

AMS

ActivityManagerService是Android系统中一个特别重要的系统服务，也是我们上层APP打交道最多的系统服务之一。ActivityManagerService（以下简称AMS）主要负责四大组件的启动、切换、调度以及应用进程的管理和调度工作。所有的APP应用都需要与AMS打交道

Activity Manager的组成主要分为以下几个部分：

- 1.服务代理：由ActivityManagerProxy实现，用于与Server端提供的系统服务进行进程间通信
- 2.服务中枢：ActivityManagerNative继承自Binder并实现IActivityManager，它提供了服务接口和Binder接口的相互转化功能，并在内部存储服务代理对象，并提供了getDefault方法返回服务代理
- 3.Client：由ActivityManager封装一部分服务接口供Client调用。ActivityManager内部通过调用ActivityManagerNative的getDefault方法，可以得到一个ActivityManagerProxy对象的引用，进而通过该代理对象调用远程服务的方法
- 4.Server:由ActivityManagerService实现，提供Server端的系统服务

ActivityManagerService的启动过程

AMS是在SystemServer中被添加的，所以先到SystemServer中查看初始化

```
1 public static void main(String[] args) {
2     new SystemServer().run();
3 }
```

```
1 private void run() {
2     ...
3     createSystemContext();
4     // Create the system service manager.
5     mSystemServiceManager = new
6     SystemServiceManager(mSystemContext);
7     mSystemServiceManager.setStartInfo(mRuntimeRestart,
8     mRuntimeStartElapsedTime, mRuntimeStartUptime);
9     LocalServices.addService(SystemServiceManager.class,
10    mSystemServiceManager);
11    // Prepare the thread pool for init tasks that can be
12    parallelized
13    SystemServerInitThreadPool.get();
14    } finally {
15        traceEnd(); // InitBeforeStartServices
16    }
17    // Start services.
18    try {
19        traceBeginAndSlog("StartServices");
20        startBootstrapServices();
21        startCoreServices();
22        startOtherServices();
23        SystemServerInitThreadPool.shutdown();
24    } catch (Throwable ex) {
25        throw ex;
26    } finally {
```

```

24         traceEnd();
25     }
26     ...
27     // Loop forever.
28     Looper.loop();
29     throw new RuntimeException("Main thread loop unexpectedly exited");
30 }

```

在SystemServer中，在startBootstrapServices()中去启动了AMS

```

1  private void startBootstrapServices() {
2      ...
3      // Activity manager runs the show.
4      traceBeginAndSlog("startActivityManager");
5      //启动了AMS
6      mActivityManagerService = mSystemServiceManager.startService(
7          ActivityManagerService.Lifecycle.class).getService();
8
9      mActivityManagerService.setSystemServiceManager(mSystemServiceManager);
10     mActivityManagerService.setInstaller(installer);
11     traceEnd();
12     ...
13     // Now that the power manager has been started, let the activity
14     manager
15     // initialize power management features.
16     traceBeginAndSlog("InitPowerManagement");
17     mActivityManagerService.initPowerManagement();
18     traceEnd();
19     // Set up the Application instance for the system process and get started.
20     traceBeginAndSlog("setSystemProcess");
21     mActivityManagerService.setSystemProcess();
22     traceEnd();
23 }

```

AMS是通过SystemServiceManager.startService去启动的，参数是ActivityManagerService.Lifecycle.class，首先看看startService方法

```

1  @SuppressWarnings("unchecked")
2  public SystemService startService(String className) {
3      final Class<SystemService> serviceClass;
4      try {
5          serviceClass = (Class<SystemService>)Class.forName(className);
6      } catch (ClassNotFoundException ex) {
7          slog.i(TAG, "Starting " + className);
8          throw new RuntimeException("Failed to create service " +
9              className
10                 + ": service class not found, usually indicates that the
11                 caller should "
12                 + "have called PackageManager.hasSystemFeature() to
13                 check whether the "
14                 + "feature is available on this device before trying to
15                 start the "
16                 + "services that implement it", ex);
17      }
18      return startService(serviceClass);
19 }

```

```
15     }
```

```
1  @SuppressWarnings("unchecked")
2  public <T extends SystemService> T startService(Class<T> serviceClass) {
3      try {
4          final String name = serviceClass.getName();
5          Slog.i(TAG, "Starting " + name);
6          Trace.traceBegin(Trace.TRACE_TAG_SYSTEM_SERVER, "StartService "
+ name);
7
8          // Create the service.
9          if (!SystemService.class.isAssignableFrom(serviceClass)) {
10             throw new RuntimeException("Failed to create " + name
11                 + ": service must extend " +
SystemService.class.getName());
12         }
13         final T service;
14         try {
15             Constructor<T> constructor =
serviceClass.getConstructor(Context.class);
16             service = constructor.newInstance(mContext);
17         } catch (InstantiationException ex) {
18             throw new RuntimeException("Failed to create service " +
name
19                 + ": service could not be instantiated", ex);
20         } catch (IllegalAccessException ex) {
21             throw new RuntimeException("Failed to create service " +
name
22                 + ": service must have a public constructor with a
Context argument", ex);
23         } catch (NoSuchMethodException ex) {
24             throw new RuntimeException("Failed to create service " +
name
25                 + ": service must have a public constructor with a
Context argument", ex);
26         } catch (InvocationTargetException ex) {
27             throw new RuntimeException("Failed to create service " +
name
28                 + ": service constructor threw an exception", ex);
29         }
30
31         startService(service);
32         return service;
33     } finally {
34         Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);
35     }
36 }
```

```

1 public void startService(@NonNull final SystemService service) {
2     // Register it.
3     mServices.add(service);
4     // Start it.
5     long time = SystemClock.elapsedRealtime();
6     try {
7         service.onStart();
8     } catch (RuntimeException ex) {
9         throw new RuntimeException("Failed to start service " +
service.getClass().getName()
10             + ": onStart threw an exception", ex);
11     }
12     warnIfTooLong(SystemClock.elapsedRealtime() - time, service,
"onStart");
13 }

```

startService方法很简单，是通过传进来的class然后反射创建对应的service服务。所以此处创建的是Lifecycle的实例，然后通过startService启动了AMS服务

那我们再去看看ActivityManagerService.Lifecycle这个类的构造方法

```

1 public static final class Lifecycle extends SystemService {
2     private final ActivityManagerService mService;
3
4     public Lifecycle(Context context) {
5         super(context);
6         mService = new ActivityManagerService(context);
7     }
8
9     @Override
10    public void onStart() {
11        mService.start();
12    }
13
14    @Override
15    public void onBootPhase(int phase) {
16        mService.mBootPhase = phase;
17        if (phase == PHASE_SYSTEM_SERVICES_READY) {
18            mService.mBatteryStatsService.systemServicesReady();
19            mService.mServices.systemServicesReady();
20        }
21    }
22
23    @Override
24    public void onCleanupUser(int userId) {
25        mService.mBatteryStatsService.onCleanupUser(userId);
26    }
27
28    public ActivityManagerService getService() {
29        return mService;
30    }
31 }

```

再来开课AMS初始化做了什么

```

1 // Note: This method is invoked on the main thread but may need to attach
  various
2 // handlers to other threads. So take care to be explicit about the
  loop.
3 public ActivityManagerService(Context systemContext) {
4     LockGuard.installLock(this, LockGuard.INDEX_ACTIVITY);
5     mInjector = new Injector();
6     mContext = systemContext; //赋值mContext
7
8     mFactoryTest = FactoryTest.getMode();
9     mSystemThread = ActivityThread.currentActivityThread(); //获取当前的
  ActivityThread
10     mUiContext = mSystemThread.getSystemUiContext(); //赋值mUiContext
11
12     Slog.i(TAG, "Memory class: " +
  ActivityManager.staticGetMemoryClass());
13
14     mPermissionReviewRequired = mContext.getResources().getBoolean(
15
  com.android.internal.R.bool.config_permissionReviewRequired);
16 //创建Handler线程，用来处理handler消息
17     mHandlerThread = new ServiceThread(TAG,
18         THREAD_PRIORITY_FOREGROUND, false /*allowIo*/);
19     mHandlerThread.start();
20     mHandler = new MainHandler(mHandlerThread.getLooper());
21     mUiHandler = mInjector.getUiHandler(this); //处理ui相关msg的Handler
22
23     mProcStartHandlerThread = new ServiceThread(TAG + ":procStart",
24         THREAD_PRIORITY_FOREGROUND, false /* allowIo */);
25     mProcStartHandlerThread.start();
26     mProcStartHandler = new
  Handler(mProcStartHandlerThread.getLooper());
27 //管理AMS的一些常量，厂商定制系统就可能修改此处
28     mConstants = new ActivityManagerConstants(this, mHandler);
29
30     /* static; one-time init here */
31     if (skillHandler == null) {
32         skillThread = new ServiceThread(TAG + ":kill",
33             THREAD_PRIORITY_BACKGROUND, true /* allowIo */);
34         skillThread.start();
35         skillHandler = new KillHandler(skillThread.getLooper());
36     }
37 //初始化管理前台、后台广播的队列，系统会优先遍历发送前台广播
38     mFgBroadcastQueue = new BroadcastQueue(this, mHandler,
39         "foreground", BROADCAST_FG_TIMEOUT, false);
40     mBgBroadcastQueue = new BroadcastQueue(this, mHandler,
41         "background", BROADCAST_BG_TIMEOUT, true);
42     mBroadcastQueues[0] = mFgBroadcastQueue;
43     mBroadcastQueues[1] = mBgBroadcastQueue;
44 //初始化管理Service的 ActiveServices对象
45     mServices = new ActiveServices(this);
46     mProviderMap = new ProviderMap(this); //初始化Provider的管理者
47     mAppErrors = new AppErrors(mUiContext, this); //初始化APP错误日志的打印
  器
48 //创建电池统计服务，并输出到指定目录
49     File dataDir = Environment.getDataDirectory();
50     File systemDir = new File(dataDir, "system");
51     systemDir.mkdirs();

```

```

52
53     mAppWarnings = new AppWarnings(this, mUiContext, mHandler,
mUiHandler, systemDir);
54
55     // TODO: Move creation of battery stats service outside of activity
manager service.
56     mBatteryStatsService = new BatteryStatsService(systemContext,
systemDir, mHandler);
57     mBatteryStatsService.getActiveStatistics().readLocked();
58     mBatteryStatsService.scheduleWriteToDisk();
59     mOnBattery = DEBUG_POWER ? true
60     //创建进程统计分析服务, 追踪统计哪些进程有滥用或不良行为 :
mBatteryStatsService.getActiveStatistics().getIsOnBattery();
61     mBatteryStatsService.getActiveStatistics().setCallback(this);
62
63     mProcessStats = new ProcessStatsService(this, new File(systemDir,
"procstats"));
64
65     mAppOpsService = mInjector.getAppOpsService(new File(systemDir,
"appops.xml"), mHandler);
66     //加载Uri的授权文件
67     mGrantFile = new AtomicFile(new File(systemDir, "urigrants.xml"),
"uri-grants");
68     //负责管理多用户
69     mUserController = new UserController(this);
70     //vr功能的控制器
71     mVrController = new VrController(this);
72     //初始化OpenGL版本号
73     GL_ES_VERSION = SystemProperties.getInt("ro.opengles.version",
ConfigurationInfo.GL_ES_VERSION_UNDEFINED);
74
75
76     if (SystemProperties.getInt("sys.use_fifo_ui", 0) != 0) {
77         mUseFifoUIScheduling = true;
78     }
79
80     mTrackingAssociations =
"1".equals(SystemProperties.get("debug.track-associations"));
81     mTempConfig.setToDefaults();
82     mTempConfig.setLocales(LocaleList.getDefault());
83     mConfigurationSeq = mTempConfig.seq = 1;
84     //管理ActivityStack的重要类, 这里面记录着activity状态信息, 是AMS中的核
心类
85     mStackSupervisor = createStackSupervisor();
86     mStackSupervisor.onConfigurationChanged(mTempConfig);
87     //根据当前可见的Activity类型, 控制keyguard遮挡, 关闭和转换。 Keyguard就是我
们的锁屏相关页面
88     mKeyguardController = mStackSupervisor.getKeyguardController();
89     管理APK的兼容性配置
90     解析/data/system/packages-compat.xml文件, 该文件用于存储那些需要考虑屏幕尺寸的
APK信息,
91     mCompatModePackages = new CompatModePackages(this, systemDir,
mHandler);
92     //Intent防火墙, Google定义了一组规则, 来过滤intent, 如果触发了, 则intent会
被系统丢弃, 且不会告知发送者
93     mIntentFirewall = new IntentFirewall(new IntentFirewallInterface(),
mHandler);
94     mTaskChangeNotificationController =

```

```

95         new TaskChangeNotificationController(this,
mStackSupervisor, mHandler);
96         //这是activity启动的处理类，这里管理者activity启动中用到的intent信息和flag
标识，也和stack和task有重要的联系
97         mActivityStartController = new ActivityStartController(this);
98         mRecentTasks = createRecentTasks();
99         mStackSupervisor.setRecentTasks(mRecentTasks);
100        mLockTaskController = new LockTaskController(mContext,
mStackSupervisor, mHandler);
101        mLifecycleManager = new ClientLifecycleManager();
102        //启动一个线程专门跟进cpu当前状态信息，AMS对当前cpu状态了如指掌，可以更加高效的安排
其他工作
103        mProcessCpuThread = new Thread("CpuTracker") {
104            @Override
105            public void run() {
106                synchronized (mProcessCpuTracker) {
107                    mProcessCpuInitLatch.countDown();
108                    mProcessCpuTracker.init();
109                }
110                while (true) {
111                    try {
112                        try {
113                            synchronized(this) {
114                                final long now =
SystemClock.uptimeMillis();
115                                long nextCpuDelay =
(mLastCpuTime.get()+MONITOR_CPU_MAX_TIME)-now;
116                                long nextWriteDelay =
(mLastWriteTime+BATTERY_STATS_TIME)-now;
117                                //Slog.i(TAG, "Cpu delay=" + nextCpuDelay
118                                //          + ", write delay=" +
nextWriteDelay);
119                                if (nextWriteDelay < nextCpuDelay) {
120                                    nextCpuDelay = nextWriteDelay;
121                                }
122                                if (nextCpuDelay > 0) {
123                                    mProcessCpuMutexFree.set(true);
124                                    this.wait(nextCpuDelay);
125                                }
126                            }
127                        } catch (InterruptedException e) {
128                        }
129                        updateCpuStatsNow();
130                    } catch (Exception e) {
131                        Slog.e(TAG, "Unexpected exception collecting
process stats", e);
132                    }
133                }
134            }
135        };
136
137        mHiddenApiBlacklist = new HiddenApiSettings(mHandler, mContext);
138        //看门狗，监听进程。这个类每分钟调用一次监视器。 如果进程没有任何返回就杀掉
139        watchdog.getInstance().addMonitor(this);
140        watchdog.getInstance().addThread(mHandler);
141
142        // bind background thread to little cores

```

```

143         // this is expected to fail inside of framework tests because apps
        can't touch cpusets directly
144         // make sure we've already adjusted system_server's internal view
        of itself first
145         updateOomAdjLocked();
146         try {
147
148             Process.setThreadGroupAndCpuset(BackgroundThread.get().getThreadId(),
                Process.THREAD_GROUP_BG_NONINTERACTIVE);
149         } catch (Exception e) {
150             Slog.w(TAG, "Setting background thread cpuset failed");
151         }
152
153     }

```

```

1  private void start() {
2      removeAllProcessGroups();
3      mProcessCpuThread.start();
4
5      mBatteryStatsService.publish();
6      mAppOpsService.publish(mContext);
7      Slog.d("AppOps", "AppOpsService published");
8      LocalServices.addService(ActivityManagerInternal.class, new
LocalService());
9      // wait for the synchronized block started in mProcessCpuThread,
10     // so that any other access to mProcessCpuTracker from main thread
11     // will be blocked during mProcessCpuTracker initialization.
12     //等待mProcessCpuThread完成初始化后， 释放锁，初始化期间禁止访问
13     try {
14         mProcessCpuInitLatch.await();
15     } catch (InterruptedException e) {
16         Slog.wtf(TAG, "Interrupted wait during start", e);
17         Thread.currentThread().interrupt();
18         throw new IllegalStateException("Interrupted wait during
start");
19     }
20 }

```

然后来看看setSystemProcess 干了什么事情

```

1  public void setSystemProcess() {
2      try {
3          ServiceManager.addService(Context.ACTIVITY_SERVICE, this, /*
allowIsolated= */ true,
4              DUMP_FLAG_PRIORITY_CRITICAL | DUMP_FLAG_PRIORITY_NORMAL
| DUMP_FLAG_PROTO);
5          ServiceManager.addService(ProcessStats.SERVICE_NAME,
mProcessStats);
6          ServiceManager.addService("meminfo", new MemBinder(this), /*
allowIsolated= */ false,
7              DUMP_FLAG_PRIORITY_HIGH);
8          ServiceManager.addService("gfxinfo", new GraphicsBinder(this));
9          ServiceManager.addService("dbinfo", new DbBinder(this));
10         if (MONITOR_CPU_USAGE) {
11             ServiceManager.addService("cpuinfo", new CpuBinder(this),

```



```

12         /* allowIsolated= */ false,
DUMP_FLAG_PRIORITY_CRITICAL);
13     }
14     ServiceManager.addService("permission", new
PermissionController(this));
15     ServiceManager.addService("processinfo", new
ProcessInfoService(this));
16
17     ApplicationInfo info =
mContext.getPackageManager().getApplicationInfo(
18         "android", STOCK_PM_FLAGS | MATCH_SYSTEM_ONLY);
19     mSystemThread.installSystemApplicationInfo(info,
getClass().getClassLoader());
20
21     synchronized (this) {
22         ProcessRecord app = newProcessRecordLocked(info,
info.processName, false, 0);
23         app.persistent = true;
24         app.pid = MY_PID;
25         app.maxAdj = ProcessList.SYSTEM_ADJ;
26         app.makeActive(mSystemThread.getApplicationThread(),
mProcessStats);
27         synchronized (mPidsSelfLocked) {
28             mPidsSelfLocked.put(app.pid, app);
29         }
30         updateLruProcessLocked(app, false, null);
31         updateOomAdjLocked();
32     }
33     } catch (PackageManager.NameNotFoundException e) {
34         throw new RuntimeException(
35             "Unable to find android system package", e);
36     }
37
38     // Start watching app ops after we and the package manager are up
and running.
39     mAppOpsService.startWatchingMode(AppOpsManager.OP_RUN_IN_BACKGROUND,
null,
40         new IAppOpsCallback.Stub() {
41             @Override public void opChanged(int op, int uid, String
packageName) {
42                 if (op == AppOpsManager.OP_RUN_IN_BACKGROUND &&
packageName != null) {
43                     if (mAppOpsService.checkOperation(op, uid,
packageName)
44                         != AppOpsManager.MODE_ALLOWED) {
45                         runInBackgroundDisabled(uid);
46                     }
47                 }
48             }
49         });
50 }

```

注册服务。首先将ActivityManagerService注册到ServiceManager中，其次将几个与系统性能调试相关的服务注册到ServiceManager。

- 查询并处理ApplicationInfo。首先调用PackageManagerService的接口，查询包名为android的应用程序的ApplicationInfo信息，对应于framework-res.apk。然后以该信息为参数调用ActivityThread上的installSystemApplicationInfo方法。

- 创建并处理ProcessRecord。调用ActivityManagerService上的newProcessRecordLocked，创建一个ProcessRecord类型的对象，并保存该对象的信息

AMS是什么？

1. 从java角度来看，ams就是一个java对象，实现了Ibinder接口，所以它是一个用于进程之间通信的接口，这个对象初始化是在systemServer.java 的run()方法里面

```
1 public Lifecycle(Context context) {  
2     super(context);  
3     mService = new ActivityManagerService(context);  
4 }
```

2. AMS是一个服务

ActivityManagerService从名字就可以看出，它是一个服务，用来管理Activity，而且是一个系统服务，就是包管理服务，电池管理服务，震动管理等。

3. AMS是一个Binder

ams实现了Ibinder接口，所以它是一个Binder，这意味着他不但可以用于进程间通信，还是一个线程，因为一个Binder就是一个线程。

如果我们启动一个hello World安卓用于程序，里面不另外启动其他线程，这个里面最少要启动4个线程

- 1 main线程，只是程序的主线程，也是日常用到的最多的线程，也叫UI线程，因为android的组件是非线程安全的，所以只允许UI/MAIN线程来操作。

- 2 GC线程，java有垃圾回收机制，每个java程序都有一个专门负责垃圾回收的线程，

- 3 Binder1 就是我们的ApplicationThread，这个类实现了Ibinder接口，用于进程之间通信，具体来说，就是我们程序和AMS通信的工具

- 4 Binder2 就是我们的ViewRoot.W对象，他也是实现了IBinder接口，就是用于我们的应用程序和wms通信的工具。

wms就是WindowManagerServicer，和ams差不多的概念，不过他是管理窗口的系统服务。

```
1 public class ActivityManagerService extends IActivityManager.Stub  
2     implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {}
```

AMS相关重要类介绍

ProcessRecord 数据结构

第一类数据：描述身份的数据

- 1.ApplicationInfo info: AndroidManifest.xml中定义的Application信息
- 2.boolean isolated: 是不是isolated进程
- 3.int uid: 进程uid
- 4.int userId: 这个是android做的多用户系统id, 就像windows可以登录很多用户一样, android也希望可以实现类似的多用户
- 5.String processName: 进程名字, 默认情况下是包名
- 6.UidRecord uidRecord: 记录已经使用的uid
- 7.IApplicationThread thread: 这个很重要, 它是ApplicationThread的客户端, AMS就是通过这个对象给apk进程发送异步消息的(管理四大组件的消息), 所以只有这个对象不为空的情况下, 才代表apk进程可是使用了
- 8.int pid: 进程的pid
- 9.String procStatFile: proc目录下每一个进程都有一个以pid命名的目录文件, 这个目录下记载着进程的详细信息, 这个目录及目录下的文件是内核创建的, proc是内核文件系统, proc就是process的缩写, 涉及的目的就是导出进程内核信息
- 10.int[] gids: gid组
- 11.CompatibilityInfo compat: 兼容性信息
- 12.String requiredAbi: abi信息
- 13.String instructionSet: 指令集信息

第二类数据：描述进程中组件的数据

- 1.pkgList: 进程中运行的包
 - 2.ArraySet pkgDeps: 进程运行依赖的包
 - 3.ArrayList activities: 进程启动的所有的activity组件记录表
 - 4.ArraySet services: 进程启动的所有的service组件记录表
 - 5.ArraySet executingServices: 正在运行(executing)是怎么定义的? 首先需要明确的是系统是怎么控制组件的? 发送消息给apk进程, apk进程处理消息, 上报消息完成, 这被定义为一个完整的执行过程, 因此正在执行(executing)被定义为发送消息到上报完成这段时间
 - 6.ArraySet connections: 绑定service的客户端记录表
 - 7.ArraySet receivers: 广播接收器的记录表
 - 8.ContentProviderRecord pubProviders: pub是publish(发布)的意思, ContentProvider需要安装然后把自己发布到系统(AMS)中后, 才能使用, 安装指的是apk进程加载ContentProvider子类、初始化、创建数据库等过程, 发布是将ContentProvider的binder客户端注册到AMS中
 - 9.ArrayList conProviders: 使用ContentProvider的客户端记录表
 - 10.BroadcastRecord curReceiver: 当前进程正在执行的广播
- 在本节中以上组件信息只是做一个简单的描述, 以后单独分析组件管理的时候在详细介绍

第三类数据：描述进程状态的数据

- 1.int maxAdj: 进程的adj上限(adjustment)
- 2.int curRawAdj: 当前正在计算的adj, 这个值有可能大于maxAdj
- 3.int setRawAdj: 上次计算的curRawAdj设置到lowmemorykiller系统后的adj
- 4.int curAdj: 当前正在计算的adj, 这是curRawAdj被maxAdj削平的值
- 5.int setAdj: 上次计算的curAdj设置到lowmemorykiller系统后的adj
- 6.int verifiedAdj: setAdj校验后的值
- 7.int curSchedGroup: 正在计算的调度组
- 8.int setSchedGroup: 保存上次计算的调度组
- 9.int curProcState: 正在计算的进程状态
- 10.int repProcState: 发送给apk进程的状态
- 11.int setProcState: 保存上次计算的进程状态

- 12.int pssProcState: pss进程状态
- 13.ProcessState baseProcessTracker: 进程状态监测器
- 14.int adjSeq: 计算adj的序列数
- 15.int lruSeq: lru序列数
- 16.IBinder forcingToForeground:强制将进程的状态设置为前台运行的IBinder, IBinder代表的是组件的ID, 这个是整个android系统唯一

第四类数据: 和pss相关的数据

我们先来普及一下一些名词:

VSS- Virtual Set Size 虚拟耗用内存 (包含共享库占用的内存)
 RSS- Resident Set Size 实际使用物理内存 (包含共享库占用的内存)
 PSS- Proportional Set Size 实际使用的物理内存 (比例分配共享库占用的内存)
 USS- Unique Set Size 进程独自占用的物理内存 (不包含共享库占用的内存)
 一般来说内存占用大小有如下规律: $VSS \geq RSS \geq PSS \geq USS$

- 1.long initialIdlePss: 初始化pss
- 2.long lastPss: 上次pss
- 3.long lastSwapPss: 上次SwapPss数据
- 4.long lastCachedPss: 上次CachedPss数据
- 5.long lastCachedSwapPss: 上次CachedSwapPss数据

第五类数据: 和时间相关的数据

- 1.long lastActivityTime: 上次使用时间
- 2.long lastPssTime: 上次计算pss的时间
- 3.long nextPssTime: 下次计算pss的时间
- 4.long lastStateTime: 上次设置进程状态的时间
- 5.long lastWakeTime: 持有wakelock的时长
- 6.long lastCpuTime: 上次计算占用cpu的时长
- 7.long curCpuTime: 当前最新占用cpu的时长
- 8.long lastRequestedGc: 上次发送gc命令给apk进程的时间
- 9.long lastLowMemory: 上次发送低内存消息给apk进程的时间
- 10.long lastProviderTime: 上次进程中ContentProvider被使用的时间
- 11.long interactionEventTime: 上次发送交互时间时间
- 12.long fgInteractionTime: 变成前台的时间

第六类数据: crash和anr相关的数据

- 1.IBinder.DeathRecipient deathRecipient: apk进程退出运行的话, 会触发这个对象的binderDied()方法, 来回收系统资源
- 2.boolean crashing: 进程已经crash
- 3.Dialog crashDialog: crash对话框
- 4.boolean forceCrashReport: 强制crash对话框显示
- 5.boolean notResponding: 是否处于anr状态
- 6.Dialog anrDialog: anr显示对话框
- 7 Runnable crashHandler: crash回调
- 8.ActivityManager.ProcessErrorStateInfo crashingReport:crash报告的进程状态
- 9.ActivityManager.ProcessErrorStateInfo notRespondingReport:anr报告的进程状态
- 10.String waitingToKill:后台进程被kill原因
- 11.ComponentName errorReportReceiver:接收error信息的组件

第七类数据: 和instrumentation相关的数据

instrumentation 也可以说是apk的一个组件, 如果我们提供的话, 系统会默认使用

Instrumentation.java类, 按照我们一般的理解, UI 线程控制activity的生命周期, 是直接调用Activity类的方法, 时间是这样的, UI线程调用的是instrumentation的方法, 由它在调用Activity涉及生命周

期的方法，所有如果我们覆写了instrumentation的这些方法，就可以了解所有的Activity的生命周期了

- 1.ComponentName instrumentationClass: AndroidManifest.xml中定义的instrumentation信息
- 2.ApplicationInfo instrumentationInfo: instrumentation应用信息
- 3.String instrumentationProfileFile: instrumentation配置文件
- 4.IInstrumentationWatcher instrumentationWatcher: instrumentation监测器
- 5.IUiAutomationConnection instrumentationUiAutomationConnection: UiAutomation连接器
- 6.ComponentName instrumentationResultClass: 返回结果组件

第八类数据：电源信息和调试信息

- 1.BatteryStatsImpl mBatteryStats:电量信息
- 2.BatteryStatsImpl.Uid.Proc curProcBatteryStats: 当前进程电量信息
- 3.boolean debugging:处于调试中
- 4.boolean waitedForDebugger:等待调试
- 5.Dialog waitDialog:等待对话框
- 6.String adjType:adj类型（或者说标示）
- 7.int adjTypeCode:adj类型码（也是一种标示）
- 8.Object adjSource:改变adj的组件记录表
- 9.int adjSourceProcState:影响adj的进程状态
- 10.Object adjTarget: 改变adj的组件
- 11.String shortStringName: 进程记录表的字符串显示
- 12.String stringName: 进程记录表的字符串显示

第九类数据：最后我们来看一下31个boolean值

- 1.进程声明周期相关的
 - a.boolean starting:进程正在启动
 - b.boolean removed:进程系统资源已经清理
 - c.boolean killedByAm:进程被AMS主动kill掉
 - d.boolean killed:进程被kill掉了
 - e.boolean persistent:常驻内存进程
- 2.组件状态影响进程行为的
 - a.boolean empty:空进程，不含有任何组件的进程
 - b.boolean cached:缓存进程
 - c.boolean bad:60s内连续crash两次的进程被定义为bad进程
 - d.boolean hasClientActivities:进程有Activity绑定其他Service
 - e.boolean hasStartedServices:进程中包含启动了的Service
 - f.boolean foregroundServices:进程中包含前台运行的Service
 - g.boolean foregroundActivities:进程中包含前台运行的Activity
 - h.boolean repForegroundActivities:
 - i.boolean systemNoUi:系统进程，没有显示UI
 - j.boolean hasShownUi:重进程启动开始，是否已经显示UI
 - k.boolean pendingUiClean:
 - l.boolean hasAboveClient:进程中有组件使用BIND_ABOVE_CLIENT标志绑定其他Service
 - m.boolean treatLikeActivity:进程中有组件使用BIND_TREAT_LIKE_ACTIVITY标志绑定其他Service
 - n.boolean execServicesFg:前台执行Service
 - o.boolean setInForeground:设置运行前台UI
- 3.其他
 - a. boolean serviceB:进程存在service B list中
 - b.boolean serviceHighRam:由于内存原因，进程强制存在service B list中

- o c.boolean notCachedSinceIdle:进程自从上次空闲，是否属于缓存进程
- o d.boolean procStateChanged:进程状态改变
- o e.boolean reportedInteraction:是否报告交互事件
- o f.boolean unlocked:解锁状态下进程启动
- o g.boolean usingWrapper:zygote是否使用了wrapper启动apk进程
- o h.boolean reportLowMemory:报告低内存
- o i.boolean inFullBackup:进程中存在backup组件在运行
- o j.boolean whitelistManager:和电源管理相关

进程主要占用的资源：ProcessRecord容器 和 组件记录表的容器

ProcessRecord容器

永久性容器

- 1.mProcessNames：根据进程名字检索进程记录表
- 2.mPidsSelfLocked：根据进程pid检索进程记录表
- 3.mLruProcesses：lru进程记录表容器，这个容器使用的是最近最少使用算法对进程记录表进行排序，越是处于上层的越是最近使用的，对于系统来说就是最重要的，在内存吃紧回收进程时，越不容易被回收，实现起来也很简单

临时性容器

- 1.mPersistentStartingProcesses：常驻内存进程启动时容器
- 2.mProcessesOnHold：进程启动挂起容器
- 3.mProcessesToGc：将要执行gc回收的进程容器
- 4.mPendingPssProcesses：将要计算Pss数据的进程容器

一个特别的容器

- 1.mRemovedProcesses:从名字上的意思是已经移除的进程，那么什么是已经移除的进程？移除的进程为什么还需要保存？后面的（进程管理(六)apk进程的回收）小节会提到

内部四大组件记录表的容器

组件运行才是进程存在的意义，由于android系统进程间的无缝结合，所以系统需要控制到组件级别，所有的组件信息都需要映射到系统，一个ActivityRecord记录对应一个Activity的信息，一个ServiceRecord记录对应一个Service的信息，一个ConnectionRecord记录对应一个bind service的客户端信息，一个ReceiverList对应处理同一事件的一组广播，一个ContentProviderRecord记录对应一个ContentProvider信息，一个ContentProviderConnection对应一个进程中的所有ContentProvider客户端

activity记录

- 1. activities：ActivityRecord的容器，进程启动的所有的activity组件记录表

service记录

- 1.services：ServiceRecord的容器，进程启动的所有的service组件记录表
- 2.executingServices：正在运行（executing）的ServiceRecord是怎么定义的？首先需要明确的是系统是怎么控制组件的？发送消息给apk进程，apk进程处理消息，上报消息完成，这被定义为一个完整的执行过程，因此正在执行（executing）被定义为发送消息到上报完成这段时间
- 3.connections：ConnectionRecord容器，绑定service的客户端记录表

广播接收器记录

- 1.receivers: ReceiverList容器，广播接收器的记录表

ContentProvider记录

- 1.pubProviders: 名字到ContentProviderRecord的映射容器，pub是publish（发布）的意思，ContentProvider需要安装然后把自己发布到系统（AMS）中后，才能使用，安装指的是apk进程加载ContentProvider子类、初始化、创建数据库等过程，发布是将ContentProvider的binder客户端注册到AMS中
- 2.conProviders: ContentProviderConnection容器，使用ContentProvider的客户端记录表

与Activity管理有关的数据结构

ActivityRecord

ActivityRecord，源码中的注释介绍：An entry in the history stack, representing an activity.

翻译：历史栈中的一个条目，代表一个activity。

```
1  /**
2   * An entry in the history stack, representing an activity.
3   */
4  final class ActivityRecord extends ConfigurationContainer implements
AppWindowContainerListener {
5      final ActivityManagerService service; // owner
6      final IApplicationToken.Stub appToken; // window manager token
7      AppWindowContainerController mWindowContainerController;
8      final ActivityInfo info; // all about me
9      final ApplicationInfo appInfo; // information about activity's app
10
11     //省略其他成员变量
12
13     //ActivityRecord所在的TaskRecord
14     private TaskRecord task; // the task this is in.
15
16     //构造方法，需要传递大量信息
17     ActivityRecord(ActivityManagerService _service, ProcessRecord
_caller, int _launchedFromPid,
18                     int _launchedFromUid, String _launchedFromPackage,
Intent _intent, String _resolvedType,
19                     ActivityInfo aInfo, Configuration _configuration,
20                     com.android.server.am.ActivityRecord _resultTo,
String _resultWho, int _requestCode,
21                     boolean _componentsSpecified, boolean
_rootVoiceInteraction,
22                     ActivityStackSupervisor supervisor, ActivityOptions
options,
23                     com.android.server.am.ActivityRecord sourceRecord) {
24
25     }
26 }
```

ActivityRecord中存在着大量的成员变量，包含了一个Activity的所有信息。

ActivityRecord中的成员变量task表示其所在的TaskRecord，由此可以看出：ActivityRecord与TaskRecord建立了联系

\\frameworks\\base\\services\\core\\java\\com\\android\\server\\am\\ActivityStarter.java

```
1 private int startActivity(IApplicationThread caller, Intent intent, Intent
ephemeralIntent,
2     String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,
3     IVoiceInteractionSession voiceSession, IVoiceInteractor
voiceInteractor,
4     IBinder resultTo, String resultWho, int requestCode, int
callingPid, int callingUid,
5     String callingPackage, int realCallingPid, int realCallingUid,
int startFlags,
6     SafeActivityOptions options,
7     boolean ignoreTargetSecurity, boolean componentsSpecified,
ActivityRecord[] outActivity,
8     TaskRecord inTask, boolean
allowPendingRemoteAnimationRegistryLookup) {
9
10     ActivityRecord r = new ActivityRecord(mService, callerApp,
callingPid, callingUid,
11     callingPackage, intent, resolvedType, aInfo,
mService.getGlobalConfiguration(),
12     resultRecord, resultWho, requestCode, componentsSpecified,
voiceSession != null,
13     mSupervisor, checkedOptions, sourceRecord);
14
15 }
```

TaskRecord

TaskRecord，内部维护一个 `ArrayList<ActivityRecord>` 用来保存ActivityRecord。

\\frameworks\\base\\services\\core\\java\\com\\android\\server\\am\\TaskRecord.java

```
1 class TaskRecord extends ConfigurationContainer implements
TaskWindowContainerListener {
2
3     final int taskId;           //任务ID
4     final ArrayList<ActivityRecord> mActivities; //使用一个ArrayList来保存所有的
ActivityRecord
5     private ActivityStack mStack; //TaskRecord所在的ActivityStack
6
7     */
8     TaskRecord(ActivityManagersService service, int _taskId, Intent _intent,
9     Intent _affinityIntent, String _affinity, String _rootAffinity,
10     ComponentName _realActivity, ComponentName _origActivity,
boolean _rootWasReset,
11     boolean _autoRemoveRecents, boolean _askedCompatMode, int
_userId,
12     int _effectiveUid, String _lastDescription,
ArrayList<ActivityRecord> activities,
13     long lastTimeMoved, boolean neverRelinquishIdentity,
```



```

14         TaskDescription _lastTaskDescription, int taskAffiliation, int
prevTaskId,
15         int nextTaskId, int taskAffiliationColor, int callingUid, String
callingPackage,
16         int resizeMode, boolean supportsPictureInPicture, boolean
_realActivitySuspended,
17         boolean userSetupComplete, int minWidth, int minHeight) {
18
19     }
20
21     //添加Activity到顶部
22     void addActivityToTop(com.android.server.am.ActivityRecord r) {
23         addActivityAtIndex(mActivities.size(), r);
24     }
25
26     //添加Activity到指定的索引位置
27     void addActivityAtIndex(int index, ActivityRecord r) {
28         //...
29
30         r.setTask(this); //为ActivityRecord设置TaskRecord, 就是这里建立的联系
31
32         //...
33
34         index = Math.min(size, index);
35         mActivities.add(index, r); //添加到mActivities
36
37         //...
38     }
39 }
40

```

可以看到TaskRecord中使用了一个ArrayList来保存所有的ActivityRecord。
 同样，TaskRecord中的mStack表示其所在的ActivityStack。
 startActivity()时也会创建一个TaskRecord

ActivityStarter

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

```

1  class ActivityStarter {
2
3      private int setTaskFromReuseOrCreateNewTask(TaskRecord
taskToAffiliate, int preferredLaunchStackId, ActivityStack topStack) {
4          mTargetStack = computeStackFocus(mStartActivity, true,
mLaunchBounds, mLaunchFlags, mOptions);
5
6          if (mReuseTask == null) {
7              //创建一个createTaskRecord, 实际上是调用ActivityStack里面的
createTaskRecord () 方法, ActivityStack下面会讲到
8              final TaskRecord task = mTargetStack.createTaskRecord(
9
10             mSupervisor.getNextTaskIdForUserLocked(mStartActivity.userId),
mNewTaskInfo != null ? mNewTaskInfo :
mStartActivity.info,

```

```

11         mNewTaskIntent != null ? mNewTaskIntent : mIntent,
    mVoiceSession,
12         mVoiceInteractor, !mLaunchTaskBehind /* toTop */,
    mStartActivity.mActivityType);
13
14         //其他代码略
15     }
16 }
17 }

```

ActivityStack

ActivityStack,内部维护了一个 `ArrayList<TaskRecord>`, 用来管理`TaskRecord`

```

1  class ActivityStack<T extends StackWindowController> extends
    ConfigurationContainer
2      implements StackWindowListener {
3
4      /**
5       * The back history of all previous (and possibly still
6       * running) activities. It contains #TaskRecord objects.
7       */
8      private final ArrayList<TaskRecord> mTaskHistory = new ArrayList<>(); //
    使用一个ArrayList来保存TaskRecord
9
10     protected final ActivityStackSupervisor mStackSupervisor; //持有一个
    ActivityStackSupervisor, 所有的运行中的ActivityStacks都通过它来进行管
11
12     ActivityStack(ActivityDisplay display, int stackId,
    ActivityStackSupervisor supervisor,
13         int windowingMode, int activityType, boolean onTop) {
14
15     }
16     TaskRecord createTaskRecord(int taskId, ActivityInfo info, Intent intent,
17         IVoiceInteractionSession voiceSession,
    IVoiceInteractor voiceInteractor,
18         boolean toTop, int type) {
19
20         //创建一个task
21         TaskRecord task = new TaskRecord(mService, taskId, info, intent,
    voiceSession, voiceInteractor, type);
22
23         //将task添加到ActivityStack中去
24         addTask(task, toTop, "createTaskRecord");
25
26         //其他代码略
27
28         return task;
29     }
30
31     //添加Task
32     void addTask(final TaskRecord task, final boolean toTop, String
    reason) {

```

```

33
34         addTask(task, toTop ? MAX_VALUE : 0, true /*
schedulePictureInPictureModeChange */, reason);
35
36         //其他代码略
37     }
38
39     //添加Task到指定位置
40     void addTask(final TaskRecord task, int position, boolean
schedulePictureInPictureModeChange,
41                 String reason) {
42         mTaskHistory.remove(task); //若存在, 先移除
43
44         //...
45
46         mTaskHistory.add(position, task); //添加task到mTaskHistory
47         task.setStack(this); //为TaskRecord设置ActivityStack
48
49         //...
50     }
51
52 }

```

可以看到ActivityStack使用了一个ArrayList来保存TaskRecord。

另外, ActivityStack中还持有ActivityStackSupervisor对象, 这个是用来管理ActivityStacks的。ActivityStack是由ActivityStackSupervisor来创建的, 实际ActivityStackSupervisor就是用来管理ActivityStack的

ActivityStackSupervisor

ActivityStackSupervisor, 顾名思义, 就是用来管理ActivityStack的

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

```

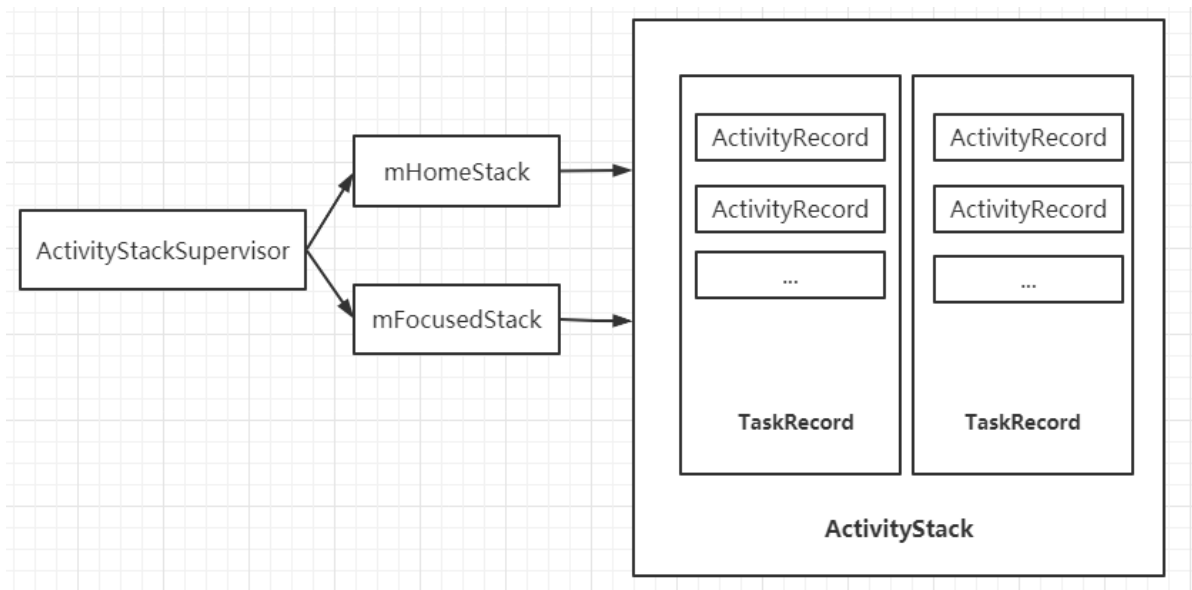
1  public class ActivityStackSupervisor extends ConfigurationContainer
   implements DisplayListener {
2
3       ActivityStack mHomeStack; //管理的是Launcher相关的任务
4
5       ActivityStack mFocusedStack; //管理非Launcher相关的任务
6
7       //创建ActivityStack
8       ActivityStack createStack(int stackId,
   ActivityStackSupervisor.ActivityDisplay display, boolean onTop) {
9           switch (stackId) {
10              case PINNED_STACK_ID:
11                  //PinnedActivityStack是ActivityStack的子类
12                  return new PinnedActivityStack(display, stackId, this,
   mRecentTasks, onTop);
13              default:
14                  //创建一个ActivityStack
15                  return new ActivityStack(display, stackId, this,
   mRecentTasks, onTop);
16          }
17      }
18
19  }

```

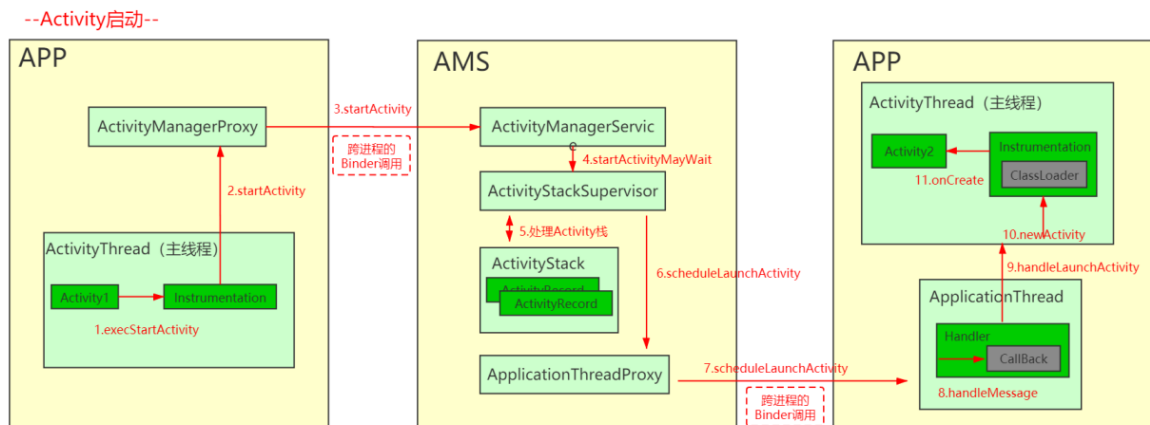
ActivityStackSupervisor内部有两个不同的ActivityStack对象：mHomeStack、mFocusedStack，用来管理不同的任务。

ActivityStackSupervisor内部包含了创建ActivityStack对象的方法。

AMS初始化时会创建一个ActivityStackSupervisor对象

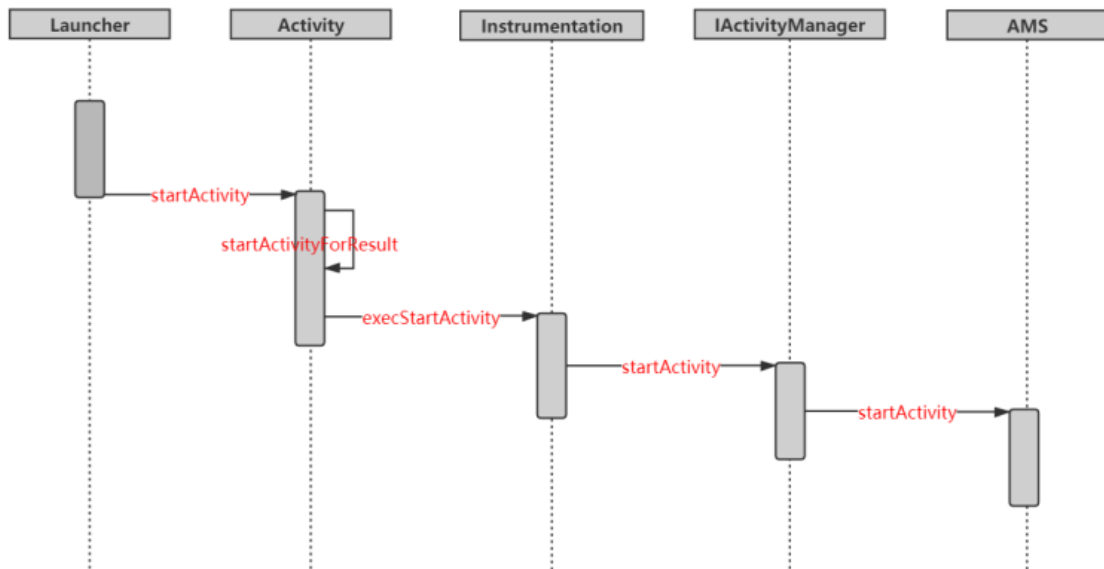


Activity启动流程相关

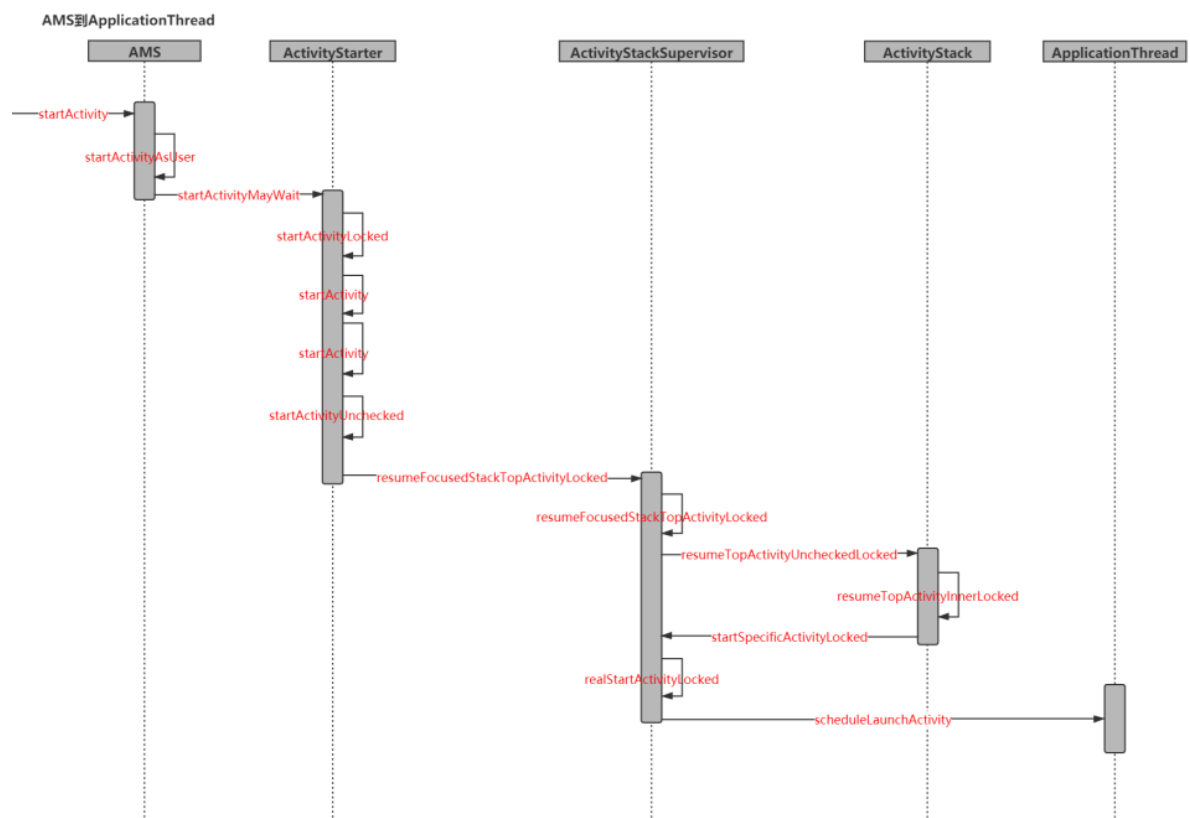


Launcher请求AMS阶段

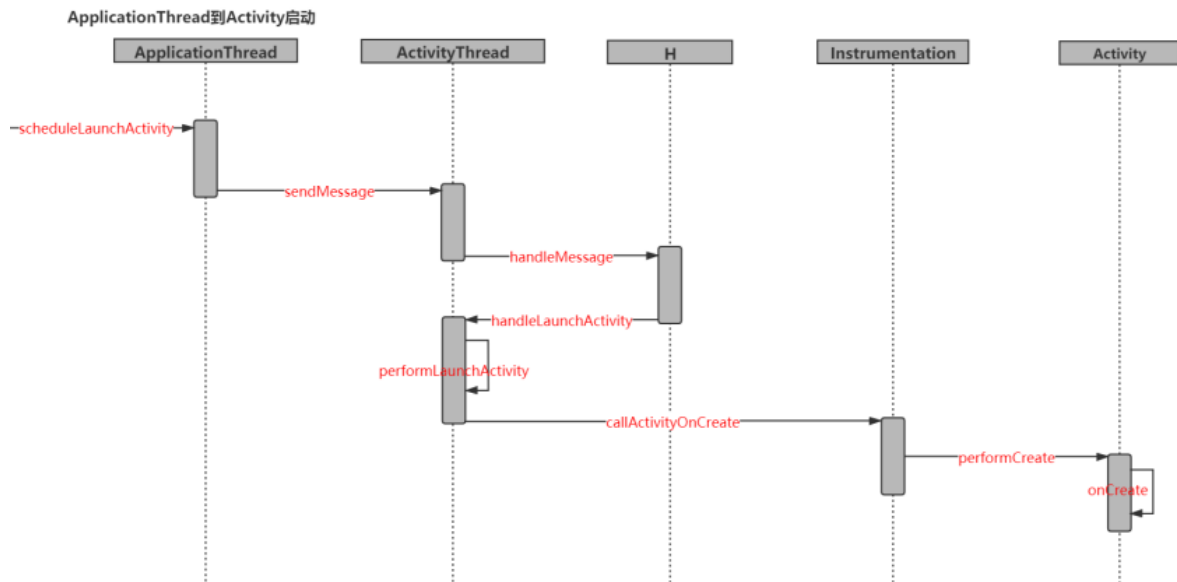
Launcher 请求 AMS



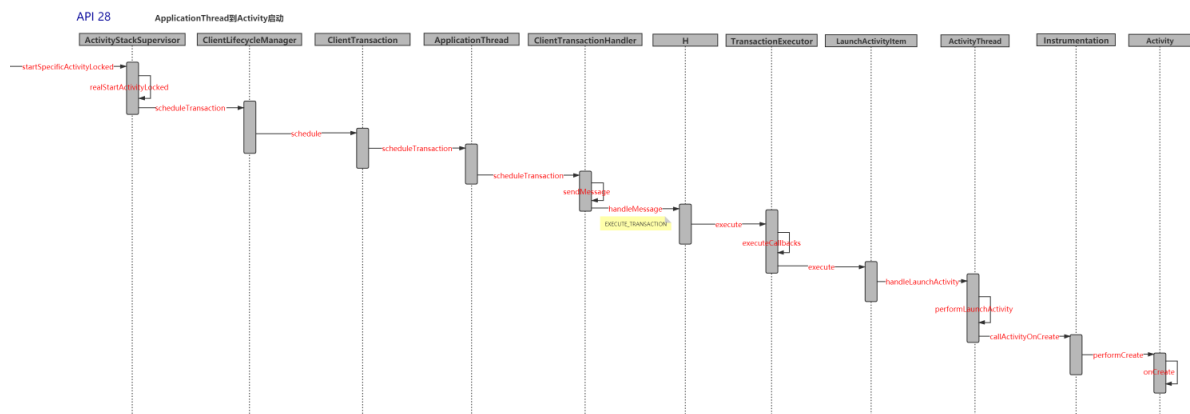
AMS到ApplicationThread阶段



ApplicationThread到Activity阶段



- API28重构之后



在哪儿判断Activity是否在AndroidManifest.xml里面注册的?

首先没有在AndroidManifest.xml注册，报错是什么样子的？

```

2020-09-10 15:42:32.460 12467-12467/com.zero.activityhookdemo E/AndroidRuntime: FATAL EXCEPTION: main
Process: com.zero.activityhookdemo, PID: 12467
android.content.ActivityNotFoundException: Unable to find explicit activity class {com.zero.activityhookdemo/com.zero.activityhookdemo.TargetActivity}; have you declared this activity in your AndroidManifest.xml?
at android.app.Instrumentation.checkStartActivityResult(Instrumentation.java:2042)
at android.app.Instrumentation.execStartActivity(Instrumentation.java:1789)
at android.app.Activity.startActivityForResult(Activity.java:5192)
at android.support.v4.app.FragmentActivity.startActivityForResult(FragmentActivity.java:767)
at android.app.Activity.startActivityForResult(Activity.java:5150)
at android.support.v4.app.FragmentActivity.startActivityForResult(FragmentActivity.java:754)
at android.app.Activity.startActivity(Activity.java:5521)
at android.app.Activity.startActivity(Activity.java:5489)
at com.zero.activityhookdemo.MainActivity.onBtnHook4Clicked(MainActivity.java:65)
  
```

从这里我们可以看到这个错误是在哪块代码报出来的

```

1 public ActivityResult execStartActivity(
2     Context who, IBinder contextThread, IBinder token, Activity
    target,
3     Intent intent, int requestCode, Bundle options) {
4     ...
5     try {
6         intent.migrateExtraStreamToClipData();
7         intent.prepareToLeaveProcess(who);
  
```

```

8      //1.通过IActivityManager调用我们执行AMS的startActivity方法，并返回执行
      结果
9      int result = ActivityManager.getService()
10         .startActivity(whoThread, who.getBasePackageName(), intent,
11
12         intent.resolveTypeIfNeeded(who.getContentResolver()),
13         token, target != null ? target.mEmbeddedID : null,
14         requestCode, 0, null, options);
15      //2. 检查结果
16      checkStartActivityResult(result, intent);
17      } catch (RemoteException e) {
18          throw new RuntimeException("Failure from system", e);
19      }
20      return null;
    }

```

我们来看下是如何检查的

```

1  public static void checkStartActivityResult(int res, Object intent) {
2      if (!ActivityManager.isStartResultFatalError(res)) {
3          return;
4      }
5      switch (res) {
6          case ActivityManager.START_INTENT_NOT_RESOLVED:
7          case ActivityManager.START_CLASS_NOT_FOUND:
8              //3. 这里我们找到了报错的地方，原来是res结果为
              START_INTENT_NOT_RESOLVED,
9              //START_CLASS_NOT_FOUND就会报这个错误
10             if (intent instanceof Intent && ((Intent)intent).getComponent()
11             != null)
12                 throw new ActivityNotFoundException(
13                     "Unable to find explicit activity class "
14                     + ((Intent)intent).getComponent().toShortString()
15                     + "; have you declared this activity in your
              AndroidManifest.xml?");
16             throw new ActivityNotFoundException(
17                 "No Activity found to handle " + intent);
18          case ActivityManager.START_PERMISSION_DENIED:
19             throw new SecurityException("Not allowed to start activity "
20             + intent);
21          case ActivityManager.START_FORWARD_AND_REQUEST_CONFLICT:
22             throw new AndroidRuntimeException(
23                 "FORWARD_RESULT_FLAG used while also requesting a
              result");
24          case ActivityManager.START_NOT_ACTIVITY:
25             throw new IllegalArgumentException(
26                 "PendingIntent is not an activity");
27          case ActivityManager.START_NOT_VOICE_COMPATIBLE:
28             throw new SecurityException(
29                 "Starting under voice control not allowed for: " +
              intent);
30          case ActivityManager.START_VOICE_NOT_ACTIVE_SESSION:
31             throw new IllegalStateException(
32                 "Session calling startVoiceActivity does not match
              active session");
33          case ActivityManager.START_VOICE_HIDDEN_SESSION:
34             throw new IllegalStateException(

```

```

34         "Cannot start voice activity on a hidden session");
35     case ActivityManager.START_ASSISTANT_NOT_ACTIVE_SESSION:
36         throw new IllegalStateException(
37             "Session calling startAssistantActivity does not match
active session");
38     case ActivityManager.START_ASSISTANT_HIDDEN_SESSION:
39         throw new IllegalStateException(
40             "Cannot start assistant activity on a hidden session");
41     case ActivityManager.START_CANCELED:
42         throw new AndroidRuntimeException("Activity could not be started
for "
43             + intent);
44     default:
45         throw new AndroidRuntimeException("Unknown error code "
46             + res + " when starting " + intent);
47     }
48 }

```

接着我们来看下AMS里面是如何判断activity没有注册的，首先我们得明白startActivity执行的主流程

我们要善于利用奥卡姆剃刀，抽出主干部分进行分析

AMS.startActivity

```

-> AMS.startActivityAsUser          [ActivityManagerService]
-> ASC.obtainStarter                [ActivityStartController]
-> ASR.execute                      [ActivityStarter]
-> ASR.startActivityMayWait
-> ASR.startActivity
-> ASR.startActivity
-> ASR.startActivity
-> ASR.startActivityUnchecked
-> AS.startActivityLocked <-      [ActivityStack]
-> ASS.resumeFocusedStackTopActivityLocked [ActivityStackSupervisor]
-> AS.resumeTopActivityUncheckedLocked
-> AS.resumeTopActivityInnerLocked
-> AS.startPausingLocked #PauseActivityItem
-> AMS.getLifecycleManager
-> CLM.scheduleTransaction         [ClientLifecycleManager]
-> CTR.getClient                   [ClientTransaction]
-> CTR.schedule
-> APT.scheduleTransaction         [IApplicationThread]
-> APT.scheduleTransaction
-> AT.scheduleTransaction          [ActivityThread]
-> CTH.scheduleTransaction         [ClientTransactionHandler]
-> AT.sendMessage #EXECUTE_TRANSACTION
-> TE.execute                      [TransactionExecutor]
-> TE.executeCallbacks
-> TE.cycleToPath
-> TE.performLifecycleSequence
-> PauseActivityItem.execute
-> AT.handlePauseActivity
-> AT.performPauseActivity
-> AT.performPauseActivityIfNeeded
-> Instrumentation.callActivityOnPause

```


- > Activity.performPause
- > Activity.onPause
- > PauseActivityItem.postExecute
- > AMS.activityPaused
- > AS.activityPausedLocked
- > AS.completePauseLocked
- > ASS.resumeFocusedStackTopActivityLocked
- > AS.resumeTopActivityUncheckedLocked
- > AS.resumeTopActivityInnerLocked
- > ASS.startSpecificActivityLocked #发现app进程还没起来 startActivity中断了
- > AMS.startProcessLocked
- > AMS.startProcess
- > Process.start
- > ZygoteProcess.start
- > ZygoteProcess.startViaZygote
- > ZygoteProcess.openZygoteSocketIfNeeded
- > ZygoteState.connect
- > ZygoteProcess.zygoteSendArgsAndGetResult
- > zygoteState.writer.write #localsocket通信
- > app_main.main #zygote从哪儿来的
- > AppRuntime.start #com.android.internal.os.ZygoteInit
- > ZygoteInit.main #app启动中zygote服务端
- > new ZygoteServer
- > ZygoteServer.runSelectLoop
- > ZygoteServer.acceptCommandPeer
- > ServerSocket.accept
- > ZygoteConnection.processOneCommand
- > Zygote.forkAndSpecialize
- > Zygote.nativeForkAndSpecialize
- > com_android_internal_os_Zygote.cpp#nativeForkAndSpecialize
- > #ForkAndSpecializeCommon
- > fork #不再往底层分析 processOneCommand <-
- > ZygoteConnection.handleChildProc
- > ZygoteInit.zygoteInit
- > RuntimeInit.redirectLogStreams
- > RuntimeInit.commonInit
- > ZygoteInit.nativeZygoteInit #在AndroidRuntime里面注册
- > com_android_internal_os_ZygoteInit_nativeZygoteInit
- > gCurRuntime->onZygoteInit #具体实现在AppRuntime即app_main里面
- > ProcessState::self->startThreadPool #开启binder线程池
- > RuntimeInit.applicationInit #回到ZygoteInit
- > RuntimeInit.findStaticMain
- > MethodAndArgsCaller.run #回到ZygoteInit.main
- > Method.invoke #android.app.ActivityThread
- > AT.main
- > new ActivityThread
- > AT.attach
- > AMS.attachApplication #binder
- > AMS.attachApplicationLocked
- > APT.bindApplication #binder
- > AT.sendMessage #BIND_APPLICATION
- > AT.handleBindApplication

- > AT.getPackageInfoNoCheck #获取LoadedApk
- > ContextImpl.createAppContext
- > 反射创建Instrumentation
- > LoadedApk.makeApplication
- > Instrumentation.newApplication
- > Application.attach
- > Instrumentation.callApplicationOnCreate
- > ASS.attachApplicationLocked#回到attachApplicationLocked
- > ASS.realStartActivityLocked
- > 通过ClientTransaction启动Activity

我们找到在ASR.startActivity[2]中返回了START_INTENT_NOT_RESOLVED,
START_CLASS_NOT_FOUND

```

1  private int startActivity(IApplicationThread caller, Intent intent, Intent
    ephemeralIntent,
2      String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,
3      IVoiceInteractionSession voiceSession, IVoiceInteractor
    voiceInteractor,
4      IBinder resultTo, String resultWho, int requestCode, int
    callingPid, int callingUid,
5      String callingPackage, int realCallingPid, int realCallingUid,
    int startFlags,
6      SafeActivityOptions options,
7      boolean ignoreTargetSecurity, boolean componentsSpecified,
    ActivityRecord[] outActivity,
8      TaskRecord inTask, boolean
    allowPendingRemoteAnimationRegistryLookup) {
9      int err = ActivityManager.START_SUCCESS;
10     ...
11     //接下来开始做一些校验判断
12     if (err == ActivityManager.START_SUCCESS && intent.getComponent() ==
    null) {
13         // We couldn't find a class that can handle the given Intent.
14         // That's the end of that!
15         err = ActivityManager.START_INTENT_NOT_RESOLVED; //从Intent中无法找
    到相应的Component
16     }
17
18     if (err == ActivityManager.START_SUCCESS && aInfo == null) {
19         // We couldn't find the specific class specified in the Intent.
20         // Also the end of the line.
21         err = ActivityManager.START_CLASS_NOT_FOUND; //从Intent中无法找到相
    应的ActivityInfo
22     }
23     ...
24     if (err != START_SUCCESS) { //不能成功启动了, 返回err
25         if (resultRecord != null) {
26             resultStack.sendActivityResultLocked(
27                 -1, resultRecord, resultWho, requestCode,
    RESULT_CANCELED, null);
28         }
29         SafeActivityOptions.abort(options);
30         return err;
31     }
32     ...

```

```

33         //创建出我们的目标ActivityRecord对象，存到传入数组0索引上
34         ActivityRecord r = new ActivityRecord(mService, callerApp,
callingPid, callingUid,
35             callingPackage, intent, resolvedType, aInfo,
mService.getGlobalConfiguration(),
36             resultRecord, resultWho, requestCode, componentSpecified,
voiceSession != null,
37             mSupervisor, checkedOptions, sourceRecord);
38         ...
39         return startActivity(r, sourceRecord, voiceSession, voiceInteractor,
startFlags,
40             true /* doResume */, checkedOptions, inTask, outActivity);
41     }

```

但是 intent.getComponent(), aInfo 又是从哪儿获取的呢，我们回溯到 startActivityMayWait

```

1 private int startActivityMayWait(IApplicationThread caller, int callinguid,
2     String callingPackage, Intent intent, String resolvedType,
3     IVoiceInteractionSession voiceSession, IVoiceInteractor
voiceInteractor,
4     IBinder resultTo, String resultWho, int requestCode, int
startFlags,
5     ProfilerInfo profilerInfo, WaitResult outResult,
6     Configuration globalConfig, SafeActivityOptions options, boolean
ignoreTargetSecurity,
7     int userId, TaskRecord inTask, String reason,
8     boolean allowPendingRemoteAnimationRegistryLookup) {
9     // Refuse possible leaked file descriptors
10    ...
11    intent = new Intent(intent);
12    if (componentSpecified
13        && !(Intent.ACTION_VIEW.equals(intent.getAction()) &&
intent.getData() == null)
14        &&
!Intent.ACTION_INSTALL_INSTANT_APP_PACKAGE.equals(intent.getAction())
15        &&
!Intent.ACTION_RESOLVE_INSTANT_APP_PACKAGE.equals(intent.getAction())
16        && mService.getPackageManagerInternalLocked()
17        .isInstantAppInstallerComponent(intent.getComponent())) {
18        // intercept intents targeted directly to the ephemeral
installer the
19        // ephemeral installer should never be started with a raw
Intent; instead
20        // adjust the intent so it looks like a "normal" instant app
launch
21        intent.setComponent(null /*component*/);
22        componentSpecified = false;
23    }
24
25    ResolveInfo rInfo = mSupervisor.resolveIntent(intent, resolvedType,
userId,
26        0 /* matchFlags */,
27        computeResolveFilterUid(

```

```

28         callingUid, realCallingUid,
mRequest.filterCallingUid));
29         if (rInfo == null) {
30             UserInfo userInfo = mSupervisor.getUserInfo(userId);
31             if (userInfo != null && userInfo.isManagedProfile()) {
32                 // Special case for managed profiles, if attempting to
launch non-crypto aware
33                 // app in a locked managed profile from an unlocked parent
allow it to resolve
34                 // as user will be sent via confirm credentials to unlock
the profile.
35                 UserManager userManager =
UserManager.get(mService.mContext);
36                 boolean profileLockedAndParentUnlockingOrUnlocked = false;
37                 long token = Binder.clearCallingIdentity();
38                 try {
39                     UserInfo parent = userManager.getProfileParent(userId);
40                     profileLockedAndParentUnlockingOrUnlocked = (parent !=
null)
41                         &&
userManager.isUserUnlockingOrUnlocked(parent.id)
42                         &&
!userManager.isUserUnlockingOrUnlocked(userId);
43                 } finally {
44                     Binder.restoreCallingIdentity(token);
45                 }
46                 if (profileLockedAndParentUnlockingOrUnlocked) {
47                     rInfo = mSupervisor.resolveIntent(intent, resolvedType,
userId,
48                         PackageManager.MATCH_DIRECT_BOOT_AWARE
49                         |
PackageManager.MATCH_DIRECT_BOOT_UNAWARE,
50                         computeResolveFilterUid(
51                             callingUid, realCallingUid,
mRequest.filterCallingUid));
52                 }
53             }
54         }
55         // Collect information about the target of the Intent.
56         ActivityInfo aInfo = mSupervisor.resolveActivity(intent, rInfo,
startFlags, profilerInfo); //收集Intent所指向的Activity信息，当存在多个可供选择的
Activity，则直接向用户弹出resolveActivity
57         ...
58         final ActivityRecord[] outRecord = new ActivityRecord[1];
59         int res = startActivity(caller, intent, ephemeralIntent,
resolvedType, aInfo, rInfo,
60             voiceSession, voiceInteractor, resultTo, resultWho,
requestCode, callingPid,
61             callingUid, callingPackage, realCallingPid,
realCallingUid, startFlags, options,
62             ignoreTargetSecurity, componentsSpecified, outRecord,
inTask, reason,
63             allowPendingRemoteAnimationRegistryLookup);
64         ...
65         return res;
66     }
67 }

```

我们看下aInfo哪儿来的

```
1  ActivityInfo resolveActivity(Intent intent, ResolveInfo rInfo, int
startFlags,
2      ProfilerInfo profilerInfo) {
3      final ActivityInfo aInfo = rInfo != null ? rInfo.activityInfo :
null;
4      if (aInfo != null) {
5          // Store the found target back into the intent, because now that
6          // we have it we never want to do this again. For example, if
the
7          // user navigates back to this point in the history, we should
8          // always restart the exact same activity.
9          intent.setComponent(new ComponentName(
10              aInfo.applicationInfo.packageName, aInfo.name));
11
12          // Don't debug things in the system process
13          ...
14      }
15      return aInfo;
16  }
```

发现是从rInfo来的

```
1  ResolveInfo resolveIntent(Intent intent, String resolvedType, int userId,
int flags,
2      int filterCallingUid) {
3      synchronized (mService) {
4          try {
5              ...
6              final long token = Binder.clearCallingIdentity();
7              try {
8                  return
mService.getPackageManagerInternalLocked().resolveIntent(
9                  intent, resolvedType, modifiedFlags, userId,
true, filterCallingUid);
10             } finally {
11                 Binder.restoreCallingIdentity(token);
12             }
13             ...
14         }
15     }
```

那么rInfo怎么获取的呢?

```
1  PackageManagerInternal getPackageManagerInternalLocked() {
2      if (mPackageManagerInt == null) {
3          mPackageManagerInt =
LocalServices.getService(PackageManagerInternal.class);
4      }
5      return mPackageManagerInt;
6  }
```

具体实现类是PackageManagerService

```

1  @Override
2      public ResolveInfo resolveIntent(Intent intent, String resolvedType,
3          int flags, int userId) {
4      return resolveIntentInternal(intent, resolvedType, flags, userId,
5          false,
6          Binder.getCallingUid());
7  }

```

再看 resolveIntentInternal

```

1  private ResolveInfo resolveIntentInternal(Intent intent, String
2      resolvedType,
3      int flags, int userId, boolean resolveForStart, int
4      filterCallingUid) {
5      try {
6          ...
7          // 获取 ResolveInfo 列表
8          final List<ResolveInfo> query =
9              queryIntentActivitiesInternal(intent, resolvedType,
10                  flags, filterCallingUid, userId, resolveForStart, true
11                  /*allowDynamicSplits*/);
12          Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
13          // 找出最好的返回
14          final ResolveInfo bestChoice =
15              chooseBestActivity(intent, resolvedType, flags, query,
16              userId);
17          return bestChoice;
18      } finally {
19          Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
20      }
21  }

```

再看 queryIntentActivitiesInternal

```

1  private @NonNull List<ResolveInfo> queryIntentActivitiesInternal(Intent
2      intent,
3      String resolvedType, int flags, int filterCallingUid, int
4      userId,
5      boolean resolveForStart, boolean allowDynamicSplits) {
6      ...
7      if (comp != null) {
8          final List<ResolveInfo> list = new ArrayList<ResolveInfo>(1);
9          final ActivityInfo ai = getActivityInfo(comp, flags, userId);
10         if (ai != null) {
11             // when specifying an explicit component, we prevent the
12             // activity from being
13             // used when either 1) the calling package is normal and the
14             // activity is within
15             // an ephemeral application or 2) the calling package is
16             // ephemeral and the
17             // activity is not visible to ephemeral applications.
18             final boolean matchInstantApp =
19                 (flags & PackageManager.MATCH_INSTANT) != 0;
20             final boolean matchVisibleToInstantAppOnly =
21                 (flags &
22                 PackageManager.MATCH_VISIBLE_TO_INSTANT_APP_ONLY) != 0;

```

```

17         final boolean matchExplicitlyVisibleOnly =
18             (flags &
PackageManager.MATCH_EXPLICITLY_VISIBLE_ONLY) != 0;
19         final boolean isCallerInstantApp =
20             instantAppPkgName != null;
21         final boolean isTargetSameInstantApp =
22             comp.getPackageName().equals(instantAppPkgName);
23         final boolean isTargetInstantApp =
24             (ai.applicationInfo.privateFlags
25                 & ApplicationInfo.PRIVATE_FLAG_INSTANT) !=
0;
26         final boolean isTargetVisibleToInstantApp =
27             (ai.flags &
ActivityInfo.FLAG_VISIBLE_TO_INSTANT_APP) != 0;
28         final boolean isTargetExplicitlyVisibleToInstantApp =
29             isTargetVisibleToInstantApp
30             && (ai.flags &
ActivityInfo.FLAG_IMPLICITLY_VISIBLE_TO_INSTANT_APP) == 0;
31         final boolean isTargetHiddenFromInstantApp =
32             !isTargetVisibleToInstantApp
33             || (matchExplicitlyVisibleOnly &&
!isTargetExplicitlyVisibleToInstantApp);
34         final boolean blockResolution =
35             !isTargetSameInstantApp
36             && ((!matchInstantApp && !isCallerInstantApp &&
isTargetInstantApp)
37                 || (matchVisibleToInstantAppOnly &&
isCallerInstantApp
38                     && isTargetHiddenFromInstantApp));
39         if (!blockResolution) {
40             final ResolveInfo ri = new ResolveInfo();
41             ri.activityInfo = ai;
42             list.add(ri);
43         }
44     }
45     return applyPostResolutionFilter(
46         list, instantAppPkgName, allowDynamicSplits,
filterCallingUid, resolveForStart,
47         userId, intent);
48     }
49
50 }

```

原来是从getActivityInfo获取的

```

1  @Override
2  public ActivityInfo getActivityInfo(ComponentName component, int flags,
int userId) {
3      return getActivityInfoInternal(component, flags,
Binder.getCallingUid(), userId);
4  }

```

```

1  private ActivityInfo getActivityInfoInternal(ComponentName component, int
flags,
2      int filterCallingUid, int userId) {
3      if (!userManager.exists(userId)) return null;

```

```

4         flags = updateFlagsForComponent(flags, userId, component);
5
6         if (!isRecentsAccessingChildProfiles(Binder.getCallingUid(),
7         userId)) {
8
9             mPermissionManager.enforceCrossUserPermission(Binder.getCallingUid(),
10            userId,
11                false /* requireFullPermission */, false /* checkShell
12            */, "get activity info");
13            }
14
15            synchronized (mPackages) {
16                //关键点
17                PackageParser.Activity a =
18                mActivities.mActivities.get(component);
19
20                if (DEBUG_PACKAGE_INFO) Log.v(TAG, "getActivityInfo " +
21                component + ": " + a);
22                if (a != null && mSettings.isEnabledAndMatchLPr(a.info, flags,
23                userId)) {
24
25                    PackageSetting ps =
26                    mSettings.mPackages.get(component.getPackageName());
27                    if (ps == null) return null;
28                    if (filterAppAccessLPr(ps, filterCallingUid, component,
29                    TYPE_ACTIVITY, userId)) {
30                        return null;
31                    }
32                    //关键点
33                    return PackageParser.generateActivityInfo(
34                        a, flags, ps.readUserState(userId), userId);
35                }
36                if (mResolveComponentName.equals(component)) {
37                    return PackageParser.generateActivityInfo(
38                        mResolveActivity, flags, new PackageUserState(),
39                        userId);
40                }
41            }
42            return null;
43        }
44    }

```