前言

Broadcast Receiver 的容器内实现跟 Service 和 Activity 一样，都是需要提前在 Manifest 中声明才能够调用的，因此对于 VirtualApp 这种容器，就需要提供一种能够在容器内实现的操作。

对于写在 Manifest 中注册的 Broadcast Receiver 在应用启动时会静态注册，而这条途径现在不行了，因此我们需要考虑在容器内应用启动时**动态注册**它的那些 Broadcast Receiver。

Broadcast Receiver 容器内实现

Receiver 实现

首先我们来看其创建的时机：

```
public BActivityManagerService() {
  mBroadcastManager = BroadcastManager.startSystem(this, mPms);
}
```

VirtualApp 在创建 `BActivityManagerService` 时会附带着把 `BroadcastManager` 一起创建：

```
public static BroadcastManager startSystem(
    BActivityManagerService ams, BPackageManagerService pms) {
  if (sBroadcastManager == null) {
    synchronized (BroadcastManager.class) {
      if (sBroadcastManager == null) {
        sBroadcastManager = new BroadcastManager(ams, pms);
      }
    }
  }
  return sBroadcastManager;
}
```

然后在 `BActivityManagerService` 调用 `systemReady` 的时候会触发 `BroadcastManager.startup` ：

```
public void systemReady() {
  mBroadcastManager.startup();
}
public void startup() {
  mPms.addPackageMonitor(this);
  List<BPackageSettings> bPackageSettings = mPms.getBPackageSettings();
```

```
  for (BPackageSettings bPackageSetting : bPackageSettings) {
    BPackage bPackage = bPackageSetting.pkg;
    registerPackage(bPackage);
  }
}
```

这里在遍历整个 VirtualApp 已经安装过的所有应用，然后对每个包都调用一次 `registerPackage` ：

```
@SuppressLint("NewApi")
private void registerPackage(BPackage bPackage) {
  synchronized (mReceivers) {
    Slog.d(TAG, "register: " + bPackage.packageName  + ", size: " +
bPackage.receivers.size());
    for (BPackage.Activity receiver : bPackage.receivers)  {
      List<BPackage.ActivityIntentInfo> intents = receiver.intents;
      for (BPackage.ActivityIntentInfo intent : intents) {
        // 新增功能1：类型转换获取IntentFilter
        BPackage.IntentInfo intentInfo = (BPackage.IntentInfo) intent;
        IntentFilter intentFilter = intentInfo.intentFilter;
        // 新增功能2：黑名单Action处理
        if (intentFilter != null) {
          int countActions = intentFilter.countActions();
          for (int i = 0; i < countActions; i++) {
            String action = intentFilter.getAction(i);
            if (isBlackAction(action)) {
              intentFilter.addAction(proxySystemAction(action));
            }
          }
        }
        ProxyBroadcastReceiver proxyBroadcastReceiver = new
ProxyBroadcastReceiver();
        // 新增功能3：系统版本判断逻辑
        if (BuildCompat.isT())  {
          SandBoxCore.getContext().registerReceiver(proxyBroadcastReceiver,
intentFilter, 2);
        } else {
          SandBoxCore.getContext().registerReceiver(proxyBroadcastReceiver,
intentFilter);
        }

        addReceiver(bPackage.packageName,  proxyBroadcastReceiver);
      }
    }
```

```
    }
}
```

首先获取包里所有需要注册的 receivers，然后往下遍历去判断 action 是否为 BlackAction：

```
public static boolean isBlackAction(String str) {
    return SYSTEM_ACTIONS.contains(str);
}
```

这里的 `SYSTEM_ACTIONS` 包括了这些：

```
public static final String ACTION_BOOT_COMPLETED =
"android.intent.action.BOOT_COMPLETED";
public static final String ACTION_CHECKIN_NOW =
"com.google.android.gms.permission.CHECKIN_NOW";
public static final String ACTION_CHECKIN_NOW_SERVER =
"android.server.checkin.CHECKIN_NOW";
public static final String ACTION_MODE_CHANGED =
"android.location.MODE_CHANGED";
public static final String ACTION_PROVIDERS_CHANGED =
"android.location.PROVIDERS_CHANGED";
public static final String ACTION_SIM_STATE_CHANGED =
"android.intent.action.SIM_STATE_CHANGED";
static {
    SYSTEM_ACTIONS.add("android.accounts.LOGIN_ACCOUNTS_CHANGED");
    SYSTEM_ACTIONS.add(ACTION_BOOT_COMPLETED);
    SYSTEM_ACTIONS.add(ACTION_CHECKIN_NOW);
    SYSTEM_ACTIONS.add(ACTION_CHECKIN_NOW_SERVER);
    SYSTEM_ACTIONS.add(ACTION_MODE_CHANGED);
    SYSTEM_ACTIONS.add(ACTION_PROVIDERS_CHANGED);
    SYSTEM_ACTIONS.add(ACTION_SIM_STATE_CHANGED);
}
```

如果属于上述的这些 action，那就需要额外加一层代理把它们替换掉：

```
public static String proxySystemAction(String str) {
    return "fake.black." + str;
}
```

最后再调用 `registerReceiver` 动态注册这些：

```
if (BuildCompat.isT())  {
    SandBoxCore.getContext().registerReceiver(proxyBroadcastReceiver,
```

```
intentFilter, 2);
} else {
  SandBoxCore.getContext().registerReceiver(proxyBroadcastReceiver,
intentFilter);
}
```

最后调用 `addReceiver` 把这个包的那些 receiver 放入 Hash 表里做个映射就算是注册完成了：

```
private void addReceiver(String packageName, BroadcastReceiver receiver) {
  List<BroadcastReceiver> broadcastReceivers = mReceivers.get(packageName);
  if (broadcastReceivers == null) {
    broadcastReceivers = new ArrayList<>();
    mReceivers.put(packageName, broadcastReceivers);
  }
  broadcastReceivers.add(receiver);
}
```

看起来好像一下子就结束了？总结一下流程就是：把容器内的那些包的每个 Receiver 都拿出来，然后把它们的 IntentFilter 用 VitualApp 自己注册一个一样的，这样就能让 VitualApp 来代管这些收到的 Message 了。

那么问题来了，VitualApp 代管了这些 Message 谁来处理呢？注意到当时注册的时候用到了一个 `ProxyBroadcastReceiver` 对象。

此处用到的 `ProxyBroadcastReceiver` 重载了 `onReceive` ：

```
@Override
public void onReceive(Context context, Intent intent) {
  intent.setExtrasClassLoader(context.getClassLoader());
  ProxyBroadcastRecord record = ProxyBroadcastRecord.create(intent);
  if (record.mIntent == null) {
    return;
  }
  PendingResult pendingResult = goAsync();
  try {
    SandBoxCore.getBActivityManager()
        .scheduleBroadcastReceiver(
            record.mIntent, new PendingResultData(pendingResult),
record.mUserId);
  } catch (RemoteException e) {
    pendingResult.finish();
  }
}
```

可以看出，当 VirtualApp 收到 Message 时会往下调用 `scheduleBroadcastReceiver` ：

```
@Override
public void scheduleBroadcastReceiver(
    Intent intent, PendingResultData pendingResultData, int userId) throws
RemoteException {
    // 提取 ResolveInfo
  List<ResolveInfo> resolves =
      BPackageManagerService.get().queryBroadcastReceivers(intent,
GET_META_DATA, null, userId);

  if (resolves.isEmpty()) {
    pendingResultData.build().finish();
    Slog.d(TAG, "scheduleBroadcastReceiver empty");
    return;
  }
  mBroadcastManager.sendBroadcast(pendingResultData);
  for (ResolveInfo resolve : resolves) {
    ProcessRecord processRecord =
        BProcessManagerService.get()
            .findProcessRecord(
                resolve.activityInfo.packageName,
resolve.activityInfo.processName, userId);
    if (processRecord != null) {
      ReceiverData data = new ReceiverData();
      data.intent = intent;
      data.activityInfo = resolve.activityInfo;
      data.data = pendingResultData;
      // 调度 ReceiverData
      processRecord.bActivityThread.scheduleReceiver(data);
    }
  }
}
```

功能也很明显，根据传来的 `Intent` 去找到对应应用的 `ProcessRecord` 对象，并通过 `scheduleReceiver` 来向它们传递 `Intent` ：

```
@Override
public void scheduleReceiver(ReceiverData data) throws RemoteException {
  if (!isInit()) {
    bindApplication();
  }
  mH.post(
      () -> {
        BroadcastReceiver mReceiver = null;
```

```
        Intent intent = data.intent;
        ActivityInfo activityInfo = data.activityInfo;
        BroadcastReceiver.PendingResult pendingResult = data.data.build();

        try {
          Context baseContext = mInitialApplication.getBaseContext();
          ClassLoader classLoader = baseContext.getClassLoader();
          intent.setExtrasClassLoader(classLoader);

          mReceiver = (BroadcastReceiver)
classLoader.loadClass(activityInfo.name).newInstance();
          BRBroadcastReceiver.get(mReceiver).setPendingResult(pendingResult);
          mReceiver.onReceive(baseContext, intent);
          BroadcastReceiver.PendingResult finish =
              BRBroadcastReceiver.get(mReceiver).getPendingResult();
          if (finish != null) {
            finish.finish();
          }
          SandBoxCore.getBActivityManager().finishBroadcast(data.data);
        } catch (Throwable throwable) {
          throwable.printStackTrace();
          Slog.e(TAG, "Error receiving broadcast " + intent + " in " +
mReceiver);
        }
      });
}
```

这里就很明显了，通过上述的信息来找到目标应用中的处理函数，通过 `loadClass` 来加载对应的代码，最后调用它的 `onReceive` 来让真正的处理函数处理这个消息。

代码过程中一直有个 `pendingResultData` 在通过 `sendBroadcast` 发送：

```
mBroadcastManager.sendBroadcast(pendingResultData);
```

主要是提供一个超时兜底，当广播超时时候会通知一个 PendingResult 表示结束，告诉发送方广播结束了。

Sender 的实现

除了接受部分需要这样适配，由容器内应用发送广播的过程同样也需要做些调整。

```
@ProxyMethod("broadcastIntent")
public static class BroadcastIntent extends MethodHook {
  @Override
```

```
  protected Object hook(Object who, Method method, Object[] args) throws
Throwable {
    int intentIndex = getIntentIndex(args);
    Intent intent = (Intent) args[intentIndex];
    String resolvedType = (String) args[intentIndex + 1];
    Intent proxyIntent =
        SandBoxCore.getBActivityManager()
            .sendBroadcast(intent, resolvedType, BActivityThread.getUserId());
    if (proxyIntent != null) {

proxyIntent.setExtrasClassLoader(AppInstrumentation.get().getDelegateAppClassL
oader());
      ProxyBroadcastRecord.saveStub(proxyIntent, intent,
BActivityThread.getUserId());
      args[intentIndex] = proxyIntent;
    }
    // ignore permission
    for (int i = 0; i < args.length; i++) {
      Object o = args[i];
      if (o instanceof String[]) {
        args[i] = null;
      }
    }
    return method.invoke(who, args);
  }

  int getIntentIndex(Object[] args) {
    for (int i = 0; i < args.length; i++) {
      Object arg = args[i];
      if (arg instanceof Intent) {
        return i;
      }
    }
    return 1;
  }
}
```

首先是通过 sendBroadcast 把真正需要发送的 Intent 包装起来：

```
@Override
public Intent sendBroadcast(Intent intent, String resolvedType, int userId)
    throws RemoteException {
  List<ResolveInfo> resolves =
      BPackageManagerService.get()
          .queryBroadcastReceivers(intent, GET_META_DATA, resolvedType,
```

```
userId);

  for (ResolveInfo resolve : resolves) {
    ProcessRecord processRecord =
        BProcessManagerService.get()
            .findProcessRecord(
                resolve.activityInfo.packageName,
resolve.activityInfo.processName, userId);
    if (processRecord == null) {
      continue;
    }
    try {
      processRecord.bActivityThread.bindApplication();
    } catch (RemoteException e) {
      e.printStackTrace();
    }
  }
  Intent shadow = new Intent();
  shadow.setPackage(SandBoxCore.getHostPkg());
  shadow.setComponent(null);
  shadow.setAction(intent.getAction());
  return shadow;
}
```

将内部应用发送的 `Intent` 伪造成由 VirtualApp 发送的 `proxyIntent` ，然后再调用原生的发送函数把这个广播重新播出去。

不过如果这个发出去的广播最终还是由容器内的应用去处理，那之前注册好的那些 `Receiver` 就会接收到这些消息然后处理了。后续流程就跟前文一致了。

参考文章

https://blog.csdn.net/ganyao939543405/article/details/76229480