# Todo

<http://gityuan.com/2017/06/03/broadcast_record/>

https://blog.csdn.net/Kitty\_Landon/article/details/79009889

数据结构

1. IntentFilter filter = **new** IntentFilter(Intent.ACTION\_BOOT\_COMPLETED);        filter.setPriority(IntentFilter.SYSTEM\_HIGH\_PRIORITY);

* 系统应该发有序广播给我Ordered Broadcast！

## unregisterReceiver()

2018-10-24 22:31:02.596 6291-6291/? E/ActivityThread: Activity com.socialin.android.photo.picsinphoto.MainPagerActivity has leaked IntentReceiver com.mopub.mobileads.MoPubView$1@1374f49 that was originally registered here. Are you missing a call to unregisterReceiver()?

unregisterReceiver检查原理，泄露原理

## screen\_ON anr

如何禁止广播？？？

## 查询广播

# Android广播机制概述

为什么需要广播

  广播是操作系统framework层对观察者模式(observer)最常用的实现方式

  一个软件系统常常要求在某一个对象的状态发生变化的时候，某些其它的对象做出相应的改变。做到这一点的设计方案有很多，但是为了使系统能够易于复用，应该选择低耦合度的设计方案。减少对象之间的耦合有利于系统的复用，但是同时设计师需要使这些低耦合度的对象之间能够维持行动的协调一致，保证高度的协作。观察者模式是满足这一要求的各种设计方案中最重要的一种。

Android广播分为两个方面：广播发送者和广播接收者，通常情况下，BroadcastReceiver指的就是广播接收者（广播接收器）。广播作为Android组件间的通信方式，可以使用的场景如下：

1.同一app内部的同一组件内的消息通信（单个或多个线程之间）；

2.同一app内部的不同组件之间的消息通信（单个进程）；

3.同一app具有多个进程的不同组件之间的消息通信；

4.不同app之间的组件之间消息通信；

5.Android系统在特定情况下与App之间的消息通信。

线程间，组件间，进程间均可相互通信，核心是AMS。基于消息的发布/订阅事件模型

从实现原理看上，Android中的广播使用了观察者模式，基于消息的发布/订阅事件模型。因此，从实现的角度来看，Android中的广播将广播的发送者和接受者极大程度上解耦，使得系统能够方便集成，更易扩展。具体实现流程要点粗略概括如下：

1.广播接收者BroadcastReceiver通过Binder机制向AMS(Activity Manager Service)进行注册；

2.广播发送者通过binder机制向AMS发送广播；

3.AMS查找符合相应条件（IntentFilter/Permission等）的BroadcastReceiver，将广播发送到BroadcastReceiver（一般情况下是Activity）相应的消息循环队列中；

4.消息循环执行拿到此广播，回调BroadcastReceiver中的onReceive()方法。

对于不同的广播类型，以及不同的BroadcastReceiver注册方式，具体实现上会有不同。但总体流程大致如上。

由此看来，广播发送者和广播接收者分别属于观察者模式中的消息发布和订阅两端，AMS属于中间的处理中心。广播发送者和广播接收者的执行是异步的，发出去的广播不会关心有无接收者接收，也不确定接收者到底是何时才能接收到。显然，整体流程与EventBus非常类似。

在上文说列举的广播机制具体可以使用的场景中，现分析实际应用中的适用性：

第一种情形：同一app内部的同一组件内的消息通信（单个或多个线程之间），实际应用中肯定是不会用到广播机制的（虽然可以用），无论是使用扩展变量作用域、基于接口的回调还是Handler-post/Handler-Message等方式，都可以直接处理此类问题，若适用广播机制，显然有些“杀鸡牛刀”的感觉，会显太“重”；

第二种情形：同一app内部的不同组件之间的消息通信（单个进程），对于此类需求，在有些教复杂的情况下单纯的依靠基于接口的回调等方式不好处理，此时可以直接使用EventBus等，相对而言，EventBus由于是针对统一进程，用于处理此类需求非常适合，且轻松解耦。可以参见文件《Android各组件/控件间通信利器之EventBus》。

第三、四、五情形：由于涉及不同进程间的消息通信，此时根据实际业务使用广播机制会显得非常适宜

## 广播发送及广播类型

根据广播的发送方式，可以将其分为以下几种类型：

1.Normal Broadcast：普通广播

2.System Broadcast: 系统广播

3.Ordered broadcast：有序广播

4.Sticky Broadcast：粘性广播(在 android 5.0/api 21中deprecated,不再推荐使用，相应的还有粘性有序广播，同样已经deprecated)

5.Local Broadcast：App应用内广播

下面分别总结下各种类型的发送方式及其特点。

### Normal Broadcast：普通广播

此处将普通广播界定为：开发者自己定义的intent，以context.sendBroadcast\_"AsUser"(intent, ...)形式。具体可以使用的方法有：

sendBroadcast(intent)/sendBroadcast(intent, receiverPermission)/sendBroadcastAsUser(intent, userHandler)/sendBroadcastAsUser(intent, userHandler,receiverPermission)。

普通广播会被注册了的相应的感兴趣（intent-filter匹配）接收，且顺序是无序的。如果发送广播时有相应的权限要求，BroadCastReceiver如果想要接收此广播，也需要有相应的权限。

### System Broadcast: 系统广播

Android系统中内置了多个系统广播，只要涉及到手机的基本操作，基本上都会发出相应的系统广播。如：开启启动，网络状态改变，拍照，屏幕关闭与开启，点亮不足等等。每个系统广播都具有特定的intent-filter，其中主要包括具体的action，系统广播发出后，将被相应的BroadcastReceiver接收。系统广播在系统内部当特定事件发生时，有系统自动发出。

### Ordered broadcast：有序广播

有序广播的有序广播中的“有序”是针对广播接收者而言的，指的是发送出去的广播被BroadcastReceiver按照先后循序接收。有序广播的定义过程与普通广播无异，只是其的主要发送方式变为：sendOrderedBroadcast(intent, receiverPermission, ...)。

对于有序广播，其主要特点总结如下：

1. 多个具当前已经注册且有效的BroadcastReceiver接收有序广播时，是按照先后顺序接收的，先后顺序判定标准遵循为：将当前系统中所有有效的动态注册和静态注册的BroadcastReceiver按照priority属性值从大到小排序，对于具有相同的priority的动态广播和静态广播，动态广播会排在前面。priority属性大，动态

2>先接收的BroadcastReceiver可以对此有序广播进行截断，使后面的BroadcastReceiver不再接收到此广播，也可以对广播进行修改，使后面的BroadcastReceiver接收到广播后解析得到错误的参数值。当然，一般情况下，不建议对有序广播进行此类操作，尤其是针对系统中的有序广播。

**if** (curr.getPriority() >= curt.priority) {  
 *// Insert this broadcast record into the final list.* receivers.add(it, curr);  
 ir++;  
 curr = **null**;  
 it++;  
 NT++;  
} **else** {

有序广播调用broadcastIntent的区别在于serialized参数，普通广播为false，有序广播为true.

原型为：

public final int broadcastIntent(IApplicationThread caller,

Intent intent, String resolvedType, IIntentReceiver resultTo,

int resultCode, String resultData, Bundle resultExtras,

String[] requiredPermissions, int appOp, Bundle options,

boolean serialized, boolean sticky, int userId) {

AMS

**final int** broadcastIntentLocked(ProcessRecord callerApp,  
 String callerPackage, Intent intent, String resolvedType,  
 IIntentReceiver resultTo, **int** resultCode, String resultData,  
 Bundle resultExtras, String[] requiredPermissions, **int** appOp, Bundle bOptions,  
 **boolean** ordered, **boolean** sticky, **int** callingPid, **int** callingUid, **int** userId) {

**boolean** ordered,

### ~~4)Sticky Broadcast：~~

粘性广播(在 android 5.0/api 21中deprecated,不再推荐使用，相应的还有粘性有序广播，同样已经deprecated)。

既然已经deprecated，此处不再多做总结。

allSticky

### Local Broadcast

App应用内广播（此处的App应用以App应用进程为界）

由前文阐述可知，Android中的广播可以跨进程甚至跨App直接通信，且注册是exported对于有intent-filter的情况下默认值是true，由此将可能出现安全隐患如下：

1.其他App可能会针对性的发出与当前App intent-filter相匹配的广播，由此导致当前App不断接收到广播并处理；

2.其他App可以注册与当前App一致的intent-filter用于接收广播，获取广播具体信息。

无论哪种情形，这些安全隐患都确实是存在的。由此，最常见的增加安全性的方案是：

1.对于同一App内部发送和接收广播，将exported属性人为设置成false，使得非本App内部发出的此广播不被接收；

2.在广播发送和接收时，都增加上相应的permission，用于权限验证；

3.发送广播时，指定特定广播接收器所在的包名，具体是通过intent.setPackage(packageName)指定在，这样此广播将只会发送到此包中的App内与之相匹配的有效广播接收器中。

App应用内广播可以理解成一种局部广播的形式，广播的发送者和接收者都同属于一个App。实际的业务需求中，App应用内广播确实可能需要用到。同时，之所以使用应用内广播时，而不是使用全局广播的形式，更多的考虑到的是Android广播机制中的安全性问题。

相比于全局广播，App应用内广播优势体现在：

1.安全性更高；

2.更加高效。

为此，Android v4兼容包中给出了封装好的LocalBroadcastManager类，用于统一处理App应用内的广播问题，使用方式上与通常的全局广播几乎相同，只是注册/取消注册广播接收器和发送广播时将主调context变成了LocalBroadcastManager的单一实例。

复制代码

1 //registerReceiver(mBroadcastReceiver, intentFilter);

2 //注册应用内广播接收器

3 localBroadcastManager = LocalBroadcastManager.getInstance(this);

4 localBroadcastManager.registerReceiver(mBroadcastReceiver, intentFilter);

5

6 //unregisterReceiver(mBroadcastReceiver);

7 //取消注册应用内广播接收器

8 localBroadcastManager.unregisterReceiver(mBroadcastReceiver);

9

10 Intent intent = new Intent();

11 intent.setAction(BROADCAST\_ACTION);

12 intent.putExtra("name", "qqyumidi");

13 //sendBroadcast(intent);

14 //发送应用内广播

15 localBroadcastManager.sendBroadcast(intent);

## 4.onReceive(context, intent)的context具体类型

1).对于静态注册的ContextReceiver，回调onReceive(context, intent)中的context具体指的是ReceiverRestrictedContext；

2).对于全局广播的动态注册的ContextReceiver，回调onReceive(context, intent)中的context具体指的是Activity Context；

3).对于通过LocalBroadcastManager动态注册的ContextReceiver，回调onReceive(context, intent)中的context具体指的是Application Context。

注：对于LocalBroadcastManager方式发送的应用内广播，只能通过LocalBroadcastManager动态注册的ContextReceiver才有可能接收到（静态注册或其他方式动态注册的ContextReceiver是接收不到的）。

// In this case, we are ready to process the next receiver for the current broadcast,

//聽but are on a queue that would like to wait for services to finish before moving

// on. If there are background services currently starting, then we will go into a

// special state where we hold off on continuing this broadcast until they are done.

解决办法：同时监听开机和sd卡挂载。（也不能只监听挂载就认为开机了，因为有的手机没有sd卡）

<uses-permission android:name="android.permission.RECEIVE\_BOOT\_COMPLETED" ⁄>

<https://blog.csdn.net/q1183345443/article/details/52943874>

查看其它应用怎么添加的

# 数据结构

https://blog.csdn.net/Kitty\_Landon/article/details/79009889

# 广播注册

registerReceiver

<https://blog.csdn.net/kitty_landon/article/details/79009606>

动态广播的注册在代码中需要调用Context类的registerReceiver方法，最终调用到ContextImpl类中的registerReceiver方法中。

## ContextImpl.registerReceiverInternal

@Override

public Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter) {

return registerReceiver(receiver, filter, null, null);

}

@Override

public Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter,

String broadcastPermission, Handler scheduler) {

return registerReceiverInternal(receiver, getUserId(),

filter, broadcastPermission, scheduler, getOuterContext());//broadcastPermission为null，scheduler为null

## AMS. registerReceiver

Intent registerReceiver(IApplicationThread caller, String callerPackage,  
 IIntentReceiver receiver, IntentFilter filter, String permission, **int** userId,  
 **int** flags)

*// The first sticky in the list is returned directly back to the client.*Intent sticky = allSticky != **null** ? allSticky.get(0) : **null**;  
**if** (DEBUG\_BROADCAST) Slog.v(TAG\_BROADCAST, **"Register receiver "** + filter + **": "** + sticky);  
**if** (receiver == **null**) {  
 **return** sticky;  
}

已经deprecated，此处不再多做总结,可以理解为放回都是为空的

https://blog.csdn.net/kitty\_landon/article/details/79009606

# 发送广播

Broadcast的发送是以AMS为中心，通过AMS的分发将消息分发到对应的接收器中。并且这个过程是通过IPC Binder 来完成的

## 原理简介

主要有以下几个步骤：

1.广播的发送者将一个特定类型的广播发送给ActivityMS。

2.AMS接收到这个广播后，首先找到与这个广播对应的广播接收者，然后将它们添加到一个广播调度队列中，再将这个调度队列传递给BroadcastQueue，最后向BroadcastQueue的消息队列发送一个类型为BROADCAST\_INTENT\_MSG的消息，此时对于广播发送者来说，一个广播的发送就完成了。

3.当消息队列中的BROADCAST\_INTENT\_MSG消息被处理时，BroadcastQueue就会从广播调度队列中找对需要接收广播的接收者，并且将对应的广播发送给它们所运行在的应用程序进程。

4.广播接收者所运行在的应用程序进程接收到广播后，并不是直接将接收到的广播分发给各个广播接收者来处理，而是将接收到的广播封装成一个消息，并且发送到主线程的消息队列中。当这个消息被处理时，应用程序进程才会将它所描述的广播发送给相应的广播接收者处理。

惯例，先看时序图，由于发送广播的过程有点复杂，所以时序图分开画的。sendBroadcast、sendOrderedBroadcast和sendStickyBroadcast方法在调用broadcastIntent方法之前的流程是一样的，这里只画出sendBroadcast方法的时序图：



说明：一个广播是使用一个Intent对象来描述的，而这个Intent对象的action名称就是用来描述它所对应的广播类型。

广播的发送也是从ContextWrapper类开始的：

Public void sendBroadcast(Intentintent){

mBase.sendBroadcast(intent);

}

ContextImpl.sendBroadcast

Public void sendBroadcast(Intent intent){

//如果调用者是系统进程的话打印log

warnIfCallingFromSystemProcess(){

if (Process.myUid() == Process.SYSTEM\_UID) {

Slog.w(TAG, "Calling a method in the system process without a qualified user: "

+ Debug.getCallers(5));

}

}

## sendBroadcast

### ContextWrapper.sendBroadcast()

Intent intent = new Intent("...");

sendBroadcast(intent);

这是第一步，发送广播的操作非常简单，生成一个Intent，然后调用sendBroadcast发送就可以了。ContextWrapper的调用其实是一个代理类，实际调用是在ContextImpl。

### ContextImpl.sendBroadcast()

@Override

public void sendBroadcast(Intent intent, String receiverPermission) {

warnIfCallingFromSystemProcess();

String resolvedType = intent.resolveTypeIfNeeded(getContentResolver());

String[] receiverPermissions = receiverPermission == null ? null

: new String[] {receiverPermission};

try {

intent.prepareToLeaveProcess(this); //准备离开应用程序进程，进入AMS进程

//ActivityManagerProxy是一个Binder对象的远程接口，而这个Binder对象就是ActivityManagerService

ActivityManagerNative.getDefault().broadcastIntent(

mMainThread.getApplicationThread(), intent, resolvedType, null,

Activity.RESULT\_OK, null, null, receiverPermissions, AppOpsManager.OP\_NONE,

null, false, false, getUserId());

} catch (RemoteException e) {

throw e.rethrowFromSystemServer();

}

}

在ContextImpl才是发送广播的起点，首先Intent会调用prepareToLeaveProcess()准备离开现有进程，然后调用AMS的broadcastIntent()开始发送过程。

## ActivityManagerNative.broadcastIntent()

这个类就是一个代理，这个函数的职责很简单，将传过来的参数打包到Parcel，然后通过**Binder**将数据传到AMS里。

## ActivityMS.broadcastIntent方法

此函数就是解析一下传进来的数据，然后调用AMS的broadcastIntentLocked进行下一步

**public final int** broadcastIntent(IApplicationThread caller,  
 Intent intent, String resolvedType, IIntentReceiver resultTo,  
 **int** resultCode, String resultData, Bundle resultExtras,  
 String[] requiredPermissions, **int** appOp, Bundle bOptions,  
 **boolean** serialized, **boolean** sticky, **int** userId) {  
 enforceNotIsolatedCaller(**"broadcastIntent"**);

//执行调用者不是独立进程的判断  
 **synchronized**(**this**) {  
 intent = verifyBroadcastLocked(intent);  
 **final** ProcessRecord callerApp = getRecordForAppLocked(caller);  
 **final int** callingPid = Binder.getCallingPid();  
 **final int** callingUid = Binder.getCallingUid();  
 **final long** origId = Binder.clearCallingIdentity();  
 **int** res = **broadcastIntentLocked**(callerApp,  
 callerApp != **null** ? callerApp.info.packageName : **null**,  
 intent, resolvedType, resultTo, resultCode, resultData, resultExtras,  
 requiredPermissions, appOp, bOptions, serialized, sticky,  
 callingPid, callingUid, userId);  
 Binder.restoreCallingIdentity(origId);  
 **return** res;  
 }  
}

## 后续时序图





## AMS.broadcastIntentLocked

这个函数比较长，我们一段一段来看

### 查找接收者

// Figure out who all will receive this broadcast.

List receivers = null;

List<BroadcastFilter> registeredReceivers = null;

if (...) {

registeredReceivers = mReceiverResolver.queryIntent(intent,...);

} else {

registeredReceivers = **mReceiverResolver**.queryIntent(intent,  
 resolvedType, **false** */\*defaultOnly\*/*, userId);

}

这一段是根据Intent的值找出相应接受的广播接收器。因为AMS会把注册的广播接收器保存到mReceiverResolver变量里。

### 替换之前的intent

这一段就是判断是否要替换之前的intent。

**final boolean** replacePending =  
 (intent.getFlags()&Intent.***FLAG\_RECEIVER\_REPLACE\_PENDING***) != 0;

### 添加到队列

**final** BroadcastQueue queue = broadcastQueueForIntent(intent);  
BroadcastRecord r = **new** BroadcastRecord(queue, intent, callerApp,  
 callerPackage, callingPid, callingUid, callerInstantApp, resolvedType,  
 requiredPermissions, appOp, brOptions, registeredReceivers, resultTo,  
 resultCode, resultData, resultExtras, ordered, sticky, **false**, userId);  
**if** (***DEBUG\_BROADCAST***) Slog.*v*(***TAG\_BROADCAST***, **"Enqueueing parallel broadcast "** + r);  
**final boolean** replaced = replacePending  
 && (queue.replaceParallelBroadcastLocked(r) != **null**);  
*// Note: We assume resultTo is null for non-ordered broadcasts.***if** (!replaced) {  
 queue.enqueueParallelBroadcastLocked(r);  
 queue.scheduleBroadcastsLocked();  
}  
registeredReceivers = **null**;  
NR = 0;

这里首先会通过broadcastQueueForIntent()从AMS的全局变量mFgBroadcastQueue或mBgBroadcastQueue中获取一个队列，里面保存着所有Broadcast对象。

然后通过获取的参数新创建一块BroadcastRecord块，将它添加到队列里面去。

并触发一次发送，接下来我们看scheduleBroadcastsLocked()的过程。

## BQueue.scheduleBroadcastsLocked

**public void** scheduleBroadcastsLocked() {  
 **if** (DEBUG\_BROADCAST) Slog.v(TAG\_BROADCAST, **"Schedule broadcasts ["** + mQueueName + **"]: current="** + mBroadcastsScheduled);  
 **if** (mBroadcastsScheduled) {  
 **return**;//如果已经排期  
 }  
 mHandler.sendMessage(mHandler.obtainMessage(BROADCAST\_INTENT\_MSG, **this**));  
 mBroadcastsScheduled = **true**;  
}

这里的mBroadcastsScheduled表示AMS当前是不是正在处理其它广播，如果是的话，这里就先不处理直接返回了，**保证所有广播串行处理**。

因为是通过消息的方式来发送，所以广播的发送和处理是异步的。成员变量mHandler是一个AMS内部的BroadcastQueue定义的Handler类变量，把一个空的类型为BROADCAST\_INTENT\_MSG的消息放到队列里。

上面mBroadcastsScheduled参数是用来标记是否已经向消息队列发送了一个类型为BROADCAST\_INTENT\_MSG消息。BroadcastQueue就是通过这个消息来调度保存在无序广播调度队列和有序广播调度队列中的广播转发任务的。

这里，虽然还没有将广播转发给各目标广播接收者，但是当它执行完成这一步之后，广播发送者就会认为这个广播已经发送成功了。从这里就可以看出，广播的发送和接收是异步的。

### handleMessage

BroadcastHandler **extends** Handler

**public void** handleMessage(Message msg) {  
 **switch** (msg.what) {  
 **case** BROADCAST\_INTENT\_MSG: {  
 **if** (DEBUG\_BROADCAST) Slog.v(  
 TAG\_BROADCAST, **"Received BROADCAST\_INTENT\_MSG"**);  
 processNextBroadcast(**true**);  
 }

Case BROADCAST\_TIMEOUT\_MSG:{

synchronized(mService){

//处理广播超时的操作，报ANR异常

broadcastTimeoutLocked(true);

}

又是通过processNextBroadcast()来处理下一条广播。

### processNextBroadcast

**if** (fromMsg) {  
 mBroadcastsScheduled = **false**;  
}

...

**while** (mParallelBroadcasts.size() > 0) {

r = mParallelBroadcasts.remove(0);

final int N = r.receivers.size();

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

Object target = r.receivers.get(i);

deliverToRegisteredReceiverLocked(r, (BroadcastFilter)target, **false**, i);

}

addBroadcastToHistoryLocked(r);

}

这里就是分发的核心了，首先将mBroadcastsScheduled设为false是让下一条消息能发送，接着循环mParallelBroadcasts的size将里面的广播记录块取出来，因为里面包含了目标target，通过deliverToRegisteredReceiverLocked将它发送给订阅了的接收器。

### deliverToRegisteredReceiverLocked

if (filter.requiredPermission != null) {

...

skip = true;

}

...

if (!skip) {

performReceiveLocked(...);

}

这一段做了很多个判断，都是在检查广播发送和接受的权限判断，如果不通过这直接跳过。在通过判断后再调用**performReceiveLocked**执行发送操作。

//BroadcastQueue将一个广播转发给一个目标广播接收者之前，需要检查这个广播的发送者和接收者的权限。

#### 发送者权限

**private void** deliverToRegisteredReceiverLocked(BroadcastRecord r,  
 BroadcastFilter filter, **boolean** ordered, **int** index) {  
 **boolean** skip = **false**;  
 **if** (filter.requiredPermission != **null**) {  
 **int** perm = mService.checkComponentPermission(filter.requiredPermission,  
 r.callingPid, r.callingUid, -1, **true**);  
 **if** (perm != PackageManager.PERMISSION\_GRANTED) {  
 Slog.w(TAG, **"Permission Denial: broadcasting "** + r.intent.toString()  
 + **" from "** + r.callerPackage + **" (pid="** + r.callingPid + **", uid="** + r.callingUid + **")"** + **" requires "** + filter.requiredPermission  
 + **" due to registered receiver "** + filter);  
 skip = **true**;  
 } **else** {/发  
 **final int** opCode = AppOpsManager.permissionToOpCode(filter.requiredPermission);  
 **if** (opCode != AppOpsManager.OP\_NONE  
 && mService.mAppOpsService.noteOperation(opCode, r.callingUid,  
 r.callerPackage) != AppOpsManager.MODE\_ALLOWED) {  
 Slog.w(TAG, **"Appop Denial: broadcasting "** + r.intent.toString()  
 + **" from "** + r.callerPackage + **" (pid="** + r.callingPid + **", uid="** + r.callingUid + **")"** + **" requires appop "** + AppOpsManager.permissionToOp(  
 filter.requiredPermission)  
 + **" due to registered receiver "** + filter);  
 skip = **true**;  
 }  
 }  
 }  
 **if** (!skip && r.requiredPermissions != **null** && r.requiredPermissions.length > 0) {  
 **for** (**int** i = 0; i < r.requiredPermissions.length; i++) {  
 String requiredPermission = r.requiredPermissions[i];  
 **int** perm = mService.checkComponentPermission(requiredPermission,  
 filter.receiverList.pid, filter.receiverList.uid, -1, **true**);  
 **if** (perm != PackageManager.PERMISSION\_GRANTED) {  
 Slog.w(TAG, **"Permission Denial: receiving "** + r.intent.toString()  
 + **" to "** + filter.receiverList.app  
 + **" (pid="** + filter.receiverList.pid  
 + **", uid="** + filter.receiverList.uid + **")"** + **" requires "** + requiredPermission  
 + **" due to sender "** + r.callerPackage  
 + **" (uid "** + r.callingUid + **")"**);  
 skip = **true**;  
 **break**;  
 }  
 **int** appOp = AppOpsManager.permissionToOpCode(requiredPermission);  
 **if** (appOp != AppOpsManager.OP\_NONE && appOp != r.appOp  
 && mService.mAppOpsService.noteOperation(appOp,  
 filter.receiverList.uid, filter.packageName)  
 != AppOpsManager.MODE\_ALLOWED) {  
 Slog.w(TAG, **"Appop Denial: receiving "** + r.intent.toString()  
 + **" to "** + filter.receiverList.app  
 + **" (pid="** + filter.receiverList.pid  
 + **", uid="** + filter.receiverList.uid + **")"** + **" requires appop "** + AppOpsManager.permissionToOp(  
 requiredPermission)  
 + **" due to sender "** + r.callerPackage  
 + **" (uid "** + r.callingUid + **")"**);  
 skip = **true**;  
 **break**;  
 }  
 }  
 }

#### 接收者权限

**if** (!skip && (r.requiredPermissions == **null** || r.requiredPermissions.length == 0)) {  
 **int** perm = mService.checkComponentPermission(**null**,  
 filter.receiverList.pid, filter.receiverList.uid, -1, **true**);  
 **if** (perm != PackageManager.PERMISSION\_GRANTED) {  
 Slog.w(TAG, **"Permission Denial: security check failed when receiving "** + r.intent.toString()  
 + **" to "** + filter.receiverList.app  
 + **" (pid="** + filter.receiverList.pid  
 + **", uid="** + filter.receiverList.uid + **")"** + **" due to sender "** + r.callerPackage  
 + **" (uid "** + r.callingUid + **")"**);  
 skip = **true**;  
 }  
 }  
 **if** (!skip && r.appOp != AppOpsManager.OP\_NONE  
 && mService.mAppOpsService.noteOperation(r.appOp,  
 filter.receiverList.uid, filter.packageName)  
 != AppOpsManager.MODE\_ALLOWED) {  
 Slog.w(TAG, **"Appop Denial: receiving "** + r.intent.toString()  
 + **" to "** + filter.receiverList.app  
 + **" (pid="** + filter.receiverList.pid  
 + **", uid="** + filter.receiverList.uid + **")"** + **" requires appop "** + AppOpsManager.opToName(r.appOp)  
 + **" due to sender "** + r.callerPackage  
 + **" (uid "** + r.callingUid + **")"**);  
 skip = **true**;  
 }  
 **if** (!skip) {  
 **final int** allowed = mService.checkAllowBackgroundLocked(filter.receiverList.uid,  
 filter.packageName, -1, **true**);  
 **if** (allowed == ActivityManager.APP\_START\_MODE\_DISABLED) {  
 Slog.w(TAG, **"Background execution not allowed: receiving "** + r.intent  
 + **" to "** + filter.receiverList.app  
 + **" (pid="** + filter.receiverList.pid  
 + **", uid="** + filter.receiverList.uid + **")"**);  
 skip = **true**;  
 }  
 }  
  
 **if** (!mService.mIntentFirewall.checkBroadcast(r.intent, r.callingUid,  
 r.callingPid, r.resolvedType, filter.receiverList.uid)) {  
 skip = **true**;  
 }  
  
 **if** (!skip && (filter.receiverList.app == **null** || filter.receiverList.app.crashing)) {  
 Slog.w(TAG, **"Skipping deliver ["** + mQueueName + **"] "** + r  
 + **" to "** + filter.receiverList + **": process crashing"**);  
 skip = **true**;  
 }  
  
 **if** (skip) {  
 r.delivery[index] = BroadcastRecord.DELIVERY\_SKIPPED;  
 **return**;  
 }  
  
 *// If permissions need a review before any of the app components can run, we drop  
 // the broadcast and if the calling app is in the foreground and the broadcast is  
 // explicit we launch the review UI passing it a pending intent to send the skipped  
 // broadcast.* **if** (Build.PERMISSIONS\_REVIEW\_REQUIRED) {  
 **if** (!requestStartTargetPermissionsReviewIfNeededLocked(r, filter.packageName,  
 filter.owningUserId)) {  
 r.delivery[index] = BroadcastRecord.DELIVERY\_SKIPPED;  
 **return**;  
 }  
 }  
  
 r.delivery[index] = BroadcastRecord.DELIVERY\_DELIVERED;  
  
 *// If this is not being sent as an ordered broadcast, then we  
 // don't want to touch the fields that keep track of the current  
 // state of ordered broadcasts.* **if** (ordered) {  
 r.receiver = filter.receiverList.receiver.asBinder();  
 r.curFilter = filter;  
 filter.receiverList.curBroadcast = r;  
 r.state = BroadcastRecord.CALL\_IN\_RECEIVE;  
 **if** (filter.receiverList.app != **null**) {  
 *// Bump hosting application to no longer be in background  
 // scheduling class. Note that we can't do that if there  
 // isn't an app... but we can only be in that case for  
 // things that directly call the IActivityManager API, which  
 // are already core system stuff so don't matter for this.* r.curApp = filter.receiverList.app;  
 filter.receiverList.app.curReceiver = r;  
 mService.updateOomAdjLocked(r.curApp);  
 }  
 }  
 **try** {  
 **if** (DEBUG\_BROADCAST\_LIGHT) Slog.i(TAG\_BROADCAST,  
 **"Delivering to "** + filter + **" : "** + r);  
 **if** (filter.receiverList.app != **null** && filter.receiverList.app.inFullBackup) {  
 *// Skip delivery if full backup in progress  
 // If it's an ordered broadcast, we need to continue to the next receiver.* **if** (ordered) {  
 skipReceiverLocked(r);  
 }  
 } **else** {

#### performReceiveLocked

performReceiveLocked(filter.receiverList.app, filter.receiverList.receiver,  
 **new** Intent(r.intent), r.resultCode, r.resultData,  
 r.resultExtras, r.ordered, r.initialSticky, r.userId);  
 }  
 **if** (ordered) {  
 r.state = BroadcastRecord.CALL\_DONE\_RECEIVE;  
 }  
 } **catch** (RemoteException e) {  
 Slog.w(TAG, **"Failure sending broadcast "** + r.intent, e);  
 **if** (ordered) {  
 r.receiver = **null**;  
 r.curFilter = **null**;  
 filter.receiverList.curBroadcast = **null**;  
 **if** (filter.receiverList.app != **null**) {  
 filter.receiverList.app.curReceiver = **null**;  
 }  
 }  
 }  
}

### performReceiveLocked

首先来看下这个函数的参数：

**void** performReceiveLocked(ProcessRecord app, IIntentReceiver receiver,  
 Intent intent, **int** resultCode, String data, Bundle extras,  
 **boolean** ordered, **boolean** sticky, **int** sendingUser) **throws** RemoteException

这里首先判断通过注册进来的广播是什么，因为例子中是通过activity注册的，这里的app参数就是代表activity的进程记录块，receiver这是注册时传给AMS的Binder对象。

在调用后会通过：app.thread.scheduleRegisteredReceiver()函数把广播分发给activity。

进程间通讯，binder

**if** (app != **null**) {  
 **if** (app.thread != **null**) {  
 *// If we have an app thread, do the call through that so it is  
 // correctly ordered with other one-way calls.* **try** {  
 **app.thread.scheduleRegisteredReceiver**(receiver, intent, resultCode,  
 data, extras, ordered, sticky, sendingUser, app.repProcState);  
 *// TODO: Uncomment this when (b/28322359) is fixed and we aren't getting  
 // DeadObjectException when the process isn't actually dead.  
 //} catch (DeadObjectException ex) {  
 // Failed to call into the process. It's dying so just let it die and move on.  
 // throw ex;* } **catch** (RemoteException ex) {  
 }  
 }   
 } **else** {  
 receiver.performReceive(intent, resultCode, data, extras, ordered,  
 sticky, sendingUser);  
 }  
}

上面performReceiveLocked方法中的参数app指向一个ProcessRecord对象，用来描述目标广播接收者所运行在的应用程序进程;参数receiver指向了一个实现了IIntentReceiver接口的Binder代理对象，用来描述目标广播接收者;参数intent用来描述即将要发送给目标广播接收者的一个广播。

## scheduleRegisteredReceiver

### ApplicationThreadNative.scheduleRegisteredReceiver()

**public void** scheduleRegisteredReceiver(IIntentReceiver receiver, Intent intent,  
 **int** resultCode, String dataStr, Bundle extras, **boolean** ordered,  
 **boolean** sticky, **int** sendingUser, **int** processState) **throws** RemoteException {  
 Parcel data = Parcel.obtain();  
 data.writeInterfaceToken(IApplicationThread.descriptor);  
 data.writeStrongBinder(receiver.asBinder());  
 intent.writeToParcel(data, 0);  
 data.writeInt(resultCode);  
 data.writeString(dataStr);  
 data.writeBundle(extras);  
 data.writeInt(ordered ? 1 : 0);  
 data.writeInt(sticky ? 1 : 0);  
 data.writeInt(sendingUser);  
 data.writeInt(processState);  
 mRemote.transact(SCHEDULE\_REGISTERED\_RECEIVER\_TRANSACTION, data, **null**,  
 IBinder.FLAG\_ONEWAY);  
 data.recycle();  
}

这里也很简单，做一个打包通过binder分发而已

### ApplicationThread.scheduleRegisteredReceiver()

传进来的第一个参数是**IIntentReceiver receiver**，其实际类型是定义在LoadedApk类的内部类ReceiverDispatcher里面的一个内部类InnerReceiver，调用performReceive函数来执行。

参数receiver指向了一个IIntentReceiver对象，前面讲过每一个IIntentReceiver对象在内部都封装了一个广播接收者，并且代替它所封装的广播接收者注册到AMS中。这样当AMS将一个广播发送给一个目标广播接收者时，实际上是将这个广播发送给了与目标广播接收者相关联的一个IIntentReceiver对象，而这个IIntentReceiver对象是通过它的performReceive方法来接收这个广播的。

### LoadedApk.ReceiverDispatcher.InnerReceiver.performReceive()

LoadedApk.ReceiverDispatcher rd = mDispatcher.get();

rd.performReceive();

这里又是一个转折，调用LoadedApk.ReceiverDispatcher类的performReceive()来执行。

### LoadedApk.ReceiverDispatcher.performReceive()

final Handler mActivityThread;

**public void** performReceive(Intent intent, **int** resultCode, String data,  
 Bundle extras, **boolean** ordered, **boolean** sticky, **int** sendingUser) {  
 **final** Args args = **new** Args(intent, resultCode, data, extras, ordered,  
 sticky, sendingUser);  
 **if** (intent == **null**) {  
 Log.wtf(TAG, **"Null intent received"**);  
 } **else** {  
 **if** (ActivityThread.DEBUG\_BROADCAST) {  
 **int** seq = intent.getIntExtra(**"seq"**, -1);  
 Slog.i(ActivityThread.TAG, **"Enqueueing broadcast "** + intent.getAction()  
 + **" seq="** + seq + **" to "** + mReceiver);  
 }  
 }  
 **if** (intent == **null** || !mActivityThread.post(args)) {  
 **if** (mRegistered && ordered) {  
 IActivityManager mgr = ActivityManagerNative.getDefault();  
 **if** (ActivityThread.DEBUG\_BROADCAST) Slog.i(ActivityThread.TAG,  
 **"Finishing sync broadcast to "** + mReceiver);  
 args.sendFinished(mgr);  
 }  
 }  
 }  
  
}

在ReceiverDispatcher类里mActivityThread的类型是一个handler，它是前面MainActivity注册广播接收器时，从ActivityThread取得的。这里ReceiverDispatcher借助这个Handler，把这个广播以消息的形式放到MainActivity所在的这个ActivityThread的消息队列中去。  
 而Args是ReceiverDispatcher的一个内部类，继承自Runnable类。

**final class** Args **extends** BroadcastReceiver.PendingResult **implements** Runnable

### Hanlder.post

post的作用就是把消息放在消息队列中，然后就返回，这个消息最终会在传进来的Runnable类型的参数的run成员函数中进行处理。

### LoadedApk.ReceiverDispatcher.Args.run

final BroadcastReceiver receiver = mReceiver;

final Intent intent = mCurIntent;

...

ClassLoader cl = mReceiver.getClass().getClassLoader();

intent.setExtrasClassLoader(cl);

setExtrasClassLoader(cl);

receiver.setPendingResult(this);

receiver.onReceive(mContext, intent);

mReceiver是ReceiverDispatcher类的成员变量,它就是MainActivity注册广播接收器时创建的BroadcastReceiver实例。  
在这里就能够通过receiver将广播分发到处理的地方了。

### BroadcastReceiver.onReceive

最后就是定义在自己类里的onReceive被调用，整个过程到此结束。

## 小结

最后我们将整个过程来总结一下

1. 第一阶段：  
   Step 1 ～ Step 7:通过自己的sendBroadcast把一个广播通过Binder进程间通信机制发送给AMS，AMS根据这个广播的Action类型找到相应的广播接收器，然后把这个广播放进自己的消息队列中去。
2. 第二阶段：  
   Step 8 ～ Step 15:AMS在消息循环中处理这个广播，并通过Binder进程间通信机制把这个广播分发给注册的广播接收分发器ReceiverDispatcher，ReceiverDispatcher把这个广播放进MainActivity所在的线程的消息队列中去。
3. 第三阶段：  
   Step 16 ～ Step 17:  
   ReceiverDispatcher的内部类Args在MainActivity所在的线程消息循环中处理这个广播，最终是将这个广播分发给所注册的BroadcastReceiver实例的onReceive函数进行处理。

## 参考

Android 浅析 Broadcast(三) 发送原理

https://www.jianshu.com/p/abb173858faf

## toDOXXX

finalSparseArray>>mStickyBroadcasts=

newSparseArray>>();

privatefinalintbroadcastIntentLocked(ProcessRecordcallerApp,

StringcallerPackage,Intentintent,StringresolvedType,

IIntentReceiverresultTo,intresultCode,StringresultData,

BundleresultExtras,String[]requiredPermissions,intappOp,Bundleoptions,

booleanordered,booleansticky,intcallingPid,intcallingUid,intuserId){

intent=newIntent(intent);

//Bydefaultbroadcastsdonotgotostoppedapps.

//设置这个flag后，intent将不会去匹配这个package中当前停止运行的组件。

intent.addFlags(Intent.FLAG\_EXCLUDE\_STOPPED\_PACKAGES);

//Ifwehavenotfinishedbooting,don'tallowthistolaunchnewprocesses.

//FLAG\_RECEIVER\_BOOT\_UPGRADE标志是广播用于系统升级的，如果设置了该标记，允许系统在启动完成前发送广播

if(!mProcessesReady&&(intent.getFlags()&Intent.FLAG\_RECEIVER\_BOOT\_UPGRADE)==0){

//追加FLAG\_RECEIVER\_REGISTERED\_ONLY标志后，只有动态注册的广播接收者能收到广播

//这是因为在系统启动过程中，PackageManagerService可能还未启动，此时AMS是无法获得静态注册的广播接收者的

intent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY);

}

if(DEBUG\_BROADCAST\_LIGHT)Slog.v(TAG\_BROADCAST,

(sticky?"Broadcaststicky:":"Broadcast:")+intent

+"ordered="+ordered+"userid="+userId);

if((resultTo!=null)&&!ordered){

Slog.w(TAG,"Broadcast"+intent+"notorderedbutresultcallbackrequested!");

}

//处理调用者uid

userId=handleIncomingUser(callingPid,callingUid,userId,

true,ALLOW\_NON\_FULL,"broadcast",callerPackage);

//Makesurethattheuserwhoisreceivingthisbroadcastisrunning.

//Ifnot,wewilljustskipit.Makeanexceptionforshutdownbroadcasts

//andupgradesteps.

if(userId!=UserHandle.USER\_ALL&&!isUserRunningLocked(userId,false)){

if((callingUid!=Process.SYSTEM\_UID

||(intent.getFlags()&Intent.FLAG\_RECEIVER\_BOOT\_UPGRADE)==0)

&&!Intent.ACTION\_SHUTDOWN.equals(intent.getAction())){

Slog.w(TAG,"Skippingbroadcastof"+intent

+":user"+userId+"isstopped");

returnActivityManager.BROADCAST\_FAILED\_USER\_STOPPED;

}

}

BroadcastOptionsbrOptions=null;

if(options!=null){

brOptions=newBroadcastOptions(options);

if(brOptions.getTemporaryAppWhitelistDuration()>0){

//Seeifthecallerisallowedtodothis.Notewearecheckingagainst

//theactualrealcaller(notwhoeverprovidedtheoperationassaya

//PendingIntent),becausethatwhoisactuallysuppliedthearguments.

if(checkComponentPermission(

android.Manifest.permission.CHANGE\_DEVICE\_IDLE\_TEMP\_WHITELIST,

Binder.getCallingPid(),Binder.getCallingUid(),-1,true)

!=PackageManager.PERMISSION\_GRANTED){

Stringmsg="PermissionDenial:"+intent.getAction()

+"broadcastfrom"+callerPackage+"(pid="+callingPid

+",uid="+callingUid+")"

+"requires"

+android.Manifest.permission.CHANGE\_DEVICE\_IDLE\_TEMP\_WHITELIST;

Slog.w(TAG,msg);

thrownewSecurityException(msg);

}

}

}

/\*

\*Preventnon-systemcode(definedheretobenon-persistent

\*processes)fromsendingprotectedbroadcasts.

\*防止非系统代码(这里定义为非持久性进程)发送受保护的广播

\*/

intcallingAppId=UserHandle.getAppId(callingUid);

if(callingAppId==Process.SYSTEM\_UID||callingAppId==Process.PHONE\_UID

||callingAppId==Process.SHELL\_UID||callingAppId==Process.BLUETOOTH\_UID

||callingAppId==Process.NFC\_UID||callingUid==0){

//Alwaysokay.

}elseif(callerApp==null||!callerApp.persistent){//调用者为null或者调用者不是持久性进程

try{

//非系统应用不能发送受保护的广播

if(AppGlobals.getPackageManager().isProtectedBroadcast(

intent.getAction())){

Stringmsg="PermissionDenial:notallowedtosendbroadcast"

+intent.getAction()+"frompid="

+callingPid+",uid="+callingUid;

Slog.w(TAG,msg);

thrownewSecurityException(msg);

}elseif(AppWidgetManager.ACTION\_APPWIDGET\_CONFIGURE.equals(intent.getAction())){

//Specialcaseforcompatibility(兼容性):wedon'twantappstosendthis,

//buthistoricallyithasnotbeenprotectedandappsmaybeusingit

//topoke(干涉)theirownappwidget.So,insteadofmakingitprotected,

//justlimitittothecaller.

if(callerApp==null){

Stringmsg="PermissionDenial:notallowedtosendbroadcast"

+intent.getAction()+"fromunknowncaller.";

Slog.w(TAG,msg);

thrownewSecurityException(msg);

//接收目标组件不为null

}elseif(intent.getComponent()!=null){

//Theyaregoodenoughtosendtoanexplicitcomponent...verify

//itisbeingsenttothecallingapp.

if(!intent.getComponent().getPackageName().equals(

callerApp.info.packageName)){

Stringmsg="PermissionDenial:notallowedtosendbroadcast"

+intent.getAction()+"to"

+intent.getComponent().getPackageName()+"from"

+callerApp.info.packageName;

Slog.w(TAG,msg);

thrownewSecurityException(msg);

}

}else{

//发送者不为null，接收者组件为null时，设置接收者只能是发送者

//Limitbroadcasttotheirownpackage.

intent.setPackage(callerApp.info.packageName);

}

}

}catch(RemoteExceptione){

Slog.w(TAG,"Remoteexception",e);

returnActivityManager.BROADCAST\_SUCCESS;

}

}

finalStringaction=intent.getAction();

if(action!=null){

//特殊的Action有不同的处理方式

switch(action){

caseIntent.ACTION\_UID\_REMOVED:

caseIntent.ACTION\_PACKAGE\_REMOVED:

caseIntent.ACTION\_PACKAGE\_CHANGED:

caseIntent.ACTION\_EXTERNAL\_APPLICATIONS\_UNAVAILABLE:

caseIntent.ACTION\_EXTERNAL\_APPLICATIONS\_AVAILABLE:

//Handlespecialintents:ifthisbroadcastisfromthepackage

//manageraboutapackagebeingremoved,weneedtoremoveallof

//itsactivitiesfromthehistorystack.

if(checkComponentPermission(

android.Manifest.permission.BROADCAST\_PACKAGE\_REMOVED,

callingPid,callingUid,-1,true)

!=PackageManager.PERMISSION\_GRANTED){

Stringmsg="PermissionDenial:"+intent.getAction()

+"broadcastfrom"+callerPackage+"(pid="+callingPid

+",uid="+callingUid+")"

+"requires"

+android.Manifest.permission.BROADCAST\_PACKAGE\_REMOVED;

Slog.w(TAG,msg);

thrownewSecurityException(msg);

}

switch(action){

caseIntent.ACTION\_UID\_REMOVED:

finalBundleintentExtras=intent.getExtras();

finalintuid=intentExtras!=null

?intentExtras.getInt(Intent.EXTRA\_UID):-1;

if(uid>=0){

mBatteryStatsService.removeUid(uid);

mAppOpsService.uidRemoved(uid);

}

break;

//app正在移动到SD卡中，发出的广播

caseIntent.ACTION\_EXTERNAL\_APPLICATIONS\_UNAVAILABLE:

//Ifresourcesareunavailablejustforcestopallthosepackages

//andflushtheattributecacheaswell.

Stringlist[]=

intent.getStringArrayExtra(Intent.EXTRA\_CHANGED\_PACKAGE\_LIST);

if(list!=null&&list.length>0){

for(inti=0;i forceStopPackageLocked(list[i],-1,false,true,true,

false,false,userId,"storageunmount");

}

//清空app的任务栈

mRecentTasks.cleanupLocked(UserHandle.USER\_ALL);

//发送app不可用的广播

sendPackageBroadcastLocked(

IApplicationThread.EXTERNAL\_STORAGE\_UNAVAILABLE,list,

userId);

}

break;

//app完成移动到SD的操作，发出的广播

caseIntent.ACTION\_EXTERNAL\_APPLICATIONS\_AVAILABLE:

mRecentTasks.cleanupLocked(UserHandle.USER\_ALL);

break;

caseIntent.ACTION\_PACKAGE\_REMOVED:

caseIntent.ACTION\_PACKAGE\_CHANGED:

Uridata=intent.getData();

Stringssp;

if(data!=null&&(ssp=data.getSchemeSpecificPart())!=null){

booleanremoved=Intent.ACTION\_PACKAGE\_REMOVED.equals(action);

booleanfullUninstall=removed&&

!intent.getBooleanExtra(Intent.EXTRA\_REPLACING,false);

finalbooleankillProcess=

!intent.getBooleanExtra(Intent.EXTRA\_DONT\_KILL\_APP,false);

if(killProcess){

forceStopPackageLocked(ssp,UserHandle.getAppId(

intent.getIntExtra(Intent.EXTRA\_UID,-1)),

false,true,true,false,fullUninstall,userId,

removed?"pkgremoved":"pkgchanged");

}

if(removed){

sendPackageBroadcastLocked(IApplicationThread.PACKAGE\_REMOVED,

newString[]{ssp},userId);

if(fullUninstall){

mAppOpsService.packageRemoved(

intent.getIntExtra(Intent.EXTRA\_UID,-1),ssp);

//Removeallpermissionsgrantedfrom/tothispackage

removeUriPermissionsForPackageLocked(ssp,userId,true);

removeTasksByPackageNameLocked(ssp,userId);

mBatteryStatsService.notePackageUninstalled(ssp);

}

}else{

cleanupDisabledPackageComponentsLocked(ssp,userId,killProcess,

intent.getStringArrayExtra(

Intent.EXTRA\_CHANGED\_COMPONENT\_NAME\_LIST));

}

}

break;

}

break;

caseIntent.ACTION\_PACKAGE\_ADDED:

//Specialcaseforaddingapackage:bydefaultturnoncompatibilitymode.

Uridata=intent.getData();

Stringssp;

if(data!=null&&(ssp=data.getSchemeSpecificPart())!=null){

finalbooleanreplacing=

intent.getBooleanExtra(Intent.EXTRA\_REPLACING,false);

mCompatModePackages.handlePackageAddedLocked(ssp,replacing);

try{

ApplicationInfoai=AppGlobals.getPackageManager().

getApplicationInfo(ssp,0,0);

mBatteryStatsService.notePackageInstalled(ssp,

ai!=null?ai.versionCode:0);

}catch(RemoteExceptione){

}

}

break;

caseIntent.ACTION\_TIMEZONE\_CHANGED:

//Ifthisisthetimezonechangedaction,queueupamessagethatwillreset

//thetimezoneofallcurrentlyrunningprocesses.Thismessagewillget

//queuedupbeforethebroadcasthappens.

mHandler.sendEmptyMessage(UPDATE\_TIME\_ZONE);

break;

caseIntent.ACTION\_TIME\_CHANGED:

//Iftheusersetthetime,letallrunningprocessesknow.

finalintis24Hour=

intent.getBooleanExtra(Intent.EXTRA\_TIME\_PREF\_24\_HOUR\_FORMAT,false)?1

:0;

mHandler.sendMessage(mHandler.obtainMessage(UPDATE\_TIME,is24Hour,0));

BatteryStatsImplstats=mBatteryStatsService.getActiveStatistics();

synchronized(stats){

stats.noteCurrentTimeChangedLocked();

}

break;

caseIntent.ACTION\_CLEAR\_DNS\_CACHE:

mHandler.sendEmptyMessage(CLEAR\_DNS\_CACHE\_MSG);

break;

caseProxy.PROXY\_CHANGE\_ACTION:

ProxyInfoproxy=intent.getParcelableExtra(Proxy.EXTRA\_PROXY\_INFO);

mHandler.sendMessage(mHandler.obtainMessage(UPDATE\_HTTP\_PROXY\_MSG,proxy));

break;

}

}

//SmartContainermodifiedbegin

if(!SmartContainerConfig.WITH\_OUT\_VIRTUAL\_BOX){

if(mAMSFunc.processSpecialIntent(intent,callingUid,userId)!=0)

returnActivityManager.BROADCAST\_SUCCESS;

}

//SmartContainermodifiedend

//Addtothestickylistifrequested.

if(sticky){

//粘性广播要加BROADCAST\_STICKY权限

if(checkPermission(android.Manifest.permission.BROADCAST\_STICKY,

callingPid,callingUid)

!=PackageManager.PERMISSION\_GRANTED){

Stringmsg="PermissionDenial:broadcastIntent()requestingastickybroadcastfrompid="

+callingPid+",uid="+callingUid

+"requires"+android.Manifest.permission.BROADCAST\_STICKY;

Slog.w(TAG,msg);

thrownewSecurityException(msg);

}

//发送粘性广播不能有其他权限

if(requiredPermissions!=null&&requiredPermissions.length>0){

Slog.w(TAG,"Can'tbroadcaststickyintent"+intent

+"andenforcepermissions"+Arrays.toString(requiredPermissions));

returnActivityManager.BROADCAST\_STICKY\_CANT\_HAVE\_PERMISSION;

}

//粘性广播不能发给指定的接收组件

if(intent.getComponent()!=null){

thrownewSecurityException(

"Stickybroadcastscan'ttargetaspecificcomponent");

}

//WeuseuserIddirectlyhere,sincethe"all"targetismaintained

//asaseparatesetofstickybroadcasts.

if(userId!=UserHandle.USER\_ALL){

//Butfirst,ifthisisnotabroadcasttoallusers,then

//makesureitdoesn'tconflictwithanexistingbroadcastto

//allusers.

//如果广播不是发给所有用户的，则确认对所有用户它不会跟当前存在的广播冲突

ArrayMap>stickies=mStickyBroadcasts.get(

UserHandle.USER\_ALL);

if(stickies!=null){

//根据Action获取action相同的Intent列表

ArrayListlist=stickies.get(intent.getAction());

if(list!=null){

intN=list.size();

inti;

for(i=0;i //粘性广播发送后是会保存下来的，故如果已经存在则不需要重新发送

if(intent.filterEquals(list.get(i))){

thrownewIllegalArgumentException(

"Stickybroadcast"+intent+"foruser"

+userId+"conflictswithexistingglobalbroadcast");

}

}

}

}

}

//在mStickyBroadcasts中根据参数userId查找以Action、广播列表为键值对的stickies，

//如果不存在，则创建并添加

ArrayMap>stickies=mStickyBroadcasts.get(userId);

if(stickies==null){

stickies=newArrayMap<>();

mStickyBroadcasts.put(userId,stickies);

}

//在stickies中查找是否存在与参数intent的Action名称对应的一个粘性广播列表list，

//如果不存在，则创建并添加

ArrayListlist=stickies.get(intent.getAction());

if(list==null){

list=newArrayList<>();

stickies.put(intent.getAction(),list);

}

finalintstickiesCount=list.size();

inti;

//遍历检查在粘性广播列表list中是否存在一个与参数intent一致的广播

for(i=0;i //如果存在则用intent参数所描述的广播来替换它

if(intent.filterEquals(list.get(i))){

//Thisstickyalreadyexists,replaceit.

list.set(i,newIntent(intent));

break;

}

}

//说明list列表中不存在与intent参数一致的广播

if(i>=stickiesCount){

//则把该intent所描述的广播添加到list列表中

list.add(newIntent(intent));

}

}

int[]users;

if(userId==UserHandle.USER\_ALL){

//Callerwantsbroadcasttogotoallstartedusers.

users=mStartedUserArray;

}else{

//Callerwantsbroadcasttogotoonespecificuser.

users=newint[]{userId};

}

//SmartContainerModifiedbegin

intalternativeBoxId=userId;

SetboxesToReceive=newHashSet();

boxesToReceive.add(userId);

//SmartContainerModifiedend

//Figureoutwhoallwillreceivethisbroadcast.

Listreceivers=null;//静态广播接收器列表

ListregisteredReceivers=null;//动态广播接收器列表

//Needtoresolvetheintenttointerestedreceivers...

if((intent.getFlags()&Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY)

==0){

//之前讲解注册广播时，静态注册的广播都保存在PMS中的mReceivers中了，

//现在到PMS中找到所有静态注册的目标广播接收者，并保存在列表receivers中

receivers=collectReceiverComponents(intent,resolvedType,callingUid,users);

}

//SmartContainermodifiedbegin

if(!SmartContainerConfig.WITH\_OUT\_VIRTUAL\_BOX){

if(receivers!=null&&receivers.size()>0&&callerPackage!=null){

mAMSFunc.processReceiverComponents(boxesToReceive,receivers,callerPackage,userId);

}

}

//SmartContainermodifiedend

//没有指定接收者组件名

if(intent.getComponent()==null){

if(userId==UserHandle.USER\_ALL&&callingUid==Process.SHELL\_UID){

//Queryonetargetuseratatime,excludingshell-restrictedusers

UserManagerServiceums=getUserManagerLocked();

for(inti=0;i if(ums.hasUserRestriction(

UserManager.DISALLOW\_DEBUGGING\_FEATURES,users[i])){

continue;

}

ListregisteredReceiversForUser=

mReceiverResolver.queryIntent(intent,

resolvedType,false,users[i]);

if(registeredReceivers==null){

registeredReceivers=registeredReceiversForUser;

}elseif(registeredReceiversForUser!=null){

registeredReceivers.addAll(registeredReceiversForUser);

}

}

}else{

//之前讲解注册广播时，动态注册的广播都存放在AMS的mReceiverResolver中了，

//这里就在里面找到动态注册的目标广播接收者，并保存在registeredReceivers列表中

registeredReceivers=mReceiverResolver.queryIntent(intent,

resolvedType,false,userId);

//SmartContainermodifiedbegin

if(userId!=UserHandle.USER\_ALL&&!SmartContainerConfig.WITH\_OUT\_VIRTUAL\_BOX&&

(registeredReceivers==null||registeredReceivers.size()==0)){

alternativeBoxId=mAMSFunc.processRegisterReceivers(registeredReceivers,

boxesToReceive,intent,resolvedType,userId,alternativeBoxId,callingUid);

}

//SmartContainermodifiedend

}

}

//查看intent的flag有没有设置FLAG\_RECEIVER\_REPLACE\_PENDING，如果设置的话，

//AMS就会在当前的系统中查看有没有相同的intent还未处理，如果有的话，就用当前这个新的intent

//来替换旧的intent。

finalbooleanreplacePending=

(intent.getFlags()&Intent.FLAG\_RECEIVER\_REPLACE\_PENDING)!=0;

if(DEBUG\_BROADCAST)Slog.v(TAG\_BROADCAST,"Enqueingbroadcast:"+intent.getAction()

+"replacePending="+replacePending);

intNR=registeredReceivers!=null?registeredReceivers.size():0;

//参数ordered标记当前发送的广播是否是有序广播，如果不是，并且存在动态注册的目标广播接收者

if(!ordered&&NR>0){

//Ifwearenotserializingthisbroadcast,thensendthe

//registeredreceiversseparatelysotheydon'twaitforthe

//componentstobelaunched.

//broadcastQueueForIntent方法判断要发送的广播是前台广播还是后台广播，如果是前台广播则返回前台广播队列，

//不同队列处理超时的时间不一样：前台10秒、后台60秒

finalBroadcastQueuequeue=broadcastQueueForIntent(intent);

//SmartContainermodifiedbegin

if(SmartContainerConfig.WITH\_OUT\_APP\_CLONE||boxesToReceive.size()==1

||userId==UserHandle.USER\_ALL){

//将intent所描述的广播，以及动态注册的目标广播接收者封装成一个BroadcastRecord对象r，

//用来描述AMS要执行的一个广播转发任务

BroadcastRecordr=newBroadcastRecord(queue,intent,callerApp,

callerPackage,callingPid,callingUid,resolvedType,requiredPermissions,

appOp,brOptions,registeredReceivers,resultTo,resultCode,resultData,

resultExtras,ordered,sticky,false,alternativeBoxId);

Slog.v(TAG\_BROADCAST,"Enqueueingparallelbroadcast"+r+"intent"+intent);

//replaceParallelBroadcastLocked方法根据r到queue中的无序调度队列中查找是否存在与intent描述一致的广播，存在则替换并返回true

finalbooleanreplaced=replacePending&&queue.replaceParallelBroadcastLocked(r);

//如果replaced为true，说明不需要在无序广播调度队列中增加新的广播转发任务

if(!replaced){

//否者就把r所描述的广播转发任务放在BroadcastQueue类中的mParallelBroadcasts无序调度队列中

queue.enqueueParallelBroadcastLocked(r);

//重新调度这个队列中的广播转发任务，从这里可以看出动态注册的广播接收者

//比静态注册的广播接收者优先接收到无序广播

queue.scheduleBroadcastsLocked();

}

}else{

ListbroadcastFilterList=newArrayList();

BroadcastRecordr=newBroadcastRecord(queue,intent,callerApp,

callerPackage,callingPid,callingUid,resolvedType,requiredPermissions,

appOp,brOptions,broadcastFilterList,resultTo,resultCode,resultData,

resultExtras,ordered,sticky,false,-1);

mAMSFunc.queueNOrderedRegisteredBroadcastForClone(registeredReceivers,r,boxesToReceive,intent,replacePending);

}

//SmartContainermodifiedend

//到这里，对于无序广播来说，AMS就相当于已经将参数intent描述的广播发给那些动态注册的目标广播接收者了。

//故，这里就将列表registeredReceivers设置为null，将标记动态注册的目标广播接收者个数的变量NR设置为0

registeredReceivers=null;

NR=0;

}

//执行到这里，无论AMS当前接收到的是一个无序广播还是有序广播，都会将该广播及其目标广播接收者封装成一个广播转发任务，

//并添加到一个有序广播调度队列中。但对于无序广播来说，当它们被真正转发时，并不会按照有序广播来转发。

//Mergeintoonelist.

intir=0;

if(receivers!=null){

//AspecialcaseforPACKAGE\_ADDED:donotallowthepackage

//beingaddedtoseethisbroadcast.Thispreventsthemfrom

//usingthisasabackdoortogetrunassoonastheyare

//installed.Maybeinthefuturewewanttohaveaspecialinstall

//broadcastorsuchforapps,butwe'dliketodeliberatelymake

//thisdecision.

StringskipPackages[]=null;

if(Intent.ACTION\_PACKAGE\_ADDED.equals(intent.getAction())

||Intent.ACTION\_PACKAGE\_RESTARTED.equals(intent.getAction())

||Intent.ACTION\_PACKAGE\_DATA\_CLEARED.equals(intent.getAction())){

Uridata=intent.getData();

if(data!=null){

//特殊广播查看ssp有没有指定包名，有则赋值给skipPackages

StringpkgName=data.getSchemeSpecificPart();

if(pkgName!=null){

skipPackages=newString[]{pkgName};

}

}

}elseif(Intent.ACTION\_EXTERNAL\_APPLICATIONS\_AVAILABLE.equals(intent.getAction())){

skipPackages=intent.getStringArrayExtra(Intent.EXTRA\_CHANGED\_PACKAGE\_LIST);

}

if(skipPackages!=null&&(skipPackages.length>0)){

for(StringskipPackage:skipPackages){

if(skipPackage!=null){

intNT=receivers.size();

for(intit=0;it ResolveInfocurt=(ResolveInfo)receivers.get(it);

//如果静态注册的广播接收者应用的包名和skipPackage一致，则从receivers移除

if(curt.activityInfo.packageName.equals(skipPackage)){

receivers.remove(it);

it--;

NT--;

}

}

}

}

}

intNT=receivers!=null?receivers.size():0;

intit=0;

ResolveInfocurt=null;

BroadcastFiltercurr=null;

//这里动态注册广播合并的是有序的，因为无序动态广播处理中NR最后被赋值为0了

while(it if(curt==null){

//静态注册的广播是ResolveInfo类型

curt=(ResolveInfo)receivers.get(it);

}

if(curr==null){

//动态注册的广播是BroadcastFilter类型，后面会根据类型判断广播是动态注册还是静态注册的

curr=registeredReceivers.get(ir);

}

//如果动态注册广播接收者优先级高于等于静态广播接收者，则把动态注册的广播接收者插入到当前位置，

//静态注册的广播接收者后移，这说明同优先级动态注册的先于静态注册的接收到广播

if(curr.getPriority()>=curt.priority){

//Insertthisbroadcastrecordintothefinallist.

receivers.add(it,curr);

ir++;

curr=null;

it++;

NT++;

}else{

//SkiptothenextResolveInfointhefinallist.

it++;

curt=null;

}

}

}

//把优先级低于所有静态注册广播接收者的动态广播接收者都追加到receivers列表中的末尾

while(ir if(receivers==null){

receivers=newArrayList();

}

receivers.add(registeredReceivers.get(ir));

ir++;

}

//到这里，对于无序广播来说，静态注册的目标广播接收者就全部保存在列表receivers中了;

//而对于有序广播来说，静态注册和动态注册的目标广播接收者也全部保存在列表receivers中了。

if((receivers!=null&&receivers.size()>0)

||resultTo!=null){

//SmartContainerModifiedbegin

if(SmartContainerConfig.WITH\_OUT\_VIRTUAL\_BOX||userId==UserHandle.USER\_ALL

||receivers==null||receivers.size()==0){

### BroadcastQueuequeue=broadcastQueueForIntent(intent);

//将intent所描述的广播，以及剩余的其他目标广播接收者封装成另外一个BroadcastRecord对象r，

//用来描述AMS要执行的另一个广播转发任务，并且添加到有序广播调度队列中。

BroadcastRecord r=newBroadcastRecord(queue,intent,callerApp,

callerPackage,callingPid,callingUid,resolvedType,

requiredPermissions,appOp,brOptions,receivers,resultTo,resultCode,

resultData,resultExtras,ordered,sticky,false,userId);

Slog.v(TAG\_BROADCAST,"Enqueueingorderedbroadcast"+r

+":prevhad"+queue.mOrderedBroadcasts.size());

Slog.i(TAG\_BROADCAST,

"Enqueueingbroadcast"+r.intent.getAction());

//replaceOrderedBroadcastLocked方法根据r到queue中的有序调度队列中查找是否存在与intent描述一致的广播，存在则替换并返回true

booleanreplaced=replacePending&&queue.replaceOrderedBroadcastLocked(r);

//如果replaced为true，说明不需要在有序广播调度队列中增加新的广播转发任务

if(!replaced){

//否者把r所描述的广播转发任务放在BroadcastQueue类中的mOrderedBroadcasts有序广播调度队列中

queue.enqueueOrderedBroadcastLocked(r);

//重新调度这个队列中的广播转发任务

queue.scheduleBroadcastsLocked();

}

}else{

ListreceiverList=newArrayList();

BroadcastQueuequeue=broadcastQueueForIntent(intent);

BroadcastRecordr=newBroadcastRecord(queue,intent,callerApp,

callerPackage,callingPid,callingUid,resolvedType,

requiredPermissions,appOp,brOptions,receiverList,resultTo,resultCode,

resultData,resultExtras,ordered,sticky,false,-1);

mAMSFunc.queueFinalBroadcastForClone(receivers,r,boxesToReceive,intent,userId,replacePending);

}

//SmartContainerModifiedend

}

returnActivityManager.BROADCAST\_SUCCESS;

}

至此，AMS就找到参数intent所描述广播的目标广播接收者了，并且分别将它们保存在了BroadcastRecord类的无序广播调度队列mParallelBroadcasts(包括动态注册的无序广播接收者)中和有序广播调度队列mOrderedBroadcasts(包括动态注册的有序广播接收者和所有静态注册的广播接收者)中。

接下来，AMS就会调用BroadcastQueue类中的scheduleBroadcastsLocked方法将intent所描述的广播转发给目标广播接收者处理：

## processNextBroadcast(booleanfromMsg)

processNextBroadcast(booleanfromMsg){

/\*可以在这里获取允许自启动的名单\*/

synchronized(mService){

BroadcastRecordr;

if(DEBUG\_BROADCAST)Slog.v(TAG\_BROADCAST,"processNextBroadcast["+mQueueName+"]:+mParallelBroadcasts.size()+"broadcasts,"+mOrderedBroadcasts.size()+"orderedbroadcasts");mService.updateCpuStats();

//fromMsg字段标记是否是从handleMessage中调用的该方法

if(fromMsg){

//设置该参数为false，表示前面发送到消息队列中的BROADCAST\_INTENT\_MSG消息已经被处理了

mBroadcastsScheduled=false;

}

//First,deliveranynon-serializedbroadcastsrightaway.

//循环处理保存在无序广播调度队列mParallelBroadcasts中的广播转发任务，

//即：将保存在无序广播调度队列中的广播发送给它的目标广播接收者(动态无序)处理

while(mParallelBroadcasts.size()>0){

//得到mParallelBroadcasts中保存的第一个广播转发任务r

r=mParallelBroadcasts.remove(0);

r.dispatchTime=SystemClock.uptimeMillis();

r.dispatchClockTime=System.currentTimeMillis();

finalintN=r.receivers.size();

if(DEBUG\_BROADCAST\_LIGHT)Slog.v(TAG\_BROADCAST,"Processingparallelbroadcast["+mQueueName+"]"+r);

for(inti=0;i Objecttarget=r.receivers.get(i);

if(DEBUG\_BROADCAST)Slog.v(TAG\_BROADCAST,"Deliveringnon-orderedon["+mQueueName+"]toregistered"+target+":"+r);

//遍历循环将无序广播发送给每一个目标广播接收者(动态注册的)

deliverToRegisteredReceiverLocked(r,(BroadcastFilter)target,false);

}

//添加r到历史队列中

addBroadcastToHistoryLocked(r);

if(DEBUG\_BROADCAST\_LIGHT)Slog.v(TAG\_BROADCAST,"Donewithparallelbroadcast["

+mQueueName+"]"+r);}

//Nowtakecareofthenextserializedone...

//接下来，继续处理保存在有序广播调度队列mOrderedBroadcasts中的广播转发任务。

//前面讲到有序广播调度队列mOrderedBroadcasts中描述的广播的目标接收者有可能是静态注册的，

//而这些静态注册的目标广播接收者可能还没有被启动起来，因此，将一个广播发送给它们处理时，先要//将它们启动起来。事实上，只需要将它们所运行在的应用程序进程启动起来就可以了，因为当这些应用程序

//进程接收到广播时，就会主动将目标广播接收者启动起来。

//Ifwearewaitingforaprocesstocomeuptohandlethenext

//broadcast,thendonothingatthispoint.Justincase,we

//checkthattheprocesswe'rewaitingforstillexists.

//该参数是用来描述一个正在等待静态注册的目标广播接收者启动起来的广播转发任务的

if(mPendingBroadcast!=null){

if(DEBUG\_BROADCAST\_LIGHT)Slog.v(TAG\_BROADCAST,"processNextBroadcast["+mQueueName+"]:waitingfor"+mPendingBroadcast.curApp);

Boolean isDead;

//检查这个静态注册的目标广播接收者所运行在的应用程序进程是否已经启动起来

synchronized(mService.mPidsSelfLocked){

ProcessRecord proc=mService.mPidsSelfLocked.get(mPendingBroadcast.curApp.pid);

isDead=proc==null||proc.crashing;

}

//如果这个应用程序进程没有死亡，就会继续等待

if(!isDead){

//It'sstillalive,sokeepwaiting

return;

}else{

Slog.w(TAG,"pendingapp["+mQueueName+"]"+mPendingBroadcast.curApp+"diedbeforerespondingtobroadcast");

mPendingBroadcast.state=BroadcastRecord.IDLE;

mPendingBroadcast.nextReceiver=mPendingBroadcastRecvIndex;

mPendingBroadcast=null;

}

}

booleanlooped=false;

//循环在有序广播调度队列mOrderedBroadcasts(动态有序、所有静态)中找到下一个需要处理的广播转发任务

do{

//判断有序广播调度队列中的广播转发任务是否已经处理完了

if(mOrderedBroadcasts.size()==0){

//Nomorebroadcastspending,soalldone!

mService.scheduleAppGcsLocked();

if(looped){

//Ifwehadfinishedthelastorderedbroadcast,then

//makesureallprocesseshavecorrectoomandsched

//adjustments.

mService.updateOomAdjLocked();

}

return;

}

//取出第一个广播转发任务r

r=mOrderedBroadcasts.get(0);

/\*这里可以统计广播转发任务r中是否包含操作widget的Action：ACTION\_APPWIDGET\_ENABLED和

\*ACTION\_APPWIDGET\_DISABLED的包名r.targetComp.getPackageName()和

\*类名r.targetComp.getClassName()以便于后面增加控制自启动的过滤操作\*/

booleanforceReceive=false;

//Ensurethatevenifsomethinggoesawry(出现差错)withthetimeout

//detection(超时检测),wecatch"hung"broadcastshere,discard(丢弃)them,

//andcontinuetomakeprogress.

//

//ThisisonlydoneifthesystemisreadysothatPRE\_BOOT\_COMPLETED

//receiversdon'tgetexecutedwithtimeouts.They'reintendedfor

//onetimeheavyliftingaftersystemupgradesandcantake

//significantamountsoftime.

//获取到r所描述的广播转发任务的目标广播接收者的个数

intnumReceivers=(r.receivers!=null)?r.receivers.size():0;

//检查前一个目标广播接收者是否在规定时间内处理完广播

if(mService.mProcessesReady&&r.dispatchTime>0){

longnow=SystemClock.uptimeMillis();

if((numReceivers>0)&&

(now>r.dispatchTime+(2\*mTimeoutPeriod\*numReceivers))){

Slog.w(TAG,"Hungbroadcast["

+mQueueName+"]discardedaftertimeoutfailure:"

+"now="+now+"dispatchTime="+r.dispatchTime+"startTime="+r.receiverTime+"intent="+r.intent+"numReceivers="+numReceivers+"nextReceiver="+r.nextReceiver+"state="+r.state);

//如果目标广播接收者不能按时处理完广播，就强制结束这个广播转发任务

broadcastTimeoutLocked(false);//forcibly(强制)finishthisbroadcast

//下面两个参数的设置表示要继续处理有序广播调度队列中的下一个广播转发任务

forceReceive=true;

r.state=BroadcastRecord.IDLE;

}

}

//检查r所描述的广播转发任务是否正在处理中，即：正在将一个有序广播转发给它的前一个目标广播接收者处理

if(r.state!=BroadcastRecord.IDLE){

if(DEBUG\_BROADCAST)Slog.d(TAG\_BROADCAST,"processNextBroadcast("+mQueueName+")calledwhennotidle(state="+r.state+")");

//如果正在将r所描述的广播转发给它的前一个目标接收者处理，则需要等待这个目标广播接收者处理完

//该有序广播，然后再转发给下一个目标接收者处理。故这里直接返回

return;

}

//检查r所描述的广播转发任务是否已经处理完成或者已经被强制结束了

if(r.receivers==null||r.nextReceiver>=numReceivers||r.resultAbort||forceReceive){

//Nomorereceiversforthisbroadcast!Sendthefinal

//resultifrequested...

if(r.resultTo!=null){

try{

if(DEBUG\_BROADCAST)Slog.i(TAG\_BROADCAST,"Finishingbroadcast["+mQueueName+"]"+r.intent.getAction()+"app="+r.callerApp);

//执行广播的发送操作

performReceiveLocked(r.callerApp,r.resultTo,newIntent(r.intent),r.resultCode,r.resultData,r.resultExtras,false,false,r.userId);

//Setthistonullsothatthereference

//(localandremote)isn'tkeptinthemBroadcastHistory.

r.resultTo=null;

}catch(RemoteExceptione){

r.resultTo=null;

Slog.w(TAG,"Failure["+mQueueName+"]sendingbroadcastresultof"+r.intent,e);

}

}

if(DEBUG\_BROADCAST)Slog.v(TAG\_BROADCAST,"CancellingBROADCAST\_TIMEOUT\_MSG");

//remove掉前面给mHandler发送的BROADCAST\_TIMEOUT\_MSG消息，

//表示r所描述的广播转发任务已经在规定时间内处理完了

cancelBroadcastTimeoutLocked();

if(DEBUG\_BROADCAST\_LIGHT)Slog.v(TAG\_BROADCAST,

"Finishedwithorderedbroadcast"+r);

//...andontothenext...

addBroadcastToHistoryLocked(r);

//将r所描述的广播转发任务从有序广播队列中删除

mOrderedBroadcasts.remove(0);

//这里将r设为null以便继续执行while循环来找到下一个需要处理的广播转发任务

r=null;

looped=true;

continue;

}

}while(r==null);

//上面循环执行完后，下一个需要处理的广播转发任务就保存在r中了

//r所描述的广播转发任务的目标广播接收者保存在它的成员变量receivers列表中，

//而下一个目标广播接收者就保存在它的成员变量nextReceiver中，这里得到的是

//r所描述的广播转发任务的下一个目标广播接收者在其目标广播接收者列表中的位置

//Getthenextreceiver...

intrecIdx=r.nextReceiver++;

//Keeptrackofwhenthisreceiverstarted,andmakesurethere

//isatimeoutmessagependingtokillitifneedbe.

r.receiverTime=SystemClock.uptimeMillis();

//如果recIdx为0说明广播转发任务刚开始被处理

if(recIdx==0){

//广播刚被处理，保存当前时间到r.dispatchTime变量中

r.dispatchTime=r.receiverTime;

r.dispatchClockTime=System.currentTimeMillis();

if(DEBUG\_BROADCAST\_LIGHT)Slog.v(TAG\_BROADCAST,"Processingorderedbroadcast["+mQueueName+"]"+r);

}

//检查是否已经向消息队列发送了BROADCAST\_TIMEOUT\_MSG消息

if(!mPendingBroadcastTimeoutMessage){

longtimeoutTime=r.receiverTime+mTimeoutPeriod;

if(DEBUG\_BROADCAST)Slog.v(TAG\_BROADCAST,"SubmittingBROADCAST\_TIMEOUT\_MSG["+mQueueName+"]for"+r+"at"+timeoutTime);

//如果还没有发送，则向消息队列发送该消息，并且指定它在timeoutTime时间后处理

setBroadcastTimeoutLocked(timeoutTime);

}

finalBroadcastOptionsbrOptions=r.options;

//取出r所描述的广播转发任务的下一个目标广播接收者

finalObjectnextReceiver=r.receivers.get(recIdx);

//前面说过动态注册广播接收者是BroadcastFilter类型的，这里处理动态有序广播接收者

if(nextReceiverinstanceofBroadcastFilter){

//Simplecase:thisisaregisteredreceiverwhogets

//adirectcall.

BroadcastFilter filter=(BroadcastFilter)nextReceiver;

if(DEBUG\_BROADCAST)Slog.v(TAG\_BROADCAST,"Deliveringordered["+mQueueName+"]toregistered"+filter+":"+r);

//这里直接向动态注册的广播接收者发送广播，因为动态注册的广播接收者肯定是已经启动起来的

deliverToRegisteredReceiverLocked(r,filter,r.ordered);

//判断r所描述的广播转发任务是否是用来转发无序广播的

if(r.receiver==null||!r.ordered){

//Thereceiverhasalreadyfinished,soscheduleto

//processthenextone.

if(DEBUG\_BROADCAST)Slog.v(TAG\_BROADCAST,"Quickfinishing["+mQueueName+"]:ordered="+r.ordered+"receiver="+r.receiver);

//如果是则将r.state设置为IDLE，表示不需要等待它的前一个目标广播接收者处理完成一个广播，//就可以将该广播继续发送给它的下一个目标广播接收者处理

r.state=BroadcastRecord.IDLE;

//为了将广播继续发送给r所描述的广播转发任务的下一个目标广播接收者处理，

//方法中实现发送BROADCAST\_INTENT\_MSG消息给消息队列以便继续处理

scheduleBroadcastsLocked();

}else{

if(brOptions!=null&&brOptions.getTemporaryAppWhitelistDuration()>0){

scheduleTempWhitelistLocked(filter.owningUid,

brOptions.getTemporaryAppWhitelistDuration(),r);

}

}

return;

}

### 静态注册广播接收者

//如果r所描述的广播转发任务的下一个目标广播接收者不是BroadcastFilter类型的，那就说明这是一个**静态注册的广播接收者，这种情况略复杂，因为静态注册的广播接收者可能还没有被启动起来**

//Hardcase:needtoinstantiatethereceiver,possibly

//startingitsapplicationprocesstohostit.

//前面说过，静态注册的广播接收者是ResolveInfo类型的，这里直接强转

ResolveInfo info=(ResolveInfo)nextReceiver;

ComponentName component=newComponentName(info.activityInfo.applicationInfo.packageName,info.activityInfo.name);

//下面主要是根据是否得到权限来决定是否跳过本次广播的发送

Boolean skip=false;

//根据目标广播接收者所需要的权限检查发送者的Pid和Uid是否符合要求

Int perm=mService.checkComponentPermission(info.activityInfo.permission,r.callingPid,r.callingUid,info.activityInfo.applicationInfo.uid,info.activityInfo.exported);

//如果发送者不符合要求，则直接跳过本次转发

if(perm!=PackageManager.PERMISSION\_GRANTED){

if(!info.activityInfo.exported){

Slog.w(TAG,"PermissionDenial:broadcasting"+r.intent.toString()+"from"+r.callerPackage+"(pid="+r.callingPid+",uid="+r.callingUid+")"+"isnotexportedfromuid"+info.activityInfo.applicationInfo.uid+"duetoreceiver"+component.flattenToShortString());

}else{

Slog.w(TAG,"PermissionDenial:broadcasting"+r.intent.toString()+"from"+r.callerPackage+"(pid="+r.callingPid+",uid="+r.callingUid+")"+"requires"+info.activityInfo.permission+"duetoreceiver"+component.flattenToShortString());

}

skip=true;

//如果发送者符合要求，就检查发送者申请的权限是否被用户拒绝，拒绝的话也直接跳过本次转发

}elseif(info.activityInfo.permission!=null){

finalintopCode=AppOpsManager.permissionToOpCode(info.activityInfo.permission);

if(opCode!=AppOpsManager.OP\_NONE

&&mService.mAppOpsService.noteOperation(opCode,r.callingUid,

r.callerPackage)!=AppOpsManager.MODE\_ALLOWED){

Slog.w(TAG,"AppopDenial:broadcasting"

+r.intent.toString()

+"from"+r.callerPackage+"(pid="

+r.callingPid+",uid="+r.callingUid+")"

+"requiresappop"+AppOpsManager.permissionToOp(

info.activityInfo.permission)

+"duetoregisteredreceiver"

+component.flattenToShortString());

skip=true;

}

}

/\*可以添加关联唤醒的判断逻辑：如根据目标广播接收者的包名/类名前缀判断是否属于第三方push平台，如果是则设置skip为true\*/

//检查接收者权限

if(!skip&&info.activityInfo.applicationInfo.uid!=Process.SYSTEM\_UID&&

r.requiredPermissions!=null&&r.requiredPermissions.length>0){

//循环根据发送者所需的权限检查所有的目标广播接收者的Pid和Uid是否符合要求，一项不符合就直接跳过

//符合要求，就检查接收者申请的权限是否被用户拒绝，拒绝的话也直接跳过

for(inti=0;i StringrequiredPermission=r.requiredPermissions[i];

try{

perm=AppGlobals.getPackageManager().

checkPermission(requiredPermission,

info.activityInfo.applicationInfo.packageName,

UserHandle

.getUserId(info.activityInfo.applicationInfo.uid));

}catch(RemoteExceptione){

perm=PackageManager.PERMISSION\_DENIED;

}

if(perm!=PackageManager.PERMISSION\_GRANTED){

Slog.w(TAG,"PermissionDenial:receiving"

+r.intent+"to"

+component.flattenToShortString()

+"requires"+requiredPermission

+"duetosender"+r.callerPackage

+"(uid"+r.callingUid+")");

skip=true;

break;

}

intappOp=AppOpsManager.permissionToOpCode(requiredPermission);

if(appOp!=AppOpsManager.OP\_NONE&&appOp!=r.appOp

&&mService.mAppOpsService.noteOperation(appOp,

info.activityInfo.applicationInfo.uid,info.activityInfo.packageName)

!=AppOpsManager.MODE\_ALLOWED){

Slog.w(TAG,"AppopDenial:receiving"

+r.intent+"to"

+component.flattenToShortString()

+"requiresappop"+AppOpsManager.permissionToOp(

requiredPermission)

+"duetosender"+r.callerPackage

+"(uid"+r.callingUid+")");

skip=true;

break;

}

}

}

if(!skip&&r.appOp!=AppOpsManager.OP\_NONE

&&mService.mAppOpsService.noteOperation(r.appOp,

info.activityInfo.applicationInfo.uid,info.activityInfo.packageName)

!=AppOpsManager.MODE\_ALLOWED){

Slog.w(TAG,"AppopDenial:receiving"

+r.intent+"to"

+component.flattenToShortString()

+"requiresappop"+AppOpsManager.opToName(r.appOp)

+"duetosender"+r.callerPackage

+"(uid"+r.callingUid+")");

skip=true;

}

if(!skip){

skip=!mService.mIntentFirewall.checkBroadcast(r.intent,r.callingUid,

r.callingPid,r.resolvedType,info.activityInfo.applicationInfo.uid);

}

booleanisSingleton=false;

try{

isSingleton=mService.isSingleton(info.activityInfo.processName,

info.activityInfo.applicationInfo,

info.activityInfo.name,info.activityInfo.flags);

}catch(SecurityExceptione){

Slog.w(TAG,e.getMessage());

skip=true;

}

if((info.activityInfo.flags&ActivityInfo.FLAG\_SINGLE\_USER)!=0){

if(ActivityManager.checkUidPermission(

android.Manifest.permission.INTERACT\_ACROSS\_USERS,

info.activityInfo.applicationInfo.uid)

!=PackageManager.PERMISSION\_GRANTED){

Slog.w(TAG,"PermissionDenial:Receiver"+component.flattenToShortString()

+"requestsFLAG\_SINGLE\_USER,butappdoesnothold"

+android.Manifest.permission.INTERACT\_ACROSS\_USERS);

skip=true;

}

}

if(r.curApp!=null&&r.curApp.crashing){

//Ifthetargetprocessiscrashing,justskipit.

Slog.w(TAG,"Skippingdeliverordered["+mQueueName+"]"+r

+"to"+r.curApp+":processcrashing");

skip=true;

}

if(!skip){

booleanisAvailable=false;

try{

isAvailable=AppGlobals.getPackageManager().isPackageAvailable(

info.activityInfo.packageName,

UserHandle.getUserId(info.activityInfo.applicationInfo.uid));

}catch(Exceptione){

//allsuchfailuresmeanweskipthisreceiver

Slog.w(TAG,"Exceptiongettingrecipientinfofor"

+info.activityInfo.packageName,e);

}

if(!isAvailable){

if(DEBUG\_BROADCAST)Slog.v(TAG\_BROADCAST,

"Skippingdeliveryto"+info.activityInfo.packageName+"/"

+info.activityInfo.applicationInfo.uid

+":packagenolongeravailable");

skip=true;

}

}

if(skip){

if(DEBUG\_BROADCAST)Slog.v(TAG\_BROADCAST,

"Skippingdeliveryofordered["+mQueueName+"]"

+r+"forwhateverreason");

r.receiver=null;

r.curFilter=null;

r.state=BroadcastRecord.IDLE;

//如果跳过，直接继续发送给r所描述的广播转发任务的下一个目标广播接收者处理

scheduleBroadcastsLocked();

return;

}

r.state=BroadcastRecord.APP\_RECEIVE;

//得到静态注册的广播接收者的android:process属性值，即它需要运行在的应用程序进程的名字

StringtargetProcess=info.activityInfo.processName;

r.curComponent=component;

finalintreceiverUid=info.activityInfo.applicationInfo.uid;

//Ifit'sasingleton,itneedstobethesameapporaspecialapp

if(r.callingUid!=Process.SYSTEM\_UID&&isSingleton

&&mService.isValidSingletonCall(r.callingUid,receiverUid)){

info.activityInfo=mService.getActivityInfoForUser(info.activityInfo,0);

}

r.curReceiver=info.activityInfo;

if(DEBUG\_MU&&r.callingUid>UserHandle.PER\_USER\_RANGE){

Slog.v(TAG\_MU,"Updatedbroadcastrecordactivityinfoforsecondaryuser,"

+info.activityInfo+",callingUid="+r.callingUid+",uid="

+info.activityInfo.applicationInfo.uid);

}

if(brOptions!=null&&brOptions.getTemporaryAppWhitelistDuration()>0){

scheduleTempWhitelistLocked(receiverUid,

brOptions.getTemporaryAppWhitelistDuration(),r);

}

//Broadcastisbeingexecuted,itspackagecan'tbestopped.

try{

AppGlobals.getPackageManager().setPackageStoppedState(

r.curComponent.getPackageName(),false,UserHandle.getUserId(r.callingUid));

}catch(RemoteExceptione){

}catch(IllegalArgumentExceptione){

Slog.w(TAG,"Failedtryingtounstoppackage"

+r.curComponent.getPackageName()+":"+e);

}

//Isthisreceiver'sapplicationalreadyrunning?

ProcessRecordapp=mService.getProcessRecordLocked(targetProcess,

info.activityInfo.applicationInfo.uid,false);

//判断静态注册的广播接收者所运行在的应用程序进程是否已经启动起来

if(app!=null&&app.thread!=null){

try{

app.addPackage(info.activityInfo.packageName,

info.activityInfo.applicationInfo.versionCode,mService.mProcessStats);

//如果已经启动，则直接将广播发送给它处理

processCurBroadcastLocked(r,app);

return;

}catch(RemoteExceptione){

Slog.w(TAG,"Exceptionwhensendingbroadcastto"

+r.curComponent,e);

}catch(RuntimeExceptione){

Slog.wtf(TAG,"Failedsendingbroadcastto"

+r.curComponent+"with"+r.intent,e);

//Ifsomeunexpectedexceptionhappened,justskip

//thisbroadcast.Atthispointwearenotinthecall

//fromaclient,sothrowinganexceptionoutfromhere

//willcrashtheentiresysteminsteadofjustwhoever

//sentthebroadcast.

logBroadcastReceiverDiscardLocked(r);

finishReceiverLocked(r,r.resultCode,r.resultData,

r.resultExtras,r.resultAbort,false);

scheduleBroadcastsLocked();

//Weneedtoresetthestateifwefailedtostartthereceiver.

r.state=BroadcastRecord.IDLE;

return;

}

//Ifadeadobjectexceptionwasthrown--fallthroughto

//restarttheapplication.

}

//Notrunning--getitstarted,tobeexecutedwhentheappcomesup.

if(DEBUG\_BROADCAST)Slog.v(TAG\_BROADCAST,

"Needtostartapp["+mQueueName+"]"+targetProcess+"forbroadcast"+r);

/\*因为下面要开始启动进程了，故这里可以增加禁止自启动判断：如果要启动进程的包名在黑名单中或者不是默认允许启动的，则直接执行启动失败的逻辑\*/

//调用startProcessLocked方法启动这个应用程序进程

if((r.curApp=mService.startProcessLocked(targetProcess,info.activityInfo.applicationInfo,true,r.intent.getFlags()|Intent.FLAG\_FROM\_BACKGROUND,"broadcast",r.curComponent,(r.intent.getFlags()&Intent.FLAG\_RECEIVER\_BOOT\_UPGRADE)!=0,false,false))==null){

//Ah,thisrecipientisunavailable.Finishitifnecessary,

//andmarkthebroadcastrecordasreadyforthenext.

Slog.w(TAG,"Unabletolaunchapp"+info.activityInfo.applicationInfo.packageName+"/"+info.activityInfo.applicationInfo.uid+"forbroadcast"+r.intent+":process is bad");

logBroadcastReceiverDiscardLocked(r);

finishReceiverLocked(r,r.resultCode,r.resultData,r.resultExtras,r.resultAbort,false);

//如果进程启动失败，继续发送给r所描述的广播转发任务的下一个目标广播接收者处理

scheduleBroadcastsLocked();

r.state=BroadcastRecord.IDLE;

return;

}

//如果成功启动进程，则保存r和recIdx，表示正在等待r所描述的广播转发任务的

//下一个目标广播接收者所在的应用程序进程启动起来

mPendingBroadcast=r;

mPendingBroadcastRecvIndex=recIdx;

}

}

假设r所描述的广播转发任务的下一个目标广播接收者是一个动态注册的广播接收者，那么执行完上面步骤后，接下来就会调用BroadcastQueue类的deliverToRegisteredReceiverLocked方法将一个广播转发给它处理：

。

## ActivityThread的内部类ApplicationThread的scheduleRegisteredReceiver

下面看ActivityThread的内部类ApplicationThread的scheduleRegisteredReceiver方法：

来看LoadedApk类中的内部类ReceiverDispatcher中的performReceive方法：

说到这里，我们分析了动态注册的广播接收者处理有序广播和无序广播的过程，下面说下静态广播的处理过程，看BroadcastQueue类中的processCurBroadcastLocked方法：

[java]view plaincopy

在CODE上查看代码片

派生到我的代码片

privatefinalvoidprocessCurBroadcastLocked(BroadcastRecordr,

ProcessRecordapp)throwsRemoteException{

if(DEBUG\_BROADCAST)Slog.v(TAG\_BROADCAST,

"Processcurbroadcast"+r+"forapp"+app);

//如果目标广播接收者所运行在的应用程序进程没有启动起来，则抛异常

if(app.thread==null){

thrownewRemoteException();

}

r.receiver=app.thread.asBinder();

r.curApp=app;

app.curReceiver=r;

app.forceProcessStateUpTo(ActivityManager.PROCESS\_STATE\_RECEIVER);

mService.updateLruProcessLocked(app,false,null);

mService.updateOomAdjLocked();

//Telltheapplicationtolaunchthisreceiver.

r.intent.setComponent(r.curComponent);

booleanstarted=false;

try{

if(DEBUG\_BROADCAST\_LIGHT)Slog.v(TAG\_BROADCAST,

"Deliveringtocomponent"+r.curComponent

+":"+r);

mService.ensurePackageDexOpt(r.intent.getComponent().getPackageName());

//调用运行在该应用程序进程中的一个ApplicationThread对象的Binder代理对象的

//scheduleReceiver方法来向它发送这个广播

app.thread.scheduleReceiver(newIntent(r.intent),r.curReceiver,

mService.compatibilityInfoForPackageLocked(r.curReceiver.applicationInfo),

r.resultCode,r.resultData,r.resultExtras,r.ordered,r.userId,

app.repProcState);

if(DEBUG\_BROADCAST)Slog.v(TAG\_BROADCAST,

"Processcurbroadcast"+r+"DELIVEREDforapp"+app);

started=true;

}finally{

if(!started){

if(DEBUG\_BROADCAST)Slog.v(TAG\_BROADCAST,

"Processcurbroadcast"+r+":NOTSTARTED!");

r.receiver=null;

r.curApp=null;

app.curReceiver=null;

}

}

}

下面看ActivityThread的内部类ApplicationThread的scheduleReceiver方法：

[java]view plaincopy

在CODE上查看代码片

派生到我的代码片

publicfinalvoidscheduleReceiver(Intentintent,ActivityInfoinfo,

CompatibilityInfocompatInfo,intresultCode,Stringdata,Bundleextras,

booleansync,intsendingUser,intprocessState){

updateProcessState(processState,false);

//将参数封装成一个ReceiverData对象，通过sendMessage方法向应用程序主线程消息队列中发送一个RECEIVER消息

ReceiverDatar=newReceiverData(intent,resultCode,data,extras,

sync,false,mAppThread.asBinder(),sendingUser);

r.info=info;

r.compatInfo=compatInfo;

sendMessage(H.RECEIVER,r);

}

privateclassHextendsHandler{

...

publicstaticfinalintRECEIVER=113;

publicvoidhandleMessage(Messagemsg){

switch(msg.what){

...

caseRECEIVER:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER,"broadcastReceiveComp");

handleReceiver((ReceiverData)msg.obj);

maybeSnapshot();

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

}

}

}

privatevoidhandleReceiver(ReceiverDatadata){

//Ifwearegettingreadytogcaftergoingtothebackground,well

//wearebackactivesoskipit.

unscheduleGcIdler();

//目标广播接收者类名

Stringcomponent=data.intent.getComponent().getClassName();

LoadedApkpackageInfo=getPackageInfoNoCheck(

data.info.applicationInfo,data.compatInfo);

IActivityManagermgr=ActivityManagerNative.getDefault();

BroadcastReceiverreceiver;

try{

java.lang.ClassLoadercl=packageInfo.getClassLoader();

data.intent.setExtrasClassLoader(cl);

data.intent.prepareToEnterProcess();

data.setExtrasClassLoader(cl);

//根据目标广播接收者类名实例化一个接收对象

receiver=(BroadcastReceiver)cl.loadClass(component).newInstance();

}catch(Exceptione){

if(DEBUG\_BROADCAST)Slog.i(TAG,

"Finishingfailedbroadcastto"+data.intent.getComponent());

data.sendFinished(mgr);

thrownewRuntimeException(

"Unabletoinstantiatereceiver"+component

+":"+e.toString(),e);

}

try{

Applicationapp=packageInfo.makeApplication(false,mInstrumentation);

if(localLOGV)Slog.v(

TAG,"Performingreceiveof"+data.intent

+":app="+app

+",appName="+app.getPackageName()

+",pkg="+packageInfo.getPackageName()

+",comp="+data.intent.getComponent().toShortString()

+",dir="+packageInfo.getAppDir());

ContextImplcontext=(ContextImpl)app.getBaseContext();

sCurrentBroadcastIntent.set(data.intent);

receiver.setPendingResult(data);

//回调目标广播接收者的onReceive方法来接收这个广播

receiver.onReceive(context.getReceiverRestrictedContext(),

data.intent);

}catch(Exceptione){

if(DEBUG\_BROADCAST)Slog.i(TAG,

"Finishingfailedbroadcastto"+data.intent.getComponent());

data.sendFinished(mgr);

if(!mInstrumentation.onException(receiver,e)){

thrownewRuntimeException(

"Unabletostartreceiver"+component

+":"+e.toString(),e);

}

}finally{

sCurrentBroadcastIntent.set(null);

}

if(receiver.getPendingResult()!=null){

//处理完本次接收后通知AMS继续给下一个目标广播接收者发送广播

data.finish();

}

}

到这里，广播的发送流程就终于分析完了。

## finishReceiverLocked

还是在settings进程哟

F:/K/1-src/3399/dpad-settings/framework-libs/framework.jar!/android/app/LoadedApk.class

ReceiverDispatcher

ReceiverDispatcher.**this**.mReceiver;=== SummaryProvider **extends** BroadcastReceiver

receiver.onReceive(ReceiverDispatcher.**this**.mContext, intent);

# 优先级设计

## 定义

*/\*\*  
 \* This only affects receiver filters.  
 \* The priority of activity filters are set in XML and cannot be changed  
 \* programatically.*

*The default priority is 0.*

*{****@link*** *#SYSTEM\_LOW\_PRIORITY} and  
 \* smaller than {****@link*** *#SYSTEM\_HIGH\_PRIORITY} .  
  
 \** ***@see*** *#SYSTEM\_LOW\_PRIORITY  
 \** ***@see*** *#SYSTEM\_HIGH\_PRIORITY  
 \*/***public final void** setPriority(**int** priority) {  
 mPriority = priority;  
}

## 实现

ntentResolver<F **extends** IntentFilter, R **extends** Object

queryIntent

**protected void** sortResults(List<R> results) {  
 Collections.sort(results, mResolvePrioritySorter);  
}

**private static final** Comparator mResolvePrioritySorter = **new** Comparator() {  
 **public int** compare(Object o1, Object o2) {  
 **final int** q1 = ((IntentFilter) o1).getPriority();  
 **final int** q2 = ((IntentFilter) o2).getPriority();  
 **return** (q1 > q2) ? -1 : ((q1 < q2) ? 1 : 0);  
 }  
};

# 权限管理

<https://blog.csdn.net/gaixvtian123/article/details/6771210>

Sender:sender是否具有某种Permission与所处的Package申明有关，与sendBroadcast方法是相互独立的。sendBroadcast可以要求接受方需要具备什么Permission。

Receiver: receive是否具有某种Permission与所处的Package申明有关, 与registerReceiver方法是相互独立的。registerReicerver可以要求发送方需要具备什么Permission。

!. receiver 的 android:permission 属性，表示该receiver只接受具有相应权限的Package的Broadcast消息  
        <receiver android:name=".customBroadcastReceiver"  
            android:permission="com.hyman.demo.P"  
        >  
            <intent-filter>  
                    <action android:name="com.hyman.demo.P\_ACTION"></action>  
            </intent-filter>  
        </receiver>

2. registerReceiver(receiver, intentFilter, PERMISSION, null); 表示该receiver只接受具有相应权限的Package的Broadcast消息

3. sendBroadcast(intent, PERMISSION);表示只有具有相应权限的Package才能接受其所发送的Broadcast消息。

4. ※注意双声明权限 此处与activity声明权限不同 。Package要求某自定义权限时，需要同时使用<permission> Tag 和 <uses-permission> Tag

<permission android:name="com.hyman.demo.P" >     </permission>  
    <uses-permission android:name="com.hyman.demo.P"></uses-permission>

## 常见问题

## 权限依赖

### 问题描述

08-07 22:36:25.049 2770-2789/system\_process W/BroadcastQueue: Permission Denial: broadcasting Intent { act=.intent.action.FULLSCREEN flg=0x10 (has extras) } from.go.v4 (pid=5317, uid=10046) requires android.permission.DUMP due to registered receiver BroadcastFilter{a477391 u-1 ReceiverList{38759b8 4959 com.android.systemui/10039/u-1 remote:631f41b}}

### 问题分析

1. Perm未被授予，dji.go发出intent.action.FULLSCREEN广播，需要申请android.permission.DUMP权限。<= com.android.systemui是接受者，定义的权限。
2. 问题发生在processNextBroadcast已经发送广播后

**if** (!skip && perm != PackageManager.PERMISSION\_GRANTED) {  
 **if** (!info.activityInfo.exported) {  
 Slog.w(TAG, **"Permission Denial: broadcasting "** + r.intent.toString()  
 + **" from "** + r.callerPackage + **" (pid="** + r.callingPid  
 + **", uid="** + r.callingUid + **")"** + **" is not exported from uid "** + info.activityInfo.applicationInfo.uid  
 + **" due to receiver "** + component.flattenToShortString());  
 } **else** {  
 Slog.w(TAG, **"Permission Denial: broadcasting "** + r.intent.toString()  
 + **" from "** + r.callerPackage + **" (pid="** + r.callingPid  
 + **", uid="** + r.callingUid + **")"** + **" requires "** + info.activityInfo.permission  
 + **" due to receiver "** + component.flattenToShortString());  
 }  
 skip = **true**;  
}

# ANR原理

# PMS

而预置到data/app,vendor/app下面的三方apk，若安装从来没有启动过和被用户手动强制停止，它就是处于“stopped state”(此应用的信息保存在/data/system/users/0/package-restrictions.xml中)，它的接收器将会无法接收任何广播

与此同时系统增加了2个Flag：FLAG\_INCLUDE\_STOPPED\_PACKAGES和FLAG\_EXCLUDE\_STOPPED\_PACKAGES ，来标识一个intent是否激活处于“stopped state”的应用。

给广播intent设置FLAG\_INCLUDE\_STOPPED\_PACKAGES来唤醒处于“stopped state”的程序，也就是用户自己写的广播intent可以控制这个机制，但是系统自带的广播intent默认都是FLAG\_EXCLUDE\_STOPPED\_PACKAGES，由于不能修改，所以就没法通过系统广播自启动了。

/\*\*

\* Broadcast Action: This is broadcast once, after the user has finished

\* booting. It can be used to perform application-specific initialization,

\* such as installing alarms. You must hold the

\* {@link android.Manifest.permission#RECEIVE\_BOOT\_COMPLETED} permission in

\* order to receive this broadcast.

\* <p>

\* This broadcast is sent at boot by all devices (both with and without

\* direct boot support). Upon receipt of this broadcast, the user is

\* unlocked and both device-protected and credential-protected storage can

\* accessed safely.

\* <p>

\* If you need to run while the user is still locked (before they've entered

\* their lock pattern or PIN for the first time), you can listen for the

\* {@link #ACTION\_LOCKED\_BOOT\_COMPLETED} broadcast.

\* <p class="note">

\* This is a protected intent that can only be sent by the system.

\*/

@SdkConstant(SdkConstantType.BROADCAST\_INTENT\_ACTION)

public static final String ACTION\_BOOT\_COMPLETED = "android.intent.action.BOOT\_COMPLETED";

private void sendPackageAddedForUser(String packageName, boolean isSystem,

int appId, int userId) {

Bundle extras = new Bundle(1);

extras.putInt(Intent.EXTRA\_UID, UserHandle.getUid(userId, appId));

sendPackageBroadcast(Intent.ACTION\_PACKAGE\_ADDED,

packageName, extras, 0, null, null, new int[] {userId});

try {

IActivityManager am = ActivityManagerNative.getDefault();

if (isSystem && am.isUserRunning(userId, 0)) {

// The just-installed/enabled app is bundled on the system, so presumed

// to be able to run automatically without needing an explicit launch.

// Send it a BOOT\_COMPLETED if it would ordinarily have gotten one.

Intent bcIntent = new Intent(Intent.ACTION\_BOOT\_COMPLETED)

.addFlags(Intent.FLAG\_INCLUDE\_STOPPED\_PACKAGES)

.setPackage(packageName);

am.broadcastIntent(null, bcIntent, null, null, 0, null, null, null,

android.app.AppOpsManager.OP\_NONE, null, false, false, userId);

}

} catch (RemoteException e) {

// shouldn't happen

Slog.w(TAG, "Unable to bootstrap installed package", e);

}

# 实例分析

## 问题log

06-12 01:00:05.497 629-1094/system\_process I/ActivityManager: Start proc 1430:com.dpad.update/1000 for broadcast com.dpad.update/.controller.DJIImageUpReceiver

06-13 06:38:46.121 W/ContextImpl( 1065): Calling a method in the system process without a qualified user: android.app.ContextImpl.startService:1358 android.content.ContextWrapper.startService:613 com.dpad.update.activity.DJIApplication.initSync:73 com.dpad.core.DpadApp.onCreate:45 com.dpad.update.activity.DJIApplication.onCreate:113

06-12 01:00:06.536 629-641/system\_process I/BroadcastQueue: Delay finish: com.dpad.update/.controller.DJIImageUpReceiver

06-12 01:00:20.729 1430-1430/com.dpad.update W/ContextImpl: Calling a method in the system process without a qualified user: android.app.ContextImpl.startService:1358

android.content.ContextWrapper.startService:613

android.content.ContextWrapper.startService:613

com.dpad.update.controller.DJIImageUpReceiver.onReceive:31

android.app.ActivityThread.handleReceiver:3095

### AMS.startProcessLocked

Process.ProcessStartResult startResult = **Process.start(e**ntryPoint,

app.processName, uid, uid, gids, debugFlags, mountExternal,

app.info.targetSdkVersion, app.info.seinfo, requiredAbi, instructionSet,

app.info.dataDir, entryPointArgs);

EventLog.writeEvent(EventLogTags.AM\_PROC\_START,

UserHandle.getUserId(uid), startResult.pid, uid,

app.processName, hostingType,

hostingNameStr != null ? hostingNameStr : "");

StringBuilder buf = mStringBuilder;

buf.setLength(0);

buf.append("Start proc ");

buf.append(startResult.pid);

buf.append(':');

buf.append(app.processName);

buf.append('/');

UserHandle.formatUid(buf, uid);

if (!isActivityProcess) {

buf.append(" [");

buf.append(entryPoint);

buf.append("]");

}

buf.append(" for ");

buf.append(hostingType);

if (hostingNameStr != null) {

buf.append(" ");

buf.append(hostingNameStr);

}

Slog.i(TAG, buf.toString());

启动进程

startResult.pid 为1430，

app.processName进程名字为com.dpad.update，

uid=1000

isActivityProcess，是一个activity进程

hostingNameStr= broadcast com.dpad.update/.controller.DJIImageUpReceiver

### BroadcastQueue. finishReceiverLocked

if (mService.mServices.hasBackgroundServices(r.userId)) {

Slog.i(TAG, "Delay finish: " + r.curComponent.flattenToShortString());

r.state = BroadcastRecord.WAITING\_SERVICES;

return false;

}

**ActiveServices**

boolean hasBackgroundServices(int callingUser) {

ServiceMap smap = mServiceMap.get(callingUser);

return smap != null ? smap.mStartingBackground.size() >= mMaxStartingBackground : false;

}

### startService

warnIfCallingFromSystemProcess

## 分析

ActivityManager 发送了开机广播发送到中了， 广播给DJIImageUpReceiver

01-18 08:50:12.162 W/PackageManager( 646): Failed to parse /system/priv-app/dpad-update: Package com.dpad.update at /system/priv-app/dpad-update ignored: updated version 42 better than this 40

### /data/app/com.dpad.update-2/base.apk

com.dpad.update-1

Feature：增加sd升级进度显示

f50b228 Aero.Tang <aero.tang@dji.com> on 2018/6/29 at 23:24

**android:versionCode="42"  
android:versionName="1.0.06.29-f50b2285"**

develop-update-aero

因此这个apk是没有问题的。。

com.dpad.update-2

launcher隐藏图标

147c9ce Aero.Tang <aero.tang@dji.com> on 2018/6/14 at 17:07

**android:versionCode="41"  
android:versionName="1.0.06.29-147c9cee"**

drwxr-xr-x 4 system system 4096 2018-06-12 01:09 com.dpad.update-1

drwxr-xr-x 4 system system 4096 2018-06-12 01:10 com.dpad.update-2

01-18 08:50:12.209 W/PackageParser( 646): Unknown element under <manifest>: meta-data at /data/app/com.dpad.update-1/base.apk Binary XML file line #33

01-18 08:50:12.240 W/PackageParser( 646): Unknown element under <manifest>: meta-data at /data/app/com.dpad.update-2/base.apk Binary XML file line #33

01-18 08:50:12.955 W/PackageManager( 646): Failed to parse /data/app/com.dpad.update-2: Application package com.dpad.update already installed. Skipping duplicate.

先扫到版本1没有改，42

后扫到版本2，版本2没有更新，因此不会被安转

<package name="com.dpad.update" codePath="/data/app/com.dpad.update-1" nativeLibraryPath="/data/app/com.dpad.update-1/lib" primaryCpuAbi="arm64-v8a" publicFlags="944291783" privateFlags="8" ft="163f18a5588" it="11e8dc5d800" ut="163f18a6e60" version="42" sharedUserId="1000" isOrphaned="true">

## 无法监听开机广播

ACTION\_BOOT\_COMPLETED = "android.intent.action.BOOT\_COMPLETED";

BOOT\_COMPLETED对应的action和uses-permission没有一起添加

<action android:name="android.intent.action.BOOT\_COMPLETED" ⁄>

<uses-permission android:name="android.permission.RECEIVE\_BOOT\_COMPLETED" ⁄>

系统广播都有哪些呢？

## 其它应用监听方法

<uses-permission android:name="android.permission.RECEIVE\_BOOT\_COMPLETED" />

<receiver android:name=".BootReceiver" android:enabled="false">

<intent-filter>

<action android:name="android.intent.action.BOOT\_COMPLETED" />

</intent-filter>

</receiver>

### 解决办法startOtherServices？

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);

}

public static final int SYSTEM\_HIGH\_PRIORITY = 1000;

intentFilter = new IntentFilter(Intent.ACTION\_BOOT\_COMPLETED);

intentFilter.setPriority(IntentFilter.SYSTEM\_HIGH\_PRIORITY);

context.registerReceiver(mBootCompletedReceiver, intentFilter);

Android针对高版本实现开机自启动时无法监听BOOT\_COMPLETED的处理方法

第三种： [**Android**](http://lib.csdn.net/base/15):installLocation="internalOnly"， 限制app的安装位置，并且在清单文件加入android:sharedUserId="android.uid.system"，使其能抓取到RECEIVE\_BOOT\_COMPLETED

第四种：也是比较靠谱的一种，就是监听android.intent.action.USER\_PRESENT用户状态广播

## 结论

你的服务是起来了的，为何打印还是无效呢？因为你的app本地版本比更高？每次都被系统覆盖掉了，测试开机广播就日狗了？？根本原因还是没有初始化。。。

这个是系统调起的KKKKKK?UID 导致的？

06-12 01:00:08.734 I/am\_proc\_start( 646): [0,1481,1000,com.dpad.update,broadcast,com.dpad.update/.controller.DJIImageUpReceiver]

06-12 01:00:08.734 I/ActivityManager( 646): Start proc 1481:com.dpad.update/1000 for broadcast com.dpad.update/.controller.DJIImageUpReceiver

不是广播调用的

DJIApplication正常onCreate

06-12 01:00:09.407 W/ContextImpl( 1481): Calling a method in the system process without a qualified user: android.app.ContextImpl.startService:1358

android.content.ContextWrapper.startService:613

com.dpad.update.activity.DJIApplication.initSync:74

com.dpad.core.DpadApp.onCreate:37

com.dpad.update.activity.DJIApplication.onCreate:117

06-12 01:00:10.025 I/BroadcastQueue( 646): Delay finish: com.dpad.update/.controller.DJIImageUpReceiver

delay了，10分钟后触发的

06-12 01:00:20.653 D/DJIImageUpReceiver( 1481): @@@dji@@@intent=Intent { act=android.intent.action.BOOT\_COMPLETED flg=0x9000010 cmp=com.dpad.update/.controller.DJIImageUpReceiver (has extras) }

06-12 01:00:20.654 W/ContextImpl( 1481): Calling a method in the system process without a qualified user: android.app.ContextImpl.startService:1358 android.content.ContextWrapper.startService:613 android.content.ContextWrapper.startService:613 com.dpad.update.controller.DJIImageUpReceiver.onReceive:31 android.app.ActivityThread.handleReceiver:3095

06-12 01:00:20.665 D/DpadLogger( 1481): @@@dji@@@ ACTION\_BOOT\_COMPLETED,startService delete

系统广播有限制的

新安转的在data分区，后面就不行了。。。

Tom的为什么有效呢

=---------------------push 到 data下面

01-18 10:06:26.995 640-640/system\_process W/PackageManager: Failed to parse /data/app/dpad-update-180629-16.apk: Application package com.dpad.update already installed. Skipping duplicate.

01-18 10:06:27.010 640-640/system\_process W/PackageParser: Unknown element under <manifest>: meta-data at /data/app/com.dpad.update-2/base.apk Binary XML file line #33

01-18 10:06:27.012 640-640/system\_process I/PackageManager: /data/app/com.dpad.update-2 changed; collecting certs

01-18 10:06:27.605 640-640/system\_process W/PackageManager: Failed to parse /data/app/com.dpad.update-2: Application package com.dpad.update already installed. Skipping duplicate.

不是听不到，而是太久了，平均

08:22- :00:21；2min，为啥需要那么久

01-18 10:10:46.261 – 06-12 01:00:32.795

00:09

20s的样子。。。

10:46

## checkBroadcastFromSystem分析

### checkBroadcastFromSystem

01:28:20.570 613-627/system\_process E/ActivityManager: Sending non-protected broadcast isentek.intent.action.UPDATE\_DATA from system 771:com.android.settings/1000 pkg com.android.settings

java.lang.Throwable

at com.android.server.am.ActivityManagerService.checkBroadcastFromSystem(ActivityManagerService.java:18300)

at com.android.server.am.ActivityManagerService.broadcastIntentLocked(ActivityManagerService.java:18900)

at com.android.server.am.ActivityManagerService.broadcastIntent(ActivityManagerService.java:18991)

at android.app.ActivityManagerNative.onTransact(ActivityManagerNative.java:502)

at com.android.server.am.ActivityManagerService.onTransact(ActivityManagerService.java:2969)

at android.os.Binder.execTransact(Binder.java:565)

#### 触发条件

**android:sharedUserId="android.uid.system"**

isCallerSystem

final boolean isCallerSystem;

switch (UserHandle.getAppId(callingUid)) {

case Process.ROOT\_UID:

case Process.SYSTEM\_UID:

case Process.PHONE\_UID:

case Process.BLUETOOTH\_UID:

case Process.NFC\_UID:

isCallerSystem = true;

break;

default:

isCallerSystem = (callerApp != null) && callerApp.persistent;

break;

}

<**protected-broadcast android:name="com.android.settings.CARRIER\_PROVISIONING"** />  
<**protected-broadcast android:name="com.android.settings.TRIGGER\_CARRIER\_PROVISIONING"** />

isProtectedBroadcast =

checkBroadcastFromSystem(intent, callerApp, callerPackage, callingUid,

isProtectedBroadcast, registeredReceivers);

#### Wtf忽略条件

##### isProtectedBroadcast

either protected

AppGlobals.getPackageManager().isProtectedBroadcast(action);

##### 系统默认action

|| Intent.ACTION\_CLOSE\_SYSTEM\_DIALOGS.equals(action)

|| Intent.ACTION\_DISMISS\_KEYBOARD\_SHORTCUTS.equals(action)

|| Intent.ACTION\_MEDIA\_BUTTON.equals(action)

|| Intent.ACTION\_MEDIA\_SCANNER\_SCAN\_FILE.equals(action)

|| Intent.ACTION\_SHOW\_KEYBOARD\_SHORTCUTS.equals(action)

|| Intent.ACTION\_MASTER\_CLEAR.equals(action)

|| AppWidgetManager.ACTION\_APPWIDGET\_CONFIGURE.equals(action)

|| AppWidgetManager.ACTION\_APPWIDGET\_UPDATE.equals(action)

|| LocationManager.HIGH\_POWER\_REQUEST\_CHANGE\_ACTION.equals(action)

|| TelephonyIntents.ACTION\_REQUEST\_OMADM\_CONFIGURATION\_UPDATE.equals(action)

|| SuggestionSpan.ACTION\_SUGGESTION\_PICKED.equals(action)

it's a public action that

// we've relaxed, so it's fine for system internals to send.

##### system components that want to send an internal broadcast to themselves

##### 触发

callerApp：797

callerPackage：com.android.settings

Log.wtf(TAG, "Sending non-protected broadcast " + action

+ " from system " + callerApp.toShortString() + " pkg " + callerPackage,

new Throwable());

## 禁用app的静态广播

在Android 8.0 及以上 在xml中注册的广播,在接收的时候收到了额外的限制,如果你的app目标等级是26及以上,将无法接收到xml注册的广播

这是google 为了app注册的静态广播导致耗电加的限制

但是解决办法还是有的

办法一:  
使用动态广播registerReceiver注册形式,这个不受限制(推荐)  
办法二:  
gradle 中的targetSdkVersion 设置小于26

Xml在未来编程，是要禁用静态注册的咯

<**receiver android:name=".widget.SettingsAppWidgetProvider"  
 android:label="@string/gadget\_title"  
 android:exported="false"  
 android:enabled="@bool/has\_powercontrol\_widget"**>

# ContextIMpl

public ComponentName startService(Intent service) {

warnIfCallingFromSystemProcess();

return startServiceCommon(service, mUser);

}

# BootReceiver

Sdf

06-13 06:38:46.339 E/BootReceiver( 577): Error reading /data/system/last-header.txt

06-13 06:38:46.339 E/BootReceiver( 577): java.io.FileNotFoundException: /data/system/last-header.txt (No such file or directory)

06-13 06:38:46.339 E/BootReceiver( 577): at java.io.FileInputStream.open(Native Method)

06-13 06:38:46.339 E/BootReceiver( 577): at java.io.FileInputStream.<init>(FileInputStream.java:146)

06-13 06:38:46.339 E/BootReceiver( 577): at android.os.FileUtils.readTextFile(FileUtils.java:263)

06-13 06:38:46.339 E/BootReceiver( 577): at com.android.server.BootReceiver.getPreviousBootHeaders(BootReceiver.java:121)

06-13 06:38:46.339 E/BootReceiver( 577): at com.android.server.BootReceiver.getBootHeadersToLogAndUpdate(BootReceiver.java:143)

06-13 06:38:46.339 E/BootReceiver( 577): at com.android.server.BootReceiver.logBootEvents(BootReceiver.java:163)

06-13 06:38:46.339 E/BootReceiver( 577): at com.android.server.BootReceiver.-wrap2(BootReceiver.java)

06-13 06:38:46.339 E/BootReceiver( 577): at com.android.server.BootReceiver$1.run(BootReceiver.java:93)

# 核心功能

## warnIfCallingFromSystemProcess系统调用者是系统进程

### 原理作用

//如果调用者是系统进程的话打印log

warnIfCallingFromSystemProcess(){

if (Process.myUid() == Process.SYSTEM\_UID) {

Slog.w(TAG, "Calling a method in the system process without a qualified user: "

+ Debug.getCallers(5));

}

}

如果是系统进程，作用是告之用户后台“偷偷启动”问题？

/\*\*

\* Logs a warning if the system process directly called a method such as

\* {@link #startService(Intent)} instead of {@link #startServiceAsUser(Intent, UserHandle)}.

\* The "AsUser" variants allow us to properly enforce the user's restrictions.

\*/

### 举例

系统进程AMS中发送开机广播，启动对应应用程序

06-13 04:55:09.685 1067-1067/? W/ContextImpl: Calling a method in the system process without a qualified user: android.app.ContextImpl.startService:1358 android.content.ContextWrapper.startService:613 android.content.ContextWrapper.startService:613 com.dpad.update.controller.DJIImageUpReceiver.onReceive:31 android.app.ActivityThread.handleReceiver:3095

## AMS.startProcessLocked

启动一个进程

boolean isActivityProcess = (entryPoint == null);

## ActivityManagerDebugConfig

## checkComponentPermission

/\*\*

\* This can be called with or without the global lock held.

\*/

int checkComponentPermission(String permission, int pid, int uid,

int owningUid, boolean exported) {

if (pid == MY\_PID) {

return PackageManager.PERMISSION\_GRANTED;

}

return ActivityManager.checkComponentPermission(permission, uid,

owningUid, exported);

}

## 解bug，应该系统分析

<http://www.voidcn.com/article/p-qdmldzpw-yz.html>

# checkBroadcastFromSystem

对系统应用发出的广播进行安全检查

## 触发逻辑：isCallerSystem

ActivityServiceManager.java

final int broadcastIntentLocked(){

final boolean isCallerSystem;

switch (UserHandle.getAppId(callingUid)) {

//说明是以下几个uid的进程都认为是system应用进程

case Process.ROOT\_UID:

case Process.SYSTEM\_UID:

case Process.PHONE\_UID:

case Process.BLUETOOTH\_UID:

case Process.NFC\_UID:

isCallerSystem = true;

break;

default:

isCallerSystem = (callerApp != null) && callerApp.persistent;

break;

}

}

这里需要注意的是除了SYSTEM\_UID被认为是系统uid，还有ROOT\_UID，PHONE\_UID，BLUETOOTH\_UID，NFC\_UID，

需要特别注意的是如果都未定义以上的UID，但应用的AndroidManifest.xml中定义了persistent属性为true，即常驻应用，也会被设定为由系统调用，同样需要进行广播的权限检查。

## checkBroadcastFromSystem

df

private void checkBroadcastFromSystem(Intent intent, ProcessRecord callerApp,

String callerPackage, int callingUid, boolean isProtectedBroadcast, List receivers) {

final String action = intent.getAction();

if (isProtectedBroadcast //A如果是受保护的广播或者特殊的公共广播

|| Intent.ACTION\_CLOSE\_SYSTEM\_DIALOGS.equals(action)

|| Intent.ACTION\_DISMISS\_KEYBOARD\_SHORTCUTS.equals(action)

|| Intent.ACTION\_MEDIA\_BUTTON.equals(action)

|| Intent.ACTION\_MEDIA\_SCANNER\_SCAN\_FILE.equals(action)

|| Intent.ACTION\_SHOW\_KEYBOARD\_SHORTCUTS.equals(action)

|| Intent.ACTION\_MASTER\_CLEAR.equals(action)

|| AppWidgetManager.ACTION\_APPWIDGET\_CONFIGURE.equals(action)

|| AppWidgetManager.ACTION\_APPWIDGET\_UPDATE.equals(action)

|| LocationManager.HIGH\_POWER\_REQUEST\_CHANGE\_ACTION.equals(action)

|| TelephonyIntents.ACTION\_REQUEST\_OMADM\_CONFIGURATION\_UPDATE.equals(action)

|| SuggestionSpan.ACTION\_SUGGESTION\_PICKED.equals(action)) {

// Broadcast is either protected, or it's a public action that

// we've relaxed, so it's fine for system internals to send.

return;

}

// This broadcast may be a problem... but there are often system components that

// want to send an internal broadcast to themselves, which is annoying to have to

// explicitly list each action as a protected broadcast, so we will check for that

// one safe case and allow it: an explicit broadcast, only being received by something

// that has protected itself.

//如果该广播收接收者，且指定了接收者的包名或者组件名

if (receivers != null && receivers.size() > 0

&& (intent.getPackage() != null || intent.getComponent() != null)) {

boolean allProtected = true;

for (int i = receivers.size()-1; i >= 0; i--) {

Object target = receivers.get(i);

if (target instanceof ResolveInfo) {

ResolveInfo ri = (ResolveInfo)target;

//如果接收者exported设置为false或者接收者设置了权限，则说明做了保护，allProtect为Ture

if (ri.activityInfo.exported && ri.activityInfo.permission == null) {

allProtected = false;

break;

}

} else {

BroadcastFilter bf = (BroadcastFilter)target;

if (bf.requiredPermission == null) {

allProtected = false;

break;

}

}

}

if (allProtected) {

// All safe!

return;

}

}

//C 这里会打出wtflog，同时在dropbox中会生成wtf文件

// The vast majority of broadcasts sent from system internals

// should be protected to avoid security holes, so yell loudly

// to ensure we examine these cases.

if (callerApp != null) {

Log.wtf(TAG, "Sending non-protected broadcast " + action

+ " from system " + callerApp.toShortString() + " pkg " + callerPackage,

new Throwable());

} else {

Log.wtf(TAG, "Sending non-protected broadcast " + action

+ " from system uid " + UserHandle.formatUid(callingUid)

+ " pkg " + callerPackage,

new Throwable());

}

}

## protected broadcast

### framework中声明的保护广播和一些放开的公共广播

isProtectedBroadcast = AppGlobals.getPackageManager().isProtectedBroadcast(action);

sProtectedBroadcast为true则代表该广播在Framework/base/core/res/AndroidManifest.xml中有声明为保护广播,这样的广播只能由系统发出。如果是phone进程的，一般在Teleservice下的AndroidManifest.xml中声明保护广播。 如果是系统应用，则可以在系统应用的AndroidManifest.xml里有声明为保护广播。

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="android" coreApp="true" android:sharedUserId="android.uid.system"

android:sharedUserLabel="@string/android\_system\_label">

<!-- ================================================ -->

<!-- Special broadcasts that only the system can send -->

<!-- ================================================ -->

<eat-comment />

<protected-broadcast android:name="android.net.tether.CONNECTEDSTA\_CHANGE" />

<protected-broadcast android:name="android.intent.action.SCREEN\_OFF" />

<protected-broadcast android:name="android.intent.action.SCREEN\_ON" />

...

<protected-broadcast android:name="android.intent.action.PRE\_BOOT\_COMPLETED" />

...

</manifest>

另外一些公共的action，虽然没有保护但是系统允许在系统内部发送；如Intent.ACTION\_CLOSE\_SYSTEM\_DIALOGS，Intent.ACTION\_DISMISS\_KEYBOARD\_SHORTCUTS等；

### 未在Framework中声明保护广播或者非放开的公共广播

B处的大意是经常有系统组件想发送内部广播给自己，如果必须明确列出每个动作作为受保护的广播是很烦人的，因此在这里进行安全检查，如果符合安全条件及可以正常发送；如果不满足安全条件，则就会走到C处，打印wtf log。wtf的意思是What a Terrible Failure: Report an exception that should never happen.

虽然检查不安全但是系统还是会允许发送该广播，logcat中会出现这样的提醒log：

Sending non-protected broadcast xxxxx from system 7747:com.xxxx.xxxx/1000

pkg com.xxxxx.xxxxx

### 未给广播加保护影响

虽然广播正常发送了，不影响广播的作用，但是这样的使用是不安全的，系统组件自定义的广播可能会被恶意软件接收或者发送，导致系统不稳定。

且在这个log打印同时会在系统的dropbox下新生成一个wtf的log文件，发多少条这样的广播，就生成多少个这样的文件。而dropbox中默认最多只有1000个文件，再多了就会冲掉旧的文件。Dropbox是我们用于分析ANR tombstone等问题的重要log来源，因此会严重影响稳定性同事对该类问题的统计和解决。

为了提高系统的安全性且避免这样的log，系统应用组件当在使用自发自收的广播时，要尽可能使用明确的广播，及指定接收的包名或者组件名，且对广播发送和接收加权限保护。同时这也使我们使用广播更加规范。

## 整改举例

如果该广播是an explicit broadcast，且该receiver的android：exported为false，或者ri.activityInfo.permission!=null,及该receiver加了权限保护，系统则认为这个广播时做过保护了的，予以正常发送，便不会打这个log，是个规范使用的广播。

动态广播，系统应用可以使用本地广播进行操作，可以满足检查安全的需求。

静态广播：

a.如果是系统独立应用的广播，可以在应用的AndroidManifest.xml里声明为保护广播就可以了。不过注意验证的时候，需要使用adb push到system/app或者system/priv-app/下再重启安卓验证；使用adb install -r后验证依然会报未保护提醒。

==================

PS:

adb install 和adb push区别在于，adb install会将应用安装在data/app下，而adb push到system/priv-app下即安装在system/priv-app下，adb install和adb push重启后都会有一个扫描解析apk的动作

pkg = pp.parsePackage(tmpPackageFile, parseFlags);

最终调到：

private Package parseBaseApkCommon(){}

1

2

3

在parseBaseApkCommon()函数中：

//frameworks\base\core\java\android\content\pm\PackageParser.java

private Package parseBaseApkCommon(Package pkg, Set<String> acceptedTags, Resources res,

XmlResourceParser parser, int flags, String[] outError) throws XmlPullParserException,

IOException {

while ((type = parser.next()) != XmlPullParser.END\_DOCUMENT

&& (type != XmlPullParser.END\_TAG || parser.getDepth() > outerDepth)) {

if(...){...

} else if (tagName.equals(TAG\_PROTECTED\_BROADCAST)) {

sa = res.obtainAttributes(parser,

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

String name = sa.getNonResourceString( com.android.internal.R.styleable.AndroidManifestProtectedBroadcast\_name);

sa.recycle();

if (name != null && (flags&PARSE\_IS\_SYSTEM) != 0) {//如果不是system的parser flag就跳过解析

if (pkg.protectedBroadcasts == null) {

pkg.protectedBroadcasts = new ArrayList<String>();

}

if (!pkg.protectedBroadcasts.contains(name)) {

pkg.protectedBroadcasts.add(name.intern());

}

}

XmlUtils.skipCurrentTag(parser);

}

}

...

}

1

2

在system/priv-app下扫描apk是是PARSE\_IS\_SYSTEM的 flag(在开机时的PMS的构造函数里就设置了扫描系统应用文件夹的parser flag)，而data/app下没有这个system的parser flag

因此在这里adb push可以生效，adb install未生效。

==========

b. 使用指定包名并且加权限保护

b1.在Androidmanifest.xml里声明receiver的时候加上自定义的权限，如果是仅需应用内接收，可以将android:exported属性设置为false；

<receiver android:name=".DemoReceiver"

android:exported="false"

android:permission="com.android.permission.RECV.XXX">

<intent-filter android:priority="1000">

<action android:name="com.android.demo.test.XXX"/>

</intent-filter>

</receiver>

1

2

3

4

5

6

7

具体加权限请参考 http://blog.csdn.net/javensun/article/details/7334230

b2. 发送广播的地方指定包名或者组件名

Intent i = new Intent("com.android.demo.test.XXX");

i.setPackage("com.XXX.broadcasttest");

sendBroadcast(i,"com.android.permission.RECV.XXX");

1

2

3

注意指给广播加权限是不够的，在checkBroadcastFromSystem对未在framework中声明为保护广播的系统应用自定义广播进行安全检查的前提是这是一个explicit的广播。因此满足上面两步后才能真正消除警告的wtf log.

c. 如果是非独立应用的系统组件，或者是framework下的应用使用的广播，推荐在framework/base/core/res/AndroidManifest.xml中有声明为保护广播。

android exported属性：http://blog.csdn.net/watermusicyes/article/details/46460347

广播权限限制：http://blog.csdn.net/javensun/article/details/7334230

本地广播使用：http://blog.csdn.net/lj2012sy/article/details/51688985

## REF

F

protected-broadcast 系统应用自定义广播规范

<https://blog.csdn.net/TaylorPotter/article/details/70194248>

# android.intent.action.SCREEN\_ON

android.intent.action.SCREEN\_OFF广播为order，即如果其中注册了此广播的任何应用在处理此广播时未返回，则会导致后续broadcast的失败，出现ANR，导致系统无法唤醒。

而导致广播未被及时处理的原因，除了可能是由于对应的Receiver处理函数中一些操作长时间未完成外，也可能是由于整个应用进程被block了，从而没有机会去调用Receiver函数。

为了验证这个问题，可以写一个应用，注册SCREEN\_OFF广播，然后在onPause中加入一个死循环，则会导致系统无法点亮屏幕进入待机！而当将SCREEN\_OFF这个广播不加以注册，则不会引起系统问题（当然应用还是会有问题）。

Notifier.sendGoToSleepBroadcast

mScreenBrightnessBoostIntent.addFlags(  
 Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY | Intent.FLAG\_RECEIVER\_FOREGROUND);

**if** (ActivityManagerNative.isSystemReady()) {  
 mContext.sendOrderedBroadcastAsUser(mScreenOffIntent, UserHandle.ALL, **null**,  
 mGoToSleepBroadcastDone, mHandler, 0, **null**, **null**);

## 分析

线程都在正常的nativePollOnce

一个是系统其他app造成的

一个是 自身app造成

其实不管源头是谁，一定在广播做了耗时操作，开启子线程处理吧

## 解决办法

查询所有app注册了这个事件的，然后子线程处理吧

adb shell dumpsys activity b

Historical broadcasts summary [foreground]:

#2: act=android.intent.action.SCREEN\_OFF flg=0x50200010

0 dispatch +486ms finish

Historical Broadcast foreground #1:

用这个方法排查一下哪些在使用？？？

### priority属性修改大点。

### 查询接收者

查看一下系统那些应用注册了这个广播，先静态的吧

源码注释中有这句话

*\* You <em>cannot</em> receive this through components declared in manifests, only by explicitly registering for it with*

动态的的怎么查看呢

https://www.jianshu.com/p/cfa9ed42e379

# AM

## checkComponentPermission

**对于普通应用将通过PackageManagerService中保持的信息进行判断是否具有相应的权限；**  
base/services/core/java/com/android/server/pm/PackageManagerService.java