**广播发送流程：**

依据罗升阳博客例子，结合Android 6 的代码分析：

<http://blog.csdn.net/luoshengyang/article/details/6730748>

广播发送的两种方式：有序发送、无序发送

在CounterService.java中的startCounter:

1. **public** **void** startCounter(**int** initVal) {
2. AsyncTask<Integer, Integer, Integer> task = **new** AsyncTask<Integer, Integer, Integer>() {
3. ......
4. @Override
5. **protected** **void** onProgressUpdate(Integer... values) {
6. **super**.onProgressUpdate(values);
7. **int** counter = values[0];
8. Intent intent = **new** Intent(BROADCAST\_COUNTER\_ACTION);
9. intent.putExtra(COUNTER\_VALUE, counter);
10. sendBroadcast(intent);
11. }
13. @Override
14. **protected** **void** onPostExecute(Integer val) {
15. **int** counter = val;
16. Intent intent = **new** Intent(BROADCAST\_COUNTER\_ACTION);
17. intent.putExtra(COUNTER\_VALUE, counter);
18. sendBroadcast(intent);
19. }
20. };
21. task.execute(0);
22. }

使用异步任务AsyncTask来实现计数器，将当前的值写入到BROADCAST\_COUNTER\_ACTION类型的广播中，并且CounterService将这个广播发送给AMS。该广播是存储在intent 中，使用函数sendBroadcast将intent 发送出去。sendBroadcast方法来自父类ContextWrapper：

1. @Override
2. **public** **void** sendBroadcast(Intent intent) {
3. mBase.sendBroadcast(intent);
4. }

成员变量mBase是一个ContextImpl实例，进入ContextImpl的方法sendBroadcast：

1. @Override
2. **public** **void** sendBroadcast(Intent intent) {
3. warnIfCallingFromSystemProcess();
4. String resolvedType = intent.resolveTypeIfNeeded(getContentResolver());
5. **try** {
6. intent.prepareToLeaveProcess();
7. ActivityManagerNative.getDefault().broadcastIntent(
8. mMainThread.getApplicationThread(), intent, resolvedType, **null**,
9. Activity.RESULT\_OK, **null**, **null**, **null**, AppOpsManager.OP\_NONE, **null**, **false**, **false**,
10. getUserId());
11. } **catch** (RemoteException e) {
12. **throw** **new** RuntimeException("Failure from system", e);
13. }
14. }

第4行，resolvedType表示这个Intent的MIME类型，我们没有设置这个Intent的MIME类型，因此，这里的resolvedType为null。第7行，ActivityManagerNative.getDefault()获得一个ActivityManagerService的代理ActivityManagerProxy，第8行我们可以看出mMainThread.getApplicationThread()还给broadcastIntent传入了一个ApplicationThread对象，给AMS说明是谁给它发了一个广播。调用它的broadcastIntent方法:

1. **public** **int** broadcastIntent(IApplicationThread caller,
2. Intent intent, String resolvedType, IIntentReceiver resultTo,
3. **int** resultCode, String resultData, Bundle map,
4. String[] requiredPermissions, **int** appOp, Bundle options, **boolean** serialized,
5. **boolean** sticky, **int** userId) **throws** RemoteException
6. {
7. Parcel data = Parcel.obtain();
8. Parcel reply = Parcel.obtain();
9. data.writeInterfaceToken(IActivityManager.descriptor);
10. data.writeStrongBinder(caller != **null** ? caller.asBinder() : **null**);
11. intent.writeToParcel(data, 0);
12. data.writeString(resolvedType);
13. data.writeStrongBinder(resultTo != **null** ? resultTo.asBinder() : **null**);
14. data.writeInt(resultCode);
15. data.writeString(resultData);
16. data.writeBundle(map);
17. data.writeStringArray(requiredPermissions);
18. data.writeInt(appOp);
19. data.writeBundle(options);
20. data.writeInt(serialized ? 1 : 0);
21. data.writeInt(sticky ? 1 : 0);
22. data.writeInt(userId);
23. mRemote.transact(BROADCAST\_INTENT\_TRANSACTION, data, reply, 0);
24. reply.readException();
25. **int** res = reply.readInt();
26. reply.recycle();
27. data.recycle();
28. **return** res;
29. }

将数据进行封装，将intent通过Binder驱动发送给AMS的broadcastIntent方法，到此为止都是在Broadcounter应用程序所在的进程进行的，接下来将进入AMS所在的进程，也就是system\_server进程。AMS收到BROADCAST\_INTENT\_TRANSACTION类型的请求，通过在ActivityManagerNative的onTransact解析，由于AMS是ActivityManagerNative的子类，在AMS中实现了ActivityManagerNative中定义的方法broadcastIntent。所以由AMS来执行ActivityManagerNative的onTransact中解析BROADCAST\_INTENT\_TRANSACTION类型的请求。看看AMS.java里面的broadcastIntent方法的具体实现:

1. **public** **final** **int** broadcastIntent(IApplicationThread caller,
2. Intent intent, String resolvedType, IIntentReceiver resultTo,
3. **int** resultCode, String resultData, Bundle resultExtras,
4. String[] requiredPermissions, **int** appOp, Bundle options,
5. **boolean** serialized, **boolean** sticky, **int** userId) {
6. enforceNotIsolatedCaller("broadcastIntent");
7. **synchronized**(**this**) {
8. intent = verifyBroadcastLocked(intent);
10. **final** ProcessRecord callerApp = getRecordForAppLocked(caller);
11. **final** **int** callingPid = Binder.getCallingPid();
12. **final** **int** callingUid = Binder.getCallingUid();
13. **final** **long** origId = Binder.clearCallingIdentity();
14. **int** res = broadcastIntentLocked(callerApp,
15. callerApp != **null** ? callerApp.info.packageName : **null**,
16. intent, resolvedType, resultTo, resultCode, resultData, resultExtras,
17. requiredPermissions, appOp, **null**, serialized, sticky,
18. callingPid, callingUid, userId);
19. Binder.restoreCallingIdentity(origId);
20. **return** res;
21. }
22. }

第8行主要来验证intent的内容的合法性，系统启动过程中，可能PMS还没有启动，AMS就无法获得相关APP的信息，静态注册的广播接受者就不能就收广播。代码如下：

1. **final** Intent verifyBroadcastLocked(Intent intent) {
2. // Refuse possible leaked file descriptors
3. **if** (intent != **null** && intent.hasFileDescriptors() == **true**) {
4. **throw** **new** IllegalArgumentException("File descriptors passed in Intent");
5. }
7. **int** flags = intent.getFlags();
9. **if** (!mProcessesReady) {
10. // if the caller really truly claims to know what they're doing, go
11. // ahead and allow the broadcast without launching any receivers
12. **if** ((flags&Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT) != 0) {
13. // This will be turned into a FLAG\_RECEIVER\_REGISTERED\_ONLY later on if needed.
14. } **else** **if** ((flags&Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY) == 0) {
15. Slog.e(TAG, "Attempt to launch receivers of broadcast intent " + intent
16. + " before boot completion");
17. **throw** **new** IllegalStateException("Cannot broadcast before boot completed");
18. }
19. }
21. **if** ((flags&Intent.FLAG\_RECEIVER\_BOOT\_UPGRADE) != 0) {
22. **throw** **new** IllegalArgumentException(
23. "Can't use FLAG\_RECEIVER\_BOOT\_UPGRADE here");
24. }
26. **return** intent;
27. }

回到broadcastIntent代码中，第10-12行是获得发送广播的应用进程的身份。第14-18行调用broadcastIntentLocked来处理参数intent所描述的广播，主要作用是查找目标广播的接收者。broadcastIntentLocked代码很长，分块分析，下面的代码块主要处理发送sticky广播的情况：

1. // Add to the sticky list if requested.
2. **if** (sticky) {
3. ......
4. ArrayMap<String, ArrayList<Intent>> stickies = mStickyBroadcasts.get(userId);
5. **if** (stickies == **null**) {
6. stickies = **new** ArrayMap<>();
7. mStickyBroadcasts.put(userId, stickies);
8. }
9. ArrayList<Intent> list = stickies.get(intent.getAction());
10. **if** (list == **null**) {
11. list = **new** ArrayList<>();
12. stickies.put(intent.getAction(), list);
13. }
14. final int stickiesCount = list.size();
15. **for** (i = 0; i < stickiesCount; i++) {
16. **if** (intent.filterEquals(list.get(i))) {
17. // This sticky already exists, replace it.
18. list.set(i, **new** Intent(intent));
19. **break**;
20. }
21. }
22. **if** (i >= stickiesCount) {
23. list.add(**new** Intent(intent));
24. }
25. }

第2行检测intent是不是一个黏性广播，如果是则要将其保存起来，方便后面注册该广播的广播接收者来获得这个广播。所以在第3-24行的代码就是为了将这个是黏性广播的intent保存起来。在AMS中，具有相同类型的黏性广播只保存在一个列表（一般用stickies 表示）中，该列表使用广播的类型（即intent的action名称）为key，这些列表又保存在AMS中的成员变量mStickyBroadcasts中。

第4-8行的代码，主要是查看该userId的广播，如果存储黏性广播的列表（stickies ）不存在，则会创建一个并且保存到mStickyBroadcasts中。第9-13行，在stickies 查看有没有intent.action所代表的类型的广播，没有的话就创建，这点其实和4.0版本的mStickyBroadcasts是相同的（4.0中mStickyBroadcasts.put(intent.getAction(), list);而在6.0里面stickies.put(intent.getAction(), list);）。第14-21行，在这里的在list中查找和intent的参数一致的广播，存在的话18行替换之，不存在则进入22-24行代码，创建一个存入list。接下来看另外一块这一块的主要工作是查询满足条件的动态广播接收者及静态广播接收者。：

1. // Figure out who all will receive this broadcast.
2. List receivers = **null**;
3. List<BroadcastFilter> registeredReceivers = **null**;
4. // Need to resolve the intent to interested receivers...
5. **if** ((intent.getFlags()&Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY)
6. == 0) {
7. receivers = collectReceiverComponents(intent, resolvedType, callingUid, users);
8. }
9. **if** (intent.getComponent() == **null**) {
10. **if** (userId == UserHandle.USER\_ALL && callingUid == Process.SHELL\_UID) {
11. // Query one target user at a time, excluding shell-restricted users
12. UserManagerService ums = getUserManagerLocked();
13. **for** (**int** i = 0; i < users.length; i++) {
14. **if** (ums.hasUserRestriction(
15. UserManager.DISALLOW\_DEBUGGING\_FEATURES, users[i])) {
16. **continue**;
17. }
18. List<BroadcastFilter> registeredReceiversForUser =
19. mReceiverResolver.queryIntent(intent,
20. resolvedType, **false**, users[i]);
21. **if** (registeredReceivers == **null**) {
22. registeredReceivers = registeredReceiversForUser;
23. } **else** **if** (registeredReceiversForUser != **null**) {
24. registeredReceivers.addAll(registeredReceiversForUser);
25. }
26. }
27. } **else** {
28. registeredReceivers = mReceiverResolver.queryIntent(intent,
29. resolvedType, **false**, userId);
30. }
31. }

第2行的receivers 用来保存静态注册的广播接收者，第3行的registeredReceivers 用来保存动态注册的广播接收者。第5-8行，FLAG\_RECEIVER\_REGISTERED\_ONLY表示该广播只能发给动态注册者，如果没有这种设置，则执行第7行，通过collectReceiverComponents方法获得所有的静态注册该广播的接收者。第9行是没有指定接收者，说明是发送给所有的广播注册者的，第18行查询所有所有的动态注册广播接收者，第24行再把查询结果registeredReceiversForUser添加到registeredReceivers（保存动态注册的广播接收者）中。到此AMS已经获得了所有的目标广播接收者。接下来在不同情况下处理广播，看下一块代码，主要是对无序广播的发送处理：

1. final boolean replacePending =
2. (intent.getFlags()&Intent.FLAG\_RECEIVER\_REPLACE\_PENDING) != 0;
4. **if** (DEBUG\_BROADCAST) Slog.v(TAG\_BROADCAST, "Enqueing broadcast: " + intent.getAction()
5. + " replacePending=" + replacePending);
7. **int** NR = registeredReceivers != null ? registeredReceivers.size() : 0;
8. **if** (!ordered && NR > 0) {
9. // If we are not serializing this broadcast, then send the
10. // registered receivers separately so they don't wait for the
11. // components to be launched.
12. final BroadcastQueue queue = broadcastQueueForIntent(intent);
13. BroadcastRecord r = **new** BroadcastRecord(queue, intent, callerApp,
14. callerPackage, callingPid, callingUid, resolvedType, requiredPermissions,
15. appOp, brOptions, registeredReceivers, resultTo, resultCode, resultData,
16. resultExtras, ordered, sticky, **false**, userId);
17. **if** (DEBUG\_BROADCAST) Slog.v(TAG\_BROADCAST, "Enqueueing parallel broadcast " + r);
18. final boolean replaced = replacePending && queue.replaceParallelBroadcastLocked(r);
19. **if** (!replaced) {
20. queue.enqueueParallelBroadcastLocked(r);
21. queue.scheduleBroadcastsLocked();
22. }
23. registeredReceivers = null;
24. NR = 0;
25. }

Intent.FLAG\_RECEIVER\_REPLACE\_PENDING如果该值为1，表示当该类型的新广播来了，但是旧广播还没有发送出去，那么就要把用新广播内容把旧的广播内容替代掉（不过在这种情况下，有些该类型的广播是接受不到的）。第1-2行，如果FLAG\_RECEIVER\_REPLACE\_PENDING等于1就把replacePending 设置为true。第7行获得动态注册该广播的广播接收者的数量。Ordered（也叫串行化） 描述当前发送的广播是否是一个无序广播，如果是一个无序广播，则AMS将广播发送给接收者就行，不考虑先给哪个接收者发，哪个接收者后发；如果是有序广播，则需要把优先级高的广播接收者先发，优先级低的广播接收者后发。第8行判断如果是不是有序广播（即该广播是一个无序广播），并且有动态注册的广播接收者，那就执行12-24行代码。第13-16行，将intent和该广播的动态目标接收者封装到BroadcastRecord 对象，该对象用来描述一个AMS要执行的广播转发任务。这个广播不是立即执行的，而是添加到AMS的无序广播调度队列queue的mParallelBroadcasts中（第12行定义了）。第18行，在replacePending 为true的情况下，在无序广播调度队列queue中查找是否存在这样的广播，如果有，替换之，并且将replaced 置为true，具体的执行如下：

1. **public** final boolean replaceParallelBroadcastLocked(BroadcastRecord r) {
2. **for** (**int** i = mParallelBroadcasts.size() - 1; i >= 0; i--) {
3. **if** (r.intent.filterEquals(mParallelBroadcasts.get(i).intent)) {
4. **if** (DEBUG\_BROADCAST) Slog.v(TAG\_BROADCAST,
5. "\*\*\*\*\* DROPPING PARALLEL ["
6. + mQueueName + "]: " + r.intent);
7. mParallelBroadcasts.set(i, r);  //非ordered的广播都被AMS保存在mParallelBroadcasts中
8. **return** **true**;  // mParallelBroadcasts中都是动态广播接收者
9. }
10. }
11. **return** **false**;
12. }

回到第20行，如果是没有替换（也就是在queue 中没有查找到），replaced为flase，那么把该BroadcastRecord 对象r添加到queue中。第21行，重新调度这个队列queue的 mParallelBroadcasts。由于到此处无序广播的AMS就已经把广播发送给动态注册的广播接收者了，第23-24行代码是将保存动态接收机的信息重置。接下里看下一块代码：

1. // Merge into one list.
2. **int** ir = 0;
3. **if** (receivers != null) {
4. // A special case for PACKAGE\_ADDED: do not allow the package
5. // being added to see this broadcast.  This prevents them from
6. // using this as a back door to get run as soon as they are
7. // installed.  Maybe in the future we want to have a special install
8. // broadcast or such for apps, but we'd like to deliberately make
9. // this decision.
10. String skipPackages[] = null;
11. **if** (Intent.ACTION\_PACKAGE\_ADDED.equals(intent.getAction())
12. || Intent.ACTION\_PACKAGE\_RESTARTED.equals(intent.getAction())
13. || Intent.ACTION\_PACKAGE\_DATA\_CLEARED.equals(intent.getAction())) {
14. Uri data = intent.getData();
15. **if** (data != null) {
16. String pkgName = data.getSchemeSpecificPart();
17. **if** (pkgName != null) {
18. skipPackages = **new** String[] { pkgName };
19. }
20. }
21. } **else** **if** (Intent.ACTION\_EXTERNAL\_APPLICATIONS\_AVAILABLE.equals(intent.getAction())) {
22. skipPackages = intent.getStringArrayExtra(Intent.EXTRA\_CHANGED\_PACKAGE\_LIST);
23. }
24. **if** (skipPackages != null && (skipPackages.length > 0)) {
25. **for** (String skipPackage : skipPackages) {
26. **if** (skipPackage != null) {
27. **int** NT = receivers.size();
28. **for** (**int** it=0; it<NT; it++) {
29. ResolveInfo curt = (ResolveInfo)receivers.get(it);
30. **if** (curt.activityInfo.packageName.equals(skipPackage)) {
31. receivers.remove(it);
32. it--;
33. NT--;
34. }
35. }
36. }
37. }
38. }
40. **int** NT = receivers != null ? receivers.size() : 0;
41. **int** it = 0;
42. ResolveInfo curt = null;
43. BroadcastFilter curr = null;
44. **while** (it < NT && ir < NR) {
45. **if** (curt == null) {
46. curt = (ResolveInfo)receivers.get(it);
47. }
48. **if** (curr == null) {
49. curr = registeredReceivers.get(ir);
50. }
51. **if** (curr.getPriority() >= curt.priority) {
52. // Insert this broadcast record into the final list.
53. receivers.add(it, curr);
54. ir++;
55. curr = null;
56. it++;
57. NT++;
58. } **else** {
59. // Skip to the next ResolveInfo in the final list.
60. it++;
61. curt = null;
62. }
63. }
64. }
65. **while** (ir < NR) {
66. **if** (receivers == null) {
67. receivers = **new** ArrayList();
68. }
69. receivers.add(registeredReceivers.get(ir));
70. ir++;
71. }

前面已经处理了无序广播的动态注册接收者的广播发送。剩下的就是无序广播的静态注册接收者的广播发送，以及所有有序广播的动态和静态注册的广播接收者的广播发送，而这些广播的处理就是在这块代码执行，所有广播不论是不是有序的，都会保存到AMS的有序广播调度队列中（queue.mOrderedBroadcasts）。第11-20行代码处理的是ACTION\_PACKAGE\_ADDED、ACTION\_PACKAGE\_RESTARTED、ACTION\_PACKAGE\_DATA\_CLEARED广播，当这些广播类型的广播，并且intent的data为空，那么就将接受该广播的Package加入到skipPackages 中。在24-38行代码中将其从receivers去除掉。这是为了防止有些应用监听这些广播，在系统启动的时候跟着启动（和广播的正常发送没有什么关系）。从第40行开始才是本块代码的重点，目的就是将静态和动态注册的广播接收者按照广播的优先级从高到低的都放到静态注册广播接收者的的列表receivers中，(在原来的registeredReceivers 和receivers列表中都是按照优先级由高到底排好序的)

所以在这里40-64行代码只是将两个list进行合并。从代码中可以看出，相同优先级的广播接收者，动态注册的广播会在静态广播之前接收到广播。第65-71行代码，有可能在44行的while循环执行完成之后，registeredReceivers中还有没有添加进去的，由于这些广播接收者的优先级比receivers里面的优先级都低导致的。最后在第65-71行代码中将其加入到receivers中。

到此，静态注册的无序广播，动态注册以及静态注册的有序广播都添加到了receivers列表中，

（疑问：为什么要将静态注册的无序广播保存在receivers中，最后进入queue.mOrderedBroadcasts队列，而不是和动态注册的广播接收者走同样的发送道路，这是由于动态广播接收者已经启动了，而静态注册的广播接收者没有启动，如果走一样的广播方式，就需要AMS循环创建这些进程，如果静态注册的很多那么对系统的压力是非常大的。所以，对于静态注册者，它们对应的广播记录都被放到mOrderedBroadcasts中保存。AMS在处理这类广播信息时，一个进程一个进程地处理，只有处理完一个接收者，才继续下一个接收者。这种做法的好处是，避免了惊群效应的出现，坏处则是延时较长。）

该广播信息的剩余接收者都在receivers中了，接下来AMS就要将这些广播发送出去：

1. **if** ((receivers != null && receivers.size() > 0)
2. || resultTo != null) {
3. BroadcastQueue queue = broadcastQueueForIntent(intent);
4. BroadcastRecord r = **new** BroadcastRecord(queue, intent, callerApp,
5. callerPackage, callingPid, callingUid, resolvedType,
6. requiredPermissions, appOp, brOptions, receivers, resultTo, resultCode,
7. resultData, resultExtras, ordered, sticky, **false**, userId);
9. **if** (DEBUG\_BROADCAST) Slog.v(TAG\_BROADCAST, "Enqueueing ordered broadcast " + r
10. + ": prev had " + queue.mOrderedBroadcasts.size());
11. **if** (DEBUG\_BROADCAST) Slog.i(TAG\_BROADCAST,
12. "Enqueueing broadcast " + r.intent.getAction());
14. boolean replaced = replacePending && queue.replaceOrderedBroadcastLocked(r);
15. **if** (!replaced) {
16. queue.enqueueOrderedBroadcastLocked(r);
17. queue.scheduleBroadcastsLocked();
18. }
19. }
21. **return** ActivityManager.BROADCAST\_SUCCESS;

和无序广播的动态注册接收者的流程类似。第3行，创建队列，第4行将receivers 封装成BroadcastRecord 对象，第14行代码， replacePending如果为true，用新的广播替换queue中OrderedBroadcast的旧广播内容。15-18行，如果没有替换，则把该BroadcastRecord 对象添加到queue的mOrderedBroadcasts队列中，18行重新调度该队列。21行表明对于AMS来说到这广播的发送已经完成了，所以返回了ActivityManager.BROADCAST\_SUCCESS。

在无序广播的动态注册接收者的广播发送和这里的receivers列表的发送都是使用了queue.scheduleBroadcastsLocked()进行重新调度，接下来详细研究该函数（该函数之前是在AMS.java中的，后来改到了BroadcastQueue.java，这是queue的类型）的执行：

1. **public** **void** scheduleBroadcastsLocked() {
2. **if** (DEBUG\_BROADCAST) Slog.v(TAG\_BROADCAST, "Schedule broadcasts ["
3. + mQueueName + "]: current="
4. + mBroadcastsScheduled);
6. **if** (mBroadcastsScheduled) {
7. **return**;
8. }
9. mHandler.sendMessage(mHandler.obtainMessage(BROADCAST\_INTENT\_MSG, **this**));
10. mBroadcastsScheduled = **true**;
11. }

BroadcastQueue(原来在AMS中)的成员变量mBroadcastsScheduled用来描述BroadcastQueue是否已经向所运行的线程消息队列发送了一个类型为BROADCAST\_INTENT\_MSG的消息。BroadcastQueue就是通过ROADCAST\_INTENT\_MSG这个消息来调度有序广播调度队列mOrderedBroadcasts和无序广播调度队列mParallelBroadcasts里面的广播转发任务。第6行判断在BroadcastQueue（其实就是AMS）所在线程的消息队列中是否已经存在一个BROADCAST\_INTENT\_MSG，如果有直接返回，如果没有，就在第9行向该消息队列发送一个该类型的消息，并且将mBroadcastsScheduled 改为true。mHandler.sendMessage的消息在HandlerMessage中的处理：

1. @Override
2. **public** **void** handleMessage(Message msg) {
3. **switch** (msg.what) {
4. **case** BROADCAST\_INTENT\_MSG: {
5. **if** (DEBUG\_BROADCAST) Slog.v(
6. TAG\_BROADCAST, "Received BROADCAST\_INTENT\_MSG");
7. processNextBroadcast(**true**);
8. } **break**;
9. **case** BROADCAST\_TIMEOUT\_MSG: {
10. synchronized (mService) {
11. broadcastTimeoutLocked(**true**);
12. }
13. } **break**;
14. **case** SCHEDULE\_TEMP\_WHITELIST\_MSG: {
15. DeviceIdleController.LocalService dic = mService.mLocalDeviceIdleController;
16. **if** (dic != null) {
17. dic.addPowerSaveTempWhitelistAppDirect(UserHandle.getAppId(msg.arg1),
18. msg.arg2, **true**, (String)msg.obj);
19. }
20. } **break**;
21. }
22. }
23. };

可以看出是调用了processNextBroadcast函数，这个代码又是很长，所以分块处理（processNextBroadcast第一块：处理mParallelBroadcasts成员）：

1. final **void** processNextBroadcast(boolean fromMsg) {
2. synchronized(mService) {
3. BroadcastRecord r;
5. **if** (DEBUG\_BROADCAST) Slog.v(TAG\_BROADCAST, "processNextBroadcast ["
6. + mQueueName + "]: "
7. + mParallelBroadcasts.size() + " broadcasts, "
8. + mOrderedBroadcasts.size() + " ordered broadcasts");
10. mService.updateCpuStats();  //更新CPU使用情况
12. **if** (fromMsg) {
13. mBroadcastsScheduled = **false**;
14. }
16. // First, deliver any non-serialized broadcasts right away.
17. **while** (mParallelBroadcasts.size() > 0) {
18. r = mParallelBroadcasts.remove(0);
19. r.dispatchTime = SystemClock.uptimeMillis();
20. r.dispatchClockTime = System.currentTimeMillis();
21. final **int** N = r.receivers.size();
22. **if** (DEBUG\_BROADCAST\_LIGHT) Slog.v(TAG\_BROADCAST, "Processing parallel broadcast ["
23. + mQueueName + "] " + r);
24. **for** (**int** i=0; i<N; i++) {
25. Object target = r.receivers.get(i);
26. **if** (DEBUG\_BROADCAST)  Slog.v(TAG\_BROADCAST,
27. "Delivering non-ordered on [" + mQueueName + "] to registered "
28. + target + ": " + r);
29. deliverToRegisteredReceiverLocked(r, (BroadcastFilter)target, **false**);
30. }
31. addBroadcastToHistoryLocked(r);
32. **if** (DEBUG\_BROADCAST\_LIGHT) Slog.v(TAG\_BROADCAST, "Done with parallel broadcast ["
33. + mQueueName + "] " + r);
34. }

fromMsg表示processNextBroadcast是不是由BROADCAST\_INTENT\_MSG消息启动的，12行代码，如果是，则将mBroadcastsScheduled 置为false 。第17-34行来处理无序广播的mParallelBroadcasts列表，在循环中处理mParallelBroadcasts中的每一个BroadcastRecord ，第18-20行，依次取出保存在r中，并设置相关信息。广播记录块BroadcastRecord的receivers列表中包含了要接收这个广播的目标列表，所以在25行取出每一个该广播的接收者，由于在广播接收者注册的时候，每一个广播接收者都是BroadcastFilter对象，所以在29行中将target强转为BroadcastFilter，调用deliverToRegisteredReceiverLocked（该函数在后面介绍）将无序广播发送给每一个广播的接收者（实际上，这种处理方式也会造成惊群效应，但影响相对较少）。第31行将这条处理过的记录保存到mHistoryBroadcast中，供调试使用。接着分析下一块代码（processNextBroadcast第二块： 现在要处理mOrderedBroadcasts中的成员）：

1. // Now take care of the next serialized one...
3. // If we are waiting for a process to come up to handle the next
4. // broadcast, then do nothing at this point.  Just in case, we
5. // check that the process we're waiting for still exists.
6. **if** (mPendingBroadcast != null) {
7. **if** (DEBUG\_BROADCAST\_LIGHT) Slog.v(TAG\_BROADCAST,
8. "processNextBroadcast [" + mQueueName + "]: waiting for "
9. + mPendingBroadcast.curApp);
11. boolean isDead;
12. synchronized (mService.mPidsSelfLocked) {
13. ProcessRecord proc = mService.mPidsSelfLocked.get(mPendingBroadcast.curApp.pid);
14. isDead = proc == null || proc.crashing;
15. }
16. **if** (!isDead) {
17. // It's still alive, so keep waiting
18. **return**;
19. } **else** {
20. Slog.w(TAG, "pending app  ["
21. + mQueueName + "]" + mPendingBroadcast.curApp
22. + " died before responding to broadcast");
23. mPendingBroadcast.state = BroadcastRecord.IDLE;
24. mPendingBroadcast.nextReceiver = mPendingBroadcastRecvIndex;
25. mPendingBroadcast = null;
26. }
27. }

我们知道在广播的发送中，如果有静态注册的广播接收者（在mOrderedBroadcasts中），为了让这些接收者收到广播，AMS要确保这些广播接收者所在的进程启动起来（如果是动态接收者，接收者所在的进程一定是启动的，没有启动怎么动态注册，对不？）。只要静态注册接收者所在的进程是启动的，当该进程收到AMS的广播之后就会启动相应的广播接收者。该块函数是为了将mOrderedBroadcasts中的广播发送出去做准备，处理mOrderedBroadcasts中的成员。如前所述，它要处理一个接一个的接收者，如果接收者所在进程还未启动，则需要等待。mPendingBroadcast变量用于标识正在等待静态注册的目标广播者启动的广播转发任务。第6行，有需要启动的广播接收者，就为true。11-19行是在判断该函数是否为一个死进程，如果不是，继续等待。如果是一个死进程，进行23-25行的操作，没有将广播接收者启动，我们需要重新设置状态。

1. boolean looped = **false**;
2. **do** {
3. **if** (mOrderedBroadcasts.size() == 0) {
4. // No more broadcasts pending, so all done!
5. mService.scheduleAppGcsLocked();
6. **if** (looped) {
7. // If we had finished the last ordered broadcast, then
8. // make sure all processes have correct oom and sched
9. // adjustments.
10. mService.updateOomAdjLocked();
11. }
12. **return**;
13. }
14. r = mOrderedBroadcasts.get(0);
15. boolean forceReceive = **false**;
17. // Ensure that even if something goes awry with the timeout
18. // detection, we catch "hung" broadcasts here, discard them,
19. // and continue to make progress.
20. //
21. // This is only done if the system is ready so that PRE\_BOOT\_COMPLETED
22. // receivers don't get executed with timeouts. They're intended for
23. // one time heavy lifting after system upgrades and can take
24. // significant amounts of time.
25. **int** numReceivers = (r.receivers != null) ? r.receivers.size() : 0;
26. **if** (mService.mProcessesReady && r.dispatchTime > 0) {
27. **long** now = SystemClock.uptimeMillis();
28. **if** ((numReceivers > 0) &&
29. (now > r.dispatchTime + (2\*mTimeoutPeriod\*numReceivers))) {
30. Slog.w(TAG, "Hung broadcast ["
31. + mQueueName + "] discarded after timeout failure:"
32. + " now=" + now
33. + " dispatchTime=" + r.dispatchTime
34. + " startTime=" + r.receiverTime
35. + " intent=" + r.intent
36. + " numReceivers=" + numReceivers
37. + " nextReceiver=" + r.nextReceiver
38. + " state=" + r.state);
39. broadcastTimeoutLocked(**false**); // forcibly finish this broadcast
40. forceReceive = **true**;
41. r.state = BroadcastRecord.IDLE;
42. }
43. }
45. **if** (r.state != BroadcastRecord.IDLE) {
46. **if** (DEBUG\_BROADCAST) Slog.d(TAG\_BROADCAST,
47. "processNextBroadcast("
48. + mQueueName + ") called when not idle (state="
49. + r.state + ")");
50. **return**;
51. }
53. **if** (r.receivers == null || r.nextReceiver >= numReceivers
54. || r.resultAbort || forceReceive) {
55. // No more receivers for this broadcast!  Send the final
56. // result if requested...
57. **if** (r.resultTo != null) {
58. **try** {
59. **if** (DEBUG\_BROADCAST) Slog.i(TAG\_BROADCAST,
60. "Finishing broadcast [" + mQueueName + "] "
61. + r.intent.getAction() + " app=" + r.callerApp);
62. performReceiveLocked(r.callerApp, r.resultTo,
63. **new** Intent(r.intent), r.resultCode,
64. r.resultData, r.resultExtras, **false**, **false**, r.userId);
65. // Set this to null so that the reference
66. // (local and remote) isn't kept in the mBroadcastHistory.
67. r.resultTo = null;
68. } **catch** (RemoteException e) {
69. r.resultTo = null;
70. Slog.w(TAG, "Failure ["
71. + mQueueName + "] sending broadcast result of "
72. + r.intent, e);
73. }
74. }
76. **if** (DEBUG\_BROADCAST) Slog.v(TAG\_BROADCAST, "Cancelling BROADCAST\_TIMEOUT\_MSG");
77. cancelBroadcastTimeoutLocked();
79. **if** (DEBUG\_BROADCAST\_LIGHT) Slog.v(TAG\_BROADCAST,
80. "Finished with ordered broadcast " + r);
82. // ... and on to the next...
83. addBroadcastToHistoryLocked(r);
84. mOrderedBroadcasts.remove(0);
85. r = null;
86. looped = **true**;
87. **continue**;
88. }
89. } **while** (r == null);

第2-89行代码的主要工作就是循环（while循环）的把mOrderedBroadcasts中的下一个等待处理的广播任务取出，如果上一条还没有处理完成，或者是其他原因，则不能去取出新的BroadcastRecord对象r，即r == null一直成立，AMS就在这一直等待，一直等到取出下一条需要处理的BroadcastRecord对象r。

第2-13行，处理mOrderedBroadcasts为空，为空则说明mOrderedBroadcasts里的广播转发任务处理完了。14行，不为空就取出，保存在BroadcastRecord对象r中（下一个需要处理的广播任务就是在mOrderedBroadcasts中的第一个位置，所以使用get(0)）。25行获得目标广播接收者的个数，26-43行，下面这段代码用于判断此条广播是否处理时间过长。28-29行判断，上一条广播（因为只有上一条执行完，才进入下一条，当执行完才在84行执行remove，不然在while中一直取出的都是同一条BroadcastRecord对象r）是否在2\*mTimeoutPeriod\*numReceivers毫秒内还没有完成，如果没有完成，则用broadcastTimeoutLocked函数强制结束。40-41表示要继续处理mOrderedBroadcasts里下一条广播。45-51判断r描述的AMS是不是正在执行一个广播转发任务，如果是则AMS需要等待，50行直接返回。53-87行就是处理当广播是已经处理完，或者是被强制结束的操作。（57-74没看懂。。。。。。）。第83行，删除消息队列中的BROADCAST\_TIMEOUT\_MSG消息（39行的broadcastTimeoutLocked函数发送的），表示r所描述的广播已经在规定时间处理完成了。

在上面的代码中，如果能结束**while** (r == null)的循环，说明已经取到了下一个需要处理的广播转发任务，接下来就该对该广播转发任务进行处理了,代码比较长， 这里先处理如果是有序广播动态注册的广播情况（该情况相对简单，因为广播接收者所在的进程已经启动了）:

1. // Get the next receiver...
2. **int** recIdx = r.nextReceiver++;
4. // Keep track of when this receiver started, and make sure there
5. // is a timeout message pending to kill it if need be.
6. r.receiverTime = SystemClock.uptimeMillis();
7. **if** (recIdx == 0) {
8. r.dispatchTime = r.receiverTime;
9. r.dispatchClockTime = System.currentTimeMillis();
10. **if** (DEBUG\_BROADCAST\_LIGHT) Slog.v(TAG\_BROADCAST, "Processing ordered broadcast ["
11. + mQueueName + "] " + r);
12. }
13. **if** (! mPendingBroadcastTimeoutMessage) {
14. **long** timeoutTime = r.receiverTime + mTimeoutPeriod;
15. **if** (DEBUG\_BROADCAST) Slog.v(TAG\_BROADCAST,
16. "Submitting BROADCAST\_TIMEOUT\_MSG ["
17. + mQueueName + "] for " + r + " at " + timeoutTime);
18. setBroadcastTimeoutLocked(timeoutTime);
19. }
21. final BroadcastOptions brOptions = r.options;
22. final Object nextReceiver = r.receivers.get(recIdx);
24. **if** (nextReceiver instanceof BroadcastFilter) {
25. // Simple case: this is a registered receiver who gets
26. // a direct call.
27. BroadcastFilter filter = (BroadcastFilter)nextReceiver;
28. **if** (DEBUG\_BROADCAST)  Slog.v(TAG\_BROADCAST,
29. "Delivering ordered ["
30. + mQueueName + "] to registered "
31. + filter + ": " + r);
32. deliverToRegisteredReceiverLocked(r, filter, r.ordered);
33. **if** (r.receiver == null || !r.ordered) {
34. // The receiver has already finished, so schedule to
35. // process the next one.
36. **if** (DEBUG\_BROADCAST) Slog.v(TAG\_BROADCAST, "Quick finishing ["
37. + mQueueName + "]: ordered="
38. + r.ordered + " receiver=" + r.receiver);
39. r.state = BroadcastRecord.IDLE;
40. scheduleBroadcastsLocked();
41. } **else** {
42. **if** (brOptions != null && brOptions.getTemporaryAppWhitelistDuration() > 0) {
43. scheduleTempWhitelistLocked(filter.owningUid,
44. brOptions.getTemporaryAppWhitelistDuration(), r);
45. }
46. }
47. **return**;
48. }

第2行，取出下一个需要被处理的接收者的位置，4-19行，为了处理广播是否超时。22行，取出接收者。24-48处理动态注册的广播（因为动态广播接收者的类型为BroadcastFilter）。32行，如果是动态接收者，则直接调用deliverToRegisteredReceiverLocked处理。33-41行，接收者为空或者是无序广播接收者，说明该广播处理完，不用等待，所以执行39行，给AMS说明，不需要等待前一个广播处理完成， 直接可以处理下一个广播。

接下来分析，如果是静态注册的广播，该广播就有点麻烦，要考虑对应的进程是否启动。该块代码实在繁琐，只拿出主要的代码分析：

1. //如果是静态接收者，则它的真实类型是ResolveInfo
2. ResolveInfo info =  (ResolveInfo)nextReceiver;
3. booleanskip = **false**;
4. //检查权限
5. **int** perm =checkComponentPermission(info.activityInfo.permission,
6. r.callingPid, r.callingUid, info.activityInfo.applicationInfo.uid,
7. info.activityInfo.exported);
8. **if** (perm!= PackageManager.PERMISSION\_GRANTED) skip = **true**;
9. **if**(info.activityInfo.applicationInfo.uid != Process.SYSTEM\_UID &&
10. r.requiredPermission != **null**) {
11. ......
12. }
13. //设置skip为true
14. **if**(r.curApp != **null** && r.curApp.crashing) skip = **true**;
16. **if** (skip){
17. r.receiver = **null**;
18. r.curFilter = **null**;
19. r.state = BroadcastRecord.IDLE;
20. scheduleBroadcastsLocked();//再调度一次广播处理
21. **return**;
22. }
24. r.state= BroadcastRecord.APP\_RECEIVE;
25. StringtargetProcess = info.activityInfo.processName;
26. r.curComponent = **new** ComponentName(
27. info.activityInfo.applicationInfo.packageName,
28. info.activityInfo.name);
29. r.curReceiver = info.activityInfo;
30. **try** {
31. //设置Package stopped的状态为false
32. AppGlobals.getPackageManager().setPackageStoppedState(
33. r.curComponent.getPackageName(),**false**);
34. } ......
35. // targetProcess是广播接收者的android:process属性
36. ProcessRecord app = getProcessRecordLocked(targetProcess,
37. info.activityInfo.applicationInfo.uid);
38. //如果该接收者对应的进程已经存在（检查是否已经启动）
39. **if** (app!= **null** && app.thread != **null**) {
40. **try** {
41. app.addPackage(info.activityInfo.packageName);
42. //该函数内部调用应用进程的scheduleReceiver函数，读者可自行分析
43. processCurBroadcastLocked(r, app);
44. **return**;//已经触发该接收者处理本广播，需要等待处理结果
45. }......
46. }
47. //最糟的情况就是该进程还没有启动
48. **if**((r.curApp=startProcessLocked(targetProcess,
49. info.activityInfo.applicationInfo, **true**,......)!= 0)) == **null**) {
50. ......//进程启动失败的处理
51. //启动失败，调用scheduleBroadcastsLocked结束对BroadcastRecord对象r所描述的广播转发任务
52. scheduleBroadcastsLocked();
53. r.state = BroadcastRecord.IDLE;
54. **return**;
55. }
56. //表明AMS正在等待进程的启动
57. mPendingBroadcast = r; //设置mPendingBroadcast
58. mPendingBroadcastRecvIndex = recIdx;
59. }

静态广播的接收者的类型为ResolveInfo 。

· 如果广播接收者为动态注册对象，则直接调用deliverToRegisteredReceiverLocked处理它。

· 如果广播接收者为静态注册对象，并且该对象对应的进程已经存在，则调用processCurBroadcastLocked处理它。

· 如果广播接收者为静态注册对象，并且该对象对应的进程还不存在，则需要创建该进程。这是最糟糕的情况。

在无序，还是有序的广播任务处理过程中，对于动态广播的处理最终都是deliverToRegisteredReceiverLocked函数来处理，接下来我们来分析该函数的执行：

1. **private** **final** **void** deliverToRegisteredReceiverLocked(BroadcastRecord r,
2. BroadcastFilter filter, booleanordered) {
3. booleanskip = **false**;
4. //检查发送进程是否有filter要求的权限
5. **if**(filter.requiredPermission != **null**) {
6. intperm = checkComponentPermission(filter.requiredPermission,
7. r.callingPid, r.callingUid, -1, **true**);
8. **if**(perm != PackageManager.PERMISSION\_GRANTED) skip = **true**;
9. }
10. //检查接收者是否有发送者要求的权限
11. **if**(r.requiredPermission != **null**) {
12. intperm = checkComponentPermission(r.requiredPermission,
13. filter.receiverList.pid, filter.receiverList.uid, -1, **true**);
14. **if**(perm != PackageManager.PERMISSION\_GRANTED) skip = **true**;
15. }
17. **if**(!skip) {
18. **if**(ordered) {
19. ......//设置一些状态，成员变量等信息，不涉及广播发送
20. }
21. **try** {
22. //发送广播
23. performReceiveLocked(filter.receiverList.app,
24. filter.receiverList.receiver,**new** Intent(r.intent), r.resultCode,
25. r.resultData, r.resultExtras, r.ordered, r.initialSticky);
27. **if**(ordered) r.state = BroadcastRecord.CALL\_DONE\_RECEIVE;
28. }......
29. }
30. }
31. }

在广播的发送过程中，需要检查广播是否有相关权限，检查是双向的，即发送端和接收端都要检查，在4-15行的代码就是在执行相关的检查工作。检查过程主要是使用checkComponentPermission函数检查对方的PID和UID（用户身份证明(User Identification)的缩写）是否符合要求。若果不需要检查权限或者权限检查通过。Skip就会被置为false。就会进入17行。通过函数performReceiveLocked来发送广播给接收者去处理：

1. **private** **static** **void** performReceiveLocked(ProcessRecord app, IIntentReceiver receiver,
2. Intent intent, **int** resultCode, String data, Bundle extras,
3. **boolean** ordered, **boolean** sticky, **int** sendingUser) **throws** RemoteException {
4. // Send the intent to the receiver asynchronously using one-way binder calls.
5. **if** (app != **null**) {
6. **if** (app.thread != **null**) {
7. // If we have an app thread, do the call through that so it is
8. // correctly ordered with other one-way calls.
9. app.thread.scheduleRegisteredReceiver(receiver, intent, resultCode,
10. data, extras, ordered, sticky, sendingUser, app.repProcState);
11. } **else** {
12. // Application has died. Receiver doesn't exist.
13. **throw** **new** RemoteException("app.thread must not be null");
14. }
15. } **else** {
16. receiver.performReceive(intent, resultCode, data, extras, ordered,
17. sticky, sendingUser);
18. }
19. }

App是一个processrecord对象，描述广播接收者所在的进程；Receiver指向了一个实现了IIntentReceiver 的Binder代理，描述了目标广播接收者；intent是即将要发送的广播，

第9行，注册动态广播接收者，应用进程ApplicationThread的scheduleRegisteredReceiver函数将被调用，进行发送广播。来看看scheduleRegisteredReceiver函数（这里是代理所以是ApplicationThreadNative.java文件中的ApplicationThreadProxy. scheduleRegisteredReceiver）：

1. **public** **void** scheduleRegisteredReceiver(IIntentReceiver receiver, Intent intent,
2. **int** resultCode, String dataStr, Bundle extras, **boolean** ordered,
3. **boolean** sticky, **int** sendingUser, **int** processState) **throws** RemoteException {
4. Parcel data = Parcel.obtain();
5. data.writeInterfaceToken(IApplicationThread.descriptor);
6. data.writeStrongBinder(receiver.asBinder());
7. intent.writeToParcel(data, 0);
8. data.writeInt(resultCode);
9. data.writeString(dataStr);
10. data.writeBundle(extras);
11. data.writeInt(ordered ? 1 : 0);
12. data.writeInt(sticky ? 1 : 0);
13. data.writeInt(sendingUser);
14. data.writeInt(processState);
15. mRemote.transact(SCHEDULE\_REGISTERED\_RECEIVER\_TRANSACTION, data, **null**,
16. IBinder.FLAG\_ONEWAY);
17. data.recycle();
18. }

ApplicationThreadProxy通过Binder代理发送了一个SCHEDULE\_REGISTERED\_RECEIVER\_TRANSACTION进程间请求。在16行设置了IBinder.FLAG\_ONEWAY，所以这个是一个异步请求。

到此在AMS中的处理告一段落，现在进入到广播接收者的应用进程中,分析接收者收到广播怎么处理（ApplicationThread.java(在ActivityThread.java中)：scheduleRegisteredReceiver）：

1. // This function exists to make sure all receiver dispatching is
2. // correctly ordered, since these are one-way calls and the binder driver
3. // applies transaction ordering per object for such calls.
4. **public** **void** scheduleRegisteredReceiver(IIntentReceiver receiver, Intent intent,
5. **int** resultCode, String dataStr, Bundle extras, **boolean** ordered,
6. **boolean** sticky, **int** sendingUser, **int** processState) **throws** RemoteException {
7. updateProcessState(processState, **false**);
8. receiver.performReceive(intent, resultCode, dataStr, extras, ordered,
9. sticky, sendingUser);
10. }

Receiver是指向了一个IntentReceiver对象，每一个IntentReceiver对象中都封装了一个广播接收者，并且代替它的广播接收者注册到AMS中。也就是当AMS给广播接受者发送广播时，其实是发送给IntentReceiver对象，IntentReceiver对象是通过performReceive来接受广播的，这也就是第8行代码。

继续分析IntentReceiver对象的performReceive函数的实现，（LoadedApk.java中ReceiverDispatcher:IntentReceiver：: performReceive）:

1. **public** **void** performReceive(Intent intent, **int** resultCode, String data,
2. Bundle extras, **boolean** ordered, **boolean** sticky, **int** sendingUser) {
3. LoadedApk.ReceiverDispatcher rd = mDispatcher.get();
4. ......
5. **if** (rd != **null**) {
6. rd.performReceive(intent, resultCode, data, extras,
7. ordered, sticky, sendingUser);
8. } **else** {
9. ......
10. }
11. }

调用LoadedApk.ReceiverDispatcher的performReceive方法：

1. **public** **void** performReceive(Intent intent, **int** resultCode, String data,
2. Bundle extras, **boolean** ordered, **boolean** sticky, **int** sendingUser) {
3. **if** (ActivityThread.DEBUG\_BROADCAST) {
4. **int** seq = intent.getIntExtra("seq", -1);
5. Slog.i(ActivityThread.TAG, "Enqueueing broadcast " + intent.getAction() + " seq=" + seq
6. + " to " + mReceiver);
7. }
8. Args args = **new** Args(intent, resultCode, data, extras, ordered,
9. sticky, sendingUser);
10. **if** (!mActivityThread.post(args)) {
11. **if** (mRegistered && ordered) {
12. IActivityManager mgr = ActivityManagerNative.getDefault();
13. **if** (ActivityThread.DEBUG\_BROADCAST) Slog.i(ActivityThread.TAG,
14. "Finishing sync broadcast to " + mReceiver);
15. args.sendFinished(mgr);
16. }
17. }
18. }

mActivityThread是一个Handler对象，指向了ActivityThread的成员变量mH， 用来向广播接收者的主线程的消息队列里发送消息的。第9行将intent描述的广播封装成一个Args对象，Args是一个runnable对象。第10行，注意这里处理消息的方式是通过Handler.post函数进行的，post函数的参数是Runnable类型的，这个消息最终会调用这个这个参数的run成员函数来处理。scheduleRegisteredReceiver最终向主线程的Handler投递了一个Args对象，这个对象的run函数将在主线程中被调用。

1. **public** **void** run() {
2. **final** BroadcastReceiver receiver = mReceiver;
3. **final** **boolean** ordered = mOrdered;
4. ......
5. **final** IActivityManager mgr = ActivityManagerNative.getDefault();
6. **final** Intent intent = mCurIntent;
7. mCurIntent = **null**;
9. **if** (receiver == **null** || mForgotten) {
10. **if** (mRegistered && ordered) {  // mRegistered 用来描述广播接收者是否已经注册到AMS中了，ordered是否是有序广播
11. **if** (ActivityThread.DEBUG\_BROADCAST) Slog.i(ActivityThread.TAG,
12. "Finishing null broadcast to " + mReceiver);
13. sendFinished(mgr);
14. }
15. **return**;
16. }
18. Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "broadcastReceiveReg");
19. **try** {
20. ClassLoader cl =  mReceiver.getClass().getClassLoader();
21. intent.setExtrasClassLoader(cl);
22. setExtrasClassLoader(cl);
23. receiver.setPendingResult(**this**);
24. receiver.onReceive(mContext, intent);
25. } **catch** (Exception e) {
26. ......
27. }
29. **if** (receiver.getPendingResult() != **null**) {
30. finish();
31. }
32. Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);
33. }

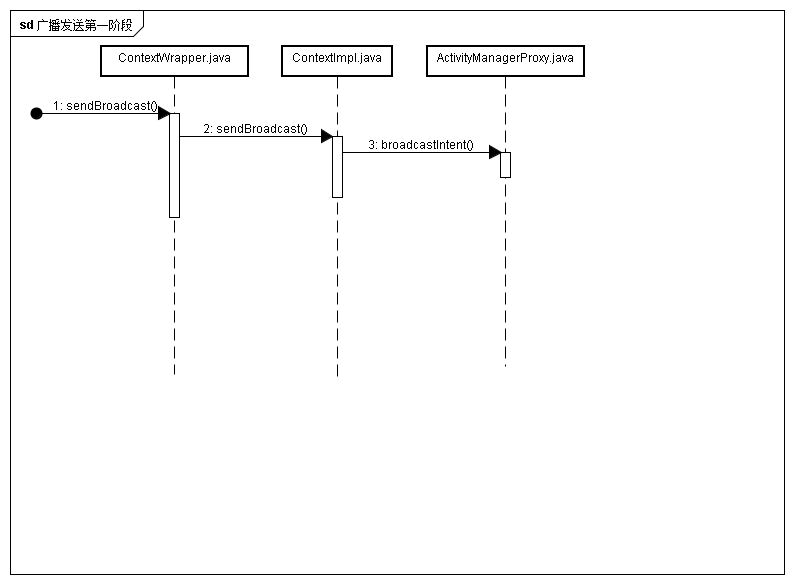
第2行，mReceiver是ReceiverDispatcher类的成员变量，它的类型是BroadcastReceiver，这里它就是MainActivity注册广播接收器时创建的BroadcastReceiver实例了。第6行，mCurIntent是用来描述一个广播。所以，24行，描述广播接收者的receiver调用接收函数onReceive来接受广播。13行，通过mgr给AMS发送一个完成有序广播完成的消息，让AMS可以进行目标广播接收者的处理。sendFinished是Args类的父类，直接调用父类的函数，进行处理。

24行的receiver.onReceive，其实也就是我们在例子代码里面的BroadcastReceiver实例counterActionReceiver中的onReceive函数，这样，发送的广播就到达接收者手里了，接收者可以对接收的广播进行处理。

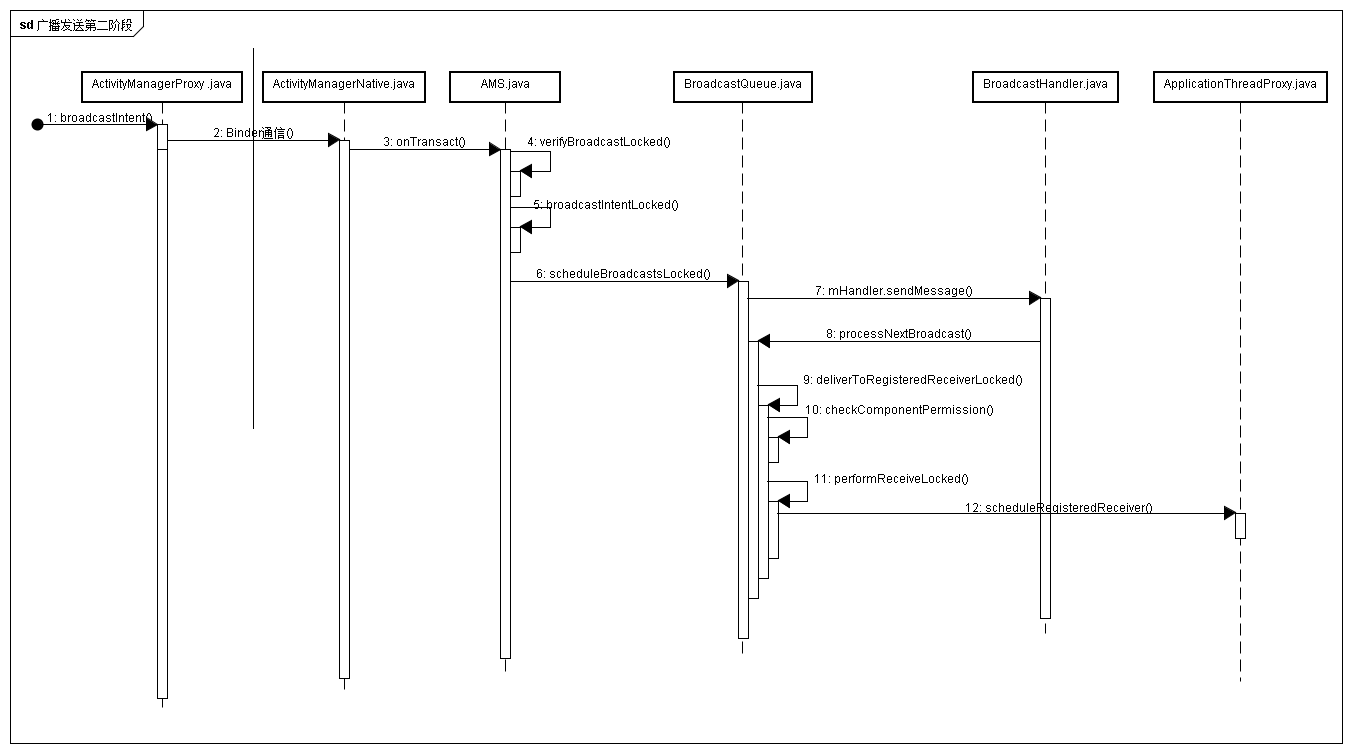
**总结：**

在广播发送的过程中，从所在进程，主要分为三个阶段，第一阶段是在广播发送者的进程中；第二阶段是在AMS的进程中（system\_server进程）;第三阶段广播进入广播接收者的进程。

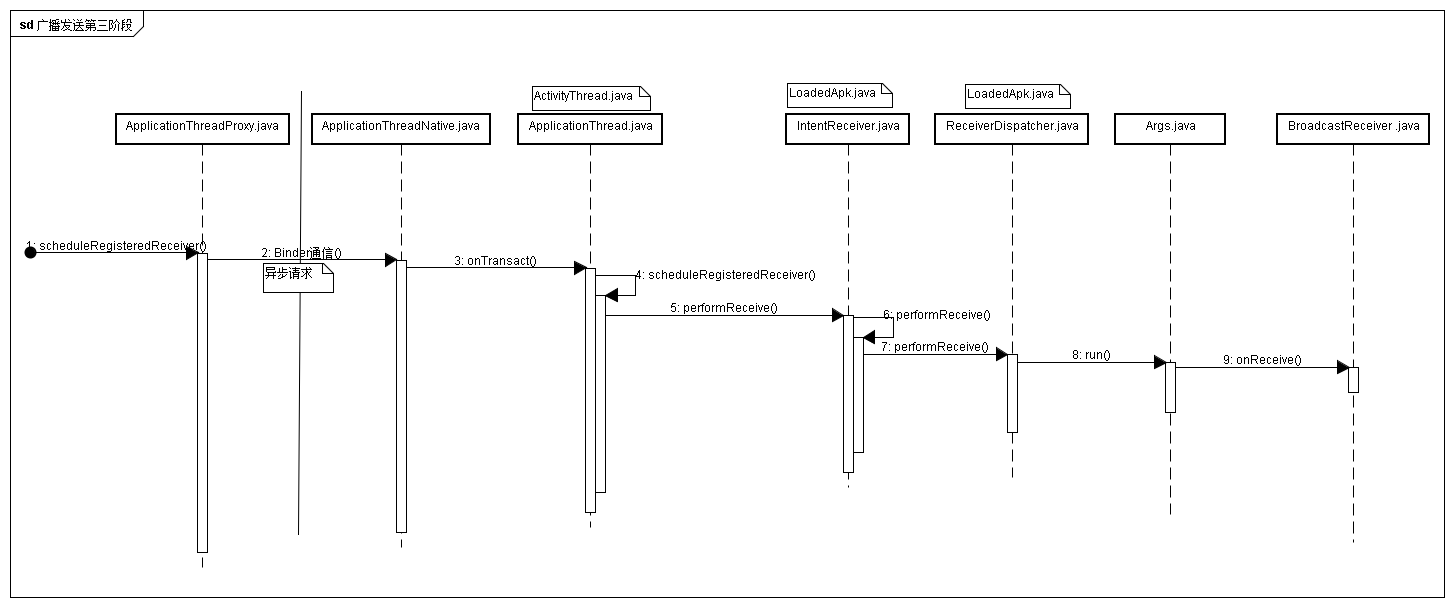
第一阶段流程图：



第二阶段流程图：



第三阶段流程图：

****