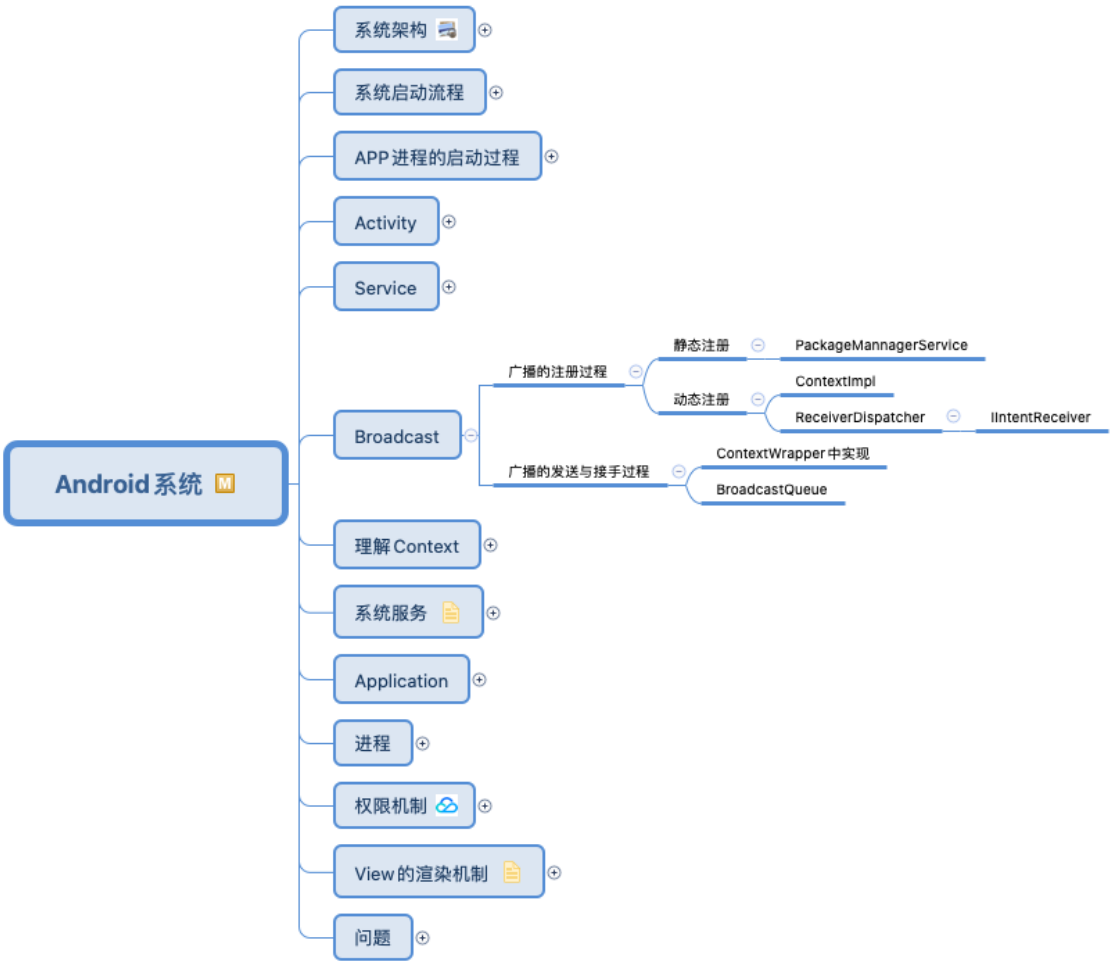
Android系统



1. 系统架构

1.1. 应用层

1.2. Framework

1.3. Native

1.3.1. C/C++ lib

1.3.2. Android Runtime

ART/5.0Dalvik

ART

使用 AOT（Ahead of time）编译

ART 内置了一个 Ahead-of-Time 编译器。在应用的安装期间，他就将 DEX 字节码翻译成机器码并存储在设备的存储器上。这个过程只在将应用安装到设备上时发生。由于不再需要 JIT 编译，代码的执行速度要快得多。

由于 ART 直接运行的是应用的机器码（native execution），它所占用的 CPU 资源要少于 使用 JIT 编译的 Dalvik。由于占用较少的 CPU 资源也就消耗更少的电池资源。

Dalvik

基于 JIT（Just in time）编译的引擎

使用 Dalvik JIT 编译器，每次应用在运行时，它实时的将一部分 Dalvik 字节码翻译成机器码。在程序的执行过程中，更多的代码被被编译并缓存。由于 JIT 只翻译一部分代码，它消耗的更少的内存，占用的更少的物理存储空间。

Core lib

1.4. 硬件抽象层HAL

1.5. Linux内核层

2. 系统启动流程

2.1. 引导程序BootLoader

2.2. Linux内核启动

2.3. init进程

2.3.1. 属性服务？

2.4. Zygote进程

2.4.1. 创建DVM或者ART

2.4.2. 为虚拟机注册JNI方法

2.4.3. 通过JNI调用ZygoteInit的main函数进入Zygote的java框架层

2.4.4. 创建一个Server端的Socket用于等待AMS请求创建APP进程

这个Socket是一个IPC，用于Zygote进程和SystemServer进程间的通信

2.4.5. 无限循环等待AMS请求

2.4.6. fork出SystemServer进程

fork出的进程可以获取到一个DVM或者ART的实例副本

启动这个进程的时候还创建了PathClassLoader

native方法启动Binder线程池

启动后才能进程IPC呀

反射调用SystemServer的main方法

main方法里面指点用了SystemServer的run方法

new SystemServer.run();

2.5. SystemServer进程

2.5.1. 创建SystemServerManager

2.5.2. 启动各种系统服务

PMS启动后会将系统中的应用程序安装完成

AMS会将Launcher启动起来

本质就是Activity的隐式调用

category设置了.HOME属性

HomeStack

Launcher会放在ActivityStackSupervisor中的一个专门的HomeStack中

3. APP进程的启动过程

3.1. AMS收到启动Activity的请求，此Activity的app进程尚未启动

3.2. AMS连接Zygote进程的Socket，向Zygote请求创建APP进程

3.3. Zygote进程收到请求后

3.3.1. 创建APP进程

3.3.2. 在app线程中创建Binder线程池

3.3.3. 反射调用ActivityThread的main方法

ActivityThread的main方法是个静态方法

3.4. ActivityThread的main方法

3.4.1. 创建Looper

3.4.2. 创建H handler

3.4.3. 开启消息循环

4. Activity

4.1. Activity启动过程

4.1.1. 点击Lancher上某个App的图标 ->

Lancher进程通过IAcivityManager请求AMS

4.1.2. AMS判断APP进程是否创建

未创建

请求Zygote进程fork新进程

已创建

4.1.3. AMS处理新Activity的堆栈逻辑

创建新的任务栈

启动模式的不同逻辑处理

。。。

ActivityStarter

ActivityStackSupervisor

ActivityStack

TaskRecord

ActivityRecord

4.1.4. 通过ApplicationThread通知ActivityThread

5.2. 跨APP启动

5.2.1. 显示启动

官方推荐

需要知道service全路径名和包名

5.2.2. 隐式启动

5.3. Service的绑定过程

5.3.1. ServiceDisPatcher

InnerConnection

5.3.2. ContextImpl中将ServiceConnection封装为IServiceConnection类型的对象

ServiceConnection不是个binder，

所以要在ContextImpl里边将它封装成Binder对象后传给AMS

5.3.3. IServiceConnection是客户端的Binder

5.3.4. 然后在调用AMS的代理对象的bindService方法时传给AMS

5.3.5. 服务端执行bindService流程将其Binder publish到AMS中

c.conn.connected(r.name, service)

IPC调用

间接回调了客户端ServiceConnection的回调方法

将服务端的Binder传会了客户端

5.3.6. 其他

在AMS中会调用ActiveServices的bindService方法？？？

6. Broadcast

6.1. 广播的注册过程

6.1.1. 静态注册

PackageMannagerService

6.1.2. 动态注册

ContextImpl

ReceiverDispatcher

IIntentReceiver

6.2. 广播的发送与接手过程

createActivityContext

attach

7.4. Service Context

7.4.1. createAppContext

7.4.2. service.attach(...)

8. 系统服务

ServiceManager

获取服务：

.get？

8.1. AMS

8.0以后IPC用的是AIDL了，IActivityManager代替了原来的ActivityManagerProxy

8.0以前是用的类似的AIDL方式

8.1.1. 相关类

ActivityManager

This class gives information about, and interacts

with, activities, services, and the containing

process.

in general, the methods in this class should

be used for testing and debugging purposes only.

getService方法获取本地代理

IActivityManager

AMP

IActivityManager单例

8.0前

AMP

AMN

AMS

8.0后

AIDL

AMS

IActivityManager

任务栈模型

ActivityStack

维护了很多ArrayList

TaskRecord

里面维护了一个活动历史记录的ArrayList

ActivityRecord

ActivityStackSupervisor

管理任务栈，其中包含多种ActivityStacck

8.1.2. 启动过程

SystemServer进程启动时启动

8.2. WMS

8.2.1. Window

应用程序窗口

子窗口

PopupWindow

系统窗口

Toast

输入法窗口

8.2.2. ViewRootImpl

最终是创建ViewRootImpl，将window设置其中

职责

View树的根

管理整个View树

触发View的测量、布局、绘制

performTraversals中会调relayoutWindow方法

relayoutWindow方法里调用IWindowSession的relayout方法来更新Window视图

管理Surface

负责与WMS进程间通信

IWindowSession

8.2.3. WMS添加窗口的实现

分配Surface

确定窗口的显示次序

将Surface交由SurfaceFlinger处理

SurfaceFlinger将Surface混合并绘制在屏幕上

8.2.4. 职责

窗口管理

核心

DisplayContent

用来描述一块屏幕

WindowToken

WindowState

窗口动画

WindowAnimator

输入系统的中转站

InputManagerService

Surface管理

窗口是一个抽象概念

需要有一块Surface来供其绘制

由WMS来给每个窗口分配Surface

SurfaceFlinger

8.2.5. 创建

SystemServer进行创建后会在启动其他服务的方法中启动WMS

调用WMS的main方法

在Android.display线程中创建wms实例

构造方法

持有IMS引用

通过DisplayManager获得Diaolay数组

遍历数组将diaplay封装成displaycontent

持有AMS引用

创建WindowAnimator

Watchdog？？？

8.2.6. 涉及三个线程

system_server

Android.display

android.ui

8.3. PMS

PackageManagerService

对APK进行安装、解析、删除、卸载等操作

9. Application

9.1. 生命周期

9.1.1. onCreate()

9.1.2. onLowMemory()

内存不够时触发

9.1.3. onTrimMemory(int level)

程序在内存清理的时候执行（回收内存）

HOME键退出应用程序、长按MENU键，打开Recent TASK都会执行

内存清理时触发

9.1.4. onConfigurationChanged(Configuration newConfig)

配置被改变时触发

9.1.5. onTerminate()

不保证一定被调用

程序终止的时候执行

10.进程

10.1. 进程的生命周期

系统会根据进程中正在运行的组件以及这些组件的状态，将每个进程分级。必要时，系统会首先回收优先级较低的进程。

10.2. 进程优先级

10.2.1. 前台进程

10.2.2. 可见进程

系统弹窗遮盖当前进程的activity

10.2.3. 服务进程

11.3. 权限变化趋势

11.3.1. Android M前

应用的权限请求是在安装时提示，确认后权限就会拥有。

11.3.2. 6.0运行时权限

在运行时做了进一步的检查，用户随时可拒绝权限。

权限组

11.4. 分类

11.4.1. 普通权限

普通权限不会涉及到用户隐私，如果应用在manifest文件中直接声明了普通权限，系统会自动授予权限给应用。比如：网络INTERNET、蓝牙BLUETOOTH、震动VIBRATE等权限。

11.4.2. 敏感权限

敏感权限则要获取到一些用户私密的信息。如果你的应用需要获取敏感权限，首先需要获取用户的授权。比如：相机CAMERA、联系人CONTACTS、存储设备STORAGE。

需要动态申请

12.View的渲染机制

Android应用程序把经过测量、布局、绘制后的surface缓存数据，通过SurfaceFlinger把数据渲染到屏幕上，通过Android的刷新机制来刷新数据。即应用层负责绘制，系统层负责渲染，通过进程间通信把应用层需要绘制的数据传递到系统层服务，系统层服务通过显示刷新机制把数据更新到屏幕。

12.1. Surface机制

应用层绘制到缓冲区，SurfaceFlinger把缓存区数据渲染到屏幕，两个进程之间使用Android的匿名共享内存SharedClient缓存需要显示的数据。

在Android的显示系统中，使用了Android的匿名共享内存：SharedClient。每一个应用和SurfaceFlinger之间都会创建一个SharedClient，每个SharedClient中，最多可以创建31个SharedBufferStack，每个Surface都对应一个SharedBufferStack，也就是一个window。这意味着一个Android应用程序最多可以包含31个窗口，同时每个SharedBufferStack中又包含两个(<4.1)或三个(>=4.1)缓冲区。

12.1.1. SurfaceFlinger服务

12.1.2. SharedClient

Android的匿名共享内存

SharedBufferStack

每个Surface都对应一个SharedBufferStack，也就是一个window

window

双缓冲

应用层将数据绘制到缓冲区

12.2. 要点

12.2.1. 一个Android应用程序最多可以包含31个窗口

每一个应用和SurfaceFlinger之间都会创建一个SharedClient，每个SharedClient中，最多可以创建31个SharedBufferStack，每个Surface都对应一个SharedBufferStack，也就是一个window。这意味着一个Android应用程序最多可以包含31个窗口，同时每个SharedBufferStack中又包含两个(<4.1)或三个(>=4.1)缓冲区。

12.3. 参考

12.3.1. 子主题 1

13.问题

13.1. app进程和AMS什么时候建立联系的？

13.2. 为什么中间要加个Instrumention

```
/**
 * Base class for implementing application instrumentation code.  When running
 * with instrumentation turned on, this class will be instantiated for you
 * before any of the application code, allowing you to monitor all of the
 * interaction the system has with the application.  An Instrumentation
 * implementation is described to the system through an AndroidManifest.xml's
 * &lt;instrumentation&gt; tag.
```

*/

allowing you to monitor all of the interaction the system has with the application

activity的生命周期都是委托这个类来回调的