# TODO

四大组件之一Service什么时候会回收，是AMS负责回收，还是虚拟机，还是内核：不会回收，和进程的生命周期一样，有内核

虚拟机的gc和内核回收差异：无关，各自不同的机制

Ams哪些时候可以杀进程，源码位置：后台不会主动杀，由内核控制

低内存是指哪一段内存，虚拟机低

每个应用app最大申请内存是多少，

App触发低内存了，是否一定触发内核oom

怎么去模拟低内存的情况

IdleHandler原理

onLowMemory() 回调的原理

requestConcurrentGC()如何调用到addGcWatcher的？虚拟机弱引用机制

intent.setFlags(Intent.***FLAG\_ACTIVITY\_CLEAR\_TOP***);

运行原理

解读Android进程优先级ADJ算法

<http://gityuan.com/2018/05/19/android-process-adj/>

内存管理，lowmemorykill.c

<http://www.voidcn.com/article/p-pzlqoclp-bkg.html>

<http://gityuan.com/2016/09/17/android-lowmemorykiller/>

<https://cloud.tencent.com/developer/article/1029670>

android 关于提高第三方app的service优先级

<https://www.cnblogs.com/yxwkf/p/5363486.html>

<https://developer.android.com/guide/components/processes-and-threads>

关于Android7.1.1后不能隐藏Foreground Service的Notification的分析

<https://www.jianshu.com/p/0929c4012347>

<https://blog.csdn.net/cscs111/article/details/77558168>原理

Activity displayed延迟）

<https://blog.csdn.net/kc58236582/article/details/60134836>

windowsDrawnLocked关闭关机动画的

语言切换

fW

<https://blog.csdn.net/zhaokai621/article/details/77102308?utm_source=blogxgwz10>

app

<https://blog.csdn.net/cekiasoo/article/details/54933135?utm_source=blogxgwz2>

# 概述

## Activity的生命周期和启动模式

### ActivityState定义

**enum** ActivityState {  
 INITIALIZING,  
 RESUMED,  
 PAUSING,  
 PAUSED,  
 STOPPING,  
 STOPPED,  
 FINISHING,  
 DESTROYING,  
 DESTROYED  
}

启动：onCreate -> onStart -> onResume  
退出：onPause -> onStop -> onDestroy  
后台：onPause -> onStop  
返回：onRestart -> onStart -> onResume  
异常情况：  
onSaveInstanceState -> onStop -> onDestroy 和onPause无时序关系  
onCreate -> onStart -> onRestoreInstanceState 和onResume无时序关系

### onStart & onStop 与 onResume & onDestroy区别：

onStart & onStop：表示activity是否可见  
onResume & onPause：表示activity是否位于前台

### 耗时操作

A启动B,A必须先onPause，B才可onResume。所以onPause不要执行重量级操作(耗时操作)，应把重量级操作放到onStop中。

### 1 Activity的生命周期全面分析

正常周期（用户参与）和异常周期（系统回收、Configuration改变）

**正常周期**



总结一下各种情况下Activity生命周期的调用情况。

**1，正常打开单个Activity，然后退出应用：**

这种情况是最普通的状况，Activity的生命周期会按照上图从上到下的方式走。即：onCreate --> onStart --> onResume --> 运行--> 按返回键结束程序--> onPause-->onStop-->onDestory

**2，打开一个Activity A，然后再打开另一个Activity B**

对于A：

onCreate --> onStart --> onResume --> A运行 --> A发出打开B的Intent --> onPause-->B可见-->onStop

此时，会打开B，B同样会经历一个完整的Activity生命周期。

等B结束，A再度可见的时候，A会经历：onRestart-->onStart-->onResume

注意，B这个Activity是在A的onPause执行后才变成可见状态的，所以为了不影响B的显示，最好不要在onPause里执行一些耗时操作，可以考虑将这些操作放到onStop里，这时B已经可见了。

**3，打开一个Activity A，然后再打开另一个Activity B（透明主题）或者是对话框B**

对于A：

onCreate --> onStart --> onResume --> A运行 --> A发出打开B的Intent --> onPause-->B可见--~~>onStop~~

此时，会打开B，B同样会经历一个完整的Activity生命周期。

等B结束，A再度可见的时候，A会经历：~~onRestart-->onStart-->~~onResume

**3，优先级低的Activity被系统回收**

Activity优先级从高到低大致有三种：

1）刚执行onResume，前台Activity。可见的前台Activity优先级最高

2）刚执行onPause，可见，但非前台的Activity。例如弹出了一个对话框，Activity可见，但是不可与用户交互。此类别优先级低一些。

3）刚执行onStop，后台Activity。这个就是不可见的Activity。

系统资源不足时，会优先回收优先级低的Activity。这个时候被结束的Activity，并不会像正常的Activity一样，在销毁的时候会走onPause，onStop，onDestory这样的生命周期，而是会多一个onSaveInstanceState的方法来保存一些数据。这个方法会在onStop前调用，但是不确定是在onPause之前还是之后。Activity重建的时候，这个方法中所保存的数据会以Bundle的形式作为参数传递给onCreate，所以我们可以用参数来判断这个Activity是新建的还是重建的，如果是重建的，则可以从Bundle中取出数据来恢复界面。

**QA：**

**Q1**: onStart和onResume, onPause和onStop从描述上来看差不多，对我们来说有什么实质的不同呢?

**A1：**从整个生命周期来说，onCreate和onDestroy是配对的，分别标识着Activity的创建和销毁，并且只可能有一次调用。从Activity是否可见来说，onStart和onStop是配对的（**可见性状态**），随着用户的操作或者设备屏幕的点亮和熄灭，这两个方法可能被调用多次;从Activity是否在前台来说，onResume和onPause是配对的（**前后台状态**），随着用户操作或者设备屏幕的点亮和熄灭，这两个方法可能被调用多次。

从实际使用过程来说，onStart和anResume. onPause和onStop看起来的确差不多，甚至我们可以只保留其中一对，比如只保留onStart和onStop 既然如此，那Android系统还要提供看起来重复的接口呢?这两个配对的回调分别表示不同的意义，onStart和onStop是从Activity是否可见这个角度来回调的，而onResume和onPause。是从Activity是否位于前台这个角度来回调的，除了这种区别，在实际使用中没有其他明显区别。

**Q2:**假设当前Activity为A，如果这时用户打开一个新Activity B，那么B的onResume和A的onPause哪个先执行呢?

**A2：**onPause先执行-》onPause和onStop都不能执行耗时的操作，尤其是onPause，应当尽量在onStop中做操作，从而使得新Activity尽快显示出来并切换到前台。

**A3：**了解这些生命周期

**Q3：**自定义与视图有关的工具的时候，也应该有这些生命周期，比如动画工具

**异常周期**

**系统配置**

在默认情况下，如果我们的Activity不做特殊处理，那么当系统配置发生改变后，Activity就会被销毁并重新创建，其生命周期如图1-3所示。系统只在Activity异常终止的时候才会调用onSaveInstanceState和onRestoreInstanceState来存储和恢复数据，其他情况不会触发这个过程。



当系统配置发生改变后，系统会调用onSaveInstanceState来保存当前Activity的状态。这个方法的调用时机是在onStop之前，它和onPause没有既定的时序关系，可前可后。这个方法只会出现在Activity被异常终止的情况下，正常情况下系统不会回调。当Activity被重新创建后，系统会调用onRestorelnstanceSiate，从时序来说，onRestoreInstanceState的调用时机在onStart之后。

Activity被销毁并重新创建后，我们再去获取之前存储的字符串。接收的位置可以选择onRestorelnstanceState或者onCreate，二者的区别onRestorelnstanceState参数非空，但是onCreate可空，onCreate如果是正常启动的话，其参数Bundle saVedlnstanceState为null,

所以必须要额外判断。这两个方法我们选择任意一个都可以进行数据恢复，但官方建议是采用onRestoreInstanceState去恢复数据

系统配置监听，取代onSaveInstanceState和onRestorelnstanceState

android:configChanges="orientation|screenSize"

@Override

public void onConfigurationChanged(Configuration newConfig) {

super.onConfigurationChanged(newConfig);

Log.d(TAG, "onConfigurationChanged, newOrientation:" + newConfig.orientation);

}

情况2:**资源内存**不足导致低优先级的Activity被杀死

这种情况我们不好模拟，但是其数据存储和恢复过程和情况1完全一致。这里我们描述一下Activity的优先级情况口Activity按照优先级从高到低，可以分为如下3种:

(1)前台Activity—正在和用户交互的Activity，优先级最高。

(2)可见但非前台Activity—比如Activity中弹出了一个对话框，导致Activity可见，但是位于后台无法和用户直接交互。

（3)后台Activity—已经被暂停的Activity，比如执行了onStop，优先级最低。

**小结**：如果一个进程中没有四大组件在执行，那么这个进程将很快被系统杀死，因此，一此后台工作不适合脱离四大组件而独自运行在后台中，这样进程很容易被杀死。比较好的方法是将后台工作放入Service中从而保证进程有一定的优先级，这样就不会轻易地被系统杀死。

### 2.Activity的LaunchMode

**引言：**

* 为什么需要启动模式（创建Activity实例的方式）：由于android存在**taskStack特性**和java的**class单例特性**，从而产生了不同的启动模式
* why we need taskStack：因为是android 只能运行单窗口app，startActivity(intent)入栈push，finish()pop
* taskStack（1）<->Activity Insurance（N）

a. standard 。标准模式，1栈N例，系统默认，每次启动会重新创建新的实例，谁启动了这个Activity，这个Activity就在谁的栈里。

正确使用场景：适用于绝大大数情况，ABCDEFC；

错误使用场景，用ApplicationContext（自身没有taskStack），去启动Standard模式的Activity会报错,所以对画框会报错

b. singleTop 栈顶复用，单例栈顶模式，1栈top1例。该Activity的onNewIntent方法会被回调，onCreate和onStart并不会被调用。

正确使用场景：A例在top模式下生效，答题；ABC

错误使用场景：B例在非top下回退为standard 模式，ABCB

c. singleTask 栈内复用，单例栈内模式栈内复用模式，1栈1例。只要该Activity在一个栈中存在，都不会重新创建，onNewIntent会被回调。如果不存在，系统会先寻找是否存在需要的栈，如果不存在该栈，就创建一个任务栈，然后把这个Activity放进去；如果存在，就会创建到已经存在的这个栈中。**FLAG\_ACTIVITY\_CLEAR\_TOP，**<http://www.2cto.com/kf/201410/347854.html>

|  |
| --- |
| android:taskAffinity="com.other" |

Manifest 配置文件这个属性的意思是 “task 空间”=== “任务空间”，系统默认给设置为当前应用的包名。我们这里设置 的和包名不一样就会新建一个taskStack

<http://www.cnblogs.com/CSU-PL/p/3794280.html>



正确使用场景：1.根据affinity重新为Activity选择宿主task（与allowTaskReparenting属性配合工作）；2.启动一个Activity过程中Intent使用了FLAG\_ACTIVITY\_NEW\_TASK标记，根据affinity查找或创建一个新的具有对应affinity的task。我们会在后面进行详细讲解。览器activity， 整个任务栈只有一个实例，节约内存和cpu的目的

错误使用场景：B例在非top下回退为standard 模式，ABCB

d. singleInstance。单例模式具有此种模式的Activity只能单独存在于一个任务栈。

 应用场景：呼叫来电界面 InCallScreen

**Q&A:什么是Activity所需要的任务栈呢:由TaskAffinity确定，默认为应用包名。TaskAffinity属性主要和singleTask启动模式或者allowTaskReparenting属性配对使用，在其他情况下没有意义。另外，任务栈分为前台任务栈和后台任务栈，后台任务栈中的Activity位于暂停状态，用户可以通过切换将后台任务栈再次调到前台。**

当TaskAffinity和singleTask启动模式配对使用法，它是具有该模式的待启动的Activity会运行在任务栈TaskAffinity中。

当TaskAffinity和**allowTaskReparenting配对使用，**这个属性用来标记一个Activity实例在当前应用退居后台后，是否能从启动它的那个task移动到有共同affinity的task，“true”表示可以移动，“false”表示它必须呆在当前应用的task中，默认值为false。例如在一个应用中要查看一个web页面，在启动系统浏览器Activity后，这个Activity实例和当前应用处于同一个task，当我们的应用退居后台之后用户再次从主选单中启动应用，此时这个Activity实例将会重新宿主到Browser应用的task内，在我们的应用中将不会再看到这个Activity实例，而如果此时启动Browser应用，就会发现，第一个界面就是我们刚才打开的web页面，证明了这个Activity实例确实是宿主到了Browser应用的task内。我们就来结合实例演示一下这个过程：

<http://www.cnblogs.com/CSU-PL/p/3794280.html>

**“拿来主义”standard模式。哪里需要调用我我就去哪里，可以多次实例化，可以几个相同的Activity重叠。一般默认模式**

**“拒绝堆叠”singleTop模式。可以多次实例化，但是不可以多个相同的Activity重叠，当堆栈的顶部为相同的Activity时，会调用onNewIntent函数。可以用于图片浏览**

**“独立门户”singleTask模式。同一个应用中调用该Activity时，如果该Activity没有被实例化，会在本应用程序的Task内实例 化，如果已经实例化，会将Task中其上的Activity销毁后，调用onNewIntent；其它应用程序调用该Activity时，如果该 Activity没有被实例化，会创建新的Task并实例化后入栈，如果已经实例化，会销毁其上的Activity，并调用onNewIntent。一句 话，singleTask就是“独立门户”，在自己的Task里，并且启动时不允许其他Activity凌驾于自己之上。可以用浏览器-**微信

**“孤独寂寞”singleInstance模式。加载该Activity时如果没有实例化，他会创建新的Task后，实例化入栈，如果已经存在，直接调用 onNewIntent，该Activity的Task中不允许启动其它的Activity，任何从该Activity启动的其他Activity都将被 放到其他task中，先检查是否有本应用的task，没有的话就创建。电话拨打界面**

**2.2 activity 的Flags**

**FLAG ACTIVITY NEW TASK**

这个标记位的作用是为Activity指定“singleTask”启动模式，其效果和在XML中指定该启动模式相同。

**FLAG\_ACTIVITY\_CLEAR\_TOP**

    如果设置，并且这个Activity已经在当前的Task中运行，因此，不再是重新启动一个这个Activity的实例，而是在这个Activity上方 的所有Activity都将关闭，然后这个Intent会作为一个新的Intent投递到老的Activity（现在位于顶端）中。

    例如，假设一个Task中包含这些Activity：A，B，C，D。如果D调用了startActivity()，并且包含一个指向Activity B的Intent，那么，C和D都将结束，然后B接收到这个Intent，因此，目前stack的状况是：A，B。

    上例中正在运行的Activity B既可以在onNewIntent()中接收到这个新的Intent，也可以把自己关闭然后重新启动来接收这个Intent。如果它的启动模式声明为 “multiple”(默认值)，并且你没有在这个Intent中设置FLAG\_ACTIVITY\_SINGLE\_TOP标志，那么它将关闭然后重新创 建；对于其它的启动模式，或者在这个Intent中设置FLAG\_ACTIVITY\_SINGLE\_TOP标志，都将把这个Intent投递到当前这个实 例的onNewIntent()中。

    这个启动模式还可以与FLAG\_ACTIVITY\_NEW\_TASK结合起来使用：用于启动一个Task中的根Activity，它会把那个Task中任 何运行的实例带入前台，然后清除它直到根Activity。这非常有用，例如，当从Notification Manager处启动一个Activity。

FLAG ACTIVITY SINGLE TOP

这个标记位的作用是为Activity指定"singleTop”启动模式，其效果和在XML中指定该启动模式相同。

**FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS**

    如果设置，新的Activity不会在最近启动的Activity的列表中保存。

**3 IntentFilter的匹配规则**

隐式 Intent

Intent intent = new Intent(); intent.setAction("com.wooyun.test"); startActivity(intent);

并不知道也不关心接收者是谁，有利于降低发送者和接收者之间的耦合，它一般用在没有明确指出目标组件名称的前提下，一般是用于在不同应用程序之间，如下：

* a. action匹配规则：要求intent中的action 存在 且 必须和过滤规则中的其中一个相同 区分大小写；
* b. category匹配规则：系统会默认加上一个android.intent.category.DEAFAULT，所以intent中可以不存在category，但如果存在就必须匹配其中一个；
* c. data匹配规则：data由两部分组成，mimeType和URI，要求和action相似。如果没有指定URI，URI但默认值为content和file（schema）

使用案例：

（1）如果我们想要匹配 http 以 “.pdf” 结尾的路径，使得别的程序想要打开网络 pdf 时，用户能够可以选择我们的程序进行下载查看。

我们可以将 scheme 设置为 “http”，pathPattern 设置为 “.\*//.pdf”，整个 intent-filter 设置为：

<intent-filter> <action android:name="android.intent.action.VIEW"></action> <category android:name="android.intent.category.DEFAULT"></category> <data android:scheme="http" android:pathPattern=".\*//.pdf"></data> </intent-filter>

如果你只想处理某个站点的 pdf，那么在 data 标签里增加 android:host=”yoursite.com” 则只会匹配<http://yoursite.com/xxx/xxx.pdf>，但这不会匹配 www.yoursite.com，如果你也想匹配这个站点的话，你就需要再添加一个 data 标签，除了 android:host 改为 “www.yoursite.com” 其他都一样。

（2）如果我们做的是一个IM应用，或是其他类似于微博之类的应用，如何让别人通过 Intent 进行调用出现在选择框里呢？我们只用注册 android.intent.action.SEND 与 mimeType 为 “text/plain” 或 “*/*” 就可以了，整个 intent-filter 设置为：

<intent-filter> <action android:name="android.intent.action.SEND" /> <category android:name="android.intent.category.DEFAULT" /> <data mimeType="\*/\*" /> </intent-filter>

这里设置 category 的原因是，创建的 Intent 的实例默认 category 就包含了 Intent.CATEGORY\_DEFAULT ，google 这样做的原因是为了让这个 Intent 始终有一个 category。



（3）如果我们做的是一个音乐播放软件，当文件浏览器打开某音乐文件的时候，使我们的应用能够出现在选择框里？这类似于文件关联了，其实做起来跟上面一样，也很简单，我们只用注册 android.intent.action.VIEW 与 mimeType 为 “audio/\*” 就可以了，整个 intent-filter 设置为：

<intent-filter> <action android:name="android.intent.action.VIEW" /> <category android:name="android.intent.category.DEFAULT" /> <data android:mimeType="audio/\*" /> </intent-filter>



IntentFilter匹配规则。

ActivityManagerService是Android的Java framework的服务框架最重要的服务之一。对于Andorid的Activity、Service、Broadcast、ContentProvider四剑客的管理，包含其生命周期都是通过ActivityManagerService来完成的。对于这四剑客的介绍，此处先略过，后续博主会针对这4剑客分别阐述。

ActivityManagerService简称AMS,它是android中很重要的一个服务，它统筹管理着android的四大组件；统一调度各应用进程；AMN由Binder类由Binder类派生，实现了IActivityManager接口，客户端使用ActivityManager类，因为AMS是系统核心服务，很多API不能直接访问，需要通过ActivityManager,ActivityManager内部通过调用AMN的getDefault方法得到一个ActivityManagerProxy对象，通过它可与AMS通信。

在这里会用到许多的类，为了方便，将一些类进行了简写：

全称 简称

ActivityManagerService AMS

ActivityManagerNative AMN

ActivityManagerProxy AMP

ActivityStackSupervisor ASS

ActivityStack AS

ApplicationThreadProxy ATP

ApplicationThreadNative ATN

ApplicationThread AT

---------------------

## 源文件

·  SystemServer.java

frameworks/base/services/java/com/android/server/SystemServer.java

·  ActivityManagerService.java

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

·  ContextImpl.java

frameworks/base/core/java/android/app/ContextImpl.java

·  ActivityThread.java

frameworks/base/core/java/android/app/ActivityThread.java

·  ActivityStack.java

frameworks/base/services/java/com/android/server/am/ActivityStack.java

·  Am.java

frameworks/base/cmds/am/src/com/android/commands/am/Am.java

·  ProcessRecord.java

frameworks/base/services/java/com/android/server/am/ProcessRecord.java

·  ProcessList.java

frameworks/base/services/java/com/android/server/am/ProcessList.java

·  RuntimeInit.java

frameworks/base/core/java/com/android/internal/os/RuntimeInit.java

## 类图





图6-1  AMS家族图谱

由图6-1可知：

·  AMS由ActivityManagerNative（以后简称AMN）类派生，并实现Watchdog.Monitor和BatteryStatsImpl.BatteryCallback接口。而AMN由Binder派生，实现了IActivityManager接口。

·  客户端使用ActivityManager类。由于AMS是系统核心服务，很多API不能开放供客户端使用，所以设计者没有让ActivityManager直接加入AMS家族。在ActivityManager类内部通过调用AMN的getDefault函数得到一个ActivityManagerProxy对象，通过它可与AMS通信。

单单就一个ActivityManagerService.java文件就代码超过2万行，我们需要需要一个线，再结合binder的知识，来把我们想要了解的东西串起来

AMS是本书碰到的第一块难啃的骨头[①]，涉及的知识点较多。为了帮助读者更好地理解AMS，本章将带领读者按五条不同的线来分析它。

·  第一条线：同其他服务一样，将分析SystemServer中AMS的调用轨迹。

·  第二条线：以am命令启动一个Activity为例，分析应用进程的创建、Activity的启动，以及它们和AMS之间的交互等知识。

·  第三条线和第四条线：分别以Broadcast和Service为例，分析AMS中Broadcast和Service的相关处理流程。

·  第五条线：以一个Crash的应用进程为出发点，分析AMS如何打理该应用进程的身后事。

除了这五条线外，还将统一分析在这五条线中频繁出现的与AMS中应用进程的调度、内存管理等相关的知识

#### AMN.getDefault

[-> ActivityManagerNative.java]

***static public IActivityManager getDefault() {***

***return gDefault.get();***

***}***

gDefault为Singleton类型对象，此次采用单例模式，mInstance为IActivityManager类的代理对象，即ActivityManagerProxy。

public abstract class Singleton<T> {

public final T get() {

synchronized (this) {

if (mInstance == null) {

//首次调用create()来获取AMP对象

mInstance = create();

}

return mInstance;

}

}

}

再来看看create()的过程：

private static final Singleton<IActivityManager> gDefault = new Singleton<IActivityManager>() {

protected IActivityManager create() {

//获取名为"activity"的服务，服务都注册到ServiceManager来统一管理

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

IActivityManager am = asInterface(b);

return am;

}

};

该方法返回的是ActivityManagerProxy对象，那么下一步调用ActivityManagerProxy.startService()方法。

通过Binder通信过程中，提供了一个IActivityManager服务接口，ActivityManagerProxy类与ActivityManagerService类都实现了IActivityManager接口。ActivityManagerProxy作为binder通信的客户端，ActivityManagerService作为binder通信的服务端，根据[Binder系列](http://gityuan.com/2015/10/31/binder-prepare/)文章，ActivityManagerProxy.startService()最终调用ActivityManagerService.startService()，整个流程图如下：



# ActMS启动

*// Activity manager runs the show.*traceBeginAndSlog(**"StartActivityManager"**);  
mActivityManagerService = mSystemServiceManager.startService(  
 ActivityManagerService.Lifecycle.**class**).getService();  
mActivityManagerService.setSystemServiceManager(mSystemServiceManager);  
mActivityManagerService.setInstaller(installer);  
traceEnd();

mSystemServiceManager.startService( ActivityManagerService.Lifecycle.class)这个方法主要是创建ActivityManagerService.Lifecycle对象并调用Lifecycle.onStart方法

## AMS.Lifecycle

通过反射创建AccountManagerService.Lifecycle对象，并调用其onStart方法

这是通过SystemServiceManager这样一个模板类来创建运行在SystemServer中的Framework服务;

**public static final class** Lifecycle **extends** SystemService {  
 **private final** ActivityManagerService mService;  
  
 **public** Lifecycle(Context context) {  
 **super**(context);  
 mService = **new** ActivityManagerService(context);  
 }  
  
 @Override  
 **public void** onStart() {  
 mService.start();  
 }

## ActivityManagerService()

public ActivityManagerService(Context systemContext) {

### 系统Context 和 ActivityThread

（将systemserver进程作为应用进程管理）

mContext = systemContext;

mFactoryTest = FactoryTest.getMode();

mSystemThread = ActivityThread.currentActivityThread();

//创建一个前台线程并获取mHandler

mHandlerThread = new ServiceThread(TAG,

android.os.Process.THREAD\_PRIORITY\_FOREGROUND, false /\*allowIo\*/);

mHandlerThread.start();

mHandler = new MainHandler(mHandlerThread.getLooper());

//创建一个UI线程,这个线程主要处理跟ams内部发出的需要进行ui处理的事件

mUiHandler = new UiHandler();

### 广播队列初始化

BroadcastQueue：前台广播队列和后台广播队列

mFgBroadcastQueue = new BroadcastQueue(this, mHandler,"foreground", BROADCAST\_FG\_TIMEOUT, false);

mBgBroadcastQueue = new BroadcastQueue(this, mHandler,"background", BROADCAST\_BG\_TIMEOUT, true);

mBroadcastQueues[0] = mFgBroadcastQueue;

mBroadcastQueues[1] = mBgBroadcastQueue;

### Service 和 Provider 管理

mServices = new ActiveServices(this);

mProviderMap = new ProviderMap(this);

### // App错误控制

mAppErrors = new AppErrors(mContext, this);

### 系统数据存放目录/data/system/

File dataDir = Environment.getDataDirectory();

File systemDir = new File(dataDir, "system");

systemDir.mkdirs();

### 电池状态信息，理

mBatteryStatsService = new BatteryStatsService(systemDir, mHandler);

### 进程状态

mProcessStats = new ProcessStatsService(this, new File(systemDir, "procstats"));

### 应用权限管

mAppOpsService = new AppOpsService(new File(systemDir, "appops.xml"), mHandler);

### 多用户管理

mStartedUsers.put(UserHandle.USER\_OWNER, new UserState(UserHandle.OWNER, true));

mUserLru.add(UserHandle.USER\_OWNER);

updateStartedUserArrayLocked();

### 最近任务，Activity，Task管理

mRecentTasks = new RecentTasks(this);

mStackSupervisor = new ActivityStackSupervisor(this, mRecentTasks);

mTaskPersister = new TaskPersister(systemDir, mStackSupervisor, mRecentTasks);

### CPU

// 创建一个新线程，用于监控和定时更新系统CPU信息，30分钟更新一次CPU和电池信息

mProcessCpuTracker.init();

mProcessCpuThread = new Thread("CpuTracker") {}

 //mProcessStats为ProcessStats类型，用于统计CPU、内存等信息。其内部工作原理就是

    //读取并解析/proc/stat文件的内容。该文件由内核生成，用于记录kernel及system

    //一些运行时的统计信息。读者可在Linux系统上通过man proc命令查询详细信息

### 加入Watchdog监控起来

Watchdog.getInstance().addMonitor(this);

Watchdog.getInstance().addThread(mHandler);

......

}

### 小结

ams的创建过程中创建了三个线程,一个ui线程，故明思意，主要处理ams本身发了需要进行ui交互的事件；当然当ams跟app进行交互时，app生命周期的调用它不是通过这个ui事件的，它的回调是通过Binder拿到ApplicationThread进行交互的，所以ApplicationThread是ams跟app进行交互的纽带

* **AMS创建过程 涉及到Android 四大组件管理的初始化：**

　　　　Broadcast --》BroadcastQueue  
　　　　Provider --》ProviderMap  
　　　　Service --》ActiveServices  
　　　　Activity --》ActivityStackSupervisor

在main函数中，我们又列出了4个关键函数，分别是：

·  创建AThread线程。虽然AMS的main函数由ServerThread线程调用，但是AMS自己的工作并没有放在ServerThread中去做，而是新创建了一个线程，即AThread线程。

·  ActivityThread.systemMain函数。初始化ActivityThread对象。

·  ActivityThread.getSystemContext函数。用于获取一个Context对象，从函数名上看，该Context代表了System的上下文环境。

·  AMS的startRunning函数。

注意，main函数中有一处等待（wait）及一处通知（notifyAll），原因是：

·  main函数首先需要等待AThread所在线程启动并完成一部分工作。

·  AThread完成那一部分工作后，将等待main函数完成后续的工作。

这种双线程互相等待的情况，在Android代码中比较少见，读者只需了解它们的目的即可。下边来分析以上代码中的第一个关键点。

## setSystemProcess

在SystemServer.startBootstrapServices调用

这个方法的主要作用就是注册各种服务

framework-res.apk系统文件的初始化；因为framework-res.apk是一个apk文件，跟其它apk文件一样，它应该运行在一个进程中，而ams是负责进程管理和调度的，所以apk的进程在ams中是有一个对应的管理结构的

PrpcessRecord就是一个跟进程相关的类，ams中的变量mPidsSelfLocked就是 负责存储相关的进程

---------------------

public void setSystemProcess() {

try {

//将ams服务添加进ServiceManager服务中进行管理

ServiceManager.addService(Context.ACTIVITY\_SERVICE, this, true);

ServiceManager.addService(ProcessStats.SERVICE\_NAME, mProcessStats);

ServiceManager.addService("meminfo", new MemBinder(this));

ServiceManager.addService("gfxinfo", new GraphicsBinder(this));

ServiceManager.addService("dbinfo", new DbBinder(this));

if (MONITOR\_CPU\_USAGE) {

ServiceManager.addService("cpuinfo", new CpuBinder(this));

}

ServiceManager.addService("permission", new PermissionController(this));

ServiceManager.addService("processinfo", new ProcessInfoService(this));

//向pms查询package名为android的applicationInfo信息，其实这个是framework-res.apk

ApplicationInfo info = mContext.getPackageManager().getApplicationInfo(

"android", STOCK\_PM\_FLAGS | MATCH\_SYSTEM\_ONLY);

//该方法调用ContextImpl的nstallSystemApplicationInfo()方法，最终调用LoadedApk的 installSystemApplicationInfo，加载名为“android”的package

mSystemThread.installSystemApplicationInfo(info, getClass().getClassLoader());

synchronized (this) {

//将运行环境和进程管理结构对应起来交给ams统一管理

ProcessRecord app = newProcessRecordLocked(info, info.processName, false, 0);

app.persistent = true;

app.pid = MY\_PID;

app.maxAdj = ProcessList.SYSTEM\_ADJ;

app.makeActive(mSystemThread.getApplicationThread(), mProcessStats);

synchronized (mPidsSelfLocked) {

mPidsSelfLocked.put(app.pid, app);

}

updateLruProcessLocked(app, false, null);

updateOomAdjLocked();

}

} catch (PackageManager.NameNotFoundException e) {

throw new RuntimeException(

"Unable to find android system package", e);

}

}

这个方法的主要作用就是注册各种服务，framework-res.apk系统文件的初始化；因为framework-res.apk是一个apk文件，跟其它apk文件一样，它应该运行在一个进程中，而ams是负责进程管理和调度的，所以apk的进程在ams中是有一个对应的管理结构的，PrpcessRecord就是一个跟进程相关的类，ams中的变量mPidsSelfLocked就是 负责存储相关的进程

这一步就是给SystemServer进程创建ProcessRecord，adj值，就是将SystemServer进程加入到AMS进程管理机制中，跟应用进程一致；

　进程调度更新优先级oomadj值，个人感觉SystemServer进程跟应用进程就不一样，却加入AMS来调度管理，这样做的意义何在？

## installSystemProviders

### 调用入口

[—>SystemServer.java]

private void startOtherServices() {

...

//安装系统Provider

// // **3.将相关provider运行在systemserver进程中：SettingsProvider**

mActivityManagerService.installSystemProviders();

}

将相关provider运行在systemserver进程中：SettingsProvider  
　　　　　　具体安装过程这里暂不详述，在应用启动过程中具体分析。

### installSystemProviders

public static final void installSystemProviders(){

 List<ProviderInfo> providers;

 synchronized (mSelf) {

    /\*

    从mProcessNames找到进程名为“system”且uid为SYSTEM\_UID的ProcessRecord，

    返回值就是前面在installSystemApplication中创建的那个ProcessRecord，它代表

    SystemServer进程

    \*/

   ProcessRecord app = mSelf.mProcessNames.get("system",Process.SYSTEM\_UID);

    //①关键调用，见下文分析

    providers= mSelf.generateApplicationProvidersLocked(app);

    if(providers != null) {

       ......//将非系统APK（即未设ApplicationInfo.FLAG\_SYSTEM标志）提供的Provider

      //从providers列表中去掉

        }

     if(providers != null) {//②为SystemServer进程安装Provider

         mSystemThread.installSystemProviders(providers);

     }

    //监视Settings数据库中Secure表的变化，目前只关注long\_press\_timeout配置的变化

    mSelf.mCoreSettingsObserver = new CoreSettingsObserver(mSelf);

     //UsageStatsService的工作，以后再讨论

    mSelf.mUsageStatsService.monitorPackages();

 }

在代码中列出了两个关键调用，分别是：

·  调用generateApplicationProvidersLocked函数，该函数返回一个ProviderInfo List。

·  调用ActivityThread的installSystemProviders函数。ActivityThread可以看做是进程的Android运行环境，那么installSystemProviders表示为该进程安装ContentProvider。

**注意**此处不再区分系统进程还是应用进程。由于只和ActivityThread交互，因此它运行在什么进程无关紧要。

### generateApplicationProvidersLocked

dfg

#### AMS的 generateApplicationProvidersLocked函数分析

[-->ActivityManagerService.java::generateApplicationProvidersLocked]

private final List<ProviderInfo> generateApplicationProvidersLocked(

                              ProcessRecordapp) {

  List<ProviderInfo> providers = null;

  try {

         //①向PKMS查询满足要求的ProviderInfo，最重要的查询条件包括：进程名和进程uid

         providers = AppGlobals.getPackageManager().

            queryContentProviders(app.processName, app.info.uid,

             STOCK\_PM\_FLAGS | PackageManager.GET\_URI\_PERMISSION\_PATTERNS);

        } ......

   if(providers != null) {

      finalint N = providers.size();

      for(int i=0; i<N; i++) {

         //②AMS对ContentProvider的管理，见下文解释

        ProviderInfo cpi =   (ProviderInfo)providers.get(i);

        ComponentName comp = new ComponentName(cpi.packageName, cpi.name);

        ContentProviderRecord cpr = mProvidersByClass.get(comp);

         if(cpr == null) {

             cpr = new ContentProviderRecord(cpi, app.info, comp);

              //ContentProvider在AMS中用ContentProviderRecord来表示

              mProvidersByClass.put(comp, cpr);//保存到AMS的mProvidersByClass中

          }

          //将信息也保存到ProcessRecord中

         app.pubProviders.put(cpi.name, cpr);

          //保存PackageName到ProcessRecord中

         app.addPackage(cpi.applicationInfo.packageName);

         //对该APK进行dex优化

         ensurePackageDexOpt(cpi.applicationInfo.packageName);

         }

     }

    returnproviders;

 }

由以上代码可知：generateApplicationProvidersLocked先从PKMS那里查询满足条件的ProviderInfo信息，而后将它们分别保存到AMS和ProcessRecord中对应的数据结构中。

先看查询函数queryContentProviders。

##### PMS中 queryContentProviders函数分析

[-->PackageManagerService.java::queryContentProviders]

public List<ProviderInfo>queryContentProviders(String processName,

    int uid,int flags) {

  ArrayList<ProviderInfo> finalList = null;

  synchronized (mPackages) {

     //还记得mProvidersByComponent的作用吗？它以ComponentName为key，保存了

     //PKMS扫描APK得到的PackageParser.Provider信息。读者可参考图4-8

     finalIterator<PackageParser.Provider> i =

                      mProvidersByComponent.values().iterator();

       while(i.hasNext()) {

          final PackageParser.Provider p = i.next();

           //下面的if语句将从这些Provider中搜索本例设置的processName为“system”，

          //uid为SYSTEM\_UID，flags为FLAG\_SYSTEM的Provider

          if (p.info.authority != null

               && (processName == null

                   ||(p.info.processName.equals(processName)

                        &&p.info.applicationInfo.uid == uid))

               && mSettings.isEnabledLPr(p.info, flags)

                && (!mSafeMode || ( p.info.applicationInfo.flags

                        &ApplicationInfo.FLAG\_SYSTEM) != 0)) {

                    if (finalList == null) {

                        finalList = newArrayList<ProviderInfo>(3);

                   }

                 //由PackageParser.Provider得到ProviderInfo，并添加到finalList中

                //关于Provider类及ProviderInfo类，可参考图4-5

                  finalList.add(PackageParser.generateProviderInfo(p,flags));

               }

           }

        }

   if(finalList != null)

      //最终结果按provider的initOrder排序，该值用于表示初始化ContentProvider的顺序

      Collections.sort(finalList, mProviderInitOrderSorter);

   returnfinalList;//返回最终结果

 }

queryContentProviders函数很简单，就是从PKMS那里查找满足条件的Provider，然后生成AMS使用的ProviderInfo信息。为何偏偏能找到SettingsProvider呢？来看它的AndroidManifest.xml文件，如图6-7所示。



由图6-7可知，SettingsProvider设置了其uid为“android.uid.system”，同时在application中设置了process名为“system”。而在framework-res.apk中也做了相同的设置。所以，现在可以确认SettingsProvider将和framework-res.apk运行在同一个进程，即SystemServer中。

**提示**从运行效率角度来说，这样做也是合情合理的。因为SystemServer的很多Service都依赖Settings数据库，把它们放在同一个进程中，可以降低由于进程间通信带来的效率损失

##### 关于ContentProvider的介绍

前面介绍的从PKMS那里查询到的ProviderInfo还属于公有财产，现在我们要将它与AMS及ProcessRecord联系起来。

·  AMS保存ProviderInfo的原因是它要管理ContentProvider。

·  ProcessRecord保存ProviderInfo的原因是ContentProvider最终要落实到一个进程中。其实也是为了方便AMS管理，例如该进程一旦退出，AMS需要把其中的ContentProvider信息从系统中去除。

AMS及ProcessRecord均使用了一个新的数据结构ContentProviderRecord来管理ContentProvider信息。图6-8展示了ContentProviderRecord相应的数据结构。



图6-8  ContentProvicerRecord及相应的“管理团队”

由图6-8可知：

·  ContentProviderRecord从ContentProviderHolder派生，内部保存了ProviderInfo、该Provider所驻留的进程ProcessRecord，以及使用该ContentProvider的客户端进程ProcessRecord（即clients成员变量）。

·  AMS的mProviderByClass成员变量及ProcessRecord的pubProviders成员变量均以ComponentName为Key来保存对应的ContentProviderRecord对象。

至此，Provider信息已经保存到AMS及ProcessRecord中了。那么，下一步的工作是什么

### ActivityThread 的installSystemProviders函数分析

在AMS和ProcessRecord中都保存了Provider信息，但这些仅仅都是一些信息，并不是ContentProvider，因此下面要创建一个ContentProvider实例（即SettingsProvider对象）。该工作由ActivityThread的installSystemProviders来完成，代码如下：

[-->ActivityThread.java::installSystemProviders]

public final void installSystemProviders(List<ProviderInfo>providers) {

  if(providers != null)

    //调用installContentProviders，第一个参数真实类型是Application

   installContentProviders(mInitialApplication, providers);

}

installContentProviders这个函数是所有ContentProvider产生的必经之路，其代码如下：

[-->ActivityThread.java::installContentProviders]

private void installContentProviders(

           Context context, List<ProviderInfo> providers) {

   finalArrayList<IActivityManager.ContentProviderHolder> results =

                   new ArrayList<IActivityManager.ContentProviderHolder>();

  Iterator<ProviderInfo> i = providers.iterator();

   while(i.hasNext()) {

    ProviderInfo cpi = i.next();

     //①调用installProvider函数，得到一个IContentProvider对象

    IContentProvider cp = installProvider(context, null, cpi, false);

     if (cp!= null) {

       IActivityManager.ContentProviderHolder cph =

                    newIActivityManager.ContentProviderHolder(cpi);

       cph.provider = cp;

       //将返回的cp保存到results数组中

       results.add(cph);

        synchronized(mProviderMap){

         //mProviderRefCountMap，；类型为HashMap<IBinder,ProviderRefCount>，

         //主要通过ProviderRefCount对ContentProvider进行引用计数控制，一旦引用计数

         //降为零，表示系统中没有地方使用该ContentProvider，要考虑从系统中注销它

         mProviderRefCountMap.put(cp.asBinder(), new ProviderRefCount(10000));

         }

     }

   }

    try {

       //②调用AMS的publishContentProviders注册这些ContentProvider，第一个参数

       //为ApplicationThread

       ActivityManagerNative.getDefault().publishContentProviders(

                      getApplicationThread(), results);

        } ......

    }

installContentProviders实际上是标准的ContentProvider安装时调用的程序。安装ContentProvider包括两方面的工作：

·  先在ActivityThread中通过installProvider得到一个ContentProvider实例。

·  向AMS发布这个ContentProvider实例。如此这般，一个APK中声明的ContentProvider才能登上历史舞台，发挥其该有的作用。

提示 上述工作其实和Binder Service类似，一个Binder Service也需要先创建，然后注册到ServiceManager中。

马上来看ActivityThread的installProvider函数。

##### ActivityThread的installProvider函数分析

Sad

[-->ActivityThread.java::installProvider]

private IContentProvider installProvider(Contextcontext,

           IContentProvider provider, ProviderInfoinfo, boolean noisy) {

//注意本例所传的参数：context为mInitialApplication，provider为null,info不为null，

// noisy为false

  ContentProvider localProvider = null;

   if(provider == null) {

      Context c = null;

      ApplicationInfo ai = info.applicationInfo;

       /\*

        下面这个if判断的作用就是为该ContentProvider找到对应的Application。

        在AndroidManifest.xml中，ContentProvider是Application的子标签，所以

       ContentProvider和Application有一种对应关系。在本例中，传入的context（

        其实是mInitialApplication）代表的是framework-res.apk，而Provider代表的

        是SettingsProvider。而SettingsProvider.apk所对应的Application还未创建，

        所以下面的判断语句最终会进入最后的else分支

      \*/

       if(context.getPackageName().equals(ai.packageName)) {

          c= context;

       }else if (mInitialApplication != null &&

                 mInitialApplication.getPackageName().equals(ai.packageName)){

          c = mInitialApplication;

       } else {

         try{

             //ai.packageName应该是SettingsProvider.apk的Package，

             //名为“com.android.providers.settings”

             //下面将创建一个Context，指向该APK

              c = context.createPackageContext(ai.packageName,

                           Context.CONTEXT\_INCLUDE\_CODE);

           }

       }//if(context.getPackageName().equals(ai.packageName))判断结束

       if (c == null)  return null;

       try {

         /\*

         为什么一定要找到对应的Context呢？除了ContentProvider和Application的

         对应关系外，还有一个决定性原因：即只有对应的Context才能加载对应APK的Java字节码，

         从而可通过反射机制生成ContentProvider实例

        \*/

         finaljava.lang.ClassLoader cl = c.getClassLoader();

         //通过Java反射机制得到真正的ContentProvider，

         //此处将得到一个SettingsProvider对象

         localProvider=(ContentProvider)cl.loadClass(info.name).newInstance();

         //从ContentProvider中取出其mTransport成员（见下文分析）

         provider =localProvider.getIContentProvider();

         if (provider == null) return null;

         //初始化该ContentProvider，内部会调用其onCreate函数

        localProvider.attachInfo(c, info);

      }......

   }//if(provider == null)判断结束

  synchronized (mProviderMap) {

      /\*

        ContentProvider必须指明一个和多个authority，在第4章曾经提到过，

         在URL中host:port的组合表示一个authority。这个单词不太好理解，可简单

         认为它用于指定ContentProvider的位置（类似网站的域名）

     \*/

      String names[] =PATTERN\_SEMICOLON.split(info.authority);

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

             ProviderClientRecord pr = newProviderClientRecord(names[i],

                                           provider, localProvider);

            try {

                //下面这句对linkToDeath的调用颇让人费解，见下文分析

                  provider.asBinder().linkToDeath(pr, 0);

                  mProviderMap.put(names[i], pr);

           }......

      }//for循环结束

     if(localProvider != null) {

         // mLocalProviders用于存储由本进程创建的ContentProvider信息

          mLocalProviders.put(provider.asBinder(),

                   new ProviderClientRecord(null, provider, localProvider));

              }

     }//synchronized 结束

  return provider;

}

以上代码不算复杂，但是涉及一些数据结构和一句令人费解的对inkToDeath函数的调用。先来说说那句令人费解的调用。

在本例中，provider变量并非通过函数参数传入，而是在本进程内部创建的。provider在本例中是Bn端（后面分析ContentProvider的getIContentProvider时即可知道），Bn端进程为Bn端设置死亡通知本身就比较奇怪。如果Bn端进程死亡，它设置的死亡通知也无法发送给自己。幸好源代码中有句注释：“Cache the pointer for the remote provider”。意思是如果provider参数是通过installProvider传递过来的（即该Provider代表远端进程的ContentProvider，此时它应为Bp端），那么这种处理是合适的。不管怎样，这仅仅是为了保存pointer，所以也无关宏旨。

至于代码中涉及的数据结构，我们整理为如图6-9所示。



图6-9  ActivityThread中ContentProvider涉及的数据结构

由图6-9可知：

·  ContentProvider类本身只是一个容器，而跨进程调用的支持是通过内部类Transport实现的。Transport从ContentProviderNative派生，而ContentProvider的成员变量mTransport指向该Transport对象。ContentProvider的getIContentProvider函数即返回mTransport成员变量。

·  ContentProviderNative从Binder派生，并实现了IContentProvider接口。其内部类ContentProviderProxy是供客户端使用的。

·  ProviderClientRecord是ActivityThread提供的用于保存ContentProvider信息的一个数据结构。它的mLocalProvider用于保存ContentProvider对象，mProvider用于保存IContentProvider对象。另外一个成员mName用于保存该ContentProvider的一个authority。注意，ContentProvider可以定义多个authority，就好像一个网站有多个域名一样。

至此，本例中的SettingProvider已经创建完毕，接下来的工作就是把它推向历史舞台——即发布该Provider。

##### publishContentProviders分析

publicContentProviders函数用于向AMS注册ContentProviders，其代码如下：

[-->ActivityManagerService.java::publishContentProviders]

 publicfinal void publishContentProviders(IApplicationThread caller,

                           List<ContentProviderHolder>providers) {

  ......

  synchronized(this){

    //找到调用者所在的ProcessRecord对象

   finalProcessRecord r = getRecordForAppLocked(caller);

   ......

   finallong origId = Binder.clearCallingIdentity();

   final intN = providers.size();

   for (inti=0; i<N; i++) {

      ContentProviderHolder src = providers.get(i);

       ......

       //①注意：先从该ProcessRecord中找对应的ContentProviderRecord

     ContentProviderRecord dst = r.pubProviders.get(src.info.name);

      if(dst != null) {

          ComponentName comp = newComponentName(dst.info.packageName,

                                                         dst.info.name);

          //以ComponentName为key，保存到mProvidersByClass中

         mProvidersByClass.put(comp, dst);

         String names[] = dst.info.authority.split(";");

         for (int j = 0; j < names.length; j++)

               mProvidersByName.put(names[j], dst);//以authority为key，保存

              //mLaunchingProviders用于保存处于启动状态的Provider

               int NL = mLaunchingProviders.size();

               int j;

               for (j=0; j<NL; j++) {

                 if (mLaunchingProviders.get(j) ==dst) {

                      mLaunchingProviders.remove(j);

                      j--;

                      NL--;

                  }//

              }//for (j=0; j<NL; j++)结束

              synchronized (dst) {

                  dst.provider = src.provider;

                  dst.proc = r;

                  dst.notifyAll();

              }//synchronized结束

             updateOomAdjLocked(r);//每发布一个Provider，需要调整对应进程的oom\_adj

            }//for(int j = 0; j < names.length; j++)结束

        }//for(int i=0; i<N; i++)结束

        Binder.restoreCallingIdentity(origId);

    }// synchronized(this)结束

 }

这里应解释一下publishContentProviders的工作流程：

·  先根据调用者的pid找到对应的ProcessRecord对象。

·  该ProcessRecord的pubProviders中保存了ContentProviderRecord信息。该信息由前面介绍的AMS的generateApplicationProvidersLocked函数根据Package本身的信息生成。此处将判断要发布的ContentProvider是否由该Package声明。

·  如果判断返回成功，则将该ContentProvider及其对应的authority加到mProvidersByName中。注意，AMS中还有一个mProvidersByClass变量，该变量以ContentProvider的ComponentName为key，即系统提供多种方式找到某一个ContentProvider，一种是通过 authority，另一种方式就是指明ComponentName。

·  mLaunchingProviders和最后的notifyAll函数用于通知那些等待ContentProvider所在进程启动的客户端进程。例如，进程A要查询一个数据库，需要通过进程B中的某个ContentProvider 来实施。如果B还未启动，那么AMS就需要先启动B。在这段时间内，A需要等待B启动并注册对应的ContentProvider。B一旦完成注册，就需要告知A退出等待以继续后续的查询工作。

现在，一个SettingsProvider就算正式在系统中挂牌并注册了，此后，和Settings数据库相关的操作均由它来打理。

#### 3.  ASM的installSystemProviders总结

AMS的installSystemProviders函数其实就是用于启动SettingsProvider，其中比较复杂的是ContentProvider相关的数据结构，读者可参考图6-9。

## AMS.setWindowManager

// Start Window Manager

wm = WindowManagerService.main();

// **4.直接保存wms对象，与WMS交互**

mActivityManagerService.setWindowManager(wm);

AMS保存对象，本身就在同一个进程，WMS与WMS之间的交互式直接调用速度会更快，其他服务为何不这样，是因为耦合太强，还是实时性要求更高？

## showBootMessageXXXX

/ **5.通过WMS 弹出“正在启动应用”框**

// R.string.android\_upgrading\_starting\_apps

ActivityManagerNative.getDefault().showBootMessage();

这段代码可以看出，第一次开机时不会弹出这个提示框的，如果想要第一次开机也要弹出提示框，可以去掉isFirstBoot判定。

　弹出“正在启动应用”框，这里为何不直接调用AMS的showBootMessage而是通过binder方式调用，其他接口都是直接调用，为何？直接调用有何不可吗？

## AMS.systemReady

|  |
| --- |
| public final class ActivityManagerService extends ActivityManagerNative  implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {    public void systemReady(final Runnable goingCallback) {  ... //update相关  mSystemReady = true;    //杀掉所有非persistent进程  removeProcessLocked(proc, true, false, "system update done");  mProcessesReady = true;  goingCallback.run(); //[见小节1.6.2]    addAppLocked(info, false, null); //启动所有的persistent进程  mBooting = true;    //启动home  **startHomeActivityLocked**(mCurrentUserId, "systemReady");  //恢复栈顶的Activity  mStackSupervisor.resumeTopActivitiesLocked();  }  } |

System\_server主线程的启动工作,总算完成, 进入Looper.loop()状态,等待其他线程通过handler发送消息再处理.

### run()

#### startObservingNativeCrashes

f

启动carash监听

try {

mActivityManagerService.startObservingNativeCrashes();

}

#### WebViewFactoryPreparation

Sdaf

if (!mOnlyCore) {

Slog.i(TAG, "WebViewFactory preparation");

Trace.traceBegin(Trace.TRACE\_TAG\_SYSTEM\_SERVER, "WebViewFactoryPreparation");

mWebViewUpdateService.prepareWebViewInSystemServer();

Trace.traceEnd(Trace.TRACE\_TAG\_SYSTEM\_SERVER);

}

作者：Yink\_Liu

链接：https://www.jianshu.com/p/29e9e1dfa6c2

來源：简书

简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

#### startSystemUi

try {

startSystemUi(context);

}

#### WatchDog

//启动WatchDog监控核心服务状态

Watchdog.getInstance().start()

#### systemRunning

if (networkScoreF != null) networkScoreF.systemReady(); if (networkManagementF != null) networkManagementF.systemReady(); if (networkStatsF != null) networkStatsF.systemReady(); if (locationF != null) locationF.systemRunning(); ...//等等，很多类似方法

### 进程并杀死

系统还没准备之前不允许启动非persistent进程

在AMS还未启动完毕就启动完了呢？对，那些声明了persistent为true的进程有可能

先于AMS启动的进程，哪些进程有如此能耐，

// **2. 收集已经启动的进程并杀死，除过persistent常驻进程**

ArrayList<ProcessRecord> procsToKill = null;

synchronized(mPidsSelfLocked) {

for (int i=mPidsSelfLocked.size()-1; i>=0; i--) {

ProcessRecord proc = mPidsSelfLocked.valueAt(i);

if (!isAllowedWhileBooting(proc.info)){

if (procsToKill == null) {

procsToKill = new ArrayList<ProcessRecord>();

}

procsToKill.add(proc);

}

}

}

synchronized(this) {

if (procsToKill != null) {

for (int i=procsToKill.size()-1; i>=0; i--) {

ProcessRecord proc = procsToKill.get(i);

Slog.i(TAG, "Removing system update proc: " + proc);

removeProcessLocked(proc, true, false, "system update done");

}

}

// Now that we have cleaned up any update processes, we

// are ready to start launching real processes and know that

// we won't trample on them any more.

mProcessesReady = true;

}

#### isAllowedWhileBooting

**boolean** isAllowedWhileBooting(ApplicationInfo ai) {  
 **return** (ai.flags&ApplicationInfo.FLAG\_PERSISTENT) != 0;  
}

### startPersistentApps

*// Only start up encryption-aware persistent apps; once user is  
// unlocked we'll come back around and start unaware apps*startPersistentApps(PackageManager.MATCH\_DIRECT\_BOOT\_AWARE);

#### persistent属性作用

persistent属性是用于application标签上的

该属性的定义在frameworks/base/core/res/res/values/attrs\_manifest.xml中，其定义如下

通过官方注释我知道该属性用于是否让你的应用一直处于运行状态（通常说的常驻内存）。设置

该属性为true的app具有如下特点：

在系统启动的时候会被系统启动起来

在该app被强制杀掉后系统会重新启动该app，这种情况只针对系统内置app，第三方安装的app不会被重启

**if** (mFactoryTest == FactoryTest.FACTORY\_TEST\_LOW\_LEVEL) **return**;  
  
**synchronized** (**this**) {  
 **try** {  
 **final** List<ApplicationInfo> apps = AppGlobals.getPackageManager()  
 .getPersistentApplications(STOCK\_PM\_FLAGS | matchFlags).getList();  
 **for** (ApplicationInfo app : apps) {  
 **if** (!**"android"**.equals(app.packageName)) {  
 addAppLocked(app, **null**, **false**, **null** */\* ABI override \*/*);  
 }  
 }  
 } **catch** (RemoteException ex) {  
 }  
}

persistent为true并且是系统app的话一定会被选中，但是如果是第三方安装的应用的话只能在非“安全模式”下才会被选中

#### PKMS中的getPersistentApplicationsInternal方

**final boolean** matchesUnaware = ((flags & MATCH\_DIRECT\_BOOT\_UNAWARE) != 0)  
 && !p.applicationInfo.isDirectBootAware();  
**final boolean** matchesAware = ((flags & MATCH\_DIRECT\_BOOT\_AWARE) != 0)  
 && p.applicationInfo.isDirectBootAware();  
  
**if** ((p.applicationInfo.flags & ApplicationInfo.FLAG\_PERSISTENT) != 0  
 && (!mSafeMode || isSystemApp(p))  
 && (matchesUnaware || matchesAware)) {

#### addAppLocked

addAppLocked(info, false, null); //启动所有的persistent进程

mBooting = true;

final ProcessRecord addAppLocked(ApplicationInfo info, boolean isolated,

String abiOverride) {

ProcessRecord app;

//传递进来的isolated=false，所有一定会调用getProcessRecordLocked方法，但是由于是第一次启动，所有返回的app = null

if (!isolated) {

app = getProcessRecordLocked(info.processName, info.uid, true);

} else {

app = null;

}

if (app == null) {

//为新的app创建新的ProcessRecord对象

app = newProcessRecordLocked(info, null, isolated, 0);

updateLruProcessLocked(app, false, null);

updateOomAdjLocked();

}

// This package really, really can not be stopped.

try {

//由于是开机第一次启动，所以新的app的启动状态是将要被启动状态，所以

//该app的停止状态stoped被设置为false

AppGlobals.getPackageManager().setPackageStoppedState(

info.packageName, false, UserHandle.getUserId(app.uid));

} catch (RemoteException e) {

} catch (IllegalArgumentException e) {

Slog.w(TAG, "Failed trying to unstop package "

+ info.packageName + ": " + e);

}

//在这里对persistent的app进行过滤，只有既是系统app，persistent为true的app才会在

//异常死亡之后被重启

if ((info.flags & PERSISTENT\_MASK) == PERSISTENT\_MASK) {

app.persistent = true;

app.maxAdj = ProcessList.PERSISTENT\_PROC\_ADJ;

}

//如果该app已经启动了，则不用处理，否则调用startProcessLocked方法启动app。

//由于启动app是异步进行的，会将正在启动而还没有启动完成的app添加到

//mPersistentStartingProcesses列表中。当启动完成后 再移除

if (app.thread == null && mPersistentStartingProcesses.indexOf(app) < 0) {

mPersistentStartingProcesses.add(app);

//启动该app

startProcessLocked(app, "added application", app.processName, abiOverride,

null /\* entryPoint \*/, null /\* entryPointArgs \*/);

}

return app;

}

}

接下来调用startProcessLocked方法启动app进程，在app启动完成后会在ActivityThread中调用AMS的attachApplication，将该app从mPersistentStartingProcesses中移除，并注册一个死亡讣告监听器AppDeathRecipient，用于在app异常被杀后的处理工作。

app.maxAdj = ProcessList.PERSISTENT\_PROC\_ADJ;非常重要，这类进程的优先级很高

#### pp被异常结束后系统重新启动persistent为true的app

进程启动时为app注册了一个死亡讣告，当该app被杀掉之后会调用AppDeathRecipient的binderDied方法，该方法会调用appDiedLocked方法进行善后处理，系统在进程死掉之后会对死掉的进程进行清理和资源回收，但是在这个过程中如果你的app是persistent的话会被重启：

binderDied

#### REF

https://www.jianshu.com/p/02aeac34e2c8

### startHomeActivityLocked

**startHomeActivityLocked**(mCurrentUserId, "systemReady");

### 发送启动广播

try {

Intent intent = new Intent(Intent.ACTION\_USER\_STARTED);

intent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY

| Intent.FLAG\_RECEIVER\_FOREGROUND);

intent.putExtra(Intent.EXTRA\_USER\_HANDLE, currentUserId);

broadcastIntentLocked(null, null, intent,

null, null, 0, null, null, null, AppOpsManager.OP\_NONE,

null, false, false, MY\_PID, Process.SYSTEM\_UID,

currentUserId);

intent = new Intent(Intent.ACTION\_USER\_STARTING);

intent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY);

intent.putExtra(Intent.EXTRA\_USER\_HANDLE, currentUserId);

broadcastIntentLocked(null, null, intent,

null, new IIntentReceiver.Stub() {

@Override

public void performReceive(Intent intent, int resultCode, String data,

Bundle extras, boolean ordered, boolean sticky, int sendingUser)

throws RemoteException {

}

}, 0, null, null,

new String[] {INTERACT\_ACROSS\_USERS}, AppOpsManager.OP\_NONE,

null, true, false, MY\_PID, Process.SYSTEM\_UID, UserHandle.USER\_ALL);

}

## StackSupervisor.resumeFocusedStackTopActivityLocked()

**boolean** resumeFocusedStackTopActivityLocked(  
 ActivityStack targetStack, ActivityRecord target, ActivityOptions targetOptions) {  
 **if** (targetStack != **null** && isFocusedStack(targetStack)) {  
 **return** targetStack.resumeTopActivityUncheckedLocked(target, targetOptions);  
 }  
 **final** ActivityRecord r = mFocusedStack.topRunningActivityLocked();  
 **if** (r == **null** || r.state != RESUMED) {  
 mFocusedStack.resumeTopActivityUncheckedLocked(**null**, **null**);  
 }  
 **return false**;  
}

### ActivityStack .resumeTopActivityUncheckedLocked

**boolean** resumeTopActivityUncheckedLocked(ActivityRecord prev, ActivityOptions options) {  
 **if** (mStackSupervisor.inResumeTopActivity) {  
 *// Don't even start recursing.* **return false**;  
 }  
  
 **boolean** result = **false**;  
 **try** {  
 *// Protect against recursion.* mStackSupervisor.inResumeTopActivity = **true**;  
 **if** (mService.mLockScreenShown == ActivityManagerService.LOCK\_SCREEN\_LEAVING) {  
 mService.mLockScreenShown = ActivityManagerService.LOCK\_SCREEN\_HIDDEN;  
 mService.updateSleepIfNeededLocked();  
 }  
 result = resumeTopActivityInnerLocked(prev, options);  
 } **finally** {  
 mStackSupervisor.inResumeTopActivity = **false**;  
 }  
 **return** result;  
}

### ActivityStack .resumeTopActivityInnerLocked

## finishBooting

### activityIdleInternalLocked-----------

### AS.moveToFront

### setFocusStackUnchecked

checkFinishBootingLocked

### Ams.postFinishBooting

## 流程图



## 参考

ActivityManagerService启动分析

<https://www.jianshu.com/p/29e9e1dfa6c2>

# AMS onCreate

在android系统中，应用程序是由Activity组成的，因此，应用程序的启动过程实际上就是应用程序中的默认Activity的启动过程，本文将详细分析应用程序框架层的源代码，了解Android应用程序的启动过程。在手机屏幕中点击应用程序图标开始——到应用MainActivity展示出来结束。

下面详细分析每一步是如何实现的。

## Luancher进程

### Step 1. Launcher.startActivitySafely

1. **void** startActivitySafely(Intent intent, Object tag) {
2. intent.addFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);
3. try {
4. startActivity(intent);
5. } catch (ActivityNotFoundException e) {
6. ......
7. } catch (SecurityException e) {
8. ......
9. }
10. }

MainActivity，这里是AndroidManifest.xml文件中配置的：

1. **<activity** android:name=".MainActivity"
2. android:label="@string/app\_name"**>**
3. **<intent-filter>**
4. **<action** android:name="android.intent.action.MAIN" **/>**
5. **<category** android:name="android.intent.category.LAUNCHER" **/>**
6. **</intent-filter>**
7. **</activity>**

因此，这里的intent包含的信息为：action = "android.intent.action.Main"，category="android.intent.category.LAUNCHER", cmp="shy.luo.activity/.MainActivity"，表示它要启动的Activity为shy.luo.activity.MainActivity。Intent.FLAG\_ACTIVITY\_NEW\_TASK表示要在一个新的Task中启动这个Activity，注意，Task是Android系统中的概念，它不同于进程Process的概念。简单地说，一个Task是一系列Activity的集合，这个集合是以堆栈的形式来组织的，遵循后进先出的原则。一个task可以包含多个process。





在Android系统中，应用程序是由Launcher启动起来的，其实，Launcher本身也是一个应用程序，其它的应用程序安装后，就会Launcher的界面上出现一个相应的图标，点击这个图标时，Launcher就会对应的应用程序启动起来。

### Step 2. Activity.startActivity

Launcher的父类Activity实现了startActivity函数，因此，这里就调用了Activity.startActivity函数。 这个函数实现很简单，它调用startActivityForResult来进一步处理，第二个参数传入-1表示不需要这个Actvity结束后的返回结果。

1. @Override
2. **public** **void** startActivity(Intent intent) {
3. startActivityForResult(intent, -1);
4. }

### Step 3. Activity.startActivityForResult

1. **public** **void** startActivityForResult(Intent **intent**, **int** requestCode) {
2. **if** (mParent == **null**) {
3. Instrumentation.ActivityResult ar =
4. mInstrumentation.execStartActivity(
5. **this**, mMainThread.getApplicationThread(), mToken, **this**,
6. **intent**, requestCode);
7. ......
8. } **else** {
9. ......
10. }

这里的mInstrumentation是Activity类的成员变量，它的类型是Intrumentation，定义在frameworks/base/core/java/android/app/Instrumentation.java文件中，它用来监控应用程序和系统的交互。

这里的mMainThread也是Activity类的成员变量，它的类型是ActivityThread，它代表的是应用程序的主线程，这里通过mMainThread.getApplicationThread获得它里面的ApplicationThread成员变量，它是一个Binder对象，后面我们会看到，AtMS会使用它来和ActivityThread来进行进程间通信。这里我们需注意的是，**这里的mMainThread代表的是Launcher应用程序运行的进程。**

这里的mToken也是Activity类的成员变量，它是一个Binder对象的远程接口。

### Step 11. ActivityStack.startPausingLocked

1. **public** **class** ActivityStack {
3. ......
5. **private** **final** **void** startPausingLocked(**boolean** userLeaving, **boolean** uiSleeping) {
6. **if** (mPausingActivity != **null**) {
7. ......
8. }
9. ActivityRecord prev = mResumedActivity;
10. **if** (prev == **null**) {
11. ......
12. }
13. ......
14. mResumedActivity = **null**;
15. mPausingActivity = prev;
16. mLastPausedActivity = prev;
17. prev.state = ActivityState.PAUSING;
18. ......
20. **if** (prev.app != **null** && prev.app.thread != **null**) {
21. ......
22. **try** {
23. ......
24. prev.app.thread.schedulePauseActivity(prev, prev.finishing, userLeaving,
25. prev.configChangeFlags);
26. ......
27. } **catch** (Exception e) {
28. ......
29. }
30. } **else** {
31. ......
32. }
34. ......
36. }
38. ......
40. }

函数首先把mResumedActivity保存在本地变量prev中。在上一步Step 10中，说到mResumedActivity就是Launcher，因此，这里把Launcher进程中的ApplicationThread对象取出来，通过它来通知Launcher这个Activity它要进入Paused状态了。当然，这里的prev.app.thread是一个ApplicationThread对象的远程接口，通过调用这个远程接口的schedulePauseActivity来通知Launcher进入Paused状态。

参数prev.finishing表示prev所代表的Activity是否正在等待结束的Activity列表中，由于Laucher这个Activity还没结束，所以这里为false；参数prev.configChangeFlags表示哪些config发生了变化，这里我们不关心它的值。

### Step 4. Instrumentation.execStartActivity

在frameworks/base/core/java/android/app/Instrumentation.java文件中：

1. **public** **class** Instrumentation {
2. **public** ActivityResult execStartActivity(
3. Context who, IBinder contextThread, IBinder token, Activity target,
4. Intent intent, **int** requestCode) {
5. IApplicationThread whoThread = (IApplicationThread) contextThread;
6. **if** (mActivityMonitors != **null**) {
7. ......
8. }
9. **try** {
10. **int** result = ActivityManagerNative.getDefault()
11. .**startActivity**(whoThread, intent,
12. intent.**resolveTypeIfNeeded**(who.getContentResolver()),
13. **null**, 0, token, target != **null** ? target.mEmbeddedID : **null**,
14. requestCode, **false**, **false**);
15. ......
16. } **catch** (RemoteException e) {
17. }
18. **return** **null**;
19. }
21. }

这里的ActivityManagerNative.getDefault返回**AtMS**的远程接口，即**ActivityManagerProxy**接口，具体可以参考Android系统在新进程中启动自定义服务过程（startService）的原理分析一文。

这里的intent.resolveTypeIfNeeded返回这个intent的MIME类型，在这个例子中，没有AndroidManifest.xml设置MainActivity的MIME类型，因此，这里返回null。

这里的target不为null，但是target.mEmbddedID为null，我们不用关注。

### Step 5. ActivityManagerProxy.startActivity

定义在frameworks/base/core/java/android/app/ActivityManagerNative.java文件中：

1. **class** ActivityManagerProxy **implements** IActivityManager
2. {
3. **public** **int** startActivity(IApplicationThread caller, Intent intent,
4. String resolvedType, Uri[] grantedUriPermissions, **int** grantedMode,
5. IBinder resultTo, String resultWho,
6. **int** requestCode, **boolean** onlyIfNeeded,
7. **boolean** debug) **throws** RemoteException {
8. Parcel data = Parcel.obtain();
9. Parcel reply = Parcel.obtain();
10. data.writeInterfaceToken(IActivityManager.descriptor);
11. data.writeStrongBinder(caller != **null** ? caller.asBinder() : **null**);
12. intent.writeToParcel(data, 0);
13. data.writeString(resolvedType);  **//null**
14. data.writeTypedArray(grantedUriPermissions, 0);   **//null**
15. data.writeInt(grantedMode);
16. data.writeStrongBinder(resultTo);
17. data.writeString(resultWho);
18. data.writeInt(requestCode);  //-1
19. data.writeInt(onlyIfNeeded ? 1 : 0);  //false
20. data.writeInt(debug ? 1 : 0);  //false
21. **mRemote**.transact(START\_ACTIVITY\_TRANSACTION, data, reply, 0);
22. reply.readException();
23. **int** result = reply.readInt();
24. reply.recycle();
25. data.recycle();
26. **return** result;
27. }
28. }

这里的参数比较多，我们先整理一下。从上面的调用可以知道，这里的参数resolvedType、grantedUriPermissions和resultWho均为null；参数caller为ApplicationThread类型的Binder实体；参数resultTo为一个Binder实体的远程接口，我们先不关注它；参数grantedMode为0，我们也先不关注它；参数requestCode为-1；参数onlyIfNeeded和debug均空false。**mRemote开始向AMS进程间通信，发送**START\_ACTIVITY\_TRANSACTION消息。

## AMS进程

上述都在launer中处理的，接下是AMS响应START\_ACTIVITY\_TRANSACTIO的进程间通信

### Step 6. AtMS.startActivity

上一步Step 5通过Binder驱动程序就进入到AtMS的startActivity函数来了，它定义在frameworks/base/services/java/com/android/server/am/AtMS.java

文件中：

1. **public** **final** **class** AtMS **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
3. **public** **final** **int** startActivity(IApplicationThread caller,
4. Intent intent, String resolvedType, Uri[] grantedUriPermissions,
5. **int** grantedMode, IBinder resultTo,
6. String resultWho, **int** requestCode, **boolean** onlyIfNeeded,
7. **boolean** debug) {
8. **return** **mMainStack**.startActivityMayWait(caller, intent, resolvedType,
9. grantedUriPermissions, grantedMode, resultTo, resultWho,
10. requestCode, onlyIfNeeded, debug, **null**, **null**);
11. }
12. }

这里只是简单地将操作转发给成员变量mMainStack的startActivityMayWait函数，这里的mMainStack的类型为ActivityStack。

### mActivityStarter.startActivityMayWait

这个函数定义在frameworks/base/services/java/com/android/server/am/ mActivityStarter.java文件中：

ActivityStackSupervisor

1. **public** **class** ActivityStack {
2. **final** **int** startActivityMayWait(IApplicationThread caller,
3. Intent intent, String resolvedType, Uri[] grantedUriPermissions,
4. **int** grantedMode, IBinder resultTo,
5. String resultWho, **int** requestCode, **boolean** onlyIfNeeded,
6. **boolean** debug, WaitResult outResult, Configuration config) {
7. **boolean** componentSpecified = intent.getComponent() != **null**;
8. // Don't modify the client's object!
9. intent = **new** Intent(intent);
10. // Collect information about the target of the Intent.
11. ActivityInfo aInfo;
12. **try** {
13. ResolveInfo rInfo =
14. AppGlobals.getPackageManager().resolveIntent(
15. intent, resolvedType,
16. PackageManager.MATCH\_DEFAULT\_ONLY
17. | AtMS.STOCK\_PM\_FLAGS);
18. aInfo = rInfo != **null** ? rInfo.activityInfo : **null**;
19. } **catch** (RemoteException e) {
20. ......
21. }
23. **if** (aInfo != **null**) {
24. // Store the found target back into the intent, because now that
25. // we have it we never want to do this again.  For example, if the
26. // user navigates back to this point in the history, we should
27. // always restart the exact same activity.
28. intent.setComponent(**new** ComponentName(
29. aInfo.applicationInfo.packageName, aInfo.name));
30. ......
31. }
33. **synchronized** (mService) {
34. **int** callingPid;
35. **int** callingUid;
36. **if** (caller == **null**) {
37. ......
38. } **else** {
39. callingPid = callingUid = -1;
40. }
42. mConfigWillChange = config != **null**
43. && mService.mConfiguration.diff(config) != 0;
44. **int** res = **startActivityLocked**(caller, intent, resolvedType,
45. grantedUriPermissions, grantedMode, aInfo,
46. resultTo, resultWho, requestCode, callingPid, callingUid,
47. onlyIfNeeded, componentSpecified);
48. **return** res;
49. }

注意，从Step 6传下来的参数outResult和config均为null，此外，表达式(aInfo.applicationInfo.flags&ApplicationInfo.FLAG\_CANT\_SAVE\_STATE) != 0为false，因此，这里忽略了无关代码。 下面语句对参数intent的内容进行解析，得到MainActivity的相关信息，保存在aInfo变量中：

1. ActivityInfo aInfo;
2. **try** {
3. ResolveInfo rInfo =   AppGlobals.getPackageManager().resolveIntent(  intent, resolvedType,
4. PackageManager.MATCH\_DEFAULT\_ONLY
5. | AtMS.STOCK\_PM\_FLAGS);
6. aInfo = rInfo != **null** ? rInfo.activityInfo : **null**;
7. } **catch** (RemoteException e) {
8. ......
9. }

解析之后，得到的aInfo.applicationInfo.packageName的值为"k.demo.asys"，aInfo.name的值为"k.demo.asys.activity.MainActivity"，这是在这个实例的配置文件AndroidManifest.xml里面配置的。此外，函数开始的地方调用intent.getComponent()函数的返回值不为null，因此，这里的componentSpecified变量为true。

接下去就调用startActivityLocked进一步处理了。

### Step 8. ActivityStarter.startActivityLocked

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **final** **int** startActivityLocked(IApplicationThread caller,
3. Intent intent, String resolvedType,
4. Uri[] grantedUriPermissions,
5. **int** grantedMode, ActivityInfo aInfo, IBinder resultTo,
6. String resultWho, **int** requestCode,
7. **int** callingPid, **int** callingUid, **boolean** onlyIfNeeded,
8. **boolean** componentSpecified) {
9. **int** err = START\_SUCCESS;
11. ProcessRecord callerApp = **null**;
12. **if** (caller != **null**) {
13. **callerApp** = mService.getRecordForAppLocked(caller);
14. **if** (callerApp != **null**) {
15. callingPid = callerApp.pid;
16. callingUid = callerApp.info.uid;
17. }
18. }
19. ActivityRecord sourceRecord = **null**;
20. ActivityRecord resultRecord = **null**;
21. **if** (resultTo != **null**) {
22. **int** index = indexOfTokenLocked(resultTo);
23. **if** (index >= 0) {
24. sourceRecord = (ActivityRecord)mHistory.get(index);
25. **if** (requestCode >= 0 && !sourceRecord.finishing) {
26. ......
27. }
28. }
29. }
31. **int** launchFlags = intent.getFlags();
33. **if** ((launchFlags&Intent.FLAG\_ACTIVITY\_FORWARD\_RESULT) != 0
34. && sourceRecord != **null**) {
35. ......
36. }
38. **if** (err == START\_SUCCESS && intent.getComponent() == **null**) {
39. ......
40. }
42. **if** (err == START\_SUCCESS && aInfo == **null**) {
43. ......
44. }
46. **if** (err != START\_SUCCESS) {
47. ......
48. }
49. ActivityRecord r = **new** ActivityRecord(mService, **this**, callerApp, callingUid,
50. intent, resolvedType, aInfo, mService.mConfiguration,
51. resultRecord, resultWho, requestCode, componentSpecified);
52. **return** **startActivityUncheckedLocked**(r, sourceRecord,
53. grantedUriPermissions, grantedMode, onlyIfNeeded, **true**);
54. }
55. }

从传进来的参数caller得到调用者的进程信息，并保存在callerApp变量中，这里就是Launcher应用程序的进程信息了。前面说过，参数resultTo是Launcher这个Activity里面的一个Binder对象，通过它可以获得Launcher这个Activity的相关信息，保存在sourceRecord变量中。再接下来，创建即将要启动的Activity的相关信息，并保存在r变量中：

1. ActivityRecord r = **new** ActivityRecord(mService, **this**, callerApp, callingUid,
2. intent, resolvedType, aInfo, mService.mConfiguration,
3. resultRecord, resultWho, requestCode, componentSpecified);

接着调用startActivityUncheckedLocked函数进行下一步操作。

### Step 9. ActivityStarter.startActivityUncheckedLocked

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **final** **int** startActivityUncheckedLocked(ActivityRecord r,
3. ActivityRecord sourceRecord, Uri[] grantedUriPermissions,
4. **int** grantedMode, **boolean** onlyIfNeeded, **boolean** doResume) {
5. **final** Intent intent = r.intent;
6. **final** **int** callingUid = r.launchedFromUid;
7. **int** launchFlags = intent.getFlags();
8. // We'll invoke onUserLeaving before onPause only if the launching
9. // activity did not explicitly state that this is an automated launch.
10. mUserLeaving = (launchFlags&Intent.FLAG\_ACTIVITY\_NO\_USER\_ACTION) == 0;
11. ActivityRecord notTop = (launchFlags&Intent.FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP)
12. != 0 ? r : **null**;
13. // If the onlyIfNeeded flag is set, then we can do this if the activity
14. // being launched is the same as the one making the call...  or, as
15. // a special case, if we do not know the caller then we count the
16. // current top activity as the caller.
17. **if** (onlyIfNeeded) {
18. ......
19. }
21. **if** (sourceRecord == **null**) {
22. ......
23. } **else** **if** (sourceRecord.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
24. ......
25. } **else** **if** (r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
26. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK) {
27. ......
28. }
30. **if** (r.resultTo != **null** && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
31. ......
32. }
34. **boolean** addingToTask = **false**;
35. **if** (((**launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK)** != 0 &&
36. (launchFlags&Intent.**FLAG\_ACTIVITY\_MULTIPLE\_TASK**) == 0)
37. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK
38. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
39. // If bring to front is requested, and no result is requested, and
40. // we can find a task that was started with this same
41. // component, then instead of launching bring that one to the front.
42. **if** (r.resultTo == **null**) {
43. // See if there is a task to bring to the front.  If this is
44. // a SINGLE\_INSTANCE activity, there can be one and only one
45. // instance of it in the history, and it is always in its own
46. // unique task, so we do a special search.
47. ActivityRecord taskTop = r.launchMode != ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
48. ? findTaskLocked(intent, r.info)
49. : findActivityLocked(intent, r.info);
50. **if** (taskTop != **null**) {
51. ......
52. }
53. }
54. }
55. **if** (r.packageName != **null**) {
56. // If the activity being launched is the same as the one currently
57. // at the top, then we need to check if it should only be launched
58. // once.
59. ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);
60. **if** (top != **null** && r.resultTo == **null**) {
61. **if** (top.realActivity.equals(r.realActivity)) {
62. ......
63. }
64. }
66. }
68. **boolean** newTask = **false**;
70. // Should this be considered a new task?
71. **if** (r.resultTo == **null** && !addingToTask
72. && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
73. // todo: should do better management of integers.
74. mService.mCurTask++;
75. **if** (mService.mCurTask <= 0) {
76. mService.mCurTask = 1;
77. }
78. r.task = **new** TaskRecord(mService.mCurTask, r.info, intent,
79. (r.info.flags&ActivityInfo.FLAG\_CLEAR\_TASK\_ON\_LAUNCH) != 0);
80. ......
81. newTask = **true**;
82. **if** (mMainStack) {
83. mService.addRecentTaskLocked(r.task);
84. }
85. }
86. **startActivityLocked**(r, newTask, doResume);
87. **return** START\_SUCCESS;
88. }
89. }

函数首先获得intent的标志值，保存在launchFlags变量中。

这个intent的标志值的位Intent.FLAG\_ACTIVITY\_NO\_USER\_ACTION没有置位，因此 ，成员变量mUserLeaving的值为true。

这个intent的标志值的位Intent.FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP也没有置位，因此，变量notTop的值为null。

由于在这个例子的AndroidManifest.xml文件中，MainActivity没有配置launchMode属值，因此，这里的r.launchMode为默认值0，表示以标准（Standard，或者称为ActivityInfo.LAUNCH\_MULTIPLE）的方式来启动这个Activity。Activity的启动方式有四种，其余三种分别是ActivityInfo.LAUNCH\_SINGLE\_INSTANCE、ActivityInfo.LAUNCH\_SINGLE\_TASK和ActivityInfo.LAUNCH\_SINGLE\_TOP，具体可以参考官方网站<http://developer.android.com/reference/android/content/pm/ActivityInfo.html>。

传进来的参数r.resultTo为null，表示Launcher不需要等这个即将要启动的MainActivity的执行结果。

由于这个intent的标志值的位Intent.FLAG\_ACTIVITY\_NEW\_TASK被置位，而且Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK没有置位，因此，下面的if语句会被执行：

1. **if** (((launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0 &&
2. (launchFlags&Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK) == 0)
3. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK
4. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
5. // If bring to front is requested, and no result is requested, and
6. // we can find a task that was started with this same
7. // component, then instead of launching bring that one to the front.
8. **if** (r.resultTo == **null**) {
9. // See if there is a task to bring to the front.  If this is
10. // a SINGLE\_INSTANCE activity, there can be one and only one
11. // instance of it in the history, and it is always in its own
12. // unique task, so we do a special search.
13. ActivityRecord taskTop = r.launchMode != ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
14. ? findTaskLocked(intent, r.info)
15. : findActivityLocked(intent, r.info);
16. **if** (taskTop != **null**) {
17. ......
18. }
19. }
20. }

这段代码的逻辑是查看一下，当前有没有Task可以用来执行这个Activity。由于r.launchMode的值不为ActivityInfo.LAUNCH\_SINGLE\_INSTANCE，因此，它通过findTaskLocked函数来查找存不存这样的Task，这里返回的结果是null，即taskTop为null，因此，需要创建一个新的Task来启动这个Activity。

接着往下看：

1. **if** (r.packageName != **null**) {
2. // If the activity being launched is the same as the one currently
3. // at the top, then we need to check if it should only be launched
4. // once.
5. ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);
6. **if** (top != **null** && r.resultTo == **null**) {
7. **if** (top.realActivity.equals(r.realActivity)) {
8. ......
9. }
10. }
12. }

这段代码的逻辑是看一下，当前在堆栈顶端的Activity是否就是即将要启动的Activity，有些情况下，如果即将要启动的Activity就在堆栈的顶端，那么，就不会重新启动这个Activity的别一个实例了，具体可以参考官方网站http://developer.android.com/reference/android/content/pm/ActivityInfo.html。现在处理堆栈顶端的Activity是Launcher，与我们即将要启动的MainActivity不是同一个Activity，因此，这里不用进一步处理上述介绍的情况。

执行到这里，我们知道，要在一个新的Task里面来启动这个Activity了，于是新创建一个Task：

1. **if** (r.resultTo == **null** && !addingToTask
2. && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
3. // todo: should do better management of integers.
4. mService.mCurTask++;
5. **if** (mService.mCurTask <= 0) {
6. mService.mCurTask = 1;
7. }
8. r.task = **new** TaskRecord(mService.mCurTask, r.info, intent,
9. (r.info.flags&ActivityInfo.FLAG\_CLEAR\_TASK\_ON\_LAUNCH) != 0);
10. ......
11. newTask = **true**;
12. **if** (mMainStack) {
13. mService.addRecentTaskLocked(r.task);
14. }
16. }

新建的Task保存在r.task域中，同时，添加到mService中去，这里的mService就是AtMS了。

最后就进入startActivityLocked(r, newTask, doResume)进一步处理了。这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **private** **final** **void** **startActivityLocked**(ActivityRecord r, **boolean** newTask,
3. **boolean** doResume) {
4. **final** **int** NH = mHistory.size();
5. **int** addPos = -1;
6. **if** (!newTask) {
7. ......
8. }
9. // Place a new activity at top of stack, so it is next to interact
10. // with the user.
11. **if** (addPos < 0) {
12. addPos = NH;
13. }
14. // If we are not placing the new activity frontmost, we do not want
15. // to deliver the onUserLeaving callback to the actual frontmost
16. // activity
17. **if** (addPos < NH) {
18. ......
19. }
21. // Slot the activity into the history stack and proceed
22. mHistory.add(addPos, r);
23. r.inHistory = **true**;
24. r.frontOfTask = newTask;
25. r.task.numActivities++;
26. **if** (NH > 0) {
27. // We want to show the starting preview window if we are
28. // switching to a new task, or the next activity's process is
29. // not currently running.
30. ......
31. } **else** {
32. // If this is the first activity, don't do any fancy animations,
33. // because there is nothing for it to animate on top of.
34. ......
35. }
36. **if** (doResume) {
37. resumeTopActivityLocked(**null**);
38. }
39. }
40. }

这里的NH表示当前系统中历史任务的个数，这里肯定是大于0，因为Launcher已经跑起来了。当NH>0时，并且现在要切换新任务时，要做一些任务切的界面操作，这段代码我们就不看了，这里不会影响到下面启Activity的过程，有兴趣的读取可以自己研究一下。

这里传进来的参数doResume为true，于是调用resumeTopActivityLocked进一步操作。

### Step 10. ActivityStack.resumeTopActivityLocked

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. /\*\*
3. \* Ensure that the top activity in the stack is resumed.
4. \*
5. \* @param prev The previously resumed activity, for when in the process
6. \* of pausing; can be null to call from elsewhere.
7. \*
8. \* @return Returns true if something is being resumed, or false if
9. \* nothing happened.
10. \*/
11. **final** **boolean** resumeTopActivityLocked(ActivityRecord prev) {
12. // Find the first activity that is not finishing.
13. ActivityRecord next = topRunningActivityLocked(**null**);
15. // Remember how we'll process this pause/resume situation, and ensure
16. // that the state is reset however we wind up proceeding.
17. **final** **boolean** userLeaving = mUserLeaving;
18. mUserLeaving = **false**;
20. **if** (next == **null**) {
21. ......
22. }
24. next.delayedResume = **false**;
26. // If the top activity is the resumed one, nothing to do.
27. **if** (mResumedActivity == next && next.state == ActivityState.RESUMED) {
28. ......
29. }
31. // If we are sleeping, and there is no resumed activity, and the top
32. // activity is paused, well that is the state we want.
33. **if** ((mService.mSleeping || mService.mShuttingDown)
34. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
35. ......
36. }
38. ......
40. // If we are currently pausing an activity, then don't do anything
41. // until that is done.
42. **if** (mPausingActivity != **null**) {
43. ......
44. }
46. ......
48. // We need to start pausing the current activity so the top one
49. // can be resumed...
50. **if** (mResumedActivity != **null**) {
51. ......
52. startPausingLocked(userLeaving, **false**);
53. **return** **true**;
54. }
56. ......
57. }
59. ......
61. }

函数先通过调用topRunningActivityLocked函数获得堆栈顶端的Activity，这里就是**MainActivity**了，这是在上面的Step 9设置好的，保存在next变量中。

接下来把mUserLeaving的保存在本地变量userLeaving中，然后重新设置为false，在上面的Step 9中，mUserLeaving的值为true，因此，这里的userLeaving为true。

这里的mResumedActivity为Launcher，因为Launcher是当前正被执行的Activity。

当我们处理休眠状态时，mLastPausedActivity保存堆栈顶端的Activity，因为当前不是休眠状态，所以mLastPausedActivity为null。

有了这些信息之后，下面的语句就容易理解了：

1. // If the top activity is the resumed one, nothing to do.
2. **if** (mResumedActivity == next && next.state == ActivityState.RESUMED) {
3. ......
4. }
6. // If we are sleeping, and there is no resumed activity, and the top
7. // activity is paused, well that is the state we want.
8. **if** ((mService.mSleeping || mService.mShuttingDown)
9. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
10. ......
11. }

就可以了；否则再看一下系统当前是否休眠状态，如果是的话，再看看要启动的Activity是否就是当前处于堆栈顶端的Activity，如果是的话，也是什么都不用做。

上面两个条件都不满足，因此，在继续往下执行之前，首先要把当处于Resumed状态的Activity推入Paused状态，然后才可以启动新的Activity。但是在将当前这个Resumed状态的Activity推入Paused状态之前，首先要看一下当前是否有Activity正在进入Pausing状态，如果有的话，当前这个Resumed状态的Activity就要稍后才能进入Paused状态了，这样就保证了所有需要进入Paused状态的Activity串行处理。

这里没有处于Pausing状态的Activity，即mPausingActivity为null，而且mResumedActivity也不为null，于是就调用startPausingLocked函数把Launcher推入Paused状态去了。

### Step 12. ApplicationThreadProxy.schedulePauseActivity

这个函数定义在frameworks/base/core/java/android/app/ApplicationThreadNative.java文件中：

1. **class** ApplicationThreadProxy **implements** IApplicationThread {
3. ......
5. **public** **final** **void** schedulePauseActivity(IBinder token, **boolean** finished,
6. **boolean** userLeaving, **int** configChanges) **throws** RemoteException {
7. Parcel data = Parcel.obtain();
8. data.writeInterfaceToken(IApplicationThread.descriptor);
9. data.writeStrongBinder(token);
10. data.writeInt(finished ? 1 : 0);
11. data.writeInt(userLeaving ? 1 :0);
12. data.writeInt(configChanges);
13. mRemote.transact(SCHEDULE\_PAUSE\_ACTIVITY\_TRANSACTION, data, **null**,
14. IBinder.FLAG\_ONEWAY);
15. data.recycle();
16. }
18. ......
20. }

这个函数通过Binder进程间通信机制进入到ApplicationThread.schedulePauseActivity函数中。

## L进程

### Step 13. ApplicationThread.schedulePauseActivity

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中，它是ActivityThread的内部类：

1. **public** **final** **class** ActivityThread {
2. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
3. **public** **final** **void** schedulePauseActivity(IBinder token, **boolean** finished,
4. **boolean** userLeaving, **int** configChanges) {
5. queueOrSendMessage(
6. finished ? H.PAUSE\_ACTIVITY\_FINISHING : H.PAUSE\_ACTIVITY,
7. token,
8. (userLeaving ? 1 : 0),
9. configChanges);
10. }
11. }
12. }

这里调用的函数queueOrSendMessage是ActivityThread类的成员函数。

上面说到，这里的finished值为false，因此，queueOrSendMessage的第一个参数值为H.PAUSE\_ACTIVITY，表示要暂停token所代表的Activity，即Launcher。

### Step 14. ActivityThread.queueOrSendMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** queueOrSendMessage(**int** what, Object obj, **int** arg1) {
3. queueOrSendMessage(what, obj, arg1, 0);
4. }
5. **private** **final** **void** queueOrSendMessage(**int** what, Object obj, **int** arg1, **int** arg2) {
6. **synchronized** (**this**) {
7. ......
8. Message msg = Message.obtain();
9. msg.what = what;
10. msg.obj = obj;
11. msg.arg1 = arg1;
12. msg.arg2 = arg2;
13. mH.sendMessage(msg);
14. }
15. }

这里首先将相关信息组装成一个msg，然后通过mH成员变量发送出去，mH的类型是H，继承于Handler类，是ActivityThread的内部类，因此，这个消息最后由H.handleMessage来处理。

### Step 15. H.handleMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
2. **private** **final** **class** H **extends** Handler {
3. **public** **void** handleMessage(Message msg) {
4. **switch** (msg.what) {
5. **case** PAUSE\_ACTIVITY:
6. handlePauseActivity((IBinder)msg.obj, **false**, msg.arg1 != 0, msg.arg2);
7. maybeSnapshot();
8. **break**;
9. }

这里调用ActivityThread.handlePauseActivity进一步操作，msg.obj是一个ActivityRecord对象的引用，它代表的是Launcher这个Activity。

### Step 16. ActivityThread.handlePauseActivity

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** handlePauseActivity(IBinder token, **boolean** finished,
3. **boolean** userLeaving, **int** configChanges) {
4. ActivityClientRecord r = mActivities.get(token);
5. **if** (r != **null**) {
6. //Slog.v(TAG, "userLeaving=" + userLeaving + " handling pause of " + r);
7. **if** (userLeaving) {
8. performUserLeavingActivity(r);
9. }
10. r.activity.mConfigChangeFlags |= configChanges;
11. Bundle state = **performPauseActivity**(token, finished, **true**);
12. // Make sure any pending writes are now committed.
13. QueuedWork.waitToFinish();
14. // Tell the activity manager we have paused.
15. **try** {
16. ActivityManagerNative.getDefault().**activityPaused**(token, state);
17. } **catch** (RemoteException ex) {
18. }
19. }
20. }
21. }

函数首先将Binder引用token转换成ActivityRecord的远程接口ActivityClientRecord，然后做了三个事情：1. 如果userLeaving为true，则通过调用performUserLeavingActivity函数来调用Activity.onUserLeaveHint通知Activity，用户要离开它了；2. 调用**performPauseActivity函数来调用Activity.onPause函数**，我们知道，在Activity的生命周期中，当它要让位于其它的Activity时，系统就会调用它的onPause函数；3. 它通知AtMS，这个Activity已经进入Paused状态了，AtMS现在可以完成未竟的事情，即启动MainActivity了。

### Step 17. ActivityManagerProxy.activityPaused

1. **class** ActivityManagerProxy **implements** IActivityManager
2. {
3. ......
5. **public** **void** activityPaused(IBinder token, Bundle state) **throws** RemoteException
6. {
7. Parcel data = Parcel.obtain();
8. Parcel reply = Parcel.obtain();
9. data.writeInterfaceToken(IActivityManager.descriptor);
10. data.writeStrongBinder(token);
11. data.writeBundle(state);
12. mRemote.transact(ACTIVITY\_PAUSED\_TRANSACTION, data, reply, 0);
13. reply.readException();
14. data.recycle();
15. reply.recycle();
16. }
18. ......
20. }

这里通过Binder进程间通信机制就进入到AtMS.activityPaused函数中去了。

## AMS

### Step 18. AtMS.activityPaused

这个函数定义在frameworks/base/services/java/com/android/server/am/AtMS.java文件中：

@Override  
**public final void** activityPaused(IBinder token) {  
 **final long** origId = Binder.*clearCallingIdentity*();  
 **synchronized**(**this**) {  
 ActivityStack stack = ActivityRecord.*getStackLocked*(token);  
 **if** (stack != **null**) {  
 stack.activityPausedLocked(token, **false**);  
 }  
 }  
 Binder.*restoreCallingIdentity*(origId);  
}

这里，又再次进入到ActivityStack类中，执行activityPaused函数。

这里判断app切换就好了。

stack.**mPausingActivity**.processName

### Step 19. ActivityStack.activityPaused

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **final** **void** activityPaused(IBinder token, Bundle icicle, **boolean** timeout) {
3. ActivityRecord r = **null**;
4. **synchronized** (mService) {
5. **int** index = indexOfTokenLocked(token);
6. **if** (index >= 0) {
7. r = (ActivityRecord)mHistory.get(index);
8. **if** (!timeout) {
9. r.icicle = icicle;
10. r.haveState = **true**;
11. }
12. mHandler.removeMessages(PAUSE\_TIMEOUT\_MSG, r);
13. **if** (mPausingActivity == r) {
14. r.state = ActivityState.PAUSED;
15. completePauseLocked();
16. } **else** {
17. ......
18. }
19. }
20. }
21. }
22. }

这里通过参数token在mHistory列表中得到ActivityRecord，从上面我们知道，这个ActivityRecord代表的是Launcher这个Activity，而我们在Step 11中，把Launcher这个Activity的信息保存在mPausingActivity中，因此，这里mPausingActivity等于r，于是，执行completePauseLocked操作。

### Step 20. ActivityStack.completePauseLocked

1. **public** **class** ActivityStack {
2. **private** **final** **void** completePauseLocked() {
3. ActivityRecord prev = mPausingActivity;
4. **if** (prev != **null**) {
5. mPausingActivity = **null**;
6. }
7. **if** (!mService.mSleeping && !mService.mShuttingDown) {
8. resumeTopActivityLocked(prev);
9. } **else** {
10. }
11. }
12. }

函数首先把mPausingActivity变量清空，因为现在不需要它了，然后调用resumeTopActivityLokced进一步操作，它传入的参数即为代表Launcher这个Activity的ActivityRecord。

### Step 21. ActivityStack.resumeTopActivityLokced

1. **public** **class** ActivityStack {
2. **final** **boolean** resumeTopActivityLocked(ActivityRecord prev) {
3. // Find the first activity that is not finishing.
4. ActivityRecord next = topRunningActivityLocked(**null**);
6. // Remember how we'll process this pause/resume situation, and ensure
7. // that the state is reset however we wind up proceeding.
8. **final** **boolean** userLeaving = mUserLeaving;
9. mUserLeaving = **false**;
10. next.delayedResume = **false**;
12. // If the top activity is the resumed one, nothing to do.
13. **if** (mResumedActivity == next && next.state == ActivityState.RESUMED) {
14. ......
15. **return** **false**;
16. }
18. // If we are sleeping, and there is no resumed activity, and the top
19. // activity is paused, well that is the state we want.
20. **if** ((mService.mSleeping || mService.mShuttingDown)
21. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
22. ......
23. **return** **false**;
24. }
26. .......

29. // We need to start pausing the current activity so the top one
30. // can be resumed...
31. **if** (mResumedActivity != **null**) {
32. ......
33. **return** **true**;
34. }
35. **if** (next.app != **null** && next.app.thread != **null**) {
36. } **else** {
37. startSpecificActivityLocked(next, **true**, **true**);
38. }
40. **return** **true**;
41. }
42. }

通过上面的Step 9，我们知道，**当前在堆栈顶端的Activity为我们即将要启动的MainActivity**，这里通过调用topRunningActivityLocked将它取回来，保存在next变量中。之前最后一个Resumed状态的Activity，即Launcher，到了这里已经处于Paused状态了，因此，mResumedActivity为null。最后一个处于Paused状态的Activity为Launcher，因此，这里的mLastPausedActivity就为Launcher。前面我们为MainActivity创建了ActivityRecord后，它的app域一直保持为null。有了这些信息后，上面这段代码就容易理解了，它最终调用startSpecificActivityLocked进行下一步操作。

### Step 22. ActivityStack.startSpecificActivityLocked

1. **public** **class** ActivityStack {
2. **private** **final** **void** startSpecificActivityLocked(ActivityRecord r,
3. **boolean** andResume, **boolean** checkConfig) {
4. // Is this activity's application already running?
5. ProcessRecord app = mService.getProcessRecordLocked(r.processName,
6. r.info.applicationInfo.uid);
7. **if** (app != **null** && app.thread != **null**) {
8. **try** {
9. realStartActivityLocked(r, app, andResume, checkConfig);
10. **return**;
11. } **catch** (RemoteException e) {
12. ......
13. }
14. }
16. mService.startProcessLocked(r.processName, r.info.applicationInfo, **true**, 0,
17. "activity", r.intent.getComponent(), **false**);
18. }
19. }

注意，这里由于是第一次启动应用程序的Activity，所以下面语句：

ProcessRecord app = mService.getProcessRecordLocked(r.processName, r.info.applicationInfo.uid); 取回来的app为null。

在Activity应用程序中的AndroidManifest.xml配置文件中，我们没有指定Application标签的process属性，系统就会默认使用package的名称，这里就是"k.demo.system"了。每一个应用程序都有自己的uid，因此，这里uid + process的组合就可以为每一个应用程序创建一个ProcessRecord。当然，我们可以配置两个应用程序具有相同的uid和package，或者在AndroidManifest.xml配置文件的application标签或者activity标签中显式指定相同的process属性值，这样，不同的应用程序也可以在同一个进程中启动。

函数最终执行AtMS.startProcessLocked函数进行下一步操作。

### Step 23. AtMS.startProcessLocked

这个函数定义在frameworks/base/services/java/com/android/server/am/AtMS.java文件中：

1. **final** ProcessRecord startProcessLocked(String processName,
2. ApplicationInfo info, **boolean** knownToBeDead, **int** intentFlags,
3. String hostingType, ComponentName hostingName, **boolean** allowWhileBooting) {
4. ProcessRecord app = getProcessRecordLocked(processName, info.uid);
5. String hostingNameStr = hostingName != **null**
6. ? hostingName.flattenToShortString() : **null**;
7. **if** (app == **null**) {
8. app = **new** ProcessRecordLocked(**null**, info, processName);
9. mProcessNames.put(processName, info.uid, app);
10. } **else** {
11. // If this is a new package in the process, add the package to the list
12. app.addPackage(info.packageName);
13. }
14. startProcessLocked(app, hostingType, hostingNameStr);
15. **return** (app.pid != 0) ? app : **null**;
16. }
17. }

这里再次检查是否已经有以process + uid命名的进程存在，在我们这个情景中，返回值app为null，因此，后面会创建一个ProcessRecord，并存保存在成员变量mProcessNames中，最后，调用另一个startProcessLocked函数进一步操作：

1. **private** **final** **void** startProcessLocked(ProcessRecord app,
2. String hostingType, String hostingNameStr) {
3. **if** (entryPoint == **null**) entryPoint = **"android.app.ActivityThread"**;  
   Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, **"Start proc: "** +  
    app.processName);  
   checkTime(startTime, **"startProcess: asking zygote to start proc"**);  
   ProcessStartResult startResult;  
   **if** (hostingType.equals(**"webview\_service"**)) {  
    startResult = startWebView(entryPoint,  
    app.processName, uid, uid, gids, debugFlags, mountExternal,  
    app.info.targetSdkVersion, seInfo, requiredAbi, instructionSet,  
    app.info.dataDir, **null**, entryPointArgs);  
   } **else** {  
    startResult = Process.start(entryPoint,  
    app.processName, uid, uid, gids, debugFlags, mountExternal,  
    app.info.targetSdkVersion, seInfo, requiredAbi, instructionSet,  
    app.info.dataDir, invokeWith, entryPointArgs);  
   }
5. }

#### Process.start

这里主要是调用**Process.start**接口来创建一个新的进程，新的进程会导入android.app.ActivityThread类，并且执行它的main函数，这就是为什么我们前面说每一个应用程序都有一个ActivityThread实例来对应的原因。

Process.ProcessStartResult startResult = Process.start(entryPoint,

app.processName, uid, uid, gids, debugFlags, mountExternal,

app.info.targetSdkVersion, app.info.seinfo, requiredAbi, instructionSet,

app.info.dataDir, entryPointArgs);

### Step 24. ActivityThread.main

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** attach(**boolean** system) {
3. mSystemThread = system;
4. **if** (!system) {
5. IActivityManager mgr = ActivityManagerNative.getDefault();
6. **try** {
7. mgr.attachApplication(mAppThread);
8. } **catch** (RemoteException ex) {
9. }
10. } **else** {
11. }
12. }
13. **public** **static** **final** **void** main(String[] args) {
14. ActivityThread thread = **new** ActivityThread();
15. thread.attach(**false**);
16. Looper.loop();
17. thread.detach();
18. }
19. }

这个函数在进程中创建一个ActivityThread实例，然后调用它的attach函数，接着就进入消息循环了，直到最后进程退出。

函数attach最终调用了AtMS的远程接口ActivityManagerProxy的attachApplication函数，传入的参数是mAppThread，这是一个ApplicationThread类型的Binder对象，它的作用是用来进行进程间通信的。

#### ActivityThread. Attach

Asf

**private void** attach(**boolean** system) {  
 sCurrentActivityThread = **this**;  
 mSystemThread = system;  
 **if** (!system) {  
 ViewRootImpl.addFirstDrawHandler(**new** Runnable() {  
 @Override  
 **public void** run() {  
 ensureJitEnabled();  
 }  
 });  
 android.ddm.DdmHandleAppName.setAppName(**"<pre-initialized>"**,  
 UserHandle.myUserId());  
 RuntimeInit.setApplicationObject(mAppThread.asBinder());  
 **final** IActivityManager mgr = ActivityManager.getService();  
 **try** {  
 mgr.attachApplication(mAppThread);  
 } **catch** (RemoteException ex) {  
 **throw** ex.rethrowFromSystemServer();  
 }  
 *// Watch for getting close to heap limit.* BinderInternal.addGcWatcher(**new** Runnable() {  
 @Override **public void** run() {  
 **if** (!mSomeActivitiesChanged) {  
 **return**;  
 }  
 Runtime runtime = Runtime.getRuntime();  
 **long** dalvikMax = runtime.maxMemory();  
 **long** dalvikUsed = runtime.totalMemory() - runtime.freeMemory();  
 **if** (dalvikUsed > ((3\*dalvikMax)/4)) {  
 **if** (DEBUG\_MEMORY\_TRIM) Slog.d(TAG, **"Dalvik max="** + (dalvikMax/1024)  
 + **" total="** + (runtime.totalMemory()/1024)  
 + **" used="** + (dalvikUsed/1024));  
 mSomeActivitiesChanged = **false**;

/\*\*

这里添加Gc的监听，如果超过虚拟机分配最大内存的 3/4，那么触发mgr.releaseSomeActivities

\*/

**try** {  
 mgr.releaseSomeActivities(mAppThread);  
 } **catch** (RemoteException e) {  
 **throw** e.rethrowFromSystemServer();  
 }  
 }  
 }  
 });  
 } **else** {  
 *// Don't set application object here -- if the system crashes,  
 // we can't display an alert, we just want to die die die.* android.ddm.DdmHandleAppName.setAppName(**"system\_process"**,  
 UserHandle.myUserId());  
 **try** {  
 mInstrumentation = **new** Instrumentation();  
 ContextImpl context = ContextImpl.createAppContext(  
 **this**, getSystemContext().mPackageInfo);  
 mInitialApplication = context.mPackageInfo.makeApplication(**true**, **null**);  
 mInitialApplication.onCreate();  
 } **catch** (Exception e) {  
 **throw new** RuntimeException(  
 **"Unable to instantiate Application():"** + e.toString(), e);  
 }  
 }  
  
 *// add dropbox logging to libcore* DropBox.setReporter(**new** DropBoxReporter());  
  
 ViewRootImpl.ConfigChangedCallback configChangedCallback  
 = (Configuration globalConfig) -> {  
 **synchronized** (mResourcesManager) {  
 *// We need to apply this change to the resources immediately, because upon returning  
 // the view hierarchy will be informed about it.* **if** (mResourcesManager.applyConfigurationToResourcesLocked(globalConfig,  
 **null** */\* compat \*/*)) {  
 updateLocaleListFromAppContext(mInitialApplication.getApplicationContext(),  
 mResourcesManager.getConfiguration().getLocales());  
  
 *// This actually changed the resources! Tell everyone about it.* **if** (mPendingConfiguration == **null** || mPendingConfiguration.isOtherSeqNewer(globalConfig)) {  
 mPendingConfiguration = globalConfig;  
 sendMessage(H.CONFIGURATION\_CHANGED, globalConfig);  
 }  
 }  
 }  
 };  
 ViewRootImpl.addConfigCallback(configChangedCallback);  
}

 我们很明显的看到了一个 BinderInternal.addGcWatcher()这个方法，我们可以看字面意思就知道，这个是一个Gc 回收的监听器，作用是在AppThread 作用域下的GcRoots Gc开始的时候，我们去监听这个GC回收，具体如何做到呢

### Step 25. ActivityManagerProxy.attachApplication

这个函数定义在frameworks/base/core/java/android/app/ActivityManagerNative.java文件中：

1. **class** ActivityManagerProxy **implements** IActivityManager
2. {
3. **public** **void** attachApplication(IApplicationThread app) **throws** RemoteException
4. {
5. Parcel data = Parcel.obtain();
6. Parcel reply = Parcel.obtain();
7. data.writeInterfaceToken(IActivityManager.descriptor);
8. data.writeStrongBinder(app.asBinder());
9. mRemote.transact(ATTACH\_APPLICATION\_TRANSACTION, data, reply, 0);
10. reply.readException();
11. data.recycle();
12. reply.recycle();
13. }
14. }

这里通过Binder驱动程序，最后进入AtMS的attachApplication函数中。

## AMS

### Step 26. AtMS.attachApplication

1. **public** **final** **class** AtMS **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
3. **public** **final** **void** attachApplication(IApplicationThread thread) {
4. **synchronized** (**this**) {
5. **int** callingPid = Binder.getCallingPid();
6. **final** **long** origId = Binder.clearCallingIdentity();
7. attachApplicationLocked(thread, callingPid);
8. Binder.restoreCallingIdentity(origId);
9. }
10. }
11. }

这里将操作转发给attachApplicationLocked函数。

### Step 27. AtMS.attachApplicationLocked

1. **public** **final** **class** AtMS **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
4. ......
6. **private** **final** **boolean** attachApplicationLocked(IApplicationThread thread,
7. **int** pid) {
8. // Find the application record that is being attached...  either via
9. // the pid if we are running in multiple processes, or just pull the
10. // next app record if we are emulating process with anonymous threads.
11. ProcessRecord app;
12. **if** (pid != MY\_PID && pid >= 0) {
13. **synchronized** (mPidsSelfLocked) {
14. app = mPidsSelfLocked.get(pid);
15. }
16. } **else** **if** (mStartingProcesses.size() > 0) {
17. ......
18. } **else** {
19. ......
20. }
22. **if** (app == **null**) {
23. ......
24. **return** **false**;
25. }
27. ......
29. String processName = app.processName;
30. **try** {
31. thread.asBinder().linkToDeath(**new** AppDeathRecipient(
32. app, pid, thread), 0);
33. } **catch** (RemoteException e) {
34. ......
35. **return** **false**;
36. }
38. ......
40. app.thread = thread;
41. app.curAdj = app.setAdj = -100;
42. app.curSchedGroup = Process.THREAD\_GROUP\_DEFAULT;
43. app.setSchedGroup = Process.THREAD\_GROUP\_BG\_NONINTERACTIVE;
44. app.forcingToForeground = **null**;
45. app.foregroundServices = **false**;
46. app.debugging = **false**;
48. ......
50. **boolean** normalMode = mProcessesReady || isAllowedWhileBooting(app.info);
52. ......
54. **boolean** badApp = **false**;
55. **boolean** didSomething = **false**;
57. // See if the top visible activity is waiting to run in this process...
58. ActivityRecord hr = mMainStack.topRunningActivityLocked(**null**);
59. **if** (hr != **null** && normalMode) {
60. **if** (hr.app == **null** && app.info.uid == hr.info.applicationInfo.uid
61. && processName.equals(hr.processName)) {
62. **try** {
63. **if** (mMainStack.realStartActivityLocked(hr, app, **true**, **true**)) {
64. didSomething = **true**;
65. }
66. } **catch** (Exception e) {
67. ......
68. }
69. } **else** {
70. ......
71. }
72. }
74. ......
76. **return** **true**;
77. }
79. ......
81. }

在前面的Step 23中，已经创建了一个ProcessRecord，这里首先通过pid将它取回来，放在app变量中，然后对app的其它成员进行初始化，最后调用mMainStack.realStartActivityLocked执行真正的Activity启动操作。这里要启动的Activity通过调用mMainStack.topRunningActivityLocked(null)从堆栈顶端取回来，这时候在堆栈顶端的Activity就是MainActivity了。

### Step 28. ActivityStackSupervisor.realStartActivityLocked

1. **public** **class** ActivityStack {
3. ......
5. **final** **boolean** realStartActivityLocked(ActivityRecord r,
6. ProcessRecord app, **boolean** andResume, **boolean** checkConfig)
7. **throws** RemoteException {
9. ......
11. r.app = app;
13. ......
15. **int** idx = app.activities.indexOf(r);
16. **if** (idx < 0) {
17. app.activities.add(r);
18. }
20. ......
22. **try** {
23. ......
25. List<ResultInfo> results = **null**;
26. List<Intent> newIntents = **null**;
27. **if** (andResume) {
28. results = r.results;
29. newIntents = r.newIntents;
30. }
32. ......
34. app.thread.scheduleLaunchActivity(**new** Intent(r.intent), r,
35. System.identityHashCode(r),
36. r.info, r.icicle, results, newIntents, !andResume,
37. mService.isNextTransitionForward());
39. ......
41. } **catch** (RemoteException e) {
42. ......
43. }
45. ......
47. **return** **true**;
48. }
50. ......
52. }

这里最终通过app.thread进入到ApplicationThreadProxy的scheduleLaunchActivity函数中，注意，这里的第二个参数r，是一个ActivityRecord类型的Binder对象，用来作来这个Activity的token值。

当框架通过ApplicationThread的代理回调到ActivityThread的时候，将对应的步骤一种生成的token代理传入。 ActivityStackSupervisor.realStartActivityLocked()

### Step 29. ApplicationThreadProxy.scheduleLaunchActivity

这个函数定义在frameworks/base/core/java/android/app/ApplicationThreadNative.java文件中：

1. **class** ApplicationThreadProxy **implements** IApplicationThread {
2. **public** **final** **void** scheduleLaunchActivity(Intent intent, IBinder token, **int** ident,
3. ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
4. List<Intent> pendingNewIntents, **boolean** notResumed, **boolean** isForward)
5. **throws** RemoteException {
6. Parcel data = Parcel.obtain();
7. data.writeInterfaceToken(IApplicationThread.descriptor);
8. intent.writeToParcel(data, 0);
9. data.writeStrongBinder(token);
10. data.writeInt(ident);
11. info.writeToParcel(data, 0);
12. data.writeBundle(state);
13. data.writeTypedList(pendingResults);
14. data.writeTypedList(pendingNewIntents);
15. data.writeInt(notResumed ? 1 : 0);
16. data.writeInt(isForward ? 1 : 0);
17. mRemote.transact(SCHEDULE\_LAUNCH\_ACTIVITY\_TRANSACTION, data, **null**,
18. IBinder.FLAG\_ONEWAY);
19. data.recycle();
20. }
21. }

这个函数最终通过Binder驱动程序进入到ApplicationThread的scheduleLaunchActivity函数中。

### Step 30. ApplicationThread.scheduleLaunchActivity

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
7. ......
9. // we use token to identify this activity without having to send the
10. // activity itself back to the activity manager. (matters more with ipc)
11. **public** **final** **void** scheduleLaunchActivity(Intent intent, IBinder token, **int** ident,
12. ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
13. List<Intent> pendingNewIntents, **boolean** notResumed, **boolean** isForward) {
14. ActivityClientRecord r = **new** ActivityClientRecord();
16. r.token = token;
17. r.ident = ident;
18. r.intent = intent;
19. r.activityInfo = info;
20. r.state = state;
22. r.pendingResults = pendingResults;
23. r.pendingIntents = pendingNewIntents;
25. r.startsNotResumed = notResumed;
26. r.isForward = isForward;
28. queueOrSendMessage(H.LAUNCH\_ACTIVITY, r);
29. }
31. ......
33. }
35. ......
36. }

函数首先创建一个ActivityClientRecord实例，并且初始化它的成员变量，然后调用ActivityThread类的queueOrSendMessage函数进一步处理。

### Step 31. ActivityThread.queueOrSendMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
7. ......
9. // if the thread hasn't started yet, we don't have the handler, so just
10. // save the messages until we're ready.
11. **private** **final** **void** queueOrSendMessage(**int** what, Object obj) {
12. queueOrSendMessage(what, obj, 0, 0);
13. }
15. ......
17. **private** **final** **void** queueOrSendMessage(**int** what, Object obj, **int** arg1, **int** arg2) {
18. **synchronized** (**this**) {
19. ......
20. Message msg = Message.obtain();
21. msg.what = what;
22. msg.obj = obj;
23. msg.arg1 = arg1;
24. msg.arg2 = arg2;
25. mH.sendMessage(msg);
26. }
27. }
29. ......
31. }
33. ......
34. }

函数把消息内容放在msg中，然后通过mH把消息分发出去，这里的成员变量mH我们在前面已经见过，消息分发出去后，最后会调用H类的handleMessage函数。

### Step 32. H.handleMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** H **extends** Handler {
7. ......
9. **public** **void** handleMessage(Message msg) {
10. ......
11. **switch** (msg.what) {
12. **case** LAUNCH\_ACTIVITY: {
13. ActivityClientRecord r = (ActivityClientRecord)msg.obj;
15. r.packageInfo = getPackageInfoNoCheck(
16. r.activityInfo.applicationInfo);
17. handleLaunchActivity(r, **null**);
18. } **break**;
19. ......
20. }
22. ......
24. }
26. ......
27. }

这里最后调用ActivityThread类的handleLaunchActivity函数进一步处理。

### Step 33. ActivityThread.handleLaunchActivity

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {
3. Activity a = performLaunchActivity(r, customIntent);
4. if (r.window == null && !a.mFinished && willBeVisible) {
5. //获得为当前Activity创建的窗口PhoneWindow对象
6. r.window = r.activity.getWindow();
7. //获取为窗口创建的视图DecorView对象
8. View decor = r.window.getDecorView();
9. decor.setVisibility(View.INVISIBLE);
10. //在attach函数中就为当前Activity创建了WindowManager对象
11. ViewManager wm = a.getWindowManager();
12. //得到该视图对象的布局参数
13. WindowManager.LayoutParams l = r.window.getAttributes();
14. //将视图对象保存到Activity的成员变量mDecor中
15. a.mDecor = decor;
16. l.type = WindowManager.LayoutParams.TYPE\_BASE\_APPLICATION;
17. l.softInputMode |= forwardBit;
18. ......
19. if (a.mVisibleFromClient && !a.mWindowAdded) {
20. a.mWindowAdded = true;
21. //将创建的视图对象DecorView添加到Activity的窗口管理器中
22. wm.addView(decor, l);
23. }
24. ......
25. if (r.activity.mVisibleFromClient) {
26. r.activity.makeVisible();
27. }
28. }
29. **if** (a != **null**) {
30. r.createdConfig = **new** Configuration(mConfiguration);
31. Bundle oldState = r.state;
32. handleResumeActivity(r.token, **false**, r.isForward);
33. }
34. }
35. }

这里首先调用performLaunchActivity函数来加载这个Activity类，即shy.luo.activity.MainActivity，然后调用它的onCreate函数，最后回到handleLaunchActivity函数时，

再调用handleResumeActivity函数来使这个Activity进入Resumed状态，即会调用这个Activity的onResume函数，这是遵循Activity的生命周期的。会调用Activity.makeVisible(),该方法继续调用便会执行到WindowManagerImpl.addView(), 该方法内部再调用WindowManagerGlobal.addView(),

### Step 34. ActivityThread.performLaunchActivity

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
7. ActivityInfo aInfo = r.activityInfo;
8. **if** (r.packageInfo == **null**) {
9. r.packageInfo = getPackageInfo(aInfo.applicationInfo,
10. Context.CONTEXT\_INCLUDE\_CODE);
11. }
13. ComponentName component = r.intent.getComponent();
14. **if** (component == **null**) {
15. component = r.intent.resolveActivity(
16. mInitialApplication.getPackageManager());
17. r.intent.setComponent(component);
18. }
20. **if** (r.activityInfo.targetActivity != **null**) {
21. component = **new** ComponentName(r.activityInfo.packageName,
22. r.activityInfo.targetActivity);
23. }
25. Activity activity = **null**;
26. **try** {
27. java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
28. activity = mInstrumentation.newActivity(
29. cl, component.getClassName(), r.intent);
30. r.intent.setExtrasClassLoader(cl);
31. **if** (r.state != **null**) {
32. r.state.setClassLoader(cl);
33. }
34. } **catch** (Exception e) {
35. ......
36. }
38. **try** {
39. Application app = r.packageInfo.makeApplication(**false**, mInstrumentation);
41. ......
43. **if** (activity != **null**) {
44. ContextImpl appContext = **new** ContextImpl();
45. appContext.init(r.packageInfo, r.token, **this**);
46. appContext.setOuterContext(activity);
47. CharSequence title = r.activityInfo.loadLabel(appContext.getPackageManager());
48. Configuration config = **new** Configuration(mConfiguration);
49. ......
50. activity.attach(appContext, **this**, getInstrumentation(), r.token,
51. r.ident, app, r.intent, r.activityInfo, title, r.parent,
52. r.embeddedID, r.lastNonConfigurationInstance,
53. r.lastNonConfigurationChildInstances, config);
55. **if** (customIntent != **null**) {
56. activity.mIntent = customIntent;
57. }
58. r.lastNonConfigurationInstance = **null**;
59. r.lastNonConfigurationChildInstances = **null**;
60. activity.mStartedActivity = **false**;
61. **int** theme = r.activityInfo.getThemeResource();
62. **if** (theme != 0) {
63. activity.setTheme(theme);
64. }
66. activity.mCalled = **false**;
67. mInstrumentation.callActivityOnCreate(activity, r.state);
68. ......
69. r.activity = activity;
70. r.stopped = **true**;
71. **if** (!r.activity.mFinished) {
72. activity.performStart();
73. r.stopped = **false**;
74. }
75. **if** (!r.activity.mFinished) {
76. **if** (r.state != **null**) {
77. mInstrumentation.callActivityOnRestoreInstanceState(activity, r.state);
78. }
79. }
80. **if** (!r.activity.mFinished) {
81. activity.mCalled = **false**;
82. mInstrumentation.callActivityOnPostCreate(activity, r.state);
83. **if** (!activity.mCalled) {
84. **throw** **new** SuperNotCalledException(
85. "Activity " + r.intent.getComponent().toShortString() +
86. " did not call through to super.onPostCreate()");
87. }
88. }
89. }
90. r.paused = **true**;
92. mActivities.put(r.token, r);
94. } **catch** (SuperNotCalledException e) {
95. ......
97. } **catch** (Exception e) {
98. ......
99. }
101. **return** activity;
102. }
104. ......
105. }

函数前面是收集要启动的Activity的相关信息，主要package和component信息：

1. ActivityInfo aInfo = r.activityInfo;
2. **if** (r.packageInfo == **null**) {
3. r.packageInfo = getPackageInfo(aInfo.applicationInfo,
4. Context.CONTEXT\_INCLUDE\_CODE);
5. }
7. ComponentName component = r.intent.getComponent();
8. **if** (component == **null**) {
9. component = r.intent.resolveActivity(
10. mInitialApplication.getPackageManager());
11. r.intent.setComponent(component);
12. }
14. **if** (r.activityInfo.targetActivity != **null**) {
15. component = **new** ComponentName(r.activityInfo.packageName,
16. r.

#### ClassLoader

将shy.luo.activity.MainActivity类加载进来：

1. Activity activity = **null**;
2. **try** {
3. java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
4. activity = mInstrumentation.newActivity(
5. cl, component.getClassName(), r.intent);
6. r.intent.setExtrasClassLoader(cl);
7. **if** (r.state != **null**) {
8. r.state.setClassLoader(cl);
9. }
10. } **catch** (Exception e) {
11. ......
12. }

#### 创建Application对象

这是根据AndroidManifest.xml配置文件中的Application标签的信息来创建的：

1. Application app = r.packageInfo.makeApplication(**false**, mInstrumentation);

#### 创建Activity的上下文信息

Context appContext = createBaseContextForActivity(r, activity); CharSequence title = r.activityInfo.loadLabel(appContext.getPackageManager());

#### 创建 Configuration

Fgh

Configuration config = new Configuration(mCompatConfiguration);

if (r.overrideConfig != null) {

config.updateFrom(r.overrideConfig);

}

if (DEBUG\_CONFIGURATION) Slog.v(TAG, "Launching activity "

+ r.activityInfo.name + " with config " + config);

Window window = null;

if (r.mPendingRemoveWindow != null && r.mPreserveWindow) {

window = r.mPendingRemoveWindow;

r.mPendingRemoveWindow = null;

r.mPendingRemoveWindowManager = null;

}

#### attach

调用acitvity的attach，将创建的这些对象关联到activity

后面的代码主要创建Activity的上下文信息，并通过attach方法将这些上下文信息设置到MainActivity中去：

1. activity.attach(appContext, **this**, getInstrumentation(), r.token,
2. r.ident, app, r.intent, r.activityInfo, title, r.parent,
3. r.embeddedID, r.lastNonConfigurationInstance,
4. r.lastNonConfigurationChildInstances, config);

最后还要调用MainActivity的onCreate函数：

Instrumentation.callActivityOnCreate(activity, r.state);

这里不是直接调用MainActivity的onCreate函数，而是通过mInstrumentation的callActivityOnCreate函数来间接调用，前面我们说过，mInstrumentation在这里的作用是监控Activity与系统的交互操作，相当于是系统运行日志。

### Step 35. MainActivity.onCreate

1. **public** **class** MainActivity **extends** Activity  **implements** OnClickListener {
3. ......
5. @Override
6. **public** **void** onCreate(Bundle savedInstanceState) {
7. ......
9. Log.i(LOG\_TAG, "Main Activity Created.");
10. }
12. ......

Dfg

handleLaunchActivity()：会调用Activity.onCreate(), 该方法内再调用setContentView(),经过AMS与WMS的各种交互,层层调用后,进入step2

handleResumeActivity()：会调用Activity.makeVisible(),该方法继续调用便会执行到WindowManagerImpl.addView(), 该方法内部再调用WindowManagerGlobal.addView(),

if (app.persistent) {

Watchdog.getInstance().processStarted(app.processName

## 小结

这样，MainActivity就启动起来了，整个应用程序也启动起来了。

       整个应用程序的启动过程要执行很多步骤，但是整体来看，主要分为以下五个阶段：

       一. Step1 - Step 11：Launcher通过Binder进程间通信机制通知AtMS，它要启动一个Activity；

       二. Step 12 - Step 16：AtMS通过Binder进程间通信机制通知Launcher进入Paused状态；

       三. Step 17 - Step 24：Launcher通过Binder进程间通信机制通知AtMS，它已经准备就绪进入Paused状态，于是AtMS就创建一个新的进程，用来启动一个ActivityThread实例，即将要启动的Activity就是在这个ActivityThread实例中运行；

       四. Step 25 - Step 27：ActivityThread通过Binder进程间通信机制将一个ApplicationThread类型的Binder对象传递给AtMS，以便以后AtMS能够通过这个Binder对象和它进行通信；

       五. Step 28 - Step 35：AtMS通过Binder进程间通信机制通知ActivityThread，现在一切准备就绪，它可以真正执行Activity的启动操作了。

       这里不少地方涉及到了Binder进程间通信机制，相关资料请参考[Android进程间通信（IPC）机制Binder简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/6618363)一文。

       这样，应用程序的启动过程就介绍完了，它实质上是启动应用程序的默认Activity，在下一篇文章中，我们将介绍在应用程序内部启动另一个Activity的过程，即新的Activity与启动它的Activity将会在同一个进程（Process）和任务（Task）运行，敬请关注。

1、AMS接到客户端的调用过后，开始启动，把启动的工作交给了ActivityStarter的startActivityMayWait

2、startActivityMayWait分了三步：

2.1>先从PMS中搜索匹配信息

2.2>然后startActivityUnchecked启动activity startActivityUnchecked：处理launchmode、以怎样的task来启动activity、打开动画处理、由resumeTopActivityInnerLocked分两个方向走

2.2.1>>如果mResumedActivity不为空，则需要先暂停这个Activity。由startPausingLocked来完成

2.2.2>>当mResumedActivity为空时，若待启动的Activity对应的应用存在，那么仅需要重新启动该Activity 否则，需要调用ActivityStackSupervisor的startSpecificActivityLocked函数，启动整个进程。

2.2.2.1>>>通知进程启动目标Activity，此流程最后调用到performLaunchActivity直接创建了activity

2.2.2.2>>>如果进程不存在需要创建进程来运行

2.3>最后处理返回值。

联系作者获得授权并注明出处。

## 参考

[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)

[**Android深入四大组件（一）应用程序启动过程**](http://blog.csdn.net/itachi85/article/details/69388942)

<http://blog.csdn.net/itachi85/article/details/69388942>

startService启动过程分析

<http://gityuan.com/2016/03/06/start-service/>

深入理解ActivityManagerService

http://wiki.jikexueyuan.com/project/deep-android-v2/activity.html

# 栈管理

我们知道,activity 在 AMS 中的形式是 ActivityRecord,

task 在 AMS 中的形式为TaskRecord,

进程在 AMS 中的管理形式为 ProcessRecord。

**Android系统界面用户体验组织（TASKS），**，用户体验到的直观界面都是由很多Tasks组成的。比如，我们从最近任务栏，就可以看到很多TASKS。用户可以随意的切换到其中的一个TASK。也可以按HOME键退出一个TASK。TASK的组织形式是ActivityStack，一个ActivityStack是由好多Activity组成的堆栈。从Android系统设计者的角度来看，一个Task定义了一组行为，而这组行为是由跨多个应用程序的多个Activity组织而构成的。这个打破了以往由应用程序（进程）来定义的资源边界。一个Task就是一个场景的实现。从此构建系统用户体验和行为的边界在于TASK，而非应用程序或者进程。

## 概念

### Task

Task 是activities的集合，通过activity stack来管理，依靠先进后出队列来实现;每个task中都至少有一个activity，新实例出来的activity置于栈顶，Task可以被切换到后台

### Activity Stack

如上所诉，Activity承担了大量的显示和交互工作，从某种角度上将，我们看见的应用程序就是许多个Activity的组合。为了让这许多 Activity协同工作而不至于产生混乱，Android平台设计了一种堆栈机制用于管理Activity，其遵循先进后出的原则，系统总是显示位于栈顶的Activity，从逻辑上将，位于栈顶的Activity也就是最后打开的Activity，这也是符合逻辑的。

在操作应用程序时，每次启动新的Activity，都会将此压入Activity Stack，当用户执行返回操作时，移除Activity Stack顶上的Activity，这样就实现了返回上一个Activty的功能。直到用户一直返回到Home Screen，这时候可以理解为移除了Activity Stack所有的Activity，这个Activity Stack不再存在，应用程序也结束了运行。

可以通过 adb shell dumpsys 查看 TASKS的ActivityStacks

### task的taskAffinity

 任务归属，taskAffinity 这个属性主要是决定持有每个activity属于哪个task。 默认情况下，同一个包中的activity共享同一个affinity（任务共用性）。

### **task的launchMode**

 standard(default)：standard，标准的Activity是可以随意插入到TASK中去的一个组织结构，可以去TaskA，也可以去TaskB，也可以去TaskC，直接并无任何的联系

 singleTop，如果在任务的栈顶正好存在该Activity的实例，就重用该实例，否则就创建新的实例并放入栈顶。

 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

## 源码分析

### 主要涉及4个类

**1) ActivityRecord：activity记录的基本数据结构，**存在历史栈的一个实例，代表一个Activity。

**2) TaskRecord：**Activity栈，内部维护一个ArrayList<ActivityRecord>

**3) ActivityStack：**并不是一个Activity栈，真正意义上的Activity栈是TaskRecord，这个类是负责管理各个Activity栈，内部维护一个ArrayList<TaskRecord>

**4) ActivityStackSupervisor：**内部持有一个ActivityStack，而ActivityStack内部也持有ActivityStackSupervisor，相当于ActivityStack的辅助管理类

#### **ActivityRecord**

/frameworks/base/services/core/java/com/android/server/am/ActivityRecord.java

**void** **[dump](http://androidxref.com/5.1.0_r1/s?refs=dump&project=frameworks)**([PrintWriter](http://androidxref.com/5.1.0_r1/s?defs=PrintWriter&project=frameworks) **[pw](http://androidxref.com/5.1.0_r1/s?refs=pw&project=frameworks)**, [String](http://androidxref.com/5.1.0_r1/s?defs=String&project=frameworks) **[prefix](http://androidxref.com/5.1.0_r1/s?refs=prefix&project=frameworks)**) 定义的方法用于dumpsys activity对于2.2.1.1. ActivityRecord的打印，

final class ActivityRecord {

TaskRecord **task**; // the task this is in.

final IApplicationToken.Stub appToken;

final int userId;

int theme;

int launchMode;

...

}

成员变量task表示自己所在的TaskRecord，这样要找到自己所在的TaskRecord就不必遍历查找了。

#### TaskRecord

final class TaskRecord {

/\*\* List of all activities in the task arranged in history order \*/

final ArrayList<ActivityRecord> mActivities;

/\*\* Current stack \*/

ActivityStack stack;

}

同样的道理，成员变量stack表示自己所在的ActivityStack

#### ActivityStack

final class ActivityStack {

private ArrayList<TaskRecord> **mTaskHistory** = new ArrayList<>();

/\*\* Run all ActivityStacks through this \*/

final ActivityStackSupervisor mStackSupervisor;

ActivityStack(ActivityStackSupervisor.ActivityContainer activityContainer, RecentTasks recentTasks) {

mStackSupervisor = activityContainer.getOuter();

...

}

}

#### ActivityStackSupervisor

public final class ActivityStackSupervisor {

private ActivityStack mFocusedStack;

}

### 场景分析

#### 二、场景解析

##### 1、从桌面第一次启动App

startActivityLocked里构造一个ActivityRecord  
新建一个TaskRecord，并存入mTaskHistory  
ActivityRecord存入mActivities



final int startActivityUncheckedLocked(...) {

final int startActivityUncheckedLocked(...) {

if (reuseTask == null) {

r.setTask(targetStack.createTaskRecord(...);

...

targetStack.startActivityLocked(r, newTask, doResume, keepCurTransition, options);

...

}

}

1) TaskRecord存入mTaskHistory

TaskRecord createTaskRecord(int taskId, ActivityInfo info, Intent intent,

IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,

boolean toTop) {

TaskRecord task = new TaskRecord(mService, taskId, info, intent, voiceSession,

voiceInteractor);

addTask(task, toTop, false);

return task;

}

void addTask(final TaskRecord task, final boolean toTop, boolean moving) {

task.stack = this;

if (toTop) {

insertTaskAtTop(task, null);

} else {

mTaskHistory.add(0, task);

updateTaskMovement(task, false);

}

...

}

private void insertTaskAtTop(TaskRecord task, ActivityRecord newActivity) {

...

mTaskHistory.add(taskNdx, task);

updateTaskMovement(task, true);

}

2) ActivityRecord存入mActivities



final void startActivityLocked(ActivityRecord r, boolean newTask, ...) {

...

task = mTaskHistory.get(taskNdx);

...

task.addActivityToTop(r);

}

void addActivityToTop(ActivityRecord r) {

addActivityAtIndex(mActivities.size(), r);

}

void addActivityAtIndex(int index, ActivityRecord r) {

...

mActivities.add(index, r);

...

}

##### 2、App启动一个Activity

会不会新建一个TaskRecord取决于launchMode，默认的standard模式不会创建新的TaskRecord  
构造一个ActivityRecord存入mActivities，与上面第二步一样

##### 3、回退



/\*\* @return true if this was the last activity in the task \*/

boolean removeActivity(ActivityRecord r) {

mActivities.remove(r);

...

if (mActivities.isEmpty()) {

return !mReuseTask;

}

...

return false;

}

#### 4、再次回退，回到桌面

从mActivities移除当前ActivityRecord与上面一样，只是当mActivities为空时，会触发mTaskHistory移除当前TaskRecord，如果mTaskHistory为空，则切换到桌面，给mStackSupervisor.mFocusedStack重新赋值

private void removeActivityFromHistoryLocked(ActivityRecord r, String reason) {

...

final TaskRecord task = r.task;

if (task != null && task.removeActivity(r)) {

if (mStackSupervisor.isFrontStack(this) && task == topTask() &&

task.isOverHomeStack()) {

mStackSupervisor.moveHomeStackTaskToTop(task.getTaskToReturnTo(), reason);

}

removeTask(task, reason);

}

}

void removeTask(TaskRecord task, String reason, boolean notMoving) {

...

mTaskHistory.remove(task);

...

if (mTaskHistory.isEmpty()) {

final boolean notHomeStack = !isHomeStack();

if (isOnHomeDisplay()) {

String myReason = reason + " leftTaskHistoryEmpty";

if (mFullscreen || !adjustFocusToNextVisibleStackLocked(null, myReason)) {

mStackSupervisor.moveHomeStack(notHomeStack, myReason);

}

}

...

}

...

task.stack = null;

}

如果不是从Activity调用startActivity，那么目标Activity就不知道自己该属于哪个TaskRecord，所以得指定FLAG\_ACTIVITY\_NEW\_TASK，就会新建一个TaskRecord

作者：风风风筝  
链接：http://www.jianshu.com/p/dbea4645cbc7  
來源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## REF

[Activity（三）栈管理](http://www.jianshu.com/p/dbea4645cbc7)

# ActivityRecord

BroadcastRecord，ServiceRecord都继承于Binder对象，而ActivityRecord并没有继承于Binder。 但ActivityRecord的成员变量appToken的数据类型为Token，Token继承于IApplicationToken.Stub。

appToken：system\_server进程通过调用scheduleLaunchActivity()将appToken传递到App进程，

* 调用createActivityContext()，保存到ContextImpl.mActivityToken
* 调用activity.attach()，保存到Activity.mToken；

ServiceRecord本身继承于Binder对象，传递到客户端的代理：

* 调用Service.attach()，保存到Service.mToken；
* 用途：stopSelf,startForeground, stopForeground

## 二. ActivityRecord结构体

先以一幅图来展示AMS管理Activity所涉及的相关数据结构： [点击查看大图](http://www.gityuan.com/images/ams/activity/activity_record.jpg)



* ActivityRecord: 记录着Activity信息
* TaskRecord: 记录着task信息
* ActivityStack: 栈信息

### 2.1 ActivityRecord

Activity的信息记录在ActivityRecord对象, 并通过通过成员变量task指向TaskRecord

* ProcessRecord app //跑在哪个进程
* TaskRecord task //跑在哪个task
* ActivityInfo info // Activity信息
* int mActivityType //Activity类型
* ActivityState state //Activity状态
* ApplicationInfo appInfo //跑在哪个app
* ComponentName realActivity //组件名
* String packageName //包名
* String processName //进程名
* int launchMode //启动模式
* int userId // 该Activity运行在哪个用户id

再来说一说Activity类型和Activity状态的常量：

mActivityType：

* APPLICATION\_ACTIVITY\_TYPE：普通应用类型
* HOME\_ACTIVITY\_TYPE：桌面类型
* RECENTS\_ACTIVITY\_TYPE：最近任务类型

ActivityState：

* INITIALIZING
* RESUMED：已恢复
* PAUSING
* PAUSED：已暂停
* STOPPING
* STOPPED：已停止
* FINISHING
* DESTROYING
* DESTROYED：已销毁

最后，说一说时间相关的成员变量：

| **时间点** | **赋值时间** | **含义** |
| --- | --- | --- |
| createTime | new ActivityRecord | Activity首次创建时间点 |
| displayStartTime | AS.setLaunchTime | Activity首次启动时间点 |
| fullyDrawnStartTime | AS.setLaunchTime | Activity首次启动时间点 |
| startTime |  | Activity上次启动的时间点 |
| lastVisibleTime | AR.windowsVisibleLocked | Activity上次成为可见的时间点 |
| cpuTimeAtResume | AS.completeResumeLocked | 从Rsume以来的cpu使用时长 |
| pauseTime | AS.startPausingLocked | Activity上次暂停的时间点 |
| launchTickTime | AR.startLaunchTickingLocked | Eng版本才赋值 |
| lastLaunchTime | ASS.realStartActivityLocked | 上一次启动时间 |

其中AR是指ActivityRecord, AS是指ActivityStack。

### 2.2 TaskRecord

Task的信息记录在TaskRecord对象.

* ActivityStack stack; //当前所属的stack
* ArrayList mActivities; // 当前task的所有Activity列表
* int taskId
* String affinity； 是指root activity的affinity，即该Task中第一个Activity;
* int mCallingUid;
* String mCallingPackage； //调用者的包名

### 2.3 ActivityStack

* ArrayList mTaskHistory //保存所有的Task列表
* ArrayList mStacks; //所有stack列表
* final int mStackId;
* int mDisplayId;
* ActivityRecord mPausingActivity //正在pause
* ActivityRecord mLastPausedActivity
* ActivityRecord mResumedActivity //已经resumed
* ActivityRecord mLastStartedActivity

所有前台stack的mResumedActivity的state == RESUMED, 则表示allResumedActivitiesComplete, 此时mLastFocusedStack = mFocusedStack;

### 2.4 ActivityStackSupervisor

* ActivityStack mHomeStack //桌面的stack
* ActivityStack mFocusedStack //当前聚焦stack
* ActivityStack mLastFocusedStack //正在切换
* SparseArray mActivityDisplays //displayId为key
* SparseArray mActivityContainers // mStackId为key

home的栈ID等于0,即HOME\_STACK\_ID = 0;

## 三. Activity栈关系

### 3.1 Stack组成图

Activity栈结构体的组成关系，[点击查看大图](http://www.gityuan.com/images/ams/activity/ams_relations.jpg)



* 一般地，对于没有分屏功能以及虚拟屏的情况下，ActivityStackSupervisor与ActivityDisplay都是系统唯一；
* ActivityDisplay主要有Home Stack和App Stack这两个栈；
* 每个ActivityStack中可以有若干个TaskRecord对象；
* 每个TaskRecord包含如果个ActivityRecord对象；
* 每个ActivityRecord记录一个Activity信息。

(1)正向关系链表：

ActivityStackSupervisor.mActivityDisplays

-> ActivityDisplay.mStacks

-> ActivityStack.mTaskHistory

-> TaskRecord.mActivities

-> ActivityRecord

(2)反向关系链表：

ActivityRecord.task

-> TaskRecord.stack

-> ActivityStack.mStackSupervisor

-> ActivityStackSupervisor

注：ActivityStack.mDisplayId可找到所对应的ActivityDisplay；

# 情景分析

## ActivityManagerService是如何启动Launcher

ActivityManagerService .startHomeActivityLocked()。

而在**startHomeActivityLocked()**中我们首先是读取查询intent中Category类型为HOME的Activity，这个信息是保存在PackageManagerService，我们可以通过方法intent.resolveActivityInfo(mContext.getPackageManager(),STOCK\_PM\_FLAGS)

而intent中ategory类型为HOME，在新版本的android中已经不在用这个方法了，而是用

调用ActivityManagerService中的resolveActivityInfo(),实际上是在resolveActivityInfo()调用了resolveIntent()调用了chooseBestActivity()，由于intent中Category类型为HOME的Activity可能有多个，若优先级(android:priority)相同系统会弹出提示框让用户选择，优先级(android:priority)不同，则会选择启动优先级高的Activity，



# Destroy

**final boolean** destroyActivityLocked(ActivityRecord r, **boolean** removeFromApp, String reason) {  
 **if** (DEBUG\_SWITCH || DEBUG\_CLEANUP) Slog.v(TAG\_SWITCH,  
 **"Removing activity from "** + reason + **": token="** + r  
 + **", app="** + (r.app != **null** ? r.app.processName : **"(null)"**));  
 EventLog.writeEvent(EventLogTags.AM\_DESTROY\_ACTIVITY,  
 r.userId, System.identityHashCode(r),  
 r.getTask().taskId, r.shortComponentName, reason);  
  
 **boolean** removedFromHistory = **false**;  
  
 cleanUpActivityLocked(r, **false**, **false**);  
  
 **final boolean** hadApp = r.app != **null**;  
  
 **if** (hadApp) {  
 **if** (removeFromApp) {  
 r.app.activities.remove(r);  
 **if** (mService.mHeavyWeightProcess == r.app && r.app.activities.size() <= 0) {  
 mService.mHeavyWeightProcess = **null**;  
 mService.mHandler.sendEmptyMessage(  
 ActivityManagerService.CANCEL\_HEAVY\_NOTIFICATION\_MSG);  
 }  
 **if** (r.app.activities.isEmpty()) {  
 *// Update any services we are bound to that might care about whether  
 // their client may have activities.* mService.mServices.updateServiceConnectionActivitiesLocked(r.app);  
 *// No longer have activities, so update LRU list and oom adj.* mService.updateLruProcessLocked(r.app, **false**, **null**);  
 mService.updateOomAdjLocked();  
 }  
 }  
  
 **boolean** skipDestroy = **false**;  
  
 **try** {  
 **if** (DEBUG\_SWITCH) Slog.i(TAG\_SWITCH, **"Destroying: "** + r);  
 r.app.thread.scheduleDestroyActivity(r.appToken, r.finishing,  
 r.configChangeFlags);  
 } **catch** (Exception e) {  
 *// We can just ignore exceptions here... if the process  
 // has crashed, our death notification will clean things  
 // up.  
 //Slog.w(TAG, "Exception thrown during finish", e);* **if** (r.finishing) {  
 removeActivityFromHistoryLocked(r, reason + **" exceptionInScheduleDestroy"**);  
 removedFromHistory = **true**;  
 skipDestroy = **true**;  
 }  
 }  
  
 r.nowVisible = **false**;  
  
 *// If the activity is finishing, we need to wait on removing it  
 // from the list to give it a chance to do its cleanup. During  
 // that time it may make calls back with its token so we need to  
 // be able to find it on the list and so we don't want to remove  
 // it from the list yet. Otherwise, we can just immediately put  
 // it in the destroyed state since we are not removing it from the  
 // list.* **if** (r.finishing && !skipDestroy) {  
 **if** (DEBUG\_STATES) Slog.v(TAG\_STATES, **"Moving to DESTROYING: "** + r  
 + **" (destroy requested)"**);  
 r.state = ActivityState.DESTROYING;  
 Message msg = mHandler.obtainMessage(DESTROY\_TIMEOUT\_MSG, r);  
 mHandler.sendMessageDelayed(msg, DESTROY\_TIMEOUT);  
 } **else** {  
 **if** (DEBUG\_STATES) Slog.v(TAG\_STATES,  
 **"Moving to DESTROYED: "** + r + **" (destroy skipped)"**);  
 r.state = ActivityState.DESTROYED;  
 **if** (DEBUG\_APP) Slog.v(TAG\_APP, **"Clearing app during destroy for activity "** + r);  
 r.app = **null**;  
 }  
 } **else** {  
 *// remove this record from the history.* **if** (r.finishing) {  
 removeActivityFromHistoryLocked(r, reason + **" hadNoApp"**);  
 removedFromHistory = **true**;  
 } **else** {  
 **if** (DEBUG\_STATES) Slog.v(TAG\_STATES, **"Moving to DESTROYED: "** + r + **" (no app)"**);  
 r.state = ActivityState.DESTROYED;  
 **if** (DEBUG\_APP) Slog.v(TAG\_APP, **"Clearing app during destroy for activity "** + r);  
 r.app = **null**;  
 }  
 }  
  
 r.configChangeFlags = 0;  
  
 **if** (!mLRUActivities.remove(r) && hadApp) {  
 Slog.w(TAG, **"Activity "** + r + **" being finished, but not in LRU list"**);  
 }  
  
 **return** removedFromHistory;  
}

# context的初始化过程

ActivityThread管理着一个应用进程的主线程，用来调度和执行运行在该进程中的Activities，Broadcasrs以及其他的相关操作。在android中，每个应用都运行在一个独立的进程中，在这个进程中至少含有一个主线程，这个主线程由ActivityThread来管理,ActivityThread本身运行在主线程中

# Dump

Fasd

**else if** (**"broadcasts"**.equals(cmd) || **"b"**.equals(cmd))

## dumpBroadcastsLocked

主要输出的对象：

* ReceiverList, BroadcastFilter,
* IntentResolver,
* BroadcastQueue, BroadcastRecord
* Handler, Looper

Receiver Resolver Table:

Historical broadcasts summary [foreground]:

#2: act=android.intent.action.SCREEN\_OFF flg=0x50200010

0 dispatch +486ms finish

Historical Broadcast foreground #1:

BroadcastRecord{d14d13c u-1 android.intent.action.SCREEN\_OFF} to user -1

Intent { act=android.intent.action.SCREEN\_OFF flg=0x50200010 }

caller=android 834:system/1000 pid=834 uid=1000

enqueueClockTime=1970-01-07 12:47:20 dispatchClockTime=1970-01-07 12:47:20

dispatchTime=-10s785ms (0 since enq) finishTime=-10s456ms (+329ms since disp)

resultTo=null resultCode=0 resultData=null

resultAbort=false ordered=true sticky=false initialSticky=false

nextReceiver=16 receiver=null

Deliver #0: BroadcastFilter{1b91e7f u0 ReceiverList{546439e 834 system/1000/u0 local:2705dd9}}

Deliver #1: BroadcastFilter{45f0587 u0 ReceiverList{96c1c6 834 system/1000/u0 local:7e8dba1}}

Deliver #2: BroadcastFilter{27265ff u0 ReceiverList{e81f91e 834 system/1000/u0 local:1b7959}}

Deliver #3: BroadcastFilter{673ce83 u0 ReceiverList{d656132 834 system/1000/u0 local:ed1e53d}}

Deliver #4: BroadcastFilter{2f98a7b u0 ReceiverList{7ed320a 834 system/1000/u0 local:f359275}}

Deliver #5: BroadcastFilter{5a16f55 u-1 ReceiverList{c89a0c 834 system/1000/u-1 local:bb1713f}}

Deliver #6: BroadcastFilter{6a8f8ca u0 ReceiverList{f7d3c35 834 system/1000/u0 local:7720e6c}}

Deliver #7: BroadcastFilter{3abcfb5 u0 ReceiverList{6b8bfec 834 system/1000/u0 local:515029f}}

Deliver #8: BroadcastFilter{a9cb36a u0 ReceiverList{896bd55 834 system/1000/u0 local:82a600c}}

Deliver #9: BroadcastFilter{c19c0ea u0 ReceiverList{bd190d5 985 com.android.systemui/10025/u0 remote:606518c}}

Deliver #10: BroadcastFilter{98651a5 u0 ReceiverList{1a5539c 1130 com.android.phone/1001/u0 remote:a2dd70f}}

Deliver #11: BroadcastFilter{5f17c47 u-1 ReceiverList{321db86 985 com.android.systemui/10025/u-1 remote:52d4461}}

Deliver #12: BroadcastFilter{e885abe u0 ReceiverList{adddd79 985 com.android.systemui/10025/u0 remote:abf1a40}}

Deliver #13: BroadcastFilter{112a6ef u-1 ReceiverList{cc7e7ce 2265 com.android.nfc/1027/u-1 remote:fe34c9}}

Deliver #14: BroadcastFilter{9e6ce3c u10 ReceiverList{534182f 26814 com.android.launcher3/1010013/u10 remote:82f560e}}

Deliver #15: BroadcastFilter{742f36e u-1 ReceiverList{dd5aae9 985 com.android.systemui/10025/u-1 remote:608f770}}

# ANR

## Service Timeout原理

不介绍key，便于理解。

Service Timeout是位于”ActivityManager”线程中的AMS.MainHandler收到SERVICE\_TIMEOUT\_MSG消息时触发。

对于Service有两类:对于前台服务，则超时为SERVICE\_TIMEOUT = 20s；

对于后台服务，则超时为SERVICE\_BACKGROUND\_TIMEOUT = 200s

[Service前台服务的使用](http://www.jianshu.com/p/5505390503fa)

在一般情况下，Service几乎都是在后台运行，一直默默地做着辛苦的工作。但这种情况下，**后台运行的Service系统优先级相对较低**，当系统内存不足时，在后台运行的Service就有可能被回收。  
　　那么，如果我们希望Service可以**一直保持运行状态且不会在内存不足的情况下被回收**时，可以选择将需要保持运行的Service设置为**前台服务**。App中的**音乐播放服务**

由变量ProcessRecord.execServicesFg来决定是否前台启动

[理解Android ANR的触发原理](http://gityuan.com/2016/07/02/android-anr/)

### 埋炸弹realStartServiceLocked

文章[startService流程分析](http://gityuan.com/2016/03/06/start-service/)详细介绍Service启动流程. 其中在Service进程attach到system\_server进程的过程中会调用realStartServiceLocked()方法来埋下炸弹. 通过发送delay消息(SERVICE\_TIMEOUT\_MSG). 炸弹已埋下, 我们并不希望炸弹被引爆, 那么就需要在炸弹爆炸之前拆除炸弹.

|  |
| --- |
| **//1.埋炸弹**  ActiveServices.realStartServiceLocked(ServiceRecord r,  ProcessRecord app, boolean execInFg) throws RemoteException {  //最终执行服务的onCreate()方法  bumpServiceExecutingLocked(r, execInFg, "create");  try {  app.thread.scheduleCreateService(r, r.serviceInfo,  mAm.compatibilityInfoForPackageLocked(r.serviceInfo.applicationInfo),  app.repProcState);  } catch (DeadObjectException e) {  mAm.appDiedLocked(app);  throw e;  }  } |
| ActiveServices.bumpServiceExecutingLocked(ServiceRecord r, boolean fg, String why) {  scheduleServiceTimeoutLocked(r.app);  } |
| ActiveServices.scheduleServiceTimeoutLocked(ProcessRecord proc) {  long now = SystemClock.uptimeMillis();  Message msg = mAm.mHandler.obtainMessage(  ActivityManagerService.SERVICE\_TIMEOUT\_MSG);  msg.obj = proc;  **mAm.mHandler.sendMessageAtTime(msg,**  **proc.execServicesFg ? (now+SERVICE\_TIMEOUT) : (now+ SERVICE\_BACKGROUND\_TIMEOUT));**  } |

### 拆炸弹ActivityThread.handleCreateService

在system\_server进程AS.realStartServiceLocked()调用的过程会埋下一颗炸弹, 超时没有启动完成则会爆炸. 那么什么时候会拆除这颗炸弹的引线呢? 经过Binder等层层调用进入目标进程的主线程handleCreateService()的过程.

|  |
| --- |
| **//2.拆炸弹**  ActivityThread.handleCreateService(CreateServiceData data) {  java.lang.ClassLoader cl = packageInfo.getClassLoader();  Service service = (Service) cl.loadClass(data.info.name).newInstance();  try {  //创建ContextImpl对象  ContextImpl context = ContextImpl.createAppContext(this, packageInfo);  context.setOuterContext(service);  //创建Application对象  Application app = packageInfo.makeApplication(false, mInstrumentation);  service.attach(context, this, data.info.name, data.token, app,  ActivityManagerNative.getDefault());  //调用服务onCreate()方法  service.onCreate();  ActivityManagerNative.getDefault().serviceDoneExecuting(  data.token, SERVICE\_DONE\_EXECUTING\_ANON, 0, 0);  }  }  AS.serviceDoneExecutingLocked(ServiceRecord r, boolean inDestroying,  boolean finishing) {  if (r.executeNesting <= 0) {  if (r.app != null) {  r.app.execServicesFg = false;  r.app.executingServices.remove(r);  if (r.app.executingServices.size() == 0) {  //当前服务所在进程中没有正在执行的service,则移除服务超时消息SERVICE\_TIMEOUT\_MSG **mAm.mHandler.removeMessages(ActivityManagerService.SERVICE\_TIMEOUT\_MSG, r.app);**  }  } |

### 引爆炸弹

前面介绍了埋炸弹和拆炸弹的过程, 如果在炸弹倒计时结束之前成功拆卸炸弹,那么就没有爆炸的机会, 但是世事难料. 总有些极端情况下无法即时拆除炸弹,导致炸弹爆炸, 其结果就是App发生ANR. 接下来,带大家来看看炸弹爆炸的现场:

在system\_server进程中有一个Handler线程, 名叫”ActivityManager”.当倒计时结束便会向该Handler线程发送 一条信息SERVICE\_TIMEOUT\_MSG,

|  |
| --- |
| **//3.引爆炸弹**  ActivityManagerService.MainHandler **extends** Handler{  **public void** handleMessage(Message msg){  **switch**(msg.what){  **case** SERVICE\_TIMEOUT\_MSG:{  mServices.serviceTimeout((ProcessRecord)msg.obj);  }**break**;  }  }  } AS.serviceTimeout(ProcessRecord proc){  String anrMessage=**null**;  **synchronized**(mAm){  …  **if**(timeout!=**null**&&mAm.mLruProcesses.contains(proc)){  Slog.w(TAG,**"Timeout executing service: "**+timeout);  … mAm.mHandler.postDelayed(mLastAnrDumpClearer,LAST\_ANR\_LIFETIME\_DURATION\_MSECS);  anrMessage=**"executing service "**+timeout.shortName;  }  }  **if**(anrMessage!=**null**){   ***//当存在timeout的service，则执行appNotResponding***mAm.appNotResponding(proc,**null**,**null**,**false**,anrMessage);  } } |

#### ActivityManagerService.appNotResponding()

getprop dalvik.vm.stack-trace-file

缩减版

|  |
| --- |
| final void appNotResponding(ProcessRecord app, ActivityRecord activity,  ActivityRecord parent, boolean aboveSystem, final String annotation) {  //firstPids和lastPids两个集合存放那些将会在traces中输出信息的进程的进程号  synchronized (this) {  //一些特定条件下会忽略ANR  // PowerManager.reboot() can block for a long time, so ignore ANRs while shutting down.  if (mShuttingDown) {  Slog.i(TAG, "During shutdown skipping ANR: " + app + " " + annotation);  return;  } else if (app.notResponding) {  Slog.i(TAG, "Skipping duplicate ANR: " + app + " " + annotation);  return;  } else if (app.crashing) {  Slog.i(TAG, "Crashing app skipping ANR: " + app + " " + annotation);  return;  }  //当前发生ANR的应用进程被第一个添加进firstPids集合中.  //**所以会第一个向traces文件中写入信息**。  //反过来说，traces文件中出现的第一个进程正常情况下就是发生ANR的那个进程。  // Dump thread traces as quickly as we can, starting with "interesting" processes.  firstPids.add(app.pid);  ..  // Log the ANR to the main log.  StringBuilder info = new StringBuilder();  info.setLength(0);  info.append("ANR in ").append(app.processName);  if (activity != null && activity.shortComponentName != null) {  info.append(" (").append(activity.shortComponentName).append(")");  }  info.append("\n");  info.append("PID: ").append(app.pid).append("\n");  if (annotation != null) {  info.append("Reason: ").append(annotation).append("\n");  }  //dumpStackTraces()是输出ANR信息到traces文件的函数  File tracesFile = dumpStackTraces(true, firstPids, processCpuTracker, lastPids,  NATIVE\_STACKS\_OF\_INTEREST);  //输出ANR发生前一段时间内的CPU使用率  info.append(processCpuTracker.printCurrentLoad());  info.append(cpuInfo);  }  //输出ANR发生后一段时间内的CPU使用率  info.append(processCpuTracker.printCurrentState(anrTime));  //此处， 把ANR的信息输出到main log中.  Slog.e(TAG, info.toString());  // 显示ANR提示对话框  // Bring up the infamous App Not Responding dialog  Message msg = Message.obtain();  HashMap<String, Object> map = new HashMap<String, Object>();  msg.what = SHOW\_NOT\_RESPONDING\_MSG;  msg.obj = map;  map.put("app", app);  if (activity != null) {  map.put("activity", activity);  }  mUiHandler.sendMessage(msg);  }  }  public static File dumpStackTraces(boolean clearTraces, ArrayList<Integer> firstPids,  ProcessCpuTracker processCpuTracker, SparseArray<Boolean> lastPids, String[] nativeProcs) {  //系统属性“dalvik.vm.stack-trace-file”用来配置trace信息输出文件  String tracesPath = SystemProperties.get("**dalvik.vm.stack-trace-file**", null);  //clearTraces为true，会删除旧文件，创建新文件  if (clearTraces && tracesFile.exists()) tracesFile.delete();  tracesFile.createNewFile();  FileUtils.setPermissions(tracesFile.getPath(), 0666, -1, -1); // -rw-rw-rw-  } catch (IOException e) {  Slog.w(TAG, "Unable to prepare ANR traces file: " + tracesPath, e);  return null;  }  dumpStackTraces(tracesPath, firstPids, processCpuTracker, lastPids, nativeProcs);  return tracesFile;  } |

boolean showBackground = Settings.Secure.getInt(mContext.getContentResolver(),

Settings.Secure.ANR\_SHOW\_BACKGROUND, 0) != 0;

if (mService.canShowErrorDialogs() || showBackground) {

d = new AppNotRespondingDialog(mService,

mContext, proc, (ActivityRecord)data.get("activity"),

msg.arg1 != 0);

proc.anrDialog = d;

# onConfigurationChange

旋屏事件上报流程：

1、传感器（默认为加速度传感器）计算数据，决定是否上报旋屏事件。

2、上报是通过回调函数实现的，在PhoneWindowManger中实现指定接口。

3、PhoneWindowManger与WMS进行交互，通知其更新rotation。

4、WMS更新rotation后，发现的确发生改变了，去通知AMS处理。

5、AMS获取WMS中rotation数据，然后更新处理。通常流程是通过ApplicationThread与ActivityThread交互。最后调用流程是 onConfigurationChanged --> Activity重新创建。

## 屏幕方向概念

需要注意的是两者(ActivityInfo和Configuration)有关方向的常量定义是不一致的，ActivityInfo中有关常量是用于决策显示方向的策略，而Configuration对象中的常量则是明确的 实际 显示方向，共有4中可能：未定义(UNDEFINED) 、横屏 (LANDSCAPE) 、竖屏 (PORTRAIT) 、以及正方形 (SQUARE) 。

### screenOrientations（ActivityInfo）

是对Activity而言的

->

#### screenOrientations属性值介绍

screenOrientations属性共有7中可选值(常量定义在 android.content.pm.ActivityInfo类中 ) ：

landscape：横屏(风景照) ，显示时宽度大于高度；

portrait：竖屏 (肖像照) ， 显示时 高 度大于 宽 度 ；

user：用户当前的首选方向；

behind：继承Activity堆栈中当前Activity下面的那个Activity的方向；

sensor：由物理感应器决定显示方向，它取决于用户如何持有设备，当 设备 被旋转时方向会随之变化——在横屏与竖屏之间；

nosensor：忽略物理感应器——即显示方向与物理感应器无关，不管用户如何旋转设备显示方向都不会随着改变("unspecified"设置除外)；

unspecified ：未指定，此为默认值，由Android系统自己选择适当的方向，选择策略视具体设备的配置情况而定，因此不同的设备会有不同的方向选择；

#### get

Activity.getRequestedOrientation()

#### setRequestedOrientationt

因为是针对Activity而言的 ，可以在AndroidManifest.xml，通过<activity> 标记的screenOrientation 属性静态设定，

<activity

android:name=".SketchpadActivity"

android:screenOrientation="landscape"/><!--让该Activity总是显示为横屏-

通过setRequestedOrientation(xxx)方法设置与在AndroidManifest.xml文件中配置是等效的，因此通过以上例程明确指定方向后，Activity将不再自动根据物理传感器进行横竖屏切换，若要恢复，再调用setRequestedOrientation(UNSPECIFIED)即可。

### Configuration

而Configuration对象中的常量则是明确的 实际 显示方向，共有4中可能：未定义(UNDEFINED) 、横屏 (LANDSCAPE) 、竖屏 (PORTRAIT) 、以及正方形 (SQUARE) 。

首先，要了解Android默认在情况下，在系统的配置改变的情况下会重新创建activity，这个很好理解，因为系统环境变了，当然要重新创建activity来适应这个改变喽。但是在很多时候我们不想在系统配置改变时重新加载activity，那么就需要

android:configChanges="orientation|screenSize|smallestScreenSize|keyboard|keyboardHidden|navigation"

onConfigurationChanged

但是在很多时候我们不想在系统配置改变时重新加载activity，那么就需要**android:configChanges=“xxx|xxx”**这个样子来搞定了。这里的xxx表这不同的系统配置的改变。

这里的xxx包含了很多种，就像我上边写的，可以用|来连接多个值。下面简单介绍下各个值的不同含义：

**mmc**：这里的意思是SIM卡的标识IMSI中的mmc发生了改变，这个mmc的用途是区分不同的国家的3位数号码，比如中国的是460，玩过水货的估计有了解这个东西的。

**mnc：**IMSI中的运营商代码改变，就是用来区分当前SIM卡的网络运营商的，中国移动是00，联通是01，电信是03。

**locale：**本地位置改变触犯，恩，这个貌似跟语言切换有关，没用过，不是跟了解。

**keyboard、keyboardHidden：**这两个一起说一下，就是跟键盘有关的操作，前一个是键盘类型变化，如插了个键盘在设备上，后一个指的是键盘的可访问性发生变化。

**teachScreen：**触摸屏变化。

**navigation：**系统导航方式变化。

**screenLayout：**屏幕布局发生了改变。

**layloutDirection**：布局的方向发生变化，注意，是布局的方向，不是屏幕的，比较难理解的一个概念，是API17之后出现的

**orientation：**屏幕方向改变，具体就是屏幕的横竖屏切换。

**uiMode**：用户界面改变，这个是api18后出现的，比如切换夜间模式等场景会出现这个。

**fontScale**：系统字体大小改变。

**screenSize**：设备的屏幕的尺寸信息发生了改变，旋转屏幕时也会触发这个，但是这个值是API13之后才有的，因此13之前的设备没有这个。

**smallestScreenSize**：设备的物理屏幕发生改变，同样是API13之后的东西，指的是设备的物理屏幕发生变化，比如外接显示器之类的。。。大体上是这个意思。

其实讲道理在平时的开发中能用到的也就是**android:configChanges="orientation"**罢了，其他的很少会用到。。。

解决Android横竖屏切换调用相关的生命周期

android:configChanges="keyboardHidden|orientation"

在配置文件Activity中加入这么一段话不管怎么横竖屏切换都不会调用任何生命周期方法

#### getConfiguration获得方向

还可以通过Configuration对象来取得Activity当前的显示方向：

// 通过Configuration对象 确认当前显示方向

Configuration conf = getResources().getConfiguration();

String orientName = "undefined";

switch (orient) {

case Configuration.ORIENTATION\_LANDSCAPE:

orientName = "landscape";

case Configuration.ORIENTATION\_PORTRAIT:

orientName = "portrait";

case Configuration.ORIENTATION\_SQUARE:

orientName = "square";

default:

orientName = "undefined";

}

Logger.get().i("conf.orient:" + orientName);

#### getScreenRotation

private int getScreenRotation() {

WindowManager wm = (WindowManager) getSystemService(Context.WINDOW\_SERVICE);

Display display = wm.getDefaultDisplay();

try {

Method m = display.getClass().getDeclaredMethod("getRotation");

return (Integer) m.invoke(display);

} catch (Exception e) {

return Surface.ROTATION\_0;

}

}

private int getScreenOrientation() {

switch (getScreenRotation()) {

case Surface.ROTATION\_0:

return ActivityInfo.SCREEN\_ORIENTATION\_PORTRAIT;

case Surface.ROTATION\_90:

return ActivityInfo.SCREEN\_ORIENTATION\_LANDSCAPE;

case Surface.ROTATION\_180:

return (Build.VERSION.SDK\_INT >= 8 ? ActivityInfo.SCREEN\_ORIENTATION\_REVERSE\_PORTRAIT

: ActivityInfo.SCREEN\_ORIENTATION\_PORTRAIT);

case Surface.ROTATION\_270:

return (Build.VERSION.SDK\_INT >= 8 ? ActivityInfo.SCREEN\_ORIENTATION\_REVERSE\_LANDSCAPE

: ActivityInfo.SCREEN\_ORIENTATION\_LANDSCAPE);

default:

return 0;

}

}

## 调用的流程框架图



ConfigurationChanged的流程。屏幕旋转、mcc、系统语言切换等均是通过该流程来实现的。

以下以屏幕旋转为例。关于屏幕旋转对于Activity的生命周期的影响也在以下可以体现，如何处理屏幕旋转对Act的影响也有启示作用。

## 屏幕旋转事件上传

Input的子系统不细说

G-Sensor将旋转事件由底层上传到FW处理，改变Configuration中orientation的值并将事件继续上传：Configuration中对屏幕方向的定义：

/\*\*

\* Overall orientation of the screen. May be one of

\* {@link #ORIENTATION\_LANDSCAPE}, {@link #ORIENTATION\_PORTRAIT}.

\*/

public int orientation;

取之范围0-3，

## AMNative.updatePersistentConfiguration

调用ActivityManagerNative.getDefault().updatePersistentConfiguration(newConfig)将事件上传即调用AMS的同名方法updatePersistentConfiguration

public void updatePersistentConfiguration(Configuration values) throws RemoteException

{

Parcel data = Parcel.obtain();

Parcel reply = Parcel.obtain();

data.writeInterfaceToken(IActivityManager.descriptor);

values.writeToParcel(data, 0);

mRemote.transact(UPDATE\_PERSISTENT\_CONFIGURATION\_TRANSACTION, data, reply, 0);

reply.readException();

data.recycle();

reply.recycle();

｝

## AMS.updatePersistentConfigurationLocked

然后在AMS中，遍历每一个最近运行的程序，同步顺序执行以下方法

enforceCallingPermission(android.Manifest.permission.CHANGE\_CONFIGURATION,

"updateConfiguration()");

enforceWriteSettingsPermission("updateConfiguration()");

int userId = UserHandle.getCallingUserId();获得UID的方法

**private void** updatePersistentConfigurationLocked(Configuration values, @UserIdInt **int** userId) {  
 **final long** origId = Binder.clearCallingIdentity();  
 **try** {  
 updateConfigurationLocked(values, **null**, **false**, **true**, userId, **false** */\* deferResume \*/*);  
 } **finally** {  
 Binder.restoreCallingIdentity(origId);  
 }  
}

## AMS. updateConfigurationLocked

然后在AMS中，遍历每一个最近运行的程序，同步顺序执行以下方法

boolean updateConfigurationLocked(Configuration values, ActivityRecord starting, boolean persistent, boolean initLocale) {

...

mSystemThread.applyConfigurationToResources(configCopy);

for (int i=mLruProcesses.size()-1; i>=0; i--) {

ProcessRecord app = mLruProcesses.get(i);

try {

if (app.thread != null) {

if (DEBUG\_CONFIGURATION) Slog.v(TAG, "Sending to proc " + app.processName + " new config " + mConfiguration);

app.thread.scheduleConfigurationChanged(configCopy);

}

} catch (Exception e) {

}

}

...

广播通知作用？？

Intent intent = new Intent(Intent.ACTION\_CONFIGURATION\_CHANGED);

intent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY

| Intent.FLAG\_RECEIVER\_REPLACE\_PENDING

| Intent.FLAG\_RECEIVER\_FOREGROUND);

broadcastIntentLocked(null, null, intent, null, null, 0, null, null,

null, AppOpsManager.OP\_NONE, null, false, false,

MY\_PID, Process.SYSTEM\_UID, UserHandle.USER\_ALL);

其中scheduleConfigurationChanged()实际是通过Binder远程调用(该过程同步)ActivityThread中的同名方法//咳

## AppThreadNative. scheduleConfigurationChanged

final boolean reportToActivity = data.readInt() == 1;

scheduleActivityConfigurationChanged(b, overrideConfig, reportToActivity);

App

public final void scheduleActivityConfigurationChanged(IBinder token,

Configuration overrideConfig, boolean reportToActivity) throws RemoteException {

Parcel data = Parcel.obtain();

data.writeInterfaceToken(IApplicationThread.descriptor);

data.writeStrongBinder(token);

if (overrideConfig != null) {

data.writeInt(1);

overrideConfig.writeToParcel(data, 0);

} else {

data.writeInt(0);

}

data.writeInt(reportToActivity ? 1 : 0);

mRemote.transact(SCHEDULE\_ACTIVITY\_CONFIGURATION\_CHANGED\_TRANSACTION, data, null,

IBinder.FLAG\_ONEWAY);

data.recycle();

}

### ActivityThread .cheduleConfigurationChanged

1. public void scheduleConfigurationChanged(Configuration config) {
2. updatePendingConfiguration(config)*;*
3. sendMessage(H.CONFIGURATION\_CHANGED, config)*;*
4. }

### handleConfigurationChanged

将该进程下执行对应CONFIGURATION\_CHANGED处理：

1. final void handleConfigurationChanged(Configuration config, CompatibilityInfo compat) {

**if** (callbacks != **null**) {  
 **final int** N = callbacks.size();  
 **for** (**int** i=0; i<N; i++) {  
 ComponentCallbacks2 cb = callbacks.get(i);  
 **if** (cb **instanceof** Activity) {  
 *// If callback is an Activity - call corresponding method to consider override  
 // config and avoid onConfigurationChanged if it hasn't changed.* Activity a = (Activity) cb;  
 performConfigurationChangedForActivity(**mActivities**.get(a.getActivityToken()),  
 config);  
 } **else** {  
 performConfigurationChanged(cb, config);  
 }  
 }  
}

 从上面可以看出，在回调中，如果是Activity，则回调performConfigurationChangedForActivity()方法，如果是Services、Application等，回调performConfigurationChanged()方法，按注释解释，这样做是为了Activity在更新配置时重写配置和避免没有修改时回调onConfigurationChanged()方法，但不管是什么组件，最终都是调用performConfigurationChanged()方法

### Resources

#### ResourcesManager .applyConfigurationToResourcesLocked

Reload New Resources: 将Config应用到Resource的一系列操作。

1. public final boolean applyConfigurationToResourcesLocked(Configuration config,
2. CompatibilityInfo compat) {
3. ... *//对Configuration的比较以及更新到Resource*
5. Resources.updateSystemConfiguration(config, defaultDisplayMetrics, compat);
7. ApplicationPackageManager.configurationChanged();*//清空Icon和String缓存*
9. Configuration tmpConfig = null;
11. for (int i=mActiveResources.size()-1; i>=0; i--) {
12. ResourcesKey key = mActiveResources.keyAt(i);
13. Resources r = mActiveResources.valueAt(i).get();
14. if (r != null) {
15. ...*//更新Resource的Config*
17. } else {
18. *//Slog.i(TAG, "Removing old resources " + v.getKey());*
19. mActiveResources.removeAt(i);
20. }
21. }
22. return changes != 0;

#### updateConfiguration更新资源Res

.updateConfiguration(config, dm, compat);

在Resource中更新资源，在重新加载时就会使用新的资源

public final boolean applyConfigurationToResourcesLocked(Configuration config,

CompatibilityInfo compat) {

... //对Configuration的比较以及更新到Resource

Resources.updateSystemConfiguration(config, defaultDisplayMetrics, compat);

ApplicationPackageManager.configurationChanged();//清空Icon和String缓存

Configuration tmpConfig = null;

for (int i=mActiveResources.size()-1; i>=0; i--) {

ResourcesKey key = mActiveResources.keyAt(i);

Resources r = mActiveResources.valueAt(i).get();

if (r != null) {

...//更新Resource的Config

} else {

//Slog.i(TAG, "Removing old resources " + v.getKey());

mActiveResources.removeAt(i);

}

}

return changes != 0;

}

7.8.更新资源Res

在Resource中更新资源，在重新加载时就会使用新的资源

public void updateConfiguration(Configuration config, DisplayMetrics metrics, CompatibilityInfo compat) {

synchronized (mAccessLock) {

...

//更新Resource指向

//{@kth add 20151127 start

//像字体大小切换、语言切换等都会在此处开始更新资源的指向

//kth add 20151127 end@}

...

//清空drawable资源

clearDrawableCachesLocked(mDrawableCache, configChanges);

clearDrawableCachesLocked(mColorDrawableCache, configChanges);

mAnimatorCache.onConfigurationChange(configChanges);

mStateListAnimatorCache.onConfigurationChange(configChanges);

mColorStateListCache.clear();

flushLayoutCache();

}

...

}

### 回调performConfigurationChanged

当Configuration的操作执行完后，实现了ComponentCallbacks2接口的组件如Activity、Services、Application等将会执行回调onConfigurationChanged()方法(接口回调)，从而实现正在运行的app中所有组件对Config的更新响应。针对屏幕旋转更新前台显示，其他Configuration如字体、语言等需要通知所有。

该方法针对同一进程下Activity的状态进行甄别，将符合条件的Act放入list以方便后面操作.

private static void performConfigurationChanged(ComponentCallbacks2 cb, Configuration config) {

// Only for Activity objects, check that they actually call up to their

// superclass implementation.ComponentCallbacks2 is an interface, so

// we check the runtime type and act accordingly.

Activity activity = (cb instanceof Activity) ? (Activity) cb : null;

...

boolean shouldChangeConfig = false;

if ((activity == null) || (activity.mCurrentConfig == null)) {

shouldChangeConfig = true;

} else {

// If the new config is the same as the config this Activity

// is already running with then don't bother calling

// onConfigurationChanged

int diff = activity.mCurrentConfig.diff(config);

if (diff != 0) {

// If this activity doesn't handle any of the config changes

// then don't bother calling onConfigurationChanged as we're

// going to destroy it.

if ((~activity.mActivityInfo.getRealConfigChanged() & diff) == 0) {

shouldChangeConfig = true;

}

}

}

...

**if (shouldChangeConfig) {**  
 activity.mCalled = **false**;  
  **activity.onConfigurationChanged(configToReport);**  
 **if** (!activity.mCalled) {  
 **throw new** SuperNotCalledException(**"Activity "** + activity.getLocalClassName() +  
 **" did not call through to super.onConfigurationChanged()"**);  
 }  
}

Activity、Service、Application、Provider同样实现了ComponentCallbacks接口，从而实现四大组件全部更新状态和资源

ArrayList<ComponentCallbacks2> collectComponentCallbacks(boolean allActivities, Configuration newConfig) {

ArrayList<ComponentCallbacks2> callbacks= new ArrayList<ComponentCallbacks2>();

synchronized (mResourcesManager) {

final int NAPP = mAllApplications.size();// Application

for (int i=0; i<NAPP; i++) {

callbacks.add(mAllApplications.get(i));

}

final int NACT = mActivities.size();// Activity

for (int i=0; i<NACT; i++) {

ActivityClientRecord ar = mActivities.valueAt(i);

Activity a = ar.activity;

if (a != null) {

Configuration thisConfig = applyConfigCompatMainThread(

mCurDefaultDisplayDpi, newConfig,

ar.packageInfo.getCompatibilityInfo());

if (!ar.activity.mFinished && (allActivities || !ar.paused)) {

// If the activity is currently resumed, its configuration

// needs to change right now.

callbacks.add(a);

} else if (thisConfig != null) {

// Otherwise, we will tell it about the change

// the next time it is resumed or shown. Note that

// the activity manager may, before then, decide the

// activity needs to be destroyed to handle its new

// configuration.

if (DEBUG\_CONFIGURATION) {

Slog.v(TAG, "Setting activity " + ar.activityInfo.name + " newConfig=" + thisConfig);

}

ar.newConfig = thisConfig;

}

}

}

final int NSVC = mServices.size();// Service

for (int i=0; i<NSVC; i++) {

callbacks.add(mServices.valueAt(i));

}

}

synchronized (mProviderMap) {

final int NPRV = mLocalProviders.size();// Provider

for (int i=0; i<NPRV; i++) {

callbacks.add(mLocalProviders.valueAt(i).mLocalProvider);

}

}

return callbacks;

### onConfigurationChanged四大组件响应

如果在manifest.xml中配置了configChnages属性则表示由app自己来处理configuration change，就会回调Activity等组件的onConfigurationChanged方法。否则就重启当前这个activity（这个重启步骤位于当activity回到前台时执行onDestroy->onStart->onResume），而重启之前，旧的resources已经被清空, 那么就会装载新的资源。对于未启动的应用则会在启动时加载新的资源。参考如下

默认都没有通知的啊，为何为吊起onConfigurationChanged

if (reportToActivity) {

// Apply the ContextThemeWrapper override if necessary.

// NOTE: Make sure the configurations are not modified, as they are treated

// as immutable in many places.

final Configuration configToReport = createNewConfigAndUpdateIfNotNull(

newConfig, contextThemeWrapperOverrideConfig);

cb.onConfigurationChanged(configToReport);

}

## 其他

### OrientationEventListener

onOrientationChanged(int rotation)方法会将当前屏幕旋转的度数返回给用户，返回的旋转度数的计算方法;



上图中金色区域就是手机,角度就是绿线和红线之间的角度,顺时针旋转手机,角度增大,角度范围0-360;手机平放的角度为-1;

### 用户手动切换播放器方向和屏幕方向改变时自动切换并存

<https://www.cnblogs.com/a284628487/p/3361555.html>

## 语言切换

### 设置语言接口

final Locale[] newList2 = new Locale[]

{Locale.US, Locale.SIMPLIFIED\_CHINESE, Locale.TRADITIONAL\_CHINESE, Locale.JAPAN}

LocaleList ll = new LocaleList(newList2);  
LocalePicker.updateLocales(ll);

**public static void** updateLocales(LocaleList locales) {  
 **try** {  
 IActivityManager am = ActivityManagerNative.getDefault();  
 Configuration config = am.getConfiguration();  
 config.setLocales(locales);  
 config.userSetLocale = **true**;  
 am.updatePersistentConfiguration(config);  
 BackupManager.dataChanged(**"com.android.providers.settings"**);  
 } **catch** (RemoteException var3) {  
 ;  
 }  
  
}

如果app想要设置，通过反射可以试试

可以持久化系统语言设置，也就是说重启手机后不会恢复默认系统语言）：

#### 6.0反射

private void changeSystemLanguage(Locale locale) {

if (locale != null) {

try {

Class classActivityManagerNative = Class.forName("android.app.ActivityManagerNative");

Method getDefault = classActivityManagerNative.getDeclaredMethod("getDefault");

Object objIActivityManager = getDefault.invoke(classActivityManagerNative);

Class classIActivityManager = Class.forName("android.app.IActivityManager");

Method getConfiguration = classIActivityManager.getDeclaredMethod("getConfiguration");

Configuration config = (Configuration) getConfiguration.invoke(objIActivityManager);

config.setLocale(locale);

//config.userSetLocale = true;

Class clzConfig = Class

.forName("android.content.res.Configuration");

java.lang.reflect.Field userSetLocale = clzConfig

.getField("userSetLocale");

userSetLocale.set(config, true);

Class[] clzParams = {Configuration.class};

Method updateConfiguration = classIActivityManager.getDeclaredMethod("updateConfiguration", clzParams);

updateConfiguration.invoke(objIActivityManager, config);

BackupManager.dataChanged("com.android.providers.settings");

} catch (Exception e) {

Log.d(TAG, "changeSystemLanguage: " + e.getLocalizedMessage());

}

}

}

调用时如下：

Local locale = Locale.ENGLISH;

changeSystemLanguage(locale);

系统语言就变成了英文。（不再赘述）

#### 7.0反射

protected void changeSystemLanguage(LocaleList locale) {

if (locale != null) {

try {

Class classActivityManagerNative = Class.forName("android.app.ActivityManagerNative");

Method getDefault = classActivityManagerNative.getDeclaredMethod("getDefault");

Object objIActivityManager = getDefault.invoke(classActivityManagerNative);

Class classIActivityManager = Class.forName("android.app.IActivityManager");

Method getConfiguration = classIActivityManager.getDeclaredMethod("getConfiguration");

Configuration config = (Configuration) getConfiguration.invoke(objIActivityManager);

config.setLocales(locale);

Class[] clzParams = {Configuration.class};

Method updateConfiguration = classIActivityManager.getDeclaredMethod("updatePersistentConfiguration", clzParams);

updateConfiguration.invoke(objIActivityManager, config);

} catch (Exception e) {

Log.d(TAG, "changeSystemLanguage: " + e.getLocalizedMessage());

}

}

}

6.0和7.0设置系统语言源码不同之出，有两点：

1.设置参数的方法不同，6.0是updateConfiguration，7.0是updatePersistentConfiguration，这点需要注意

2.6.0传递的local，而7.0是一个列表LocaleList

7.0反射方法调用如下：

Locale newLocale = new Locale("zh", "CN");

final LocaleList localeList = new LocaleList(newLocale);

changeSystemLanguage(localeList);

---------------------

作者：zhaokai621

来源：CSDN

原文：https://blog.csdn.net/zhaokai621/article/details/77102308

版权声明：本文为博主原创文章，转载请附上博文链接！

---------------------

作者：zhaokai621

来源：CSDN

原文：https://blog.csdn.net/zhaokai621/article/details/77102308

版权声明：本文为博主原创文章，转载请附上博文链接！

---------------------

作者：zhaokai621

来源：CSDN

原文：https://blog.csdn.net/zhaokai621/article/details/77102308

版权声明：本文为博主原创文章，转载请附上博文链接！

### updatePersistentConfiguration

**if** (!initLocale && !values.getLocales().isEmpty() && values.userSetLocale) {

*//得到选择的国家语言列表*  
 **final** LocaleList locales = values.getLocales();  
 **int** bestLocaleIndex = 0;  
 **if** (locales.size() > 1) {  
 **if** (**mSupportedSystemLocales** == **null**) {

*//获取系统支持国家语言列表*  
 **mSupportedSystemLocales** = Resources.*getSystem*().getAssets().getLocales();  
 }

*//匹配国家，获取选择默认国家语言下标*  
 bestLocaleIndex = Math.max(0, locales.getFirstMatchIndex(**mSupportedSystemLocales**));  
 }  
 SystemProperties.set(**"persist.sys.locale"**,  
 locales.get(bestLocaleIndex).toLanguageTag());

*//设置为选择的国家语言为默认国家语言*  
 LocaleList.*setDefault*(locales, bestLocaleIndex);

*//发送消息通知挂载守护进程国家语言变更*  
 **mHandler**.sendMessage(**mHandler**.obtainMessage(***SEND\_LOCALE\_TO\_MOUNT\_DAEMON\_MSG***,  
 locales.get(bestLocaleIndex)));  
}

**private static native** @Nullable String[] nativeGetLocales(**long** ptr, **boolean** excludeSystem);

# Memory 管理

而从内存回收的角度来看，释放内存的地点包含三个。

第一个是在AmS中进行，即Android所声称的当系统内存低时，优先释放没有任何Activity的进程，然后释放非前台Activity对应的进程。

第二个是在OOM Killer中，此时AmS只要告诉OOM各个应用进程的优先级，然后OOM就会调用Linux内部的进程管理方法杀死优先级较低的进程。这个部分不在本书讲解的范围内。

第三个是在应用进程本身之中，当AmS认为目标进程需要被杀死时，首先会通知目标进程进行内存释放，这包括调用目标进程的scheduleLowMemory()方法和processInBackground()方法

# 进程优先级ADJ算法

## 概述

Z

### 1.1 进程

Android框架对进程创建与管理进行了封装，对于APP开发者只需知道Android四大组件的使用。当Activity, Service, ContentProvider, BroadcastReceiver任一组件启动时，当其所承载的进程存在则直接使用，不存在则由框架代码自动调用startProcessLocked创建进程。一个APP可以拥有多个进程，多个APP也可以运行在同一个进程，通过配置Android:process属性来决定。所以说对APP来说进程几乎是透明的，但了解进程对于深刻理解Android系统是至关关键的。

### 1.2 优先级

Android系统的设计理念正是希望应用进程能尽量长时间地存活，以提升用户体验。应用首次打开比较慢，这个过程有进程创建以及Application等信息的初始化，所以应用在启动之后，即便退到后台并非立刻杀死，而是存活一段时间，这样下次再使用则会非常快。对于APP同样希望自身尽可能存活更长的时间，甚至探索各种保活黑科技。物极必反，系统处于低内存的状态下，手机性能会有所下降；系统继续放任所有进程一直存活，系统内存很快就会枯竭而亡，那么需要合理地进程回收机制。

到底该回收哪个进程呢？系统根据进程的组件状态来决定每个进程的优先级值ADJ，系统根据一定策略先杀优先级最低的进程，然后逐步杀优先级更低的进程，依此类推，以回收预期的可用系统资源，从而保证系统正常运转。

谈到优先级，可能有些人会想到Linux进程本身有nice值，这个能决定CPU资源调度的优先级；而本文介绍Android系统中的ADJ，主要决定在什么场景下什么类型的进程可能会被杀，影响的是进程存活时间。ADJ与nice值两者定位不同，不过也有一定的联系，优先级很高的进程，往往也是用户不希望被杀的进程，是具有有一定正相关性。

### 1.3 ADJ级别

| **ADJ级别** | **取值** | **含义** |
| --- | --- | --- |
| NATIVE\_ADJ | -1000 | native进程, 是由init进程fork出来的Native进程，并不受system管控 |
| SYSTEM\_ADJ | -900 | 仅指system\_server进程 |
| PERSISTENT\_PROC\_ADJ | -800 | 系统persistent进程, 是指在AndroidManifest.xml中申明android:persistent=”true”的系统(即带有FLAG\_SYSTEM标记)进程，persistent进程一般情况并不会被杀，即便被杀或者发生Crash系统会立即重新拉起该进程。 |
| PERSISTENT\_SERVICE\_ADJ | -700 | 关联着系统或persistent进程. 是由startIsolatedProcess()方式启动的进程，或者是由system\_server或者persistent进程所绑定(并且带有BIND\_ABOVE\_CLIENT或者BIND\_IMPORTANT)的服务进程 |
| FOREGROUND\_APP\_ADJ | 0 | 前台进程 |
| VISIBLE\_APP\_ADJ | 100 | 可见进程 |
| PERCEPTIBLE\_APP\_ADJ | 200 | 可感知进程，比如后台音乐播放 |
| BACKUP\_APP\_ADJ | 300 | 备份进程. 执行bindBackupAgent()过程的进程 |
| HEAVY\_WEIGHT\_APP\_ADJ | 400 | 重量级进程, realStartActivityLocked()过程，当应用的privateFlags标识PRIVATE\_FLAG\_CANT\_SAVE\_STATE的进程； |
| SERVICE\_ADJ | 500 | 服务进程 |
| HOME\_APP\_ADJ | 600 | Home进程. 当类型为ACTIVITY\_TYPE\_HOME的应用，比如桌面APP |
| PREVIOUS\_APP\_ADJ | 700 | 上一个进程 |
| SERVICE\_B\_ADJ | 800 | B List中的Service |
| CACHED\_APP\_MIN\_ADJ | 900 | 不可见进程的adj最小值 |
| CACHED\_APP\_MAX\_ADJ | 906 | 不可见进程的adj最大值 |

从Android 7.0开始，ADJ采用100、200、300；在这之前的版本ADJ采用数字1、2、3，这样的调整可以更进一步地细化进程的优先级，比如在VISIBLE\_APP\_ADJ(100)与PERCEPTIBLE\_APP\_ADJ(200)之间，可以有ADJ=101、102级别的进程。

* 省去lmk对oom\_score\_adj的计算过程，Android 7.0之前的版本，oom\_score\_adj= oom\_adj \* 1000/17； 而Android 7.0开始，oom\_score\_adj= oom\_adj，不用再经过一次转换。

### 1.4 LMK

为了防止剩余内存过低，Android在内核空间有lowmemorykiller(简称LMK)，LMK是通过注册shrinker来触发低内存回收的，这个机制并不太优雅，可能会拖慢Shrinkers内存扫描速度，已从内核4.12中移除，后续会采用用户空间的LMKD + memory cgroups机制，这里先不展开LMK讲解。

进程刚启动时ADJ等于INVALID\_ADJ，当执行完attachApplication()，该该进程的curAdj和setAdj不相等，则会触发执行setOomAdj()将该进程的节点/proc/pid/oom\_score\_adj写入oomadj值。下图参数为Android原生阈值，当系统剩余空闲内存低于某阈值(比如147MB)，则从ADJ大于或等于相应阈值(比如900)的进程中，选择ADJ值最大的进程，如果存在多个ADJ相同的进程，则选择内存最大的进程。 如下是64位机器，LMK默认阈值图：



在updateOomLevels()过程，会根据手机屏幕尺寸或内存大小来调整scale，默认大多数手机内存都大于700MB，则scale等于1。对于64位手机，阈值会更大些，具体如下。

ProcessList.java

**private void** updateOomLevels(**int** displayWidth, **int** displayHeight, **boolean** write)

**for** (**int** i=0; i<mOomAdj.length; i++) {  
 **int** low = mOomMinFreeLow[i];  
 **int** high = mOomMinFreeHigh[i];  
 **if** (is64bit) {  
 *// Increase the high min-free levels for cached processes for 64-bit* **if** (i == 4) high = (high\*3)/2;  
 **else if** (i == 5) high = (high\*7)/4;  
 }  
 mOomMinFree[i] = (**int**)(low + ((high-low)\*scale));  
}

## 解读ADJ

接下来，解读每个ADJ值都对应着怎样条件的进程，包括正在运行的组件以及这些组件的状态几何。这里重点介绍上图标红的ADJ级别所对应的进程。

Android系统中计算各进程ADJ算法的核心方法:

* updateOomAdjLocked：更新adj，当目标进程为空或者被杀则返回false；否则返回true;
* computeOomAdjLocked：计算adj，返回计算后RawAdj值;
* applyOomAdjLocked：应用adj，当需要杀掉目标进程则返回false；否则返回true。

当Android四大组件状态改变时会updateOomAdjLocked()来同步更新相应进程的ADJ优先级。这里需要说明一下，当同一个进程有多个决定其优先级的组件状态时，取优先级最高的ADJ作为最终的ADJ。另外，进程会通过设置maxAdj来限定ADJ的上限。

关于分析进程ADJ相关信息，常用命令如下：

* dumpsys meminfo，
* dumpsys activity o
* dumpsys activity p

### SYSTEM\_ADJ(-900)

SYSTEM\_ADJ: 仅指system\_server进程。在执行SystemServer的startBootstrapServices()过程会调用AMS.setSystemProcess()，将system\_server进程的maxAdj设置成SYSTEM\_ADJ，源码如下：

**public void** setSystemProcess() {  
 **try** {  
 ApplicationInfo info = mContext.getPackageManager().getApplicationInfo(  
 **"android"**, STOCK\_PM\_FLAGS | MATCH\_SYSTEM\_ONLY);  
 mSystemThread.installSystemApplicationInfo(info, getClass().getClassLoader());  
  
 **synchronized** (**this**) {  
 ProcessRecord app = newProcessRecordLocked(info, info.processName, **false**, 0);  
 app.persistent = **true**;  
 app.pid = MY\_PID;  
 app.maxAdj = ProcessList.SYSTEM\_ADJ;  
 app.makeActive(mSystemThread.getApplicationThread(), mProcessStats);  
 **synchronized** (mPidsSelfLocked) {  
 mPidsSelfLocked.put(app.pid, app);  
 }  
 updateLruProcessLocked(app, **false**, **null**);  
 updateOomAdjLocked();  
 }  
 } **catch** (PackageManager.NameNotFoundException e) {  
 **throw new** RuntimeException(  
 **"Unable to find android system package"**, e);  
 }  
}

但system\_server的ADJ并非等于-900，而是-800？是由于startPersistentApps()过程直接把其adj重新被设置为-800，这算是一个小BUG，但 其实目前来说对于ADJ<0的进程，LMK不会杀，两者没有什么区别。

### PERSISTENT\_PROC\_ADJ(-800)

PERSISTENT\_PROC\_ADJ：在AndroidManifest.xml中申明android:persistent=”true”的系统(即带有FLAG\_SYSTEM标记)进程，称之为persistent进程。对于persistent进程常规情况都不会被杀，一旦被杀或者发生Crash，进程会立即重启。

AMS.addAppLocked(）或AMS.newProcessRecordLocked()过程会赋值：

#### AMS.newProcessRecordLocked

**final** ProcessRecord newProcessRecordLocked(ApplicationInfo info, String customProcess,  
 **boolean** isolated, **int** isolatedUid) {  
 String proc = customProcess != **null** ? customProcess : info.processName;  
 **final** ProcessRecord r = **new** ProcessRecord(stats, info, proc, uid);  
 **if** (!mBooted && !mBooting  
 && userId == UserHandle.USER\_SYSTEM  
 && (info.flags & PERSISTENT\_MASK) == PERSISTENT\_MASK) {  
 r.persistent = **true**;  
 r.maxAdj = ProcessList.PERSISTENT\_PROC\_ADJ;  
 }  
 addProcessNameLocked(r);  
 **return** r;  
}

在每一次进程启动的时候都会判断该进程是否persistent进程，如果是则会设置maxAdj=PERSISTENT\_PROC\_ADJ。 system\_server进程应该也是persistent进程,但是会被覆盖为app.maxAdj = ProcessList.SYSTEM\_ADJ;

#### addAppLocked

**final** ProcessRecord addAppLocked(ApplicationInfo info, String customProcess, **boolean** isolated,  
 String abiOverride) {  
 ProcessRecord app;  
 **if** (app == **null**) {  
 app = newProcessRecordLocked(info, customProcess, isolated, 0);  
 updateLruProcessLocked(app, **false**, **null**);  
 updateOomAdjLocked();  
 }  
 **if** ((info.flags & PERSISTENT\_MASK) == PERSISTENT\_MASK) {  
 app.persistent = **true**;  
 app.maxAdj = ProcessList.PERSISTENT\_PROC\_ADJ;  
 }  
 **if** (app.thread == **null** && mPersistentStartingProcesses.indexOf(app) < 0) {  
 mPersistentStartingProcesses.add(app);  
 startProcessLocked(app, **"added application"**,  
 customProcess != **null** ? customProcess : app.processName, abiOverride,  
 **null** */\* entryPoint \*/*, **null** */\* entryPointArgs \*/*);  
 }  
  
 **return** app;  
}

as

### PERSISTENT\_SERVICE\_ADJ(-700)

PERSISTENT\_SERVICE\_ADJ: startIsolatedProcess()方式启动的进程，或者是由system\_server或者persistent进程所绑定的服务进程。

**场景1：**newProcessRecordLocked

final ProcessRecord newProcessRecordLocked(ApplicationInfo info, String customProcess, boolean isolated, int isolatedUid) {

String proc = customProcess != null ? customProcess : info.processName;

final int userId = UserHandle.getUserId(info.uid);

int uid = info.uid;

...

final ProcessRecord r = new ProcessRecord(stats, info, proc, uid);

if (!mBooted && !mBooting

&& userId == UserHandle.USER\_SYSTEM

&& (info.flags & PERSISTENT\_MASK) == PERSISTENT\_MASK) {

r.persistent = true;

r.maxAdj = ProcessList.PERSISTENT\_PROC\_ADJ;

}

if (isolated && isolatedUid != 0) { //startIsolatedProcess

r.maxAdj = ProcessList.PERSISTENT\_SERVICE\_ADJ;

}

return r;

}

调用链：

startOtherServices

WebViewUpdateService.prepareWebViewInSystemServer

WebViewUpdateServiceImpl.prepareWebViewInSystemServer

WebViewUpdater.prepareWebViewInSystemServer

WebViewUpdater.onWebViewProviderChanged

SystemImpl.onWebViewProviderChanged

WebViewFactory.onWebViewProviderChanged

WebViewLibraryLoader.prepareNativeLibraries

WebViewLibraryLoader.createRelros

WebViewLibraryLoader.createRelroFile

AMS.startIsolatedProcess

### BACKUP\_APP\_ADJ(300)

if (mBackupTarget != null && app == mBackupTarget.app) {

if (adj > ProcessList.BACKUP\_APP\_ADJ) {

adj = ProcessList.BACKUP\_APP\_ADJ;

if (procState > ActivityManager.PROCESS\_STATE\_TRANSIENT\_BACKGROUND) {

procState = ActivityManager.PROCESS\_STATE\_TRANSIENT\_BACKGROUND;

}

app.adjType = "backup";

app.cached = false;

}

if (procState > ActivityManager.PROCESS\_STATE\_BACKUP) {

procState = ActivityManager.PROCESS\_STATE\_BACKUP;

app.adjType = "backup";

}

}

* 执行bindBackupAgent()过程，设置mBackupTarget值；
* 执行clearPendingBackup()或unbindBackupAgent()过程，置空mBackupTarget值；

### HEAVY\_WEIGHT\_APP\_ADJ(400)

* realStartActivityLocked()过程，当应用的privateFlags标识PRIVATE\_FLAG\_CANT\_SAVE\_STATE，设置mHeavyWeightProcess值；
* finishHeavyWeightApp(), 置空mHeavyWeightProcess值；

### HOME\_APP\_ADJ(600)

当类型为ACTIVITY\_TYPE\_HOME的应用启动后会设置mHomeProcess，比如桌面APP。

### PREVIOUS\_APP\_ADJ(700)

##### activityStoppedLocked

用户上一个使用的包含UI的进程，为了给用户在两个APP之间更好的切换体验，将上一个进程ADJ设置到PREVIOUS\_APP\_ADJ的档次。 当activityStoppedLocked()过程会更新上一个应用

computeOomAdjLocked(ProcessRecord app, **int** cachedAdj, ProcessRecord TOP\_APP,  
 **boolean** doingAll, **long** now)

**if** (app == mPreviousProcess && app.activities.size() > 0) {  
 **if** (adj > ProcessList.PREVIOUS\_APP\_ADJ) {  
 *// This was the previous process that showed UI to the user.  
 // We want to try to keep it around more aggressively, to give  
 // a good experience around switching between two apps.* adj = ProcessList.PREVIOUS\_APP\_ADJ;  
 schedGroup = ProcessList.SCHED\_GROUP\_BACKGROUND;  
 app.cached = **false**;  
 app.adjType = **"previous"**;  
 **if** (DEBUG\_OOM\_ADJ\_REASON) Slog.d(TAG, **"Raise to prev: "** + app);  
 }  
 **if** (procState > ActivityManager.PROCESS\_STATE\_LAST\_ACTIVITY) {  
 procState = ActivityManager.PROCESS\_STATE\_LAST\_ACTIVITY;  
 app.adjType = **"previous"**;  
 **if** (DEBUG\_OOM\_ADJ\_REASON) Slog.d(TAG, **"Raise to prev: "** + app);  
 }  
}

##### **场景2**

当provider进程，上一次使用时间不超过20S的情况下，优先级不低于PREVIOUS\_APP\_ADJ。provider进程这个是Android 7.0以后新增的逻辑 ，这样做的好处是在内存比较低的情况下避免拥有provider的进程出现颠簸，也就是启动后杀，然后又被拉

**if** (app.lastProviderTime > 0 &&  
 (app.lastProviderTime+mConstants.CONTENT\_PROVIDER\_RETAIN\_TIME) > now) {  
 **if** (adj > ProcessList.PREVIOUS\_APP\_ADJ) {  
 adj = ProcessList.PREVIOUS\_APP\_ADJ;  
 schedGroup = ProcessList.SCHED\_GROUP\_BACKGROUND;  
 app.cached = **false**;  
 app.adjType = **"recent-provider"**;  
 **if** (DEBUG\_OOM\_ADJ\_REASON) Slog.d(TAG, **"Raise to recent provider: "** + app);  
 }  
 **if** (procState > ActivityManager.PROCESS\_STATE\_LAST\_ACTIVITY) {  
 procState = ActivityManager.PROCESS\_STATE\_LAST\_ACTIVITY;  
 app.adjType = **"recent-provider"**;  
 **if** (DEBUG\_OOM\_ADJ\_REASON) Slog.d(TAG, **"Raise to recent provider: "** + app);  
 }  
}

### FOREGROUND\_APP\_ADJ(0)

#### 场景1：

满足以下任一条件的进程都属于FOREGROUND\_APP\_ADJ(0)优先级：

* 正处于resumed状态的Activity
* 正执行一个生命周期回调的Service（比如执行onCreate,onStartCommand,onDestroy等）
* 正执行onReceive()的BroadcastReceiver
* 通过startInstrumentation()启动的进程

Sfd

**if** (app == TOP\_APP) {  
 *// The last app on the list is the foreground app.* adj = ProcessList.FOREGROUND\_APP\_ADJ;  
 schedGroup = ProcessList.SCHED\_GROUP\_TOP\_APP;  
 app.adjType = **"top-activity"**;  
 foregroundActivities = **true**;  
 procState = PROCESS\_STATE\_CUR\_TOP;  
 **if** (DEBUG\_OOM\_ADJ\_REASON) Slog.d(TAG, **"Making top: "** + app);  
} **else if** (app.instr != **null**) {  
 *// Don't want to kill running instrumentation.* adj = ProcessList.FOREGROUND\_APP\_ADJ;  
 schedGroup = ProcessList.SCHED\_GROUP\_DEFAULT;  
 app.adjType = **"instrumentation"**;  
 procState = ActivityManager.PROCESS\_STATE\_FOREGROUND\_SERVICE;  
 **if** (DEBUG\_OOM\_ADJ\_REASON) Slog.d(TAG, **"Making instrumentation: "** + app);  
} **else if** (isReceivingBroadcastLocked(app, mTmpBroadcastQueue)) {  
 *// An app that is currently receiving a broadcast also  
 // counts as being in the foreground for OOM killer purposes.  
 // It's placed in a sched group based on the nature of the  
 // broadcast as reflected by which queue it's active in.* adj = ProcessList.FOREGROUND\_APP\_ADJ;  
 schedGroup = (mTmpBroadcastQueue.contains(mFgBroadcastQueue))  
 ? ProcessList.SCHED\_GROUP\_DEFAULT : ProcessList.SCHED\_GROUP\_BACKGROUND;  
 app.adjType = **"broadcast"**;  
 procState = ActivityManager.PROCESS\_STATE\_RECEIVER;  
 **if** (DEBUG\_OOM\_ADJ\_REASON) Slog.d(TAG, **"Making broadcast: "** + app);  
}

#### **场景2：**

**场景2：** 当客户端进程activity里面调用bindService()方法时flags带有BIND\_ADJUST\_WITH\_ACTIVITY参数，并且该activity处于可见状态，则当前服务进程也属于前台进程，源码如下：

for (int is = app.services.size()-1; is >= 0; is--) {

ServiceRecord s = app.services.valueAt(is);

for (int conni = s.connections.size()-1; conni >= 0; conni--) {

ArrayList<ConnectionRecord> clist = s.connections.valueAt(conni);

for (int i = 0; i < clist.size(); i++) {

ConnectionRecord cr = clist.get(i);

if ((cr.flags&Context.BIND\_WAIVE\_PRIORITY) == 0) {

...

}

final ActivityRecord a = cr.activity;

if ((cr.flags&Context.BIND\_ADJUST\_WITH\_ACTIVITY) != 0) {

if (a != null && adj > ProcessList.FOREGROUND\_APP\_ADJ &&

(a.visible || a.state == ActivityState.RESUMED ||

a.state == ActivityState.PAUSING)) {

adj = ProcessList.FOREGROUND\_APP\_ADJ;

if ((cr.flags&Context.BIND\_NOT\_FOREGROUND) == 0) {

if ((cr.flags&Context.BIND\_IMPORTANT) != 0) {

schedGroup = ProcessList.SCHED\_GROUP\_TOP\_APP\_BOUND;

} else {

schedGroup = ProcessList.SCHED\_GROUP\_DEFAULT;

}

}

app.cached = false;

app.adjType = "service";

app.adjTypeCode = ActivityManager.RunningAppProcessInfo

.REASON\_SERVICE\_IN\_USE;

app.adjSource = a;

app.adjSourceProcState = procState;

app.adjTarget = s.name;

}

}

}

}

}

#### **场景3：** 对于provider进程，

还有以下两个条件能成为前台进程：

* 当Provider的客户端进程ADJ<=FOREGROUND\_APP\_ADJ时，则Provider进程ADJ等于FOREGROUND\_APP\_ADJ
* 当Provider有外部(非框架)进程依赖，也就是调用了getContentProviderExternal()方法，则ADJ至少等于FOREGROUND\_APP\_ADJ
* **if** (adj > clientAdj) {  
   **if** (app.hasShownUi && app != mHomeProcess  
   && clientAdj > ProcessList.PERCEPTIBLE\_APP\_ADJ) {  
   adjType = **"cch-ui-provider"**;  
   } **else** {  
   adj = clientAdj > ProcessList.FOREGROUND\_APP\_ADJ  
   ? clientAdj : ProcessList.FOREGROUND\_APP\_ADJ;  
   adjType = **"provider"**;  
   }  
   app.cached &= client.cached;  
  }

*// If the provider has external (non-framework) process  
// dependencies, ensure that its adjustment is at least  
// FOREGROUND\_APP\_ADJ.***if** (cpr.hasExternalProcessHandles()) {  
 **if** (adj > ProcessList.FOREGROUND\_APP\_ADJ) {  
 adj = ProcessList.FOREGROUND\_APP\_ADJ;  
 schedGroup = ProcessList.SCHED\_GROUP\_DEFAULT;  
 app.cached = **false**;  
 app.adjType = **"ext-provider"**;  
 app.adjTarget = cpr.name;  
 **if** (DEBUG\_OOM\_ADJ\_REASON) Slog.d(TAG, **"Raise to external provider: "** + app);  
 }  
 **if** (procState > ActivityManager.PROCESS\_STATE\_IMPORTANT\_FOREGROUND) {  
 procState = ActivityManager.PROCESS\_STATE\_IMPORTANT\_FOREGROUND;  
 }  
}

### VISIBLE\_APP\_ADJ(100)

可见进程：当ActivityRecord的visible=true，也就是Activity可见的进程。

**if** (r.visible) {  
 *// App has a visible activity; only upgrade adjustment.* **if** (adj > ProcessList.VISIBLE\_APP\_ADJ) {  
 adj = ProcessList.VISIBLE\_APP\_ADJ;  
 app.adjType = **"vis-activity"**;  
 **if** (DEBUG\_OOM\_ADJ\_REASON) Slog.d(TAG, **"Raise to vis-activity: "** + app);  
 }

从Android P开始，进一步细化ADJ级别，增加了VISIBLE\_APP\_LAYER\_MAX(99)，是指VISIBLE\_APP\_ADJ(100)跟PERCEPTIBLE\_APP\_ADJ(200)之间有99个槽，则可见级别ADJ的取值范围为[100,199]。 算法会根据其所在task的mLayerRank来调整其ADJ，100加上mLayerRank就等于目标ADJ，layer越大，则ADJ越小。

**if** (adj == ProcessList.VISIBLE\_APP\_ADJ) {  
 adj += minLayer;  
}

关于TaskRecord的mLayerRank的计算方式是在updateOomAdjLocked()过程调用ASS的rankTaskLayersIfNeeded()方法，如下：

[-> ActivityStackSupervisor.java]

**void** rankTaskLayersIfNeeded() {  
 **if** (!mTaskLayersChanged) {  
 **return**;  
 }  
 mTaskLayersChanged = **false**;  
 **for** (**int** displayNdx = 0; displayNdx < mActivityDisplays.size(); displayNdx++) {  
 **final** ArrayList<ActivityStack> stacks = mActivityDisplays.valueAt(displayNdx).mStacks;  
 **int** baseLayer = 0;  
 **for** (**int** stackNdx = stacks.size() - 1; stackNdx >= 0; --stackNdx) {  
 baseLayer += stacks.get(stackNdx).rankTaskLayers(baseLayer);  
 }  
 }  
}

[-> ActivityStack.java]

**final int** rankTaskLayers(**int** baseLayer) {  
 **int** layer = 0;  
 **for** (**int** taskNdx = mTaskHistory.size() - 1; taskNdx >= 0; --taskNdx) {  
 **final** TaskRecord task = mTaskHistory.get(taskNdx);  
 ActivityRecord r = task.topRunningActivityLocked();  
 **if** (r == **null** || r.finishing || !r.visible) {  
 task.mLayerRank = -1;  
 } **else** {  
 task.mLayerRank = baseLayer + layer++;  
 }  
 }  
 **return** layer;  
}

当TaskRecord顶部的ActivityRecord为空或者结束或者不可见时，则设置该TaskRecord的mLayerRank等于-1; 每个ActivityDisplay的baseLayer都是从0开始，从最上面的TaskRecord开始，第一个ADJ=100，从上至下依次加1，直到199为上限。



#### service客户端

ServiceRecord的成员变量startRequested=true，是指被显式调用了startService()方法。当service被stop或kill会将其置为false。

一般情况下，即便客户端进程处于前台进程(ADJ=0)级别，服务进程只会提升到可见(ADJ=1)级别。以下flags是由调用bindService()过程所传递的flags来决定的。

| **flag** | **含义** |
| --- | --- |
| BIND\_WAIVE\_PRIORITY | 是指客户端进程的优先级不会影响目标服务进程的优先级。比如当调用bindService又不希望提升目标服务进程的优先级的情况下，可以使用该flags |
| BIND\_ADJUST\_WITH\_ACTIVITY | 是指当从Activity绑定到该进程时，允许目标服务进程根据该activity的可见性来提升优先级 |
| BIND\_ABOVE\_CLIENT | 当客户端进程绑定到一个服务进程时，则服务进程比客户端进程更重要 |
| BIND\_IMPORTANT | 标记该服务对于客户端进程很重要，当客户端进程处于前台进程(ADJ=0)级别时，会把服务进程也提升到前台进程级别 |
| BIND\_NOT\_VISIBLE | 当客户端进程处于可见(ADJ=1)级别，也不允许被绑定的服务进程提升到可见级别，该类服务进程的优先级上限为可感知(ADJ=2)级别 |
| BIND\_NOT\_FOREGROUND | 不允许被绑定的服务进程提升到前台调度优先级，但是内存优先级可以提升到前台级别。比如不希望服务进程占用 |

作为工程师很多时候可能还是想看看源码，show me the code。但是关于ADJ计算这一块源码场景computeOomAdjLocked()，Google真心写得比较乱，为了更清晰地说明客户端进程如何影响服务进程，在保证不失去原意的情况下重写了这块部分逻辑：

这个过程主要根据service本身、client端情况以及activity状态分别来调整adj和schedGroup

for (int is = app.services.size()-1; is >= 0; is--) {

ServiceRecord s = app.services.valueAt(is);

if (s.startRequested) {

... // 根据service本身调整adj和adjType

}

for (int conni = s.connections.size()-1; conni >= 0; conni--) {

ArrayList<ConnectionRecord> clist = s.connections.valueAt(conni);

for (int i = 0; i < clist.size(); i++) {

ConnectionRecord cr = clist.get(i);

//根据client端来调整adj

if ((cr.flags&Context.BIND\_WAIVE\_PRIORITY) == 0) {

if (adj > clientAdj) {

if (app.hasShownUi && app != mHomeProcess

&& clientAdj > ProcessList.PERCEPTIBLE\_APP\_ADJ) {

...

} else {

int newAdj = clientAdj;

if ((cr.flags&(Context.BIND\_ABOVE\_CLIENT

|Context.BIND\_IMPORTANT)) != 0) {

if(clientAdj < ProcessList.PERSISTENT\_SERVICE\_ADJ) {

newAdj = PERSISTENT\_SERVICE\_ADJ;

}

} else if ((cr.flags&Context.BIND\_NOT\_VISIBLE) != 0) {

if(clientAdj < ProcessList.PERCEPTIBLE\_APP\_ADJ) {

newAdj = PERCEPTIBLE\_APP\_ADJ;

}

} else {

if (clientAdj < ProcessList.VISIBLE\_APP\_ADJ) {

newAdj = VISIBLE\_APP\_ADJ;

}

}

if (adj > newAdj) {

adj = newAdj;

adjType = "service";

}

}

}

}

final ActivityRecord a = cr.activity;

// 根据client的activity来调整adj和schedGroup

if ((cr.flags&Context.BIND\_ADJUST\_WITH\_ACTIVITY) != 0) {

...

}

}

}

}

上段代码说明服务端进程优先级(adj)不会低于客户端进程优先级(newAdj)，而newAdj的上限受限于flags，具体服务端进程受客户端进程影响的ADJ上限如下：

BIND\_ABOVE\_CLIENT或BIND\_IMPORTANT的情况下，ADJ上限为PERSISTENT\_SERVICE\_ADJ；

BIND\_NOT\_VISIBLE的情况下， ADJ上限为PERCEPTIBLE\_APP\_ADJ；

否则，一般情况下，ADJ上限为VISIBLE\_APP\_ADJ；

由此，可见当bindService过程带有BIND\_ABOVE\_CLIENT或者BIND\_IMPORTANT flags的同时，客户端进程ADJ小于或等于PERSISTENT\_SERVICE\_ADJ的情况下，该进程则为PERSISTENT\_SERVICE\_ADJ。另外，即便是启动过Activity的进程，当客户端进程ADJ<=200时，还是可以提升该服务进程的优先级。

### PERCEPTIBLE\_APP\_ADJ(200)

可感知进程：当该进程存在不可见的Activity，但Activity正处于PAUSING、PAUSED、STOPPING状态，则为PERCEPTIBLE\_APP\_ADJ

## releaseSomeActivitiesLocked

首先我们要找到整个内存不足然后回调回收activity的入口，在哪里呢？在ActivityThread里，通过之前的源码阅读我们发现，当一个新的app启动的时候，系统将从Zygote进程fork一个子进程出来，当然我们知道App不一定只存在一个进程，只要定义了Activity进程要游离，我们也可以将其游离，但本质上都是由核心Android 进程 Zygote 进程去启动，但app进程启动之后，ActivityThread将被初始化，代码如下：

AppThread attch（）有一段gc监听的代码

这里添加Gc的监听，如果超过虚拟机分配最大内存的 3/4，那么触发mgr.releaseSomeActivities

### Watcher

*// Watch for getting close to heap limit.*BinderInternal.addGcWatcher(**new** Runnable() {  
 @Override **public void** run() {  
 **if** (!mSomeActivitiesChanged) {  
 **return**;  
 }  
 Runtime runtime = Runtime.getRuntime();  
 **long** dalvikMax = runtime.maxMemory();  
 **long** dalvikUsed = runtime.totalMemory() - runtime.freeMemory();  
 **if** (dalvikUsed > ((3\*dalvikMax)/4)) {  
 **if** (DEBUG\_MEMORY\_TRIM) Slog.d(TAG, **"Dalvik max="** + (dalvikMax/1024)  
 + **" total="** + (runtime.totalMemory()/1024)  
 + **" used="** + (dalvikUsed/1024));  
 mSomeActivitiesChanged = **false**;  
 **try** {  
 mgr.releaseSomeActivities(mAppThread);  
 } **catch** (RemoteException e) {  
 **throw** e.rethrowFromSystemServer();  
 }  
 }  
 }  
});

我们很明显的看到了一个 BinderInternal.addGcWatcher()这个方法，我们可以看字面意思就知道，这个是一个Gc 回收的监听器，作用是在AppThread 作用域下的GcRoots Gc开始的时候，我们去监听这个GC回收，具体如何做到呢？我们量看一下GcWatcher 的实现思路

#### BinderInternal.addGcWatcher

**public static void** addGcWatcher(Runnable watcher) {  
 **synchronized** (sGcWatchers) {  
 sGcWatchers.add(watcher);  
 }  
}

#### BinderInternal.GcWatcher

**static final class** GcWatcher {  
 @Override  
 **protected void** finalize() **throws** Throwable {  
 handleGc();  
 sLastGcTime = SystemClock.uptimeMillis();  
 **synchronized** (sGcWatchers) {  
 sTmpWatchers = sGcWatchers.toArray(sTmpWatchers);  
 }  
 **for** (**int** i=0; i<sTmpWatchers.length; i++) {  
 **if** (sTmpWatchers[i] != **null**) {  
 sTmpWatchers[i].run();  
 }  
 }  
 sGcWatcher = **new** WeakReference<GcWatcher>(**new** GcWatcher());  
 }  
}

这里重写了finallize()方法，从JVM原理我们就知道，第一次进行GC回收的时候，GC线程将会回调finallize()方法，注意注意！！！敲黑板！这里有个sGcWatcher, 这玩意儿是干嘛的？来来来，我们看看sGcWatcher 的定义

#### BinderInternal .sGcWatcher

---------------------

//路径 BinderInternal.java

static WeakReference<GcWatcher> sGcWatcher

= new WeakReference<GcWatcher>(new GcWatcher());

static ArrayList<Runnable> sGcWatchers = new ArrayList<>();

static Runnable[] sTmpWatchers = new Runnable[1];

static long sLastGcTime;

我们看到了这是一个静态弱引用GcWatcher, 目的是啥？因为因为，如果静态强引用，这个引用就存在静态引用方法区，这时这个强引用GC线程无法回收！！所以这里当然要使用弱引用，当Gc 回收之后触发弱引用的scGcWather 的finallize()方法，但是此时只会触发一次啊！当然没关系，因为在方法最后我们又重新开辟了一个新的弱引用对象，所以，这个引用逃逸了，逃过一劫。原来如此。

### App回收是何时触发: forceGc

BinderInternal里的方法：

//路径 BinderInternal.java

**public static void** forceGc(String reason) {  
 EventLog.writeEvent(2741, reason);  
 VMRuntime.getRuntime().requestConcurrentGC();  
}

Ok, 我们看到这个方法一定会去触发运行时GC，那这个方法在何处调用呢？我们倒着走回去看看

分析源码发现，这里有两条线，一条是 doGcIfNeeded()方法，一条是handleLowMemory().

先分析第一条线，doGcIfNeeded()方法，我们继续往上走，看看，是谁来调用了我们的doGcIfNeeded()方法

---------------------

#### ActivityThread. doGcIfNeeded

我们之后再来分析第二条线，这条线是系统检测内存不足的情况出发的，现在

Safd

**void** doGcIfNeeded() {  
 mGcIdlerScheduled = **false**;  
 **final long** now = SystemClock.uptimeMillis();  
 *//Slog.i(TAG, "\*\*\*\* WE MIGHT WANT TO GC: then=" + Binder.getLastGcTime()  
 // + "m now=" + now);* **if** ((BinderInternal.getLastGcTime()+MIN\_TIME\_BETWEEN\_GCS) < now) {  
 *//Slog.i(TAG, "\*\*\*\* WE DO, WE DO WANT TO GC!");* BinderInternal.forceGc(**"bg"**);  
 }  
}

##### ActivityThread.GcIdler

**final class** GcIdler **implements** MessageQueue.IdleHandler {  
 @Override  
 **public final boolean** queueIdle() {  
 doGcIfNeeded();  
 **return false**;  
 }  
}

GcIdler 是一个扩展了IdleHandler的类，我们翻一下这个类的注释描述会发现有这么一段

这个接口是在Messagequeue队列里的所有message都已经处理完了之后想等待更多的handler，如果queueIdle()返回true，这个handler就会一直保持存活，false执行完就丢弃，如果有持续的消息从IdleHandler进入，那么将在队列里等待。

##### ActivityThread.scheduleGcIdler

**void** scheduleGcIdler() {  
 **if** (!mGcIdlerScheduled) {  
 mGcIdlerScheduled = **true**;  
 Looper.myQueue().addIdleHandler(mGcIdler);  
 }  
 mH.removeMessages(H.GC\_WHEN\_IDLE);  
}  
  
**void** unscheduleGcIdler() {  
 **if** (mGcIdlerScheduled) {  
 mGcIdlerScheduled = **false**;  
 Looper.myQueue().removeIdleHandler(mGcIdler);  
 }  
 mH.removeMessages(H.GC\_WHEN\_IDLE);  
}

在ActivityThread 里实例化的，终于又回到了ActivityThread

Looper.myQueue() ，也就是主线程里的handler线程队列内容全部处理 结束，这个GcIdler 的 queueIdle() 就会被触发，那么GC就会被触发。已经到这，大概我们就明白GC调用的时机，我们接下来继续往上走，看看，何时我们会把这个GcIdler 加入执行队列，一直往上走我们来到了ActivityMS, 看里面的这段代码

##### ActivityMS.performAppGcsLocked

**final void** performAppGcsLocked() {  
 **final int** N = mProcessesToGc.size();  
 **if** (N <= 0) {  
 **return**;  
 }  
 **if** (canGcNowLocked()) {  
 **while** (mProcessesToGc.size() > 0) {  
 ProcessRecord proc = mProcessesToGc.remove(0);  
 **if** (proc.curRawAdj > ProcessList.PERCEPTIBLE\_APP\_ADJ || proc.reportLowMemory) {  
 **if** ((proc.lastRequestedGc+mConstants.GC\_MIN\_INTERVAL)  
 <= SystemClock.uptimeMillis()) {  
 *// To avoid spamming the system, we will GC processes one  
 // at a time, waiting a few seconds between each.* performAppGcLocked(proc);  
 scheduleAppGcsLocked();  
 **return**;  
 } **else** {  
 *// It hasn't been long enough since we last GCed this  
 // process... put it in the list to wait for its time.* addProcessToGcListLocked(proc);  
 **break**;  
 }  
 }  
 }  
  
 scheduleAppGcsLocked();  
 }  
}

上面的代码，我们看这一段， if (proc.curRawAdj > ProcessList.PERCEPTIBLE\_APP\_ADJ || proc.reportLowMemory),这个从字面理解就是，如果目前的oom\_adj 比 ProcessList.PERCEPTIBLE\_APP\_ADJ 级别要高，或者进程在低内存环境下运行，就会触发这个方法，我们来看看oom\_adj 是如何定义的：

可以得出结论，就是如果此时App正在前台显示运行，并且不是低内存状态，那么进程全局的GC就不会被触发！

2. 实际上触发GC的方法在 performAppGcLocked(proc) 这个方法里，下面scheduleAppGcsLocked() 这个方法是用来通知Activity, Service, Application onLowMemory() 回调的

###### scheduleAppGcsLocked();

##### ActivityMS .performAppGcLocked(proc);

*/\*\*  
 \* Ask a given process to GC right now.  
 \*/***final void** performAppGcLocked(ProcessRecord app) {  
 **try** {  
 app.lastRequestedGc = SystemClock.uptimeMillis();  
 **if** (app.thread != **null**) {  
 **if** (app.reportLowMemory) {  
 app.reportLowMemory = **false**;  
 app.thread.scheduleLowMemory();  
 } **else** {  
 app.thread.processInBackground();  
 }  
 }  
 } **catch** (Exception e) {  
 *// whatever.* }  
}

如果是低内存情况下执行scheduleLowMemory()方法，最终由 handleLowMemory() 实现GC回收；如果说不是的话，那么只有一种情况就是App是在后台运行的情况会进行GC回收，也就是没有前台展示界面的情况，这种情况走 processInBackground() 方法，最终由 doGcIfNeeded() 方法去实现回收。

app.thread.processInBackground();

**public void** processInBackground() {  
 mH.removeMessages(H.GC\_WHEN\_IDLE);  
 mH.sendMessage(mH.obtainMessage(H.GC\_WHEN\_IDLE));  
}

**case** GC\_WHEN\_IDLE:  
 scheduleGcIdler();

##### ActivityMS.scheduleAppGcsLocked

**final void** scheduleAppGcsLocked() {  
 mHandler.removeMessages(GC\_BACKGROUND\_PROCESSES\_MSG);  
  
 **if** (mProcessesToGc.size() > 0) {  
 *// Schedule a GC for the time to the next process.* ProcessRecord proc = mProcessesToGc.get(0);  
 Message msg = mHandler.obtainMessage(GC\_BACKGROUND\_PROCESSES\_MSG);  
  
 **long** when = proc.lastRequestedGc + mConstants.GC\_MIN\_INTERVAL;  
 **long** now = SystemClock.uptimeMillis();  
 **if** (when < (now+mConstants.GC\_TIMEOUT)) {  
 when = now + mConstants.GC\_TIMEOUT;  
 }  
 mHandler.sendMessageAtTime(msg, when);  
 }  
}

到这里我们大概已经知道这个Activity的触发时机了.

接下来，我们来分析第二条线，就是ActivityThread.java 里的 final void handleLowMemory() 方法，这个又是什么时候会调用呢？我们也从这条线往上走，我们发现一样也是走到了上面ActivityManagerService 里的 performAppGcsLocked()方法，把代码复制下来我们继续分析！

---------------------

作者：HaleYangD

来源：CSDN

原文：https://blog.csdn.net/cscs111/article/details/77558168

版权声明：本文为博主原创文章，转载请附上博文链接！

---------------------

作者：HaleYangD

来源：CSDN

原文：https://blog.csdn.net/cscs111/article/details/77558168

版权声明：本文为博主原创文章，转载请附上博文链接！

---------------------

作者：HaleYangD

来源：CSDN

原文：https://blog.csdn.net/cscs111/article/details/77558168

版权声明：本文为博主原创文章，转载请附上博文链接！

#### ActivityThread .handleLowMemory

同idea类似，消息LOW\_MEMORY处理，由AMS的performAppGcLocked

调用

**final void** handleLowMemory() {  
 ArrayList<ComponentCallbacks2> callbacks = collectComponentCallbacks(**true**, **null**);  
  
 **final int** N = callbacks.size();  
 **for** (**int** i=0; i<N; i++) {  
 callbacks.get(i).onLowMemory();  
 }  
  
 *// Ask SQLite to free up as much memory as it can, mostly from its page caches.* **if** (Process.myUid() != Process.SYSTEM\_UID) {  
 **int** sqliteReleased = SQLiteDatabase.releaseMemory();  
 EventLog.writeEvent(SQLITE\_MEM\_RELEASED\_EVENT\_LOG\_TAG, sqliteReleased);  
 }  
  
 *// Ask graphics to free up as much as possible (font/image caches)* Canvas.freeCaches();  
  
 *// Ask text layout engine to free also as much as possible* Canvas.freeTextLayoutCaches();  
  
 BinderInternal.forceGc(**"mem"**);  
}

**public void** scheduleLowMemory() {  
 sendMessage(H.LOW\_MEMORY, **null**);  
}

#### 小结

以上代码我们可以得出结论：Activity如果要被回收，那么要在虚拟机内存使用超过系统分配的最大内存的 3 / 4, 这时将会触发一次大规模GC 释放Activity（这个不管是app是在后台运行还是app低内存状态下运行都是如此），其余情况只有Activity 在onStop() 之后才会触发

接下来看看mgr.releaseSomeActivities(mAppThread);这个方法

mgr = ActivityManagerNative.getDefault(); 哦吼，你发现了， 这玩意儿其实就是ActivityManagerService, 那么接下来我们回到AMS中，看看，这个releaseSomeActivities()函数：

### AMS.releaseSomeActivities

究竟回收哪些呢？和linux的回收有什么区别呢？

@Override  
**public void** releaseSomeActivities(IApplicationThread appInt) {  
 ProcessRecord app = getRecordForAppLocked(appInt);  
 mStackSupervisor.releaseSomeActivitiesLocked(app, **"low-mem"**);  
}

#### StackSupervisor. releaseSomeActivitiesLocked

StackSupervisor 是什么？可以理解为activity任务栈的管理中心，系统所有应用的activity任务都在此管理：

**void** releaseSomeActivitiesLocked(ProcessRecord app, String reason) {  
 *// Examine all activities currently running in the process.* TaskRecord firstTask = **null**;  
 *// Tasks is non-null only if two or more tasks are found.* ArraySet<TaskRecord> tasks = **null**;  
 **if** (DEBUG\_RELEASE) Slog.d(TAG\_RELEASE, **"Trying to release some activities in "** + app);  
 **for** (**int** i = 0; i < app.activities.size(); i++) {  
 ActivityRecord r = app.activities.get(i);  
 *// First, if we find an activity that is in the process of being destroyed,  
 // then we just aren't going to do anything for now; we want things to settle  
 // down before we try to prune more activities.*

Destroy的不会回收**if** (r.finishing || r.state == DESTROYING || r.state == DESTROYED) {  
 **if** (DEBUG\_RELEASE) Slog.d(TAG\_RELEASE, **"Abort release; already destroying: "** + r);  
 **return**;  
 }  
 *// Don't consider any activies that are currently not in a state where they  
 // can be destroyed.* **if** (r.visible || !r.stopped || !r.haveState || r.state == RESUMED || r.state == PAUSING  
 || r.state == PAUSED || r.state == STOPPING) {  
 **if** (DEBUG\_RELEASE) Slog.d(TAG\_RELEASE, **"Not releasing in-use activity: "** + r);  
 **continue**;  
 }  
  
 **final** TaskRecord task = r.getTask();  
 **if** (task != **null**) {  
 **if** (DEBUG\_RELEASE) Slog.d(TAG\_RELEASE, **"Collecting release task "** + task  
 + **" from "** + r);  
 **if** (firstTask == **null**) {  
 firstTask = task;  
 } **else if** (firstTask != task) {  
 **if** (tasks == **null**) {  
 tasks = **new** ArraySet<>();  
 tasks.add(firstTask);  
 }  
 tasks.add(task);  
 }  
 }  
 }  
 **if** (tasks == **null**) {  
 **if** (DEBUG\_RELEASE) Slog.d(TAG\_RELEASE, **"Didn't find two or more tasks to release"**);  
 **return**;  
 }  
 *// If we have activities in multiple tasks that are in a position to be destroyed,  
 // let's iterate through the tasks and release the oldest one.* **final int** numDisplays = mActivityDisplays.size();  
 **for** (**int** displayNdx = 0; displayNdx < numDisplays; ++displayNdx) {  
 **final** ArrayList<ActivityStack> stacks = mActivityDisplays.valueAt(displayNdx).mStacks;  
 *// Step through all stacks starting from behind, to hit the oldest things first.* **for** (**int** stackNdx = 0; stackNdx < stacks.size(); stackNdx++) {  
 **final** ActivityStack stack = stacks.get(stackNdx);  
 *// Try to release activities in this stack; if we manage to, we are done.* **if** (stack.releaseSomeActivitiesLocked(app, tasks, reason) > 0) {  
 **return**;  
 }  
 }  
 }  
}

从以上的if (r.finishing || r.state == ActivityState.DESTROYING

|| r.state == ActivityState.DESTROYED) {

if (DEBUG\_RELEASE) Slog.d(TAG, "Abort release; already destroying: " + r);

return;

} 和 if (r.visible || !r.stopped || !r.haveState

|| r.state == ActivityState.RESUMED || r.state == ActivityState.PAUSING

|| r.state == ActivityState.PAUSED || r.state == ActivityState.STOPPING)这几个条件判断可以发现

1.只要当前进程有一个activity处在销毁阶段，直接退出，因为此时会触发其他activity的生命周期，没有必要回收

2.可以用户释放的，绝不考虑后台销毁，AMS只是处理用户可能没有机会处理的activity

3.收集tasks: 发现同一个进程，有2个任务沾才收集。

4.回收的单位是task，不是activity哦！？

只有在activity 执行到onStop，还没有被finish 或者destroy 的情况下。换言之，activity在后台&&没有退出。才会进行Activity的回收。

至此之后就会执行到Activity 的 performDestroy方法进行ondestroy，然后就等待GC回收的处理了。

#### releaseSomeActivitiesLocked

**final int** releaseSomeActivitiesLocked(ProcessRecord app, ArraySet<TaskRecord> tasks,  
 String reason) {  
 *// Iterate over tasks starting at the back (oldest) first.* **if** (DEBUG\_RELEASE) Slog.d(TAG\_RELEASE, **"Trying to release some activities in "** + app);  
 **int** maxTasks = tasks.size() / 4;  
 **if** (maxTasks < 1) {  
 maxTasks = 1;  
 }  
 **int** numReleased = 0;  
 **for** (**int** taskNdx = 0; taskNdx < mTaskHistory.size() && maxTasks > 0; taskNdx++) {  
 **final** TaskRecord task = mTaskHistory.get(taskNdx);  
 **if** (!tasks.contains(task)) {  
 **continue**;  
 }  
 **if** (DEBUG\_RELEASE) Slog.d(TAG\_RELEASE, **"Looking for activities to release in "** + task);  
 **int** curNum = 0;  
 **final** ArrayList<ActivityRecord> activities = task.mActivities;  
 **for** (**int** actNdx = 0; actNdx < activities.size(); actNdx++) {  
 **final** ActivityRecord activity = activities.get(actNdx);  
 **if** (activity.app == app && activity.isDestroyable()) {  
 **if** (DEBUG\_RELEASE) Slog.v(TAG\_RELEASE, **"Destroying "** + activity  
 + **" in state "** + activity.state + **" resumed="** + mResumedActivity  
 + **" pausing="** + mPausingActivity + **" for reason "** + reason);  
 destroyActivityLocked(activity, **true**, reason);  
 **if** (activities.get(actNdx) != activity) {  
 *// Was removed from list, back up so we don't miss the next one.* actNdx--;  
 }  
 curNum++;  
 }  
 }  
 **if** (curNum > 0) {  
 numReleased += curNum;  
 maxTasks--;  
 **if** (mTaskHistory.get(taskNdx) != task) {  
 *// The entire task got removed, back up so we don't miss the next one.* taskNdx--;  
 }  
 }  
 }  
 **if** (DEBUG\_RELEASE) Slog.d(TAG\_RELEASE,  
 **"Done releasing: did "** + numReleased + **" activities"**);  
 **return** numReleased;  
}

### 参考

Android回收部分activity的流程分析

<https://blog.csdn.net/cscs111/article/details/77558168>

# getTasks

获取栈顶app

**public boolean** isBlackApkRunning() {  
 List<ActivityManager.RunningTaskInfo> runningTasks;  
 **try** {  
 runningTasks = ActivityManagerNative.*getDefault*().getTasks(1, 0);  
 **if** (runningTasks == **null** || runningTasks.size() == 0) {  
 **return false**;  
 }  
 } **catch** (RemoteException e) {  
 LOG(**"Failed to detect top activity"**+e);  
 **return false**;  
 }  
 ComponentName topActivity = runningTasks.get(0).**topActivity**;  
 LOG(**"topActivity="**+topActivity);  
 **for** (String black\_list: ***BLACK\_LIST\_OF\_HEADS\_UP***) {  
 **if** (topActivity.getPackageName().equals(black\_list)) {  
 LOG(**"in list"**);  
 **return true**;  
 }  
 }  
 **return false**;  
}

# 隐式调用

有利于降低发送者和接收者之间的耦合，它一般用在没有明确指出目标组件名称的前提下

### Intent Filter match过程

#### 使用规范

隐式 Intent

Intent intent = new Intent();

intent.setAction("com.wooyun.test"); startActivity(intent);

* a. action匹配规则：要求intent中的action 存在 且 必须和过滤规则中的其中一个相同 区分大小写；
* b. category匹配规则：系统会默认加上一个android.intent.category.DEAFAULT，所以intent中可以不存在category，但如果存在就必须匹配其中一个；
* c. data匹配规则：data由两部分组成，mimeType和URI，要求和action相似。如果没有指定URI，URI但默认值为content和file（schema）

### 源码分析

https://www.yuanmas.com/info/9ezZpgB0y6.html

主线流程：先match action, 再match data, 最后match category



# BOOT

Sdf

ActivityManager: Sending BOOT\_COMPLETE user #0

06-13 03:22:38.163 549-1189/? E/BootReceiver: Error reading /data/system/last-header.txt

java.io.FileNotFoundException: /data/system/last-header.txt (No such file or directory)

at java.io.FileInputStream.open(Native Method)

at java.io.FileInputStream.<init>(FileInputStream.java:146)

at android.os.FileUtils.readTextFile(FileUtils.java:263)

at com.android.server.BootReceiver.getPreviousBootHeaders(BootReceiver.java:121)

at com.android.server.BootReceiver.getBootHeadersToLogAndUpdate(BootReceiver.java:143)

at com.android.server.BootReceiver.logBootEvents(BootReceiver.java:163)

06-13 11:23:27.954 1528-1567/system\_process D/ActivityManager: Finishing user boot 0

06-13 11:23:28.281 1528-1550/system\_process D/ActivityManager: Sending BOOT\_COMPLETE user #0

06-13 11:23:34.238 2448-2448/com.dji D/DJIMessageReceiver: DJIMessageReceiver android.intent.action.BOOT\_COMPLETED

06-13 11:23:34.239 2448-2448/com.dji D/DJIMessageReceiver: recv ACTION\_BOOT\_COMPLETED.

06-13 11:23:34.248 1528-2135/system\_process W/BroadcastQueue: Unable to launch app cn.goapk.market/10047 for broadcast Intent { act=android.intent.action.BOOT\_COMPLETED flg=0x9000010 (has extras) }: process is bad

06-13 11:23:35.311 1528-2478/system\_process I/BootReceiver: Copying audit failures to DropBox

Checking for fsck errors

06-13 11:23:35.312 1528-2478/system\_process E/BootReceiver: Can't log boot events

java.io.FileNotFoundException: /dev/fscklogs/log (Permission denied)

at java.io.FileInputStream.open(Native Method)

at java.io.FileInputStream.<init>(FileInputStream.java:146)

at android.os.FileUtils.readTextFile(FileUtils.java:263)

at com.android.server.BootReceiver.addFsckErrorsToDropBox(BootReceiver.java:311)

at com.android.server.BootReceiver.logBootEvents(BootReceiver.java:203)

at com.android.server.BootReceiver.-wrap2(BootReceiver.java)

at com.android.server.BootReceiver$1.run(BootReceiver.java:93)

06-13 11:23:35.322 1753-1753/com.android.settings D/HdmiReceiver: hdmi receiver action=android.intent.action.BOOT\_COMPLETED

06-13 11:23:35.324 1753-1753/com.android.settings D/HdmiReceiver: BOOT\_COMPLETED

06-13 11:23:35.375 1528-1936/system\_process I/ActivityManager: Start proc 2502:com.facebook.katana/u0a32 for broadcast com.facebook.katana/com.facebook.push.mqtt.receiver.BootCompleteBroadcastReceiver

**06-13 11:23:35.814 1528-1936/system\_process I/ActivityManager: Start proc 2522:com.dpad.ui/1000 for broadcast com.dpad.ui/com.dji.systemui.BootBroadcastReceiver**

**06-13 11:23:38.398 1528-1936/system\_process I/ActivityManager: Start proc 2758:com.twitter.android/u0a44 for broadcast com.twitter.android/com.evernote.android.job.JobBootReceiver**

**No Debuggable Processes**

06-13 04:43:24.375 588-625/? I/ActivityManager: Start proc 1062:com.dpad.update/1000 for broadcast com.dpad.update/.controller.DJIImageUpReceiver

06-13 04:43:26.520 1062-1062/? W/ContextImpl: Calling a method in the system process without a qualified user: android.app.ContextImpl.startService:1358 android.content.ContextWrapper.startService:613 android.content.ContextWrapper.startService:613 com.dpad.update.controller.DJIImageUpReceiver.onReceive:31 android.app.ActivityThread.handleReceiver:3095

06-13 12:43:50.224 1522-2128/system\_process I/ActivityManager: Start proc 2389:com.dpad.update/1000 for broadcast com.dpad.update/.controller.DJIImageUpReceiver

# 总结

## 组件超时统计

### Service

| **序号** | **App端方法** | **生命周期** | **计时起点** | **计时终点** |
| --- | --- | --- | --- | --- |
| 1 | AT.handleCreateService | onCreate | AS.realStartServiceLocked | serviceDoneExecuting |
| 2 | AT.handleServiceArgs | onStartCommand | AS.sendServiceArgsLocked | serviceDoneExecuting |
| 3 | AT.handleBindService | onBind/onRebind | AS.requestServiceBindingLocked | serviceDoneExecuting |
| 4 | AT.handleUnbindService | onUnbind | AS.removeConnectionLocked | serviceDoneExecuting |
| 5 | AT.handleStopService | onDestroy | AS.bringDownServiceLocked | serviceDoneExecuting |

说明:

* 其中AS是指ActiveServices；
* 方法1,2,5组成startService/stopService方式的生命周期;
* 方法1,3,4,5组成bindService/unbindService方式的生命周期;
* 每一个生命周期回调方法ANR情况
  + 计时方式: 起点是对端方法, 终点是serviceDoneExecuting()方法
  + 前台进程启动的service不允许超过20s(ActiveServices.SERVICE\_TIMEOUT)
  + 后台进程启动的service不允许超过200s
  + 前后台判断标准callerFg = callerApp.setSchedGroup != Process.THREAD\_GROUP\_BG\_NONINTERACTIVE;
* 必须等到QueuedWork执行完成才结束的生命周期：
  + handleServiceArgs
  + handleServiceArgs

另外, AS.bringDownServiceLocked过程也会触发handleUnbindService

### Broadcast

| **序号** | **App端方法** | **生命周期** | **System端方法** | **计数终点** |
| --- | --- | --- | --- | --- |
| 1 | handleReceiver | onReceive | BQ.processCurBroadcastLocked | sendFinished |
| 2 | ReceiverDispatcher.Args.run | onReceive | BQ.performReceiveLocked | sendFinished |

说明:

* 其中BQ是指BroadcastQueue，ReceiverDispatcher是LoadedApk的静态内部类；
* 静态注册的广播接收者:
  + 生命周期回调为handleReceiver；
  + 不论何种广播都会调用sendFinished();
* 动态注册的广播接收者:
  + 周末周期回调为ReceiverDispatcher.Args.run；
  + 发送的是串行广播, 则会调用sendFinished();
  + 发送的是并行广播, 则无需调用sendFinished();
* 广播ANR的情况:
  + 计时方式: 在广播没有处理完之前, 采用周期为mTimeoutPeriod的轮询方式
  + 静态注册的广播, 以及发送的本身就是串行广播, 都会采用串行方式处理.
  + 串行方式ANR情况1：某个广播总处理时间 > 2\* receiver总个数 \* mTimeoutPeriod;
    - 前台队列mTimeoutPeriod默认为10s(AMS.BROADCAST\_FG\_TIMEOUT)，
    - 后台队列mTimeoutPeriod默认为60s;
    - 前后台判定isFg = (intent.getFlags() & Intent.FLAG\_RECEIVER\_FOREGROUND) != 0;
  + 串行方式ANR情况2：某个receiver的执行时间超过mTimeoutPeriod；
* 必须等到QueuedWork执行完成才结束的生命周期：
  + handleReceiver

### ContentProvider

| **序号** | **App端方法** | **生命周期** | **计数起点** | **计数终点** |
| --- | --- | --- | --- | --- |
| 1 | installProvider | onCreate | AMS.attachApplicationLocked | AMS.publishContentProviders |

说明：

* Provider发布过程，从计数起点到终点，当超过10s没有执行完成，则会弹出ANR;
* 其中AMS.CONTENT\_PROVIDER\_PUBLISH\_TIMEOUT=10s;

### 1.4 Activity

| **序号** | **App端方法** | **生命周期** | **计时起点** | **计时终点** |  |
| --- | --- | --- | --- | --- | --- |
| 1 | handleLaunchActivity | onCreate/onStart/onResume |  |  |  |
| 2 | handleResumeActivity | onResume |  |  |  |
| 3 | handlePauseActivity | onPause | startPausingLocked | activityPausedLocked |  |
| 4 | handleStopActivity | onStop | stopActivityLocked | activityStoppedLocked |  |
| 5 | handleDestroyActivity | onDestroy | destroyActivityLocked | activityDestroyedLocked |  |
| 6 | handleRelaunchActivity |  |  |  |  |
| 7 | handleNewIntent | onNewIntent |  |  |  |
| 8 | handleSleeping |  |  |  |  |
| 9 | handleSendResult | onActivityResult |  |  |  |

说明:

* onPause
  + 当超时500ms没有执行完成handlePauseActivity(), 则直接进入AS.activityPausedLocked();
* ActivityRecord.setSleeping
  + 该过程会触发handleSleeping.
* 必须等到QueuedWork执行完成才结束的生命周期：
  + handleStopActivity
  + handleSleeping

#### 1.4.1 Activity超时常量

| **事件** | **Timeout** | **文件** |
| --- | --- | --- |
| LAUNCH\_TICK | 0.5s | ActivityStack |
| PAUSE\_TIMEOUT | 0.5s | ActivityStack |
| STOP\_TIMEOUT | 10s | ActivityStack |
| DESTROY\_TIMEOUT | 10s | ActivityStack |
| APP\_SWITCH\_DELAY\_TIME | 5s | AMS |
| SLEEP\_TIMEOUT | 5s | ASS |
| IDLE\_TIMEOUT | 10s | ASS |
| LAUNCH\_TIMEOUT | 10s | ASS |

注：ASS是指ActivityStackSupervisor.

## Handler角度

H

### 四大组件相关Handler

| **Handler** | **数据类型** | **运行线程** |
| --- | --- | --- |
| AMS.mUiHandler | UiHandler | android.ui |
| AMS.mBgHandler | Handler | android.bg |
| AMS.mHandler | MainHandler | ActivityManager |
| ASS.mHandler | ActivityStackSupervisorHandler | ActivityManager |
| AS.mHandler | ActivityStackHandler | ActivityManager |
| BroadcastQueue.mHandler | BroadcastHandler | ActivityManager |
| ActiveServices.mServiceMap | ServiceMap | ActivityManager |

说明：

* AMS.MainHandler
  + 处理service、process、provider的超时问题；
* BroadcastHandler：
  + 处理broadcast的超时问题；
* ActivityStackSupervisorHandler：
  + 处理IDLE\_TIMEOUT，SLEEP\_TIMEOUT，LAUNCH\_TIMEOUT
* ActivityStackHandler：
  + 处理PAUSE\_TIMEOUT，STOP\_TIMEOUT，DESTROY\_TIMEOUT
  + 处理TRANSLUCENT\_TIMEOUT，LAUNCH\_TICK
* ActiveServices.ServiceMap：
  + 处理BG\_START\_TIMEOUT

以上所有跟**超时相关的工作都运行在ActivityManager**线程，唯独input的超时处理过程并非发生在ActivityManager线程，而是inputDispatcher线程发生的。

### 2.2 UI相关Handler

对于ANR/Crash/Error等几乎所有错误、警告相关的对话框都运行在android.ui线程，例如：

* BaseErrorDialog.mHandler
* AppErrorDialog
* StrictModeViolationDialog
* AppNotRespondingDialog
* AppWaitingForDebuggerDialog
* UserSwitchingDialog