Android ActivityService 相关的原理

deepwaterooo

August 29, 2022

Contents

1 android 内核剖析学习笔记：AMS（ActivityManagerService）内部原理和工作机制

1.1 一、ActivityManagerService 提供的主要功能：

- （1）统一调度各应用程序的 Activity
- （2）内存管理
- （3）进程管理

1.2 二、启动一个 Activity 的方式有以下几种：

- （1）在应用程序中调用 startActivity 启动指定的 Activity
- （2）在 Home 程序中单击一个应用图标，启动新的 Activity
- （3）按“Back”键，结束当前 Activity，返回到上一个 Activity
- （4）长按“Home”键，显示出当前正在运行的程序列表，从中选择一个启动
- 这四种启动方式的主体处理流程都会按照第一种启动方式运行，后面三种方式只是在前端消息处理上各有不同。

1.3 三、进程数据类 ProcessRecord

- 该类的源代码在 ~\am 路径下。
- 一般情况下，一个 APK 文件运行时会对应一个进程，ProcessRecord 用来记录一个进程中的相关信息，主要包含的变量有：

1.3.1 （1）进程文件信息：与该进程对应的 APK 文件的内部信息，如

```
final ApplicationInfo info; // all about the first app in the process
final String processName;   // name of the process
final ArrayMap<String, ProcessStats.ProcessState> pkgList
    = new ArrayMap<String, ProcessStats.ProcessState>();   //保存进程中所有 APK 文件包名
```

1.3.2 （2）进程的内存状态信息：用于 Linux 系统的 out of memory(OOM) 情况的处理，当发生内存紧张时，Linux 系统会根据进程的内存状态信息杀掉低优先级的进程，包括的变量有

```
int maxAdj;                 // Maximum OOM adjustment for this process
int curRawAdj;              // Current OOM unlimited adjustment for this process
int setRawAdj;              // Last set OOM unlimited adjustment for this process
int curAdj;                 // Current OOM adjustment for this proce
int setAdj;                 // Last set OOM adjustment for this process
```

- 变量中 Adj 的含义是调整值（adjustment）

1.3.3 （3）进程中包含的 Activity、Provider、Service 等，如下

```
final ArrayList<ActivityRecord> activities = new ArrayList<ActivityRecord>();
final ArraySet<ServiceRecord> services = new ArraySet<ServiceRecord>();
final ArraySet<ServiceRecord> executingServices = new ArraySet<ServiceRecord>();
final ArraySet<ConnectionRecord> connections = new ArraySet<ConnectionRecord>();
final ArraySet<ReceiverList> receivers = new ArraySet<ReceiverList>();
final ArrayMap<String, ContentProviderRecord> pubProviders = new ArrayMap<String,            ContentProviderRecord>();
final ArrayList<ContentProviderConnection> conProviders = new ArrayList<ContentProviderConnection>();
```

1.4 四、ActivityRecord 数据类（Android 2.3 以前版本叫 HistoryRecord 类）

- ActivityManagerService 使用 ActivityRecord 数据类来保存每个 Activity 的信息，ActivityRecord 类基于 IApplicationToken.Stub 类，也是一个 Binder, 所以可以被 IPC 调用。
- 主要包含的变量有：

1.4.1 （1）环境信息：Activity 的工作环境，比如进程名称、文件路径、数据路径、图标、主题等，这些信息一般是固定的，比如以下变量

```
final String packageName; // the package implementing intent's component
final String processName; // process where this component wants to run
final String baseDir;   // where activity source (resources etc) located
final String resDir;   // where public activity source (public resources etc) located
final String dataDir;   // where activity data should go
int theme;              // resource identifier of activity's theme.
int realTheme;          // actual theme resource we will use, never 0.
```

1.4.2 （2）运行状态数据信息：如 idle、stop、finishing 等，一般为 boolean 类型，如下

```
boolean haveState;      // have we gotten the last activity state?
boolean stopped;        // is activity pause finished?
boolean delayedResume;  // not yet resumed because of stopped app switches?
boolean finishing;      // activity in pending finish list?
boolean configDestroy;  // need to destroy due to config change?
```

1.5 五、TaskRecord 类

- ActivityManagerService 中使用任务的概念来确保 Activity 启动和退出的顺序。
- TaskRecord 中的几个重要变量如下：

```
final int taskId;       // 每个任务的标识.
Intent intent;          // 创建该任务时对应的 intent
int numActivities;   //该任务中的 Activity 数目
final ArrayList<ActivityRecord> mActivities = new ArrayList<ActivityRecord>();  //按照出现的先后顺序列出该任务中的所有 Activit
```

1.6 六、ActivityManagerService 中一些重要的与调度相关的变量

- （1）记录最近启动的 Activity，如果 RAM 容量较小，则记录的最大值为 10 个，否则为 20 个，超过该值后，Ams 会舍弃最早记录的 Activity

```
static final int MAX_RECENT_TASKS = ActivityManager.isLowRamDeviceStatic() ? 10 : 20;
```

- （2）当 Ams 通知应用程序启动（Launch）某个 Activity 时，如果超过 10s，Ams 就会放弃

```
static final int PROC_START_TIMEOUT = 10*1000;
```

- （3）当 Ams 启动某个客户进程后，客户进程必须在 10s 之内报告 Ams 自己已经启动，否则 Ams 会认为指定的客户进程不存在

```
static final int PROC_START_TIMEOUT = 10*1000;
```

- （4）等待序列：
  - 当 Ams 内部还没有准备好时，如果客户进程请求启动某个 Activity，那么会被暂时保存到该变量中，

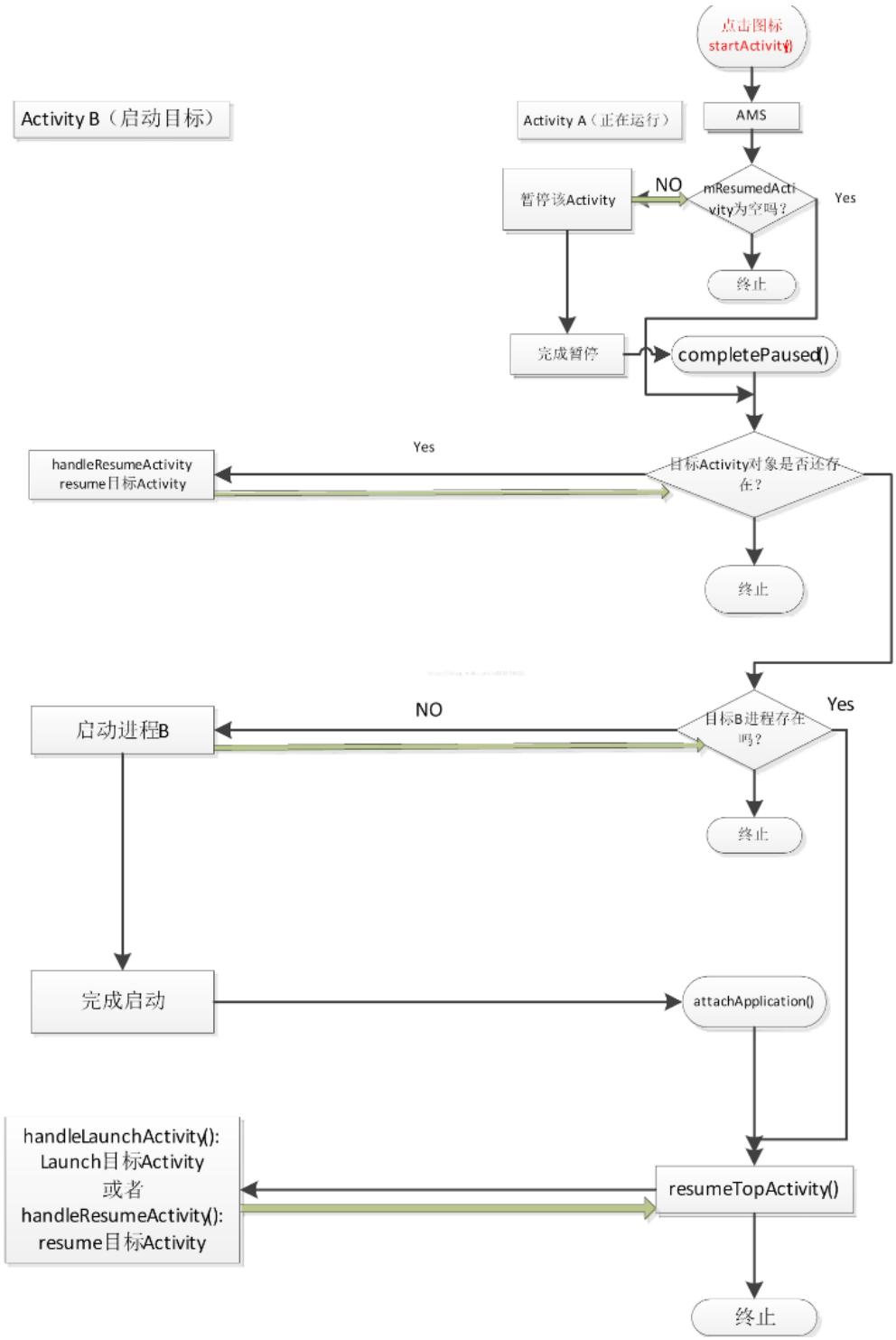
```
final ArrayList<PendingActivityLaunch> mPendingActivityLaunches
    = new ArrayList<PendingActivityLaunch>();
```

- （5）优先启动，其次再停止。进程 A1 包含两个 Activity，启动顺序为 A1->A2，当用户请求启动 A2 时，如果 A1 正在运行，Ams 会先暂停 A1，然后启动 A2，当 A2 启动后再停止 A1。

```
private final ArrayList<TaskRecord> mRecentTasks = new ArrayList<TaskRecord>();
```

1.7 七、startActivity() 的流程 (后面有详尽较好的讲解)

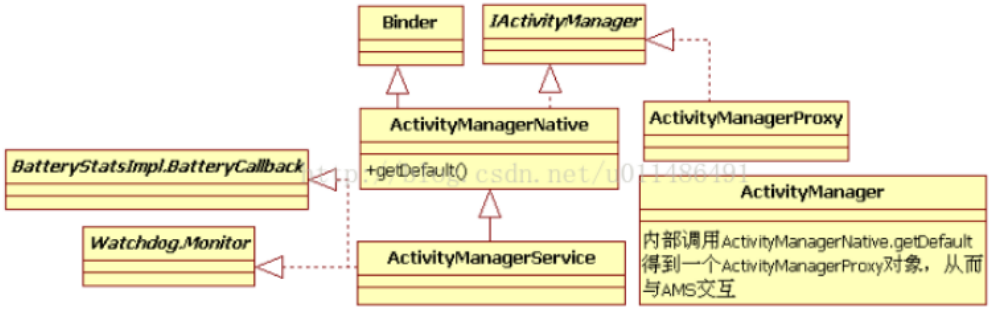
- 当用户单击某个应用图标后，执行程序会在该图标的 onClick() 事件中调用 startActivity() 方法，该方法会调用 startActivityForResult()，在这个方法内部会调用 Instrumentation 对象的 executeStartActivity() 方法，每个 Activity 内部都有一个 Instrumentation 对象的引用，它就是一个管家，ActivityThread 要创建或者暂停某个 Activity 都是通过它实现的。
- 流程图如下所示：



2 ActivityManagerService 源码分析

- AMS 是 Android 系统服务中很重要的一个，他负责四大组件的启动、切换、调度、生命周期的管理等等，接下来我们根据 AMS 的启动来分析 AMS 的源码

- AMS 是 Android 中最核心的服务，主要负责系统中四大组件的启动、切换、调度及应用进程的管理和调度等工作，其职责与操作系统中的进程管理和调度模块相类似，因此它在 Android 中非常重要。
- AMS 比较复杂，本章将带领读者按五条不同的线来分析它：
  - 1. 第一条线：同其他服务一样，将分析 SystemServer 中 AMS 的调用轨迹。
  - 2. 第二条线：以 am 命令启动一个 Activity 为例，分析应用进程的创建、Activity 的启动，以及它们和 AMS 之间的交互等知识。
  - 3. 第三条线和第四条线：分别以 Broadcast 和 Service 为例，分析 AMS 中 Broadcast 和 Service 的相关处理流程。
  - 4. 第五条线：以一个 Crash 的应用进程为出发点，分析 AMS 如何打理该应用进程的身后事。
- AMS 的家族图谱：



- AMS 由 ActivityManagerNative（以后简称 AMN）类派生，并实现 Watchdog.Monitor 和 BatteryStatsImpl.BatteryCallback 接口。而 AMN 由 Binder 派生，实现了 IActivityManager 接口。
- 客户端使用 ActivityManager 类。由于 AMS 是系统核心服务，很多 API 不能开放供客户端使用，所以设计者没有让 ActivityManager 直接加入 AMS 家族。在 ActivityManager 类内部通过调用 AMN 的 getDefault 函数得到一个 ActivityManagerProxy 对象，通过它可与 AMS 通信。

2.1 SystemServer 启动 AMS

- ActivityManagerService 是在 SystemServer.java 中启动并注册的：

```
private void startBootstrapServices() {

    // 启动 AMS，见小节 2.1
    mActivityManagerService = mSystemServiceManager.startService(
        ActivityManagerService.Lifecycle.class).getService();

    // 设置 AMS
    mActivityManagerService.setSystemServiceManager(mSystemServiceManager);

    // 安装 App 安装器
    mActivityManagerService.setInstaller(installer);

    // 电源管理已经开启，在 AMS 中初始化 PM，见小节 3
    Trace.traceBegin(Trace.TRACE_TAG_SYSTEM_SERVER, "InitPowerManagement");
    mActivityManagerService.initPowerManagement();
    Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);

    // 设置应用实例，在系统进程开始的时候，见小节 4
    mActivityManagerService.setSystemProcess();
}
```

- 看源码可以看到，startBootstrapServices 中启动并注册了很多其他的服务，比如：PowerManagerService, DisplayManagerService, LightsService, PackageManagerService, UserManagerService, SensorService(native)，这写服务彼此之间有依赖，所以都放在 startBootstrapServices 方法里面

2.2 注册并启动

2.2.1 startService: 文件：SystemServiceManager.java

```
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);
        // 创建服务: 这个服务的类必须是 SystemService 的子类，于是我们需要传入带有 ActivityManagerService 的 Lifecycle
        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);
        }
        // 注册
        mServices.add(service);
        // 开始服务
        try {
            service.onStart();
        } catch (RuntimeException ex) {
            throw new RuntimeException("Failed to start service " + name
                                       + ": onStart threw an exception", ex);
        }
        return service;
    } finally {
        Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);
    }
}
```

- 可以看出这方法的作用是创建并且开始一个服务，但是这个服务的类必须是 SystemService 的子类，于是我们需要传入带有 ActivityManagerService 的 Lifecycle

2.2.2 Lifecycle: 文件：SystemServiceManager.java

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

    public Lifecycle(Context context) {
        super(context);
        // 构建一个新的 AMS，见小节 2.3
        mService = new ActivityManagerService(context);
    }
    @Override public void onStart() {
        // 开始服务，2.1 中的 service.onStart() 调用的就是它，见小节 2.4
        mService.start();
    }
    public ActivityManagerService getService() {
        return mService;
    }
}
```

- 可以看出 Lifecycle 是继承 SystemService 的，并且在构造里构建了 AMS，接下来我们来看看 AMS 的构建函数

2.2.3 AMS 的构建函数

- 这个方法会在 main thread 中被唤醒，但是它需要通过各个 handers 和其他 thread 通信，所以要注意明确 looper。该构造函数，里面是初始化一些变量，及创建了一些线程

```
public ActivityManagerService(Context systemContext) {
    mContext = systemContext;
    mFactoryTest = FactoryTest.getMode();

    mSystemThread = ActivityThread.currentActivityThread();
    Slog.i(TAG, "Memory class: " + ActivityManager.staticGetMemoryClass());

    // 创建一个 mHandlerThread 线程，默认名是: ActivityManager，异步线程 [ServiceThread 与 HandlerThread 有什么不同 ？]
    mHandlerThread = new ServiceThread(TAG, android.os.Process.THREAD_PRIORITY_FOREGROUND, false /*allowIo*/);
    mHandlerThread.start();
    // 创建一个基于 ActivityManager 线程的 Handler
    mHandler = new MainHandler(mHandlerThread.getLooper());

    // 创建一个 UiHandler 线程，是异步线程吗？
    mUiHandler = new UiHandler();

    // 用单例的方式创建一个名叫 ActivityManager:kill 的线程，并且创建一个 killHandler
    if (sKillHandler == null) {
        sKillThread = new ServiceThread(TAG + ":kill",
                                        android.os.Process.THREAD_PRIORITY_BACKGROUND, true /* allowIo */);
        sKillThread.start();
        // 用于杀死进程
        sKillHandler = new KillHandler(sKillThread.getLooper());
    }

    // 构建一个可以延时 10 秒的前台广播队列
    mFgBroadcastQueue = new BroadcastQueue(this, mHandler,
                                           "foreground", BROADCAST_FG_TIMEOUT, false);
    // 构建一个可以延时 60 秒的普通广播队列 (一定全都是后台广播吗？)
    mBgBroadcastQueue = new BroadcastQueue(this, mHandler,
                                           "background", BROADCAST_BG_TIMEOUT, true);
    mBroadcastQueues[0] = mFgBroadcastQueue; // 不故道这个数组的定义是在哪里，反正长度 >= 2
    mBroadcastQueues[1] = mBgBroadcastQueue;

    mServices = new ActiveServices(this);
    mProviderMap = new ProviderMap(this);
    mAppErrors = new AppErrors(mContext, this);

    // 新建一个 data/system 目录
    File dataDir = Environment.getDataDirectory();
    File systemDir = new File(dataDir, "system");
    systemDir.mkdirs();

    // 创建一个 BatteryStatsService 类
    mBatteryStatsService = new BatteryStatsService(systemDir, mHandler);
    // 把最新的数据写入硬盘
    mBatteryStatsService.scheduleWriteToDisk();
    mOnBattery = DEBUG_POWER ? true
        : mBatteryStatsService.getActiveStatistics().getIsOnBattery();
    mBatteryStatsService.getActiveStatistics().setCallback(this);

    // 创建进程统计服务类，并新建一个 data/system/procstats 目录
    mProcessStats = new ProcessStatsService(this, new File(systemDir, "procstats"));
    // 创建一个应用权限检查类，新建一个 data/system/appops.xml 文件，并注册对应的回调接口
    mAppOpsService = new AppOpsService(new File(systemDir, "appops.xml"), mHandler);
    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);
                                                 }
                                             }
                                         }
                                     });
    mGrantFile = new AtomicFile(new File(systemDir, "urigrants.xml"));
```

```
    // 创建多用户控制器，user 0 是第一个，同时也是唯一开机过程中运行的用户
    mUserController = new UserController(this);

    // 获取 OpenGL 版本，如果没有找到，则默认为 0
    GL_ES_VERSION = SystemProperties.getInt("ro.opengles.version",
                                            ConfigurationInfo.GL_ES_VERSION_UNDEFINED);

    mTrackingAssociations = "1".equals(SystemProperties.get("debug.track-associations"));
    // 设置系统的一些默认配置信息
    mConfiguration.setToDefaults();
    mConfiguration.setLocales(LocaleList.getDefault());
    mConfigurationSeq = mConfiguration.seq = 1;

    // 初始化进程 CPU 跟踪器
    mProcessCpuTracker.init();
    // 解析/data/system/packages-compat.xml 文件，当设备屏幕大小不满足 APK 所需要的大小，
    // 则从 packages-compat.xml 都去尺寸，用兼容的方式运行
    mCompatModePackages = new CompatModePackages(this, systemDir, mHandler);

    // 根据 AMS 传入规则，过滤一些 Intent
    mIntentFirewall = new IntentFirewall(new IntentFirewallInterface(), mHandler);

    // 用来管理 Activity 栈: 应试就是当系统有多个任务栈时，它们多任务栈的管理者，用于管理系统下存在的多个任务栈
    mStackSupervisor = new ActivityStackSupervisor(this);
    // 解释怎样启动 Activity
    mActivityStarter = new ActivityStarter(this, mStackSupervisor);
    // 管理最近任务列表
    mRecentTasks = new RecentTasks(this, mStackSupervisor);

    // 创建一个统计 进程使用 CPU 情况 的线程，名叫 CpuTracker
    mProcessCpuThread = new Thread("CpuTracker") {
        @Override
        public void run() {
            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);
                }
            }
        }
    };
    // watchdog 添加对 AMS 的监控
    Watchdog.getInstance().addMonitor(this);
    Watchdog.getInstance().addThread(mHandler);
}
```

- 这个方法会在 main thread 中被唤醒，但是它需要通过各个 handers 和其他 thread 通信，所以要注意明确 looper。该构造函数，里面是初始化一些变量，及创建了一些线程，大部分我都进行了注释。

2.2.4 start: 文件：ActivityManagerService.java

```
private void start() {
    // 移除所有的进程组
    Process.removeAllProcessGroups();
    // 开始监控进程的 CPU 使用情况
    mProcessCpuThread.start();
    // 注册电池统计服务
    mBatteryStatsService.publish(mContext);
```

```
    // 注册应用权限检测服务
    mAppOpsService.publish(mContext);
    Slog.d("AppOps", "AppOpsService published");
    // 注册 LocalService 服务
    LocalServices.addService(ActivityManagerInternal.class, new LocalService());
}
```

- 启动 ProcessCpuThread，注册电池统计服务，应用权限检测服务和 LocalService，其中 LocalService 继承了 ActivityManagerInternal。
- 小结：创建 AMS，启动 AMS

2.3 初始化 PM: initPowerManagement: 文件:ActivityManagerService.java

```
public void initPowerManagement() {
    // Activity 堆栈管理器和电池统计服务初始化 PM
    mStackSupervisor.initPowerManagement();
    mBatteryStatsService.initPowerManagement();
    mLocalPowerManager = LocalServices.getService(PowerManagerInternal.class);
    PowerManager pm = (PowerManager)mContext.getSystemService(Context.POWER_SERVICE);
    mVoiceWakeLock = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "*voice*");
    // 该唤醒锁为不计数锁，即无论 acquire() 多少次，一次 release() 就可以解锁
    mVoiceWakeLock.setReferenceCounted(false);
}
```

- 小结：这主要是在 AMS 中初始化 PM

2.4 设置应用实例

2.4.1 setSystemProcess: 文件：ActivityManagerService.java

```
public void setSystemProcess() {
    try {
        // 以下都是想 ServiceManager 注册服务
        ServiceManager.addService(Context.ACTIVITY_SERVICE, this, true); // 注册 AMS 自己
        ServiceManager.addService(ProcessStats.SERVICE_NAME, mProcessStats); // 注册进程统计服务
        ServiceManager.addService("meminfo", new MemBinder(this)); // 注册内存信息的服务
        ServiceManager.addService("gfxinfo", new GraphicsBinder(this)); // 注册输出渲染信息的服务
        ServiceManager.addService("dbinfo", new DbBinder(this)); // 注册输出数据库信息的服务
        // MONITOR_CPU_USAGE 默认为 true
        if (MONITOR_CPU_USAGE) {
            ServiceManager.addService("cpuinfo", new CpuBinder(this)); // 输出进程使用 CPU 的情况
        }
        ServiceManager.addService("permission", new PermissionController(this)); // 注册权限管理
        ServiceManager.addService("processinfo", new ProcessInfoService(this)); // 注册进程信息

        // 查询名为 android 的应用信息
        ApplicationInfo info = mContext.getPackageManager().getApplicationInfo(
            "android", STOCK_PM_FLAGS | MATCH_SYSTEM_ONLY);
        // 调用 installSystemApplicationInfo ，见小节 4.2
        mSystemThread.installSystemApplicationInfo(info, getClass().getClassLoader());

        synchronized (this) {
            // 创建一个 ProcessRecord 对象 ，见小节 4.5
            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);
    }
}
```

2.4.2 installSystemApplicationInfo: 文件：ActivityThread.java

```
public void installSystemApplicationInfo(ApplicationInfo info, ClassLoader classLoader) {
    synchronized (this) {
        // 看 SystemService 中创建的 ContextIml 的 installSystemApplicationInfo，见小节 4.3
        getSystemContext().installSystemApplicationInfo(info, classLoader);
        // give ourselves a default profiler
        mProfiler = new Profiler();
    }
}
```

2.4.3 ContextIml.installSystemApplicationInfo: 文件：ContextImpl.java

```
void installSystemApplicationInfo(ApplicationInfo info, ClassLoader classLoader) {
    // 调用的是 LoadeApk 里面的 installSystemApplicationInfo，见小节 4.4
    mPackageInfo.installSystemApplicationInfo(info, classLoader);
}
```

2.4.4 LoadeApk.installSystemApplicationInfo: 文件：LoadeApk.java

```
void installSystemApplicationInfo(ApplicationInfo info, ClassLoader classLoader) {
    // 断言只有 packageName 为 android 才能使用
    assert info.packageName.equals("android");
    mApplicationInfo = info;
    mClassLoader = classLoader;
}
```

- 将 ApplicationInfo 加入到 LoadeApk 中，因为 SystemService 创建 LoadeApk 时，PKMS 并没有完成对手机中文件的解析

2.4.5 AMS 进程管理: 文件：ActivityManagerService.java

```
synchronized (this) {
    // 调用进程管理函数，见 4.6
    ProcessRecord app = newProcessRecordLocked(info, info.processName, false, 0);
    app.persistent = true;
    app.pid = MY_PID;
    app.maxAdj = ProcessList.SYSTEM_ADJ;
    // 将 SystemServer 对应的 ApplicationThread 保存到 ProcessRecord 中
    app.makeActive(mSystemThread.getApplicationThread(), mProcessStats);
    synchronized (mPidsSelfLocked) {
        // 根据 ProcessRecord 的 pid，将 ProcessRecord 存在 mPidsSelfLocked 中
        mPidsSelfLocked.put(app.pid, app);
    }
    updateLruProcessLocked(app, false, null);
    updateOomAdjLocked();
}
```

2.4.6 newProcessRecordLocked: 文件：ActivityManagerService.java

```
final ProcessRecord newProcessRecordLocked(ApplicationInfo info, String customProcess,
        boolean isolated, int isolatedUid) {
    String proc = customProcess != null ? customProcess : info.processName;
    BatteryStatsImpl stats = mBatteryStatsService.getActiveStatistics();
    final int userId = UserHandle.getUserId(info.uid);
    int uid = info.uid;
    // isolated 为 false
    if (isolated) {
            }
    // 创建一个进程记录对象，见小节 4.7
    final ProcessRecord r = new ProcessRecord(stats, info, proc, uid);
    // 判断是否为常驻的进程
    if (!mBooted && !mBooting
            && userId == UserHandle.USER_SYSTEM
            && (info.flags & PERSISTENT_MASK) == PERSISTENT_MASK) {
        r.persistent = true;
    }
    // 将 ProcessRecord 保存在 AMS 里的 mProcessNames 里
    addProcessNameLocked(r);
    return r;
}
```

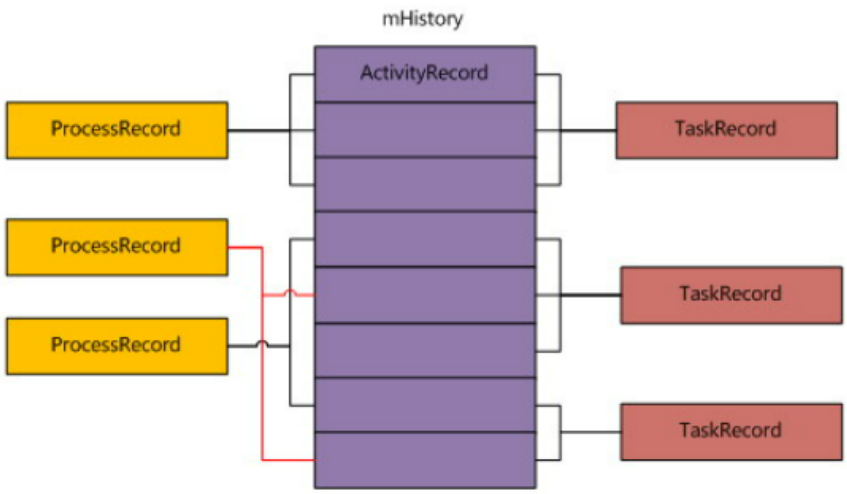
2.4.7 ProcessRecord: 文件：ProcessRecord.java

```
ProcessRecord(BatteryStatsImpl _batteryStats, ApplicationInfo _info,
        String _processName, int _uid) {
    mBatteryStats = _batteryStats;
    info = _info;
    isolated = _info.uid != _uid;
    uid = _uid;
    userId = UserHandle.getUserId(_uid);
    processName = _processName;
    pkgList.put(_info.packageName, new ProcessStats.ProcessStateHolder(_info.versionCode));
    maxAdj = ProcessList.UNKNOWN_ADJ;
    curRawAdj = setRawAdj = ProcessList.INVALID_ADJ;
    curAdj = setAdj = verifiedAdj = ProcessList.INVALID_ADJ;
    persistent = false;
    removed = false;
    lastStateTime = lastPssTime = nextPssTime = SystemClock.uptimeMillis();
}
```

- 这主要是保存一些 ProcessRecord 里面的属性。
- 小结：
  - 第四节的主要作用就是将一些服务注册到 ServiceManger 中，包括 AMS 自己；然后将 framework-res-.apk 中 applicationInfo 信息加入到 SystemServeice 生成的 LoadedApk 中，同时构建 SystemService 对应的 ProcessRecord，最后通过 addProcessNameLocked(r) 来把 SystemService 加入 AMS 的管理中来。

3 这一小部分，感觉和自己所掌握的远程服务绑定基础，讲解得很透彻

- AMS(ActivityManagerService) 是贯穿 Android 系统组件的核心服务，负责了系统中四大组件的启动、切换、调度以及应用进程管理和调度工作。因此想要了解 Android 的内部工作机制，就必须先了解 AMS 的工作原理。在本文中，我将尽可能用通俗的语言去描述 AMS 涉及到的知识点帮助大家理解。
- 先梳理一下这个远程服务它所管理纪录着的几样对它来说狠重要的内容类别：
- AMS 提供了一个 ArrayList mHistory 来管理所有的 activity，activity 在 AMS 中的形式是 ActivityRecord，task 在 AMS 中的形式为 TaskRecord，进程在 AMS 中的管理形式为 ProcessRecord。如下图所示



- 从图中我们可以看出如下几点规则：

- 1. 所有的 ActivityRecord 会被存储在 mHistory 管理；
- 2. 每个 ActivityRecord 会对应到一个 TaskRecord，并且有着相同 TaskRecord 的 ActivityRecord 在 mHistory 中会处在连续的位置；
- 3. 同一个 TaskRecord 的 Activity 可能分别处于不同的进程中，每个 Activity 所处的进程跟 task 没有关系；
- 因此，在分析 Activity 管理之前，先了解一下这个规则。

3.1 AMS 的内部实现

3.1.1 AMS 原理模型

1. 1. ActivityManager

   `/frameworks/base/core/java/android/app/ActivityManager.java`

   - **ActivityManager 是 客户端用来 管理系统中正在运行的所有 Activity 包括 Task、Memory、Service 等信息的工具。**但是 **这些信息的维护工作却不是又 ActivityManager 负责的。在 ActivityManager 中有大量的 get() 方法，那么也就说明了他只是提供信息给 AMS，由 AMS 去完成交互和调度工作。**

2. 2. AMS

   `/frameworks/base/services/java/com/android/server/am/ActivityManagerService.java`

   - AMS 是作为 **管理 Android 系统组件的核心服务**，他在 **SystemServer 执行 run() 方法的时候被创建，并运行在独立的进程中。**具体来说就是 **SystemServer 管理着 Android 中所有的系统服务，这些系统服务的生命周期回调都由 SystemServer 去调度负责。**

```
private void startBootstrapServices() {
    Installer installer = mSystemServiceManager.startService(Installer.class);

    // Activity manager runs the show.
    mActivityManagerService = mSystemServiceManager.startService(
            ActivityManagerService.Lifecycle.class).getService();
    mActivityManagerService.setSystemServiceManager(mSystemServiceManager);
    mActivityManagerService.setInstaller(installer);
}
```

   - 在 **SystemServer 调用 run() 方法中开启必要的系统服务，并将这些服务注册和添加到管理列表中，并执行这些服务在进程中的生命周期。**ActivityManagerService 作为一个重要的核心服务就是在这里被初始成功的。

3.2 AMS 与 ActivityManager 的通信实现

- 我们知道 **AMS 和 ActivityManager 之间通信需要利用 Binder 来完成 (跨进程远程服务绑定)**，那么我们接下来分析一下这个通信机制是如何实现的。
- **ActivityManagerNative(AMN) 中实现的代码是运行在 Android 应用程序的进程空间内，可直接使用的对象，Intent 会由应用程序通过这个类将方法对应的 Binder 命令发送出去。**
- 那么上面这一句话，是否可以理解为，是 AWS 在安卓应用层的客户端呢？在一个安卓系统中，会有几个这样的客户端，还是唯一的呢？
- ActivityManagerNative(AMN) 是一个抽象的类，他包含了如下特点：
  - （1）继承 Binder 类
    - （2）实现 IActivityManager 接口

- 由于 **继承了 Binder 类，他就拥有了远程通信的条件**。
- **实现了 IActivityManager 接口，他能够得到 ActivityManager 管理关于内存、任务等内部信息**。那么 AMS 作为 AMN 的子类也就自然享有了这些特性。
- 我们再回过头来看看 **ActivityManager 中的方法是如何被调用的**，举个栗子：

```
public List<ActivityManager.AppTask> getAppTasks() {
    ArrayList<AppTask> tasks = new ArrayList<AppTask>();
    List<IAppTask> appTasks;
    try {
        appTasks = ActivityManagerNative.getDefault().getAppTasks(mContext.getPackageName());
    } catch (RemoteException e) { // <<<<<<<<<<
        throw e.rethrowFromSystemServer();
    }
// 上面可以得到 appTasks 值，再怎么稍微转化一下成为 tasks，并返回
    return tasks;
}
```

- 我们在代码中发现,类似的 get() 方法的调用逻辑都是 **先通过 ActivityManagerNative.getDefault() 来获得 ActivityManager 的代理接口对象**。getDefault() 到底做了什么？

```
/**
 * Retrieve the system's default/global activity manager.
 */
static public IActivityManager getDefault() {
    return gDefault.get();
}
private static final Singleton<IActivityManager> gDefault = new Singleton<IActivityManager>() {
    protected IActivityManager create() {
        IBinder b = ServiceManager.getService("activity");
        if (false) {
            Log.v("ActivityManager", "default service binder = " + b);
        }
        IActivityManager am = asInterface(b);
        if (false) {
            Log.v("ActivityManager", "default service = " + am);
        }
        return am;
    }
};
```

- **ServiceManager 是系统提供的服务管理类，所有的 Service 都通过他被注册和管理**，并且**通过 getService() 方法能够得到 ActivityManager 与 AMS 的远程通信 Binder 对象**。
- 在模块化设计里，前两天刚学习的 BinderPool 类通过管理所有多个模块化的远程服务的 IBinder 来对这些远程服务绑定统一管理，感觉很类似

```
/**
 * Cast a Binder object into an activity manager interface, generating
 * a proxy if needed.
 */
static public IActivityManager asInterface(IBinder obj) {
    if (obj == null) {
        return null;
    }
    IActivityManager in =
        (IActivityManager)obj.queryLocalInterface(descriptor); // <<<<<<<<<<
    if (in != null) {
        return in;
    }
    return new ActivityManagerProxy(obj); // <<<<<<<<<<
}
```

- **得到了 AMS 的 Binder 对象之后，也就相当于拿到了与 ActivityManager 远程通信的许可证 (句柄，接下来就可以调用其所定义的远程方法了)**。接着，**在 asInterface() 这个方法中，这个许可证的使用权利被移交给了 ActivityManagerProxy，那么 ActivityManagerProxy 就成为了 ActivityManager 与 AMS 远程通信的代理**。
- **ActivityManagerProxy 也实现了 IActivityManager 接口。当客户端 (ActivityManager) 发起向服务端 (AMS) 的远程请求时，客户端提供的数据参数信息被封装打包，然后由**

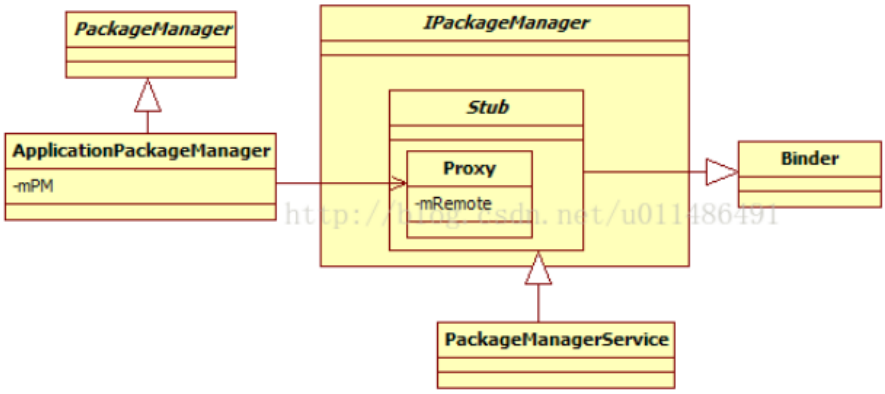
ActivityManager 的远程通信 binder 对象通过 transact() 方法把数据提交，然后再把数据写出返回给 binder 对象。

```
public int startActivity(IApplicationThread caller, String callingPackage, Intent intent,
                         String resolvedType, IBinder resultTo, String resultWho, int requestCode,
                         int startFlags, ProfilerInfo profilerInfo, Bundle options) throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeStrongBinder(caller != null ? caller.asBinder() : null);
    data.writeString(callingPackage);
    intent.writeToParcel(data, 0);
    data.writeString(resolvedType);
    data.writeStrongBinder(resultTo);
    data.writeString(resultWho);
    data.writeInt(requestCode);
    data.writeInt(startFlags);
    if (profilerInfo != null) {
        data.writeInt(1);
        profilerInfo.writeToParcel(data, Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
    } else {
        data.writeInt(0);
    }
    if (options != null) {
        data.writeInt(1);
        options.writeToParcel(data, 0);
    } else {
        data.writeInt(0);
    }
    mRemote.transact(START_ACTIVITY_TRANSACTION, data, reply, 0);
    reply.readException();
    int result = reply.readInt();
    reply.recycle();
    data.recycle();
    return result;
}
```

- 通过这种方式，AMS 在自己的进程中就能获得 ActivityManager 进程发来的数据信息，从而完成对于 Android 系统组件生命周期的调度工作。
- https://blog.csdn.net/yueliangniao1/article/details/7227165?spm=1001.2101.3001.6650.9&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-9-7227165-blog-8891414.topnsimilarv1&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-9-7227165-blog-topnsimilarv1&utm_relevant_index=10 上面这个讲到了活动启动模式以及任务栈和活动在不同栈中的迁移，感觉讲到了很多细节，明天早上再仔细看一遍
- Windows 上的截图是一个被我放了很久的 bug，没能安排好时间占用了早上最宝贵的时间来修改，有点儿可惜，但过程中在需要的时候一定能够解决自己的应用便利需要，并解决过程中所遇到的一切问题，包括用键的一再精简与优化，snipaste 截图截出的图过浅等问题，还是很开心的。因为使用 powershell 来拿到 Windows 系统的剪贴板，可能具备 wsl ubuntu emacs 中使用的能力，但是 Ubuntu 中暂时保留一个简炼启动快的版本，暂时不再配置
- 晚上会动动笔，写或总结一些基础算法题；希望早上或白天的时间都能够用来深入学习安卓系统

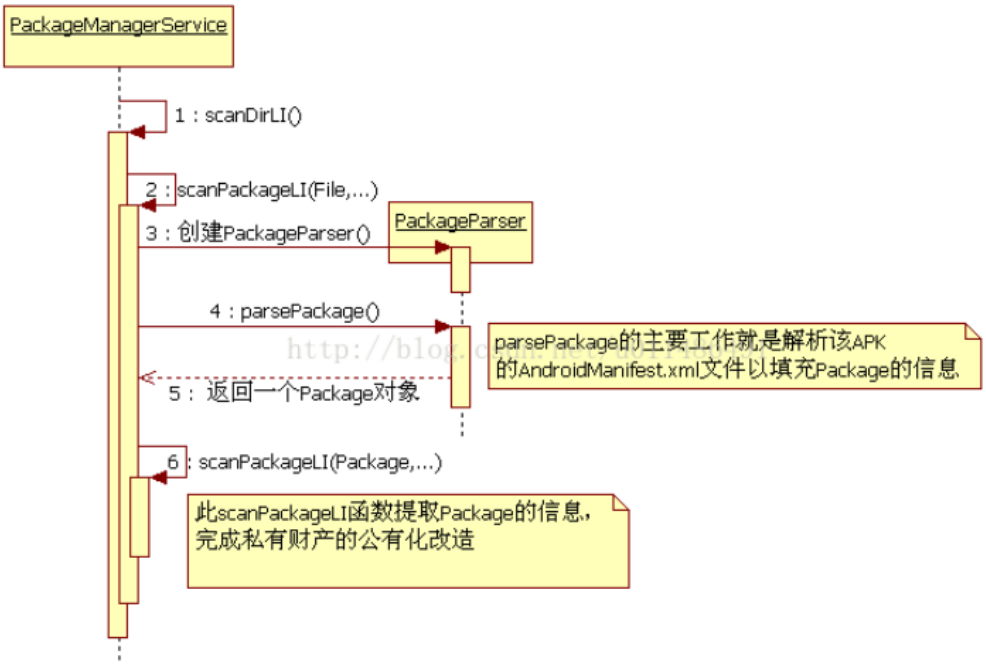
4 PackageManagerService

- 它是 Android 系统中最常用的服务之一。它负责系统中 Package 的管理，应用程序的安装、卸载、信息查询等。
- 它的一家老小：



- 分析上图：
  - 1.IPackageManager 接口类中定义了服务端和客户端通信的业务函数，还定义了内部类 Stub，该类从 Binder 派生并实现了 IPackageManager 接口。
  - 2.PackageManagerService 继承自 IPackageManager.Stub 类，由于 Stub 类从 Binder 派生，因此 **PackageManagerService 将作为服务端参与 Binder 通信**。
  - 3.Stub 类中定义了一个内部类 Proxy，该类有一个 IBinder 类型（实际类型为 BinderProxy）的成员变量 mRemote，**mRemote 用于和服务端 PackageManagerService 通信**。
  - 4.IPackageManager 接口类中定义了许多业务函数，但是出于安全等方面的考虑，Android 对外（即 SDK）提供的只是一个子集，该子集被封装在抽象类 PackageManager 中。客户端一般通过 Context 的 getPackageManager 函数返回一个类型为 PackageManager 的对象，该对象的实际类型是 PackageManager 的子类 ApplicationPackageManager。这种基于接口编程的方式，虽然极大降低了模块之间的耦合性，却给代码分析带来了不小的麻烦。
  - 5.ApplicationPackageManager 类继承自 PackageManager 类。它并没有直接参与 Binder 通信，而是通过 mPM 成员变量指向一个 IPackageManager.Stub.Proxy 类型的对象。
- PKMS 构造函数的主要功能是，扫描 Android 系统中几个目标文件夹中的 APK，从而建立合适的数据结构以管理诸如 Package 信息、四大组件信息、权限信息等各种信息。抽象地看，PKMS 像一个加工厂，它 **解析实际的物理文件（APK 文件）以生成符合自己要求的产品**。

4.1 PKMS 工作流程



- PKMS 构造函数的工作流程大体可分三个阶段：
  - · 扫描目标文件夹之前的准备工作。
  - · 扫描目标文件夹。
  - · 扫描之后的工作。
- 该函数涉及到的知识点较多，代码段也较长，因此我们将通过分段讨论的方法，集中解决相关的重点问题。

4.2 扫描目标文件夹之前的准备工作

4.2.1 设置 setting

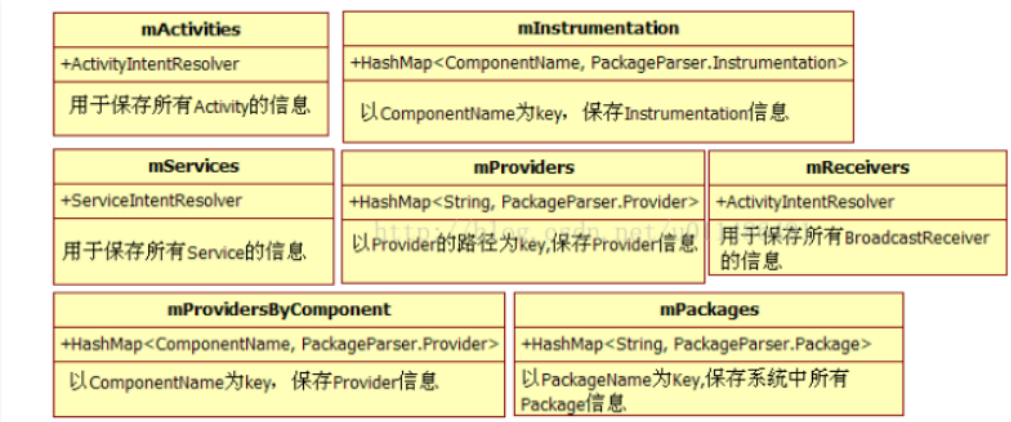
- 进入 PKMS 构造函数，就会遇到第一个较为复杂的数据结构 Setting 及它的 addSharedUserLPw 函数。Setting 的作用是管理 Android 系统运行过程中的一些设置信息，包括：字符串“android.uid.system “，UID（用户 ID，），用于标识系统 Package 的 FLAG_SYSTEM 标志。

4.2.2 扫描文件夹并解析 xml 文件

- 创建一个 Installer 对象，同时并解析对应的 xml 文件。包括：权限，包信息等。
- PKMS 将扫描以下几个目录。
  - 1. **/system/frameworks** ： 该目录中的文件都是系统库，例如 framework.jar、services.jar、framework-res.apk。不过 scanDirLI 只扫描 APK 文件，所以 framework-res.apk 是该目录中唯一“受宠”的文件。
  - 2. **/system/app** ：该目录下全是默认的系统应用，例如 Browser.apk、SettingsProvider.apk 等。

- 3. **/vendor/app**：该目录中的文件由厂商提供，即厂商特定的 APK 文件，不过目前市面上的厂商都把自己的应用放在 **/system/app** 目录下。

- 下面总结一下 Package 扫描的流程:
  - scanDirLI 用于对指定目录下的 APK 文件进行扫描。
  - 扫描完 APK 文件后，Package 的私有财产就充公了。PKMS 提供了好几个重要数据结构来保存这些财产:



4.2.3 构造函数分析之扫尾工作

- 这部分任务比较简单，就是将第二阶段收集的信息再集中整理一次，比如汇总并更新和 Permission 相关的信息，将信息写到 package.xml、package.list 及 package-stopped.xml 文件中。

4.3 PKM 的 install

- 故事从 adbinstall 开始。

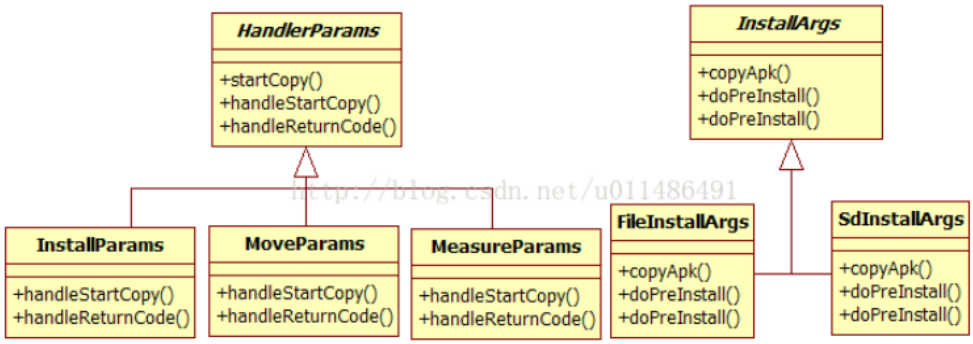
4.3.1 adb install 分析

- 找到 apk 的目录位置，把相关信息发送给指定的 Verification 程序（另外一个 APK），由它对要安装的 APK 进行检查（Verify）。
- 然后开始调用 pm_command 开始后续工作。

4.3.2 调用 pm_command 进行安装

- 在编译 system.image 时，Android.mk 中会将该脚本复制到 system/bin 目录下。从 pm 脚本的内容来看，它就是通过 app_process 执行 pm.jar 包的 main 函数。app_process 是一个 Native 进程，它通过创建虚拟机启动了 Zygote，从而转变为一个 Java 进程。实际上，app_process 还可以通过类似的方法（即先创建 Dalvik 虚拟机，然后执行某个类的 main 函数）来转变成其他 Java 程序。
- Pm 解析参数后，最终通过 PKMS 的 Binder 客户端调用 **installPackageWithVerification** 以完成后续的安装工作。
- installPackageWithVerification 函数倒是蛮清闲，检查下权限，然后简简单单创建几个对象，发送 INIT_COPY 消息给 mHandler，就甩手退出了。

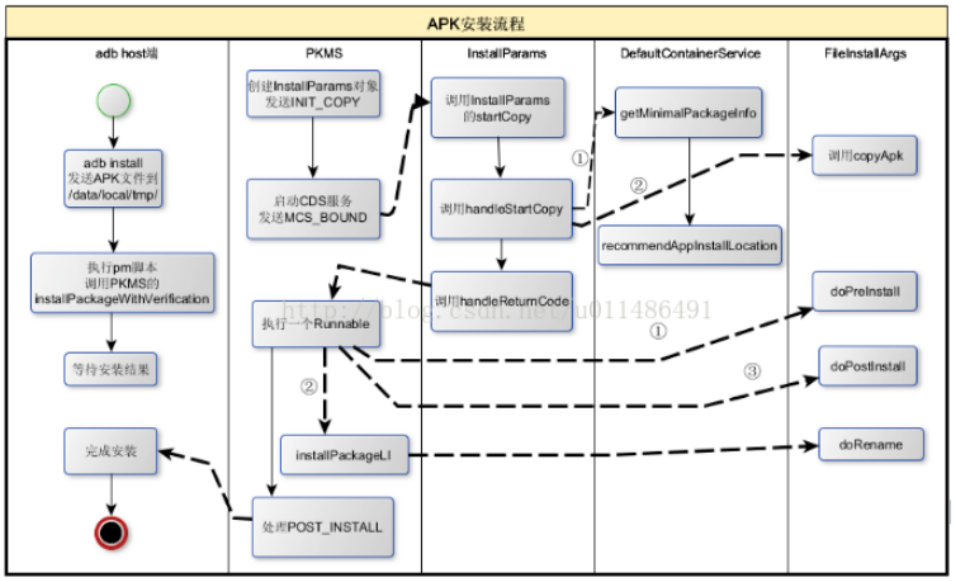
- 在 mHandler 中 APK 的安装居然需要使用另外一个 APK 提供的服务，该服务就是 DefaultContainerService，由 DefaultCotainerService.apk 提供。
- 相关类的一家子：



- HandlerParams 和 InstallArgs 均为抽象类。
- **HandlerParams 有三个子类，分别是 InstallParams、MoveParams 和 MeasureParams** 。其中，InstallParams 用于处理 APK 的安装，MoveParams 用于处理某个已安装 APK 的搬家请求（例如从内部存储移动到 SD 卡上），MeasureParams 用于查询某个已安装的 APK 占据存储空间的大小（例如在设置程序中得到的某个 APK 使用的缓存文件的大小）。
- 对于 **InstallParams** 来说，它 **还有两个伴儿，即 InstallArgs 的派生类 FileInstallArgs 和 SdInstallArgs** 。其中，FileInstallArgs 针对的是安装在内部存储的 APK，而 SdInstallArgs 针对的是那些安装在 SD 卡上的 APK。
- 在 mhandler 中处理请求：
  - 1. 调用 DCS 的 getMinimalPackageInfo 函数，将得到一个 PackageLite 对象，该对象是一个轻量级的用于描述 APK 的结构（相比 PackageParser.Package 来说）。在这段代码逻辑中，主要想取得其 recommendedInstallLocation 的值。此值表示该 APK 推荐的安装路径。
    - 具体步骤：通过用户在 Settings 数据库中设置的安装位置。检查外部存储或内部存储是否有足够空间。
  - 2. 调用 installLocationPolicy 检查推荐的安装路径。例如系统 Package 不允许安装在 SD 卡上。
  - 3.createInstallArgs 将根据安装位置创建不同的 InstallArgs。如果是内部存储，则返回 FileInstallArgs，否则为 SdInstallArgs。
  - 4. 在正式安装前，应先对该 APK 进行必要的检查。
  - 5. 调用 InstallArgs 的 copyApk。
- 调用 PKMS 的 installPackageLI 函数进行 APK 安装，该函数内部将调用 InstallArgs 的 doRename 对临时文件进行改名。另外，还需要扫描此 APK 文件。
- 该 APK 已经安装完成（不论失败还是成功），继续向 mHandler 抛送一个 **POST_INSTALL** 消息，该消息携带一个 token，通过它可从 mRunningInstalls 数组中取得一个 PostInstallData 对象
- 在接收到 POST_INSTALL 消息之后，发送 **PACKAGE_ADDED** 广播。

4.3.3 APK 安装流程总结

- APK 的安装流程竟然如此复杂，其目的无非是 **让 APK 中的私人财产公有化**。

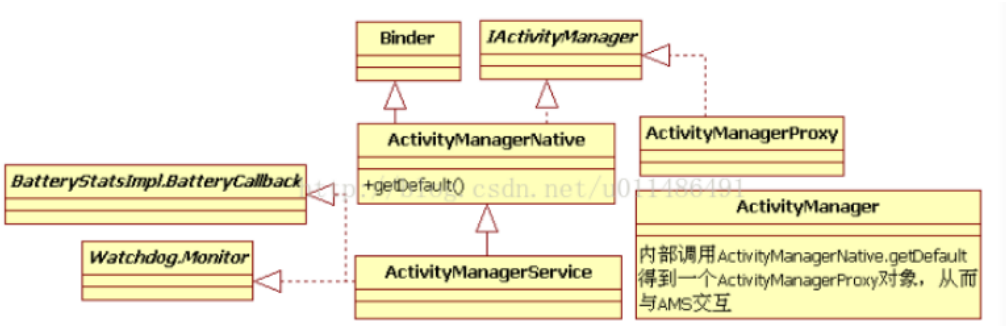


- 1. 安装 APK 到内部存储空间这一工作流程涉及的主要对象包括：PKMS、DCS、InstallParams 和 FileInstallArgs。
- 2. 此工作流程中每个对象涉及到的关键函数。
- 3. 对象之间的调用通过虚线表达，调用顺序通过 等标明。

5 ActivityManagerService: 今天早上再读，才觉得这篇讲得深入又浅显易懂，赶快收藏

- AMS 是 Android 中最核心的服务，主要负责系统中四大组件的启动、切换、调度及应用进程的管理和调度等工作，其职责与操作系统中的进程管理和调度模块相类似，因此它在 Android 中非常重要。
- AMS 比较复杂，本章将带领读者按五条不同的线来分析它：
  - 1. 第一条线：同其他服务一样，将分析 SystemServer 中 AMS 的调用轨迹。
  - 2. 第二条线：以 am 命令启动一个 Activity 为例，分析应用进程的创建、Activity 的启动，以及它们和 AMS 之间的交互等知识。
  - 3. 第三条线和第四条线：分别以 Broadcast 和 Service 为例，分析 AMS 中 Broadcast 和 Service 的相关处理流程。
  - 4. 第五条线：以一个 Crash 的应用进程为出发点，分析 AMS 如何打理该应用进程的身后事。

5.1 AMS 的家族图谱：



- AMS 由 ActivityManagerNative（以后简称 AMN）类派生，并实现 Watchdog.Monitor 和 BatteryStatsImpl.BatteryCallback 接口。而 **AMN 由 Binder 派生，实现了 IActivityManager 接口**。
- **客户端使用 ActivityManager 类**。由于 AMS 是系统核心服务，很多 API 不能开放供客户端使用，所以设计者没有让 ActivityManager 直接加入 AMS 家族。**在 ActivityManager 类内部通过调用 AMN 的 getDefault 函数得到一个 ActivityManagerProxy 对象，通过它可与 AMS 通信**。

5.2 AMS 的调用轨迹

- AMS 由 SystemServer 的 ServerThread 线程创建，它的调用轨：
  - 1. 调用 main 函数，得到一个 Context 对象
  - 2. 将 SystemServer 进程可加到 AMS 中，并被它管理
  - 3. 将 ActivityManagerService 的 SettingsProvider 放到 SystemServer 进程中来运行。
  - 4. 在内部保存 WindowManagerService（以后简称 WMS）

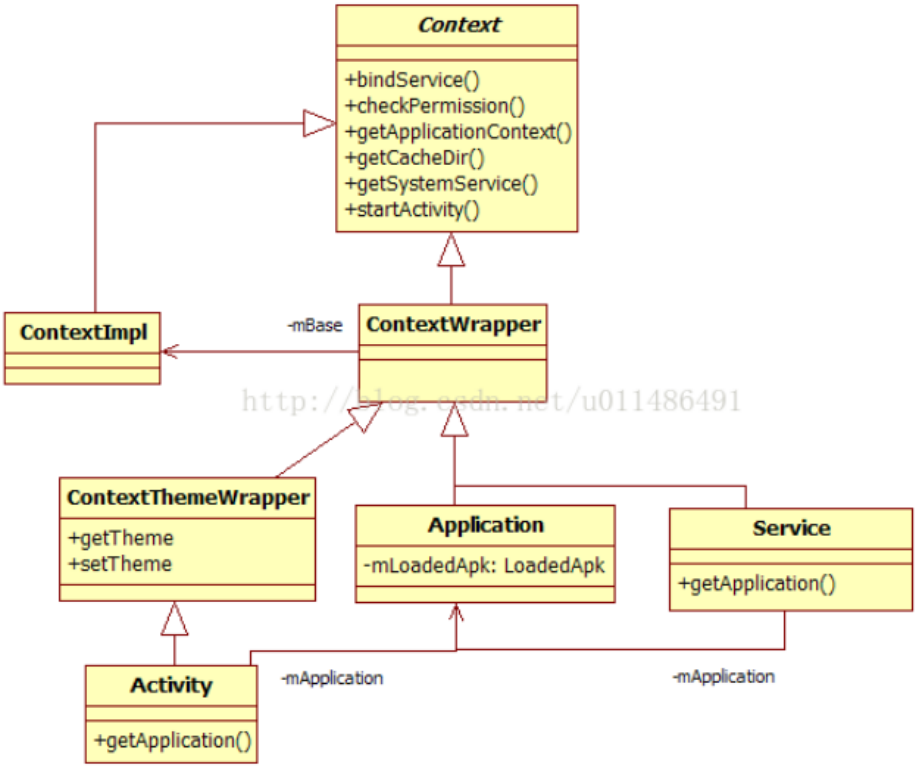
5.2.1 1.Main 中的调用

- 1. 创建 AThread 线程。
- 2.ActivityThread.systemMain 函数。初始化 ActivityThread 对象。
- 3.ActivityThread.getSystemContext 函数。用于获取一个 Context 对象，从函数名上看，该 Context 代表了 System 的上下文环境。
- 4.AMS 的 startRunning 函数。

1. 步骤 1：创建 AThread 线程
   - 虽然 AMS 的 main 函数由 ServerThread 线程调用，但是 AMS 自己的工作并没有放在 ServerThread 中去做，而是新创建了一个线程，即 AThread 线程。其主要工作就是创建 AMS 对象，然后通知 AMS 的 main 函数。
   - AMS 构造函数的工作：
     - 创建 BSS、USS、mProcessStats（ProcessStats 类型）、mProcessStatsThread 线程，这些都与系统运行状况统计相关。
     - 创建/data/system 目录，为 mCompatModePackages（CompatModePackages 类型）和 mConfiguration（Configuration 类型）等成员变量赋值。
2. 步骤 2：ActivityThread.systemMain 函数。初始化 ActivityThread 对象

- 它 **ActivityThread 代表一个应用进程的主线程（对于应用进程来说，ActivityThread 的 main 函数确实是由该进程的主线程执行）**，其职责就是调度及执行在该线程中运行的四大组件。
- **应用进程**指那些 **运行 APK 的进程，它们由 Zyote 派生（fork）而来，上面运行了 dalvik 虚拟机**。与应用进程相对的就是 **系统进程（包括 Zygote 和 SystemServer）**。
- 前面所说的 **ActivityThread 代表应用进程**（其上运行了 APK）的主线程，而 **SystemServer 并非一个应用进程**，那么为什么此处也需要 ActivityThread 呢?
  - 还记得在 **PackageManagerService 分析中提到的 framework-res.apk** 吗? **这个 APK 除了包含资源文件外，还包含一些 Activity（如关机对话框），这些 Activity 实际上运行在 SystemServer 进程中。从这个角度看，SystemServer 是一个特殊的应用进程**。
  - **通过 ActivityThread 可以把 Android 系统提供的组件之间的交互机制和交互接口（如利用 Context 提供的 API）也拓展到 SystemServer 中使用**。
- 调用 systemMain 函数结束后:
  - 1. 得到一个 ActivityThread 对象，它代表应用进程的主线程。
  - 2. 得到一个 Context 对象，它背后所指向的 Application 环境与 framework-res.apk 有关。
- **systemMain 函数将为 SystemServer 进程搭建一个和应用进程一样的 Android 运行环境**。

3. 步骤 3：ActivityThread.getSystemContext 函数。用于获取一个 Context 对象，从函数名上看，该 Context 代表了 System 的上下文环境。
   - 调用该函数后，将得到一个代表系统进程的 Context 对象。



4. 步骤 4：AMS 的 startRunning 函数。

- 就是调用 startRunning

5. 总结：

   - AMS 的 main 函数的目的有两个：
     - 1. 首先也是最容易想到的目的是创建 AMS 对象。
     - 2. 另外一个目的比较隐晦，但是非常重要，那就是创建一个供 SystemServer 进程使用的 Android 运行环境。

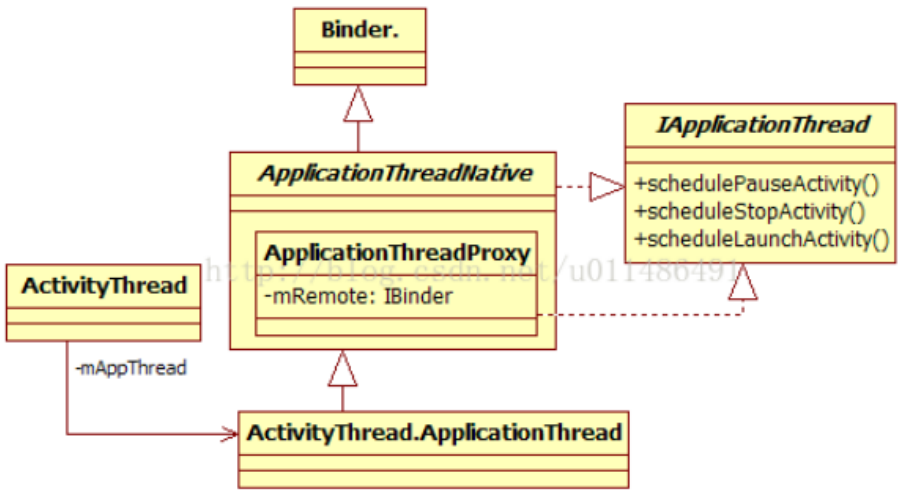
5.2.2 2.AMS 的 setSystemProcess(将 SystemServer 进程可加到 AMS 中，并被它管理)

1. 调用 ActivityThread 的 installSystemApplicationInfo 函数

   - installSystemApplicationInfo 函数的参数为一个 ApplicationInfo 对象，该对象由 AMS 通过 Context 查询 PKMS 中一个名为“android”的 package 得来，目的仅仅是为了创建一个 Android 运行环境。
   - 由于 framework-res.apk 是一个 APK 文件，和其他 APK 文件一样，它应该运行在一个进程中。而 **AMS 是专门用于进程管理和调度的**，所以运行 APK 的进程应该在 AMS 中有对应的管理结构。因此 **AMS 下一步工作就是将这个运行环境和一个进程管理结构对应起来并交由 AMS 统一管理**。

2. AMS 对进程的管理

   - AMS 中的 **进程管理结构是 ProcessRecord**。

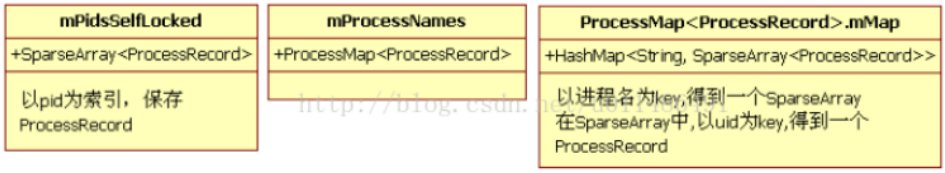


   - 由上图可知：
     - 1. **ApplicationThreadNative 实现了 IApplicationThread 接口**。从该接口定义的函数可知，**AMS 通过它可以和应用进程进行交互**，例如，AMS 启动一个 Activity 的时候会调用该接口的 scheduleLaunchActivity 函数。
     - 2. **ActivityThread 通过成员变量 mAppThread 指向它的内部类 ApplicationThread**，而 ApplicationThread 从 ApplicationThreadNative 派生。
   - **当 AMS 想要停止（stop）一个 Activity 时，会调用对应进程 IApplicationThread Binder 客户端的 scheduleStopActivity 函数**。该函数 **服务端**实现的就是 **向（客户端应用的)ActivityThread 所在线程发送一个消息**。在 **应用进程（客户端中）中，ActivityThread 运行在主线程中，所以这个消息最终在主线程被处理**。

- **ProcessRecord 除保存的信息包括：**和**与应用进程通信的 IApplicationThread 对象**外，还保存了**进程名**、不同状态对应的 **Oom_adj** 值及一个 **ApplicationInfo** 。一个进程虽然可运行多个 Application，但是 **ProcessRecord 一般保存该进程中先运行的那个 Application 的 ApplicationInfo**。

3. 保存该 ProcessRecord 对象

   - AMS 中有两个成员变量用于保存 ProcessRecord，一个是 **mProcessNames** ，另一个是 **mPidsSelfLocked** ，如下图所示为这两个成员变量的数据结构示意图。



   - 仔细看上面的图，是不是至少有一处地方 (uid vs pid) 写错了？我觉得两个要么都是 uid 要么都是 pid 才对?

4. AMS 的 setSystemProcess 总结：

   - 现在来总结回顾 setSystemProcess 的工作：
     - 1. 注册 AMS、meminfo、gfxinfo 等服务到 ServiceManager 中。
     - 2. **根据 PKMS(PackageManagerService) 返回的 ApplicationInfo 初始化 Android 运行环境**，并 **创建一个代表 SystemServer 进程的 ProcessRecord，从此，SystemServer 进程也并入 AMS 的管理范围内**。

5.2.3 3.AMS 的 installSystemProviders(3. 将 ActivityManagerService 的 SettingsProvider 放到 SystemServer 进程中来运行)

- SystemServer 中很多 Service 都需要向 settings 数据库查询配置信息。为此，Android 提供了一个 SettingsProvider 来帮助开发者。该 Provider 在 SettingsProvider.apk 中，installSystemProviders 就会加载该 APK 并把 SettingsProvider 放到 SystemServer 进程中来运行。

- 这里 AMS 向 PKMS 查询满足要求的 ProviderInfo，最重要的查询条件包括：进程名和进程 uid。同时 AMS 对 ContentProvider 进行管理（这也是查询 ProviderInfo 的原因）。

1. ASM 的 systemReady

   - 可以分为三个阶段的工作:

(a) 第一阶段：

   - 发送并处理与 **PRE_BOOT_COMPLETED** 广播相关的事情。**该广播接收者的工作似乎和系统升级有关。(目前的工作案例中我还不曾涉及到这一块儿)**

(b) 第二阶段：

   - 1. 杀死那些竟然在 AMS 还未启动完毕就先启动的应用进程。注意，这些应用进程一定是 APK 所在的 Java 进程，因为只有应用进程才会向 AMS 注册，而一般 Native（例如 mediaserver）进程是不会向 AMS 注册的。
   - 2. **从 Settings 数据库中获取配置信息,目前只取 4 个配置参数**,分别是:"**debug_app**"（设置需要 *debug* 的 *app* 的名称）、"**wait_for_debugger**"（如果为 *1*，则等待调试器，否则正常启动 *debug_app* ）、"**always_finish_activities**"（当一个 *activity* 不再有地方使用时，是否立即对它执行 *destroy* ）、"**font_scale**"（用于控制字体放大倍数，这是 *Android 4.0* 新增的功能）。

- 以上配置项由 Settings 数据库的 System 表提供。
- 附加：自己先前读 **安卓系统 Settings 源码的时候，它提供了四个缺省表，System，Default, Global, + ?**

(c) 第三阶段：

- 1. 调用 systemReady 设置的回调对象 goingCallback 的 run 函数。
  - 在 run 中，调用一些服务的 systemReady 函数和启动 Watchdog。
- 2. 启动那些声明了 persistent 的 APK。persistent apk 就是，在系统启动的时候就可以运行的 apk.

```
<application
    android:persistent="true|false">
</application>
```

- 在我们开发系统级的 App 时，很有可能就会用 persistent 属性。当在 AndroidManifest.xml 中将 persistent 属性设置为 true 时，那么该 App 就会具有如下两个特性：
  - 系统刚起来的时候，该 App 也会被启动起来
  - App 被强制杀掉后，系统会重启该 App。这种情况只针对系统内置的 App，第三方安装的 App 不会被重启。
- 3. 启动桌面，在 Home 启动成功后，AMS 才发送 **ACTION_BOOT_COMPLETED** 广播 (这个广播就是自己工作应用中经常会用到/会接听用于开机启动完成后应用必要的优先配置等使用的广播了)

5.2.4 ActivityManagerService 总结

- 1. **AMS 的 main 函数**：**创建 AMS 实例**，其中最重要的工作是 **创建 Android 运行环境，得到一个 ActivityThread 和一个 Context 对象**。
- 2. **AMS 的 setSystemProcess() 函数**：该函数 **注册 AMS 和 meminfo 等服务到 ServiceManager 中**。另外，**它为 SystemServer 创建了一个 ProcessRecord 对象**。由于 **AMS 是 Java 世界的进程管理及调度中心，要做到对 Java 进程一视同仁，尽管 SystemServer 贵为系统进程，此时也不得不将其并入 AMS 的管理范围内**。
- 3. **AMS 的 installSystemProviders**：为 **SystemServer 加载 SettingsProvider**。
- 4.AMS 的 **systemReady** ： 做 **系统启动完毕前最后一些扫尾工作。该函数调用完毕后，HomeActivity 将呈现在用户面前**。

6 am 命令启动一个 Activity

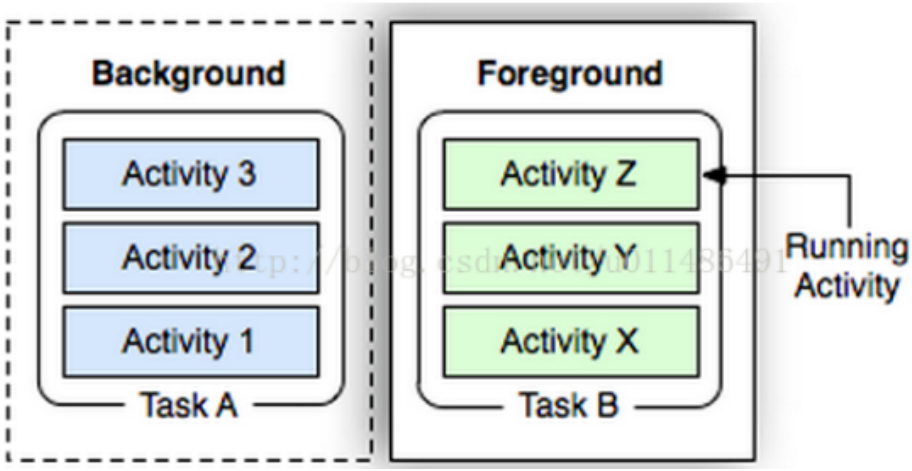
- am 和 pm 一样，也是一个脚本，它用来和 AMS 交互，如启动 Activity、启动 Service、发送广播等。其核心文件在 Am.java 中。

6.1 AMS 的 startActivityAndWait 函数分析

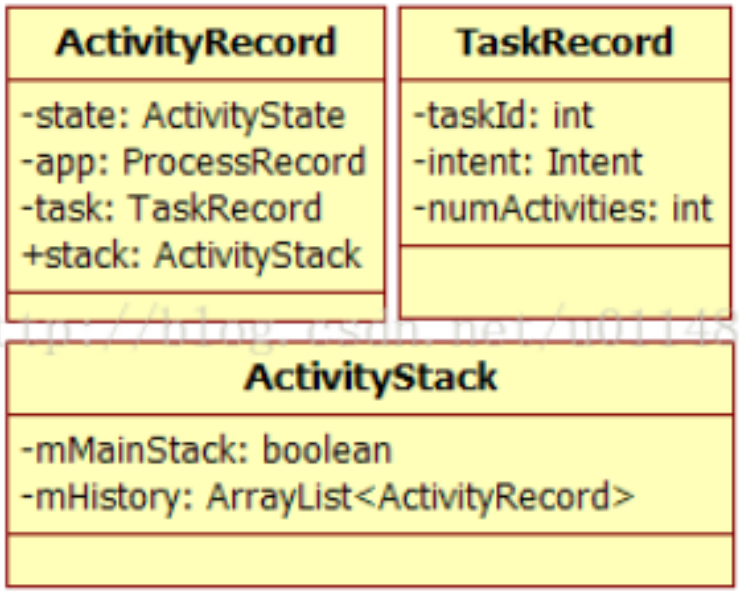
6.2 Task、Back Stack、ActivityStack 及 Launch mode

- 一个 Acitivity 的启动是由一个应用进程发起的，IApplicationThread 是应用进程和 AMS 交互的通道，也可算是调用进程的标示。
- mMainStack 为 AMS 的成员变量，类型为 ActivityStack。他通过 startActivityAndWait 开始启动 activity。
- 对多 Task 的情况来说，系统只支持一个处于前台的 Task，即用户当前看到的 Activity 所属的 Task，其余的 Task 均处于后台，这些后台 Task 内部的 Activity 保持顺序不变。用户可以一次将整个 Task 挪到后台或者置为前台。

- Task 内部 Activity 的组织方式如下：



- Android 通过先入后出的方式来组织 Activity。数据结构中的 Stack 即以这种方式工作。
- 为了应对多个 Task 的组织及管理方式，Android 设计了一个 ActivityStack 类来负责上述工作。



- **Activity 由 *ActivityRecord* 表示，Task 由 *TaskRecord* 表示。**ActivityRecord 的 task 成员指向该 Activity 所在的 Task。state 变量用于表示该 Activity 所处的状态（包括 INITIALIZING、RESUMED、PAUSED 等状态）。
- ActivityStack 用 mHistory 这个 ArrayList 保存 ActivityRecord。**令人大跌眼镜的是，该 mHistory 保存了系统中所有 Task 的 ActivityRecord，而不是针对某个 Task 进行保存。**
  - 每每读到这种时候，小伙伴们就惊呆了：镜片掉了一地，隐形掉了一地，眼珠也掉了一地。。。。。
  - **ActivityStack 的 mMainStack** 成员比较有意思，**它代表此 ActivityStack 是否为主 ActivityStack** 。有主必然有从，但是目前系统中只有一个 ActivityStack，并且它的 mMainStack 为 true。

  - 从 ActivityStack 的命名可推测，Android 在开发之初也想用 ActivityStack 来管理单个 Task 中的 ActivityRecord（在 ActivityStack.java 的注释中说过，该类为 “**State and management of a single stack of activities** ”），但不知何故，在现在的代码实现将所有 Task 的 ActivityRecord 都放到 mHistory 中了，并且依然保留 mMainStack。

- **ActivityStack 采用数组的方式保存所有 Task 的 ActivityRecord(这里真的是数组吗？还是写错了？)**，并且 **没有成员保存 TaskRecord** 。
  - **这种实现方式少了 TaskRecord 一级的管理，直接以 ActivityRecord 为管理单元**。这种做法 **能降低管理方面的开销。但是缺点是弱化了 Task 的概念，结构不够清晰**。
- **启动模式**：分别是 **standard**、**singleTop**、**singleTask 和 singleInstance** 。描述的是 **activity 之间的关系**。
- **标志**:**FLAG_ACTIVITY_NEW_TASK,FLAG_ACTIVITY_CLEAR_TASK,FLAG_ACTIVITY_CLEA**
  。描述的是 **Activity 和 Task 关系的**。

6.3 startActivityAndWait

- 该函数的目标是启动 com.dfp.test.TestActivity，假设系统之前没有启动过该 Activity，则
  - 1. 由于在 am 中设置了 FLAG_ACTIVITY_NEW_TASK 标志，因此除了会创建一个新的 ActivityRecord 外，还会新创建一个 TaskRecord。
  - 2. 还需要启动一个新的应用进程以加载并运行 com.dfp.test.TestActivity 的一个实例。
  - 3. 如果 TestActivity 不是 Home，还需要停止当前正在显示的 Activity。

6.3.1 具体步骤:

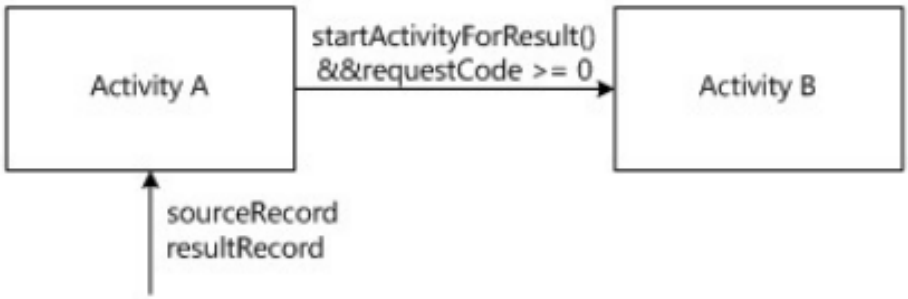
- 1. 首先需要通过 PKMS 查找匹配该 Intent 的 ActivityInfo。
- 2. 处理 FLAG_CANT_SAVE_STATE 的情况，但系统目前不支持此情况。
- 3. 另外，获取调用者的 pid 和 uid。由于本例的 caller 为 null，故所得到的 pid 和 uid 均为 am 所在进程的 uid 和 pid。
- 4. 启动 Activity 的核心函数是 startActivityLocked。
- 5. 根据返回值做一些处理，因为目标 Activity 要运行在一个新的应用进程中，就必须等待那个应用进程正常启动并处理相关请求

6.4 startActivityLocked 主要工作

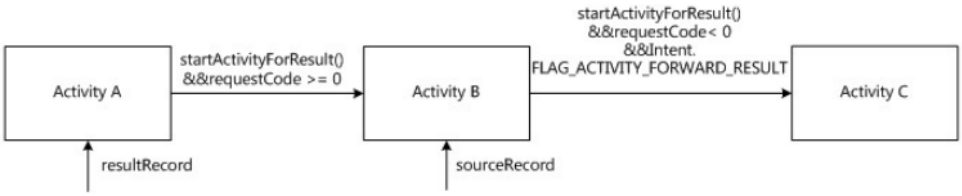
- 1. 处理 sourceRecord 及 resultRecord。其中，sourceRecord 表示发起本次请求的 Activity，resultRecord 表示接收处理结果的 Activity(启动一个 Activity 肯定需要它完成某项事情,当目标 Activity 将事情成后,就需要告知请求者该事情的处理结果)。**在一般情况下,sourceRecord 和 resultRecord 应指向同一个 Activity** (这里这么写是故意误导弱弱们的吧？)。
- 2. 处理 app Switch。如果 AMS 当前禁止 appswitch，则只能把本次启动请求保存起来，以待允许 app switch 时再处理。从代码中可知，AMS 在处理本次请求前，会先调用 doPendingActivityLaunchesLocked 函数，在该函数内部将启动之前因系统禁止 appswitch 而保存的 Pending 请求。
- 3. 调用 startActivityUncheckedLocked 处理本次 Activity 启动请求，为新创建的 ActivityRecord 找到一个合适的 Task。
- **下面是来自另一处更变详尽的解释**

- startActivityLocked() 方法在创建 ActivityRecord 前，还做了如下几个操作：

  1. 确定 sourceRecord 和 resultRecord，这两个变量均为 ActivityRecord 类型，前者代表请求启动当前 activity 的 activity；后者表示当前的 activity 在启动之后需要返回结果的 ActivityRecord，一般情况下，如果 sourceRecord 的 activity 使用 startActivityForResult() 启动当前 activity 并且 requestCode>=0 时，则 resultRecord 不为空，且 resultRecord=sourceRecord。



- 还有一种特殊的情况，当启动一个 activity 时，启动的 Intent 设置了 Intent.FLAG_ACTIVITY_FORWAR 标志，在这种情况 resultRecord 并不指向 sourceRecord，而是指向 sourceRecord 的 sourceRecord，比较绕上个图先



- 如上图所示，Activity A 启动了 Activity B，Activity B 又启动了 C，A->B->C，这种情况下，A 启动 B 要求 B 返回 result 给 A，但是如果 B 在启动 C 时，Intent 设置了 Intent.FLAG_ACTIVITY_FORWARD_RESULT 标志，那么此时将会交由 C 向 A setResult。为了避免冲突，B 启动 C 时不得指定 resultRecord>=0。

```
ActivityRecord sourceRecord = null;
ActivityRecord resultRecord = null;
if (resultTo != null) {
    int index = indexOfTokenLocked(resultTo);
    if (DEBUG_RESULTS) Slog.v(
        TAG, "Sending result to " + resultTo + " (index " + index + ")");
    if (index >= 0) {
        sourceRecord = (ActivityRecord)mHistory.get(index);
        if (requestCode >= 0 && !sourceRecord.finishing)
            resultRecord = sourceRecord;
    }
}
int launchFlags = intent.getFlags();
if ((launchFlags&Intent.FLAG_ACTIVITY_FORWARD_RESULT) != 0
        && sourceRecord != null) {
    // Transfer the result target from the source activity to the new
    // one being started, including any failures.
    if (requestCode >= 0) {
        return START_FORWARD_AND_REQUEST_CONFLICT;
    resultRecord = sourceRecord.resultTo;
    resultWho = sourceRecord.resultWho;
    requestCode = sourceRecord.requestCode;
    sourceRecord.resultTo = null;
    if (resultRecord != null)
        resultRecord.removeResultsLocked(sourceRecord, resultWho, requestCode);
}
```

6.5 startActivityUncheckedLocked 的分析：

- 步骤 1：确定是否需要为新的 Activity 创建一个 Task，即是否设置 FLAG_ACTIVITY_NEW_TASK 标志。
- 步骤 2：找到一个合适的 Task，然后对应做一些处理。
- 步骤 3：根据条件判断使用原有的 TaskRecord 还是新建一个，并调用 startActivityLocked 函数进行处理。
- 步骤 4：进行 Activity 之间的动画切换。
- 总结：首先创建 ActivityRecord 和 TaskRecord 并将 ActivityRecord 添加到 mHistory 末尾，然后调用 resumeTopActivityLocked 启动它。

6.6 resumeTopActivityLocked 分析

- 在当中 mResumedActivity 指向上一次启动的 Activity，也就是当前界面显示的这个 Activity，如果 mHistory 中没有要启动的 Activity，则启动 Home。
- （接 startActivityUncheckedLocked 的分析）如果该 ActivityRecord 已有对应的进程存在，则只需要重启 Activity。此进程还不存在，所以要先创建一个应用进程，通过 startProcessLocked。

6.7 startProcessLocked 分析

- 通过发送消息给 Zygote 以派生一个应用进程（应用进程）。

6.7.1 应用进程的创建及初始化

- 应用进程的入口是 ActivityThread 的 main 函数，它是在主线程中执行的。
  - 1、在 main 函数内部将创建一个消息循环 Loop，接着调用 ActivityThread 的 attach 函数，最终将主线程加入消息循环。
  - 2、AMS 创建一个应用进程后，会设置一个超时时间（一般是 10 秒）。如果超过这个时间，应用进程还没有和 AMS 交互，则断定该进程创建失败。所以，应用进程启动后，需要尽快和 AMS 交互，即调用 AMS 的 attachApplication 函数。在该函数内部将调用 attachApplicationLocked。

6.7.2 attachApplicationLocked 分析

1. 第一步：
   - **设置代表该应用进程的 ProcessRecrod 对象的一些成员变量**，例如 **用于和应用进程交互的 thread 对象、进程调度优先级**及 **oom_adj 的值**等。
   - 从消息队列中撤销 **PROC_START_TIMEOUT_MSG** (这种超时机制在车载系统的双向信号传递控制器的信号下发中也是经常用到)。
   - 至此，该进程启动成功。
2. 第二步：
   - 在 generateApplicationProvidersLocked 函数内部查询（根据进程名，uid 确定）PKMS 以获取需运行在该进程中的 ContentProvider，为调用 ApplicationThread 的 **bindApplication** 做准备。

- 刚创建的这个进程并不知道自己的历史使命是什么，甚至连自己的进程名都不知道，只能设为"**< pre-initialized >**"。其实，**Android 应用进程的历史使命是 AMS 在其启动后才赋予它的**，这一点和我们理解的一般意义上的进程不太一样。根据之前的介绍，**Android 的组件应该运行在 Android 运行环境中。创建应用进程这一步只是创建了一个能运行 Android 运行环境的容器**。
- **bindApplication** 的功能就是 **创建并初始化位于该进程中的 Android 运行环境**。

3. 第三步：应用进程已经准备好了 Android 运行环境，**接着将获取 ActivityStack 中一个需要运行的 ActivityRecord 并启动，最后通知应用进程启动 Activity 和 Service 等组件**，其中**用于启动 Activity 的函数是 ActivityStack.realStartActivityLocked**。

6.7.3 ActivityStack 的 realStartActivityLocked 分析

- 它里面有两个关键函数，分别是：**scheduleLaunchActivity 和 completeResumeLocked**。
  - scheduleLaunchActivity 用于和应用进程交互，通知它启动目标 Activity。
  - completeResumeLocked 将继续 AMS 的处理流程。

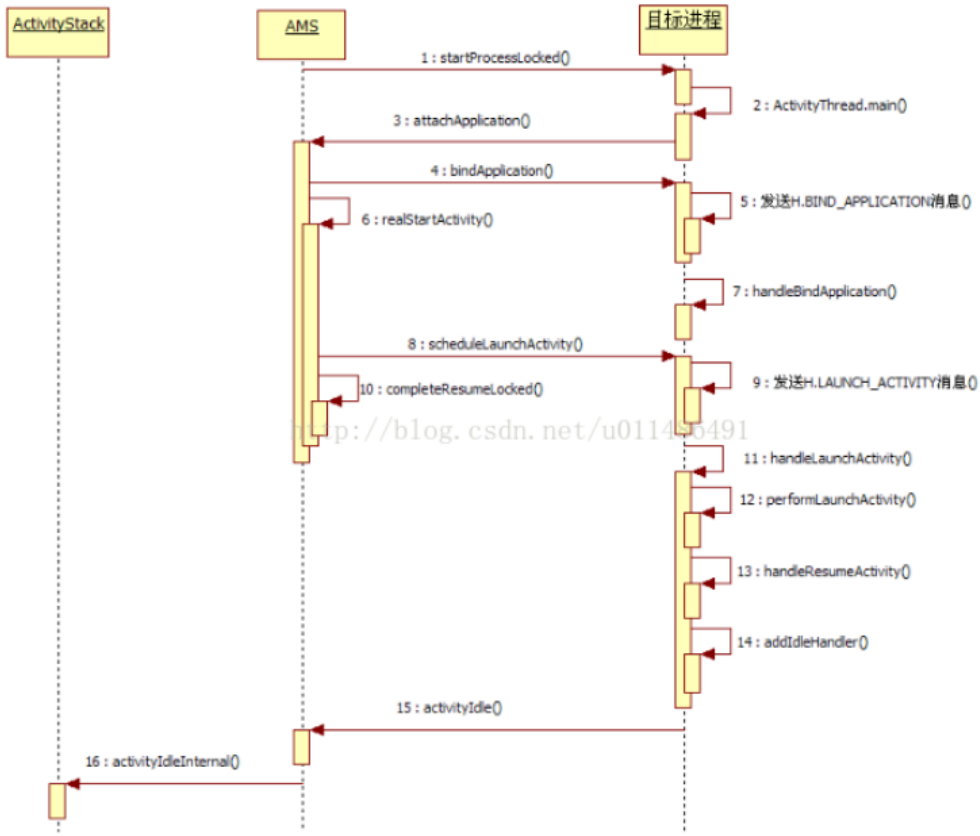
1. scheduleLaunchActivity
   - 它 **保存 AMS 发送过来的参数信息，向主线程发送消息**，该消息的处理在 **handleLaunchActivity** 中进行。在其中 **根据 ApplicationInfo 得到对应的 PackageInfo**。
   - **通过 Java 反射机制创建目标 Activity，将在内部完成 Activity 生命周期的前两步，即调用其 onCreate 和 onStart 函数**。我们的目标 com.dfp.test.TestActivity 创建完毕。
   - 调用 **handleResumeActivity** ，会 **在其内部调用目标 Activity 的 onResume 函数**。
2. completeResumeLocked
   - AMS 给了应用进程 10 秒的时间，希望它在 10 秒内调用 activityIdle 函数。这个时间不算长，和前面 AMS 等待应用进程启动的超时时间一样。
   - 在 activityIdle 中设置了请求超时处理。并调用 activityIdleInternal 函数，处理被暂停的 Activity。
   - 如果 **被暂停的 Activity 处于 finishing 状态**（例如 Activity 在其 onStop 中调用了 finish 函数），则调用 **finishCurrentActivityLocked** 。否则，要调用 **stopActivityLocked** 处理暂停的 Activity。

6.7.4 ApplicationThread 的 bindApplication 分析

- 该函数将为应用进程绑定一个 Application。
- **ApplicationThread** 接收到来自 AMS 的指令后，均会 **将指令中的参数封装到一个数据结构中**，然后 **通过发送消息的方式转交给主线程去处理**。**BIND_APPLICATION** 最终将由 **handleBindApplication** 函数处理。
- bindApplication 函数将设置一些初始化参数，其中最重要的有：
  - 1. 为之前的无名进程设置进程名，并初始化参数
  - 2. 创建一个 Application 对象，该对象是本进程中运行的第一个 Application。
  - 3. 如果该 Application 有 ContentProvider，则应安装它们。

6.8 总结:

- 在应用进程启动后，需要尽快调用 AMS 的 attachApplication 函数，该函数是这个刚呱呱坠地的应用进程第一次和 AMS 交互。此时的它还默默“无名”，连一个确定的进程名都没有。不过没关系，attachApplication 函数将根据创建该应用进程之前所保存的 ProcessRecord 为其准备一切“手续”。
- attachApplication 准备好一切后，将调用应用进程的 bindApplication 函数，在该函数内部将发消息给主线程，最终该消息由 handleBindApplication 处理。handleBindApplication 将为该进程设置进程名，初始化一些策略和参数信息等。另外，它还创建一个 Application 对象。同时，如果该 Application 声明了 ContentProvider，还需要为该进程安装 ContentProvider。



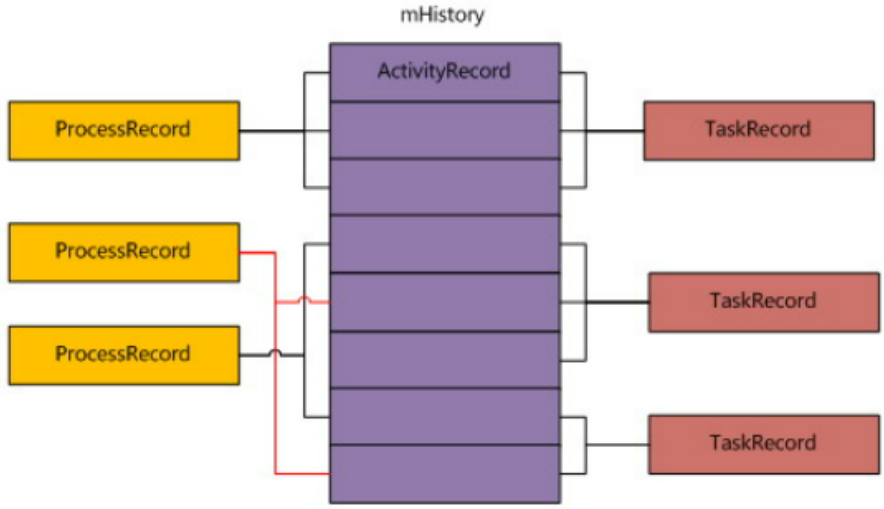
6.9 startActivity 总结

- 1. 行程的起点是 am。am 是 Android 中很重要的程序。我们利用 amstart 命令，发起本次目标 Activity 的启动请求。
- 2. 接下来进入 ActivityManagerService 和 ActivityStack 这两个核心类。对于启动 Activity 来说，这段行程又可分细分为两个阶段：第一阶段的主要工作就是根据启动模式和启动标志找到或创建 ActivityRecord 及对应的 TaskRecord；第二阶段工作就是处理 Activity 启动或切换相关的工作。
- 3. 首先 AMS 直接创建目标进程并运行 Activity 的流程，其中涉及目标进程的创建，在目标进程中 Android 运行环境的初始化，目标 Activity 的创建以及触发 onCreate、onStart 及 onResume 等其生命周期中重要函数调用等相关知识点。
- 4. 接着 AMS 先 pause 当前 Activity，然后再创建目标进程并运行 Activity 的流程。其中牵扯到两个应用进程和 AMS 的交互，其难度之大可见一斑。

7 task 管理

7.1 task 和 activity 以及 process 整体的关系

- AMS 提供了一个 ArrayList mHistory 来管理所有的 activity，activity 在 AMS 中的形式是 ActivityRecord，task 在 AMS 中的形式为 TaskRecord，进程在 AMS 中的管理形式为 ProcessRecord。如下图所示



- 从图中我们可以看出如下几点规则：
  - 1. 所有的 ActivityRecord 会被存储在 mHistory 管理；
  - 2. 每个 ActivityRecord 会对应到一个 TaskRecord，并且有着相同 TaskRecord 的 ActivityRecord 在 mHistory 中会处在连续的位置；
  - 3. 同一个 TaskRecord 的 Activity 可能分别处于不同的进程中，每个 Activity 所处的进程跟 task 没有关系；
- 因此，在分析 Activity 管理之前，先了解一下这个规则。
- 在 android 应用开发中，task 是一个很重要的概念，在文章开始，我就画出了 task 和 activity 以及 process 整体的关系，在这里还需要说明一下，task 和 application 的区别，
- application 在 android 中的作用仅仅是 activity 在未被使用前的一个容器，我们开发 android 应用程序时，需要一个 application 来组织我们开发的 activity，application 和 activity 之间是一个静态关系，并且是一一对应的关系；也就是说我们开发的 activity 在 PM 中的最终形式是唯一的，永远对应一个 application。
- 而 task 和 activity 之间的关系是动态的关系，是我们在运行应用程序时，activity 的调用栈，同一个 task 中的 activity 可能来自不同的 application。

7.2 1 Intent.FLAG_ACTIVITY_NO_USER_ACTION

- 检查 Intent 是否设置了 Intent.FLAG_ACTIVITY_NO_USER_ACTION，如果设置了，则在 activity pause 之前将不再调用 activity 的 onUserLeaveHint() 方法。

```
mUserLeaving = (launchFlags&Intent.FLAG_ACTIVITY_NO_USER_ACTION) == 0;
if (DEBUG_USER_LEAVING) Slog.v(TAG, "startActivity() => mUserLeaving=" + mUserLeaving);
```

7.3 2 Intent.FLAG_ACTIVITY_PREVIOUS_IS_TOP

- 检查 Intent 是否设置了 Intent.FLAG_ACTIVITY_PREVIOUS_IS_TOP，这个标志我有点困惑，从它的注释可以看出 **它的含义是指如果设置了该 flag，那么 mHistory 中最 top 的 activity 在后续的处理中将不被视为 top，而将前一个 activity 视为 top，如 A–>B–>C，将 B 视为 top**。

- 这个 top activity 的作用很大，涉及到后面对 task 的处理。但是目前来看这个 flag 并没有起到该有的作用，代码中判断如果设置了该标志，那么 AMS 将会视当前正在启动的 activity 为 top，然后去 mHistory 中去查找它的前一个 activity 为后续 task 处理的 top activity(topRunningNonDelayedActivityLocked() 中实现)，但是现在的问题是此时此刻，正在启动的 activity 并不存在于 mHistory 中，因为我们在前一个函数中刚刚创建了这个 ActivityRecord。如下面代码所示：

```
ActivityRecord notTop = (launchFlags&Intent.FLAG_ACTIVITY_PREVIOUS_IS_TOP) != 0 ? r : null;
```

- 因此，感觉这个 flag 的意义不是太大。

7.4 3 何时应该创建新的 task

- sourceRecord 为空；

- sourceRecord 的 activity 的 launch mode 为 ActivityInfo. **LAUNCH_SINGLE_INSTANCE** ，也就是 sourceRecord activity 的 task 只允许一个 activity；

- 当前 activity 的 launch mode 为 ActivityInfo. **LAUNCH_SINGLE_INSTANCE** 或者 r.launchMode == ActivityInfo. **LAUNCH_SINGLE_TASK** 。

- **以上几种情况**，均可视为需要为启动的 activity 创建一个新的 task.

```
if (sourceRecord == null) {
// This activity is not being started from another...  in this
// case we -always- start a new task.
    if ((launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) == 0) {
        Slog.w(TAG, "startActivity called from non-Activity context; forcing Intent.FLAG_ACTIVITY_NEW_TASK for: "
              + intent);
        launchFlags |= Intent.FLAG_ACTIVITY_NEW_TASK;
    }
} else if (sourceRecord.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE) {
// The original activity who is starting us is running as a single
// instance...  this new activity it is starting must go on its
// own task.
    launchFlags |= Intent.FLAG_ACTIVITY_NEW_TASK;
} else if (r.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE
           || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TASK) {
// The activity being started is a single instance...  it always
// gets launched into its own task.
    launchFlags |= Intent.FLAG_ACTIVITY_NEW_TASK;
}
```

7.5 4 Intent.FLAG_ACTIVITY_NEW_TASK 时断开与 Caller 依赖

- 如果启动的 activity 需要新的 task，那么新启动的 activity 将会与其 caller 断开依赖关系，这个关系主要是指 result 反馈，A–>B, 如果 A 是通过 startActivityForResult() 请求启动的，并且 requestCode >=0, 那么如果 B 是在新的 task 中，那么 B 在 finish 的时候将不再向 A 反馈 result，而是在启动过程中就会向 A 反馈一个 RESULT_CANCELED。

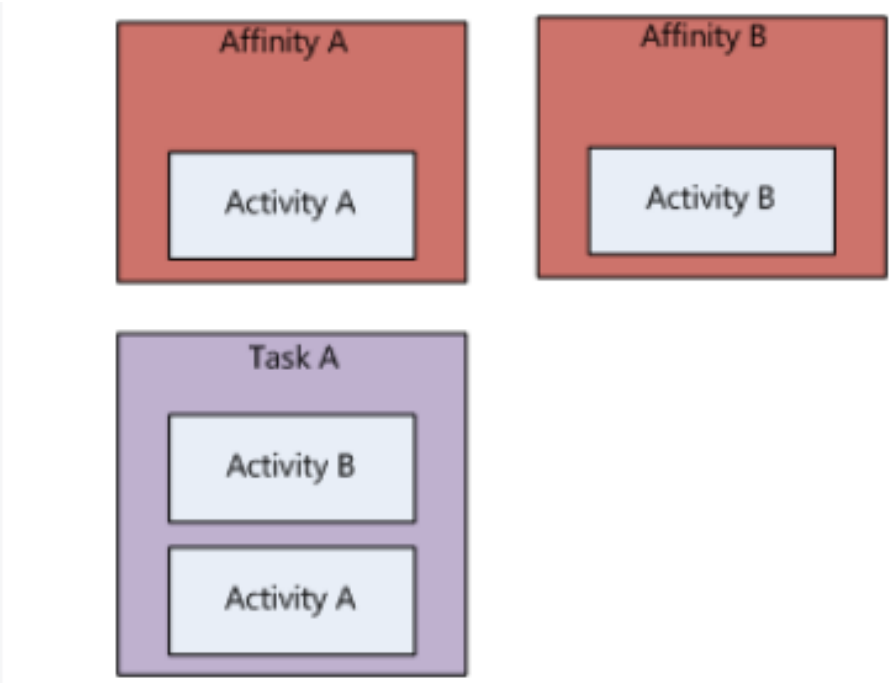
```
if (r.resultTo != null && (launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) != 0) {
// For whatever reason this activity is being launched into a new
// task...  yet the caller has requested a result back.  Well, that
// is pretty messed up, so instead immediately send back a cancel
// and let the new task continue launched as normal without a
// dependency on its originator.
    Slog.w(TAG, "Activity is launching as a new task, so cancelling activity result.");
    sendActivityResultLocked(-1,
```

```
                            r.resultTo, r.resultWho, r.requestCode,
                            Activity.RESULT_CANCELED, null);
    r.resultTo = null;
}
```

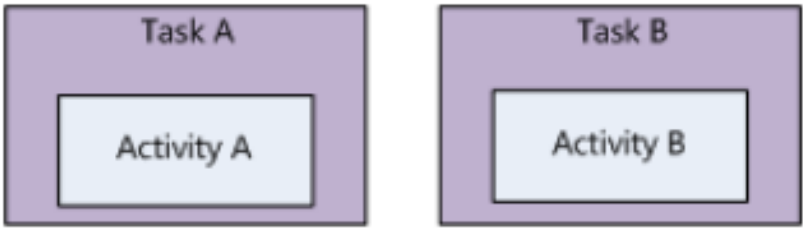
7.6 5 Task 复用

7.6.1 5.1 Task 的基本属性

- 检查 mHistory 中是否有 task 可复用，在分析这段之前，先了解一下 **task 的一些基本概念**
  - task 的 root activity 是指如果一个 activity 启动时创建的了一个新的 task，那么这个 activity 是 task 的 root activity；
  - task.affinity 是指 root activity 的 affinity；
  - task.intent 是指启动 root activity 的 Intent；
  - task.affinityIntent 是指 activity 在进行了 TaskReparenting 之后，AMS 为 activity 分配了新的 task，该 task 的 affinityIntent 则是启动该 activity 时的 Intent，此时 task.intent==null。
- TaskReparenting 操作举例说明一下，假如有 2 个 activity 拥有不同的 affinity，且自 Activity A 中启动 Activity B，假如 Activity A 是所在 task 的 root activity，如下图示：



- 假如 Activity B 设置了 ActivityInfo.FLAG_ALLOW_TASK_REPARENTING，那么如果此时另外一个 application 启动了 Activity B 并要求其在新的 task 中，那么此时的 Activity B 将被从 Task A 中移动到新的 task 中，如下图所示：



- 这个过程就称之为 TaskReparenting，关于 TaskReparenting，我会专门写一篇文章分析一下。下面来分析 task 复用的过程。

7.6.2 5.2 查找可复用的 task

- 以下 3 种条件需要检查是否有有 task 可复用

```
(launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) != 0 &&
    (launchFlags&Intent.FLAG_ACTIVITY_MULTIPLE_TASK) == 0;
r.launchMode == ActivityInfo.LAUNCH_SINGLE_TASK
r.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE
```

- 第 是一个组合条件，Intent. **FLAG_ACTIVITY_MULTIPLE_TASK** 不能单独使用，它 **是和 Intent.FLAG_ACTIVITY_NEW_TASK 结合起来使用的，如果设置了 Intent.FLAG_ACTIVITY_MU 那么将会永远启动一个新的 task，不管是否有可复用的 task**。
- 为什么是这 3 种条件，从 android 的开发文档中，我们可知 LAUNCH_SINGLE_TASK 和 LAUNCH_SINGLE_INSTANCE 两种 launch mode 的 activity 只允许作为 task 的 root activity，既然是作为 root activity，那么它所处的 task 的 affinity 必然是和它的是一样的，因此从 mHistory 中找打一个和自己的 affinity 相同的 task 是非常有必要的。
- 而对于设置了 Intent.FLAG_ACTIVITY_NEW_TASK 的 Intent 来说，并且没有设置 Intent.FLAG_ACTI 那么同样的，它也必须是作为它所处的 task 的 root activity。道理是一样的。

1. AMS 去 mHistory 中查找可复用的 task，查找这个 task 的规则如下：

   (a) launchMode != ActivityInfo.LAUNCH_SINGLE_INSTANCE 的情况，遵循如下规则：@findTaskLocked()

   - 查找 mHistory 中是否有与要启动的 activity 相同 affinity 的 task，上面也提过这几类 activity 启动时，均是作为 task 的 root activity，并且其 task 的 affinity 必须和自己的 affinity 相同，因此首先需要去 mHistory 查找和自己 affinity 相同的 task。
   - 如果 activity 没有 affinity，即属性 android:taskAffinity 设置为 “”，空字符串时。此时 AMS 就会去 mHistory 中去查找是否有 task 的 root activity 和启动的 activity 相同，通过比较 **task.intent.getComponent()** 和 **启动 activity 的 Comeponent** 比较，为什么是 root activity，前面分析过了；
   - 如果 task.Intent 为空，这种情况发生在 TaskReparenting 之后，TaskReparenting 之后，AMS 为这个 activity 创建一个新的 task，并将启动这个 activity 的 Intent 赋值给 task.affinityIntent，并且此时的 task.Intent==null。此时就需要比较 **task.affinityIntent.getComponent()** 和 **启动 activity 的 Comeponent** 比较，看是否和启动的 activity 相同。
   - **以上 3 个规则中**，均是返回找的 task 中最上面的 activity，而不一定是 task 的 activity，至于如何处理要启动的 activity 和 task 中已有的 activity，后面会介绍。

   (b) launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE 的情况，遵循如下规则：@findActivityLocked()

   - 对于 ActivityInfo. **LAUNCH_SINGLE_INSTANCE** 启动模式来说，**它所处的 task 中只允许有它一个 activity** ，因此它的规则只符合上面规则中的 二；

- 对于第 条，由于设置了 ActivityInfo.LAUNCH_SINGLE_INSTANCE 启动模式的 activity，它只能自己独处一个 task，不可能和别人共享同一个 task，因此 mHistory 即使存在了与该 activity 有相同的 affinity 的 activity，如果这个 activity 和启动的 activity 不同,那么 ActivityInfo.LAUNCH_SINGLE_INSTANCE 启动模式的 activity 也不可能和它共用一个 task，因此这第 条完全可以不用检查。
- 对于第 条，由于该模式的 activity 独处一个 task，因此完全没有可能所处的 task 的 affinity 和自己的 affinity 不同，因此，假如 mHistory 存在相同的 activity 与启动的 activity 相同，那么这个 activity 的 affinity 必然和自己的相同。所以对于这种模式，第 条囊括了其他模式的 两条。
- 对于第 条，同样的道理，ActivityInfo.LAUNCH_SINGLE_INSTANCE 启动模式的 activity 不可能处在与自己不同 affinity 的 task 中，因此不可能出现 TaskReparenting 操作，所以这条也不需要。

```
ActivityRecord taskTop = r.launchMode != ActivityInfo.LAUNCH_SINGLE_INSTANCE
    ? findTaskLocked(intent, r.info)
    : findActivityLocked(intent, r.info);
```

- 在获得 taskTop 之后，下面来分析这个 taskTop 的意义。

7.6.3 5.3 task 移到 mHistory 前端

- 由于我们要复用 task，因此需要将 taskTop 所在的 task 移到 mHistory 前端。

```
ActivityRecord curTop = topRunningNonDelayedActivityLocked(notTop);
if (curTop.task != taskTop.task) {
    r.intent.addFlags(Intent.FLAG_ACTIVITY_BROUGHT_TO_FRONT);
    boolean callerAtFront = sourceRecord == null
        || curTop.task == sourceRecord.task;
    if (callerAtFront) {
// We really do want to push this one into the
// user's face, right now.
        moveTaskToFrontLocked(taskTop.task, r);
    }
}
```

7.6.4 5.4 Reset Task

- 如果 Intent 设置 Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED，最常见的情况，当从 Home 启动应用程序时，会设置这个 flag；从 recently task 进入应用程序，则不会设置这个 falg。
- 设置了 FLAG_ACTIVITY_RESET_TASK_IF_NEEDED，AMS 会对复用的 task 作如下处理，下面称这个可复用的 task 为复用 task：

1. 设置了 FLAG_ACTIVITY_RESET_TASK_IF_NEEDED，AMS 会对复用的 task 作如下处理

   (a) mHistory 中，对于复用 task 中的除 root activity 外的 activity，有如下处理

   - 在此之前，先介绍 **activity 的几个关键属性：**
     - 一 如果 **复用 task 在后台时间超过 30min** ，那么 **在这个过程中将删除除 root activity 之外的所有的 activity** ；
     - 二 如果 **新启动的 activity** 设置了属性 ActivityInfo. **FLAG_ALWAYS_RETAIN_TASK_** ，那么表明 **它并不要求后台 20min 的复用 task 删除 activity;**
     - 三 如果 **新启动的 activity** 设置了属性 ActivityInfo. **FLAG_CLEAR_TASK_ON_LAUN** ，那么表明 **不论复用 task 在后台是否超过 30min，一律要求删除除 root activity 之外的所有的 activity;**
     - 四 **复用 task 中的 activity** 设置了属性 ActivityInfo. **FLAG_FINISH_ON_TASK_LAU** ，那么 **复用 task 从 home 中再次被启动到前台时，这个 activity 会被删除;**

- 五 **复用 task 中的 activity** 设置了属性 ActivityInfo. **FLAG_ALLOW_TASK_REPAREN**，并且这个 **activity 的 resultTo 为空**，那么也就是说这个 activity 和它的 caller 没有依赖关系，那么 AMS 认为这个 activity 暂时没有用处了，**需要对其进行 TaskReparenting 操作，最好的方法是把它放到 mHistory 栈底，不影响其他 task**。
- 六 **复用 task 中的 activity** 的 Intent 设置属性 Intent. **FLAG_ACTIVITY_CLEAR_WH**，那么下次再从 home 中进入到 task 中，那么 **将删除设置了该属性的 activity 以上所有的 activity**，例如 A–>B–>C ==> D–>E, 加入在 C 启动 D 时设置了该属性，那么下次从 HOME 中再次进入到这个 task 中时，将会是 A–>B–>C。
- 七 如果 **复用 task 中的 activity 的 resultTo 不为空**，也就是启动这个 activity 的是一个 activity，那么 **这个 activity 的处理将按照它的前一个 activity 的处理方式来处理**，不管 **在何时情况下，它的前一个 activity 都是启动它的 activity**，(即便 resultTo 不是前一个 activity，如设置了 Intent. **FLAG_ACTIVITY_FORWARD_RESULT**) 。如果复用 task 中每个 activity 的 resultTo 都不为空，并且上述处理优先级在其前面的属性没有设置的话，那么 **这个复用 task 中的 activity 将不作任何的处理 (这种情况没有看懂，不知道这种情况为什么可以不用处理 ???)** 。

- 一般情况下,activity 的 resultTo 都不为空,除非设置了 Intent. **FLAG_ACTIVITY_FORWAR**，那么此时被启动的 activity 的 caller 的 resultTo 将会为空。
- task 中的 activity 的属性设置是上述属性的组合，因此 **reset task 过程要按照一定的优先级来处理**，上述属性的处理优先级是: **六 = 四 > 七 > 五 > 三 = 二 > 一**
  - 具体操作顺序如下:
  - 根据六，四 条件来 **删除复用 task 中相应的 activity;**
  - 七 条件下，将会 **暂时不做处理，再根据它的前一个 activity 的属性来做处理，即使这个 activity 设置了 allowTaskReparenting;**
  - **如果 activity 的 resultTo 为空，并且满足条件五**，那么 **将其及其以上未作处理的，满足条件七 的所有 activity，一并进行 TaskReparenting 操作，并放置在 mHistory 栈底。它们在 mHistory 栈底顺序如同在复用 task 中的顺序;**

  根据一 二 三 的条件来删除复用 task 中相应的 activity。

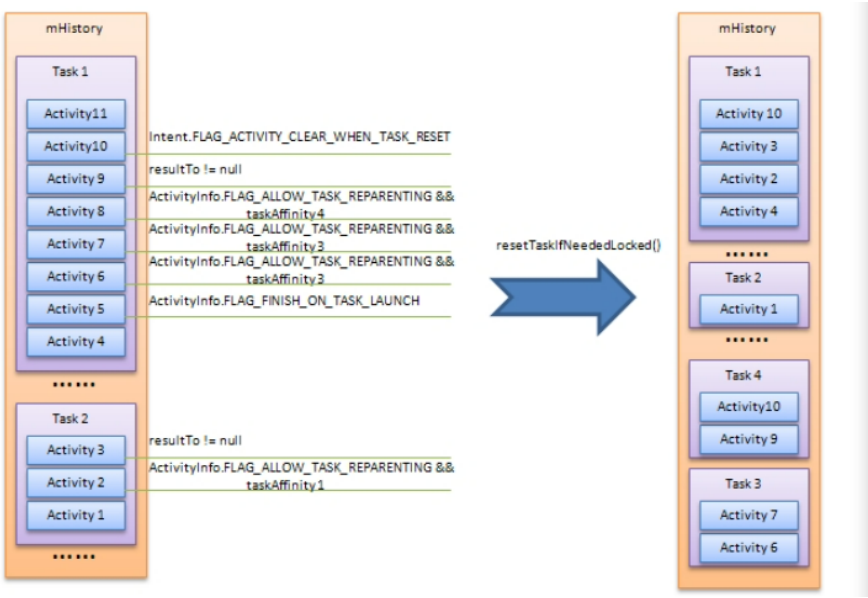
(b) mHistory 中，不属于复用 task 的 activity，并且它的 resultTo 不为空，那么将根据它的前一个 activity 的处理来处理;

(c) mHistory 中，不属于复用 task，但是和当前启动的 activity 有相同 affinity，并且允许 TaskReparenting 操作，那么将进行以下操作:

- 如果满足上述的 的条件，但是其中的 task 不是复用 task，而是这个 activity 所处的 task，那么将输出这个 activity，而不是进行 TaskReparenting 操作。
- 为什么非复用 task 中的 activity，和当前启动的 activity 有相同 affinity，并且允许 TaskReparenting 操作，满足了 的条件之后要删除呢，为什么非复用 task 中的其他 activity，不需要删除呢?
- **正因为它和启动的 activity 有相同的 affinity，因此 AMS 认为这个 activity 是和启动 activity 相关的，以后可能会重新调用，所以当其满足删除条件后，这时它将不允许 TaskReparenting 操作，并且不应该再允许它存在于其他的 task 中，此时应该删除**。
- 如果没有满足 的条件，那么将会对其进行 TaskReparenting 操作，重新将其移动到复用 task 或新启动的 task 中。

```
// If the caller has requested that the target task be
// reset, then do so.
if ((launchFlags&Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED) != 0) {
    taskTop = resetTaskIfNeededLocked(taskTop, r);
}
```

- 上面就是这个复用 task 的 reset 过程，它的执行过程是按照上述的 的顺序来执行的，下面给出一个图示，便于更好的理解 reset task 的过程。



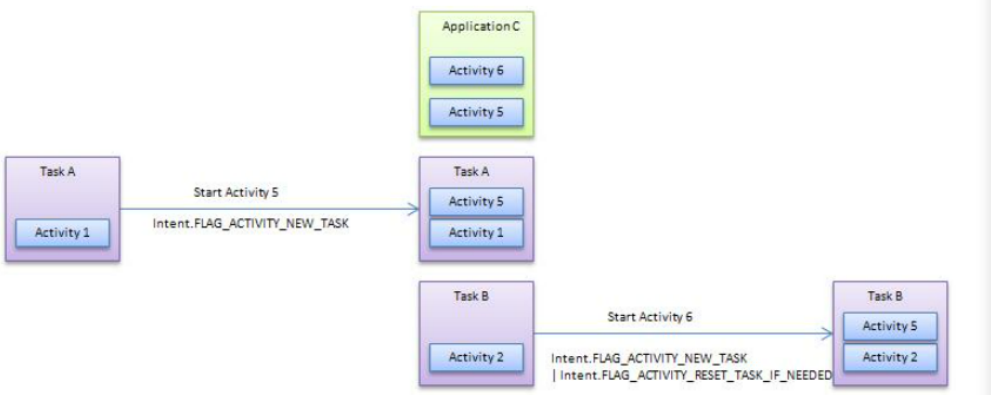
7.6.5 5.5 判断是否有可复用的 activity

- 如果 mHistory 中有可复用的 task，那么在某些情况下并不需要启动这个 activity，下面分析具体是什么情况：
- Intent 设置了 Intent.FLAG_ACTIVITY_CLEAR_TOP，或者 launchMode = ActivityInfo.LAUNCH_S 或者 r.launchMode = ActivityInfo.LAUNCH_SINGLE_INSTANCE；这 3 种条件有一个共同点，就是启动的 activity 启动之后，在这个 task 中，这个 activity 之上不能有其他的 activity。
  - 一般情况下，需要将复用 task 中启动的 activity 之上的所有的 activity 删除，
  - 当 activity 的 launchMode == ActivityInfo. **LAUNCH_MULTIPLE，即普通模式**，并且 **Intent 并未要求 singletop 模式**，这种情况是 **连复用 task 中与启动 activity 相同的 activity 都要删除，也就是不希望复用相同的 activity**。
  - **performClearTaskLocked()** 实现了上述功能，并返回可复用的 activity。

```
ActivityRecord top = performClearTaskLocked(
    taskTop.task.taskId, r, launchFlags, true);
```

- 如果有可复用的 activity，并且这个 activity 是 task 的 root activity，由于 task 的 Intent 是 root activity 的 Intent，所以需要重新设置 task 的 Intent。
- 向可复用的 activity 发送新的 Intent，通知它 Intent 的变化，最终会调用到这个 activity 的 onNewIntent() 方法。
- 如果不满足 条件的话，但是启动的 activity 与复用 task 的 root activity 相同。
  - 如果此时 Intent 设置了 Intent.FLAG_ACTIVITY_SINGLE_TOP，并且复用 task 的 top activity 正好是要启动的 activity，则复用这个 activity，同时更新 activity 的 Intent，如果需要更新 task 的 Intent。
  - 如果 Intent 没有设置了 Intent.FLAG_ACTIVITY_SINGLE_TOP，即使设置了，但是当前的 top activity 不是正要启动的 activity，那么会判断当前启动的 Intent 和 task 的 Intent 不同，那么将会重新启动这个 activity。
  - 其他情况，将直接 resume top 的 activity。

- 如果 条件均不满足，其实如果不满足 条件的话，复用的 task 中就不存在与启动的 activity 相同的 activity 了，如果启动的 Intent 没有设置 Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED，那么一定不会复用任何的 activity。
- (4) 如果 条件均不满足，并且 Intent 设置了 Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED，那么需要检查当前复用 task 的 Intent 是否设置了 Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDE
- 如果没有设置，重新设置新的 Intent，同样不可能复用 activity。
- 这种情况下，将不会显示要启动的 activity，而是改为显示复用的 task 中的内容，如下图：



- 至此，整个 Task 复用，已经 activity 复用的过程就介绍完了，如果没有可复用的 activity，则需要启动一个新的 activity，如果有可复用的 activity，那么 activity 的启动过程至此结束，直接调用 resumeTopActivityLocked()resume top 的 activity 即可。
- **以后的处理均为 Task 复用和 activity 复用失败之后的处理**。

7.7 6 singleTop 和 singleTask 属性的处理

- 这一部分是针对 singleTop 和 singleTask 属性的处理，前面分析 Task 复用的时候，也有对 singleTop 和 singleTask 属性的处理，两者有什么不同呢?
- Task 复用中是启动的 activity 需要在新的 task 中，而这里的处理主要是针对同一个 task。
- 当设置 Intent.FLAG_ACTIVITY_SINGLE_TOP 或者 launchMode = ActivityInfo.LAUNCH_SINGLE_ 或者 launchMode = ActivityInfo.LAUNCH_SINGLE_TASK 这几种情况下，如果 top activity 与启动的 activity 为同一个 activity，那么将复用 top activity，并直接 resume top activity。

```
if (top != null && r.resultTo == null) {
    if (top.realActivity.equals(r.realActivity)) {
        if (top.app != null && top.app.thread != null) {
            if ((launchFlags&Intent.FLAG_ACTIVITY_SINGLE_TOP) != 0
                || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TOP
                || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TASK) {
                logStartActivity(EventLogTags.AM_NEW_INTENT, top, top.task);
// For paranoia, make sure we have correctly
// resumed the top activity.
                if (doResume) {
                    resumeTopActivityLocked(null);
                }
                if (onlyIfNeeded) {
                    // We don't need to start a new activity, and
                    // the client said not to do anything if that
                    // is the case, so this is it!
                    return START_RETURN_INTENT_TO_CALLER;
                }
                top.deliverNewIntentLocked(callingUid, r.intent);
                return START_DELIVERED_TO_TOP;
            }
```

```
        }
    }
}
```

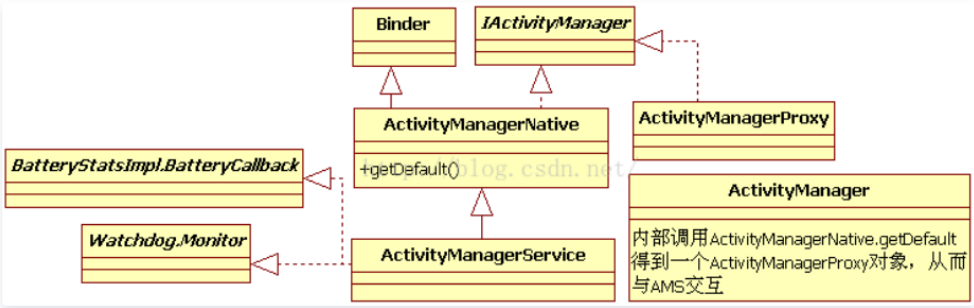
- r.resultTo == null 这个条件是在 startActivityForResult() 的 requestCode<0 时成立。
- 为什么没有 ActivityInfo. **LAUNCH_SINGLE_INSTANCE**？这是 **因为这种启动模式，如果 Task 复用失败之后，直接启动为其启动一个 Intent.FLAG_ACTIVITY_NEW_TASK 即可**。

7.8 7 standard 和 singleInstance 模式

- 为什么代码中没有明显的针对 ActivityInfo.LAUNCH_SINGLE_INSTANCE 模式的处理? 这是因为这种启动模式,如果 Task 复用失败之后,直接启动为其启动一个 Intent.FLAG_ACTIVITY_NEW_T 即可。
- 设置了 Intent.FLAG_ACTIVITY_NEW_TASK，则为该 activity 创建一个新的 task;
- 在当前的 task 中启动新的 activity,
  - 一 当前的 caller 是一个 activity，如果设置 Intent.FLAG_ACTIVITY_CLEAR_TOP，当前的 task 如果存在要启动的 activity(这个和上一节中的 Task 复用时的 clear top 过程不同，两者是互斥的过程，不冲突)，清除其上的所有的 activity;
  - 二 当前的 caller 是一个 activity,如果设置 Intent.FLAG_ACTIVITY_REORDER_TO_FRONT，这个 flag 表示如果启动的 activity 已经在当前的 task 中，那么如果当前启动的 Intent 设置了该 flag，那么则会将这个 activity 从 task 中移动到 top。
    - 如果 A–>B–>C–>D，D 启动 B 时，设置了该 flag，那么将变为 A–>C–>D–>B
    - 两个条件，则不需要再启动新的 activity，直接 resume top。
  - 三 当前的 caller 是一个 activity，其他情况则需要启动新的 activity。
- 当前的 caller 不是 activity，那么仍将新启动的 activity 放在 top 的 task 中。

8 AMS 在 Android 起到什么作用？简单的分析下 Android 的源码

- https://cloud.tencent.com/developer/article/1466430
- 概述
- 相信大多数动脑同学对文章中提到的 ActivityManagerService（以后简称 AMS）都有所耳闻。
- **AMS 是 Android 中最核心的服务**，主要负责 **系统中四大组件的启动、切换、调度及应用进程的管理和调度等工作，其职责与操作系统中的进程管理和调度模块相类似，因此它在 Android 中非常重要**。AMS 是碰到的第一块难啃的骨头 []，涉及的知识点较多。为了帮助读者更好地理解 AMS，接下来将带小伙伴么按五条不同的线来分析它。
  - **第一条线**：同其他服务一样，将分析 SystemServer 中 AMS 的调用轨迹。
  - **第二条线**：以 am 命令启动一个 Activity 为例，分析应用进程的创建、Activity 的启动，以及它们和 AMS 之间的交互等知识。
  - **第三条线和第四条线**：分别以 Broadcast 和 Service 为例，分析 AMS 中 Broadcast 和 Service 的相关处理流程。
  - **第五条线**：以一个 Crash 的应用进程为出发点，分析 AMS 如何打理该应用进程的身后事。除了这五条线外，还将统一分析在这五条线中频繁出现的与 AMS 中应用进程的调度、内存管理等相关的知识。提示 ContentProvider 将放到下一章分析，不过本章将涉及和 ContentProvider 有关的知识点。先来看 AMS 的家族图谱:



- 由图可知：
  - AMS 由 ActivityManagerNative（以后简称 AMN）类派生，并实现 Watchdog.Monitor 和 BatteryStatsImpl.BatteryCallback 接口。而 AMN 由 Binder 派生，实现了 IActivityManager 接口。
  - 客户端使用 ActivityManager 类。由于 AMS 是系统核心服务，很多 API 不能开放供客户端使用，所以设计者没有让 ActivityManager 直接加入 AMS 家族。在 ActivityManager 类内部通过调用 AMN 的 getDefault 函数得到一个 ActivityManagerProxy 对象，通过它可与 AMS 通信。
  - AMS 由 SystemServer 的 ServerThread 线程创建;

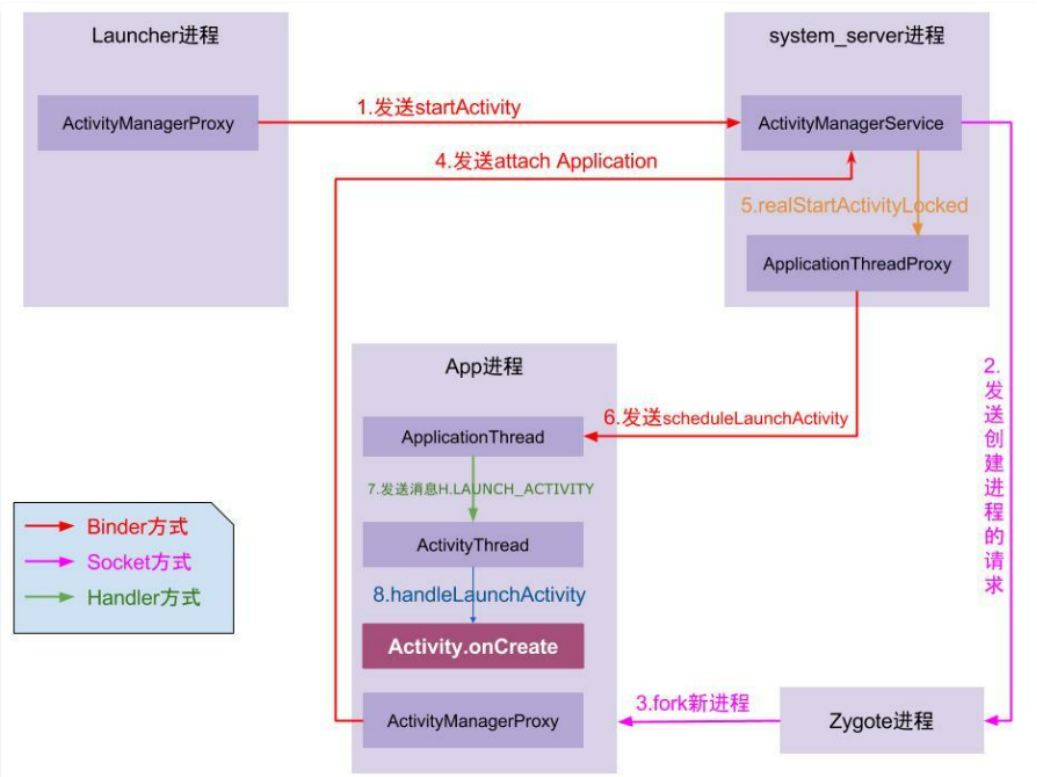
8.1 1. 初识 ActivityManagerService 总结

- 本节所分析的 4 个关键函数均较复杂，与之相关的知识点总结如下：
  - AMS 的 main 函数：创建 AMS 实例，其中最重要的工作是创建 Android 运行环境，得到一个 ActivityThread 和一个 Context 对象。
  - AMS 的 setSystemProcess 函数：该函数注册 AMS 和 meminfo 等服务到 ServiceManager 中。另外，它为 SystemServer 创建了一个 ProcessRecord 对象。由于 AMS 是 Java 世界的进程管理及调度中心，要做到对 Java 进程一视同仁，尽管 SystemServer 贵为系统进程，此时也不得不将其并入 AMS 的管理范围内。
  - AMS 的 installSystemProviders：为 SystemServer 加载 SettingsProvider。
  - AMS 的 systemReady：做系统启动完毕前最后一些扫尾工作。该函数调用完毕后，HomeActivity 将呈现在用户面前。对 AMS 调用轨迹分析是我们破解 AMS 的第一条线，希望读者反复阅读，以真正理解其中涉及的知识点，尤其是和 Android 运行环境及 Context 相关的知识。

8.2 2. startActivity

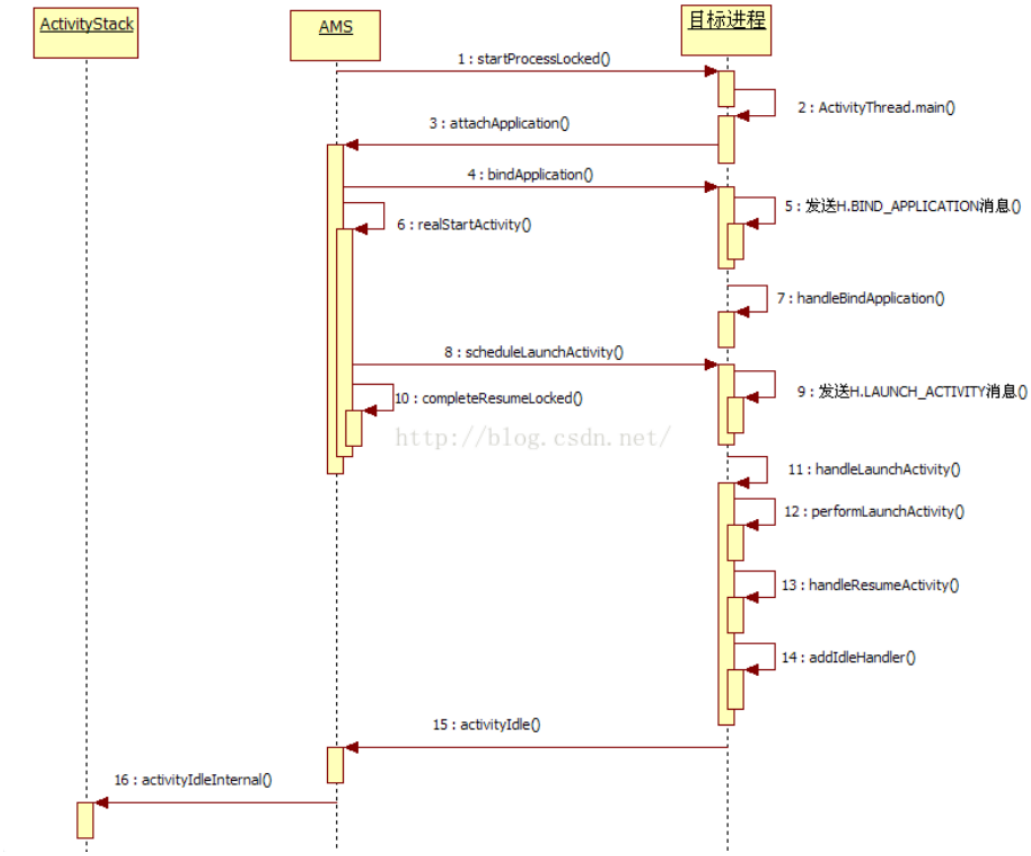
- **总结**本文详细 startActivity 的整个启动流程，
  - 流程 [2.1 ~2.4]: 运行在调用者所在进程，比如从桌面启动 Activity，则调用者所在进程为 launcher 进程，launcher 进程利用 ActivityManagerProxy 作为 Binder Client，进入 system_server 进程 (AMS 相应的 Server 端)。
  - 流程 [2.5 ~2.18]: 运行在 system_server 系统进程，整个过程最为复杂、核心的过程，下面其中部分步骤：
  - 流程 [2.7]：会调用到 resolveActivity()，借助 PackageManager 来查询系统中所有符合要求的 Activity，当存在多个满足条件的 Activity 则会弹框让用户来选择;
  - 流程 [2.8]：创建 ActivityRecord 对象，并检查是否运行 App 切换，然后再处理 mPendingActivityLaunches 中的 activity;
  - 流程 [2.9]：为 Activity 找到或创建新的 Task 对象，设置 flags 信息;

- 流程 [2.13]：当没有处于非 finishing 状态的 Activity，则直接回到桌面；否则，当 mResumedActivity 不为空则执行 startPausingLocked() 暂停该 activity; 然后再进入 startSpecificActivityLocked() 环节；
- 流程 [2.14]：当目标进程已存在则直接进入流程 [2.17]，当进程不存在则创建进程，经过层层调用还是会进入流程 [2.17]；
- 流程 [2.17]：system_server 进程利用的 ATP(Binder Client)，经过 Binder，程序接下来进入目标进程。
- 流程 [2.19 ~2.18]: 运行在目标进程，通过 Handler 消息机制，该进程中的 Binder 线程向主线程发送 H.LAUNCH_ACTIVITY，最终会通过反射创建目标 Activity，然后进入 onCreate() 生命周期。从另一个角度下图来概括：



- **启动流程：**
  - 点击桌面 App 图标，Launcher 进程采用 Binder IPC 向 system_server 进程发起 startActivity 请求；
  - system_server 进程接收到请求后，向 zygote 进程发送创建进程的请求；
  - Zygote 进程 fork 出新的子进程，即 App 进程；
  - App 进程，通过 Binder IPC 向 sytem_server 进程发起 attachApplication 请求；
  - system_server 进程在收到请求后，进行一系列准备工作后，再通过 binder IPC 向 App 进程发送 scheduleLaunchActivity 请求；
  - App 进程的 binder 线程（ApplicationThread）在收到请求后，通过 handler 向主线程发送 LAUNCH_ACTIVITY 消息；
  - 主线程在收到 Message 后，通过发射机制创建目标 Activity，并回调 Activity.onCreate() 等方法。到此，App 便正式启动，开始进入 Activity 生命周期，执行完 onCreate/onStart/onResume 方法，UI 渲染结束后便可以看到 App 的主界面。

8.3 startActivity 后半程总结



- **starActivity 总结**

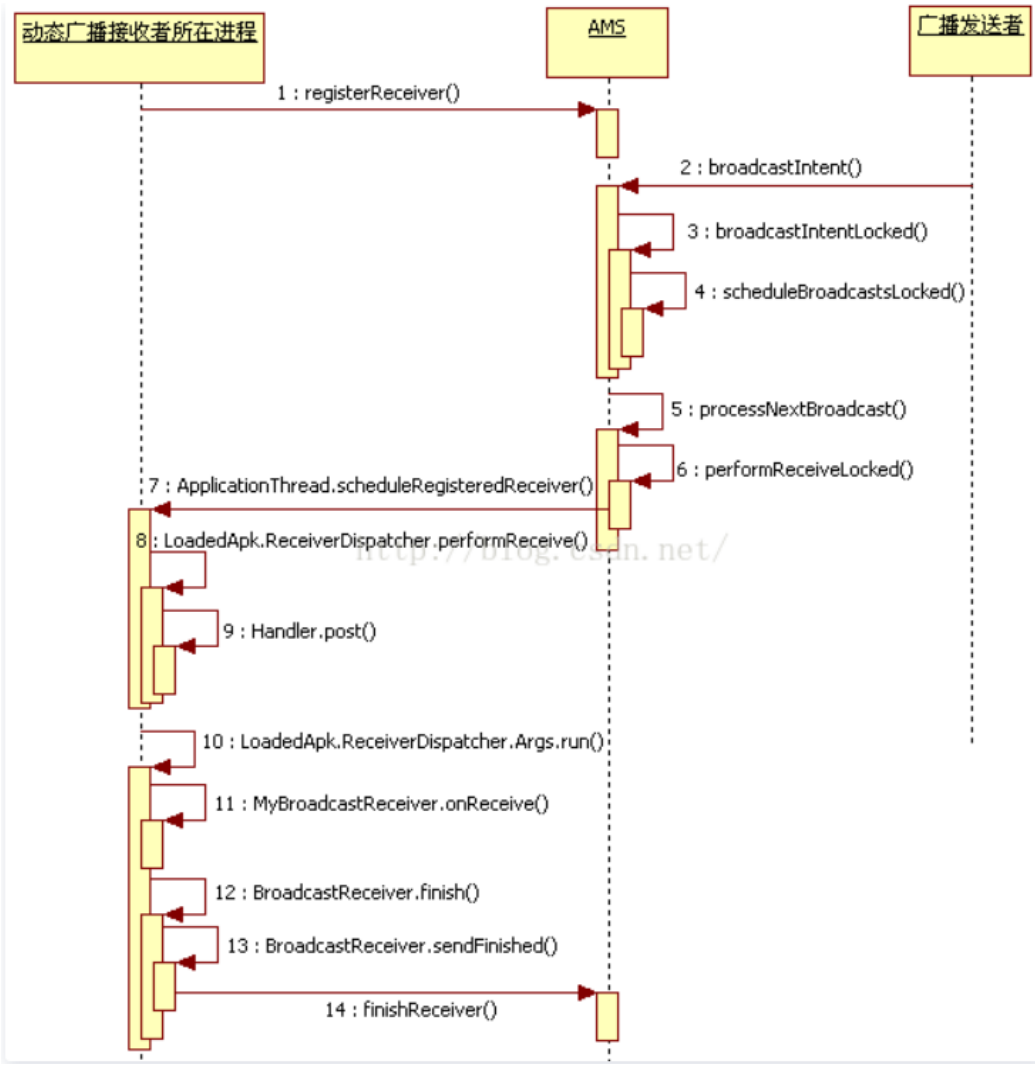
- Activity 的启动就介绍到这里。这一路分析下来，相信读者也和笔者一样觉得此行绝不轻松。先回顾一下此次旅程：

  - 行程的起点是 am。am 是 Android 中很重要的程序，读者务必要掌握它的用法。我们利用 am start 命令，发起本次目标 Activity 的启动请求。
  - 接下来进入 ActivityManagerService 和 ActivityStack 这两个核心类。对于启动 Activity 来说，这段行程又可分细分为两个阶段：第一阶段的主要工作就是根据启动模式和启动标志找到或创建 ActivityRecord 及对应的 TaskRecord；第二阶段工作就是处理 Activity 启动或切换相关的工作。
  - 首先讨论了 AMS 直接创建目标进程并运行 Activity 的流程，其中涉及目标进程的创建，在目标进程中 Android 运行环境的初始化，目标 Activity 的创建以及触发 onCreate、onStart 及 onResume 等其生命周期中重要函数调用等相关知识点。
  - 接着又讨论了 AMS 先 pause 当前 Activity，然后再创建目标进程并运行 Activity 的流程。其中牵扯到两个应用进程和 AMS 的交互，其难度之大可见一斑。读者在阅读本节时，务必要区分此旅程中两个阶段工作的重点：其一是找到合适的 ActivityRecord 和 TaskRecord；其二是调度相关进程进行 Activity 切换。在 SDK 文档中，介绍最为详细的是第一阶段中系统的处理策略，例如启动模式、启动标志的作用等。第二阶段工作其实是与 Android 组件调度相关的工作。SDK 文档只是针对单个 Activity 进行生命周期方面的介绍。坦诚地说，这次旅程略过不少逻辑情况。原因有二，一方面受限于精力和篇幅，另方面是作为调度核心类，和 AMS 相关的代码及处理逻辑非常复杂，而且其间还夹杂了与 WMS 的交互逻辑，使复杂度更甚。再者，笔者个人感觉这部分代码绝谈不上高效、

严谨和美观，甚至有些丑陋（在分析它们的过程中，远没有研究 Audio、Surface 时那种畅快淋漓的感觉）。此处列出几个供读者深入研究的点：

- 各种启动模式、启动标志的处理流程。
- Configuration 发生变化时 Activity 的处理，以及在 Activity 中对状态保存及恢复的处理流程。
- Activity 生命周期各个阶段的转换及相关处理。Android 2.3 以后新增的与 Fragment 的生命周期相关的转换及处理。

8.4 3. 广播处理总结

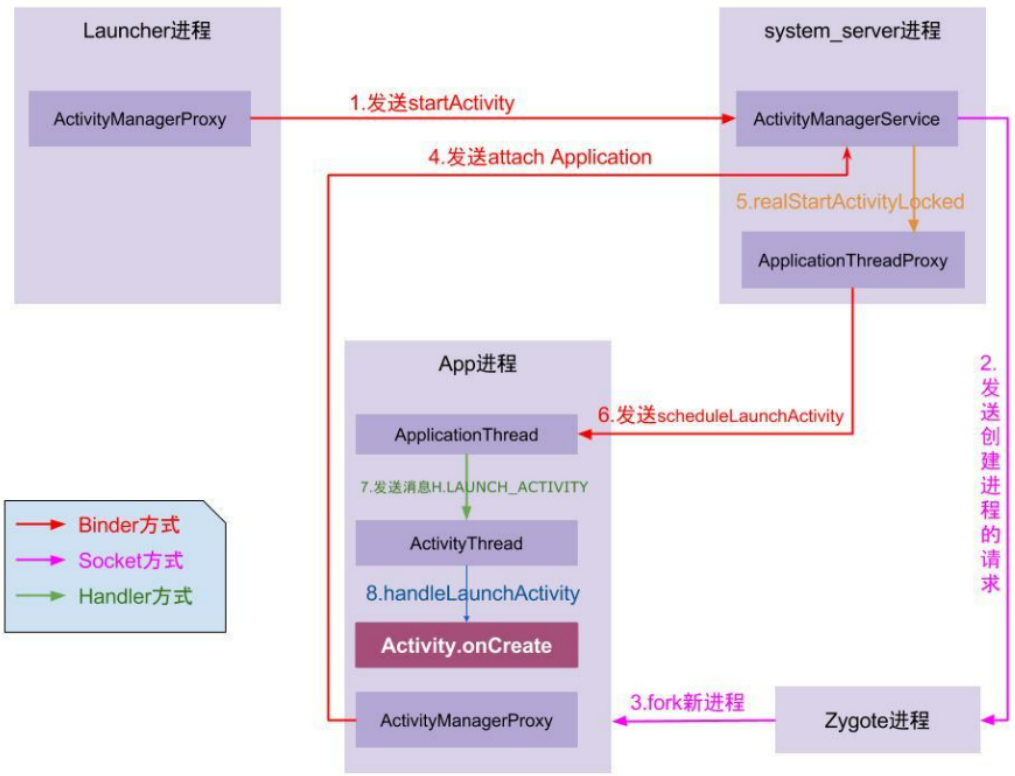


8.5 4. startService 流程图

总结 5.1 流程说明在整个 startService 过程，从进程角度看服务启动过程

- **Process A 进程**：是指调用 startService 命令所在的进程，也就是启动服务的发起端进程，比如点击桌面 App 图标，此处 Process A 便是 Launcher 所在进程。

- **system_server 进程**：系统进程，是 java framework 框架的核心载体，里面运行了大量的系统服务，比如这里提供 ApplicationThreadProxy（简称 ATP），ActivityManagerService（简称 AMS），这个两个服务都运行在 system_server 进程的不同线程中，由于 ATP 和 AMS 都是基于 IBinder 接口，都是 binder 线程，binder 线程的创建与销毁都是由 binder 驱动来决定的，每个进程 binder 线程个数的上限为 16。
- **Zygote 进程**：是由 init 进程孵化而来的，用于创建 Java 层进程的母体，所有的 Java 层进程都是由 Zygote 进程孵化而来；
- **Remote Service 进程**：远程服务所在进程，是由 Zygote 进程孵化而来的用于运行 Remote 服务的进程。主线程主要负责 Activity/Service 等组件的生命周期以及 UI 相关操作都运行在这个线程；另外，每个 App 进程中至少会有两个 binder 线程 ApplicationThread(简称 AT) 和 ActivityManagerProxy（简称 AMP），当然还有其他线程，这里不是重点就不提了。



- 图中涉及 3 种 IPC 通信方式：Binder、Socket 以及 Handler，在图中分别用 3 种不同的颜色来代表这 3 种通信方式。一般来说，同一进程内的线程间通信采用的是 Handler 消息队列机制，不同进程间的通信采用的是 binder 机制，另外与 Zygote 进程通信采用的 Socket。
- **启动流程：**
  - Process A 进程采用 Binder IPC 向 system_server 进程发起 startService 请求；
  - system_server 进程接收到请求后，向 zygote 进程发送创建进程的请求；
  - zygote 进程 fork 出新的子进程 Remote Service 进程；
  - Remote Service 进程，通过 Binder IPC 向 sytem_server 进程发起 attachApplication 请求；
  - system_server 进程在收到请求后，进行一系列准备工作后，再通过 binder IPC 向 remote Service 进程发送 scheduleCreateService 请求；

- Remote Service 进程的 binder 线程在收到请求后，通过 handler 向主线程发送 CREATE_SERVICE 消息；
- 主线程在收到 Message 后，通过发射机制创建目标 Service，并回调 Service.onCreate() 方法。到此，服务便正式启动完成。当创建的是本地服务或者服务所属进程已创建时，则无需经过上述步骤 2、3，直接创建服务即可。

8.6 5. AMS 中的进程管理

- 前面曾反复提到，Android 平台中很少能接触到进程的概念，取而代之的是有明确定义的四大组件。但是作为运行在 Linux 用户空间内的一个系统或框架，Android 不仅不能脱离进程，反而要大力利用 Linux OS 提供的进程管理机制和手段，更好地为自己服务。作为 Android 平台中组件运行管理的核心服务，ActivityManagerService 当仁不让地接手了这方面的工作。目前，AMS 对进程的管理仅涉及两个方面：
  - 调节进程的调度优先级和调度策略。
  - 调节进程的 OOM 值。

8.7 6. App 的 Crash 处理总结

- 应用进程进行 Crash 处理的流程。[[图片上传失败...(image-bd3406-1561648443575)]

8.7.1 一、概述

- ActivityManagerService 是 Framework 层的核心服务之一，ActivityManagerService 是 Binder 的子类，它的功能主要以下三点：
  - 四大组件的统一调度
  - 进程管理
  - 内存管理

8.7.2 二、ActivityManagerService 的启动过程

- ActivityManagerService 的启动是在 systemserver 进程的 startBootstrapServices 方法中启动的。
- SystemServiceManager.startService(ActivityManagerService.Lifecycle.class) 功能主要：创建 ActivityManagerService.Lifecycle 对象；调用 Lifecycle.onStart() 方法。

8.7.3 三、主要功能之一的四大组件的统一调度

- **ActivityManagerService 最主要的功能就是统一的管理者 activity，service，broadcast，provider 的创建，运行，关闭.** 我们在应用程序中启动 acitivity，关闭 acitiviy 等操作最终都是要通过 ams 来统一管理的. 这个过程非常的复杂，不是一下子可以讲的清楚的，我这里推荐老罗的博客来讲解四大组件的启动过程：
- Android 应用程序内部启动 Activity 过程（startActivity）的源代码分析 Android 系统在新进程中启动自定义服务过程（startService）的原理分析 Android 应用程序注册广播接收器（registerReceiver）的过程分析 Android 应用程序发送广播（sendBroadcast）的过程分析 Android 应用程序组件 Content Provider 简要介绍和学习计划

8.7.4 四、主要功能之一的内存管理

- 我们知道当一个进程中的 acitiviy 全部都关闭以后, 这个空进程并不会立即就被杀死，而是要等到系统内存不够时才会杀死。但是实际上 ActivityManagerService 并不能够管理内存，android 的内存管理是 Linux 内核中的内存管理模块和 OOM 进程一起管理的。
- Android 进程在运行的时候, 会通过 Ams 把每一个应用程序的 oom_adj 值告诉 OOM 进程, 这个值的范围在-16-15, 值越低说明越重要，越不会被杀死。当发生内存低的时候,Linux 内核内存管理模块会通知 OOm 进程根据 AMs 提供的优先级强制退出值较高的进程。因此 Ams 在内存管理中只是扮演着一个提供进程 oom_adj 值的功能. 真正的内存管理还是要调用 OOM 进程来完成，下面通过调用 Activity 的 finish() 方法来看看内存释放的情况。
- 当我们手动调用 finish() 方法或者按 back 键时都是会关闭 activity 的，在调用 finish 的时候只是会先调用 ams 的 finishActivityLocked 方法将当前要关闭的 acitiviy 的 finish 状态设置为 true，然后就会先去启动新的 acitiviy，当新的 acitiviy 启动完成以后就会通过消息机制通知 Ams，Ams 在调用 activityIdleInternalLocked 方法来关闭之前的 acitiviy。