本文依据罗升阳博客，以及Android 6进行调试总结而得。

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

说明：四大组件 Activity、Service、Broadcast Receiver、Content Provider。Broadcast Receiver是广播接收者，Activity、Service可以用于发送广播，也就是广播的发送者。

广播的注册中心：AMS

注册方式：静态注册：在AndroidManifest.xml中配置，信息是保存在PMS中，该接收者科可能没有

动态注册：在代码中注册，所以所在进程一定是启动的

**广播接收者的注册过程：**

在APP中的Broadcounter中的onResume中：

1. **public** **void** onResume() {
2. **super**.onResume();
3. IntentFilter counterActionFilter = **new** IntentFilter(CounterService.BROADCAST\_COUNTER\_ACTION);
4. registerReceiver(counterActionReceiver, counterActionFilter);
5. }

调用Activity的父类方法ContextWrapper 的registerReceiver：

1. @Override
2. **public** Intent registerReceiver(
3. BroadcastReceiver receiver, IntentFilter filter,
4. String broadcastPermission, Handler scheduler) {
5. **return** mBase.registerReceiver(receiver, filter, broadcastPermission,scheduler);
6. }

mBase其实是定义为Context，其实是其子类ContextImpl：

1. @Override
2. **public** Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter) {
3. **return** registerReceiver(receiver, filter, **null**, **null**);
4. }

再调用ContextImpl里的registerReceiver(receiver, filter, **null**, **null**)：

1. **public** Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter,
2. String broadcastPermission, Handler scheduler) {
3. **return** registerReceiverInternal(receiver, getUserId(),
4. filter, broadcastPermission, scheduler, getOuterContext());
5. }

在Android 6 中比老罗用的版本多了一个参数getUserId(),看下这个参数干什么了:

1. /\*\* {@hide} \*/
2. @Override
3. **public** **int** getUserId() {
4. **return** mUser.getIdentifier();
5. }

这里的mUser是UserHandle类（这个类据说是在4.2.2以后加入多用户使用的）：

1. **private** **final** UserHandle mUser;

继续看UserHandle类的方法getIdentifier()：

1. /\*\*
2. \* Returns the userId stored in this UserHandle.
3. \* @hide
4. \*/
5. @SystemApi
6. **public** **int** getIdentifier() {
7. **return** mHandle;
8. }

这里返回了一个mHandle，具体干嘛的？？？？

在ContextImpl的registerReceiver 的返回方法registerReceiverInternal的参数中调用了getOuterContext()方法：

1. **final** Context getOuterContext() {
2. **return** mOuterContext;
3. }

其中，mOuterContext的定义：

1. **private** Context mOuterContext;

mOuterContext是用来指向一个Activity组件的，这里就是指向Broadcounter组件。

结束registerReceiver方法，进入registerReceiverInternal方法：

1. **private** Intent registerReceiverInternal(BroadcastReceiver receiver, **int** userId,
2. IntentFilter filter, String broadcastPermission,
3. Handler scheduler, Context context) {
4. IIntentReceiver rd = **null**;
5. **if** (receiver != **null**) {
6. **if** (mPackageInfo != **null** && context != **null**) {
7. **if** (scheduler == **null**) {
8. scheduler = mMainThread.getHandler();
9. }
10. rd = mPackageInfo.getReceiverDispatcher(
11. receiver, context, scheduler,
12. mMainThread.getInstrumentation(), **true**);
13. } **else** {
14. **if** (scheduler == **null**) {
15. scheduler = mMainThread.getHandler();
16. }
17. rd = **new** LoadedApk.ReceiverDispatcher(
18. receiver, context, scheduler, **null**, **true**).getIIntentReceiver();
19. }
20. }
21. **try** {
22. **return** ActivityManagerNative.getDefault().registerReceiver(
23. mMainThread.getApplicationThread(), mBasePackageName,
24. rd, filter, broadcastPermission, userId);
25. } **catch** (RemoteException e) {
26. **return** **null**;
27. }
28. }

第8行mMainThread.getHandler():

1. **final** ActivityThread mMainThread;

mMainThread用来描述当前进程。ActivityThread. getHandler()：

1. **final** Handler getHandler() {
2. **return** mH;
3. }

得到一个mH（handler对象），用来向当前的应用进程的主线程的消息队列发送消息。因为广播接收者不是从AMS直接获得广播的，而是广播被是封装成消息发送到应用的进程的主线程的消息队列中的，最后分发给消息的接收者。这个Hanlder是后面用来分发ActivityManagerService发送过的广播用的。

registerReceiverInternal方法的第10-12行：mMainThread.getInstrumentation()是获得该应用进程的mInstrumentation。mPackageInfo的定义：

1. **final** LoadedApk mPackageInfo;

进入LoadedApk.getReceiverDispatcher方法:

1. **public** IIntentReceiver getReceiverDispatcher(BroadcastReceiver r,
2. Context context, Handler handler,
3. Instrumentation instrumentation, **boolean** registered) {
4. **synchronized** (mReceivers) {
5. LoadedApk.ReceiverDispatcher rd = **null**;
6. ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher> map = **null**;
7. **if** (registered) {
8. map = mReceivers.get(context);
9. **if** (map != **null**) {
10. rd = map.get(r);
11. }
12. }
13. **if** (rd == **null**) {
14. rd = **new** ReceiverDispatcher(r, context, handler,
15. instrumentation, registered);
16. **if** (registered) {
17. **if** (map == **null**) {
18. map = **new** ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>();
19. mReceivers.put(context, map);
20. }
21. map.put(r, rd);
22. }
23. } **else** {
24. rd.validate(context, handler);
25. }
26. rd.mForgotten = **false**;
27. **return** rd.getIIntentReceiver();
28. }
29. }

getReceiverDispatcher方法是将receiver封装成IIntentReceiver类型的对象rd（Binder对象），当AMS收到相应的广播就通过该Binder来通知对应的Broadcounter(注册广播的Activity)。每个广播的接收者的Activity组件在LoadedApk中都有一个对应的ReceiverDispatcher对象，该对象就是将广播的注册者和注册的Activity关联起来，这些ReceiverDispatcher对象保存在ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>中，以广播接收者为关键字。第4行mReceivers 是一个ArrayMap对象：

1. **private** **final** ArrayMap<Context, ArrayMap<BroadcastReceiver, ReceiverDispatcher>> mReceivers
2. = **new** ArrayMap<Context, ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>>();

以所关联的Activity组件的Context接口为关键字，以广播的注册者和注册的Activity关联存储的ArrayMap为值。

第7-12行，判断context（getOuterContext（）获得，代表了广播的接收者）所指向的Broadcounter（广播的接收者的注册是在Broadcounter中）是否在ArrayMap<BroadcastReceiver, ReceiverDispatcher>（广播接收者BroadcastReceiver是关键字）中存在，如果不存在，13行rd==null成立，进入；如果存在则rd = map.get(r)，通过查找r，返回，13行rd==null不成立，跳转到24行。这里考虑rd==null成立的情况，接下来分析14-22行代码。既然rd不存在，那么在14行就创建一个ReceiverDispatcher（该类是LoadedApk的内部类）。创建了rd，总要放到第6行创建的map中吧，map没有初始化就初始化（第18行），map是刚创建的，自然也没有在mReceivers，所以要put进去（put进去就发现mReceivers的size变为了1）。map被put进了mReceivers，那么rd是不是也该put进map了（第21行就进行put，map之前已经初始化了就直接跳到这里，将rd put进map），被put进去之后就map的size就+1。（第26行不知道。。。干嘛了）。第27行的实现rd.getIIntentReceiver()，这里的rd是LoadedApk.ReceiverDispatcher对象，下来看看ReceiverDispatcher的getIIntentReceiver()的相关实现：

1. **static** **final** **class** ReceiverDispatcher {
2. **final** **static** **class** InnerReceiver **extends** IIntentReceiver.Stub {
3. **final** WeakReference<LoadedApk.ReceiverDispatcher> mDispatcher;
4. **final** LoadedApk.ReceiverDispatcher mStrongRef;
6. InnerReceiver(LoadedApk.ReceiverDispatcher rd, **boolean** strong) {
7. mDispatcher = **new** WeakReference<LoadedApk.ReceiverDispatcher>(rd);
8. mStrongRef = strong ? rd : **null**;
9. }
10. ......
11. **final** IIntentReceiver.Stub mIIntentReceiver;
12. **final** BroadcastReceiver mReceiver;
13. **final** Context mContext;
14. **final** Handler mActivityThread;
15. **final** Instrumentation mInstrumentation;
16. **final** **boolean** mRegistered;
17. **final** IntentReceiverLeaked mLocation;
18. RuntimeException mUnregisterLocation;
19. **boolean** mForgotten;
20. ......
21. ReceiverDispatcher(BroadcastReceiver receiver, Context context,
22. Handler activityThread, Instrumentation instrumentation,
23. **boolean** registered) {
24. ......
25. mIIntentReceiver = **new** InnerReceiver(**this**, !registered);
26. mReceiver = receiver;
27. mContext = context;
28. mActivityThread = activityThread;
29. mInstrumentation = instrumentation;
30. mRegistered = registered;
31. mLocation = **new** IntentReceiverLeaked(**null**);
32. mLocation.fillInStackTrace();
33. }
34. ......
35. IIntentReceiver getIIntentReceiver() {
36. **return** mIIntentReceiver;
37. }
38. ......
40. }
41. }

getIIntentReceiver()返回的mIIntentReceiver定义在第11行，由于InnerReceiver 是IIntentReceiver.Stub的子类，所以mIIntentReceiver是一个InnerReceiver 对象（在25行就用InnerReceiver 实例化了mIIntentReceiver）。在6-9行中具体实现了mIIntentReceiver的new的过程。看见内部有一个类型为LoadedApk.ReceiverDispatcher 的弱引用（WeakReference）指向了外部的ReceiverDispatcher（rd）对象。

第26-28行的三个变量mReceiver指Broadcounter组件内部用于接受广播的counterActionReceiver； mContext指向Mianactivity组件； mActivityThread是一个handler对象，指向Broadcounter组件所在进程的主线程的Handler对象。

到这，回到ContextImpl的registerReceiverInternal方法第17行，rd获得了返回的将本地对象封装的实现了IIntentReceiver的本地Binder对象。接下来会进入到registerReceiverInternal方法的22行（由于离的有点远，再把代码复制下来）：

1. **return** ActivityManagerNative.getDefault().registerReceiver(
2. mMainThread.getApplicationThread(), mBasePackageName,
3. rd, filter, broadcastPermission, userId);

ActivityManagerNative.getDefault()是获得了一个AMS的代理，即ActivityManagerProxy（在ActivityManagerNative.java内定义）对象。看来要是进行Bindler通信了，查看其registerReceiver方法：

1. **public** Intent registerReceiver(IApplicationThread caller, String packageName,
2. IIntentReceiver receiver,
3. IntentFilter filter, String perm, **int** userId) **throws** RemoteException
4. {
5. Parcel data = Parcel.obtain();
6. Parcel reply = Parcel.obtain();
7. data.writeInterfaceToken(IActivityManager.descriptor);
8. data.writeStrongBinder(caller != **null** ? caller.asBinder() : **null**);
9. data.writeString(packageName);
10. data.writeStrongBinder(receiver != **null** ? receiver.asBinder() : **null**);
11. filter.writeToParcel(data, 0);
12. data.writeString(perm);
13. data.writeInt(userId);
14. mRemote.transact(REGISTER\_RECEIVER\_TRANSACTION, data, reply, 0);
15. reply.readException();
16. Intent intent = **null**;
17. **int** haveIntent = reply.readInt();
18. **if** (haveIntent != 0) {
19. intent = Intent.CREATOR.createFromParcel(reply);
20. }
21. reply.recycle();
22. data.recycle();
23. **return** intent;
24. }

通过 mRemote.transact，发送了类型为 REGISTER\_RECEIVER\_TRANSACTION的请求，这样就在Broadcounter进程中的操作就完成了，接下来就要进入system\_server进程的AMS了，在AMS的native端，解析请求：

1. **case** REGISTER\_RECEIVER\_TRANSACTION:
2. {
3. data.enforceInterface(IActivityManager.descriptor);
4. IBinder b = data.readStrongBinder();
5. IApplicationThread app =
6. b != **null** ? ApplicationThreadNative.asInterface(b) : **null**;
7. String packageName = data.readString();
8. b = data.readStrongBinder();
9. IIntentReceiver rec
10. = b != **null** ? IIntentReceiver.Stub.asInterface(b) : **null**;
11. IntentFilter filter = IntentFilter.CREATOR.createFromParcel(data);
12. String perm = data.readString();
13. **int** userId = data.readInt();
14. Intent intent = registerReceiver(app, packageName, rec, filter, perm, userId);
15. reply.writeNoException();
16. **if** (intent != **null**) {
17. reply.writeInt(1);
18. intent.writeToParcel(reply, 0);
19. } **else** {
20. reply.writeInt(0);
21. }
22. **return** **true**;
23. }

其中核心函数14行，调用本地端的registerReceiver（AMS.java中）：

1. **public** Intent registerReceiver(IApplicationThread caller,
2. IIntentReceiver receiver, IntentFilter filter, String permission) {
3. **synchronized**(**this**) {
4. ProcessRecord callerApp = **null**;
5. **if** (caller != **null**) {
6. callerApp = getRecordForAppLocked(caller);
7. **if** (callerApp == **null**) {
8. ......
9. }
10. }
12. List allSticky = **null**;
14. // Look for any matching sticky broadcasts...
15. Iterator actions = filter.actionsIterator();
16. **if** (actions != **null**) {
17. **while** (actions.hasNext()) {
18. String action = (String)actions.next();
19. allSticky = getStickiesLocked(action, filter, allSticky);
20. }
21. } **else** {
22. ......
23. }
25. // The first sticky in the list is returned directly back to
26. // the client.
27. Intent sticky = allSticky != **null** ? (Intent)allSticky.get(0) : **null**;
29. ......
31. **if** (receiver == **null**) {
32. **return** sticky;
33. }
35. ReceiverList rl
36. = (ReceiverList)mRegisteredReceivers.get(receiver.asBinder());
37. **if** (rl == **null**) {
38. rl = **new** ReceiverList(**this**, callerApp,
39. Binder.getCallingPid(),
40. Binder.getCallingUid(), receiver);
42. **if** (rl.app != **null**) {
43. rl.app.receivers.add(rl);
44. } **else** {
45. ......
46. }
47. mRegisteredReceivers.put(receiver.asBinder(), rl);
48. }
50. BroadcastFilter bf = **new** BroadcastFilter(filter, rl, permission);
51. rl.add(bf);
52. ......
53. mReceiverResolver.addFilter(bf);
55. // Enqueue broadcasts for all existing stickies that match
56. // this filter.
57. **if** (allSticky != **null**) {
58. ......
59. }
61. **return** sticky;
62. }
63. }

第4行callerApp 是用来描述请求注册广播的接收者所在的Broadcounter的应用程序进程。第6行就通过参数caller将callerApp 实例化。第15-20行，在AMS中查找有没有和filter对应的黏性广播，如果存在将他们保存在allSticky （List ）列表中。广播接收者在AMS注册时，不是将接收者直接注册到AMS中，而是将与之关联的InnerReceiver（receivers是binder代理类，是一个类型为InnerReceiver的Binder本地对象）。在AMS中每一个广播接收者都用一个BroadcastFilter对象来描述， 由于在同一个应用中，不同的Activity可能在可能会使用同一个InnerReceiver来注册不同的广播接收者，