1. ActivityThread，AMS，WMS 的工作原理(源码)

https://www.jianshu.com/p/47eca41428d6

Activity 与 WIndow：

Activity 只负责生命周期和事件处理

Window 只控制视图

一个 Activity 包含一个 Window，如果 Activity 没有 Window，那就相当于 Service

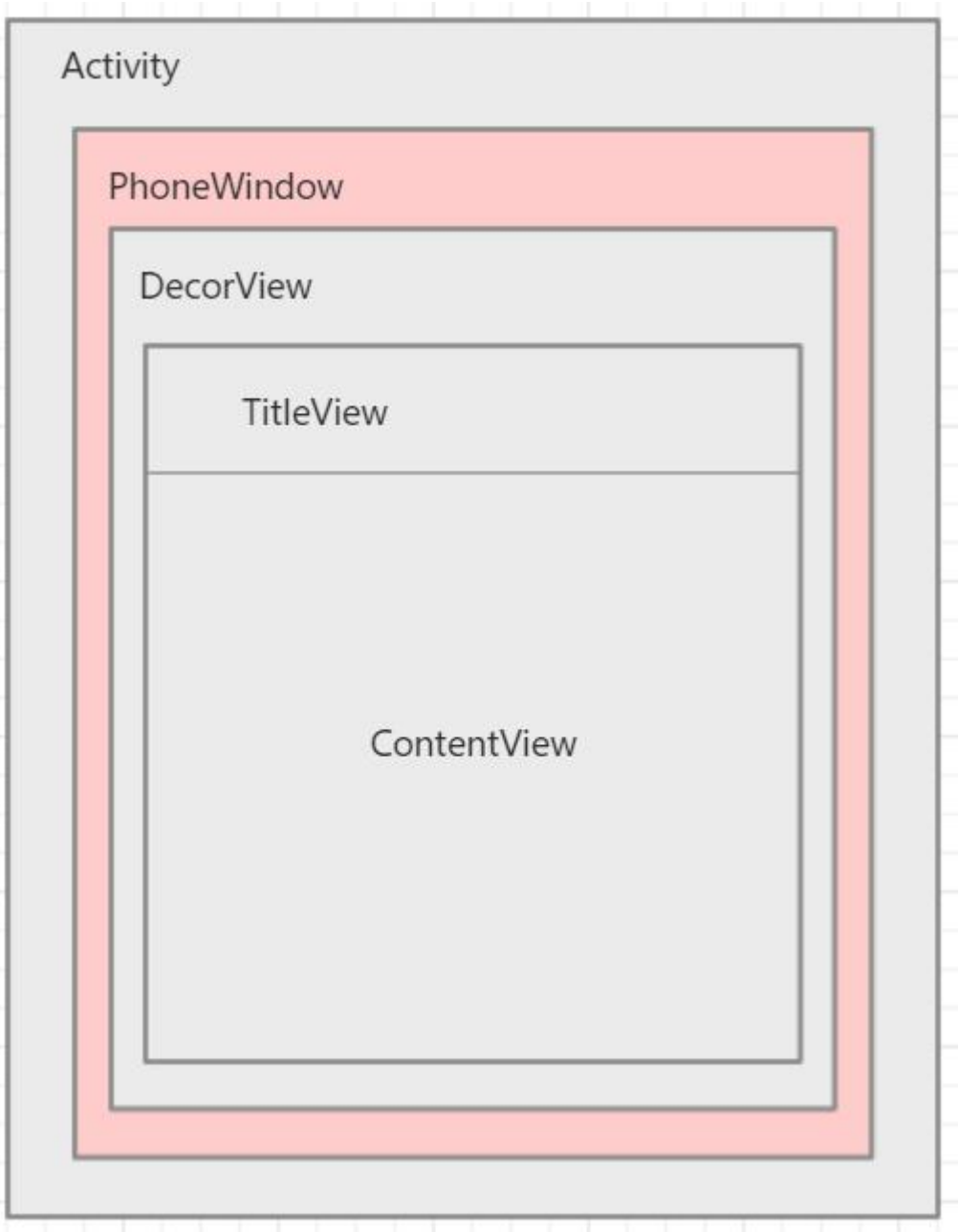
AMS 与 WMS：

AMS 统一调度所有应用程序的 Activity

WMS 控制所有 Window 的显示与隐藏以及要显示的位置

在视图层次中，Activity 在 WIndow 之上，如下图

ActivityThread 是 Android 应用的主线程（UI 线程）



WMS(WindowManagerService)

管理的整个系统所有窗口的 UI

作用:

1) **为所有窗口分配 Surface**：客户端向 WMS 添加一个窗口的过程，

其实就是 WMS 为其分配一块 Suiface 的过程，一块块 Surface 在 WMS 的管理下有序的排布在屏幕上。Window 的本质就是 Surface。（简单的说 Surface 对应了一块屏幕缓冲区）

2）管理 Surface 的显示顺序、尺寸、位置

3）管理窗口动画

4）**输入系统相关：**WMS 是派发系统按键和触摸消息的最佳人选，当接收到一个触摸事件，它需要寻找一个最合适的窗口来处理消息，而 WMS 是窗口的管理者，系统中所有的窗口状态和信息都在其掌握之中，完成这一工作不在话下。

什么是 Window：

“Window”表明它是和窗口相关的，“窗口”是一个抽象的概念，从用户的角度来讲，它是一个“界面”；

从 SurfaceFlinger 的角度来看，它是一个 Layer，承载着和界面有关的数据和属性；（SurfaceFlinger 是 Android multimedia（多媒体）的一个部分，在 Android 的实现中它是一个 service，提供系统 范围内的 surface composer 功能，它能够将各种应用 程序的 2D、3D surface 进行组合。）

从 WMS 角度来看，它是一个 WIndowState，用于管理和界面有关的状态。

比喻：

整个界面就像由 N 个演员参与的话剧：SurfaceFling 是摄像机，它只负责客观的捕捉当前的画面，然后真实的呈现给观众；

WMS 就是导演，它要负责话剧的舞台效果、演员站位；

ViewRoot 就是各个演员的长相和表情，取决于它们各自的条件与努力。

可见，WMS 与 SurfaceFling 的一个重要区别就是——后者只做与“显示”相关的事情，而 WMS 要处理对输入事件的派发。

Android 支持的窗口类型很多，统一可以分为三大类，另外各个种类下还细分为若干子类型，且都在 WindowManager.java 中有定义。

1) Application Window

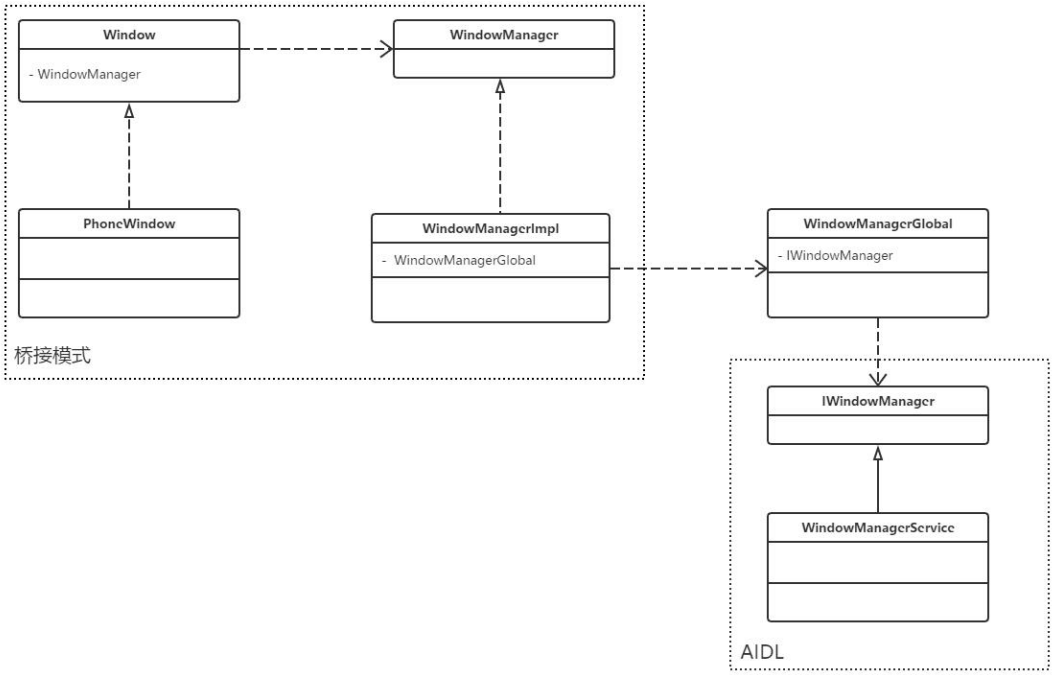
2) SystemWindow

3) Sub Window

使用的设计模式：

桥接模式

http://www.runoob.com/design-pattern/bridge-pattern.html



解惑：

Q：WMS 是系统服务，有 SystemServer 负责启动，启动时机相对较晚，那么在 WMS 运行之前，终端显示屏就一团黑？

A：在 WMS 启动之前，系统只需显示开机动画，它们都有特殊的方式来向屏幕输出图像，比如直接通过 OpenGL ES 与 SurfaceFling 的配合来完成。这也从侧面告诉我们，要想在 Android 上显示 UI，并不一定要通过 WMS。

AMS（ActivityManagerService）

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

作用：

1）统一调度所有应用程序的 Activity 的生命周期

2）启动或杀死应用程序的进程

3）启动并调度 Service 的生命周期

4）注册 BroadcastReceiver，并接收和分发 Broadcast

5）启动并发布 ContentProvider

6）调度 task

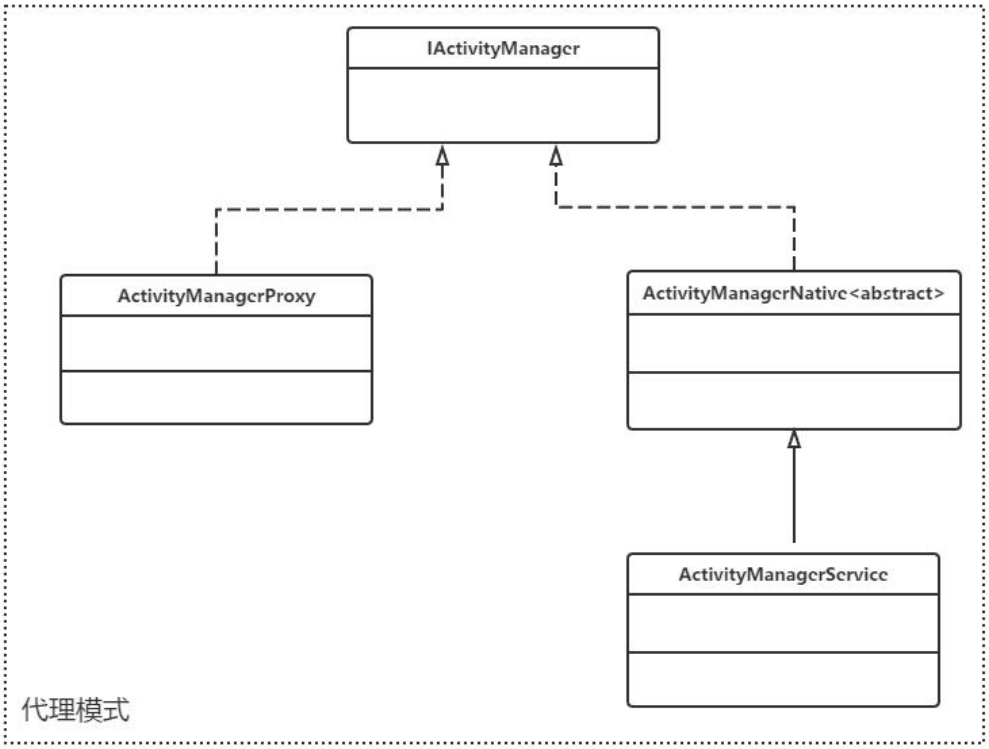
7）处理应用程序的 Crash

8）查询系统当前运行状态

设计模式：

代理模式

http://www.runoob.com/design-pattern/proxy-pattern.html



从类图可以看出，ActivityManagerProxy 和 ActivityManagerNative 都实现了 IActivityManager，ActivityManagerProxy 就是代理部分，ActivityManagerNative 就是实现部分，但 ActivityManagerNative 是个抽象类，并不处理过多的具体逻辑，大部分具体逻辑是由 ActivityManagerService 承担，这就是为什么我们说真实部分应该为 ActivityManagerService。

代码逻辑：

启动流程：

第一阶段：启动 ActivityManagerService。

第二阶段：调用 setSystemProcess。

第三阶段：调用 installSystemProviders 方法。

第四阶段：调用 systemReady 方法。

工作流程:

AMS 的工作流程，其实就是 Activity 的启动和调度的过程，所有的启动方式，最终都是通过 Binder 机制的 Client 端，调用 Server 端的 AMS 的 startActivityXXX()系列方法。所以可见，工作流程又包括 Client 端和 Server 端两个。

Client 端流程：

Launcher 主线程捕获 onClick()点击事件后，调用 Launcher.startActivitySafely()方法。Launcher.startActivitySafely()内部调用了 Launcher.startActivity()方法，Launcher.startActivity()内部调用了 Launcher 的父类 Activity 的 startActivity()方法。

Activity.startActivity()调用 Activity.startActivityForResult()方法，传入该方法的 requestCode 参数若为-1，则表示 Activity 启动成功后，不需要执行 Launcher.onActivityResult()方法处理返回结果。

启动 Activity 需要与系统 ActivityManagerService 交互，必须纳入 Instrumentation 的监控，因此需要将启动请求转交 instrumentation，即调用 Instrumentation.execStartActivity()方法。

Instrumentation.execStartActivity()首先通过 ActivityMonitor 检查启动请求，然后调用 ActivityManagerNative.getDefault()得到 ActivityManagerProxy

代理对象，进而调用该代理对象的 startActivity()方法。

ActivityManagerProxy 是 ActivityManagerService 的代理对象，因此其内部存储的是 BinderProxy，调用 ActivityManagerProxy.startActivity() 实质是调用 BinderProxy.transact() 向 Binder 驱动发送 START_ACTIVITY_TRANSACTION 命令。Binder 驱动将处理逻辑从 Launcher 所在进程切换到 ActivityManagerService 所在进程。

Server 端流程：

启动 Activity 的请求从 Client 端传递给 Server 端后，便进入了启动应用的七个阶段，这里也是整理出具体流程。

1）预启动

ActivityManagerService.startActivity()

ActivityStack.startActivityMayWait()

ActivityStack.startActivityLocked()

ActivityStack.startActivityUncheckedLocked()

ActivityStack.startActivityLocked()（重载）

ActivityStack.resumeTopActivityLocked()

2）暂停

ActivityStack.startPausingLocked()

ApplicationThreadProxy.schedulePauseActivity()

ActivityThread.handlePauseActivity()

ActivityThread.performPauseActivity()

ActivityManagerProxy.activityPaused()

completePausedLocked()

3）启动应用程序进程

第二次进入 ActivityStack.resumeTopActivityLocked()

ActivityStack.startSpecificActivityLocked()

startProcessLocked()

startProcessLocked()（重载）

Process.start()

4）加载应用程序 Activity

ActivityThread.main()

ActivityThread.attach()

ActivityManagerService.attachApplication()

ApplicationThread.bindApplication()

ActivityThread.handleBindApplication()

5）显示 Activity

ActivityStack.realStartActivityLocked()

ApplicationThread.scheduleLaunchActivity()

ActivityThead.handleLaunchActivity()

ActivityThread.performLaunchActivity()

ActivityThread.handleResumeActivity()

ActivityThread.performResumeActivity()

Activity.performResume()

ActivityStack.completeResumeLocked()

6）Activity Idle 状态的处理

7）停止源 Activity

ActivityStack.stopActivityLocked()

ApplicationThreadProxy.scheduleStopActivity()

ActivityThread.handleStopActivity()

ActivityThread.performStopActivityInner()

2. 对 Dalvik、ART 虚拟机有什么了解？

3. Art 和 Dalvik 对比

https://www.jianshu.com/p/58f817d176b7

Dalivk 与 ART 虚拟机

Dalvik

Dalvik 是 Google 公司自己设计用于 Android 平台的虚拟机。它可以支持已转换为** .dex 格式**的 Java 应用程序的运行，.dex 格式是专 Dalvik 设计的一种压缩格式，适合内存和处理器速度有限的系统。

Dalvik 经过优化，允许在有限的内存中同时运行多个虚拟机的实例，并且每一个 Dalvik 应用作为一个独立的 Linux 进程执行。独立的进程可以防止在虚拟机崩溃的时候所有程序都被关闭。

在程序运行过程中 Dalvik 虚拟机不断的进行将字节码转换为机器码的工作。

ART

即 Android Runtime。从 Android L (安卓 5.1.1)开始，Android 开始启用了新设计的虚拟机 ART。

ART 的机制与 Dalvik 不同。在 Dalvik 下，应用每次运行的时候，字节码都需要通过即时编译器（just in time ，JIT）转换为机器码，这会拖慢应用的运行效率，而在 ART 环境中，应用在第一次安装的时候，字节码就会预先编译成机器码，极大的提高了程序的运行效率，同时减少了手机的耗电量，使其成为真正的本地应用。这个过程叫做预编译（AOT,Ahead-Of-Time）。这样的话，应用的启动(首次)和执行都会变得更加快速。

采用 AOT 策略后的好处显而易见，应用的启动速度会因此快很多，但是与此同时，应用的安装时间就会因为执行 AOT 操作而变长，但是相比之下还是非常值得。

另外，ART 的另一个缺点就是对存储空间占用变大。Art 模式下应用程序的安装需要消耗更多的时间，同时也需要跟多的安装空间。一般的字节码在编译转码后占用的空间大小比之前要增大 10%-20%。

除了 AOT 机制，ART 另外一个显著的提升就是垃圾回收方面的提升。相比 Dalvik 虚拟机，ART 虚拟机具有更高的回收性能。

优点：

1、系统性能的显著提升。

2、应用启动更快、运行更快、体验更流畅、触感反馈更及时。

3、更长的电池续航能力。

4、支持更低的硬件。

缺点：

1.机器码占用的存储空间更大，字节码变为机器码之后，可能会增加 10%-20%（不过在应用包中，可执行的代码常常只是一部分。比如最新的 Google+ APK 是 28.3 MB，但是代码只有 6.9 MB。）

2.应用的安装时间会变长。

Dalvik 和 JVM 有啥关系？

主要区别：

Dalvik 是基于寄存器的，而 JVM 是基于栈的。

Dalvik 运行 dex 文件，而 JVM 运行 java 字节码

自 Android 2.2 开始，Dalvik 支持 JIT（just-in-time，即时编译技术）。

优化后的 Dalvik 较其他标准虚拟机存在一些**不同特性**:

1.占用更少空间

2.为简化翻译，常量池只使用 32 位索引

3.标准 Java 字节码实行 8 位堆栈指令,Dalvik 使用 16 位指令集直接作用于局部变量。局部变量通常来自 4 位的“虚拟寄存器”区。这样减少了 Dalvik 的指令计数，提高了翻译速度。

当 Android 启动时，Dalvik VM 监视所有的程序（APK），并且创建依存关系树，为每个程序优化代码并存储在 Dalvik 缓存中。Dalvik 第一次加载后会生成 Cache 文件，以提供下次快速加载，所以第一次会很慢。

Dalvik 解释器采用预先算好的 Goto 地址,每个指令对内存的访问都在 64 字节边界上对齐。这样可以节省一个指令后进行查表的时间。为了强化功能, Dalvik 还提供了快速翻译器（Fast Interpreter）。

总结

Dalvik 其实可以理解为一个专为移动设备优化过的 JVM，它的大部分地方都遵守了 JVM 规范，其实那些不符合规范的地方，就可以理解为为移动设备做的优化工作。而 ART 是一个具有更高性能的 Android 虚拟机，从一开始他就是为取代 Dalvik 而来，它的 AOT 机制相比 Dalvik 的 JIT 机制使得应用有更快的启动速度。同时 ART 虚拟机在垃圾回收方面也比 Dalvik 更加高性能。

4. 虚拟机原理，如何自己设计一个虚拟机(内存管理，类加载，双亲委派)（暂无）

5. 谈谈你对双亲委派模型理解

https://www.jianshu.com/p/353c26c744df

双亲委派模型基本概念

定义：

双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。

双亲委派模型的工作过程是：

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时（搜索范围中没有找到所需的类），子加载器才会尝试自己去加载。

很多人对“双亲”一词很困惑。这是翻译的锅，，，“双亲”只是“parents”的直译，实际上并不表示汉语中的父母双亲，而是一代一代很多 parent，即 parents。

作用

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。因此，使用双亲委派模型来组织类加载器之间的关系，有一个显而易见的好处：类随着它的类加载器一起具备了一种带有优先级的层次关系。

（

例如类 java.lang.Object，它由启动类加载器加载。双亲委派模型保证任何类加载器收到的对 java.lang.Object 的加载请求，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此 Object 类在程序的各种类加载器环境中都是同一个类。

相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为 java.lang.Object 的类，并用自定义的类加载器加载，那系统中将会出现多个不同的 Object 类，Java 类型体系中最基础的行为也就无法保证，应用程序也将会变得

一片混乱。

）

结构

系统提供的类加载器

在双亲委派模型的定义中提到了“启动类加载器”。包括启动类加载器，绝大部分 Java 程序都会使用到以下 3 种系统提供的类加载器：

1）启动类加载器（Bootstrap ClassLoader）

负责将存放在＜JAVA_HOME＞/lib 目录中的，或者被-Xbootclasspath 参数所指定的路径中的，并且是虚拟机按照文件名识别的（如 rt.jar，名字不符合的类库即使放在 lib 目录中也不会被加载）类库加载到虚拟机内存中。

启动类加载器无法被 Java 程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器，那直接使用 null 代替即可。（JDK 中的常用类大都由启动类加载器加载，如 java.lang.String、java.util.List 等。需要特别说明的是，启动类 Main class 也由启动类加载器加载。）

2）扩展类加载器（Extension ClassLoader）

由 sun.misc.Launcher$ExtClassLoader 实现。

负责加载＜JAVA_HOME＞/lib/ext 目录中的，或者被 java.ext.dirs 系统变量所指定的路径中的所有类库。

开发者可以直接使用扩展类加载器。

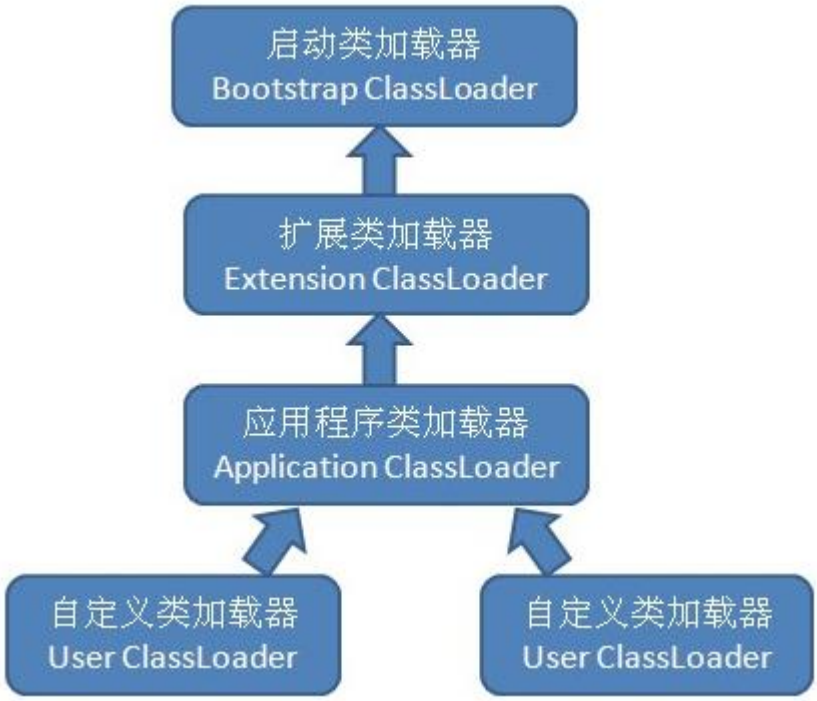
3) 应用程序类加载器（Application ClassLoader）

由 sun.misc.Launcher$AppClassLoader 实现。由于这个类加载器是 ClassLoader.getSystemClassLoader()方法的返回值，所以一般也称它为系统类加载器。

它负责加载用户类路径 ClassPath 上所指定的类库，开发者可以直接使用这个类加载器。如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

启动类 Main class、其他如工程中编写的类、maven 引用的类，都会被放置在类路径下。Main class 由启动类加载器加载，其他类由应用程序类加载器加载。

JVM 建议用户将应用程序类加载器作为自定义类加载器的父类加载器。则类加载的双亲委派模型如图：



实现原理

实现双亲委派的代码都集中在ClassLoader#loadClass()方法之中。将统计部分的代码去掉之后，简写如下：

```java
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException {
    synchronized (getClassLoadingLock(name)) {
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            ...
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
            }

            if (c == null) {
                ...
                c = findClass(name);
                // do some stats
                ...
            }
        }
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }
}
```

1) 首先，检查目标类是否已在当前类加载器的命名空间中加载（即，使用二元组<类加载器实例，全限定名>区分不同类）。
2) 如果没有找到，则尝试将请求委托给父类加载器（如果指定父类加载器为 null，则将启动类加载器作为父类加载器；如果没有指定父类加载器，则将应用程序类加载器作为父类加载器），最终所有类都会委托到启动类加载器。
3) 如果父类加载器加载失败，则自己加载。
4) 默认 resolve 取 false，不需要解析，直接返回。

6. Ubuntu 编译安卓系统

1）初始化编译环境

```
. build/envsetup.sh
```

2）选择编译目标

```
lunch aosp_arm64-eng
```

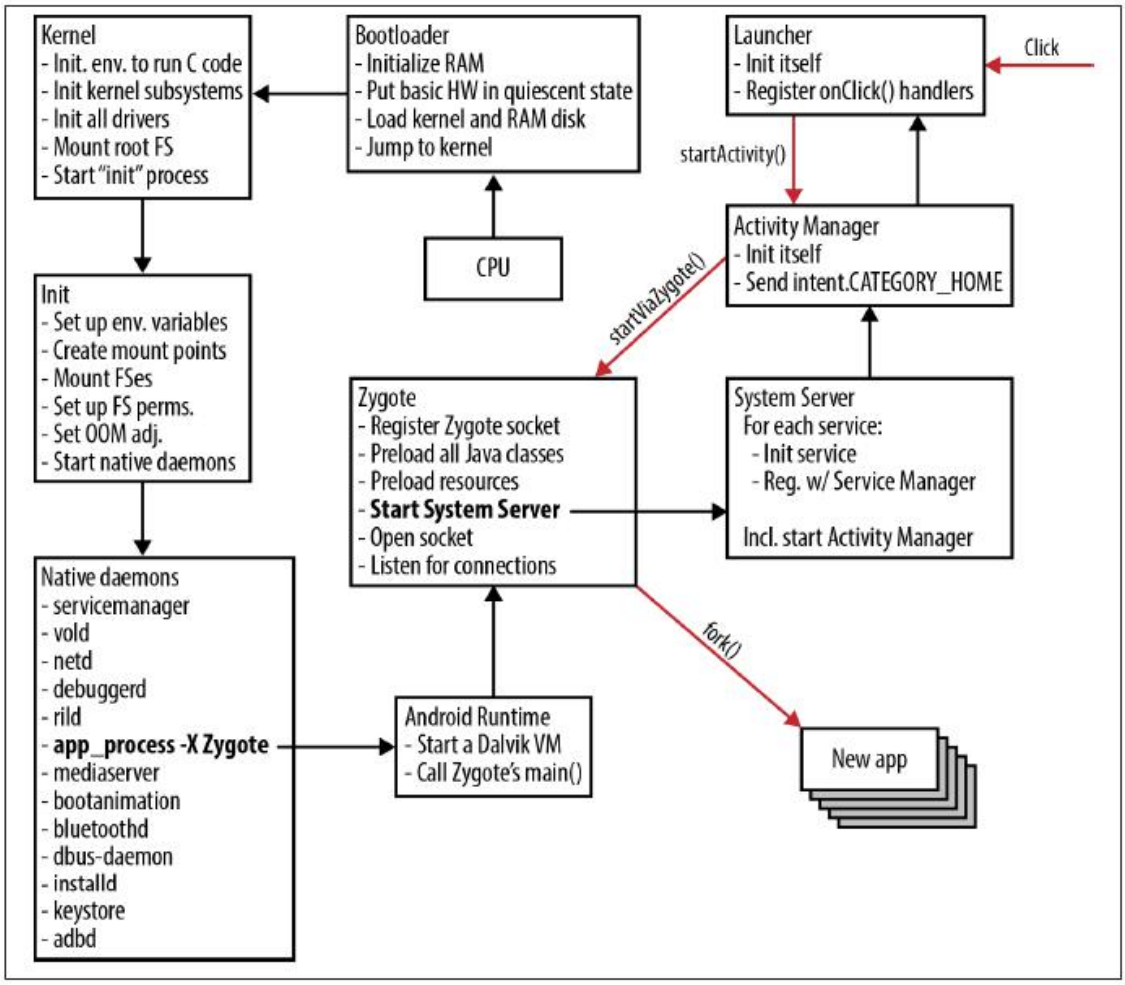
3）开始编译

```
make -j8
```

7. 系统启动流程是什么？（提示：Zygote 进程 -> SystemServer 进程 -> 各种系统服务 -> 应用进程）

https://www.jianshu.com/p/2ca0f6c974c9

对于 Android 系统整个启动过程来说，基本可以划分成三个阶段：**Bootloader 引导、Linux kernel 启动、Android 启动**。如下的流程图可以看出三个阶段的执行顺序和联系



(1)Bootloader 引导

BootLoader 是在操作系统运行之前运行的一段程序，它可以将系统的软硬件环境带到一个合适状态，为运行操作系统做好准备。它的任务确实不多，终极目标就是把系统拉起来运行。

(2)Linux kernel 启动

Linex kernel 启动后会初始化环境/驱动等，接着会启动 init 的进程。

1) Init 进程的启动

init 进程：它是一个由内核启动的用户级进程。

内核自行启动（已经被载入内存，开始运行，并已初始化所有的设备驱动程序和数据结构等）之后，就通过启动一个用户级程序 init 的方式，完成引导进程。

启动过程就是代码 init.c 中 main 函数执行过程：system\core\init\init.c

在函数中执行了：文件夹建立，挂载，rc 文件解析，属性设置，启动服务，执行动作，socket 监听……

在 rc 文件解析中会启动 servicemanager 服务和 Zygote 进程

2) servicemanager 服务

ServiceManager 用来管理系统中所有的 binder service，不管是本地的 c++实现的还是 java 语言实现的都需要这个进程来统一管理，最主要的管理就是，注册添加服务，获取服务。所有的 Service 使用前都必须先在 servicemanager 中进行注册。

3) Zygote 进程的启动

Zygote 这个进程起来才会建立起真正的 Android 运行空间。

不过 Init 进程不会直接启动 Zygote 进程，而是使用 app_process 命令来通过 Android Runtime 来启动，Android Runtime 会启动第一个 Davlik 虚拟机，并调用 Zygote 的 main 方法。

初始化建立的 Service 都是 Navtive service.在.rc 脚本文件中 zygote 的描述：

service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server

(3)Android 启动

1.Zygote 启动

Zygote 从 main(…)@frameworks/base/cmds/app_main.cpp 开始

(1)main(...)@frameworks/base/cmds/app_main.cpp

runtime.start("com.android.internal.os.ZygoteInit", startSystemServer); //建立 android runtime

(2)runtime.start("com.android.internal.os.ZygoteInit",...)@/frameworks/base/core/jni/AndroidRuntime.cpp //调用 java 的 ZygoteInit

(3)main()@com.android.internal.os.ZygoteInit //真正的 Zygote

在 com.android.internal.os.ZygoteInit 的 main 函数中至少可以看到

registerZygoteSocket(...);//登记 Listen 端口

startSystemServer(...); //启动 SystemServer 服务

2. SystemServer 启动

在上一步中 com.android.internal.os.ZygoteInit 的 main 函数中调用了 startSystemServer， Zygote 上 fork 了一个进程：com.android.server.SystemServer. 于是

SystemServer@(SystemServer.java）就建立了。

com.android.server.SystemServer 的 main 函数中会启动一些服务

{ startBootstrapServices();

startCoreServices();

startOtherServices(); }

在 startBootstrapServices()中看到

mActivityManagerService = mSystemServiceManager.startService(ActivityManagerService.Lifecycle.class).getService();

ActivityManagerService.Lifecycle 是 ActivityManagerService 的静态内部类， 持有 ActivityManagerService 实例。所以 ActivityManagerService 在这里被启动起来.

当然还有很多的其他服务（eg:PackageManagerService/PowerManagerService/DisplayManagerService/BatteryService 等）都会被启动起来。

在所有的服务启动完成后，会调用 Service.systemReady(...)来通知各对应的服务，系统已经就绪。

3.ActivityManagerService 启动

在 ActivityManagerService 的 systemReady()方法中

startHomeActivityLocked(...);

mStackSupervisor.startHomeActivity(...)

这样 Home 程序被启动起来，整个 android 的启动过程就完成了，进入用户界面。

8. 大体说清一个应用程序安装到手机上时发生了什么

https://cdn2.jianshu.io/p/2da052f4b5cd?utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation

(1)**拷贝 apk 文件到指定目录：**用户安装的 apk 首先会被拷贝到 /data/app 目录下

/data/app 目录是用户有权限访问的目录，在安装 apk 的时候会自动选择该目录存放用户安装的文件

系统出厂的 apk 文件则被放到了/system 分区下，包括 /system/app，/system/vendor/app，以及 /system/priv-app 等等，该分区只有 Root 权限的用户才能

(2)**解压 apk，拷贝文件，创建应用的数据目录：**为了加快 app 的启动速度，apk 在安装的时候，会首先将 app 的可执行文件（dex）

拷贝到 /data/dalvik-cache 目录，缓存起来。

然后，在/data/data/目录下创建应用程序的数据目录（以应用的包名命名），存放应用的相关数据，如数据库、xml 文件、cache、二进制的 so 动态库等等。

(3)**解析 apk 的 AndroidManifinest.xml 文件：**解析的内容会被存储到/data/system/packages.xml 和/data/system/packages.list 中。

packages.list：中指名了该应用默认存储的位置/data/data/cn.hadcn.example。

packages.xml：中包含了该应用申请的权限、签名和代码所在位置等信息，并且两者都有一个 userId 为 10060。之所以每个应用都有一个 userId，是因为 Android 在系统设计上把每个应用当作 Linux 系统上的一个用户对待，这样就可以利用已有的 Linux 上用户管理机制来设计 Android 应用，比如应用目录，应用权限，应用进程管理等。

做完以上操作，就相当于应用在系统注册了，可以被系统识别。

注：目录是由 包名-1 组成，有时候此处是 -2。这是为了升级使用，升级时会新创建一个-1 或 -2 的目录，如果升级成功，则删除原目录并更改 packages.xml 中 codePath 到新目录

在 Dalvik 模式下，会使用 dexopt 把 base.apk 中的 dex 文件优化为 odex，存储在/data/dalvik-cache 中，

如果是 ART 模式，则会使用 dex2oat 优化成 oat 文件也存储在该

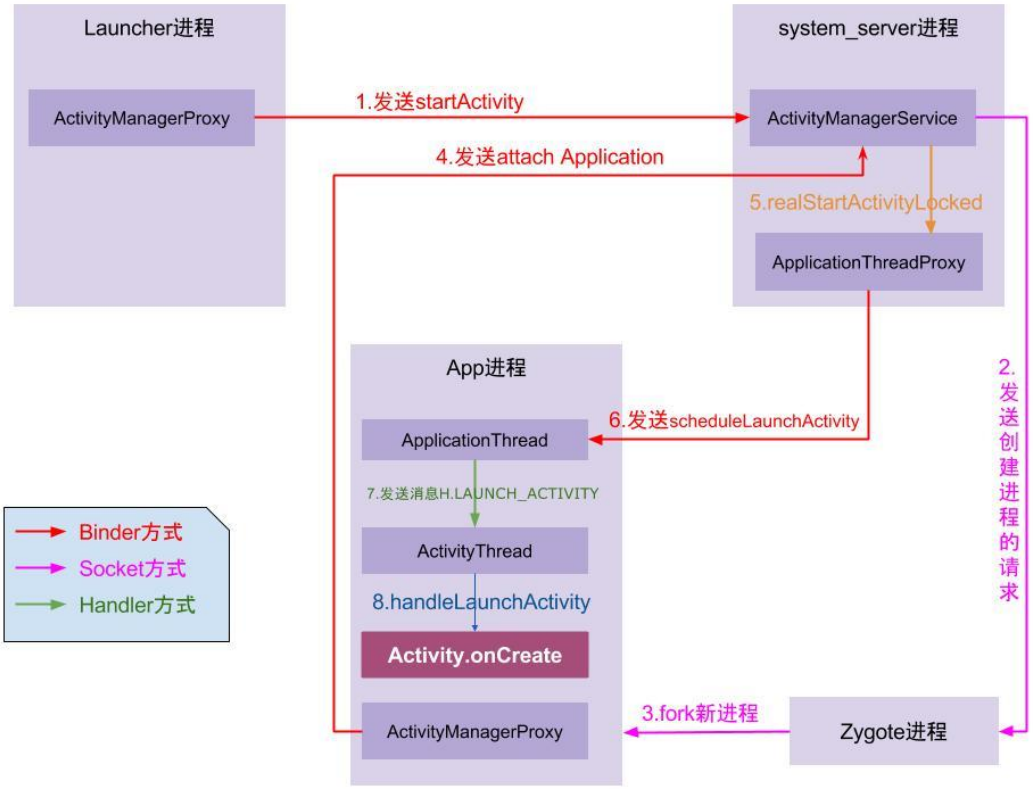
目录下，并且文件名一样，但文件大小会大很多，因为 ART 模式会在安装时把 dex 优化为机器码，所以在 ART 模式下的应用运行更快，但 apk 安装速度相对 Dalvik 模式下变慢，并且会占用更多的 ROM。

优化后的 dex 文件被载入到虚拟机中就可以运行。

9. App 启动流程，从点击桌面开始

https://blog.csdn.net/zhencheng20082009/article/details/78218907

https://www.jianshu.com/p/8f0ceb55f590



启动流程：

①点击桌面 App 图标，Launcher 进程采用 Binder IPC 向 system_server 进程发起 startActivity 请求；

②system_server 进程接收到请求后，向 zygote 进程发送创建进程的请求；

③Zygote 进程 fork 出新的子进程，即 App 进程；

④App 进程，通过 Binder IPC 向 sytem_server 进程发起 attachApplication 请求；

⑤system_server 进程在收到请求后，进行一系列准备工作后，再通过 binder IPC 向 App 进程发送 scheduleLaunchActivity 请求；

⑥App 进程的 binder 线程（ApplicationThread）在收到请求后，通过 handler 向主线程发送 LAUNCH_ACTIVITY 消息；

⑦主线程在收到 Message 后，通过发射机制创建目标 Activity，并回调 Activity.onCreate()等方法。

⑧到此，App 便正式启动，开始进入 Activity 生命周期，执行完 onCreate/onStart/onResume 方法，UI 渲染结束后便可以看到 App 的主界面。

Instrumentation 和 ActivityThread

每个 Activity 都持有 Instrumentation 对象的一个引用，但是整个进程只会存在一个 Instrumentation 对象。

Instrumentation 这个类里面的方法大多数和 Application 和 Activity 有关，**这个类就是完成对 Application 和 Activity 初始化和生命周期的工具类**。Instrumentation 这个类很重要，对 Activity 生命周期方法的调用根本就离不开他，他可以说是一个大管家。

ActivityThread，依赖于 UI 线程。App 和 AMS 是通过 Binder 传递信息的，那么 ActivityThread 就是专门与 AMS 的外交工作的。

启动流程

(1)创建进程：

①先从 Launcher 的 startActivity()方法，通过 Binder 通信，调用 ActivityManagerService 的 startActivity 方法。

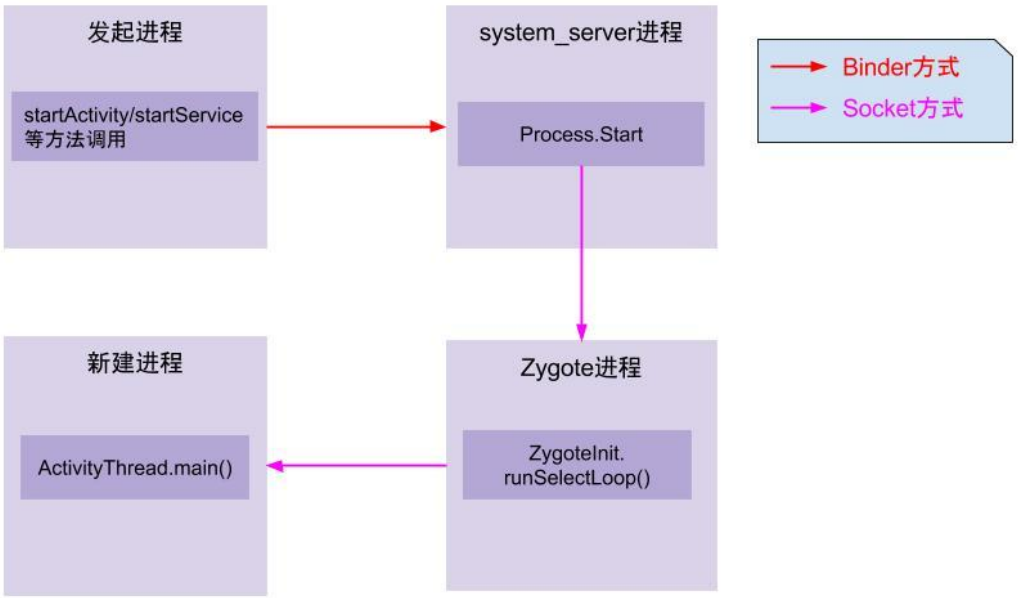
②一系列折腾，最后调用 startProcessLocked()方法来创建新的进程。

③该方法会通过前面讲到的 socket 通道传递参数给 Zygote 进

程。Zygote 孵化自身。调用 ZygoteInit.main()方法来实例化 ActivityThread 对象并最终返回新进程的 pid。

④调用 ActivityThread.main()方法，ActivityThread 随后依次调用 Looper.prepareLoop()和 Looper.loop()来开启消息循环。

更直白的流程解释：



①App 发起进程：当从桌面启动应用，则发起进程便是 Launcher 所在进程；当从某 App 内启动远程进程，则发送进程便是该 App 所在进程。发起进程先通过 binder 发送消息给 system_server 进程；

②system_server 进程：调用 Process.start()方法，通过 socket 向 zygote 进程发送创建新进程的请求；

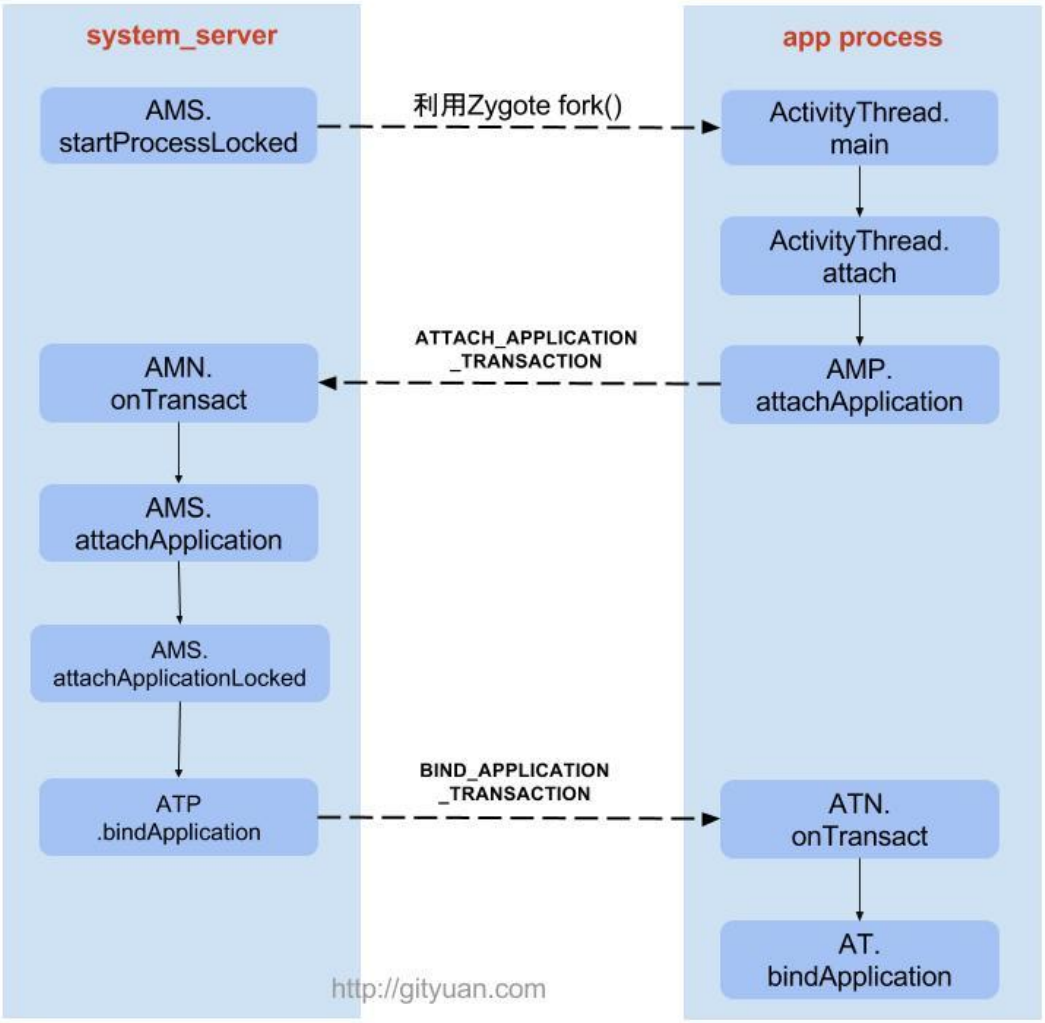
③ zygote 进程： 在执行 ZygoteInit.main() 后便进入 runSelectLoop() 循环体内， 当有客户端连接时便会执行 ZygoteConnection.runOnce()方法， 再经过层层调用后 fork 出新的应用进程；

④新进程：执行handleChildProc方法，最后调用ActivityThread.main()方法。

(2)绑定Application：

1. 上面创建进程后，执行ActivityThread.main()方法，随后调用attach()方法。
2. 将进程和指定的Application绑定起来。这个是通过上节的ActivityThread对象中调用bindApplication()方法完成的。该方法发送一个BIND_APPLICATION的消息到消息队列中，最终通过handleBindApplication()方法处理该消息。然后调用makeApplication()方法来加载App的classes到内存中。

更直白的流程解释：



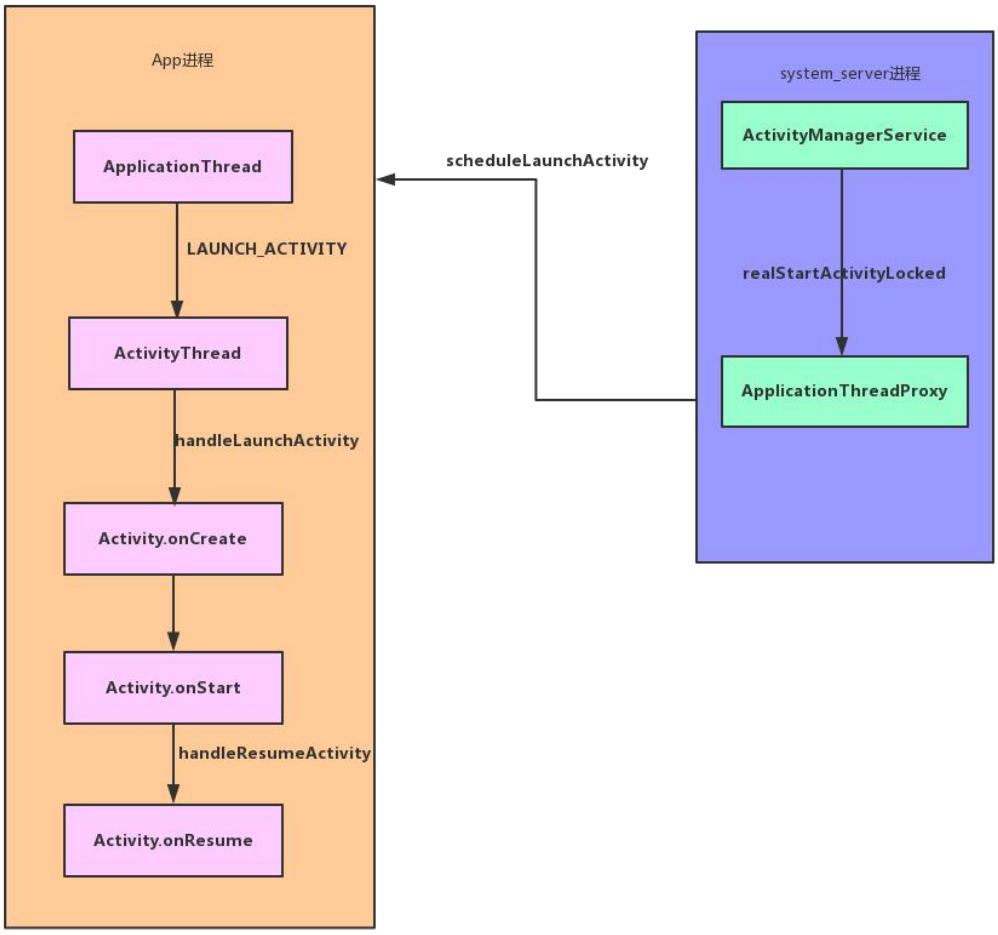
(3)显示 Activity 界面：

经过前两个步骤之后，系统已经拥有了该 application 的进程。后面的调用顺序就是普通的从一个已经存在的进程中启动一个新进程的 activity 了。

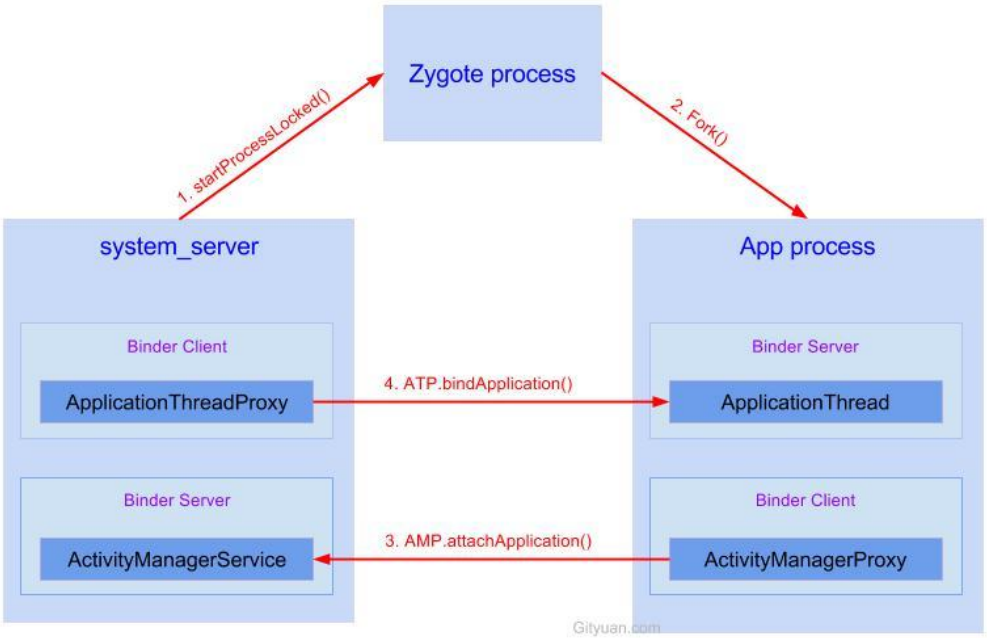
实际调用方法是 realStartActivity()，它会调用 application 线程对象中的 scheduleLaunchActivity() 发送一个 LAUNCH_ACTIVITY 消息到消息队列中，通过 handleLaunchActivity() 来处理该消息。

在 handleLaunchActivity()通过 performLaunchActiivty()方

法回调 Activity 的 onCreate()方法和 onStart()方法，然后通过 handleResumeActivity()方法，回调 Activity 的 onResume()方法，最终显示 Activity 界面。



Binder 通信



10. 逻辑地址与物理地址，为什么使用逻辑地址？

逻辑地址：存储单元的地址可以用段基值和段内偏移量来表示，段基值确定它所在的段居于整个存储空间的位置，偏移量确定它在段内的位置，这种地址表示方式称为逻辑地址。8086 体系的 CPU 一开始是 20 根地址线， 寻址寄存器是 16 位， 16 位的寄存器可以访问 64K 的地址空间， 如果程序要想访问大于 64K 的内存， 就要把内存分段，每段 64K， 用段地址+偏移量的方法来访问 。386CPU 出来之后， 采用了 32 条地址线， 地址寄存器也扩为 32 位， 这样就可以不用分段了， 直接用一个地址寄存器来线性访问 4G 的内存了. 这就叫平面模式.

线性地址：又叫虚拟地址，是一个 32 位无符号整数，可以用来表示高达 4GB 的地址，跟逻辑地址类似，它也是一个不真实的地址，如果逻辑地址是对应的硬件平台段式管理转换前地址的话，那么线性地址则对应了硬件页式内存的转换前地址。

物理地址：用于内存芯片级的单元寻址，与处理器和 CPU 连接的地址总线相对应。

注意：

1.CPU 将一个虚拟内存空间中的地址转换为物理地址，需要进行两步：首先将给定一个逻辑地址（其实是段内偏移量，这个一定要理解！！！），CPU 要利用其段式内存管理单元，先将为个逻辑地址转换成一个线性地址，再利用其页式内存管理单元，转换为最终物理地址。

逻辑地址----段式内存管理单元----线性地址----页式内存管理单元----物理地址

这样做两次转换，的确是非常麻烦而且没有必要的，因为直接可以把线性地址抽像给进程。之所以这样冗余，Intel 完全是为了兼容而已。

11. Android 为每个应用程序分配的内

存大小是多少？

https://blog.csdn.net/u011506413/article/details/50965435

准确的说话是 google 原生 OS 的默认值是 16M，但是各个厂家的系统会对这个值进行修改。不同厂商的值不同

（1）未设定属性 android:largeheap = "true"时，可以申请到的最大内存空间。

（2）设定属性 android:largeheap = "true"时， 可以申请的最大内存空间为原来的两倍多一些。

先看机器的内存限制，在/system/build.prop 文件中：

heapgrowthlimit 就是一个普通应用的内存限制， 用 ActivityManager.getLargeMemoryClass()获得的值就是这个。

而 heapsize 是在 manifest 中设置了 largeHeap=true 之后，可以使用的最大内存值

结论就是，设置 largeHeap 的确可以增加内存的申请量。但不是系统有多少内存就可以申请多少，而是由 dalvik.vm.heapsize 限制。

你可以在 app manifest.xml 加 largetHeap=true

可以申请较多的内存，但还是有机会爆掉

<application

.....

android:label="XXXXXXXXXX"

```
            android:largeHeap="true">
        .......
</application>
```

cat /system/build.prop　　//读取这些值

getprop dalvik.vm.heapsize　　// 如果 build.prop 里面没有 heapsize 这些值，可以用这个抓取默认值

setprop dalvik.vm.heapsize 256m　　//设置

12. Android 中进程内存的分配，能不能自己分配定额内存？

https://www.cnblogs.com/yaowen/p/6347682.html

同上

13. 进程保活的方式

https://www.jianshu.com/p/53c4d8303e19

进程保活的关键点有两个，一个是进程优先级的理解，优先级越高存活几率越大。二是弄清楚哪些场景会导致进程会 kill，然后采取下面的策略对各种场景进行优化：

(1)提高进程的优先级

(2)在进程被 kill 之后能够唤醒

进程优先级：

Android 一般的进程优先级划分：

1.前台进程 (Foreground process)

2.可见进程 (Visible process)

3.服务进程 (Service process)

4.后台进程 (Background process)

5.空进程 (Empty process)

这是一种粗略的划分，进程其实有一种具体的数值，称作 oom_adj，

注意：数值越大优先级越低：

ADJ级别	取值	解释
UNKNOWN_ADJ	16	一般指将要会缓存进程，无法获取确定值
CACHED_APP_MAX_ADJ	15	不可见进程的adj最大值 1
CACHED_APP_MIN_ADJ	9	不可见进程的adj最小值 2
SERVICE_B_AD	8	B List中的Service（较老的、使用可能性更小）
PREVIOUS_APP_ADJ	7	上一个App的进程(往往通过按返回键)
HOME_APP_ADJ	6	Home进程
SERVICE_ADJ	5	服务进程(Service process)
HEAVY_WEIGHT_APP_ADJ	4	后台的重量级进程，system/rootdir/init.rc文件中设置
BACKUP_APP_ADJ	3	备份进程
PERCEPTIBLE_APP_ADJ	2	可感知进程，比如后台音乐播放
VISIBLE_APP_ADJ	1	可见进程(Visible process)
FOREGROUND_APP_ADJ	0	前台进程（Foreground process）
PERSISTENT_SERVICE_ADJ	-11	关联着系统或persistent进程
PERSISTENT_PROC_ADJ	-12	系统persistent进程，比如telephony
SYSTEM_ADJ	-16	系统进程
NATIVE_ADJ	-17	native进程（不被系统管理）

- 红色部分是容易被回收的进程，属于 android 进程
- 绿色部分是较难被回收的进程，属于 android 进程
- 其他部分则不是 android 进程，也不会被系统回收，一般是 ROM 自带的 app 和服务才能拥有

```
root@hwH60:/proc # cat 28321/oom_adj
cat 28321/oom_adj
0
root@hwH60:/proc # cat 28321/oom_adj
cat 28321/oom_adj
6
root@hwH60:/proc #
```

可见，当 app 在前台时 oom_adj = 0，对应上面的表格是前台进程。

当 app 退到后台时，oom_adj = 6，对应后台进程。

进程被 kill 的场景

(1)点击 home 键使 app 长时间停留在后台，内存不足被 kill

处理这种情况前提是你的 app 至少运行了一个 service，然后通过 Service.startForeground() 设置为前台服务，可以将 oom_adj 的数值由 4 降低到 1，大大提高存活率。

要注意的是 android4.3 之后 Service.startForeground() 会强制弹出通知栏，解决办法是再启动一个 service 和推送共用一个通知栏，然后 stop 这个 service 使得通知栏消失。

Android 7.1 之后 google 修复这个 bug，目前没有解决办法

(2)在大多数国产手机下，进入锁屏状态一段时间，省电机制会 kill 后台进程

这种情况和上面不太一样，是很过国产手机 rom 自带的优化，当锁屏一段时间之后，即使手机内存够用为了省电，也会释放掉一部分内存。

策略：

1) 注册广播监听锁屏和解锁事件， 锁屏后启动一个 1 像素的透明 Activity,这样直接把进程的 oom_adj 数值降低到 0,0 是 android 进程的最高优先级。 解锁后销毁这个透明 Activity。这里我把这个 Activity 放到:remote 进程也就是我那个后台服务进程，当然你也可以放到主进程，看你打算保活哪个进程。

```
<activity android:name=".application.KeepLiveActivity"
    android:process=":remove"
    android:theme="@style/LiveActivityStyle"
    android:excludeFromRecents="true"
    android:exported="false"
    android:finishOnTaskLaunch="false"
    android:launchMode="singleInstance"/>
```

我们可以写一个 KeepLiveManager 来负责接收广播，维护这个 Activity 的常见和销毁，注意锁屏广播和解锁分别是：ACTION_SCREEN_OOF 和 ACTION_USER_PRESENT，并且只能通过动态注册来绑定，并且是绑定到你的后台 service 里面,onCreate 绑定,onDestroy 里面解绑

2) 锁屏后循环播放无声音乐，解释后停止播放，实测在华为能永久保活，我们项目中就有用到。

(3)用户手动释放内存：包括手机自带清理工具，和第三方 app（360，猎豹清理大师等）

清理内存软件会把 优先级低于 前台进程(oom_adj = 0)的所有进程放入清理列表，而当我们打开了清理软件就意味着其他 app 不可能处于前台。所以说理论上可以 kill 任何 app。

因此这类场景唯一的处理办法就是加入 手机 rom 白名单，比如你打开小米，魅族的权限管理 -> 自启动管理可以看到 QQ，微信，天猫默认被勾选，这就是厂商合作。

那我们普通 app 可以这么做：在 app 的设置界面加一个选项，提

示用户自己去勾选自启动，我封装了一个工具类给出国内各厂商的自启动的 Intent 跳转方法：

```
switch (brand.toLowerCase()){
    case "samsung":
        componentName = new ComponentName("com.samsung.android.sm",
                "com.samsung.android.sm.app.dashboard.SmartManagerDashBoardA
        break;
    case "huawei":
        componentName = new ComponentName("com.huawei.systemmanager",
                "com.huawei.systemmanager.startupmgr.ui.StartupNormalAppList
        break;
    case "xiaomi":
        componentName = new ComponentName("com.miui.securitycenter",
                "com.miui.permcenter.autostart.AutoStartManagementActivity")
        break;
    case "vivo":
        componentName = new ComponentName("com.iqoo.secure",
                "com.iqoo.secure.ui.phoneoptimize.AddWhiteListActivity");
        break;
    case "oppo":
        componentName = new ComponentName("com.coloros.oppoguardelf",
                "com.coloros.powermanager.fuelgaue.PowerUsageModelActivity")
        break;
    case "360":
        componentName = new ComponentName("com.yulong.android.coolsafe",
                "com.yulong.android.coolsafe.ui.activity.autorun.AutoRunList
        break;
    case "meizu":
        componentName = new ComponentName("com.meizu.safe",
                "com.meizu.safe.permission.SmartBGActivity");
        break;
    case "oneplus":
```

进程唤醒

分两种情况，一是主进程（含有 Activity 没有 service），这种进程由于内存不足被 kill 之后，用户再次打开 app 系统会恢复到上次的 Acti vity，这个不在本文话题之内。另一种是 service 的后台进程被 kill，可以通过 service 自有 api 来重启 service：

```
@Override

    public int onStartCommand(Intent intent, int flags, int startId) {

        //.....

        return START_STICKY;    // service 被异常停止后，系统尝试重启 service，不能保证 100%重启成功

    }
```

但它不是 100%保证重启成功，比如下面 2 种情况：

Service 第一次被异常杀死后会在 5 秒内重启，第二次被杀死会在 10 秒内重启，第三次会在 20 秒内重启，一旦在短时间内 Service 被杀死达到 5 次，则系统不再拉起。

进程被取得 Root 权限的管理工具或系统工具通过 forestop 停止掉，无法重启。

总结

本文通过两种 提高进程优先级的方法，针对锁屏 和非锁屏模式下进程在后台被 kill 的场景处理，把后台进程优先级提升到可见级别，基本可以保证绝大多数场景不会被 kill。另外，针对含有 service 的进程被 kill 给出了可唤醒的办法。

14. 如何保证一个后台服务不被杀死？（相同问题：如何保证 service 在后台不被 kill？）比较省电的方式是什么？

(1) **用 sticky 粘性的 servcie:**Service 设置成 START_STICKY kill 后会被重启（等待 5 秒左右），重传 Intent，保持与重启前一样

(2) **多媒体锁：**锁屏后循环播放无声音乐，解释后停止播放，实测在华为能永久保活，我们项目中就有用到。

(3) **在 onDestroy 中再次启动:**service +broadcast 方式，就是当 service 走 ondestory 的时候，发送一个自定义的广播，当收到广播的时候，重新启动 service;

(4) **提高进程优先级，将其设置为前台状态：**通过 startForeground 将进程设置为前台进程，做前台服务，优先级和前台应用一个级别 ，除非在系统内存非常缺，否则此进程不会被 kill

(5) **用广播保持监听:**注册高频率广播接收器，唤起进程。如网络变化，解锁屏幕，开机等

(6) **双进程 Service：** 让 2 个进程互相保护，其中一个 Service 被清理后，另外没被清理的进程可以立即重启进程

（7） **1 像素存活**：在应用退到后台后，另起一个只有 1 像素的页面停留在桌面上，让自己保持前台状态，保护自己不被后台清理工具杀死

15. App 中唤醒其他进程的实现方式

https://blog.csdn.net/wapchief/article/details/79657026

唤起的进程：

1）自定义启动协议。

AndroidManifest.xml 中配置通过唤起启动的页面。

```
<!--唤醒app-->
<activity android:name=".SecondActivity"
android:screenOrientation="portrait"
android:theme="@style/ThemeSplash"
android:alwaysRetainTaskState="true"
android:launchMode="singleTask"
android:noHistory="true">
    <intent-filter android:autoVerify="true">
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <!-- 指定启动协议 -->
        <data android:scheme="wapchief" />
    </intent-filter>
</activity>
```

2）Activity 中获取其它进程传递的 Data 数据

```
Intent intent = getIntent();


Uri deeplink = intent.getData();
```

```
String result = deeplink.toString();
```

3） 正则匹配要跳转的界面

```

```

```
/**登录*/
public static final Pattern IS_LOGIN = Pattern.compile("wapcheif://login+");
/**注册*/
public static final Pattern IS_REGISTER = Pattern.compile("wapcheif://register+");

    if (IS_LOGIN.matcher(result).find()) {
        intent = new Intent(context, LoginActivity.class);
        context.startActivity(intent);

    }else if (IS_REGISTER.matcher(result).find()) {
        intent = new Intent(context, RegisterActivity.class);
        context.startActivity(intent);

    }
```

在调用方 APP 使用指定协议跳转

```
    mButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent action = new Intent(Intent.ACTION_VIEW, Uri.parse("wapcheif://register"));
            startActivity(action);
        }
    });
```