# TODO

存储方案

[https://www.smwenku.com/a/5c0a5c67bd9eee6fb37b9a6d/zh-cn](javascript:%20void%200)

Android-Vold, Framework和UI的通信-大结局(12)

<https://blog.csdn.net/gzshun/article/details/7172182>

*ACTION\_MEDIA\_UNMOUNTED must be added for receive ACTION\_MEDIA\_EJECT*

*ACTION\_MEDIA\_EJECT会触发多次么？由于只有变化才发送广播，因此最好是监听ACTION\_MEDIA\_UNMOUNTED！！！？？？*

# 概述

1、进行读取时需要READ\_EXTERNAL\_STORAGE权限；

2、访问应用所属的目录下（如：android/data/[package name]）存储的数据是不需要任

何权限的。

外部存储(external storage)被分割成了多个部分：一个“primary”部分，一个或多个“secondary”部分

使用FUSE (Filesysgem in Userspace ) 对external storage进行管理

SAF框架（[Storage Access Framework](http://developer.android.com/guide/topics/providers/document-provider.html)）

从 Android 6.0 开始，Android 支持移动存储（adoptable storage），例如 SD 卡或者 USB 。移动存储可以像内部存储一样加密和格式化，可以存储所有类型的应用数据。

Android 6.0 引入了新的运行时权限（runtime permissions ）模型，用于应用在运行中必要时申请权限。由于新模型包含了 READ/WRITE\_EXTERNAL\_STORAGE ，因此平台需要在不杀死或者重启运行中的应用的前提下，动态对存储访问授权。

# 场景分析

## 擦除外部存储

在系统设置中发送恢复出厂设置的广播，附带擦除存储的标志位

Intent intent = new Intent(Intent.ACTION\_MASTER\_CLEAR);

intent.addFlags(Intent.FLAG\_RECEIVER\_FOREGROUND);

intent.putExtra(Intent.EXTRA\_REASON, "MasterClearConfirm");

intent.putExtra(Intent.EXTRA\_WIPE\_EXTERNAL\_STORAGE, mEraseSdCard);

调用MasterClearReceiver，

StorageManager.wipeAdoptableDisks

MountService.partitionPublic

MasterClearReceiver处理

**if** (wipeExternalStorage) {  
 *// thr will be started at the end of this task.* **new** WipeAdoptableDisksTask(context, thr).execute();  
} **else** {  
 thr.start();  
}

### MasterClearReceiver.WipeAdoptableDisksTask

@Override  
**protected** Void doInBackground(Void... params) {  
 Slog.*w*(***TAG***, **"Wiping adoptable disks"**);  
 StorageManager sm = (StorageManager) **mContext**.getSystemService(  
 Context.***STORAGE\_SERVICE***);  
 sm.wipeAdoptableDisks();  
 **return null**;  
}

### StorageManager.wipeAdoptableDisks

**mStorageManager** = IStorageManager.Stub.asInterface(ServiceManager.*getServiceOrThrow*(**"mount"**));

**public void** wipeAdoptableDisks() {  
 *// We only wipe devices in "adoptable" locations, which are in a  
 // long-term stable slot/location on the device, where apps have a  
 // reasonable chance of storing sensitive data. (Apps need to go through  
 // SAF to write to transient volumes.)* **final** List<DiskInfo> disks = getDisks();  
 **for** (DiskInfo disk : disks) {  
 **final** String diskId = disk.getId();  
 **if** (disk.isAdoptable()) {  
 Slog.*d*(***TAG***, **"Found adoptable "** + diskId + **"; wiping"**);  
 **try** {  
 *//* ***TODO: switch to explicit wipe command when we have it,*** *// for now rely on the fact that vfat format does a wipe* **mStorageManager**.partitionPublic(diskId);  
 } **catch** (Exception e) {  
 Slog.*w*(***TAG***, **"Failed to wipe "** + diskId + **", but soldiering onward"**, e);  
 }  
 } **else** {  
 Slog.*d*(***TAG***, **"Ignorning non-adoptable disk "** + disk.getId());  
 }  
 }  
}

### StorageMS.partitionPublic

@Override  
**public void** partitionPublic(String diskId) {  
 enforcePermission(android.Manifest.permission.***MOUNT\_FORMAT\_FILESYSTEMS***);  
 waitForReady();  
 **final** CountDownLatch latch = findOrCreateDiskScanLatch(diskId);  
 **mConnector**.execute(**"volume"**, **"partition"**, diskId, **"public"**);  
 waitForLatch(latch, **"partitionPublic"**, 3 \* DateUtils.***MINUTE\_IN\_MILLIS***);  
}

#### findOrCreateDiskScanLatch

**private** CountDownLatch findOrCreateDiskScanLatch(String diskId) {  
 **synchronized** (**mLock**) {  
 CountDownLatch latch = **mDiskScanLatches**.get(diskId);  
 **if** (latch == **null**) {  
 latch = **new** CountDownLatch(1);  
 **mDiskScanLatches**.put(diskId, latch);  
 }  
 **return** latch;  
 }  
}

**mConnector** = **new** NativeDaemonConnector(**this**, **"vold"**, ***MAX\_CONTAINERS*** \* 2, ***VOLD\_TAG***, 25,  
 **null**);  
**mConnector**.setDebug(**true**);  
**mConnector**.setWarnIfHeld(**mLock**);  
**mConnectorThread** = **new** Thread(**mConnector**, ***VOLD\_TAG***);

# StorageManagerService

publishBinderService(**"mount"**, **mStorageManagerService**);

**public long** getStorageLowBytes(File path) {  
 **final long** lowPercent = Settings.Global.getInt(mResolver,  
 Settings.Global.SYS\_STORAGE\_THRESHOLD\_PERCENTAGE, DEFAULT\_THRESHOLD\_PERCENTAGE);  
 **final long** lowBytes = (path.getTotalSpace() \* lowPercent) / 100;  
  
 **final long** maxLowBytes = Settings.Global.getLong(mResolver,  
 Settings.Global.SYS\_STORAGE\_THRESHOLD\_MAX\_BYTES, DEFAULT\_THRESHOLD\_MAX\_BYTES);  
  
 **return** Math.min(lowBytes, maxLowBytes);  
}

**// /小于500M时为低存储**

**private static final int DEFAULT\_THRESHOLD\_PERCENTAGE = 5**;  
**private static final long DEFAULT\_THRESHOLD\_MAX\_BYTES = 500** \* MB\_IN\_BYTES;

C:\Users\10288>adb shell settings get Global SYS\_STORAGE\_THRESHOLD\_PERCENTAGE

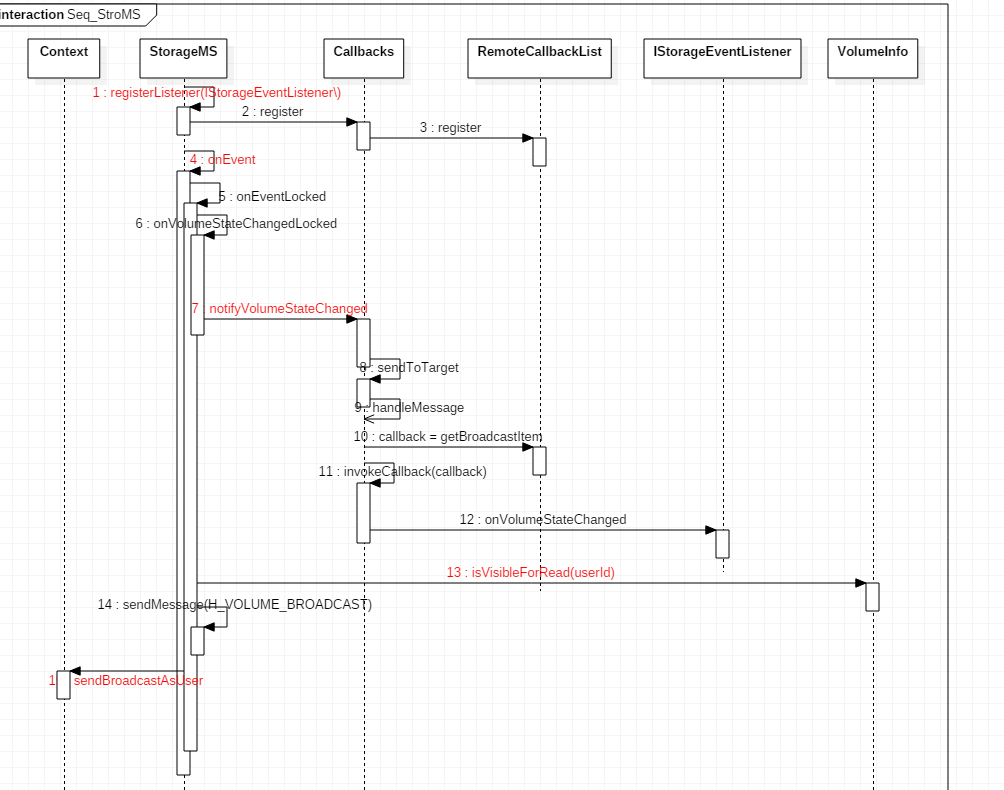
null

C:\Users\10288>adb shell settings get Global SYS\_STORAGE\_THRESHOLD\_MAX\_BYTES

Null

## 监听设置

### 时序图-注册与回调



### StorageMS.registerListener

@Override  
**public void** registerListener(IStorageEventListener listener) {  
 **mCallbacks**.register(listener);  
}

### 发送系统内部组件广播

**if** (**mBootCompleted** && isBroadcastWorthy(vol)) {  
 **final** Intent intent = **new** Intent(VolumeInfo.***ACTION\_VOLUME\_STATE\_CHANGED***);  
 intent.putExtra(VolumeInfo.***EXTRA\_VOLUME\_ID***, vol.**id**);  
 intent.putExtra(VolumeInfo.***EXTRA\_VOLUME\_STATE***, newState);  
 intent.putExtra(VolumeRecord.***EXTRA\_FS\_UUID***, vol.**fsUuid**);  
 intent.addFlags(Intent.***FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT*** | Intent.***FLAG\_RECEIVER\_INCLUDE\_BACKGROUND***);  
 **mHandler**.obtainMessage(***H\_INTERNAL\_BROADCAST***, intent).sendToTarget();  
}

**case *H\_INTERNAL\_BROADCAST***: {  
 *// Internal broadcasts aimed at system components, not for  
 // third-party apps.* **final** Intent intent = (Intent) msg.**obj**;  
 **mContext**.sendBroadcastAsUser(intent, UserHandle.***ALL***,  
 android.Manifest.permission.WRITE\_MEDIA\_STORAGE);

这个需要一些特殊权限，好处就是什么事件都能接受到

android.Manifest.permission.WRITE\_MEDIA\_STORAGE

恩，系统设置就应该监听这个广播啊

过滤条件是开机之后和**感兴趣的广播**

**private boolean** isBroadcastWorthy(VolumeInfo vol) {  
 **switch** (vol.getType()) {  
 **case** VolumeInfo.TYPE\_PRIVATE:  
 **case** VolumeInfo.TYPE\_PUBLIC:  
 **case** VolumeInfo.TYPE\_EMULATED:  
 **break**;  
 **default**:  
 **return false**;  
 }  
  
 **switch** (vol.getState()) {  
 **case** VolumeInfo.STATE\_MOUNTED:  
 **case** VolumeInfo.STATE\_MOUNTED\_READ\_ONLY:  
 **case** VolumeInfo.STATE\_EJECTING:  
 **case** VolumeInfo.STATE\_UNMOUNTED:  
 **case** VolumeInfo.STATE\_UNMOUNTABLE:  
 **case** VolumeInfo.STATE\_BAD\_REMOVAL:  
 **break**;  
 **default**:  
 **return false**;  
 }  
  
 **return true**;  
}

### 发送第三方app广播

设为内部之后（isVisibleForRead为false），就不能监听插拔了，因此还是要监听器来整才好

**for** (**int** userId : **mSystemUnlockedUsers**) {  
 **if** (vol.isVisibleForRead(userId)) {  
 **final** StorageVolume userVol = vol.buildStorageVolume(**mContext**, userId, **false**);  
 **mHandler**.obtainMessage(***H\_VOLUME\_BROADCAST***, userVol).sendToTarget();  
  
 **mCallbacks**.notifyStorageStateChanged(userVol.getPath(), oldStateEnv,  
 newStateEnv);  
 }  
}

#### VolumeInfo

广播分类

*sEnvironmentToBroadcast*.put(Environment.***MEDIA\_UNMOUNTED***, Intent.***ACTION\_MEDIA\_UNMOUNTED***);  
*sEnvironmentToBroadcast*.put(Environment.***MEDIA\_CHECKING***, Intent.***ACTION\_MEDIA\_CHECKING***);  
*sEnvironmentToBroadcast*.put(Environment.***MEDIA\_MOUNTED***, Intent.***ACTION\_MEDIA\_MOUNTED***);  
*sEnvironmentToBroadcast*.put(Environment.***MEDIA\_MOUNTED\_READ\_ONLY***, Intent.***ACTION\_MEDIA\_MOUNTED***);  
*sEnvironmentToBroadcast*.put(Environment.***MEDIA\_EJECTING***, Intent.***ACTION\_MEDIA\_EJECT***);  
*sEnvironmentToBroadcast*.put(Environment.***MEDIA\_UNMOUNTABLE***, Intent.***ACTION\_MEDIA\_UNMOUNTABLE***);  
*sEnvironmentToBroadcast*.put(Environment.***MEDIA\_REMOVED***, Intent.***ACTION\_MEDIA\_REMOVED***);  
*sEnvironmentToBroadcast*.put(Environment.***MEDIA\_BAD\_REMOVAL***, Intent.***ACTION\_MEDIA\_BAD\_REMOVAL***);

对应含义

<**string name="ext\_media\_status\_unmounted" msgid="2551560878416417752"**>"已弹出"</**string**>  
<**string name="ext\_media\_status\_checking" msgid="6193921557423194949"**>"正在检查…"</**string**>  
<**string name="ext\_media\_status\_mounted" msgid="7253821726503179202"**>"就绪"</**string**>  
<**string name="ext\_media\_status\_mounted\_ro" msgid="8020978752406021015"**>"只读"</**string**>  
<**string name="ext\_media\_status\_bad\_removal" msgid="8395398567890329422"**>"未安全移除"</**string**>  
<**string name="ext\_media\_status\_unmountable" msgid="805594039236667894"**>"已损坏"</**string**>  
<**string name="ext\_media\_status\_unsupported" msgid="4691436711745681828"**>"不支持"</**string**>  
<**string name="ext\_media\_status\_ejecting" msgid="5463887263101234174"**>"正在弹出…"</**string**>  
<**string name="ext\_media\_status\_formatting" msgid="1085079556538644861"**>"正在格式化…"</**string**>  
<**string name="ext\_media\_status\_missing" msgid="5638633895221670766"**>"未插入"</**string**>

**public static** @Nullable String getBroadcastForEnvironment(String envState) {  
 **return** *sEnvironmentToBroadcast*.get(envState);  
}

#### MS发送核心代码

**final** String action = VolumeInfo.*getBroadcastForEnvironment*(envState);  
**if** (action != **null**) {  
 **final** Intent intent = **new** Intent(action,  
 Uri.*fromFile*(userVol.getPathFile()));  
 intent.putExtra(StorageVolume.***EXTRA\_STORAGE\_VOLUME***, userVol);  
 intent.addFlags(Intent.***FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT*** | Intent.***FLAG\_RECEIVER\_INCLUDE\_BACKGROUND***);  
 **mContext**.sendBroadcastAsUser(intent, userVol.getOwner());  
}

intent.addFlags(Intent.***FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT*** | Intent.***FLAG\_RECEIVER\_INCLUDE\_BACKGROUND***);

因此接收端规范

IntentFilter filter = new IntentFilter();

filter.addAction(Intent.ACTION\_MEDIA\_MOUNTED);

filter.addAction(Intent.ACTION\_MEDIA\_UNMOUNTED);

filter.addDataScheme("file");

registerReceiver(mReceiver, filter);

自定义的Receiver总是无法接收到SD卡插拔，大部分是因为每天添加这一个filter.addDataScheme("file");

<https://blog.csdn.net/silenceburn/article/details/6083375>

发送的打印日志如下：

MountReceiver: intent = Intent { act=android.intent.action.MEDIA\_MOUNTED dat=file:///storage/3D6B-1CF9 flg=0x4000010 (has extras) }

### 回调和广播区别

广播只是发送vol.isVisibleForRead(userId)的分区

# 加解密

**public void** prepareUserStorage(String volumeUuid, **int** userId, **int** serialNumber, **int** flags) {  
 enforcePermission(android.Manifest.permission.STORAGE\_INTERNAL);  
 waitForReady();  
  
 **try** {  
 **mCryptConnector**.execute(**"cryptfs"**, **"prepare\_user\_storage"**, *escapeNull*(volumeUuid),  
 userId, serialNumber, flags);  
 } **catch** (NativeDaemonConnectorException e) {  
 **throw** e.rethrowAsParcelableException();  
 }  
}

mCryptConnector是连接通过socket连接cryptd　daemon, 然后运行cryptfs中的prepare\_user\_storage方法。中间流程不再叙述，最后跳转到system/vold/Ext4Crypt.cpp的e4crypt\_prepare\_user\_storage

**bool** e4crypt\_prepare\_user\_storage(**const char**\* volume\_uuid, userid\_t user\_id, **int** serial,  
 **int** flags) {  
 LOG(DEBUG) << **"e4crypt\_prepare\_user\_storage for volume "** << escape\_null(volume\_uuid)  
 << **", user "** << user\_id << **", serial "** << serial << **", flags "** << flags;  
  
 **if** (flags & FLAG\_STORAGE\_DE) {  
 *// DE\_sys key* **auto** system\_legacy\_path = android::vold::BuildDataSystemLegacyPath(user\_id);  
 **auto** misc\_legacy\_path = android::vold::BuildDataMiscLegacyPath(user\_id);  
 **auto** profiles\_de\_path = android::vold::BuildDataProfilesDePath(user\_id);  
  
 *// DE\_n key* **auto** system\_de\_path = android::vold::BuildDataSystemDePath(user\_id);  
 **auto** misc\_de\_path = android::vold::BuildDataMiscDePath(user\_id);

*/路径地址，实际上就是/data/user\_de/*  
 **auto** user\_de\_path = android::vold::BuildDataUserDePath(volume\_uuid, user\_id);  
  
 **if** (!prepare\_dir(system\_legacy\_path, 0700, AID\_SYSTEM, AID\_SYSTEM)) **return false**;  
#if MANAGE\_MISC\_DIRS  
 **if** (!prepare\_dir(misc\_legacy\_path, 0750, multiuser\_get\_uid(user\_id, AID\_SYSTEM),  
 multiuser\_get\_uid(user\_id, AID\_EVERYBODY))) **return false**;  
#endif  
 **if** (!prepare\_dir(profiles\_de\_path, 0771, AID\_SYSTEM, AID\_SYSTEM)) **return false**;  
  
 **if** (!prepare\_dir(system\_de\_path, 0770, AID\_SYSTEM, AID\_SYSTEM)) **return false**;  
 **if** (!prepare\_dir(misc\_de\_path, 01771, AID\_SYSTEM, AID\_MISC)) **return false**;  
 **if** (!prepare\_dir(user\_de\_path, 0771, AID\_SYSTEM, AID\_SYSTEM)) **return false**;  
  
 *// For now, FBE is only supported on internal storage* **if** (e4crypt\_is\_native() && volume\_uuid == **nullptr**) {  
 std::string de\_raw\_ref;  
 **if** (!lookup\_key\_ref(s\_de\_key\_raw\_refs, user\_id, &de\_raw\_ref)) **return false**;  
 **if** (!ensure\_policy(de\_raw\_ref, system\_de\_path)) **return false**;  
 **if** (!ensure\_policy(de\_raw\_ref, misc\_de\_path)) **return false**;  
 **if** (!ensure\_policy(de\_raw\_ref, user\_de\_path)) **return false**;  
 }  
 }  
  
 **if** (flags & FLAG\_STORAGE\_CE) {  
 *// CE\_n key* **auto** system\_ce\_path = android::vold::BuildDataSystemCePath(user\_id);  
 **auto** misc\_ce\_path = android::vold::BuildDataMiscCePath(user\_id);  
 **auto** media\_ce\_path = android::vold::BuildDataMediaCePath(volume\_uuid, user\_id);  
 **auto** user\_ce\_path = android::vold::BuildDataUserCePath(volume\_uuid, user\_id);  
  
 **if** (!prepare\_dir(system\_ce\_path, 0770, AID\_SYSTEM, AID\_SYSTEM)) **return false**;  
 **if** (!prepare\_dir(misc\_ce\_path, 01771, AID\_SYSTEM, AID\_MISC)) **return false**;  
 **if** (!prepare\_dir(media\_ce\_path, 0770, AID\_MEDIA\_RW, AID\_MEDIA\_RW)) **return false**;  
 **if** (!prepare\_dir(user\_ce\_path, 0771, AID\_SYSTEM, AID\_SYSTEM)) **return false**;  
  
 *// For now, FBE is only supported on internal storage* **if** (e4crypt\_is\_native() && volume\_uuid == **nullptr**) {  
 std::string ce\_raw\_ref;  
 **if** (!lookup\_key\_ref(s\_ce\_key\_raw\_refs, user\_id, &ce\_raw\_ref)) **return false**;  
 **if** (!ensure\_policy(ce\_raw\_ref, system\_ce\_path)) **return false**;  
 **if** (!ensure\_policy(ce\_raw\_ref, misc\_ce\_path)) **return false**;  
 **if** (!ensure\_policy(ce\_raw\_ref, media\_ce\_path)) **return false**;  
 **if** (!ensure\_policy(ce\_raw\_ref, user\_ce\_path)) **return false**;  
  
 *// Now that credentials have been installed, we can run restorecon  
 // over these paths  
 // NOTE: these paths need to be kept in sync with libselinux* android::vold::RestoreconRecursive(system\_ce\_path);  
 android::vold::RestoreconRecursive(misc\_ce\_path);  
 }  
 }  
  
 **return true**;  
}

prepare\_dir最终会调用到系统C库system/core/libcutils/fs.c中的fs\_prepare\_path\_impl，代码中省略了很多其它路径，因为原理是相同的

e4crypt\_is\_native方法在system/extras/ext4\_utils/ext4\_crypt.cpp

查看手机e4crypt\_is\_native是返回false的，系统属性是block。

FBE是file-based encryption，加密方式的一种，文件加密详细见http://blog.csdn.net/myfriend0/article/details/77094890，文中也解释了CE和DE是什么

    凭据加密 (CE) 存储空间：这是默认存储位置，只有在用户解锁设备后才可用。

    设备加密 (DE) 存储空间：在直接启动模式期间以及用户解锁设备后均可用。

CE和DE流程类似，不贴代码了。

第一段代码的作用就是创建user加密所需相应目录和设置权限。注意最后并没有地方使用serial参数

---------------------

作者：李光宇

来源：CSDN

原文：https://blog.csdn.net/firedancer0089/article/details/78327265

版权声明：本文为博主原创文章，转载请附上博文链接！

# DeviceStorageMonitorService

## 低存储空间逻辑

**private static final int *DEFAULT\_THRESHOLD\_PERCENTAGE*** = 10;  
**private static final long *DEFAULT\_THRESHOLD\_MAX\_BYTES*** = 500 \* MB\_IN\_BYTES;  
**private static final long *DEFAULT\_FULL\_THRESHOLD\_BYTES*** = MB\_IN\_BYTES;

**public long** getStorageLowBytes(File path) {  
 **final long** lowPercent = Settings.Global.*getInt*(**mResolver**,  
 Settings.Global.SYS\_STORAGE\_THRESHOLD\_PERCENTAGE, ***DEFAULT\_THRESHOLD\_PERCENTAGE***);  
 **final long** lowBytes = (path.getTotalSpace() \* lowPercent) / 100;  
  
 **final long** maxLowBytes = Settings.Global.*getLong*(**mResolver**,  
 Settings.Global.SYS\_STORAGE\_THRESHOLD\_MAX\_BYTES, ***DEFAULT\_THRESHOLD\_MAX\_BYTES***);  
  
 **return** Math.*min*(lowBytes, maxLowBytes);  
}

rm500:/ # settings put global sys\_storage\_threshold\_percentage 100

rm500:/ # settings put global sys\_storage\_threshold\_max\_bytes 107374182400

frameworks/base/services/core/java/com/android/server/storage/DeviceStorageMonitorS

**private final** AtomicInteger mSeq = **new** AtomicInteger(1);

设备存储监视器服务是一个模块用来

    1.监视设备存储（“/ data”）。

    2.每60秒扫描一次免费存储空间(谷歌默认值)

    3.当设备的存储空间不足时生成“低存储”通知。 //updateNotifications

    4.引导用户管理设备中安装的所有应用程序，并发送意图。 //updateBroadcasts

    5.存储严重不足时显示警告对话框。

    6.为AMS/PMS提供公共API以查询存储状态

## onStart()

@Override  
**public void** onStart() {  
   
 mCacheFileDeletedObserver = **new** CacheFileDeletedObserver();  
 mCacheFileDeletedObserver.startWatching();  
  
 *// Ensure that the notification channel is set up* PackageManager packageManager = context.getPackageManager();  
 **boolean** isTv = packageManager.hasSystemFeature(PackageManager.FEATURE\_LEANBACK);  
  
 publishBinderService(SERVICE, mRemoteService);  
 publishLocalService(DeviceStorageMonitorInternal.**class**, mLocalService);  
  
 *// Kick off pass to examine storage state* mHandler.removeMessages(MSG\_CHECK);  
 mHandler.obtainMessage(MSG\_CHECK).sendToTarget();  
}

### CacheFileDeletedObserver

**private static class** CacheFileDeletedObserver **extends** FileObserver {  
 **public** CacheFileDeletedObserver() {  
 **super**(Environment.getDownloadCacheDirectory().getAbsolutePath(), FileObserver.DELETE);  
 }  
  
 @Override  
 **public void** onEvent(**int** event, String path) {  
 EventLogTags.writeCacheFileDeleted(path);  
 }  
}

### mHandler

**private final** Handler mHandler = **new** Handler(IoThread.get().getLooper()) {  
 @Override  
 **public void** handleMessage(Message msg) {  
 **switch** (msg.what) {  
 **case** MSG\_CHECK:  
 check();  
 **return**;  
 }  
 }  
};

## check():60S

**private static final long** DEFAULT\_CHECK\_INTERVAL = DateUtils.MINUTE\_IN\_MILLIS;

@WorkerThread  
**private void** check() {  
 **final** StorageManager storage = getContext().getSystemService(StorageManager.**class**);  
 **final int** seq = mSeq.get();  
  
 *// Check every mounted private volume to see if they're low on space* **for** (VolumeInfo vol : storage.getWritablePrivateVolumes()) {  
 **final** File file = vol.getPath();  
 **final long** fullBytes = storage.getStorageFullBytes(file);  
 **final long** lowBytes = storage.getStorageLowBytes(file);  
  
 *// Automatically trim cached data when nearing the low threshold;  
 // when it's within 150% of the threshold, we try trimming usage  
 // back to 200% of the threshold.* **if** (file.getUsableSpace() < (lowBytes \* 3) / 2) {  
 **final** PackageManagerService pms = (PackageManagerService) ServiceManager  
 .getService(**"package"**);  
 **try** {  
 pms.freeStorage(vol.getFsUuid(), lowBytes \* 2, 0);  
 } **catch** (IOException e) {  
 Slog.w(TAG, e);  
 }  
 }  
  
 *// Send relevant broadcasts and show notifications based on any  
 // recently noticed state transitions.* **final** UUID uuid = StorageManager.convert(vol.getFsUuid());  
 **final** State state = findOrCreateState(uuid);  
 **final long** totalBytes = file.getTotalSpace();  
 **final long** usableBytes = file.getUsableSpace();  
  
 **int** oldLevel = state.level;  
 **int** newLevel;  
 **if** (mForceLevel != State.LEVEL\_UNKNOWN) {  
 *// When in testing mode, use unknown old level to force sending  
 // of any relevant broadcasts.* oldLevel = State.LEVEL\_UNKNOWN;  
 newLevel = mForceLevel;  
 } **else if** (usableBytes <= fullBytes) {  
 newLevel = State.LEVEL\_FULL;  
 } **else if** (usableBytes <= lowBytes) {  
 newLevel = State.LEVEL\_LOW;  
 } **else if** (StorageManager.UUID\_DEFAULT.equals(uuid) && !isBootImageOnDisk()  
 && usableBytes < BOOT\_IMAGE\_STORAGE\_REQUIREMENT) {  
 newLevel = State.LEVEL\_LOW;  
 } **else** {  
 newLevel = State.LEVEL\_NORMAL;  
 }  
  
 *// Log whenever we notice drastic storage changes* **if** ((Math.abs(state.lastUsableBytes - usableBytes) > DEFAULT\_LOG\_DELTA\_BYTES)  
 || oldLevel != newLevel) {  
 EventLogTags.writeStorageState(uuid.toString(), oldLevel, newLevel,  
 usableBytes, totalBytes);  
 state.lastUsableBytes = usableBytes;  
 }  
  
 updateNotifications(vol, oldLevel, newLevel);  
 updateBroadcasts(vol, oldLevel, newLevel, seq);  
  
 state.level = newLevel;  
 }  
  
 *// Loop around to check again in future; we don't remove messages since  
 // there might be an immediate request pending.* **if** (!mHandler.hasMessages(MSG\_CHECK)) {  
 mHandler.sendMessageDelayed(mHandler.obtainMessage(MSG\_CHECK),  
 DEFAULT\_CHECK\_INTERVAL);  
 }  
}

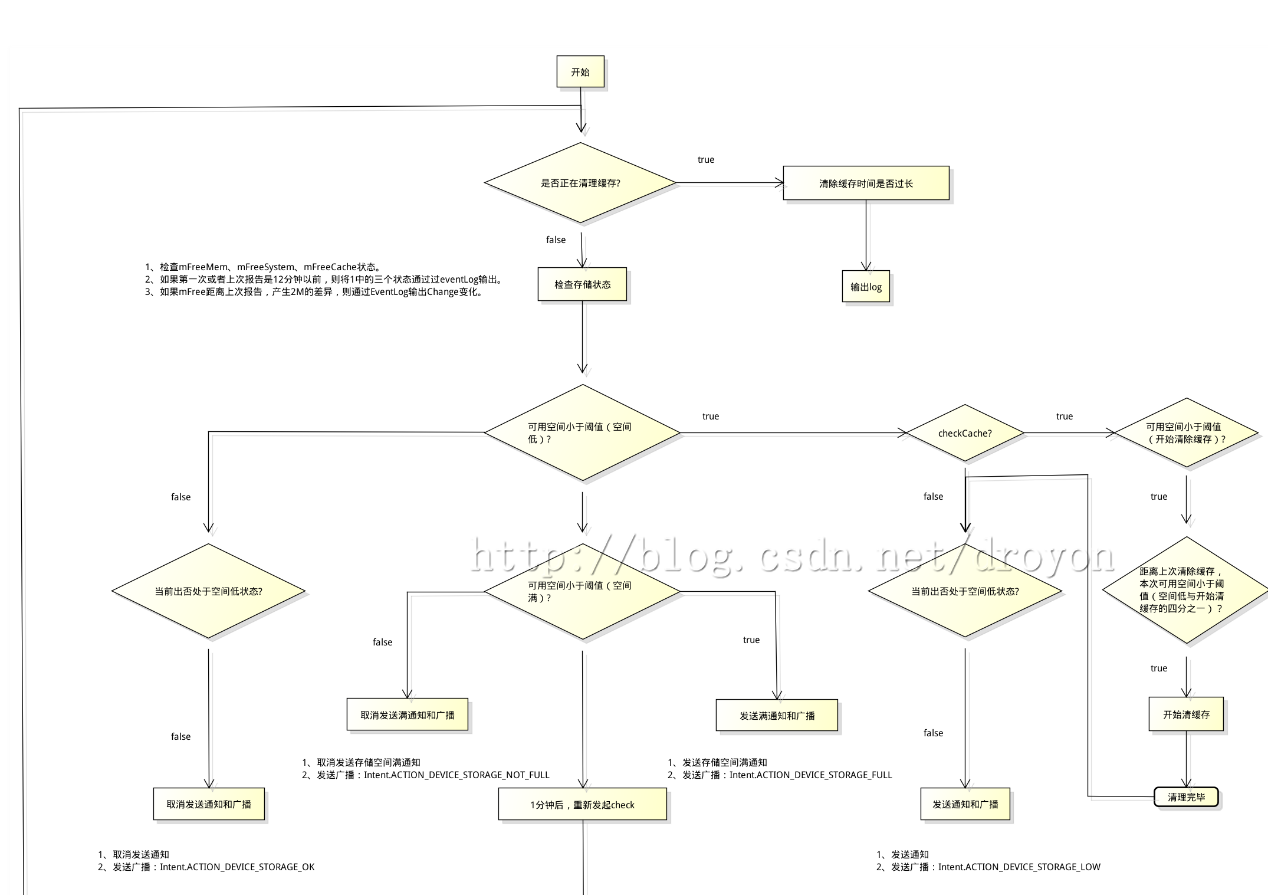
## updateNotifications

**private void** updateNotifications(VolumeInfo vol, **int** oldLevel, **int** newLevel) {  
 **final** Context context = getContext();  
 **final** UUID uuid = StorageManager.convert(vol.getFsUuid());  
  
 **if** (State.isEntering(State.LEVEL\_LOW, oldLevel, newLevel)) {  
 Intent lowMemIntent = **new** Intent(StorageManager.ACTION\_MANAGE\_STORAGE);  
 lowMemIntent.putExtra(StorageManager.EXTRA\_UUID, uuid);  
 lowMemIntent.addFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);  
  
 **final** CharSequence title = context.getText(  
 com.android.internal.R.string.low\_internal\_storage\_view\_title);  
  
 **final** CharSequence details;  
 **if** (StorageManager.UUID\_DEFAULT.equals(uuid)) {  
 details = context.getText(isBootImageOnDisk()  
 ? com.android.internal.R.string.low\_internal\_storage\_view\_text  
 : com.android.internal.R.string.low\_internal\_storage\_view\_text\_no\_boot);  
 } **else** {  
 details = context.getText(  
 com.android.internal.R.string.low\_internal\_storage\_view\_text);  
 }  
  
 PendingIntent intent = PendingIntent.getActivityAsUser(context, 0, lowMemIntent, 0,  
 **null**, UserHandle.CURRENT);  
 Notification notification =  
 **new** Notification.Builder(context, SystemNotificationChannels.ALERTS)  
 .setSmallIcon(com.android.internal.R.drawable.stat\_notify\_disk\_full)  
 .setTicker(title)  
 .setColor(context.getColor(  
 com.android.internal.R.color.system\_notification\_accent\_color))  
 .setContentTitle(title)  
 .setContentText(details)  
 .setContentIntent(intent)  
 .setStyle(**new** Notification.BigTextStyle()  
 .bigText(details))  
 .setVisibility(Notification.VISIBILITY\_PUBLIC)  
 .setCategory(Notification.CATEGORY\_SYSTEM)  
 .extend(**new** Notification.TvExtender()  
 .setChannelId(TV\_NOTIFICATION\_CHANNEL\_ID))  
 .build();  
 notification.flags |= Notification.FLAG\_NO\_CLEAR;  
 mNotifManager.notifyAsUser(uuid.toString(), SystemMessage.NOTE\_LOW\_STORAGE,  
 notification, UserHandle.ALL);  
 } **else if** (State.isLeaving(State.LEVEL\_LOW, oldLevel, newLevel)) {  
 mNotifManager.cancelAsUser(uuid.toString(), SystemMessage.NOTE\_LOW\_STORAGE,  
 UserHandle.ALL);  
 }  
}

## updateBroadcasts

**private void** updateBroadcasts(VolumeInfo vol, **int** oldLevel, **int** newLevel, **int** seq) {  
 **if** (!Objects.equals(StorageManager.UUID\_PRIVATE\_INTERNAL, vol.getFsUuid())) {  
 *// We don't currently send broadcasts for secondary volumes* **return**;  
 }  
  
 **final** Intent lowIntent = **new** Intent(Intent.ACTION\_DEVICE\_STORAGE\_LOW)  
 .addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT  
 | Intent.FLAG\_RECEIVER\_INCLUDE\_BACKGROUND  
 | Intent.FLAG\_RECEIVER\_VISIBLE\_TO\_INSTANT\_APPS)  
 .putExtra(EXTRA\_SEQUENCE, seq);  
 **final** Intent notLowIntent = **new** Intent(Intent.ACTION\_DEVICE\_STORAGE\_OK)  
 .addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT  
 | Intent.FLAG\_RECEIVER\_INCLUDE\_BACKGROUND  
 | Intent.FLAG\_RECEIVER\_VISIBLE\_TO\_INSTANT\_APPS)  
 .putExtra(EXTRA\_SEQUENCE, seq);  
  
 **if** (State.isEntering(State.LEVEL\_LOW, oldLevel, newLevel)) {  
 getContext().sendStickyBroadcastAsUser(lowIntent, UserHandle.ALL);  
 } **else if** (State.isLeaving(State.LEVEL\_LOW, oldLevel, newLevel)) {  
 getContext().removeStickyBroadcastAsUser(lowIntent, UserHandle.ALL);  
 getContext().sendBroadcastAsUser(notLowIntent, UserHandle.ALL);  
 }  
  
 **final** Intent fullIntent = **new** Intent(Intent.ACTION\_DEVICE\_STORAGE\_FULL)  
 .addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT)  
 .putExtra(EXTRA\_SEQUENCE, seq);  
 **final** Intent notFullIntent = **new** Intent(Intent.ACTION\_DEVICE\_STORAGE\_NOT\_FULL)  
 .addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT)  
 .putExtra(EXTRA\_SEQUENCE, seq);  
  
 **if** (State.isEntering(State.LEVEL\_FULL, oldLevel, newLevel)) {  
 getContext().sendStickyBroadcastAsUser(fullIntent, UserHandle.ALL);  
 } **else if** (State.isLeaving(State.LEVEL\_FULL, oldLevel, newLevel)) {  
 getContext().removeStickyBroadcastAsUser(fullIntent, UserHandle.ALL);  
 getContext().sendBroadcastAsUser(notFullIntent, UserHandle.ALL);  
 }  
}

## 流程图



# DUMP

127|rm500:/ # dumpsys devicestoragemonitor

Current DeviceStorageMonitor state:

mFreeMem=8.32 GB mTotalMemory=11.24 GB

mFreeMemAfterLastCacheClear=8.32 GB

mLastReportedFreeMem=8.32 GB mLastReportedFreeMemTime=-8m19s502ms

mLowMemFlag=false mMemFullFlag=false

mIsBootImageOnDisk=true mClearSucceeded=false mClearingCache=false

mMemLowThreshold=500 MB mMemFullThreshold=1.00 MB

mMemCacheStartTrimThreshold=375 MB mMemCacheTrimToThreshold=750 MB