Android 的Activity生命周期

首先说几点常识哈：

1. 全程都需要用Trace 类来追踪执行轨迹

ActivityThread

#### main( Int argc , String[] argv )

这里是所有Activity运行的起始位置，也就相当于安卓系统开始于孵化器 zygote一样。

主要的几件事：：

//用户配置加载

Environment.initForCurrentUser();

//为Looper做准备

Looper.*prepareMainLooper*();

// 创建AndroidThread【后面简称AT】 对象

ActivityThread thread = **new** ActivityThread();

thread.attach(**false**);

// 开始循环执行

Looper.*loop*();

下面这条语句在正常情况下是不会执行的，实际上也很少见到。

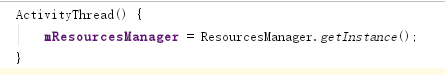
所以，Activity执行过程中应该是完全卡在Looper.loop()中的.

下文见Looper

**throw new** RuntimeException(**"Main thread loop unexpectedly exited"**);

#### ActivityThread构造函数

事实上构造函数里面啥也没做，只是获取了一个ResourcesManager



#### handle<??>()函数

处理、转发相关操作。

#### perform<??>()函数

执行相关操作。这里相比handle类函数更强硬一些【开始即运行】

#### performLaunchActivity()函数

这个函数是Activity的入口函数，我们重点分析一下。

一些信息的获取，用于startIntent的合成

ActivityInfo aInfo = r.**activityInfo**;

ComponentName component = r.**intent**.getComponent();

// **Context的构造**

ContextImpl appContext = createBaseContextForActivity(r);

// **使用反射方法 构造Activity （简单粗暴，我们也可以学习借鉴::【热修复、插件化】）**

**try** {  
 java.lang.ClassLoader cl = appContext.getClassLoader();  
 activity = **mInstrumentation**.newActivity(  
 cl, component.getClassName(), r.**intent**);  
 … … …  
}

// **后面是application的构造及Activity的启动**

**try** {  
 Application app = r.**packageInfo**.makeApplication(**false**, **mInstrumentation**);  
// 系统日志写法可以借鉴一下  
 **if** (***localLOGV***) Slog.v(  
 ***TAG***, r + **": app="** + app  
 + **", appName="** + app.getPackageName()  
 + **", pkg="** + r.**packageInfo**.getPackageName()  
 + **", comp="** + r.**intent**.getComponent().toShortString()  
 + **", dir="** + r.**packageInfo**.getAppDir());  
  
 **if** (activity != **null**) {  
 CharSequence title = r.**activityInfo**.loadLabel(appContext.getPackageManager());  
 Configuration config = **new** Configuration(**mCompatConfiguration**);  
 Window window = r.**mPendingRemoveWindow**;  
 appContext.setOuterContext(activity);

// **关键部分，初始化Activity了**  
 **activity.attach(appContext, this, getInstrumentation(), r.token,  
 r.ident, app, r.intent, r.activityInfo, title, r.parent,  
 r.embeddedID, r.lastNonConfigurationInstances, config,  
 r.referrer, r.voiceInteractor, window, r.configCallback);**  
  
 **if** (customIntent != **null**) {  
 activity.mIntent = customIntent;  
 }

// 系统的禁止联网策略执行  
 checkAndBlockForNetworkAccess();

// 应用**主题**了  
 **int** theme = r.**activityInfo**.getThemeResource();  
 **if** (theme != 0) {  
 activity.setTheme(theme);  
 }

// 这里隐含了 **onCreate()** 方法调用  
 **if** (r.isPersistable()) {  
 **mInstrumentation**.**callActivityOnCreate**(activity, r.**state**, r.**persistentState**);  
 } **else** {  
 **mInstrumentation**.callActivityOnCreate(activity, r.**state**);  
 }

r.**activity** = activity;  
 r.**stopped** = **true**;

// 注意到这里的三次次相同条件判断，分别用于几个回调函数  
 **if** (*!r.****activity****.mFinished*) {  
 activity.**performStart**();  
 }  
 **if** (*!r.****activity****.mFinished*) {  
 **mInstrumentation**.**callActivityOnRestoreInstanceState**(activity, r.**state**);  
 }  
 **if** (*!r.****activity****.mFinished*) {  
 **mInstrumentation**.**callActivityOnPostCreate**(activity, r.**state**,);  
 }  
 }  
  
}

Looper

Loop本意循环，顾名思义，Loop是个长时间工作， 可能包括时延，挂起，或者循环；；Looper应该是安卓系统里面比较核心的一个组件，从Looper这个类名在程序崩溃记录里面的存在感也不低就能看出来。

* Looper开始执行前必须调用一次 prepare()

#### prepare()

实际上调用了prepare(**true**);

创建了消息队列，也是**关键点**之一。

mQueue = **new** MessageQueue(quitAllowed);

这里后接此函数内容了

只有下面一句，

sThreadLocal.set(**new** Looper(quitAllowed));

这里我们去看看ThreadLocal

#### Loop()

主循环函数，地位相当于Thread.run() ，就不用多解释了吧e \_ e

前面两句不知道做了些什么

Binder.*clearCallingIdentity*();  
**final long** ident = Binder.*clearCallingIdentity*();

然后是一个for语句死循环，不知道为什么不用while()

**for** (;;) {

关键语句，全程都在接收消息，如果没有消息的话就直接退出函数，说明**msg很关键**。

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

这里 **对Message进行dispatch()**，另外值得一提的是 这个地方使用的是Handler的方法，因为所有Message的target永远是一个 handler.

**try** {

**// targer::android.os.Handler**  
 msg.target.dispatchMessage(msg);   
}

}

Message

#### dispatchMessage(Message msg)

分发消息方法，由系统来分配调用。

优先调用发送方的callback，其次是接收方的callback，最后采用默认机制处理。

**public void** dispatchMessage(Message msg) {  
 **if** (msg.callback != **null**) {  
 *handleCallback*(msg);  
 } **else** {  
 **if** (**mCallback** != **null**) {  
 **if** (**mCallback**.handleMessage(msg)) {  
 **return**;  
 }  
 }  
 handleMessage(msg);  
 }  
}

android.app.ActivityThread$H 安卓自定义Handler

这里可以看到一系列的独立响应方式定义，也就是安卓Activity相关状态变化及操作的处理核心（AT中也有很多handle开头的函数，都是用来响应这个系统msg的）

在这里可以看到许多Activity方法调用的痕迹。

* 启动Activity

**case *LAUNCH\_ACTIVITY***: {  
 Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, **"activityStart"**);  
 **final** ActivityClientRecord r = (ActivityClientRecord) msg.**obj**;  
  
 r.**packageInfo** = getPackageInfoNoCheck(  
 r.**activityInfo**.**applicationInfo**, r.**compatInfo**);  
 handleLaunchActivity(r, **null**, **"LAUNCH\_ACTIVITY"**);  
 Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);  
} **break**;

* 内存不足

**case *LOW\_MEMORY***:  
 Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, **"lowMemory"**);  
 handleLowMemory();  
 Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);  
 **break**;