目 录

第一篇 组件体系

第 1 章 核心组件

1.1 Activity

1.1.1 Activity 的启动

为了不局限在调用逻辑和代码细节，尝试更直观地，或者在更高的层次来理解 Activity 的启动，从下面几个角度来看 Activity 的启动。

1、主要调用逻辑

在调用 Context 的 startActivity 方法时，系统会访问 Context 的实例对象，例如 Activity 的 startActivity 方法，下面就从 Activity.startActivity 调用开始，列出 Activity 启动过程涉及到的一系列调用逻辑，部分做了少量简化。

```
Activity.startActivity
    -> Instrumentation.execStartActivity
        -> ActivityManagerService.startActivity
            -> ActivityStarter.startActivityMayWait*
                ->ActivityStackSupervisor.resumeFocusedStackTopActivityLocked
                    -> ActivityStack.resumeTopActivityUncheckedLocked
                -> ActivityStackSupervisor.startSpecificActivityLocked
        -> ActivityManagerService.startProcessLocked
                -> ActivityStackSupervisor.realStartActivityLocked
                    -> ActivityThread.handleLaunchActivity
```

-> Activity.performCreate

之后，Activity 的 onCreate 函数会被回调，进而执行具体 Activity 实例对象的回调。这里的 ActivityStackSupervisor、ActivityStack 属于 Activity 任务栈管理模型的成员，Instrumentation 则有利于自动化测试的设计， 执行到 Activity.performCreate 时，即将创建目标 Activity，并执行 onCreate 以及后续的 onStart、onResume 等生命周期回调。

2、进程切换角度

在启动 Activity 时，通常会涉及到进程的切换 根 Activity（应用启动时的主 Activity）往往涉及到应用程序进程的创建或切换，多次跨进程的调用。

例如，我们从 Launcher 点击 App 图标新启动一个应用时，系统级的进程切换往往是这样的：

Launcher 进程 -> system_server 进程 -> Zygote 进程 -> 应用程序进程

如果是启动一个已经启动过的 Activity（对应热启动方式），则会涉及到两个进程，也就是 Activity 所在的应用程序进程，和系统进程（system_process）进程。

再看前面的调用流程，大部分是在系统进程中完成的，例如 ActivityManagerService.startActivity、ActivityStarter.startActivityMayWait 和 ActivityStackSupervisor.resumeFocusedStackTopActivityLocked 等，部分在应用进程完成，例如 ActivityThread.handleLaunchActivity。

和 Zygote 进程有关的调用是 ActivityManagerService.startProcessLocked，因为该调用会涉及到目标 Activity 所在应用进程的创建，在 Android 虚拟机层面，只有 Zygote 进程才有权限 fork 一个新的进程。并且，Android 利用这套进程创建逻辑，基于 Linux 的父子

进程资源共享机制，实现了一定程度上的资源复用，避免了重复的初始化工作。

3、生命周期回调

分析 Activity 的生命周期回调机制，最好结合特定的场景。

场景 1，新启动某 Activity（冷启动方式），然后退出，该 Activity 的生命周期回调执行顺序依次为：

onCreate->onStart->onResume->onPause->onStop->onDestroy

场景 2，新启动某 Activity（冷启动方式），然后旋转屏幕，在默认配置下也就是不设置 android:configChanges、screenOrientation 属性时，该 Activity 的生命周期回调执行顺序依次为：

onCreate->onStart->onResume->

（切换横屏操作）

->onPause->onSaveInstance->onStop->onDestroy

->onCreate->onStart->onRestoreInstance

如果设置了 android:configChanges 属性（例如设为 orientation|screenSize），则不会重新创建该 Activity，而是回调 onConfigurationChanged 函数：

onCreate->onStart->onResume->

（切换横屏操作）

->onConfigurationChanged

这样可以避免不必要的创建工作，因为 Activity 的创建如果没有优化，有时候会带来不好的用户体验，例如短暂白屏。

其他还有若干场景，例如启动一个后台 Activity（热启动）使其变为前台 Activity，再回到

后台，生命周期回调流程如下：

onNewIntent->onStart->onResume->onPause->onStop

这里 onNewIntent 的回调还和 Activity 启动模式有关。

1.1.2 启动模式及使用场景

启动模式有四种：

1、standard

标准模式，系统默认的启动模式。在启动 Activity 时，系统总是创建一个新的 Activity 实例。其缺点是：复用性差、占用内存，当 Activity 已经在栈顶时，还是会创建实例。

2、singleTop

这种模式可以实现栈顶复用的效果。

如果目标 Activity 已经存在于栈顶，则调用实例的 onNewIntent，否则创建一个新的实例，可以用于通知启动的内容显示，例如新闻客户端的内容页面。信息类应用的列表 Activity 往往也会使用这种启动模式，例如，收到新短信时，如果信息列表 Activity 已经在栈顶了，则直接调用其 onNewIntent，而不需要创建新的 Activity。

该模式下，当 Activity 不在栈顶时，还是会创建新的实例。例如 A、B、C 三个 Activity 处于一个任务栈中，且 C 在栈顶，如果 C 启动 A，则会创建一个新的实例并置于栈顶，使得当前任务栈有两个 A 实例。

3、singleTask

该模式实现栈内复用的效果。如果栈内已经有目标 Activity 实例，则清除目标 Activity 上面的所有 Activity、使得目标 Activity 直接放到栈顶，并调用目标 Activity 的 onNewIntent 方法；否则创建一个新的 Activity 实例。

补充，该模式 Activity 会在跟它有相同 taskAffinity 的任务中启动，并且位于这个任务的堆栈顶端，这样如果我们没有设置 taskAffinity，从本应用启动的 Activity 就会复用当前 Task，也就不会创建一个新的 Task。

该启动模式，适合作为程序入口点，例如原生 Launcher 以及相机应用中的 CameraActivity，就是使用了这种启动模式。

4、singleInstance

系统级的复用。启动 Activity 时，如果系统中不存在该 Activity 实例，则创建一个新的 ActivityTask，并且独占这个 Task，否则显示已有的 Activity 实例。通常用于系统级的应用，在整个系统中只有一个实例，例如来电提醒、闹钟提醒。

这四种启动模式创建一个新 Activity 实例的几率，是依次降低的。实际在使用时，往往会用到动态设置启动模式，给 Intent 设置特定的 Flag，例如 FLAG_ACTIVITY_SINGLE_TOP（对应 singleTop）、FLAG_ACTIVITY_NEW_TASK（对应 singleTask）。

1.1.3 注意事项

首先要注意 Flag 的动态设置方式，优先级高于静态方式（通过 AndroidManifest.xml 设置）。如果这两种设置方式混合在一起，则 Activity 启动过程、最终效果要分为多种情况来考虑。

其次，从非 Activity 组件例如 Service 启动 Activity 时，必须添加 FLAG_ACTIVITY_NEW_TASK 这个 Flag。可以思考下，系统这样设计的原因?

再次，由于部分启动模式在启动 Activity 时，有时候并不会重新创建一个实例，也就不会调用其 onCreate 方法，而是回调 onNewIntent 方法，如果在启动时需要根据 Intent 重新加载数据，注意在 onNewIntent 中添加对应的处理。由于 Activity 在某些情况下，可能会被系统销毁，为了保险，Intent 获取数据的处理，最好在 onCreate、onNewIntent 这两个函数中都做添加。

最后，如果我们用了 FLAG_ACTIVITY_CLEAR_TOP 这个 flag 启动 Activity，当栈内已有目标 Activity 实例时，系统的处理会分为两种情况：

a. 只设置了 FLAG_ACTIVITY_CLEAR_TOP，系统会销毁目标 Activity 和它之上的所有 Activity，重新创建目标 Activity，并不调用其 onNewIntent 方法。

b. 设置了 FLAG_ACTIVITY_CLEAR_TOP+FLAG_ACTIVITY_SINGLE_TOP，系统只销毁之上的所有 Activity，复用目标 Activity，调用 onNewIntent 方法，效果相当于设置 FLAG_ACTIVITY_NEW_TASK，也就是 singleTask 模式。

补充，适合使用 singleInstance 启动模式的 Activity，基本都具有高优先级、业务逻辑的交互相对简单，也就是与其他应用的交互场景不多。而 singleTask 模式更适用于那些具有一定频度的交互场景的 Activity，例如图库、联系人、一些第三方 App 问问需要启动相机拍照，然后再返回，这时候如果使用 singleInstance 则会始终独占一个任务栈，体验

不好。

1.2 Service

1.2.1 调用方式

1、启动服务

只启动一个服务，不进行通信，包括 startService、startForegroundService 两种调用方式。第二种方式适用于后台应用启动前台服务，在启动后的 10s 内（具体时间由 ActiveServices.SERVICE_START_FOREGROUND_TIMEOUT 定义），需要 Service 调用 startForeground 启动一个 Notification，不然会出现 ANR。

整个启动流程基于 ActivityManagerProxy、ActiveServices、AMS 以及 ActivityThread 完成，两种调用方式对应的流程主要是 ContextImpl.startServiceCommon 方法的 requireForeground 参数不同。

停止服务使用 stopService 方法，服务被停止的时候，系统自动回调 onDestory，注意服务只会被停止一次。

调用 startService 之后，Service 组件的生命周期：onCreate -> onStartCommand-> onDestory。如果服务已经启动，startService 方法不会重复执行 onCreate，而是执行 onStartCommand（该函数会调用 onStart，保持兼容）。

2、绑定服务

具体通过 bindService 调用完成，其特点是可以与服务端通信，调用服务里面的方法（本地或远程）。绑定服务的过程，通过 ContextImpl、LoadedApk、ActivityThread、AMS 以及 ActiveServices 完成。

绑定过程中，系统会通过 ActivityThread.handleBindService 方法回调 Service 组件的 onBind 方法，而 Service 组件在这里需要返回一个 IBinder 实例对象给客户端调用。如果是对跨进程服务的绑定，客户端在 onServiceConnected 回调获得的 IBinder 类型的 service 入参是 BinderProxy 实例，如果是同一进程，则 service 是个 Binder 实例。

不再需要服务时，调用者必须通过 unbindService 方法解除绑定，避免 ServiceConnection 对象导致的内存泄漏。调用者被销毁时，Service 也会退出。如果是多

个 Activity 绑定一个 Service，则在最后绑定的 Activity 销毁之后，onUnbind 才会被调用。

生命周期：onCreate -> onBind-> onUnbind->onDestory。绑定服务不会调用 onStartCommand 方法，如果服务已经绑定，bindService 方法不会重复执行 onBind。

3、启动+绑定

特点是，可以保证服务长期后台运行，又可以调用服务里面的方法，也能和服务之间传递数据。

具体可以先 startService 也可以先 bindService，生命周期顺序有差异。在停止服务时，需要同时调用 stopService 与 unbindService 方法。

如果先执行 stopService，则 unbindService 方法会使得系统依次回调 onUnbind 和 onDestroy 方法；反之，如果先执行 unbindService，则 unbindService 方法只会调用 onUnbind，然后在 stopService 时，回调 onDestroy 方法。

这种方式启动的服务生命周期分为两种：

- 先 start 后 bind：onCreate -> onStartCommand -> onBind。
- 先 bind 后 start：onCreate -> onBind -> onStartCommand。

1.2.2 常见服务分类

1、本地服务

依附在主进程上而不是独立的进程，不需要 IPC，也不需要 AIDL。如果是支持绑定的本地服务，只需实现 onBind 并返回一个实现 IBinder 接口的对象(通常是 Binder 派生类)。

2、远程服务

使用独立的进程，对应进程名格式为所在包名加上指定的 android:process 字符串。调用者所在进程被杀，该服务依然在运行。

一般需要使用 AIDL 进行 IPC，主要步骤包括：

1）客户端（调用端）定义远程服务对应的 aidl 文件。

2）在服务端的 Service 中定义一个 IxxService.Stub 实例对象，扩展实现具体业务逻辑的方法（类名 IxxService 及其接口必须和 aidl 文件中的定义一致）。

3）服务端的 Service 中的 onBind 方法，返回上述 Stub 实例对象。

4）客户端创建一个 ServiceConnection 对象，并重写 onServiceConnected 和 onServiceDisconnected 方法。在 onServiceConnected 方法中通过 IxxService.Stub.asInterface 获取 Proxy 代理对象（IxxService 类型）。

5）客户端通过 bindService 启动并绑定远程服务，具体需要传入 ServiceConnection 对象，通过该实例完成绑定，触发回调。

6）客户端可以通过代理对象调用远程服务的各项功能；在销毁时调用 unbindService 解除绑定。

注意，如果 aidl 文件中需要访问自定义类型例如 MyData（必须实现了 parcelable 接口），可以新增一个 aidl 文件，声明自定义类，parcelable MyData；并在调用该类的 aidl 文件中，导入这个类。

思考：如何实现客户端和服务端 AIDL 双向通信?

首先，定义两个 aidl 文件，例如 IMyInterface、IMyListener，分别用于提供客户端调用、给服务端回调（这两个 aidl 文件会同步应用到客户端和服务端）。

其次，在 IMyListener.aidl 中，定义一个回调接口 IMyListener，及回调方法 onCallBack(); 在 IMyInterface.aidl 中定义接口 IMyInterface 以及 registerCallback(IMyListener iListener)方法。

最后，在客户端，创建一个实现 IMyListener 接口的对象 IMyListener.Stub，再通过 aidl 调用服务端的 registerCallback 方法、传入 IMyListener 对象；这样在服务端就可以通过其代理（实现了 IMyListener 接口的对象，实际上是 IMyListener.Stub.Proxy），执行回调。

这里关键是通过一个 Listener 对象，对应扩展 aidl 与注册方法，由客户端注册并传入该对象。在服务端通过这个 Listener 的代理执行回调接口、实现反向通信。对于这类的 Listener，有时候需要使用 RemoteCallbackList 来辅助管理。

3、系统服务

一种特殊的系统级的服务，例如 AMS、WMS 等，这些服务不属于 Service 组件的范畴，而是由 ServiceManager 统一管理、启动。这些系统服务也实现了 AIDL 通信机制，例如 AMS，直接派生于 IActivityManager.Stub。

用户访问这些服务时，同样需要 aidl 文件，需要先通过 ServiceManager.getService 方法，获取到对应的 IBinder 实例，再利用 Stub 的 asInterface 方法获取 Proxy 对象，进

而调用具体的功能实现。

每个系统服务有个不同的 name，通过 ServiceManager 的 getService 获取 IBinder 对象时用到。

在系统服务内部，往往会有个内部类 Lifecycle(派生于 SystemService)，提供给调用者（SystemServer）启动服务。实际上只是调用了 Lifecycle.onStart 方法，之后的具体实现流程各有不同。

1.3 ContentProvider

1.3.1 数据访问机制

客户端/调用者通过 getContentResolver 调用，由 ActivityThread、AMS 获取到 ContentProvider 的代理，再通过这个代理对象调用服务端的实现（也即派生类中的自定义方法）。在自定义的 ContentProvider 中，对数据库的操作是通过 SQLiteDatabase 类完成的。

1.3.2 多线程并发

如果 ContentProvider 的数据存储方式是使用一个 SQLite 数据库，则不需要同步保护，因为 SQLite 内部实现好了线程同步，若是使用了多个 SQLite 数据库则需要同步保护，因为 SQL 对象之间无法进行线程同步。

如果 ContentProvider 的数据存储方式是内存，则需要自己实现线程同步，例如添加 synchronized 关键字。

1.3.3 批量处理与性能优化

如果只是批量增加数据，可以使用 bulkInsert 方法。这里需要注意该方法最终是通过循环调用 ContentProvider.insert 方法，实际上调用了用户自定义的 ContentProvider 派生类中重写的 insert，由于该方法会在短时间内被循环调用。

在重写 bulkInsert 方法时，注意有无 notifyChange 或者其它可能影响性能的操作。

Android 原生代码中，MediaProvider 的做法是，专门定义了一个 insertInternal 用于内部使用，该方法不会调用 notifyChange，而是由调用方例如 bulkInsert 在批处理全部

完成后再通知。

对于批量删除/更新也就是 UPDATE、DELETE 操作，可以使用 ContentProviderOperation 结合 ContentProvider.applyBatch 方法。

1.3.4 注意事项

ContentProvider 的 onCreate 函数中，不要做耗时操作，因为其生命周期函数是在主线程的调用的。

数据量很小的时候，例如 1MB 以下，不建议直接使用系统的增删改查函数，因为该组件的跨进程数据传输是基于 mmap 的匿名共享内存机制，这种情况下可以调用其 call 方法，降低开销。

注意访问权限的限制与合法性的校验，合理使用 setPathPermissions 方法。

另外，如果 ContentProvider 所在进程被杀，使用该 ContentProvider 的进程也会被杀，这部分的处理是在 ActivityManagerService 中执行的。

1.4 BroadcastReceiver

1.4.1 常见分类

BroadCastReceiver，按注册方式可以分为静态广播接收器和动态广播接收器。

静态广播接收器：不受程序是否启动的约束，当应用程序关闭之后，还是可以接收到广播（一般广播接收器的生命周期是和当前活动的生命周期保持同步）。

动态广播接收器：可以自由的控制注册和取消，有很大的灵活性。但是只能在程序启动之后才能收到广播。

对于广播（Broadcast），则可以分为普通广播和有序广播。

其中普通广播不在意顺序，各进程的广播接收器基本上可以同时收到这个广播。而有序广播，系统会根据接收者声明的优先级按顺序逐个接收处理，先收到有序广播的接收器，可以对该有序广播进行修改或者截断。

静态注册的广播接收器，在接收广播时，系统自动按有序广播的方式来串行处理（原因是进程的创建不能并发），此类接收器收到广播的先后顺序，和接收器所在 package 名称有关，或者说，和 PMS 扫描顺序有关。

动态注册的广播接收器，如果接收普通广播，接收器收到广播的顺序则和注册顺序有关。在所有普通广播里面，动态注册的广播接收器，相对于静态注册的广播接收器，会优先收到普通广播。

同优先级的动态有序广播，注册顺序影响广播的接收顺序；同优先级的静态有序广播，扫描顺序影响其接收顺序。

除了前面提到的这些，还有一种相对不太常用的：LocalBroadcastManager 方式注册的应用内广播接收器，只能通过 LocalBroadcastManager 动态注册。

1.4.2 注册过程简述

动态注册的广播接收器，主要通过 ContextImpl、LoadedApk，再调用 AMS 的 registerReceiver 方法完成。

静态注册的广播接收器，是在系统启动时，由 PMS 解析 apk 文件并记录 receivers 信息，然后 AMS 调用 PMS 的接口来查询 intent 匹配信息，再完成广播注册过程。

1.4.3 发送和接收过程简述

发送广播，主要通过 ContextImpl、LoadedApk、AMS、ActivityThread 完成。接收广播时，先是在 AMS 里面处理，找到接收者然后加到一个队列，再向对应的线程发送广播消息。

从 Android O 开始，系统对静态注册的广播接收器添加了限制，必须指定广播接收器所在包名才可以发送，或者使用 Intent.FLAG_RECEIVER_INCLUDE_BACKGROUND 这个 flag，但该标志是隐藏的，如果必须使用静态广播接收器、又不能指定包名（例如发送一对多的广播），则可以使用该标志的具体数值 0x01000000。

1.4.4 安全性

可以使用权限提升广播接收的安全性。如果接收器定义了权限，发送的广播需要对应声明权限才能发送；反之一样，不然收不到。

整体上，有以下几种情况：

- 指定发送方，发送者有权限才能发此广播；
- 指定接收方，接收者声明权限才能收到该广播；

- 同时指定发送方和接收方（很少使用）。

无论是哪种情况，都涉及到自定义权限，所以都需要在 AndroidManifest.xml 中声明对应的权限。

对于第一种情况（接收器定义权限来限制发送方），如果是动态广播接收器，可以在注册时通过 registerReceiver 传入该权限。静态广播接收器则是在 AndroidManifest.xml 里面，定义广播接收器的地方，添加 android:permission 属性，将该属性设为自定义权限。

1.4.5 onReceive()中的 Context 入参

这里的入参为 context 变量，其实例的具体类型，可以分为下面几种情况。

1、静态注册的广播接收器

这种情况下，入参是 android.app.ReceiverRestrictedContext 类型，不能用来启动 Activity、弹出 AlertDialog（除了系统应用且 Dialog 类型是 SYSTEM_ALERT_WINDOW 类型）。

2、动态注册的广播接收器

具体又分两种情况：

1）在 Activity 里面注册广播接收器。此时 onReceive 的入参 context 就是注册广播接收器的 Activty 对象。

由于此时入参 contex 为 Activity 的 Context 对象，可用于启动 Activity、弹出 AlertDialog。

但考虑到 onReceive()方法在主线程中，该方法需要在 10 秒内执行完毕，生命周期很短。如果弹出的对话框需要等待用户响应，就需要考虑对话框的管理问题。常用的做法是，将其放在 Service 里面管理，在 Service 启动的时候注册一个动态广播接收器，Service 停止的时候注销之。

2）在 Service 里面注册广播接收器。此时 onReceive 的入参 context 对应该 Service 对象，和 Service 是否跨进程无关。这种情况下的 Context 不能启动 Activity，有若干限制。

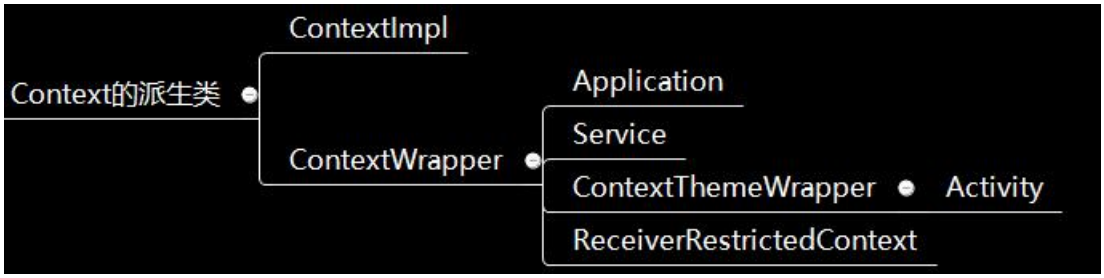
这两种情况可以统一为，onReceive 的入参 context 就是调用 registerReceiver 的组件的 Context。

其他还有 LocalBroadcastManager 注册的应用内广播接收器。其 onReceive 中的 context 是 Application 的 Context。

1.5 Context 浅析

1.5.1 Context 派生类

其派生类包括 ContextImpl、ContextThemeWrapper、Activity、Service、Application、ContextWrapper，派生关系如图所示。



1.5.2 功能与场景细分

绝大多数场景下，Activity、Service 和 Application 这三种类型的 Context 都是可以通用的，但有几种特殊场景除外。

Activity 的 Context 可以启动 Activity，使用标准模式或者其它模式都可以，但是 Service 启动 Activity 必须设置一个 flag：Intent.FLAG_ACTIVITY_NEW_TASK，对应 singleTask 模式。

弹出 Dialog 需要的 Context 实例，通常是 Activity 的 Context。一个特例是 TYPE_SYSTEM_ALERT 类型的 AlertDialog，可以不需要 Activity 的 Context，但需要在 AndroidManifest.xml 文件中声明 android.permission.SYSTEM_ALERT_WINDOW 权限（sharedUserId 也有要求）。

ReceiverRestrictedContext 用于静态广播接收器的注册，在对应的 onReceive 函数中传入实例，但这种 Context 不能 bindService，也不能注册广播接收器。这种 Context 只能通过 getApplicationContext 获取 Application 实例，而 Activity 和 Service 的 Context 可以调用 getApplication 或者 getApplicationContext 返回 Application 对象（这两个函数返回结果相同但语义不同）。

注意，在 Application 的构造函数中调用 Context 的方法就会崩溃，应该在 onCreate 方法中调用。另外不要自己实例化 Application 及其子类。ContentProvider 能使用 getContext().getApplicationContext()获取所在 Application，绝大多数情况下没有问题，

但是有可能会出现空指针的问题（app 由 ContentProvider 调起时）。

1.6 问题与思考

1.6.1 类型和依赖关系

按照功能类型来划分，这些核心组件大致可以分为前台活动、后台服务、内容/数据提供，分别对应 Activity、Service、BroadcastReceiver 和 ContentProvider，而 Context 的功能更像一种纽带或桥梁。

其中 Activity 通过 Context 启动或绑定 Service、注册 BroadcastReceiver、访问 ContentProvider。Service 通过 Context 启动 Activity、启动或绑定 Service、注册 BroadcastReceiver、访问 ContentProvider。BroadcastReceiver 通过 Context 启动 Activity、启动或绑定 Service、访问 ContentProvider。

从 Service 启动 Activity，会调用到 ContextImpl 类中的 startActivity 方法，而从 Activity 启动 Activity，会调用 Activity 类中的 startActivity 方法，前者会判断启动标志，对应启动模式必须包含 FLAG_ACTIVITY_NEW_TASK，否则抛出异常。这样的设计是结合 Activity 任务栈模型，影响多个 Activity 跳转时的返回层次效果，改善用户体验。

1.6.2 关于生命周期的对比

对比生命周期，可以发现 Activity 和 Service 都有 onCreate、onDestroy，其中 Activity 有 onResume、onPause，而 Service 没有。Service 有 onBind、onUnbind，Activity 没有。

BroadcastReceiver 只有 onReceive，ContentProvider 只有 onCreate，这两个组件在生命周期方面，与其他组件差异较大。

一般情况下，Android 应用程序是单进程的，而进程和组件类型是高度关联的。如果应用中有生命周期差异较大的组件，考虑使用多进程分别处理，具体在后面继续讨论。

第 2 章 视图类组件

2.1 输入事件分发机制

2.1.1 整体分发流程

在 Native 框架层，InputReader 通过 EventHub 读取输入事件（主要包括按键和触摸事件），并将事件通过队列转发给 InputDispatcher；之后 InputDispatcher 从队列提取事件并通过 socket 将事件分发到 Java 框架层的 InputEventReceiver。

InputEventReceiver 收到事件后，通过 ViewRootImpl、经由 DecorView 将事件转发给 Activity。

Activity 再通过 PhoneWindow、DecorView 以及 ViewGroup 将事件传给当前有焦点的子控件处理。

下面重点分析 Java 框架层的事件分发，Native 层的处理在后面再探讨。

2.1.2 Java 框架层的分发

1、Activity 中的事件分发

在 Activity 中，窗口由 PhoneWindow 负责管理，而 PhoneWindow 内部的布局视图、子控件又由 DecorView 管理。这样在事件分发时，Activity 先通过 PhoneWindow 将事件传递给 DecorView，再由 DecorView 通过 ViewGroup 一层一层传递给各级 View 组件处理，如果最底层的 View 没消费该事件，则逐层返回给上一级的 View 处理。

2、有 Dialog 组件时的事件分发

以常用组件 AlertDialog 为例，其内部也有一个 PhoneWindow、DecorView、ViewGroup，组织方式和 Activity 类似。

在 Activity 中，如果弹出一个 AlertDialog 且获取了焦点，则输入事件，不会先经过 Activity 而是直接分发给 AlertDialog。也就是说，Activity 无法拦截其弹出的 AlertDialog 的事件响应。

3、主要流程

Step1、NativeInputEventReceiver 转发给 ViewRootImpl

NativeInputEventReceiver 监听到对应的 socket 事件，然后通过 Jni 方法将事件传递给应用层 InputEventReceiver，进而传递给 ViewRootImpl。这里 ViewRootImpl 是个关键的桥梁，一端是 WindowInputEventReceiver 成员，另一端是 DecorView 成员。

Step2、ViewRootImpl 转发给 ViewGroup

ViewRootImpl 通过 InputStage 将消息传递给 Activity/Dialog。再由 Activity/Dialog 将该事件先后传递给 PhoneWindow、DecorView、ViewGroup。

Step3、ViewGroup 分发给子 View

ViewGroup 将事件分发到子 ViewGroup 和子 View。具体来说，在 ViewGroup 的 dispatchKeyEvent 函数中，事件会传给当前有焦点的 View，对应其 mFocused 变量。如果存在多重嵌套 View，则先从外层到内层，再从内层到外层返回（递归方式）。当 View 的事件处理完成后，ViewRootImpl 通过 Jni 返回事件处理完成通知。

在整个过程中，按键事件和触摸事件的处理机制类似，区别在于函数、事件的名称以及少量的分发逻辑有差异：

- 触摸事件使用 dispatchTouchEvent()，而按键事件使用 dispatchKeyEvent()，都是由 ViewGroup 调用，对事件进行分发。
- onTouchEvent() 处理触摸事件，onKeyEvent 处理按键事件。如果返回 true，表示事件已消耗，不再向下传递；返回 false，表示没有消耗事件，交回上层处理（父 View 或父 VIewGroup）。

其他还有拦截操作的差异，更详细的流程需要结合 InputManagerService 子系统源码来分析。

2.2 视图绘制

2.2.1 View 组件

View 组件有几个重要的方法需要关注，也是自定义 View 经常需要重写的方法。

1、measure

作用是测量 View 组件的尺寸。对应的方法是 onMeasure，测量 View 的宽和高。View 和 ViewGroup 都有 measure 方法，但 ViewGroup 除了测量自身尺寸，还要遍历地调用子元素的 measure 方法。

2、layout

用于确定布局位置。对应的方法是 layout、onLayout，用于确定元素的位置，ViewGroup 中的 layout 方法用来确定子元素的位置。

3、draw

作用是绘制内容背景，包括 View 的内容、子 View 的内容和背景。具体方法包括：drawBackground、onDraw、ViewGroup.dispatchDraw。

这个方法有个关键的入参也是唯一的入参：Canvas。其在 native 层有个对应的画布组件 SkCanvas，该组件内部封装的 SkBitmap 实现了类似画纸的功能。

在 Android 系统中，View.draw 的部分实现如下：

```
        // Step 1, draw the background, if needed
    int saveCount;
    if (!dirtyOpaque) {
        drawBackground(canvas);
    }

    // skip step 2 & 5 if possible (common case)
    final int viewFlags = mViewFlags;
    boolean horizontalEdges = (viewFlags & FADING_EDGE_HORIZONTAL) != 0;
    boolean verticalEdges = (viewFlags & FADING_EDGE_VERTICAL) != 0;

        if (!verticalEdges && !horizontalEdges) {
        // Step 3, draw the content
        if (!dirtyOpaque) {
            long logTime = System.currentTimeMillis();
            onDraw(canvas);
        }
          ...
```

```
        // Step 4, draw the children
        dispatchDraw(canvas);
        drawAutofilledHighlight(canvas);
```

```
        // Overlay is part of the content and draws beneath Foreground
        if (mOverlay != null && !mOverlay.isEmpty()) {
            mOverlay.getOverlayView().dispatchDraw(canvas);
        }
```

```
        // Step 6, draw decorations (foreground, scrollbars)
        onDrawForeground(canvas);
```

其中绘制背景的实现，是通过 Drawable 类型的变量 mBackground 也就是背景图来完成的。View 中的 onDraw 是个空方法，但在具体的 View 组件例如 ImageView 中，都会有对应的扩展实现。

这里的绘制，主体功能通常是 Drawable 和 Canvas，后者处理绘制区域、矩阵变换等工作，前者根据组件功能，实现具体绘制操作，例如 BitmapDrawable 中的 draw 方法，最终是通过 Paint 画笔组件和 Canvas 的 drawBitmap 方法实现绘制操作的。

对于自定义视图组件，通常需要重写 onDraw 方法，并在该方法中调用 Canvas 的

drawXXX 例如 drawBitmap、drawText 等方法来实现预期的效果，如果涉及到几何变换，还可能会用到 Matrix 组件。

2.2.2 Canvas 组件

作为绘制处理中的画布组件，Canvas 提供了裁剪区域/路径和多种图形/图像绘制功能，对应的方法有：

- clipRect、clipPath、clipRegion；
- drawBitmap、drawTex、drawLine；

具体实现都是在 native 层完成的，涉及 SkiaCanvas、SkCanvas。

```
public boolean clipRect(@NonNull RectF rect, @NonNull Region.Op op) {
    checkValidClipOp(op);
    return nClipRect(mNativeCanvasWrapper, rect.left, rect.top, rect.right, rect.bottom, op.nativeInt);
}
```

在 framework 层的 android_graphics_Canvas.cpp 中，有对应实现：

```
static jboolean clipRect(jlong canvasHandle, jfloat l, jfloat t,
                         jfloat r, jfloat b, jint opHandle) {
    bool nonEmptyClip = get_canvas(canvasHandle)->clipRect(l, t, r, b,
            opHandleToClipOp(opHandle));
    return nonEmptyClip ? JNI_TRUE : JNI_FALSE;
}
```

该方法会调用到 SkiaCanvas.cpp 中的实现，这里先不展开分析。

2.3 问题与思考

2.3.1 Activity、Window 和 View 之间关系

1、Window 和 PhoneWindow

PhoneWindow 类，派生于 Window，是连接 Activity 跟 View 的桥梁，所有 Activity 对 View 的操作都需要借助它。

2、Activity 和 Window

在 Activity 启动过程中，系统会调用 attach 方法，并创建 PhoneWindow 实例；之后，在具体 Activity 的 onCreate 函数中，通常都会调用 setContentView 方法，进而调用 PhoneWindow 的 setContentView 方法设置 DecorView。

3、Window 和 View

Window 通过 WindowManagerImp 类型的成员变量 mWindowManager 操作 View。WindowManagerImpl 是 WindowManager 接口的具体实现，该类通过 WindowManagerGlobal 完成 addView、removeView、updateViewLayout 这三个方法。

进一步地，PhoneWindow 的成员变量 DecorView，可以通过 getViewRootImpl 方法获取 ViewRoot 实例。与之对应的 ViewRootImpl 类，有个 IWindowSession 类型的成员变量 mWindowSession，能够访问 WMS。

小结:PhoneWindow 类，是连接 Activity 跟 View 的桥梁；而 ViewRootImpl 是 View 和 WindowMangerService 之间的桥梁。

2.3.2 事件分发、视图绘制相关核心服务

本章仍然属于组件体系，对输入事件包括触摸、按键事件分发流程重点是从 Java 层 Activity 组件、Dialog 组件以及 ViewGroup 组件的角度来分析的，实际上，输入事件的处理与分发流程复杂很多，包括 Native 层的 InputReader、InputDispatcher 以及

InputMonitor 等，涉及到 InputManagerService、ActivityManagerService 等核心服务。

视图绘制，特别是 View 组件的 draw 分发的执行场景或触发时机，又和图形渲染子系统密切相关，相关的组件还包括 Surface、SurfaceSession、SurfaceView 等。视图绘制的触发时机，则分为垂直同步信号、视图可见性、位置等属性变化而触发重绘，具体在第三篇（核心框架服务）进一步分析。

第 3 章 数据存储组件

3.1 序列化与反序列化

序列化，就是将对象转换为可以传输的二进制序列。其作用主要有：

- 永久性地保存对象
- 在网络中传递对象
- 在进程间传递对象

Serializable 接口，可以实现磁盘序列化，将数据保存到磁盘（Flash），基于 ObjectInputStream/ObjectOutputStream 类实现。使用简单、但性能较差。支持通过 writeObject 和 readObject 方法加以客制化。注意：static、transient 变量不会被序列化，另外反序列化时，默认不会执行构造函数。

Parcelable 接口对应的序列化操作，默认只在内存中进行，可用于 Intent 数据传递和进程间通信。写入和读取时候需要手动添加自定义代码，使用复杂但效率高。

如果要持久化存储（保存到磁盘文件），推荐使用 Serializable。其他方案还有：JSON、Protocol Buffers、Serial。

3.2 Parcel 与 Bundle

3.2.1 Parcel

Parcel，是一种存储基本数据类型和引用数据类型的容器，定位是轻量级的、高效的对象序列化机制，操作的是连续的内存空间，可用于跨进程的数据传递。

3.2.2 Bundle

Bundle 派生于 BaseBundle，实现了 Parcelable 接口。BaseBundle 类中定义了 ArrayMap<String, Object>类型的成员变量 mMap，基于 key-value 键值对读写数据，多用于 Activity 之间的数据传递（数据大小不要超过 1MB）。

3.3 SharePreferences

3.3.1 使用场景

SharePreference 使用方便，但是容易出现各种性能问题，例如：跨进程不安全、加载较慢、不支持增量写入、可能引发等。适用于存储一些非常简单、轻量级的数据，例如业务功能的参数配置，不适合存储过于复杂的、较大的数据。如果要跨进程存储、访问数据，可以使用文件，或者使用 ContentProvider。

注意，该组件的 apply 调用会以异步方式将数据提交到内存，而 commit 是同步方式提交到数据库。不关心提交操作是否成功时，优先考虑 apply。

3.3.2 优化思路

1、合并多次的 apply 操作。

2、使用文件锁以保证跨进程安全。

3、使用 mmap 避免数据丢失。

4、根据需要扩展加密功能。

具体可以参考微信开源的 MMKV 组件的实现。

3.4 文件存储

用于实现文件存储的类主要有两个：FileInputStream 和 FileOutputStream。保存文件内容，可通过 Context.openFileOutput 获取输出流，参数分别为文件名和存储模式；读取文件内容：可通过 Context.openFileInput 获取输入流，参数为文件名；删除文件，可通过 Context.deleteFile 删除指定的文件，参数为将要删除的文件的名称。

几个常用的读写模式：

- MODE_PRIVATE：私有，只有自身能够访问，覆盖写入
- MODE_APPEND ：私有，只有自身能够访问，末尾追加
- MODE_WORLD_READABLE：公有，外部应用可读
- MODE_WORLD_WRITEABLE：公有，外部应用可写

3.5 SQLite 数据库

SQLite 数据库相关的常用组件，是 SQLiteDatabase，派生于 SQLiteClosable，通过 SQLiteStatement 类完成具体 SQL 语句的执行。

该组件默认支持多进程并发操作（通过文件锁加以控制），支持多线程并发（通过文件锁）；支持读并发但不支持写并发，会出现异常。

其性能优化思路：

- 索引优化，需要建一个索引表（需要维护表的更新）；
- 建立正确的索引
- 选择合适的字段（Projection）
- 合理配置页大小与缓存大小（页大小默认是 1KB，缓存大小默认 1000 页）
- 慎用“select*”（尽量精确）
- 必要时拆分表格
- 及时清理无用数据

3.6 问题与思考

3.6.1 使用场景对比

1、Parcel：实时性要求较高的、无需永久存储的进程间数据共享

2、Bundle：四大组件间的数据共享和进程通信

3、文件：简单的、实时性要求不高的场景

4、SQLite 数据库：复杂的、多个字段互相关联的大批量数据存储

5、ContentProvider：一对多的进程间的数据共享

3.6.2 依赖关系

Bundle 实现了 Parcelable 接口，其构造函数依赖于 Parcel，父类的 mParcelledData 成员变量用于管理数据。

这里可能容易想到，Bundle 为何需要支持序列化？首先，Bundle 用于 Intent 中的数据传递；其次，Intent 启动其他组件往往需要通过 Binder 跨进程；最后，Binder 跨进程通

信的数据传递必须支持序列化。

Parcel 和 Bundle，都可以利用文件实现持久的序列化存储。ContentProvider 可以使用数据库也可以使用文件存储具体内容。

第二篇 进程与线程体系

第 4 章 进程管理机制

4.1 Linux 中的进程管理

在 Linux 中，进程是指处理器上执行的一个实例，可使用任意资源以便完成它的任务，具体的进程管理，是通过“进程描述符”来完成的，对应 Linux 内核中的 task_struct 数据结构。进程描述符，包括进程标识、进程的属性、构建进程的资源。

一个进程可以通过 fork()或者 vfork()调用创建出子进程，这些子进程可以访问父进程的地址空间，包括文本段、数据段、堆栈段。

通常情况下，调用 fork()的进程处于 task_running 状态，则 fork 出来的子进程默认也处于 task_running 状态，具体来说，在 fork 之后、exec 之前，子进程处于 task_running 状态中的就绪状态。

进程的运行状态包括以下几种。

1、task_running：可执行状态。包含正在 CPU 上执行的、可执行但是尚未被调度执行这两种子状态，后者对应就绪状态。

2、task_interruptible：可中断的睡眠状态。因为等待某事件的发生而被挂起。当等待的事件发生时，处于该状态的进程将被唤醒。

3、task_uninterruptible：不可中断的睡眠状态。处于睡眠状态，但是此刻进程是不可中断的。此时进程不响应异步信号，不能通过发送信号的方式 kill 之，但可以响应硬件中断，例如磁盘 IO，网络 IO 等。

4、task_stopped / task_traced：暂停状态或跟踪状态。处于 task_traced 状态的进

程不能响应 SIGCONT 信号而被唤醒，只能等到调试进程通过 ptrace 系统调用执行 ptrace_cont、 ptrace_detach 等操作，或调试进程退出，被调试的进程才能恢复 task_running 状态。

5、task_zombie ： 僵尸状态。在进程收到 SIGSTOP、SIGTTIN、SIGTTOU 等信号，即将终止时，会进入该状态，此时进程成为僵尸进程。该状态的进程会处理一些资源释放工作，然后发送 SIG_CHLD 信号给父进程。

4.2 Android 进程管理机制

在 linux 系统中，应用程序执行完成后，最终会清理一些进程使用的文件描述符、释放掉进程用户态使用的相关的物理内存，清理页表，然后发送信号给父进程。但在 Android 系统中，应用程序执行完成后，该应用所在的进程通常还是会在后台继续运行，除非应用程序在执行完成后主动调用 System.exit 或者 Process.killProcess 之类的方法。

这样设计的好处，主要是加快应用程序再次启动的速度，改善用户体验。当内存不足时，系统会按照特定规则，包括进程优先级、占用内存等信息，来清理进程并释放对应的资源。

4.2.1 进程优先级

在 Android 中，进程按优先级可以分为：前台进程、可见进程、服务进程、后台进程、空进程。优先级依次降低。

1、Forground（前台进程）

这种进程优先级最高，可以细分下面几种情况：

case1：有个前台 Activity，特指已经执行了 onResume 但还没执行 onPause 的 Activity；

case2：有个 Service 且和一个前台 Activity 绑定的进程；

case3：调用了 startForground 的前台 Service 所在进程（这种服务会带个通知）；

case4：正在执行 onReceive 函数的 BroadcastReceiver 所在进程，以及正在执行服务的生命周期方法诸如 onCreate、onStartCommand 的进程。

2、Visible（可见进程）

可见进程没有处于前台的组件，但是用户仍然能看到进程中的组件，例如某进程的 Activity 调用了申请权限对话框，具体包括：

case1：有个仅 onPause 被调用的 Activity（可见但被遮挡）；

case2：进程中有个 Service 且和一个可见 Activity 绑定。

注意这里的可见 Activity 不包括前台 Activity（否则就是前台进程了），并且这种进程在内存不足时也是可能被杀掉的。

3、Service（服务进程）

服务进程是指有个通过 startService 方式启动的 Service 进程，并且不属于前面两类进程，例如 MediaScannerService。

4、Background（后台进程）

当前不可见的 Activity 所在进程属于后台进程，即它们的 onStop 被调用过，例如用户按下 home 键。

对于后台进程，系统会保存这些进程到一个 LRU 列表，当系统需要回收内存时，LRU 中那些最近最少使用的进程将被杀死。

5、Empty（空进程）

空进程不包含任何组件，当系统重新需要它们时（例如用户在别的进程中通过 startActivity 启动了它们），可以省去 fork 进程、创建 Android 运行环境等漫长的工作，节省时间，缓存性质。这种进程优先级最低，在任何时候都可能被杀掉。

如前所述，大部分应用是单进程的，但是进程和组件类型是高度关联的。如果应用中有生命周期差异较大的组件，考虑使用多进程分别处理。一方面是让占用资源较多的进程可以被系统及时回收，另一方面，避免那些需要长时间持续运行的任务由于组件生命周期的影响进入后台进程执行。

4.2.2 内存不足时的杀进程策略

在 Linux 系统中，进程的优先级对应一个参数，也就是 oom_score_adj，lowmemorykiller 程序会根据内存使用情况和进程优先级，动态杀进程以释放内存资源。

当内存不足或者发生 oom 错误时，lowmemorykiller 根据特定策略先杀优先级最低的进程，然后逐步杀优先级更低的进程（同样优先级会按照内存占用情况排序），依此类推，以回收预期的可用系统资源，从而保证系统正常运转。

在 Android 系统中，进程的组件状态变化时，组件所在进程的优先级也会发生变化。一个典型的场景是，切换到后台的进程，其优先级低于前台进程。

App 的前台/后台切换操作对 oom_score_adj 的影响，我们可以使用 adb 命令 cat/proc/[pid]/oom_score_adj 来查看。详细的优先级信息在 ProcessList.java 中有定义，对应的部分进程类型如下。

- Cached，缓存进程，包括空进程、只有 activity 的后台进程，其 adj 在 900~906；
- B Services，无 UI 组件且在 Lru 进程表中位于后 2/3 的服务进程（比较旧的后台服务进程），其 adj 为 800；
- Previous，上一个应用进程，例如按 home 键进入后台的进程，其 adj 为 700；
- Home，也就是 launcher 进程，其 adj 为 600；
- A Services，没有 UI 组件且在 Lru 进程表中位于后 2/3 的服务进程（比较旧的后台服务进程），其 adj 为 500；
- Perceptible，有着用户可感知组件的进程，包括前台服务进程，adj 为 200；
- Visible，可见进程，其 adj 为 100；
- Foreground，前台应用进程，其 adj 为 0；
- Persistent，系统常驻进程，例如 systemui、phone 进程，其 adj 为-700 或-800。如果是在 AndroidManifest.xml 中申明 android:persistent="true"的进程，adj 为-800；如果是由 startIsolatedProcess()方式启动或由 SystemServer 进程、persistent 进程绑定的服务进程，则为-700；
- System 进程，典型的是 SystemServer 进程。
- Native 进程，例如 init、surfaceflinger、mediaserver 进程，其 adj 为-1000；

内存不足时，AMS 会根据上述动态优先级信息，通过 ProcessRecord，在 native 层给指定进程发送信号以终止进程，进而释放内存资源。相关函数包括：killProcessesBelowForeground，killProcessesBelowAdj，ProcessRecord.kill，killPids 等。

第 5 章 进程间通信

5.1 常见通信方式

进程间常见的通信方式有以下几种：

1、Socket：通用接口，传输效率低，主要用在跨网络通信和本机进程间通信，传输过程需要拷贝 2 次数据；

2、共享内存：虽然无需拷贝，但控制复杂；

3、Binder：基于 C/S 模式，只需 1 次拷贝，安全性高。

不同的通信方式使用场景也不同：

- Socket：适合网络间的通信，或者效率要求不高的本机进程间通信；
- 共享内存：适合效率要求较高的底层进程间通信；
- Binder：兼顾效率和安全性的进程间通信，需要返回结果；
- Messenger：低并发的、一对多的进程间通信，无需返回结果。

5.2 Binder 和 AIDL

5.2.1 Binder 机制浅析

Binder 机制，整体上采用 C/S 架构。客户端先获取远程服务端的对象引用，通过该引用调用远程函数，然后由 Binder 驱动找到对应的服务端函数再执行。

其中，Binder 驱动是作为一个特殊字符型设备存在，设备节点为/dev/binder，遵循 Linux 设备驱动模型。

在驱动实现过程中，系统主要通过 binder_ioctl 函数与用户空间的进程交换数据。

binder_thread_write 函数用于发送请求或返回结果，而 binder_thread_read 函数用于读取结果。

5.2.2 AIDL

AIDL 本质是系统提供的一套可快速实现 Binder 调用的工具/语言，其中工具也就是 aidl.exe，可将 aidl 文件转为 java 文件，语言就是 aidl 文件使用的接口描述语言，用于.aidl 文件。

1、关键类和方法

asInterface()：AIDL 接口中的一个方法，提供客户端调用。该方法可以将服务端的返回的 Binder 对象或 BinderProxy 对象，转换成客户端所需要的 AIDL 接口类型对象（一个 Stub.Proxy 实例）。

Proxy 类：服务端在客户端进程的代理类。客户端通过 Binder 对象的 asInterface 方法获取 Proxy 实例，再通过该实例调用服务端的自定义方法。

transact()方法：运行在客户端。客户端调用具体功能方法时，需要在接口类中，通过 BinderProxy 类型的 mRemote 对象的 transact 方法发起远程请求，同时将当前线程挂起，直到远程请求返回才继续执行。

AIDL 接口：继承 android.os.IInterface，由客户端的 aidl 文件通过编译自动生成。例如 IActivityManager 这个接口就是由 IActivityManager.aidl 自动生成的（客户端和服务端都有）。

Stub 类：Binder 的实现类。服务端通过这个类来提供服务。注意是抽象类，其 static 方法 asInterface 给客户端调用，其他方法给系统调用，最终会调用到具体实现(一般在 Stub 的实例对象中)

onTransact()方法：运行在服务端的 Binder 线程池中，当客户端发起 transact 跨进程请求时，系统回调此方法处理请求。

补充，transact 调用中的 code 参数，一般对应具体的功能，类似 index，一个 Binder 对象使用的 code 值，在应用内部必须能够区分，但不同的 Binder 对象使用的 code 值可以重复。

在服务端进程中，有个 Stub 类，这个是 Binder 的实现类，服务端通过这个类来提供服务。注意该类是抽象类，其 static 方法 asInterface 给客户端调用，其他方法给系统调用，最终会调用到具体实现（一般在 Stub 的实例对象中），其 onTransact()方法运行在服务端的 Binder 线程池中，当客户端发起 transact 跨进程请求时，系统回调此方法处理请求)。

在客户端进程中，asInterface()是 AIDL 接口中的一个方法，提供客户端调用。该方法可以将服务端的返回的 Binder 对象或 BinderProxy 对象，转换成客户端所需要的 AIDL 接口类型对象（一个 Stub.Proxy 实例）。

2、代理类的设计

BinderProxy 是 Android 系统用于跨进程通信、访问远程服务中的 Binder 实例对象而设计的，侧重系统调用，对用户不透明。实际上它是在 native 层 javaObjectForIBinder 方法中创建的，属于客户端进程中的对象，和 Native 层的 BpBinder 有对应关系。

形如 IxxxService.Stub.Proxy 这种代理类，用于跨进程通信、访问远程服务中的用户自定义方法而设计的，侧重应用层，实际上该类基本都是通过其实现的 AIDL 接口例如 IActivityManager 来调用。

对于 Service 组件的远程绑定，BinderProxy 与 Binder 实例（比如 IxxxService.Stub 类型的变量 mBinder）的映射关系，是通过 BinderProxy.ProxyMap 这个类维护的。而 IxxxService.Stub.Proxy 实例与 Binder 实例（IxxxService.Stub 类型的变量 mBinder）的

映射关系，是通过 Proxy 类的构造函数及其成员变量 mRemote 维护的，这里 mRemote 就是 mBinder 在客户端进程的代理也就是 BinderProxy 对象。

如果我们获取到 Proxy 类的实例，可以调用其 asBinder 方法获取该实例维护的 BinderProxy 对象，反过来也可以通过 BinderProxy 实例的 asInterface 方法来获取 Proxy 对象。

5.3 问题与思考

5.3.1 Binder 常见使用场景

场景 1：绑定服务，例如 WallpaperService，其代理对象所需的 BinderProxy 实例（远程服务时），是 onServiceConnected 回调参数中 IBinder 类型的 service。

场景 2：系统服务，例如 AMS、WMS 等，其代理对象所需的 BinderProxy 实例，是通过 ServiceManager.getService 方法调用返回的对象。

场景 3：间接访问核心组件，例如 IWindowSession。IWindowSession 的实现类是 Session，不属于服务。客户端例如 WindowManagerGlobal，先通过 ServiceManager 获取 WMS 的本地代理也就是 IWindowManager 的实例，再利用此代理的 openSession 方法得到 Session 的本地代理 IWindowSession。

Binder 跨进程通信的关键，是一个实现了 IBinder 接口的对象，该对象可用于调用远程服务的本地 Binder 类的 asInterface 方法，获得远程类的本地代理。如前所述，获得 IBinder 对象的途径有 3 种以上，包括利用 ServiceConnection 回调（绑定服务）直接获得；或者利用 ServiceManager（系统服务）直接获得；或者通过别的代理（基于系统服务或绑定服务）对象的某些方法间接获得。

除了上面 3 种场景，还有一种特殊情况：IServiceManger。该类派生于 android.os.IInterface，定义了用于实现具体功能的接口，例如 addService、getService。ServiceManagerNative 派生于 Binder，作为远程 ServiceManager 的本地 Binder 类，提供 asInterface 方法（对应工具自动生成的 java 类中的 IxxService.Stub）。ServiceManagerProxy 是远程 ServiceManager 的本地代理类，提供了各项具体功能方法，利用 Parcel、IBinder 和远程 ServiceManager 通信并返回结果（对应于工具生成的 IxxService.Stub.Proxy）。获取本地代理，首先需要得到一个 IBinder 接口实例，这里主要是通过 BinderInternal.getContextObject()完成的，属于系统级的一个特殊处理。实际上是 native 层创建的。远程 ServiceManager，也就是 addService 等功能具体的实现，最终是 service_manager.c 中完成。

这里可以看出，Binder 通信的灵活性，不一定需要 aidl 文件，远端服务也不一定要包含一个 Stub 对象或者派生于 Stub 类，可以直接在 native 层。

AIDL 机制既可以实现跨进程通信，也支持在同一进程内的调用；在同一进程时，onServiceConnected 回调的入参 Binder 对象，就是服务端返回的 Binder 实例，无需转换（对于进程的判断，以及 Binder 对象的对应区分，例如是 Binder 还是 BinderProxy，是在驱动里面判断的，详见 binder.c 中的 binder_translate_handle）。

5.3.2 Binder 和 Parcelable 接口

Binder 进程通信机制帮我们解决的最大问题，就是能够通过一个代理来访问其他进程中的实例对象及其方法，其中的数据传输，大量使用了 Parcel 以及 Parcelable 接口。

如果是跨进程传输自定义数据类（可序列化的），可以使用实现了 Parcelable 接口的自定义数据类，同样需要对应的 aidl 文件。如果一个类在创建实例时，必须调用无法序列

化的组件，则需要给客户端提供代理才能访问该类，也就是说，服务端的实现必须派生于Ixxx.Stub（Binder 的派生类），使得客户端通过某种方式可以获取其代理。

对于功能简单的调用，例如数据访问相关功能，使用一个实现了 Parcelable 接口的类基本就可以了，最多是引用其他 aidl 接口。例如 KeyEvent，用于序列化的构造函数 KeyEvent(Parcel in)里面都是可以可以序列化的数据类型。但如果自定义类的功能复杂，依赖大量组件，重点是构造函数参数涉及到不属于 aidl 可以序列化的类时，则无法实现 Parcelable 接口，只能给调用者/客户端提供一个代理来访问这些功能，这时候就需要定义 aidl，并且让功能实现类派生于 Stub（实质上是派生于 Binder 类）。例如 Session 类，其构造函数的几个参数都不能序列化，这个是其没有实现 Parcelable 接口却却实现了 IWindowSession.Stub 的关键原因。

5.3.3 Messenger 跨进程通信

Messenger 跨进程通信可以和 Message-Handler 多线程通信结合起来。从 Message 源码可以发现，这个类实现了 Parcelable 接口，如前所述，该接口可以方便地用于跨进程的消息通信。这种 Messenger 通信机制，可以实现一对多的通信，也可以实现任意两个进程之间的通信。

该通信方式，支持回调，也不需要编写 aidl 文件。其通信过程，重点是在服务端定义一个 Service，再声明一个 Messenger 对象，然后在 Service 的 onBind()方法返回 mMessenger.getBinder()、并重新 Handler.handleMessage 方法。之后就是客户端发送消息发送到服务端，回调其 handleMessage 方法，根据 what 属性执行对应的任务并返回结果。

注意，Message 在服务端处理时，也需要排队；另外，处理结果不能立即获取，而是

异步方式返回给客户端。

第 6 章 多线程及其实现

6.1 多线程实现方式

在 Java 编程中，通常有两种方式实现多线程，一种是继承 Thread 类并重写 run()方法，另外一种就是实现 Runnable 接口并实现该接口的 run()方法。

推荐用实现 Runnable 接口的方式。原因是，一个类应该在其需要加强或者修改时才会被继承。如果没有必要重写 Thread 类，那么最好用实现 Runnable 接口的方式。

这里说的用 Runnable 接口实现多线程，是指创建一个 Thread 对象，然后用 Runnable 对象（包括实现了 Runnable 接口的类或匿名内部类）作为参数构造该线程对象，再调用线程的 run 方法来启动线程。另外，在创建线程时，无论是何种方式，尽可能给线程命名，以方便定位线程相关问题。

6.2 线程池

作用是减少线程创建和销毁次数，以减少系统资源消耗。使用线程池时，系统先判断核心线程池中的线程是否已满，没满则创建一个核心线程；否则，再判断工作队列是否已满，没满则加入工作队列等待；否则继续判断线程数是否达到最大值，如果不是，则创建一个普通线程，如果线程数已到最大值，执行饱和策略，抛出异常。

小结，根据核心线程池+工作队列+线程数综合判断，进行线程的创建或者等待或是抛出异常，包括 4 种情况，注意有先后顺序：

1) 核心线程池未满，则创建核心线程；

2) 核心线程池已满但队列未满，则排队等待；

3) 队列已满但线程数未达最大，则创建普通线程；

4) 队列已满且线程数最大，则抛出异常。

常用的 4 种类型线程池：

- FixedThreadPool：固定线程数
- CachedThreadPool：缓存线程池（按需创建）
- SingleThreadExecutor：单工作线程
- ScheduledThreadPool：周期性调度

6.3AsyncTask

AsyncTask 用于处理异步请求的一种相对简单快捷的组件，适用于待执行的任务和 UI 交互比较多、但不是很耗时的场景。其原理是，通过线程池和阻塞队列 BlockingQueue 以及 Handler、WorkerRunnable 来处理异步请求；内部自带线程池 SerialExecutor 处理任务排队；并使用了阻塞队列 LinkedBlockingQueue。

注意该组件的线程池中的核心线程数有限制，其他线程需要排队等待；虽然调用 cancel 方法可以取消任务，但有时候未必会立即起作用，任务未取消则最终回调 onPostExecute 方法（可返回执行结果）。

该组件使用的线程池，和系统自带的 4 种都不完全一样，相对来说比较接近第一种 FixedThreadPool。特别提醒：不建议用于耗时任务的执行（因为核心线程数有限制，如果核心线程满了，新任务会排队等待，依次执行）。

异步方式处理耗时任务，可以使用 Handler+Runnable，也可以使用自定义线程池，定义为能够并发执行任务的线程池。

由于 AsyncTask 的生命周期可能比 Activity 的长，当 Activity 进行销毁 AsyncTask 还在执行时，由于 AsyncTask 持有 Activity 的引用，导致 Activity 对象无法回收，进而产生内存泄露。

除了 AsyncTask 组件，还有一些多线程的实现方式，例如 HandlerThread、IntentService。前者是一种有着自己的内部 Looper 对象的线程类，而后者则是一种 Service，通过自带的 HandlerThread 和 ServiceHandler 来处理 Intent 中的消息。

第 7 章 线程管理与通信

7.1 线程状态切换

7.1.1 线程状态

- New：新建（线程已创建、还没启动）
- Runnable：可运行（未必处于运行状态）
- Blocked：阻塞，表示线程被锁阻塞
- Waiting：等待，直到线程调度器重新激活
- Timed waiting：限时等待，可以在指定时间自行返回 Runnable 状态
- Terminated：终止（包括线程执行完毕的正常退出以及未捕获异常导致的提前退出）

线程管理相关的常用方法：

- run：线程内调用 Runnable 对象的 run()方法，可多次调用；
- start：启动一个新线程，新线程会执行相应的 run()方法。不能被重复调用。对 New 状态的线程，调用该方法会使其进入 Runnable 状态。
- join：等待其他线程执行结束。调用该方法会使得线程进入 Waiting 或 Timed waiting 状态（取决于有无入参）。
- sleep：Thread 的方法，用于控制线程、使得当前线程在指定时间内休眠，此调用不会释放锁；调用该方法会使得线程进入 Timed waiting 状态。
- wait：Object 的方法，用于线程间通信（需要配合使用 notify 或者 notifyAll），该调用会释放锁；调用该方法会使得线程进入 Timed waiting 状态。
- yield：是当前线程放弃对处理器的占用，降低线程优先级。

7.2 互斥与同步

互斥是指通过竞争对资源进行独占使用，彼此之间不需要知道对方的存在，执行顺序可能是乱序。而同步则是协调多个相互关联的线程，合作完成任务，彼此之间知道对方存在，执行顺序往往是有序的。

在 Java 编程中，并发量比较小时，优先选择使用 synchronized 关键字，如果性能要求较高，考虑使用 Lock 类例如重入锁 ReentrantLock，更灵活、更广泛、粒度更细。

注意 synchronized 是 Java 关键字、基于 JVM 实现，系统会自动释放锁；而 Lock 则是一个 Java 接口，基于 JDK 实现，使用 Lock 时必须手动释放锁，并且尽量把解锁操作放在 finally 代码块。

ReentrantLock 可重入锁，是基于内部的公平锁和非公平锁实现的。这种细分的设计，是因为某些业务场景中会需要保证先到的线程先得到锁。

公平锁的锁获取，严格按照排队等待顺序进行锁获取；而非公平锁，获取锁的顺序是随机的。在获取锁的时候，非公平锁会多次尝试获取锁，并不着急加入队列排队，如果多次获取也没获取成功，就加入排队，此时和公平锁一样。

其两者的区别在于：非公平锁，加大了新线程抢到锁的概率。

问题：如何停止一个线程的执行?

分析：可使用 interrupt()方法或者自定义的 boolean 变量（须 volatile 修饰）。具体来说，用 interrupt()方法中断 sleep()也就是休眠状态线程的执行，用自定义 boolean 变量停止其他状态的线程的执行。

7.3 Message-Handler 机制

7.3.1 原理

1、静态结构

Message-Handler 消息处理系统在静态结构方面，可以分为：Message、MessageQueue、Looper、Handler。

Message 也就是消息，通常用于表示一个可执行任务，消息内部含有 what、target、callback，以及消息参数。可以通过 what 区分不同消息、对应执行不同的任务，也可以直接携带一个 Runnable，也就是 callback 成员。

2、主要流程

首先，系统创建一个 Handler 对象，通过调用其 sendMessage 方法将消息加入消息队列；其次，创建 Looper 并使其循环起来，以持续将消息从 MessageQueue 中取出来；最后 Looper 根据取出的 Message 回调 Hander 的 handleMessage 方法，从而实现线程间的通信。

这里如果 Message 的 Runnable 类型成员 callback 不为空，则执行之。Message-Handler 机制支持延时发送消息，具体是在 Looper 调用 MessageQueue.next() 方法取出 Message 时，next()方法会判断准备取出的 Message 的时间属性，通过该时间属性判断是否立即返回给 Looper。

7.3.2 使用场景

1、在主线程创建 Handler

此时我们不需要手动创建 Looper，因为在应用启动时，系统创建应用进程时，也创建了一个默认的线程，就是主线程，ActivityThread 的 main 方法就运行在这个线程中。在 main 方法中，有个 Looper.prepareMainLooper 方法，会创建一个主线程的 Looper 实例，之后调用 Looper.loop()使其循环起来，所以不需要再另外创建 Looper 实例。

2、在子线程创建 Handler

此时由于这个线程中没有默认开启的消息队列，所以我们需要手动调用 Looper.prepare()创建 Looper 对象，再通过 Looper.loop 开启消息循环，最后再创建 Handler。

这里 Looper.prepare()的用途，是创建一个 Looper 对象并保存到 ThreadLocal 类型的 sThreadLocal 对象中，而 Looper.loop 方法则是使其循环起来。

ThreadLocal 的功能是，在不同的线程之中互不干扰地存储、提供数据，可以让多个线程使用同一个变量（线程局部变量）的副本，来解决多线程中的数据并发问题。ThreadLocal 还可以保证每个线程都有各自的 Looper，且只创建一个 Looper。

7.4 问题与思考

7.4.1 Looper 和循环

问题：主线程中的 Looper 死循环，为什么不会影响主线程例如生命周期回调的执行？

首先，ActivityThread 有个内部类 H，派生于 Handler。ActivityThread 在收到不同的 Message 时，通过 handleMessage 执行对应的操作，各个生命周期是通过这里的消息处理完成的。

另外，这里的死循环，不会过于占用 CPU 资源而影响其他线程的执行，是因为 Looper 的 queue.next()中，有个 nativePollOnce()的调用。该调用会让线程释放 CPU 资源进入休眠状态，直到下个消息到达或者有事务发生,并不会消耗大量 CPU 资源。或者说，Looper 对应的队列没有消息时，所在线程会释放 CPU 资源并进入休眠。

问题：在子线程也可以显示 Toast，但需要调用 Looper.prepare，为什么?

这和 Toast 的显示原理有关。Toast 在调用 show 方法时，会通过 Binder 调用 NotificationManagerService 的 enqueueToast 方法，排队处理。如果轮到当前 Toast，才会执行显示处理。简单来说，就是 Toast 的显示，基于队列实现，如果没有能够循环的 Looper 去推动队列的处理，即便是调用了 show 也不会显示。

7.4.2 唯一性问题

再看下面的几个问题。

1、一个线程是否可以有多个 Looper、多个 Handler?

2、一个线程有多个 Handler 时，Looper 如何区分这些 Handler?

问题 1，一个线程可以有多个 Handler，但只能有一个 Looper，并且如果存在多个 Handler，这些 Handler 使用同一个消息队列和同一个 Looper。只有一个 Looper 的原因，可以参见前面 ThreadLocal 的说明。

问题 2，这里先看下 Handler 和 Looper 是如何关联起来的。Handler 是在创建时，和当前线程的 Looper 关联起来的，有两种方式，隐式关联和显示关联。其中隐式关联就是不直接指定 Handler 关联的 Looper，由系统在 Handler 的构造函数中，通过 Looper.myLooper 调用来完成关联；后者是在创建 Handler 实例时，直接传入指定的 Looper。

然后通过 Handler 发消息时，Message 有个 target 属性，这个可以被 Looper 用来区分使用哪一个 Handler 分发、处理消息。

7.4.3 Handler 的内存泄漏问题

我们在使用 Handler 时，往往会使用匿名的 Handler 内部类来处理消息，这时候可能会出现内存泄漏。

典型场景是，在 Activity 的 UI 线程使用非静态匿名 Handler 内部类时，会出现内存泄漏，但静态匿名内部类不会。

其原因以及拓展分析有下面几点：

1、匿名内部类默认会持有外部 Activity 实例的引用，在 Activity 被销毁时，由于该 Activity 被匿名内部类持有而不能释放；

2、匿名内部类之所以会持有外部类的引用，和编译器有关，匿名内部类在编译时会有个专门的 class 文件，并且构造函数传入了外部类的实例引用。

3、静态内部类，编译时不会产生专门的 class 文件，也不会产生外部类的引用。

4、该场景下，存在内存泄漏时，GC Root 引用链为：Activity->Handler->Message->Queue->UI 线程，所以，如果在一个可以运行结束的子

线程使用非静态内部匿名 Handler，则不会出现内存泄漏，因为子线程结束时，引用链也会断掉。

5、优化方案：使用静态内部类、弱引用，在 Activity 销毁时，调用 Handler 的 removeCallbacksAndMessage 方法，目的是使得引用链断掉，而不影响 GC 操作。

第三篇 基础框架体系

第 8 章 开机启动

系统开机后，首先会执行 bootloader，将内核程序加载到内存，再通过内核启动用户空间的 init 程序。其次，init 程序解析 init.rc 配置文件，并开启 zygote 进程。在这之后，Zygote 进程启动子进程 SystemServer、开启核心系统服务并将服务添加到 ServiceManager，然后进入 SystemReady 状态。最后，ActivityManagerService 利用 socket 通知 zygote 启动桌面应用 Launcher ，完成整个启动过程。

8.1 启动 init 进程

8.1.1 初始化并启动属性服务

在 system/core/init/init.cpp，系统先调用 property_init 来对属性进行初始化，再通过 start_property_service 启动属性服务，最后再调用 sigchld_handler_init 以设置子进程信号处理函数。其中属性服务用于存储、读取系统属性包括用户自定义属性。

property_init 和 start_property_service 函数的实现，在 property_service.cpp 中。其中 start_property_service 会创建一个 Socket 用于监听客户端请求，并调用 register_epoll_handler，给客户端请求设置回调函数。

```
    property_set_fd = CreateSocket(PROP_SERVICE_NAME, SOCK_STREAM | SOCK_CLOEXEC | SO
CK_NONBLOCK,false, 0666, 0, 0, nullptr);
    if (property_set_fd == -1) {
        PLOG(FATAL) << "start_property_service socket creation failed";
    }
    listen(property_set_fd, 8);
    register_epoll_handler(property_set_fd, handle_property_set_fd);
```

属性服务接收到客户端请求时，会调用 handle_property_set_fd 回调函数。如果要修改系统默认行为，例如给特定只读属性提供写入功能，需要修改这里的 property_service.cpp，在 PropertySet 函数中添加对应判断、特殊处理。

8.1.2 解析 init.rc 配置文件

在 init.cpp 中,启动属性服务之后,系统就会调用 LoadBootScripts 加载并解析 init.rc。

```
start_property_service();
set_usb_controller();

    ......
LoadBootScripts(am, sm);
```

init.rc 包含五种类型语句： Action、Commands、Services、Options 和 Import，解析器有三种，分别为 ServiceParser、ActionParser、ImportParser，分别解析"service"、"on"、"import"语句。这里解析处理的重点是：如何创建 Zygote 进程。

8.1.3 启动 zygote 进程

Zygote 进程，是由 init 程序根据 init.rc 文件中描述的内容加以解析之后，通过 ServiceParser 解析而启动。

service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
class main
priority -20
user root

主要流程如下。

1）system/core/init/builtins.cpp，执行 do_class_start 函数，这里的 name 参数是 app_process（也可能是 app_prcocess32 或 app_prcocess64，取决于 rc 文件内容），

最终会启动 app_process 程序；

2）system/core/init/service.cpp，执行 StartIfNotDisabled 函数，通过 clone()或者 fork()系统调用创建一个进程，在该进程中执行 app_process 程序；

3）framework/cmds/app_process/app_main.cpp，在 main 函数中，先创建一个 AppRuntime 对象，再调用其 start 方法并传入 ZygoteInit 对应参数来启动 zygote。

```
    if (zygote) {
    runtime.start("com.android.internal.os.ZygoteInit", args, zygote);
} else if (className) {
    runtime.start("com.android.internal.os.RuntimeInit", args, zygote);
} else {
    fprintf(stderr, "Error: no class name or --zygote supplied.\n");
```

8.2 虚拟机和 Zygote 的启动

8.2.1 启动虚拟机

前面提到，启动 app_process 程序时，在 app_main.cpp 的 main 函数中，系统会创建一个 AppRuntime 对象(AndroidRuntime 的派生类)，再调用其 start 方法启动 zygote，这里注意 AppRuntime.start()方法对虚拟机的处理。

```
if (zygote) {
    runtime.start("com.android.internal.os.ZygoteInit", args, zygote);
} else if (className) {
    runtime.start("com.android.internal.os.RuntimeInit", args, zygote);
}
```

AppRuntime.start()方法，最终会调用到其父类 AndroidRuntime 的 start()，这里又会调用 startVm 函数。其功能是，配置参数并通过 JNI_CreateJavaVM 启动 art 虚拟机。JNI_CreateJavaVM 的实现在 art\runtime\java_vm_ext.cc，具体的创建包括参数解析在在

art\runtime\runtime.cc。

补充，JNIEnv*指针每个线程都有一个实例对象，并且不能互相访问，就是说线程 A 不能使用线程 B 的 JNIEnv 指针。而作为一个结构体它里面定义了 JNI 系统操作方法，包括 FindClass、GetMethodID、GetFileID、CallStaticVoidMethod 等。

在启动虚拟机之后，系统通过 JNIEnv 调用 CallStaticVoidMethod 方法来执行 ZygoteInit.main 方法。

8.2.2 启动 SystemServer

ZygoteInit.main 方法中，有个重要的环节就是预加载处理，都封装在 preload 函数里面，包括 preloadClasses、preloadResources 等类资源的预加载。

```
static void preload(TimingsTraceLog bootTimingsTraceLog) {
    Log.d(TAG, "begin preload");
    bootTimingsTraceLog.traceBegin("BeginIcuCachePinning");
    beginIcuCachePinning();
    bootTimingsTraceLog.traceEnd(); // BeginIcuCachePinning
    bootTimingsTraceLog.traceBegin("PreloadClasses");
    preloadClasses();
    bootTimingsTraceLog.traceEnd(); // PreloadClasses
    bootTimingsTraceLog.traceBegin("PreloadResources");
    preloadResources();
```

在预加载处理完成之后，forkSystemServer 方法实现了 SyetemServer 进程的启动。

```
if (startSystemServer) {
        Runnable r = forkSystemServer(abiList, socketName, zygoteServer);
```

```
        // {@code r == null} in the parent (zygote) process, and {@code r != null} in the
        // child (SystemServer) process.
        if (r != null) {
            r.run();
            return;
```

```
        }
    }
```

期间，系统还创建了一个服务端 Socket，其目的是，监听 ActivityManagerService 的创建新进程的请求，核心方法是 registerServerSocket。在 zygoteServer.runSelectLoop 中，该函数以循环方式实现了监听 AMS 的 socket 连接请求（例如创建新的 app 进程），具体是通过一个 Runnable 执行 AMS 的请求。

8.3 启动 SyetemServer 进程

8.3.1 创建 SystemServer 进程

在 ZygoteInit.main 方法中，系统先定义一个 boolean 变量 startSystemServer，并给其赋值（根据启动 Zygote 进程时，传入的参数是否为"start-system-server"），进而通过 forkSystemServer 创建 SystemServer 进程，此后的调用流程如下。

1) ZygoteInit.forkSystemServer
2) Zygote.forkSystemServer
3) Zygote.nativeForkSystemServer
4) handleSystemServerProcess
5) zygoteInit
6) nativeZygoteInit
7) RuntimeInit.applicationInit

其中，在调用 Zygote.nativeForkSystemServer 创建 SystemServer 进程之后，如果创建成功（pid 为 0），则通过内部函数 handleSystemServerProcess 处理该进程。

RuntimeInit.applicationInit 方法，最终会通过 Java 反射机制获取 SystemServer 的 main 函数入口地址，再封装为一个 Runnable 并启动。

8.3.2 启动系统服务

在 SystemServer.java 中，main 函数里面执行了各种系统服务的创建和启动，包括 BootstrapServices、核心系统服务以及其他系统服务。

在 PackageManagerService 的构造函数中，系统会初始化成员类并调用 scanDirLI 开始扫描、优化，最终通过 Installer.dexopt 调用 mInstalld.dexopt 和底层通信，完成 dex2oat 处理，将 Java 字节码转换为机器码，以提高执行效率。

8.4 启动 Launcher

8.4.1 SystemServer 中的准备工作

在 SystemServer 中，系统调用 startOtherServices 启动其他服务时，有个关键的处理：mActivityManagerService.systemReady。

在 systemReady 里面又做了若干启动阶段的准备工作，包括电源管理服务 PowerManagerInternal 、电池统计服务 BatteryStatsService 、最后调用 startHomeActivityLocked 启动 Launcher 应用。

8.4.2 AMS 启动 Launcher 应用

启动 Launcher 应用的过程比较简单，主要是先通过 getHomeIntent()方法获取对应的 Intent，再利用 resolveActivityInfo 解析之，以得到正确的 ActivityInfo。最后通过

ActivityStartController 启动对应的 Activity。

8.5 问题与思考

8.5.1、关于 Zygote 的设计

其作用之一是在开机过程中，创建虚拟机并启动 SystemServer 进程，作用之二是创建应用进程（任意 App），但这里为何要使用 Zygote 来创建、而不直接创建，意义是什么？——主要是资源复用、性能优化。因为从零开始、创建一个新的进程直到应用完全启动，其中大部分的启动过程以及资源分配管理方式是类似的，通过 Zygote 创建出来的应用进程属于 Zygote 子进程，可以减少重复创建过程、复用其资源，优化速度和内存的使用。

8.5.2 开机启动异常类问题

1、开机启动失败

case1：黑屏有背光不能正常启动

case2：黑屏无背光不能正常启动

定位手段：有背光时，获取 logcat 以及 anrlog，无背光需要查看串口 log 及 can log

可能的原因

有背光不能正常启动：可能是某个系统服务出现致命异常或者死锁，也可能是 hal 层或更底层存在卡死进而阻塞了开机流程

黑屏无背光不能正常启动：需要检查 mcu 上电情况

2、开机启动过慢

case1：某个环节 bug 导致启动流程被阻塞了一会

case2：启动过程中出现异常立即重启

定位手段：串口 log+logcat+bootchart+anr log，尽可能找到复现途径，另外，需要熟悉开机流程，在合适的地方添加打印

可能的原因

非首次开机：某个 App 或系统服务调用底层驱动出现卡死（过早调用、驱动尚未初始化完成）

首次开机：App 太多且没有使用编译时优化（systrace 或 logcat 可以看出 dex2oat 耗时太长）

3、开机时系统重启

case1：MCU 造成的重启

定位手段：检查心跳 log（sendHeart）+CAN 报文，MCU 的 log，有时候需要逻辑分析仪

可能的原因：CAN 控制器或 mcu 的 bug

补充：之前的部分版本，can 通信/驱动存在 bug 导致 can 数据不对，部分 mcu 版本存在 bug 会导致氛围灯某个设置打开时出现重启。新版本已修复

case2：Android 系统卡死

定位手段：分析 logcat 以及 anrlog，其中 logcat 最好现场复现抓取（adb 或串口），anr 日志可以后来导出

可能的原因：系统服务出现死锁导致 SystemServer 进程出现 ANR、WatchDog 触发重启

case3：kernel 层

定位手段：串口 log 或 dmesg log，查找 panic 字样且有段错误、寄存器/堆栈信息的打印

可能的原因：kernel panic 导致系统重启，内核或者驱动的 bug，一般需要方案商或原厂分析

第 9 章 ActivityManagerService

ActivityManagerService 是用于管理系统中的 Activity 和其他组件状态切换，并提供查询功能的一个系统服务，其主要工作就是管理、记录、查询，以线程方式运行在 SystemServer 系统进程中。

9.1 主要成员

9.1.1 ActivityStack

ActivityStack 负责管理当前系统中 Activity 状态，主要成员有：

- ActivityState。enum 类型，包括 INITIALIZING、RESUMED、PAUSING 等状态。
- 若干 ArrayList 变量，包括 mTaskHistory、mLRUActivities，以及一些 ActivityRecord 类型的变量 mPausingActivity、mResumedActivity 等。

ActivityState 的状态迁移，和 Activity 生命周期变化密切相关，以几个场景为例：

1）ActivityRecord 实例被创建、Activity 尚未启动时，对应的 ActivityState 为 INITIALIZING；

2）ActivityStack 中，启动 Activity 或者恢复 Activity 时，resumeTopActivityInnerLocked 函数会将待启动或待恢复的 Activity 状态设为 RESUMED；

3）ActivityStack 中，销毁 Activity 时，destroyActivityLocked 函数会将状态设为 DESTROYING 或 DESTROYED。

9.1.2 TaskRecord

TaskRecord 是用于完成某个 Activity 相关任务的“集合”。TaskRecord 和 ActivityStack 结合起来使用，会影响 Activity 的跳转和返回行为，有利于复用系统中的资源，也符合用户的逻辑思维和使用习惯。

影响 Activity 跳转和返回行为有以下两种方法。

方法 1：在<activity>标签中指定 android:taskAffinity 属性，可指定 Activity 希望归属的任务栈。

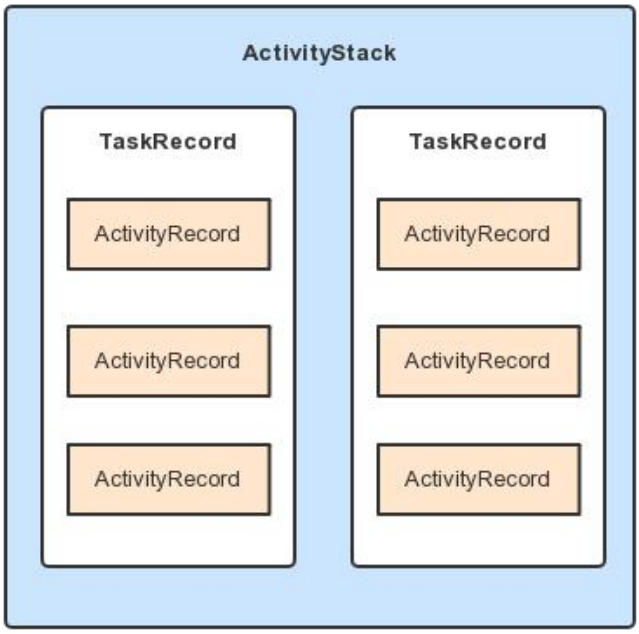
如果指定了该属性，则在启动 Activity 的 Intent 中带有 FLAG_ACTIVITY_NEW_TASK 标志时，或者将 Activity 中的 allowTaskReparenting 属性设置为 true 时，属性生效。默认情况下，同一个 App 所有的 Activity 都有着相同的 taskAffinity。

另外，设置 android:launchMode 属性也就是启动模式以及 android:clearTaskOnLaunch 属性，也会影响 Activity 跳转、返回行为。

方法 2：使用 FLAG。如 FLAG_ACTIVITY_NEW_TASK、FLAG_ACTIVITY_SINGLE_TOP、FLAG_ACTIVITY_CLEAR_TOP 等。

9.1.3 Activity 任务栈模型浅析

所谓的任务栈，是一个虚拟的概念，通过上述几个类的组合来实现。以某个 ActivityStack 为例，这里的组合与依赖关系大致如下，一个 ActivityStack 内部包含一个或者多个 TaskRecord；一个 TaskRecord 内部包含一个或多个 ActivityRecord，而 ActivityRecord 又包含了本次 Activity 启动相关的若干信息，除了 ComponentName 类型的变量 realActivity 代表本次要启动的 Activity 的组件名称，还有 launchMode、ActivityState、taskAffinity、requestCode 等。



图中 TaskRecord 在 ActivityStack 也是以后进先出方式管理的，TaskRecord 更像是“栈中栈”。

启动 Activity 时，根据启动模式的不同，系统会判断是否需要在 ActivityStack 中创建新的 TaskRecord（对应一个 Task）、是否需要创建 ActivityRecord（对应一个 Activity 实例）。

ActivityStack 有时候也需要创建不同的实例，其类型通常有如下三种：

- WindowConfiguration.ACTIVITY_TYPE_HOME;
- WindowConfiguration.ACTIVITY_TYPE_RECENTS;
- WindowConfiguration.ACTIVITY_TYPE_STANDARD;

这三种分别用于 Launcher、RecentTask 以及常规 Activity，具体可以通过 adb shell dumpsys activity activities 命令查看当前存在的 ActivityStack 及其包含的 TaskRecord 信息。

9.2 Activity 启动过程

在调用 Context.startActivity 之后，AMS 先通过 ActivityStackSupervisor 判断是否需要创建新进程，如果不需要创建，则直接调用其 realStartActivityLocked 真正启动

Activity，否则要先创建一个进程，并且等新进程的主线程创建之后，通过 attach 方法调用该函数，完成启动工作。具体可以分为下面几个阶段。

9.2.1 启动前的准备

按照调用顺序列出如下：

1) Context.startActivity

2) Activity.startActivity

3) Activity.startActivityForResult

4) Instrumentation.execStartActivity (先通过 ActivityMonitor 处理 requestCode，再调用 intent.prepareToLeaveProcess 准备离开进程，最后通过 Binder 调用 AMS)

5) Instrumentation.ActivityManager.getService().startActivity （应用进程到 SystemServer 进程的 Binder 调用）

6) ActivityManagerService.startActivityAsUser。

在 AMS 中， startActivityAsUser 方法先通过 ActivityStartController 获取 ActivityStarter 再依次设置调用者、ActivityResult 接收对象、RequestCode、启动标志等参数，然后并执行后续启动操作。

```
return mActivityStartController.obtainStarter(intent, "startActivityAsUser")
                .setCaller(caller)
                .setCallingPackage(callingPackage)
                .setResolvedType(resolvedType)
                .setResultTo(resultTo)
                .setResultWho(resultWho)
                .setRequestCode(requestCode)
                .setStartFlags(startFlags)
                .setProfilerInfo(profilerInfo)
                .setActivityOptions(bOptions)
                .setMayWait(userId)
                .execute();
```

9.2.2 目标 Activity 的匹配

目标匹配主要是在 ActivityStarter.startActivityMayWait 方法中处理的。此方法根据传入参数解析 Intent，匹配目标 Activity，先获取 ResolveInfo 再获取 ActivityInfo：

```
 ResolveInfo rInfo = mSupervisor.resolveIntent(intent, resolvedType, userId,
        0 /* matchFlags */,
              computeResolveFilterUid(
                    callingUid, realCallingUid, mRequest.filterCallingUid));
    if (rInfo == null) {
       UserInfo userInfo = mSupervisor.getUserInfo(userId);
...
```

之后再调用内部方法 startActivity，这里涉及到几个功能有所差异的重载函数，最终会调用到 startActivityUnchecked。

9.2.3 启动模式处理

启动模式包括启动相关 FLAG 的判断，涉及 ActivityStarter.startActivityUnchecked 和 ActivityStarter.computeLaunchingTaskFlags 等方法。

在 startActivityUnchecked 方法中，系统先调用 computeLaunchingTaskFlags 内部函数实现了部分启动模式的判断，如 LAUNCH_SINGLE_INSTANCE、LAUNCH_SINGLE_TASK；再通过 setTaskFromIntentActivity 函数，判断是否 LAUNCH_SINGLE_TOP 启动模式，并对应处理 Intent；其中 getReusableIntentActivity 方法，用于获取可以复用的 Activity 任务栈。注意这里调用了 ActivityStack 的 findTaskLocked 方法，用于遍历 mTaskHistory 列表并判断 taskAffinity，匹配预期的任

务栈。

在 ActivityInfo 这个类中，系统定义了几个常量：LAUNCH_MULTIPLE、LAUNCH_SINGLE_TOP、LAUNCH_SINGLE_TASK、LAUNCH_SINGLE_INSTANCE，分别对应四种启动模式。这里分析较为常见的 singleTop 和 SingleTask 模式。

1、singleTop 模式

这种启动模式，对应到 AMS 中的具体实现，包括两种情况：

1）LaunchFlag 包含 LAUNCH_SINGLE_TOP；

2）ActivityStarter.mLaunchMode 是 LAUNCH_SINGLE_TOP。

如果待启动的 Activity 满足这两种情况之一、栈顶 Activity 就是待启动 Activity 且 userId 和线程也完全匹配，则根据 mDoResume 变量判断是否需要调用 resumeFocusedStackTopActivityLocked 将目标 Activity 恢复到 RESUMED 状态。最后调用 ActivityRecord 的 deliverNewIntentLocked，该函数会利用 Binder 通信触发 Activity.onNewIntent 回调【应该在 RESUMED 状态之前？】。

```
// If the activity being launched is the same as the one currently at the top, then
        // we need to check if it should only be launched once.
        final ActivityStack topStack = mSupervisor.mFocusedStack;
        final ActivityRecord topFocused = topStack.getTopActivity();
        final ActivityRecord top = topStack.topRunningNonDelayedActivityLocked(mNotTop);
        final boolean dontStart = top != null && mStartActivity.resultTo == null
                && top.realActivity.equals(mStartActivity.realActivity)
                && top.userId == mStartActivity.userId
                && top.app != null && top.app.thread != null
                && ((mLaunchFlags & FLAG_ACTIVITY_SINGLE_TOP) != 0
                || isLaunchModeOneOf(LAUNCH_SINGLE_TOP, LAUNCH_SINGLE_TASK));
        if (dontStart) {
            // For paranoia, make sure we have correctly resumed the top activity.
            topStack.mLastPausedActivity = null;
            if (mDoResume) {
                mSupervisor.resumeFocusedStackTopActivityLocked();
            }
            ActivityOptions.abort(mOptions);
```

```
        if ((mStartFlags & START_FLAG_ONLY_IF_NEEDED) != 0) {
            // 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;
        }

        deliverNewIntent(top);
```

2、SingleTask 模式

主要分为两种情况：

1）LaunchFlag 包含 FLAG_ACTIVITY_NEW_TASK；

2）ActivityStarter.mLaunchMode 是 LAUNCH_SINGLE_TASK。

首先，在 computeLaunchingTaskFlags 函数中，针对调用者所属 Task 和待启动的 Task 加以比较，排除一些错误的启动方式，抛出对应的 IllegalArgumentException。

```
private void computeLaunchingTaskFlags() {
    // If the caller is not coming from another activity, but has given us an explicit task into
    // which they would like us to launch the new activity, then let's see about doing that.
    if (mSourceRecord == null && mInTask != null && mInTask.getStack() != null) {
        final Intent baseIntent = mInTask.getBaseIntent();
        final ActivityRecord root = mInTask.getRootActivity();
        if (baseIntent == null) {
            ActivityOptions.abort(mOptions);
            throw new IllegalArgumentException("Launching into task without base intent: "
                    + mInTask);
        }

        // If this task is empty, then we are adding the first activity -- it
        // determines the root, and must be launching as a NEW_TASK.
```

其次，在前面判断 singleTop 模式的地方，也就是给 boolean 类型变量 dontStart 赋值的逻辑中，如果待启动的 Activity 对应的 mLaunchMode 是 LAUNCH_SINGLE_TASK、栈顶 Activity 就是待启动 Activity 且 userId 和线程也完全匹配，则不会重新创建 Activity，

而是复用栈顶 Activity。

从 Activity 实例复用的角度，singleTask 模式，最主要的特点就是栈内复用，其中一种场景是栈顶复用，该场景下的处理逻辑可以参见 singleTop 模式；另外一种场景就是栈内有目标 Activity（ActivityRecord）但不在栈顶，此时会销毁目标 Activity 之上的所有 Activity，这个是通过 performClearTaskForReuseLocked 调用完成的。

```
    if ((mLaunchFlags & FLAG_ACTIVITY_CLEAR_TOP) != 0
            || isDocumentLaunchesIntoExisting(mLaunchFlags)
            || isLaunchModeOneOf(LAUNCH_SINGLE_INSTANCE, LAUNCH_SINGLE_TASK)) {
        final TaskRecord task = reusedActivity.getTask();

        // In this situation we want to remove all activities from the task up to the one
        // being started. In most cases this means we are resetting the task to its initial
        // state.
        final ActivityRecord top = task.performClearTaskForReuseLocked(mStartActivity,
                mLaunchFlags);
```

```
        // The above code can remove {@code reusedActivity} from the task, leading to the
        // the {@code ActivityRecord} removing its reference to the {@code TaskRecord}. The
        // task reference is needed in the call below to
        // {@link setTargetStackAndMoveToFrontIfNeeded}.
        if (reusedActivity.getTask() == null) {
            reusedActivity.setTask(task);
        }
```

如果目标 Activity 所在 ActivityStack 和启动者所属 ActivityStack 不同，则还需要调用 setTargetStackAndMoveToFrontIfNeeded 方法，将目标 ActivityStack 切换到前台。

在启动模式以及 TargetStack 相关处理完成之后，系统调用 ActivityStackSupervisor 的 resumeFocusedStackTopActivityLocked 方法继续启动流程。

```
    // If the target stack was not previously focusable (previous top running activity on that stack
was not visible) then any prior calls to move the stack to the will not update the focused stack.  I
f starting the new activity now allows the task stack to be focusable, then ensure that we now u
pdate the focused stack accordingly.
    if (mTargetStack.isFocusable() && !mSupervisor.isFocusedStack(mTargetStack)) {
```

```
            mTargetStack.moveToFront("startActivityUnchecked");
        }
        mSupervisor.resumeFocusedStackTopActivityLocked(mTargetStack, mStartActivity,
                mOptions);
```

9.2.4 创建进程

创建进程的处理，可以细分为下面几个步骤：

1) ActivityStackSupervisor.resumeFocusedStackTopActivityLocked；

2) ActivityStack.resumeTopActivityUncheckedLocked；

3) ActivityStackSupervisor.startSpecificActivityLocked；

startSpecificActivityLocked 中的处理，又可以分为两种情况：已有目标进程（就是说 Activity 所在应用程序已启动）、暂无目标进程。

case1：已有目标进程，则无需创建新进程，直接调用 realStartActivityLocked。

```
if (app != null && app.thread != null) {
        try {
            if ((r.info.flags&ActivityInfo.FLAG_MULTIPROCESS) == 0
                    || !"android".equals(r.info.packageName)) {
                // Don't add this if it is a platform component that is marked
                // to run in multiple processes, because this is actually
                // part of the framework so doesn't make sense to track as a
                // separate apk in the process.
                app.addPackage(r.info.packageName, r.info.applicationInfo.longVersionCode,
                        mService.mProcessStats);
            }
            realStartActivityLocked(r, app, andResume, checkConfig);
            return;
        } catch (RemoteException e) {
            Slog.w(TAG, "Exception when starting activity "
                    + r.intent.getComponent().flattenToShortString(), e);          }
```

case2：需要创建新进程，先调用 AMS 的 startProcessLocked。这里先通过 startProcess 调用 Process.start，进而通过 Zygote 创建一个新进程。

```
    mService.startProcessLocked(r.processName, r.info.applicationInfo, true, 0,
```

```
            "activity", r.intent.getComponent(), false, false, true);
```

4) ActivityManagerService.startProcessLocked。

该函数经过一些处理之后，会调用 startProcess，如下。

```
        startResult = Process.start(entryPoint,app.processName, uid, uid, gids, runtimeFlags, mo
untExternal, app.info.targetSdkVersion, seInfo, requiredAbi, instructionSet, app.info.dataDir, invo
keWith, new String[] {PROC_START_SEQ_IDENT + app.startSeq});
```

Process 的 start 方法会创建一个新的进程，同时也创建了一个名为“main”的主线程。在这个线程中，Zygote 通过反射机制，执行 ActivityThread 的 main 方法，进而调用其 attach 以及 AMS 的 attachApplication、attachApplicationLocked 将 Application 和 Binder 线程绑定起来，再调用 ActivityStackSupervisor.attachApplicationLocked 和 realStartActivityLocked 方法继续启动 Activity。

```
boolean attachApplicationLocked(ProcessRecord app) throws RemoteException {
    final String processName = app.processName;
    boolean didSomething = false;
    for (int displayNdx = mActivityDisplays.size() - 1; displayNdx >= 0; --displayNdx) {
        final ActivityDisplay display = mActivityDisplays.valueAt(displayNdx);
        for (int stackNdx = display.getChildCount() - 1; stackNdx >= 0; --stackNdx) {
            final ActivityStack stack = display.getChildAt(stackNdx);
            if (!isFocusedStack(stack)) {
                continue;
            }
            stack.getAllRunningVisibleActivitiesLocked(mTmpActivityList);
            final ActivityRecord top = stack.topRunningActivityLocked();
            final int size = mTmpActivityList.size();
            for (int i = 0; i < size; i++) {
                final ActivityRecord activity = mTmpActivityList.get(i);
                if (activity.app == null && app.uid == activity.info.applicationInfo.uid
                        && processName.equals(activity.processName)) {
                    try {
                        if (realStartActivityLocked(activity, app,
                                top == activity /* andResume */, true /* checkConfig */)) {
                            didSomething = true;
                        }
```

这里的 attach 操作，会创建一个 Binder 线程（实际上是 ApplicationThread），用作 Binder 的服务端，和 AMS 通信。如果启动超时（超过 PROC_START_TIMEOUT_MSG，Android9.0 中定义为 20s），则会出现 ANR。

9.2.5 执行生命周期回调

在进程创建之后的 realStartActivityLocked 调用中，系统利用 ClientLifecycleManager、ClientTransaction 等类，通过 Binder 跨进程调用，执行 ActivityThread.handleLaunchActivity 方法，再通过 performLaunchActivity 方法调用 Instrumentation.callActivityOnCreate，进而回调 Activity.performCreate 和 Activity.onCreate。

详细流程如下：

1) 【系统进程】ActivityStackSupervisor.realStartActivityLocked
2) 【系统进程】mService.getLifecycleManager().scheduleTransaction(clientTransaction);
3) 【系统进程】ClientLifecycleManager.scheduleTransaction
4) 【系统进程】ClientTransaction.schedule
5) 【系统进程】IApplicationThread.scheduleTransaction
6) 【应用进程】ApplicationThread.scheduleTransaction
7) 【应用进程】ClientTransactionHandler.scheduleTransaction
8) 【应用进程】mTransactionExecutor.execute
9) 【应用进程】LaunchActivityItem.execute
10) 【应用进程】ActivityThread.handleLaunchActivity
11) 【应用进程】ActivityThread.handleLaunchActivity
12) 【应用进程】performLaunchActivity
13) 【应用进程】mInstrumentation.callActivityOnCreate
14) 【应用进程】Activity.performCreate
15) 【应用进程】Activity.onCreate

Activity.onResume 的回调流程与之类似，也是通过 ClientLifecycleManager 和 ClientTransaction，基于 Binder 通信触发回调，只是添加回调 Item 的方式和 Item 对象有所不同。

9.3 Activity 销毁过程

9.3.1 正常销毁

1) Activity.finish
2) ActivityManagerService.finishActivity
3) ActivityStack.requestFinishActivityLocked
4) ActivityThread.schedulePauseActivity
5) ActivityThread.scheduleDestroyActivity

   Activity.onStop

   Activity.onDestroy
6) ActivityManagerService.broadcastIntentLocked

期间有若干判断条件，具体逻辑要结合实际情况；broadcastIntentLocked 可以用于某 app 退出客制化。

9.3.2 异常销毁

异常销毁主要有两种情况。一种是配置改变例如横竖屏切换导致 Activity 被销毁而后重新创建。还有一种就是系统后台强制杀进程，例如内存不足，此时栈里面的第一个没有销毁的 activity 会执行 onDestroy 方法，其他的不会执行。

从 Activity 生命周期角度来看，在配置改变时，系统会先调用 onSaveInstanceState(Bundle)以保存当前状态 ，再依次回调 onPause、onStop 以及 onDestory，此后调用 onCreate(Bundle)、onRestoreInstanceState(Bundle)以重新创建 Activity 并恢复状态。

9.4 问题与思考

9.4.1 Activity 任务栈

1、什么情况下需要创建一个新的 ActivityStack?

分析：启动模式为 singleTask 时，如果栈内没有目标 Activity 实例，则需要创建一个新的 TaskRecord，并在这个 TaskRecord 中创建一个新的 ActivityRecord 实例。但这里是否会创建一个 ActivityStack，已有资料未见提及。

根据实验可以判断，用户从 Launcher 启动一个应用时，这里会创建一个新的 ActivityStack 以及 TaskRecord 实例，之后从该应用启动其他 Activity，则可能会创建新的 TaskRecord 或者 ActivityRecord（取决于启动模式以及 taskAffinity 等属性）。

2、是否每个 TaskRecord 中有且只有一个处于栈顶的 ActivityRecord?

分析：每个 TaskRecord 中有且只有一个处于栈顶的 ActivityRecord，但这个 ActivityRecord 所对应的 Activity 未必具有焦点。

系统通常会存在多个 ActivityStack，每个时刻只有一个 ActivityStack 具有焦点，也就是 ActivityStackSupervisor.mFocusedStack，该 ActivityStack 中，前台 TaskRecord 的栈顶 Activity 具有焦点。

如果使用 adb shell dumpsys activity activities 命令打印，输出结果中的第一个 ActivityStack 就是 mFocusedStack，该 Stack 中的第一个 TaskRecord 的栈顶 Activity 就是当前焦点 Activity。

3、前面提到的栈内复用、栈中复用，“栈”是指 TaskRecord 还是 ActivityStack?

分析：任务栈包括 TaskRecord、ActivityStack。例如单纯 singleTop 模式（不使用其他 Flag 和 taskAffinity），只在当前任务栈也就是调用者所属的 TaskRecord 中判断栈顶 ActivityRecord 是否对应目标 Activity，这时候不考虑其他 TaskRecord 更不考虑其他 ActivityStack。

默认的 singleTask 模式，会在所有标准类型的 ActivityStack 中（不局限于当前 TaskRecord）判断有无目标 ActivityRecord，如果没有，再创建新的 ActivityRecord。

这里可以和前面组件部分关于 Activity 启动模式的分析结合起来考虑。singleTask 模式下，如果我们没有设置 taskAffinity，从本应用启动的 Activity 就会复用当前 TaskRecord，不会创建新的 TaskRecord 实例。是否创建一个新的 ActivityStack 要取决于调用者，如果是 Launcher 则会创建一个新的 ActivityStack。

9.4.2 新启动应用时的进程创建

1、新进程的创建

在 Activity 启动流程分析部分提到，如果是新启动的应用，系统会通过 AMS 的 startProcessLocked 先创建一个新的进程。这部分的处理流程， 会用到前面开机启动过程（8.2.2 节）所提到的 Server 端的 Socket。

2、涉及的进程

以一个典型的场景为例，开机后，从 Launcher 点击 Camera 应用图标，也就是首次启动相机应用。完成应用启动，涉及到的进程执行顺序为：

Launcher 进程->SystemServer 进程->Zygote 进程->Camera 进程。

9.4.3 Activity 状态迁移机制

1、Activity 启动时的状态迁移

【目标 Activity】INITIALIZING

->【当前顶层 Activity】PAUSING to PAUSED

->【目标 Activity】RESUMED

典型 log 之一：

InCallActivity t-1} from:null to:INITIALIZING reason:ActivityRecord ctor
SystemSettingsActivity t36} from:RESUMED to:PAUSING reason:startPausingLocked
SystemSettingsActivity t36} from:PAUSING to:PAUSED reason:completePausedLocked
InCallActivity t82} from:INITIALIZING to:RESUMED reason:minimalResumeActivityLocked

备注：从系统设置窗口启动通话窗口。ActivityRecord ctor 的意思是，ActivityRecord 的构造函数设置的状态。

典型 log 之二：

SystemSettingsActivity t-1} from:null to:INITIALIZING reason:ActivityRecord ctor
CommonWidgetActivity t312} from:RESUMED to:PAUSING reason:startPausingLocked
CommonWidgetActivity t312} from:PAUSING to:PAUSED reason:completePausedLocked
SystemSettingsActivity t313} from:INITIALIZING to:RESUMED
reason:minimalResumeActivityLocked
CommonWidgetActivity t312} from:PAUSED to:STOPPING reason:stopActivityLocked
CommonWidgetActivity t312} from:STOPPING to:STOPPED reason:activityStoppedLocked

2、Activity 销毁时的状态迁移

【目标 Activity】RESUMED->PAUSING

->PAUSED->FINISHING->DESTROYING

->【之前 Activity】PAUSED to RESUMED

->【目标 Activity】DESTROYING to DESTROYED

典型 log 之一：

InCallActivity t82 f} from:RESUMED to:PAUSING reason:startPausingLocked
InCallActivity t82 f} from:PAUSING to:PAUSED reason:completePausedLocked
InCallActivity t82 f} from:PAUSED to:FINISHING reason:finishCurrentActivityLocked
InCallActivity t82 f} from:FINISHING to:DESTROYING reason:destroyActivityLocked. finishing and not skipping destroy
SystemSettingsActivity t36} from:PAUSED to:RESUMED
reason:resumeTopActivityInnerLocked
InCallActivity t82 f} from:DESTROYING to:DESTROYED
reason:removeActivityFromHistoryLocked

典型 log 之二：

SystemSettingsActivity t313 f} from:RESUMED to:PAUSING reason:startPausingLocked
SystemSettingsActivity t313 f} from:PAUSING to:PAUSED reason:completePausedLocked

SystemSettingsActivity t313 f} from:PAUSED to:STOPPING
reason:finishCurrentActivityLocked

注意这里的执行有 STOPPING 状态而上面没有，为什么?

首先，在 ActivityStack 的 finishCurrentActivityLocked，会判断当前 mode 如果是 FINISH_AFTER_VISIBLE 且待销毁的 ActivityRecord 对应窗口可见、且下个（待恢复）的 Activity 不为空且不可见，则设置待销毁的 Activity 状态为 STOPPING。否则直接设置 FINISHING。

其次，上面 InCallActivity 背后的 Activity 是 SystemSettingsActivity，这个 Activity 处于可见状态，InCallActivity 以一个小窗口的方式显示，只遮挡了系统设置的部分区域。

再次，下面的 log，待恢复的 Activity 也就是 CommonWidgetActivity，处于不可见状态，因为 SystemSettingsActivity 全屏方式显示。

CommonWidgetActivity t312} from:STOPPED to:RESUMED
reason:resumeTopActivityInnerLocked
SystemSettingsActivity t313 f} from:STOPPING to:FINISHING
reason:finishCurrentActivityLocked
SystemSettingsActivity t313 f} from:FINISHING to:DESTROYING
reason:destroyActivityLocked. finishing and not skipping destroy
SystemSettingsActivity t313 f} from:DESTROYING to:DESTROYED
reason:removeActivityFromHistoryLocked

第 10 章 InputManagerService

10.1 输入事件类型

从设备角度划分，包括按键事件、触摸事件、鼠标事件以及其他事件；在 Java 框架层，较为常见的输入事件对应的 InputEvent，可以分为 KeyEvent、MotionEvent。

更为全面的分类，可以参考 EventHub.h 里面定义的：

```
enum {
    /* The input device is a keyboard or has buttons. */
    INPUT_DEVICE_CLASS_KEYBOARD     = 0x00000001,

    /* The input device is an alpha-numeric keyboard (not just a dial pad). */
    INPUT_DEVICE_CLASS_ALPHAKEY     = 0x00000002,

    /* The input device is a touchscreen or a touchpad (either single-touch or multi-touch). */
    INPUT_DEVICE_CLASS_TOUCH        = 0x00000004,

    /* The input device is a cursor device such as a trackball or mouse. */
    INPUT_DEVICE_CLASS_CURSOR       = 0x00000008,

    /* The input device is a multi-touch touchscreen. */
    INPUT_DEVICE_CLASS_TOUCH_MT     = 0x00000010,

    /* The input device is a directional pad (implies keyboard, has DPAD keys). */
    INPUT_DEVICE_CLASS_DPAD         = 0x00000020,

    /* The input device is a gamepad (implies keyboard, has BUTTON keys). */
    INPUT_DEVICE_CLASS_GAMEPAD      = 0x00000040,
```

这些枚举常量主要用于 Device 数据中的 classes 属性：

```
struct Device {
    Device* next;

    int fd; // may be -1 if device is closed
    const int32_t id;
    const String8 path;
    const InputDeviceIdentifier identifier;
```

```
    uint32_t classes;
```

```
    uint8_t keyBitmask[(KEY_MAX + 1) / 8];
    uint8_t absBitmask[(ABS_MAX + 1) / 8];
    uint8_t relBitmask[(REL_MAX + 1) / 8];
    uint8_t swBitmask[(SW_MAX + 1) / 8];
    uint8_t ledBitmask[(LED_MAX + 1) / 8];
    uint8_t ffBitmask[(FF_MAX + 1) / 8];
    uint8_t propBitmask[(INPUT_PROP_MAX + 1) / 8];
```

该属性在 EventHub::openDeviceLocked 方法中赋值，例如触摸型的输入设备：

```
} else if (test_bit(BTN_TOUCH, device->keyBitmask)
        && test_bit(ABS_X, device->absBitmask)
        && test_bit(ABS_Y, device->absBitmask)) {
    device->classes |= INPUT_DEVICE_CLASS_TOUCH;
// Is this a BT stylus?
```

赋值之后，创建具体输入设备时，还要用到这些属性，具体实现在 InputReader::createDeviceLocked 方法中。

```
 if (keyboardSource != 0) {
    device->addMapper(new KeyboardInputMapper(device, keyboardSource, keyboardType));
}

// Cursor-like devices.
if (classes & INPUT_DEVICE_CLASS_CURSOR) {
    device->addMapper(new CursorInputMapper(device));
}
```

```
// Touchscreens and touchpad devices.
if (classes & INPUT_DEVICE_CLASS_TOUCH_MT) {
    device->addMapper(new MultiTouchInputMapper(device));
} else if (classes & INPUT_DEVICE_CLASS_TOUCH) {
    device->addMapper(new SingleTouchInputMapper(device));
}
```

这些不同的输入设备，对应不同的 InputMapper，包括 KeyboardInputMapper、

TouchInputMapper、RotaryEncoderInputMapper、JoystickInputMapper 等。

10.2 主要成员

10.2.1 InputManager

InputManager 是 InputManagerService（简称 IMS）在 Native 层的实现，其功能主要是：

- 提供 initialize 方法以创建 InputReaderThread、InputDispatcherThread 线程实例；
- 提供 start 和 stop 方法以启动、停止这两个线程；
- 提供获取两个线程实例的方法。

在 Java 层的 InputManagerService 类中，系统通过 nativeInit、nativeStart 等函数调用 native 层 InputManager 提供的上述方法，完成 Input 子系统初始化工作。

10.2.2 InputReaderThread

这个线程类，功能是持续地轮询相关设备节点是否有新的事件发生，核心的功能实现在 InputReader 类中。这里对设备节点的访问，是通过 EventHub 来完成的。

EventHub 通过读取/dev/input/下的相关文件来判断是否有新事件（包括设备增删事件和原始输入事件这两种 type），再转发事件通知。

不管是哪种 InputMapper，在转发过程中都是通过 QueuedInputListener::notifyKey 或 notifyMotion 将事件添加到 QueuedInputListener.mArgsQueue 队列，最后调用 Listener 的 flush 函数传递给 InputDispatcher。其 flush 方法中，通过 NotifyArgs::notify 由具体对象回调 InputDispatcher 的 notifyKey 或 notifyMotion。

InputReader 的 loopOnce 函数，会被 InputReaderThread 线程循环调用（这是由于

threadLoop 处于一个更高层的循环中），而 InputDispatcher 对应 mQueuedListener，是在 InputManager 中创建 InputReader 实例时建立关联的。

10.2.3 InputDispatcherThread

这个线程类的功能是分发输入事件，其实现核心是 InputDispatcher 类。 分发输入事件过程中，有个关键的子过程，就是查找与当前按键事件匹配的目标。

在 InputDispatcher::findFocusedWindowTargetsLocked 中，InputDispatcher 根据 mFocusedWindowHandle 获取有焦点的目标（这个是由 InputMonitor 提供的），之后通过 InputChannel 通知应用程序窗口有按键事件。

InputDispatcher 的 dispatchOnce 函数，会被 InputDispatcherThread 线程循环调用。InputDispatcherThread 有个 Looper 类型的变量 mLooper，会维护一个死循环，在收到特定事件时会触发回调 handleReceiveCallback。mLooper 和事件以及回调的关联是在 registerInputChannel 里面完成的，功能之一是处理应用进程发来的输入 Finish 事件（也就是完成了输入事件的消费）。

10.2.4 InputMonitor

与前面几个 native 成员类不同，这是一个 Java 框架类，作为 WindowManagerService 窗口管理子系统与 InputManagerService 子系统之间的桥梁。InputMonitor 类实现了 InputManagerService.WindowManagerCallbacks 接口。该接口有几个重点方法：notifyANR、interceptKeyBeforeQueueing。

InputMonitor 的 interceptKeyBeforeQueueing 接口实现，最终会调用到 PhoneWindowManager 的 interceptKeyBeforeQueueing，可用于实现按键或触摸事件

拦截相关客制化修改。

10.3 事件分发/传递流程

10.3.1 初始化

如前所述，在 systemServer 创建 InputManagerService 实例时，系统通过 nativeInit、nativeStart 等 jni 调用 InputManager 提供的初始化方法，完成 Input 子系统初始化工作。具体包括创建 InputManager 实例、创建 InputReaderThread、InputDispatcherThread 实例，以及启动 InputReaderThread 和 InputDispatcherThread。

10.3.2 读取输入事件

这部分主要是又 InputReader 在 loopOnce 方法中，通过 EventHub 读取/dev/input/下的相关文件。InputReaderTread 启动后，其 threadLoop 会循环调用 loopOnce 函数。

重点是，InputReader 根据事件类型，通过对应的 InputMapper 实例、QueuedInputListener，将事件转发给 InputDispatcher，再调用 QueuedInputListener::notifyKey 或 notifyMotion，以及 QueuedInputListener::flush 方法，触发 InputDispatcher 对应的回调（notifyKey 或 notifyMotion）。

10.3.3 输入事件预处理

这部分流程重点是，InputDispatcher 根据事件类型，校验事件有无问题，然后调用拦截接口 interceptKeyBeforeQueueing，再通过 enqueueInboundEventLocked 方法将事件添加到 mInboundQueue 队列尾部，准备分发。

10.3.4 分发输入事件

重点流程：InputDispatcher 通过线程循环，从 mInboundQueue 队列头部提取事件，然后根据事件类型调用 dispatchKeyLocked 或 dispatchMotionLocked，将事件分发到目标窗口。

详细步骤包括：

1) InputDispatcher
2) dispatchOnce（InputDispatcher 线程启动后，其 threadLoop 会循环调用该函数）
3) dispatchOnceInnerLocked
4) dispatchKeyLocked 或 dispatchMotionLocked
5) findFocusedWindowTargetsLocked【补充：该过程会判断是否等待超过 5s 来决定是否调用 onANRLocked()】
6) dispatchEventLocked
7) InputPublisher::publishKeyevent
8) InputChannel::sendMessage
9) NativeInputEventReceiver::handleEvent

10.3.5 消费输入事件

这部分的处理逻辑大致可以分为两部分，首先是事件从 InputEventReceiver 到 ViewRootImpl 的分发。

1) NativeInputEventReceiver::consumeEvents
2) InputEventReceiver.dispatchInputEvent

3) ViewRootImpl.WindowInputEventReceiver.onInputEvent

4) ViewRootImpl.WindowInputEventReceiver.enqueueInputEvent

5) ViewRootImpl.WindowInputEventReceiver.doProcessInputEvents

6) ViewRootImpl.WindowInputEventReceiver.deliverInputEvent

7) ViewRootImpl.InputStage.deliver

8) ViewRootImpl.ViewPostImeInputStage.onProcess

在这之后，系统根据 QueuedInputEvent 的 mEvent 实例类型以及 source 信息，分别调用 processKeyEvent、processPointerEvent 等几种事件处理函数之一，例如按键事件调用 processKeyEvent、触摸事件调用 processPointerEvent。以触摸事件为例，第二部分的事件消费逻辑如下。

9) ViewRootImpl.processPointerEvent

10) View.dispatchPointerEvent

11) DecorView.dispatchTouchEvent

12) WindowCallbackWrapper.dispatchTouchEvent

13) Activity.dispatchTouchEvent

14) DecorView.superDispatchTouchEvent

15) ViewGroup.dispatchTouchEvent

16) Activity.onTouchEvent，该方法仅当事件没有被 ViewGroup 消费时（返回 false）才会执行。

10.3.6 输入事件的 ANR 超时

主线程耗时导致 input ANR 触发的主要原因是 waitQueue 没有及时复位。如果 UI 线

程阻塞或者超时，这里的 checkWindowReadyForMoreInputLocked 就会触发 handleTargetsNotReadyLocked，之后调用 onANRLocked。实际超时场景还有其他几种。

主要流程如下。

1）在应用处理完毕（消费事件）之后，会通过 InputConsumer 给 InputDispatcher 对象发送对应的信号/通知；

2）InputDispatcher 回调 handleReceiveCallback 方法，进而调用 doDispatchCycleFinishedLockedInterruptible 将本次处理的事件从 waitQueue 里面移除；

3）开始下一轮的分发处理。在 InputDispatcher 分发事件时，系统会执行 findFocusedWindowTargetsLocked、checkWindowReadyForMoreInputLocked 检查当前窗口是否准备好。具体分为 3 种情况。

case1：按键事件，outboundQueue 不为空或者 waitQueue 不为空，返回对应的 StringPrintf: “Waiting to send key event because...”

case2：触摸事件，waitQueue 不为空且超过 5s，返回对应的 StringPrintf: “Waiting to send non-key event because...”

case3：当前窗口已准备好，返回空字符串。

10.5 问题与思考

10.5.1 输入事件分发和 AMS

平时在使用过程中，如果出现 ANR，往往会发现系统弹出一个“xx 无响应”的提示窗

口，这个是在 AMS 中处理的，相关函数为 inputDispatchingTimedOut。

该函数会先做一些判断，然后调用 mAppErrors.appNotResponding 继续 ANR 处理，例如通过 isSilentANR 区分当前 ANR 是否 silentANR，必要时给发生 ANR 的进程发送 SIGNAL_QUIT 信号。如果不是 silentANR 则会给 AMS 中的 mUiHandler 对象发送 SHOW_NOT_RESPONDING_UI_MSG 消息，进而弹出提示窗口。

10.5.2 输入事件的扩展

1、新增按键输入

重点是增加新的键值定义，以及需要处理这个键值的地方， 修改涉及：

1）在 frameworks/base/api/current.txt 中添加键值定义；

2）在 KeyEvent.java 中添加同样的键值定义；

3）在 frameworks/base/core/res/res/values/attrs.xm 添加 enum 定义；

4）修改 PhoneWindowManager.java，添加对应的处理，如果需要转给应用层，注意转发逻辑；

5）frameworks/native/include/android/keycodes.h，添加 enum 定义；

6）/frameworks/native/libs/input/Input.cpp，hasDefaultAction 函数中添加 case 分支，使其返回 true，可以转发；

7）修改 kernel 层的.h 文件，避免使用 magic number。

2、新增旋钮输入

对照 Native 层 EventHub.h 中的定义，旋钮一般属于 INPUT_DEVICE_CLASS_ROTARY_ENCODER 这种类型。所以需要关注这种类型的输入事

件，在 kernel 层上报时，type 属性的设置，以及 value 值的传递。

其次是，Native Framework 中的 INPUT_DEVICE_CLASS_ROTARY_ENCODER 类型事件的预处理和转发流程。

该类型的输入事件，通过 RotaryEncoderInputMapper::sync、QueuedInputListener::notifyMotion 调用 InputDispatcher::notifyMotion，后续处理逻辑使用系统已实现的 NativeInputEventSender::sendMotionEvent，之后进入 Java 框架层，通过 ViewRootImpl.ViewPostImeInputStage.onProcess，调用 processGenericMotionEvent：

```
final class ViewPostImeInputStage extends InputStage {
    public ViewPostImeInputStage(InputStage next) {
        super(next);
    }

    @Override
    protected int onProcess(QueuedInputEvent q) {
        if (q.mEvent instanceof KeyEvent) {
            return processKeyEvent(q);
        } else {
            final int source = q.mEvent.getSource();
            if ((source & InputDevice.SOURCE_CLASS_POINTER) != 0) {
                return processPointerEvent(q);
            } else if ((source & InputDevice.SOURCE_CLASS_TRACKBALL) != 0) {
                return processTrackballEvent(q);
            } else {
                return processGenericMotionEvent(q);
            }
        }
```

之后回调 View.onGenericMotionEvent，应用程序可以自定义 View，重写该方法；也可以在 Activity 中实现 OnGenericMotionListener 接口，并重写该接口的 onGenericMotion 方法。

整体上，如果是旋钮类设备的输入事件（不考虑旋转+按钮的复合类型），使用系统预设框架基本就可以满足要求，不过需要注意参数的传递，以及 kernel 层与 EventHub 之间的适配。

第 11 章 WindowManagerService

11.1 基本功能与静态结构

WindowManagerService 基本功能有窗口管理、输入事件处理以及窗口显示功能。具体包括窗口的创建、删除、布局与 Z 序计算、窗口动画管理等，其中窗口的输入事件处理，依赖 InputMonitor 辅助 InputManagerService 完成，窗口显示功能则依赖 SurfaceSession 辅助 SurfaceFlinger 完成。

核心类与接口：

- WindowManager 接口，派生于 ViewManager 接口，主要方法有 getDefaultDisplay、removeViewImmediate 等；
- WindowManagerService，派生于 IWindowManager.Stub，核心成员有 WindowState、WindowManagerPolicy、WindowAnimator 等，实现窗口管理服务的大部分核心功能；
- WindowManagerImpl，实现了 WindowManager 接口，核心成员有 WindowManagerGlobal、Window 等，提供 addView、removeView、updateViewLayout 等方法调用，具体功能是通过 WindowManagerGlobal 实现的。
- PhoneWindow，派生于 Window，实现了 setContentView、setTitle、getDecorView、installDecor 等功能。
- WindowManagerPolicy 接口，其实现类是 PhoneWindowManager，可用于拦截各类输入事件，实现按键、触摸事件的客制化处理。

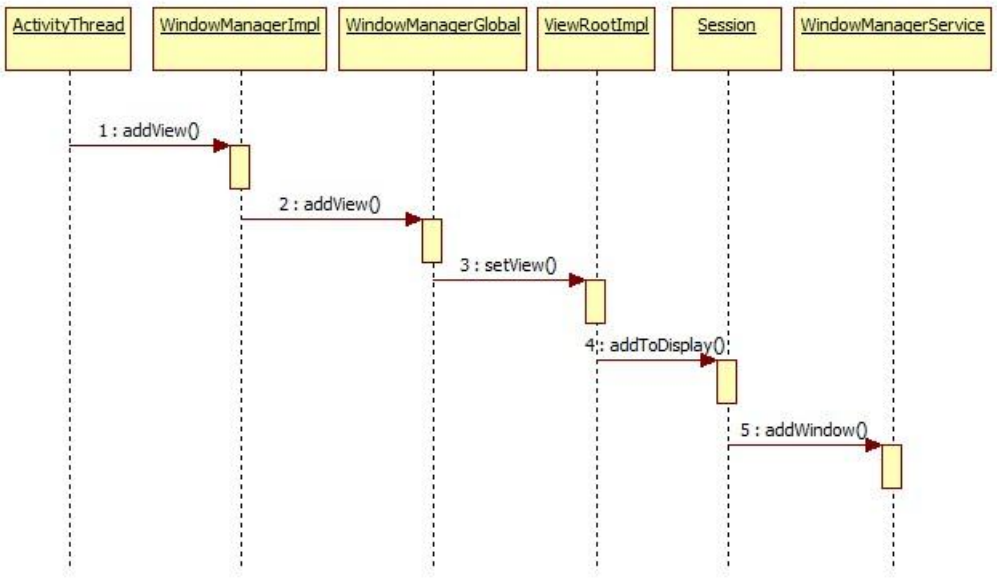
11.2 主要流程

11.2.1 窗口添加流程

1、概述

这里主要分析应用启动时的窗口添加过程。涉及 ActivityThread、WindowManagerImpl、WindowManagerGlobal、ViewRootImpl、Session 以及 WindowManagerService 等类，其中关键的调用是 addView 方法，通过 ViewRootImpl、Session，最终会调用到 WMS 的 addWindow 方法。

2、具体流程



1) ActivityThread.handleResumeActivity

2) WindowManagerImpl.addView

3) WindowManagerGlobal.addView

4) ViewRootImpl.setView

5) Session.addToDisplay

6) WindowManagerService.addWindow。

在这之后，WMS 首先创建 WindowState 实例，再通过 WindowManagerPolicy 调整窗口布局参数，调用 PhoneWindowManager.prepareAddWindowLw 方法处理 StatusBar、NavigationBar。

由于窗口和输入事件处理密切相关，这里还会调用 WindowState 的 openInputChannel 方法来注册 InputChannel，最后调用 WindowState.attach，经过一系列的处理，创建 SurfaceSession 实例。

SurfaceSession 是当前窗口和 SurfaceFlinger 通信的桥梁，可以用来创建 Surface。其中的关键，是 native 层的 SurfaceComposerClient。在 WindowState 类中，其成员 mSession 的 mSurfaceSession.mNativeClient 变量就是 SurfaceComposerClient 类型。

11.2.2 窗口销毁流程

1、概述

与窗口添加流程类似，也会涉及 ActivityThread、WindowManagerImpl、WindowManagerGlobal、ViewRootImpl、Session 以及 WindowManagerService、WindoState 等类，其中关键的调用是 removeView 方法，通过 ViewRootImpl、Session，最终会调用到 WindowState 的 removeIfPossible 方法。

2、具体流程

1) ActivityThread.handleDestroyActivity

2) WindowManagerImpl.removeViewImmediate

3) WindowManagerGlobal.removeView

4) ViewRootImpl.doDie

5) WindowManagerGlobal.doRemoveView

6) Session.remove

7) WindowManagerService.removeWindow

8) WindowState.removeIfPossible

这里的 removeIfPossible 方法，并不是直接执行删除操作，而是进行多个条件判断过滤，满足其中一个条件就推迟删除操作。典型场景是播放窗口动画（播放结束时才会调用 removeImmediately）。

在删除窗口时，会检查删除线程的正确性，如果正确才会继续后面流程，包括：从 ViewRootImpl 列表、布局参数列表和 View 列表中删除与当前窗口关联的 ViewRootImpl 所对应的元素。，清理相关资源。

11.2.4 窗口切换流程

1、概述

WMS 服务在执行 Activity 窗口的切换操作的时候，会给参与切换操作的 Activity 组件的设置一个动画，以便可以向用户展现切换效果，从而提高用户体验

这里除了设置切换动画，还有窗口进入动画。如果该窗口是附加在其他窗口且这个被附加的窗口正在显示动画，那么这个动画也同时会被设置到该窗口（3 种类型）。

2、主要流程

1) ActivityStack.startActivityLocked

2) WindowManagerService.prepareAppTransition

3) ActivityStack.ensureActivitiesVisibleLocked

4) WindowManagerService.executeAppTransition

5) WindowManagerService.performSurfacePlacement

6) WindowSurfacePlacer.performSurfacePlacement

7) RootWindowContainer.performSurfacePlacement

8) DisplayContent.applySurfaceChangesTransaction

其中 DisplayContent 的 mApplySurfaceChangesTransaction 回调函数（实现了 Consumer 接口），先调用 WindowStateAnimator.commitFinishDrawingLocked，进而根据 mAppToken 或窗口属性判断是否调用 WindowState.performShowLocked。mUpdateWindowsForAnimator 回调函数，如果窗口动画处于 READY_TO_SHOW 且 mAppToken 满足条件则调用 WindowState.performShowLocked。

11.3 问题与思考

11.3.1 Activity-AMS-WMS 之间关系

1、Activity

Activity 通过 IActivityManager 访问 AMS。在应用进程，Activity 通过 ActivityManagerNative 的 getDefault 静态方法获取到 IActivityManager 实例对象，进

而通过其 Server 端的实现是 ActivityManagerService 类。

Activity 通过 IWindowSession 访问 WMS。IWindowSession 在远程 Server 端（WMS）中的实现是 Session 类，实例为 WindowState 的成员变量 mSession。Activity 通过其 PhoneWindow 类型的成员变量 mWindow 以及 ViewRootImpl 访问 IWindowSession，从而间接访问 WMS。

2、ActivityManagerService

AMS 通过其 WindowManagerService 类型的成员变量 mWindowManager 访问 WMS。通过 ActivityStackSupervisor 类型的 mStackSupervisor 成员变量及其 ActivityStack 成员，访问 Activity 对象。

3、WindowManagerService

WMS 通过 IWindow 访问 Activity，IWindow 在客户端的实现是 ViewRootImpl.W 类，而 ViewRootImpl 和 Activity 密切关联，这样 WMS 可以间接访问 Activity。

WMS 通过 IActivityManager 访问 AMS。如前所述，IActivityManager 在 Server 端的实现是 ActivityManagerService 类。这里 IActivityManager 实例，具体是通过 WMS 的成员变量 mActivityManager 的 getService 方法获取的。

（WMS 和 AMS 运行在一个进程的不同线程，为何还需要通过 IActivityManager 基于 Binder 调用？AMS 可以通过成员变量直接访问，WMS 理论上也可以通过成员变量直接访问 AMS，只需要 SystemServer 做好设置）

11.3.2 WindowToken 和 WindowState

1、WindowToken

功能是将属于同一个应用组件的窗口组织在一起，包括 Activity、InputMethod、Wallpaper 等。该类派生于 WindowContainer<WindowState>类，用 WindowToken 指代一个应用组件，而对于 WMS 的客户端来说，Token 仅仅是个 Binder 对象。

2、AppWindowToken

派生于 WindowToken 类，实现了 WMS 的 AppFreezeListener 接口。在 Activity 启动时，AMS 会调用 WMS 的 addAppToken 方法来为它增加一个 AppWindowToken。注意，应用程序进程中的每个 Activity 组件，在 AMS 中都有个对应的 ActivityRecord 对象，在 WMS 中都有个 AppWindowToken 对象。

3、WindowState

WMS 的 addWindow 方法，可创建对应的 WindowState 实例。AppWindowToken、WindowToken，都用来描述一组有着相同令牌的窗口。一个 WindowState 实例，要么与 AppWindowToken 实例对应，要么与 WindowToken 实例对应。

问题：为什么 Acivity 或者 Dialog 封装了 PhoneWindow 对象，而 PopupWindow 或 Toast 不使用 PhoneWindow 对象?

第 12 章 图形渲染

12.1 基础概念与相关组件

概念名称	功能描述	备注
View	视图组件，待绘制的内容。	
Surface	基本的图形绘制单元，类似画纸功能。每个 Surface 都有一个 Canvas 对象，封装了底层 Skia 引擎提供的 2D 绘制接口。	
Graphic Buffer	绘制数据缓存队列，有 3 块，类似画板功能。三缓冲机制，是在双缓冲机制基础上增加了一个 Graphic Buffer，利用空闲时间，提高渲染效率。	某些场景下，Display、CPU、GPU 分别使用一块缓冲区，可以避免空闲等待
图形引擎	常见的引擎有 Skia、OpenGL ES，类似画笔功能，分别绘制 2D、2D/3D。其中	硬件加速需要使用 OpenGL ES，通过 GPU 完成计算和渲染处理
Canvas	封装了 Skia 的图形绘制接口，提供应用或 Java 框架层调用。	Java 层的 Canvas 类，native 层有对应的实现
SurfaceSession	窗口到 SurfaceFlinger 的连接会话，在这个会话里，系统可以创建一个或多个 Surface，最后合成，送到屏幕上显示。	
SurfaceView	View 的子类，提供了一个专门用于绘图的 Surface（有对应的专用 WindowState），一般用于视频播放、相机预览。绘制效率高，但不支持几何变换（平移、缩放、旋转等）。	其渲染放在单独的线程。 子类有：GLSurfaceView、RSSurfaceView、VideoView
TextureView	与 SurfaceView 一样继承于 View，可以将内容流直接投影到 View 中，支持几何变换。	需要硬件加速，占用内存比 SurfaceView 高
SurfaceTexture	将图像流转为 GL 外部纹理，可用于图像流数据的二次处理（如 Camera 滤镜，桌面特效等），一般和 GLSurfaceView 或 TextureView 结合使用。	用于 Camera 预览时，需要创建一个对应的 Surface，再通过 CaptureRequest 将其设置为图像输出 Target。

12.2 主要流程

首先，VSync 同步信号通知 CPU 准备待渲染的数据，然后图形引擎将数据按特定方式渲染到缓冲区（如果开启硬件加速方式则会使用 GPU），再由 SurfaceFlinger 汇总所有图层，合成显示内容并输出到屏幕上。

详细流程如下。

Step1、ViewRootImpl.scheduleTraversals

Step2、ViewRootImpl.doTraversal

这里 scheduleTraversals 只是添加 callback、启动调度功能，后面的 doTraversal 遍历操作是由 Choreographer 收到 VSync 信号进而回调这里的 callback 触发的。如果要取消这里的调度功能（不自动触发遍历），可以调用 unscheduleTraversals 方法，例如 WMS 在添加窗口时就通过 setView 调用了该方法。

Step3、ViewRootImpl.performTraversals

遍历操作的具体实现，可以分为三步。

1）、performMeasure，测量尺寸，回调 View.onMeasure。

2）、performLayout，计算视图/组件位置布局，回调 View.onLayout。严格意义上，完整的布局过程包括两部分调用： measure()+layout()，对应 View.onMeasure()和 View.onLayout()。

3）、performDraw，绘制处理，包括 draw、View.onDraw、updateRootDisplayList、nSetFrameCallback 等调用。如果是硬件绘制，会通过 mAttachInfo 成员变量调用其 mThreadedRenderer.draw 方法，而软件绘制则直接调用 drawSoftware 方法。这里遍历子 View 进行绘制，采用了深度优先的方式。

Step4、ViewRootImpl.drawSoftware，负责软件绘制工作，包括通过 mSurface.lockCanvas 获取画布、在 natvie 层创建对应的 Canvas，向 SurfaceFlinger 获取一个 GraphicBuffer；通过 mTranslator.translateCanvas 实现坐标变换，包括平移、缩放等处理（具体是在 native 层利用 matrix 实现的），再调用 mView.draw 从顶层 View 开

始，将数据绘制到 Canvas 上。最后执行 mSurface.unlockCanvasAndPost 提交绘制操作。

Step5、View.draw，这里又可以分为 5 个子步骤：

1）、绘制背景：drawBackground。

2）绘制内容：onDraw，该回调的具体实现有多种场景，例如：

case1：调用 Layout.onDraw 例如 LinearLayout.onDraw；

case2：调用 ViewGroup.onDraw（实际上 Layout 往往是 ViewGroup 派生类）；

case3：调用各种具体 View 例如 TextView 的 onDraw 方法。

3）绘制子 View：dispatchDraw。

4）绘制其他装饰部分（例如滚动条）：onDrawForeground。

5）绘制焦点高亮部分：drawDefaultFocusHighlight。

补充：Step2 和 Step5 分别用于特定场景下保存图层和恢复图层。

——前面的流程，最开始是 ViewRootImpl.scheduleTraversals，这个函数是如何触发执行的？分为两种情况。

如果是系统收到周期性的 VSync 信号这类外部事件触发，流程如下：

1）Choreographer 收到 VSync 信号之后通知主线程执行 doFrame

2）在 doFrame 方法中通过 doCallBack 回调不同类型的渲染处理，包括 input、动画以及视图绘制遍历等，这些 callback 由各个组件自己实现。这里的视图遍历 callback（CALLBACK_TRAVERSAL 类型）的添加，是在 ViewRootImpl.scheduleTraversals 中调用 mChoreographer.postCallback 来设置的。Vsync 信号在目前系统中主要由 DispSyncThread 产生，软件中断方式（硬件中断可以用于校准，此功能由 needsHwVsync

控制，在 SurfaceFlinger.cpp 中）。

如果是 Activity、View 的可见性、位置、大小发生变化这些内部事件触发视图绘制，则流程有所不同：

1） View.invalidate

2） ViewParent.invalidateChild（具体实现是在 ViewGroup.invalidateChild 中，通过循环调用各层 ViewGroup 的 invalidateChildInParent 来实现了向上递归的绘制）

3）ViewRootImpl.invalidateChildInParent

```
public ViewParent invalidateChildInParent(int[] location, Rect dirty) {
    checkThread();
    if (DEBUG_DRAW) Log.v(mTag, "Invalidate child: " + dirty);

    if (dirty == null) {
        invalidate();
        return null;
    } else if (dirty.isEmpty() && !mIsAnimating) {
        return null;
    }
```

…

```
void invalidate() {
    mDirty.set(0, 0, mWidth, mHeight);
    if (!mWillDrawSoon) {
        scheduleTraversals();
    }
}
```

12.3 常见性能问题

12.3.1 问题分类

1、经常掉帧

需要通过工具检查掉帧发生在哪个阶段。

2、过度绘制

浪费 CPU、GPU 资源，往往是因为某些像素在同一帧内被重复绘制，例如不可见 UI 组件的绘制。

3、硬件加速时的崩溃

如果是 libhwui.so 库文件出错，检查 RenderThread 与主线程数据同步有无问题，是否需要规避；另外检查使用的 api 兼容性有无问题，如果使用了不支持的 API，系统就需要通过 CPU 软件模拟。

12.3.2 解决思路与方法

UI 渲染类性能问题，可以从下面几个方面去分析定位，进一步缩小范围，以便针对性地优化解决。

- 打开调试开关“开发者选项-显示 GPU 过度绘制”，查看过度绘制区域；
- 使用 Hierarchy View 工具，分析布局层次关系；
- 使用 Tracer for OpenGL ES 工具分析 OpenGL ES 的绘制过程；

- 使用 adb shell 命令包括 dumpsys gfxinfo 包名 framestats、dumpsys SurfaceFlinger 等。

UI 渲染优化思路，主要分为背景优化、渲染区域优化、布局优化以及设计层面的对象复用、资源复用等，可以细分为下面几点。

- 移除窗口、Layout、View 中不必要的背景，根据需要显示占位背景图片。
- 使用 Canvas.clipRect 方法帮助系统识别可见区域。
- 优化布局设计，减少嵌套层次。推荐使用 ConstraintLayout。嵌套层次较少时不用 RelativeLayout 和权重线性布局。RelativeLayout 的子 View 如果高度和 RelativeLayout 不同，则会引发效率问题，尽量使用 padding 代替 margin。
- ListView 组件的优化。除了优化 ListItem 的布局，减少嵌套层级，还可以利用 convertView 参数将之前加载好的布局进行缓存、利用 ViewHolder 进行优化、对图片使用异步加载、使用缓存、快速滑动时，不加载资源，以及分页/分批加载数据。

补充，ConstraintLayout 性能好于 RelativeLayout。在不增加层级深度的情况下，LinearLayout 和 FrameLayout 的性能好于 RelativeLayout，但两层嵌套的 LinearLayout 不如一层 RelativeLayout。

第 13 章 网络管理

13.1 Socket 通信

13.1.1 常用组件

原生系统中，常用的 Socket 通信组件，按协议可以分为 TCP 和 UDP 类型，其中相对常用的 TCP 类型的组件有 java.net.Socket、java.net.ServerSocket，分别对应客户端和服务端 Socket。

其用法比较简单，需要注意的地方，是和多线程的结合、线程间的互斥、数据包的解析包括防丢包/粘包的设计以及异常处理。

1、ServerSocket 组件

1）常用方法

主要是实例化一个对象，传入监听端口参数：

```
mServerSocket = new ServerSocket(SOCKET_PORT);
```

然后通过 accept 方法等待客户端的连接。

2）内部实现

在其构造函数内部，先判断参数的合法性，然后创建一个 InetSocketAddress 实例，并调用组件内部的 bind 函数，进行绑定和监听操作。

```
try {
    SecurityManager security = System.getSecurityManager();
    if (security != null)
        security.checkListen(epoint.getPort());
    getImpl().bind(epoint.getAddress(), epoint.getPort());
    getImpl().listen(backlog);
    bound = true;
} catch(SecurityException e) {
    bound = false;
    throw e;
} catch(IOException e) {
    bound = false;
    throw e;
```

其中 bind、listen 的具体实现，是通过 PlainSocketImpl、Libcore 中间类，最终调用到 libcore.io.Linux 的 native 层接口。

再看 ServerSocket 组件的 accept 方法内部实现，先创建了一个 Socket 实例，然后调用 PlainSocketImpl 中的 socketAccept 函数，如下：

```
void socketAccept(SocketImpl s) throws IOException {
    if (fd == null || !fd.valid()) {
        throw new SocketException("Socket closed");
    }

    // poll() with a timeout of 0 means "poll for zero millis", but a Socket
    // "wait forever". When timeout == 0 we pass -1 to poll.
    if (timeout <= 0) {
        IoBridge.poll(fd, POLLIN | POLLERR, -1);
    } else {
        IoBridge.poll(fd, POLLIN | POLLERR, timeout);
    }

    InetSocketAddress peerAddress = new InetSocketAddress();
    try {
        FileDescriptor newfd = Libcore.os.accept(fd, peerAddress);
```

注意这个函数是阻塞的，阻塞是由调用 poll 接口导致的（底层也是调了 Linux 的 poll 接口，使用监听文件描述符事件的方式）。

当 Socket 连接队列中有已建立的连接后，ServerSocket 的 accpet 所运行的线程就会被唤醒，继续执行后面的流程，重点是根据 accept 的文件描述符，给入参 SocketImpl 实例更新一些属性。

2、Socket 组件

1）常用方法

一种简单的方法是，创建 Socket 实例，并初始化服务端的 IP 地址和端口，例如：

```
if (mSocket == null) {
    mSocket = new Socket("127.0.0.1", PORT);
}
```

但这种方法，如果不能顺利连接到服务端，会阻塞比较长的时间才会抛出异常。

推荐使用如下方法，自定义超时时间，这样可以及时抛出异常，捕获并处理之。

```
socket = new Socket();
SocketAddress saddress = new InetSocketAddress(ip, port);
socket.connect(saddress, SOCKET_TIMEOUT);
if (socket.isConnected()) {
    ...
}
```

2）内部实现

和前面分析 ServerScoket 类似，如果使用第一种方法，在创建实例的同时，就会去连接服务端，因为对应的构造函数会执行到下面这个私有函数中。

```
private Socket(InetAddress[] addresses, int port, SocketAddress localAddr,
        boolean stream) throws IOException {
    if (addresses == null || addresses.length == 0) {
        throw new SocketException("Impossible: empty address list");
    }

    for (int i = 0; i < addresses.length; i++) {
        setImpl();
        try {
            InetSocketAddress address = new InetSocketAddress(addresses[i], port);
            createImpl(stream);
            if (localAddr != null) {
                bind(localAddr);
            }
            connect(address);
            break;
        } catch (IOException | IllegalArgumentException | SecurityException e) {
```

最终也是调用 Linux 的 socket api。

13.1.2 收发线程管理

对于服务端的 Socket 通信线程设计，一种典型的做法是，启动一个专门的服务线程，

通过 ServerSocket 监听客户端的连接请求（对应前面的 accept 调用所在线程），再创建一个数据读取线程，通过已连接的 Socket 读取数据。

当客户端断开连接或者出现异常时，服务端监听线程、数据读取线程需要对应处理。

关于线程的处理，如下几点值得注意：

1、线程的启动和停止；

2、线程间的同步与互斥，锁的使用；

3、线程结束（正常和异常）时的资源释放；

4、线程内的阻塞调用。

13.1.3 防粘包处理

在收发数据量较大且频繁发送时，需要在接收端考虑粘包处理，比较常见的是，读取缓存区时，一次性读到多帧数据。

粘包问题发生的原因，主要有两种：

- 客户端发送的数据大于 socket 缓冲区的数据，导致需要多次 write()操作才能把数据写入到缓冲去，这样导致服务端读取数据的时候，一次读不完整，需要读多次才能读完；
- 客户端发送的每帧数据小于 socket 缓冲区的数据，但发送端 write()数据的速度太快，而接收端 read()数据的比较慢，来不及接收每帧数据，容易出现一次读取到多帧数据。

解决思路是，设计帧头、帧的长度以及校验码，并在接收端利用这些信息，把一次读取到的缓存区数据中的每帧数据都提取出来。一种代码实现如下。

```
while ( mClientSocket != null
    && !mClientSocket.isClosed()
    && inputStream != null
    && (readSize = inputStream.read(buffer)) != -1 ) {

    mActualReadSize = 0;

    if (readSize > FRAME_HEAD_SIZE) {
        frameSize = ((int) buffer[2] << 8) + buffer[3];
    }

    while (readSize >= frameSize) {//read a frame data or sevral frames
        mRecvFrameBuffer = new byte[frameSize];
        mFrameData = new byte[frameSize - FRAME_HEAD_SIZE];//for data

        System.arraycopy(buffer, mActualReadSize,
                mRecvFrameBuffer, 0, frameSize);

        //check crc to get data
        int checksum = socketDataChecksum(mRecvFrameBuffer, frameSize - 1);

        if (checksum == mRecvFrameBuffer[frameSize - 1]) {

            System.arraycopy(mRecvFrameBuffer, FRAME_HEAD_SIZE - 1,
                    mFrameData, 0, frameSize - FRAME_HEAD_SIZE);

            mActualReadSize += frameSize;
```

13.2 NetworkManagementService 浅析

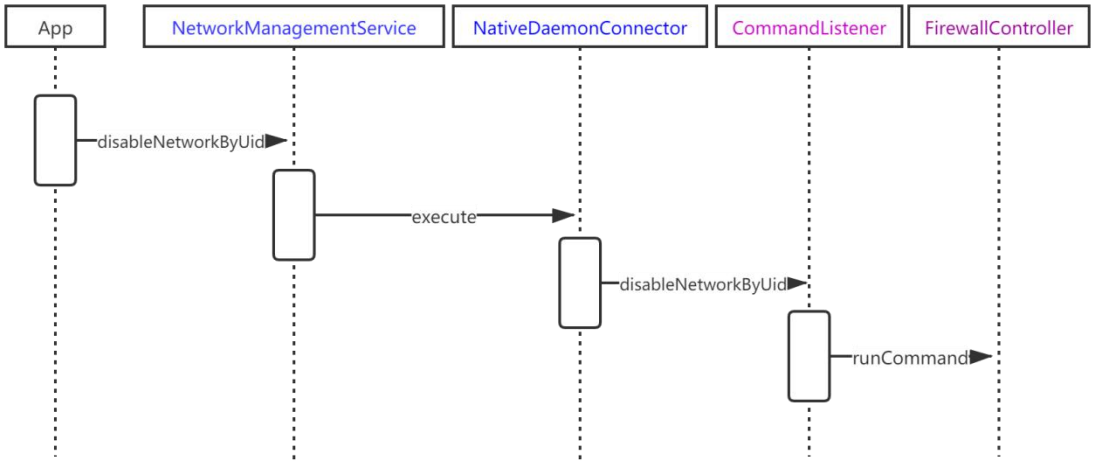
NetworkManagementService 的主要作用是，提供应用层网络管理相关接口，转换为对应的命令，进而通过 NativeDaemonConnector 转发给 netd 处理。另外，还用于接收底层传上来的一些 notify 消息，并做出相应处理。

这里具体通信机制，包括下面几点：

- 通过 onEvent 回调处理底层上报的消息
- 通过 NativeDaemonConnector 创建一个 socket 来通信
- 通过 NativeDaemonConnector 将命令发到底层，例如 firewall。

例如，现在需要扩展一个自定义的防火墙功能，针对某个应用（根据 uid 区分），屏蔽其网络访问功能，在 Java 框架层添加一个 disableNetworkByUid 接口，相关主要流程调

用如下。



13.3 Netd 解析

13.3.1 基础流程架构

Netd 总体上，负责网络管理和控制功能，具体又包括：防火墙、SoftAP 控制、DNS 管理、网络路由管理等功能。

Netd 所属进程由 init 进程根据 init.rc 的对应配置项而启动。在启动时，系统将创建 4 个 TCP 监听 socket，其名称分别为 netd，dnsproxyd,mdns 和 fwmarked。在 netd.rc 中，对应的服务如下：

```
service netd /system/bin/netd
    class main
    socket netd stream 0660 root system
    socket dnsproxyd stream 0660 root inet
    socket mdns stream 0660 root system
    socket fwmarkd stream 0660 root inet
        onrestart restart zygote
        onrestart restart zygote_secondary
```

框架层的 NetworkManagementService、NsdService 分别和 netd、mdns 监听 socket 建立链接，而和域名解析相关的 socket API（如 getaddrinfo 或 gethostbyname

等）的进程都会借由 dnsproxyd 与 netd 建立链接。

在 Netd 的 main 函数中，初始化操作主要包括下面几点：

- 创建 NetlinkManager
- 创建 CommandListener（创建"netd"这个 socket）
- 设置 NetlinkManager 的消息发送者为 CommandListener
- 启动 NetlinkManager
- 初始化 Netd 的环境变量
- 创建 DnsProxyListener（创建"dnsproxyd"这个 socket）
- 启动无限延时循环

其核心组件包括 NetlinkManager 以及 CommandListener，前者负责接收并解析来自 Kernel 的 UEvent 消息，后者用于接收框架层 NetworkManageService 的命令， 定义了多个和网络相关的 Command 类、控制类。

Netd 中的主要调用流程可以分为两种，底层的网络事件上报和上层的命令执行。

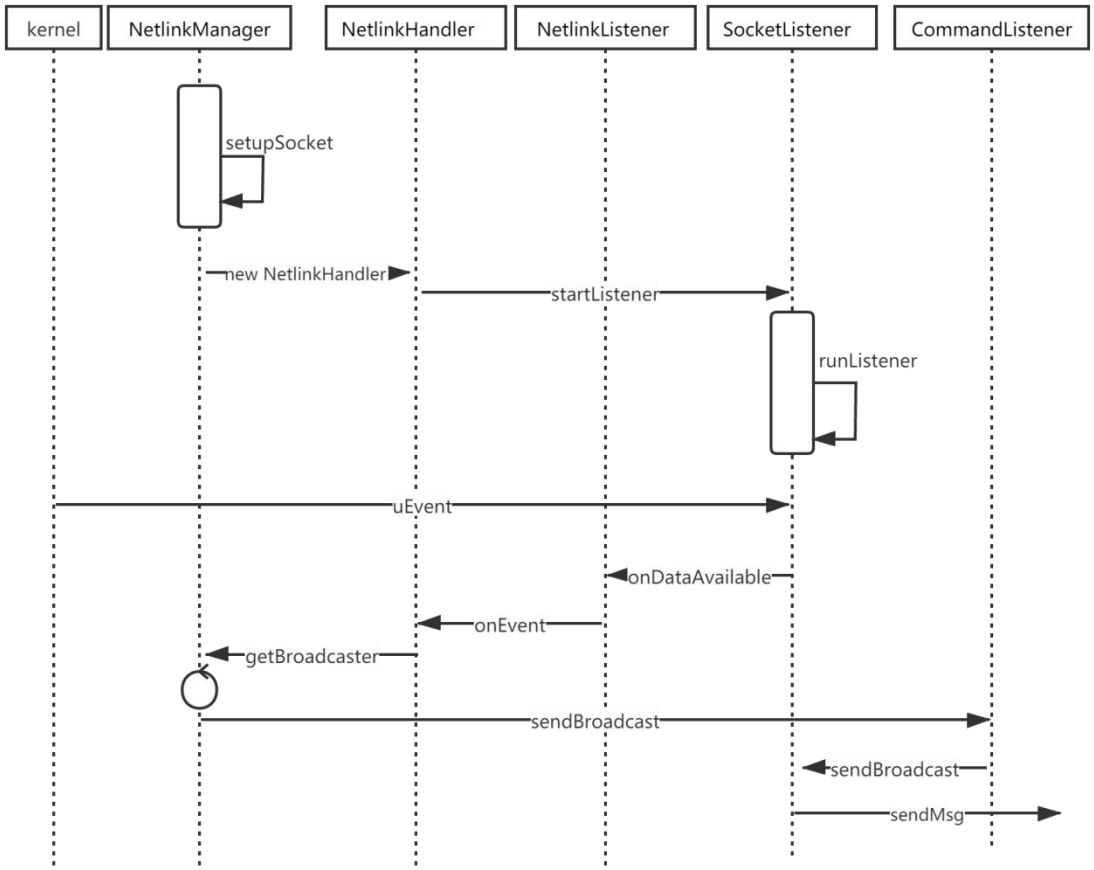
关于底层事件上报，首先，NetlinkManager 在初始化过程中，会通过 start 函数调用 setupSocket 函数中，进而创建 NetlinkHandler ，每个 NetlinkHandler 对象又会通过其派生的 SocketListener 单独创建一个线程用于接收 socket 消息，实际上是通过 socket 以多路复用方式监听。

其次，在 kernel 层有新的事件时，SocketListener 会调用其 onDataAvailable 来处理，这里 onDataAvailable 方法在 NetlinkListener 具体实现。

再次， NetlinkListener 的 onDataAvailable 方法，将内核的 uEvent 消息转化为

NetlinkEvent，并调用 onEvent 来处理消息。这里 onEvent 方法在 NetlinkHandler 具体实现。

最后，NetlinkHandler 在其 onEvent 方法中，先对事件所属的子系统（对应 subsys 变量）进行分类，再判断 action，最后对应调用 notify 函数。而 notify 函数会调用 NetlinkManager 的 Broadcaster 也就是 CommandListener 的 sendBroadcast 方法，发送消息给发送给 Framework 层的 NetworkManagermentService。



关于上层的命令执行流程，在前面已经说明，这里不再重复。

13.3.2 网络防火墙功能客制化

该功能客制化，实现方案的核心是，在开机时调用一个接口，发送一个指令给 Netd，

将需要屏蔽网络功能的进程 uid 的网络请求通过 iptable 规则全部屏蔽掉，具体涉及到 App、Java 框架以及 Native 层的修改。

其中，App 层主要是开机广播和 NetworkManagermentService 的扩展，重点是 Native 层 Netd 中的命令规则，FirewallController.cpp 的修改，参考代码如下。

```
int FirewallController::disableNetworkByUid(int uid) {
    ALOGD("xh-disableNetworkByUid uid: %d", uid);
    int res = 0;
    std::string command = "*filter\n";
    StringAppendF(&command, "-A OUTPUT -m owner --uid-owner %d -j DROP\n",uid);
    StringAppendF(&command, "COMMIT\n");
    res |= execIptablesRestore(V4V6,   command.c_str());
    ALOGD("xh-disableNetworkByUid res: %d", res);
    return res;
}
```

13.4 网络路由

13.4.1 基础概念

路由管理策略，基本概念包括：

- 路由规则（rule）：一条路由策略 rule 主要包含三个信息：优先级（数小优先级高 0>32766>32767）、条件、路由表。系统根据满足何种路由策略条件，来指定选择哪个路由表执行路由。在添加规则时，必须先确定好“条件”、“优先级别”及“路由表 ID”，此后才可以执行添加规则的操作。（注：这里的规则 rule，在很多地方称为策略，个人更倾向规则的叫法）
- 路由表（table）：存有到达特定网络终端路径的一种表，在某些情况下，还有一些与这些路径相关的度量，例如 ip 地址段。其功能是为需要路由转发的每个数据包寻找一条最佳的传输路径。

- 路由规则和路由表的联系：多个策略 rule 可以指向同一张路由表。某些路由表可以没有策略指向它。如果指向某个路由表的所有策略 rule 被删除，那么这个路由表是没有用的，但它在系统中仍然存在。

常用的命令/工具有：

- 查看路由规则列表：ip rule list。
- 查看特定 id 的路由表：ip route list table [ID]，例如 ip route list table 254。
- 查看当前在使用的/实际生效的路由表：busybox route -n。
- 添加路由表 1 并设置优先级为 9000：ip rule add from all table 1 pref 9000。
- 添加路由规则（示例）：ip rule add from 192.168.1.10 table 10，如果数据包的来源端 IP 是 192.168.1.10，就参考路由表 10；ip rule add from 192.168.2.0/24 table 20，如果来源端 IP 为 192.168.2.0/24 网段的 IP，就参考路由表 20。其中 24 对应子网掩码 255.255.255.0。更常用的是根据目的地 IP 地址或者网段进行转发，将这里的 from 改为 to 即可。还可以根据数据包输入的接口进行判断，使用 dev 关键字指定。
- 删除路由规则：删除方式很灵活，可以根据优先级、table 或者条件，例如 ip rule del table 1。
- 刷新路由缓存区：ip route flush cache。

13.4.2 多网络并存策略

在某些设备中，往往需要多个网络同时并存，例如：插入网线时，自动使用以太网，此时 wifi 也可以成功连接。并且在 wifi 连接之后，可以使用 wifi 上外网。以太网和 wifi 同时可用，使用同一网段的测试机，都可以 ping 通。

核心修改思路是：修改网络评分机制、网络路由管理策略，包括在一个网络连接之后，不断开另外一个网络。

不同的平台实现可能会有差异，一般可以通过脚本的方式来修改网络路由规则，通过 init.rc 启动一个服务，通过服务的方式执行脚本，并在网络可用时，利用属性变化来触发路由配置脚本的执行，注意在设置路由之后，直接复位属性。

例如：

```
#!/system/bin/sh
ip route add 192.168.30.0/24 dev eth0 proto static scope link table legacy_system
setprop eth.route ""
```

这里新增自定义复位，往往需要对应地配置 SELinux 权限，关于权限方面的问题与解决方法，在后面章节说明。

第四篇 性能优化体系

第 14 章 内存优化

14.1 内存管理机制

14.1.1 Java 内存管理

1、主内存和工作内存

在基于 JRE 的 Java 运行环境中，所有的变量都存储在主内存中，每个线程有自己的工作内存，而工作内存又保存了该线程所用变量的主内存的副本，所有线程对变量的操作必须在工作内存中进行。

这里有三个重要的概念。

1）原子性：不能被线程调度中断。一般认为基本数据类型的读写操作具备原子性， 可使用 synchronized 来保证操作的原子性。

2) 可见性：一个线程修改了共享变量的值，其他线程能够立即得知修改。可使用 volatile 保证可见性，另外 synchronized 和 final 也可以实现可见性。

3）有序性：如果在本线程观察，所有操作都有序；但在一个线程观察另一个线程，所有操作都是无序的。无序是由于指令重排、工作内存与主内存的同步延迟导致。可使用 synchronized 和 volatile 保证多线程操作的有序性。

——synchronized 都可以实现上面三个特性，但不能滥用，不然会产生性能问题（涉及锁优化）。

2、引用类型

1）强引用：不会回收。即使内存空间不足，垃圾回收器也不会回收强引用对象。

2） 软引用：内存不足才回收。只在内存空间不足时，才回收这些对象的内存。

3） 弱引用：一发现就回收。只要发现弱引用对象，不管当前内存空间足够与否，都会回收。

4）虚引用/幽灵引用：一发现就加队列。回收一个对象时，如果发现它还有虚引用，就会把它加入到与之关联的引用队列中。对象被收集器回收时可以收到一个系统通知，须和 ReferenceQueue 联合使用。

14.1.2 Linux 内存管理模型

1、基本概念

1）虚拟地址：在段中的偏移地址，这里“段”可以理解为：基地址+段的界限+类型。

2）线性地址：在某个段中“基地址+偏移地址”得出的地址。

3）物理地址：在 x86 中，如果未开启分页机制，则线性地址就等于物理地址；否则需要经过分页机制换算后，线性地址才能转换成物理地址

内存管理机制，核心是分段和分页机制。分段和分页机制，提供虚拟地址到物理地址的映射方法。具体又包括内核态内存管理和用户态内存管理这两种类型。

内核态内存管理，负责连续物理地址的内存分配，通过 kmalloc 函数为内核申请连续的内存空间，或者通过 vmalloc 函数来获得非连续物理地址的内存分配。

用户态内存管理，使用 libc 的 malloc 库分配空间。在进程地址空间内，线性地址空间的底部维护了 text、data、bss 段，在高端维护了栈区域，中间则有 heap 区域和 mmap 区域。其中 bss 段存放程序中未初始化的全局变量，text 段也就是代码段。

不管是哪种类型的内存管理，内存分配操作，最终分配的都是线性地址空间，不会直接

分配物理内存。

2、 Lowmemorykiller 机制

1） 基本原理

Kernel 层有个 kswapd 内核线程，会遍历一张链表并执行回调，触发 LMK 的执行。当某个 app 分配内存、发现可用内存不足时，内核会阻塞分配操作，并在该进程中去执行 lowmemorykiller 来杀进程以释放内存。

2） 杀进程机制

根据当前系统的可用内存（对应不同警戒级数/水位）、进程 oom_adj 以及进程的内存占用情况来杀进程。

如果进程的 oom_adj 大于对应的警戒级数，进程将会被杀死。具有相同 oom_adj 的进程，则优先杀死占用内存较多的。oom_adj 越小，代表进程越重要，这里 oom_adj 的值可以客制化。

14.1.3 JVM 虚拟机内存管理

虚拟机内存管理包括内存区域模型、内存的分配与回收，具体来说，虚拟机在堆上给对象分配内存，并且几乎所有的对象实例和数组都要在堆上分配。

1、内存区域模型

内存区域分为线程私有部分和线程共有部分，前者又包括程序计数器、虚拟机栈、Native 方法栈；而后者包括 Java 堆和方法区。

其中私有部分，程序计数器是当前线程执行的字节码的行号指示器，占用一块较小的、唯一不会出现 OOM 的内存区域。虚拟机栈，描述的是 Java 方法执行的内存模型，这里每

个方法在执行的同时会创建一个栈帧（方法运行时的基础数据结构，存储了局部变量表、操作数栈、动态链接等信息）。调用方法时，栈帧入栈；执行完成时，栈帧出栈。

共有部分，Java 堆包括新生代和老年代，还可以细分为 Eden 空间、From Survivor 空间、To Survivor 空间等区域。方法区，是各个线程共享的内存区域，用于存储已被虚拟机加载的类信息、常量、静态变量等数据。

2、内存分配与回收

关于内存分配，重点是给对象在堆上分配内存。几乎所有的对象实例和数组都要在堆上分配。

首先，对象主要分配在新生代的 Eden 区，偶尔直接分配在老年代区域，细节取决于垃圾回收器以及虚拟机参数配置。其次，对象在新生代 Eden 区分配时，如果可用空间不够则会发起 Minor GC（对应 GC_FOR_ALLOC）。再次，大对象进入老年代，这里要尽量避免使用很快就会回收的大对象（需要大量连续内存空间的 Java 对象）。最后，长期存活对象进入老年代，具体是根据相同年龄对象总大小以及年龄综合判断（是否进入老年代），不只是根据阈值判断。

14.2 优化思路

14.2.1 静态数据分析

静态数据分析，包括统计不同场景下的内存占用情况、找到对应的应用程序、进程，并优化设计或实现，减少内存占用。做好数据分析，也有利于制定合理的优化目标。

这里在设计和实现层面的优化，大致可以分为以下几种：

1、使用合适的数据结构

- 使用 StringBuilder 代替+（优化拼接操作）
- 尽量避免使用 Enum 枚举，改用静态常量
- 元素不超过 1000 个时，推荐用 SparseArray 或 ArrayMap 取代 HashMap。如果 key 是 int 型，优先使用 SparseArray，否则使用 ArrayMap。
- 尽可能复用系统资源（含图片对象）。
- 谨慎使用 Service，必要时停止后台服务，或者使用 IntentService。
- 谨慎使用 large heap，只有在清楚地知道时功能与副作用时才去使用。

优化过程通常不是一蹴而就的，需要反复验证、修改，在优化过程中，需要注意控制每一次的修改数量/范围，以方便验证优化效果。

14.2.1 排除内存泄漏

内存泄漏的常见类型有如下几种。

类型	描述	优化
Activity 的泄漏	场景 1：非静态内部类默认支持有外部类的引用，导致 Activity 泄漏，含多线程、Handler 相关的内部类	需要改为静态内部类。
	场景 2：Activity Context 被传递到其他实例中导致泄漏	建议使用 Application Context 代替 Activity Context
Bitmap 对象泄漏		及时回收
WebView 的泄漏		考虑为 WebView 开启新进程
静态 View 的泄漏		在 onDestroy 方法中将 View 对象设置 null

14.2.2 检查虚拟机参数配置

内存相关的虚拟机参数可以关注下面几个。

1、堆起始大小 heapstartsize：虚拟机启动时分配的初始堆内存大小。

2、堆增长上限 heapgrowthlimit：系统给 App 可使用的 Heap 的最大值，超过这个值，App 就会产生 OOM。

3、堆内存最大值 heapsize：不受控情况下的最大堆内存大小，如果 App 的 manifest 文件中配置了 largeHeap 属性，则 App 可使用的 Heap 的最大值为此项设定值。

这里 heapsize 和 heapgrowthlimit 需要根据系统 RAM 以及常用 app 的内存使用情况，多次调整看效果。

heapgrowthlimit 参考值：

【中内存】2G/3G RAM 配置 128MB~256MB

【低内存】512M/1G RAM 配置 96MB~128MB

heapsize 参考配置：

【中内存】2G/3G RAM 配置 256MB~512MB

【低内存】512M/1G RAM 配置 128MB~256MB

14.2.3 Kernel 层的内存优化

Linux 内核层面，内存优化思路包括 KSM、swap 内存置换包括内存压缩等。

1、启用 KSM：使用 ksm 服务扫描内存分页，合并相同内容的内存分页，以减少内存占用。

2、启用 swap：内存置换。可调整 vm.swappiness，对应置换操作的发生概率，在

RAM 较小时，建议适当增大该参数。在 Android 系统中，swap 交换机制是在 RAM 中，结合 zRAM 内存压缩来实现的，内存压缩需要占用 CPU 资源，所以 swappiness 也不宜设置过大，避免卡顿。

14.3 相关工具

工具	功能&使用场合	原理
Android Profiler	检查部分进程的内存泄漏	
MAT	根据 hprof 文件分析可能的内存泄漏	
LeakCanary	检查 Activity 和 fragment 的泄露	在应用启动时，监听 Activity 和 fragment 生命周期；在生命周期结束后，开始检测泄漏。尝试两次回收，再判断实例是否存在，若存在则判定泄漏。有泄漏时，分析具体信息并弹出提示
dumpsys meminfo	打印各进程占用的内存信息，适合 Java 进程和 Native 进程	
libc_malloc_debug.so	检测 Native 层内存泄漏	

14.4 问题与思考

14.4.1 不同层次的优化

内存优化，一般可以优先从应用层面分析、排除内存泄漏，再根据初步分析结果判断是否需要在框架层、虚拟机甚至内核层面评估优化。在采用不同方案修改验证时，推荐分批调试验证，在推荐值的基础上前后微调，以便分析效果和副作用。

有次，某项目的图库应用，在多次查看视频详情后退出窗口，再次进入无法播放，log 中可以看到 Out of memory 字样，初步分析可能是在解析/播放视频引起内存泄露导致 OOM。

这个问题，实际通过两方面来定位解决。一个是复现、反复操作，通过 Android Profiler 跟踪图库进程没有发现异常，但通过 procrank 发现 mediaextractor 进程 PSS 显著升高。使用 libc_malloc_debug.so 工具，定位到 MetadataRetrieverClient::setDataSource，对应 MediaMetadataRetriever::setDataSource(int fd, int64_t offset, int64_t length)，在 jni 中，对应 android_media_MediaMetadataRetriever_setDataSourceFD 函数，java 层则是对应 MediaMetadataRetriever 的 setDataSource 函数。

另外，同步在应用层进行代码走查，特别是检查播放、查看详情相关调用流程有无异常，用法是否合理。

综上发现，图库应用在调用 MediaMetadataRetriever.setDataSource 之后没有 release，导致 native 进程 mediaextrator 出现内存泄露，次数多了就会出现底层 OOM。解决方案是配对使用，setDataSource 之后，及时 release。

第 15 章 卡顿优化

15.1 相关指标和工具

15.1.1 常用卡顿相关性能指标

1、CPU 的使用率

使用率的查询/获取方法：cat /proc/stat 或/proc/[pid]/stat。其中 stat 文件各个属性的，包括用户时间、系统时间、缺页次数等，具体可以使用 top、vmstat、strace 等 shell 命令查询。

2、CPU 的饱和度

饱和度反映线程排队等待 CPU 的情况（负载情况），饱和度首先和应用的线程数有关（线程切换很占用 CPU）。可通过 vmstat 或 cat /proc/[pid]/schedstat 查看线程切换情况，通过 uptime 查看特定时间段的平均负载。

这里注意优先级，线程优先级会影响 CPU 饱和度，例如，是否存在高优先级的线程在空等低优先级线程？特别是后台线程锁。

15.1.2 相关工具

1、systrace（Monitor)

原理：封装了 atrace 命令来开启 ftrace 事件，并读取 ftrace 缓冲区，基于性能探针/埋点方式监测耗时情况，生成可视化的 HTML 日志。ftrace 是 Linux 下常用的内核跟踪调试工具，而 atrace 在 ftrace 基础上扩展了一些 Android 专用的 categories 和 tag。

2、Android Profiler

- 支持 Sample Java Methods：类似 TraceView 的 sample 类型
- 支持 Trace Java Methods：类似 TraceView 的 instrument 类型
- 支持 System Calls：类似 systrace
- 支持 SampleNative（26+）：类似 Simleperf

结果展示方式有 Call Chart、Flame Chart。

其他工具还有 Traceview，可用于分析耗时方法的具体情况。另外，Hierarchy Viewer 可以用于分析 UI 渲染问题，也可以用于分析卡顿问题，因为一些卡顿问题和渲染处理、过度绘制有关。

15.2 常见卡顿原因

1、布局复杂，渲染时间过长

需要合理使用各种布局组件，层级较多时推荐相对布局，层级不多但嵌套较多时推荐线性布局，使用 Merge 标签去掉多余层级（适用于帧布局），必要时考虑使用 ViewStub。

2、GPU 过度绘制

可能是布局时，控件重叠且设置了同样的背景，或者 onDraw 方法实现不合理（考虑使用 canvas.clipRect 属性指定绘制区域），可以使用开发者选项，进入过度绘制模式，辅助调试。

3、 UI 线程耗时过多

分析、优化耗时任务的执行方式，例如不在生命周期执行耗时任务，谨慎采用异步任务组件。另外检查有无锁等待、不合理的同步。

4、GC 操作导致线程暂停

频繁 GC 导致卡顿，主要是非并发 GC，典型的是 GC_FOR_ALLOC，内存分配失败时触发，会暂停应用线程，容易引起卡顿。这时候需要结合内存使用情况综合优化。

这里需要注意把卡顿优化和内存优化结合起来分析，因为内存优化如果存在问题，容易导致内存分配失败引发的 GC_for_Alloc，引起卡顿。

5、ROM 存储空间不足导致写放大效应

ROM 存储空间不足时，碎片较多，容易导致写放大效应，增加 CPU 资源的占用。这种情况需要结合存储优化综合考虑。

15.3 关于 ANR

15.3.1 日志分析

重点分析 trace 文件，必要时主动生成 trace 文件，例如设计一套监控机制，在主线程卡顿时，主动向系统发生 SIGQUIT 信号，或者现场通过 adb 命令 kill 卡死进程。

分析 ANR 问题，先根据 log 信息判断是哪种类型、有无死锁或者不合理的耗时任务。主要分类包括：Input 事件超时、Service 超时、广播处理超时、主线程锁等待、

15.3.2 ANR 的触发机制

不同类型的 ANR，其流程也会有所差异，不过整体上的流程都差不多。首先是启动倒计时，然后判断是否在规定的时间内完成任务，完成则取消倒计时；如果没有在规定的时间内完成任务，则系统立即封装现场，打印相关信息、kill 问题进程，根据场景判断是否提示用户。

1、服务启动超时

以 startService、startForegroundService 方式启动前台服务时，启动的超时时间为 20s，而后台服务超时为 200s。还有一种是后台进程启动前台服务，超时阈值是 10s。

2、广播处理超时

也就是 onReceive 方法执行超出时限，这里前台广播超时为 10s，后台广播超时为 60s。

3、输入事件处理超时

包括按键处理、触摸响应等输入事件处理，即使某次事件执行时间超过 timeout 时长，只要用户后续在没有再生成输入事件，则不会触发 ANR。

ANR 还可以分为前台 ANR 和后台 ANR，具体取决于该应用发生 ANR 对用户是否可感知。后台 ANR 不会弹框提示（但可能伴随 Crash），例如后台进程启动前台服务的场景。

15.4 问题与思考

15.4.1 关于卡顿问题分析

卡顿只是一个现象，其原因是多方面的，例如内存不足导致非并发 GC、ROM 空间不足导致写放大，布局优化不够导致过度绘制等，需要根据初步分析，再结合内存优化或者存储空间优化、UI 渲染优化来进一步分析解决。

卡顿问题有时候不太好复现，可能需要考虑设计一套监控机制。常用的方法有：消息队列、hook 插桩等，有时候还需要结合线程管理对线程的创建、数量以及时间加以监控。

15.4.2 ANR 中的广播超时

1、前台广播和后台广播

前台广播和后台广播，本质上的差异是处理广播的队列不同。这两种队列的差异体现在：

1）名称不同，查看 AMS 的 mFgBroadcastQueue、mBgBroadcastQueue 对象的创建可以发现；

2）超时时间不同，分别对应 BROADCAST_FG_TIMEOUT、BROADCAST_BG_TIMEOUT，在 Android9.0 中，取值为 10s 和 60s。

3）队列的构造函数中的入参 allowDelayBehindServices 不同，后台队列可能出现等待。所以，使用前台队列处理广播发送，可以提高实时性。

2、串行广播和并行广播

在前面组件部分提过，静态注册的广播接收器始终采用串行方式来处理，这里加以补充：

动态注册的 registeredReceivers 处理方式是串行还是并行方式，取决于广播的发送方式，例如发送有序广播则采用串行方式，否则采用并行方式处理。

3、广播中的 ANR 超时

有一种说法是，仅仅串行广播才需要考虑超时，串行广播既可以采用前台队列也可以采用后台队列处理。如果 onReceive 执行在主线程，则 ANR 超时时间基本上就是 onReceive 的处理时间，取决于广播使用的是前台队列还是后台队列，分别对应 10s、60s。

但普通广播，默认采用并行方式、后台队列，当 onReceive 执行在主线程并且超过 10s（或 60s），也会超时。所以这里不能简单根据串行还并行来判断是否可能出现广播 ANR。

问题：如果 onReceive 执行在子线程，如果超时，还会出现 ANR 吗？如果出现 ANR，是否 Silent ANR？（动态注册广播时，在子线程创建对应的 Handler 即可让 onReceive 运行在子线程）

第 16 章 启动优化

16.1 开机启动优化

16.1.1 首次开机启动优化

首次开机启动耗时方面的性能优化，需要关注两个耗时点。

首先是 dex2oat 及其触发的 CompileDriver 操作(由 PMS 的扫描操作 scanDirLI 通过 installd 调用），功能是将 dex 字节码转换为机器码；其次是预加载流程包括 preloadClasses、preloadResources 等。

从优化的性价比角度出发，主要思路是将开机启动时的 dex2oat 处理挪到版本编译时，或者说，用空间换时间。

具体来说，可以根据开机 log 选择 5-8 个较为耗时的 App，在其 mk 文件里面添加 LOCAL_DEX_PREOPT :=true，这样对空间的增加不会太大。 相关工具有：systrace、logcat。

注意事项：正确设置 32 位、64 位的编译配置，否则不会生成对应的 odex/vdex 文件。

16.1.2 非首次开机启动优化

非首次开机优化，需要针对具体问题，包括某个环节存在 bug 或者实现方案不合理，考虑不周到，还有就是根据产品需求，对开机环节中的某些不必要的耗时等待进行裁剪。

例如：

- 设备驱动初始化不合理，存在不必要的耗时操作，导致 kernel 初始化耗时过长；
- 底层 wifi 驱动尚未初始化完成，上层就去访问，导致等待；
- 产品不需要锁屏、墙纸功能，在 Framework 层可以针对性地缩短这部分绘制等待耗时。

16.2 App 启动

16.2.1 问题分析

首先确定问题性质、类型，包括：启动过程中点击图标不响应、主窗口显示太慢、窗口显示后不响应操作。目的是对应到具体的启动阶段、进一步精确定位问题。

例如，主窗口显示太慢，可能是 onCreate 函数中存在耗时任务，也可能是布局问题、影响了渲染性能，还可能是图片资源问题、出现不必要的缩放处理导致耗时增加。

16.2.2 思路

1、启动过程分析

这部分通常包括 Splash 窗口显示、主窗口显示、界面可操作 (初始化完成) 几个阶段。

2、优化思路

1）Splash 窗口优化。

2）业务梳理。裁剪不必要的业务模块，只加载需要的。

3）业务优化。分析主线程主要的耗时点，评估能否通过算法优化或异步线程完成，检查有无大量的监听回调、框架、hook 初始化集中运行。

4）线程优化。重点是，减少 CPU 调度带来的波动，让应用的启动时间更加稳定。一方面控制线程数量(使用线程池、注意线程被动切换次数),另一方面检查线程锁,通过 systrace 检查锁等待事件，排查这些等待是否可以优化，避免主线程长时间等待。

5）GC 优化。目的是减少 GC 次数。可通过 systrace 查看启动过程中的 GC 时间，也可以使用 Debug.startAllocCounting 来监控启动过程中 GC 耗时情况，如果发现较多 GC

耗时，则需通过 Allocation 工具进一步分析。

另外，注意避免大量字符串操作、序列化或反序列化操作，复用对象、必要时采用 Native 实现，避免内存泄露。

6）系统调用优化。通过 systrace 的 SystemService 类型可查看 SystemServer 的 CPU 工作情况。在启动过程中，尽量不做系统调用，包括 PMS、Binder 调用等；尽量不要启动其他进程。

补充，在分析启动性能问题时，往往需要用到卡顿问题分析的思路和方法，例如在启动过程中，避免执行耗时任务、优化线程锁、优化 GC、合理使用异步操作。

第 17 章 崩溃优化

常见的崩溃可以分为 Java 层崩溃、Native 层崩溃，其原因可能是特殊情况保护不够、出现未捕获异常，也可能是程序错误地访问了非法地址。

17.1 异常捕获

17.1.1 Java 层异常捕获

- 使用合适的 Exception 派生类
- 使用 UncaughtExceptionHandler

17.1.2 Native 层崩溃

Native 层崩溃的分析难点是，如何保证客户端在各种极端情况下可以生成崩溃日志，这里有几种场景：

1、文件句柄泄露导致创建日志文件失败

2、由于栈溢出导致日志生成失败

3、整个堆的内存耗尽导致日志生成失败

4、堆破坏或二次崩溃导致日志生成失败

捕获异常崩溃的第三方工具/服务，常用的有：Breakpad（Chromium）、Bugly（腾讯）、啄木鸟平台（阿里）。

17.2 获取现场

分析崩溃问题，需要尽可能获取更多的现场信息，包括崩溃信息、系统信息、内存信息和资源信息等。

崩溃信息包括：进程名、线程名（前台、后台、UI 线程），以及崩溃堆栈和类型（Java/Native/ANR）。

系统信息包括：机型、厂商、CPU、Android/Linux 版本，设备状态等。

内存信息包括：

- 系统剩余内存（直接读取/proc/meminfo）
- 应用使用内存（包括 Java 内存、RSS、PSS，可通过/proc/self/smap 计算）
- 虚拟内存（可通过/proc/self/status 得到）

很多崩溃和内存有关，包括 OOM、ANR、虚拟内存耗尽等，2GB 内存以下的崩溃率相对会高出很多。

资源信息包括文件句柄 FD（限制可读取/proc/self/limits）、线程数（可通过/proc/self/status 得到）等。

应用信息，包括具体的 Activity 或 Fragment、关键操作路径以及其他自定义信息（和具体业务、log 打印有关）。

17.3 问题分析

17.3.1 确定重点

首先确认严重程度（性价比），例如复现概率较高、对用户使用影响严重的优先分析、必要时通过专项测试、压力测试复现获取更多信息。

其次，尽可能获取崩溃发生时的基本信息。

如果发生了 Java 层崩溃，需要根据 Exception 异常类型，判断对应原因与优化方案，OOM 则需查看内存和资源信息。

对于 Native 层崩溃，则需观察 signal、code、fault addr、堆栈等信息，常见的有 SIGSEGV（指针）、SIGABRT（ANR 和 abort）。

Native 层崩溃一般都是由 NativeCrashListener 监听处理的。在 SystemServer 进程启动过程中，启动各种其他系统服务时，会创建一个用于监听 native crash 事件的 NativeCrashListener 对象， 该对象通过 socket 机制来监听，等待 debuggerd 与该线程创建连接并处理相应事件，进而调用 AMS 的 handleApplicationCrashInner 来处理 Native 层崩溃。

如果发生了 ANR，首先根据 trace 文件查看主线程堆栈，是否有锁等待，再看 iowait、CPU、GC、systemserver 等信息。例如出现 CPU 竞争时，查找"anr in"或"not responding"发生时间，并根据该时间点前后 5s 的 log 计算 CPU 耗时，看哪个进程占用最多，再分析该进程存在的问题。如果发现大量 GC 操作导致卡死，可参考内存优化的处理方案。

在 Logcat 文件中，需要重点关注警告和错误信息，区分不同的关键字（例如 am_anr、am_kill），当一条崩溃日志无法确认原因时，可查看相同崩溃点下的更多日志。

17.3.2 尝试复现

分析崩溃问题时，关注尝试查找某类崩溃有无共性，将一些信息例如机型、os 等系统信息或者应用信息（url、video、国家、地区等）加以聚合，有利于问题的复现和定位。

复现崩溃需要根据崩溃现场信息操作，例如根据系统信息获取对应机型、软件版本复现；根据应用信息、复现路径，对应测试相关功能；根据内存或 ROM 使用信息，使用测试工具

消耗内存或 ROM 存储空间并尝试复现。

一些小概率崩溃，可能需要借助自动化工具例如 MonkeyRunner 进行复现。

17.3.3 常见异常和错误

异常分类	异常名称	具体信息	优化思路
Java 语法相关	空指针	NullPointException	参数判空、 返回值中不为空、
	角标越界/数组越界	IndexOutOfBoundsException	预判数组/集合是否为空、是否超出长度
		StringIndexOutOfBoundsException	
		ArrayIndexOutOfBoundsException	
	类异常	ClassNotFoundException	检查类名有无错误、是否成功导入了依赖 jar 包
		NoClassDefFoundError	检查 DexClassLoader 相关使用，包括第三方 sdk（重点关注开发期间可以找到但运行时找不到的类）
Activity 相关异常	Activity 未找到	ActivityNotFoundException	检查 Activity 是否正确定义，Intent 参数例如 uri 有无问题
	无法实例化 Activity	RuntimeException:Unable to instantiate activity ComponentInfo	检查包名或者 activity 的类名是否修改过
Window/Dialog 相关异常	视图未附着于窗口管理器	View not attached to window manager（Dialog 所属的 Activity 已经不存在了）	不要在非 UI 线程中管理对话框，合理使用 Activity.onCreateDialog 回调，让对话框对象在 Activity 的可控制范围之内和生命周期之内
	无法添加窗口 -Activity 未运行	BadTokenException：Unable to add window-is your activity running	判断 Activity 是否存在然后再弹出；判断是否需要开启一个新线程（使用 Runnable），或延时执行对话框的显示、修改弹框时机
	无法添加窗口 -token 错误	BadTokenException：Unable to add window --token null is not for an application	改为使用 Activity 的 Context
资源相关的异常	链接异常	UnsatisfiedLinkError	检查 libs 的 armeabi 目录下的 so 文件是否存在，还要看 x86 目录下 so 文件是否存在
	布局解析异常	InfiateException:Binary XML file	在 Activity 的 onStop 方法中，

		line#18:	手动释放每一张图片资源

第五篇 权限管理体系

第 18 章 基于 PackageManager 的权限管理

18.1 基础知识

18.1.1 权限保护等级

Android 系统中，上层（主要指 App 层）常用的一种权限管理机制，就是基于 PackageManager 的权限（permission）管理。

当一个应用在安装时，PackageManager 包管理器会把应用在 AndroidManifest.xml 中声明的权限，添加到“权限数据库”中去，这个数据库也是是包数据库（ data/system/packages.xml ）的一部分。

这种权限管理方式，和保护等级密切相关。其保护等级主要有：

1、normal 级别

默认值，定义了访问系统或其它 APP 的低风险权限。无需用户确认，会自动授权。

2、dangerous 级别

可以访问用户数据或某种程序的控制用户设备。系统会弹出一个确认对话框显示请求信息，并由用户手动确认。

3、signature 级别

只会授权给那些与权限声明者使用相同证书的 APP。使用这种权限，必须持有 APP 或平台的签名私钥。

4、signatureOrSystem 级别

这是一种折中的方案，权限可被赋予系统应用或拥有相同签名的应用。上述保护等级，在 AndroidManifest 文件中，对应 android:protectionLevel 属性。

18.1.2 权限申请方式

基于 PMS 的权限管理包括静态申请、动态申请方式，具体和级别有关。

其中静态申请需要在 AndroidManifest.xml 文件中定义，不管是 normal 级别还是 dangerous 级别都需要静态申请。

而动态申请是针对 dangerous 级别的权限，需要在运行时向系统申请并弹出提示，由用户确认是否授予所需权限，主要通过 checkSelfPermission、requestPermissions 和 onRequestPermissionsResult 回调处理。在部分终端产品中，为了改善用户体验，对部分重要应用会配置白名单，避免不必要的弹框。

18.1.3 权限白名单

如前所述，系统应用往往需要使用一些重要的 dangrous 权限，为了避免弹框提示用户，改善用户体验，可以在系统框架层面上配置一些权限白名单。

方法 1：通过 xml 配置。首先在 frameworks/base/data/etc/permissions 目录下添加对应的文件，例如系统应用对应 privapp-permissions-NAME.xml（推荐将其权限列在 privapp-permissions-platform.xml 中）然后使用 development/tools/privapp_permissions/privapp_permissions.py 工具，执行之前需

要 source、lunch，这种方法没有使用过，实际使用的是下面的方法。

方法 2：修改源码。首先修改 PackageManagerService 的 grantRuntimePermission 实现，其次是修改 DefaultPermissionGrantPolicy 的实现，对 package 加以判断。

注意事项：如果遇到 log 提示权限异常，检查 android:persistent 是否设置为 true，另外检查 mk 文件中有无 LOCAL_CERTIFICATE := platform。

18.1.4 核心组件中的权限

1、Activity

在 Activity 组件中，某个 Activity 调用 startActivity()或 startActivityForResult()方法时，AMS 会检查目标 Activity 有无权限要求，如果有权限要求，则会检查调用者是否已经申请了权限。

2、Service

与 Activity 组件类似，在启动服务时，包括调用 startService()、stopService()和 bindService()方法，AMS 也会检查待启动的目标 Service 有无权限要求，和 Activity 类似。

3、Broadcast

广播组件稍有不同，可以设置双向权限，例如发送者可以要求接收者具有某个权限，而接收者也可以要求发送者具有某个权限。具体是在收到广播时检查上述权限。

4、ContentProvider

该组件涉及到的权限分为几种。

首先是读写权限。该组件可为读、写操作分别指定权限，读权限在 query 被调用时执行，写权限在 insert、update、delete 被调用时执行。其读写权限的配置，需要在 AndroidManifest 文件中使用 permission、android:readPermission 等关键字声明。

其次，ContentProvider 还可以设置 Uri 访问权限，或者叫路径权限。需要在 AndroidManifest 文件中使用 path-permission 和 android:pathPrefix、android:path、android:pathPatten 关键字声明。调用者也要对应声明权限（整个 provider 或者特定路径的权限）。

另外，需要注意一点，就是权限传递。例如，某个 ContentProvider 将自己的 Uri 权限通过 Intent 传递给其他需要的应用，需要在 AndroidManifest 中设置 android:grantUriPermissions 为 true，再调用 Intent 的 setData 方法，以设置需要访问的 Uri 参数，最后给 Intent 设置 Intent.FLAG_GRANT_READ_URI_PERMISSION)。

18.2 问题与思考

18.2.1 权限不足问题

1、缺少权限声明导致的权限问题

这种属于最简单的一种，在 AndroidManifest.xml 中对应声明即可。

2、动态权限申请导致的问题

从 Android6.0 开始，部分权限需要动态申请，用户授权之后才能操作，否则会导致 Crash 问题。存在多个权限时，注意合理安排权限申请的顺序，既不冗余也不遗漏，例如

Manifest.permission.READ_EXTERNAL_STORAGE 、Manifest.permission.WRITE_EXTERNAL_STORAGE，有的只需要读，有的需要读+写。

另外就是从应用层面，考虑首次启动用户未授权时、首次启动用户已授权、首次启动用户拒绝授权以及部分授权的场景，某些情况下，还需要考虑引导用户打开权限设置窗口。

补充一个特殊的动态权限申请，写系统设置（例如通过更新 SettingProvider 来调整全局背光设置），需要判断是否有权限写入，没有权限则通过 Intent 弹出权限授予窗口。

3、其他权限问题

如果是文件读写权限问题，除了上面的几种类型，往往还会涉及到 uid、gid 以及不同分区的 mount 配置，这里需要结合 Linux 中的权限管理机制具体问题具体分析。

第 19 章 Linux 中的权限管理

19.1 基础知识

Linux 中的权限管理机制，基于标准 UNIX 的安全模型，具体包括以下几个要素：

- 每个用户都有一个数字形式表示的用户 ID，对应 uid；
- 每个用户都有一个数字形式表示的主组 ID，其 id 就是组 id（gid），同时用户可以加入其他的组，成为这些组的成员。
- 文件的访问权限由特定用户（ owner ）、组（ group）和其他人员三部分组成。
- uid 为 0 的用户（对应 root 用户）在系统中拥有最高权限，可以访问所有的文件资源。

概括来说，就是基于用户和分组的文件访问权限管理机制，其中用户的属性包括 uid、gid，用户需要访问的资源（文件）的属性，其中 owner 属性对应用户属性 uid，group 属性对应用户的组 id 也就是 gid。

补充，在 Android 上，一个用户 UID 对应一个应用程序，每个用户都至少属于一个组，也可能属于多个组。

另外不同的应用可以通过 shareUserId 属性共享一个 uid，一个应用可以有多个 gid，以获得多个权限。

19.2 问题与思考

19.2.1 权限的映射

Android 中的基于 PackageManager 的权限，和 Linux 中的基于用户/分组的权限，有着重要的联系，这个联系的桥梁在什么地方？

首先，能够被映射的权限都定义在/etc/permissions/platform.xml，只有权限名到组名 group gid 的映射关系。

其次，组名到 GID 的映射是静态的，定义在 android_filesystem_config.h 文件中的 android_ids 数组 。具体是通过 APP 申请的权限找到对应组名，在 android_ids 数组中通过组名找到 GID，之后这个 GID 会作为补充 GID 赋予 APP（在 data/system/packages.list 可以查看）。

19.2.2 不同的应用程序间的数据共享

不同进程：使用进程通信机制例如 Binder，或者 ContentProvider、文件等方式。

同一进程：必须使用相同的私钥签署这些应用程序、使用 AndroidManifest 文件给它们分配相同的 UID（设置 android:sharedUserId）。

19.2.3 权限不足问题

1、不同分区的读写权限问题

在 Android 系统中，不同的分区例如 system、data 分区，以及特殊的 tmp 分区，其读写权限是不同的，默认情况下，system 分区只读，data 分区可以写入但是还要看 uid、gid。

如果要将数据写入 system 分区，可以通过 init.rc 启动一个 native 服务，并在启动该服务执行 su 以获取 root 权限。

在 recovery 模式下，如果要中转数据，例如备份某些不希望被擦除的数据，可以通过

tmp 分区进行中转，并且在 data 分区下面选一个合适的目录作为数据的来源以及备份后写入的目录。

第 20 章 基于 SELinux 的权限管理

20.1 基础知识

20.1.1 TE 访问控制

1、SELinux 策略规则

- allow ： 允许主体对客体进行操作
- neverallow ：拒绝主体对客体进行操作
- dontaudit ： 表示不记录某条违反规则的决策信息
- auditallow ：记录某项决策信息，通常 SELinux 只记录失败的信息，应用这条规则后会记录成功的决策信息。

2、te 文件语法

te 文件有固定的格式，例如：rule_name source_type target_type:class perm_set；其中 rule_name 代表规则名，分别有 allow,dontaudit,neverallow 等；source_type 主要作用是用来填写一个域(domain)。target_type 代表类型。class 是指类别，主要有 File,Dir,Socket,SEAndroid 还有 Binder 等。perm_set 代表动作集，read、write、open、create、execute 等。

一个典型的例子：allow appdomain app_data_file:file rw_file_perms;

后面结合具体问题进一步说明。

20.1.2 SELinux 安全机制

1、基础框架逻辑

SELinux 所使用的基本方法是“打标签”。标签是资源（object）分配一个类型（type），或为进程（对应主体 subject）分配一个安全域（security domain），目的是，使得在同一个域中的进程才可以访问授权的资源。

换句话来说，同一个资源，对于不同的标签的进程，其访问权限往往也是不同的。

2、几个概念：

- 安全上下文：一个附加在对象上的标签。实际上是一个字符串，由四部分组成（用户、角色、类型、安全级别）。
- 安全策略。安全策略是在安全上下文的基础上进行描述的，具体对应到 te 文件。在 system/sepolicy 目录和其他所有客制化的 te 目录，所有以.te 为后缀的文件经过编译之后，就会生成一个 sepolicy 文件。这个 sepolicy 文件会打包在 ROM 中并且保存在设备上的根目录下，需要我们在系统启动的时候进行加载。
- Security Server。它主要是用来保护用户空间资源的，以及用来操作内核空间对象的安全上下文的。整体上由应用程序安装服务 PackageManagerService、应用程序安装守护进程 installd、应用程序进程孵化器 Zygote 进程以及 init 进程组成。其中 PackageManagerService 和 installd 负责创建 App 数据目录的安全上下文，Zygote 进程负责创建 App 进程的安全上下文，而 init 进程负责控制系统属性的安全访问。

20.1.3 SELinux 相关配置

1、强制执行等级

- 宽容模式（permissive）：仅记录但不强制执行 SELinux 安全政策。
- 强制模式（enforcing）：强制执行并记录安全政策。

2、SELinux 配置

临时关闭可以使用 adb 命令 setenforce 0（断电之后，节点值会复位）。如果要永久关闭，可以修改属性，设置 ro.boot.SELinux=permissive 或者修改 system/core/init/Android.mk 中的 DALLOW_PERMISSIVE_SELinux 为 1（userdebug、eng 版本默认是 1）。

还有其他方法，例如修改内核.SECURITY_SELinux 设置为 false，重新编译 kernel。还可以修改 init.cpp，不过这种方法没有试过，不推荐。

20.2 问题与思考

20.2.1 编译时的报错问题

1、重复定义

- 原因：device/mediatek/（或者 device/qcom/或者其他类似的目录）和 system/sepolicy 下面重复定义了
- 解决方案：优先在 device/mediatek/或类似的目录下修改，不要在 system/sepolicy 下面重复添加。

2、某个进程 sepolicy 配置冲突，提示 neverallow 字样

这类问题表现类似，但是可能的原因较多，例如下面几种。

- 原因 1：其 te 文件中，属性关联了 coredomain，但是 domain.te 里面系统默认不允许

coredomain 程序访问 vendor_files 的 dir 权限。

✧ 解决方案：te 文件中去掉 coredomain 的属性关联，再修改编译 mk 文件，使其放到 vendor/bin 下面。

原因 2：客制化的修改，与原生系统 domain.te 中的 neverallow 配置出现冲突。

✧ 解决方案 1：通过设定目标类型或者属性关联绕过冲突，指定要访问的目标为一个“源类型”有权限访问的“目标类型”，例如：在 file_contexts 中添加： /dev/xxx u:object_r:audio_device:s0。

✧ 解决方案 2：修改 domain.te，添加白名单，但是需要谨慎修改，避免影响 CTS 或者相关测试。

20.2.2 运行时的权限不足问题

1、基础方法

根据 log 中的 avc denied 关键字使判断需要添加的权限以及对应的 te 文件，也可以用 audit2allow 工具生成对应的 policy 规则，按照特定关键信息设定好。

例如：avc: denied { bind } for scontext=u:r:hal_vehicle_default:s0 tcontext=u:r:hal_vehicle_default:s0 tclass=tcp_socket。

这段 log 可以分解为：

- 缺少什么权限： { bind }等权限
- 谁缺少权限：u:r:hal_vehicle_default:s0
- 对哪个资源缺少权限：tcontext=u:r:hal_vehicle_default:s0
- 什么类型的资源： tclass=tcp_socket

2、某进程读取属性失败（错误码 0x18）

✧ 原因：新增自定义属性的读写，需要添加对应的权限配置。

✧ 解决方案：在 property_contexts 里面添加对应的属性，例如 vendor.test.prop u:object_r:system_prop:s0。

补充：如果客制化的是 ro 属性，则默认只能读取（要写入 ro 属性，需要修改底层 init 模块里面的属性管理逻辑，对属性名称加以特殊处理）。

3、hostapd 进程写入属性失败

该问题和前面类似，但有些不同。

✧ 关键 Log：hostapd: type=1400 audit(0.0:325): avc: denied { write } for name="property_service" dev="tmpfs" ino=12621 scontext=u:r:hal_wifi_hostapd_default:s0 tcontext=u:object_r:property_socket:s0 tclass=sock_file permissive=1

✧ 原因分析：主要是根据 log 解析需要的权限：

❖ 缺少什么权限： { write }权限
❖ 谁缺少权限：u:r:hal_wifi_hostapd_default:s0
❖ 对哪个资源缺少权限：tcontext=u:object_r:property_socket:s0
❖ 什么类型的资源： tclass=sock_file

✧ 解决方案：在 system\sepolicy\vendor 的 hal_wifi_hostapd_default.te，添加一行：

allow hal_wifi_hostapd_default property_socket:sock_file { read write };

还是这个进程，在运行时还有一个权限问题，与前面类似，根据以下 log：

hostapd: type=1400 audit(0.0:619): avc: denied { connectto } for path="/dev/socket/property_service" scontext=u:r:hal_wifi_hostapd_default:s0 tcontext=u:r:init:s0 tclass=unix_stream_socket permissive=1

hal_wifi_hostapd_default init:unix_stream_socket { connectto };

得出需要添加的权限配置为：

allow hal_wifi_hostapd_default init:unix_stream_socket { connectto };

20.2.3 混合型问题

在实际开发过程中，我们往往会遇到一些既有权限不足也有编译报错问题，可以归为混合型问题，这里以一个 hal 层的进程（hal_vehicle_default）为例来说明。

1、访问 tcp_socket 资源的 node_bind 权限问题

- 关键 log：avc: denied { node_bind } for src=33452 scontext=u:r:hal_vehicle_default:s0 tcontext=u:object_r:node:s0 tclass=tcp_socket permissive=1
- 权限解析：hal_vehicle_default 对 device 资源缺少 chr_file 类型的资源权限，包括 { read append open }
- 解决方案：

1）在 hal_vehicle_default.te，添加 allow hal_vehicle_default ttysWK_device:chr_file { read append open };

2）在 public/device.te，添加 type ttysWK_device, dev_type;

3）在 vendor/file_contexts，添加/dev/ttysWK[0-9]* u:object_r:ttysWK_device:s0

4)在 prebuilts/api/26.0/public/device.te 以及 27、28 对应文件夹添加 type ttysWK_device,

dev_type;

补充：这里新建一个自定义 device 和 log 中的节点对应并声明，避免权限放大，而不用修改原生配置就可以解决 neverallow 问题。另外系统中可能存在多个类似的设备节点，例如 ttysWK0、ttysWK1，在 file_contexts 定义新的文件设备时，可以使用通配符，加以简化。

2、访问 socket 资源的 ioctl 权限问题

- ✧ 关键 log：avc: denied { ioctl } for path="socket:[34593]" dev="sockfs" ino=34593 ioctlcmd=8933 scontext=u:r:hal_vehicle_default:s0 tcontext=u:r:hal_vehicle_default:s0 tclass=socket permissive=1
- ✧ 权限解析：hal_vehicle_default 对 hal_vehicle_default 资源缺少 socket 类型的特定资源权限，包括 ioctl { SIOCGIFINDEX }。注意 log 中的 8933 是 ioctlcmd 值，可在 system/sepolicy/ioctl_defines 找到对应定义。
- ✧ 解决方案：在 hal_vehicle_default.te，添加 allowxperm hal_vehicle_default hal_vehicle_default:socket ioctl { SIOCGIFINDEX };

补充：这里如果直接添加 allow hal_vehicle_default hal_vehicle_default:socket { bind create ioctl };会导致 neverallow 编译问题，因为 ioctl 权限默认有限制。还有一种修改方案是：将 hal_vehicle_default hal 关联的 automotive_socket_exemption 属性，在 domain.te 里面对应加入白名单：neverallowxperm { domain -automotive_socket_exemption } domain:socket_class_set ioctl { 0 };

整体上，建议使用改动最小、不修改原生 domain 配置、不存在权限放大的方案，更为稳妥、方便维护。

第六篇 虚拟机体系

第 21 章 ART 启动流程分析

21.1 系统开机时的 ART

21.1.1 准备工作

在 Android 系统开机时，init 进程启动后，init 会解析 system/core/rootdir/init.rc，注意在 rc 文件中，有个 zygote 服务，对应 app_process 程序：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    class main
    priority -20
    user root
```

Init 在解析到该服务时，会通过 do_class_start、StartIfNotDisabled 一系列调用启动进程。

21.1.2 创建 VM 实例

启动 zygote 进程后，系统会创建一个 AppRuntime 对象，再调用其 start 方法，最终会调用到其父类 AndroidRuntime 的 start()，进而调用 startVm 函数。具体包括：

1、通过 JniInvocation 来初始化运行环境（利用系统属性来动态加载 libart.so）；

2、通过 AndroidRuntim 的 startVM 来创建虚拟机实例；

3、在 Zygote 进程加载指定的类。

在框架层的 jni 实现，AndroidRuntime.cpp 中，startVM 函数主要做了两件事：

1、通过若干系统属性初始化虚拟机选项配置；

2、通过 JNI_createJavaVM 创建虚拟机实例。

在 native 库（源码 JniInvocation.cpp）中，JNI_createJavaVM 会调用到 art 的 jni 层具体实现（源码 java_vm_ext.cc），这里的 JNI_CreateJavaVM 完成了下面几点工作：

- Runtime::Create(options, ignore_unrecognized)；
- runtime->Start();

另外还获取了 Jni 运行环境和虚拟机的实例，分别通过 GetJniEnv()和 GetJavaVM()方法。

在使用 Create 方法创建 VM 实例时，之前解析的虚拟机配置选项 options 参数会被传入，最终通过 instance_ = new Runtime 来完成实例的创建，再通过 instance_->Init(std::move(runtime_options))调用，使得配置选项生效。

21.1.3 启动 ART

继续前面的 runtime->Start()的执行流程分析，该函数的主要工作大致可以简化为如下几点：

1、InitNativeMethods()：初始化 VM 相关的若干 native 方法，包括注册 DexFile、VMDebug、VMRuntime、以及反射、线程、ClassLoader 等，其次是加载一系列的 native 库包括 libjavacore.so、libopenjdk.so；其中 WellknowClass 的初始化在 well_known_classes.cc 里面定义，都是一些比较基础的类。

2、CreateSystemClassLoader(this)：创建系统的类加载器。

3、StartDaemonThreads()：启动守护进程，通过 jni 回调 java.lang.Deamons 的 start 方法，进而启动 ReferenceQueueDaemon、FinalizerDaemon、FinalizerWatchdogDaemon 等。

4、RegisterAppInfo：字面含义是注册 App 信息，最终通过 jit 的 StartProfileSaver 方法完成注册操作。其主要功能，推测是为 app 的预编译、jit 编译提供方便，例如存储 app 的一些

信息，以便系统根据这里的信息判断是否需要执行预编译、jit、以及 dex 文件存储到什么地方。

前面分析了系统开机时的 ART 启动流程主体，后面再分析新启动一个 app 进程时，对应的虚拟机的创建和启动流程。

21.2 创建进程时的 ART

21.2.1 不同层次下的调用逻辑

进程的创建，在组件体系之 Activity 的启动、基础框架体系之创建进程提过，这里从 ART 的角度，再看下相关的调用逻辑。

1、基于源码层次的调用逻辑

从源码层次例如 java 框架、native 框架的角度来看，关键调用涉及到：

- 【java 框架】/frameworks/base/core/java/com/android/internal/os/，ZygoteInit、RuntimeInit、Zygote 等 java 实现；
- 【java 框架】/frameworks/base/core/java/android/os/，Process.java；
- 【jni】com_android_internal_os_Zygote.cpp、AndroidRuntime.cpp；
- 【native 框架】/frameworks/base/cmds/app_process/，App_main.cpp；
- 【native 库】/bionic/libc/bionic/，fork.cpp、pthread_atfork.cpp；
- 【native 库】/libcore/dalvik/src/main/java/dalvik/system/ZygoteHooks.java；
- 【art】/art/runtime/，Runtime.cc、Thread.cc 等。

2、基于进程切换的调用逻辑

如果从 Launcher 首次启动某个应用的 Activity 或服务，这里的进程切换分为几步：

Step1：Launcher 进程，通过 java 框架层 startActivity、包括 AMS 的一系列启动模式、stack 相关处理，最后调用到 Process.start，binder 方式和 system_server 系统服务进程通信;

Step2: system_server 进程，Process.start 方法会利用 socket 调用到 Zygote 进程，以创建新的进程；

Step3：Zygote 进程，由于系统初始化的时候，有个 runSelectLoop 循环在执行，这里通

过 socket 监听 system_server 的进程创建请求，并通过 libc 中的底层接口 fork 出新进程(子进程)；

Step4：Zygote 进程，通过 ZygoteConnection 以及反射调用 ActivityThread.main()方法，完成新进程的创建和初始化。

第 22 章 预编译机制

22.1 预编译的场景

预编译机制的优点：提前编译出机器码，提升应用程序的执行效率。缺点有两方面：一是存储空间增加，二是某些场景下的耗时增加，包括应用安装时间、系统首次开机启动时间、源码编译时间变长。

我们需要根据具体情况，合理配置，从整体上考虑系统性能的优化。

预编译的执行时机有 3 个地方：

case1：linux 服务器源码环境编译版本时（需在 BoardConfig.mk 中配置 WITH_DEXPREOPT := true 或者模块的 mk 文件里面配置 LOCAL_DEX_PREOPT :=true）

case2：首次开机启动过程中，由 PMS 扫描操作调用 installd 完成预编译（系统预置应用）

case3：在应用安装时，由 PMS 调用 installd 完成预编译

其中系统首次启动时的预编译调用流程，如下：

step1：PMS（PackageManagerService）初始化

step2：PMS 的扫描、解析与判断

step3：PMS 调用 Installer 代理类的相关函数

step4：Installer 通过 socket 调用 installd 的 dex2oat（源码在 dexopt.cpp 中）

step5：installd 中 dex2oat 调用 CompileDriver 启动预编译（多线程，线程数量和 CPU 核心数有关）

这里，在扫描时，PMS 会根据时间戳判断是否 ota 升级后的首次开机扫描，避免执行不必

要的预编译工作。

源码环境下的预编译，需要关注的是如何配置编译开关，包括 LOCAL_DEX_PREOPT 以及 32 位、64 位的 MULTI_LIB 编译控制。

22.2 相关补充

22.2.1 性能优化

ART 本质是个 Runtime，由于 ART 实现了 3 个 JVM 接口，所以有时候称为 ART 虚拟机。这三个接口是：获取初始化参数、在进程中创建 JVM 实例、获取已创建的 JVM 实例。

相对于 Dalvik，ART 的改进主要有两点：

- ART 支持 AOT 预编译机制，可以提前将字节码编译为机器码，运行效率高；
- GC 机制有优化，减少了线程暂停次数，降低了 GC 导致的卡顿程度。

和预编译有关的性能优化配置，主要是首次开机启动提速，可以调整预编译过程，选择最为耗时的几个 apk，修改其 mk 文件，使其在 aosp 源码编译阶段就执行 dex2oat 调用而不是在首次开机阶段。

在修改预编译配置之后，需要注意 system 分区的增加情况，避免分区设置不合理、预留的余量不够导致一些异常情况。

22.2.2 关于 dex/odex/oat 文件格式

1、dex

dex 是将 apk 中使用到的 class 文件信息集合在一起的文件，其中也包含了很多 jar 包中的类，对应字节码。在一些 jar 包里面有 dex 文件，例如 classes.dex。

2、odex

case1：未经过 AOT 预编译的 odex 文件，只是针对 dex 做了部分优化（减少冗余信息、提高加载和运行效率）

case2：经过 AOT 预编译的 odex 文件，ELF 格式封装的（这里的 odex 其实就是 oat 文件，只是文件名还是 odex 后缀）

3、oat

一种 ELF 格式的二进制可运行文件，包含 DEX 文件和编译出的机器码文件。比 ODEX 文件占用空间更大。

注意：封装格式是 ELF 格式的文件，其文件后缀可能是 oat、odex；后缀是 odex 的，也不一定就是预编译过的机器码文件

4、art

odex 进行优化生成的可执行二进制码文件。生成 art 文件后，/system/app 下的 odex 和 vdex 会无效（所以导入 framework.jar 文件后要删除 boot.art）。

第 23 章 类加载机制

23.1 加载时机和过程

23.1.1 加载时机

在虚拟机执行 new、getstatic、putstatic 或 invokestatic 这 4 条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。

而生成这 4 条指令的几个场景包括：

1）使用 new 关键字实例化对象、访问类的静态字段，以及调用一个类的静态方法的时候。

2）对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

3）当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

4）当虚拟机启动时，用户需要指定一个要执行的主类（包含 main（）方法的那个类），虚拟机会先初始化这个主类。

23.1.2 加载过程

先看 JVM 的加载规范：

在加载阶段，虚拟机需要完成以下 3 件事情：

1）通过一个类的全限定名来获取定义此类的二进制字节流。

2）将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

3）在内存中生成一个代表这个类的 java.lan g .Class 对象，作为方法区这个类的各种数据的访问入口。

结合规范，类加载过程可以简化概括为：加载->验证->准备 ->解析->初始化 ->使用>卸

载。

其中，加载、验证、准备、初始化和卸载的顺序固定，而解析则不一定，因为 Java 是动态语言（可以在初始化之后也就是运行时解析）。

解析阶段也是虚拟机将常量池内的符号引用替换为直接引用的过程，解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符 7 类符号引用进行。如果解析失败会抛出抛出 java.lang.NoSuchFieldError 异常。

23.2 类加载器

23.2.1 类加载器的分类

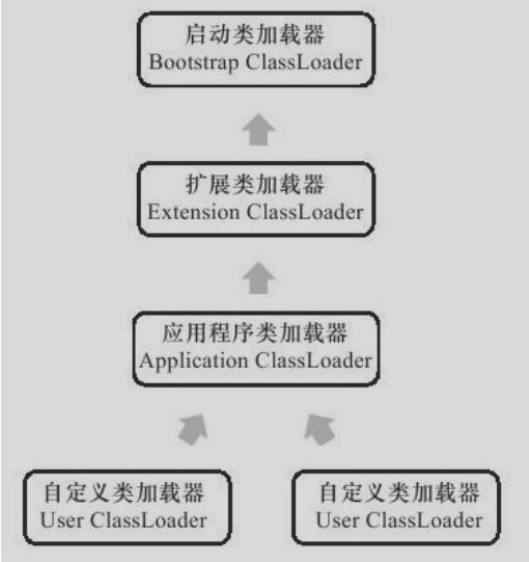
从某种意义上，类加载器可以简单分为一种是启动类加载器（BootstrapClassLoader），这个类加载器使用 C++语言实现，是虚拟机自身的一部分；另一种就是其他类加载器，这些类加载器都由 Java 语言实现，独立于虚拟机外部，并且全都继承自抽象类 java.lang.ClassLoader。

具体还可以细分为：

- 启动类加载器
- 扩展类加载器
- 应用程序类加载器

23.2.2 双亲委派模型

双亲委派模型，首先是层次关系。前面提到的三种类加载器，按照自上而下的层次结构组织起来，最底层是自定义的类加载器。



其次是，加载器收到类加载请求时，先委派给父类加载器去处理，只有父类加载器无法加载时，才会尝试自己去加载。

综合上面的层次结构和处理顺序来说，可以概况为：

- 如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成；
- 每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中；
- 只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

可以结合 ClassLoader.java 的源码实现来理解。

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
        // First, check if the class has already been loaded
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }

            if (c == null) {
                // If still not found, then invoke findClass in order
                // to find the class.
                c = findClass(name);
            }
        }
        return c;
} « end loadClass »
```

上面源码的核心逻辑：

Step1：如果父加载器不为空，则使用父加载器去加载类；否则，查找启动类加载器；期间如果父加载器没有找到对应的类，则抛出 ClassNotFoundException。

Step2：如果没有父加载器也没有周到启动类加载器，则尝试自己去加载。

第 24 章 VM 内存管理机制

24.1 内存区域的划分

24.1.1 线程私有部分

在虚拟机的内存管理模型中，线程私有部分的内存区域包括以下几种：

1、程序计数器。

这是一块较小的内存空间，用作当前线程执行的字节码的行号（位置）指示器，也是唯一不会出现 OOM 的内存区域。

2、虚拟机栈。

这个栈，描述的是 Java 方法执行的内存模型，每个方法在执行的同时会创建一个栈帧，栈帧是方法运行时的基础数据结构，存储了局部变量表、操作数栈、动态链接等信息。在通过 Android Studio 断点调试 Java 代码的时候，就用到了这些栈帧数据。调用 Java 方法时，栈帧入栈；执行完成时，栈帧出栈。

3、Native 方法栈

该栈描述的是 Native 方法执行的内存模型，虚拟机规范对这部分没有强制规定。在使用 libmalloc_debug.so 调试 native 层内存泄漏的时候，就用到了 native 方法栈，每个线程都有自己的一个调用栈。

24.1.2 线程共有部分

线程共有部分包括：

- Java 堆。又分为新生代、老年代的 Java 堆。
- 方法区。用于存储已被虚拟机加载的类信息、常量、静态变量等数据。

24.2 内存分配与 GC

24.2.1 分配机制浅析

内存分配的重点是，对象在堆上分配内存。几乎所有的对象实例和数组都要在堆上分配（除了局部变量对应的临时对象在栈里面分配）。

1、对象主要分配在新生代的 Eden 区。偶尔直接分配在老年代区域，细节取决于垃圾回收器以及虚拟机参数配置。

2、对象在新生代 Eden 区分配时，如果可用空间不够则会发起 Minor GC（对应 GC_FOR_ALLOC？）

3、大对象进入老年代。这里注意，在创建实例对象时，要尽量避免使用很快就会回收的大对象。

4、长期存活对象进入老年代。判断是否进入老年代的规则是，根据相同年龄的对象总大小以及年龄综合判断，而不只是根据阈值判断。

24.2.2 垃圾收集

24.2.2.1 GC 算法

1、标记-清除（Mark-Sweep）：从根集合开始，将内存整个遍历一次，保留可以被根节点引用到的对象，回收其余对象。优点是算法简单，缺点：效率低、容易产生内存碎片。

2、复制算法（Copying）：将现有内存空间分为两块，每次只使用其中一块。垃圾回收时，将存活对象复制到未使用的内存块中，再清理待回收的内存空间。优点：效率较高，速度快。缺点：占用空间（空间换时间）。

3、标记-整理（也叫标记压缩，Mark-Compact）：先从根节点遍历一次，将存活对象压

缩到内存的另一端，再清理。优点：避免碎片、空间占用不高，缺点：压缩操作会占用 CPU、暂停应用。

4、分代算法。这种算法属于上述算法的组合运用，主要包括：对新生代采用“复制” 算法，而对老年代采用了“标记-清除”或“标记-整理”算法。分代算法的好处是，效率较高，注意复制算法不适合老年代，因为老年代存活周期长

24.2.2.2 GC 的分类

从触发原因的角度来分，包括下面几种类型：

1、GC_FOR_ALLOC：内存分配失败时触发，会暂停应用线程，容易引起卡顿。重点：非并发，比较耗时，尽量避免。

2、GC_CONCURRENT：并发 GC，应用的堆内存达到一定值或快要满时，系统会自动触发。重点：并发。

3、GC_BEFOR_OOM：准备抛出 OOM 之前所进行的最后努力。重点：非并发。

4、GC_EXPLICIT：显式触发，应用调用 System.gc、VMRuntime.gc 时触发。非并发。

上述 GC 都是在内存分配过程中触发的。

24.3 参数调优浅析

24.3.1 常用 VM 参数解析

1、起始大小 heapstartsize：虚拟机启动时分配的初始堆内存大小。

2、增长上限 heapgrowthlimit：系统给 App 可使用的 Heap 的最大值，超过这个值，App 就会产生 OOM。

3、堆内存最大值 heapsize：不受控情况下的最大堆内存大小。如果 App 的 manifest 文件中

配置了 largeHeap 属性，则 App 可使用的 Heap 的最大值为此项设定值。

4、其他参数还有下面几个，一般不做修改。

- 堆目标利用率 heaptargetutilization，作用：设定内存利用率的百分比。虚拟机会在 GC 的时候调整堆内存大小，让实际利用率向这个百分比靠拢。例如高通 msm89xx 平台，mk 文件里面有默认设置：dalvik.vm.heaptargetutilization=0.75。注意，该参数不要太大也不要太小。
- 堆最小空闲值 heapminfree
- 堆最大空闲值 heapmaxfree

24.3.2 低内存设备的优化

低内存设备，关注 heapsize 和 heapgrowthlimit 的配置调整，需要根据系统 RAM 以及常用 app 的内存使用，多种因素综合考虑，多次调整看效果。下面是一种参考配置：

Heapgrowthlimit：1G~2G RAM 的设备，考虑配置 128MB~192MB，512M~1G RAM 可考虑配置 96MB~128MB。

Heapsize：1G~2G RAM 的设备考虑配置 192MB~256MB，512M~1G RAM 考虑配置 128MB~192MB。

其他还有 oom_adj 相关的内存阈值配置，也可以根据实际情况微调，使得待机时的可用内存保持一个相对合适的水平，避免 OOM 或者过于频繁的 lmk 杀进程。

无论是哪种类型的参数调整，修改后都需要大量的测试和观察验证。

《Android 系统研究笔记》

持续完善中...