# 前言

分析系统应用，需要按照步骤来进行分析，一般需要注意的几点主要有：

最好把源码单独提取出来，并编译运行，这样哪里不清楚就改哪里，不够这里可能有系统api无法识别的问题，需要做系统签名。

* 寻找一个程序的入口，Java，C，C＃都是从main函数开始执行的，分析源码程序时也要先照一个入口
* 先弄清楚程序实现的主要功能是什么
* 关注流程和逻辑，不要过于关注细节。要学会抓大放小。

编译系统app：批量删除product属性的string资源, 未加特殊说明，都是用的android5.1.1（SDK22）

# Android系统应用

## 什么是Android系统应用

Android系统应用具有如下几个特点，这些特点都是普通的Android应用不具备的。

* 嵌入到Android ROM中，通过普通的方法无法卸载这些程序。
* 拥有更高的权限。例如，可以实现静默安装，不会出现权限提示对话框，由于Google Play是Android系统应用，所以即使在没有root权限的情况下，Google Play也可以读写/data/app目录1，所以只需要将APK文件复制到这个目录即可安装程序，根本就不需要通过PackageInstaller2进行安装。
* 可以调用Android SDK的私有API，这些API在ADT开发环境下无法调用。

注：安装apk的本质就是拷贝文件到**/data/app或者其子目录**，1）在root的条件下C:\Users\key.guan>adb push A.apk /data/app 然后reboot就完成了A.apk的安装了；2）adb install –r A.apk

## 为什么要研究Android系统应用

有很多直接与Android应用交互的接口都是在这些系统应用中实现的。了解系统应用，意味着对Android系统最上层的API有一个非常透彻的了解。所有的Android系统应用的源代码都在packages目录。其中packages/app是最核心的目录，所有内嵌的APK程序都在该目录中。app目录告诉我们一切Android系统的窗口（Activity）。研究Android系统应用的源代码好处多多：

* 了解Android系统中有哪些窗口、Content Provider、Service和Broadcast可以与之交互。
* 充分掌握很多高级应用的使用方法，例如，OTA升级是如何实现的。
* 对实现原理比较感兴趣的读者可以通过这些源代码了解像Launcher2、短信管理等应用的内容构造。
* 对于想编写可以完成系统级操作的应用的读者，可以学会如何将Android应用嵌入到ROM，升级为Android系统应用，进而可以做任何自己想做的事。
* 对于那些有极客情结的读者，完全定制自己的ROM是最令人振奋的梦想，而Android系统应用将是实现这一梦想最重要的基石。

## 如何编写Android系统应用

Android系统应用和普通的Android应用基本相同，都是主要用Java语言编写的APK程序。不过前者与后者有如下两点不同。

* 签名不同。Android系统应用使用的是系统签名，或称为平台（Platform）签名，而普通的Android应用使用的是一般的签名。
* 可以访问的API不同。在Android SDK中有很多API（类、接口、方法等）都声明为hide1。这类API不允许在普通的Android应用中访问，而只有经过Platform签名的Android系统应用才能使用这些API。

Android源代码中包含了4个签名，前面使用的platform就是其中之一。这些签名文件中在如下的目录中。

<Android源代码根目录>/build/target/product/security，含义。

* testkey：普通的Key，默认情况下使用。如果Android.mk文件不设置LOCAL\_CERTIFICATE 变量，就使用该签名。
* **platform** ：使用该签名的系统应用可以**执行平台的核心功能**。
* shared ：使用该签名的系统应用可以和home/contacts进程共享数据。
* media ：使用该签名的系统应用将成为media/download 系统的一部分。

LOCAL\_CERTIFICATE 变量可以指定上述4个值，例如，如果值为platform，会使用security目录的platform.pk8文件对APK文件进行签名。如果值为media，会使用media.pk8文件对APK文件进行签名。

我们开发的apk需要用到系统权限，需要在AndroidManifest.xml中添加共享系统进程属性：

android:sharedUserId="android.uid.system"

android:sharedUserId="android.uid.shared"

android:sharedUserId="android.media

apk的签名就需要是系统签名(platform、shared或media)才能正常使用。否则报INSTALL\_FAILED\_SHARED\_USER\_INCOMPATIBLE错，如何编译和运行Android系统应用呢？

### 源码命令行签名（Android.mk）

这种方式比较麻烦，你需要有编译过的源码环境，并按如下步骤：

1、拷贝App源码到Android源码的packages/apps/目录或者子目录下，且App源码是普通(Eclipse)格式的

2、配置Android.mk，在其中添加LOCAL\_CERTIFICATE := platform 或 shared 或 media

3、使用mm编译App，生成的apk即系统签名

Android.mk是Android源代码专用的编译文件，相当于GCC的Makefile文件，内容只需要在apps目录中找一个Android系统应用，如PackageInstaller，将该程序中Android.mk文件复制一份，然后进行适当修改即可。

下面先看一下FirstSystemApp程序的Android.mk文件的内容。

src/ch06/FirstSystemApp/Android.mk

LOCAL\_PATH:= $(call my-dir)

include $(CLEAR\_VARS)

LOCAL\_MODULE\_TAGS := optional

LOCAL\_SRC\_FILES :=$(call all-java-files-under, src)

LOCAL\_STATIC\_JAVA\_LIBRARIES += android-support-v4

#　将编译生成FirstSystemApp.apk文件

**LOCAL\_PACKAGE\_NAME := FirstSystemApp**

**LOCAL\_CERTIFICATE := platform**

include $(BUILD\_PACKAGE)

Android.mk文件的大多数内容都是标准的写法，只有下面两行代码需要了解一下。

LOCAL\_PACKAGE\_NAME := FirstSystemApp

LOCAL\_CERTIFICATE := platform

其中第一行需要根据不同的Android系统应用进行修改。系统会根据LOCAL\_PACKAGE\_NAME变量的值生成APK文件，例如，本例该变量的值是FirstSystemApp，所以会在out/target/product/ maguro/system/app目录生成FirstSystemApp.apk文件， adb push命令上传到/system/app目录即可完成安装。

LOCAL\_CERTIFICATE变量表示签名类型，系统应用通常设为platform，表示Platform签名。

### Java命令行签名

这种方式签名简单，App可以在Eclipse或Android Studio下编译，然后给apk重新签名即可。但调试比较痛苦，即使写成脚本，也需要重复一样的操作，相关文件：

../build/target/product/security/platform.x509.pem

../build/target/product/security/platform.pk8

../out/host/linux-x86/framework/signapk.jar （源码路径: ../build/tools/signapk）

步骤

1、将相关文件及源apk文件置于同一路径下  
2、检查源apk包，去掉META-INF/CERT.SF 和 META-INF/CERT.RSA 文件（可以忽略这一步）  
3、执行签名命令java -jar signapk.jar platform.x509.pem platform.pk8 old.apk new.apk即可

### keytool-importkeypair集成系统签名

让Android Studio集成系统签名，需要用到一个工具keytool-importkeypair，这个工具的作用是将系统签名的相关信息导入到已有的签名文件里。可从[这里](https://github.com/getfatday/keytool-importkeypair)下载。工具的使用方法参考[使用keytool工具](http://czj4451.iteye.com/blog/1487684)这篇文章。最好使用mac 或Linux系统下的AS生成，因为keytool-importkeypair工具是执行在类**Unix**内核的系统上的，不要整成**win**格式的文本了，在window上会有些问题（在git的sh执行，需要修改命令tmpdir=`mkdir -p "/tmp/${scriptname}.XXXX"`）,步骤

1. 在AS下生成demo.jks签名文件：菜单栏->build->Generate Signed APK..->

证书查看：keytool -list -v -keystore SystemSignDemo2.jks -storepass 123456

1. 执行命令编译出demo.jks ：linux下需要为脚本文件添加可执行权限

./keytool-importkeypair -k demo.jks -p 123456 -pk8 platform.pk8 -cert platform.x509.pem -alias demo

# demo.jks : 签名文件

# 123456 : 签名文件密码

# platform.pk8、platform.x509.pem : 系统签名文件

# demo : 签名文件别名

3.builde.gradle配置demo.jks

在android区域下(与defaultConfig同级)添加配置：

signingConfigs {

release {

storeFile file("../signature/demo.jks")

storePassword '123456'

keyAlias 'demo'

keyPassword '123456'

}

debug {

storeFile file("../signature/demo.jks")

storePassword '123456'

keyAlias 'demo'

keyPassword '123456'

}

}

## 小结

完整的系统签名的app过程

1. 应该是使用平台签名，例如此处的Settings.apk的Android.mk文件中指定的签名平台是platform即系统平台签名，所以在签名的时候会使用系统的签名文件进行签名。platform对应的系统签名文件的位置为：

android源码根目录/build/target/product/security/platform.pk8和android源码根目录/build/target/product/security/platform.x509.pem 两个文件

2. 在应用工程的清单配置文件AndroidManifest.xml文件中指定共享用户ID，并将coreApp属性设置为true

3. ndroid源码中使用mm/mmm命令进行编译，其权限就会与系统设置一致；

4. 将apk文件复制到Android设备的/system/app目录中

# [Android 初始化设置向导Provision](http://www.cnblogs.com/mythou/p/3425570.html)

源码路径：packages/apps/ Provision

## 概述

Provision是一个系统初始化引导程序，类似于我们的设置向导，原生的Android里面Provision只做了一件事，就是写入一个 DEVICE\_PROVISIONED标记。不过这个标记作用很大，这个标记只会在系统全新升级（双清）的时候写入**一次**，**代表了Android系统升级准备完成**，指示其他Framework和其他程序，机器已经Provisioned。没有Provisioned的机器，有些功能的表现是不一样的，比如，锁屏程序不会锁屏；对HOME key的处理也不同；电话也是打不进来的等。

Settings.Global.DEVICE\_PROVISIONED设置的变化。这一设置表示此设备是否已经归属于某一个用户。比如当用户打开一个新购买的设备时，初始化设置向导将会引导用户阅读使用条款、设置帐户等一系列的初始化操作。在初始化设置向导完成之前， Settings.Global.DEVICE\_PROVISIONED的值为false，表示这台设备并未归属于某一个用户。当设备并未归属于某以用户时，状态栏会禁用一些功能以避免信息的泄露。

## 源码

### Manifest

|  |
| --- |
| <**application**>  <**activity android:name="DefaultActivity"  android:excludeFromRecents="true"**>  <**intent-filter android:priority="1"**>  <**action android:name="android.intent.action.MAIN"** />  <**category android:name="android.intent.category.HOME"** />  <**category android:name="android.intent.category.DEFAULT"** />  </**intent-filter**>  </**activity**> </**application**> |

从上面代码可以看到两个比较有意思的地方，

第一：Provision配置了category.HOME属性，这是桌面程序的标记，简单点说就是Launcher程序才会配置的标记。

第二：priority=1，配置了优先级，也就是说它的优先级比我们原生的Launcher优先级还要高，它会在Launcher启动前就运行起来。一般android广播分为两种一种是Normal Broadcast（普通广播）：Noraml Broadcast是完全异步的，可以在同一时刻(逻辑上)被所有接收者接收到，消息传递的效率比较高。但缺点是接收者不能将处理结果传递给下一个接收者，并且无法终止Broadcast Intent的传播。一种是Ordered Broadcast(有序广播):Ordered Broadcast的接收者将按预先声明的优先级依次接收Broadcast。如A的级别高于B,B的级别高于C,那么Broadcast先传给A,再传 给B,最后传给C。优先级别声明在元素的android:priority属性中，数越大优先级别越高，取值范围为-1000~1000。

### DefaultActivity

|  |
| --- |
| **public class** DefaultActivity **extends** Activity {  @Override  **protected void** onCreate(Bundle icicle)  {  **super**.onCreate(icicle);   *// Add a persistent setting to allow other apps to know the device has been provisioned.* Settings.Global.*putInt*(getContentResolver(), Settings.Global.***DEVICE\_PROVISIONED***, 1); Settings.Secure.*putInt*(getContentResolver(), Settings.Secure.USER\_SETUP\_COMPLETE, 1);  *// remove this activity from the package manager.* PackageManager pm = getPackageManager();  ComponentName name = **new** ComponentName(**this**, DefaultActivity.**class**);  pm.setComponentEnabledSetting(name, PackageManager.***COMPONENT\_ENABLED\_STATE\_DISABLED***,  PackageManager.***DONT\_KILL\_APP***);   *// terminate the activity.* finish();  } } |

上面就是Provision的全部源码，Provision只有一个Activity，只做了这两件事：

* 设置DEVICE\_PROVISIONED标记
* 禁止Provision自己的Activity组件

第一，DEVICE\_PROVISIONED就是我们上面说的系统升级完标记。这个标记是保存在：/data/data/com.android.providers.settings/database/settings.db数据库中，准确来说是settings.db的secure表里面。

第二，禁止组件，所以这个组件只会运行一次，所以我们如果没有格式化/data目录，这个组件就会被禁止。所以会导致一个问题，就是我们的DEVICE\_PROVISIONED标记不会再次被写入，如果我们第一次DEVICE\_PROVISIONED标记没有被写成功，这样可能导致异常的出现。所以，可以在Provision中禁止组件前加入判断语句确保标记写入成功。

## 定制

通过Provision，可以定制：

1、加入一些初始设置项的设定，比如时区/时间初始设定，背景数据是否允许，是否允许安装非Android市场上的程序，等不需要用户干预就可以完成的设置。

2、加入UI设计，引导用户一步步完成需要用户参与选择或输入的设置过程，也就是设置向导的工作。

# Laucher

android7.x Launcher3

http://blog.csdn.net/picasso\_l/article/details/70140694

## ActivityManagerService是如何启动Launcher



在SystemServer中启动PackageManagerService和ActivityManagerService后，PackageManagerService主要负责安装系统的应用程序，并把保存应用程序信息在这里就不多阐述，本文主要讲解。

ActivityManagerService.systemReady()

ActivityStack.resumeTopActivityLocked()，

ActivityManagerService .startHomeActivityLocked()。

而在**startHomeActivityLocked()**中我们首先是读取查询intent中Category类型为HOME的Activity，这个信息是保存在PackageManagerService，我们可以通过方法intent.resolveActivityInfo(mContext.getPackageManager(),STOCK\_PM\_FLAGS)

而intent中ategory类型为HOME，在新版本的android中已经不在用这个方法了，而是用

调用ActivityManagerService中的resolveActivityInfo(),实际上是在resolveActivityInfo()调用了resolveIntent()调用了chooseBestActivity()，由于intent中Category类型为HOME的Activity可能有多个，若优先级(android:priority)相同系统会弹出提示框让用户选择，优先级(android:priority)不同，则会选择启动优先级高的Activity，



# 安装与卸载应用程序PackageInstaller

源码目录/package/apps/PackageInstaller，其主要的功能就是实现应用的安装和卸载功能。

## **mainfest.xml**

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"***?>* <**manifest xmlns:android="http://schemas.android.com/apk/res/android"  package="com.android.packageinstaller"**>   <**original-package android:name="com.android.packageinstaller"** />   <**uses-permission android:name="android.permission.INSTALL\_PACKAGES"** />  <**uses-permission android:name="android.permission.DELETE\_PACKAGES"** />  <**uses-permission android:name="android.permission.CLEAR\_APP\_CACHE"** />  <**uses-permission android:name="android.permission.READ\_PHONE\_STATE"** />  <**uses-permission android:name="android.permission.CLEAR\_APP\_USER\_DATA"** />  <**uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE"** />  <**uses-permission android:name="android.permission.MANAGE\_USERS"** />  <**uses-permission android:name="android.permission.GRANT\_REVOKE\_PERMISSIONS"** />  <**uses-permission android:name="android.permission.INTERACT\_ACROSS\_USERS\_FULL"** />  <**uses-permission android:name="android.permission.READ\_INSTALL\_SESSIONS"** />   <**application android:label="@string/app\_name"  android:allowBackup="false"  android:theme="@style/Theme.DialogWhenLarge"  android:supportsRtl="true"**>   <**activity android:name=".PackageInstallerActivity"  android:configChanges="orientation|keyboardHidden|screenSize"  android:excludeFromRecents="true"**>  <**intent-filter**>  <**action android:name="android.intent.action.VIEW"** />  <**action android:name="android.intent.action.INSTALL\_PACKAGE"** />  <**category android:name="android.intent.category.DEFAULT"** />  <**data android:scheme="file"** />  <**data android:mimeType="application/vnd.android.package-archive"** />  </**intent-filter**>  <**intent-filter**>  <**action android:name="android.intent.action.INSTALL\_PACKAGE"** />  <**category android:name="android.intent.category.DEFAULT"** />  <**data android:scheme="file"** />  <**data android:scheme="package"** />  </**intent-filter**>  <**intent-filter**>  <**action android:name="android.content.pm.action.CONFIRM\_PERMISSIONS"** />  <**category android:name="android.intent.category.DEFAULT"** />  </**intent-filter**>  </**activity**>   <**activity android:name=".InstallAppProgress"  android:configChanges="orientation|keyboardHidden|screenSize"  android:exported="false"** />   <**activity android:name=".UninstallerActivity"  android:configChanges="orientation|keyboardHidden|screenSize"  android:excludeFromRecents="true"  android:theme="@style/Theme.AlertDialogActivity"**>  <**intent-filter**>  <**action android:name="android.intent.action.DELETE"** />  <**action android:name="android.intent.action.UNINSTALL\_PACKAGE"** />  <**category android:name="android.intent.category.DEFAULT"** />  <**data android:scheme="package"** />  </**intent-filter**>  </**activity**>   <**activity android:name=".UninstallAppProgress"  android:configChanges="orientation|keyboardHidden|screenSize"  android:exported="false"** />   <**activity android:name=".GrantActivity"  android:configChanges="orientation|keyboardHidden|screenSize"  android:excludeFromRecents="true"  android:theme="@android:style/Theme.DeviceDefault.Dialog.NoActionBar"**>  <**intent-filter**>  <**action android:name="android.content.pm.action.REQUEST\_PERMISSION"** />  <**category android:name="android.intent.category.DEFAULT"** />  </**intent-filter**>  </**activity**>  </**application**> </**manifest**> |

从mainfest.xml文件，可以总结一些信息：

1）共定义了两对四个窗口。其中PackageInstallerActivity和InstallAppProgress用于安装应用程序;UninstallerActivity和UninstallAppProgress用于卸载应用程序.

2）intent-filter包含android.intent.action.MAIN的Activity Action的Activity会在系统程序列表中列出相应的应用图标。PackageInstaller中的Activity并**没有注册MAIN**的Action，图标**不会列在应用程序列表中**

3）PackageInstallerActivity采用隐示调用，包含了两个Intent Filter，也就是支持两种方式开启该PackageInstallerActivity；

|  |
| --- |
| Intent intent = new Intent(Intent.ACTION\_INSTALL\_PACKAGE);  intent.setDataAndType(Uri.fromFile(new File("/sdcard/qq.apk")),"application/vnd.android.package-archive");  startActivity(intent); |
| Intent intent = new Intent(Intent.ACTION\_INSTALL\_PACKAGE);  intent.setData(Uri.fromFile(new File("/sdcard/qq.apk")));  startActivity(intent); |

4）类似的，UninstallerActivity也有两种意图去启动

|  |
| --- |
| Intent intent = new Intent(Intent.ACTION\_DELETE);  intent.setData(Uri.parse("package:com.tencent.mobileqq"));  startActivity(intent); |
| Intent intent = new Intent(Intent.ACTION\_UNINSTALL\_PACKAGE);  intent.setData(Uri.parse("package:com.tencent.mobileqq"));  startActivity(intent); |

## 安装Android应用前的校验

上面我们知道安装应用时会调用PackageInstallerActivity弹出安装界面，在安装时，我们可以留意到，还会将要安装的应用所用到的权限以列表的形式进行展示，应用名称，应用图标等信息也会一一进行展示，所以我们要看看PckageInstallerActivity中是如何获取到这些信息的。在PackageInstallerActivity中主要完成的工作如下：

1).从Intent中获取Package URI，Scheme等信息。   
2).对从Intent对象获取的信息进行校验。主要校验Scheme   
3).根据Scheme的具体值（file或者package）进行相应的处理   
4).获取ApplicationInfo对象，该对象包含与Android应用相关的信息，如应用名称，应用图标，应用权限等   
5).初始化用于显示名称和应用图标的控件   
6).校验当前Android系统是否允许“未知来源”的应用被安装   
7).进行安装前的准备工作，显示校验窗口

### PackageInstallerActivity.onCreate

主要完成的功能：   
1. 从Intent对象获取Package URL、Scheme信息   
2. 校验Scheme，根据Scheme不同的值进行逻辑处理。Scheme为file或者package   
3. 获取ApplicationInfo对象，用于获取应用名称，报名，应用图标等信息

|  |
| --- |
| @Override **protected void** onCreate(Bundle icicle) {  **super**.onCreate(icicle);   **mPm** = getPackageManager();  **mInstaller** = **mPm**.getPackageInstaller();   **final** Intent intent = getIntent();  **if** (PackageInstaller.ACTION\_CONFIRM\_PERMISSIONS.equals(intent.getAction())) {  **final int** sessionId = intent.getIntExtra(PackageInstaller.***EXTRA\_SESSION\_ID***, -1);  **final** PackageInstaller.SessionInfo info = **mInstaller**.getSessionInfo(sessionId);  **if** (info == **null** || !info.sealed || info.resolvedBaseCodePath == **null**) {  Log.*w*(***TAG***, **"Session "** + **mSessionId** + **" in funky state; ignoring"**);  finish();  **return**;  }  **mSessionId** = sessionId;  *//获取待安装Android应用的路径或Package* **mPackageURI** = Uri.*fromFile*(**new** File(info.resolvedBaseCodePath));  **mOriginatingURI** = **null**;  **mReferrerURI** = **null**;  } **else** {  **mSessionId** = -1;  **mPackageURI** = intent.getData();  **mOriginatingURI** = intent.getParcelableExtra(Intent.***EXTRA\_ORIGINATING\_URI***);  **mReferrerURI** = intent.getParcelableExtra(Intent.***EXTRA\_REFERRER***);  }   **boolean** requestFromUnknownSource = isInstallRequestFromUnknownSource(intent);  **mInstallFlowAnalytics** = **new** InstallFlowAnalytics();  **mInstallFlowAnalytics**.setContext(**this**);  **mInstallFlowAnalytics**.setStartTimestampMillis(SystemClock.*elapsedRealtime*());  **mInstallFlowAnalytics**.setInstallsFromUnknownSourcesPermitted(  isInstallingUnknownAppsAllowed());  **mInstallFlowAnalytics**.setInstallRequestFromUnknownSource(requestFromUnknownSource);  **mInstallFlowAnalytics**.setVerifyAppsEnabled(isVerifyAppsEnabled());  **mInstallFlowAnalytics**.setAppVerifierInstalled(isAppVerifierInstalled());  **mInstallFlowAnalytics**.setPackageUri(**mPackageURI**.toString());   **final** String scheme = **mPackageURI**.getScheme();  *//从此处可以看到，scheme只有两个值：file或package，为空呢* **if** (scheme != **null** && !**"file"**.equals(scheme) && !**"package"**.equals(scheme)) {  Log.*w*(***TAG***, **"Unsupported scheme "** + scheme);  setPmResult(PackageManager.INSTALL\_FAILED\_INVALID\_URI);  **mInstallFlowAnalytics**.setFlowFinished(  InstallFlowAnalytics.***RESULT\_FAILED\_UNSUPPORTED\_SCHEME***);  finish();  **return**;  }   **final** PackageUtil.AppSnippet as;  *//scheme是package时* **if** (**"package"**.equals(**mPackageURI**.getScheme())) {  **mInstallFlowAnalytics**.setFileUri(**false**);  **try** {  *//获取与package对应的Android应用的信息,包含应用名称,权限列表，应用图标等信息* **mPkgInfo** = **mPm**.getPackageInfo(**mPackageURI**.getSchemeSpecificPart(),  PackageManager.***GET\_PERMISSIONS*** | PackageManager.***GET\_UNINSTALLED\_PACKAGES***);  } **catch** (NameNotFoundException e) {  }  **if** (**mPkgInfo** == **null**) {  Log.*w*(***TAG***, **"Requested package "** + **mPackageURI**.getScheme()  + **" not available. Discontinuing installation"**);  showDialogInner(***DLG\_PACKAGE\_ERROR***);  setPmResult(PackageManager.INSTALL\_FAILED\_INVALID\_APK);  **mInstallFlowAnalytics**.setPackageInfoObtained();  **mInstallFlowAnalytics**.setFlowFinished(  InstallFlowAnalytics.***RESULT\_FAILED\_PACKAGE\_MISSING***);  **return**;  }  *//创建AppSnippet对象。该对象封装了用于待安装Android应用的标题和图标* as = **new** PackageUtil.AppSnippet(**mPm**.getApplicationLabel(**mPkgInfo**.**applicationInfo**),  **mPm**.getApplicationIcon(**mPkgInfo**.**applicationInfo**));  } **else** {  *//scheme为file的情况，或者scheme为空，及从apk文件安装程序* **mInstallFlowAnalytics**.setFileUri(**true**);  *//获取APK文件的绝对路径* **final** File sourceFile = **new** File(**mPackageURI**.getPath());  *//创建APK文件的分析器* PackageParser.Package parsed = PackageUtil.*getPackageInfo*(sourceFile);   *// Check for parse errors* **if** (parsed == **null**) {  Log.*w*(***TAG***, **"Parse error when parsing manifest. Discontinuing installation"**);  showDialogInner(***DLG\_PACKAGE\_ERROR***);  setPmResult(PackageManager.INSTALL\_FAILED\_INVALID\_APK);  **mInstallFlowAnalytics**.setPackageInfoObtained();  **mInstallFlowAnalytics**.setFlowFinished(  InstallFlowAnalytics.***RESULT\_FAILED\_TO\_GET\_PACKAGE\_INFO***);  **return**;  }  **mPkgInfo** = PackageParser.generatePackageInfo(parsed, **null**,  PackageManager.***GET\_PERMISSIONS***, 0, 0, **null**,  **new** PackageUserState());  **mPkgDigest** = parsed.manifestDigest;  as = PackageUtil.*getAppSnippet*(**this**, **mPkgInfo**.**applicationInfo**, sourceFile);  }  **mInstallFlowAnalytics**.setPackageInfoObtained();    *//set view* setContentView(R.layout.install\_start);  **mInstallConfirm** = findViewById(R.id.install\_confirm\_panel);  **mInstallConfirm**.setVisibility(View.***INVISIBLE***);  PackageUtil.*initSnippetForNewApp*(**this**, as, R.id.app\_snippet);   **mOriginatingUid** = getOriginatingUid(intent);   *// 未知源安装检测* **if** ((requestFromUnknownSource) && (!isInstallingUnknownAppsAllowed())) {  *//ask user to enable setting first* showDialogInner(***DLG\_UNKNOWN\_APPS***);  **mInstallFlowAnalytics**.setFlowFinished(  InstallFlowAnalytics.***RESULT\_BLOCKED\_BY\_UNKNOWN\_SOURCES\_SETTING***);  **return**;  }  *//为安装应用做一些准备工作* initiateInstall(); } |

### startInstallConfirm

该方法的功能是显示一个确认对话框，该对话框会列出请求的权限列表；在startInstallConfirm方法中，涉及到了AppSecurityPermissions组件，另外涉及权限的还有三类：   
- AppSecurityPermissions.WHICH\_PERSONAL 涉及隐私的权限项   
- AppsecurityPermissions.WHICH\_DEVICE 涉及与设备相关的权限项   
- AppSecurityPermissions.WHICH\_NEW 涉及的新增的权限列表项 应用更新时如果有额外的新的权限增加会使用到该项。

|  |
| --- |
| **private void** startInstallConfirm() {  TabHost tabHost = (TabHost)findViewById(android.R.id.***tabhost***);  tabHost.setup();  ViewPager viewPager = (ViewPager)findViewById(R.id.pager);  TabsAdapter adapter = **new** TabsAdapter(**this**, tabHost, viewPager);  adapter.setOnTabChangedListener(**new** TabHost.OnTabChangeListener() {  @Override  **public void** onTabChanged(String tabId) {  **if** (***TAB\_ID\_ALL***.equals(tabId)) {  **mInstallFlowAnalytics**.setAllPermissionsDisplayed(**true**);  } **else if** (***TAB\_ID\_NEW***.equals(tabId)) {  **mInstallFlowAnalytics**.setNewPermissionsDisplayed(**true**);  }  }  });  *//该字段表示否时显示权限列表* **boolean** permVisible = **false**;  **mScrollView** = **null**;  **mOkCanInstall** = **false**;  **int** msg = 0;  **if** (**mPkgInfo** != **null**) {  *//AppSecurityPermissions 是一个组件 封装了一些列处理权限的功能* AppSecurityPermissions perms = **new** AppSecurityPermissions(**this**, **mPkgInfo**);  *//获取与隐私相关的权限数量* **final int** NP = perms.getPermissionCount(AppSecurityPermissions.WHICH\_PERSONAL);  *//获取与设备相关的权限数量* **final int** ND = perms.getPermissionCount(AppSecurityPermissions.WHICH\_DEVICE);  **if** (**mAppInfo** != **null**) {  msg = (**mAppInfo**.**flags** & ApplicationInfo.***FLAG\_SYSTEM***) != 0  ? R.string.install\_confirm\_question\_update\_system  : R.string.install\_confirm\_question\_update;  *//显示权限列表的ScrollView控件* **mScrollView** = **new** CaffeinatedScrollView(**this**);  **mScrollView**.setFillViewport(**true**);  *//当安装的应用已经存在时(更新应用时),获取是否有额外的权限请求 AppSecurityPermissions.WHICH\_NEW* **boolean** newPermissionsFound =  (perms.getPermissionCount(AppSecurityPermissions.WHICH\_NEW) > 0);  **mInstallFlowAnalytics**.setNewPermissionsFound(newPermissionsFound);  **if** (newPermissionsFound) {  permVisible = **true**;  **mScrollView**.addView(perms.getPermissionsView(  AppSecurityPermissions.WHICH\_NEW));  } **else** {  LayoutInflater inflater = (LayoutInflater)getSystemService(  Context.***LAYOUT\_INFLATER\_SERVICE***);  TextView label = (TextView)inflater.inflate(R.layout.label, **null**);  label.setText(R.string.no\_new\_perms);  **mScrollView**.addView(label);  }  adapter.addTab(tabHost.newTabSpec(***TAB\_ID\_NEW***).setIndicator(  getText(R.string.newPerms)), **mScrollView**);  } **else** {  findViewById(R.id.tabscontainer).setVisibility(View.***GONE***);  findViewById(R.id.divider).setVisibility(View.***VISIBLE***);  }  *//将要安装的应用设置的权限请求数量大于0时，将设置的权限列表列出来;* **if** (NP > 0 || ND > 0) {  permVisible = **true**;  LayoutInflater inflater = (LayoutInflater)getSystemService(  Context.***LAYOUT\_INFLATER\_SERVICE***);  View root = inflater.inflate(R.layout.permissions\_list, **null**);  **if** (**mScrollView** == **null**) {  **mScrollView** = (CaffeinatedScrollView)root.findViewById(R.id.scrollview);  }  *//向权限列表控件中添加 私有请求权限控件* **if** (NP > 0) {  ((ViewGroup)root.findViewById(R.id.privacylist)).addView(  perms.getPermissionsView(AppSecurityPermissions.WHICH\_PERSONAL));  } **else** {*//私有请求权限数量为0时，就将控件进行隐藏* root.findViewById(R.id.privacylist).setVisibility(View.***GONE***);  }  **if** (ND > 0) {  ((ViewGroup)root.findViewById(R.id.devicelist)).addView(  perms.getPermissionsView(AppSecurityPermissions.WHICH\_DEVICE));  } **else** {  root.findViewById(R.id.devicelist).setVisibility(View.***GONE***);  }  adapter.addTab(tabHost.newTabSpec(***TAB\_ID\_ALL***).setIndicator(  getText(R.string.allPerms)), root);  }  }  **mInstallFlowAnalytics**.setPermissionsDisplayed(permVisible);  **if** (!permVisible) {  **if** (**mAppInfo** != **null**) {  *// This is an update to an application, but there are no  // permissions at all.* msg = (**mAppInfo**.**flags** & ApplicationInfo.***FLAG\_SYSTEM***) != 0  ? R.string.install\_confirm\_question\_update\_system\_no\_perms  : R.string.install\_confirm\_question\_update\_no\_perms;  } **else** {  *// This is a new application with no permissions.* msg = R.string.install\_confirm\_question\_no\_perms;  }  tabHost.setVisibility(View.***GONE***);  **mInstallFlowAnalytics**.setAllPermissionsDisplayed(**false**);  **mInstallFlowAnalytics**.setNewPermissionsDisplayed(**false**);  findViewById(R.id.filler).setVisibility(View.***VISIBLE***);  findViewById(R.id.divider).setVisibility(View.***GONE***);  **mScrollView** = **null**;  }  **if** (msg != 0) {  ((TextView)findViewById(R.id.install\_confirm\_question)).setText(msg);  }  **mInstallConfirm**.setVisibility(View.***VISIBLE***);  **mOk** = (Button)findViewById(R.id.ok\_button);  **mCancel** = (Button)findViewById(R.id.cancel\_button);  **mOk**.setOnClickListener(**this**);  **mCancel**.setOnClickListener(**this**);  **if** (**mScrollView** == **null**) {  *// There is nothing to scroll view, so the ok button is immediately  // set to install.* **mOk**.setText(R.string.install);  **mOkCanInstall** = **true**;  } **else** {  *//如果设置了权限列表，则当滚动到权限列表末尾时，显示“install”按钮,否则显示“next"下一步* **mScrollView**.setFullScrollAction(**new** Runnable() {  @Override  **public void** run() {  **mOk**.setText(R.string.install);  **mOkCanInstall** = **true**;  }  });  } } |

## 安装应用程序

尽管安装Android应用前的校验工作非常复杂，但这些工作并不是PackageInstaller的主要工作。PackageInstaller的主要任务是安装android应用，而且是静默安装。如果不显示用于校验的窗口，PackageInstaller的安装过程可以让用户完全感觉不到，并且可以是后台异步进行。要想实现静默安装Android应用程序，首先我们应该了解PackageInstaller的安装原理。

1.PackageInstaller的安装原理：当我们浏览了权限列表后，点击安装，会出现一个进度条，提示我们正在安装。如果成功安装了应用，会显示应用已安装的提示信息。我们到PackageInstallerActivity的源码中查看源码是如何实现的,从onclick事件开始分析：

|  |
| --- |
| *//单击“next/ok“按钮浏览权限列表* **if**(v == **mOk**) {  *//已经浏览完所有权限 “next"按钮已经变成了“ok”按钮，已经准备好可以显示安装口* **if** (**mOkCanInstall** || **mScrollView** == **null**) {  **mInstallFlowAnalytics**.setInstallButtonClicked();  **if** (**mSessionId** != -1) {  **mInstaller**.setPermissionsResult(**mSessionId**, **true**);   *// We're only confirming permissions, so we don't really know how the  // story ends; assume success.* **mInstallFlowAnalytics**.setFlowFinishedWithPackageManagerResult(  PackageManager.INSTALL\_SUCCEEDED);  } **else** {  *// Start subactivity to actually install the application  //构造开启安装程序的Intent对象* Intent newIntent = **new** Intent();  newIntent.putExtra(PackageUtil.***INTENT\_ATTR\_APPLICATION\_INFO***,  **mPkgInfo**.**applicationInfo**);  newIntent.setData(**mPackageURI**);  *//将要显示的Activity类界面* newIntent.setClass(**this**, InstallAppProgress.**class**);  newIntent.putExtra(InstallAppProgress.***EXTRA\_MANIFEST\_DIGEST***, **mPkgDigest**);  newIntent.putExtra(  InstallAppProgress.***EXTRA\_INSTALL\_FLOW\_ANALYTICS***, **mInstallFlowAnalytics**);  String installerPackageName = getIntent().getStringExtra(  Intent.***EXTRA\_INSTALLER\_PACKAGE\_NAME***);  **if** (**mOriginatingURI** != **null**) {  newIntent.putExtra(Intent.***EXTRA\_ORIGINATING\_URI***, **mOriginatingURI**);  }  **if** (**mReferrerURI** != **null**) {  newIntent.putExtra(Intent.***EXTRA\_REFERRER***, **mReferrerURI**);  }  **if** (**mOriginatingUid** != VerificationParams.NO\_UID) {  newIntent.putExtra(Intent.EXTRA\_ORIGINATING\_UID, **mOriginatingUid**);  }  **if** (installerPackageName != **null**) {  newIntent.putExtra(Intent.***EXTRA\_INSTALLER\_PACKAGE\_NAME***,  installerPackageName);  }  **if** (getIntent().getBooleanExtra(Intent.***EXTRA\_RETURN\_RESULT***, **false**)) {  newIntent.putExtra(Intent.***EXTRA\_RETURN\_RESULT***, **true**);  newIntent.addFlags(Intent.***FLAG\_ACTIVITY\_FORWARD\_RESULT***);  }  **if**(**localLOGV**) Log.*i*(***TAG***, **"downloaded app uri="**+**mPackageURI**);  startActivity(newIntent);  }  finish(); |

上述方法就是事件驱动型的代码，只是事件的分类逻辑处理并不困难，最重要的就当是构造Intent启动安装窗口对象。

### InstallAppProgress

onCreate方法中主要就是获取在PackageInstallerActivity的点击方法中构造的intent对象，然后获取传递的构造的ApplicationInfo对象，将要安装的应用包的URI，获取将要安装的应用包的Scheme的具体对应的值（file或者package），最后调用初始化视图操作initview。主要分析initView方法如下：

|  |
| --- |
| **public void** initView() {  setContentView(R.layout.op\_progress);  **int** installFlags = 0;  PackageManager pm = getPackageManager();  **try** {  PackageInfo pi = pm.getPackageInfo(**mAppInfo**.**packageName**,   PackageManager.***GET\_UNINSTALLED\_PACKAGES***);  **if**(pi != **null**) {  *//如果应用已安装, 则设置应用的安装模式为更新* installFlags |= PackageManager.INSTALL\_REPLACE\_EXISTING;  }  } **catch** (NameNotFoundException e) {  }  **if**((installFlags & PackageManager.INSTALL\_REPLACE\_EXISTING )!= 0) {  Log.*w*(**TAG**, **"Replacing package:"** + **mAppInfo**.**packageName**);  }   **final** PackageUtil.AppSnippet as;  *//如果scheme为package，则意味着是更新程序* **if** (**"package"**.equals(**mPackageURI**.getScheme())) {  as = **new** PackageUtil.AppSnippet(pm.getApplicationLabel(**mAppInfo**),  pm.getApplicationIcon(**mAppInfo**));  } **else** {  *//否则scheme为file，则通过APK文件的路径获取显示Android应用相关信息的视图* **final** File sourceFile = **new** File(**mPackageURI**.getPath());  as = PackageUtil.*getAppSnippet*(**this**, **mAppInfo**, sourceFile);  }  **mLabel** = as.**label**;  PackageUtil.*initSnippetForNewApp*(**this**, as, R.id.app\_snippet);  **mStatusTextView** = (TextView)findViewById(R.id.center\_text);  **mStatusTextView**.setText(R.string.installing);  **mExplanationTextView** = (TextView) findViewById(R.id.center\_explanation);  **mProgressBar** = (ProgressBar) findViewById(R.id.progress\_bar);  **mProgressBar**.setIndeterminate(**true**);  *// Hide button till progress is being displayed* **mOkPanel** = (View)findViewById(R.id.buttons\_panel);  **mDoneButton** = (Button)findViewById(R.id.done\_button);  **mLaunchButton** = (Button)findViewById(R.id.launch\_button);  **mOkPanel**.setVisibility(View.***INVISIBLE***);   *//获取安装应用包的包名* String installerPackageName = getIntent().getStringExtra(  Intent.***EXTRA\_INSTALLER\_PACKAGE\_NAME***);  Uri originatingURI = getIntent().getParcelableExtra(Intent.***EXTRA\_ORIGINATING\_URI***);  Uri referrer = getIntent().getParcelableExtra(Intent.***EXTRA\_REFERRER***);  *//如上的两个Uri，对于普通的Anroid应用来说为null。* **int** originatingUid = getIntent().getIntExtra(Intent.EXTRA\_ORIGINATING\_UID,  VerificationParams.NO\_UID);  ManifestDigest manifestDigest = getIntent().getParcelableExtra(***EXTRA\_MANIFEST\_DIGEST***);  VerificationParams verificationParams = **new** VerificationParams(**null**, originatingURI,  referrer, originatingUid, manifestDigest);  PackageInstallObserver observer = **new** PackageInstallObserver();   **if** (**"package"**.equals(**mPackageURI**.getScheme())) {  *//scheme为package时，调用更新方法更新程序* **try** {  pm.installExistingPackage(**mAppInfo**.**packageName**);  observer.packageInstalled(**mAppInfo**.**packageName**,  PackageManager.INSTALL\_SUCCEEDED);  } **catch** (PackageManager.NameNotFoundException e) {  observer.packageInstalled(**mAppInfo**.**packageName**,  PackageManager.INSTALL\_FAILED\_INVALID\_APK);  }  } **else** {  *//scheme为file时,调用安装方法* pm.installPackageWithVerificationAndEncryption(**mPackageURI**, observer, installFlags,  installerPackageName, verificationParams, **null**);  } } |

initView方法一如既往的还是获取将要安装程序的一些信息，然后进行做了一些操作，该方法最重要的操作和调用发生在最后，如果scheme为package，则调用pm.installExistingPackage()方法更新应用，反之如果scheme为file，则调用pm.installPackageWithVerificationAndEncryption()方法进行应用的安装。上面说到的两个方法都是PakcageManager中的方法，**两个方法都是静默安装**，在安装的时候不会出现任何的提示。但是由于上述两个方法在PackageManager中是被注释为@hide的，所以，普通的Android应用中无法调用和访问。静默安装是一个异步过程，所以，无论安装成功或者安装失败，都会向用户弹出结果，所以我们在调用方法中看到有个行参是observer，为PackageInstallObserver实例。PackageInstallObserver主要就是处理安装的结果，其中定义了Handler变量mHandler来进行UI操作，用于实现对用户提示。

总结：从技术上来说，**实现静默安装Android应用就是调用PackageManager.installPackageWithVerificationAndEncryption方法即可**。该方法的参数较多，最重要的是需要一个异步安装结果监听器用于处理安装结果。此监听器必须是**IPackageInstallObserver.Stub**的子类。但**是只有安装结果，并没有具体的进度。**

最后还要说明的是：PackageInstaller安装应用程序实现安装应用的功能是需要申请权限：android.permission.INSTALL\_PACKAGES.该权限属于**系统级别的权限**，在普通的应用中无法使用。

## 卸载Android应用

和安装应用时一样，在卸载应用时，PackageInstaller会首先弹出卸载确认提示框，用户点击确认后，再显示卸载进度提示框，最后显示卸载结果.

### showDialogFragment

卸载前的确认框是一个Dialog，在5.1.1源码中为showDialogFragment(new UninstallAlertDialogFragment());

在UninstallAlertFragment中获取要卸载的应用的名字，图标等信息，然后设置title，icon，PositiveButton和NegativeButton事件等。PositiveButton的点击事件会触发UnInstallerActivity中的startUninstallProgress()方法，如下：

### startUninstallProgress

|  |
| --- |
| void startUninstallProgress() {  //构造要卸载的应用的Intent对象  Intent newIntent = new Intent(Intent.*ACTION\_VIEW*);  newIntent.putExtra(Intent.*EXTRA\_USER*, mDialogInfo.user);  newIntent.putExtra(Intent.EXTRA\_UNINSTALL\_ALL\_USERS, mDialogInfo.allUsers);  newIntent.putExtra(PackageInstaller.EXTRA\_CALLBACK, mDialogInfo.callback);  newIntent.putExtra(PackageUtil.*INTENT\_ATTR\_APPLICATION\_INFO*, mDialogInfo.appInfo);  if (getIntent().getBooleanExtra(Intent.*EXTRA\_RETURN\_RESULT*, false)) {  newIntent.putExtra(Intent.*EXTRA\_RETURN\_RESULT*, true);  newIntent.addFlags(Intent.*FLAG\_ACTIVITY\_FORWARD\_RESULT*);  }  newIntent.setClass(this, UninstallAppProgress.class);  startActivity(newIntent); } |

### UninstallAppProgress

展示卸载进度界面,与InstallAppProgress类类似的，onCreate方法中获取了传递的Intent对象中的ApplicationInfo对象以及获取是否删除所有用户数据的标志，重要的卸载方法调用还是在initView方法中，核心调用如下：

|  |
| --- |
| IPackageManager packageManager =  IPackageManager.Stub.asInterface(ServiceManager.getService("package")); PackageDeleteObserver observer = new PackageDeleteObserver(); try {// 静默卸载Android应用   packageManager.deletePackageAsUser(mAppInfo.packageName, observer,  mUser.getIdentifier(),  mAllUsers ? PackageManager.DELETE\_ALL\_USERS : 0); } catch (RemoteException e) {  // Shouldn't happen.  Log.*e*(TAG, "Failed to talk to package manager", e); } |

同样的，deletePackageAsUser是PackageManager类中标注为@hide的方法，在普通的应用中无法访问和调用，需要是系统应用才能调用。另外，PackageDeleteObserver卸载监听器需要继承IPackageDeleteObserver.Stub，这个也是使用了Handler的变量mHandler来处理卸载结果，卸载成功或者卸载失败的UI操作由此mHandler变量来实现。当然和安装应用时一样，需要系统级别的权限申请：android.permission.DELETE\_PACKAGES.

## 小结

普通的Android应用程序不允许静默安装和静默卸载程序，如果想要实现静默安装或者静默卸载需要将Android应用变成系统应用（进行系统签名）

在技术实现上，静默安装调用的是PackageManager类中的installPackageWithVerificationAndEncryption方法，然后通过一个安装监听器来处理安装结果，Handler变量根据安装结果处理UI显示；静默卸载调用的是PackageManager类中的deletePackageAsUser方法，通过设置一个卸载监听器监听卸载结果，Handler变量根据卸载结果处理UI显示。

* + 静默安装和静默卸载的方法都是标注为@hide的方法，普通应用中无法访问和调用
  + 安装应用需要使用系统级别的android.permission.INSTALL\_PACAGES权限；
  + 卸载应用需要使用系统级别的android.permission.DELETE\_PACKAGES权限。

## REF

# SettingsProvider

Content Provider几乎已经成为了Android系统中两个或多个应用(系统应用和普通应用)之间共享数据库的最直接的方法。

在Android源码中，有两个目录是专门存放Content Provider。

一个是/源码目录/frameworks/providers，允许任何Adroid应用访问，包括：ApplicationProvider、CalendarProvider、ContactsProvider、DownloadProvider、MedialProvider、PartnerBookmarksProvider、TelephonyProvider、TvProvider、UserDictionaryProvider；

另一个是/源码目录/frameworks/base/packages，通常只允许系统级别的Android应用读写数据，**普通Android应用只能读数据**。包含：ExternalStorageProvider、SettingsProvider

具体到Setting Content Provider 是所有系统设置的管理者。在M(Android5.0)版本之前，SettingsProvider中系统设置是存储在settings.db[数据库](http://lib.csdn.net/base/mysql)中；但是在L(Android6.0)之后，SettingsProvider中系统设置改为由xml存储在data分区。

如Wifi状态，蓝牙是否开启，当前设备所处的模式等等，很多很多类似的全局属性控制的属性值。最重要的是global表，该表包含两个字段：name和value.其中name表示系统信息的key，value表示该系统设置项所对应的选项值。数据库结构如下：



## Android M之前的SettingsProvider

进入**/data/data/com.**[**android**](http://lib.csdn.net/base/android)**.providers.settings/databases**

# cd /data/data/com.android.providers.settings/databases

# sqlite3 settings.db

sqlite3 有可能提示找不到，这是由于在当前版本中sqlite3程序没有被编译到system中，这里我们只需要手动push一个sqlite3就好了。Cs果然没有

在使用sqlite3打开settings.db数据库之后，可以使用.help来查看有关sqlite3的一些帮助信息。这里常用的如下所示：



源码

**frameworks/base/packages/SettingsProvider/src/com/Android/providers/settings/DatabaseHelper.**[**Java**](http://lib.csdn.net/base/java)   
**frameworks/base/packages/SettingsProvider/res/values/defaults.xml**

**如何添加属性：**系统属性的设置，我们可以通过修改源码，编译生成SettingsProvider.apk并安装到系统中，就能生效。

### [自定义Android系统默认设置](http://www.cnblogs.com/l2rf/p/5102230.html)

无法在普通app上通过以下方式使用，即使申请了android.permission.WRITE\_SETTINGS等权限也不行。会报SecurityException。

Settings.Global.*putString*(getContentResolver(),  
 **"myKey"**,**"v1"**);  
String v1 = Settings.Global.*getString*(getContentResolver(),  
 **"myKey"**);

### 修改方法默认值方法1

**adb shell settings**,有关其详细用法，可以参照如下提示：

$ adb shell settings get system ringtone

content://media/internal/audio/media/59

$ adb shell settings set system ringtone "content://media/internal/audio/media/60"

Invalid command: set

usage: settings [--user NUM] get namespace key

settings [--user NUM] put namespace key value

settings [--user NUM] delete namespace key

'namespace' is one of {system, secure, global}, case-insensitive

If '--user NUM' is not given, the operations are performed on the owner user.

$ adb shell settings put system ringtone "content://media/internal/audio/media/60"

$ adb shell settings get system ringtone

content://media/internal/audio/media/60

### 修改方法默认值方法2,标准方法

如果要在系统中添加默认设置，需要如下步骤（以带物理键盘的系统中默认打开软键盘为例）：

1、在defaults.xml中为新加的数据定义一个默认值

<integer name="show\_ime\_with\_hard\_keyboard">1</integer>

2、在DatabaseHelper.java中的loadSecureSettings(SQLiteDatabase db)方法中加入新字段的添加代码

loadStringSetting(stmt, Settings.Secure.SHOW\_IME\_WITH\_HARD\_KEYBOARD,

R.string.show\_ime\_with\_hard\_keyboar);

3、重新编译

## Android L 的SettingsProvider

在Android L版本中，Google团队为了优化系统性能、提高安全性等等一些列需求，重构了SettingsProvider模块，在L版本中，不在使用数据库来存储系统设置，而是通过xml将系统设置存储在了**/data/system/user/** 目录下



有关L版本中SettingsProvider的修改更新方法同之前的相同

## 源码

### manifest

|  |
| --- |
| <manifest xmlns:android="http://schemas.android.com/apk/res/android"  package="com.android.providers.settings"  coreApp="true"  android:sharedUserId="android.uid.system">  <application android:allowClearUserData="false"  android:label="@string/app\_label"  android:process="system"  android:backupAgent="SettingsBackupAgent"  android:killAfterRestore="false"  android:icon="@mipmap/ic\_launcher\_settings">    <!-- todo add: android:neverEncrypt="true" -->  <provider android:name="SettingsProvider" android:authorities="settings"  android:multiprocess="false"  android:exported="true"  android:writePermission="android.permission.WRITE\_SETTINGS"  android:singleUser="true"  android:initOrder="100" />  </application>  </manifest> |

通过如上文件内容，我们大概可以得出几个结论：   
1. 拥有system用户权限。标签的coreApp属性是true,且android:sharedUserId=”android.uid.system”意味着SettingProvder与Settings同样拥有system用户的权限，且通过android:process=”system”可以看出，SettingProvider运行在**系统进程中**。   
2. provider标签中android:authorities=”settings”可以知道，访问SettingsProvider的Uri的开头部分一定是“content://settings”;

比如

CONTENT\_URI = Uri.parse("content://settings/global");

3. provider标签中android:writePermission=”android.permission.WRITE\_SETTINGS”可以知道，在执行写操作时需要指定WRITE\_SETTINGS权限。

### Src

在SettingsProvider的src源代码中，共有四个类：   
1. DatabaseHelper类。数据库操作类，实现对setting.db的读写功能，继承自SQLiteOpenHelper类   
2. SettingsBackupAgent类。用于备份和恢复系统设置等功能的类，继承自BackupAgentHelper类   
3. SettingsHelper类。Settings的帮助类   
4. SettingsProvider类。继承自ContentProvider类的最核心的文件   
在这四个类中，最最重要的就是SettingsProvider，其次我关心的可能就是DatabaseHelper类是如何执行数据库语句的。

#### DatabaseHelper

首先，来分析DatabaseHelper类，在DatabaseHelper类中，就是对数据库及表的操作。主要是表的创建，数据库版本更新。onCreate方法中负责创建表，执行表创建语句，然后进行表的初始化，也就是系统的初始设置；onUpgrade方法最长，需要从第一个版本到最新版本一个一个判断，升级。我们看一下具体的初始设置：

|  |
| --- |
| //onCreate中的初始化设置调用  // Load initial volume levels into DB  loadVolumeLevels(db);  // Load inital settings values  loadSettings(db);  //下面是loadSettings方法：  private void loadSettings(SQLiteDatabase db) {  loadSystemSettings(db);  loadSecureSettings(db);  // The global table only exists for the 'owner' user  if (mUserHandle == UserHandle.USER\_OWNER) {  loadGlobalSettings(db);  }  }  //我们选择loadGlobalSettings进行查看，下面分别是设置boolean值，String值，int值  // --- Previously in 'system'  loadBooleanSetting(stmt, Settings.Global.AIRPLANE\_MODE\_ON,  R.bool.def\_airplane\_mode\_on);  loadStringSetting(stmt, Settings.Global.AIRPLANE\_MODE\_RADIOS,  R.string.def\_airplane\_mode\_radios);  loadIntegerSetting(stmt, Settings.Global.WIFI\_SLEEP\_POLICY,  R.integer.def\_wifi\_sleep\_policy); |

通过上面的代码我们看到，最后其实就是引用（或者说是读取）了integer或者bool再或者string中的一些具体的字段，设置到数据库中。我们要想查看这些默认的值，可以到res/values中进行查看。和本地化无关的存放在res/values/default.xml文件中，和本地化有关的（主要指string,涉及语言）则放在相应的语言分包下面；另外，我们还能注意到的是，sqlite不支持boolean值，如果要保存boolean值，是通过integer来保存的。一般1为true,0为false。

#### SettingsProvider

看onCreate（）方法，创建表和各表的索引，然后调用了loadBookmarks(db); loadVolumeLevels(db); loadSettings(db); 三个方法，loadBookmarks(db);是存系统几个常用的应用的intent ( 如，通讯录app，emailApp, 短信app,)，loadVolumeLevels(db); 初始化的是与声音相关的配制， loadSettings(db);方法就是从defaults.xml文件中读取默认值存入相应的表中，验证了之前 的猜测。

loadSettings(db); 里面的三个方法者是从defaults.xml文件中读取默认值存入相应的表中，loadSystemSettings(db)是存入System表中，loadSecureSettings(db)是存入 secure表中。

下面来分析最最核心的SettingsProvider的代码，核心就是insert,delete,update,query四个方法，我们依次看四个方法：

insert方法：insert方法中调用了insertForUser(Uri url, ContentValues initialValues, int desiredUserHandle) 方法。在insert方法中在执行insert方法之前，调用了检查写权限的方法checkWritePermissions，更新db都需要android.Manifest.permission.WRITE\_SECURE\_SETTINGS权限（需要系统签名）和android.Manifest.permission.WRITE\_、SETTINGS。读取则不需要任何权限。

|  |
| --- |
| **private void** checkWritePermissions(SqlArguments args) {  **if** ((TABLE\_SECURE.equals(args.table) || TABLE\_GLOBAL.equals(args.table)) &&  getContext().checkCallingOrSelfPermission(  android.Manifest.permission.WRITE\_SECURE\_SETTINGS) !=  PackageManager.PERMISSION\_GRANTED) {  **throw new** SecurityException(  String.format(**"Permission denial: writing to secure settings requires %1$s"**,  android.Manifest.permission.WRITE\_SECURE\_SETTINGS));  } } |

delete和update方法中也调用了如上的权限检查方法，然后再执行相应的数据库操作方法

query方法：query方法中调用了queryForUser()方法进而执行query查询返回Cursor结果对象。

## settings.db 详解

globl

主意：

正常的

root@zs600b:/system/priv-app # ls -al | grep Set

drwxr-xr-x root root 2017-07-06 11:32 Settings

drwxr-xr-x root root 2017-07-06 11:30 SettingsProvider

异常：

root@zs600b:/system/priv-app # ls -al | grep Set

drwxr-xr-x root root 2017-08-29 21:42 Settings

drwxr-xr-x root root 2017-08-29 21:40 SettingsProvider

-rw-r--r-- root root 126118 2017-08-31 21:35 SettingsProvider.apk

Log：

09-01 10:10:20.509 I/PackageManager( 486): Package com.android.providers.settings at /system/priv-app/SettingsProvider.apk ignored: updated version 22 better than this 22

09-01 10:10:20.509 W/PackageManager( 486): Code path for hidden system pkg : com.android.providers.settings changing from /system/priv-app/SettingsProvider to /system/priv-app/SettingsProvider.apk

09-01 10:10:20.509 W/PackageManager( 486): Failed to parse /system/priv-app/SettingsProvider.apk: null

root@zs600b:/system/priv-app/SettingsProvider/arm # ls -al

-rw-r--r-- root root 254440 2017-08-29 21:40 SettingsProvider.odex

09-01 10:47:21.478 W/art ( 489): Failed to find OatDexFile for DexFile /system/priv-app/SettingsProvider/SettingsProvider.apk ( canonical path /system/priv-app/SettingsProvider/SettingsProvider.apk) with checksum 0x6ade6c23 in OatFile /system/priv-app/SettingsProvider/arm/SettingsProvider.odex

09-01 10:47:21.504 I/dex2oat ( 549): Error opening the directory: /system/lib/plugins

09-01 10:47:21.505 I/dex2oat ( 549): /system/bin/dex2oat --runtime-arg -classpath --runtime-arg --instruction-set=arm --instruction-set-features=div --runtime-arg -Xrelocate --boot-image=/system/framework/boot.art --dex-file=/system/priv-app/SettingsProvider/SettingsProvider.apk --oat-fd=76 --oat-location=/data/dalvik-cache/arm/system@priv-app@SettingsProvider@SettingsProvider.apk@classes.dex --runtime-arg -Xms64m --runtime-arg -Xmx512m

adb install –r

INSTALL\_FAILED\_ALREADY\_EXISTS

means you uninstall before install -r remove purpose

## 实例分析

在framework/base/core/java定义成另一个了

Settings.Secure.putInt(getContext().getContentResolver(), Settings.Secure.LOCATION\_MODE, mode);

## 小结：Get/set方法

### 脚本命令settings

settings get system ringtone

settings get global ringtone

## Ref

[**Android SettingProvider详解**](http://blog.csdn.net/peng_cao/article/details/50887509)

[Settings.Global](https://developer.android.com/reference/android/provider/Settings.Global.html)

# Settings

系统设置是Android系统中非常重要的系统应用，也是整个android系统的控制中枢，关于设备的硬件，状态，软件，安全等都需要通过系统设置进行控制和查看。比如wifi状态，网络连接状。具体的功能是SettingProvider造作数据库实现的。

特别的，系统设置并不同于大多数的其他系统应用，系统设置不仅拥有platform签名（即系统签名），而且属于内核应用，所以系统设置要比非内核应用的系统应用有更大的权限。系统签名方法见第1章。

|  |
| --- |
| <activity android:name="Settings"  android:taskAffinity="com.android.settings"  android:label="@string/settings\_label\_launcher"  android:launchMode="singleTask">  <intent-filter>  <action android:name="android.settings.SETTINGS" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  <meta-data android:name="com.android.settings.PRIMARY\_PROFILE\_CONTROLLED"  android:value="true" /> </activity> |

在5.1.1的Settings源码中,Setting extends SettingActivity.查看SettingActivity发现，在SettingActivity中加载了设置的主界面布局，名为dashboard\_categories，文件目录为项目目录/res/xml/dashboard\_categories文件，布局文件中使用的布局标签为以及列表项,SettingsActivity中加载该布局文件的方法为loadCategoriesFromResource(),该方法存在于buildDashboardCategories(List categories)方法中,其中的行参就是所包含的设置列表项,加载该xml布局的方法就是解析xml解析器，对布局文件进行解析，现将该方法贴在此处，顺便回顾一下解析xml文件的方法：

如上的加载和解析的方法其实是在onResume方法中调用的,在onResume方法中调用了invalidateCategories方法，在invalidateCategories方法中我们会发现其实是采用了回发消息然后处理的消息处理机制，在此方法中发送了MSG\_BUILD\_CATEGORIES消息，在类声明处定义的名为mHandler的Handler中进行处理，随即调用buildDashboardCategories方法.

下面我们重新说回设置界面，上面说到解析和加载xml文件及相关数据展示，下面设计到的就是我们点击每一项时

## 网络

### 数据流量

入口类:DataUsageSummary

Kw: DataUsageSummary 源码

## WIFI选项

Wifi是系统设置中的第一项功能，通过该功能可以进行“关闭/打开WIFI”、扫描可用网络，连接网络等操作，setWifiApEnabled需要系统权限@SystemApi。核心代码如下：

|  |
| --- |
| @Override **public void** onCheckedChanged(CompoundButton switchView, **boolean** isChecked) {  DJILogUtil.*E*(**TAG**, **"onCheckedChanged,isChecked="** + isChecked);  *//Do nothing if called as a result of a state machine event* setSwitchBarEnabled(**false**);  **if** (**mStateMachineEvent**) {//避免循环调用，造成死循环  **return**;  }  *// Disable tethering if enabling Wifi。关闭热点* **int** wifiApState = **mWifiManager**.getWifiApState();  **if** (isChecked && ((wifiApState == WifiManager.WIFI\_AP\_STATE\_ENABLING) ||  (wifiApState == WifiManager.WIFI\_AP\_STATE\_ENABLED))) {  **mWifiManager**.setWifiApEnabled(**null**, **false**);  }   **if** (!**mWifiManager**.setWifiEnabled(isChecked)) {//打开wifi  *// Error* **mSwitchBar**.setEnabled(**true**);  Toast.*makeText*(**mContext**, **"wifi error"**, Toast.***LENGTH\_SHORT***).show();  } } |

**飞行模式**的初衷是为了在飞机飞行状态下一次性关闭所有可以发出无线信号的设备而加入的功能。这些//设备主要包括3G/2G, Wi-Fi、蓝牙、NFC等设备。

热点(Access Point)和Wi-Fi不能共存

### SoftAp

市面上大多数手机支持SoftAp功能, 系统的Setting程序TetherSettings.java将响应此动作。tether这个词意思是拴绳，拴住的意思，理解起来有点晦涩，这里我的理解是分享的意思，比如 WIFI\_TETHERING(用WIFI分享网路),  USB\_TETHERING（用USB分享网络）……

那么这些网络分享如何满足什么条件才开启呢？疑问

### 从WirelessSettings开始说

在packages/apps/Settings/src/com/android/settings/WirelessSettings.java 的onCreate方法中会去检测系统是否Tether

1. ConnectivityManager cm =
2. (ConnectivityManager) activity.getSystemService(Context.CONNECTIVITY\_SERVICE);
3. **if** (isSecondaryUser || !cm.isTetheringSupported()) {
4. getPreferenceScreen().removePreference(findPreference(KEY\_TETHER\_SETTINGS));
5. } **else** {
6. Preference p = findPreference(KEY\_TETHER\_SETTINGS);
7. p.setTitle(Utils.getTetheringLabel(cm));
8. }
9. protectByRestrictions(KEY\_TETHER\_SETTINGS);

1）调用获取ConnectivityManager实例对象，并调用isTetheringSupported方法来检测是否支持Tether

2）调用getTetheringLabel查看支持哪些网络共享方式，并显示到界面

先看isTetheringSupported方法

1. **public** boolean isTetheringSupported() {
2. **try** {
3. **return** mService.isTetheringSupported();
4. } **catch** (RemoteException e) {
5. **return** **false**;
6. ...

它调用了frameworks/base/services/core/java/com/android/server/ConnectivityService.java 的isTetheringSupported 方法

**[cpp]** [view plain](http://blog.csdn.net/jshazk1989/article/details/17094415) [copy](http://blog.csdn.net/jshazk1989/article/details/17094415)

1. **public** boolean isTetheringSupported() {
2. enforceTetherAccessPermission();
3. **int** defaultVal = (SystemProperties.get("ro.tether.denied").equals("true") ? 0 : 1);
4. boolean tetherEnabledInSettings = (Settings.Global.getInt(mContext.getContentResolver(),
5. Settings.Global.TETHER\_SUPPORTED, defaultVal) != 0);
6. **return** tetherEnabledInSettings && ((mTethering.getTetherableUsbRegexs().length != 0 ||
7. mTethering.getTetherableWifiRegexs().length != 0 ||
8. mTethering.getTetherableBluetoothRegexs().length != 0) &&
9. mTethering.getUpstreamIfaceTypes().length != 0);
10. }

 读取系统ro.tether.denied的属性，返回true则表示关闭tether功能，返回false表示打开tether功能。（有些设备不能显示portable wifi 就是因为此属性返回true,这里需要修改设置为false）

之后会检测是否有配置文件配置了tether，有则使用设置的值，没有查询到配置这使用查询ro.tether.denied属性的默认值。

这里我们假设得到的tetherEnableInSettings为ture，也不意味着就能正确打开Portable Wi-Fi hostpot 界面。因为还需要检测是否支持WIFI网络共享配置

mTethering.getTetherableWifiRegexs().length!=0 检测是否支持网络共享配置，这里Android4.4 与之前的版本有一定区别，但总体不变，大家可以参考其他代码做一些变通

### getTetherableWifiRegexs()方法

frameworks/base/services/java/com/android/server/connectivity/Tethering.java

**[cpp]** [view plain](http://blog.csdn.net/jshazk1989/article/details/17094415) [copy](http://blog.csdn.net/jshazk1989/article/details/17094415)

1. **return** mTetherableWifiRegexs;

直接返回了一个变量。 此变量在frameworks/base/services/java/com/android/server/connectivity/Tethering.java

updateConfiguration()方法中赋值

**[cpp]** [view plain](http://blog.csdn.net/jshazk1989/article/details/17094415) [copy](http://blog.csdn.net/jshazk1989/article/details/17094415)

1. mTetherableWifiRegexs = tetherableWifiRegexs;

而 tetherableWifiRegexs变量赋值也在此方法中

**[cpp]** [view plain](http://blog.csdn.net/jshazk1989/article/details/17094415) [copy](http://blog.csdn.net/jshazk1989/article/details/17094415)

1. String[] tetherableWifiRegexs = mContext.getResources().getStringArray(
2. com.android.internal.R.array.config\_tether\_wifi\_regexs);

这里的config\_tether\_wifi\_regexs数组由配置文件生成

### 配置文件config.xml

frameworks/base/core/res/res/values/config.xml

**[html]** [view plain](http://blog.csdn.net/jshazk1989/article/details/17094415) [copy](http://blog.csdn.net/jshazk1989/article/details/17094415)

1. **<span** style="font-size:14px;"**>**    <!-- List of regexpressions describing the interface (if any) that represent tetherable
2. Wifi interfaces.  If the device doesn't want to support tethering over Wifi this
3. should be empty.  An example would be "softap.\*" --**>**
4. **<string-array** translatable="false" name="config\_tether\_wifi\_regexs"**>**
5. **</string-array>**
6. **</span>**

这里看到需要在此默认的config.xml中的   name="config\_tether\_wifi\_regexs" 子标签下添加用作softap的接口(一般为wlan0 或者eth0 视你的WIFI驱动程序而定)

修改后的标签：

**[html]** [view plain](http://blog.csdn.net/jshazk1989/article/details/17094415) [copy](http://blog.csdn.net/jshazk1989/article/details/17094415)

1. **<span** style="font-size:14px;"**>**    <!-- List of regexpressions describing the interface (if any) that represent tetherable
2. Wifi interfaces.  If the device doesn't want to support tethering over Wifi this
3. should be empty.  An example would be "softap.\*" --**>**
4. **<string-array** translatable="false" name="config\_tether\_wifi\_regexs"**>**
5. **<item>**"wlan0"**</item>**
6. **</string-array>**
7. **</span>**

OK 此处mTethering.getTetherableWifiRegexs().length!=0 返回true， 整个isTetheringSupported 返回真

## 蓝牙

## 流量使用情况

## 恢复出厂设置

最近看恢复出厂的一个问题，以前也查过这方面的流程，所以这里整理一些AP+framework层的流程；

入口：setting-->备份与重置--->恢复出厂设置--->重置手机--->清除全部内容：**自动执行-->手机关机--->开机--->进行恢复出厂的操作--->开机流程**；

App层: sendBroadcast(new Intent("Android.intent.action.MASTER\_CLEAR"));

Adb层: adb shell am broadcast -a android.intent.action.MASTER\_CLEAR

发出了这个广播，后台server会完成后续操作。主意应用程序要有android.permission.MASTER\_CLEAR 这个权限

## 日期和时间

com.android.settings. DateTimeSettings

((AlarmManager) context.getSystemService(Context.ALARM\_SERVICE)).setTime(when);

IAlarmManager mService.setTime(millis);

## 跳转到各个系统设置的方法

// 第一个参为包名，第二个各个设置的类名(可以参考下面，包名不用改变)

ComponentName cm = new ComponentName("com.android.settings",

"com.android.settings.RadioInfo");

intent.setComponent(cm);

intent.setAction("android.intent.action.VIEW");

startActivity(intent);

### 跳转到各个系统设置的列表

com.android.settings.AccessibilitySettings 辅助功能设置

　　com.android.settings.ActivityPicker 选择活动

　　com.android.settings.ApnSettings APN设置

　　com.android.settings.ApplicationSettings 应用程序设置

　　com.android.settings.BandMode 设置GSM/UMTS波段

　　com.android.settings.BatteryInfo 电池信息

　　com.android.settings.DateTimeSettings 日期和坝上旅游网时间设置

　　com.android.settings.DateTimeSettingsSetupWizard 日期和时间设置

　　com.android.settings.DevelopmentSettings 应用程序设置=》开发设置

　　com.android.settings.DeviceAdminSettings 设备管理器

　　com.android.settings.DeviceInfoSettings 关于手机

　　com.android.settings.Display 显示——设置显示字体大小及预览

　　com.android.settings.DisplaySettings 显示设置

　　com.android.settings.DockSettings 底座设置

　　com.android.settings.IccLockSettings SIM卡锁定设置

　　com.android.settings.InstalledAppDetails 语言和键盘设置

　　com.android.settings.LanguageSettings 语言和键盘设置

　　com.android.settings.LocalePicker 选择手机语言

　　com.android.settings.LocalePickerInSetupWizard 选择手机语言

　　com.android.settings.ManageApplications 已下载（安装）软件列表

　　com.android.settings.MasterClear 恢复出厂设置

　　com.android.settings.MediaFormat 格式化手机闪存

　　com.android.settings.PhysicalKeyboardSettings 设置键盘

　　com.android.settings.PrivacySettings 隐私设置

　　com.android.settings.ProxySelector 代理设置

　　com.android.settings.RadioInfo 手机信息

　　com.android.settings.RunningServices 正在运行的程序（服务）

　　com.android.settings.SecuritySettings 位置和安全设置

　　com.android.settings.Settings 系统设置

　　com.android.settings.SettingsSafetyLegalActivity 安全信息

　　com.android.settings.SoundSettings 声音设置

　　com.android.settings.TestingSettings 测试——显示手机信息、电池信息、使用情况统计、Wifi information、服务信息

　　com.android.settings.TetherSettings 绑定与便携式热点

　　com.android.settings.TextToSpeechSettings 文字转语音设置

　　com.android.settings.UsageStats 使用情况统计

　　com.android.settings.UserDictionarySettings 用户词典

　　com.android.settings.VoiceInputOutputSettings 语音输入与输出设置

　　com.android.settings.WirelessSettings 无线和网络设置

参考自：<http://blog.csdn.net/aikongmeng/article/details/44748841>

# 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。 |

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 的开机启动

#### 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()

#### 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. }

#### 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. }

#### 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. };

### 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

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（好像 4.4 之后 android 原生的 UI 都是用这个了，想看 Tablet UI 的可以拿 4.2 玩一下） 来说明一下 addNotification 的流程。

### SystemBars的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的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. }
10. }

启动时序图：



### BaseStatusBar 的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,
3. ExpandableNotificationRow.ExpansionLogger, NotificationData.Environment,
4. ExpandableNotificationRow.OnExpandClickListener,
5. OnGutsClosedListener {
6. **public** **void** start() {
7. /\*mProvisioningOberver是一个ContentObserver。
8. 它负责监听Settings.Global.DEVICE\_PROVISIONED设置的变化（Provision.apk改变）。mProvisioningObserver 即是用来监听设备归属状态的变化，以禁用或启用某些功能 \*/
9. mProvisioningObserver.onChange(**false**); // set up
10. mContext.getContentResolver().registerContentObserver(
11. Settings.Global.getUriFor(Settings.Global.DEVICE\_PROVISIONED), **true**,
12. mProvisioningObserver);
13. /\* // 向 SM 获取 SBMS：实例化IStatusBarService,
14. 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。
15. IStatusBarService会保存SystemUi的状态信息,避免SystemUi崩溃而造成信息的丢失
16. \*/
17. mBarService = IStatusBarService.Stub.asInterface(
18. ServiceManager.getService(Context.STATUS\_BAR\_SERVICE));
19. //IStatusBarService与BaseStatusBar进行通信的桥梁。
20. mCommandQueue = **new** CommandQueue(**this**);
22. /\*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
23. 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。
24. \*/
25. **int**[] switches = **new** **int**[9];
26. ArrayList<IBinder> binders = **new** ArrayList<IBinder>();
27. /\*它保存了用于显示在状态栏的系统状态
28. 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
29. 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。
30. \*/
31. ArrayList<String> iconSlots = **new** ArrayList<>();
32. ArrayList<StatusBarIcon> icons = **new** ArrayList<>();
34. Rect fullscreenStackBounds = **new** Rect();
35. Rect dockedStackBounds = **new** Rect();
37. //调用 SBMS 的注册接口注册：IStatusBarService注册一些信息
38. mBarService.registerStatusBar(mCommandQueue, iconSlots, icons, switches, binders,
39. fullscreenStackBounds, dockedStackBounds);
41. //创建状态栏窗口
42. createAndAddWindows();
43. }
44. }

/\* 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端将会保存在 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。

### StatusBarManagerService

/services/core/java/com/android/server/statusbar/StatusBarManagerService.java

它的注册方法做一些数据的初始化

**[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. **public** **void** registerStatusBar(IStatusBar bar, List<String> iconSlots,
3. List<StatusBarIcon> iconList, **int** switches[], List<IBinder> binders,
4. Rect fullscreenStackBounds, Rect dockedStackBounds) {
5. //检查权限
6. enforceStatusBarService();
7. mBar = bar;
8. **synchronized** (mIcons) {
9. **for** (String slot : mIcons.keySet()) {
10. iconSlots.add(slot);
11. iconList.add(mIcons.get(slot));
12. }
13. }
14. **synchronized** (mLock) {
15. switches[0] = gatherDisableActionsLocked(mCurrentUserId, 1);
16. switches[1] = mSystemUiVisibility;
17. switches[2] = mMenuVisible ? 1 : 0;
18. switches[3] = mImeWindowVis;
19. switches[4] = mImeBackDisposition;
20. switches[5] = mShowImeSwitcher ? 1 : 0;
21. switches[6] = gatherDisableActionsLocked(mCurrentUserId, 2);
22. switches[7] = mFullscreenStackSysUiVisibility;
23. switches[8] = mDockedStackSysUiVisibility;
24. binders.add(mImeToken);
25. fullscreenStackBounds.set(mFullscreenStackBounds);
26. dockedStackBounds.set(mDockedStackBounds);
27. }
28. }
29. }

这几者的关系如下



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());
16. }
17. **protected** **void** inflateStatusBarWindow(Context context) {
18. mStatusBarWindow = (StatusBarWindowView) View.inflate(context, R.layout.super\_status\_bar, **null**);
19. }

### 状态栏布局介绍

整个状态栏的父布局是R.layout.super\_status\_bar,对应的是StatusBarWindowView这个自定义布局.

**[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.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>**

我这里以主要的布局层次做介绍,结合图片分析会更加清楚,StatusBarWindowView里有几个主要的布局:

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

如下图



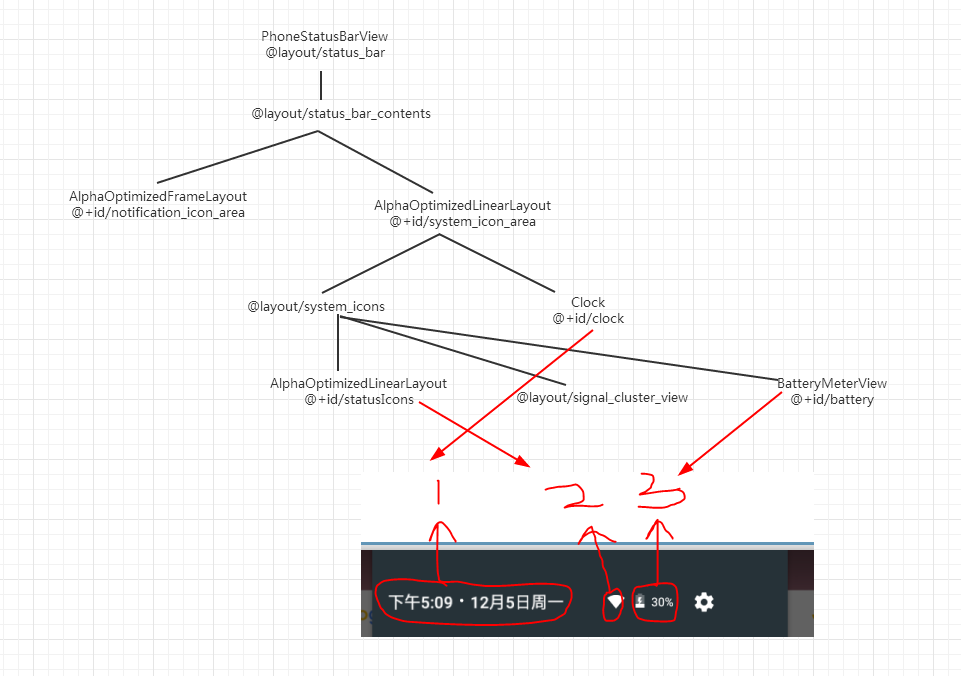
#### layout/status\_bar

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>**

以下是细节图,连线表示层次结构



其中,状态栏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

#### 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

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

### 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>

## 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。

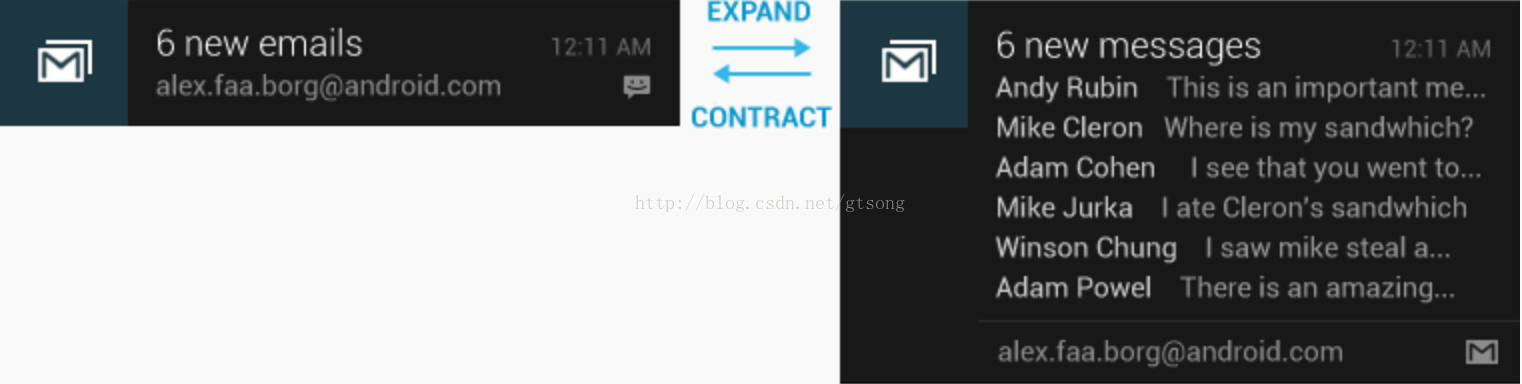


#### Notification的堆栈处理

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

#### 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);

## 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>**

### 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()

### 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)

## 其他

### StatusBar加图标AddIcons

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：



## 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：



## 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/)

# Api

## **禁止4大组件**

为什么要关闭组件？ 在用到组件时，有时候我们可能暂时性的不使用组件，但又不想把组件kill掉，比如创建了一个broadcastReceiver广播监听器，用来想监听第一次开机启动后获得系统的许多相关信息，并保存在文件中，这样以后每次开机启动就不需要再去启动该服务了，也就是说如果没有把receiver关闭掉，就算是不做数据处理，但程序却还一直在后台运行会消耗电量和内存，这时候就需要把这个receiver给关闭掉。

又或者是开机引导程序。

### 禁止组件的API：

void setComponentEnabledSetting (ComponentName componentName, int newState, int flags)

* componentName：组件名称
* newState：组件新的状态，可以设置三个值，分别是如下：

不可用状态：COMPONENT\_ENABLED\_STATE\_DISABLED

可用状态：COMPONENT\_ENABLED\_STATE\_ENABLED

默认状态：COMPONENT\_ENABLED\_STATE\_DEFAULT

* flags:行为标签，值可以是DONT\_KILL\_APP或者0。 0说明杀死包含该组件的app

禁止组件功能后，系统的package信息会记录下来，有关系统的package信息都是保存在/data/system/中package.Xml中，如下：

|  |
| --- |
| <**package name="com.android.provision"   codePath="/system/app/Provision.apk"   nativeLibraryPath="/data/data/com.android.provision/lib"  flags="1" ft="11b7e237e00" it="11b7e237e00"  ut="11b7e237e00" version="15" userId="10005"**>  <**sigs count="1"**>  <**cert index="1"** />  </**sigs**>  <**disabled-components**>  <**item name="com.android.provision.DefaultActivity"** />  </**disabled-components**> </**package**> |

上面绿色部分就是记录了package里面**provision**被禁止的组件信息。禁止之后升级app安装会提示MainActivity} does not exist.如果更新，应用程序列表也搜索不到，只能通过adb uninstall卸载，再重装的方式。

### 获取组件的状态

public int getComponentEnabledSetting(ComponentName componentName)

## 参考

Pk

[**Android 禁用和开启四大组件之setComponentEnabledSetting方法**](http://blog.csdn.net/daiqiquan/article/details/40615115)

# 普通应用知识

[AndroidManifest 中original-package标签](http://blog.csdn.net/ccc20134/article/details/50540800)

<manifest  package="com.android.launcher" >

<original-package android:name="com.android.launcher2" />

这里package="com.android.launcher"，产生的R.[**Java**](http://lib.csdn.net/base/java)就会在com.android.launcher中。  
<original-package android:name="com.android.launcher2" /> 这个地方表示，源码包是com.android.launcher2。

所以在代码中引用的R.java必须是import com.android.launcher.R;应用运行包名是com.android.launcher。

代码的包名是com.android.launcher2。即在项目中需要修改包名时，可以设置 ：<original-package android:name="" />

* 第8章　系统设置（一）
* 第9章　系统设置（二）
* 第10章　系统设置内容提供者（SettingsProvider）
* 第11章　电话与联系人
* 第12章　短信和彩信管理
* 第13章　AndroidHome应用：Launcher2（一）
* 第14章　AndroidHome应用：Launcher2（二）
* 第15章　近场通信（NFC）的实现原理

# REF

[应用程序的安装与卸载笔记](http://blog.csdn.net/sdjzyuxinburen/article/details/50645193)

[如何控制android系统中NavigationBar 的显示与隐藏](http://blog.csdn.net/a2758963/article/details/42675773)

[Android5.1 -Recents分析](http://blog.csdn.net/u013656135/article/details/49686425)

# TASK

* Provisioned标志研究限定哪些程序？我们的为啥是文件方式判断？
* 系统应该发有序广播给我Ordered Broadcast！
* Launcher setComponentEnabledSetting，为啥我的设置不生效？
* 缓存目录结构，seetings app学习
* [Android 7.0 SystemUI(3)--RecentsActivity](http://blog.csdn.net/kebelzc24/article/details/53765379)
* Aidl在服务中的设计

AccessibilityManager：https://www.jianshu.com/p/4cd8c109cdfb

UserManager分析

[Android 多窗口框架全解析](http://blog.csdn.net/xiaosayidao/article/details/75045087)

etActivityInfo(cn, PackageManager.***GET\_META\_DATA***)