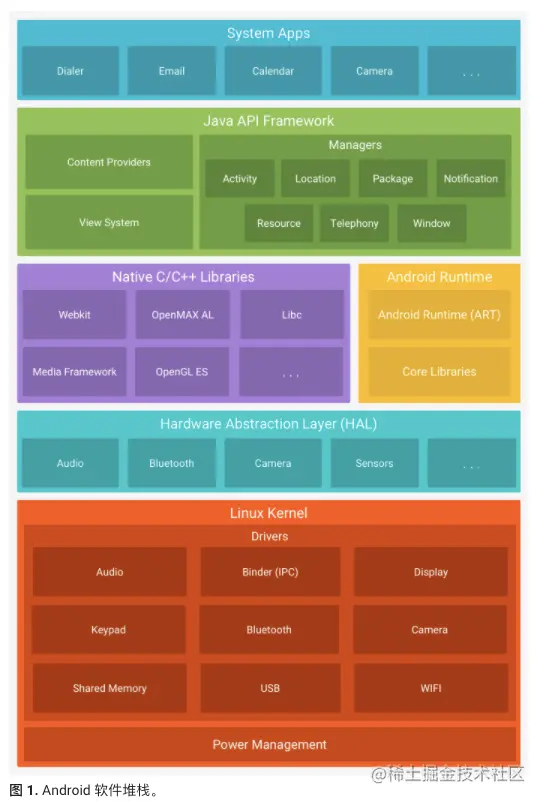
**Android系统启动流程分析**

**Android平台架构**



**1.Linux内核**

Android平台的基础的Linux内核。Android Runtime(ART)依靠Linux内核来执行底层功能，例如\*\*线程和底层内存管理。\*\*使用Linux内核可让Android利用[主要安全功能](https://link.juejin.cn/?target=http%3A%2F%2Fxiazai.zol.com.cn%2Fdetail%2F15%2F146785.shtml)(比如进程隔离，基于用户的权限模式等)，并且允许设备制造商为著名的内核开发硬件驱动程序。

**2.硬件抽象层(HAL)**

硬件抽象层提供标准界面，向更高级别的Java API框架显示设备硬件功能。\*\*HAL包含多个库模块，其中每个模块都为特定类型的硬件组件实现一个界面，例如相机或蓝牙模块。\*\*当框架API要求访问硬件设备时，Android系统将为该硬件组件加载库模块。

**3.Android Runtime**

Android5.0(API 21)以上，每个应用都运行在自己的进程中，有其自己的Android Runtime(ART)实例，ART编写为通过执行Dex文件在低内存设备上运行多个虚拟机，Dex文件是一种专为Android设计的字节码格式，经过优化使用的内存很少。编译工具链(例如Jack)将Java源代码编译为Dex字节码，使其可在Android平台上运行。

ART的部分主要功能包括：

* 预先(AOT)和即时(JIT)编译
* 优化的垃圾回收(GC)
* Android 9(API 28)及更高版本的系统中，支持将应用软件包中的Dalvik Executable格式(Dex)文件转化为更紧凑的机器代码。
* 更好的调试支持，包括专用采样分析器、详细的诊断异常和崩溃报告，并且能够设置观察点以监控特定字段。

在5.0之前，Dalvik是Android Runtime。如果您的应用在ART上运行效果很好，那么它应该也可在Dalvik上运行，但反过来不一定。 Android还包含一套**核心运行时库**，可提供Java API框架所使用的Java编程中年的大部分功能，包括一些Java8语言功能。

**4.原生C/C++库**

许多核心Android系统组件和服务(例如ART和HAL)构建自原生代码，需要以C和C++编写原生库。Android平台提供java框架API以向应用显示其中部分原生库的功能。比如可以通过Android框架的Java OpenGL API访问OpenGL ES，以支持在应用中绘制和操作2D/3D图形。

如果开发的是需要C或C++代码的应用，可以使用Android NDK直接从原生代码访问某些原生平台库。

**5.Java API框架**

您可通过以 Java 语言编写的 API 使用 Android OS 的整个功能集。这些 API 形成创建 Android 应用所需的构建块，它们可简化核心模块化系统组件和服务的重复使用，包括以下组件和服务：

* 丰富、可扩展的[视图系统](https://link.juejin.cn/?target=https%3A%2F%2Fdeveloper.android.com%2Fguide%2Ftopics%2Fui%2Foverview%3Fhl%3Dzh-cn)，可用以构建应用的 UI，包括列表、网格、文本框、按钮甚至可嵌入的网络浏览器
* [资源管理器](https://link.juejin.cn/?target=https%3A%2F%2Fdeveloper.android.com%2Fguide%2Ftopics%2Fresources%2Foverview%3Fhl%3Dzh-cn)，用于访问非代码资源，例如本地化的字符串、图形和布局文件
* [通知管理器](https://link.juejin.cn/?target=https%3A%2F%2Fdeveloper.android.com%2Fguide%2Ftopics%2Fui%2Fnotifiers%2Fnotifications%3Fhl%3Dzh-cn)，可让所有应用在状态栏中显示自定义提醒
* [Activity 管理器](https://link.juejin.cn/?target=https%3A%2F%2Fdeveloper.android.com%2Fguide%2Fcomponents%2Factivities%3Fhl%3Dzh-cn)，用于管理应用的生命周期，提供常见的[导航返回栈](https://link.juejin.cn/?target=https%3A%2F%2Fdeveloper.android.com%2Fguide%2Fcomponents%2Ftasks-and-back-stack%3Fhl%3Dzh-cn)
* [内容提供程序](https://link.juejin.cn/?target=https%3A%2F%2Fdeveloper.android.com%2Fguide%2Ftopics%2Fproviders%2Fcontent-providers%3Fhl%3Dzh-cn)，可让应用访问其他应用（例如“联系人”应用）中的数据或者共享其自己的数据

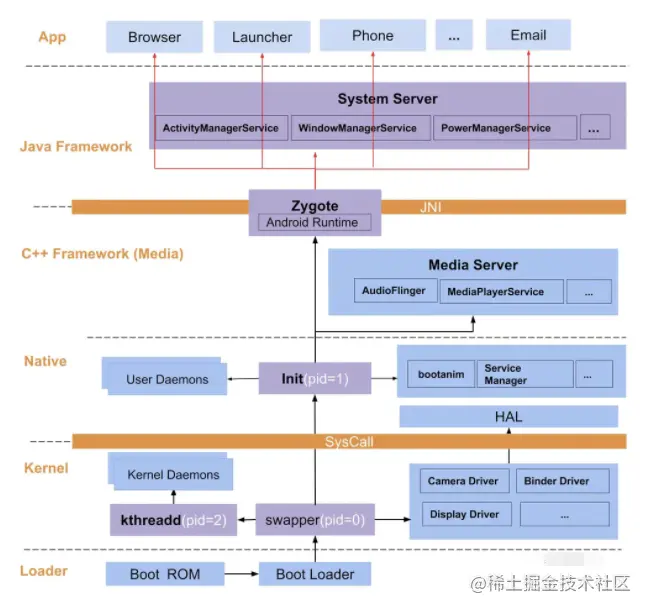
开发者可以完全访问 Android 系统应用使用的[框架 API](https://link.juejin.cn/?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fpackages%3Fhl%3Dzh-cn)。

**6.系统应用**

Android 随附一套用于电子邮件、短信、日历、互联网浏览和联系人等的核心应用。平台随附的应用与用户可以选择安装的应用一样，没有特殊状态。因此第三方应用可成为用户的默认网络浏览器、短信 Messenger 甚至默认键盘（有一些例外，例如系统的“设置”应用）。

系统应用可用作用户的应用，以及提供开发者可从其自己的应用访问的主要功能。例如，如果您的应用要发短信，您无需自己构建该功能，可以改为调用已安装的短信应用向您指定的接收者发送消息。

**Android系统启动的大概流程**



**Loader层**

1. 系统电源以及系统启动

当电源按下，引导芯片代码开始从预定义的地方(固化在ROM)开始执行。加载引导程序到RAM，然后执行。

1. 引导程序

引导程序是在Android操作系统开始运行前的一个小程序，是运行的第一个程序，因此它是针对特定的主板与芯片的，可以使用redboot,uboot,qibootloader或者开发自己的引导程序，**它不是Android操作系统的一部分**，引导程序是OEM厂商或者运营商加锁和限制的地方。

引导程序分两个阶段执行：

* 1. 检测外部的RAM以及加载对第二阶段有用的程序；
  2. 引导程序设置网络，内存等，这些对于运行内核是必要的，为了达到特殊的目标，引导程序可以根据配置参数或者输入数据设置内核。

Android引导程序可以在\bootable\bootloader\legacy\usbloader找到。传统的加载器包含两个文件，需要在这里说明： init.s初始化堆栈，清零BBS段，调用main.c的\_main()函数；

main.c初始化硬件(闹钟，主板，键盘，控制台)，创建linux标签

**Kernel层**

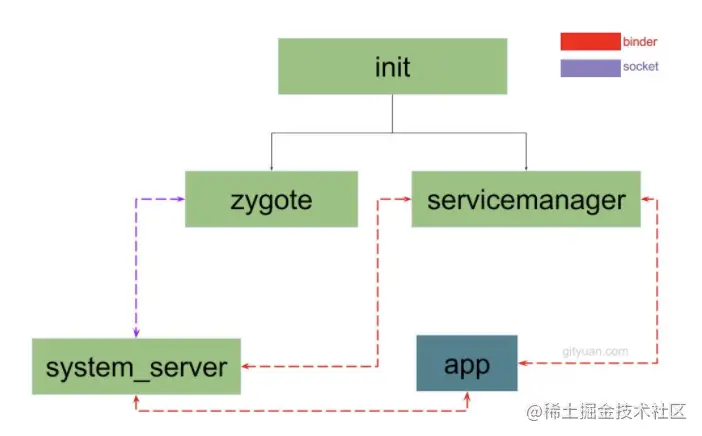
Kernel层指的就是Android内核层，这里一般开机刚结束进入Android系统，Kerner层的启动流程如下：

1. 启动swapper进程（pid=0），这是**系统初始化过程kernel创建的第一个进程**，用于初始化进程管理、内存管理、加载Display、Camera、Binder等驱动相关工作
2. 启动kthreadd进程，这是Linux系统的内核进程，会创建内核工作线程kworkder、软中断线程ksoftirqd和thermal等内核守护进程。**kthreadd是所有内核进程的鼻祖**。

**Native层**

这里的native层主要包括有init进程孵化的用户空间的守护进程，bootanim开机动画和hal层等。**init是Linux系统的守护进程，是所有用户空间进程的鼻祖。init进程是Linux系统中用户空间的第一个进程，进程号固定为1。**

* init进程会孵化出ueventd、logd、healthd、installd、adbd、lm这里写代码片kd等用户守护进程；
* init进程还会启动ServiceManager（Binder服务管家）、bootanim（开机动画）等重要服务。
* init进程孵化出zygote进程，Zygote进程是Android系统的第一个Java进程（虚拟机进程），zygote进程是所有Java进程的父进程。



**Framework层**

framework层有native层和java层共同组成，协调系统平稳有序的工作。framework 层主要包括以下内容：

* Media Server进程，是由init进程fork而来，负责启动和管理整个C++ framework，包含AudioFlinger，Camera Service等服务。
* Zygote进程，**由Init进程通过解析init.rc文件生成，Zygote是Android系统的第一个Java进程，是所有Java进程的父进程。**
* System Server进程，由Zygote进程fork而来，**是Zygote进程孵化的第一个子进程**，负责启动和管理整个Java Framework，包括Ams、Pms等。

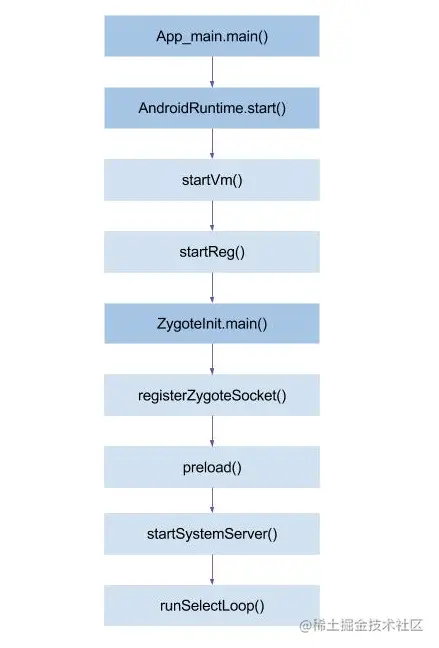
**App层**

Zygote进程孵化的第一个App进程是Launcher进程，也就是我们的桌面进程，也就是我们打开手机看到的用户界面。因为在前面的framework 生成了各种守护进程和管理进程，对于Launcher也就有对应的点击、长按、滑动、卸载等监听。Zygote进程也会创建Browser、Phone、Email等App进程。也就是说所有的App进程都是由Zygote进程fork生成的。而且上层的进程全部由下层的进程进行管理，包括但不限于界面的注册、跳转，消息的传递。

**Zygote启动过程**

app\_main.cpp的main函数中，主要做的事情就是参数解析，这个函数有两种启动模式

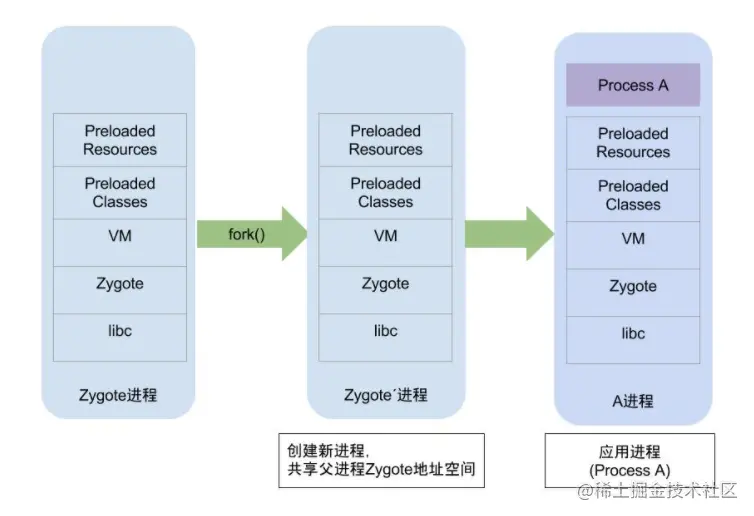
1. 一种是zygote模式，也就是初始化zygote进程，传递的参数有--start-system-server --socket-name=zygote，前者表示启动SystemServer，后者指定socket的名称。
2. 一种是application模式，也就是启动普通应用程序，传递的参数有class名字以及class带的参数



startVm：创建Java虚拟机（其实是Dalvik/Art虚拟机）方法的主要篇幅是关于虚拟机的参数设置。

ZygoteInit.main()：这里开始进入Java层

preload()：预加载位于/system/etc/preloaded-classes文件中的类，预加载资源，预加载OpenGL等等，zygote进程内加载了方法中的所有资源，当需要**fork新进程时，采用copy on write技术**，如下：



**Zygote采用高效的I/O多路复用机制，保证在没有客户端连接请求或数据处理时休眠，否则响应客户端的请求。**

**小结**

1. 解析init.zygote.rc中的参数，创建AppRuntime并调用AppRuntime.start()方法；
2. 调用AndroidRuntime的startVM()方法创建虚拟机，再调用startReg()注册JNI函数；
3. 通过JNI方式调用ZygoteInit.main()，第一次进入Java世界；
4. registerZygoteSocket()建立socket通道，zygote作为通信的服务端，用于响应客户端请求；
5. preload()预加载通用类、drawable和color资源、openGL以及共享库以及WebView，用于提高app启动效率；
6. zygote完毕大部分工作，接下来再通过startSystemServer()，fork得力帮手system\_server进程，也是上层framework的运行载体。
7. zygote功成身退，调用runSelectLoop()（死循环），随时待命，当接收到请求创建新进程请求时立即唤醒并执行相应工作。

**什么情况下Zygote进程会重启呢？**

* zygote：触发media、netd以及子进程(包括system\_server进程)重启；
* system\_server: 触发zygote重启;
* surfaceflinger：触发zygote重启;
* servicemanager: 触发zygote、healthd、media、surfaceflinger、drm重启

**fork函数**

**pid\_t** **fork**(**void**)

复制代码

1. 参数：不需要参数
2. 需要的头文件<sys/types.h>和<unnistd.h>
3. 返回值分两种情况：
   * 返回0表示成功创建子进程，并且接下来进入子进程执行流程
   * 返回PID(>0),成功创建子进程，并且继续执行父进程流程代码
   * 返回非正数(<0)，创建子进程失败，失败原因主要有： 进程数超过系统所能创建的上限，errno会被设置为EAGAIN系统内存不足，errno会被设置为ENOMEM

使用fork函数得到的子进程是父进程的一个复制品，它从父进程处继承了整个进程的地址空间。地址空间：包括进程上下文，进程堆栈、打开的文件描述符、信号控制设定、进程优先级、进程组号等。子进程中所独有的只有它的进程号，计时器等。因此，使用fork函数的代价是很大的。

**父子进程有哪些不同**

* fork之后的返回值不同，进程ID也不同
* 子进程未处理信号设置为空
* 子进程不继承父进程设置的文件锁
* 一般子进程会执行与父进程不完全一样的代码流程
* ··· ···

**孤儿进程，僵尸进程**

fork系统调用之后，父子进程将交替执行，执行顺序不定。如果父进程先退出，子进程还没退出那么子进程的父进程将变为init进程（托孤给了init进程）。如果子进程先退出，父进程还没退出，那么子进程必须得等到父进程捕获到了子进程的退出状态才能真正结束，否则这个手子进程就成为僵尸进程（僵尸进程：只保留一些退出信息供父进程查询）

**多进程线程的Fork调用**

在POSIX标准中，fork的行为是这样的：复制整个用户空间的数据（通常使用copy-on-write的策略，所以可以实现的速度很快）已经所有系统对象，然后复制当前线程到子进程。这里：所有父进程中别的线程，到了子进程中都是突然蒸发掉的。

假设这么一个环境，在 fork 之前，有一个子线程 lock 了某个锁，获得了对锁的所有权。fork 以 后，在子进程中，所有的额外线程都人间蒸发了。而锁却被正常复制了，在子进程看来，这个锁没 有主人，所以没有任何人可以对它解锁。当子进程想 lock 这个锁时，不再有任何手段可以解开 了。程序发生死锁 。

**面试题**

问题1：你了解Android系统系统启动流程吗？

当按电源触发开机，首先会从ROM中预定义的地方加载引导程序BootLoader到RAM中，并执行BootLoader程序启动Linux kernel，然后启动用户级别的第一个进程：init进程。init进程会解析init.rc脚本做一些初始化工作，包括挂载文件系统，创建功能做目录以及启动系统服务进程等，其中系统服务进程包括Zygote，Service manager，media等。在Zygote中会进一步去启动system\_server，然后在system\_server进程中会启动AMS,WMS,PMS等服务，等这些服务启动后，AMS中就会打开Launcher应用的Home Activity，最终看到手机的‘桌面’。

问题2：system\_server为什么要在Zygote中启动，而不是由init直接启动呢？

Zygote作为一个孵化器，可以提前加载一些资源，这样fork()时基于Copy-On-Write机制创建的其他进程就直接能使用这些资源，而不用重新加载。比如system\_server就可以直接使用Zygote中的JNI函数，共享库，常用的类，以及主题资源。

问题3：为什么要专门使用Zygote进程去孵化应用进程，而不是让system\_server去孵化呢？

首先system\_server相比Zygote多运行了AMS,WMS等服务，这些对一个应用程序来说是不需要的。另外进程的fork对多线程不友好，仅会将发起调用的线程拷贝到子进程，这可能会导致死锁，而ststem\_server中肯定是有很多线程的。

问题4：能具体说说是怎么导致死锁的吗？

在POSIX标准中，fork的行为是这样的：复制整个用户空间的数据（通常使用copy-on-write的策略，所以可以实现的速度很快）已经所有系统对象，然后复制当前线程到子进程。这里：所有父进程中别的线程，到了子进程中都是突然蒸发掉的。

假设这么一个环境，在 fork 之前，有一个子线程 lock 了某个锁，获得了对锁的所有权。fork 以 后，在子进程中，所有的额外线程都人间蒸发了。而锁却被正常复制了，在子进程看来，这个锁没 有主人，所以没有任何人可以对它解锁。当子进程想 lock 这个锁时，不再有任何手段可以解开 了。程序发生死锁 。

问题5：Zygote为什么不采用Binder机制进行IPC通信？

Binder机制中存在Binder线程池，是多线程的，如果Zygote采用Binder的话就存在上面说的fork()与多线程的问题了。其实严格来说，Binder机制不一定要多线程，所谓的Binder线程只不过是在循环读取Binder驱动的消息而已，只注册一个Binder线程也是可以工作的，比如server manager就是这样的。实际上Zygote尽管没有采取Binder机制，它也不是单线程的，但它在fork()前主动停止了其他线程，fork()后重启启动了。

参考文章：

[developer.android.com/guide/platf…](https://link.juejin.cn/?target=https%3A%2F%2Fdeveloper.android.com%2Fguide%2Fplatform%3Fhl%3Dzh-cn)

[gityuan.com/2016/02/13/…](https://link.juejin.cn/?target=http%3A%2F%2Fgityuan.com%2F2016%2F02%2F13%2Fandroid-zygote%2F)