Android ActivityManagerService 启动流程详解

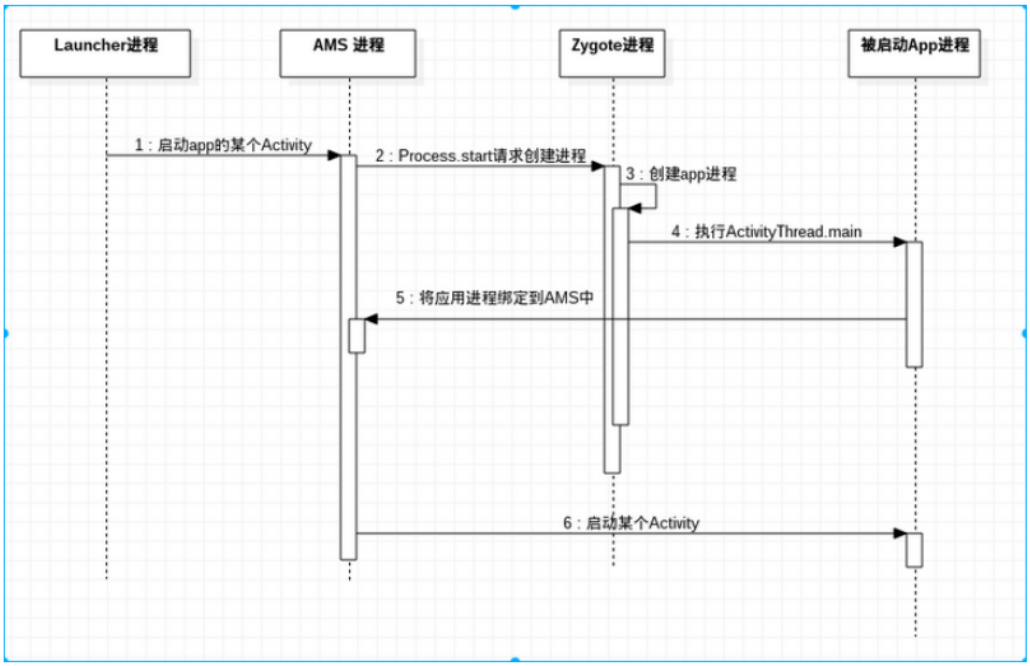
deepwaterooo

August 30, 2022

Contents

1 Android ActivityManagerService 启动流程详解

- 用户从 Launcher 程序点击应用图标可启动应用的入口 Activity，Activity 启动时需要多个进程之间的交互，如下图所示。



- 其中，AMS 进程实际上是 SystemServer 进程，因为 AMS 只是 SystemServer 启动的一个服务而已，运行在 SystemServer 的某个线程中。

- 具体的，用户在 Launcher 程序里点击应用图标时，会通知 ActivityManagerService 启动应用的主 Activity，ActivityManagerService 发现这个应用还未启动，则会通知 Zygote 进程执行 ActivityThread 的 main 方法。应用进程接下来通知 ActivityManagerService 应用进程已启动，ActivityManagerService 保存应用进程的一个代理对象，这样 ActivityManagerService 可以通过这个代理对象控制应用进程，然后 ActivityManagerService 通知应用进程创建主 Activity 的实例，并执行它的生命周期方法，也就是诸如 OnCreadte() 等方法。

1.1 Launcher 启动

- 当点击应用程序图标后，Launcher 使用一个带有 Intent.FLAG_ACTIVITY_NEW_TASK flag 的 Intent，调用 startActivity 方法来启动 App。相关源码如下：

```
public static Intent makeLaunchIntent(Context context, LauncherActivityInfoCompat info,
            UserHandleCompat user) {
        long serialNumber = UserManagerCompat.getInstance(context).getSerialNumberForUser(user);
        return new Intent(Intent.ACTION_MAIN)
            .addCategory(Intent.CATEGORY_LAUNCHER)
            .setComponent(info.getComponentName())
            .setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED)
            .putExtra(EXTRA_PROFILE, serialNumber);
    }
```

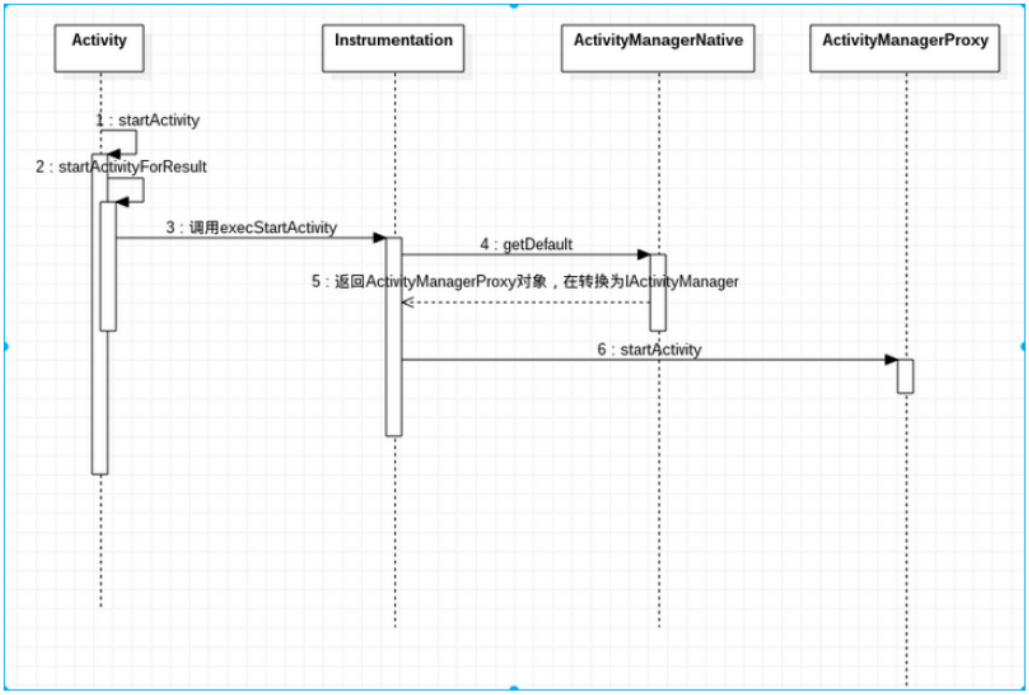
- 当点击 app 的图标时会执行如下的代码调用流程。

```
public void onClick(View v) {
    Object tag = v.getTag();
    if (tag instanceof ShortcutInfo)
        onClickAppShortcut(v);
}
protected void onClickAppShortcut(final View v) {
    // Start activities
    startAppShortcutOrInfoActivity(v);
}
void startAppShortcutOrInfoActivity(View v) {
    // 得到 launcher 提供的启动这个 app 主 activity 的 intent
    intent = shortcut.intent;
    boolean success = startActivitySafely(v, intent, tag);
}
boolean startActivitySafely(View v, Intent intent, Object tag) {
    success = startActivity(v, intent, tag);
}
private boolean startActivity(View v, Intent intent, Object tag) {
    intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    startActivity(intent, optsBundle);
}
```

- 从以上代码流程可知当 Launcher 启动一个 app 时，会在自己的 startActivity() 方法中为 Intent 中添加一个 FLAG_ACTIVITY_NEW_TASK flag，然后调用继承自 Activity 的 startActivity() 方法来进一步启动 app。

1.2 Activity 向 AMS 发起请求启动 App

- Activity 启动 Activity 的流程如下，具体可以查看相关的源码，需要注意的是 Android 6.0 的实现和 8.0 版本实现有略微的区别。



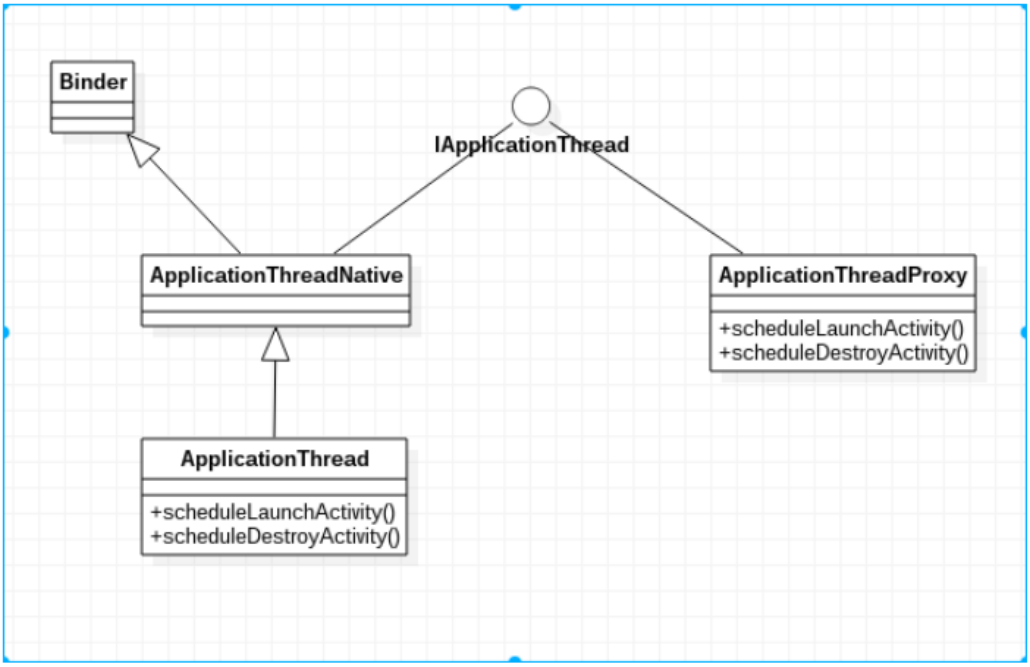
- 下面我们看一下 ActivityThread 类，ActivityThread 类是 Android 应用进程的核心类，这个类包含了应用框架中其他重要的类。其源码如下：

```
public final class ActivityThread {

    static IPackageManager sPackageManager;
    private ContextImpl mSystemContext;

// 保存该 app 中所有的 Activity
    final ArrayMap<IBinder, ActivityClientRecord> mActivities = new ArrayMap<>();
// 保存该 app 中所有的 service: 对当前应用中的所有的服务进行管理
    final ArrayMap<IBinder, Service> mServices = new ArrayMap<>();
// 保存该 app 中所有的 provider
    final ArrayMap<ProviderKey, ProviderClientRecord> mProviderMap
        = new ArrayMap<ProviderKey, ProviderClientRecord>();
// 管理应用的资源
    private final ResourcesManager mResourcesManager;

// 存储包含代码，即 dex 文件的 apk 文件保存在该变量中
    final ArrayMap<String, WeakReference<LoadedApk>> mPackages
        = new ArrayMap<String, WeakReference<LoadedApk>>();
// 不包含代码，紧紧包含资源的 apk 放在该变量中
    final ArrayMap<String, WeakReference<LoadedApk>> mResourcePackages

    // 如果 app 中自己实现了 Application 的子类，并在清单文件中声明了，那么该变量就指向自己实现的那个子类对象
    Application mInitialApplication;
    AppBindData mBoundApplication;

// 用于 binder 通信，AMS 通过它来调用应用的接口
    final ApplicationThread mAppThread = new ApplicationThread();

// 主线程中的 Handler
    static Handler sMainThreadHandler;  // set once in main()
    final Looper mLooper = Looper.myLooper();

// H 继承自 Handler,mH 用来发送和处理 ApplicationThread 通过 binder 接受的 AMS 请求
    final H mH = new H();
}
```

- ActivityThread 类中没有定义数据结构来存储 BroadcastReceiver 对象，因为 BroadcastReceiver 对象生命周期很短暂，属于调用一次运行一次的类型，因此不需要保存其对象。AppBindData 类为 ActivityThread 的内部类，定义如下，记录了与之绑定的 app 的相关数据。

```
static final class AppBindData {
    LoadedApk info;
    String processName;
    ApplicationInfo appInfo;
    List<ProviderInfo> providers;
    ComponentName instrumentationName;

    Bundle instrumentationArgs;
    IInstrumentationWatcher instrumentationWatcher;
    IUiAutomationConnection instrumentationUiAutomationConnection;

    int debugMode;
    boolean enableOpenGlTrace;
    boolean restrictedBackupMode;

    boolean persistent;
    Configuration config;
    CompatibilityInfo compatInfo;

    /** Initial values for {@link Profiler}. */
    ProfilerInfo initProfilerInfo;

    public String toString() {
        return "AppBindData{appInfo=" + appInfo + "}";
    }
}
```

- 其中 ApplicationThread 类型的变量 mAppThread 用于 AMS 所在 app 的接口，应用进程需要调用 AMS 提供的功能，而 AMS 也需要主动调用应用进程以控制应用进程并完成指定操作。ApplicationThread 的运作流程如下图：



- 如上图可知，AMS 通过 IApplicationThread 接口管理应用进程，ApplicationThread 类实现了 IApplicationThread 接口，实现了管理应用的操作，ApplicationThread 对象运行在应用进程里。ApplicationThreadProxy 对象是 ApplicationThread 对象在 AMS 线程 (AMS 线程运行在 system_server 进程) 内的代理对象，AMS 通过 ApplicationThreadProxy 对象调用 ApplicationThread 提供的功能，比如让应用进程启动某个 Activity。ApplicationThread 中的 scheduleDestroyActivity 的源码如下：

```
// ApplicationThread 中的 scheduleDestroyActivity 的源码如下
public final void scheduleDestroyActivity(IBinder token, boolean finishing, int configChanges) {
    sendMessage(H.DESTROY_ACTIVITY, token, finishing ? 1 : 0, configChanges);
}
```

- 而 Binder 服务端的最终调用的是 ActivityThread 的 sendMessage 函数。

```
private void sendMessage(int what, Object obj, int arg1, int arg2) {
    sendMessage(what, obj, arg1, arg2, false);
}
private void sendMessage(int what, Object obj, int arg1, int arg2, boolean async) {
    if (DEBUG_MESSAGES)
        Slog.v(TAG, "SCHEDULE " + what + " " + mH.codeToString(what) + ": " + arg1 + " / " + obj);
    Message msg = Message.obtain();
    msg.what = what;
    msg.obj = obj;
    msg.arg1 = arg1;
    msg.arg2 = arg2;
    if (async)
        msg.setAsynchronous(true);
    mH.sendMessage(msg);
}
```

- 而 ActivityThread 类中内部类 H（继承自 Handler，mH 就是 H 的对象）中则定义了处理消息的方法，该函数用来处理接收到的数据。

1.3 AMS 启动 Activity

- 前面讲到 AMS 使用 startActivity 启动 APP，为了加深印象在来看一下 startActivity 函数（需要注意的是，6.0 和 8.0 的代码有细微的区别）。

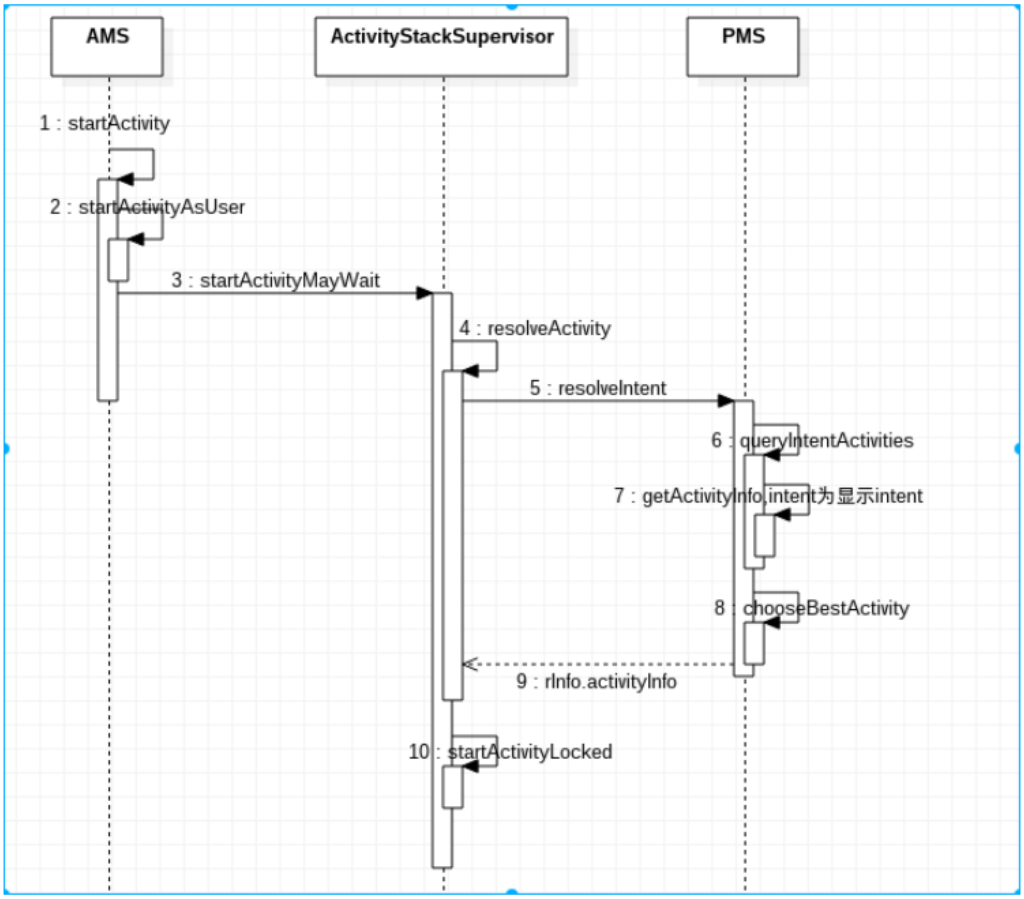
```
public final int startActivity(IApplicationThread caller, String callingPackage,
                               Intent intent, String resolvedType, IBinder resultTo, String resultWho, int requestCode,
                               int startFlags, ProfilerInfo profilerInfo, Bundle options) {
    return startActivityAsUser(caller, callingPackage, intent, resolvedType, resultTo,
                               resultWho, requestCode, startFlags, profilerInfo, options,
                               UserHandle.getCallingUserId());
}
public final int startActivityAsUser(IApplicationThread caller, String callingPackage,
                                     Intent intent, String resolvedType, IBinder resultTo, String resultWho, int requestCod
                                     int startFlags, ProfilerInfo profilerInfo, Bundle options, int userId) {
    // 如果是隔离的应用的话，不允许其打开其他 app 的 activity
    // appid 在 99000-99999 之间的属于隔离 app
    enforceNotIsolatedCaller("startActivity");
    userId = handleIncomingUser(Binder.getCallingPid(), Binder.getCallingUid(), userId,
                                false, ALLOW_FULL_ONLY, "startActivity", null);
    // TODO: Switch to user app stacks here. 转换到用户应用栈 (任务栈)？
    return mStackSupervisor.startActivityMayWait(caller, -1, callingPackage, intent,
                                                 resolvedType, null, null, resultTo, resultWho, requestCode, startFlags,
                                                 profilerInfo, null, null, options, false, userId, null, null);
}
```

- 判断发起者是否是隔离的 app，不允许隔离的 app 调用其他 app。然后调用 ActivityStackSupervisor 类中的 startActivityMayWait 方法。

```
final int startActivityMayWait(
    IApplicationThread caller,// AMS 通过这个参数可以和发起者进行交互
    int callingUid,           // 发起者 uid
    String callingPackage,    // 发起者包名
    Intent intent,       // 启动 activity 的 intent
    String resolvedType, // intent 的类型，也就是 MIME type
    IVoiceInteractionSession voiceSession,
    IVoiceInteractor voiceInteractor,
    IBinder resultTo, // 用于接收 startActivityForResult 的结果,launcher 启动 app 这种情景下没有用，为 null
    String resultWho,
    int requestCode,  // 这个是调用者来定义其意义，若值大于等于 0，则 AMS 内部保存该值并通过 onActivityResult 返回调用者，这里为-
    int startFlags,   // 传入的为 0
    ProfilerInfo profilerInfo,
    WaitResult outResult,
    Configuration config,
    Bundle options,
    boolean ignoreTargetSecurity,
    int userId,
    IActivityContainer iContainer, // 传入的为 null
    TaskRecord inTask) // 传入为 null
{
    // Refuse possible leaked file descriptors
    if (intent != null && intent.hasFileDescriptors())
```

```
            throw new IllegalArgumentException("File descriptors passed in Intent");
        // 当启动一个 app 时 , launcher 会构造一个 intent, 前面已经介绍了, 是一个显示的 intent
        // 所以这里为 true,
        boolean componentSpecified = intent.getComponent() != null;

        // Don't modify the client's object!
        // 创建一个新的 intent, 方便改动
        intent = new Intent(intent);

        // 收集 要启动的 app 的主 activity 的信息
        ActivityInfo aInfo = resolveActivity(intent, resolvedType, startFlags, profilerInfo, userId);

        // 传入的该参数为 null
        ActivityContainer container = (ActivityContainer)iContainer;
        synchronized (mService) { // mService: ArrayMap<IBinder, Service> 对应用中的所有服务进行管理
            if (container != null && container.mParentActivity != null &&
                container.mParentActivity.state != RESUMED) {
                // Cannot start a child activity if the parent is not resumed.
                return ActivityManager.START_CANCELED;
            }
            final ActivityStack stack; // 去定义活动所在的任务栈
            if (container == null || container.mStack.isOnHomeDisplay()) { // <<<<<<<<<< ActivityContainer.mStack
                stack = mFocusedStack;
            } else
                stack = container.mStack;

            // 传入的 config 为 null
            stack.mConfigWillChange = config != null && mService.mConfiguration.diff(config) != 0;
            if (DEBUG_CONFIGURATION)
                Slog.v(TAG_CONFIGURATION, "Starting activity when config will change = " + stack.mConfigWillChange);

            final long origId = Binder.clearCallingIdentity();
            if (aInfo != null &&
                (aInfo.applicationInfo.privateFlags
                 & ApplicationInfo.PRIVATE_FLAG_CANT_SAVE_STATE) != 0) { // <<<<<<<<<< 这里是干什么来着 ?
            }

            int res = startActivityLocked(caller, intent, resolvedType, aInfo,
                                          voiceSession, voiceInteractor, resultTo, resultWho,
                                          requestCode, callingPid, callingUid, callingPackage,
                                          realCallingPid, realCallingUid, startFlags, options, ignoreTargetSecurity,
                                          componentSpecified, null, container, inTask);
            Binder.restoreCallingIdentity(origId);
            if (stack.mConfigWillChange) {
                // 应该是针对配置要求对任务栈中的活动进行一定的处理吧
            }
            // 传入的为 null
            if (outResult != null)
                mService.wait(); //等待应用进程的 activity 启动完成
        }
        return res;
    }
```

- startActivityAsUser() 方法最主要的目地是进行权限检查，检查发起者是否被隔离，是的话，是不允许调用别的 app 的 activity 的。startActivityMayWait() 方法主要是利用传入的 intent 去向 PMS 搜集要启动的 APP 的信息，储存到 aInfo 中.。名字中有 wait 字眼，预示着该方法可能导致线程等待，不过在我们这个场景中不会出现这种情况，因为 wait 出现在对结果的处理中, 我们这个场景中是不需要处理结果的。



1.4 ActivityThread.main

- Android APP 的入口类在 ActivityThread 中，有一个 Main 函数，该函数的源码如下：

```
public static void main(String[] args) {
    Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "ActivityThreadMain");
    SamplingProfilerIntegration.start();
    CloseGuard.setEnabled(false);

    // 环境初始化，主要是 app 运行过程中 需要使用到的系统路径，比如外部存储路径等等
    Environment.initForCurrentUser();

    // Set the reporter for event logging in libcore
    EventLogger.setReporter(new EventLoggingReporter());
    // 增加一个保存 key 的 provider
    AndroidKeyStoreProvider.install();

    // 为应用社会当前用户的 CA 证书保存的位置
    final File configDir = Environment.getUserConfigDirectory(UserHandle.myUserId());
    TrustedCertificateStore.setDefaultUserDirectory(configDir);

    // 设置 app 进程的名字
    // 通过前面的分析可知，前面的过程中已经设置过名字了，这里又改为了 “pre-initialized” ，不知道为啥，
    // 因为后面还要再调用该方法，重新设置进程名字为 app 包名或者 app 指定的名字。
    Process.setArgV0("<pre-initialized>"); // 现在还是没有初始化的无名进程

    // 创建主线程 looper
    Looper.prepareMainLooper();
    // 创建 ActivityThread 对象。
    ActivityThread thread = new ActivityThread();
    // 将创建的 ActivityThread 附加到 AMS 中，这样 AMS 就可以控制这个 app 中组件的生命周期了
    thread.attach(false); // 是什么时候 attached 呢？后来。。。
    if (sMainThreadHandler == null) // sMainThreadHandler: ActivityThread 应用主线程的 Handler
```

```
        sMainThreadHandler = thread.getHandler();
    if (false) // 这句话：是在发疯吗？
        Looper.myLooper().setMessageLogging(new LogPrinter(Log.DEBUG, "ActivityThread"));

    // End of event ActivityThreadMain.
    Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);

    // App 主线程开始执行消息处理循环
    Looper.loop();
    throw new RuntimeException("Main thread loop unexpectedly exited");
}
```

- 当 ActivityThread 对象创建之后，就开始调用其 attach() 方法，这是一个很重要的方法，参数为 false 表明是普通 app 进程。

```
private void attach(boolean system) { // system: 是系统进程吗？应用进程不是!
    sCurrentActivityThread = this; // 当前线程，也即是主线程
    mSystemThread = system;
    // app 进程传入 fasle
    if (!system) {
        ViewRootImpl.addFirstDrawHandler(new Runnable() {
                @Override public void run() {
                    ensureJitEnabled();
                }
            });
        android.ddm.DdmHandleAppName.setAppName("<pre-initialized>", UserHandle.myUserId()); // 无名氏。。。。应用
        // mAppThread 是 ApplicationThread 对象；
        // 下面这个方法会把 mAppThread 放到 RuntimeInit 类中的静态变量 mApplicationObject 中
        RuntimeInit.setApplicationObject(mAppThread.asBinder()); // 应用线程 IBinder
            // 上面，RuntimeInit.setApplicationObject 方法源码如下：
            public static final void setApplicationObject(IBinder app) {
                mApplicationObject = app;
            }

        // 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。Pag
        // 跨进程交互：下面这句就相当于是拿到了 AMS 的句柄，可以调用其方法
        final IActivityManager mgr = ActivityManagerNative.getDefault(); // is ActivityManagerNative static class ?
        try {
            // 执行 AMS 的 attachApplication 方法：将 mAppThread 传入 AMS，这样 AMS 就可以通过它来控制 app 了
            mgr.attachApplication(mAppThread);
        } catch (RemoteException ex) {}

        // Watch for getting close to heap limit.
        BinderInternal.addGcWatcher(new Runnable() {});
    } // 。。。

    // add dropbox logging to libcore
    DropBox.setReporter(new DropBoxReporter());

    ViewRootImpl.addConfigCallback(new ComponentCallbacks2() {});
}
```

1.5 AMS 的 attachApplication 方法

- attachApplication 方法主要负责 APP 与 AMS 的绑定操作，该方法的源码如下：

```
public final void attachApplication(IApplicationThread thread) { // AMS
    synchronized (this) {
        int callingPid = Binder.getCallingPid(); // Binder: 远程服务端的 IBinder 管理器静态类
        final long origId = Binder.clearCallingIdentity();
        attachApplicationLocked(thread, callingPid); // <<<<<<<<<<
        Binder.restoreCallingIdentity(origId);
    }
}
```

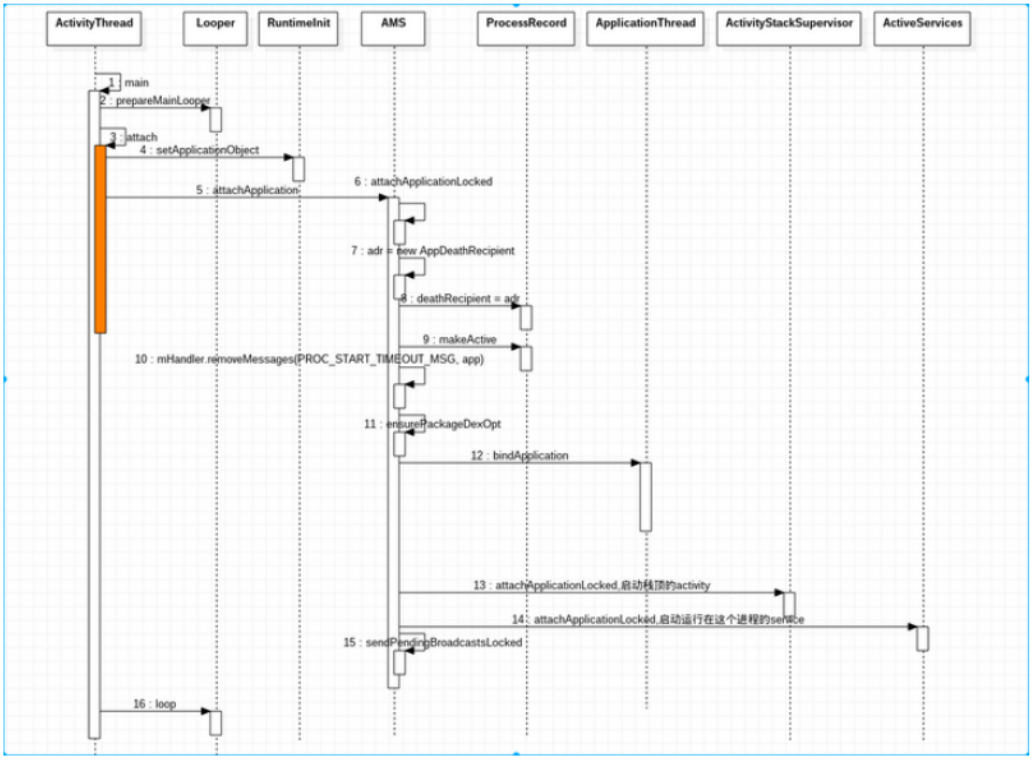
- 该方法最终调用了 attachApplicationLocked() 方法。

```
private final boolean attachApplicationLocked(IApplicationThread thread, int pid) {
    ProcessRecord app;
    if (pid != MY_PID && pid >= 0) {
        synchronized (mPidsSelfLocked) { // 多线程安全，数据结构 上锁
            // 在创建 startProcessLocked() 方法中调用 Process.start() 方法创建进程后
            // 会以接收传递过来的进程号为索引，将 ProcessRecord 加入到 AMS 的 mPidsSelfLocked 中
            // 这里可以以进程号从 mPidsSelfLocked 中拿到 ProcessRecord
```

```
            app = mPidsSelfLocked.get(pid);
        }
    } else
        app = null;
    if (app == null) return false;
    if (app.thread != null) // <<<<<<<<<<<<<<<<<<< 不知道 ProcessRecord.thread: 指的是什么线程？
        handleAppDiedLocked(app, true, true); // 什么意思呢？

    // 注册 app 进程死亡通知处理机制，也就是创建监听 app 死亡的对象
    // App 进程死亡后，会调用 AppDeathRecipient.binderDied() 方法
    // 这不是同跨进程远程服务绑定的，当远程服务死掉的时候客户端收到通知一样吗，很像呀，有不同吗？死亡代理类的定义不同
    final String processName = app.processName;
    try { // 跨进程服务绑定，捕获异常么
        AppDeathRecipient adr = new AppDeathRecipient(app, pid, thread);
        thread.asBinder().linkToDeath(adr, 0); // AMS 注册应用层当前应用 app 死亡的监听回调. thread ? // <<<<<<<<<<
        app.deathRecipient = adr; // 双向通知；应用也说，如果我死了，一定通知你。。。
    } catch (RemoteException e) {
        app.resetPackageList(mProcessStats);
        startProcessLocked(app, "link fail", processName);
        return false;
    }

    // 调用 ProcessStatsService 开始记录 process 的状态
    // 该方法中将 thread 赋值给 app.thread
    app.makeActive(thread, mProcessStats);
    // 初始化 App 进程优先级等信息
    app.curAdj = app.setAdj = -100;
    app.curSchedGroup = app.setSchedGroup = Process.THREAD_GROUP_DEFAULT;
    app.forcingToForeground = null;
    updateProcessForegroundLocked(app, false, false);
    app.hasShownUi = false;
    app.debugging = false;
    app.cached = false;
    app.killedByAm = false;

    // 移除 PROC_START_TIMEOUT_MSG 消息
    // 前面在 AMS.startProcessLocked 方法中会在调用 Process.start() 方法之后，将这个消息放入消息队列中
    // 如果没有在规定的时间内将该消息移除消息队列，那么会导致进程启动超时
    mHandler.removeMessages(PROC_START_TIMEOUT_MSG, app); // AMS 启动应用的超时机制

    // mProcessesReady 为 true
    boolean normalMode = mProcessesReady || isAllowedWhileBooting(app.info);
    // 拿到 App 的 provider
    List<ProviderInfo> providers = normalMode ? generateApplicationProvidersLocked(app) : null;
        // If the app is being launched for restore or full backup, set it up specially
    boolean isRestrictedBackupMode = false;
    if (mBackupTarget != null && mBackupAppName.equals(processName)) {
        isRestrictedBackupMode = (mBackupTarget.backupMode == BackupRecord.RESTORE)
            || (mBackupTarget.backupMode == BackupRecord.RESTORE_FULL)
            || (mBackupTarget.backupMode == BackupRecord.BACKUP_FULL);
    }

    // 判断是否需要执行 dex2oat 命令
    // 在 app 安装的时候，会执行一次 dex2oat
    // 当生成的 oat 文件被破外或者删除的时候，需要重新执行 dex2oat
    ensurePackageDexOpt(app.instrumentationInfo != null
                        ? app.instrumentationInfo.packageName
                        : app.info.packageName);
    // instrument app 技术先关
    // 比如 Android studio 开发时，修改某些代码时，没必要重新安装 apk，即可查看之后的结果
    // 后续单独在分析 instrument 技术
    if (app.instrumentationClass != null)
        ensurePackageDexOpt(app.instrumentationClass.getPackageName());

    // 调用 ApplicationThread 的 bindApplication 接口方法去启动 Application
    thread.bindApplication(processName, appInfo, providers, app.instrumentationClass,
                           profilerInfo, app.instrumentationArguments, app.instrumentationWatcher,
                           app.instrumentationUiAutomationConnection, testMode, enableOpenGlTrace,
                           isRestrictedBackupMode || !normalMode, app.persistent,
                           new Configuration(mConfiguration), app.compat,
                           getCommonServicesLocked(app.isolated),
                           mCoreSettingsObserver.getCoreSettingsLocked());
    updateLruProcessLocked(app, false, null);
    app.lastRequestedGc = app.lastLowMemory = SystemClock.uptimeMillis();
// 这段代码很花，不知道哪里来的
// } catch (Exception e) {
//     return false;
// }
```

```
    boolean badApp = false;
    boolean didSomething = false;
// See if the top visible activity is waiting to run in this process...
// 为 true
    if (normalMode) {
        try {
            // 执行 ActivityStackSupervisor.attachApplicationLocked() 方法去启动 ActivityStack 栈顶的 Activity
            if (mStackSupervisor.attachApplicationLocked(app)) { // <<<<<<<<<<
                didSomething = true;
            }
        } catch (Exception e) {
            Slog.wtf(TAG, "Exception thrown launching activities in " + app, e);
            badApp = true;
        }
    }
// Find any services that should be running in this process...
    if (!badApp) {
        try {
            // 处理要运行这个进程中的 service: ActiveServices 调用的 attachApplicationLocked() 方法启动在当前 App 进程中的 serv
            didSomething |= mServices.attachApplicationLocked(app, processName); // <<<<<<<<<<
        } catch (Exception e) {
            Slog.wtf(TAG, "Exception thrown starting services in " + app, e);
            badApp = true;
        }
    }
// Check if a next-broadcast receiver is in this process...
    if (!badApp && isPendingBroadcastProcessLocked(pid)) {
        try {
            // 处理广播: 检查是否有广播 broadcast 到这个 application, 如果有则广播
            didSomething |= sendPendingBroadcastsLocked(app); // <<<<<<<<<<<<<<<<<<<< ???
        } catch (Exception e) {
            // If the app died trying to launch the receiver we declare it 'bad'
            Slog.wtf(TAG, "Exception thrown dispatching broadcasts in " + app, e);
            badApp = true;
        }
    }
    if (!didSomething)
        updateOomAdjLocked();
    return true;
}
```

- attachApplicationLocked 函数比较长，首先以传入的 app 进程号为索引从 AMS 的 mPidsSelfLocked 中取出 app 进程的 ProcessRecord 对象。然后调用 ProcessRecord 对象的 makeActive 方法调用 ProcessStatsService 开始记录 process 的状态,接着将 PROC_START_TIMEOUT_M 消息, 从消息循环中移除, 检查是否重新执行 dex2oat 生成 app 的 oat 文件。
- 该方法主要做了一下四件事情：
  - 调用 ActivityThread 的 bindApplication 方法去启动 Application；
  - 是调用 ActivityStackSupervisor 的 attachApplicationLocked() 方法去启动 ActivityStack 栈顶的 Activity；
  - 是 ActiveServices 调用的 attachApplicationLocked() 方法启动在当前 App 进程中的 service；
  - 是检查是否有广播 broadcast 到这个 application，如果有则广播。



1.6 ApplicationThread.bindApplication 方法

- 接下来重点分析下 bindApplication() 方法，这个方法最终效果是调用了 App 的 Application 对象的 onCreate 方法。其源码如下：

```
public final void bindApplication(
    String processName, // ProcessRecord 中记录的进程名字
    ApplicationInfo appInfo,
    List<ProviderInfo> providers, // app 中的 providers
    ComponentName instrumentationName,
    ProfilerInfo profilerInfo,
    Bundle instrumentationArgs, // 测试相关
    IInstrumentationWatcher instrumentationWatcher,
    IUiAutomationConnection instrumentationUiConnection,
    int debugMode,
    boolean enableOpenGlTrace, boolean isRestrictedBackupMode, boolean persistent,
    Configuration config, CompatibilityInfo compatInfo, Map<String, IBinder> services,
    Bundle coreSettings) {
    if (services != null)
        // Setup the service cache in the ServiceManager
        ServiceManager.initServiceCache(services);

    // 发送 SET_CORE_SETTINGS 消息
    // 获取系统的设定并设置到 ActivityThread 中
    setCoreSettings(coreSettings);

    // 拿到 PMS
    IPackageManager pm = getPackageManager();
    android.content.pm.PackageInfo pi = null;
    try {
        // 以包名从 PMS 中获得 PackageInfo
        pi = pm.getPackageInfo(appInfo.packageName, 0, UserHandle.myUserId());
    } catch (RemoteException e) {}
    if (pi != null) {
        // 该 app 是否设置了共享 uid
        boolean sharedUserIdSet = (pi.sharedUserId != null);

        // app 进程名字是否被设定为与包名不一致
        // 默认情况下,app 进程名字就是其包名
```

```
        // 当显示设置 process name 的时候可以执行进程的名字
        boolean processNameNotDefault =
            (pi.applicationInfo != null &&
             !appInfo.packageName.equals(pi.applicationInfo.processName));

        // 如果设置了共享 uid 或者进程名字设置为了其他名字,
        // 这就导致该 app 可能运行在一个已经运行的进程中
        boolean sharable = (sharedUserIdSet || processNameNotDefault);

        // 如果 app 是单独的进程, 那么要向 VM 注册相关信息
        // 是在/data/dalvik-cache/profiles/创建一个以包名为名字的空文件, 另外两个参数没用到
        if (!sharable)
            VMRuntime.registerAppInfo(appInfo.packageName, appInfo.dataDir, appInfo.processName);
    }
    // 创建兵初始化 AppBindData 对象
    // 在这里设置了进程名字,app 的 provider,ApplicationInfo
    AppBindData data = new AppBindData();
    data.processName = processName;
    data.appInfo = appInfo;
    data.providers = providers;
    // 测试相关
    data.instrumentationName = instrumentationName;
    data.instrumentationArgs = instrumentationArgs;
    data.instrumentationWatcher = instrumentationWatcher;
    data.instrumentationUiAutomationConnection = instrumentationUiConnection;
    data.debugMode = debugMode;
    data.enableOpenGlTrace = enableOpenGlTrace;
    // 是否允许 adb backup
    data.restrictedBackupMode = isRestrictedBackupMode;
    // 进程是否常驻内存, 杀掉后, 会被重启
    data.persistent = persistent;
    data.config = config;
    data.compatInfo = compatInfo;
    data.initProfilerInfo = profilerInfo;
    // 发送 BIND_APPLICATION 消息
    sendMessage(H.BIND_APPLICATION, data); // <<<<<<<<<< H: Handler
}
```

- bindApplication() 方法要通过 PMS 检查启动的 app 是否设置了共享 uid，以及检查当前 app 进程的名字是否设定的与包名不一致，符合两者中的任一种情况下，则说明该 app 进程可能运行在另一个已经存在的进程中。

- bindApplication() 方法主要是创建和初始化了 AppBindData 对象，并发送两个消息：一个是 SET_CORE_SETTINGS；另一个是 BIND_APPLICATION。SET_CORE_SETTINGS 主要是获取系统的设定并设置到 ActivityThread 中。BIND_APPLICATION 用于启动 App 并安装所有的 provider，并回调 App 的 oncreate 方法 BIND_APPLICATION 消息。

- ActivityThread 中处理 BIND_APPLICATION 消息的方法是 handleBindApplication()，其源码如下：

```
private void handleBindApplication(AppBindData data) {
    mBoundApplication = data;

    // 设置进程的名字, 因为前面 ActivityThread.main 将其设置为了"<pre-initialized>"
    Process.setArgV0(data.processName); // 终于不再是无名氏了。。。
    // 设置 app 在 ddms 中显示的进程名字
    android.ddm.DdmHandleAppName.setAppName(data.processName, UserHandle.myUserId());

    // 普通 app 进程, 一般情况下为 false
    // 除非 xml 设置 persistent 为 true
    // 带有 persistent 标记的进程在低内存设备中部支持使用硬件加速
    if (data.persistent) {
        if (!ActivityManager.isHighEndGfx())
            HardwareRenderer.disable(false);
    }
    if (mProfiler.profileFd != null)
        mProfiler.startProfiling();

    // 根据 app 编译时指定的 sdk 版本与当前系统 sdk 版本设置 AsyncTask
    if (data.appInfo.targetSdkVersion <= android.os.Build.VERSION_CODES.HONEYCOMB_MR1)
        AsyncTask.setDefaultExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
    Message.updateCheckRecycle(data.appInfo.targetSdkVersion);

    // 恢复时区和位置信息
```

```
TimeZone.setDefault(null);
Locale.setDefault(data.config.locale);
// 资源管理初始化设置
mResourcesManager.applyConfigurationToResourcesLocked(data.config, data.compatInfo);
mCurDefaultDisplayDpi = data.config.densityDpi;
applyCompatConfiguration(mCurDefaultDisplayDpi);

// 设置 AppBindData 中 LoadedApk info 属性字段
// 这里会根据传入 app 的 ActivityInfo 和 CompatibilityInfo 创建一个 LoadedApk 对象
data.info = getPackageInfoNoCheck(data.appInfo, data.compatInfo); // <<<<<<<<<<  设置属性字段

// 如果应用没有指定使用设备的 density, 那么默认使用 mdpi
if ((data.appInfo.flags&ApplicationInfo.FLAG_SUPPORTS_SCREEN_DENSITIES) == 0) {
    mDensityCompatMode = true;
    Bitmap.setDefaultDensity(DisplayMetrics.DENSITY_DEFAULT);
}
updateDefaultDensity();

// 创建 ContextImpl 上下文, 里面也设计到了资源管理相关的内容 , 如从 LoadedApk 中提取资源
// 后续还需对其进行初始化
final ContextImpl appContext = ContextImpl.createAppContext(this, data.info); // <<<<<<<<<<
// 普通 app 启动时,isIsolated 为 false
if (!Process.isIsolated()) {
    // 在沙箱目录中创建 cache 文件夹
    final File cacheDir = appContext.getCacheDir();
    if (cacheDir != null) {
        // 将创建的 cache 文件夹与属性"java.io.tmpdir" 关联
        System.setProperty("java.io.tmpdir", cacheDir.getAbsolutePath());
    } else
        Log.v(TAG, "Unable to initialize \"java.io.tmpdir\" property due to missing cache directory");

    // Use codeCacheDir to store generated/compiled graphics code
    // 在沙箱目录创建 code-cache 文件夹
    final File codeCacheDir = appContext.getCodeCacheDir();
    if (codeCacheDir != null) {
        setupGraphicsSupport(data.info, codeCacheDir);
    } else
        Log.e(TAG, "Unable to setupGraphicsSupport due to missing code-cache directory");
}
// 设置时间格式
final boolean is24Hr = "24".equals(mCoreSettings.getString(Settings.System.TIME_12_24));
DateFormat.set24HourTimePref(is24Hr);
View.mDebugViewAttributes =
    mCoreSettings.getInt(Settings.Global.DEBUG_VIEW_ATTRIBUTES, 0) != 0;
// 调试相关
if ((data.appInfo.flags &
     (ApplicationInfo.FLAG_SYSTEM | ApplicationInfo.FLAG_UPDATED_SYSTEM_APP)) != 0)
    StrictMode.conditionallyEnableDebugLogging();
if (data.appInfo.targetSdkVersion > 9)
    StrictMode.enableDeathOnNetwork();

NetworkSecurityPolicy.getInstance().setCleartextTrafficPermitted(
    (data.appInfo.flags & ApplicationInfo.FLAG_USES_CLEARTEXT_TRAFFIC) != 0);
if (data.debugMode != IApplicationThread.DEBUG_OFF) {}

// Enable OpenGL tracing if required
if (data.enableOpenGlTrace)
    GLUtils.setTracingLevel(1);

// Allow application-generated systrace messages if we're debuggable.
boolean appTracingAllowed = (data.appInfo.flags&ApplicationInfo.FLAG_DEBUGGABLE) != 0;
Trace.setAppTracingAllowed(appTracingAllowed);

/**
 * Initialize the default http proxy in this process for the reasons we set the time zone.
 */
IBinder b = ServiceManager.getService(Context.CONNECTIVITY_SERVICE);
if (b != null) {
    IConnectivityManager service = IConnectivityManager.Stub.asInterface(b);
    try {
        // 设置网络代理
        final ProxyInfo proxyInfo = service.getProxyForNetwork(null);
        Proxy.setHttpProxySystemProperty(proxyInfo); // 设置网络代理
    } catch (RemoteException e) {}
}
// 为 null
if (data.instrumentationName != null) {
} else {
    // 创建 Instrumentation 对象
```

```
            mInstrumentation = new Instrumentation();
        }
        if ((data.appInfo.flags & ApplicationInfo.FLAG_LARGE_HEAP) != 0) {
            dalvik.system.VMRuntime.getRuntime().clearGrowthLimit();
        } else
            dalvik.system.VMRuntime.getRuntime().clampGrowthLimit();
        final StrictMode.ThreadPolicy savedPolicy = StrictMode.allowThreadDiskWrites();
        try {
            // 创建 app 的 Application 对象
            Application app = data.info.makeApplication(data.restrictedBackupMode, null);
            mInitialApplication = app;

            // don't bring up providers in restricted mode; they may depend on the app's custom Application class
            if (!data.restrictedBackupMode) {
                List<ProviderInfo> providers = data.providers;
                if (providers != null) {
                    installContentProviders(app, providers);
                    // For process that contains content providers, we want to
                    // ensure that the JIT is enabled "at some point".
                    mH.sendEmptyMessageDelayed(H.ENABLE_JIT, 10*1000);
                }
            }

            // Do this after providers, since instrumentation tests generally start their
            // test thread at this point, and we don't want that racing.
            try {
                // 执行 instrumentation 的 onCreate() 方法
                mInstrumentation.onCreate(data.instrumentationArgs);
            } catch (Exception e) {}
            // 执行 Application 的 onCreate 生命周期方法
            try {
                mInstrumentation.callApplicationOnCreate(app);
            } catch (Exception e) {}
        } finally {
            StrictMode.setThreadPolicy(savedPolicy);
        }
    }
```

- handleBindApplication 函数主要完成了如下的一些操作:
  - 确定了进程的最终名字, 以及其在 ddms 中显示的进程名字;
  - 恢复进程的时区和位置信息;
  - 调用 getPackageInfoNoCheck() 创建 LoadApk 对象;
  - 创建 ContextImpl 对象, 是 AppContext;
  - 设置网络代理;
  - 创建 Instrumentation 对象。

1.7 LoadedApk

- LoadedApk 类用来记录描述一个被加载运行的 APK, 的代码、资源等信息。

```
public final class LoadedApk {
    private static final String TAG = "LoadedApk";
    private final ActivityThread mActivityThread; // App 的 ActivityThread 对象
    private ApplicationInfo mApplicationInfo;   // 描述 App 信息的 ApplicationInfo, 如果 App 中重载了 Application 类, 那么其类
    final String mPackageName;// app 的包名

    private final String mAppDir;// app 在/data/app/< 包名 > 路径
    private final String mResDir;// 资源路径
    private final String[] mSplitAppDirs;
    private final String[] mSplitResDirs;
    private final String[] mOverlayDirs;
    private final String[] mSharedLibraries;// 共享 java 库
    private final String mDataDir;//数据沙箱目录
    private final String mLibDir;// native so 库位置
    private final File mDataDirFile;

    private final ClassLoader mBaseClassLoader;//getPackageInfoNoCheck() 创建的 LoadedApk 对象中该字段初始化为 null
    private final boolean mSecurityViolation;
    private final boolean mIncludeCode;// 这个 apk 是否包含 dex
    private final boolean mRegisterPackage;
```

```
    private final DisplayAdjustments mDisplayAdjustments = new DisplayAdjustments();
    Resources mResources;
    private ClassLoader mClassLoader;//
    private Application mApplication;// 这个 app 的 Application 对象，如果 App 继承了 Application，那么为其子类对象

// ReceiverDispatcher
    private final ArrayMap<Context, ArrayMap<BroadcastReceiver, ReceiverDispatcher>> mReceivers
        = new ArrayMap<Context, ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>>();
// LoadedApk.ReceiverDispatcher
    private final ArrayMap<Context, ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>> mUnregisteredReceivers
        = new ArrayMap<Context, ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>>();
// LoadedApk.ServiceDispatcher 两个 !
    private final ArrayMap<Context, ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>> mServices
        = new ArrayMap<Context, ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>>();
    private final ArrayMap<Context, ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>> mUnboundServices
        = new ArrayMap<Context, ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>>();

    int mClientCount = 0;
    Application getApplication() {
        return mApplication;
    }
}
```

- 通过分析可知，在 handleBindApplication() 方法中通过调用 getPackageInfoNoCheck() 方法创建 LoadedApk 对象。getPackageInfoNoCheck() 的源码如下：

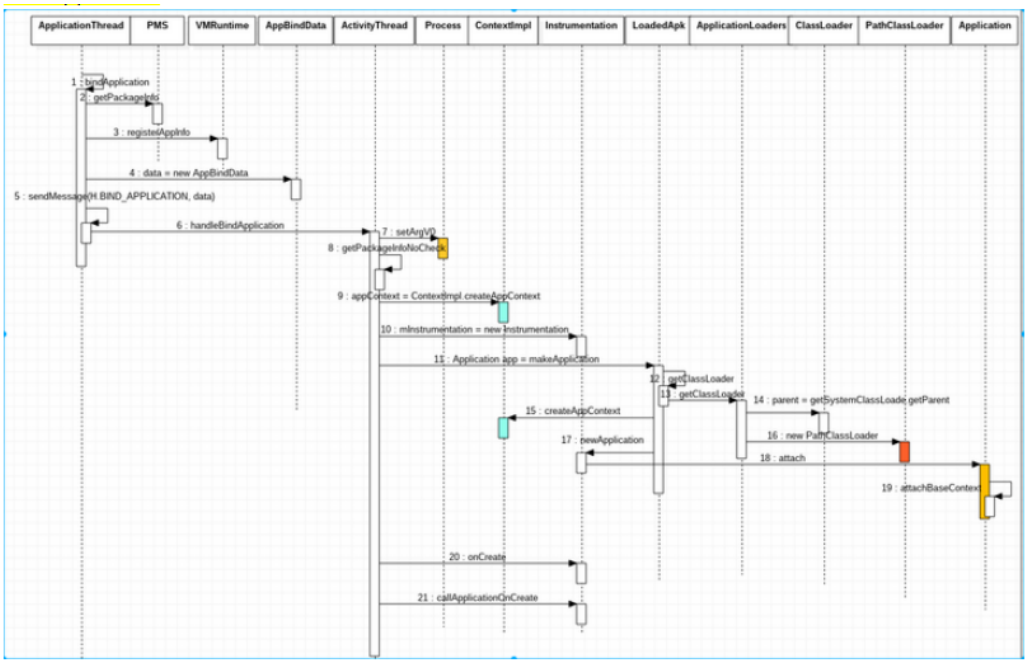
```
public final LoadedApk getPackageInfoNoCheck(ApplicationInfo ai, CompatibilityInfo compatInfo) {
    return getPackageInfo(ai, compatInfo, null, false, true, false);
}
// getPackageInfoNoCheck() 又调用了 getPackageInfo()。
private LoadedApk getPackageInfo(
    ApplicationInfo aInfo, // app 的 Application 信息
    CompatibilityInfo compatInfo, // 兼容性
    ClassLoader baseLoader,// 传入 null
    boolean securityViolation,// 传入 false
    boolean includeCode,// 传入 true
    boolean registerPackage // 传入 false
) {
// 要启动 app 的拥有者与当前系统用户不一致
    final boolean differentUser = (UserHandle.myUserId() != UserHandle.getUserId(aInfo.uid));
    synchronized (mResourcesManager) {
        WeakReference<LoadedApk> ref; // <<<<<<<<<<<<<<<<<<<<
        if (differentUser) {
            ref = null;
        } else if (includeCode) {
            // 如果包含了 dex，那么从 ActivityThread.mPackages 中先查找是否已经有了 apk 对应的 LoadedApk
            ref = mPackages.get(aInfo.packageName);
        } else {
            // 如果没有包含了 dex，那么从 ActivityThread.mResourcePackages 中先查找是否已经有了 apk 对应的 LoadedApk
            ref = mResourcePackages.get(aInfo.packageName);
        }
        // 如果前面已经从 mPackages 或者 mResourcePackages 中找到了 apk 对应的 LoadedApk，那么就可以直接返回了
        // 没有找到的话，就要创建 LoadedApk 对象了
        if (packageInfo == null ||
            (packageInfo.mResources != null && !packageInfo.mResources.getAssets().isUpToDate())) {
            // 创建 LoadedApk 对象
            packageInfo = new LoadedApk(this, aInfo, compatInfo, baseLoader,
                            securityViolation, includeCode &&
                            (aInfo.flags&ApplicationInfo.FLAG_HAS_CODE) != 0, registerPackage);
            if (mSystemThread && "android".equals(aInfo.packageName))
                packageInfo.installSystemApplicationInfo(aInfo, getSystemContext().mPackageInfo.getClassLoader());

            // 创建 LoadedApk 对象之后，将其加入对应的缓存列表中
            if (differentUser) {
                // Caching not supported across users
            } else if (includeCode) {
                mPackages.put(aInfo.packageName, new WeakReference<LoadedApk>(packageInfo));
            } else {
                mResourcePackages.put(aInfo.packageName, new WeakReference<LoadedApk>(packageInfo));
            }
        }
        return packageInfo;
    }
}
```

- 由以上代码可知，当要获取一个 LoadedApk 对象时，先从 ActivityThread 的两个缓存列表：mPackages 和 mResourcePackages 中寻找，没找到的话才会新建 LoadedApk 对象，然

后将其加入对应的缓存列表中。当找到 apk 对应的 LoadedApk 对象后，以此为参数创建 Application 的 Context 对象。

```
final ContextImpl appContext = ContextImpl.createAppContext(this, data.info);
static ContextImpl createAppContext(ActivityThread mainThread, LoadedApk packageInfo) {
    if (packageInfo == null) throw new IllegalArgumentException("packageInfo");
    return new ContextImpl(null, mainThread,
                           packageInfo, null, null, false, null, null, Display.INVALID_DISPLAY);
}
private ContextImpl(
    ContextImpl container, // 传入 null
    ActivityThread mainThread,// app 的 ActivityThread 对象
    LoadedApk packageInfo, // apk 对应的 LoadedApk 对象
    IBinder activityToken, // 传入为 null
    UserHandle user, boolean restricted,
    Display display, Configuration overrideConfiguration, int createDisplayWithId) {
    mOuterContext = this;

    mMainThread = mainThread;
    mActivityToken = activityToken;
    mRestricted = restricted;

    if (user == null)
        user = Process.myUserHandle();
    mUser = user;
    // context 中会记录 apk 对应的 LoadedApk 对象
    mPackageInfo = packageInfo;
    // 资源管理相关, 后续单独开篇介绍
    mResourcesManager = ResourcesManager.getInstance();

    Resources resources = packageInfo.getResources(mainThread);
    if (resources != null) {
        if (displayId != Display.DEFAULT_DISPLAY
            || overrideConfiguration != null
            || (compatInfo != null && compatInfo.applicationScale
                != resources.getCompatibilityInfo().applicationScale)) {
            resources = mResourcesManager.getTopLevelResources(packageInfo.getResDir(),
                                                               packageInfo.getSplitResDirs(),
                                                               packageInfo.getOverlayDirs(),
                                                               packageInfo.getApplicationInfo().sharedLibraryFiles,
                                                               displayId, overrideConfiguration, compatInfo);
        }
    }
    mResources = resources;

    if (container != null) {
        mBasePackageName = container.mBasePackageName;
        mOpPackageName = container.mOpPackageName;
    } else {
        // 记录 app 包名
        mBasePackageName = packageInfo.mPackageName;
        ApplicationInfo ainfo = packageInfo.getApplicationInfo();
        if (ainfo.uid == Process.SYSTEM_UID && ainfo.uid != Process.myUid()) {
            mOpPackageName = ActivityThread.currentPackageName();
        } else
            mOpPackageName = mBasePackageName;
    }
    // 内容提供者相关
    mContentResolver = new ApplicationContentResolver(this, mainThread, user);
}
```

- bindApplication() 方法关键时序图如下：



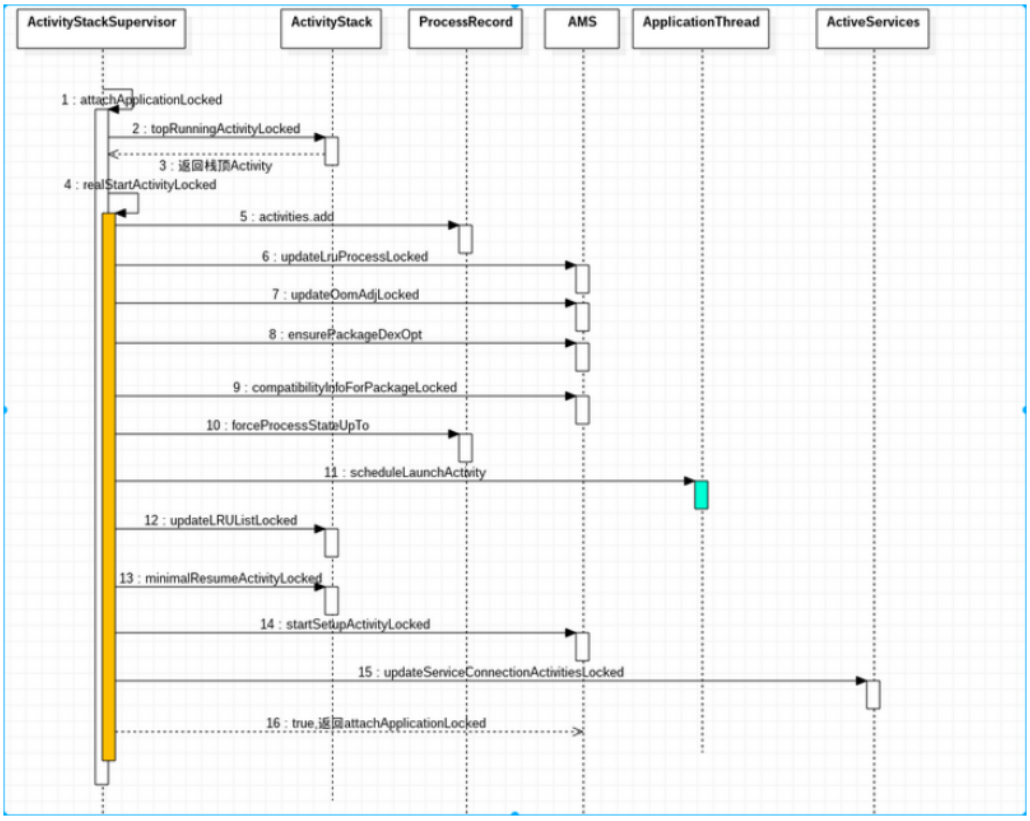
- 在这个方法中创建了 Classloader, 以及 Application 对象。然后执行 Application 对象的 attach 方法，这个方法中又会调用 attachBaseContext() 方法。也就是说 Application 对象首先被执行的方法不是 onCreate() 方法，而是 attach() 方法。

1.8 attachApplicationLocked

- 由 ActivityThread.main 的整体执行时序图中可知，启动 activity 的最终是 attachApplicationLocked() 方法。

```
boolean attachApplicationLocked(ProcessRecord app) throws RemoteException {
    final String processName = app.processName;
    boolean didSomething = false;
    for (int displayNdx = mActivityDisplays.size() - 1; displayNdx >= 0; --displayNdx) {
        ArrayList<ActivityStack> stacks = mActivityDisplays.valueAt(displayNdx).mStacks;
        for (int stackNdx = stacks.size() - 1; stackNdx >= 0; --stackNdx) {
            final ActivityStack stack = stacks.get(stackNdx);
            // 从 如何启动 app 中篇之 Task 的管理 可知, 此时 mFocusedStack 指向即将要运行的 activity 所在的 ActivityStack
            // 下面这个方法就是为了从众多 ActivityStack 找到这个 ActivityStack
            if (!isFrontStack(stack)) continue;
            // 找到了所需的 ActivityStack
            // 然后找到其栈顶的 Activity, 实际就是 mTaskHistory 数组末端的 Task 的顶端 Activity
            ActivityRecord hr = stack.topRunningActivityLocked(null);
            if (hr != null) {
                if (hr.app == null && app.uid == hr.info.applicationInfo.uid
                        && processName.equals(hr.processName)) {
                    try {
                        if (realStartActivityLocked(hr, app, true, true))
                            didSomething = true;
                    } catch (RemoteException e) {
                        Slog.w(TAG, "Exception in new application when starting activity "
                                + hr.intent.getComponent().flattenToShortString(), e);
                        throw e;
                    }
                }
            }
        }
    }
    if (!didSomething)
        ensureActivitiesVisibleLocked(null, 0);
    return didSomething;
}
```

- ActivityStackSupervisor 的流程调用关系可以用下面的流程图表示。


ActivityStackSupervisor
ActivityStack
ProcessRecord
AMS
ApplicationThread
ActiveServices
1 : attachApplicationLocked
2 : topRunningActivityLocked
3 : 返回栈顶Activity
4 : realStartActivityLocked
5 : activities.add
6 : updateLruProcessLocked
7 : updateOomAdjLocked
8 : ensurePackageDexOpt
9 : compatibilityInfoForPackageLocked
10 : forceProcessStateUpTo
11 : scheduleLaunchActivity
12 : updateLRUListLocked
13 : minimalResumeActivityLocked
14 : startSetupActivityLocked
15 : updateServiceConnectionActivitiesLocked
16 : true,返回attachApplicationLocked