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像，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对像的引用，进而通过该代理对像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

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

```
private void run() {
            ...
            createSystemContext();
            // Create the system service manager.
            mSystemServiceManager = new
SystemServiceManager(mSystemContext);
            mSystemServiceManager.setStartInfo(mRuntimeRestart,
                    mRuntimeStartElapsedTime, mRuntimeStartUptime);
            LocalServices.addService(SystemServiceManager.class,
mSystemServiceManager);
            // Prepare the thread pool for init tasks that can be
parallelized
            SystemServerInitThreadPool.get();
        } finally {
            traceEnd();  // InitBeforeStartServices
        }
        // Start services.
        try {
            traceBeginAndSlog("StartServices");
            startBootstrapServices();
            startCoreServices();
            startOtherServices();
            SystemServerInitThreadPool.shutdown();
        } catch (Throwable ex) {
            throw ex;
        } finally {
```

```
            traceEnd();
        }
        ...
        // Loop forever.
        Looper.loop();
        throw new RuntimeException("Main thread loop unexpectedly exited");
    }
```

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

```
private void startBootstrapServices() {
       ...
        // Activity manager runs the show.
        traceBeginAndSlog("StartActivityManager");
        //启动了AMS
        mActivityManagerService = mSystemServiceManager.startService(
                ActivityManagerService.Lifecycle.class).getService();

mActivityManagerService.setSystemServiceManager(mSystemServiceManager);
        mActivityManagerService.setInstaller(installer);
        traceEnd();
        ...
        // Now that the power manager has been started, let the activity
manager
        // initialize power management features.
        traceBeginAndSlog("InitPowerManagement");
        mActivityManagerService.initPowerManagement();
        traceEnd();
// Set up the Application instance for the system process and get started.
        traceBeginAndSlog("SetSystemProcess");
        mActivityManagerService.setSystemProcess();
        traceEnd();

    }
```

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

```
  @SuppressWarnings("unchecked")
    public SystemService startService(String className) {
        final Class<SystemService> serviceClass;
        try {
            serviceClass = (Class<SystemService>)Class.forName(className);
        } catch (ClassNotFoundException ex) {
            Slog.i(TAG, "Starting " + className);
            throw new RuntimeException("Failed to create service " +
className
                    + ": service class not found, usually indicates that the
caller should "
                    + "have called PackageManager.hasSystemFeature() to
check whether the "
                    + "feature is available on this device before trying to
start the "
                    + "services that implement it", ex);
        }
        return startService(serviceClass);
```

```
    }
```

```
 @SuppressWarnings("unchecked")
    public <T extends SystemService> T startService(Class<T> serviceClass) {
        try {
            final String name = serviceClass.getName();
            Slog.i(TAG, "Starting " + name);
            Trace.traceBegin(Trace.TRACE_TAG_SYSTEM_SERVER, "StartService "
+ name);

            // Create the service.
            if (!SystemService.class.isAssignableFrom(serviceClass)) {
                throw new RuntimeException("Failed to create " + name
                        + ": service must extend " +
SystemService.class.getName());
            }
            final T service;
            try {
                Constructor<T> constructor =
serviceClass.getConstructor(Context.class);
                service = constructor.newInstance(mContext);
            } catch (InstantiationException ex) {
                throw new RuntimeException("Failed to create service " +
name
                        + ": service could not be instantiated", ex);
            } catch (IllegalAccessException ex) {
                throw new RuntimeException("Failed to create service " +
name
                        + ": service must have a public constructor with a
Context argument", ex);
            } catch (NoSuchMethodException ex) {
                throw new RuntimeException("Failed to create service " +
name
                        + ": service must have a public constructor with a
Context argument", ex);
            } catch (InvocationTargetException ex) {
                throw new RuntimeException("Failed to create service " +
name
                        + ": service constructor threw an exception", ex);
            }

            startService(service);
            return service;
        } finally {
            Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);
        }
    }
```

```
public void startService(@NonNull final SystemService service) {
        // Register it.
        mServices.add(service);
        // Start it.
        long time = SystemClock.elapsedRealtime();
        try {
            service.onStart();
        } catch (RuntimeException ex) {
            throw new RuntimeException("Failed to start service " +
service.getClass().getName()
                    + ": onStart threw an exception", ex);
        }
        warnIfTooLong(SystemClock.elapsedRealtime() - time, service,
"onStart");
    }
```

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

```
 public static final class Lifecycle extends SystemService {
        private final ActivityManagerService mService;

        public Lifecycle(Context context) {
            super(context);
            mService = new ActivityManagerService(context);
        }

        @Override
        public void onStart() {
            mService.start();
        }

        @Override
        public void onBootPhase(int phase) {
            mService.mBootPhase = phase;
            if (phase == PHASE_SYSTEM_SERVICES_READY) {
                mService.mBatteryStatsService.systemServicesReady();
                mService.mServices.systemServicesReady();
            }
        }

        @Override
        public void onCleanupUser(int userId) {
            mService.mBatteryStatsService.onCleanupUser(userId);
        }

        public ActivityManagerService getService() {
            return mService;
        }
    }
```

再来开课AMS初始化做了什么

```

```

```
// Note: This method is invoked on the main thread but may need to attach various
    // handlers to other threads.  So take care to be explicit about the looper.
    public ActivityManagerService(Context systemContext) {
        LockGuard.installLock(this, LockGuard.INDEX_ACTIVITY);
        mInjector = new Injector();
        mContext = systemContext;//赋值mContext

        mFactoryTest = FactoryTest.getMode();
        mSystemThread = ActivityThread.currentActivityThread();//获取当前的ActivityThread
        mUiContext = mSystemThread.getSystemUiContext();//赋值mUiContext

        Slog.i(TAG, "Memory class: " + ActivityManager.staticGetMemoryClass());

        mPermissionReviewRequired = mContext.getResources().getBoolean(

 com.android.internal.R.bool.config_permissionReviewRequired);
 //创建Handler线程，用来处理handler消息
        mHandlerThread = new ServiceThread(TAG,
                THREAD_PRIORITY_FOREGROUND, false /*allowIo*/);
        mHandlerThread.start();
        mHandler = new MainHandler(mHandlerThread.getLooper());
        mUiHandler = mInjector.getUiHandler(this);//处理ui相关msg的Handler

        mProcStartHandlerThread = new ServiceThread(TAG + ":procStart",
                THREAD_PRIORITY_FOREGROUND, false /* allowIo */);
        mProcStartHandlerThread.start();
        mProcStartHandler = new Handler(mProcStartHandlerThread.getLooper());
//管理AMS的一些常量，厂商定制系统就可能修改此处
        mConstants = new ActivityManagerConstants(this, mHandler);

        /* static; one-time init here */
        if (sKillHandler == null) {
            sKillThread = new ServiceThread(TAG + ":kill",
                    THREAD_PRIORITY_BACKGROUND, true /* allowIo */);
            sKillThread.start();
            sKillHandler = new KillHandler(sKillThread.getLooper());
        }
//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
        mFgBroadcastQueue = new BroadcastQueue(this, mHandler,
                "foreground", BROADCAST_FG_TIMEOUT, false);
        mBgBroadcastQueue = new BroadcastQueue(this, mHandler,
                "background", BROADCAST_BG_TIMEOUT, true);
        mBroadcastQueues[0] = mFgBroadcastQueue;
        mBroadcastQueues[1] = mBgBroadcastQueue;
//初始化管理Service的 ActiveServices对象
        mServices = new ActiveServices(this);
        mProviderMap = new ProviderMap(this);//初始化Provider的管理者
        mAppErrors = new AppErrors(mUiContext, this);//初始化APP错误日志的打印器
 //创建电池统计服务，  并输出到指定目录
        File dataDir = Environment.getDataDirectory();
        File systemDir = new File(dataDir, "system");
        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",
74              ConfigurationInfo.GL_ES_VERSION_UNDEFINED);
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 =
```

```
                new TaskChangeNotificationController(this,
mStackSupervisor, mHandler);
        //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
        mActivityStartController = new ActivityStartController(this);
        mRecentTasks = createRecentTasks();
        mStackSupervisor.setRecentTasks(mRecentTasks);
        mLockTaskController = new LockTaskController(mContext,
mStackSupervisor, mHandler);
        mLifecycleManager = new ClientLifecycleManager();
    //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
        mProcessCpuThread = new Thread("CpuTracker") {
            @Override
            public void run() {
                synchronized (mProcessCpuTracker) {
                    mProcessCpuInitLatch.countDown();
                    mProcessCpuTracker.init();
                }
                while (true) {
                    try {
                        try {
                            synchronized(this) {
                                final long now =
SystemClock.uptimeMillis();
                                long nextCpuDelay =
(mLastCpuTime.get()+MONITOR_CPU_MAX_TIME)-now;
                                long nextWriteDelay =
(mLastWriteTime+BATTERY_STATS_TIME)-now;
                                //Slog.i(TAG, "Cpu delay=" + nextCpuDelay
                                //        + ", write delay=" +
nextWriteDelay);
                                if (nextWriteDelay < nextCpuDelay) {
                                    nextCpuDelay = nextWriteDelay;
                                }
                                if (nextCpuDelay > 0) {
                                    mProcessCpuMutexFree.set(true);
                                    this.wait(nextCpuDelay);
                                }
                            }
                        } catch (InterruptedException e) {
                        }
                        updateCpuStatsNow();
                    } catch (Exception e) {
                        Slog.e(TAG, "Unexpected exception collecting
process stats", e);
                    }
                }
            }
        };

        mHiddenApiBlacklist = new HiddenApiSettings(mHandler, mContext);
//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
        Watchdog.getInstance().addMonitor(this);
        Watchdog.getInstance().addThread(mHandler);

        // bind background thread to little cores
```

```
        // this is expected to fail inside of framework tests because apps can't touch cpusets directly
        // make sure we've already adjusted system_server's internal view of itself first
        updateOomAdjLocked();
        try {

 Process.setThreadGroupAndCpuset(BackgroundThread.get().getThreadId(),
                    Process.THREAD_GROUP_BG_NONINTERACTIVE);
        } catch (Exception e) {
            Slog.w(TAG, "Setting background thread cpuset failed");
        }

    }
```

```
 private void start() {
        removeAllProcessGroups();
        mProcessCpuThread.start();

        mBatteryStatsService.publish();
        mAppOpsService.publish(mContext);
        Slog.d("AppOps", "AppOpsService published");
        LocalServices.addService(ActivityManagerInternal.class, new LocalService());
        // Wait for the synchronized block started in mProcessCpuThread,
        // so that any other acccess to mProcessCpuTracker from main thread
        // will be blocked during mProcessCpuTracker initialization.
         //等待mProcessCpuThread完成初始化后， 释放锁，初始化期间禁止访问
        try {
            mProcessCpuInitLatch.await();
        } catch (InterruptedException e) {
            Slog.wtf(TAG, "Interrupted wait during start", e);
            Thread.currentThread().interrupt();
            throw new IllegalStateException("Interrupted wait during start");
        }
    }
```

然后来看看setSystemProcess 干了什么事情

```
 public void setSystemProcess() {
        try {
            ServiceManager.addService(Context.ACTIVITY_SERVICE, this, /* allowIsolated= */ true,
                    DUMP_FLAG_PRIORITY_CRITICAL | DUMP_FLAG_PRIORITY_NORMAL | DUMP_FLAG_PROTO);
            ServiceManager.addService(ProcessStats.SERVICE_NAME, mProcessStats);
            ServiceManager.addService("meminfo", new MemBinder(this), /* allowIsolated= */ false,
                    DUMP_FLAG_PRIORITY_HIGH);
            ServiceManager.addService("gfxinfo", new GraphicsBinder(this));
            ServiceManager.addService("dbinfo", new DbBinder(this));
            if (MONITOR_CPU_USAGE) {
                ServiceManager.addService("cpuinfo", new CpuBinder(this),
```

```
                        /* allowIsolated= */ false,
DUMP_FLAG_PRIORITY_CRITICAL);
            }
            ServiceManager.addService("permission", new
PermissionController(this));
            ServiceManager.addService("processinfo", new
ProcessInfoService(this));

            ApplicationInfo info =
mContext.getPackageManager().getApplicationInfo(
                    "android", STOCK_PM_FLAGS | MATCH_SYSTEM_ONLY);
            mSystemThread.installSystemApplicationInfo(info,
getClass().getClassLoader());

            synchronized (this) {
                ProcessRecord app = newProcessRecordLocked(info,
info.processName, false, 0);
                app.persistent = true;
                app.pid = MY_PID;
                app.maxAdj = ProcessList.SYSTEM_ADJ;
                app.makeActive(mSystemThread.getApplicationThread(),
mProcessStats);
                synchronized (mPidsSelfLocked) {
                    mPidsSelfLocked.put(app.pid, app);
                }
                updateLruProcessLocked(app, false, null);
                updateOomAdjLocked();
            }
        } catch (PackageManager.NameNotFoundException e) {
            throw new RuntimeException(
                    "Unable to find android system package", e);
        }

        // Start watching app ops after we and the package manager are up
and running.
        mAppOpsService.startWatchingMode(AppOpsManager.OP_RUN_IN_BACKGROUND,
null,
                new IAppOpsCallback.Stub() {
                    @Override public void opChanged(int op, int uid, String
packageName) {
                        if (op == AppOpsManager.OP_RUN_IN_BACKGROUND &&
packageName != null) {
                            if (mAppOpsService.checkOperation(op, uid,
packageName)
                                    != AppOpsManager.MODE_ALLOWED) {
                                runInBackgroundDisabled(uid);
                            }
                        }
                    }
                });
    }
```

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

```
public Lifecycle(Context context) {
            super(context);
            mService = new ActivityManagerService(context);
        }
```

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，震动管理服务等。

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。

```
public class ActivityManagerService extends IActivityManager.Stub
        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 >= RSS >= PSS >= 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，是是直接调用Activity类的方法，时间是这样子的，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 setIsForeground: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

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

```
/**
 * An entry in the history stack, representing an activity.
 */
final class ActivityRecord extends ConfigurationContainer implements AppWindowContainerListener {
 final ActivityManagerService service; // owner
        final IApplicationToken.Stub appToken; // window manager token
        AppWindowContainerController mWindowContainerController;
        final ActivityInfo info; // all about me
        final ApplicationInfo appInfo; // information about activity's app

        //省略其他成员变量

        //ActivityRecord所在的TaskRecord
        private TaskRecord task;        // the task this is in.

        //构造方法，需要传递大量信息
        ActivityRecord(ActivityManagerService _service, ProcessRecord _caller, int _launchedFromPid,
                       int _launchedFromUid, String _launchedFromPackage, Intent _intent, String _resolvedType,
                       ActivityInfo aInfo, Configuration _configuration,
                       com.android.server.am.ActivityRecord _resultTo, String _resultWho, int _reqCode,
                       boolean _componentSpecified, boolean _rootVoiceInteraction,
                       ActivityStackSupervisor supervisor, ActivityOptions options,
                       com.android.server.am.ActivityRecord sourceRecord) {

        }
}
```

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

```
private int startActivity(IApplicationThread caller, Intent intent, Intent ephemeralIntent,
            String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,
            IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
            IBinder resultTo, String resultWho, int requestCode, int callingPid, int callingUid,
            String callingPackage, int realCallingPid, int realCallingUid, int startFlags,
            SafeActivityOptions options,
            boolean ignoreTargetSecurity, boolean componentSpecified, ActivityRecord[] outActivity,
            TaskRecord inTask, boolean allowPendingRemoteAnimationRegistryLookup) {

        ActivityRecord r = new ActivityRecord(mService, callerApp, callingPid, callingUid,
                callingPackage, intent, resolvedType, aInfo, mService.getGlobalConfiguration(),
                resultRecord, resultWho, requestCode, componentSpecified, voiceSession != null,
                mSupervisor, checkedOptions, sourceRecord);

}
```

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

```
class TaskRecord extends ConfigurationContainer implements TaskWindowContainerListener {

final int taskId;       //任务ID
final ArrayList<ActivityRecord> mActivities;   //使用一个ArrayList来保存所有的ActivityRecord
private ActivityStack mStack;   //TaskRecord所在的ActivityStack

 */
    TaskRecord(ActivityManagerService service, int _taskId, Intent _intent,
            Intent _affinityIntent, String _affinity, String _rootAffinity,
            ComponentName _realActivity, ComponentName _origActivity, boolean _rootWasReset,
            boolean _autoRemoveRecents, boolean _askedCompatMode, int _userId,
            int _effectiveUid, String _lastDescription, ArrayList<ActivityRecord> activities,
            long lastTimeMoved, boolean neverRelinquishIdentity,
```

```
            TaskDescription _lastTaskDescription, int taskAffiliation, int
prevTaskId,
            int nextTaskId, int taskAffiliationColor, int callingUid, String
callingPackage,
            int resizeMode, boolean supportsPictureInPicture, boolean
_realActivitySuspended,
            boolean userSetupComplete, int minWidth, int minHeight) {

    }

//添加Activity到顶部
        void addActivityToTop(com.android.server.am.ActivityRecord r) {
            addActivityAtIndex(mActivities.size(), r);
        }

        //添加Activity到指定的索引位置
        void addActivityAtIndex(int index, ActivityRecord r) {
            //...

            r.setTask(this);//为ActivityRecord设置TaskRecord，就是这里建立的联系

            //...

            index = Math.min(size, index);
            mActivities.add(index, r);//添加到mActivities

            //...
  }
}

```

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

```
class ActivityStarter {

        private int setTaskFromReuseOrCreateNewTask(TaskRecord
taskToAffiliate, int preferredLaunchStackId, ActivityStack topStack) {
            mTargetStack = computeStackFocus(mStartActivity, true,
mLaunchBounds, mLaunchFlags, mOptions);

            if (mReuseTask == null) {
                //创建一个createTaskRecord，实际上是调用ActivityStack里面的
createTaskRecord（）方法，ActivityStack下面会讲到
                final TaskRecord task = mTargetStack.createTaskRecord(

 mSupervisor.getNextTaskIdForUserLocked(mStartActivity.userId),
                        mNewTaskInfo != null ? mNewTaskInfo :
mStartActivity.info,


```

```
                        mNewTaskIntent != null ? mNewTaskIntent : mIntent,
mVoiceSession,
                        mVoiceInteractor, !mLaunchTaskBehind /* toTop */,
mStartActivity.mActivityType);

                //其他代码略
            }
        }
    }
```

ActivityStack

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

```
class ActivityStack<T extends StackWindowController> extends
ConfigurationContainer
        implements StackWindowListener {

/**
     * The back history of all previous (and possibly still
     * running) activities.  It contains #TaskRecord objects.
     */
    private final ArrayList<TaskRecord> mTaskHistory = new ArrayList<>();//
使用一个ArrayList来保存TaskRecord

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

    ActivityStack(ActivityDisplay display, int stackId,
ActivityStackSupervisor supervisor,
            int windowingMode, int activityType, boolean onTop) {

    }
TaskRecord createTaskRecord(int taskId, ActivityInfo info, Intent intent,
                                    IVoiceInteractionSession voiceSession,
IVoiceInteractor voiceInteractor,
                                    boolean toTop, int type) {

            //创建一个task
            TaskRecord task = new TaskRecord(mService, taskId, info, intent,
voiceSession, voiceInteractor, type);

            //将task添加到ActivityStack中去
            addTask(task, toTop, "createTaskRecord");

            //其他代码略

            return task;
        }

        //添加Task
        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

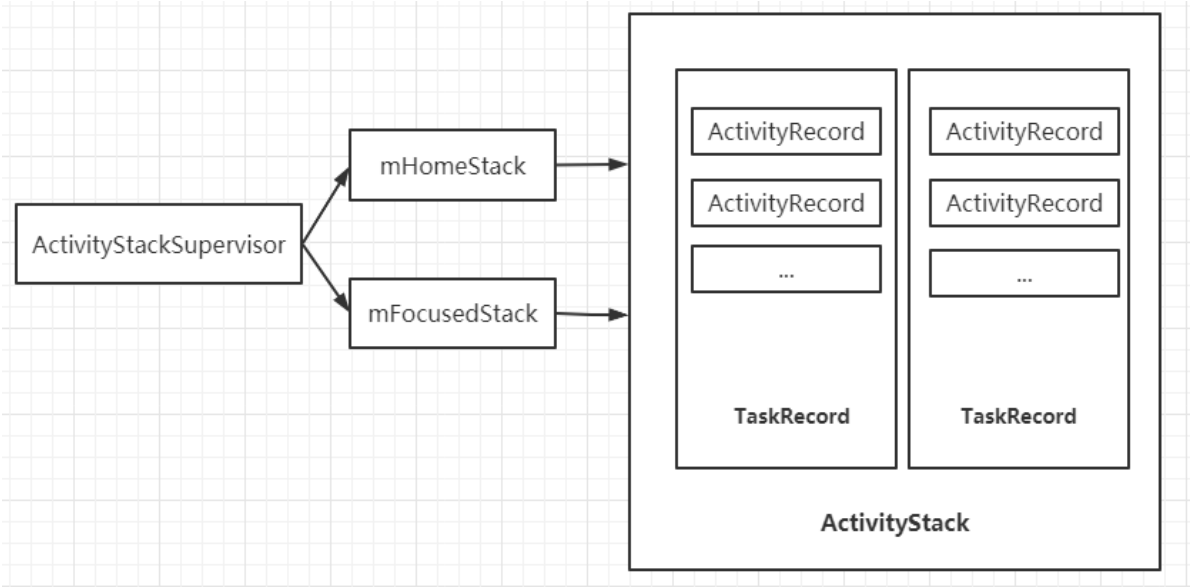
ActivityStackSupervisor

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

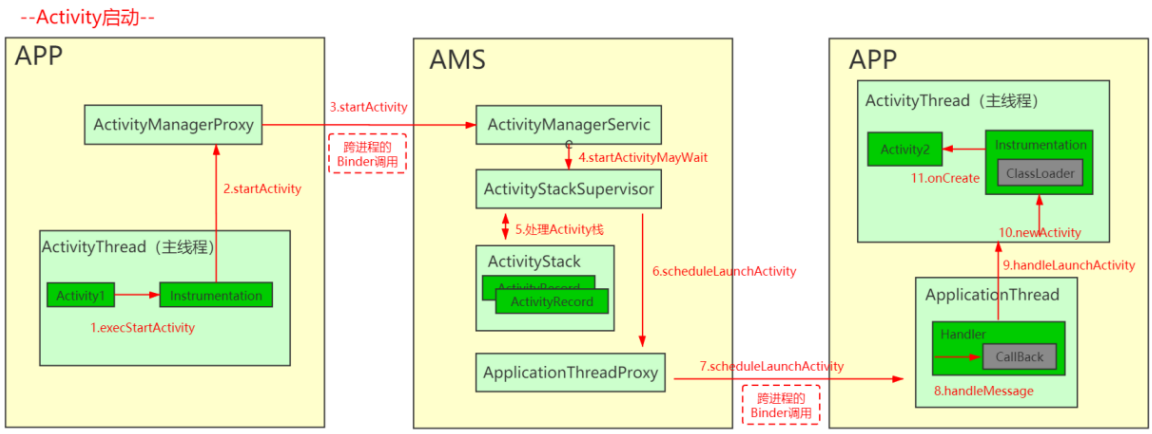
frameworks/base/services/core/java/com/android/server/am/ActivityStackSupervisor.java

```
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      }
```

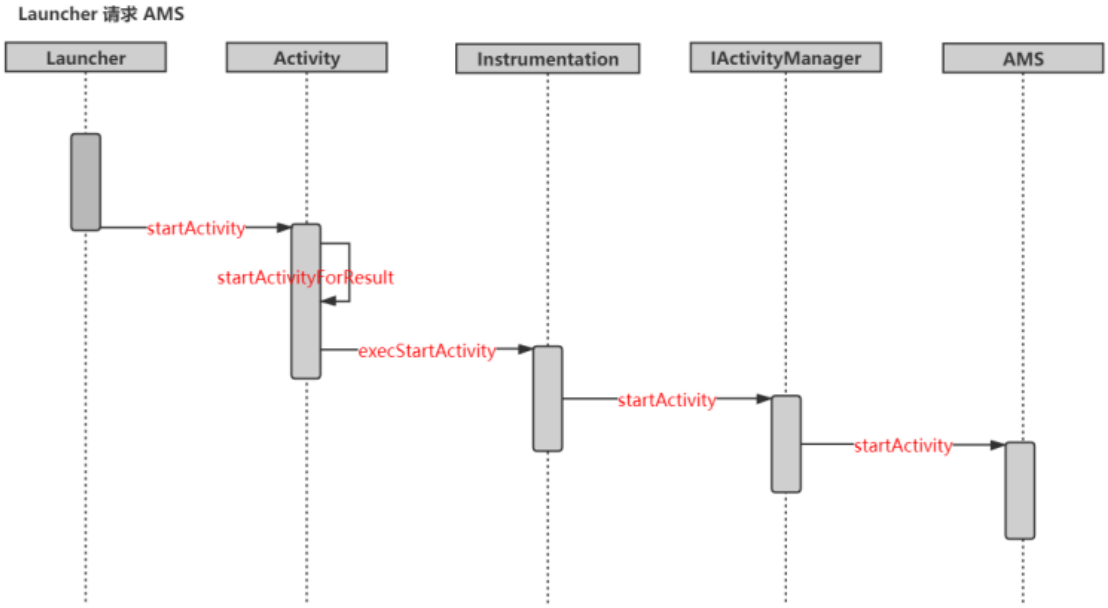
ActivityStackSupervisor内部有两个不同的ActivityStack对象：mHomeStack、mFocusedStack，用来管理不同的任务。
ActivityStackSupervisor内部包含了创建ActivityStack对象的方法。
AMS初始化时会创建一个ActivityStackSupervisor对象



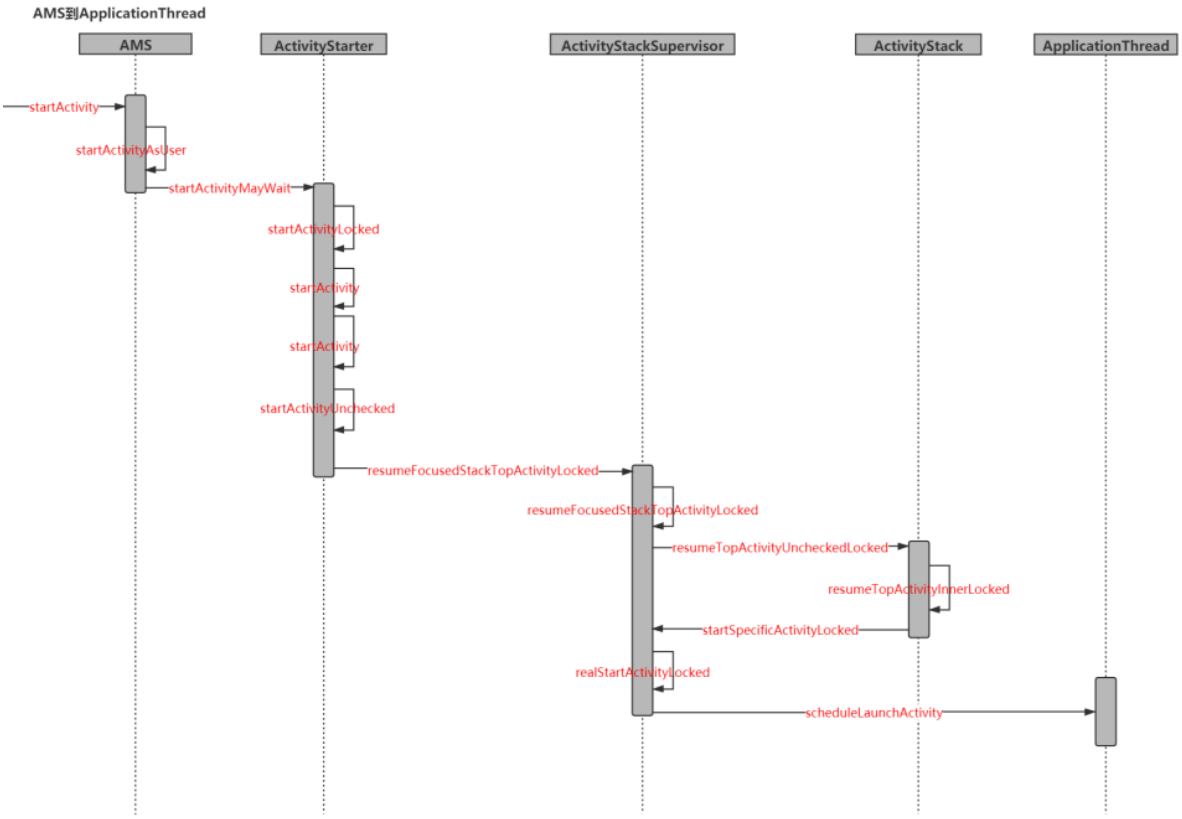
Activity启动流程相关



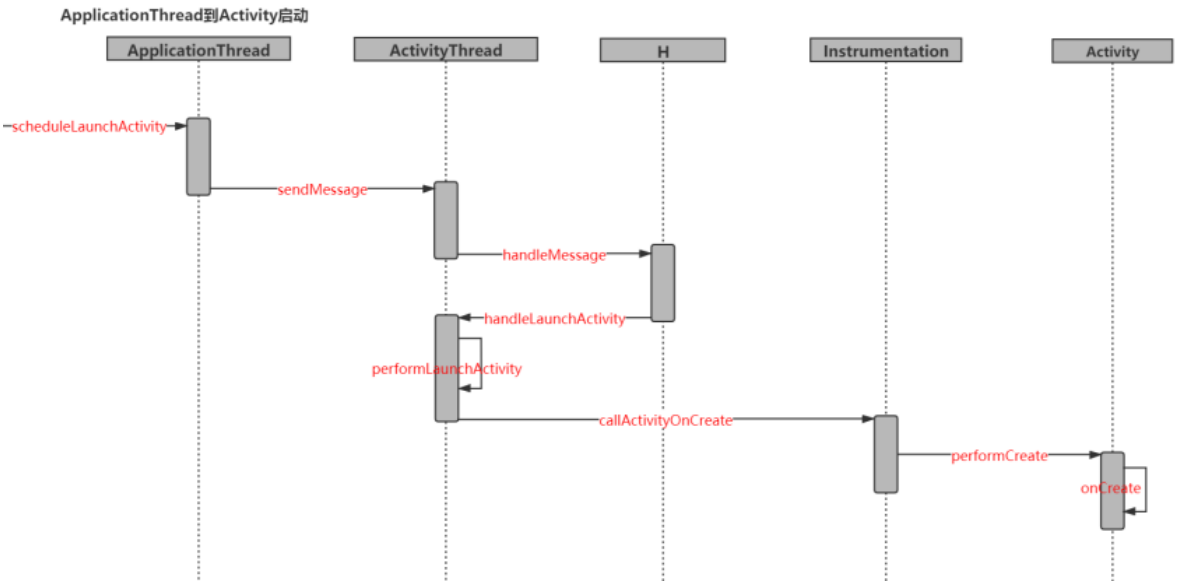
Launcher请求AMS阶段



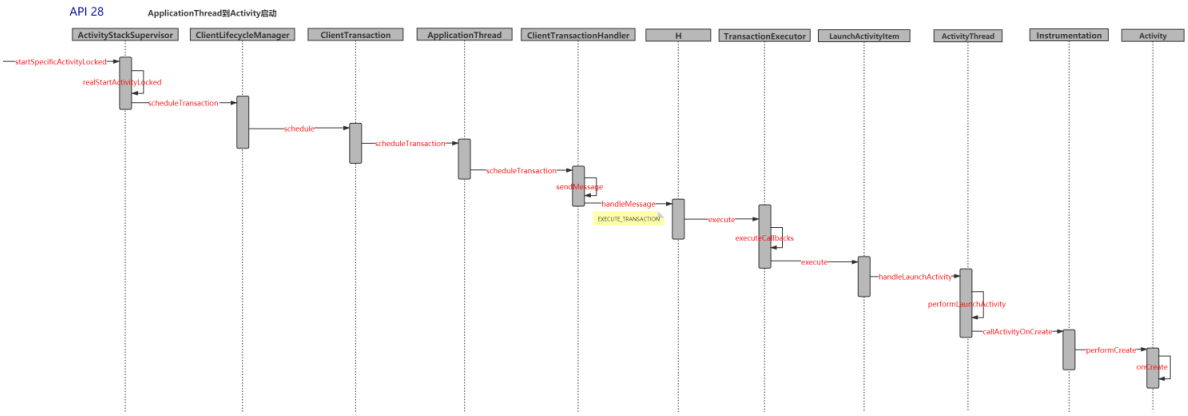
AMS到ApplicationThread阶段



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:2049)
        at android.app.Instrumentation.execStartActivity(Instrumentation.java:1709)
        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);
```

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

我们来看下是如何检查的

```
 public static void checkStartActivityResult(int res, Object intent) {
        if (!ActivityManager.isStartResultFatalError(res)) {
            return;
        }
    switch (res) {
        case ActivityManager.START_INTENT_NOT_RESOLVED:
        case ActivityManager.START_CLASS_NOT_FOUND:
            //3. 这里我们找到了报错的地方，原来是res结果为START_INTENT_NOT_RESOLVED,
            //START_CLASS_NOT_FOUND就会报这个错误
            if (intent instanceof Intent && ((Intent)intent).getComponent() != null)
                throw new ActivityNotFoundException(
                        "Unable to find explicit activity class "
                        + ((Intent)intent).getComponent().toShortString()
                        + "; have you declared this activity in your AndroidManifest.xml?");
            throw new ActivityNotFoundException(
                    "No Activity found to handle " + intent);
        case ActivityManager.START_PERMISSION_DENIED:
            throw new SecurityException("Not allowed to start activity "
                    + intent);
        case ActivityManager.START_FORWARD_AND_REQUEST_CONFLICT:
            throw new AndroidRuntimeException(
                    "FORWARD_RESULT_FLAG used while also requesting a result");
        case ActivityManager.START_NOT_ACTIVITY:
            throw new IllegalArgumentException(
                    "PendingIntent is not an activity");
        case ActivityManager.START_NOT_VOICE_COMPATIBLE:
            throw new SecurityException(
                    "Starting under voice control not allowed for: " + intent);
        case ActivityManager.START_VOICE_NOT_ACTIVE_SESSION:
            throw new IllegalStateException(
                    "Session calling startVoiceActivity does not match active session");
        case ActivityManager.START_VOICE_HIDDEN_SESSION:
            throw new IllegalStateException(
```

```
                    "Cannot start voice activity on a hidden session");
        case ActivityManager.START_ASSISTANT_NOT_ACTIVE_SESSION:
            throw new IllegalStateException(
                    "Session calling startAssistantActivity does not match active session");
        case ActivityManager.START_ASSISTANT_HIDDEN_SESSION:
            throw new IllegalStateException(
                    "Cannot start assistant activity on a hidden session");
        case ActivityManager.START_CANCELED:
            throw new AndroidRuntimeException("Activity could not be started for "
                    + intent);
        default:
            throw new AndroidRuntimeException("Unknown error code "
                    + res + " when starting " + intent);
    }
}
```

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

```
  private int startActivity(IApplicationThread caller, Intent intent, Intent ephemeralIntent,
            String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,
            IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
            IBinder resultTo, String resultWho, int requestCode, int callingPid, int callingUid,
            String callingPackage, int realCallingPid, int realCallingUid, int startFlags,
            SafeActivityOptions options,
            boolean ignoreTargetSecurity, boolean componentSpecified, ActivityRecord[] outActivity,
            TaskRecord inTask, boolean allowPendingRemoteAnimationRegistryLookup) {
        int err = ActivityManager.START_SUCCESS;
        ...
        //接下来开始做一些校验判断
        if (err == ActivityManager.START_SUCCESS && intent.getComponent() == null) {
            // We couldn't find a class that can handle the given Intent.
            // That's the end of that!
            err = ActivityManager.START_INTENT_NOT_RESOLVED;//从Intent中无法找到相应的Component
        }

        if (err == ActivityManager.START_SUCCESS && aInfo == null) {
            // We couldn't find the specific class specified in the Intent.
            // Also the end of the line.
            err = ActivityManager.START_CLASS_NOT_FOUND;//从Intent中无法找到相应的ActivityInfo
        }
        ...
        if (err != START_SUCCESS) {//不能成功启动了，返回err
            if (resultRecord != null) {
                resultStack.sendActivityResultLocked(
                        -1, resultRecord, resultWho, requestCode, RESULT_CANCELED, null);
            }
            SafeActivityOptions.abort(options);
            return err;
        }
        ...
```

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

}
```

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

```
private int startActivityMayWait(IApplicationThread caller, int callingUid,
            String callingPackage, Intent intent, String resolvedType,
            IVoiceInteractionSession voiceSession, IVoiceInteractor
voiceInteractor,
            IBinder resultTo, String resultWho, int requestCode, int
startFlags,
            ProfilerInfo profilerInfo, WaitResult outResult,
            Configuration globalConfig, SafeActivityOptions options, boolean
ignoreTargetSecurity,
            int userId, TaskRecord inTask, String reason,
            boolean allowPendingRemoteAnimationRegistryLookup) {
        // Refuse possible leaked file descriptors
        ...
        intent = new Intent(intent);
        if (componentSpecified
                && !(Intent.ACTION_VIEW.equals(intent.getAction()) &&
intent.getData() == null)
                &&
!Intent.ACTION_INSTALL_INSTANT_APP_PACKAGE.equals(intent.getAction())
                &&
!Intent.ACTION_RESOLVE_INSTANT_APP_PACKAGE.equals(intent.getAction())
                && mService.getPackageManagerInternalLocked()

.isInstantAppInstallerComponent(intent.getComponent())) {
            // intercept intents targeted directly to the ephemeral
installer the
            // ephemeral installer should never be started with a raw
Intent; instead
            // adjust the intent so it looks like a "normal" instant app
launch
            intent.setComponent(null /*component*/);
            componentSpecified = false;
        }

        ResolveInfo rInfo = mSupervisor.resolveIntent(intent, resolvedType,
userId,
                0 /* matchFlags */,
                        computeResolveFilterUid(
```

```
                                callingUid, realCallingUid,
mRequest.filterCallingUid));
        if (rInfo == null) {
            UserInfo userInfo = mSupervisor.getUserInfo(userId);
            if (userInfo != null && userInfo.isManagedProfile()) {
                // Special case for managed profiles, if attempting to
launch non-cryto aware
                // app in a locked managed profile from an unlocked parent
allow it to resolve
                // as user will be sent via confirm credentials to unlock
the profile.
                UserManager userManager =
UserManager.get(mService.mContext);
                boolean profileLockedAndParentUnlockingOrUnlocked = false;
                long token = Binder.clearCallingIdentity();
                try {
                    UserInfo parent = userManager.getProfileParent(userId);
                    profileLockedAndParentUnlockingOrUnlocked = (parent !=
null)
                            &&
userManager.isUserUnlockingOrUnlocked(parent.id)
                            &&
!userManager.isUserUnlockingOrUnlocked(userId);
                } finally {
                    Binder.restoreCallingIdentity(token);
                }
                if (profileLockedAndParentUnlockingOrUnlocked) {
                    rInfo = mSupervisor.resolveIntent(intent, resolvedType,
userId,
                            PackageManager.MATCH_DIRECT_BOOT_AWARE
                                    |
PackageManager.MATCH_DIRECT_BOOT_UNAWARE,
                            computeResolveFilterUid(
                                    callingUid, realCallingUid,
mRequest.filterCallingUid));
                }
            }
        }
        // Collect information about the target of the Intent.
        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
        ...
            final ActivityRecord[] outRecord = new ActivityRecord[1];
            int res = startActivity(caller, intent, ephemeralIntent,
resolvedType, aInfo, rInfo,
                    voiceSession, voiceInteractor, resultTo, resultWho,
requestCode, callingPid,
                    callingUid, callingPackage, realCallingPid,
realCallingUid, startFlags, options,
                    ignoreTargetSecurity, componentSpecified, outRecord,
inTask, reason,
                    allowPendingRemoteAnimationRegistryLookup);
            ...
            return res;
        }
    }
```

我们看下aInfo哪儿来的

```
 ActivityInfo resolveActivity(Intent intent, ResolveInfo rInfo, int
startFlags,
            ProfilerInfo profilerInfo) {
        final ActivityInfo aInfo = rInfo != null ? rInfo.activityInfo :
null;
        if (aInfo != null) {
            // Store the found target back into the intent, because now that
            // we have it we never want to do this again.  For example, if
the
            // user navigates back to this point in the history, we should
            // always restart the exact same activity.
            intent.setComponent(new ComponentName(
                    aInfo.applicationInfo.packageName, aInfo.name));

            // Don't debug things in the system process
            ...
        }
        return aInfo;
    }
```

发现是从rInfo来的

```
 ResolveInfo resolveIntent(Intent intent, String resolvedType, int userId,
int flags,
            int filterCallingUid) {
        synchronized (mService) {
            try {
                ...
                final long token = Binder.clearCallingIdentity();
                try {
                    return
mService.getPackageManagerInternalLocked().resolveIntent(
                            intent, resolvedType, modifiedFlags, userId,
true, filterCallingUid);
                } finally {
                    Binder.restoreCallingIdentity(token);
                }
            ...
        }
    }
```

那么rInfo怎么获取的呢?

```
PackageManagerInternal getPackageManagerInternalLocked() {
        if (mPackageManagerInt == null) {
            mPackageManagerInt =
LocalServices.getService(PackageManagerInternal.class);
        }
        return mPackageManagerInt;
    }
```

具体实现类是PackageManagerService

```
@Override
   public ResolveInfo resolveIntent(Intent intent, String resolvedType,
           int flags, int userId) {
       return resolveIntentInternal(intent, resolvedType, flags, userId,
false,
               Binder.getCallingUid());
   }
```

再看resolveIntentInternal

```
private ResolveInfo resolveIntentInternal(Intent intent, String
resolvedType,
            int flags, int userId, boolean resolveForStart, int
filterCallingUid) {
        try {
           ...
            //获取ResolveInfo列表
            final List<ResolveInfo> query =
queryIntentActivitiesInternal(intent, resolvedType,
                    flags, filterCallingUid, userId, resolveForStart, true
/*allowDynamicSplits*/);
            Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
            //找出最好的返回
            final ResolveInfo bestChoice =
                    chooseBestActivity(intent, resolvedType, flags, query,
userId);
            return bestChoice;
        } finally {
            Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
        }
    }
```

再看 queryIntentActivitiesInternal

```
private @NonNull List<ResolveInfo> queryIntentActivitiesInternal(Intent
intent,
            String resolvedType, int flags, int filterCallingUid, int
userId,
            boolean resolveForStart, boolean allowDynamicSplits) {
        ...
        if (comp != null) {
            final List<ResolveInfo> list = new ArrayList<ResolveInfo>(1);
            final ActivityInfo ai = getActivityInfo(comp, flags, userId);
            if (ai != null) {
                // When specifying an explicit component, we prevent the
activity from being
                // used when either 1) the calling package is normal and the
activity is within
                // an ephemeral application or 2) the calling package is
ephemeral and the
                // activity is not visible to ephemeral applications.
                final boolean matchInstantApp =
                        (flags & PackageManager.MATCH_INSTANT) != 0;
                final boolean matchVisibleToInstantAppOnly =
                        (flags &
PackageManager.MATCH_VISIBLE_TO_INSTANT_APP_ONLY) != 0;
```

```
                final boolean matchExplicitlyVisibleOnly =
                        (flags &
PackageManager.MATCH_EXPLICITLY_VISIBLE_ONLY) != 0;
                final boolean isCallerInstantApp =
                        instantAppPkgName != null;
                final boolean isTargetSameInstantApp =
                        comp.getPackageName().equals(instantAppPkgName);
                final boolean isTargetInstantApp =
                        (ai.applicationInfo.privateFlags
                                & ApplicationInfo.PRIVATE_FLAG_INSTANT) !=
0;
                final boolean isTargetVisibleToInstantApp =
                        (ai.flags &
ActivityInfo.FLAG_VISIBLE_TO_INSTANT_APP) != 0;
                final boolean isTargetExplicitlyVisibleToInstantApp =
                        isTargetVisibleToInstantApp
                        && (ai.flags &
ActivityInfo.FLAG_IMPLICITLY_VISIBLE_TO_INSTANT_APP) == 0;
                final boolean isTargetHiddenFromInstantApp =
                        !isTargetVisibleToInstantApp
                        || (matchExplicitlyVisibleOnly &&
!isTargetExplicitlyVisibleToInstantApp);
                final boolean blockResolution =
                        !isTargetSameInstantApp
                        && ((!matchInstantApp && !isCallerInstantApp &&
isTargetInstantApp)
                                || (matchVisibleToInstantAppOnly &&
isCallerInstantApp
                                        && isTargetHiddenFromInstantApp));
                if (!blockResolution) {
                    final ResolveInfo ri = new ResolveInfo();
                    ri.activityInfo = ai;
                    list.add(ri);
                }
            }
            return applyPostResolutionFilter(
                    list, instantAppPkgName, allowDynamicSplits,
filterCallingUid, resolveForStart,
                    userId, intent);
        }

    }
```

原来是从getActivityInfo获取的

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

```
private ActivityInfo getActivityInfoInternal(ComponentName component, int
flags,
            int filterCallingUid, int userId) {
        if (!sUserManager.exists(userId)) return null;
```

```
        flags = updateFlagsForComponent(flags, userId, component);

        if (!isRecentsAccessingChildProfiles(Binder.getCallingUid(),
userId)) {

 mPermissionManager.enforceCrossUserPermission(Binder.getCallingUid(),
userId,
                    false /* requireFullPermission */, false /* checkShell
*/, "get activity info");
        }

        synchronized (mPackages) {
            //关键点
            PackageParser.Activity a =
mActivities.mActivities.get(component);

            if (DEBUG_PACKAGE_INFO) Log.v(TAG, "getActivityInfo " +
component + ": " + a);
            if (a != null && mSettings.isEnabledAndMatchLPr(a.info, flags,
userId)) {
                PackageSetting ps =
mSettings.mPackages.get(component.getPackageName());
                if (ps == null) return null;
                if (filterAppAccessLPr(ps, filterCallingUid, component,
TYPE_ACTIVITY, userId)) {
                    return null;
                }
                //关键点
                return PackageParser.generateActivityInfo(
                        a, flags, ps.readUserState(userId), userId);
            }
            if (mResolveComponentName.equals(component)) {
                return PackageParser.generateActivityInfo(
                        mResolveActivity, flags, new PackageUserState(),
userId);
            }
        }
        return null;
    }
```