if** (child != mSearchBar && child.getVisibility() != GONE) {
20. child.layout(left, top, left + child.getMeasuredWidth(),
21. top + child.getMeasuredHeight());
22. }
23. }
24. }

    RecentsView包含TaskStackView，TaskStackView里面又包含什么：

**[java]** [view plain](http://blog.csdn.net/u013656135/article/details/49686425) [copy](http://blog.csdn.net/u013656135/article/details/49686425)

1. ...
2. @Override
3. **public** TaskView createView(Context context) {
4. **return** (TaskView) mInflater.inflate(R.layout.recents\_task\_view, **this**, **false**);
5. }
6. ...
7. /\*\*
8. \* This is called with the size of the space not including the top or right insets, or the
9. \* search bar height in portrait (but including the search bar width in landscape, since we want
10. \* to draw under it.
11. \*/
12. @Override
13. **protected** **void** onLayout(**boolean** changed, **int** left, **int** top, **int** right, **int** bottom) {
14. // Layout each of the children
15. **int** childCount = getChildCount();
16. **for** (**int** i = 0; i < childCount; i++) {
17. TaskView tv = (TaskView) getChildAt(i);
18. **if** (tv.getBackground() != **null**) {
19. tv.getBackground().getPadding(mTmpRect);
20. } **else** {
21. mTmpRect.setEmpty();
22. }
23. tv.layout(mLayoutAlgorithm.mTaskRect.left - mTmpRect.left,
24. mLayoutAlgorithm.mTaskRect.top - mTmpRect.top,
25. mLayoutAlgorithm.mTaskRect.right + mTmpRect.right,
26. mLayoutAlgorithm.mTaskRect.bottom + mTmpRect.bottom);
27. }
28. **if** (mAwaitingFirstLayout) {
29. mAwaitingFirstLayout = **false**;
30. onFirstLayout();
31. }
32. }

    TaskStackView里面包含TaskView,也就是说，在Recents界面有几个Recent 应用，就会有几个TaskView:



    每一个TaskView包含的View（recents\_task\_view.xml）：

**[html]** [view plain](http://blog.csdn.net/u013656135/article/details/49686425) [copy](http://blog.csdn.net/u013656135/article/details/49686425)

1. **<com.android.systemui.recents.views.TaskView**
2. xmlns:android="http://schemas.android.com/apk/res/android"
3. android:layout\_width="match\_parent"
4. android:layout\_height="match\_parent"
5. android:focusable="true"**>**
6. **<FrameLayout**
7. android:id="@+id/task\_view\_content"
8. android:layout\_width="match\_parent"
9. android:layout\_height="match\_parent"**>**
10. **<com.android.systemui.recents.views.TaskViewThumbnail**
11. android:id="@+id/task\_view\_thumbnail"
12. android:layout\_width="match\_parent"
13. android:layout\_height="match\_parent" **/>**
14. **<include** layout="@layout/recents\_task\_view\_header" **/>**
15. **<FrameLayout**
16. android:id="@+id/lock\_to\_app\_fab"
17. android:layout\_width="@dimen/recents\_lock\_to\_app\_size"
18. android:layout\_height="@dimen/recents\_lock\_to\_app\_size"
19. android:layout\_gravity="bottom|right"
20. android:layout\_marginRight="15dp"
21. android:layout\_marginBottom="15dp"
22. android:translationZ="2dp"
23. android:contentDescription="@string/recents\_lock\_to\_app\_button\_label"
24. android:background="@drawable/recents\_lock\_to\_task\_button\_bg"**>**
25. **<ImageView**
26. android:layout\_width="@dimen/recents\_lock\_to\_app\_icon\_size"
27. android:layout\_height="@dimen/recents\_lock\_to\_app\_icon\_size"
28. android:layout\_gravity="center"
29. android:src="@drawable/recents\_lock\_to\_app\_pin" **/>**
30. **</FrameLayout>**
31. **</FrameLayout>**
32. lt;/com.android.systemui.recents.views.TaskView**>**

    在这其中比较重要的是TaskViewThumbnail，因为它就是用来承载每个Recents app缩略图的View，而对于recents\_lock\_to\_app\_pin这个新东西是从Android L开始加进来的一个新功能--屏幕固定，在设置-安全中可以看到。视图有了，剩下就是把数据填充进去，而这部分逻辑要关注RecentsActivity被Resume之前的PreLoad和RecentsActivity Resume时候的updateRecentsTasks()。这个就不赘述，只要晓得了整个流程的主要走向，就可以通过查看代码和debug查看数据来分析。

显示的UI：com.android.systemui/.recents.RecentsActivity

调起方法：

显示调用：com.android.systemui/.recents.RecentsActivity

隐式调用：

### 启动RecentsActivity

上一步，初始化PhoneStatusar时，初始化了Recent的点击事件；mRecentsClickListener：

这里 toggleRecentApps() 方法给BaseStatusBar发Handler消息 Message







流程如下，时序图：

核心源码如下：

#### toggleRecents

f

@Override

public void toggleRecents(Display display) {

int currentUser = sSystemServicesProxy.getCurrentUser();

if (sSystemServicesProxy.isSystemUser(currentUser)) {

// 见1.2.1

mImpl.toggleRecents(growTarget);

} else {

......

}

}

这个方法的从名字也能看出来是要启动Recents activity，主要通过RecentsImpl的toggleRecents方法，我们继续往下看

#### mImpl.toggleRecents

RecentsImpl.java

public void toggleRecents(int growTarget) {

// Skip this toggle if we are already waiting to trigger recents via alt-tab

if (mFastAltTabTrigger.isDozing()) {

return;

}

mDraggingInRecents = false;

mLaunchedWhileDocking = false;

mTriggeredFromAltTab = false;

try {

SystemServicesProxy ssp = Recents.getSystemServices();

MutableBoolean isHomeStackVisible = new MutableBoolean(true);

long elapsedTime = SystemClock.elapsedRealtime() - mLastToggleTime;

if (ssp.isRecentsActivityVisible(isHomeStackVisible)) {

......

} else {

// 间隔的时间要大于350

if (elapsedTime < MIN\_TOGGLE\_DELAY\_MS) {

return;

}

// 获得正在运行的任务栈

ActivityManager.RunningTaskInfo runningTask = ssp.getRunningTask();

// 见1.2.2

startRecentsActivity(runningTask, isHomeStackVisible.value, true /\* animate \*/,

growTarget);

// Only close the other system windows if we are actually showing recents

ssp.sendCloseSystemWindows(BaseStatusBar.SYSTEM\_DIALOG\_REASON\_RECENT\_APPS);

mLastToggleTime = SystemClock.elapsedRealtime();

}

} catch (ActivityNotFoundException e) {

Log.e(TAG, "Failed to launch RecentsActivity", e);

}

}

这里最重要的就是startRecentsActivity，也就是启动任务管理器的Acitvity

#### startRecentsActivity

文件：RecentsImpl.java

protected void startRecentsActivity(ActivityManager.RunningTaskInfo runningTask,

boolean isHomeStackVisible, boolean animate, int growTarget) {

......

// animate为true

if (!animate) {

startRecentsActivity(ActivityOptions.makeCustomAnimation(mContext, -1, -1));

return;

}

ActivityOptions opts;

if (isBlacklisted) {

opts = getUnknownTransitionActivityOptions();

} else if (useThumbnailTransition) {

// Try starting with a thumbnail transition

opts = getThumbnailTransitionActivityOptions(runningTask, mDummyStackView,

windowOverrideRect);

} else {

// 当栈的数量大于0时，hasRecentTasks为true

opts = hasRecentTasks

? getHomeTransitionActivityOptions() // 是这个，一直过渡动画

: getUnknownTransitionActivityOptions();

}

// 见1.2.3

startRecentsActivity(opts);

mLastToggleTime = SystemClock.elapsedRealtime();

}

主要做启动RecentsActivity之前的一些准备工作

#### 开始启动RecentsActivity

RecentsImpl.java

private void startRecentsActivity(ActivityOptions opts) {

Intent intent = new Intent();

intent.setClassName(RECENTS\_PACKAGE, RECENTS\_ACTIVITY);

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

| Intent.FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS

| Intent.FLAG\_ACTIVITY\_TASK\_ON\_HOME);

if (opts != null) {

// 启动 RecentsActivity

mContext.startActivityAsUser(intent, opts.toBundle(), UserHandle.CURRENT);

} else {

mContext.startActivityAsUser(intent, UserHandle.CURRENT);

}

EventBus.getDefault().send(new RecentsActivityStartingEvent());

}

#### RecentsActivity

我们先来看看onCreate

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. @Override
2. **public** **void** onCreate(Bundle savedInstanceState) {
3. **super**.onCreate(savedInstanceState);
4. mFinishedOnStartup = **false**;
5. // 如果在Recents组件初始化完毕之前就启动了该Activity,那么就finish该Acitivity
6. // 这种情况一般发生在debug时，push SysUI apk的时候
7. SystemServicesProxy ssp = Recents.getSystemServices();
8. **if** (ssp == **null**) {
9. mFinishedOnStartup = **true**;
10. finish();
11. **return**;
12. }
14. // Register this activity with the event bus
15. EventBus.getDefault().register(**this**, EVENT\_BUS\_PRIORITY);
17. // 初始化包监听器
18. // 包监听器用来监听从PackageManagerd到recent list里面内容的更新
19. mPackageMonitor = **new** RecentsPackageMonitor();
20. mPackageMonitor.register(**this**);
22. takeKeyEvents(**true**); // 即使在Activity中没有取得焦点的View，也会处理此按键事件
24. getWindow().addPrivateFlags(LayoutParams.PRIVATE\_FLAG\_NO\_MOVE\_ANIMATION);
26. // 见1.3.1 重新加载stack view
27. reloadStackView();
28. }

#### 重新加载stack view。reloadStackView

RecentsActivity.jav

private void reloadStackView() {

// If the Recents component has preloaded a load plan, then use that to prevent

// reconstructing the task stack

RecentsTaskLoader loader = Recents.getTaskLoader();

RecentsTaskLoadPlan loadPlan = RecentsImpl.consumeInstanceLoadPlan();

if (loadPlan == null) {

loadPlan = loader.createLoadPlan(this);

}

// Start loading tasks according to the load plan

RecentsConfiguration config = Recents.getConfiguration();

RecentsActivityLaunchState launchState = config.getLaunchState();

if (!loadPlan.hasTasks()) {

// 见2， 加载任务

loader.preloadTasks(loadPlan, launchState.launchedToTaskId,

!launchState.launchedFromHome);

}

RecentsTaskLoadPlan.Options loadOpts = new RecentsTaskLoadPlan.Options();

loadOpts.runningTaskId = launchState.launchedToTaskId;

loadOpts.numVisibleTasks = launchState.launchedNumVisibleTasks;

loadOpts.numVisibleTaskThumbnails = launchState.launchedNumVisibleThumbnails;

loader.loadTasks(this, loadPlan, loadOpts);

TaskStack stack = loadPlan.getTaskStack();

mRecentsView.onReload(mIsVisible, stack.getTaskCount() == 0);

mRecentsView.updateStack(stack, true /\* setStackViewTasks \*/);

// Update the nav bar scrim, but defer the animation until the enter-window event

boolean animateNavBarScrim = !launchState.launchedViaDockGesture;

mScrimViews.updateNavBarScrim(animateNavBarScrim, stack.getTaskCount() > 0, null);

// If this is a new instance relaunched by AM, without going through the normal mechanisms,

// then we have to manually trigger the enter animation state

boolean wasLaunchedByAm = !launchState.launchedFromHome &&

!launchState.launchedFromApp;

if (wasLaunchedByAm) {

EventBus.getDefault().send(new EnterRecentsWindowAnimationCompletedEvent());

}

// Keep track of whether we launched from the nav bar button or via alt-tab

if (launchState.launchedWithAltTab) {

MetricsLogger.count(this, "overview\_trigger\_alttab", 1);

} else {

MetricsLogger.count(this, "overview\_trigger\_nav\_btn", 1);

}

// Keep track of whether we launched from an app or from home

if (launchState.launchedFromApp) {

Task launchTarget = stack.getLaunchTarget();

int launchTaskIndexInStack = launchTarget != null

? stack.indexOfStackTask(launchTarget)

: 0;

MetricsLogger.count(this, "overview\_source\_app", 1);

// If from an app, track the stack index of the app in the stack (for affiliated tasks)

MetricsLogger.histogram(this, "overview\_source\_app\_index", launchTaskIndexInStack);

} else {

MetricsLogger.count(this, "overview\_source\_home", 1);

}

// Keep track of the total stack task count

int taskCount = mRecentsView.getStack().getTaskCount();

MetricsLogger.histogram(this, "overview\_task\_count", taskCount);

// After we have resumed, set the visible state until the next onStop() call

mIsVisible = true;

}

如果loadPlan.hasTasks()等于false，那么通过preLoadTasks来加载Task，接下来我们进入第二部份—Task的获取

### 缩略图的获取和显示

[Andorid N 最近任务管理器流程详解（三）](http://blog.csdn.net/u013122625/article/details/73201468)

#### 获取或者更新缩略图

RecentsTaskLoader.getAndUpdateThumbnail

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. Bitmap getAndUpdateThumbnail(Task.TaskKey taskKey, **boolean** loadIfNotCached) {
2. SystemServicesProxy ssp = Recents.getSystemServices();
4. // 如果key存在，则返回缩略图的缓存
5. ThumbnailData thumbnailData = mThumbnailCache.getAndInvalidateIfModified(taskKey);
6. **if** (thumbnailData != **null**) {
7. **return** thumbnailData.thumbnail;
8. }
10. **if** (loadIfNotCached) {
11. RecentsConfiguration config = Recents.getConfiguration();
12. **if** (config.svelteLevel < RecentsConfiguration.SVELTE\_DISABLE\_LOADING) {
13. // 见3.2 从系统中获取thumbnail
14. thumbnailData = ssp.getTaskThumbnail(taskKey.id);
15. **if** (thumbnailData.thumbnail != **null**) {
16. mThumbnailCache.put(taskKey, thumbnailData);
17. **return** thumbnailData.thumbnail;
18. }
19. }
20. }
21. // 没有装载任何缩略图
22. **return** **null**;
23. }

Task的key是有效的，那就返回缩略图的缓存，否则重新获得缩略图，并存储

#### 从系统中获取缩略图

Adsf

1. **public** ThumbnailData getTaskThumbnail(**int** taskId) {
2. **if** (mAm == **null**) **return** **null**;
3. ThumbnailData thumbnailData = **new** ThumbnailData();
5. ......
7. // 见3.3 获取task thumbnail
8. getThumbnail(taskId, thumbnailData);
9. **if** (thumbnailData.thumbnail != **null**) {
10. thumbnailData.thumbnail.setHasAlpha(**false**);
11. // We use a dumb heuristic for now, if the thumbnail is purely transparent in the top
12. // left pixel, then assume the whole thumbnail is transparent. Generally, proper
13. // screenshots are always composed onto a bitmap that has no alpha.
14. **if** (Color.alpha(thumbnailData.thumbnail.getPixel(0, 0)) == 0) {
15. mBgProtectionCanvas.setBitmap(thumbnailData.thumbnail);
16. mBgProtectionCanvas.drawRect(0, 0, thumbnailData.thumbnail.getWidth(),
17. thumbnailData.thumbnail.getHeight(), mBgProtectionPaint);
18. mBgProtectionCanvas.setBitmap(**null**);
19. Log.e(TAG, "Invalid screenshot detected from getTaskThumbnail()");
20. }
21. }
22. **return** thumbnailData;
23. }

根据task id返回top task thumbnail

#### 获取task thumbnail

文件：SystemServicesProxy.java

public void getThumbnail(int taskId, ThumbnailData thumbnailDataOut) {

if (mAm == null) {

return;

}

// 见3.4

ActivityManager.TaskThumbnail taskThumbnail = mAm.getTaskThumbnail(taskId);

if (taskThumbnail == null) {

return;

}

......

}

从Activity manager返回一个任务缩略图

#### getTaskThumbnail

通过binder调用，我们知道最后会调用AMS的getTaskThumbnail

文件：ActivityManagerService.java

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. @Override
2. **public** ActivityManager.TaskThumbnail getTaskThumbnail(**int** id) {
3. **synchronized** (**this**) {
4. // 首先要检查权限，必须有READ\_FRAME\_BUFFER权限
5. enforceCallingPermission(android.Manifest.permission.READ\_FRAME\_BUFFER,
6. "getTaskThumbnail()");
7. **final** TaskRecord tr = mStackSupervisor.anyTaskForIdLocked(
8. id, !RESTORE\_FROM\_RECENTS, INVALID\_STACK\_ID);
9. **if** (tr != **null**) {
10. // 见3.5 根据ID找到栈，然后获取缩略图
11. **return** tr.getTaskThumbnailLocked();
12. }
13. }
14. **return** **null**;
15. }

#### 根据ID找到栈，然后获取缩略图

文件：TaskRecord.java

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** TaskThumbnail getTaskThumbnailLocked() {
2. **if** (stack != **null**) {
3. **final** ActivityRecord resumedActivity = stack.mResumedActivity;
4. **if** (resumedActivity != **null** && resumedActivity.task == **this**) {
5. // 准备去截图
6. **final** Bitmap thumbnail = stack.screenshotActivitiesLocked(resumedActivity);
7. setLastThumbnailLocked(thumbnail);
8. }
9. }
10. **final** TaskThumbnail taskThumbnail = **new** TaskThumbnail();
11. // 见3.6
12. getLastThumbnail(taskThumbnail);
13. **return** taskThumbnail;
14. }

除了栈顶正在显示的TaskRecord会去实时的截取屏幕图像，其他走的是getLastThumbnail

#### getLastThumbnail

文件：TaskRecord.java

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **void** getLastThumbnail(TaskThumbnail thumbs) {
2. thumbs.mainThumbnail = mLastThumbnail;
3. thumbs.thumbnailInfo = mLastThumbnailInfo;
4. thumbs.thumbnailFileDescriptor = **null**;
5. **if** (mLastThumbnail == **null**) {
6. thumbs.mainThumbnail = mService.mRecentTasks.getImageFromWriteQueue(
7. mLastThumbnailFile.getAbsolutePath());
8. }
9. // Only load the thumbnail file if we don't have a thumbnail
10. **if** (thumbs.mainThumbnail == **null** && mLastThumbnailFile.exists()) {
11. **try** {
12. thumbs.thumbnailFileDescriptor = ParcelFileDescriptor.open(mLastThumbnailFile,
13. ParcelFileDescriptor.MODE\_READ\_ONLY);
14. } **catch** (IOException e) {
15. }
16. }
17. }

这里是去读取缩略图，这些缩略图的保存位置在：/data/system\_ce/0/recent\_images文件夹

### Task的获取

本章主要承接上面的来讲，解析一下Task的获取流程

#### RecentsTaskLoader. preloadTasks

public void preloadTasks(RecentsTaskLoadPlan plan, int runningTaskId,

boolean includeFrontMostExcludedTask) {

// 见2.2

plan.preloadPlan(this, runningTaskId, includeFrontMostExcludedTask);

}

#### RecentsTaskLoadPlan.preloadPlan

文件：RecentsTaskLoadPlan .java

public synchronized void preloadPlan(RecentsTaskLoader loader, int runningTaskId,

boolean includeFrontMostExcludedTask) {

Resources res = mContext.getResources();

ArrayList<Task> allTasks = new ArrayList<>();

if (mRawTasks == null) {

// 见2.2.1 主要是为了获取mRecentTasks,并转换成RecentTaskInfo

preloadRawTasks(includeFrontMostExcludedTask);

}

SparseArray<Task.TaskKey> affiliatedTasks = new SparseArray<>();

SparseIntArray affiliatedTaskCounts = new SparseIntArray();

String dismissDescFormat = mContext.getString(

R.string.accessibility\_recents\_item\_will\_be\_dismissed);

String appInfoDescFormat = mContext.getString(

R.string.accessibility\_recents\_item\_open\_app\_info);

long lastStackActiveTime = Prefs.getLong(mContext,

Prefs.Key.OVERVIEW\_LAST\_STACK\_TASK\_ACTIVE\_TIME, 0);

if (RecentsDebugFlags.Static.EnableMockTasks) {

lastStackActiveTime = 0;

}

long newLastStackActiveTime = -1;

int taskCount = mRawTasks.size();

for (int i = 0; i < taskCount; i++) {

ActivityManager.RecentTaskInfo t = mRawTasks.get(i);

// Compose the task key

Task.TaskKey taskKey = new Task.TaskKey(t.persistentId, t.stackId, t.baseIntent,

t.userId, t.firstActiveTime, t.lastActiveTime);

// This task is only shown in the stack if it statisfies the historical time or min

// number of tasks constraints. Freeform tasks are also always shown.

boolean isFreeformTask = SystemServicesProxy.isFreeformStack(t.stackId);

boolean isStackTask = isFreeformTask || !isHistoricalTask(t) ||

(t.lastActiveTime >= lastStackActiveTime && i >= (taskCount - MIN\_NUM\_TASKS));

boolean isLaunchTarget = taskKey.id == runningTaskId;

// The last stack active time is the baseline for which we show visible tasks. Since

// the system will store all the tasks, we don't want to show the tasks prior to the

// last visible ones, otherwise, as you dismiss them, the previous tasks may satisfy

// the other stack-task constraints.

if (isStackTask && newLastStackActiveTime < 0) {

newLastStackActiveTime = t.lastActiveTime;

}

// 加载标题，图标和颜色

ActivityInfo info = loader.getAndUpdateActivityInfo(taskKey);

String title = loader.getAndUpdateActivityTitle(taskKey, t.taskDescription);

String titleDescription = loader.getAndUpdateContentDescription(taskKey, res);

String dismissDescription = String.format(dismissDescFormat, titleDescription);

String appInfoDescription = String.format(appInfoDescFormat, titleDescription);

Drawable icon = isStackTask

? loader.getAndUpdateActivityIcon(taskKey, t.taskDescription, res, false)

: null;

// 见3 缩略图的获取

Bitmap thumbnail = loader.getAndUpdateThumbnail(taskKey, false /\* loadIfNotCached \*/);

int activityColor = loader.getActivityPrimaryColor(t.taskDescription);

int backgroundColor = loader.getActivityBackgroundColor(t.taskDescription);

boolean isSystemApp = (info != null) &&

((info.applicationInfo.flags & ApplicationInfo.FLAG\_SYSTEM) != 0);

// 将任务添加到栈

Task task = new Task(taskKey, t.affiliatedTaskId, t.affiliatedTaskColor, icon,

thumbnail, title, titleDescription, dismissDescription, appInfoDescription,

activityColor, backgroundColor, isLaunchTarget, isStackTask, isSystemApp,

t.isDockable, t.bounds, t.taskDescription, t.resizeMode, t.topActivity);

allTasks.add(task);

affiliatedTaskCounts.put(taskKey.id, affiliatedTaskCounts.get(taskKey.id, 0) + 1);

affiliatedTasks.put(taskKey.id, taskKey);

}

if (newLastStackActiveTime != -1) {

Prefs.putLong(mContext, Prefs.Key.OVERVIEW\_LAST\_STACK\_TASK\_ACTIVE\_TIME,

newLastStackActiveTime);

}

// Initialize the stacks

mStack = new TaskStack();

mStack.setTasks(mContext, allTasks, false /\* notifyStackChanges \*/);

}

#### 获取mRecentTasks

Ghdf

文件：RecentsTaskLoadPlan.java

public synchronized void preloadRawTasks(boolean includeFrontMostExcludedTask) {

int currentUserId = UserHandle.USER\_CURRENT;

updateCurrentQuietProfilesCache(currentUserId);

SystemServicesProxy ssp = Recents.getSystemServices();

// 见2.3

mRawTasks = ssp.getRecentTasks(ActivityManager.getMaxRecentTasksStatic(),

currentUserId, includeFrontMostExcludedTask, mCurrentQuietProfiles);

// 按照most-recent to least-recent的顺序进行排序

Collections.reverse(mRawTasks);

}

#### getRecentTasks

SystemServicesProxy

public List<ActivityManager.RecentTaskInfo> getRecentTasks(int numLatestTasks, int userId,

boolean includeFrontMostExcludedTask, ArraySet<Integer> quietProfileIds) {

if (mAm == null) return null;

......

// 移除掉 home/recents/excluded 这三种tasks

int minNumTasksToQuery = 10;

int numTasksToQuery = Math.max(minNumTasksToQuery, numLatestTasks);

int flags = ActivityManager.RECENT\_IGNORE\_HOME\_STACK\_TASKS |

ActivityManager.RECENT\_INGORE\_DOCKED\_STACK\_TOP\_TASK |

ActivityManager.RECENT\_INGORE\_PINNED\_STACK\_TASKS |

ActivityManager.RECENT\_IGNORE\_UNAVAILABLE |

ActivityManager.RECENT\_INCLUDE\_PROFILES;

if (includeFrontMostExcludedTask) {

flags |= ActivityManager.RECENT\_WITH\_EXCLUDED;

}

List<ActivityManager.RecentTaskInfo> tasks = null;

try {

// 见2.4 得到RecentTaskInfo

tasks = mAm.getRecentTasksForUser(numTasksToQuery, flags, userId);

} catch (Exception e) {

Log.e(TAG, "Failed to get recent tasks", e);

}

// 如果没有获得可用的tasks，那就提前退出

if (tasks == null) {

return new ArrayList<>();

}

boolean isFirstValidTask = true;

Iterator<ActivityManager.RecentTaskInfo> iter = tasks.iterator();

while (iter.hasNext()) {

ActivityManager.RecentTaskInfo t = iter.next();

// NOTE: The order of these checks happens in the expected order of the traversal of the

// tasks

// Remove the task if it or it's package are blacklsited

if (sRecentsBlacklist.contains(t.realActivity.getClassName()) ||

sRecentsBlacklist.contains(t.realActivity.getPackageName())) {

iter.remove();

continue;

}

// Remove the task if it is marked as excluded, unless it is the first most task and we

// are requested to include it

boolean isExcluded = (t.baseIntent.getFlags() & Intent.FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS)

== Intent.FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS;

isExcluded |= quietProfileIds.contains(t.userId);

// 显示每一个任务信息的LOG\*\*\*

Task.TaskKey taskKey = new Task.TaskKey(t.persistentId, t.stackId, t.baseIntent,

t.userId, t.firstActiveTime, t.lastActiveTime);

Log.d(TAG, "getRecentTasks:TASK = " + taskKey.toString()

+ "/isExcluded = " + isExcluded

+ "/includeFrontMostExcludedTask = " + includeFrontMostExcludedTask

+ "/isFirstValidTask = " + isFirstValidTask

+ "/t.id = " + t.id);

if (isExcluded && (!isFirstValidTask || !includeFrontMostExcludedTask)) {

iter.remove();

}

isFirstValidTask = false;

}

return tasks.subList(0, Math.min(tasks.size(), numLatestTasks));

}

这个方法的作用是返回一个最近任务的list

#### 得到RecentTaskInfo

通过binder调用，可以得知最后调用的是AMS的getRecentTasks

文件：ActivityManagerService.java

@Override

public ParceledListSlice<ActivityManager.RecentTaskInfo> getRecentTasks(int maxNum, int flags,

int userId) {

......

// 见2.4.1 根据userId来获取mRecentTasks

mRecentTasks.loadUserRecentsLocked(userId);

......

for (int i = 0; i < recentsCount && maxNum > 0; i++) {

TaskRecord tr = mRecentTasks.get(i);

// Only add calling user or related users recent tasks

if (!includedUsers.contains(Integer.valueOf(tr.userId))) {

if (DEBUG\_RECENTS) Slog.d(TAG\_RECENTS, "Skipping, not user: " + tr);

continue;

}

if (tr.realActivitySuspended) {

if (DEBUG\_RECENTS) Slog.d(TAG\_RECENTS, "Skipping, activity suspended: " + tr);

continue;

}

if (i == 0

|| withExcluded

|| (tr.intent == null)

|| ((tr.intent.getFlags() & Intent.FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS)

== 0)) {

...... // 根据之前设置的flag去排除掉不需要的task

// 见2.4.2 调用createRecentTaskInfoFromTaskRecord()方法把mRecentTasks里的TaskRecord转换成RecentTaskInfo

ActivityManager.RecentTaskInfo rti = createRecentTaskInfoFromTaskRecord(tr);

if (!detailed) {

rti.baseIntent.replaceExtras((Bundle)null);

}

res.add(rti);

maxNum--;

}

}

return new ParceledListSlice<>(res);

}

}

<http://blog.csdn.net/u013122625/article/details/73124181>

framwork不再展开了

### Task的移除

#### removeTask

r

public void removeTask(final int taskId) {

if (mAm == null) return;

if (RecentsDebugFlags.Static.EnableMockTasks) return;

// Remove the task.

BackgroundThread.getHandler().post(new Runnable() {

@Override

public void run() {

// 见4.2

mAm.removeTask(taskId);

}

});

}

调用AM的removeTask方法来移除Task

http://blog.csdn.net/u013122625/article/details/73224707

### Task 的更新

* RecentsImpl：实现了TaskStackListener 的接口TaskStackListenerImpl
* RecentsImpl：的构造函数注册监听SystemServicesProxy ssp = Recents.*getSystemServices*();ssp.registerTaskStackListener(**mTaskStackListener**);
* SystemServicesProxy ：**mIam**.registerTaskStackListener(**mTaskStackListener**);

**private** ITaskStackListener.Stub **mTaskStackListener** = **new** ITaskStackListener.Stub()

### 多窗口框架全解析

Android N的的多窗口框架中，总共包含了三种模式。

Split-Screen Mode: 分屏模式。

Freeform Mode 自由模式：类似于Windows的窗口模式。

Picture In Picture Mode：画中画模式（PIP）

总结一句话：多窗口框架的核心思想是分栈和设置栈边界。本文会从系统源码角度分析分栈以及设置栈边界的步骤和原理，从而解析多窗口三种模式的实现方式。

#### 栈

在Android系统中，启动一个Activity之后，必定会将此Activity存放于某一个Stack，在Android N中，系统定义了5种Stack ID，系统所有Stack的ID属于这5种里面的一种。不同的Activity可能归属于不同的Stack，但是具有相同的Stack ID。StackID如下图所示：

### 其他

<http://blog.csdn.net/u013122625/article/details/73250143>

一般事件的发出方为：TaskView和TaskStackView

事件定义的路径为：

frameworks/base/packages/SystemUI/src/com/android/systemui/recents/events

截图的时机是在Activity onPause时完成的，首先找到onPause的代码ActivityStack.java

Recent是SystemUI的之类；Recent启动RecentsActivity；

RecentsActivity里面装载RecentPanelView；

如果RecentTaskLoader取得的FirstTask为Null，则显示无最近打开的APP。否在刷新显示列表(refreshRecentTasksList())， 后台加载最近打开的APP列表loadTasksInBackground() 。

此处的mForeground分别在activity的生命周期里面更新，想法很好哦

#### 第三方APP访问Recent

Recent键的点击事件是在framework层进行处理的，不再往下传递；因此第三方APP在**onKeyDown()**与**dispatchKeyEvent()**方法中都**不能捕获RECENT\_APP\_KEY**的动作。通过广播方式可以屏蔽，在应用中添加以下代码就可以屏蔽了RECNET\_APP\_KEY：遗憾的是5.1以上貌似不可了，但是这种广播思路我们可以做全屏。屏蔽虚拟建的功能

#### 如何让app不出现在Recents列表中

在manifest.xml配置android:excludeFromRecents="true"

### 他山之石

ScreenPinningRequest：可以做悬浮框

BackgroundThread:HandlerThread 创建一个异步的后台线程

**mHandler**.removeMessages(H.***ON\_ACTIVITY\_PINNED***);  
**mHandler**.sendEmptyMessage(H.***ON\_ACTIVITY\_PINNED***);

把Event移植到core李曼

激活只需要维护一个变量就好了吧。不可靠啊，一旦系统被root。。就可以修改了啊。

*/\*\*  
 \** ***@return*** *whether this device is provisioned and the current user is set up.  
 \*/***private boolean** isUserSetup() {  
 ContentResolver cr = **mContext**.getContentResolver();  
 **return** (Settings.Global.*getInt*(cr, Settings.Global.***DEVICE\_PROVISIONED***, 0) != 0) &&  
 (Settings.Secure.*getInt*(cr, Settings.Secure.USER\_SETUP\_COMPLETE, 0) != 0);  
}

RecentsDebugFlags集中管理

RecentsConfiguration配置化

监听apk的更新用PackageMonitor **extends** BroadcastReceiver吧

@StringDef({定义常量

registerOnSharedPreferenceChangeListener(listener);

**oneway interface** IRecentsSystemUserCallbacks.aidl

Interpolators 加一些动画吧，太单调了

### recents.RecentsActivity

http://blog.csdn.net/qq\_27215521/article/details/61414105

RecentsActivity是SystemUI用于显示最近使用的应用列表，当用户点击Switch按键时会启动RecentsActivity。先分析启动的过程。

首先是用户点击SWITCH按键，PhoneWindowManager会在事件分发前先拦截该事件:

|  |
| --- |
| public long interceptKeyBeforeDispatching(WindowState win, KeyEvent event, int policyFlags) {  else if (keyCode == KeyEvent.KEYCODE\_APP\_SWITCH) {  if (!keyguardOn) {  if (down && repeatCount == 0) {  preloadRecentApps();  } else if (!down) {  toggleRecentApps();  }  }  return -1;  }  } |

根据down（down = event.getAction() == KeyEvent.ACTION\_DOWN）标识分两种情况

#### preloadRecentApps()

//预加载最近使用应用

|  |
| --- |
| private void preloadRecentApps() {  mPreloadedRecentApps = true;  try {  IStatusBarService statusbar = getStatusBarService();  if (statusbar != null) {  statusbar.preloadRecentApps();  }  } catch (RemoteException e) {  Slog.e(TAG, "RemoteException when preloading recent apps", e);  // re-acquire status bar service next time it is needed.  mStatusBarService = null;  }  } |

#### toggleRecentApps();

|  |
| --- |
| private void toggleRecentApps() {  mPreloadedRecentApps = false;  try {  IStatusBarService statusbar = getStatusBarService();  if (statusbar != null) {  statusbar.toggleRecentApps();  }  } catch (RemoteException e) {  mStatusBarService = null;  }  } |

这两种情况都获取了IStatusBarService的对象statusbar,并由该对象调用preloadRecentApps()和toggleRecentApps()方法。IStatusBarService.aidl是一个aidl服务，aidl接口中的内部抽象类Stub是由StatusBarManagerService这个类来实现的。Stub类继承了Binder，并继承我们在aidl文件中定义的接口。

### 其他

AOSP Android-7.1.1-R9

Android N的的多窗口框架中，总共包含了三种模式。

Split-Screen Mode: 分屏模式。

Freeform Mode 自由模式：类似于Windows的窗口模式。

Picture In Picture Mode：画中画模式（PIP）

经过一段时间的研究，总结一句话：多窗口框架的核心思想是分栈和设置栈边界。本文会从系统源码角度分析分栈以及设置栈边界的步骤和原理，从而解析多窗口三种模式的实现方式

* [最近任务管理器的触发](http://blog.csdn.net/u013122625/article/details/72929435)
* [Task的获取](http://blog.csdn.net/u013122625/article/details/73124181)
* [缩略图的获取](http://blog.csdn.net/u013122625/article/details/73201468)
* [Task的移除](http://blog.csdn.net/u013122625/article/details/73224707)
* [其他](http://blog.csdn.net/u013122625/article/details/73250143)

## ScreenShot事件流程

Android原生截屏是同时按下 电源键 和 音量减，开始截屏，

Android源码中对按键的捕获位于文件**PhoneWindowManager.**java (\frameworks\base\policy\src\com\android\internal\policy\impl) 中；我们可以在interceptKeyBeforeQueueing()中看到，按下截屏组合键之后，进入interceptScreenshotChord()方法，此方法比较关键，代码如下：



这里进行了判断，是否截屏可用，是否已经按下电源键和音量减，然后Handler处理mScreenshotRunnable线程；线程里调用takeScreenshot()方法开始截屏；

方法中通过bindServiceAsUser()，启动服务**TakeScreenshotService.Java**，TakeScreenshotService在onBind()事件中，调用handle启动**GlobalScreenshot.Java**对象进行真正的截屏操作mScreenshot.takeScreenshot()；这里也是多线程操作。

基本上快到最关键最核心的地方了，在此方法中，和以前的Android版本有些不同，之前的surface操作是写到surface类里，现在增加了这个surfacecontrol类来控制surface



最后，跟到Surfacecontrol里面就到了 底层 native方法了：nativeScreenshot()



UML时序图如下：



## 屏幕录制

其实官方提供的[android-ScreenCapture](https://github.com/googlesamples/android-ScreenCapture)这个Sample中已经有了MediaRecorder的实现与使用方式，还有使用MediaRecorder实现的录制屏幕到本地文件的Demo，从中我们都能了解这些API的使用。，demo极其简陋

参考意义最大的一个Demo是网友Yrom的GitHub项目[ScreenRecorder](https://github.com/yrom/ScreenRecorder)，Demo中实现了录屏并将视频流存为本地的MP4文件（咳咳，其实Yrom就是Bilibili的员工吧？

好在有个开源的第三方源码[ScreenRecorder](https://github.com/yrom/ScreenRecorder)



### 录屏原理

Android Media Projection服务漏洞深度剖析

<https://4hou.win/wordpress/?cat=3133>

https://blog.csdn.net/jiangwei0910410003/article/details/52135654

* Display 可以“投影”到一个 VirtualDisplay
* 通过 MediaProjectionManager 取得的 MediaProjection创建VirtualDisplay
* VirtualDisplay 会将图像渲染到 Surface中，而这个Surface是由MediaCodec所创建的

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. mEncoder = MediaCodec.createEncoderByType(MIME\_TYPE);
2. mSurface = mEncoder.createInputSurface();
3. mVirtualDisplay = mMediaProjection.createVirtualDisplay(name, mWidth, mHeight, mDpi, DisplayManager.VIRTUAL\_DISPLAY\_FLAG\_PUBLIC, mSurface, **null**, **null**);

f

* MediaMuxer 将从 MediaCodec 得到的图像元数据封装并输出到MP4文件中

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. int index = mEncoder.dequeueOutputBuffer(mBufferInfo, TIMEOUT\_US);
2. ByteBuffer encodedData = mEncoder.getOutputBuffer(index);
3. mMuxer.writeSampleData(mVideoTrackIndex, encodedData, mBufferInfo);

### ScreenRecorder

Demo很简单，两个Java文件：

* MainActivity.java
* ScreenRecorder.java

#### MainActivity

最重要的方法是onActivityResult，因为MediaProjection就需要从该方法开启。但是别忘了先进行MediaProjectionManager的初始化

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. @Override
2. **protected** **void** onActivityResult(**int** requestCode, **int** resultCode, Intent data) {
3. **if** (requestCode == REQUEST\_MEDIA\_PROJECTION) {
4. // NOTE: Should pass this result data into a Service to run ScreenRecorder.
5. // The following codes are merely exemplary.
7. MediaProjection mediaProjection = mMediaProjectionManager.getMediaProjection(resultCode, data);
8. **if** (mediaProjection == **null**) {
9. Log.e("@@", "media projection is null");
10. **return**;
11. }
13. VideoEncodeConfig video = createVideoConfig();
14. AudioEncodeConfig audio = createAudioConfig(); // audio can be null
15. **if** (video == **null**) {
16. toast("Create ScreenRecorder failure");
17. mediaProjection.stop();
18. **return**;
19. }
21. File dir = getSavingDir();
22. **if** (!dir.exists() && !dir.mkdirs()) {
23. cancelRecorder();
24. **return**;
25. }
26. SimpleDateFormat format = **new** SimpleDateFormat("yyyyMMdd-HHmmss", Locale.US);
27. **final** File file = **new** File(dir, "Screen-" + format.format(**new** Date())
28. + "-" + video.width + "x" + video.height + ".mp4");
29. Log.d("@@", "Create recorder with :" + video + " \n " + audio + "\n " + file);
30. mRecorder = newRecorder(mediaProjection, video, audio, file);
31. **if** (hasPermissions()) {
32. startRecorder();
33. } **else** {
34. cancelRecorder();
35. }
36. }
37. }

#### ScreenRecorder

结构很清晰，run()方法中完成了MediaCodec的初始化，VirtualDisplay的创建，以及循环进行编码的全部实现。<https://blog.csdn.net/zxccxzzxz/article/details/54150396> 的介绍过时了！

##### 线程主体

### REF

[Android实现录屏直播（一）ScreenRecorder的简单分析](https://blog.csdn.net/zxccxzzxz/article/details/54150396)

## 移植-通知栏

NotificationStackScrollLayout

BaseStatusBar

## 开发实战

更新了SystemUI之后需要重启生效，方法如下：

|  |
| --- |
| root@zs600b:/ # ps | grep zygote  root 173 1 1362924 42992 ffffffff b6e3f300 S zygote  root@zs600b:/ # kill 173 |

pid=`adb shell ps | grep 'zygote' | cut -c10-15`

adb shell kill $pid $heap\_dump\_location

## REF

1. [Android5.1SystemUI详解](http://wenku.baidu.com/link?url=HIeIaxfSPVjkLEPLf1dx852oww6DEgF6Zlez6K00E4stD9SDYFs0xvInZ89KnlHsSd9onD8DQ2mNqTKcEz-olcsv0DpygZdI7ORU210-YjW)
2. [AndroidManifest.Xml](http://blog.csdn.net/zhudaozhuan/article/details/50816086)
3. [Android 7.0 SystemUI 之启动和状态栏和导航栏简介](http://blog.csdn.net/qq_31530015/article/details/53507968)
4. [第7章 深入理解SystemUI（节选）](https://wizardforcel.gitbooks.io/deepin-android-vol3/content/7.html)
5. [Android SystemUI 分析——通知](http://light3moon.com/2015/02/06/Android%20SystemUI%20%E5%88%86%E6%9E%90%E2%80%94%E2%80%94%E9%80%9A%E7%9F%A5/)
6. [Android 多窗口框架全解析](http://blog.csdn.net/xiaosayidao/article/details/75045087)

Android的SystemUI应用,为用户提供查看最近使用应用的列表，当用户点击Switch按键时，PhoneWindowManager会拦截此次key事件，拦截后调用StatusBarManagerService 接口，StatusBarManagerService通过调用mBar（SystemUI 注册到StatusBarManagerService的Binder对象）通知SystemUI启动RecentsActivity显示最近使用应用的列表。   
启动的大概流程图如下：   


# rentsActivity的启动流程

## step1拦截key\_Switch事件：

在key事件分发之前，会调用PhoneWindowManager的interceptKeyBeforeDispatching函数优先处理key事件.

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **long** interceptKeyBeforeDispatching(WindowState win, KeyEvent event, **int** policyFlags) {
2. } **else** **if** (keyCode == KeyEvent.KEYCODE\_APP\_SWITCH) {
3. **if** (!keyguardOn) {
4. **if** (down && repeatCount == 0) {
5. preloadRecentApps();//预加载流程自行分析
6. } **else** **if** (!down) {
7. toggleRecentApps();
8. }
9. }
10. **return** -1;
11. }

preloadRecentApps();主要是预加载流程，和启动流程关键点基本一致，感兴趣自行分析。 当收到swith事件，不是down事件时调用toggleRecentApps函数：

## step 2 toggleRecentApps函数

f

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **private** **void** toggleRecentApps() {
2. mPreloadedRecentApps = **false**; // preloading no longer needs to be canceled
3. **try** {
4. IStatusBarService statusbar = getStatusBarService();
5. **if** (statusbar != **null**) {
6. statusbar.toggleRecentApps();
7. }
8. } **catch** (RemoteException e) {
9. Slog.e(TAG, "RemoteException when toggling recent apps", e);
10. // re-acquire status bar service next time it is needed.
11. mStatusBarService = **null**;
12. }
13. }

直接调用StatusBarManagerService的toggleRecentApps函数

## step 3 StatusBarManagerService .toggleRecentApps()

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **void** toggleRecentApps() {
2. **if** (mBar != **null**) {
3. **try** {
4. mBar.toggleRecentApps();
5. } **catch** (RemoteException ex) {}
6. }
7. }

mBar 是SystemUI启动的时候，在BaseStatusBar Start函数中通过调用StatusBarManagerService 的registerStatusBar函数注册过来的如下：

mCommandQueue = new CommandQueue(this, iconList);

mBarService.registerStatusBar(mCommandQueue, iconList, switches, binders);

mBar 是一个（IStatusBar）binder代理对象，他的本地对象就是运行在SystemUI的mCommandQueue对象。

这样StatusBarManagerService 就通过mBar进程间调用调用到mCommandQueue的toggleRecentApps函数。

## step 4 mCommandQueue. toggleRecentApps

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. @Override
2. **public** **void** toggleRecentApps() {
3. **int** msg = MSG\_TOGGLE\_RECENTS\_APPS;
4. mHandler.removeMessages(msg);
5. mHandler.sendEmptyMessage(msg);
6. }

这个函数只是发送了一个消息到主线程消息队列。最后处理该消息的代码如下：

mCallbacks.toggleRecentApps();

mCallbacks 是BaseStatusBar 在step3中初始化CommandQueue时传入this对象。

BaseStatusBar 又发送消息到消息队列。最后处理该消息的是BaseStatusBar 的toggleRecents函数。

## Step5 BaseStatusBar. toggleRecents

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **protected** **void** toggleRecents() {
2. **if** (mRecents != **null**) {
3. sendCloseSystemWindows(mContext, SYSTEM\_DIALOG\_REASON\_RECENT\_APPS);
4. mRecents.toggleRecents(mDisplay, mLayoutDirection, getStatusBarView());
5. }
6. }

mRecents 是Recents类对象他继承自SystemUI，在系统启动的时候，SystemServer会调用startSystemUi函数启动SystemUI应用的SystemUIService服务，

在SystemUIService 的的onCreate函数中会调用

SystemUIApplication 的startServicesIfNeeded，

在这个函数中会启动会实例化多个继承自SystemUI的类，并调用他们的Start函数，其中就包括Recents的start函数。

在Start函数中，调用putComponent(RecentsComponent.class, this);把Recents对象保存到了SystemUI的mComponents中。

BaseStatusBar中的mRecents成员变量也就是从里面取出来的。

## Step6 Recents. toggleRecents

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **void** toggleRecents(Display display, **int** layoutDirection, View statusBarView) {
2. **if** (mUseAlternateRecents) {
3. mAlternateRecents.onToggleRecents(statusBarView);
4. **return**;
5. }
6. ......
7. }

mUseAlternateRecents 默认为真，mAlternateRecents.onToggleRecents(statusBarView);该函数直接调用了toggleRecentsActivity

## step7 AlternateRecentsComponent. toggleRecentsActivity

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **void** toggleRecentsActivity() {
2. ActivityManager.RunningTaskInfo topTask = getTopMostTask();
3. AtomicBoolean isTopTaskHome = **new** AtomicBoolean();
4. **if** (isRecentsTopMost(topTask, isTopTaskHome)) {
5. Intent intent = **new** Intent(ACTION\_TOGGLE\_RECENTS\_ACTIVITY);
6. intent.setPackage(mContext.getPackageName());
7. intent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT |
8. Intent.FLAG\_RECEIVER\_FOREGROUND);
9. mContext.sendBroadcastAsUser(intent, UserHandle.CURRENT);
10. mLastToggleTime = System.currentTimeMillis();
11. **return**;
12. } **else** {
13. startRecentsActivity(topTask, isTopTaskHome.get());
14. }
15. }

这个函数主要是首先判断当前界面是不是RecentsActivity

如果是就发送一个广播，关闭RecentsActivity.广播肯定是RecentsActivity接收了，接收后调用退出动画，退出完成后调用RecentsActivity的finish函数。

如果不是则调用startRecentsActivity启动RecentsActivity。

RecentsActivity启动：界面的展示离不开数据，从界面上看，RecentsActivity至少需要应用名称，图片，以及截图，接下来看RecentsActivity数据的获取流程

## Step8 在RecentsActivity的onStart函数中有调用updateRecentsTasks来更新获取界面显示的数据

D

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **void** updateRecentsTasks(Intent launchIntent) {
2. ……
3. RecentsTaskLoader loader = RecentsTaskLoader.getInstance();
4. SpaceNode root = loader.reload(**this**,
5. Constants.Values.RecentsTaskLoader.PreloadFirstTasksCount,
6. mConfig.launchedFromHome);
7. ArrayList<TaskStack> stacks = root.getStacks();
8. **if** (!stacks.isEmpty()) {
9. mRecentsView.setTaskStacks(root.getStacks());
10. }
11. ……
12. **if** (mConfig.launchedWithNoRecentTasks) {
13. **if** (mEmptyView == **null**) {
14. mEmptyView = mEmptyViewStub.inflate();
15. }
16. mEmptyView.setVisibility(View.VISIBLE);
17. mRecentsView.setSearchBarVisibility(View.GONE);
18. } **else** {
19. **if** (mEmptyView != **null**) {
20. mEmptyView.setVisibility(View.GONE);
21. }
22. **if** (mRecentsView.hasSearchBar()) {
23. mRecentsView.setSearchBarVisibility(View.VISIBLE);
24. } **else** {
25. addSearchBarAppWidgetView();
26. }
27. }
28. mScrimViews.prepareEnterRecentsAnimation();
29. }

这个函数主要就是

1 调用RecentsTaskLoader的reload函数获取相关最近使用应用的相关信息。

2 调用mRecentsView.setTaskStacks，把获得的TaskStack设置到RecentsView中创建UI结构图中的TaskStackView

3 如果没有最近使用的信息则显示空的提示信息。这一步比较简单只是设置View的显示隐藏。

## Step9 RecentsTaskLoader.reload函数

先看一下RecentsTaskLoader构造函数中初始化的几个重要的类和变量

1 mSystemServicesProxy = new SystemServicesProxy(context);

SystemServicesProxy 这个类主要承接了RecentsActivity和系统服务的交互，包含获取最近任务列表，启动制定任务所在的应用等等。

2 mLoadQueue = new TaskResourceLoadQueue();一个队列主要用于缓存任务

3 mApplicationIconCache = new DrawableLruCache(iconCacheSize);

mThumbnailCache = new BitmapLruCache(thumbnailCacheSize);

mActivityLabelCache = new StringLruCache(100);

三个缓存，主要用于缓存应用图标，名称，截屏。

4 mLoader = new TaskResourceLoader(mLoadQueue, mApplicationIconCache, mThumbnailCache, mDefaultThumbnail, mDefaultApplicationIcon);

TaskResourceLoader 继承自Runnable 他的构造函数中启动了一个HandlerThread，在他的Run函数中一直循环从mLoadQueue中读取任务并获取任务所对应的截图，并缓存到mThumbnailCache中，然后通知TaskView加载截图并显示。

## Step10 mRecentsView.setTaskStacks

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **void** setTaskStacks(ArrayList<TaskStack> stacks) {
2. **int** childCount = getChildCount();
3. **for** (**int** i = childCount - 1; i >= 0; i--) {
4. View v = getChildAt(i);
5. **if** (v != mSearchBar) {
6. removeViewAt(i);
7. }
8. }
9. mStacks = stacks;
10. **int** numStacks = mStacks.size();
11. **for** (**int** i = 0; i < numStacks; i++) {
12. TaskStack stack = mStacks.get(i);
13. TaskStackView stackView = **new** TaskStackView(getContext(), stack);
14. stackView.setCallbacks(**this**);
15. **if** (mConfig.debugModeEnabled) {
16. stackView.setDebugOverlay(mDebugOverlay);
17. }
18. addView(stackView);
19. }
20. mAlreadyLaunchingTask = **false**;
21. }

这个函数先移除RecentsView中的所有子View然后把获取的Taskstack保存到mStacks中，然后创建TaskStackView 视图，并把视图中要显示的TaskStack保存到TaskStackView中。

最后把创建的TaskStackView视图作为子View添加到RecentsView中。

到此装载过程完成。

当RecentsActivity中ViewRootImpl对所有的View进程一次Measure ,过程中。会调用TaskStackView的onMeasure函数，在onMeasure函数中会调用到TaskView的synchronizeStackViewsWithModel函数。

## Step11 TaskStackView. synchronizeStackViewsWithModel

这个函数比较长就不贴代码了，主要完成两个工作，

1 创建UI结构图中的TaskView并作为子View添加到TaskStackView中。

2 为TaskView计算在TaskStackView中的坐标并做一个TaskViewTransform平移动画。（这部分代码就不详细分析了）

在这个函数中就是通过调用tv = mViewPool.pickUpViewFromPool(task, task);来创建TaskView

mViewPool = new ViewPool

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. V pickUpViewFromPool(T preferredData, T prepareData) {
2. V v = **null**;
3. **boolean** isNewView = **false**;
4. **if** (mPool.isEmpty()) {
5. v = mViewCreator.createView(mContext);
6. isNewView = **true**;
7. }
8. ……
9. mViewCreator.prepareViewToLeavePool(v, prepareData, isNewView);
10. **return** v;
11. }

从ViewPool的构造函数可以知道mViewCreator 就是TaskStackView

TaskStackView的createView就是创建了一个TaskView

在调用TaskStackView的prepareViewToLeavePool函数

## Step12 TaskStackView.prepareViewToLeavePool函数

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **void** prepareViewToLeavePool(TaskView tv, Task task, **boolean** isNewView) {
2. tv.onTaskBound(task);
3. ……
4. RecentsTaskLoader.getInstance().loadTaskData(task);
5. tv.setClipViewInStack(**true**);
6. **if** (isNewView) {
7. addView(tv, insertIndex);
8. tv.setTouchEnabled(**true**);
9. tv.setCallbacks(**this**);
10. } **else** {
11. attachViewToParent(tv, insertIndex, tv.getLayoutParams());
12. }
13. }

首先调用TaskView的onTaskBound函数，这个函数主要就是设置TaskView所关联的Task，也就是说所要显示的应用，然后设置TaskView为Task的成员变量mCb，而mCb主要用来通知刷新的。然后把创建的TaskView添加到TaskStackView中。

然后调用RecentsTaskLoader.getInstance().loadTaskData(task)函数

## Step13 RecentsTaskLoader. loadTaskData

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **void** loadTaskData(Task t) {
2. Drawable applicationIcon = mApplicationIconCache.getAndInvalidateIfModified(t.key);
3. Bitmap thumbnail = mThumbnailCache.getAndInvalidateIfModified(t.key);
4. **boolean** requiresLoad = (applicationIcon == **null**) || (thumbnail == **null**);
5. applicationIcon = applicationIcon != **null** ? applicationIcon : mDefaultApplicationIcon;
6. **if** (requiresLoad) {
7. mLoadQueue.addTask(t);
8. }
9. t.notifyTaskDataLoaded(thumbnail == mDefaultThumbnail ? **null** : thumbnail, applicationIcon);
10. }

这个函数主要是判断当前Task的应用启动图标和截图是否已经在mApplicationIconCache中缓存。

如果没有缓存，则把Task加入到等待加载截图的队列mLoadQueue中，前面也介绍过这个队列TaskResourceLoader的run函数会一直从mLoadQueue 读取Task去获取其对应的应用截图和启动图标。

## Step14 TaskResourceLoader.run

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **void** run() {
2. **while** (**true**) {
3. **if** (mCancelled) {
4. ……
5. } **else** {
6. SystemServicesProxy ssp = mSystemServicesProxy;
7. **final** Task t = mLoadQueue.nextTask();
8. **if** (t != **null**) {
9. Drawable cachedIcon = mApplicationIconCache.get(t.key);
10. Bitmap cachedThumbnail = mThumbnailCache.get(t.key);
11. **if** (cachedIcon == **null**) {
12. cachedIcon = getTaskDescriptionIcon(t.key, t.icon, t.iconFilename, ssp,
13. mContext.getResources());
14. **if** (cachedIcon == **null**) {
15. ActivityInfo info = ssp.getActivityInfo(t.key.baseIntent.getComponent(),
16. t.key.userId);
17. **if** (info != **null**) {
18. cachedIcon = ssp.getActivityIcon(info, t.key.userId);
19. }
20. }
22. **if** (cachedIcon == **null**) {
23. cachedIcon = mDefaultApplicationIcon;
24. }
25. mApplicationIconCache.put(t.key, cachedIcon);
26. }
27. **if** (cachedThumbnail == **null**) {
28. cachedThumbnail = ssp.getTaskThumbnail(t.key.id);
29. **if** (cachedThumbnail != **null**) {
30. cachedThumbnail.setHasAlpha(**false**);
31. } **else** {
32. cachedThumbnail = mDefaultThumbnail;
33. }
34. mThumbnailCache.put(t.key, cachedThumbnail);
35. }
36. **if** (!mCancelled) {
37. **final** Drawable newIcon = cachedIcon;
38. **final** Bitmap newThumbnail = cachedThumbnail == mDefaultThumbnail
39. ? **null** : cachedThumbnail;
40. mMainThreadHandler.post(**new** Runnable() {
41. @Override
42. **public** **void** run() {
43. t.notifyTaskDataLoaded(newThumbnail, newIcon);
44. }
45. });
46. }
47. }
48. ……
49. }
50. }
51. }
52. }

这个函数比较好分析，首先从缓存中查看对应的Task的应用启动图标和截图是否已经存在，如果不存在就去获取，获取过程都是调用的系统接口，自行查看。

获取完成后缓存，然后往主线程发送一个消息，去执行t.notifyTaskDataLoaded(newThumbnail, newIcon);函数。

## Task. notifyTaskDataLoaded

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. /\*\* Notifies the callback listeners that this task has been loaded \*/
2. **public** **void** notifyTaskDataLoaded(Bitmap thumbnail, Drawable applicationIcon) {
3. **this**.applicationIcon = applicationIcon;
4. **this**.thumbnail = thumbnail;
5. **if** (mCb != **null**) {
6. mCb.onTaskDataLoaded();
7. }
8. }

这个函数就是保存获取的应用启动图标和应用截图，然后调用mCb. onTaskDataLoaded函数。

在step12中为TaskView关联要显示的Task后，还把TaskView设置为Task的mCb

TaskView的onTaskDataLoaded函数大家就知道了，把刚刚装载的应用启动图标和截图设置到TaskView对应的View中显示出来。

## REF

[RecentsTask分析Android5.0](http://blog.csdn.net/zhangyawen1i/article/details/53350006)

# VolumeUI音量调节

7.0进行了重构

VolumeDialog

VolumeDialogController

VolumeDialog，CustomDialog **extends** Dialog

Binder

1.VC **extends** IVolumeController.Stub

volumeChanged

2. onVolumeChangedW

**final boolean** showUI = (flags & AudioManager.***FLAG\_SHOW\_UI***) != 0;

VolumeDialog **implements** TunerService.Tunable

原生为何需要自己绘制一个窗口呢？

## And5

<https://blog.csdn.net/hello_gson/article/details/43795251>

Android5.0 静音模式下让闹钟仍然响铃

<!-- Interruption level: None. [CHAR LIMIT=20] -->

<string name="interruption\_level\_none">None</string>

<!-- Interruption level: Priority. [CHAR LIMIT=20] -->

<string name="interruption\_level\_priority">Priority</string>

<!-- Interruption level: All. [CHAR LIMIT=20] -->

<string name="interruption\_level\_all">All</string>

interruption\_level\_priority

### ZenModePanel

VolumePanel.java (packages\systemui\src\com\android\systemui\volume): private ZenModePanel mZenPanel;

VolumePanel.java (packages\systemui\src\com\android\systemui\volume): mZenPanel = (ZenModePanel) mView.findViewById(com.android.systemui.R.id.zen\_mode\_panel);

VolumePanel.java (packages\systemui\src\com\android\systemui\volume): initZenModePanel();

VolumePanel.java (packages\systemui\src\com\android\systemui\volume): private void initZenModePanel() {

VolumePanel.java (packages\systemui\src\com\android\systemui\volume): mZenPanel.setCallback(new ZenModePanel.Callback() {

VolumeUI.java (packages\systemui\src\com\android\systemui\volume): ZenModePanel.ZEN\_SETTINGS, true /\* onlyProvisioned \*/, true /\* dismissShade \*/);

ZenModePanel.java (packages\systemui\src\com\android\systemui\volume):public class ZenModePanel extends LinearLayout {

ZenModePanel.java (packages\systemui\src\com\android\systemui\volume): private static final String TAG = "ZenModePanel";

ZenModePanel.java (packages\systemui\src\com\android\systemui\volume): public ZenModePanel(Context context, AttributeSet attrs) {

ZenModePanel.java (packages\systemui\src\com\android\systemui\volume): if (DEBUG) Log.d(mTag, "new ZenModePanel");

ZenModePanel.java (packages\systemui\src\com\android\systemui\volume): return mContext.getSharedPreferences(ZenModePanel.class.getSimpleName(), 0);

Zen\_mode\_panel.xml (packages\systemui\res\layout):<com.android.systemui.volume.ZenModePanel xmlns:android="http://schemas.android.com/apk/res/android"

Zen\_mode\_panel.xml (packages\systemui\res\layout):</com.android.systemui.volume.ZenModePanel>

## 状态栏PhoneStatusBarPolicy

**PhoneStatusBarPolicy**.java (packages\systemui\src\com\android\systemui\statusbar\phone): zenDescription = mContext.getString(R.string.interruption\_level\_priority);

Strings.xml (packages\systemui\res\values): <string name="interruption\_level\_priority">Priority only</string>

## ZenModePanel

**ZenFooter**.java (packages\systemui\src\com\android\systemui\volume): isZenPriority() ? mContext.getString(R.string.interruption\_level\_priority)

ZenModePanel.java (packages\systemui\src\com\android\systemui\volume): R.string.interruption\_level\_priority,

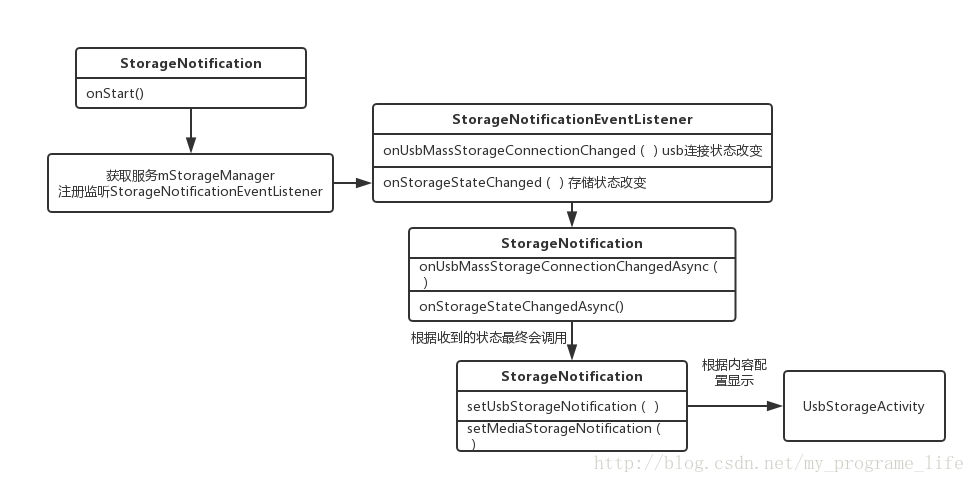
**ZenModePanel**.java (packages\systemui\src\com\android\systemui\volume): modeText = mContext.getString(R.string.interruption\_level\_priority);

# StorageNotification

StorageNotification **extends** SystemUI

类只有StorageNotification.java和UsbStorageActivity.java .StorageNotification负责逻辑处理，UsbStorageActivity负责显示

## **StorageNotification.java 的主要函数关系图**



## Start

函数，会去注册StorageEventListener

**public void** start() {  
 **mNotificationManager** = **mContext**.getSystemService(NotificationManager.**class**);  
  
 **mStorageManager** = **mContext**.getSystemService(StorageManager.**class**);  
 **mStorageManager**.registerListener(**mListener**);//systemui和系统的生命周期一样长，因此没有必要unregisterListener  
  
 **mContext**.registerReceiver(**mSnoozeReceiver**, **new** IntentFilter(***ACTION\_SNOOZE\_VOLUME***),  
 android.Manifest.permission.***MOUNT\_UNMOUNT\_FILESYSTEMS***, **null**);  
 **mContext**.registerReceiver(**mFinishReceiver**, **new** IntentFilter(***ACTION\_FINISH\_WIZARD***),  
 android.Manifest.permission.***MOUNT\_UNMOUNT\_FILESYSTEMS***, **null**);  
  
 *// Kick current state into place* **final** List<DiskInfo> disks = **mStorageManager**.getDisks();  
 **for** (DiskInfo disk : disks) {  
 onDiskScannedInternal(disk, disk.**volumeCount**);  
 }  
 **final** List<VolumeInfo> vols = **mStorageManager**.getVolumes();  
 **for** (VolumeInfo vol : vols) {  
 onVolumeStateChangedInternal(vol);  
 }  
 updateMissingPrivateVolumes();  
}

## StorageEventListener

对于U盘或SD卡插拔事件，我们常用的是通过广播的方式去接收，具体可以参考：

<https://blog.csdn.net/silenceburn/article/details/6083375>

基本可以满足普通app的需求，但是sd卡设置为内部卡的时候，则接收不到广播，因此还得用监听的方式

StorageEventListener的实现如下：

**private final** StorageEventListener **mListener** = **new** StorageEventListener() {  
 @Override  
 **public void** onVolumeStateChanged(VolumeInfo vol, **int** oldState, **int** newState) {  
 onVolumeStateChangedInternal(vol);  
 }  
  
 @Override  
 **public void** onVolumeRecordChanged(VolumeRecord rec) {  
 *// Avoid kicking notifications when getting early metadata before  
 // mounted. If already mounted, we're being kicked because of a  
 // nickname or init'ed change.* **final** VolumeInfo vol = **mStorageManager**.findVolumeByUuid(rec.getFsUuid());  
 **if** (vol != **null** && vol.isMountedReadable()) {  
 onVolumeStateChangedInternal(vol);  
 }  
 }  
  
 @Override  
 **public void** onVolumeForgotten(String fsUuid) {  
 *// Stop annoying the user* **mNotificationManager**.cancelAsUser(fsUuid, SystemMessage.NOTE\_STORAGE\_PRIVATE,  
 UserHandle.***ALL***);  
 }  
  
 @Override  
 **public void** onDiskScanned(DiskInfo disk, **int** volumeCount) {  
 onDiskScannedInternal(disk, volumeCount);  
 }  
  
 @Override  
 **public void** onDiskDestroyed(DiskInfo disk) {  
 onDiskDestroyedInternal(disk);  
 }  
};

StorageEventListener中还有其他的call back函数，对于外置存储设备插拔，我觉得这两个函数应该都可以拿到对应的事件：

public void onStorageStateChanged(String path, String oldState, String newState)   
public void onVolumeStateChanged(VolumeInfo vol, int oldState, int newState)

在StorageNotification使用的是onVolumeStateChanged和onVolumeRecordChanged

<https://blog.csdn.net/zhudaozhuan/article/details/50876755>

<https://blog.csdn.net/my_programe_life/article/details/78616970>

# 业务定制

## 绝对全屏

PhoneStatusBar.setSystemUiVisibility

# 亮度调节

Sdfas

<**activity  
 android:name=".settings.BrightnessDialog"  
 android:label="@string/quick\_settings\_brightness\_dialog\_title"  
 android:theme="@android:style/Theme.DeviceDefault.Dialog"  
 android:finishOnCloseSystemDialogs="true"  
 android:launchMode="singleInstance"  
 android:excludeFromRecents="true"  
 android:exported="true"**>  
 <**intent-filter**>  
 <**action android:name="android.intent.action.SHOW\_BRIGHTNESS\_DIALOG"** />  
 <**category android:name="android.intent.category.DEFAULT"** />  
 </**intent-filter**>  
</**activity**>

# TASK

分析如何优雅的屏蔽导航栏和qs菜单 & StatusBarManager.DISABLE\_EXPAN

# REF

[system ui powerUI](ttps://blog.csdn.net/tiantao2012/article/details/52314389)

《深入理解Android 卷III》第七章 深入理解[SystemUI](https://blog.csdn.net/ccjhdopc/article/details/52424477)