# Android插件化原理解析——广播的管理

发表于    |   [54](http://weishu.me/2016/04/12/understand-plugin-framework-receiver/" \l "comments)条评论   |   14855次阅读

在[Activity生命周期管理](http://weishu.me/2016/03/21/understand-plugin-framework-activity-management/) 以及 [插件加载机制](http://weishu.me/2016/04/05/understand-plugin-framework-classloader/) 中我们详细讲述了插件化过程中对于Activity组件的处理方式，为了实现Activity的插件化我们付出了相当多的努力；那么Android系统的其他组件，比如BroadcastReceiver，Service还有ContentProvider，它们又该如何处理呢？

相比Activity，BroadcastReceiver要简单很多——广播的生命周期相当简单；如果希望插件能够支持广播，这意味着什么？

回想一下我们日常开发的时候是如何使用BroadcastReceiver的：**注册**, **发送**和**接收**；因此，要实现BroadcastReceiver的插件化就这三种操作提供支持；接下来我们将一步步完成这个过程。

阅读本文之前，可以先clone一份 [understand-plugin-framework](https://github.com/tiann/understand-plugin-framework)，参考此项目的receiver-management 模块。另外，插件框架原理解析系列文章见[索引](http://weishu.me/2016/01/28/understand-plugin-framework-overview/)。

如果连BroadcastReceiver的工作原理都不清楚，又怎么能让插件支持它？老规矩，知己知彼。

## **源码分析**

我们可以注册一个BroadcastReceiver然后接收我们感兴趣的广播，也可以给某有缘人发出某个广播；因此，我们对源码的分析按照两条路线展开：

### **注册过程**

不论是静态广播还是动态广播，在使用之前都是需要注册的；动态广播的注册需要借助Context类的registerReceiver方法，而静态广播的注册直接在AndroidManifest.xml中声明即可；我们首先分析一下动态广播的注册过程。

Context类的registerReceiver的真正实现在ContextImpl里面，而这个方法间接调用了registerReceiverInternal，源码如下：

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

可以看到，BroadcastReceiver的注册也是通过AMS完成的；在进入AMS跟踪它的registerReceiver方法之前，我们先弄清楚这个IIntentReceiver类型的变量rd是什么。首先查阅API文档，很遗憾SDK里面没有导出这个类，我们直接去 [grepcode](http://grepcode.com/file/repository.grepcode.com/java/ext/com.google.android/android/5.0.1_r1/android/content/IIntentReceiver.java?av=f) 上看，文档如下：

System private API for dispatching intent broadcasts. This is given to the activity manager as part of registering for an intent broadcasts, and is called when it receives intents.

这个类是通过AIDL工具生成的，它是一个Binder对象，因此可以用来跨进程传输；文档说的很清楚，它是用来进行广播分发的。什么意思呢？

**由于广播的分发过程是在AMS中进行的，而AMS所在的进程和BroadcastReceiver所在的进程不一样，因此要把广播分发到BroadcastReceiver具体的进程需要进行跨进程通信，这个通信的载体就是IIntentReceiver类。**其实这个类的作用跟 [Activity生命周期管理](http://weishu.me/2016/03/21/understand-plugin-framework-activity-management/) 中提到的 IApplicationThread相同，都是App进程给AMS进程用来进行通信的对象。另外，IIntentReceiver是一个接口，从上述代码中可以看出，它的实现类为LoadedApk.ReceiverDispatcher。

OK，我们继续跟踪源码，AMS类的registerReceiver方法代码有点多，这里不一一解释了，感兴趣的话可以自行查阅；这个方法主要做了以下两件事：

1. 对发送者的身份和权限做出一定的校检
2. 把这个BroadcastReceiver以BroadcastFilter的形式存储在AMS的mReceiverResolver变量中，供后续使用。

就这样，被传递过来的BroadcastReceiver已经成功地注册在系统之中，能够接收特定类型的广播了；那么注册在AndroidManifest.xml中的静态广播是如何被系统感知的呢？

在 [插件加载机制](http://weishu.me/2016/04/05/understand-plugin-framework-classloader/) 中我们知道系统会通过PackageParser解析Apk中的AndroidManifest.xml文件，因此我们有理由认为，系统会在解析AndroidMafest.xml的<receiver>标签（也即静态注册的广播）的时候保存相应的信息；而Apk的解析过程是在PMS中进行的，因此**静态注册广播的信息存储在PMS中**。接下来的分析会证实这一结论。

### **发送和接收过程**

#### **发送过程**

发送广播很简单，就是一句context.sendBroadcast()，我们顺藤摸瓜，跟踪这个方法。前文也提到过，Context中方法的调用都会委托到ContextImpl这个类，我们直接看ContextImpl对这个方法的实现：

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

嗯，发送广播也是通过AMS进行的，我们直接查看ActivityManagerService类的broadcastIntent方法，这个方法仅仅是调用了broadcastIntentLocked方法，我们继续跟踪；broadcastIntentLocked这个方法相当长，处理了诸如粘性广播，顺序广播，各种Flag以及动态广播静态广播的接收过程，这些我们暂时不关心；值得注意的是，在这个方法中我们发现，其实**广播的发送和接收是融为一体的**。某个广播被发送之后，AMS会找出所有注册过的BroadcastReceiver中与这个广播匹配的接收者，然后将这个广播分发给相应的接收者处理。

#### **匹配过程**

某一条广播被发出之后，并不是阿猫阿狗都能接收它并处理的；BroadcastReceiver可能只对某些类型的广播感兴趣，因此它也只能接收和处理这种特定类型的广播；在broadcastIntentLocked方法内部有如下代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | // Figure out who all will receive this broadcast.  List receivers = null;  List<BroadcastFilter> registeredReceivers = null;  // Need to resolve the intent to interested receivers...  if ((intent.getFlags()&Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY)  == 0) {  receivers = collectReceiverComponents(intent, resolvedType, callingUid, users);  }  if (intent.getComponent() == null) {  if (userId == UserHandle.USER\_ALL && callingUid == Process.SHELL\_UID) {  // Query one target user at a time, excluding shell-restricted users  // 略  } else {  registeredReceivers = mReceiverResolver.queryIntent(intent,  resolvedType, false, userId);  }  } |

这里有两个列表receivers和registeredReceivers，看名字好像是广播接收者的列表；下面是它们的赋值过程：

|  |  |
| --- | --- |
| 1  2 | receivers = collectReceiverComponents(intent, resolvedType, callingUid, users);  registeredReceivers = mReceiverResolver.queryIntent(intent, resolvedType, false, userId); |

读者可以自行跟踪这两个方法的代码，过程比较简单，我这里直接给出结论：

1. receivers是对这个广播感兴趣的**静态BroadcastReceiver**列表；collectReceiverComponents 通过PackageManager获取了与这个广播匹配的静态BroadcastReceiver信息；这里也证实了我们在分析BroadcasrReceiver注册过程中的推论——静态BroadcastReceiver的注册过程的确实在PMS中进行的。
2. mReceiverResolver存储了**动态注册**的BroadcastReceiver的信息；还记得这个mReceiverResolver吗？我们在分析动态广播的注册过程中发现，动态注册的BroadcastReceiver的相关信息最终存储在此对象之中；在这里，通过mReceiverResolver对象匹配出了对应的BroadcastReceiver供进一步使用。

现在系统通过PMS拿到了所有符合要求的静态BroadcastReceiver，然后从AMS中获取了符合要求的动态BroadcastReceiver；因此接下来的工作非常简单：唤醒这些广播接受者。简单来说就是回调它们的onReceive方法。

#### **接收过程**

通过上文的分析过程我们知道，在AMS的broadcastIntentLocked方法中找出了符合要求的所有BroadcastReceiver；接下来就需要把这个广播分发到这些接收者之中。在broadcastIntentLocked方法的后半部分有如下代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | BroadcastQueue queue = broadcastQueueForIntent(intent);  BroadcastRecord r = new BroadcastRecord(queue, intent, callerApp,  callerPackage, callingPid, callingUid, resolvedType,  requiredPermissions, appOp, brOptions, receivers, resultTo, resultCode,  resultData, resultExtras, ordered, sticky, false, userId);  boolean replaced = replacePending && queue.replaceOrderedBroadcastLocked(r);  if (!replaced) {  queue.enqueueOrderedBroadcastLocked(r);  queue.scheduleBroadcastsLocked();  } |

首先创建了一个BroadcastRecord代表此次发送的这条广播，然后把它丢进一个队列，最后通过scheduleBroadcastsLocked通知队列对广播进行处理。

在BroadcastQueue中通过Handle调度了对于广播处理的消息，调度过程由processNextBroadcast方法完成，而这个方法通过performReceiveLocked最终调用了IIntentReceiver的performReceive方法。

这个IIntentReceiver正是在广播注册过程中由App进程提供给AMS进程的Binder对象，现在AMS通过这个Binder对象进行IPC调用通知广播接受者所在进程完成余下操作。在上文我们分析广播的注册过程中提到过，这个IItentReceiver的实现是LoadedApk.ReceiverDispatcher；我们查看这个对象的performReceive方法，源码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public void performReceive(Intent intent, int resultCode, String data,  Bundle extras, boolean ordered, boolean sticky, int sendingUser) {  Args args = new Args(intent, resultCode, data, extras, ordered,  sticky, sendingUser);  if (!mActivityThread.post(args)) {  if (mRegistered && ordered) {  IActivityManager mgr = ActivityManagerNative.getDefault();  args.sendFinished(mgr);  }  }  } |

这个方法创建了一个Args对象，然后把它post到了mActivityThread这个Handler中；我们查看Args类的run方法：(坚持一下，马上就分析完了 ^ ^)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | public void run() {  final BroadcastReceiver receiver = mReceiver;  final boolean ordered = mOrdered;  final IActivityManager mgr = ActivityManagerNative.getDefault();  final Intent intent = mCurIntent;  mCurIntent = null;    if (receiver == null || mForgotten) {  if (mRegistered && ordered) {  sendFinished(mgr);  }  return;  }  try {  ClassLoader cl = mReceiver.getClass().getClassLoader(); // Important!! load class  intent.setExtrasClassLoader(cl);  setExtrasClassLoader(cl);  receiver.setPendingResult(this);  receiver.onReceive(mContext, intent); // callback  } catch (Exception e) {  if (mRegistered && ordered) {  sendFinished(mgr);  }  if (mInstrumentation == null ||  !mInstrumentation.onException(mReceiver, e)) {  throw new RuntimeException(  "Error receiving broadcast " + intent  + " in " + mReceiver, e);  }  }    if (receiver.getPendingResult() != null) {  finish();  }  } |

这里，我们看到了相应BroadcastReceiver的onReceive回调；因此，广播的工作原理到这里就水落石出了；我们接下来将探讨如何实现对于广播的插件化。

## **思路分析**

上文中我们分析了BroadcastReceiver的工作原理，那么怎么才能实现对BroadcastReceiver的插件化呢？

从分析过程中我们发现，Framework对于静态广播和动态广播的处理是不同的；不过，这个不同之处仅仅体现在**注册过程**——静态广播需要在AndroidManifest.xml中注册，并且注册的信息存储在PMS中；动态广播不需要预注册，注册的信息存储在AMS中。

从实现Activity的插件化过程中我们知道，需要在AndroidManifest.xml中预先注册是一个相当麻烦的事情——我们需要使用『替身』并在合适的时候进行『偷梁换柱』；因此看起来动态广播的处理要容易那么一点，我们先讨论一下如何实现动态注册BroadcastReceiver的插件化。

首先，广播并没有复杂的生命周期，它的整个存活过程其实就是一个onReceive回调；而动态广播又不需要在AndroidManifest.xml中预先注册，所以动态注册的BroadcastReceiver其实可以当作一个普通的Java对象；我们完全可以用纯ClassLoader技术实现它——不就是把插件中的Receiver加载进来，然后想办法让它能接受onReceive回调嘛。

静态BroadcastReceiver看起来要复杂一些，但是我们连Activity都搞定了，还有什么难得到我们呢？对于实现静态BroadcastReceiver插件化的问题，有的童鞋或许会想，我们可以借鉴Activity的工作方式——用替身和Hook解决。但是很遗憾，这样是行不通的。为什么呢？

BroadcastReceiver有一个IntentFilter的概念，也就是说，每一个BroadcastReceiver只对特定的Broadcast感兴趣；而且，AMS在进行广播分发的时候，也会对这些BroadcastReceiver与发出的广播进行匹配，只有Intent匹配的Receiver才能收到广播；在分析源码的时候也提到了这个匹配过程。如果我们尝试用替身Receiver解决静态注册的问题，那么它的IntentFilter该写什么？我们无法预料插件中静态注册的Receiver会使用什么类型的IntentFilter，就算我们在AndroidManifest.xml中声明替身也没有用——我们压根儿收不到与我们的IntentFilter不匹配的广播。其实，我们对于Activity的处理方式也有这个问题；如果你尝试用IntentFilter的方式启动Activity，这并不能成功；这算得上是DroidPlugin的缺陷之一。

那么，我们就真的对静态BroadcastReceiver无能为力吗？想一想这里的难点是什么？

没错，主要是在静态BroadcastReceiver里面这个IntentFilter我们事先无法确定，它是动态变化的；但是，动态BroadcastReceiver不是可以动态添加IntentFilter吗！！！

**可以把静态广播当作动态广播处理**

既然都是广播，它们的功能都是订阅一个特定的消息然后执行某个特定的操作，我们完全可以把插件中的静态广播全部注册为动态广播，这样就解决了静态广播的问题。当然，这样也是有缺陷的，静态BroadcastReceiver与动态BroadcastReceiver一个非常大的不同之处在于：动态BroadcastReceiver在进程死亡之后是无法接收广播的，而静态BroadcastReceiver则可以——系统会唤醒Receiver所在进程；这算得上缺陷之二，当然，瑕不掩瑜。

## **静态广播非静态的实现**

上文我们提到，可以把静态BroadcastReceiver当作动态BroadcastReceiver处理；我们接下来实现这个过程。

### **解析**

要把插件中的静态BroadcastReceiver当作动态BroadcastReceiver处理，我们首先得知道插件中到底注册了哪些广播；这个过程归根结底就是获取AndroidManifest.xml中的<receiver>标签下面的内容，我们可以选择手动解析xml文件；这里我们选择使用系统的 PackageParser 帮助解析，这种方式在之前的 [插件加载过程][] 中也用到过，如果忘记了可以温习一下。

PackageParser中有一系列方法用来提取Apk中的信息，可是翻遍了这个类也没有找到与「Receiver」名字相关的方法；最终我们发现BroadcastReceiver信息是用与Activity相同的类存储的！这一点可以在PackageParser的内部类Package中发现端倪——成员变量receivers和activities的范型类型相同。所以，我们要解析apk的<receiver>的信息，可以使用PackageParser的generateActivityInfo方法。

知道这一点之后，代码就比较简单了；使用反射调用相应的隐藏接口，并且在必要的时候构造相应参数的方式我们在插件化系列文章中已经讲述过很多，相信读者已经熟练，这里就不赘述，直接贴代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | private static void parserReceivers(File apkFile) throws Exception {  Class<?> packageParserClass = Class.forName("android.content.pm.PackageParser");  Method parsePackageMethod = packageParserClass.getDeclaredMethod("parsePackage", File.class, int.class);  Object packageParser = packageParserClass.newInstance();  // 首先调用parsePackage获取到apk对象对应的Package对象  Object packageObj = parsePackageMethod.invoke(packageParser, apkFile, PackageManager.GET\_RECEIVERS);  // 读取Package对象里面的receivers字段,注意这是一个 List<Activity> (没错,底层把<receiver>当作<activity>处理)  // 接下来要做的就是根据这个List<Activity> 获取到Receiver对应的 ActivityInfo (依然是把receiver信息用activity处理了)  Field receiversField = packageObj.getClass().getDeclaredField("receivers");  List receivers = (List) receiversField.get(packageObj);  // 调用generateActivityInfo 方法, 把PackageParser.Activity 转换成  Class<?> packageParser$ActivityClass = Class.forName("android.content.pm.PackageParser$Activity");  Class<?> packageUserStateClass = Class.forName("android.content.pm.PackageUserState");  Class<?> userHandler = Class.forName("android.os.UserHandle");  Method getCallingUserIdMethod = userHandler.getDeclaredMethod("getCallingUserId");  int userId = (Integer) getCallingUserIdMethod.invoke(null);  Object defaultUserState = packageUserStateClass.newInstance();  Class<?> componentClass = Class.forName("android.content.pm.PackageParser$Component");  Field intentsField = componentClass.getDeclaredField("intents");  // 需要调用 android.content.pm.PackageParser#generateActivityInfo(android.content.pm.ActivityInfo, int, android.content.pm.PackageUserState, int)  Method generateReceiverInfo = packageParserClass.getDeclaredMethod("generateActivityInfo",  packageParser$ActivityClass, int.class, packageUserStateClass, int.class);  // 解析出 receiver以及对应的 intentFilter  for (Object receiver : receivers) {  ActivityInfo info = (ActivityInfo) generateReceiverInfo.invoke(packageParser, receiver, 0, defaultUserState, userId);  List<? extends IntentFilter> filters = (List<? extends IntentFilter>) intentsField.get(receiver);  sCache.put(info, filters);  }  } |

### **注册**

我们已经解析得到了插件中静态注册的BroadcastReceiver的信息，现在我们只需要把这些静态广播动态注册一遍就可以了；但是，由于BroadcastReceiver的实现类存在于插件之后，我们需要手动用ClassLoader来加载它；这一点在 [插件加载机制](http://weishu.me/2016/04/05/understand-plugin-framework-classloader/) 已有讲述，不啰嗦了。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | ClassLoader cl = null;  for (ActivityInfo activityInfo : ReceiverHelper.sCache.keySet()) {  Log.i(TAG, "preload receiver:" + activityInfo.name);  List<? extends IntentFilter> intentFilters = ReceiverHelper.sCache.get(activityInfo);  if (cl == null) {  cl = CustomClassLoader.getPluginClassLoader(apk, activityInfo.packageName);  }  // 把解析出来的每一个静态Receiver都注册为动态的  for (IntentFilter intentFilter : intentFilters) {  BroadcastReceiver receiver = (BroadcastReceiver) cl.loadClass(activityInfo.name).newInstance();  context.registerReceiver(receiver, intentFilter);  }  } |

就这样，我们对插件静态BroadcastReceiver的支持已经完成了，是不是相当简单？至于插件中的动态广播如何实现插件化，这一点**交给读者自行完成**，希望你在解决这个问题的过程中能够加深对于插件方案的理解 ^ ^

## **小节**

本文我们介绍了BroadcastReceiver组件的插件化方式，可以看到，插件方案对于BroadcastReceiver的处理相对简单；同时「静态广播非静态」的特性以及BroadcastReceiver先天的一些特点导致插件方案没有办法做到尽善尽美，不过这都是大醇小疵——在绝大多数情况下，这样的处理方式是可以满足需求的。

虽然对于BroadcastReceiver的处理方式相对简单，但是文章的内容却并不短——我们花了大量的篇幅讲述BroadcastReceiver的原理，这也是我的初衷：借助DroidPlugin更深入地了解Android Framework。

接下来为文章会讲述四大组件中的另外两个——Service和ContentProvider的插件化方案；喜欢就点个赞吧～持续更新，请关注github项目 [understand-plugin-framework](http://weishu.me/2016/04/05/understand-plugin-framework-classloader/) 和我的 [博客](http://weishu.me/) ! 如果你觉得能从文中学到皮毛，还请支持一下 :)