**一14一 一14一 一12一 一11一** 一10.5一 一10.5一 '一10一"

**目 录**

[1. Android架构图 4](#_Toc25975)

[1.1. 资料一 4](#_Toc18999)

[1.1.1. 架构图 4](#_Toc355)

[1.1.2. 本质 4](#_Toc4743)

[1.1.3. android分为四个层 4](#_Toc18002)

[1.1.4. 最后在谈谈Android所采用的语言 5](#_Toc5057)

[1.2. 资料二 5](#_Toc21935)

[1.2.1. Android系统架构 5](#_Toc8996)

[1.2.1.1. 应用层 6](#_Toc2878)

[1.2.1.2. 应用框架层（Java Framework) 6](#_Toc755)

[1.2.1.3. 系统运行库层（Native) 7](#_Toc26489)

[1.2.1.3.1. C/C++程序库 7](#_Toc995)

[1.2.1.3.2. Android运行时库 7](#_Toc2529)

[1.2.1.4. 硬件抽象层（HAL) 7](#_Toc31400)

[1.2.1.5. Linux内核层 7](#_Toc8896)

[2. Android源码结构 7](#_Toc29471)

[2.1. 整体结构 7](#_Toc16442)

[2.2. 应用层部分 8](#_Toc5555)

[2.3. 应用框架层部分 8](#_Toc14490)

[2.4. C/C++程序库部分 9](#_Toc6372)

[3. API与版本对应关系及分布比例 9](#_Toc17461)

[4. Android从按下开机键到启动发生了什么 10](#_Toc10274)

[4.1. 基于Linux的pc启动过程 10](#_Toc11115)

[4.2. Android手机的启动过程 11](#_Toc12884)

[5. Activity生命周期 12](#_Toc20861)

[5.1. 生命周期图 13](#_Toc26052)

[5.2. 把Activity作为参数传给一个静态方法,会影响这个Activity的正常销毁吗 13](#_Toc27482)

[5.3. onNewIntent调用时机 13](#_Toc3145)

[5.4. 横竖屏切换的时候的生命周期 13](#_Toc27026)

[6. Activity之间传输数据 13](#_Toc20968)

[6.1. Activity最佳启动方法 14](#_Toc5894)

[6.2. 通过intent 14](#_Toc4412)

[6.3. 通过静态static 14](#_Toc14875)

[6.4. 存Bitmap和map等对象： 15](#_Toc23656)

[6.5. 取Bitmap和map等对象： 15](#_Toc13428)

[7. Handler的基础用法 15](#_Toc28262)

[7.1. 创建Message对象 15](#_Toc15741)

[7.2. Message的arg1，arg2和what 16](#_Toc20454)

[7.3. 简单的使用方式 17](#_Toc30285)

[8. Handler机制原理 17](#_Toc5156)

[8.1. 概述 17](#_Toc18860)

[8.2. 子线程初始化handler，相关注意 17](#_Toc30038)

[8.3. Handler创建消息 17](#_Toc24902)

[8.4. Handler发送消息 18](#_Toc5536)

[8.5. Handler处理消息 18](#_Toc322)

[8.6. 制作欢迎界面使用sendEmptyMessageDelayed 19](#_Toc4284)

[8.7. 使用handler.post()方法，不会执行handleMessage()方法 20](#_Toc26079)

[8.8. 用Post在子线程里操作主线程 20](#_Toc11182)

[8.8.1. Post基本用法 20](#_Toc23246)

[8.8.2. 延迟1s执行这个Runnable 20](#_Toc6967)

[8.8.3. Activity.runOnUiThread 20](#_Toc16976)

[8.8.4. View.post 20](#_Toc24308)

[8.9. 使用Post的分析Handler机制 21](#_Toc12910)

[8.9.1. Handler源码 21](#_Toc24127)

[8.9.2. 再来看post() 21](#_Toc16576)

[8.9.3. Android消息机制 22](#_Toc7502)

[8.10. 最后谈谈handler的内存泄露问题 24](#_Toc6560)

[9. Activity启动之Handle 25](#_Toc21547)

[9.1. 从Tread的创建流程开始 25](#_Toc9657)

[9.1.1. 线程创建的起始点init() 25](#_Toc15985)

[9.1.2. 第二个init2() 26](#_Toc22869)

[9.2. 启动线程，开车啦！ 26](#_Toc29174)

[9.3. 几个常见的线程手段(操作) 27](#_Toc25305)

[9.3.1. Thread.sleep()那不可告人的秘密 27](#_Toc15310)

[9.3.2. Thread.yield()究竟隐藏了什么？ 29](#_Toc23111)

[9.3.3. 无处不在的wait()究竟是什么？ 29](#_Toc15938)

[9.4. Looper、Handler、MessageQueue之间的爱恨情仇 29](#_Toc25034)

[9.4.1. 从Looper.prepare()开始 29](#_Toc8121)

[9.5. 总结 30](#_Toc26509)

[10. Activity启动之流程 30](#_Toc22941)

[10.1. 一张图明白Activity的启动流程 30](#_Toc31604)

[10.1.1. 一切从main()方法开始 31](#_Toc28618)

[10.1.2. 创建Application的消息是如何发送的呢？ 31](#_Toc32620)

[10.1.2.1. IActivityManager mgr是个啥？ 31](#_Toc23556)

[10.1.2.2. ApplicationThread mAppThread又是个啥？ 33](#_Toc28528)

[10.1.2.3. ActivityManagerService调度发送初始化消息 33](#_Toc5763)

[10.1.3. 收到初始化消息之后的世界 34](#_Toc9841)

[10.1.3.1. Instrumentation仪表，什么鬼？ 35](#_Toc28608)

[10.1.3.2. LoadedApk就是data.info哦！ 35](#_Toc4148)

[10.1.3.3. 现在把目光移回Instrumentation 35](#_Toc28860)

[10.1.4. LaunchActivity 36](#_Toc12521)

[11. Activity启动之View加载和app卡顿分析 37](#_Toc5310)

[11.1. 从setContentView()说起 37](#_Toc24348)

[11.1.1. PhoneWindow 38](#_Toc13727)

[11.1.2. DecorView 38](#_Toc2994)

[11.1.3. WindowManager 38](#_Toc19795)

[11.1.4. ViewRootImpl 40](#_Toc2769)

[11.2. App总是卡顿到底是什么原因？ 40](#_Toc8552)

[11.2.1. CPU、GPU是搞什么鬼的？ 40](#_Toc11746)

[11.2.2. RefreshRate刷新率和FrameRate帧率 41](#_Toc8041)

[11.2.3. Vsync(垂直同步)是什么？ 42](#_Toc12002)

[11.2.3.1. HWComposer与Vsync不得不说的事 42](#_Toc6522)

[11.2.3.2. Vsync offset机制 42](#_Toc32540)

[11.2.4. 回到ViewRootImpl 43](#_Toc19619)

[12. Activity跳转防重 43](#_Toc23738)

[13. 序列化Serializable、Parcelabel 46](#_Toc9891)

[13.1. 为什么要序列化？ 46](#_Toc28110)

[13.2. 实现序列化的方法 46](#_Toc12107)

[13.3. Serializable实现与Parcelabel实现的区别 47](#_Toc21703)

[14. Activity的四种启动模式和一些标签 48](#_Toc22953)

[14.1. 四种启动模式 48](#_Toc4463)

[14.2. Activity其他的标记的一些解释 48](#_Toc25484)

[15. getIntent()、getStringExtra---Activity在oncrat（）中 49](#_Toc23260)

[16. Intent 49](#_Toc24102)

[16.1. Intent的Component属性 50](#_Toc16901)

[16.2. Intent的Action属性 50](#_Toc4321)

[16.3. Intent的Category属性 50](#_Toc10708)

[16.4. Intent的Data和Type属性 50](#_Toc12182)

[16.5. Intent的Extra属性 51](#_Toc18667)

[16.6. IntentFilter与隐式Intent 51](#_Toc26573)

[16.6.1. 测试action属性 51](#_Toc19065)

[16.6.2. 测试category属性 51](#_Toc9391)

[16.6.3. 测试data属性 51](#_Toc3793)

[16.7. Intent的data属性，以及intent-filter 51](#_Toc10189)

[16.8. Intent跳转Activity 52](#_Toc20900)

[16.9. action和type，data可以实现不同的功能 52](#_Toc17649)

[16.10. Intent FLAG介绍: 53](#_Toc27396)

[17. Application全局化 54](#_Toc16779)

[17.1. 用处一：Activity之间传递数据 54](#_Toc18853)

[18. Context 56](#_Toc7293)

[18.1. 什么是Context 56](#_Toc30704)

[18.2. 看代码 56](#_Toc26964)

[18.3. getApplicationContext和getApplication1 56](#_Toc19172)

[18.4. Context的生命周期 57](#_Toc554)

[18.5. Context相关类的继承关系 57](#_Toc4323)

[18.6. 什么时候创建Context实例 58](#_Toc11724)

[19. SharedPreferences保存本地配置 60](#_Toc1993)

[20. startActivityForResult、request、resultCode 61](#_Toc982)

[20.1. 调用startActivityForResult后，onActivityResult立刻响应 61](#_Toc23827)

[20.2. 请求码的作用 61](#_Toc26336)

[20.3. 结果码的作用 62](#_Toc20442)

[21. onSaveInstanceState 62](#_Toc27898)

[21.1. 保存Activity的状态 62](#_Toc3043)

[21.2. 恢复Activity的状态 62](#_Toc4349)

[21.3. 保存数据（意外被清理的情况下）onRestoreInstanceState和onSaveInstanceState学习 62](#_Toc20003)

[22. 静态导入import static和import的区别 63](#_Toc7041)

[23. Android工程目录结构 63](#_Toc9826)

[23.1. src 63](#_Toc26751)

[23.2. gen 63](#_Toc1755)

[23.3. Android n.n.n 64](#_Toc5851)

[23.4. assets（资产财产） 64](#_Toc6676)

[23.5. res文件夹（资源） 64](#_Toc22893)

[23.6. AndroidManifest.xml 64](#_Toc22596)

[23.7. libs 64](#_Toc7501)

[23.8. bin 64](#_Toc24822)

[24. res/raw和assets的使用 64](#_Toc5987)

[24.1. res/raw和assets的相同点： 64](#_Toc723)

[24.2. res/raw和assets的不同点： 64](#_Toc7051)

[24.3. 读取文件资源： 64](#_Toc27324)

[25. getFilesDir、getCacheDir、openFileInput 65](#_Toc29135)

[25.1. 检查是否挂载SD卡 65](#_Toc31889)

[25.2. 通过API获取路径 65](#_Toc17032)

[25.3. 文件的I/O是存放在/data/data/<package name>/file/filename目录下。 65](#_Toc11101)

[25.4. 获取外置内存卡的方法 65](#_Toc14432)

[25.4.1. 方法一使用System类（Api<=23） 65](#_Toc12173)

[25.4.2. 方法二通过反射 66](#_Toc29393)

[26. 播放系统提示音和默认震动 66](#_Toc17340)

[27. Notification通知栏 66](#_Toc9411)

[27.1. 代码 66](#_Toc23177)

[27.2. 自定义布局的写法 67](#_Toc29209)

[27.3. 通常驻状态下想要取消 68](#_Toc30322)

[28. PendingIntent 68](#_Toc26424)

[29. 音频播放（MediaPlayer和SoundPool） 68](#_Toc16040)

[29.1. 使用MediaPlayer播放音频 69](#_Toc4481)

[29.1.1. 从资源文件中播放 69](#_Toc18991)

[29.1.2. 从文件系统（内存卡）播放 69](#_Toc5735)

[29.1.3. 从网络播放 69](#_Toc28966)

[29.1.4. 结束播放 69](#_Toc31778)

[29.2. SoundPool播放raw文件夹里的提示音 69](#_Toc15668)

[29.3. SoundPool工具类 70](#_Toc30944)

[30. 视频播放VideoView 71](#_Toc6758)

[30.1. 要实现video在容器里全屏，需重新videoview的onMeasure方法 72](#_Toc1194)

[30.2. 确保VideoView全屏且尽量不拉伸 72](#_Toc13319)

[31. MediaPlayer 72](#_Toc2910)

[31.1. 概述 73](#_Toc1342)

[31.2. 奇怪的例子 73](#_Toc18272)

[31.3. MediaPlayer +ViewPager+surfaceView 77](#_Toc9065)

[32. SurfaceView 78](#_Toc21234)

[32.1. 概述 79](#_Toc14485)

[32.2. 和MediaPlayer组合使用时 79](#_Toc11618)

[33. 录音机工具类 81](#_Toc5769)

[34. Json解析 82](#_Toc6750)

[34.1. 传统解析 82](#_Toc20273)

[34.2. 使用GSON库解析单纯jsonobject 83](#_Toc13617)

[34.3. 使用GSON库解析单纯jsonarray 83](#_Toc7871)

[34.4. 解析有标签的object（用内部类） 83](#_Toc458)

[34.5. 解析有标签的Array 84](#_Toc9328)

[34.6. 类转json格式字符串 85](#_Toc12251)

[34.7. 大类解析 86](#_Toc1769)

[34.8. 解析某些特定符号会转成Unicode 87](#_Toc28647)

[35. XML解析 87](#_Toc1322)

[35.1. 解析XML的几种方式的原理与特点：DOM、SAX、PULL 89](#_Toc12250)

[36. 压缩与解压 89](#_Toc7889)

[36.1. Gzip压缩与解压 89](#_Toc14730)

[36.1.1. Gzip压缩 89](#_Toc21616)

[36.1.2. Gzip解压 90](#_Toc7357)

[36.2. Zip压缩与解压工具类ZipUtil 90](#_Toc10829)

# Android架构图

## 资料一

### 架构图



### 本质

Android其本质就是在标准的Linux系统上增加了Java虚拟机Dalvik，并在Dalvik虚拟机上搭建了一个JAVA的application framework，所有的应用程序都是基于JAVA的application framework之上。  
 Android主要应用于ARM平台，但不仅限于ARM，通过编译控制，在X86、MAC等体系结构的机器上同样可以运行。

### android分为四个层

从高层到低层分别是应用程序层、应用程序框架层、系统运行库层和linux核心层。蓝色的代表java程序，黄色的代表为运行JAVA程序而实现的虚拟机，绿色部分为C/C++语言编写的程序库，红色的代码内核(linux内核+driver)。在Application Framework之下，由C/C++的程序库组成，通过JNI完成从JAVA到C的调用。

**1) 应用程序**  
 所有的应用程序都是使用JAVA语言编写的，每一个应用程序由一个或者多个活动组成，活动必须以Activity类为超类，活动类似于操作系统上的进程，但是活动比操作系统的进程要更为灵活，与进程类似的是，活动在多种状态之间进行切换。  
利用JAVA的跨平台性质，基于Android框架开发的应用程序可以不用编译运行于任何一台安装有android系统的平台，这点正是Android的精髓所在。

**2) 应用程序框架**  
 应用程序的架构设计简化了组件的重用；任何一个应用程序都可以发布它的功能块并且任何其它的应用程序都可以使用其所发布的功能块（不过得遵循框架的安全性限制）。帮助程序员快速的开发程序，并且该应用程序重用机制也使用户可以方便的替换程序组件。  
隐藏在每个应用后面的是一系列的服务和系统, 其中包括；  
 a.丰富而又可扩展的视图（Views），可以用来构建应用程序， 它包括列表（lists），网格（grids），文本框（text boxes），按钮（buttons），甚至可嵌入的web浏览器。  
 b.内容提供器（Content Providers）使得应用程序可以访问另一个应用程序的数据（如联系人数据库）， 或者共享它们自己的数据。  
 c.资源管理器（Resource Manager）提供非代码资源的访问，如本地字符串，图形，和布局文件（layout files）。  
 d.通知管理器（Notification Manager）使得应用程序可以在状态栏中显示自定义的提示信息。  
 e.活动管理器（Activity Manager）用来管理应用程序生命周期并提供常用的导航回退功能。

**3) 系统运行库**  
 a)程序库  
    Android包含一些C/C++库，这些库能被Android系统中不同的组件使用。它们通过Android应用程序框架为开发者提供服务。主要包括基本的C库、以及多媒体库以支持各种多媒体格式、位图和矢量字体、2D和3D图形引擎、浏览器、数据库支持。  
 1.Bionic系统C库。  
 2.媒体库，基于PacketVideo OpenCORE。  
 3.Surface Manager 顾名思义，用于管理Surface。  
 4.Webkit,LibWebCore 浏览器，基于Webkit引擎。  
 5.SGL 底层的2D图形引擎  
 6.3D libraries 基于OpenGL ES 1.0 APIs实现  
 7.FreeType 位图（bitmap）和矢量（vector）字体显示。  
 8.SQLite 一个对于所有应用程序可用，功能强劲的轻型关系型数据库引擎。  
 另外这里还有一个硬件抽象层。其实Android并非所有的设备驱动都放在linux内核里面，有一部分实现在用户空间，  
这么做的主要原因是可以避开Linux所遵循的GPL协议，一般情况下如果要将Android移植到其他硬件去运行，  
只需要实现这部分代码即可。包括：显示器驱动，声音，相机，GPS，GSM等等。  
b)Android 运行库  
Android 包括了一个核心库，该核心库提供了JAVA编程语言核心库的大多数功能。  
每一个 Android应用程序都在它自己的进程中运行，都拥有一个独立的Dalvik虚拟机实例。  
Dalvik被设计成一个设备可以同时高效地运行多个虚拟系统。      
Dalvik虚拟机执行（.dex）的Dalvik可执行文件，该格式文件针对小内存使用做了优化。  
同时虚拟机是基于寄存器的，所有的类都经由JAVA编译器编译，然后通过SDK中的 "dx" 工具转化成.dex格式由虚拟机执行。  
Dalvik虚拟机依赖于linux内核的一些功能，比如线程机制和底层内存管理机制。  
**4) Linux 内核**  
 Android 的核心系统服务依赖于 Linux 2.6 内核 ，如安全性，内存管理，进程管理， 网络协议栈和驱动模型。   
Linux 内核也同时作为硬件和软件栈之间的抽象层。其外还对其做了部分修改，主要涉及两部分修改：  
a)Binder     
  (IPC)：提供有效的进程间通信，虽然linux内核本身已经提供了这些功能，但Android系统很多服务都需要用到该功能，为了某种原因其实现了自己的一套。  
b)电源管理

为手持设备节省能耗。

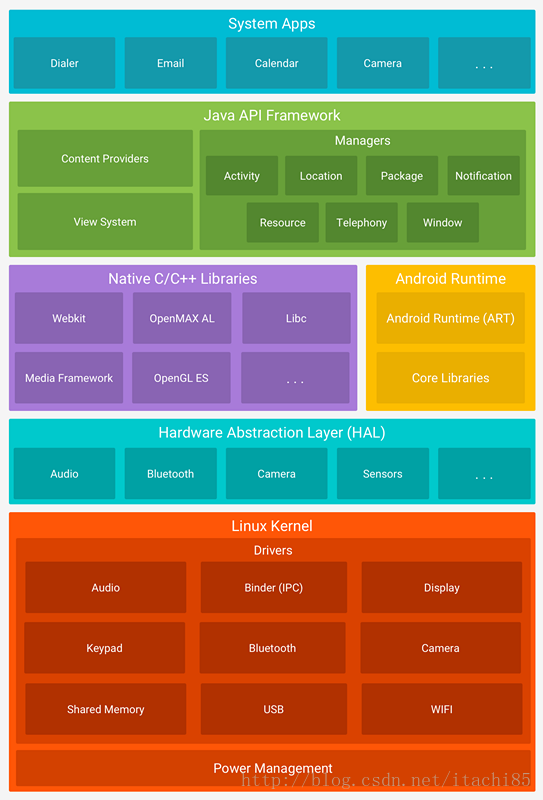
### 最后在谈谈Android所采用的语言

其应用开发采用java语言，我们所说的java一般包含三个部分：  
a) java语言：即其语法，其写代码的程式  
b) java虚拟机：为了实现一次编译到处可以运行的原则，java在编译连接以后并没有产生目标机器语言，而是采用了Java bytecode这种Java共用指令，这时就需要一个虚拟机来执行改指令。  
c) 库：跟我们常用的C语言一样提供一些常用的库  
 后两者结合就是Java Runtime Environment。

## 资料二

### Android系统架构

Android系统架构分为五层，从上到下依次是应用层、应用框架层、系统运行库层、硬件抽象层和Linux内核层



#### 应用层

系统内置的应用程序以及非系统级的应用程序都是属于应用层。负责与用户进行直接交互，通常都是用Java进行开发的。

#### 应用框架层（Java Framework)

应用框架层为开发人员提供了可以开发应用程序所需要的API，我们平常开发应用程序都是调用的这一层所提供的API，当然也包括系统的应用。这一层的是由Java代码编写的，可以称为Java Framework。下面来看这一层所提供的主要的组件。



#### 系统运行库层（Native)

系统运行库层分为两部分，分别是C/C++程序库和Android运行时库。下面分别来介绍它们。

##### C/C++程序库

C/C++程序库能被Android系统中的不同组件所使用，并通过应用程序框架为开发者提供服务，主要的C/C++程序库：



##### Android运行时库

运行时库又分为核心库和ART(5.0系统之后，Dalvik虚拟机被ART取代)。核心库提供了Java语言核心库的大多数功能，这样开发者可以使用Java语言来编写Android应用。相较于JVM，Dalvik虚拟机是专门为移动设备定制的，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。而替代Dalvik虚拟机的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，使其成为真正的本地应用

#### 硬件抽象层（HAL)

硬件抽象层是位于操作系统内核与硬件电路之间的接口层，其目的在于将硬件抽象化，为了保护硬件厂商的知识产权，它隐藏了特定平台的硬件接口细节，为操作系统提供虚拟硬件平台，使其具有硬件无关性，可在多种平台上进行移植。 从软硬件测试的角度来看，软硬件的测试工作都可分别基于硬件抽象层来完成，使得软硬件测试工作的并行进行成为可能。通俗来讲，就是将控制硬件的动作放在硬件抽象层中。

#### Linux内核层

Android 的核心系统服务基于Linux 内核，在此基础上添加了部分Android专用的驱动。系统的安全性、内存管理、进程管理、网络协议栈和驱动模型等都依赖于该内核。

# Android源码结构

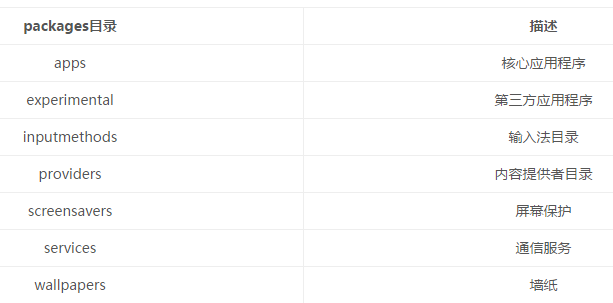
## 整体结构

各个版本的源码目录基本是类似，如果是编译后的源码目录会多增加一个out文件夹，用来存储编译产生的文件。Android7.0的根目录结构说明如下图所示。



## 应用层部分

应用层部分在根目录的packages中，它位于整个Android系统的最上层，开发者开发的应用程序以及系统内置的应用程序都是在应用层。它的目录结构如图所示



从目录结构可以发现，packages目录存放着系统核心应用程序、第三方的应用程序和输入法等等，这些应用都是运行在系统应用层的，因此packages目录对应着系统的应用层。

## 应用框架层部分

应用框架层是系统的核心部分，一方面向上提供接口给应用层调用，另一方面向下与C/C++程序库以及硬件抽象层等进行衔接。 应用框架层的主要实现代码在/frameworks/base和/frameworks/av目录下，其中/frameworks/base目录结构如下



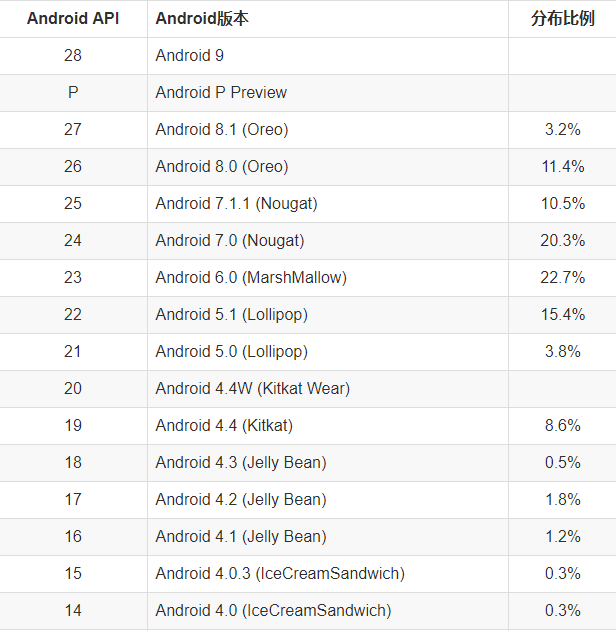
## C/C++程序库部分

系统运行库层（Native)中的 C/C++程序库的类型繁多，功能强大，C/C++程序库并不完全在一个目录中，这里给出几个常用且比较重要的C/C++程序库所在的目录位置。



讲完 C/C++程序库部分，剩下的部分我们在第一幅图已经给出：Android运行时库的代码放在art/目录中。硬件抽象层的代码在hardware/目录中，这一部分是手机厂商改动最大的一部分，根据手机终端所采用的硬件平台会有不同的实现。

# API与版本对应关系及分布比例



# Android从按下开机键到启动发生了什么

## 基于Linux的pc启动过程

我们都知道，所有的程序软件包括操作系统都是运行在内存中的，然而我们的操作系统一般是存放在硬盘上的，当我们按下开机键的时候，此时内存中什么程序也没有，因此需要借助某种方式，将操作系统加载到内存中，而完成这项任务的就是BIOS。装过系统的人一定知道BIOS这个东西，那么它究竟是什么呢？

BIOS：Basic Input/Output System（基本输入输出系统），在IBM PC兼容系统上，是一种业界标准的固件接口（来自维基百科）。有点难以理解，其实BIOS是我们电脑启动时加载的第一个程序，这个程序不是由Java语言编写也不是由C语言编写，一般是汇编程序。BIOS程序固化在主板上的一块芯片上，是连接计算机硬件与操作系统的桥梁，它保存着计算机最重要的基本输入输出的程序、开机后自检程序和系统自启动程序。

那么问题来了，BIOS程序又是怎么启动的？BIOS的启动，是由硬件完成的，Intel 80x86系列的cpu的硬件都设计为加电（即开机瞬间）就进入16位实模式状态运行，此时将cpu的硬件逻辑设计为强行将CS的值设置为0xFFFF,IP的值设置为0x0000，这样CS:IP就指向了0xFFFF0这个位置，而这个位置就是BIOS程序的入口地址。

因此这是一个硬件厂商之间的约定，所有的BIOS程序入口地址均为0xFFFF0，这样在开机的时候，就找到这个地址，如果该地址并没有代码段，那么计算机将会死机，如果这个地址处有代码段，将会执行这个代码段，并由此执行下去，即BIOS程序开始启动。

**补充：**

* CS:代码段寄存器，存在于CPU中，指向CPU当前执行代码在内存中所在的区域。
* IP：指令寄存器，存在于CPU中，记录将要执行的指令在代码段内的偏移地址，与CS组合即为将要执行的指令的内存地址。

当BIOS程序启动时，就会检测硬件设备，比如我们的显卡、内存等信息。BIOS会在内存中建立中断向量表和中断服务程序。中断向量表中有256个中断向量，每个中断向量占4个字节，每个中断向量指向一个中断服务程序，这些中断服务程序完成了将操作系统由硬盘加载到内存中的任务。

基于linux的操作系统而言，计算机将分三批逐次加载操作系统的代码，第一批由BIOS中断int 0x19将第一扇区bootsect的内容加载到内存；第二批和第三批在bootsect的指挥下，分别加载后面扇区的内容到内存中。经过执行一系列的BIOS代码后，计算机完成了自检等操作，计算机硬件体系会与BIOS联合操作，让cpu接收到一个int 0x19中断，cpu接收到这个中断后，会立即在中断向量表中找到int 0x19中断向量，此时会找到对应的中断服务程序，并由该中断服务程序将硬盘中第一个扇区的引导程序加在到内存中的指定位置。随后，在引导程序的作用下，陆续将操作系统的其他程序载入内存，完成实模式到保护模式的转变，为执行操作系统的入口函数main做准备，后面就是操作系统的初始化工作了，最后完成计算机的启动。

## Android手机的启动过程

Android系统虽然也是基于linux系统的，但是由于Android属于嵌入式设备，并没有像pc那样的BIOS程序。取而代之的是Bootloader——系统启动加载器。它类似于BIOS，在系统加载前，用以初始化硬件设备，建立内存空间的映像图，为最终调用系统内核准备好环境。

在Android里没有硬盘，而是ROM，它类似于硬盘存放操作系统，用户程序等。ROM跟硬盘一样也会划分为不同的区域，用于放置不同的程序

**在Android中主要划分为一下几个分区：**

* /boot：存放引导程序，包括内核和内存操作程序
* /system：相当于电脑c盘，存放Android系统及系统应用
* /recovery：恢复分区，可以进入该分区进行系统恢复
* /data：用户数据区，包含了用户的数据：联系人、短信、设置、用户安装的程序
* /cache：安卓系统缓存区，保存系统最常访问的数据和应用程序
* /misc：包含一些杂项内容，如系统设置和系统功能启用禁用设置
* /sdcard：用户自己的存储区，可以存放照片，音乐，视频等文件

那么Bootloader是如何被加载的呢？我们可以想到，应该跟pc一样，当开机加电的时候，cpu会从cpu制造厂商预设的地址上取指令，这个地址是各厂商约定俗称的，类似于上面80x86架构里的0xFFFF0地址，因此Android手机会将固态存储设备ROM预先映射到该地址上，当开机加电的时候，cpu就会从该地址执行/boot分区下的Bootloader程序，载入linux内核到RAM中。

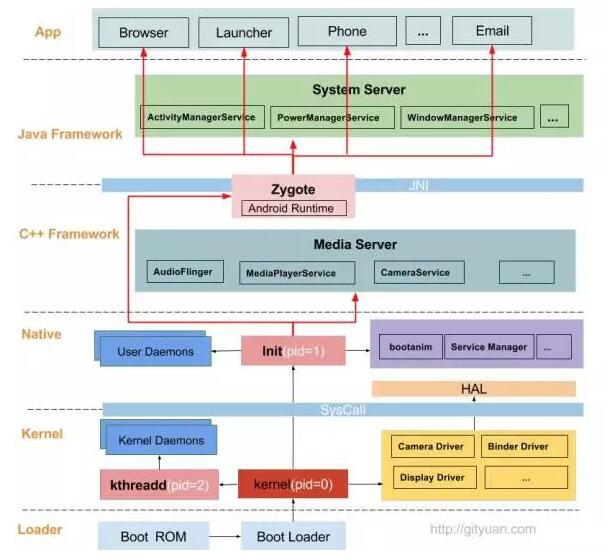
当linux内核启动后会初始化各种软硬件环境，加载驱动程序，挂载根文件系统，并开始执行根文件系统的init程序，init程序是Android启动过程中最重要的核心程序。init进程是Android系统中用户进程的鼻祖进程。init进程会启动各种系统本地服务，如：Media Server、Service Manager、bootanim（开机动画）等。init进程会在解析init.rc文件后fork出Zygote，而Zygote是所有Java进程的父进程，我们的App都是由Zygote fork出来的。

**Zygote进程主要包含：**

* 加载ZygoteInit类，注册Zygote Socket服务端套接字；
* 加载虚拟机；
* 预加载Android核心类
* 预加载系统资源

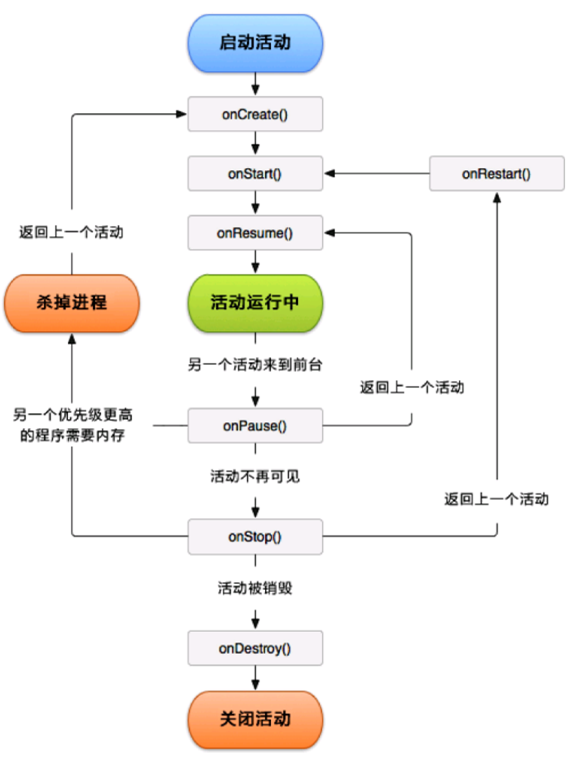
随后Zygote进程会fork出System Server进程，System Server进程负责启动和管理整个framework，包括Activity Manager，PowerManager等服务。当System Server将系统服务启动就绪后，就会通知ActivityManager启动首个Android程序Home即我们看到的桌面程序。

至此，从Android手机开机到看到桌面程序所有过程分析完了。



# Activity生命周期

## 生命周期图



## 把Activity作为参数传给一个静态方法,会影响这个Activity的正常销毁吗

* 内存泄露与方法是否是静态与否无关，与内部的方法体实现有关系。
* 内存泄露可以简单理解成：生命周期长的对象不正确持有了持有了生命周期短的对象，导致生命周期短的对象无法回收。
* 比如Activity实例被Application对象持有，Activity实例被静态变量持有

## onNewIntent调用时机

首先，在默认情况下，当通过Intent启到一个Activity的时候，就算已经存在一个相同的正在运行的Activity,系统都会创建一个新的Activity实例并显示出来。为了不让Activity实例化多次，我们需要通过在AndroidManifest.xml配置activity的加载方式（launchMode）以实现单任务模式，如下所示：

<activity android:label="@string/app\_name" android:launchmode="singleTask"android:name="Activity1"/>

launchMode为singleTask的时候，通过Intent启到一个Activity,如果系统已经存在一个实例，系统就会将请求发送到这个实例上，但这个时候，系统就不会再调用通常情况下我们处理请求数据的onCreate方法，而是调用onNewIntent方法

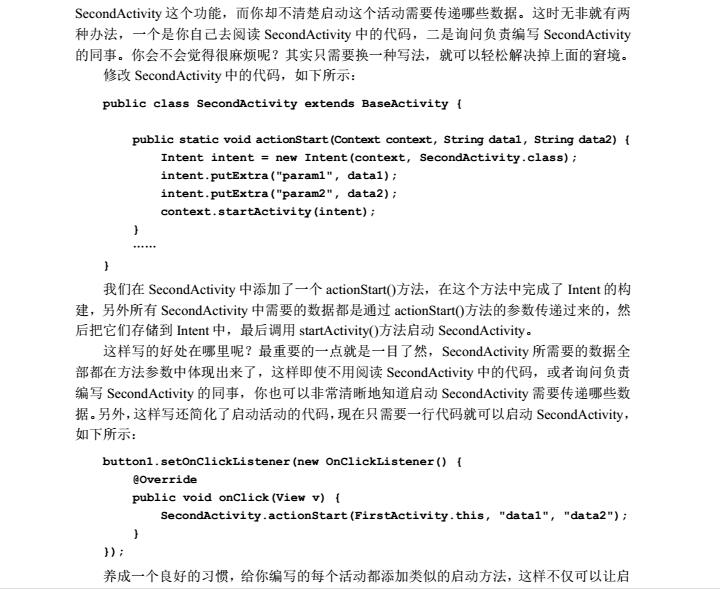
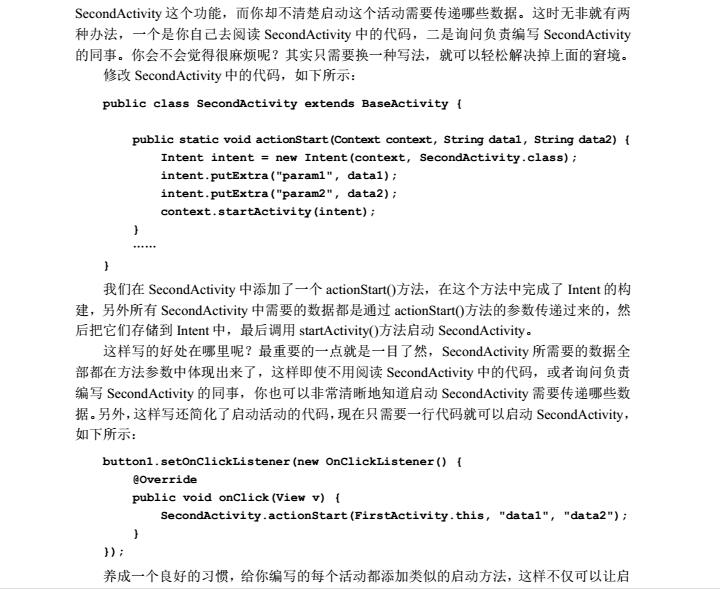
不要忘记，系统可能会随时杀掉后台运行的Activity，如果这一切发生，那么系统就会调用onCreate方法，而不调用onNewIntent方法，一个好的解决方法就是在onCreate和onNewIntent方法中调用同一个处理数据的方法

## 横竖屏切换的时候的生命周期

* 不设置Activity的android:configChanges时，切屏会重新调用各个生命周期，切横屏时会执行一次，切竖屏时会执行两次
* 设置Activity的android:configChanges="orientation"时，切屏还是会重新调用各个生命周期，切横、竖屏时只会执行一次
* 设置Activity的android:configChanges="orientation|keyboardHidden"时，切屏不会重新调用各个生命周期，只会执行onConfigurationChanged方法

# Activity之间传输数据

## Activity最佳启动方法



## 通过intent

public class Obj implements Serializable {// 实现虚拟化接口，传值的时候才能把对象传过去

private String name;

**//传**

Intent intent = new Intent();

intent.setClass(MainActivity.this, OtherActivity.class);

intent.putExtra("guang", "我是光哥");

Obj obj = new Obj();

obj.setName("光哥");

intent.putExtra("xixi", obj);

startActivity(intent);

**//收1**

Bundle bundle = getIntent().getExtras();

Bundle bundle = intent.getExtras();

String gaga = bundle.getString("gaga");

Obj xixi = (Obj) bundle.get("guang");

String xixiString = xixi.getName();

**//收2**

Intent intent1 = this.getIntent();  String flag = intent1.getStringExtra("flag");

## 通过静态static

public class Obj {

private String name;

public class OtherActivity extends Activity {

static Obj obj = new Obj();

Log.d("OtherActivity", "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"+obj.getName());

protected void onDestroy() {

super.onDestroy();

obj=null;//用完之后没有用了，将他回收，节省

Obj obj=new Obj();

obj.setName("光哥");

OtherActivity.obj=obj;//使用静态变量来传

**注意：bundle不能传大于0.25M的对象**

## 存Bitmap和map等对象：

**Bundle** bundle = **new** Bundle();

**Bitmap** bitmap = **BitmapFactory**.decodeResource(getResources(), **R**.**drawable**.**ic\_launcher**);

bundle.putParcelable("bitmap", bitmap);

**HashMap**<**String**,**String**> data = **new** HashMap<**String**,**String**>();

data.put("gaga", "gaga");

**SerializableMap** tmpmap=**new** SerializableMap();

tmpmap.setMap(data);

bundle.putSerializable("map", tmpmap);

intent.putExtras(bundle);

## 取Bitmap和map等对象：

**Bundle** bundle = getIntent().getExtras();

**SerializableMap** map = (**SerializableMap**) bundle.get("map");

**HashMap**<**String**, **String**> hashMap = map.getMap();

**String** gaga = hashMap.get("gaga");

**Bitmap** bitmap = (**Bitmap**) bundle.get("bitmap");

**实现接口：**

**public** **class** **SerializableMap** **implements** Serializable {

**private** **HashMap**<**String**, **String**> map;

**public** **HashMap**<**String**, **String**> **getMap**() {

**return** map;

}

**public** **void** **setMap**(**HashMap**<**String**, **String**> map) {

**this**.map = map;

}

}

# Handler的基础用法

## 创建Message对象

创建Message对象的时候，有三种方式，这三种方式有什么区别呢？

* Message msg = new Message();
* Message msg1 = handler1.obtainMessage();
* Message msg2 = Message.obtain();

Message msg = new Message();

这种就是直接初始化一个Message对象，没有什么特别的。

Message msg1 = handler1.obtainMessage();

从整个Messge池中返回一个新的Message实例，通过obtainMessage能避免重复Message创建对象。 并且绑定这个handler到target上

public final Message obtainMessage() {

return Message.obtain(this);

}

public static Message obtain(Handler h) {

Message m = obtain();

m.target = h;

return m;

}

/\*\*

\* Return a new Message instance from the global pool. Allows us to

\* avoid allocating new objects in many cases.

\*/

public static Message obtain() {

synchronized (sPoolSync) {

if (sPool != null) {

Message m = sPool;

sPool = m.next;

m.next = null;

m.flags = 0; // clear in-use flag

sPoolSize--;

return m;

}

}

return new Message();

}

Message msg2 = Message.obtain();

同上

可以看到，第二种跟第三种其实是一样的，都可以避免重复创建Message对象，所以建议用第二种或者第三种任何一个创建Message对象。

## Message的arg1，arg2和what



## 简单的使用方式

private Handler handler = new Handler() {

@Override

public void handleMessage(Message msg) {

if(msg.what = 10086) {

Toast.makeText(MainActivity.this, "hanlder" + msg.arg1 , Toast.LENGTH\_SHORT).show();

}

}

};

new Thread(new Runnable() {

@Override

public void run() {

Message msg = Message.obtain();

msg.arg1 = 1;

msg.what = 10086;

handler.sendMessage(msg);

}

}).start();

# Handler机制原理

## 概述

andriod提供了Handler 和 Looper 来满足线程间的通信。Handler先进先出原则。Looper类用来管理特定线程内对象之间的消息交换(MessageExchange)。  
1)Looper: 一个线程可以产生一个Looper对象，由它来管理此线程里的MessageQueue(消息队列)。   
2)Handler: 你可以构造Handler对象来与Looper沟通，以便push新消息到MessageQueue里;或者接收Looper从Message Queue取出)所送来的消息。  
3) Message Queue(消息队列):用来存放线程放入的消息。

1. 线程：UIthread 通常就是main thread，而[Android](http://lib.csdn.net/base/15" \o "undefined" \t "http://blog.csdn.net/itachi85/article/details/_blank)启动程序时会替它建立一个MessageQueue。

## 子线程初始化handler，相关注意

handler初始化的时候会使用一个looper，这个looper在子线程中没有，所以子线程不能初始化handler，

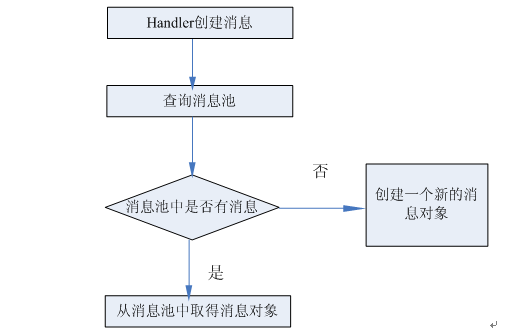
handler = **new** Handler(){}；**(错误！！！！)**

当然也可以给handler传递一个主线程的looper，那么就可以初始化

handler = **new** Handler(**Looper**.getMainLooper()){}；**(正确)**

## Handler创建消息

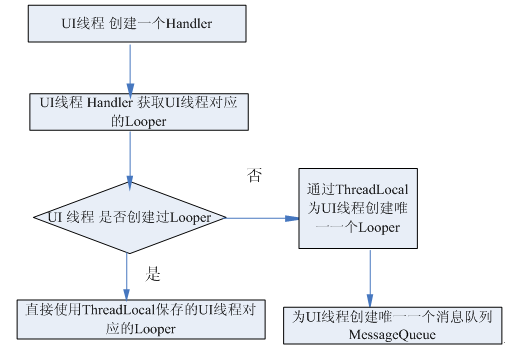
每一个消息都需要被指定的Handler处理，通过Handler创建消息便可以完成此功能。Android消息机制中引入了消息池。Handler创建消息时首先查询消息池中是否有消息存在，如果有直接从消息池中取得，如果没有则重新初始化一个消息实例。使用消息池的好处是：消息不被使用时，并不作为垃圾回收，而是放入消息池，可供下次Handler创建消息时使用。消息池提高了消息对象的复用，减少系统垃圾回收的次数。消息的创建流程如图所示。



## Handler发送消息

UI主线程初始化时会通过ThreadLocal创建一个Looper，该Looper与UI主线程一一对应。使用ThreadLocal的目的是保证每一个线程只创建唯一一个Looper。Handler初始化的时候直接获取已创建的Looper。Looper初始化的时候会创建一个消息队列MessageQueue。至此，主线程、消息循环、消息队列之间的关系是1:1:1。

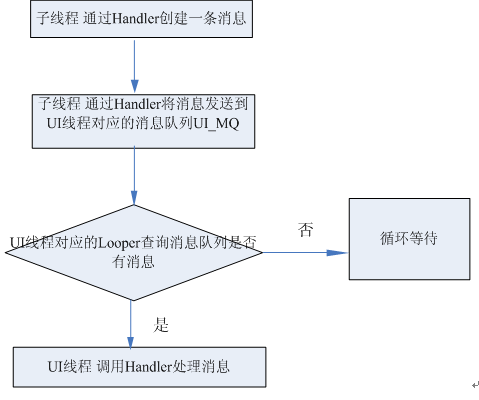
Handler、Looper、MessageQueue的初始化流程如图所示:



Hander持有对UI主线程消息队列MessageQueue和消息循环Looper的引用，子线程可以通过Handler将消息发送到UI线程的消息队列MessageQueue中。

## Handler处理消息

UI主线程通过Looper循环查询消息队列UI\_MQ，当发现有消息存在时会将消息从消息队列中取出。首先分析消息，通过消息的参数判断该消息对应的Handler，然后将消息分发到指定的Handler进行处理。子线程通过Handler、Looper与UI主线程通信的流程如图所示。



## 制作欢迎界面使用sendEmptyMessageDelayed

**public** **class** **WelcomeActivity** **extends** **Activity**{

**private** **boolean** isgo = **true**;**//是否继续执行handle里的跳转**

**private** **Handler** handler = **new** Handler(){

**public** **void** **handleMessage**(android.os.**Message** msg) {

**if** (isgo) {

startActivity(**new** Intent(**WelcomeActivity**.**this**, **MainActivity**.**class**));

finish();

}

}

};

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_welcome**);

**Button** button = (**Button**) findViewById(**R**.**id**.**btn**);

button.setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** **onClick**(**View** v) {

isgo = **false**;

startActivity(**new** Intent(**WelcomeActivity**.**this**, **MainActivity**.**class**));

finish();

}

});

handler.sendEmptyMessageDelayed(0, 3000);

}

@Override

**public** **void** **onBackPressed**() {

**super**.onBackPressed();

isgo = **false**;

}

}

## 使用handler.post()方法，不会执行handleMessage()方法

我们来看下post的源码吧：

public final boolean post(Runnable r){

return sendMessageDelayed(getPostMessage(r), 0);

}

......

public final boolean sendMessageDelayed(Message msg, long delayMillis) {

if (delayMillis < 0) {

delayMillis = 0;

}

return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);

}

......

private final Message getPostMessage(Runnable r) {

Message m = Message.obtain();

m.callback = r;

return m;

}

在这个方法中将消息的callback字段的值指定为传入的Runnable对象。咦？这个callback字段看起来有些眼熟啊，喔！在Handler的dispatchMessage()方法中原来有做一个检查，如果Message的callback等于null才会去调用handleMessage()方法，否则就调用handleCallback()方法

所以使用handler.post()方法，不会执行初始化handler时，里面的handleMessage()方法

## 用Post在子线程里操作主线程

### Post基本用法

**new** Thread(){

**public** **void** **run**() {

mHandler.post(**new** Runnable()

@Override

**public** **void** **run**() {

tv.setText(text);

}

});

}

}.start();

### 延迟1s执行这个Runnable

postDelayed(**this**, 1000);

### Activity.runOnUiThread

runOnUiThread(**new** Runnable() {

@Override

**public** **void** **run**() {

tv.setText(text);

}

});

### View.post

tv.post(**new** Runnable() {

@Override

**public** **void** **run**() {

tv.setText(text);

}

});

## 使用Post的分析Handler机制

从表面上来看，给post方法传了个Runnable，像是开了个子线程，可是在子线程里并不能更新UI啊，那么问题来了，这是怎么个情况呢？带着这个疑惑，来翻翻Handler的源码：

### Handler源码

public final boolean sendEmptyMessage(int what) {

return sendEmptyMessageDelayed(what, 0);

}

public final boolean sendEmptyMessageDelayed(int what, long delayMillis) {

Message msg = Message.obtain();

msg.what = what;

return sendMessageDelayed(msg, delayMillis);

}

将我们传入的参数封装成了一个消息，然后调用sendMessageDelayed

public final boolean sendMessageDelayed(Message msg, long delayMillis) {

if (delayMillis < 0) {

delayMillis = 0;

}

return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);

}

public boolean sendMessageAtTime(Message msg, long uptimeMillis) {

MessageQueue queue = mQueue;

if (queue == null) {

RuntimeException e = new RuntimeException(this + " sendMessageAtTime() called with no mQueue");

Log.w("Looper", e.getMessage(), e);

return false;

}

return enqueueMessage(queue, msg, uptimeMillis);

}

### 再来看post()

public final boolean post(Runnable r) {

return sendMessageDelayed(getPostMessage(r), 0);//getPostMessage方法是两种发送消息的不同之处

}

方法只有一句，内部实现和普通的sendMessage是一样的，但是只有一点不同,那就是 getPostMessage(r) 这个方法：

private static Message getPostMessage(Runnable r) {

Message m = Message.obtain();

m.callback = r;

return m;

}

这个方法我们发现也是将我们传入的参数封装成了一个消息，只是这次是m.callback = r,刚才是msg.what=what,至于Message的这些属性就不看了，看到这里，我们只是知道了post和sendMessage原理都是封装成Message，但是还是不清楚Handler的整个机制是什么样子，继续探究下去。

### Android消息机制

刚才看到那两个方法到最终都调用了sendMessageAtTime

public boolean sendMessageAtTime(Message msg, long uptimeMillis) {

MessageQueue queue = mQueue;

if (queue == null) {

RuntimeException e = new RuntimeException(this + " sendMessageAtTime() called with no mQueue");

Log.w("Looper", e.getMessage(), e);

return false;

}

return enqueueMessage(queue, msg, uptimeMillis);

}

这个方法又调用了 enqueueMessage，看名字应该是把消息加入队列的意思，点进去看下：

private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {

msg.target = this;

if (mAsynchronous) {

msg.setAsynchronous(true);

}

return queue.enqueueMessage(msg, uptimeMillis);

}

mAsynchronous这个异步有关的先不管，继续将参数传给了queue的enqueueMessage方法，至于那个msg的target的赋值我们后面再看，现在继续进入MessageQueue类的enqueueMessage方法，方法较长，我们看看关键的几行：

Message prev;

for (;;) {

prev = p;

p = p.next;

if (p == null || when < p.when) {

break;

}

if (needWake && p.isAsynchronous()) {

needWake = false;

}

}

msg.next = p; // invariant: p == prev.next

prev.next = msg;

果然像方法名说的一样，一个无限循环将消息加入到消息队列中（链表的形式）,但是有放就有拿，这个消息怎样把它取出来呢？翻看MessageQueue的方法，我们找到了next()，代码太长，不赘述，我们知道它是用来把消息取出来的就行了。不过这个方法是在什么地方调用的呢，不是在Handler中，我们找到了Looper这个关键人物，我叫他环形使者，专门负责从消息队列中拿消息，关键代码如下：

for (;;) {

Message msg = queue.next(); // might block

...

msg.target.dispatchMessage(msg);

...

msg.recycleUnchecked();

}

简单明了，我们看到了我们刚才说的msg.target，刚才在Handler中赋值了msg.target=this，所以我们来看Handler中的dispatchMessage：

public void dispatchMessage(Message msg) {

if (msg.callback != null) {

handleCallback(msg);

} else {

if (mCallback != null) {

if (mCallback.handleMessage(msg)) {

return;

}

}

handleMessage(msg);

}

}

* msg的callback不为空，调用handleCallback方法（message.callback.run()）
* mCallback不为空，调用mCallback.handleMessage(msg)
* 最后如果其他都为空，执行Handler自身的 handleMessage(msg) 方法

msg的callback应该已经想到是什么了，就是我们通过Handler.post(Runnable r)传入的Runnable的run方法，这里就要提提java基础了，直接调用线程的run方法相当于是在一个普通的类调用方法，还是在当前线程执行，并不会开启新的线程。所以到了这里，我们解决了开始的疑惑，为什么在post中传了个Runnable还是在主线程中可以更新UI。继续看如果msg.callback为空的情况下的mCallback，这个要看看构造方法：

1.

public Handler() {

this(null, false);

}

2.

public Handler(Callback callback) {

this(callback, false);

}

3.

public Handler(Looper looper) {

this(looper, null, false);

}

4.

public Handler(Looper looper, Callback callback) {

this(looper, callback, false);

}

5.

public Handler(boolean async) {

this(null, async);

}

6.

public Handler(Callback callback, boolean async) {

if (FIND\_POTENTIAL\_LEAKS) {

final Class<? extends Handler> klass = getClass();

if ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocalClass()) &&

(klass.getModifiers() & Modifier.STATIC) == 0) {

Log.w(TAG, "The following Handler class should be static or leaks might occur: " +

klass.getCanonicalName());

}

}

mLooper = Looper.myLooper();

if (mLooper == null) {

throw new RuntimeException(

"Can't create handler inside thread that has not called Looper.prepare()");

}

mQueue = mLooper.mQueue;

mCallback = callback;

mAsynchronous = async;

}

7.

public Handler(Looper looper, Callback callback, boolean async) {

mLooper = looper;

mQueue = looper.mQueue;

mCallback = callback;

mAsynchronous = async;

}

具体的实现就只有最后两个，已经知道mCallback是怎么来的了，在构造方法中传入就行。最后如果这两个回调都为空的话就执行Handler自身的handleMessage(msg)方法，也就是我们熟知的新建Handler重写的那个handleMessage方法。

**Looper**

看到了这里有一个疑惑，那就是我们在新建Handler的时候并没有传入任何参数，也没有哪里显示调用了Looper有关方法，那Looper的创建以及方法调用在哪里呢？其实这些东西Android本身已经帮我们做了，在程序入口ActivityThread的main方法里面我们可以找到：

public static void main(String[] args) {

...

Looper.prepareMainLooper();

...

Looper.loop();

...

}

**总结**

* 已经大概梳理了一下Handler的消息机制，以及post方法和我们常用的sendMessage方法的区别。来总结一下，主要涉及四个类Handler、Message、MessageQueue、Looper：
* 新建Handler，通过sendMessage或者post发送消息，Handler调用sendMessageAtTime将Message交给MessageQueue
* MessageQueue.enqueueMessage方法将Message以链表的形式放入队列中
* Looper的loop方法循环调用MessageQueue.next()取出消息，并且调用Handler的dispatchMessage来处理消息
* 在dispatchMessage中，分别判断msg.callback、mCallback也就是post方法或者构造方法传入的不为空就执行他们的回调，如果都为空就执行我们最常用重写的handleMessage

## 最后谈谈handler的内存泄露问题

再来看看我们的新建Handler的代码：

private Handler mHandler = new Handler() {

@Override

public void handleMessage(Message msg) {

...

}

};

当使用内部类（包括匿名类）来创建Handler的时候，Handler对象会隐式地持有Activity的引用。而Handler通常会伴随着一个耗时的后台线程一起出现，这个后台线程在任务执行完毕后发送消息去更新UI。然而，如果用户在网络请求过程中关闭了Activity，正常情况下，Activity不再被使用，它就有可能在GC检查时被回收掉，但由于这时线程尚未执行完，而该线程持有Handler的引用（不然它怎么发消息给Handler？），这个Handler又持有Activity的引用，就导致该Activity无法被回收（即内存泄露），直到网络请求结束。

另外，如果执行了Handler的postDelayed()方法，那么在设定的delay到达之前，会有一条MessageQueue -> Message -> Handler -> Activity的链，导致你的Activity被持有引用而无法被回收。

**解决方法之一，使用弱引用：**

static class MyHandler extends Handler {

WeakReference<Activity > mActivityReference;

MyHandler(Activity activity) {

mActivityReference= new WeakReference<Activity>(activity);

}

@Override

public void handleMessage(Message msg) {

final Activity activity = mActivityReference.get();

if (activity != null) {

mImageView.setImageBitmap(mBitmap);

}

}

}

# Activity启动之Handle

## 从Tread的创建流程开始

在这一个环节，我们将一起一步步的分析Thread的创建流程。话不多说，直接代码里看。

### 线程创建的起始点init()

*//创建Thread的公有构造函数，都调用的都是这个私有的init()方法。我们看看到底干什么了。*

public Thread() {  
 init(null, null, "Thread-" + *nextThreadNum*(), 0);  
}

......

public Thread(Runnable target, String name) {  
 init(null, target, name, 0);  
}

*/\*\*  
 \** ***@param*** *线程组  
 \** ***@param*** *就是我们平时接触最多的Runnable同学  
 \** ***@param*** *指定线程的名称  
 \** ***@param*** *指定线程堆栈的大小  
 \*/*private void init(ThreadGroup g, Runnable target, String name, long stackSize) {  
 // 获取当前正在运行的线程  
 // 当前正在运行的线程就是该我们要创建的线程的父线程  
 // 我们要创建的线程会从父线程那继承一些参数过来  
 // 注意哦，此时仍然是在原来的线程，新线程此时还没有创建的哦！  
 Thread parent = currentThread();  
 if (g == null) {  
 g = parent.getThreadGroup();//如果没有指定ThreadGroup，将获取父线程的TreadGroup  
 }  
 //将ThreadGroup中的就绪线程计数器增加一。注意，此时线程还并没有被真正加入到ThreadGroup中。  
 g.addUnstarted();  
 this.group = g;//将Thread实例的group赋值。从这里开始线程就拥有ThreadGroup了。  
 this.target = target;//给Thread实例的Runnable赋值。以后start()的时候执行的就是它了。  
 this.priority = parent.getPriority();//设置线程的优先权重为父线程的权重  
 this.daemon = parent.isDaemon();//根据父线程是否是守护线程来确定Thread实例是否是守护线程。  
 setName(name);//设置线程的名称   
 init2(parent);//纳尼？又一个初始化，参数还是父线程。不急，稍后在看。  
 /\* Stash the specified stack size in case the VM cares \*/  
 this.stackSize = stackSize;//设置线程的堆栈大小  
 tid = nextThreadID();//线程的id。这是个静态变量，调用这个方法会自增，然后作为线程的id。  
}

在Thread的init()方法中，比较重要的是会通过一个currentThread()这样的native函数通过底层从虚拟机中获取到当前运行的线程。所以在Thread初始化的时候，仍然是在创建它的线程中。不难猜测出，其实Java层的Thread只是对底层的封装而已。

### 第二个init2()

private void init2(Thread parent) {  
 this.contextClassLoader = parent.getContextClassLoader(); //设置ClassLoader成员变量  
 this.inheritedAccessControlContext = AccessController.getContext(); //设置访问权限控制环境  
 if (parent.inheritableThreadLocals != null) {  
 /\*创建Thread实例的ThreadLoacaleMap。需要用到父线程的ThreadLocaleMap，  
 目的是为了将父线程中的变量副本拷贝一份到当前线程中。  
 ThreadLocaleMap是一个Entry型的数组，Thread实例会将变量副本保存在这里面\*/  
 this.inheritableThreadLocals = ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);  
 }  
}

至此，我们的Thread就初始化完成了，Thread的几个重要成员变量都赋值了

## 启动线程，开车啦！

通常，我们这样了启动一条线程。

Thread threadDemo = new Thread(() -> {  
  
});  
threadDemo.start();

让我们一起点进start()。探寻start()背后的秘密。

/\*这个方法是加了锁的。原因是避免开发者在其它线程调用同一个Thread实例的这个方法，从而尽量避免抛出异常。  
这个方法之所以能够执行我们传入的Runnable里的run()方法，是应为JVM调用了Thread实例的run()方法。\*/  
public synchronized void start() {  
 //检查线程状态是否为0，为0表示是一个新状态，即还没被start()过。不为0就抛出异常。  
 //就是说，我们一个Thread实例，我们只能调用一次start()方法。  
 if (threadStatus != 0) throw new IllegalThreadStateException();  
 /\*从这里开始才真正的线程加入到ThreadGroup组里。再重复一次，  
 前面只是把nUnstartedThreads这个计数器进行了增量，并没有添加线程。  
 同时，当线程启动了之后，nUnstartedThreads计数器会-1。因为就绪状态的线程少了一条啊！\*/  
 group.add(this);  
 started = false;  
 try {  
 //又是个Native方法。这里交由JVM处理，会调用Thread实例的run()方法。  
 nativeCreate(this, stackSize, daemon);  
 started = true;  
 } finally {  
 try {  
 if (!started) {  
 //如果没有被启动成功，Thread将会被移除ThreadGroup，同时nUnstartedThreads计数器又增量1  
 group.threadStartFailed(this);   
 }  
 } catch (Throwable ignore) {  
 }  
 }  
}

最精华的函数是nativeCreate(this, stackSize, daemon)，会去调用底层的JNI函数Thread\_nativeCreate()，进一步的会调用底层的Thread类的Thread::CreateNativeThread()函数。

Thread::CreateNativeThread()函数在/art/runtime/thread.cc文件中（注：本例用的是6.0.0-r1的源码）。它会在去创建一个c/c++层的Thread对象，并且会关联Java层的Thread对象（其实就是保存一个Java层Thread对象的引用而已）。接着，会通过c/c++层的pthread\_create()函数去创建并启动一个新线程。这条代码必须要看看了：

pthread\_create\_result = pthread\_create(&new\_pthread, &attr, Thread::CreateCallback, child\_thread);

这里我们需要注意第三个参数位置的Thread::CreateCallback，它会返回一个Java层Thread类的run()方法指针，在Linux层的pthread线程创建成功后，将会调用这个run()方法。这就是为什么我们调用start()方法后，run()方法会被调用的原因。

从上面的分析我们可以知道，其实Java的线程Thread还是用的Linux那一套 pthread 的东西，并且一条线程真正创建并运行在虚拟机中时，是在调用start()方法之后。所以，如果你创建了一条线程，但是从没调用过它的start()方法，就不会有条新线程生成，此时的Thread对象和主线程里的一个普通对象没什么区别。如果你企图调用 run() 方法去试图启动你的线程，那真是大错特错了！这样不过相当于在主线程中调用了一个Java方法而已。

所以，Java中的线程在Android中实际上走的还是Linux的pthread那一套。

/\* 没错，就是这么简单！仅仅调用了Runnable类型的成员变量target的run()方法。  
 至此，我们需要执行的代码就执行起来了。  
 至于这个@Overrid的存在，完全是因为Thread本身也是一个Runnable！  
 就是说，我们的Thread也可以作为一个Runnable来使用。\*/  
@Override  
public void run() {  
 if (target != null) {  
 target.run();  
 }  
}

看，如果不调用start()方法，你可以把Thread当作一个Handler去使用！！

Thread thread1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println(Thread.*currentThread*().getName());  
 }  
}, "Thread\_1");  
  
Thread thread2 = new Thread(thread1, "Thread\_2");  
thread2.start();

输出：Thread\_2

## 几个常见的线程手段(操作)

### Thread.sleep()那不可告人的秘密

我们平时使用Thread.sleep()的频率也比较高，所以我们在一起研究研究Thread.sleep()被调用的时候发生了什么。在开始之前，先介绍一个概念——纳秒。1纳秒=十亿分之一秒。可见用它计时将会非常的精准。但是由于设备限制，这个值有时候并不是那么准确，但还是比毫秒的控制粒度小很多。

//平时我们调用的Thread.sleep(long)最后调用到这个方法来，后一个陌生一点的参数就是纳秒。  
//你可以在纳秒级控制线程。  
public static void sleep(long millis, int nanos) throws InterruptedException {  
 //下面三个检测毫秒和纳秒的设置是否合法。  
 if (millis < 0) {  
 throw new IllegalArgumentException("millis < 0: " + millis);  
 }  
 if (nanos < 0) {  
 throw new IllegalArgumentException("nanos < 0: " + nanos);  
 }  
 if (nanos > 999999) {  
 throw new IllegalArgumentException("nanos > 999999: " + nanos);  
 }  
 if (millis == 0 && nanos == 0) {  
 //当睡眠时间为0时，检测线程是否中断，并清除线程的中断状态标记。这是个Native的方法。  
 if (Thread.*interrupted*()) {  
 /\*如果线程被设置了中断状态为true了(调用Thread.interrupt())。那么他将抛出异常。  
 如果在catch住这个异常之后return线程，那么线程就停止了。  
 需要注意，在调用了Thread.sleep()之后，再调用isInterrupted()得到的结果永远是False。  
 别忘了Thread.interrupted()在检测的同时还会清除标记位置哦！\*/  
 throw new InterruptedException();   
 }  
 return;  
 }  
 long start = System.*nanoTime*(); //类似System.currentTimeMillis()。但是获取的是纳秒，可能不准。  
 long duration = (millis \* NANOS\_PER\_MILLI) + nanos;  
 Object lock = currentThread().lock; //获得当前线程的锁。  
 synchronized (lock) { //对当前线程的锁对象进行同步操作  
 while (true) {  
 //这里又是一个Native的方法，并且也会抛出InterruptedException异常。  
 *sleep*(lock, millis, nanos);   
 //据我估计，调用这个函数睡眠的时长是不确定的。  
 long now = System.*nanoTime*();  
 long elapsed = now - start; //计算线程睡了多久了  
 if (elapsed >= duration) {//如果当前睡眠时长，已经满足我们的需求，就退出循环，睡眠结束。  
 break;  
 }  
 duration -= elapsed; //减去已经睡眠的时间，重新计算需要睡眠的时长。  
 start = now;  
 millis = duration / NANOS\_PER\_MILLI; //重新计算毫秒部分  
 nanos = (int) (duration % NANOS\_PER\_MILLI); //重新计算微秒部分  
 }  
 }  
}

通过上面的分析可以知道，使线程休眠的核心方法就是一个Native函数sleep(lock, millis, nanos)。这个sleep()对应底层的一个JNI函数，这个JNI函数最终会调用到c/c++中对应的Thread的条件变量的 TimedWait()函数。这个条件变量是应该是Android中自己定义的条件变量，当然，这里的TimedWait()函数自然也是Android自己实现的。在这个函数里，Android直接使用了Linux的futex()函数。这个futex()函数会调用syscall()函数，通过一种名为【快速用户区互斥锁】的锁去执行锁定的。futex()的效率比phtread\_cond\_wait()要高很多。

Android为了确保休眠的准确性，在这里还使用了一个while()循环，在每次线程从底层被唤醒后，检查一下是否休眠够了足够的时长。如果不够就让它继续休眠。

同时，需要注意一点，如果线程的interruted状态在调用sleep()方法时被设置为true，那么在开始休眠循环前会抛出InterruptedException异常。

### Thread.yield()究竟隐藏了什么？

这个方法是Native的。调用这个方法可以提示cpu，当前线程将放弃目前cpu的使用权，和其它线程重新一起争夺新的cpu使用权限。当前线程可能再次获得执行，也可能没获得。

### 无处不在的wait()究竟是什么？

大家一定经常见到，不论是哪一个对象的实例，都会在最下面出现几个名为wait()的方法。等待？它们究竟是怎样的一种存在，让我们一起点击去看看。哎哟我去，都是Native函数啊。那就看看文档它到底是什么吧。

根据文档的描述，wait()配合notify()和notifyAll()能够实现线程间通讯，即同步。在线程中调用wait()必须在同步代码块中调用，否则会抛出IllegalMonitorStateException异常。因为wait()函数需要释放相应对象的锁。当线程执行到wait()时，对象会把当前线程放入自己的线程池中，并且释放锁，然后阻塞在这个地方。直到该对象调用了notify()或者notifyAll()后，该线程才能重新获得，或者有可能获得对象的锁，然后继续执行后面的语句。呃。。。好吧，在说明一下notify()和notifyAll()的区别。

* notify(),调用notify()后，对象会从自己的线程池中(也就是对该对象调用了wait()函数的线程)随机挑选一条线程去唤醒它。也就是一次只能唤醒一条线程。如果在多线程情况下，只调用一次notify()，那么只有一条线程能被唤醒，其它线程会一直在
* notifyAll(),调用notifyAll()后，对象会唤醒自己的线程池中的所有线程，然后这些线程就会一起抢夺对象的锁。

## Looper、Handler、MessageQueue之间的爱恨情仇

我们可能过去都写过形如这样的代码：

new Thread(new Runnable() {  
 @Override  
 public void run() {  
 Looper.*prepare*();  
 Handler handler = new Handler() {  
 @Override  
 public void handleMessage(Message msg) {  
 super.handleMessage(msg);  
 }  
 };  
 Looper.*loop*();  
 }  
}).start();

很多同学知道，在线程中使用Handler时(除了Android主线程)必须把它放在Looper.prepare()和Looper.loop()之间。否则会抛出RuntimeException异常。但是为什么要这么做呢？下面我们一起来扒一扒这其中的内幕。

### 从Looper.prepare()开始

当Looper.prepare()被调用时，发生了什么？

private static void prepare(boolean quitAllowed) {  
 //先尝试获取是否已经存在一个Looper在当前线程中，如果有就抛个异常。  
 if (sThreadLocal.get() != null) {   
 //这就是为什么我们不能在一个Thread中调用两次Looper.prepare()的原因。  
 throw new RuntimeException("Only one Looper may be created per thread");  
 }  
 sThreadLocal.set(new Looper(quitAllowed)); //首次调用的话，就创建一个新的Looper。  
}  
//Looper的私有构造函数  
private Looper(boolean quitAllowed) {  
 mQueue = new MessageQueue(quitAllowed); //创建新的MessageQueue，稍后在来扒它。  
 mThread = Thread.*currentThread*(); //把当前的线程赋值给mThread。  
}

经过上面的分析，我们已经知道Looper.prepare()调用之后发生了什么。

但是问题来了！sThreadLocal是个静态的ThreadLocal<Looper> 实例(在Android中ThreadLocal的范型固定为Looper)。那么，Looper.prepare()既然是个静态方法，Looper是如何确定现在应该和哪一个线程建立绑定关系的呢？我们接着往里扒。来看看ThreadLocal的get()、set()方法。

public void set(T value) {  
 Thread t = Thread.*currentThread*(); //同样先获取到当前的线程  
 ThreadLocalMap map = getMap(t); //获取线程的ThreadLocalMap  
 if (map != null)  
 map.set(this, value); //储存键值对  
 else  
 createMap(t, value);  
}  
public T get() {  
 Thread t = Thread.*currentThread*(); //重点啊！获取到了当前运行的线程。  
 //取出当前线程的ThreadLocalMap。这个东西是个重点，前面已经提到过。忘了的同学在前面再看看。  
 ThreadLocalMap map = getMap(t);   
 if (map != null) {  
 ThreadLocalMap.Entry e = map.getEntry(this);  
 //可以看出，每条线程的ThreadLocalMap中都有一个<ThreadLocal,Looper>键值对。  
 // 绑定关系就是通过这个键值对建立的。  
 if (e != null)  
 return (T)e.value;  
 }  
 return setInitialValue();  
}

..................................................

剩下的知识：<https://www.jianshu.com/p/8862bd2b6a29>

..................................................

## 总结

* Android中Thread在创建时进行初始化，会使用当前线程作为父线程，并继承它的一些配置。
* Thread初始化时会被添加到指定/父线程的ThreadGroup中进行管理。
* Thread正真启动是一个native函数完成的。
* 在Android的线程间通信中，需要先创建Looper，就是调用Looper.prepare()。这个过程中会自动依赖当前Thread，并且创建MessageQueue。经过上一步，就可以创建Handler了，默认情况下，Handler会自动依赖当前线程的Looper，从而依赖相应的MessageQueue，也就知道该把消息放在哪个地方了。MessageQueue通过Message.next实现了一个单链表结构来缓存Message。消息需要送达Handler处理，还必须调用Looper.loop()启动线程的消息泵送循环。loop()内部是无限循环，阻塞在MessageQueue的next()方法上，因为next()方法内部也是一个无限循环，直到成功从链表中抽取一条消息返回为止。然后，在loop()方法中继续进行处理，主要就是把消息派送到目标Handler中。接着进入下一次循环，等待下一条消息。由于这个机制，线程就相当于阻塞在loop()这了。

# Activity启动之流程

## 一张图明白Activity的启动流程

图片地址：[https://upload-images.jianshu.io/upload\_images/1869462-882b8e0470adf85a.jpg](https://upload-images.jianshu.io/upload_images/1869462-882b8e0470adf85a.jpg?imageMogr2/auto-orient/)

本篇主要讲的是从一个App启动，到Activity执行onCreate()的流程。后面关于Activity的生命周期相信大家基本都耳熟能详了。

上图中我把涉及到的类名方法命均列出来了，你可以看着流程，打开源码跟着过一遍。相信在过完一遍之后，在今后的开发中你会更加自信！

上图乍一看可能感觉有些眼花缭乱，但请不要惧怕。其实根本就没什么东西，你只需要从蓝色箭头开始看下去，会发现一下就看完了。在结合下面简要的分析，3分钟内你就能搞明白Activity的启动流程。

### 一切从main()方法开始

Android中，一个应用程序的开始可以说就是从ActivityThread.java中的main()方法开始的。Java的程序入口就是main()方法。从这点而言，我们可以把它想成是一个Java程序（注意，不是说Android是个Java程序哦）去理解。

从上图可以看到，main()方法中主要做的事情有：

1. 初始化主线程的Looper、主Handler。并使主线程进入等待接收Message消息的无限循环状态。下面是main()方法中比较关键的代码：

public static void main(String[] args){  
...

//初始化Looper  
 Looper.*prepareMainLooper*();  
...

//实例化一个ActivityThread  
 ActivityThread thread = new ActivityThread();

//这个方法最后就是为了发送出创建Application的消息  
 thread.attach(false);  
...

//主线程进入无限循环状态，等待接收消息  
 Looper.*loop*();  
}

1. 调用attach()方法，主要就是为了发送出初始化Application的消息。这个流程下文会再捋一捋。

### 创建Application的消息是如何发送的呢？

上面提到过，ActivityThread的attach()方法最终的目的是发送出一条创建Application的消息——H.BIND\_APPLICATION，到主线程的主Handler中。那我们来看看attach()方法干了啥。attach()关键代码：

public void attach(boolean system){  
...

//获得IActivityManager实例，下面会看看它是个啥  
 final IActivityManager mgr = ActivityManagerNative.getDefault();  
 try {

//看见没？关键啊。mAppThread这个参数下面也会说一下  
 mgr.attachApplication(mAppThread);  
 } catch (RemoteException ex) {  
 throw ex.rethrowFromSystemServer();  
 }  
...  
}

#### IActivityManager mgr是个啥？

从上图也可以看到，IActivityManager是一个接口，当我们调用ActivityManagerNative.getDefault()获得的实际是一个代理类的实例——ActivityManagerProxy，这个东西实现了IActivityManager接口。打开源码你会发现，ActivityManagerProxy是ActivityManagerNative的一个内部类。可以看出，Android团队在设计的过程中是实践了最小惊异原则的，就是把相关的东西尽量放在一起。那么既然是个代理类，它究竟代理了谁？代码里看看喽。

1. 先看ActivityManagerProxy的构造函数：

public ActivityManagerProxy(IBinder remote) {  
 mRemote = remote;  
}

这个构造函数非常的简单。首先它需要一个IBinder参数，然后赋值给mRemote变量。这个mRemote显然是ActivityManagerProxy的成员变量，对它的操作是由ActivityManagerProxy来代理间接进行的。这样设计的好处是保护了mRemote，并且能够在操作mRemote前执行一些别的事务，并且我们是以IActivityManager的身份来进行这些操作的！这就非常巧妙了。

1. 那么这个构造函数是在那调用的呢？

static public IActivityManager asInterface(IBinder obj) {  
 if (obj == null) {  
 return null;  
 }  
 IActivityManager in =(IActivityManager)obj.queryLocalInterface(descriptor);  
 //先检查一下有没有  
 if (in != null) {  
 return in;  
 }  
...

//这个地方调用了构造函数  
 return new ActivityManagerProxy(obj);  
}

上面这个方法是ActivityManagerNative中的一个静态方法，它会调用到ActivityManagerProxy的构造方法。然而，这个静态方法也需要一个IBinder作为参数!老夫被绕晕了。但是不怕，咱们继续往找！

1. getDefault()获取到的静态常量gDefault

private static final Singleton<IActivityManager> *gDefault* =  
 new Singleton<IActivityManager>() {  
 protected IActivityManager create() {  
 //重点啊！IBinder实例就是在这里获得的。

IBinder b = ServiceManager.getService("activity");  
 ...  
 //调用了上面的方法。

IActivityManager am = asInterface(b);  
 ...  
 return am;  
 }  
 };

这是ActivityManagerNative的静态常量，它是一个单例。在其中终于获得了前面一直在用的IBinder实例。

IBinder b = ServiceManager.getService("activity");

这里是通过ServiceManager获取到IBinder实例的。如果你以前了解AIDL通讯流程的话。这可能比较好理解一点，这只是通过另一种方式获取IBinder实例罢了。获取IBinder的目的就是为了通过这个IBinder和ActivityManager进行通讯，进而ActivityManager会调度发送H.BIND\_APPLICATION即初始化Application的Message消息。如果之前没接触过Binder机制的话，只需知道这个目的就行了。

1. 再来看看attachApplication(mAppThread)方法。

public void attachApplication(IApplicationThread app){  
...  
 mRemote.transact(ATTACH\_APPLICATION\_TRANSACTION, data, reply, 0);   
...  
}

这个方法我在上图中也体现出来了。这个方法中上面这一句是关键。调用了IBinder实例的tansact()方法，并且把参数app(这个参数稍后就会提到)放到了data中，最终传递给ActivityManager。

现在，我们已经基本知道了IActivityManager是个什么东东了。其实最重要的就是它的一个实现类ActivityManagerProxy，它主要代理了内核中与ActivityManager通讯的Binder实例。下面再看看ApplicationThread mAppThread。

#### ApplicationThread mAppThread又是个啥？

1. 在ActivityThread的成员变量中，你能够发现：

final ApplicationThread mAppThread = new ApplicationThread();

ApplicationThread是作为ActivityThread中的一个常量出现的。这表明系统不希望这个变量中途被修改，可见这个变量具有特定而十分重要的作用。

1. 我们看看他是啥。

private class ApplicationThread extends ApplicationThreadNative{  
...  
}

ApplicationThread是ActivityThread中的一个内部类，为什么没有单独出来写在别的地方呢？我觉得这也是对最小惊异原则的实践。因为ApplicationThread是专门真对这里使用的对象。

1. 它继承自ApplicationThreadNative，我们再看看它是个啥。

public abstract class ApplicationThreadNative extends Binder implements IApplicationThread{  
...  
 //无参构造函数  
 public ApplicationThreadNative() {  
 //这是Binder的  
 attachInterface(this, descriptor);  
 }  
...  
}

那么很明显，ApplicationThread最终也是一个Binder！同时，由于实现了IApplicationThread接口，所以它也是一个IApplicationThread。以上这系对应关系你都可以在上图中找到。

我们在ActivityThread中看到的ApplicationThread使用的构造函数是无参的，所以看上面无参构造函数都干了啥！

Binder的attachInterface(IInterface owner, String descriptor)方法没什么特别的，就是赋值了。

public void attachInterface(IInterface owner, String descriptor) {  
 mOwner = owner;  
 mDescriptor = descriptor;  
}

1. 那么IApplicationThread又是啥？老铁，走着！我们继续挖。

public interface IApplicationThread extends IInterface {  
...  
 String *descriptor* = "android.app.IApplicationThread";  
 //留意下这个参数  
...  
}

好吧，这在上图中没有，挖的有点什么了。但是学习嘛，咱就看看喽。

IApplicationThread是继承了IInterface的一个接口，我们需要关注一下里面的descriptor参数。后面会用它，它是一个标识，查询的时候很重要。

好，我们终于知道attach()方法中出现的两个对象是啥了。ApplicationThread作为IApplicationThread的一个实例，承担了最后发送Activity生命周期、及其它一些消息的任务。也就是说，前面绕了一大圈，最后还是回到这个地方来发送消息。我擦！

也许你会想，既然在ActivityThread中我们已经创建出了ApllicationThread的了，为什么还要绕这么弯路？，当然是为了让系统根据情况来控制这个过程喽，不然为什么要把ApplicationThread传到ActivityManager中呢？

#### ActivityManagerService调度发送初始化消息

经过上面的辗转，ApplicationThread终于到了ActivityManagerService中了。请在上图中找到对应位置！从上图中可以看到，ActivityManagerService中有一这样的方法：

private final boolean attachApplicationLocked(IApplicationThread thread, int pid) {  
...  
 thread.bindApplication();//注意啦！  
...  
}

ApplicationThread以IApplicationThread的身份到了ActivityManagerService中，经过一系列的操作，最终被调用了自己的bindApplication()方法，发出初始化Applicationd的消息。

public final void bindApplication(String processName,  
 ApplicationInfo appInfo,  
 List<ProviderInfo> providers,  
 ComponentName instrumentationName,  
 ProfilerInfo profilerInfo,  
 Bundle instrumentationArgs,  
 IInstrumentationWatcher instrumentationWatcher,  
 IUiAutomationConnection instrumentationUiConnection,  
 int debugMode,  
 boolean enableBinderTracking,  
 boolean trackAllocation,  
 boolean isRestrictedBackupMode,  
 boolean persistent,  
 Configuration config,  
 CompatibilityInfo compatInfo,  
 Map<String, IBinder> services,  
 Bundle coreSettings){  
  
...  
 sendMessage(H.BIND\_APPLICATION, data);  
}

吓屎老纸！这么多参数。这明明很违反参数尽量要少的原则嘛！所以说，有的时候，开发过程中还是很难避免一些参数堆积的情况的。但是，这个地方，我们只要知道最后发了一条H.BIND\_APPLICATION消息，接着程序开始了。

### 收到初始化消息之后的世界

上面我们已经找到初始化Applicaitond的消息是在哪发送的了。现在，需要看一看收到消息后都发生了些什么。

现在上图的H下面找到第一个消息：H.BIND\_APPLICATION。一旦接收到这个消息就开始创建Application了。这个过程是在handleBindApplication()中完成的。看看这个方法。在上图中可以看到对应的方法。

private void handleBindApplication(AppBindData data) {  
...  
 mInstrumentation = (Instrumentation)  
 cl.loadClass(data.instrumentationName.getClassName()).newInstance();  
 //通过反射初始化一个Instrumentation仪表。后面会介绍。  
...  
 Application app = data.info.makeApplication(data.restrictedBackupMode, null);  
 //通过LoadedApp命令创建Application实例  
 mInitialApplication = app;  
...  
 mInstrumentation.callApplicationOnCreate(app);  
 //让仪器调用Application的onCreate()方法  
...  
}

handleBindApplication()是一个很长的方法，但是我为各位看官精选出了上面这几句代码。对于本篇的主题来说，他们是至关重要的。上面短短的代码中出现了几个新对象。下面我会一一道来。

#### Instrumentation仪表，什么鬼？

1.这个叫Instrumentation仪表的东西十分诡异，姑且翻译为仪器吧。字面上看不出任何它是干什么的线索。但是，我们可以打开文档看看喽：Instrumentation会在应用程序的任何代码运行之前被实例化，它能够允许你监视应用程序和系统的所有交互。

2.但是，从上面的代码我们可以看出，Instrumentation确实是在Application初始化之前就被创建了。那么它是如何实现监视应用程序和系统交互的呢？打开这个类你可以发现，最终Apllication的创建，Activity的创建，以及生命周期都会经过这个对象去执行。简单点说，就是把这些操作包装了一层。通过操作Instrumentation进而实现上述的功能。

3.那么这样做究竟有什么好处呢？仔细想想。Instrumentation作为抽象，当我们约定好需要实现的功能之后，我们只需要给Instrumentation仪表添加这些抽象功能，然后调用就好。剩下的，不管怎么实现这些功能，都交给Instrumentation仪器的实现对象就好。啊！这是多态的运用。啊！这是依赖抽象，不依赖具体的实践。啊！这是上层提出需求，底层定义接口，即依赖倒置原则的践行。呵！抽象不过如此。从代码中可以看到，这里实例化Instrumentation的方法是反射！而反射的ClassName是来自于从ActivityManagerService中传过来的Binder的。套路太深！就是为了隐藏具体的实现对象。但是这样耦合性会很低。

4.好了，不瞎扯了。既然在说Instrumentation，那就看看最后调的callApplicationOnCreate()方法。

public void callApplicationOnCreate(Application app) {  
 app.onCreate();  
}

你没看错，它啥也没干。只是调用了一下Application的onCreate()方法。这就是为什么它能够起到监控的作用。在上图中你能够看到Instrumentation，以及它的交互过程。

#### LoadedApk就是data.info哦！

关于它是怎么来的本篇就不说了，以后可能会介绍下。本篇就看流程就好。所以直接进去看它的makeApplication()干了啥，就把Application给创建了。

public Application makeApplication(boolean forceDefaultAppClass,  
 Instrumentation instrumentation) {  
...  
 String appClass = mApplicationInfo.className;  
 //Application的类名。明显是要用反射了。  
...  
 ContextImpl appContext = ContextImpl.createAppContext(mActivityThread, this);  
 //留意下Context  
 app = mActivityThread.mInstrumentation.newApplication( cl, appClass, appContext);  
 //通过仪表创建Application  
...  
}

在这个方法中，我们需要知道的就是，在取得Application的实际类名之后，最终的创建工作还是交由Instrumentation去完成，就像前面所说的一样。

值得留意的是，就像上图所标注的一样，当需要第二次获取Application时，同样只需要调用这个方法就好。“真是方便！”

#### 现在把目光移回Instrumentation

看看newApplication()中是如何完成Application的创建的。

static public Application newApplication(Class<?> clazz  
 , Context context) throws InstantiationException  
 , IllegalAccessException  
 , ClassNotFoundException {  
 Application app = (Application)clazz.newInstance();  
 //反射创建，简单粗暴  
 app.attach(context);  
 //关注下这里，Application被创建后第一个调用的方法。目的是为了绑定Context。  
 return app;  
}

我的天，绕了这么多，这Application可算是创建出来了。快给自己一个小红花吧！

### LaunchActivity

当Application初始化完成后，系统会更具Manifests中的配置的启动Activity发送一个Intent去启动相应的Activity。这个过程本篇先不提，下次再说。主要看流程！

直接的，H就收到了一条LAUNCH\_ACTIVITY的消息。然后开始初始化Activity之旅。收到消息后，真正处理是在ActivityThread中的handleLaunchActivity()中进行的。是不是迫不及待的想要知道发生了啥？快在上图中找到对应的步骤吧！

private void handleLaunchActivity(ActivityClientRecord r  
 , Intent customIntent  
 , String reason) {  
...  
 Activity a = performLaunchActivity(r, customIntent);  
 //妈蛋！又封装到另一个方法中创建了。  
...  
 if (a != null) {  
 ...  
 handleResumeActivity(r.token  
 , false  
 , r.isForward  
 ,!r.activity.mFinished && !r.startsNotResumed  
 , r.lastProcessedSeq, reason);  
 //Activity创建成功就往onResume()走了！  
 ...  
 }  
}

从上面的代码中可以看出...好吧，什么都看不出来！再走一个方法。

private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {  
...  
 activity = mInstrumentation.newActivity(cl, component.getClassName(), r.intent);  
 //通过仪表来创建Activity  
...  
 Application app = r.packageInfo.makeApplication(false, mInstrumentation);  
 //前面说过，是在获取Application  
...  
 activity.attach(appContext  
 , this  
 , getInstrumentation()  
 , r.token  
 ,.ident  
 , app  
 , r.intent  
 , r.activityInfo  
 , title  
 , r.parent  
 , r.embeddedID  
 , r.lastNonConfigurationInstances  
 , config  
 ,r.referrer  
 , r.voiceInteractor  
 , window);  
 //方法怪出现！  
...  
 if (r.isPersistable()) {  
 mInstrumentation.callActivityOnCreate(activity, r.state, r.persistentState);  
 } else {  
 mInstrumentation.callActivityOnCreate(activity, r.state);  
 }  
 //根据是否可持久化选择onCreate()方法。  
...  
}

这个方法内容较多，我们一个个看。

activity = mInstrumentation.newActivity(cl, component.getClassName(), r.intent);

正如前面所说，Activity、Application的创建及生命周期都被承包给Instrumentation仪表了。所以由它来负责。看看Instrumentation干了啥。

public Activity newActivity(ClassLoader cl, String className, Intent intent)  
 throws InstantiationException, IllegalAccessException, ClassNotFoundException {  
 return (Activity)cl.loadClass(className).newInstance();  
 //真的没干啥。反射实例化Activity而已  
}

就是反射出一个Activity而已。

if (r.isPersistable()) {  
 mInstrumentation.callActivityOnCreate(activity, r.state, r.persistentState);  
} else {  
 mInstrumentation.callActivityOnCreate(activity, r.state);  
}

根据是否可持久化选择Activity的onCreate()方法。同样是通过Instrumentation仪表来执行onCreate()的。它两分别对应的onCreate()方法为：

//可获得持久化数据

onCreate(icicle, persistentState);

和

//平时重写的最多的。  
onCreate(icicle);

中间两个方法留意一下就好，就不在解释的，感兴趣的点源码看看。到此，Activity就跑起来了！怎么样？是不是并不复杂。

# Activity启动之View加载和app卡顿分析

## 从setContentView()说起

public class AnalyzeViewFrameworkActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

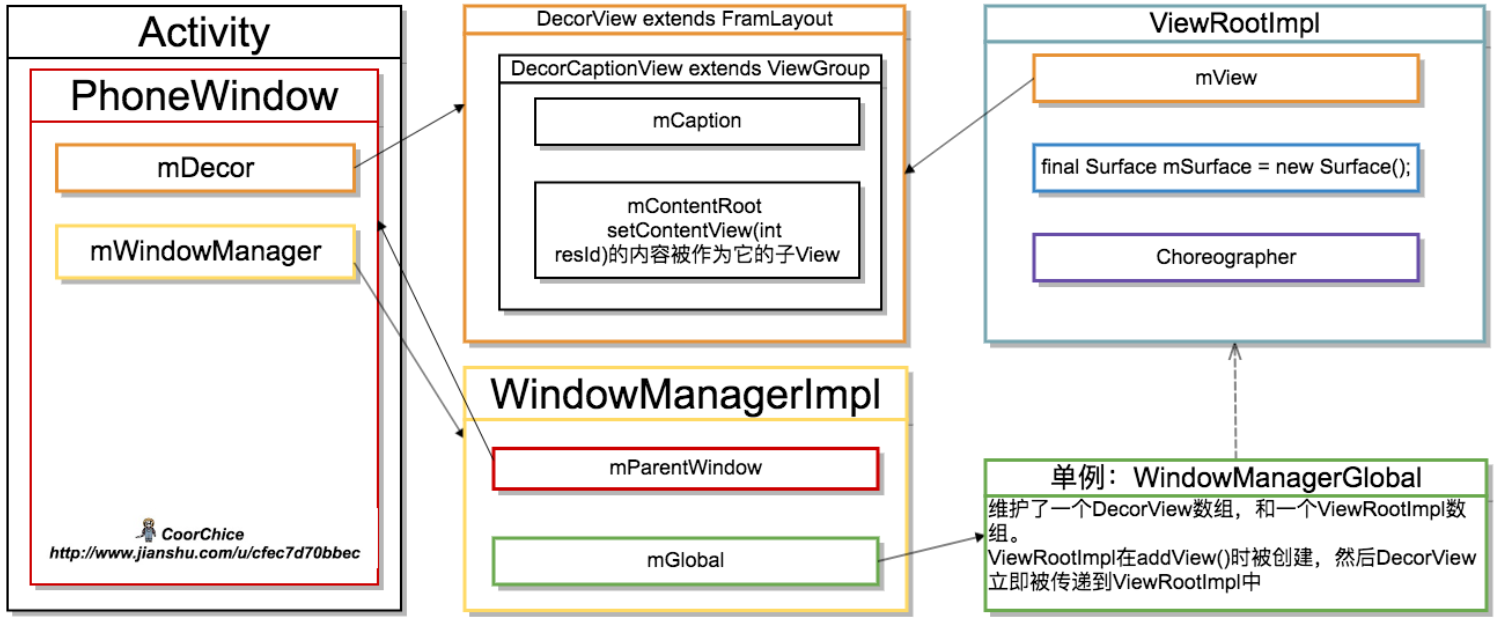
setContentView(R.layout.activity\_analyze\_view\_framwork);

}

}

上面这段代码想必Androider们大都已经不能再熟悉的更多了。但是你知道这样写了之后发生什么了吗？这个布局到底被添加到哪了？我的天，知识点来了！

可能很多同学也知道这个布局是被放到了一个叫做DecorView的父布局里，但是我还是要再说一遍。且看下图



这个图可能和伙伴们在书上或者网上常见的不太一样，为什么不太一样呢？因为是我自己画的，哈哈哈...下面就来看着图捋一捋Android最基本的视图框架。

### PhoneWindow

估计很多同学都知道，每一个Activity都拥有一个Window对象的实例。这个实例实际是PhoneWindow类型的。那么PhoneWindow从名字很容易看出，它应该是Window的子类

知识点：每一个Activity都有一个PhoneWindow对象。

那么，PhoneWindow有什么用呢？它在Activity充当什么角色呢？下面我就姑且把PhoneWindow等同于Window来称呼吧。**Window**从字面看它是一个窗口，看图说。可以看到，我们要显示的布局是被放到它的属性**mDecor**中的，这个mDecor就是**DecorView**的一个实例。下面会专门撸DecorView，现在先把关注点放到Window上。Window还有一个比较重要的属性**mWindowManager**，它是**WindowManager**(这是个接口)的一个实现类的一个实例。我们平时通过**getWindowManager**()方法获得的东西就是这个mWindowManager。顾名思义，它是Window的管理者，负责管理着窗口及其中显示的内容。它的实际实现类是**WindowManagerImpl**。mWindowManager是在它爹那里就实例化好的。下面代码是在Window.java中的。

public void setWindowManager(WindowManager wm, IBinder appToken,  
 String appName, boolean hardwareAccelerated) {  
 ...  
 if (wm == null) {

//获取了一个WindowManager  
 wm = (WindowManager)mContext.getSystemService(Context.WINDOW\_SERVICE);  
 }  
 //通过这里我们可以知道，上面获取到的wm实际是WindowManagerImpl类型的。  
 mWindowManager = ((WindowManagerImpl)wm).createLocalWindowManager(this);  
}

通过上面的介绍，我们已经知道了Window中有负责承载布局的DecorView，有负责管理的WindowManager（事实上它只是个代理，后面会讲它代理的是谁）。

### DecorView

前面提到过，在Activity的onCreate()中通过setContentView()设置的布局实际是被放到DecorView中的。我们在图中找到DecorView。

从图中可以看到，DecorView继承了FrameLayout，并且一般情况下，它会在先添加一个预设的布局。比如DecorCaptionView，它是从上到下放置自己的子布局的，相当于一个LinearLayout。通常它会有一个标题栏，然后有一个容纳内容的mContentRoot，这个布局的类型视情况而定。我们希望显示的布局就是放到了mContentRoot中。

知识点：通过setContentView()设置的布局是被放到DecorView中，DecorView是视图树的最顶层。

### WindowManager

前面已经提到过，WindowManager在Window中具有很重要的作用。我们先在图中找到它。这里需要先说明一点，在PhoneWindow中的mWindowManager实际是WindowManagerImpl类型的。WindowManagerImpl自然就是接口WindowManager的一个实现类喽。这一点是我没有在图中反映的。

WindowManager是在Activity执行attach()时被创建的，attach()方法是在onCreate()之前被调用的

final void attach(Context context, ActivityThread aThread,  
 Instrumentation instr, IBinder token, int ident,  
 Application application, Intent intent, ActivityInfo info,  
 CharSequence title, Activity parent, String id,  
 NonConfigurationInstances lastNonConfigurationInstances,  
 Configuration config, String referrer, IVoiceInteractor voiceInteractor,  
 Window window){  
 ...  
 mWindow = new PhoneWindow(this, window);  
 //创建Window  
 ...  
 mWindow.setWindowManager(  
 (WindowManager)context.getSystemService(Context.*WINDOW\_SERVICE*),  
 mToken, mComponent.flattenToString(),  
 (info.flags & ActivityInfo.*FLAG\_HARDWARE\_ACCELERATED*) != 0);  
 //注意！这里就是在创建WindowManager。  
 //这个方法在前面已经说过了。  
 if (mParent != null) {  
 mWindow.setContainer(mParent.getWindow());  
 }  
 mWindowManager = mWindow.getWindowManager();  
}

继续看图。WindowManagerImpl持有了PhoneWindow的引用，因此它可以对PhoneWindow进行管理。同时它还持有一个非常重要的引用mGlobal。这个mGlobal指向一个WindowManagerGlobal类型的单例对象，这个单例每个应用程序只有唯一的一个。在图中，我说明了WindowManagerGlobal维护了本应用程序内所有Window的DecorView，以及与每一个DecorView对应关联的ViewRootImpl。这也就是为什么我前面提到过，WindowManager只是一个代理，实际的管理功能是通过WindowManagerGlobal实现的。我们来看个源码的例子就比较清晰了。开始啦！

WimdowManagerImpl.java

public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params) {  
...  
 mGlobal.addView(view, params, mContext.getDisplay(), mParentWindow);  
 //实际是通过WindowManagerGlobal实现的。  
}

从上面的代码可以看出，WindowManagerImpl确实只是WindowManagerGlobal的一个代理而已。同时，上面这个方法在整个Android的视图框架流程中十分的重要。我们知道，在Activity执行onResume()后界面就要开始渲染了。原因是在onResume()时，会调用WindowManager的addView()方法(实际最后调用的是WindowManagerGlobal的addView()方法)，把视图添加到窗口上。

ActivityThread.java

final void handleResumeActivity(IBinder token, boolean clearHide, boolean isForward,

boolean reallyResume, int seq, String reason) {  
...  
 ViewManager wm = a.getWindowManager();  
 //获得WindowManager，实际是WindowManagerImpl  
...  
 wm.addView(decor, l);  
 //添加视图  
...  
 wm.updateViewLayout(decor, l);  
 //需要刷新的时候会走这里  
...  
}

从上面可以看到，当Activity执行onResume()的时候就会添加视图，或者刷新视图。需要解释一点：**WindowManager实现了ViewManager接口**。

如图中所说，WindowManagerGlobal调用addView()的时候会把DecorView添加到它维护的数组中去，并且会创建另一个关键且极其重要的ViewRootImpl(这个必须要专门讲一下)类型的对象，并且也会把它存到一个数组中维护。

WindowManagerGlobal.java

public void addView(View view,ViewGroup.LayoutParams params,Display display,Window parentWindow) {  
...  
 root = new ViewRootImpl(view.getContext(), display);  
 //重要角色登场  
 view.setLayoutParams(wparams);  
 mViews.add(view);  
 mRoots.add(root);  
 //保存起来维护  
 mParams.add(wparams);  
...  
 root.setView(view, wparams, panelParentView);  
 //设置必要属性view是DecorView，panelParentView是PhoneWindow  
...  
}

可以看出ViewRootImpl是在Activity执行onResume()的时候才被创建的，并且此时才把DecorView传进去让它管理。

知识点：WindowManager是在onCreate()时被创建。它对窗口的管理能力实际是通过WindowManagerGlobal实现的。在onResume()是视图才通过WindowManager被添加到窗口上。

### ViewRootImpl

ViewRootImpl能够和系统的WindowManagerService进行交互，并且管理着DecorView的绘制和窗口状态。非常的重要。赶紧在图中找到对应位置吧！

ViewRootImpl并不是一个View，而是负责管理视图的。它配合系统来完成对一个Window内的视图树的管理。从图中也可以看到，它持有了DecorView的引用，并且视图树它是视图树绘制的起点。因此，ViewRootImpl会稍微复杂一点，需要我们更深入的去了解，在图中我标出了它比较重要的组成Surface和Choreographer等都会在后面提到。

到此，我们已经一起把第一张图撸了一遍了，现在童鞋们因该对Android视图框架有了大致的了解。下面将更进一步的去了解Android的绘制机制。

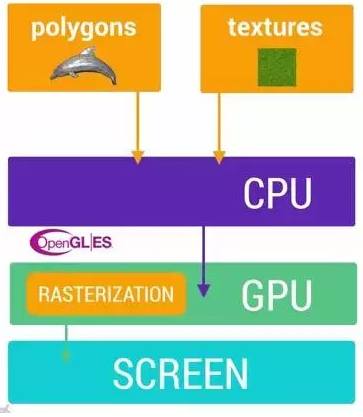
## App总是卡顿到底是什么原因？

下面将会详细的讲解为什么我们设置的视图能够被绘制到屏幕上？这中间究竟隐藏着怎样的离奇？看完之后，你自然就能够从根源知道为什么你的App会那么卡，以及开始有思路着手解决这些卡顿。

图片地址：<https://upload-images.jianshu.io/upload_images/1869462-c71f7527c48dac86.png>

### CPU、GPU是搞什么鬼的？

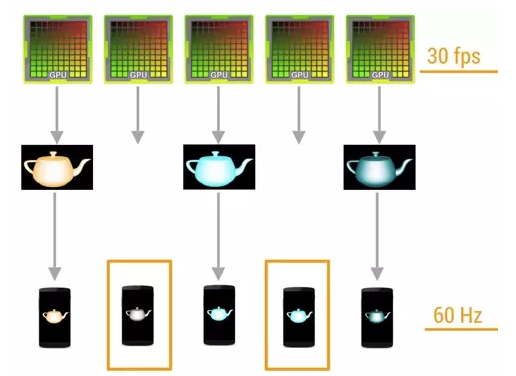
* CPU主要负责了视图的测量、布局、记录、把内容计算成Polygons多边形或者Texture纹理
* 而GPU主要负责把Polygons或者Textture进行Rasterization栅格化，这样才能在屏幕上成像。
* 在使用硬件加速后，GPU会分担CPU的计算任务，而CPU会专注处理逻辑，这样减轻CPU的负担，使得系统效率更高



### RefreshRate刷新率和FrameRate帧率

RefreshRate刷新率是屏幕每秒刷新的次数，是一个与硬件有关的固定值。在Android平台上，这个值一般为60HZ，即屏幕每秒刷新60次。

FrameRate帧率是每秒绘制的帧数。通常只要帧数和刷新率保持一致，就能够看到流畅的画面。在Android平台，我们应该尽量维持60FPS的帧率。但有时候由于视图的复杂，它们可能就会出现不一致的情况。



如图，当帧率小于刷新率时，比如图中的30FPS < 60HZ，就会出现相邻两帧看到的是同一个画面，这就造成了卡顿。这就是为什么我们总会说，要尽量保证一帧画面能够在16ms(1000/60=16.67ms)内绘制完成，就是为了和屏幕的刷新率保持同步。

下面将会介绍Android是如何来确保刷新率和帧率保持同步的。

### Vsync(垂直同步)是什么？

你可能在游戏的设置中见过Vsync，开启它通常能够提高游戏性能。在Android中，同样使用Vsync垂直同步来提高显示性能。它能够使帧率FrameRate和硬件的RefreshRate刷新强制保持一致。

#### HWComposer与Vsync不得不说的事

看图啦看图啦。首先在最左边我们看到有个叫HWComposer的类，这是一个c++编写的类。它Android系统初始化时就被创建，然后开始配合硬件产生Vsync信号，也就是图中的HW\_Vsync信号。当然它不是一直不停的在产生，这样会导致Vsync信号的接收者不停的接收到绘制、渲染命令，即使它们并不需要，这样会带来严重的性能损耗，因为进行了很多无用的绘制。所以它被设计设计成能够唤醒和睡眠的。这使得HWComposer在需要时才产生Vsync信号(比如当屏幕上的内容需要改变时)，不需要时进入睡眠状态(比如当屏幕上的内容保持不变时，此时屏幕每次刷新都是显示缓冲区里没发生变化的内容)。

如图，Vsync的两个接收者，一个是SurfaceFlinger(负责合成各个Surface)，一个是Choreographer(负责控制视图的绘制)。我们稍后再介绍，现在先知道它们是干什么的就行了。

#### Vsync offset机制

为了提高效率，尽量减少卡顿，在Android 4.1时引入了Vsync机制，并在随后的4.4版本中加入Vsync offset偏移机制。

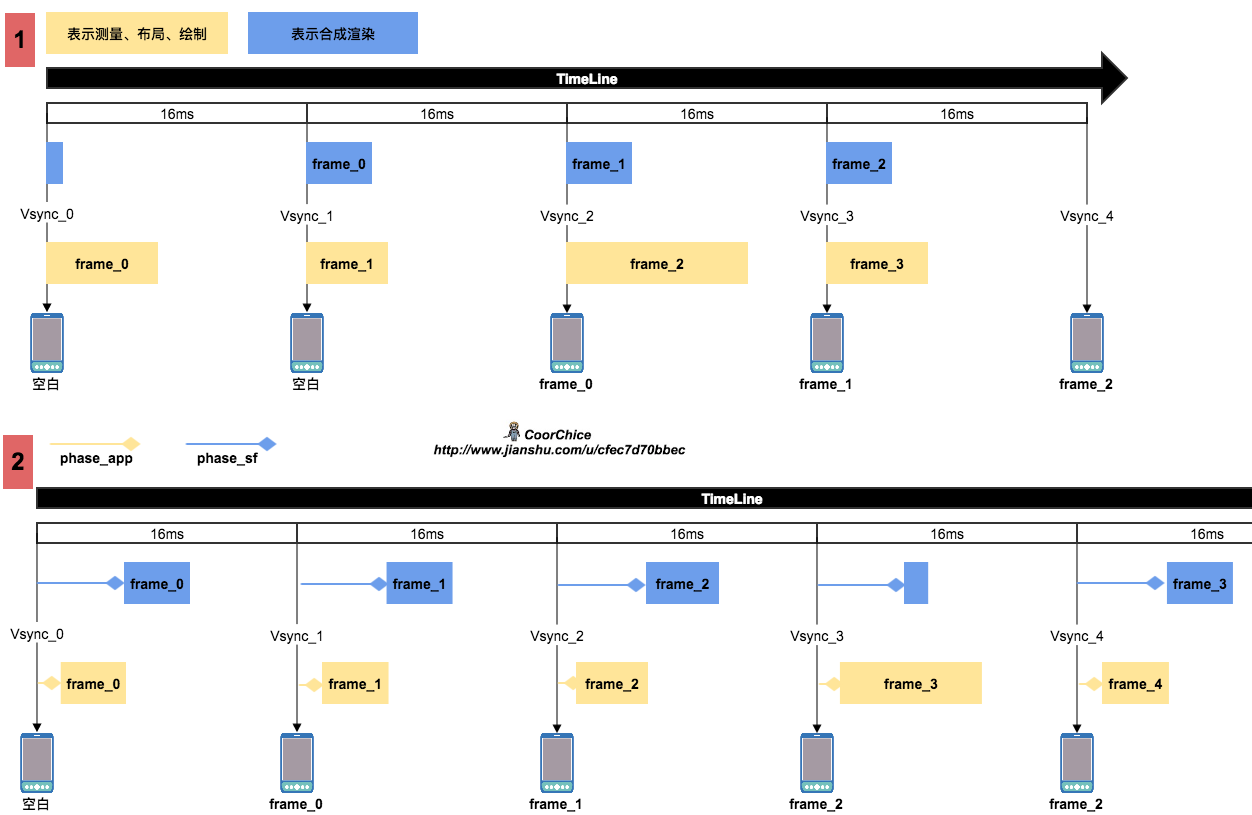


图1为4.1时期的Vsync机制。可以看到，当一个Vsync信号到来时，SurfaceFlinger和UI绘制进程会同时启动，导致它们竞争CPU资源，而CPU分配资源会耗费时间，着降低系统性能。同时当收到一个Vsync信号时，第N帧开始绘制。等再收到一个Vsync信号时，第N帧才被SurfaceFlinger合成。而需要显示到屏幕上，需要等都第三个Vsync信号。这是比较低效率。于是才有了图4.4版本加入的Vsync offset机制。

图2为Google加入Vsync offset机制后，原本的HW\_Vsync信号会经过DispSync会分成Vsync和SF\_Vsync两个虚拟化的Vsync信号。其中Vsync信号会发送到Choreographer中，而SF\_Vsync会发送到SurfaceFlinger中。理论上只要phase\_app和phase\_sf这两个偏移参数设置合理，在绘制阶段消耗的时间控制好，那么画面就会像图2中的前几帧那样有序流畅的进行。理想总是美好的。实际上很难一直维持这种有序和流畅，比如frame\_3是比较复杂的一帧，它的绘制完成的时间超过了SurfaceFlinger开始合成的时间，所以它必须要等到下一个Vsync信号到来时才能被合成。这样便造成了一帧的丢失。但即使是这样，如你所见，加入了Vsync offset机制后，绘制效率还是提高了很多。

从图中可以看到，Vsync和SF\_Vsync的偏移量分别由phase\_app和phase\_sf控制，这两个值是可以调节的，默认为0，可为负值。你只需要找到BoardConfig.mk文件，就可以对这两个值进行调节。

### 回到ViewRootImpl

前面介绍了几个关键的概念，现在我们回到ViewRootImpl中去，在图中找到ViewRootImpl的对应位置。前面说过，ViewRootImpl控制着一个Window中的整个视图树的绘制。那它是如何进行控制的呢？一次绘制究竟是如何开始的呢？

在ViewRootImpl创建的时候，会获取到前面提到过过的一个关键对象Choreographer。Choreographer在一个线程中仅存在一个实例，因此在UI线程只有一个Choreographer存在。也就说，通常情况下，它相当于一个应用中的单例。

在ViewRootImpl初始化时，会实现一个Choreographer.FrameCallback（这是一个Choreographer中的内部类），并向Choreographer中post。顾名思义，FrameCallback会在每次接收到Vsync信号时被回调。

Choreographer.java

public interface FrameCallback {  
 public void doFrame(long frameTimeNanos);  
 //一旦注册到CallbackQueue中，那么  
 //每次Choreographer接收到Vsync信号时都会回调。  
}

FrameCallback一旦被注册，那么每次收到Vsync信号时它都会被回调。利用它，我们可以实现会帧率的监听。

ViewRootImpl.java

//这个方法只有在ViewRootImpl初始化时才会被调用  
private void profileRendering(boolean enabled) {  
...  
 mRenderProfiler = new Choreographer.FrameCallback() {  
 @Override  
 public void doFrame(long frameTimeNanos) {  
 ...  
 scheduleTraversals();  
 //请求一个Vsync信号，后面还会提到这个方法  
 mChoreographer.postFrameCallback(mRenderProfiler);  
 //每次回调时，重新将FrameCallback post到Choreographer中  
 ...  
 }  
 };  
...  
 mChoreographer.postFrameCallback(mRenderProfiler);  
 //将FrameCallback post到Choreographer中  
...  
}

上面代码出现了一个重要方法scheduleTraversals()。下面我们看看它究竟为何重要。

ViewRootImpl.java

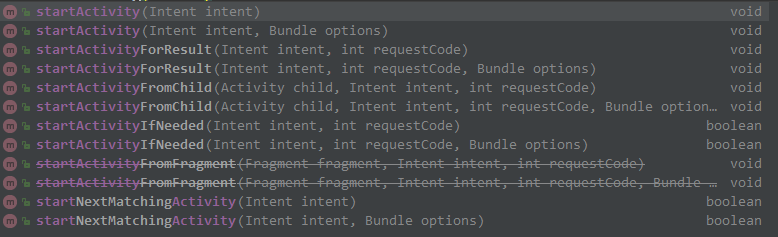
void scheduleTraversals() {  
...  
 mChoreographer.postCallback(  
 Choreographer.CALLBACK\_TRAVERSAL, mTraversalRunnable, null);  
 //向Choreographer中post一个TraversalRunnable  
 //这又是一个十分重要的对象  
...  
}

# Activity跳转防重

解决方案有至少以下三种：

1. AOP
2. 启动模式
3. 基类中startActivityForResult路由拦截

分析一下第三种，因为一般这种路由跳转都会写个通用的方法来处理，这样后续想添加什么功能都方便，登录拦截等等，我们通常跳转Activity用的是startActivity和startActivityForResult，查看源码可以看出



@Override  
public void startActivity(Intent intent) {  
 this.startActivity(intent, null);  
}

@Override  
public void startActivity(Intent intent, @Nullable Bundle options) {  
 if (options != null) {  
 startActivityForResult(intent, -1, options);  
 } else {  
  
 startActivityForResult(intent, -1);  
 }  
}

public void startActivityForResult(@RequiresPermission Intent intent, int requestCode) {  
 startActivityForResult(intent, requestCode, null);  
}

public void startActivityForResult(@RequiresPermission Intent intent, int requestCode, @Nullable Bundle options) {  
 if (mParent == null) {  
 options = transferSpringboardActivityOptions(options);  
 Instrumentation.ActivityResult ar =  
 mInstrumentation.execStartActivity(  
 this, mMainThread.getApplicationThread(), mToken, this,  
 intent, requestCode, options);  
 if (ar != null) {  
 mMainThread.sendActivityResult(  
 mToken, mEmbeddedID, requestCode, ar.getResultCode(),  
 ar.getResultData());  
 }  
 if (requestCode >= 0) {  
 mStartedActivity = true;  
 }  
 cancelInputsAndStartExitTransition(options);  
 } else {  
 if (options != null) {  
 mParent.startActivityFromChild(this, intent, requestCode, options);  
 } else {  
 mParent.startActivityFromChild(this, intent, requestCode);  
 }  
 }  
}

最终都会调用到startActivityForResult(Intent intent, int requestCode, @Nullable Bundle options),所以我们在基类中复写该方法即可，首先需要一个双击判断工具类：

public final class DoubleClickHelper {  
 private static final long[] *TIME\_ARRAY* = new long[2]; // 数组的长度为2代表只记录双击操作  
 //是否在短时间内进行了双击操作  
 public static boolean isOnDoubleClick() {  
 // 默认间隔时长  
 return *isOnDoubleClick*(1200);  
 }  
 //是否在短时间内进行了双击操作  
 public static boolean isOnDoubleClick(int time) {  
 if (false) {  
 //将数组整体往前移动一位，最后一位保持不变，即[1,3,1,4,5]变为[3,1,4,5,5]  
 System.*arraycopy*(*TIME\_ARRAY*, 1, *TIME\_ARRAY*, 0, *TIME\_ARRAY*.length - 1);  
 } else {  
 //上面的代码可以简写为下面的  
 *TIME\_ARRAY*[0] = *TIME\_ARRAY*[1];  
 }  
 //如果两次点击事件大于间隔，才保存最新一次的点击的时间  
 if (*TIME\_ARRAY*[1] - *TIME\_ARRAY*[0] <= time) {  
 return true;  
 } else {  
 //如果数组最后一位赋值为当前时间  
 *TIME\_ARRAY*[1] = SystemClock.*uptimeMillis*();  
 return false;  
 }  
 }  
}

重写基类中的startActivityForResult

public abstract class BaseActivity extends AppCompatActivity {  
 @Override  
 public void startActivityForResult(Intent intent, int requestCode, @Nullable Bundle options) {  
 if (DoubleClickHelper.*isOnDoubleClick*(500)) return;  
 super.startActivityForResult(intent, requestCode, options);  
 }  
}

但这样写存在两个问题：

1. 如果这个界面需要多重跳转，直接写死 BaseActivity 不利于扩展
2. 如果需要同时跳转两个不同的 Activity，结果只有第一个成功跳转，而第二个却没有跳转

想知道重复跳转的 Activity 是不是同一个，我们可以通过 Intent 这个对象来进行判断

显式意图启动

构造方法：new Intent(Context packageContext, Class<?> cls)

对象方法：intent.setClass(Context packageContext, Class<?> cls)

隐式意图启动

构造方法：new Intent(String action)

对象方法：intent.setAction(String action)

public abstract class BaseActivity extends AppCompatActivity {  
 private String mActivityJumpTag;  
 private long mActivityJumpTime;  
 private final static long *DOUBULE\_TIME\_LONG* = 800;  
 @Override  
 public void startActivityForResult(Intent intent, int requestCode, @Nullable Bundle options) {  
 if (DoubleClickHelper.*isOnDoubleClick*(500)) return;  
 super.startActivityForResult(intent, requestCode, options);  
 }  
 *//检查当前 Activity 是否重复跳转了，不需要检查则重写此方法并返回 true 即可* private boolean startActivitySelfCheck(Intent intent) {  
 String tag;// 标记对象,不判断同一个class同时显式、隐式调用的情况  
 if (intent.getComponent() != null) { // 显式跳转  
 tag = intent.getComponent().getClassName();  
 } else if (intent.getAction() != null) { // 隐式跳转  
 tag = intent.getAction();  
 } else {  
 return true;  
 }  
 if (tag.equals(mActivityJumpTag) && SystemClock.*uptimeMillis*() - mActivityJumpTime <= *DOUBULE\_TIME\_LONG*) {  
 return false;// 检查不通过  
 }  
 mActivityJumpTag = tag;//检查通过才保存最近一次点击跳转的class  
 mActivityJumpTime = SystemClock.*uptimeMillis*();//检查通过才保存最近一次点击时间  
 return true;  
 }  
}

# 序列化Serializable、Parcelabel

类去实现Serializable或Parcelabel接口

## 为什么要序列化？

1）永久性保存对象，保存对象的字节序列到本地文件中；

2）通过序列化对象在网络中传递对象；

3）通过序列化在进程间传递对象。

## 实现序列化的方法

Android中实现序列化有两个选择：一是实现Serializable接口（是JavaSE本身就支持的），一是实现Parcelabel接口（是Android特有功能，效率比实现Serializable接口高效，可用于Intent数据传递，也可以用于进程间通信（IPC））。实现Serializable接口非常简单，声明一下就可以了，而实现Parcelable接口稍微复杂一些，但效率更高，推荐用这种方法提高性能。

注：Android中Intent传递对象有两种方法：一是Bundle.putSerializable(Key，Object)，另一种是Bundle.putParcelable(Key，Object)。当然这些Object是有一定的条件的，前者是实现了Serializable接口，而后者是实现了Parcelable接口。

**public** **class** **Student** **implements** Parcelable{

**private** **String** name;

**private** **int** age;

**private** **String** subjcet;

private boolean isSelect;?//boolean数据有点特殊，因为只有parcel.writeBooleanArray();

public boolean isSelect() {

return isSelect;

}

public void setSelect(boolean select) {

isSelect = select;

}

//...............................

//从Parcel中获取变量的内容

**protected** **Student**(**Parcel** in) {

**this**.name = in.readString();

**this**.age = in.readInt();

**this**.subjcet = in.readString();

this.isSelect = in.readByte() != 0; //myBoolean == true if byte != 0

}

**public** **static** **final** Creator<**Student**> **CREATOR** = **new** Creator<**Student**>() {

//反序列化方法，将writeToParcel方法写入的Parcel反序列成ParcelDemo

@Override

**public** **Student** **createFromParcel**(**Parcel** in) {

**return** **new** Student(in);

}

//这个方法是提供给其它类反序列化该类的数组时调用

@Override

**public** **Student**[] **newArray**(**int** size) {

**return** **new** **Student**[size];

}

};

//接口内容描述 ，通常返回0即可

@Override

**public** **int** **describeContents**() {

**return** 0;

}

//实现序列化操作的方法，将类数据写到Parcel窗口中

@Override

**public** **void** **writeToParcel**(**Parcel** parcel, **int** i) {

parcel.writeString(name);

parcel.writeInt(age);

parcel.writeString(subjcet);

parcel.writeByte((byte) (isSelect ? 1 : 0));//if myBoolean == true, byte == 1

}

}

## Serializable实现与Parcelabel实现的区别

1）Serializable的实现，只需要implements  Serializable 即可。这只是给对象打了一个标记，系统会自动将其序列化。

2）Parcelabel的实现，不仅需要implements  Parcelabel，还需要在类中添加一个静态成员变量CREATOR，这个变量需要实现 Parcelable.Creator 接口。

**实现接口：**

**public** **class** **demoClass** **implements** Serializable { //不能是内部类

**存数据：**

**demoClass** dClass = **new** demoClass();

dClass.setAge(20);

dClass.setName("服服服");

intent.putExtra("demoClass", dClass);

**取数据：**

**demoClass** dClass = (**demoClass**) getIntent().getSerializableExtra("demoClass");

**String** name = dClass.getName();

**int** age = dClass.getAge();

# Activity的四种启动模式和一些标签

## 四种启动模式

Standard

默认的启动模式

singleTop

如果在任务的栈顶正好存在该Activity的实例， 就重用该实例，否者就会创建新的实例并放入栈顶(即使栈中已经存在该Activity实例，只要不在栈顶，都会创建实例)。

singleTask

如果在栈中已经有该Activity的实例，就重用该实例(会调用实例的onNewIntent())。重用时，会让该实例回到栈顶，因此在它上面的实例将会被移除栈。如果栈中不存在该实例，将会创建新的实例放入栈中

singleInstance

在一个新栈中创建该Activity实例，并让多个应用共享改栈中的该Activity实例。一旦改模式的Activity的实例存在于某个栈中，任何应用再激活改Activity时都会重用该栈中的实例，其效果相当于多个应用程序共享一个应用，不管谁激活该Activity都会进入同一个应用中

## Activity[其他的标记的一些解释](http://www.cnblogs.com/xiaoQLu/archive/2012/07/17/2595294.html)

taskAffinity属性:

* 如果没有给activity设置taskAffinity，默认都会读取application标签下的taskAffinity属性值，如果application标签下也没有设置taskAffinity，那taskAffinity默认值就是manifest标签下设置的包名。
* 不仅可以给同一个应用的不同activity设置不同的affinity，也可以给不同应用的activity设置相同的affinity，使它们在用户角度看来好像属于同一个应用

(1) taskAffinity属性应和FLAG\_ACTIVITY\_NEW\_TASK标志及allowTaskReparenting属性结合使用, 如果只使用taskAffinity属性,请参考上面Task默认的行为.

(2) 与FLAG\_ACTIVITY\_NEW\_TASK标志结合:

a. 前提: Activity A和Activity B在同一个应用中, Activity A与Activity B设置不同的taskAffinity属性.

操作: Activity A启动开僻Task堆栈,命名为TaskA(TaskA堆栈状态: A), 在Activity A中启动Activity B, 启动Activity B的Intent中设置FLAG\_ACTIVITY\_NEW\_TASK标志,这时系统会新开僻一个Task堆栈,TaskB(TaskB堆栈状态: B).

b. 前题: Activity A在"TaskOne应用"中, Activity C在"TaskTwo应用"中, Activity A和ActivityC设置了相同的taskAffinity属性.

操作: Activity A启动开僻Task堆栈,命名为TaskA(TaskA堆栈状态: A), 在Activity A中启动Activity C, 启动Activity C的Intent中设置FLAG\_ACTIVITY\_NEW\_TASK标志,这时Activity C会压入与Activity A堆栈相同的TaskA堆栈(TaskA堆栈状态: AC).

(3) 与allowTaskReparenting属性:

例: 在"TaskOne应用"中有一个天气预报Activity A, Activity A与"TaskOne应用"中的其它Activity有默认的关系(taskAffinity属性都没有设置), 并且allowTaskReparenting属性设置为true, 现在存在一个"TaskTwo应用"启动了"TaskOne应用"中的天气预报Activity A,  这时Activity A与"TaskTwo应用"中的Activity在同一个Task,命名这个Task堆栈为TaskA, 这时"TaskOne应用"启动, 并且又打开发天气预报Activity A, 这时Activity A会从TaskA堆栈中转移到"TaskOne应用"所在的堆栈, 即Activity A可以在多个堆栈中来回转移.         
alwaysRetainTaskState属性:

如果Task堆栈中的Root Activity设置了此属性值为true, 不管出现任何情况, 一直会保留Task栈中Activity的状态.

clearTaskOnLaunch属性:

如果Task堆栈中的Root Activity设置了此属性值为true, 只要你一离开这个Task栈, 则系统会马上清理除了Root Activity的全部Activity.

finishOnTaskLaunch属性:

如果某Activity设置了finishOnTaskLaunch属性, 只要你一离开这个Task栈, 则系统会马上清除这个Activity,不管这个Activity在堆栈的任何位置.

# getIntent()、getStringExtra---Activity在oncrat（）中

**Intent intent = getIntent();**

一定会有intent，intent不会等于null

**Bundle bundle = intent.getExtras();**

如果启动Activity没有在intent中put值，bundle = null, 在使用bundle时会报空指针异常

**int i = bundle.getInt("i", 22);**

如果bundle中没有该key，i = 22

**int i2 = bundle.getInt("i2");**

如果bundle中没有该key，i2 = 0；

**String s = bundle.getString("s");**

如果bundle中没有该key，s= null

**int i3 = intent.getIntExtra("i", 22);**

如果intent中没有该key，i3 = 22

**int i4 = intent.getIntExtra("i2");**

没有这个方法，必须要有默认值

**String s2 = intent.getStringExtra("s");**

如果intent中没有该key，s2 = null

# Intent

## Intent的Component属性

Intent对象的setComponent(ComponentName comp)方法用于设置Intent的Component属性. ComponentName包含如下几个构造器:

* ComponentName(String pkg, String cls)
* ComponentName(Context pkg, String cls)
* ComponentName(Context pkg, Class<?> cls)

由以上的构造器可知, 创建一个ComponentName对象需要指定包名和类名--这就可以唯一确定一个组件类, 这样应用程序即可根据给定的组件类去启动特定的组件

## Intent的Action属性

action属性是一个字符串, 代表某一种特定的动作. Intent类预定义了一些action常量, 开发者也可以自定义action. 一般来说, 自定义的action应该以application的包名作为前缀, 然后附加特定的大写字符串, 例如"cn.xing.upload.action.UPLOAD\_COMPLETE"就是一个命名良好的action.

Intent类的setAction()方法用于设定action, getAction()方法可以获取Intent中封装的action.

以下是Intent类中预定义的部分action:

* ACTION\_CALL--目标组件为activity, 代表拨号动作;
* ACTION\_EDIT--目标组件为activity, 代表向用户显示数据以供其编辑的动作;
* ACTION\_MAIN--目标组件为activity, 表示作为task中的初始activity启动;
* ACTION\_BATTERY\_LOW--目标组件为broadcastReceiver, 提醒手机电量过低;
* ACTION\_SCREEN\_ON--目标组件为broadcast, 表示开启屏幕.

## Intent的Category属性

**资料一：**

Category：于指定当前动作（Action）被执行的环境

即这个activity在哪个环境中才能被激活。不属于这个环境的，不能被激活。

常用的Category属性如下所示：

* CATEGORY\_DEFAULT：Android系统中默认的执行方式，按照普通Activity的执行方式执行。表示所有intent都可以激活它
* CATEGORY\_HOME：设置该组件为Home Activity。
* CATEGORY\_PREFERENCE：设置该组件为Preference。
* CATEGORY\_LAUNCHER：设置该组件为在当前应用程序启动器中优先级最高的Activity，通常为入口ACTION\_MAIN配合使用。
* CATEGORY\_BROWSABLE：设置该组件可以使用浏览器启动。表示该activity只能用来浏览网页。
* CATEGORY\_GADGET：设置该组件可以内嵌到另外的Activity中。

注意：

如果该activity想要通过隐式intent方式激活，那么不能没有任何category设置，至少包含一个android.intent.category.DEFAULT

**资料二：**

category属性也是一个字符串, 用于指定一些目标组件需要满足的额外条件. Intent对象中可以包含任意多个category属性. Intent类也预定义了一些category常量, 开发者也可以自定义category属性.

Intent类的addCategory()方法为Intent添加Category属性, getCategories()方法用于获取Intent中封装的所有category.

以下是Intent类中预定义的部分category:

* CATEGORY\_HOME--表示目标activity必须是一个显示home screen的activity;
* CATEGORY\_LAUNCHER--表示目标activity可以作为task栈中的初始activity, 常与ACTION\_MAIN配合使用;
* CATEGORY\_GADGET--表示目标activity可以被作为另一个activity的一部分嵌入.

## Intent的Data和Type属性

data属性指定所操作数据的URI. data经常与action配合使用, 如果action为ACTION\_EDIT, data的值应该指明被编辑文档的URI; 如果action为ACTION\_CALL, data的值应该是一个以"tel:"开头并在其后附加号码的URI; 如果action为ACTION\_VIEW, data的值应该是一个以"http: "开头并在其后附加网址的URI...

Intent类的setData()方法用于设置data属性, setType()方法用于设置data的MIME类型, setDataAndType()方法可以同时设定两者. 可以通过getData()方法获取data属性的值, 通过getType()方法获取data的MIME类型.

## Intent的Extra属性

通过Intent启动一个component时, 经常需要携带一些额外的数据过去. 携带数据需要调用Intent的putExtra()方法, 该方法存在多个重载方法, 可用于携带基本数据类型及其数组, String类型及其数组, Serializable类型及其数组, Parcelable类型及其数组, Bundle类型等. Serializable和Parcelable类型代表一个可序列化的对象, Bundle与Map类似,可用于存储键值对.

## IntentFilter与隐式Intent

android系统处理隐式Intent时, 会比较Intent和IntentFilter的action, data, category属性, 如果以上3个属性全都相符的话, 则IntentFilter所属的component就可以作为目标组件的候选(存在多个符合条件的component时).

### 测试action属性

intent最多只能定义1个action, 而filter可以定义1个或多个action. 通过action测试的条件为: filter定义了intent的action. 例如intent的action为"android.intent.action.MAIN", 则定义了"android.intent.action.MAIN"这个action的filter都能通过action测试(当然, filter还可以包含更多额外的action).如果filter没有定义action, 则这个filter将阻塞所有intent. 如果intent没有定义action, 那么只要filter定义了action就可以通过action测试.

### 测试category属性

intent可以任意多个category, filter也可以任意个category. 通过category测试的条件为: filter定义了intent的所有category. 例如intent定义了"android.intent.category.DEFAULT"和"cn.xing.intent.category.UPLOAD"这2个category, 则定义了以上2个category属性的filter才能通过测试(当然, filter还可以包含更多额外的category).

根据上面的规则, 如果一个intent没有定义category, 则所有filter都可以通过category测试. 但是有一种例外: 以startActivity(intent)方式启动一个activity时, 系统为会intent增加一个值为"android.intent.category.DEFAULT"的category, 这就意味着每一个期望通过category测试的activity, 都要在其filter中定义"android.intent.category.DEFAULT"(除了作为程序入口的activity).

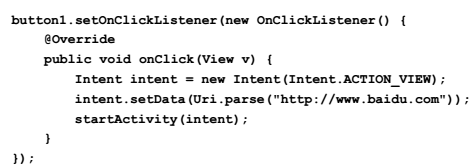
### 测试data属性

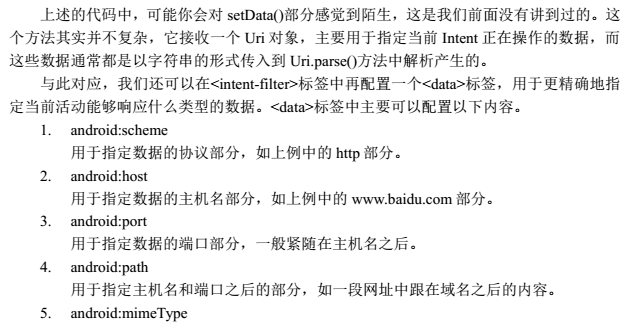
intent最多只能定义1个data, filter则可以定义多个data.，通过data测试的条件为，一模一样，不多不少

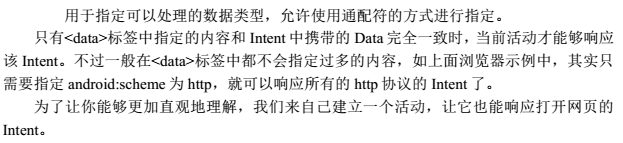
data属性是一个URI, URI中包含scheme, host, post和path, 典型的URI为:scheme://host:port/path

scheme, host, post和path都是可选的. 比较2个data时, 只比较filter中包含的部分. 比如filter的一个data只是指定了scheme部分, 则测试时只是比较data的scheme部分, 只要两者的scheme部分相同, 就视为"相同的data".

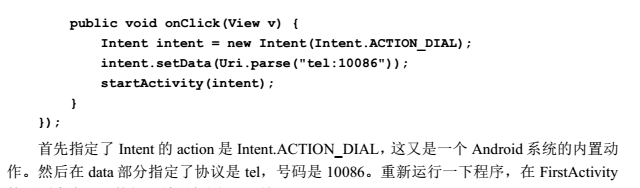
## Intent的data属性，以及intent-filter







**打电话**



## Intent跳转Activity

intent.setClass(this, Activity01.class);

intent2.setClassName(this, "com.zxw51.explicit.Activity01");

intent3.setClassName("com.example.bu\_togglebutton","activity.MainActivity");//这个包名是程序的包名，不是类的包名,实现打开其他应用Activity，如果是主Activity，则不用配置将要打开的Activity的exported（共享权限）为true，若不是，则必须指定为true

intent4.setComponent(new ComponentName(this,Activity01.class));//component——组件

Intent intent5=new Intent("com.zxw666.activity01");

intent5.addCategory(Intent.CATEGORY\_DEFAULT);

## action和type，data可以实现不同的功能

1.打开图片intent.setAction(**Intent**.**ACTION\_GET\_CONTENT**);

intent.setType("image/\*");

2.分享文字intent.setAction(**Intent**.**ACTION\_SEND**);

intent.setType("text/plain");

intent.putExtra(**Intent**.**EXTRA\_TEXT**, "I'm a dingding girl!");

3.打开网页intent.setAction(**Intent**.**ACTION\_VIEW**);

**Uri** url = **Uri**.parse("http://www.baidu.com");

intent.setData(url);

4.拨打电话intent.setAction(**Intent**.**XXXX**);

**Uri** url = **Uri**.parse("tel:15102391937");

intent.setData(url);

**ACTION\_DIAL,进入系统拨号键盘**

**ACTION\_CALL,系统直接拨号播出**

**ACTION\_VIEW,徐泽能拨电话的软件**

## Intent FLAG介绍:

flag属性是一个int值, 用于通知android系统如何启动目标activity, 或者启动目标activity之后应该采取怎样的后续操作. 所有的flag都在Intent类中定义, 部分常用flag如下:

* FLAG\_ACTIVITY\_NEW\_TASK--通知系统将目标activity作为一个新task的初始activity;
* FLAG\_ACTIVITY\_NO\_HISTORY--通知系统不要将目标activity放入历史栈中;
* FLAG\_FROM\_BACKGROUND--通知系统这个Intent来源于后台操作, 而非用户的直接选择...

Intent的setFlags和addFlags

setFlags是把之前的替换掉，addFlags是添加新的,一般使用setFlags就够了

(1) FLAG\_ACTIVITY\_NEW\_TASK:

设置此状态，记住以下原则，首先会查找是否存在和被启动的Activity具有相同的亲和性的任务栈（即taskAffinity，注意同一个应用程序中的activity的亲和性一样，所以下面的a情况会在同一个栈中，前面这句话有点拗口，请多读几遍），如果有，刚直接把这个栈整体移动到前台，并保持栈中的状态不变，即栈中的activity顺序不变，如果没有，则新建一个栈来存放被启动的activity

a. 前提: Activity A和Activity B在同一个应用中.

操作: Activity A启动开僻Task堆栈(堆栈状态: A), 在Activity A中启动Activity B, 启动Activity B的Intent的Flag设为FLAG\_ACTIVITY\_NEW\_TASK, Activity B被压入Activity A所在堆栈(堆栈状态: AB).

原因: 默认情况下同一个应用中的所有Activity拥有相同的关系(taskAffinity).

b. 前提: Activity A在名称为"TaskOne应用"的应用中, Activity C和Activity D在名称为"TaskTwo应用"的应用中.

操作1: 在Launcher中单击"TaskOne应用"图标, Activity A启动开僻Task堆栈, 命名为TaskA(TaskA堆栈状态: A),在Activity A中启动Activity C, 启动Activity C的Intent的Flag设为FLAG\_ACTIVITY\_NEW\_TASK,Android系统会为Activity C开僻一个新的Task, 命名为TaskB(TaskB堆栈状态: C), 长按Home键, 选择TaskA,Activity A回到前台, 再次启动Activity C（两种情况1.从桌面启动；2.从Activity A启动，两种情况一样）, 这时TaskB回到前台, Activity C显示, 供用户使用, 即:包含FLAG\_ACTIVITY\_NEW\_TASK的Intent启动Activity的Task正在运行, 则不会为该Activity创建新的Task,而是将原有的Task返回到前台显示.

操作2: 在Launcher中单击"TaskOne应用"图标, Activity A启动开僻Task堆栈, 命名为TaskA(TaskA堆栈状态: A),在Activity A中启动Activity C,启动Activity C的Intent的Flag设为FLAG\_ACTIVITY\_NEW\_TASK,Android系统会为Activity C开僻一个新的Task, 命名为TaskB(TaskB堆栈状态: C),  在Activity C中启动Activity D(TaskB的状态: CD) 长按Home键, 选择TaskA, Activity A回到前台, 再次启动Activity C(从桌面或者ActivityA启动，也是一样的),这时TaskB回到前台, Activity D显示,供用户使用.说明了在此种情况下设置FLAG\_ACTIVITY\_NEW\_TASK后，会先查找是不是有Activity C存在的栈，根据亲和性(taskAffinity)，如果有，刚直接把这个栈整体移动到前台，并保持栈中的状态不变，即栈中的顺序不变

(2) FLAG\_ACTIVITY\_CLEAR\_TOP:

前提: Activity A, Activity B, Activity C和Activity D在同一个应用中.

操作: Activity A启动开僻Task堆栈(堆栈状态: A), 在Activity A中启动Activity B(堆栈状态: AB), 在Activity B中启动Activity C(堆栈状态: ABC), 在Activity C中启动Activity D(堆栈状态: ABCD), 在Activity D中启动Activity B,启动Activity B的Intent的Flag设置为FLAG\_ACTIVITY\_CLEAR\_TOP, (堆栈状态: AB).

(3) FLAG\_ACTIVITY\_BROUGHT\_TO\_FRONT:

前提: Activity A在名称为"TaskOne应用"的应用中, Activity C和Activity D在名称为"TaskTwo应用"的应用中.

操作: 在Launcher中单击"TaskOne应用"图标, Activity A启动开僻Task堆栈, 命名为TaskA(TaskA堆栈状态: A),在Activity A中启动Activity C,启动Activity C的Intent的Flag设为FLAG\_ACTIVITY\_NEW\_TASK,Android系统会为Activity C开僻一个新的Task, 命名为TaskB(TaskB堆栈状态: C), 在Activity C中启动Activity D(TaskB的堆栈状态: CD), 长按Home键, 选择TaskA, Activity A回到前台, 在Activity A中再次启动Activity C,在启动Activity C的Intent中设置Flag为FLAG\_ACTIVITY\_BROUGHT\_TO\_FRONT, TaskB回到前台,Activity C显示, (TaskB的堆栈状态: C).

(4) FLAG\_ACTIVITY\_MULTIPLE\_TASK:

与FLAG\_ACTIVITY\_NEW\_TASK结合使用, 首先在Intent中设置FLAG\_ACTIVITY\_NEW\_TASK, 打开Activity,则启动一个新Task, 接着在Intent中设置FLAG\_ACTIVITY\_MULTIPLE\_TASK, 再次打开同一个Activity,则还会新启动一个Task.

(5) FLAG\_ACTIVITY\_SINGLE\_TOP:

当前Task堆栈中存在ABCD四个Activity, A是栈顶Activity, D为栈底Activity, 存在打开A的Intent中设置了

FLAG\_ACTIVITY\_SINGLE\_TOP标志, 则会使用栈顶A, 而不会从新New A.

(6) FLAG\_ACTIVITY\_RESET\_TASK\_IF\_NEEDED:

一般与FLAG\_ACTIVITY\_CLEAR\_WHEN\_TASK\_RESET结合使用，如果设置该属性，这个activity将在一个新的task中启动或者或者被带到一个已经存在的task的顶部，这时这个activity将会作为这个task的首个页面加载。将会导致与这个应用具有相同亲和力的task处于一个合适的状态(移动activity到这个task或者从中移出)，或者简单的重置这个task到它的初始状态

FLAG\_ACTIVITY\_CLEAR\_WHEN\_TASK\_RESET：在当前的Task堆栈中设置一个还原点，当带有FLAG\_ACTIVITY\_RESET\_TASK\_IF\_NEEDED的Intent请求启动这个堆栈时(典型的例子是用户从桌面再次启动这个应用)，还原点之上包括这个应用将会被清除。应用场景：在email程序中预览图片时，会启动图片观览的actvity，当用户离开email处理其他事情，然后下次再次从home进入email时，我们呈现给用户的应该是上次email的会话，而不是图片观览，这样才不会给用户造成困惑。

例: 存在Activity A, Activity B, Activity C, Activity A启动开僻Task堆栈, 命名为TaskA(TaskA堆栈状态: A),

在Activity A中启动Activity B(TaskA堆栈状态: AB), 接着Activity B启动Activity C(TaskA堆栈状态: ABC),

  启动Activity C的Intent中设置FLAG\_ACTIVITY\_CLEAR\_WHEN\_TASK\_RESET标题, 这样TaskA中有一个还原点,

当有包含FLAG\_ACTIVITY\_RESET\_TASK\_IF\_NEEDED的Intent请求TaskA堆栈时(比如请求Activity A)

系统就会将还原点以上的Activity清除, TaskA堆栈中只剩下了AB.

# Application全局化

## 用处一：Activity之间传递数据

**public** **class** **MainActivity** **extends** **Activity** {

...

**HashMap**<**String**, **Object**> map = **new** HashMap<>();

**String**[] arr = **new** **String**[]{"fasdf", "fasdgasdgdf", "fasfddf"};

map.put("arr", arr);

**MyApplication**.getInstance().setMap(map);

startActivity(**new** Intent(**MainActivity**.**this**, **MainActivity2**.**class**));

finish();

}

}

**public** **class** **MainActivity2** **extends** **Activity** {

...

**HashMap**<**String**, **Object**> map = **MyApplication**.getInstance().getMap();

**String** string = **Arrays**.toString((**String**[]) map.get("arr"));

**Log**.e("tga",string);

**TextView** textView = (**TextView**) findViewById(**R**.**id**.**tv**);

textView.setText(string);

}

}

**public** **class** **MyApplication** **extends** **Application** {

**private** **static** **MyApplication** **mInstance**;

@Override

**public** **void** **onCreate**() {

**super**.onCreate();

**mInstance** = **this**;

}

**public** **static** **MyApplication** **getInstance**(){

**return** **mInstance**;

}

**private** **HashMap**<**String**, **Object**> map = **new** HashMap<>();

**public** **void** **setMap**(**HashMap**<**String**, **Object**> map){

**this**.map = map;

}

**public** **HashMap**<**String**, **Object**> **getMap**(){

**return** map;

}

}

注意：Application全局的时候不要用double-check的单例模式，否则获取出来的AppLication有问题

正确获取方式

public class App extends Application{  
 private static App *instance*;  
 @Override  
 public void onCreate() {  
 super.onCreate();  
 *instance* = this;  
 }  
 public static App getInstance() {  
 return *instance*;  
 }  
}

错误获取方式

public class App extends Application{  
 private static App *instance*;  
 @Override  
 public void onCreate() {  
 super.onCreate();  
 }  
 public static synchronized App getInstance() {  
 if (*instance* == null) {  
 synchronized (App.class) {  
 if (*instance* == null) {  
 *instance* = new App();  
 }  
 }  
 }  
 return *instance*;  
 }  
}

# Context

## 什么是Context

Context，翻译为上下文，环境。不过又想问啥又是上下文，啥又是环境，程序还有上下文。。。为了不误人子弟，来Google的官方说法：Interface to global information about an application environment. This is an abstract class whose implementation is provided by the Android system. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting and receiving intents, etc翻译下：它是一个应用程序的全局环境，是Android系统的一个抽象类，可以通过它获取程序的资源，比如：加载Activity，广播，接收Intent信息等等。总的来说它就像是一个程序运行的时候的环境，如果Activity，Service这些是水里的鱼，那它就是水

## 看代码

首先它是个抽象类

public abstract class Context {}

它提供了哪些方法，哎，太多了，随便看几个吧：

//获取主线程的Looper

public abstract Looper getMainLooper();

//获取当前应用上下文

public abstract Context getApplicationContext();

//开启activity

public abstract void startActivity(Intent intent);

//获取valus/strings.xml声明的字符串

public final String getString(@StringRes int resId) {

return getResources().getString(resId);

}

//获取valus/colors.xml声明的颜色

public final int getColor(int id) {

return getResources().getColor(id, getTheme());

}

//发送广播

public abstract void sendBroadcast(Intent intent);

//开启服务

public abstract ComponentName startService(Intent service);

//获取系统服务（ALARM\_SERVICE，WINDOW\_SERVICE，AUDIO\_SERVICE、、、）

public abstract Object getSystemService(@ServiceName @NonNull String name);

## getApplicationContext和getApplication1

严格意义上来说getApplicationContext和getApplication是不一样的，虽然很多时候他们返回的都是同一个对象，但是getApplication只存在于Activity或者Service中，我们要注意具体的情况，这个我们后面再说

**Application为何物**

Application和Activity都继承自Context,他们都是环境，只不过Application是随着我们的应用（或者包）启动的时候就存在的环境，Activity是一个界面的环境

**使用方法**

既然Application是在应用一创建就初始化了，而且是在应用运行时一直存在的，那我们可以把它当做是一个全局变量来使用，可以保存一些共享的数据，或者说做一些工具类的初始化工作。要自己来使用Application的话我们需要先新建一个类来继承Application

**说说Application的获取问题**

一个方法是我们直接 (MyApplication)getApplication()，但是还有一种更常见的做法，要在其他没有Context的地方也能拿到怎么办呢？可以这样，仿照单例的做法（只是仿照！），在MyApplication声明一个静态变量

public class MyApplication extends Application {

private static MyApplication instance;

@Override

public void onCreate() {

super.onCreate();

instance = this;

}

// 获取ApplicationContext

public static Context getMyApplication() {

return instance;

}

}

至此我们拿到了MyApplication实例，注意这跟我们常见的单例不一样，不要自作聪明去在getMyApplication里面做一下空的判断，Application在应用中本来就是一个单例，所以每次返回的都是同一个实体

**总结**

Application，Activity，Service都是继承自Context，是应用运行时的环境，我们可以把Application看做是应用，Activity看做是一个界面，至于getApplicationContext和getApplication，他们返回的对象有可能不一样（虽然大部分时间是一样的，都是整个应用的上下文），如果想要拿到在manifest里面声明的那个Application，务必用getApplication

还有就是因为他们都继承自Context，比如在打开Dialog的时候好像是都可以，其实不然，比如我们大多数情况：

AlertDialog.Builder builder = new Builder(Activity.this);//可以

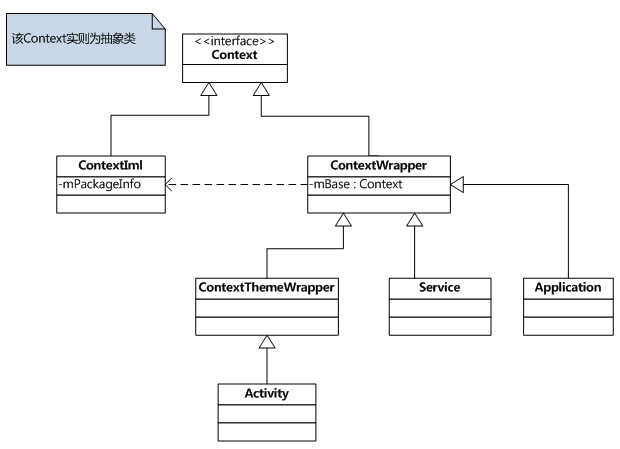
AlertDialog.Builder builder = new Builder(getApplicationContext());//内存泄漏

如果把this换成getApplicationContext()，不会报错，但是就如我们刚才所说，getApplicationContext() 返回的上下文会随着应用一直存在，而这里的Dialog应该属于Activity，Activity关闭了我们无法销毁上下文（Dialog持有全局的上下文）

## Context的生命周期

getApplicationContext() 生命周期是整个应用，应用摧毁它摧毁 Activity.this的context属于activity ，activity 摧毁他摧毁activity.this要返回一个activity，而getApplicationContext()就不一定返回一个activity。通过它我们可以获取应用程序的资源和类，也包括一些应用级别操作，例如：启动一个Activity，发送广播，接受Intent信息等。。

## Context相关类的继承关系



**Context类，**路径： /frameworks/base/core/Java/android/content/Context.java

说明：抽象类，提供了一组通用的API。

**ContextIml类，**路径 ：/frameworks/base/core/java/android/app/ContextImpl.java

说明：该Context类的实现类为ContextIml，该类实现了Context类的功能。请注意，该函数的大部分功能都是直接调用。其属性mPackageInfo去完成，这点我们后面会讲到。

**ContextWrapper类，**路径 ：\frameworks\base\core\java\android\content\ContextWrapper.java

说明：正如其名称一样，该类只是对Context类的一种包装，该类的构造函数包含了一个真正的Context引用，即ContextIml对象。

**ContextThemeWrapper类，**路径：/frameworks/base/core/java/android/view/ContextThemeWrapper.java

说明：该类内部包含了主题(Theme)相关的接口，即android:theme属性指定的。只有Activity需要主题，Service不需要主题，所以Service直接继承于ContextWrapper类。

## 什么时候创建Context实例

熟悉了Context的继承关系后，我们接下来分析应用程序在什么情况需要创建Context对象的？应用程序创建Context实例的

情况有如下几种情况：

1、创建Application 对象时， 而且整个App共一个Application对象

2、创建Service对象时

3、创建Activity对象时

因此应用程序App共有的Context数目公式为：

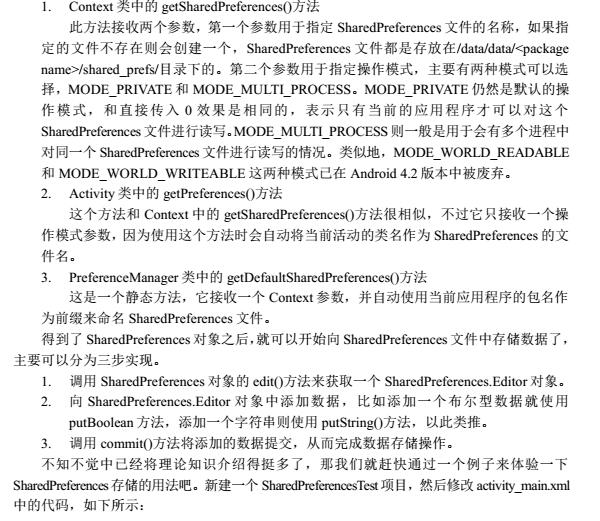
总Context实例个数 = Service个数 + Activity个数 + 1（Application对应的Context实例）

具体创建Context的时机





# SharedPreferences保存本地配置



SharedPreferences config0 = getPreferences(**MODE\_PRIVATE**);

SharedPreferences config1 = **PreferenceManager**.getDefaultSharedPreferences(**MainActivity**.**this**);

SharedPreferences config = getSharedPreferences("config", **MODE\_PRIVATE**);

Editor edit = config.edit();

edit.putBoolean("isChecked", isChecked);

edit.remove("username");

edit.commit();

config.getString("password", "");

# startActivityForResult、request、resultCode

## 调用startActivityForResult后，onActivityResult立刻响应

因是在AndroidManifest.xml 中跳转到的页面我自己设置了android:launchMode="singleTask"，因为需要传值的 Activity 不容许设置该属性或者 singleInstance，或只能设为标准模式，不然将在 startActivityForResult()后直接调用 onActivityResult()。  
解决办法：去掉跳转到的页面的lanchMode属性即可。  
另外，requestCode值必须要[大于等于](https://www.baidu.com/s?wd=%E5%A4%A7%E4%BA%8E%E7%AD%89%E4%BA%8E&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y1PW61uHfknHPhrHR1myRk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErH61rj6LnWTknj0LPjcLnWD4" \t "http://zhidao.baidu.com/_blank)0，不然，startActivityForResult就变成了 startactivity。

## 请求码的作用

使用startActivityForResult(Intent intent, int requestCode)方法打开新的Activity，我们需要为startActivityForResult()方法传入一个请求码(第二个参数)。请求码的值是根据业务需要由自已设定，用于标识请求来源。例如：一个Activity有两个按钮，点击这两个按钮都会打开同一个Activity，不管是那个按钮打开新Activity，当这个新Activity关闭后，系统都会调用前面Activity的onActivityResult(int requestCode, int resultCode, Intent data)方法。在onActivityResult()方法如果需要知道新Activity是由那个按钮打开的，并且要做出相应的业务处理

**case** **R**.**id**.**btn\_xuanze**:

//得到新打开Activity关闭后返回的数据

//第二个参数为请求码，可以根据业务需求自己编号

startActivityForResult(**new** Intent(**MainActivity**.**this**, **ResultActivity**.**class**), 1);

**break**;

**case** **R**.**id**.**btn\_xuanze2**:

startActivityForResult(**new** Intent(**MainActivity**.**this**, **ResultActivity**.**class**), 2)

**protected** **void** **onActivityResult**(**int** requestCode, **int** resultCode, **Intent** data) {

**super**.onActivityResult(requestCode, resultCode, data);

**Bundle** bundle = data.getExtras();

**String** string = bundle.getString("haha");

**switch** (requestCode) {

**case** 1:

**Toast**.makeText(**MainActivity**.**this**, string, 1000).show();

**break**;

**case** 2:

**Toast**.makeText(**MainActivity**.**this**, string, 1000).show();

**public** **class** ResultActivity **extends** Activity {

//数据是使用Intent返回

**Intent** intent = **new** Intent();

//把返回数据存入Intent

intent.putExtra("result", "My name is linjiqin");

//设置返回数据

**ResultActivity**.**this**.setResult(**RESULT\_OK**, intent);

//关闭Activity

**ResultActivity**.**this**.finish();

## 结果码的作用

在一个Activity中，可能会使用startActivityForResult()方法打开多个不同的Activity处理不同的业务，当这些新Activity关闭后，系统都会调用前面Activity的onActivityResult(int requestCode, int resultCode, Intent data)方法。为了知道返回的数据来自于哪个新Activity，在onActivityResult()方法中可以这样做(ResultActivity和NewActivity为要打开的新Activity)：

# onSaveInstanceState

## 保存Activity的状态

public void onSaveInstanceState(Bundle outState) {

outState.putString("str", str);

super.onSaveInstanceState(outState);

}

## 恢复Activity的状态

protected void onRestoreInstanceState(Bundle savedInstanceState) {

super.onRestoreInstanceState(savedInstanceState);

if(savedInstanceState==null) {

return;

}

str=savedInstanceState.getString("str");

}

## [保存数据（意外被清理的情况下）onRestoreInstanceState和onSaveInstanceState学习](http://blog.csdn.net/ljz2009y/article/details/8518720)

一、打开应用程序，然后使用back按钮回退出来

点击BACK按钮的时候，没有调用onSaveInstanceState的方法，原因在于BACK是用户主动退出需要销毁的，所以就没有进行保存的必要了。

二、启动程序（先前没有启动过），按HOME按钮回到HOME窗口

onCreate   
onStart   
onResume   
onSaveInstanceState  
onPause   
onStop

当我点击HOME按钮的时候，系统就自动调用了onSaveInstanceState的函数而启动360清除全部内存（模拟内容匮乏的情况），在点击应用程序启动应用。  
onCreate   
onStart   
onRestoreInstanceState  
onResume

onSaveInstanceState方法会在什么时候被执行，有这么几种情况：

1、当用户按下HOME键时。  
2、长按HOME键，选择运行其他的程序时。  
3、按下电源按键（关闭屏幕显示）时。  
4、从activity A中启动一个新的activity时。  
5、屏幕方向切换时，例如从竖屏切换到横屏时。

总而言之，onSaveInstanceState的调用遵循一个重要原则，即当系统“未经你许可”时销毁了你的activity，则onSaveInstanceState会被系统调用，这是系统的责任  
至于onRestoreInstanceState方法，

需要注意的是，onSaveInstanceState方法和onRestoreInstanceState方法“不一定”是成对的被调用的，onRestoreInstanceState被调用的前提是，activity A“确实”被系统销毁了，例如，当正在显示activity A的时候，用户按下HOME键回到主界面，然后用户紧接着又返回到activity A，这种情况下activity A一般不会因为内存的原因被系统销毁，故activity A的onRestoreInstanceState方法不会被执行（ondestroy都不会执行）

这个时候就算是没有id的view都不会消失  
另外，onRestoreInstanceState的bundle参数也会传递到onCreate方法中，你也可以选择在onCreate方法中做数据还原

**(View要存档必须在xml中要有一个id，否则存不了!!!!!!!!)**

**原因：**

# 静态导入import static和import的区别

import static静态导入是JDK1.5中的新特性。一般我们导入一个类都用 import com….ClassName;而静态导入是这样：import static com….ClassName.\*;这里的多了个static，还有就是类名ClassName后面多了个 .\* ，意思是导入这个类里的静态方法。当然，也可以只导入某个静态方法，只要把 .\* 换成静态方法名就行了。然后在这个类中，就可以直接用方法名调用静态方法，而不必用ClassName.方法名 的方式来调用。这种方法建议在有很多重复调用的时候使用，如果仅有一到两次调用，不如直接写来的方便

在Java 5中，import语句得到了增强，以便提供甚至更加强大的减少击键次数功能，虽然一些人争议说这是以可读性为代价的。这种新的特性成为静态导入。当你想使用static成员时，可以使用静态导入（在API中的类和你自己的类上，都可以使用该特性）。下面是静态导入前后的代码实例：

在静态导入之前：

import com.zhualaizhuaqu.gift.utils.StrNumUtil;

public class TestStatic {

public static void main(String[] args) {

int a = StrNumUtil.Str2Int("7");

String b = StrNumUtil.getEmptyStr(titleStr, "抓来抓趣");

}

}

在静态导入之后：

import static com.zhualaizhuaqu.gift.utils.StrNumUtil.getEmptyStr;

public class TestStaticImport {

public static void main(String[] args) {

int a = Str2Int("7");

String b = getEmptyStr(titleStr, "抓来抓趣");

}

}

# Android工程目录结构

## src

顾名思义（src, source code）该文件夹是放项目的源代码的

## gen

该 文件夹下面有个R.java文件，R.java是在建立项目时自动生成的，这个文件是只读模式的，不能更改。R.java文件中定义了一个类——R，R类 中包含很多静态类，且静态类的名字都与res中的一个名字对应，即R类定义该项目所有资源的索引。看我们的HelloWorld项目是不是如此，如下图：

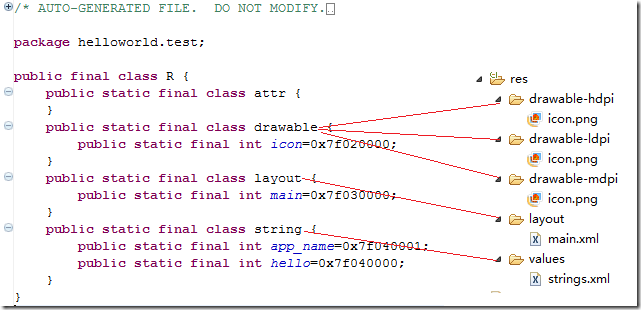
[](http://images.51cto.com/files/uploadimg/20130117/1551101.png" \t "http://mobile.51cto.com/_blank)

图2、R.java对应res

通过R.java我们可以很快地查找我们需要的资源，另外编绎器也会检查R.java列表中的资源是否被使用到，没有被使用到的资源不会编绎进软件中，这样可以减少应用在手机占用的空间

## Android n.n.n

该 文件夹下包含android.jar文件，这是一个Java 归档文件，其中包含构建应用程序所需的所有的Android SDK 库（如Views、Controls）和APIs。通过android.jar将自己的应用程序绑定到Android SDK和Android Emulator，这允许你使用所有Android的库和包，且使你的应用程序在适当的环境中调试。

## assets（资产财产）

包含应用系统需要使用到的诸如mp3、视频类的文件

## res文件夹（资源）

资源目录，包含你项目中的资源文件并将编译进应用程序。向此目录添加资源时，会被R.java自动记录。

## AndroidManifest.xml

项目的总配置文件，记录应用中所使用的各种组件

## libs

项目手动导入的jar包，可以是第三方的（需要手动Build），可以是扩展包（support，自动build到Android Private Libraies）

## bin

系统运行自动编译的文件，不用管它，系统运行后，apk生成在里边

# [res/raw和assets的使用](http://263229365.iteye.com/blog/1314063)

## res/raw和assets的相同点：

1. 两者目录下的文件在打包后会原封不动的保存在apk包中，不会被编译成二进制。

## res/raw和assets的不同点：

1.res/raw中的文件会被映射到R.java文件中，访问的时候直接使用资源ID即R.id.filename；assets文件夹下的文件不会被映射到R.java中，访问的时候需要AssetManager类。  
2.res/raw不可以有目录结构，而assets则可以有目录结构，也就是assets目录下可以再建立文件夹

## 读取文件资源：

1.读取res/raw下的文件资源，通过以下方式获取输入流来进行写操作

InputStream is = getResources().openRawResource(R.raw.filename);

2.读取assets下的文件资源，通过以下方式获取输入流来进行写操作

AssetManager am = null;

am = getAssets();

InputStream is = am.open("filename");  //比如：assetManager.open("yshd-ky-api-1.1.0.js")  
getAssets().list("");来获取assets目录下所有文件夹和文件的名称

注意1：Google的Android系统处理Asset有个bug，在AssetManager中不能处理单个超过1MB的文件，不然会报异常，raw没这个限制可以放个4MB的Mp3文件没问题。

注意2：assets 文件夹是存放不进行编译加工的原生文件，即该文件夹里面的文件不会像 xml， java 文件被预编译，可以存放一些图片，html，js, css 等文件。

# getFilesDir、getCacheDir、openFileInput

## 检查是否挂载SD卡

**private** **void** **existSDcard**(){

**String** state = **Environment**.getExternalStorageState();**//获取SD卡状态**

**if** (**Environment**.**MEDIA\_MOUNTED**.equals(state)) {

**Toast**.makeText(**MainActivity**.**this**, "有SD卡", **Toast**.**LENGTH\_LONG**).show();

}

}

## 通过API获取路径

**private** **void** **listPath**(){

file = **Environment**.getExternalStorageDirectory();**//获取sd卡目录**

file2 = getFilesDir(); **//app内部存储目录**

file3 = getCacheDir(); **//app内部存储空间的缓存目录**

}

**/storage/sdcard1**

**/data/data/com.example.duxiebendiwenjian/files**

**/data/data/com.example.duxiebendiwenjian/cache**

## 文件的I/O是存放在/data/data/<package name>/file/filename目录下。



在4.4版本后由于安全问题，mode已经无效，建议使用contentProvider和contentResolver

## 获取外置内存卡的方法

### 方法一使用System类（Api<=23）

//内置sd卡路径 String sdcardPath = System.getenv("EXTERNAL\_STORAGE");

//内置sd卡路径 String sdcardPath = Environment.getExternalStorageDirectory().getAbsolutePath();

//外置置sd卡路径 String extSdcardPath = System.getenv("SECONDARY\_STORAGE");

在Enviroment类的源码中获得sd卡路径其实也是通过 System.getnv() 方法来实现的，如隐藏的方法:

/\*\* {@hide} \*/

public static File getLegacyExternalStorageDirectory() {

return new File(System.getenv(ENV\_EXTERNAL\_STORAGE));

}

另外要注意的是，在API 23版本中 SECONDARY\_STORAGE 被移除。

### 方法二通过反射

//is\_removable为false时得到的是内置sd卡路径，为true则为外置sd卡路径

**private** **static** **String** **getStoragePath**(**Context** mContext, **boolean** is\_removale) {

**StorageManager** mStorageManager = (**StorageManager**) mContext.getSystemService(**Context**.**STORAGE\_SERVICE**);

**Class**<?> storageVolumeClazz = **null**;

**try** {

storageVolumeClazz = **Class**.*forName*("android.os.storage.StorageVolume");

**Method** getVolumeList = mStorageManager.getClass().getMethod("getVolumeList");

**Method** getPath = storageVolumeClazz.getMethod("getPath");

**Method** isRemovable = storageVolumeClazz.getMethod("isRemovable");

**Object** result = getVolumeList.invoke(mStorageManager);

**final** **int** length = **Array**.*getLength*(result);

**for** (**int** i = 0; i < length; i++) {

**Object** storageVolumeElement = **Array**.*get*(result, i);

**String** path = (**String**) getPath.invoke(storageVolumeElement);

**boolean** removable = (**Boolean**) isRemovable.invoke(storageVolumeElement);

**if** (is\_removale == removable) {

**return** path;

}

}

} **catch** (**ClassNotFoundException** e) {

e.printStackTrace();

} **catch** (**InvocationTargetException** e) {

e.printStackTrace();

} **catch** (**NoSuchMethodException** e) {

e.printStackTrace();

} **catch** (**IllegalAccessException** e) {

e.printStackTrace();

}

**return** **null**;

}

# 播放系统提示音和默认震动

**Uri notification = RingtoneManager.getDefaultUri(RingtoneManager.TYPE\_NOTIFICATION);  
Ringtone r = RingtoneManager.getRingtone((context, notification);  
r.play();**

**((Vibrator) context.getSystemService(Context.VIBRATOR\_SERVICE)).vibrate(200);**

# Notification通知栏

注意在26

## 代码

**private** **NotificationManager** mNotificationManager;

**private** **NotificationCompat**.**Builder** mBuilder;

**int** notifyId = 100;

mNotificationManager = (**NotificationManager**) getSystemService(**NOTIFICATION\_SERVICE**);

Notification notify = mBuilder.build();

notify.flags = Notification.FLAG\_ONGOING\_EVENT;

**//FLAG\_ONGOING\_EVENT 在顶部常驻， FLAG\_AUTO\_CANCEL 点击清理**

mNotificationManager.notify(notifyId, notify);

**private** **void** **initNotify**(){

**//点击的意图ACTION是跳转到Intent**

**Intent** resultIntent = **new** Intent(**this**, **MainActivity**.**class**);

resultIntent.setFlags(**Intent**.**FLAG\_ACTIVITY\_SINGLE\_TOP**);

**//创建构造Builder**

mBuilder = **new** **NotificationCompat**.Builder(**this**);

**//设置通知栏标题**

mBuilder.setContentTitle("测试标题")

**//设置通知栏显示内容**

.setContentText("测试内容")

//**设置通知栏点击意图，只能用pendingIntent**

**PendingIntent** pendingIntent = **PendingIntent**.getActivity(**this**, 0,resultIntent, **PendingIntent**.**FLAG\_UPDATE\_CURRENT**);

.setContentIntent(pendingIntent)

**//通知首次出现在通知栏，带上升动画效果的**

.setTicker("测试通知来啦")

**//设置通知小ICON**

.setSmallIcon(**R**.**drawable**.**ic\_launcher**)

**//通知产生的时间，会在通知信息里显示**

.setWhen(**System**.currentTimeMillis())

**//设置该通知优先级**

.setPriority(**Notification**.**PRIORITY\_DEFAULT**)

**//.setNumber(number)//显示数量**

**//.setAutoCancel(true)//设置这个标志当用户单击面板就可以让通知将自动取消**

**//ture，设置他为一个正在进行的通知。他们通常是用来表示一个后台任务,用户积极参与(如播放音乐)或以某种方式正在等待,因此占用设备(如一个文件下载,同步操作,主动网络连接)**

.setOngoing(**false**)

**//向通知添加声音、闪灯和振动效果，使用defaults属性，可以组合，Notification.DEFAULT\_ALL Notification.DEFAULT\_SOUND 添加声音 // requires VIBRATE permission**

.setDefaults(**Notification**.**DEFAULT\_VIBRATE**|**Notification**.**DEFAULT\_LIGHTS**|**Notification**.**DEFAULT\_SOUND**);

}

## 自定义布局的写法

**Notification** notification = **new** Notification();

notification.tickerText = fileInfo.getFileName();

notification.when = **System**.currentTimeMillis();

notification.icon = **R**.**drawable**.**ic\_launcher**;

notification.flags = **Notification**.**FLAG\_ONGOING\_EVENT**;

**Intent** intent = **new** Intent(mContext, **MainActivity**.**class**);

**PendingIntent** pIntent = **PendingIntent**.getActivity(mContext, 0, intent, 0);

notification.contentIntent = pIntent;

**//创建RemoteViews对象**

**RemoteViews** remoteViews = **new** RemoteViews(mContext.getPackageName(), **R**.**layout**.**notification\_layout**);

**//设置开始按钮操作**

**Intent** intentStart = **new** Intent(mContext, **MainActivity**.**class**);

intentStart.setAction(**DownloadService**.**ACTION\_START**);

intentStart.putExtra("fileInfo", fileInfo);

**PendingIntent** pIntentStart = **PendingIntent**.getService(mContext, 0, intentStart, 0);

remoteViews.setOnClickPendingIntent(**R**.**id**.**btStart**, pIntentStart);

**//设置结束按钮操作**

**Intent** intentStop = **new** Intent(mContext, **MainActivity**.**class**);

intentStop.setAction(**DownloadService**.**ACTION\_STOP**);

intentStop.putExtra("fileInfo", fileInfo);

**PendingIntent** pIntentStop = **PendingIntent**.getService(mContext, 0, intentStop, 0);

remoteViews.setOnClickPendingIntent(**R**.**id**.**btStop**, pIntentStop);

**//设置Notification的视图**

notification.contentView = remoteViews;

**Notification** notification = **new** Notification();

**Uri** soundUri = **Uri**.fromFile(**new** File(path));

notification.sound = soundUri;

**long**[] vibrates = {0,1000,1000,1000};//等待0，震动1000，等待1000，震动1000.......

notification.vibrate = vibrates;

notification.defaults = **Notification**.**DEFAULT\_ALL**;//默认的震动声音灯光

**//发出通知**

mNotificationManager.notify(fileInfo.getId(), notification);

## 通常驻状态下想要取消

**可以通过**mNotificationManager.cancle(id),id的值就是notify时传入的值；

# PendingIntent

intent英文意思是意图，pending表示即将发生或来临的事情。 PendingIntent这个类用于处理即将发生的事情。比如在通知Notification中用于跳转页面，但不是马上跳转。 Intent 是及时启动，intent 随所在的activity 消失而消失。

PendingIntent 可以看作是对intent的包装，通常通过getActivity,getBroadcast ,getService来得到pendingintent的实例，[分别对应着Intent的3个行为，跳转到一个activity组件、打开一个广播组件和打开一个服务组件。](http://www.cmd100.com/docs/reference/android/app/PendingIntent.html" \l "getService(android.content.Context, int, android.content.Intent, int))当前activity并不能马上启动它所包含的intent,而是在外部执行 pendingintent时，调用intent的。正由于pendingintent中 保存有当前App的Context，使它赋予外部App一种能力，使得外部App可以如同当前App一样的执行pendingintent里的 Intent， 就算在执行时当前App已经不存在了，也能通过存在pendingintent里的Context照样执行Intent。另外还可以处理intent执行后的操作。常和alermanger 和notificationmanager一起使用。   
 Intent一般是用作Activity、Sercvice、BroadcastReceiver之间传递数据，而Pendingintent，一般用在 Notification上，可以理解为延迟执行的intent，PendingIntent是对Intent一个包装。

PendingIntent.getActivity(context, 0, notificationIntent, flag);

**当使用FLAG\_UPDATE\_CURRENT时：**

FLAG\_UPDATE\_CURRENT会更新之前PendingIntent的消息，比如，你推送了消息1，并在其中的Intent中putExtra了一个值“ABC”，在未点击该消息前，继续推送第二条消息，并在其中的Intent中putExtra了一个值“CBA”，好了，这时候，如果你单击消息1或者消息2，你会发现，他俩个的Intent中读取过来的信息都是“CBA”，就是说，第二个替换了第一个的内容

**当使用FLAG\_CANCEL\_CURRENT时：**

依然是上面的操作步骤，这时候会发现，点击消息1时，没反应，第二条可以点击。导致上面两个问题的原因就在于第二个参数requestCode，当requestCode值一样时，后面的就会对之前的消息起作用，所以为了避免影响之前的消息，requestCode每次要设置不同的内容。

# 音频播放（MediaPlayer和SoundPool）

## 使用MediaPlayer播放音频

### 从资源文件中播放

**通过Creat方式：**

MediaPlayer player = new MediaPlayer.create(this,R.raw.test);

player.start();//不用prepare,又因为creat()中已经prepare()

**通过设置数据源的方式：**

player.setDataSource(**MainActivity**.**this**, **Uri**.parse("android.resource://" + getPackageName() + "/" + **R**.**raw**.test))

player.prepare();

player.start();

### 从文件系统（内存卡）播放

String path = Environment.getExternalStorageDirectory() + "/mediademo.mp3";

player.setDataSource(path);

player.prepare();

player.start();

### 从网络播放

**通过URI的方式:**

String path = "http://\*\*\*\*\*\*\*\*\*\*\*\*\*\*.mp3"; //这里给一个歌曲的网络地址就行了

Uri uri = Uri.parse(path);

MediaPlayer player = new MediaPlayer.create(this,uri);

player.start();

**通过设置数据源的方式：**

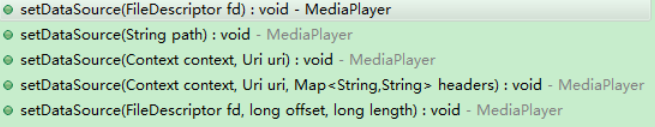
MediaPlayer player = new MediaPlayer();

String path = "http://\*\*\*\*\*\*\*\*\*\*\*\*\*\*.mp3"; //这里给一个歌曲的网络地址就行了

player.setDataSource(path);

player.prepare();

player.start();



### 结束播放

@Override

protected void onPause() {

super.onPause();

if(player!=null){

isplay = false;

player.pause();//假如视频在播放，则暂停

}

}

## SoundPool播放raw文件夹里的提示音

**private** **void** **initSoundPool**() {

poolMap = **new** HashMap<**String**, **Integer**>();

/\*\* 创建一个SoundPool（构造函数）

SoundPool(int maxStream, int streamType, int srcQuality)

maxStream —— 同时播放的流的最大数量

streamType —— 流的类型，一般为STREAM\_MUSIC(具体在AudioManager类中列出)

srcQuality —— 采样率转化质量，当前无效果，使用0作为默认值\*/

sndPool = **new** **SoundPool**(4, **AudioManager**.**STREAM\_MUSIC**, 0);

/\*\* 加载音频资源

int load(Context context, int resId, int priority)

一个SoundPool能同时管理多个音频，所以可以通过多次调用load函数来记载，

如果记载成功将返回一个非0的soundID，用于播放时指定特定的音频

API中指出，其中的priority参数目前没有效果，建议设置为1\*/

poolMap.put("openmouth", sndPool.load(**this**, **R**.**raw**.**cloudwalk\_detection\_mouth\_open**, 1));

poolMap.put("turnleft", sndPool.load(**this**, **R**.**raw**.**cloudwalk\_detection\_yaw\_left**, 1));

poolMap.put("turnright", sndPool.load(**this**, **R**.**raw**.**cloudwalk\_detection\_yaw\_right**, 1));

}

**switch** (v.getId()) {

**case** **R**.**id**.**openmouth**:

/\*\* 播放控制

有以下几个函数可用于控制播放：

final int play(int soundID, float leftVolume, float rightVolume, int priority, int loop, float rate)

播放指定音频的音效，并返回一个streamID。

leftVolume、rightVolume左右声道音量，默认1.0f；

priority —— 流的优先级，值越大优先级高，

loop —— 循环播放的次数，0为值播放一次，-1为无限循环，其他值为播放loop+1次

rate —— 播放的速率，范围0.5-2.0(0.5为一半速率，1.0为正常速率，2.0为两倍速率)\*/

sndPool.play(poolMap.get("openmouth"), 1.0f, 1.0f, 1, 0, 1.0f);

**break**;

**case** **R**.**id**.**turnleft**:

sndPool.play(poolMap.get("turnleft"), 1.0f, 1.0f, 1, 0, 1.0f);

**break**;

**case** **R**.**id**.**turnright**:

sndPool.play(poolMap.get("turnright"), 1.0f, 1.0f, 1, 0, 1.0f);

**break**;

**default**: **break**; }

## SoundPool工具类

public class SoundPlayUtils {

public static SoundPool mSoundPlayer;

public static SoundPlayUtils soundPlayUtils;

private boolean isPlay = false;

/\*\*

\* 初始化

\*/

public static SoundPlayUtils init(Context context) {

if (soundPlayUtils == null) {

soundPlayUtils = new SoundPlayUtils();

mSoundPlayer = new SoundPool(4, AudioManager.STREAM\_MUSIC, 0);

mSoundPlayer.load(context, PlatformUtil.getPlatformForBackMusic(), 1);

mSoundPlayer.setOnLoadCompleteListener((soundPool, sampleId, status) -> {

try {

play(1);

} catch (Exception e) {

e.printStackTrace();

}

});

}

return soundPlayUtils;

}

/\*\*

\* 播放声音

\*/

public static void play(int soundID) {

if (mSoundPlayer != null) {

if (!soundPlayUtils.isPlay) {

mSoundPlayer.play(soundID, 1.0f, 1.0f, 1, -1, 1.0f);

}

soundPlayUtils.isPlay = true;

}

}

public static void resume() {

if (mSoundPlayer != null) {

mSoundPlayer.autoResume();

}

}

public static void pause() {

if (mSoundPlayer != null) {

mSoundPlayer.autoPause();

}

}

public static void destroyPlayer() {

try {

mSoundPlayer.stop(1);

mSoundPlayer.autoPause();

mSoundPlayer.release();

soundPlayUtils = null;

mSoundPlayer = null;

} catch (Exception e) {

e.printStackTrace();

}

}

}

# 视频播放VideoView



videoView = (**VideoView**) findViewById(**R**.**id**.**videoview**);

videoView.setOnTouchListener(**this**);

videoView.setVideoPath(dataPath);

//String videoPath = "android.resource://" + this.getPackageName() + "/" + R.raw.intro\_video;

//videoView.setVideoURI(Uri.parse(videoPath);

videoView.start();

@Override

**public** **boolean** **onTouch**(**View** v, **MotionEvent** event) {

getSurfaceHW(videoView);

**if** (isplay) {

mediaPause();

}**else** {

mediaRe();

}

**return** **false**;

}

## 要实现video在容器里全屏，需重新videoview的onMeasure方法

@Override

**protected** **void** **onMeasure**(**int** widthMeasureSpec, **int** heightMeasureSpec) {

**//下面的代码是让视频的播放的长宽是根据你设置的参数来决定**

**int** width = getDefaultSize(mVideoWidth, widthMeasureSpec);

**int** height = getDefaultSize(mVideoHeight, heightMeasureSpec);

setMeasuredDimension(width, height);

}

## 确保VideoView全屏且尽量不拉伸

@Override

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {

int w = MeasureSpec.getSize(widthMeasureSpec);

setMeasuredDimension(w, (int) (w / 0.56f));

}

# MediaPlayer

## 概述

mediaPlayer = **MediaPlayer**.create(**this**, **R**.**raw**.**test**);

1.在初始化MediaPlayer时，通过create方法设置数据源。则不能写MediaPlayer.prepare()方法，这时，会报错。

2.如果是使用MediaPlayer构造函数初始化MediaPlayer，然后通过setDataSource方法设置数据源时，就需要在start（）之前，使用MediaPlayer.prepare()方法，对数据源进行一次编译。能够避免出现（-38,0）这种错误。



## 奇怪的例子

/\*\*

\* 该实例中使用MediaPlayer完成播放，同时界面使用SurfaceView来实现

\* 这里我们实现MediaPlayer中很多状态变化时的监听器

\* 使用Mediaplayer时，也可以使用MediaController类，但是需要实现MediaController.mediaController接口

\* 实现一些控制方法。

\* 然后，设置controller.setMediaPlayer(),setAnchorView(),setEnabled(),show()就可以了，这里不再实现

\* **@author** Administrator

\*/

**public** **class** **MainActivity** **extends** **Activity** **implements** OnCompletionListener,OnErrorListener,OnInfoListener,

OnPreparedListener, OnSeekCompleteListener,OnVideoSizeChangedListener,SurfaceHolder.Callback,OnClickListener{

**private** **Display** currDisplay;

**private** **SurfaceView** surfaceView;

**private** SurfaceHolder holder;

**private** **MediaPlayer** player;

**private** **int** vWidth,vHeight;

**private** **boolean** isplay = **true**;

**private** **static** **String** **DEFAULT\_MEDIA\_PATH** = **Environment**.getExternalStorageDirectory().getPath()+"/mv.mp4";

**private** **String** dataPath = **DEFAULT\_MEDIA\_PATH**;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

surfaceView = (**SurfaceView**) findViewById(**R**.**id**.**surfaceview**);

**//给SurfaceView添加CallBack监听**

holder = surfaceView.getHolder();

holder.addCallback(**this**);

**//为了可以播放视频或者使用Camera预览，我们需要指定其Buffer类型**

//holder.**~~setType~~**(SurfaceHolder.**~~SURFACE\_TYPE\_PUSH\_BUFFERS~~**);

**//下面开始实例化MediaPlayer对象**

player = **new** MediaPlayer();

player.setOnCompletionListener(**this**);

player.setOnErrorListener(**this**);

player.setOnInfoListener(**this**);

player.setOnPreparedListener(**this**);

player.setOnSeekCompleteListener(**this**);

player.setOnVideoSizeChangedListener(**this**);

**//然后指定需要播放文件的路径，初始化MediaPlayer**

**try** {

player.setDataSource(dataPath);

} **catch** (**Exception** e) {

e.printStackTrace();

}

**//然后，我们取得当前Display对象**

currDisplay = **this**.getWindowManager().getDefaultDisplay();

surfaceView.setOnClickListener(**this**);

}

@Override

**public** **void** **surfaceCreated**(SurfaceHolder holder) {

**//当SurfaceView中的Surface被创建的时候被调用**

**//在这里我们指定MediaPlayer在当前的Surface中进行播放**

player.setDisplay(holder);

**//在指定了MediaPlayer播放的容器后，我们就可以使用prepare或者prepareAsync来准备播放了**

**try** {

player.prepare();

player.setLooping(**true**);**//循环播放**

} **catch** (**Exception** e) {

e.printStackTrace();

}

}

@Override

**public** **void** **surfaceChanged**(SurfaceHolder holder, **int** format, **int** width,

**int** height) {

**// 当Surface尺寸等参数改变时触发**

}

@Override

**public** **void** **surfaceDestroyed**(SurfaceHolder holder) {

**// 当SurfaceView中的Surface被销毁的时候被调用**

**if**(player!=**null**){

player.stop();**//假如视频在播放，则停止**

player.release();

player = **null**;

}

}

@Override

**public** **void** **onVideoSizeChanged**(**MediaPlayer** mp, **int** width, **int** height) {

**//当video大小改变时触发**

**//这个方法在设置player的source后至少触发一次**

}

@Override

**public** **void** **onSeekComplete**(**MediaPlayer** mp) {

**// seek操作完成时触发**

}

@Override

**public** **void** **onPrepared**(**MediaPlayer** mp) {

**// 当prepare完成后，该方法触发，在这里我们播放视频**

vWidth = player.getVideoWidth();

vHeight = player.getVideoHeight();

**if** (vWidth > currDisplay.**~~getWidth~~**() || vHeight > currDisplay.**~~getHeight~~**()) {

**//如果video的宽或者高超出了当前屏幕的大小，则要进行缩放**

**float** wRatio = (**float**)vWidth/(**float**)currDisplay.**~~getWidth~~**();

**float** hRatio = (**float**)vHeight/(**float**)currDisplay.**~~getHeight~~**();

**//选择大的一个进行缩放**

**float** ratio = **Math**.max(wRatio, hRatio);

vWidth = (**int**)**Math**.ceil((**float**)vWidth/ratio);

vHeight = (**int**)**Math**.ceil((**float**)vHeight/ratio);

//还有一种情况，万一是在电视上边放，视频本身大小小于屏幕的情况

**//设置surfaceView的布局参数**

surfaceView.setLayoutParams(**new** **RelativeLayout**.LayoutParams(vWidth, vHeight));

}

**//然后开始播放视频**

player.start();

}

@Override

**public** **boolean** **onInfo**(**MediaPlayer** mp, **int** what, **int** extra) {

**// 当一些特定信息出现或者警告时触发**

**switch**(what){

**case** **MediaPlayer**.**MEDIA\_INFO\_BAD\_INTERLEAVING**:

**break**;

**case** **MediaPlayer**.**MEDIA\_INFO\_METADATA\_UPDATE**:

**break**;

**case** **MediaPlayer**.**MEDIA\_INFO\_VIDEO\_TRACK\_LAGGING**:

**break**;

**case** **MediaPlayer**.**MEDIA\_INFO\_NOT\_SEEKABLE**:

**break**;

}

**return** **false**;

}

@Override

**public** **boolean** **onError**(**MediaPlayer** mp, **int** what, **int** extra) {

**switch** (what) {

**case** **MediaPlayer**.**MEDIA\_ERROR\_SERVER\_DIED**:

**break**;

**case** **MediaPlayer**.**MEDIA\_ERROR\_UNKNOWN**:

**break**;

**default**:

**break**;

}

**return** **false**;

}

@Override

**public** **void** **onCompletion**(**MediaPlayer** mp) {

**// TODO Auto-generated method stub**

**// 当MediaPlayer播放完成后触发**

}

@Override

**protected** **void** **onPause**() {

**// TODO Auto-generated method stub**

**super**.onPause();

mediaStop();

}

@Override

**protected** **void** **onDestroy**() {

**// TODO Auto-generated method stub**

**super**.onDestroy();

}

**//视频停止播放**

**private** **void** **mediaStop**(){

**if**(player!=**null**){

player.stop();**//假如视频在播放，则停止**

player.release();

player = **null**;

}

}

**//视频暂停播放**

**private** **void** **mediaPause**(){

**if**(player!=**null**){

isplay = **false**;

player.pause();**//假如视频在播放，则暂停**

}

}

**//视频继续播放**

**private** **void** **mediaRe**(){

**if**(player!=**null**){

isplay = **true**;

player.start();**//假如视频在播放，则暂停**

}

}

@Override

**public** **void** **onClick**(**View** v) {

**if** (isplay) {

mediaPause();

}**else** {

mediaRe();

}

}

}

## MediaPlayer +ViewPager+surfaceView

**public** **class** **MainActivity** **extends** **Activity** **implements** SurfaceHolder.Callback, OnPreparedListener{

**private** **ViewPager** mViewPager;

**private** List<**View**> views;

**private** SurfaceHolder holder1, holder2, holder3;

**private** **MediaPlayer** player;

**private** **String** Path;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

mViewPager = (**ViewPager**) findViewById(**R**.**id**.**viewpager**);

views = **new** ArrayList<>();

**View** view1 = getLayoutInflater().inflate(**R**.**layout**.**welcome\_one**, **null**);

**View** view2 = getLayoutInflater().inflate(**R**.**layout**.**welcome\_two**, **null**);

**View** view3 = getLayoutInflater().inflate(**R**.**layout**.**welcome\_three**, **null**);

**SurfaceView** sf1 = (**SurfaceView**) view1.findViewById(**R**.**id**.**sf1**);

**SurfaceView** sf2 = (**SurfaceView**) view2.findViewById(**R**.**id**.**sf2**);

**SurfaceView** sf3 = (**SurfaceView**) view3.findViewById(**R**.**id**.**sf3**);

views.add(view1); views.add(view2); views.add(view3);

holder1 = sf1.getHolder(); holder2 = sf2.getHolder(); holder3 = sf3.getHolder();

holder1.addCallback(**this**); holder2.addCallback(**this**); holder3.addCallback(**this**);

mViewPager.setAdapter(**new** MyAdapter());

}

/\*\*

//使用raw文件夹里的文件

**private** **void** **playerSetData**(**int** id){

**try** {

player.setDataSource(**MainActivity**.**this**, **Uri**.parse("android.resource://" + getPackageName() + "/" + id));

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

@Override

**public** **void** **surfaceCreated**(SurfaceHolder holder) {

...

playerSetData(**R**.**raw**.**yq1**);//使用raw文件夹里的文件

...

\*/

@Override

**public** **void** **surfaceCreated**(SurfaceHolder holder) {

player = **new** MediaPlayer();

player.setDisplay(holder); // 设置播放在surfaceview上

player.setOnPreparedListener(**this**);

player.setLooping(**true**);

**if** (holder1.equals(holder)) {

Path = **Environment**.getExternalStorageDirectory().getPath()+ "/yiquan1.mp4";

}

**if** (holder2.equals(holder)) {

Path = **Environment**.getExternalStorageDirectory().getPath()+ "/mv.avi";

}

**if** (holder3.equals(holder)) {

Path = **Environment**.getExternalStorageDirectory().getPath()+ "/yiquan3.mp4";

}

**try** {

player.setDataSource(Path); // 设置文件的路径

player.prepare(); // 准备播放

} **catch** (**IllegalArgumentException** e) {

e.printStackTrace();

} **catch** (**IllegalStateException** e) {

e.printStackTrace();

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

@Override

**public** **void** **surfaceChanged**(SurfaceHolder holder, **int** format, **int** width, **int** height) {

}

@Override

**public** **void** **surfaceDestroyed**(SurfaceHolder holder) {

}

@Override

**public** **void** **onPrepared**(**MediaPlayer** mp) {

mp.start(); //开始播放

}

**class** **MyAdapter** **extends** **PagerAdapter**{

@Override

**public** **int** **getCount**() {

**return** views.size();

}

@Override

**public** **boolean** **isViewFromObject**(**View** arg0, **Object** arg1) {

**return** arg0 == arg1;

}

//初始化position位置的界面

@Override

**public** **Object** **instantiateItem**(**ViewGroup** container, **int** position) {

container.addView(views.get(position));

**return** views.get(position);

}

@Override

**public** **void** **destroyItem**(**ViewGroup** container, **int** position, **Object** object) {

container.removeView(views.get(position));

}

}

}

# SurfaceView

## 概述

通常情况程序的View和用户响应都是在同一个线程中处理的。但是在其他线程中却不能修改UI元素是不允许的。如果需要在另外的线程绘制界面、需要迅速的更新界面或则渲染UI界面需要较长的时间，这种情况就要使用SurfaceView了。SurfaceView中包含一个Surface对象，而Surface是可以在后台线程中绘制的。Surface属于 OPhone底层显示系统。SurfaceView的性质决定了其比较适合一些场景：需要界面迅速更新、对帧率要求较高的情况。

使用SurfaceView需要注意以下几点情况： SurfaceView和SurfaceHolder.Callback函数都从当前SurfaceView窗口线程中调用（一般而言就是程序的主线程）。有关资源状态要注意和绘制线程之间的同步。 在绘制线程中必须先合法的获取Surface才能开始绘制内容，在SurfaceHolder.Callback.surfaceCreated() 和SurfaceHolder.Callback.surfaceDestroyed()之间的状态为合法的。使用SurfaceView 只要继承SurfaceView类并实现SurfaceHolder.Callback接口就可以实现一个自定义的SurfaceView了，SurfaceHolder.Callback在底层的Surface状态发生变化的时候通知View，

**SurfaceHolder.Callback具有如下的接口：**

surfaceCreated(SurfaceHolder holder)：当Surface第一次创建后会立即调用该函数。程序可以在该函数中做些和绘制界面相关的初始化工作，一般情况下都是在另外的线程来绘制界面，所以不要在这个函数中绘制Surface。

surfaceChanged(SurfaceHolder holder, int format, int width,int height)：当Surface的状态（大小和格式）发生变化的时候会调用该函数，在surfaceCreated调用后该函数至少会被调用一次。

surfaceDestroyed(SurfaceHolder holder)：当Surface被摧毁前会调用该函数，该函数被调用后就不能继续使用Surface了，一般在该函数中来清理使用的资源。

通过SurfaceView的getHolder()函数可以获取SurfaceHolder对象，Surface 就在SurfaceHolder对象内。虽然Surface保存了当前窗口的像素数据，但是在使用过程中是不直接和Surface打交道的，由SurfaceHolder的Canvas lockCanvas()或则Canvas lockCanvas(Rect dirty)函数来获取Canvas对象，通过在Canvas上绘制内容来修改Surface中的数据。如果Surface不可编辑或则尚未创建调用该函数会返回null，在 unlockCanvas() 和 lockCanvas()中Surface的内容是不缓存的，所以需要完全重绘Surface的内容，为了提高效率只重绘变化的部分则可以调用lockCanvas(Rect dirty)函数来指定一个dirty区域，这样该区域外的内容会缓存起来。在调用lockCanvas函数获取Canvas后，SurfaceView会获取Surface的一个同步锁直到调用unlockCanvasAndPost(Canvas canvas)函数才释放该锁，这里的同步机制保证在Surface绘制过程中不会被改变（被摧毁、修改）。当在Canvas中绘制完成后，调用函数unlockCanvasAndPost(Canvas canvas)来通知系统Surface已经绘制完成，这样系统会把绘制完的内容显示出来。

## 和MediaPlayer组合使用时

**public** **class** **MainActivity** **extends** **Activity** **implements** Callback, OnPreparedListener{

**private** **SurfaceView** surfaceView;

**private** SurfaceHolder holder;

**private** **MediaPlayer** player;

**private** **boolean** isplay;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

surfaceView = (**SurfaceView**) findViewById(**R**.**id**.**sf**);

holder = surfaceView.getHolder();

holder.addCallback(**this**);

}

@Override

**public** **void** **surfaceCreated**(SurfaceHolder holder) {

player = **new** MediaPlayer();

player.setDisplay(holder); // 设置播放在surfaceview上

player.setOnPreparedListener(**this**);

player.setLooping(**true**);

**try** {

player.setDataSource(**MainActivity**.**this**, **Uri**.parse("android.resource://"

+ getPackageName() + "/" + **R**.**raw**.**yq1**));

player.prepare();

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

@Override

**public** **void** **surfaceChanged**(SurfaceHolder holder, **int** format, **int** width, **int** height) {

}

@Override

**public** **void** **surfaceDestroyed**(SurfaceHolder holder) {

}

@Override

**public** **void** **onPrepared**(**MediaPlayer** mp) {

player.start();

isplay = **true**;

}

//视频停止播放

**private** **void** **mediaStop**(){

**if**(player!=**null**){

player.stop();//假如视频在播放，则停止

player.release();

player = **null**;

}

}

//视频暂停播放

**private** **void** **mediaPause**(){

**if**(player != **null**){

player.pause();//假如视频在播放，则暂停

isplay = **false**;

}

}

//视频继续播放

**private** **void** **mediaRe**(){

**if**(player != **null**){

player.start();//假如视频在播放，则暂停

isplay = **true**;

}

}

@Override

**public** **boolean** **onTouchEvent**(**MotionEvent** event) {

**super**.onTouchEvent(event);

**if** (event.getAction() == **MotionEvent**.**ACTION\_UP**) {

**if** (isplay) {

mediaPause();

}**else** {

mediaRe();

}

}

**return** **true**;

}

@Override

**protected** **void** **onDestroy**() {

mediaStop();

**super**.onDestroy();

}

}

# 录音机工具类

**public** **class** **RecordUtil** {

**private** **static** **final** **int** **SAMPLE\_RATE\_IN\_HZ** = 8000;

**private** **MediaRecorder** recorder = **new** MediaRecorder();

**private** **String** mPath;

**public** **RecordUtil**(**String** path) {

mPath = path;

}

**public** **void** **start**() **throws** IOException {

**String** state = android.os.**Environment**.*getExternalStorageState*();

**if** (!state.equals(android.os.**Environment**.**MEDIA\_MOUNTED**)) {

**throw** **new** IOException("SD Card is not mounted,It is " + state + ".");

}

**File** directory = **new** File(mPath).getParentFile();

**if** (!directory.exists() && !directory.mkdirs()) {

**throw** **new** IOException("Path to file could not be created");

}

recorder.setAudioSource(**MediaRecorder**.**AudioSource**.**MIC**);

recorder.setOutputFormat(**MediaRecorder**.**OutputFormat**.**RAW\_AMR**);

recorder.setAudioEncoder(**MediaRecorder**.**AudioEncoder**.**AMR\_NB**);

recorder.setAudioSamplingRate(**SAMPLE\_RATE\_IN\_HZ**);

recorder.setOutputFile(mPath);

recorder.prepare();

recorder.start();

}

**public** **void** **stop**() **throws** IOException {

recorder.stop();

recorder.release();

}

**public** **double** **getAmplitude**() {//获取音量

**if** (recorder != **null**) {

**return** (recorder.getMaxAmplitude());

}

**return** 0;

}

}

**public** **class** **MainActivity** **extends** **Activity** {

**private** **RecordUtil** util;

**private** **String** path;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

path = **Environment**.*getExternalStorageDirectory*().getPath() + "/cgyrecord/temp.mp3";;

util = **new** RecordUtil(path);

((**TextView**) findViewById(**R**.**id**.**aaa**)).setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** **onClick**(**View** v) {

**try** {

util.start();

} **catch** (**IOException** e) {

e.printStackTrace();

}

**Toast**.*makeText*(**MainActivity**.**this**, "录音开始哈哈", **Toast**.**LENGTH\_SHORT**).show();

}

});

((**TextView**) findViewById(**R**.**id**.**bbb**)).setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** **onClick**(**View** v) {

**try** {

util.stop();

} **catch** (**IOException** e) {

e.printStackTrace();

}

**Toast**.*makeText*(**MainActivity**.**this**, "录音结束了", **Toast**.**LENGTH\_SHORT**).show();

}

});

}

}

**Manifest.xml权限：**

<uses-permission android:name=*"android.permission.WRITE\_EXTERNAL\_STORAGE"* />

<uses-permission android:name=*"android.permission.RECORD\_AUDIO"* />

# Json解析

## 传统解析

**String** jsonStr ="{\"username\":\"张木生\",\"age\":\"20\",\"jicheng\":[{\"zhengshu\":\"PMP\",\"date\":\"2011年\"},{\"zhengshu\":\"信息系统项目管理师\",\"date\":\"2012年\"},{\"zhengshu\":\"程序员\",\"date\":\"2013年\"},{\"zhengshu\":\"测试人员\",\"date\":\"2014年\"}]}";

list = **new** ArrayList<**Info**>();

**try** {

**JSONObject** jsonObject = **new** JSONObject(a);

name = jsonObject.getString("username");

age = jsonObject.getString("age");

addr = jsonObject.getString("addr");

**JSONArray** jsonArray = jsonObject.getJSONArray("jicheng");

**for** (**int** i = 0; i < jsonArray.length(); i++) {

**JSONObject** jsonObject2 = jsonArray.getJSONObject(i);

**Info** info = **new** Info();

info.setzhengshu(jsonObject2.getString("zhengshu"));

info.setdate(jsonObject2.getString("date"));

list.add(info);

}

} **catch** (**JSONException** e) {

e.printStackTrace();

}

## 使用GSON库解析单纯jsonobject

**Gson** gson = **new** Gson();

**Person** p = gson.fromJson("{\"name\":\"gagafa\",\"age\":20, \"sex\":true}", **Person**.**class**);

## 使用GSON库解析单纯jsonarray

**Gson** gson = **new** Gson();

**String** jsonString = "[{\"name\":\"gagafa\",\"age\":20, \"sex\":true}, " +

"{\"name\":\"gagafdfa\",\"age\":2540, \"sex\":true}, " +

"{\"name\":\"gagfdafa\",\"age\":240, \"sex\":false}]";

List<**Person**> persons = gson.fromJson(jsonString,**new** TypeToken<List<**Person**>>(){}.getType());

## 解析有标签的object（用内部类）

**String** js = "{\"weatherinfo\":{\"city\":\"北京\",\"cityid\":\"101010100\",\"temp\":\"10\"}}";

**Weather** w = gson.fromJson(js, **Weather**.**class**);

**public** **class** **Weather** {

**private** **weatherinfo** weatherinfo;

**public** **Weather**.**weatherinfo** **getWeatherinfo**() {

**return** weatherinfo;

}

**public** **void** **setWeatherinfo**(**Weather**.**weatherinfo** weatherinfo) {

**this**.weatherinfo = weatherinfo;

}

**public** **class** **weatherinfo**{

**private** **String** city;

**private** **String** cityid;

**private** **String** temp;

**public** **String** **getCity**() {

**return** city;

}

**public** **void** **setCity**(**String** city) {

**this**.city = city;

}

**public** **String** **getCityid**() {

**return** cityid;

}

**public** **void** **setCityid**(**String** cityid) {

**this**.cityid = cityid;

}

**public** **String** **getTemp**() {

**return** temp;

}

**public** **void** **setTemp**(**String** temp) {

**this**.temp = temp;

}

@Override

**public** **String** **toString**() {

**return** "weatherinfo [city=" + city + ", cityid=" + cityid + ", temp=" + temp + "]";

}

}

@Override

**public** **String** **toString**() {

**return** "Weather [weatherinfo=" + weatherinfo.toString() + "]";

}

}

## 解析有标签的Array

**String** jsonStr ="{\"username\":\"张木生\",\"age\":\"20\",\"jicheng\":[{\"zhengshu\":\"PMP\",\"date\":\"2011年\"},{\"zhengshu\":\"信息系统项目管理师\",\"date\":\"2012年\"},{\"zhengshu\":\"程序员\",\"date\":\"2013年\"},{\"zhengshu\":\"测试人员\",\"date\":\"2014年\"}]}";

**Type** type = **new** TypeToken<**MyPeople**>(){}.getType();

**MyPeople** myPeople = gson.fromJson(jsonStr, type);

**Log**.e("tag", myPeople.toString());

E/tag: MyPeople{username='张木生', age='20', jicheng=[JiCheng{zhengshu='PMP', date='2011年'}, JiCheng{zhengshu='信息系统项目管理师', date='2012年'}, JiCheng{zhengshu='程序员', date='2013年'}, JiCheng{zhengshu='测试人员', date='2014年'}]}

**public** **class** **MyPeople** {

**private** **String** username;

**private** **String** age;

**private** List<**JiCheng**> jicheng;

**public** **String** **getUsername**() {

**return** username;

}

**public** **void** **setUsername**(**String** username) {

**this**.username = username;

}

**public** **String** **getAge**() {

**return** age;

}

**public** **void** **setAge**(**String** age) {

**this**.age = age;

}

**public** List<**JiCheng**> **getJicheng**() {

**return** jicheng;

}

**public** **void** **setJicheng**(List<**JiCheng**> jicheng) {

**this**.jicheng = jicheng;

}

**public** **class** **JiCheng** {

**private** **String** zhengshu;

**private** **String** date;

**public** **String** **getZhengshu**() {

**return** zhengshu;

}

**public** **void** **setZhengshu**(**String** zhengshu) {

**this**.zhengshu = zhengshu;

}

**public** **String** **getDate**() {

**return** date;

}

**public** **void** **setDate**(**String** date) {

**this**.date = date;

}

@Override

**public** **String** **toString**() {

**return** "JiCheng{" + "zhengshu='" + zhengshu + '\'' + ", date='" + date + '\'' + '}';

}

}

@Override

**public** **String** **toString**() {

**return** "MyPeople{" + "username='" + username + '\'' + ", age='" + age + '\'' + ", jicheng=" + jicheng.toString() + '}';

}

}

## 类转json格式字符串

**PrintInfoBean** bean = **new** PrintInfoBean();

bean.setTitle("移动充值");

List<**String**> contentlist = **new** ArrayList<**String**>();

contentlist.add("----------\*\*\*\*\*\*\*\*\*\*\*\*-------");

contentlist.add("操作员：001");

contentlist.add("充值号码：18888888888");

contentlist.add("充值金额：50元");

contentlist.add("充值时间：2016-05-01");

bean.setContentlist(contentlist);

**Gson** gson = **new** Gson();

**String** data = gson.toJson(bean, **PrintInfoBean**.**class**);

/\*{"contentlist":["----------\*\*\*\*\*\*\*\*\*\*\*\*-------","操作员：001","充值号码：18888888888","充值金额：50元","充值时间：2016-05-01"],"title":"移动充值"}\*/

**public** **class** **PrintInfoBean** {

**private** **String** title;

**private** List<**String**> contentlist;

**public** **String** **getTitle**() {

**return** title;

}

**public** **void** **setTitle**(**String** title) {

**this**.title = title;

}

**public** List<**String**> **getContentlist**() {

**return** contentlist;

}

**public** **void** **setContentlist**(List<**String**> contentlist) {

**this**.contentlist = contentlist;

}

}

## 大类解析

**public** **class** **Data**{

**private** **String** terminal\_no;

**private** **String** store\_name;

**private** List<**Paymethodlist**> paymethodlist;

**public** **String** **getTerminal\_no**() {

**return** terminal\_no;

}

**public** **void** **setTerminal\_no**(**String** terminal\_no) {

**this**.terminal\_no = terminal\_no;

}

**public** **String** **getStore\_name**() {

**return** store\_name;

}

**public** **void** **setStore\_name**(**String** store\_name) {

**this**.store\_name = store\_name;

}

**public** List<**Paymethodlist**> **getPaymethodlist**() {

**return** paymethodlist;

}

**public** **void** **setPaymethodlist**(List<**Paymethodlist**> paymethodlist) {

**this**.paymethodlist = paymethodlist;

}

}

**public** **class** **Paymethodlist** {

**private** **String** code;

**private** **String** name;

**public** **String** **getCode**() {

**return** code;

}

**public** **void** **setCode**(**String** code) {

**this**.code = code;

}

**public** **String** **getName**() {

**return** name;

}

**public** **void** **setName**(**String** name) {

**this**.name = name;

}

}

com.cgy.gdwsdemo.bean.dosettlement.**Data** data = **new** com.cgy.gdwsdemo.bean.dosettlement.Data();

data.setStore\_name("助民通便民服务代办点");

data.setTerminal\_no("800020");

List<**Paymethodlist**> paymethodlists = **new** ArrayList<>();

**Paymethodlist** paymethodlist1 = **new** Paymethodlist();

**Paymethodlist** paymethodlist2 = **new** Paymethodlist();

paymethodlist1.setCode("3");

paymethodlist1.setName("微信");

paymethodlist2.setCode("2");

paymethodlist2.setName("支付宝");

paymethodlists.add(paymethodlist1);

paymethodlists.add(paymethodlist2);

data.setPaymethodlist(paymethodlists);

**String** str = gson.toJson(data, com.cgy.gdwsdemo.bean.dosettlement.**Data**.**class**);

**Log**.*e*("tag\_json", str);

**return** str;

{"paymethodlist":[{"code":"3","name":"微信"},{"code":"2","name":"支付宝"}],"store\_name":"助民通便民服务代办点","terminal\_no":"800020"}

## 解析某些特定符号会转成Unicode

比如等号：new Gson().toJson(http://www.baidu.com/id=1),结果却是："http://www.baidu.com/id\u003d1"

解决办法：GsonBuilder().disableHtmlEscaping().create().toJson(http://www.baidu.com/id=1),得到正确结果:"http://www.baidu.com/id=1"

# XML解析

**public** **class** **PullBookParser** {

/\*\*

\* 1、事件回调类型

\* a.读取到XML文档开头（声明）返回： XmlPullParser.START\_DOCUMENT（0）

\* b.读取到XML文档结束返回： XmlPullParser.END\_DOCUMENT （1）

\* c.读取到XML节点开始返回： XmlPullParser.START\_TAG （2）

\* d.读取到XML节点结束返回： XmlPullParser.END\_TAG （3）

\* e.读取到XML文本返回： XmlPullParser.TEXT （4）

\*

\* 2、XmlPullParser有几个主要方法:

\* a.XmlPullParser.getEventType() : 【获取当前事件回调类型】

\* b.XmlPullParser.getName():【获取当前节点名字】

\* c.XmlPullParser.getAttributeValue(int index):【根据id获取节点属性值】

\* d.XmlPullParser.getAttributeValue(String namespace, String name):【根据name获取节点属性值】

\* e.XmlPullParser.netxText():【回调节点START\_TAG时，通过此方法获取节点内容】

\*/

**public** List<**Book**> **parse**(**InputStream** in) **throws** Exception {

List<**Book**> books = **null**;

**Book** book = **null**;

//由android.util.Xml创建一个XmlPullParser实例

XmlPullParser parser = **Xml**.newPullParser();

//设置输入流 并指明编码方式

parser.setInput(in, "UTF-8");

**int** eventType = parser.getEventType();

**while** (eventType != XmlPullParser.**END\_DOCUMENT**) {

**switch** (eventType) {

**case** XmlPullParser.**START\_DOCUMENT**:

books = **new** ArrayList<**Book**>();//读到开头实例化list；

**break**;

**case** XmlPullParser.**START\_TAG**:

**String** tagName = parser.getName();

**if** (tagName.equals("book")) {

book = **new** Book();//读到节点实例化book；

//book.setId(Integer.parseInt(parser.getAttributeValue(0)));

}**else** **if** (tagName.equals("id")) {

book.setId(**Integer**.parseInt(parser.nextText()));

}**else** **if** (tagName.equals("name")) {

book.setName(parser.nextText());

} **else** **if** (tagName.equals("price")) {

book.setPrice(**Float**.parseFloat(parser.nextText()));

}

**break**;

**case** XmlPullParser.**END\_TAG**:

**if** (parser.getName().equals("book")) {

books.add(book);

book = **null**;

}

**break**;

**default**:

**break**;

}

eventType = parser.next();

}

**return** books;

}

//写成xml

**public** **String** **serialize**(List<**Book**> books) {

**try** {

**File** file = **new** File(**Environment**.getExternalStorageDirectory(), "book2.xml");

XmlSerializer serializer = **Xml**.newSerializer();

serializer.setOutput(**new** FileOutputStream(file), "UTF-8");

serializer.startDocument("UTF-8", **true**);

serializer.startTag("", "books");

**for** (**Book** book : books) {

serializer.startTag("", "book");

serializer.attribute("", "id", book.getId() + "");

serializer.startTag("", "name");

serializer.text(book.getName());

serializer.endTag("", "name");

serializer.startTag("", "price");

serializer.text(book.getPrice() + "");

serializer.endTag("", "price");

serializer.endTag("", "book");

}

serializer.endTag("", "books");

serializer.endDocument();

**return** "OK";

}**catch** (**Exception** e){

e.printStackTrace();

**return** "NO";

}

}

}

## 解析XML的几种方式的原理与特点：DOM、SAX、PULL

* DOM：消耗内存：先把xml文档都读到内存中，然后再用DOM API来访问树形结构，并获取数据。这个写起来很简单，但是很消耗内存。要是数据过大，手机不够牛逼，可能手机直接死机
* SAX：解析效率高，占用内存少，基于事件驱动的：更加简单地说就是对文档进行顺序扫描，当扫描到文档(document)开始与结束、元素(element)开始与结束、文档(document)结束等地方时通知事件处理函数，由事件处理函数做相应动作，然后继续同样的扫描，直至文档结束。
* PULL：与 SAX类似，也是基于事件驱动，我们可以调用它的next（）方法，来获取下一个解析事件（就是开始文档，结束文档，开始标签，结束标签），当处于某个元素时可以调用XmlPullParser的getAttributte()方法来获取属性的值，也可调用它的nextText()获取本节点的值。

# 压缩与解压

## Gzip压缩与解压

gzip压缩文件的后缀为.gz，gzip除了是一种压缩文件之外，还常常被用在网络传输中进行数据压缩。在Java中，GZIP针对的是单文件的压缩与解压。如果想实现多文件压缩，需要先将文件压缩成tar，在将tar压缩为gz，这里就不说了。

### Gzip压缩

Gzip压缩对应的是GZIPOutputStream，我们只需要将普通的输出流包装成GZIPOutputStream，就可以了。不过，需要注意的是在使用完毕一定要关闭 一定要关闭，否则你去解压的时候可是会装载异常的，那么我们看下代码：

**private** **void** **compress**() {

**String** path = **null**;

**GZIPOutputStream** gzipOutputStream = **null**;

**if** (**Environment**.**MEDIA\_MOUNTED**.equals(**Environment**.*getExternalStorageState*())) {

path = **Environment**.*getExternalStorageDirectory*() + "/";

}

**try** {

**File** file = **new** File(path ,"text.gz");

**if** (file.exists()) {

file.delete();

}

file.createNewFile();

**byte**[] by = **new** String("测试gzip压缩").getBytes("utf-8");

gzipOutputStream = **new** GZIPOutputStream(**new** FileOutputStream(file));

gzipOutputStream.write(by);

} **catch** (**IOException** e) {

e.printStackTrace();

}**finally** {

**if** (gzipOutputStream != **null**) {

**try** {

gzipOutputStream.close();

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

}

}

### Gzip解压

同样解压就是包装一下普通流，直接上代码。

**private** **String** **uncompress**() {

**String** path = **null**;

**BufferedReader** reader = **null**;

**StringBuilder** builder = **new** StringBuilder();

**if** (**Environment**.**MEDIA\_MOUNTED**.equals(**Environment**.*getExternalStorageState*())) {

path = **Environment**.*getExternalStorageDirectory*() + "/";

**try** {

**File** file = **new** File(path ,"text.gz");

**GZIPInputStream** gzipInputStream = **new** GZIPInputStream(**new** FileInputStream(file));

reader = **new** BufferedReader(**new** InputStreamReader(gzipInputStream));

**String** line = **null**;

**while** ((line = reader.readLine()) != **null**){

builder.append(line + "\r\n");

}

} **catch** (**IOException** e) {

e.printStackTrace();

}**finally** {

**if** (reader != **null**) {

**try** {

reader.close();

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

}

}

**return** builder.toString();

}

## Zip压缩与解压工具类ZipUtil

虽然说java原生api中gzip是对单文件(流)进行操作的，可是我们的zip却能对多文件进行操作。并且Zip支持两种Checksum类来计算和校验文件的校验和方法，支持两种：

* Adler32 速度稍微快点
* CRC32 速度慢，但是更准确

zip的压缩，每一个文件对应一个ZipEntey，然后输出数据即可；zip解压，只需要通过ZipInputStream.getNextEntry的方法把每一个文件读出来即可

**public** **class** **ZipUtil** {

/\*\*

\* 压缩ZIP方法

\* **@param** src 需要压缩的文件或文件夹的路径

\* **@param** dest 压缩完毕后生成的zip文件

\*/

**public** **static** **void** **zip**(**String** srcFilePath, **String** outFilePath) **throws** IOException {

**ZipOutputStream** out = **null**;

**try** {

**File** outFile = **new** File(outFilePath);//压缩完毕后生成的zip文件

out = **new** ZipOutputStream(**new** FileOutputStream(outFile));

**File** fileOrDirectory = **new** File(srcFilePath);//需要压缩的文件或文件夹的路径

*zipFileOrDirectory*(out, fileOrDirectory, "");//压缩的细节，使用了递归

} **catch** (**IOException** e) {

e.printStackTrace();

} **finally** {

**if** (out != **null**) {

**try** {

out.close();

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

}

}

/\*\*

\* 压缩文件或文件夹具体方法

\* **@param** out zip输出流

\* **@param** fileOrDirectory 需要压缩的文件或文件夹

\* **@param** curPath 在压缩文件中的哪个文件夹中

\*/

**private** **static** **void** **zipFileOrDirectory**(**ZipOutputStream** out, **File** fileOrDirectory, **String** curPath) **throws** IOException {

**FileInputStream** in = **null**; //从文件中读取字节的输入流

**try** {

**if** (fileOrDirectory.isFile()) {//如果是文件

**ZipEntry** entry = **new** ZipEntry(curPath + fileOrDirectory.getName());//实例代表一个条目内的ZIP归档

out.putNextEntry(entry);//条目的信息写入底层流

**byte**[] buffer = **new** **byte**[4096];

**int** bytes\_read;

in = **new** FileInputStream(fileOrDirectory);

**while** ((bytes\_read = in.read(buffer)) != -1) {

out.write(buffer, 0, bytes\_read);

}

out.closeEntry();

} **else** {//如果是文件夹

**File**[] entries = fileOrDirectory.listFiles();

**for** (**int** i = 0; i < entries.length; i++) {

*zipFileOrDirectory*(out, entries[i], curPath + fileOrDirectory.getName() + "/");// 递归压缩，更新curPaths

}

}

} **catch** (**IOException** e) {

e.printStackTrace();

} **finally** {

**if** (in != **null**) {

**try** {

in.close();

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

}

}

/\*\*

\* 解压Zip方法

\* **@param** zipFileName 被解压的zip文件的路径

\* **@param** outputDirectory 输出文件夹路径

\*/

**public** **static** **void** **unzip**(**String** zipFilePath, **String** outputDirectory) **throws** IOException {

**ZipFile** zipFile = **null**;

**try** {

zipFile = **new** ZipFile(zipFilePath);//被解压的zip文件的路径

Enumeration e = zipFile.entries();//压缩文件们的实例

**ZipEntry** zipEntry = **null**;

**File** dest = **new** File(outputDirectory);//解压到xx文件夹

dest.mkdirs();//创建这个路径

**while** (e.hasMoreElements()) {

zipEntry = (**ZipEntry**) e.nextElement();

**String** entryName = zipEntry.getName();

**InputStream** in = **null**;

**FileOutputStream** out = **null**;

**try** {

**if** (zipEntry.isDirectory()) {//如果是文件夹

**String** name = zipEntry.getName();

name = name.substring(0, name.length() - 1);//截取原字符串少一位，少了"/"

**File** f = **new** File(outputDirectory + **File**.**separator** + name);//创建子级文件夹

f.mkdirs();

} **else** {//如果是文件

**int** index = entryName.lastIndexOf("\\");//如果出现"\\",那么他在子级文件夹下，继续创建子级文件夹

**if** (index != -1) {

**File** df = **new** File(outputDirectory + **File**.**separator** + entryName.substring(0, index));

df.mkdirs();

}

index = entryName.lastIndexOf("/");//如果出现"/",那么他在子级文件夹下，继续创建子级文件夹

**if** (index != -1) {

**File** df = **new** File(outputDirectory + **File**.**separator** + entryName.substring(0, index));

df.mkdirs();

}

//上面都是在创建文件夹，现在开始解压文件了

**File** f = **new** File(outputDirectory + **File**.**separator** + zipEntry.getName());//解压出来的的文件

in = zipFile.getInputStream(zipEntry);

out = **new** FileOutputStream(f);

**int** c;

**byte**[] by = **new** **byte**[1024];

**while** ((c = in.read(by)) != -1) {

out.write(by, 0, c);

}

out.flush();

}

} **catch** (**IOException** ex) {

**throw** **new** IOException("解压失败：" + ex.toString());

} **finally** {

**if** (in != **null**) {

**try** {

in.close();

} **catch** (**IOException** ex) {

ex.printStackTrace();

}

}

**if** (out != **null**) {

**try** {

out.close();

} **catch** (**IOException** ex) {

ex.printStackTrace();

}

}

}

}

} **catch** (**IOException** e) {

**throw** **new** IOException("解压失败：" + e.toString());

} **finally** {

**if** (zipFile != **null**) {

**try** {

zipFile.close();

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

}

}

}