Android广播机制分析

Last edited by **caoquanli** 1 month ago

Android Broadcast广播机制分析

广播类型

- 静态注册广播：通过 `<receiver> </receiver>` 的形式在AndroidManifest.xml中注册的广播;
- 动态注册广播：通过context.registerReceiver在程序中显示注册的广播；

静态广播和动态广播的区别：

1.静态广播在进程没有运行的情况下也会接收广播，假如进程没有启动的话，会优先调用AMS中的startProcessLocked中的方法，拉起进程，然后处理广播onReceive函数。动态广播是在程序中通过代码显示注册的，因此必须要在进程已经运行的时候才能收到广播。

2.静态广播处理的时候，每次都会创建一个广播接收器的对象，动态广播一般都是同一个广播接收器对象。

3.静态广播无法接收隐式广播，在Android 8 以上已经不起作用，个别广播除外。至于为什么要做这一变更，还是为了节省电量，提升续航，增强性能，提高用户体验。

4.发送一个无序广播，动态注册的广播要优先于静态的注册的广播，同一个应用内，先注册的接收器先收到广播

广播发送类型：

从发送方式上区分：无序广播和有序广播 ，调用不同的方法发送。
从处理类型上区分：前台广播(10s)和后台广播(60s) 设置flags来区别
从发送者区分：系统广播和自定义广播
此外还有protect broadcast(只允许指定应用可以发送)framework的资源包里面的AndroidManifest.xml列出那些广播是protect的。 ** 注意:** android 8以上已经放弃使用Sticky广播.sticky广播通过Context.sendStickyBroadcast()函数来发送，用此函数发送的广播会一直滞留，当有匹配此广播的广播接收器被注册后，该广播接收器就会收到此条信息.

广播处理机制

当发送串行广播(ordered=true)的情况下：

- 静态注册的广播接收者(receivers)，采用串行处理；
- 动态注册的广播接收者(registeredReceivers)，采用串行处理；

当发送并行广播(ordered=false)的情况下：

- 静态注册的广播接收者(receivers)，依然采用串行处理；
- 动态注册的广播接收者(registeredReceivers)，采用并行处理；

简单来说，静态注册的receivers始终采用串行方式来处理； 动态注册的广播处理方式是串行还是并行方式, 取决于广播的发送方式。

静态注册的广播往往其所在进程还没有创建，而进程创建相对比较耗费系统资源的操作，所以 让静态注册的广播串行化，能防止出现瞬间启动大量进程的喷井效应。

广播ANR

只有串行广播才需要考虑超时，因为接收者是串行处理的，前一个receiver处理慢，会影响后一个receiver；并行广播通过一个循环一次性向所有的receiver分发广播事件，所以不存在彼此影响的问题，则没有广播超时；

串行广播超时情况1：
某个广播总处理时间 > 2* receiver总个数 * mTimeoutPeriod, 其中mTimeoutPeriod，前台队列默认为10s，后台队列默认为60s;

串行广播超时情况2：
某个receiver的执行时间超过mTimeoutPeriod；

注册广播机制分析

静态广播在AndroidManifest.xml中注册即可。

动态注册广播的过程 时序图：

动态注册广播在APP进程的过程

ContextImpl.registerReceiver

(patch：frameworks/base/core/java/android/app/ContextImpl.java)

```
    @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(), 0);
    }
```

从上面的可以看到在ContextImpl类中最后调到了registerReceiverInternal方法。
registerReceiverInternal函数包含了几个参数：
receiver 广播接收器对象.
userId 用户id，默认情况下，车载一般只有一个用户，如果支持多用户的话，可能就要变。所以默认的useid是0。
filter 注册广播的filter.
broadcastPermission 指定要注册的广播的权限. **scheduler** 指定广播接收的所在的线程，也就是onReceive所在的线程,也就是说注册的时候就可以指定好广播处理放在哪个线程，如果receiver中事情太多，可以放在另外一个线程，这样可以避免主线程被卡住.
context 通过getOuterContext()获取到context对象
flags 设置注册广播时的flags

ContextImpl.registerReceiverInternal

```
    private Intent registerReceiverInternal(BroadcastReceiver receiver, int userId,
            IntentFilter filter, String broadcastPermission,
            Handler scheduler, Context context, int flags) {
        IIntentReceiver rd = null;
        if (receiver != null) {
            if (mPackageInfo != null && context != null) {
                if (scheduler == null) {
                //假如没有在注册的时候指定那个线程来处理，则默认的指定主线程来处理。
                    scheduler = mMainThread.getHandler();
                }
                //获取IIntentReceiver的binder代理对象。
                rd = mPackageInfo.getReceiverDispatcher(
                    receiver, context, scheduler,
                    mMainThread.getInstrumentation(), true);
            } else {
                if (scheduler == null) {
                    scheduler = mMainThread.getHandler();
                }
                rd = new LoadedApk.ReceiverDispatcher(
                        receiver, context, scheduler, null, true).getIIntentReceiver();
            }
        }
        try {
            //向ActivityManagerService里面注册广播
            final Intent intent = ActivityManager.getService().registerReceiver(
                    mMainThread.getApplicationThread(), mBasePackageName, rd, filter,
                    broadcastPermission, userId, flags);
            if (intent != null) {
                intent.setExtrasClassLoader(getClassLoader());
                intent.prepareToEnterProcess();
            }
            return intent;
        } catch (RemoteException e) {
            throw e.rethrowFromSystemServer();
        }
    }
```

LoadedApk.getReceiverDispatcher

(patch:frameworks/base/core/java/android/app/LoadedApk.java)

```
    public IIntentReceiver getReceiverDispatcher(BroadcastReceiver r,
            Context context, Handler handler,
            Instrumentation instrumentation, boolean registered) {
        synchronized (mReceivers) {
            LoadedApk.ReceiverDispatcher rd = null;
            ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher> map = null;
            //此处registered=true，则进入该分支
            if (registered) {
               //mReceivers是一个二级map，一级key是context，二级key是BroadcastReceiver，value是R
               //默认是首次注册，因此这里是map获取到的是null。
                map = mReceivers.get(context);//
                if (map != null) {
                    rd = map.get(r);
                }
            }
            if (rd == null) {
                //创建ReceiverDispatcher对象。
                rd = new ReceiverDispatcher(r, context, handler,
                        instrumentation, registered);
                if (registered) {
                    if (map == null) {
                        map = new ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>
                        mReceivers.put(context, map);
                    }
                    map.put(r, rd);
                }
            } else {
               //验证context，handler和原来是否相同
                rd.validate(context, handler);
            }
            rd.mForgotten = false;
            //获取到了IntentReceiver的registerReceiver对象，也就是binder的代理对象。
            return rd.getIIntentReceiver();
        }
    }
```

ReceiverDispatcher类的构造函数

(patch:frameworks/base/core/java/android/app/LoadedApk.java)
ReceiverDispatcher(广播分发者)，这个类是LoadedApk的内部类，创建了InnerReceiver的对象，保存了这个receiver，ActivityThread 的信息，用于在广播派发到本进程的时候执执行，并且这个类的getIIntentReceiver方法会返回一个binder对象。

```
        ReceiverDispatcher(BroadcastReceiver receiver, Context context,
                Handler activityThread, Instrumentation instrumentation,
                boolean registered) {
            if (activityThread == null) {
                throw new NullPointerException("Handler must not be null");
            }
            //创建了InnerReceiver的对象
            mIIntentReceiver = new InnerReceiver(this, !registered);
            mReceiver = receiver;
            mContext = context;
            mActivityThread = activityThread;
            mInstrumentation = instrumentation;
            mRegistered = registered;
            mLocation = new IntentReceiverLeaked(null);
            mLocation.fillInStackTrace();
        }
```

InnerReceiver类的构造函数

(patch:frameworks/base/core/java/android/app/LoadedApk.java)
ReceiverDispatcher(广播分发者)有一个内部类InnerReceiver，该类继承于IIntentReceiver.Stub，因此可以确定这是一个binder的服务端，

```
        final static class InnerReceiver extends IIntentReceiver.Stub {
            final WeakReference<LoadedApk.ReceiverDispatcher> mDispatcher;
            final LoadedApk.ReceiverDispatcher mStrongRef;

            InnerReceiver(LoadedApk.ReceiverDispatcher rd, boolean strong) {
                mDispatcher = new WeakReference<LoadedApk.ReceiverDispatcher>(rd);
                mStrongRef = strong ? rd : null;
            }
```

以上总结：我们首先指定了onReceive方法所在的线程，在ReceiverDispatcher中的构造函数创造了InnerReceiver对象实现了IIntentReceiver接口，并且把receiver当作key和创建的ReceiverDispatcher对象当作value以键值对的形式存入到了map中，然后把context当作key，map当作value存入到了mReceivers中，这个过程实际上就是将广播接收者receiver封装成一个实现了IIntentReceiver接口的Binder对象rd，然后将其放置到LoadedApk对象中的mReceivers中保存起来。最后调用到了getIIntentReceiver()的方法获取到了binder代理的服务端的对象rd。这个过程我们其实是在APP进程中执行的，因此我们可以发现动态注册的广播，随着APP的销毁，也会自动销毁。

广播在system_server进程注册过程

我们继续分析registerReceiverInternal函数中的如何向ActivityManagerService里面注册了广播，也就是如何从APP进程传入到了system_service进程实现了跨进程传输

ActivityManager.getService

```
    public static IActivityManager getService() {
        return IActivityManagerSingleton.get();
    }
    private static final Singleton<IActivityManager> IActivityManagerSingleton =
            new Singleton<IActivityManager>() {
                @Override
                protected IActivityManager create() {
                    //获取到IBinder类型的ActivityManagerService的引用
                    final IBinder b = ServiceManager.getService(Context.ACTIVITY_SERVICE);
                    //aidl的实现
                    final IActivityManager am = IActivityManager.Stub.asInterface(b);
                    return am;
                }
            };
```

我们可以看到IActivityManagerSingleton的数据类型是Singleton，查看Singleton的get方法：

```
public abstract class Singleton<T> {
    private T mInstance;
    protected abstract T create();
    public final T get() {
        synchronized (this) {
            if (mInstance == null) {
                mInstance = create();
            }
            return mInstance;
        }
    }
}
```

Singleton是一个抽象类，我们创建了这个抽象类的对象，实现了create()函数，在这个函数中，ActivityManagerService存储到ServiceManager中进行管理，通过Context.ACTIVITY_SERVICE获取到IBinder类型的ActivityManagerService的引用， 后面就是aidl的实现，IActivityManager.java会在编译后生成，Stub是其内部类并继承binder，调用其asInterface()方法会将IBinder的对象b转换成IActivityManager接口，返回其生成代理对象Stub.Proxy， ActivityManagerService继承IActivityManager.Stub。 因此Proxy与ActivityManagerService通过binder形成了IPC通信机制。

ActivityManagerService.registerReceiver

```
    public Intent registerReceiver(IApplicationThread caller, String callerPackage,
            IIntentReceiver receiver, IntentFilter filter, String permission, int userId,
            int flags) {
        enforceNotIsolatedCaller("registerReceiver");
        ArrayList<Intent> stickyIntents = null;
```

```
ProcessRecord callerApp = null;//注册广播所在的进程信息
final boolean visibleToInstantApps
        = (flags & Context.RECEIVER_VISIBLE_TO_INSTANT_APPS) != 0;
int callingUid; //注册广播的uid
int callingPid;//注册广播的pid
boolean instantApp;//即时app
synchronized(this) {
    if (caller != null) {
        /*从mLruProcesses查询调用者的进程信息，每个进程启动后，该进程有Zygote孵化出来的。并且
        保存在ArrayList<ProcessRecord>类型的mLruProcesses中，每个ProcessRecord中都封装有
        据mLruProcesses去遍历然后查询到和 caller相同的binder，然后返回一个index，根据index
        的进程信息*/
        callerApp = getRecordForAppLocked(caller);
        if (callerApp == null) {
            throw new SecurityException(
                    "Unable to find app for caller " + caller
                    + " (pid=" + Binder.getCallingPid()
                    + ") when registering receiver " + receiver);
        }
        if (callerApp.info.uid != SYSTEM_UID &&
                !callerApp.pkgList.containsKey(callerPackage) &&
                !"android".equals(callerPackage)) {
            throw new SecurityException("Given caller package " + callerPackage
                    + " is not running in process " + callerApp);
        }
        callingUid = callerApp.info.uid;
        callingPid = callerApp.pid;
    } else {
        callerPackage = null;
        callingUid = Binder.getCallingUid();
        callingPid = Binder.getCallingPid();
    }
    //判断caller是否为instant app
    instantApp = isInstantApp(callerApp, callerPackage, callingUid);
    //获取应用用户id
    userId = mUserController.handleIncomingUser(callingPid, callingUid, userId, tru
            ALLOW_FULL_ONLY, "registerReceiver", callerPackage);
    //获取广播注册的filter中的action封装到list中
    Iterator<String> actions = filter.actionsIterator();
    if (actions == null) {
        ArrayList<String> noAction = new ArrayList<String>(1);
        noAction.add(null);
        actions = noAction.iterator();
    }

    /* 收集stick广播，最新的android 8 已经放弃了stick广播。mStickyBroadcasts是一个二级map,
    stickies，然后根据这个action，stickies查询到所有intent。*/
    int[] userIds = { UserHandle.USER_ALL, UserHandle.getUserId(callingUid) };
    while (actions.hasNext()) {
        String action = actions.next();
        for (int id : userIds) {
            ArrayMap<String, ArrayList<Intent>> stickies = mStickyBroadcasts.get(id
            if (stickies != null) {
                ArrayList<Intent> intents = stickies.get(action);
                if (intents != null) {
                    if (stickyIntents == null) {
                        stickyIntents = new ArrayList<Intent>();
                    }
                    //将sticky Intent加入到队列
                    stickyIntents.addAll(intents);
                }
            }
        }
    }
}

ArrayList<Intent> allSticky = null;
// 这里不为null表示本次注册的广播中有sticky广播
if (stickyIntents != null) {
    final ContentResolver resolver = mContext.getContentResolver();
    //查找匹配的sticky广播
    for (int i = 0, N = stickyIntents.size(); i < N; i++) {
        Intent intent = stickyIntents.get(i);
        if (instantApp &&
                (intent.getFlags() & Intent.FLAG_RECEIVER_VISIBLE_TO_INSTANT_APPS)
```

```
                    continue;
                }
                if (filter.match(resolver, intent, true, TAG) >= 0) {
                    if (allSticky == null) {
                        allSticky = new ArrayList<Intent>();
                    }
                    allSticky.add(intent);
                }
            }
        }

        //  直接把最近的一个匹配到的sticky广播返回。
        Intent sticky = allSticky != null ? allSticky.get(0) : null;
        if (DEBUG_BROADCAST) Slog.v(TAG_BROADCAST, "Register receiver " + filter + ": " + s
        if (receiver == null) {
            return sticky;
        }
        //将本次注册的广播放到mRegisteredReceivers中记录
        synchronized (this) {
            //校验caller进程是否正常
            if (callerApp != null && (callerApp.thread == null
                    || callerApp.thread.asBinder() != caller.asBinder())) {
                // Original caller already died
                return null;
            }
            //获取通过该广播接收器receiver的所有已经注册的广播。
            ReceiverList rl = mRegisteredReceivers.get(receiver.asBinder());
            if (rl == null) {
                rl = new ReceiverList(this, callerApp, callingPid, callingUid,
                        userId, receiver);
                if (rl.app != null) {
                   //把ReceiverList添加到processRecoder中, 这样processRecoder就记录了这个进程所有
                    rl.app.receivers.add(rl);
                } else {
                    try {
                        receiver.asBinder().linkToDeath(rl, 0);
                    } catch (RemoteException e) {
                        return sticky;
                    }
                    rl.linkedToDeath = true;
                }
                /* HashMap<IBinder, ReceiverList>类型的mRegisteredReceivers保存了整个系统所有注
                mRegisteredReceivers.put(receiver.asBinder(), rl);
            } else if (rl.uid != callingUid) {
                throw new IllegalArgumentException(
                        "Receiver requested to register for uid " + callingUid
                        + " was previously registered for uid " + rl.uid
                        + " callerPackage is " + callerPackage);
            } else if (rl.pid != callingPid) {
                throw new IllegalArgumentException(
                        "Receiver requested to register for pid " + callingPid
                        + " was previously registered for pid " + rl.pid
                        + " callerPackage is " + callerPackage);
            } else if (rl.userId != userId) {
                throw new IllegalArgumentException(
                        "Receiver requested to register for user " + userId
                        + " was previously registered for user " + rl.userId
                        + " callerPackage is " + callerPackage);
            }
            // 把filter, rl进行封装, 这样每个IntentFilter对应一个BroadcastFilter,
            BroadcastFilter bf = new BroadcastFilter(filter, rl, callerPackage,
                    permission, callingUid, userId, instantApp, visibleToInstantApps);
            //每调用一次register就会add一次。
            rl.add(bf);
            if (!bf.debugCheck()) {
                Slog.w(TAG, "==> For Dynamic broadcast");
            }
            // mReceiverResolver中存放所有的BroadcastFilter, 也可以理解存放了所有IntentFilter。
            mReceiverResolver.addFilter(bf);

            //有匹配的sticky广播, 则直接开始调度派发
            if (allSticky != null) {
                ArrayList receivers = new ArrayList();
                receivers.add(bf);
                // 对于每一个sticky广播, 创建BroadcastRecord并入队 (并行)
                final int stickyCount = allSticky.size();
                for (int i = 0; i < stickyCount; i++) {
                    Intent intent = allSticky.get(i);
                    BroadcastQueue queue = broadcastQueueForIntent(intent);
                    BroadcastRecord r = new BroadcastRecord(queue, intent, null,
                            null, -1, -1, false, null, null, AppOpsManager.OP_NONE, null, r
                            null, 0, null, null, false, true, true, -1);
                     // 入队，并行队列
                    queue.enqueueParallelBroadcastLocked(r);
                    // 启动广播的调度，也就是开始派发广播
                    queue.scheduleBroadcastsLocked();
                }
            }

            return sticky;
        }
    }
```

变量的意义：
ArrayList <BroadcastFilter> 类型的ReceiverList 保存了通过receive注册到的所有广播
HashMap < IBinder, ReceiverList> 类型的RegisteredReceivers 保存了所有注册的广播接收器
IntentResolver<BroadcastFilter, BroadcastFilter> 类型的 mReceiverResolver 保存了整个系统的注册的广播

广播注册总结：
传递了BroadcastReceiver，IntentFilter参数，把BroadcastReceiver进行了封装打包到一个实现了InnerReceiver接口的Binder的代理对象receiver，通过AMS把当前进程的ApplicationThread和InnerReceiver对象的代理对象，注册登记到system_server进程中，并且创建广播接收者队列ReceiverList，把所有通过代理对象receiver的注册的广播都保存到ReceiverList，并把ReceiverList保存到AMS中的所有注册的广播接收器RegisteredReceivers。创建BroadcastFilter，并添加到AMS.mReceiverResolver，将BroadcastFilter添加到该广播接收者的ReceiverList

广播发送机制

广播发送端所在进程:

默认的广播发送为：ContextImp.sendBroadcast

ContextImp.sendBroadcast函数解析:

frameworks/base/core/java/android/app/ContextImpl.java

```
    @Override
    public void sendBroadcast(Intent intent) {
        warnIfCallingFromSystemProcess();
        String resolvedType = intent.resolveTypeIfNeeded(getContentResolver());
        try {
            //离开APP进程，这样APP进程和系统进程的intent的互不干扰，
            intent.prepareToLeaveProcess(this);
            //ActivityManagerservice代理对象通过binder驱动进入system_server进程发送广播。
            ActivityManager.getService().broadcastIntent(
                    mMainThread.getApplicationThread(), intent, resolvedType, null,
                    Activity.RESULT_OK, null, null, null, AppOpsManager.OP_NONE, null, fals
                    getUserId());
        } catch (RemoteException e) {
            throw e.rethrowFromSystemServer();
        }
    }
```

广播发送所在的system_service进程：

ActivityManagerService.broadcastIntent

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```
    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");
        synchronized(this) {
            //获取发送广播的intent是否有效
            intent = verifyBroadcastLocked(intent);
            //获取发送广播的App的进程信息
            final ProcessRecord callerApp = getRecordForAppLocked(caller);
            final int callingPid = Binder.getCallingPid();
            final int callingUid = Binder.getCallingUid();
            final long origId = Binder.clearCallingIdentity();
            //调用AMS.broadcastIntentLocked进行广播发送。
            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;
        }
    }
```

ActivityManagerService.broadcastIntentLocked

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java
对这个方法的参数进行解析：

- callerApp 发送广播的进程信息
- callerPackage 发送广播的包名
- intent 发送广播的intent
- resolvedType 发送广播的类型
- resultTo 最终接受的receiver，默认是null
- resultCode 有序广播在一个receiver处理之后可以设置值，这样下一个receiver就可以拿到这个Bundle，data等信息，，默认是-1
- resultData 有序广播在一个receiver处理之后可以设置值，这样下一个receiver就可以拿到这个数据，默认是null
- resultExtras 有序广播在一个receiver处理之后可以设置值，这样下一个receiver就可以拿到这个bundle，默认是null
- requiredPermissions 接收这个广播需要的权限，默认是null
- appOp 接受这个广播需要的AppopsManager权限
- bOptions 设置在这个广播发送的时候将APP放入到deviceIdle白名单中，有时长限制。
- ordered 是否是有序广播
- sticky 是否是stick广播
- callingPid 发送广播应用的pid
- callingUid 发送广播的应用的Uid
- userId 用户id

```
    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) {
            //Intent支持跨进程传输，创建一个新的intent，这样APP进程和系统进程就不会互相干扰。
        intent = new Intent(intent);
        //检查是否是即时App
        final boolean callerInstantApp = isInstantApp(callerApp, callerPackage, callingUid)
        // 如果是即时APP则不能发送可见的广播，则不能使用FLAG_RECEIVER_VISIBLE_TO_INSTANT_APPS
        if (callerInstantApp) {
            intent.setFlags(intent.getFlags() & ~Intent.FLAG_RECEIVER_VISIBLE_TO_INSTANT_AP
        }

        // 默认不发给停止的应用
        intent.addFlags(Intent.FLAG_EXCLUDE_STOPPED_PACKAGES);

        // 系统如果没有启动完成，则只能发给动态注册的广播
        if (!mProcessesReady && (intent.getFlags()&Intent.FLAG_RECEIVER_BOOT_UPGRADE) == 0)
            intent.addFlags(Intent.FLAG_RECEIVER_REGISTERED_ONLY);
        }

        if (DEBUG_BROADCAST_LIGHT) Slog.v(TAG_BROADCAST,
                (sticky ? "Broadcast sticky: ": "Broadcast: ") + intent
                + " ordered=" + ordered + " userid=" + userId);
        if ((resultTo != null) && !ordered) {
            Slog.w(TAG, "Broadcast " + intent + " not ordered but result callback requested
        }
        //userid校验,
        userId = mUserController.handleIncomingUser(callingPid, callingUid, userId, true,
                ALLOW_NON_FULL, "broadcast", callerPackage);

        //检查用户是否停止运行。
        if (userId != UserHandle.USER_ALL && !mUserController.isUserRunningLocked(userId, 0
            if ((callingUid != SYSTEM_UID
                    || (intent.getFlags() & Intent.FLAG_RECEIVER_BOOT_UPGRADE) == 0)
                    && !Intent.ACTION_SHUTDOWN.equals(intent.getAction())) {
                Slog.w(TAG, "Skipping broadcast of " + intent
                        + ": user " + userId + " is stopped");
                return ActivityManager.BROADCAST_FAILED_USER_STOPPED;
            }
        }
         //bOptions会改变deviceidle的临时白名单，检查caller是否有改变deviceidle名单的权限
        BroadcastOptions brOptions = null;
        if (bOptions != null) {
            brOptions = new BroadcastOptions(bOptions);
            if (brOptions.getTemporaryAppWhitelistDuration() > 0) {
                // See if the caller is allowed to do this.  Note we are checking against
                // the actual real caller (not whoever provided the operation as say a
                // PendingIntent), because that who is actually supplied the arguments.
                if (checkComponentPermission(
                        android.Manifest.permission.CHANGE_DEVICE_IDLE_TEMP_WHITELIST,
                        Binder.getCallingPid(), Binder.getCallingUid(), -1, true)
                        != PackageManager.PERMISSION_GRANTED) {
                    String msg = "Permission Denial: " + intent.getAction()
                            + " broadcast from " + callerPackage + " (pid=" + callingPid
                            + ", uid=" + callingUid + ")"
                            + " requires "
                            + android.Manifest.permission.CHANGE_DEVICE_IDLE_TEMP_WHITELIST
                    Slog.w(TAG, msg);
                    throw new SecurityException(msg);
                }
            }
        }

        //获取action，检查广播是否是受系统保护的广播，如果是系统保护的广播，则只能有系统的
        //root,system,phone,Bluetooth,nfc 的uid，以及persist应用可以发送.
        final String action = intent.getAction();
        final boolean isProtectedBroadcast;
        try {
            isProtectedBroadcast = AppGlobals.getPackageManager().isProtectedBroadcast(acti
        } catch (RemoteException e) {
            Slog.w(TAG, "Remote exception", e);
            return ActivityManager.BROADCAST_SUCCESS;
        }

        final boolean isCallerSystem;
        switch (UserHandle.getAppId(callingUid)) {
            case ROOT_UID:
            case SYSTEM_UID:
            case PHONE_UID:
            case BLUETOOTH_UID:
            case NFC_UID:
                isCallerSystem = true;
                break;
            default:
                isCallerSystem = (callerApp != null) && callerApp.persistent;
                break;
        }

        //进行安全检查：禁止非系统应用程序发送受保护的广播。
        if (!isCallerSystem) {
            if (isProtectedBroadcast) {
                String msg = "Permission Denial: not allowed to send broadcast "
                        + action + " from pid="
                        + callingPid + ", uid=" + callingUid;
                Slog.w(TAG, msg);
                throw new SecurityException(msg);

            } else if (AppWidgetManager.ACTION_APPWIDGET_CONFIGURE.equals(action)
                    || AppWidgetManager.ACTION_APPWIDGET_UPDATE.equals(action)) {
                    //对于这些特殊广播 callerPackage 不允许是空的,
                if (callerPackage == null) {
                    String msg = "Permission Denial: not allowed to send broadcast "
                            + action + " from unknown caller.";
                    Slog.w(TAG, msg);
                    throw new SecurityException(msg);
                } else if (intent.getComponent() != null) {
                    if (!intent.getComponent().getPackageName().equals(
                            callerPackage)) {
                        String msg = "Permission Denial: not allowed to send broadcast "
                                + action + " to "
                                + intent.getComponent().getPackageName() + " from "
                                + callerPackage;
                        Slog.w(TAG, msg);
                        throw new SecurityException(msg);
                    }
                } else {
                    // Limit broadcast to their own package.
                    intent.setPackage(callerPackage);
                }
            }
        }

        //对于一些特殊的广播，判断在后台的应用有没有此广播，如果有，则添加一个FLAG_RECEIVER_INCLUDE_BA
        //这样后台的应用就可以接收到了隐式广播。通常只有当广播指定了显式组件或包名称时，它们才会接收广播。
        if (action != null) {
            if (getBackgroundLaunchBroadcasts().contains(action)) {
                if (DEBUG_BACKGROUND_CHECK) {
                    Slog.i(TAG, "Broadcast action " + action + " forcing include-background
                }
                intent.addFlags(Intent.FLAG_RECEIVER_INCLUDE_BACKGROUND);
            }
            //主要是针对一些package相关的广播处理。
            switch (action) {
                case Intent.ACTION_UID_REMOVED:
                case Intent.ACTION_PACKAGE_REMOVED:
                case Intent.ACTION_PACKAGE_CHANGED:
                case Intent.ACTION_EXTERNAL_APPLICATIONS_UNAVAILABLE:
                case Intent.ACTION_EXTERNAL_APPLICATIONS_AVAILABLE:
                case Intent.ACTION_PACKAGES_SUSPENDED:
                case Intent.ACTION_PACKAGES_UNSUSPENDED:
            .....
            .....
            }
        }

        // 如果是stick广播，就需要对stick广播进行一些处理
        if (sticky) {
            // sticky广播需要在manifest中声明BROADCAST_STICKY权限
            if (checkPermission(android.Manifest.permission.BROADCAST_STICKY,
                    callingPid, callingUid)
                    != PackageManager.PERMISSION_GRANTED) {
                String msg = "Permission Denial: broadcastIntent() requesting a sticky broa
                        + callingPid + ", uid=" + callingUid
                        + " requires " + android.Manifest.permission.BROADCAST_STICKY;
                Slog.w(TAG, msg);
                throw new SecurityException(msg);
            }
            //stick广播不能有权限要求
            if (requiredPermissions != null && requiredPermissions.length > 0) {
                return ActivityManager.BROADCAST_STICKY_CANT_HAVE_PERMISSION;
            }
            //stick广播不允许设置特定component
            if (intent.getComponent() != null) {
                throw new SecurityException(
                        "Sticky broadcasts can't target a specific component");
            }
           //验证非USER_ALL的广播与USER_ALL的广播不冲突。
            if (userId != UserHandle.USER_ALL) {
                ArrayMap<String, ArrayList<Intent>> stickies = mStickyBroadcasts.get(
                        UserHandle.USER_ALL);
                if (stickies != null) {
                    ArrayList<Intent> list = stickies.get(intent.getAction());
                    if (list != null) {
                        int N = list.size();
                        int i;
                        for (i=0; i<N; i++) {
                            if (intent.filterEquals(list.get(i))) {
                                throw new IllegalArgumentException(
                                        "Sticky broadcast " + intent + " for user "
                                        + userId + " conflicts with existing global broadca
                            }
                        }
                    }
                }
            }
            //把stick广播添加到mStickyBroadcasts
            ArrayMap<String, ArrayList<Intent>> stickies = mStickyBroadcasts.get(userId);
            if (stickies == null) {
                stickies = new ArrayMap<>();
                mStickyBroadcasts.put(userId, stickies);
            }
            ArrayList<Intent> list = stickies.get(intent.getAction());
            if (list == null) {
                list = new ArrayList<>();
                stickies.put(intent.getAction(), list);
            }
            final int stickiesCount = list.size();
            int i;
            for (i = 0; i < stickiesCount; i++) {
                if (intent.filterEquals(list.get(i))) {
                    // 找到相同的stick广播，则进行替换。stick广播在注册的时候就已经把最近的一个发出去了
                    list.set(i, new Intent(intent));
                    break;
                }
            }
            if (i >= stickiesCount) {
               //stickiesCount为0的时候，则直接添加。
                list.add(new Intent(intent));
            }
        }//以上是stick广播处理过程

       //广播发送给特定的用户还是全部用户
        int[] users;
        if (userId == UserHandle.USER_ALL) {
            // Caller wants broadcast to go to all started users.
            users = mUserController.getStartedUserArrayLocked();
        } else {
            // Caller wants broadcast to go to one specific user.
            users = new int[] {userId};
        }

        // 查找注册的广播接收器
        List receivers = null; //静态广播接收器
        List<BroadcastFilter> registeredReceivers = null;  //动态广播接收器
        //检查intent有没有设置指定动态接收器，如果没有，将会查找静态广播
        if ((intent.getFlags()&Intent.FLAG_RECEIVER_REGISTERED_ONLY)
                 == 0) {
                 //查找静态广播接收器
            receivers = collectReceiverComponents(intent, resolvedType, callingUid, users);
        }
        if (intent.getComponent() == null) {
            //查找USER_ALL的用户shell的uid的动态广播接收器
            if (userId == UserHandle.USER_ALL && callingUid == SHELL_UID) {
                // Query one target user at a time, excluding shell-restricted users
                for (int i = 0; i < users.length; i++) {
                    if (mUserController.hasUserRestriction(
                            UserManager.DISALLOW_DEBUGGING_FEATURES, users[i])) {
                        continue;
                    }
                    List<BroadcastFilter> registeredReceiversForUser =
                            mReceiverResolver.queryIntent(intent,
                                    resolvedType, false /*defaultOnly*/, users[i]);
                    if (registeredReceivers == null) {
```

```
                    registeredReceivers = registeredReceiversForUser;
                } else if (registeredReceiversForUser != null) {
                    registeredReceivers.addAll(registeredReceiversForUser);
                }
            }
        } else {
           //默认情况下，查找对于当前用户的动态接收器
            registeredReceivers = mReceiverResolver.queryIntent(intent,
                    resolvedType, false /*defaultOnly*/, userId);
        }
    }
    //查看intent是否设置了FLAG_RECEIVER_REPLACE_PENDING，如果设置还没有派发，则替换掉之前的广播。
    final boolean replacePending =
            (intent.getFlags()&Intent.FLAG_RECEIVER_REPLACE_PENDING) != 0;

    //处理并行广播
    int NR = registeredReceivers != null ? registeredReceivers.size() : 0;
    if (!ordered && NR > 0) {
        //对系统的广播进行校验。
        if (isCallerSystem) {
            checkBroadcastFromSystem(intent, callerApp, callerPackage, callingUid,
                    isProtectedBroadcast, registeredReceivers);
        }
        //创建了BroadcastRecord，并入列。
        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) {
            //添加到mParallelBroadcasts中，准备并发
            queue.enqueueParallelBroadcastLocked(r);
            //执行并发
            queue.scheduleBroadcastsLocked();
        }
        registeredReceivers = null;
        NR = 0;
    }

    int ir = 0;
    if (receivers != null) {
        //对于一些特殊的广播进行检查。不满足条件的直接从receivers移除
        String skipPackages[] = null;
        if (Intent.ACTION_PACKAGE_ADDED.equals(intent.getAction())
                || Intent.ACTION_PACKAGE_RESTARTED.equals(intent.getAction())
                || Intent.ACTION_PACKAGE_DATA_CLEARED.equals(intent.getAction())) {
            Uri data = intent.getData();
            if (data != null) {
                String pkgName = data.getSchemeSpecificPart();
                if (pkgName != null) {
                    skipPackages = new String[] { pkgName };
                }
            }
        } else if (Intent.ACTION_EXTERNAL_APPLICATIONS_AVAILABLE.equals(intent.getActio
            skipPackages = intent.getStringArrayExtra(Intent.EXTRA_CHANGED_PACKAGE_LIST
        }
        if (skipPackages != null && (skipPackages.length > 0)) {
            for (String skipPackage : skipPackages) {
                if (skipPackage != null) {
                    int NT = receivers.size();
                    for (int it=0; it<NT; it++) {
                        ResolveInfo curt = (ResolveInfo)receivers.get(it);
                        if (curt.activityInfo.packageName.equals(skipPackage)) {
                            receivers.remove(it);
                            it--;
                            NT--;
                        }
                    }
                }
            }
        }
        //处理串行广播，动态广播和静态广播会根据priority的值进行排序，并且全部合并到receivers，
        int NT = receivers != null ? receivers.size() : 0;
        int it = 0;
        ResolveInfo curt = null;
        BroadcastFilter curr = null;
        while (it < NT && ir < NR) {
            if (curt == null) {
                curt = (ResolveInfo)receivers.get(it);
            }
            if (curr == null) {
                curr = registeredReceivers.get(ir);
            }
            if (curr.getPriority() >= curt.priority) {
                // Insert this broadcast record into the final list.
                receivers.add(it, curr);
                ir++;
                curr = null;
                it++;
                NT++;
            } else {
                // Skip to the next ResolveInfo in the final list.
                it++;
                curt = null;
            }
        }
    }
    //把剩余的没有添加到receivers的动态广播添加进去，放在最后面
    while (ir < NR) {
        if (receivers == null) {
            receivers = new ArrayList();
        }
        receivers.add(registeredReceivers.get(ir));
        ir++;
    }
    //检查发送的广播是否是ProtectedBroadcast。
    if (isCallerSystem) {
        checkBroadcastFromSystem(intent, callerApp, callerPackage, callingUid,
                isProtectedBroadcast, receivers);
    }
    //假如receivers不为null，开始入队列。先创建一个BroadcastRecord，把BroadcastRecord加入到队列
    if ((receivers != null && receivers.size() > 0)
            || resultTo != null) {
        //根据flag监测获取是前台还是后台广播的queue
        BroadcastQueue queue = broadcastQueueForIntent(intent);
        BroadcastRecord r = new BroadcastRecord(queue, intent, callerApp,
                callerPackage, callingPid, callingUid, callerInstantApp, resolvedType,
                requiredPermissions, appOp, brOptions, receivers, resultTo, resultCode,
                resultData, resultExtras, ordered, sticky, false, userId);
        final BroadcastRecord oldRecord =
                replacePending ? queue.replaceOrderedBroadcastLocked(r) : null;
        if (oldRecord != null) {
            // Replaced, fire the result-to receiver.
            if (oldRecord.resultTo != null) {
                final BroadcastQueue oldQueue = broadcastQueueForIntent(oldRecord.inten
                try {
                    oldQueue.performReceiveLocked(oldRecord.callerApp, oldRecord.result
                            oldRecord.intent,
                            Activity.RESULT_CANCELED, null, null,
                            false, false, oldRecord.userId);
                } catch (RemoteException e) {
                }
            }
        } else {
            //如果不替换，加入到有序队列mOrderedBroadcasts中，入列
            queue.enqueueOrderedBroadcastLocked(r);
            queue.scheduleBroadcastsLocked();
        }
    } else {
        // 如果没有查找到对应的广播，那么只做一个记录。
        if (intent.getComponent() == null && intent.getPackage() == null
                && (intent.getFlags()&Intent.FLAG_RECEIVER_REGISTERED_ONLY) == 0) {
            // This was an implicit broadcast... let's record it for posterity.
            addBroadcastStatLocked(intent.getAction(), callerPackage, 0, 0, 0);
        }
    }
```

```
        return ActivityManager.BROADCAST_SUCCESS;
    }
```

可以发现broadcastIntentLocked函数主要做了,设置广播的flag,各种权限校验，protect-Broadcast的广播校验，特殊action的处理。stick广播的处理，查询匹配到的静态广播和动态广播，动态注册广播的并行广播入列，静态广播和动态排序合并用于有序广播入列。 从上面我们可以发现不管发送的是并行广播还是有序广播，静态广播接收器都是有序处理。并且在发送串行的广播过程中，动态注册的优先于静态注册的接收器。

广播处理过程

在发送广播过程中会执行scheduleBroadcastsLocked方法来处理相关的广播。

BroadcastQueue.scheduleBroadcastsLocked

(frameworks/base/services/core/java/com/android/server/am/BroadcastQueue.java)

```
    public void scheduleBroadcastsLocked() {
        if (DEBUG_BROADCAST) Slog.v(TAG_BROADCAST, "Schedule broadcasts ["
                + mQueueName + "]: current="
                + mBroadcastsScheduled);

        if (mBroadcastsScheduled) {
            return;
        }
        mHandler.sendMessage(mHandler.obtainMessage(BROADCAST_INTENT_MSG, this));
        mBroadcastsScheduled = true;
    }
```

在BroadcastQueue对象创建时，BroadcastQueue的构造函数创建了mHandler=new BroadcastHandler(handler.getLooper());那么此处交由mHandler的handleMessage来处理：

BroadcastHandler解析

```
public ActivityManagerService(Context systemContext) {
       //指定了一个名为activitymanager线程来处理广播
        mHandlerThread = new ServiceThread(TAG,
                THREAD_PRIORITY_FOREGROUND, false /*allowIo*/);
        mHandlerThread.start();
        mHandler = new MainHandler(mHandlerThread.getLooper());
                ......
                ......
        //创建前台队列和后台队列的BroadcastQueue对象,
        mFgBroadcastQueue = new BroadcastQueue(this, mHandler,
                "foreground", BROADCAST_FG_TIMEOUT, false);
        mBgBroadcastQueue = new BroadcastQueue(this, mHandler,
                "background", BROADCAST_BG_TIMEOUT, true);
                ......
}
    BroadcastQueue(ActivityManagerService service, Handler handler,
            String name, long timeoutPeriod, boolean allowDelayBehindServices) {
        mService = service;
        //创建BroadcastHandler,
        mHandler = new BroadcastHandler(handler.getLooper());
        mQueueName = name;
        mTimeoutPeriod = timeoutPeriod;
        mDelayBehindServices = allowDelayBehindServices;
    }
```

由此我们可以发现广播的处理过程在system进程中的一个名为activitymanager线程来处理广播。

BroadcastQueue.processNextBroadcast

```
    final void processNextBroadcast(boolean fromMsg) {
        synchronized(mService) {
            BroadcastRecord r;
            //更新CPU状态，提起为ANR做准备
            mService.updateCpuStats();
            if (fromMsg) {
                mBroadcastsScheduled = false;
            }
            //part1: 处理并发广播
            while (mParallelBroadcasts.size() > 0) {
                r = mParallelBroadcasts.remove(0);
                r.dispatchTime = SystemClock.uptimeMillis();
                r.dispatchClockTime = System.currentTimeMillis();
                final int N = r.receivers.size();
                for (int i=0; i<N; i++) {
                    Object target = r.receivers.get(i);
                    //分发广播给已注册的广播接受者
                    deliverToRegisteredReceiverLocked(r, (BroadcastFilter)target, false, i)
                }
                //添加广播到历史统计中
                addBroadcastToHistoryLocked(r);
            }

             //part2: 处理有序广播
             //在我们发送有序广播队列之前，需要先检查我们等待的进程是否仍然存在。
            if (mPendingBroadcast != null) {
                boolean isDead;
                synchronized (mService.mPidsSelfLocked) {
                   //从mPidsSelfLocked获取正在处理该广播进程，判断该进程是否死亡
                    ProcessRecord proc = mService.mPidsSelfLocked.get(mPendingBroadcast.cur
                    isDead = proc == null || proc.crashing;
                }
                if (!isDead) {
                    //正在处理广播的进程保持活跃状态，则继续等待其执行完成
                    return;
                } else {
                    //如果进程没有存在。在进程启动之后，会在attachApplication中会进行处理此广播。
                    mPendingBroadcast.state = BroadcastRecord.IDLE;
                    mPendingBroadcast.nextReceiver = mPendingBroadcastRecvIndex;
                    mPendingBroadcast = null;
                }
            }
            boolean looped = false;
            do {
                if (mOrderedBroadcasts.size() == 0) {
                    //所有串行广播处理完成，则调度执行gc
                    mService.scheduleAppGcsLocked();
                    if (looped) {
                        // If we had finished the last ordered broadcast, then
                        // make sure all processes have correct oom and sched
                        // adjustments.
                        mService.updateOomAdjLocked();
                    }
                    return;
                }
                //获取第一个广播队列
                r = mOrderedBroadcasts.get(0);
                boolean forceReceive = false;
                //获取广播接收者的的数量
                //一个有序广播，所有的receiver的派发时间加起来
                // 不能大于2 * mTimeoutPeriod * numReceivers
                // mTimeoutPeriod是本广播队列的超时，前台广播队列10s，后台广播队列60s
                int numReceivers = (r.receivers != null) ? r.receivers.size() : 0;
                if (mService.mProcessesReady && r.dispatchTime > 0) {
                    long now = SystemClock.uptimeMillis();
                    if ((numReceivers > 0) &&
                            (now > r.dispatchTime + (2*mTimeoutPeriod*numReceivers))) {
                        //当广播处理时间超时，则强制结束这条广播
                        broadcastTimeoutLocked(false);
                        forceReceive = true;
                        r.state = BroadcastRecord.IDLE;
                    }
                }

                if (r.state != BroadcastRecord.IDLE) {
                    return;
                }
                // 这个广播没有更多的receiver了，或者被打断了
                // 或者被上面因为超时强制停止了
                // 此时需要直接执行resultReceiver（发送有序广播时指定的最后一个receiver）
                if (r.receivers == null || r.nextReceiver >= numReceivers
                        || r.resultAbort || forceReceive) {
                    if (r.resultTo != null) {
```

```
                try {
                    //这次广播没有更多的接收器了!如果需要，发送最终结果……
                    performReceiveLocked(r.callerApp, r.resultTo,
                        new Intent(r.intent), r.resultCode,
                        r.resultData, r.resultExtras, false, false, r.userId);
                    r.resultTo = null;
                } catch (RemoteException e) {
                    r.resultTo = null;
                }
            }
            //取消超时
            cancelBroadcastTimeoutLocked();
            // 添加到广播历史记录中，用于记录
            addBroadcastToHistoryLocked(r);
            if (r.intent.getComponent() == null && r.intent.getPackage() == null
                    && (r.intent.getFlags()&Intent.FLAG_RECEIVER_REGISTERED_ONLY) =
                mService.addBroadcastStatLocked(r.intent.getAction(), r.callerPacka
                        r.manifestCount, r.manifestSkipCount, r.finishTime-r.dispat
            }
            // 从有序广播队列中移除
            mOrderedBroadcasts.remove(0);
            r = null;
            looped = true;
            continue;
        }
    } while (r == null);
    // 经过了上面的do-while循环，取出来的就是马上要派发的广播了
    //一个广播会有多个receiver，这个是用来记录当前派发的数量的
    int recIdx = r.nextReceiver++;
    r.receiverTime = SystemClock.uptimeMillis();
    // 这个广播第一次派发，记录时间
    if (recIdx == 0) {
        r.dispatchTime = r.receiverTime;
        r.dispatchClockTime = System.currentTimeMillis();

    }
    if (! mPendingBroadcastTimeoutMessage) {
        long timeoutTime = r.receiverTime + mTimeoutPeriod;
        setBroadcastTimeoutLocked(timeoutTime);
    }

    final BroadcastOptions brOptions = r.options;
    final Object nextReceiver = r.receivers.get(recIdx);
    //处理动态广播接收者
    if (nextReceiver instanceof BroadcastFilter) {
        BroadcastFilter filter = (BroadcastFilter)nextReceiver;
         //进行动态广播派发
        deliverToRegisteredReceiverLocked(r, filter, r.ordered, recIdx);
        if (r.receiver == null || !r.ordered) {
            // The receiver has already finished, so schedule to
            // process the next one.
            if (DEBUG_BROADCAST) Slog.v(TAG_BROADCAST, "Quick finishing ["
                    + mQueueName + "]: ordered="
                    + r.ordered + " receiver=" + r.receiver);
            r.state = BroadcastRecord.IDLE;
            scheduleBroadcastsLocked();
        } else {
            //需要设置deviceidle白名单就在此处设置
            if (brOptions != null && brOptions.getTemporaryAppWhitelistDuration() >
                scheduleTempWhitelistLocked(filter.owningUid,
                        brOptions.getTemporaryAppWhitelistDuration(), r);
            }
        }
        return;
    }
  // 代码走到这里就说明这个receiver是静态注册的
    ResolveInfo info =
        (ResolveInfo)nextReceiver;
    ComponentName component = new ComponentName(
            info.activityInfo.applicationInfo.packageName,
            info.activityInfo.name);

    boolean skip = false;

    ...
```

```
//这中间是一些权限的校验，此处不再罗列

String targetProcess = info.activityInfo.processName;
ProcessRecord app = mService.getProcessRecordLocked(targetProcess,
        info.activityInfo.applicationInfo.uid, false);
// 这里也是权限的校验，但是这里比较重要，因为Android新版本的限制越来越多在此处判断
// getAppStartModeLocked中会根据idle白名单，tmp名单，targetSdkVersion，appops权限等-
//针对 Android O 的应用无法继续在其AndroidManifest为隐式广播注册广播接收器的限制
if (!skip) {
    final int allowed = mService.getAppStartModeLocked(
            info.activityInfo.applicationInfo.uid, info.activityInfo.packageNam
            info.activityInfo.applicationInfo.targetSdkVersion, -1, true, false
    if (allowed != ActivityManager.APP_START_MODE_NORMAL) {
        if (allowed == ActivityManager.APP_START_MODE_DISABLED) {

            skip = true;
            //在Android O上对后台app不允许接收广播的管控
        } else if (((r.intent.getFlags()&Intent.FLAG_RECEIVER_EXCLUDE_BACKGROUN
                || (r.intent.getComponent() == null
                    && r.intent.getPackage() == null
                    && ((r.intent.getFlags()
                            & Intent.FLAG_RECEIVER_INCLUDE_BACKGROUND) == 0)
                    && !isSignaturePerm(r.requiredPermissions))) {
            mService.addBackgroundCheckViolationLocked(r.intent.getAction(),
                    component.getPackageName());
            skip = true;
        }
    }
}

if (skip) {
// 经过前面的校验，如果skip，则进行调度下一个
    r.delivery[recIdx] = BroadcastRecord.DELIVERY_SKIPPED;
    r.receiver = null;
    r.curFilter = null;
    r.state = BroadcastRecord.IDLE;
    scheduleBroadcastsLocked();
    return;
}

r.delivery[recIdx] = BroadcastRecord.DELIVERY_DELIVERED;
r.state = BroadcastRecord.APP_RECEIVE;
r.curComponent = component;
r.curReceiver = info.activityInfo;
if (brOptions != null && brOptions.getTemporaryAppWhitelistDuration() > 0) {
    scheduleTempWhitelistLocked(receiverUid,
            brOptions.getTemporaryAppWhitelistDuration(), r);
}

// 因为广播在派发，设置package不能被stop
try {
    AppGlobals.getPackageManager().setPackageStoppedState(
            r.curComponent.getPackageName(), false, UserHandle.getUserId(r.call
} catch (RemoteException e) {
} catch (IllegalArgumentException e) {
    Slog.w(TAG, "Failed trying to unstop package "
            + r.curComponent.getPackageName() + ": " + e);
}

// 此处判断这个receiver的进程是否存在，如果存在，那么在processCurBroadcastLocked派发
if (app != null && app.thread != null && !app.killed) {
    try {
        app.addPackage(info.activityInfo.packageName,
                info.activityInfo.applicationInfo.versionCode, mService.mProces
        processCurBroadcastLocked(r, app);
        return;
    } catch (RemoteException e) {
        Slog.w(TAG, "Exception when sending broadcast to "
              + r.curComponent, e);
    } catch (RuntimeException e) {
        Slog.wtf(TAG, "Failed sending broadcast to "
                + r.curComponent + " with " + r.intent, e);
        logBroadcastReceiverDiscardLocked(r);
        finishReceiverLocked(r, r.resultCode, r.resultData,
                r.resultExtras, r.resultAbort, false);
                scheduleBroadcastsLocked();
                // We need to reset the state if we failed to start the receiver.
                r.state = BroadcastRecord.IDLE;
                return;
            }

            // If a dead object exception was thrown -- fall through to
            // restart the application.
        }

        // 如果进程不在，启动新的进程，。
        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, fa
                        == null) {
            logBroadcastReceiverDiscardLocked(r);
            //创建失败，则结束该receiver
            finishReceiverLocked(r, r.resultCode, r.resultData,
                    r.resultExtras, r.resultAbort, false);
            scheduleBroadcastsLocked();
            r.state = BroadcastRecord.IDLE;
            return;
        }

        // 记录需要处理的广播，在进程启动之后，进程attach到system server的时候处理
        mPendingBroadcast = r;
        mPendingBroadcastRecvIndex = recIdx;
    }
}
```

- 如果是动态广播接收者，则调用deliverToRegisteredReceiverLocked 处理；
- 如果是静态广播接收者，且对应进程已经创建，则调用processCurBroadcastLocked处理；
- 如果是静态广播接收者，且对应进程尚未创建，则调用startProcessLocked创建进程。

为什么静态注册要跟动态注册有区别呢?
重点在于动态注册的时候可以指定处理receiver的handler，而静态注册无法指定，因此只能用主线程处理。

BroadcastQueue.deliverToRegisteredReceiverLocked

```
private void deliverToRegisteredReceiverLocked(BroadcastRecord r,
        BroadcastFilter filter, boolean ordered, int index) {
    ...
    //权限校验，不做过多罗列
    if (mService.mPermissionReviewRequired) {
        if (!requestStartTargetPermissionsReviewIfNeededLocked(r, filter.packageName,
                filter.owningUserId)) {
            r.delivery[index] = BroadcastRecord.DELIVERY_SKIPPED;
            return;
        }
    }

    r.delivery[index] = BroadcastRecord.DELIVERY_DELIVERED;

    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) {
            r.curApp = filter.receiverList.app;
            filter.receiverList.app.curReceivers.add(r);
            mService.updateOomAdjLocked(r.curApp, true);
        }
    }
    try {
        //广播接收器的应用在备份中，并且是有序广播，不再派发直接跳过。
        if (filter.receiverList.app != null && filter.receiverList.app.inFullBackup) {
            if (ordered) {
                skipReceiverLocked(r);
            }
        } else {
            // 执行receiver的派发
            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) {
        if (ordered) {
            r.receiver = null;
            r.curFilter = null;
            filter.receiverList.curBroadcast = null;
            if (filter.receiverList.app != null) {
                filter.receiverList.app.curReceivers.remove(r);
            }
        }
    }
```

BroadcastQueue.performReceiveLocked

```
    void performReceiveLocked(ProcessRecord app, IIntentReceiver receiver,
            Intent intent, int resultCode, String data, Bundle extras,
            boolean ordered, boolean sticky, int sendingUser) throws RemoteException {
        if (app != null) {
            if (app.thread != null) {
                try {
                   // 此处经过binder call到了receiver所在进程
                    app.thread.scheduleRegisteredReceiver(receiver, intent, resultCode,
                            data, extras, ordered, sticky, sendingUser, app.repProcState);
                } catch (RemoteException ex) {
                    synchronized (mService) {
                        Slog.w(TAG, "Can't deliver broadcast to " + app.processName
                                + " (pid " + app.pid + "). Crashing it.");
                        app.scheduleCrash("can't deliver broadcast");
                    }
                    throw ex;
                }
            } else {
                throw new RemoteException("app.thread must not be null");
            }
        } else {
            //调用者进程为空，则执行该分支
            receiver.performReceive(intent, resultCode, data, extras, ordered,
                    sticky, sendingUser);
        }
    }
```

~~~
这个方法终于到了app进程。

#### BroadcastQueue.processCurBroadcastLocked
``` java
private final void processCurBroadcastLocked(BroadcastRecord r,
            ProcessRecord app) throws RemoteException {
        // app进程必须存在才可以派发
        if (app.thread == null) {
            throw new RemoteException();
        }
        if (app.inFullBackup) {
            skipReceiverLocked(r);
            return;
        }
        // 要派发之前，调整这个进程的优先级
        // 所以app在receiver执行的过程中优先级是很高的
        r.receiver = app.thread.asBinder();
        r.curApp = app;
        app.curReceivers.add(r);
        app.forceProcessStateUpTo(ActivityManager.PROCESS_STATE_RECEIVER);
        mService.updateLruProcessLocked(app, false, null);
        mService.updateOomAdjLocked();
        r.intent.setComponent(r.curComponent);
~~~

```
        boolean started = false;
        try {
            mService.notifyPackageUse(r.intent.getComponent().getPackageName(),
                                       PackageManager.NOTIFY_PACKAGE_USE_BROADCAST_RECEIVER)
            // binder call到app进程
            app.thread.scheduleReceiver(new Intent(r.intent), r.curReceiver,
                    mService.compatibilityInfoForPackageLocked(r.curReceiver.applicationInf
                    r.resultCode, r.resultData, r.resultExtras, r.ordered, r.userId,
                    app.repProcState);
            started = true;
        } finally {
            if (!started) {
                r.receiver = null;
                r.curApp = null;
                app.curReceivers.remove(r);
            }
        }
    }
```

上面这个方法是针对静态注册的广播，调度到其进程执行receiver。

ApplicationThread.scheduleRegisteredReceiver

从这里开始进入APP进程

```
public void scheduleRegisteredReceiver(IIntentReceiver receiver, Intent intent,
     int resultCode, String dataStr, Bundle extras, boolean ordered,
      boolean sticky, int sendingUser, int processState) throws RemoteException {
      ////更新虚拟机进程状态
      updateProcessState(processState, false);
      //执行广播
      receiver.performReceive(intent, resultCode, dataStr, extras, ordered,
                    sticky, sendingUser);
      }
```

终于到了app的进程 从前面的处理流程可以知道，动态注册的receiver无论是并行还是串行最终调度都会到这里。 在注册的时候可以指定receiver的处理handler，如果没有指定才会用主线程。 这里就用到了之前注册时携带的binder对象innerReceiver.

InnerReceiver.performReceive

```
public void performReceive(Intent intent, int resultCode, String data,
    Bundle extras, boolean ordered, boolean sticky, int sendingUser) {
    final LoadedApk.ReceiverDispatcher rd;
    if (intent == null) {
        rd = null;
    } else {
        // innerReceiver是ReceiverDispatcher的内部类
        // 在注册广播的时候innerReceiver保存了对dispatcher的弱饮用，在创建的时候就记录了receiverDiso
        rd = mDispatcher.get();
    }
    if (ActivityThread.DEBUG_BROADCAST) {
        int seq = intent.getIntExtra("seq", -1);
    }
    if (rd != null) {
        //执行广播
        rd.performReceive(intent, resultCode, data, extras,
                ordered, sticky, sendingUser);
    } else {
        IActivityManager mgr = ActivityManager.getService();
        try {
            if (extras != null) {
                extras.setAllowFds(false);
            }
            mgr.finishReceiver(this, resultCode, data, extras, false, intent.getFlags());
        } catch (RemoteException e) {
            throw e.rethrowFromSystemServer();
        }
    }
}
```

ReceiverDispatcher.performReceive

```
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.getRunnable())) {
        if (mRegistered && ordered) {
            IActivityManager mgr = ActivityManager.getService();
            if (ActivityThread.DEBUG_BROADCAST) Slog.i(ActivityThread.TAG,
                    "Finishing sync broadcast to " + mReceiver);
            args.sendFinished(mgr);
        }
    }
}
```

这个方法的重点在于mActivityThread.post(args.getRunnable()) 不要被mActivityThread的名字迷惑，其实这是一个Handler，正是在广播动态注册的时候设置的那个handler 通过这个post，切换了线程，开始执行receiver。 //广播接收端所在进程

Args.run

此方法是在我们注册广播的时候指定的handler的线程里面执行的。

```
public final Runnable getRunnable() {
    return () -> {
        final BroadcastReceiver receiver = mReceiver;
        final boolean ordered = mOrdered;

        final IActivityManager mgr = ActivityManager.getService();
        final Intent intent = mCurIntent;
        if (intent == null) {
        }

        mCurIntent = null;
        mDispatched = true;
        mPreviousRunStacktrace = new Throwable("Previous stacktrace");
        if (receiver == null || intent == null || mForgotten) {
            if (mRegistered && ordered) {
                sendFinished(mgr);
            }
            return;
        }

        try {
            //获取mReceiver的类加载器
            ClassLoader cl = mReceiver.getClass().getClassLoader();
            intent.setExtrasClassLoader(cl);
            intent.prepareToEnterProcess();
            setExtrasClassLoader(cl);
            receiver.setPendingResult(this);
            //回调广播onReceive方法
            receiver.onReceive(mContext, intent);
        } catch (Exception e) {
            if (mRegistered && ordered) {
                sendFinished(mgr);
            }
            if (mInstrumentation == null ||
                    !mInstrumentation.onException(mReceiver, e)) {
                Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
                throw new RuntimeException(
                        "Error receiving broadcast " + intent
                                + " in " + mReceiver, e);
            }
        }
        if (receiver.getPendingResult() != null) {
            finish();
        }
    };
}
```

到这里，终于看到了BroadcastReceiver的回调。 不过这只是动态注册的receiver，而且在动态注册的receiver执行完毕之后，如果是有序广播则还需要发送finish。 在处理有序广播的时候，前一个receiver处理完之后，可以留下一些信息，后面一个receiver在处理的时候可以根据这些信息做进一步的操作，而这些信息的传递正是通过getPendingResult

PendingResult.finish

```
        public final void finish() {
            if (mType == TYPE_COMPONENT) {  //TYPE_COMPONENT代表的是静态的广播
                final IActivityManager mgr = ActivityManager.getService();
                if (QueuedWork.hasPendingWork()) {
                    QueuedWork.queue(new Runnable() {
                        @Override public void run() {
                            sendFinished(mgr);
                        }
                    }, false);
                } else {
                    if (ActivityThread.DEBUG_BROADCAST) Slog.i(ActivityThread.TAG,
                            "Finishing broadcast to component " + mToken);
                    sendFinished(mgr);
                }
            } else if (mOrderedHint && mType != TYPE_UNREGISTERED) { //动态注册的广播接收器
                if (ActivityThread.DEBUG_BROADCAST) Slog.i(ActivityThread.TAG,
                        "Finishing broadcast to " + mToken);
                final IActivityManager mgr = ActivityManager.getService();
                sendFinished(mgr);
            }
        }
```

主要功能:

静态注册的广播接收者:

- 当QueuedWork工作未完成, 即SharedPreferences写入磁盘的操作没有完成, 则等待完成再执行sendFinished方法;因此我们可以发现加入这里的写入磁盘操作时间长 也会发生ANR
- 当QueuedWork工作已完成, 则直接调用sendFinished方法;

动态注册的广播接收者:

- 当发送的是串行广播, 则直接调用sendFinished方法.

另外常量参数说明:
TYPE_COMPONENT: 静态注册
TYPE_REGISTERED: 动态注册
TYPE_UNREGISTERED: 取消注册

PendingResult.sendFinished

```
public void sendFinished(IActivityManager am) {
    synchronized (this) {
        if (mFinished) {
            throw new IllegalStateException("Broadcast already finished");
        }
        mFinished = true;
        try {
            if (mResultExtras != null) {
                mResultExtras.setAllowFds(false);
            }
            //mOrderedHint代表的是串行广播
            if (mOrderedHint) {
                am.finishReceiver(mToken, mResultCode, mResultData, mResultExtras,
                        mAbortBroadcast, mFlags);
            } else {
                //如果发送的是并发广播，但是静态广播是有序来发送的，因此也要告诉AMS
                am.finishReceiver(mToken, 0, null, null, false, mFlags);
            }
        } catch (RemoteException ex) {
        }
    }
}
```

此处AMP.finishReceiver，经过binder调用，进入AMS.finishReceiver方法然后处理下一条广播。